РЫСОВАНЫЙ
Александр Николаевич –

Родился 22.09.1957.
Закончил с отличием Харьковское высшее военное авиационное командное училище связи в 1977 г., Военно-воздушную инженерную орденов Ленина и Октябрьской революции Краснознаменную академию им. проф. Н.Е. Жуковского в 1984 г., г. Москва.
Специальность "Автоматизированные системы управления", квалификация “военный инженер системотехник”.

Служба в Вооруженных Силах – 31 год: ХВВАКУС; в/ч пп 81524, Wittstock/Dosse; ВВИА; ХВВАУРЭ; ХВУ; ХУВС.
Воинское звание – полковник.

Работает в НТУ “ХПИ” с 2006 г.
Проф. кафедры “Вычислительная техника и программирование”.
Кандидат технических наук (год защиты – 1995), доцент.

Имеет более 250 научных публикаций, из них 6 учебников, 38 учебных пособий, больше 100 изобретений, оформленных в виде авторских свидетельств, патентов, патентов на полезные модели и др.

Руководитель магистров и аспирантов.

Область научных исследований: развитие теории недвоичного сигнатурного анализа для повышения эффективности диагностирования цифровых систем.
e-mail: rysov@rambler.ru

Рассмотрен широкий круг вопросов, начиная с основ программирования на ассемблере с использованием базовой системы команд и заканчивая использованием современных технологий обработки данных, таких, как MMX, SSE –SSE4, AVX. Кроме того, рассмотрены вопросы оконного программирования и организации управления аппаратными устройствами. Теоретический материал подкреплен большим количеством примеров программного кода – от самых коротких программ до программ средней сложности, выполненных в **макроассемблере masm32.** Программы с использованием команд SSE – SSE4, AVX выполнены в пакете **Visual Studio 2013.** Все программы протестированы с помощью отладчика OllyDbg.

Предназначено для студентов специальностей 6.050102-01 «Компьютерные системы и сети», 6.050102-02 «Системное программирование» и 6.050102-03 «Специализованные компьютерные системы». Также может быть полезным для специалистов.

О.М. Рисований

Системне програмування

О.М. Рисований

СИСТЕМНЕ ПРОГРАМУВАННЯ

Підручник
Том 1

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

О.М. Рисований

СИСТЕМНЕ ПРОГРАМУВАННЯ

Том 1

Підручник

для студентів напрямку “Компютерна інженерія”
вищих навчальних закладів

Видання четверте: виправлено та доповнено

Затверджено Міністерством освіти і науки України

Харків 2015

http://blogs.kpi.kharkov.ua/v2/asm/knigi/

ББК 32.973-018.14
Р54
УДК 004.42

Рецензенти:

І.О. Фурман, д-р техн. наук, проф., академік АН Вищої школи України, Харківський національний технічний університет сільського господарства;

Г.Ф. Кривуля, д-р техн. наук, проф., Харківський державний технічний університет радіоелектроніки;

Г.І. Загарій, д-р техн. наук, проф., Українська державна академія залізничного транспорту.

Гриф надано Міністерством освіти і науки України,
лист № 1.4/18-Г-115 від 10.01.09

Р54 **Рисований О.М.** Системне програмування [Текст]: підручник для студентів напрямку "Компютерна інженерія" вищих навчальних закладів в 2-х томах. Том 1. – Видання четверте: виправлено та доповнено – Х.: "Слово", 2015. – 576 с.

Розглянуто широке коло питань, починаючи з основ програмування на асемблері з використанням базової системи команд та закінчуючи використанням сучасних технологій обробки даних, таких, як MMX, SSE – SSE4, AVX. Крім того, розглянуто питання віконного програмування та організації управління апаратними пристроями. Теоретичний матеріал підкріплено великою кількістю прикладів програмного коду – від самих коротких програм до програм середньої складності, виконаних в макроасемблері masm32. Програми з використанням команд SSE2 – SSE4 виконані в пакеті Visual Studio. Всі програми протестовані за допомогою налагоджувача OllyDbg.

Призначено для студентів спеціальностей 6.050102-01 «Комп'ютерні системи та мережі», 6.050102-02 «Системне програмування» і 6.050102-03 «Спеціалізовані комп'ютерні системи». Також може бути корисним для спеціалістів.

ББК 32.973-018.1

© О.М. Рисований, 2015

ЗМІСТ

ВСТУП **9**
1. ЕТАПИ СТВОРЕННЯ ПРОГРАМИ МОВОЮ АСЕМБЛЕРА **13**
1.1. Підготовка вихідного тексту програми в Masm32 13
1.2. Отримання об'єктного коду в Masm32 14
1.3. Отримання виконуваного файлу програми в Masm32 16
1.4. Зменшення розміру програми за рахунок ключів компіляції та лінкування в Masm32 18
1.5. Налагодження програми 20
1.6. Редактори тексту 20
1.7. Настройка середовища Visual Studio 2013 21
1.8. Угорська нотація 25
2. АРХІТЕКТУРА МІКРОПРОЦЕСОРА INTEL **27**
2.1. Архітектурні особливості 32-розрядного МП Intel 27
2.1.1. Типи даних МП Intel 27
2.1.2. Регістри 32-розрядного МП Intel 29
2.1.3. Моделі пам'яті МП Intel 35
2.1.4. Способи адресації 32-розрядного МП Intel 38
2.2. Архітектурні особливості 64-розрядного МП Intel 49
2.2.1. Угода про виклики параметрів функцій 49
2.2.2. Регістри архітектури x64 49
2.2.3. Використання регістрів архітектури x64 51
2.2.4. Скалярні типи даних архітектури x64 53
2.2.5. Використання стека архітектури x64 53
3. ОСНОВНІ ДИРЕКТИВИ ТА ОПЕРАНДИ АСЕМБЛЕРА **56**
3.1. Директиви асемблера 56
3.1.1. Директиви MODEL, DATA, CONST, STACK, CODE 56
3.1.2. Директиви SEGMENT та ENDS 58
3.1.3. Директиви PROC та ENDP 60
3.1.4. Директива ASSUME 60
3.1.5. Директива ORG 61
3.1.6. Директиви визначення даних DB, DW, DD, DF, DP, DQ, DT 61
3.1.7. Директиви символьних констант: = , EQU, TEXTEQU 63
3.1.8. Директива END 64
3.1.9. Директива LABEL 64
3.1.10. Директива ALIGN 64
3.1.11. Директива LOCALS 65
3.1.12. Директиви керування файлами INCLUDE, INCLUDELIB 65
3.2. Оператори мови асемблера 66
3.3. Лабораторна робота «Подання даних» 69
4. ОПЕРАЦІЇ ПЕРЕСИЛАННЯ ДАНИХ 80
4.1. Команди пересилання даних загального призначення 80
4.2. Команди роботи зі стеком 83

4.3. Команди роботи з адресами і вказівником 88
4.4. Команди перетворення даних 89
4.5. Команди введення та виведення в порт 89
4.6. Команди пересилання бітів умов 90
5. ОСНОВНІ АРИФМЕТИЧНІ ОПЕРАЦІЇ 92
5.1. Арифметичні операції **92**
5.2. Лабораторна робота «Програмування арифметичних операцій» 112
5.3. Додатковий матеріал про налагоджувач OLLYDBG 118
6. ЛОГІЧНІ КОМАНДИ ТА КОМАНДИ ЗСУВУ 123
6.1. Команди булевих операцій 123
6.2. Команди перевірки і модифікації бітів 124
6.3. Команди сканування бітів 126
6.4. Команди зсуву (зрушення) і циклічного зсуву 127
6.4.1. Команди зсуву 127
6.4.2. Команди циклічного зсуву 129
6.4.3. Команди подвійного зсуву 130
6.5. Команди установки байта за умовою 130
6.6. Команда перевірки 131
7. КОМАНДИ ПЕРЕДАННЯ КЕРУВАННЯ 133
7.1. Команда безумовного переходу JMP 133
7.2. Команди умовної передання керування Jcc 134
7.3. Команди керування циклами LOOPx 143
8. ПРОЦЕДУРИ 148
8.1. Виклик (команда CALL) процедури
та її повернення (команда RET) 148
8.2. Угоди про виклик функцій 149
8.3. Команди керування стеком при виконанні процедур 150
8.4. Директива LOCAL 152
8.5. Директива INVOKE 153
8.6. Оператор ADDR 153
8.7. Директива PROTO 154
8.8. Оператор USES 154
8.9. Організація процедур та їх дослідження 155
8.10. Windows API-подібні процедури 160
8.11. Непрямий виклик процедур 162
8.12. Використання загальних змінних у процедурах 163
8.13. Лабораторна робота «Передача параметрів через таблицю адрес» ... 176
8.14. Лабораторна робота «API-подібні процедури» 180
8.15. Лабораторна робота «Зовнішні процедури» 182
9. ОРГАНІЗАЦІЯ ВВЕДЕННЯ – ВИВЕДЕННЯ В WIN32 187
9.1. Загальні відомості 187
9.2. Набір символів і API-функції Windows 189
9.3. Типи даних Windows 189
9.4. Дескриптори консолі 190

9.5. API-функції консолі в Win32 191
9.6. Виведення повідомлень 193
9.7. Виведення чисел 200
9.8. Введення з консолі 209
9.9. Керування кольором 212
9.10. Лабораторна робота «Введення-виведення даних» 215
10. ДИНАМІЧНІ БІБЛІОТЕКИ 227
10.1. Створення динамічних бібліотек та їх використання 228
10.2. Функція точки входу в DLL 231
10.3. Лабораторна робота "Dll-файли" 235
11. ФАЙЛИ 239
11.1. Створення файлу 241
11.2. Читання файлу 244
11.3. Переміщення файлового вказівника 247
11.4. Відмітки часу для файлу 254
11.5. Отримання інформації про файл та його ідентифікатор 255
11.6. Створення, перейменування і видалення каталогів 256
11.7. Файлові операції засобами ShellAPI 259
11.8. Лабораторна робота «Файли» 263
12. АРИФМЕТИЧНИЙ СПІВПРОЦЕСОР 267
12.1. Типи даних співпроцесора 267
12.2. Архітектура співпроцесора 269
12.3. Система команд математичного співпроцесора 271
12.3.1. Команди передання даних 273
12.3.2. Команди порівняння 280
12.3.3. Арифметичні команди 287
12.3.4. Трансцендентні команди 305
12.3.5. Команди маніпуляції константами 311
12.3.6. Команди керування 311
12.4. Лабораторна робота «Обробка даних у співпроцесорі» 313
13. РЯДКИ 328
13.1. Обробка рядків 328
13.2. Команда MOVS(B/W/D/Q) 330
13.3. Команда CMPS(B/W/D/Q) 334
13.4. Команда SCAS (B/W/D/Q) 337
13.5. Команда LODS (B/W/D/Q) 343
13.6. Команда STOS (B/W/D/Q) 344
13.7. Лабораторна робота «Рядки» 346
14. МАКРОВИЗНАЧЕННЯ 353
14.1. Введення в макровизначення 353
14.1.1. Порівняльний аналіз процедур і макрозасобів 353
14.1.2. Місця використання макровизначень 354
14.2. Директиви MACRO і ENDM 355
14.3. Директива LOCAL 359

14.4. Булеві вирази ... 359
14.5. Оператори в макровизначеннях ... 363
14.5.1. Оператор заміни (&) ... 363
14.5.2. Оператор виразу (%) ... 365
14.5.3. Оператор виділення тексту (<>) ... 365
14.5.4. Оператор виділення символу (!) ... 366
14.5.5. Оператор макрокоментаря (;;) ... 366
14.6. Додаткові макровизначення і директиви ... 366
14.6.1. Директива REPT ... 366
14.6.2. Директива IRP ... 367
14.6.3. Директива IRPC ... 368
14.7. Макроси циклів: FOR, FORC, REPEAT, WHILE ... 369
14.8. Категорії макросів в masm32 ... 371
14.9. Макрос @ для запису команд в один рядок ... 375
14.10. Лабораторна робота "Макроси" ... 377
15. ЛОГІЧНІ КОНСТРУКЦІЇ ВИСОКОГО РІВНЯ ... 382
15.1. Директива IF ... 382
15.2. Логічна директива switch ... 390
15.3. Логічна директива switch$ ... 391
15.4. Логічні директиви .REPEAT та .WHILE ... 391
15.5. Директива .BREAK ... 398
15.6. Директива .CONTINUE ... 398
15.7. Лабораторна робота "Директиви умовного асемблювання" ... 399
15.8. Лабораторна робота "Двовимірні масиви" ... 407
16. СТРУКТУРИ ... 413
16.1. Організація структур ... 413
16.2. Структура SYSTEMTIME для визначення системного часу ... 416
16.3. Структура MSGBOXPARAMS в функції MessageBoxIndirect ... 418
16.4. Складні структури ... 422
16.5. Створення зображення іконки ... 423
16.6. Лабораторна робота "Структури" ... 423
17. АНТИНАЛАГОДЖУВАЛЬНІ ПРИЙОМИ ... 430
17.1. Трасування в x86-процесорах ... 430
17.2. Переривання в масці ... 431
17.3. Трасування за часом виконання інструкцій ... 432
17.4. Обробка виключень ... 433
18. MMX-РОЗШИРЕННЯ ... 437
18.1. MMX-команди передачі даних ... 440
18.2. Арифметичні MMX-команди ... 440
18.2.1. Команди додавання ... 440
18.2.2. Команди віднімання ... 445
18.2.3. Команди множення ... 449
18.3. MMX-команди пакування та розпакування даних ... 452
18.4. MMX-команди порівняння ... 460

18.5. Логічні MMX-команди 464
18.6. Команди зсуву 464
18.7. MMX-команди в Pentium III 467
18.8. MMX-команди в Pentium IV 471
18.9. Лабораторна робота “MMX-команди” 472
19. SSE-РОЗШИРЕННЯ 480
19.1. Команди пересилання даних 484
19.2. Арифметичні команди 488
19.2.1. Команди додавання 488
19.2.2. Команди віднімання 490
19.2.3. Команди паралельного та скалярного множення 491
19.2.4. Команди паралельного та скалярного ділення 492
19.2.5. Додаткові арифметичні команди 494
19.3. Команди порівняння 500
19.4. Команди перетворення 506
19.5. Логічні команди 508
19.6. Команди управління станом 510
19.7. Команди розпаковування даних 511
19.8. Команди управління кешуванням 513
19.9. Асемблювання та компонування програми через безпосереднє звертання до Visual Studio 2008 514
19.10. Лабораторна робота “SSE-команди” 516
20. SSE2-РОЗШИРЕННЯ 524
20.1. Команди передавання даних 526
20.1.1. Обмін 128-bit кодами між оперативною пам’яттю (ОЗП) та регістрами xmm 526
20.1.2. Обмін 64-розрядними кодами між ОЗП і регістрами xmm 527
20.1.3. Використання масок 528
20.2. Арифметичні операції 531
20.2.1. Паралельні операції з упакованими числами 531
20.2.2. Арифметичні операції зі скалярними числами 534
20.3. Команди порівняння 536
20.3.1. Порівняння без зміни стану розрядів EFLAGS 536
20.3.2. Команди порівняння, які змінюють регістр EFLAGS 537
20.4. Логічні операції 538
20.5. Команди розпаковування і розподілення даних 538
20.6. Команди перетворення форматів даних 540
20.6.1. Групові перетворення форматів даних 540
20.6.2. Перетворення формату одного числа 543
20.7. Управління кешем 544
20.8. Операції з цілими числами. Нові можливості інструкцій MMX 545
20.9. Асемблювання через безпосередній запис до каталогу Visual Studio 2008 та компонування через masm32 545
20.10. Асемблювання та компонування через безпосереднє

звертання до Visual Studio 2010 546
20.11. Лабораторна робота “SSE2-команди” 548
21. SSE3-РОЗШИРЕННЯ 556
21.1. Перетворення чисел з плаваючою точкою (x87) в цілі числа 556
21.2. Дублювання даних 556
21.3. Завантаження невирівняних змінних 557
21.4. Одночасне додавання/віднімання 558
21.5. Горизонтальне додавання/віднімання 558
21.6. Синхронізація потоків 560
21.7. Використання SSE3 в розробці і оптимізації програм 560
21.7.1. Розрахункові завдання, що використовують x87 FPU 560
21.7.2. Обчислення з комплексними числами 562
21.7.3. Кодування відео 562
21.7.4. Векторні операції 563
21.8. SSSE3-доповнення 564
21.9. Лабораторна робота “SSE3-команди” 566
22. SSE4-РОЗШИРЕННЯ 568
22.1. SSE4.1-розширення 568
22.1.1. Прискорення відео 568
22.1.2. Векторні примітиви 569
22.1.3. Вставки/витягання 569
22.1.4. Скалярне множення векторів 570
22.1.5. Змішування 570
22.1.6. Перевірка бітів 570
22.1.7. Округлення 571
22.1.8. Читання WC пам’яті 571
22.2. SSE4.2-розширення 571
22.2.1. Обробка рядків 571
22.2.2. Підрахунок контрольної суми CRC32 572
22.2.3. Підрахунок популяції одиничних бітів 572
22.2.4. Векторні примітиви 572
22.3. Формати даних SSE 1-4 команд 572
23. AVX-РОЗШИРЕННЯ 574
23.1. Особливості використання команд AVX-розширення 574
23.2. Схема кодування 575
23.3. Набор команд AVX 575
23.4. Використання AVX-инструкций 576
23.5. Команди AVX-512 582
23.6. Лабораторна робота “AVX-команди” 583
СПИСОК ЛІТЕРАТУРИ 585

http://blogs.kpi.kharkov.ua/v2/asm/knigi/

Один день образованного человека дольше самого долгого века невежды
Посидоний из Апамеи (ок. 135 — 51 гг. до н.э.)

ВСТУП

Системне програмування – це велика область знань, завданням якої є розробка операційних систем (ОС), створення обслуговуючих програм і утиліт, програмування драйверів, дослідження показників продуктивності ОС, аналіз процесів, які виникають у її ядрі та між різними менеджерами і диспетчерами [11 – 13].

У цьому виданні розглядаються важливі аспекти системного програмування на мові асемблера. Асе́мблер (від англ. assembler – збиральник) – комп'ютерна програма, яка компілює початковий текст програми, написаний мовою асемблера, в програму машинною мовою [1].

Асемблер, як мова програмування, є мнемонічне позначення машинних кодів, які розуміють обчислювальні пристрої. Поки вони існуватимуть, мова асемблера буде актуальна. Незважаючи на бурхливий розвиток математичного забезпечення і алгоритмічних мов високого рівня, інтерес до мови асемблера для IBM PC постійно зростає. Мова асемблера допомагає розкрити всі секрети апаратного і програмного забезпечення. Більшість програмістів працюють з мовами високого рівня, де окремий оператор перетворюється в безліч мікропроцесорних команд. Кожна команда мови асемблера має взаємну однозначну відповідність з машинними командами, що дає підставу вважати її мовою низького рівня та відрізняє від таких мов високого рівня, як Pascal, BASIC і C++, для яких характерна трансляція одного оператора в безліч машинних команд.

Найчастіше мова асемблера використовується для безпосереднього управління операційною системою та для прямого доступу до апаратури. На асемблері пишуть оптимальні драйвери та невелики утиліти, прошивки BIOS, завантажувачі та ядра ОС, "движки" іграшок, віруси, компілятори і багато що інше. Асемблер служить інструментом для зв'язку з нетрадиційними зовнішніми пристроями. Також вона необхідна при оптимізації критичних блоків у прикладних програмах з метою підвищення їх швидкодії [6 – 8].

Відомо, що зрозуміти роботу комп'ютера можна, тільки освоївши асемблер і управління комп'ютером на самому нижньому рівні. **Знання асемблера і уміння працювати з комп'ютером на самому нижньому рівні обов'язкові для системного програміста.**

Мова асемблера існує для кожного типу мікропроцесорів або цілого сімейства мікропроцесорів.

При написанні цього підручника ураховувалося те, що питаннями системного програмування в НТУ "ХПІ" студенти займаються за такими навчальними дисциплінами: "Системне програмування", "Системне програмне забезпечення" та "Операційні системи". Тому знання, які студенти отримують з дисципліни "Системне програмування", є базовими для наступних дисциплін з системного програмування.

Мова асемблера з'явилася однією з перших. Але скільки би мов високого рівня не виникло після неї та скільки б ще не з'явилося в майбутньому, їх доля вирішена наперед, ще до їх появи. Всі вони колись зникнуть. Але тільки не мова асемблера . Через своє призначення вона не зникне доти, поки існуватимуть комп'ютери. Найголовнішим аргументом переконання в цьому є те, що будь-яка програма, яка написана на мові високого рівня, має виконуваний файл. Проте всі виконувані файли для свого вирішення були отримані шляхом перетворення команд мови високого рівня в машинні команди (коди), які у свою чергу і є командами мови асемблера. І тому якщо в якійсь мові високого рівня з'являються нові можливості щодо обробки чисел, то засоби для такої можливості вже повинні бути раніше закладені в асемблері. І поки такі можливості не будуть закладені в асемблері, то і не удосконалюватимуться інші мови. Тому мова асемблера постійно удосконалюється. ***А якщо фахівець вивчав мову асемблера 7 – 10 років тому, то можна стверджувати, що такий програміст вже не знає її, тому що за цей час з'явилося так багато нових наборів команд, що вони майже повністю змінили знання про асемблер.***

Екскурс в історію

Одним з перших удосконалень мови асемблера стала розробка набору команд для виконання операцій з плаваючою комою в мікропроцесорі 8087.

Потім були реалізовані система команд SIMD і технологія Intel® MMX™. Система команд SIMD використовувалася корпорацією Intel для збільшення рівня паралелізму при обробці цілочислових даних в мікроархітектурі P5 (січень 1997 р.) за допомогою застосування спеціальних команд (57 команд на першому етапі). Використовуючи набір команд технології Intel MMX, програмісти могли виконувати команди для декількох елементів даних, завантажених в регістри MMX (виконаних на основі стека співпроцесора), що забезпечувало підвищення продуктивності мультимедійних застосувань, зокрема застосувань для роботи з графікою, ігор, потокового відео і так далі.

У мікроархітектурі P6 корпорація Intel ввела набір команд Strea-ming SIMD Extensions (SSE) на основі нових регістрів XMM0-XMM7. Набір команд SSE, розроблений для мікропроцесора Intel® Pentium® III (початок 1999 р.), розширював можливості технології Intel MMX і дозволяв одночасно виконувати обчислення SIMD з чотирма елементами даних з плаваючою комою з одинарною точністю, використовуючи 128-розрядні регістри (назва XMM0-XMM7).

З мікроархітектурою Intel Netburst® (процесор Intel® Pentium® 4) корпорація Intel ввела набір команд SSE2 (червень 2000 р.) для розширення набору команд SSE (і технології Intel MMX). Набір команд SSE2 забезпечив можливість паралельно виконувати більше обчислень, розширюючи команди, введені в технології Intel MMX і набір команд SSE, та забезпечуючи підтримку 128-розрядних типів даних цілих чисел і упакованих чисел з плаваючою комою з подвійною точністю. У набір команд SSE2 були додані 144 нові ко-

манди, що забезпечило підвищення продуктивності різноманітних застосувань. Вважаємо за необхідне нагадати, що перший набір так званих базових команд теж містив всього лише 144 команди.

Після випуску мікропроцесора Intel® Pentium® 4 на базі 90-нанометрової виробничої технології був розроблений набір команд SSE3. У початковий набір команд SSE3 (лютий 2004 р.) входило 13 додаткових команд SIMD. Ці команди перш за все були направлені на поліпшення синхронізації потоків і розширення можливостей математичних операцій x87 і операцій з плаваючою комою.

Наступний крок був зроблений із створенням додаткового набору команд SSE3, який реалізований сьогодні в мікроархітектурі Intel® Core™. Додатковий набір команд SSE3, використовуваний в мікропроцесорах Intel® Xeon® 5100 (для серверів і робочих станцій) і Intel® Core™2 Duo (для мобільних і настільних ПК), додає 32 нових коди операцій, включаючи операції приведення у відповідність, множення і складання, що забезпечує ще більшу продуктивність системи.

Корпорація Intel для архітектури Intel® 64 (ISA), починаючи з мікропроцесора Core 2 Duo E6700 та корпорація AMD для МП Phenom 9600+ впровадили новий набір команд SSE4, який включає компоненти, що дозволяють використовувати розширені функціональні можливості мікропроцесору, підвищену продуктивність і покращувану енергоекономічність для більшості застосувань. Крім того, вже з 2000 року використовується 64-розрядний режим, який, з початку, фірмою Intel в мікропроцесорах Itanium використовував нову архітектуру IA-64. В цій архітектурі використовувався режим емуляції 32-розрядної архітектури, який був надзвичайно повільним. Компанія AMD запропонувала альтернативне вирішення проблеми збільшення розрядності мікропроцесора: було запропоновано ввести 64-розрядне розширення до вже існуючої 32-розрядної архітектури x86. Нова архітектура спочатку називалася x86-64, а потім AMD64 <http://ru.wikipedia.org/wiki/AMD64>. Фірма Intel створила набір інструкцій, повністю сумісний з AMD64. При цьому були додані ряд специфічних інструкцій, не присутніх в початковому наборі AMD64. Нова версія архітектури отримала назву EM64T.

З 2008 року на основі інструкцій SSE4 для мікропроцесорів Intel® Core™2 компанією Intel введені інструкції AVX (Advanced Vector Extensions), які збільшують ступінь паралелізму і пропускну спроможність в дійсних SIMD обчисленнях та зменшують навантаження на регістри завдяки неруйнуючим трехоперандним операціям.

В 2011 р. корпорація Intel підтвердила, що її процесори з кодовим ім'ям Haswell отримали підтримку набору інструкцій AVX2:

– AVX здібності розповсюдилися на цілочисельні (integer) типи і операції з ними;

– з'явилися інструкції, що дозволяють робити GATHER тип операцій; при таких операціях не потрібний, щоб дані знаходилися в безперервних ділянках пам'яті;

– FMA (Fused Multiply-Add) – одна інструкція виконує декілька операцій

над упакованими даними;

– додані спеціалізовані інструкції по маніпуляції бітів.

Не можна забувати і про те, що з постійним введенням нових команд не затихають роботи, які спрямовані на удосконалення модульності асемблера. З'явилися API-функції та програмування під 32-розрядну операційну систему Windows.

З виходом ОС Windows x64 актуальною є задача використання 64-розрядного розширення. ***Тому програмування під 16-розрядні додатки MS-DOS є не тільки архаїчним правилом, але й правилом "поганого тону", яке, звичайно, можна виправдовувати, але тільки ... ледарством розуму.***

Природно що у мови асемблера є й недоліки. Головним її недоліком є те, що програма, написана для одного типу комп'ютерів, не може бути перекомпільована і використана на інших типах комп'ютерів. Для кожного сімейства мікропроцесорів використовується своя мова асемблера зі своїм синтаксисом. Це є наслідком того, що мова асемблера дуже близька до машинних команд і до архітектури мікропроцесора. Тому, щоб глибше зрозуміти архітектуру мікропроцесора, без мови асемблера не обійтися.

Крім того, вже майже ніхто не пише великі закінчені програми на "чистому" асемблері. Це – безумство. Хоча треба визнати, що це безумство від великого розуму. Перед програмістами, що створюють такі програми, треба схилятися, бо теоретично, ці програми повинні бути ідеальними: найменшими за об'ємом і найбільш швидкодіючими. На практиці ж великі за об'ємом програми пишуться на мові високого рівня з асемблерними вставками для критичних ділянок коду. Навіть коли програма середньої складності написана на мові високого рівня, то все одно більшість бібліотечних функцій, що входять в Visual C++ чи Delphi, які були підключені, написані на асемблері. А паралельні обчислення, які засновані на технології SIMD, написані тільки з використанням сучасних асемблерних команд.

Нарешті, без асемблера хакерам **не обійтися** там, де початковий текст програми на мові високого рівня не доступний. В цьому випадку тільки асемблер може допомогти розібратися в програмі.

У підручнику розглядається **макроасемблер masm32** автора **Стівена Хатчессона** (Сідней, Австралія). Всі програми протестовані в 11-й версії цього пакета.

Вивчення сучасного асемблера – завдання, звичайно ж, не зовсім просте. І цей підручник написаний для тих, хто прагне отримати загальні уявлення про мову асемблера і полегшити розуміння асемблерних програм, а також для тих, хто про цю мову ніколи не чув.

Фірма Microsoft підтримує свій продукт під назвою Microsoft Macro Assembler. Вона продовжує розвиватися і до сьогодні, останні версії включено в набори DDK.

Хочеться закінчити словами: "*программирование на ассемблере дает ощущение власти над компьютером, а жажда власти – один из сильнейших инстинктов человека" – В. Пирогов,* http://asm.shadrinsk.net

http://blogs.kpi.kharkov.ua/v2/asm/knigi/

Каждый человек имеет определенный горизонт.
Когда он сужается и становится малым, он превращается в точку.
Тогда человек говорит: "Это моя точка мнения".
Д. Гильберт

1. ЕТАПИ СТВОРЕННЯ ПРОГРАМИ МОВОЮ АСЕМБЛЕРА

Асемблер може створювати лістинг програми з номерами рядків, адресами змінних, операторами та таблицею перехресних посилань символів і змінних. Нарівні з асемблером використовується програма, що називається компонувальником (linker), яка об'єднує окремі файли, створені асемблером, в єдину виконувану програму.

При вивченні мови асемблера та розробленні реальних програм цією мовою під операційну систему (ОС) Windows процес розподілення комірок пам'яті для збереження кодів команд та чисел виконується автоматизовано.

Етапи створення програми мовою асемблера такі:

- підготовка (або внесення змін) вихідного тексту програми;
- асемблювання програми (отримання об'єктного коду);
- компонування програми (отримання виконуваного файлу програми);
- налагодження програми (виправлення помилок).

Зазвичай ці етапи циклічно повторюються, оскільки при виявленні помилок на всіх етапах доводиться повертатися до першого етапу і вносити зміни в текст програми для виправлення помилок.

Асемблери бувають двох типів: однофазні та двофазні.

Однофазні асемблери можуть обробляти тільки такі програми, в яких символи з'являються в полі назви до того, як на них дається посилання в полі операндів.

Трансляція програми двофазним асемблером проводиться в два етапи: у першій фазі трансляції асемблер послідовно рахує кожне речення початкової програми, частково її транслює і будує повну таблицю символів; у другій фазі асемблер закінчує трансляцію програми, використовуючи таблицю символів першої фази як вхідну інформацію.

Для обох типів асемблерів символ, записаний у полі операндів, обов'язково повинен бути визначений у полі назви, інакше буде повідомлення про помилку.

1.1. Підготовка вихідного тексту програми в Masm32

Текст програми мовою асемблера записується в один чи декілька текстових файлів. Імена файлів і їх розширення можуть бути будь-які, але прийнято використовувати розширення *.asm. Ці файли є текстовими, їх можна підготовити за допомогою будь-якого текстового редактора, наприклад блокнота

або NotePad, і зберігати у вигляді звичайних файлів у форматі ASCII. Використання Word небажано, тому що такий текст містить велику кількість службових символів, які є невидимими та призводять до помилок трансляції.

Найпростіше при підготовці тексту можна використати саме середовище Masm32.

Структура програми під Win32 може бути такою:

```
TITLE  <имя программы
.386              ; директива определения типа микропроцессора
.model flat       ; задання линейной модели памяти
.data             ; директива определения данных
                        ; данные, которые определены
.data?                  ; неинициализированные данные
        ;
.code                       ; директива начала кода программы
label:                  ; метка начала программы
              ;<code>
ret                     ; возвращение управления ОС
 end  label             ; окончание программы
```

1.2. Отримання об'єктного коду в Masm32

Підготовлені тексти програми з розширенням *.asm є вхідними даними для програм, які називаються асемблерами (наприклад, програми Masm, Tasm, Wasm, Fasm). Завдання програми асемблера – перетворити текст програми у форму двійкових команд, які може виконати МП. Після асемблювання отримують файли об'єктних модулів, що мають розширення *.obj. Процес асемблювання частіше називають ***трансляцією***.

Для простоти використання асемблера Masm32 (www.masm32. com) його необхідно інсталювати на системний диск c:/. Якщо поставити асемблер Masm32 не на системний диск, а на інший логічний диск, то в такому випадку необхідно конкретно вказувати шляхи підключення бібліотек розташування API-функцій.

Програма ml.exe з Masm32 виконує трансляцію початкового тексту програми в проміжний об'єктний файл. Програма link.exe виконує компонування об'єктного (об'єктних) файлу (файлів) і бібліотек в єдину виконувану програму.

В асемблері masm32 трансляцію файлів із розширенням asm можна виконати **з командного рядка** (якщо програма складається з одного файлу):

ml.exe /c /coff ім'я_файлу.asm

Створений за допомогою цієї команди об'єктний файл має формат COFF. Якщо параметр /coff не заданий, то форматом створеного об'єктного файлу буде OMF.

Параметр /C означає, що виконується тільки асемблювання. Виклик лінкеру link.exe поки не виконується, тому що його викликаємо вручну пізніше.

Опції компіляції ML.EXE:

ML [/ опції] список_файлів [/link опції_лінкеру]
/AT– дозволити надмаленьку модель пам'яті tiny (.COM файл);
/Bl<linker> – використовувати альтернативний лінкер;
/c – асемблювати без лінкування (використання зовнішнього лінкеру (наприклад link.exe) для компонування файлів);
/Cp – зберегти регістр ідентифікаторів користувача;
/Cu – перевести всі ідентифікатори у верхній регістр;
/Cx – зберегти регістр publics, externs;
/coff – генерувати об'єктний файл у форматі COFF;
/D<name>[=text] – визначити текст макросу;
/EP – виводити лістинг препроцесора в стандартний потік виведення;
/F <hex> – встановити розмір стека (в байтах);
/Fe<file> – встановити ім'я виконуваного файлу;
/Fl[file] – генерувати лістинг;
/Fm[file] – генерувати карту пам'яті;
/Fo<file> – ім'я об'єктного файлу;
/FPi – генерувати 80x87 емулятор;
/Fr[file] – встановити обмежену інформацію;
/FR[file] – встановити повну інформацію;
/G<c|d|z> – використовувати формат виклику в стилі Pascal, C або Stdcall;
/H<number> – встановити максимальну довжину зовнішніх імен;
/I<name> – додати include-шлях;
/link <опції лінкеру та бібліотеки>;
/nologo – заборонити повідомлення ML.EXE про авторське право;
/Sa – максимізувати лістинг;
/Sc – видавати таймінги в лістинг;
/Sf – видавати лістинг першого проходження асемблера;
/Sl<width> – встановити ширину рядка;
/Sn – подавити лістинг таблиці символів;
/Sp<length> – встановити довжину сторінки;
/Ss<string> – встановити підзаголовок;
/St<string> – встановити заголовок;
/Sx – скласти список умовних виразів помилок;
/Ta<file> – асемблювати не .ASM файли;
/w – аналогічно /W0 /WX;
/WX – розцінювати попередження як помилку;
/W<number> – встановити рівень попереджень;
/X – проігнорувати INCLUDE шлях оточення;
/Zd – додати номери рядків у налагоджувальну інформацію;
/Zf – зробити всі символи загальнодоступними;
/Zi – додати символічну налагоджувальну інформацію;
/Zm – забезпечити сумісність із MASM 5.10;
/Zp[n] – встановити вирівнювання структур;

/Zs – виконати тільки перевірку синтаксису.

Для трансляції та лінкування *простих* програм **зручно** використовувати середовище **masm32**. Для цього необхідно запустити програму masm32 Editor, вставити або набрати свою програму, зберегти її з розширенням asm та вибрати шлях меню Project/ Assemble & Link. Якщо помилок не буде, то буде створений exe-файл.

1.3. Отримання виконуваного файлу програми в Masm32

Після успішної трансляції початкового тексту асемблерної програми результат у вигляді об'єктного файлу передається компонувальнику link.exe. Він виконує об'єднання об'єктних модулів в один файл. Сегменти, визначені в програмі, групуються відповідно до інструкцій, що містяться в об'єктному файлі. Вся інформація про розміщення сегментів записується в заголовок виконуваного файлу.

Результатом компонування є виконувані файли, що мають розширення *.exe.

Компонувальник link.exe оперує з OBJ-файлами як у форматі COFF, так і у форматі OMF, при цьому виконується автоматичне перетворення формату файлу з OMF в COFF. Зазвичай при генерації виконуваних файлів використовується формат COFF. Якщо файл об'єктного модуля повинен використовуватися у застосуванні, написаному на Visual C++ .NET, то формат його обов'язково повинен бути COFF. У той же час при використанні об'єктного файлу у застосуванні, розробленому в Borland Delphi 2005, єдиним сприйманим форматом є OMF.

У процесі трансляції початкового тексту програми виконуються такі дії:

1. Аналізуються директиви умовного асемблювання, і у випадку істинності вказаних у них умов виконуються ті або інші кроки.
2. Розгортаються макроси.
3. Обчислюються константні вирази, при цьому вони заміщаються обчисленими значеннями.
4. Декодуються команди та операнди, що не знаходяться в пам'яті. Наприклад, декодується команда mov EAX, 2008, оскільки вона не має операндів, розташованих у пам'яті.
5. Зберігаються зсуви змінних у пам'яті відносно зміщення у сегментах, в яких ці змінні розташовані.
6. Сегменти та їх атрибути розміщуються в об'єктному файлі.
7. В об'єктному файлі зберігаються адреси, що переміщуються (relocatable addresses).
8. За необхідності створюється файл лістингу.
9. Безпосередньо програмі link.exe передаються деякі директиви (наприклад, INCLUDELIB і DOSSEG).

В асемблері masm32 для генерування 32-розрядних EXE-файлів може бути використаний один із командних рядків:

link.exe /SUBSYSTEM:WINDOWS /OPT:NOREF ім'я_файлу.obj

або

link.exe /SUBSYSTEM:CONSOLE ім'я_файлу.obj

У **більшості випадків** для виконання консольних програм зручно створити командний bat-файл, наприклад, зі змістом:

```
ml.exe  /c  /coff   7_9_1.asm
link.exe /subsystem:console 7_9_1.obj
```

Після того, як такий командний bat-файл створено, необхідно його виконати (двічі натиснути на нього).

Схему асемблювання, компонування і виконання програми наведено на рис. 1.1.

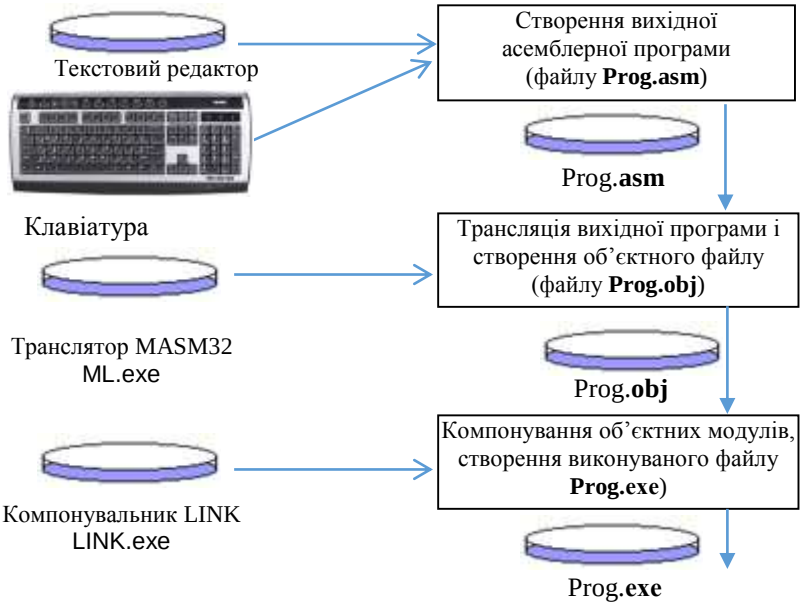


Рис. 1.1. Схема асемблювання, компонування і виконання програми

Структура програми (не тільки на асемблері) визначається декількома чинниками:

– архітектурою мікропроцесора;

– особливостями тієї операційної системи, під керуванням якої ця програма виконуватиметься;

– правилами роботи вибраного компілятора – різні компілятори ставлять різні вимоги до початкового тексту програми.

MASM і TASM підтримують ***спрощену сегментацію***. Суть такої сегментації в тому, що директиви .CODE і .DATA можуть з'являтися в тексті програми кілька разів. Транслятор потім збирає код і дані разом. Основною метою такого підходу є можливість наблизити в тексті програми дані до тих рядків, де вони використовуються.

Основні опції компонування наведено в табл. 1.1.

Таблиця 1.1

/DEBUG	Налагодження (при використанні налагоджувача)
/DEBUGTYPE:CV| COFF	Тип налагодження: codeview / coff (вибирає вихідний формат налагоджувальної інформації. Налагоджувачі SoftIce і Visual C++ можуть обробляти CV (codeview))
/DEF:ім'я_файлу	DEF файл (вказує файл визначення (.def). Використовується з dll для функцій, що експортуються)
/DLL	DLL (вихідний файл із розширенням .DLL)
/LIBPATH:path	Шлях до бібліотек (вказує шлях до файлів бібліотек (*.lib).
/I< ім'я >	Встановлює шлях для inc-файлів
/OUT: ім'я_файлу	Out:ім'я_файлу (для зміни імені вихідного файлу)
/SUBSYSTEM:{...}	Підсистема (вибір ОС: NATIVE|WINDOWS| CONSOLE| WINDOWSCE| POSIX)

1.4. Зменшення розміру програми за рахунок ключів компіляції та лінкування в Masm32

При розгляді цього питання використовуємо матеріал циклу статей «Сам себе Iczelion» (http://www.cyberforum.ru/post3944787.html). Найпростіший крок для зменшення розміру програми – це використання службових ключів.

Ключ /ALIGN: розмір – ключ /ALIGN[MENT] – указує компонувальнику на необхідність вирівнювання сегментів у виконуваному файлі на межу, кратну значенню розміру сегмента. Тут розмір – це число, яке дорівнює степеню двійки ($2^0 = 1$, $2^1 = 2$, ..., $2^9 = 512$ до 64 К включно).

Якщо про цей ключ не згадувати, то вирівнювання дорівнює 512 байт для сумісності з програмами, створеними для Windows 95/98. Упізнати значення вирівнювання можна за допомогою редактора бінарних файлів hiew32. Для цього відкриваємо exe-файл програмою hiew32, натискаємо F4 (Mode) і вибираємо Hex. Після чого натискаємо F8 (Header). Розмір сегментів розташовується в полі «File alignment».

Ключ компіляції /Cp – «зберегти регістр символів у всіх ідентифікаторах» дозволяє не писати щоразу в тексті асемблерного файлу «option casemap:none».

Ключ компіляції /I: шлях – «встановити шлях для файлів, що включаються» – дозволяє не указувати щоразу повний шлях до inc-файлів.

Ключ компіляції /Gz — «визначити узгодженість про імена та стиль виклику функцій stdcall» дозволяє скоротити напис «.model flat, stdcall» до «.model flat».

Ключ /LIBPATH: шлях указує лінкеру, де знаходяться бібліотеки імпорту, що дозволяє не писати щоразу повний шлях до lib-файлів.

Таким чином, командній bat-файл може бути таким:

```
\masm32\bin\ml /c /Cp /Gz /I\masm32\include /coff "msgbox.asm"
\masm32\bin\Link /SUBSYSTEM:WINDOWS /ALIGN:256
/LIBPATH:\masm32\lib "msgbox.obj"
```

Значення можна зробити меншими, ніж того вимагає система. В цьому випадку з'являється попередження системи

```
LINK: warring LNK4108: ALIGN specified without /DRIVER; image may not run,
```

але exe-файл формується.

Крок другий – об'єднання сегментів коду та даних.

За цей крок відповідає ключ /MERGE:from = to. Цьому ключу можна привласнити значення секції коду (0C0000040h – атрибути секції «виконувати» та «читати») і функції секції даних (60000020h – атрибути секції «читати» і «писати»)

```
\ml /c /Cp /Gz /I\masm32\include /coff "msgbox.asm"
\Link /SUBSYSTEM:WINDOWS /ALIGN:16 /LIBPATH:\masm32\lib ^
/MERGE:.rdata=.text "msgbox.obj"
```

Об'єднати сегмент коду та сегмент даних можна також, якщо використовувати опцію лінкеру /SECTION. Опція командного рядка компонувальника link.exe /SECTION:name,[[!]{DEKPRSW}][,ALIGN=#] дозволяє примусово призначати атрибути секціям PE-файлу. Для секції можна задати один або декілька атрибутів. Слід задавати всі атрибути, які повинна мати секція. Якщо який-небудь знак атрибуту не вказаний, то його біт буде вимкнений. Якщо не вказаний атрибут R, W або E, то існуючий стан читання, запису або виконання залишається незмінним. Щоб інвертувати атрибут, перед його символом указують знак «!». За допомогою параметра ALIGN=# можна задати значення вирівнювання для конкретної секції. Значення знаків атрибутів можуть бути такими.

D / Discardable / Marks the section as discardable / секція позначається як вивантажувана;

E / Execute / The section is executable / секція є виконуваною;

K / Cacheable / Marks the section as not cacheable / секція позначається як некешована;

P / Pageable / Marks the section as not pageable / секція позначається як секція без сторінкової організації;

R / Read / Allows read operations on data / допускаються операції читання даних;

S / Shared / Shares the section among all processes that load the image / секція спільно використовується всіма процесами, що завантажують образ;

W / Write / Allows write operations on data / допускаються операції запису даних.

В даному випадку секції з ім'ям «.text» (програми, що містять код) і що вже мають атрибути R (доступна для читання) та E (здійснима), встановлюється атрибут W (доступна для запису).

Змінений bat-файл може бути таким:

```
\masm32\bin\ml /c /Cp /Gz /I\masm32\include /coff "msgbox.asm"
\masm32\bin\Link /SUBSYSTEM:WINDOWS /ALIGN:16 ^
/LIBPATH:\masm32\lib /SECTION:.text,W "msgbox.obj"
```

1.5. Налагодження програми

Будь-яка програма потребує налагодження (виправлення помилок). Сучасні налагоджувальні програми для асемблера платформи x86, наприклад програми Dbg_x86 (http://www.microsoft.com/whdc/devtools/ debugging), SoftICE (яка вже не підтримується, але за звичкою ще використовується), Emu8086 v.4.04, OllyDbg 1.10 (**http://cracklab.ru**/), OllyDbg 2 (**http://www.ollydbg.de/**), Syser (www.syser.com), ImDbg.v1.50 (Immunity Debugger на http://cracklab.ru/) дозволяють у процесі виконання програми контролювати значення регістрів загального призначення чи змінних і змінювати їх.

За допомогою налагоджувачів можна, наприклад, переглядати вміст різних ділянок пам'яті, виконувати програму крок за кроком, змінювати програмний код, зберігати зміни в .exe-файлі та ін.

Для налагодження професійних **асемблерних** програм, які застосовують останні версії команд паралельної обробки дійсних чисел або нові API-функції, які постійно з'являються, використовують середовище **Visual Studio**, порядок роботи в якому буде наведено далі.

При написанні програм **обов'язково** використовувати довідник MSDN (**http://msdn.microsoft.com/ru-ru/library**).

1.6. Редактори тексту

Як редактор тексту при написанні та налагодженні невеликих навчальних програм простіше за все використовувати офісний блокнот, в якому пишуть текст. Виконують такі програми з використанням командного bat-файлу. ***Як середовище виконання програми на початковому етапі частіше використовують саме середовище masm32*** **(програму masm32 Editor**) або командний рядок файлового менеджера FAR.

У цей час застосування Windows готуються, як правило, за допомогою інтегрованих середовищ розробки (**Integrated Development Environment, IDE**), які забезпечують весь цикл розробки застосування (підготовку початкового тексту, компіляцію, налагоджування та пробного виконання) у повноекранному інтерактивному режимі в середовищі Windows. При цьому сама

IDE широко використовує стандартні засоби графічного інтерфейсу Windows: меню, діалогові вікна, інструментальні кнопки та ін. Для невеликих програм можна використовувати блокнот Notepad++ (http://notepad-plus.sourceforge.net/) або UltraEdit (www.ultraedit.com). Всі такі редактори підтримують підсвічування синтаксису, можливість запуску зовнішніх застосувань.

Для професійного використання можна рекомендувати IDE-редактори: RadASM *(***Ketil Olsen, http://radasm.cherrytree.at),** Negatory Assembly Studio (www.negatory.com/asmstudio/), Visual SlickEdit (www.slickedit.com), **WinAsm** (**Antonis Kyprianou,** http://winasm. how.to/), Source Insight 3.5 (Source Dynamics, Inc., www.sourceinsight. com) та інші. **Найбільш повними можливостями з асемблювання програм володіє середовище Visual Studio останніх версій, тому що підтримує всі нові API-функції та нові асемблерні команди.**

При самостійному опрацьовуванні навчального матеріалу можна використати програми з сайта ресурсу **wasm.ru.**

1.7. Настройка середовища Visual Studio 2013

Для створення та налагоджування програми в Visual Studio 2013 необхідно виконати наступні дії.

1. Створити проект:

File/ New/ Project (Файл/ Создать/ Проект).

У лівому стовпці вікна, що відкрилося, виділяємо Win32, а потім «Консольное приложение Win32» (Win32 Console Application). Після чого вказуємо ім'я проекту і директорію його розташування (рис. 1.2.)

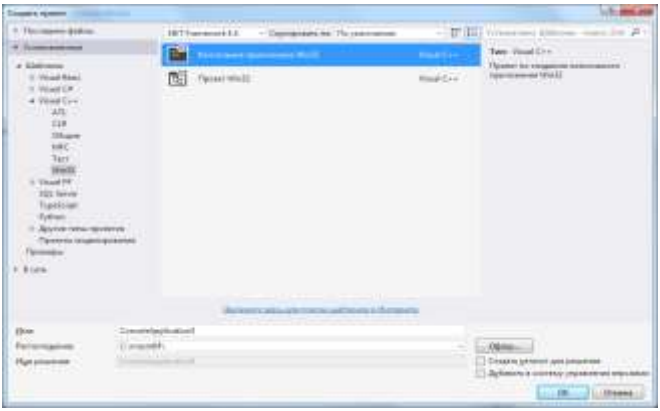


Рис. 1.2. Вікно «Создать проект»

У вікні, що відкрилося «Мастер приложения Win32» підсвічуємо мишкою пункт «Параметры приложения» (Application Settings) і відзначаємо пункт «Пустой проект» (Empty project).

І натискаємо кнопку «Готово» (Finish) (рис. 1.3).

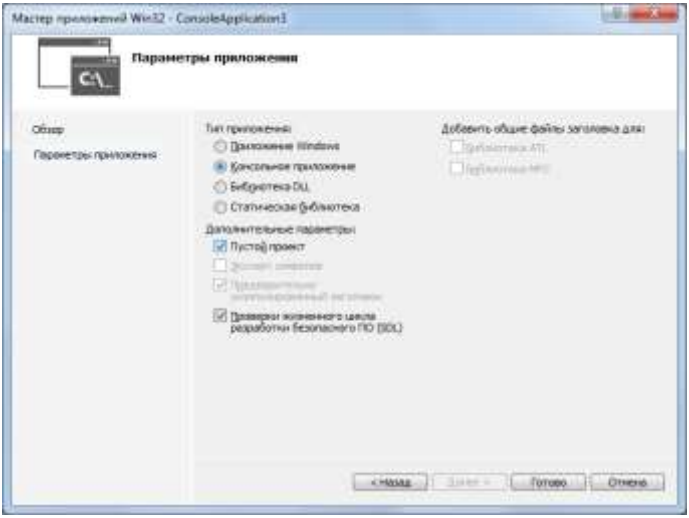


Рис. 1.3. Вікно «Мастер приложений…»

2. У головному меню вибираємо ПРОЕКТ/Настройки сборки (PROJECT/ Build Customizations) і у вікні, що відкрилося, вибираємо значення masm() (рис. 1.4).

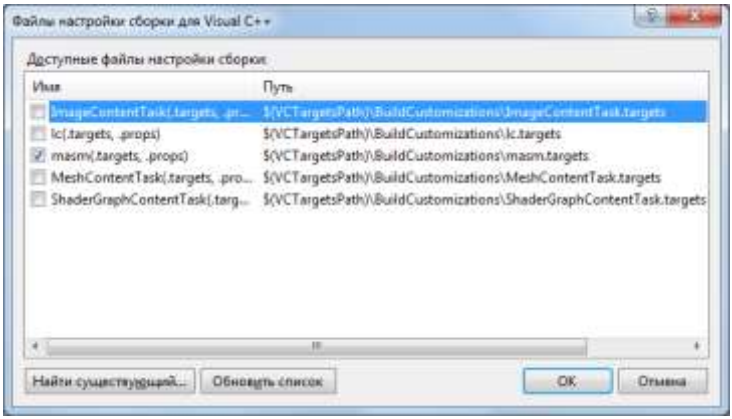


Рис. 1.4. Вікно «Файлы настройки…»

3. Настроюємо значення «тформы решения». Для цього підводимо покажчик мишки до значка . Внаслідок чого з'являється ім'я «Параметры панели инструментов Стандартная». Ставимо відмітку в пункті «Платформы решения» (рис. 1.5).

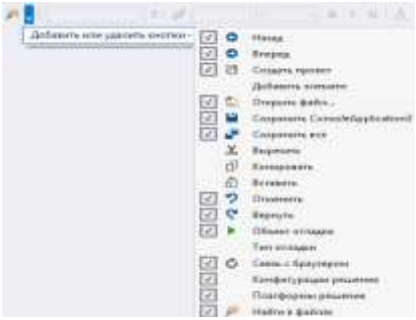


Рис. 1.5. Вікно обрання «Платформы решения»

З'явилося вікно Win32. У цьому вікні в правій її частині вибираємо значення «Диспетчер конфигурации» (рис. 1.6).

Рис. 1.6. Вікно «Диспетчер конфигурации»

У вікні диспетчера конфігурації замість активної платформи рішення вибираємо значення «Создать».

Замість платформи ARM вибираємо значення x64 (рис. 1.7).

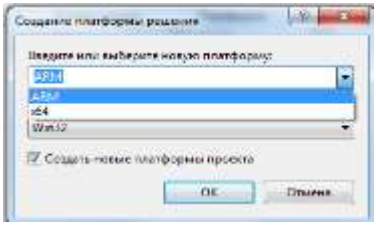


Рис. 1.7. Вікно «Создать платформы решения»

4. У меню головного вікна вибрати ПРОЕКТ/ Свойства/ Компоновщик/ Дополнительно (рис. 1.8).

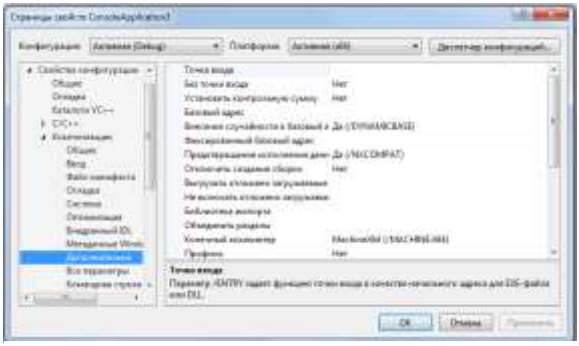


Рис. 1.8. Вікно «Страница свойств…»

У правій колонці вікна вибрати «Точка входа», а в кінці цього рядка, натиснувши на зображення трикутника вибрати значення «Изменить».

У вікні, що з'явилося, ввести значення точки входу (влучна) в програму, наприклад, main. Натиснути «Применить» та «ОК».

5. Вибираємо «Вид/ Обозреватель решений». Підсвічуємо мишкою «Файлы ресурсов», натискаємо праву клавішу мишки і вибираємо «Добавить/ Создать элемент» (рис. 1.9).

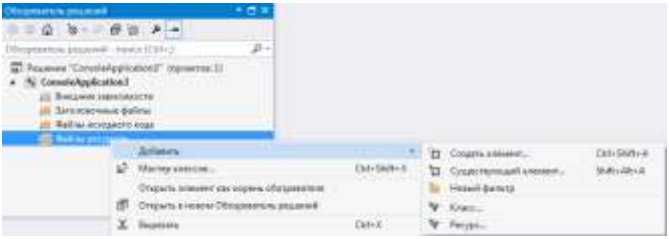


Рис. 1.9. Вікно обрання «Обозреватель решений»

6. У новому вікні вибираємо ім'я програми, наприклад myAsm.asm та визначаємося з директорією її розташування. Натискаємо «Добавить».

7. У основному вікні пишемо програму.

Для відображення нумерації рядків необхідно вибрати: «СЕРВИС/ Параметры / Текстовый редактор/ Все языки/ Общие» та включити «Номера строк».

8. Для виконання програми вибираємо пункти «СБОРКА/ Собрать решение» (або Ctrl+Shift+B).

9. Для налагодження програми вибираємо пункти «ОТЛАДКА/ Шаг с обходом (F10)». Стани регістрів відображаються, якщо в середині вікна «Регистры» натиснути праву клавішу мишки та вибрати ЦП (центральний процесор) (рис. 1.10).

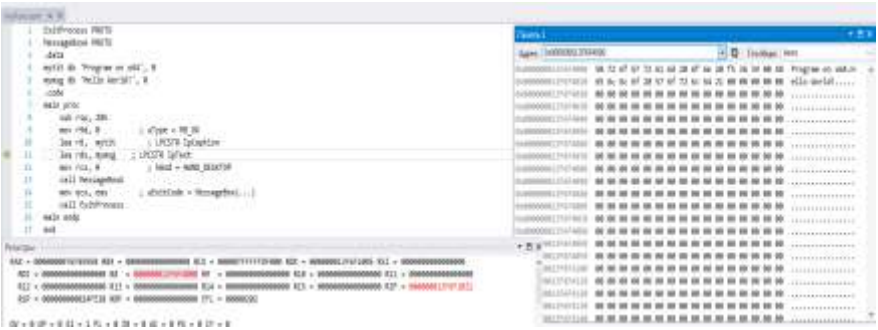


Рис. 1.10. Вікна налагоджувача середовища Visual Studio 2013

1.8. Угорська нотація

Деякі змінні в іменах мають малі літери. Це не що інше як виконання угорської нотації, що була прийнята при програмуванні під Windows на честь легендарного програміста Microsoft Чарльза Симоні. Суть цієї нотації полягає в тому, що ім'я змінної починається з малої літери або букв, які відзначають тип даних змінної. Наприклад, префікс sz в szCmdLine означає, що рядок завершується нулем (string terminated by zero). Префікс h в hInstance і hPrevInstance означає описувач (handle); префікс i в iCmdShow означає ціле (integer).

Угорська нотація допомагає уникати помилок в програмі ще до її компонування. Оскільки ім'я змінної описує і саму змінну і тип її даних, то набагато знижується вірогідність введення в програму помилок, пов'язаних з розбіжністю типу даних у змінних.

У табл. 13.1 наведено префікси змінних, які використовуються найчастіше.

Таблиця 13.1 – Префікси змінних

Префікс	Тип даних
c	Символ
by	BYTE (беззнаковий символ)
n	Коротке ціле
i	Ціле
x, y	Ціле (координати *x* та *y*)

cx, cy	Ціле (довжина *x* та *y*), *c* позначає "рахунок" — (count)
b або *f*	BOOL (булеве ціле); *f n*означає "прапорець" — (flag)
w	WORD (беззнакове коротке ціле)
l	LONG (довге ціле)
dw	DWORD (беззнакове довге ціле)
fn	Функція
s	Рядок
sz	Рядок із завершенням нулем
h	Описувач (handle)
p	Вказівник (pointer)

Контрольні запитання

1. Які переваги має асемблер над іншими мовами програмування?
2. Які етапи створення програми мовою асемблера?
3. Які дії виконуються при трансляції програми?
4. Для чого виконується компонування програми та які опції компонування частіше за все використовуються?
5. Який вплив ключів компіляції та лінкування на розмір програми?

2. АРХІТЕКТУРА МІКРОПРОЦЕСОРА INTEL

2.1. Архітектурні особливості 32-розрядного МП INTEL

Успішне застосування мови асемблера неможливе без знання архітектури мікропроцесорів Intel, відомості про яку наведено за адресою: http://www.intel.com/products/processor/manuals.

Архітектура мікропроцесорних систем (МПС) – функціональні можливості комплексу апаратних, програмних і мікропрограмних засобів, які використовуються для подання даних, машинних операцій, опису алгоритмів та процесів обчислення [9].

Елементами архітектури є:

- типи оброблюваних даних та засоби їх подання;
- структури пам'яті та засоби їх адресації;
- структури та системи команд (мікрокоманд);
- структури мікропроцесорів (МП) та їх програмні моделі.

32-розрядні мікропроцесори можуть оперувати з 8-, 16- і 32-бітовими операндами, рядками байтів, слів і подвійних слів, а також із бітами, бітовими полями і рядками бітів.

У архітектуру мікропроцесорів вводять засоби налагодження і тестування.

2.1.1. Типи даних МП INTEL

32-розрядні мікропроцесори безпосередньо підтримують знакові і беззнакові цілі числа, рядки байтів, цифр і символів, бітові рядки, вказівники і числа з рухомою крапкою. У сімействі x86 слова прийнято записувати в двох суміжних байтах пам'яті, починаючи з молодшого. ***Адресою слова* є адреса його молодшого байта.** Подвійні слова записують у чотирьох суміжних байтах. Цей порядок називається Little-Endian Memory Format. В інших сімействах мікропроцесорів застосовують і зворотний порядок – Big-Endian Memory Format. Для взаємного перетворення форматів слова є інструкція XCHG, подвійного слова – BSWAP (486+).

Біт (Bit) – одиниця інформації. Біт у пам'яті задається *базою* (адресою слова) і *зміщенням* (номером біта в слові).

Бітове поле (Bit Field) – група до 32 суміжних бітів, що розміщуються не більш ніж у 4 байтах.

Бітовий рядок (Bit String) – набір суміжних бітів довжиною до 4 Гбітів.

Байт (Byte) – 8 бітів.

Числа без знака – байт (8 розрядів)/слово (16 розрядів)/подвійне (32 розряди)/збільшене у 4 рази (64 розряди) слово (Unsigned Byte/ Word/Double Word/Quade Word), 8/16/32/64 біти.

Цілі числа зі знаком – байт/слово/подвійне/збільшене у 4 рази слово (Integer Byte/ Word/Double Word/Quade Word). Одиничне значення найстаршого біта (знак) є ознакою від'ємного числа, що зберігається в додатковому коді.

Двійково-десяткові числа (BCD – Binary Coded Decimal):

• 8-розрядні упаковані (Packed BCD), що містять два десяткових розряди в одному байті;

• 8-розрядні неупаковані (Unpacked BCD), що містять один десятковий розряд у байті (значення бітів 7–4 при додаванні і відніманні є несуттєвими, при множенні та діленні вони повинні бути нульовими).

Рядки – байти, слова і подвійні слова (Bit String, Byte String, Word String, Double Word String) довжиною до 4 Гбайт.

Вказівники:

• довгий вказівник (48 біт) – 16-бітовий селектор (чи сегмент) і 32-бітове зміщення;

• короткий вказівник – 32-бітове зміщення;

• просто вказівник (32 біт, єдиний тип вказівника для 8086 і 80286) – 16-бітовий селектор (чи сегмент) і 16-бітове зміщення.

Числа у форматі з рухомою крапкою і упаковані 80-бітові BCD-числа обробляються блоками FPU мікропроцесорів класу 486 і вище, а також співпроцесорами 8087/287/387. Упаковані 64- і 128-бітові дані обробляються мікропроцесорами з MMX (мультимедійні інструкції), SSE (потокові інструкції) та AVX (векторні інструкції).

Дійсні числа у форматі з рухомою крапкою:

- одинарної точності (Single Precision), 32–23 біт мантиса, 8 біт порядок;
- подвійної точності (Double Precision), 64–52 біт мантиса, 11 біт порядок;
- підвищеної точності (Extended Precision), 80–64 біт мантиса, 15 біт порядок.

Двійково-десяткові 80-бітові упаковані числа (18 десяткових розрядів і знак).

Упаковані дійсні числа одинарної точності у *форматі з рухомою крапкою* обробляються блоком XMM, який призначений для роботи з векторними та скалярними інструкціями.

Упаковані цілі числа, знакові і беззнакові, обробляються блоком MMX:

• упаковані байти (Packed byte) – вісім байтів;

• упаковані слова (Packed word) – чотири слова;

• упаковані подвійні слова (Packed doubleword) – два подвійних слова;

• збільшене у 4 рази слово (Quadword) – одне слово.

32-розрядні мікропроцесори платформи x86 мають регістри:

• регістри загального призначення;

• вказівник інструкцій;

• регістр прапорців;

• регістри сегментів;
• системні адресні регістри;
• керуючі регістри;
• регістри налагодження;
• регістри тестування;
• модельно-специфічні (залежні від конкретної моделі мікропроцесора) регістри.

Типи даних, які обробляються блоками FPU/MMX та XMM, наведено на рис. 2.1 [2–5].

2.1.2. Регістри 32-розрядного МП Intel

Основні регістри мікропроцесора архітектури IA-32, з якими працюють прикладні програми, показано на рис. 2.2 [2].

Ці регістри належать до видимої для прикладних програм частини архітектури x86 і являють собою розширення набору регістрів 16-розрядних мікропроцесорів 8086/8088 і 80286.

У 32-розрядних мікропроцесорів регістри загального призначення EAX, EBX, ECX, EDX складаються з двох 16-бітових половинок, молодша з яких теж ділиться на дві половинки. До них можна незалежно звертатися за символічними іменами AH, BH, CH, DH (старші байти – High) і AL, BL, CL, DL (молодші байти – Low).

МП платформи x86 мають акумуляторну структуру. Це означає, що всі операції виконуються «через» акумулятор, тобто через регістр EAX.

Регістри-вказівники ESP (Stack Pointer – вказівник стека), EBP (Base Pointer – базовий регістр або вказівник бази) та індексні регістри ESI (Source Index – індекс джерела), EDI (Destination Index – індекс призначення) допускають 32-розрядне звертання.

Адреса поточної інструкції зберігається в 32-бітовому вказівнику команд EIP (Instruction Pointer).

Регістри в командах можуть адресуватись явно. У ряді команд існує неявне використання регістрів:

- EAX – множення, ділення, введення і виведення подвійного слова;
- AX – множення, ділення, введення і виведення слова;
- AL – множення, ділення, введення і виведення байта; десяткова арифметика, трансляція (XLAT);
- AH – множення і ділення байта;
- EBX – трансляція;
- ECX – лічильник циклів і вказівник довжини рядкових операцій;
- CL – зміщення з указанням змінної;
- EDX – множення і ділення слова, введення і виведення з непрямою адресацією;
- SP – операції зі стеком;
- ESI, EDI – рядкові операції.

У мікропроцесорах IA-32 всі ці регістри мають префікс E (Extended – розширений).

Рис. 2.1. Типи даних, які обробляються блоками FPU/MMX та XMM

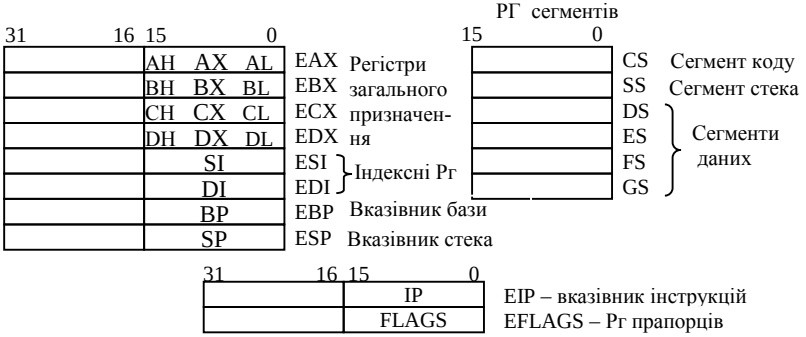


Рис. 2.2. Загальні Рг 32-розрядного МП

Відсутність префікса в імені означає посилання на молодші 16 біт розширених регістрів.

У реальному режимі за умовчуванням розрядність адреси і даних – 16 біт.

У захищеному режимі розрядність адреси і даних за умовчуванням визначається дескриптором кодового сегмента.

Вказівник інструкцій EIP (регістр команд) містить зміщення наступної виконуваної інструкції (команди) щодо бази сегмента коду. При 16-бітовій адресації використовуються тільки молодші 16 біт (IP).

Регістр прапорців EFLAGS є розширенням регістра FLAGS до 32 бітів (рис. 2.3).

0	0	..	0	0	0	0	0	0	0	VIPx	VIFx	ACx	VMx	RFx	0	NTx	IOPL	O	DFc	IFx	TFx	SFs	ZFs	0	AFs	0	PFs	1	CFs

Рис. 2.3. Регістр прапорців EFLAGS (*x* – системний прапорець; *s* – прапорець стану; *c* – керуючий прапорець)

Біти 0–15, визначені для 8086 і 80286, мають колишнє призначення. ***Прапорці станів*** OF, SF, ZF, AF, PF і CF зберігають ознаки результатів виконання арифметичних і логічних операцій над операндами, розміщеними в регістрах загального призначення й у пам'яті. ***Операції пересилання даних на прапорці стану не впливають.*** Значення цих прапорців аналізується при виконанні умовних інструкцій. Крім прапорців стану в регістрі є ***керуючі та системні прапорці,*** що впливають на поведінку мікропроцесора, у тому числі на обробку маскованих апаратних переривань (IF) і на напрямок руху в рядкових операціях (DF). Прапорці стану можуть бути скопійовані в регістр AH і, навпаки, завантажені з нього. Крім того, прапорці стану можуть бути завантажені з блоку FPU, і тоді результати виконання інструкцій у FPU можна використовувати в умовних інструкціях.

Призначення бітів регістра EFLAGS:

• **ID** (Id Flag) – прапорець доступності команди ідентифікації CPUID (P5+ і деякі 486).

• **VIP** (Virtual Interrupt Pending) – віртуальний запит переривання (P5+).

• **VIF** (Virtual Interrupt Flag) – віртуальна версія прапорця IF (дозволу переривання) для багатозадачних систем (P5+).

• **AC** (Alignment Check) – прапорець контролю вирівнювання. При виконанні програм із рівнем привілеїв 3 у випадку звертання до операнда, що не

вирівняний за відповідною межею (2, 4, 8 байтів), і при встановленому прапорці AC відбувається виключення #AC з нульовим кодом похибки. На рівнях привілеїв 0, 1, 2 контроль вирівнювання не здійснюється (486+).

• **VM** (Virtual 8086 Mode) – у захищеному режимі вмикає режим віртуального мікропроцесора 8086. Спроба використати привілейовані інструкції у цьому режимі спричинює виключення #GP. Біт VM може встановлюватися тільки в захищеному режимі (інструкцією IRET на нульовому рівні чи привілеєм перемикання задач на будь-якому рівні). На біт VM не діє інструкція POPF, а інструкція PUSHF у цей біт завжди заносить 0. Одиничне значення біта VM може зберегтися тільки в регістрі EFLAGS, воно встановлюється при перериванні, перемиканні задач чи переході в режим SMM.

• **RF** (Resume Flag) – прапорець поновлення. Використовується разом із регістрами точки зупину .

• **IOPL** (Input/Output Privilege Level) – рівень привілеїв введення-виведення.

• **NT** (N sted Task Flag) – прапорець вкладеної задачі.

• **OF** (Overflow Flag) – прапорець переповнення. Встановлюється, якщо результат арифметичної операції не вміщується в операнді призначення.

• **DF** (Direction Flag) – прапорець керування напрямком у рядкових операціях. При одиничному значенні цього біта індексні регістри, що беруть участь у рядкових операціях, автоматично декрементуються на кількість байтів операнда, при нульовому – інкрементуються. Прапорець керування програмно змінюється інструкціями CLD і STD.

• **IF** (Interrupt-enable Flag) – прапорець керування перериваннями. При одиничному значенні біта **IF** дозволяється виконання маскованих апаратних переривань. На цей прапорець можна впливати програмно, явними інструкціями (CLI → IF = 1 та STI → IF = 0) і неявними інструкціями відновлення регістра прапорців; мікропроцесор при обробці переривань маніпулює прапорцем автоматично. У захищеному режимі з'являються додаткові тонкощі.

• **TF** (Trap Flag) – прапорець трасування (покрокового режиму). При його встановленні після виконання кожної команди викликається внутрішнє переривання типу 1 (INT 1).

• **SF** (Sign Flag) – прапорець знака. Вказує на одиничне значення старшого біта результату – ознаку від'ємного числа.

• **ZF** (Zero Flag) – прапорець нульового результату.

• **AF** (Auxiliary Flag) – прапорець додаткового переносу (позики) у зошиті для десяткової арифметики.

• **PF** (Parity Flag) – прапорець паритету. Встановлюється при парній кількості одиниць результату.

• **CF** (Cany Flag) – прапорець переносу (позики) старшого біта в арифметичних операціях. Цей прапорець може бути програмно встановлений чи скинутий спеціальними інструкціями (STC, CLC, CMC); завдяки цій властивості

його широко використовують для сигналізації умови завершення процедур (наприклад, сигналізації про помилки).

Прапорці перенесення і переповнювання для операції складання наведено в табл. 2.1.

Таблиця 2.1

Перенесення з розрядної сітки	Перенесення в знаковий біт	Прапорці
Є	Є	CF = 1, OF = 0
Є	Немає	CF = 1, OF = 1
Немає	Є	CF = 0, OF = 1
Немає	Немає	CF = 0, OF = 0

Прапорці позики і переповнювання для операції віднімання наведено в табл. 2.2.

Таблиця 2.2

Позика в 7-й (старший) біт	Позика з 7-го (знакового) біту	Прапорці
Є	Є	CF = 1, OF = 0
Є	Немає	CF = 1, OF = 1
Немає	Є	CF = 0, OF = 1
Немає	Немає	CF = 0, OF = 0

Регістри сегментів містять 16-бітові вказівники (в реальному режимі) чи селектори дескрипторів (у захищеному режимі) сегментів CS (Code Segment – сегмент кодів команд), SS (Stack Segment – сегмент стека), DS (Data Segment – сегмент даних), ES, FS і GS – додаткові сегменти даних. Сегментних регістрів FS і GS у 16-розрядних мікропроцесорах не було. Вміст сегментних регістрів не може бути модифікованим (збільшеним чи зменшеним), вони допускають лише завантаження і збереження вмісту.

Використання сегментних регістрів визначається типом звертання до пам'яті. Для багатьох типів звертань можливе застосування альтернативних сегментних регістрів, що вводяться префіксами команд CS:, SS:, DS:, ES:, FS: чи GS:. З кожним із шести сегментних регістрів пов'язані програмно-недоступні приховані регістри дескрипторів (їх ще називають Segment Descriptor Cache – кеш сегментних регістрів), які завантажуються автоматично при завантаженні відповідних сегментних регістрів.

У захищеному режимі в регістри дескрипторів із таблиці дескрипторів завантажуються 32-бітова базова адреса, 32-бітовий **ліміт (розмір сегмента)** і атрибути сегментів. За вмістом цих прихованих регістрів при кожному звертанні до пам'яті виконуються обчислення лінійної адреси і перевірка захисту; причому саме ці регістри задають властивості сегментів як у захищеному, так і в

реальному режимах. Образ цих 96-бітових регістрів доступний у режимах SMM і внутрішньосхемній емуляції (ICE), формати яких специфічні для кожної моделі (покоління) мікропроцесорів. У реальному режимі, в якому мікропроцесор входить за апаратним скиданням, ліміт (розмір сегмента) фіксований – 64 кбайт, атрибути не використовуються, а як базова адреса заноситься значення сегментного регістра, зсунуте на 4 біт уліво. В захищеному режимі ліміт може задаватися в межах 1 байт – 4 Гбайти. Позаштатним перемиканням із захищеного режиму в реальний режим можна досягти і нестандартного розміру сегментів.

Крім регістрів загального призначення, необхідних для використання прикладних програм, мікропроцесори мають ряд регістрів системного призначення. У мікропроцесорах 8086/88 цих регістрів не було, а в мікропроцесорі 80286 була присутня лише частина з них і не в повному обсязі.

Керуючі регістри (Control Registers) CR0, CR1, CR2, CR3 зберігають ознаки стану мікропроцесора.

Системні адресні регістри призначені для посилань на сегменти і таблиці в захищеному режимі.

Регістри* GDTR *(Global Descriptor Table Register) і* IDTR *(Interrupt Descriptor Table Register) програмно завантажуються 6-байтовими операндами, що включають 32-бітову лінійну базову адресу і 16-бітовий ліміт глобальної таблиці дескрипторів і таблиці дескрипторів переривань. У регістр задачі TR (Task Register) і регістр селектора локальної таблиці дескрипторів LDTR (Local Descriptor Table Register) завантажуються 16-бітові селектори дескрипторів сегмента стану задачі TSS і локальної таблиці дескрипторів LDT. Це завантаження викликає автоматичне завантаження самих дескрипторів, що містять 32-бітові поля лінійної базової адреси і ліміту, а також полів атрибутів у зв'язані з ними невидимі регістри дескрипторів.

Регістри налагодження (Debug Register) призначені для задавання точок зупину і керування ними.

Регістри* DR0...DR3 *(Linear Breakpoint Address* 0...3*) зберігають 32-бітові лінійні адреси точок зупину.

***Регістри* DR4, DR5** у 80386 і 486 не використовуються, звертання до них еквівалентне звертанню до регістрів DR6, DR7. У мікропроцесорі Pentium при увімкненому розширенні налагодження звертання до цих регістрів викликає вимкнення неприпустимого коду операції (#UD).

Регістр* DR6 *(Breakpoint Status) відбиває стан контрольної точки.

Регістр* DR7 *(Breakpoint Control) керує встановленням контрольних точок.

Склад ***регістрів тестування (Test Register)*** варіюється залежно від типу мікропроцесора. Мікропроцесори 80386 мали тільки два регістри, призначені для тестування кеша сторінкової переадресації: TR6 і TR7. Для мікропроцесора 486 склад регістрів розширений: TR3 – регістр даних внутрішнього кеша;

TR4 – тестовий регістр стану кеша; TR5 – керуючий регістр тестування кеша; TR6 (Test Control) – керуючий регістр для тесту кешування сторінок; TR7 (Test Status) – регістр даних для тесту кешування сторінок.

У мікропроцесорах Pentium і вище тестові регістри входять до групи модельно-специфічних регістрів MSR. Для цих мікропроцесорів звертання до регістрів TRx викликає виключення #UD неприпустимого коду операції.

Модельно-специфічні регістри **MSR (*Model-Specific Registers*)** призначені для керування розширеннями налагодження, моніторингом продуктивності, машинним контролем, кешуванням ділянок фізичної пам'яті й інших функцій. Їх призначення прив'язується до мікроархітектури конкретного мікропроцесора, склад змінюється від моделі до моделі, доступ привілейований у регістрі ECX, що дозволяє необмежено (до 4 млрд) збільшувати кількість цих регістрів.

Розмір регістрів та їх позначення зведено в табл. 2.3.

Таблиця 2.3

Тип	Біти	Регістри
General	8	al cl dl bl ah ch dh bh
	16	ax cx dx bx sp bp si di
	32	eax ecx edx ebx esp ebp esi edi
	64	rax rcx rdx rbx rsp rbp rsi rdi
Segment	16	es cs ss ds fs gs
Control	32	cr0 cr2 cr3 cr4
Debug	32	dr0 dr1 dr2 dr3 dr6 dr7
FPU	80	st0 st1 st2 st3 st4 st5 st6 st7
MMX	64	mm0 mm1 mm2 mm3 mm4 mm5 mm6 mm7
SSE	128	xmm0 xmm1 xmm2 xmm3 xmm4 xmm5 xmm6 xmm7
AVX	256	Для 32-разрядной ОС:YMM0 — YMM07 Для 64-разрядной ОС:YMM0 — YMM15

Команди FPU – це команди співпроцесора для обробки даних розміром у 80 розрядів (бітів).

Команди MMX – це команди паралельної обробки цілочислових даних розміром у 64 біти.

SSE-команди – це команди паралельної обробки даних у форматі з рухомою крапкою розміром у 128 бітів.

2.1.3. Моделі пам'яті МП Intel

Модель пам'яті визначає спосіб організації програм і даних у пам'яті комп'ютера. У 32-розрядній архітектурі мікропроцесора Intel Pentium використовуються три моделі пам'яті:

- **плоска**, або лінійна, (flat memory model), яка є безперервним простором адрес, що називається лінійним. У ньому розташовуються програмний код, дані, а також ділянка стека. Адресний простір у цій моделі адресується побайтно, а діапазон адрес дорівнює 2^{32}.
- **сегментована** (segmented memory model), яка складається з трьох окремих просторів адрес, що називаються сегментами. При цьому програмний код, дані і стек розміщуються в окремих сегментах пам'яті. Для того щоб звернутися до байта в пам'яті, програма формує логічну адресу, що складається з адреси сегмента (селектора сегмента) і зсуву. Програми, що виконуються в 32-розрядному режимі, можуть використовувати до 16 383 сегментів різного розміру, кожний з яких може мати розмір 2^{32} байт.

Механізм перетворення адрес при використанні сегментованої моделі такий: спочатку всі адреси сегментів, визначених у програмі, відображаються на загальний лінійний простір адрес, доступний мікропроцесору. Потім логічна адреса операнда в пам'яті перетворюється на лінійну адресу з простору адрес. Всі ці перетворення прозорі для працюючої програми. Подібна модель дозволяє підвищити надійність роботи програм. Наприклад, якщо розмістити програмний код і ділянку стека в різних сегментах, то у випадку різкого збільшення розміру стека не відбудеться перекриття програмного коду даними зі стека;

- **модель реального режиму адресації** (real-address mode memory model), яка використовується в мікропроцесорах 8086. Ця модель пам'яті підтримується Для того щоб забезпечити сумісність із раніше розробленими 16-розрядними застосуваннями. У цій моделі застосовується механізм сегментації, причому максимальний розмір сегмента не перевищує 64 кбайт. Максимальний розмір лінійного адресного простору, доступного в цьому режимі, становить 2^{20} байт.

При розробленні 32-розрядних програм мовою асемблера зазвичай використовується лінійна, або плоска, модель пам'яті. При 32-розрядній адресації операндів логічна адреса складається з 16-розрядного селектора сегмента і 32-розрядного зсуву. При 16-розрядній адресації логічна адреса складається з 16-розрядного селектора сегмента і 16-розрядного зсуву.

Інструкції асемблера включають найрізноманітніші команди, що працюють як з операндами, так і без них. Деякі команди потребують явного указання операндів, тоді як інші використовують операнди за умовчанням. Дані операнда-джерела можуть знаходитися в регістрі, пам'яті, порту введення-виведення або задаватися безпосередньо в інструкції. Операнд-приймач може розташовуватися в оперативній пам'яті, регістрі або бути портом введення-виведення.

Селектор сегмента можна вказати явно або неявно. Селектор сегмента завантажується в сегментний регістр, а сам регістр вибирається залежно від типу виконуваної операції (табл. 2.4).

Адреса операнда формується за схемою *сегмент: зсув.* Залежно від використовуваної моделі пам'яті адреса змінної в пам'яті може формуватися як 16:16 або 16:32. Зсув операнда називається його ефективною або виконавчою адресою (Effective Address, EA).

Таблиця 2.4

Тип операції	Регістр	Опис
Команди процесора	CS – програмний сегмент	Використовується при викликах команд процесора
Звернення до стека	SS – сегмент стека	Використовується при операціях зі стеком, а також в операціях із пам'яттю, в яких базовими є регістри ESP та EBP
Дані	DS – сегмент даних	Використовується в операціях із даними
Рядки-приймачі	ES – сегмент даних, який адресується регістром ES	Використовується як операнд-приймач в рядкових операціях

Процесор автоматично вибирає сегмент відповідно до умов. При збереженні операнда в пам'яті або при завантаженні його з пам'яті в якості сегментного регістра за умовчанням використовується DS, але можна і явно вказати сегментний регістр.

Наприклад, потрібно зберегти вміст регістра EAX в пам'яті, що адресується сегментним регістром ES і зсувом, який знаходиться в регістрі EBX. В цьому випадку можна використовувати команду

```
mov ES:[EBX], EAX
```

Після імені сегмента указується символ двокрапки.

На рівні мікропроцесора заміна сегмента задається спеціальним префіксом заміни, який є однобайтовим числом, розташованим перед кодом команди. В деяких випадках заміна сегмента не допускається:

- для сегмента програмного коду всі команди використовують виключно сегментний регістр CS;
- при виконанні рядкових операцій рядок-приймач адресується тільки регістром ES;
- операції розміщення в стеці і витягання зі стека завжди використовують регістр SS.

Деякі інструкції мікропроцесора потребують явної ініціалізації сегментних регістрів. У таких випадках селектор сегмента може бути витягнутий із 16-розрядного регістра або змінної в пам'яті, як, наприклад, у такій команді:

```
mov DS, BX        ; переслати вміст регістра BX в регістр DS.
```

У цій команді селектор сегмента, що знаходиться в регістрі BX, поміщається в сегментний регістр DS.

У деяких випадках селектор сегмента може визначатися через 48-розрядний вказівник, що знаходиться в пам'яті. При цьому молодше подвійне слово

містить 32-розрядний зсув, а старше слово – 16-розрядний селектор сегмента.

В більшості випадків програмісти мають справу з ефективною адресою операнда, тобто з тією частиною повної адреси операнда, яка визначає зсув операнда у вказаному сегменті. Дуже часто терміни «ефективна адреса» і «зсув» сприймаються як синоніми при аналізі способів адресації операндів, хоча це не зовсім так.

Ефективна адреса операнда, що знаходиться в пам'яті, може бути задана декількома способами. У загальному випадку можливо визначити ефективну адресу операнда, якщо вона складається з декількох частин:

– зсуву, що має 8-, 16- і 32-розрядне значення;

– бази, що є вмістом одного з регістрів загального призначення;

– індексу, що є вмістом одного з регістрів загального призначення;

– масштабного множника, що дорівнює 2, 4 або 8.

Існують певні обмеження щодо застосування регістрів загального призначення як базових, або індексних, при формуванні ефективної адреси:

– регістр ESP не можна використовувати як індексний регістр;

– якщо як базовий використовується регістр ESP або EBP, то сегментним регістром буде SS. У решті випадків сегментним регістром за умовчанням є DS.

2.1.4. Способи адресації 32-розрядного МП Intel

При виконанні будь-якої програми мікропроцесор звертається до пам'яті, у якій зберігаються команди і дані. *Спосіб або метод визначення в команді адреси операнда або адреси переходу називається режимом адресації або просто адресацією.*

У найбільш простому режимі адресації, що наззивається прямою адресацією, адреса знаходиться в самій команді. Однак використання цього режиму, до речі, передбаченого в більшості сучасних мікропроцесорів, призводить до надмірної довжини команд, особливо в умовах, коли постійно збільшується ємність пам'яті. Тому сьогодні у мікропроцесорах застосовується багато інших режимів адресації, спрямованих на досягнення таких **цілей:**

- визначення адреси пам'яті найменшою кількістю бітів, що скорочує довжину команд;
- обчислення адреси щодо поточної команди (так звана відносна адресація), що забезпечує навантаження програм без модифікацій у будь-яку ділянку пам'яті;
- здійснення доступу до комірок пам'яті, адреси яких обчислюються при виконанні програми, що спрощує доступ до регулярних структур даних;
- оперування адресами у формі, найзручнішій для таких структур даних, як масиви і стеки.

Режими адресації значно розширюють гнучкість і зручність користування системою команд.

Система команд 32-розрядних мікропроцесорів передбачає **11 режимів адресації** [6–11]. При цьому тільки в двох випадках операнди не зв'язані з пам'яттю. Це операнд, що береться з кожного 8-, 16- чи 32-бітового регістра мікропроцесора, і безпосередній операнд (8, 16 чи 32 біт), що міститься в самій команді.

1. ***Безпосередня адресація.*** Регістр сегмента не використовується. Наприклад,

MOV EAX, 12345678h ; завантаження в EAX const = 12345678_{16}.

2. ***Регістрова адресація.*** Наприклад, MOV EAX, ECX.

Інші дев'ять режимів так чи інакше звертаються до пам'яті.

При звертанні до пам'яті ефективна адреса обчислюється з використанням таких компонентів:

• ***зміщення (Displacement чи Disp)*** – 8-, 16- чи 32-бітового числа, включеного в команду;

• ***бази (Base)*** – вмісту базового регістра. Зазвичай використовується для зазначення на початок деякого масиву;

• ***індексу (Index)*** – вмісту індексного регістра. Зазвичай використовується для вибору елемента масиву;

• ***масштабу (Scale)*** – множника (1, 2, 4 чи 8), зазначеного у коді інструкції. Цей елемент, який використовується для зазначення розміру елемента масиву, доступний тільки при 32-бітовій адресації.

Ефективна адреса обчислюється за формулою

$$EA = Base + Index \cdot Scale + Disp.$$

Окремі складові в цій формулі можуть бути відсутніми.

Відмінності 16- і 32-бітових режимів адресації наведено у табл. 2.5.

Таблиця 2.5

Компонент	16-бітова адресація	32-бітова адресація
Базовий регістр	BX або BP	32-бітовий регістр загального призначення
Індексний регістр	SI або DI	32- бітовий регістр загального призначення, крім ESP
Масштаб	Немає (завжди 1)	1,2, 4 або 8 (число)
Зміщення	0, 8 або 16 біт	0, 8 або 32 біт

Процесор може працювати з 32- або 16-бітовою адресацією. 16-бітова адресація функціонує так само, як і в мікропроцесорах 8086 і 80286, при цьому як компоненти адреси використовуються молодші 16 біт відповідних регістрів.

При 32-бітовій адресації застосовуються розширені 32-розрядні регістри і додаткові режими з масштабуванням індексу.

У реальному режимі за умовчанням використовується 16-бітова адресація, але за допомогою *префікса зміни розрядності адреси* (Address Length Prefix) для поточної інструкції можна перемкнутися на 32-бітову. При цьому з'являються додаткові можливості адресації (масштабування), але обчислюване значення ефективної адреси все одно не може подолати 64-кілобайтновий бар'єр – при спробі звернутися за такою ефективною адресою генерується виключення #GP – General Protection Fault.

У захищеному режимі адресація за умовчанням визначається бітом D дескриптора використовуваного кодового сегмента при D = 0 – 16 біт і при D = 1 – 32 біти. Префікс розрядності адреси перемикає розрядність для поточної інструкції на протилежну.

3. ***Пряма адресація*** (Direct Mode). Адреса **EA = Disp**. Наприклад,

```
MOV EAX, mem1 ;  завантаження в EAX змінної з іменем mem1.
```

У реальних програмах для задання статичних змінних зазвичай використовують директиви визначення даних, які дозволяють посилатися на статичні змінні не за адресою, а за іменем. Тоді, якщо в сегменті, вказаному в ES, була описана змінна word_var розміром у слово, можна записати ту саму команду як

```
mov  ax, es:word_var
```

У такому випадку асемблер сам замінить слово word_var на відповідну адресу. Якщо селектор сегмента даних знаходиться в DS, то ім'я сегментного регістра при прямій адресації можна не указувати (DS використовується за умовчанням).

Якщо, наприклад, операнд у пам'яті є подвійним словом, то в команді потрібно явно вказати старшу або молодшу частину за допомогою оператора PTR:

```
mov AX, word ptr meml ; завантажити в AX змінну з пам'яті з ім'ям
                      ; meml розміром у слово.
```

Якщо потрібно зберегти в регістрі AX значення старшого слова змінної meml, то слід застосувати команду

```
mov AX, word ptr meml+2
```

У цьому випадку в регістр AX поміщаються старші два байти змінної meml.

У загальному випадку оператор PTR можна зобразити у вигляді

тип PTR *вираз*

За допомогою оператора PTR змінна або мітка, що задається виразом, може трактуватися як змінна або мітка вказаного типу. Цей тип задається одним з імен або значень (табл. 2.6).

Вираз включає будь-які операнди. Типи BYTE, WORD, DWORD,

FWORD, QWORD, TWORD та DQWORD використовуються тільки з операндами.

Оператор PTR служить для організації доступу до об'єкта (наприклад, для доступу до старшого байта змінної розміру WORD), який при іншому способі адресації призвів би до помилки компіляції. Якщо PTR не використовується, то асемблер має на увазі тип посилання за умовчанням.

Приклад використання деяких директив визначення даних із табл. 2.6 наведено в програмі з лістингу 2.1.

Таблиця 2.6

Тип	Розмірність
DB (BYTE)	1 байт – 8 розрядів
DW (WORD)	2 байти – 16 розрядів
DD (DWORD)	4 байти – 32 розряди
DF (FWORD)	6 байтів – 48 розрядів
DP	6 байтів – далекий вказівник
DQ (QWORD)	8 байтів – 64 розряди
DT (TBYTE)	10 байтів – 80 розрядів
REAL4	32-розрядне коротке дійсне число
REAL8	64-розрядне дійсне число
REAL10	80-розрядне розширене дійсне число

Лістинг 2.1. Приклад використання деяких директив визначення даних:

```
title l1 – использование директив
        ; masm32
        ; компиляция – ml.exe /c /coff   1_7L.asm
        ; компоновка – link.exe /SUBSYSTEM:CONSOLE 1_7L.obj
.386                    ; директива определения типа микропроцессора
.model flat             ; задание линейной модели памяти
.data                   ; директива определения данных
a1B byte 11h        ; запись в 8-разрядную ячейку памяти
                                        ; с именем a1B числа 11h =  8A1
a2B db 22h      ; запись в 8-разрядную ячейку памяти с именем  a2B числа 22h
b1W word 3A3Bh   ; запись в 16-разрядную ячейку памяти числа 3A3Bh
b2W dw 4A4Bh     ; запись в 16-разрядную ячейку памяти числа 4A4Bh
c1D dword 5A5B5C5Dh             ; запись в 32-разрядную ячейку памяти
      c2D dd 6A6B6C6Dh          ; запись в 32-разрядную ячейку памяти
      d1Q dq 7A7B7C7D7E7F7A7Bh  ; 8 байтов
      d2Q qword 8A8B8C8D8E8F8A8Bh       ; 8 байтов
      e1t  tbyte  9A9B9C9D9E9F9A9B9C9Dh  ; 10 байтов
.code                           ; директива начала кода программы
 _start:                        ; метка начала программы с именем start
      ret                       ; возвращение управления ОС
       end _start               ; окончание программы с именем _start
```

Вікно налагодження програми з лістингу 2.1 програмою OllyDbg наведено на рис. 2.4. З ділянки розташування даних видно, що дані розташовуються в пам'яті, починаючи з молодших адрес.

Адресація відрізняється для реального і захищеного режимів. У реальному режимі (так само, як і в режимі V86) зсув завжди 16-бітовий. Це означає, що ні безпосередньо вказаний зсув, ні результат складання вмісту різних регістрів у складніших методах адресації не можуть перевищувати межі слова. При роботі в Windows DOS4G, PMODE і в інших ситуаціях, коли програма запускається в захищеному режимі, зсув не повинен перевищувати межі подвійного слова.

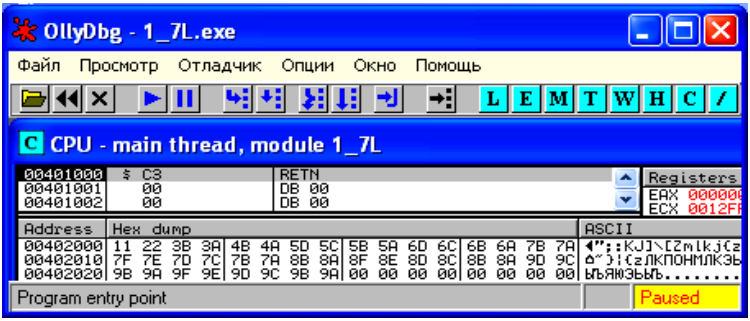


Рис. 2.4. Вікно налагодження програми з лістингу 2.1

4. ***Непряма регістрова адресація*** (Register Indirect Mode). Адреса **EA = Base**. Наприклад,

```
MOV EAX, [EBX] ; завантаження в EAX змінної з пам'яті,
                                    ; адреса якої знаходиться в регістрі EBX
```

До мікропроцесора 80386 для непрямої адресації можна було використовувати тільки BX, SI, DI і BP, але потім обмеження були зняті і адресу операнда дозволили прочитувати також із EAX, EBX, ECX, EDX, ESI, EDI, EBP і ESP. Наприклад, наступна команда розміщує у регістр EAX слово з елемента пам'яті, селектор сегмента якої знаходиться в DS, а зсув – у EBX:

```
mov  eax, [ebx] ;   eax :=  ds:[ebx] – завантаження в EAX
                                    ; змінної з пам'яті, адреса якої знаходиться в регістрі EBX
```

Як і у випадку з прямою адресацією, DS використовується за умовчанням, але не завжди: якщо зсув беруть із регістрів ESP, EBP або BP, то як сегментний регістр застосовується SS. У реальному режимі можна вільно працювати зі всіма 32-бітовими регістрами, необхідно тільки стежити, щоб їх вміст не перевищував межі 16-бітового слова.

5. ***Базова адресація*** (Baset Mode). Для формування ефективної адреси використовується тільки вміст базового регістра (база). Така адресація називається базовою і служить для адресації динамічних структур даних, наприклад

рядків і масивів. **Однак на відміну від МП x86, де базовими регістрами могли бути лише BX та BP, а (забігаючи наперед) індексними – тільки SI та DI, то 32-розрядні МП допускають використання як базових та індексних практично всіх регістрів загального призначення.**

Розглянемо приклад програмного коду:

```
lea EBX, meml   ; завантаження в EBX адреси пам'яті,
                                        ; де знаходиться змінна meml
 mov EAX, [EBX]  ; завантаження в EAX змінної з пам'яті,
                                        ; адреса якої знаходиться в регістрі EBX
```

У цьому прикладі meml – змінна в пам'яті розміром у подвійне слово. Перша команда завантажує адресу змінної в регістр EBX, а друга поміщає в регістр EAX значення, що міститься за адресою, яка знаходиться у EBX.

В загальному випадку адреса **EA = Base + Disp**. Наприклад,

```
MOV EAX, [4 + EBX] ; завантаження в EAX змінної з пам'яті, адреса якої
                   ; знаходиться в регістрі EBX зі зміщенням на чотири байти
```

Така форма адресації використовується в тих випадках, коли в регістрі знаходиться адреса початку структури даних, а доступ необхідно здійснити до якого-небудь її елемента.

Ще один варіант застосування базової адресації із зсувом – доступ із підпрограми до параметрів, переданих до стека з використанням регістра BP (EBP) як бази і номера параметра як зсуву. Інші допустимі форми запису цього способу адресації такі:

```
mov   eax, [ebp] + 2
mov   eax, 2[ebp]
```

До появи мікропроцесора 80386 як базовий регістр дозволялося використовувати тільки BX, BP, SI або DI і зсув міг бути тільки байтом або словом (із знаком). **Починаючи з моделі 80386 і вище, мікропроцесори Intel дозволяють додатково використовувати EAX, EBX, ECX, EDX, EBP, ESP, ESI і EDI, так само, як і для звичайної непрямої адресації.** За допомогою цього методу дозволяється організовувати доступ до одновимірних масивів байтів: зсув відповідає адресі початку масиву, а число в регістрі – індексу елемента масиву, який потрібно рахувати. Очевидно, що якщо масив складається не з байтів, а зі слів, доведеться множити базовий регістр на два, а якщо з подвійних слів – на чотири.

6. ***Індексна адресація*** (Index Mode). Адреса **EA = Index + Disp**. Наприклад,

```
MOV EDX, [EDI]  ; завантаження в EDX змінної з пам'яті,
                                        ; адреса якої знаходиться в регістрі EDI
```

Регістр ESP не можливо використовувати як індексний. Однак це не означає, що ESP не дозволено використовувати як другий операнд:

```
mov EAX,[ECX][ESP].
```

Недопустима тільки конструкція, в якій ESP помножується на масштабний коефіцієнт.

Таким чином, поняття базової і індексної адресації в 32-розрядних МП незначно розмиваються. Якщо регістр указується з масштабним множником, то це, звичайно, індексна адресація. Якщо ж множник відсутній, то адресацію і через EBX, і через ESI однаково можна віднести **як до базової, так і до індексної.**

7. ***Масштабована індексна адресація*** (Scaled Index Mode). Адреса **EA = Scale × Index + Disp**. Наприклад,

```
MOV ECX, [EDI*4 + EAX] ; завантаження в ECX змінної з пам'яті,
                       ; адреса якої знаходиться як застосування значень
                       ; регістра EAX та перемноження значення регістра EDI на чотири
MOV EAX, [ESI*2] + 2  ; завантаження в EAX змінної  з пам'яті,
                       ; адреса якої знаходиться як застосування двійки та перемноження
                       ; значення регістра ESI на два
```

Множник, який дорівнює 1, 2, 4 або 8, відповідає розміру елемента масиву – байту, слову, подвійному або збільшеному учетверо слову. З регістрів у цьому варіанті адресації з масштабуванням можна використовувати тільки EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, але не SI, DI, BP або SP.

8. ***Базово-індексна адресація*** (Based Index Mode). Адреса **EA = Base + Index**. Наприклад,

```
MOV EAX, [EBX]ALPHA[ESI] ; завантаження в EAX змінної з пам'яті з ім'ям ALPHA,
; адреса якої знаходиться як застосування значень двох регістрів: EBX та ESI
```

У цьому методі адресації зсув операнда в пам'яті обчислюється як сума чисел, що містяться в двох регістрах, і зсуву, якщо він вказаний. Всі перераховані нижче команди є різними формами запису тієї самої дії:

```
MOV EAX, [EBX + ESI + 2]
MOV EAX, [EBX][ESI] + 2
MOV EAX, [EBX + 2][ESI]
MOV EAX, [EBX][ESI + 2]
MOV EAX, 2[EBX][ESI]
```

У регістр EAX поміщається слово з елемента пам'яті із зсувом, що дорівнює сумі чисел, що містяться у EBX, ESI, і числа 2. З 16-бітових регістрів так можна складати тільки BX + SI, BX + DI, BP + SI і BP + DI, а з 32-бітових – всі регістри загального призначення. Як і для прямої адресації, замість безпосереднього зазначення числа дозволено використовувати ім'я змінної, заданої одній з директив визначення даних. Таким чином, можна розрахувати, наприклад, число з двовимірного масиву: якщо задана таблиця 10 × 10 байт, 2 – зсув її початку від початку сегмента даних (на практиці використовується ім'я цієї

таблиці), BX = 20, а SI = 7, наведені команди прочитають слово, яке складається з сьомого і восьмого байтів третього рядка. Якщо таблиця складається не з одиночних байтів, а із слів або подвійних слів, зручніше використовувати як найповнішу форму – базову адресацію з індексуванням і масштабуванням.

9. ***Масштабована базово-індексна адресація*** (Based Scaled Index Mode). Адреса **EA = Base + Scale · Index**. Наприклад,

```
MOV ECX, [EBX + 4*ESI] ; завантаження в ECX змінної
                       ; з пам'яті, адреса якої знаходиться як застосування значень
                       ; регістра EBX та перемноження значення регістра ESI на чотири
```

10. ***Базово-індексна адресація зі зміщенням*** (Based Index Mode with Displacement). Адреса **EA = Base + Index + Disp**. Наприклад,

```
MOV EAX, [EBX + ESI + 00ABCDF] ; завантаження в EAX змінної з пам'яті,
                               ; адреса якої знаходиться як застосування значень регістрів EBX,
                                                                    ESI ; та константи ABCDF
```

Такий спосіб адресації зазвичай використовується для адресації елементів у двовимірних масивах даних або для доступу до окремих елементів у масивах, що містять записи. Проаналізуємо фрагмент програмного коду, в якому застосовується цей спосіб адресації:

```
s1   DB "ABCD EFGH IJKLM"
s2   DB "abed efgh ijklm"
s3   DB "0123 4567 89"
sarray label dword
 DD s1
 DD s2
 DD s3
…
 mov   EBX, sarray+4
 mov   ESI, 10
 mov   AL, byte ptr [EBX][ESI][2]
```

Тут у сегменті даних визначено масив рядків sarray, що містить адреси рядків s1 – s3. У кожному рядку визначено групи елементів, розділені символом пропуску. Припустимо, потрібно дістати доступ до символу *k*, що знаходиться в рядку s2. Використаємо регістр EBX як базовий, а регістр ESI як індексний. Помістимо на EBX адресу рядка s2, де знаходиться шуканий елемент (команда mov EBX. sarray+4), а в регістр ESI – зсув групи елементів, в якій знаходиться символ *k* (величина зсуву в даному випадку дорівнює 10). Для цього виконаємо команду

```
mov ESI, 10      ; завантаження в ESI константи 10
```

Символ *k* знаходиться за зсувом 2 щодо групи елементів ijklm, тому остання команда поміщає символ в регістр AL:

```
mov AL, byte ptr [EBX][ESI][2]
```

11. ***Масштабована базово-індексна адресація зі зміщенням*** (Based Scaled Index Mode with Displacement).

Адреса **EA = Base + Scale × Index + Disp**. Наприклад,

```
MOV ECX, [EBX + 36] + [8*ESI]   ; завантаження в ECX змінної з пам'яті,
                ; адреса якої знаходиться як застосування значення регістра EBX,
        ; збільшеного на 36, та значення регістра ESI, збільшеного в вісім разів
```

Множник зазвичай використовується для доступу до елементів, що мають розмір, кратний 2, наприклад до слів, подвійних слів і т.д. Зсув при такому способі адресації указує на початок масиву чисел або рядка, а індексний регістр містить номер елемента даних. Далі показаний приклад, який демонструє цей спосіб адресації:

```
    …
  S1  DB "0123456789ABCDEF"    ; резервування у пам'яті комірок
      ; розміром у байт для збереження кожного числа змінної S1
    mov   EBX, 7                          ; занесення в EBX константи 7
    mov   AL, byte ptr  S1[EBX*2]          ; занесення в AL числа
       ; з пам'яті з масиву S1, адреса якого знаходиться в EBX*2
```

При вказаному значенні регістра EBX в регістр AL буде поміщений символ E, оскільки він знаходиться за зсувом 14 (7 × 2) у масиві S1.

Це найповніша схема адресації, в яку входять всі випадки, розглянуті раніше як окремі.

Повну адресу операнда можна записати як вираз, наведений на рис. 2.5.

Прочерк в колонці «індекс» позначує, що ESP не можна використовувати як індексний. Однак можлива конструкція

```
mov EAX,[ECX][ESP]
```

Неприпустимо помножувати ESP на масштабуючий коефіцієнт:

```
mov EAX,[ECX][ESP*8] ; команда не припустима
```

Зсув може бути байтом або подвійним словом. Якщо ESP або EBP використовуються як базовий регістр, селектор сегмента операнда береться за умовчанням із регістра SS, у решті випадків – з DS.

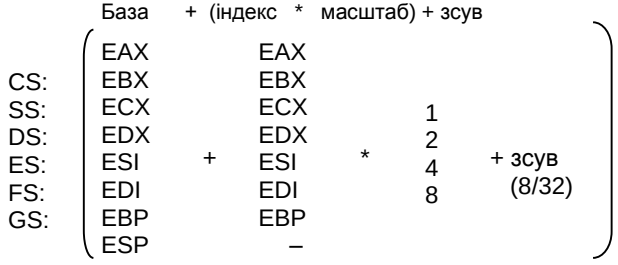


Рис. 2.5. Повна схема адресації 32-розрядного МП

Масштабування індексу можливе тільки при 32-розрядній адресації.

Процесор може працювати з 32- чи 16-бітовою адресацією. 16-бітова адресація працює так само, як і в мікропроцесорах 8086 і 80286, при цьому як компоненти адреси використовуються молодші 16 бітів відповідних регістрів. При 32-бітовій адресації використовуються розширені 32-розрядні регістри і застосовуються додаткові режими з масштабуванням індексу.

Приклад використання безпосередньої адресації наведено у лістингу 2.2.

Лістинг 2.2. Програма з використанням безпосередньої адресації:

```
title l2 (3асоби адресації_1)  masm32
        ; компиляция – ml.exe /c /coff   A1.asm
        ; компоновка – link.exe /SUBSYSTEM:CONSOLE   A1.obj
        ; pause
        ; start  A1.exe
    .386            ; директива определения типа микропроцессора
.model  flat,stdcall  ; задання линейной модели памяти и соглашения ОС Windows
  .data                                   ;  директива определения данных
data_8 db -3 ; число -3 в памяти имеет имя data_8 и размер  8 біт
data_16 dw ? ; неопределенное число (?) в памяти имеет имя data_16 и размер  16 бит
  .code                                   ; директива начала кода программы
    _start:                               ; метка начала программы с именем start
      mov eax,0ffffffffh      ;  занесение в Рг еах операнда ffffffffh
      mov al, data_8           ; занесение в Рг al  операнда -3
      mov data_16, ax          ; сохранение в ячейке памяти Рг ах
       ret                     ; возвращение управления  ОС
       end _start              ; окончание программы с именем _start
```

Ця програма спочатку заносить до EAX 32-розрядну константу ffffffffh. Потім до AL заноситься статична константа з іменем data_8, яка в десятковій системі числення дорівнює –3, а у шістнадцятковій FD.

Команда mov data_16, ax заносить до комірки пам'яті DS:[402001] як молодшу частину з AL, так і старшу з AH. Таким чином, до пам'яті буде занесено слово FF FD.

Для того щоб налагодити програму, спочатку необхідно **створити exe-файл**, наприклад, через файловий менеджер FAR. Для цього потрібно виконати такі дії:

– написати програму у текстовому редакторі та зберегти її з розширенням .asm, наприклад:

masm32\bin\ім'я_програми.**asm**;

– виконати компіляцію програми:

ml.exe /c /coff ім'я_програми.**asm**;

– виконати компонування програми:

link.exe /SUBSYSTEM:CONSOLE   ім'я_програми.**obj**.

Однак для прискорення часу створення exe-файлу рекомендується створити окремий ***файл із розширенням .bat***, в якому потрібно міняти тільки ім'я.

При натисканні клавіші <Enter> автоматично буде створений файл із розширенням .**obj** та, якщо немає помилок, і файл із розширенням .exe. Зміст bat-файлу може бути таким:

ml /c /coff "A1.asm"
link /SUBSYSTEM:CONSOLE "A1.obj"

Після таких дій створюється файл із розширенням .exe, який можна для налагодження завантажити у налагоджувач, наприклад, у OllyDbg. Для цього необхідно виконати команди Файл/Открыть та вказати директорію, де знаходиться необхідний файл із розширенням .exe. Якщо були виконані попередні дії, які рекомендувалися, то такий файл повинен знаходитися таким чином:

masm32\bin\ім'я_програми.exe.

Вікно налагодження програми з лістингу 2.2 налагоджувачем OllyDbg (http://cracklab.ru/_dl/new/ollydbg110xp.rar або www.ollydbg.de/ odbg2.01 alpha 3) наведено на рис. 2.6.

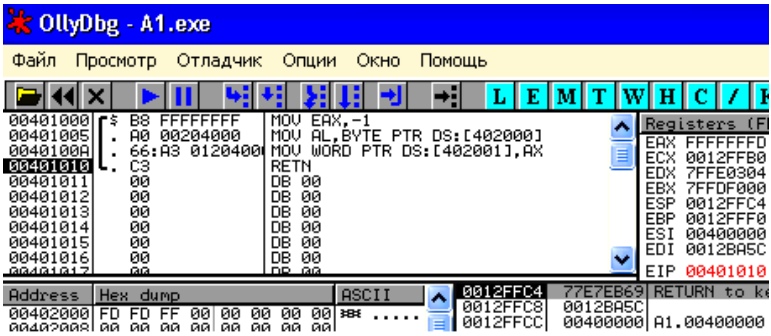


Рис. 2.6. Вікно налагодження програми з лістингу 2.3

Покрокове виконання програми виконується натисканням клавіші F7.

Приклад використання непрямої адресації наведено у лістингу 2.3.

Лістинг 2.3. Програма з використанням непрямої адресації:

```
.386                ; директива определения типа микропроцессора
.model  flat,stdcall  ; задание линейной модели памяти  и соглашения ОС Windows
.code                          ; директива начала кода программы
  _start:                      ; метка начала программы с именем start
    mov eax,11223344h       ; занесение в EAX числа 11223344h
    mov [esp],eax ; занесение из EAX в память (стек) числа,
                           ; адрес которого находиться в ESP
  ret                          ; возвращение управления  ОС
    end _start                 ; окончание программы с именем _start
```

У правому нижньому кутку наведено вершину стека з адресою 0012FFC4 та результатом 11223344.

У програмі лістингу 2.3 як регістр непрямої адресації використовується регістр ESP. Ця програма виконує занесення до EAX шістнадцяткового числа 11223344h та перезапис його до стека за адресою, яка розміщена в регістрі

ESP. Вікно налагодження програми з лістингу 2.3 програмою OllyDbg наведено на рис. 2.7.

2.2. Архітектурні особливості 64-розрядного МП Intel

2.2.1. Угода про виклики параметрів функцій

Двома найважливішими відмінностями між архітектурою x86 та x64 є можливість 64-бітової адресації і набір із 16 64-бітових регістрів загального призначення. x64 використовує тільки угоду про виклики **__fastcall** і модель RISC-архітектури. Модель угоди про виклики __fastcall використовує регістри для перших чотирьох аргументів, а для передання інших параметрів використовується кадр стека.

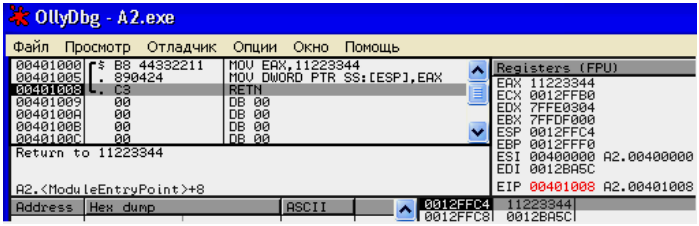


Рис. 2.7. Вікно налагодження програми з лістингу 2.3

Якщо аргументи є типами float або double, то вони передаються в регістрах XMM0L, XMM1L, XMM2L і XMM3L. 16-байтові аргументи передаються посиланням. Регістри RAX, R10, R11, XMM4 і XMM5 є енергозалежними.

Функція відповідає за виділення простору для параметрів функції, що викликається, і повинна завжди виділяти достатній простір для 4 параметрів, навіть якщо функція, що викликається, не містить такої кількості параметрів.

64-розрядний мікропроцесор має «довгий» режим (long mode), який поділяється на два види режимів:

– режим сумісності з x86 (compatibility mode);

– 64-бітовий режим (64-bit mode).

Режим сумісності передбачений Для того щоб 64-розрядна операційна система могла виконувати старі 32-бітові застосування.

2.2.2. Регістри архітектури x64

64-розрядна архітектура ґрунтується на відомих раніше 32-розрядних регістрах, які мають імена RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, RIP та RFLAGS (рис. 2.8).

В архітектурі Intel 64 (AMD64) використовуються такі регістри:

– шістнадцять цілочисельних 64-бітових загального призначення (RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8 – R15);

– вісім 80-бітових із рухомою крапкою (ST0 – ST7);

– вісім 64-бітових Multimedia Extensions (MM0 – MM7, які мають загальний простір із регістрами (FPR0 – FPR 7),

– шістнадцять 128-бітових SSE (XMM0 – XMM15);

– 64-бітовий вказівник RIP та 64-бітовий регістр прапорців RFLAGS.

З'явилися нові регістри R8–R15. Для звернення до молодших 8-, 16- та 32-бітових нових регістрів можна використовувати суфікси b, w та d. Наприклад 8b – молодший байт регістра R8 (за аналогією до AL), а R8w – молодше слово (те саме, що AX в EAX). Адресувати регістр AH неможливо.

Регістр-вказівник RIP указує на наступну, а не поточну інструкцію, а адресується так само, як і решта всіх регістрів загального призначення, наприклад mov al,[rip].

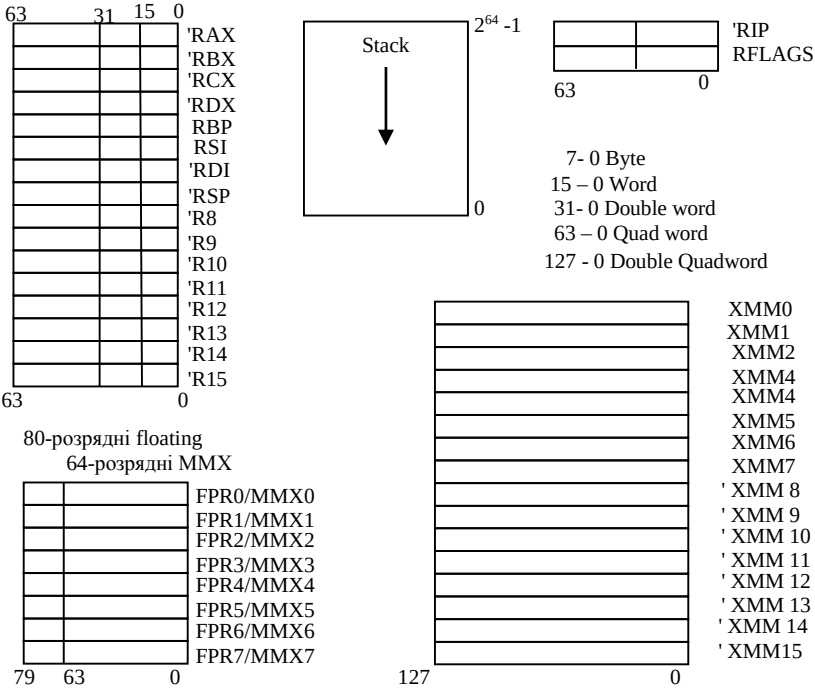


Рис. 2.8. 64-розрядна архітектура МП INTEL

Угода щодо викликів на платформі x86-64 схожа на угоду fastcall, що існує в x86. Параметри API-функцій передаються через регістри, а не через стек. Перші чотири аргументи всіх API-функций передаються в регістрах RCX, RDX, R8 та R9 (регістри перераховані за проходженням аргументів, крайній лівий аргумент поміщається в RCX). Решта параметрів передаються через стек. Все це називається x86-64 fast calling conversion (угода про швидку передачу параметрів для x86-64).

http://blogs.kpi.kharkov.ua/v2/asm/knigi/

Якщо передаються значення з рухомою крапкою, то перші чотири з них передаються в регістрах XMM0-XMM3, а подальші – через стек.

Наприклад, виклик функції з п’ятьма аргументами API_func(1,2,3,4,5) виглядає так:

```
mov  dword ptr [rsp+20h], 5 ; занесення в стек першого справа аргументу
     mov     r9d, 4             ; передавання другого справа аргументу
     mov     r8d, 3             ; передавання третього справа аргументу
     mov     edx, 2             ; передавання четвертого справа аргументу
     mov     ecx, 1             ; передавання першого справа аргументу
     call    API_func
```

Зміщення 20h необхідне для резервування під комірки для неініціалізованого сміття і називається «spill», що перекладаєтьсяся як «втрата». Це зміщення складається з адреси повернення (8 байтів) та 8 + 8 + 4 + 4 для перших чотирьох аргументів, переданих через регістри.

Лістинг 2.4. 64-бітове застосування «Hello, world» під Windows на MASM’е в середовищі Visual Studio 2013.

```
    ExitProcess PROTO
    MessageBoxA PROTO
.data
    mytit db 'Program on x64', 0
    mymsg db 'Hello World!', 0
.code
main proc
        sub rsp, 28h
    mov r9d, 0          ; uType = MB_OK
    lea r8,  mytit      ; LPCSTR lpCaption
    lea rdx, mymsg      ; LPCSTR lpText
    mov rcx, 0          ; hWnd = HWND_DESKTOP
    call MessageBoxA
    mov ecx, eax        ; uExitCode = MessageBox(...)
    call ExitProcess
main endp
end
```

Результат виконання програми наведено на рис. 2.9.

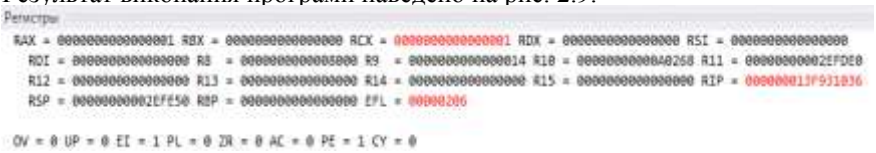


Рис.2.9. Результат виконання програми

2.2.3. Використання регістрів архітектури x64

Архітектура x64 підтримує 16 регістрів загального призначення (надалі називатимуться цілочисловими регістрами), а також 16 регістрів XMM, що ви-

користовуються для операцій з рухомою крапкою. Тимчасові регістри скидаються під час виконання виклику. Постійні регістри повинні зберігати свої значення в процесі виконання функції і мають зберігатися об'єктом, що викликається, у випадку використання.

У табл. 2.7 наведено способи використання регістрів під час виконання виклику функції.

Таблиця 2.7

Регістри	Стан	Застосування
RAX	Тимчасовий	Регістр поверненого значення
RCX	Тимчасовий	Перший цілочисловий аргумент
RDX	Тимчасовий	Другий цілочисловий аргумент
R8	Тимчасовий	Третій цілочисловий аргумент
R9	Тимчасовий	Четвертий цілочисловий аргумент
R10:R11	Тимчасовий	Повинен зберігатися об'єктом. Використовується в інструкціях syscall/sysret
R12:R15	Постійний	Повинен зберігатися об'єктом, що викликається
RDI	Постійний	Повинен зберігатися об'єктом, що викликається
RSI	Постійний	Повинен зберігатися об'єктом, що викликається
RBX	Постійний	Повинен зберігатися об'єктом, що викликається
RBP	Постійний	Може використовуватися як вказівник кадру. Повинен зберігатися об'єктом, що викликається
RSP	Постійний	Вказівник стека
XMM0	Тимчасовий	Перший аргумент із рухомою крапкою
XMM1	Тимчасовий	Другий аргумент із рухомою крапкою
XMM2	Тимчасовий	Третій аргумент із рухомою крапкою
XMM3	Тимчасовий	Четвертий аргумент із рухомою крапкою
XMM4:XMM5	Тимчасовий	Повинен зберігатися об'єктом
XMM6:XMM15	Постійний	Повинен зберігатися об'єктом, що викликається

Регістри RAX, RCX, RDX, R8, R9, R10 та R11 вважаються тимчасовими і мають знищуватися при виклику функції (якщо інше не потрібне, виходячи з міркувань безпеки, наприклад у процесі оптимізації програми).

Регістри RBX, RBP, RDI, RSI, RSP, R12, R13, R14 та R15 вважаються захищеними. Значення цих регістрів мають зберігатися і відновлюватися у функції, що використовує їх.

2.2.4. Скалярні типи даних архітектури x64

Не зважаючи на те що звернення до даних можливо при будь-якому вирівнюванні, з мемтою підвищення продуктивності рекомендується використовувати вирівнювання даних у початковому діапазоні. Перерахування є константами цілого типу і обробляються як 32-розрядні цілі числа. У табл. 2.9 наводиться визначення типу та обсяг пам'яті, що рекомендується для вирівнювання з використанням таких значень:

Byte – 8 бітів;
Word – 16 бітів;
Double Word – 32 біти;
Quad Word – 64 біти;
Octa Word – 128 бітів.

2.2.5. Використання стека архітектури x64

Всі адреси пам'яті, що знаходяться поза поточною адресою RSP, вважаються тимчасовими. Під час дії оброблювача переривань або виконання призначеного для користувача сеансу налагоджування ці адреси можуть перезаписуватися за допомогою функцій операційної системи або налагоджувача. Таким чином, перед читанням значень із кадру стека або записом до нього необхідно задати RSP. Визначення типу та обсяг пам'яті наведено в табл. 2.8.

Таблиця 2.8

Скалярний тип	Тип даних *C*	Обсяг пам'яті (у байтах)	Вирівнювання
INT8	char	1	Byte
UINT8	unsigned char	1	Byte
INT16	short	2	Word
UINT16	unsigned short	2	Word
INT32	int, long	4	Doubleword
UINT32	unsigned int, unsigned long	4	Doubleword
INT64	__int64	8	Quadword
UINT64	unsigned __int64	8	Quadword
FP32 (одиночної точності)	float	4	Doubleword
FP64 (подвійної точності)	double	8	Quadword
pointer	*	8	Quadword
__m64	struct __m64	8	Quadword
__m128	struct __m128	16	Octaword

Виділення пам'яті в стеку

Виділення пам'яті в стеку для локальних змінних, збережених регістрів, а також параметрів стека і регістра здійснюється в пролозі функції.

Ділянка параметрів завжди розташовується в нижній частині стека (навіть якщо використовується функція alloca) і завжди примикає до адреси, яка повертається при виклику функції. У ділянці параметрів міститься як мінімум чотири записи і завжди виділяється обсяг пам'яті, достатній для зберігання всіх параметрів будь-якої функції, що викликається.

Для параметрів регістра пам'ять виділяється завжди, навіть якщо вони ніколи не розміщуються в стеку. Також гарантовано виділяється пам'ять для всіх параметрів об'єкта, що викликається. Щоб забезпечити виділення безперервної ділянки пам'яті при виклику функції, що приймає адресу списку аргументів (va_list) або окремого аргументу, необхідно надати внутрішні адреси аргументів регістра. У цій ділянці також зберігаються аргументи регістра під час виконання перетворювань. Крім того, в ній зручно зберігати аргументи регістра, що використовуються як параметри налагоджування (наприклад пошук аргументів у процесі налагоджування є ефективнішим, якщо вони зберігаються за внутрішньою адресою в коді прологу). Навіть якщо функція, що викликається, набуває менше чотирьох параметрів, для неї виділяється чотири комірки стека, які можуть використовуватися не тільки для зберігання значень регістра параметрів, але і для інших цілей. Таким чином, під час виклику функції об'єкт не може зберігати дані в цій ділянці стека.

Якщо пам'ять для функції виділяється динамічно (за допомогою функції alloca), необхідно використовувати захищений регістр як вказівник кадру. За допомогою захищеного регістра позначається основа фіксованої частини стека, а також вказується необхідність збереження та ініціалізації регістра в пролозі. При використанні функції alloca (під час виклику одного і того самого об'єкта) для параметрів регістра можуть використовуватися різні внутрішні адреси.

У стеку завжди використовується 16-байтове вирівнювання, за винятком прологу (наприклад при розміщенні зворотної адреси) та інших випадків, визначених для конкретних класів або функцій кадру.

Динамічне створення параметра ділянки стека в цей час у компіляторі x64 не виконується.

Типи функцій

Існує два основні типи функцій. Функція, якій потрібний кадр стека, називається функцією з кадром. Функція, яка не потребує кадру стека, називається кінцевою функцією.

Функція з кадром – це функція, яка розподіляє простір у стеку, викликає інші функції, зберігає захищені регістри або використовує оброблювач виключень, а також потребує наявності запису в таблиці функції. Функція з кадром вимагає наявність прологу і епілогу. Функція з кадром потребує динамічно розподіляти простір у стеку і використовувати вказівник кадру. Функція з кадром має всі необхідні характеристики для використання виклику цього стандарту.

Якщо функція з кадром не викликає іншу функцію, то вирівнювати стек не потрібно.

Кінцева функція є єдиною функцією, яка не потребує наявності запису в таблиці функції. Вона не викликає інші функції, не розподіляє простір і не зберігає захищені регістри. Функції дозволено не вирівнювати стек під час свого виконання.

Функція malloc використовується для повернення пам'яті по 16-байтовій межі.

Функції _alloca потрібне вирівнювання по 16-байтовій межі і використання вказівника кадру стека.

Контрольні запитання

1. Що розуміють під поняттям архітектури мікропроцесорних систем?
2. Які типи даних використовуються у 32-розрядних мікропроцесорах?
3. У чому полягає призначення загальних регістрів 32-розрядного мікропроцесора?
4. У чому полягає призначення бітів регістра прапорців?
5. Для яких цілей використовуються регістри FPU, MMX, SSE?
6. У чому полягає організація пам'яті у 32-розрядних мікропроцесорах?
7. Яка модель пам'яті використовується для 32-розрядних мікропроцесорів?

3. ОСНОВНІ ДИРЕКТИВИ ТА ОПЕРАНДИ АСЕМБЛЕРА

3.1. Директиви асемблера

Директиви призначені для керування процесом асемблювання і формування лістингу. Вони діють тільки в процесі асемблювання програми і не переводяться в машинні коди. Мова асемблера містить такі основні директиви:

- керування моделями пам'яті MODEL та сегментами DATA, CONST, STACK, CODE;
- початку і закінчення сегмента SEGMENT та ENDS;
- початку і закінчення процедури PROC та ENDP;
- визначення пам'яті та призначення сегментів ASSUME;
- початку ORG;
- визначення даних DB, DW, DD, DF, DP, DQ, DT;
- символьних констант: = , EQU, TEXTEQU;
- завершення програми END;
- відзначення LABEL;
- вирівнювання адреси ALIGN;
- локальності ідентифікаторів LOCALS.

3.1.1. Директиви MODEL, DATA, CONST, STACK, CODE

Директива .**MODEL** *модель_пам'яті [,угода_про_виклики] [,тип_ОС]* [*,параметр_стека*] визначає модель пам'яті, що використовується програмою. Ця директива повинна знаходитися перед будь-якою з директив оголошення сегментів. Вона зв'язує певним чином різні сегменти програми, що визначаються її параметрами tiny, small, compact, medium, large, huge або flat. Параметр *модель_пам'яті* є обов'язковим.

Для включення в програму процедури, написаної мовою високого рівня, необхідно вказати модель пам'яті, яка використовується компілятором мови високого рівня. Крім того, модель пам'яті повинна відповідати режиму роботи (типу) мікропроцесора. Це має значення для плоскої моделі пам'яті, яку можна застосовувати тільки в режимах . 386, .486, .586, .686. Модель пам'яті визначає, який тип адресації даних і команд підтримує програма (near або far). Це має сенс для команд переходу, викликів і повернення з процедур. У табл. 3.1 наведено ці особливості.

- Модель tiny – коди і дані разом повинні займати не більше 64 кбайт, працює тільки в 16-розрядних застосуваннях MS-DOS. У цій моделі всі дані і код розташовуються в одному фізичному сегменті. Розмір програмного файлу в цьому випадку не перевищує 64 кбайт. З іншого боку, модель flat припускає несегментовану конфігурацію програми і використовується тільки в 32-розрядних операційних системах. Ця модель подібна до моделі tiny в тому сенсі, що дані і код розміщені в одному сегменті, тільки 32-розрядному.

Таблиця 3.1

Модель пам'яті	Адресація коду	Адресація даних	Операційна система
Tiny	Near	Near	MS-DOS
Small	Near	Near	MS-DOS, Windows
Medium	Far	Near	MS-DOS, Windows
Compact	Near	Far	MS-DOS, Windows
Large	Far	Far	MS-DOS, Windows
Huge	Far	Far	MS-DOS, Windows
Flat	Near	Near	Windows NT/2000/XP/2003

- Модель small – code ≤ 64 кбайт, data ≤ 64 кбайт. Один сегмент коду, один сегмент даних, дані і код при використанні цієї моделі адресуються як near (близькі).
- Модель large – code > 64 кбайт, data > 64 кбайт – підтримує декілька сегментів кодів і декілька сегментів даних. За умовчанням всі посилання на код і дані вважаються далекими (far).
- Модель medium – data ≤ 64 кбайт, код будь-якого розміру. Багато сегментів коду, один сегмент даних, при цьому всі посилання в сегментах програмного коду за умовчанням вважаються далекими (far), а посилання в сегменті даних – близькими (near).
- Модель compact – code ≤ 64 кбайт, дані будь-якого розміру. Один сегмент коду, багато сегментів даних, в яких використовується далека адресація даних (far), і один сегмент коду з близькою адресацією (near).
- Модель huge – code > 64 кбайт, data > 64 кбайт. Ця модель практично еквівалентна моделі пам'яті large.

Для розробки програми для моделі flat перед директивою .model flat слід розмістити одну з таких директив: .386, .486, .586 або .686. При цьому адресація даних є близькою (near), а всі адреси і вказівники є 32-розрядними. Сегменти відсутні. 32-розрядна адресація використовується як для коду, так і для даних.

При використанні всіх моделей пам'яті, окрім tiny, створюються виконувані програми типу .exe, а при використанні моделі пам'яті tiny – програми .com. Всі моделі, виключаючи flat, розраховані на використання в реальному режимі мікропроцесора. Модель flat може використовуватися тільки для роботи в захищеному режимі для створення застосувань Windows.

Параметр *угода_про_виклики* використовується для визначення способу передання параметрів при виклику процедури з інших мов, у тому числі і з мов високого рівня (C++, Pascal). Параметр може набувати таких значень: C++, BASIC, FORTRAN, PASCAL, SYSCALL, STDCALL.

Параметр *тип ОС* дорівнює OS DOS, і на даний момент це єдине підтримуване значення цього параметра.

У прикладі

.model flat, C

параметр C указує на те, що при виклику асемблерної процедури з іншої програми (можливо, написаною іншою мовою) буде задіяний спосіб передання параметрів, прийнятий у мові C.

У прикладі

.model flat, stdcall

параметр stdcall указує на те, що застосовується спосіб передання параметрів при використанні процедур справа наліво.

Після того як модель пам'яті встановлено, набувають чинності ***спрощені директиви визначення сегментів***, що об'єднують дії директив SEGMENT і ASSUME. Крім того, сегменти, оголошені спрощеними директивами, не потрібно закривати директивою ENDS – вони закриваються автоматично, як тільки асемблер виявляє нову директиву визначення сегмента або закінчення програми.

Директива **.DATA** визначає початок сегмента ініційованих даних з ім'ям DATA. Цій директиві повинна передувати директива .MODEL, яка вказує модель пам'яті, що використовується програмою.

Директива **.DATA?** визначає сегмент даних, в якому розташовуються неініційовані дані. За наявності попереднього сегмента новий сегмент завершує його. Неініційовані дані можуть оголошуватися в сегменті .DATA? за допомогою знака оператора «?» Цей сегмент зазвичай не включається в програму, а розташовується за кінцем пам'яті, тому всі описані в ньому змінні на момент завантаження програми мають невизначені значення. Перевагою директиви .DATA? є те, що при її використанні зменшується розмір виконуваного файлу і забезпечується краща сумісність з іншими мовами. Цій директиві має передувати директива .MODEL.

Директива **.CONST** визначає початок сегмента незмінних даних:

CONST segment word public 'CONST'

За наявності попереднього сегмента новий сегмент завершує його. Сегмент, визначений директивою .CONST, має атрибут «тільки для читання». Цій директиві має передувати директива .MODEL.

Директива **.STACK** [*розмір*] визначає початок сегмента стека з вказаним *розміром* пам'яті, який повинен бути виділений під ділянку стека:

STACK segment para public 'stack'

Якщо параметр не вказаний, розмір стека передбачається таким, що дорівнює 1 кбайт.

Директива **.CODE** [*ім'я*] визначає сегмент програмного коду і закінчує попередній сегмент. Цій директиві повинна передувати директива .MODEL, яка вказує модель пам'яті, використовувану програмою.

3.1.2. Директиви **SEGMENT** *та* **ENDS**

Директиви початку* SEGMENT *і кінця* ENDS *сегмента призначені для опису сегментів, які використовує програма. Директиви початку і кінця сегмента використовуються разом. Наприклад:

```
Назва       Мнемокод        Операнд
DATASG   SEGMENT      {<параметри>}
            . . .     ; Інші команди або директиви сегмента
DATASG   ENDS
```

Обидві директиви SEGMENT і ENDS повинні мати однакові назви. Директива SEGMENT може містити три типи параметрів:

- вирівнювання;
- об'єднання;
- класу.

Параметр ***вирівнювання*** визначає початкову адресу або *межу* сегмента, наприклад:

BYTE – з різної адреси – байт,
WORD – з парної адреси – слово,
DWORD – з адреси, кратної 4 – подвійне слово,
QWORD – слово, збільшене у чотири рази (64 розряди = 8 байтів),
TWORD – слово розміром 80 розрядів,
PARA – межа за умовчанням – з адреси, кратної 16,
PAGE – з адреси, кратної 256.

Якщо немає параметра вирівнювання за умовчанням, береться параметр *PARA*, який вказує на те, що сегмент розташовується на початку параграфа, а початкова адреса сегмента є кратною 16. ***Параграфом*** називається область пам'яті розміром 16 байт, початкова адреса якої кратна 16, тобто має праворуч чотири нульових розряди.

Параметр ***об'єднання*** вказує на спосіб обробки сегмента при компонуванні декількох програмних модулів:

- NONE: значення за умовчанням. Сегмент має бути логічно відокремленим від інших сегментів, хоча фізично він може розташовуватися поряд. Передбачається, що сегмент має власну базову адресу;
- PUBLIC: усі PUBLIC – сегменти з однаковими назвою та класом завантажуються у суміжні ділянки і мають одну базову адресу;
- STACK: призначення аналогічне параметру PUBLIC. У будь-якій програмі має бути визначений принаймні один сегмент STACK. Якщо визначено більше одного стека, то вказівник стека SP (Stack Pointer) встановлюється на початок першого стека;
- COMMON: для сегментів COMMON з однаковими назвами та класами встановлюється одна спільна базова адреса. Під час виконання програми здійснюється накладання другого сегмента на перший. Розмір загальної ділянки визначається найдовшим сегментом;
- AT: параграф, який слід визначати заздалегідь. Цей параметр забезпечує визначення міток та змінних за фіксованими адресами у фіксованих ділянках пам'яті;

- ‘Клас’: цей параметр може мати будь-яку правильну назву, яка розміщується в одинарних лапках. Параметр використовується для обробки сегментів, які мають однакові назви та класи. Типовими є класи ‘STACK’ та ‘CODE’, наприклад:

Назва	Мнемокод	Операнд
STACKSG	SEGMENT	PARA STACK ‘STACK’

У випадку, якщо програма не повинна об’єднуватися з іншими програмами, параметр об’єднання не вказується.

3.1.3. Директиви **PROC** *та* **ENDP**

Директиви початку і кінця процедури **PROC** *та* **ENDP** використовуються для визначення підпрограм у сегменті кодів і мають такий формат:

<Назва> PROC {<тип процедури>}.

Можливі два типи процедур:

- NEAR – процедура, яка знаходиться в тому самому сегменті, що і команди, які її викликають;
- FAR – процедура, яка знаходиться за межами сегмента.

За умовчанням береться тип процедури NEAR.

Сегмент кодів може містити декілька процедур. Опис сегмента кодів, що містить тільки одну процедуру, має такий вигляд:

Назва	Мнемокод	Операнд
Ім’я сегмента	SEGMENT	PARA
Ім’я_процедури	PROC	FAR
		RET
Ім’я_процедури	ENDP	
Ім’я сегмента	ENDS	

Ім’я процедури повинно бути визначене обов’язково і має збігатися з іменем у директиві ENDP, яка визначає закінчення процедури.

3.1.4. Директива **ASSUME**

Директива призначення сегментів **ASSUME** використовується для встановлення відповідності між сегментами та сегментними регістрами і має такий формат:

ASSUME <сегментний регістр>:<ім’я>{,}{...}.

Наприклад, запис SS:ім_стек показує, що ім’я стека визначається вмістом регістра SS. Одна директива ASSUME може призначати до чотирьох сегментних регістрів у будь-якій послідовності, наприклад:

Мнемокод Операнд(и)

ASSUME SS: ім_стек, DS: ім_дані, CS: ім_код, ES:ім_додаткові дані

Для скасування будь-якого призначеного раніше в директиві ASSUME сегментного регістра потрібно використовувати слово NOTHING:

Мнемокод Операнд(и) ASSUME

ES: NOTHING.

Якщо програма не використовує якийсь сегмент, то відповідний до нього операнд можна відпустити або вказати слово NOTHING.

3.1.5. Директива ORG

***Директива* ORG** використовується для зміни вмісту програмного лічильника без команд умовного чи безумовного переходу. Найчастіше цю директиву використовують для встановлення початкової адреси програми. Наприклад, директива ORG 100H встановлює програмний лічильник на зміщення 100H відносно початку сегмента кодів. Операнд зі знаком долара "$" має поточне значення програмного лічильника. Наприклад, директива

ORG $ + 10h

збільшує адресу, завантажену в програмний лічильник, на 10h.

3.1.6. Директиви визначення даних DB, DW, DD, DF, DP, DQ, DT

Директиви визначення даних використовуються для визначення вмісту та резервування комірок пам'яті.

Директива має формат

[<ім'я>] Dn [кількість повторень] <вираз>,

де мнемокод Dn = DB, DW, DD, DF, DP, DQ, DT вказує на довжину даних: DB – байт; DW – слово (два байти); DD – подвійне слово; DF, DP – шість байтів, DQ – чотири слова; DT – десять байтів. Якщо у форматі наявне ім'я, то далі в програмі воно може використовуватися для позначення комірки пам'яті.

<Вираз> у форматі директиви містить одну або декілька констант – для задавання початкових значень вмісту комірок пам'яті або знак «?» – для невизначеного значення вмісту. Наприклад, директива

ALPHA DB 34

означає, що 8-розрядна комірка пам'яті з іменем ALPHA містить число 34. У процесі виконання програми вміст комірки може бути змінений. Директива

BETA DW ?

означає, що комірка з іменем BETA має розрядність 16, але вміст комірки є невизначеним. Директива може містити декілька констант, розділених комами і обмежених лише довжиною рядка. Наприклад, вираз

ARRAY DB 01, 02, 11, 12, 21, 22

визначає 6 констант у вигляді послідовності суміжних байтів.

Посилання на комірку з іменем ARRAY вказує на першу константу (01), з іменем ARRAY + 1 – на другу (02), з іменем ARRAY + 2 – на третю (11) і т. д. Запис MOV AL, ARRA + 4 завантажує у регістр AL значення 21.

Одна директива може визначити декілька комірок пам'яті. У цьому випадку директива має вигляд

{ <ім'я> } Dn {кількість повторень} DUP <вираз>

Наприклад, директива, що означає 5 байт, які містять число 21, записується таким чином: DB 5 DUP(21)

Приклади директив визначення даних можуть бути такими:

```
db1 DB 5      ; запис у 8-розрядну комірку пам'яті з ім'ям db1 числа 5
db2 DB 'ABCD'   ; запис у пам'ять з ім'ям db2  8-розрядних літер
db3 DB ?      ; резервування в пам'яті з ім'ям db3 даних, які не визначені
dw1 DW 1234    ; запис у 16-розрядну комірку пам'яті з ім'ям dw1 числа 1234
dw2 DW 5*6       ; запис у 16-розрядну комірку пам'яті з ім'ям dw2 числа 30
    dw3 DW 1. ' $'
    dw4 DW array
    dd1 DD  2.7              ; число займає 4 байти в пам'яті
    dd2 DD 'abcd' ; запис у пам'ять з ім'ям dd2  32-розрядних літер
    dql  DQ 123456789    ; число з ім'ям  dql   займає 8 байтів в пам'яті
    mixl DB 5 DUP(5 DUP(5 DUP(10)))
    dt1 DT 56789     ; запис у 80-розрядну комірку пам'яті з ім'ям
                                        ; dt1 числа 56789,  число займає в пам'яті 10 байтів
```

Директиви визначення даних можуть використовуватися як директиви завдання типу. Базові типи, які можна використовувати для визначення даних, наведено у табл. 3.2.

Таблиця 3.2

Тип	Значення
BYTE	8-розрядне ціле число без знака
SBYTE	8-розрядне ціле число зі знаком
WORD	16-розрядне ціле число без знака
SWORD	16-розрядне ціле число зі знаком
DWORD	32-розрядне ціле число без знака
SDWORD	32-розрядне ціле число зі знаком
FWORD	48-розрядне ціле число (використовується як указівник)
QWORD	64-розрядне ціле число
TBYTE	80-розрядне ціле число
REAL4	32-розрядне коротке дійсне число
REAL8	64-розрядне дійсне число
REAL10	80-розрядне розширене дійсне число

Приклади:

1. Записати в байт пам'яті десяткове число 23 і присвоїти цьому байту ім'я a:
   a db 23.
2. Зарезервувати 1 байт пам'яті:
   db ?

3. Записати в слово пам'яті шістнадцяткове число 1234:
dw 1234h.

3.1.7. Директиви символьних констант: =, EQU, TEXTEQU

Директива знака рівності створює константу, присвоюючи імені числове значення *ім'я = вираз*

На відміну від директив DB і DW, директива знака рівності не розподіляє пам'ять. Під час трансляції програми всі відповідні імена будуть замінені виразами. В результаті обчислення виразу має вийти 32-розрядне число зі знаком або без нього (для мікропроцесорів 386 і вище), як показано в прикладі:

```
prod = 10 * 5        ; обчислення виразу
maxInt = 7FFFh    ; макс. значення 16-розрядного числа зі знаком
minInt = 80000h   ; мінім. значення 16-розрядного числа зі знаком
maxUInt = 0FFFFh             ; макс. значення 16-розряд. числа без знака.
```

Директива EQU присвоює символічне ім'я рядку символів або цифровій константі. Це полегшує читання програм і дозволяє змінити численні входження констант в одному місці. Але є одне важливе обмеження директиви EQU: визначене один раз ім'я не можна перевизначити.

Вирази, що включають цілочислові значення, визначають цифрову константу, але вирази з дійсними числами інтерпретуються як рядок символів. Рядки символів можуть бути поміщені в кутові дужки (<...>) для підтвердження того, що це рядок символів. Це виключає неоднозначність при визначенні імен:

```
    Приклад                    Тип значення
maxint   equ  32767     ;    Цифровий
maxuint  equ  0FFFFh    ;    Цифровий
coun     equ  10 * 20   ;    Цифровий
floatl   equ  <2,345>   ;    Рядок символів
len      equ  $-string1
```

Запис len equ $-string1 дозволяє обчислити кількість байтів, яку займає у пам'яті змінна з ім'ям string1, та присвоїти цю кількість змінній len.

За допомогою директиви **TEXTEQU** можна присвоїти рядку символів ім'я і потім використовувати рядок під цим ім'ям. Синтаксис присвоювання такий:

```
ім'я TEXTEQU <текст>
ім'я TEXTEQU макровизначення тексту
ім'я TEXTEQU % константний вираз
```

У даному випадку текст (*текст*) є послідовністю будь-яких символів, поміщених у кутові дужки. Макровизначення тексту являє собою заздалегідь певний текст, а константний вираз перетворюється в текст. Макровизначення тексту можуть створюватися в будь-якому місці програми, а ім'я, визначене за

допомогою TEXTEQU, може бути пізніше перевизначене.

Символічне ім'я може бути присвоєне рядку символів, який надалі буде завжди замінюватися рядком символів. Наприклад, змінна *promptl* визначає макровизначення тексту continueMsg:

```
continueMsg text equ  <"Do you wish to continue (Y/N)?">
.data                                   ; директива определения данных
promptl db continueMsg
```

3.1.8. Директива **END**

Директива завершення програми **END** є останньою у програмі і має такий формат: END {<стартова адреса>}.

Параметр директиви <стартова адреса> використовується лише при створенні EXE-файлів.

3.1.9. Директива **LABEL**

Директива відзначення **LABEL** призначена для встановлення відповідності між іменем і типом змінних та не займає у пам'яті ніяких комірок пам'яті. Вона має такий формат:

<ім'я> LABEL {<*тип*>}.

Як *тип* можна використовувати слова BYTE, WORD, DWORD, що визначають довжину даних: байт, слово або подвійне слово. Директива LABEL перевизначає параметри процедур NEAR або FAR. Наприклад, директива

```
TOS    LABEL   WORD
```

присвоює комірці пам'яті ім'я TOS і зазначає, що її вміст є словом. Вона не займає у пам'яті ніяких комірок пам'яті.

3.1.10. Директива **ALIGN**

Директива **ALIGN** використовується для вирівнювання адреси змінною на межу байта, слова, подвійного слова, збільшеного учетверо слова або параграфа (тобто 16-ти байтів). Вона має такий формат:

ALIGN *межа*

Тут замість параметра *межа* слід підставити число 1, 2, 4, 8 або 16. Якщо значення параметра дорівнює 1, то змінна, що йде за цією директивою, вирівнюється на межу 1-го байта (тобто не вирівнюється зовсім). Це значення прийняте за умовчанням. Якщо значення параметра дорівнює 2, то змінна, що йде за директивою ALIGN, вирівнюється на межу слова (тобто розташовується з парної адреси). Якщо значення параметра дорівнює 4, то наступна змінна вирівнюється на межу подвійного слова (тобто її адреса ділиться на 4) і так далі. За необхідності асемблер автоматично пропускає після директиви ALIGN необхідну кількість байтів, щоб розташувати змінну за потрібною адресою. Цю

директиву використовують для підвищення швидкості обробки даних, які розташовані в пам'яті. Наприклад, якщо адреса подвійного слова кратна 4, тобто вирівняна на межу подвійного слова, то доступ до нього здійснюється за 1 машинний цикл, а якщо немає, то за 2.

3.1.11. Директива LOCALS

***Директива* LOCALS** оголошує асемблера, що ідентифікатори, які мають у двох перших символах вказаний *префікс,* є локальними, тобто діють у межах тільки цієї процедури. Вона має такий формат:

LOCALS {*префікс*}

За умовчанням префікс є парою символів «комерційне at» (@@).

Ця директива дозволяє використовувати той самий ідентифікатор у різних процедурах.

Приклад

```
LOCALS
addW   proc            ; процедура  addW
    jmp    @@go        ; мітка належить до addW
@@go:  …
    ret
    endp
addD   proc            ; процедура   addD
    jmp    @@go        ; ця мітка належить до addD
@@go:  ...
    ret
    endp
```

У випадку, якщо зустрічається команда з унікальною міткою @@, то це означає, що до неї звертається у циклі команда, яка розташовується після неї в тексті та має символ із літерою @b.

3.1.12. Директиви керування файлами INCLUDE, INCLUDELIB

INCLUDE ім'я_файлу – директива, яка використовується для включення файлів, що містять визначення констант, структур і макросів. Використовується аналогічно команді препроцесора C #include.

INCLUDELIB ім'я_файлу – директива, яка вказує компонувальнику ім'я додаткової бібліотеки або об'єктного файлу, який буде потрібен при складанні даної програми. Наприклад, якщо використовуються виклики процедур або звернення до даних, визначених в інших модулях. Використання цієї директиви дозволяє не указувати імена додаткових бібліотек при виклику компонувальника.

3.2. Оператори мови асемблера

- **OFFSET** – оператор, який заносить (повертає) відстань змінної від початку сегмента в приймач або, іншими словами, передає адресу змінної в приймач інформації. Оператор OFFSET функціонує аналогічно команді lea.

```
MOV EBX,OFFSET count ; у EBX заноситься адреса змінною count (EBX := [count])
```

У наступному прикладі зсув адреси кожного елемента масиву переміщається у індексний і базовий регістри. Адреси початку масивів обчислює транслятор, а адреси зміщень в масивах можна задати через відповідні команди.

```
    .data                          ; директива определения данных
bmas  db 10h,20h,30h,40h           ; масив чисел розміром у байт кожен
wmas dw 1000h,2000h,3000h          ; масив чисел розміром у слово  кожен
    .code                          ; директива начала сегмента команд
        mov edi,offset bMas        ; EDI = [bMas]
        mov ebx,offset bMas+1      ; EBX = ]bMas+1]
        mov esi,offset wMas+2      ; ESI = [wMas+2]
```

- **PTR** – оператор, який примусово встановлює розмір операнда.

PTR може використовуватися у поєднанні зі стандартними типами даних: BYTE, WORD, DWORD, QWORD та TBYTE.

Оператор PTR також корисний, коли необхідно змінити розміри операнда, встановлені за умовчанням.

- **SEG** – оператор, який передає (повертає) сегментну частину сегментного регістра в приймач інформації. Оператор SEG застосовується, коли змінна знаходиться в сегменті, відмінному від того, на який указує регістр DS.

```
mov ebx,seg ds ; поточне значення сегмента DS розміщується в ebx
```

- **TYPE** – оператор, який записує (повертає) у приймач інформації байт, в якому визначається розмір змінної. Результуючий біт розрізняє мітку і змінну в програмі, визначає дані, внутрішній і зовнішній ідентифікатори. Наприклад, оператор TYPE для змінної типу слово повертає значення 2.

Лістинг 3.1. Використання оператора TYPE:

```
        ; masm 32: компиляция – ml.exe /c /coff   1_1L.asm
                    ; компонування – link.exe /SUBSYSTEM:CONSOLE  1_1L.obj
    .386                        ; директива определения типа микропроцессора
    .model flat                   ; задання линейной модели памяти
    option casemap:none           ; отличие строчных и прописных букв
.data                                          ; директива определения данных
    var1 DB 2               ; var1 := 02 размером в байт
    var2 DW 1000h           ; var2 := 1000h размером в слово (2 байта)
    var3 DD ?               ; var3 := неопределенные данные размером в 4 байта
    var4 DQ ?               ; var4 := неопределенные данные размером в 8 байтов
    var5 DT ?               ; var5 := неопределенные данные размером в 10 байтов
    var6 DB 6,7,8,9,10        ; каждое число размещается в байте
    msg DB 'hello', 0 ;
.code                       ; директива начала сегмента команд
_start:                     ; метка начала программы с именем  _start
```

```
MOV AL,TYPE var1          ; AL = 01
MOV AL,TYPE var2          ; AL = 02
MOV AL,TYPE var3          ; AL = 04
MOV AL,TYPE var4          ; AL = 08
MOV AX,TYPE var5          ; AX = 000A
MOV AX,TYPE var6          ; AX = 0001
MOV AX,TYPE msg           ; AX = 0001
MOV AX,TYPE _start        ; AX = FF04h
ret                       ; возврат управления  ОС
end _start                ; завершение программы с именем _start
```

Вікна налагоджувача OllyDbg для досліджування оператора TYPE наведено на рис. 3.1.

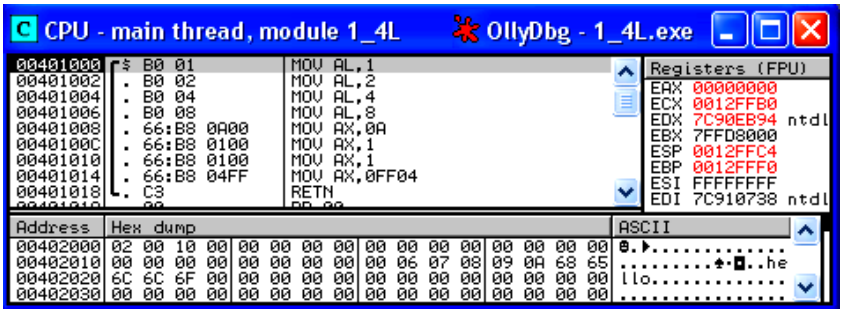


Рис. 3.1. Вікна налагоджувача OllyDbg для дослідження оператора TYPE

➢ **LENGTH** – оператор, який повертає кількість байтів, слів, подвійних слів, збільшених учетверо слів або десятибайтових елементів змінної. Це має значення тільки для змінних, створених за допомогою оператора DUP.

Лістинг 3.2. Використання оператора LENGTH:

```
    .386                        ; директива определения типа микропроцессора
    .model flat                 ; задание линейной модели памяти
    option casemap:none         ; отличие строчных и прописных букв
.data                           ; директива определения данных
    var1 DB 2                   ; переменная var1 := 2 занимает в памяти один байт
    var2 DB 3 DUP(5)            ; три числа со значением 5 каждый
    var3 DD 4 DUP(0)          ; четыре нулевых числа, каждый размером 32 бита
    var4 DQ 5 DUP(2 DUP(2))
    var5 DT ?
    var6 DB 6,7,8,9,10
    msg DB 'hello', 0
.code                                   ; директива начала сегмента команд
_start:                                 ; метка начала программы с именем _start
    MOV AL,LENGTH var1   ; AL = 01 (оператор DUP не  используется)
    MOV AL,LENGTH var2   ; AL = 03
    MOV AL,LENGTH var3   ; AL = 04
    MOV AL,LENGTH var4   ; AL = 05
     MOV AX,LENGTH var5  ; AX = 0001
```

```
MOV AX,LENGTH var6   ; AX = 0001
MOV AX,LENGTH msg    ; AX = 0001
MOV AX,LENGTH _start ; AX = 0001
  ret                          ; возврат управления  ОС
  end _start                   ; завершение программы с именем _start
```

Вікна налагоджувача OllyDbg для дослідження оператора LENGTH наведено на рис. 3.2.

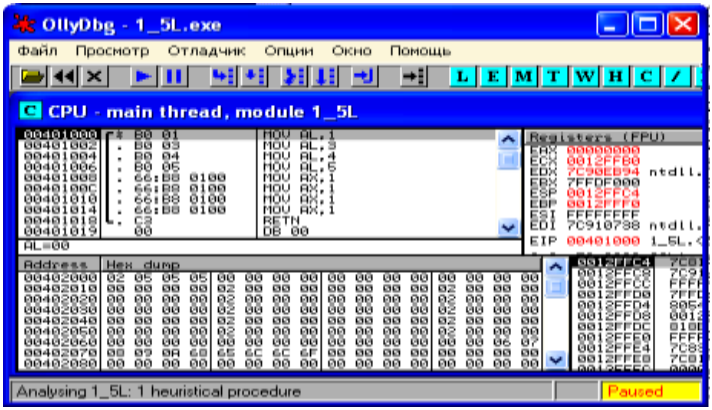


Рис. 3.2. Вікна налагоджувача OllyDbg для дослідження оператора LENGTH

- **AND, OR, NOT** – порозрядні оператори для цілих констант. . Наприклад:

```
mov   ax,1234h AND 4321h ; mov ax,0220h
```

- **+, -, *, /** – оператори складання, віднімання, множення і ділення цілих чисел. Наприклад:

```
mov      eax, 2*(3+4)        ; mov eax,14
```

- **EQ, NE, LT, LE, GT, GE** – оператори відношення: =, ≠, <, ≤, >, ≥. Асемблер повертає значення 0ffffh, коли результатом відношення є *правда,* і 0 – коли брехня.
- **HIGH** – оператор, який повертає старші 8 бітів константного виразу.
- **HIGHWORD** – оператор, який повертає старші 16 бітів у 32-розрядному операнді.
- **LOW** – оператор, який повертає молодші 8 бітів константного виразу.
- **LOWWORD** – оператор, який повертає молодші 16 біт 32-розрядного операнда.
- **MASK** – оператор, який повертає бітову маску змінної. Бітова маска зберігає тільки важливі біти, скидаючи всі останні в нуль. Змінна має бути визначена директивою RECORD.
- **MOD** – оператор модуля, який повертає цілий залишок операції ділення. Наприклад: mov al,90 mod 7 ; mov al,6.

- **SHORT** – оператор, який повертає атрибут мітки в SHORT. Часто використовується в командах JMP, наприклад, JMP SHORT Label.
- **SIZE** – оператор, який повертає загальну кількість байтів, виділених для змінної. Розраховується як довжина (LENGTH), помножена на тип (TYPE).
- **FIELD (.)** – ім'я з подальшою точкою (.), яке визначає поле із зумовленою структурою, додаючи зсув поля до зсуву змінної. Формат: variable.field.
- **THIS** – оператор, який створює операнд певного типу в програмі. Тип може визначатися оператором PTR або директивою LABEL.
- **WIDTH** – оператор, який повертає кількість бітів, зайнятих змінною, яка була оголошена за допомогою директиви RECORD.
- **&** – символ, що підставляє фактичне значення параметра макрокоманди.
- – символ, що інтерпретує текст як літерал, незалежно від спеціальних символів, що можливо містяться в ньому.
- | – символ, що інтерпретує цей символ як літерал, незалежно від спеціального значення, яке він може мати.
- **%** – символ, що інтерпретує текст як вираз, обчислює його значення і замінює текст отриманим результатом. Текстом може бути або числовий вираз, або текстове привласнення.
- **;;** – символ, що пригнічує виділення пам'яті для коментарю в макровизначенні.

3.3. Лабораторна робота «Подання даних»

Мета заняття

1. Виконати дослідження переводу чисел з десяткової в двійкову систему числення. Дати їх внутрішнє (машинне) подання відповідно до діапазону в знакових і беззнакових форматах типів Shortlnt (signed char), Byte (unsigned char), Integer (int), Word (unsigned int). Машинне подання даних має бути наведено в двійковій і шістнадцятковій системах числення.

2. Виконати дослідження переведення чисел із десяткової в двійкову систему числення. Дати їх внутрішнє (машинне) подання у форматах типів Single (float), Double (double), Extended (long double). Машинне подання даних має бути наведено в двійковій і шістнадцятковій системах числення.

Вхідний контроль знань

1. Перевести числа вашого дня, місяця та року народження у двійкову, вісімкову та шістнадцяткову системи числення.

2. Перевести число DD.MM у двійкову та шістнадцяткову системи числення для 32-розрядного формату дійсного числа, де DD – день вашого народження, MM – місяць вашого народження.

Завдання 1. Внутрішнє подання цілочислових даних у IBM PC:

– вибрати для свого варіанта цілі числа;

– перевести їх із десяткової у двійкову (або шістнадцяткову) систему числення;

– одержати їх внутрішнє подання;

– написати програму опису цих чисел мовою асемблера й одержати лістинг програми;

– перевірити правильність своїх викладок.

Завдання 2. Внутрішнє подання дійсних даних у IBM PC:

– обчислити для свого варіанта дійсні числа;

– перевести їх із десяткової у двійкову систему числення і зробити нормалізацію чисел;

– одержати їх внутрішнє подання (числа вказує викладач);

– написати програму опису цих чисел мовою асемблера й одержати лістинг програми;

– перевірити правильність своїх викладок.

Зміст звіту

1. Постановка задачі для конкретного варіанта.
2. Протокол переведення всіх заданих чисел із десяткової у двійкову і шістнадцяткову системи числення.
3. Лістинг програм.
4. Висновки за результатами роботи.

Порядок виконання роботи

Для виконання ***завдання 1*** необхідно вибрати свій варіант із табл. 3.3 та записати тільки число X зі знаками.

Таблиця 3.3

№	Завдання		№	Завдання		№	Завдання	
з/п	$\pm X$	$\pm Y$	з/п	$\pm X$	$\pm Y$	з/п	$\pm X$	$\pm Y$
1	200	20	11	900	70	21	1400	120
2	250	25	12	950	75	22	1450	125
3	300	30	13	1000	80	23	1500	130
4	350	35	14	1050	85	24	1550	135
5	400	40	15	1100	90	25	1600	140
6	450	45	16	1150	95	26	1650	145
7	500	50	17	1200	100	27	1700	150
8	600	55	18	1250	105	28	1750	155
9	700	60	19	1300	110	29	1800	160
10	800	65	20	1350	115	30	2000	170

Наприклад, для 1-го варіанта цілі числа будуть такі:

+ 200 – 200

Написати програму з використанням всіх директив визначення даних. При поданні даних **необхідно враховувати те, що не всі числа можуть вміститися в обрану розрядну сітку.** У такому випадку такий формат даних використовувати непотрібно. Правильність одержаних результатів необхідно проконтролювати. Як приклад напишемо програму з ім'ям Lab1.asm (лістинг 3.3).

Лістинг 3.3. Програма подання цілих чисел:

```
.386            ; директива определения команд микропроцессора
  .MODEL flat ; задание линейной модели памяти
  .DATA                 ; начало сегмента данных
; целые числа в памяти размером в байт (8 разрядов)
    db 80 ;
    db -80 ;
    byte 80 ;
    byte -80 ;
    sbyte 80 ;
    sbyte -80 ;
; целые числа в памяти размером в слово (16 разрядов)
    dw 80 ;
    dw -80 ;
    word 80 ;
    word -80 ;
    sword 80 ;
    sword -80 ;
; целые числа в памяти размером в двойное слово (32 разряда)
    dd 80 ;
    dd -80 ;
    dword 80 ;
    dword -80 ;
    sdword 80 ;
    sdword -80 ;
; целые числа в памяти размером в 48 разрядов
    fword 80 ;
    fword -80 ;
; целые числа в памяти размером в 64 разрядов
    dq 80 ;
    dq -80 ;
    qword 80 ;
    qword -80 ;
; целые числа в памяти размером в 80 разрядов
    dt 80 ;
    dt -80 ;
    tbyte 80 ;
    tbyte -80 ;
 .CODE          ; початок сегмента команд
 _start:                ; метка начала основной части программы
      ret               ; виход из ОС Windows
      end _start        ; завершение программы с именем _start
```

Для одержання лістингу програми в асемблері MASM32 (опція Fl – **fil**e) необхідно викликати файловий менеджер FAR, відкрити

C:\masm32\bin

та поставити курсор на файл ml.exe. Натиснути Ctrl + <Enter>. У результаті цього в командний рядок буде скопійовано:

C:\masm32\bin> ml.exe

Потім необхідно добрати додаткові опції компілятора та натиснути <Enter>:

Ml.exe /c /coff /Fl Lab1.asm <Enter>

Командний рядок у програмі FAR наведено на рис. 3.3.

У результаті таких дій буде створено файл **Lab1.lst.**

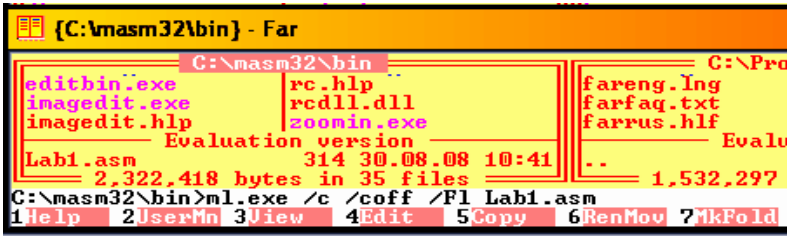


Рис. 3.3. Командний рядок у програмі FAR

Опції компілятора MASM, які використовуються в цьому командному рядку, означають:

c – компілювати без компонування;

coff – генерувати об'єктний файл у форматі COFF;

Fl (file) – генерувати лістинг.

Файловий менеджер FAR зручний тим, що дозволяє контролювати помилки, які виникли при трансляції. Для видимості результату трансляції необхідно натиснути клавіші Ctrl + F1 або Ctrl + F2. Повторне натиснення цих клавіш поверне зображення вікон файлового менеджера в початковий стан.

Лістинг програми Lab1.**lst** може мати такий вигляд:

```
Microsoft (R) Macro Assembler Version 6.14.8444    10/06/10 16:42:36
Lab1.asm                                                Page 1 - 1
        .386    ; директива определения команд микропроцессора
       .MODEL flat ; задание линейной модели памяти
 00000000       .DATA ; начало сегмента данных
                ; целые числа в памяти размером в байт (8 разрядов)
 00000000 50            db 80 ;
 00000001 B0            db -80 ;
 00000002 50            byte 80 ;
 00000003 B0            byte -80 ;
 00000004 50            sbyte 80 ;
 00000005 B0            sbyte -80 ;
                ; целые числа в памяти размером в слово (16 разрядов)
 00000006 0050          dw 80 ;
 00000008 FFB0          dw -80 ;
 0000000A 0050          word 80 ;
 0000000C FFB0          word -80 ;
 0000000E 0050          sword 80 ;
 00000010 FFB0          sword -80 ;
                ; целые числа в памяти размером в двойное слово (32 разряда)
 00000012 00000050      dd 80 ;
 00000016 FFFFFFB0      dd -80 ;
```

```
0000001A  00000050    dword 80 ;
0000001E  FFFFFFB0  dword -80 ;
00000022  00000050    sdword 80 ;
00000026  FFFFFFB0  sdword -80 ;
                ; целые числа в памяти размером в 48 разрядов
0000002A  000000000050          fword 80 ;
00000030  FFFFFFFFFFB0          fword -80 ;
                ; целые числа в памяти размером в 64 разряда
00000036  0000000000000050    dq 80 ;
0000003E  FFFFFFFFFFFFFFB0          dq -80 ;
00000046  0000000000000050          qword 80 ;
0000004E  FFFFFFFFFFFFFFB0          qword -80 ;
                ; целые числа в памяти размером в 80 разрядов
00000056  00000000000000000050          dt 80 ;
00000060  FFFFFFFFFFFFFFFFFFB0          dt -80 ;
0000006A  00000000000000000050          tbyte 80 ;
00000074  FFFFFFFFFFFFFFFFFFB0          tbyte -80 ;
00000000                .CODE               ; качан сегменту команд
00000000                _start:  ; метка начала основной части программы
00000000  C3                ret      ; выход из ОС Windows
                            end _start ;
Microsoft (R) Macro Assembler Version 6.14.8444   09/03/08 13:44:50
    Lab1.asm                                                    Symbols 2 - 1
    Segments and Groups:
            N a m e             Size     Length   Align   Combine Class
    FLAT . . . . . .GROUP
    _DATA  . . . . . .32 Bit   0000007E DWord   Public  'DATA'
    _TEXT  . . . . . 32 Bit   00000001 DWord   Public  'CODE'
    Symbols:
            N a m e           Type        Value         Attr
    @CodeSize  . . . . . .      Number          00000000h
    @DataSize  . . . . . .      Number          00000000h
    @Interface . . . . . .      Number          00000000h
    @Model . . . . . .          Number          00000007h
    @code  . . . . . .          Text            _TEXT
    @data  . . . . . .          Text            FLAT
    @fardata? . . . . . .       Text            FLAT
    @fardata . . . . . .        Text            FLAT
    @stack . . . . . .          Text            FLAT
    _start . . . . . .          L Near          00000000 _TEXT   Public
          0 Warnings
          0 Errors
```

Порядок розміщення цілих чисел у пам'яті необхідно також проконтролювати за допомогою налагоджувача OllyDbg. Для цього спочатку слід одержати .exe-файл за допомогою команди

link.exe /subsystem:console Lab1.obj.

Скопіювати вигляд налагоджувача у звіт можна, натиснувши клавіші PrtScr або Alt + PrtScr.

Потім потрібно викликати налагоджувач OllyDbg (http:// cracklab.ru/_dl/new/ollydbg110xp.rar або www.ollydbg.de/) та завантажити одержаний файл Lab1.exe (рис. 3.4).

На завершення слід проаналізувати порядок розміщення даних у пам'яті комп'ютера та їх відповідність до адрес.

Для виконання ***завдання 2*** необхідно згідно з варіантом із табл. 3.1 вибрати числа:

115.	3200.	115.3200
–115.	–3200.	–115.3200
0.115	0.32	
–0.115	–0.32	

Перевести одержані числа в шістнадцяткову систему числення.

Правильність одержаних результатів необхідно проконтролювати. Дійсні числа у записі повинні мати точку у форматі числа.

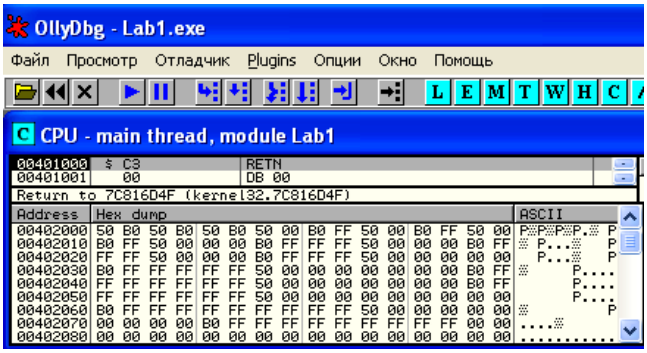


Рис. 3.4. Розміщення цілих чисел у налагоджувачі OllyDbg

Для цього напишемо програму з ім'ям, наприклад Lab1_2.asm (лістинг 3.4).

Лістинг 3.4. Програма подання дійсних чисел:

```
    .386                ; директива определения типа микропроцессора
    .model flat                 ; задание линейной модели памяти
.data                           ; директива определения данных
  ; вещественные числа в памяти размером в 32 разряда
    dd 115.
    dd -115.
    dd 0.115
    dd -0.115
    dd 3200.
    dd -3200.
    dd 0.32
    dd -0.32
    dd 115.3200
    dd -115.3200
    real4 115.
    real4 -115.
  ; вещественные числа в памяти размером в 64 разряда
      dq 115.
      dq -115.
      dq 0.115
      dq -0.115
      dq 3200.
```

```
        dq -3200.
        dq 0.32
        dq -0.32
        dq 115.3200
        dq -115.3200
        real8 115.
        real8 -115.
    ; вещественные числа в памяти размером в 80 разрядов
            dT 115.
            dT -115.
            dT 0.115
            dT -0.115
            dT 3200.
            dT -3200.
            dT 0.32
            dT -0.32
            dT 115.3200
            dT -115.3200
            real10 115.
            real10 -115.
.code                       ; директива начала кода программы
  _start:                   ; метка начала программы с именем start
     ret                            ; возвращение управления ОС
        end _start                  ; завершение программы с именем _start
```

Як відомо, для одержання лістингу програми необхідно викликати файловий менеджер FAR та вже з нього відкомпілювати програму за допомогою компілятора з мови асемблера MASM32, набравши в командному рядку команду

ml.exe /c /coff /Fl Lab1_2.asm <Enter>

Сторінка 1 лістингу програми Lab1_2.asm буде мати такий вигляд:

```
Microsoft (R) Macro Assembler Version 6.14.8444  09/03/10 15:40:46
Lab1_2.asm                                              Page 1 - 1
                .386        ; директива определения типа микропроцессора
                .model flat     ; задание линейной модели памяти
00000000        .data           ; директива определения данных
                ; вещественные числа в памяти размером в 32 разряда
00000000  42E60000              dd 115.
00000004  C2E60000              dd -115.
00000008  3DEB851F              dd 0.115
0000000C  BDEB851F              dd -0.115
00000010  45480000              dd 3200.
00000014  C5480000              dd -3200.
00000018  3EA3D70A              dd 0.32
0000001C  BEA3D70A              dd -0.32
00000020  42E6A3D7              dd 115.3200
00000024  C2E6A3D7              dd -115.3200
00000028  42E60000              real4 115.
0000002C  C2E60000              real4 -115.
                        ; вещественные числа в памяти размером в 64 разряда
00000030  405CC00000000000              dq 115.
00000038  C05CC00000000000              dq -115.
00000040  3FBD70A3D70A3D71              dq 0.115
00000048  BFBD70A3D70A3D71              dq -0.115
00000050  40A9000000000000              dq 3200.
00000058  C0A9000000000000              dq -3200.
00000060  3FD47AE147AE147B              dq 0.32
00000068  BFD47AE147AE147B              dq -0.32
```

```
00000070  405CD47AE147AE14          dq 115.3200
00000078  C05CD47AE147AE14          dq -115.3200
00000080  405CC00000000000          real8 115.
00000088  C05CC00000000000          real8 -115.
                    ; вещественные числа в памяти размером в 80 разрядов
00000090  4005E600000000000000      dT 115.
0000009A  C005E600000000000000      dT -115.
000000A4  3FFBEB851EB851EB851F      dT 0.115
000000AE  BFFBEB851EB851EB851F      dT -0.115
000000B8  400AC800000000000000      dT 3200.
000000C2  C00AC800000000000000      dT -3200.
000000CC  3FFDA3D70A3D70A3D70A      dT 0.32
000000D6  BFFDA3D70A3D70A3D70A      dT -0.32
000000E0  4005E6A3D70A3D70A3D7       dT 115.3200
000000EA  C005E6A3D70A3D70A3D7      dT -115.3200
000000F4  4005E600000000000000      real10 115.
000000FE  C005E600000000000000      real10 -115.
00000000                .code       ; директива начала кода программы
00000000                _start:     ; метка начала программы с именем start
00000000  C3            ret         ; возвращение управления ОС
                        end _start  ; завершение программы с именем  _start
```

Порядок розміщення дійсних чисел у пам’яті необхідно також проконтролювати за допомогою налагоджувача OllyDbg. Для цього потрібно одержати .exe-файл за допомогою команди

link.exe /subsystem:console Lab1_2.obj,

або, що й ще зручніше, – створити bat-файл, та вже при виконанні цього файлу автоматично отримати зразу два файли: .obj та .exe. Такий bat-файл може мати вміст:

```
ml /c /coff /Fl   "Lab1_2.asm"
link /SUBSYSTEM:CONSOLE  "Lab1_2.obj"
```

Потім слід викликати налагоджувач OllyDbg та завантажити одержаний файл Lab1_2.exe (рис. 3.5).

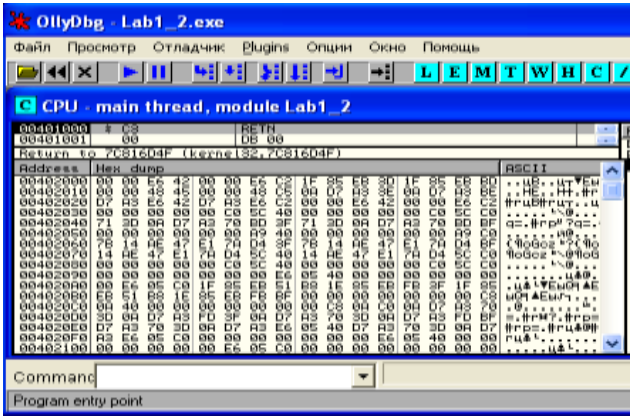


Рис. 3.5. Розміщення дійсних чисел у налагоджувачі OllyDbg

На закінчення, потрібно проаналізувати порядок розміщення даних у пам'яті комп'ютера та їх відповідність до адрес.

Довідковий матеріал

Формати даних для подання двійкових чисел

У форматі з фіксованою крапкою числа подаються в додаткових кодах.

Знак числа кодується в старшому біті (0 – "+", 1 – "–").

Наприклад, для подання числа $A = [-1990]_{10}$ скористаємося переведенням його в додатковий код і форматом для цілих чисел із знаком, причому доповнимо число А2 до п'ятнадцяти розрядів:

$$A = [-1990]_{10} = [-11111000110]_{2пк} = [-000\ 0111\ 1100\ 0110]_{2пк} =$$
$$= [1.111\ 1000\ 0011\ 1010]_{2дк}.$$

Формати даних для подання дійсних чисел

Числа з рухомою крапкою призначені для подання дійсних чисел і можуть бути типу *float* (32 розряди), *double* (64 розряди) або *long double* (*extended)* (80 розрядів).

У masm32 довідковий матеріал відносно форматів дійсних чисел наведено за адресою masm32\tutorial\fputute\fpuchap2.htm.

Внутрішнє (машинне) подання цих чисел достатньо складне:

S	Характеристика	Нормалізована мантиса

Біт *S* – знак числа.

Характеристика = Зміщення ± Порядок.

Зміщення – число, що дорівнює ПОЛОВИНІ максимально можливого, яке може поміститься в полі «Характеристика». Таким чином, економиться місце і час, оскільки не потрібно виділяти розряд **для знаку порядку** і робити додатковий код для від'ємних порядків – він завжди додатний.

Розмір простору, який ПК використовує для зберігання характеристики і мантиси, встановлений стандартом IEEE 724-1985 і підтримується всією сучасною комп'ютерною архітектурою.

Нормалізованим двійковим числом називається двійкове число, яке починається з одиниці і має такий вигляд:

$$X = \pm 1.m_2 \cdot 2^p,$$

де m_2 – двійкова мантиса числа; p – порядок двійкового числа.

Формат короткого (32-розрядного) дійсного числа:

1 – прихований розряд

v

S	Характеристика – **8 розрядів**			Нормалізована мантиса – **23 розряди**				
31	30	. . .	23	22	21	. . .	1	0

Зміщення становить $\mathbf{7F}_{16} = 127_{10}$.

Загальний формат 32-розрядного дійсного числа дорівнює

S (1 біт) + E (8 біт) + M (23 біти),

де S – знак числа; E – зміщена характеристика (порядок + $7F_{16}$); M – мантиса.

Задача 1. Подати число $A = 0.25_{10}$ у форматі REAL4 (DD) – 32-розрядного числа з рухомою крапкою.

Розв'язання

Мантиса: $A = 0.25_{10} = 0.01_2 = 1.0_2 \cdot 2^{-2.}$

Характеристика: $7F - 2 = 7D_{16} = 01111101_2$.

$[0.25_{10}]_p = 0\ 01111101\ 00000000000000000000000_2 = 3E800000_{16}$;

$[-0.25_{10}]_p = 1\ 01111101\ 00000000000000000000000_2 = BE800000_{16}$;

Задача 2. Подати число $A = 115._{10}$ у форматі REAL4 (DD) – 32-розрядного числа з рухомою крапкою.

Розв'язання

Мантиса: $A = 115._{10} = 1110011_2 = 1.\underline{110011}_2 \cdot 2^6$.

Характеристика: 7F + 6 = 85h = <u>1000 0101.</u>

Число: $A = 115._{10} = 0$ <u>1000 0101</u> <u>1100 11</u>00000000000000000b = 42E6 0000h.

Задача 3. Подати число $A = -115.32_{10}$ у форматі REAL4 – 32-розрядного числа з рухомою крапкою.

Розв'язання

Мантиса: $A = -115.32_{10} = 1.\underline{110011}.01010001111010111_2 \cdot 2^6$.

Характеристика: 7F + 6 = 85h = <u>1000 0101</u>b.

Число: $A = 115.32_{10} =$ **1** <u>1000 0101</u> <u>110 011</u>0 1010 0011 1101 0111 =

= C2E6 A3D7h.

Формат довгого (64-розрядного) дійсного числа:

1 – прихований розряд

v

S	Характеристика			Нормалізована мантиса				
63	62	. . .	52	51	50	. . .	1	0

Зміщення становить $\mathbf{3FF}_{16} = 1023_{10}$.

Задача 4. Подати число $A = -115.32_{10}$ у форматі REAL8 – 64-розрядного числа з рухомою крапкою.

Розв'язання

Мантиса: $A = -115.32_{10} = 1.\underline{1100\ 11}.01\ 0100\ 0111\ 1010\ 1110\ 0001\ 0100\ 0111\ 1010\ 1110\ 0001\ 0100_2 \cdot 2^6$.

Характеристика: 3FF + 6 = 405h = 100 0000 0101.

Число: A = -115.32_{10} =

= **1** 100 0000 0101 110 0 110 1 0100 0111 1010 1110 0001 0100 0111 1010 1110 0001 0100 = C05C D47A E147 AE14h.

Формат розширеного (80-розрядного) дійсного числа:

S	Характеристика			Нормалізована мантиса				
79	78	. . .	64	63	62	. . .	1	0

Особливість останнього формату полягає у тому, що мантиса НЕ МАЄ прихованого розряду.

Зміщення становить **3FFF**$_{16}$ = 16383_{10}.

Загальний формат 64-розрядного дійсного числа дорівнює

S (1 біт) + E (15 біт) + M (64 біти),

де S – знак числа; E – зміщена характеристика (порядок + $3FFF_{16}$); M – мантиса.

Задача 5. Подати число A = 115.32_{10} у форматі REAL10 – 80-розрядного числа з рухомою крапкою.

Розв'язання

Мантиса: A = 115.32_{10} = $0.110011.01010001111010111_2 \cdot 2^6$.

Характеристика: 3FFF + 6 = 4005h = 100 0000 0000 0101.

Число: A = 115.32_{10} =

= **0** 100 0000 0000 0101 **1**110 0110 1010 0011 1101 0111 0000 1010 0011 1101 0111 0000 1010 0011 1101 0111 = 4005 E6A3 D70A 3D70 A3D7h.

Контрольні запитання

1. Для чого використовуються директиви асемблера?
2. Для чого використовуються директиви керування моделями пам'яті?
3. Яку модель пам'яті використовує ОС Windows NT?
4. Для яких цілей використовують директиви визначення даних?
5. Які директиви визначення даних використовують для визначення числа зі знаком?
6. Які директиви визначення даних використовують для визначення дійсних чисел?
7. Для чого використовують директиви символьних констант?

«Нам не дано предугадать, как наше слово отзовется…»
Ф.И. Тютчев, 1869 г.

4. ОПЕРАЦІЇ ПЕРЕСИЛАННЯ ДАНИХ

Команди пересилання даних зручніше розглядати відповідно до їх функціонального призначення, згідно з яким їх можна розбити на такі групи команд:

– пересилання даних загального призначення;
– роботи зі стеком;
– роботи з адресами і вказівниками;
– перетворення даних;
– введення-виведення в порт.

При розгляді команд приймемо умовні позначення:

• r8, r16, r32, r64 – позначають будь-який 8-, 16-, 32- або 64-розрядний регістр загального призначення відповідно;

• m8, m16, m32, m64 – позначають адресу 8-, 16-, 32- або 64-розрядного елементу оперативної пам'яті відповідно;

• imm8, imm16, imm32, imm64 – позначають 8-, 16-, 32- або 64-розрядну константу, яка указується безпосередньо в команді;

• rel8, rel16, rel32 – позначають величину зсуву при використанні відносної адресації там, де це важливо виділити.

4.1. Команди пересилання даних загального призначення

До групи команд пересилання даних загального призначення належать команди MOV та XCHG. Команда **MOV** виконує пересилання байта чи слова з джерела (правий операнд) у приймач (лівий операнд).

Команди пересилання даних загального призначення можуть застосовуватись із множиною операндів, хоча й існують деякі винятки. Команди пересилання даних загального призначення (крім команди XCHG) – це єдині команди, що дозволяють як операнд використовувати регістр сегмента (http://www.intel.com/content/dam/www/public/us/en/ documents/ manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf).

MOV приймач, джерело	Пересилання даних

MOV r8,imm8	MOV r8,r/m8	MOV m8,r8	MOV m8,imm8
MOV r16,imm16	MOV r16,r/m16	MOV m16,r16	MOV m16,imm16
MOV r32,imm32	MOV r32,r/m32	MOV m32,r32	MOV m32,imm32
MOV r64, imm(32,64)	MOV r64,r/m64	MOV m64,r64	MOV m64,imm64

При використанні команди MOV необхідно враховувати такі обмеження:

– обидва операнди повинні мати однакову довжину;

– один з операндів обов'язково повинен використовуватися регістр (пересилка типу «пам'ять-пам'ять» в команді MOV не підтримується);

– як одержувач не можна указувати регістри CS, EIP та IP;

– не можна переслати безпосередньо задане значення в сегментний регістр.

MOV mem/reg1, mem/reg2

```
MOV EDX, ECX    ; передати подвійне слово з Рг ECX у Рг EDX
MOV EBX, [ECX]  ; передати дані з пам'яті, початкова адреса якої
                                        ; знаходиться в регістрі ECX до Рг EBX
MOV  TEMP, BL  ; передати байт із Рг BL у масив пам'яті  з ім'ям TEMP
MOV  [EDI], BX  ; передати слово з Рг BX у комірку пам'яті,
                                                ; адреса якої знаходиться в Рг EDI
MOV  EDX,[EBP][ESI] ; передати в Рг EDX дані з пам'яті,
                                ; адреса яких визначається як сума вмістів Рг EBP і Рг ESI
```

MOV mem/reg, data

```
MOV [EDI], 8400H  ; занесення операнда 8400h у комірку пам'яті,
                                                ; адреса якої знаходиться в Рг EDI
MOV [EBP][ESI], 2CH  ; передати число 2Ch у комірку пам'яті
                                        ; за адресою, яка визначається як сума EBP та ESI
MOV   ECX, 87654321H   ; передати число 87654321H в Рг ECX
```

MOV AC, mem

```
MOV EAX, TEMP    ; передати дані з пам'яті з ім'ям TEMP в Рг EAX
```

MOV mem, AC

```
MOV BETA, EAX    ; передати Рг EAX у пам'ять з іменем BETA
```

MOV sreg, mem/reg

```
MOV SS, BP        ; передати слово з Рг BP у сегментний Рг SS
MOV ES, [EDX]     ; передати число у сегментний Рг ES з комірки
                                        ; пам'яті, адреса якої знаходиться в Рг EDX
```

MOV mem/reg, sreg

```
MOV [BX], ES   ; передати слово із сегментного Рг ES
                                        ; у комірку пам'яті, адреса якої знаходиться в Рг BX
MOV CX, SS     ; передати слово із сегментного Рг SS у Рг CX
```

Форми команди MOV зберігають або завантажують такі спеціальні регістри в (із) регістри(ів) загального призначення:

- керуючі регістри CR0, CR2 і CR3;
- регістри налагоджування DR0, DR1, DR2, DR3, DR6 і DR7;
- тестові регістри TR3, TR4, TR5, TR6 і TR7.

XCHG операнд_1, операнд_2	Обмін операндів між собою

Команда **XCHG** змінює байт чи слово, які адресуються операндами. Регістри сегментів не можуть бути операндами команди XCHG.

XCHG reg, mem/reg

```
XCHG   AL, SEMA ; обмін байтами між Рг AL та змінною,
                                ; яка знаходиться в пам'яті та має ім'я SEMA
XCHG   AL, BL             ; обмін байтами між регістрами
XCHG   EAX, ECX           ; обмін словами між регістрами
```

CMOVcc приймач, джерело	Умовне пересилання даних

Команди **CMOVcc** копіюють вміст джерела в приймач, якщо задовольняється та або інша умова (табл. 4.1). Джерелом може бути регістр загального призначення або змінна, а приймачем – тільки регістр. Вимога, яка повинна виконуватися, – прирівнення до нуля або до одиниці тих або інших прапорців із регістра EFLAGS. Якщо використовувати команди CMOVcc відразу після команди CMP (порівняння) з тими ж операндами, умови набувають особливого сенсу, наприклад:

```
cmp eax,ebx          ; Порівняти eax  та  ebx (eax – ebx)
cmovb  eax,ebx   ; якщо eax < ebx, то скопіювати ebx в eax
```

Слова «вище» і «нижче» в табл. 4.1 стосуються до порівняння чисел без знака, слова "більше" і "менше" враховують знак.

Таблиця 4.1

Код команди	Реальна умова	Умова для CMP
CMOVA CMOVNBE	CF = 0 та ZF = 0	Якщо вище Якщо не нижче і не дорівнює
CMOVAE CMOVNB CMOVNC	CF = 0	Якщо вище або дорівнює Якщо не нижче Якщо немає перенесення
CMOVB CMOVNAE CMOVC	CF = 1	Якщо нижче Якщо не вище і не дорівнює Якщо перенесення
CMOVBE CMOVNA	CF = 1 або ZF = 1	Якщо нижче або дорівнює Якщо не вище
CMOVE CMOVZ	ZF = 1	Якщо дорівнює Якщо нуль
CMOVG CMOVNLE	ZF = 0 та SF = OF	Якщо більше Якщо не менше і не дорівнює
CMOVGE CMOVNL	SF = OF	Якщо більше або дорівнює Якщо не менше
CMOVL CMOVNGE	SF <>OF	Якщо менше Якщо не більше і не дорівнює
CMOVLE CMOVNG	ZF = 1 або SF <> OF	Якщо менше або дорівнює Якщо не більше

Продовження табл. 4.1

CMOVNE CMOVNZ	ZF = 0	Якщо не дорівнює Якщо не нуль
CMOVNO	OF = 0	Якщо немає переповнення
CMOVO	OF = 1	Якщо є переповнення
CMOVNP CMOVPO	PF = 0	Якщо немає парності Якщо непарне
CMOVP CMOVPE	PF = 1	Якщо є парність Якщо парне
CMOVNS	SF = 0	Якщо немає знака
CMOVS	SF = 1	Якщо є знак

BSWAP r32 **BSWAP r64**	Обмін байтами у регістрі

Команда **BSWAP** виконує обмін байтами у 32-бітовому регістрі. Біти 0–7 (молодший байт молодшого слова) міняються місцями з бітами 24–31 (старший байт старшого слова), а біти 8–15 (старший байт молодшого слова) – з бітами 16–23 (молодший байт старшого слова).

```
mov    eax,12345678h
bswap  eax          ; в eax знаходиться 78563412h
```

Щоб виконати обмін байтами у 16-бітовому регістрі, слід використовувати команду XCHG:

```
xchg   al, ah               ; обмін байтами AX
```

У мікропроцесорах Intel команду BSWAP можна використовувати і для обміну байтами в 16-бітових регістрах, але в деяких сумісних мікропроцесорах інших фірм цей варіант не реалізований.

4.2. Команди роботи зі стеком

Стек є спеціальною дылянкою пам'яті, яка може організовуватися до відповідно одного з принципів: останнє значення, поміщене в стек, витягується першим або останнє значення, поміщене в стек, витягується останнім. Для адресації стека використовуються регістри SS:ESP (32-розрядні програми). Регістр SP (ESP) називається вказівником стека і містить 16- або 32-розрядну адресу останнього елементу, поміщеного в стек. Якщо останнє значення, поміщене в стек, витягується першим, то подібна структура називається LIFO (Last In, First Out – прибув останнім, обслужений першим). Стек зростає до менших адрес, тобто останнє значення, що надійшло в стек, зберігається за найменшою адресою.

Мінімальний розмір операнда, яким оперують команди стекових операцій, дорівнює слову (2 байти). З цієї причини дані в стеку віддалені один від одного на величину, кратну двом. Наприклад, при розміщенні в стеку слова значення вказівника стека SP (ESP) зменшується на 2, при розміщенні подвійного слова – на 4 і т.д. При цьому молодші байти операндів поміщаються в стек за молодшими адресами, а старші байти – за старшими адресами.

Для того щоб помістити яке-небудь значення в стек, потрібно використовувати команду push (в деяких випадках команди pusha, pushaw, pushad). Ця команда як параметр може приймати будь-який регістр або елемент пам'яті. При цьому вміст вказівника стека SP (ESP) зменшується на 2 (для слова) або на 4 (для подвійного слова). Команда допускає один із форматів:

```
push  reg16/reg32              ; запис у стек значення регістра
push  mem16/mem32              ; запис у стек змінної з пам'яті
push  sreg      ; запис у стек із сегментного регістра (CS, DS, ES)
push  immed                    ; запис у стек безпосереднього операнда
```

Команда **PUSH** (помістити) декрементує вказівник стека (регістр ESP), потім копіює операнд-джерело у вершину стека. Команда PUSH часто використовується для розміщення в стеку параметрів перед викликом процедури. Усередині процедури вона може бути використана для резервування простору в стеку під тимчасові змінні. Команда PUSH працює з операндами, розміщеними в пам'яті, з безпосередніми операндами і з регістровими операндами (включаючи регістри сегмента). Команда **PUSH** зменшує ESP на розмір джерела у байтах (2 або 4) та зберігає вміст джерела у пам'яті за адресою SS:[ESP].

PUSH r/m64	Переслати дані у стек

Набір команди роботи зі стеком змінювався у міру збільшення розрядності мікропроцесора, забезпечуючи зворотну сумісність і можливість роботи з операндами різної розрядності. Краще уникати неоднородностей і обмінюватися зі стеком тільки 64-розрядними даними.

Команда **PUSH** зменшує вказівник стека на розмір операнда. Потім команда PUSH поміщає операнд у нову вершину стека, на яку вказує вказівник стека.

PUSH mem/(s)reg

```
PUSH ECX   ; завантажити в стек слово з Рг ECX
PUSH [EBX] ; завантажити в стек дані, адреса яких знаходиться  в Рг EBX
PUSH SI        ; включення в стек регістра SI
PUSH ES        ; включення в стек сегментного регістра ES
PUSH [EDI]     ; включення в стек подвійного слова з пам'яті
```

PUSHF	Завантажити регістр FLACS у стек
PUSHFD	Завантажити регістр EFLACS у стек

Команда **PUSHF** (засилання бітів умов у стек) зменшує вказівник стека SP на два і пересилає всі біти умов у два байти стека, що знаходяться за адресою, зазначеною в регістрі SP.

POP r/m64	Зчитати дані зі стека

Команда **POP** – відновити (дані) зі стека – передає слово або подвійне слово з вершини стека (на яку вказує регістр RSP) операнда призначення і потім збільшує значення регістра RSP, щоб той указував на нову вершину стека. Команда POP переміщає інформацію зі стека в регістр загального призначення, в регістр сегмента або в пам'ять. Для передання подвійного слова зі стека в регістр загального призначення використовується спеціальна форма команди POP.

Команда POP заміняє вміст операнда пам'яті або регістра даними, знятими з вершини стека, який адресується за допомогою RSP (при атрибуті розміру адреси 64 біти). Вказівник стека SP набуває збільшення на 2 при розмірі операнда 16 бітів або на 4 при розмірі операнда 32 біти. Після цього вказівник установлюється на нову вершину стека.

Команда POP CS не є командою мікропроцесора. Витягання даних зі стека в регістр CS здійснюється командою RET.

POP mem/(s)reg

```
POP  ALPHA ; витягання зі стека в ділянку пам'яті,  яка має ім'я ALPHA
POP  DS      ; витягання зі стека в сегментний регістр
POP  EDX    ; витягання зі стека в регістр EDX
```

POPA	Зчитати дані зі стека всіх 16-розр. Рг загального призначення
POPAD	Зчитати дані зі стека всіх 32-розр. Рг загального призначення

Команди **POPA/PUSHA** в IA-32e ***не підтримуються***.

Команда **POPA/POPAD** – витягання зі стека всіх регістрів загального призначення у такому порядку: DI, SI, BP, BX, DX, CX і AX. Команди **POPA/POPAD** виконують дії, протилежні діям **PUSHA/PUSHAD**.

Команда POPA витягає зі стека вісім 16-бітових регістрів загального призначення. Однак значення SP замість завантаження регістра SP знищується. Команда POPA реверсує дію команди PUSHA, відновлюючи значення регістрів загального призначення, які вони мали перед виконанням команди PUSHA.

POPF	Завантажити регістр FLACS зі стека
POPFD	Завантажити регістр EFLACS зі стека

Команда **POPF** (витягання бітів умов IP із стека) пересилає біти умов зі стека, що знаходиться за адресою, зазначеною в регістрі SP, у регістр бітів умов і збільшує регістр SP на два.

POPFQ	Витягання зі стека регістра прапорців RFLAGS

Команда pop використовує ті самі типи операндів, що і команда push. Наприклад, в програмі на лістингу 4.1 регістри міняються вмістом.

Лістинг 4.1. Програма, в якій регістри міняються вмістом:

```
        ; masm32:  компиляция – ml.exe /c /coff   A1.asm
        ; компоновка – link.exe /SUBSYSTEM:CONSOLE   A1.obj
                                ; pause
                                ; start  A1.exe
    .686              ; директива определения типа микропроцессора
.model  flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
    .code                 ; директива начала кода программы
     _start:              ; метка начала программы с именем start
      mov eax, 2                  ; eax = 2
      mov ecx, 3                  ; ecx = 3
      push eax                    ; запоминание в стеке значения регистра eax
      push ecx                    ; запоминание в стеке значения регистра ecx
      pop  eax                    ; считать данные из вершины стека в eax (eax = 3)
      pop  ecx                    ; считать данные из вершины стека в ecx (ecx = 2)
      ret                         ; возврат управления  ОС
      end _start                  ; завершение программы с именем _start
```

Через те, що розмірність регістрів у попередньому прикладі була однаковою, передача значень здійснена успішно. Розглянемо, наприклад, лістинг 4.2, в якому розмірність регістрів вибрана не однаковою.

Лістинг 4.2. Програма, в якій розмірність регістрів вибрана не однаковою:

```
      .686                ; директива определения типа микропроцессора
      .model  flat        ; задание линейной модели памяти
  .code                   ; директива начала кода программы
    _start:               ; метка начала программы с именем _start
      mov ax,  3210h
      mov ecx, 0BA987654h
      push ax                     ; esp := esp – 2
      push ecx                    ; esp := esp – 4
      pop  ax                     ; esp := esp + 2, ax = 7654h
      pop  ecx                    ; esp := esp + 4, ecx = 3210BA98h
      ret                         ; возврат управления  ОС
     end _start                   ; завершение программы с именем _start
```

Стан стека при виконанні команд push і pop має такий вигляд:

Адреса стека	вміст	
0012FFC4	77E7EB69	початкове значення стека
0012FFC2	EB693210	після виконання push ax
0012FFBE	BA987654	після виконання push ecx
0012FFC0	3210BA98	після виконання pop ax
0012FFC4	77E7EB69	після виконання pop ecx

Стан програми з лістингу 4.2 після виконання команди push ecx наведено на рис. 4.1.

Стек зростає у напрямі зменшення адрес. Наприклад, після команди push ax в стек додається два байти, а адреса вершини зменшується на два. Команда push ecx 4-байтова, тому вершина стека зменшується на 4.

Команда pop збільшує адресу вершини стека, але не стирає самі числа.

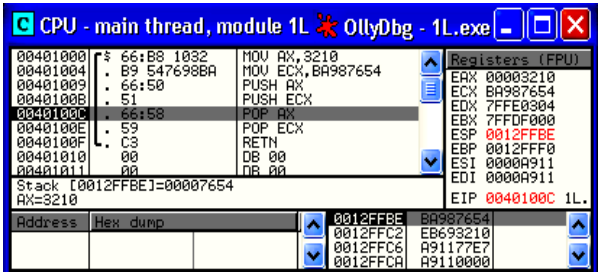


Рис. 4.1. Програма після виконання команди push ecx

Програма з лістингу 4.3 підбирає в стеку числа, що залишилися.

Лістинг 4.3. Програма, яка підбирає в стеку числа, що залишилися:

```
.686                    ; директива определения типа микропроцессора
.model flat             ; задание линейной модели памяти
.code                   ; директива начала программы
_start:                 ; метка начала программы с именем _start
  mov ax,  3210h        ; занесение в ax шестнадцатиричной константы
  mov ecx, 0BA987654h   ; занесение в ecx шестнадцатиричной константы
  push ax               ; esp = esp  – 2
  push ecx              ; esp = esp  – 4
  pop  ax               ; esp = esp + 2, ax = 7654h
  pop  ecx              ; esp = esp + 4, ecx = 3210BA98h
  mov  ecx,[esp-6]      ; ecx = BA987654h
  mov  ax, [esp-2]      ; ax = 3210h
  ret                   ; возврат управления  ОС
 end _start             ; завершение программы с именем _start
```

PUSHA	Переслати дані у стек всіх 16-розр. Рг загального призначення
PUSHAD	Переслати дані у стек всіх 32-розр. Рг загального призначення

Команда **PUSHA/PUSHAD** (помістити регістри) зберігає вміст восьми регістрів загального призначення в стеку (AX, CX, DX, BX, SP, BP, SI, DI) / (EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI). Ця команда спрощує виклики процедур шляхом скорочення кількості команд, необхідних для збереження вмісту регістрів загального призначення. Результат виконання команди PUSHA протилежний дії команди POPA.

Команда PUSHA зменшує вказівник стека (SP) на 16, що відповідає запам'ятовуванню значень восьми регістрів, які розташовуються у стеку розміром у слово в такій послідовності: AX, CX, DX, BX, SP, BP, SI, DI.

Команда PUSHAD декрементує вказівник стека (ESP) на 32, що відповідає запам'ятовуванню значень восьми регістрів розміром у подвійне слово, та розташовує регістри у такій послідовності: EAX, ECX, EDX, EBX, ESP, EBP,

ESI, EDI.

Існують спеціальні модифікації команди push:

- **pushf** – збереження 16-розрядного регістра прапорців мікропроцесора в стеку;
- **pushfd** – збереження 32-розрядного регістра прапорців мікропроцесора в стеку.

4.3. Команди роботи з адресами і вказівником

Розглянуті команди відрізняються від інших команд передання тим, що при їх виконанні в адресований регістр передаються не дані, а адреси. Основне застосування команд – це ініціалізація регістрів перед виконанням ланцюгових команд.

LEA приймач, джерело	Обчислення ефективної адреси

LEA (LDS) (LES) reg, mem

Команда **LEA** (завантаження виконавчої адреси) передає зміщення, яке адресується в пам'яті правим операндом у загальний регістр-вказівник чи індексний регістр, який адресується лівим операндом.

```
lea ebx, [ebx] [esi]
lea esi,mas1 ; занесення в esi адреси початку елементів масиву mas1
```

Команду LEA часто використовують для виконання арифметичних обчислень, таких, наприклад, як множення

```
lea  ebx,[ebx + ebx*4]          ; EBX = EBX+EBX × 5
```

або додавання

```
lea   ebx,[eax + 12]            ; EBX = EAX + 12
```

Ці команди не змінюють прапорці.

LDS прм, джерело	Завантажити адресу, використовуючи DS
LES прм, джерело	Завантажити адресу, використовуючи ES
LFS прм, джерело	Завантажити адресу, використовуючи FS
LGS прм, джерело	Завантажити адресу, використовуючи GS
LSS прм, джерело	Завантажити адресу, використовуючи SS

Ці команди роботи з указівниками завантажують далекий покажчик в регістри процесора. Далекий указівник складається з 16-розрядного селектора сегмента, який завантажується в сегментний регістр і 32-розрядного зсуву усередині сегмента, який завантажується в регістр загального призначення.

Приклад: LDS ESI, STRING_X

Команда завантажує в регістр DS селектор сегмента для сегмента, адреса якого задана в STRING_X, і завантажує зсув усередині сегмента, що адресу-

ється STRING_X, в регістр ESI. Задання регістра ESI як операнда призначення зручно при підготовці до виконання операцій із рядками, коли рядок-джерело знаходиться не в поточному сегменті.

4.4. Команди перетворення даних

До групи команд перетворення даних можна віднести декілька команд процесора, але більшість із них мають особливості, які вимагають віднести їх до інших функціональних груп. До команд перетворення даних можна віднести тільки одну команду

xlat [адреса_таблиці_перекодування]

XLAT	Трансляція відповідності до таблиці

Команда **XLAT** здійснює пошук у таблиці і заміну операнда на відповідний байт із таблиці.

Команда **XLAT** поміщає в регістр AL байт, що знаходиться за адресою RBX+AL – використовує AL як індекс у таблиці, адреса якої знаходиться RBX (EBX).

Лістинг 4.4. Програма дослідження команди xlatb:

```
    .686                    ; директива определения типа микропроцессора
    .model  flat            ; задание линейной модели памяти
.data                       ; директива начала  сегмента данных
    table DB 1,2,3,4,5,6,7,8,9,0Ah
.code                       ; директива начала кода программы
  _start:                   ; метка начала программы с именем _start
    mov  al, 06h            ; занесение в al числа 06h
    mov  ebx, offset table  ; занесение в ebx адреса
; расположение у памяти переменной с именем  table
    xlatb                   ; поиск в таблице и замена операнда
        ret                 ; возврат управления  ОС
        end _start          ; завершение программы с именем _start
```

Стан налагоджувача після виконання програми з лістингу 4.4 наведено на рис. 4.2. З рис. 4.2 видно, що в результаті виконання програми до регістра EAX занесене число, яке розташовувалося в 6-й комірці пам'яті, рахуючи від нуля (число 7).

4.5. Команди введення та виведення в порт

Команда **IN** пересилає байт (слово або подвійне слово) зі вхідного порту в регістр AL (AX або EAX). Порт вказується або байтом даних (доступ до портів від нуля до двохсот п'ятидесяти п'яти), або номером порту в регістрі DX, що допускає звертання до 210 портів введення.

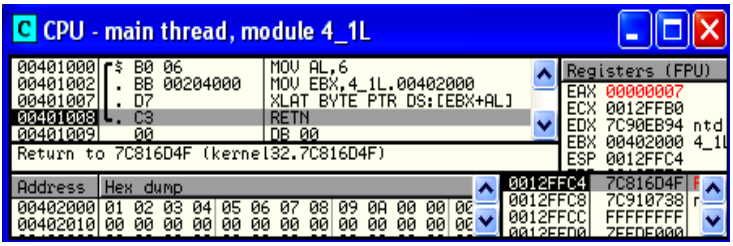


Рис. 4.3. Стан налагоджувача після виконання програми з лістингу 4.4

IN приймач, джерело	Зчитати дані з порту

IN ac, port
IN ac, DX

```
IN AX, 40H      ; Введення слова з 40-го порту
IN AL, DX       ; Введення байта з порту, адреса якого знаходиться в Pr DX
```

OUT приймач, джерело	Записати дані в порт

Команда **OUT** працює подібно до команди IN, але передача інформації здійснюється з акумулятора в порт виведення.

OUT port, ac
OUT DX, ac

```
OUT 80H, AL     ; Виведення байта з Pr AL через порт 80H
OUT DX, AX      ; Виведення слова з Pr AX через порт,
                ; адреса якого зазначена у Pr DX
```

4.6. Команди пересилання бітів умов

Існує чотири команди пересилання бітів умов.

LAHF	Завантажити регістр AH значеннями бітів умов

Команда **LAHF** (завантажити регістр AH значеннями бітів умов) передає біти умов SF, ZF, AF, PF і CF у регістр AH.

SAHF	Завантажити біти умов із регістра AH

Команда **SAHF** (завантажити біти умов із регістра AH) передає біти умов із регістра AH в однорозрядні регістри SF, ZF, AF, PF і SF.

STC	Встановити прапорець перенесення CF
CLC	Скинути прапорець перенесення CF
CMC	Інвертувати прапорець перенесення CF
STD	Встановити прапорець напрямку DF

CLD	Скинути прапорець напрямку DF
CLI	Заборона переривання

Команда **CLI** скидає прапорець IF в 0. Після виконання цієї команди мікропроцесор ігнорує всі переривання від зовнішніх пристроїв (окрім NMI). У захищеному режимі ця команда, так само, як і всі інші команди, що модифікують прапорець IF (POPF або IRET), виконується, тільки якщо програмі відповідають привілеї (CPL ≤ IOPL).

STI	Дозволити переривання

Команда **STI** встановлює прапорець IF в 1, відміняючи тим самим дію команди CLI.

SALC	Встановити AL відповідно до CF

Команда **SALC** встановлює AL в 0FFh, якщо прапорець CF = 1, і скидає в 00h, якщо CF = 0. Це недокументована команда з кодом 0D6h.

Контрольні запитання та завдання

1. Для яких цілей використовують команди пересилання даних?
2. Які команди пересилання даних використовують для передання даних за умовою?
3. Для чого використовують команди роботи зі стеком?
4. Записати день свого народження в 8-розрядну комірку пам’яті, місяць народження – в 16-розрядну, рік народження – у 32-розрядну та зарезервувати ще одну комірку пам’яті для тимчасових даних. Виконати такі дії:

– передати день свого народження із однієї комірки пам’яті в іншу;

– передати дані свого місяця народження в 32-розрядний регістр;

– передати адресу комірки пам’яті з роком свого народження в регістр.

5. ОСНОВНІ АРИФМЕТИЧНІ ОПЕРАЦІЇ

До цілочислових арифметичних команд належать операції:
– двійкової арифметики:
додавання (add, adc, inc);
віднімання (sub, sbb, dec);
множення (imul, mul);
ділення (idiv, div);
зміни знака (neg);
– десяткової арифметики
корекції додавання (aaa, daa);
корекції віднімання (aas, das);
корекції множення (aam);
корекції ділення (aad);
– перетворення типів (cbw, cwd, cwde, cdq, movsx, movzx);
– додаткові команди (bswap);
– інші команди з арифметичним принципом дії (cmp, cmpxchg, xadd).

5.1. Арифметичні операції

МП виконує основні математичні дії (з різноманітними модифікаціями) з урахуванням знака і без нього. *Для чисел зі знаком прийняте стандартне подання в* ***додатковому*** *коді.* При виконанні арифметичних дій встановлюються чи скидаються шість бітів умов: CF, AF, ZF, SF, PF, OF.

Команди додавання

ADD приймач, джерело	Додавання
ADC приймач, джерело	Додавання з урахуванням перенесення

Команда **ADD** здійснює додавання операндів і запам'ятовує результат за адресою, що задається першим операндом. *Тому* ***неможливо*** *виконувати операції, коли* ***обидва*** *операнди знаходяться у пам'яті.*

ADD r/m8, imm8	ADD r8, r/m8	ADD m8, r8
ADD r/m16, imm(8,16)	ADD r16, r/m16	ADD m16, r16
ADD r/m32, imm(8,32)	ADD r32, r/m32	ADD m32, r32
ADD r/m64, imm(8,32)	ADD r64, r/m64	ADD m64, r64

ADD mem/reg1, mem/reg2

```
ADD [EBX], ECX ; додати ECX до комірки пам'яті,
                                    ; адреса якої міститься у Рг EBX
ADD  ECX, EDX      ; ECX := ECX + EDX
ADD  EDI, ALPHA    ; EDI := EDI + ALPHA – додати змінну
                                    ; з пам'яті  з ім'ям ALPHA до Рг EDI
```

ADD mem/reg, data

```
ADD  AL, 02H          ; AL := AL + 02₁₆
ADD BET, 0FFH ; додати константу FFH до змінної у пам'яті  з ім'ям BET
```

Команда ADC здійснює додавання операндів і вмісту біта CF, результат запам'ятовується за адресою першого операнда.

ADC mem/reg1, mem/reg2

```
ADC  ECX, EDX       ; ECX := ECX + EDX + CF, регістр – регістр
ADC  EDI, ALPHA     ; EDI = EDI + ALPHA + CF – додати змінну
                    ; з пам'яті  з ім'ям ALPHA до Рг EDI та вміст біта CF
```

ADC mem/reg, data

```
ADC EAX, 7777H ; EAX = EAX + 00007777H + CF (прапорець CF)
```

Лістинг 5.1. Програма додавання двох чисел розміром у слово *без урахування переповнення* розрядної сітки:

```
      .686                    ; директива определения типа микропроцессора
      .MODEL flat             ; задание линейной модели памяти
      .data                         ; директива начала  сегмента данных
    x   dw  5       ; резервирование в памяти 2-х байтов для переменной X
    y   dw  3       ; резервирование в памяти 2-х байтов для переменной Y
    res  dw  ?      ; резервирование в памяти 2-х байтов для переменной Z
.code                               ; директива начала сегмента команд
  _start:                           ; метка начала программы
      mov  ax,x             ; загрузка операнда X
      add   ax,y            ; добавление операндов X и Y
     mov res,ax    ; сохранение результата в памяти в ячейке с именем  res
      ret                   ; выход из ОС Windows
      end   _start          ; окончание программы с именем _start
```

При написанні програми враховувалось те, що числа невеликі та результат їх додавання повністю розміститься в розрядній сітці.

Вікна налагоджувача OllyDbg з програмою з ім'ям 1_1L.exe наведено на рис. 5.1.

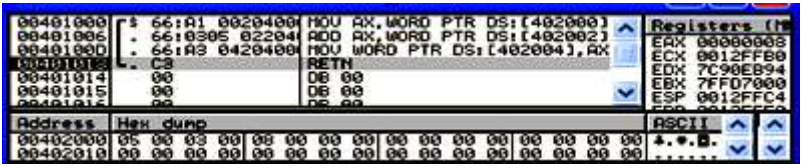


Рис. 5.1. Вікна налагоджувача OllyDbg з програмою з лістингу 5.1

У вікні CPU налагоджувача перша команда виконує занесення в регістр AX числа розміром у слово з комірки пам'яті за адресою 402000 вікна дампа пам'яті (число 5). Друга команда виконує додавання вмісту регістра AX (число 5) зі вмістом комірки пам'яті за адресою 402002 вікна дампа пам'яті числа 3

розміром у слово та збереження результату в регістрі AX. Третя команда записує результат додавання розміром у слово (16 розрядів) з регістра AX у комірку пам'яті за адресою 402004.

Для врахування можливого *переповнення розрядної сітки* можна використати команду ADC та занести це значення у відповідну комірку памяті, наприклад:

```
    .data                    ; директива начала  сегмента данных
   x  dw  5       ; резервування в пам'яті 2-х байтів для змінної X
   y  dw  3       ; резервування в пам'яті 2-х байтів для змінної Y
   res   dw  ?      ; резервування в пам'яті 2-х байтів для змінної RES
   resc  dw  0      ; резервування в пам'яті 2-х байтів для змінної RESC
.code                  ; директива начала сегмента команд
  _start:              ; мітка початку основного тіла програми
     mov  ax,x         ; завантаження операнда X
     add   ax,y        ; додавання операндів X та Y
     mov res,ax  ; збереження результату у пам'яті у комірці з ім'ям  RES
     adc resc,0   ; resc := resc + 0+ CF
     ret                          ; вихід із ОС Windows
     end   _start                 ; закінчення програми з іменем _start
```

У програмі для збереження переповнення розрядної сітки використовується комірка пам'яті resc та привласнюється їй значення 0 директивою resc dw 0. Командою adc resc,0 виконується додавання нульового значення комірки з ім'ям resc з нулем та зі значенням ознаки перенесення. Якщо ця ознака є, то в цю комірку записується 1.

Лістинг 5.2. Програма додавання з вісімковими та шістнадцятковими регістрами:

```
    .686              ; директива определения типа микропроцессора
     .model flat              ; задание линейной модели памяти
    .code                     ; директива начала сегмента команд
     _start:             ; метка начала программы с именем _start
         mov al, -120          ; al = 88h – пересылка  в al
         mov bl, -127          ; bl = 81h – пересылка  в bl
         add al, bl            ; сложение –   al = 09h, O = 1,  C = 1, S = 0
         mov ax, -120          ; al = ff88h,
         mov bx, -127          ; bx = ff81h = -127
         add ax, bx            ; ax = ff09h = -247, O = 0, S = 1
         ret                 ; возврат управления  ОС
     end _start              ; завершение программы с именем _start
```

Програма з лістингу 5.2 починається зі спроби скласти два числа –120 та –127, що зберігаються в байтах AL і BL. 8-розрядні регістри здатні мати 28 = 256 різних станів. Тому результат виконання команди

```
add al, bl          ; al = 09h
```

буде невірний. Щоб отримати правильний результат, у програмі здійснено розширення регістрів до розміру в слово. В такому випадку результат додавання двох чисел розміром у слово буде збережений у регістрі EAX та дорівнюватиме 0FF09h.

Вікна налагоджувача OllyDbg з програмою лістингу 5.2 наведено на рис. 5.2.

Рис. 5.2. Вікна налагоджувача OllyDbg з програмою з лістингу 5.2

Лістинг 5.3. Програма додавання двох чисел розміром у байт. Результат операції зберігається у пам'яті:

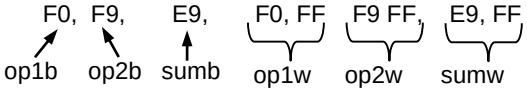
```
        .686                    ; директива определения типа микропроцессора
       .model flat              ; задание линейной модели памяти
.data                           ; директива определения данных
     op1b db -30  ; определение в памяти операнда размером в байт
     op2b db -12  ; определение в памяти операнда размером в байт
     sumb db (0)  ; определение в памяти результата размером в байт
.code                           ; директива начала сегмента команд
  _start:                       ; метка начала программы с именем _start
        mov cl, op1b      ; cl = E2h – занесение в cl
        mov al, op2b      ; bl = F4h – занесение к al
        add al, cl        ; al = D6h – сложение двух операндов
        mov sumb,al             ; сохранение результата в памяти
        ret                     ; возврат управления  ОС
        end _start              ; завершение программы с именем _start
```

У пам'яті дані будуть розміщуватися в такій послідовності:

E2, F4, D6.

Результатом додавання перших двох чисел розміром у байт буде число E9h. *Ніяких переповнень розрядної сітки не буде* тому, що числа в пам'яті комп'ютера подаються у додаткових кодах. А при додаванні в додаткових кодах одиниця переносу, яка може з'явитись, не враховується.

У пам'яті дані будуть розміщуватися в такій послідовності:



Результатом виконання програми буде число D6h (рис. 5.3).

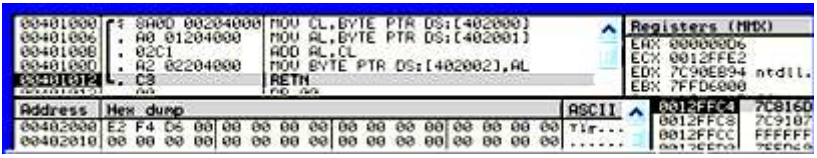


Рис. 5.3. Вікна налагоджувача OllyDbg з програмою з лістингу 5.3

Лістинг 5.4. Програма додавання двох чисел розміром 64 розряди та 80 розрядів. Результат операції зберігається у пам'яті:

```
        title Rysovaniy A.N.
        .686                    ; директива определения типа микропроцессора
        .model flat             ; задание линейной модели памяти
   .data                        ; директива определения данных
        _a dq  0FFFFEEEEDDCCBBAAh       ; 64-разр. операнд в памяти
        _b dt 0AAAABBBBCCCCDDDDEEEEh ; 80-разр. операнд в памяти
        sumML dt 0      ; определение в памяти ячеек под результат
.code                   ; директива начала сегмента команд
  _start:                       ; метка начала программы с именем _start
      mov eax, dword ptr _a     ; занесение младшей части _a
      add eax, dword ptr _b     ; сложение младших частей
      mov dword ptr sumML,eax   ; сохранение младшего результата
      mov eax, dword ptr _a+4   ; занесение старшей части _a
adc eax, dword ptr  _b+4        ; сложение старших частей с учетом CF
      mov dword ptr sumML+4,eax  ; сохранение  старших частей
      mov dx,word ptr _b+8      ; занесение части, которая осталась
      adc dx,0                  ; учет возможного переноса
          mov  word ptr sumML+8,dx;   сохранение последней части
          ret                   ; возврат управления  ОС
          end _start            ; завершение программы с именем _start
```

При виконанні цієї програми спочатку завантажується в eax, завдяки оператору dword ptr, молодша 32-розрядна частина числа _a та за другою командою add eax, dword ptr _b виконується додавання молодших 32-розрядних частин чисел _a та _b. При виконананні команди додавання може встановитися одиниця перенесення з 32-розрядної сітки (або ознака OF, що в цьому випадку однаково).

Старші частини чисел _a та _b завантажуються завдяки зміщенню на 4 байти та обов'язковому указанню розмірності чисел оператором dword ptr. Додавання старших частин чисел відбувається з урахуванням попереднього значення ознаки CF командою **adc** eax, dword ptr _b+4. При виконанні цієї операції також може бути встановлено ознаку CF. Але ця ознака буде врахована в команді **adc** dx,0.

Збереження в пам'яті старших частин результату також відбувається з урахуванням зміщення на 4 байти та указанням оператора розмірності командою mov dword ptr sumML+4,eax.

Через те, що числа мають різну розмірність, залишається ще старша частина числа _b (залишок AAAA), яка має довжину розміром у слово й розташовується в пам'яті, починаючи з 8-го байта, та не враховано можливе переповнення в попередній команді додавання старших частин. Ці дії виконує команда **adc** dx,0, яка виконує додавання найстаршої частини числа _b з нулем (тому що друге число вже повністю оброблено) та ознакою CF.

Результат виконання програми наведено на рис. 5.4, в якому видно, що покрокове виконання зупинено (команда не виконана) на команді **adc** dx,0, а старша частина числа _b (значення AAAA) знаходиться в регістрі DX.

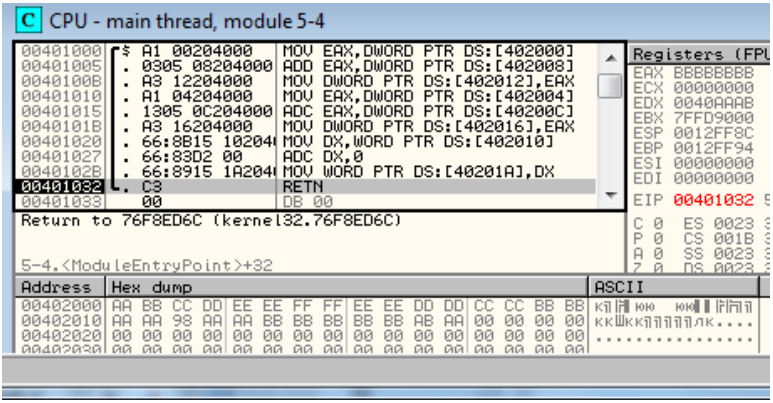


Рис. 5.4. Вікна налагоджувача OllyDbg з програмою з лістингу 5.4

Лістинг 5.5. Програма додавання трьох чисел розміром у байт, слово та подвійне слово кожне *без урахування можливого переповнення* розрядної сітки:

```
        386                 ; директива определения типа микропроцессора
        .model flat         ; задание линейной модели памяти
        .data               ; директива определения данных
        a1 db 12            ; 8-разрядный операнд в памяти
        a2 dw 12            ; 16-разрядный операнд в памяти
        a3 dd 12            ; 32-разрядный операнд в памяти
        res dd 0            ; определение в памяти ячеек под результат
  .code                       ; директива начала сегмента команд
  _start:                     ; метка начала программы с именем _start
        movzx ax,a1         ; увеличение a1 до ax
        add ax,a2           ; ax := a1 + a2
        movzx eax,ax        ; (cwde) - увеличение ax  до eax
        add eax,a3          ; eax := (a1 + a2) + a3
        mov res,eax         ; сохранение результата в памяти
        ret                 ; возврат управления  ОС
        end _start          ; завершение программы с именем _start
```

Особливістю програми є те, що використано команди розширення розмірності регістра AL та AX. Замість цих команд можна використати команду із кодом операції movzx або movsx.

Для додавання трьох чисел розміром у байт кожен з максимальними значеннями фрагмент коду може бути таким:

```
        a1 db 0ffh          ;
        a2 db 255           ;
        a3 db 255           ;
.code
 _st:
        xor eax,eax
        mov al,a1       ; завантаження a1
        add al,a2       ; al := a1 + a2
```

```
adc ah,0          ; ah := ah  + cf + 0
add al,a3         ; al := (a1 + a2) + a3
adc ah,0        ; ah := ah + cf + 0
…
```

В цьому фрагменту додавання трьох чисел виконується в регістрі al, а в регістрі ah накопичуються переповнення. В результаті, в регістрі ax буде сформовано результат виконання додавання трьох чисел.

INC приймач	Додавання одиниці (інкремент)

Команда **INC** збільшує на 1 вміст адресного елементу і розміщає результат за тією самою адресою.

INC mem/reg

```
INC  BL                ; bl := bl + 1 (8-розрядний регістр)
INC  ECX                 ; ecx := ecx + 1 (32-розрядний регістр)
INC  GAMMA[EDI]   ; збільшити на 1 адресу комірки пам'яті
                                  ; з іменем  GAMMA, адреса якої міститься в Рг EDI
```

XADD приймач, джерело	Обміняти між собою та додати	80486

При виконанні команди **XADD** джерелом є завжди регістр, а приймач може бути як регістром, так і змінною, наприклад:

```
.code
mov eax, 1000 0000h
mov ebx, 3000 0000h
xadd eax,ebx              ; eax = 4000 0000h, ebx = 1000 0000h
```

Команди перетворення типів

MOVSX приймач, джерело	Пересилання з розширенням знака

```
MOVSX    r(16,32,64), r/m8;
MOVSX    r(32,64), r/m16;
MOVSXD  r64, r/m32.
```

Команда **MOVSX** копіює вміст джерела (регістр або змінна розміром у байт або слово) в приймач і розширює знак аналогічно командам CBW/CWDE, наприклад:

```
x db -5 ; занесення в комірку пам'яті з іменем "x" числа -5 (x=0000 00FB)
   …
movsx   eax,x ; пересилання з розширенням знака (x=FFFF FFFB)
```

MOVZX приймач, джерело	Пересилання з розширенням нулями

```
MOVZX  r(16,32,64),r/m8;
MOVZX  r(32,64),r/m16.
```

Команда MOV r32, r/m32 заповнює нулями старшу половину r64 (тому аналог MOVSXD не потрібний).

Команда **MOVZX** копіює вміст джерела (регістр або змінна) в приймач (**регістр**) і виконує розширення формату приймача нулями, тобто команда

```
movzx    ax,  bl
```

еквівалентна парі команд

```
mov      al, bl
mov      ah,0
```

CWD	Розширення слова в подвійне слово
CDQ	Розширення EAX до EAX:EDX
CQO	Розширення RAX до RDX:RAX

Команда **CWD** перетворює слово в регістр AX у подвійне слово, молодша половина якого (біти 0–15) залишається в регістрі AX, а старша (біти 16–31) розташовується в регістрі DX.

Команда **CDQ** виконує аналогічну дію що до подвійного слова в регістрі EAX, розширюючи його до збільшеного учетверо слова в регістрах EDX:EAX. Команда **CDQ** корисна при підготовці операції ділення.

Ці команди лише встановлюють всі біти регістра DX або EDX у значення, що дорівнює величині старшого біта регістра AX або EAX, зберігаючи таким чином його знак.

CBW	Розширення байта в слово
CWDE	Розширення слова в подвійне слово
CDQE	Розширення EAX до RAX

Команда **CBW** розширює байт, що знаходиться в регістрі AL, до слова в регістрі AX. Команда **CWDE** розширює слово в регістрі AX до подвійного слова в регістрі EAX. Команди CWDE і CWD відрізняються тим, що CWDE розміщує свій результат в регістрі EAX, тоді як команда CWD, що виконує таку саму дію, розташовує результат в парі регістрів DX:AX. Так само, як і в командах CWD/CDQ, розширення виконується шляхом установлення кожного біта старшої половини результату, що дорівнює старшому біту початкового байта або слова, тобто

```
mov   al,0F5h          ; AL = 0F5h  = 245 = –11
cbw                    ; тепер AX = 0FFF5h  = 65 525 = – 11
```

Команда **CDQE** розширює EAX до RAX.

Команди віднімання

SUB приймач, джерело	Віднімання

Команда SUB (http://www.intel.com/techtrends/technologies/hyperthreading.htm, документ 25366617.pdf) здійснює віднімання другого операнда із першого. Результат записується за адресою першого операнда

SUB mem/reg1, reg2; SUB reg1, mem/reg2

```
SUB   ECX, EBX          ; ECX :=  ECX  – EBX
SUB   EDX, ALEX[ESI] ; відняти з Рг EDX вміст комірки пам'яті
                              ; з іменем ALEX, адреса якої міститься у Рг ESI
```

```
SUB   [EBP+2],  ECX   ; [EBP + 2] := [EBP+2] – ECX
```

SUB mem/reg, data

```
SUB   AL,  10H          ; AL := AL – 10h
SUB   [SI], 0F000H      ; [SI] := [SI] – F000h
```

SBB приймач, джерело	Віднімання з позикою

Команда **SBB** здійснює віднімання джерела від приймача із урахуванням прапорця CF.

SBB mem/reg1, mem/reg2

```
SBB   EAX, EBP              ; EAX := EAX – EBP –  CF
SBB   {EBP + 40H], EBX   ; [EBP + 40H] := [EBP + 40H] – EBX  – CF
```

SBB mem/reg, data

```
SBB   [EDI],  30H     ; [EDI] := [EDI] – 30h –CF
```

Лістинг 5.6. Віднімання з меншого числа більшого:

```
        .686   ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none ; отличие строчных и прописных букв
        include \masm32\include\windows.inc  ; файлы структур, констант …
        include \masm32\include\kernel32.inc  ; системные функции применений…
        include \masm32\include\\user32.inc    ; файлы интерфейса …
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
.data
a1 dd 12
c1 dd 97
     titl db "Вывод через функцию MessageBox",0; название упрощенного окна
     st1 dw ?,0  ; буфер выведения сообщения. dw для небольших чисел
     ifmt  db "Вывод чисел с памяти через MessageBox:", 0dh,0ah,
     "a = 12", 0dh,0ah, 'c = 97', 0dh,0ah,
     "a - c = -%d", 0dh,0ah,0ah,  ; задание превращения символу
     "Автор программы:     , КИТ 14А",0
.code
_start:
        mov eax,a1
        sub eax,c1
        not eax
        inc eax
invoke  wsprintf, ADDR st1, ADDR ifmt, eax
invoke MessageBox, 0, addr st1,addr titl,MB_ICONINFORMATION
invoke ExitProcess, 0 ; возвращение управления  ОС и освобождение ресурсов
        END   _start          ; окончание программы с именем _start
```

При віднаманні з меншого числа більшого отримується додатковий код, який коиандами

```
        not eax
        inc eax
```

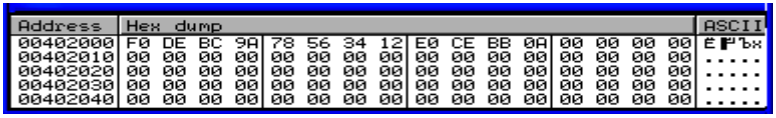
перетворюється в прямий код. Знак числа примусово проставлен у рядку форматного виведення, як = -%d.

Лістинг 5.7. Віднімання багатобайтових чисел різного розміру:

```
        .686                ; директива определения типа микропроцессора
        .model flat         ; задание линейной модели памяти
        .data               ; директива определения данных
        op1 dq 1311768467463790320      ; 123 4567 89ABC DEF0h
        len equ $-op1               ; вычисление количества байтов переменной op1
        op2d dd 180080352       ; 0ABB CEE0h
        op2 dq 0        ;
        rez dq 0        ; резервирование ячейки памяти для переменную rez
.code                   ; директива начала сегмента команд
_start:                 ; метка начала программы с именем _start
   mov ecx, len         ; занос в регистр-счетчик количества байтов op1
   mov eax,op2d                 ; занесение op2 (32 разряда) в Рг EAX
   cdq          ; преобразование двойного слова в 64 разряда – EDX:EAX
mov dword ptr op2,eax       ; сохранение младших разрядов в массиве op2
mov dword ptr op2+4,edx     ; сохранение старших разрядов в массиве op2
xor eax,eax     ; заполнение нулями EAX (сложение “исключающее ИЛИ”)
   clc                      ; сброс флажка CF
     lea esi,byte ptr op1       ; занесення адреси op1 до Рг ESI
     lea edi,byte ptr op2       ; занесення адреси op2 до Рг EBX
     lea ebx,byte ptr rez    ; занесення адреси результату rez до Рг EBX
m1: mov al,[esi]             ; занесення байта з числа op1
   sbb al,[edi]              ; вычитание с учетом признака CF
   mov byte ptr [ebx],al     ; занесение в rez через [EBX] байта результата
     inc esi                 ; увеличение адреса op1
     inc edi                 ; увеличение адреса op2
     inc ebx                 ; увеличение адреса rez
     dec ecx                ; ECX := ECX – 1
     jnz m1                 ; перейти к  m1, если z ≠ 0
     lea eax,rez               ; занесение адреса массива rez
       ret                  ; возврат управления  ОС
       end _start           ; завершение программы с именем _start
```

Оператор equ обчислює різницю між поточним значенням $ і початком того розположення змінної op1 в байтах.

Початковий вигляд вмісту комірок пам’яті в масиві даних наведено на рис. 5.5.



Address	Hex dump												ASCII
00402000	F0	DE	BC	9A	78	56	34	12	E0	CE	BB	0A	00	00	00	00	ЄЮ╝Ъx
00402010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00402020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00402030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00402040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

Рис. 5.5. Початковий вигляд вмісту комірок пам’яті в масиві даних

DEC	Декремент

Команда **DEC** віднімає одиницю з вмісту адресованого елемента і запам’ятовує результат за тією ж адресою.

```
DEC   [EBP] ; зменшити на 1 вміст комірки пам’яті (стека),
                                    ; адреса якої міститься в Рг EBP
DEC   AL   ; зменшити на 1 вміст Рг AL розміром 8 бітів
DEC   EBX           ; EBX := EBX – 1
```

NEG приймач	Зміна знака

Команда **NEG** здійснює віднімання вмісту адресованого елемента з нуля і розміщує результат на тому самому місці (зміни знака для утворення додаткового коду).

NEG reg

CMP приймач, джерело	Порівняння

Команда **CMP** (порівняння) здійснює **віднімання** подібно до команди SUB, установлює біти умов, але не здійснює запам'ятовування результату.

CMP mem/reg1, mem/reg2
CMP mem/reg, data

CMPXCHG приймач, джерело	Порівняти і обміняти між собою

CMPXCHG r/m8,r8; CMPXCHG r/m32,r32;
CMPXCHG r/m16,r16; CMPXCHG r/m64,r64;

Команда **CMPXCHG** порівнює значення, що містяться в регістрах AL/AX/EAX/RAX (залежно від розміру операндів), з приймачем (регістром). Якщо вони рівні, інформація з джерела копіюється в приймач, і прапорець ZF встановлюється в 1, інакше вміст приймача копіюється в регістри AL/AX/EAX/RAX і прапорець ZF встановлюється в 0. Решта прапорців визначається за результатом операції порівняння, як після команди CMP. Джерело – завжди регістр, приймач може бути регістром і змінною.

CMPXCHG8B приймач	Порівняти й обміняти 8 байтів

CMPXCHG8B m64

Команда **CMPXCHG8B** виконує порівняння вмісту регістрів EDX:EAX як 64-бітового числа (молодше подвійне слово – в EAX, старше – в EDX) з приймачем (8-байтова змінна в пам'яті). Якщо вони рівні, вміст регістрів ECX:EBX як 64-бітове число (молодше подвійне слово в EBX, старше – в ECX) поміщається в приймач. Інакше вміст приймача копіюється в EDX:EAX.

CMPXCHG16B m128

Команда **CMPXCHG16B** порівнює пару RDX:RAX з операндом у пам'яті. Якщо вони дорівнюють, то зберігає RCX:RBX в операнді. Інакше завантажує операнд у RDX:RAX.

Команди множення

MUL джерело	Множення чисел без знаків

Розряд-ність	Команда	Множник	Результат
8	MUL r/m8	AL	AX
16	MUL r/m16	AX	DX:AX
32	MUL r/m32	EAX	EDX:EAX
64	MUL r/m64	RAX	RDX:RAX

Команда **MUL** здійснює множення без знака вмісту акумулятора (AL чи AX) і вмісту, що адресується операндом, розміщуючи результат подвійної довжини в акумуляторі і його розширенні (AL і AH – для 8-розрядних операндів, AX і DX – для 16-розрядних). Біти умов CF і OF установлюються, якщо старша половина результату не дорівнює 0.

Множене	**Множник**	**Результат**
AL	оператор_8	AX
AX	оператор_16	DX:AX
EAX	оператор_32	EDX:EAX
RAX	оператор_64	RDX:RAX

MUL reg

```
MUL  ECX     ; EDX, EAX ← ECX × EAX
MUL  BL      ; перемножити вміст регістрів AL  і BL без знаків
                                  ; із запам'ятовуванням результату в Рг AH  та  AL
```

MUL mem

```
MUL mem2 ; перемножити вміст Рг AL/AX/EAX (розмір регістра дорівнює
                  ; розміру комірки пам'яті) і операнда з пам'яті  з іменем mem2
MUL  [EDI – 40H] ; перемножити вміст Рг EAX і операнда
                  ; з пам'яті, адреса якого знаходиться в Рг EDI з урахуванням
; зміщення – 40H  (без знака). Результат запам'ятовується в Рг EDX і EAX
    MUL [0040]          ; DX, AX ← [0040] × AX, DH = 00
```

Як приклад розглянемо виконання операції множення. На першому етапі виконується множення двох операндів op1_w і op2_w, що мають розмірність слова, а результат поміщається в змінну res1_w розмірністю в подвійне слово.

На другому етапі виконується множення операнда op_b розмірністю в 1 байт на op_w, що має розмірність слова, і результат поміщається в двословну змінну res2_w (лістинг 5.8). Для розширення розмірності операнда op_byte використовується команда movzx (Move with Zero-Extend – копіювання з розширенням нуля).

Лістинг 5.8. Перемноження двох пар чисел:

```
      .686              ; директива определения типа микропроцессора
      .model flat       ; задание линейной модели памяти
      option casemap: none   ; отличие строчных и прописных букв
.data                                ; директива определения данных
  op1_w   DW  46          ; первый операнд размером в слово
```

```
   op2_w   DW  65535        ; второй операнд размером в слово
   res1_w  DW 2 dup(0)      ; комірка розміром у 2 слова для res1_w
   op_b   DB  25         ; запись в память с именем op_b 8-разрядного числа 25
   op_w   DW  65244  ; запись в 16-разрядную ячейку памяти числа 65244
   res2_w   DD  0           ; ячейка размером в 2 слова для res2_w
.code                       ; директива начала сегмента команд
_start:                     ; метка начала программы с именем _start
; перемножение op1_w  на  op2_w
     mov AX,op1_w
     mul op2_w                  ; DX,AX := AX × op2_w
     mov word ptr res1_w, AX
     mov word ptr res1_w+2, DX
; перемножение  op_b на  op_w
          movzx AX, op_b            ; беззнаковое расширение операнда op_b
          mul   op_w                ; DX,AX := AX × op_w
          mov   word ptr res2_w, AX
          mov   word ptr res2_w+2, DX
          ret                       ; возврат управления  ОС
        end _start                  ; завершение программы с именем _start
```

Для збереження результату множення обираються коёмірки в два рази більші, ніж розмірність більшого операнда в парі чисел. Стан програми перемножения чисел різної розмірності після її виконання наведено на рис. 5.6.

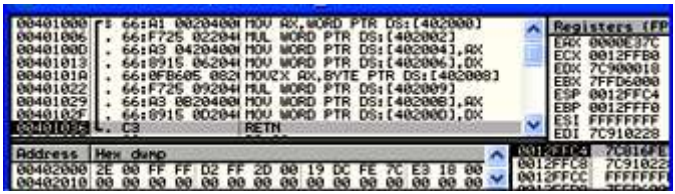


Рис. 5.6. Стан програми перемножения чисел різної розмірності після її виконання

IMUL джерело **IMUL приймач, джерело** **IMUL приймач, джерело, const**	Множення чисел без знаків	8086 80386

IMUL r16, r/m16	IMUL r16,imm(8,16)	IMUL r16,r/m16,imm(8,16)
IMUL r32, r/m32	IMUL r32,imm(8,32)	IMUL r32,r/m32,imm(8,32)
IMUL r64, r/m64	IMUL r64,imm(8,32)	IMUL r64,r/m64,imm(8,32)

Розрядність	Команда	Множник	Результат
8	IMUL r/m8	AL	AX
16	IMUL r/m16	AX	DX:AX
32	IMUL r/m32	EAX	EDX:EAX
64	IMUL r/m64	RAX	RDX:RAX

Команда **IMUL** подібна до команди MUL, але за нею виконується множення з урахуванням знака. Біти умов CF і OF установлюються, якщо старша

половина результату не є розширенням для формування знака молодшої половини.

IMUL reg/ mem

```
IMUL  ECX        ; EDX, EAX =  EAX × ECX зі знаками
IMUL ALPHA       ; EDX, EAX = EAX × [ALPHA] з урахуванням знаків
```

Команда **IMUL** має форми, що розрізняються числом операндів:

1. IMUL джерело – джерело (регістр або змінна) помножується на AL/AX/EAX (залежно від розміру операнда), і результат розташовується в DX:AX або EDX:EAX відповідно.

2. **IMUL приймач, джерело** – джерело (число, регістр або змінна) помножується на приймач (регістр), і результат заноситься в приймач:

```
imul   eax,25     ;  eax := eax x 25
```

3. **IMUL приймач, джерело, const** – перемножить другий і третій операнди, а результат збереже в перший операнд. Перший операнд – тільки регістр, другий може бути будь-якого типу, а третій повинен бути тільки безпосереднім значенням:

```
imul   eax,ebx,8     ;  eax := ebx x 8
```

Значення прапорців SF, ZF, AF і PF після виконання команди IMUL не визначені.

Розглянемо приклад множення однобайтового від'ємного операнда на подвійне слово (лістинг 5.9).

Лістинг 5.9. Множення однобайтового від'ємного операнда на подвійне слово:

```
    .686          ; директива определения типа микропроцессора
    .model flat                    ; задание линейной модели памяти
    option casemap: none      ; отличие строчных и прописных букв
.data                          ; директива определения данных
       op_b  DB  -46               ; -46 = D2h – один байт
       op_dword  DD 678        ; 678 = 2A6h – четыре байта
       op_rez  DQ  0                ; десять байтів з нульовим вмістом
.code                                ; директива начала сегмента команд
_start:                              ; метка начала программы с именем _start
       movsx AX,op_b             ; AX := FFD2 – знаковое расширение
       movsx EAX,AX          ; EAX := FFFF FFD2 – знаковое расширение
       mov   EBX,op_dword    ; EBX := 0000 02A6
       imul  EBX                              ; EDX,EAX := EAX × EBX
       mov   dword ptr op_rez,EAX
       mov   dword ptr op_rez+4,EDX
       lea   EAX,op_rez             ; сохранение адреса результата
       ret                                ; возврат управления  ОС
       end _start                   ; завершение программы с именем _start
```

Стан програми перемноження знакових чисел різної розмірності після її виконання наведено на рис. 5.7.

У цьому прикладі задіяно два операнди: op_b розміром в 1 байт і двослівна змінна op_dword. Результат операції поміщається в 8-бітову змінну

op_rez. Перед виконанням множення необхідно розширити розмірність операнда op_b до подвійного слова. Для цього використовуються дві команди movsx (Move with Sign Extend – копіювання з розширенням знака).

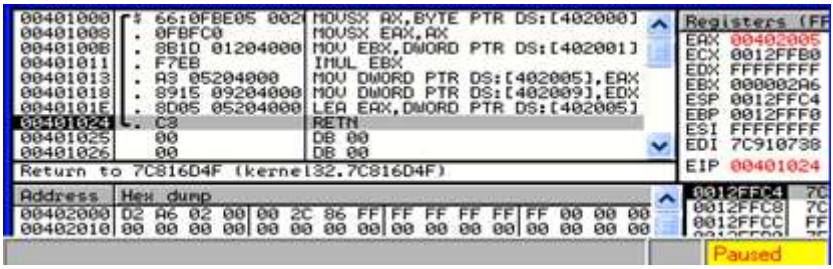


Рис. 5.7. Стан програми перемноження знакових чисел різної розмірності після її виконання

Перша команда movsx поміщає 8-бітовий операнд у 16-розрядний регістр AX, розширюючи знак на старшу половину AX (регістр AH). Друга команда movsx перетворює 16-розрядне значення в 32-розрядне і поміщає його в регістр EAX. Можна виконати й одну команду

movsx EAX,op_b

Якщо множене і множник мають однаковий знаковий біт, то команди mul і imul генерують однаковий результат. Але якщо співмножники мають різні знакові біти, то результатом виконання команди mul буде додатне число, тоді як команда imul дасть від'ємне значення.

При розв'язанні практичних задач дуже часто потрібно множити операнди, що складаються з декількох байтів або слів. Один із простих варіантів множення великих чисел припускає множення парами окремих слів і складання одержаних результатів. Алгоритм цієї процедури нагадує множення стовпчиком при знаходженні вмісту десяткових чисел.

Команди ділення

DIV джерело	Цілочислове ділення чисел без знаків

Розрядність	Команда	Ділене	Частка	Результат
8	DIV r/m8	AX	AL	AH
16	DIV r/m16	DX:AX	AX	DX
32	DIV r/m32	EDX:EAX	EAX	EDX
64	DIV r/m64	RDX:RAX	RAX	RDX

Команда **DIV** здійснює ділення без знака вмісту акумулятора і його розширення (AL і AH – для 8-розрядної операції, DX і AX – для 16-розрядної, EDX і EAX – для 32-розрядної) на вміст, що адресується операндом, і розміщує частку в акумуляторі (AL/AX/EAX) та залишок – в AH/DX/EDX. Біти умов не змінюються. Ділення на нуль дає переривання за вектором 0.

Ділене	Дільник	Частка	Остача
AX	оператор_8	AL	AH
DX:AX	оператор_16	AX	DX
EDX:EAX	оператор_32	EAX	EDX
RDX:RAX	оператор_64	RAX	RDX

```
DIV 8H      ; поділити вміст Рг AX на 8H без урахування знака.
                    ; результат формується у Рг AL, а залишок – у Рг AH.
DIV [BP + 20H]  ; поділити вміст регістрів DX  і  AX на число,
                ; адреса якого знаходиться в Рг BP з урахуванням зміщення + 20H.
```

Перед виконанням кожної операції ділення необхідно обнулити регістр edx. Інакше в регістрі edx виконуватиметься додавання залишків від попередньої операції ділення. Розглянемо приклад, в якому отримується **невірний** результат при виконанні двох операцій ділення в програмі:

```
  _b dd 3 ; занесення дільника
.code
start:
 mov eax,1 ; занесення діленого
  mov edx,0 ; підготування регістра частки
   div _b      ; edx,eax/ _b      (результат: eax = 0, edx = 1)
   mov eax,1 ; занесення нового діленого
   div _b      ;                        eax = 0, edx = 2
  mov ecx,edx ; збереження частки від ділення
```

В результаті виконання цього прикладу за останьою командою ділення регістр edx = 2. Це відбулося тому, що перед останнім діленням регістр збереження частки edx не було заповнено нулями (необхідно вставити команду mov edx,0).

IDIV джерело	Цілочислове ділення чисел зі знаком

Команда IDIV працює подібно до команди DIV, за винятком того, що вона здійснює ділення з урахуванням знака.

Для ділення беззнакових даних в мікропроцесорах Intel використовується команда div, а для ділення знакових – idiv. Залежно від розмірності операндів існують такі формати операції ділення:

- Ділення слова на байт. Ділене знаходиться в регістрі AX, а дільник – у байті пам'яті або в однобайтовому регістрі. Після ділення остача поміщається в регістр AH, а частка – в AL. Операція з даними типами операндів має обмежене застосування через невеликий діапазон допустимих значень (однобайтова частка не перевищує +255 – для беззнакового ділення і +127 – для знакового);
- Ділення подвійного слова на слово. Ділене знаходиться в регістровій парі DX:AX, а дільник – у слові пам'яті або в регістрі. Після ділення остача

поміщається в регістр DX, а частка – в регістр AX. Частка в одному слові допускає максимальне значення +32 767 – для беззнакового ділення і +16 383 – для знакового;

- Ділення збільшеного учетверо слова на подвійне слово. Ділене знаходиться в регістровій парі EDX:EAX, а дільник – у подвійному слові пам'яті або в регістрі. Після ділення остача поміщається в регістр EDX, а частка – в регістр EAX.

Команди div і idiv мають єдиний операнд, що є дільником. Якщо при діленні ділене менше, ніж дільник, може здатися, що регістр EDX не потрібен. Однак це не так. Якщо в такому випадку регістр EDX не ініціювати (записати хоча б нулі), то ділення не виконуватиметься.

Як приклад розглянемо ділення 64-розрядного беззнакового числа на однобайтове ціле. Програму ділення беззнакового числа на байт наведено в лістингу 5.10.

Лістинг 5.10 Ділення беззнакового числа на байт:

```
    .686                ; директива определения типа микропроцессора
    .model flat         ; задание линейной модели памяти
    option casemap: none    ; отличие строчных и прописных букв
.data                       ; директива определения данных
    div1  DQ 1234  ; 1234 = 4D2h
div2 DB 74      ; запись в 8-разр. ячейку памяти с именем div2 числа 74 = 4Ah
    res1 DD 0       ; резервирование ячейки для сохранения целой части
    res2 DD 0       ; резервирование ячейки для сохранения остатка
.code                       ; директива начала сегмента команд
 _start:                    ; метка начала программы с именем _start
    movzx EBX, div2             ; EBX = 0000 004Ah
    mov  EAX, dword ptr div1    ; EAX = 0000 04D2h
    mov  EDX, dword ptr div1+4  ; EDX = 0000 0000h
    div  EBX                    ; EDX,EAX = EAX/EBX
    mov  res1, EAX          ; сохранение целой части результата
    mov  res2, EDX          ; сохранение остатка результату
    ret                     ; возврат управления  ОС
    end _start              ; завершение программы с именем _start
```

Стан програми ділення беззнакових чисел різної розмірності після її виконання показано на рис. 5.8.

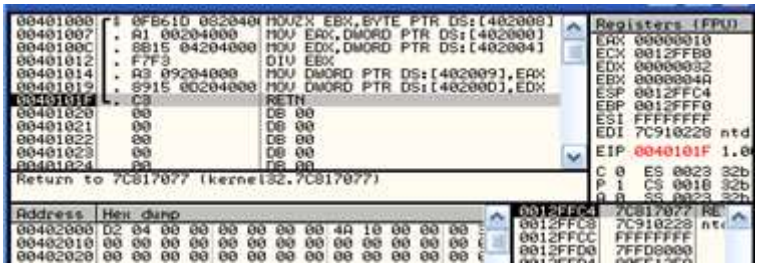


Рис. 5.10. Стан програми ділення беззнакових чисел різної розмірності з лістингу 5.10 після її виконання

У програмі з лістингу 5.9 ділене знаходиться в змінній div1, а дільник – у змінній div2.

Готуємо операцію ділення. Для цього поміщаємо молодше подвійне слово діленого в регістр EAX, а старше подвійне слово змінної – в регістр EDX. Нарешті, після виконання операції ділення зберігаємо частку в змінній res1, яка дорівнює 10h, а остачу – у змінній res2.

Ділення знакових чисел виконується за допомогою команди idiv. Модифікуємо попередній приклад. Для цього потрібно замінити команду div на команду idiv і використовувати замість команд movzx команди movsx.

Текст програми показано в лістингу 5.11.

Лістинг 5.11. Ділення 64-розрядного знакового числа на байт:

```
      .686                        ; директива определения типа микропроцессора
      .model flat                 ; задание линейной модели памяти
      option casemap: none        ; отличие строчных и прописных букв
    .data                         ; директива определения данных
      div1  DQ -1234              ; -1234 = FFFF FFFF FFFF B2Eh
      div2  DB 74                 ; 74 = 4Ah – один байт
    res1 DD 0           ; резервирование ячейки для сохранения целой части
    res2 DD 0           ; резервирование ячейки для сохранения остатка
.code                             ; директива начала сегмента команд
  _start:                         ; метка начала программы с именем _start
    movsx BX, div2                      ; BX = 004Ah
    movsx EBX, BX                       ; EBX = 0000 004Ah
    mov  EAX, dword ptr div1            ; EAX = FFFF FB2Eh
    mov  EDX, dword ptr div1+4          ; EDX = FFFF FFFFh
    idiv  EBX                           ; EDX=FFFFFFCEh, EAX=FFFF FFF0h
    mov  res1, EAX                      ; сохранение целой части результата
    mov  res2, EDX                      ; сохранение остатка результата
      ret                       ; возврат управления  ОС
      end  _start               ; завершение программы с именем _start
```

Стан комірок пам’яті до початку виконання програми з лістингу 5.11 наведено на рис. 5.9.

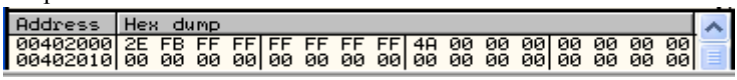


Рис. 5.9. Стан комірок пам’яті до початку виконання програми

Якщо ділене і дільник мають однаковий знаковий біт, то команди div і idiv генерують однаковий результат. Але якщо ділене і дільник мають різні знакові біти, то команда div генерує додатну частку, а команда idiv – від’ємну.

У деяких випадках можна добитися підвищення продуктивності при виконанні операцій ділення на степінь числа 2 (2, 4 і т.д.). В цьому випадку ефективнішим є зсув управо на необхідну кількість бітів.

Припускаємо, що ділене знаходиться в регістрі AX.

Ділення на 2:
```
shr  AX,1
```
Ділення на 8:
```
mov  CL,3
shr  AX,CL
```

При використанні команд div і idiv може виникнути переповнення, що спричиняє переривання. Подібна ситуація може трапитися при діленні на нуль.

Команди двійково-десяткової арифметики

AAA, DAA – корекція після додавання;

AAS, DAS – корекція після віднімання;

AAM – корекція після множення;

AAD – корекція після ділення.

У 64- розрядному режимі ці команди не підтримуються.

AAA	ACSII– корекція після додавання

Команда **AAA** корегує суму двох неупакованих двійково-десяткових чисел у регістрі AL. Якщо корекція приводить до десяткового перенесення, регістр AH збільшується на 1. Цю команду краще використовувати відразу після команди складання двох таких чисел. Наприклад, якщо при складанні 05 і 06 в регістрі AX опиниться число 000Bh, то команда AAA скорегує його в 0l0lh (неупаковане десяткове 11). Прапорці CF і OF встановлюються в 1, якщо відбулося перенесення з регістра AL в регістр AH, інакше вони дорівнюють нулю. Значення прапорців OF, SF ZF і PF не визначено.

DAA	BCDI– корекція після додавання

Команда **DAA** здійснює корекцію числа в регістрі AL, отриманого в результаті додавання двох упакованих десяткових операндів. При цьому маємо упаковану десяткову суму.

Якщо ця команда виконується відразу після команди ADD (ADC, INC або XADD) і в регістрі AL знаходиться сума двох упакованих двійково-десяткових чисел, то в регістр AL записується упаковане двійково-десяткове число, яке повинне було стати результатом складання. Наприклад, якщо регістр AL містить число 19h, то послідовність команд

```
inc  al
daa
```

приведе до того, що в AL опиниться 20h (а не 1Ah, як було б після команди INC).

Команда DAA виконує такі дії:

1. Якщо молодші чотири біти регістра AL більші від 9 або прапорець AF = 1, то регістр AL збільшується на 6; прапорець CF встановлюється, якщо при цьому додаванні відбулося перенесення, і прапорець AF встановлюється в 1.

2. Інакше прапорець AF = 0.

3. Якщо тепер старші чотири біти регістра AL більші від 9 або прапорець CF = 1, то регістр AL збільшується на 60h і прапорець CF встановлюється в 1.

4. Інакше прапорець CF = 0.

Прапорці AF і CF встановлюються, якщо під час корекції відбувалося перенесення з першої або другої цифри результату. Прапорці SF, ZF і PF встановлюються відповідно до результату, прапорець OF не визначений.

AAS	ACSII – корекція після віднімання

Команда **AAS** здійснює корекцію числа в регістрі AL, отриманого в результаті віднімання двох розпакованих десяткових операндів, і дає в результаті розпаковану десяткову різницю.

DAS	BCD – корекція після віднімання

Команда **DAS** здійснює корекцію результату віднімання двох упакованих десяткових чисел і в результаті дає упаковану десяткову різницю.

Якщо ця команда виконується відразу після команди SUB (SBB або DEC) і в регістрі AL знаходиться різниця двох упакованих двійково-десяткових чисел, то в AL записується упаковане двійково-десяткове число, яке повинне було бути результатом віднімання. Наприклад, якщо AL містить число 20h, послідовність команд

```
dec  al
das
```

приведе до того, що в регістрі опиниться число 19h (а не 1Fh, як було б після команди DEC).

Команда DAS виконує такі дії:

1. Якщо молодші чотири біти регістра AL більші від 9 або якщо прапорець AF = 1, то регістр AL зменшується на 6, прапорець CF встановлюється, якщо при цьому відніманні відбулася позика, і прапорець AF встановлюється в 1.
2. Інакше AF = 0.
3. Якщо тепер старші чотири біти AL більші від 9 або прапорець CF = 1, то AL зменшується на 60h і CF встановлюється в 1.
4. Інакше CF = 0.

Приклад незвичайного використання цієї команди – найкомпактніший варіант перетворення шістнадцяткової цифри в ASCII-код відповідного символу:

```
cmp   al, 10
sbb   al, 69h
das
```

Після команди SBB числа 0–9 перетворюються в 96H – 9Fh, а числа 0Ah – 0Fh – в 0A1h – 0A6h. Потім команда DAS віднімає 66h з першої групи чисел, переводячи їх в 30h – 39h, і число 60h з другої групи чисел, переводячи їх в 41h – 46h.

Прапорці AF і CF встановлюються, якщо під час корекції відбувалася позика з першої або другої цифри. Прапорці SF, ZF і PF встановлюються відповідно до результату, прапорець OF не визначений.

AAM	ACSII– корекція після множення

Команда **AAM** корегує результат множення неупакованих двійково-десяткових чисел, який знаходиться в регістрі AX після виконання команди MUL,

перетворюючи одержане в пару неупакованих двійково-десяткових чисел (у AH і AL). Наприклад:

```
mov   al,5
mov   bl,5   ; bl ← 5
mul   bl     ; AH,AL ← al × bl
aam          ; AX містить 0205h.
```

Команда AAM встановлює прапорці SF, ZF і PF відповідно до результату і залишає прапорці OF, AF і CF невизначеними.

Код команди AAM – D4h 0Ah, де 0Ah – основа системи числення, по відношенню до якої виконується корекція. Цей байт можна замінити на будь-яке інше число (окрім нуля), і AAM перетворить AX на дві неупаковані цифри будь-якої системи числення. Така узагальнена форма AAM працює на всіх мікропроцесорах (починаючи з 8086), але з'являється в документації Intel тільки з мікропроцесорів Pentium. Фактично дія, яку виконує команда AAM, – цілочислове ділення AL на 0Ah (або будь-яке інше число в загальному випадку), частка поміщається в AH і остачу – в AL, тому цю команду часто використовують для швидкого ділення у високооптимізованих алгоритмах.

AAD	ACSII– корекція перед діленням

Команда **AAD** здійснює корекцію діленого в регістрі перед діленням двох розпакованих десяткових операндів, тому, що в результаті отримуємо розпаковану десяткову частку.

Команда **AAD** виконує корекцію неупакованого двійково-десяткового числа, що знаходиться в регістрі AX, тому подальше ділення привело до десяткового результату. Наприклад, розділимо десяткове 25 на 5:

```
mov   ax,0205h ;   25 в неупакованому форматі
mov   bl,5
aad            ;  Тепер в AX знаходиться  19h
div   bl       ;  AX = 0005
```

Прапорці SF, ZF і PF встановлюються відповідно до результату, прапорці OF, AF і CF не визначені.

Команда AAD, як і команда AAM, використовується з будь-якою системою числення: її код – D5h 0Ah, і другий байт можна замінити на будь-яке інше число. Дія команди AAD полягає в тому, що вміст регістра AH помножується на другий байт цієї команди (0Ah за умовчанням) і складається з AL, після чого AH заповнюється нулями, тому AAD можна використовувати для швидкого множення на будь-яке число.

5.2. Лабораторна робота «Програмування арифметичних операцій»

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм, написаних мовою асемблера для програмування арифметичних виразів для МП платформи x86.

Вхідний контроль знань

Написати та налагодити програму згідно з одержаним варіантом задання. Результат записати у пам'ять.

1. Скласти день, місяць та рік свого народження.
2. Перемножити день, місяць та рік свого народження.
3. Поділити рік свого народження на місяць народження.
4. Виконати зсув дня свого народження ліворуч на 2 розряди.
5. Виконати зсув дня свого народження ліворуч на 4 розряди.
6. Виконати зсув року свого народження праворуч на 2 розряди.
7. Виконати операцію логічного додавання дня та року свого народження.
8. Виконати операцію логічного перемноження місяця та року свого народження.
9. Виконати операцію логічного додавання місяця та року свого народження.
10. Виконати операцію логічного порівняння місяця та року свого народження.

Постановка задачі

Згідно з номером у групі вибрати варіант та написати на асемблері програму обчислення одного з виразів:

1. $a + a/b - e/c$;	11. $a + 11de - e/b$;	21. $(de + 21e)/(c - a)$;
2. $a + 2d - d/b$;	12. $(b + 12de)/(a - c)$;	22. $(a + 22e)b/(a + c)$;
3. $3a/d - e/b$;	13. $ae + 13b - d/c$;	23. $a + 23ac - e/b$;
4. $4e/b - d/b$;	14. $(ab + 14d)/(d - b)$;	24. $(a + 24b) - d/c$;
5. $a + 5de - e/b$;	15. $(de + 15d)/(c + a)$;	25. $a + 25db - e/b$;
6. $(b + 6de)/(a/c)$;	16. $(a + 16e)b/(e - c)$;	26. $(b + 26de)/(a - b)$;
7. $ae + 7b - d/c$;	17. $a + 17ab - d/b$;	27. $ae + 27b - d/a$;
8. $(ab + 8d)/(b - d)$;	18. $(a + 18e) + d/a$;	28. $(ab + 28d)/(b - a)$;
9. $(de + 9e)/(c - a)$;	18. $c + 19bd - d/b$;	29. $(de + 29a)/(c - a)$;
10. $(a + 10e)b/(a + c)$,	20. $(a + 20e)a - d/b$;	30. $(a + 30d)e/(a - c)$.

де a – номер у списку; b – номер навчальної групи; c – pp – дві останні цифри року народження; e – $pppp$ – чотири цифри року народження; d – *ддмм*: *дд* – день народження; *мм* – місяць народження.

Результат обчислення виразу зберегти в пам'яті. Навести значення та порядок розміщення даних у пам'яті.

Потрібно пам'ятати, що при виконанні операції ***ділення*** *в регістрі EDX зберігається не дрібна частина результату, а остача від цілочислового ділення, що не є таким самим.*

Тому дробова частина результату виконання операції не відповідатиме дійсному значенню.

Зміст звіту

1. Постановка задачі для конкретного варіанта.

2. Лістинг програми з детальним коментарем та описом роботи.

3. Print screen екрана 32-розрядного налагоджувача з виконаною програмою.

4. Короткий опис виконання програми.

5. Висновки за результатами роботи.

Приклад 5.1. Написати на асемблері програму обчислення виразу

$$(a + d – ed)/a,$$

де *a* – номер в списку; *b* – номер навчальної групи; *c* – *pp* – дві останні цифри року народження; *e* – *pppp* – чотири цифри року народження; *d* – *ддмм*: *дд* – день народження; *мм* – місяць народження.

Результат обчислення виразу зберегти в пам'яті. Навести значення та порядок розміщення даних у пам'яті.

Приклад використання формату даних, збільшеного учетверо, наведено у лістингу 5.12. В цій програмі операцію віднімання виконано у додаткових кодах. Тому **операцію віднімання замінено на операцію додавання в додаткових кодах.** Результат обчислення виразу теж наведено у додаткових кодах.

Лістинг 5.12. Приклад використання формату даних, збільшеного учетверо:

```
        title CopyRight by Rysovaniy A. N.    (e + d – ed)/a
        .686                    ; директива определения типа микропроцессора
       .model flat              ; задание линейной модели памяти
     .data                      ; директива определения данных
        a dd 6                      ;  a = 00000006h
        d dd 2112                   ;  d = 00000840h
        e dd 1989                   ;  e = 000007C5h
    rez1 dd 0                       ;  rez1 = 00000000h
    rez2 dd 0                       ;  rez2 = 00000000h
        .code                       ; (e + d – ed)/a – начало сегмента команд
_start:                         ; метка начала программы с именем _start
     ; e*d
        mov eax,d               ; eax = 00000840h
        imul e                  ; edx = 00000000h, eax = 00401940h
     ; – e*d  (преобразование в дополнительный код)
        not eax                 ; инверсия eax = FFBFE6BFh
        not edx                 ; инверсия edx = FFFFFFFFh
        inc eax                 ; eax = FFBFE6C0h
        adc edx,0               ; edx = FFFFFFFFh  (CF=0)
     ; + d – e*d
        add eax,d               ; eax = FFBFEF00h
        adc edx,0               ; edx = FFFFFFFFh  (CF=0)
     ;e + d – e*d
        add eax,e               ; eax = FFBFF6C5h
        adc edx,0               ; edx = FFFFFFFFh  (CF=0)
     ; (e+d-e*d)/a
        idiv a                      ; edx = FFFFFFFDh,  eax = FFF553CCh
     mov rez1,eax               ; rez1 = FFF553CCh – частное
     mov rez2,edx               ; rez2 = FFFFFFFDh – остаток
        ret                     ; возврат управления  ОС
        end _start              ; завершение программы с именем _start
```

Порядок розміщення даних у пам’яті та стан налагоджувача OllyDbg з програмою лістингу 5.12 наведено на рис. 5.10.

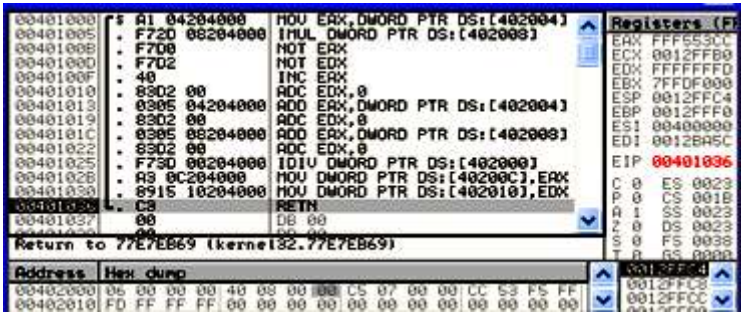


Рис. 5.10. Порядок розміщення даних у пам’яті

Через те, що значення виразу (e+d-e*d) при таких початкових даних повністю розміщується у регістрі eax та не «переходить» у регістр edx, програма стає достатньо простою. А якщо ні – потрібно виконувати спочатку ділення регістра edx, потім остачц пристиковувати до регістра eax та знову виконувати ділення (ділення у стовпчик).

Для одержання результату в **прямому** коді необхідно провести зворотне перетворення вмісту комірок пам’яті з додаткового коду.

Приклад 5.2. Написати на асемблері програму обчислення виразу: $(b+12ed)/(a-c)$:

Лістинг 5.13. Програма виконання прикладу 5.2:

```
.686   ; директива определения типа микропроцессора
.model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
option casemap:none ; отличие строчных и прописных букв
include \masm32\include\windows.inc  ; файлы структур, констант …
include \masm32\include\kernel32.inc  ; системные функции применений…
include \masm32\include\\user32.inc    ; файлы интерфейса …
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
.data
    a1 dd 12
    b1 dd 14
    c1 dd 97
    d1 dd 310
    e1 dd 1997
      temp3 dd ?
      res   dd ?
      rsign db ?
    titl db "Вывод через функцию MessageBox",0; название упрощенного окна
    st1 dw ?,0  ; буфер выведения сообщения. dw для небольших чисел
    ifmt  db "Вывод чисел с памяти через MessageBox:", 0dh,0ah,
    "result = -%d", 0dh,0ah,0ah,  ; задание превращения символу
    "Автор программы:        , группа КИТ ",0
.code
_start:
      mov eax,a1
```

```
        imul eax,310
        imul e1
        add eax,b1
        mov res,eax
        mov eax,a1
        sub eax,c1
        not eax
        inc eax
        mov temp3,eax
        mov eax,res
        idiv temp3
invoke  wsprintf, ADDR st1, ADDR ifmt, eax
invoke MessageBox, 0, addr st1,addr titl,MB_ICONINFORMATION+180000h
invoke ExitProcess, 0 ; возвращение управления  ОС и освобождение ресурсов
        END   _start          ; окончание программы с именем _start
```

Приклад 5.3. Написати на асемблері програму обчислення виразу *(a + b)c – d/c*, де *a* = 5; *b* = 16; *c* = 81; *d* = 2209.

Лістинг 5.14. Програма обчислення прикладу 5.3 з використанням змінних розміром у байт:

```
      .686                ; директива определения типа микропроцессора
        .model flat       ; задание линейной модели памяти
.data                   ; директива определения данных
_a db 5           ; запись в 8-разрядную ячейку памяти с именем _a числа 5
     _b db 16             ; _b = 10h
     _c db 81             ; _c = 51h
     _d dw 2209            ; запись в 16-разрядную ячейку памяти числа _d = 8A1
      rez1 dw 0           ; резервирование памяти для сохранения переменной rez1
      rez2 dw 0           ; резервирование памяти для сохранения переменной rez2
.code                       ; директива начала сегмента команд
_start:                     ; метка начала программы с именем _start
      mov al,_a                     ; al = 05h
      add al,_b                     ; al = 5h + 10h = 15h        ;  – (a + b)
      mul _c                        ; ax = 15h × 51h = 06A5h ;   – (a + b)c
      push ax                       ; стек = XXXX06A5h
      mov ax,_d                     ; ax = 08A1h
      div _c                        ; ah,al = ax/_c = 16 1Bh    ;   –  d/c
      movzx bx,ah                   ; bx = 0016h – остаток
      xor cx,cx                     ; cx = 0
      sub cx,bx                     ; cx = cx – bx = FFEAh – остаток
      mov rez1,cx                   ; сохранение остатка
      pop cx                        ; cx = 06A5h  – восстановление (a + b)c
      movzx ax,al          ; al = 001Bh – выравнивание формата
      sbb cx,ax            ; cx = 06A5h – 001Bh – 1 = 0689h – целая часть
      mov rez2,cx          ; сохранение целой части результата
      ret                  ; возврат управления  ОС
        end _start         ; завершение программы с именем _start
```

Порядок розміщення даних у пам’яті та стан налагоджувача OllyDbg програми наведено на рис. 5.12. Результатом виконання програми буде число 0689.FFEA, де 689h – ціла частина, а FFEAh – остача (не дрібна частина) результату.

Для копіювання активного вікна програми, наприклад програми OllyDbg, потрібно натиснути Alt + PrtScr.

Збереження тимчасових результатів частини виразу (a + b)c виконується в стеку, хоча для цього можна використовувати як один із вільних регістрів, так і комірку пам'яті.

Приклад 5.4. Написати на асемблері програму обчислення виразу e/3c + 5ac, враховуючи можливе перенесення результату.

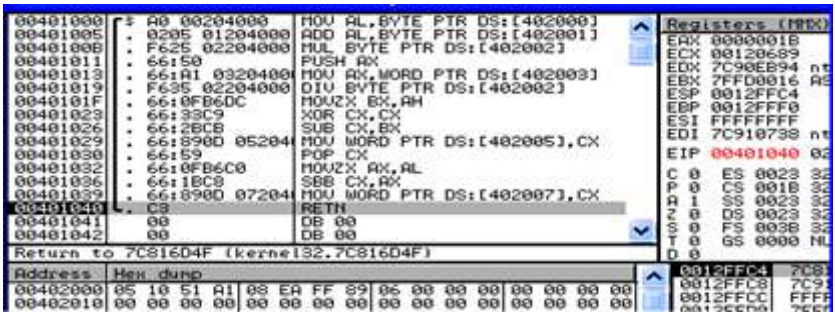


Рис. 5.12. Стан налагоджувача OllyDbg з програмою прикладу 5.4

Лістинг 5.15. Програма обчислення прикладу 5.4 з використанням можливого перенесення результату:

```
        ; e/3*c + 5*a*c = 1993/3*93 + 5*5*93
        .686            ; директива определения типа микропроцессора
        .model flat  ; задание линейной модели памяти
.data           ; директива определения данных
 _e dd 1993
 _c dd 93
 _a dd 5
res1 dd ? ; определение в памяти ячейки под младшую часть
res2 dd ?       ; ячейка для сохранения старшей части результата
.code           ; директива начала сегмента команд
_start:         ; метка начала основной части программы
    mov eax, _c             ; eax:=_c
    mov ebx, _a             ; ebx:=_a
    mul ebx      ; edx:eax=_c*_a (ст. часть в EDX, младшая – в EAX)
    mov ebx, 5              ; ebx:=5
    mul ebx                 ; edx:eax:=_c*_a*5
    mov esi, edx             ; сохранение старшей части
    mov edi, eax            ; сохранение младшей части
    mov eax, _c             ; eax:=_c
    mov ebx, 3              ; ebx:=3
    mul ebx                 ; edx:eax:=_c*3
    mov ebx, eax     ; ebx:=_c*3; сохранение промежуточного результата в EBX
    mov eax, _e             ; eax:=_e
    mov edx, 0    ; edx:=0, заполнение нулями для последующего деления
    div ebx         ; edx:eax:=_e/(_c*3)( частное ->EAX, остаток ->EDX)
    add eax, edi            ; eax:=93*5*5 + 1993/(93*3)
    jc m1                   ; перейти, если есть перенесение
    jmp m2                  ; безусловный переход на сохранение результата
m1:     inc esi             ; добавление единицы переноса к старшей части
m2:     mov res1, eax    ; сохранение в памяти младшей части
   mov res2, esi          ; сохранение в памяти старшей части
   ret              ; возвращение управления ОС
```

```
end _start      ; завершение программы с именем  _start
```

Через те, що результат множення 5*93 повністю розміщується у регістрі EAX та не переходить у регістр EDX (старша частина результату), то для спрощення програми у наступній дії він не враховується. Значення 5ac (5*93*5) також повністю розміщується в EAX, однак для наочності значення регістра EDX зберігається в ESI (у нашому випадку 00000000), та використовується при подальшому обчисленні.

При виконанні команди **add eax, edi** якщо є перенесення, враховується одиниця переносу, додаючи її до старшої частини результату, яка зберігається у ESI командою **inc esi.**

5.3. Додатковий матеріал про налагоджувач OLLYDBG

Для ***виправлення команд в exe-файлі*** у налагоджувачі OllyDbg необхідно виконати таке:

– скопіювати програму в нове вікно для корекції. Для цього натиснути праву кнопку миші у вікні дизасемблера і вибрати у нижній частині висхідного меню пункт Copy to execution / Selection. У результаті весь дизасембльований модуль разом із виправленими командами буде скопійований у нове вікно;

– вказати ім'я нового файлу. Для цього клацнути правою кнопкою миші по новому вікну і вибрати пункт Save file та ввести ім'я нового exe-файлу;

– виконати необхідні зміни в програмі. Для цього підвести курсор до команди, яку необхідно змінити, клацнути на ній, натиснути на праву кнопку миші та обрати пункт Assemble;

– зберегти змінений файл. Для цього натиснути на праву кнопку миші та обрати Copy/To file.

ТОЧКИ ЗУПИНУ

Звичайні точки зупину

Для того щоб поставити *звичайні точки зупину* (ordinary breakpoints), необхідно у вікні дизасемблера використовувати клавішу <F2> або подвійне клацання миші в другій колонці вікна коду. У цьому випадку команда забарвлюється в червоний колір. Цей вид зупину необхідний Для того щоб у точці зупину можна було б перевірити стан регістрів, змінних, стека. Вторинне натиснення на клавішу <F2> в точці зупину або подвійне клацання миші видаляють точку зупину.

Умовні точки зупину

Умовні точки зупину (conditional breakpoints) необхідні Для того щоб здійснити зупинку, якщо виконується необхідна умова. Умовні точки зупину встановлюються натисненням комбінації клавіш <Shift>+<F2>. При цьому з'являється вікно з комбінованим списком, куди можна занести точку зупину, наприклад, із такою умовою:

- EAX == 5 – зупинка на відміченій команді (перед її виконанням) буде здійснена, якщо вміст регістра EAX дорівнюватиме 5;
- EAX = 0 AND ECX > 6 – зупинка на відміченій команді буде здійснена, якщо вміст регістра EAX дорівнюватиме нулю та вміст регістра ECX буде більший від 6;
- [403050] = 1234 – зупинка, якщо вміст елемента пам'яті дорівнює 1234.

Умовні точки зупину із записом у журнал

Умовні точки зупину із записом у журнал (conditional logging breakpoint) є розширенням умовних точок зупину , яке встановлюється натисненням комбінації клавіш <Shift>+<F4>. Для розгляду вмісту журналу необхідно скористатися комбінацією клавіш <Alt>+<L> або вибрати з меню пункт **View | Log.**

Точка зупину на повідомлення Windows

Для встановлення зупину на повідомлення Windows необхідно натиснути комбінацію клавіш <Ctrl>+<F8>. Для того щоб вийти на функцію вікна, необхідно викликати список створених застосувань вікон за допомогою пункту меню **View | Windows.** При роботі з віконними функціями слід ефективніше використовувати точку зупину на повідомлення. Для цього клацнемо по вікну, яке з'явилося, і виберемо з контекстного меню пункт Message breakpoint on Classproc. У вікні, що з'явилося, можна встановити параметри точки зупину , а саме:

– вибрати повідомлення з випадного списку;

– визначити перелік відстежуваних вікон щодо надходження цього повідомлення;

– визначити лічильник спрацьовувань точки зупину ;

– встановити чи зупинити виконання програми;

– встановити чи провести запис у журнал.

Точка зупину на функції імпорту

Список імен, що імпортуються з налагоджуваним модулем, можна отримати за допомогою натиснення комбінації клавіш <Ctrl>+<N>. Далі, клацнувши правою кнопкою миші по вікну, можна встановити:

– точку зупину на виклик функції, що імпортується (команда **Toggle breakpoint on import);**

– умовну точку зупину на виклик функції, що імпортується (команда **Conditional breakpoint on import);**

– умовну точку зупину на імпорт із записом у журнал (команда **Conditional log breakpoint on import);**

– точки зупину на всі посилання на це ім'я (команда **Set breakpoint on every reference);**

– точки зупину із записом у журнал на всі посилання на це ім'я (команда **Set log breakpoint on every reference)** або видалити всі точки зупину (команда **Remove all breakpoints).**

Точка зупину на ділянку пам'яті

Налагоджувач Ollydbg дозволяє встановити одну точку зупину на ділянку пам'яті. Вибираємо вікно дизасемблера або вікно даних (dump). Далі використовуємо контекстне меню і вибираємо пункт **Breakpoint | Memory on access** (на доступ до пам'яті) або **Breakpoint | Memory on write** (на запис у пам'ять). Після цього точка зупину готова до використання. Видалити точку зупину на ділянку пам'яті можна з контекстного меню **Breakpoint | Remove memory breakpoint.**

Точка зупину у вікні Memory

Для цієї точки зупину використовується контекстне меню, що з'являється за допомогою клацання правою кнопкою миші і вибору пункту **Set memory breakpoint on access** (встановити точку зупину на доступ до пам'яті) або **Set memory breakpoint on write** (встановити точку зупину на запис у пам'ять). Видалити точку зупину можна з того контекстного меню командою **Remove memory breakpoint.**

Апаратні точки зупину

Зазвичай для точки зупину використовують стандартний вектор переривання int3. У мікропроцесора Intel Pentium є чотири налагоджувальні регістри: DR0 – DR3. Ці регістри можуть містити чотири контрольні точки. Як тільки адреса, яку використовує команда, дорівнює адресі в одному з вказаних регістрів, то тоді генерується вимкнення, що перехоплюється налагоджувачем. Встановити апаратну точку зупину можна з вікна дизасемблера за допомогою пункту **Breakpoint | Hardware on execution** контекстного меню або у вікні даних за допомогою пунктів **Breakpoint | Hardware on access** або **Breakpoint | Hardware on access.** Видалити апаратні точки зупину можна за допомогою контекстного меню **Breakpoint | Remove hardware breakpoints.**

Пошук інформації

Налагоджувач Ollydbg дозволяє ефективно шукати різну інформацію. За командою від натиснення комбінації клавіш <Ctrl>+<B> з'являється вікно пошуку, де можливо визначити рядок, який розшукуватиметься. Рядок для пошуку можна вводити у вигляді послідовності символів, байтів, символів у кодуванні Unicode.

Для пошуку команд використовуються комбінації клавіш <Ctrl>+<F> – для одиночної команди і <Ctrl>+<S> – для послідовності команд.

Натиснення комбінації клавіш <Ctrl>+<L> повторює останній зроблений пошук.

Завдання для практичного виконання

1. Задано числа Д, М, Р в різних форматах (db, dw, dd), де Д – день народження, М – місяць народження, Р – рік народження. Написати програму обчислення одного з виразів:

1. Д + М ⊕ Р;
2. Д ∨ М × Р;
3. Г/М + Р;
4. Д ⊕ М ∧ Р;
5. Д ∨ М ∧ Р;
6. Д × М ⊕ Р;
7. Д ∧ М + Р;
8. Д ∨ М × Р;
9. (Г – Д) ⊕ Р;
10. Д ⊕ М + Р.

2. Написати програму додавання трьох чисел з урахуванням можливого переповнення розрядної сітки з максимально можливими числами у 16-ковій системі зчислення, два з яких задані в форматах згідно з отриманим варіантом, а третє число вибрати в іншому форматі. Розміри двох чисел задані в таблиці, в якій цифри – це порядковий номер студента у групі. Результат виконання операції зберегти в пам'яті.

X\Y\Z	*db\dt*	*dw\dq*	*dd\dq*	*dq\dw*	*dt\dw*
db	1	6	11	16	21
dw	2	7	12	17	22
dd	3	8	13	18	23
dq	4	9	14	19	24
dt	5	10	15	20	25

3. Написати програму додавання трьох чисел з урахуванням можливого переповнення розрядної сітки, два з яких задані в форматах згідно з отриманим варіантом, а третє число вибрати в іншому форматі. Розміри двох чисел задані в таблиці, в якій цифри – це остання цифра порядкового номера студента у групі. Результат виконання операції зберегти в пам'яті.

X\Y	*dt*	*dq*	*dd*	*dw*
dt	1	2	3	4
dq	5	6	7	8
dd	9	0		

4. Написати програму додавання по байтах багатобайтового слова.

5. Написати програму перемноження (ділення) двох чисел згідно з отриманим варіантом та порядковим номером студента у групі. Розміри чисел задані в таблиці. Результат виконання операції зберегти в пам'яті.

X\Y	*db*	*dw*	*dd*	*dq*
Байт	1	5	9	13
Слово	2	6	10	14
Подвійне слово	3	7	11	15
Слово, збільшене учетверо	4	8	12	16

Контрольні запитання

1. Які групи команд належать до арифметичних операцій?
2. Які арифметичні команди формують ознаки виконання результату операції?
3. В яких випадках використовують команди перетворення типів?
4. Які арифметичні команди використовують позику та як вона використовується?
5. Які арифметичні команди мають трьохоперандний формат та як він використовується?
6. Яка мінімальна кількість команд необхідна для виконання операції ділення?

6. ЛОГІЧНІ КОМАНДИ ТА КОМАНДИ ЗСУВУ

Групи логічних операцій включають команди:
- булевих операцій;
- перевірки і модифікації бітів;
- сканування бітів;
- зсуву та циклічного зсуву;
- установка значення байта за умовою.

6.1. Команди булевих операцій

Логічні операції виконуються командами AND, OR, XOR та NOT. Команди AND, OR та XOR очищають прапорці OF та CF, залишаючи прапорець AF невизначеним і оновлюючи прапорці SF, ZF та PF.

AND приймач, джерело	Логічне *I* (множення)

Команда **AND** здійснює розрядами логічне множення операндів і розміщує результат за адресою, що задається першим операндом.

AND r/m8, imm8	AND r8, r/m8	AND m8, r8
AND r/m16, imm(8,16)	AND r16, r/m16	AND m16, r16
AND r/m32, imm(8,32)	AND r32, r/m32	AND m32, r32
AND r/m64, imm(8,32)	AND r64, r/m64	AND m64, r64

AND прм, прд ; прм := (прм) ∧ (прд)

```
and  al,0000111b ; заповнення нулями старших чотирьох бітів в al
```

Приклад 6.1. Перевірити 3-й біт слова. Якщо він нульовий – записати в комірку пам'яті нуль, а якщо він ненульовий – одиницю.

Лістинг 6.1. Програма виконання прикладу 6.1:

```
        .686                ; директива определения типа микропроцессора
        .model flat         ; задание линейной модели памяти
        .data               ; директива определения данных
_a db 21            ; запись в 8-разрядную ячейку памяти с именем _a числа  21
          res db 0ffh       ; резервирование ячейки памяти под сохранение результата
   .code                    ; директива начала сегмента команд
   _start:                  ; метка начала программы с именем _start
        mov al, _a ;        ; занесение в регистр
      and al, 00001000b     ; выделение 3-го бита (логическое умножение)
        jz m1                   ; перейти, если в результате нули
        mov res, 1              ; в другом случае записать 1
        jmp m2                  ; перейти в заключение
m1:  mov res, 0             ; записать 0 в ячейку результата
m2:   ret               ; возврат управления  ОС
         end _start     ; завершение программы с именем _start
```

У програмі, для того щоб виділити комірку під результат, у початковому стані записуються всі одиниці директивою

```
res db 0ffh.
```

Стан програми з лістингу 6.1 у вікнах налагоджувача OllyDbg після її виконання наведено на рис. 6.1. Щоб у пам'яті не змінити значення змінної _a, вона переноситься до регістра eax, а вже там виконується логічне перемноження. В такому випадку краще використати команду TEST. Через те, що логічне перемноження виконується порозрядно, то всі розряди, крім 3-го (рахуючи зліва та починаючи з нуля), «затираються».

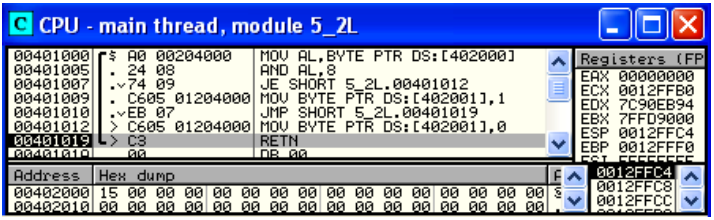


Рис. 6.1. Стан програми з лістингу 6.1 після її виконання

OR приймач, джерело	Логічне *АБО* (додавання)

Команда **OR** виконує розрядами логічне додавання операндів і розміщує результат за першою адресою.

OR прм, прд ; прм := (прм) ∨ (прд)

XOR приймач, джерело	Логічне виключне *АБО*

Команда **XOR** виконує розрядами логічне додавання за модулем два двох операндів і розміщує результат за першою адресою.

XOR прм, прд ; прм := (прм) ⊕ (прд)

```
xor ax, ax ; заповнення нулями ax
```

NOT приймач	Інверсія

Команда **NOT** здійснює операцію інвертування вмісту, що адресується полем операнда, і повертає результат на те саме місце. Біти умов не змінюються.

NOT reg/mem ; інверсія

6.2. Команди перевірки і модифікації бітів

Ця група команд працює з одним бітом, який може знаходитися в пам'яті або в регістрі загального призначення. Положення біта задається у вигляді зсуву щодо молодшого біта операнда. Значення зсуву може задаватися у вигляді безпосереднього значення або міститися в регістрі загального призначення.

Описувані команди спочатку привласнюють значення вибраного біта прапорцю CF. Потім вибраному біту привласнюється нове значення, яке визначається кожною конкретною операцією. Стан прапорців OF, SF, ZF, AF та PF не визначено.

BT база, зміщення	Перевірка біта та встановлення CF	80386

BT r/m16, r16	BT r/m16, imm8
BT r/m32, r32	BT r/m32, imm8
BT r/m64, r32	BT r/m64, imm8

Команда **BT** перевіряє вказаний у команді біт та встановлює прапорець CF у те саме значення.

Після виконання команди BT прапорець CF дорівнює значенню обчисленого біта, прапорці OF, SF, ZF, AF і PF не визначені.

Наприклад, у фрагменті коду

```
bt eax,3 ; перевірка 3-го біта eax та встановлення CF в те саме значення
  jc a2              ; якщо CF = 1, то перейти на мітку a2
```

перевіряється 3-й біт регістра eax та встановлюється прапорець CF, за значенням якого у наступній команді виконується перехід на мітку a2.

Розглянемо роботу з елементами масиву. Масив є впорядкований набір однотипних елементів, об'єднаних під одним ім'ям. Кожен елемент масиву позначається ім'ям, елементи масиву розділені комами.

Приклад 6.2. Задано масив A з $N = 4$ елементів. Навести програму визначення суми елементів масиву A, для яких біти 0 та 1 в кожному числі однакові.

Лістинг 6.2

```
        .686                ; директива определения типа микропроцессора
        .model flat         ; задание линейной модели памяти
.data                       ; директива определения данных
        mas dw 10,16,13,14          ; масив чисел розміром у слово кожен
        sum dw 0                            ; ячейка для хранения результата
        n dw 3          ; ячейка для хранения количества чисел в массиве mas
.code                           ; директива начала сегмента команд
 _start:                        ; метка начала программы с именем _start
        mov ecx, n              ; счетчик количества чисел массива
        clc                     ; сбросить флажок переноса CF
        xor ebx,ebx     ; подготовка регистра ebx
        lea esi, mas    ; занесениеос адреса масиву mas
m1:  mov ax,[esi]       ; занесение элемента массива mas
        bt ax, 0                ; выделение  нулевого бита
        setc bh                 ; если cf = 1, то bh := 1
        bt ax, 1                ; выделение  первого бита
        setc bl                 ; если cf = 1, то bl := 1
        cmp  bh, bl             ; сравнение битов
        jne m2                  ; если несравнимые, то перейти на m2
        add sum,ax              ; сложение выбранных элементов
m2:  add esi,2                  ; подготовление адреса нового элемента
        loop m1                 ; перейти на m1, если ecx ≠ 0
         ret            ; возврат управления  ОС
        end _start      ; окончание программы с именем _start
```

Спочатку в програмі завантажується лічильник ecx. Потім прочитується елемент масиву командою mov ax,[esi] та виділяються необхідні біти (команда bt ax, …). Для тих елементів масиву, для яких біти збіглися, виконується додавання (add sum,ax). Через те, що числа мають формат у два байти, командою add esi,2 виконується підготування до вибірки наступного елемента. Цикл вибірки закінчується, якщо у лічильнику ecx опиниться нульове значення.

Результат виконання програми з лістингу 6.2 наведено на рис. 6.2.

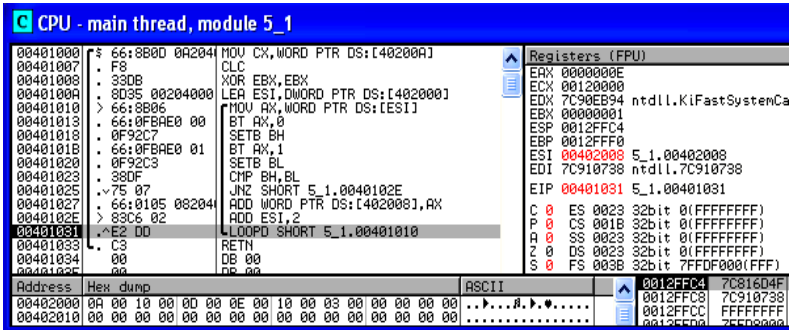


Рис. 6.2. Результат виконання програми з лістингу 6.2

BTS приймач, джерело	Перевірка та встановлення біта
BTR приймач, джерело	Перевірка та скидання біта
BTC приймач, джерело	Перевірка та інвертування біта

Ці три команди відповідно спочатку переносять у прапорець cf значення біта, номер якого вказано в джерелі з приймача, а потім встановлюють в 1 (BTS), скидають в 0 (BTR) і інвертують (BTC) значення біта приймача, який вказано в джерелі.

Після виконання команд BTS, BTR і BTC прапорець CF дорівнює значенню обчисленого біта до його зміни в результаті дії команди; прапорці OF, SF, ZF, AF і PF не визначено. Наприклад:

```
bts ecx,10 ; перенесення значення 10-го біта ecx в CF та встановлення ecx[10] в 1
```

6.3. Команди сканування бітів

Ці команди сканують слово або подвійне слово у пошуках встановленого біта і заносять у регістр номер першого встановленого біта (ціле число, що визначає позицію знайденого біта). Сканований рядок може знаходитися як в регістрі, так і в пам’яті. Якщо все слово дорівнює нулю, то встановлюється прапорець ZF. Якщо одиничний біт знайдений, то прапорець ZF очищається. Якщо одиничних бітів не знайдено, значення регістра призначення не визначено. Стан прапорців OF, SF, ZF, PF та CF не визначено.

BSF приймач, джерело	Прямий пошук біта
BSR приймач, джерело	Зворотний пошук біта

```
BSF    r16, r/m16
BSF    r32, r/m32
BSF    r64, r/m64
```

Команда **BSF** сканує джерело (регістр або змінну), починаючи з **наймолодшого** біта, і записує в приймач (регістр) номер першого біта, що зустрівся, який дорівнює 1.

Команда **BSR** сканує джерело, починаючи з **найстаршого** біта, і повертає номер першого ненульового біта, що зустрівся, рахуючи від нуля. Тобто, якщо джерело дорівнює 0000 0000 0000 0010b, то BSF поверне 1, а BSR – 14.

6.4. Команди зсуву (зміщення) і циклічного зсуву

Команди зсуву і циклічного зсуву переставляють біти усередині операнда. Ці команди підрозділяються на три класи:

– команди зсуву;
– команди циклічного зсуву;
– команди подвійного зсуву.

6.4.1. Команди зсуву

Команди зсуву виконують арифметичний або логічний зсув. Арифметичний зсув праворуч копіює знаковий біт у порожню позицію старшого біта операнда, тоді як логічний зсув праворуч, зрушивши операнд праворуч, очищає порожні позиції.

Арифметичний зсув є найшвидшим способом виконання простих обчислень. Наприклад, арифметичний зсув праворуч на один біт виконує ділення цілого на два. Логічний зсув ділить ціле без знака або додатне ціле, але від'ємне ціле зі знаком втрачає свій знаковий біт. Так, число – 10b = 2_{10} після зсуву на 1 біт ліворуч перетворюється в 0100b = 4_{10}.

SAL приймач, лічильник	Арифметичний зсув ліворуч
SAR приймач, лічильник	Арифметичний зсув праворуч
SHL приймач, лічильник	Логічний (беззнаковий) зсув ліворуч
SHR приймач, лічильник	Логічний (беззнаковий) зсув праворуч

Команда SAR може бути використана для ділення цілих на ціле, та таке, що є степенем двійки, результат ділення не буде тим самим, що й при виконанні команди IDIV. Різниця виявляється тільки для від'ємних чисел. Частка при виконанні команди IDIV округлюється у бік нуля, тоді як «частка» при виконанні команди SAR округлюється у бік від'ємного

нескінченності. Наприклад, коли використовується команда IDIV для ділення –9 на 4, результатом буде –2 із остачею –1. Якщо використовувати команду SAR для зсуву –9 праворуч на два біти, результатом буде –3. «Остачею» такого виду ділення буде –13; проте команда SAR зберігає тільки біт старшого розряду остачі (у прапорці CF).

SAR r/m8, imm8	SAR r/m8, CL
SAR r/m16, imm8	SAR r/m16, CL
SAR r/m32, imm8	SAR r/m32, CL
SAR r/m64, imm8	SAR r/m64, CL

Зсуви на 1 змінюють значення прапорця OF: команди SAL/SHL встановлюють його в 1, якщо після зсуву старший біт змінився (тобто старші два біти початкового числа не були однаковими), і в 0, якщо старший біт залишився таким самим. Команда SAR встановлює OF в 0, а SHR – в значення старшого біта початкового числа. Для зсувів на декілька бітів значення OF не визначене. Прапорці SF, ZF, PF встановлюються всіма зсувами відповідно до результату, прапорець AF не визначений (крім випадку, коли лічильник зсувів дорівнює нулю: нічого не відбувається і прапорці не змінюються).

SHL (SHR, SAL, SAR) mem/reg, const

```
SHL  DL, 1    ; логічний зсув ліворуч Pr DL на 1
SAL  AX, CL ; арифметичний зсув праворуч на число, що знаходиться в Pr CL
```

Розглянемо програму множення вмісту регістра EAX на 10 з використанням команд зсуву, пересилання та складання (лістинг 6.3).

Лістинг 6.3. Програма множення на 10 без використання команди множення:

```
    .686          ; директива определения типа микропроцессора
    .model flat     ; задание линейной модели памяти
    option casemap: none    ; отличие строчных и прописных букв
.data                                ; директива определения данных
    dOp1 DD 3           ; запись в 32-разрядную ячейку памяти с именем dOp1 числа
    res DD 0              ; резервирование ячейки памяти под хранение результата
.code                                 ; директива начала сегмента команд
  _start:              ; метка начала программы с именем _start
      mov eax,dOp1           ; занесение числа dOp1
      mov ebx, eax           ; сохранение в дополнительном регистре
      shl eax,3                 ; сдвиг влево на 3 разряда – умножение на 8
      shl ebx,1                 ; сдвиг влево на 1 разряд – умножение на 2
      add eax,ebx               ; получение результата
      mov res, eax       ; сохранение результата в памяти
      ret                       ; возврат управления  ОС
      end _start           ; завершение программы с именем _start
```

З використанням команд зсуву можна навести ще декілька подібних варіантів множення на 10, фрагменти яких можуть бути такими:

```
mov ebx, eax                    mov ebx,eax
```

```
shl ebx, 2                      shl eax,3
add ebx, eax                    mov cx,2
shl ebx,1                  m1:  add eax,ebx
mov res, ebx                    loop m1
                                mov res, eax
```

Таким чином, виконання операцій множення та ділення на степінь двійки найкраще виконувати за допомогою операцій зсуву на відповідну кількість розрядів.

6.4.2. Команди циклічного зсуву

ROL приймач, лічильник	Арифметичний (циклічний) зсув ліворуч
ROR приймач, лічильник	Арифметичний (циклічний) зсув праворуч
RCL приймач, лічильник	Логічний (циклічний) зсув ліворуч
RCR приймач, лічильник	Логічний (циклічний) зсув праворуч

Циклічний зсув є операцією зсуву, при якій біт, що висовується, займає місце, яке стало вільним. Чотири команди циклічного зсуву з використанням біта CF виконуються над вмістом пам'яті чи регістрів: **ROL, ROR, RCL, RCR**. Можливі циклічні зсуви на один біт і змінну кількість бітів, зазначену в регістрі CL. Біт CF стає таким, що дорівнює останньому біту, який зсувається; біт OF визначається тільки для зсувів на один розряд і встановлюється в тому випадку, якщо біт знака результату операції відрізняється від первінного значення біта знака.

Існують чотири логічні команди над двома операндами. При всіх цих операціях біти CF і OF скидаються, а біти SF, PF і ZF відбивають результат виконання операції.

ROL (ROR, RCL, RCR) mem/reg, count

RCL (циклічний зсув уліво через перенесення) циклічно зсуває дані на один біт або на кількість бітів, заданих в операнді-лічильнику (безпосереднє значення або значення, що міститься в регістрі CL). Ця команда відрізняється від ROL тим, що вона інтерпретує прапорець CF як однобітовє розширення операнда призначення з боку старших розрядів. Для кожного розряду біт, який витісняється з лівого кінця операнда, переміщається в прапорець CF. У той самий час біт, що міститься в прапорці CF, вводиться з правого боку.

RCR (циклічний зсув управо через перенесення) циклічно зміщає дані праворуч на один біт або на кількість бітів, заданих в операнді-лічильнику (безпосереднє значення або значення, що міститься в регістрі CL). Ця команда відрізняється від ROR тим, що вона інтерпретує прапорець CF як однобітовє розширення операнда призначення з боку молодших розрядів. Для кожного розряду біт, який витісняється з правого кінця операнда, переміщається в прапорець CF. У той самий час біт, що міститься в прапорці CF, вводиться з лівого боку.

6.4.3. Команди подвійного зсуву

SHLD r/m16, r16, imm8	SHLD r/m16, r16, CL
SHLD r/m32, r32, imm8	SHLD r/m32, r32, CL
SHLD r/m64, r64, imm8	SHLD r/m64, r64, CL

Команди SHLD/SHRD (ліворуч/праворуч) – мають формат із трьома операндами. Третій операнд визначає кількість розрядів зсуву. Розряди першого операнда зміщуються ліворуч/праворуч, розряди, що при цьому звільняються, заповнюються значеннями старших/молодших розрядів другого операнда. Сам другий операнд не зсувується:

```
mov   cl,16
shld  ebx,eax,cl ; сдвиг без разрушения eax
```

6.5. Команди установки байта за умовою

SETcc (встановити значення байта залежно від умови cc) завантажує значення 1 в байт, якщо умова cc істинна, інакше очищає байт.

SETcc приймач	Установка байта за умовою

Це набір команд, що встановлюють приймач (8-бітовий регістр або змінна розміром в 1 байт) в 1 або 0, якщо задовольняється або не задовольняється певна умова. Фактично у кожному випадку перевіряється стан тих або інших прапорців, але, коли команда з набору SETcc використовується відразу після команди CMP, умови набувають формулювань, відповідних до відношень між операндами CMP (табл. 6.1).

Тобто якщо операнди команди CMP були нерівні, то команда SETNE, що була виконана відразу після команди CMP, встановить значення свого операнда в 1.

Слова «вище» і «нижче» в табл. 6.1 належать до порівняння чисел без знака, слова «більше» і «менше» враховують знак.

Таблиця 6.1

Код команди	Реальна умова	Умова для CMP
SETA SETNBE	CF = 0 та ZF = 0	Якщо вище Якщо не нижче і не дорівнює
SETAE SETNB SETNC	CF = 0	Якщо вище або дорівнює Якщо не нижче Якщо немає перенесення
SETB SETNAE SETC	CF = 1	Якщо нижче Якщо не вище і не дорівнює Якщо перенесення
SETBE SETNA	CF = 1 або ZF = 1	Якщо нижче або дорівнює Якщо не вище
SETE	ZF = 1	Якщо дорівнює

SETZ		Якщо нуль
SETG SETNLE	ZF = 0 та SF = OF	Якщо більше Якщо не менше і не дорівнює
SETGE SETNL	SF = OF	Якщо більше або дорівнює Якщо не менше
SETL SETNGE	SF < > OF	Якщо менше Якщо не більше та не дорівнює
SETLE SETNG	ZF = 1 або SF < > OF	Якщо менше або дорівнює Якщо не більше
SETNE SETNZ	ZF = 0	Якщо не дорівнює Якщо не нуль
SETNO	OF = 0	Якщо немає переповнення
SETO	OF = 1	Якщо є переповнення
SETNP SETPO	PF = 0	Якщо немає парності Якщо непарне
SETP SETPE	PF = 1	Якщо є парність Якщо парне
SETNS	SF = 0	Якщо немає знака
SETS	SF = 1	Якщо є знак

Наприклад:

```
bt ax, 0          ; виділення  нульового біта
setc bh           ; якщо cf = 1, то bh := 1
```

6.6. Команда перевірки

TEST приймач, джерело	Логічне порівняння

Команда **TEST** здійснює те саме, що і команда AND (логічне множення). Вона встановлює біти умов, але не формує результат.

TEST прм, прд ; (прм) ∧ (прд)

Команда обчислює результат дії побітового логічного ***I*** над приймачем (регістром або змінною) і джерелом (числом, регістром або змінною; джерело і приймач не можуть бути змінними одночасно) і встановлює прапорці SF, ZF і PF відповідно до одержаного показника, не зберігаючи результату (прапорці OF і CF заповнюються нулями, значення прапорця AF не визначене). Команда TEST, так само як і команда CMP, використовується здебільшого в поєднанні з командами умовного переходу (Jcc), умовного пересилання даних (CMOVcc) і умовного установлення байтів (SETcc).

Завдання для практичного виконання

1. Задано масив *A* з *N = 5* елементів. Сформувати новий масив елементів, для яких біти 0, 1 та 2 в кожному числі однакові.

2. Задано масив A з $N = 6$ елементів. Сформувати новий масив елементів, для яких біти 0 та 2 в кожному числі мають одиниці.

3. Задано масив A з $N = 7$ елементів. Сформувати новий масив елементів, для яких біти 1, 2 та 3 в кожному числі мають нулі.

4. Задано масиви A і B по $N = 4$ елементи. Додати однойменні елементи масивів без урахування переповнення розрядної сітки та сформувати новий масив C елементів, для яких біти 7 та 8 однакові.

5. Задано масиви A і B по $N = 6$ елементів. Перемножити однойменні елементи масивів без урахування переповнення розрядної сітки та сформувати новий масив C елементів, для яких біти 6, 7 та 8 мають одиниці.

6. Задано масиви A і B по $N = 6$ елементів. Поділити однойменні елементи масивів та сформувати новий масив C елементів із цілих часток результату, для яких біти 5 та 8 мають нулі.

7. Задано масиви A і B по $N = 6$ елементів. Виконати команду TEST над однойменними елементами масивів та сформувати новий масив C елементів, для яких біти 0 та 2 результату мають одиниці.

8. Задано масиви A і B по $N = 8$ елементів. Виконати команду AND над однойменними елементами масивів та сформувати новий масив C елементів, для яких біти 0 та 2, та 4 результату мають нулі.

9. Задано масиви A і B по $N = 8$ елементів. Виконати команду OR над однойменними елементами масивів та сформувати новий масив C елементів, для яких результат дорівнює 9.

10. Задано масиви A і B по $N = 7$ елементів. Додати за модулем два однойменні елементи масивів та сформувати новий масив C елементів, для яких результат дорівнює 5.

Контрольні запитання

1. Наведіть приклади використання логічних команд та команд зсуву.

2. Поясніть різницю між арифметичним зсувом та логічним.

3. Наведіть приклади використання подвійного зсуву та їх практичне значення.

7. КОМАНДИ ПЕРЕДАННЯ КЕРУВАННЯ

Більшість програм не є лінійними і потребують, залежно від установлених умов, організації різних підходів. Більшість команд асемблера встановлюють біти прапорців, завдяки яким з'являється можливість організації обчислювальних алгоритмів і різних логічних структур. Крім того, існує можливість організації безумовних переходів.

7.1. Команда безумовного переходу JMP

JMP операнд	Безумовний перехід

Команда JMP передає керування в іншу точку програми, не зберігаючи інформацію для повернення. Операндом може бути безпосередня адреса для переходу (в програмах використовують ім'я мітки, встановленої перед командою, на яку виконується перехід), а також регістр або змінна, що містить адресу.

Залежно від типу переходу розрізняють:

- перехід типу short (короткий перехід) – якщо адреса переходу знаходиться в межах – 128...+ 127 байт від команди JMP;
- перехід типу near (близький перехід) – якщо адреса переходу знаходиться в тому самому сегменті пам'яті, що і команда JMP;
- перехід типу far (далекий перехід) – якщо адреса переходу знаходиться в іншому сегменті. Далекий перехід може виконуватися і в той самий сегмент за умови, що в сегментній частині операнда вказано число, що збігається з поточним значенням CS;
- перехід із перемиканням завдання – передача керування іншому завданню в багатозадачному середовищі.

При виконанні переходів типу short та near команда JMP фактично перетворює значення регістра RIP, змінюючи тим самим зсув наступної виконуваної команди щодо початку сегмента коду. Якщо операнд – регістр або змінна в пам'яті, то його показник просто копіюється в регістр RIP, наче це була команда MOV. Якщо операнд для команди JMP – безпосередньо вказане число, то його значення додається до вмісту регістра RIP, приводячи його до відносного переходу. В асемблерних програмах як операнд зазвичай указують імена міток, але на рівні коду, що виконується, асемблер обчислює і записує саме відносні зсуви.

Виконуючи далекий перехід у реальному, віртуальному і захищеному режимах (при переході в сегмент з тими самими привілеями), команда JMP завантажує нове значення в регістр RIP і новий селектор сегмента коду – в сегментний регістр CS, використовуючи старші 16 бітів операнда як нове значення для CS і молодші 16 або 32 біти як значення IP або EIP.

Команду jmp можна використовувати не тільки для безумовного переходу в сегменті програмного коду, але і для організації розгалужень. Для цього можна застосувати один із її форматів:

```
JMP    rel8;    JMP   r/m16;
JMP    rel16;   JMP    r/m32;
JMP    rel32;   JMP    r/m64
```

Наприклад:

```
lea EDI, L1              ; завантажити в EDI адресу змінної L1
jmp EDI                  ; безумовний перехід за вмістом EDI
```

У цьому прикладі в регістр EDI поміщається зсув мітки L1, після чого за допомогою команди jmp EDI керування передається на цю мітку.

```
    .data                        ; директива определения данных
mas1 dd  L1 ; резервування в пам'яті 4-х байтів для збереження змінної L1
    .code                        ; директива начала сегмента команд
    ...
    lea esi, mas1        ; завантаження в esi адреси змінної mas1
    jmp dword ptr [esi]      ; перейти за адресою, яка знаходиться в esi
```

У даному прикладі в регістр ESI поміщається адреса змінної mas1, тоді як сама змінна mas1 містить адресу мітки L1.

Команда jmp dword ptr [esi] в цьому випадку передає керування на мітку L1.

7.2. Команди умовної передання керування Jcc

Jcc мітка	Умовний перехід

Це набір команд, що виконують перехід у програмі (типу short або near), якщо задовольняється відповідна умова, якою в кожному випадку реально є стан тих або інших прапорців. Але, коли команда з набору Jcc використовується відразу після команди CMP, умови набувають формулювань, відповідних взаємодіям між операндами команди CMP (табл. 7.1). Слова «вище» і «нижче» в табл. 7.1 належать до порівняння чисел без знака; слова «більше» і «менше» враховують знак.

Таблиця 7.1

Код команди	Реальна умова	Умова для команди CMP
JA JBE	CF = 0 та ZF = 0	Якщо вище Якщо нижче і дорівнює
JAE JNB JNC	CF = 0	Якщо вище або дорівнює Якщо не нижче Якщо немає перенесення
JB	CF = 1	Якщо нижче

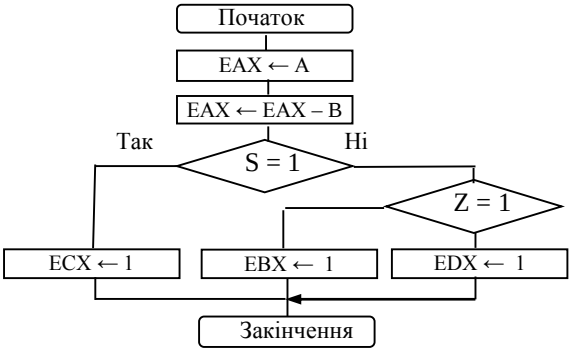
JNAE JC		Якщо не вище і не дорівнює Якщо перенесення
JBE JNA	CF = 1 та ZF = 1	Якщо нижче або дорівнює Якщо не вище
JE JZ	ZF = 1	Якщо дорівнює Якщо нуль
JG JNLE	ZF = 0 та SF = OF	Якщо більше Якщо не менше і не дорівнює
JGE JNL	SF = OF	Якщо більше або дорівнює Якщо не менше
JL JNGE	SF < > OF	Якщо менше Якщо не більше і не дорівнює
JLE JNG	ZF = 1 та SF < > OF	Якщо менше або дорівнює Якщо не більше
JNE JNZ	ZF = 0	Якщо не дорівнює Якщо не нуль
JNO	OF = 0	Якщо немає переповнення
JO	OF=1	Якщо є переповнення
JNP JPO	PF = 0	Якщо немає парності Якщо непарне
JP JPE	PF = 1	Якщо є парність Якщо парне
JNS	SF = 0	Якщо немає знака
JS	SF =1	Якщо є знак

Команди Jcc не підтримують далеких переходів, тому, якщо потрібно виконати умовний перехід на далеку мітку, необхідно використати команду з набору Jcc зі зворотною умовою і далекий JMP, як наприклад:

```
    cmp   ax,0
    jne   local_1
    jmp   far_label          ; перехід, якщо AX = 0.
local_1:
```

Розглянемо декілька програм із використанням команд організації розгалужень.

Приклад 7.1. Написати програму аналізу двох чисел:
якщо A < B, то записати 1 в Рг ECX;
якщо A = B, то записати 1 в Рг EBX;
якщо A > B, то записати 1 в Рг EDX.
Алгоритм виконання програми наведено на рисунку.



Лістинг 7.1. Програма виконання прикладу 7.1:

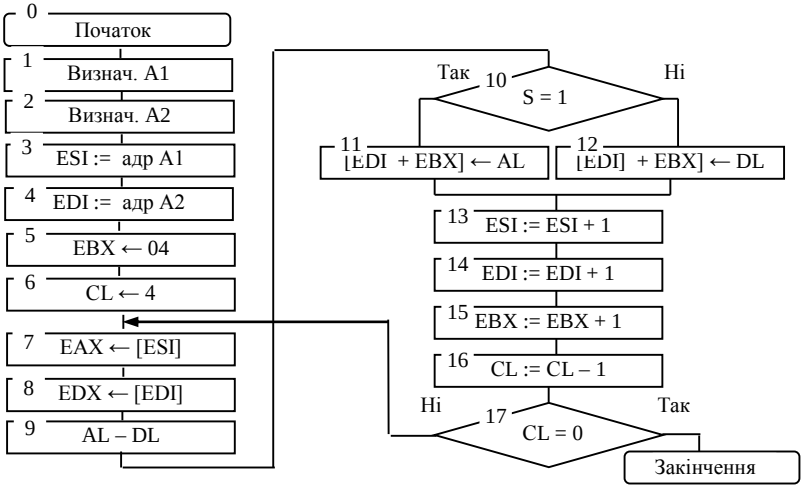
```
    .686                    ; директива определения типа микропроцессора
    .model flat             ; задание линейной модели памяти
.data                       ; директива определения данных
    a  dd  3 ; запись в 32-разрядную ячейку памяти с именем a числа 3
                            ; (первый операнд)
    b   dd  5               ; второй операнд
.code                       ; директива начала сегмента команд
_start:                     ; метка начала программы с именем _start
    mov eax, a          ; занесение первого числа
    sub eax, b          ; сравнение двух чисел для установления признаков
    js  minus       ; перейти, если a < b
    jz  ravno               ; перейти, если a = b
    mov edx, 0001h
    jmp a1                  ; безусловный переход на a1
ravno: mov ebx, 0001h
    jmp a1                  ; безусловный переход на a1
minus: mov ecx, 0001h
a1:    ret                  ; возврат управления  ОС
        end _start          ; директива завершение программы с именем _start
```

У програмі не використовуються комірки пам'яті для запису результату. Тому комірки для збереження результату й не резервуються.

Приклад 7.2. Задано масиви A та B по 4 елементи. Навести алгоритм та програму формування масиву C за таким правилом: якщо $A_i < B_i$, то $C_i = A_i$, інакше $C_i = B_i$.

У прикладі потрібно сформувати новий масив із мінімальних елементів з кожної пари, який для вибраних елементів масивів повинен бути: 0, 1, 1, 3.

Блок-схему виконання прикладу наведено на рисунку.



Лістинг 7.2. Програма виконання прикладу 7.2:

```
        title CopyRight by Rysovaniy A. N.
        .686            ; директива определения типа микропроцессора
       .model flat          ; задание линейной модели памяти
  .data                         ; директива определения данных
a1 db 0,1,2,3          ; запись в массив памяти с именем a1 8-разрядных чисел
a2 db 1,2,1,4          ; запись в массив памяти с именем a2 8-разрядных чисел
   .code           ; директива начала сегмента команд
   _start:         ; метка начала программы с именем _start
        lea esi,a1              ; занесение адреса первого массива чисел
        mov edi,offset a2  ; занесение адреса второго массива чисел
        mov ebx, 4              ; учет смещения адресов между массивами
        mov cl, 04                      ; счетчик количества пар чисел
m3: mov eax,[esi]    ; занесение числа из массива a1
        mov edx,[edi]                   ; занесение числа из массива a2
        cmp al,dl                       ; сравнение чисел (al – bl)
        js m1                           ; перейти на  m1, якщо al < dl
        mov [edi+ebx],dl
        jmp m2                  ; безусловный переход на метку m2
m1:  mov [edi+ebx],al          ; сохранение в новом массиве памяти
m2:  add esi, 01                ; увеличение адреса массива a1
        add edi, 01             ; увеличение адреса массива a2
        add ebx, 01 ;
        dec cl          ; cl := cl – 1 и формирование признаков
        jnz m3          ; переход на m3, если не нуль
        ret             ; возврат управления  ОС
        end _start   ; директива завершение программы с именем _start
```

Через те, що числа масивів у пам’яті розташовані послідовно один за одним, перше число масиву a2 відстає на 4 байти від адреси першого

числа масиву a1, тому, щоб сформувати адресу масиву результату, необхідна команда mov ebx, 4.

Вікна налагоджувача OllyDbg з відповідною програмою наведено на рис. 7.1.

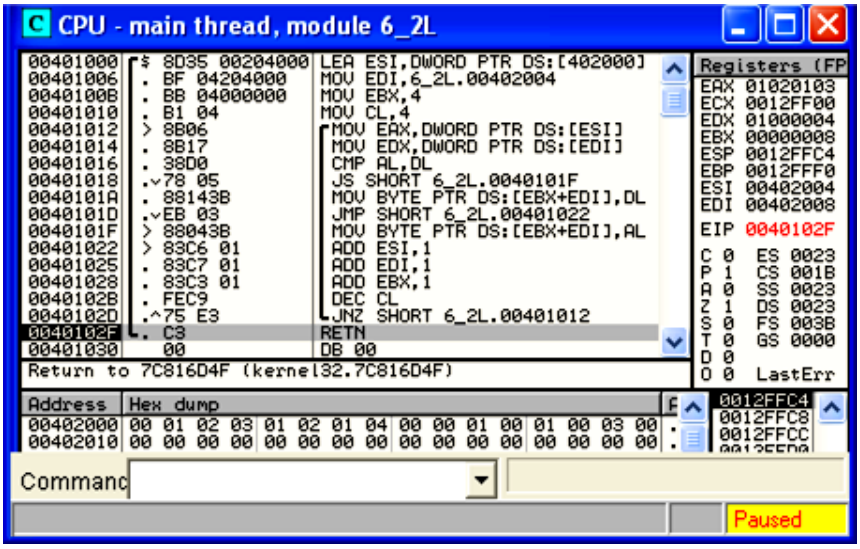


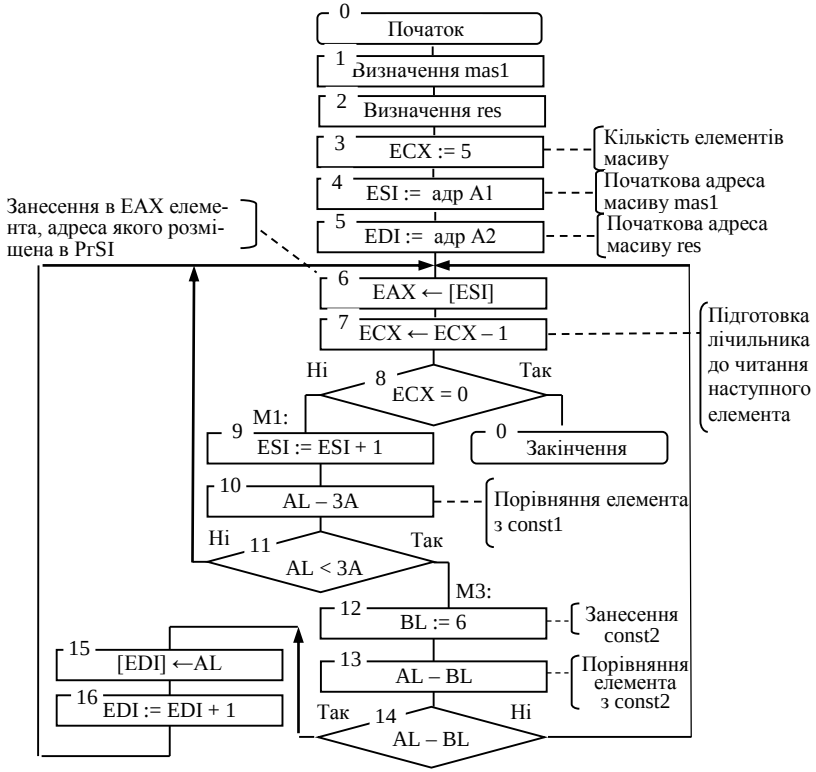
Рис. 7.1. Вікна налагоджувача OllyDbg з програмою з лістингу 7.2

Результат формування чисел нового масиву після їх взаємного порівняння розташовано в дампі пам'яті, починаючи з адреси 00402000h.

Приклад 7.3. Задано масив A з 5 елементів. Навести алгоритм та програму визначення елементів масиву A, які задовольняють умову $6 \le A_i < 3A$, та розмістити їх у новому масиві.

Блок-схему виконання прикладу наведено на рисунку.

Як лічильник елементів початкового масиву з ім'ям mas1 використовується регістр ECX. Масив, куди будуть записуватися числа, що задовольняють задану умову, визначено як масив з ім'ям res. Початкові адреси масивів записуються в регістри ESI та EDI, хоча можна використовувати й комірки пам'яті, але це дещо знижує швидкодію виконання програми.



Лістинг 7.3. Програма виконання прикладу 7.3:

```
      title CopyRight by Rysovaniy A. N.
      .686          ; директива определения типа микропроцессора
      .model flat ; задание линейной модели памяти
.data                                   ; директива определения данных
      mas1 db 0,4,6,3ah,3bh
      res  db 5 dup(?)
.code                       ; директива начала сегмента команд
_start:           ; метка начала программы с именем _start
   mov ecx,5
   lea esi,mas1   ; занесение адреса начала элементов массива mas1
   lea edi,res      ; занесение адреса массива результата
m4: mov eax, [esi]
      dec ecx
      jz m1                     ; переход на метку m1, если нуль
      inc esi                   ; esi := esi + 1
      cmp al,3ah                ; сравнение значений
      jl m3
      jmp m4                    ; безусловный переход на метку m4
      m3: mov bl,06
      cmp al,bl                 ; сравнение значений
      jnl m5                    ; если не меньше
```

```
        jmp m4                  ; безусловный переход на метку m4
m5: mov [edi],al
        inc edi                 ; edi := edi + 1 (увеличение на единицу edi)
        jmp m4                  ; безусловный переход на метку m4
m1: ret                         ; возврат управления  ОС
        end _start      ; директива завершение программы с именем _start
```

Вікна налагоджувача OllyDbg з програмою з лістингу 6.3 наведено на рис. 7.2.

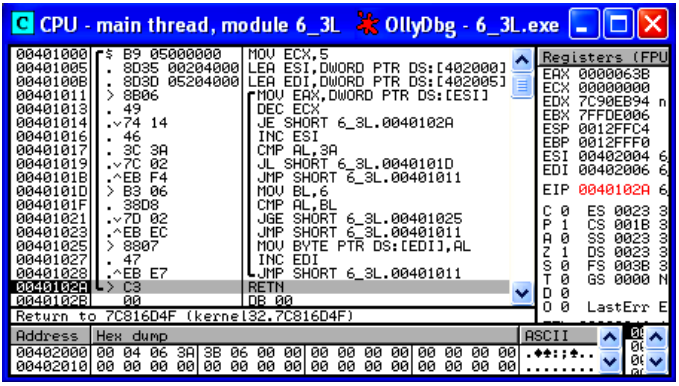
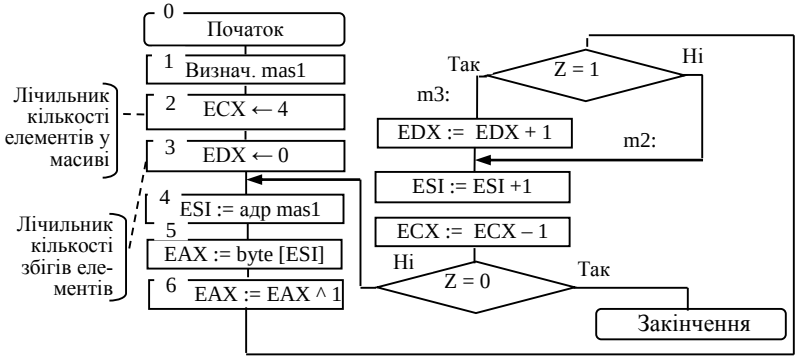


Рис. 7.2. Вікна налагоджувача OllyDbg з програмою з лістингу 7.3

Приклад 7.4. Задано масив A з $N = 4$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, у яких біти 0 мають нулі.

Блок-схему алгоритму виконання програми наведено на рисунку.



Лістинг 7.4. Програма виконання прикладу 7.4:

```
 title CopyRight by Rysovaniy A. N.
 .686                   ; директива определения типа микропроцессора
 .model flat            ; задание линейной модели памяти
.data                   ; директива определения данных
 mas1 db 5,7,8,9  ; резервирование участка памяти  с именем mas1
```

```
                                        ; ячеек размером в байт для чисел этого массива
        len equ $-mas1   ; вычисление количества байтов массива mas1
  .code                  ; директива начала сегмента команд
_start:                          ; метка начала программы
        mov ecx, len   ; счетчик количества байтов в массиве mas1
        mov edx, 0               ; счетчик количества элементов mas1[0] = 0
        lea esi, mas1          ; занесение адреса массива mas1 в esi
begin: movzx eax, byte ptr [esi]          ; занесение элемента  mas1  в  eax
        and eax, 01              ; выделение 1-го бита (логическое умножение)
        jz m3                    ; перейти на m3, если ecx = 0 (z = 1)
        jmp m2                   ; безусловный переход на метку m2
m3:  inc edx                     ; увеличение счетчика элементов mas1[0] = 0
m2:  inc esi             ; подготовка адреса для считывания нового элемента
        dec ecx          ; проверка счетчика количества байтов в массиве mas1
        jnz begin        ; если в не ecx нуль, то перейти  к следующему циклу
        ret                      ; возврат управления  ОС
        end  _start              ; директива завершение программы с именем _start
```

Вікна налагоджувача OllyDbg з відповідною програмою наведено на рис. 7.3. Командою movzx eax, byte ptr [esi] в регістр eax заноситься з масива mas1 один елемент розміром в байт.

В регістрі EDX зберігається кількість елементів масиву mas1, у яких біти 0 мають нулі. Для чисел масиву mas1 тільки число 8 має в нульовому розряді нуль. Тому в регістрі edx в результаті виконання програми буде 1. Замість команди **and eax, 01** частіше використовують команду **TEST eax, 01**, яка не змінює результат.

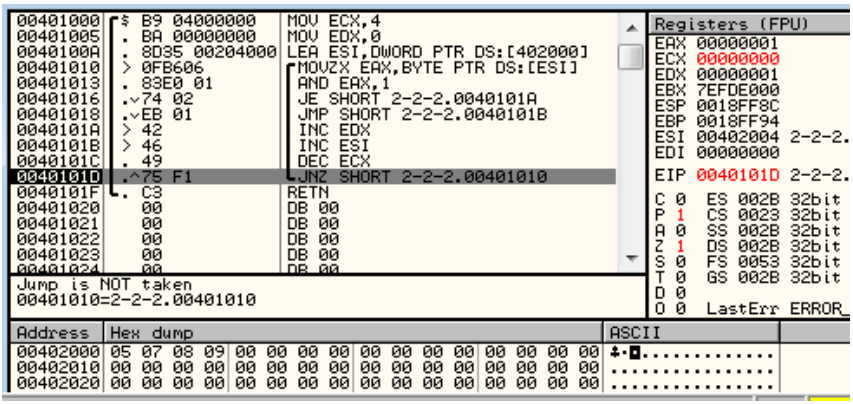


Рис. 7.3. Вікна налагоджувача OllyDbg з програмою з лістингу 7.4

Приклад 7.5. Підрахувати кількість додатних чисел.

Лістинг 7.5. Програма виконання прикладу 7.5:

```
        .686             ; директива определения типа микропроцессора
        .model flat   ; задание линейной модели памяти
        option casemap:none   ; отличие строчных и прописных букв
.data                                   ; директива определения данных
```

```
        mas1 db 1,-1,2,-2,3,-3,4,-4,5,-5 ; для отладки
        len equ $-mas1      ; счетчик чисел
.code
_st:
        xor edx,edx    ;
        lea esi,mas1  ; загрузка адреса массива mas1
        mov ecx,len   ; занесение счетчика количеством чисел массива
m1:   mov al,[esi]   ; занесение числа из памяти, адрес которого в esi
        inc esi      ; подготовление для считывания следующего числа
        add al,0   ; определение признаков числа
        js m2
        inc edx ; увеличение счетчика
m2:  loop m1
        ret
       end _st
```

Приклад 7.6. Збільшення всіх від'ємних чисел масиву з 128 однобайтових чисел на одиницю.

Лістинг 7.6. Програма виконання прикладу 7.6:

```
        .686          ; директива определения типа микропроцессора
        .model flat   ; задание линейной модели памяти
        option casemap:none   ; отличие строчных и прописных букв
        .data                                   ; директива определения данных
        mas128 db 1,-1,2,-2,3,-3,4,-4,5,-5 ; для отладки
        ; mas128 db 128 dup(?)
        len equ $-mas128 ; счетчик чисел
.code
_start:
        lea esi,mas128  ; занесение адреса массива mas128
        mov ecx,len       ; занесение счетчика количеством чисел масива
m1:  mov al,[esi]   ; занесение числа из памяти, адрес которого в esi
        add al,0       ; определение признаков числа
        js m2
        jmp m3
m2:  add byte ptr [esi],01h ; увеличение на единицу отрицательного числа
m3:  inc esi
        loop m1
        ret
        end _start
```

Приклад 7.7. Визначити кількість чисел, які менші від числа (–10000).

Лістинг 7.7. Програма виконання прикладу 7.7:

```
        .686            ; директива определения типа микропроцессора
        .model flat     ; задание линейной модели памяти
        option casemap:none  ; отличие строчных и прописных букв
.data                                   ; директива определения данных
        const1 dw -1;0000 ; проверка выполнена на маленьком числе (-1)
        ;mas dw 64 dup(?)
        mas dw 1,-1,2,-2,3,-3,4,-4,5,-5 ; для отладки
```

```
        len equ ($-mas)/type mas ; счетчик чисел
.code
_start:
        xor dx,dx     ; счетчик чисел < -10000
        lea esi,mas  ; занесение адреса массива mas
        mov ecx,len     ; занесение счетчика количеством чисел массива
m1:  mov ax,[esi]      ; занесение числа из памяти, адрес которого в esi
        add ax,0
        jns m1
;  для від'ємних чисел
        cmp const1,ax
        jnc m2
        jmp m3
; для додатних чисел
m1:  cmp const1,ax
        jc m2
        jmp m3
m2:  inc dx
m3:  add esi,2
        loop m1
        ret                          ; возврат управления  ОС
        end _start                ; окончание программы с именем _start
```

Від'ємні числа в пам'яті зберігаються в додатковому коді і, отже, їх старший розряд завжди дорівнює 1. За рахунок чого модуль від'ємного числа завжди більший від додатного. При порівнянні (відніманні) від'ємного числа з додатним ніколи не буде ознаки CF. Ознаки CF не буде і при порівнянні найменшого від'ємного (але більшого за модулем) з іншим від'ємним числом.

Для налагодження програми використовується невеликий масив, який потім може бути змінено, наприклад, на рядок mas1 db 256 dup(?).

JCXZ мітка	Перехід, якщо CX = 0
JECXZ мітка	Перехід, якщо ECX = 0

Команда JCXZ виконує близький перехід на вказану мітку, якщо регістр CX/ECXR/CX (для JCXZ і JECXZ відповідно) дорівнює нулю. Так само, як і команди із серії Jcc, команди JCXZ і JECXZ не можуть виконувати далеких переходів. Перевірка рівності RCX = 0, наприклад, може бути потрібна на початку циклу, організованого командою LOOPNE. Якщо у цей цикл увійти з CX = 0, то він буде виконаний 65 535 (2^{16})разів.

7.3. Команди керування циклами LOOPx

LOOP мітка	Умовний перехід

Команда **LOOP** зменшує регістр RCX на 1 і виконує перехід типу short на мітку, яка ***не може бути далі*** за відстань – 128...+ 127 байтів від

команди LOOP, якщо RCX не дорівнює нулю. Ця команда використовується для організації циклів, в яких регістр RCX виконує роль лічильника. Так, у наступному фрагменті команда ADD виконується 10 разів:

```
        mov         ecx,0Ah
loop_start:
        add         eax,ecx
        loop        loop_start
```

Команда LOOP повністю еквівалентна парі команд:

```
        dec         ecx
        jnz         мітка
```

Але команда LOOP коротша за ці дві команди на один байт і не змінює значення прапорців. Розглянемо виконання цієї команди у прикладі 7.8.

Приклад 7.8. Виконати додавання парами двох масивів із числами розміром у байт кожне.

Лістинг 7.8. Програма виконання прикладу 7.8:

```
        title CopyRight by  Rysovaniy A.N.
        .686                    ; директива определения типа микропроцессора
        .model flat             ; задание линейной модели памяти
   .data                                    ; директива определения данных
   mas1 db 1,-2,3,-4,5          ; первый массив чисел размером в байт каждый
        len equ $-mas1          ; вычисление количества байтов массива mas1
   mas2 db -2,3,-4,5,-6         ; другой массив чисел размером в байт каждое
   res db  5 dup(0ffh)          ; заполнение пяти байтов массива res FFh
        .code                    ; директива начала сегмента команд
_start:                                 ; метка начала программы
        mov ecx, len              ; занесение числа байтов массива mas1
        lea eax,mas1              ; занесение адреса массива mas1
        lea ebx,mas2              ; занесение адреса массива mas2
        lea esi,res               ; занесение адреса массива res
m1:  mov dl,[eax]                 ; занесение байта массива mas1
       add dl,[ebx]       ; сложение байта из массива mas1 с байтом массива mas2
        mov byte ptr[esi],dl    ; занесение результата к новому массиву
        inc eax                   ; увеличение адреса массива mas1
        inc ebx                   ; увеличение адреса массива mas2
        inc esi                   ; увеличение адреса массива res
     loop m1                      ; перейти на метку M1, если ECX ≠ 0
        ret                       ;  возврат управления  ОС
        end _start              ; директива завершение программы с именем _start
```

У наведеному прикладі масив результату для наочності заповнюється всіма одиницями (0ffh). Це необхідно Для того щоб при налагодженні програми було видно в покроковому режимі змінення змісту комірок.

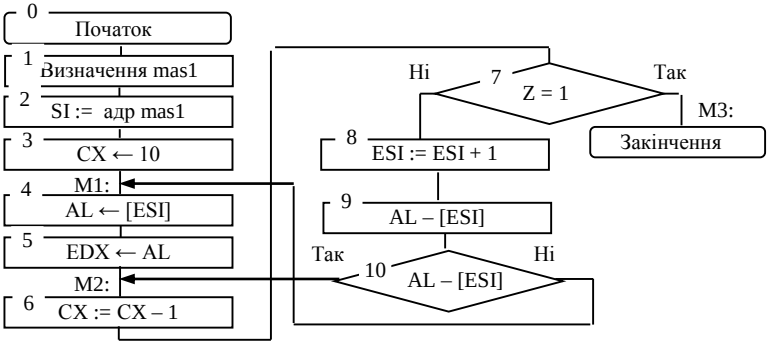
У наведеному прикладі при занесенні адрес масивів у регістри, наприклад, командою lea слід обов'язково використовувати тільки 32-розрядні регістри, тому що при звертанні до пам'яті необхідно мати всю повну адресу, а не її частину.

LOOPE мітка	Цикл, поки дорівнює
LOOPZ мітка	Цикл, поки нуль
LOOPNE мітка	Цикл, поки не дорівнює
LOOPNZ мітка	Цикл, поки не нуль

Всі перераховані команди зменшують регістр ECX на одиницю, після чого виконують перехід типу short, якщо ECX не дорівнює нулю і якщо виконується певна умова. Для команд LOOPE і LOOPZ такою умовою є рівність ZF = 0; для команд LOOPNE і LOOPNZ – рівність ZF = 0. Самі команди LOOPcc не змінюють значень прапорців, тому прапорець ZF повинен бути встановлений (або скинутий) попередньою командою.

Приклад 7.9. Навести програму пошуку максимального беззнакового числа.

Блок-схему виконання прикладу наведено на рисунку



У програмі, що наводиться нижче, максимальне беззнакове число в масиві адресується регістром ESI. Регістр CX містить кількість елементів у масиві mas1. Після закінчення програми в регістрі EDX знаходиться максимальне значення елемента масиву.

Спочатку як максимальний приймається перший елемент масиву mas1, а потім з ним порівнюються наступні елементи. Якщо поточний елемент більший від раніше знайденого максимуму, він заміщає його в регістрі AL.

Лістинг 7.9. Програма виконання прикладу 7.9:

```
 title CopyRight by Rysovaniy A. N.
 .686                ; директива определения типа микропроцессора
 .model flat         ; задание линейной модели памяти
.data                ; директива определения данных
 mas1 db 1,9,1,2,8,10,3,4,6,5
.code                ; директива начала сегмента команд
_start:              ; метка начала программы с именем _start
 lea esi,mas1        ; занесение адреса массива mas1
 mov cx,10           ; занесение счетчика количеством элементов
```

```
m1:  mov al,[esi]       ; занесение числа из памяти, адрес которого в esi
       movsx edx,al              ; сохранение максимума
m2:  dec cx                      ; уменьшение счетчика
       jz m3;                 ; переход на метку m3, если cx = 0
       inc esi                ; увеличение адреса массива чисел на единицу
       cmp al,[esi]              ; сравнение двух байтов
       jb m1                     ; перейти, если al < [esi]
       jmp  m2                   ; продолжение поиска
m3:  ret                         ; возврат управления  ОС
       end _start                ; закінчення програми з іменем _start
```

Максимальне число зберігається в Рг EAX. В дампі пам'яті даних видно, як розташовуються дані: початкові дані – за меншими адресами.

Пример 7.10. Сложить нулевой, четвертый и шестой элементы массива.

Лістинг 7.10. Програма виконання прикладу 7.10:

```
      .686      ; директива определения типа микропроцессора
      .model flat,stdcall ; задание линейной модели памяти
      option casemap:none ; отличие строчных и прописных букв
    include \masm32\include\windows.inc ; файлы структур, констант …
    include \masm32\macros\macros.asm
    uselib user32,kernel32,masm32
.data
    mas1 dd 1,2,3,4,5,6,7
      titl db "Сложение элементов массива",0
      inf1 db "Нажата кнопка ОК",0
      inf2 db "Нажата кнопка CANCEL",0
      buf dd 0,0              ; буфер для вывода сообщения
      ifmt db "Сумма 0-го, 4-го и 6-го элементов массива", 0dh,0ah,0ah,
      "Старшая часть результата: %d, младшая часть результата: %d ",
      "Автор программы:            , НТУ ХПИ",0
.code
_st:
      xor edx,edx
      mov eax,mas1[0]
      add eax,mas1+12
      adc edx,0
      add eax,mas1[24]
      adc edx,0
      invoke wsprintf,ADDR buf,ADDR ifmt,edx,eax
      fn MessageBox,0,addr buf,addr titl,MB_OKCANCEL or MB_ICONINFORMATION
.IF eax == IDOK
      fn MessageBox,0,addr inf1, chr$("Обработка кнопки ОК"),MB_ICONWARNING
.ELSE
      fn MessageBox,0,addr inf2, chr$("Обработка кнопки CANCEL"),
      MB_ICONERROR
.ENDIF
      invoke ExitProcess,0
      end _st
```

Всвязи с тем, что числа массива имеют размерность DD, то и смещение в дампе памяти соседних элементов составляет также 4 байта. Ну-

левой элемент массива имеет нулевое смещение mas1[0], четвертый элемент имеет смещение mas1+12, а шестой – mas1[24].

Контрольні запитання

1. Написати програму обчислення виразу

$$x^5/120 + x^4/12 + 7x^3/24 + 5x^2/12 + x/5$$

та довести, що при цілому x значення виразу є цілим числом.

2. Написати програму обчислення виразу

$$n^2 + 3n + 5$$

та довести, що ні за якого n вираз не ділиться на 121.

3. Написати програму обчислення суми кубів трьох послідовних цілих чисел та довести, що такий результат ділиться на 9.

4. Написати програму обчислення виразу

$$(x - 1)(x - 3)(x - 4)(x - 6) + 10$$

та довести, що за всіх дійсних значень x отримується позитивний результат виразу.

5. Написати програму знаходження найменшого значення виразу

$$(x - 1)(x - 2)(x - 3)(x - 4) + 10.$$

6. Написати програму доведення того, що числа 49, 4489, 444889, …, які отримуються додаванням числа 48 в середину попереднього числа, є точним квадратом.

7. Написати програму доведення того, що для додатних чисел a, b, c має місце нерівність

$$a/b + b/c + c/a \geq 3,$$

причому рівність має місце при $a = b = c$.

8. Написати програму доведення того, що для всіх додатних чисел a_1, a_2, …, a_n має місце нерівність

$$a_1/a_2 + a_2/a_3 + \ldots + a_n/a1 \geq n.$$

9. Написати програму доведення того, що

$$1/2 \cdot 3/4 \cdot 5/6 \ \ldots \ 99/100 < 1/10.$$

10. Написати програму знаходження цілих чисел n, для яких сума

$$1! + 2! + 3! + \ldots + n!$$

є повним квадратом.

8. ПРОЦЕДУРИ

Процедура – частина коду, призначена для багатократного використання в різних частинах програми. У спрощеному вигляді процедура – це пойменований блок команд, який описано за допомогою директив PROC та ENDP. Процедура завжди закінчується командою RET.

При виклику процедури керування передається за адресою її першої команди, тобто завжди в одне і те саме місце. Повернення з процедури відбувається в точку виклику, тобто за різними адресами.

Процедури необхідні для зменшення складності програми, зменшення часу пошуку помилок у програмі, а також Для того щоб зробити програму зрозумілою і керованою. Якщо програму зробити з незалежних модулів-процедур, то час пошуку помилок різко зменшується.

8.1. Виклик (команда CALL) процедури та її повернення (команда RET)

Виклик процедури	Повернення з процедури
CALL rel16/32 CALL r/m(16,32,64)	RET RET imm16

Команда CALL зберігає в стеку адресу наступної команди та здійснює передачу керування за адресою, заданою операндом (аналогічно команді JMP).

RET число **RETN** число **RETF** число	Повернення з процедури

Команда RET відновлює вміст RIP із стека, потім збільшує указівник стека на imm16. Команда може бути використана для очищення стека від параметрів, які були передані в процедуру через стек і більше не потрібні.

Команда **RETN** (для 32-розрядного режиму) зчитує зі стека слово (або подвійне слово, залежно від режиму адресації) і завантажує його в регістр IP (або EIP), виконуючи тим самим дії, протилежні близькому виклику процедури командою CALL. Команда RETF завантажує зі стека IP (EIP) і CS, повертаючись із далекої процедури. Якщо в програмі вказано команду RET, асемблер замінить її на команду RETN або RETF залежно від того, як було описано процедуру, яку ця команда завершує.

Операнд для RET необов'язковий, але, якщо він присутній, після зчитування адреси повернення зі стека буде видалено вказану кількість

байтів. Це потрібно, якщо при виклику процедури їй передавалися параметри через стек.

8.2. Угоди про виклик функцій

Угоди про виклик функцій визначають:

– спосіб передання параметрів (через регістри, стек, пам'ять або комбіновано);

– порядок розміщення параметрів (зліва направо або справа наліво);

– розрядність параметрів, що передаються;

– спосіб повернення значення;

– адресат відновлення стека (функції, що викликаються або викликають);

– регістри, які підлягають відновленню;

– правила перетворення імені функції для розміщення в таблицю перехресних посилань.

Властивості угод про виклик функцій у Windows 64 наведено в табл. 8.1.

Таблиця 8.1

Властивості	Windows 64
Параметри (адреси і цілочислові)	RCX, RDX, R8, R9 останні в стеку
Повернуте значення	RAX
Відновлення стека	Функція, що викликає
Регістри, які можна змінювати	RAX, RCX, RDX, R8, R9, R10, R11
Регістри, які необхідно відновити	RBX, RBP, RSP, RDI, RSI, R12, R13, R14, R15

Для спрощення виклику функцій на практиці користуються таким прийомом:

– місце для локальних змінних відводиться в стеку;

– для спрощення програмування значення указівника стека при вході у функцію копіюється в який-небудь регістр, відносно якого адресуються і параметри, і локальні змінні. Це дає можливість використовувати стек без перерахування зсувів відносно RSP.

Передача аргументів і угоди іменування

Компілятором Visual C/C++ підтримуються такі угоди про виклики:

- cdecl – параметри поміщаються в стек в зворотному порядку (справа наліво). Очищає стек зухвалий об'єкт;
- clrcall – параметри поміщаються в стек зліва направо. Стек не очищається;

- _ stdcall – параметри поміщаються в стек в зворотному порядку (справа наліво). Очищає стек об'єкт, що викликається;
- fastcall – параметри зберігаються в регістрах, потім поміщаються в стек. Очищає стек об'єкт, що викликається;
- thiscall – параметри поміщається в стек; покажчик this зберігається в ECX. Очищає стек об'єкт, що викликається;
- vectorcall – параметри зберігаються в регістрах, потім поміщаються в стек. Очищає стек об'єкт, що викликається.

Якщо передавати параметри через стек, кожного разу його настроюючи у функції або в точці виклику, то для проходження ланцюжка викликів коротких функцій будуть порівняно великі витрати: дуже багато копіювання параметрів. Для таких випадків був заведений тип програмної угоди fastcall, при якому аргументи передаються через регистри.

За угодою C *(Ci)* вирівнювання стека займається зухвала процедура, а за угодою PASCAL – така, що викликається. Угода *C* застосовують, коли невідомо, як багато параметрів будуть передані функції, як наприклад, у функції wsprintf(), коли не відомо скільки параметрів будуть покладено в стек.

8.3. Команди керування стеком при виконанні процедур

INT число	Виклик переривання

Команда **INT** аналогічно команді CALL поміщає в стек вміст регістрів EFLAGS, CS і EIP, після чого передає керування програмі, яка називається оброблювачем переривань із указаним як операнд номером (числом від 0 до 0FFh). У реальному режимі адреси оброблювачів переривань зчитуються з таблиці, що починається в пам'яті за адресою 0000:0000h. Адреса кожного оброблювача займає 4 байти. Тому, наприклад, оброблювач переривання 10h знаходиться в пам'яті за адресою 0000:0040h. У захищеному режимі адреси оброблювач ів переривань знаходяться в таблиці IDT і недоступні для прямого зчитування або запису, тому для установлення власного оброблювач а програма повинна звертатися до операційної системи. У DOS виклики переривань використовуються для виконання більшості системних функцій: роботи з файлами, введенням/виведенням і т.д.

Наприклад, наступний фрагмент коду завершує виконання програми і повертає керування DOS:

```
mov    ax,4C01h
int    21h
```

IRET **IRETD**	Повернення з оброблювача переривання

Команда **IRET** завантажує зі стека значення IP, CS і FLAGS, а команда IRETD – значення регістрів EIP, CS і EFLAGS відповідно. Єдина

відмінність IRET від RETF полягає в тому, що значення регістра прапорців відновлюється, через що багатьом оброблювачам переривань доводиться змінювати величину EFLAGS, що знаходиться в стеку, щоб, наприклад, повернути прапорець CF, встановлений у разі помилки.

INT3	Виклик переривання 3

Розмір команди **INT3** – один байт (код 0CCh), що робить її зручною для доведення програм налагоджувачами, які працюють у реальному режимі. Вони записують цей байт замість першого байта команди, перед якою потрібна точка зупину, і перевизначають адресу оброблювача переривання 3 на відповідну процедуру усередині налагоджувача.

INTO	Виклик переривання 4 при переповнюванні

Команда **INTO** – ще одна спеціальна форма команди INT. Вона викликає оброблювач переривання 4, якщо прапорець OF встановлений в 1.

BOUND індекс, межі	Перевірка виходу за межі масиву

Команда **BOUND** перевіряє, чи не виходить значення першого операнда (регістра), узяте як число зі знаком, за межі, вказані в другому операнді (змінній). Межі – два слова або подвійне слово (залежно від розрядності операндів), що розглядаються як цілі зі знаком і розташовані в пам'яті підряд. Перша межа вважається нижньою, друга – верхньою. Якщо індекс менший від нижньої межі або більший від верхньої, викликається переривання 5 (або виключення #BR), причому адреса повернення вказує не на наступну команду, а на команду BOUND. Тому оброблювач повинен виправити значення індексу або меж, перш ніж виконувати команду IRET.

ENTER розмір, рівень	Вхід у процедуру

Команда **ENTER** створює стековий кадр заданого розміру і рівня вкладеності (обидва операнди – числа; рівень вкладеності може набувати значення тільки від 0 до 31) з метою виклику процедури, що використовує динамічний розподіл пам'яті в стеку для своїх локальних змінних. Так, команда

```
enter     2048,3
```

поміщає в стек указівники на стековий кадр поточної процедури і тієї процедури, з якої викликалася поточна, створює стековий кадр розміром 2 кбайти для процедури, що викликається, і поміщає на EBP адресу початку кадра.

Команда ENTER imm16,0 створює стек процедури, що викликається:

– зберігає в стеку регістр RBP;

– копіює RSP в RBP (в результаті RBP вказує на своє збережене значення);

– додає до RSP значення першого операнда, тим самим виділяє в стеку ділянку локальних змінних.

Команда ENTER imm16, imm8 призначена для підтримки правил видимості для вкладених процедур.

LEAVE	Вихід із процедури

Команда **LEAVE** виконує дії, протилежні команді ENTER. Фактично LEAVE тільки копіює вміст EBP в ESP, тим самим викидаючи зі стека весь кадр, створений останньою виконаною командою ENTER, і зчитує зі стека значення регістра EBP для попередньої процедури, що одночасно відновлює і значення, яке мав регістр ESP до виклику останньої команди ENTER.

Команда LEAVE відновлює стек процедури, що викликає, тобто:

– копіює RBP в RSP;

– відновлює RBP зі стека.

8.4. Директива LOCAL

Директива LOCAL призначена для оголошення однієї або декількох локальних змінних ***усередині процедури.*** У початковому коді вона повинна розташовуватися відразу за директивою PROC. Синтаксис директиви LOCAL такий:

LOCAL *Список змінних*

Тут список змінних розуміється як перелік описів змінних, розділених комою, який може займати декілька рядків. Опис кожною змінною задається в такому вигляді:

ім'я: тип

Як ім'я змінної можна вибрати будь-який допустимий у мові асемблера ідентифікатор. Як тип можна задати один із вбудованих типів мови асемблера, таких, як WORD, DWORD та ін., або один із нестандартних типів, визначених програмістом. Часто локальні мітки починаються з префікса @@, хоча асемблер дозволяє задавати й інші двосимвольні префікси. Ділянка дії локальної мітки – тільки процедура, в якій вона описана. Така ж за виглядом локальна мітка, але введена в іншій процедурі, буде мати інше значення.

Приклад 8.1. У процедурі Proc1 створюються три локальні змінні. Одна з них має ім'я temp1 і займає подвійне слово, інша називається temp2 і займає в пам'яті два байти, а третя називається temp3 і займає в пам'яті один байт:

```
Proc1 PROC                              ; процедура з ім'ям Proc1
        LOCAL temp1:DWORD, temp2:WORD, temp3:BYTE
```

Приклад 8.2. Змінна MasArray є масивом із десяти подвійних слів. Розмір масиву указується в квадратних дужках:

```
LOCAL MasArray[10]:DWORD
```

8.5. Директива INVOKE

Директива INVOKE є зручним засобом виклику процедур і по суті замінює команду CALL. Вона дозволяє передати в процедуру декілька аргументів. Синтаксис директиви INVOKE такий:

INVOKE *Ім'я_процедури* [, *Список_аргументів]*

Аргументи, які передаються процедурі в директиві INVOKE, перераховуються через кому та можуть бути відсутніми. Різниця між директивою INVOKE і командою CALL: у тому, що у останньої немає списку аргументів.

У директиві INVOKE можна вказати довільну кількість аргументів.

Приклад 8.3. У наведеному нижче фрагменті коду за допомогою директиви INVOKE викликається процедура ADDW**,** якою передаються два 32-розрядних цілих числа:

```
dig1  DWORD  12345678h   ; збереження в пам'яті 32-розрядних числа
dig2  DWORD  56ABCDEFh  ; збереження в пам'яті 32-розрядних числа
      .code
invoke AddW,dig1,dig2     ; директива виклику процедури з параметрами
```

Без директиви INVOKE довелося б перед викликом команди CALL помістити в стек значення змінних dig1 і dig2, причому в зворотному порядку, як прийнято в мові С++:

```
push  dig2     ; збереження в стеку змінної dig2
push  dig1     ; збереження в стеку змінної dig1
call  AddW     ; виклик процедури (підпрограми) з ім'ям AddW
```

8.6. Оператор ADDR

Оператор ADDR використовується в директиві INVOKE для того щоб передати в процедуру *вказівник* на змінну (тобто адресу змінної), а не значення самої змінної. Такий спосіб передання аргументів називається *передачею параметрів за посиланням.* Наприклад, у наведеному нижче фрагменті коду в процедуру FillArray передається адреса масиву myArray**:**

```
INVOKE  FillArray, ADDR myArray
```

Приклад 8.4. У наведеному нижче фрагменті коду викликається процедура FillArray**,** якій передається адреса масиву байтів:

```
myArray BYTE   50 DUP(?)
.code
  INVOKE   FillArray, ADDR myArray ;  адреса  масиву
```

Приклад 8.5. Нижче показано, як можна передати в процедуру FillArray адреси двох перших елементів масиву подвійних слів:

```
.data
  Array    DWORD   10  DUP(?)
.code
  …
  INVOKE FillArray,  ADDR Array, ADDR Array+4
```

Оператор addr використовується для передання адреси мітки (і не тільки) функції. Він дійсний в контексті директиви invoke. Ось деякі відмінності між ними:

– оператор addr не може бути використаний з мітками, які визначені в програмі раніше, а offset може;

– оператор addr підтримує локальні змінні, тоді як offset не підтримує.

8.7. Директива PROTO

Директива PROTO створює **прототип** існуючої процедури. У прототипі описується ім'я процедури і список її параметрів.

Прототипи дозволяють викликати процедуру до того, як вона буде формально визначена.

Директива PROTO повинна передувати в початковому файлі операторові INVOKE, наприклад:

```
MyProg   PROTO              ; прототип процедури
  INVOKE   MyProg           ; виклик процедури
  MyProg   PROC    ; тіло процедури
  …
  MyProg   ENDP
```

Застосовується варіант, коли тіло процедури знаходиться в програмі перед директивою INVOKE, яка її викликає. В цьому випадку оголошенням прототипу процедури вважається директива PROC:

```
  MyProg    PROC ; тіло процедури
  …
  MyProg   ENDP
  INVOKE  MyProg              ; директива виклику процедури
```

Раніше написана процедура може бути змінена для використання з директивою PROTO. В цьому випадку вносяться такі зміни:

– ключове слово PROC замінюють на PROTO;

– видаляють ключове слово USES і наступний за ним список регістрів, якщо вони вказані при оголошенні процедури.

8.8. Оператор USES

Оператор USES, вказаний відразу після директиви PROC, дозволяє перерахувати імена всіх регістрів, значення яких змінюється в процедурі. При обробці такого оператора компілятор асемблера виконує дії:

1. На початку процедури компілятор автоматично генерує послідовність команд PUSH, за допомогою яких в стеку зберігаються значення регістрів, указаних в операторі USES.

2. При виході з процедури (точніше, перед кожною командою RET, якщо в процедурі їх декілька) він автоматично відновлює значення цих регістрів.

Оператор USES указується відразу за ключовим словом PROC. Список регістрів надходить відразу за ключовим словом USES, при цьому імена регістрів розділяються пропуском або символом табуляції (не комою!).

8.9. Організація процедур та їх дослідження

При організації процедур необхідно визначити заголовок процедури та її закінчення. Розглянемо організацію процедур на прикладі програми з простою процедурою складання двох чисел (лістинг 8.1).

Лістинг 8.1. Програма з простою процедурою складання двох чисел:

```
      .686                    ; директива определения типа микропроцессора
.model flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
   option casemap:none        ; отличие строчных и прописных букв
    .code                  ; директива начала программы
_start:                              ; метка начала программы с именем _start
       mov ax,2
       mov bx,3
       call AddW                  ; вызов процедуры AddW
       ret                        ; возврат управления  ОС
          AddW proc               ; процедура AddW
          add ax,bx
           ret                    ; возвращение из процедуры
          AddW endp               ; окончание процедуры AddW
     end _start                   ; завершение программы с именем _start
```

У цій програмі можна виділити заголовок AddW proc, що складається з імені процедури і слова proc – ознаки закінчення процедури, що складається з імені і слова endp, та «тіло» процедури, тобто виконувані нею інструкції. Процедура викликається інструкцією call <ім'я>. Після виклику виконуються інструкції, з яких складається процедура, і потім мікропроцесор переходить до інструкцій, що йдуть безпосередньо за викликом процедури.

Регістр eip містить адресу поточної команди. ***Команда call запам'ятовує в стеку адресу повернення і завантажує на eip адресу першої інструкції процедури. Коли виконуються інструкції процедури, eip міняється автоматично – залежно від самих інструкцій. Коли ж мікропроцесор доходить до команди ret, із стека береться адреса повернення, завантажується в eip, і мікропроцесор прямує за вказаною адресою.***

При виклику однієї процедури стек не має великого значення. Але якщо одна процедура викликає іншу, то стек необхідний. В цьому випадку стек спочатку збереже адресу повернення в основну програму. Далі мікропроцесор перейде до виконання процедури і займатиметься цим, поки не зустріне інструкцію call. Після цього він запам'ятає в стеку адресу повернення в процедуру і знову перейде до команд, розташованих вже в іншій ділянці пам'яті. Натикнувшись на команду ret, мікропроцесор зніме з вершини стека адресу повернення в процедуру, а знову натикнувшись на ret, поверне із стека адресу повернення в основну програму і, якщо не буде інших викликів, там і закінчить свою роботу.

Розглянемо приклад передання параметрів через стек із використанням процедур (лістинг 8.2).

Лістинг 8.2. Програма передання параметрів через стек із використанням процедур:

```
    .686          ; директива определения типа микропроцессора
    .model  flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
    option casemap:none   ; отличие строчных и прописных букв
.code                                   ; директива начала программы
_start:                                 ; метка начала программы с именем _start
    push dword ptr 3
    push dword ptr 4
    call AddW                           ; вызов процедуры
    ret                                 ; повернення управління  ОС
      AddW proc                         ; начало процедуры
      mov eax,[esp+8]                   ; eax = 3
      add eax,[esp+4]          ; eax = 7
     ret 8   ; возвращение из процедуры и увеличения на 8  указателя стека
          AddW endp             ; окончание процедуры
      end _start                ; завершение программы с именем _start
```

Команда push dword ptr 3 записує в стек подвійне слово (4 байти), що містить число 3. Асемблер MASM записав би подвійне слово і за командою push 3, але надійніше указувати розмір числа явно.

У стеку зберігаються три 4-байтових числа, першим (його адреса найбільша) йде число 00000003, потім число 00000004 і, нарешті, на вершині стека знаходиться адреса повернення.

Таким чином, число 3, поміщене в стек першим, відстає від вершини на 8 байтів, а число 4 – на 4. Тому параметри процедури матимуть адреси esp+4 і esp+8 [8].

Команда ret 8 видаляє 8 байтів зі стека (очищує стек).

Зазвичай програмам допомагає взаємодіяти з навколишнім світом операційна система, яка бере на себе всі деталі взаємодії із зовнішніми пристроями. У системі Windows цьому служать процедури Windows API, які викликаються так само, як і будь-яка інша процедура асемблера – інструкцією call. Параметри, необхідні цим функціям, передаються через стек.

Розглянемо процедуру ExitProcess, яку викликає кожна Windows-програма, щоб завершити свою роботу. Дотепер замість ExitProcess ми користувалися простою інструкцією повернення ret. Але ExitProcess діє набагато правильніше, не тільки повертаючи керування операційній системі, але і звільняючи зайняті програмою ресурси.

Розглянемо програму для Windows, яка тільки правильно завершується (лістинг 8.3).

Лістинг 8.3. Програма з правильним завершенням:

```
    .686                    ; директива определения типа микропроцессора
    .model  flat, stdcall  ; задание линейной модели памяти и соглашения ОС Windows
    option  casemap:none          ; отличие строчных и прописных букв
    ExitProcess   proto :dword              ; прототип функции
    includelib \masm32\lib\kernel32.lib    ; вызов библиотеки kernel32.lib
.code                                       ; директива начала программы
  _start:                          ; метка начала программы с именем _start
    push 0            ; запись в стек кода завершения программы для ExitProcess
call ExitProcess ; возврат управления  ОС Windows та вивільнення ресурсів
      end _start                          ; завершение программы с именем _start
```

Процедура ExitProcess, що викликається в ній, потребує одного параметра – коду завершення, який повертається операційній системі. Він передається процедурі командою push 0. Число 0 вважається ознакою вдалого завершення. Код завершення залежно від обставин може бути іншим.

Оскільки ExitProcess є зовнішньою процедурою, яка не визначена в нашій програмі, асемблер повинен знати, де вона знаходиться, а також кількість і розмір її параметрів.

Відомості про адресу і параметри процедури зберігаються у файлі бібліотеки kernel32.1ib, який підключається до асемблерного тексту директивою

Includelib \masm32\lib\kernel32.lib

Перш ніж створити інструкцію виклику цієї процедури, компонувальник порівнює відомості з бібліотеки з *прототипом* ExitProcess proto :DWORD і, якщо все збігається, створює придатний до виконання файл із розширенням .exe. Прототип процедури дуже простий і складається з імені, слова proto і параметрів. У нашому випадку параметр один – це подвійне слово (тобто 4 байти) DWORD. Якщо параметрів у процедури декілька, вони розділяються в списку комами.

Команди push <параметр> і виклик процедури call в асемблері фірми Microsoft можна скорочено записати як invoke <ім'я>. параметр, параметр... Мінімальна програма, що використовує виклик процедури invoke, має вигляд, наведений у лістингу 8.4.

Лістинг 8.4. Мінімальна програма, що використовує виклик процедури invoke:

```
    .686                    ; директива определения типа микропроцессора
.model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
```

```
    option  casemap: none            ; отличие строчных и прописных букв
   ExitProcess   proto: dword        ;  прототип API- функции
   includelib \masm32\lib\kernel32.lib  ; вызов библиотеки kernel32.lib
.code                                ; директива начала программы
     _start:                         ; метка начала программы с именем _start
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
   end    start                      ; завершение программы с именем _start
```

Директива **invoke** *– не команда мікропроцесора, а лише зручний запис виклику процедури, яка буде перетворена асемблером на команди* **push** *(їх буде стільки, скільки параметрів у процедури) та* **call**.

Розглянемо програму, яка обчислює суму і різницю двох цілих чисел із використанням двох процедур (лістинг 8.5).

Лістинг 8.5. Програма обчислення суми та різниці двох цілих чисел із використанням процедур:

```
      .686                       ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
   option casemap: none           ; отличие строчных и прописных букв
   .data                          ; директива определения данных
   op1 dd -27       ; запись в 32-разрядную ячейку op1 первого операнда
   op2 dd  38       ; запись в 32-разрядную ячейку op2 второго операнда
   rez dd 2  dup(0)         ; резервирование в памяти ячейки 32 × 2 под rez
   .code                        ; директива начала программы
_start:                         ; метка начала программы с именем _start
      call Addw                   ; вызов подпрограммы сложения
      mov rez,eax                ; сохранение в памяти результата сложения
      call Subw                        ; вызов подпрограммы вычитания
      mov rez+4,eax      ; сохранение в памяти результата вычитания
      ret                         ; возврат управления  ОС
  Addw proc                  ; начало процедуры с именем Addw
  mov eax,op1                     ; занесення першого операнда op1 в eax
  add eax,op2 ; додавання двох операндів та збереження результату в eax
  ret                             ; возвращение из процедуры
      Addw endp                   ; окончание процедуры с именем Addw
          Subw proc               ; начало процедуры с именем Subw
          clc                     ; сброс флажка переноса
             mov eax,op1          ; занесение первого операнда op1 в eax
             sbb eax,op2     ; (op1 – op2) и сохранение результата  в eax
          ret                     ; возвращение из процедуры
          Subw endp               ; окончание процедуры с именем Subw
      end _start                  ; завершение программы с именем _start
```

Стан програми після виконання останньої команди основної програми (ret) наведено на рис. 8.1.

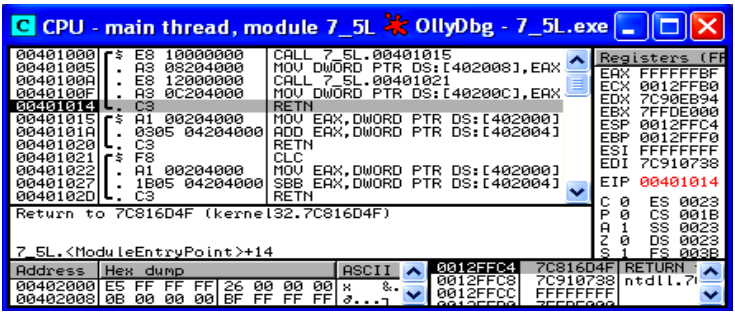


Рис. 8.1. Стан програми з лістингу 8.5 після виконання останньої команди основної програми (ret)

Мітка start необхідна, щоб компонувальник відрізняв початок та закінчення програми. Процедура Addw proc виконує додавання двох чисел розміром у 4 байти, процедура Subw proc – відповідне віднімання.

У програмі за адресою 00402008h розміщується результат виконання процедури Addw proc, а за адресою 0040200Ch – результат виконання процедури Subw proc.

У наступній програмі використаємо процедуру Windows API – ExitProcess. Текст модернізованої програми розглядається в лістингу 8.6.

Лістинг 8.6. Модернізована програма із процедурою Windows API:

```
        .686            ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
option casemap: none        ; отличие строчных и прописных букв
include \masm32\include\kernel32.inc ; системные функцих приложений …
includelib \masm32\lib\kernel32.lib
.data                                ; директива определения данных
      op1 dd -27      ; определение у памяти переменной op1 размером в 32 разряда
      op2 dd 38       ; определение у памяти переменной op2 размером в 32 разряда
      rez dd 2dup(0)
.code                  ; директива начала программы
_start:                ; метка начала программы с именем _start
      call Addw              ; вызов процедуры сложжения
      mov rez,eax            ; сохранение в памяти результата сложения
      call Subw              ; вызов процедуры вычитания
      mov rez+4,eax          ; сохранение в памяти результата вычитания
invoke ExitProcess, 0  ; директива окончания программы
        Addw proc  ; початок процедури Addw
          mov eax,op1
          add eax,op2
          ret                          ; возвращение из процедуры
        Addw endp              ; окончание процедуры Addw
            Subw proc                  ; начало процедуры Subw
            clc                        ; сброс флажка CF
            mov eax,op1
            sbb eax,op2
            ret                        ; возвращение из процедуры
            Subw endp                  ; окончание процедуры Subw
      end _start                   ; завершение программы с именем _start
```

Вікна налагоджувача OllyDbg з модернізованою програмою наведено на рис. 8.2.

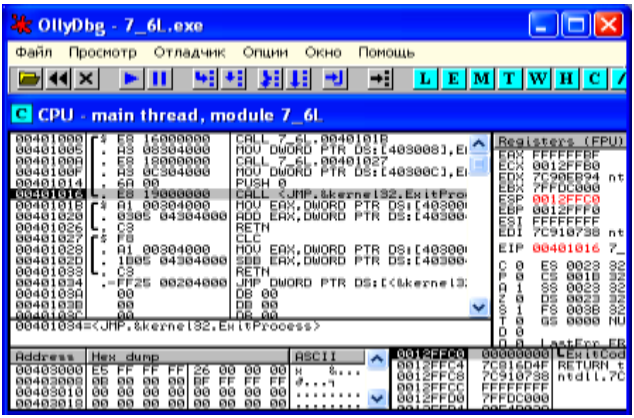


Рис. 8.2. Вікна налагоджувача OllyDbg з модернізованою програмою з лістингу 8.6

При цьому можна побачити зміни, які з'являються при застосуванні Windows API-функцій. Процедура invoke ExitProcess, 0 перетворюється у налагоджувачі на додаткову команду PUSH 0 для успішного завершення програми.

8.10. Windows API-подібні процедури

Відомо [6], що процедура Windows API має прототип та заголовок, в якому вказуються її параметри. Тому якщо оформити самостійно написану процедуру як процедуру Windows API **з параметрами**, то вона стає Windows API-подібною.

Як приклад розглянемо програму додавання двох чисел (лістинг 8.7).

Лістинг 8.7. Програма додавання двох чисел із використанням директиви invoke:

```
        .686                    ; директива определения типа микропроцессора
.model  flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
option casemap:none             ; отличие строчных и прописных букв
        includelib \masm32\lib\kernel32.lib  ; вызов библиотеки kernel32.lib
        include \masm32\include\kernel32.inc ; системные функции применений…
        AddDD proto :DWORD, :DWORD ; прототип процедуры
.code                           ; директива начала кода программы
_start:                         ; метка начала программы с именем _start
    invoke AddDD,2,3            ; вызов директивы AddDD с параметрами
    invoke ExitProcess,0        ; директива окончания программы
      AddDD proc arg1:DWORD,arg2:DWORD
        mov eax,[esp+8]             ; eax = 2
        add eax,[esp+12]            ; eax = 5
```

```
  ret                              ; возвращение из процедуры
AddDD endp              ; окончание процедуры с именем AddDD
end _start              ; завершение программы с именем _start
```

Стан програми після виконання команди add eax,[esp+12] наведено на рис. 8.3.

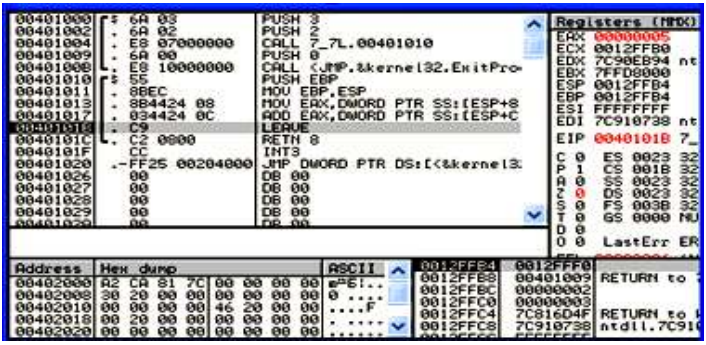


Рис. 8.3. Стан програми після виконання команди add eax,[esp+12]

Тепер у цій програмі перше число (2) відстає від вершини на 8 байтів, а друге – на 12. Налагоджувач виявляє в створеній процедурі багато сторонніх інструкцій:

```
push ebp
mov ebp,esp
mov eax,dword ptr ss:[esp+8]
add eax,word ptr ss:[esp+c]
leave
retn 8
```

Слово leave еквівалентне парі інструкцій

```
mov esp,ebp
pop ebp
```

Асемблер зберігає вказівник стека в регістрі ebp. Якщо регістр ebp не міняється усередині процедури, то регістр esp можна відновити перед виходом із неї.

Щоб при цьому зберегти ebp для зовнішнього застосування, його спочатку відправляють в стек, а перед поверненням із процедури знову виймають звідти. Саме через те що ebp зберігається в стеку, міняється положення параметрів процедури. ***Першим в стек відправляється число* 3, *потім* 2, *потім* ebp, *потім адреса повернення***. Отже, ebp відстає на 4 байти від вершини, число 2 – на 8, а число 3 – на 12 байт.

Значення регістра esp зберігається в регістрі ebp, оскільки асемблер не визначає, чи міняється вказівник стека, і тому зберігає його на випадок, якщо стек використовуватиметься для зберігання *локальних* змінних.

Локальні змінні необхідні процедурі тільки у момент її виконання, от чому для них не слід займати місце в комп'ютерній пам'яті, виділене директивою .data. Але якщо зберігати їх в стеку, вони виникнуть при вході в процедуру і зникнуть при виході з неї.

У лістингу 8.7 можна для простоти обійтися без передання безпосередніх даних через стек. Для цього необхідно замість команд

```
mov eax,[esp+8]
add eax,[esp+12]
```

використати команди

```
mov  eax,arg1
add eax,arg2
```

Всі інші команди залишаються без зміни.

8.11. Непрямий виклик процедур

Непрямий виклик процедур дозволяє використовувати різноманітні їх способи адресації:

```
call BX      ;  адреса підпрограми знаходиться в регістрі BX
call [BX]    ; адреса підпрограми знаходиться в елементі пам'яті,
                                                ; адреса якої розміщується в регістрі BX
call [BX][SI] ; у BX – адреса таблиці адрес підпрограм у SI індекс таблиці
tb1 [SI]  ; змінна tbl містить адресу таблиці адрес підпрограм
                                                ;   у SI – індекс в цій таблиці
```

Розглянемо програму, в якій потрібно проаналізувати масиви A та B, в кожному з яких по 10 елементів типу WORD. Необхідно визначити кількість елементів масиву Ai, що задовольняють умову $A_i \le B_i$, та використати передачу параметрів через таблицю адрес. Таку програму наведено у лістингу 8.8.

Лістинг 8.8. Передача параметрів через таблицю адрес:

```
      title CopyRight by Rysovaniy A. N. ; передача параметров через табл. адресов
      .686                ; директива определения типа микропроцессора
.model flat,stdcall  ; задание линейной модели памяти и соглашения ОС Windows
      .data                                          ; директива определения данных
 A1 DW 0,1,2,3,4,5,6,7,8,9  ; сохранение в массиве памяти с именем  A1
                                                ; чисел каждое в 16-разрядные ячейки
       B1 DW 9,8,7,6,5,4,3,2,1,0
       N1 DW 10
       _TA DD 4 DUP (0)
.code                ; директива начала кода программы
_start:              ; метка начала программы с именем _start
      mov _TA,offset A1          ; занесение адреса массива A1
      mov _TA+4,offset B1        ; занесение адреса массива B1
      mov _TA+8,offset N1 ; занесение адреса счетчика N1
      mov ebx,[_TA]       ; занесение адреса начала таблицы адресов
    call prog1  ;
    ret                          ; возврат управления  ОС
      prog1 proc
```

```
      mov cx,[ebx+40]          ; счетчик = 0Ah (10)
      movzx ecx,cx             ; заполнение старшей части регистра нулями
      mov si,[ebx]             ; выбрать 1-й элемент массива A1
      mov di,[ebx+20]          ; выбрать 1-й элемент массива B1
      xor edx,edx   ;
 m1: cmp si,di                 ; сравнение элементов массивов
        jl  m2                 ; перейти, если меньше
        jmp m3                 ; перейти в противоположном случае
 m2:  inc dx                   ; подсчитать количество элементов
 m3:  add si,2                 ; подготовка следующего элемента массива A1
        add di,2               ; подготовка следующего элемента массива B1
        loop m1                ; перейти на m1, если CX. ≠ 0
        ret                    ; возвращение из процедуры
      prog1 endp               ; окончание процедуры
      end _start               ; завершение программы с именем _start
```

У цій програмі, для того щоб заповнити нулями старшу частину лічильника, використано команду movzx ecx,cx, хоча можна спочатку заповнити нулями цей регістр, а потім завантажити його необхідним числом. У розглянутій програмі використовується одна процедура.

Головну програму, з якої викликаються інші процедури, також можна оформити у вигляді процедури. Таку програму, що обчислює суму та різницю двох цілих чисел із використанням трьох процедур та непрямого виклику двох додаткових, розглянуто в лістингу 8.9.

Лістинг 8.9. Програма обчислення суми та різниці двох цілих чисел із використанням непрямого виклику процедур:

```
    title CopyRight by Rysovaniy A. N.
    .686                  ; директива определения типа микропроцессора
    .model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
    option casemap: none    ; отличие строчных и прописных букв
.data                     ; директива определения данных
     tbl label dword  ; присвоение tbl двойного слова
    DD sub1 ; резервирование под sub1 в памяти ячеек
                                                ; размером в двойное слово (32 разряда)
    DD sub2 ; резервирование в памяти ячеек размером в двойное слово
        op1  DD -29           ; –29 = FFFF FFE3h
        op2  DD 59            ;   59 = 0000 003Bh
res DD 2 DUP(0) ; резервирование в памяти ячеек размером в два двойных слова
.code                          ; директива начала кода программы
_start:                        ; метка начала программы с именем _start
      _main proc               ; главная процедура
         lea   ESI, tbl              ; занесение в ESI адреса tbl
         mov  [ESI], offset sub1     ; занесение адреса процедуры sub1
         mov  [ESI+4], offset sub2 ;
         call  dword ptr [ESI]       ; непрямой вызов процедуры sub1
         call  dword ptr [ESI+4]     ; непрямой вызов процедуры sub2
         lea   EAX, res              ; занесение адреса результата в EAX
         ret                   ; возврат управления  ОС
      _main endp               ; окончание главной процедуры
      sub1 proc
         clc                         ; очистка флажка переноса
         mov EAX, op1
         adc EAX, op2
         mov res, EAX
         ret                         ; возвращение из процедуры
      sub1 endp
```

```
sub2 proc
   clc                    ; очистка флажка переноса
   mov EAX, op1
   sbb EAX, op2
   mov res+4, EAX
   ret                    ; возвращение из процедуры
sub2 endp                 ; закінчення процедури
end _start                ; завершение программы с именем _start
```

Мітка start при виконанні декількох процедур є обов'язковою. Значенню sub1 привласнюється адреса 0040101F початку цієї процедури, значенню sub2 – адреса 00401031 початку відповідної процедури.

У цій програмі для простоти не застосовуються Windows API-функції. Стан програми після виконання останньої команди основної програми (ret) наведено на рис. 8.4.

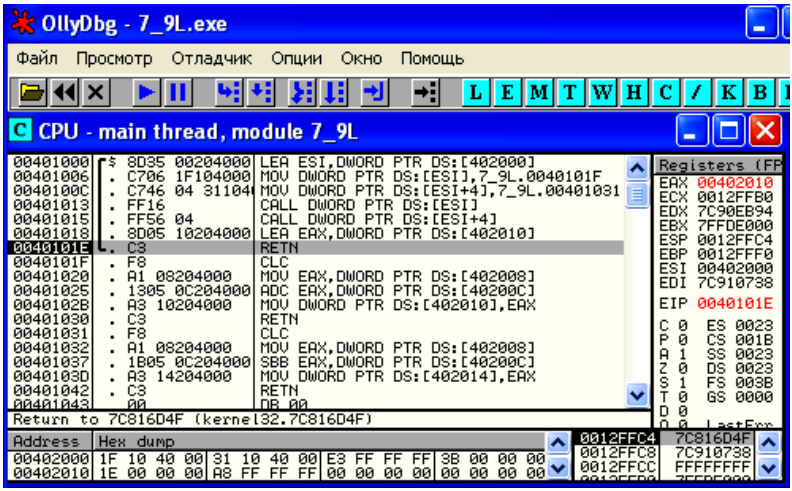


Рис. 8.4. Стан програми з лістингу 8.9 після виконання

Програма, що розглядається, складається з трьох процедур: 1) головної – з ім'ям _main; 2) такої, що виконує додавання, – з ім'ям sub1; 3) такої, що виконує віднімання, – з ім'ям sub2. Процедура sub1 обчислює суму чисел i1 та i2 і розташовує результат у молодше подвійне слово змінною res. Процедура sub2 виконує віднімання тих самих чисел та розміщення результату у старше подвійне слово змінної res.

Головна процедура _main викликає процедури за адресою, яка знаходиться у регістрі ESI, а регістр ESI отримує цю адресу з таблиці tbl, яка вміщує відповідну адресу. За командою lea ESI, tbl в ESI фактично заноситься адреса процедури sub1.

8.12. Використання загальних змінних у процедурах

При розробці великих програм виникає ситуація, коли потрібно обробляти ті самі обсяги даних декількома процедурами або навіть окремими програмами.

Загальні або, як їх ще називають, глобальні змінні дозволяють працювати з даними при мінімальному використанні стека, що економить мікропроцесорний час. Крім того, фіксовані прив'язки загальних змінних на етапі компонування прискорюють доступ до них.

Для роботи із загальними змінними в мові асемблера використовуються директиви public і extern.

Директива public оголошує змінну або процедуру, яка доступна для інших модулів, директива extern указує на те, що змінна або процедура є зовнішньою до виконуваної процедури. Обидві директиви застосовуються для компонування основної програми або процедури з декількох об'єктних модулів і є дуже зручними для побудови великих програм.

Глобальні змінні оголошують таким чином:

– у об'єктному модулі, де знаходиться така змінна, необхідно вказати на можливість доступу до неї за допомогою директиви public;

– у об'єктних модулях, з яких походить звернення до загальної змінної, необхідно оголосити її з директивою extern.

Тип, указаний у директиві extern, повинен відповідати дійсному типу об'єкта та може кодуватися одним із слів:

– BYTE;
– WORD;
– DWORD;
– QWORD;
– TWORD.

Розглянемо **приклад**. Нехай потрібно обчислити вираз $a_1 + a_2 - b_3$, де a_1, a_2, b_3, – цілі числа. Обчислення виконаємо за допомогою трьох процедур, розташованих у різних файлах: головної процедури _add_sub, процедури _add1, яка обчислює суму $a_1 + a_2$, та процедури _sub1, яка віднімає з цієї суми число b_3. Головна частина обов'язково повинна бути оформлена ***тільки у вигляді процедури***.

Підкреслення спереду відображає стандарт ANSI, який наказує всім зовнішнім іменам (доступним декільком модулям) автоматично додавати символ підкреслення.

Скільки б не під'єднувалося об'єктних модулів, лише один із них є головним, тому що з цього модуля починається виконання програми. ***Тільки головний модуль на початку сегмента коду містить мітку початку програми***, наприклад мітку START. Це ж ім'я указується після директиви END – транслятор повинен знати точку входу програми, щоб указати її в заголовку завантажуваного модуля.

При об'єднанні декількох модулів ***першим*** записується головний, а інші – в довільному порядку.

Результат обчислень повертається в головну процедуру (_add_sub). Всі три підпрограми реалізовано у вигляді окремих файлів ASM, які після компіляції утворюють три файли об'єктних модулів із розширенням .OBJ за допомогою команди

ml.exe /c /coff 06_07L. asm 06_07_1L.asm 06_07_2L.asm <enter>

За допомогою компонувальника (link або іншого) ці файли можна включити в 32-розрядне застосування, наприклад:

link.exe /subsystem:console 06_07L.obj 06_07_1L.obj 06_07_2L.obj <enter>

Зручніше створити **командний bat-файл** зі змістом:

ml.exe /c /coff 06_07L. asm 06_07_1L.asm 06_07_2L.asm
link.exe /subsystem:console 06_07L.obj 06_07_1L.obj 06_07_2L.obj

Після натискання на цей bat-файл, якщо відсутні помилки, буде створено exe-файл.

У підпрограмі _add_sub визначимо змінні a_1, a_2, і b_3 як подвійні числа, яким привласнимо конкретні значення. Оскільки всі змінні мають бути доступними з інших підпрограм, що знаходяться в інших об'єктних файлах, необхідно їх оголосити з директивою public. Крім того, зовнішні підпрограми _add1 та _sub1, що використовуються процедурою _add_sub, повинні бути оголошені з директивою extern. Текст підпрограми _add_sub наведено в лістингу 8.10.

Лістинг 8.10. Початковий текст підпрограми _add_sub:

```
 title CopyRight by Rysovaniy A. N.
 .686                    ; директива определения типа микропроцессора
 .model flat             ; задание линейной модели памяти
 .stack 100h
 option casemap: none    ; отличие строчных и прописных букв
extern _add1:proc ; процедура, которая принимается (указывание на внеш. проц.)
extern _sub1:proc ; процедура, которая принимается (указывание на внеш. проц.)
public a1, a2, b3        ; переменные, которые передаются в другие модули
 .data                   ; директива определения данных
   a1   DD  171          ; a1 = 0000 00ABh
   a2   DD  17           ; a2 = 0000 0011h
   b3   DD  34           ; b3 = 0000 0022h
 .code                   ; директива начала кода программы
_start:                  ; метка начала программы с именем _start
_add_sub proc
   clc                   ; очищаем флажок переноса
call _add1               ; вычисляем сумму a1 + a2
push EAX  ; промежуточный результат a1 + a2 = 0000 00BCh помещаем в стек,
      ; поскольку он будет использоваться процедурой _sub1 (esp = esp + 4)
call _sub1    ; вычисляем разницу полученной суммы (a1 + a2) і числа b3.
            ; Результат возвращается в основную программу через регистр EAX
   ret                   ; возврат управления ОС
 _add_sub endp           ; окончание процедуры
   end  _start           ; завершение программы с именем _start
```

У головній програмі початок програми обов'язково позначається міткою.

Якщо не вказати процедури як зовнішні (extern), то компілятор, не виявивши їх у поточному модулі, видасть помилку. Якщо не вказати змінні з атрибутом public, то при компіляції цього ASM-файлу ніяких помилок не буде, але вони з'являться в процесі компіляції інших модулів, де ці змінні використовуються (лістинг 8.11).

Лістинг 8.11. Текст процедури _add1:

```
      .686                    ; директива определения типа микропроцессора
      .model flat             ; задание линейной модели памяти
    option casemap: none      ; отличие строчных и прописных букв
      public _add1            ; доступ для других модулей
      extern a1: DWORD        ; указывание на внешнюю процедуру
      extern a2: DWORD        ; указывание на внешнюю процедуру
  .code                       ; директива начала кода программы
_add1 proc
      mov EAX, a1  ; EAX = 0000 00ABh
      add EAX, a2  ; EAX = 0000 00BCh
      ret          ; возвращение управления ОС (возвращение из процедуры)
_add1 endp                    ; окончание процедуры
      end                     ; окончание программы
```

Оскільки процедура _add1 використовується в підпрограмі _add_sub, то вона має бути оголошена як загальна (public) для зовнішніх щодо даного об'єктного файлу програм. Крім того, процедура _add1 має дві змінні, визначені в іншому об'єктному файлі (де визначена процедура _add_sub), тому вони повинні бути вказані як зовнішні (extern).

Результат складання повертається, як завжди, в регістр EAX. Проаналізуємо останню з трьох підпрограм – _sub1 (лістинг 8.12).

Лістинг 8.12. Текст процедури _sub1:

```
    .686            ; директива определения типа микропроцессора
    .model flat             ; задание линейной модели памяти
     option casemap: none   ; отличие строчных и прописных букв
    extern b3: DWORD      ; указывание на внешнюю процедуру
    public _sub1     ; доступ для других модулей
.code               ; директива начала кода программы
     _sub1 proc
      push EBP ; сохранение в стеке регистра базы (ESP = ESP + 8 – потому,
                                  ; что в главной программе была команда push  EAX)
    mov  EBP, ESP  ;
    mov EAX, dword ptr [EBP+8]   ; пересылка a1 + a2 = 0000 00BCh в EAX
    sub  EAX, b3
    pop  EBP
    ret  4  ; возобновление начальных значений и возвращение управления ОС
    _sub1 endp      ; окончание процедуры _sub1
     end            ; окончание программы
```

Процедура використовує змінну b_3 з модуля, в якому знаходиться підпрограма _add_sub, тому вона оголошена як зовнішня. Сама процедура

_sub1 оголошена доступною для програм з інших модулів, оскільки вона викликається із зовнішньої підпрограми _add_sub.

Процедура _sub1 викликається з _add_sub таким чином:

```
push   EAX
call _sub1
```

Стан програми після виконання команди pop ebp наведено на рис. 8.5.

Для звернення до пам'яті прийнято за умовчанням використовувати регістр EBP. Тому команда push EBP необхідна на той випадок, якщо якась зовнішня процедура звертається до пам'яті. А для того щоб значення EBP не змінилося, його спочатку необхідно зберегти, а після використання – відновити.

Значення змінних у шістнадцятковій системі числення можна побачити у вікні налагоджувача.

Процедурі передається один параметр через стек (сума a_1 і a_2). Для витягання параметра зі стека в процедурі _sub1 використовується регістр EBP. Перед поверненням в головну підпрограму в стеку знаходиться один параметр, поміщений туди при виклику, тому команда ret повинна очистити стек. Потрібно видалити 4 байти, що і робить команда ret 4.

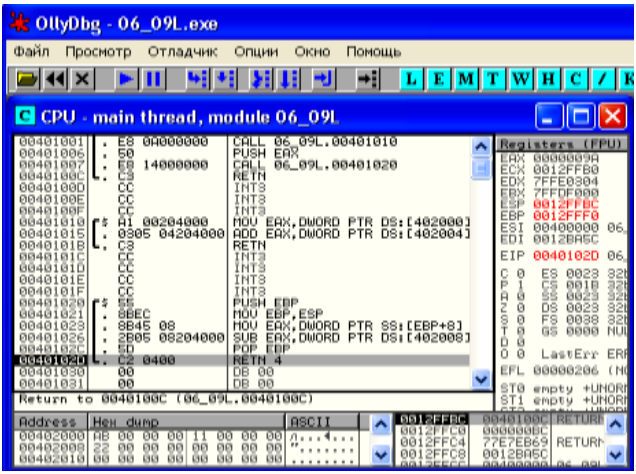


Рис. 8.5. Стан програми після виконання команди pop ebp

Як і в попередніх випадках, процедура повертає результат у регістр EAX. Таким чином, головна підпрограма _add_sub перед інструкцією повернення міститиме результат у регістрі EAX. Цей приклад демонструє основні аспекти використання загальних змінних у програмах на асемблері.

Приклад 8.6. Написати на асемблері програму обчислення виразу *a b+c/d.*

Програма виконання цього прикладу складається з трьох окремих файлів (лістинги 8.13–8.15).

Лістинг 8.13. Текст головної процедури:

```
        title CopyRight by Rysovaniy A. N.
        .686                    ; директива определения типа микропроцессора
        .model flat             ; задание линейной модели памяти
        .stack 100h
        option casemap: none    ; отличие строчных и прописных букв
  extern _ab:proc ; проц., какая принимается (указывание на внешние процедуры)
  extern _cd:proc ; проц., какая принимается (указывание на внешние процедуры)
        public _a, _b, _c, _d   ; переменные, которые передаются в другие модули
        .data                                   ; директива определения данных
        _a DD 200               ; расположение у памяти переменной _a = 200
        _b DD 11                ; расположение у памяти переменной _b  = 11
        _c DD 45                ; расположение у памяти переменной _c  = 45
        _d DD 22                ; расположение у памяти переменной _d  = 22
_res_abcdRem DD         ; резервирование в памяти 32-разрядной ячейки
_res_abcdUnit1 DD ?
_res_abcdUnit2 DD ?
.code                                   ; директива начала кода программы
_start:                                 ; метка начала программы с именем _start
   _abcd proc                           ; имя процедуры _abcd: (ab + c/d)
        call _ab                        ; вызов процедуры _ab
        mov esi, edx        ; сохранение старшей части умножения
        mov edi, eax        ; сохранение младшей части умножения
        call _cd                        ; вызов процедуры _cd
        mov _res_abcdRem, edx           ; сохранение остатка
        add edi,eax                     ; получение целой младшей части
        mov _res_abcdUnit1,edi      ; сохранение целой младшей части
        adc esi,0                       ; получение целой старшей части
        mov _res_abcdUnit2,esi          ; сохранение целой старшей части
        ret             ; возврат управления  ОС (возвращение из процедуры)
          _abcd endp                    ; окончание процедуры с именем _abcd
        end _start                      ; завершение программы с именем _start
```

Лістинг 8.14. Текст додаткової процедури _ab:

```
        .686                    ; директива определения типа микропроцессора
        .model flat                     ; задание линейной модели памяти
        option casemap: none    ; отличие строчных и прописных букв
        public _ab                      ; переменные, которые передаются в другие мо-
дулі
        extern _a: DWORD, _b: DWORD     ; переменные, которые принимаются
  .code                                 ; директива начала кода программы
        _ab proc                        ; имя процедуры _ab: (a x b)
        mov eax, _a
        mul _b
        ret         ; возврат управления  ОС (повернення з процедури)
        _ab endp                        ; окончание процедуры
        end
```

Лістинг 8.15. Текст додаткової процедури _cd:

```
        .686                    ; директива определения типа микропроцессора
        .model flat             ; задание линейной модели памяти
        option casemap: none    ; отличие строчных и прописных букв
        extern _c: DWORD, _d: DWORD     ; переменные, которые принимаются
```

```
    public _cd                  ; переменные, которые передаются в другие модули
.code                       ; директива начала кода программы
    _cd proc                        ; имя процедуры _cd: (c/d)
    mov eax, _c
    div _d
    ret     ; возврат управления  ОС (возвращение из процедуры)
    _cd endp                                ; окончание процедуры
    end
```

Для простоти в програмі стек не використовується, а результати виконання додаткових процедур передаються через вміст регістрів. Стан програми прикладу 8.6 після її виконання наведено на рис. 8.6.

Кінцевий результат виконання програми (089A,01) знаходиться у комірках пам'яті.

Приклад 8.7. Написати на асемблері програму обчислення виразу

$$(a + b)c - d/c,$$

де a – номер студента в списку; b – номер навчальної групи; c – pp – дві останні цифри свого року народження; e – $pppp$ – чотири цифри свого року народження; d – $ддмм$: $дд$ – день свого народження; $мм$ – місяць свого народження.

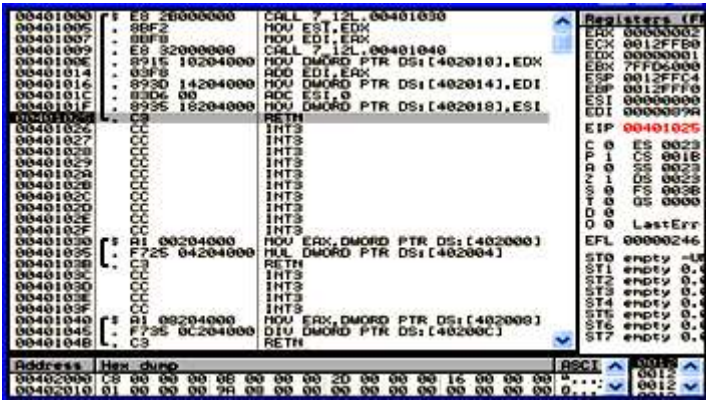


Рис. 8.6. Стан програми прикладу 8.6 після її виконання

Результат обчислення виразу зберегти в пам'яті. Навести значення та порядок розміщення даних у пам'яті.

Спочатку отримаємо результат обчислення виразу (5 + 16)81 – 2209/81 ***без використання*** зовнішніх процедур (лістинг 8.16).

Лістинг 8.16. Програма обчислення прикладу 8.7 з використанням змінних розміром у байт:

```
 title CopyRight by Rysovaniy A. N.
.686                    ; директива определения типа микропроцессора
```

```
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
  .data                                          ; директива определения данных
    _a db 5  ; збереження у 8-розрядній комірці з ім'ям _a числа 5
    _b db 16                    ; _b = 10h
    _c db 81                    ; _c = 51h
    _d dw 2209                  ; _d = 8A1
    rez1 dw 0       ; запис у 16-розрядну комірку пам'яті значення 0
    rez2 dw 0
  .code                         ; директива начала кода программы
_start:                         ; метка начала программы с именем _start
    mov al,_a                   ; al = 05h
    add al,_b                   ; al = 15h
    mul _c                      ; ax = 15h × 51h = 06A5h
    push ax                     ; стек = XXXX06A5
    mov ax,_d                   ; ax = 08A1h
    div _c                      ; ah,al = ax/_c = 16 1Bh
    movzx bx,ah                 ; bx = 0016h – остача
    xor cx,cx                   ; cx = 0 (заповнювання нулями)
    sub cx,bx                   ; cx = cx – bx = FFEAh
    mov rez1,cx
    pop cx                      ; cx = 06A5h
    movzx ax,al                 ; al = 001Bh
    sbb cx,ax                   ; cx = 06A5h – 001Bh = 0689h
    mov rez2,cx
      ret                       ; возврат управления  ОС
      end _start                ; завершение программы с именем _start
```

Програму обчислення прикладу 8.7 з використанням даних розміром у слово, а також ***без використання*** зовнішніх процедур наведено у лістингу 8.17.

Лістинг 8.17. Програма обчислення прикладу 8.7 з використанням змінних розміром у слово:

```
      .686          ; директива определения типа микропроцессора
      .model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
.data                                          ; директива определения данных
    _a dw 5  ; збереження у 16-розрядній комірці з ім'ям _a числа 5
    _b dw 16                    ; _b = 10h
    _c dw 81                    ; _c = 51h
    _d dw 2209      ; _d = 8A1
res dw 3 dup (0) ; запис у масив пам'яті з ім'ям res трьох слів з нульовими значеннями
.code
  _start:
      mov ax,_a                 ; ax = 0005h
      add ax,_b                 ; ax = 0015h
      mul _c                    ; dx = 0000h, ax = 06A5h
      push dx                   ; стек := xxxx 0000h
      push ax                   ; стек := 0000 06A5h
      mov ax,_d                 ; ax = 08A1h
      div _c                    ; dx = 0016h, ax = 001Bh
      xor si,si                 ; si = 0
      sub si,dx                 ; si := 0000 – 0016h = FFEAh – дріб
      mov res,si                ; запис остачі
      pop si                    ; si = 06A5h – старша частина результату (a-b)c
      sbb si,ax                 ;
      mov res+2,si ; запис молодшої цілої частини
      pop si
      sbb si,0
```

```
mov res+4,si ; запис цілої старшої частини
ret                    ; возврат управления  ОС
end _start             ; завершение программы с именем _start
```

Результатом виконання програми буде число 0689.FFEA, де 689h – ціла частина, а FFEAh – залишкова частина результату.

Збереження тимчасових результатів частини виразу (a + b)c виконується в стеку, хоча для цього можна використовувати як один із вільних регістрів, так і комірку пам'яті.

Порядок розміщення даних у пам'яті та стан налагоджувача OllyDbg з програмою наведено на рис. 8.7.

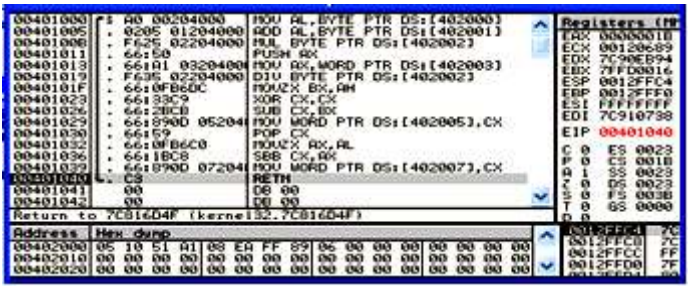


Рис. 8.7. Стан налагоджува OllyDbg з програмою прикладу 8.7

Результат виконання прикладу не змінюється та дорівнює 0000 0689, FFEAh.

Приклад 8.7 також можна роз'єднати ***на декілька зовнішніх процедур.*** Програма складається з трьох частин: головної процедури та двох зовнішніх з іменами _res1 та _res2. В лістингах 8.18–8.20 наведено цю можливість.

Лістинг 8.18. Програма обчислення прикладу 7.6 з використанням зовнішніх процедур:

```
        title CopyRight by Rysovaniy A. N., головна програма
        .686                    ; директива определения типа микропроцессора
        .model flat                     ; задание линейной модели памяти
        .stack 100h
        option casemap: none  ; отличие строчных и прописных букв
          extern _res1: proc            ; вказання на зовнішню процедуру
          extern _res2: proc            ; вказання на зовнішню процедуру
          public  _a, _b, _c, _d  ; доступ для інших модулів
.data                           ; директива определения данных
      _a WORD 5   ; збереження у 16-розрядній комірці з ім'ям _a числа 5
      _b WORD 16
      _c WORD 81
      _d WORD 2209
.code                                   ; директива начала кода программы
_start:                                 ; початок програми з ім'ям _start
        _result proc
          clc                           ; скидання прапорця перенесення CF
```

```
        call _res1              ; виклик процедури _res1
        call _res2              ; виклик процедури _res2
        ret                             ; возврат управления  ОС
    _result endp                        ; закінчення процедури
    end _start                          ; завершение программы с именем _start
```

Лістинг 8.19. Текст процедури _res1:

```
    ; (a-b)c
    .686                    ; директива определения типа микропроцессора
    .model flat                         ; задание линейной модели памяти
    option casemap: none   ; отличие строчных и прописных букв
    public _res1        ; оголошення змінної _res1 як внутрішньої
    extern _a: WORD                     ; оголошення змінної _a як зовнішньої
    extern _b: WORD                     ; оголошення змінної _b як зовнішньої
    extern _c: WORD                     ; оголошення змінної _c як зовнішньої
.code                                   ; директива начала кода программы
_res1 proc                              ; заголовок процедури
      mov ax,_a             ; занесення _a до ax
      add ax,_b                         ; обчислення _a + _b
      mul _c                            ; dx,ax = ax × _c
      mov si,ax             ; запам'ятовування молодшої частини
      mov di,dx             ; запам'ятовування старшої частини
      ret                       ; повернення до початку процедури
_res1 endp                  ; закінчення процедури з іменем _res1
  end                           ; закінчення програми
```

Лістинг 8.20. Текст процедури _res2:

```
    ; -d/c
    .686                ; директива определения типа микропроцессора
    .model flat                     ; задание линейной модели памяти
        option casemap: none    ; отличие строчных и прописных букв
        public _res2            ; оголошення змінної _res2 як внутрішньої
        extern _d: WORD  ; оголошення змінної _d як зовнішньої
        extern _c: WORD  ; оголошення змінної _c як зовнішньої
.data                                   ; директива определения данных
    _res dw 3 dup (0) ;
.code                                   ; директива начала кода программы
  _res2 proc                    ; заголовок процедури
    mov ax,_d                   ;  занесення  в ax змінної _d
    div _c                      ; ah,al := ax/_c
    xor bx,bx                   ; заповнювання нулями регістра
    sub bx,dx                   ; обчислення остачі
    mov _res,bx  ; збереження остачі
    sbb si,ax       ; обчислення молодших частин цілих чисел
                        ; з урахуванням можливої позики cf при отриманні остачі
    mov _res+2,si               ; збереження молодших частин цілих чисел
    sbb di,0        ; віднімання із старшої частини результату
                        ; можливої позики від попередньої операції
    mov _res+4,di                   ; збереження старших частин цілих чисел
    ret                     ; повернення до початку процедури
  _res2 endp                            ; закінчення процедури з іменем _res1
    end                         ; закінчення програми
```

Для кожної процедури спочатку отримують об'єктні файли з розширенням .obj. Ці файли можна отримати двома шляхами: послідовно, або одночасно. При послідовному отриманні у файловому менеджері, наприклад у менеджері FAR, виконують команди

ml.exe /c /coff 03P.asm,

де 03P – ім’я одної програми.

Для інших файлів змінюються лише імена.

При одночасному отриманні виконується команда

ml.exe /c /coff 03P.asm 03_1P.asm 03_2P.asm

Після отримання об’єктних файлів надається одна команда для отримання файлу з розширенням .exe:

link.exe /SUBSYSTEM:CONSOLE 03P.obj 03_1P.obj 03_2P.obj

На наступному етапі викликається налагоджувач та перевіряється правильність функціонування програми.

Стан програми з декількома зовнішніми процедурами та порядок розміщення даних у пам’яті наведено на рис. 8.8.

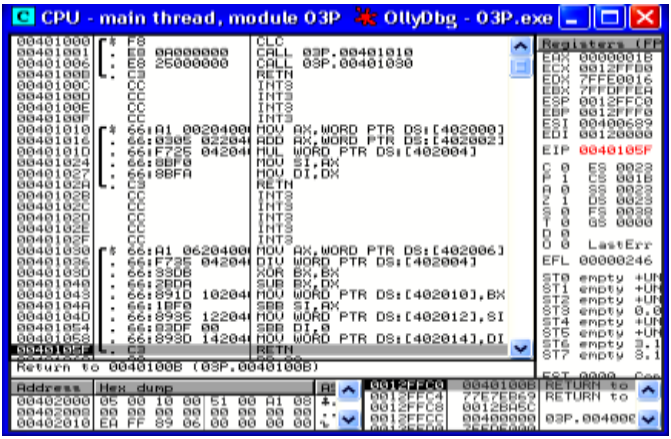


Рис. 8.8. Стан програми з декількома зовнішніми процедурами та порядок розміщення даних у пам’яті

Процес об’єднання декількох частин програми можна здійснити ще на 1-й стадії трансляції. Це досягається за допомогою директиви INCLUDE. Наприклад, один файл містить код програми, а константи, дані (визначення змінних) і прототипи зовнішніх процедур поміщаються в окремі файли. Часто такі файли записують із розширенням .INC.

Саме таке розбиття демонструється в лістингах 8.21–8.23).

Лістинг 8.21:

```
; файл CONS.INC
CONS1  EQU 1000
CONS2  EQU 2000
CONS3  EQU 3000
CONS4  EQU 4000
CONS5  EQU 5000
CONS6  EQU 6000
```

```
CONS7  EQU 7000
CONS8  EQU 8000
CONS9  EQU 9000
CONS10 EQU 10000
CONS11 EQU 11000
CONS12 EQU 12000
```

Лістинг 8.22:

```
; файл DAT.INC
DAT1  DWORD 0
DAT2  DWORD 0
DAT3  DWORD 0
DAT4  DWORD 0
DAT5  DWORD 0
DAT6  DWORD 0
DAT7  DWORD 0
DAT8  DWORD 0
DAT9  DWORD 0
DAT10 DWORD 0
DAT11 DWORD 0
DAT12 DWORD 0
```

Лістинг 8.23:

```
; файл PROG1.ASM
.686p                          ; директива определения типа микропроцессора
.model flat, stdcall           ; плоска модель
include cons.inc               ; підключити файл констант
_data segment dword public use32 ‘data’      ; сегмент даних
include dat.inc                ; підключити файл даних
_data ends
_text segment dword public use32 ‘code’      ; сегмент коду
_start:
    mov eax,cons1
    shl eax,1                  ; множення на 2
    mov dat1,eax
      mov eax,cons2
      shl eax,2                ; множення на 4
      mov dat2,eax
    mov eax, cons3
    add eax, 1000              ; додавання 1000
    mov dat3, eax
      mov eax, cons4
      add eax, 2000   ; додавання 2000
      mov dat4, eax
    mov eax, cons5
    sub eax, 3000              ; віднімання 3000
    mov dat5, eax
      mov eax, cons6
      sub eax, 4000   ; віднімання 4000
      mov dat6, eax
    mov eax, cons7
    mov edx, 3
    imul edx                   ; множення на 3
    mov dat7, eax
      mov eax, cons8
      mov edx, 7               ; множення на 7
      imul edx
```

```
    mov dat8, eax
mov eax, cons9
mov ebx, 3                    ; ділення на 3
mov edx, 0
idiv ebx
mov dat9, eax
    mov eax, cons10
    mov ebx, 7                ; ділення на 7
    mov edx, 0
    idiv ebx
    mov dat10, eax
  mov eax, cons11
  shr eax, 1                  ; ділення на 2
  mov dat11, eax
mov eax, cons12
shr eax, 2                    ; ділення на 4
mov dat12, eax
    ret                       ; возврат управления  ОС
_text ends
end _start                    ; завершение программы с именем _start
```

Для запуску програми необхідно створити три файли, наприклад, з іменами CONS.INC, DAT.INC, PROG1.ASM. З файлу PROG1.ASM створити exe-файл із відповідним іменем, а потім здійснити налагодження програми.

8.13. Лабораторна робота «Передача параметрів через таблицю адрес»

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм, написаних мовою асемблера для програмування задач організації масивів для МП платформи x86.

Вхідний контроль знань

Написати та налагодити програму згідно з отриманим варіантом з обов'язковим урахуванням необхідних ознак виконання операції. Результат записати у пам'ять.

1. Виконати операцію логічного порівняння (команда TEST) числа та місяця свого народження. Якщо нульовий розряд результату операції дорівнює нулю, то додати ці числа.

2. Виконати операцію логічного перемноження (команда AND) місяця та року свого народження. Якщо другий розряд результату операції дорівнює одиниці, то додати ці числа.

3. Виконати зсув року свого народження праворуч на 3 розряди. Якщо перший розряд результату операції дорівнює одиниці, то виконати зсув праворуч на один розряд.

4. Скласти день, місяць та рік свого народження. Якщо третій розряд результату операції дорівнює нулю, то з року народження відняти місяць та день.

5. Перемножити два числа. Якщо нульовий розряд результату операції дорівнює нулю, то додати ці числа.

6. Поділити рік свого народження на місяць народження. Якщо перший розряд результату операції дорівнює нулю, то додати ці числа.

7. Виконати зсув дня свого дня народження ліворуч на 3 розряди. Якщо четвертий розряд результату операції дорівнює одиниці, то виконати зсув праворуч на 1 розряд.

8. Виконати операцію логічного додавання за модулем (команда XOR) місяця та року свого народження. Якщо нульовий розряд результату операції дорівнює нулю, то виконати зсув праворуч на 2 розряди.

9. Виконати зсув року свого народження ліворуч на 4 розряди. Якщо третій розряд результату операції дорівнює нулю, то додати числа дня та місяця свого народження.

10. Виконати операцію логічного додавання за модулем (команда XOR) місяця та дати свого народження. Якщо нульовий розряд результату операції дорівнює нулю, то додати ці числа.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати варіант завдання та написати на асемблері програму обчислення одного з виразів.

1. Задані натуральні числа $a, a_2...a_n$. Вказати ті з них, у яких залишок від ділення на M буде дорівнювати L ($0 <= L <= M - 1$)

2. Задано масив A з $N = 55$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L \geq A_i \geq M$, де $L = 9$ та $M = 30$.

3. Задано масив A з $N = 45$ елементів. Навести алгоритм та програму формування масиву B з останніх 13 елементів масиву A, які дорівнюють нулю.

4. Задано масив A з $N = 55$ елементів. Навести алгоритм та програму формування масиву B з елементів масиву A, які задовольняють умову $A_i \leq E$ при $E = 12$.

5. Задано масив A з $N = 70$ елементів. Навести алгоритм та програму визначення суми і кількості елементів масиву A, які задовольняють умову $A_i \leq E$ при $E = -13$.

6. Задано масиви A і B по $N = 50$ елементів. Навести алгоритм та програму визначення кількості пар елементів, які задовольняють умову $A_i \leq B_i$.

7. Задано масиви A і B по $N = 65$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо $A_i - B_i \leq 0$, то $C_j = A_i$.

8. Задано масив A з $N = 50$ елементів. Навести алгоритм та програму визначення максимального з від'ємних елементів масиву A.

9. Задано масив A з $N = 50$ елементів. Структура масиву A така: $X1$, $Y1$; $X2$, $Y2$; … . Навести алгоритм та програму визначення кількості пар, для яких виконується умова $X_i < Y_i$.

10. Задано масив A з $N = 28$ елементів. Навести алгоритм та програму формування масиву B з елементів масиву A, у яких біти 0, 2 та 5 мають нулі.

11. Задано масив A з $N = 40$ елементів. Навести алгоритм та програму визначення суми елементів масиву A, для яких біти 2 та 10 збігаються.

12. Задано масив A з $N = 80$ елементів. Структура масиву A така: $X1$, $Y1$; $X2$, $Y2$; … . Навести алгоритм та програму визначення кількості пар, для яких виконується умова $X_i > Y_i$.

13. Задано масив A з $N = 20$ елементів. Навести алгоритм та програму визначення мінімального з додатних елементів масиву A.

14. Задано масиви A і B по $N = 30$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо $A_i + B_i > 0$, то $C_j = B_i$.

15. Задано масиви A і B по $N = 20$ елементів. Навести алгоритм та програму визначення кількості пар елементів, які задовольняють умову $A_i > B_i$.

16. Задано масив A з $N = 40$ елементів. Навести алгоритм та програму визначення суми і кількості елементів масиву A, які задовольняють умову $A_i > E$ при $E = -10$.

17. Задано масив A з $N = 22$ елементів. Навести алгоритм та програму формування масиву B з елементів масиву A, які задовольняють умову $A_i > E$ при $E = 5$.

18. Задано масив A з $N = 30$ елементів. Навести алгоритм та програму формування масиву B з перших 8 додатних елементів масиву A.

19. Задано масив A з $N = 20$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L < A_i \le M$, де $L = 2$ та $M = 10$.

20. Задано масиви A і B по $N = 11$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо $A_i > B_i$, то $C_i = A_i + B_i$; інакше – $C_i = A_i - B_i$.

21. Задано масив A з $N = 50$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову$L > A_i > M$, де $L = 8$ та $M = 28$.

22. Задано масиви A і B по $N = 10$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо $Ai < Bi$, то $C_i = B_i - A_i$; інакше – $C_i = A_i + B_i$.

Зміст звіту

1. Постановка задачі для отриманого варіанта.

2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.

3. Лістинг програми з виведенням даних на екран монітора з використанням API-функцій та з детальним коментарем і описом роботи.

4. Print screen екрана 32-розрядного налагоджувача з виконаною програмою.

5. Короткий опис виконання програми.

6. Висновки за результатами роботи.

Приклад 8.8. Задан массив A из N элементов. Зформувати массив B з індексів элементів масива A, де A_i меньше E = 12.

Програма складається з 2-х файлів (лістинги 8.24 та 8.25).

Лістинг 8.24.

```
        ;title Білоусов О. КІТ-23а
        .686                    ;визначення типу мікропроцесора
        .model flat,stdcall     ;задание линейной модели памяти
        option casemap:none     ;відмінність рядкових та прописних літер
        include \masm32\include\windows.inc     ;файли структур, констант
        include \masm32\include\kernel32.inc    ;системні функції застосувань
        include \masm32\include\user32.inc      ;файли інтерфейсу
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
extern func:proc
.data                   ;відкриття сегменту даних
   mas1 dd 12, 32, 0, 34, 8, 100, 1, 20, 11, 76    ;масив A
   len equ ($-mas1) / type mas1             ;довжина масиву A
   mas2 dd len dup(0)                       ;масив B
   ta dd mas1, len, mas2                    ;таблиця адресів
   _title db "Результаты решения программы",0
   info db "Задан массив A из N элементов.",0Ah,0Dh,
   "Сформировать массив B из индексов элементов массива A, ", 0Ah, 0Dh,
   "де Ai меньше E = 12.",0Ah,0Dh,0Dh
   buf dd len dup (?),0       ; буфер виведення повідомлення
   ifmt db "Ответ: mas2 = %d %d %d %d %d %d %d %d %d %d",0
.code                   ;відкриття сегменту коду
start:
        push offset  ta
  call func             ;виклик функції
invoke wsprintf,ADDR buf,ADDR ifmt, mas2[0],mas2[4],mas2[8],mas2[12], mas2[16],
mas2[20], mas2[24], mas2[28], mas2[32], mas2[36] ;
        invoke MessageBox, 0, addr info, addr _title, MB_ICONERROR
        invoke ExitProcess, 0   ;повернення управління ОС та визволення ресурсів
        end start
```

Лістинг 8.25.

```
        .686                ;визначення типу мікропроцесора
        .model flat           ;задание линейной модели памяти
        option casemap: none    ;відмінність рядкових та прописних літер
        public _func
.code               ;відкриття сегменту коду
        _func proc
        push ebp
        mov ebp, esp
        mov ebx, [ebp+8]
        mov ecx, [ebx+4]        ;лічильник
```

```
    mov esi, [ebx+0]       ;масив A
    mov edi, [ebx+8]       ;масив B
begin:
      xor eax, eax
      mov eax, [esi]     ;в реєстр eax заносимо і-й єлемент масиву A
      cmp eax, 13         ;порівнюємо і-й єлемент масиву A з 13
      jc m1              ;якщо і-й єлемент менше то переходимо на мітку m1
      jmp m2              ;якщо більше переходимо на m2
    m1:
      mov[edi], eax      ;в і-й єлемент масиву B заносимо x реєстру eax
      add edi, 4         ; переходимо на наступний єлемент масиву B
    m2:
      add esi, 4         ;переходимо на наступний єлемент масиву A
      loop begin          ;якщо ecx > 0 то переходимо на мітку begin
    pop ebp
     ret
      _func endp            ;кінець функції
     End
```

8.14. Лабораторна робота «API-подібні процедури»

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм, написаних мовою асемблера для програмування API-подібних функцій для МП платформи x86.

Вхідний контроль знань

Написати та налагодити програму згідно з отриманим варіантом та оформити її у вигляді процедури з параметрами. Результат записати у пам'ять.

1. Виконати операцію логічного додавання дня та року свого народження.
2. Виконати операцію логічного порівняння місяця та року свого народження.
3. Виконати зсув дня свого народження ліворуч на 2 розряди.
4. Перемножити день, місяць та рік свого народження.
5. Поділити рік свого народження на місяць народження.
6. Виконати зсув дня свого народження ліворуч на 4 розряди.
7. Скласти день, місяць та рік свого народження.
8. Виконати зсув року свого народження праворуч на 2 розряди.
9. Виконати операцію логічного перемноження місяця та року свого народження.
10. Виконати операцію логічного додавання за модулем місяця та року свого народження.

Постановка задачі

Згідно з номером студента в групі вибрати варіант завдання та написати на асемблері програму обчислення одного з виразів:

1. $2d/c - cd$;	8. $2ab - 8c/b$;	15. $d/3c - 15ac$;	22. $2b - e/22b$;
2. $2a - e/2c$;	9. $8d - 9d/c$;	16. $2d/3c - 16cd$;	23. $ab/3a - 23b$;
3. $b/a + c/a$;	10. $3e/b + 10c$;	17. $2a/7 - d/17e$;	24. $d/4a - 24d/c$;
4. $d/b - d/4c$;	11. $2d/3b - 11c$;	18. $2ab - 18c/d$;	25. $e/8b + 25ac$;
5. $e/3c + ac$;	12. $3b - 12c/d$.	19. $8d/b - 19d/c$;	26. $4d/4a - 26cd$;
6. $2d/c - cd$;	13. $cb/3a - 13a$;	20. $3d/b + 20e$;	27. $8a/7 - d/27b$;
7. $2a/7 - c/7e$;	14. $e/4b - d/14c$;	21. $2d/a - 21bd$;	28. $4ac - 28c/b$,

де a – номер студента за списком в навчальній групі; b – номер навчальної групи; c – pp – дві останні цифри свого року народження; e – $pppp$ – чотири цифри свого року народження; d – *ддмм*: *дд* – день свого народження; *мм* – місяць свого народження.

Результат обчислення виразу зберегти в пам'яті. Навести значення та порядок розміщення даних у пам'яті.

Зміст звіту

1. Постановка задачі для отриманого варіанта завдання.
2. Блок-схема алгоритму виконання прикладу з детальним коментарем.
3. Лістинг програми з детальним коментарем до кожної команди.
4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання.
5. Короткий опис виконання програми.
6. Висновки за результатами роботи.

Приклад 8.9. Написати на асемблері програму обчислення виразу

$$a\ b + c,$$ де $a = 4$, $b = 5$, $c = 6$,

оформити її у вигляді API-подібної функції та зберегти результат виконання виразу у пам'яті.

Лістинг 8.26. Програма обчислення прикладу 8.8:

```
        title CopyRight by Rysovaniy A. N.                    rysov@rambler.ru
        .686                ; директива определения типа микропроцессора
        .model  flat,stdcall   ; задание линейной модели памяти и соглашения ОС
        option casemap:none                ; отличие строчных и прописных букв
        include \masm32\include\windows.inc   ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32
        OperDigs proto :WORD, :WORD, :WORD   ; прототип процедури
.data                                   ; директива определения данных
        res dd 2 dup(0)                 ; резервирование ячеек под результат
        titl db "Вывод через функцию MessageBox",0; название упрощенного окна
        ifmt db "a b + c, де a = 4, b = 5, c = 6:", 0dh,0ah,
        "Результат. Ст. часть = %d, Мл.часть = %d", 0dh,0ah, "Автор:       НТУ ХПИ", 0;
        st1 dd ?,0 ; буфер выведения сообщения
.code                                   ; директива начала программы
```

```
_start:                 ; метка начала программы с именем _start
        invoke OperDigs,4,5,6          ; вызов директивы с параметрами
        invoke wsprintf,ADDR st1,ADDR ifmt,edx,eax;
        invoke MessageBox,0,addr st1,addr titl,MB_ICONINFORMATION
        invoke ExitProcess,0           ;

            OperDigs proc arg1:WORD,arg2:WORD,arg3:WORD
            mov ax,arg1                ; ax := arg1 (число 4)
            mul arg2                   ; dx,ax = ax × arg2
            add ax, arg3               ; ax := ax + arg3 (число 6)
            jc m1                      ; перейти, если есть перенос
            jmp m2          ; безусловный переход на сохранение результата
        m1: inc dx                     ; dx + 1, есть перенос есть
        m2:
            movzx eax,ax
            movzx edx,dx
            ret                        ; возвращение из процедуры
            OperDigs endp ;
        end _start                     ; завершение программы с именем _start
```

Через те, що числа вибрані невеликі, то для їх визначення вибрано слово WORD. Команди розгалуження jc та jmp необхідні для випадку, коли використовуватимуть числа, які можуть дати переповнення розрядної сітки.

Параметри процедури OperDigs розміщуються в стеку за порядком справо наліво (спочатку – число 6 := arg3, потім – число 5 := arg2, а на остачу – число 4 := arg1). *Директива invoke замінює дві команди: CALL та PUSH 0, де PUSH 0 – код успішного завершення, який повертається операційній системі.* Процедура OperDigs написата так, що результат виконання передається в регістрах.

Результат виконання програми наведено на рис. 8.9.

Рис. 8.9 Результат виконання програми

8.15 Лабораторна робота «Зовнішні процедури»

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм, написаних мовою асемблера з використанням ***зовнішніх процедур*** для МП платформи x86.

Постановка задачі

Згідно з номером студента в групі вибрати варіант завдання та написати на асемблері програму обчислення одного з виразів:

1. $(a - bc) + e/d$;	9. $c - ad + b/e$;	17. $(c/e + 2a)b$;
2. $(ce + a)b - 2d/a$;	10. $(a - ed)b - d/b$;	18. $(a - bc)/b$;
3. $(a - cb)/b + 3d$;	11. $(a - b)ab - d/e$;	19. $8c - ab - b/19e$;
4. $8d - 4d/a$;	12. $(b - 2d)/(12a - c)$;	20. $(a - cd) - a/b$;
5. $5ac - 5b/c$;	13. $ae + b - b/c$;	21. $(a - b)ab - d/e$;
6. $(a - b/c) + 2ed$;	14. $8ad - b - d/a$;	22. $(a - 2d)/(a - c)$;
7. $2(c/e + a)b$;	15. $a - 5ac - b/c$;	23. $ab + c - b/c$;
8. $(a - c/b)/8b$;	16. $2(a - b/c) + ed$;	24. $8ab - d/a$.

Вимоги до програм:

1. Передбачити ситуацію, при якій числа будуть змінюватися від мінімального до максимального значень (урахувати можливі переповнення розрядної сітки та позики);

2. Для перших десяти за списком студентів змінні повинні мати розмір ***байта***, для других десяти за списком студентів – розмір ***слова***, а для третіх десяти за списком студентів – розмір ***подвійного слова***;

3. Навести програму з використанням ***зовнішніх*** процедур.

4. Вказати діапазон зміни змінних.

Зміст звіту

1. Постановка задачі для отриманого варіанта завдаання.

2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.

3. Лістинг програм: головної та зовнішньої процедури з детальним коментарем та описом роботи.

4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання виразу.

5. Короткий опис виконання програми.

6. Висновки за результатами роботи.

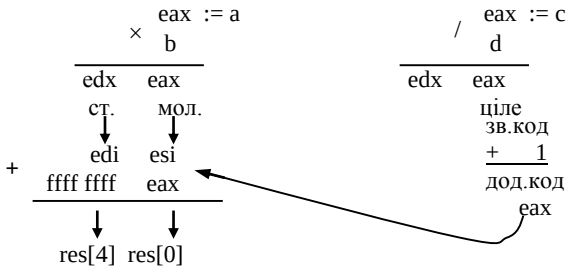
Приклад 8.10. Нехай потрібно обчислити вираз $ab - c/d$, де a, b, c, d – цілі числа розміром у подвійне слово.

Обчислення виконаємо за допомогою однієї процедури, яка розташована у файлі 7_9_1.asm, що обчислює $ab - c/d$. В головній програмі виконується тільки виклик зовнішньої процедури.

Через те, що в прикладі необхідно виконати віднімання операцій, а значення результату віднімання не відомо (ab може бути меншим, ніж c/d), то спочатку результат c/d перетворюється в додатковий код, а вже

потім виконується віднімання. Остаточний результат також буде в додатковому коді. Остачу ділення можна не запам'ятовувати. Згідно зі схемою складемо головну та додаткову програми.

Складемо схему виконання прикладу.



Головну програму з іменем 7_9_1.asm наведено у лістингу 8.27.

Лістинг 8.27. Головна програма обчислення виразу *ab – c/d*.

```
        title CopyRight by Rysovaniy A. N.
        .686                    ; директива определения типа микропроцессора
        .model flat             ; задание линейной модели памяти
Extern _abcd:proc   ; указання на зовнішню процедуру
Public _a, _b,_c, _d, _res      ; доступ для інших модулів
.data                   ; директива определения данных (ab  –  c/d)
        _a   dd  1              ; запис у 32-розрядну комірку з ім'ям _a
        _b   dd  2              ; запис у 32-розрядну комірку з ім'ям _b
        _c   dd  7              ; запис у 32-розрядну комірку з ім'ям _c
        _d   dd  2              ; запис у 32-розрядну комірку з ім'ям _d
        _res  dd 3 dup(0)       ; комірки для збереження результату
.code                           ; директива начала программы
_start:                 ; метка начала программы с именем _start
          call _abcd            ; виклик процедури
        ret                     ; возврат управления  ОС
       end _start       ; директива завершение программы с именем _start
```

Як команда асемблера програма з лістингу 8.25 складається з однієї команди call _abcd. Всі інші команди наведено у іншому файлі з іменем 7_9_2.asm, який містить процедуру обчислення виразу *ab – c/d* (лістинг 8.28).

Лістинг 8.28. Текст процедури з іменем _abcd:

```
        .686                    ; директива определения типа микропроцессора
       .model  flat             ; задание линейной модели памяти
        public _abcd
        extern _a:dword, _b:dword, _c:dword,_d:dword,_res:dword
   .code                        ; директива начала программы
     _abcd  proc                ;  ab - c/d
        mov  eax,_a     ; пересилання з комірки пам'яті з ім'ям _a в eax
        mul _b                  ; edx, eax := eax × _b
        mov esi,eax ; збереження молодшої частини результату множення
        mov edi,edx             ; збереження старшої частини
```

```
        mov eax,_c              ; підготовка до ділення
        div _d                  ; eax, edx := eax/_d ; в edx – остача
        not eax                 ; інверсія (зворотній код цілого числа)
        add eax,1               ; додатковий код цілого числа
        add eax,esi             ; ціла молодша частина
mov _res[0],eax ; запам'ятовування молодшої цілої частини у пам'яті
        adc edi,0ffffffffh          ; ціла старша частина
mov _res[4],edi  ; запам'ятовування старшої цілої частини у пам'яті
        ret
        _abcd  endp
        end                     ; директива окончания программы
```

Для виконання програми зручно створити командний bat-файл зі змістом:

```
ml.exe  /c  /coff   7_9_1.asm  7_9_2.asm
link.exe /subsystem:console 7_9_1.obj  7_9_2.obj
del *.obj
pause
start 7_9_1.exe
```

Після того, як такий командний bat-файл створено, необхідно його виконати (двічі натиснути на нього). Процес отримання exe-файлу наведено на рис. 8.9. Результат виконання програми в додатковому коді можна побачити на рис. 8.10 у вікнах налагоджувача.

Рис. 8.9. Процес отримання exe-файлу

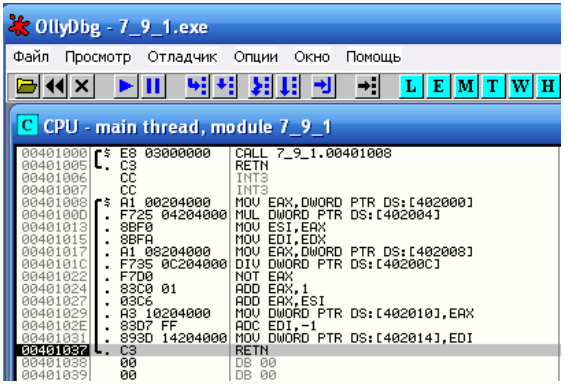


Рис. 8.10. Процес виконання програми та одержання результату

Контрольні запитання

1. Які переваги та недоліки використання процедур?
2. Яка директива використовується для оголошення однієї або декількох локальних змінних усередині процедури?
3. Які переваги дає використання директиви invoke?
4. Який оператор може використовуватись у директиві invoke для передання їй вказівника на змінну?
5. Який оператор, вказаний відразу після директиви PROC, дозволяє перерахувати імена всіх регістрів, значення яких змінюється в процедурі?
6. Як організовується та виконується процедура?
7. Які процедури є Windows API-подібними?
8. Як виконується непрямий виклик процедур?
9. Як оголошуються глобальні змінні в процедурах?
10. Які директиви використовуються для роботи із загальними змінними?
11. Який модуль при об'єднанні декількох модулів записується першим?
12. Які файли мають розширення .INC?
13. За допомогою якої директиви виконується підключення додаткового файлу?

9. ОРГАНІЗАЦІЯ ВВЕДЕННЯ-ВИВЕДЕННЯ В WIN32

9.1. Загальні відомості

Операційна система Windows на відміну від MS-DOS функціонує в 32-розрядних застосуваннях і запускається в захищеному режимі (Protected mode), тоді як програми для MS-DOS працюють у режимі реальних адрес (Real-address mode). Тому використовуються абсолютно різні бібліотеки. 32-розрядні застосування працюють із тими ж бібліотеками функцій, які використовують і графічні застосування Windows, а програми для MS-DOS використовують BIOS і переривання MS-DOS, які існують із дня появи перших персональних комп'ютерів IBM-PC.

Системні програми – це програми загального користування, що виконуються разом із прикладними й службовими програмами для керування ресурсами комп'ютера – центральним мікропроцесором, пам'яттю, процесом введення-виведення. Звернення до клавіатури і монітора (***консольне введення-виведення***) в Windows здійснюється за допомогою виклику функцій, а не переривань, характерних для DOS. Тут немає передання параметрів у регістрах при зверненні до сервісних функцій і відповідно немає і безлічі результуючих значень, повернених у регістрах загального призначення і регістрі прапорців. У Win32 не можна безпосередньо працювати з апаратним рівнем, як було прийнято для програм DOS.

Операційні системи MS-DOS і Windows підтримують дві абсолютно різні ідеології програмування. Програма DOS після свого запуску має бути постійно активною. Якщо їй щось потрібно (наприклад, одержати чергову порцію даних із пристрою введення-виведення), то вона сама має виконувати відповідні запити до операційної системи. При цьому програма DOS працює за певним алгоритмом і вона завжди знає, що і коли їй слід робити.

У Windows все навпаки. Програма є пасивною. Після запуску вона чекає, коли їй приділить увагу операційна система. Операційна система робить це посиланням спеціально оформлених груп даних, що називаються ***повідомленнями.*** Повідомлення можуть бути різного типу, вони функціонують у системі достатньо хаотично, і застосування не знає, якого типу повідомлення прийде наступним. Звідси випливає, що логіка побудови Windows-застосування повинна забезпечувати коректну і передбачену роботу під час оброблення повідомлень будь-якого типу.

Для забезпечення нормального функціонування своєї програми програміст повинен уміти ефективно використовувати функції інтерфейсу прикладного програмування (API, Application Program Interface) операційної системи.

Windows підтримує два типи застосування :

– ***віконне застосування,*** яке будується на базі спеціального набору функцій (API), що становлять графічний інтерфейс користувача (GUI, Graphic User Interface). Віконне застосування є програмою, яка проводить виведення на ек-

ран у графічному вигляді. Першим результатом роботи віконного застосування є відображення на екрані спеціального об'єкта – вікна. Після відображення вікна на екрані вся робота застосування спрямовується на те, щоб підтримувати його в актуальному стані;

– ***невіконне*** або ***консольне застосування****, яке* є програмою, що працює в текстовому режимі. Робота консольного застосування забезпечується спеціальними функціями Windows.

У переважній більшості випадків програма повинна взаємодіяти з користувачем, одержуючи від нього дані за допомогою клавіатури і видаючи результати своєї роботи на екран. Для здійснення введення-виведення програма використовує можливості самого комп'ютера (переривання BIOS) і операційної системи, в середовищі якої ця програма працює.

Таким чином, Windows підтримує роботу двох типів застосувань – віконних, що повною мірою використовують всі переваги графічного інтерфейсу, і консольних, що працюють виключно в текстовому режимі.

Консоль складається з одного вхідного і декількох екранних буферів. **Вхідний буфер** є чергою, кожен запис якої містить інформацію щодо окремої вхідної події консолі. **Екранний буфер –** двовимірний масив, що містить символи, які виводяться у вікно консолі, і дані про їх колір.

Черга вхідного буфера містить інформацію про такі події:

– натиснення і відпускання клавіш;

– маніпуляцію мишею – рух, натиснення- відпускання кнопок;

– зміну розміру активного екранного буфера, стан прокручування.

Функції Win API які працюють в текстовому режимі, забезпечують два рівні доступу до консолі – **високий і низький**. Консольні функції введення **високого рівня** дозволяють застосуванню витягнути дані, одержані при введенні з клавіатури і збережені у вхідному буфері консолі. Консольні функції виведення високого рівня дозволяють застосуванню записати дані в пристрої стандартного виведення або помилки Для того щоб відобразити цей текст в екранному буфері консолі. Функції високого рівня також підтримують перепризначення стандартних дескрипторів введення-виведення і керування режимами роботи консолі. Консольні функції низького **рівня** дозволяють застосуванням одержати детальну інформацію про введення з клавіатури, про події натиснення- відпускання кнопок миші і про маніпуляції користувача з вікном консолі, а також здійснити більший контроль над виведенням даних на екран.

Функції API Win32 високого рівня забезпечують простоту процесу введення-виведення шляхом використання стандартних дескрипторів введення-виведення. Функції низького рівня потребують обліку більшої кількості деталей і обсягу коду, але це компенсується більшою гнучкістю програми.

Високорівневе і низькорівневе консольне введення-виведення не є взаємовиключними, і застосування може використовувати будь-яку комбінацію цих функцій.

При запуску застосування для Windows воно створює або консольне вікно (консоль), або графічне вікно.

Для того щоб одержати консольне застосування, що працює в захищеному режимі, необхідно при виклику компонувальника використовувати опцію /SUBSYSTEM:CONSOLE.

Читання з консолі, запис у консоль та помилки записуються в стандартні файли. Консоль має один *вхідний буфер* і один або декілька *екранних буферів.*

Вхідний буфер містить послідовність вхідних записів, кожен з яких включає дані про вхідні події (наприклад, про натиснення клавіш, клацання миші або зміну розмірів вікна).

Екранний буфер містить двовимірний масив символів і даних про колір, які необхідні для відображення символів на екрані.

9.2. Набір символів і API-функції Windows

При виклику функцій Win32 API можна використовувати один із двох *наборів символів.* Це набір 8-розрядних символів ASCII/ANSI і набір 16-розрядних символів Unicode (доступні в Windows NT, 2000 і XP). Функції Windows API, що працюють із текстом, зазвичай мають дві версії: одна із них призначена для роботи з 8-розрядними символами ASCII/ANSI (імена закінчуються символом A), а інша – для роботи з розширеним набором символів, включаючи Unicode (імена закінчуються символом W). Наприклад, для відображення інформації на консолі використовуються такі функції:

WriteConsoleA
WriteConsoleW

Імена функцій, що закінчуються на W, не підтримуються в Windows 95 або 98. Навпаки, в Windows NT, 2000 і XP набір символів Unicode є вбудованим для цих систем. Тому коли викликається така функція, як WriteConsoleA, операційна система спочатку перетворює символи з набору ANSI в Unicode, а потім викликає функцію WriteConsoleW.

9.3. Типи даних Windows

Функції Win32 API базуються на синтаксисі, що використовується для мов C або C++. У оголошенні функцій Win32 API типи всіх параметрів засновані на стандартних типах C або на зумовлених типах MS-Windows, як наведено у табл. 9.1. Необхідно відрізняти дані типу «значення» від даних типу «вказівник». Типи, імена яких починаються з букв «LP», є *далекими вказівниками* об'єктів.

Таблиця 9.1

Типи MS-Windows	Типи MASM	Опис
BOOL	BYTE	Булеве значення
BSTR	PTR BYTE	32-розрядний вказівник символів

BYTE	BYTE	8-розрядне ціле число без знака
COLORREF	DWORD	32-розрядне число, що використовується як характеристика кольору
DWORD	DWORD	32-розрядне ціле число без знака або адреса сегмента і пов'язаний із ним зсув
HANDLE	DWORD	32-розрядне ціле число без знака
LONG	SDWORD	32-розрядне ціле число зі знаком
LPARAM	DWORD	32-розрядне значення, яке передається як аргумент у віконні процедури або функції зворотного виклику (може бути вказівником)
LPCSTR	PTR BYTE	32-розрядний вказівник константи типу «символьний рядок»
LPSTR	PTR BYTE	32-розрядний вказівник символьного рядка
LPCTSTR	DWORD	32-розрядний вказівник константи типу «символьний рядок», складений із символів Unicode або з набору двобайтових символів
LPTSTR	DWORD	32-розрядний вказівник символьного рядка, складений із символів Unicode або з набору двобайтових символів
LPVOID	DWORD	32-розрядний вказівник невизначеного типу
LRESULT	DWORD	32-розрядне значення, яке повертається віконною процедурою або функцією зворотного виклику
UINT	DWORD	32-розрядне ціле число без знака
WNDPROC	DWORD	32-розрядний вказівник віконної процедури
WORD	WORD	16-розрядне ціле число без знака
WPARAM	DWORD	32-розрядне значення, яке передається як аргумент у віконні процедури або у функції зворотного виклику
LPCRECT	PTR RECT	32-розрядний вказівник структури RECT, що не модифікується

9.4. Дескриптори консолі

Майже всі функції вимагають передання дескриптора як першого аргументу. ***Дескриптор*** *є 32-розрядним цілим числом без знака, що унікально ідентифікує об'єкт*, такий, як растрове зображення (bitmap) або креслярське перо, або вхідні і вихідні пристрої. Дескриптор, handle, описувач – число, що задає номер якого-небудь ресурсу (блок пам'яті, файл, курсор, шрифт, вікно і тому подібне) при роботі з ним. Дескриптор використовується зазвичай при роботі через деякий інтерфейс (API), причому сенс значення дескриптора прихований за цим інтерфейсом.

Після створення вікна функція повертає дескриптор вікна (window handle), який унікально ідентифікує вікно. Прикладна програма використовує цей дескриптор в інших функціях, щоб направити їх дії на це вікно. Дескриптор вікна має тип даних HWND; прикладна програма повинна використовувати цей тип при оголошенні змінної, яка містить у собі дескриптор вікна.

Найчастіше використовуються дескриптори для таких пристроїв:

- STD_INPUT_HANDLE – стандартний вхід;
- STD_OUTPUT_HANDLE – стандартний вихід;
- STD_ERROR_HANDLE – стандартний вихід помилок.

Функція GetStdHandle повертає дескриптор потоку для консолі: вхід, вихід або вихід помилок. Дескриптор необхідний для створення входу-виходу в програмах, що працюють із консоллю. Прототип функції GetStdHandle

```
GetStdHandle  PROTO   nStdHandle:DWORD
```

Він розміщений у файлі windows.inc.

Функція GetStdHandle може набувати значення STD_INPUT_ HANDLE, STD_OUTPUT_HANDLE або STD_ERROR_HANDLE. Ці константи описано в стандартних заголовних файлах і мають значення –10, –11 і –12 відповідно. Функція повертає значення дескриптора в регістр EAX; це значення має бути збережене для подальшого використання. Нижче показаний невеликий приклад:

```
.DATA                                   директива визначення даних
    inputHandle DWORD ?          ; резервування 32-розрядної комірки
.CODE                                    ; директива начала кода программы
invoke GetStdHandle, STD_OUTPUT_HANDLE ; одержання дескриптора
    mov inputHandle, EAX                 ; запис дескриптора в пам'ять
```

Оскільки константа STD_INPUT_HANDLE визначає пристрій, для якого в Windows використовується числове значення –10, то в даному випадку замість ідентифікатора STD_INPUT_HANDLE можна підставити число –10. Проте використання зрозумілого ідентифікатора значно полегшує роботу компілятора і звільняє від запам'ятовування абсолютно неінформативних чисел. До того ж ідентифікатору STD_INPUT_ HANDLE десь має бути привласнене значення –10. Зазвичай це робиться в *заголовних* (що підключаються) файлах, які мають розширення .inc і які повинні бути підключені до бібліотеки або програми за допомогою ключового слова INCLUDE.

9.5. API-функції консолі в Win32

У табл. 9.2 наведено короткий опис всіх функцій для роботи з консоллю Win32. Повні описи API-функцій можна знайти в бібліотеці MSDN (http://msdn.microsoft.com/library/).

Таблиця 9.2

Функція	Опис
AllocConsole	Створює нову консоль
CreateConsole-ScreenBuffer	Створює екранний буфер консолі
FillConsoleOutput Attribute	Установлює атрибути тексту і фону для вказаних символів

FillConsoleOutput-Character	Записує символ в екранний буфер задану кількість разів
FlushConsole-InputBuffer	Виводить весь вміст буфера консолі
FreeConsole	Відключає консоль від процесу, що викликав її
GenerateConsole CtrlEvent	Посилає сигнал у процеси, пов'язані з консоллю, для дозволу додаткового доступу до процесу
GetConsoleCP	Витягує вхідну кодову сторінку, що використовуються консоллю процесу, який викликав її
GetConsoleCursor-Info	Витягує інформацію про розмір і видимість курсора для вказаного екранного буфера консолі
GetConsoleMode	Витягує інформацію про поточний режим введення буфера консолі або про режим виведення екранного буфера консолі
GetConsole-OutputCP	Витягує вихідну кодову сторінку, що використовується консоллю процесу, який викликав її
GetConsole-ScreenBufferInfo	Витягує інформацію про вказаний екранний буфер консолі
GetConsoleTitle	Витягує рядок заголовка вікна для поточного вікна консолі
GetConsole-Window	Витягує дескриптор вікна консолі
GetLargestConsole-WindowSize	Витягує найбільший допустимий розмір вікна консолі
GetNumberOf-ConsoleInputEvents	Витягує кількість непрочитаних вхідних записів у вхідний буфер консолі
GetNumberOf-ConsoleMouse-Buttons	Повертає кількість кнопок миші, що використовуваних з консоллю
GetStdHandle	Повертає дескриптори для стандартного введення, стандартного виведення або стандартного виведення помилок
MessageBox	Створення вікна консолі
HandlerRoutine	Оголошує у застосуванні функцію
PeekConsoleInput	Зчитує дані з вхідного буфера консолі без видалення даних із буфера
ReadConsole	Зчитує дані з вхідного буфера консолі і видаляє дані з буфера
ReadConsoleInput	Зчитує дані з вхідного буфера консолі і видаляє дані з буфера
ReadConsoleOutput	Зчитує символи і атрибути кольору з вказаного прямокутного блока в екранному буфері
ReadConsole-Outpu-Attribute	Копіює вказану кількість послідовних кольорових атрибутів для тексту і фону з екранного буфера консолі
ReadConsole-OutputCharacter	Копіює вказану кількість послідовних символів з екранного буфера консолі

ScrollConsole-ScreenBuffer	Переміщає блок даних в екранний буфер
SetConsoleActive-Screcn-Buffer	Встановлює вказаний екранний буфер як поточний екранний буфер консолі
SetConsoleCP	Встановлює вхідну кодову сторінку, що використовується консоллю для процесу, який викликав її
SetConsoleCtrl Handler	Додає або видаляє певні дескриптори у застосуванні HandlerRoutine із списку оброблювачів функцій для процесу, що викликається
SetConsoleCursor-Info	Встановлює розмір і видимість курсора для екранного буфера консолі
SetConsole-CursorPosition	Встановлює позицію курсора в екранному буфері консолі
SetConsoleMode	Встановлює режим введення вхідного буфера консолі або режим виведення екранного буфера консолі
SetConsole-OutputCP	Встановлює кодову сторінку для виведення символів, пов'язану з процесом, який викликається
SetConsole-ScreenBuffer-Size	Змінює розмір екранного буфера консолі
SetConsole-TextAttribute	Встановлює кольорові атрибути тексту і фону символів, що записуються в екранний буфер консолі
SetConsoleTitle	Встановлює заголовок поточного вікна консолі
SctConsole-WindowInfo	Встановлює поточний розмір і положення вікна для екранного буфера консолі
SetStdHandle	Встановлює дескриптор для стандартного введення, стандартного виведення та стандартного виведення помилок
WriteConsole	Записує символьний рядок в екранний буфер консолі, починаючи з поточного положення курсора
WriteConsolelnput	Записує дані безпосередньо у вихідний буфер консолі
WriteConsoleOutput	Записує символ і кольорові атрибути у вказаний прямокутний блок в екранному буфері консолі
WriteConsoleOutput Attribut	Копіює декілька кольорових атрибутів для тексту і фону в середовища екранного буфера консолі
WriteConsoleOutput-Character	Копіює декілька символів у середовища екранного буфера консолі

9.6. Виведення повідомлень

Для виведення повідомлень розглянемо функцію WriteConsoleA.

Функція (процедура) WriteConsoleA, що виводить символи на екран, має прототип

WriteConsoleA proto :dword,:dword,:dword,:dword,:dword

та п'ять параметрів, але п'ятий завжди дорівнює нулю. Перші чотири параметри такі:

1. stdout – це дескриптор стандартного пристрою виведення (екрана монітора), одержаний процедурою GetStdHandle.

2. ADDR msg – адреса початку повідомлення. Щоб одержати його, використовується спеціальний оператор отримання адреси ADDR.

3. SIZEOF msg – розмір повідомлення, тобто кількість байтів у ньому. Наше повідомлення, поміщене в лапки, складається з 18 байтів (16 байтів займають букви і два байти – символи 0dh, 0ah, які дають команду процедурі WriteConsoleA перевести рядок). **Розмір повідомлення** (кількість байтів від вказаної мітки msg до наступної CWritten) **програма-асемблер обчислює під час компіляції.**

4. ADDR cWritten – адреса ділянки пам'яті, де процедура WriteConsoleA зберігає кількість виведених на екран символів.

Прототип функції WriteConsoleA розміщений у файлі kernel32.lib.

Розглянемо невеликий приклад виведення на консоль тексту (лістинг 9.1).

Лістинг 9.1. Виведення повідомлення на екран монітора:

```
        title виведення повідомлення
        .686                    ; директива определения типа микропроцессора
.model flat, stdcall    ; задание линейной модели памяти  и соглашения ОС Windows
option casemap:none             ; отличие строчных и прописных букв
    SetConsoleOutputCP proto :dword             ;  прототип функції
    ExitProcess proto :dword                    ;  прототип функції
    GetStdHandle proto :dword                   ;  прототип функції
WriteConsoleA proto :dword,:dword,:dword,:dword,:dword  ;  прототип
    Includelib  \masm32\lib\kernel32.lib
.data                           ; директива определения данных
        stdout  dd ?            ; комірка для збереження дескриптора
        msg     db "Дайте сказать!!!",0dh,0ah
        cWritten dd ?           ; адреса для збереження виведених символів
.code                           ; директива начала кода программы
start:                          ; метка начала программы с именем start
invoke SetConsoleOutputCP,1251  ; для відображення кирилиці
invoke GetStdHandle, -11 ; задання дескриптора стандартного пристрою
    mov stdout, eax             ; збереження одержаного дескриптора у пам'яті
invoke WriteConsoleA, stdout, \         ; дескриптор стандартного пристрою виведення
  ADDR msg,\                    ; адреса початку повідомлення
   sizeof msg, \                ; розмір повідомлення
    ADDR cWritten, 0            ; адреса, де зберігається кількість символів
invoke  ExitProcess, 0 ; возвращение управления ОС та вивільнення ресурсів
        end start                                   ; закінчення програми
```

Для прискорення виконання програми рекомендується створити файл з розширенням .bat, де в лапках слід вказувати імена програм, що будуть налагоджуватися, наприклад:

```
ml /c /coff "01L.asm"
link /SUBSYSTEM:CONSOLE "01L.obj"
```

Після утворення .exe-файлу необхідно виконати його, натиснувши кнопку Enter. У файловому менеджері FAR буде відображене таке (рис. 9.1).

Рис. 9.1. Відображення на екрані процесу виконання програми лістингу 9.1

Через те, що програма відноналежить до консольних програм, то при її налагодженні в середовищі MASM32 Editor для отримання exe-файлу необхідно використати шлях **Project/Console Assemble & Link** або bat-файл.

У програмі для відображення літер кирилиці використано функцію SetConsoleOutputCP, яка встановлює потрібну кодову сторінку.

Прототипи використаних функцій наводяться на початку програми. Якщо прототип функції не уміщається на одному рядку, то її опис може продовжуватися на наступному рядку при використанні навскісної межі або коми.

Процедура GetStdHandle одержує *дескриптор стандартного пристрою* – 32-розрядне число, яке потрібно указувати іншим процедурам, що взаємодіють із цим пристроєм. Щоб, наприклад, дізнатися про дескриптор стандартного пристрою виведення, куди буде відправлено фразу «Не могу молчать», параметр має дорівнювати –11. Рядок

```
invoke GetStdHandle, -11
```

замінюється на команду push -0B та виклик процедури call (рис. 9.2).

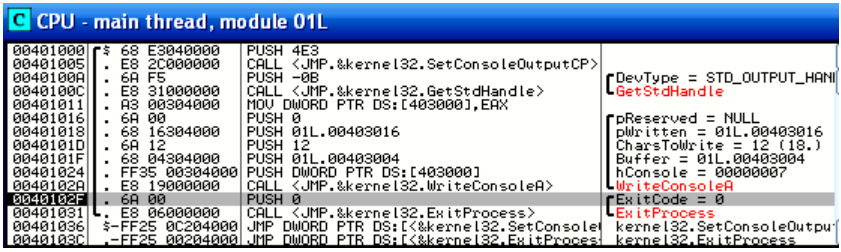


Рис. 9.2. Стан програми лістингу 9.1 після її завершення

Процедура GetStdHandle розміщує результат своєї роботи в регістрі eax. Тому команда mov stdout. eax зберігає одержаний дескриптор у пам'яті.

У програмі з лістингу 9.2 прототипи процедур розміщені у файлах відповідних бібліотек.

Лістинг 9.2. Використання готових прототипів процедур:

```
        title готові прототипи процедур
        .686                    ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option  casemap:none    ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\include\kernel32.inc ; системные функции применений…
        includelib \masm32\lib\kernel32.lib
.data                                       ; директива определения данных
     stdout DWORD ? ; резервування в пам'яті 32-розрядної комірки
                                            ; з ім'ям stdout для збереження дескриптора виведення
        msg db "Дайте сказать!!!",0dh,0ah
        cWritten        dd ?            ; адреса для збереження виведених символів
.code                                   ; директива начала кода программы
 start:                         ; метка начала программы с именем start
invoke SetConsoleOutputCP,1251  ; для відображення кирилиці
invoke  GetStdHandle, STD_OUTPUT_HANDLE   ; одержання дескриптора
        mov stdout, eax    ; збереження одержаного дескриптора у пам'яті
invoke WriteConsoleA, \  ; API-функція виведення на екран
 stdout, \              ; дескриптор стандартного пристрою виведення
  ADDR msg,\                ; адреса початку повідомлення
    sizeof msg, \               ; розмір повідомлення
     ADDR cWritten, 0       ; адреса, де зберігається число символів
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start                       ; закінчення програми
```

Прототипи процедур знаходяться у файл kernel32.inc, що підключається. А файл windows.inc містить всього один рядок програми STD_OUTPUT_HANDLE equ –11, який повідомляє асемблеру, що всі імена STD_OUTPUT_HANDLE, наявні в програмі, потрібно замінити на -11.

Розглянемо програму визначення парного числа та виведення результату на монітор (лістинг 9.3).

Лістинг 9.3. Програма визначення парного числа:

```
        .686                    ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option  casemap:none    ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\include\kernel32.inc ; системные функции применений…
        includelib \masm32\lib\kernel32.lib
.data                                       ; директива определения данных
        FellowDig    BYTE "Парне",13,10
        NFellowDig  BYTE "Непарне",13,10
        stdout DWORD ?    ; для збереження дескриптора виведення
        cWritten  DWORD  ?    ; для адреси символів виведення
.code                                   ; директива начала кода программы
 start:                                 ; метка начала программы з ім'ям
invoke GetStdHandle, STD_OUTPUT_HANDLE   ; запис дескриптора в eax
mov stdout, eax                 ; збереження одержаного дескриптора у пам'яті
        mov ah, 23               ; завантаження числа, яке аналізується
        test ah, 00000001b     ; виділяємо молодший біт
        jz  next                ; перейти, якщо парне
```

```
invoke  WriteConsoleA, stdout, ADDR NFellowDig, sizeof NFellowDig, ADDR cWritten,0
        jmp  exit                                   ; безумовний перехід на мітку exit
next:   invoke  WriteConsoleA, stdout, ADDR FellowDig,sizeof FellowDig, ADDR cWritten,0
exit:   invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start                   ; директива завершение программы с именем start
```

Програма обчислює парність числа, яке записане в регістр ah. Команда test ah, 00000001b виконує логічну операцію «*І*», але не змінює вмісту регістра ah, а лише встановлює прапорці виконання операції.

Всі символи, що відображаються на екрані термінала або виводяться на друк, мають закріплені за ними коди, які називаються кодами ASCII. Кожен код ASCII займає один байт і може набувати значень від 0 до 255. Сукупність всіх 255 кодів разом із зображеннями символів становлять кодову таблицю ASCII.

Перша половина таблиці ASCII – стандартна, вона містить коди цифр, розділового знака, латинських малих і великих літер. На початку таблиці розміщено коди спеціальних піктограм (зображення піки, гральних карт, ноток та ін.). Деякі з цих кодів при операціях виведення за допомогою функцій DOS сприймаються як керівні. Так, код 9 відображається у вигляді жирного кружка, але функціями DOS він інтерпретується як код табуляції. Код 13 подає зображення нотки, а програмами DOS це сприймається як повернення каретки.

Друга половина таблиці ACSII є «національно-залежною», її коди відповідають здебільшого знакам національного алфавіту і символам псевдографіки. Тому програми, що виводять на екран (або на друк) псевдографічні зображення, працюють однаково як за наявності русифікатора, так і без нього.

Транслятор, зустрівшись у тексті програми з символьним рядком, який повинен бути виведений на екран, перетворить зображення символів у цьому рядку на відповідні до них коди ASCII. При виконанні програми коди ASCII посилаються на термінал, який знову перетворює їх у зображення символів. Якщо ж у рядку, що виводиться на термінал, вказані не символи (у апострофах), а їх числові коди, то транслятор залишає їх у програмі без зміни. При посиланні на термінал вони сприйматимуться як коди ASCII і приведуть до того ж результату. Таким чином, пропозиції

```
string    db     'ABCD'
string    db   65,66,67,68
```

абсолютно еквівалентні, враховуючи, що код ASCII великої латинської літери A дорівнює 65, код букви B – 66 і т.д.

Очевидно, що описувати в програмі текстові рядки за допомогою кодів ASCII щонайменше незручно. В той же час для виведення на екран псевдографічних зображень або інших піктограм, для яких немає відповідних клавіш, доводиться користуватися кодами ASCII. Проте мати на увазі відповідність символів і кодів ASCII доводиться досить часто. Наприклад, впорядковування символів або символьних рядків за абеткою фактично виконується за кодами ASCII.

Для демонстрації техніки організації **циклів** розглянемо програму, в якій створюється і виводиться на екран текстовий символьний масив, заповнений

кодами алфавітно-цифрових і псевдографічних символів. Ці символи мають коди від 32 (пропуск) до 254 (суцільний квадратик). Такий масив можна створити вручну в полях даних програми за допомогою оператора db:

```
symbols    db    32,33,34,35,36,37,38,39...
```

проте простіше заповнити його даними програмно.

Лістинг 9.4. Виведення символьного масиву, заповненого кодами алфавітно-цифрових і псевдографічних символів:

```
    title CopyRight by Rysovaniy A. N.
    .686        ; директива определения команд микропроцессора
    .model  flat,stdcall  ; задание линейной модели памяти и соглашения ОС Windows
    option casemap:none        ; отличие строчных и прописных букв
    includelib \masm32\lib\kernel32.lib
    include \masm32\include\windows.inc ; файлы структур, констант …
    include \masm32\include\kernel32.inc ; системные функции применений…
.data                          ; директива определения данных
      output db 223 dup('*')   ; заповнюваний масив
  .code                            ; начало сегмента данных
  _st:                             ; метка начала программы
      mov ecx,223              ; кількість кроків у циклі
      mov esi,0                ; індекс елемента, що адресується
      mov al,32                ; код першого символу – пропуску
a1:  mov output[esi],al          ; занесення чергового  коду в масив
      inc al                       ; створення коду наступного символу
      inc esi                      ; зсув у масиві
    loop a1                        ; ecx – 1 та перехід на a1, якщо ecx ≠ 0
invoke GetStdHandle, STD_OUTPUT_HANDLE ; одержання дескриптора
invoke  WriteConsoleA, eax, \  ; дескриптор стандартного пристрою виведення
   offset output, \            ; адреса початку повідомлення
    223d,0,0 ; розмір повідомлення та адреса, де зберігається число символів
invoke ExitProcess,0 ; возврат управления  ОС и освобождение ресурсов
      end _st        ; директива завершение программы с именем_st
```

Результат роботи програми з лістингу 9.4 наведено на рис. 9.3.

Рис. 9.3. Результат роботи програми з лістингу 9.4

У полях даних програми оголошений масив байтів (оператор *db*), який ініціалізовано кодами символу “*”. Це дуже зручний і поширений прийом, який полегшує налагодження програми. Якщо через якісь помилки в програмі масив буде заповнений не весь або не заповнений взагалі, на екран будуть виведені зірочки. Якщо ж програма працює правильно, початкові зірочки зітруться.

Лічильником кроків циклу служить регістр ECX. Тому в регістр ECX потрібно занести необхідну кількость кроків циклу, що дорівнює довжині заповнюваного масиву. Робота з масивом здійснюється, як правило, за допомогою одного з індексних регістрів (ESI або EDI), в яких зберігається і нарощується індекс елемента масиву, що адресується, тобто номер байта масиву, до якого здійснюється звернення в даному кроці циклу. Оскільки обробка масиву виконується із самого початку, то в регістр ESI заноситься 0. Регістр AL вибраний для зберігання поточного коду символу, що відправляється в масив. Так точно цю роль міг би виконати будь-який інший байтовий регістр AH, BL, BH, DL або DH (регістри CL і CH, що входять до складу регістра CX, вже зайняті). В регістр AL заноситься код першого символу – пропуску. Підготувавши всі необхідні регістри, можна скласти саме тіло циклу. Для цього код із регістра AL відправляється в елемент (байт) масиву output, номер якого визначається вмістом індексного регістра ESI. Це так звана індексна адресація, в якій існує декілька різновидів. Зокрема, як індексний регістр так точно можна було би використовувати регістри EBX або EDI. У першому кроці циклу заповнюється елемент масиву з індексом 0. У наступних двох реченнях виконується інкремент (збільшення на 1) коду чергового символу й індексу в масиві. Нарешті, команда loop (петля) повертає керування на мітку a1. Причому робить вона це рівно 223 рази відповідно до початкового вмісту регістра ECX.

На рис. 9.4 наведено вікна налагоджувача після виведення результату виконанням програми з лістингу 9.4. В дампі пам'яті наведено коди символів, які виводяться.

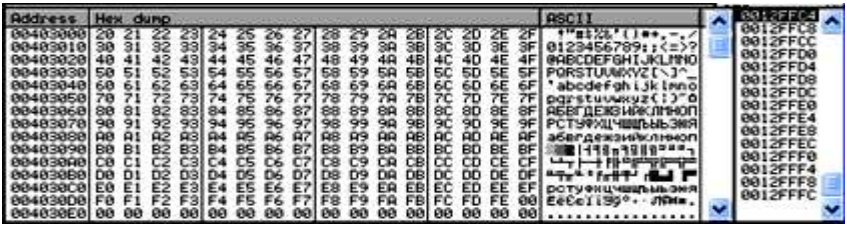


Рис. 9.4. Вікна налагоджувача після виведення результату виконанням програми з лістингу 9.4

Для **виведення звуку** через системний динамік використовують функції MessageBeep та Beep. Приклад використання функції Beep може бути таким:

```
invoke Beep,\ ;  генерування звуку в системному динаміку
 01234,\        ; частота звуку в герцах
  300           ; тривалість звуку в мілісекундах
```

Функція MessageBeep має один параметр та конкретизує звуковий сигнал:

```
invoke MessageBeep, uType
```

Параметр uType має значення:

```
0xFFFFFFFF  ; стандартний звуковой сигнал комп'ютерного спікера
MB_ICONASTERISK    ; SystemAsterisk
```

```
MB_ICONEXCLAMATION          ; SystemExclamation
MB_ICONHAND                 ; SystemHand
MB_ICONQUESTION   ; SystemQuestion
MB_OK                       ; SystemDefault
```

9.7. Виведення чисел

Для виведення чисел спочатку потрібно перетворити числа в послідовність символів, а потім виводити цю послідовність на екран процедурою WriteConsoleA.

Для такого перетворення в системі Windows є спеціальна процедура wsprintf.

Функція wsprintf

Функція (процедура) **wsprintf** (http://ru.wikipedia.org/wiki/Printf) необхідна для перетворення 32-розрядного числа в послідовність символів. Ця функція має такі параметри:

- адресу буфера, де процедура зберігає число у вигляді послідовності символів. Максимальний розмір буферу складає 1024 байти;
- адресу форматного рядка, що вказує процедурі, яке слід виконати перетворення;
- перетворюване число.

Прототип функції wsprintf зберігається у файлах user32.inc та user32.lib.

На відміну від багатьох інших процедур, кількість параметрів wsprintf **змінюється** і залежить від кількості перетворюваних чисел.

При форматуванні тексту включення в рядок форматів позначення \n приведе до зсуву повідомлення на наступний ряд (символ табуляції).

Формат перетворення змінної починається із знака відсотка (%), за яким слідує один або декілька символів. Найбільш поширені позначення форматів наведено в табл. 9.3.

Таблиця 9.3

Позначення формату	Призначення
%x, %X	Значення змінної виводиться у формі шістнадкового числа рядковими (%x) або великими (%X) символами
%#x, %#X	Те саме, але перед числом виводиться позначення 0x, що позначує шістнадцяткове число (для наочності)
%d, %i	Значення змінної виводиться у формі десяткового числа із знаком
%u	Значення змінної виводиться у формі десяткового числа без знака
%0*кіл*&	Те саме, але виводяться *кіл* десяткових розрядів, що вирівняні по правому краю. Розряди зліва заповнюються нулями

%c	Виводиться один символ, відповідний коду ASCII змінної
%s	Виводиться символьний рядок; змінна має бути ім'ям цього рядка або самими рядком

Програму, що виводить на екран ціле число 123456, показано в лістингу 9.5.

Лістинг 9.5. Виведення чисел на екран монітора:

```
    .686            ; директива определения типа микропроцессора
    .model  flat, stdcall; задание линейной модели памяти и соглашения ОС Windows
    option casemap:none       ; отличие строчных и прописных букв
    include \masm32\include\windows.inc ; файлы структур, констант …
    include \masm32\include\\user32.inc   ; файлы  интерфейса …
    include \masm32\include\kernel32.inc ; системные функции применений…
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
    BSIZE           equ 15      ; задание реального количества байтов
  .data
       lfmt     db "%d",0           ; задання перетворення одного символу
       buf      db BSIZE dup(?)  ; резервування пам'яті для буфера
       dig      dd 123456           ; число, яке необхідно вивести на консоль
stdout DWORD ? ; для збереження дескриптора виведення
cWritten  DWORD  ? ; для адреси символів виведення
  .code                             ; директива начала кода программы
  start:                            ; метка начала программы с именем start
invoke GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
     mov stdout, eax            ; збереження одержаного дескриптора у пам'яті
invoke  wsprintf, \ ; функція перетворення 32-розрядного числа в символи
  ADDR buf, \ ; адреса буфера, куди буде записано послідовність символів
    ADDR ifmt, \            ; адреса рядка перетворення формату
       dig                  ; число, яке перетворюється
invoke  WriteConsoleA, \  ; API-функція виведення на екран
   stdout, \                  ; дескриптор стандартного пристрою виведення
        ADDR buf, \         ; адреса початку повідомлення
            BSIZE, \            ; розмір повідомлення
               ADDR cWritten, 0  ; адреса, де зберігається число символів
invoke ExitProcess, 0 ; возвращение управления ОС та вивільнення ресурсів
       end start                ; директива завершение программы с именем start
```

У цій програмі використовуються файли user32.inc і user32.lib, що зберігають інформацію про функцію wsprintf. Кількість її параметрів у нас поки дорівнює трьом. Перший параметр ADDR buf – адреса буфера, куди буде записано послідовність символів. Пам'ять для буфера виділяється рядком buf db BSIZE dup(?), який резервує кількість байтів, заданих змінною BSIZE. Про резервування байтів пам'яті свідчить слово dup – скорочене від англійського слова *duplication* (повторення). Знак питання в дужках після dup означає, що значення байтів наперед не визначене.

Розмір буфера, позначений ім'ям BSIZE, є реальним числом (яким асемблер замінить BSIZE), що задається рядком BSIZE equ 15.

Другий параметр функції wsprlntf – ADDR ifmt – це адреса рядка формату, який задає тип перетворення. Цей рядок складається з символів і завжди завершується нульовим байтом. Рядок “%d”,0 задає перетворення одного цілого числа в послідовність символів, а рядок “%d%d”,0 – перетворення двох чисел.

Після виклику функції wsprintf асемблер додає в лістинг інструкцію add esp, 12. Це свідчить про те, що програма завантажує в стек три параметри процедури wsprintf, які здійснює інструкція add esp, 12.

Програму, що показує виведення чисел на екран від 1 до 10, наведено в лістингу 9.6.

Лістинг 9.6. Виведення на екран чисел від 1 до 10:

```
        .686            ; директива определения команд микропроцессора
.model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none             ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\\user32.inc ; файлы  интерфейса …
include \masm32\include\kernel32.inc ; системные функции применений …
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
        BSIZE     equ 15              ; задание реального количества байтов
.data                                 ; директива определения данных
        frmt   db  "%d",0             ; задание преобразования одного символа
        buf    db BSIZE dup(?)   ; резервирование памяти для буфера
        crlf    db 0dh,0ah
stdout DWORD ?  ; для сохранения дескриптора вывода
cWritten  DWORD  ?  ; для адреса символов вывода
.code                                   ; директива начала кода программы
  start:                                ; метка начала программы с именем start
    invoke  GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
        mov stdout, eax       ; запам’ятовування отриманого дескриптора
        mov edx,1             ; занесення первинного значення
        mov ecx,10              ; занесення максимального значення
  nxt:                            ; початок циклу
        push    ecx
        push    edx
invoke  wsprintf, \  ; API-функція перетворення числа
  ADDR buf, \ ; адреса буфера, куди буде записана послідовність символів
     ADDR frmt, \               ; адреса рядка перетворення формату
        edx                             ; регістр, вміст якого перетворюється
invoke  WriteConsoleA, \  ; API-функція виведення на екран
 stdout, \              ; дескриптор стандартного пристрою виведення
 ADDR buf, BSIZE,\ ; адреса початку повідомлення  та розмір повідомлення
  ADDR cWritten, 0 ;      ; адреса, де зберігається кількість символів
invoke  WriteConsoleA, stdout, \   ; API-функція виведення на екран
             ADDR crlf, 2, ADDR cWritten, NULL
        pop    edx
        inc    edx
        pop    ecx
  loop   nxt                ; ecx – 1 та перевірка ecx. Якщо ecx = 0, то – вихід
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start               ; директива завершение программы с именем start
```

У зв’язку з тим, що ця програма консольна, то неможливо використовувати середовище masm32. Необхідно створити bat-файл та вказати параметр

/SUBSYSTEM:CONSOLE. А щоб побачити процес виконання exe-файла простіше використати файловий менеджер FAR.

У цій програмі функція wsprintf перетворює число, яке знаходиться у регістрі edx, в послідовність символів.

Процедура WriteConsoleA виводить ці символи на екран. За першим параметром цієї функції (stdout) перетворене у символ число виводиться через консоль, за другим (параметр ADDR buf) вказується адреса початку повідомлення, за третім (BSIZE) – розмір повідомлення, за четвертим (ADDR cWritten) – адреса ділянки пам'яті, де процедура WriteConsoleA зберігає кількість виведених символів на екран.

Другий виклик процедури WriteConsoleA потрібний для переведення рядка і повернення курсора виведення (каретки) до лівого краю екрана (цим відають символи 0dh, 0ah або, в десятковому поданні, – символи 13, 10). Цією процедурою параметр ADDR crlf вказує на те, що виконуватиметься звернення до змінної crlf, яка має опис db 0dh, 0ah. Третій параметр (2) вказує на те, що в змінній crlf є два символи.

Відображення на екрані процесу виконання програми наведено на рис. 9.5.

Перед початком циклу в регістр ecx посилається число 10, а в регістрі edx опиняється одиниця. Далі обидва регістри зберігаються в стеку:

```
push    ecx
push    edx
```

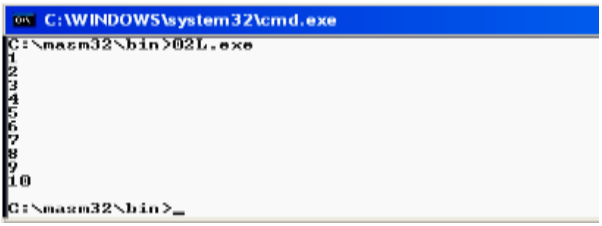


Рис . 9.5. Відображення на екрані процесу виконання програми з лістингу 9.6

Робиться це через те, що **процедури Windows самі використовують ці регістри** для своїх внутрішніх потреб.

Після збереження регістрів на екран виводиться поточне число.

Для того щоб числа вивести за горизонталлю, необхідно закоментувати рядок invoke WriteConsoleA, stdout, ADDR crlf, 2, ADDR cWritten, 0

Розглянемо API-функцію **MessageBox**, що створює спрощене вікно, в яке можна занести результат виконання програми. Приклад використання цієї функції наведено у програмі з лістингу 9.7.

Лістинг 9.7. Програма з використанням API-функції MessageBox:

```
.686            ; директива определения типа микропроцессора
```

```
    .model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
    option casemap:none ; відмінність рядкових та великих літер
    include \masm32\include\windows.inc  ; файлы структур, констант …
    include \masm32\include\kernel32.inc  ; системные функции применений …
    include \masm32\include\\user32.inc    ; файлы  интерфейса …
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
.data        ; директива определения данных
       mem1 dd 111
titl db " Исследование функции MessageBox ",0 ; название упрощенного окна
st1 dw ?,0                    ; буфер для вывода сообщения
ifmt  db "Вывод чисел с памяти и регистров:", 0dh,0ah,
 "mem1 = %d", 0dh,0ah, ; задание преобразования символа  mem1
  "x = %d",0dh,0ah,         ; задання преобразования символа  x
   "y = %d", 0dh,0ah,0ah,       ; задание перетворення символу  y
    "Автор программы:",0       ;
.code
start:
       mov edx,333                ; число в edx для вывода
       mov ebx,7777               ; число в ebx для вивода
invoke  wsprintf, \     ; API- функция преобразования числа
 ADDR st1, \     ; адрес буф., куда будет записана послед. символов
  ADDR ifmt, \               ; адрес строки преобразования формата
   mem1, edx, ebx            ; регистр, содержание которого преобразовывается
invoke MessageBox, \      ; API- функция вывода окна консоли
 NULL, \              ; hwnd – идентификатор окна
  addr st1, \                ; адрес строки, которая содержит  текст сообщения
   addr titl, \          ; адрес строки, которая содержит  заглавие сообщения
    MB_ICONINFORMATION;+180000h          ; вид окна
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
       end start        ; директива окончания программы с именем start
```

Рядком st2 dw ?,0 для змінної st2 зарезервовано всього 2 байти. Цього достатньо для збереження найбільшого числа в цій програмі, тому що число 7777 = 1E61h, для якого необхідно лише 2 байти пам'яті. Якщо якесь число в програмі буде збільшено, то необхідно скоректувати й розмір буфера. Результат виконання програми з лістингу 9.7 наведено на рис. 9.6,а. А якщо до типу вікна додати константу 180000h або 1 572 864 (наприклад, **MB_OK + 180000h**), то вікно буде мати вигляд, який наведено на рис. 9.6,б. Для додаткового переведення рядка (для пропуску рядка) використано два курівних коди 0ah після перетворення символу y.

Виведенням цього вікна відповідає API-функція MessageBox, яка має прототип

```
int MessageBox(
  HWND hWnd,
  LPCSTR lpText,
  LPCSTR lpCaption,
  UINT uType,
  WORD wLanguageId ; зарезервовано
)
```

а б

Рис. 9.6. Результат виконання програми з лістингу 9.7:
а – нормальний вигляд вікна; б – «перевернутий» вигляд вікна

Параметри:

- hWnd – ідентифікатор вікна, з якого викликається MessageBox;
- lpText – адреса рядка, яка містить текст повідомлення;
- lpCaption – адреса рядка, яка містить заголовок повідомлення;
- uType – вигляд вікна.

Суфікс A у функції MessageBox означає, що ця функція працює з однобайтовими рядками ANSI. Для двобайтових Unicode-рядків до цієї функції додається суфікс W: MessageBoxW.

Оператор addr використовується для передавання ***адреси*** мітки функції. Він дійсний тільки в контексті директиви invoke.

Останній параметр (utype) задає вигляд діалогового вікна:

MB_OK – вікно повідомлення, яке містить одну командну кнопку «OK» (за умовчанням).

MB_OKCANCEL – вікно повідомлення, яке містить дві командні кнопки: «OK» та «Отменить» (Cancel).

MB_RETRYCANCEL – вікно повідомлення, яке містить дві командні кнопки: «Повторить» (Повторення) та «Отменить» (Відміна).

MB_YESNO – вікно повідомлення, яке містить дві командні кнопки: «Да» (Yes) та «Нет» (No).

MB_YESNOCANCEL – вікно повідомлення, яке містить три командні кнопки: «Да» (Yes), «Нет» (No) та «Отменить» (Cancel).

MB_HELP – додає у вікно повідомлення кнопку «Справка» (Help). Коли користувач клацає по кнопці «Справка» (Help) або натискає клавішу F1, система відправляє власникові повідомлення WM_HELP.

MB_CANCELTRYCONTINUE – вікно повідомлення, яке містить три командні кнопки: «Отменить» (Cancel), «Попытаться снова» (Try Again), «Продолжить» (Continue). Використовується замість типу вікна MB_ABORTRETRYIGNORE.

MB_ABORTRETRYIGNORE – вікно повідомлення, яке містить три командні кнопки: «Прекратить» (Abort), «Повторить» (Retry) та «Пропустить» (Ignore).

Для того щоб показати на екрані значок у вікні повідомлення, необхідно встановити одне з перерахованих нижче значень:

MB_ICONEXCLAMATION – у вікні повідомлення з'являється ікона знака оклику.

MB_ICONWARNING – у вікні повідомлення з'являється ікона знака оклику.

MB_ICONINFORMATION – у вікні повідомлення з'являється значок, що складається з рядкової букви *i* в колі.

MB_ICONASTERISK – у вікні повідомлення з'являється значок, що складається з рядкової букви *i* в колі.

MB_ICONQUESTION – у вікні повідомлення з'являється ікона знака питання.

MB_ICONSTOP – у вікні повідомлення з'являється значок стоп-сигналу.

MB_ICONERROR – у вікні повідомлення з'являється значок стоп-сигналу.

MB_ICONHAND – у вікні повідомлення з'являється значок стоп-сигналу.

Для того щоб вказати основну кнопку (за умовчанням), потрібно встановити одне з перерахованих нижче значень:

MB_DEFBUTTON1 – перша основна кнопка (кнопка, що використовується за умовчанням). MB_DEFBUTTON1 – значення за умовчанням, якщо MB_DEFBUTTON2, MB_DEFBUTTON3, або MB_DEFBUTTON4 не визначені;

MB_DEFBUTTON2 – друга основна кнопка;

MB_DEFBUTTON3 – третя основна кнопка;

MB_DEFBUTTON4 – четверта основна кнопка.

Для того щоб вказати модальність діалогового вікна, необхідно встановити одне з перерахованих нижче значень:

MB_APPLMODAL – користувач повинен відповісти вікну повідомлення перед продовженням роботи у вікні, ідентифікованому параметром *hwnd*. При цьому користувач може переміщатися у вікна інших потоків і працювати в цих вікнах. Залежно від ієрархії вікон у застосуванні користувач може переміщатися в інші вікна в межах потоку. Всі повідомлення дочірнього вікна автоматично блокуються, проте висхідні вікна не блокуються.

MB_APPLMODAL є значенням за умовчанням, якщо ні MB_SYSTEMMODAL, ні MB_TASKMODAL не визначені.

MB_SYSTEMMODAL – те саме, що і MB_APPLMODAL, за винятком того, що вікно повідомлення має стиль WS_EX_TOPMOST. Використовуйте повідомлення, що працюють в системному режимі (недоступному для користувача) вікна, щоб повідомити користувача про серйозні, потенційно руйнівні помилки, які потребують негайної уваги (наприклад, вихід за межі обсягу пам'яті). Цей прапорець не має ніякого впливу на здатність користувача взаємодіяти з іншими вікнами, а не тільки з вікнами, пов'язаними з *hwnd*.

MB_TASKMODAL – те саме, що і MB_APPLMODAL, за винятком того, що всі вікна верхнього рівня, що належать поточному потоку, блокуються, якщо параметр *hwnd* = 0. Використовується прапорець тоді, коли програма або

бібліотека не мають доступного дескриптора вікна, але при цьому вони повинні перешкодити введенню даних в інші вікна в зайнятому потоці, не припиняючи роботу інших потоків.

Інші різновиди параметра повністю розглянуто в MSDN (www. netcode.ru).

Параметр utype дозволяє виконувати об'єднання, наприклад

MB_YESNO or MB_ICONERROR.

Результат такого об'єднання наведено на рис. 9.7.

Рис. 9.7. Результат об'єднання параметрів utype: MB_YESNO or MB_ICONERROR

Для того щоб дізнатися про вплив параметра utype на внутрішні регістри мікропроцесора, можна зробити таке:

1. Відкрити налагоджувану програму, яка містить функцію Message-Box у вікні налагоджувача OllyDbg.

2. Підвести курсор екрана на рядок CALL <JMP.&user32.MessageBoxA> та поставити ***контрольну точку*** (натиснути на клавішу F2). При цьому поле адрес зафарбується у червоний колір.

3. Натиснути на клавішу F9 (виконати фрагмент програми до контрольної точки).

4. Натиснути клавішу F8 (повністю виконати вибрану функцію).

5. На екрані з'явиться вікно MessageBox з відповідними кнопками. Натиснути на вибрану кнопку та прослідкувати за зміною регістра EAX. Для того щоб зняти контрольну точку, необхідно ще раз натиснути на кнопку F2.

Значення, які повераються зведені в табл. 9.4.

Отримані значення регістра EAX можна використовувати для продовження виконання програми та виведення додаткових вікон (наприклад, з результатами).

Таблиця 9.4

Псевдоним	Значення	Кнопка
IDOK	1	OK
IDCANCEL	2	Отмена (Cancel)
IDABORT	3	Прервать (Abort)
IDRETRY	4	Повтор (Retry)
IDIGNORE	5	Пропустить (Ignore)
IDYES	6	Да (Yes)
IDNO	7	Нет (No)

Для виведення декількох чисел (**масиву чисел**, якщо масив невеликий**)** можна вказувати місця розташування цих чисел так, як наведено у фрагмені програми:

```
mas1 dd 0Ah,0Bh,1,2,3,4 ; определение массива с именем mas1
_title db "Результаты решения программы",0
```

```
        buf dd ?,0                  ; буфер вывода сообщения
        ifmt  db  6 dup(" %d "),0   ; формат преобразования чисел
.code
        _st:            ; метка начала программы
invoke  wsprintf, \     ; API- функция преобразования числа (чисел)
 ADDR buf, \ ; адрес буфера, куда будет записана последовательность символов
  ADDR ifmt, \          ; адрес строки преобразования формата
 mas1,mas1[4],mas1[8],mas1[12],mas1[16],mas1[20] ; ячейки для преобразования
invoke MessageBox, 0, addr buf, addr _title, MB_ICONINFORMATION
        invoke ExitProcess,0
        end _st         ; директива завершение программы с именем_st
```

Для того щоб числа при виведенні функцією MessageBox не зливалися один з одним, в рядку формату перетворення ifmt добавлені додаткові пропуски. Результат такого виведення наведено на рис. 9.8.

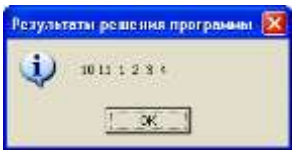


Рис. 9.8. Результат виведення масиву чисел

Для тимчасового виведення повідомлення, обмеженого часом використовується недокументована функція **MessageBoxTimeOut**, яка має такий синтаксис:

```
invoke MessageBoxTimeOut, \ ; виведення обмеженого часом повідомлення
 hWnd,\                     ; хендл вікна
  lpText,\                  ; адреса рядка, яка містить  текст повідомлення
   lpCaption, \             ; адреса рядка, яка містить  заголовок повідомлення
    uType, \                ; вигляд діалогового вікна
    wLanguageId,\                   ; мова повідомлення
     dwMilliseconds                 ; мілісекунди відображення вікна
```

Приклад використання функції MessageBoxTimeOut може бути таким:

```
invoke MessageBoxTimeOut, \ ; виведення обмеженого часом повідомлення
   NULL, \                                  ; hwnd – ідентифікатор вікна
    addr st2, \                 ; адреса рядка, яка містить  текст повідомлення
     addr st1, \          ; адреса рядка, яка містить  заголовок повідомлення
      MB_OK,\                               ; вигляд діалогового вікна
       0,2000           ; мова повідомлення та мілісекунди відображення вікна
```

Функція MessageBoxTimeOut може використовуватися у програмах автоматичної обробки даних, коли потрібно вивести запит на дію користувача і, якщо користувач недоступний, виконати дію за умовчанням.

У наступному фрагменті програми використовується команда **RDTSC** (англ. Read Time Stamp Counter), яка зчитує лічильник TSC (Time Stamp Counter) та повертає в регістри EDX:EAX 64-бітову кількість тактів із моменту

останнього скидання мікропроцесора. Для визначення частоти мікропроцесора використовуються дві команди **rdtsc**. Друга команда виконується через 1 с після першої, та за рахунок віднімання з останьої кількості тактів попереднього значення отримується 64-розрядна кількість тактів за 1 с. Для того щоб отримати ціле значення частоти, використовується розмірність у кілогерцах. Це значення перетворюється в десятковий код функцією wsprintf,addr buf1,addr frmt,digEAX за допомогою параметра перетворення %u, визначеного рядком перетворення з іменем frmt. Ця команда використовується для мікропроцесорів, починаючи з 586.

```
        .586
        …
      titl db "Исследование команды RDTSC",0  ; заголовок окна
      digEAX dd ?     ; змінна
      buf1 dd ?       ; символьний рядок числа
   frmt db "Частота работы микропроцессора %u -кГц",0 ; десяткова система
.code
  start:
rdtsc ; в EDX:EAX кількість тактів з моменту останього скидання процесора
        mov esi,eax ; пересилання молодшої частини числа для порівняння
        mov edi,edx     ; пересилання старшої частини числа для порівняння
        invoke Sleep, 1000  ; затримка роботи процесора на 1 секунду
rdtsc                   ; рахуємо лічильник TSC з кількістю тактів за 1 секунду
      sub eax,esi ; розрахунок молодшого числа тактів процесора за секунду
      sbb edx,edi ; розрахунок старшого числа тактів процесора за секунду
      mov ebx,1000000 ; з Гц в кГц, але збільшуємо цілу частина в 1000 разів
      div ebx         ; EDX:EAX/EBX та результат - в EAX
      mov digEAX,eax
invoke wsprintf,addr buf1,addr frmt,digEAX ; переводимо число в символи
invoke MessageBox,0,addr buf1,addr titl,0 ; виводимо символ у вікно
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start               ; директива завершение программы с именем start
```

Результат роботи програми наведено на рис. 9.9. Можлива неточність обчислення отримується за рахунок округлення, наприклад до 8 тактів.

Для отримання значення частоти у вигляді дійсного числа необхідно використовувати команди співпроцесора.

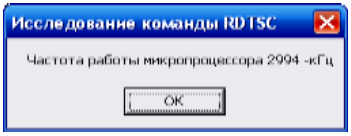


Рис. 9.9. Вікно з виведення частоти роботи мікропроцесора

9.8. Введення з консолі

Консоль Win32 має вхідний буфер, що містить масив вхідних записів. Кожна подія, така, як натиснення клавіші, переміщення миші або клацання кнопкою миші, створює запис у вхідному буфері консолі. Функції високого

рівня ReadConsole фільтрують і обробляють вхідні дані, повертаючи тільки потік вхідних символів.

Функція ReadConsole використовується для зчитування текстових даних і розміщення їх у буфер. Прототип цієї функції

```
ReadConsole PROTO
 handle:DWORD,                          ; дескриптор входу
  pBuffer:PTR BYTE      ; вказівник буфера, куди потрапляють символи
   maxBytes:DWORD,                      ; кількість символів для читання
    pBytesRead:PTR :DWORD  ; вказівник на кількість зчитаних символів
     notUsed :DWORD                     ; не використовується
```

Параметр handle – це дескриптор, який повертається функцією GetStdHandle. Параметр pBuffer є адресою символьного масиву. Параметр maxBytes є 32-розрядним цілим числом і указує на максимальну кількість символів, які повинні бути зчитані. Параметр pBytesRead указує на подвійне слово, яке означає кількість зчитаних символів. Останній параметр не використовується і його можна замінювати будь-яким числом, наприклад нулем.

Процедура ReadConsole схожа з процедурою WriteConsoleA. Програма, показана в лістингу 9.8, вводить із клавіатури послідовність символів і потім відображає її на екрані.

Лістинг 9.8. Введення з клавіатури і відображення на екрані символів:

```
    .686                ; директива определения типа микропроцессора
    .model  flat, stdcall  ; задание линейной модели памяти и соглашения ОС Windows
    option casemap:none        ; отличие строчных и прописных букв
    include \masm32\include\windows.inc ; файлы структур, констант …
    include \masm32\include\kernel32.inc ; системные функции…
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
    BSIZE   equ 128
.data                           ; директива определения данных
    buf     BYTE BSIZE dup(?)  ; резервування пам'яті для буфера
    stdout DWORD ?          ; комірка для збереження дескриптора виведення
    stdin  DWORD ?          ; комірка для збереження дескриптора введення
      cRead DWORD ?
      cWritten  DWORD ?
.code                               ; директива начала кода программы
  start:                            ; метка начала программы с именем start
    invoke  GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
    mov  stdout, eax ; збереження дескриптора виведення
    invoke  GetStdHandle, STD_INPUT_HANDLE ;
    mov  stdin, eax   ; збереження дескриптора введення в комірці пам'яті
NewLine:
 invoke  ReadConsole, stdin, ADDR buf, BSIZE,\
                        ADDR cRead, NULL     ; читання символу з клавіатури
 invoke  WriteConsoleA, stdout, ADDR buf, cRead,\
                ADDR cWritten, 0   ; відображення символу на екрані
      cmp   cRead,2                 ; перевірка натискання клавіші Enter
      jnz     NewLine               ; перейти, якщо не натиснуто клавішу Enter
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
      end start                     ; закінчення програми
```

У програмі з лістингу 9.8 використовуються дескриптори двох стандартних пристроїв: екрана (stdout) і клавіатури (stdin). Процедура ReadConsole приймає ті самі параметри, що і WriteConsoleA: дескриптор пристрою (stdin), адресу масиву байтів, куди потраплять введені символи (ADDR buf), розмір масиву (BSIZE), адресу подвійного слова, куди процедура запише кількість прочитаних символів (ADDR cRead), та значення NULL.

Введення символів завершується клавішею Enter. При натисканні клавіші Enter вводяться два символи: 0Dh та 0Ah. Перший символ повертає курсор до лівого краю екрана, а другий переводить курсор на новий рядок. Змінна cRead := 2 свідчить про закінчення введення.

Після натиснення клавіші Enter функція ReadConsole закінчує роботу і виконується функція WriteConsoleA, що повторює введені символи на екрані.

Відображення на екрані процесу виконання програми наведено на рис. 9.10.

Рис. 9.10. Відображення на екрані процесу виконання програми з лістингу 9.8

Розглянемо програму, яка виконує додавання двох цілих чисел та виведення на екран послідовні запити на введення чисел і результат операції.

У лістингу 9.9 використовується функція crt_atoi, яка перетворює символ в число та зберігає результат в eax.

Лістинг 9.9:

```
title CopyRight by Rysovaniy A. N.
; програма консольного введення чисел та виведення для операції a + b
.686                ; директива определения типа микропроцессора
.model  flat, stdcall ; задание линейной модели памяти
option casemap:none         ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include    \masm32\include\\user32.inc   ; файлы  интерфейса …
include \masm32\include\kernel32.inc ; файли систем. функцій застосувань…
include   \masm32\include\msvcrt.inc
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\msvcrt.lib
.data                                   ; директива определения данных
  buf db 30 dup(?)              ; резервування пам'яті для буфера
  stdout dd ?                   ; комірка для збереження дескриптора виведення
  stdin dd ?                    ; комірка для збереження дескриптора введення
  cRead dd ?                    ; кількість зчитаних символів
   msg1 db "Input digit a: "
   msg2 db "Input digit b: "
   msg3 db "Result a+b = "
   fmt db "%d",0 ;
   temp dd 0                    ; комірка для тимчасового збереження
.code
_start:
```

```
invoke GetStdHandle,STD_OUTPUT_HANDLE
                mov stdout,eax     ; збереження дескриптора виведення у пам'яті
invoke GetStdHandle,STD_INPUT_HANDLE
                mov stdin,eax      ; збереження дескриптора введення у пам'яті
invoke WriteConsoleA,                          ; виведення повідомлення msg1
  stdout,                        ; дескриптор стандартного пристрою виведення
   ADDR msg1,                                  ; адреса початку повідомлення
    15,                        ; розмір повідомлення (кількість символів в msg1)
      NULL,NULL
invoke ReadConsole,                            ; читання  числа "a" з консолі
 stdin,                                        ; дескриптор введення
  ADDR buf,                          ; адреса буфера для введених символів
    10, ADDR cRead,  0  ; кількість символів для читання та адреса масиву символів
invoke crt_atoi,addr buf      ; перетворення рядка в число, результат в eax
      mov temp,eax     ; зберегти число a
invoke WriteConsoleA,stdout,ADDR msg2,15,NULL,NULL  ; виведення msg2
invoke ReadConsole,stdin,ADDR buf,10,addr cRead,NULL ; читання числа b
invoke crt_atoi,addr buf      ; перетворення рядка в число, результат в eax
      add eax,temp                     ; a + b
invoke  wsprintf, ADDR buf, \ ; адр. буф., куди буде записана послідовність символів
     ADDR fmt, \                       ; адреса рядка перетворення формату
      eax                        ; регістр, вміст якого перетворюється
invoke WriteConsoleA,stdout,ADDR msg3,13,NULL,NULL ; виведення msg3
invoke WriteConsoleA,stdout,ADDR buf,10,NULL,NULL ; виведення результату
invoke Sleep,1000 ; затримка виконання програми  на число мілісекунд
invoke ExitProcess,0 ; возвращение управления ОС Windows та вивільнення ресурсів
      end _start                       ; закінчення програми
```

Результат виконання програми з лістингу 9.9 наведено на рис. 9.11.

Рис. 9.11. Результат виконання програми з лістингу 9.9

9.9. Керування кольором

Встановлювати кольори вікна косолі можна двома шляхами:

1) змінювати поточний колір символів за допомогою функції SetConsoleTextAttribute, яка діє на всі подальші символи, що виводяться на консоль;

2) встановлювати атрибути окремих комірок, викликаючи функцію WriteConsoleOutputAttribute.

Функція ***SetConsoleTextAttribute*** дозволяє встановити колір символів і колір фону для тексту, що виводиться на консоль. Її прототип такий:

```
SetConsoleTextAttribute PROTO
 outHandle:dword                ; дескриптор консолі
  nColor:dword                  ; атрибути кольору
```

Значення кольору зберігаються в молодшому байті параметра nColor. Параметр nColor формується як логічне АБО таких значень:

Foreground_blue=0001h – синій текст;
Foreground_green=0002h – зелений текст;
Foreground_red=0004h – червоний текст;
Foreground_intensity=0008h – текст підвищеної яскравості;
Background_blue=0010h – блакитний фон;
Background_green=0020h – зелений фон;
Background_red=0040h – червоний фон;
Background_intensity=0080h – фон підвищеної яскравості.

Для задання білого кольору складаються три компоненти, для задання чорного компоненти не задаються.

У програмі з лістингу 9.9 наведено виведення 15 рядків кольорових символів на чорному фоні.

Лістинг 9.10. Виведення кольорових символів у консоль:

```
        .686            ; директива определения типа микропроцессора
        .model  flat, stdcall  ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none         ; отличие строчных и прописных букв
    include \masm32\include\windows.inc ; файлы структур, констант …
    include C:\masm32\include\\user32.inc ; файлы  интерфейса …
    include \masm32\include\kernel32.inc ; системные функции
    includelib C:\masm32\lib\user32.lib
    includelib C:\masm32\lib\kernel32.lib
.data                               ; директива определения данных
            stdout      dd          0
            mytitle     db          "Console Color Text Example", 0
            color       dd          1
            messg1      db          "**********************", 10, 13, 0
            cwrite      dd          0
messg2 db "Пример использования цветного текста в консольном окне", 0
            titl    db    "Color text sample", 0
.code
    start:
    invoke GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
    mov stdout, eax ; збереження одержаного дескриптора у пам'яті
invoke SetConsoleTitle, offset mytitle          ; назва вікна консолі
            mov ecx, 15                         ; лічильник виведених рядків
_cycle:   push ecx
invoke SetConsoleTextAttribute, stdout, color ; задання кольору
invoke WriteConsoleA, stdout, offset messg1, sizeof messg1,  offset cwrite, 0
            inc color
            pop ecx
            loop _cycle
            invoke MessageBox, 0, offset messg2, offset titl, 0
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end _start          ; закінчення програми
```

Результат виведення символів у консоль наведено на рис. 9.12.

Розглянемо приклад введення та виведення в(з) консоль(і) кольорового фону. Ця програма виконує віднімання двох цілих чисел (лістинг 9.11).

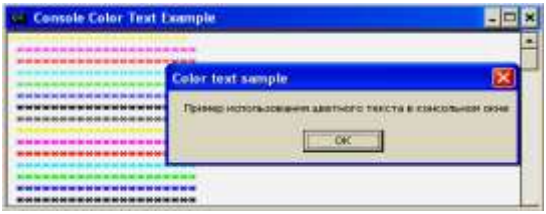


Рис. 9.12. Результат роботи програми з лістингу 9.10

Лістинг 9.11:

```
;  програма консольного введення цілих чисел та виведення
; результату операції a - b в консоль кольорового фону
     .686  ; директива определения типа микропроцессора
     .model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
     option casemap:none          ; отличие строчных и прописных букв
     include \masm32\include\windows.inc ; файлы структур, констант …
     include    \masm32\include\\user32.inc ; файлы  интерфейса …
     include \masm32\include\kernel32.inc ; системные функции
     include   \masm32\include\msvcrt.inc
     includelib \masm32\lib\user32.lib
     includelib \masm32\lib\kernel32.lib
     includelib \masm32\lib\msvcrt.lib
     BSIZE equ 40
.data                                        ; директива определения данных
       buf BYTE BSIZE dup(0)
       buf1 BYTE BSIZE dup(0)
     stdout DWORD 0        ; комірка для збереження дескриптора виведення
     stdin   DWORD 0        ; комірка для збереження дескриптора введення
       a DWORD 0             ; резервування комірки для збереження числа a
       b DWORD 0             ; резервування комірки для збереження числа b
       cWritten DWORD 0
       frmt db "%d",0
msg db "Program of calculation the difference of two numbers",
                                  0dh,0ah  ; назва програми
msg1 db "Enter first number:",0  ; повідомлення на введення 1-го числа
msg2 db "Enter second number:",0 ; повідомлення на введення 2-го числа
msg3 db "Result: ",0  ;
       temp dd 0                             ; ячейка для хранения результата
       dig dd 0
.code                                        ; директива начала кода программы
_st:                                         ; метка начала программы с именем_st
     invoke GetStdHandle,STD_OUTPUT_HANDLE ; отримання дескриптора
       mov stdout,eax      ; збереження дескриптора виведення у пам'яті
     invoke GetStdHandle,STD_INPUT_HANDLE
       mov stdin,eax         ; збереження дескриптора введення у пам'яті
     invoke SetConsoleTitle,addr msg
     invoke SetConsoleTextAttribute,stdout, BACKGROUND_GREEN
invoke WriteConsole,   stdout,          ; дескриптор стандартного пристрою виведення
  ADDR msg, sizeof msg,    ; адреса початку повідомлення та  розмір повідомлення
    ADDR cWritten,NULL       ; адреса, де зберігається кількість символів
invoke WriteConsole,stdout,ADDR msg1,sizeof msg1,
                      ADDR cWritten,NULL     ; виведення 1-го запиту
invoke ReadConsole,  ; зчитування числа з консолі
 stdin,                ; дескриптор введення
  ADDR buf1,       ; адреса буфера для введених символів
   BSIZE,               ; кількість символів для читання
    ADDR a, NULL        ;  вказівник на кількість зчитаних символів
```

```
invoke crt_atoi,addr buf1  ; перетворення рядка в число, результат в eax
        mov temp,eax              ; збереження числа a
invoke WriteConsole,stdout,ADDR msg2,sizeof msg2,
                          ADDR cWritten,NULL       ; виведення 2-го запиту
invoke ReadConsole,stdin,ADDR buf,BSIZE, ADDR b,NULL ; зчитування
invoke crt_atoi,addr buf   ; перетворення рядка в число, результат в eax
        sub temp,eax         ; a - b
invoke WriteConsole,stdout,ADDR msg3,sizeof msg3, ADDR cWritten,NULL
invoke wsprintf,   ; API-функція перетворення числа
 addr buf,     ; адреса буфера, куди буде записана послідовність символів
  addr frmt,    ; адреса рядка перетворення формату
      temp        ; комірка, яка перетворюється
invoke WriteConsole,  stdout,  ADDR buf, BSIZE, ; адреса початку  та розмір повідомлення
    ADDR dig,NULL    ; адреса, де зберігається кількість символів
invoke Sleep, 2000d  ; затримка виконання програми  на кількість мілісекунд
invoke ExitProcess,0  ; возврат управления  ОС  та вивільнення ресурсів
        end _st            ; закінчення програми
```

Результат виконання програми з лістингу 9.11 наведено на рис. 9.13.

Рис. 9.13. Результат виконання програми з лістингу 9.11

Функція WriteConsoleOutputAttribute

Ця функція копіює масив значень атрибутів послідовних комірок екранного буфера консолі, починаючи з вказаної позиції.

Її прототип такий:

```
WriteConsoleOutputAttribute PROTO
 outHandle:dword                ; дескриптор виходу
  pAttribute:ptr word           ; записувані атрибути
   nLength:dword                        ; кількість комірок
    xyCoord:coord                       ; координати першої комірки
     lpCountr:ptr dword                 ; кількість записаних комірок
```

Параметр pAttribute вказує на масив атрибутів, де кожен молодший байт включає інформацію про колір. Параметр nLength визначає довжину масиву. Параметр xyCoord визначає початкову комірку. Параметр lpCountr указує на змінну, яка містить кількість записаних комірок.

9.10. Лабораторна робота «Введення-виведення даних»

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм із використанням масивів, написаних мовою асемблера для програмування розгалужених процесів МП платформи x86 із застосуванням API-функцій під Win32 та **введенням-виведенням** даних на екран.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати варіант завдання та написати на асемблері програму обчислення одного з прикладів з ***виведенням*** даних. При цьому використати **спрощені** віконця функцію MessageBoxIndirect з повідомленнями: в одному віконці вивести своє прізвище та номер своєї навчальної групи; при натисканні на другу кнопку – вивести повне завдання із звуковим повідомленням. Використання інших кнопок спрощеного віконця виведення повідомлень – за бажанням. Використати – "перевернутий" вигляд вікна.

Варіанти завдань

1. Задано масив A з $N = 5$ елементів. Вивести суму елементів масиву A, для яких біти 0 дорівнюють нулю та вивести значення цих чисел.

2. Задано масив A з $N = 6$ елементів Вивести суму елементів масиву елементів масиву A, для яких біти 2 та 7 збігаються.

3. Задано масиви A і B з $N = 5$ елементів. Вивести масив C, який сформовано за правилом: якщо у елементів масивів Ai та Bi біти 0 та 1 збігаються, то $Ci = Ai – Bi$.

4. Задано масив A з $N = 7$ елементів. Вивести масив B з елементів масиву A, у яких 0, 2 та 5 біти мають нулі.

5. Задано масиви A і B з $N = 8$ елементів. Вивести масив C, який сформовано за правилом: якщо у елементів Ai та Bi біти 0 та 2 збігаються, то $Ci = Ai + Bi$.

6. Задано масиви A і B з $N = 4$ елементів. Вивести масив C, який сформовано за правилом: якщо $Ai + Bi \le 0$, то $Cj = Bi$.

7. Задано масив A з $N = 5$ елементів. Вивести кількість елементів масиву A, які задовольняють умову $L \le Ai < M$, де $L = –3$ та $M = 15$.

8. Задано масив A з $N = 6$ елементів. Вивести кількість елементів масиву A, які задовольняють умову $L < Ai < M$, де $L = 4$ та $M = 18$.

9. Задано масив A з $N = 7$ елементів. Вивести кількість елементів масиву A, які задовольняють умову $L \le Ai \le M$, де $L = –5$ та $M = 19$.

10. Задано масив A з $N = 8$ елементів. Вивести кількість елементів масиву A, які задовольняють умову $L \ge Ai > M$, де $L = 6$ та $M = 20$.

Зміст звіту

1. Постановка задачі для конкретного варіанта завдання.

2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.

3. Лістинг програми та коментарі до всіх команд.

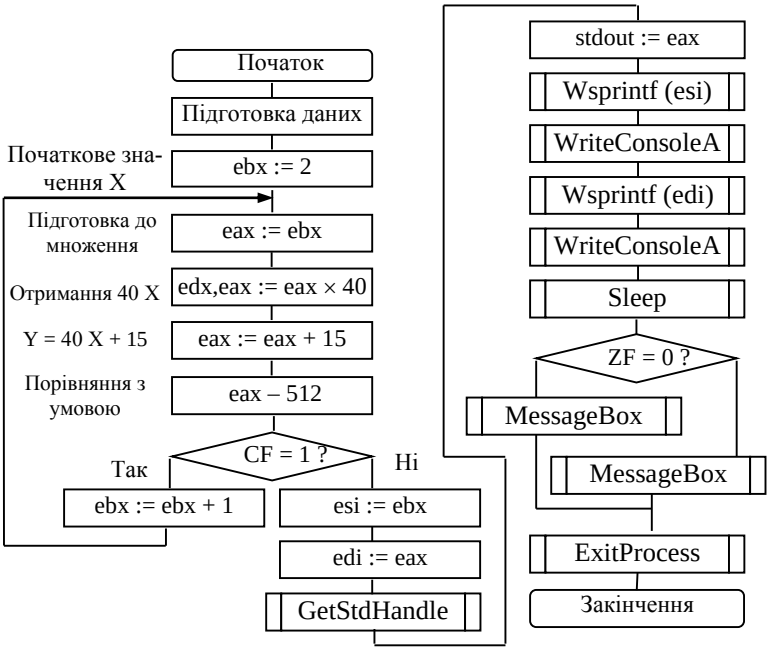
4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання.

5. Короткий опис виконання програми.

6. Висновки за результатами роботи.

Приклад 9.1. Для функції $Y = 40X + 15$ одержати перше значення, що перевищує 512, починаючи з $X = 2$. Значення аргументу та функції вивести на екран.

Алгоритм розв'язання прикладу 9.1 наведено на рисунку.



Лістинг 9.12. Програма виконання прикладу 9.1:

```
        title CopyRight by Rysovaniy A. N.
        .686                    ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none     ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include    \masm32\include\\user32.inc   ; файлы  интерфейса …
        include \masm32\include\kernel32.inc ; системные функции
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
BSIZE equ 15                    ; задание реального количества байтов
.data
        x dd 40          ; резервування 32-розрядної комірки пам'яті для x = 40
        y dd ?        ; резервування 32-розрядної комірки пам'яті для змінної Y
buf db BSIZE dup(?)          ; резервування пам'яті для буфера
frmt db "%d",0               ; задання перетворення одного символу
stdout  DWORD  ?             ; резервування в пам'яті 32-розрядної комірки
cWritten DWORD ? ; комірка пам'яті для адреси символів виведення
    st1 db "       y = 40X + 15",0
    st2 db "Автор: Рысованый А.Н.",0
    st3 db " ЛР Организация ввода-вывода",0
    st4 db "Программа выполнена успешно",0
    st6 db "До следующей лабораторной работы !!!",0
```

```
.code                                   ; початок сегмента  даних
start:                                  ; метка начала программы
   mov ebx,2                            ; початкове значення аргументу X
m1: mov eax,ebx              ; підготування до множення
   mul x                                ; виконання частини функції Y (40X)
   add eax,15                           ; виконання всієї функції Y = 40X + 15
   cmp eax,512                          ; порівняння з умовою
   jnc exit                             ; перейти, якщо Y > 128
   inc ebx                              ; збільшити лічильник аргументу X
   jmp m1                               ; перейти при  Y < 128
exit:  mov esi,ebx           ; збереження X
       mov edi,eax           ; збереження Y
invoke GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
  mov stdout, eax    ; збереження одержаного дескриптора у пам'яті
invoke  wsprintf, ADDR buf, \ ; адреса буфера, куди буде записана послідовність символів
     ADDR frmt, \     ; адреса рядка перетворення формату
       esi            ; регістр, вміст якого перетворюється
invoke  WriteConsoleA, stdout, \  ; дескриптор стандартного пристрою виведення
ADDR buf, BSIZE, ADDR cWritten, 0 ; розмір повідомлення та адреса для збереження
invoke wsprintf, ADDR buf, ADDR frmt,edi  ; функція перетворення edi
invoke  WriteConsoleA, stdout,  ADDR buf, \ ; адреса початку повідомлення
   BSIZE,  \     ; розмір повідомлення
    ADDR cWritten, 0   ; адреса, де зберігається кількість символів
invoke Sleep, 2000d    ; API-функція затримки зображення
invoke MessageBox,0 \ ; API-функція виведення вікна
  addr st2, \            ; адреса рядка, який містить  текст повідомлення
   addr st1, MB_OKCANCEL  ; адреса заголовка та вигляд вікна
       xor eax,1
       jz a1
invoke MessageBox, NULL, addr st4, addr st3, MB_ICONINFORMATION
       jmp a2
a1:
invoke MessageBox, NULL, addr st6, addr st3, MB_OK
a2:
invoke  ExitProcess, 0 ; возвращение управления ОС та вивільнення ресурсів
        end start              ; директива завершение программы с именем start
```

Наприкінці програми показано використання API-функції MessageBox. Останній її параметр повертає до регістра eax або 1 (якщо OK), або 2 (якщо CANCEL). За допомогою команди xor (додавання за mod2), якщо значення регістра eax дорівнювало 1, прапорець ZF установиться в 1. У такому випадку виводиться одне повідомлення, у іншому випадку – друге повідомлення.

Ліворуч наведено змінну *X*, а праворуч – *Y*. Є декілька варіантів підпису. Найбільш простим є варіант, при якому можна використати директиви

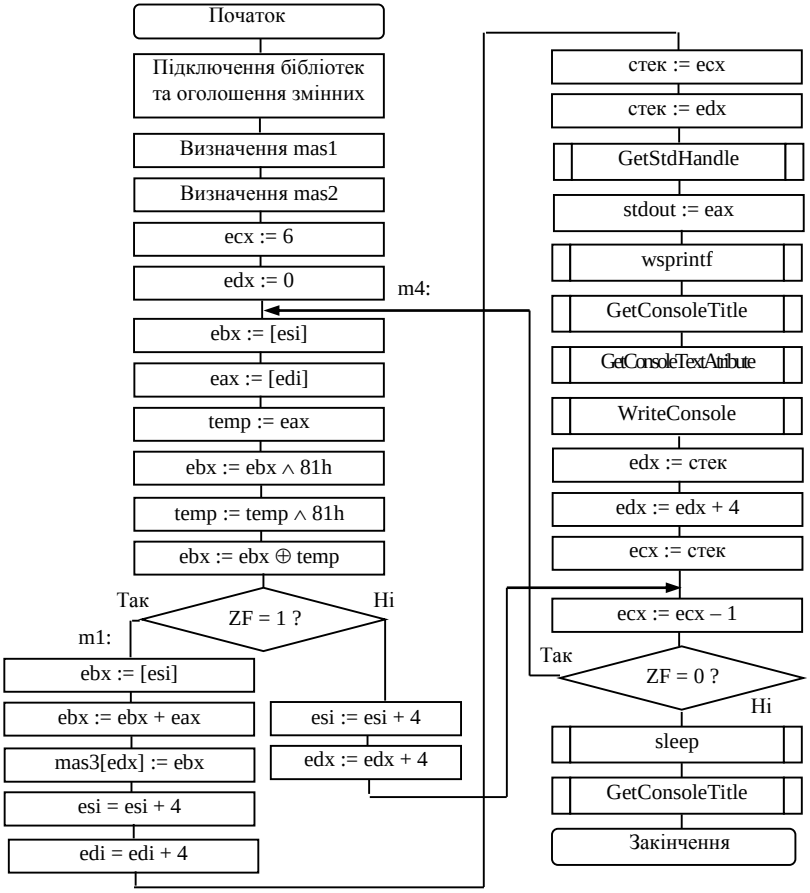
```
frmt1 db "x = %d",0
frmt2 db "y = %d",0
```

На рис. 9.14 наведено результат виконання лістингу 9.12.

Рис. 9.14. Результат виконання лістингу 9.12

Приклад 9.2. Задано масиви A і B з $N = 6$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо у елементів A_i та B_i біти 0 та 7 збігаються, то $C_i = A_i + B_i$.

Алгоритм розв'язання прикладу 9.2 наведено на рисунку.



Особливістю алгоритмів із трьома та більше масивами є те, що для таких задач не достатньо регістрів загального призначення.

Цю нестачу можна виправити використанням непрямої адресації та (чи) комірок пам'яті для тимчасового збереження даних. У програмі для запису даних у третій масив використовується змінна в пам'яті mas3 зі зміщенням, що розташоване в регістрі edx.

Вирівнювання адрес комірок у масивах виконується за допомогою команд

```
esi = esi + 4;
edi = edi + 4;
edx := edx + 4.
```

Лістинг 9.13. Програма виконання прикладу 9.2:

```
        title CopyRight by Rysovaniy A. N.
; масиви A і B з N = 6 елементів. Сформувати масив C:
; якщо у  Ai та Bi  біти 0 та 7 збігаються, то Ci = Ai + Bi.
        .686                            ; директива определения типа микропроцессора
.model  flat, stdcall  ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none             ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include    \masm32\include\\user32.inc   ; файлы  интерфейса …
include \masm32\include\kernel32.inc ; системные функции
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
        BSIZE equ 15                            ; задание реального количества байтов
.data                                           ; директива определения данных
mas1 DD 0, 81h, 33, 81h, 0FFh, 5   ; резервування пам’яті для mas1
mas2 DD 0, 81h, 0, 99h, 0FFh, 6    ; резервування пам’яті для mas2
temp dd ?                                       ; резервування пам’яті для змінної temp
mas3 dd 6 dup(0) ;
stdout DWORD ?                  ; резервування в пам’яті 32-розрядної комірки
buf db BSIZE dup(?)                     ; резервування пам’яті для буфера
frmt db "%d",0                          ; задання перетворення одного символу
cWritten DWORD ?    ; резервування для адреси символів виведення
mytitle     db "laboratory work", 0
color  dd  40h or 20h or 80h ; фон червоний + зелений +  підвищеної яскравості
.code                                                   ; начало сегмента данных
   start:                                               ; метка начала программы
        lea esi,mas1                ; записати адресу початку масиву mas1
        lea edi,mas2                ; записати адресу початку масиву mas2
        mov ecx,6                                       ; лічильник кількості пар чисел
        xor edx,edx                 ; початкове значення зміщення в масиві mas3
m4:  mov ebx, [esi]                     ; переслати число з масиву mas1
        mov eax,[edi]                   ; переслати число з масиву mas2
        mov temp,eax                ; перезбереження для подальшого додавання
        and ebx,81h             ; логічне множення числа з mas1 на маску
        and temp,81h                    ; логічне множення числа з mas2 на маску
        xor ebx,temp                ; mas1i ⊕ mas2i
        jz m1                           ; перейти, якщо 0
    add esi,4           ; підготування до читання наступного елемента з mas1
    add edi,4           ; підготування до читання наступного елемента з mas2
        jmp m3 ;
m1:  mov ebx,[esi]              ; читання елемента з mas1
         add ebx,eax            ; формування елемента mas3
         mov mas3[edx],ebx              ; запис в mas3
    add esi,4           ; підготування до читання наступного елемента з mas1
    add edi,4           ; підготування до читання наступного елемента з mas2
        push ecx                        ; підготовка до виклику API-функції
        push edx                        ; підготовка до виклику API-функції
invoke GetStdHandle, STD_OUTPUT_HANDLE          ; отримання дескриптора
    mov stdout, eax             ; збереження одержаного дескриптора у пам’яті
 invoke  wsprintf, \        ; API-функція перетворення числа
  ADDR buf, \ ; адреса буфера, куди буде записано послідовність символів
```

```
  ADDR frmt, \    ; адреса рядка перетворення формату
    mas3[edx]       ; комірка пам'яті, вміст якої перетворюється
invoke SetConsoleTitle, offset mytitle
invoke SetConsoleTextAttribute, stdout, color
invoke  WriteConsoleA,stdout, \     ; дескриптор стандартного пристрою виведення
 ADDR buf, BSIZE \ ; адреса початку повідомлення та розмір повідомлення
     ADDR cWritten, 0     ; адреса, де зберігається кількість символів
        pop edx                    ; встановлення edx
         add edx,4                 ; підготування зміщення для запису в mas3
         pop ecx                   ; встановлення ecx
m3:
        dec ecx                    ; зменшення лічильника пар чисел
        jnz m4                  ; перейти, якщо в лічильнику не нуль
invoke Sleep, 2000d                ; API-функція затримки зображення
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start               ; директива завершение программы с именем start
```

Результат виконання програми наведено у вікні налагоджувача OllyDbg на рис. 9.15.

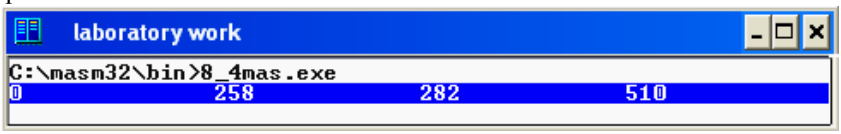


Рис. 9.15. Результат виконання програми з лістингу 9.13

Програма виконує виведення рядка результату у консоль, яка має свої колір фону та назву. Індивідуальну назву консолі забезпечує API-функція SetConsoleTitle. Колір фону задається API-функцією SetConsoleTextAttribute.

Приклад 9.3. Задано масив A з $N = 4$ елементів. Навести програму визначення суми елементів масиву A, для яких біти 0 та 5 збігаються.

Лістинг 9.14. Програма виконання прикладу 9.3:

```
        title CopyRight by Rysovaniy A. N.
        .686      ; директива определения типа микропроцессора
.model flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none     ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include    \masm32\include\\user32.inc   ; файлы  интерфейса …
include \masm32\include\kernel32.inc ; системные функции
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
        BSIZE equ 15                ; задание реального количества байтов
.data                               ; директива определения данных
        buf db 10 dup(?),0   ; місце у вікні для виведення повідомлення
ifmt db "Задано массив из 4-х элементов:",0dh,0ah,\
 "10   12   13   14",0dh,0ah,0ah,\
  "Сумма элементов массива, для которых биты 0 и 5 совпадают: = %d",0dh,0ah,0ah,\
   "Автор программы:  Рысованый А.Н., НТУ ХПИ",0
        titl1 db "Сумма элементов главной диагонали",0 ; назва віконця
```

```
        mas1 dw 10, 12, 13, 14                  ; масив mas1 слів
        len equ ($-mas1)/ 2     ; обчислення кількості  слів в mas1
        sum dw 0                        ; ячейка для хранения результата
.code                                   ; початок сегмента-даних
 _st:                                   ; метка начала программы
        mov ecx,len                     ; кількість байтів
        lea esi,mas1                ; початкова адреса масиву mas1
m1:   mov ax,[esi]                  ; в ах заноситься елемент масиву
        bt ax,0                         ; вибір нульового біта
        setc bh                         ; якщо cf=1, то встановлення 1 в bh
        bt ax,5                         ; вибір п'ятого біта
        setc bl                         ; якщо cf = 1, то встановлення 1 в bl
        cmp bh,bl                       ; порівняння бітів
        jne m2                          ; якщо не дорівнює, то перейти на m2
        add sum,ax      ; додавання вибраних елементів масиву
m2:  add esi,2          ; збільшення адреси mas1 для вибірки нового числа
        dec ecx         ; зменшення лічильника чисел у масиві mas1
        jnz m1                          ; перейти на мітку m1 , якщо не нуль
        movzx edx,sum  ; збереження результату з розширенням розрядності
invoke wsprintf,ADDR buf,ADDR ifmt,edx              ; функція перетворення edx
invoke MessageBox, NULL, ADDR buf, ADDR titl1, MB_OK
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end  _st                ; директива завершение программы с именем_st
```

Рядок len equ ($-mas1)/2 доцільніше записувати як

len equ ($-mas1)/**type** mas1,

щоб самостійно не розраховувати кількість байтів, які вміщуються у той або інший формат даних. Наприклад, якщо це dw, то кількість байтів у цьому форматі – 2; якщо це dd, то кількість байтів у цьому форматі – 4 й т.ін. Оператор **type** сам підставляє необхідну кількість байтів.

Результат виконання програми з лістингу 9.15 наведено на рис. 9.16.

Рис. 9.16. Результат виконання програми з лістингу 9.14

Результатом виконання програми є сума трьох чисел: 10, 12, 14, для яких біти 0 та 5 збігаються.

Приклад 9.4. Помножити 2 числа. Якщо нульовий біт результату операції дорівнює 0, то додати ці 2 числа.

Лістинг 9.15. Програма виконання прикладу 9.4:

```
 .686 ; директива определения команд микропроцессора
 .model flat,stdcall
option casemap:none
```

```
   include \masm32\include\windows.inc ; файлы структур, констант …
   include \masm32\include\kernel32.inc ; системные функции
   include \masm32\include\\user32.inc   ; файлы  интерфейса …
   includelib \masm32\lib\user32.lib
   includelib \masm32\lib\kernel32.lib
.data
        stl db "Результат перемножения двух чисел",0
        st2 dw ?,0; Буфер виведення повідомлення
ifmt db "Задание: перемножить два числа.",0dh,0ah,\
"Если нулевой бит результата операции равен 0, то сложить эти два числа.",0dh,0ah,0ah,\
"Ответ:",0dh,0ah,\
"a = 2",0dh,0ah,\
"b = 125",0dh,0ah,\
"Результат (a x b) = %d",0;
        _a dw 2
        _b dw 125
        _res dd ?,0
.code
_st:
        clc ; скинути прапорець перенесення CF
        xor edx,edx
        mov ax, _a  ; ax=_a
        mov bx, _b  ; bx=_b
        mul _b      ; dx:ax=_a*_b
bt ax, 0  ; виділення нульового біта, встановлення CF у це значення
        JNC ml  ; перейти, якщо CF=0
        JMP k  ;
ml:  mov ax, _a
        add ax, _b
k:  movzx eax,ax
        mov _res,eax
invoke wsprintf, ADDR st2,ADDR ifmt,_res; API-функція перетворення числа
        invoke MessageBox, 0,addr st2,addr stl,MB_OK
invoke ExitProcess, 0 ; возвращение управления ОС та вивільнення ресурсів
        end _st
```

Результат виконання програми з лістингу 9.15 наведено на рис. 9.17.

Рис. 9.17. Результат виконання програми з лістингу 9.15

Приклад 9.5. Задано масив *A* з N = 100 елементів (констант типу WORD). Сформувати масив з індексів елементів масиву *A*, у яких 1, 3 і 5 біти містять одиниці.

Для налагодження програми зручно використовувати невеликий масив із конкретними даними. Після налагодження програми масив може бути визначений декількома варіантами.

Варіант1:

```
  mas1 dw 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
```

21,22,23,24,25,26,27,28,29,30, й так далі до 99.

Варіант2:

```
buf dw 10 dup (1)
    dw 20 dup (0)
    dw 70 dup (255),0        ; буфер виведення повідомлення
```

Лістинг 9.16. Програма виконання прикладу 9.5:

```
    title Рысованый А.Н.                    rysov@rambler.ru
    .686            ; директива определения команд микропроцессора
    .model  flat,stdcall ; задание линейной модели памяти
    option casemap:none ; відмінність рядкових та великих літер
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\kernel32.inc ; системные функции применений…
    include \masm32\include\\user32.inc ; файлы  интерфейса …
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
.data                               ; директива определения данных
  mas1 dw 10,42,58,6,7,8 ; резервування пам'яті з іменем mas1
            ; комірок розміром у байт для чисел цього масиву
len equ ($-mas1)/ type mas1   ; обчислення кількості слів масиву mas1
    memRes dd 6 dup(0),0
    _title db "Результаты решения программы",0
    info db "Задан массив A из N = 6 элементов.",0Ah,0Dh,
"Сформировать массив B из индексов элементов массива A, ", 0Ah, 0Dh,
    "у которых 1,3 и 5 биты содержат единицы.",0Ah,0Dh,0Dh
    buf dd len dup (?),0                ; буфер виведення повідомлення
    ifmt  db "Ответ: mas3 =  %d  %d  %d  %d  %d  %d",0
.code                           ; директива начала сегмента команд
_start:                                 ; метка начала программы
    mov ecx, len     ; лічильник кількості байтів у масиві mas1
    mov edx, 0                  ; лічильник кількості елементів mas1[5,3,1] = 1
    mov ebx,0   ; комірка для індексування елементів вихідного масиву
      lea esi, mas1             ; занесення адреси масиву mas1 в esi
begin: mov eax, [esi]           ; занесення елемента  mas1  в  eax
      bt eax,1                  ; виділення 1-го біта в eax
      jc a1                     ; якщо CF = 1, то перейти на a1
      jmp _exit                 ; а якщо CF = 0, то перейти на мітку _exit
a1:   bt eax,3                  ; виділення 3-го біта в eax
      jc a2                     ; якщо CF = 1, то перейти на a2
      jmp _exit                 ; а якщо CF = 0, то перейти на мітку _exit
a2:   bt eax,5                  ; виділення 5-го біта в eax
      jnc _exit                 ; якщо CF = 0, то перейти на мітку _exit
     mov memRes[ebx],edx   ; переписати номер елемента в пам'ять
      add ebx,4 ; підготовка адреси для запису нового номера елемента
     inc edx            ; збільшення лічильника елементів mas1[0] = 0
_exit:  add esi,2          ; підготовка адреси для зчитування нового елемента
      dec ecx             ; перевірка лічильника кількості байтів у масиві mas1
      jnz begin           ; якщо в ecx не нуль, то перейти  до наступного циклу
invoke  wsprintf, ADDR buf, \  ; адреса буф., куди буде записано послідовність символів
   ADDR ifmt, \         ; адреса рядка перетворення формату
 memRes[0],memRes[4],memRes[8],memRes[12],memRes[16],memRes[20] ;
invoke MessageBox, 0, addr info, addr _title, MB_ICONINFORMATION
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
    end  _start         ; директива завершение программы с именем _start
```

Результат виконання програми з лістингу 9.16 наведено на рис. 9.18.

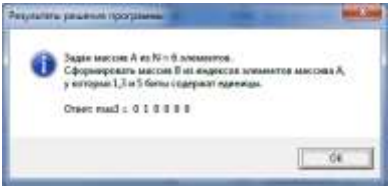


Рис. 9.18. Результат виконання програми з лістингу 9.16

Як лічильник елементів, у яких біти 1, 3 та 5 дорівнюють одиницям, використано регістр edx. Через те, що функція wsprintf використовує 32-розрядні дані в комірках пам'яті memRes, то для розміщення номерів елементів у пам'яті відбувається збільшення адреси на 4 байти командою

```
add ebx,4
```

Для виведення результату зарезервовано 6 комірок. Однак при наведених даних тільки числа 42 та 58 (перший та другий елементи) мають одиниці в 1, 3 та 5 розрядах.

Завдання для практичного виконання

При виконанні практичного завдання використати спрощене віконце з виведенням відомостей про автора програми:

– прізвище, ім'я та по батькові;

– навчальну групу;

– e-mail;

– результат операції,

які розташувати вертикально. Для виведення результатів використати функцію MessageBox.

Варіанти завдань:

1. Задано два масиви по 4 числа кожен. Знайти менші числа в однакових парах масивів та організувати їх виведення.

2. Задано два масиви по 5 чисел кожен. Виконати операцію логічного додавання над однойменними елементами масивів та організувати їх виведення.

3. Задано два масиви по 6 чисел кожен. Виконати операцію логічного додавання за модулем над однойменними елементами масивів та організувати їх виведення.

4. Задано два масиви по 7 чисел кожен. В однойменних парах елементів масивів знайти більші числа, виконати віднімання з них меншого числа та організувати їх виведення.

5. Задано два масиви по 8 чисел кожен. В однойменних парах елементів масивів знайти нульові числа та організувати їх виведення.

6. Задано два масиви по 4 числа кожен. Виконати операцію перемноження однойменних елементів масивів та організувати їх виведення.

7. Задано два масиви по 5 чисел кожен. Над однойменними елементами масивів виконати операцію ділення більшого числа на менше та організувати виведення цілої частини результату.

8. Задано два масиви по 6 чисел кожен. Над однойменними елементами масивів виконати операцію ділення більшого числа на менше та організувати виведення залишкової частини результату.

9. Задано два масиви по 7 чисел кожен. Знайти більші числа в однакових парах масивів та організувати їх виведення.

10. Задано два масиви по 8 чисел кожен. В однойменних парах елементів масивів знайти однакові числа та організувати їх виведення.

11. Відсортувати за збільшенням елементи масиву чисел, кратні 3.

Контрольні запитання

1. Що собою являє дескриптор консолі?
2. Які функції Win32 API використовуються для виведення повідомлень?
3. Які функції Win32 API використовуються для виведення звуку на системний динамік?
4. Яка функція Win32 API використовується для перетворення 32-розрядного числа в послідовність символів?
5. Яка функція Win32 API створює спрощене віконце та які її параметри?

10. ДИНАМІЧНІ БІБЛІОТЕКИ

Динамічні бібліотеки, що підключаються (Dynamic Link Libraries, DLLs) є сховищем загальнодоступних процедур. Механізм DLL-библиотек з'явився разом з операційною системою Windows і є її невід'ємною частиною. **Суть цього механізму** в тому, що в процесі компоновки виконуваного модуля з використанням зовнішніх процедур в нього розміщуються не самі процедури, а тільки їх назви (номери) разом з назвами DLL-библиотек, в яких вони містяться.

Windows не завантажує декілька копій DLL в пам'ять при одночасному виконанні декілька екземплярів програми. У реальності, у всіх процесів, що використовують одну і ту ж dll є своя копія цієї бібліотеки, проте Windows робить так, щоб всі процеси розділяли один і той же код цієї dll. Проте, секція даних копіюється для кожного процесу.

З погляду програміста – DLL є бібліотекою функцій (ресурсів), якими може користуватися будь-який процес, що завантажив цю бібліотеку. Саме завантаження віднімає час і збільшує витрату споживаної додатком пам'яті, тому бездумне дроблення одного застосування на безліч DLL не приводить до поліпшення програми.

Інша справа – якщо якісь функції використовуються декількома застосуваннями. Тоді, помістивши їх в одну DLL можна позбавитися від дублювання коду і збільшення об'єму додатків.

Процесу, що завантажив, DLL доступні не всі її функції, а що лише явно надаються самій DLL – тобто що експортуються. Чим більше функцій експортує DLL – тим повільніше вона завантажується.

При розробці DLL передбачалося, що введення DLL дозволить ефективно організувати пам'ять і дисковий простір, використовуючи тільки один екземпляр бібліотечного модуля для різних застосувань. Це було особливо важливо для ранніх версій Microsoft Windows з жорсткими обмеженнями по пам'яті.

Повних переваг від впровадження DLL отримати не вдалося унаслідок явища, званого DLL hell («ад DLL»). DLL hell виникає, коли декілька застосувань вимагають одночасно різні, не повністю сумісні версії бібліотек, що приводить до збоїв в цих застосуваннях і до конфліктів типу DLL hell, різко знижуючи загальну надійність операційних систем. Пізні версії Microsoft Windows почали вирішувати паралельне використання різних версій DLL (технологія SxS), що звело нанівець переваги початкового принципу модульності.

Таким чином, застосування динамічних бібліотек дозволяє добитися ряду переваг:

– зменшується розмір виконуваного файлу додатку та займані ним ресурси;

– функції DLL можуть використовувати декілька процесів одночасно;

– управління динамічними бібліотеками покладається на операційну систему;

– внесення змін в DLL не вимагає перекомпіляції всього проекту;

– одні DLL можуть використовувати програми, написані на різних мовах.

10.1. Створення динамічних бібліотек та їх використання

Бібліотека (dynamic-link library) (DLL), що динамічно підключається – це модуль, що містить функції і дані, які можуть бути використані іншим модулем (додатком або DLL).

В Windows є три найважливіші DLL-бібліотеки:

– kernel32.dll (управління пам'яттю, процесами й потоками);

– user32.dll (підтримка призначеного для користувача інтерфейсу, зокрема функцій, пов'язаних із створенням вікон і передачею повідомлень);

– gdi32.dll (графіка й виведення тексту).

Крім того, у Windows є й інші DLL, функції яких призначені для більш спеціалізованих завдань. Наприклад, в AdvAPI32.dll містяться функції для захисту об'єктів, роботи з реєстром і реєстрації подій, в ComDlg32.dll – стандартні діалогові вікна (ніби File Open та File Save), а ComCtl32 dll підтримує стандартні елементи управління.

DLL може визначити два види функцій: експортні та внутрішні. Експортні функції призначаються для виклику іншими модулями та усередині DLL. Внутрішні функції призначаються для виклику усередині DLL. Однак можна використовувати різні функції для читання або запису даних.

Операційна система Windows і всі її драйвери (окрім віртуальних), а також інші розширення в деякому розумінні є не чим іншим, як набором бібліотек динамічного компонування. Жодне крупне застосування Windows не обходиться без власних бібліотек динамічного компонування, і жодне застосування не може обійтися без виклику функцій, розташованих в таких бібліотеках. Зокрема, всі функції програмного інтерфейсу Windows знаходяться саме в бібліотеках динамічного компонування DLL (Dynamic-link Libraries).

Для звернення до зовнішніх по відношенню до програм модулів операційної системи MS-DOS і модулів BIOS використовується механізм програмних переривань. У будь-який момент часу в оперативній і постійній пам'яті комп'ютера знаходиться весь код, необхідний для роботи як програми, так і самої операційної системи MS-DOS. Якщо ж програма є складною і не поміщається цілком в 640 Кбайт пам'яті, на допомогу приходять оверлеї, що дозволяють завантажувати в оперативну пам'ять тільки ті сегменти програми, які були потрібні для вирішення поточного завдання.

У середовищі мультизадачної операційної системи, такої, як Windows, OS/2 або UNIX, статичне компонування неефективне, оскільки воно призводить до неекономного використання дуже дефіцитного ресурсу – оперативної пам'яті.

При використанні динамічного компонування завантажувальний код декількох функцій об'єднується в окремі файли, що завантажуються в оперативну пам'ять в єдиному екземплярі. Програми, що працюють паралельно, викликають функції, завантажені в пам'ять з файлів бібліотек динамічного компонування, а не з файлів програм.

Механізм динамічного компонування використовується не тільки в системі Windows. Більш того, він був винайдений задовго до появи Windows.

Для того щоб були зрозумілі відмінності між застосуванням і DLL-бібліотекою, уточнимо такі поняття, як завдання (task), копія застосування (instance) і модуль (module).

Коли запускаються декілька копій одного застосування, Windows завантажує в оперативну пам'ять тільки одну копію коду та ресурсів. Для кожної копії застосування створюється окремий сегмент даних, стек і черга повідомлень.

Завдання (task) можна розглядати як сукупність віртуального процесора і ресурсів, таких, як стек, регістри процесора, глобальна пам'ять, черга повідомлень і так далі. В операційній системі Windows є база даних запущених завдань, в якій зберігається вся інформація, що стосується завдання, наприклад, ідентифікатор застосування, ідентифікатор модуля застосування і так далі.

Копія застосування (instance) є контекстом, в якому виконується модуль застосування. Ідентифікатор копії застосування, який отримується через параметр hinstance функції Winmain, є ідентифікатором сегмента даних DGROUP, використовуваного при виконанні програмного модуля.

Модуль застосування (module) складається із виконавчого коду та ресурсів. Усередині Windows є база даних модулів, в якій зберігається така інформація, як посилання на функції, що експортуються, розташування сегментів коду та сегментів, що містять ресурси. Оскільки модуль застосування не змінюється (сегменти коду та ресурсів не змінюються), всі запущені копії одного застосування можуть ними користуватися.

DLL-бібліотека також є модулем. Вона знаходиться в пам'яті в єдиному екземплярі, містить сегменти коду та ресурси, а також один сегмент даних. Можна сказати, що для DLL-бібліотеки створюється одна копія (instance), що складається тільки з сегмента даних, і один модуль, що складається з коду та ресурсів.

DLL-бібліотеки використовуються для економії пам'яті, оскільки всі запущені застосування можуть використовувати один модуль DLL-бібліотеки, не включаючи стандартні функції до складу своїх модулів.

За допомогою DLL-бібліотек можна організувати колективне використання ресурсів або даних, розташованих в сегменті даних бібліотеки.

Використання DLL-бібліотек підвищує модульність застосувань і самої операційної системи Windows. З погляду застосування DLL-бібліотека є не більш ніж набором функцій з тим або іншим інтерфейсом, а також, можливо, набором ресурсів. Внутрішня побудова і алгоритми роботи функцій, а також використовувані функціями структури даних повністю приховані від застосування. Тому при внесенні змін або удосконалень до DLL-бібліотеки немає необхідності виконувати повторне збирання застосувань (якщо не змінився інтерфейс або набір функцій, що входять в бібліотеку) (http://samoucka.ru/).

Для створення своєї динамічної бібліотеки з файлом myio.dll **без конкретної точки входу** необхідно виконати певні кроки.

Крок 1. В текстовому редакторі створимо основну програму з ім'ям myio.asm з міткою початку програми. В цій програмі необхідно підключити динамічну бібліотеку зі своїм ім'ям (**mydll.lib**). Така програма наведена в лістингу 10.1.

Лістинг 10.1: Основний файл myio.asm

```
.686                    ; директива определения типа микропроцессора
.model  flat,stdcall ; завдання лінійної моделі пам'яті  та угоди ОС Windows
 option casemap:none    ; отличие строчных и прописных букв
 includelib\masm32\lib\kernel32.lib
 includelib mydll.lib
 AddDD proto :DWORD, :DWORD      ; прототип приватної процедури
.code                            ; директива начала  сегмента данных
 start:                          ; метка начала программы с именем start
   invoke AddDD,5,4   ; виклик директиви AddDD з параметрами
   invoke ExitProcess,0
 end start               ; директива завершение программы с именем start
```

Крок 2. В текстовому редакторі створити файл та надати йому розширення .def, який зберігає всі функції, що викликаються основним файлом myio.asm. Файл mydll.def буде таким, як у лістингу 10.2.

Лістинг 10.2. Файл mydll.def:

```
LIBRARRY   mydll      ; ім'я програми, яка включається до бібліотеки
EXPORTS   AddDD       ; ім'я процедури, яка викликається
```

Крок 3. Створимо в текстовому редакторі програму з ім'ям mydll.asm, в якій розкриємо приватну процедуру AddDD. Така програма не має мітки початку та наведена в лістингу 10.3:

Лістинг 10.3. Програму з ім'ям mydll.asm:

```
.686          ; директива определения типа микропроцессора
.model  flat,stdcall ; задание линейной модели памяти
option casemap:none      ; отличие строчных и прописных букв
includelib\masm32\lib\kernel32.lib
include \masm32\include\kernel32.inc ; системные функции применений…
   AddDD proto :DWORD, :DWORD          ; прототип процедури
.code                                  ; директива начала  сегмента данных
   AddDD proc arg1:DWORD,arg2:DWORD
     mov eax, arg1                     ;
     add eax, arg2                     ;
     ret                    ; повернення управління  ОС
   AddDD endp               ; закінчення процедури з ім'ям AddDD
   end                      ; завершение программы с именем start
```

Крок 4. Створимо командний bat-файл для отримання **dll-файлу**, такого, як у лістингу 10.4. Для цього треба підключити асемблерний файл **без мітки початку** програми (наприклад mydll.asm).

Лістинг 10.4. Командний bat-файл з ім'ям, наприклад mydll.bat, для створення файлу dll:

```
ml.exe /c /coff mydll.asm
link.exe /DLL /DEF:mydll.def /NOENTRY mydll.obj
```

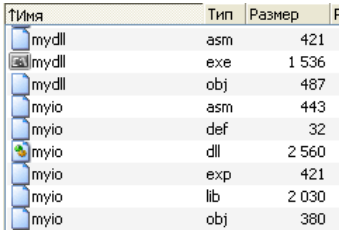
Виконаємо mydll.bat-файл. В результаті чого отримуються нові файли mydll**.dll**, mydll**.exp**, mydll**.lib** та mydll**.obj**.

Крок 5. Створимо ще один bat-файл з ім'ям, наприклад myio.bat, для створення виконуючого exe-файлу:

```
ml.exe /c /coff " myio.asm"
link.exe /subsystem:console " myio.obj"
```

В результаті **виконання** файлу myio.bat отримується файл myio.exe, який й підключає приватну динамічну бібліотеку.

Усі програми, які були створені та використовувались при викликах dll-функції, відображено у вікні файлового менеджера (рис. 10.1).



Имя	Тип	Размер
mydll	asm	421
mydll	exe	1 536
mydll	obj	487
myio	asm	443
myio	def	32
myio	dll	2 560
myio	exp	421
myio	lib	2 030
myio	obj	380

Рис. 10.1. Програми, які використовувались при створені та виклику dll-функції

Процес виклику особистого dll-файлу та обчислення результату виконання програми наведено на рис. 10.2.

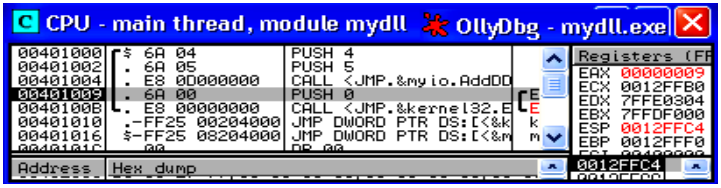


Рис. 10.2. Результат виконання програми з лістингу 10.4

Програма з лістингу 10.4 викликає процедуру, яка наведена у лістингу 10.3. Результат додавання 4 + 5 = 9 зберігається у регістрі EAX.

10.2. Функція точки входу в DLL

DLL може довільно визначити функцію точки входу. Функція Dllmain – додаткова точка входу в бібліотеку, що динамічно підключається (www.netcode.ru, http://www.i2r.ru/static/565/out_17611.shtml). Якщо функція використовується, то вона викликається системою тоді, коли процеси і потоки ініціалізовані та завершили роботу або при виклику функції LoadLibrary та FreeLibrary. Точка входу найчастіше має ім'я DllMain та синтаксис:

```
invoke DllMain,\ ;
  HINSTANCE hinstDLL,\          ; дескриптор модуля DLL
   DWORD fdwReason,\            ; причина виклику функції
    LPVOID lpvReserved          ; зарезервований
```

Базовий стиль параметра hinstDLL – це HINSTANCE.

Параметр fdwReason вказує причину виклику DLL. Коли fdwReason дорівнює DLL_PROCESS_ATTACH, то DLL завантажується у віртуальний адресний простір поточного процесу в результаті операції запуску або в результаті виклику LoadLibrary. При значенні DLL_THREAD_ATTACH поточний процес створює новий потік. При значенні DLL_THREAD_DETACH потік вивільнює пам'ять. А при DLL_PROCESS_DETACH DLL вивантажується з віртуального адресного простору зайнятого процесу в результаті невдалого завантаження DLL, завершення роботи процесу або виклику функції FreeLibrary.

Фрагмент програми з використанням точки входу в DLL-бібліотеку може бути таким:

```
AddDD proto :DWORD,:DWORD ; прототип
.data
hInst dd 0
.code
;   точка входу:
DllEntry proc instance:HINSTANCE, reason:DWORD, unused:DWORD
  mov eax, reason          ; проверка причины вызова
  cmp eax, DLL_PROCESS_ATTACH
; Если причина отличается от подсоединяемого процесса - выход
  jnz _exit
  push instance  ;
  pop hInst          ; обязательное сохранение хендла программы
_exit:
    and eax, 1
    ret
DllEntry endp   ; окончание процедуры точки входа
   AddDD proc arg1:DWORD,arg2:DWORD
   mov eax,arg1 ; eax =
   add eax,arg2 ; eax =
   ret        ; возвращение управления ОС
   AddDD endp   ; окончание процедуры с именем AddDD
   end ; окончание программы с именем start
```

У зв'язку з тим, що директива end вказує на ім'я директиви (DllEntry endp), то в файлі .def вже не треба використати директиву EXPORTS. Крім того, для компіляції бібліотеки не використовується ключ /NOENTRY (лістинг 10.5).

Лістинг 10.5. Командний файл для створення dll-бібліотеки з точкою входу:

```
ml /c /coff "myio.asm"
link /DLL /DEF:myio.def "myio.obj"
```

Приклад 10.1. Задано масиви A і B по $N = 9$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо $Ai \le Bi$, то $Ci = Ai + Bi$; інакше – $Ci = Ai - Bi$.

Крок 1. В текстовому редакторі створимо програму з ім'ям lab10.asm. В цій програмі визначаються масиви чисел _A, _B, _C. Сама програми складається з набору приватних (крім ExitProcess) функцій, які розташовані в окремому файлі (lab10_dll.asm). Основна программа наведена в лістингу 10.6.

Лістинг 10.6: Основна програма з ім'ям lab10.asm:

```
.686            ; директива определения типа микропроцессора
.model  flat,stdcall ; завдання лінійної моделі пам'яті  та угоди ОС Windows
option casemap:none         ; отличие строчных и прописных букв
include \masm32\include\\user32.inc   ; файлы  интерфейса …
include \masm32\include\kernel32.inc ; системные функции применений…
include lab10_dll.inc                          ; бібліотека константа
    includelib \masm32\lib\kernel32.lib
    includelib lab10_dll.lib   ; динамічна бібліотека, що підключається
.data
    _A dd 8, 7, 6, 5, 4, 3, 2, 1      ; масив з початковими даними
    _B dd 1, 2, 3, 4, 5, 6, 7, 8      ; масив з початковими даними
    _C dd 8 dup(0)                    ; масив для занесення результату
_str db "Массив _A:",9, 8 dup(" %d "),10, 13, "Массив _B:",9, 8 dup(" %d "),\
10, 13, "Массив _C:",9,8 dup(" %d "),0  ; перетворення ; 9 - символ табуляції
    .code                             ; директива начала  сегмента данных
_start:
        invoke arrCompare, addr _A, addr _B, addr _C     ;формування
invoke arr2str,addr _A, addr _B, addr _C, addr _str      ; конвертація в рядок
    invoke msgResult, addr _str       ; виведення вікна з результатами
      invoke msgAuthor                ; виведення відомостей про автора
      invoke ExitProcess, 0           ; коректний вихід з програми
    end _start                        ; завершение программы с именем start
```

Крок 2. В текстовому редакторі створимо файл з розширення .def. Цей файл lab10_dll.def буде таким, як у лістингу 10.7.

Лістинг 10.7. Файл lab10_dll.def:

```
LIBRARY lab10_dll      ; ім'я програми, яка має 4  бібліотеки
EXPORTS arrCompare         ; ім'я процедури, яка викликається
EXPORTS arr2str            ; ім'я процедури, яка викликається
EXPORTS msgResult          ; ім'я процедури, яка викликається
EXPORTS msgAuthor          ; ім'я процедури, яка викликається
```

Крок 3. Створемо в текстовому редакторі програму з ім'ям lab10_dll.asm, в якій розкриємо всі приватні процедури. Така програма наведена в лістингу 10.8:

Лістинг 10.8. Програму з ім'ям lab10_dll.asm:

```
.686            ; директива определения типа микропроцессора
.model  flat,stdcall ; завдання лінійної моделі пам'яті  та угоди ОС Windows
option casemap:none         ; отличие строчных и прописных букв
include \masm32\include\\user32.inc   ; файлы  интерфейса …
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\kernel32.inc ; системные функции применений…
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
    public arrCompare
    public arr2str
    public msgResult
```

```
        public msgAuthor
                                        ; функція порівняння двох масивів і заповнення третього:
  arrCompare proto pmass_A:DWORD, pmass_B:DWORD,\
    pmass_res:DWORD
                        ; Функція конвертації масиву чисел в символьний рядок. Аргументи –
                                ; адреси масивів і рядка, що містить шаблон для форматування
  arr2str proto pmass_A:DWORD,pmass_B:DWORD, pmass_C:DWORD, \
   pstr:DWORD ; функція виведення. Аргумент - рядок, що заздалегідь
                                                                ; відформатований для виводу
  msgResult proto pstr:DWORD
  msgAuthor proto      ; функція виведення відомостей про автора
  .data
  hInst dd 0
  _title_rez db "Результат:",0              ; заголовок вікна з результатом
  _title_auth db "Об авторе",0 ; заголовок вікна відомостей про автора. 9 - знак табуляції:
  _str_auth db "Имя:", 9 ,"Кутя Никита", 10, 13, "Группа:",9,"КИТ-28а",0
  .code
            ;   точка входу:
  DllEntry proc instance:HINSTANCE, reason:DWORD, unused:DWORD
        mov eax, reason                 ; перевірка причини виклику
        cmp eax, DLL_PROCESS_ATTACH
                                ; якщо причина відрізняється від під'єднуваного процесу - вихід
        jnz _exit
        push instance    ;
        pop hInst          ; обов'язкове збереження хендлу програми
  _exit:   and eax, 1
                ret
  DllEntry endp     ; закінчення процедури точки входу
                                        ; Тексти функцій:
  msgAuthor proc
     invoke MessageBox, 0, addr _str_auth, addr _title_auth, MB_OK
     ret
  msgAuthor endp
arrCompare proc pmass_A:DWORD, pmass_B:DWORD, pmass_res:DWORD
  ; збереження адрес аргументів функції
     mov esi, pmass_A
     mov edi, pmass_B
     mov edx, pmass_res
     mov ecx, 8       ; кількість елементів масиву - 8
  m2:
  ; порівняння масивів відбувається за допомогою регістрів:
     mov eax, dword ptr[esi]
     mov ebx, dword ptr[edi]
  ; формування елементів третього масиву:
     .IF eax<= ebx
     add eax, ebx
     .ELSE
     sub eax, ebx
     .ENDIF
     mov dword ptr[edx], eax       ; результат зберігається в пам'яті
  ; перехід до наступного кроку ітерації:
     add esi, 4
     add edi, 4
     add edx, 4
     loop m2
   ret
  arrCompare endp
  ; функція заповнення рядка з результатом. Використовуються останні два аргументи:
    arr2str proc pmass_A:DWORD,pmass_B:DWORD, pmass_C:DWORD, \
      pstr:DWORD
        mov ecx, 24  ; занесення загального числа елементів в трьох масивах
```

```
  ; перехід до останнього елемента  третього масиву:
     mov edi, pmass_C
     add edi,28 ; 8*4-4
  ; аргументи для wsprintf заносяться в стек з кінця:
  m1:        push dword ptr[edi]
     sub edi, 4
     loop m1
     mov esi, pstr ; після занесення чисел заносимо в стек решту аргументів
    push esi          ; використовується один і той же рядок
    push esi
   call wsprintf              ;  виклик wsprintf
     ret
   arr2str endp
     msgResult proc pstr:DWORD
        invoke MessageBox, 0, pstr, addr _title_rez, MB_OK
        ret
     msgResult endp
end DllEntry
```

Крок 4. Створимо командний bat-файл для отримання **dll-файлу**, такого, як у лістингу 10.9. Для цього треба використати асемблерний файл **без мітки початку** програми.

Лістинг 10.9. Командний bat-файл з ім'ям, наприклад lab10_dll.bat, для створення файлу dll:

```
ml.exe /c /coff "lab10_dll.asm"
link.exe /DLL /DEF:lab10_dll.def "lab10_dll.obj"
```

Виконаємо lab10_dll.bat-файл. В результаті чого отримуються нові файли: lab10_dll**.dll**, lab10_ dll**.exp**, lab10_ dll**.lib** та lab10_ dll**.obj**.

Крок 5. Створимо ще один bat-файл з ім'ям, наприклад lab10.bat, для створення виконуючого exe-файлу:

```
ml.exe /c /coff "lab10.asm"
link.exe /subsystem:console "lab10.obj"
```

В результаті **виконання** файлу lab10.bat отримується файл lab10.exe, який й підключає приватну динамічну бібліотеку. Результат виконання програми з лістингу 10.6 наведено на рис. 10.3.

Рис. 10.3. Результат виконання програми з лістингу 10.6

10.3. Лабораторна робота "Dll-файли"

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм з використанням **двох Dll-файлів з використанням точки входу**, написаних мовою асемблера для програмування МП платформи x86. Дані, які виводяться, необхідно підписати. Вивести довідку про автора програми, номер варіанту, повністю все завдання та своєї e-mail.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати свій варіант та написати програму на асемблері обчислення одного з виразів:

1. Задано масив A з $N = 100$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L < Ai \le M$, де $L = 2$ та $M = 10$.

2. Задано масив A з $N = 20$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L \le Ai < M$, де $L = 5$ та $M = 15$.

3. Задано масив A з $N = 30$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L < Ai < M$, де $L = 4$ та $M = 18$.

4. Задано масив A з $N = 40$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L \le Ai \le M$, де $L = 5$ та $M = 20$.

5. Задано масив A з $N = 50$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L \ge Ai > M$, де $L = 6$ та $M = 22$.

6. Задано масив A з $N = 60$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L > Ai \ge M$, де $L = 7$ та $M = 24$.

7. Задано масив A з $N = 70$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L > Ai > M$, де $L = 8$ та $M = 28$.

8. Задано масив A з $N = 80$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L \ge Ai \ge M$, де $L = 9$ та $M = 30$.

9. Задано масиви A і B по $N = 90$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо $Ai \le Bi$, то $Ci = Ai \times Bi$; інакше – $Ci = Ai + Bi$.

10. Задано масиви A і B по $N = 100$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо $Ai \le Bi$, то $Ci = Bi - Ai$; інакше – $Ci = Ai \times Bi$.

Зміст звіту

1. Постановка задачі для конкретного варіанта.

2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.

3. Лістинг програми та коментарі до всіх команд.

4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання.

5. Короткий опис виконання програми.

6. Висновки за результатами роботи.

Приклад 10.2. Задано масив A з $N = 10$ елементів. Навести алгоритм та програму визначення кількості елементів масиву A, які задовольняють умову $L \geq Ai > M$, де $L = 6$ та $M = 22$.

Лістинг 10.10. Основний файл myio.asm:

```
.686
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc          ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib user32,kernel32 ;
includelib \masm32\bin\mydll.lib
FindMas proto :BYTE,:BYTE,:BYTE,:BYTE  ;масив,L,M,N
BSIZE equ 30
.data
st1 db "DLL-файлы",0
masA db 5,16,24,8,1,3,36,26,25,33
st2 db "Исходный массив:",0ah
st3 db "5,16,24,8,1,3,36,26,25,33",0ah,0ah
st4 db "Условие: 6>=Ai>22",0ah,0ah
buf db BSIZE dup(?)
frmt db "количество элементов",0ah
st5 db "удовлетворяюших условие = %d",0ah,0ah
st6 db "автор:  КИТ",0
    N db 10
    L db 6
    M db 22
.code
_st:    invoke FindMas,masA,L,M,N
invoke wsprintf,ADDR buf,ADDR frmt, ebx
invoke MessageBox,0,ADDR st2,ADDR st1,MB_OK
invoke ExitProcess,0
end _st
```

Лістинг 10.11. Файл myio.def:

```
LIBRARRY   mydll       ; ім'я програми, яка включається до бібліотеки
EXPORTS   FindMas ; ім'я процедури, яка викликається
```

Лістинг 10.12. Файл mydll.bat

```
ml.exe /c /coff mydll.asm
link.exe /DLL /DEF:mydll.def "mydll.obj"
```

Лістинг 10.13. Програму з ім'ям mydll.asm:

```
.686
.model flat,stdcall
option casemap:none
include \masm32\include\windows.inc          ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib user32,kernel32
FindMas proto :BYTE,:BYTE,:BYTE,:BYTE
```

```
.data
        hInst dd 0
.code
        DllEntry proc instance:HINSTANCE, reason:DWORD, unused:DWORD
          mov eax, reason                ; перевірка причини виклику
          cmp eax, DLL_PROCESS_ATTACH
        ; Якщо причина відрізняється від під'єднуваного процесу - вихід
          jnz _exit
          push instance  ;
          pop hInst           ; обов'язкове збереження хендлу програми
        _exit:
           and eax, 1
           ret
        DllEntry endp

         FindMas proc arg1:BYTE,arg2:BYTE,arg3:BYTE,arg4:BYTE
             movzx ecx,arg4
             mov ebx,0
             mov edx, 0
             lea edi, arg1
        m1:
              mov al,[edi]
             .IF (al <= arg2)||(al > arg3)
                  add ebx, 1
                  add edi, 2
             .ELSE
                  add edi, 2
             .ENDIF
        loop m1
        ret
        FindMas endp
        end DllEntry
```

Лістинг 10.14. Файл myio.bat:

```
ml.exe /c /coff "myio.asm"
link.exe /subsystem:console "myio.obj"
```

Результат виконання програми з лістингу 10.6 наведено на рис. 10.4.

Рис. 10.4. Результат виконання програми

Контрольні запитання

1. Які переваги та недоліки використання динамічної бібліотеки?
2. Які кроки необхідно виконати для створення своєї динамічної бібліотеки?
3. Як створюється точка входу в динамічну бібліотеку?

11. ФАЙЛИ

При роботі з програмами часто виникає необхідність зберігати отримані результати на жорсткому диску комп'ютера, щоб надалі їх використовувати у розрахунках. Середовище Win32 здатне підтримувати декілька файлових систем. Ці системи повинні дотримуватися правила привласнення імен файлам і каталогам.

Функції API Win32, що мають відношення до роботи з файловою системою, наведено у табл. 11.1.

Таблиця 11.1

Функція	Призначення
AreFileApisANSI	Визначення набору символів файлу
Cancello	Відмінити всі очікувані процеси обробки операції введення і виведення (I/O)
CloseHandle	Закрити відкритий дескриптор файлу
CopyFile, CopyFileEx	Копіювання існуючого файлу в новий файл
CopyProgress-Routine	Визначення зворотного виклику. Використовується з функціями CopyFileEx та MoveFileWith-Progress. Викликається, коли завершується частина операції копіювання або пересилання
CreateDirectory, CreatedrectoryEx	Створити каталог
CreateFile	Створити файл або об'єкт спеціального типу
DefineDosDevice	Визначити, перевизначити або видалити ім'я пристрою MS DOS
DeleteFile	Видалити файл
FindClose	Закрити вказаний пошуковий дескриптор
FindClose-ChangeNotification	Закрити об'єкт-повідомлення про зміну файлу
FindFirstChange-Notification	Створити об'єкт-повідомлення про зміну файлу
FindFirstFile, Find-FirstFileEx, FindNextFile	Пошук файлів
FindNextChange-Notification	Скидання об'єкта-повідомлення в зайнятий стан

FlushFileBuffers	Очищення буфера для вказаного файлу і запис всіх буферізованих даних у файл
GetBinaryType	Визначити, чи є файл виконуваним і для якої підсистеми – Win32, MS DOS, OS/2, POSIX та ін.
GetCompressed-FileSize	Визначити реальні розміри ущільненого файлу
GetCurrent-Directory	Отримати поточний каталог
GetDiskFreeSpace, GetDiskFree-SpaceEx	Інформація щодо вказаного диска, включаючи кількість вільного простору на ньому
GetDriveType	Визначити тип диска – знімний, фіксований, CD-ROM, електронний або мережний
GetFileAttributes, GetFileAttributesEx	Отримати атрибути файлу або каталогу
GetFileInformationBy Handle	Знайти інформацію щодо вказаного файлу
GetFileSize,Get-FileSizeEx	Отримати розмір вказаного файлу
GetFileType	Отримати тип вказаного файлу
GetFullPathName	Отримати повний шлях і ім’я для вказаного файлу
GetLogicalDrives, GetLogicalDrive-Strings	Визначити доступні в даний час дисководи
GetLongPathName	Перетворити вказаний шлях до його довгої форми
GetShortPathName	Отримати псевдонім файлу
GetTempFileNam	Створити ім’я для тимчасового файлу
GetTempPath	Отримати шлях каталогу для тимчасових файлів
LockFile, LockFileEx	Блокування файлу
MoveFile, MoveFileEx, MoveFileWith-Progress	Перейменування файлу
QueryDosDevice	Отримати інформацію про імена пристрою MS DOS
ReadDirectory-ChangesW	Отримати інформацію про зміни в межах каталогу
ReadFile, ReadFileEx	Читання файлу з поточної позиції
ReadFileScatter	Читання даних із файлу зі збереженням їх у наборі буферів
RemoveDirectory	Видалити існуючий порожній каталог

SearchPath	Пошук вказаного файлу
SetCurrentDirectory	Встановити поточний каталог
SetEndOfFile	Помістити символ кінця файлу (EOF) в поточну позицію вказівника файлу
SetFileApisToANSI	Використовувати кодову сторінку набору символів ANSI при роботі функцій файлового введення-виведення
SetFileApisToOEM	Використовувати кодову сторінку набору символів при роботі функцій файлового введення-виведення
SetFileAttributes	Встановити атрибути файлу
SetFilePointer, SetFilePointerEx	Перемістити вказівник файлу в певну позицію
SetVolumeLabel	Задати мітку тому
UnlockFile, UnlockFileEx	Зняти блокування файлу
WriteFile, WriteFileEx	Запис у файл
WriteFileGather	Запис даних у файл із набору буферів

11.1. Створення файлу

Програма, наведена у лістингу 11.1, створює новий файл з іменем main та зберігає в ньому числа від 1 до 10, які отримуються в циклі програми.

В програмі після отримання числа одразу формується рядок перетворення за допомогою функції wsprintf. Майже всі функції API при своєму виконанні використовують регістри загального призначення, а тому, перед їх викликом ті регістри, які використовуються в програмі, повинні бути збережені.

Лістинг 11.1. Створення файлу з іменем main.txt:

```
; консольная!! Для создания exe-файла в masm32: Project/Console Assemble & Link
.686                    ; директива определения типа микропроцессора
.model  flat, stdcall   ; задание линейной модели памяти  и соглашения ОС Windows
option casemap:none             ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\kernel32.inc ; системные функции
include   \masm32\include\\user32.inc  ; файлы  интерфейса …
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
  BSIZE   equ 13          ; кількість байтів, які записуються у файл
.data                     ; директива определения данных
fmt  db "%d ",0           ; задання перетворення одного символу
szBuf   db BSIZE dup(?) ; резервування пам'яті для буфера
fName   BYTE "main.txt",0 ; комірки для ім'я файлу
fHandle DWORD ?   ; резервування в пам'яті 32-розрядної
                          ; комірки з ім'ям fHandle для дескриптора  збереження файлів
cWritten  DWORD  ?  ; резервування 32-розрядної комірки пам'яті
                                  ; з ім'ям cWritten для адреси символів виведення
```

```
.code                                   ; директива начала  сегмента данных
start:                                  ; метка начала программы с именем start
        mov edx,1           ; занесення первинного значення
        mov ecx,10          ; занесення максимального значення
  nxt:                  ; початок циклу
        push    ecx ; збереження перед викликом API-функції
        push    edx ; збереження перед викликом API-функції
invoke  wsprintf, ADDR szBuf[edx], ADDR fmt,edx ; перетворення
        pop     edx ; відновлення
        inc     edx  ; збільшення числа
        pop     ecx  ; відновлення
     loop    nxt  ; цикл
invoke CreateFile, ADDR fName,     ; адреса імені файлу з символами
 GENERIC_WRITE, 0,0,   ; запис у файл та параметри багатозадачності
   CREATE_ALWAYS,               ; знищити та створити новий файл
    FILE_ATTRIBUTE_ARCHIVE, 0
        mov fHandle, eax        ; запам'ятовування дескриптора файлу
invoke WriteFile, fHandle,   ; запис у файл за дескриптором пристрою
 ADDR szBuf,        ; адреса ділянки пам'яті, що зберігає символи
  BSIZE, ADDR cWritten, 0 ; кількість символів та адреса пам'яті  числа
        invoke  CloseHandle, fHandle  ; закрити дескриптор файлу
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start               ; директива окончания программы з іменем start
```

Функція **CreateFile** дозволяє створити новий файл або відкрити існуючий файл. Якщо операція закінчується успішно, в програму, що викликала її, повертається дескриптор відкритого файлу. Інакше повертається спеціальна константа Invalid_handle_value. Функція **CreateFile** має декілька параметрів.

Перший параметр містить адресу імені файлу, що складається з символів. Ознакою закінчення імені служить нульовий байт.

Другий параметр показує процедурі, ***для чого*** відкривається або створюється файл (режим доступу).

Значення 0 – це запит на доступ до пристрою як до файлу. При цьому прикладна програма може опитати атрибути пристрою без доступу до нього на фізичному рівні.

Значення GENERIC_READ – це запит на доступ до файлу при читанні. При цьому дані можуть бути зчитані з файлу за допомогою подальших викликів функції ReadFile. Зчитування даних із файлу викликає переміщення внутрішнього вказівника на величину лічильника прочитаних даних. Щоб дістати доступ до файлу щодо читання/запису додається в атрибути константа GENERIC_WRITE.

Значення GENERIC_WRITE – це запит на доступ до файлу при записуванні. Запис даних у файл виконується за допомогою подальших викликів функції WriteFile. При цьому змінюється значення внутрішнього вказівника позиції у файлі на величину лічильника записаних даних. Щоб дістати доступ до файлу щодо читання/запису, додається в атрибути константа GENERIC_READ.

Наприклад, параметр GENERIC_WRITE означає, що дозволений запис у файл, а параметр GENERIC_READ дозволяє тільки читання. Для читання і запису обидва параметри об'єднуються оператором *АБО*:

GENERIC_READ or GENERIC_WRITE

Третій параметр показує, чи може файл використовуватися іншими програмами в багатозадачному режимі.

Четвертий параметр теж має відношення до багатозадачності і містить адресу ділянки даних, в якій вказано, чи може файл використовуватися програмами, породженими даною програмою. Будь-яка програма в системі Windows може запускати інші програми.

П'ятий параметр показує, що робити, якщо файл вже існує.

Значення CREATE_NEW наказує створити новий файл. Якщо файл із вказаним ім'ям існує, функція аварійно завершує свою роботу.

Значення CREATE_ALWAYS наказує створити новий файл. Якщо файл із вказаним ім'ям існує, його вміст буде затертий. При цьому існуючі атрибути файлу скидаються. Потім функція об'єднує атрибути файлу і прапорців, вказані в параметрі з константою FILE_ATTRIBUTE_ARCHIVE.

Значення OPEN_EXISTING наказує відкрити існуючий файл. Якщо файл не знайдений, функція аварійно завершує свою роботу.

Значення OPEN_ALWAYS наказує відкрити існуючий файл. Якщо файл не знайдений, він буде створений відповідно до значення CREATE_NEW.

Значення TRUNCATE_EXISTING наказує відкрити існуючий файл щодо запису і його довжина зрізується до нуля. При цьому в запиті на доступ до файлу повинен бути вказаний атрибут GENERIC WRITE.

Якщо файл не знайдений, функція аварійно завершує свою роботу

Шостий параметр задає атрибут файлу: архівний, прихований, тільки для читання, системний і т.д.

Значення FILE_ATTRIBUTE_ARCHIVE наказує створити файл, що архівується. Застосування використовують значення цього атрибуту для відбору файлів для резервного копіювання або відновлення.

Значення FILE_ATTRIBUTE_HIDDEN наказує створити прихований файл. Подібні файли не включаються в список файлів при звичайному перегляді каталогу.

Значення FILE_ATTRIBUTE_NORMAL наказує не використовувати ніякі інші атрибути. Цей атрибут повинен використовуватися тільки сам по собі.

Значення FILE_ATTRIBUTE_READONLY наказує створити файл тільки для читання. Застосуванню дозволяється прочитувати дані з файлу, але забороняється змінювати в ньому дані і видаляти файл.

Значення FILE_ATTRIBUTE_TEMPORARY наказує використовувати дані для тимчасового зберігання.

Сьомий параметр дорівнює нулю.

Процедура CreateFile повертає в регістр eax *дескриптор* файлу, що використовується аналогічно до дескриптора екрана або клавіатури.

Функція **WriteFile** використовується для запису у файл символів, має ті самі параметри, що і функція WriteConsole: дескриптор файлу fHandle; адресу ділянки пам'яті, що зберігає символи; кількість символів BSIZE; адресу пам'яті, де зберігається кількість записаних у файл символів ADDR cWritten;

і, нарешті, нікому не потрібний нульовий вказівник NULL. Процедура WriteFile пише у файл байти.

Завершує роботу з файлом процедура **CloseHandle**, у якої всього один параметр – дескриптор файлу. Ця процедура від'єднує файл від дескриптора, і після її виконання файл знову повинен бути відкритий, щоб стали можливими читання або запис.

Після того як стандартними засобами отримано файл з ім'ям, наприклад 9_1L.exe, натискаємо на клавішу Enter та отримуємо файл з ім'ям main. Для того щоб відкрити цей файл, необхідно натиснути в оборочці FAR на клавішу F3.

11.2. Читання файлу

Для читання даних із файлу застосовують послідовність таких дій:
– відкрити існуючий файл (функція Createfile);
– зберегти дескриптор файлу;
– прочитати всі дані з файлу (функція ReadFile);
– перетворити дані в символи (функція wsPrintf);
– вивести символи на екран (функція WriteConsoleA);
– закрити раніше відкритий файл (функція CloseHandle).

Дані у файлі зберігаються в ASCII-кодах. Тому при ручному введенні даних у файл зручно використовувати HEX-редактори, наприклад програму xvi32. Вікно програми xvi32 HEX-редактора з ручним введенням даних наведено на рис. 11.1.

Програму, що читає дані і виводить числа на екран, наведено в лістингу 11.2.

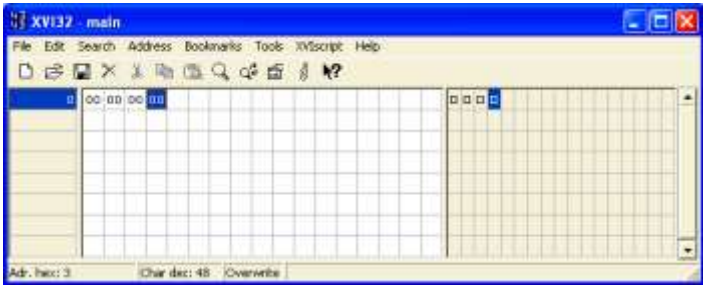


Рис. 11.1. Вікно програми xvi32 HEX-редактора

Лістинг 11.2. Програма читання даних із файлу та їх виведення на екран:

```
.686                    ; директива определения типа микропроцессора
.model  flat, stdcall ; задание линейной модели памяти  и соглашения ОС Windows
option casemap:none         ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\kernel32.inc ; системные функции
include    \masm32\include\\user32.inc ; файлы  интерфейса …
includelib \masm32\lib\user32.lib
 includelib \masm32\lib\kernel32.lib
```

```
      NOFDIG equ 1       ; кількість цифр
      BSIZE  equ 4            ; кількість байтів, які необхідно прочитати
        DIGSZ  equ 4    ; розмір буфера  для перетворення чисел у символи
.data                                   ; директива определения данных
  fName    BYTE "main",0      ; комірки для ім'я файлу
  fmt      BYTE "%d",0
  fHandle  DWORD ?             ; дескриптор файлу
  stdout   DWORD ? ;
cRead DWORD ?  ; комірка для змінної з реально прочитаними  символами
cWritten DWORD ?  ; кількість виведених байтів
  buf  DWORD BSIZE dup (?) ; буфер для збереження прочитаних символів
  bufOut   BYTE DIGSZ dup (?)  ;
.code                                           ; директива начала  сегмента данных
start:                                          ; метка начала программы с именем start
 invoke  GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
  mov     stdout, eax                   ; збереження дескриптора
invoke  CreateFile,\ ; функція відкриття створеного файлу
  ADDR fName,\       ; адреса імені файлу
   GENERIC_READ,\   ; запит на доступ до файлу при читанні
    0, NULL, OPEN_EXISTING,\      ; відкрити існуючий файл
     FILE_ATTRIBUTE_NORMAL, 0  ;
mov fHandle, eax            ; збереження дескриптора файлу
invoke ReadFile, fHandle \  ; функція читання файлу та дескриптор файлу
ADDR buf, BSIZE \ ; адр. буфера для символів та кількість символів
   ADDR cRead, NULL ; адреса для збереження числа  реально прочитаних символів
     mov     ecx, NOFDIG ;
     mov     esi, 0                 ;
nxt:
      push    ecx
invoke  wsprintf, ADDR bufOut, ADDR fmt, buf[esi]
invoke WriteConsoleA, stdout,\                  ; дескриптор пристрою стандартного виведення
    ADDR bufOut,\   ; адреса буфера для виведення
      DIGSZ,   ADDR cWritten, 0
          add     esi, 2
          pop     ecx
        loop    nxt
invoke  CloseHandle, fHandle  ; повернення дескриптора  (закриття файлу)
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start       ; директива окончания программы з іменем start
```

Кінець файлу досягається при порівнянні кількості прочитаних та кількості вказаних символів.

У програмі в стеку зберігається змінна в регістрі ecx. Це необхідно тому, що процедури Windows API використовують цей регістр для своєї мети. Регістр esi зберігає відносну адресу числа в масиві buf.

Результат читання файлу з ім'ям main наведено на рис. 11.2.

Рис. 11.2. Результат читання файлу з ім'ям main

У результаті читання файлу на екрані відображається одна цифра файлу main. Вміст цього файлу – нуль.

У випадку, якщо вказаного файлу немає (або в його імені є помилка), то програма може себе повести некоректно. В цьому випадку необхідно перевірити значення, яке повертає функція в регістр eax.

У випадку, якщо це значення INVALID_HANDLE_VALUE, то необхідно передбачити інші шляхи виконання програми. Таку програму з додатковими командами (виділеними жирним шрифтом) перевірки імені файлу наведено у лістингу 11.3.

Лістинг 11.3. Програма читання файлу з перевіркою його імені:

```
    .686              ; директива определения типа микропроцессора
    .model  flat, stdcall   ; задание линейной модели памяти и соглашения ОС Windows
    option casemap:none               ; отличие строчных и прописных букв
  include \masm32\include\windows.inc ; файлы структур, констант …
  include \masm32\include\kernel32.inc ; системные функции
  include    \masm32\include\\user32.inc ; файлы  интерфейса …
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
    NOFDIG equ 1  ; кількість цифр
    BSIZE  equ 4      ; кількість байтів, які необхідно прочитати
    DIGSZ  equ 4       ; розмір буфера для перетворення чисел у символи
.data                                       ; директива определения данных
    fName   BYTE "main",0  ; комірки для ім'я файлу
      fmt     BYTE "%d",0
    fHandle  DWORD ?          ; дескриптор файлу
    stdout   DWORD ? ;
    cRead DWORD ? ; комірка для змінної, з реально прочитаними символами
    cWritten DWORD ?     ; кількость виведених байтів
    buf DWORD BSIZE dup (?) ; буфер для збереження прочитаних символів
    bufOut   BYTE DIGSZ dup (?)  ;
  OpenError BYTE "ошибка в имени файла"
.code
 start:
  invoke  GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
    mov     stdout, eax   ; збереження дескриптора
  invoke  CreateFile, \  ; функція відкриття створеного файлу
       ADDR fName,\        ; адреса імені файлу
        GENERIC_READ,\       ; запит на доступ до файлу при читанні
          0, NULL, OPEN_EXISTING,\     ; відкрити існуючий файл
           FILE_ATTRIBUTE_NORMAL, 0  ;
  mov fHandle, eax                ; збереження дескриптора файлу
            cmp eax,INVALID_HANDLE_VALUE
            jz outEnd
invoke ReadFile, fHandle, \  ; функція читання файлу та  дескриптор файлу
ADDR buf, BSIZE,\ ; адр. буфера для символів та кільк. символів
 ADDR cRead, 0\ ; адр. змінної для збереження кількості символів
   mov     ecx, NOFDIG ;
   mov     esi, 0  ;
  nxt:
  push    ecx
invoke  wsprintf, ADDR bufOut,\ ; функція перетворення
     ADDR fmt, buf[esi]
invoke WriteConsoleA, \ ; функція виведення рядка символів на екран
  stdout,\              ; дескриптор пристрою стандартного виведення
    ADDR bufOut,\               ; адреса буфера для виведення
      DIGSZ, \
       ADDR cWritten, 0  ; адреса, де зберігається кількість символів
  add     esi, 2
```

```
  pop     ecx
  loop    nxt
invoke  CloseHandle, fHandle  ; повернення дескриптора  (закриття файлу)
        jmp m1
outEnd:
        invoke WriteConsole, stdout,\
          ADDR OpenError,\
           sizeof OpenError,\
            ADDR cWritten, 0
m1:
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start              ; директива окончания программы з іменем start
```

Результат виконання програми з лістингу 11.3 наведено на рис. 11.3.

Рис. 11.3. Результат виконання програми з лістингу 11.3

11.3. Переміщення файлового вказівника

За допомогою функції **SetFilePointer** можна організувати переміщення вказівника у відкритому файлі до довільних ділянок даних. Прототип функції такий:

```
SetFilePointer PROTO,   ; функція встановлення вказівника
    handle:DWORD,                ; дескриптор файлу
  nDistanceLo:SDWORD,            ; кількість байтів для переміщення
 pDistanceHi:PTR SDWORD,         ; адреса 32-розрядної змінної
                ; яка містить старше слово 64-розрядного числа байтів для переміщення
moveMethod:dword                 ; початкова точка для переміщення
```

Параметр moveMethod визначає відправну точку, відносно якої виконується переміщення вказівника. Він може набувати одного з трьох значень: FILE_BEGIN, FILE_CURRENT та FILE_END. Значення, що визначає кількість байтів для переміщення вказівника, є 64-розрядним числом із знаком, розділеним на 2 частини:

– nDistanceLo – молодші 32 біти (число зі знаком);

– pDistanceHi – адреса змінної, що містить старші 32 біти (число зі знаком).

Позиція вказівника – це число зі знаком, яке зберігається у подвійному слові. Від'ємне число позначає переміщення вказівника назад, а додатне – уперед. Для задання точки обліку використовують такі значення:

– FILE_BEGIN – від початку файлу;

– FILE_END – від кінця файлу;

– FILE_GURRENT – від поточної позиції вказівника.

Для того щоб вставити дані в початок файлу, необхідно виконати певні дії:

– відкрити файл (функція CreateFile);

– зберегти дескриптор файлу;

– поставити вказівник на нульовий байт (функція SetFilePointer);

– прочитати та зберегти в пам'яті (буфері) дані з файлу, починаючи з вказівника;

– поставити вказівник на нульовий байт (функція SetFilePointer);

– записати на вказане місце нові дані (функція WriteFile);

– повернути дескриптор та закрити файл (функція CloseHandle).

Якщо при виклику функції **SetFilePointer** параметр pDistanceHi дорівнює нулю, то для переміщення файлового вказівника використовуватиметься тільки значення параметра nDistanceLo.

Для подальшої операції запису даних у кінець файлу необхідно застосувати функцію SetFilePointer з параметрами:

```
invoke SetFilePointer
 fileHandle      ;  дескриптор файлу
  0              ;  молодше значення числа байт
   0             ;  вказівник на старше значення
    FILE_END     ;  переміщення відносно кінця файлу
```

Наприклад, потрібно дописати у файл з іменем main з початковими даними 0123 прізвище та ініціали. Варіант такої програми наведено у лістингу 11.4.

Лістинг 11.4. Програма з редагуванням файлу:

```
        .686          ; директива определения типа микропроцессора
        .model  flat, stdcall    ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none         ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\include\kernel32.inc ; системные функции
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
        BSIZE  equ 128
.data                                          ; директива определения данных
        fName    BYTE "main",0   ; комірки для ім'я файлу
        fHandle  DWORD ?
        cRead    DWORD ?
        cWrite   DWORD ?
        cWritten  DWORD ?
        cName    BYTE "Rysovaniy A.N."
        buf      BYTE BSIZE dup (?)
.code                                          ; директива початку сегмента  даних
 start:                                        ; метка начала программы с именем start
invoke CreateFile, ADDR fName,\ ; створеного файлу та адреса імені файлу
   GENERIC_READ+GENERIC_WRITE,  ;  читання та запис
      0, NULL, OPEN_EXISTING,
       FILE_ATTRIBUTE_NORMAL, 0
        mov     fHandle, eax   ; збереження дескриптора файлу
 invoke SetFilePointer, fHandle, ; встановлення вказівника файлу
      0,NULL,  ; кількість байтів для переміщення та адреса змінної
        FILE_BEGIN        ; початкова точка для переміщення з початку
 invoke  ReadFile, fHandle,  ; функція читання файлу та дескриптор файлу
 ADDR buf, BSIZE, ; адр. буф. для символів та кількість символів
 ADDR cRead, 0 ; адр. змінної для збереж. реально прочитаних символів
 invoke  SetFilePointer,         ;   функція встановлення вказівника
      fHandle, 0,0,FILE_BEGIN
```

```
invoke  WriteFile, fHandle, ; функція запису у файл та дескриптор пристрою
  ADDR cName,       ; адреса ділянки пам'яті, що зберігає символи
   14,                      ; кількість символів, які записуються з cName
     ADDR cWrite, 0; адреса пам'яті, де зберігаються  символіви
invoke  WriteFile, fHandle, ADDR buf,                ; запис буфера
              cRead, ADDR cWrite, NULL
invoke  CloseHandle, fHandle ; повернення дескриптора  (закриття файлу)
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start         ; директива окончания программы з іменем start
```

Результат виконання програми з лістингу 11.4 наведено на рис. 11.4.

Рис. 11.4. Результат виконання програми з лістингу 11.4

Стисло розглянемо часто використовувані функції.

Функції LockFile та UniockFile дозволяють блокувати доступ до певної ділянки відкритого файлу, щоб до неї не можна було звернутися (прочитати або записати) з інших процесів. Ці функції застосовуються, якщо потрібно захистити файл від запису або читання з інших програм на час обробки загальнодоступних даних (наприклад, запису у файл бази даних).

Функції CopyFile, DeleteFile та MoveFile дозволяють відповідно копіювати, видаляти або переміщати вказаний файл. За допомогою функції CopyFile можна не тільки виконувати копіювання з одного диска на інший, але і перейменовувати будь-який доступний файл. Перші два аргументи функції визначають імена і шляхи початкового і кінцевого файлів. Останній аргумент встановлює дію при виявленні аналогічного файлу (якщо він дорівнює FALSE, то існуючий файл буде перезаписаний). Функція MoveFile за необхідності також виконує перейменування переміщуваного файлу. Крім того, вона підтримує переміщення цілих каталогів зі всіма підкаталогами і файлами. Проте всі переміщення можуть бути здійснені лише в межах одного логічного диска. Щоб перемістити файл між дисками, спочатку слід скопіювати його, а потім видалити початкову копію:

```
OldFile = "c:\1.txt" ;
NewFile = "c:\2.txt" ;
...
Invoke CopyFile,addr OldFile,addr NewFile,0
```

Для створення і видалення каталогів застосовуються функції CreateDirectory та RemoveDirectory. Видалити можна тільки порожній каталог, що не має вкладених каталогів і файлів.

Лістинг 11.5. Програма додавання одного файла в кінець другого:

```
.686
.model flat, stdcall
 option casemap :none
 include \masm32\include\windows.inc
```

```
        include \masm32\macros\macros.asm
        uselib kernel32,user32
.data
        _file1 db "fText1.txt",0
        _file2 db "fText2.txt",0
        hFile01 HANDLE ?
        hFile02 HANDLE ?
        from_file db 4096 dup(?)
        read_by_file dd ?
        write_by_file dd ?
        file01_size dd ?
        file02_size dd ?
      title1 db "Добавление одного файла в конец другого",0
      msg1 db "Ошибка! Файл 'fText1.txt' не доступен!",0
      msg2 db "Размер файла 'fText1.txt': %d, размер файла 'fText2.txt': %d.",10,13,
              "Считано байт: %d, записано байт: %d.",10,13,10,13,
              "НТУ ХПИ, каф. ВТП",0
      buf db 0
.code
start:
invoke CreateFile,addr _file1,GENERIC_READ,0,0,OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0
        mov hFile01, eax ; сохранение дескриптора файла
        .IF hFile01 == INVALID_HANDLE_VALUE ; если CreateFile возвращает ошибку
invoke MessageBox, 0, addr msg1, chr$("Добавление одного файла в конец другого"), 0
                                                                    ; сообщение об ошибке
        .ENDIF
; Открытие второго файла для записи
invoke CreateFile, addr _file2, GENERIC_WRITE, FILE_SHARE_READ, 0, OPEN_ALWAYS,
FILE_ATTRIBUTE_NORMAL, 0
        mov hFile02, eax ; сохранение дескриптора файла
invoke GetFileSize, hFile01, 0 ; получение размера файла 01
        mov file01_size, eax
invoke GetFileSize, hFile02, 0 ; получение размера файла 02
        mov file02_size, eax
; Чтение из файла и запись в from_file
  invoke ReadFile, hFile01, addr from_file, file01_size, addr read_by_file, NULL
; Установка указателя на конец файла
  invoke SetFilePointer, hFile02, 0, 0, FILE_END
; Запись в файл (конец) из from_file
  invoke WriteFile, hFile02, addr from_file, read_by_file, addr write_by_file, NULL
; Преобразование данных и вывод (информация работы программы и автор)
      invoke wsprintf, addr buf, addr msg2, file01_size, file02_size, read_by_file, write_by_file
      invoke MessageBox, 0, addr buf, addr title1, MB_ICONINFORMATION
        invoke ExitProcess, 0
        end start
```

Результат виконання програми з лістингу 11.5 наведено на рис. 11.5.

Рис. 11.5. Результат виконання програми з лістингу 11.5

Окрім основних файлових функцій, є набір додаткових функцій, що використовуються переважно для отримання та встановлення різних параметрів файлової системи. До них належать функції пошуку (FindFirstFile,

FindNextFile, Findciose), отримання та встановлення атрибутів файлу (SetFileAttributes та GetFileAttributes в бібліотеці kernel32.inc), скидання кеша на диск (FlushFiieBuffers), отримання та встановлення поточного каталогу (GetCurrentDirectory та SetCurrentDirectory), визначення вільного місця на диску (GetDiskFreeSpaceEx), отримання типу диска (GetDriveType) та визначення розміру файлу (GetFileSize та Get-FileSizeEx).

Синтаксис функції **SetFileAttributes**:

```
BOOL SetFileAttributes(
 LPCTSTR lpFileName, // ім'я файлу
 DWORD dwFileAttributes // атрибути
 );
```

Параметри:

lpFileName – вказівник на рядок символів, що визначає ім'я файлу, атрибути якого повинні бути встановлені;

dwFileAttributes – атрибути файлу. Цей параметр може складатися з одного або декількох наведених нижче значень. Проте всі інші значення відміняє атрибут FILE_ATTRIBUTE_NORMAL.

Атрибути файлу:

FILE_ATTRIBUTE_ARCHIVE – архівний файл. Застосування використовують цей атрибут, щоб відзначити файли для резервування або переміщення;

FILE_ATTRIBUTE_HIDDEN – файл прихований. Він не включається до звичайного списку каталогу;

FILE_ATTRIBUTE_NORMAL – у файлі немає іншого набору атрибутів. Цей атрибут допустимий тільки в тому випадку, якщо використовується поодинці;

FILE_ATTRIBUTE_NOT_CONTENT_INDEXED – файл не індексується службою, що містить індексацію;

FILE_ATTRIBUTE_OFFLINE – дані файлу безпосередньо не доступні. Цей атрибут вказує, що дані файлу були фізично переміщені на автономний запам'ятовувальний пристрій. Застосування не повинні довільно змінювати цей атрибут;

FILE_ATTRIBUTE_READONLY – файл тільки для читання. Застосування можуть читати файл, але не можуть записати в нього або видалити його;

FILE_ATTRIBUTE_SYSTEM – файл є частиною операційної системи або використовується виключно нею;

FILE_ATTRIBUTE_TEMPORARY – файл використовується для тимчасового збереження. Файлові системи уникають писати дані назад на запам'ятовувальний пристрій великої ємності, якщо доступна достатня кеш-пам'ять, тому що застосування часто видаляє тимчасовий файл після того, як дескриптор закривається. В цьому випадку система може повністю відмінити запис даних. Інакше, дані повинні бути записані після того, як дескриптор закриється.

Значення, що повертаються: якщо функція завершується успішно, значення, що повертається – не нуль. Щоб отримати додаткові дані про помилку, можна викликати функцію GetLastError.

Функція FlushFileBuffers дає можливість негайно записати всі буферізовані дані для відкритого файлу на диск. Оскільки операційна система зберігає дані в проміжному буфері (для прискорення роботи) і записує їх на диск в слушний для себе час (наприклад, під час простою), користувач ризикує втратити важливі дані при раптовому вимкненні живлення або при іншому апаратно-програмному перебої. Виклик функції FlushFileBuffers після запису всіх даних у файл гарантує їх збереження на диск. Ця функція має всього один аргумент (дескриптор відкритого файлу).

Для отримання та встановлення поточного каталогу (у поточному процесі) застосовуються відповідно функції GetCurrentDirectory та SetCurrentDirectory. Їх зручно викликати перед роботою з файлами, розташованими в поточній папці застосування. Перша функція має два аргументи: розмір та вказівник на виділений буфер. Друга просто визначає рядкове значення для установлення поточного каталогу.

Функція GetDiskFreeSpaceEx визначає величину вільного місця на логічному диску. Функція GetDiskFreeSpaceEx має чотири аргументи. Перший аргумент задає ім'я диска. Якщо встановити його в нуль, буде повернено інформацію для поточного диска. Другим аргументом є вказівник на змінну, що містить загальну кількість вільного місця на диску в байтах для зайнятого потоку (доступного користувачеві). Третій указує на загальний розмір диска. Останній аргумент повертає загальну кількостб вільного місця на диску в байтах.

Функції GetmeSize та GetFileSizeEx дозволяють визначити розмір файлу в байтах, відкритого з використанням одного з прапорців: GENERIC_WRITE або GENERIC_READ. Перша функція має два аргументи – дескриптор відкритого файлу та вказівник на старше 16-розрядне значення розміру. Другий аргумент застосовується для вимірювання великих файлів, інакше він повинен бути встановлений в нуль. При роботі з великими файлами можна викликати функцію GetFileSizeEx (тільки для Windows NT/2000/XP/SR3), яка також має два аргументи, але підтримує великі файли.

Лістинг 11.6. Програма визначення розміру файлу:

```
    .686                        ; директива определения типа микропроцессора
    .model flat,stdcall         ; задання моделі пам'яті та угоди ОС Windows
  option casemap:none           ; отличие строчных и прописных букв
  include  \masm32\include\windows.inc ; файлы структур, констант …
  include \masm32\include\kernel32.inc ; системные функции
  include  \masm32\include\\user32.inc   ; файлы  интерфейса …
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
    BS equ 128                  ; задание реального количества байтов
.data                           ; директива определения данных
  Fname BYTE "F_1.asm ",0 ; ім'я файлу, якого необхідно дізнатись та розмір
  ms1 db "Size of file (in bits) ",0    ; рядок повідомлення
  totalsize dd ?                ; комірка пам'яті, в якій зберігатиметься розмір
    stdout dd ?             ; комірка для збереження дескриптора виведення
    hFile dd ?                  ; дескриптор файлу
    buf db BS dup (?)           ; буфер
    fmt  db "%d",0          ; форматування числа
```

```
        ms2 db "Error!!File Not found"   ; повідомлення
.code                                    ; директива начала кода программы
Start:                                   ; метка начала программы с именем start:
   invoke CreateFile,ADDR Fname,GENERIC_READ,0,0, OPEN_EXISTING,\
    FILE_ATTRIBUTE_NORMAL,NULL  ; відкриття файлу
            mov hFile,eax               ; збереження отриманого дескриптора
        .IF (hFile==0FFFFFFFFh)                 ; якщо помилка
            jmp m1                              ; перейти на мітку m1
        .ENDIF
invoke GetFileSizeEx,hFile,addr totalsize       ; отримання розміру файлу
invoke wsprintf,addr buf,addr fmt,totalsize,0
invoke MessageBox,0,ADDR buf,ADDR ms1, MB_ICONINFORMATION
        jmp m2
        invoke CloseHandle,hFile
        jmp m2
m1:
 invoke MessageBox,0,ADDR ms2,ADDR ms1,MB_ABORTRETRYIGNORE
m2:
    invoke ExitProcess,0 ; вивільнення ресурсів та передача керування  ОС
        end Start                   ; директива окончания программы
```

Результат виконання програми з лістингу 11.6 наведено на рис. 11.6.

Рис. 11.6. Результат виконання програми з лістингу 11.6

Лістинг 11.7. Програма визначення останньої модифікації файлу:

```
        .686
        .model flat,stdcall
        option casemap:none
      include \masm32\include\windows.inc
      include \masm32\macros\macros.asm
      uselib kernel32,user32,comctl32
         program proto
.data
      nfile db 'kurs.asm',0
      szDateFormat db "MM.dd.yyyy",0
      szDate db 128 dup (?),0
      szTimeFormat db "HH:mm:ss", 0
      szTime db 128 dup (?),0
   inf db 'НТУ ХПИ, каф. ВТП, Рысованый А.Н., rysov@rambler.ru',10,13
   buffer db 256 dup(0),0
   FmtSysTime db "Время последнего сохранения файла: %s @ %s", 0
.code
_start:
        invoke program
        invoke ExitProcess,0
program proc;
   local fi:BY_HANDLE_FILE_INFORMATION ; для инф. от GetFileInformationByHandle
        local systime:SYSTEMTIME
invoke CreateFile,addr nfile,GENERIC_READ, FILE_SHARE_READ,0,\
                    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS,0
        mov esi, eax
fn GetFileInformationByHandle,esi,addr fi ;
fn CloseHandle, esi             ; закрыть файл
fn FileTimeToSystemTime,addr fi.ftLastWriteTime,addr systime
; отформатированная текстовая строка в соответствии с набором нац. параметров
```

```
invoke GetDateFormat,0,0, addr systime,addr szDateFormat,addr szDate,64
        ; текстовая строка времени
fn GetTimeFormat, 0, 0, addr systime,addr szTimeFormat,addr szTime, 20
fn wsprintf,addr buffer,addr FmtSysTime,addr szDate,addr szTime
fn MessageBox,0,addr inf,chr$("Дата модификации файла"),MB_ICONINFORMATION
        ret
        program endp
        end _start
```

Результат виконання програми з лістингу 11.7 наведено на рис. 11.7.

Рис. 11.7. Результат виконання програми з лістингу 11.6

11.4 Відмітки часу для файлу

Операційна система Windows зберігає для кожного файлу окремо дату і час його створення, дату і час моменту останнього доступу до файлу, а також дату і час моменту, коли останній раз виконувався запис даних у файл.

Ця інформація міститься у функції GetFileTime, прототип якої наведений нижче:

```
BOOL GetFileTime(
  HANDLE     hFile,          // идентификатор файла
  LPFILETIME lpCreationTime,   // время создания
  LPFILETIME lpLastAccessTime, // время доступа
  LPFILETIME lpLastWriteTime); // время записи
```

Перед викликом цієї функції необхідно підготувати три структури типу FILETIME і передати їх адреси через параметри lpCreationTime, lpLastAccessTime і lpLastWriteTime. У ці структури буде записана, відповідно, дата і час створення файлу hFile, дата і час моменту останнього доступу до файлу, а також дата і час моменту, коли останній раз виконувався запис даних в цей файл.

Структура FILETIME визначена таким чином:

```
typedef struct _FILETIME
{
  DWORD dwLowDateTime;  // младшее слово
  DWORD dwHighDateTime; // старшее слово
} FILETIME;
```

У структурі FILETIME зберігається 64-розрядне значення дати і часу у вигляді кількості інтервалів розміром 100 наносекунд від 1 січня 1601 року. Існує набір функцій, призначених для перетворення цього формату часу в звичніші формати, а також для порівняння значень часу у форматі структури FILETIME.

За допомогою функції FileTimeToSystemTime можна перетворити дату і час з формату структури FILETIME в зручніший для використання формат, визначуваний структурою SYSTEMTIME:

```
BOOL FileTimeToSystemTime(
  CONST FILETIME *lpFileTime,  // указатель на структуру FILETIME
  LPSYSTEMTIME  lpSystemTime); // указатель на структуру SYSTEMTIME
```

Структура SYSTEMTIME визначена так:

```
typedef struct _SYSTEMTIME
{
  WORD wYear;         // год
  WORD wMonth;  // месяц (1 - январь, 2 - февраль, и т. д.)
  WORD wDayOfWeek; // день недели (0 - воскресение, 1 - понедельник и т. д.)
  WORD wDay;          // день месяца
  WORD wHour;         // часы
  WORD wMinute;       // минуты
  WORD wSecond;       // секунды
  WORD wMilliseconds; // миллисекунды
} SYSTEMTIME;
```

Зворотне перетворення формату часу з формату структури SYSTEMTIME у формат структури FILETIME можна зробити за допомогою функції SystemTimeToFileTime:

```
BOOL SystemTimeToFileTime(
  CONST SYSTEMTIME *lpSystemTime, // указатель на структуру SYSTEMTIME
  LPFILETIME  lpFileTime); // указатель на структуру FILETIME
```

Для установки новых отметок времени файла необходимо воспользоваться функцией SetFileTime:

```
BOOL SetFileTime(
  HANDLE     hFile,           // идентификатор файла
  LPFILETIME lpCreationTime,   // время создания
  LPFILETIME lpLastAccessTime, // время доступа
  LPFILETIME lpLastWriteTime); // время записи
```

Якщо необхідно порівняти два значення часу у форматі FILETIME, то найпростіше це зробити за допомогою функції CompareFileTime:

```
LONG CompareFileTime(
  CONST FILETIME *lpTime1,  // адрес первой структуры FILETIME
  CONST FILETIME *lpTime2); // адрес второй структуры FILETIME
```

Якщо часи і дати, записані в обох структурах, рівні, функція CompareFileTime повертає нульове значення. Якщо перша дата і час більше другого, повертається значення 1, якщо ж менше - повертається негативне значення -1.

Можуть представляти інтерес функції, що виконують перетворення формату часу. Це функції FileTimeToDosDateTime и DosDateTimeToFileTime, що виконують, відповідно, перетворення дати і часу з формату структури FILETIME у формат, прийнятий в операційної системи MS-DOS, і назад.

11.5. Отримання інформації про файл та його ідентифікатор

За допомогою функції GetFileInformationByHandle можна отримати різноманітну інформацію про відкритий файл по його ідентифікатору:

```
BOOL GetFileInformationByHandle(
  HANDLE  hFile,                  // идентификатор файла
  LPBY_HANDLE_FILE_INFORMATION lpFileInformation ); // адрес
   // структуры, в которую будет записана информация о файле
```

Функція GetFileInformationByHandle записує інформацію про файл в структуру типу BY_HANDLE_FILE_INFORMATION, визначену таким чином:

```
typedef struct _BY_HANDLE_FILE_INFORMATION
{
  DWORD    dwFileAttributes;      // атрибуты файла
  FILETIME ftCreationTime;        // время создания файла
  FILETIME ftLastAccessTime;     // время доступа к файлу
  FILETIME ftLastWriteTime;       // время записи в файл
  DWORD    dwVolumeSerialNumber; // серийный номер тома
  DWORD    nFileSizeHigh;     // размер файла (старшее слово)
  DWORD    nFileSizeLow;      // размер файла (младшее слово)
  DWORD    nNumberOfLinks;       // количество связей файла
  DWORD    nFileIndexHigh;     // системный номер файла
                               // (старшее слово)
  DWORD    nFileIndexLow;      // системный номер файла
                               // (младшее слово)
} BY_HANDLE_FILE_INFORMATION;
```

Поле nNumberOfLinks використовується додатками в стандарті POSIX.

Що ж до системного номера файлу, то він відрізняється від ідентифікатора файлу, отриманого за допомогою функції CreateFile по сенсу і значенню. Системні номери файлів є глобальними і розрізняються для всіх файлів, відкритих в системі.

11.6. Створення, перейменування і видалення каталогів

CreateDirectory – функция для создания каталога:

```
CreateDirectory(
LPCTSTR lpPathName, // Указатель на строку, содержащую путь к новой директории
LPSECURITY_ATTRIBUTES lpSecurityAttributes // Указатель на атрибуты
);
```

Приклад:

```
szDirName  db  "C:\MyFolder",0
…
invoke CreateDirectory, ADDR szDirName, NULL
```

CreateDirectoryEx – расширенная функция для создания новой директории. При создании используется указанный шаблон.

Синтаксис функции CreateDirectoryEx:

```
BOOL  CreateDirectoryEx(
LPCTSTR lpTemplateDirectory, // указатель на директорию-шаблон
LPCTSTR lpPathName,  // указатель на строку, содержащую путь к новой директории
LPSECURITY_ATTRIBUTES lpSecurityAttributes   // указатель на атрибуты
);
```

Приклад:

```
' устанавливаем желаемы атрибуты безопасности
secattr.nLength = Len(secattr)  ' размер структуры
secattr.lpSecurityDescriptor = 0  ' уровень безопасности по умолчанию
secattr.bInheritHandle = 1  ' стандартная установка
```

```
' Создаем папку, используя папку C:\Recycled как шаблон.
retval = CreateDiectoryEx("C:\Recycled", "C:\RussianProject\Papka", secattr)
```

RenameDirectory – функция для переименования уже существующего каталога:

```
BOOL RemoveDirectory (LPCTSTR lpPathName   // имя каталога
);
```

Функция удаляет только папку, не содержающую файлов.

Пример:

```
RemoveDirectory("D:\14")
   If retval = 1 Then
      MsgBox "Папка была успешна удалена", vbInformation
   Else
      MsgBox "Операция удаления провалилась", vbCritical
   End If
```

FindFirstFile – функция для запуска поиска файла в указанной директории.

Для C/C++ объявлена в winbase.h. Для Delphi в модуле windows.

Синтаксис функции FindFirstFile:

```
HANDLE FindFirstFile(
LPCTSTR lpFileName, // Строка, содержащая путь для поиска файлов.
LPWIN32_FIND_DATA lpFindFileData // Информация о файле
);
```

lpFileName – строка, содержащая путь для поиска файла. Эта строка может указывать на конкретный файл типа 'c:\filename.txt' или может хранить шаблон 'c:*.txt'. При указании шаблона можно перечислить все удовлетворяющие файлы.

lpFindFileData – структура WIN32_FIND_DATA, в которую будет записана информация о найденном файле.

Расмотрим структуру WIN32_FIND_DATA:

```
typedef  struct _WIN32_FIND_DATA {
DWORD  dwFileAttributes; // aAтрибуты файла
FILETIME  ftCreationTime; // время создания
FILETIME  ftLastAccessTime; // время последнего доступа
FILETIME  ftLastWriteTime;  // время последней записи в файл
DWORD  nFileSizeHigh;  // верхний байт размера файла
DWORD  nFileSizeLow;  // нижний байт размера файла
DWORD  dwReserved0; // зарезервировано
DWORD  dwReserved1; // зарезервировано
TCHAR  cFileName[ MAX_PATH ];  // имя файла
TCHAR  cAlternateFileName[14];  // имя файла для отображения в DOS (8:3)
} WIN32_FIND_DATA;
```

Атрибутами файла может быть комбинация из флагов:

```
FILE_ATTRIBUTE_ARCHIVE –  архивный;
FILE_ATTRIBUTE_COMPRESSED –  сжатый;
FILE_ATTRIBUTE_HIDDEN –  скрытый;
FILE_ATTRIBUTE_NORMAL  – нормальный;
```

FILE_ATTRIBUTE_OFFLINE – данные файла недоступны;
FILE_ATTRIBUTE_READONLY – только для чтения;
FILE_ATTRIBUTE_SYSTEM – системный;
FILE_ATTRIBUTE_TEMPORARY – временный.

Размер файла разложен на два байта. Чтоб получить полный размер файла нужно выполнить действие (FInfo.nFileSizeHigh * MAXDWORD) + FInfo.nFileSizeLow (эффективнее сдвигать).

Функция возвращает указатель на найденный файл, если нет, то возврат будет типа ERROR_NO_MORE_FILES.

FindNextFile – функция продолжает поиск начатый с помощью FindFirstFile. Для C/C++ объявлена в winbase.h.

Синтаксис функции:

```
BOOL  FindNextFile(
HANDLE  hFindFile,  // Указатель на файл из предыдущего поиска
LPWIN32_FIND_DATA  lpFindFileData   // Информация о файле
);
```

FindClose – функция закрытия дескриптора поиска файла, открытого функцией FindFirstFile (FindFirstFileEx или FindFirstStreamW).

```
BOOL FindClose(
  HANDLE hFindFile // Дескриптор поиска файла
);
```

При ошибке возвращается ноль. Для получения дополнительной информации об ошибке, используют функцию GetLastError.

После вызова функции FindClose, дескриптор, указанный в параметре hFindFile, не может использоваться в последующих вызовах функций FindNextFile, FindNextStreamW или FindClose.

GetCurrentDirectory – функция позволяет узнать текущую директорию, с которой сейчас работает программа.

Синтаксис функции GetCurrentDirectory:

```
DWORD  GetCurrentDirectory(
DWORD  nBufferLength, // размер буфера, в котором будет храниться путь
LPTSTR  lpBuffer  // буфер
);
```

SetCurrentDirectory – функция позволяет узнать текущую директорию, с которой сейчас работает программа. Для C/C++ объявлена в winbase.h.

Синтаксис функции SetCurrentDirectory:

```
BOOL  SetCurrentDirectory(
LPCTSTR  lpPathName // новая диретория
);
```

11.7. Файлові операції засобами ShellAPI

Функція **SHFileOperation** з бібліотеки Shell32.dll виконує копіювання, переміщення, перейменування і видалення об'єктів файлової системи (папок та файлів) і **показує** віконце з індикатором виконання. В процесі виконання виводиться стандартне вікно Windows для відображення такого процесу.

Якщо користувач відміняє операцію, то встановлюється прапорець fAnyOperationsAborted у структурі SHFILEOPSTRUCT. Якщо відбудеться помилка при виконанні операції, то у користувача буде можливість або припинити операцію, або спробувати знову. Якщо користувач вибере припинення, то встановиться прапорець fAnyOperations-Aborted. При цьому функція поверне 0 (функція завершилася успішно).

Якщо функції SHFileOperation переданий прапорець FOF_NOERRORUI (тобто не показувати користувачеві повідомлень про помилки), то відбудеться помилка і операція буде перервана, а значить прапорець fAnyOperationsAborted встановлений не буде. Замість цього функція поверне код помилки, яка відбулася.

Функція має єдиний аргумент – структуру типу TSHFileOpStruct, у якій і передаються всі необхідні дані.

Тип даних LPSHFILEOPSTRUCT – це вказівник на структуру SHFILEOPSTRUCT:

```
typedef struct _SHFILEOPSTRUCT{
        HWND hwnd;
        UINT wFunc;
        LPCSTR pFrom;
        LPCSTR pTo;
        FILEOP_FLAGS fFlags;
        BOOL fAnyOperationsAborted;
        LPVOID hNameMappings;
        LPCSTR lpszProgressTitle;
        } SHFILEOPSTRUCT, FAR *LPSHFILEOPSTRUCT;
```

Поля цієї структури мають таке призначення:

hwnd – хендл вікна, на яке виводитимуться діалогові вікна про перебіг операції.

wFunc – необхідна операція. Може набувати одне із значень:

FO_COPY – копіює файли, вказані в pFrom у папку, зазначену в pTo.

FO_DELETE – видаляє файли, вказані pFrom (pTo ігнорується).

FO_MOVE – переміщає файли, вказані в pFrom у папку, зазначену в pTo.

FO_RENAME – перейменовує файли, вказані в pFrom.

pFrom – вказівник на буфер, що містить шляхи до одного або декількох файлів. Якщо файлів декілька, між шляхами ставиться нульовий байт. Список повинен закінчуватися двома нульовими байтами.

pTo – аналогічно до поля pFrom, але містить шлях до директорії – адресатові, в якому проводиться копіювання або переміщення файлів. Також може містити декілька шляхів. При цьому потрібно встановити прапорець FOF_MULTIDESTFILES.

fFlags – керівні прапорці над якими може виконуватися конкатенація прапорців:

FOF_ALLOWUNDO – зберігає інформацію про можливу відміну операції;

FOF_FILESONLY – виконує операцію тільки для файлів, якщо задано маску файлів;

FOF_MULTIDESTFILES – указує, що поле pTo описує декілька файлів (по одному на кожен файл джерела) замість одного каталогу, куди мають бути поміщені початкові файли;

FOF_NOCONFIRMATION – відповідає «Да для всех» на всі запитання;

FOF_NOCONFIRMMKDIR – не вимагає підтвердження створення нового каталогу;

FOF_RENAMEONCOLLISION – привласнює файлу нове ім'я при операціях копіювання, переміщення або перейменування, якщо файл з вказаним ім'ям вже існує;

FOF_SILENT – не показує вікно процесу;

FOF_SIMPLEPROGRESS – відображає вікно прогресу, але не показує імен файлів;

lpszProgressTitle – вказівник на рядок заголовка для вікна індикатора процесу; використовується тільки, якщо серед прапорців є FOF_SIMPLEPROGRESS.

Дескриптор повинен бути вивільнений функцією SHFree-NameMappings.

fAnyOperationsAborted – набуває значення TRUE, якщо користувач перервав будь-яку файлову операцію до її завершення і FALSE в іншому випадку.

hNameMappings – дескриптор об'єкта відображення імені файлу, який містить масив структур SHNAMEMAPPING. Кожна структура містить старі і нові імена шляху для кожного файлу, який переміщався, скопійований, або перейменований. Цей елемент використовується тільки, якщо встановлений прапорець FOF_WANTMAPPINGHANDLE.

lpszProgressTitle – вказівник на рядок, що використовується як заголовок для діалогового вікна процесу. Цей елемент використовується тільки, якщо встановлений прапорець FOF_SIMPLEPROGRESS.

Якщо папка, в яку необхідно копіювати файли, не існує, то виводиться діалогове вікно із запитом щодо створення такої папки. Відображення процесу запису наведено на рис. 11.8.

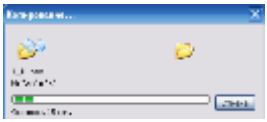


Рис. 11.8. Відображення процесу запису

Якщо при копіюванні, переміщенні або перейменуванні файл із вказаним ім'ям вже існує в папці-адресатові, то він буде замінений без застережливого повідомлення. Інакше – виводиться діалог для підтвердження заміни (рис. 11.9).

Рис. 11.9. Відображення діалогу для підтвердження заміни

Приклад використання функції SHFileOperation для перезапису всіх файлів із розширенням .asm з каталогу c:\masm32\bin\ в каталог зі шляхом c:\x наведено в лістингу 11.8.

Лістинг 11.8. Перезапис файлів із розширенням .asm:

```
      .686                        ; директива определения типа микропроцессора
     .model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
      option casemap:none    ; відмінність великих та малих літер
      include \masm32\include\windows.inc ; описи структур, констант і ін.
      include \masm32\include\kernel32.inc ; описи API, які  включено в ядро
      include \masm32\include\user32.inc    ; файлы  интерфейса …
      include \masm32\include\shell32.inc
      includelib\masm32\lib\kernel32.lib
      includelib \masm32\lib\user32.lib
      includelib\masm32\lib\shell32.lib
.data                                    ; директива описания данных
_SHFILEOPSTRUCT struc          ; тип даних СТРУКТУРА
    hwnd    dd ?  ; дескриптор вікна
    wFunc   dd ? ; операція, яка виконується
    pFrom   dd ? ; адреса рядка з найменуванням файлів
    pTo     dd ? ; адреса рядка з указанням папки
    fFlags  dd ? ;
    fAnyOperationsAborted dd ? ;
    hNameMappings dd ?
    lpszProgressTitle dd ?
_SHFILEOPSTRUCT ends
File db "c:\masm32\bin\*.asm",0,0  ; шлях із двома нульовими байтами
Dir db "c:\x", 0                ; шлях створення папки
shf _SHFILEOPSTRUCT <0,\     ; hwnd хэндл окна
  FO_
_st: COPY,\  ; копіює файли, вказані в File в папку, вказану в Dir
   File,\     ; вказівник на буфер, що містить шляхи до файлів
     Dir, \         ; шлях до адресата
      FOF_FILESONLY,\     ; виконує операцію для файлів із маскою
       0, 0, 0 >    ; ознака операції, яку перервали, та заголовок вікна
.code
     invoke SHFileOperation, addr shf
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
     end  _st   ; директива окончания программы
```

Основна частина програми складається з однієї директиви invoke SHFileOperation з одним параметром addr shf, за адресою якого викликається структура shf <>.

Функція **SHEmptyRecycleBin** – це функція для очищення корзини на диску (або на всіх). Функція має синтаксис:

```
HRESULT SHEmptyRecycleBin(
  __in_opt  HWND hwnd,
  __in_opt  LPCTSTR pszRootPath,
  DWORD dwFlags
);
```

де:

hwnd – handle батьківського або діалогового вікна, яке відображається. Цей параметр може дорівнювати нулю.

pszRootPath – вказівник (адреса) на рядок, що містить шлях до кореневого диска, на якому знаходиться корзина. Цей параметр може містити адресу рядка або шлях до каталогу, або підкаталогу, наприклад c:\windows\system... . Якщо це значення – порожній рядок або NULL, то всі корзини на всіх дисках будуть очищені.

dwFlags – прапорці операції:

SHERB_NOCONFIRMATION – не показувати діалог підтвердження видалення всіх файлів із корзини;

SHERB_NOPROGRESSUI – не показувати «прогрес бар», що відображає видалення файлів із корзини;

SHERB_NOSOUND – не підтверджувати завершення видалення звуковим сигналом.

Функція **SHQueryRecycleBin** повертає розмір корзини і кількість елементів у ній на диску. Функція має синтаксис:

```
HRESULT SHQueryRecycleBin(
  __in_opt  LPCTSTR pszRootPath,
  __inout   LPSHQUERYRBINFO pSHQueryRBInfo
);
```

де:

pszRootPath – вказівник на рядок, що містить шлях до кореневого диска, на якому знаходитися корзина. Цей параметр може містити адресу рядка або шлях до якого-небудь каталогу або підкаталогу, наприклад c:\windows\system...

PSHQueryRBInfo – вказівник на структуру SHQUERYRBINFO, яка отримує інформацію про корзину. Значення поля структури CbSize має бути встановлено рівним розміру структури перед запитом цієї функції.

Оголошення структури SHQUERYRBINFO:

```
typedef struct {
  DWORD   cbSize;
  __int64 i64Size;
  __int64 i64NumItems;
} SHQUERYRBINFO, *LPSHQUERYRBINFO;
```

Опис полів:

cbSize – розмір структури, в байтах;

I64Size – повний розмір всіх елементів у вказаній корзині, в байтах;

I64NumItems – загальна кількість елементів у вказаній корзині.

11.8. Лабораторна робота «Файли»

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм зі створенням файлів та запису в них даних.

Зміст звіту

1. Постановка задачі для конкретного варіанта завдання.
2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.
3. Програма та коментарі до всіх команд.
4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання.
5. Короткий опис виконання програми.
6. Висновки за результатами роботи.

Постановка задачі

Завдання 1

Згідно з останньою цифрою номера студента в групі вибрати варіант завдання та написати програму ***з виведенням даних у файл та в функцію*** **MessageBox.** Додати своє прізвище та ініціали в файл, який був отриманий у результаті виконання завдання. Парні номери роблять запис у кінець файлу, а непарні – з потокової позиції вказівника файлу. Визначити розмір створеного файлу. В спрощене вікно вивести свої прізвище та ініціали, номер навчальної групи, e-mail, завдання та результати його виконання. Використати функцію **SHFileOperation.**

1. Проаналізувати масив даних із 15 елементів. Підрахувати і зберегти кількість елементів масиву, якщо їх значення менше або більше 132, та кількість елементів масиву, значення яких дорівнюють 132. Вивести відповідні повідомлення.
2. Проаналізувати масив даних із 10 елементів. Додавати елементи масиву доти, доки значення суми не перевищить 512. Зберегти номер елемента, на якому відбулося переповнення. Якщо сума елементів не досягла значення 512, то видати відповідне повідомлення.
3. Проаналізувати масив даних із 16 елементів. Елементами масиву є числа 32, 64, 96 і 128. Підрахувати та вивести на екран кількість повторень кожного елемента.
4. Проаналізувати масив даних із 12 елементів. Створити масив, до якого входять елементи першого масиву, що дорівнюють 128. Перервати виконання програми, якщо буде знайдено 5 елементів зі значенням 128. Вивести відповідні повідомлення.
5. Проаналізувати 2 масиви, що складаються з 15 елементів кожен. Підрахувати кількість елементів першого масиву, що мають рівні значення в другому масиві. Вивести відповідні повідомлення.

6. Проаналізувати масив даних із 14 елементів. Підрахувати кількість елементів, значення яких дорівнює 55h. Рахування перервати, якщо кількість елементів перевищить 3. Вивести відповідні повідомлення.

7. Проаналізувати масив даних із 15 елементів. Елементами масиву є числа 10, 20, 30 і 180. Підрахувати кількість повторень кожного елемента. Вивести відповідні повідомлення.

8. Для функції $Y = 40X + 65$ знайти перше значення аргументу, при якому значення функції перевищить 1024. Початкове значення аргументу $X = 20$. Вивести відповідні повідомлення.

9. Для функції $Y = 40X + 10$ одержати перше значення, що перевищує 512, починаючи з $X = 1$. Вивести значення аргументу та функції.

10. Для функції $7X + 85$ знайти перше значення аргументу, при якому молодші цифри результату виконання функції дорівнюють 155. Вивести результат виконання функції та його аргумент.

Завдання 2

Згідно з останньою цифрою номера студента в групі вибрати варіант завдання та написати програму:

1. Визначення версії програми.
2. Визначення дати останньої зміни файлу.
3. Визначення дати останнього доступу до файлу.
4. Визначення дати створення файлу.
5. Визначення типу файлу.
6. Визначення властивостей файлу.
7. Визначення дати створення файлу.
8. Видалення каталога з підкаталогами.
9. Визначення файлів у певній директорії.
10. Визначення шляху до особистої програми.
11. Копіювання директорії.

При написанні програми можна використати посилання на сайт http://www.desksoft.ru/ з вибором ікони сайта з підписом DRKB.

Приклад 11.1. Проаналізувати масив даних із 16 елементів. Елементами масиву є числа 32, 64, 96 і 128. Підрахувати та вивести на екран кількість повторень кожного елемента.

Програма з лістингу 11.9 виводить в спрощене віконце та створює файл із записом кількості чисел, які збіглися за умовами програми.

Лістинг 11.9:

```
    .686                        ; директива определения типа микропроцессора
    .model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
    option casemap:none     ; отличие строчных и прописных букв
  include \masm32\include\windows.inc ; файлы структур, констант …
  include \masm32\include\\user32.inc   ; файлы  интерфейса …
  include \masm32\include\kernel32.inc ; системные функции…
  include \masm32\include\\masm32.inc
  includelib \masm32\lib\user32.lib
  includelib \masm32\lib\kernel32.lib
  includelib \masm32\lib\masm32.lib
.data
  _mas dd 32, 64, 32, 128, 96, 96, 64, 128, 64, 64, 32, 64, 96, 128, 128, 32
```

```
      len equ ($-_mas)/ type _mas ; визначення кількості чисел
      _n1 dd ?                    ; комірка для збереження _n1
      _n2 dd ?                    ; комірка для збереження _n2
      _n3 dd ?                    ; комірка для збереження _n3
      _n4 dd ?                    ; комірка для збереження _n4
        const1 dd 32  ; привласнення змінній значення 32
        const2 dd 64  ; привласнення змінній значення 64
        const3 dd 96  ; привласнення змінній значення 96
        const4 dd 128             ; привласнення змінній значення 128
fName db "10_.txt", 0  ; комірки для ім'я файлу
fHandle DWORD ?   ; комірки з ім'ям fHandle для дескриптора  збереження файлів
cWritten  DWORD  ? ; для адреси символів виведення
msg1 db "Кількість елементів:", 0ah, 0dh,\
 " 32 - %d",0ah,0dh, " 64 - %d",0ah,0dh,\
  " 96 - %d",0ah,0dh,  "128 - %d",0
        titl db "Результат програми",0
.code
  start:
        lea esi, _mas  ; занесення адреси   mas
        mov ecx, len   ; лічильник чисел
cycle2:
        mov eax,[esi]  ; занесення числа
  .IF eax==const1    ; якщо число дорівнює 32
        inc _n1                   ; додати 1 у лічильник число 32
  .ELSEIF eax==const2             ;
        inc _n2                   ; додати 1 у лічильник число 64
  .ELSEIF eax==const3             ;
        inc _n3                   ; додати 1 у лічильник число 96
  .ELSE
        inc _n4                   ; додати 1 у лічильник число 32
  .ENDIF
        add esi,4h ; переведення вказівника на наступний елемент масиву
loop cycle2
next:
invoke CreateFile, ADDR fName, GENERIC_WRITE, ; запис у файл
  0, NULL,                        ; параметри багатозадачності
   CREATE_ALWAYS,                 ; знищити та створити новий файл
    FILE_ATTRIBUTE_ARCHIVE, 0
        mov   fHandle, eax      ; запам'ятання дескриптора пристрою
invoke WriteFile, fHandle,  ; запис у файл за дескриптором пристрою
 ADDR _n1,    ; адреса ділянки пам'яті, що зберігає символи
  16, ADDR cWritten, 0 ; кількість символів та  адр. для кількості символів
   invoke  CloseHandle, fHandle             ; закрити дескриптор файлу
        mov eax, _n1;
        mov ebx, _n2;
        mov ecx, _n3;
        mov edx, _n4;
      invoke wsprintf, ADDR msg1,ADDR msg1,eax, ebx, ecx, edx
      invoke MessageBox,0,ADDR msg1,ADDR titl,MB_ICONINFORMATION ;
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start               ; директива окончания программы з іменем start
```

У програмі функція wsprintf перетворює числа в символи відразу всіх 4-х регістрів, які потім виводяться функцією MessageBox в спрощене віконце (рис. 11.10).

Рис. 11.10. Результат виконання програми з лістингу 11.9

Для запуску зовнішнього файла може використатися функція ShellExecute (лістинг 11.10).

Лістинг 11.10. Програма запуску файлу з іменем 11_8.exe:

```
        .686
        .model flat, stdcall
        option casemap :none
    include \masm32\include\windows.inc
    include \masm32\macros\macros.asm
    uselib kernel32, user32, shell32
.data
        open db "c:\masm32\bin\11_8.exe",0
        open db "open",0
.code
start:
        invoke ShellExecute, 0, addr open, addr oper,0,0, SW_SHOWNORMAL
        invoke ExitProcess, 0
        end start
```

Завдання для практичного виконання

1. Создание нового файла из содержимого двух файлов.
2. Создание нового файла из содержимого трех файлов.
3. Создание пяти файлов с содержимым из известного файла.
4. Создать два файла: в первый записать содержимое 1 + 2 файлов, а во второй – 2 + 1 файлов.
5. Создать два файла: в первый записать содержимое 1 + 2 + 3 файлов, а во второй – 3 + 2 + 1 файлов.
6. Создать директорию (CreateDirectoryEx) и записать в нее файл.
7. Переименовать директорию (RenameDirectory) и записать в нее файл.
8. Запуск поиска файла в указанной директории (FindFirstFile).
9. Вывести текущую директорию, с которой работает программа.

Контрольні запитання

1. Яка функція Win32 API призначена для створення файлу?
2. Яка функція Win32 API призначена для запису у файл?
3. Які дії необхідно виконати для читання даних із файлу?
4. Які дії необхідно виконати Для того щоб вставити дані в початок файлу?

12. АРИФМЕТИЧНИЙ СПІВПРОЦЕСОР

Арифметичний співпроцесор призначений для виконання операцій над числами у форматі з рухомою крапкою(дійсні числа) і довгими цілими числами. Він значно (у десятки разів) прискорює обчислення, пов'язані з дійсними числами. Співпроцесор може обчислювати такі функції, як синус, косинус, тангенс, логарифми і т.д. За допомогою співпроцесора можна виконувати і прості арифметичні операції складання, віднімання, множення і ділення.

Основна сфера застосування арифметичного співпроцесора – наукові розрахунки і машинна графіка.

Паралельна робота мікропроцесора і співпроцесора підвищує продуктивність виконання програм у цілому. Співпроцесор розширює математичні можливості основного мікропроцесора, але не заміщає жодну з його команд. З погляду програміста, система з математичним співпроцесором виглядає як єдиний мікропроцесор із розширеним набором команд.

Співпроцесор запускається центральним мікропроцесором. Після запуску він виконує всі обчислення самостійно і паралельно з роботою центрального мікропроцесора. Якщо центральний мікропроцесор видає чергову команду співпроцесору у момент часу, коли той ще не закінчив виконання попередньої команди, центральний мікропроцесор переводиться в стан очікування. Якщо ж співпроцесор нічим не зайнятий, центральний мікропроцесор, видавши команду співпроцесору, продовжує свою роботу, не чекаючи завершення обчислення. Існують спеціальні засоби синхронізації пристроїв (команда FWAIT).

На початку будь-якої команди розташовується код операції, відповідно до якого мікропроцесор обробляє цю команду самостійно або передає її співпроцесору, а сам продовжує виконання програми з наступної команди.

Асемблерні мнемоніки всіх команд співпроцесора починаються з букви F, наприклад: FADD, FDIV, FSUB тощо. Команди співпроцесора можуть адресуватися до операндів, аналогічно до звичайних команд центрального мікропроцесора. Для команд арифметичного співпроцесора можливі всі види адресації даних, що використовуються центральним мікропроцесором.

12.1. Типи даних співпроцесора

До форматів даних співпроцесора належать:

– цілі двійкові числа розрядністю 16, 32 і 64 бітів;

– упаковані цілі десяткові (BCD) числа розміром до 9 байтів;

– дійсні числа в короткому (32 біти), довгому (64 біти) і розширеному (80 бітів) форматах.

Межі зміни даних у цих форматах наведено у табл. 12.1.

Крім цих форматів, підтримуються і спеціальні числові значення, до яких належать:

– денормалізовані дійсні числа;

– нуль;

– додатні і від'ємні значення нескінченності;

– нечисла;

– формати різного роду невизначеності.

Таблиця 12.1

Тип даних	Біт	Кількість значущих цифр	Межі
Ціле слово	16	4	– 32 768–32 767
Коротке ціле слово	32	9	$-2*10^9$–$2*10^9$
Довге ціле слово	64	18	$-9*10^{18}$–$9*10^{18}$
Упаковане десяткове слово	80	18	– 99..99–+99..99 (18 цифр)
Коротке дійсне слово	32	7	$1{,}18*10^{-38}$–$3{,}40*10^{38}$
Довге дійсне слово	64	15–16	$2{,}23*10^{-308}$–$1{,}79*10^{308}$
Розширене дійсне слово	80	19	$3{,}37*10^{-4932}$–$1{,}18*10^{4932}$

Співпроцесор має єдине внутрішнє подання даних і зберігає всі числа в єдиному 80-розрядному розширеному форматі. Цей формат відповідає формату регістрів стека співпроцесора. Будь-які операнди, подані у вигляді 16-, 32- і 64-розрядних цілих чисел, 32-, 64- або 80-розрядних чисел з рухомою крапкою, а також упакованих BCD-чисел, зображені 18 цифрами, при завантаженні в регістри співпроцесора автоматично переводяться в розширений формат. Результати обчислень переводяться назад в один з цих форматів даних і зберігаються в регістрах або пам'яті.

При виборі формату необхідно враховувати, що, хоча співпроцесор і підтримує цілочисельні формати, операції з ними виконуються недостатньо ефективно. Причина цього криється в тому, що всі цілочисельні дані переводяться в розширений формат дійсного числа (внутрішнє подання чисел в співпроцесорі), що вимагає додаткових операцій перетворення.

Співпроцесор включає дві команди для роботи з упакованими десятковими числами.

Дійсні числа можуть мати розмірність 32, 64 або 80 біт.

При записі дійсних чисел завжди виконують нормалізацію – помножують число на такий ступінь двійки, щоб перед десятковою точкою була одиниця.

При використанні цього алгоритму перша цифра мантиси завжди дорівнює 1, тому її можна не писати, збільшуючи тим самим точність подання числа додатково на 1 біт. Крім того, значення експоненти зберігають не у вигляді цілого числа із знаком, а у вигляді суми чисел з деяким числом так, щоб завжди зберігати тільки позитивне число і щоб було легко порівнювати дійсні числа – в більшості випадків досить порівняти експоненту.

Для подання 32-розрядних дійсних чисел використовується директива DD, а для 64-розрядних – директива DQ.

До не-чисел (Non-A-Number, NAN) належать послідовності бітів, які не можна зіставити з жодним форматом. До спеціальних числових значень зара-

ховують нуль і нескінченність. Значення “нуль” може, наприклад, бути результатом роботи команд з нульовими операндами. У перелік спеціальних числових значень входять і так звані денормалізовані числа. Це числа, що виходять за межі діапазону даного формату і наближаються до нуля.

12.2. Архітектура співпроцесора

Програмна модель співпроцесора є набором додаткових регістрів, типів даних і команд. До групи регістрів співпроцесора входять :

– вісім 80-розрядних регістрів, що окремо адресуються та організовані у вигляді регістрового стека;

– три службових 16-розрядних регістри: регістр стану swr (Status Word Register) співпроцесора; регістр cwr (Control Word Register); регістр тегів twr (Tags Word Register);

– регістри-вказівники даних dpr (Data Point Register) і команд ipr (Instruction Point Register), які використовуються при обробці виняткових ситуацій.

Кожний із восьми числових регістрів у стеку має розмір 80 бітів і складається з окремих полів відповідно до розширеного дійсного типу даних (рис. 12.1).

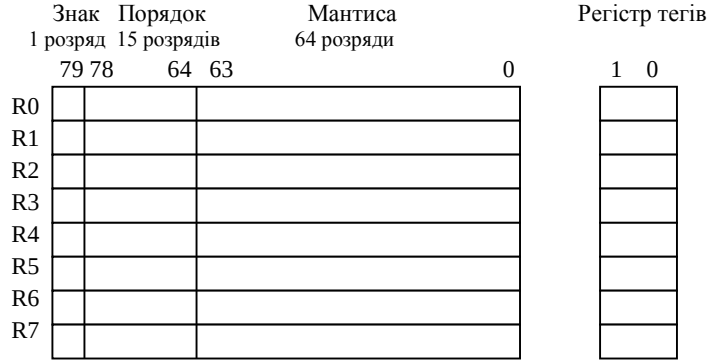


Рис. 12.1. Регістровий стек співпроцесорів

У будь-який момент часу номер регістра міститься в полі top регістра стану (swr) співпроцесора.

Регістр **стану** (swr) містить інформацію про поточний стан співпроцесора. Деякі найбільш важливі поля наведено на рис. 12.2.

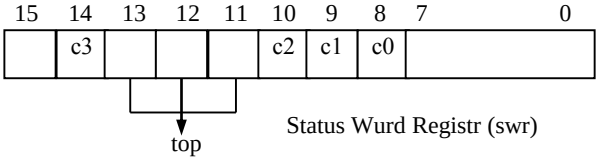


Рис. 12.2. Регістр стану (swr) співпроцесора

Поля **c3–c0** регістра, що встановлюються певним чином після виконання математичних операцій, є аналогом прапорців у регістрі прапорців EFLAGS основного мікропроцесора. Поля 13–11 містять поточний номер регістра стека (R0–R7), який в даний момент є вершиною.

Слово стану регістра swr зберігається в пам'яті за допомогою команд fstsw/fnstsw, fstenv/fnstenv і fsave/fnsave, після чого його можна передати в регістр AX за допомогою команд fstsw AX/fnstsw AX, дозволяючи програмі перевірити слово стану співпроцесора.

Команда sahf може копіювати поля c3 – c0 безпосередньо в біти прапорців мікропроцесора, що спрощує організацію розгалужень у програмах.

Регістр **тегів** (twr) розподіляється на вісім двобітових полів. Кожне поле належить до свого чисельного регістра (рис. 12.3).

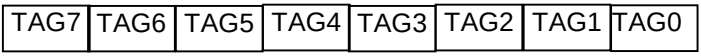


Рис. 12.3. Формат регістра тегів

Поля регістра тегів (twr) класифікують вміст «свого» числового регістра, опис якого наведено в табл. 12.2.

Таблиця 12.2

Поле	Опис
00	Регістр містить дійсне ненульове число
01	У регістрі знаходиться нуль
10	Регістр містить недійсне число – нечисло, нескінченність, невизначеність
11	Порожній неініціалізований регістр

Наприклад, якщо всі регістри співпроцесора спочатку були порожні, а потім у стек числових регістрів було занесене одне дійсне ненульове значення, то тоді вміст регістра тегів становитиме 3FFFh. Дані полів тегів відображають стан фізичних регістрів R0 – R7, а не логічних st(0) – st(7). Для отримання повної інформації про стан регістрів стека потрібно буде використовувати поле top регістра стану (swr).

Для встановлення режимів обробки даних співпроцесора служить регістр керування (**cwr**) (рис. 12.4). Він включає шість масок виключень, поля керування точністю (precision control, pc) і поля керування округленням (rounding control, re).

За допомогою бітів поля **pc** вибирається довжина мантиси:

- 00 – довжина мантиси, що дорівнює 24 бітам;
- 10 – довжина мантиси, що дорівнює 53 бітам;

- 11 – довжина мантиси, що дорівнює 64 бітам.

Рис. 12.4. Регістр стану керування (cwr)

Далі показано можливі комбінації поля **rc**:

- 00 – число, що міститься в стеку st(0) співпроцесора та округляється до найближчого цілого значення;
- 01– число в st(0), що округляється в меншу сторону;
- 10 – число в st(0), що округляється у більшу сторону;
- 11 – число, від якого відкидається дробова частина.

Маски призначено для маскування виняткових ситуацій, виникнення яких фіксується в регістрі стану. При установленні бітів виключень в 1 відбуватиметься обробка цих виключень математичним співпроцесором. При виникненні таких виключень збуджується переривання. При цьому оброблювач переривання повинен або знаходитися в самій операційній системі, або розроблятися програмістом.

При виникненні помилки мікропроцесор намагається порушити переривання за особливою ситуацією, встановлюючи відповідні біти в слові стану. Проте, якщо ситуація замаскована за допомогою керуючого слова, співпроцесор сам реагує на неї.

Біти керування (біти 8–9) можуть бути використані Для того щоб встановити меншу точність внутрішніх операцій пристрою, порівняно з точністю, заданою за умовчанням (розмір мантиси дорівнює 64 бітам).

12.3. Система команд математичного співпроцесора

Команди з рухомою крапкою можна згрупувати в класи:

- команди передавання даних;
- команди порівняння;
- арифметичні команди;
- трансцендентні команди;
- команди маніпуляції константами;
- команди керування.

В макроасемблері masm32 є довідник щодо системи команд співпроцесора за адресою masm32/tutorial/fputute.

Формати команд співпроцесора x87 аналогічні до форматів команд x86. Асемблерні мнемоніки команд співпроцесора починаються з букви F (Floating).

Якщо другою літерою є буква I (Integer), то це позначає операцію з цілим двійковим числом із пам'яті. Якщо другою літерою є літера B (Binari-coded decimal), то це позначає операцію з десятковим операндом із пам'яті.

Передостання або остання літера R указує на зворотну операцію (для віднімання і ділення).

Остання літера P (Pop) ідентифікує команду, завершальною дією якої є витягання числа із стека.

У командах неявно указується числовий регістр співпроцесора, що адресується вершиною стека st або st(0) (Stack Top).

Числові регістри адресуються щодо вершини стека. Наприклад, команда

```
fadd st(0),st(4)
```

додає вміст четвертого регістра від вершини стека до вмісту верхнього регістра стека.

Верхній регістр стека можна позначити st або st(0). Наприклад, команда

```
fadd st(0),st(0)
```

додає вміст верхнього регістра стека до нього ж, тобто подвоює вміст регістра st(0).

Для операндів у пам'яті допускаються всі режими адресації мікропроцесора x86, наприклад:

```
fadd    [bx]
fadd    name [bx] [si]
```

У командах FPU ***не використовується безпосередня адресація*** операндів.

Masm32-функції перетворення форматів даних

Для спрощення програмування на співпроцесорі використовуються **вбудовані masm32-функції перетворення:**

- **FpuAtoFL** – функція перетворення десяткового числа в дійсне число формату REAL10 (80-розрядне)
- **FpuFLtoA** – функція перетворення дійсного числа формату REAL10 в десяткове число.

Функція **FpuAtoFL** має прототип:

```
FpuAtoFL (
  lpszSrc   // вказівник на джерело з нулем на кінці
  lpDest    // вказівник на приймач результату
  uID       // параметр місця призначення
  )
```

Параметр *uID* місця призначення може бути таким:

DEST_FPU – результат залишається на FPU;

DEST_MEM – результат записується у пам'ять розмірністю TBYTE.

Якщо результат повинен бути повернений на FPU, то регістр ST7 даних FPU має бути вільний, інакше вміст цього регістра зміниться.

Наприклад, invoke FpuAtoFL, ADDR Src, 0, DEST_FPU.

У цьому прикладі десяткове число з іменем Src перетворюється на 80-розрядне дійсне число і зберігається на FPU.

Функція **FpuFLtoA** має такий прототип:

FpuFLtoA (
lpSrc1 // вказівник (адреса) на 80-розрядне дійсне число
lpSrc2 // кількість десяткових знаків після коми
lpszdest // вказівник на місце призначення результату з нулем на кінці
uID // параметр джерел та формату
)

Параметр *uID* джерел та формату може бути таким:

SRC1_FPU – операнд Src1 є вже на FPU;
SRC1_REAL – вказівник на 80-розрядне дійсне число в пам'яті;
SRC2_DMEM – вказівник на 32-розрядне ціле число із знаком;
SRC2_DIMM – операнд Src2 є 32-розрядним цілим числом із знаком;
STR_REG – рядок повинен бути повернений в десятковому форматі;
STR_SCI – рядок повинен бути повернений в експоненціальному форматі.

12.3.1. Команди передавання даних

Команди передання даних переміщують дані між регістрами стека і пам'яттю. Ці команди можна розділити на три групи:

– команди передання чисел із рухомою крапкою;
– команди передання цілих чисел;
– команди передання десяткових чисел.

Команди передання даних автоматично оновлюють регістр стану співпроцесора.

FLD джерело	Завантажити дійсне число в стек

Команда **fld** ***операнд-джерело*** завантажує дійсне число з елемента пам'яті, поданої операндом-джерелом, у вершину стека співпроцесора. Команда **FLD** поміщає вміст джерела (32- , 64- або 80- бітову змінну або ST(*n*)) за наведеною адресою і зменшує поле TOP регістра стану (swr) співпроцесора на 1. Команда FLD ST(0) робить копію вершини стека.

```
fld  st(4)  ; ST(0) ← ST(4) та зсув всіх регістрів стека співпроцесора
                                        ; так, що  ST(4) переміщується в ST(5)
fld  st     ; ST(0) переписується в ST(0) та переміщується в ST(1)
fld real4_var ; змінна  real4_var перетворюється на формат REAL10 та копіюється в ST(0)
fld  qword ptr[ebx] ; в ST(0) записується число розміром QWORD
                                        ; з адреси пам'яті, яка вказана в ebx
```

```
fld  tbyte ptr[edi+30] ; в ST(0) записується число розміром REAL10
                       ; з адреси пам'яті, яка вказана в edi + 30
```

FST приймач	Скопіювати дійсне число зі стека
FSTP приймач	Скопіювати дійсне число зі стека та виштовхнути його зі стека

Команди **fst** ***операнд-приймач,*** **fstp** ***операнд-приймач*** зберігають дійсне число з вершини стека в пам'яті. Команда **fst\fstp** копіює вершину стека ST(0) в приймач (32- або 64-бітову змінну або порожній ST(*n*) – у випадку FST; 32-, 64- або 80-бітову змінну або порожній ST(*n*) – у випадку FSTP). Крім того, команда fstp ще виштовхує операнд із стека, збільшуючи тим самим значення top на 1. В цьому випадку вершиною стека стає фізичний регістр із великим номером.

```
    fst  st(5)  ; ST(5) ← ST(0)  та залишається в ST(0)
fst dword ptr [eax]  ; запам'ятовування значення ST(0) у форматі REAL4
                                  ; в пам'яті за адресою, вказаною EAX
fst  real_var ; запам'ятовування значення ST(0) в пам'яті у відповідному
                                  ; форматі  за адресою real_var змінної пам'яті
fstp st(4)   ; ST(4) ← ST(0)  та ST(0) виштовхнути зі стека;
                                  ; в результаті чого ST(4) зсунеться в ST(3)
```

Для передання ***цілочислових*** даних використовуються такі команди:

FILD джерело	Завантажити ціле число в стек

Команда **fild** ***операнд-джерело*** завантажує число з елемента пам'яті, поданої операндом-джерелом, у вершину стека. Команда **FILD** перетворює ціле число зі знаком із джерела (16-, 32- або 64-бітову змінну) в дійсний формат, розташовує його у вершині стека і зменшує поле TOP регістра стану (swr) співпроцесора на 1.

FIST приймач **FISTP** приймач	Скопіювати ціле число зі стека

Команда **fist** ***операнд-приймач,*** **fistp** ***операнд-приймач*** зберігає ціле число з вершини стека в пам'яті. Перетворюють число з вершини стека в ціле число зі знаком і записують його в приймач (16- або 32- бітову змінну – для FIST; 16-, 32- або 64-бітову змінну – для FISTP). Команда flstp ще виштовхує операнд із стека, збільшуючи тим самим значення поля top регістра стану (swr) співпроцесора на 1. В цьому випадку вершиною стека стає фізичний регістр з великим номером.

Команди передання даних, поданих у ***десятковому*** форматі:

FBLD джерело	Завантажити десяткове число в стек

Команда **fbld** ***операнд-джерело*** завантажує число з 80-розрядного елемента пам'яті, поданого операндом-джерелом, у вершину стека.

FBSTP приймач	Завантажити десяткове число зі стека та виштовхнути

Команда **fbstp** ***операнд-приймач*** зберігає ціле число з вершини стека в пам'яті і виштовхує операнд із стека, збільшуючи тим самим значення поля top на 1. В цьому випадку вершиною стека стає фізичний регістр із великим номером.

Перед використанням будь-яких команд математичного співпроцесора необхідно виконати його ініціалізацію командою **finit**. Алгоритм роботи команди такий:

1. У регістр керування (cwr) поміщається число 37h, яке означає, що округлення виконуватиметься до найближчого цілого числа (поле rc = 0).

2. Біти з нульового по п'ятий встановлюються в 1, що означає маскування всіх виключень.

3. Поле керування точністю встановлюється для роботи з максимальною точністю в 64 біти (pc =11).

4. Регістр стану (swr) заповнюються нулями, а це означає, що виключення відсутні і фізичний регістр R0 регістрового стека стає вершиною стека st(0).

5. У регістр тегів (twr) заносяться одиниці, а це означає, що всі регістри співпроцесора вважаються порожніми.

6. Регістри вказівників даних (dpr) і команд (ipr) заповнюються нулями.

Приклад 12.1. Перемістити п'ять дійсних чисел з одного масиву пам'яті в інший.

Програму розв'язання прикладу 12.1 наведено у лістингу 12.1.

Лістинг 12.1. Програма розв'язання прикладу 12.1:

```
        title CopyRight by Rysovaniy A. N.
        .686                            ; директива определения типа микропроцессора
        .model flat,stdcall  ; задание линейной модели памяти  и соглашения ОС Windows
        option casemap: none            ; отличие строчных и прописных букв
        include    \masm32\include\kernel32.inc
        includelib \masm32\lib\kernel32.lib
.data                                                   ; директива определения данных
   mas1 DD 129.235, -1024.01, -12.5, 5.06, 67895.025
   len EQU ($- mas1)/type mas1  ; обчислення кількості чисел масиву mas1
   mas2 DD len DUP(0)        ; резервування len комірок пам'яті для mas2
.code
  start:
        finit               ; ініціювання співпроцесора
      mov ECX, len       ; занесення len = 5 – кількості байтів масиву mas1
      lea  ESI, mas1          ; занесення адреси початку масиву
      lea  EDI, mas2          ; занесення адреси початку масиву
next: fld  dword ptr [ESI]    ; занесення в вершину стека з [ESI]
        fstp dword ptr [EDI] ; занесення з вершини стека в пам'ять
                                                        ; за адресою, яка знаходиться в EDI
      add  ESI, 4        ; вирівнювання на межу подвійного слова масиву mas1
```

```
    add  EDI, 4         ; вирівнювання на межу подвійного слова масиву mas2
     loop  next         ; перейти, якщо у лічильнику ECX не нуль
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
     end start          ; директива завершение программы с именем start
```

Вікна налагоджувача при першому проходженні циклу програми з лістингу 12.1 наведено на рис. 12.5.

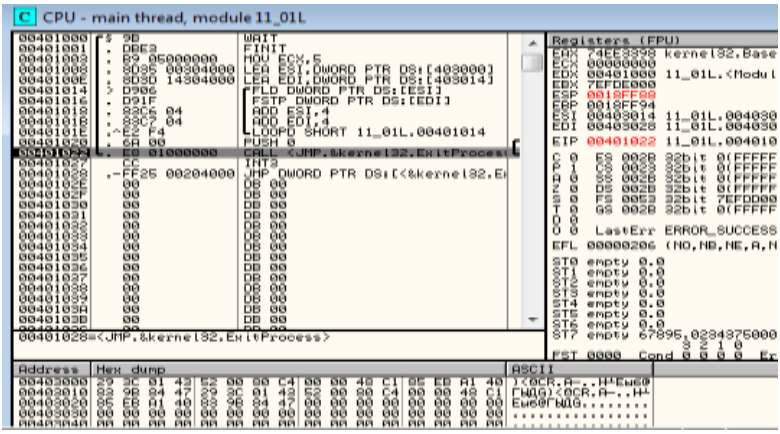


Рис. 12.5. Вікна налагоджувача при виконанні програми з лістингу 12.1

У вікні регістрів (Registers) можна бачити, що перше число з масиву mas1 заноситься у вершину стека st(0). В комірках пам'яті, починаючи з адреси 00403000, наведено п'ять чисел масиву mas1 розміром подвійного слова, які після виконання програми перемістяться у масив mas2.

Лістинг 12.2. Програма розв'язання прикладу 12.1 з виведенням результату у спрощене вікно:

```
.686
.model flat,stdcall
option casemap: none
include \masm32\include\windows.inc ; файлы структур, констант
include \masm32\macros\macros.asm
uselib user32,kernel32,fpu
.data
mas1 DD 129.235, -1024.01, -12.5, 5.06, 67895.025
len1 EQU ($- mas1)/type mas1 ; вычисление количества чисел массива mas1
mas2 DD len1 DUP(0)          ; резервирование len1 ячеек памяти для mas2
_y1 dt 0.0
_y2 dt 0.0
_y3 dt 0.0
_y4 dt 0.0
_y5 dt 0.0
CrLf equ 0A0Dh
ttl db "Перемещение из mas1 в mas2",0
info db "y1 = "
     _res1 db 14 DUP(0),10,13
     db "y2 = "
```

```
            _res2 db 14 DUP(0),10,13
            db "y3 = "
            _res3 db 14 DUP(0),10,13
            db "y4 = "
            _res4 db 14 DUP(0),10,13
            db "y5 = "
            _res5 db 14 DUP(0),0
        info_sz = $ - info          ; количество байтов информации для info
.code
_start:
        finit                   ; инициирование сопроцессора
        mov ECX, len1  ; занесение len1 = 5 - количество байтов массива mas1
        lea ESI, mas1          ; занесение адреса начала массива
        lea EDI, mas2          ; занесение адреса начала массива
        next: fld  dword ptr [ESI] ; занесение в вершину стека из [ESI]
        fst dword ptr [EDI] ; занесение из вершины стека в память
                                                                ; по адресу, который находится в EDI
        add  ESI, 4         ; выравнивание на границу двойного слова массива mas1
        add  EDI, 4         ; выравнивание на границу двойного слова массива mas2
        loop  next       ; перейти, если в счетчике ecx не нуль
          fstp _y5
          fstp _y4
          fstp _y3
          fstp _y2
          fstp _y1
    invoke FpuFLtoA,offset _y1,3,offset _res1,SRC1_REAL or SRC2_DIMM
    mov word ptr _res1 + 14, CrLf
    invoke FpuFLtoA,offset _y2,2,offset _res2,SRC1_REAL or SRC2_DIMM
    mov word ptr _res2 + 14, CrLf
    invoke FpuFLtoA,offset _y3,1,offset _res3,SRC1_REAL or SRC2_DIMM
    mov word ptr _res3 + 14, CrLf
    invoke FpuFLtoA,offset _y4,2,offset _res4,SRC1_REAL or SRC2_DIMM
    mov word ptr _res4 + 14, CrLf
    invoke FpuFLtoA,offset _y5,3,offset _res5,SRC1_REAL or SRC2_DIMM
; в цикле просматривается строка на наличие нулей, которые в случае обнаружения
; заменяются на пробелы
            mov ecx, info_sz
            dec ecx     ; нужен последний нолик
            mov edi, offset info
            mov esi, edi
        @l: lodsb         ; al <- [ESI]
            test al,al
            jnz @f
            mov al,20h ; символ пробела
        @@: stosb         ; [DI] <=  al
        loop    @l
         invoke MessageBox, 0, addr info, addr ttl, MB_ICONINFORMATION
         invoke ExitProcess, 0 ; возврат управления  и освобождение ресурсов
        end _start
```

Результат виконання програми з лістингу 12.2 наведено на рис. 12.6.

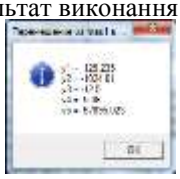


Рис. 12.6. Результат виконання програми

У останньому фрагменті програми показаний процес заміни всіх нулів в рядку чисел, які виводяться, на символи пропуску (20h). Це робиться для того, щоб MessageBox розділив числа, які виводяться.

Приклад 12.2. Заповнити масив конкретним значенням дійсного числа. Програму розв'язання прикладу 12.2 наведено у лістингу 12.3.

Лістинг 12.3. Програма розв'язання прикладу 12.2:

```
    .686            ; директива определения типа микропроцессора
    .model flat,stdcall ; задание линейной модели памяти и соглашения ОС
    option casemap: none     ; отличие строчных и прописных букв
    include \masm32\include\kernel32.inc ; системные функции
    includelib \masm32\lib\kernel32.lib
.data                                ; директива определения данных
    dig  DD  2.25                    ; число 2.25 з іменем dig  подвійного формату
    mas1 DD 8 DUP (12345678h)  ; першочергове заповнення масиву
    len  EQU $-mas1             ; обчислення кількості байтів масиву mas1
  .code                          ; директива начала кода программы
    start:                           ; метка начала программы
    finit         ; ініціювання співпроцесора
    mov  ECX, len   ; занесення кількості байтів масиву mas1 в ECX
shr ECX, 2   ; зсув праворуч на 2 розряди (ділення на 4 для визначення
                                                          ; кількості 32-розрядних чисел
    lea  ESI, mas1      ; занесення адреси початку масиву mas1
    fld  dword ptr dig   ; занесення 32-розрядного числа dig у вершину стека
next:
      fst dword ptr [ESI]     ; занесення з вершини стека в пам'ять по [ESI]
      add  ESI, 4      ; вирівнювання на межу подвійного слова масиву mas1
       dec  ECX          ; зменшення лічильника
       jnz  next         ; перейти, якщо ну лічильнику не нуль
       fincstp           ; очищення вершини стека
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
       end start         ; директива завершение программы с именем start
```

Програма з лістингу 12.2 виконує заповнення комірок пам'яті, які раніше (директивами визначення даних) були визначені. У цьому випадку вони були визначені константою 12345678h, яка в комірках пам'яті розташовується починаючи з молодших адрес даних.

У цих вікнах показано момент занесення константи у вершину стека (в регістр st(0)) командою fld dword ptr dig.

Приклад 12.3. Сложение всех єлементов массива:

```
      .686              ; директива определения типа микропроцессора
      .model flat,stdcall ; задание линейной модели памяти и соглашения ОС
      option casemap: none     ; отличие строчных и прописных букв
      include \masm32\include\windows.inc
      include \masm32\include\kernel32.inc
      include \masm32\include\fpu.inc
      includelib \masm32\lib\kernel32.lib
      include \masm32\include\user32.inc
      includelib \masm32\lib\user32.lib
      includelib \masm32\lib\fpu.lib
.data
```

```
        mas1 dd 0.,1.,2.,3.,4.,5.,6.,7.
      size1 equ ($-mas1)/type mas1 ; вычисление количества чисел массива mas1
       inf_res db "Сумма всех элементов массива:"
        res dt 0.,0
        titl db "FPU Result",0    ; титул окна
.code
_st:
      mov ecx,size1
      xor ebx,ebx
begin:
        finit
        fld mas1[ebx]
        fld res
        fadd st(0),st(1)
        fstp res
        add ebx,type mas1 ; 04h
      loop begin
        invoke FpuFLtoA,addr res,4,addr res,SRC1_REAL or SRC2_DIMM
        invoke MessageBox,0,addr inf_res,addr titl,MB_ICONINFORMATION
        invoke ExitProcess,0; возврат управления  ОС и освобождение ресурсов
            end _st
```

Результат выполнения программы приведено на рис. 12.7.

Рис. 12.7. Результат выполнения программы

Количество чисел массива вычисляется в переменной size1 еще на этапе компиляции. Оператор equ вычисляет разницу между текущим значением $ и началом массива mas1 в байтах. А операция деление в переменной size1 на type mas1 вычисляет число элементов массива в приведенной размерности, например в DD (на 4).

Особенностью программы является то, что при ее выполнении заполняется весь стек регистром и в командах вычисления не приняты меры для его очищения. Вот поэтому команда finit стоит в начале цикла, которая и очищает стек после выполнения каждого цикла.

В функции MessageBox выводится параметр inf_res, который не несет результат выполнения программы, а только выводит текстовую строку и не имеет признака окончания строки (т.е. «,0»). Следовательно, после выполнения строки inf_res выводится следующая строка res – пока не найдется строка с символами окончания строки (т.е. «,0»), где и находится переменная res с результатом, но с нулем на конце команды.

FXCH приймач	Поміняти місцями два регістри стека

Команда **FXCH** виконує обмін місцями вмісту регістра ST(0) і джерела (регістр ST(n)). Якщо операнд не вказаний, обмінюється вміст ST(0) і ST(1).

12.3.2. Команди порівняння

У центральному мікропроцесорі команди умовних переходів виконуються відповідно до встановлення окремих бітів регістра прапорців мікропроцесора. В арифметичному співпроцесорі існують спеціальні команди порівнянь, внаслідок виконання яких встановлюються біти кодів умов у регістрі стану.

FCOM джерело	Порівняти дійсні числа
FCOMP джерело	Порівняти і виштовхнути їх зі стека
FCOMPP джерело	Порівняти і виштовхнути зі стека два числа

- команда **fcom** порівнює вміст вершини стека st(0) з вмістом регістра st(1);
- команда **fcomp mem** порівнює вміст вершини стека st(0) з операндом пам'яті mem, після чого виштовхує число з st(0) і збільшує вказівник вершини стека на 1 в полі top регістра swr. Операнд mem може бути 32- або 64-розрядним числом з рухомою крапкою;
- команда **fcompp** порівнює вміст вершини стека st(0) з вмістом регістра st(1), після чого виштовхує зі стека значення, що знаходяться в регістрах st(0) і st(1).

Розглянуті вище команди виконують порівняння вмісту регістра ST(0) з джерелом (32- або 64-бітова змінна або регістр ST(n)) і встановлюють прапорці C0, C2 і C3 в регістрі swr (табл. 12.3).

Існує спосіб, за допомогою якого можна визначити результат порівняння операндів звичним способом, використовуючи бітові прапорці регістра прапорців мікропроцесора. Для цих цілей до складу системи команд співпроцесора включено команду **fstsw**, що дозволяє зберегти слово стану в регістрі AX або в елементі пам'яті. Командою **sahf** вміст регістра AH записується в молодший байт регістра прапорців. При цьому біт c0 поміщається на місце прапорця CF, біт c2 – на місце прапорця PF, біт c3 – на місце ZF, а біт c1 не використовується.

Таблиця 12.3

c3	c2	c1	c0	Значення
0	0	?	0	st(0) більше операнда
0	0	?	1	st(0) менше операнда
1	0	?	0	st(0) дорівнює операнду
1	1	?	1	st(0) та операнд непорівнянні

FSTSW	Зберегти слово стану в регістрі AX

Команда **fstsw** зберігає слово стану в регістрі AX для подальшого отримання ознак (прапорців) виконання операції.

Приклад 12.3. Знайти максимальне число з двох дійсних чисел та зберегти його у пам'яті.

Програму розв'язання прикладу 12.3 наведено у лістингу 12.3.

Лістинг 12.3. Програма розв'язання прикладу 12.3:

```
        .686                        ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС
option casemap: none            ; отличие строчных и прописных букв
 include  \masm32\include\kernel32.inc ; системные функции применений…
 includelib \masm32\lib\kernel32.lib
.data                               ; директива определения данных
        dig1  DD 2.25         ; подвійне число 2,25 з іменем dig1
        dig2  DD 2.26         ; подвійне число 2,26 з іменем dig2
        res    DD 0           ; резервування пам'яті з іменем res
.code                               ; директива начала кода программы
  start:                            ; метка начала программы
      finit                         ; ініціювання співпроцесора
fld dword ptr dig1   ; занесення в вершину стека 32-розрядного числа dig1
fld dword ptr dig2 ; занесення в вершину стека 32-розрядного числа dig2
fcom    ; порівняння вмісту вершини стека st(0) з вмістом регістра st(1)
fstsw AX                ; зберегти слово стану в регістрі AX
sahf                        ; завантажити прапорці стану з AH
        jnc m1              ; перейти, якщо CF = 0    [st(0) > st(1)]
        fxch st(1)          ; обміняти місцями вершину стека st(0) з st(1)
        fstp res  ; зберегти вершину стека st(0) в комірці пам'яті з ім'ям res
        jmp m2  ; безумовний перехід на m2
m1: fstp res   ; зберегти вершину стека st(0) в комірці пам'яті з ім'ям res
m2:    invoke ExitProcess, 0 ; возврат управления  та вивільнення ресурсів
        end start               ; директива завершение программы с именем start
```

Послідовність команд **fcom – fstsw AX – sahf** у теперішній час **вважається застарілою** і не використовується. Замість неї використовується команда **fcomi**, яка буде розглянута пізніше.

У вікнах налагоджувача видно (рис. 12.6), що перше початкове число розміщується у комірці пам'яті за номером 0040300h, друге число – за номером 00403004h, а результат виконання програми – за номером 00403008h.

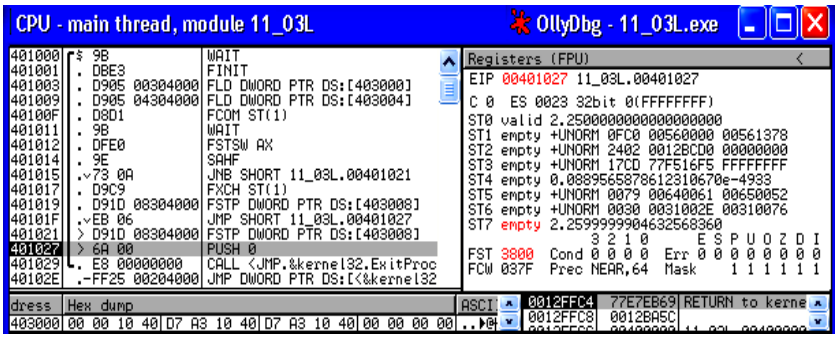


Рис. 12.6. Вікна налагоджувача при виконанні програми з лістингу 12.3

Значення регістрів стека ST0 – ST7 наведено у вікні FPU.

Після виконання програми стек не було очищено. Тому якщо оформлювати програму у вигляді функції, то необхідно потурбуватися про очищення стека у самій функції.

FICOM джерело	Порівняти цілі числа
FICOMP джерело	Порівняти цілі числа і виштовхнути зі стека

- Команда **ficom** – порівнює цілочислові значення st(0) і st(1);
- Команда **ficomp mem** – порівнює вміст вершини стека st(0) і операнда пам'яті mem, після чого виштовхує число з st(0) і збільшує вказівник вершини стека на 1 в полі top регістра swr. Операнд mem може бути 16- або 32-розрядним цілим числом.

FTST	Перевірити st(0) з нулем і встановити прапорці c3, c2 і c0

Після виконання команди **ftst** коди умов встановлюються відповідно до табл. 12.4.

Таблиця 12.4

c3 → zf	c0 → cf	Умова
0	0	st(0) > 0
0	1	st(0) < 0
1	0	st(0) = 0
1	1	st(0) та 0 не порівнюються

FXAM	Проаналізувати вміст ST(0)

Команда **fxam** – аналізує операнд, що знаходиться у вершині стека співпроцесора і за результатом такого аналізу встановлює біти c0-c3 спеціальним способом (табл. 12.5).

Таблиця 12.5

Тип числа	c3	c2	c0
Непідтримуване	0	0	0
Не число	0	0	1
Нормальне скінченне число	0	1	0
Нескінченність	0	1	1
Нуль	1	0	0
Регістр порожній	1	0	1
Денормалізоване число	1	1	0

FCOMI st(0), джерело	Порівняти st(0) та джерело й встановити EFLAGS

FCOMIP st(0), джерело	Порівняти, встановити EFLAGS і виштовхнути число із вершини стека
FUCOMI st(0), джерело	Порівняти без урахування порядків і встановити EFLAGS
FUCOMIP st(0), джерело	Порівняти без урахування порядків, встановити EFLAGS і виштовхнути число із вершини стека

Починаючи з мікропроцесора Intel Pentium Pro, група команд порівняння поповнилася двома новими командами: **fcomi** і **fcomip**. Ці команди виконують ті самі операції, що і fcom і fcomp, але на відміну від останніх не вимагають спеціальних команд для обробки бітів c0 – c3 регістра swr, а встановлюють прапорці ZF, CF і PF безпосередньо в регістрі прапорців EFLAGS центрального мікропроцесора.

Команди fcomi і fcompi порівнюють вміст вершини стека st(0) з вмістом регістра st(1) та ***безпосередньо*** встановлюють прапорці ZF, CF і PF в регістрі прапорців EFLAGS центрального мікропроцесора відповідно до табл. 12.6. Команди записуються, наприклад: fcomi st(0),st(1).

Таблиця 12.6

Умови	ZF	PF	CF
ST(0) > джерело	0	0	0
ST(0) < джерело	0	0	1
ST(0) = джерело	1	0	0
Непорівнянні	1	1	1

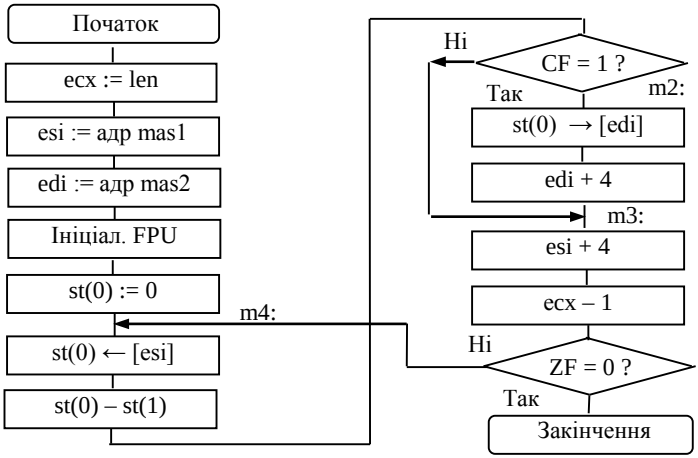
fucom st(i)	Порівняння невпорядкованих чисел в регістрах st(0) і st(i)
fucomp st(i)	Порівняння невпорядкованих чисел в регістрах st(0) і st(i) з виштовхуванням числа з вершини стека
fucompp st(i)	Порівняння невпорядкованих чисел в регістрах st(0) і st(i) з виштовхуванням чисел з регістрів st(0) і st(1)

Команди використовуються для того щоб виконати операції над некоректними цілими або дійсними числами та встановити прапорці згідно з табл. 12.6.

Приклад 12.4. Знайти в масиві подвійних чисел від'ємні елементи й записати їх в новий масив.

Алгоритм розв'язання прикладу 12.4 наведено на рисунку.

Першою операторною вершиною є вершина для збереження кількості байтів масиву в регістрі ecx, яка раніше була обчислена директивою len equ $-mas1. Визначення кількості 32-розрядних чисел виконується за допомогою зсуву на 2 розряди, що дорівнює діленню на 4 (через те що 32-розрядне число містить 4 байти).



Програму розв'язання прикладу 12.4 наведено у лістингу 12.4.

Лістинг 12.4. Програма розв'язання прикладу 12.4:

```
        title CopyRight by Rysovaniy A. N.
        .686                ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
option casemap: none        ; отличие строчных и прописных букв
include \masm32\include\kernel32.inc ; системные функции
includelib \masm32\lib\kernel32.lib
.data                                   ; директива определения данных
        mas1  DD -12.25, 25.05, -8.1, 12.01     ; визначення в пам'яті масиву
        len equ ($-mas1)/4          ; обчислення кількості чисел масиву mas1
        mas2 DD len dup(0)          ; резервування комірок пам'яті для mas2
        dig DD 0. ; значення для порівняння
.code               ; директива начала кода программы
start:              ; метка начала программы
        mov ecx,len             ; занесення кількості байтів масиву mas1
        lea esi,mas1            ; занесення адреси початку масиву mas1
        lea edi,mas2            ; занесення адреси початку масиву mas2
        finit                   ; ініціювання співпроцесора
        fld dig
m4: fld dword ptr [esi]       ; занесення у вершину стека числа з [esi]
        fcomi st(0),st(1)
        jc m2                       ; перехід на m2, якщо є перенесення
        jmp m3                      ; безумовний перехід на m3
m2: fst dword ptr [edi]  ; занесення з вершини стека в пам'ять
                                                    ; за адресою, яка знаходиться в EDI
        add edi,4      ; вирівнювання на межу подвійного слова масиву mas2
m3: add esi,4  ; вирівнювання на межу подвійного слова масиву mas1
        loop m4
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start               ; директива завершение программы с именем start
```

Програму з лістингу 12.4 складено так, щоб після її завершення у регістрах FPU залишилися числа за порядком їх виклику, щоб можна було всі числа побачити у стеку.

FCMOVCC	Копіювання st(i) в st(0), якщо виконується умова

У команді **fcmovCC** префікс CC – мнемонічне позначення коду умови (b, nb, e, ne, be, nbe). Ця команда виконує копіювання регістра st(i) в регістр st(0), якщо виконується вказана в ній умова.

Формати команд і аналізовані ними прапорці наведено в табл. 12.7.

Приклад 12.5. Виконати попарне порівняння двох масивів дійсних чисел розміром у 4 подвійних слова та записати більше число у новий масив.

Таблиця 12.7

Мнемоніка	Прапорці	Умова встановлення прапорця
fcmovb ST,ST(i)	CF = 1	Якщо менше
fcmovnb ST,ST(i)	CF = 0	Якщо не менше
fcmove ST,ST(i)	ZF = 1	Якщо дорівнює
fcmovne ST,ST(i)	ZF = 0	Якщо не дорівнює
fcmovbe ST,ST(i)	(CF или ZF) = 1	Якщо менше або дорівнює
fcmovnbe ST,ST(i)	(CF или ZF) = 0	Якщо не менше або не дорівнює
fcmovu ST,ST(i)	PF = 1	Якщо нерівнянні
fcmovnu ST,ST(i)	PF = 0	Якщо порівнянні

Програму розв'язання прикладу 12.5 наведено у лістингу 12.5.

Лістинг 12.5. Програма розв'язання прикладу 12.5:

```
        .686                ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap: none            ; отличие строчных и прописных букв
        include  \masm32\include\kernel32.inc ; системные функции
        includelib \masm32\lib\kernel32.lib
.data                                   ; директива определения данных
   mas1 DD -25.25, 10.05, -10., 1.01 ; визначення в пам'яті масиву mas1
   len equ $-mas1                   ; обчислення кількості байтів масиву mas1
   mas2 DD -2., 10.06, -21., 2.     ; визначення в пам'яті масиву mas2
   mas3 DD  len  dup(0)             ; резервування комірок пам'яті
.code                                   ; директива начала кода программы
start:                                  ; метка начала программы
      mov ecx,len                       ; занесення кількості байтів масиву mas1
      shr ecx,2       ; ділення на 4 – визначення кількості 32-розрядних чисел
      lea esi,mas1      ; занесення адреси початку масиву mas1
      lea edi,mas2      ; занесення адреси початку масиву mas2
      lea edx,mas3       ; занесення адреси початку масиву mas3
      finit                         ; ініціювання співпроцесора
m4:
      fld dword ptr [esi]         ; занесення в вершину стека числа з [esi]
      fld dword ptr [edi]         ; занесення в вершину стека числа з [edi]
      fcomi st(0),st(1)             ; порівняння та встановлення eflags
      fcmovb st(0),st(1)    ; копіювання st(0) ← st(1), якщо CF = 1. Інакше – ні
      fst dword ptr [edx]   ; занесення у пам'ять (у масив mas3) більшого числа
      add esi,4     ; вирівнювання на межу подвійного слова масиву mas1
      add edi,4     ; вирівнювання на межу подвійного слова масиву mas2
      add edx,4     ; вирівнювання на межу подвійного слова масиву mas3
```

```
    loop m4      ; перейти на  m4, якщо ecx ≠ 0
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
       end start          ; директива завершение программы с именем start
```

У програмі команду len equ $-mas1 потрібно завжди використовувати після тієї директиви визначення даних, яка обчислюється. Якщо цю команду в програмі поставити після, наприклад, команди визначення mas2, то незалежно від позначення змінних буде обчислено кількість байтів вже двох масивів, що є невірним у даному випадку.

Команда fcmovb st(0),st(1) спочатку порівнює числа за допомогою віднімання st(0) – st(1), а потім вже копіює st(0) ← st(1), якщо st(0) < st(1).

Доступ до елементів трьох масивів виконується за допомогою регістрів esi, edi та edx з використанням непрямої адресації.

Як лічильник 32-розрядних чисел використовується регістр ecx, що дає можливість використати команду loop.

Masm32-функції порівняння (FpuComp)

Вбудована masm32-функція **FpuComp** виконує порівняння двох чисел та має такий прототип:

```
FpuComp (
lpSrc1  // вказівник (адреса) на 1-е число
lpSrc2  // вказівник (адреса) на 2-е число
uID     // параметр джерел та формату
        )
```

Вказівники можуть мати формати REAL10 або DWORD. Порівняння виконують із константами FPU:

Параметр *uID* джерел та формату може бути таким:

SRC1_FPU – Src1 є вже на FPU;

SRC1_REAL – вказівник на 80-розрядне дійсне число в пам'яті;

SRC1_DMEM – вказівник на 32-розрядне ціле число із знаком;

SRC1_DIMM – Src1 є 32-розрядним цілим числом із знаком;

SRC1_CONST – одна із спеціальних констант FPU

Для вказівника SRC2 значення параметра аналогічні.

Порівняння може виконуватися з константами (файл Fpu.inc):

```
CMP_EQU         Src1 = Src2
CMP_GREATER     Src1 > Src2
CMP_LOWER       Src1 < Src2
```

Наприклад:

```
invoke FpuComp,0,10,SRC1_FPU or SRC2_DIMM
```

У цьому прикладі перше число знаходиться в вершині стека FPU (St(0)) та порівнюється з безпосереднім значенням 10.

12.3.3. Арифметичні команди

До групи **арифметичних** команд належать команди, які реалізують операції складання, віднімання, множення і ділення. У свою чергу, арифметичні команди поділяються на дві підгрупи для роботи з цілочисловими і дійсними операндами.

Мнемоніки базових арифметичних команд наведено в табл. 12.8.

Співпроцесор використовує шість основних типів арифметичних команд, які наведено у табл. 12.9.

Таблиця 12.8

Команди	Основні	Ціле у пам'яті	З вилученням
Додавання	fadd	fiadd	faddp
Віднімання	fsub	fisub	fsubp
Зворотне віднімання	fsubr	fisubr	fsubrp
Множення	fmul	fimul	fmulp
Ділення	fdiv	fidiv	fdivp
Зворотне ділення	fdivr	fidivr	fdivrp

Команди **додавання** для роботи з дійсними числами складання мають такий синтаксис:

fadd *приймач, джерело*
faddp *приймач,* st
fadd *джерело*

Тут *приймач* – операнд-приймач, а *джерело* – операнд-джерело. Якщо обидва операнди є регістрами стека, то одним із них обов'язково має бути st(0). Якщо єдиним операндом є елемент пам'яті (32 або 64 розряди), то другим операндом буде регістр st(0), в якому і зберігається результат.

Рядок «xxx» може набувати значень, які наведено в табл. 12.8.

Таблиця 12.9

Команда	Опис
Fxxx	Перший операнд береться з ST(0) (джерело); другий операнд є наступним елементом стека. Результат виконання команди записується в стек
Fxxx пам'ять	Джерело береться з пам'яті, приймачем є ST(0). Вказівник стека ST не змінюється, команда дійсна тільки для операндів з одинарною і подвійною точністю
Fixxx пам'ять	Аналогічно до попереднього типу команди, але операндами можуть бути 16- або 32-розрядні цілі числа

Fxxx ST, ST(i)	Для цього типу регістр ST(i) є джерелом, а ST(0) – приймачем. Вказівник стека не змінюється
Fxxx ST(i), ST	Для цього типу регістр ST(0) є джерелом, а ST(i) – приймачем. Вказівник стека не змінюється
FxxxP ST(i), ST	Регістр ST(i) – приймач, регістр ST(0) – джерело. Після виконання команди джерело ST(0) витягується із стека

Далі наведено докладніший опис команд додавання:

- **fadd** – додавання вмісту вершини стека st(0) і регістра st(i); результат складання зберігається в регістрі st(0). *При використанні команди без параметрів відбувається очищення вершини стека*;
- **fadd *джерело*** – додавання вмісту вершини стека st(0) і елемента пам'яті *джерело;* результат складання зберігається в регістрі st(0);
- **fadd st, st(i)** – додавання вмісту регістра стека st(i) і вершини стека st(0); результат зберігається в регістрі st(0);
- **fadd st(i), st** – додавання вмісту регістра стека st(i) і вершини стека st(0); результат зберігається в регістрі st(i);
- **faddp st(i), st** – додавання вмісту регістра стека st(i) і вершини стека st(0), з наступним виштовхуванням значення з вершини стека; результат операції зберігається в регістрі st(i – 1).

Приклади команд додавання:

```
fadd mem1                   ; ST(0) ← ST(0) + mem1
fadd dword ptr [eax]        ; ST(0) ← ST(0) +  REAL4 [EAX]
fadd st,st                  ; ST(0) ← ST(0) +  ST(0)
fadd st,st(4)               ; ST(0) ← ST(0) +  ST(4)
fadd st(5),st               ; ST(5) ← ST(5) +  ST(0)
faddp st(1),st              ; ST(0 ← 1) ← ST(1) +  ST(0)
                                ; додавання та вивільнення вершини стека ST(0)
```

При ***зворотних операціях*** операнди міняються місцями. Наприклад, при зворотному діленні DIVR частка і дільник міняються місцями.

Команди для роботи з цілочисловими операндами маніпулюють даними розміром в 16 і 32 біти і включають такі операнди:

- **fiadd *операнд*** – складання вмісту вершини стека st(0) з цілочисловим операндом розміром 16 або 32 біти; результат складання зберігається в регістрі st(0);
- **fisub *операнд*** – віднімання цілочислового операнда розміром 16 або 32 біти з вмісту вершини стека st(0); результат віднімання зберігається в регістрі st(0);
- **fimul *операнд*** – множення вмісту вершини стека st(0) на цілочисловий операнд розміром 16 або 32 біти; результат множення зберігається в регістрі st(0);
- **fdiv *операнд*** – ділення вмісту вершини стека st(0) на цілочисловий операнд розміром 16 або 32 біта; результат ділення зберігається в регістрі

st(0);

- **fIsubr *операнд*** – віднімання вмісту регістра st(0) з цілочислового операнда розміром 16 або 32 біти; результат віднімання зберігається в регістрі st(0);
- **fidivr *операнд*** – ділення цілочислового операнда розміром 16 або 32 біти на вміст вершини стека st(0); результат ділення зберігається в регістрі st(0).

Команди **віднімання** мають такий синтаксис:

fsub *приймач, джерело*
fsubp *приймач,* st
fsub *джерело*

Тут *приймач* – операнд-приймач, а *джерело* – операнд-джерело. Команди виконують віднімання вмісту операнда-джерела з вмісту операнда-приймача, зберігаючи результат в операнді-приймачі. Якщо обидва операнди є регістрами стека, то одним із них обов'язково повинен бути st(0). Якщо одним операндом є елемент пам'яті (32 або 64 розряди), то другим операндом буде регістр st(0), в якому і зберігається результат.

Віднімання чисел із рухомою крапкою виконується такими командами:

- **fsub** – віднімання вмісту регістра st(0) з вмісту st(1); результат віднімання зберігається в регістрі st(0): **[st(0) := st(1) – st(0)]**;
- **fsub *джерело*** – віднімання вмісту операнда *джерело,* розташованого в пам'яті, з вмісту вершини стека st(0); результат віднімання зберігається в регістрі st(0);
- **fsub st, st(i)** – віднімання вмісту регістра st(i) з вмісту регістра стека st(0); результат зберігається в регістрі st(0);
- **fsub st(i), st** – віднімання вмісту регістра стека st(0) з вмісту будь-якого з регістрів st(i); результат – в регістрі st(i): **[st(i) := st(i) – st(0)]**;
- **fsubp st(i), st** – віднімання вмісту регістра стека st(0) з вмісту будь-якого з регістрів st(i) із наступним виштовхуванням значення з вершини стека; результат операції зберігається в регістрі st(i – 1);
- **fsubr st(i), st** – віднімання вмісту регістра стека st(i) з вмісту регістра st(0); результат зберігається в регістрі st(0): **[st(0) := st(0) – st(i)]**;
- **fsubrp st(i), st** – віднімання вмісту регістра стека st(0) з вмісту будь-якого з регістрів st(i), із наступним виштовхуванням значення з вершини стека; результат операції зберігається в регістрі st(i).

Команди **множення** мають такий синтаксис:

fmul *приймач, джерело*
fmulp *приймач,* st
fmul *джерело*

Тут *приймач* – операнд-приймач, а *джерело* – операнд-джерело. Команди цієї групи виконують множення операнда-приймача на операнд-джерело, зберігаючи результат в операнді-приймачі. Якщо як операнди вказано

регістри стека співпроцесора, то один із них повинен бути st(0). Якщо як єдиний операнд виступає елемент пам'яті, то другим операндом за умовчанням є регістр st(0), що зберігає результат операції. Операнди, подані змінними в пам'яті, можуть бути 32- або 64-розрядними.

До цієї групи включено такі команди:

- **fmul** – множення вмісту регістра вершини стека st(0) на вміст регістра st(1); результат зберігається в регістрі st(0), st(1) заповнюються нулями;
- **fmul st, st(i)** – множення регістра st(0) на вміст регістра st(i); результат зберігається в регістрі st(0);
- **fmul st(i), st** – множення вмісту регістра st(0) на вміст одного з регістрів st(i); результат зберігається в регістрі st(i);
- **fmulp st(i), st** – множення вмісту регістра st(0) на вміст одного з регістрів **st(i)** з наступним виштовхуванням вмісту вершини стека st(0); результат множення залишається в регістрі st(i – l).

Приклади команд множення:

```
fmul
fmul st(i),st(0)
fmul st(0),st(i)
fmul mem
fmulp
fimul mem

fmul real8_var              ; ST(0) ← ST(0) × real8_var
fmul dword ptr [eax]        ; ST(0) ← ST(0) ×  REAL4 [EAX]
fmul qword ptr [esi+8]      ; ST(0) ← ST(0) × REAL8 [ESI+8]
fmul st,st                  ; ST(0) ← ST(0) ×  ST(0)
fmul st,st(5)               ; ST(0) ← ST(0) ×  ST(5)
fmul st(4),st               ; ST(4) ← ST(4) ×  ST(0)
```

Команди **ділення** мають такий синтаксис:

fdiv *приймач, джерело*
fdivp *приймач,* st
fdiv *джерело*

Тут ***приймач*** – операнд-приймач, а ***джерело*** – операнд-джерело. Команди цієї групи виконують ділення операнда-приймача на операнд-джерело, зберігаючи частку в операнді-приймачі. Якщо як операнди вказано регістри стека співпроцесора, то один із них повинен бути st(0). Якщо як єдиний операнд виступає елемент пам'яті, то другим операндом за умовчанням є регістр st(0), в якому зберігається результат операції. Операнди, подані змінними в пам'яті, можуть бути 32- або 64-розрядними.

До цієї групи включено такі команди:

- **fdiv** – ділення вмісту регістра st(1) на значення, що знаходиться у вершині стека st(0); результат операції зберігається в регістрі st(0);
- **fdiv st, st(i) –** ділення вмісту регістра st(0) на значення, що знаходиться в регістрі st(i); результат операції зберігається в регістрі st(0);
- **fdiv st(i), st** – ділення вмісту регістра st(i) на значення, що знаходиться в регістрі st(0); результат операції зберігається в регістрі st(i);

- **fdivp st(i), st** –ділення вмісту регістра st(i) на значення, що знаходиться в регістрі st(0), з наступним виштовхуванням вмісту вершини стека; результат операції поміщається в регістр st(i – 1);
- **fdivr st(i), st** – ділення вмісту регістра st(i) на значення, що знаходиться у вершині стека st(0); результат операції поміщається в регістр st(0);
- **fdivrp st(i), st** – ділення вмісту регістра st(i) на значення, що знаходиться у вершині стека st(0); результат операції поміщається в регістр st(i), після чого вміст st(0) виштовхується із стека, а результат ділення залишається в регістрі st(i – 1).

Приклад 12.6. Обчислити вираз $ab - cd$, де $a = 0{,}1$; $b = 10{,}05$; $c = 100$; $d = 0{,}5$.

Програму обчислення прикладу 12.6 наведено у лістингу 12.6.

Лістинг 12.6. Програма обчислення прикладу 12.6:

```
    title CopyRight by Rysovaniy A. N.
    .686                ; директива определения типа микропроцессора
    .model  flat, stdcall  ; задание линейной модели памяти  и соглашения ОС Windows
    option casemap:none     ; отличие строчных и прописных букв
    include \masm32\include\windows.inc ; файлы структур, констант …
    include \masm32\include\kernel32.inc ; системные функции применений…
    include    \masm32\include\fpu.inc
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
    includelib \masm32\lib\fpu.lib
      BSIZE equ  30   ; розмір буфера
  .data                             ; директива определения данных
    a1   DWORD  0.1     ; збереження у 32-розрядній комірці змінної a1
    b1   DWORD  10.05  ; збереження у 32-розрядній комірці змінної b1
    a2   DWORD  100.0  ; збереження у 32-розрядній комірці змінної a2
    b2   DWORD  0.5     ; збереження у 32-розрядній комірці змінної b2
stdout DWORD ?       ; резервування в пам'яті 32-розрядної комірки
                                    ; з ім'ям stdout для збереження дескриптора виведення
cWritten  DWORD  ? ; резервування 32-розрядної комірки пам'яті
                                                ; з ім'ям cWritten для адреси символів виведення
buf BYTE BSIZE dup (?)   ; кількість байтів для запису послідовності символів
  .code                             ; директива начала программы
  start:                            ; метка начала программы с именем start
     main proc                ; процедура з іменем main
       finit                    ; ініціювання співпроцесора
       fld    a1      ; занесення в вершину стека a1: st(0) := a1
       fld    b1      ; занесення в вершину стека b1: st(0) := b1, st(1) := a1
       fmul             ; st(0) := b1 × a1
       fld    a2      ; занесення a2; st(0) := a2, st(1) := b1 × a1
       fld    b2      ; занесення b2: st(0) := b2, st(1) := a2, st(2) := b1 × a1
       fmul             ; st(0) := b2 × a2, st(1) := b1 × a1
       fsub             ; st(0) := st(1) – st(0)
   invoke  GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
   mov stdout, eax      ; збереження одержаного дескриптора у пам'яті
invoke  FpuFLtoA, \  ; функція перетворення 80-розрядного числа
  0, \                  ; адреса числа, що відображається
     10, \               ; кількість десяткових знаків після коми (10)
      ADDR buf, \        ; адреса буфера для символів, які перетворяться
      SRC1_FPU or SRC2_DIMM  ; 1-е число – в FPU, результат – число
invoke  WriteConsoleA, \  ; API-функція виведення на екран
```

```
 stdout, \              ; дескриптор стандартного пристрою виведення
 ADDR buf, BSIZE \ ; адреса початку повідомлення та розмір повідомлення
     ADDR cWritten, 0  ; адреса, де зберігається кількість символів
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        main endp          ; закінчення процедури з іменем main
        end start          ; директива окончания программы з іменем start
```

Результат виконання програми 48.9949999660

Процедура FpuFLtoA підготовляє 80-розрядне число до виведення на екран: розшифровує і записує в буфер послідовність символів із десятьма знаками після коми. Ця процедура входить до складу бібліотеки fpu.lib.

У процедури FpuFLtoA чотири параметри:

– адреса числа, що відображається;

– кількість десяткових знаків після коми (10);

– адреса буфера, де опиняться символи, на які перетвориться число;

– константи, процедури, що керують роботою. Константа SRC1_**REAL** передає функції, що її перший параметр – це адреса 80-бітового числа, яка зберігається в звичайній **пам'яті**. Константа SRC2_DIMM вказує функції, що другий її параметр – просто число (10).

Приклад 12.7. Обчислити вираз $a - b/4 - ab$, де $a = 0{,}1$; $b = 10{,}05$.

Програму обчислення прикладу 12.7 наведено у лістингу 12.7.

Лістинг 12.7. Програма обчислення прикладу 12.7:

```
    title CopyRight by Rysovaniy A. N.
    .686                          ; директива определения типа микропроцессора
    .model flat, stdcall ; задание линейной модели памяти  и соглашения ОС Windows
    option casemap:none                ; отличие строчных и прописных букв
    include C:\masm32\include\windows.inc   ;
    include C:\masm32\include\kernel32.inc  ;
    include C:\masm32\include\\user32.inc   ;
    include C:\masm32\include\fpu.inc       ;
    includelib C:\masm32\lib\kernel32.lib   ;
    includelib C:\masm32\lib\user32.lib     ;
    includelib C:\masm32\lib\fpu.lib
    BSIZE equ  30      ; розмір буфера
.data
    a1 DWORD  0.1         ; збереження у 32-розрядній комірці змінної a1
    b1 DWORD 10.05        ; збереження у 32-розрядній комірці змінної b1
    c1 DWORD  4.          ; збереження у 32-розрядній комірці змінної c1
    stdout DWORD ?        ; для збереження дескриптора виведення
    cWritten  DWORD  ?    ; комірка для адреси символів виведення
    _title db 'НТУ ХПИ.  Результат выполнения программы',0
    inf1 db 'Уравнение: a – b/4 – ab, где a = 0,1;  b = 10,05 ',0ah,0dh,0dh,
    'Результат:  '
    buf dt ?,0 ; кількість байтів для запису символів
.code
start:
main proc              ; процедура з іменем main
   finit       ; ініціювання співпроцесора
   fld    b1                    ; занесення в вершину стека b1
   fld    c1                    ; занесення в вершину стека числа 4
   fdiv                  ; st(0) := st(1)/st(0)
   fld    a1                    ; занесення в вершину стека a1
```

```
fsub st(0),st(1)  ; st(0) := st(0) – st(1)
fld a1                          ; занесення в вершину стека a1
fld b1                          ; занесення в вершину стека b1
fmul                  ; st(0) := st(0) ? st(1)
fsubp st(1),st(0) ; st(0)  ? st(1) := st(1) – st(0)
invoke  GetStdHandle, STD_OUTPUT_HANDLE
mov stdout, eax          ; збереження одержаного дескриптора у пам'яті
invoke FpuFLtoA,0,10,ADDR buf, SRC1_FPU or SRC2_DIMM
invoke WriteConsoleA,stdout, ADDR buf,BSIZE, ADDR cWritten,0;
invoke MessageBox, 0, addr inf1, addr _title, MB_ICONINFORMATION
invoke ExitProcess,0 ; возврат управления  ОС и освобождение ресурсов
main endp            ; закінчення процедури з іменем main
end start
```

Результат виконання програми з лістингу 12.7 наведено на рис. 12.7. Він дорівнює –3,4175.

Рис. 12.7. Результат виконання програми з лістингу 12.7

Окрім основних арифметичних команд, є ***додаткові арифметичні команди:***

- **fsqrt** – обчислення квадратного кореня;
- **fscale** – масштабування на степінь числа 2;
- **fprem** – обчислення часткової остачі;
- **frndint** – округлення до цілого;
- **fxtract** – виділення порядку числа та мантиси;
- **fabs** – обчислення абсолютної величини числа;
- **fchs** – зміна знака числа.

Команда fsqrt обчислює значення квадратного кореня числа, яке було записане у вершину стека st(0), та змінює його. Розглянемо використання цієї команди.

Приклад 12.8. Обчислити функцію $S = \sqrt{p \times (p - A)\,(p - B)\,(p - C)}$, де p – півпериметр трикутника.

Лістинг 12.8. Програма виконання прикладу 12.8:

```
    .686                ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти  и соглашения ОС Windows
option casemap: none          ; отличие строчных и прописных букв
include \masm32\include\kernel32.inc ; системные функции применений…
includelib \masm32\lib\kernel32.lib
.data                           ; директива определения данных
    opA  dd  5.6            ; визначення в 32-розрядній комірці операнда A
    opB  dd  6.1            ; визначення в 32-розрядній комірці операнда B
    opC  dd  7.5            ; визначення в 32-розрядній комірці операнда C
    const  dd  2            ; комірка для const при обчисленні півпериметра
    res  dd ?                ; ячейка для хранения результата
.code                     ; директива начала кода программы
_st:                        ; метка начала программы
```

```
    finit                   ; ініціювання співпроцесора
    fld opA          ; st(0) := opA
    fld opB          ; st(0) := opB, st(1) := opA
    fld opC          ; st(0) := opC, st(1) := opB, st(2) := opA
    fld st(0)        ; повторення (місце для виконання операцій)
                     ; st(0) := opC, st(1) := opC, st(2) := opB, st(3) := opA
; обчислення периметра
    fadd st(0),st(2)  ; st(0) := opC + opB
    fadd st(0),st(3)  ; st(0) := (opC + opB) + opA
    fidiv const                 ; обчислення півпериметра: st(0) := st(0)/2
    fsubr st(1),st              ; st(1) := st(0) – st(1),  (p – C)
    fsubr st(2),st              ; st(2) := st(0) – st(2),  (p – B)
    fsubr st(3),st              ; st(3) := st(0) – st(3),  (p – A)
    fmulp st(1),st              ; st(0) := st × st(1), p(p – C)
    fmulp st(1),st                ; p(p – C)(p – B)
    fmulp st(1),st                ; p(p – C)(p – B)(p – A)
    fsqrt                       ; обчислення кореня
    fstp res                    ; збереження у пам'яті результату
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
      end _st                   ; директива окончания программы
```

У наведеній програмі команда fld st(0) необхідна Для того щоб у вершині стека регістрів виконувати тимчасові операції без порушення початкових значень операндів. А для коректного виконання ділення при обчисленні півпериметра слід використати команду цілочислового ділення fidiv Per2.

При іншій мнемоніці коду операції результат буде невірним. Крім того, при використанні API-функцій обов'язково необхідно використати умову stdcall.

Приклад 12.9. Обчислити корені квадратного рівняння

$$x_{1,2} = -p/2 \pm \sqrt{p^2/4 - q}.$$

Лістинг 12.9. Програма виконання прикладу 12.9:

```
      .686              ; директива определения типа микропроцессора
.model  flat, stdcall ; задание линейной модели памяти  и соглашения ОС Windows
option casemap:none       ; отличие строчных и прописных букв
include    \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\kernel32.inc ; системные функции применений…
      include    \masm32\include\fpu.inc
      includelib \masm32\lib\user32.lib
      includelib \masm32\lib\kernel32.lib
      includelib \masm32\lib\fpu.lib
      BSIZE equ  30                   ; розмір буфера
.data                   ; директива определения данных
p DWORD 8.0  ; визначення 32-розрядної комірки для збереження p = 8.0
q  DWORD   5.0
two  DWORD  -2.0
    x1      TBYTE   ?       ; 80-розрядна комірка пам'яті для x1
    x2      TBYTE   ?       ; 80-розрядна комірка пам'яті для x2
    stdout DWORD ?      ; резервування в пам'яті 32-розрядної комірки
                                ; з ім'ям stdout для збереження дескриптора виведення
    cWritten  DWORD  ?  ; резервування 32-розрядної комірки пам'яті
                                    ; з ім'ям cWritten для адреси символів виведення
buf BYTE BSIZE dup (?) ; кількість байтів для запису послідовності символів
.code                           ; директива начала программы
start:                          ; метка начала программы с именем start
```

```
        finit                       ; ініціювання співпроцесора
        fld p                       ; st = p
        fld  two                    ; st = -2, st(1) = p
        fdiv                        ; st(1) = p/2
        fld ST                      ; st(0) = st(1) = p/2
        fmul                        ; st(1) = st(1) × st(0) = p2/4
        fld q                       ; st(0) = q
        fsub                        ; st(0) = (p2/4) – q
        fsqrt                       ; √(p2/4) – q)
        fld p                       ; st(0) = p
        fld two                     ; -2.0
        fdiv                        ; st(0) = – p/2
        fld  st                     ; st(1) = st(0)
        fsub st, st(2)  ; st = – p/2  – √(p2/4) – q)
        fstp  x1                    ; зберегти корінь
        fadd st, st(1)  ; st = – p/2 + √(p2/4) – q)
        fstp  x2                    ; зберегти корінь
invoke GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
        mov stdout, eax   ; збереження одержаного дескриптора у пам'яті
invoke  FpuFLtoA, ADDR x1, 10,  ADDR buf, SRC1_REAL or SRC2_DIMM
invoke  WriteConsoleA, stdout, ADDR buf, BSIZE, ADDR cWritten, 0
invoke  FpuFLtoA, ADDR x2, 10,  ADDR buf, SRC1_REAL or SRC2_DIMM
invoke  WriteConsoleA, stdout, ADDR buf,  BSIZE, ADDR cWritten, NULL
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start       ; директива окончания программы з іменем start
```

У наведеній програмі під коренем повинні бути додатні числа. Якщо ж передбачаються будь-які значення під коренем, то необхідно виконати аналіз передбачуваного знака і вивести відповідні повідомлення.

Команда fscale виконує піднесення числа з st(1) до степеня двійки та множення на число з st(0) за формулою $y = y\ 2^x$, де x – округлене число в сторону нуля до цілого та занесене в st(1), y – число з st(0).

Приклад використання команди fscale наведено в лістингу 12.10.

Лістинг 12.10. Програма дослідження виконання команди fscale:

```
        title обчислення степеня двійки. CopyRight by Rysovaniy A. N.
        .686                    ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти  и соглашения ОС Windows
        option casemap: none  ; отличие строчных и прописных букв
        include \masm32\include\windows.inc   ; файлы структур, констант …
        include \masm32\macros\macros.asm
      uselib kernel32, user32, fpu
        BSIZE equ  30           ; розмір буфера
  .data                         ; директива определения данных
        dig1  DD 4.9
        dig2  DD 2.0
        res TBYTE ?
stdout DWORD ?   ; резервування в пам'яті 32-розрядної комірки
                                        ; з ім'ям stdout для збереження дескриптора виведення
cWritten  DWORD  ? ; резервування 32-розрядної комірки пам'яті
                                        ; з ім'ям cWritten для адреси символів виведення
buf BYTE BSIZE dup (?) ; кількість байтів для запису послідовності символів
.code                   ; директива начала кода программы
_st:                    ; метка начала программы
        finit                       ; ініціювання співпроцесора
        fld dword ptr dig1          ; st(0) := dig1
        fld dword ptr dig2          ; st(0) := dig2, st(1) := dig1
```

```
        fscale                  ; y = dig2 × 2^dig1
        fstp res                ; збереження результату в res
invoke  GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
        mov  stdout, eax    ; збереження дескриптора виведення у пам'яті
invoke  FpuFLtoA, \   ; функція перетворення 80-розрядного числа
 ADDR res, \     ; адреса числа, що відображається
 10, \           ; кількість десяткових знаків після коми (10)
 ADDR buf, \     ; адреса буфера для символів, які перетворяться
 SRC1_REAL or SRC2_DIMM ; адреса 80-розрядного числа та саме число
invoke  WriteConsoleA,  stdout,  ADDR buf, \      ; адреса початку повідомлення
   BSIZE, \             ; розмір повідомлення
    ADDR cWritten, 0    ; адреса, де зберігається число символів
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end _st                 ; директива окончания программы
```

Команда fprem обчислює остачу від ділення діленого st(0) на дільник st(1). Знак результату дорівнює знаку st(0), а сам результат знаходиться у вершині стека st(0). Дія команди полягає в зсувах і відніманнях, аналогічно до ручного ділення «в стовпчик».

Після виконання команди прапорець c2 регістра стану може набувати таких значень:

0 – остача від ділення, одержана в st(0), менша від дільника st(1); команда завершується повністю;

1 – st(0) містить часткову остачу; програма повинна ще раз виконати команду для отримання точного значення остачі.

Команда frndint округляє st(0) відповідно до змісту поля rc регістра керування.

Команда fabs обчислює абсолютне значення st(0). Команда fchs змінює знак st(0) на протилежний.

Команда frndint виконує округлення числа, що знаходиться у вершині стека st(0), до цілого числа. Команда не має операндів, а тому використовує операнд із регістра st(0) і після перетворення його там і залишає (повертає результат у регістр st(0)). Допускаються чотири режими округлення, причому визначаються вони значенням поля rc керівного регістра cwr співпроцесора. Режим округлення можна встановлювати в процесі виконання програми до того, як виникне потреба в цій операції. Для цього можна встановити потрібну комбінацію бітів у полі rc регістра cwr, використовуючи команди fstcw (збереження регістра cwr) і fldcw (завантаження регістра cwr).

Параметр, який вказує режим округлення, може набувати значень від 0 до 3, при цьому 0 відповідає режиму з rc = 00 (округлення до найближчого цілого числа), 1 – режиму з rc = 01 (округлення в меншу сторону), 2 – режиму з rc = 10 (округлення у більшу сторону) і, нарешті, 3 – режиму з rc = 11 (відкидання дробової частини).

Приклад 12.10. Написати програму додавання двох чисел та округлити результат до найближчого цілого числа.

Програму обчислення прикладу 12.10 наведено у лістингу 12.11.

Лістинг 12.11. Програма обчислення прикладу 12.10:

```
        title CopyRight by Rysovaniy A. N.
        .686                ; директива определения типа микропроцессора
        .model flat,stdcall  ; задание линейной модели памяти и соглашения ОС Windows
        option casemap: none    ; отличие строчных и прописных букв
        include \masm32\include\kernel32.inc ; системные функции
        includelib \masm32\lib\kernel32.lib
.data                       ; директива определения данных
op1 dd 10.1 ; збереження в 32-розрядній комірці з ім'ям op1 числа 10.1
        op2 dd 10.2
        tmp1  DW   0F3FFh   ; rc = mode 00
        res  DD 0
.code                                   ; директива начала кода программы
_st:                                    ; метка начала программы
        finit                           ; ініціювання співпроцесора
        fldcw  tmp1                     ; завантажити cwr константою з іменем tmp1
        fld     op1                 ; занести op1 в st(0)
        fadd  op2                   ; st(0) := st(0) + op2
        frndint                     ; округлити st(0)
        fstp  dword ptr res     ; занести st(0) у комірку пам'яті з іменем res
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end _st             ; директива окончания программы з іменем _st
```

Вікна налагоджувача після виконання програми лістингу 12.10 наведено на рис. 12.9.

З рис. 12.16 видно, що після округлення у вершині стека st(0) буде число 20.0_{16}. Таким чином, щоб змінювати режими округлення, необхідно заносити у регістр cwr такі константи:

```
F3FFh   ; rc = mode 00
F7FFh   ; rc = mode 01
FBFFh   ; rc = mode 10
FFFFh   ; rc = mode 11.
```

У програмі з лістингу 12.9 режим роботи задається константою з іменем tmp1. Результат виконання операції зберігається в комірках пам'яті з іменем res.

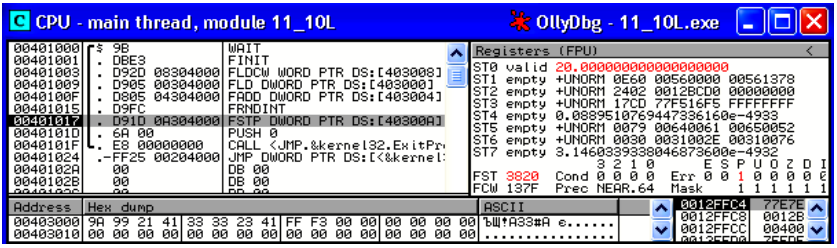


Рис. 12.9. Вікна налагоджувача після виконання програми

Розглянемо команду **fxtract** – виділення з числа, яке знаходиться у вершині стека, порядку та мантиси. При цьому порядок розміщується в регістрі st(1), а мантиса – в st(0). Програму виділення порядку та мантиси числа наведено у лістингу 12.12.

Лістинг 12.12. Програма з використанням команди fxtract:

```
        title CopyRight by Rysovaniy A. N.
        .686                ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap: none      ; отличие строчных и прописных букв
        include \masm32\include\kernel32.inc ; системные функции
        includelib \masm32\lib\kernel32.lib
.data                               ; директива определения данных
        dig dd 12.25 ;
        significand dd 0 ;
        exponent dd 0 ;
.code                       ; директива начала кода программы
_st:                        ; метка начала программы
        finit               ; ініціювання співпроцесора
        fld dig                 ; занести числа в st(0) – вершину стека
        fxtract                 ; st(0) := порядок, st(1) :=  мантиса
        fst dword ptr significand       ; збереження в пам'яті порядку числа
        fst dword ptr exponent          ; збереження в пам'яті  мантиси числа
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end _st         ; директива окончания программы з іменем _st
```

Програму написано так, що в стеку залишаються порядок та мантиса. Крім того, ці дані розміщені й у комірках пам'яті, але вже у форматі для 32-розрядного дійсного числа.

Арифметичні вбудовані masm32-функції

У співпроцесорі використовуються **арифметичні вбудовані masm32-функції** (masm32 v11, Help):

- **FpuAdd** – додавання двох чисел;
- **FpuSub** – віднімання одного числа від іншого;
- **FpuMul** – множення одного числа на інше;
- **FpuDiv** – ділення одного числа на інше;
- **FpuSqrt** – квадратний корінь додатного числа;
- **FpuXexpY** – обчислення експоненти;
- **FpuAbs** – обчислення абсолютної величини числа;
- **FpuChs** – зміна знака числа;
- **FpuRound** – округлення числа до найближчого цілого числа;
- **FpuTrunc** – відкидання десяткової частини числа.

Як приклад розглянемо функцію **FpuAdd,** яка додає два числа та повертає результат як 80-розрядне дійсне число до вказаного місця призначення:

Src1 + Src2 → Dest

Прототип функції FpuAdd

```
FpuAdd (
lpSrc1    // вказівник (адреса) або значення (1-й параметр)
lpSrc2    // вказівник (адреса) або значення (2-й параметр)
lpDest    // вказівник (адреса) на місце призначення результату
uID       // параметр  джерел та місця призначення
```

)

Вказівники можуть мати формати REAL10, DWORD або можуть використовувати константи FPU:
FPU_PI або FPU_NAPIER (число e).

Параметр *ulD* джерел та місця може бути таким:
SRC1_FPU – Src1 є вже на FPU;
SRC1_REAL – вказівник на 80-розрядне дійсне число в пам'яті;
SRC1_DMEM – вказівник на 32-розрядне ціле число із знаком;
SRC1_DIMM – Src1 є 32-розрядним цілим числом із знаком;
SRC1_CONST – одна із спеціальних констант FPU;
DEST_FPU – результат залишається на FPU;
DEST_MEM – lpDest-вказівник (TBYTE – типове значення за умовчанням) пам'яті.
Для вказівника SRC2 значення параметра аналогічні SRC1.

Наприклад:

```
.data
    Src1 DT 2.0   ; може бути: Src1 DD 2.0
    Dest DT ?
.code
    start:
    mov eax,3
    finit
invoke FpuAdd, ADDR Src1, eax, ADDR Dest, SRC1_REAL or SRC2_DIMM or DEST_FPU
```

У цьому прикладі, 80-розрядне дійсне число з комірки пам'яті Src1 додається з цілим числом із регістра EAX, а результат залишається в st(0) FPU та є просто числом.

В прикладі:
invoke FpuAdd,2,3,0, SRC1_DIMM or SRC2_DIMM or DEST_FPU
безпосередні числа 2 та 3 додаються та залишаються на вершині стека (st(0)) співпроцесора вже як дійсний результат. Якщо вказати інші регістри стека співпроцесора для збереження результата, то результат все одно буде зберігатися в вершині стека.

Параметри функції **FpuSub, FpuMul, FpuDiv** аналогічні функції **FpuAdd** (тому що беруть участь 2 операнди).

В прикладі:

```
    include \masm32\include\windows.inc   ; файлы структур, констант …
    include \masm32\macros\macros.asm
    uselib kernel32, user32, fpu
.data                ; директива определения данных
    c1  DT 1.1 ; визначення в 80-розрядній комірці числа 1.1
    c2  DT 5.0 ; визначення в 80-розрядній комірці числа 5.0
    res TBYTE ?  ; 80-розрядна комірка для результатів
```

```
    info db  "сумма двух чисел = "
    buf db 20 dup (?),0
    titl db "Решение уравнения",0
.code                   ; директива начала кода программы
_st:                    ; метка начала программы
invoke FpuAdd, addr c1, addr c2, addr res, SRC1_REAL or SRC2_REAL or DEST_MEM
  invoke FpuFLtoA,addr res,5,ADDR buf,SRC1_REAL or SRC2_DIMM
  invoke MessageBox,0,addr info,addr titl,MB_ICONINFORMATION
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
      end _st                   ; директива окончания программы
```

виконується додавання двох дійсних чисел та виведення результату у спрощене віконце.

Функція **FpuSqrt** обчислює квадратний корінь додатного числа за схемою

sqrt(Src) → Dest

Прототип функції **FpuSqrt**:

FpuSqrt(
lpSrc1 // вказівник (адреса) або значення
lpDest // вказівник (адреса) на місце призначення результату
uID // ID параметр джерел та місця призначення
)

Параметр *uID* джерел та місця може бути таким:

SRC1_FPU – Src1 є вже на FPU;
SRC1_REAL – вказівник на 80-розрядне дійсне число в пам'яті;
SRC1_DMEM – вказівник на 32-розрядне ціле число із знаком;
SRC1_DIMM – Src1 є 32-розрядним цілим числом із знаком;
SRC1_CONST – одна із спеціальних констант FPU;
DEST_FPU – результат залишається на FPU;
DEST_MEM – lpDest-вказівник (TBYTE – типове значення за умовчанням).

Наприклад:

```
invoke FpuSqrt, edx, 0, SRC1_DIMM or DEST_FPU
```

У цьому прикладі обчислюється квадратний корінь 32-розрядного числа з EDX, результат заноситься st(0).

Функція **FpuXexpY** виконується за схемою Src1^Src2 → Dest та має параметри, як і функція FpuAdd**.**

Функція **FpuAbs** обчислює абсолютну величину числа за схемою

sqrt(Src) → Dest

Прототип функції FpuAbs:

FpuAbs (
lpSrc // вказівник на початковий параметр (Src)
lpDest // вказівник на місце призначення результату
uID // параметр для джерела і місця призначення
)

Параметр *uID* джерел та місця може бути таким:
SRC1_FPU – Src1 є вже на FPU;
SRC1_REAL – вказівник на 80-розрядне дійсне число в пам'яті;
DEST_FPU – результат залишається на FPU;
DEST_MEM – lpDest-вказівник (TBYTE – типове значення за умовчанням).

Функція **FpuChs** зміни знака числа аналогічна функції **FpuSqrt**.

Функція **FpuRound** округляє число до найближчого цілого числа за схемою

round(Src) → Dest

Прототип функції FpuRound:

```
FpuRound (
lpSrc     // вказівник на початковий параметр (Src)
lpDest    // вказівник на місце призначення результату
uID       // параметр для джерела і місця призначення
)
```

Параметр *uID* джерел та місця може бути таким:
SRC1_FPU – Src1 є вже на FPU;
SRC1_REAL – вказівник на 80-розрядне дійсне число в пам'яті;
DEST_FPU – результат залишається на FPU;
DEST_IMEM – lpDest-вказівник на DWORD;
DEST_MEM – lpDest-вказівник (TBYTE – типове значенням за умовчанням).

Наприклад:

```
invoke FpuMul,addr a1, 3, 0, SRC1_REAL or SRC2_DIMM or DEST_FPU
invoke FpuRound, 0, 0,SRC1_FPU or DEST_MEM
```

В цьому прикладі перша директива виконує перемноження числа a1, яке знаходиться в памяті, на число 3 та збереження результату в st(0). Друга директива округляє число до найближчого цілого числа та виконує збереження результату в st(0).

Функція **FpuTrunc** відкидання десяткової частини числа аналогічна функції **FpuRound**.

Розглянемо приклад обчислення виразу $y = a/b - \sqrt{c}$, оформлений у вигляді процедури (лістинг 12.13).

Лістинг 12.13. Програма обчислення виразу $y = a/b - \sqrt{c}$:

```
title CopyRight by Rysovaniy A. N.   Вычисление y = a/b – Sqrt c
.686                    ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти  и соглашения ОС Windows
option casemap:none         ; отличие строчных и прописных букв
include \masm32\include\windows.inc   ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib kernel32, user32, fpu
Calc proto
BSIZE equ 10      ; задание реального количества байтов
```

```
      .data                ; директива определения данных
        buf db BSIZE dup (?)
        ifmt db "%d",0
        stdout dd ?
        cWritten dd ?
        res dt 0
        a dt 10.
        b dt 4.
        c1 dt 2.25
.code                      ; директива начала кода программы
_st:                       ; метка начала программы
     finit                 ; ініціювання співпроцесора
     invoke Calc
   Calc proc
     invoke FpuDiv,addr a,addr b,addr res, SRC1_REAL or SRC2_REAL or DEST_MEM
     invoke FpuSqrt,addr c1, 0, SRC1_REAL or DEST_FPU
     invoke FpuAdd,addr res,0,addr res,SRC1_REAL or SRC2_FPU or DEST_MEM
   Calc endp
  invoke GetStdHandle,STD_OUTPUT_HANDLE
  mov stdout,eax       ; збереження дескриптора виведення у пам'яті
invoke FpuFLtoA,addr res,5,ADDR buf,SRC1_REAL or SRC2_DIMM
invoke WriteConsoleA,stdout,ADDR buf,BSIZE,ADDR cWritten,NULL
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
         end _st             ;  директива закінчення програми
```

Розглянемо **приклад**. Задати масив із чотирьох дійсних чисел: перше число в масиві – A, друге – B, третє – C, четверте – D. Обчислити вираз *y = ab/c + Sqrt(d)* з використанням вбудованих masm32-функций та вивести результат через MessageBox (лістинг 12.14).

Лістинг 12.14. Програма обчислення виразу *ab/c + Sqrt(d)*:

```
      ; y=ab/c + Sqrt(d) = 2.4*4.5/5.4+Sqrt(4)= 2+2=4
      .686
      .model flat, stdcall
      option casemap:none
      include \masm32\include\kernel32.inc
      include \masm32\include\windows.inc
      include \masm32\include\user32.inc
      include \masm32\include\fpu.inc
      includelib \masm32\lib\user32.lib
      includelib \masm32\lib\kernel32.lib
      includelib \masm32\lib\fpu.lib
.data
      titl db "Летучка: Вар. №2",0
      _y dt 0.0
      info db "y = ab/c + Sqrt(d) ",0Ah,0Dh,0Dh,
      "При a=2.4, b=4.5, c=5.4, d=4.0",0Ah,0Dh,
      "y = "
      _res dt 0
      mas dt 2.4, 4.5, 5.4, 4.0          ; a=2.4 b=4.5 c=5.4 d=4.0
.code
_st:
      finit
invoke FpuMul, ADDR mas, ADDR mas+10, 0, SRC1_REAL or SRC2_REAL or DEST_FPU
invoke FpuDiv, 0, ADDR mas+20, ADDR _y, SRC1_FPU or SRC2_REAL or DEST_MEM
```

```
invoke FpuSqrt, ADDR mas+30, 0, SRC1_REAL or DEST_FPU
invoke FpuAdd, 0,ADDR _y,ADDR _y, SRC1_FPU or SRC2_REAL or DEST_MEM
invoke FpuFLtoA, offset _y, 2, offset _res, SRC1_REAL or SRC2_DIMM
invoke MessageBox,NULL,offset info,ADDR titl,MB_ICONINFORMATION
      invoke ExitProcess,0
      end _st
```

Спрощене вікно результату виконання програми наведено на рис. 12.10.

Рис. 12.10. Результат виконання програми з лістингу 12.14

Приклад 12.11. Обчислити 3 значення функції $Y = 12^x$, x змінюється від 0,3 з кроком 1.

У лістингу 12.14 розглянуто таку програму. Її особливістю є те, що виведення даних на консоль виконується послідовно при кожному натисканні на клавішу "Enter".

Лістинг 12.15. Програма обчислення виразу $y = 12^x$, x змінюється з кроком 1:

```
       title Рысованый А.Н. ; y = 12^x,  x  змінюється від 0,3 з кроком 1
       .686                ; директива определения типа микропроцессора
       .model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
       option casemap:none         ; отличие строчных и прописных букв
       include \masm32\include\windows.inc   ; файлы структур, констант …
       include \masm32\macros\macros.asm
      uselib kernel32, user32, fpu, masm32
.data              ; директива определения данных
 x REAL10 12.  ; визначення у 80-розрядній комірці змінної x = 12.0
 y REAL10 0.3  ; визначення у 80-розрядній комірці змінної y = 0.3
   st2 db 20 dup(?),0             ; буфер st2 з нулем на кінці
.code                    ; директива начала кода программы
_st:                     ; метка начала программы
   mov ecx,3             ; кількість циклів обчислення y
   mov ebx,0             ; регістр для збереження кроку
@@:
   push ecx              ; підготовка до виклику API-функції
   push ebx              ; підготовка до виклику API-функції
invoke FpuAdd, addr y, ebx,\  ; 1-й операнд  y пам'яті, 2-й операнд y регістрі
  0, \                   ; результат y FPU
   SRC1_REAL or SRC2_DIMM or DEST_FPU  ; призначення
invoke FpuXexpY, \      ; masm32-функція піднесення до степеня
 addr x,0,0\ ; 1-й операнд y пам'яті, 2-й операнд y в FPU, результат y FPU
   SRC1_REAL or SRC2_FPU or DEST_FPU ; призначення
 ; 1-й операнд y пам'яті, 2-й операнд y FPU, результат DEST y FPU
invoke FpuFLtoA, \  ; функція перетворення 80-розрядного числа
0, \     ; адреса числа, що відображається (нульовий регістр стека FPU)
10, addr st2, \ ; кількість десяткових знаків після коми та  адр. для символів
   SRC1_FPU or SRC2_DIMM        ; 1-й операнд y FPU, 2-ге - число
invoke StdOut, addr st2  ; виведення на екран та адреса буфера
```

```
invoke StdIn,          ; отримує текст від консолі і розміщує в буфері
    ADDR st2, 20; введення в буфер st2 з консолі та довжина буфера st2
;  функція  StdIn  закінчується після натискання  на Enter
   pop ebx          ; відновлення ebx після виконання функцій
   inc ebx            ;
   pop ecx          ; відновлення лічильника після виконання функцій
   loop @b            ;
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end _st            ; директива окончания программы
```

Функція StdOut виводить число, яке знаходиться в буфері (в даному випадку це число з st2).

Призначення функції StdIn – введення рядка з екрана в буфер після натискання на клавішу Enter. У цій програмі ні числа, ні текст не вводяться, оскільки вона передбачає лише натискання на цю клавішу.

Результат виконання програми з лістингу 12.15 наведено на рис. 12.11.

Рис. 12.11. Результат виконання програми з лістингу 12.15

Приклад обробки *цілих чисел* вбудованими функціями наведено в лістингу 12.16.

Лістинг 12.16. Виконання виразу ab/c+sqr(d):

```
      .686      ; директива определения типа микропроцессора
      .model flat,stdcall ; задание линейной модели памяти
      option casemap:none ; отличие строчных и прописных букв
      include \masm32\include\windows.inc ; файлы структур, констант …
      include \masm32\macros\macros.asm
      uselib user32,kernel32,fpu
.data
      a1 DD 4
      b1 DD 2
      c1 DD 4
      d1 DD 4
      result dd ?
    res dd ?
      info db "res= "
      titl db "Вывод результата",0 ; название упрощенного окна
      st1 dw ?,0 ; буфер выведения сообщения. dw для небольших чисел
      ifmt db "Вывод чисел с памяти через MessageBox:", 0dh,0ah,
      "result = %d", 0dh,0ah,"Автор программы: ",0
.code
 start:
      finit
    invoke FpuMul,addr a1,addr b1,0,SRC1_REAL or SRC2_REAL or DEST_FPU
    invoke FpuDiv,0,addr c1,0,SRC1  FPU or SRC2  DMEM or DEST  FPU
    invoke FpuSqrt,addr d1, 1,SRC1_DMEM or DEST_FPU;
    invoke FpuAdd,1,0,0,SRC1_FPU or SRC2_FPU or DEST_FPU
```

```
fistp res
  invoke wsprintf, ADDR st1,ADDR ifmt,res
  invoke MessageBox, 0, addr st1, addr titl, MB_ICONINFORMATION
  invoke ExitProcess, 0 ; возвращение управления ОС и освобождение ресурсов
  end _start
```

Для восприятия целых чисел сопроцессором необходимо, чтобы эти числа имели размерность DD. Кроме того, результат обработки из вершины стека сопроцессора необходимо переписать или в целочисленный регист, или в 32-разрядную ячейку памяти для дальнейшей обработки функцией wsprintf.

12.3.4. Трансцендентні команди

Трансцендентні команди призначені для обчислення всіх звичайних тригонометричних (sin, cos, tg,...), зворотних тригонометричних (arcsin, arccos,...), гіперболічних (sh, ch, th,...), зворотних гіперболічних (arsh, arch, arcth,...), логарифмічних і показникових функцій (x^y, 2^x, 10^x, e^x). Зазвичай такі обчислення займають багато часу і є дуже інтенсивними щодо використання ресурсів мікропроцесора.

Трансцендентними командами співпроцесора є:

- **fsin** – обчислення sin(x). Спочатку в регістр стека st(0) заноситься значення аргументу в радіанах, а потім обчислюється функція;
- **fcos –** обчислення cos(x). Спочатку в регістр стека st(0) заноситься значення аргументу в радіанах, а потім обчислюється функція;
- **fsincos** – обчислення sin(x) і cos(x) одночасно. Спочатку в регістр стека st(0) заноситься значення аргументу в радіанах, а потім обчислюється функція. Після виконання цієї команди в регістрі стека st(0) зберігається синус кута, а в регістрі стека t(1) – косинус кута;
- **fptan –** обчислення часткового тангенса;
- **fpatan –** обчислення часткового арктангенса;
- **fyl2x –** обчислення y $\log_2(x)$. Початкове значення x розміщується у вершині стека співпроцесора, а початкове значення y – в регістрі st(l). Значення x повинне знаходитися в діапазоні $0 < x < +\infty$, а значення y – в діапазоні $-\infty < y < +\infty$. Перед записом результату у вершину стека команда fyl2x виштовхує із стека значення x та y і тільки після цього поміщає результат в st(0);
- **fyl2xp1** – обчислення $y\log_2(x+1)$. При цьому початкове значення x розміщується у вершині стека st(0), а початкове значення y – в регістрі st(1). Значення x повинне знаходитися в діапазоні $0<|x|\le 1-1/2$, а значення y – в діапазоні $-\infty < y < +\infty$. Перед записом команда виштовхує значення x і y із стека, після чого результат записується в регістр стека st(0);
- **f2xm1 –** обчислення $2^x - 1$. Початкове значення параметра x повинне знаходитися у вершині стека st(0) і бути в діапазоні $-1 < x < 1$, результат розміщується у вершині стека st(0);
- **f2xml** – обчислення значення функції $y = 2x^{-1}$. Початкове значення параметра x повинне знаходитися у вершині стека st(0) і бути в діапазоні $-1 < x < 1$, результат

розміщується у вершині стека st(0).

Команда fptan обчислює частковий тангенс аргументу в радіанах з st(0) та розташовує результат в st(0). Після виконання команди число *y* розташовується в st(0), а число *x* – в st(1). Аргумент команди fptan повинен знаходиться в таких межах:

0 <= st(0)<= π/4.

Остаточно тангенс кута в радіанах буде отримано після виконання фрагмента коду

```
fptan ; обчислення часткового тангенса
fdivr st,st(1) ; обчислення тангенса кута
```

Користуючись набутим значенням часткового тангенса, можна обчислити інші тригонометричні функції:

sin(*z*) = 2*(*y*/*x*) / (1 + (*y*/*x*)2);
cos(*z*) = (1 - (*y*/*x*)2) / (1 + (*y*/*x*)2);
tg(*z*/2) = *y*/*x*;
ctg(*z*/2) = *x*/*y*;
cosec(*z*) = (1 + (*y*/*x*)2) / 2*(*y*/*x*);
sec(*z*) = (1 + (*y*/*x*)2) / (1 - (*y*/*x*)2),

де *z* – значення, що знаходилося в ST(0) до виконання команди FPTAN, *x* і *y* – значення в регістрах ST(0) і ST(1) відповідно.

Команди для обчислення тригонометричних функцій працюють з аргументами, вираженими ***в радіанах***, тому для обчислення функції кута, заданого в градусах, слід спочатку перетворити це значення в радіани. Для такого перетворення можна використовувати формулу

Кут (рад) = Кут (град) х π/180, де π/180 = 0,0174.

Команда fsin заміщає аргумент (у радіанах), розташований у *вершині* стека st(0), значенням синуса. Команда fcos обчислює косинус кута і поміщає його в st(0). Команда fsincos комбінує попередні дві команди і обчислює обидві функції таким чином:

– значення синуса кута поміщається в st(0);

– косинус кута поміщається в стек регістрів.

Отже, після виконання операції косинус кута опиниться у вершині стека st(0), синус – в регістрі st(i).

Приклад 12.12. Обчислити sin 60° та cos 60° та перевірити правильність їх обчислення.

Тригонометричні функції обчислюються в радіанах, тому значення градусів необхідно перемножити на π/180 = 0,0174. Обчислювання sin 60° виконаємо за допомогою команди fsin, а cos 60° – за допомогою команди fcos. А вже за допомогою команди fsincos виконаємо перевірку правильності отриманих результатів. Програму виконання прикладу 12.12 наведено у лістингу 12.16.

Лістинг 12.16. Програма виконання прикладу 12.11:

```
    .686                          ; директива определения типа микропроцессора
   .model flat                    ; задание линейной модели памяти
    option casemap:none   ; отличие строчных и прописных букв
.data                             ; директива определения данных
    x1 DD 60.                ; величина градусів
    const  DD  0.01745       ; π/180
    sin_1  DD  0   ; перша комірка для обчислення sin
    cos_1 DD  0   ; перша комірка для обчислення cos
    sin_2  DD  0   ; друга комірка для обчислення sin
    cos_2 DD  0   ; друга комірка для обчислення cos
.code                             ; директива начала кода программы
_st:                              ; метка начала программы
    finit              ; ініціювання співпроцесора
    fld   dword ptr x1           ; занесення аргументу у вершину стека
    fmul  dword ptr const ;
    fsin               ;
    fstp  dword ptr sin_1   ; занесення аргументу в комірку  sin_1
    fld   dword ptr x1
    fmul  dword ptr const
    fcos                         ;
    fstp  dword ptr cos_1 ; занесення аргументу в комірку cos_1
    fld   dword ptr x1
    fmul  dword ptr const
    fsincos                      ;
    fstp  dword ptr cos_2  ; збереження у пам'яті cos_2
    fstp  dword ptr sin_2
       ret                       ; возврат управления  ОС
    end _st                      ; директива окончания программы
```

Значення градусів вводиться змінною *x*1, а переведення в радіани виконується в результаті множення величини кута в радіанах на константу командою fmul dword ptr const. Виконання команди fsincos підтверджує результати виконання команд fsin та fcos.

Значення sin $60^o = (\sqrt{3})/2$ розташовується в комірках, починаючи з адрес 402008 та 402010, а значення cos $60^o = 1/2$ – у комірках, починаючи з адреси 40200C та 402014.

Приклад 12.13. Написати програму обчислення виразу для дійсних чисел із ***передаванням параметрів у процедурі*** для рівняння $y = ab - 2^c$. Результат виконання операції зберегти в пам'яті.

Лістинг 12.17. Програма виконання прикладу 12.13:

```
    title CopyRight by Rysovaniy A. N.       Обчислення y = ab – 2^c
    .686                          ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
option casemap:none               ; отличие строчных и прописных букв
include \masm32\include\windows.inc   ; файлы структур, констант …
include \masm32\macros\macros.asm
    uselib kernel32, user32, fpu, masm32
.data                             ; директива определения данных
    _a DD 1.1          ; визначення в 32-розрядній комірці числа 1.1
    _b DD 5.0          ; визначення в 32-розрядній комірці числа 5.0
```

```
        _c DD 0.3 ;
        res dd ?                ; комірка для результату
        myProc proto :dword, :dword, :dword
.code                   ; директива начала кода программы
  start:                ; метка начала программы
        invoke myProc, _a, _b, _c
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
myProc proc arg1:dword, arg2:dword, arg3:dword
        finit ; ініціювання співпроцесора
        fld _a                  ; занесення _a у вершину стека
        fmul _b                 ; st := st(0) x _b
        fld _c                  ; st := _c
        f2xm1                   ; st := 2^c – 1
        fld1                    ; занесення 1 для корекції
        fadd                    ; корекція зайвої одиниці
        fsub st(1), st(0)       ; st(0) := st(1) – st
        fst res                 ; збереження в пам'яті
myProc endp         ; закінчення процедури
        ret                     ; возвращение из процедуры
        end start               ; директива окончания программы
```

Результат виконання програми з лістингу 12.17 наведено на рис. 12.12.

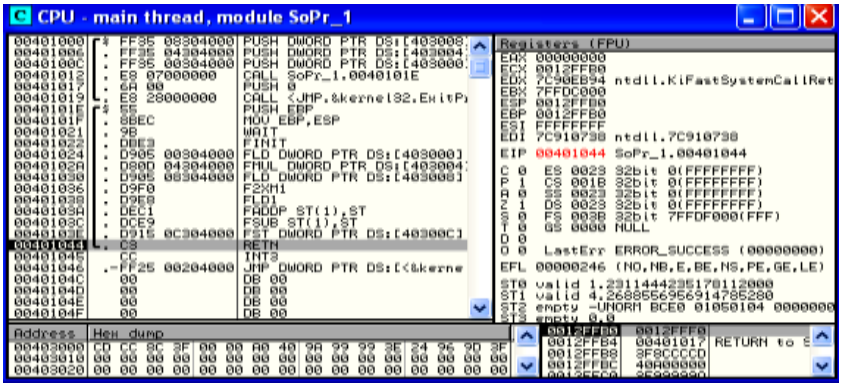


Рис. 12.12. Вікна налагоджувача з отриманими результатами

Видно, що команда fadd перетворюється на команду faddp. Ця команда змінює вміст регістрів співпроцесора, тому потрібно бути уважними при її використанні та налагодженні програми.

Приклад 12.14. Написати програму обчислення виразу ab – 2^c ***з використанням вбудованих masm-функцій***. Результат виконання операції зберегти в пам'яті.

Лістинг 12.18. Програма обчислення виразу *ab – 2^c*:

```
title CopyRight by Rysovaniy A. N.     Обчислення y = ab – 2^c
.686                            ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
option casemap:none             ; отличие строчных и прописных букв
```

```
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\include\kernel32.inc ; системные функции…
        include    \masm32\include\fpu.inc
        include    \masm32\include\\user32.inc   ; файлы  интерфейса …
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
        includelib \masm32\lib\fpu.lib
        BSIZE equ 30                    ; задание реального количества байтов
.data                       ; директива определения данных
      _a DT 9.5                     ; 80-розрядне число a = 9.5
      _b DT 12.43                   ; 80-розрядне число b = 12.43
      _c DT 5.35                    ; 80-розрядне число d = 5.35
      _x DT 2.0                     ; 80-розрядне число x = 2.0
       res TBYTE 0                  ; ячейка для хранения результата
       tmp DT ?                     ; комірка для тимчасового результату
      info db "x1 = "   ; підпис числа для віконця результату
buf BYTE BSIZE dup (?),0 ; розмір буфера для символів, які перетворяться
      st1 db "y=a*b-2^c",0          ; назва віконця
.code                               ; директива начала кода программы
start:                              ; метка начала программы
        finit                       ; ініціювання співпроцесора
invoke FpuMul, ADDR _a, ADDR _b, 0, SRC1_REAL or SRC2_REAL or DEST_FPU
invoke FpuXexpY, ADDR _x, ADDR _c, ADDR tmp,
                                    SRC1_REAL or SRC2_REAL or DEST_MEM ; tmp:=2^c
invoke FpuSub, 0, ADDR tmp, ADDR res,
          SRC1_FPU or SRC2_REAL or DEST_MEM ; rez := st(0) - tmp
invoke  FpuFLtoA, \       ; функція перетворення 80-розрядного числа
  ADDR res, 10,\ ; адреса. числа та кількість десяткових знаків після коми
    ADDR buf, SRC1_REAL or SRC2_DIMM ; адреса 80-розрядного 1-го числа в пам'яті
                                                        ; та другий операнд – число
invoke MessageBox, 0, addr info, addr st1, MB_ICONINFORMATION
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start                   ; директива окончания программы
```

В програмі результат виводиться у віконце повідомлення (рис. 12.13).

Рис. 12.13.Віконце виведення результату програми

Для надання інформаційності спрощеного віконця використано параметр MB_ICONINFORMATION. Підпис числа результату "x1 = " виконано за допомогою послідовності команд

info db "x1 = " ; підпис числа для віконця результату

buf BYTE BSIZE dup (?),0 ; буфер для виведення повідомлення

Виклик підпису з результатом виконується параметром addr info функції MessageBox. А у зв'язку з тим що в імені info відсутній 0 на кінці, виконується зчитування й наступного буфера, де безпосередньо розташовується результат операції.

Тригонометричні вбудовані masm32-функції

У співпроцесорі використовуються тригонометричні **вбудовані masm32-функції**:

- **FpuSin** – (повертає) обчислює синус кута;
- **FpuCos** – обчислює косинус кута;
- **FpuTan** – обчислює тангенс кута;
- **FpuArcsin** – обчислює арксинус;
- **FpuArccos** – обчислює арккосинус;
- **FpuArctan** – обчислює арктангенс.

Функції **FpuSin/FpuCos/FpuTan** є аналогічними та мають прототип

```
FpuSin/FpuSin/FpuTan (
    lpSrc     // вказівник на початковий параметр (Src)
    lpDest    // вказівник на місце призначення результату
    uID       // параметр для джерела, кута і місця призначення
    )
```

Параметр *uID* джерел та місця може бути таким:

SRC1_FPU – Src1 є вже на FPU;
SRC1_REAL – вказівник на 80-розрядне дійсне число в пам'яті;
SRC1_DMEM – вказівник на 32-розрядне ціле число із знаком;
SRC1_DIMM – Src1 є 32-розрядним цілим числом із знаком;
ANG_RAD – значення кута виражається в радіанах;
ANG_DEG – значення кута виражається десятковим числом (типове значення, яке можна не вказувати).

Наприклад:

```
invoke FpuSin, ADDR Src, 0,  SRC1_REAL or ANG_RAD or DEST_FPU
```

У цьому прикладі обчислюється синус 80-розрядного дійсного числа з пам'яті у форматі TBYTE з іменем Scr, який виражається в радіанах та зберігається на FPU.

Функції **FpuArcSin/FpuArcCos/FpuArctan** є аналогічними та мають прототип

```
FpuArcSin/FpuArcCos/FpuArctan (
 lpSrc     // вказівник на значення sin/cos/tan (Src)
 lpDest    // вказівник на місце призначення результату
 uID       // параметр для джерела, кута і місця призначення
 )
```

Параметр *uID* джерел та місця може бути таким:

SRC1_FPU – Src1 є вже на FPU;
SRC1_REAL – вказівник на 80-розрядне дійсне число в пам'яті;
ANG_RAD – значення кута виражається в радіанах;
ANG_DEG – значення кута виражається десятковим числом (типове значення, яке можна не вказувати);
DEST_FPU – результат залишається на FPU;
DEST_MEM – lpDest-вказівник (TBYTE – типове значення за умовчанням).

У співпроцесорі використовуються гіперболічні masm32-функції:

- **FpuSinh** – (повертає) обчислює гіперболічний синус числа;
- **FpuCosh** – обчислює гіперболічний косинус числа;
- **FpuTanh** – обчислює гіперболічний тангенс числа;
- **FpuArcsinh** – обчислює число, відповідне до гіперболічного синуса;
- **FpuArccosh** – обчислює число, відповідне до гіперболічного косинуса;
- **FpuArctanh** – обчислює число, відповідне до гіперболічного тангенса.

12.3.5. Команди маніпуляції константами

До команд передання належить також група команд для завантаження в стек регістрів зумовлених констант:

- **fldz** – st(0) ← 0 (завантаження числа 0 у вершину стека співпроцесора);
- **fld1** – st(0) ← 1 (завантаження числа 1 у вершину стека співпроцесора);
- **fldpi** – st(0) ← π (завантаження числа π у вершину стека співпроцесора);
- **fldl2t** – st(0) ← $\log_e 10$ (завантаження двійкового логарифма 10 у вершину стека співпроцесора);
- **fldl2e** – st(0) ← $\log_2 e$ (завантаження двійкового логарифма числа e у вершину стека співпроцесора);
- **fldlg2** – st(0) ← $\log_{10} 2$ (завантаження десяткового логарифма числа 2 у вершину стека співпроцесора);
- **fldln2** – st(0) ← $\log_e 2$ (завантаження натурального логарифма числа 2 у вершину стека співпроцесора).

Логарифмічні вбудовані masm32-функції

У співпроцесорі використовуються **логарифмічні вбудовані masm32-функції:**

- **FpuLogx** – (повертає) обчислює десятковий логарифм додатного числа;
- **FpuLnx** – обчислює натуральний логарифм додатного числа;
- **FpuTexpX** – обчислює 10^(Src) → Dest;
- **FpuEexpX** – обчислює e^(Src) → Dest.

12.3.6. Команди керування

Команди керування не використовуються в обчисленнях, а керують діями співпроцесора на системному рівні. Синтаксис основних керівних команд такий:

- **fwait** – синхронізує роботу мікропроцесора і математичного співпроцесора (якщо мікропроцесор зустрічає цю команду, він припиняє свою роботу до закінчення виконання чергової команди співпроцесора);
- **finit** – ініціалізує співпроцесор із розташуванням зумовлених значень у керівні регістри співпроцесора;
- **fstsw dest** – зберігає регістр стану в 16-розрядному операнді dest; слово стану може зберігатися і в 16-розрядному регістрі AX;
- **fstcw dest** – зберігає вміст регістра керування в 16-розрядній змінній dest в пам'яті (команда використовується для аналізу полів регістра cwr);
- **fldcw src** – завантажує вміст 16-розрядної змінної src в регістр cwr (команда використовується для задання режимів роботи співпроцесора);
- **fclex** – скидає прапорці виключень у регістрі swr співпроцесора;
- **fincstp** – збільшує вказівник стека на одиницю в полі top регістра стану співпроцесора (swr). Ця команда не має операндів і за своєю дією нагадує команду fst. Вміст вершини стека при виконанні команди віддаляється;
- **fdecstp** – зменшує вказівник стека на одиницю в полі top регістра стану співпроцесора (swr). Ця команда не має операндів і за своєю дією нагадує команду fld, але операнд у стек не поміщається;
- **ffree st(i)** – звільняє регістр стека st(i), позначаючи його як порожній у регістрі тегів (twr). Вміст поля top у регістрі стану (swr) і сам вміст регістра не змінюються;
- **fnop** – нічого не робить і може використовуватися для тимчасових затримок;
- **fsave dest** – запам'ятовує стан співпроцесора в 94-байтовій ділянці пам'яті dest (16-розрядний режим роботи) або в 108-байтовій ділянці пам'яті (32-розрядний режим роботи);
- **frstor src** – відновлює стан співпроцесора з 94-байтової ділянки пам'яті src (16-розрядний режим роботи) або з 108-байтової ділянки пам'яті (32-розрядний режим роботи);
- **fstenv dest** – зберігає стан середовища співпроцесора (14 байтів для 16-розрядного режиму роботи і 28 байтів для 32-розрядного режиму) в ділянці пам'яті dest. Команда не зберігає вміст стека регістрів (80 байтів);
- **fldenv src** – виконує часткове відновлення стану середовища співпроцесора (14 байтів для 16-розрядного режиму роботи і 28 байтів для 32-розрядного режиму) з ділянки пам'яті src.

У співпроцесорі використовуються **змішані API-функції**:

FpuSize – повертає розмір експоненціального формату;

FpuExam – перевіряє параметр REAL10 на його переконливість, його знак, значення нуля, абсолютну величину меншу, ніж 1, і/або значення нескінченності;

FpuState – повертає рядок, що закінчується нулем, у буфер пам'яті за номером, який ідентифікує ділянку цього буфера.

12.4. Лабораторна робота «Обробка даних у співпроцесорі»

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм із використанням команд **співпроцесора**, написаних мовою асемблера для програмування МП платформи x86.

Постановка задачі

Згідно з останньою цифрою номера в групі вибрати варіант завдання та написати на асемблері програму обчислення одного з виразів із використанням дійсних чисел та виведенням їх на екран. При цьому обов'язково використати ***спрощені віконця*** з повідомленнями: в одному віконці вивести своє прізвище та номер навчальної групи; при натисканні на другу кнопку спрощеного віконця – вивести умову завдання. Використання інших кнопок спрощеного віконця виведення повідомлень – за бажанням.

Кожне завдання виконати двома способами:

1) з виконанням команд співпроцесора;

2) з використанням вбудованих функцій masm32.

Методичні рекомендації

Для налагодження остаточного варіанта програми спочатку необхідно налагодити частину програми з отриманням одного результату та його виведенням. На другому етапі – отримання масиву результатів. На третьому – виведення всього масиву.

Завдання 1. Дослідити виконання арифметичних операцій співпроцесора. Результат вивести на екран функцією MessageBox з горизонтальним (у рядок) розташуванням результатів.

1. Обчислити 6 значень функції $Y_n = 25x^3 - 2{,}1$ (x змінюється з кроком 0.2).
2. Обчислити 5 значень функції $Y_n = 7x^3/(2x^2 + 1{,}6)$ (x змінюється від 1 з кроком 4).
3. Обчислити 4 значення функції $Y_n = 37/(2x^2 + 7{,}3)$ (x змінюється від 1 з кроком 2).
4. Обчислити 5 значень функції $Y_n = 5{,}1x^2 + 5{,}3$ (x змінюється від 4,7 з кроком 3). Результат округлити до найближчого цілого числа та вивести на екран.
5. Обчислити 6 значень функції $Y_n = 4x/(x + 5)$ (x змінюється від 3 з кроком 1,25). Результат округлити до цілого числа.
6. Обчислити 4 значення функції: $Y_n = 125/(3x^2 - 1{,}1)$ (x змінюється від 3 з кроком 1,5). Результат округлити в меншу сторону.

7. Обчислити 4 значення функції $Y_n = 2{,}5x^2 - 3{,}2$ (x змінюється від 4 з кроком 1).

8. Обчислити 3 значення функції $Y_n = 25x^2 + 2{,}1$ (x змінюється від 3 з кроком 2,5).

9. Обчислити 4 значення функції $Y_n = 150/(x^2 - 7)$ (x змінюється від 2 з кроком 3,1).

10. Обчислити 6 значень функції $Y_n = 256/(3x^2 + 31)$ (x змінюється від 2 з кроком 3).

Завдання 2. Дослідити виконання операцій порівняння співпроцесора.

1. Знайти перше значення аргументу функції $Y = 7(x + 0{,}3)$, при якому молодші цілі цифри результату виконання функції дорівнюватимуть 15 (x змінюється від 2 з кроком 3,5).

2. Знайти ціле значення аргументу, при якому функція $Y = 10/(x^3 + 1{,}7)$ стане менше за 0,3 (x змінюється від 2 з кроком 2,5).

3. Знайти значення x, при якому функція $Y = 3^x/4 - 6$ дорівнюватиме 12,5.

4. Знайти значення x, при якому виконується функція $8 \operatorname{arctg} 0{,}1 + \operatorname{arctg} x = \pi/4$.

5. Визначити номер (x) елемента функції $x_n = 2^x + 5$, при якому сума елементів перевищить 12 000.

6. Знайти значення x, при якому функція $\lg(2^x + 3)$ буде більшою 2,5.

7. Знайти ціле значення аргументу, при якому функція $Y = 5{,}6^x/3x^2$ перевищить 125.

8. Знайти ціле значення аргументу, при якому функція $Y = 2^x + 30$ перевищить 100.

9. Знайти перше значення аргументу функції $Y = 9(x^2 + 0{,}6)$, при якому молодші цілі цифри результату виконання функції дорівнюватимуть 12 (x змінюється від 3 з кроком 5,5).

10. Знайти ціле значення аргументу, при якому функція $Y = 2^x/5 + 4$ перевищить 400.

Завдання 3. Дослідити виконання команд співпроцесора з логарифмічними і показниковими функціями. Результат вивести на екран функцією MessageBox з вертикальним (у стовпчик) або горизонтально-вертикальним розташуванням результатів із підписом кожного елемента результату.

1. Обчислити 5 значень функції $Y = 5{,}2 * \ln(\sin x)$ (x змінюється в градусах із кроком 1,5).

2. Обчислити 4 значення функції $Y = 5^x + \cos x$ (x змінюється від 0,4 з кроком 0,15).

3. Обчислити 6 значень функції $Y = 4{,}5 \lg(\operatorname{tg} x)$ (x змінюється в градусах із кроком 10).

4. Обчислити 6 значень функції $Y = 12^x - \sin x$ (x змінюється з кроком 0,05).

5. Обчислити 8 значень функції $Y = 3{,}3 * \log_2(x^2 + 1)$ (x змінюється з кроком 0,3).

6. Обчислити 4 значення функції $Y = 5{,}1(\sin x)$ (x змінюється в градусах із кроком 8,5).

7. Обчислити 6 значень функції $Y = \lg(x^2 + \sqrt{x})$ (x змінюється з кроком 0,3).

8. Обчислити 5 значень функції $Y = 4{,}3(x^2 + 1)$ (x змінюється з кроком 1,7).

9. Обчислити 6 значень функції $Y = 6{,}2 \lg(\cos x)$ (x змінюється у градусах із кроком 1,5).

10. Обчислити 7 значень функції $Y = 3{,}1 + 2/\cos x$ (x змінюється у градусах від 10 з кроком 8).

Зміст звіту

1. Постановка задачі для конкретного варіанта.
2. Блок-схема алгоритму виконання прикладу.
3. Лістинг програми з детальним коментарем та описом роботи.
4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання.
5. Короткий опис виконання програми.
6. Висновки за результатами роботи.

Приклад 12.15. Обчислити 3 значення функції $Y_n = 5x^2/1{,}1$ (x змінюється від 1 з кроком 1). Результат розмістити в пам'яті та вивести на екран.

Через те, що необхідно вивести три числа, для розділення комірок використовується додатковий елемент (регістр EDX). Спочатку цей регістр дорівнює нулю. Співпроцесор записує в пам'ять 80-розрядні числа (десять байтів). Тому в циклі виконується команда

```
add edx,0Ah
```

для виконання зміщення відносно попередньої комірки, яка потім буде адресуватися як `res[edx]`.

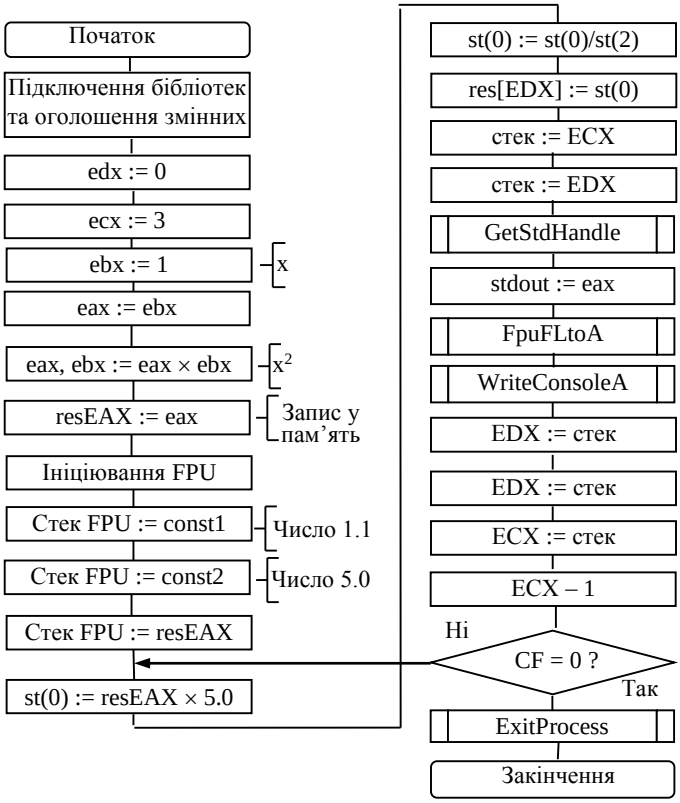
У прикладі при отриманні x^2 використовуються цілі числа. Тому необхідно виконати команду

```
fild dword ptr resEAX.
```

Якщо виконати команду `fld …` , то в даному випадку результат буде з похибкою.

А оскільки API-функції при своєму виконанні змінюють регістри, то значення тих регістрів, які потрібні при виконанні програми, необхідно зберігати (найкраще – в стеку).

Алгоритм розв'язання прикладу 12.15 наведено на рисунку.



Лістинг 12.19. Програма виконання прикладу 12.15:

```
        title CopyRight by Rysovaniy A. N.      Обчислення Yn = 5x2 / 1,1
        .686                            ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти  и соглашения ОС Windows
        option casemap:none                     ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\include\kernel32.inc ; систем. функції застосувань…
        include \masm32\include\fpu.inc
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
         includelib \masm32\lib\fpu.lib
         BSIZE equ  30              ; розмір буфера
    .data                               ; директива определения данных
    const1  DWORD 1.1   ; визначення в 32-розрядній комірці числа 1.1
    const2  DWORD 5.0   ; визначення в 32-розрядній комірці числа 5.0
resEAX DWORD 0          ; 32-розрядна комірка для тимчасового результату
res TBYTE 3 dup (0)     ; три 80-розрядні комірки для результатів
stdout DWORD ?          ; резервування в пам'яті 32-розрядної комірки
                                ; з ім'ям stdout для збереження дескриптора виведення
cWritten  DWORD  ?      ; резервування 32-розрядної комірки пам'яті
                                        ; з ім'ям cWritten для адреси символів виведення
buf BYTE BSIZE dup (?); кількість байтів для запису символів
    .code                           ; директива начала кода программы
```

```
  _st:                          ; метка начала программы
      xor edx,edx               ; початкове значення (для зміщення)
      mov ecx,3                 ; лічильник циклів
      mov ebx,1                 ; початкове значення x
m1: mov eax,ebx                 ; підготування до піднесення до степеня
      mul ebx                   ; піднесення в степінь
      mov resEAX,eax            ; збереження x^2 в пам'яті
    finit                       ; ініціювання співпроцесора
      fld dword ptr const1   ; st(0) := 1.1
      fld dword ptr const2   ; st(0) := 5.0, st(1) := 1.1
      fild dword ptr resEAX ; st(0) := resEAX, st(1) := 5.0, st(2) := 1.1
      fmul st,st(1)       ; st(0) := resEAX x 5.0
      fdiv st,st(2)       ; st(0) :=  st(0)/st(2)
      fstp res[edx]         ; збереження результату в комірці res[edx]
      push ecx                ; підготовка до виклику API-функції
      push edx                ; підготовка до виклику API-функції
  invoke GetStdHandle, STD_OUTPUT_HANDLE ; отримання дескриптора
    mov stdout, eax    ; збереження одержаного дескриптора у пам'яті
invoke  FpuFLtoA, \    ; функція перетворення 80-розрядного числа
  ADDR res[edx], \     ; адреса числа, що відображається
    10, \         ; кількість десяткових знаків після коми (10)
    ADDR buf, SRC1_REAL or SRC2_DIMM  ; операнд SRC1в пам'яті
                                                ;  операнд SRC2 з параметром DIMM
invoke  WriteConsoleA, stdout,\ ; API-функція виведення на екран
    ADDR buf, \      ; адреса початку повідомлення
      BSIZE, \           ; розмір повідомлення
        ADDR cWritten, 0   ; адреса, де зберігається кількість символів
      pop edx              ; встановлення edx
      add edx,0Ah        ; підготовка до вибору наступної комірки
      inc ebx              ; збільшення x
      pop ecx              ; встановлення ecx
    loop m1                  ; перейти, якщо не нуль
invoke ExitProcess, 0 ; возвращение управления ОС та вивільнення ресурсів
      end _st               ; директива окончания программы
```

На рис. 12.14 наведено результат виконання лістингу 12.19.

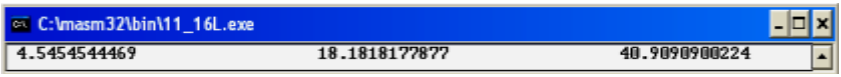


Рис. 12.14. Результат виконання лістингу 12.18

При наведеній послідовності виконання команд виведення відбувається в один рядок. Якщо поставити відповідні параметри у функції WriteConsoleA, то можна переводити виведення в декілька рядків.

Для того щоб самостійно не рахувати кількість байтів, яку займає змінна, наприклад, в команді add edx,0Ah, простіше використати команду

add edx, TYPE res.

Для виведення чисел із пам'яті використовується функція FpuFLtoA, яка перетворює **80-розрядні** числа в символи. Тому й комірки пам'яті потрібно визначати як 80-розрядні, наприклад:

res **TBYTE** 3 dup (0).

Крім того, в параметрах функції треба вказати, що дані виводяться з пам'яті. Для цього служить параметр SRC1_**REAL**.

Наведена програма з лістингу 12.18 може бути написана виключно за допомогою вбудованих функцій masm32. Таку програму наведено в лістингу 12.19.

Лістинг 12.20. Програма виконання прикладу 12.15 з використанням вбудованих функцій masm32:

```
        title CopyRight by Rysovaniy A. N.          Обчислення Yn = 5x^2 / 1,1
        ; використання вбудованих функцій masm32
        .686                            ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none                 ; отличие строчных и прописных букв
        include    \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\include\kernel32.inc ; системные функции применений…
        include    \masm32\include\fpu.inc
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
        includelib \masm32\lib\fpu.lib
        BSIZE equ  30               ; розмір буфера
.data                               ; директива определения данных
      const1  DT 1.1                    ; визначення в 32-розрядній комірці числа 1.1
      const2  DT 5.0                    ; визначення в 32-розрядній комірці числа 5.0
      res TBYTE 3 dup (0)           ; три 80-розрядні комірки для результатів
      stdout DWORD ? ; резервування в пам'яті 32-розрядної комірки
                                        ; з ім'ям stdout для збереження дескриптора виведення
      cWritten  DWORD  ?  ; резервування 32-розрядної комірки пам'яті
                                          ; з ім'ям cWritten для адреси символів виведення
      buf BYTE BSIZE dup (?) ; кількість байтів для запису послідовності символів
.code                                   ; директива начала кода программы
_st:                                    ; метка начала программы
        xor edx,edx         ; початкове значення (для зміщення)
        mov ecx,3           ; лічильник циклів
        mov ebx,1
m1: mov eax,ebx         ; початкове значення x
  invoke FpuMul, eax, ebx, 0, SRC1_DIMM or SRC2_DIMM or DEST_FPU
invoke FpuMul, ADDR const2, 0, 0, SRC1_REAL or SRC2_FPU or DEST_FPU
invoke FpuDiv, 0,ADDR const1,ADDR res[edx], SRC1_FPU or SRC2_REAL or DEST_MEM
        push ecx      ; підготовка до виклику API-функції
        push edx      ; підготовка до виклику API-функції
invoke GetStdHandle, STD_OUTPUT_HANDLE     ; отримання дескриптора
    mov stdout, eax    ; збереження одержаного дескриптора у пам'яті
invoke  FpuFLtoA, ADDR res[edx], \    ; адреса числа, що відображається
   10, \                ; кількість десяткових знаків після коми (10)
     ADDR buf, \        ; адреса буфера для символів, які перетворяться
     SRC1_REAL or SRC2_DIMM ; операнд SRC1в пам'яті
                                                ;  операнд SRC2 з параметром DIMM (число)
invoke  WriteConsoleA, stdout, \       ; дескриптор стандартного пристрою виведення
   ADDR buf, \     ; адреса початку повідомлення
    BSIZE, \        ; розмір повідомлення
     ADDR cWritten, 0     ; адреса, де зберігається кількість символів
        pop edx             ; встановлення edx
        add edx,0Ah         ; підготовка до вибору наступної комірки
        inc ebx             ; збільшення x
        pop ecx             ; встановлення ecx
        dec ecx
        jnz m1              ; перейти, якщо не нуль
invoke  ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end _st             ; директива окончания программы
```

Приклад 12.16. Обчислити 6 значень функції: Y = cos *x*, де *x* змінюється в градусах від 10 з кроком 8.

Лістинг 12.21. Програма виконання прикладу 12.16:

```
        title Rysovaniy A.N.                                    rysov@rambler.ru
; Y = cosx, x змінюється в градусах від 10 з кроком 8.
        .686                    ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
option casemap:none         ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\kernel32.inc ; системные функции применений…
        include \masm32\include\fpu.inc
        include \masm32\include\user32.inc      ; для MessageBox
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
        includelib \masm32\lib\fpu.lib
        BSIZE equ  30  ; розмір буфера
.data                           ; директива определения данных
        const  DWORD 0.01745  ; (π = 3,14)/180 – градуси в радіанах
        i1  DWORD 10                ; початкове значення градусів
        st1 db "         y=cosx",0
        st2 db "Автор: Рысованый А.Н.",0
         st3 db "       Прощание",0
         st4 db "До свидания",0
          st6 db "До следующей лабораторной работы !!!",0
        res TBYTE 6 dup (0)  ; шість 80-розрозрядних комірок для результатів
stdout DWORD ?      ; для збереження дескриптора виведення
cWritten  DWORD  ?    ; резервування 32-розрядної комірки пам'яті
                            ; з ім'ям cWritten для адреси символів виведення
buf BYTE BSIZE dup (?); кількість байтів для запису послідовності символів
.code                       ; директива начала кода программы
  start:                    ; директива начала кода программы
        xor edx,edx             ; початкове значення (для зміщення)
        mov ecx,6                   ; лічильник циклів
      finit                         ; ініціювання співпроцесора
@@: fild dword ptr i1      ; завантаження в st(0) змінної i1 (градусів)
        fmul dword ptr const ; st(0) := st(0) × const
        fcos                        ; st(0) := cos(i1) в радіанах
        fstp res[edx]           ; збереження результату в комірці res[edx]
        push ecx        ; підготовка до виклику API-функції
        push edx        ; підготовка до виклику API-функції
   invoke GetStdHandle, STD_OUTPUT_HANDLE  ; отримання дескриптора
      mov stdout, eax           ; збереження одержаного дескриптора у пам'яті
invoke  FpuFLtoA, ADDR res[edx], \ ; адреса числа, що відображається
     10, \          ; кількість десяткових знаків після коми (10)
      ADDR buf, \  ; адреса буфера для символів, які перетворяться
SRC1_REAL or SRC2_DIMM ; адреса 80-розрядного числа в пам'яті
                                                                ; та саме число
 invoke  WriteConsoleA, stdout, ADDR buf, \ ; адреса початку повідомлення
        BSIZE, ADDR cWritten, 0 ; адреса, де зберігається кількість символів
        pop edx                                 ; встановлення edx
        add edx,0Ah       ; підготовка до вибору наступної комірки
        add i1,8                ; збільшення градусів на  крок дискретизації
        pop ecx                         ; встановлення ecx
        loop @b                         ; перейти, якщо не нуль
invoke MessageBox, 0, addr st2, \ ; адреса рядка, яка містить  текст повідомлення
        addr st1, \          ; адреса рядка, яка містить  заголовок повідомлення
          MB_YESNO or MB_ICONINFORMATION   ; вигляд діалогового вікна
        xor eax,6           ; виділення розрядів коду повернення кнопки YES
```

```
        jz m1
invoke MessageBox, NULL, addr st6, addr st3, MB_ICONWARNING
        jmp m2
m1:
invoke MessageBox, 0, addr st4, addr st3, MB_ICONERROR
m2:
        invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end start             ; директива окончания программы з іменем start
```

У цій програмі команда з унікальною міткою @@: fild означає, що до неї звертається у циклі команда, яка розташовується після неї в тексті та має символ із літерою @b. Такі мітки можна змінити на звичайні.

Результат виконання програми з лістингу 12.21 у вигляді вікна наведено на рис. 12.15.

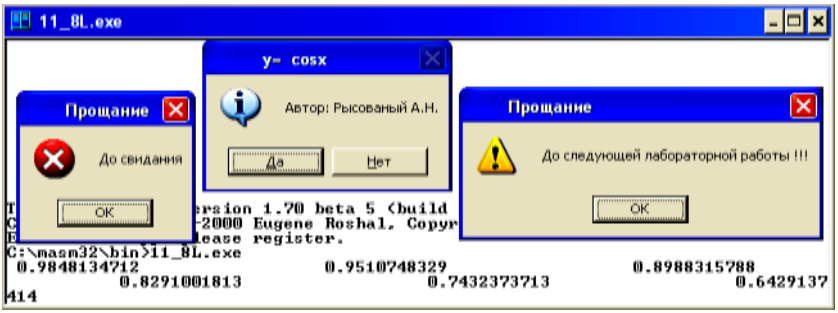


Рис. 12.15. Вікно результату виконання програми з лістингу 12.21

Константи NULL та MB_OK, використані у функції Messagebox, покращують читабельність коду. До них можна безпосередньо звертатися в програмі.

Оператор addr використовується для передання адреси мітки функції. Він дійсний тільки в контексті директиви invoke.

Приклад 12.17. Обчислити 5 значень функції Y = 4,3(x^2 + 1) (x змінюється з кроком 1,7). Результат округлити до цілого числа.

Лістинг 12.22:

```
        .686 ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        include\masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\include\kernel32.inc ; системные функции применений…
        include \masm32\include\\user32.inc   ; файлы  интерфейса …
        include \masm32\include\fpu.inc
        includelib \masm32\lib\kernel32.lib
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\fpu.lib
.data
CrLf equ 0A0Dh    ; Y = 4,3(x^2 + 1) (x изменяется с  шагом 1,7)
            _y1 dt 0.0
            _y2 dt 0.0
```

```
            _y3 dt 0.0
            _y4 dt 0.0
            _y5 dt 0.0
_x DWORD 3.0
            _op1 DWORD 4.3
            _op2 DWORD 1.0
            _zero DWORD 0.0
            _step DWORD 1.7
            info db "y1 = "
            _res1 db 14 DUP(0),10,13
            db "y2 = "
            _res2 db 14 DUP(0),10,13
            db "y3 = "
            _res3 db 14 DUP(0),10,13
            db "y4 = "
            _res4 db 14 DUP(0),10,13
            db "y5 = "
            _res5 db 15 DUP(0),0
            ttl db "Lab #6",0
.code
_start:
finit
mov ecx, 5
m1:   fld _x        ; st(0) := x
   fmul _x          ; st(0) := x*x
   fadd _op2        ; st(0) := x*x + 1
   fmul _op1        ; st(0) := 4.3(x*x + 1)
   fld _x           ; st(0) := x
   fadd _step       ; st(0) := x + 1.7 ; st(1) := 4.3(x*x + 1)
fstp _x    ; сохранение st(0) в памяти и в st(0) перезапись 4.3(x*x + 1)
loop m1
  fstp _y5
  fstp _y4
  fstp _y3
  fstp _y2
  fstp _y1
 invoke FpuFLtoA,offset _y1,10,offset _res1,SRC1_REAL or SRC2_DIMM
 mov word ptr _res1 + 14, CrLf
 invoke FpuFLtoA,offset _y2,10,offset _res2,SRC1_REAL or SRC2_DIMM
 mov word ptr _res2 + 14, CrLf
 invoke FpuFLtoA,offset _y3,10,offset _res3,SRC1_REAL or SRC2_DIMM
 mov word ptr _res3 + 14, CrLf
 invoke FpuFLtoA,offset _y4,10,offset _res4,SRC1_REAL or SRC2_DIMM
 mov word ptr _res4 + 14, CrLf
 invoke FpuFLtoA,offset _y5,10,offset _res5,SRC1_REAL or SRC2_DIMM
 invoke MessageBox, 0, offset info, offset ttl, MB_ICONINFORMATION
 invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end _start
```

Спрощене вікно результату виконання програми з вертикальним виведенням чисел із лістингу 12.22 наведено на рис. 12.16.

Рис. 12.16. Спрощене вікно результату виконання програми

Приклад 12.18. Обчислити 8 значень функції $Y = 3{,}3 * \log_2(x^2 + 1)$ (x змінюється з кроком 0,3).

Лістинг 12.23:

```
        .686                        ; директива определения типа микропроцессора
        .model flat, stdcall    ; задание линейной модели памяти и соглашенияОС Windows
        option casemap:none             ; відмінність малих та великих букв
        include C:/masm32/include/kernel32.inc ; бібліотеки, що підключаються
        include C:/masm32/include/\user32.inc   ; файлы  интерфейса …
        include C:/masm32/include/fpu.inc
        includelib C:/masm32/lib/kernel32.lib
        includelib C:/masm32/lib/user32.lib
        includelib C:/masm32/lib/fpu.lib
.data
            st1 db "Лаба 5 Частина 3. Автор Глущук Дмитро",0
            CrLf equ 0D0Ah
            _x  DWORD 1.0
            hod  DWORD 0.3
            a  DWORD 3.3
            k  DWORD 1.0
            res  dt 6 dup(0)
            ifmt  db "Функція Y=3,3*log^2(x2+1) Знайти 5 значень",10
db "     y1= "
 res1 db 14 dup (32)
db "y2= "
 res2 db 14 dup (32)
db "y3= "
 res3 db 14 dup (32)
db "y4= "
 res4 db 14 dup (32)
db "y5= "
 res5 db 14 dup (32),0
.code
_st:
  finit
      mov ecx,5
      xor edx,edx
m1:      fld _x
         fmul _x
         fadd k
         fld a
         fyl2x
         fstp res[edx]
         fld _x
         fadd hod
         fstp _x
         add edx, 10
loop m1
invoke FpuFLtoA, offset res[0], 5, offset res1, SRC1_REAL or SRC2_DIMM
   mov word ptr res1+7, CrLf
invoke FpuFLtoA, offset res[10], 5, offset res2, SRC1_REAL or SRC2_DIMM
   mov word ptr res2+7, CrLf
invoke FpuFLtoA, offset res[20], 5, offset res3, SRC1_REAL or SRC2_DIMM
    mov word ptr res3+7, CrLf
invoke FpuFLtoA, offset res[30], 5, offset res4, SRC1_REAL or SRC2_DIMM
```

```
  mov word ptr res4+7, CrLf
invoke FpuFLtoA, offset res[40], 4, offset res5, SRC1_REAL or SRC2_DIMM
invoke MessageBox,0,offset ifmt,ADDR st1,MB_OK
invoke ExitProcess, 0 ; возврат управления  ОС и освобождение ресурсов
        end _st
```

Спрощене вікно результату виконання програми з лістингу 12.23 наведено на рис. 12.17.

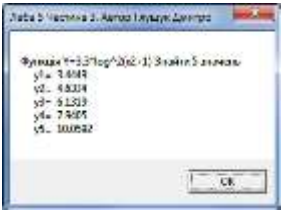


Рис. 12.17. Спрощене вікно результату виконання програми

Приклад 12.18. Обчислити 4 значення функції:$Yn = 125/(3x^2 – 1,1)$ (x змінюється від 3 з кроком 1,5).

Лістинг 12.24:

```
        .686
        .model flat, stdcall
        option casemap:none
        include    \masm32\include\windows.inc
        include    \masm32\include\kernel32.inc
        include    \masm32\include\user32.inc
        include    \masm32\include\fpu.inc
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
        includelib \masm32\lib\fpu.lib
.data
     y dt 4 dup(0.0)
     x dd 3.
     N dd 4
     g dd 3.
     h dd 1.1
     j dd 125.
     CrLf equ 0A0Dh      ; перевод строки
     _step DWORD 1.5  ; шаг изменения X
     temp dd 0
     titl db "Рассчитать 4-е значения выражения: ",0
     info db "Дано выражение: Y = 125/(3x2 – 1,1) , шаг x = 1.5, x0 = 3", 0dh,0ah,0ah,
        "res (a1) = "
        _res1 db 14 DUP(0),10,13
        db "res (a2) = "
        _res2 db 14 DUP(0),10,13
        db "res (a3) = "
        _res3 db 15 DUP(0),10;,13
        db "res (a4) = "
        _res4 db 15 DUP(0),10,13
        db "НТУ ХПИ Рысованый А.Н., rysov@rambler.ru",0
        info_sz = $ - info.code
start:
        lea esi,y
```

```
        mov ecx,N
        finit
m1:   fld x      ; st(0) := x
        fld x      ; st(0) := x
        fmul   ; st(0) := x2
        fld g  ; st(0) := 3
        fmul   ; 3x2
        fld h    ; 1,1
        fxch st(1)
        fsub st, st(1)
        ffree st(1) ;  объявление очистки регистра (для удаления мусора)
        fld j
        fdiv st,st(1)
        fstp REAL10 ptr [esi]   ; tbyte
        add esi,10
        fld x
        fadd _step
        fstp x
        dec ecx
        fstp temp ; для удаления мусора (чтобы "не забился" стек)
        jnz m1
         invoke FpuFLtoA,offset y[0],10,offset _res1,SRC1_REAL or SRC2_DIMM
         mov word ptr _res1 + 14, CrLf
         invoke FpuFLtoA,offset y[10],10,offset _res2,SRC1_REAL or SRC2_DIMM
         mov word ptr _res2 + 14, CrLf
         invoke FpuFLtoA,offset y[20],10,offset _res3,SRC1_REAL or SRC2_DIMM
         mov word ptr _res3 + 14, CrLf
         invoke FpuFLtoA,offset y[30],10,offset _res4,SRC1_REAL or SRC2_DIMM
         mov word ptr _res4 + 14, CrLf
; В цикле просматривается строка на наличие нулей, которые в случае обнаружения
;заменяются на пробел
            mov ecx, info_sz
            dec ecx     ; последний нолик нам нужен
            mov edi, offset info
            mov esi, edi
@l: lodsb         ; AL <- [ESI]
            test    al,al
            jnz @f
            mov al,20h ; символ пробела
@@: stosb         ; [DI] <=  AL
        loop    @l
         invoke MessageBox, 0, offset info, offset titl, MB_ICONINFORMATION
         invoke ExitProcess, 0
         end start
```

В этой программе большое число вводимых переменных, которые засоряют все регистры стека сопроцессора. Поэтому применены дополнительные команды очистки регистров стека, такие как ffree st(1), fstp temp. Если такие команды не использовать, то в MessageBox не будут выводиться все результаты.

Кроме того, в программе все четыре результата записываются в одну строку. А все функции API требуют признака нуля окончания строки. Для этого в цикле просматривается строка на наличие нулей, которые в случае обнаружения заменяются на символ пробел (http://www.cyberforum.ru/assembler/thread666469.html) командой mov al,20h.

Результат выволнения программы наведен на рис. 12.18.

Программу можно упростить, если функцию FpuFLtoA преобразования результата использовать в цикле рассчета. А функцию MessageBox применять уже за таким циклом.

Рис. 12.18. Результат виконання програми

Завдання для практичного виконання

1,*а*. Написати програму обчислення виразу для 3-х значень змінної «а» дійсних чисел згідно з варіантом завдання ***з передаванням параметрів у процедурі*** з коментарями до кожної команди. Результат виконання операції вивести на екран одним рядком функцією MessageBox та підписом результату.

1*б*. Задати масив із чотирьох дійсних чисел: перше число в масиві – A, друге – B, третє – C, четверте – D. Обчислити вираз із використанням вбудованих masm32-функций та вивести результат через MessageBox.

Варіант 1

1. $\sqrt{a} - bc/d$;
2. $a/bc + \sqrt{d}$;
3. $a/b\sqrt{c} - d$;
4. $a - b/c\sqrt{d}$;
5. $ab/\sqrt{c} - d$;
6. $2/a + b\sqrt{(cd)}$;
7. $ab/c + \sqrt{d}$;
8. $\sqrt{(ab)} + c/d$;
9. $\sqrt{a} + (b/c)\,d$.
10. $a\sqrt{b} + c/d$.

Варіант 2

1. $2^a b$;
2. $2^{-a}/b$;
3. $3^a/\sqrt{b}$;
4. $4^{-a}/\sqrt{c}$;
5. $5^a\sqrt{b}$;
6. $2a + 6^b$;
7. $2^a/7b$;
8. $8b/8^{-c}$;
9. $ab + 9^b$;
10. $b2^{-a}$.

2. Написати програму обчислення виразу для 5-х значень змінної «а» дійсних чисел згідно з варіантом завдання ***з зовнішньою процедурою*** та коментарями до кожної команди. Результат виконання операції вивести на екран вертикально однією функцією MessageBox та підписом до кожного числа результату.

1. $2^a - \sqrt{b}/ac$;
2. $2^a/b + a\sqrt{c}$;
3. $3a/\sqrt{b} - a2^d$;
4. $2a - 2^b/\sqrt{c}$;
5. $5a\sqrt{b} - 2^c$;
6. $2^a + 6b\sqrt{c}$;
7. $2^a/7b + b\sqrt{c}$;
8. $\sqrt{a} + 8b/2^c$;
9. $ab + 2^b/9c$;
10. $b2^a + 10b/\sqrt{c}$,

3. Написати програму обчислення виразу для n значень (n = № + 4, де № – номер студента в його навчальній групі) змінної x дійсних чисел згідно з варіантом завдання з використанням ***вбудованих функцій masm32*** та коментарями до кожної команди. Результати вивести на екран функцією MessageBox з вертикально-горизонтальним розташуванням чисел. У випадку, якщо результат перевищує якесь значення,– вивести аварійне повідомлення з графічним позначенням цієї функції.

Варіант 1

1. $(1 - \sin 5x)/\sqrt{a}$;
2. $[\sin(x + 30^o)]b/c$;
3. $10^a/\sqrt{3} - \sqrt{2}(\cos x)$;
4. $a^b \sin[(x + y)/2]$;
5. $a\sqrt{b} - (\sin 2x)/2^a$;
6. $10^b \sin 2x + \operatorname{tg} x$;
7. $3a + 4\cos 2x$;
8. $a^b - \cos(x + y)/2$;
9. $1/(1 + (1/\sin x)) + a$;
10. $10^a/\sqrt{b} - \sin x + \cos x$.

Варіант 2

1. $(\lg 2a + 2^b)/(a/4 + 2)$;
2. $(a^2 - \ln b + 5)/3^a - 1$;
3. $(2a - \lg(b/2))/(\ln(1 - 4a)$;
4. $(3\lg a - b/2)/3^b + 1$;
5. $(a/b + \ln(3/a))/ 5^a - 7$;
6. $(2^a - \ln(b/2))a^2 - 1$;
7. $(\lg(10 - 2a/b))/a^b - 2$;
8. $7(\ln(a - 1))/a^2 + ab$;
9. $(\lg(4a - b))/(a^b - 1)$;
10. $(\ln(ab + 2))/2^a - b^2$.

4. Написати програму обчислення виразу для n значень (n = № + 3, де № – номер студента в його навчальній групі) змінної x дійсних чисел згідно з варіантом завдання. Результати вивести на екран функцією MessageBox з вертикально-горизонтальним розташуванням чисел та графічним виведенням інформаційності повідомлення. У випадку, якщо результат перевищує якесь значення, вивести аварійне повідомлення з графічним позначенням цієї функції.

1. $(1/a)\ln|ax + \sqrt{(a^2x^2 + b^2)}|$;
2. $(a - b)\ln|\sqrt{(a + x^2)} + \sqrt{(b + x)}|$;
3. $(a + b)\arcsin\sqrt{[(x + b)/(a + b^2)]}$;
4. $(a - b)\arccos\sqrt{[(x + b)/(a^2 - b)]}$;
5. $(1/a^2)\sqrt{(a^2x^2 + b^2)}$;
6. $(1/2a^3)(-ax\sqrt{(b^2 - a^2x^2)} + \arcsin ax$;
7. $\sqrt{(b^2 - a^2x^2)} + \arccos ax$;
8. $2 - b^3 \arcsin|b/ax|$;
9. $b \ln|b - \sqrt{(b^3 - a^2x^2)}/x|$;
10. $\sqrt{(b^3 - a^2x^2)}/\arcsin ax$.

5. Написати програму доведення того, що $\sin x + \cos x = 0$.

6. Написати програму доведення того, що $\sin^2 x + \cos x + 1 = 0$

Контрольні запитання та завдання

1. Для чого призначений арифметичний співпроцесор?
2. Які типи даних використовує співпроцесор?
3. Яку архітектуру має співпроцесор?
4. Які класи команд має співпроцесор?
5. Що являють собою вбудовані masm32-функції співпроцесора?

6. На які групи команд розділяються команди передання даних співпроцесора?

7. На які групи команд розділяються арифметичні команди співпроцесора?

8. Що являють собою додаткові арифметичні команди співпроцесора?

9. Визначити результат виконання фрагмента програми:

```
_a DD 3.
_b dd 2.
_c dd 1.
…
fld _a
fld _b
fld _c
invoke  FpuAdd,2,3,0, SRC1_DIMM or SRC2_FPU or DEST_FPU
```

а) 1; б) 2; в) 5; г) 4; д) 3.

10. Визначити результат виконаня фрагменту програми:

```
a1 dt 3.
b1 dt 2.
c1 dt 1.
…
fld c1
fld b1
fld a1
invoke FpuMul,ADDR a1,1,2,SRC1_REAL or SRC2_FPU or DEST_FPU
invoke  FpuAdd,0,2,0, SRC1_FPU or SRC2_DIMM or DEST_FPU
```

а) 6; б) 7; в) 9; г) 10; д) 11.

13. РЯДКИ

13.1. Обробка рядків

Рядкові команди дозволяють проводити дії над блоками пам'яті, що є послідовностями елементів розміру:

– 8 бітів, тобто байт;

– 16 бітів, тобто слово;

– 32 біти, тобто подвійне слово;

– 64 біти, тобто слово збільшене у четверо.

Існує п'ять елементарних команд роботи з рядками, які наведено у табл. 13.1 (http://www.intel.com/techtrends/technologies/hyperthreading. htm, документ 25366717.pdf).

Таблиця 13.1

Команда	Опис	Операція
MOVS(B/W/D/Q)	Переміщення рядків даних (DB, DW, DD, DQ) з однієї частини пам'яті в іншу, поки RCX $\neq 0$	[RDI] ← [RSI]
STOS(B/W/D/Q)	Збереження вмісту регістра AL/AX/EAX/RAX у пам'яті, комірка якої визначається регістром RDI	[DI] ← AL [DI] ← AX [EDI] ← EAX [RDI] ← RAX
CMPS(B/W/D/Q)	Порівняння у пам'яті [RDI] – [RSI], поки RCX $\neq 0$	[RDI] – [RSI]
SCAS(B/W/D Q)	Сканування рядків. Порівняння байта/слова/ подвійного слово/збільшеного у четверо з пам'яті за адресою RDI зі вмістом AL (SCASB), AX (SCASW), EAX (SCASD), RAX (SCASQ) та встановлення відповідних прапорців (аналогічно команді CMPS). Команда сама визначає розмір операнда залежно від типу	[DI] – AL [DI] – AX [EDI] – EAX [RDI] – RAX
LODS(B/W/D/Q)	Збереження у регістрі AL/AX/EAX/RAX з пам'яті, адреса якої визначається вмістом регістра RSI	AL ← [RSI] AX ← [RSI] EAX ← [RSI] RAX ← [RSI]

Кожна з рядкових команд залежно від розміру елементів може **додатково використовувати** такі літери: **B** – робота з байтом; **W** – робота зі словом; **D** – робота з подвійним словом, **Q** – зі збільшеним учетверо.

Особливістю рядкових команд є те, що вони, окрім обробки поточного елемента рядка, коректують вміст відповідних регістрів для того, щоб автоматично просунутися до наступнонр елемента рядка.

Без префікса повторення команди обробки рядків виконуються тільки один раз.

Однобайтові команди обробки рядків, крім одноразового їх виконання, можуть бути виконані й багаторазово, якщо команді передує **префікс-повторювач** (мнемокоди: REP, REPE, REPZ, REPNE, REPNZ). Префікс-повторювач вказує, що команда повинна повторюватися доти, доки лічильник у регістрі RCX не буде вичерпаний. Якщо вміст лічильника в регістрі RCX дорівнює нулю, то операція виконуватись не буде. Префікс повторення визначає також величину, що повинна порівнюватися з бітом ZF. Якщо елементарна команда впливає на біт ZF і його значення не дорівнює позначеній величині після деякої кількості повторень операції, то її повторення припиняється. Це дозволяє організувати цикл типу WHILE.

Окремі операції можуть поєднуватись для формування складних операцій над рядками. У таких випадках повторення забезпечується операціями ітерації. Наявність префікса дозволяє передавати однією командою рядок з однієї ділянки пам'яті в іншу при відповідній ініціалізації регістрів RSI, RDI, RCX. Для адресації операндів використовуються індексні регістри RSI та RDI. Якщо біт DF = 0 (скинутий), то вміст індексних регістрів збільшується після кожної операції над байтами на одиницю, а після операції над словами – на два. Якщо встановлений біт DF, то вміст індексних регістрів операнда і результату зменшується після кожної операції.

Під час виконання повторюваної елементарної команди індексні регістри RSI, RDI і лічильник повторень (RCX) змінюються після кожного повторення, а лічильник команд буде, як і раніше, містити зміщення байта префікса (у припущенні, що він безпосередньо передує команді над рядками). Перервана команда над рядками буде вірно продовжена після возврат управления з переривальної програми, якщо перед елементарною командою розміщений тільки один префікс. Тому варто уникати використання двох інших префіксів разом із префіксом-повторювачем у командах роботи з рядками, тобто префікса сегмента і префікса LOCK. Це пов'язано з тим, що після виходу з переривальної програми виконання відновиться з адреси на один байт меншої від адреси команди (тобто адреси, у якій приблизно розміщений префікс-повторювач, а додаткові префікси будуть ігноруватись).

Префікс REP використовується з командами MOVS і STOS і ініціює дію «повторювати, поки не буде досягнутий кінець ланцюжка», тобто доти, доки вміст регістра RCX не досягне нуля.

Префікси REPE і REPZ діють аналогічно і є таким же байтом, як і префікс REP. Вони використовуються з командами CMPS і SCAS і оперують із прапорцем ZF = 1, стан якого визначається результатом виконання цих команд. Префікси REPNE і REPNZ діють аналогічно попереднім префіксам, але прапорець ZF повинен дорівнювати нулю. Інакше повторення закінчується.

13.2. Команда MOVS(B/W/D/Q)

Команда MOVS(B/W/D/Q) пересилає байт (MOVSB), слово (MOVSW), подвійне слово (MOVSD) або MOVSQ з поля, що вказується регістром RSI, в поле, що задається регістром RDI (MOVS [RDI],[RSI]), поки RCX ≠ 0.

При використанні команди з префіксом REP команда MOVS здійснює блокове передавання пам'ять – пам'ять ланцюжка (поки RCX ≠ 0), елементами якого є байти. А залежно від прапорця напрямку значення в регістрах RDI, RSI або збільшується, або зменшується.

Прапорець напрямку. Рядкові команди використовують прапорець напрямку DF регістра Flags, щоб визначити, який із регістрів (RSI або RDI) буде інкрементуватись або декрементуватись при виконанні команди.

Значення прапорця	Дія на RSI або RDI	Напрямок
0	Інкрементується	від низу до верху
1	Декрементується	зверху вниз

Прапорець напрямку змінюється командами CLD і STD:

```
CLD   ; скинути  прапорець напрямку,   напрямок – вверх (DF = 0)
STD   ; встановити  прапорець  напрямку,   напрямок – вниз (DF = 1)
```

Наступна програма переміщує 10 байтів із ділянки пам'яті mas1 в ділянку пам'яті mas2:

```
    CLD ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
    MOV ESI, offset mas1  ; занесення адреси  передавача
    MOV EDI, offset mas2      ; занесення адреси  приймача
    MOV ECX, 10               ; встановлює лічильник в 10
REP  MOVSB                    ; переміщення байтів (B) поки ECX ≠ 0
    …
  mas1 DB 'ABCDEFGHIJ'      ; резервування в пам'яті комірок
                    ; розміром у байт для кожної літери масиву mas1
  mas2 DB 10 dup(?)         ; резервування в пам'яті десяти комірок
              ; розміром у байт кожна для збереження невизначених
                                                  ; елементів  масиву mas2
```

Наступні дві програми показують переваги використання команди обробки рядків.

Приклад 13.1. Скопіювати всі значення з масиву mas1 в масив mas2 **без використання** команд обробки рядків.

Лістинг 13.1. Програма виконання прикладу 13.1:

```
     title CopyRight by Rysovaniy A. N.          rysov@rambler.ru
    .686                ; директива определения типа микропроцессора
.model  flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
     option casemap:none  ; отличие строчных и прописных букв
includelib\masm32\lib\kernel32.lib
include \masm32\include\kernel32.inc ; файли функцій застосувань…
  .data                             ; директива определения данных
```

```
    mas1 dw -1h,5h,10h,15h,20h  ; визначення масиву з ім'ям mas1
    mas2 dw 5 dup(1) ; масив mas2, всі елементи якого дорівнюють 1
.code                           ; директива начала  сегмента данных
_st:                            ; метка начала программы с именем_st
      mov ecx,5     ; кількість слів у масиві
      lea esi, mas1 ; завантаження адреси масиву mas2
      lea edi, mas2 ; завантаження адреси масиву mas2
@@:  mov bx,[esi]             ; завантаження числа з масиву mas1
      mov [edi],bx   ; завантаження з bx в пам'ять,
                              ; адреса комірки якої знаходиться в Рг EDI
      add esi,2    ; вирівнювання вмісту Рг ESI на границю слова
      add edi,2    ; вирівнювання вмісту Рг EDI на границю слова
    loop @b        ; перейти на m1, якщо ECX не нуль
invoke ExitProcess,0 ; возврат управления  ОС и освобождение ресурсов
      end _st        ; директива завершение программы с именем_st
```

Початкові значення дампа пам'яті даних для наочності наведено на рис. 13.1.

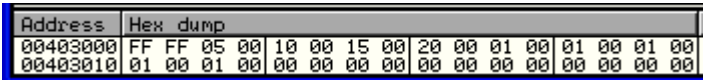


Рис. 13.1. Початкові значення дампа пам'яті даних прикладу 13.1

Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.1 без використання рядкових команд наведено на рис. 13.2.

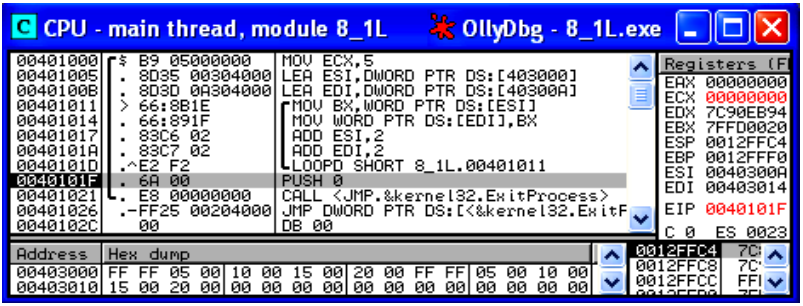


Рис. 13.2. Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.1 без використання рядкових команд

Для наочності заповнення масиву mas2 елементами масиву mas1 використовується заповнення нового масиву одиницями розміром у слово.

Приклад 13.2. Скопіювати всі значення з масиву mas1 в масив mas2 з ***використанням*** команд обробки рядків.

Лістинг 13.2. Програма виконання прикладу 13.1 з використанням команд обробки рядків:

```
title CopyRight by Rysovaniy A. N.
.686                    ; директива определения типа микропроцессора
```

```
        .model  flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none          ; отличие строчных и прописных букв
        includelib\masm32\lib\kernel32.lib
        include \masm32\include\kernel32.inc ; файли функцій застосувань…
  .data                                          ; директива определения данных
        mas1 dw -1h,5h,10h,15h,20h ; визначення масиву з ім'ям mas1
        len equ ($-mas1)/type mas1   ; обчислення кількості елементів mas1
        mas2 dw len dup(1)           ; масив mas2, всі елементи якого дорівнюють 1
.code                                ; директива начала  сегмента данных
_st:                                  ; метка начала программы с именем_st
        mov ecx,len                  ; кількість елементів у масиві
        lea esi, mas1                ; завантаження адреси масиву mas1
        lea edi, mas2                ; завантаження адреси масиву mas2
  cld ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
   rep movsw          ; переміщення слів (W), поки ECX ≠ 0
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
     end _st                 ; директива завершение программы с именем_st
```

Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.2 з використанням команд обробки рядків наведено на рис. 13.3.

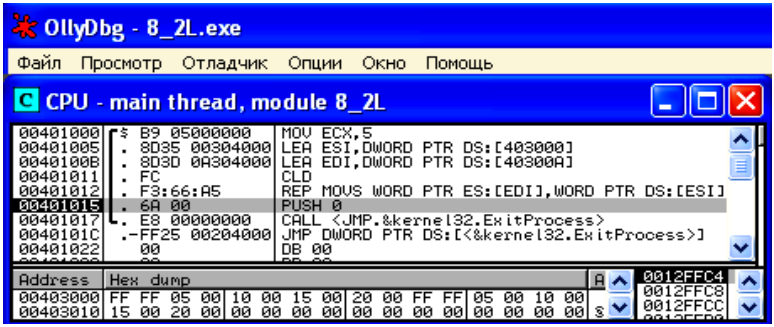


Рис. 13.3. Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.2 з використанням команд обробки рядків

Масив даних mas2 розташовується, починаючи з адреси 0040300Ah, та повністю повторює елементи масиву mas1.

За допомогою команди MOVS можна об'єднати (зчепити) два масиви даних, що означає, що до кінця одного масиву потрібно пристикувати другий масив.

Приклад 13.3. Є два масиви. Потрібно перший масив зчепити з другим.

Лістинг 13.3. Програма виконання прикладу 13.3 з виконанням зчеплення масивів:

```
        title CopyRight by Rysovaniy A. N.   зчеплення символьних рядків
        .686            ; директива определения команд микропроцессора
        .model  flat,stdcall             ; задание линейной модели памяти
        option casemap:none     ; отличие строчных и прописных букв
        includelib\masm32\lib\kernel32.lib
        include \masm32\include\kernel32.inc ; системные функции
.data
     str1 db 'STUDY OF LINE COMMANDS ' ; масив байтів з іменем str1
```

```
                                                    ; із символами коду ASCII
    len_str1 equ $-str1             ; визначення байтів у рядку str1
    str2 db ' APPLICATION '
    len_str2 equ $-str2             ; визначення байтів у рядку str2
.code                                   ; директива начала  сегмента данных
_st:                                    ; метка начала программы с именем_st
    lea esi, str1                ; завантаження адреси масиву str1
    lea edi, str2+len_str2    ; адреса нового масиву
    mov ecx, len_str1       ; встановити в лічильник max значення літер
  cld ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
      rep movsb     ; повторювати поки ECX <> 0
invoke ExitProcess,0 ; возврат управления  ОС и освобождение ресурсов
end _st                          ; директива завершение программы с именем_st
```

Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.3 з використанням команд обробки рядків наведено на рис. 13.4. У пам'яті даних наведено результат виконання програми.

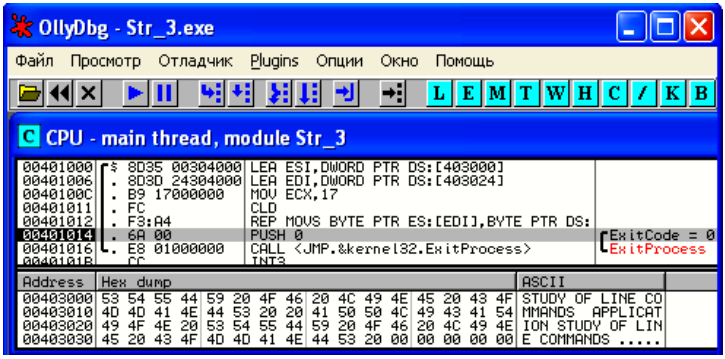


Рис. 13.4. Вікна налагоджувача OllyDbg при використанні зчеплення масивів програми з лістингу 13.4

Поряд зі зчепленням є й таке поняття як **конкатенація**. Конкатенація означає зчеплення (об'єднання) з урахуванням розміру операндів. Таку програму наведено в лістингу 13.4.

Лістинг 13.4. Програма виконання прикладу 13.3 з виконанням об'єднання масивів:

```
    title CopyRight by Rysovaniy A. N.,   об'єднання символьних рядків
    .686                        ; директива определения команд микропроцессора
    .model  flat,stdcall                    ; задание линейной модели памяти
    option casemap:none                     ; отличие строчных и прописных букв
    includelib\masm32\lib\kernel32.lib
    include \masm32\include\kernel32.inc ; системные функции
.data
    str1  dd  1.0, 2.1, 3.2   ; масив з іменем str1 32-розрядних чисел
    len_str1  equ  $-str1    ; визначення байтів у рядку str1
    str2  dd  0.5, 2.0, 3.0
    len_str2  equ  $-str2  ; визначення байтів у рядку str2
.code                                   ; директива начала  сегмента данных
_st:                                    ; метка начала программы с именем_st
    mov ecx,len_str1     ; кількість байтів str1
```

```
        shr ecx,2               ; кількість 32-розрядних слів масиву str1
        lea esi, str1           ; завантаження адреси масиву str1
        lea edi, str2+len_str2  ; адреса нового масиву
    cld ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
        rep movsd               ; повторювати, поки ECX <> 0
invoke ExitProcess,0 ; возврат управления  ОС и освобождение ресурсов
end _st                         ; директива завершение программы с именем_st
```

Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.3 з використанням об'єднання масивів наведено на рис. 13.5.

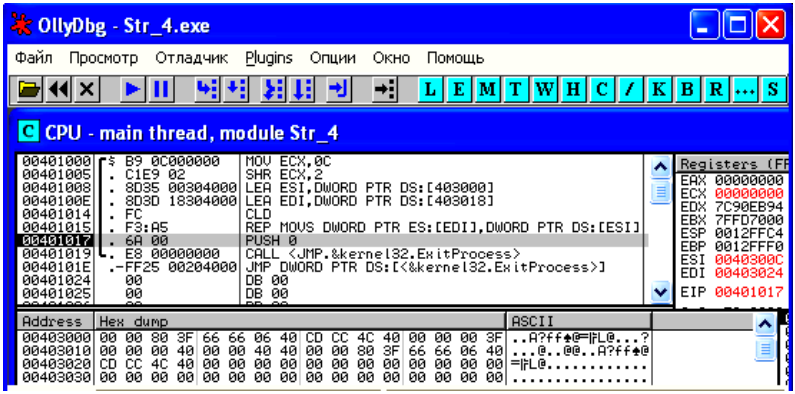


Рис. 13.5. Вікна налагоджувача OllyDbg при використанні об'єднання масивів програми з лістингу 13.4

У програмі використано масив дійсних чисел. Для виведення масиву дійсних чисел у консоль необхідно виконати складні перетворення: виділити 80-розрядні комірки; записати числа через регістри співпроцесора у пам'ять; здійснити виведення.

13.3. Команда CMPS(B/W/D/Q)

Команда CMPS здійснює порівняння (віднімання) байта (слова, подвійного слова, збільшеного учетверо), який адресується другим операндом з [RSI], з байтом (словом, подвійним словом, збільшеним учетверо), який адресується першим операндом з [RDI] (CMPS [RDI], [RSI]), поки RCX ≠ 0.

Команда CMPS впливає на біти умов, але не формує результат. Вона призначена для порівняння рядків. При відповідному префіксі-повторювачі можливо визначити, після якого елемента два рядки починають відрізнятись один від одного і, таким чином, установити порядок, в якому вони повинні розміщуватись.

Коли перед командою CMPS знаходиться префікс REPE (або REPZ), операція інтерпретується як «порівнювати, доки не буде досягнутий кінець ланцюжків або доки елементи ланцюжків не будуть рівні». За наявності префікса

REPNE (або REPNZ) операція трактується як «порівнювати не до кінця, а до першого виконання умови слідкувальної команди (або доки елементи ланцюжків будуть рівні)». Таким чином, команду CMPS зручно застосовувати для знаходження однакових або таких, що розрізняються, елементів ланцюжків.

Прапорець напрямку визначає інкремент або декремент регістрів RSI і RDI.

Таким чином, можуть бути використані такі префікси повторення:

```
REPE,   REPZ          ; повторювати, поки відправник  = одержувачу
REPNE, REPNZ          ; повторювати, поки відправник >< одержувача
```

Якщо рядок операнда-відправника менший, ніж рядок операнда-одержувача, то CF = 1. Якщо рядки рівні, то ZF = 1.

Якщо рядок операнда-відправника більший, ніж рядок операнда-одержувача, то ZF = 0 та CF = 0. Умовні переходи, що використовуються з командою CMPS, наведено в табл. 13.2.

Таблиця 13.2

Умова	Без знака	Із знаком
Одержувач < відправник	JB	JL
Одержувач <= відправник	JBE	JLE
Одержувач < > відправник	JNE (JNZ)	JNE (JNZ)
Одержувач = відправник	JE (JZ)	JE (JZ)
Одержувач >= відправник	JAE	JGE
Одержувач > відправник	JA	JG

Приклад 13.4. Порівняти літери двох змінних. У випадку, якщо кількість літер у першій змінній менша, ніж кількість літер другої змінної, то записати у комірку пам'яті res ← 1. А якщо ні,то у res завантажити 0.

Лістинг 13.5. Програма виконання прикладу 13.3:

```
    .686            ; директива определения типа микропроцессора
    .model  flat,stdcall  ; задание линейной модели памяти и соглашения ОС Windows
    option casemap:none       ; отличие строчных и прописных букв
    includelib\masm32\lib\kernel32.lib
    include \masm32\include\kernel32.inc ; системные функции
  .data                                 ; директива определения данных
mas1 db 'MARTIN'   ; збереження в пам'яті з ім'ям mas1 слова
                                        ; MARTIN, кожна літера якого займає один байт
mas2 db 'MARTINEZ' ; визначення масиву з ім'ям mas2
       len equ $-mas2 ; визначення кількості байтів у масиві mas2
       res db (0FFh)    ; наперед визначена комірка зі значенням 0FFh
.code                                   ; директива начала  сегмента данных
_st:                                    ; метка начала программы с именем_st
       mov ecx,len                      ; кількість слів у масиві
       lea edi,  mas1                   ; завантаження адреси масиву mas1
       lea esi,  mas2                   ; завантаження адреси масиву mas2
   cld ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
      repe cmpsb                ; повторювати, поки CF = 0 та ECX > 0
        jl min                  ; перехід, якщо mas1 (i) < mas2 (i)
```

```
        mov res, 0              ; res ← 0, якщо mas1 (i) > mas2 (i)
        jmp m1                  ; безумовний перехід на  мітку m1
  min: mov res, 1               ; res ← 1, якщо mas1 (i) < mas2 (i)
    m1:
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
       end _st                  ; директива завершение программы с именем_st
```

Початкові значення дампа пам'яті даних для наочності наведено на рис. 13.6.

Address	Hex dump	ASCII
00403000	4D 41 52 54 49 4E 4D 41 52 54 49 4E 45 5A FF 00	MARTINMARTINEZ .
00403010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

Рис. 13.6. Початкові значення дампа пам'яті даних прикладу 13.4

Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.4 наведено на рис. 13.7. Для наочності початкове значення комірки пам'яті при запису результату res процесу порівняння становить FFh. Для наочності досліджень рядкових команд у налагоджувачі OllyDbg при невеликих масивах зручніше для покрокового режиму використати кнопку F7, тому що можна прослідкувати процес зміни лічильника ECX та порівняння масивів. При натисканні на F8 рядкова команда виконується одразу.

Порівняння чисел із знаком. Команда CMPS може порівнювати два списки цілих чисел із знаком до першого незбіжного числа.

Приклад 13.5. Порівняти два списки цілих чисел із знаком до першого незбіжного числа. Якщо перший масив чисел менший, ніж другий, то записати в комірку пам'яті результату одиницю, а якщо ні, то записати 0.

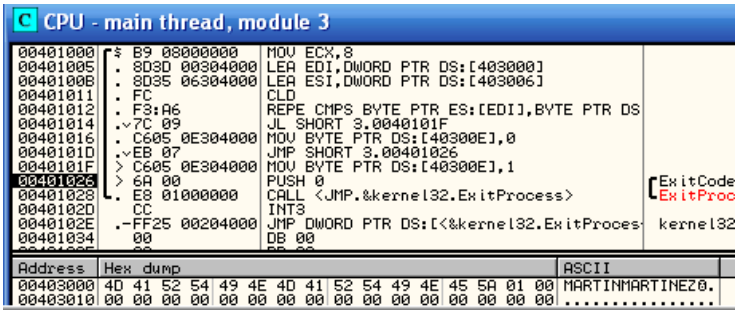


Рис. 13.7. Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.4

Лістинг 13.6. Програма виконання прикладу 13.5:

```
        .686                    ; директива определения типа микропроцессора
        .model  flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none             ; отличие строчных и прописных букв
        includelib\masm32\lib\kernel32.lib
        include \masm32\include\kernel32.inc ; системные функции
.data                           ; директива определения данных
    mas1 db 12,2,-3,4                   ; визначення масиву з ім'ям mas1
```

```
    mas2 db 12,2,3,-4                ; визначення масиву з ім'ям mas2
    res db (17)                      ; резервування комірки під результат
  .code                              ; директива начала  сегмента данных
  _st:                               ; метка начала программы с именем_st
       mov ecx,4                     ; кількість слів у масиві
       lea esi, mas1                 ; завантаження адреси масиву mas1
       lea edi, mas2                 ; завантаження адреси масиву mas2
  cld ; напрямок – вверх (DF = 0) - від менших номерів комірок до більших
    repe cmpsb                       ; повторювати, поки CF = 0 та CX > 0
       jl min                        ; перехід, якщо mas1 (i) < mas2 (i)
       mov res, 0                    ; res ← 0, якщо mas1 (i) > mas2 (i)
       jmp m1                        ; безумовний перехід на  мітку m1
min: mov res, 1                      ; res ← 1, якщо mas1 (i) < mas2 (i)
m1:
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
       end _st                       ; директива завершение программы с именем_st
```

Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.5 наведено на рис. 13.8.

У прикладі кожне число в масиві *a*1 порівнюється з відповідними числами з масиву *a*2. Початкова частина масиву *a*1 вважається меншою, оскільки його третє число буде –3, а відповідне число з масиву *a*2 буде дорівнювати 3. Числа в пам'яті комп'ютера зберігаються у додаткових числах. Порівняння четвертих елементів масивів не відбувається.

Результат порівняння записаний у комірку 0040300Ch.

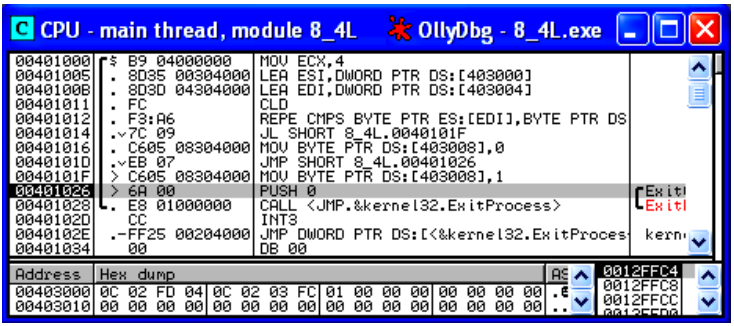


Рис. 13.8. Вікна налагоджувача OllyDbg при налагодженні програми

13.4. Команда SCAS(B/W/D/Q)

Команда SCAS порівнює вміст AL (SCASB), AX (SCASW), EAX (SCASD), RAX (SCASQ) з байтом/словом/подвійним словом/збільшеним учетверо з пам'яті за адресою RDI та встановлює відповідні прапорці (аналогічно команді CMPS), й сама визначає розмір операнда залежно від типу.

Ця команда має такий синтаксис: **SCAS** ; [RDI], RAX. Після її виконання RDI збільшується на одиницю.

У випадку ітеративного використання команду SCAS можна застосовувати для пошуку входження деякої величини в рядок.

З префіксом REPE (або REPZ) цю команду можна використовувати для пошуку елемента ланцюжка із значенням, що відрізняється від заданого значення. Якщо перед командою SCAS знаходиться префікс REPNE (або REPNZ), то операція інтерпретується як «переглядати доти, доки не буде досягнутий кінець ланцюжка або поки значення елемента ланцюжка не буде дорівнювати значенню, що відшукується».

Час виконання команди SCAS без префікса повторення становить 15 тактів, а з префіксом повторення – 9 + 15* (кількість повторень) тактів синхронізації.

Для явного зазнначення типу елементів ланцюжка зазвичай використовуються мнемокоди SCASB і SCASW; при цьому операнд у команді відсутній.

Сканування для збіжних символів

Приклад 13.6. Сканувати рядок символів доти, доки символ F не буде знайдений.

Коли символ F буде виявлений, регістр EDI указуватиме на адресу першого байта після цього символу, і значення цього регістра потрібно декременувати, щоб він указував на символ F.

Відразу після того, як команду SCASB буде виконано, регістр EDI вкаже на символ, що іде за шуканим символом, а регістр ECX – на номер розряду, який дорівнює ‘f’, рахуючи справа та починаючи від нуля.

Виконання прикладу 13.6 з використанням виведення даних у спрощене віконце наведено у лістингу 13.7.

Лістинг 13.7:

```
        title CopyRight by Rysovaniy A. N.
        .686                            ; директива определения типа микропроцессора
.model  flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
option casemap:none                 ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\kernel32.inc ; системные функции
include \masm32\include\\user32.inc   ; файлы  интерфейса …
          includelib \masm32\lib\kernel32.lib
          includelib \masm32\lib\user32.lib
.data
        mas1 db ‘abcdefgh’        ; масив байтів із символами коду ASCII
        len equ $-mas1              ; визначення кількості байтів у масиві mas1
        titl db " Результат ",0      ; назва вікна
        buf db ?,0                  ; буфер виведення повідомлення
ifmt db "Буква f находится на  %d  позиции  в слове abcdefgh", 0dh,0ah,0ah,
           "Автор программы: ",0      ; текстове повідомлення
.code       ; директива начала  сегмента данных
_st:        ; метка начала программы с именем_st
    lea edi, mas1     ; завантаження адреси масиву mas1
    mov al, ‘f’         ; завантаження символу ‘f’
    mov ecx, len      ; встановити в лічильник max значення літер
    cld ; напрямок – вверх (DF = 0) - від менших номерів комірок до більших
```

```
      repne scasb       ; повторювати, поки не буде дорівнювати
        dec edi           ; знайдений: декременувати DI
        mov eax,len
        sub eax,ecx
invoke  wsprintf, ADDR buf, ADDR ifmt, eax ;  регістр, вміст якого перетворюється
invoke MessageBox, 0,addr buf, addr titl,  MB_OK
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
      end _st                   ; директива завершение программы с именем_st
```

Результат виконання програми прикладу 13.6 з використанням виведення даних наведено на рис. 13.9. У спрощене віконце виводиться вміст регістра EAX, який містить номер розряду збігу символів, рахуючи ліворуч.

Рис. 13.9. Вікна налагоджувача OllyDbg при налагодженні програми

У програму для зручності виведення тексту включено коди керування курсором:

09 – табуляція;

10 – переведення каретки;

13 – повернення каретки – повернення курсора на початок рядка.

Використання команд умовного переходу. Відразу після виконання команди CMPSB ще не можна визначити, що саме стало причиною її завершення: те, що ECX = 0, або те, що відповідний символ вже знайдений, причому і те, і інше може відбутися одночасно.

При використанні префікса REPNZ зазвичай використовують команду JNZ для переходу, коли символ знайдений.

У цьому випадку керуються таким принципом: якщо прапорець нуля скинутий, то символ не знайдений, а якщо прапорець нуля встановлений, то символ знайдений:

```
    repnz  cmpsb
    jnz not_found
```

або:

```
    repnz cmpsb
    jz found
```

Може виникнути помилка при використанні команди JCXZ під час ухвалення рішення після того, як символ знайдений. Символ може бути знайдений у найостанній позиції, і регістр CX стане дорівнювати нулю після останнього порівняння. Тому наступні команди будуть помилковими:

```
repnz scasb
jcxz exit
```

Приклад 13.7. Задано рядки *A* і *B* (константи типу BYTE). У всіх байтах рядка *A* замінити символи *X* на відповідні символи із масиву *B*.

Лістинг 13.8. Програма виконання прикладу 13.7:

```
        title CopyRight by Rysovaniy A. N.
        .686                    ; директива определения типа микропроцессора
.model  flat,stdcall  ; задание линейной модели памяти и соглашения ОС Windows
option casemap:none         ; отличие строчных и прописных букв
includelib\masm32\lib\kernel32.lib
include \masm32\include\kernel32.inc ; системные функции
.data                                   ; директива определения данных
        mas1 db 'atxfx'         ; масив байтів із символами коду ASCII
        mas2 db 'abcef'
.code                           ; директива начала  сегмента данных
_st:                            ; метка начала программы с именем_st
    cld ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
        lea edi,mas1            ; завантаження адреси масиву A в edi
        lea esi,mas2            ; завантаження адреси масиву B в esi
        mov al,'x'                      ; завантаження символу 'x'
        mov ecx,5                       ; кількість слів у масиві
  a2: scasb                             ; [edi] – al та [edi] + 1
        jnz m1                          ; перейти, якщо не нуль
        dec edi                         ; повернення edi
        mov bl,[esi]
        mov [edi],bl    ; movsb
        inc edi                 ; встановлення edi для порівняння
m1:  inc esi            ; просування esi – підготуваннння до наступного циклу
     loop a2                            ; цикл, поки cx ≠ 0
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
        end _st         ; директива завершение программы с именем_st
```

Початкові значення дампа пам'яті даних для наочності наведено на рис. 13.10.

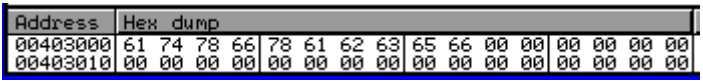


Рис. 13.10. Початкові значення дампа пам'яті даних для прикладу 13.7

Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.7 наведено на рис. 13.11.

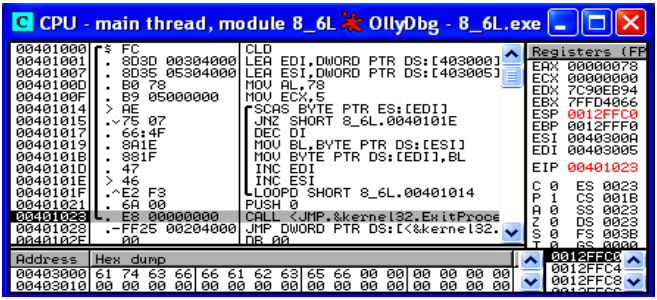


Рис. 13.11. Вікна налагоджувача OllyDbg при налагодженні програми

При порівнянні дампа пам'яті до та після виконання порівняння видно, які байти поміняно.

Видалення додаткових пропусків. Команда SCASB може використовуватися для видалення зайвих пропусків у рядку з кодуванням ASCIIZ. У наступному прикладі регістри ES:DI указують на останній байт рядка, CX дорівнює довжині рядка, AL = 20h (символ пропуску) і DF = 1. Для повторення циклу використовується префікс REPE, доки значення регістра AL збігається із значенням у пам'яті.

```
    .data
dest db "DESTINATION STRING     ",0
destlen = $ - dest                  ; розрахунок довжини в байтах
    .code
      mov di, seg dest
      mov es, di
      mov di,offset dest                ; встановити зсув рядка
      add di, destlen                   ; вказати на останній байт
      mov ex, destlen                   ; встановити лічильник
      mov al, 20h                       ; порівняння з пропусками
      std                               ; напрямок – вниз
repe scasb                              ; сканування рядка
      jnz exit                          ; пропусків більше немає
      dec di                            ; скоректувати DI, якщо CX = 0
exit:
    mov byte ptr es:[di + 2],0          ; вставити нульовий байт
```

Як тільки буде знайдено перший символ, що не дорівнює пропуску, то відразу після нього вставляється нульовий байт, внаслідок чого змінюється довжина рядка. В даному випадку необхідна команда

```
mov byte  ptr [di + 2],0
```

Після виконання команди SCASB регістр DI указуватиме на останній символ перед початком пропусків. Для отримання позиції наступного символу (тобто початку пропусків) необхідно до DI додати 2.

Підрахування пропусків у словах. Для виконання цих дій завантажується знак пропуску командою **mov al, ' '**, в якій знак пропуску задається не через символ 20h, а в лапках.

Приклад 13.8. Задано текст із 35 символів. Визначити кількість слів, які містять більше 3-х символів. Слова розділяються одним пропуском.

Лістинг 13.9. Підрахування пропусків у словах:

```
      title Лєбєдєва Олена, НТУ ХПІ, КІТ29Б
      .686                          ; директива определения команд микропроцессора
      .model  flat,stdcall      ; задание линейной модели памяти
      option casemap:none  ; відмінність рядкових та великих літер
include C:\masm32\include\windows.inc ; файлы структур, констант …
include C:\masm32\include\kernel32.inc ; системные функции
      include C:\masm32\include\\user32.inc ; файлы  интерфейса …
```

```
        includelib C:\masm32\lib\user32.lib
        includelib C:\masm32\lib\kernel32.lib
.data                   ; директива определения данных
s1 db 'Lesson of programming Assemler MASM', 0 ; визначення рядка
    _count dd 0         ; лічильник
    quan dd 4           ; умова кількості символів у слові+1 (пропуск)
    _d dd 35            ; кількість символів у рядку
    n dd 0              ; кількість пройдених символів у рядку
        _title db "Результати розв'язання програми",0
info db "Задано текст із 35 символів. Визначити кількість слів,", 0Ah, 0Dh,
"які містять більше 3-х символів. Слова розділяються одним пропуском.",0Ah,0Dh,0Dh,
        "'Lesson of programming Assemler MASM'",0Ah,0Dh,0Dh
        buf dd ?,0          ; буфер виведення повідомлення
        ifmt db "Кількість слів, в яких більше 3-х літер= %u", 0
.code                                   ; початок сегмента-даних
  _st:                                  ; метка начала программы
        mov ecx, _d         ; кількість слів у рядку
        lea edi, s1             ; завантаження адреси масиву mas2
        mov al, ' '             ; завантаження знака пропуску 20h
        cld             ; напрямок – вверх (DF = 0)
m:
repne scasb ; повторювати порівняння [edi] – al та [edi] + 1, поки не дорівнює
        ; в ecx - кількість символів, що залишилися
        mov ebx, _d             ; ebx:=35
        sub ebx, ecx            ; визначення літер у рядку
        sub ebx, n              ; визначення кількості літер у слові
        add n, ebx          ; визначення кількості пройдених символів у рядку
        cmp ebx, quan       ; порівняння кількості літер з умовою
        ja m1               ; перейти на мітку, якщо в слові понад 3 літери
        cmp ecx, 0
        jnz m           ; якщо рядок не скінчився, то перейти на m
m1:         inc _count      ; збільшення лічильника слів
        cmp ecx, 0
        jnz m           ; якщо рядок не скінчився, то перейти на m
m3:  invoke wsprintf,addr buf ,addr ifmt, _count
invoke MessageBox,NULL, addr info, addr _title,MB_ICONINFORMATION
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
        end _st         ; директива завершение программы с именем_st
```

Вікна налагоджувача OllyDbg при налагодженні програми з лістингу 13.10 наведено на рис. 13.12.

При знаходженні в рядку символу пропуску, регістр EBX указуватиме на адресу першого байта наступного слова. Спочатку з ebx (кількість символів у рядку) віднімається кількість символів за даний прохід до пробілу, а потім віднімається кількість символів за усі попередні проходи. Таким чином визначається кількість символів поточного слова.

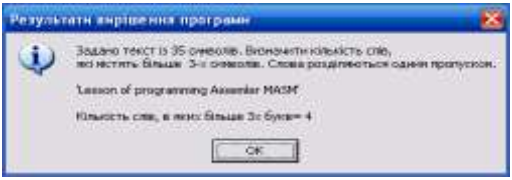


Рис. 12.12. Вікна налагоджувача OllyDbg при налагодженні програми

13.5. Команда LODS(B/W/D/Q)

Команда LODS пересилає операнд у регістр AL/AX/EAX/RAX із пам'яті, адреса якого знаходиться в RSI (LODS AL/AX/EAX/RAX, [RSI]). Зазвичай ця команда не використовує префікс-повторювач.

У регістрі RSI міститься адреса операнда-відправника. З командою LODSB використовується 8-розрядний операнд, з командою LODSW – 16-розрядний операнд, з командою LODSD – 32-розрядний, з командою LODSQ – 64-розрядний. Прапорець напрямку визначає інкрементування або декрементування регістрів RSI і RDI.

Префікси повторення рідко використовуються з командами LODS, тому що кожне нове значення, що завантажується в регістри AL/AX/EAX/RAX, руйнує їх колишні значення. Замість окремої команди LODSB можна використовувати такі дві команди:

```
mov al, [si] ; перемістити  байт за адресою DS:SI  у AL
inc  si          ; вказати  на  наступний байт
```

Приклад 13.9. Просканувати масив та при цьому очистити старший біт кожного байта, і отримане значення зберегти в другому масиві.

Лістинг 13.10. Програма виконання прикладу 13.9:

```
        title CopyRight by Rysovaniy A. N.
        .686                            ; директива определения типа микропроцессора
.model  flat,stdcall   ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none             ; отличие строчных и прописных букв
        includelib\masm32\lib\kernel32.lib
        ExitProcess proto :DWORD        ; прототип API-функції
        .data                                   ; директива определения данных
mas1 db 0C8h, 0FBh, 0F5h, 0CAh, 41h, 42h, 43h, 64h, 87h, 8Ch
len1 equ $-mas1
mas2 db 10 dup(1)               ; попереднє заповнення масиву одиницями
.code                                   ; директива начала  сегмента данных
_st:                                    ; метка начала программы с именем_st
        cld                     ; напрямок –  вверх
        lea esi,mas1            ; завантаження адреси масиву
        lea edi,mas2            ; завантаження адреси масиву
        mov ecx, len1   ; кількість слів у масиві
    cld ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
L1:  lodsb              ; AL < - [SI]
        and al, 7Fh     ; логічне перемноження
        stosb           ; [edi] <- al]
        loop L1         ; цикл, поки CX ≠ 0
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
        end _st          ; директива завершение программы с именем_st
```

Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.9 наведено на рис. 13.13.

Попереднє заповнення масиву *output* одиницями корисно використовувати на етапі налагодження програми для візуального контролю заповнення елементів пам'яті.

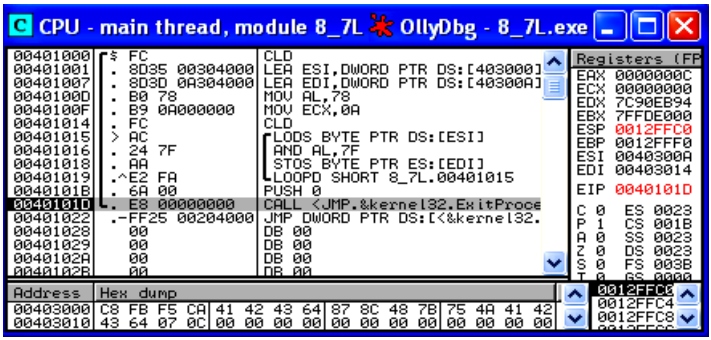


Рис. 13.13. Вікна відображення пам'яті налагоджувача OllyDbg при налагодженні програми прикладу 13.9

13.6. Команда STOS(B/W/D/Q)

Команда STOS пересилає операнд із регістра AL/AX/EAX/RAX за адресою RDI (STOS [RDI], AL/AX/EAX/RAX). За допомогою префікса-повторювача вона може використовуватись для заповнення рядка визначеною величиною.

З префіксом повторення ця команда є зручним засобом ініціалізації ланцюжка на фіксоване значення, наприклад нуль або пропуск. Час виконання команди STOS без префікса повторення становить 11 тактів, а з префіксом повторення 9 + 10 * (кількість повторень) тактів синхронізації.

Регістр RDI містить адресу операнда-одержувача. З командою STOSB використовується 8-розрядний операнд, з командою STOSW – 16-розрядний операнд, з командою STOSD – 32-розрядний, з командою STOSQ – 64-розрядний.

Прапорець напрямку визначає інкрементування або декрементування регістрів RSI та RDI.

Також можна використовувати команду STOS для ініціалізації пам'яті єдиним значенням.

Приклад 13.10. Заповнити кожен байт рядка mas1 значенням 0FFh.

Лістинг 13.11. Програма виконання прикладу 13.10:

```
        .686                    ; директива определения типа микропроцессора
.model  flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none     ; отличие строчных и прописных букв
        includelib\masm32\lib\kernel32.lib
include \masm32\include\kernel32.inc ; системные функции
        .data                                   ; директива определения данных
```

```
mas1 db 10 dup(1)          ; попереднє заповнення масиву одиницями
.code                                   ; директива начала  сегмента данных
  _st:                                  ; метка начала программы с именем_st
        lea edi,mas1                    ; завантаження адреси масиву в edi
        mov al,0FFh
        mov ecx,10                      ; кількість слів у масиві
    cld ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
        rep stosb                       ; заповнити вмістом AL
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
        end _st                         ; директива завершение программы с именем_st
```

Початкові значення дампа пам'яті даних для наочності наведено на рис. 13.14.

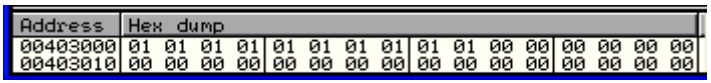


Рис. 13.14. Початкові значення дампа пам'яті даних прикладу 13.9

Вікна налагоджувача OllyDbg при налагодженні програми прикладу 13.10 наведено на рис. 13.15.

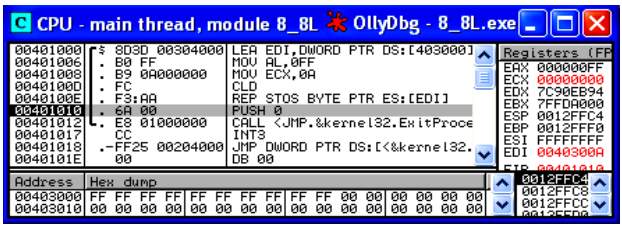


Рис. 13.15. Вікна налагоджувача OllyDbg при налагодженні програми

Приклад 13.11. Необхідно порівняти два масиви байтів, що знаходяться в пам'яті, і при рівності елементів вийти з циклу.

Припустимо, що регістр SI адресує поточний елемент першого масиву, регістр DI – поточний елемент другого масиву, а вміст регістра CX є довжиною масивів. Тоді стандартна програма порівняння масивів байтів набуває такого вигляду:

```
A1:  MOV  AL, [SI]          ; елемент першого масиву
     CMP  [DI], AL          ; порівняти
     JZ    A2               ; елементи рівні
     INC   SI               ; просунути вказівники
     INC   DI
     DEC  CX                ; перевірити закінчення
     JNZ   A1
A2:  …
```

При розподілі регістрів завдання порівняння масивів розв'язується значно коротшим фрагментом:

```
REPNZ   CMPSB
        JZ   A2
```

Команда STOS є зручною для ініціалізації ділянкою даних на конкретне значення. Наступна програма скидає масив слів із початковою адресою ARRAY і довжиною, що визначається змінною LENGTH:

```
        LES   DI, ARRAY             ; адреса в ES: DI
        MOV  CX,  LENGTH            ; довжина в CX
        MOV   AX, 0                 ; початкове значення
REP   STOSW                         ; ініціалізація
```

13.7. Лабораторна робота «Рядки»

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм, написаних мовою асемблера, з використанням команд обробки рядків та API-функцій під Win32 для програмування МП платформи x86.

Постановка задачі

Згідно з цифрою номера студента в групі вибрати свій варіант та написати програму з використанням **команд обробки рядків**, з обов'язковим виведення у **файл** та й у **спрощене віконце**. Дані, які виводяться, необхідно підписати, вивести умову завдання та довідку про автора програми.

1. Задано текст, що складається з 6 слів по 8 символів, розділених пропуском. Переставити слова в тексті так, щоб кожне наступне слово починалося з тієї букви, на яку закінчилося попереднє. Перше слово залишити на місці.

2. Задано послідовність букв у вигляді прізвища студента. Розставити їх за абеткою.

3. Задано текст у вигляді прізвища, імені та по батькові, а також e-mail студента. Ущільнити текст, залишивши між словами по одному пропуску.

4. Задано текст із 20 символів. Визначити кількість повторень поєднань «*ОР*» в тексті і замінити такі повторення символом «!».

5. Задано текст із 40 символів. Визначити кількість слів, які містять більше 4-х символів. Слова розділяються одним пропуском.

6. Задано текст із 8 слів із різною кількістю символів. У словах із парним номером змінити порядок букв на зворотний.

7. Побудувати послідовність з n ($n \le 30$) символів такого вигляду: *АББВВВГГГГДДДДДЕЕЕЕЕЕ...*

8. Задано текст із 34 символів, що складається із слів, розділених одним пропуском. Визначити кількість слів, в яких буква *Е* зустрічається понад 2 рази.

9. Задано послідовність із 37 символів. Визначити частоту повторення кожного символу.

10. Задано 8 слів по 6 символів. На початку кожного слова записано номер із двох символів. Розставити слова за збільшенням номерів.

11. Задано текст із 15 слів, розділених пропуском. Визначити кількість слів, в яких буква *А* зустрічається більше 3-х разів.

12. Задано текст із 10 слів, розділених пропусками (одним і більше). Визначити кількість слів, що містять більше 4 символів.

13. Задано текст із 8 слів, розділених пропуском. Визначити кількість повторень букви *Е* в кожному слові.

14. Задано текст із 12 слів, розділених пропуском. Розставити слова відповідно до латинського алфавіту.

15. Задано текст із 26 символів. Визначити кількість різних символів і частоту їх повторень.

16. Задано текст, що складається з 5 слів по 7 символів. Розставити слова відповідно до російського алфавіту.

17. Задано текст, що складається з 10 слів, розділених деякою кількістю пропусків. Визначити кількість букв *А* в кожному слові.

18. Задано текст, що складається з 8 слів по 5 символів. Визначити кількість голосних букв у кожному слові.

19. Задано текст, що складається з 4 слів по 8 символів. Визначити кількість різних букв у кожному слові.

20. Задано текст, що складається з 7 слів по 5 символів. Видалити слова, що містять більше 3-х букв *О*.

21. Задано текст, розділений на слова пропусками. Змінити порядок букв у словах на протилежний.

22. Задано текст із 25 символів. Знайти слова, порядок букв в яких зворотний щодо першого слова. (Слова розділено пропусками).

23. Задано текст із 18 символів: 3 слова по 6 символів. Здійснити кільцевий зсув кожного слова ліворуч: 1-го – на 1 символ; 2-го – на 2 символи; 3-го – на 3 символи.

24. Побудувати послідовність вигляду *АББССС* із 30 символів відповідно до заданої таблиці, що містить символ і кількість його повторень. Наприклад: *А* – 1 раз; *Б* – 2 рази; *С* – 3 рази і т.ін.

25. Задано текст із 30 символів, які можуть бути поділені пропусками та розділовими знаками. У кожному слові рядка замінити першу букву на останню, другу на передостанню і т.ін.

26. Ввести довільний символьний рядок. Розділити його на два рядки: рядок голосних і рядок приголосних. Перший рядок вивести на екран, а другий – у файл.

Зміст звіту

1. Постановка задачі для конкретного варіанта.

2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.

3. Лістинг програми з виведенням даних на екран монітора з використанням API-функцій та детальним коментарем і описом роботи.

4. Print screen екрана 32-розрядного налагоджувача з виконаною програмою.

5. Короткий опис виконання програми.

6. Висновки за результатами роботи.

Приклад 13.12. Задано текст, розділений на слова пропусками. Змінити порядок букв у словах на протилежний.

Лістинг 13.12:

```
        .686                    ; директива определения типа микропроцессора
.model  flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none             ; отличие строчных и прописных букв
include \masm32\include\kernel32.inc ; системные функции применений…
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\\user32.inc   ; файлы  интерфейса …
        includelib\masm32\lib\kernel32.lib
        includelib\masm32\lib\user32.lib
        BSIZE equ 30            ; задание реального количества байтов
.data                           ; директива определения данных
        mas1 db ‘abc def ghk lmn’
        mas2 db 15 dup(0)
cWritten DWORD ?   ; для адреси символів виведення
stdout DWORD ?      ; для збереження дескриптора виведення
.code                           ; директива начала кода программы
_start:                         ; метка начала программы с именем _start
    cld ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
        lea edi,mas1   ; адреса першого масиву в edi
        mov esi, 0
        mov edx,0
        mov al,’ ‘              ; заносимо в al пропуск
        mov ecx,21
        push ecx
        jmp m2
m4:  mov ecx,4                  ; лічильник слів
m1:  mov ah, [ebx]  ; заносимо в ah останній символ масиву
        mov mas2[edx],ah        ; зберігаємо цей символ
        inc edx                 ; збільшуємо індекс
        dec ebx                 ; перехід на наступний елемент
        cmp [ebx],esi
        dec ecx
        jnz m1                  ;
m2:  pop ecx
        repne scasb  ; шукаємо пропуск в масиві
        mov ebx,edi         ; зберігаємо в ebx місце, де був знайдений пропуск
        dec ebx
        push ecx
        dec ecx
        jnz m4
invoke GetStdHandle, STD_OUTPUT_HANDLE ; одержання дескриптора
       mov stdout,eax           ; збереження одержаного дескриптора
invoke  WriteConsoleA, \  ; API-функція виведення на екран
 stdout, \              ; дескриптор стандартного пристрою виведення
  ADDR mas2,\               ; адреса рядка, яка містить  текст повідомлення
   sizeof mas2+1, \     ; адреса рядка, яка містить  заголовок повідомлення
    ADDR cWritten, 0   ; адреса, де зберігається кількість символів
```

```
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
end _start                          ; директива окончания программы
```

Результат виконання програми з лістингу 13.13 наведено на рис. 13.16.

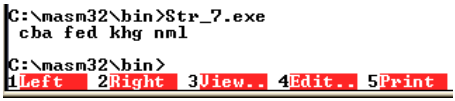


Рис. 13.16. Результат виконання програми з лістингу 13.13

Приклад 13.13. Побудувати послідовність вигляду ААААБББССССС із 40 символів відповідно до заданої таблиці, що містить символ і кількість його повторень.

Лістинг 13.13:

```
      title Рысованый А.Н.
     .686                           ;директива определения типа микропроцессора
     .model flat, stdcall                    ;завдання лінійної моделі памяті
option casemap:none             ;відмінність малих та великих літер
include \masm32\include\kernel32.inc ; системные функции
include \masm32\include\windows.inc ; файлы структур, констант …
   include \masm32\include\\user32.inc   ; файлы  интерфейса …
   includelib \masm32\lib\user32.lib
   includelib \masm32\lib\kernel32.lib
.data                           ; директива определения данных
  info db "Масив = "            ; підпис числа для віконця результату
  mas BYTE 45 dup (?),0         ; визначення масиву
  c1 db 61h                     ; c1 := 61h = 97 – код літери “a”
  titl db " Результат ",0       ; назва віконця
  .code                         ; директива начала кода программы
  _start:                       ; метка начала программы с именем _start
     mov ebx,40       ; EBX := 28h
     mov edx,3        ; EDX := 3 - кількість початкових однакових літер
     lea edi,mas            ; занесення адреси початку рядка до EDI
  m1:  mov al,c1                ; al := c1
  mov ecx,edx                   ; ECX := EDX
  cld                           ; напрямок – вверх
  rep stosb                     ; [di]<- al, заповнити [di] вмістом al
  mov ecx,edx                   ; ECX := EDX
  inc c1                        ; збільшення коду літери
  inc edx                       ; збільшення кількості повторень літери
  sub ebx,ecx         ; обчислення кількості повторень, які залишилися
  cmp ebx,0                     ; порівняння вмісту EBX з нулем
  jnl m1                        ; якщо не менше – перейти на m1
  jmp m2                        ; інакше – на m2
m2:
invoke MessageBox,0, ADDR info,\     ; адреса рядка, яка містить  текст повідомлення
        ADDR titl,\         ; адреса рядка, яка містить  заголовок повідомлення
          MB_ICONINFORMATION +180000h         ; вигляд діалогового вікна
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
        end _start                          ; директива окончания программы
```

Результат виконання програми з лістингу 13.14 наведено на рис. 13.17.

Рис. 13.17. Результат виконання
програми з лістингу 13.13

У програмі початок перебирання починається з коду літери *a*, а потім у циклі цей код збільшується, після чого підраховується кількість літер, яка не повинна перевищувати 40.

Приклад 13.14. Задано рядки A та B. Знайти перші числа, що не співпадають та переписати наступні числа рядка A в рядок C.

Лістинг 13.14:

```
        .686            ; директива определения команд микропроцессора
        .model  flat,stdcall ; задание линейной модели памяти
option casemap:none ; відмінність рядкових та великих літер
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\include\kernel32.inc ; системные функции применений…
        include \masm32\include\\user32.inc   ; файлы  интерфейса …
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
.data                                   ; директива определения данных
        mas1 dd 0Ah,0Bh,1,2,3,4         ; визначення масиву з ім'ям mas1
        mas2 dd 0Ah,0Ch,5,6,7,8         ; визначення масиву з ім'ям mas2
len equ ($-mas2)/ type mas2 ; обчислення кількості подвійних слів в mas2
        mas3 dd len DUP(?),0 ; резервування 32-розрядних комірок для mas3
        _title db "Результаты решения программы",0
info db "Заданы строки A и B. Найти первые несовпадающие числа", 0Ah, 0Dh,
"и переписать последующие числа строки A в строку C",0Ah,0Dh,0Dh
buf dd ?,0                              ; буфер виведення повідомлення
        ifmt  db "Ответ: mas3 =  %d  %d  %d  %d  %d  %d",0
.code                                   ; початок сегмента-даних
_st:                                    ; метка начала программы
        mov ecx,len   ; кількість слів у масиві
        lea esi, mas1 ; завантаження адреси масиву mas2
        lea edi, mas2                   ; завантаження адреси масиву mas2
cld ; напрямок – вверх (DF = 0) – від менших номерів комірок до більших
repne cmpsd           ; [esi] - [edi] поки не дорівнює
        jz ravno      ; перехід, якщо source = destination
        jmp exit
ravno:  lea edi,mas3                    ; завантаження адреси масиву mas3
        dec ecx                         ;
        add esi,4                       ; підготовка вибірки наступного числа
        rep movsd                       ; edi<- esi
invoke  wsprintf,ADDR buf, \; адреса буф., куди буде записана послідовність символів
   ADDR ifmt,  mas3,mas3[4],mas3[8],mas3[12],mas3[16],mas3[20] ;
invoke MessageBox, NULL, addr info, addr _title, MB_ICONINFORMATION
exit:
invoke ExitProcess,0  ; возврат управления  ОС и освобождение ресурсов
        end _st       ; директива завершение программы с именем_st
```

У програмі порівнюються два масиви: A та B. При порівнянні чисел mas1: 0Bh та mas2: 0Ch виявляється, що числа не дорівнюють один одному. Тому слід наступні числа рядка A (1, 2, 3, 4) переписати в рядок C (mas3). Для цього використовується команда add esi,4 збільшення адреси числа mas1.

Для того щоб числа при виведенні функцією MessageBox не розташовувались один біля одного, в рядку форматування даних

ifmt db "Ответ: mas3 = %d %d %d %d %d %d",0

зроблено по два пропуски (рис. 13.18).

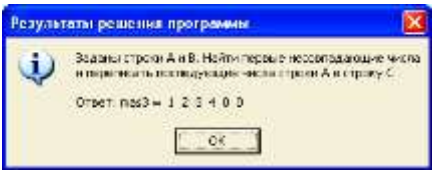


Рис. 13.18. Результат виконання програми з лістингу 13.15

Приклад 13.15. Задано текст із 32 символів, що складається із слів, розділених одним пропуском. Визначити кількість слів і кількість букв у кожному слові.

Лістинг 13.15:

```
        .686
        .MODEL flat, stdcall
        option casemap: none
        include \masm32\include\windows.inc
        include \masm32\include\kernel32.inc
        include \masm32\include\user32.inc
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
.data
        input db "Входная строка состоит из 35 знаков",0
        sz equ ($-input)
        words dd 0
        symbols dd 0
        tmp dd 0
        ttl db "Результат выполнения программы",0
        format db "Слово №%d, состоит из %d символов",0
        len dd 0
        filename db "C:\\outputa.txt",0
        fHnd dd ?
        buf db ?
.CODE
        strlen proc arg:dword
            mov edi,arg
            mov ecx, -1
            mov al, 0
            repne scasb
            not ecx
            dec ecx
            ret
            strlen endp
_start:
```

```
        mov edx,offset input
        add edx,sz
        lea edi,input
        lea esi,buf
        mov ecx,sz
cycle:
        cld
        mov al,20h
        repne scasb
        inc words
        mov ebx,edi
        sub ebx,offset input
        sub ebx,tmp
        add tmp,ebx
        sub ebx,1
        mov symbols,ebx
          push ecx
          push edi
          push esi
invoke wsprintf,addr[esi],addr format,words,symbols
          pop esi
          pop edi
          pop ecx
        add esi,eax
        mov al,0ah
        mov [esi],al
        inc esi
        cmp ecx,0
    jnz cycle
invoke strlen,addr buf
        mov len,ecx
invoke CreateFile,addr filename,GENERIC_WRITE,0,0, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL,0
        mov fHnd,eax
invoke WriteFile,fHnd,addr buf,len,addr len,0
        invoke MessageBox,0,addr buf,addr ttl,MB_OK
        jnz cycle
        END   _start
```

Результат виконання програми наведено на рис. 13.19

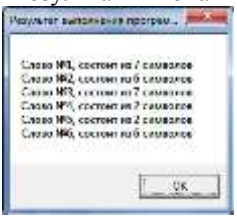


Рис. 13.19. Результат виконання програми

Контрольні запитання

1. Які команди належать до команд обробки рядків?
2. В яких випадках використовується префікс команд та для чого він призначений?
3. Яка різниця між зчепленням та конкатенацією?
4. Яка різниця між командами CMPS та SCAS?

14. МАКРОВИЗНАЧЕННЯ

14.1. Введення в макровизначення

Макровизначення – це одне або декілька стверджень мови асемблера, яким привласнено символічне ім'я і які можна викликати по цьому імені. Макроси використовуються для ділянок програми, що часто ***повторюються***. При виклику макровизначення копія команд, що подають дане макровизначення, вставляється в програму.

Перевагою застосування макровизначення є те, що програма виконується швидше, ніж ***процедура***. Це відбувається тому, що в макросах відсутні команди CALL і RET, які примушують процесор організовувати перехід на іншу гілку і робити повернення з неї в початкову точку.

Недоліком макровизначення є те, що при частому його використанні збільшується об'єм програмного коду. Це відбувається тому, що кожен виклик макровизначення вставляє в програму код самого макровизначення.

Після того як макровизначення буде описане, його можна викликати з будь-якого місця програми. Вставка кодів відбувається при першому проходженні асемблером початкового файлу (http://www.ht-group.net/27/).

14.1.1. Порівняльний аналіз процедур і макрозасобів

Дії (фрагменти), що повторюються, в програмі можна описати і як процедуру, і як макровизначення. При цьому в обох випадках ділянка коду, що повторюється, описана тільки один раз.

Після трансляції процедура так і залишається описаною один раз, а тіло макровизначення підставляється у всі місця виклику і тим самим збільшує розмір програми.

Тому:

1. Застосування процедур робить код компактнішим, тобто економиться пам'ять.

При зверненні до процедури:

– виконується засилання параметрів в регістри або стек;

– запам'ятовується адреса повернення;

– здійснюється перехід;

– після закінчення роботи процедури відновлюється адреса повернення;

– очищаються регістри або стек і тому подібне

При роботі процедури витрачається час на переходи і передачу параметрів під час виконання програми.

При заміні макрокоманд на макророзширення теж витрачається час, але це відбувається на етапі трансляції, а не під час виконання програми.

2. Застосування макрозасобів економить час виконання програми.

Тому в програмах критичних за часом слід застосовувати макрозасоби, а якщо необхідно економити пам'ять слід застосовувати процедури.

Якщо в ділянці коду, що повторюється, багато команд (тобто великий фрагмент) краще описати його як процедуру. Якщо ж невелику групу команд описати процедурою, то число допоміжних команд по її виклику і передачі параметрів стане порівнянним з числом команд самої процедури, її час виконання стане на багато більше.

3. Великі ділянки коди рекомендується описувати як процедури, а маленькі – як макровизначення.

Параметрами процедур можуть бути тільки операнди команд, а параметрами макрокоманд можуть бути будь-які послідовності символів, у тому числі і самі команди.

14.1.2. Місця використання макровизначень

Макровизначення можуть використовуватися в трьох варіантах:

1. На початку програми до сегменту кода з тим, щоб не погіршувати читабельність програми.

Цей варіант слід застосовувати у випадках, якщо визначувані макрокоманди актуальні тільки в межах однієї цієї програми.

2. У окремому файлі.

Цей варіант підходить при роботі над декількома програмами однієї проблемної області. Щоб зробити доступними ці макровизначення в конкретній програмі, необхідно на початку початкового тексту цієї програми записати директиву include имя_файла.inc

3. У макробібліотеці.

У макробібліотеці розміщуються макрокоманди, які використовуються практично у всіх програмах. Під'єднування цієї бібліотеки здійснюється за допомогою директиви include.

Недолік двох останніх способів в тому, що в початковий текст програми включаються абсолютно всі макровизначення.

Для виправлення ситуації можна використовувати директиву purge, як операнди якої через кому перераховуються імена макрокоманд, які не повинні включатися в текст програми.

В макроасемблері masm32 використовується так багато макровизначень, що одне їх перерахування займе дуже багато місця. Тому в цьому розділі наведені тільки загальні відомості про них.

Існує два типи макросів: **макрофункції та макропроцедури.**

Макропроцедури при виклику просто замінюються на послідовність команд, викликаються аналогічно звичайній команді, і може бути викликані тільки на початку рядка.

Макрофункції відрізняються тим, що можуть повертати якесь значення і можуть викликатися у виразах. Параметри виклику вказуються в круглих дужках. Для вказівки макрофункції усередині макросу використовується директива "EXITM текст, що <повертається>".

Макроси можуть мати локальні змінні. Оголошуються вони перерахуванням через кому після слова "local". Вони існують тільки в самому макросі, і тому не створюють конфлікт з вже оголошеними в інших місцях змінними з таким же ім'ям (http://subscribe.ru/archive/ comp.soft.prog.asm32/ 200702/ 11080945.html).

Зазвичай макрокоманда містить **список аргументів**.

Залежно від місця розташування макрокоманди значення цих аргументів може мінятися (наприклад, якщо в якості аргументів використані регістри загкального призначення). Процес такого заміщення називається макрогенерацією, а результатом цього процесу є макророзширення.

14.2. Директиви MACRO і ENDM

Макровизначення може бути створене в будь-якому місці програми за допомогою директив MACRO і ENDM.

Директива MACRO позначає початок визначення макросу.

Директива ENDM позначає закінчення визначення макросу.

Синтаксис:

```
mIм'я MACRO parameter_1, parameter_2...
    список команд
        ENDM
```

Мала буква "m" в префіксі імені застосовується для зручності ідентифікації макровизначення.

Для виділення ***коментарів*** в макровизначеннях двічі ставиться крапка з комою (;;). При цьому коментарі з'являються тільки в написаному макровизначенні, а не в тих фрагментах, які вставляються в тіло програми.

Приклад 14.1. Скласти програму знаходження максимального елемента для трьох чисел за допомогою макросів.

Лістинг 14.1. Програма виконання прикладу 14.1:

```
.686                    ; директива определения типа микропроцессора
.model  flat, stdcall ; задание линейной модели памяти
option casemap:none    ; отличие строчных и прописных букв
includelib\masm32\lib\kernel32.lib
ExitProcess proto :DWORD        ;  прототип API-функції
mMax macro x,y,z                ; макрос з ім'ям mMax
    local m1,m2
    mov eax,x                   ;; завантаження першого операнда
    cmp eax,y                   ;; завантаження другого операнда
```

```
            jge m1                      ;; перейти на m1, якщо op1 >= op2
            mov ebx,y                   ;; збереження у тимчасовому регістрі
            mov v,ebx                   ;; v ← y   як більше число
            jmp m2                      ;; перейти на закінчення
    m1:
           mov v,eax                    ;; v ← x   як більше число
    m2:
          endm                          ;; закінчення макросу
.data                                           ; директива определения данных
    x dd 4 ; записати в 32-розрядну комірку пам'яті число x = 4
    y dd 3 ; записати в 32-розрядну комірку пам'яті число y = 3
    z dd 5 ; записати в 32-розрядну комірку пам'яті число z = 5
    v dd ? ; резервувати в пам'яті 32-розрядну комірку для змінної v
  .code                                     ; директива начала программы
_start:                                     ; метка начала программы с именем _start
       echo Hello!!!        ; виведення повідомлення при компіляції
       mMax [x],[y],[v]                     ; виклик макросу mMax
       mMax [v],[z],[v]                     ; виклик макросу mMax
invoke ExitProcess, 0 ; повернення управління  ОС  та вивільнення ресурсів
       end _start           ; директива завершение программы с именем _start
```

В програмі для виведення повідомлення використано макровизначення **echo**, яке виводить повідомлення **при компіляції**, наприклад через FAR та має синтаксис:

ECHO message

У макровизначенні є можливість ***включення параметрів***, наприклад (http://www.ht-group.net/27/):

MyMacro macro param0, param1:REQ, param2 := <0>,param3:VARARG

де:

Param0 – приклад визначення параметра;

Param1:REQ – ключове слово REQ указує на те, що цей параметр обов'язковий. Якщо він не буде вказаний, то генерується помилка на етапі компіляції;

Param2:=<0> – приклад параметра, який має значення за умовчанням. Якщо цей параметр не буде вказаний, то він буде дорівнювати цьому значенню.

Param3:**vararg** – стає ім'ям параметра, який сприймає все, що залишилося далі, як рядок. Параметри типу vararg зазвичай використовуються при створенні макросів із змінною кількістю параметрів. Після параметра з директивою vararg не можливо оголосити інші параметри.

Якщо якийсь параметр не визначено, то запис може бути таким:

MyMacro param1, ,param3

Макрос VARARG – це передача параметрів списком, який здійснює передачу через стек. Виклик відповідає конвенції cdecl.

Оскільки стек очищається процесом, який викликає, то він може реалізовувати функції зі змінним числом аргументів. Угода про виклики __cdecl створює виконувані файли більшого розміру, ніж __stdcall, оскільки для нього необхідно, щоб виклик функції містив код очищення стека.

Приклад використання макросу vararg приведений в лістингу 14.2.

Лістинг 14.2. Приклад використання макросу vararg:

```
        .686
        .model  flat, stdcall
        option casemap:none
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib user32,kernel32,masm32
        AddDigs proto C :dword, :vararg
.data
        buffer db 64 dup (0)
.code
start:
        fn AddDigs,2,3,4
        fn wsprintf, offset buffer, chr$("%d"), eax
        fn MessageBox, 0, offset buffer, offset buffer, 0
            fn AddDigs,4,3,4,5,6
            fn wsprintf, offset buffer, chr$("%d"), eax
            fn MessageBox, 0, offset buffer, offset buffer, 0
            fn ExitProcess,0
        AddDigs proc C argc:dword, argv:vararg
           xor eax, eax
           xor ecx, ecx
        @@:
           add eax, argv[ecx * 4]
           inc ecx
           cmp ecx, argc
           jl @B
           ret
        AddDigs endp
        end start
```

Результат виконання програми наведено на рис. 14.1.

Рис. 14.1. Результат виконання програми

В цій програмі спочатку виконується функція fn AddDigs з параметрами 2, 3, 4. При безпосередньому виконанні цієї функції визначено, що перший аргумент визначений як argc:dword. А інші параметри – є список параметрів після першого параметра. Це числа 3 та 4.

При виконанні цієї функції з параметрами fn AddDigs,4,3,4,5,6 також визначено, що перший параметр – це число 4. А інші параметри – є список.

Кожен параметр (аргумент) є значенням, яке передається в макровизначення. Це значення замінює відповідні параметри в макровизначенні при його

виклику. Порядок аргументів повинен відповідати порядку параметрів, але кількість аргументів не обов'язково дорівнює кількості цих параметрів. Директива умови IFB в асемблері дозволяє виконувати перевірку пропущених аргументів.

Приклад макрофункції:

```
  mXorReg macro reg     ;; макрофункція з параметром reg
    xor reg,reg
    EXITM <reg>
   ENDM
  …
  push mXorReg (eax) ; виклик макро
```

откомпілюється, як:

```
  xor eax, eax
  push eax
```

Приклад макрофункції:

```
mZeroReg macro param1:=<eax>
  xor param1,param1
 ENDM
.data                          ; директива определения данных
RReg equ edx
.code
 _start:
echo Hello!!!          ; виведення повідомлення при компіляції
 mZeroReg              ; параметр невказаний (xor eax,eax)
 mZeroReg ecx        ; xor ecx,ecx
 mZeroReg RReg     ; xor edx,edx
```

Приклад макрофункції:

```
moveXx macro dest,src
  mov e&dest&x,e&src&x
 ENDM
.code
_start:
echo Hello!!!     ; виведення повідомлення при компіляції
 moveXx a,b
```

откомпілюється, як:

```
  mov eax, ebx
```

Приклад макрофункції:

```
ZeroRegs macro reglist:vararg
  FOR param:=<eax>,<reglist>
    xor param,param
  ENDM
 ENDM
.code
_start:
ZeroRegs ebx,,ecx
```

откомпілюється в xor ebx,ebx / xor eax,eax и xor ecx,ecx

Приклад макрофункції:

```
mZeroRegs macro reglist:vararg
  FOR param,<eax,ebx,ecx,edx,esi,edi>
```

```
        xor param,param
      ENDM
    ENDM
.code
_start:
    mZeroRegs
```

откомпілюється як xor з eax до edi.

Для підключення бібліотек функцій та бібліотек прототипів з константами в masm32 є макрос **USELIB**, який, крім бібліотеки windows.inc, дозволяє підключити їх одним рядком. В такому випадку початок програми може бути наступним:

```
.686                    ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти
option casemap:none     ; отличие строчных и прописных букв
include \masm32\include\windows.inc     ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib kernel32, fpu
```

14.3. Директива LOCAL

Директива LOCAL позначає тимчасові імена міток, які замінюються в процесі повторного макророзширення на унікальні імена.

При багатократному використанні макросу і за наявності внутрішніх міток в ньому необхідно використовувати директиву **LOCAL** <мітка> (див. розділ "Процедури"). В цьому випадку компілятор формує кожного разу унікальну мітку. Директива LOCAL повинна бути записана відразу після заголовка макровизначення. При кожному наступному розширенні числова частина мітки отримуватиме приріст 1.

Синтаксис:

LOCAL мітка [, мітка] …

14.4. Булеві вирази

У асемблері MASM дозволені оператори відносин такі ж, як й при використанні директив високого рівня: ==, != , > , >=, <, <= та інші.

Приклад 14.2. Написати програму з використанням макросу, який виконує множення числа на число, причому, якщо це число є степенем двійки та не перевищує 16, то множення буде виконуватися за більш швидкою схемою множення за допомогою зсувів.

Ця програма може бути такою (лістинг 14.2):

Лістинг 14.2. Програма виконання прикладу 14.2:

```
.686                    ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти
option casemap:none     ; отличие строчных и прописных букв
include \masm32\include\windows.inc     ; файлы структур, констант …
```

```
    include \masm32\macros\macros.asm
    uselib kernel32, user32
  mMulD macro digit          ; макрос з ім'ям mMulD
      .IF digit == 2  ;; якщо digit = 2, то
        shl eax,1             ;; зсув ліворуч на  1 розряд
      .ELSEIF digit == 4      ;; якщо digit = 4, то
        shl eax,2     ;; зсув ліворуч на 2 розряди
      .ELSEIF digit == 8      ;; якщо digit = 8, то
        shl eax,3     ;; зсув ліворуч на 3 розряди
      .ELSEIF digit == 16  ;; якщо digit = 16, то
        shl eax,4     ;; зсув ліворуч на 4 розряди
      .ELSE                   ;; інакше
      mov ebx, digit
      mul ebx       ;; eax, edx = eax*ebx – звичайне перемноження
      .ENDIF
  endm                        ;; закінчення макросу
.data                   ; директива определения данных
      digit dd 6              ; запис у 32-розрядну комірку з ім'ям digit числа 6
      titl db " Ускоренное умножение ",0 ; назва вікна
      st2 dw ?,0 ; буфер виведення повідомлення
ifmt db "Исходное число = %d", 0dh,0ah,
 "%d x %d", 0dh,0ah,
  "Результат умножения = %d", 0dh,0ah,0ah, ; завдання перетворення символу
   "Автор программы: ",0; текстове повідомлення
.code                   ; директива начала программы
_start:                 ; метка начала программы с именем _start
        mov eax,digit       ; занесення в eax числа з комірки пам'яті з ім'ям digit
      mMulD [digit]                   ; виклик макросу
fn  wsprintf, ADDR st2,\ ; адреса буфера, куди буде записана послідовність
  ADDR ifmt,\; адреса рядка перетворення формату
   digit,digit,digit,eax ; комірка пам'яті та регістри, вміст яких перетворюється
fn MessageBox, 0, addr st2, \  ; адреса рядка, яка містить  текст повідомлення
   addr titl, \          ; адреса рядка, яка містить  заголовок повідомлення
    MB_ICONWARNING   ; вигляд діалогового вікна – одна кнопка OK
invoke ExitProcess, 0  ; повернення управління  ОС  та вивільнення ресурсів
      end _start        ; директива завершение программы с именем _start
```

Результат виконання програми з лістингу 14.2 наведено на рис. 14.2.

Рис. 14.2. Результат виконання програми з лістингу 14.2

Наведена програма може бути змінена, щоб збільшити кількість чисел степеня двійки, для множення яких виконуються зсуви.

Приклад 14.3. Написати програму на асемблері з використанням макросу обчислення виразу

$$x/4a + x/a.$$

Приклад обчислення виразу $x/4a + x/a$ наведено в лістингу 14.3. Програма виконує введення дійсних чисел за допомогою функції StdIn та виведення дійсних чисел на консоль за допомогою функції StdOut. Функція StrToFloat

(masm32\include\masm32.inc) перетворює введене дійсне число в 64-розрядне число, а функція FloatToStr виконує зворотне перетворення для виведення результату на екран.

Лістинг 14.3. Програма обчислення прикладу 14.3 з використанням макросів та змінних розміром у байт:

```
        title Rysovaniy A. N., RYSOV@RAMBLER.RU, x/4a + x/a; макроси; дійсні числа
        .686                    ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти
        option casemap:none         ; отличие строчных и прописных букв
        include \masm32\include\windows.inc  ; файлы структур, констант …
        include \masm32\macros\macros.asm
      uselib kernel32, fpu
        mDiv macro x,a          ; макрос з ім'ям mSub та параметрами a та b
          fld x                             ;; завантаження x
          fld a                             ;; завантаження a
          fdiv                              ;; x/a
        endm                                ;; закінчення макросу
.data                                 ; директива определения данных
    stdout dd ?                     ; комірка для збереження дескриптора виведення
    msg1 db "Input digit a: "       ;
    msg2 db "Input digit b: "       ;
    msg3 db "Result = "                     ;
    crlf    db 0dh,0ah                      ;
    msg4 db " Press Enter "     ;
    x dq ?              ; резервування у 80-розрядній комірці місця для змінної x
    a dq ?              ; резервування у 80-розрядній комірці місця для змінної x
    resM dq ?                           ; комірка для тимчасового результату resM
    const1 dd 4.0                   ; комірка для збереження коефіцієнта
    resM2 dt ?                          ; комірка для тимчасового результату resM2
    st2 db 30 dup(?),0
.code                                                   ; директива начала кода программы
_st:                                                    ; метка начала программы
invoke GetStdHandle,STD_OUTPUT_HANDLE
mov stdout,eax                  ; збереження дескриптора виведення у пам'яті
invoke WriteConsoleA,                           ; виведення повідомлення msg1
  stdout, ADDR msg1,  ; дескриптор пристрою виведення та адр. повідомлення
    15,   NULL,NULL         ; розмір повідомлення (кількість символів в msg1)
finit                                                   ; ініціювання FPU
invoke StdIn,                       ; отримує текст від консолі і розміщує в буфері
    ADDR st2, 20      ; введення в буфер st2 з консолі та довжина буфера st2
                                                ;  функція StdIn закінчується після натискання Enter
invoke StrToFloat,   ; перетворення рядка з плаваючою точкою в 64-розр. дані
 ADDR st2,                  ; адреса буфера для числа з плаваючої точкою (в байтах)
   ADDR x                           ; адреса числа QWORD для перетворення
invoke WriteConsoleA,stdout,ADDR msg2,15,NULL,NULL   ; виведення msg2
invoke StdIn, ADDR st2, 20  ; введення в буфер st2 та довжина буфера st2
                                                ;  функція StdIn закінчується після натискання Enter
 invoke StrToFloat,                 ; перетворення рядка з ПТ в 64-розрядні дані
   ADDR st2,                        ; адреса буфера для числа з ПТ (в байтах)
     ADDR a                             ; адреса числа QWORD для перетворення
    mDiv  [x],[a]                           ; виклик макросу
    fld1                        ; завантаження одиниці
    fdiv  dword ptr const1      ; 1/4
    fstp resM2                  ; збереження у пам'яті
invoke FpuMul, 0,    ; операнд SRC1 в FPU
  addr resM2,                   ; операнд SRC2 в пам'яті
    0,                          ; результат DEST в FPU
      SRC1_FPU or SRC2_REAL or DEST_FPU ; атрибути операндів
```

```
   mDiv  [x],[a]                          ; виклик макросу
   fadd                                   ; x/4a + x/a
   fstp qword ptr resM                    ; resM := x/4a + x/a
invoke FloatToStr, resM, addr st2              ; перетворення числа в рядок
invoke WriteConsoleA,stdout,ADDR msg3,9,NULL,NULL ; виведення msg3
invoke StdOut,ADDR st2                    ; виведення на екран
invoke WriteConsoleA,stdout,ADDR crlf,2,0,0 ; переведення курсора на новий рядок екрана
invoke WriteConsoleA,stdout,ADDR msg4,13,NULL,NULL
invoke StdIn, ADDR st2, 10    ; почекати, поки не натиснете ENTER
  invoke ExitProcess,0              ; повернення управління  ОС Windows
  end _st                                ; директива окончания программы
```

Результат виконання програми з лістингу 14.3 наведено на рис. 14.3.

У програмі виводяться запити на введення чисел з консолі та виводиться результат виконання програми.

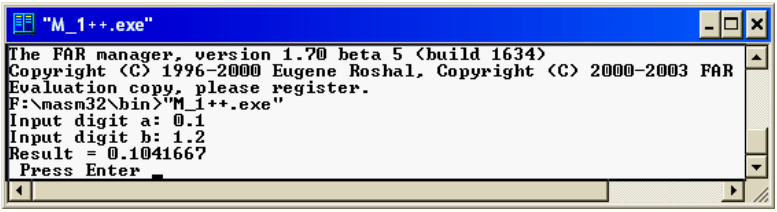


Рис. 14.3. Результат виконання програми з лістингу 14.3

Приклад 14.4. Скласти програму для обміну місцями даних за допомогою макросів.

У лістингу 14.4 наведено приклад без використання оператора заміни.

Лістинг 14.4. Програма виконання прикладу 14.4:

```
        title CopyRight by Rysovaniy A. N.
        .686                    ; директива определения типа микропроцессора
        .model  flat, stdcall   ; завдання лінійної моделі пам'яті  та угоди ОС Windows
        option casemap:none    ; отличие строчных и прописных букв
        includelib\masm32\lib\kernel32.lib
        include\masm32\include\kernel32.inc ; файли систем. функцій застосувань…
        mExchange  macro a, b              ; макрос з ім'ям mExchange
                push eax      ; запам'ятовування у стеку початкового значення
                push ebx      ; запам'ятовування у стеку початкового значення
                mov eax, a                ; занесення до регістра eax операнда a
                mov ebx, b                ; занесення до регістра ebx операнда b
                mov a, ebx                ; обміну місцями даних
                mov b, eax                ; обміну місцями даних
                pop ebx                   ; відновлення з стеку початкового значення
                pop eax                        відновлення з стеку початкового значення
                endm                ;; закінчення макросу
        .data                                   ; директива определения данных
op1 dd 1      ; записати в 32-розрядну комірку пам'яті з іменем op1 число 1
op2 dd 2      ; записати в 32-розрядну комірку пам'яті з іменем op2 число 2
.code                                   ; директива начала программы
_start:                                  ; метка начала программы с именем _start
        mExchange [op1], [op2]      ; виклик макросу
invoke ExitProcess, 0 ; повернення управління ОС та вивільнення ресурсів
        end _start        ; директива завершение программы с именем _start
```

У лістингу 14.4 виконується обмін даними між комірками пам'яті a та b через регістри EAX та EBX.

Стан програми з лістингу 12.4 після її виконання наведено на рис. 14.4.

Рис. 14.4. Стан програми з лістингу 14.4 після її виконання

14.5. Оператори в макровизначеннях

У макровизначеннях можуть використовуватися такі оператори:

& – оператор заміни;

<> – оператор виділення тексту;

! – оператор виділення символу;

% – оператор виразу;

;; – коментар макровизначення.

14.5.1. Оператор заміни (&)

Оператор (макрооператор) заміни (&) дозволяє замінювати параметр значенням, яке передається як аргумент. Синтаксис оператора заміни такий:

&параметр

Часто доводиться в тілі макровизначення змінювати частину формального параметра. В цьому випадку послідовність символів формального аргументу відокремлюють з двох сторін від решти контексту за допомогою спеціального символу **&**. Цей прийом часто використовується для завдання ідентифікаторів, що модифікуються, і кодів операцій.

Приклад макрофункції:

```
moveXx macro dest,src
  mov e&dest&x,e&src&x
 ENDM
.code
_start:
echo Hello!!!     ; виведення повідомлення при компіляції
 moveXx a,b      ; ebx,,ecx
 invoke  ExitProcess, 0 ;
end _start
```

У лістингу 14.5 наведено приклад програми з використанням операторів заміни в макросах. Параметр, який може бути замінений, виділяється з двох сторін символом &.

Лістинг 14.5:

```
        title CopyRight by Rysovaniy A. N.       RYSOV@RAMBLER.RU
        .686                        ; директива определения типа микропроцессора
       .model  flat, stdcall ; завдання лінійної моделі пам'яті  та угоди ОС Windows
        option casemap:none    ; отличие строчных и прописных букв
includelib\masm32\lib\kernel32.lib
include\masm32\include\kernel32.inc ; файли систем. функцій застосувань…
         mExchange  macro a, b, t                    ;; макрос з ім'ям mExchange
            push eax
            push ebx
            mov &t&ax,a
            mov &t&bx,b
            mov a,&t&bx
            mov b,&t&ax
            pop ebx
            pop eax
          endm                                  ;; закінчення макросу
.data                                       ; директива определения данных
op1d dd 0Bh ; записати в 32-розрядну комірку пам'яті з іменем op1d число 11
     op2d   dd 16h                ; 16h = 22
     op1w   dw 21h                ; 21h = 33
     op2w   dw 2Ch                ; 2Ch = 44
.code                                                   ; директива начала программы
_start:                                          ; метка начала программы с именем _start
  mExchange [op1w], [op2w]              ; виклик макросу з двома параметрами
  mExchange [op1d], [op2d],e            ; виклик макросу з 3 параметрами
invoke ExitProcess, 0 ; повернення управління  ОС та визволення ресурсів
        end _start      ; директива окончания программы з іменем _start
```

Стан програми з лістингу 14.5 після її виконання наведено на рис. 14.5.

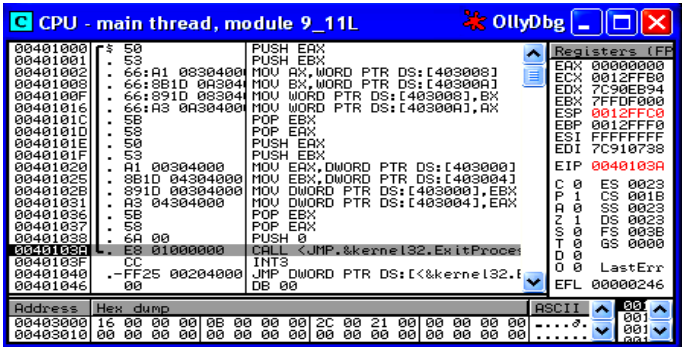


Рис. 14.5. Стан програми з лістингу 14.5 після її виконання

При першому виклику макросу виконується обмін словами, а при другому – подвійними словами, причому замість параметра t виконується підставлення символу e. Це приводить не до виклику регістра AX, а вже до виклику регістра EAX.

Цей оператор також використовується, якщо як аргумент використовується текстове повідомлення і цей аргумент передається рядку або команді. За допомогою оператора заміни можна вирішувати неоднозначні посилання на імена параметрів в макровизначенні.

14.5.2. Оператор виразу **(%)**

Макрооператор % (відсоток) указує, що текст, що знаходиться за ним, є виразом і повинен бути обчислений. Зазвичай це потрібно для того, щоб передавати як параметр в макрос не сам вираз, а його результат. Оператор виразу **(%)** дозволяє розрахувати результат виразу і передати його як аргумент в макровизначення або перетворити константний вираз в його текстове зображення. Оператор указує асемблера на те, що необхідно передати результат розрахунку виразу, а не сам вираз. При сумісному використанні оператора виразу з директивою TEXTEQU виконується обчислення виразу і перетворення його в цілочислове значення.

У наступному прикладі обчислюється вираз (4+count) і повертається значення 14 як символьний текст:

```
count  =  10
sumV TEXTEQU %(4 + count)   ; = "14"
```

Якщо в макровизначенні потрібен цілочисловий аргумент, то оператор виразу % надає різні можливості передачі цілих чисел. Обчислюється вираз, після чого цілочисловий результат передається в макровизначення.

Коли оператор виразу (%) – перший символ в рядку, то препроцесор повинен розвернути всі тексти макровизначення і знайти рядок.

14.5.3. Оператор виділення тексту **(<>)**

Оператор виділення тексту (<>) об'єднує разом рядки символів. Це запобігає інтерпретації асемблером списку як окремих аргументів. Оператор виділень тексту необхідний в директивах умови, таких, як IFB і IFIDN. Він також використовується, коли рядок включає спеціальні символи (",", "%", "&" та ";"), які самі по собі можуть бути інтерпретовані як макрооператори. У наступному прикладі написано макровизначення з ім'ям mMessage:

```
mMessage MACRO text
    DB "&text",0
ENDM
```

Наприклад, макровизначення можна викликати так:

```
mMessage <A1  50%  &A2;>
```

і асемблер запише:

```
DB "A1 50% &A2;",0
```

14.5.4. Оператор виділення символу (!)

Оператор виділення символу ! (знак оклику) використовується аналогічно кутовим дужкам, але діє тільки на один наступний символ, так що, якщо цей символ – кома або кутова дужка, він все одно буде переданий макросу як частина параметра.

14.5.5. Оператор макрокоментаря (;;)

Макрооператор ;; (дві крапки з комою) – початок макрокоментаря. На відміну від звичайних коментарів текст макрокоментаря не потрапляє в лістинг і в текст програми при підстановці макросу. Це заощадить пам'ять при асемблюванні програми з великою кількістю макроозначень.

14.6. Додаткові макровизначення і директиви

14.6.1. Директива REPT

Одну або декілька команд можна повторити з використанням директив REPT, IRP і IRPC. Це дозволяє за допомогою простого макровизначення створити велику структуру даних. Директива REPT повторює блок команд відповідно до лічильника. Синтаксис цієї директиви такий:

```
REPT вираз
   блок команд
ENDM
```

Вираз визначає кількість повторень і задається 16-розрядним числом без знака. Припустимо, необхідно використовувати REPT для розподілу таблиці, що включає 100 записів про співробітників:

```
index LABEL byte                ; початок таблиці
REPT 100                        ; початок циклу REPT
     DB 7 DUP (?)               ; номери співробітників
     DB 20 DUP (' ')            ; прізвище
     DW ?                       ; відповідність
ENDM
```

Цю техніку можна використовувати для створення макровизначення, яке зсуває операнд вліво на задане число байтів. У наступному прикладі лічильник count визначає кількість команд SHL, що були згенеровані асемблером:

```
mSHL MACRO dest,count
    REPT count
      SHL dest,1
    ENDM
ENDM
```

В наступному прикладі директива (псевдокоманда) DB буде повторюватися 155 разів:

```
hexnumber = 0
hextable  label   byte     ; ім'я масиву
          rept  256        ; початок блоку
          db    hexnumber  ; ці два рядки асемблюються
```

```
hexnumber = hexnumber+1          ; 256 разів.
        endm
```

Макровизначення можна викликати так:

```
mSHL AX,1
mSHL  BX,4
```

Код, що при цьому був згенерований асемблером, буде такий:

```
SHL AX,1
SHL BX,1
SHL BX,1
SHL BX,1
SHL BX,1
```

14.6.2. Директива IRP

Директива IRP створює блок, що повторюється, де кожне повторення включає різні значення. Синтаксис цієї директиви такий:

```
IRP аргумент,< аргумент [, аргумент] ...>
    команди
ENDM
```

Блок повторюється тільки для кожного аргументу. При кожному повторенні черговий аргумент передається як параметр. Ця директива корисна при ініціалізації таблиць або блоків даних, де містяться різні значення. Аргумент може бути символічним ім'ям, рядком або числовою константою. Використовуючи наступне написання макровизначення, проаналізуємо розподіл даних, що були згенеровані асемблером:

```
    IRP parm,<10,20,30,40>
        DW parm, parm * 2, parm * 3, parm * 4
    ENDM
Згенерувало.
 DW 10,  10 * 2,   10 * 3,     10 * 4
 DW 20,  20 * 2,   20 * 3,     20 * 4
 DW 30,  30 * 2,   30 * 3,     30 * 4
 DW 40,  40 * 2,   40 * 3,     40 * 4
```

За допомогою директиви IRP можна створити таблицю зсувів процедур, що використовується для вибору декількох гілок за індексом:

```
MOV BX,indexvalue                ; вибір розділу таблиці
CALL proctable[BX]               ; непрямий виклик
```

Наприклад, наступний блок повторень збереже в стеці регістри eax, ebx, ecx та edx:

```
  irp     reg,<eax,ebx,ecx,edx>
          push    reg
          endm
```

У наступному фрагменті з використанням директиви IRP як аргументи передаються чотири імені процедур. Кожне ім'я послідовно передається в параметр procname, внаслідок чого виходить таблиця зсувів процедур:

```
proctable LABEL word
    IRP procname,<MOVup,MOVdn,MOV1ft,MOVrt>
    DW procname
ENDM
```

Асемблер згенерує такі команди:

```
proctable LABEL word
DW MOVup
DW MOVdn
DW MOVlft
DW MOVrt
```

Поєднання зсувів і повторень

Раніше використовувалася директива REPT для створення макровизначення mSHL, яке зрушувало операнд вліво на задане число бітів. Для того щоб такі ж можливості створити для всіх інших команд зсуву і циклічного зсуву, необхідно написати вісім різних макровизначень. Проте є набагато кращий шлях: можна скомбінувати директиву IRP з написанням макровизначення, щоб одержати різні варіанти. Наступне макровизначення генерує вісім різних макроозначень з іменами mSHL, mSHR, mSAL, mSAR, mROL, mROR, mRCL та mRCR:

```
IRP Styp,<SHL,SHR,SAL,SAR,ROL,ROR,RCL,RCR>
 m&styp MACRO dest,count
 PUSH CX
 MOV CL,count
 styp dest,CL
 POP CX
 ENDM
 ENDM
```

Параметр dest може бути 8-, 16- або 32-розрядним регістром або змінною, окрім регістрів CL, CX або ECX. Параметр count може бути безпосереднім значенням, регістром або операндом пам'яті розміром 8 біт.

Оператор заміни (&) дозволяє замінювати рядок аргументів при виклику макровизначення. У наведеному прикладі директива IRP проведе виклик макровизначення вісім разів, використовуючи різні значення змінної styp при кожному виклику.

14.6.3. Директива IRPC

Директива IRPC подібна до директиви IRP, за винятком того, що число символів в аргументі "рядок" визначає число повторень. Синтаксис цієї директиви такий:

```
IRPC параметр, рядок
    команди
```

```
ENDM
```

Рядок може бути поміщений в кутові дужки (< >), якщо він містить пропуски, коми або інші спеціальні символи. У наступному прикладі створюється п'ять змінних (value_A, value_B й т. д.), а як аргумент використовується рядок ABCDE:

```
IRPC parm,ABCDE
value_&parm DB 'parm' ENDM
```

Команди, що були згенеровані асемблером:

```
value_A     DB     'A'
value_B     DB     'B'
value_C     DB     'C'
value_D     DB     'D'
value_E     DB     'E'
```

Макровизначення надають величезні можливості для розширення системи команд. Коли збереться достатньо велика колекція макровизначень, доводиться кожного разу копіювати їх в створювану програму. Натомість створіть окремий файл з макровизначеннями і використайте директиву INCLUDE для підключення його при асемблюванні. Тепер необхідні макровизначення стануть частиною остаточної програми. Якщо використовується двопрохідний асемблер, такий, як Microsoft Assembler, то можна обмежити INCLUDE директивами IF1 і ENDIF, які примусять асемблер включати макровизначення тільки при першому проході:

```
IF1             ; якщо це перший прохід
    include  macros.inc       ; підключити файл макровизначень
ENDIF
```

Якщо не вказаний повний шлях файлу, що підключається, асемблер шукатиме його в поточному каталозі. Можна додатково вказати повний або відносний шлях.

14.7. Макроси циклів: FOR, FORC, REPEAT, WHILE

Директива **FOR** повторює цикл стільки раз, скільки параметрів міститься у вказаному текстовому рядку.

Кожен елемент списку задає одну ітерацію циклу. Синтаксис директиви FOR наступний:

```
FOR Параметр, <арг1, арг2, арг3, . . . >
            Блок-операторів
ENDM
```

Під час першої ітерації циклу параметру привласнюється значення елемента арг1; під час другої ітерації – елемента арг2 і так далі до вичерпання елементів списку.

Можна задати також значення за умовчанням, або модифікатор req.

У Masm 5 замість директиви FOR застосовувалася директива IRP.

Приклад:

```
mZeroRegs macro reglist:vararg ;
  FOR param:=<eax>,<reglist>
    xor param,param
  ENDM                    ;; закінчення директиви циклів FOR
 ENDM                     ;; закінчення макросу з іменем mZeroRegs
.data
.code
 _start:
    echo Hello!!!          ; виведення повідомлення при компіляції
    mZeroRegs ebx,,ecx ; відкомпілюється в xor ebx,ebx ; xor eax,eax ;  xor ecx,ecx
```

Приклад заповнення нулями значення регістрів, вказаних в кутових дужках:

```
mZeroRegs macro reglist:vararg
   FOR curreg,<eax,ebx,ecx,edx,esi,edi>
   xor curreg,curreg
  ENDM                     ;; закінчення директиви циклів FOR
 ENDM                      ;; закінчення макросу з іменем mZeroRegs
.data                      ; директива определения данных
…
echo Hello!!!    ; виведення повідомлення при компіляції
 mZeroRegs ;
```

Директива **EXITM** використовується для передчасного переривання циклу або макросу використовується.

Директива **FORC** parametr, < рядок > організовує цикл, що повторюється для кожного символу наведеного рядка.

Синтаксис директивы FORC следующий:

```
FORC  Параметр,   <рядок>
        Блок операторів
 ENDM
```

Під час першої ітерації циклу параметру привласнюється значення першого символу рядка; під час другої ітерації – другого символу і так далі до вичерпання символів в рядку, наприклад:

```
Delim  LABEL    BYTE
FORC   code, <@#$%"&* !<!»
   BYTE   "&code"
ENDM
```

Директива **REPEAT** вираз – віконується задана кількість разів. Її синтаксис наступний:

```
REPEAT  Константний вираз
        Блок операторів
 ENDM
```

У Masm 5 замість директиви REPEAT застосовувалася директива REP.

Директива **WHILE** выражение віконується поки істинне (не нульове) значення виразу. Її синтаксис наступний:

WHILE *Константний вираз*
Блок операторів
ENDM

Кінець блоку позначається директивою ENDM.

14.8. Категорії макросів в masm32

В masm32 є велика кількість макросів, які зведені в категорії.

Макроси запитів:
FUNC – отримання значення, що повертається, від процедури
fn – горизонтальний виклик процедури стилю;
rv – отримання значення, що повертається, від процедури;
fnc – горизонтальний виклик процедури стилю;
rvc – отримання значення, що повертається, від процедури стилю.

Макрос помилок повідомлень:
LastError$ – повернення помилкового рядка з GetLastError().

Макроси файлового введення-виведення:
InputFile – читання всього файлу в буфері пам'яті;
OutputFile – запис буфера пам'яті у диск;
fcreate – створення нового файлу та повернення його хендлу;
fdelete – видалення дискового файлу;
fopen – відкриття існуючого файлу та повернення його хендлу;
fclose – закриття відкритого хендлу файлу;
fread – читання даних з відкритого файлу;
fwrite – запис даних до відкритого файлу;
fprint – запис тексту до відкритого файлу;
fprintc – запис тексту для відкриття файлу з форматуванням стилю;
fseek – встановлення поточної координати у відкритому файлі;
fseteof – встановлення кінця відкритого файлу до поточного розміщення;
fsize – повернення розміру в байти відкритого файлу.

Макроси розміщення пам'яті:
alloc – розміщення виправленого блоку пам'яті;
free – вільний виправлений блок пам'яті;
alloc$** – звільнити пам'ять рядка, використовуючи макрокоманду **free$;
free$ – звільнити пам'ять, використовуючи **GlobalAlloc();**
halloc – розміщення пам'яті купи;
hsize – повернення розміру пам'яті купи;

hfree – вільна пам'ять купи.

Консольні макроси:

cls – очищення екрана консолі;

ccout – показати текст в консолі, використовуючи форматування стилю;

GetConsoleCaption$ – перевірка адреси заголовка вікна консолі;

exit – закінчення програми;

getkey – чекання ключового друку та повернення його в EAX;

inkey – чекання ключа, призначеного для користувача;

input – отримання тексту від входу консолі;

loc – встановлення позиції текстового курсора;

print – друк тексту на консоль;

SetConsoleCaption – встановлення тексту заголовка вікна поточної консолі.

Макроси рядкових команд:

add$ – конкатенація рядків;

append$ – приєднання нуля на закінчений рядок та повернення наступної позиції;

cat$ – конкатенація рядків кратного числа;

chr$ – повернення адреси рядка та байтів;

Приклад:

```
invoke MessageBox, 0, chr$("Прог. работает"), chr$("Титул сообщения"), 0
```

Приклад:

```
buffer dd ?,0
…
invoke wsprintf, offset buffer, chr$("EAX = %u, ECX = %u"), eax, ecx
                    ; запис рядка EAX = [число], ECX = [число] до буфера  buffer
invoke MessageBox, 0, offset buffer, chr$(" Титул сообщения "), 0
                                ; виведення назв регістрів з перетвореними числами
```

cmd$ – повернення адреси командного рядка;

ptr$ – приведення буфера до вказівника;

len – повернення довжини рядка з нулем на кінці;

istring – знайдення підрядка в початковому рядку;

lcase$ – перетворення рядка на нижній регістр;

ucase$ – перетворення рядка на верхній регістр;

left$ – читання символу із лівої сторони рядка;

right$ – читання символу з лицьової сторони рядка;

ltrim$ – упорядкування лівої сторони рядка з інтервалами;

rtrim$ – упорядкування лицьової сторони рядка інтервалів;

trim$ – упорядкування обох кінців рядка інтервалів;

remove$ – видалення рядка з початкового рядка;

rev$ – зміна рядка;

cfm$ – форматування рядка зі стилем;

hval – перетворення шістнадцяткового рядка на 32-розрядні цілі числа;
sval – перетворення підписаного рядка на 32-розрядні цілі числа;
uval – перетворення непідписаного рядка на 32-розрядні цілі числа;
ustr$ – перетворення без знака 32-розрядного цілого числа до закінчення рядка;
sstr$ – перетворення підписаного 32 розрядного цілого числа до закінчення рядка;
uhex$ – перетворення без знака 32-розрядного цілого числа до закінчення шістнадцяткового рядка.

Макроси динамічних рядкових масивів:
arralloc$ – створення порожнього масиву із змінними межами;
arrealloc$ – перерозподілення існуючого масиву до більшого або меншого члена;
arrfree$ – звільнення масиву та повернення використаної пам'яті;
arrset$ – перезапис даних вказаної довжини з нулями на кінці;
arrbin$ – перезапис даних вказаної довжини;
arrfile$ – завантаження текстового файлу в новому масиві;
arrtxt$ – завантаження тексту з нулем на кінці;
arrget$ – отримання стартової адреси члена масиву;
arr2disk$ – запис текстових даних з масиву до дискового файлу;
arr2mem$ – запис двійкових даних з масиву до заздалегідь розміщеного буфера пам'яті;
arr2text$ – запис текстових даних з масиву до заздалегідь розміщеного буфера пам'яті;
arrlen$ – отримання довжини члена масиву;
arrcnt$ – отримання індексу членів масиву;
arrtotal$ – обчислення повного розміру буфера пам'яті для масиву;
arrtrunc$ – зменшення розміру масиву;
arrextnd$ – розширення масиву.

Макроси діалогових файлів:
OpenFileDlg – відкриття файлу діалогів та повернення імені файлу;
SaveFileDlg – зберігання файлу діалогу та повернення імені файлу.

Макроси шляхів та директорій:
pth$ – повернення шляху поточного застосування;
CurDir$ – повернення шляху поточного каталогу;
SysDir$ – повернення шляху системної директорії;
WinDir$ – повернення шляху директорія Windows;
chdir – зміна директорії;
mkdir – створення директорії;
rmdir – переміщення директорії;
rndir – перейменування директорії.

Макрос текстових панелей інструментів:
TextItem – текстовий елемент.

Робочі макроси:
memalign – вирівнювання пам'яті до кратності 2;
chtype$ – розпізнавання символів великої швидкості.

Макроси розподілення пам'яті:
linein$ – визначення кількості позицій в тексті (читання тексту в буфер):

```
        .data
        ifmt db "%d", 0
        _textinfo db "Номер следующей позиции после текста: "
        res1 db 30 dup (0), 0
        spos    dd  0
        _title  db "Результат", 0
        source  db 'Линия 1,',13,10,'Линия 2,',13,10,
                'Линия 3',13,10, 0
        buffer db 100 dup(?), 0
    .code
    start:
            invoke MessageBox,0, addr source, addr _title, MB_YESNO
        m1:
            mov spos, linein$ (addr source,addr buffer, spos)
            .if(eax == 0)
                jmp m2
            .endif
            mov  spos,eax
            push spos
            invoke wsprintf, addr res1, addr ifmt, spos
            invoke MessageBox,0, addr buffer, addr _title, MB_OK
            invoke MessageBox,0, addr _textinfo, addr _title, MB_OK
           pop eax
           or  eax,eax
           jz  m2
           jmp m1
        m2:
            invoke  ExitProcess,0
              END start
```

lineout$ – запис тексту від користувача в буфер;
tstline$ – перевірка на наявність тексту;
CountLines – процедура перерахування текстового файлу.

Макроси оточення:
setenv – встановлення рядка оточення;
env$ – отримання рядка оточення.

Макроси бібліотек:
reparg – заміна рядка з даними;
tstarg – випробування для цитованого тексту та повернення помилки, якщо вона є;

argcount – повернення індексу аргументів, що передаються макрокоманді;
getarg – повернення аргументу із списку його індексом;
op_type – повернення типу операнда макроаргументу;
cfm$ – форматувати рядок, використовуючи стиль.

Макроси перетворення:
ubyte$() – беззнакове BYTE-введення;
sbyte$() – знакове BYTE-введення;
xbyte$() – шістнадцяткове знакове BYTE-виведення;
uword$() – беззнакове WORD-введення;
sword$() – знакове WORD-введення;
xword$() – шістнадцяткове знакове WORD-виведення;
udword$() – беззнакове DWORD-введення;
sdword$() – знакове DWORD-введення;
xdword$() – шістнадцяткове знакове DWORD-виведення;
uqword$() – беззнакове QWORD-введення;
sqword$() – знакове QWORD-введення;
xqword$() – шістнадцяткове знакове QWORD-виведення;
real4$() – 32-розрядне введення з плаваючою точкою;
real8$() – 64-розрядне введення з плаваючою точкою;
real10$() – 80-розрядне введення з плаваючою точкою.

Макроси спеціалізованих прототипів:
SPROTO – створіть прототип STDCALL;
CPROTO – створіть прототип C.

Макроси псевдомнемонік:
m2m – копіювання змінної з пам’яті в пам’ять;
mrm – копіювання змінної з пам’яті в пам’ять з використанням EAX;
ass – призначення рядка змінної LOCAL;
cst – копіювання одного рядка до іншого з нулем на кінці.

14.9. Макрос @ для запису команд в один рядок

Макрос @ можна застосувати для скорочення числа рядків в програмі. Цей метод актуальний тоді, коли, наприклад, необхідно виконати блок команд, що має загальне призначення. В цьому випадку доцільно написати коментар на весь блок і привести всі команди в однин рядок. Наприклад, макрос з ім'ям @m (можна просто @)

```
@m MACRO c0,c1,c2,c3,c4,c5,c6
c0
 c1
  c2
   c3
    c4
```

```
        c5
        c6
    ENDM
```
дозволяє написати до 7 команд в одну строчку. Тоді послідовність вигляду
```
    mov al,a
    mov bl,res1
    mul bl
    mov bx,ax
```
замінюється рядком
```
@m<mov al,a>,<mov bl,res1>,<mul bl>,<mov bx,ax>
```

Приклад виведення **системного часу** за допомогою макросів наведено в лістингу 14.6.

Лістинг 14.6. Виведення **системного часу** за допомогою макросів:
```
    .686                    ; директива определения типа микропроцессора
    .model flat,stdcall ; задание линейной модели памяти
    option casemap:none     ; отличие строчных и прописных букв
    include \masm32\include\windows.inc   ; файлы структур, констант …
    include \masm32\macros\macros.asm
    uselib kernel32, user32
        mLL macro
          mov [esi],eax
          add esi,4
         endm      ;; закінчення макросу
    @m macro a0,a1,a2 ; макрос розміщення команд одним рядком
     a0
      a1
       a2
       endm
.data
    systime  SYSTEMTIME   <?>      ; системний час
        buf dd 6 dup (?),0
        buf2 dd 2 dup (?),0
fmt db "Год = %d",0ah, "Месяц = %d",0ah, "День недели = %d",0ah,
"Число=  %d",0ah,"Часы = %d",0ah, "Минуты = %d",0ah,
"Секунды = %d",0ah, "Миллисекунды = %d",0ah,0ah,
  "Автор: ",0  ; див. c:\masm32\datetime
     fmt2 db "Текущее время -- %d",":%d",0
        mes db "Часы",0
        mes2 db "Текущее время",0
        mas dd 6 dup (?)
.code
  start:
    lea esi,mas ;
invoke GetLocalTime, ADDR systime
    @m<movzx eax,systime.wYear>,<mLL>
    @m<movzx eax,systime.wMonth>,<mLL>
    @m<movzx eax,systime.wDayOfWeek>,<mLL>
    @m<movzx eax,systime.wDay>,<mLL>
    @m<movzx eax,systime.wHour>,<mLL>
    @m<movzx eax,systime.wMinute>,<mLL>
    @m<movzx eax,systime.wSecond>,<mLL>
    @m<movzx eax,systime.wMilliseconds>,<mLL>
invoke wsprintf,addr buf,addr fmt,mas,mas[4],mas[8],mas[12],mas[16], mas[20], mas[24],
mas[28]
invoke MessageBox,0,addr buf,addr mes,MB_ICONASTERISK or MB_OKCANCEL
```

```
invoke wsprintf,addr buf2,addr fmt2,mas[16],mas[20]
invoke MessageBox,NULL,addr buf2,addr mes2,MB_OK
        invoke ExitProcess, 0
        end start
```

Результат виведення системного часу наведено на рис. 14.6.

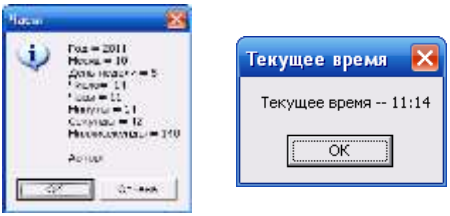


Рис. 14.6. Результат виведення системного часу

14.10. Лабораторна робота “Макроси”

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм для програмування макросів, написаних мовою асемблера для МП платформи x86 з організацією виведення результату на екран з використанням API-функцій під Win32.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати свій варіант та написати на асемблері програму обчислення одного з виразів з використанням макросів та виведенням їх на екран. При цьому обов’язково використати ***спрощені віконця*** з повідомленнями: в одному віконці вивести своє прізвище, номер своєї навчальної групи та своїх фотографії в якості іконки; при натисканні на другу кнопку спрощеного віконця – вивести умову завдання. Використання інших кнопок спрощеного віконця виведення повідомлень – за бажанням.

Завдання 1

1. $(2a/b) + a/3{,}1b$;
2. $1{,}1(x - a) + 2^{10}/(x - a)$;
3. $x + a - 4{,}5/(x + a)$;
4. $3{,}5(a - b) - (a - b)/5{,}1$;
5. $2x - 3{,}3 + 8{,}5(2x - 3{,}3)$;
6. $(1{,}1ab - 3)/ab$;
7. $2{,}3(a - b) + 2{,}3(a - b)/a$;
8. $5{,}6(ab + d)/[5{,}6(ab - d)]$;
9. $(de - e)/[2{,}2(de - e)]$;
10. $(a/2{,}3b) - (2{,}3a/b)$.

Завдання 2. Самостійно придумати завдання та написати програму з обов'язковим використанням макросів masm32 за варіантом згідно з останньою цифрою номера студента в групі. Використання інших макросів – за бажанням.

1. Макроси запитів.
2. Макроси файлового введення-виведення.
3. Макроси розміщення пам'яті.
4. Консольні макроси.
5. Макроси рядкових команд.
6. Макроси динамічних рядкових масивів.
7. Макроси шляхів та директорій.
8. Макроси розподілення пам'яті.
9. Макроси бібліотек.
10. Макроси перетворення.

Вимоги до програм

Програма повинна містити введення та виведення дійсних чисел на екран монітора з використанням API-функцій.

Зміст звіту

1. Постановка задачі для конкретного варіанта.
2. Блок-схема алгоритму виконання прикладу з детальним коментарем.
3. Лістинг програми з детальним коментарем до кожної команди.
4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання.
5. Короткий опис виконання програми.
6. Висновки за результатами роботи.

Приклад 14.5. Написати програму з використанням макросів для обчислення виразу $a(a-b)-(a-b)$ та слів розміром у байт.

Програму без виведення результату на монітор наведено в лістингу 14.6.

Лістинг 14.6:

```
title CopyRight by Rysovaniy A. N.     RYSOV@RAMBLER.RU
.686                    ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти
option casemap:none         ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib kernel32,user32,masm32;,fpu
mSubB macro _a,_b                ;; макрос з ім'ям mSubB для (a – b)
mov al,_a                 ;; занесення змінної a
sub al,_b                 ;; віднімання a – b
mov res1,al               ;; збереження результату у пам'яті
endm                             ;; закінчення макросу
@ macro c0,c1,c2,c3 ;;макрос для запису команд в рядок
c0
```

```
        c1
         c2
          c3
        endm  ;; закінчення макросу @
.data                                                   ; директива определения данных
  _a  db 6      ; збереження в комірці пам'яті розміром в байт операнда 6
  _b  db 5      ; збереження в комірці пам'яті розміром в байт операнда 5
  res1 db 0                                     ; резервування пам'яті для результату res1
  buf dd ?,0                ; буфер виведення повідомлення
  ifmt  db "    Уравнение: ", 0dh,0ah,"Y = a(a – b) - (a – b)", 0dh,0ah,0ah, ;
    "При a = 6, b = 5, Y = %d",0dh,0ah,0ah, "Автор программы:",0  ;
  titl db "Результат решения уравнения",0 ; назва спрощеного вікна
  fName db "c:\\asm.txt",0
  fHandle dd ?
  number dd 84
  _inf db "Результаты программы записаны в файл",0
.code
_start:
        ; виконання [a x mSubB]
          @<mSubB [_a],[_b]>,<mov al,_a>,<mul res1>,<mov bx,ax>
        ; виконання  [- mSubB]
          @<mSubB [_a],[_b]>,<movzx ax,res1>,<sub bx,ax>,<movzx ebx,bx>
          push ebx
invoke wsprintf, ADDR buf,\ ;  адреса буф. з символами
 ADDR ifmt, ebx ; адреса рядка перетворення формату та регістр з результатом
invoke MessageBox, 0,addr buf,\   ; hwnd та адр. буфера текст повідомлення
  addr titl, MB_ICONINFORMATION ; адреса заголовку вікна та вигляд вікна
         pop ebx
invoke CreateFile, ADDR fName,       ; адреса імені файла з символами
   GENERIC_WRITE,0,0, ; запис у файл,  ; параметри багатозадачності
    CREATE_ALWAYS, FILE_ATTRIBUTE_ARCHIVE, 0
       mov fHandle, eax             ; запам'ятовування дескриптора пристрою
invoke WriteFile, fHandle,   ; запис у файл за дескриптором пристрою
   ADDR buf,        ; адреса області пам'яті, що зберігає символи
    number,                         ; кількість символів
    ADDR number, 0    ; адреса пам'яті, де зберігається число
       invoke  CloseHandle, fHandle   ; закрити дескриптор файла
       invoke MessageBox,0,addr _inf,addr titl,MB_ICONINFORMATION
       invoke ExitProcess, 0 ; повернення управління  ОС та визволення ресурсів
       end  _start         ; директива окончания программы з іменем start
```

У зв'язку з тим, що після перемноження регістрів al та bl результат зберігається у регістрі ax розміром у слово, виникає необхідність у подальшому зберіганні результату розміром 16 розрядів. Через цю причину використовується команда розширення розрядності movzx.

У наведеній програмі на екран виводиться вміст регістра ebx. Причому API-функції потребують, щоб число, яке виводиться, мало розрядність 32. Інакше буде генерована помилка при трансляції. Щоб помилок не виникало, використано команду збільшення розрядності регістра до 32 (movzx ebx,bx).

Для виведення чисел на екран за допомогою API-функцій необхідно, щоб результат був розташований у 32-розрядному регістрі. Тому в програмі використано команду movzx, яка розширює результат до 32-розрядного формату. Результат виконання програми наведено на рис. 14.7.

При налагодженні програми не слід копіювати команди з редактора Word, тому що він вставляє у текст службові символи, деякі з яких є невидимими

потім навіть у блокноті. Слід зразу використовувати блокнот чи інші текстові редактори, призначені для таких цілей.

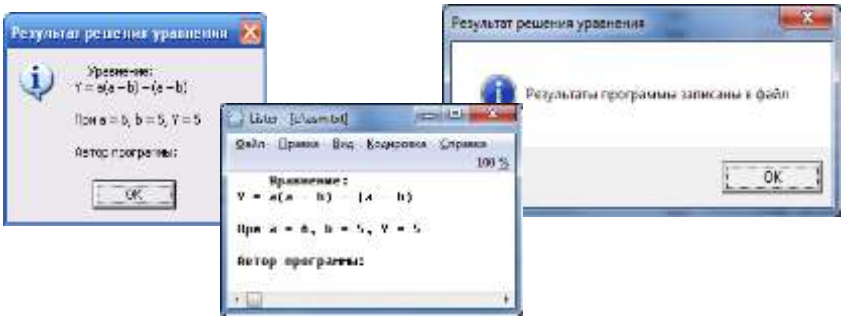


Рис. 14.7. Результат виконання програми з лістингу 14.6

Приклад 14.6. Написати програму з використанням макросів для обчислення виразу

$$1{,}1(x - a) + 210/(x - a).$$

Лістинг 14.7:

```
        .686          ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти
        option casemap:none         ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
      uselib kernel32,user32, fpu
            mSub macro x, a         ; макрос з ім`ям mSub
            fld x                       ; завантаження x
            fld a                       ; завантаження a
            fsub                        ; x-a                       ;; 2*x - 3.3
            endm                        ; закінчення макросу
.data                                   ; директива определения данных
      st1 db "1,1(x – a) + 2^10/(x – a), де x=2.7, a=3.2  ",0
      buf db "  Виконала студентка групи КІТ-29б Вернидуб Анна", 10,13
      st2 db "     Результат:    "
      sum dt 0
      x dd 2.7
      a dd 3.2
      res dt ?
      const1 dd 1.1
      const2 dd 2
.code                                   ; директива начала кода программы
 start:                                 ; метка начала программы
      mov ecx,8
      finit                             ; ініціювання FPU
      fild const2
      fild const2
_1:  fmul st(0),st(1)
     loop _1
      fmulp st(1),st
      fld const1
```

```
fadd
mSub [x], [a]       ; виклик макросу
fmul
fstp res                    ; збереження результату
invoke FpuFLtoA,ADDR res, 2, ADDR sum, SRC1_REAL or SRC2_DIMM
invoke MessageBox, NULL, addr buf, addr st1, MB_ICONINFORMATION
invoke ExitProcess, 0
end start
```

Результат виконання програми з лістингу 14.7 наведено на рис. 14.8.

Рис. 14.8. Результат виконання програми з лістингу 14.8

Контрольні запитання

1. Що називається макровизначенням?
2. Які переваги та недоліки використання макросів?
3. Як прописуються коментарі в макросах та навіщо?
4. Яке макровизначення виводить повідомлення при компіляції?
5. Які дії необхідно виконати, якщо планується використовувати макроси в одній програмі декілька разів?

15. ЛОГІЧНІ КОНСТРУКЦІЇ ВИСОКОГО РІВНЯ

Директиви умовної трансляції часто називають логічними конструкціями високого рівня, які як й **макровизначення**, дозволяють об'єднати декілько логічних команд для виконання частини програмного коду.

Кожен рядок програми на мові високого рівня є закінченою думкою, тоді як асемблерна програма вимагає для реалізації тієї ж думки безлічі рядків. Проте використання конструкцій високого рівня приводить до полегшення процесу програмування і до підвищення надійності програм. Такі конструкції роблять програмування менш складним, що дозволяє програмісту концентруватися на логіці програми.

MASM забезпечує можливість використання логічних конструкцій високого рівня через засоби макро. Макро є "суперкомандами", які розвантажують MASM від тієї частини команд, що часто повторюється.

15.1. Директива IF

У макроасемблері Microsoft (MASM) визначено директиву .IF, яка дозволяє записувати логічні вирази так, як це робиться в мовах високого рівня.

Конструкція 1:

```
.IF умова_1
      вираз
.ENDIF
```

Конструкція 1 означає, що, якщо умова_1 виконується, то вираз обчислюється; якщо умова_1 не виконується, то вираз не обчислюється, про що свідчить директива .ENDIF.

Конструкція 2:

```
.IF умова_1
     вираз_1
[.ELSEIF умова_2
          вираз_2]
...
[.ELSE
          вираз_n]
.ENDIF
```

Конструкція 2 означає, що, якщо умова_1 виконується, то обчислюється вираз_1, а якщо умова_1 не виконується, то відбувається перевірка наступної умови і т.д. Інакше обчислюється вираз_n.

Директива .ELSE визначає альтернативний блок коду у випадку, якщо попередні .IF та .ELSEIF вирази невірні.

Оператор .ENDIF закриває поточний блок коду .IF, .ELSEIF або .ELSE.

Повний список операторів відношень наведено в табл. 15.1.

Таблиця 15.1 – Оператор відношень і логічні оператори

Оператор	Опис
вираз1 == вираз2	Вирази рівні між собою
вираз1 != вираз2	Вирази не рівні між собою
вираз1 > вираз2	Вираз1 більший від виразу2
вираз1 >= вираз2	Вираз1 більший від виразу2 або дорівнює йому
вираз1 < вираз2	Вираз1 менший від виразу2
вираз1 <= вираз2	Вираз1 менший від виразу2 або дорівнює йому
! вираз	Вираз неправдивий
вираз1 && вираз2	Логічне І між виразами
вираз1 || вираз2	Логічне АБО між виразами
вираз1 & вираз2	Порозрядне І між виразами
CARRY?	Прапорець перенесення встановлений ?
OWERFLOW?	Прапорець переповнення встановлений ?
PARITY?	Прапорець парності встановлений ?
ZERO?	Прапорець нуля встановлений ?
SIGN?	Прапорець знака встановлений ?

У наступному прикладі спочатку розглядається умова “якщо eax = = 1”. У випадку, якщо eax ≠ 1, то внутрішні рядки програми не виконуються, а виконується перехід на останню директиву .ENDIF.

При eax = 1 виконується наступний рядок, в якому теж виконується перевірка – чи ecx ≠ 2. При виконанні останньої умови – в ebx заноситься одиниця. Директивою закінчення циклу IF ecx!=2 є внутрішня директива .ENDIF.

```
.IF eax==1              ; якщо eax = 1, то перевірити наступну умову
  .IF ecx!=2            ; а якщо й ecx != 2, то виконати
   mov ebx,1            ;  занесення 1 в ebx
  .ENDIF                        ; закінчення директиви .IF ecx!=2
.ENDIF                          ; закінчення директиви  .IF eax==1
```

Цей фрагмент програми можна зробити простішим:

```
.IF (eax==1 && ecx!=2)          ; якщо eax = 1 та ecx ≠ 2, то виконати
   mov ebx,1                            ;  занесення 1 в ebx
.ENDIF                          ; закінчення циклу директиви .IF
```

Асемблер виконує оптимізацію для отримання, якщо це можливо, ефективнішого коду. Наприклад директива

```
.IF  ax==0
```

буде подана як порівняння

```
or   ax,ax
```

яке набагато ефективніше, ніж

```
cmp  ax, 0
```

Під словами "умова_1" і "умова_n" ховаються булеві вирази, які використовують такі оператори, як <, >, == або !=.

Можна використовувати як одинарну умову, так і **складену**, які можуть мати такий вигляд:

```
(EAX < 10h) && (EAX > l00h)
(op1 <= 100) || (op2 <= 100}
(op1 != EDX) && SIGN?
```

Конструкція вигляду

```
        .IF eax==3
invoke MessageBox, NULL, addr st2, addr st1, MB_ICONINFORMATION
        .ELSEIF eax==4
  invoke MessageBox, NULL, addr st4, addr st3, MB_ICONQUESTION
        .ELSE
   invoke MessageBox, NULL, addr st6, addr st5, MB_ICONERROR
        .ENDIF
```

означає, що, якщо eax = 3, то виводиться повідомлення з параметром MB_ICONINFORMATION, а якщо ні, то перевіряється наступна умова. При eax = 4 виводиться повідомлення з параметром MB_ICONQUESTION. А якщо eax не дорівнював ні 3, ні 4, то виконується дія, яка розташована після директиви .ELSE, тобто виводиться повідомлення з параметром MB_ICONERROR.

Якщо обрано функцію MessageBox з параметром IDYES (з двома кнопками), то зручніше використовувати назви параметрів:

```
.IF eax == IDYES ; якщо вибрано кнопку YES
  …
 .ELSEIF eax== IDNO ; якщо вибрано кнопку NO
  …
 .ELSE  ; інакше
  …
.ENDIF ; закінчення циклу IF
```

Окрім загальних директив IF та ELSEIF асемблери підтримують набір спеціальних команд, кожна з яких перевіряє **спеціальну умову**:

IF1/ELSEIF1 – якщо асемблер виконує перший прохід асемблювання;

IF2/ELSEIF2 – якщо асемблер виконує другий прохід асемблювання (часто не працює на сучасних асемблерах);

IFE вираз /ELSEIFE вираз – якщо вираз дорівнює нулю (помилково);

IFDEF мітка/ELSEIFDEF мітка – якщо мітка визначена;

IFNDEF мітка/ELSEIFNDEF мітка – якщо мітка не визначена;

IFB <аргумент>/ELSEIFB <аргумент> – якщо значення аргументу – пропуск (ці та всі наступні директиви використовуються в макровизначеннях для перевірки параметрів);

IFNB <аргумент>/ELSEIFNB <аргумент> – якщо значення аргументу - не пропуск (використовується в макровизначеннях для перевірки переданих параметрів);

IFDIF <арг1>,<арг2>/ELSEIFDIF <арг1>,<арг2> – якщо аргументи відрізняються (з відмінністю великих і маленьких букв);

IFDIFI <арг1>,<арг2>/ELSEIFDIFI <арг1>,<арг2> – якщо аргументи відрізняються (без відмінності великих і маленьких букв);

IFIDN <арг1>,<арг2>/ELSEIFIDN <арг1>,<арг2> – якщо аргументи однакові (з відмінністю великих і маленьких букв);

IFIDNI <арг1>,<арг2>/ELSEIFIDNI <арг1>,<арг2> – якщо аргументи однакові (без відмінності великих і маленьких букв).

Зустрівши директиву .ERR, асемблер припинить роботу з повідомленням про помилку. Аналогічно командам умовного асемблювання існують модифікації команди .ERR:

.ERR1 – помилка при першому проході асемблювання;

.ERR2 – помилка при другому проході асемблювання;

.ERRE вираз – помилка, якщо вираз дорівнює нулю (помилково);

.ERRNZ вираз – помилка, якщо вираз не дорівнює нулю (істинно);

.ERRDEF мітка – помилка, якщо мітка визначена;

.ERRNDEF мітка – помилка, якщо мітка не визначена;

.ERRB <аргумент> – помилка, якщо аргумент порожній (ця і всі наступні директиви використовуються в макровизначеннях для перевірки параметрів);

.ERRNB <аргумент> – помилка, якщо аргумент не порожній;

.ERRDIF <арг1>,<арг2> – помилка, якщо аргументи різні;

.ERRDIFI <арг1>,<арг2> – помилка, якщо аргументи відрізняються (порівняння не розрізняє великі і маленькі букви);

.ERRIDN <арг1>,<арг2> – помилка, якщо аргументи співпадають;

.ERRIDNI <арг1>,<арг2> – помилка, якщо аргументи співпадають (порівняння не розрізняє великі і маленькі букви).

Приклад 15.1. Порівняти число з трьома операндами. Якщо це число більше від першого операнда, більше від другого або дорівнює та дорівнює третьому, то слід додати всі ці операнди, інакше – на вихід.

У лістингу 15.1 наведено таку програму, яка є послідовністю умовних переходів без використання директиви .IF.

Лістинг 15.1. Програма розв'язання прикладу 15.1 з послідовністю умовних переходів без використання директиви .IF:

```
    .686                    ; директива визначення типу мікропроцесора
    .model  flat,stdcall  ; завдання лінійної моделі пам'яті та угоди ОС Windows
    option casemap:none   ; відмінність малих та великих літер
    includelib\masm32\lib\kernel32.lib
    include \masm32\include\kernel32.inc ; системні функції застосувань…
.data                                   ; директива визначення даних
    op   dd 23          ; 17h  – операнд для порівняння
    op1 dd 22           ; 16h – перший операнд розміром у подвійне слово
    op2 dd 21           ; 15h – другий операнд розміром у подвійне слово
    op3 dd 23           ; 17h – третій операнд розміром у подвійне слово
.code                                   ; директива початку сегмента даних
_start:                                 ; мітка початку програми з ім'ям _start
```

```
        mov eax, op                    ; занесення операнда, з яким порівнюються
        cmp  eax,op1                   ; порівняння з першим операндом
        jg  L1                         ; перехід до L1, якщо AL  >  op1 (зі знаком)
        jmp L2                                 ; безумовний перехід
L1:  cmp eax,op2               ; порівняння з другим операндом
        jge   L3                       ; перехід  до  L1, якщо  AL  >=  op2
        jmp L2                                 ; безумовний перехід
L3:  cmp eax,op3               ; порівняння з третім операндом
        je  L4                         ; перехід  до  L1, якщо  AL  =  op3
        jmp  L2                                ; інакше  перехід  до  L2
L4:  add eax,op1                               ; додавання першого операнда
        adc eax,op2            ; додавання другого операнда з урахуванням CF
        adc eax,op3            ; додавання третього операнда з урахуванням CF
L2:  invoke  ExitProcess, 0    повернення управління  ОС та вивільнення ресурсів
        end _start         ; директива закінчення програми з ім'ям _start
```

Результатом виконання програми з лістингу 2.1 є зміст регістра EAX = 0000 0059h = 23 + 22 + 21 + 23 = 89D.

Для налагодження програм у налагоджувачі OllyDbg, які мають функції, що не потребують перевірки функціонування, доцільніше використовувати у покроковому режимі клавішу F8. Такими функціями можуть бути функції API Win32 або функції, раніше використані та перевірені. У такому випадку використання клавіші F8 різко прискорює процес перевірки програми. Модернізовану програму з лістингу 15.1 з використанням директиви IF наведено в лістингу 15.2.

Лістинг 15.2. Програма розв'язання прикладу 15.1 з використанням директиви IF:

```
        .686                           ; директива определения типа микропроцессора
       .model  flat,stdcall ; завдання лінійної моделі пам'яті та угоди ОС Windows
        option casemap:none          ; отличие строчных и прописных букв
includelib\masm32\lib\kernel32.lib
include \masm32\include\kernel32.inc ; системные функции применений…
   .data                               ; директива определения данных
        op   dd 23                     ; операнд для порівняння
        op1 dd 22
        op2 dd 21
        op3 dd 23
.code
  _start:
        mov eax, op
  .IF (eax > op1) && (eax >= op2) && ( eax == op3)
        add eax, op1 a
        adc eax,op2
        adc eax,op3
.ENDIF
        invoke  ExitProcess, 0  ; повернення управління  ОС та вивільнення ресурсів
        end _start                  ; директива завершение программы с именем _start
```

У програмі директивою високого рівня .IF виконуються операції логічного порівняння. Як видно, це порівняння так розкриває логічні конструкції високого рівня, що програми майже не відрізняються.

Приклад 15.2. Проаналізувати масив даних з 20 елементів. Підрахувати кількість елементів, які знаходяться у діапазонах 10 – 20 та 30 – 40. Вивести відповідні повідомлення.

Лістинг 15.3. Програма виконання прикладу 15.2:

```
        .686                            ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти
        option casemap:none             ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\include\kernel32.inc ; системные функции применений…
        include \masm32\include\user32.inc   ; для MessageBox
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
.data                                   ; директива определения данных
        mas dd 1,2,13,4,15,23,34,56,67,87,34,22,34,2,44,67,45,33,42,40
        sum1 dd 0
        sum2 dd 0
        len equ ($-mas)/ type mas  ; визначення кількості елементів mas
        tit1 db "Елементи масиву в діапазоні 10 - 20",0       ; назва вікна
        buf1 db 0,0                     ; буфер виведення повідомлення
ifmt   db "  Кількість=%d",0            ; завдання перетворення одного символу
tit2 db "Елементи масиву в діапазоні 30 - 40",0  ; назва вікна
buf2 db 0,0                     ; буфер виведення повідомлення
ifmt2 db "   Кількість=%d",0   ; завдання перетворення одного символу
.code                                   ; директива начала  сегмента данных
_start:                                 ; метка начала программы с именем _start
        mov ecx,len                     ; кількість чисел
        lea esi, mas                            ;  адреса початку масиву
m1:  mov eax,[esi]                      ; занесення числа в регістр
  .IF (eax<=20)&&(eax>=10)                      ; перевірка умови
        inc sum1                        ; sum1 =sum1 +1
        add esi,4                       ; перехід на наступний елемент
  .ELSEIF (eax>=30)&&(eax<=40)          ; перевірка умови
        inc sum2                                ; sum1=sum1+1
        add esi,4                       ; перехід на наступний елемент
  .ELSE
        add esi,4                       ; перехід на наступний елемент
  .ENDIF
        loop m1
 invoke  wsprintf, ADDR buf1, \ ; адреса буфера, куди буде записана послідовність символів
   ADDR ifmt, \                         ; адреса рядка перетворення формату
    sum1                                ; комірка, вміст якої перетворюється
 invoke MessageBox, 0, \                        ; hwnd – ідентифікатор вікна
   addr buf1, \                 ; адреса рядка, яка містить  текст повідомлення
    addr tit1, \          ; адреса рядка, яка містить  заголовок повідомлення
      MB_ICONINFORMATION     ; вигляд діалогового вікна – одна кнопка ОК
invoke  wsprintf, \                     ; API-функція перетворення числа
 ADDR buf2, \  ; адреса буфера, куди буде записана послідовність символів
  ADDR ifmt2, \                 ; адреса рядка перетворення формату
   sum2                         ; комірка, вміст якої перетворюється
invoke MessageBox, NULL,  addr buf2, addr tit2, MB_ICONQUESTION
 invoke  ExitProcess, 0  ; повернення управління  ОС та вивільнення ресурсів
        end _start          ; директива завершение программы с именем _start
```

Результат виконання програми з лістингу 15.3 наведено на рис. 15.2.

Рис. 15.2. Результат виконання програми з лістингу 15.3

У програмі виводяться два числа в два віконця.

У прикладі 15.3 за допомогою функції MessageBox в кожному рядку виводяться дві цифри, які формуються раніше, та текстовий рядок, якій повторюється в кожному рядку результату.

Приклад 15.3. Задано три масиви чисел. Знайти в кожному рядку максимальне число та вивести результат в спрощене віконце.

Лістинг 15.4:

```
        .686                            ; директива определения типа микропроцессора
        .model  flat, stdcall  ; завдання лінійної моделі пам'яті  та угоди ОС Windows
        option casemap:none             ; отличие строчных и прописных букв
        include    \masm32\include\windows.inc ; файлы структур, констант …
        include    \masm32\include\\user32.inc   ; файлы  интерфейса …
        include \masm32\include\kernel32.inc ; системные функции применений…
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
.data                                                   ; директива определения данных
        max db 0
        nomer dd 1
          str1 db 20,9,2,15                     ; рядок з іменем str1
          str2 db 6,31,1,3                      ; рядок з іменем str2
          str3 db 4,6,155,2                     ; рядок з іменем str3
        st1 db " Вывод максимальных чисел ",0
        st2 db 180 dup(?),0
        stemp db 0
        stemp2 db 3 dup(?),0
        st4 db "              Нажмите ОК",0
        st3 db "-й ряд имеет максимальное число = "
        kol3 = $ - st3
        kol4 = $ - st4
        ifmt db "%d ",0
.code                                                   ; директива начала  сегмента данных
  start:                                              ; метка начала программы с именем start
        xor edx,edx                             ; заповнювання нулями
        mov ebx,3                                       ; завантаження кількості рядків
        lea esi, str1           ; завантаження адреси першого рядка str1
        lea edi,st2
m1:  mov ecx,4                                  ; кількість елементів у рядку
        mov max,0
m2:  mov al,[esi]                       ; завантаження елемента з рядка str1
.IF (al > max)
        mov max,al
.ENDIF
        inc esi                         ; підготовка адреси нового елемента
```

```
        loop m2                         ; ecx := ecx – 1 та перехід на m3, якщо не нуль
invoke  wsprintf, \          ; API-функція перетворення числа
ADDR stemp, \ ; адреса буфера, куди буде записана послідовність символів
  ADDR ifmt, \                                  ; адреса рядка перетворення формату
   nomer      ; комірка з номером рядка, вміст якого перетворюється
            mov al,stemp
            mov [edi],al
            inc edi
        lea esi, st3
        mov ecx,kol3
_m1:
            mov al,[esi]
            mov [edi],al
            inc esi
            inc edi
            loop _m1
            movzx eax,max
invoke  wsprintf, \             ; API-функція перетворення числа
 ADDR stemp2, \  ; адреса буфера, куди буде записана послідов. символів
  ADDR ifmt, \                                  ; адреса рядка перетворення формату
   eax                                          ; регістр, вміст якого перетворюється
            lea esi, stemp2
            mov ecx,3
z1:
        mov al,[esi]
        mov [edi],al
        inc edi
        inc esi
        loop z1
        mov al,10
        mov [edi],al
        inc edi
        inc nomer
        dec ebx                                 ; ebx := ebx – 1
.IF ebx == 2
        lea esi,str2                    ; завантаження адреси нового рядка
        jmp m1                                  ; перехід на новий цикл
.ELSEIF ebx == 1
            lea esi,str3                ; завантаження адреси нового рядка
            jmp m1                              ; перехід на новий цикл
.ENDIF
        mov al,10
        mov [edi],al
        inc edi
        lea esi, st4
        mov ecx,kol4
        inc esi
_z1:
        mov al,[esi]
        mov [edi],al
        inc esi
        inc edi
        loop _z1
invoke MessageBox, \      ; API-функція виведення вікна консолі
 NULL, \                                        ; hwnd – ідентифікатор вікна
  addr st2, \                           ; адреса рядка, яка містить  текст повідомлення
   addr st1, \              ; адреса рядка, яка містить  заголовок повідомлення
    MB_ICONINFORMATION              ; вигляд діалогового вікна
invoke ExitProcess, 0   ; повернення управління  ОС та вивільнення ресурсів
        end start               ; директива окончания программы з іменем start
```

У програмі рядок для виведення результату формується в змінній st2. Спочатку в цю комірку пам'яті записується номер рядка з іменем nomer, потім – символьний масив з іменем st3, за ним – максимальний елемент рядка, а потім – символ переведення рядка (константа 0Ah = 10). Цикл повторюється три рази.

Результат виконання програми з лістингу 15.4 наведено на рис. 15.3.

Рис. 15.3. Результат виконання програми

15.2. Логічна директива switch

Ця директива є альтернативою директиви IF. Правила порівняння такі ж, як й у команди CMP.

Приклад:

```
switch variable
case value1   ; оператор вибору
              ; фрагмент коду
case value2   ; необов'язковий додатковий оператор вибору
              ; фрагмент коду
…
endsw
```

Приклад:

```
include \masm32\macros\macros.asm
…
Switch m1
 case 1
   mov ebx,1            ;  занесення 1 в ebx
 case 2
   mov ebx,2
default
   mov ebx,9
Endsw                   ; закінчення циклу директиви
```

В попередньому прикладі змінна m1 перевіряється логічною директивою switch. Якщо m1 = 1 (case 1), то виконується команда mov ebx,1. При невиконанні умови m1 = 1 буде виконана наступна перевірка (case 2). Якщо не буде виконані перші дві умови, то буде виконана команда mov ebx,9.

15.3. Логічна директива switch$

Ця директива використовується для порівняння текстових рядків. Є дві версії такої директиви: документована (**SENSITIVE)** з символом $ на кінці та недокументована (**INSENSITIVE**) – з літерою "i" перед символом $:

```
Case SENSITIVE        Case INSENSITIVE
    switch$                 switchi$
     case$                  casei$
     else$                  elsei$
     endsw$                 endswi$
```

Приклад:

```
switch$  string_address
case$ "quoted_text1"          ; оператор вибору
                ; фрагмент коду
case$ "quoted_text2"          ; оператор вибору
; фрагмент коду
…
endsw$
```

Робочий фрагмент коду може бути таким:

```
switch$ lpstr
 case$ "word1"
fn  MessageBox,hWnd,"word1","Title",MB_OK
 case$ "word2"
fn  MessageBox,hWnd,"word2","Title",MB_OK
 else$
fn  MessageBox,hWnd,"default","Title",MB_OK
endsw$
```

15.4. Логічні директиви .REPEAT та .WHILE

Директива (оператор) while повторює виконання оператора наступної умови, до тих пір, поки ця умова істинна. Якщо ця умова не істинна із самого початку або стає не істинною в процесі виконання даного оператора, то управління передається операторові, наступному за оператором циклу.

У директиві .WHILE відбувається перевірка умови перед виконанням групи команд. У директиві .WHILE умова записується на початку циклу, а реально перевіряється в кінці циклу. Спочатку програма "стрибає" в кінець циклу та перевіряє там зазначену умову, а потім повертається в початок циклу **та починає виконуватися**. Доти, поки умова залишається *істинною,* виконання цієї групи команд повторюється.

```
.WHILE умова                ; перевірка умови
    код       ; виконується, якщо умова ітинна
```

.ENDW ; закінчення блоку умов, починаючи з директиви .WHILE

Наприклад:

```
.while eax > 0
    sub eax, 1
.endw
```

Директива .REPEAT повторює тіло циклу доти, поки результат виконання умовного виразу для директиви .UNTIL **не набуде помилкового значення**. Умова виконання записується в логічній директиві закінчення .UNTIL.

```
.REPEAT
      код  ; виконується, якщо умова  НЕ істинна
.UNTIL умова
```

У директиві .REPEAT умова записується в кінці циклу, а реально перевіряється на початку циклу.

Например:

```
.repeat
    sub eax, 1
.until eax < 1
```

Якщо в директиві .WHILE, наприклад, треба аналізувати регістр ecx, то перед циклом .WHILE у регістр ecx посилається число. При виконанні директиви .WHILE програма переходить в кінець циклу та перевіряє, чи дорівнює ecx нулю. Якщо так – цикл завершується, якщо ні – здійснюється новий оберт. Природно, регістр ecx потрібно міняти усередині циклу. Тому перед директивою . ENDW розміщують інструкцію dec ecx.

Приклад 15.4. В масиві з 10 чисел зі знаками вибрати числа, які менше 5 та вивести їх на екран

Розв'язання цього прикладу ***без використання*** директив високого рівня наведено у лістингу 15.5.

Лістинг 15.5. Програма розв'язання прикладу 15.2 без використання директив високого рівня:

```
    title CopyRight by Rysovaniy A. N.
    .686            ; директива определения типа микропроцессора
    .model  flat, stdcall  ; завдання лінійної моделі пам'яті  та угоди ОС Windows
    option casemap:none         ; отличие строчных и прописных букв
    include    \masm32\include\windows.inc ; файлы структур, констант …
    include    \masm32\include\\user32.inc   ; файлы  интерфейса …
    include \masm32\include\kernel32.inc ; системные функции применений…
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
    BSIZE equ 10   ; розмір буфера
.data                                   ; директива определения данных
   mas1 db 1, -1, 25, 4, 37, 6, 0, -17, 86, -2         ; масив чисел
   len equ $-mas1    ; визначення кількості байтів масиву mas1
frmt  db "%d",0 ; завдання перетворення одного числа у  послідовність символів
```

```
buf  db BSIZE dup(?)          ; резервування пам'яті для буфера
     crlf  db 0dh,0ah         ; резервування пам'яті для переведення рядка
     stdout DWORD ?           ; резервування в пам'яті 32-розрядної комірки
                                        ; з ім'ям stdout для збереження дескриптора виведення
     cWritten  DWORD  ?    ; для адреси символів виведення
.code
   _start:          ; метка начала программы с именем _start
      invoke  GetStdHandle, STD_OUTPUT_HANDLE
         mov stdout, eax          ; збереження дескриптора
         mov ecx,len              ; занесення кількості байтів масиву mas1
         lea esi, byte ptr mas1 ; занесення адреси масиву mas1
nxt: mov eax, [esi]               ; занесення числа з пам'яті, адреса якого в esi
         cmp al,5                 ; порівняння числа з 5
         jl m1                  ; перейти, якщо менше (для чисел зі знаками)
         jmp m2          ;
m1: push esi             ;
         push ecx        ;
         movsx eax,al     ; розширення числа до подвійного слова
invoke  wsprintf, \    ; API-функція перетворення 32-розрядного числа
 ADDR buf, \ ; адреса буфера, куди буде записана послідовність символів
  ADDR frmt, \             ; адреса рядка перетворення формату
   eax                     ; регістр, зміст якого необхідно перетворити
invoke  WriteConsoleA, stdout, \    ; API-функція виведення на екран
   ADDR buf, BSIZE, \ ; адреса початку повідомлення та розмір повідомлення
         ADDR cWritten, 0    ; адреса, де зберігається число символів
invoke  WriteConsoleA, stdout, ADDR crlf, 2, ADDR cWritten, NULL
         pop ecx
         pop esi
m2: inc esi           ; підготовка нової комірки масиву
         loop  nxt       ; перейти на nxt, якщо ecx ≠ 0
invoke  ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
         end _start          ; директива завершение программы с именем _start
```

Процес отримання повідомлень при розв'язанні прикладу 15.4 через файловий менеджер FAR наведено на рис. 15.4.

Перед початком циклу в регістр ecx заноситься кількість байтів масиву. Перед викликом процедур Windows зберігаються регістри, тому що вони можуть бути змінені при обробці цих процедур. Другий виклик процедури WriteConsoleA необхідний для переведення рядка на другу позицію.

Рис. 15.4. Процес отримання повідомлень при розв'язанні прикладу 15.4

Перший параметр функції wsprintf – це ADDR buf, тобто адреса буфера, куди буде записана послідовність символів.

Другий параметр функції wsprlntf – ADDR ifmt, тобто адреса рядка формату, яка задає тип перетворення.

При перевірці правильності функціонування програми у налагоджувачі OllyDbg при використанні функцій, які не потрібно перевіряти, у покроковому режимі треба натиснути кнопку F8.

Разом з числом, яке виводиться, можна вивести й текст. Для цього необхідно в команді frmt db "%d",0 в скобках навести текст, наприклад frmt db "digit = %d",0

Для того щоб прослідкувати виконання програми не входячи в API-функцію слід перед виконанням цієї функції натискувати кнопку F8.

Спробуємо замінити команду loop в лістингу 15.4 директивами .WHILE і .ENDW, за допомогою яких виведення на екран чисел може виглядати так, як у лістингу 15.6.

Лістинг 15.6. Програма розв'язання прикладу 15.2 з використанням директив високого рівня:

```
    title CopyRight by Rysovaniy A. N.
    .686            ; директива определения типа микропроцессора
    .model  flat, stdcall ; задание линейной модели памяти
    option casemap:none         ; отличие строчных и прописных букв
    include    \masm32\include\windows.inc ; файлы структур, констант …
    include  \masm32\include\\user32.inc   ; файлы  интерфейса …
    include \masm32\include\kernel32.inc ; системные функции применений…
    includelib \masm32\lib\user32.lib
    includelib \masm32\lib\kernel32.lib
    BSIZE equ 10
.data
        mas1 db 1, -1, 25, 4, 37, 6, 0, -17, 86, -2
        len equ $-mas1  ; обчислення кількості байтів у масиві mas1
        frmt   db "%d",0  ; завдання перетворення одного числа  у послідовність символів
        buf  db BSIZE dup(?)    ; резервування пам'яті для буфера
        crlf  db 0dh,0ah            ; резервування пам'яті для переведення рядка
stdout DWORD ?       ; резервування в пам'яті 32-розрядної комірки
                                        ; з ім'ям stdout для збереження дескриптора виведення
cWritten  DWORD  ? ; резервування 32-розрядної комірки пам'яті
                                                ; з ім'ям cWritten для адреси символів виведення
.code                                   ; директива начала  сегмента данных
_start: invoke  GetStdHandle, STD_OUTPUT_HANDLE
        mov stdout, eax                         ; збереження дескриптора
        mov ecx,len             ; занесення кількості байтів масиву mas1
        lea esi, byte ptr mas1              ; занесення адреси масиву mas1
  .WHILE ecx!=0                                 ; продовжувати, якщо ecx ≠ 0
        mov eax, [esi]
        cmp al,5                    ; порівняння числа з 5
        jl m1                ; перейти, якщо менше (для чисел зі знаками)
        jmp m2
m1:  push esi
        push ecx
        movsx eax,al
invoke  wsprintf, \             ; API-функція перетворення 32-розрядного числа
 ADDR buf, \ ; адреса буфера, куди буде записана послідовність символів
  ADDR frmt, \                      ; адреса рядка перетворення формату
   eax                                ; регістр, зміст якого необхідно перетворити
invoke  WriteConsoleA, stdout, \    ; дескриптор стандартного пристрою виведення
     ADDR buf, BSIZE, \ ; адреса початку повідомлення та  розмір повідомлення
```

```
        ADDR cWritten, 0            ; адреса, де зберігається число символів
invoke  WriteConsoleA, stdout, ADDR crlf, 2, ADDR cWritten, 0
        pop ecx
        pop esi
m2:  inc esi                        ; підготовка нової комірки масиву
        dec ecx                     ; зменшення кількості проходжень циклу
  .ENDW             ; закінчення директиви
invoke  ExitProcess, 0  ; повернення управління  ОС та вивільнення ресурсів
        end _start          ; директива завершение программы с именем _start
```

Приклад 15.5. Обчислити 4 значення функції $Y = 7^x$, де x змінюється від 0,3 з кроком 0,1. Результат зберегти у пам'яті та відобразити на екрані.

Для математичного розв'язання такої задачі запишемо, що

$$Y = 7^x = 2^{x\log_2 7}$$

Таким чином, на першому етапі за допомогою команди fyl2x виконаємо обчислення показника степеня: $x*\log_2(7)$. На другому етапі виконаємо команду f2xm1 для обчислення вже $2^y - 1$ при $-1 \le x \le +1$, де $y = \log_2 7$. На третьому етапі додамо до результату одиницю, щоб скоректувати результат виконання команди f2xm1.

Лістинг 15.7. Програма розв'язання прикладу 15.5 з використанням директиви .WHILE:

```
        .686    ; директива определения типа микропроцессора
        .model  flat, stdcall  ; задание линейной модели памяти
        option casemap:none       ; відмінність малих та великих  літер
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib user32,kernel32,fpu
.data
        name_tit db " z = 7^x ",0           ; назва вікна MsgBox
        temp DWORD 10 dup (?)               ; буфер
        y DWORD 0.3                         ; показник степеня
        x DWORD 7.0                         ; основа степеня
        one dd 1.0                  ; для корекції x^y – 1
        step DWORD  0.1             ; крок step := 0.1
        res DT 0                    ; змінна для збереження результату
        tmpY DD 0                   ; для збереження тимчасового Y
.code                               ; директива начала  сегмента данных
  start:                            ; метка начала программы с именем start
        mov ecx,4                   ; лічильник кількості обчислень
        finit               ; ініціалізація співпроцесора
        fld y                       ; завантаження y = 0.3
        fst tmpY                    ; тимчасове збереження Y
.WHILE ecx != 0                     ; цикл
        fld x                       ; завантажуємо в стек  x = 7
        fyl2x                       ; z = y*log2(7)
        f2xm1                       ; 2^y – 1,    -1 ≤ x ≤ +1
        fld one                     ; завантаження 1  для корекції x^y – 1
        fadd st(0),st(1)            ; ST(0) := x^y – 1 + 1
        fstp res                    ; запам'ятовування результату
        push ecx                    ; підготовка до виклику API-функцій
invoke FpuFLtoA,          ; API-функція перетворення дійсного числа
 ADDR res,                ; адреса операнда для перетворення
  10,             ; кількість десяткових знаків після коми (10)
   ADDR temp,                ; адреса буфера для символів, які перетворяться
    SRC1_REAL or SRC2_DIMM       ; 1-й операнд – в пам'яті, 2-й – число
invoke MessageBox, NULL,addr temp, addr name_tit, MB_OK
```

```
    pop ecx                    ; відновлення лічильника циклів
    fld tmpY                   ; відновлення тимчасового У
    fadd step                  ; збільшення кроку на 0.1
    fst tmpY                   ; збереження нового тимчасового У
    dec ecx                    ; зменшення лічильника на 1
.ENDW                          ; закінчення циклу WHILE
invoke  ExitProcess, 0  ; повернення управління  ОС  та вивельнення ресурсів
      end start       ; директива завершение программы с именем start
```

Вигляд вікон з обчислюваними результатами виконання цієї програми наведено на рис. 15.5.

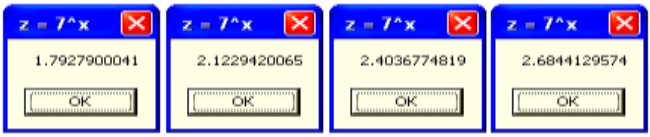


Рис. 15.5. Вигляд вікон з обчисленими результатами прикладу 15.5

При налаголженні програми треба звернути увагу на те, як розкривається директива .WHILE: спочатку програма "стрибає" на закінчення циклу (в добавлені нею команди)

```
or ecx,ecx
jnz … ,
```

а потім повертається на початок циклу та виконує його.

Цими додатковими командами налагоджувач перевіряє значення лічильника (умови в даному випадку) та встановлює відповідні прапорці, а потім повертається на початок циклу й виконує його.

У цій програмі не очищуються регістри співпроцесора. Тому таку програму можна використати тільки по-особливому, щоб не зіпсувати результати інших підпрограмам.

При використанні директиви .REPEAT треба стежити за кількістю циклів. Розглянемо приклад.

Приклад 15.6. Додати парні елементи двох масивів дійсних чисел та вивести результат у вікно.

Лістинг 15.8. Програма виконання прикладу 15.6:

```
title CopyRight by Rysovaniy A. N.   Обчислення mas1[i] + mas2[i]
.686    ; директива определения типа микропроцессора
.model  flat, stdcall  ; задание линейной модели памяти
option casemap:none       ; відмінність малих та великих  літер
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib user32,kernel32,fpu
.data
BSIZE equ  25             ; 24  (6 елементів масиву) + inf  - розмір буфера
inf2 db "Нажата кнопка OK. Задача решена", 0
titl2 db "Автор программы:  ...", 0
inf3 db "Нажата кнопка CANCEL. Вы уверены в своих действиях?",0
mas1 dd 1.1, -1.1, 2.5, 4.0, 3.5, -1.7
```

```
        mas2 dd 1.2, -2.2, 3.6, 4.0, 5.5, -6.7
        len1  EQU ($-mas2)/4   ; обчислення кількості цифр
        mas3 dt 6 dup(0)                    ; резервування комірок під mas3
        st1 db "Program of addition of arrays of numbers",0 ; назва віконця
        inf db "Сумма элементов массивов:",10,13
        buf BYTE BSIZE dup (?),0    ; кількість байтів для запису послід. символів
.code
 start:
        mov ecx, len1-1          ; лічильник перевіряється в .UNTIL ecx >= 5
        lea  esi, mas1       ; завантаження адреси масиву mas1
        lea  edi, mas2           ; завантаження адреси масиву mas2
        xor edx,edx    ; зміщення для mas3
        finit                    ; ініціалізація співпроцесора
.REPEAT                          ; повторювати, поки не виконується умова
        fld dword ptr [esi] ; занесення числа з пам'яті, адреса якого в esi
        fld dword ptr [edi] ; занесення числа з пам'яті, адреса якого в edi
        fadd                             ; додавання елементів двох масивів
        fstp mas3[edx]                   ; занесення до mas3
        push ecx                    ; збереження перед викликом API-функцій
        push edx                    ; збереження перед викликом API-функцій
invoke FpuFLtoA,ADDR mas3[edx], ; адреса операнда для перетворення
  3,ADDR buf[edx], ; кількість знаків після коми та адр. буфера для перетворення
   SRC1_REAL or SRC2_DIMM ; 1-й операнд – в пам'яті, 2-й – число
        add esi, 4       ; підготування для вибору наступного елемента в esi
        add edi, 4       ; підготування для вибору наступного елемента в edi
        pop edx            ; повернення значення зміщення
        add edx, 6      ; 6 знаків під число
        pop ecx         ; повернення значення лічильника
        dec ecx         ; рахування циклів, що залишилися
.UNTIL ecx >= 5
invoke MessageBox, 0, addr inf, addr titl2, MB_OKCANCEL or MB_ICONINFORMATION
        cmp eax,1
        jz m1
        invoke MessageBox, 0, addr inf3, addr titl2, MB_ICONERROR
        jmp m2
m1: invoke MessageBox, 0, addr inf2, addr titl2, MB_ICONWARNING
m2: invoke ExitProcess, 0 ; повернення управління  та визволення ресурсів
        end start
```

Результат виконання програми з лістингу 15.4 наведено на рис. 15.6.

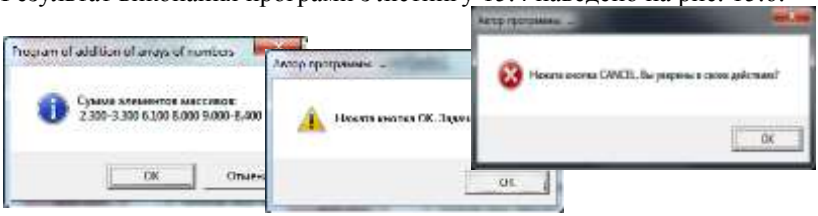


Рис. 15.6. Результат виконання програми

Такого ж результату можна отримати, якщо змінити код на наступний:

```
        .UNTIL ecx >= 5
invoke MessageBox, 0, addr inf, addr titl2, MB_OKCANCEL or MB_ICONINFORMATION
        .IF eax == IDOK
invoke MessageBox, 0, addr inf2, addr titl2, MB_ICONWARNING
        .ELSE
```

```
invoke MessageBox, 0, addr inf3, addr titl2, MB_ICONERROR
    .ENDIF
    invoke ExitProcess, 0
```

У програмі в ecx заноситься число 5, яке на одиницю менше, ніж кількість елементів масивів. Це відбувається з тієї причини, що цикл виконується при ecx = 0.

У зв'язку з тим, що в програмі виводиться 6 цифр та на кожну з них відводиться по 6 розрядів, буфер в 36 байтів достатній, щоб відобразити всі числа, але ще недостатній для виведення назви повідомлення функції MessageBox. Тому необхідне хоча б один байт для початку зберігання назви віконця. Тобто 37 байтів – це мінімальний розмір буфера.

15.5. Директива .BREAK

Директива .BREAK використовується, якщо потрібно перервати цикл і вийти з циклу, наприклад:

```
.WHILE edx==1
inc eax
.IF eax==8
.BREAK
.ENDIF
.ENDW
```

Якщо eax=8, цикл while буде перерваний.

15.6. Директива .CONTINUE

Директива .CONTINUE має синтаксис

.CONTINUE [.IF умова]

Ця директива робить пропуск в коді, що виконує умову циклу найближчого циклу, в якому вона розміщена. Якщо вказаний умовний вираз розміщений в директиві .IF, то пропуск буде виконаний тільки в тому випадку, якщо значення <умова> істинне.

Ця директива вірна тільки усередині блока .REPEAT або .WHILE

Завдання для практичного виконання

Написати програми з використанням директив високого рівня згідно з останньою цифрою студента в групі. Видати на екран через спрощене віконце: призначення програми, послідовні запити на введення чисел та результат операції. Вказати прізвище автора програми та свій e-mail.

1. У масиві дійсних чисел знайти найменше число та вивести на екран сам масив й це число.

2. Задано два масиви дійсних чисел. Додати тільки ті пари чисел, які більше нуля. Сформувати новий масив з цих чисел. Вивести на екран всі три масиви.

3. Задано два масиви дійсних чисел. Сформувати масив для чисел, які попарно дорівнюють одне одному. Вивести на екран усі три масиви.

4. Задано два масиви дійсних чисел. Додати мінімальні числа з 2-х масивів. Вивести на екран масиви та отримане число.

5. У масиві дійсних чисел знайти корінь найбільшого числа та вивести на екран масив, число та його корінь.

6. Знайти числа, які отримані при піднесенні 2 до степеня чисел від 0 до 20. Вивести на екран всі числа та їх степінь.

7. У масиві дійсних чисел знайти числа, у яких розряд [0] цілої частини дорівнює 1. Вивести на екран масив та знайдені числа.

8. У масиві дійсних чисел знайти негативні числа та вивести на екран початковий масив та масив, який отримано.

9. У масиві дійсних чисел знайти числа, у яких розряд [2^{-1}] дрібної частини дорівнює 0. Вивести на екран масив ці числа.

10. Задано два масиви дійсних чисел. Перемножити тільки ті пари чисел, які менше нуля. Сформувати новий масив з цих чисел. Вивести на екран всі три масиви.

15.8. Лабораторна робота "Директиви умовного асемблювання"

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм з використанням директив умовного асемблювання, написаних мовою асемблера для МП платформи x86.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати свій варіант та написати програму з використанням **директив умовного асемблювання та дійсних чисел** з обов'язковим виведенням результатів у спрощене вікно (вікна) та додати прізвище та ініціали про автора програми, свій e-mail, варіант завдання та умову цього завдання.

Використати функцію MessageBox разом із графічними позначками цієї функції та вказати прізвище автора програми.

1. Задано масив A з $N = 5$ елементів. Навести алгоритм та програму визначення суми елементів масиву A, для яких біти 2 та 10 збігаються. Вивести відповідні повідомлення.

2. Проаналізувати масив даних mas1 з 6 елементів. Підрахувати і зберегти в комірках пам'яті кількість елементів масиву, якщо їх значення менші або більші від першого елемента та дорівнюють останньому. Вивести відповідні повідомлення.

3. Задано масив A з $N = 7$ елементів. Навести алгоритм та програму формування масиву B з елементів масиву A, у яких 0, 2 та 5 біти мають нулі.

4. Проаналізувати масив даних з 5 елементів. Додавати елементи масиву доти, поки значення суми не перевищить 512 та не буде менше 510. Зберегти номер елемента, на якому відбулося переповнення. Якщо сума елементів не досягла значення 512, то видати відповідне повідомлення.

5. Проаналізувати масив даних з 6 елементів. Елементами масиву є числа 32 та 128. Підрахувати та вивести на екран кількість елементів, які за значенням знаходяться у середині цього діапазону.

6. Проаналізувати масив даних A з 5 елементів. Створити масив B, до якого входять елементи першого масиву, що задовольняють умову $-1 \le B_i > 128$. Перервати виконання програми, якщо буде знайдено 5 елементів зі значенням 12. Вивести відповідні повідомлення.

7. Проаналізувати 2 масиви, що складаються з 6 елементів кожен. Підрахувати кількість елементів першого масиву, що мають рівні значення в другому масиві. Вивести відповідні повідомлення.

8. Проаналізувати масив даних з 4 елементів. Підрахувати кількість елементів, значення яких дорівнює 55h та рахунок перервати, якщо кількість таких елементів перевищить 3. Вивести відповідні повідомлення.

9. Задано масиви A і B по $N = 6$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо у елементів Ai та Bi біти 2, 3 та 8 збігаються, то $Ci = Ai + Bi$. Вивести відповідні повідомлення.

10. Проаналізувати масив даних з 7 елементів. Підрахувати кількість елементів, які находяться у діапазонах 10 – 20 та 30 – 40. Вивести відповідні повідомлення.

12. Задано масив A з $N = 7$ елементів. Структура масиву A: $X1, Y1; X2, Y2$; … . Навести алгоритм та програму визначення кількості пар, для яких виконується умова $Xi \le Yi$. Вивести відповідні повідомлення.

13. Задано масив A з 5 елементів. Навести алгоритм та програму визначення елементів масиву A, які задовольняють умову $6 \le A_i < 3A$, та розмістити їх в новому масиві. Вивести відповідні повідомлення.

14. Задано масиви A і B по $N = 6$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо $Ai - Bi \le 0$, то $Cj = Ai$.

15. Задано масив A з $N = 5$ елементів. Навести алгоритм та програму визначення суми та кількості елементів масиву A, які задовольняють умову $Ai \ge E$ при $E = -11$. Вивести відповідні повідомлення.

Зміст звіту

1. Постановка задачі для конкретного варіанта.

2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.

3. Лістинг програми та коментарі до всіх команд.

4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання.

5. Короткий опис виконання програми.

6. Висновки за результатами роботи.

Приклад 15.7:

Проаналізувати масив даних з 10 елементів розміром у подвійне слово. Додавати елементи масиву доти, поки значення суми не перевищить 15. Зберегти номер елемента, на якому відбулося переповнення. Видати повідомлення на екран монітора (текстове або/чи числове).

Блок-схему алгоритму виконання прикладу наведено на рис. 15.7.

Розв'язання прикладу 15.3 з використанням директив високого рівня, але поки без виведення даних, наведено у лістингу 15.9.

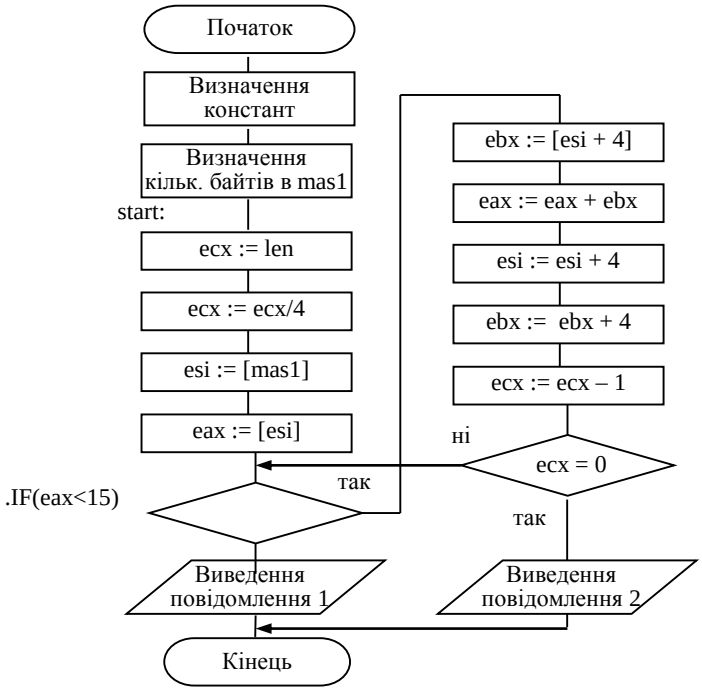


Рис. 15.7. Блок-схема алгоритму виконання прикладу 15.7

У наведеній програмі при виникненні необхідності додавання двох чисел одного масиву слід використати додатковий регістр (для зберігання адреси другого числа). Таким регістром вибрано регістр ebx. Завантаження в цей регістр значення адреси другого числа виконується командою mov ebx,[esi+4].

У зв'язку з тим, що за завданням вибрано формат регістрів у 32 розряди (4 байти), для підготовки зчитування нових слів виконуються команди

```
add esi,4
add ebx,4.
```

Вікна налагоджувача після налагодження програми з лістингу 15.9 наведено на рис. 15.8.

Лістинг 15.9:

```
        title CopyRight by Rysovaniy A. N.                    rysov@rambler.ru
        .686                          ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти
        option casemap:none           ; отличие строчных и прописных букв
.data                                                  ; директива определения данных
        mas1 dd 1,4,5,4,6,7,3,8,2,10                          ; масив чисел
        len equ $-mas1                  ; визначення кількості байтів в масиві
.code                           ; директива начала  сегмента данных
_start:                                 ; метка начала программы с именем _start
        mov ecx,len             ; завантаження лічильника
        shr ecx,2                       ; визначення кількості подвійних слів
        lea esi, mas1                   ; завантаження адреси початку масиву
        mov eax, [esi]                    ; завантаження числа
m1:
    .IF(eax<15)                 ;  умова
        mov ebx,[esi+4]                 ; завантаження нового числа з масиву
        add eax,ebx             ; додавання елементів
        add esi,4                       ; розрахунок адреси нового числа
        add ebx,4               ; розрахунок адреси наступного числа
        loop m1             ; перейти на m1, якщо ecx = 0
    .ENDIF                      ; закінчення директиви високого рівня
        ret                     ; повернення управління  ОС
        end _start          ; директива завершение программы с именем _start
```

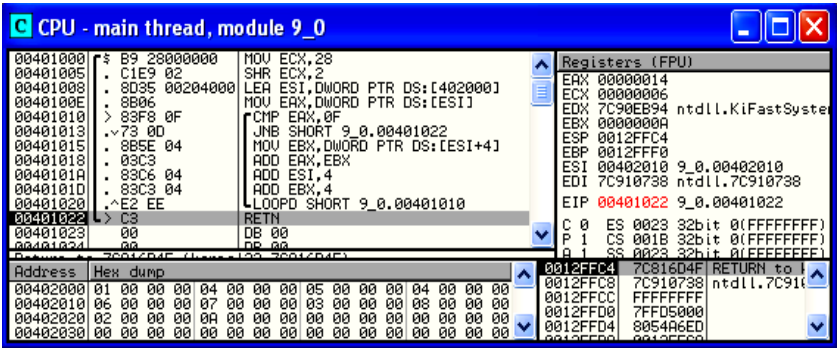


Рис. 15.8. Вікна налагоджувача після налагодження програми з лістингу 15.6

Для виконання другої частини роботи необхідно організувати виведення даних на екран. Для цього доцільно використати API-функцію WriteConsoleA, фрагмент застосування якої може бути таким:

```
    ...
    BSIZE     equ 15
   .data                                ; директива определения данных
     stdout dd ? ; дані для дескриптора виведення, що не визначені
     buf    db BSIZE dup(?)   ; резервування пам'яті для буфера
     frmt    db  "%d",0             ; задання перетворення одного символу
 cWritten    dd  ?    ; комірки пам'яті для адрес  символів виведення
   . . .
   .code                                  ; директива начала  сегмента данных
   _start:                              ; метка начала программы с именем _start
   . . .
invoke GetStdHandle, STD_OUTPUT_HANDLE ; отриманого дескриптора
      mov stdout, eax               ; запам'ятовування отриманого дескриптора
   . . .
invoke  wsprintf, \                 ; API-функція перетворення числа
 ADDR buf, \ ; адреса буфера, куди буде записана послідовність символів
  ADDR frmt, \                  ; адреса рядка перетворення формату
   edx                              ; регістр, вміст якого перетворюється
invoke  WriteConsoleA, stdout,\                 ; API-функція виведення на екран
    ADDR buf,\                      ; адреса початку повідомлення
      BSIZE, \                      ; розмір повідомлення
        ADDR cWritten, 0  ; адреса, де зберігається кількість символів
invoke ExitProcess, 0 ; повернення управління ОС та вивільнення ресурсів
      end _start                ; директива окончания программы з іменем _start
```

У цій програмі використовуються файли user32.inc і user32.lib, що зберігають інформацію про функцію **wsprintf**. Кількість параметрів цієї функції поки дорівнює трьом:

1. Параметр ADDR buf – це адреса буфера, куди буде записана послідовність символів. Пам'ять для буфера виділяється рядком buf db BSIZE dup(?), який резервує кількість байтів, заданих змінною BSIZE. Про резервування байтів пам'яті говорить слово dup – скорочене від англійського слова *duplication* (повторення). Знак питання в дужках після dup свідчить про те, що значення байтів наперед не визначене.

Розмір буфера, позначений ім'ям BSIZE, є реальним числом, яким асемблер замінить змінну BSIZE та яке задається рядком BSIZE equ 15.

2. Параметр ADDR ifmt – це адреса рядка формату, яка задає тип перетворення. Цей рядок складається з символів і завжди завершується нульовим байтом. Рядок "%d",0 задає перетворення одного цілого числа в послідовність символів, а рядок "%d%d",0 – перетворення двох чисел.

3. Параметр edx указує функції wsprintf на те, що число, яке потребує перетворення в послідовність символів, знаходиться, наприклад, у регістрі edx.

Процедура WriteConsoleA виводить ці символи на екран. За першим параметром цієї функції (stdout) перетворене у символ число виводиться через консоль, за другим (параметр ADDR buf) указується адреса початку повідомлення, за третім (BSIZE) – розмір повідомлення, за четвертим (ADDR

cWritten) – адреса ділянки пам'яті, де процедура WriteConsoleA зберігає кількість виведених на екран символів.

Приклад 15.8. Проаналізувати масив даних mas1 з 6 елементів. Підрахувати і зберегти в комірках пам'яті кількість елементів масиву, якщо їх значення менші або більші від першого елемента та дорівнюють останньому.

Лістинг 15.10:

```
        .686                        ; директива определения типа микропроцессора
        .model flat,stdcall         ; завдання лінійної моделі пам`яті та угоди ОС Windows
        option casemap:none         ; отличие строчных и прописных букв
        include C:\masm32\include\mcdsrv32.inc
        include C:\masm32\include\windows.inc ; файлы структур, констант …
        include C:\masm32\include\kernel32.inc ; файли систем. функцій застосувань…
        include C:\masm32\include\\user32.inc   ; файлы  интерфейса …
        includelib C:\masm32\lib\mcdsrv32.lib
        includelib C:\masm32\lib\kernel32.lib
        includelib C:\masm32\lib\user32.lib
        IDI_ICON EQU 1001
.data
   _min dd 0
   _max dd 0
   _toj dd 0
   _st1 db "ЛР7 Директиви умовного асемблювання",0
   _st2 dd ?,0
   _mas1 dd 4,3,44,12,22,4
   _El1 dd 4
   ifmt db "Исходный массив:",0dh,0ah,\
   "04  03  44  12  22  04",0dh,0ah,0ah,\
   "Количество елементов, равных первому = %d",0dh,0ah,0ah,\
   "Количество елементов меньших, чем первый = %d",0dh,0ah,0ah,\
   "Количество елементов больших, чем первый = %d",0dh,0ah,0ah,\
   "Автор:  Гладких А.А., КІТ-39",0
   params MSGBOXPARAMS <>
.code
_nach:
        mov ecx,5
        lea esi,_mas1
        add esi,4
m1:
        mov ebx,_El1
        mov eax,dword ptr [esi]
        .IF eax==ebx
        add _toj,1
        .ELSEIF eax > ebx
        add _max,1
        .ELSEIF eax < ebx
        add _min,1
        .ENDIF
        add esi,4
      loop m1
        invoke wsprintf,ADDR _st2,ADDR ifmt,_toj,_min,_max
           mov params.cbSize,SIZEOF MSGBOXPARAMS
           mov params.hwndOwner,0
           invoke GetModuleHandle,0
           mov params.hInstance,eax
           mov params.lpszText,offset _st2
           mov params.lpszCaption,offset _st1
           mov params.dwStyle,MB_USERICON
```

```
    mov params.lpszIcon,IDI_ICON
    mov params.dwContextHelpId,0
    mov params.lpfnMsgBoxCallback,0
    mov params.dwLanguageId,LANG_NEUTRAL
    invoke MessageBoxIndirect,ADDR params
  invoke ExitProcess,0                    ; закінчення програми
  end _nach
```

Результат виконання програми наведено на рис. 15.9.

Рис. 15.9. Результат виконання програми з лістингу 15.10

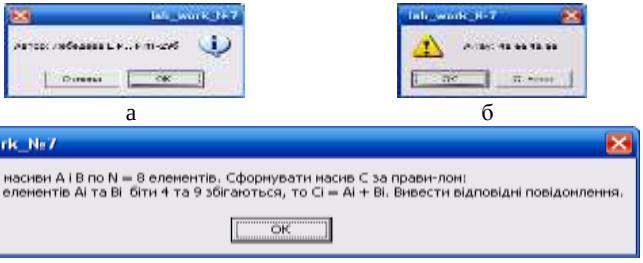
Приклад 15.9. Задано масиви *A* і *B* по *N* = 6 елементів. Навести алгоритм та програму формування масиву *C* за таким правилом: якщо у елементів *Ai* та *Bi* біти 4 та 9 збігаються, то *Ci = Ai + Bi.* Вивести відповідні повідомлення.

Лістинг 15.11:

```
      .686                          ; директива определения типа микропроцессора
      .model flat,stdcall ; задание линейной модели памяти та угоди ОС Windows
      option casemap:none           ; отличие строчных и прописных букв
      include C:\masm32\include\windows.inc ; файлы структур, констант …
      include\masm32\include\kernel32.inc ; файли систем. функцій застосувань
      include C:\masm32\include\\user32.inc ; файлы  интерфейса …
      includelib C:\masm32\lib\kernel32.lib
      includelib C:\masm32\lib\user32.lib
.data                               ; директива определения типа микропроцессора
  mas1 dd 2 dup(25h, 147h, 55h, 0A4h) ; резервування пам`яті для mas1
  mas2 dd 2 dup (25h, 34h, 55h, 155h) ; резервування пам`яті для mas2
  temp dd 0                         ; резервування пам`яті для змінної temp
  mas3 dd 8 dup(0)
  stdout DWORD ?                    ; резервування 32р. комірки
  buf db BSIZE dup(?)               ; резервування пам`яті для буфера
  frmt db "%x", 0                   ; задання перетворення числа
  cWritten DWORD ? ; резервув. 32-розр. комірки для адреси виведення
  mt db " lab_work_№7", 0     ; назва віконця
  ifmt db "Автор: Лебедева Е.И., КИТ-296", 0
  ifmt1 db "Задано масиви A і B по N = 8 елементів. Сформувати масив C за правилом:",
0ah, 0dh,
  "якщо у елементів Ai та Bi  біти 4 та 9 збігаються, то Ci = Ai + Bi. Вивести відповідні по-
відомлення.", 0ah, 0dh, 0
  ifmt2 db "Array: %x %x %x %x " , 0  ; формат виведення  mas3
.code
_start:                     ; метка начала программы
     lea esi, mas1              ; записати адресу початку масиву mas1
     lea edi, mas2              ; записати адресу початку масиву mas2
     mov ecx, 8                 ; лічильник кількості пар чисел
     xor edx, edx               ; edx = 0
m4: mov ebx, [esi]     ; помістити число з mas1
```

```
      mov eax, [edi]       ; помістити число з mas2
      mov temp, eax        ;  збереження для подальшого складання
      and ebx, 108h   ; логічне множення числа з першого масиву на маску
      and temp, 108h   ; логічне множення числа з другого масиву на маску
      sub ebx, temp          ; EBX:=ebx-temp
     .IF ebx==0            ; перевірка різниці на рівність 0
        mov ebx, [esi]
        add ebx, eax        ; формування елемента mas3
        mov mas3[edx], ebx  ; запис в mas3
        add esi, 4          ; просування по першому масиву
        add edi, 4          ; просування по другому масиву
        dec ecx
      .IF ecx!=0          ; перевірка регістра ecx на нерівність 0
       jmp m4             ; якщо регістр ecx не дорівнює 0, то перейти на m4
      .ENDIF              ; закінчення директиви ecx!=0
    .ELSE                ; альтернативний блок коду
      add esi, 4          ; просування по першому масиву
      add edi, 4          ; просування по другому масиву
      add edx, 4
      dec ecx
      .IF ecx!=0          ; перевірка регістра ecx на нерівність 0
       jmp m4             ; якщо регістр ecx не дорівнює 0, то перейти на m4
      .ENDIF              ; закінчення директиви ecx!=0
   .ENDIF                 ; закінчення директиви ebx==0
      invoke Sleep, 1000d
invoke MessageBox, NULL, addr ifmt, addr mt, \
  MB_OKCANCEL+MB_ICONINFORMATION +180000h
      xor eax, 1
      jz a1                          ; якщо натиснуто OK
        jmp  a2                              ;  інакше – “Отмена”
a1:  invoke Beep, 01234, 300                 ; звуковий сигнал
invoke wsprintf, ADDR buf, ADDR ifmt2, mas3, mas3[4], mas3[8], mas3[12]
invoke MessageBox, NULL, addr buf, addr mt, \
   MB_OKCANCEL +MB_ICONEXCLAMATION
a2:
       invoke Beep, 02578, 200
       invoke MessageBox, NULL, addr ifmt1, addr mt, MB_OK
a3: invoke ExitProcess, 0
      end _start ; завершение программы с именем _start
```

Результат виконання програми наведено на рис. 15.10.



а б

с

Рис. 15.10. Результат виконання програми з лістингу 15.11:
а – перше вікно програми;
б – вікно програми при натисканні на першому вікні кнопки OK;
с – останнє вікно

15.9. Лабораторна робота "Двовимірні масиви"

Мета заняття:

набути практичних навичок складання, налагодження і виконання програм для програмування двовимірних масивів, написаних мовою асемблера для МП платформи x86 з організацією виведення результату на екран з використанням API-функцій під Win32.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати свій варіант та написати програму з використанням **директив умовного асемблювання та дійсних чисел** з обов'язковим виведенням результатів у спрощене вікно (вікна) та додати прізвище та ініціали про автора програми, свій e-mail, варіант завдання та умову цього завдання.

Використати функцію MessageBox разом із графічними позначками цієї функції та вказати прізвище автора програми.

1. Задано масив A з $N = 5$ елементів. Навести алгоритм та програму визначення суми елементів масиву A, для яких біти 2 та 10 збігаються. Вивести відповідні повідомлення.

2. Проаналізувати масив даних mas1 з 6 елементів. Підрахувати і зберегти в комірках пам'яті кількість елементів масиву, якщо їх значення менші або більші від першого елемента та дорівнюють останньому. Вивести відповідні повідомлення.

3. Задано масив A з $N = 7$ елементів. Навести алгоритм та програму формування масиву B з елементів масиву A, у яких 0, 2 та 5 біти мають нулі.

4. Проаналізувати масив даних з 5 елементів. Додавати елементи масиву доти, поки значення суми не перевищить 512 та не буде менше 510. Зберегти номер елемента, на якому відбулося переповнення. Якщо сума елементів не досягла значення 512, то видати відповідне повідомлення.

5. Проаналізувати масив даних з 6 елементів. Елементами масиву є числа 32 та 128. Підрахувати та вивести на екран кількість елементів, які за значенням знаходяться у середині цього діапазону.

6. Проаналізувати масив даних A з 5 елементів. Створити масив B, до якого входять елементи першого масиву, що задовольняють умову $-1 \le B_i > 128$. Перервати виконання програми, якщо буде знайдено 5 елементів зі значенням 12. Вивести відповідні повідомлення.

7. Проаналізувати 2 масиви, що складаються з 6 елементів кожен. Підрахувати кількість елементів першого масиву, що мають рівні значення в другому масиві. Вивести відповідні повідомлення.

8. Проаналізувати масив даних з 4 елементів. Підрахувати кількість елементів, значення яких дорівнює 55h та рахунок перервати, якщо кількість таких елементів перевищить 3. Вивести відповідні повідомлення.

9. Задано масиви A і B по $N = 6$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо у елементів Ai та Bi біти 2, 3 та 8 збігаються, то $Ci = Ai + Bi$. Вивести відповідні повідомлення.

10. Проаналізувати масив даних з 7 елементів. Підрахувати кількість елементів, які знаходяться у діапазонах 10–20 та 30–40. Вивести відповідні повідомлення.

12. Задано масив A з $N = 7$ елементів. Структура масиву A: $X1$, $Y1$; $X2$, $Y2$; … . Навести алгоритм та програму визначення кількості пар, для яких виконується умова $Xi \le Yi$. Вивести відповідні повідомлення.

13. Задано масив A з 5 елементів. Навести алгоритм та програму визначення елементів масиву A, які задовольняють умову $6 \le A_i < 3A$, та розмістити їх у новому масиві. Вивести відповідні повідомлення.

14. Задано масиви A і B по $N = 6$ елементів. Навести алгоритм та програму формування масиву C за таким правилом: якщо $Ai - Bi \le 0$, то $Cj = Ai$.

15. Задано масив A з $N = 5$ елементів. Навести алгоритм та програму визначення суми та кількості елементів масиву A, які задовольняють умову $Ai \ge E$ при $E = -11$. Вивести відповідні повідомлення.

Зміст звіту

1. Постановка задачі для конкретного варіанта.

2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.

3. Лістинг програми та коментарі до всіх команд.

4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання.

5. Короткий опис виконання програми.

6. Висновки за результатами роботи.

Приклад 15.10. Знайти максимальне значення в рядках двовимірному масиві A(N,M). Вивести ці числа функцію MessageBox.

Лістинг 15.12:

```
        …
.data               ;
        mas1 dd 10,33,3,64,6,5,82,
                7,80,6,78,8,15,4
        len equ ($-mas1)/4/2      ; визначення чисел в одному рядку
        max1 dd 0         ; змінна для максимального числа першого рядка
        max2 dd 0         ; змінна для максимального числа другого рядка
         titl db "Результат программы",0
 buf dd 0,0
 ifmt db "Условие: найти максимальное значение ",0dh,0ah,\
  "в строках двумерных массивов:",0dh,0ah,\
   "10   33   3   64   6   5   82",0dh,0ah,\
   "  7   80   6   78   8  15    4",0dh,0ah,0ah,\
   "Максимальное значение в первой строке:    %d",0dh,0ah,\
   "Максимальное значение во второй строке:  %d",0
.code                                ;
  _start:
```

```
        mov ecx,len       ; кількість чисел в одному рядку
        lea esi, mas1       ; завантаження адреси початку масиву
        mov eax, [esi]        ; завантаження числа
m1:
  .IF(eax >= max1)   ; порівняння числа з першого рядка з максимальним
        mov max1, eax
        add esi,4             ; розрахунок адреси нового числа
        mov eax,[esi]
        loop m1         ; пересилка нового числа
  .ELSE       ; то
        add esi,4                   ; розрахунок адреси нового числа
        mov eax,[esi]
       loop m1
  .ENDIF
         mov ecx,len                ; кількість чисел в одному рядку
 m2:
   .IF(eax >= max2)  ; порівняння числа з другого рядка з максимальним
       mov max2, eax
        add esi,4                    ; розрахунок адреси нового числа
        mov eax,[esi]
        loop m2
   .ELSE   ; иначе
        add esi,4                    ; розрахунок адреси нового числа
        mov eax,[esi]
       loop m2
   .ENDIF
       invoke wsprintf, ADDR buf, ADDR ifmt, max1, max2
       invoke MessageBox, 0, addr buf, addr titl, MB_ICONEXCLAMATION
        invoke ExitProcess,0
        end _start
```

Результат виконання програми наведено на рис. 15.11.

Рис. 15.11. Результат виконання програми

У зв’язку з тим, що числа в масиві всі позитивні, то для знаходження максимального на початку обирається нульове число та з ним порівнюється наступне. На наступному етапі вже це число є максимальним й з ним порівнюється інше, й так далі.

В лістингу 15.13 наведено програму пошуку максимального елемента в кожному рядку.

Лістинг 15.13:

```
.686
.model flat, stdcall
```

```
        option casemap:none
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32
        BSIZE equ 20
                mLeaReg macro reg
                lea esi,reg
                ENDM
                    DATE1 STRUCT
                    elem1 dd ?
                    elem2 dd ?
                    elem3 dd ?
                    elem4 dd ?
                    elem5 dd ?
                    DATE1 ENDS
.data
          str1 DATE1 <1,-1,-2,3,1>
          str2 DATE1 <6,-5,4,2,4>
          str3 DATE1 <4,1,67,-3,2>
          str4 DATE1 <12,-2,5,1,5>
          str5 DATE1 <1,5,2,8,1>
          str6 DATE1 <11,7,-8,4,0>
          str7 DATE1 <18,23,1,-7,9>
        titl1 db "Робота с елементами структури",0
        buf db 10 dup(?),0
        ifmt db "Задана матриця:",10,13,
           " 1 -1 -2  3  1",10,13,
           " 6 -5  4  2  4",10,13,
           " 4  1 67 -3  2",10,13,
           "12 -2  5  1  5",10,13,
           " 1  5  2  8  1",10,13,
           "11  7 -8  4  0",10,13,
           "18 23  1 -7  9",10,13,13,
           "Результат пошуку максимального елемента в кожному рядку:",10,13,
           "A1= %d",10,13,
           "A2= %d",10,13,
           "A3= %d",10,13,
           "A4= %d",10,13,
           "A5= %d",10,13,
           "A6= %d",10,13,
           "A7= %d",0
           res dd 7 dup(?),0
           max dd ?
.code
start:
        mov edx,2
        mov ebx,7
        lea edi,res
        mLeaReg str1
m1: mov ecx,4
        mov eax,[esi]
        mov max,eax
        add esi,4
        cycle:
              mov eax,[esi]
              cmp eax,max
              jl m2
              mov max,eax
              m2:add esi,4
        loop cycle
        mov ebp,max
```

```
        mov [edi],ebp
        add edi,4
        dec ebx
        jz _exit
Switch edx
        case 2
              mLeaReg str2
        case 3
              mLeaReg str3
        case 4
              mLeaReg str4
        case 5
              mLeaReg str5
        case 6
              mLeaReg str6
        case 7
              mLeaReg str7
Endsw
        inc edx
        jmp m1
          _exit:
invoke wsprintf,ADDR buf,ADDR ifmt, res[0],res[4],res[8],res[12],res[16], res[20], res[24]
invoke MessageBox, 0, chr$("Никифоров Денис"), chr$("Автор программы"), 0
        invoke MessageBox, 0,  addr buf, addr titl1, MB_ICONINFORMATION
        invoke ExitProcess, 0
        end start
```

Результат виконання програми наведено на рис. 15.12.

Рис. 15.12. Результат виконання програми

Приклад. Задано масив A з N = 7 елементів. структура масиву A: X1,Y1; X2,Y2; ... . навести алгоритм та програму визначення кількості пар, для яких виконується умова Xi <= Yi

Лістинг 15.14:

```
.686
.model flat, stdcall
option casemap:none
include \masm32\include\windows.inc
include \masm32\macros\macros.asm
```

```
        uselib kernel32, user32
        BSIZE  equ 13
                array  struct
                   _Xi dd ?
                   _Yi dd ?
                Array  ends
.data
        szBuf   db BSIZE dup(?)
        str3    db "«Завдання:",0
        str4    db "Задано масив A з N = 7 елементів. Структура масиву A: X1,Y1; X2,Y2; ... .
Навести алгоритм та програму визначення кількості пар, для яких виконується умова Xi
<= Yi",0
           fmt     db "result = %d",0
           caption db "Lab7",0
A array  <5.3,5.3>,<3.0,5.2>,<7.1,2.6>,<8.9,4.6>,<3.7,8.3>,<4.2,2.1>,<2.1,3.9>
           var_res dd ?
.code
_start:
            xor ecx, ecx        ; Очистка регистра
              xor esi, esi
              mov ecx, 8          ; Размер массива
              .while ecx != 0     ; Цикл проход по массиву
              mov eax, A[edi]._Xi ; Извлечение поля структуры в регистр
              mov edx, A[edi]._Yi ; Извлечение поля структуры в регистр
              .if (eax <= edx)    ; Сравнение полей структуры
              inc esi             ; Увеличение счетчика пар
              .endif              ; Конец условия
              inc edi             ; Увеличение индекса массива
              dec ecx             ; Уменьшение счетчика
              .endw
              invoke wsprintf, ADDR szBuf, ADDR fmt, esi
              invoke MessageBox, NULL, addr szBuf, addr caption, MB_ICONINFORMATION
              invoke MessageBox, NULL, addr str4, addr str3, MB_ICONINFORMATION
              invoke ExitProcess,0
        end _start
```

Контрольні запитання

1. Які конструкції належать до логічних конструкцій високого рівня?
2. За допомогою яких операторів описуються умови порівняння?
3. Які переваги та недоліки логічної конструкції порівняння текстових рядків?
4. В якій конструкції відбувається перевірка умови перед виконанням групи команд?
5. Яка директива використовується для переривання виконання циклу?

16. СТРУКТУРИ

16.1. Організація структур

Структура – це тип даних, що складається з фіксованого числа елементів ***різного*** типу. Наприклад, одне поле може бути байтом, інше – словом і так далі.

Масиви є сукупністю однотипних елементів. У застосуваннях виникає необхідність розглядати деяку сукупність даних **різного** типу як деякий єдиний тип. Це дуже актуально, наприклад, для програм баз даних, де необхідно пов'язувати сукупність даних різного типу з одним об'єктом.

Для звернення до членів структурної змінної використовується спеціальний оператор-крапка.

Для використання структур в програмі необхідно виконати три дії:

1. *Задати шаблон структури.* Це означає визначити новий тип даних, який згодом можна використовувати для визначення змінних цього типу.
2. *Виконати ініціалізацію* конкретної змінної структури (за допомогою шаблона).
3. Організувати *звернення до елементів структури*.

Описати структуру в програмі означає лише вказати її схему або шаблон; пам'ять при цьому не виділяється. Шаблон структури можна розглядати лише як інформацію для транслятора про розташування полів і їх значення за умовчанням.

Визначити структуру означає дати вказівку транслятору виділити пам'ять і привласнити цій області пам'яті символічне ім'я.

Описати структуру в програмі можна тільки один раз, а визначити – будь-яку кількість разів.

Опис шаблона структури

Опис шаблона структури має такий синтаксис:

```
Ім'я_структури    STRUCT
    <опис полів>
Ім'я_структури    ENDS
```

Тут ***<опис полів>*** є послідовністю директив опису даних **db, dw, dd, dq** і **dt**.

Їх операнди визначають розмір полів і, при необхідності, початкові значення. Цими значеннями, можливо, ініціалізуються відповідні поля при визначенні структури (http://subscribe.ru/archive/comp.soft.prog.asm 32/ 200702/11080945.html).

Місцеположення шаблона в програмі може бути довільним, але, відповідно до логіки роботи однопрохідного транслятора, він повинен бути розташований до того місця, де визначається змінна з типом даної структури. Тобто при описі в сегменті даних змінної з типом деякої структури її шаблон необхідно помістити на начало сегмента данных або перед ним.

Спочатку розглянемо спрощену програму з використанням структур.

Приклад 16.1. Обчислити суму елементів головної діагоналі матриці розміром 4 × 4, з використанням даних структурного типу.

Як приклад в програмі спеціально використано **спрощену** сегментацію з декількома директивами .CODE та .DATA. Транслятор потім збирає код і дані разом.

Лістинг 16.1. Програма обчислення елементів головної діагоналі:

```
        title CopyRight by Rysovaniy A. N.
        .686
        .model flat, stdcall
        option casemap:none
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32
DATE1 STRUCT          ; визначення даних СТРУКТУРА з іменем DATE1
     CarName db ?             ; ім'я першого поля структури
     Number  dw ?             ; ім'я другого поля структури
     Name1   dw ?             ; ім'я третього поля структури
     Viol    dd  ?                    ; ім'я четвертого поля структури
DATE1 ENDS                    ; закінчення даних СТРУКТУРА з іменем DATE1
   .data                              ; директива определения данных
        Car1  DATE1 <1,10,2,3>                ; структура з іменем Car1
        Car2  DATE1 <2,-11,6,7>               ; структура з іменем Car2
        Car3  DATE1 <3,12,10,11>              ; структура з іменем Car3
        Car4  DATE1 <4,13,14,15>              ; структура з іменем Car4
        ifmt db "Исходный массив - ввиде матрицы:",0dh,0ah,\
        "01   10   02   03",0dh,0ah,\
        "02  -11   06   07",0dh,0ah,\
        "03   12   10   11",0dh,0ah,\
        "01   13   14   15",0dh,0ah,0ah,\
        "Результат  сложения значений ",0dh,0ah,\
        "элементов главной диагонали: = %d",0dh,0ah,0ah,\
        "Автор программы:  Рысованый А.Н., НТУ ХПИ",0
        titl1 db "Сумма элементов главной диагонали",0 ; назва віконця
        buf db 10 dup(?),0 ;
.code
start:
        xor ebx,ebx
        movzx bx,Car1.CarName ; заповнювання нулями старшої частини
        add bx,Car2.Number              ; bx := Car1.CarName  +  Car2.Number
        add bx,Car3.Name1               ;
        add  ebx,Car4.Viol              ;
.data                 ; директива определения данных (спрощена сегментація)
        res dd ?      ; резервування 32-розрядної комірки пам'яті з ім'ям res
.code                           ; директива початку сегмента-даних
        mov res,ebx ;
        invoke wsprintf,ADDR buf,ADDR ifmt,res   ; функція перетворення res
invoke MessageBox, 0, ADDR buf, ADDR titl1, MB_ICONINFORMATION+180000h
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
        end start     ; директива окончания программы з іменем start
```

На рис. 16.1 наведено результат виконання програми з лістингу 16.1.

У програмі лістингу 16.1 кожній з чотирьох структур присвоєно ім'я, а в свою чергу, кожен елемент структури теж має своє ім'я, яке наведено в структурі DATE1 STRUCT.

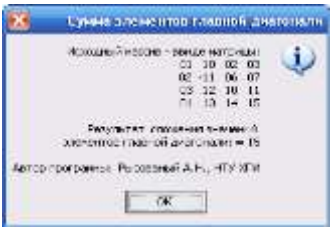


Рис. 16.1. Результат виконання програми з лістингу 16.1

При перенесенні значення якогось поля в один з регістрів необхідно слідкувати за розміром цього регістра, щоб розміри поля елемента та розміри регістра збігалися.

Для визначення **розміру** структури або її полів (на прикладі попередньої програми) можна використати вирази:

```
mov eax,sizeof(Date1) ; повернення розміру структури (EAX = 9)
mov ebx,Date1.Name1 ; повернення розміру поля Name1 + 1
```

Приклад 16.2. Задано структуру, яка містить такі поля: день Day, місяць Month, рік Year. Замість числа місяця вивести скорочену назву.

Програму прикладу 16.2 наведено у лістингу 16.2.

Лістинг 16.2. Програма виконання прикладу 16.2:

```
        .686
        .model flat, stdcall
        option casemap:none
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32
DATE STRUCT                                 ; тип даних СТРУКТУРА
    Day      DWORD ?
    Month  DWORD ?
    Year    DWORD ?
DATE ENDS
        DateDisp proto :DATE
        BSIZE equ 15                ; розмір буфера
.data                               ; директива определения данных
ifmt BYTE "%d",0 ; завдання перетворення числа у символол
        buf     BYTE BSIZE dup(0)
        Date   DATE <02,11,2011>                ; змінна типу DATE
        MName   BYTE "янв фев мар апр май июн июл авг "
                BYTE "сен окт ноя дек"
.data?                                          ; невизначені структури даних
stdout DWORD ?   ; збереження дескриптора виведення
cWritten  DWORD  ? ; для адреси символів виведення
.code
  start:                                ; метка начала программы с именем start
        invoke  GetStdHandle, STD_OUTPUT_HANDLE
```

```
        mov     stdout, eax                     ; збереження дескриптора
        invoke SetConsoleOutputCP,1251  ; для відображення кирилиці
        invoke  DateDisp, Date          ; показати дату
invoke  ExitProcess, 0 ; повернення управління  ОС та визволення ресурсів
        DateDisp proc arg1:DATE
invoke wsprintf, ADDR buf, \ ; адреса буфера для зберігання символів
ADDR ifmt, arg1.Day   ; адреса форматного рядка та перетворюване число
invoke WriteConsole, stdout,  ; дескриптор стандартн. пристрою виведення
 ADDR buf, 3 ; адреса початку повідомлення та розмір (кількість байтів в ньому)
   ADDR cWritten, NULL          ; адреса ділянки пам'яті, де процедура
                                                ; зберігає число виведених на екран символів
        mov eax, arg1.Month     ; завантаження поля структури Month (eax = 0Bh)
        dec  eax                ; зменшення – тому що нумерація з нуля (0Ah = 10)
        shl  eax, 2     ; помножити на 4 (обчислення кількості байтів: 10 ? 4 = 40
        mov   esi, offset MName       ; занесення адреси масиву назв
        add    esi, eax       ; обчислення лінійної адреси поля Month
invoke  WriteConsole, stdout, esi, 4, ADDR cWritten, NULL
        xor     eax, eax
invoke  wsprintf, ADDR buf, ADDR ifmt, arg1[8]
invoke  WriteConsole, stdout, ADDR buf, 4, ADDR cWritten, NULL
        ret                                     ; возвращение из процедуры
        DateDisp endp                ; закінчення процедури з ім'ям DateDisp
        end start                               ; закінчення програми
```

Після отримання файлу з розширенням .exe та його виконання в середовищі програми FAR буде отримано надпис, який наведено на рис. 16.2.

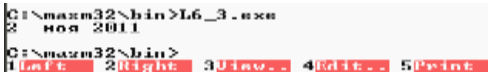


Рис. 16.2. Результат виконання програми з лістингу 16.2

16.2. Структура SYSTEMTIME для визначення системного часу

Для виведення поточних дати та часу використовується структура SYSTEMTIME, яка має параметри:

```
    Systime  SYSTEMTIME   <?>                ; системний час
; структура SYSTEMTIME у файлі windows.inc
;       SYSTEMTIME STRUCT
; wYear         WORD   ?      // поточний рік
; wMonth        WORD  ?      // поточний місяць (0 – січень, 1 – лютий  й т.д.)
; wDayOfWeek            WORD ?              //  0 – неділя, 1 – понеділок  й т. д.)
; wDay          WORD     ?          // число
; wHour         WORD     ?          // година
; wMinute               WORD     ?          // хвилини
; wSecond               WORD     ?          // секунди
;  wMilliseconds        WORD     ?          // мілісекунди
; SYSTEMTIME ENDS
```

Використання структури SYSTEMTIME для отримання поточних дати та часу наведено в лістингу 16.3.

Лістинг 16.3. Використання структури SYSTEMTIME:

```
        .686
        .model flat, stdcall
        option casemap:none
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32
.data
        stdout   dd    0                        ; дескриптор виведення
        systime   SYSTEMTIME   <?>      ; системний час
        txtDay  db 3 DUP (0),"/ " ; 2 байта - під 2 цифри, байт - під лінію
        txtMon  db 3 DUP (0),"/ "    ; комірка для збереження значення місяця
        txtYear db 6 DUP (0)          ; рік
        txtHour db 3 DUP (0), ": "   ; година
        txtMin   db 3 DUP (0), ": "   ; хвилини
        txtSec  db 3 DUP (0), 0       ; секунди  (див. c:\masm32\datetime)
        N  equ  $ - txtDay             ; довжина текстового буфера
        Rd  dd  0                        ; кількість виведених символів
.code
start:
   invoke GetStdHandle, STD_OUTPUT_HANDLE ; дескриптор виведення
        mov stdout, eax             ;
  invoke GetSystemTime, ADDR systime ; отримання поточних дати та часу
        movzx eax, systime.wDay             ; перезапис числа wDay
  invoke dwtoa, eax, ADDR txtDay   ; переклад числа в текстовий вигляд
        movzx eax, systime.wMonth         ; перезапис числа
  invoke dwtoa, eax, ADDR txtMon
        movzx eax, systime.wYear              ; перезапис числа
  invoke dwtoa, eax, ADDR txtYear
        movzx eax, systime.wHour           ; перезапис числа
  invoke dwtoa, eax, ADDR txtHour
        movzx eax, systime.wMinute       ; перезапис числа
  invoke dwtoa, eax, ADDR txtMin
        movzx eax, systime.wSecond ; перезапис числа
  invoke dwtoa, eax, ADDR txtSec
  invoke WriteConsole, stdout, ADDR txtDay, N, ADDR Rd, 0 ; виведення
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
        end start        ; директива окончания программы з іменем start
```

Результат консольного виведення наведено на рис. 16.3.

Рис. 16.3. Виведення системних дати та часу

Фрагмент програми виведення **системного часу** у спрощене віконце може бути й таким (див. **c:\masm32\datetime**):

```
        dateformat   db  "dd MMM yyyy", 0
        timeformat  db  "HH:mm:ss", 0
        titl   db "Системное время", 0
        date_buf  db  40 dup (?)
;       time_buf   db  20 dup (?)
        sys_time  SYSTEMTIME  <>
.const
        LOCALE_SYSTEM_DEFAULT  equ  0
.code
start:
        invoke  GetLocalTime, addr sys_time ;
```

```
 invoke GetDateFormat, LOCALE_SYSTEM_DEFAULT, NULL, \
   ADDR sys_time, ADDR dateformat, ADDR date_buf, 40
        mov  ecx, offset date_buf
        add  ecx, eax    ; додавання довжини до GetDateFormat
        dec  ecx
        mov  byte ptr [ecx], ' '  ;
        inc  ecx
invoke  GetTimeFormat, LOCALE_SYSTEM_DEFAULT, NULL, \
            ADDR sys_time, ADDR timeformat, ecx, 20
invoke MessageBox, NULL, addr date_buf, addr titl, MB_ICONASTERISK
```

Результат наведеного фрагмента наведено на рис. 16.4.

Рис. 16.4. Приклад виведення системних дати та часу

Фрагмент програми виведення поточного **часу** у спрощене віконце може бути й таким:

```
formats db 'Год: %d', 13,10, 'Месяц: %d', 13,10, 'День: %d',0
   result db 256 dup(?)
   systime SYSTEMTIME <>
.code
   start:
fn GetLocalTime, ADDR systime; возвращает текущие дату и время в формате UTC
      movzx ebx,  systime.wDay
      push ebx
      movzx ebx,  systime.wMonth
      push ebx
      movzx ebx, systime.wYear
      push ebx
      pop eax
fn wsprintf, addr result, addr formats, eax
fn MessageBox, 0, addr result, chr$("Дата"), MB_ICONASTERISK
      fn ExitProcess, 0
```

Результат наведеного фрагмента наведено на рис. 16.5.

Рис. 16.5. Приклад виведення системних дати та часу

16.3. Структура MSGBOXPARAMS у функції MessageBoxIndirect

Функція MessageBoxIndirect створює, показує на екрані та використовує вікно повідомлення. Вікно повідомлення містить визначуваний програмою текст повідомлення та заголовок, будь-який значок і будь-яку комбінацію зумовлених командних кнопок.

Функція MessageBoxIndirect має на мові C синтаксис

int MessageBoxIndirect(const LPMSGBOXPARAMS *lpMsgBoxParams*);

а на мові асемблера синтаксис

invoke MessageBoxIndirect, ADDR params

Ця функція має один параметр *lpMsgBoxParams,* який містить вказівник (адресу) на структуру MSGBOXPARAMS. Значення, які повертаються функцією, наведені в табл. 16.1.

Таблиця 16.1

Значення	Призначення
IDABORT	Була вибрана кнопка Прекратить (Abort)
IDCANCEL	Була вибрана кнопка Отменить (Cancel)
IDCONTINUE	Була вибрана кнопка Продолжить (Continue)
IDIGNORE	Була вибрана кнопка Пропустить (Ignore)
IDNO	Була вибрана кнопка Нет (No)
IDOK	Була вибрана кнопка OK
IDRETRY	Була вибрана кнопка Повторить (Retry)
IDTRYAGAIN	Була вибрана кнопка Попытаться снова (Try Again)
IDYES	Була вибрана кнопка Да (Yes)

Структура MSGBOXPARAMS має синтаксис

```
typedef struct {
   UINT  cbSize;
   HWND  hwndOwner;
   HINSTANCE  hInstance;
   LPCTSTR  lpszText;
   LPCTSTR  lpszCaption;
   DWORD  dwStyle;
   LPCTSTR  lpszIcon;
   DWORD_PTR  dwContextHelpId;
   MSGBOXCALLBACK  lpfnMsgBoxCallback;
   DWORD  dwLanguageId;
      } MSGBOXPARAMS, *PMSGBOXPARAMS;
```

Параметр *cbSize* встановлює розмір структури в байтах.

Параметр *hwndOwner* визначає дескриптор вікна власника.

Параметр *hInstance* визначає дескриптор модуля, який містить ресурс значка, ідентифікований членом структури *lpszIcon* та рядковий ресурс, ідентифікований членом структури *lpszText* або *lpszCaption*.

Параметр *lpszText* визначає вказівник на символьний рядок з нулем в кінці або ідентифікатор рядкового ресурсу, який містить повідомлення, що з'являється на екрані.

Параметр *lpszCaption* визначає вказівник на символьний рядок з нулем в кінці або ідентифікатор рядкового ресурсу, який містить заголовок вікна повідомлення. Якщо цей член структури NULL, то використовується заданий за умовчанням заголовок (Error (Помилка)).

Параметр *dwStyle* встановлює вміст і характер роботи діалогового вікна. Цей член структури може бути комбінацією прапорців, описаних для параметра *utype* функції MessageBoxEx. Можна встановити прапорець

MB_USERICON, якщо необхідно, щоб вікно повідомлення показало на екрані значок, визначений членом структури *lpszIcon.*

Параметр *lpszIcon* ідентифікує ресурс значка. Цим параметром може бути або символьний рядок з нулем в кінці, або цілочисловий ідентифікатор ресурсу, який переданий в макрокоманду MAKEINTRESOURCE. Щоб завантажити один із стандартних, визначуваних системою значків, треба встановити член структури *hInstance* у NULL а параметр *lpszIcon* у одне із значень, які перераховуються у функції LoadIcon. Цей член структури ігнорується, якщо член структури *dwStyle* не встановлює прапорець MB_USERICON.

Параметр *dwContextHelpId* ідентифікує контекст довідки. Якщо відбувається подія довідки, то це значення визначається в структурі HELPINFO, яке вікно повідомлення відправляє у вікно власника, або функцію повторного виклику.

Параметр *lpfnMsgBoxCallback* визначає вказівник на функцію повторного виклику, яка обробляє події довідки для вікна повідомлення. Функція повторного виклику має один параметр і таку форму:

VOID CALLBACK MsgBoxCallback(LPHELPINFO lpHelpInfo);

Якщо член структури *lpHelpInfo* NULL, то вікно повідомлення відправляє повідомлення WM_HELP у вікно власника (коли відбуваються події довідки).

Параметр *dwlanguageid* встановлює мову, на якій наводиться на екрані текст, що міститься в зумовлених командних кнопках. Це значення повинне знаходитися у формі, поверненій макросом MAKELANGID. Відповідно до цього американська версія пропонує, наприклад, використовувати параметр LANG_ENGLISH, французька версія – LANG_FRENCH, німецька версія – LANG_GERMAN, а японська версія – LANG_JAPANESE. Кожна версія пропонує параметр LANG_NEUTRAL.

У лістингу 16.4 наведено програму найпростішого використання функції MessageBoxIndirect. Ця програма з нульовими параметрами виводить повідомлення аналогічно функції MessageBox.

Лістинг 16.4. Найпростіше використання функції MessageBoxIndirect:

```
        .686
        .model flat, stdcall
        option casemap:none
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32
.data                                   ; директива определения данных
    TheMsg      db "Message",0          ; повідомлення
    Caption     db "Caption",0      ; заголовок вікна повідомлення
    params MSGBOXPARAMS <>
.code                                   ; директива начала  сегмента данных
start:                                  ; метка начала программы
      mov params.cbSize, SIZEOF MSGBOXPARAMS   ; розмір структури
      mov params.hwndOwner, 0                  ; дескриптор вікна власника
      invoke GetModuleHandle, 0          ; отримання дескриптора програми
```

```
    mov params.hInstance, eax          ; збереження дескриптора програми
    mov params.lpszText, offset TheMsg          ; адреса повідомлення
    mov params.lpszCaption, offset Caption      ; адреса заголовку вікна
    mov params.dwStyle, MB_ICONINFORMATION   ; стиль вікна
    mov params.lpszIcon, 0                      ; ресурс значка
    mov params.dwContextHelpId, 0               ; контекст довідки
    mov params.lpfnMsgBoxCallback, 0 ;
    mov params.dwLanguageId, LANG_NEUTRAL    ; мова повідомлення
    invoke MessageBoxIndirect, ADDR params ;
invoke ExitProcess, 0  ; повернення управління  ОС та визволення ресурсів
  end start                             ; директива окончания программы
```

Результат використання функції MessageBoxIndirect в програмі з лістингу 16.4 наведено на рис. 16.6.

Рис. 16.6. Результат використання функції MessageBoxIndirect

Змінити значок відображення програми у файловому менеджері, наприклад в Total Commander, можна, якщо додати файл ресурсів з розширенням .rc та вмістом:

```
1 ICON DISCARDABLE "disc.ico"
```

Файл disc.ico із зображенням іконки розміщується у тому ж місці, де й файл .asm.

Крім того, командний .bat-файл, наприклад, повинен мати такий вміст:

```
ml /c /coff "8_2MesB.asm"
rc "8_2MesB.rc"
link /SUBSYSTEM:windows "8_2MesB.obj"  "8_2MesB.res"
 del  *.obj
 del  *.res
  pause
start 8_2MesB.exe
```

Для відображення **власної іконки** на спрощеному віконці необхідно виконати таке:

– додати у файл ресурсів рядок, наприклад #define IDI_ICON 1001;
– додати у програму рядок IDI_ICON EQU 1001;
– змінити параметр dwStyle на MB_USERICON;
– змінити параметр lpszIcon на IDI_ICON.

У результаті виконаних дій програма відображення власної іконки буде такою, як наведено в лістингу 16.5.

Лістинг 16.5. Програма відображення власної іконки:

```
    .686                          ; директива определения типа микропроцессора
  .model  flat, stdcall  ; задание линейной модели памяти
  option casemap:none                ; відмінність малих та великихлітер
    include \masm32\include\windows.inc ; файлы структур, констант …
    include \masm32\include\\user32.inc   ; файлы  интерфейса …
    include \masm32\include\kernel32.inc ; системные функции применений…
```

```
      includelib \masm32\lib\user32.lib
      includelib \masm32\lib\kernel32.lib
IDI_ICON  EQU  1001
 .data                                  ; директива определения данных
     TheMsg     db "Message",0              ; повідомлення
     Caption    db "Caption",0          ; заголовок вікна повідомлення
     params MSGBOXPARAMS <>
.code
start:                                      ; метка начала программы
      mov params.cbSize, SIZEOF MSGBOXPARAMS    ; розмір структури
      mov params.hwndOwner, 0                   ; дескриптор вікна власника
      invoke GetModuleHandle, 0          ; отримання дескриптора програми
      mov params.hInstance, eax          ; збереження дескриптора програми
      mov params.lpszText, offset TheMsg            ; адреса повідомлення
      mov params.lpszCaption, offset Caption        ; адреса заголовка вікна
      mov params.dwStyle, MB_USERICON               ; стиль вікна
      mov params.lpszIcon, IDI_ICON                 ; ресурс значка
      mov params.dwContextHelpId, 0                 ; контекст довідки
      mov params.lpfnMsgBoxCallback, 0 ;
      mov params.dwLanguageId, LANG_NEUTRAL   ; мова повідомлення
   invoke MessageBoxIndirect, ADDR params ;
invoke ExitProcess, 0 ; повернення управління   ОС та визволення ресурсів
      end start                                 ; директива окончания программы
```

Лістинг 16.6. Файл ресурсів для програми з лістингу 16.5:

```
#define IDI_ICON 1001
IDI_ICON  ICON  DISCARDABLE  "Burning ROM.ico"
```

Результат використання функції MessageBoxIndirect з відображенням власної іконки у програмі з лістингу 16.5 наведено на рис. 16.7.

Рис. 16.7. Результат використання функції MessageBoxIndirect з відображенням власної іконки

16.4. Складні структури

У випадку, якщо необхідно обробляти 64-розрядні дані на 32-розрядних регістрах можна використовувати додаткову структуру. У цій додатковій структурі необхідно вказати старше 32-розрядне число і молодше 32-розрядне число. І вже потім обрабляти частини цього числа, наприклад:

```
      .data
      DWORDDWORD  STRUCT
        lowDWORD    DWORD ?
        highDWORD   DWORD ?
      DWORDDWORD   ENDS
      …
      mov eax,memstat.ullTotalPhys.DWORDDWORD.lowDWORD;
      mov edx,memstat.ullTotalPhys.DWORDDWORD.highDWORD;
         mov ecx,1024
         div ecx
         mov [esi],eax
```

У випадку, коли полів структури багато, то можно використати вкладення структур, наприклад

str1 rr1 <,<>>

16.5. Створення зображення іконки

Для створення зображення іконки можна скористатися програмою IcoFX, зовнішній вигляд вікон яких наведено на рис. 16.8.

Рис. 16.8. Зовнішній вигляд вікон програми IcoFX

Після завантаження цієї програми необхідно виконати такі дії:

– створити зображення ікони: Файл/Создать/Размеры/48x 48;

– у вікні, яке з'явилося, натиснути на піктограму і вказати шлях імпортування файлу з розширенням .jpg;

– у вікні Создать изображение вибрати розмір 256 × 256;

– видалити раніше створену ікону з розміром 48 × 48 і порожнім зображенням. Для цього необхідно підвести курсор до порожнього вікна зображення ікони, виділити це вікно лівою клавішею мишки і після натиснення правої клавіші – видалити це вікно;

– вказати розмір ікони: Изображение/Размеры/48 × 48;

– зберегти ікону: Файл/ Сохранить как.

16.6. Лабораторна робота "Структури"

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм з використанням структур та API-функцій під Win32, написаних мовою асемблера МП платформи x86.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати свій варіант завдання та написати на асемблері програму обчислення одного з виразів з виведенням свого завдання, отриманих даних та довідку про автора програми функцією MessageBoxIndirect з відображенням **своєї фотографії** в якості іконки.

Завдання 1

1. Задано матрицю 3×5. Виконати транспонування цієї матриці. Результат виконання програми вивести у вікно консолі.

2. Задано матрицю 6×6. Визначити суму елементів під головною діагоналлю. Результат виконання програми вивести у вікно консолі.

3. Задано матрицю 3×4. Визначити рядок з максимальною сумою позитивних елементів. Результат виконання програми вивести у вікно консолі.

4. Задано матрицю 4×5. Визначити мінімальний елемент кожного стовпця. Результат виконання програми вивести у вікно консолі.

5. Задано матрицю 5×7. Визначити максимальний елемент кожного парного рядка. Результат виконання програми вивести у вікно консолі.

6. Задано матрицю 4×6. Визначити суму елементів кожного стовпця. Результат виконання програми вивести у вікно консолі.

7. Задано матрицю 3×6. Визначити рядок з максимальною сумою елементів. Результат виконання програми вивести у вікно консолі.

8. Задано матрицю 3×5. Визначити рядок з мінімальною сумою елементів. Результат виконання програми вивести у вікно консолі.

9. Задано матрицю 3×6. Визначити елементи, кратні 3, в кожному рядку і помістити на їх місце елемент, номер якого збігається з номером рядка. Результат виконання програми вивести у вікно консолі.

10. Задано матрицю 4×6. Визначити суму негативних елементів кожного рядка і помістити її на місце першого елемента. Результат виконання програми вивести у вікно консолі.

Завдання 2

1. Задано послідовність структур. Структура містить поля: назва автомобіля, порядковий номер, ім'я власника, кількість порушень. Обчислити кількість власників, у яких більше трьох порушень.

2. Задано послідовність структур. Структура містить поля: ім'я студента, стипендія, середня оцінка, вік. Обчислити середній вік студентів.

3. Задано послідовність структур. Структура містить поля кредитної картки: номер, прізвище власника, розмір грошової суми і ступінь захисту. Обчислити кількість кредитних карток з грошовою сумою більше $300 і ступенем захисту більше 10.

4. Задано послідовність структур. Структура містить поля (відповідно до комп'ютера): серійний номер, ціна, назва, прізвище власника, розмір монітора в дюймах. Обчислити середню ціну 26-дюймових моніторів.

5. Задано послідовність структур. Структура містить поля за характеристиками студентських груп: назва групи, успішність (середня цілочислова оцінка), номер курсу. Вивести на екран назву групи з максимальною успішністю.

6. Задано послідовність структур. Структура містить поля з торгової діяльності комп'ютерної фірми: назва товару (комп'ютер, модем, гвинт), ціна (у доларах), кількість проданих одиниць. Обчислити прибуток фірми (підсумовування "кількості проданих одиниць помножену на ціну").

7. Задано послідовність структур. Структура містить поля з торгової діяльності комп'ютерної фірми: назва товару (комп'ютер, модем, гвинт), ціна, кількість проданих одиниць. Обчислити прибуток по кожному товару (прибуток = кількість проданих одиниць, помножена на ціну).

8. Задано послідовність структур. Структура містить поля з зарплати викладачів: прізвище викладача, назва курсу (який веде викладач), посада, зарплата. Обчислити прізвище самого низькооплачуваного викладача.

9. Задано послідовність структур. Структура містить поля про дані податкової інспекції: назва фірми, прибуток, розмір оплачуваних податків, кількість штрафів. Обчислити кількість фірм, дохід яких перевищує середній дохід фірм.

10. Задано послідовність структур. Структура містить поля: назва автомобіля, порядковий номер, ім'я власника, кількість порушень. Вивести на екран ім'я власника з максимальною кількостю порушень.

11. Задано послідовність структур. Структура містить поля: назва автомобіля, порядковий номер, ім'я власника, кількість порушень. Вивести на екран ім'я власника з мінімальним числом порушень.

12. Задано послідовність структур. Структура містить поля: ім'я студента, стипендія, середня оцінка, вік. Обчислити середню оцінку студентів.

13. Задано послідовність структур. Структура містить поля: назва автомобіля, порядковий номер, ціна автомобіля, рік випуску, розмір податків. Обчислити кількість автомобілів з останнім (максимальним) роком випуску.

14. Задано послідовність структур. Структура містить поля даних про комп'ютер: серійний номер, ціна, назва, прізвище власника, розмір монітора в дюймах. Обчислити середню ціну комп'ютера.

15. Задано послідовність структур. Структура містить поля за характеристиці студентських груп: назва групи, успішність (середня цілочислова оцінка), номер курсу. Вивести на екран назву групи з мінімальною успішністю.

16. Задано послідовність структур. Структура містить поля за торговою діяльності комп'ютерної фірми: назва товару (комп'ютер, модем, гвинт), ціна (у доларах), кількість проданих одиниць. Обчислити загальну кількість проданих одиниць.

17. Задано послідовність структур. Структура містить поля з зарплати викладачів: прізвище викладача, назва курсу (який веде викладач), посада, зарплата, адреса. Вивести на екран адресу максимально оплачуваного викладача.

18. Задано послідовність структур. Структура містить поля з зарплати викладачів: прізвище викладача, назва курсу (який веде викладач), посада, зарплата, адреса. Обчислити середню, мінімальну і максимальну зарплату викладачів.

19. Задано послідовність структур. Структура містить поля за даними податкової інспекції: назва фірми, прибуток, розмір оплачуваних податків, кількість штрафів. Вивести на екран назву фірми з максимальною кількістю штрафів.

20. Задано послідовність структур. Структура містить поля за даними податкової інспекції: назва фірми, прибуток, розмір оплачуваних податків, кількість штрафів. Вивести на екран назву фірми з максимальним прибутком.

Зміст звіту

1. Постановка задачі для конкретного варіанта.
2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.
3. Лістинг програми та коментарі до всіх команд.
4. Print screen екрана 32-розрядного налагоджувача з виконанням програми та результатами виконання.
5. Короткий опис виконання програми.
6. Висновки за результатами роботи.

Приклад 16.3. Задано матрицю 2 × 4. Визначити суму всіх негативних елементів матриці.

Лістинг 16.6. Програма виконання прикладу 16.3:

```
title CopyRight by Rysovaniy A. N., rysov@rambler.ru
.686                        ; директива определения типа микропроцессора
.model  flat, stdcall  ; завдання лінійної моделі пам'яті  та угоди ОС Windows
option casemap:none        ; отличие строчных и прописных букв
include    \masm32\include\windows.inc ; файлы структур, констант …
include    \masm32\include\\user32.inc   ; файлы  интерфейса …
include \masm32\include\kernel32.inc ; системные функции применений…
includelib \masm32\lib\user32.lib
includelib \masm32\lib\kernel32.lib
   DATE1 STRUCT       ; тип даних СТРУКТУРА з іменем DATE1
   elem1 dd ?         ; ім'я першого поля структури
   elem2 dd ?         ; ім'я другого поля структури
   elem3 dd ?         ; ім'я третього поля структури
   elem4 dd ?         ; ім'я четвертого поля структури
   DATE1 ENDS
.data                                   ; директива определения данных
    str1 DATE1 <1,-1,-2,3>                  ; структура з іменем str1
    str2 DATE1 <0,-2,-1,-3>                 ; структура з іменем str2
   titl1 db " Работа с элементами структуры",0
   buf db 10 dup(?),0
ifmt db "Задана матрица:",0dh,0ah,\
 "1   -1   -2   3",0dh,0ah,\
 "0   -2   -1  -3",0dh,0ah,0ah,\
  "Результат сложения отрицательных элементов",0dh,0ah,\
   "   = %d                  ",0dh,0ah,0ah, "Автор программы:              НТУ ХПИ",0
```

```
.code                                   ; директива начала  сегмента данных
 start:                                 ; метка начала программы с именем start
        xor edx,edx                             ; заповнювання нулями
        mov ebx,2                               ; завантаження кількості рядків
        lea esi,str1        ; завантаження адреси першого рядка структури
   m5: mov ecx,4                ; кількість елементів у рядку
   m3: mov eax,[esi]          ; завантаження елемента з рядка структури
        add eax,0               ; визначення ознак елемента
       js m1                      ; перейти на m1, якщо елемент негативний
       jmp m2                     ; безумовний перехід, якщо навпаки
   m1: add edx,eax           ; додавання негативних елементів рядка структури
   m2: add esi,4             ; підготовка адреси нового елемента
        loop m3                 ; ecx := ecx – 1 та перехід на m3, якщо не нуль
        dec ebx                    ; ebx := ebx – 1
       jz m4                      ; якщо ebx = 0 (z = 1), то перехід на закінчення
        lea esi,str2              ; завантаження адреси нового рядка
       jmp m5                     ; перехід на новий цикл
   m4:
invoke  wsprintf, ADDR buf, \; адреса буфера, куди буде записана послідовність символів
  ADDR ifmt, edx ; адреса рядка перетворення формату та регістр перетворення
invoke MessageBox, 0,  addr buf, \ ; адреса рядка, яка містить текст повідомлення
 addr titl1, MB_ICONINFORMATION+180000h ; заголовок та вигляд вікна
invoke ExitProcess, 0  ; повернення управління  ОС та вивільнення ресурсів
        end start               ; директива окончания программы з іменем start
```

У результаті виконання програми отримується число –9, яке заноситься до регістра edx, а потім виводиться у вікно консолі.

Результат виконання програми наведено на рис. 16.9.

Рис. 16.9. Вікно з результатом виконання програми

Приклад 16.4. Задано послідовність структур. Структура містить поля: ім'я студента, стипендія, середня оцінка, вік. Обчислити середній вік студентів**.** Результат виконання програми вивести у спрощені вікна.

Лістинг 16.7:

```
    Title Вернидуб А. В., НТУ "ХПІ", КІТ-29б
    .686                            ;директива определения типа микропроцессора
    .model flat,stdcall  ; завдання лінійної моделі пам`яті та угоди ОС Windows
    option casemap:none      ; отличие строчных и прописных букв
    include C:\masm32\include\windows.inc ; файлы структур, констант …
    include C:\masm32\include\kernel32.inc ; файли систем. функцій застосувань…
    include C:\masm32\include\\user32.inc   ; файлы  интерфейса …
    includelib C:\masm32\lib\kernel32.lib
    includelib C:\masm32\lib\user32.lib
  array STRUCT              ; Тип даних структура
    _suc      db   ?          ; поле структури
    _course    db   ?         ; поле структури
    _name db 10 dup(?)     ; поле структури
    _step      db   ?
       array ENDS       ; Закінчення даних СТРУКТУРА
```

```
.data
        max db 0
        sum dd ?
        first array  < 21, 5, "Иванов",70>     ; структура з іменем first
        second array < 35, 5, "Петров",50>; структура з іменем second
        third array  < 21, 5, "Сидоров",30>     ; структура з іменем third
        fourth array < 35, 5, "Киселёв",60>     ; структура з іменем fourth
        date  db "Ім'я          Степендія   Середній бал       Вік      ", 0ah, 0dh,
              "Иванов  К.В.       70          4.7           21 ", 0ah, 0dh,
              "Петров  М.В.       50          5             35", 0ah, 0dh,
              "Сидоров Р.Д.       30          3.2           15 ", 0ah, 0dh,
              "Киселёв Е.З.       60          4             35",0ah,0dh,0
        mas dd 8 dup(?),0
        st1 db "Обчислити середній вік студентів в групі", 0
        ifmt db "Середній вік студентів: %d ",0
.code
Begin:
        invoke MessageBox, NULL, addr  date, addr st1, MB_ICONQUESTION
        xor edx, edx
        mov ecx, 4
        lea esi, first
next:
        mov eax, [esi]
        add max,al
        add esi,13
      loop next
        movsx eax,max
        mov ebx,4
        xor edx, edx
        div ebx
        mov sum,eax
        invoke  wsprintf,ADDR mas, ADDR ifmt, sum
        invoke MessageBox, NULL, addr  mas, addr st1, MB_ICONINFORMATION
        invoke ExitProcess, 0
        end Begin
```

Результат виконання програми наведено на рис. 16.10.

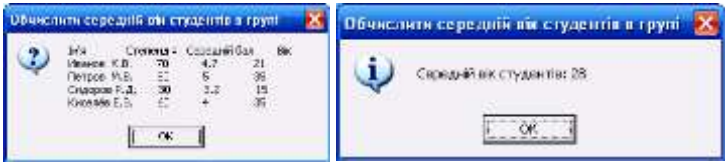


Рис. 16.10. Результат виконання програми

Приклад 16.4. Задано масив A из N = 7 элементов. Структура массива A: X1,Y1; X2,Y2; Определить количество пар, для которых используется условие Xi <= Yi.

Лістинг 16.8. Виконання прикладу 16.4:

```
.686
.model flat, stdcall
option casemap:none
include    \masm32\include\windows.inc
include    \masm32\include\kernel32.inc
include    \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
```

```
        includelib \masm32\lib\user32.lib
BSIZE  equ 13
array struct
    _Xi dd ?
  _Yi dd ?
array ends

.data
   szBuf   db BSIZE dup(?)
   str3    db "Задание:",0
   str4    db "Задано масив A из N = 7 элементов. Структура массива A: X1,Y1; X2,Y2;",
   " Определить количество пар, для которых используется условие Xi <= Yi",0
   fmt     db "result = %d",0
   caption db "Lab",0
A array  <5,5>,<3,5>,<7,2>,<8,4>,<3,8>,<4,2>,<2,3>
   var_res dd ?
.code
_start:
   xor esi, esi       ; Очистка регистра
  mov ecx, 8          ; Размер массива
  .while ecx != 0     ; Цикл проход по массиву
     mov eax, A[edi]._Xi ; Извлечение поля структуры в регистр
     mov edx, A[edi]._Yi ; Извлечение поля структуры в регистр
          .if (eax <= edx)    ; Сравнение полей структуры
          inc esi             ; Увеличение счетчика пар
          .endif              ; Конец условия
     inc edi             ; Увеличение индекса массива
     dec ecx              ; Уменьшение счетчика
  .endw
  invoke wsprintf, ADDR szBuf, ADDR fmt, esi
  invoke MessageBox, NULL, addr str4, addr str3, MB_ICONINFORMATION
  invoke MessageBox, NULL, addr szBuf, addr caption, MB_ICONINFORMATION
  invoke ExitProcess,0
end _start
```

Контрольні запитання

1. Що являє собою структура?
2. Які дії необхідно виконати для використання структур в програмі?
3. Які переваги та недоліки структур?
4. Як обробляти 64-розрядні числа структур?

17. АНТИНАЛАГОДЖУВАЛЬНІ ПРИЙОМИ

Антиналагоджувальних прийомів існує велика кількість. Деяку кількість можна подивитись на сайті wasm.ru. Розглянемо декілька з них.

17.1. Трасування в x86-процесорах

Трасування – це покрокове виконання налагоджуваної програми. Якщо TF-прапорець, що зберігається в регістрі EFLSGS зведений, то процесор перед кожною інструкцією генерує виключення int 1 або EXCEPTION_SINGLE_STEP (80000004h) – як його «назвали» розробники Windows, скидає TF та припиняє потік. Налагоджувач функцією WaitForDebugEvent повертає параметр EXCEPTION_DEBUG_EVENT + EXCEPTION_SINGLE_STEP. Причому налагоджувач встановлює біт TF EFLAGS за допомогою функції SetThreadContext, яка дозволяє в будь-якому потоці змінити будь-який з регістрів.

Виняток становлять команди, які модифікують регістр SS (селектор стека), які маскують переривання на виконання наступної команди. На цей крок розробники процесорів пішли тому, що в коді часто зустрічаються конструкції вигляду MOV SS, new_ss/MOV ESP, new_ESP. Це зроблено для того, щоб переривання не відбулося раніше ініціалізації вершини стека (http://www.xakep.ru/magazine/xa/112/080/1.asp).

Найпростіший спосіб виявлення трасування полягає в читанні регістра прапорців (EFLAGS) і перевірці стану біта TF. Якщо він не дорівнює нулю – програму трасують, лістинг 17.1.

Лістинг 17.1: програма для перевірки біта TF:

```
    .686p    ; директива определения типа микропроцессора
    .model  flat, stdcall ; задание линейной модели памяти
    option casemap:none    ; відмінність малих та великих  літер
    include \masm32\include\windows.inc ; файлы структур, констант …
    include \masm32\macros\macros.asm
    uselib user32, kernel32
.data
       szTest db "Программу НЕ трассируют",0
       szTest2 db "Программу трассируют",0
.code
start:
        pushfd              ; зберігаємо прапорці в стеку, включаючи і TF
        pop eax             ; виштовхуємо збережені прапорці в eax
        and eax, 100h   ; перевіряємо стан TF-біта
        jnz under_debugger   ; якщо TF зведений, то програму  трасують
       invoke MessageBoxA,FUNC(GetForegroundWindow), addr szTest,0,0
       jmp exit1
under_debugger:
       invoke MessageBoxA,FUNC(GetForegroundWindow), addr szTest2,0,0
exit1:
   invoke ExitProcess,0
        end start
```

Для того, щоб обійти цей фрагмент досить припинити трасування і встановити точку зупинку за її кінцем. Після чого ввести у налагоджувач команду <Run> або <Go>. Тоді фрагмент коду виконається без трасування. Ще один шлях – замінити команду and eax,100h на and eax,0h і тоді програма втратить здатність розпізнавати відладчик.

Функція GetForegroundWindow повертає дескриптор пріоритетного вікна (вікна, з яким користувач в даний час працює). Система привласнює трохи більш високий пріоритет потоку, який створює пріоритетне вікно, ніж той, який вона дає іншим потокам.

Налагоджувач OllyDbg відстежує команду PUSHFD, емулює її виконання, «вичищаючи» TF-біт із стека і тому трасування не виявляє.

Дуже часто програмісти вважають за краще зберігати регістр прапорців в одному місці програми, а перевіряти його в іншому.

Краще, звичайно, додати перевірку цілісності програми.

Вміст регістра прапорців можна прочитати:

– командою PUSHFD, що заштовхує прапорці в стек;

– генерацією виключення (при якій SEH-обработчику передається контекст потоку разом зі всіма регістрами, включаючи регістр прапорців);

– отриманням контексту API- функцією GetThreadContext.

17.2. Переривання в масці

При додаванні перед інструкцією PUSHFD команд MOV AX,SS/MOV SS,AX налагоджувач OllyDbg поводиться неоднозначно (http://www.xakep.ru/magazine/xa/112/080/1.asp). І хоча реальної модифікації регістра SS при цьому не відбувається, мікропроцесор все одно маскує переривання трасування на час команди, наступної за MOV SS,AX, якою і є PUSHFD (лістинг 17.2).

Лістинг 17.2:

```
        .686    ; директива определения типа микропроцессора
        .model  flat, stdcall  ; задание линейной модели памяти
        option casemap:none    ; відмінність малих та великих  літер
      include \masm32\include\windows.inc ; файлы структур, констант …
      include \masm32\macros\macros.asm
      uselib user32, kernel32
.data
        szTitl db "Исследование трассировки",0
        szTest db 'Программа выполнена не под отладчиком',0
        szTest2 db 'Отладчик обнаружен',0
.code     ;
_start:    ;
         mov ax,ss ; маскуємо переривання трасування...
         mov ss,ax ; ... на час виконання команди PUSHFD
         pushfd ; зберігаємо прапори в стеку, включаючи  і TF
         pop eax ; виштовхуємо збережені прапорці в eax
         and eax,100h ; перевіряємо стан TF-біта
         jnz under_debugger   ; якщо TF зведений, нас трасують
```

```
        nop
        nop
        nop
        nop
        nop
        nop
invoke MessageBoxA,FUNC(GetForegroundWindow), addr szTest,addr szTitl,0
        jmp @end
        under_debugger:
        invoke MessageBox, 0, offset szTest2, addr szTitl, MB_OK
        @end: invoke ExitProcess,0
        end _start
```

Якщо за командою mov ss,ax поставити команду, наприклад nop, то налагоджувач не виявить трасування.

17.3. Трасування за часом виконання інструкцій

Факт виконання коди під налагоджувачем приховати досить складно, це особливо відноситься до часу виконання інструкцій. Дійсно, кожна інструкція виконується мікропроцесором за певну кількість тактів. Можна вводити поправку і на різницю в частотах, і на кількість ядер мікропроцесора, і на технології оптимізації виконання коду.

Команда rdtsc (Read Time Stamp Counter) – асемблерна інструкція для платформи x86, що читає лічильник TSC (Time Stamp Counter) і повертає в регістрах EDX:EAX 64-бітну кількість тактів з моменту останнього скидання мікропроцесора. Команда rdtsc (код 0F 31) підтримується в процесорах Pentium і новіших. У багатозадачних операційних системах інструкція може бути перетворена на привілейовану (встановлений 3 біт в управляючому регістрі Cr4), і її використання приведе до генерації виключення в програмі.

Для того, щоб використовувати інструкцію RDTSC в антиналагоджувальних цілях, необхідно виконати її двічі: до і після виконання коду або частини коду, для якої проводитиметься вимір, наприклад, як показано в лістингу 17.3.

Лістинг 17.3:

```
        .686p    ; директива определения типа микропроцессора
        .model  flat, stdcall  ; задание линейной модели памяти
        option casemap:none    ; відмінність малих та великих  літер
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib user32, kernel32,comctl32
.data
          szTest  db 'Отладчик не обнаружен',0
          szTest2 db 'Отладчик обнаружен',0
.code
start:
          rdtsc
            xchg eax, ecx
          rdtsc
            sub eax, ecx
            cmp eax, 1000
         ja Not_DEBUGGED          ; якщо >
```

```
        invoke MessageBox, 0, offset szTest, 0, MB_OK
          jmp exit1
        Not_DEBUGGED:
           invoke MessageBox, 0, offset szTest2, 0, MB_OK
        exit1: invoke ExitProcess,0
          end start
```

Можна скористатися функцією GetTickCount. Функція GetTickCount витягує число мілісекунд, які закінчилися з тих пір, як система була запущена. Вона обмежується роздільною здатністю системного таймера. Повертане значення – число мілісекунд, які закінчилися після того, як система була запущена.

У лістингу 17.4 порівнюються два значення числа мілісекунд, отриманих функцією GetTickCount за певний інтервал часу і якщо різниця між ними більша 1000 мС, то виводиться повідомлення про те, що програма трасується у налагоджувачі.

Лістинг 17.4:

```
        .686p    ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти
        option casemap:none    ; відмінність малих та великих  літер
      include \masm32\include\windows.inc ; файлы структур, констант …
      include \masm32\macros\macros.asm
      uselib user32, kernel32,comctl32
.data
        szTest  db 'Отладчик не обнаружен',0
.code
start:
invoke GetTickCount
              mov ebx, eax
  ; Здесь у нас 10 nop'ов (90h = опкод nop'a)
              db 10 dup(90h)
    invoke GetTickCount
              sub eax, ebx
              cmp eax, 1000
              Ja Not_DEBUGGED         ; если >
invoke MessageBox, 0, offset szTest, 0, MB_OK
              jmp exit1
Not_DEBUGGED:
    invoke MessageBox, 0, offset szTest2, 0, MB_OK
exit1:  invoke ExitProcess,0
        end start
```

Щоб отримати дані про роздільну здатність системного таймера можна використовувати функцію GetSystemTimeAdjustment.

17.4. Обробка виключень

Структурна обробка виключень (англ. SEH – Structured Exception Handling) – механізм обробки програмних і апаратних виключень в ОС Microsoft Windows, який дозволяє програмістам контролювати обробку виключень, а також є налагоджувальним засобом.

Виключення – це подія при виконанні програми, яка приводить до її ненормальної або неправильної поведінки. Існує два види виключень: апаратні, які генеруються мікропроцесором, і програмні, такі, що генеруються операційною системою і прикладними програмами. Механізм структурної обробки виключень дозволяє обробляти їх однотипно.

Кожен потік будь-якого процесу використовує 16-розрядний селекторний регістр FS для зберігання покажчика на структуру даних Thread Information Block, що містить інформацію про цей потік. У цій структурі зберігається покажчик на останню із зв'язаного списку зареєстровану структуру _EXCEPTION_REGISTRATION_RECORD, яка включає покажчик на обробник виключення і покажчик на попередній запис _EXCEPTION_REGISTRATION_RECORD.

При винятковій ситуації (http://teamx.fatal.ru/) мікропроцесор виконує перехід (залежно від причини виключення) на обробник, визначений в одному з шлюзів дескрипторної таблиці переривань. При цьому відбувається зміна кільця захисту на нульове, перезавантаження стека з сегменту стану завдання на ядерний стек, збереження в ядерному стеку регістра прапорців EFlags, адреси інструкції, яка викликала виключення, селектори стека і покажчика на стек.

На стороні ядра обробники іменуються KiTrapXX, відповідно коду виключення (http://sasm.narod.ru/docs/pm/pm_app/app_1/app_1.htm). Ядро виконує деякі дії з диспетчеризації виключення, зокрема викликає ядерний налагоджувач (якщо він активний), готує фрейм (структуру), який містить опис виключення і передає управління назад в програму.

Виключення, що згенерувало мікропроцесором перехоплюється ядром операційної системи і після обробки повертає управління на прикладний рівень, викликаючи функцію NTDLL.DLL! KiUserCallback-Dispatcher. При покроковому трасуванні налагоджувач прикладного ядерного рівня пропускає ядерний код, відразу ж виявляючись в NTDLL.DLL!KiUserCallbackDispatcher. Тобто, при трасуванні коду XOR EAX,EAX/MOV EAX,[EAX] наступною виконуваною командою виявляється перша інструкція функції NTDLL.DLL!KiUserCallback-Dispatcher.

Таким чином, при виникненні виключення потік починає виконувати диспетчер виключень.

При вході в диспетчер виключень в стеку знаходиться структура, яка повністю описує виключення. Регістр Esp указує на структуру:

```
EXCEPTION_POINTERS struct
    ExceptionRecord  PEXCEPTION_RECORD ?
    ContextRecord    PCONTEXT   ?
EXCEPTION_POINTERS  ends
```

Перше поле містить покажчик на іншу структуру:

```
EXCEPTION_RECORD   struct
    ExceptionCode       NTSTATUS  ?       ; 0000h
    ExceptionFlags      DWORD    ?        ; 0004h
    ExceptionRecord     PVOID     ?       ; 0008h
    ExceptionAddress    PVOID     ?       ; 000Ch
    NumberParameters            DWORD   ?          ; 0010h
ExceptionInformation  DWORD  EXCEPTION_MAXIMUM_PARAMETERS dup(?) ; 0014h
```

```
EXCEPTION_RECORD ends
```

Другий параметр структури EXCEPTION_POINTERS – ContextRecord – покажчик на повний контекст потоку (зміст всіх його регістрів) на момент виникнення виключення. Завдяки цій структурі потік може повернутися на інструкцію, яка викликала виключення, повністю відновивши стан процесора.

Структура CONTEXT, що містить повний стан мікропроцесора на момент виникнення виключення:

```
CONTEXT  struct
    ContextFlags        DWORD ?         ; 0000h
;CONTEXT_DEBUG_REGISTERS
    regDr0              DWORD ?         ; 0004h
    regDr1              DWORD ?         ; 0008h
    regDr2              DWORD ?         ; 000Ch
    regDr3              DWORD ?         ; 0010h
    regDr6              DWORD ?         ; 0014h
    regDr7              DWORD ?         ; 0018h
;CONTEXT_FLOATING_POINT
    FloatSave           FLOATING_SAVE_AREA <> ; 001Ch
;CONTEXT_SEGMENTS
    regSegGs            DWORD ?         ; 008Ch
    regSegFs            DWORD ?         ; 0090h
    regSegEs            DWORD ?         ; 0094h
    regSegDs            DWORD ?         ; 0098h
;CONTEXT_INTEGER
    regEdi              DWORD ?         ; 009Ch
    regEsi              DWORD ?         ; 00A0h
    regEbx              DWORD ?         ; 00A4h
    regEdx              DWORD ?         ; 00A8h
    regEcx              DWORD ?         ; 00ACh
    regEax              DWORD ?         ; 00B0h
;CONTEXT_CONTROL
    regEbp              DWORD ?         ; 00B4h
    regEip              DWORD ?         ; 00B8h
    regSegCs            DWORD ?         ; 00BCh
    regEFlags           DWORD ?         ; 00C0h
    regEsp              DWORD ?         ; 00C4h
    regSegSs            DWORD ?         ; 00C8h
;CONTEXT_EXTENDED_REGISTERS
    ExtendedRegisters  BYTE  MAXIMUM_SUPPORTED_EXTENSION dup(?) ; 00CCh
CONTEXT ends
```

Поле ContextFlags – набір прапорців, яке визначає групи регістрів для включення в цю структуру (константи CONTEXT_XX). При входе в диспетчер исключений сохраняется полный контекст потока.

Структура EXCEPTION_RECORD містить причину виключення. Поле ExceptionCode містить код виключення. Ці коди визначені у файлі ntstatus.h.

Поле ExceptionFlags містить прапорці, наприклад, чи може виключення бути оброблено. Звичайне це поле можна ігнорувати.

Поле ExceptionAddress містить адресу інструкції, яка викликала виключення.

Поле NumberParameters – число параметрів, наступних далі. Вони містять докладнішу інформацію про виключення. Найбільш поширене виключення – це виключення по доступу до пам'яті. Воно має код STATUS_ACCESS_VIOLATION. Для цього виключення міститься додаткова інформація – адреса, до якої відбулося звернення.

З цією інформацією можна повністю ідентифікувати виключення. Практично необхідно тільки два поля структури EXCEPTION_RECORD – це адреса інструкції що викликала виключення і його код. Останнє можна витягувати із структури контекст (всі регістри).

Приклад використання структурної обробки виключень наведено в лістингу 17.5.

Лістинг 17.5:

```
        .686p    ; директива определения типа микропроцессора
        .model  flat, stdcall  ; задание линейной модели памяти
        option casemap:none    ; відмінність малих та великих  літер
      include \masm32\include\windows.inc ; файлы структур, констант …
      include \masm32\macros\macros.asm
      uselib user32, kernel32,comctl32
.data
      szTest  db 'Отладчик НЕ обнаружен',0
      szTest2 db 'Отладчик обнаружен',0
.code
start:
        ; Встановлюємо новий seh обробник
assume fs: nothing ; відміняє зв'язок між сегментними регістрами і сегментом
push offset seh_handler ; призначаємо свій обробник виключень з  kernel32
   push dword ptr fs:[0]     ; зберігаємо старий обробник
   mov fs:[0],esp            ; реєструємо новий обробник
call  DebugBreak          ; функція виклику виняткової ситуації
WeUnderDebugger:
        ; якщо DebugBreak не викликав помилок, то нас відладжують!!!
invoke MessageBox, 0, offset szTest2, 0, MB_OK
        jmp exit1       ; вихід
seh_handler:
; Якщо ми тут, то все ОК і нас не відладжують!!!
        mov esi, [esp+0ch]   ; в esi покажчик на CONTEXT
        assume esi: PTR CONTEXT
; Встановлюємо новий eip (адреса поточної машинної команди) і виходимо
        mov [esi].regEip, offset WeArentBeingDebugged
        xor eax, eax
         ret
WeArentBeingDebugged:
invoke MessageBox, 0, offset szTest, 0, MB_OK
exit1:
        invoke ExitProcess,0
          end start
```

Функція DebugBreak примушує в контрольній точці поточного процесу відбутися винятковій ситуації. Це дає можливість зухвалому потоку подати сигнал налагоджувачу, щоб він обробив виняткову ситуацію.

Щоб викликати виняткову ситуацію в контрольній крапці в іншому процесі можна використовувати функцію DebugBreakProcess.

18. MMX-РОЗШИРЕННЯ

Команди MMX – це команди **паралельної** обробки **цілочисельних** даних для використання у мультимедійних застосуваннях (розширеннях) (MultiMedia extensions – мультимедійні розширення). Вони призначені для одночасної обробки декількох елементів даних за одну інструкцію. Розширення MMX з'явилося в процесорах модифікації Pentium P54C і присутнє у всіх останніх модифікаціях цього процесора (http://www.intel.com/ design/ pentium4/ manuals/index_new.htm#aorm).

Команди MMX включено до групи команд технології під назвою SIMD (Single Instruction, Multiple Data – одна команда, багато даних). Технологія SIMD реалізована як дві взаємозв'язані технології обробки даних: MMX і SSE. Застосування асемблера в SIMD-розширеннях дозволяє писати компактний і швидкий код для критичних фрагментів коду з використанням спеціальних інструкцій процесора для SIMD-розширень.

Команди технології MMX працюють з типами даних: 64-розрядними цілочисловими даними, а також з даними, упакованими в групи загальною довжиною 64 біти. Такі дані можуть знаходитися у пам'яті або у восьми MMX-регістрах, які позначаються як **MM0 – MM7**.

Команди MMX-розширення виконуються в тому ж режимі процесора, що і команди з плаваючою точкою.

При роботі з MMX-командами використовуються регістри стека математичного співпроцесора R0 – R7. При цьому використовуються лише 64 розряди, а стекова організація, що потрібна для операцій співпроцесора, не використовується. При спільному використанні математичного співпроцесора і MMX-розширення останньою виконуваною командою MMX-розширення повинна бути команда **emms**. Ця команда забезпечує коректний перехід процесора від виконання фрагмента програмного коду з MMX-командами до обробки звичайних команд з плаваючою точкою співпроцесора.

Команди MMX-розширення забезпечують паралельну обробку до восьми байтів, слів або подвійних слів, а також підтримують роботу з такими типами даних:

- упаковані байти (packed byte) – один 64-розрядний регістр містить 8 байтів;
- упаковані слова (packed word) – один 64-розрядний регістр містить чотири 16-розрядних слова;
- упаковані подвійні слова (packed doubleword) – один 64-розрядний регістр містить два 32-розрядних слова;
- 64-розрядні слова (quadword).

Обробка даних MMX-розширення може виконуватися одним з двох способів: з використанням або циклічної арифметики (wraparound arithmetic), або арифметики з насиченням (saturation arithmetic).

Якщо команда задіює **циклічну** арифметику і результат операції виходить за двійкову розрядну сітку використовуваного типа даних, то результат отримується шляхом віднімання від максимально допустимого числа цього результату.

Якщо команда використовує арифметику з **насиченням** і результат операції перевищує максимальне уявне значення, то у вихідний операнд записується це максимальне значення (відбувається "насичення").

Першою літерою MMX-команд є літера "p".

Більшість команд мають суфікс, який визначає тип даних і використовувану арифметику:

- us (unsigned saturation) – беззнакове насичення, при якому, якщо результат операції перевищує максимальне значення, у вихідний операнд записується це максимальне значення (відбувається "насичення"). Якщо результат операції виявився меншим за нижню межу допустимого діапазону, то у вихідний операнд записується мінімально можливе значення;
- s або ss (signed saturation) – знакове насичення, при якому використовується арифметика з насиченням, дані зі знаком;
- якщо в суфіксі немає ні символу s, ні символів ss, то застосовується циклічна арифметика (wraparound). Якщо в цьому випадку результат операції виходить за двійкову розрядну сітку використовуваного типу даних, то "зайві" старші біти результату відкидаються;
- b, w, d, q – це букви, що указують тип даних. Якщо в суфіксі є дві з цих букв, то перша з них відповідає вхідному операнду, а друга – вихідному.

Дані зі знаком і без знака мають різний допустимий діапазон. Отже, якщо використовується арифметика з насиченням, то при виході результату операції за межі допустимого діапазону у вихідний операнд будуть записані різні значення залежно від типу даних. Допустимий діапазон даних наведено у табл. 16.1.

При складанні слова зі знаком 7F38h (32568) та слова 1707h (5895) результат дорівнює 7FFFh (32767). Математична сума виявилася більшою за межу 7FFFh, і відбулося "насичення". Якщо ті ж самі два значення скласти як слова без знака, то вийде 963Fh (38463). У даному випадку "насичення" не відбувається, оскільки результат менший за максимально можливий FFFFh.

MMX-команди умовно можна розділити на такі групи:

- команди передачі даних між регістрами MMX і цілочисловими регістрами та пам'яттю;
- арифметичні команди (складання, віднімання, множення і комбінація множення і складання);
- команди пакування і розпакування;
- команди порівняння на рівність або за величиною;
- логічні команди;
- команди зсуву (логічного та арифметичного);

- команди управління станом (очищення MMX – встановлення ознак пустих регістрів в слові тегів).

Таблиця 18.1 – Допустимий діапазон даних

Тип даних	Мінімальне значення	Максимальне значення
Байт зі знаком	80h (-128)	7Fh (127)
Байт без знака	00h	FFh (256)
Слово зі знаком	8000h (-32768)	7FFFh (32767)
Слово без знака	0000h	FFFFh (65535)
Подвійне слово зі знаком	80000000h (-2147483648)	7FFFFFFFh (2147483647)
Подвійне слово без знака	00000000h	FFFFFFFFh (4294967295)
Слово, збільшене учетверо зі знаком	1000 0000 0000 0000h	7FFF FFFF FFFF FFFFh
Слово, збільшене учетверо без знака	0000 0000 0000 0000h	0FFFF FFFF FFFF FFFFh

У слові тегів вільному регістру відповідає комбінація 11, решта комбінацій вказує тільки на зайнятість регістра. Після кожної операції MMX біти тегів регістра призначення очищуються. Вільні в MMX біти [79:64] регістрів FPU заповнюються одиницями, так що помилкова обробка даних MMX інструкцією FPU приведе до операції виключення.

Якщо в ланцюжок інструкцій FPU потрібно вставити інструкції MMX, після чого продовжити обчислення FPU, то перед першою інструкцією MMX доводиться зберігати контекст (стан регістрів) FPU в пам'яті, а після цих інструкцій знову завантажувати контекст.

Для визначення підтримки MMX-команд можна скористатися програмою з лістингу 18.1.

Лістинг 18.1. Тестова програма для визначення підтримки MMX-команд:

```
        .686                    ; директива определения типа микропроцессора
        .model flat                             ; лінійна модель пам'яті
        option casemap: none   ; отличие строчных и прописных букв
   .data                                        ; директива определения данных
      testMMX DB 1; збереження у 8-розрядній комірці пам'яті змінної testMMX
.code                                   ; директива начала кода программы
   st:                                  ; метка начала программы с именемst
        mov   EAX, 1
        cpuid                           ; ідентифікація мікропроцесора
       test  EDX, 800000h ; edx:= edx ∧ 80 0000h – визначення 23 розряду
        jnz    exit                     ; перейти на exit, якщо не нуль
        mov  testMMX, 0
exit:  xor   EAX, EAX                   ; заповнення нулями eax
        mov  AL, testMMX
        ret                                     ; повернення управління  ОС
        end st                  ; директива завершение программы с именемst
```

Програма з лістингу 18.1 повертає 1 у регістр AL, якщо технологія MMX підтримується процесором.

18.1. MMX-команди передачі даних

MOVD приймач, джерело	Копіювання 32-розрядного числа
MOVQ приймач, джерело	Копіювання 64-розрядного числа

Команда **movd** копіює 32-розрядне число:

– з молодших розрядів одного MMX-регістра в молодші розряди іншого регістра (старші розряди заповнюються нулями);

– із змінної в пам'яті або з цілочислового регістра в молодші 32 розряди MMX-регістра (старші розряди заповнюються нулями);

– з молодших розрядів MMX-регістра в елемент пам'яті або в цілочисловий регістр.

Команда **movq** виконує копіювання 64 бітів:

– з одного MMX-регістра в інший: movq mmx_i, mmx_j;

– з пам'яті в MMX-регістр: movq mmx_i, mem;

– з MMX-регістра в пам'ять: movq mem, mmx_j.

Серед усіх MMX-команд тільки команди movd і movq можуть мати вихідний операнд в пам'яті, а команда movd – це єдина команда, операнд якої може знаходитися в 32-розрядному регістрі процесора.

18.2. Арифметичні MMX-команди

18.2.1. Команди додавання

MMX-команди **паралельного** додавання працюють з упакованими байтами і словами зі знаком і без знака, а також з упакованими подвійними словами зі знаком. Вони можуть використовувати як циклічну арифметику, так і арифметику з насиченням.

PADDB, PADDW, PADDD приймач, джерело	Циклічне додавання байтів (**B**), слів(**W**), подвійних слів(**D**)

Команди **paddb, paddw, paddd** виконують циклічне додавання. Якщо сума виходить за межу допустимого діапазону, то, за правилами циклічної арифметики, надлишок відлічується від іншої межі діапазону (результат отримується шляхом віднімання максимально допустимого числа від цього результату). **Перенесення одиниці з одного елемента даних в інший не відбувається.** Вхідний операнд може знаходитися в MMX-регістрі або в пам'яті; вихідний операнд повинен знаходитися в MMX-регістрі.

PADDSB, , приймач, джерело	Додавання зі знаковим насиченням

Команди **paddsb, paddsw** виконують додавання із знаковим насиченням. Якщо сума виходить за граничне значення допустимого діапазону, то результатом вважається це граничне значення. Вхідний операнд може знаходитися в MMX-регістрі або в пам'яті; вихідний операнд повинен знаходитися в MMX-регістрі.

PADDUSB, PADDUSW, приймач, джерело	Додавання з беззнаковим насиченням

Команди **paddusb, paddusw** виконують додавання з беззнаковим насиченням. Якщо сума виходить за межі граничного значення з допустимого діапазону, то результатом вважається це граничне значення. Вхідний операнд може знаходитися в MMX-регістрі або в пам'яті; вихідний операнд повинен знаходитися в MMX-регістрі.

Максимальне значення для насичення байтів розміром у слово дорівнює 65 535.

Розглянемо програму переведення великих літер в малі. Різниця між числовими значеннями цих літер складає 20h.

Лістинг 18.2. Програма переведення великих літер в малі за допомогою команди циклічного паралельного додавання байтів paddb:

```
        .686            ; директива определения типа микропроцессора
        .model flat,stdcall   ; задание линейной модели памяти
        .MMX            ; директива визначення команд MMX
        option casemap:none ; отличие строчных и прописных букв
      includelib \masm32\lib\kernel32.lib
      include \masm32\include\kernel32.inc ; системные функции применений…
.data                                                   ; директива определения данных
        mas1  DB " STUDENTS OF THE UNIVERSITY"
        len  EQU $-mas1       ; len := 26 – обчислення кількості байтів повідомлення
        mas2 DB len DUP (20h)             ; резервування 26 байтів зі значенням 20h
        masOut DB len DUP(' '),0     ; масив збереження модифікованих даних
.code           ; директива начала кода программы
_st:            ; для налагоджування як самостійної програми
  _paddb_ex  proc           ; початок процедури з ім'ям _paddb_ex
        mov   EAX, len    ; завантаження кількості байтів у eax
        mov   EBX, 8              ; максимальна кількість літер в 64 розрядах
        xor   EDX, EDX                ; ініціалізація EDX перед діленням
        div   EBX         ; визначення кількості циклів по 8 байтів
        mov   ECX, EAX ; кількість циклів для паралельної обробки
        lea   ESI, mas1                ; завантаження адреси масиву mas1
        lea   EDI, masOut          ; завантаження адреси масиву masOut
        lea   EBX, mas2                ; завантаження адреси масиву mas2
m1:
        movq MM0, qword ptr [ESI]     ; занесення 8 байтів масиву mas1 в MM0
        paddb MM0, qword ptr [EBX]   ; mas1 + mas2
        movq  qword ptr [EDI], MM0    ; збереження результату в masOut
        add   ESI, 8     ; зміщення на 64 розряда адреси даних масиву mas1
        add   EDI, 8   ; зміщення на 64 розряда адреси даних масиву masOut
        add   EBX, 8  ; підготування нової адреси даних масиву mas2
        loop m1                 ; перейти на m1, якщо ecx ≠ 0
        cmp   EDX, 0          ; порівняння залишку
        jz    exit              ; перейти на exit, якщо EDX = 0 (z = 1)
        mov   ECX, EDX ; кількість циклів для послідовної обробки
```

```
m2:  mov  AL, byte ptr [ESI]
        add   AL, 20h
        mov  byte ptr [EDI], AL
        inc   ESI        ; esi := esi + 1
        inc   EDI        ; edi := edi + 1
        dec  ECX         ; ecx := ecx – 1
        jnz   m2         ; перейти на m2, якщо не нуль (ecx ≠ 0)
exit:   lea eax,masOut          ; збереження адреси масиву masOut
    _paddb_ex  endp              ; закінчення процедури
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
        end _st          ; для налагоджування як самостійної програми
```

Програма виконує паралельну обробку даних обсягом по 8 байтів. Вікно початкових даних налагоджувача перед виконанням програми наведено на рис. 18.1.

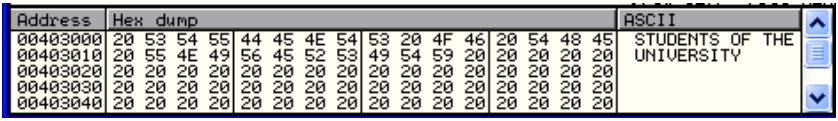


Рис. 18.1. Вікно початкових даних налагоджувача перед виконанням програми з лістингу 18.2

Обробку даних по 8 байтів виконують команди від мітки next: до команди loop m1 (рис. 18.2).

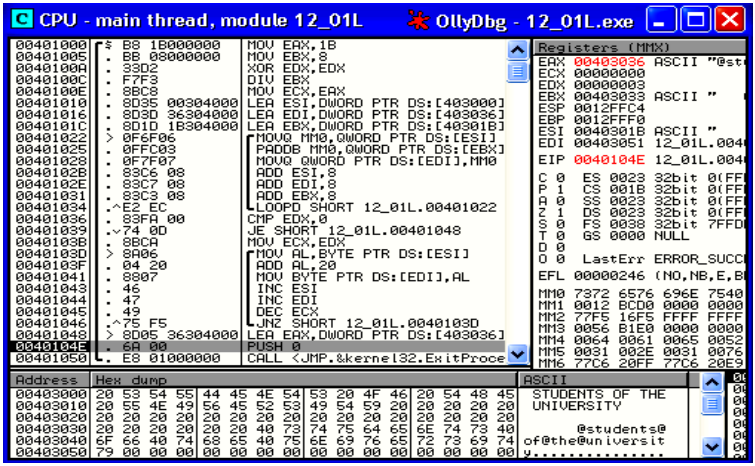


Рис. 18.2. Вікна налагоджувача при виконанні програми з лістингу 18.2

Команда cmp EDX, 0 обчислює байти, що залишились, та вже послідовно оброблює їх командами від mov ECX, EDX до мітки exit.

Модифіковані символи зберігаються в області пам'яті masOut. Масив masOut на 1 більший, ніж масиви mas2 та mas1, за рахунок нуля на кінці.

Розглянемо наступну програму. Використаємо масиви чисел розміром у **32 розряди**. Але для того, щоб подивитися, як працює арифметична команда з циклічним перенесенням, використаємо команду **paddw** додавання **16-розрядних** чисел. Крім того, використання 32-розрядних чисел при використанні API-функцій робить програму набагато простішою, тому, що навіть функція wsprintf перетворює 32-розрядні числа. Оберемо числа, які вміщаються у 16-розрядну сітку, але використаємо розмірність DD.

Розглянемо результат додавання двох масивів:

```
    1, 20000, 30000, 40000, 100, 50000, 3, 50000,   6000
+   2, 40000, 50000, 25538, 200, 40000, 1,        0, 60000
-------------------------------------------------------------
    3, 60000, 14464,        2, 300, 24464, 4, 50000,    464
```

Команда додавання слів paddw формує результат за принципом циклічного перенесення, що означає таке: якщо сума для формату слова перевищує межу максимально допустимого числа 65 536, то результат отримується як різниця між отриманим числом та цією межею. Наприклад,

30 000 + 50 000 = 80 000, 80 000 – 65 536 = 14 464.

Лістинг 18.3. Додавання двох масивів даних за допомогою команди paddw:

```
        title Рысованый А.Н.
        .686                            ; директива определения типа микропроцессора
        .model flat,stdcall  ; задание линейной модели памяти
        .MMX                                    ; директива визначення команд MMX
        option casemap:none             ; отличие строчных и прописных букв
        include \masm32\include\windows.inc    ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32
.data               ; директива определения данных
        mas1 DW 1, 20000, 30000, 40000, 100, 40000, 3, 50000, 6000
        mas2 DW 2, 40000, 50000, 25538, 200, 40000, 1, 0, 60000
        len1 EQU ($-mas2)/type mas2 ; обчислення кількості подвійних слів mas2
        sum DW len1 DUP(0),0            ; резервування байтів під змінну sum
        sum2 DD len1 DUP(0),0           ; резервування байтів під змінну sum
_title db "Результат параллельного сложения массивов с помощью команды paddw MMX",
0
        buf dw len1 DUP(0),0                ; буфер виведення повідомлення
        ifmt db "Использование команды paddw MMX.",0Ah,0Dh,0Dh,
        " Ответ =", len1 dup("   %d "),0Ah,0Dh,0Dh,
          "Автор: Рысованый А.Н., e-mail: rysov@rambler.ru",0
.code                                   ; директива начала кода программы
_st:                                    ; метка начала программы с именемst
          mov  EAX, len1          ; підготовка до визначення кількості циклів
          mov  EBX, 4 ; кількість паралельно оброблюємих чисел за один цикл
          xor  EDX, EDX             ; підготовка регістра до ділення (онулення EDX)
          div  EBX                  ; визначення кількості циклів  eax := 2 , edx := 1
          mov  ECX, EAX      ; занесення кількості циклів до лічильника
          lea   ESI, mas1           ; esi := addr mas1
          lea   EDI, mas2           ; edi := addr mas2
          lea   EBX, sum            ; ebx := addr sum
m1:
```

```
          movq  MM0, qword ptr [ESI]   ; завантаження 4-х слів масиву mas1
          paddw MM0, qword ptr [EDI]  ; паралельне циклічне додавання
          movq  qword ptr [EBX], MM0  ; збереження в новому масиві
          add   ESI, 8                ; підготовка адреси mas1 до нового зчитування
          add   EDI, 8                ; підготовка адреси mas2 до нового зчитування
          add   EBX, 8                ; підготовка адреси sum  до нового зчитування
        loop m1             ; ecx := ecx – 1 та перехід, якщо ecx != 0
          cmp   EDX, 0  ; порівняння з нулем
        jz    exit1        ; перейти, якщо всі числа оброблені (edx = 0)
          mov   ECX, EDX ; занесення остачі необроблених слів
m2:
          mov   AX, word ptr [ESI]
          add   AX, word ptr [EDI]
          mov   word ptr [EBX], AX
          add   ESI, 2
          add   EDI, 2
          add   EBX, 2
         loop m2
exit1:       ; розширення слів до подвійних слів для перетворення wsprintf
       movzx eax, sum
       mov  sum2[0],eax
          movzx eax,sum[2]
          mov  sum2[4],eax
       movzx eax,sum[4]
       mov  sum2[8],eax
         movzx eax,sum[6]
         mov  sum2[12],eax
       movzx eax,sum[8]
       mov  sum2[16],eax
         movzx eax,sum[10]
         mov  sum2[20],eax
       movzx eax,sum[12]
       mov  sum2[24],eax
         movzx eax,sum[14]
         mov  sum2[28],eax
       movzx eax,sum[16]
       mov  sum2[32],eax
 invoke  wsprintf, ADDR buf, ADDR ifmt, \
sum2,sum2[4],sum2[8],sum2[12],sum2[16],sum2[20],sum2[24],sum2[28],sum2[32]
invoke MessageBox, 0, addr buf, addr _title, MB_ICONINFORMATION
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
       end _st                                    ; завершение программы с именем_st
```

В циклі m1 виконується операція паралельного додавання чотирьох 16-розрядних чисел, а в циклі m2 – послідовне додавання чисел, які залишилися.

При виведенні результату слід звернути увагу на формат перетворення символів в числа (**ifmt db** …). Для відступу одних чисел від других в операторі **dup(" %d ")** формата перетворення в лапках додані додаткові пропуски. Два коди **0Dh,0Dh** дозволяють здійснити пропуск одного рядка для зручності виведення. А необхідні підписи слід наводити в різних рядках цього формату, розділених комами. Завершуватися цей формат повинен нулем на кінці.

Результат виконання програми наведено на рис. 18.3.

Рис. 18.3. Результат виконання програми з лістингу 18.3

18.2.2. Команди віднімання

ММХ-команди **паралельного** віднімання, як і команди додавання, працюють з упакованими байтами і словами зі знаком і без знака, а також з упакованими подвійними словами зі знаком. Вони можуть використовувати як циклічну арифметику, так і арифметику з насиченням.

PSUBB, PSUBW, PSUBD,PSUBQ приймач, джерело	Віднімання байтів (**B**), слів(**W**), подвійних слів(**D**), збільшених учетверо слів(**Q**) за правилами циклічної арифметики

Команди віднімання **psubb, psubw, psubd,psubq** застосовуються для виконання операції між операндами (байтами, словами, подвійними словами). Якщо при відніманні виникає позика, то вона ігнорується. Вхідний операнд може знаходитися в ММХ-регістрі або в пам'яті; вихідний операнд повинен знаходитися в ММХ-регістрі:

psubb/w/d/q mmx_i, mmx_j/mem64.

При циклічній арифметиці при виході результату за межі діапазону відповідне число відраховується від другої межі діапазону.

PSUBSB, PSUBSW приймач, джерело	Віднімання байтів (**B**), слів(**W**) зі знаковим насиченням

При виконанні команд **psubsb, psubsw** зі знаковим насиченням, якщо різниця виходить за межі використаної сітки, результат виконується відповідне максимальне або мінімальне число. Наприклад:

для слів $\begin{array}{r} -30000 \\ 5000 \\ \hline -32768 \end{array}$

Максимальна межа для знакових чисел розміром у слово:

FFFF/2 = 7FFF = 32 767.

Проте, з урахуванням нульового значення, остаточна межа складає 32 768.

Розглянемо програму використання команди psubb для віднімання двох масивів даних розміром у байт.

Лістинг 18.4. Програма віднімання двох масивів даних розміром у байт:

```
    .686                              ; директива определения типа микропроцессора
    .model flat                                ; завдання лінійної моделі пам'яті
    .MMX                                     ; директива визначення команд MMX
    option casemap:none             ; отличие строчных и прописных букв
  .data                                                    ; директива определения данных
       mas1  DB 5,-5, -200, 100,-150, 21,  55, 25, 5
       mas2  DB 2,-2,  200,-200, 150,-11, -79, 20, 3
       len EQU $-mas2           ; визначення кількості байтів в масиві mas2
       res DB len dup (0)
  .code                           ; директива начала кода программы
  _st:                            ; метка начала программы с именем_st
  _psub_ex proc
       mov   EAX, len                        ; eax := 9
       mov   EBX, 8
xor EDX, EDX      ; заповнення нулями EDX (додавання за модулем 2)
       div     EBX    ; визначення кількості груп по 8 байтів та залишку
       mov   ECX, EAX               ; збереження у лічильнику цілої частини
       lea    ESI, mas1             ; збереження в ESI адреси масиву mas1
       lea    EDI, mas2             ; збереження в EDI адреси масиву mas2
       lea    EBX, res              ; збереження в EBX адреси масиву res
  m1:
  movq MM0, qword ptr [ESI]      ; занесення 8 байтів mas1 в MM0
  movq MM1, qword ptr [EDI]      ; занесення 8 байтів mas2 в MM1
  psubb MM0, MM1  ; паралельне віднімання байтів та збереження в MM0
  movq qword ptr [EBX], MM0  ; збереження результату в масиві res
  add    ESI, 8         ; підготування нової адреси даних масиву mas1
  add    EDI, 8         ; підготування нової адреси даних масиву mas2
  add    EBX, 8         ; підготування нової адреси даних масиву res
    loop m1              ; ecx := ecx – 1 та перехід, якщо ecx ≠ 0
       cmp    EDX, 0                         ; порівняння залишку
       jz     exit                           ; перейти, якщо edx = 0
       mov    ECX, EDX                       ; занесення залишку в лічильник
m2:   mov    AL, byte ptr [ESI]
        sub    AL, byte ptr [EDI]
        mov    byte ptr [EBX], AL
        add    ESI, 1
        add    EDI, 1
        add    EBX, 1
       loop m2                    ; ecx := ecx – 1 та перехід, якщо ecx ≠ 0
  exit:
       lea  EAX, word ptr res     ; збереження адреси масиву res
       ret                                    ; возвращение из процедуры
  _psub_ex endp    ; закінчення процедури з ім'ям _psub_ex
       end _st                    ; завершение программы с именем_st
```

Процес отримання результату у вікнах налагоджувача наведено на рис. 18.4.

Розглянемо результат додавання двох масивів:

```
    5,  -5,  -200,   100, -150,   21,   55,  25,   5
-   2,  -2,   200,  -200,  150,  -11,  -79,  20,   3
  ------------------------------------------------------
    3, FDh,  70h,   2Ch,   D4h    20h, 86h,  5h,  2h
```

Слід ураховувати, що результати наведено у додаткових кодах. Наприклад,

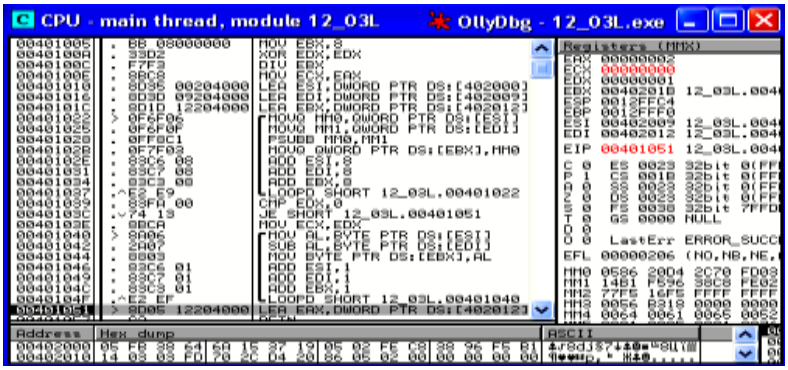


Рис. 18.4. Вікна налагоджувача при виконанні програми з лістингу 18.4

$-200 - 200 = -400$, $|400| - 256 = 144 = 1001\,0000_{пр} = 0111\,0000_{дод} = 70h$,
$2Ch = 0010\,1100 = 44$,
$D4h = 1101\,0100_{дод} = 0010\,1100 = 44$ і т.д.

PSUBUSB, PSUBUSW приймач, джерело	Беззнакове віднімання байтів (**B**), слів (**W**) з насиченням

Команди **psubusb, psubusw** виконують віднімання беззнакових байтів або слів з насиченням. Якщо значення результату менше нуля, то результат операції насищається до 0h.

Лістинг 18.5. Виконання паралельного віднімання 2-х масивів по 19 знакових даних розміром у слово:

```
        Title Рысованый А.Н.
        ; виконати віднімання двох масивів, кількість елементів яких дорівнює 19
        .686            ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти
        .MMX            ; директива визначення команд MMX
        option casemap:none  ; отличие строчных и прописных букв
        include \masm32\include\windows.inc    ; файлы структур, констант …
        include \masm32\macros\macros.asm
    uselib kernel32, user32,masm32
          mZeroRegs1 macro reglist:vararg
            FOR param:=<eax>,<reglist>
              xor param,param
            ENDM
          ENDM
  PSubW1 proto
   BSIZE equ 150
.data
mas1 dw 12, 32,-4,6,120,126,-33,88,80,  10,-55,65,123,-74,33, 77,16,94,  -1
mas2 dw 42,-123,3,21, 32,-4, 32,11,75, -44, 55,29, 12, 32,-3, -4,65,13, -46
len1 EQU ($-mas2)/type mas2 ; обчислення кількості чисел масиву mas2
 res1 dw len1 dup(?),0
titl1    db "Параллельная обработка массива слов командой MMX psubsw",0
```

```
 info1 db "Результат  параллельной обработки командой psubsw:",10,13,13
 buf dw BSIZE dup (?);0
frmt   db " %d   ",0
.code
_st:
            invoke PSubW1
            invoke ExitProcess, 0
PSubW1 proc
  mZeroRegs1 edx;
            mov eax, len1         ; підготування до ділення
            mov ebx, 4 ; кількість паралельно оброблюємих чисел за один цикл
            div ebx                 ; eax=len/4, edx-залишок
            mov ecx, eax          ; занесення вмісту eax у ecx
    lea esi, mas1     ; занесення у esi адреси mas1
    lea edi, mas2     ; занесення у edi адреси mas2
    lea ebx, res1     ; занесення у ebx адреси res
m1:    movq MM0, qword ptr [esi]      ; занесення восьми байтів з mas1
  psubsw MM0, qword ptr [edi]  ; паралельне віднімання
  movq qword ptr [ebx], MM0    ; занесення вмісту MM0 у res
       add esi,8    ; зміщення вказівника на mas1 на 8 байтів
       add edi,8   ; зміщення вказівника на mas2 на 8 байтів
       add ebx,8  ; зміщення вказівника на res1 на 8 байтів
     loop m1
            cmp edx,0h    ; порівняння edx із нулем
            jz exit1          ; перейти, якщо нуль
            mov ecx, edx ; занесення вмісту edx до ecx
 m2:    mov ax, [esi]       ; занесення елемента mas1
            sub ax, [edi]       ; віднімання елемента mas2
            mov word ptr [ebx], ax ; занесення вмісту у res1
            add esi,2   ; зміщення вказівника на mas1
            add edi,2   ; зміщення вказівника на mas2
            add ebx,2  ; зміщення вказівника на res1
     loop m2
exit1: lea esi, word ptr res1 ; занесення до esi адреси res
   mZeroRegs1 ebx      ; заповнення нулями ebx
            mov ecx, len1 ; занесення len1 у ecx
@@:    mov ax, [esi]    ; занесення елемента res1
            movsx eax,ax  ; розширення вмісту до eax
            push ecx        ; занесення ecx у стек
            push esi        ; занесення esi у стек
            push ebx        ; занесення ebx у стек
invoke wsprintf,ADDR buf[ebx],ADDR frmt,eax
            pop ebx                 ; витягнення ebx зі стеку
            add ebx,4; збільшення ebx
            mov buf[ebx],20h ; занесення до буфера пробілу
            add ebx,1
            pop esi
            add esi,2
            pop ecx
     loop @b
invoke MessageBox, 0, ADDR info1, ADDR titl1, MB_OK
            ret
            PSubW1 endp
        end _st
```

Програма виконана з використанням процедури користувача PSubW1 proc. Для заповнення нулями викооооористан макрос mZeroRegs1 macro.

У зв'язку з тим, що розмірність чисел вибрана dw кількість паралельно оброблюваних в циклі чисел дорівнює 4 (mov ebx, 4). Кількість циклів для паралельної обробки визначається у фрагменті коду:

```
mov eax, len1   ; підготування до ділення
mov ebx, 4 ; кількість паралельно оброблюємих чисел за один цикл
div ebx             ; eax=len/4, edx-залишок
mov ecx, eax   ; занесення вмісту eax у ecx
```

і зберігається в eax, а залишок чисел, який повинен оброблятися по одному числу зберігається в edx. Цей залишок чисел обробляється в циклі m2.

Функція wsprintf розташована в циклі m3 і перетворює отримані числа в символи та зберігає їх в буфері buf[ebx]. Буфер зберігає символи розміром dw, тому що числа масивів, які перетворюються також мають розмір dw. Якщо буфер вибрати більшим (наприклад dd), то вже при перетворенні першого числа отримаємо зайві числа символів. А в подальшому, при перетворенні другого числа на це місце буде записаний результат нового перетворення. Й так далі. В результаті програми виведе тільке один результат, тому що інші сформовані невірно.

Для того, щоб функція MessageBox виводила результати через пропуски в програмі використовується два прийоми. Перший – це в рядку форматування frmt db " %d ",0 після %d вставлені пропуски. Але число таких пропусків в цьому рядку з розмірністю db не може бути великим. Другий прийом – це вставка в буфер символу пропуску з подальшим зрушенням буфера на 1. За це відповідає франмент коду:

```
mov buf[ebx],20h ; занесення до буфера пробілу
add ebx,1
```

Результат виконання програми з лістингу 18.5 наведено на рис. 18.5.

Рис. 18.5. Результат виконання програми з лістингу 18.5

18.2.3. Команди множення

ММХ-команди попарно (паралельно) перемножують 16-розрядні слова операндів. Усі команди формують результат за принципом циклічної арифметики.

PMULHW приймач, джерело	Множення знакових упакованих слів та вибірка старших бітів результату (старше множення)

Команда **pmulhw** виконує попарне множення знакових упакованих 16-розрядних слів зі знаками. Результатом операції є чотири 32-розрядних добутки, при цьому старші розряди зберігаються в 16-розрядних словах вихідного

операнда, а молодші розряди не зберігаються. Тому при перемножуванні чисел, які вміщаються в 16-розрядній сітці результат буде дорівнювати нулю.

Фрагмент використання команди може бути таким:

```
movq MM0, qword ptr [esi]  ; занесення восьми байтів з масиву
pmulhw MM0, qword ptr [edi] ; паралельне множення
```

PMULLW приймач, джерело	Множення знакових упакованих слів та вибірка молодших бітів результату (молодше множення)

Команда **pmullw** виконує попарне множення знакових упакованих 16-розрядних слів зі знаком вхідного і вихідного операндів, що дає чотири 32-розрядних добутки. Молодші розряди добутку зберігаються в 16-розрядних словах вихідного операнда, а старші розряди добутку втрачаються. Вхідним операндом може виступати MMX-регістр або елемент пам'яті, а вихідним операндом повинен бути MMX-регістр.

Програма виконання паралельного множення 2-х масивів даних розміром у слово наведена в лістингу 18.6.

Лістинг 18.6. Виконання паралельного множення 2-х масивів даних розміром у слово:

```
        .686                    ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти
        .MMX                    ; директива визначення команд MMX
        option casemap:none ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32
.data                                   ; директива определения данных
        mas1 DW 1, 20, 3, 4, 100, 4, 3, 5, 6
        mas2 DW 2, 4, 5, 2, 200, 4, 1, 0, 6
        len1 EQU ($-mas2)/type mas2 ; обчислення кількості подвійних слів mas2
        sum DW len1 DUP(0),0            ; резервування байтів під змінну sum
        sum2 DD len1 DUP(0),0           ; резервування байтів під змінну sum
_title db "Результат параллельного умножения массивов с помощью команды PMULLW
MMX", 0
        buf dw len1 DUP(0),0            ; буфер виведення повідомлення
        ifmt db "Использование команды PMULLW MMX.",10,13,13
            db "mas1    1,  20,  3,  4,  100,  4,  3,  5,  6",10,13
            db "mas2    2,   4,  5,  2,  200,  4,  1,  0,  6",10,13
            db "Ответ =", len1 dup(" %d "),0
.code                                   ; директива начала кода программы
_st:                                    ; метка начала программы с именемst
          mov  EAX, len1        ; підготовка до визначення кількості циклів
          mov  EBX, 4 ; кількість паралельно оброблюємих чисел за один цикл
          xor  EDX, EDX         ; підготовка регістра до ділення (онулення EDX)
          div  EBX    ; визначення кількості циклів  eax := 2 , edx := 1
          mov  ECX, EAX ; занесення кількості циклів до лічильника
          lea   ESI, mas1       ; esi := addr mas1
          lea   EDI, mas2       ; edi := addr mas2
          lea   EBX, sum        ; ebx := addr sum
m1:   movq  MM0, qword ptr [ESI]  ; завантаження 4-х слів масиву mas1
          pmullw MM0, qword ptr [EDI] ; паралельне циклічне додавання
          movq  qword ptr [EBX], MM0  ; збереження в новому масиві
          add   ESI, 8   ; підготовка адреси mas1 до нового зчитування
          add   EDI, 8   ; підготовка адреси mas2 до нового зчитування
```

```
          add   EBX, 8   ; підготовка адреси sum  до нового зчитування
        loop m1              ; ecx := ecx – 1 та перехід, якщо ecx != 0
          cmp   EDX, 0  ; порівняння з нулем
          jz    exit1        ; перейти, якщо всі числа оброблені (edx = 0)
          mov   ECX, EDX ; занесення остачі необроблених слів
m2:   mov   AX, word ptr [ESI]
          ;add   AX, word ptr [EDI]
          ;mov   BX, word ptr [EDI]
          mul word ptr [EDI]
          mov   word ptr [EBX], AX
          add   ESI, 2
          add   EDI, 2
          add   EBX, 2
         loop m2
exit1:  ; розширення слів до подвійних слів для перетворення wsprintf
        movzx eax,sum
        mov  sum2[0],eax
           movzx eax,sum[2]
           mov  sum2[4],eax
        movzx eax,sum[4]
        mov  sum2[8],eax
          movzx eax,sum[6]
          mov  sum2[12],eax
        movzx eax,sum[8]
        mov  sum2[16],eax
          movzx eax,sum[10]
          mov  sum2[20],eax
        movzx eax,sum[12]
        mov  sum2[24],eax
          movzx eax,sum[14]
          mov  sum2[28],eax
        movzx eax,sum[16]
        mov  sum2[32],eax
 invoke  wsprintf, ADDR buf, ADDR ifmt, \   ; адреса рядка перетворення формату
sum2,sum2[4],sum2[8],sum2[12],sum2[16],sum2[20],sum2[24],sum2[28],sum2[32]
        invoke MessageBox, 0, addr buf, addr _title, MB_ICONINFORMATION
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
        end _st         ; завершение программы с именем_st
```

Результат виконання програми з лістингу 18.6 наведено на рис. 18.6.

Рис. 18.6. Результат виконання програми

PMADDWD приймач, джерело	Множення та додавання знакових упакованих слів

Команда **pmaddwd** виконує попарне множення 16-розрядних знакових упакованих слів зі знаком, які знаходяться в двох операндах. Після отримання в результаті чотирьох 32-розрядних добутків перший добуток складається з

другим, а третій – з четвертим. Суми записуються в 32-розрядні слова вихідного операнда. Якщо всі слова на вході дорівнюють 8000h, то результат дорівнює 8000 0000h (єдиний випадок, коли перемноження негативних чисел дає негативний результат). Вхідним операндом може виступати MMX-регістр або елемент пам'яті, а вихідним операндом повинен бути MMX-регістр. Алгоритм роботи команди pmaddwd наведено на рис. 18.6.

Команди pmulhw і pmullw дозволяють виконати множення чотирьох 16-розрядних операндів одночасно, при цьому розрядність результатів множення виявляється в два рази більшою за розрядність операндів. Для отримання повного результату множення за допомогою цих команд необхідно виконати таку послідовність кроків:

1. Одержати старші 16 біт добутку, використовуючи команду pmulhw.
2. Одержати молодші 16 біт добутку, використовуючи команду pmullw.
3. Об'єднати добутки в одне подвійне слово за допомогою команд punpckhwd і punpcklwd.

Фрагмент використання команди може бути таким:

```
...
@@:
  movq MM0, qword ptr [esi]  ; занесення восьми байтів з mas1
  movq MM1, qword ptr [edi]  ; занесення восьми байтів з mas1
  PMADDWD MM0, MM1 ; паралельне множення
  movq qword ptr [ebx], MM0 ; занесення вмісту MM0 у res
    add esi,8  ; зміщення вказівника на mas1 на 8 байтів
    add edi,8  ; зміщення вказівника на mas2 на 8 байтів
    add ebx,8 ; зміщення вказівника на res1 на 8 байтів
   loop @b
```

18.3. MMX-команди пакування та розпакування даних

MMX-команди пакування перетворюють довгі елементи даних (16- і 32-розрядні слова) в короткі. Якщо початкове значення не розміщується в короткому форматі даних, то відбувається насичення. В такому випадку результатом вважається граничне значення допустимого діапазону вихідного типу даних. Цими командами можна користуватися для збільшення числа значущих розрядів при обчисленнях.

PACKSSWB, PACKSSDW	Пакування зі знаковим насиченням
приймач, джерело	

Команди **packsswb, packssdw** перетворюють довгі дані (16- та 32-розрядних слів із знаком) в короткі (байти або 16-розрядні слова зі знаком). Якщо початкове значення було за межами допустимого діапазону для вихідного типу даних, то результатом пакування вважається найближче граничне значення діапазону. Вхідним операндом може виступати MMX-регістр або елемент пам'яті; вихідний операнд повинен знаходитися в MMX-регістрі. Наприклад, дію команди packsswb MM0, MM1 наведено на рис. 18.7.

Розглянемо практичний приклад використання команди packsswb.

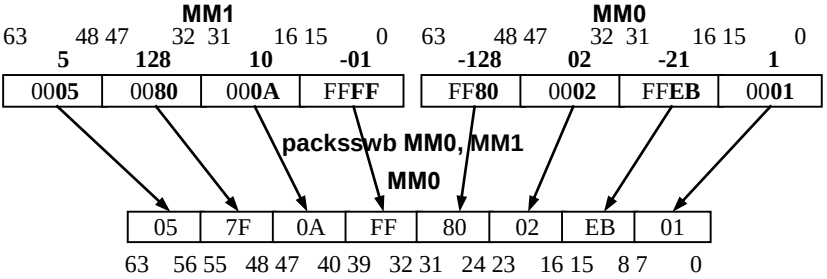


Рис. 18.7. Дія команди packsswb MM0, MM1

Лістинг 18.7. Програма пакування знакових слів у байти за допомогою команди packsswb:

```
        title Rysovaniy A.N.                            rysov@rambler.ru
        .686                            ; директива определения типа микропроцессора
.model  flat, stdcall   ; задание линейной модели памяти
.MMX            ; директива визначення команд XMM-розширення
option casemap:none             ; отличие строчных и прописных букв
include \masm32\include\kernel32.inc ; системные функции применений…
includelib \masm32\lib\kernel32.lib     ; виклик бібліотеки kernel32.lib
.data                                           ; директива определения данных
        mas1  DW 01, -21, 02, -128, -01, 10, 128, 5, 1, 2
 len EQU ($-mas1)/ type mas1 ; обчислення кількості чисел змінної mas1
        res DB len DUP(0)       ; резервування байтів для змінної res
.code                           ; директива начала кода программы
_start:                         ; метка начала программы с именем _start
        mov  EAX, len           ; eax := 1
        xor EDX, EDX ; підготування до ділення (додавання за модулем 2)
        mov  EBX, 8
        div  EBX        ; edx := 0, eax := 8
        mov  ECX, EAX           ; лічильник циклів по 8 слів
        lea  ESI, mas1                  ; збереження в ESI адреси масиву mas1
        lea  EDI, res       ; збереження в EDI адреси масиву res
m1:  movq MM0, qword ptr [ESI]          ; 8 байтів в масив mas1 в MM0
        movq MM1,[esi+8]                ; занесення 8 байтів масиву mas1 в MM1
        packsswb MM0, MM1 ;
        movq   qword ptr [EDI], MM0  ; збереження результату в пам'яті
        add     ESI, 16                         ; esi := 10
        add     EDI, 4                          ; edi := 14
        loop    m1                              ; ecx := ecx – 1 та перехід, якщо ecx ? 0
        cmp     EDX, 0                  ; порівняння залишку
        jz _end
        mov ecx,edx             ; занесення залишку від ділення
m2:  mov al,byte ptr [esi]      ; перенесення чисел без обробки
        mov byte ptr [edi],al ;
        add esi,2 ; збільшення адреси для вибірки нового числа
        add edi,2 ; збільшення адреси для збереження числа
        loop m2
_end:
invoke ExitProcess,0 ; повернення управління  ОС та вивільнення ресурсів
        end _start                      ; завершение программы с именем _start
```

Програма з лістингу 18.6 упаковує слова, які розташовані по 4 слова в масиві mas1. Команди

```
add     ESI, 16  ; esi := 10
```

```
add     EDI, 4    ; edi := 14
```

необхідні для випадку, коли потрібно упакувати більше 12 слів. Якщо в масиві кількість слів не є кратною чотирьом, то слова, які залишилися, не упаковуються.

Команди

```
movq  MM1,[esi+8]
packsswb MM0, MM1
```

наведено для того, щоб наочніше простежити за процесом перетворення операндів. Ці команди можна замінити однією командою packsswb MM0, [esi+8].

Приклад 18.3. Виконати паралельне пакування 2-х масивів по 8 знакових даних розміром у байт.

Лістинг 18.8:

```
.686                    ; директива определения типа микропроцессора
.model flat                         ; завдання лінійної моделі пам'яті
.MMX                                ; директива визначення команд MMX
option casemap:none   ; отличие строчных и прописных букв
…
BSIZE equ 50                        ; завдання реальної кількості байтів
 .data                              ; директива определения данных
     mas1   dw   -4,2,9,-38,77,1,6,43   ; масив mas1
     mas2   dw   -6,1,-11,88,19,-26,8,-7; масив mas2
     rez db 16 dup(?)                       ; результат пакування
     buf BYTE BSIZE dup(0)          ; резервування пам'яті для буфера
     st1 db "The Rezult",0     ; назва спрощеного вікна
     ifmt  db "%d",0                    ; завдання перетворення символу
stdout DWORD ?        ; резервування в пам'яті 32-розрядної комірки
cWritten DWORD ?  ; резервування комірки для адреси символів виведення
    crlf db 0dh, 0ah
.code                                   ; начало сегмента данных
 _st:                                   ; метка начала программы
invoke GetStdHandle, STD_OUTPUT_HANDLE    ; одержання дескриптора
    mov stdout,eax              ; запам'ятовування отриманого дескриптора
    lea esi,mas1                        ; завантаження адреси mas1
  movq MM0,qword ptr [esi] ; завантаження молодшої частини mas1 в MM0
    packsswb MM0,qword ptr mas2         ; пакування mas1 та mas2
    movq qword ptr [rez],MM0            ; збереження результату
    lea esi,mas1+8              ;
    movq MM0,qword ptr [esi]  ; завантаження старшої частини mas1 в MM0
    packsswb MM0,qword ptr mas2+8     ; пакування масивів mas1 та mas2
    movq qword ptr [rez+8],MM0          ; збереження результату
          lea esi,rez
          mov ecx,16        ; лічильник
.WHILE (ecx!=0)                 ; виведення на екран
          push ecx
          mov bl,[esi]
          movsx eax,bl    ; записуємо число в eax з розширенням знака
          invoke wsprintf,addr buf,addr ifmt,eax ; перетворення чисел в рядок
invoke  WriteConsoleA, stdout, ADDR buf, BSIZE, ADDR cWritten, 0
invoke WriteConsoleA, stdout, ADDR crlf,2,ADDR cWritten, 0 ; каретка
          pop ecx
          inc esi               ; перехід на наступне число
          dec ecx               ; декремент лічильника
.ENDW
          invoke Sleep,20000
invoke ExitProcess,0 ; повернення управління  ОС та вивільнення ресурсів
```

```
end _st                                  ; завершение программы с именем_st
```

Числа виводяться вертикально, але видно, що вони вже упаковані. Результат виконання програми з лістингу 18.7 наведено на рис. 18.8.

Рис. 18.8. Результат виконання програми з лістингу 18.7

Дія команди packssdw MM0, MM1, яка перетворює подвійне слово в слово, наведена на рис. 18.9.

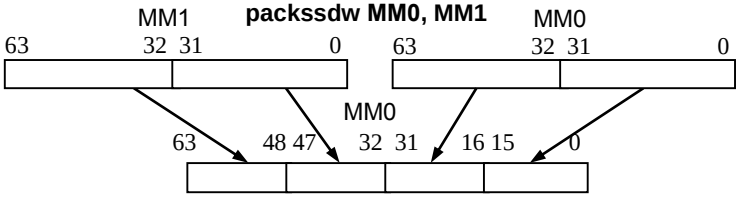


Рис. 18.9. Дія команди packssdw MM0, MM1

PACKUSWB приймач, джерело	Пакування 16-розрядних чисел зі знаком в байти із беззнаковим насиченням

Команда packuswb виконує перетворення 16-розрядних чисел зі знаком з обох операндів в байти без знака і записує їх у вихідний операнд. Якщо початкове слово зі знаком було більше FFh, результатом перетворення вважається FFh. Якщо початкове слово зі знаком негативне, результатом перетворення вважається 00h. Вхідним операндом може виступати MMX-регістр або елемент пам'яті; вихідний операнд повинен знаходитися в MMX-регістрі. Розглянемо приклад використання команди **packuswb.**

Лістинг 18.9:

```
        .686 ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти
        .MMX          ; директива визначення команд XMM-розширення
        option casemap:none             ; отличие строчных и прописных букв
        include \masm32\include\kernel32.inc ; системные функции применений…
        includelib \masm32\lib\kernel32.lib   ; виклик бібліотеки kernel32.lib
.data                                   ; директива определения данных
        mas1 WORD 1,2,3,4 ; резервування комірок пам'яті для масиву mas1
        len1 equ ($-mas1)/type mas1
        mas2 WORD 5,6,7,8 ; резервування комірок пам'яті для масиву mas2
        len2 equ ($-mas2)/type mas2
res BYTE (len1+len2) dup(O) ; резервування комірок пам'яті для масиву res
.code                                   ; директива начала кода программы
_start:                                 ; метка начала программы
        movq MM1,QWORD PTR mas1 ; завантаження всього масиву mas1
```

```
        movq MM2,QWORD PTR mas2 ; завантаження всього масиву mas2
        packuswb MM1,MM2        ; пакування 16 розрядних чисел в байти
        movq QWORD PTR res,MM1 ; паралельне збереження результату
invoke ExitProcess,0 ; повернення управління  ОС Windows та вивільнення ресурсів
        end _start                      ; завершение программы с именем _start
```

Заповнення регістрів MM0 та MM1 до виконання команди packuswb наведено на рис. 18.10.

```
MM0 0004 0003 0002 0001
MM1 0008 0007 0006 0005
```

Рис. 18.10. Заповнення регістрів MM0 та MM1 до виконання команди packuswb

Результат виконання програми з лістингу 18.8 наведено на рис. 18.11.

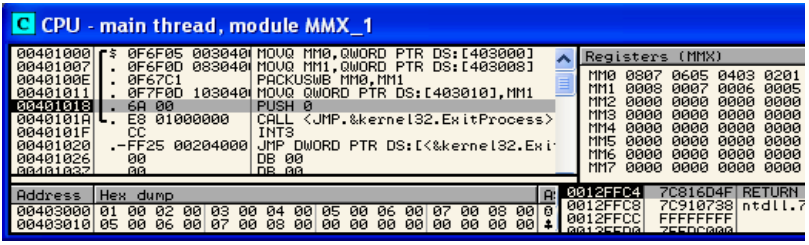


Рис. 18.11. Результат виконання програми з лістингу 18.9

PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ приймач, джерело	Розпакування та об'єднання старших елементів

Команди **punpckhbw, punpckhwd, punpckhdq** виконують попарне об'єднання початкових елементів даних (байтів, 16- або 32-розрядних слів), що знаходилися в ***старших*** 32 розрядах обох операндів.

Результат об'єднання записується у вихідний операнд (приймач). Початкові значення молодших розрядів операндів на результат не впливають. Вхідним операндом може виступати MMX-регістр або елемент пам'яті; вихідний операнд повинен знаходитися в MMX-регістрі:

punpckhbw/d/q mmx_i, mmx_j/mem64.

Якщо відомо, що при формуванні нового збільшеного учетверо слова старші подвійні слова мають усі нулі, то молодше слово можна таким чином збільшити.

Розглянемо практичний приклад використання команди punpckhbw. Наприклад, дію команди punpckhbw MM0, mem наведено на рис. 18.12.

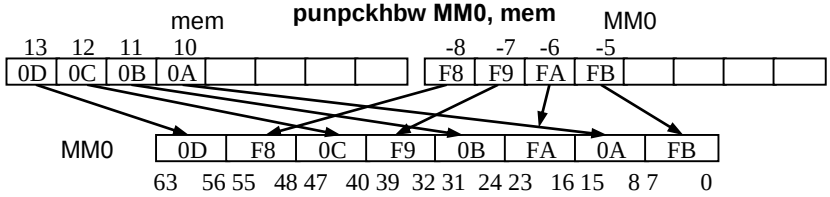


Рис. 18.12. Дія команди punpckhbw MM0, mem

Лістинг 18.10. Програма використання команди punpckhbw для об'єднання двох масивів даних розміром у байт:

```
    .686            ; директива определения типа микропроцессора
    .model flat,stdcall ; задание линейной модели памяти
    .MMX                                ; директива визначення команд MMX
    option casemap: none      ; отличие строчных и прописных букв
    includelib \masm32\lib\kernel32.lib
    ExitProcess proto :dword
  .data                                         ; директива определения данных
      str1  DB -1, -2, -3, -4, -5, -6,  -7,  -8
      str2  DB  6,  7,  8,   9, 10, 11, 12, 13
      res   DB  8 DUP(0)
  .code                                         ; директива начала кода программы
  _st:                                          ; метка начала программы с именемst
      movq    MM0, qword ptr str1
      punpckhbw MM0, qword ptr str2
      movq   qword ptr res, MM0
      lea     EAX, res              ; збереження адреси масиву res
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
      end _st                       ; завершение программы с именем_st
```

Програма повертає в регістр EAX адресу масиву res. Дія команди punpckhbw MM0, mem полягає в створенні 64-розрядного слова з двох старших половинок операндів, які розміщуються у регістрах MM0 та mem.

Вікна налагоджувача при виконанні програми з лістингу 18.9 наведено на рис. 18.13. У цих вікнах в регістрі MM0 залишився вміст створеного 64-розрядного слова, яке потім було збережене в комірках пам'яті, починаючи з адреси 00403010.

Схема на рис. 18.14 показує дію команди punpckhwd з використанням регістрів MM0 та MM1.

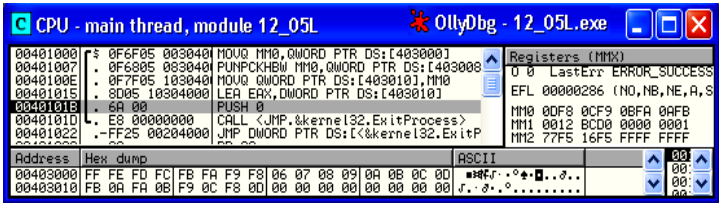


Рис. 18.13. Вікна налагоджувача при виконанні програми з лістингу 18.10

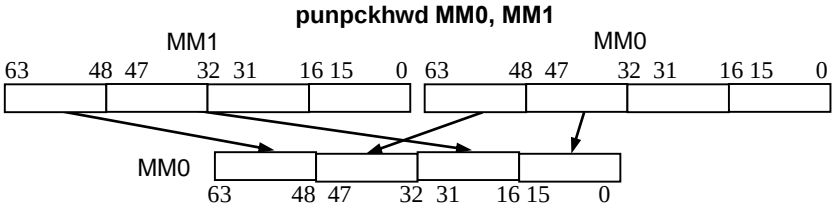


Рис. 18.14. Дія команди punpckhwd MM0, MM1

Команда punpckhdg розпаковує старші подвійні слова операндів в операнд-приймач.

Схема на рис. 18.15 показує дію команди punpckhdg з використанням регістрів MM0 та MM1.

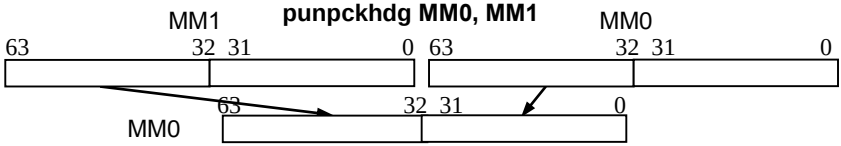


Рис. 18.15. Дія команди punpckhdg MM0, MM1

PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ приймач, джерело	Розпакування та об'єднання молодших елементів

Команди **punpcklbw, punpcklwd, punpckldq** виконують попарне об'єднання початкових елементів даних (байтів, 16- або 32-розрядних слів), що знаходилися в ***молодших*** 32 розрядах обох операндів. Результат об'єднання записується у вихідний операнд (приймач). Початкові значення старших розрядів операндів на результат не впливають. Вхідним операндом може виступати MMX-регістр або елемент пам'яті; вихідний операнд повинен знаходитися в MMX-регістрі.

Механізм роботи команд розпаковування стане зрозумілішим, якщо подивитися на рисунки, що ілюструють їх роботу (рис. 18.16 – 18.18).

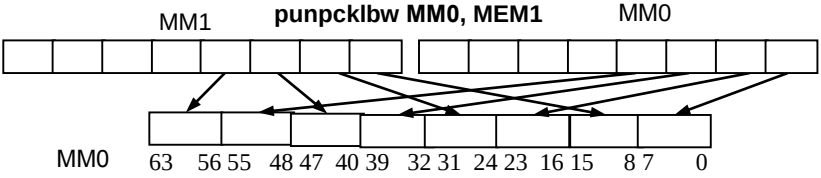


Рис. 18.16. Дія команди punpcklbw MM0. MM1

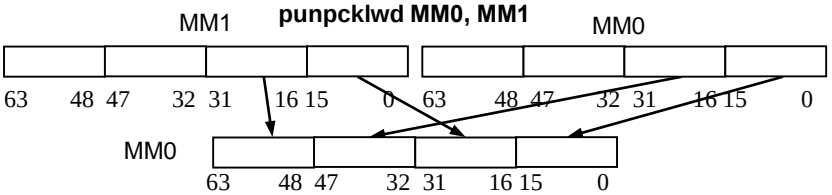


Рис. 18.17. Дія команди punpcklwd MM0, MM1

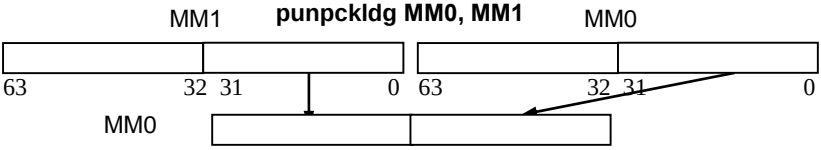


Рис. 18.18. Дія команди punpckldg MM0, MM1

Розглянемо програму з лістингу 18.3 з даними розміром у **слово**. Фрагмент такої програми може бути таким:

```
    …
    len EQU ($-mas2)/2        ; обчислення кількості слів масиву mas2
    …
_st:                          ; метка начала программы с именемst
    mov  EAX, len             ; eax := 9
    mov  EBX, 4               ; кількість одночасно занесених слів до MMX
    xor  EDX, EDX ; додавання за модулем 2 (заповнення нулями  EDX)
    div  EBX                  ; eax := 2 , edx := 1
    mov  ECX, EAX             ; лічильник цілих циклів
    lea  ESI, mas1            ; esi := addr mas1
    lea  EDI, mas2            ; edi := addr mas2
    lea  EBX, sum     ; занесення початку масиву sum;   ebx := addr sum
m1:  movq  MM0, qword ptr [ESI] ;
    paddw MM0, qword ptr [EDI]     ; паралельне циклічне додавання
    movq MM1,MM0 ;
    pxor MM2,MM2 ;
  punpcklwd MM1,MM2 ;
  movq  qword ptr [EBX], MM1 ;
  punpckhwd MM0,MM2 ;
  movq  qword ptr [EBX+8], MM0 ;
  add   ESI, 8                ; підготовка адреси mas1 до нового зчитування
  add   EDI, 8                ; підготовка адреси mas2 до нового зчитування
  add   EBX, 16               ; підготовка адреси sum  до нового зчитування
loop m1           ; ecx := ecx – 1 та перехід, якщо ecx != 0
      cmp   EDX, 0  ; визначення залишку необроблених елементів масивів
      jz    exit          ; якщо необроблені елементи є, то перейти на exit
      mov ECX, EDX     ; занесення до лічильника кількості необроблених чисел
m2:  mov  AX, word ptr [ESI] ; занесення необробленого елемента  з mas1
        add   AX, word ptr [EDI]     ; додавання елементів двох масивів
        mov  word ptr [EBX], AX    ; збереження результату
        add   ESI, 4               ; підготовка до вибірки елемента з mas1
        add   EDI, 4               ; підготовка до вибірки елемента з mas2
        add   EBX, 4               ; підготовка до занесення результату в пам'ять
        dec   ECX   ; зменшення лічильника на одиницю
        jnz    m2     ; якщо у лічильнику не нуль, то перейти на m2
    exit:
```

При проходженні першого циклу з міткою m1 по команді

movq MM0, qword ptr [ESI]

в регістр MM0 заносяться числа 9C40 7530 4E20 0001. По команді

paddw MM0, qword ptr [EDI]

виконується додавання двох значень масивів чисел. В результаті додавання в регістрі MM0 отримується результат: 0002 3880 EA60 0003, в якому знаходиться чотири числа розміром у слово. У зв'язку з тим, що числа будуть оброблятися функцією wsprintf для виведення функцією MessageBox, то необхідно записати їх не як 16-розрядні, а як 32-розрядні. Для цього необхідні перетворення, тобто розпакування. Спочатку потрібно взяти два молодших 16-розрядних числа EA60 0003 та переписати їх в формат 32-розрядних чисел: 0000 EA60 0000 0003. Молодші два числа будемо розширювати в регістрі MM1, а старші два числа – в регістрі MM2. Для цього спочатку необхідно обнулити

регістр MM2. Якщо такого не зробити, то від співпроцесора сміття, що залишилося буде збільшувати розмір чисел, але в регістрах ММХ такого не буде видно.

По команді punpcklwd MM1,MM2 молодше 16-розрядне число залишиться на місці, а наступне старше молодше перепишиться в молодшу частину старшого 32-розрядного числа. Молодші два нульових числа з MM2 перепишуться в старші частини регістра MM1. В результаті в MM1 отримається значення: 0000 EA60 0000 0003, яке потім записується до пам'яті. А після команди punpckhwd MM0,MM2 в MM0 залишиться 0000 0002 0000 3880.

Виведення масиву чисел відбудеться аналогічно лістингу 18.3.

18.4. ММХ-команди порівняння

ММХ-команди порівняння попарно порівнюють елементи даних (байти, 16- або 32-розрядні слова) вхідного і вихідного операндів. Залежно від результату порівняння відповідний елемент даних вихідного операнда заповнюється **нулями** (якщо умова не виконується) або **одиницями** (умова виконується). Ці команди, як і вся решта ММХ-команд, не встановлюють ознак виконання операції.

ММХ-команди порівняння поділяються на дві групи:

- команди звичайного порівняння (дорівнює або не дорівнює);
- команди порівняння за величиною (більше або менше).

Операції порівняння проводяться над байтами, словами і подвійними словами.

PCMPEQB, PCMPEQW, PCMPEQD приймач, джерело	Перевірка на рівняння

Команди **pcmpeqb, pcmpeqw, pcmpeqd** виконують попарне порівняння елементів даних (байтів, 16- або 32-розрядних слів) вхідного і вихідного операндів. Якщо елемент даних вихідного операнда дорівнює відповідному елемента вхідного, то такий елемент вихідного операнда заповнюється всіма одиницями. Якщо елементи не рівні, то він заповнюється нулями. Вхідним операндом можуть виступати ММХ-регістр або елемент пам'яті; а вихідний операнд повинен знаходитися в ММХ-регістрі.

Лістинг 18.11. Дослідження команди pcmpeqw:

```
.686                    ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти
.MMX
option casemap:none             ; отличие строчных и прописных букв
include \masm32\include\windows.inc    ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib kernel32, user32
.data
  mas1 DW 1,5,8,2,3,1,4,69,8,5,1,78,96,52,12,44
  mas2 DW 1,4,5,10,2,5,8,69,8,5,1,78,96,52,12,44
  len1 EQU ($-mas2)/type mas2
```

```
   sum DW len1 DUP(0),0
   sum2 DD len1 DUP(0),0
   _title db "Параллельное сравнение слов", 0
   buf dd len1 DUP(0),0
   ifmt db"Исследование команды PCMPEQW",0Ah,0Dh,0Dh,
   " Ответ =", len1 dup(" %d "),0
.code
_st:
        mov EAX, len1
        mov EBX, 4
        xor EDX, EDX
        div EBX
        mov ECX, EAX
        lea ESI, mas1
        lea EDI, mas2
        lea ebx, sum
m1:
        movq MM0, qword ptr[ESI]
        movq MM1, qword ptr[edi]
   pcmpeqw MM0,MM1
        movq qword ptr[EBX], MM0
        add ESI, 8
        add EDI, 8
        add EBX, 8
     loop m1
        xor ebx,ebx
        xor edx,edx
        mov ecx,len1
@@:  mov ax,sum[ebx]
        movzx eax,ax   ; розширення розрядності
        mov sum2[edx],eax
        add ebx,2
        add edx,4
     loop @b
invoke wsprintf,ADDR buf, ADDR ifmt, sum2,sum2[4],sum2[8],sum2[12], sum2[16], sum2[20],
sum2[24],sum2[28],sum2[32],sum2[36],sum2[40],sum2[44],sum2[48], sum2[52], sum2[56],
sum2[60]
        invoke MessageBox, 0, addr buf, addr _title, MB_ICONINFORMATION
        invoke ExitProcess, 0
        end _st
```

Результат виконання програми наведено на рис. 18.19.

Рис. 18.19. Результат виконання програми

При дорівнюванні значень в регістр-приймач записуються всі одиниці. А якщо значення не дорівнені – нули.

Для виведення результату перед функцією перетворення wsprintf виконується розширення значення числа до 32 розрядів командою movzx eax,ax. А командою mov sum2[edx],eax розширене до 32 розрядів число записується в комірку пам'яті за адресою, яка залежить йще від значення регістра edx.

PCMPGTB, PCMPGTW, PCMPGTD приймач, джерело	Порівняння на більше

Команди **pcmpgtb, pcmpgtw, pcmpgtd** виконують попарне порівняння елементів даних (байтів, 16- або 32-розрядних слів із знаком) вхідного і вихідного операндів. Якщо елемент даних вихідного операнда більший від відповідного елемента вхідного, то такий елемент вихідного операнда заповнюється одиницями. Якщо ж він не більший від вхідного, то він заповнюється нулями. Вхідний операнд може знаходитися в ММХ-регістрі або в пам'яті, а вихідний операнд – в ММХ-регістрі.

Для операцій звичайного порівняння нульові значення формуються, якщо відповідні байти, слова або подвійні слова не рівні. Одиничні значення формуються у випадку, якщо відповідні байти, слова або подвійні слова рівні. Для операцій порівняння за величиною одиничні значення формуються, якщо відповідні байти, слова або подвійні слова першого операнда більші від відповідних байтів, слів або подвійних слів другого операнда.

Розглянемо програму паралельного порівняння двох масивів знакових чисел розміром у байт за допомогою команд pcmpeqb та pcmpgtb (лістинг 18.12).

Лістинг 18.12. Програма паралельного порівняння двох масивів цілих чисел розміром у байт кожен:

```
    .686                    ; директива определения типа микропроцессора
    .model flat,stdcall     ; задание линейной модели памяти
    .MMX                                    ; директива визначення команд MMX
    option casemap: none            ; отличие строчных и прописных букв
    includelib \masm32\lib\kernel32.lib
    include \masm32\include\kernel32.inc
  .data                                             ; директива определения данных
    mas1  DB -1, -2, 3, 4, 0, 1, 12, -8
    mas2  DB -1, -2, 3, 4, 0, 1, 12, -8
    res  DB 16 DUP (5)
  .code                                         ; директива начала кода программы
  _st:  movq  MM0, qword ptr mas1             ; занесення 8 байтів масиву mas1
    pcmpeqb MM0, qword ptr mas2           ; попарне порівняння на рівняння
      movq   qword ptr res, MM0           ; збереження результату в пам'яті
      movq   MM1, qword ptr mas1
      pcmpgtb MM1, qword ptr mas2           ; попарне порівняння на більше
      movq  qword ptr res+8, MM1          ; збереження результату в пам'яті
      lea    EAX, res                     ; збереження в EAX адреси масиву res
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
      end _st                                   ; завершение программы с именем_st
```

Початкові значення комірок пам'яті наведено на рис. 18.20.

Address	Hex dump															
00403000	FF	FE	03	04	00	01	0C	F8	FF	FE	03	04	00	01	0C	F8
00403010	05	05	05	05	05	05	05	05	05	05	05	05	05	05	05	05
00403020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Рис. 18.20. Початкові значення комірок пам'яті налагоджувача при виконанні програми з лістингу 18.12

Для того щоб відрізнити початкові значення масиву res, в кожну комірку пам'яті, починаючи з 00403010, занесено 05h. У зв'язку з тим, що масиви однакові, за командою pcmpeqb MM0, qword ptr mas2 в регістр MM0 заносяться всі одиниці, а за командою pcmpgtb MM1, qword ptr mas2 в регістр MM1 – всі нулі.

Програма повертає командою lea EAX, res адресу масиву res в регістр EAX.

Розглянемо програму виконання порівняння елементів з однаковими індексами двох масивів цілих чисел та створити масив з найбільших чисел.

Лістинг 18.13. Програма паралельного порівняння двох масивів цілих чисел та створення масиву з найбільших чисел:

```
        .686    ; директива определения типа микропроцессора
        .MMX
        .model  flat, stdcall  ; задание линейной модели памяти
        option casemap:none    ; відмінність малих та великих  літер
      include \masm32\include\windows.inc ; файлы структур, констант …
      include \masm32\macros\macros.asm
      uselib user32, kernel32, fpu
.data
        _title db "Команда MMX pcmpgtw",0 ; заголовок вікна
        _buf dd 4 dup(?),0 ; буфер для виведення данных
_message db "Result array: %d, %d, %d, %d",0 ; повідомлення, що виводиться
_arr1 DW 1, 7, 3, 2 ; перший масив
_arr2 DW 4, 1, 3, 2 ; другий масив
_res1 DW 4 dup(0) ; масив для збереження результатів порівняння
_res2 DD 4 dup(0) ; для виведення результатів через функцію wsprinf
        len1 equ ($-_arr1)/type _arr1 ; обчислення довжини масиву
.code
_st:
   movq MM0, qword ptr _arr1 ; запис першого масиву до регістру MM0
   movq MM1, qword ptr _arr2 ; запис другого масиву до регістру MM1
   pcmpgtw MM0, MM1   ; попарне порівняння на більше
   movq qword ptr _res1, MM0 ; результат порівняння в пам'ять
        mov ecx,len1      ; довжина масивів
        lea esi,_arr1     ; адреса масиву _arr1
        lea edi,_arr2     ; адреса масиву _arr2
        lea ebx,_res1     ; адреса масиву _res1
        lea edx,_res2     ; адреса масиву _res2
@@:   mov ax,[ebx] ; результат порівняння елементів масивів
      .IF ax == 0FFFFh ; якщо результат 0FFFFh, то елемент _arr1 > _arr2
      mov ax,[esi]   ;
      .ELSE          ; якщо результат дорівнює 0, то елемент _arr2 > _arr1
      mov ax,[edi]   ; заносимо елемент другого масиву до регістру eax
      .ENDIF
        movzx eax,ax ; для запису до масиву _res2
        mov [edx],eax ; записуємо вміст регістру eax до масиву _res2
add esi,2 ; збільшення адреси для доступу до наступного елементу масиву
add edi,2 ; збільшення адреси для доступу до наступного елементу масиву
add ebx,2 ; збільшення адреси для доступу до наступного елементу масиву
add edx,4 ; збільшення адреси для доступу до наступного елементу масиву
      loop @b ;
invoke wsprintf, addr _buf, addr _message, _res2[0], _res2[4], _res2[8], _res2[12] ;
invoke MessageBox, 0, addr _buf, addr _title, MB_OK ; виведення
        invoke ExitProcess, 0
        end _st
```

Результат виконання програми наведено на рис. 18.21.

Рис. 18.21. Результати виконання програми

18.5. Логічні ММХ-команди

Логічні ММХ-команди виконують порозрядні логічні операції над всіма 64 бітами своїх операндів. Ці команди реалізують логічні операції *І, АБО, І-НІ*, виключене *АБО*.

PAND приймач, джерело	Логічне *І* (порозрядне множення)

Команда **pand** (логічне *І*) обчислює порозрядне логічне *І* своїх операндів:

pand mmx_i, mmx_j/mem64.

PANDN приймач, джерело	Логічне *І – НІ* (порозрядні інверсія та множення)

Команда **pandn** (логічне *І – НІ*) обчислює звернення (порозрядне *НІ*) вихідного операнда, а потім порозрядне логічне *І* між вхідним операндом і оберненим значенням вихідного:

pandn mmx_i, mmx_j/mem64.

POR приймач, джерело	Логічне *АБО* (порозрядне додавання)

Команда **por** (логічне *АБО*) обчислює порозрядне логічне *АБО* своїх операндів:

pand mmx, mmx/mem64.

PXOR приймач, джерело	Виключне *АБО*

Команда **pxor** (виключне *АБО*) обчислює порозрядне логічне виключне *АБО* своїх операндів (застосування за модулем два):

pxor mmx_i, mmx_j/mem64.

18.6. Команди зсуву

Команди ММХ-зсуву виконують зсув кожного елемента даних (16-, 32- або 64-розрядного слова) у вихідному операнді на значення, що задається вхідним операндом. Серед команд зсуву виділяють команди арифметичного і логічного зсуву. При виконанні команд арифметичного зсуву розряди елементів, що звільняються, заповнюються знаком числа (старший біт) і можуть набувати таких значень, як 0 і 1, тоді як при виконанні команд логічного зсуву розряди, що звільнилися, заповнюються нулями.

PSLLW, PSLLD, PSLLQ приймач, джерело	Логічний зсув ліворуч

Команди psllw, pslld, psllq виконують логічний зсув ліворуч елементів даних (16-, 32- або 64-розрядних слів) у вихідному операнді на число бітів, що задається вхідним операндом. Молодші розряди, що звільнилися, заповнюються нулями. Вхідний операнд може бути безпосереднім операндом або може знаходитися в ММХ-регістрі чи в пам'яті:

psllw/d/q mmx_i, imm8/mmx_j/mem64.

Приклад використання ММХ-команд зсуву розглянуто в програмі з лістингу 18.14.

Лістинг 18.14. Приклад використання ММХ-команди зсуву pslld:

```
        .686                    ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти
        .MMX                    ; директива визначення команд MMX
        option casemap:none ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32
.data
        arr dd 10,15,29,25,62,14,18,17,36,-2,4,6
        sz equ ($-arr)/type arr
        result dw sz*type result dup(0)
        ttlIn db "Вх. набор",0
        ttlOut db "Вых. набор",0
        author db "Выполнил",0
        authorN db "Горбенко Я.И. Кит-22А",0
        format db "%d ",0
        buf db ?
.CODE
display proc array:dword,_size:dword ; процедура перетворення масиву в рядок
        lea esi,buf
        mov edi,array
        mov ecx,_size
@@:       mov eax,[edi]
          push ecx ; зберегти лічильник в стеку
          invoke wsprintf,addr [esi],addr format,dword ptr [edi]
          pop ecx                                   ; відновити лічильник із стека
          add edi,type array
          add esi,eax     ; додати до esi кількість виведених в буфер символів
          dec ecx
    jnz @b
            ret
  display endp
_start:
     invoke display,addr arr,sz
     invoke MessageBox,0,addr buf,addr ttlIn,MB_OK
     mov ecx,sz/2
     lea esi,arr
     lea edi,result
cycle:  movq MM0,qword ptr [esi]     ; завантажити 2 числа типу dd в регістр з пам'яті
pslld MM0,1    ; зсув на 1 біт вліво
movq qword ptr [edi],MM0   ; вивантажити 2 числа типу dd в пам'ять з регістра
  add edi,8                 ; збільшити адресу на 8 байт (2 dd)
  add esi,8
  dec ecx
```

```
 jnz cycle
invoke display,addr result,sz
invoke MessageBox,0,addr buf,addr ttlOut,MB_OK
invoke MessageBox,0,addr authorN,addr author,MB_OK
      invoke ExitProcess,0
      END   _start
```

У програмі послідовно виводиться 3 спрощених вікна. У першому вікні виводиться вхідний набір даних. У другому вікні виводяться дані, які отримані після виконання над ними команди pslld зсуву вліво. У третьому вікні виводиться прізвище автора програми. Перед виведенням вікон з даними проводиться їх перетворення в рядок процедурою display.

PSRLW, PSRLD, PSRLQ приймач, джерело	Логічний зсув праворуч

Команди psrlw, psrld, psrlq виконують логічний зсув елементів даних праворуч (16-, 32- або 64-розрядних слів) у вихідному операнді на число бітів, що задається вхідним операндом. Старші розряди, що звільнилися, заповнюються нулями. Вхідний операнд може бути безпосереднім операндом або може знаходитися в ММХ-регістрі чи в пам'яті. Вихідний операнд повинен знаходитися в ММХ-регістрі:

psrlw/d/q mmx_i, imm8/mmx_j/mem64.

PSRAW, PSRAD приймач, джерело	Арифметичний зсув праворуч

Команди **psraw, psrad** виконують арифметичний зсув елементів даних праворуч (16- або 32-розрядних слів) у вихідному операнді на число бітів, що задається вхідним операндом. Якщо зсовується позитивне число, то старші розряди, що звільнилися, заповнюються нулями; якщо негативне – то одиницями. Вхідний операнд може бути безпосереднім операндом або може знаходитися в ММХ-регістрі чи в пам'яті. Вихідний операнд повинен знаходитися в ММХ-регістрі:

psraw/d mmx_i, imm8/mmx_j/mem64.

Приклад використання ММХ-команд зсуву розглянуто в програмі з лістингу 18.15.

Лістинг 18.15. Приклад використання ММХ-команд зсуву:

```
      .686                ; директива определения типа микропроцессора
      .model flat,stdcall ; задание линейной модели памяти
      .MMX                ; директива визначення команд MMX
      option casemap:none ; отличие строчных и прописных букв
      includelib \masm32\lib\kernel32.lib
      ExitProcess proto :dword
  .data                                ; директива определения данных
      mas1 dw 2, 4369, 273, 15         ; масив слів розміром в слово
      a1 dq 2                          ; визначення кількості бітів зсуву
      res dq 3 dup (0)          ; масив результату розміром 64 × 3 бітів
  .code                          ; директива начала кода программы
  _st:  movq  MM0, qword ptr mas1     ; занесення масиву mas1 в MM0
      psllw MM0, qword ptr a1         ; логічний зсув слів ліворуч
      movq  qword ptr res, MM0        ; збереження результату в пам'яті
      movq  MM1, qword ptr mas1
```

```
    psrlw MM1, qword ptr a1          ; логічний зсув слів праворуч
    movq  qword ptr res+8, MM1 ;
    movq  MM2, qword ptr mas1
    psraw MM2, qword ptr a1          ; арифметичний зсув слів праворуч
    movq  qword ptr res+16, MM2 ;
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
    end _st                          ; завершение программы с именем_st
```

У наведеній програмі паралельно завантажується весь масив даних у відповідний регістр MMX. Потім виконується зсув та запис результату у визначені раніше комірки пам'яті. При цьому треба приділити увагу правильності використання директиви визначення даних для кількості бітів зсуву a1 dq 2, від якої результат може бути істотно змінений.

18.7. MMX-команди в Pentium III

Ці команди включено у всі покоління процесорів Intel Pentium, починаючи з Pentium III. Розглянемо деякі з них.

Команди **pavgb, pavgw** обчислюють середнє значення двох чисел, наведених байтами або словами. Значення операндів інтерпретуються як беззнакові цілі числа. Як вхідний операнд можуть виступати MMX-регістр або 64-розрядний елемент пам'яті; вихідним операндом служить один з MMX-регістрів:

pavgb/w mmx_i, mmx_j/mem64.

Команда **pextrw** (розвернути слово) витягує одне з чотирьох упакованих слів вхідного операнда. Команда має три аргументи: вхідний операнд, вихідний операнд і маску. Молодші два біти маски задають у вхідному операнді номер упакованого слова, що підлягає витяганню. Витягнуте слово зберігається в молодшому слові вихідного операнда. Вихідним операндом цієї команди може виступати один з 32-розрядних регістрів загального призначення. Старше слово вихідного операнда обнуляється:

pextrw r32/r64, mmx, imm8.

Приклад використання команди **pextrw** розглянуто в програмі з лістингу 18.16.

Лістинг 18.16. Приклад використання команди **pextrw**:

```
    .686
    .model  flat, stdcall
    .XMM
    option casemap:none
    include \masm32\include\windows.inc ; файлы структур, констант …
    include \masm32\macros\macros.asm
    uselib kernel32, user32
.data
    mas1 dw 10, 20,30,40     ;
    mas2 dw 100,200,300,400  ;
    res dd 4 dup(0),0
    _title db "MMX-команда pextrw",0
    _info db " Использование команды MMX - pextrw",0Ah,0Dh,0Dh,
```

```
    "Задание: распаковать из чисел первого массива первое слово,",0Ah,0Dh,
    "а из второго массива второе слово, результат записать в новый
массив.",0Ah,0Dh,0Dh,
    "Результирующий массив:",0Ah,0Dh
    _buf dd 128 dup (?),0
    _ifmt   db "%d, %d",0 ; %X - hex, %d - dec
.code
_start:
    lea esi,mas1 ; адреса mas1
    lea edi,mas2 ; адреса mas2
    lea ebx,res    ; адреса результатів
     movq MM0,qword ptr [ESI] ; заносимо в MM0 елемент mas1
     movq MM1,qword ptr [EDI] ; заносимо в MM1 елемент mas2
        pextrw eax,MM0,0     ; розпаковування MM0 (з MM0 перше слово в eax)
        mov res,eax     ; зберігаємо результат
        pextrw eax,MM1,1          ; розпаковування (з MM1 в eax)
    mov res+4,eax   ; зберігаємо результат
    invoke wsprintf, ADDR _buf, ADDR _ifmt, res, res+4;
    invoke MessageBox, 0, ADDR _info, ADDR _title, 0
    invoke ExitProcess, 0
    end _start end _start
```

Результат виконання програми наведено на рис. 18.22.

Рис. 18.22. Результат виконання програми

Команда **pinsrw** вставляє слово в одне з чотирьох упакованих слів вихідного операнда. Вихідним операндом є один з MMX-регістрів, а вхідним операндом може виступати один з 32-розрядних регістрів загального призначення, молодше слово якого буде вставлене у вихідний операнд.

Номер позиції, куди поміщається операнд, визначається молодшими двома бітами маски і може набувати значень від 0 до 3:

pinsrw mmx, r32/mem16, imm8.

Команди **pmaxub, pmaxsw** обчислюють максимальне значення з кожної пари упакованих елементів у вихідному і вхідному операндах. Операції можуть виконуватися як над беззнаковими байтами (**pmaxub**), так і над знаковими словами (**pmaxsw**). Вихідним операндом може виступати MMX-регістр, а вхідним – MMX-регістр або 64-розрядний елемент пам'яті:

pmaxub/w mmx_i, mmx_j/mem64.

Приклад використання команди pmaxsw розглянуто в програмі з лістингу 18.17. В лістингу порівнюються елементи двох масивів та формується новий масив з більших значень однойменних елементів масивів.

Лістинг 18.17. Формування масиву з більших значень двох масивів:

```
        .686
        .model  flat, stdcall
        .MMX
        .xmm
        option casemap:none
        include \masm32\include\windows.inc  ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32,masm32
.data
     mas1 dw 12, 32,-4,6, 120,126,-33,88,80, 10,-55,65,123,-74,33,77,16,94,1, 24
     mas2 dw 42,-123,3,21,32, -4, 32, 11,75,-44, 55,29, 12, 32,-3,-4,65,13,-46,123
     len1 EQU ($-mas2)/type mas2; обчислення кількості чисел масиву mas2
     res1 dd len1 dup(?),0
     frmt  db " %d   ",0
     titl1  db "Параллельная обработка массива слов командой MMX pmaxsw",0
     info1 db "Результат  параллельной обработки командой pmaxsw:",10,13,13
     buf dw ?
.code
_st:
              mov eax, len1/4      ; підготування до порівняння
              mov ecx, eax         ; занесення вмісту eax у ecx
              lea esi, mas1        ; занесення у esi адреси mas1
              lea edi, mas2        ; занесення у edi адреси mas2
              lea ebx, res1     ; занесення у ebx адреси res
m1:
   movq MM0, qword ptr [esi]  ; занесення восьми байтів з mas1
   movq MM1, qword ptr [edi]  ; занесення восьми байтів з mas2
pmaxsw MM0,MM1  ; паралельне порівняння
   movq qword ptr [ebx], MM0; занесення вмісту MM1 у res
              add esi,8  ; зміщення вказівника на mas1 на 8 байтів
              add edi,8  ; зміщення вказівника на mas2 на 8 байтів
              add ebx,8  ; зміщення вказівника на res1 на 8 байтів
   loop m1
              lea esi,res1
              lea edi,buf
              mov ecx,len1
@@:
              push ecx
              movzx eax,word ptr [esi]
              invoke wsprintf, addr [edi], addr frmt,eax
              ;sub esp,0ch
              pop ecx
              add edi, eax
              add esi, 2
              dec ecx
              jnz @b
invoke MessageBox, 0, ADDR info1, ADDR titl1, MB_OK
              invoke ExitProcess, 0
        end _st
```

Результат виконання програми наведено на рис. 18.23.

Рис. 18.23. Результат виконання програми

Команди **pminub, pminsw** обчислюють мінімальне значення з кожної пари упакованих елементів у вихідному і вхідному операндах. Операції можуть виконуватися як над беззнаковими байтами (**pminub**), так і над знаковими словами (pminsw). Вихідним операндом може виступати ММХ-регістр, а вхідним – ММХ-регістр або 64-розрядний елемент пам'яті:

pminub/w mmx_i, mmx_j/mem64.

Приклад використання команди pminub розглянуто в програмі з лістингу 18.18.

Лістинг 18.18. Приклад використання команди pminub:

```
        .686                            ; директива определения типа микропроцессора
        .model flat,stdcall  ; задание линейной модели памяти
        .XMM
        option casemap:none                 ; отличие строчных и прописных букв
        include \masm32\include\windows.inc    ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32
.data
   mas1 Dw 1,5,7,98,52,12,36,45
   mas2 Dw 2,3,89,5,14,23,6,58
   len1 EQU ($-mas2)/type mas2
   sum dw len1 DUP(0),0
   sum2 DD len1 DUP(0),0
   _title db "Модуль 1", 0
   zero dd 0.0
   buf dd len1 DUP(0),0
   ifmt  db"Использование  команды  pminub.  Команда  находит  минимальный  элемент",0Ah,0Dh,0Dh,
   " Ответ=", len1 dup(" %d "),0
.code
_st:
            mov EAX, len1
            mov EBX, 4
            xor EDX, EDX
            div EBX
            mov ECX, EAX
            lea ESI, mas1
            lea EDI, mas2
            lea ebx, sum
@@:  movq XMM0, qword ptr[ESI]
        movq XMM1, qword ptr[edi]
pminub XMM0,XMM1  ; якщо порівняти неможливо, то повертається 0
        movq qword ptr[EBX], XMM0
        add ESI, 8
        add EDI, 8
        add EBX, 8
  loop @b
        mov ebx,0
        mov edx,0
        mov ecx, len1
m2:  mov ax,sum[ebx]
        movzx eax,ax
        mov sum2[edx],eax
```

```
        add ebx,2
        add edx,4
  loop m2
invoke wsprintf,ADDR buf, ADDR ifmt, sum2,sum2[4],sum2[8],sum2[12], sum2[16], sum2[20],
sum2[24],sum2[28],sum2[32]
        invoke MessageBox, 0, addr buf, addr _title, MB_ICONINFORMATION
        invoke ExitProcess, 0
        end _st
```

Результат виконання програми наведено на рис. 18.24.

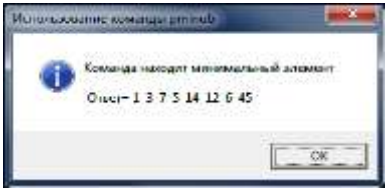


Рис. 18.24. Результат виконання програми

Команда **pmovmskb** формує байт, що містить знакові біти восьми байтів, які містяться у вхідному операнді. Як вхідний операнд може виступати один з MMX-регістрів. Вихідним операндом є 32-розрядний регістр загального призначення, молодший байт якого міститиме результат. Ця команда дуже зручна для формування умовних галужень в програмах:

pmovmskb r32/r64, mmx/mem64.

Команда **psadbw** обчислює сумарну різницю значень беззнакових байтів вхідного і вихідного операндів. Вхідним операндом може виступати MMX-регістр або 64-разрядний елемент пам'яті, а вихідним – один з MMX-регістрів:

psadbw mmx_i, mmx_j/mem64.

Команда **psignb/w/d** виконує перемноження поля з src1 на {-1,0,1} залежно від знаку відповідного поля в src2 (множення на 0 коли поле в src2 дорівнює нулю):

psignb/w/d mmx_i, mmx_j /mem64.

18.8. MMX-команди в Pentium IV

З появою Pentium 4 MMX-команди дістали доступ до 128-бітових регістрів (xmm). В цих командах першим операндом записується приймач інформації (dest), а другим – джерело (src).

Наведемо приклади MMX-команд та їх опис:

- **paddq** xmm,xmm/mem128 – складання двох 128-бітових операндів;
- **psubq** xmm,xmm/mem128 – віднімання 128-бітових операндів;

- **pmuludq** xmm,xmm/mem128 – множення 64-бітових операндів. Результат не повинен перевищувати 128-бітовий розмір;
- **pslldq** xmm, imm8 – логічний зсув вмісту вліво на imm8 бітів;
- **psrldq** xmm, imm – логічний зсув вмісту вправо на imm8 бітів;
- **pshufhw** xmm xmm/mem128, imm – пересилання з перегрупуванням чотирьох 16-бітових слів з молодшої половині dest в молодшу половину src. Перегрупування задається вмістом константи imm;
- **pshuflw** xmm xmm/mem128, imm – пересилання з перегрупуванням чотирьох 16-бітових слів із старшої половини dest в старшу половину src. Перегрупування задається вмістом константи imm;
- **pshufd** xmm, xmm /mem128, imm – пересилання з перегрупуванням чотирьох 32-бітових слів з dest в src. Перегрупування задається вмістом константи imm;
- **punpckhqdq** xmm, xmm /mem128 – у dest записується вміст старших половин src(xmm/m128) та dest;
- **punpcklqdq** xmm, xmm/mem128 – у dest (xmm) записується вміст молодших половин src (xmm/m128) та dest;
- **movdq2q** mmx, xmm – молодша половина xmm копіюється в mmx;
- **mov2qdq** xmm, mmx – вміст регістра mmx копіюється в молодшу половину xmm;
- **movntdq** mem128,xmm – пересилання вмісту 128-бітового регістра в пам'ять без кешування. Адреса повинна бути кратна 16;
- **movdqa** xmm, xmm/m128 – пересилання 128-бітового коду;
- **movdqa** xmm/m128,xmm – пересилання даних з адресою, яка кратна 16;
- **movdqu** xmm/mem128 – пересилання 128-бітового коду з пам'яті;
- **movdqu** xmm/mem128,xmm – пересилання даних, які можуть не мати 16-бітового вирівнювання;
- **movmskpd** r32, xmm – копіювання вмісту знакових розрядів (63 і 127) в біти 0 і 1 регістра r32. Решта бітів регістра очищається;
- **maskmovdqu** xmm, xmm – пересилання за маскою. Перший операнд xmm (src) містить код, що пересилається, другий операнд xmm (dest) – маску пересилання. Адреса третього операнда (куди проводитиметься пересилка) повинна знаходитися в DS:DI або в DS:EDI. Якщо знаковий (старший) розряд 1-го числа маски встановлений, то dest[i] = src[i]; якщо знаковий розряд байта маски очищений, то вміст dest не змінюється.

18.9. Лабораторна робота "MMX-команди"

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм з використанням команд MMX, написаних мовою асемблера для програмування МП платформи x86.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати свій варіант та написати на асемблері програму обчислення одного з виразів з використанням цілих чисел за допомогою команд MMX, з виведенням результату обчислень в одне спрощене віконце за допомогою функції MessageBoxIndirect та прізвище й ініціали автора програми і свій e-mail.

Вхідний контроль знань

1. Виконати паралельне додавання 2-х масивів з 18 знакових даних розміром у байт.
2. Виконати паралельне віднімання 2-х масивів з 19 знакових даних розміром у байт.
3. Виконати паралельне пакування 2-х масивів з 10 знакових даних розміром у байт.
4. Виконати паралельне пакування 2-х масивів з 11 знакових даних розміром у слово.
5. Виконати паралельне пакування 2-х масивів з 10 знакових даних розміром у подвійне слово.
6. Виконати паралельне множення 2-х масивів з 11 знакових даних розміром у слово.
7. Виконати паралельну логічну операцію ”Виключне *АБО*” 2-х масивів по 15 знакових чисел.
8. Виконати командою pmulhw операцію над двома масивами.
9. Виконати командою pmullw операцію над двома масивами.
10. Виконати командою pmaddwd операцію над двома масивами.
11. Виконати паралельну логічну операцію *АБО* 2-х масивів з 14 знакових чисел.
12. Виконати паралельну порозрядну інверсію та множення командою pandn 2-х масивів з 13 знакових даних.
13. Виконати паралельне логічне множення 2-х масивів по 11 знакових чисел.

Завдання

1. Ввести з клавіатури два дійсних числа та виконати їх додавання. Якщо результат позитивний, то виконати паралельне порівняння за допомогою MMX-команд 2-х масивів з 10 знакових цілих чисел розміром у байт. Якщо один масив більший від іншого, то виконати операцію

$a - e/c - ab$, де $a = 0{,}2$; $b = 10{,}05$; $c = 2{,}3$; $e = 21{,}07$;

інакше – виконати операцію ab.

2. Ввести з клавіатури два дійсних числа та виконати їх перемноження. Якщо результат менше 5, то виконати паралельне порівняння за допомогою ММХ-команд 2-х масивів по 10 знакових цілих чисел розміром у байт. Якщо один масив більший від іншого, то виконати операцію

$$(a - c)b + c/a, \text{ де } a = 0{,}2;\ b = 10{,}05;\ c = 2{,}3;$$

інакше – виконати операцію $a - c$.

3. Ввести з клавіатури два дійсних числа та виконати їх додавання. Якщо результат негативний, то виконати паралельне порівняння за допомогою ММХ-команд 2-х масивів по 11 знакових цілих чисел розміром у байт. Якщо один масив більший від іншого, то виконати операцію

$$(a - e)b - d/b, \text{ де } a = 0{,}1;\ b = 1{,}05;\ c = 2{,}1;\ d = 3{,}2;$$

інакше – виконати операцію d/b.

4. Ввести з клавіатури два дійсних числа та виконати їх додавання. Якщо результат перевищує число 7, то виконати операцію паралельного логічного додавання за допомогою ММХ-команд над масивами цілих чисел. Якщо друге слово більше 55, то виконати операцію

$$a - e/b - de, \text{ де } a = 0{,}2;\ b = 8{,}05;\ c = 2{,}2;\ d = 3{,}3;$$

інакше – виконати операцію $a - e/b$.

5. Ввести з клавіатури два дійсних числа та виконати їх віднімання. Якщо результат позитивний, то виконати операцію паралельного логічного множення за допомогою ММХ-команд над масивами цілих чисел. Якщо друге слово більше 45, то виконати операцію

$$(b + d)/ba - c, \text{ де } a = 0{,}2;\ b = 7{,}05;\ c = 2{,}3;\ d = 3{,}4;$$

інакше – виконати операцію $b + d$.

6. Ввести з клавіатури два дійсних числа та виконати їх віднімання. Якщо результат негативний, то виконати операцію паралельного додавання pxor за модулем за допомогою ММХ-команд над масивами 11 цілих чисел. Якщо друге слово менше 15, то виконати операцію

$$(b - dc)/a - c, \text{ де } a = 0{,}2;\ b = 6{,}05;\ c = 1{,}3;\ d = 3{,}5;$$

інакше – виконати операцію $b - dc$.

7. Ввести з клавіатури дійсне число та добути з нього корінь. Якщо результат менше 7, то виконати команду логічного зсуву ліворуч pslld масиву з 10 чисел та залежно від значення другого байта виконати операцію

$$ae - d/c - b, \text{ де } a = 1{,}2;\ b = 5{,}05;\ c = 0{,}3;\ d = 3{,}5;$$

інакше – виконати операцію d/c.

8. Ввести з клавіатури дійсне число. Якщо число позитивне, то виконати команду арифметичного зсуву елементів даних праворуч psraw масиву з 9 чисел та залежно від значення другого байта виконати операцію

$$(ab - d)/b - d, \text{ де } a = 3{,}6;\ b = 9{,}17;\ c = 2{,}8;\ d = 3{,}9;$$

інакше – виконати операцію $ab - d$.

9. Ввести з клавіатури дійсне число. Якщо число позитивне, то виконати команду логічного зсуву елементів даних праворуч psrlw масиву з 9 чисел та залежно від значення другого байта виконати операцію: (de – e)/(c – a), де $a = 3{,}1;\ b = 9{,}11;\ c = 2{,}3;$ d = 5,5;

інакше – виконати операцію de – e.

10. Ввести з клавіатури дійсне число. Якщо число більше 10, то виконати паралельне пакування за допомогою MMX-команд 2-х масивів по 11 чисел та залежно від значення другого байта виконати операцію

$$a/b - cd,$$ де $a = 4{,}5$; $b = 12{,}3$; $c = 2{,}5$; $d = 6{,}5$;

інакше – виконати операцію *a/b*.

Зміст звіту

1. Постановка задачі для конкретного варіанта.

2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.

3. Лістинг програми з виведенням даних на екран монітора з використанням API-функцій під Win32 та детальним коментарем і описом роботи.

4. Print screen екрана 32-розрядного налагоджувача з виконаною програмою.

5. Короткий опис виконання програми.

6. Висновки за результатами роботи.

Приклад 18.1. Виконати паралельне порівняння 2-х масивів по 8 знакових даних розміром у байт. Якщо старші половини масивів однакові, то виконати операцію

$$a - b/4,$$ де $a = 0{,}2$; $b = 10{,}05$;

інакше – операцію не виконувати.

Програму обчислення прикладу наведено у лістингу 18.19.

Лістинг 18.19. Програма з виведенням результату у спрощене вікно:

```
        title CopyRight by Rysovaniy A. N.
        .686                    ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        .MMX                            ; директива визначення команд MMX
      option casemap: none      ; отличие строчных и прописных букв
      include \masm32\include\windows.inc   ; файлы структур, констант …
      include \masm32\macros\macros.asm
      uselib kernel32, user32, fpu;
.data                                   ; директива определения данных
        str1  DB -1, -2, 3, 4, 0, 1, 12, -8
        str2  DB -1, -2, 3, 4, 0, 1, 12, -8
        res  DB 8 DUP (5)
        a1 DWORD  0.2
        b1 DWORD  10.05
        c1 DWORD  4.
        res2 DD ?
      st1 db "result",0             ; назва віконця
      st2 db 10 dup(?),0              ; буфер чисел для виведення повідомлення
.code                                   ; директива начала кода программы
  _st:                                  ; метка начала программы с именем_st
        movq    MM0, qword ptr str1     ; копіювання 64-розрядного числа
        pcmpeqb MM0, qword ptr str2
        movq    qword ptr res, MM0              ; збереження результату
      emms                              ; остання MMX-команда (для співпроцесора)
```

```
        add dword ptr res,0          ; порівняння старшої частини res
        jz m1                ; перехід на m1, якщо старша частина дорівнює нулю
    finit                    ; ініціювання співпроцесора
        fld b1               ; завантаження у вершину стека ST(0) змінної b1
        fld c1               ; завантаження у вершину стека ST(0) змінної c1
        fdiv                 ; st(0) := st(1)/st(0)
        fld a1               ; завантаження у вершину стека ST(0) змінної a1
        fsub st(0),st(1)             ; st(0) := st(0) – st(1)
        ;fst res2                          ; можливе збереження у пам'яті
m1:
invoke FpuFLtoA, \           ; API-функція перетворення 80-розрядного числа
 0, \                        ; адреса числа, що відображається
  10, \                      ; кількість десяткових знаків після коми (10)
   ADDR st2, SRC1_FPU or SRC2_DIMM ; адреса 80-розрядного числа та саме число
invoke MessageBox, 0, addr st2, \ ; адреса рядка, яка містить  текст повідомлення
       addr st1, \ ; адреса рядка, яка містить  заголовок повідомлення
          MB_OK                  ; вигляд діалогового вікна
invoke ExitProcess, 0 ; повернення управління ОС та вивільнення ресурсів
       end _st                   ; директива окончания программы
```

Рис. 18.25. Результат виконання програми з лістингу 18.19

Результат виконання програми, отриманий за допомогою API-функції MessageBox, наведено у вигляді вікна на рис. 18.25.

За назву вікна відповідає параметр addr st1.

У випадку необхідності порівняння всього масиву з 8-ма чисел, розміром у байт можна використати порівняння за два етапи. Такий прийом здійснено у прикладі 18.2.

Приклад 18.2. Виконати паралельне порівняння 2-х масивів по 8 знакових цілих чисел розміром у байт кожне. Якщо один масив більший від другого, то виконати операцію

$$(a - c)/b + c, \text{ де } a = 0{,}2;\ b = 10{,}05;\ c = 2{,}3;$$

інакше – виконати операцію $a - c$.

Лістинг 18.20. Програма обчислення прикладу 18.2:

```
        title CopyRight by Rysovaniy A. N.                rysov@rambler.ru
        .686          ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
        .MMX                  ; директива визначення команд MMX
option casemap: none        ; отличие строчных и прописных букв
      include \masm32\include\windows.inc ; файлы структур, констант …
      include \masm32\macros\macros.asm
      uselib kernel32, user32, fpu;
      BSIZE equ  30                 ; задание реального количества байтов
.data             ;
str1  db -1, -2, 3, 2, 5, 1, 12, -8 ; резервування в пам'яті комірок для чисел масиву str1
str2  db -1, -2, 3, 2, 0, 1, 12, -8 ;
  res  DD 8 DUP (0)    ; резервування в пам'яті комірок для res
  a1 DWORD  0.2        ; константа a
  b1 DWORD  10.05      ; константа b
  c1 DWORD  2.3        ; константа c
   stdout DWORD ?      ; для збереження дескриптора виведення
   cWritten  DWORD  ? ; для адреси символів виведення
buf  BYTE BSIZE dup (?) ; кількість байтів для запису послідовності символів
```

```
res2 DWORD ?
.code          ; директива начала кода программы
  _st:         ; метка начала программы с именем_st
       lea esi,str1          ; занесення в esi адреси масиву str1
       lea edi,str2          ; занесення в edi адреси масиву str2
       movq    MM0, qword ptr [esi]          ; занесення 8-ми байтів до Рг MM0
       pcmpeqb MM0, qword ptr [edi]          ; перевірка на рівняння
       movq  qword ptr res, MM0          ; збереження результату порівняння
       emms                                ; визвільнити  регістри MMX
    cmp dword ptr res,0              ; порівняння старшої частини результату
      jnz m1                         ; перейти на m1, якщо результат не нульовий
    cmp dword ptr res+4,0       ; порівняння молодшої частини результату
      jnz m1                          ; перейти на m1, якщо результат не нульовий
  finit                             ; ініціалізація співпроцесора
    fld c1                      ; занесення у вершину стека st(0) змінної c1
    fld b1                      ; занесення у вершину стека st(0) змінної b1
    fld a1                      ; занесення у вершину стека st(0) змінної a1
      fsub st(0),st(2)     ; a – c  та збереження в st(0)
      fdiv st(0),st(1)     ; (a – c)/b  та збереження в st(0)
      fadd st(0),st(2)     ; (a – c)/b + a  та збереження в st(0)
      fst res2              ; збереження вершини стека в пам'яті з ім'ям res2
      jmp m2                 ; безумовний перехід на мітку m2
m1:  finit                  ; ініціалізація співпроцесора
    fld c1                          ; занесення у вершину стека st(0) змінної c1
    fld a1                          ; занесення у вершину стека st(0) змінної a1
      fsub st(0),st(1)      ; a1 – c1 та збереження результату у вершині стека
      fst res2               ; збереження вершини стека в пам'яті з ім'ям res2
m2:  invoke  GetStdHandle, STD_OUTPUT_HANDLE
    mov  stdout, eax       ; збереження одержаного дескриптора у пам'яті
invoke FpuFLtoA, 0, 10,\ ADDR buf, \  ; адреса буфера для символів, які перетворяться
    SRC1_FPU or SRC2_DIMM    ; адреса 80-розрядного числа та саме число
invoke  WriteConsoleA, stdout, \    ; дескриптор стандартного пристрою виведення
    ADDR buf, BSIZE, \ ; адреса початку повідомлення та розмір повідомлення
        ADDR cWritten,0    ; адреса, де зберігається кількість символів
invoke ExitProcess, 0 ; повернення управління ОС та вивільнення ресурсів
       end _st          ; директива окончания программы
```

Правильність отриманих результатів підтверджено результатами, отриманими на калькуляторі.

Результат виконання програми наведено на рис. 18.26.

Рис. 18.26. Результат виконання програми з лістингу 18.20

Приклад 18.3. Виконати операцію паралельного додавання за допомогою ММХ-команд над масивами цілих чисел. Якщо друге слово більше 55, то виконати операцію

$$a - e/b - de,$$ де $a = 0{,}2$; $b = 8{,}05$; $c = 2{,}2$; $d = 3{,}3$;

інакше – виконати операцію $a - e/b$.

Лістинг 18.21:

```
        title Rysovaniy A. N. & Gerasimovich N., НТУ "ХПІ",  КІТ-27v
      ;    a - c/b - dc;  a = 0,2; b = 8,05; c = 2,2; d = 3,3;
      ;    a - c/b   if ( second word < 55 )
      .686                    ; директива определения типа микропроцессора
      .model flat,stdcall  ; задание линейной модели памяти
      .MMX                                    ; директива визначення команд MMX
      option casemap: none                ; отличие строчных и прописных букв
      include \masm32\include\windows.inc   ; файлы структур, констант …
      include \masm32\macros\macros.asm
         uselib kernel32, user32, fpu
          fpuDiv macro _a,_c,_b          ; макрос з ім'ям fpuDiv
             fld _c
             fdiv _b
             fld _a
             fsubr
          endm                                ;; закінчення макросу
.data                                         ; директива определения данных
    _a REAL4 0.2
    _b REAL4 8.05
    _c REAL4 2.2
    _d REAL4 3.3
    arr1 WORD 1,2,3,4                   ; масив чисел arr1 розміром у слово
    len1 equ ($-arr1)/type arr1 ; кількість чисел масиву
    arr2 WORD 5,6,7,5                   ; масив чисел arr2 розміром у слово
    len2 equ ($-arr2)/type arr2 ; кількість чисел масиву
    arr1_2 BYTE (len1+len2) dup(0) ;  розмір буфера для чисел масивів
    st1 db "FPU-MMX Result",0                    ; назва віконця
   info db "x1 = "                    ; підпис числа для віконця результату
   st2 db 10 dup(?),0                 ; буфер чисел для виведення повідомлення
.code                                         ; директива начала кода программы
_st:                                          ; метка начала программы с именем_st
    movq MM1,QWORD PTR arr1           ; завантаження масиву чисел arr1
    movq MM2,QWORD PTR arr2           ; завантаження масиву чисел arr2
    paddb MM1,MM2          ; паралельне циклічне додавання чисел масивів
    movq QWORD PTR arr1_2,MM1      ; збереження результату додавання
.if((WORD ptr arr1_2+2)>55)  ; якщо друге слово більше 55
    emms
    fpuDiv [_a],[_c],[_b]
    fld _d
    fmul _c
    fsub
    invoke FpuFLtoA, 0, 10, ADDR st2, SRC1_FPU or SRC2_DIMM
    invoke MessageBox, NULL, addr st2, addr st1, MB_ICONQUESTION
.else                    ; інакше
    emms                                          ; остання MMX-команда
    fpuDiv [_a],[_c],[_b]
    invoke FpuFLtoA, 0, 10, ADDR st2, SRC1_FPU or SRC2_DIMM
    invoke MessageBox, NULL, addr info, addr st1, MB_ICONINFORMATION
.endif                   ; закінчення директиви .if
invoke ExitProcess, 0 ; повернення управління  ОС Windows та вивільнення ресурсів
        end _st                                   ; директива окончания программы
```

Результат виконання програми з лістингу 18.21 наведено на рис. 18.28.

Рис. 18.28. Результат виконання програми з лістингу 18.21

Для надання інформаційності спрощеного віконця використано параметр MB_ICONINFORMATION. Підпис числа результату "x1 = " виконано за допомогою послідовності команд:

```
info db "x1 = "                    ; підпис числа для віконця результату
st2 db 10 dup(?),0        ; буфер чисел для виведення повідомлення
```

Виклик підпису з результатом виконується функцією MessageBox параметром addr info. А у зв'язку з тим, що в імені info відсутній 0 на кінці, виконується зчитування й наступного буфера, де безпосередньо розташовується результат операції.

Контрольні запитання

1. Яке призначення ММХ-команд?
2. З яких груп команд складаються ММХ-команди?
3. Які особливості спільного використання ММХ-команд та команд співпроцесора?
4. Які способи арифметичної обробки чисел використовують ММХ-команди?
5. Що означає суфікс у більшості ММХ-команд?

19. SSE-РОЗШИРЕННЯ

У 3D-графіці часто зустрічаються групи операцій, які можна виконати за один такт за допомогою SIMD-команд (single instruction, multiple data, тобто одна команда – багато даних). Такими операціями є інтерполяція векторів, скалярне множення векторів, нормування векторів, інтерполяція компонент кольору (наприклад, RGB) і так далі. До SIMD-команд належать й команди SSE-розширення.

Технологія SSE дозволила подолати 2 основних проблеми MMX: при використанні MMX неможливо було одночасно використовувати інструкції співпроцесора, оскільки його регістри були загальними з регістрами MMX, і можливість MMX працювати тільки з цілими числами.

SSE-розширення з'явилося в процесорах Intel Pentium III і доповнює MMX-розширення засобами обробки даних з плаваючою точкою (http://www.intel.com/design/pentium4/manuals/ index_new.htm #aorm). SSE-команди можна паралельно обробляти з командами математичного співпроцесора. Набір інструкцій SSE-розширення складається з 70 команд.

SSE включає в архітектуру мікропроцесора вісім 128-бітових регістрів і набір інструкцій, що працюють з скалярними і упакованими типами даних. Кожен регістр може містити **чотири 32-бітові значення з плаваючою точкою одинарної точності.**

SSE-розширення реалізоване як апаратно-програмний модуль, який включає додаткові вісім регістрів розрядністю в 128 бітів, що мають позначення XMM0 – XMM7 і 32-розрядний регістр управління /стану MXCSR.

Формат подання даних SSE-розширення з плаваючою точкою в короткому форматі (Single Precision Floating Point, SPFP) наведено на рис. 19.1.

Рис. 19.1. Формат даних SSE-розширень

Кожне 32-розрядне число з плаваючою комою має 1 знаковий біт, 8 бітів порядку і 23 біти мантиси, що відповідає стандарту IEEE-754 на формат подання чисел одинарної точності з плаваючою комою.

Діапазон зміни чисел в SSE-форматі дорівнює $2^{-126} – 2^{127}$.

У зв'язку з тим, що формати команд співпроцесора та SSE-команд не однакові, в деяких випадках при різних межах вирівнювання результати обчислень з використанням форматів FPU і SSE можуть розрізнятися.

Структура полів регістра управління /стану (MXCSR) багато в чому нагадує структуру, що реалізована в регістрах стану (swr) і управління (cwr) математичного співпроцесора. Станом обчислень можна керувати шляхом установлення певних значень в поля цього регістра.

Для завантаження і збереження MXCSR використовуються команди:

LDMXCSR m32 – завантажити MXCSR з пам'яті;

STMXCSR m32 – зберегти MXCSR у пам'ять;

FXRSTOR m512bytes – завантажити з пам'яті середовище виконання (всі регістри) FPU, MMX, SSE;

FXSAVE m512bytes – зберегти в пам'ять середовище виконання (всі регістри) FPU, MMX, SSE.

При виникненні виняткових ситуацій встановлюються біти 0 – 5 в регістрі управління /стану (MXCSR). Формат полів регістра MXCSR і їх призначення наведено на рис. 19.2.

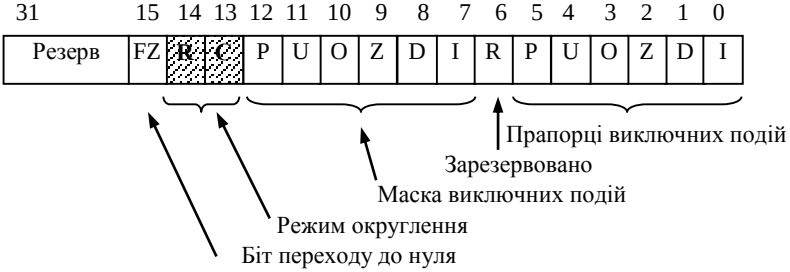


Рис. 19.2. Регістр управління /стану команд SSE-розширень

Кожна виняткова ситуація може бути замаскована шляхом установлення в 1 бітів 7 – 12 регістра MXCSR. Якщо яке-небудь виключення замасковане, то воно обробляється процесором за стандартним алгоритмом, після чого продовжується виконання програмного коду.

Найбільше значення в регістрі управління /стану команд SSE-розширень мають біти 13 – 14 поля rc. За умовчанням встановлюється режим округлення до найближчого значення числа з плаваючою точкою в короткому форматі.

Ці біти можна встановити програмно, причому можливі такі комбінації:

- 00 – округлення до найближчого числа;
- 01 – округлення до меншого числа;
- 10 – округлення до більшого числа;
- 11 – округлення з відкиданням дробової частини.

Біт 15 використовується, якщо результат операції близький до нуля. При цьому процесор виконує такі дії:

– повертає значення 0 і знак результату;

– встановлює прапорці P (біт 5) і U (біт 4);

– маскує біти виключень.

Значна частина команд SSE може виконуватися в двох варіантах: ***скалярному*** і ***паралельному***. Це стосується арифметичних команд та команд порівняння.

Команди ***паралельних*** арифметичних операцій обробляють одночасно 4 подвійних слова і мають в своїй мнемоніці **суфікс ps**.

Команди ***скалярних*** операцій обробляють тільки молодші 32-розрядні подвійні слова упакованих операндів, не зачіпаючи при цьому три старших подвійних слова. Виняток становлять деякі команди скалярного пересилання даних. Мнемонічне позначення цих команд включає **суфікс ss**.

У процесі обробки даних команди SSE-розширення можуть порушувати виняткові ситуації, які виникають, якщо відбувається одна з наступних подій:

- некоректна операція (invalid operation) – I;
- денормалізований операнд (denormalized operand) – D;
- ділення на 0 (divide-by-zero) – Z;
- арифметичне переповнювання (numeric overflow) – O;
- втрата значущих розрядів (numeric underflow) – U;
- втрата точності (inexact result) – P.

Тип команд, які використовує мікропроцесор визначається відповідними розрядами:

```
SSE:      CPUID.01H:EDX[bit25]=1;  Streaming SIMD Extension
SSE2:     CPUID.01H:EDX[bit26]=1
SSE3:     CPUID.01H:ECX[bit0]=1
SSSE3:    CPUID.01H:ECX[bit9]=1
SSE4.1:   CPUID.01H:ECX[bit19]=1
SSE4.2:   CPUID.01H:ECX[bit20]=1
AVX:      CPUID.01H:ECX[bit28]=1;  Advanced Vector Extension
FMA:      CPUID.01H:ECX[bit12]=1;  Fused Multiply Add
```

Всі команди SSE, окрім спеціально обумовлених, при роботі з пам'яттю вимагають вирівнювання операнда на межу 16 байтів. Інакше виникає помилка виконання. Для вирівнювання даних компілятори надають спеціальне ключове слово

```
ALIGN 16 ; округляє лічильник інструкцій
a1   dword   ?
…
```

Для визначення підтримки SSE-команд можна скористатися програмою з лістингу 19.1.

Лістинг 19.1. Тестова програма для визначення підтримки SSE-команд:

```
.686                    ; директива определения типа микропроцессора
.model flat                              ; лінійна модель пам’яті
option casemap: none                ; отличие строчных и прописных букв
.data                                    ; директива определения данных
    testSSE DB 1
```

```
.code                                   ; директива начала кода программы
st:                                     ; метка начала программы с именемst
    mov   EAX, 1
    cpuid                               ; ідентифікація мікропроцесора
    test  EDX, 2000000h    ; edx:= edx ∧ 200 0000h  (25 розряд)
    jnz   exit                          ; перейти на exit, якщо не нуль
    mov  testSSE, 0
exit:  xor   EAX, EAX
    mov  AL, testSSE
    ret                                 ; повернення управління  ОС
    end st                              ; завершение программы с именемst
```

Програма з лістингу 19.1 повертає 1 у регістр AL, якщо технологія SSE підтримується процесором.

Команди SSE-розширення можна розділити на декілька груп:

- команди для чисел з плаваючою точкою;
- команди для цілих чисел;
- інші команди.

Команди для чисел з плаваючою точкою:

– команди пересилання:

- скалярні типи: MOVSS;
- упаковані типи: MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS;

– арифметичні команди:

- скалярні типи: ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS;
- упаковані типи: ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS;

– команди порівняння:

- скалярні типи: CMPSS, COMISS, UCOMISS;
- упаковані типи: CMPPS;

– перемішування і розпаковування:

- упаковані типи: SHUFPS, UNPCKHPS, UNPCKLPS;

– команди для перетворення типів:

- скалярні типи: CVTSI2SS, CVTSS2SI, CVTTSS2SI;
- упаковані типи: CVTPI2PS, CVTPS2PI, CVTTPS2PI;

– бітові логічні операції:

- упаковані типи: ANDPS, ORPS, XORPS, ANDNPS.

Команди для цілих чисел:

– арифметичні команди: PMULHUW, PSADBW, PAVGB, PAVGW, PMAXUB, PMINUB, PMAXSW, PMINSW;

– команди пересилки: PEXTRW, PINSRW;

– інші: PMOVMSKB, PSHUFW.

Інші команди:

– робота з регістром MXCSR: LDMXCSR, STMXCSR;

– управління кешем і пам'яттю: MOVNTQ, MOVNTPS, MASKMOVQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE.

19.1. Команди пересилання даних

MOVAPS приймач, джерело	Пересилання вирівняних по 16-байтовій межі 4 × 32-розрядних операндів

Команда **movaps** пересилає вирівняні по 16-байтовій межі 4 × 32-розрядних операнди. Як вхідний операнд може бути XMM-регістр або 128-розрядний елемент пам'яті; вихідним операндом може бути один з XMM-регістрів. Якщо адреса елемента пам'яті не буде вирівняна по 16-байтовій межі, то відбудеться виключення загального захисту:

```
movaps  xmm, xmm/mem128;
movaps xmm/mem128, xmm.
```

MOVUPS приймач, джерело	Пересилання невирівняних по 16-байтовій межі 4 × 32-розрядних операндів

Команда **movups** пересилає невирівняні дані; як вхідний операнд може бути XMM-регістр або 128-розрядний елемент пам'яті; вихідним операндом може служити один з XMM-регістрів. Команда не вимагає вирівнювання даних по 16-байтовій межі:

```
movups  xmm, xmm/mem128;
movups xmm/mem128, xmm.
```

Розглянемо приклад:

```
mas1  DD 1.,2.0,3.,4. ; резервування комірок пам'яті для масиву mas1
...
movups XMM0, mas1; занесення
```

Массив mas1 зберігає в пам'яті 32-розрядні дійсні числа, як наведено на рис. 19.3.

Address	Hex dump															
00403000	00	00	80	3F	00	00	00	40	00	00	40	40	00	00	80	40

Рис. 19.3. Вміст комірок пам'яті при зберіганні масиву mas1

За цією причиною за командою movups XMM0, mas1 до 128-розрядного регістра XMM0 завантажаться чотири 32-розрядні числа (рис. 19.4).

XMM0	4.000000,	3.000000,	2.000000,	1.000000

Рис. 19.4. Вміст 128-розрядного регістра XMM0 при зберіганні масиву mas1

MOVHPS приймач, джерело	Пересилання невирівняних старших 2 × 32-бітів xmm-регістра у вихідний

Команда **movhps** пересилає невирівняні старші 64 (2 × 32) біти з вхідного операнда у вихідний. Один з операндів обов'язково повинен бути XMM-

регістром, а другим може бути 64-розрядний елемент пам'яті. Пересилаються старші 64 біти із джерела в молодшу частину приймача. Молодші 64 біти обох операндів не змінюються. Якщо дані передаються з XMM-регістра, то пересиланню підлягають тільки старші 64 біти. Команда не вимагає вирівнювання по 16-байтовій межі адреси елемента пам'яті:

movhps xmm,mem64;
movhps mem64,xmm.

Схему виконання команди movhps наведено на рис. 19.5.

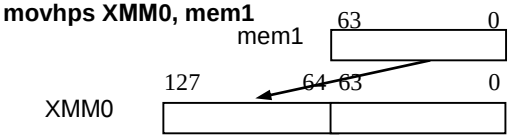


Рис. 19.5. Дія команди movhps XMM1, mem0

MOVHLPS приймач, джерело	Пересилання невирівняних старших 64 бітів з вхідного операнда у молодший вихідний

Команда **movhlps** пересилає невирівняні 64 біти з вхідного операнда у вихідний без зміни вмісту 64 старших розрядів вихідного регістра. Обидва операнди повинні знаходитися в XMM-регістрах. Пересилаються тільки старші 64 біти вхідного операнда. У результаті виконання цієї команди *змінюються молодші* 64 біти регістра-приймача:

movhlps xmm_i, xmmj.

Схему виконання команди movhlps наведено на рис. 19.6.

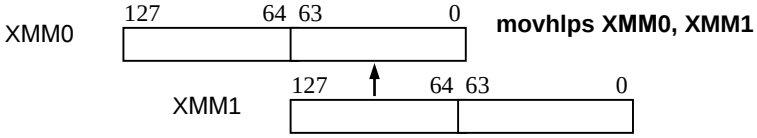


Рис. 19.6. Дія команди movhps XMM1, mem0

Розглянемо приклад використання команди **movlps** передачі даних (приклад 19.1).

Приклад 19.1. Навести програму, яка замінює старше слово одного масиву на молодше слово другого масиву.

Хай надано два масиви даних: 1., 3., 5., 7. та 0, 2., 4., 6. Сформувати масив даних 5., 7., 4., 6.

Процес отримання результату можна виконати за допомогою команди movhlps (рис. 19.7).

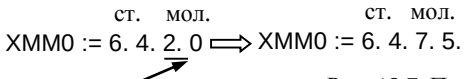


Рис. 19.7. Процес отримання результату командою movhlps

Лістинг 19.2. Приклад використання команд передавання даних:

```
        title Rysovaniy A.N.                                rysov@rambler.ru
        .686p                       ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти
        .XMM          ; директива визначення команд XMM-розширення
        option casemap:none             ; отличие строчных и прописных букв
        include \masm32\include\kernel32.inc ; системные функции применений…
        includelib \masm32\lib\kernel32.lib   ; виклик бібліотеки kernel32.lib
.data                                   ; директива определения данных
        mas1 DD 0, 2.,4.,6. ; резервування комірок пам'яті для масиву mas1
        mas2 DD 1.,3.,5.,7. ; резервування комірок пам'яті для масиву mas2
        res DD 4 DUP(0)     ; резервування комірок пам'яті для масиву res
.code                                   ; директива начала кода программы
_start:                                 ; директива начала кода программы
movups  XMM0, mas1 ; пересилання 4 × 32-розрядних чисел масиву mas1
movups  XMM1, mas2 ; пересилання 4 × 32-розрядних чисел масиву mas2
movhlps XMM0, XMM1 ; пересилання з обміном невирівняними 64 бітами
        movups  res, XMM0               ; запам'ятовування результату
        lea     EAX, res            ; запам'ятовування адреси масиву результату
invoke ExitProcess,0 ; повернення управління  ОС та визволення ресурсів
        end _start                  ; завершение программы с именем _start
```

Для того щоб у налагодувачі OllyDbg v.1.10 побачити регістри SSE, необхідно поставити мітку в пункті **Опции/Опции отладки/Registers/Decode SSE.** Результат виконання програми з лістингу 19.2 наведено на рис. 19.8.

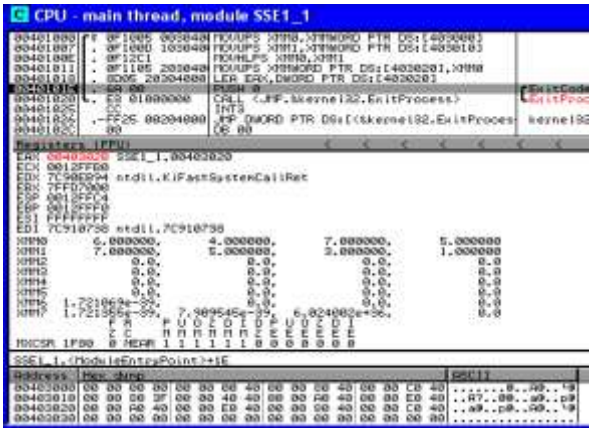


Рис. 19.8. Результат виконання програми з лістингу 19.2

Зверніть увагу на послідовність розташування операндів до виконання програми з обміном та після.

MOVLHPS приймач, джерело	Пересилання невирівняних молодших 64 бітів з вхідного операнда в старші вихідні

Команда **movlhps** пересилає невирівняні молодші 64 біти з вхідного операнда у вихідний. Обидва операнди повинні знаходитися в XMM-регістрах. Пересилаються тільки молодші 64 біти вхідних операндів.

У результаті виконання цієї команди *змінюються старші* 64 біти регістра-приймача:

movlhps xmm$_i$, xmmj.

Схему виконання команди movlhps наведено на рис. 19.9.

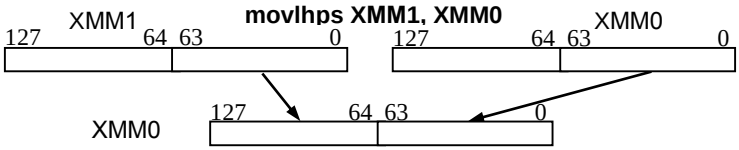


Рис. 19.9. Дія команди movhps XMM1, XMM0

Результат виконання фрагмента коду:

```
 .data                                  ; директива определения данных
mas1 DD -1.1,-2.1,-3.1,-4.1 ; резервування комірок пам'яті для mas1
mas2 DD 5.5,6.6,7.7,8.8     ; резервування комірок пам'яті для масиву mas2
.code                                   ; директива начала кода программы
 _start:                                ; директива начала кода программы
 movups  XMM0, mas1 ; пересилання 4 × 32-розрядних чисел масиву mas1
 movups  XMM1, mas2 ; пересилання 4 × 32-розрядних чисел масиву mas2
movlhps XMM0, XMM1  ; пересилання з обміном невирівняними 64 бітами
```

Результат дослідження команди **movlhps** наведено на рис. 19.10.

```
XMM0    -4.100000,    -3.100000,    -2.100000,    -1.100000
XMM1     8.800000,     7.700000,     6.600000,     5.500000
```

а – занесення чисел до регістрів

```
XMM0     6.600000,     5.500000,    -2.100000,    -1.100000
XMM1     8.800000,     7.700000,     6.600000,     5.500000
```

б – результат перетворення чисел

Рис. 19.10. Результат дослідження команди **movlhps**

MOVLPS приймач, джерело	Пересилання невирівняних молодших 64 бітів з вхідного операнда у вихідний

Команда **movlps** пересилає невирівняні 64 біти з вхідного операнда у вихідний (рис. 19.5). Один з операндів обов'язково повинен бути XMM-регістром, а другий може бути 64-розрядним елементом пам'яті. Пересилаються тільки молодші 64 біти вхідних операндів. Старші 64 біти обох операндів не змінюються. Якщо дані передаються з XMM-регістра, то пересиланню підлягають тільки молодші 64 біти. Команда не вимагає вирівнювання по 16-байтовій межі адреси елемента пам'яті:

movlps xmm$_i$, mem64;

```
movlps mem64, xmm.
```

MOVMSKPS приймач, джерело	Пересилання знакових бітів 32-розрядних слів

Команда **movmskps** пересилає знакові біти кожного з чотирьох упакованих чисел вхідного операнда в молодші чотири біти вихідного операнда. Як вхідний операнд може бути тільки XMM-регістр, а як вихідний операнд – 32-розрядний регістр загального призначення. Ця команда використовується для організації умовних переходів:

```
movmskps  r32, xmm;
movmskps  r64,xmm.
```

MOVSS приймач, джерело	Пересилання 32 молодших бітів

Команда **movss** пересилає 32 молодших біти з джерела в приймач, при цьому як мінімум один з операндів повинен бути XMM-регістром. Другий операнд повинен бути 32-розрядним елементом пам'яті. При виконанні операції пересилання з операнда в пам'ять молодші 32 біти поміщаються в молодші 32 біти XMM-регістра.

Якщо виконується пересилання з XMM-регістра, то вибираються молодші 32 розряди регістра, решта розрядів не змінюється:

```
movss xmm, xmm/mem32;
movss xmm/mem32, xmm.
```

19.2. Арифметичні команди

19.2.1. Команди додавання

ADDPS приймач, джерело	Паралельне додавання 128-розрядних операндів

Команда **addps** виконує паралельне додавання 128-розрядних операндів, при цьому як вхідний операнд (операнд-джерело) може бути один з XMM-регістрів або 128-розрядного елемента пам'яті. Вихідним операндом команди (операндом-приймачем) є один з XMM-регістрів:

```
addps  xmm, xmm/mem128.
```

ADDSS приймач, джерело	Скалярне складання операндів

Команда **addss** виконує скалярне складання операндів. Молодші подвійні слова операндів повинні бути числами з плаваючою точкою в короткому форматі (single precision). Результат поміщається в операнд-приймач, яким може бути XMM-регістр. Вхідним операндом можуть бути XMM-регістр або 32-розрядний елемент пам'яті:

```
addss xmm, xmm/mem32.
```

Лістинг 19.3. Паралельне додавання операндів двох масивів через тимчасове розміщення в регістрах XMM:

```
        title Rysovaniy A.N.                    rysov@rambler.ru
        .686p                ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти
        .XMM
        option casemap:none              ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32,fpu
.data
        mas1 dd  1.,2.,3.,4.
        mas2 dd 5.,6.,7.,8.
        len1 equ ($-mas2)/type mas2
        _title db "Исследование команды ADDSS",0 ; назва вікна
        _inf db "Скалярное сложение чисел массивов.",10,13,13,
         "Результат:",10,13
        _res DD len1 dup(0) ; масив для збереження результатів порівняння
.code
_st:    mov ecx,len1
        movaps XMM0, mas1
        movaps XMM1, mas2
        lea esi, _res
@@:
        addss xmm0,xmm1
        movups [esi], XMM0 ; запис результату у вихідний масив
        fld dword ptr [esi]
        invoke FpuFLtoA,0,2,ADDR [esi],SRC1_FPU or SRC2_DIMM ;
        add esi, len1
      shufps XMM1,XMM1,11111001b
      shufps XMM0,XMM0,11111001b
    loop @b
        invoke MessageBox, 0, addr _inf, addr _title, MB_ICONINFORMATION
        invoke ExitProcess, 0
        end _st
```

У програмі в лічильник ecx завантажується число цифр в масиві mas2. Це число дорівнює числу операцій складання над масивами mas1 та mas2.

У зв'язку з тим, що результати складання для організації виведення через MessageBox оброблятимуться скалярно, то запис в регістри XMM відбувається командами movaps:

```
movaps XMM0, mas1
movaps XMM1, mas2
```

Результат складання чисел масиву відбувається в циклі @@ – @b. Команда

```
addss xmm0,xmm1
```

здійснює складання тільки молодших чисел.

Для виведення чисел через MessageBox результат виконання додавання двох дійсних чисел заздалегідь записується у вершину стека співпроцесора, а потім перетворюється функцією FpuFLtoA.

Після кожної операції складання однойменних чисел масивів відбувається підготовка до наступної операції складання командою

```
add esi, len1,
```

яка фактично робить зсув адреси знаходження наступного числа на 4 байти (len1 = 4).

Команда shufps за маскою переміщає старші числа в молодші позиції.

Результат виконання програми наведено на рис. 19.11.

Рис. 19.11. Результат виконання програми з лістингу 19.3

19.2.2. Команди віднімання

SUBPS приймач, джерело	Паралельне віднімання 128-розрядних операндів

Команда **subps** виконує паралельне віднімання 128-розрядних операндів; як операнд-джерело буде один з XMM-регістрів або 128-розрядний елемент пам'яті. Вихідним операндом команди є один з XMM-регістрів:

subps xmm, xmm/mem128.

SUBSS приймач, джерело	Скалярне віднімання операндів

Команда **subss** виконує скалярне віднімання операндів. Молодші подвійні слова операндів повинні бути числами з плаваючою точкою в короткому форматі. Результат поміщається в операнд-приймач, яким може бути XMM-регістр. Вхідним операндом можуть бути XMM-регістр або 32-розрядний елемент пам'яті:

subss xmm, xmm/mem32.

Лістинг 19.4. Паралельне віднімання 128-розрядних дійсних чисел:

```
        .686p                          ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти
        .XMM          ; директива визначення команд XMM-розширення
        option casemap:none           ; отличие строчных и прописных букв
        include \masm32\include\kernel32.inc ; системные функции применений…
        includelib \masm32\lib\kernel32.lib
        .data                                  ; директива определения данных
mas1  DD 12.13, -15.01, -325.12, 77.1      ; резервування комірок для mas1
mas2  DD -25.25, 35.21, -34.56, 23.77      ; резервування комірок для mas2
        len EQU ($ - mas2)/4        ; кількість 32-розрядних чисел в mas2
        res  DD len DUP(0)   ; резервування 4-х 32-розрядних комірок для res
.code                                  ; директива начала кода программы
_st:                                   ; директива начала кода программы
movups XMM0, mas1  ; пересилання 4 × 32-розрядних чисел масиву mas1
     subps  XMM0, mas2                 ; паралельне віднімання чисел масивів
     movups res, XMM0                  ; запам'ятовування результату
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
        end _st                        ; директива окончания программы
```

19.2.3. Команди паралельного та скалярного множення

MULPS приймач, джерело	Паралельне множення 128-розрядних операндів

Команда **mulps** виконує паралельне множення 128-розрядних операндів; як операнд-джерело буде один з XMM-регістрів або 128-розрядний елемент пам'яті. Вихідним операндом команди повинен бути один з XMM-регістрів:

mulps xmm, xmm/mem128.

Лістинг 19.5. Паралельне множення 128-розрядних чисел:

```
.686p                    ; директива определения типа микропроцессора
.model flat              ; завдання лінійної моделі пам'яті
.XMM              ; директива визначення команд XMM-розширення
option casemap:none    ; отличие строчных и прописных букв
include \masm32\include\kernel32.inc ; системные функции применений…
includelib \masm32\lib\kernel32.lib
.data                                  ; директива определения данных
mas1  DD 12.13, -15.01, -325.12, 77.1
mas2  DD -25.25, 35.21, -34.56, 23.77
len EQU ($- mas2)/ type mas2  ; кількість 32-розрядних чисел в mas2
res  DD len DUP(0) ; резервування 4-х 32-розрядних комірок для res
.code                                  ; директива начала кода программы
_st:                                   ; директива начала кода программы
   lea    ESI, mas1                    ; занесення адреси масиву mas1 в esi
   lea    EDI, mas2                    ; занесення адреси масиву mas2 в edi
   movups XMM0, [ESI]
   mulps  XMM0, [EDI]
   movups res, XMM0
invoke ExitProcess, 0 ; повернення управління  ОС та вивільнення ресурсів
end _st                                ; директива окончания программы
```

MULSS приймач, джерело	Скалярне множення операндів

Команда **mulss** виконує скалярне множення операндів. Молодші подвійні слова операндів повинні бути числами з плаваючою точкою в короткому форматі. Результат поміщається в операнд-приймач, яким повинен бути XMM-регістр. Вхідним операндом можуть бути XMM-регістр або 32-розрядний елемент пам'яті:

mulss xmm, xmm/mem128.

Лістинг 19.6. Скалярне множення операндів:

```
.686
.model  flat, stdcall ;
.XMM          ;
option casemap:none          ;
include \masm32\include\windows.inc ;
include \masm32\macros\macros.asm
uselib kernel32, user32,masm32, fpu
BSIZE equ 14
.data  ; директива определения данных
a dd 10.25
b dd 12.5
c1 dd 5.0
res DD 4 dup(0)    ; резервування комірок пам'яті для масиву res
```

```
        titl db "Результати роботи",0
        info db "a*b+5, где a=10.25, b=12.5", 10, 13, "Ответ:"
        buffer db 256 dup(0), 0
.code
_start:
        movss XMM0,a
        mulss XMM0,b
        addss XMM0,c1
        movss res,XMM0
        fld dword ptr res
        invoke FpuFLtoA,0,2,ADDR buffer,SRC1_FPU or SRC2_DIMM
        invoke MessageBox,NULL,addr info,ADDR titl,MB_ICONINFORMATION
        invoke ExitProcess,0 ;
        end _start
```

19.2.4. Команди паралельного та скалярного ділення

DIVPS приймач, джерело	Паралельне ділення 128-розрядних операндів

Команда **divps** виконує паралельне ділення 128-розрядних операндів; як операнд-джерело буде один з XMM-регістрів або 128-розрядний елемент пам'яті. Вихідним операндом команди повинен бути один з XMM-регістрів, при цьому як ділене виступає операнд-приймач, а як дільник – операнд-джерело. Результат поміщається в операнд-приймач:

divps xmm, xmm/mem128.

DIVSS приймач, джерело	Скалярне ділення операндів

Команда **divss** виконує скалярне ділення операндів; молодші подвійні слова операндів повинні бути числами з плаваючою точкою в короткому форматі. Результат поміщається в операнд-приймач, яким повинен бути XMM-регістр. Вхідним операндом можуть бути XMM-регістр або 32-розрядний елемент пам'яті:

divss xmm, xmm/mem32.

Лістинг 19.7. Скалярне ділення масивів дійсних 128-розрядних чисел:

```
        .686                    ; директива определения типа микропроцессора
        .model flat, stdcall ; задание линейной модели памяти
        .XMM            ; директива визначення команд XMM-розширення
        option casemap:none  ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32,masm32
.data                                   ; директива определения данных
        mas1 DD  167.13,567.01,347.1,77.1  ; резервування комірок пам'яті
        mas2 DD  25.13,35.01,34.5,23.01 ; резервування комірок пам'яті
        len1 EQU ($-mas2)/ type mas2 ; визначення кількості подвійних слів
        titl1 db "Скалярная обработка массива чисел командами SSE",0
        info1 db "mas1    167.13, 567.01, 347.1,  77.1",10,13
              db "mas2     25.13,   35.01,   34.5,  23.01",10,13,13
              db "divss     "
        buf dd 30 dup(?),0        ; буфер
        frmt db "  %d  ",0
.code
```

```
_st:
        lea ESI, mas1 ; занесення адреси масиву mas1 в esi
        lea EDI, mas2 ; занесення адреси масиву mas2 в edi
        mov ECX, len1 ; занесення кількості подвійних слів у лічильник
         xor ebx,ebx
@@:
      movups XMM0, dword ptr [ESI] ; занесення 32-розрядних даних в XMM0
      divss XMM0, dword ptr [EDI] ; скалярне ділення молодших 32 чисел
      cvtss2si eax, XMM0 ;  float-число в ціле 32-розрядне число
        push edi
        push esi
        push ecx
        push ebx
invoke wsprintf,ADDR buf[ebx],ADDR frmt,eax
        pop ebx
        pop ecx
        pop esi
        pop edi
        add esi, 4 ;
        add edi, 4 ;
        add ebx, 4 ;
   loop @b ;
        invoke MessageBox,0,ADDR info1,ADDR titl1, MB_ICONINFORMATION
        invoke ExitProcess,0  ; повернення управління  ОС та визволення ресурсів
        end _st  ; завершение программы с именем_st
```

В програмі командою divss відбувається ділення молодшого (скалярного) числа, яке знаходиться в регістрі XMM0 на 32-розрядне число, яке знаходиться в пам'яті, адреса комірки якої записана в регістрі [EDI].

Командою cvtss2si відбувається перетворення дійсного числа в 32-розрядне ціле та запис результату в регістр eax.

Функція wsprintf для свого виконання використовує регістри загального призначення. Тому ті регістри, які будуть змінені цією функцією зберігаються в стеці командами push. Ця функція перетворює 32-розрядне ціле число в символи для виведення функцією MessageBox.

Результат виконання програми наведено на рис. 19.12.

Рис. 19.12. Результат виконання програми з лістингу 19.7

Лістинг 19.8. Виконання виразу a/b – c:

```
.686
.model flat, stdcall
.XMM
option casemap:none
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib kernel32, user32,masm32
.data
  data dd 1995.0, 5.0, 3.0  ;a = 1995.0, b = 5.0, c = 3.0
  res dd ?
  titl db "Автор:          ",0
  buff dd 0,0
```

```
    out1 db "       a/b - 3 = %d",0
.code
start:
    lea esi, data
    movss XMM0, [esi]
    add esi, 4
    divss XMM0, [esi]   ;a/b
    add esi, 4
    subss XMM0, [esi]     ; a/b - 3
    cvtss2si eax, XMM0   ; переводим результат в целое число
    mov res, eax              ; заносим в переменную res
    invoke wsprintf,addr buff, addr out1, res
    invoke MessageBox, NULL, addr out1, ADDR titl, MB_ICONINFORMATION
    invoke ExitProcess, 0
    ret
end start
```

В цій програмі результат виконання виразу в дійсному вигляді перетворюється в ціле число.

Цей результат можні вивести й дійсним числом, якщо його занести у вершину співпроцесора, а потім перетворити функцією FpuFLtoA:

```
movss res,XMM0
fld dword ptr res
invoke FpuFLtoA,0,5,ADDR buffer,SRC1_FPU or SRC2_DIMM
invoke MessageBox, NULL, addr out1, ADDR titl, MB_ICONINFORMATION
```

19.2.5. Додаткові арифметичні команди

SQRTPS приймач, джерело	Паралельне обчислення квадратного кореня

Команда **sqrtps** виконує паралельне обчислення квадратного кореня з упакованих чисел з плаваючою точкою. Команда має два операнди: джерело і приймач. Як операнд-джерело може бути XMM-регістр або 128-розрядний елемент пам'яті; як приймач – XMM-регістр:

sqrtps xmm, xmm/mem128.

SQRTSS приймач, джерело	Скалярне обчислення квадратного кореня

Команда **sqrtss** виконує скалярне обчислення квадратного кореня з упакованого числа з плаваючою точкою. Як операнд-джерело може бути 32-розрядний елемент пам'яті або XMM-регістр. У тому випадку, якщо джерелом є XMM-регістр, використовується молодший 32-розрядний операнд, при цьому решта операндів не змінюється. Як операнд-приймач повинен бути XMM-регістр:

sqrtss xmm, xmm/mem32.

Розглянемо приклад паралельного обчислення квадратного кореня з масиву упакованих чисел з плаваючою точкою.

Лістинг 19.9. Паралельне обчислення квадратного кореня з масиву упакованих чисел з плаваючою точкою:

```
        .686p                   ; директива определения типа микропроцессора
        .model flat                     ; завдання лінійної моделі пам'яті
        .XMM                    ; директива визначення команд XMM-розширення
        option casemap:none             ; отличие строчных и прописных букв
.data                                   ; директива определения данных
   mas1  DD 12.13, 15.01, 325.12, 77.1, 55.66, 77.88
   len EQU $- mas1 ; визначення кількості байтів масиву mas1
   res DD len DUP(0)            ; резервування пам'яті для масиву res
.code                                   ; директива начала кода программы
   _sqrt_ex proc
      mov   EAX, len  ; завантаження 24 байтів масиву mas1
      shr    EAX, 2                   ; len/4 = 6 – визначення кількості чисел
      mov   EBX, 4
      xor    EDX, EDX                ; ініціалізація EDX перед діленням
      div    EBX                     ; (eax, edx) визначення кількості циклів
      mov   ECX, EAX                 ; кількість циклів – у лічильник
      lea   ESI, mas1 ; завантаження адреси масиву mas1
      lea   EDI, res                 ; завантаження адреси масиву
next:
      sqrtps XMM0, [esi]        ; паралельне обчислення квадратних коренів
        movups [EDI], XMM0
        add    ESI, 16  ; підготовка до нового циклу
        add    EDI, 16
      loop    next                   ; перейти, якщо ecx ≠ 0
        cmp    EDX, 0 ; порівняння залишку
        jz     exit
        mov    ECX, EDX
      next1:
        sqrtss XMM0, [ESI]      ; обчислення кореня скалярною командою
        movss  [EDI], XMM0      ; скалярне пересилання
        add    ESI, 4
        add    EDI, 4
        dec    ECX
        jnz    next1
      exit:   lea    EAX, res        ; збереження адреси масиву res в eax
              ret                    ; возвращение из процедуры
        _sqrt_ex endp                ; закінчення процедури
      end                            ; закінчення програми
```

У прикладі початковий масив містить 6 подвійних слів (подання коротких чисел з плаваючою точкою). За допомогою команди sqrtps можна обробити 4 подвійних слова (16 байтів), а ті, що залишилися (2 подвійних слова), – командою sqrtss.

MAXPS приймач, джерело	Паралельне обчислення максимального значення для кожної пари упакованих чисел з плаваючою точкою

Команда **maxps** виконує паралельне обчислення максимального значення для кожної пари упакованих чисел з плаваючою точкою. Команда має два операнди: джерело і приймач. Як операнд-джерело може бути XMM-регістр або 128-розрядний елемент пам'яті; як приймач – XMM-регістр, в який поміщаються максимальні елементи кожної пари. Вміст операнда-джерела після виконання операції не змінюється:

maxps xmm, xmm/mem128.

MINPS приймач, джерело	Паралельне обчислення мінімального значення для кожної пари упакованих чисел

Команда **minps** виконує паралельне обчислення мінімального значення для кожної пари упакованих чисел з плаваючою точкою. Команда має два операнди: джерело і приймач. Як операнд-джерело може бути XMM-регістр або 128-розрядний елемент пам'яті; як приймач – XMM-регістр, в який поміщаються максимальні елементи кожної пари. Вміст операнда-джерела після виконання операції не змінюється:

minps xmm, xmm/mem128.

MAXSS приймач, джерело	Скалярне обчислення максимального значення для молодшої пари упакованих чисел 32-розрядних операндів

Команда **maxss** виконує скалярне обчислення максимального значення для молодшої пари упакованих 32-розрядних операндів приймача і джерела. Результат поміщається в операнд-приймач, яким може бути один з XMM-регістрів, при цьому його молодший операнд заміщається максимальним значенням. Як операнд-джерело може бути 32-розрядне значення в пам'яті або молодший операнд XMM-регістра. Після виконання операції вміст операнда-джерела залишається незмінним:

maxss xmm, xmm/mem32.

Розглянемо приклад скалярного обчислення максимальних значень упакованих чисел з плаваючою точкою двох масивів.

Лістинг 19.10. Програма скалярного обчислення максимальних значень упакованих чисел з плаваючою точкою двох масивів:

```
        .686p    ; директива определения типа микропроцессора
        .xmm  ; директива визначення команд SSE
        .model  flat, stdcall  ; задание линейной модели памяти
        option casemap:none    ; відмінність малих та великих  літер
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib user32, kernel32, fpu
.data
        mas1 DD 1.0, 7.0, 5.0, 2.0          ; перший масив
        mas2 DD 4.0, 1.0, 3.0, 8.0          ; другий масив
        _len equ ($-mas2)/type mas2         ; довжина масиву
        _res DD 4 dup(0)       ; масив для збереження результатів порівняння
        _title db "Определение максимальных значений",0 ; назва вікна
        _message db "mas1  1.0,  7.0,  5.0,  2.0 ;",10,13,\
                    "mas2  4.0,  1.0,  3.0,  8.0 ;",10,13, "Res:   "
        buf db 50 dup(0),0 ; буфер для запису результату для виведення на екран
        count dd 0 ; змінна для обчислення зсуву для запису даних у буфер
.code
_st:
        movups XMM0, mas1      ; 1й массив у регістр XMM0
        movups XMM1, mas2      ; 2й массив у регістр XMM1
        maxss XMM0,XMM1      ; максимальне значення молодшої пари чисел
movhlps XMM2, XMM0 ; 2 старших  числа з XMM0 у молодшу частину XMM2
```

```
movhlps XMM3, XMM1 ; 2 старших числа з XMM1 у молодшу частину XMM3
maxss XMM3,XMM2     ; максимальне значення  молодшої пари чисел
unpcklpd XMM0,XMM3 ;
movups _res, XMM0     ; запис результату у вихідний масив
; запис чисел з вихідного масиву до буферу для виведення через MessageBox
        finit
        lea esi,_res ; записуємо адресу масиву з результатами обчислень
        lea edi,_buf ; записуємо адресу буферу
        mov ecx,_len ; записуємо кількість елементів масиву
m1:
  fld dword ptr [esi] ; в st(0) для переведення у 80-розрядний формат
        invoke FpuFLtoA,0,1,ADDR [edi],SRC1_FPU or SRC2_DIMM ;
        add esi,4 ; перехід до наступного числа
        add edi,4
    loop m1    ; повторюємо, доки не закінчиться масив з результатом
        invoke MessageBox, 0, addr _message,addr _title, MB_OK ;
        invoke ExitProcess, 0 ;
        end _st
```

В програмі виводяться числа без збільшення кількості знаків після коми. У випадку, якщо потрібно збільшити кількість таких цифр, то можна використати наступний фрагмент коду:

```
    …
invoke FpuFLtoA,0,4,ADDR [edi],SRC1_FPU or SRC2_DIMM ;
     add esi,4 ; перехід до наступного числа
; пошук нетермінального 0, який FpuFLtoA записує в буфер та при знаходженні
; якого виведення зупиняється
        xor edx,edx ;
        mov edx,edi ;
        mov count,0 ; занулюємо лічильник
m2:
.IF [edx] != byte ptr 0 ; якщо символ не дорівнює 0, то до наступного символа
        inc edx
        add count,1 ;
        jmp m2 ; продовжуємо пошук 0
.ENDIF
        add edi, count ;
        mov word ptr [edi],20h ; запис символу пробілу
        add edi,4
        loop m1 ; повторюємо, доки не закінчиться масив з результатом.
```

В даному фрагменті функцією FpuFLtoA виводяться чотири знаки. Досягається це встановленням додаткового нуля після кожного числа.

Результат виконання програми наведено на рис. 19.13.

Рис. 19.13. Результат виконання програми

MINSS приймач, джерело	Скалярне обчислення мінімального значення для молодшої пари упакованих чисел 32-розрядних операндів

Команда **minss** виконує скалярне обчислення мінімального значення для молодшої пари упакованих 32-розрядних операндів приймача і джерела. Результат поміщається в операнд-приймач, яким може бути один з XMM-регістрів, при цьому його молодший операнд заміщається мінімальним значенням. Як операнд-джерело може бути 32-розрядне значення в пам'яті або молодший операнд XMM-регістра. Після виконання операції вміст операнда-джерела залишається незмінним:

```
minss  xmm, xmm/mem32.
```

Арифметичні команди обчислення обернених значень

Арифметичні команди обчислення обернених значень, як і більшість SSE-команд, оперують або чотирма упакованими 32-розрядними значеннями (паралельні), або молодшими 32-розрядними операндами (скалярні). Команди обчислення обернених значень забезпечують точність не менше 11 бітів. Це означає, що максимальне значення відносної похибки буде менш ніж $1{,}5 \times 2^{-12}$. Для кращого розуміння можна виразити значення відносної похибки формулою

$$\frac{\text{Точне значення} - \text{приблизне значення}}{\text{приблизне значення}} \le 1{,}5 \times 2^{-12}.$$

RCPPS приймач, джерело	Паралельне обчислення обернених значень упакованих операндів

Команда **rcpps** виконує паралельне обчислення обернених значень упакованих операндів. Якщо X – одне з значень з 32-розрядних операндів, то воно обчислюється як $1/x$. Вхідним операндом може бути XMM-регістр або 128-розрядний елемент пам'яті, а вихідним операндом повинен бути XMM-регістр:

```
rcpps  xmm, xmm/mem128.
```

RCPSS приймач, джерело	Скалярне обчислення оберненого значення молодшого 32-розрядного операнда

Команда **rcpss** виконує скалярне обчислення оберненого значення молодшого 32-розрядного операнда. Як вхідний операнд може бути XMM-регістр або 32-розрядний елемент пам'яті, а вихідним операндом повинен бути XMM-регістр. Операція не зачіпає вмісту старших операндів XMM-регістра:

```
rcpss  xmm, xmm/mem32.
```

Лістинг 19.11. Програма обчислення зворотних чисел:

```
.686p                ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти
.XMM                 ; директива визначення команд MMX
option casemap:none ; отличие строчных и прописных букв
include \masm32\include\windows.inc ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib kernel32, user32,fpu
.data
arr1 dd 5.0,2.0,3.0,1.0
arr2 dd 4 dup(0)
```

```
        ttl db "Результат",0
        buf dd ?
.CODE
_start:
        lea edi,arr1
        lea esi,arr2
        mov ecx,4
@@:
        mov edx,esi
movss XMM0,[edi] ; завантажити 1 число одинарної точності (тип dd) в регістр
rcpss XMM1,XMM0  ; обчислити зворотне число 1/xmm0
movss [esi],XMM1 ; вивантажити 1 число одинарної точності (тип dd) в пам'ять
        add esi,4       ; збільшити адреси
        add edi,type arr1
        dec ecx
    jnz @b
        lea esi,arr2
        lea edi,buf
        mov ecx,4
@@:
    finit
    fld dword ptr [esi] ; для перетворення типу (dd->dt) завантажити в st0
invoke FpuFLtoA,0,5,edi,SRC1_FPU or SRC2_DIMM ; перетворити st0 в рядок
    add esi,type arr2 ; збільшити адреси
    add edi,8 ; додати до edi кількість символів, виведених в буфер
    dec ecx
    jnz @b
        invoke MessageBox,0,addr buf,addr ttl,MB_OK
        end   _start
```

Результат виконання програми наведено на рис. 19.14.

Рис. 19.14. Результат виконання програми

Для виведення результату в програмі використовано функцію FpuFLtoA, яка, як відомо, входить да функцій бібліотеки masm32 по обробці команд fpu. Тому і результати не зовсім співпадають.

RSQRTPS приймач, джерело	Паралельне обчислення обернених значень квадратного кореня упакованих операндів

Команда **rsqrtps** виконує паралельне обчислення обернених значень квадратного кореня упакованих операндів. Якщо X – значення з 32-розрядних операндів, то воно обчислюється як $1/\sqrt{X}$. Як вхідний операнд може використовуватися XMM-регістр або 128-розрядний елемент пам’яті, а вихідним операндом повинен бути XMM-регістр:

```
rsqrtps  xmm, xmm/mem128.
```

RSQRTSS приймач, джерело	Скалярне обчислення оберненого значення квадратного кореня молодшого операнда

Команда **rsqrtss** виконує скалярне обчислення оберненого значення квадратного кореня молодшого 32-розрядного операнда. Як вхідний операнд може бути XMM-регістр або 32-розрядний елемент пам'яті, а вихідним операндом повинен бути XMM-регістр. Операція не зачіпає вмісту старших операндів XMM-регістра:

rsqrtss xmm, xmm/mem32.

19.3. Команди порівняння

Команди порівняння дозволяють визначати відповідність операндів до вказаних умов і, залежно від результату порівняння, встановлюють в потрібному елементі операнда-приймача **двійкові нулі (якщо умова не виконується)** або **двійкові одиниці (якщо умова виконується)**. Команди порівняння можуть виконуватися паралельно над упакованими операндами, скалярні команди – над молодшими подвійними словами. Усі команди мають два операнди. Як вхідний операнд, або операнд-джерело, може бути XMM-регістр або елемент пам'яті (128-розрядний операнд для паралельних команд і 32-розрядна – для скалярних). Вихідним операндом, або операндом-приймачем, може бути тільки один з XMM-регістрів, в якому задіяні або 128 бітів (паралельні команди), або молодші 32 біти(скалярні команди). ***Результатом порівняння*** є або всі нулі, або всі одиниці.

Група паралельних команд порівняння

CMPPS приймач, джерело, предикат	Паралельна перевірка упакованих операндів одинарної точності

Команда **cmpps** порівнює упаковані операнди з одинарною точністю, використовуючи imm8 як предикат порівняння.

Третій операнд команди (безпосереднє значення imm8) задає умова операції порівняння (0 – дорівнює, 1 – менше, 2 – менше або дорівнює, 3 – незрівняні, 4 – не дорівнює, 5 – не менше, 6 – не менше і не дорівнює, 7 – порівнянні):

```
CMPEQPS xmm1, xmm2  – CMPPS xmm1, xmm2, 0
CMPLTPS xmm1, xmm2  –  CMPPS xmm1, xmm2, 1
CMPLEPS xmm1, xmm2  –  CMPPS xmm1, xmm2, 2
CMPUNORDPS xmm1, xmm2  –  CMPPS xmm1, xmm2, 3
CMPNEQPS xmm1, xmm2  –  CMPPS xmm1, xmm2, 4
CMPNLTPS xmm1, xmm2  –  CMPPS xmm1, xmm2, 5
CMPNLEPS xmm1, xmm2  –  CMPPS xmm1, xmm2, 6
CMPORDPS xmm1, xmm2  –  CMPPS xmm1, xmm2, 7
```

Умова порівняння вважається виконаною (True), якщо виконується: DEST умова SRC. Наприклад, для вхідного значення imm8 = 1, умова DEST <

SRC виконується (True), якщо DEST = 2, SRC = 8, і не виконується, якщо DEST = 1, SRC = -5.

Якщо задана умова порівняння для відповідної пари коротких дійсних операндів виконується, то в відповідне 32-розрядне поле операнда-призначення поміщаються всі 1 (True), інакше – все 0 (False).

Розглянемо приклад. Задано два масива. Якщо перший масив менше другого, то другий масив ділиться на перший. Якщо перший масив більше другого, то два масиви перемножуються.

Лістинг 19.12:

```
        .686p                   ; директива определения типа микропроцессора
        .model flat,stdcall  ; задание линейной модели памяти
        .XMM
        option casemap:none                 ; отличие строчных и прописных букв
        include \masm32\include\windows.inc    ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32
.data
        mas1 Dq 3.,2.
        mas2 Dq 9.,4.
        len1 EQU ($-mas2)/type mas2
        sum2 DD len1 DUP(?),0
        buf dd len1 DUP(0),0
ifmt db "Массив 1: 3.0, 2.0",0Ah,0Dh,
"Массив 2: 9.0, 4.0",0Ah,0Dh,0Dh,
"Если первый массив меньше второго,то второй делится на первый.",0Ah,0Dh,0Dh,
"Если первый массив больше второго,то два массива перемножаются.",0Ah,0Dh,0Dh,
" Ответ = ", " %d; %d",0
        _title db "Задача с SSE2", 0
.code
_st:    movupd XMM0, mas1
        movupd XMM1, mas2
   cmpps XMM0,XMM1,1  ; порівняння на менше
        movups sum2, XMM0
        lea ebx, sum2
.IF(sum2[4] == -1) ; одиниці якщо умова виконується: mas1<mas2
        movupd XMM1, mas1
        movupd XMM0, mas2
        divpd XMM0, XMM1
        unpcklpd XMM4, XMM0
        unpckhpd XMM4, XMM6
        unpckhpd XMM5, XMM0
        unpckhpd XMM5, XMM4
        cvtpd2pi MM0,XMM4
        cvtpd2pi MM1,XMM5
        movd dword ptr sum2,MM0
        movd dword ptr sum2[4],MM1
.ELSEIF
        movupd XMM1, mas1
        movupd XMM0, mas2
        mulpd XMM0, XMM1
        unpcklpd XMM4, XMM0
        unpckhpd XMM4, XMM6
        unpckhpd XMM5, XMM0
        unpckhpd XMM5, XMM4
        cvtpd2pi MM0,XMM4
        cvtpd2pi MM1,XMM5
```

```
        movd dword ptr sum2,MM0
        movd dword ptr sum2[4],MM1
.ENDIF
        invoke wsprintf,ADDR buf, ADDR ifmt, sum2,sum2[4];
        invoke MessageBox, 0, addr buf, addr _title, MB_ICONINFORMATION
        invoke ExitProcess, 0
        end _st
```

Результат виконання програми наведено на рис. 19.15.

Рис. 19.15. Результат виконання програми

CMPEQPS приймач, джерело	Паралельна перевірка умови рівності 32-розрядних упакованих операндів

Команда **cmpeqps** перевіряє умову рівності 32-розрядних упакованих операндів і встановлює відповідні значення в операнді-приймачі. Вміст операнда-джерела після виконання операції не змінюється:

cmpeqps xmm, xmm.

CMPLTPS приймач, джерело	Паралельна перевірка умови «менше» для 32-розрядних упакованих операндів

Команда **cmpltps** перевіряє умову «менше» (less-than) для 32-розрядних упакованих операндів і встановлює відповідні значення в операнді-приймачі. Вміст операнда-джерела після виконання операції не змінюється:

cmpltps xmm, xmm.

CMPLEPS приймач, джерело	Паралельна перевірка «менше або дорівнює» для 32-розрядних упакованих операндів

Команда **cmpleps** перевіряє умову «менше або дорівнює» (less-than-or-equal) для 32-розрядних упакованих операндів і встановлює відповідні значення в операнді-приймачі. Вміст операнда-джерела після виконання операції не змінюється:

cmpleps xmm, xmm.

CMPUNORDPS приймач, джерело	Паралельна перевірка умови «невпорядкованості» для 32-розрядних упакованих операндів

Команда **cmpunordps** перевіряє умову «невпорядкованості» (unordered) для 32-розрядних упакованих операндів і встановлює відповідні значення в операнді-приймачі. Вміст операнда-джерела після виконання операції не змінюється:

cmpunordps xmm, xmm.

CMPNEQPS приймач, джерело	Паралельна перевірка умови «недорівненості» для 32-розрядних упакованих операндів

Команда **cmpneqps** перевіряє умову «не дорівнює» (not-equal-to) для 32-розрядних упакованих операндів і встановлює відповідні значення в операнді-приймачі. Вміст операнда-джерела після виконання операції не змінюється:

cmpneqps xmm, xmm.

CMPNLTPS приймач, джерело	Паралельна перевірка умови «не менше» для 32-розрядних упакованих операндів

Команда **cmpnltps** перевіряє умову «не менше» (not-less-than) для 32-розрядних упакованих операндів і встановлює відповідні значення в операнді-приймачі. Вміст операнда-джерела після виконання операції не змінюється:

cmpnltps xmm, xmm.

CMPNLEPS приймач, джерело	Паралельна перевірка умови «не менше або дорівнює» для 32-розрядних упакованих операндів

Команда **cmpnleps** перевіряє умову «не менше або дорівнює» (not-less-than-or-equal-to) для 32-розрядних упакованих операндів і встановлює відповідні значення в операнді-приймачі. Вміст операнда-джерела після виконання операції не змінюється:

cmpnleps xmm, xmm.

CMPORDPS приймач, джерело	Паралельна перевірка умови «впорядкованості» для 32-розрядних упакованих операндів

Команда **cmpordps** перевіряє умову «впорядкованості» (ordered) для 32-розрядних упакованих операндів і встановлює відповідні значення в операнді-приймачі. Вміст операнда-джерела після виконання операції не змінюється:

cmpordps xmm, xmm.

Група скалярних команд порівняння

CMPEQSS приймач, джерело	Перевірка умови рівності молодших 32-розрядних операндів і встановлення відповідного значення в молодшому подвійному слові операнда-приймача

Команда **cmpeqss** перевіряє умову рівності молодших 32-розрядних операндів і встановлює відповідне значення в молодшому подвійному слові операнда-приймача. Вміст операнда-джерела після виконання операції не змінюється:

cmpeqss xmm, xmm.

CMPLTSS приймач, джерело	Перевірка умови «менше» для молодших 32-розрядних упакованих операндів і встановлення відповідного значення в молодшому подвійному слові операнда-приймача

Команда **cmpltss** перевіряє умову «менше» (less-than) для молодших 32-розрядних упакованих операндів і встановлює відповідне значення в молодшому подвійному слові операнда-приймача. Вміст операнда-джерела після виконання операції не змінюється:

cmpltss xmm, xmm.

CMPLESS приймач, джерело	Перевірка умови «менше або дорівнює» для молодших 32-розрядних упакованих операндів і встановлення відповідного значення в молодшому подвійному слові операнда-приймача

Команда **cmpless** перевіряє умову «менше або дорівнює» (less-than-or-equal) для молодших 32-розрядних упакованих операндів і встановлює відповідне значення в молодшому подвійному слові операнда-приймача. Вміст операнда-джерела після виконання операції не змінюється:

cmpless xmm, xmm.

CMPUNORDSS приймач, джерело	Перевірка умови «невпорядкованості» для молодших 32-розрядних упакованих операндів і встановлення відповідного значення в молодшому подвійному слові операнда-приймача

Команда **cmpunordss** перевіряє умову «невпорядкованості» (unordered) для молодших 32-розрядних упакованих операндів і встановлює відповідне значення в молодшому подвійному слові операнда-приймача. Вміст операнда-джерела після виконання операції не змінюється:

cmpunordss xmm, xmm.

CMPNEQSS приймач, джерело	Перевірка умови «не дорівнює» для молодших 32-розрядних упакованих операндів і встановлення відповідного значення в молодшому подвійному слові операнда-приймача

Команда **cmpneqss** перевіряє умову «не дорівнює» (not-equal-to) для молодших 32-розрядних упакованих операндів і встановлює відповідне значення в молодшому подвійному слові операнда-приймача. Вміст операнда-джерела після виконання операції не змінюється:

cmpneqss xmm, xmm.

CMPNLTSS приймач, джерело	Перевірка умови «не менше» для молодших 32-розрядних упакованих операндів і встановлення відповідного значення в молодшому подвійному слові операнда-приймача

Команда **cmpnltss** перевіряє умову «не менше» (not-less-than) для молодших 32-розрядних упакованих операндів і встановлює відповідне значення в молодшому подвійному слові операнда-приймача. Вміст операнда-джерела після виконання операції не змінюється:

```
cmpnltss   xmm, xmm.
```

CMPNLESS приймач, джерело	Перевірка умови «не менше або дорівнює» для молодших 32-розрядних упакованих операндів і встановлення відповідного значення в молодшому подвійному слові операнда-приймача

Команда **cmpnless** перевіряє умову «не менше або дорівнює» (not-less-than-or-equal-to) для молодших 32-розрядних упакованих операндів і встановлює відповідне значення в молодшому подвійному слові операнда-приймача. Вміст операнда-джерела після виконання операції не змінюється:

```
cmpnless   xmm, xmm.
```

CMPORDSS приймач, джерело	Перевірка умови «впорядкованості» для молодших 32-розрядних упакованих операндів і встановлення відповідного значення в молодшому подвійному слові операнда-приймача

Команда **cmpordss** перевіряє умову «впорядкованості» (ordered) для молодших 32-розрядних упакованих операндів і встановлює відповідне значення в молодшому подвійному слові операнда-приймача. Вміст операнда-джерела після виконання операції не змінюється:

```
cmpordss   xmm, xmm.
```

COMISS/UCOMISS приймач, джерело	Скалярне порівняння молодших 32-розрядних операндів. Вміст обох операндів після виконання операції порівняння залишається незмінним, але в регістрі прапорців EFLAGS процесора певним чином встановлюються деякі прапорці

Команди **comiss** і **ucomiss** мають два операнди і виконують скалярне порівняння молодших 32-розрядних операндів. Особливістю цих команд є те, що вміст обох операндів після виконання операції порівняння залишається незмінним, але в регістрі прапорців EFLAGS процесора певним чином встановлюються прапорці ZF, PF і CF, а прапорці OF, SF і AF скидаються в 0. Вхідними операндами обох команд можуть бути XMM-регістри або 32-розрядні змінні в пам'яті, вихідними операндами можуть бути тільки XMM-регістри:

```
comiss  xmm, xmm/mem32;
ucomiss xmm, xmm/mem32.
```

У табл. 19.1 наводиться співвідношення між порівнюваними операндами і встановлюваними прапорцями.

Таблиця 19.1

Результат порівняння op1 та op2	ZF	PF	CF
Операнди неврегульовані	1	1	1
op1 < op2	0	0	1
op1 > op2	0	0	0
op1 = op2	1	0	0

Команди ucomiss та comiss відрізняються тим, що вони генерують виняткові ситуації для різних форматів не-чисел (NAN). Ці команди дуже зручні при організації галужень в програмах, оскільки за станом прапорців можна порізному інтерпретувати результат порівняння.

19.4. Команди перетворення

Команди перетворення можуть виконуватися, як і більшість решти SSE-команд, в паралельному і скалярному контекстах. Наведемо цю групу команд.

CVTPS2PI приймач, джерело	Паралельне перетворення двох молодших упакованих 32-розрядних чисел з плаваючою точкою в короткому форматі в два 32-розрядних цілих числа

Команда **cvtps2pi** виконує паралельне перетворення двох молодших упакованих 32-розрядних чисел з плаваючою точкою в короткому форматі в два 32-розрядних цілих числа. Команда має два операнди. Як вхідний операнд може бути XMM-регістр або 128-розрядний елемент пам'яті. Вихідним операндом може служити один з регістрів MMX. На результат перетворення впливає установлення бітів поля rc, що визначає режим округлення, регістра управління /стану (MXCSR):

cvtps2pi mmx, xmm/mem64.

CVTSS2SI приймач, джерело	Скалярне перетворення молодшого 32-розрядного числа з плаваючою точкою в короткому форматі в 32-розрядне ціле число

Команда **cvtss2si** виконує скалярне перетворення молодшого 32-розрядного числа з плаваючою точкою в короткому форматі в 32-розрядне ціле число. Команда має два операнди. Як вхідний операнд може бути XMM-регістр або 128-розрядний елемент пам'яті. Вихідним операндом може служити один з 32-розрядних регістрів загального призначення. Як і для команди cvtps2pi, на результат перетворення впливає установлення бітів поля rc, що визначає режим округлення, регістра управління /стану (MXCSR):

cvtss2si r32/r64, xmm/mem32.

Наприклад:

```
mas1  DD 1.,2.0,3.,4. ; резервування комірок пам'яті для масиву mas1
mas2  DD 2.,3.,4.,5.0 ; резервування комірок пам'яті для масиву mas2
…
movups XMM0, mas1          ; пересилання невирівняними даними
```

```
mulss  XMM0, mas2   ; скалярне перемноження: 1. x 2. =
cvtss2si eax,XMM0 ; скалярне перетворення одного мол. числа
```

Результат фрагмента програми та вміст регістрів наведено на рис. 19.16.

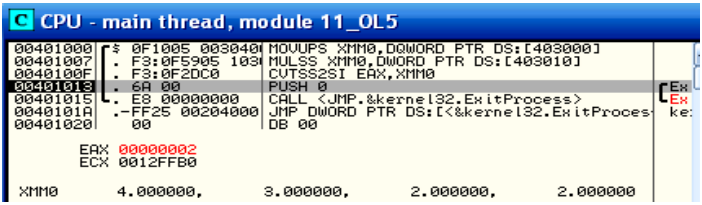


Рис. 19.16. Результат фрагмента програми та вміст регістрів

За командою movups XMM0, mas1 до регістра XMM0 заносяться всі чотири числа масиву mas1. За командою скалярного перемноження відбувається множення одного молодшого числа (1.) масиву mas1 на одне молодше число (2.) масиву mas2. Для того, щоб перетворити дійсне число та занести його до регістра eax використана команда cvtss2si. В результаті в регістрі XMM0 зберігатимуться числа, які наведені на рис. 19.16.

CVTTPS2PI приймач, джерело	Паралельне перетворення двох молодших упакованих 32-розрядних чисел з плаваючою точкою в короткому форматі в два 32-розрядних цілих числа шляхом відсікання дробової частини початкових операндів

Команда **cvttps2pi** виконує паралельне перетворення двох молодших упакованих 32-розрядних чисел з плаваючою точкою в короткому форматі в два 32-розрядних цілих числа шляхом відсікання дробової частини початкових операндів. Команда має два операнди. Як вхідний операнд може бути XMM-регістр або 128-розрядний елемент пам'яті. Вихідним операндом може служити один з MMX-регістрів. Встановлення бітів поля rc, що визначає режим округлення, регістра управління /стану (MXCSR) на результат не впливає:

cvttps2pi mmx, xmm/mem64.

CVTTSS2SI приймач, джерело	Скалярне перетворення молодшого 32-розрядного числа з плаваючою точкою в короткому форматі в 32-розрядне ціле число шляхом відсікання дробової частини вхідного операнда

Команда **cvttss2si** виконує скалярне перетворення молодшого 32-розрядного числа з плаваючою точкою в короткому форматі в 32-розрядне ціле число шляхом відсікання дробової частини вхідного операнда. Команда має два операнди. Як вхідний операнд можуть бути XMM-регістр або 128-розрядний елемент пам'яті, а вихідним операндом може служити один з 32-розрядних регістрів загального призначення. Установлення бітів поля rc, що визначає

режим округлення, регістра управління /стану (MXCSR) на результат не впливає:

cvttss2si r32/r64, xmm/mem32.

CVTPI2PS приймач, джерело	Перетворення двох цілих 32-розрядних чисел із знаком в два 32-розрядних числа з плаваючою точкою в короткому форматі

Команда **cvtpi2ps** виконує перетворення двох цілих 32-розрядних чисел із знаком в два 32-розрядних числа з плаваючою точкою в короткому форматі. Команда має два операнди. Як вхідний операнд, або операнд-джерело, може бути MMX-регістр або 64-розрядний операнд в пам'яті. Вихідним операндом може служити тільки XMM-регістр. Результат перетворення поміщається в молодших два 32-розрядних елементи XMM-регістра, старші два елементи залишаються без змін. На результат перетворення впливає установлення бітів поля rc, що визначає режим округлення, регістра управління /стану (MXCSR):

cvtpi2ps xmm, mmx/mem64.

CVTSI2SS приймач, джерело	Перетворення 32-розрядного числа в цілочисельному форматі в 32-розрядне число з плаваючою точкою в короткому форматі

Команда **cvtsi2ss** виконує перетворення 32-розрядного числа в цілочисловому форматі в 32-розрядне число з плаваючою точкою в короткому форматі. Команда має два операнди. Як вхідний операнд, або операнд-джерело, може бути 32-розрядний регістр загального призначення або операнд в пам'яті. Вихідним операндом може служити тільки XMM-регістр. Результат перетворення поміщається в молодший 32-розрядний елемент XMM-регістра, при цьому три старших елементи залишаються без змін. На результат перетворення впливає встановлення бітів поля rc, що визначає режим округлення, регістра управління /стану (MXCSR):

cvtsi2ss xmm, r/mem32;
cvtsi2ss xmm, r/mem64.

19.5. Логічні команди

На відміну від більшості команд SSE-расширения, всі логічні команди є паралельними і дозволяють виконувати операції логічного *І, АБО, І-НІ*, що виключає *АБО* над окремими парами бітів операндів.

ANDPS приймач, джерело	Паралельна операція логічного *І* над парами бітів упакованих чисел

Команда **andps** виконує паралельну операцію логічного *І* над парами бітів упакованих чисел з плаваючою точкою операнда-джерела і операнда-приймача. Команда має два операнди. Як вхідний операнд (джерело) може

бути XMM-регістр або 128-розрядний елемент пам'яті, а як вихідний (приймач) – XMM-регістр. Після виконання команди вміст операнда-джерела не змінюється, а результат поміщається в операнд-приймач:

andps xmm, xmm/mem128.

Використання команди andps наведено у фрагменті програми:

```
mas1  DD 12.13, -15.01, -325.12, 77.1       ; резервування комірок для mas1
mas2  DD -25.25, 35.21, -34.56, 23.77       ; резервування комірок для mas2
      len EQU ($- mas2)/4
      res  DD len DUP(0)
.code
_st:  movups XMM0, mas1                     ; пересилання невирівняними даними
      andps  XMM0, mas2
      movups res, XMM0
```

Результат виконання логічної команди andps наведено на рис. 19.17.

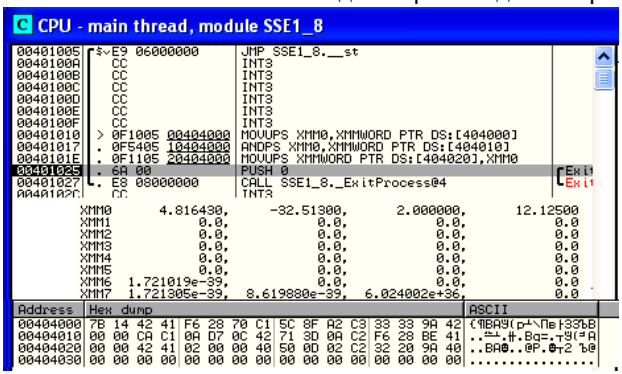


Рис. 19.17. Результат виконання логічної команди andps

ANDNPS приймач, джерело	Паралельна операція логічного *I-НІ* над парами бітів упакованих чисел з плаваючою точкою

Команда **andnps** виконує паралельну операцію логічного *I-НІ* над парами бітів упакованих чисел з плаваючою точкою операнда-джерела і операнда-приймача. Команда має два операнди. Як вхідний операнд (джерело) може бути XMM-регістр або 128-розрядний елемент пам'яті, а як вихідний (приймач) – XMM-регістр. Після виконання команди вміст операнда-джерела не змінюється, а результат поміщається в операнд-приймач:

andnps xmm, xmm/mem128.

ORPS приймач, джерело	Паралельна операція логічного *АБО* над парами бітів упакованих чисел з плаваючою точкою

Команда **orps** виконує паралельну операцію логічного *АБО* над парами бітів упакованих чисел з плаваючою точкою операнда-джерела і операнда-приймача. Команда має два операнди. Як вхідний операнд (джерело) може

бути XMM-регістр або 128-розрядний елемент пам'яті, а як вихідний (приймач) – XMM-регістр. Після виконання команди вміст операнда-джерела не змінюється, а результат поміщається в операнд-приймач:

orps xmm, xmm/mem128.

XORPS приймач, джерело	Паралельна логічна операція "виключне *АБО"* над парами бітів упакованих чисел з плаваючою точкою

Команда **xorps** виконує паралельну логічну операцію "виключне *АБО"* над парами бітів упакованих чисел з плаваючою точкою операнда-джерела і операнда-приймача. Команда має два операнди: як вхідний операнд (джерело) може бути XMM-регістр або 128-розрядний елемент пам'яті, а як вихідний (приймач) – XMM-регістр. Після виконання команди вміст операнда-джерела не змінюється, а результат поміщається в операнд-приймач:

xorps xmm, xmm/mem128.

19.6. Команди управління станом

До групи команд управління станом належать команди, що виконують завантаження/збереження регістрів стану і управління .

LDMXCSR приймач, джерело	Завантаження регістра управління /стану вмістом 32-розрядного елемента пам'яті

Команда **ldmxcsr** виконує завантаження регістра управління управління /стану (MXCSR) вмістом 32-розрядного елемента пам'яті, яка і є єдиним операндом:

ldmxcsr mem32.

STMXCSR приймач, джерело	Збереження вмісту регістра управління /стану в 32-розрядному осередку пам'яті

Команда **stmxcsr** виконує збереження вмісту регістра управління /стану (MXCSR) в 32-розрядному осередку пам'яті, яка і є єдиним операндом:

stmxcsr mem32.

FXRSTOR приймач, джерело	Завантаження заздалегідь збереженого стану співпроцесора, MMX- і SEE-розширення з області пам'яті розміром 512 байт

Команда **fxrstor** виконує завантаження заздалегідь збереженого стану співпроцесора, MMX- і SEE-розширення з області пам'яті розміром 512 байт. Як операнд виступає адреса області пам'яті, який повинен бути вирівняний по 16-байтовій межі:

fxrstor mem512byte.

FXSAVE приймач, джерело	Збереження стану співпроцесора, MMX- і SSE-розширення в область пам'яті розміром в 512 байт

Команда **fxsave** виконує збереження стану співпроцесора, MMX- і SSE-расширения в область пам'яті розміром в 512 байт. Як операнд виступає адреса області пам'яті:

fxsave mem512byte.

19.7. Команди розпаковування даних

UNPCKHPS приймач, джерело	Паралельне переміщення старших подвійних слів

Команда **unpckhps** виконує паралельне переміщення старших подвійних слів з операнда-джерела і операнда-приймача в операнд-приймач. При цьому два старших подвійних слова операнда-джерела стають старшими подвійними словами в 64-розрядних елементах операнда-приймача, а два старших подвійних слова операнда-приймача – молодшими подвійними словами в 64-розрядних елементах операнда-приймача. Вхідним операндом (джерелом) може бути XMM-регістр або 128-розрядний елемент пам'яті; як вихідний операнд повинен бути XMM-регістр:

unpckhps xmm, xmm/mem128.

Схему виконання команди наведено на рис. 19.18.

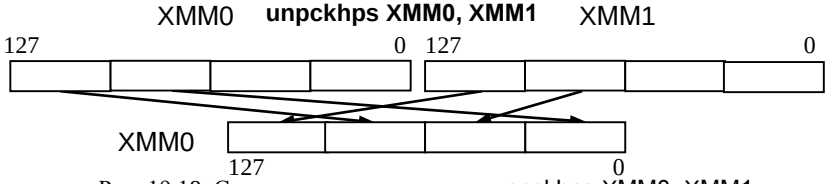


Рис. 19.18. Схема виконання команди unpckhps XMM0, XMM1

UNPCKLPS приймач, джерело	Паралельне переміщення молодших подвійних слів

Команда **unpcklps** виконує паралельне переміщення молодших подвійних слів з операнда-джерела і операнда-приймача в операнд-приймач.

При цьому два молодших подвійних слова операнда-джерела стають старшими двійковими словами в 64-розрядних елементах операнда-приймача, а два молодших подвійних слова операнда-приймача – молодшими подвійними словами в 64-розрядних елементах операнда-приймача. Вхідним операндом (джерелом) може бути XMM-регістр або 128-розрядний елемент пам'яті; як вихідний операнд повинен бути XMM-регістр:

unpcklps xmm, xmm/mem128.

Схему роботи команди наведено на рис. 19.19.

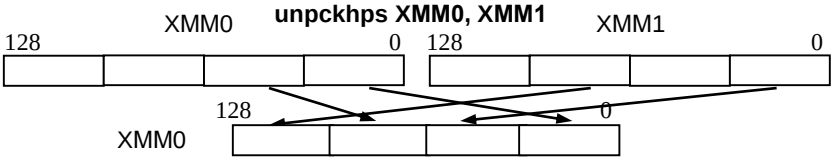


Рис. 19.19. Схема команди unpckhps XMM0, XMM1

SHUFPS приймач, джерело, маска	Переставлення 32-розрядних упакованих операндів відповідно до заданої маски

Команда **shufps** виконує переставлення 32-розрядних упакованих операндів відповідно до заданої маски. Команда має три операнди: вхідний, вихідний і операнд-маску. Маска є безпосереднім 8-розрядним значенням, яка задає порядок переставлення операндів. Кожна пара бітів маски визначає **номер** упакованого 32-розрядного операнда в приймачі або джерелі, яке повинне поміщатися в операнд-приймач:

shufps xmm, xmm/mem128, imm8.

Переставлення виконуються одночасно, тобто паралельно. Схему роботи команди ілюструє рис. 19.20.

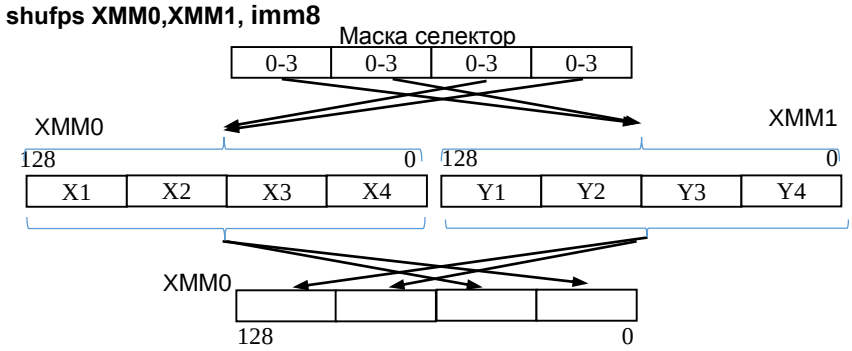


Рис. 19.20. Схема команди shufps XMM0,XMM1, 7Ah

Бітова маска селектор фактично розбита на 4 частини по два біта в кожній: 00, 01, 10, 11. Відповідно, кожна з позицій маски може приймати значення від 0 до 3. Біти 0,1 параметра imm8 задають номер копійованого слова для молодшого поля операнда-призначення, біти 2, 3 – для наступного, 4, 5 – для третього і 6, 7 – для самого старшого.

Приклади:

shufps xmm0,xmm0,10101010b – за маскою 10b команда помістить вміст XMM02 (одна з 32-бітових частин регістра XMM0) по всіх XMM0X частинам.

shufps xmm0,xmm0,00011011 – команда виконає перестановку всіх чотирьох 32 бітових частин регістра XMM0 у зворотному порядку (переміщення 3-го (11) числа на місце розташування нульового числа; переміщення 2-го (10) числа на місце першого числа, переміщення 1-го (01) числа на місце другого числа, переміщення 0-го (00) числа на місце четвертого числа.

Приклад додавання всіх 32-розрядних чисел одного масива наведено в лістингу 19.13

Лістинг 19.13: програма додавання всіх чисел одного масива

```
        .686p       ; директива определения типа микропроцессора
        .model flat,stdcall ; задание линейной модели памяти
      .XMM
        option casemap:none         ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32,fpu
.data
        mas1 dd  1.,2.,3.,4.
        _res DD 4 dup(0) ; масив для збереження результатів порівняння
        _title db "Сложение чисел массива ",0 ; назва вікна
_message db "Сложение 32-разрядных дробных чисел одного массива", 10,13,
"с использованием команд SSE", 10,13, "mas1:  1., 2., 3., 4.", 10,13, "Сумма: "
        buf db 256 dup(0),0 ; буфер
.code
_st:   movaps XMM0, mas1                ; XMM0:= 4. 3. 2. 1.
        movaps XMM1, XMM0               ; XMM1:= 4. 3. 2. 1.
        shufps XMM1,XMM1,11111001b  ; XMM1:= 4. 4. 3. 2.
        addss xmm0,xmm1                 ; XMM0:= 4. 3. 2. 3.
        shufps XMM1,XMM1,11111001b  ; XMM1:= 4. 4. 4. 3.
        addss xmm0,xmm1                 ; XMM0:= 4. 3. 2. 6.
        shufps XMM1,XMM1,11111001b  ; XMM1:= 4. 4. 4. 4.
        addss xmm0,xmm1                 ; XMM0:= 4. 3. 2. 10.
        movups _res, XMM0 ; запис результату у вихідний масив
      fld dword ptr _res
      invoke FpuFLtoA,0,2,ADDR buf,SRC1_FPU or SRC2_DIMM ;
      invoke MessageBox, 0, addr _message,addr _title, MB_OK ;
      invoke ExitProcess, 0
        end _st
```

Результат виконання програми наведено на рис. 19.21.

Рис. 19.21. Результат виконання програми

Для пересилання даних використано команду movaps, яка пересилає вирівняні по 16-байтовій межі 4 × 32-розрядних операнди.

Для скорочення коду можна додавання чисел використати в циклі:

```
        len1 equ ($-mas1)/type mas1
        …
        mov ecx,len1-1
        movaps XMM0, mas1
        movaps XMM1, XMM0
m1:       shufps XMM1,XMM1,11111001b
        addss xmm0,xmm1
        loop m1
```

19.8. Команди управління кешуванням

Необхідність управління кешуванням викликана тим, що більшість мультимедійних застосувань оперують великими обсягами даних, при цьому

може трапитися, що дані, завантажені в кеш, ніколи не знадобляться. Щоб оптимізувати роботу кеша, в систему команд SSE-розширення і були включені команди управління кешем.

MASKMOVQ приймач, джерело	Вибіркове збереження в пам'яті упакованих даних MMX-регістра

Команда **maskmovq** виконує вибіркове збереження в пам'яті упакованих даних MMX-регістра. Як операнд-джерело використовується один з MMX-регістрів, а операндом-приймачем служить область пам'яті, адреса якої задана в регістрі EDI. Маска указує, які байти будуть збережені в пам'яті. Вона формується із старших розрядів кожного байта, що знаходиться в MMX-регістрі:

maskmovq mem, mem.

MOVNTQ приймач, джерело	Запис в пам'ять, минаючи кеш, цілочислових упакованих даних у форматі MMX

Команда **movntq** виконує запис в пам'ять, минаючи кеш, цілочислових упакованих даних у форматі MMX. Операндом-джерелом тут виступає MMX-регістр, а операндом-приймачем – 64-розрядний елемент пам'яті:

movntq mem64, mmx.

MOVNTPS приймач, джерело	Запис в пам'ять, минаючи кеш, упакованих чисел з плаваючою точкою в короткому форматі

Команда **movntps** виконує запис в пам'ять, минаючи кеш, упакованих чисел з плаваючою точкою в короткому форматі. Операндом-джерелом тут виступає XMM-регістр, а операндом-приймачем – 128-розрядний елемент пам'яті, адреса якої повинна бути вирівняною по 16-байтовій межі:

movntps mem128, xmm.

PREFETCHT0 mem8	Перенести 64-розрядні дані у кеш T0
PREFETCHT1 mem8	Перенести 64-розрядні дані у кеш T1
PREFETCHT2 mem8	Перенести 64-розрядні дані у кеш T2
PREFETCHTA mem8	Перенести 64-розрядні дані у кеш NTA

Команда prefetcht mem8 переміщує дані з адреси mem8 в кеш:

- T0 – переміщує дані в кеш всіх рівнів;
- T1 – переміщує дані в кеш всіх рівнів, крім нульового;
- T2 – переміщує дані в кеш всіх рівнів, крім нульового та першого;
- NTA – переміщує дані в кеш для постійних даних.

Приклад використання цієї команди може бути таким:

```
mas1  DQ 12.13, -15.01, -325.12, 77.1
…
prefetcht0 mas1
```

19.9. Асемблювання та компонування програми через безпосереднє звертання до Visual Studio 2008

Послідовність дій може складатися з декількох етапів (тільки для SSE1).

Етап 1. Настроювання асемблера ml.exe.

Відкрити програму Visual Studio 2008 та прописати шлях находження ml.exe та його параметрів.

Вибрати в меню програми Tools\Externel Tools, у вікні, яке з'явиться (рис. 19.22), натиснути Add. У рядок Title написати ім'я, наприклад masm32.

Рис. 19.22. Вигляд вікна прописування шляхів та параметрів асемблювання Visual Studio 2008 для masm32

У рядку Command натиснути на клавішу ... та вказати шлях находження ml.exe: C:\masm32\bin\ml.exe.

У рядок Arguments вставити параметри асемблювання: /c /coff /Cp /nologo /Fm /Zi /Zd /Fl $(ItemFileName).asm. Обов'язково перевірити, щоб перед першим символом параметра не було пропуску.

У рядок Initial directory прописати параметр $(ItemDir).

Зняти прапорець Close on exit.

Натиснути на кнопку Apply.

Етап 2. Настроювання лінкера link.exe.

Для цього необхідно прописати шлях находження файлу link.exe та його параметрів.

У вікні External Tools натиснути Add. У рядку Title написати ім'я, наприклад link32.exe. У рядок Title написати ім'я, наприклад link32 (рис. 19.23).

У рядку Command натиснути на клавішу ... та вказати шлях находження link: C:\masm32\bin\link.exe.

У рядок Arguments вставити параметри асемблювання /SUBSYSTEM:CONSOLE /DEBUG /VERSION:4.0 $(ItemFileName).obj.

У рядок Initial directory прописати параметр $(ItemDir).

Зняти прапорець Close on exit.

Натиснути на кнопку Apply.

Етап 3. Виконання програми.

Для цього необхідно завантажити файл в Visual Studio 2008: File\Open\File…

Виконати **асемблювання**: Tools\ masm32. У результаті таких дій створиться .obj-файл, який розташується за адресою C:\masm32\bin\sse1_2.obj.

Виконати **лінкування**: Tools\ link32.exe. Натиснути Enter. Буде створено .exe-файл, який знаходитиметься за адресою C:\masm32\bin\sse1_2.exe.

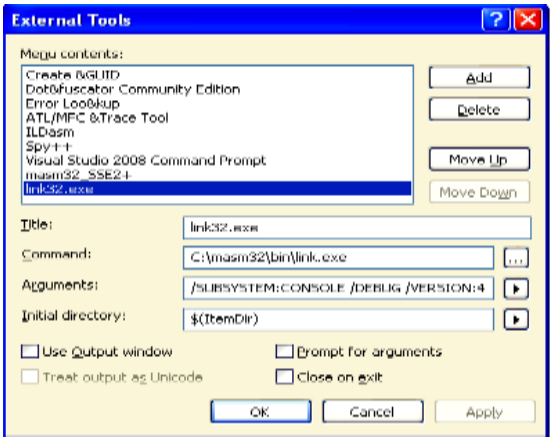


Рис. 19.23. Вигляд командного рядка Visual Studio 2008 Command Prompt

19.10. Лабораторна робота "SSE-команди"

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм з використанням команд SSE, написаних мовою асемблера для програмування МП платформи x86.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати свій варіант та написати на асемблері за допомогою SSE-команд програму обчислення одного з виразів з використанням дійсних чисел та ***виведенням варіанту завдання та його вмісту, результату і прізвища автора*** програми на екран за допомогою функції MessageBoxIndirect.

Методичні рекомендації

Програми з використанням команд SSE першого доповнення можна виконувати на masm32. Однак вважається доцільним виконувати асемблювання та лінкування в прогресивнішому середовищі, в такому, як Visual Studio.

Завдання:

1. Виконати паралельне порівняння за допомогою SSE-команд 2-х масивів по 11 чисел. Якщо всі числа другого масиву більші від усіх чисел першого, то виконати операцію $(2a - e)b^2 - d/\sqrt{b}$, де $a = 0{,}1$; $b = 1{,}05$; $c = 2{,}1$; $d = 3{,}2$;
інакше – виконати операцію $d/\sqrt{b}$.

2. Виконати операцію паралельного логічного множення чисел за допомогою SSE-команд. Якщо всі числа другого масиву більші від першого, то виконати операцію $(de - 2e)/(c - 2^a)$, де $a = 3{,}1$; $b = 9{,}11$; $c = 2{,}3$; $d = 5{,}5$;
інакше – виконати операцію $de - 2e$.

3. Виконати паралельне порівняння за допомогою SSE-команд 2-х масивів по 10 чисел. Якщо всі числа першого масиву більші від другого, то виконати операцію $(2^a - c)/b + 2^c$, де $a = 0{,}2$; $b = 10{,}05$; $c = 2{,}3$;
інакше – виконати операцію $2^a - c$.

4. Виконати паралельне порівняння за допомогою SSE-команд 2-х масивів по 9 чисел. Якщо всі числа одного масиву менші від другого, то виконати операцію $2^a - e/c^3 - ab$, де $a = 0{,}2$; $b = 10{,}05$; $c = 2{,}3$; $e = 21{,}07$,
інакше – виконати операцію e/c^3.

5. Виконати паралельну операцію логічного *АБО* за допомогою SSE-команд 2-х масивів. Якщо всі значення другого числа результату більше 25, то виконати операцію $2^a/b - cd^3$, де $a = 4{,}5$; $b = 12{,}3$; $c = 2{,}5$; $d = 6{,}5$;
інакше – виконати операцію $2^a/b$.

6. Виконати операцію паралельного логічного додавання за допомогою SSE-команд над масивами чисел. Якщо друге число результату порівняння більше 55, то виконати операцію $a - e/b - 2^d e$, де $a = 0{,}2$; $b = 8{,}05$; $c = 2{,}2$; $d = 3{,}3$;
інакше – виконати операцію $2^d e$.

7. Виконати операцію паралельного логічного множення за допомогою SSE-команд над масивами чисел. Якщо друге число результату порівняння більше 555, то виконати операцію
$(b + 4d)/ba - 2^c$, де $a = 0{,}2$; $b = 7{,}05$; $c = 2{,}3$; $d = 3{,}4$;
інакше – виконати операцію $b + 4d$.

8. Виконати паралельне обчислення квадратного кореня з масиву чисел і додати всі отримані результати. Якщо 0 та 1 біти дорівнюють одиницям, то виконати операцію $a^2 e - d/c - 2^b$, де $a = 1{,}2$; $b = 5{,}05$; $c = 0{,}3$; $d = 3{,}5$;
інакше – виконати операцію $a^2 e$.

9. Виконати паралельне обчислення мінімального значення для 2-х масивів. Якщо всі числа одного масиву менші від другого, то виконати операцію
$(ab^3 - d)/b - 2^d$, де $a = 3{,}6$; $b = 9{,}17$; $c = 2{,}8$; $d = 3{,}9$;
інакше – виконати операцію $ab^3 - d$.

10. Виконати операцію паралельного додавання за модулем за допомогою SSE-команд над масивами цілих чисел. Якщо друге слово менше 15, то виконати операцію $(b - d\sqrt{c})/2^a - c$, де $a = 0{,}2$; $b = 6{,}05$; $c = 1{,}3$; $d = 3{,}5$;
інакше – виконати операцію $b - d\sqrt{c}$.

Зміст звіту

1. Постановка задачі для конкретного варіанта.

2. Лістинг програми з виведенням даних на екран монітора з використанням API-функцій та з детальним коментарем і описом роботи.

3. Print screen екрана 32-розрядного налагоджувача з виконаною програмою.

5. Короткий опис виконання програми та висновки.

Приклад 19.2. Виконати паралельне порівняння за допомогою SSE-команд 2-х масивів по 8 чисел. Якщо всі елементи першого масиву більші ніж однойменні елементи другого масиву, то виконати операцію $(a - c)b - d/b$, де $a = 3{,}0$; $b = 0.2$; $c = 1{,}0$; $d = 2{,}2$; інакше – виконати операцію d/b.

Лістинг 19.14:

```
title Rysovaniy A.N.                                    rysov@rambler.ru
.686p                  ; директива определения типа микропроцессора
.model flat,stdcall ; задание линейной модели памяти и соглашения ОС Windows
.XMM                                    ; директива визначення команд SSE
option casemap:none                     ; отличие строчных и прописных букв
include \masm32\include\\user32.inc   ; файлы  интерфейса …
include \masm32\include\kernel32.inc ; системные функции применений…
include \masm32\include\fpu.inc
include \masm32\include\windows.inc ; файлы структур, констант …
includelib \masm32\lib\user32.lib
includelib \masm32\fpulib\fpu.lib
includelib \masm32\lib\kernel32.lib
  BSIZE equ 14                          ; задание реального количества байтов
.data                                   ; директива определения данных
mas1  dd  1.4,  2.1, 3.8, 4.6, 5.4, 6.12,7.54, 8.1,  9.0,  10.65
mas2  dd -1.34,-5.0,-3.54,1.5,-5.8,-6.53,7.5, -8.34,-9.54,-10.1
len equ ($-mas2)/ type mas2  ; кількість чисел масиву mas2
a   dd  3.0 ;
b   dd  0.2 ;
c1  dd  1.0 ;
d   dd  2.2 ;
info db "x1 = "                 ; підпис числа для віконця результату
buf BYTE BSIZE dup(?),0 ; розмір буфера для символів, які перетворяться
st2 db "Результат",0            ; назва віконця
.code                           ; директива начала кода программы
_st:                            ; метка начала программы с именем_st
mov eax, len            ; кількість подвійних слів у масиві
mov ebx,4               ; кількість подвійних слів у XMM регістрі
xor edx,edx             ; підготування до ділення
div ebx ; визначення кількості циклів для паралельного обробки
mov ecx,eax             ; лічильник циклів для паралельного зчитування
lea esi,mas1            ; занесення в esi адреси масиву mas1
lea edi,mas2            ; занесення в esi адреси масиву mas2
next:  movups XMM0,[esi]    ; пересилання 4-х 32 чисел з mas1
movups XMM1,[edi]       ; пересилання 4-х 32 чисел з mas2
cmpltps XMM0,XMM1   ; порівняння на менше: якщо менше, то нулі
movmskps ebx,XMM0   ; перенесення знакових бітів
add esi,16          ; підготування адреси для нового зчитування mas1
add edi,16          ; підготування адреси для нового зчитування mas2
dec ecx       ; зменшення лічильника циклів паралельного зчитування
```

```
        jnz m1                  ; перевірка лічильника на ненульове значення
        jmp m2                  ;
m1:     shl ebx,4               ; зсув ліворуч на 4 біти
        jmp next                ; на новий цикл
m2:     cmp edx,0                       ; перевірка остачі
        jz _end                 ; якщо в остачі нуль, то на мітку _end
        mov ecx,edx   ; якщо в остачі не нуль, то встановлення лічильника
m4:
movss XMM0,dword ptr[esi] ; пересилання одного 32-розр. числа з mas1
movss XMM1,dword ptr[edi] ; пересилання одного 32-розр. числа з mas2
        comiss XMM0,XMM1            ; порівняння молодших чисел масивів
        jg m6                   ; якщо більше
        shl ebx,1               ; зсув ліворуч на 1 розряд
        or ebx,1                ; встановлення 1, оскільки XMM0[0] < XMM1[0]
        jmp m3
m6:     shl ebx,1               ; зсув ліворуч на 1 розряд
m3:     add esi,4               ; адреса для нового числа mas1
        add edi,4               ; адреса для нового числа mas2
        loop m4
_end:   cmp ebx,0                       ; перевірка знакових бітів
        finit
        jz mb                   ; якщо ebx = 0, то перейти на мітку mb
        fld d                   ; занесення d
        fdiv b                  ; d/b
        jmp m5
mb:     fld  a                  ;
        fsub c1                 ; a – c
        fmul b                  ; (a – c)b
        fld  d                  ; занесення d
        fdiv b                  ; d/b
        fsubp  st(1),st(0)      ; (a – c)b – d/b
m5:
invoke  FpuFLtoA, 0, 10, ADDR buf,SRC1_FPU or SRC2_DIMM ; 1-е число – в FPU
invoke MessageBox,NULL,addr info,ADDR st2,MB_ICONINFORMATION
invoke ExitProcess, 0 ; повернення управління ОС та вивільнення ресурсів
        end _st                 ; директива окончания программы
```

У програмі для паралельного порівняння чотирьох 32-розрядних чисел використовується цикл з міткою next. У регістр ebx записуються всі знакові біти чисел, які перевіряються. Числа масивів, які не перевіряються паралельно, залежно від змісту регістра edx обробляються поодинці (скалярно).

Результат виконання програми з лістингу 19.8 наведено на рис. 19.24.

Рис. 19.24. Результат виконання програми з лістингу 18.8

Приклад 19.3. Числа $a \in \{a1, a2, a3\}$ задані масивом та мають розмірність Real4. Провести розрахунки рівняння $d/b\ sqrt(a) + a$ за допомогою команд SSE.

Лістинг 19.15:

```
        .686
        .XMM
        .model flat, stdcall
        option casemap :none
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32,user32,fpu,masm32
.data
        buf db 80 dup(?)
        buf2 db 20 dup(?)
        term db 0ah
        arr1 real4   16.,25.,36.,1244.
        len1 equ ($-arr1)/type arr1
        arr2 real4    2.,4.,16.       ; b, c, d
        res dd ?
.code      ;ml.exe /c /coff     ;link.exe /subsystem:console
main:
     mov ecx,len1
     mov ebx,0
     movss xmm1, arr2[0]         ; xmm1 - b
     movss xmm2, arr2[4]     ; xmm2 - c
     movss xmm3, arr2[8]     ; xmm3 - d
     divss xmm3, xmm1  ; d/b
cycle:
        movss xmm0,arr1[ebx]    ; xmm0 - a
        sqrtss xmm4,xmm0      ;sqrt(a)
        mulss xmm4,xmm3       ; d/b x sqrt(a)
        addss xmm4,xmm0       ; d/b x sqrt(a) + a
        ;cvttss2si eax,xmm4   ; для отладки
        movss res,xmm4
        fld res
        add ebx,4
        push ebx
        push ecx
invoke FpuFLtoA,0,5,addr buf2,SRC1_FPU or SRC2_DIMM
invoke szCatStr,addr buf,addr buf2
invoke szCatStr,addr buf,addr term
        pop ecx
        pop ebx
        dec ecx
        jnz cycle
invoke MessageBox,0,addr buf,NULL,MB_ICONINFORMATION
        invoke ExitProcess, 0
        end main
```

Результат виконання програми наведено на рис. 20.17.

Рис. 20.17. Результат виконання програми

Завдання для практичного використання

1. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати паралельне ділення операндів командою **DIVPS** одного масиву на інший. Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати розпакування чисел, операцію $d/\sqrt{b}$ та вивести результат через MessageBox.

2. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати скалярне ділення операндів командою **DIVSS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати розпакування чисел, операцію $(2a - e)$ та вивести результат через MessageBox.

3. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати паралельне обчислення максимального значення для кожної пари упакованих чисел з плаваючою точкою командою **MAXPS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати розпакування чисел, операцію (c – 2), та вивести результат через MessageBox.

4. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати паралельне обчислення мінімального значення для кожної пари упакованих чисел командою **MINPS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати розпакування чисел, операцію $2 - e/c$, та вивести результат через MessageBox.

5. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати скалярне обчислення максимального значення елементів масивів командою **MAXSS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати розпакування чисел, операцію $b - cd$, та вивести результат через MessageBox.

6. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати скалярне обчислення мінімального значення елементів масивів командою **MINSS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати операцію $c - d$, та вивести результат через MessageBox.

7. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати паралельне обчислення обернених значень упакованих операндів командою **RCPPS.** Отриманий масив має вигляд masRes

:={a,b,c,d,e}. Виконати розпакування чисел, операцію *в + с*, та вивести результат через MessageBox.

8. Задано масив. Виконати скалярне обчислення оберненого значення командою **RCPSS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати операцію *с/3*, та вивести результат через MessageBox.

9. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати паралельне обчислення обернених значень квадратного кореня командою **RSQRTPS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати розпакування чисел, операцію *вd + са*, та вивести результат через MessageBox.

10. Задано масив. Виконати скалярне обчислення оберненого значення квадратного кореня командою **RSQRTSS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати операцію *е/b +с*, та вивести результат через MessageBox.

11. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати паралельну перевірку умови рівності командою **CMPEQPS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати розпакування чисел, операцію *d + са*, та вивести результат через MessageBox.

12. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати паралельну перевірку умови «менше» командою **CMPLTPS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати розпакування чисел, операцію *в/d + с*, та вивести результат через MessageBox.

13. Задано 2 масиви. Виконати паралельну перевірку «менше або дорівнює» командою **CMPLEPS.** Отриманий масив має вигляд masRes :={a,b,c,d,e}. Виконати розпакування чисел, операцію *ad + с*, та вивести результат через MessageBox.

14. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати паралельну перевірку умови «невпорядкованості» командою **CMPUNORDPS.** Виконати розпакування чисел та результати операції вивести через MessageBox.

15. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати паралельну перевірку умови «недорівненості» командою **CMPNEQPS.** Виконати розпакування чисел та результати операції вивести через MessageBox.

16. Задано 2 масиви. Виконати паралельну перевірку умови «не менше» командою **CMPNLTPS.** Виконати розпакування чисел та результати операції вивести через MessageBox.

17. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати перевірку умови рівності командою командою **CMPEQSS.** Результати операції вивести через MessageBox.

18. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати перевірку умови «менше» елементів масивів командою **CMPLTSS.** Результати операції вивести через MessageBox.

19. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати перевірку умови «менше або дорівнює» елементів масивів командою **CMPLESS.** Результати операції вивести через MessageBox.

20. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати перевірку умови «не дорівнює» елементів масивів командою **CMPNEQSS.** Результати операції вивести через MessageBox.

21. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати скалярне порівняння елементів масивів командою **COMISS/UCOMISS.** Результати операції вивести через MessageBox.

22. Задано 2 масиви з числом елементів, які не вміщуються в розрядність регістра SSE. Виконати перевірку умови «не менше або дорівнює» елементів масивів командою **CMPNLESS.** Результати операції вивести через MessageBox.

Контрольні запитання

1. Яке призначення команд SSE-розширення?
2. Який формат даних SSE-розширень?
3. Який суфікс мають команди паралельних арифметичних операцій та для чого ці команди використовують?
3. Який суфікс мають команди скалярних арифметичних операцій та для чого ці команди використовують?
4. Які події при обробці даних команд SSE-розширень викликають виняткові ситуації?
5. На які групи поділяються команди SSE-розширень?

20. SSE2-РОЗШИРЕННЯ

Технологію SSE2 (Streaming SIMD Extensions 2) розроблено для застосування в процесорах Intel Pentium 4. Її призначення – підвищити ефективність операцій з 128-розрядними даними у форматі плаваючої точки з подвійною точністю (double-precision floating point) та з цілочисловими даними. Ця технологія дозволяє розробляти високопродуктивні застосування для 3D-графіки і 3D-геометрії, моделювання і симуляції процесів, обробки сигналів, 3D-анімації, кодування/декодування, розпізнавання мови і так далі.

Технологія SSE2 розширює можливості MMX за рахунок використання 128-розрядних регістрів замість 64-розрядних, забезпечуючи високу ефективність паралельних обчислень. Досягнення вищої продуктивності можливе також за рахунок включення в SSE2 нових типів даних: 128-розрядних операндів з плаваючою точкою подвійної точності і 128-розрядних упакованих цілих чисел. Технологія SSE2 дозволяє:

- поліпшити управління даними в кеші;
- підвищити продуктивність операцій, що вимагають вищої точності;
- розширити до 128 бітів діапазон оброблюваних 64-розрядними командами операндів.

До групи SSE2 входять інструкції, що виконують 144 нові операції. Якщо ж підрахувати кількість нових імен інструкцій, то їх опиниться значно менше. Розбіжність пояснюється тим, що одному імені інструкції може відповідати декілька різних кодів операцій. Типовий приклад інструкції пересилання даних, у яких код операції залежить від напряму пересилання, з регістра в пам'ять або з пам'яті в регістр.

Більшість нових інструкцій двоадресні. Перший операнд є приймачем (dest), а другий – джерелом (src). Приймач, як правило, знаходиться в 128-бітовому регістрі XMM, джерело може знаходитися як в регістрі XMM, так і в оперативній пам'яті. Виключенням є тільки інструкції пересилання, у яких приймач може розташовуватися в пам'яті. Третій операнд (якщо він є) – це ціле число, розмір якого не перевищує одного байта.

При роботі з дійсними числами можливе виконання тієї самої операції над однією парою або над двома парами чисел. У першому випадку ім'я операції закінчується буквами **sd** (скалярні дані), а в другому – буквами **pd** (упаковані дані). Для SSE2 2 вираз "упаковані дані" означає, що два 64-бітових числа розташовані в одному 128-бітовому регістрі або в пам'яті підряд одне за одним. А вираз "скалярні дані" означає, що одне 64-бітове число розташоване в молодшій половині 128-bit регістра.

SSE-команди використовують вісім 128-розрядних регістрів (XMM0 – XMM7) і можуть працювати в скалярному або паралельному режимі. SSE2-команди оперують з такими типами даних:

– упаковані і скалярні числа з плаваючою точкою в короткому форматі;

– упаковані і скалярні числа з плаваючою точкою подвійної точності;

– упаковані і скалярні цілі числа розміром 128 бітів.

Формат даних 64-розрядного упакованого числа SSE2-розширення наведено на рис. 20.1.

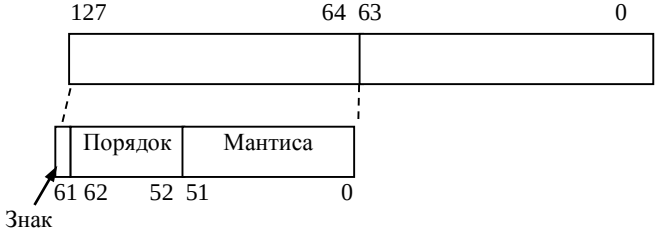


Рис. 20.1. Формат даних 64-розрядного упакованого числа SSE2-розширення

За допомогою паралельних команд можна одночасно обробляти всі упаковані операнди, тоді як за допомогою скалярних – тільки молодший операнд. Команди SSE2-розширення в більшості випадків вимагають вирівнювання адрес операндів в пам'яті по 16-байтовій межі, хоча з цього правила є деякі винятки, наприклад команда завантаження або збереження операнда в невирівняній області пам'яті.

Перш ніж розробляти програмний код з використанням технології SSE2, необхідно переконатися в тому, що процесор її підтримує. Для цього потрібно, виконавши команду cpuid, проаналізувати 26-й біт регістра EDX: якщо цей біт відмінний від нуля, то технологія SSE2 підтримується процесором.

Для визначення підтримки SSE-команд можна скористатися програмою з лістингу 20.1.

Лістинг 20.1. Тестова програма для визначення підтримки SSE2-команд:

```
    .686p                          ; директива определения типа микропроцессора
    .model flat                          ; лінійна модель пам'яті
    option casemap: none           ; отличие строчных и прописных букв
    .data                           ; директива определения данных
    testSSE2  DB  1
.code                                ; директива начала кода программы
st:                                   ; метка начала программы с именемst
    mov   EAX, 1
    cpuid                             ; ідентифікація мікропроцесора
    test  EDX, 4000000h               ; edx:= edx ∧ 400 0000h  (26 розряд)
    jnz    exit                       ; перейти на exit, якщо не нуль
    mov  testSSE2, 0
exit:
    xor   EAX, EAX
    mov  AL, testSSE2
      ret                             ; повернення управління  ОС
    end st                            ; завершение программы с именемst
```

Програма з лістингу 20.1 повертає 1 у регістр AL, якщо технологія SSE підтримується процесором.

Команди обробки 128-розрядних даних з плаваючою точкою поділяються на такі групи команд:

- переміщення (пересилання, передавання) даних;
- арифметичні (складання, віднімання, множення, ділення, добування квадратного кореня і пошуку максимуму/мінімуму);
- порівняння;
- логічних операцій;
- розпаковування і розподілення даних;
- перетворення форматів даних;
- управління станом обчислень;
- управління кешуванням даних.

20.1. Команди передавання даних

20.1.1. Обмін 128-бітовими кодами між оперативною пам'яттю (ОЗП) та регістрами xmm

Для вирівнювання 128-розрядного коду ці інструкції необхідно розташовувати за адресами, кратними 16 розрядам. Передбачено спеціальну інструкцію для запису коду в ОЗП без кешування, тобто при її виконанні вміст кеша не змінюється.

MOVNTPD приймач, джерело	Пересилання 128-розрядних кодів в ОЗП без кешування за адресами, кратними 16

Команда **movntpd** пересилає 128-розрядні коди в ОЗП без кешування. Код, що пересилається, повинен мати адресу, кратну 16, інакше виникає аварійна ситуація (порушення захисту):

```
movntpd  m128, xmm.
```

MOVAPD приймач, джерело	Пересилання 128-розрядних кодів між ОЗП та регістром xmm за адресами, кратними 16

Команда **movapd** пересилає 128-розрядні коди з ОЗП в регістр xmm. Код, що пересилається, повинен мати адресу, кратну 16, інакше виникає аварійна ситуація (порушення захисту):

```
movapd  xmm, xmm/m128;
movapd  xmm/m128, xmm.
```

MOVUPD приймач, джерело	Пересилання 128-розрядними кодами між ОЗП і регістрами xmm

Команда **movupd** пересилає 128-розрядні коди між ОЗП та регістром xmm. Ця інструкція призначена для застосування в тих випадках, коли дані не вирівняні або коли невідомо, вирівняні вони чи ні:

```
movupd  xmm, xmm/m128;
movupd  xmm/m128, xmm.
```

Програма для перевірки команд пересилання та процесу формування виведення даних наведена у лістингу 20.2.

Лістинг 20.2:

```
.686p ;          директива определения типа микропроцессора
.model flat,stdcall ; завдання лінійної моделі пам'яті та угоди ОС Windows
.XMM                        ; директива визначення команд SSE
option casemap:none    ; отличие строчных и прописных букв
include \masm32\include\windows.inc   ; файлы структур, констант …
include \masm32\macros\macros.asm
uselib kernel32, user32
.data
mas1 dq 1.,2.                    ; масив 1
titl1 db "SSE2-команды",0      ; назва вікна
fmt db "x1 = %d, x2 = %d",0    ; форматування чисел
buf db 2 dup(?)                  ; буфер
res dd 2 dup(0),0
.code
 _st:
lea ebx,res
  movupd XMM0,mas1            ; занесення mas1 до XMM0
  cvtpd2pi MM0,XMM0      ; перетворення у 32-розрядне число
  movq qword ptr[EBX], MM0
    invoke wsprintf,addr buf,addr fmt,res,res[4]  ; перетворення числа
    invoke MessageBox,NULL,addr buf,addr titl1,MB_ICONINFORMATION;
    invoke ExitProcess,0    ; звільнення ресурсів та передача управління  ОС
      end _st       ; кінець програми з ім'ям _st
```

Для підготування даних використовована функція wsprintf, яка перетворює цілі числа в символи для виведення. Тому на початку програми використана функція cvtpd2pi для перетворення результату у 32-розрядні цілі числа. Результати роботи програми наведені на рис. 20.2.

Рис. 20.2. Результат роботи програми

20.1.2. Обмін 64-розрядними кодами між ОЗП і регістрами xmm

Якщо дані “вирівняні”, то звернення до ОЗП займає 1 такт, а якщо “вирівняні” – 2 такти. Вирівнювання означає, що адреса 64-розрядних кодів повинна бути кратною 8.

MOVSD приймач, джерело	Пересилання скалярних даних між ОЗП та молодшими 64 розрядами регістра xmm

Команда movsd пересилає дані між ОЗП та молодшими 64 розрядами регістра xmm. При копіюванні вмісту одного регістра xmm в інший, вміст старшої половини приймача не змінюється:

```
movsd xmm, xmm/m64;
movsd xmm/m64, xmm.
```

MOVHPD приймач, джерело	Пересилання даних між ОЗП та старшими 64 розрядами регістра xmm

Команда movhpd пересилає дані між ОЗП та старшими 64 розрядами регістра xmm:

```
movhpd xmm, m64;
movhpd m64, xmm.
```

MOVLPD приймач, джерело	Пересилання даних між ОЗП та молодшими 64 розрядами регістра xmm

Команда **movlpd** пересилає дані між ОЗП та молодшими 64 розрядами регістра xmm:

```
movlpd xmm, m64;
movlpd m64, xmm.
```

20.1.3. Використання масок

Ці інструкції працюють зі знаковими розрядами регістрів xmm. Одна з них копіює стан 63-го і 127-го розрядів в регістр загального призначення (r32). В документації це називається формуванням знакової маски.

Друга інструкція дозволяє вибірково змінювати вміст 16 байтів при накладенні на екрані одного рисунка на іншій (на початковий фон). Для цього застосовується маска, яка вказує, які байти підлягають зміні, а які не підлягають. Код (або частина коду) точки займає 1 байт. Код маски для кожної точки так само займає 1 байт. Якщо знаковий розряд байта маски встановлений, то код точки початкового фону замінюється новим, інакше він залишається без змін. Зазвичай інструкція працює з відеопам'яттю, тому запис проводиться, минаючи кеш. Початкова адреса пам'яті повинна знаходитися в регістрах ds:di або ds:edi. Вирівнювання адреси не потрібне, тому що в даному випадку така вимога була б безглуздою.

MOVMSKPD приймач, джерело	Пересилання вмісту знакових розрядів xmm в два молодших розряди r32

Команда **movmskpd** пересилає вміст знакових розрядів xmm (63 та 127) в два молодших розряди r32 (0 і 1), решта розрядів r32 очищається:

```
movmskpd r32, xmm.
```

MASKMOVDQU приймач, джерело	Пересилання за маскою

Команда **maskmovdqu** виконує пересилання за маскою. Перший операнд містить код, що пересилається, а другий операнд – маску. Третій операнд явно не указується, його адреса повинна знаходитися в DS:DI/EDI/RDI.

Числа можуть бути різного формату, але в тім же форматі повинна бути й маска. Наприклад, якщо числа в форматі DB, то й маска повинна бути 0FFh. Якщо числа в форматі DW, то й маска повинна бути 0FFFFh.

Формат команди:

maskmovdqu xmm, xmm.

Використання команди maskmovdqu для пересилання чисел за маскою наведено в лістингу 20.3.

Лістинг 20.3. Програма пересилання чисел за маскою:

```
        .686                    ; директива определения типа микропроцессора
        .model flat, stdcall ; задание линейной модели памяти
        .XMM                    ; директива визначення команд XMM
        option casemap:none    ; отличие строчных и прописных букв
        include \masm32\include\windows.inc ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32,masm32
        BSIZE equ 350
.data
        mas1 dd 10.0, 14.0, 54.0, 23.0, -53.0, -9.0;
        len1 EQU ($-mas1)/type mas1 ; обчислення кількості чисел масиву mas1
      mask1 dd 0EFFFFFF0h, 0, 08FFFFFFh, 0, 0,0FFFFFFF0h;
        len2 EQU ($-mask)/type mask
        res1 dd len1 dup(?), 0
        titl1 db "Команда MMX maskmovdqu",0
_info db "Результат параллельной обработки командой maskmovdqu:",0ah,0dh,
"Массив чисел:    10.0,  14.0,  54.0,  23.0,  -53.0,    -9.0;",0ah,0dh,
 "Массив-маска: F...F,      0,    F...F,      0,        0,     F...F;",0ah,0dh,0dh,
  "Массив-результат:  "
        buf dd BSIZE dup (?)
        frmt db " %d  ", 0
.code
_start:
        mov eax, len1 ; підготовка до визначення кількості циклів
        mov ebx, 4     ; кількість паралельно оброблюємих чисел за один цикл
        xor edx, edx  ; підготовка регістра до ділення (онулення EDX)
        div ebx          ; визначення кількості циклів  eax := 2 , edx := 1
        mov ecx, eax ; занесення кількості циклів до лічильника
        lea eax, mas1
        lea ebx, mask1
      lea edi, res1
@@:  movups XMM0, dword ptr [eax]
        movups XMM1, dword ptr [ebx]
        maskmovdqu XMM0, XMM1
        ;movups XMM0, dword ptr [edi]
        add ebx, 16
        add eax, 16
        add edi, 16
      loop @b
        cmp edx, 0
        jz _exit
        mov ecx, edx
@2: movss XMM0, dword ptr [eax]
        movss XMM1, dword ptr [ebx]
        maskmovdqu XMM0, XMM1
        ;  movss XMM0, dword ptr [edi]
        add eax,4
        add ebx,4
        add edi,4
      loop @2
_exit:  lea esi, res1
        mov ecx, len1
        xor ebx, ebx
@3: movups XMM0, dword ptr [esi]
```

```
        cvtss2si eax, XMM0
        push esi
        push ecx
        push ebx
    invoke wsprintf, ADDR buf[ebx], ADDR frmt, eax
        pop ebx
        pop ecx
        pop esi
        add esi, 4
        add ebx, 4
loop @3
        invoke MessageBox,NULL,addr _info,addr titl1,MB_ICONINFORMATION
        invoke ExitProcess,0 ; звільненя ресурсів та передача управління ОС
        end _start
```

Результат роботи програми наведено на рис. 20.3.

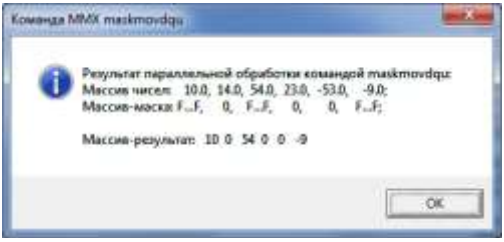


Рис. 20.3. Результат роботи програми

Для повного повторення числа при передачі необхідно, щоб маска дорівнювала 0FFFFFFFFh (всі одиниці). Для негативних чисел недостатньо, щоб тільки знаковий розряд був встановлений (в одиницю), а інші – ні.

Якщо регістр EDI ініціалізований, наприклад lea edi, res1, то немає необхідності використовувати послідовність команд:

```
maskmovdqu XMM0, XMM1
movups XMM0, dword ptr [edi]
```

Така послідовність не викличе помилки асемблювання, але друга команда буде зайвою.

В циклі @@ – @b відбувається паралельна обробки чисел за маскою. В циклі @2 відбувається скалярна обробка по одному числу за маскою.

В циклі @3 командою movups XMM0, dword ptr [esi] відбувається запис одного числа в регістр XMM0, а командою cvtss2si це число з дійсного перетворюється в ціле. Функція wsprintf перетворює це число в символ для виведення через MessageBox.

20.2. Арифметичні операції

20.2.1. Паралельні операції з упакованими числами

ADDPD приймач, джерело	Паралельне додавання двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності

Команда **addpd** виконує паралельне додавання двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати XMM-регістр або елемент пам'яті; вихідним операндом може бути тільки XMM-регістр:

addpd xmm, xmm/m128.

SUBPD приймач, джерело	Паралельне віднімання двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності

Команда **subpd** виконує паралельне віднімання двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати XMM-регістр або елемент пам'яті; вихідним операндом може бути тільки XMM-регистр. Команда віднімає вміст операнда-джерела з вмісту операнда-приймача і зберігає результат в операнді-приймачі:

subpd xmm, xmm/m128.

MULPD приймач, джерело	Паралельне множення двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності

Команда **mulpd** виконує паралельне множення двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати XMM-регістр або елемент пам'яті; вихідним операндом може бути тільки XMM-регістр:

mulpd xmm, xmm/m128.

DIVPD приймач, джерело	Паралельне ділення двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності

Команда **divpd** виконує паралельне ділення двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Команда приймає два операнди: як вхідний операнд може виступати XMM-регістр або елемент пам'яті; вихідним операндом може бути тільки XMM-регістр. Команда виконує ділення вмісту операнда-приймача на вміст операнда-джерела і зберігає результат в операнді-приймачі:

divpd xmm, xmm/m128.

SQRTPD приймач, джерело	Паралельне витягання квадратного кореня з двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності

Команда **sqrtpd** виконує паралельне добування квадратного кореня з двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати XMM-регістр або елемент пам'яті; вихідним операндом може бути тільки XMM-регістр:

sqrtpd xmm, xmm/m128.

Використання команди **sqrtpd** для обчислення квадратного кореня з масиву чисел та додавання всіх результатів наведено в лістингу 20.4.

Лістинг 20.4. Програма обчислення квадратного кореня з масиву чисел та додавання всіх результатів:

```
        .686p
        .xmm
        .model flat, stdcall
        option casemap: none
        include \masm32\include\windows.inc
        include \masm32\include\kernel32.inc
        include \masm32\include\user32.inc
        include \masm32\include\fpu.inc
        includelib \masm32\lib\user32.lib
        includelib \masm32\lib\kernel32.lib
        includelib \masm32\lib\fpu.lib
.data
        arr1 dq 4.0,36.0,9.0,16.0
        sz equ ($-arr1)/type arr1 ;
        res dq 2 dup(0)
        ttl db "Результат",0
        buf db ?
.CODE
_start:
        lea esi,arr1
        mov ecx,sz/2; (кількість циклів обробки (за 1 раз обробляються 2 числа)
@@:  movupd xmm0,[esi] ; помістити два числа з пам'яті за адресою esi
        sqrtpd xmm0,xmm0 ; витягувати корінь з хмм0
        movupd xmm1,res
        addpd xmm1,xmm0 ; додати число з пам'яті (старшу частину хмм0)
        movupd res,xmm1
        add esi,16              ; збільшити адресу (2dq=16)
        dec ecx
    jnz @b
   movlps xmm0,[res]
   movlps xmm1,[res+8]
   addsd xmm0,xmm1
   movlps res,xmm0
     finit
        fld qword ptr[res]
        invoke FpuFLtoA,0,2,addr buf,SRC1_FPU or SRC2_DIMM ; st0 в рядок
        invoke MessageBox,0,addr buf,addr ttl,MB_OK
        invoke ExitProcess,0
        END   _start
```

Результат роботи програми наведено на рис. 20.4.

Рис. 20.4. Результат роботи програми

В програмі рядок sz equ ($-arr1)/type arr1
обчислює кількість чисел в масиві arr1. А адреса цього масиву зберігається в регістрі esi командою завантаження адреси

lea esi,arr1.

Команда movupd xmm0,[esi]
завантажує в регістр два числа, з яких наступною командою

sqrtpd xmm0,xmm0

витягується два кореня. У звязку з тим, що в масиві arr1 чотири 64-розрядних числа, то цикл @@ виконується два рази і додавання парами старших й молодших чисел відбувається командою

addpd xmm1,xmm0.

Результат у вигляді суми старших та суми младших чисел заноситься до комірки пам'яті з іменем res командою

movupd res,xmm1.

Підготовка до додавання відбувається фрагментом коду:

movlps xmm0,[res]
movlps xmm1,[res+8].

А підготовка до виведення функцією MessageBox відбувається фрагментом коду:

```
fld qword ptr[res]
invoke FpuFLtoA,0,2,addr buf,SRC1_FPU or SRC2_DIMM ; st0 в рядок
```

MAXPD приймач, джерело	Паралельний пошук максимального значення в парах упакованих 64-розрядних чисел з плаваючою точкою подвійної точності

Команда **maxpd** виконує паралельний пошук максимального значення в парах упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати XMM-регістр або елемент пам'яті; вихідним операндом може бути тільки XMM-регістр. Результат зберігається у вихідному операнді:

maxpd xmm, xmm/m128.

MINPD приймач, джерело	Паралельний пошук мінімального значення в парах упакованих 64-розрядних чисел з плаваючою точкою подвійної точності

Команда **minpd** виконує паралельний пошук мінімального значення в парах упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати XMM-регістр або елемент пам'яті; вихідним операндом може бути тільки XMM-регістр. Результат зберігається у вихідному операнді:

minpd xmm, xmm/m128.

20.2.2. Арифметичні операції зі скалярними числами

При виконання арифметичних операцій зі скалярними числами дійсне число подвійної точності розташоване в молодших половинах регістрів xmm. Результат операцій розташовується в молодшій половині приймача, вміст його старшої половини не змінюється.

ADDSD приймач, джерело	Додавання молодших 64-розрядних операндів з плаваючою точкою подвійній точності

Команда **addsd** виконує складання молодших 64-розрядних операндів з плаваючою точкою подвійній точності. Команда приймає два операнди. Як вхідний операнд може виступати xmm-регістр або елемент пам'яті; вихідним операндом може бути тільки xmm-регістр:

addsd xmm, xmm/m64.

SUBSD приймач, джерело	Віднімання молодших 64-розрядних операндів з плаваючою точкою подвійної точності

Команда **subsd** виконує віднімання молодших 64-розрядних операндів з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати xmm-регістр або елемент пам'яті; вихідним операндом може бути тільки xmm-регістр. Команда віднімає вміст операнда-джерела з вмісту операнда-приймача і зберігає результат в операнді-приймачі:

subsd xmm, xmm/m64.

MULSD приймач, джерело	Скалярне множення молодших 64-розрядних операндів з плаваючою точкою подвійної точності

Команда **mulsd** виконує скалярне множення молодших 64-розрядних операндів з плаваючою точкою подвійної точності. Команда приймає два операнди: як вхідний операнд може виступати xmm-регістр або елемент пам'яті; вихідним операндом може бути тільки xmm-регістр. Команда не зачіпає старші частини операндів:

subsd xmm, xmm/m64.

DIVSD приймач, джерело	Скалярне ділення молодших 64-розрядних операндів з плаваючою точкою подвійної точності

Команда **divsd** виконує скалярне ділення молодших 64-розрядних операндів з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати xmm-регістр або елемент пам'яті; вихідним операндом може бути тільки xmm-регістр. Команда виконує ділення вмісту операнда-приймача на вміст операнда-джерела і зберігає результат в операнді-приймачі, не зачіпаючи старші частини операндів:

divsd xmm, xmm/m64.

SQRTSD приймач, джерело	Скалярне добування квадратного кореня з молодших 64-розрядних операндів з плаваючою точкою подвійної точності

Команда **sqrtsd** виконує скалярне добування квадратного кореня з молодших 64-розрядних операндів з плаваючою точкою (ПТ) подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати xmm-регістр або елемент пам'яті; вихідним операндом може бути тільки xmm-регістр. Старші операнди під час виконання не змінюються:

sqrtsd xmm, xmm/m64.

Програма паралельного ділення двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності, додавання отриманих результатів та обчислення з них квадратних корінів наведено в лістингу 20.5.

Лістинг 20.5:

```
        .686p ;          директива определения типа микропроцессора
        .xmm  ; директива визначення команд SSE
        .model flat, stdcall ;завдання лінійної моделі пам'яті та угоди ОС Windows
        option casemap: none; отличие строчных и прописных букв
        include \masm32\include\windows.inc   ; файлы структур, констант …
        include \masm32\macros\macros.asm
        uselib kernel32, user32,fpu
; Виконати паралельне ділення двох упакованих 64-розрядних чисел
; з плаваючою точкою подвійної точності.
; Додати всі результати та обчислити квадратний корінь.
.data
        arr1 dq 4.0,46.0,8.0,15.0
        sz equ ($-arr1)/type arr1
        arr2 dq 3.0,23.0,6.0,46.0
        res dq sz dup(0)
        a dq 0
        ttl db "Результат",0
        buf db ?
.CODE
_start:
        lea esi,arr1
        lea edi,arr2
        lea edx,res
        mov ecx,sz/2 ; за 1 раз обробляється 2 числа
@@:
        movupd XMM0,[esi] ; два числа подвійної точності в хмм0 з пам'яті
        movupd XMM1,[edi] ; два числа подвійної точності в хмм1 з пам'яті
        divpd XMM0,XMM1 ; поділити вміст регістра xmm0 на xmm1
        movhps a,XMM0 ; помістити старшу частину регістра хмм0 в пам'ять
        addpd XMM0,a  ;
      sqrtsd XMM0,XMM0 ; витягувати корінь з хмм0
        movlps qword ptr[edx],XMM0 ; помістити молодшу частину хмм0 в пам'ять
        add esi,16 ; збільшити адреси (2 dq = 16)
        add edi,16
        add edx,8
        dec ecx
    jnz @b
        lea edi,res
        lea esi,buf
        mov ecx,sz/2
@@:
        finit
        fld qword ptr[edi] ; для перетворення типу (dq->dt) завантажити в st0
invoke FpuFLtoA,0,2,esi,SRC1_FPU or SRC2_DIMM ; перетворити st0 в рядок
```

```
    add esi,5 ; додати до esi кількість символів виведених в буфер
     add edi,8
     dec ecx
jnz @b
    invoke MessageBox,0,addr buf,addr ttl,MB_OK
    invoke ExitProcess,0
    END   _start
```

Результат виконання програми наведено на рис. 20.5.

Рис. 20.5. Результат виконання програми

MAXSD приймач, джерело	Пошук максимального значення в парі молодших 64-розрядних операндів з ПТ подвійної точності

Команда **maxsd** виконує пошук максимального значення в парі молодших 64-розрядних операндів з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати xmm-регістр або елемент пам'яті; вихідним операндом може бути тільки xmm-регістр. Результат зберігається в молодшій частині вихідного операнда, не зачіпаючи старший операнд:

maxsd xmm, xmm/m64.

MINDS приймач, джерело	Пошук мінімального значення в парі молодших 64-розрядних операндів з ПТ подвійної точності

Команда **minds** виконує пошук мінімального значення в парі молодших 64-розрядних операндів з плаваючою точкою подвійної точності. Команда приймає два операнди. Як вхідний операнд може виступати xmm-регістр або елемент пам'яті; вихідним операндом може бути тільки xmm-регістр. Результат зберігається в молодшій частині вихідного операнда, не зачіпаючи старший операнд:

minsd xmm, xmm/m64.

20.3. Команди порівняння

20.3.1. Порівняння без зміни стану розрядів EFLAGS

Для кожної порівнюваної пари дійсних чисел при позитивному результаті порівняння встановлюються всі розряди приймача, інакше вони очищаються. Наприклад, якщо вміст регістра xmm порівняти з самим собою, то при перевірці відношення "дорівнює" всі 128 розрядів будуть встановлені, а при перевірці відношення "не дорівнює" всі 128 розрядів будуть очищені.

CMPPD приймач, джерело, маска	Паралельне порівняння двох пар упакованих 64-розрядних чисел з ПТ подвійної точності

Команда **cmppd (cmp*xxx*pd)** виконує паралельне порівняння двох дійсних чисел. Результат поміщається у відповідні половини приймача:

cmppd xmm, xmm/m128, imm8.

Умову, що перевіряється, задає третій операнд (табл. 20.1). У документації Intel обумовлено два способи вказання умови, що перевіряється.

Таблиця 20.1

Умова	Указується значення imm8	Використовується ім'я псевдооперації
Дорівнює	CMPPD xmm, xmm/m128, 0	CMPEQPD xmm, xmm/m128
Менше	CMPPD xmm, xmm/m128, 1	CMPLTPD xmm, xmm/m128
Менше або дорівнює	CMPPD xmm, xmm/m128, 2	CMPLEPD xmm, xmm/m128
Непорівнянні	CMPPD xmm, xmm/m128, 3	CMPUNORDPD xmm, xmm/m128
Не дорівнює	CMPPD xmm, xmm/m128, 4	CMPNEQPD xmm, xmm/m128
Не менше	CMPPD xmm, xmm/m128, 5	CMPNLTPD xmm, xmm/m128
Не менше або дорівнює	CMPPD xmm, xmm/m128, 6	CMPNLEPD xmm, xmm/m128
Порівнянні	CMPPD xmm, xmm/m128, 7	CMPORDPD xmm, xmm/m128

CMPSD приймач, джерело, константа	Скалярне порівняння молодших частин 128-розрядних операндів

Команда **cmpsd (cmp*xxx*sd)** виконує скалярне порівняння молодших частин 128-розрядних операндів. Результат поміщається в молодшу половину dest[l], вміст старшої половини dest[h] не змінюється:

cmpsd xmm, xmm/m128, imm8.

20.3.2. Команди порівняння, які змінюють регістр EFLAGS

Зміну станів розрядів EFLAGS виконують дві інструкції, що розрізняються за способом реагування на випадки, коли один або обидва операнди не є числами (NAN).

COMISD приймач, джерело	Скалярне порівняння молодших частин 128-розрядних операндів зі зміною ознак регістра **EFLAGS**

Команда **comisd** виконує порівняння двох дійсних чисел з можливістю виникнення виняткової ситуації:

comisd xmm, xmm/m64.

UCOMISD приймач, джерело	Скалярне порівняння молодших частин 128-розрядних операндів зі зміною ознак регістра **EFLAGS**

Команда **ucomisd** виконує порівняння двох дійсних чисел з можливістю виникнення виняткової ситуації:

ucomisd xmm, xmm/m64.

Відношення між порівнюваними числами указують ознаки OF, AF, SF, які очищаються (табл. 20.2).

Таблиця 20.2

Відношення	Стан розрядів EFLAGS		
	ZF	PF	CF
Непорівнянні	1	1	1
op1 < op2	0	0	1
op1 > op2	0	0	0
op1 = op2	1	0	0

20.4. Логічні операції

Логічні операції обчислюють функції булевої алгебри. Зазвичай вони застосовуються для виділення або об'єднання окремих частин кодів.

Pentium 4 дозволяє обчислювати логічні функції за допомогою груп інструкцій, що виконують аналогічні дії.

ANDPD приймач, джерело	Паралельна операція логічного *І* над парами бітів упакованих 128-розрядних чисел

Команда **ANDPD** виконує логічну функцію кон'юнкції ("*І*") над парами бітів упакованих 128-розрядних чисел:

andpd xmm, xmm/m128.

ANDNPD приймач, джерело	Паралельна операція інверсії та логічного *І* над парами бітів упакованих 128-розрядних чисел

Команда **andnpd** виконує інверсію операнда приймача, після чого – логічну функцію *І*:

andnpd xmm, xmm/m128.

ORPD приймач, джерело	Паралельна операція логічного *АБО* над парами бітів упакованих 128-розрядних чисел

Команда **ORPD** виконує логічну функцію диз'юнкції ("*АБО*"):

orpd xmm, xmm/m128.

XORPD приймач, джерело	Паралельна операція логічного "*виключне АБО"* над парами бітів упакованих 128-розрядних чисел

Команда **xorpd** виконує логічну функцію "виключне *АБО*":

xorpd xmm, xmm/m128.

20.5. Команди розпаковування і розподілення даних

Розпаковування і перегрупування кодів є своєрідним різновидом пересилань. Позначимо вміст двох половин прм як прмl і прмh, а вміст двох половин

прд як прдl і прдh. Наступні три операції дозволяють записати в прм комбінацію цих величин.

UNPCKHPD приймач, джерело	Операція розпаковування: передавання старших половин чисел

Команда **unpckhpd** виконує операцію розпаковування: прм = прдh, прмh – у прм копіюється вміст старших половин прд та прм:

unpckhpd xmm, xmm/m128.

Схему дії команди unpckhpd наведено на рис.20.6.

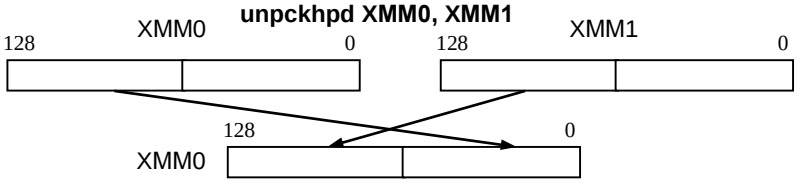


Рис. 20.6. Схема дії команди unpckhpd XMM0, XMM1

UNPCKLPD приймач, джерело	Операція розпаковування: передавання молодших половин чисел

Команда unpcklpd виконує операцію розпаковування: прм = прдl, прмl – у прм копіюється вміст молодших половин прд та прм:

unpcklpd xmm, xmm/m128.

Схему дії команди unpcklpd наведено на рис. 20.7.

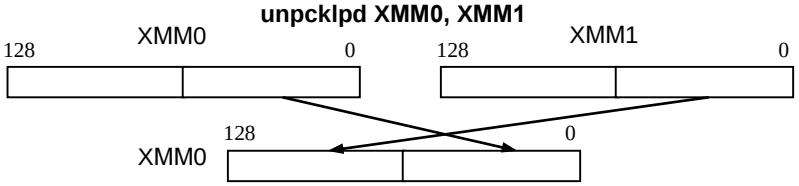


Рис. 20.7. Схема дії команди unpcklpd XMM0, XMM1

SHUFPD приймач, джерело, константа	Керуюче передавання частин операнда

Команда **shufpd** виконує операцію прм = прдh/прдl, прмh/прмl – перегрупування: bit0 imm8 вказує, яка половина прм копіюється в його молодші розряди; bit1 imm8 указує, яка половина прд копіюється в старші розряди прм. У обох випадках 0 відповідає молодшій половині, а 1 – старшій. Наприклад, якщо прм та прд знаходяться в одному регістрі, то imm8 = 1 викличе перестановку чисел в прм:

shufpd xmm, xmm/m128, imm8.

Схему виконання команди shufpd наведено на рис. 20.8.

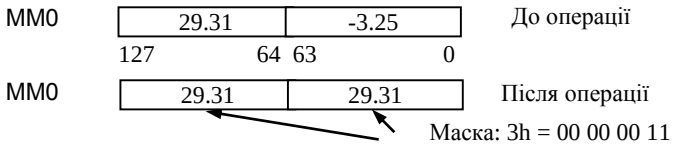


Рис. 20.8. Дія команди shufpd XMM0, XMM0, 3h

20.6. Команди перетворення форматів даних

20.6.1. Групові перетворення форматів даних

Можливі два типи перетворень:

а) дійсні числа перетворяться в дійсні числа зі зміною точності їх подання (з подвійної точності в звичайну або навпаки).

в) цілі числа перетворяться в дійсні числа або дійсні числа – в цілі. У останньому випадку можливе округлення або відсікання остачі. Два цілих числа можуть знаходитися в регістрах xmm або mmx.

CVTPD2PS приймач, джерело	Паралельне перетворення двох чисел з формату з подвійною точністю у формат із звичайною точністю

Команда **cvtpd2ps** виконує паралельне перетворення двох дійсних чисел з формату з подвійною точністю у формат із звичайною точністю:

cvtpd2ps xmm, xmm/m128.

CVTPS2PD приймач, джерело	Паралельне перетворення двох дійсних чисел з формату зі звичайною точністю у формат із подвійною точністю

Команда **cvtps2pd** виконує паралельне перетворення двох дійсних чисел з формату із звичайною точністю у формат з подвійною точністю. Якщо джерелом є xmm, то перетворяться два молодших числа:

cvtps2pd xmm, xmm/m64.

CVTPS2DQ приймач, джерело	Паралельне перетворення чотирьох дійсних чисел із звичайною точністю в чотири цілих 32-розрядних числа

Команда **cvtps2dq** виконує паралельне перетворення чотирьох дійсних чисел із звичайною точністю в 4 цілих 32-х розрядних числа:

cvtps2dq xmm, xmm/m128.

CVTTPS2DQ приймач, джерело	Паралельне перетворення чотирьох дійсних чисел із звичайною точністю в чотири цілих 32-розрядних числа з відсіканням остачі

Команда **cvttps2dq** виконує паралельне перетворення чотирьох дійсних чисел із звичайною точністю в 4 цілих 32-розрядних числа з відсіканням остачі:

cvttps2dq xmm, xmm/m128.

CVTDQ2PS приймач, джерело	Паралельне перетворення чотирьох 32- розрядних чисел в чотири дійсних числа, поданих із звичайною точністю

Команда **cvtdq2ps** виконує перетворення 4 цілих 32-розрядних чисел в 4 дійсних числа, представлених із звичайною точністю:

cvtdq2ps xmm, xmm/m128.

CVTPD2DQ приймач, джерело	Паралельне перетворення двох дійсних чисел подвійної точності в два цілих 32-розрядних числа

Команда **cvtpd2dq** виконує перетворення двох дійсних чисел подвійної точності в два цілих 32-розрядних числа:

cvtpd2dq xmm, xmm/m128.

CVTPD2DQ приймач, джерело	Паралельне перетворення двох дійсних чисел подвійної точності в два цілих 32-розрядних числа з відсіканням остачі

Команда **cvttpd2dq** виконує перетворення двох дійсних числа подвійної точності в два цілих 32-розрядних числа з відсіканням остачі:

cvttpd2dq xmm, xmm/m128.

CVTDQ2PD приймач, джерело	Паралельне перетворення двох цілих 32-розрядних числа в два дійсних числа, поданих з подвійною точністю

Команда **cvtdq2pd** виконує перетворення двох цілих 32-розрядних чисел у два дійсних числа, поданих з подвійною точністю. Якщо джерелом є xmm, то перетворяться два молодших числа:

cvtdq2pd xmm, xmm/m64.

CVTPD2PI приймач, джерело	Паралельне перетворення двох дійсних чисел подвійної точності в два цілих 32-розрядних числа та збереження результату в регістрі mmx

Команда **cvtpd2pi** виконує перетворення двох дійсних чисел подвійної точності в два цілих 32-розрядних числа та збереження результату в регістрі mmx:

cvtpd2pi mmx, xmm/m128.

CVTTPD2PI приймач, джерело	Паралельне перетворення двох дійсних чисел подвійної точності в два цілих 32-розрядних числа з відсіканням остачі і записом в регістр mmx

Команда **cvttpd2pi** виконує перетворення двох дійсних чисел подвійної точності в два цілих 32-розрядних числа з відсіканням остачі і записом в регістр mmx:

cvttpd2pi mmx, xmm/m128.

CVTPI2PD приймач, джерело	Паралельне перетворення двох цілих 32-розрядних чисел, що знаходяться в регістрі mmx, у два дійсних числа, поданих з подвійною точністю

Команда **cvtpi2pd** виконує перетворення двох цілих 32-розрядних чисел, що знаходяться в регістрі mmx, у два дійсних числа, поданих з подвійною точністю:

cvtpi2pd xmmx, mmx/m64.

Приклад 20.1. Виконати паралельний пошук максимального значення в парах упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Додати всі результати та виконати перетворення дійсного числа подвійної точності в ціле 32-розрядне число.

Лістинг 20.6:

```
        title Rysovaniy A.N.
        .686p ; директива определения типа микропроцессора
        .model flat,stdcall   ;завдання лінійної моделі пам'яті та угоди ОС Windows
        .XMM ; директива визначення команд SSE2
        option casemap:none; отличие строчных и прописных букв
         include \masm32\include\windows.inc   ; файлы структур, констант …
         include \masm32\macros\macros.asm
         uselib kernel32, user32
.data                                   ; директива определения данных
        mas1 dq 1.,2.       ; масив 1
        mas2 dq 3.,4.     ; масив 2
        mas3 dq 5.,6.       ; масив 3
        mas4 dq 7.,8.     ; масив 4
titl1 db "SSE2-команды. Параллельный поиск max значений",0 ; назва вікна
fmt db "Параллельный поиск max значений",0Ah,0Dh,\
"в парах упакованных 64-разрядных чисел с плавающей точкой.",0Ah,0Dh,0Dh,
  "Массивы:",0Ah,0Dh,"mas1:   1.,  2.",0Ah,0Dh,"mas2:   3.,  4.",0Ah,0Dh,0Dh,
   "mas3:   5.,  6.",0Ah,0Dh,"mas4:   7.,  8.",0Ah,0Dh,0Dh,\
    "Сумма максимальных значений = %d",0 ; форматування числа
        buf db 2 dup(?)             ; буфер
.code
  _st:                                  ; link /SUBSYSTEM:CONSOLE sse2_3.obj
        movupd XMM0,mas1                ; занесення masl до XMM0
        movupd XMM1,mas2    ; занесення mas2 до XMM1
        movupd XMM2,mas3                ; занесення mas3 до XMM2
        movupd XMM3,mas4                ; занесення mas4 до XMM3
        maxpd XMM0,XMM1     ; знаходження максимумів у mas1 та mas2
        maxpd XMM2,XMM3     ; знаходження максимумів у mas3 та mas4
        addpd XMM0,XMM2             ; сума максимумів
; розпакування ст. частини xmm0 в мол. частину xmm4
    unpckhpd xmm4,xmm0 ; розпак. ст. частини з xmm0 в ст. частину xmm4
    unpckhpd xmm4,xmm5 ; розпак. ст. частини з xmm5 в ст. частину xmm4
; розпакування мол. частини xmm0 в мол. частину xmm5
unpcklpd xmm5,xmm0  ; розпак. мол. частини xmm0 в ст. частину xmm5
unpckhpd xmm5,xmm4 ; розпак. ст. частини xmm4 в мол. частину xmm5
        addpd xmm4,xmm5         ; сума xmm4 і xmm5
        cvtpd2pi MM0,XMM4           ; перетворення у 32-розрядне число
        movd dword ptr ebx,MM0    ; занесення вмісту MM0 до ebx
invoke wsprintf,addr buf,addr fmt,ebx  ; API-функція перетворення числа
```

```
invoke MessageBox,NULL,addr buf,addr titl1,MB_ICONINFORMATION+180000h ;
      invoke ExitProcess,0    ; звільнення ресурсів та передача управління  ОС
      end _st          ; кінець програми з ім'ям _st
```

Сума максимальних чисел в масиві командою addpd XMM0,XMM2 зберігається в XMM0. У зв'язку з тим, що це число є упакованим з подвійною точністю, то для того, щоб його вивести функцією MessageBox, це число необхідно розпакувати. Є команди розпакування старших та молодших частин, які потім необхідно скласти. Для цього спочатку старша частина регістра xmm0 командою unpckhpd xmm4,xmm0 розпаковується та заноситься до старшої частини xmm4. А додавання буде здійснюватися **в молодших частинах** вибраного регістра. Тому старша частина максимуму із регістра xmm4 передається в молодшу частину того ж регістра xmm4 командою unpckhpd xmm4,xmm5. Таким чином, розпакування та підготування старшої частини до додавання буде закінчено.

Розпакування молодшої частини максимуму та передача в молодшу частину регістра виконано також двома командами:

```
unpcklpd xmm5,xmm0    ; розпаковування молодших чисел XMM5 і XMM0
unpckhpd xmm5,xmm4    ; розпаковування старших чисел XMM5 і XMM4
```

Після виконання розпакування всього максимуму виконується додавання двох його частин командою addpd xmm4,xmm5.

Для виведення результату (максимуму) функцією MessageBox виконується перетворення дійсного числа у ціле число командою cvtpd2pi (нажаль, з погрішністю), а потім – і його виведення (рис. 20.9).

Рис. 20.9. Результат виконання програми з лістингу 20.2

20.6.2. Перетворення формату одного числа

Можливі такі два типи перетворень:

а) змінюється точність подання дійсного числа з подвійної на звичайну або навпаки.

в) ціле число перетворюється в дійсне число або дійсне число – в ціле. У останньому випадку можливе округлення або відсікання остачі.

CVTSD2SS приймач, джерело	Скалярне перетворення дійсного числа з формату з подвійною точністю у формат із звичайною точністю

Команда **cvtsd2ss** виконує скалярне перетворення дійсного числа з формату з подвійною точністю у формат із звичайною точністю. У приймачеві не змінюється вміст розрядів 32 – 127:

cvtsd2ss xmm, xmm/m64.

CVTSS2SD приймач, джерело	Скалярне перетворення дійсного числа з формату із звичайною точністю у формат з подвійною точністю

Команда **cvtss2sd** виконує скалярне перетворення дійсного числа з формату із звичайною точністю у формат з подвійною точністю. Вміст старшої половини приймача не змінюється:

cvtss2sd xmm, xmm/m32.

CVTSD2SI приймач, джерело	Скалярне перетворення дійсного числа подвійної точності в ціле 32-розрядне число

Команда **cvtsd2si** виконує скалярне перетворення молодшого упакованого дійсного числа подвійної точності в ціле 32-розрядне число зі знаком:

cvtsd2si r32, xmm/m64.

Результат округляється відповідно до значення бітів поля rc регістра управління /стану.

CVTTSD2SI приймач, джерело	Скалярне перетворення дійсного числа подвійної точності в ціле 32-розрядне число

Команда **cvttsd2si** виконує скалярне перетворення дійсного числа подвійної точності в ціле 32-розрядне число:

cvttsd2si r32, xmm/m64.

Результат формується шляхом відкидання дробової частини, при цьому регістр управління /стану не використовується.

CVTSI2SD приймач, джерело	Скалярне перетворення двох цілих 32-розрядних чисел, що знаходяться в регістрі mmx, в два дійсних числа, поданих з подвійною точністю

Команда cvtsi2sd виконує скалярне перетворення цілого 32-розрядного числа в дійсне число, подане з подвійною точністю. Вміст старшої половини dest не змінюється:

cvtsi2sd xmm, r32/m32.

20.7. Управлінням кешем

Кеш – посередник між процесором і ОЗП. Усі інструкції, крім спеціальних, для читання і запису даних звертаються до кеша, а не до ОЗП. Все, що було записане в кеш, колись потрапить в ОЗП, а все, що було зчитано з ОЗП, колись потрапить в кеш. У Pentium III з'явилася інструкція PREFETCHT, що виконує попереджальну вибірку даних з ОЗП в кеш. Але якщо кеш містить дані, ще не записані в ОЗП, то при попереджальному завантаженні вони будуть втрачені. Тому відразу була введена інструкція **SFENCE**, яка дозволяла захистити записані в кеш дані, ще не збережені в ОЗП. У Pentium IV було додано ще дві інструкції: для захисту та очищення рядка кеша, що містить вже непотрібні дані.

Команда **LFENCE** припиняє прочитування даних з кеша до закінчення раніше запитаної операції завантаження.

Команда **MFENCE** припиняє будь-які звернення до кеша до закінчення всіх операцій обміну між ОЗП і кешем.

Команда **CFLUCH** m8 очищує рядок кеша, номер (адреса) якого в кеш задає операнд, що має розмір байта.

20.8. Операції з цілими числами. Нові можливості інструкцій MMX

Інструкції групи MMX вперше з'явилися у Pentium MMX, а потім у Pentium II. Усього в цю групу входило 57 інструкцій, яким відповідали 47 мнемонічних позначень (імен). Більшість з них виконувала групові операції з цілими числами, що мають розмір байта, слова або подвійного слова, а деякі могли працювати з 64-розрядними даними (QWORD). У Pentium III з'явилася група інструкцій SSE1, 14 з яких виконували групові операції з цілими числами, тобто доповнювали набір інструкцій MMX. Але як старі, так і нові інструкції могли працювати тільки з 64-бітовими даними, що знаходяться в регістрах mmx або в ОЗП.

З випуском Pentium IV більшість указаних інструкцій отримали доступ до 128-бітових регістрів xmm. Виключень небагато: у групі MMX – інструкція EMMS, а в групі SSE1 – три інструкції: MASKMOVQ, MOVNTQ і PSHUFW. Решту всіх інструкцій виконують групові операції як з 64-бітовими, так і з 128-бітовими даними. Імена інструкцій не змінилися, а розмір групи визначається відповідно до імені регістра (mmx або xmm). Вище були наведені варіанти запису логічних операцій групи MMX при роботі з 128-бітовими даними.

Таким чином, розширений набір інструкцій MMX дозволяє виконувати операції над групами, що складаються з 16-ти байтів, 8-ми слів, 4-х подвійних (32-біти) слів або 2-х квадрослів (64-біти). Логічні операції і зрушення можуть виконуватися і над одним 128-бітовим операндом.

Інструкції групи команд SSE2, призначених для роботи з цілими числами, які з'явилися у Pentium IV, наведено у розділі про MMX-команди.

20.9. Асемблювання через безпосередній запис до каталогу Visual Studio 2013 та компонування через masm32

Перший й самий простий шлях – переписати файл з розширенням .asm у встановлену програму Microsoft Visual Studio 2013 за такою адресою: c:\Program Files (86)\ Microsoft Visual Studio 12.0\VC\bin\. У цій директорії знаходиться програма ml.exe, яка виконує асемблювання з параметрами ml /c /coff ім'я_прог.asm. При виконанні цієї дії програма ml.exe звертається до **своїх** Visual Studio-бібліотек і тому не можна просто змінити в masm32 файл ml.exe. Треба виконувати асемблювання тільки в Microsoft Visual Studio 2013, так, як, наприклад, наведено на рис. 20.10.

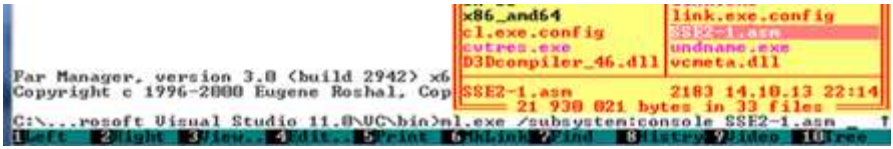


Рис. 20.10. Отримання файлу .obj в Visual Studio 2013 через файловий менеджер FAR

У такому випадку отримується необхідний .obj-файл, який після натискання на кнопку Enter буде знаходитися в цій же директорії.

Отриманий .obj-файл необхідно переписати у програму masm32 та вже там отримати .exe-файл, для чого необхідно вказати параметри link /SUBSYSTEM:CONSOLE ім'я_прог.obj, так, як, наприклад, показано на рис. 20.11.

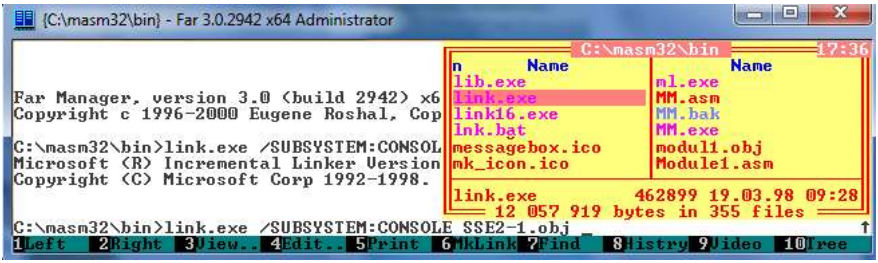


Рис. 20.11. Отримання файлу .exe через файловий менеджер FAR

20.10. Асемблювання та компонування через безпосереднє звертання до Visual Studio 2010

Послідовність дій може складатися з декількох етапів.

Для отримання виконуваного файлу початковий asm-файл зручніше розмістити за адресою C:\Program Files\ Microsoft Visual Studio 10.0\VC.

Етап 1. Настроювання асемблера ml.exe.

Відкрити програму Visual Studio 2010 та прописати шлях находження ml.exe та його параметрів.

Вибрати в меню програми Tools\Externel Tools та натиснути Add. У рядок Title написати якесь ім'я, наприклад masm32_SSE2+.

У рядку Command натиснути на клавішу … та вказати шлях находження ml.exe: C:\Program Files\Microsoft Visual Studio 10.0\VC\bin\ml.exe.

У рядок Arguments вставити параметри асемблювання: **-c -coff -Fl -Fo $(ItemFileName).obj $(ItemFileName).asm.** Обов'язково перевірити, щоб перед першим символом параметра не було пропуску.

У рядок Initial directory прописати параметр $(ItemDir)

Зняти прапорець Close on exit.

Натиснути на кнопку Apply та кнопку OK (рис. 20.12).

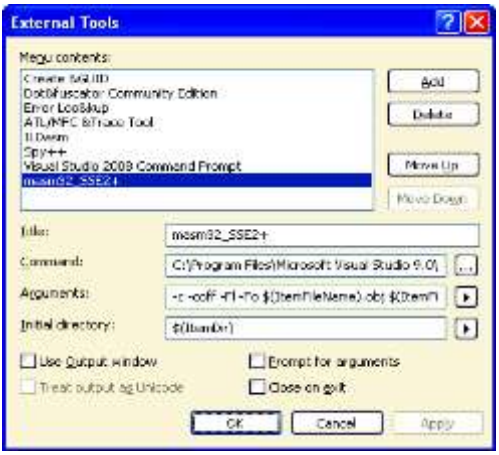


Рис. 20.12. Вигляд вікна прописування шляхів та параметрів асемблювання Visual Studio 2010 для команд SSE2+

Етап 2. Настроювання лінкера link.exe.

Для цього необхідно прописати шлях находження файлу link.exe та його параметри. У зв'язку зі складністю параметрів для лінкера link.exe з програми Visual Studio 2010 простіше безпосередньо їх вводити в командний рядок (процес введення параметрів в командний рядок розглянуто в 3-му етапі).

Етап 3. Виконання програми.

Для цього необхідно завантажити файл в Visual Studio 2008: File\ Open\File…

Виконати **асемблювання**: Tools\ masm32_SSE2+. У результаті таких дій *за змістом знаходження* asm-файлу створиться .obj-файл, який розташується, наприклад, за адресою C:\Program Files\Microsoft Visual Studio 10.0\VC\sse2_3.obj або C:\mas32\bin\sse2_3.obj.

Після виправлення помилок необхідно обов'язково знову зберігати файл : File\Save ... (рис. 20.13).

Для виконання **лінкування** необхідно викликати командний рядок Visual Studio 2010, який находиться через послідовне натискання кнопок Пуск\ Все программы\ Microsoft Visual Studio 2010\, Visual Studio Tools\ Visual Studio Command Prompt (2010). Після того, як з'явиться вікно для введення командного рядка необхідно спочатку скопіювати команду для лінкування, наприклад: **link /SUBSYSTEM:CONSOLE sse2_3.obj** (з адресою знаходження .obj-файлу**),** а потім натиснути праву кнопку миші та вставити (рис. 20.14).

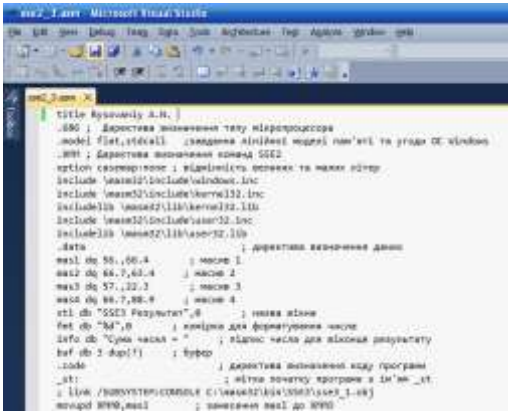


Рис. 20.13. Зберігання файлу в середовищі Visual Studio 2010

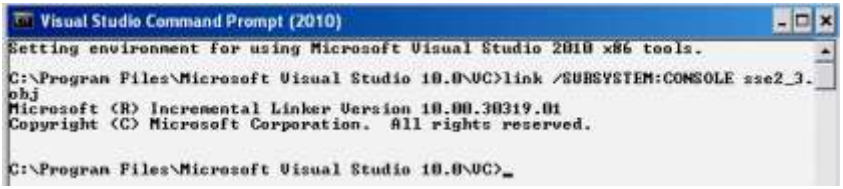


Рис. 20.14. Вигляд командного рядка Visual Studio 2010 Command Prompt

Натиснути клавішу Enter. Буде створено .exe-файл, який знаходитиметься за такою адресою: C:\Program Files**Microsoft Visual Studio 10.0\VC** sse2_3.exe.

20.11. Лабораторна робота “SSE2-команди”

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм з використанням команд SSE, написаних мовою асемблера для програмування МП платформи x86.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати свій варіант задачі та написати на асемблері за допомогою SSE2-команд програму обчислення одного з виразів з використанням дійсних чисел, ***виведенням на екран***

варіанту завдання, його вмісту, результату, прізвища та e-mail автора програми.

Методичні рекомендації

Асемблер masm32v11 не підтримує команди SSE2 та вищі. В такій ситуації можна діяти шляхами: через безпосередній запис до каталогу програми Visual Studio 2013 або через безпосереднє звертання до середовища Visual Studio, або додавати перед кожною командою константи db 66h, на величину яку відрізняються SSE2 від SSE.

Завдання 1:

1. Виконати паралельне порівняння за допомогою SSE2-команд 2-х масивів по 11 чисел. Якщо всі числа другого масиву більші від першого, то додати всі числа другого масиву, а якщо навпаки – то мінімального.

2. Задано два масиви по 5 чисел. За допомогою SSE2-команд визначити суму чисел всіх масивів та виконати аналіз чисел 0 та 1 результату. Якщо вони нульові, то виконати операцію добуття квадратного кореня з нього.

3. Виконати паралельне порівняння за допомогою SSE2-команд 2-х масивів по 10 чисел. Якщо перший масив більший від другого, то виконати операцію $(a - 3e)b - d/\sqrt{b}$, де $a = 0{,}1$; $b = 1{,}05$; $c = 2{,}1$; $d = 3{,}2$;
інакше – виконати операцію d/b.

4. Виконати паралельне порівняння за допомогою SSE2-команд 2-х масивів по 9 чисел. Якщо один масив менший від другого, то виконати операцію
$(4a - \sqrt{e})b - d/b$, де $a = 0{,}1$; $b = 1{,}05$; $c = 2{,}1$; $d = 3{,}2$;
інакше – виконати операцію d/b.

5. Виконати за допомогою SSE2-команд паралельне додавання за модулем 2-х масивів по 13 чисел. Якщо сума всіх отриманих чисел більше 255, то виконати операцію $(a - 5c)/b + \sqrt{c}$, де $a = 0{,}2$; $b = 10{,}05$; $c = 2{,}3$;
інакше – виконати операцію $a - c$.

6. Виконати операцію паралельного логічного перемноження 2-х масивів по 10 чисел. Додати всі результати. Якщо сума всіх отриманих чисел більше 100, то виконати обчислення квадратного кореня, а якщо навпаки – не виконувати.

7. Виконати паралельне множення двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Додати всі результати та обчислити квадратний корінь.

8. Виконати паралельне ділення двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Додати всі результати та обчислити квадратний корінь.

9. Виконати паралельний пошук мінімального значення в парах упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Додати всі результати та обчислити квадратний корінь.

10. Виконати паралельне добування квадратного кореня з двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Додати всі результати та обчислити квадратний корінь.

Завдання 2:

1. Виконати паралельне порівняння за допомогою SSE2-команд 2-х масивів. Якщо всі числа другого масиву більші від першого, то виконати пошук максимального значення в парах упакованих чисел, а якщо навпаки – то мінімального. Виконати розпакування чисел та результати результати операції вивести через MessageBox.

2. Виконати паралельне порівняння за допомогою SSE2-команд 2-х масивів. Якщо всі числа другого масиву більші від першого, то додати всі числа другого масиву, а якщо навпаки – то перемножити. Виконати розпакування чисел та результати операції вивести через MessageBox.

3. Виконати паралельне порівняння за допомогою SSE2-команд 2-х масивів. Результати операції вивести через MessageBox. Якщо перший масив більший від другого, то виконати паралельне додавання елементів масиву командою **ADDPD**, інакше – виконати операцію **MASKMOVDQU**. Виконати розпакування чисел та результати операції вивести через MessageBox.

4. Виконати паралельне порівняння за допомогою SSE2-команд 2-х масивів. Якщо один масив менший від другого, то виконати команду **DIVPD** над масивами чисел, інакше – виконати команду **MULPD** над масивами чисел. Виконати розпакування чисел та результати операції вивести через MessageBox.

5. Виконати за допомогою SSE2-команд паралельне додавання за модулем 2-х масивів чисел. Якщо сума всіх отриманих чисел більше 255, то виконати команду **SQRTPD** над елементами результату додавання, інакше – виконати команду **MAXPD** над елементами результату додавання. Виконати розпакування чисел та результати операції вивести через MessageBox.

6. Виконати операцію паралельного логічного перемноження 2-х масивів чисел. Результати операції логічного перемноження вивести через MessageBox. Додати всі результати. Якщо сума всіх отриманих чисел більше 100, то виконати обчислення квадратного кореня, а якщо навпаки – не виконувати. Виконати розпакування чисел та результати операції вивести через MessageBox.

7. Виконати паралельне множення двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Результати операції вивести через MessageBox. Потім додати всі результати та обчислити квадратний корінь. Виконати розпакування чисел та результати операції вивести через MessageBox.

8. Виконати паралельне ділення упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Результати операції вивести через MessageBox. Потім додати всі результати та обчислити квадратний корінь. Виконати розпакування чисел та результати операції вивести через MessageBox.

9. Виконати паралельний пошук мінімального значення в парах упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Результати операції вивести через MessageBox. Додати всі результати та обчислити квадратний корінь. Виконати розпакування чисел та результати операції вивести через MessageBox.

10. Виконати паралельне добування квадратного кореня упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Результати операції вивести через MessageBox. Потім виконати команди **MAXPD** та **MINPD**. Виконати розпакування чисел та результати операції вивести через MessageBox.

Зміст звіту

1. Постановка задачі для конкретного варіанта.

2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.

3. Лістинг програми з виведенням даних на екран монітора з використанням API-функцій та з детальним коментарем і описом роботи.

4. Print screen екрана 32-розрядного налагоджувача з виконаною програмою.

5. Короткий опис виконання програми.

6. Висновки за результатами роботи.

Приклад 20.2. Виконати паралельне порівняння за допомогою SSE2-команд 2-х масивів по 9 чисел. Якщо один масив менший від другого, то виконати операцію

$$(a - e)b - d/b, \text{ де } a = 0{,}1;\ b = 1{,}05;\ c = 2{,}1;\ d = 3{,}2;$$

інакше – виконати операцію d/b.

Лістинг 20.7:

```
    .686p     ; директива определения типа микропроцессора
    .model flat,stdcall  ; задание линейной модели памяти
    .XMM                        ; використання регістрів SSE
    option casemap:none                 ; отличие строчных и прописных букв
    include \masm32\include\windows.inc   ; файлы структур, констант …
    include \masm32\macros\macros.asm
    uselib kernel32, user32, fpu
    BSIZE equ 50
.data
 mas1 dd  11.4, 14.02, 9.8, 4.0, 12.14, 1.82, 67.54, 9.95, 7.16    ; масив №1
       len EQU $- mas1
mas2 dd 11.4, 14.02, 3.14, 2.718, 5.0, 45.45, 23.35, 9.95, 79.2 ; масив №2
       rez dd len dup(0)          ; результат порівняння
       rez2 dq ?                  ; для результату виконання операцій
       a1 dd 0.10
       b1 dd 1.05
       c1 dd 2.10
       d1 dd 3.20
```

```
        buf BYTE BSIZE dup(?)
        st1 db "The Rezult mas1!=mas2",0     ; заголовки вікна повідомлень
        st2 db "The Rezult mas1==mas2",0
        st3 dd 0
      flag db 0
.code
_st:
            xor edx,edx
            mov eax,len           ; завантаження 40 байтів масиву
            shr eax,2 ; len/4 = 10 - визначення кількості чисел
            mov ebx,4
            div ebx               ; визначення кількості циклів
            mov ecx,eax
            lea esi,mas1          ; завантаження масиву №1 в esi
            lea edi,mas2          ; завантаження масиву №2 в edi
            xor ebx,ebx
     .WHILE(ecx!=0)
            movupd XMM0,[esi]     ; завантаження масиву №1 в регістр XMM0
            movupd XMM1,[edi]     ; завантаження масиву №1 в регістр XMM0
       cmppd XMM0, XMM1, 0 ; виконання паралельного порівняння масивів
            movupd  rez[ebx],XMM0 ; збереження результату порівняння
            add esi,16              ; перехід на наступну адресу
            add edi,16
            add ebx,16
            dec ecx
     .ENDW
mov ecx,edx ; залишок чисел, що не ввійшли до паралельного порівняння
.WHILE(ecx!=0)
            movupd XMM0,[esi]     ; завантаження масиву №1 в регістр XMM0
          movupd XMM1,[edi]       ; завантаження масиву №1 в регістр XMM0
          cmpsd XMM0, XMM1,8  ; виконання скалярного порівняння масивів
            movupd  rez[ebx],XMM0   ; збереження результату порівняння
            add esi,4               ; перехід на наступну адресу
            add edi,4
            add ebx,4
            dec ecx
.ENDW
     mov ecx,9
     xor ebx,ebx
     .WHILE(ecx!=0)
.IF (rez[ebx]==0)   ; якщо елементи масивів хоча б один раз не дорівнюють
            mov al,1       ; одне одному, то встановлюємо прапорець в 1
            mov flag,al
.ENDIF
            dec ecx
            inc ebx
.ENDW
     .IF (flag==1)          ; якщо масиви не рівні, виконуємо вираз (a-c)b-d/b
            finit
            fld a1
            fsub c1
            fmul b1
            fld d1
            fdiv b1
```

```
            fsub
            fst rez2
            lea ebx,st1
    .ELSE                       ; якщо ж масиви рівні, то виконуємо d/b
          finit
            fld d1
            fdiv b1
            fst rez2
            lea ebx,st2
    .ENDIF
            mov st3, ebx                ; збереження заголовку вікна
invoke FpuFLtoA,0,10,addr buf,SRC1_FPU or SRC2_DIMM ; перетворення
invoke MessageBox, NULL, ADDR buf,  st3, MB_OK          ; виведення на екран
      invoke ExitProcess,0
      end _st
```

Результат виконання програми з лістингу 20.7 наведено на рис. 20.15.

Рис. 20.15. Результат виконання програми з лістингу 20.7

Приклад 20.3. Виконати паралельне множення двох упакованих 64-розрядних чисел з плаваючою точкою подвійної точності. Додати всі результати та обчислити квадратний корінь.

Для того, щоб середовище masm32 виконувала команди SSE2 можна додати константи db 66h, на яку відрізняються SSE2 від SSE.

Лістинг 20.8:

```
      .686p               ; директива определения типа микропроцессора
      .model flat,stdcall  ; задание линейной модели памяти
      .XMM                              ; використання регістрів
      option casemap:none    ; отличие строчных и прописных букв
      include \masm32\include\windows.inc   ;
      include \masm32\macros\macros.asm
      uselib kernel32, user32, fpu
.data
    mas1  DQ 5.,625. ;
    mas2  DQ 2.,3.  ;
    res   DQ 2 DUP(0),0  ;
    titl1 db "Произведение 1",0
    s1 DB " 5 x 2 ="
    buf1 db "      ",0
    titl2 db "Произведение 2",0
    s2 DB " 625 x 3 ="
    buf2 db "      ",0
  db 0,0 ; для безпомилкового виведення MessageBox
    titl3 db "Res",0
    s3 DB "Сложение и корень ="
    buf3 db "      ",0
```

```
.code
_start:
        db 66h
        movups  XMM0, mas1  ; пересилання 2 64-розрядних чисел масиву mas1
        db 66h
        movups  XMM1, mas2  ;
        db 66h
        mulps   XMM0, XMM1   ; mas1*mas2
        db 66h
        movups res,XMM0
        fld   QWORD PTR [res]
        invoke  FpuFLtoA,0,2,offset buf1,SRC1_FPU or SRC2_DIMM
        fld   QWORD PTR [res+8]
        invoke  FpuFLtoA,0,2,offset buf2,SRC1_FPU or SRC2_DIMM
        db 66h
        movups  XMM1, XMM0  ;  XMM1 = XMM0 = mas1*mas2
        db 66h
        shufps  XMM1,XMM1,01b
        db 66h
        addps XMM0,XMM1
        db 66h
        sqrtps XMM0,XMM0
        db 66h
        movups  res, XMM0
        fld   QWORD PTR [res] ; загружаем float в fpu
invoke FpuFLtoA,0,2,addr buf3,SRC1_FPU or SRC2_DIMM
invoke MessageBox, 0, ADDR s1, ADDR titl1, MB_ICONINFORMATION
invoke MessageBox, 0, ADDR s2, ADDR titl2, MB_ICONINFORMATION
invoke MessageBox, 0, ADDR s3, ADDR titl3, MB_OK
        invoke ExitProcess,NULL
        end _start
```

Результат виконання програми наведено на рис. 20.16.

Рис. 20.16. Результат виконання програми

Для того, щоб зовнішній вигляд повідомлень був не помилковим, необхідно додатково поставити признак закінчення рядка – тобто нуля (db 0,0).

Приклад 20.4. Числа $a \in \{a1,a2,a3\}$ задані масивом та мають розмірність Real8. Провести розрахунки рівняння $d/b\ sqrt(a) + a$ за допомогою команд SSE2.

```
        .686
        .XMM
        .model flat, stdcall
        option casemap :none
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
```

```
        uselib kernel32,user32,fpu,masm32
        .data
        buf db 80 dup(?)
        buf2 db 20 dup(?)
        term db 0ah
        arr1 real8 16.,25.,36.,1244.
        len1 equ ($-arr1)/type arr1
        arr2 real8 2.,4.,16.       ; b, c, d
        res dq ?
.code     ;ml.exe /c /coff     ;link.exe /subsystem:console
main:    mov ecx,len1
        mov ebx,0
        movsd xmm1,arr2[0]   ; xmm1 - b
        movsd xmm2,arr2[8]   ; xmm2 - c
        movsd xmm3,arr2[16]   ; xmm3 - d
        divsd xmm3,xmm1      ; d/b
        cycle:
        movsd xmm0,arr1[ebx] ; xmm0 - a
        sqrtsd xmm4,xmm0     ;sqrt(a)
        mulsd xmm4,xmm3      ; d/b x sqrt(a)
        addsd xmm4,xmm0      ; d/b x sqrt(a) + a
        cvttsd2si eax,xmm4       ; для отладки
        movsd res,xmm4
        fld res
        add ebx,8
        push ebx
        push ecx
        invoke FpuFLtoA,0,5,addr buf2,SRC1_FPU or SRC2_DIMM
        invoke szCatStr,addr buf,addr buf2
        invoke szCatStr,addr buf,addr term
        pop ecx
        pop ebx
        dec ecx
        jnz cycle
invoke MessageBox,0,addr buf,NULL,MB_ICONINFORMATION
invoke ExitProcess, 0
        end main
```

Результат виконання програми наведено на рис. 20.17.

Рис. 20.17. Результат виконання програми

Контрольні запитання

1. Яке призначення команд SSE2-розширення?
2. Який формат даних SSE2-розширень?
3. Які суфікси мають команди SSE2-розширень та що вони означають?
4. На які групи поділяються команди SSE2-розширень?

21. SSE3-РОЗШИРЕННЯ

З випуском 90-нм процесора Pentium IV під кодовою назвою Prescott (http://www.ixbt.com/cpu/intel-prescott.html), анонсованого Intel 2 лютого 2004 року, архітектура Intel Netburst придбала ряд нових інструкцій, що отримали офіційну назву – SSE3. На першому етапі це був набір з 13 нових інструкцій, які потрібні для підвищення продуктивності процесора у ряді операцій потокової обробки даних. Доповнення торкнулися набору команд SSE (Streaming SIMD Extensions), що працюють з чотирикомпонентними векторами з одинарною точністю, та набору SSE2 (Streaming SIMD Extensions 2), що працюють з двовимірними векторами чисел подвійної точності, а також з співпроцесором x87 FPU.

21.1. Перетворення чисел з плаваючою точкою (x87) в цілі числа

У цю підгрупу входить одна інструкція зі всього набору Prescott New Instructions (SSE3), яка працює на рівні x87 FPU.

Команда fisttp зберігає цілочислове значення із звільненням елемента стека x87 з округленням убік нуля. Її поведінка аналогічна поведінці стандартної інструкції fistp, але її відмінністю є використання округлення убік нуля незалежно від того, який спосіб округлення вибраний в даний момент у контрольному слові FPU.

21.2. Дублювання даних

Ця підгрупа складається з трьох інструкцій.

Команда movsldup виконує завантаження 128-бітового значення з початкового операнда (пам'яті або XMM-регістра) в операнд призначення з дублюванням першого і третього 32-бітових елементів:

*Операнд A (*128 *бітів,* 4 *елементи): a*3 | *a*2 | *a*1 | *a0*
*Операнд B (*128 *бітів,* 4 *елементи): b*3 | *b*2 | *b*1 | *b*0
movsldup *A, B*
*Результат (операнд A): b*2 | *b*2 | *b*0 | *b*0

Команда movshdup виконує завантаження 128-бітового значення з початкового операнда в операнд призначення з дублюванням другого і четвертого 32-бітових елементів:

*Операнд A (*128 *бітів,* 4 *елементи): a*3 | *a*2 | *a*1 | *a*0
*Операнд B (*128 *бітів,* 4 *елементи): b*3 | *b*2 | *b*1 | *b*0
movshdup *A, B*
*Результат (операнд A): b*3 | *b*3 | *b*1 | *b*1

Команда movddup виконує завантаження 64-бітового значення з пам'яті або початкового регістра (біти [63 – 0]) з дублюванням:

*Операнд A (*128 *бітів,* 2 *елементи): a*1 | *a*0

*Операнд B (*64 *бітів,* 1 *елемент): b*0

movddup *A, B*

*Результат (операнд A): b*0 | *b*0

Лістинг 21.1. Дублювання даних:

```
        title Rysovaniy A.N. & Legeza A., НТУ "ХПІ", КІТ-17а
        .686p  ; директива определения типа микропроцессора
.model flat,stdcall   ; задание линейной модели памяти
.XMM ; директива визначення команд SSE
option casemap:none; отличие строчных и прописных букв
 include \masm32\include\kernel32.inc ; системные функции применений…
  includelib \masm32\lib\kernel32.lib
.data                                       ; директива определения данных
        mas1 dd 11.1,22.2,33.3,44.4         ; масив 1
        mas2 dd 55.5,66.6,77.7,88.8         ; масив 2
        mas3 dq 91.1,92.2                   ; масив 3
        mas4 dq 99.9                        ; масив 4
.code                                   ; директива визначення коду програми
_st:                                    ; метка начала программы с именем_st
  ; link /SUBSYSTEM:CONSOLE C:\masm32\bin\SSE3\sse3_1.obj
        lea edi,mas1                    ; занесення адреси masl до edi
        lea esi,mas2                    ; занесення адреси mas2 до esi
        movupd XMM0,[edi]               ; XMM0 := 44.4, 33.3, 22.2, 11.1
        movupd XMM1,[esi]               ; XMM1 := 88.8, 77.7, 66.6, 55.5
        movsldup XMM0,XMM1              ; XMM0 := 77.7, 77.7, 55.5, 55.5
        movupd XMM1,[edi]               ; XMM1 := 44.4, 33.3, 22.2, 11.1
        movupd XMM2,[esi]               ; XMM2 := 88.8, 77.7, 66.6, 55.5
        movshdup XMM1 ,XMM2             ; XMM1 := 88.8, 88.87, 66.6, 66.6
        lea edi,mas3                    ; занесення адреси mas3 до edi
        movupd XMM2,gword ptr [edi]     ; занесення mas3 до XMM2
        movddup XMM2, gword ptr mas4 ; дублювання елемента dq у XMM2
        invoke ExitProcess,0  ; звільнення ресурсів та передача управління  ОС
        end _st         ; кінець програми з ім'ям _st
```

21.3. Завантаження невирівняних змінних

Дану підгрупу представляє команда lddqu. Це операція завантаження невирівняного 128-бітового значення з пам'яті, що виключає можливість «розриву» (перетину кордону) рядка кеша.

У випадку, якщо адреса завантажуваного елемента вирівняна по 16-байтній межі, команда lddqu здійснює звичайне завантаження 16-байтного значення (тобто, по суті, поводиться аналогічно інструкціям movaps/movapd/movdqa із стандартного набору SSE/SSE2). Інакше lddqu завантажує цілих 32 байти, починаючи з вирівняної адреси з подальшим витяганням потрібних 16 байтів. Використання цієї інструкції дозволяє досягти значного збільшення продуктивності при завантаженні невирівняних 128-бітових значень з пам'яті, в порівнянні із стандартними інструкціями movups/movupd/movdqu SIMD-розширень SSE/SSE2.

21.4. Одночасне додавання/віднімання

У цю підгрупу входять дві команди, перша з яких працює з чотирикомпонентними векторами чисел одинарної точності (SSE), а друга – з двокомпонентними векторами чисел подвійної точності (SSE2).

Команда addsubps виконує додавання другого і четвертого елементів з одинарною точністю з одночасним відніманням першого і третього елементів. Ця інструкція корисна при роботі з комплексними числами у разі використання відповідного типу змінних.

*Операнд A (*128 *бітів,* 4 *елементи): a*3 | *a*2 | *a*1 | *a*0
*Операнд B (*128 *бітів,* 4 *елементи): b*3 | *b*2 | b1 | *b*0

addsubps *A, B*
*Результат (операнд A): a*3+*b*3 | *a*2-*b*2 | *a*1+*b*1 | *a*0-*b*0

Команда addsubpd поводиться аналогічно команді addsubps, але працює з числами подвійної точності (двоелементними операндами SSE2):

Операнд A (128 бітів, 2 *елементи): a*1 | *a0*
*Операнд B (*128 *бітів,* 2 *елементи): b*1 | *b*0

addsubpd *A, B*
*Результат (операнд A): a*1+*b*1 | *a*0-*b*0

21.5. Горизонтальне додавання/віднімання

П'ята підгрупа представлена чотирма командами, що здійснюють принципово нові операції для SIMD-розширень сімейства SSE/SSE2. Перші дві з них працюють з чотирикомпонентними векторами з одинарною точністю, а останні – з двокомпонентними векторами з подвійною точністю.

Команда haddps здійснює горизонтальне складання елементів з одинарною точністю. Перший елемент, записуваний в операнд призначення, є сумою першого і другого елементів першого (початкового) операнда; другий елемент – сумою третього і четвертого елементів першого операнда; третій елемент – сумою першого і другого елементів другого операнда (операнда призначення) і, нарешті, четвертий елемент – сумою третього і четвертого елементів другого операнда. Для наочності зобразимо це, як і раніше, у вигляді схеми:

*Операнд A (*128 *бітів,* 4 *елементи): a*3 | *a*2 | *a*1 | *a*0
*Операнд B (*128 *бітів,* 4 *елементи): b*3 | *b*2 | *b*1 | *b*0
haddps *A, B* ; горизонтальне додавання елементів з одинарною точністю
*Результат (операнд A): b*2+*b*3 | *b0*+*b*1 | *a*2+*a*3 | *a*0+*a*1

Команда hsubps здійснює горизонтальне віднімання елементів з одинарною точністю. Її поведінка аналогічна команді haddps, єдиною її відмінністю є використання операції віднімання замість складання:

*Операнд A (*128 *бітів,* 4 *елементи): a3* | *a2* | *a1* | *a0*

*Операнд B (*128 *бітів, 4 елементи): b3 | b2 | b1 | b0*

hsubps *A, B ;* горизонтальне віднімання елементів з одинарною точністю
Результат (операнд A): b2-b3 | b0-b1 | a2-a3 | a0-a1

Фрагмент програми дослідження команд горизонтального додавання та віднімання може бути таким:

```
        mas1 REAL4 1.0, 2.3, 3.4, 4.3, 5.6, 6.7, 7.8, 8.9, 5.6          ; масив 1
        mas2 REAL4 2.0, 2.4, 3.5, 4.4, 5.5, 6.6, 7.7, 8.8, 5.5          ; масив 2
  .code                                          ; директива начала кода программы
  _st:                                                 ; метка начала программы
      mov ecx,LENGTHOF mas2/4 ; обчислення цілого значення, кратного 4
         lea esi,arr1                    ; завантаження адреси масиву 1
         lea edi,arr2                    ; завантаження адреси масиву 2
@@:    movups XMM0,[esi]    ; завантаження 4 ×32 чисел масиву 1
         haddps XMM0,[edi]    ; додавання 4 ×32 чисел масивів
         movups XMM1,[esi]    ; завантаження 4 ×32 чисел масиву 1
         hsubps XMM1,[edi]    ; віднімання 4 ×32 чисел масивів
       add esi,4*TYPE mas1 ; збільшення адреси на чотири числа масиву 1
       add edi,4*TYPE mas2 ; збільшення адреси на чотири числа масиву 2
         loop @b      ; перехід на мітку @@ для зчитування наступних чисел
```

У програмі обробляється два цикли по чотири 32-розрядних числа. Для простоти числа, які залишаються, – не обробляються. Результат виконання програми наведено на рис. 21.1.

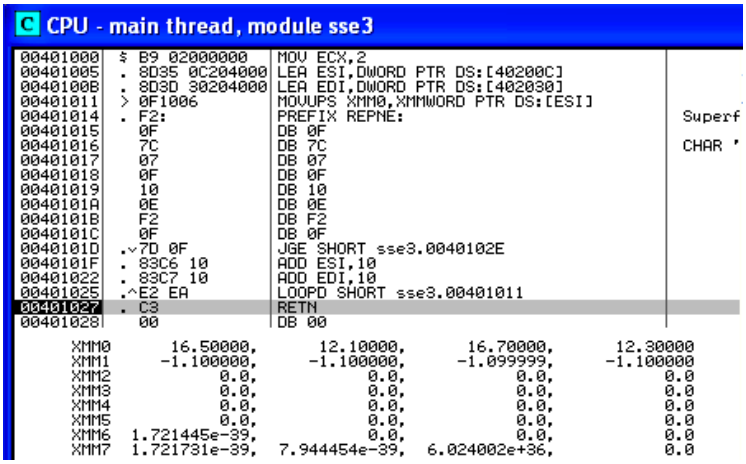


Рис. 21.1. Результат виконання програми

Незважаючи на те, що налагоджувач OllyDbg 1.10 розуміє не всі сучасні команди, тому відображає їх як дані.

Команда haddpd здійснює горизонтальне складання елементів з подвійною точністю. Першим (нижчим) результуючим елементом є сума нижньої і

верхньої частин першого (початкового) операнда, другим (верхнім) – сума нижньої і верхньої половин другого операнда (операнда призначення):

Операнд A (128 бітів, 2 *елементи): a*1 | *a*0

Операнд B (128 бітів, 2 *елементи): b*1 | *b*0

haddpd *A, B* ; горизонтальне складання елементів з подвійною точністю

*Результат (операнд A): b*0+*b*1 | *a*0+*a*1

Команда hsubpd здійснює горизонтальне віднімання елементів з подвійною точністю. Ця інструкція аналогічна команді haddps, але використовує операцію віднімання замість складання:

*Операнд A (*128 *бітів,* 2 *елементи): a*1 | *a*0

*Операнд B (*128 *бітів,* 2 *елементи): b*1 | *b*0

hsubpd *A, B* ; горизонтальне віднімання елементів з подвійною точністю

*Результат (операнд A): b*0-*b*1 | *a*0-*a*1

21.6. Синхронізація потоків

У останню підгрупу можна включити дві інструкції, націлені на використання в системному програмуванні з метою надання можливості ефективнішої синхронізації потоків. Ці інструкції використовуються при розробці операційних систем і драйверів пристроїв з метою поліпшення продуктивності процесора і зниження енергоспоживання останнього, коли він знаходиться в режимі «порожнього» очікування (разом з введеною в розширення SSE2 інструкцією pause).

Команда monitor встановлює діапазон адрес пам'яті (зазвичай використовується один рядок кеша), за яким здійснюється відстежування записів за стандартним протоколом write-back.

Команда mwait вводить логічний процесор в режим низького енергоспоживання при очікуванні записів по простору адрес, заданих інструкцією monitor. З архітектурної точки зору її поведінка ідентична команді nop. Вихід з оптимізованого стану здійснюється у разі запису по встановленому простору адрес, а також при спрацьовуванні будь-якого переривання або виключення.

21.7. Використання SSE3 в розробці і оптимізації програм

21.7.1. Розрахункові завдання, що використовують x87 FPU

У завданнях такого типу (якими є переважна більшість професійного програмного забезпечення) може виявитися вельми корисною інструкція швидкого перетворення дійсних чисел в цілих (fisttp), єдина зі всього набору SSE3, що працює на рівні x87 FPU.

Відомо, що правильним, з погляду стандарту C/C++/fortran, способом перетворення змінних типу *float* (чисел з плаваючою точкою) в змінні типу int (цілі числа) в операціях вигляду

float f;
int i = (int)f

є округлення убік нуля (truncate). У той же час вибраним за умовчанням способом округлення x87 FPU є округлення убік найближчого точного числа (round-to-nearest). Необхідність існування операції округлення пов'язана зовсім не з перетворенням дійсних чисел в цілі, а з кінцевим способом подання нескінченної безлічі чисел з плаваючою точкою (32 біти – для чисел одинарної точності, 64 біти – для чисел подвійної точності, 80 бітів – для змінних з розширеною точністю, а також для внутрішнього подання даних в x87 FPU). При цьому найбільш точне кінцеве подання чисел з плаваючою точкою досягається при такому способі округлення.

У зв'язку з цим наведений вище C-код в процесорах IA-32 буде перетворений будь-яким компілятором, відповідно до вимог стандарту ANSI C, в асемблерний код приблизно такого вигляду:

```
DWORD cwold, cwnew;
__asm
{
fld        dword ptr[f]          // завантаження значення float
 fnstcw   word ptr[cwold]     // збереження FPUCW
 movzx   eax, word ptr[cwold]
 or     eax, 0x0c00          // встановлення режиму округлення
                             // убік нуля (truncate)
mov    dword ptr[cwnew], eax
fldcw  word ptr[cwnew]  // завантаження нового значення FPUCW
      fistp    dword ptr[i]          // збереження значення int
      fldcw   word ptr[cwold]    // відновлення FPUCW
      }
```

Така процедура містить у собі три явно зайві операції, пов'язані зі збереженням, завантаженням і відновленням значення контрольного слова x87 FPU. Час виконання кожної з них вимірюється, як мінімум, декількома тактами процесора (у ряді випадків – десятьма і більше, залежно від конкретної реалізації мікроархітектури процесора). З погляду оптимізації коду, що існує в рамках архітектури IA-32, можна намітити два виходи з цієї ситуації.

Перший – це зберегти значення контрольного слова FPU, завантажити нове значення контрольного слова FPU з потрібним способом округлення, після чого зробити відразу цілий ряд перетворень і, нарешті, відновити початковий стан FPU. Саме такий спосіб рекомендується застосовувати у документації по оптимізації коду для процесорів x86 (зокрема, для AMD Athlon). Трудність такого підходу полягає в тому, що, по-перше, необхідна реорганізація коду, яка б дозволила згрупувати всі перетворення значень *float* в *int*. По-друге, саму процедуру перетворення доведеться писати у вигляді асемблерної вставки.

Другий – це використовувати доступні SIMD-розширення, на зразок SSE або 3Dnow! У першому випадку для цієї мети підходять команди cvtss2si і cvtps2pi (остання дозволяє здійснювати два перетворення одночасно, але ви-

користовує MMX-регістр, що, у свою чергу, приводить до необхідності перемикання режимів FPU/MMX). У другому випадку перетворення пари дійсних значень в цілочисельні можна здійснювати за допомогою команди pf2id. Тут з'являються труднощі, властиві набору 3Dnow! – використання MMX-регістрів, і, як наслідок, необхідність перемикання режимів роботи процесора (або очищення, або збереження/відновлення вмісту FPU-стека і MMX-регістрів). Як найбільш просту процедуру перетворення можна використати процедуру, оформлену у вигляді асемблерної вставки:

```
__asm
{
   cvttss2si  eax, dword ptr[f]
   mov        dword ptr[i], eax
}
```

Використання команд SSE на відміну від команд mmx/3Dnow! не вимагає додаткових витрат, пов'язаних з перемиканням режиму роботи процесора.

З командами розширення SSE3 це перетворення може бути таким:

```
__asm  {
fld    dword ptr[f] // завантаження значення float
fisttp dword ptr[i] // збереження значення int в режимі truncate
}
```

Використання нової інструкції fisttp дозволяє досягти значного скорочення об'єму коду, необхідного для перетворення дійсних чисел в цілі числа і для часу виконання останнього.

21.7.2. Обчислення з комплексними числами

Комплексна арифметика досить часто зустрічається у різних спектральних завданнях, зокрема – у завданнях обробки аудіоданих. До них належать швидке перетворення Фур'є, частотна фільтрація і таке інше. Серед нових розширень SSE3 можна виділити п'ять інструкцій, що дозволяють прискорити обчислення з комплексними числами. Сюди входять інструкції одночасного складання-віднімання addsubps і addsubpd і інструкції дублювання даних movsldup, movshdup і movddup (початковий операнд яких може бути адресою пам'яті). Перші дозволяють виключити операції зміни знака у частини елементів даних (зазвичай здійснювані за допомогою xorps/xorpd), другі – зайві операції розпаковування даних, що завантажуються з пам'яті (unpcklps / unpcklpd, shufps/shufpd).

Використання SSE3 в завданнях комплексної арифметики дозволяє досягти істотного виграшу в швидкості (час виконання цього коду, що використовує нові розширення, майже в два рази менший, ніж час виконання аналогічного SSE2-коду).

21.7.3. Кодування відео

Найбільш витратною з погляду процесорного часу операцією в завданнях кодування відеоданих зазвичай є операція, в якій блоки поточного кадру порівнюються з блоками попереднього кадру з метою знаходження якнайкращої

відповідності (критерієм останнього зазвичай є сума абсолютних різниць). Особливість такого алгоритму полягає в тому, що адреси блоків поточного кадру є вирівняними, тоді як блоки попереднього кадру зазвичай не є вирівняними за 16-байтною межею. У існуючій архітектурі IA-32 операції невирівняного завантаження SSE/SSE2 мають два серйозних недоліки:

- відсутність апаратної підтримки;
- можливість «розриву» (перетину кордону) рядка кеша.

Інструкція спеціалізованого завантаження невирівняних 128-бітових значень lddqu покликана вирішити цю проблему. В той же час, оскільки ця команда завантажує більшу кількість даних (32 байти, починаючи з вирівняної адреси), є ряд обмежень на її використання. Зокрема, її не рекомендується використовувати для некешованих даних, або для даних з об'єднанням запису, а також в ситуаціях, коли може очікуватися «завантаження після збереження».

Заміна movdqu на lddqu зменшує час виконання такого коду.

21.7.4. Векторні операції

Векторні операції найчастіше присутні в 3D-графічних застосуваннях, які мають відношення до геометрії. До них належать тривимірне моделювання, методи трасування променів і моделювання комп'ютер-них ігор.

Це операції з три- або чотиривимірними векторами: складання, віднімання і скалярного множення (множення вектора на число). Операндами інструкцій набору SSE є 128-бітові регістри XMM (або 128-бітове значення в пам'яті, яке повинне бути вирівняне по 16-байтній межі), вміст яких можна розглядати як чотиривимірні вектори, побудовані з чисел з одинарною точністю:

```
struct __declspec(align(16)) VECTOR4F
{
   float x;
   float y;
   float z;
   float w;
}
```

Арифметичні інструкції набору SSE дозволяють оперувати з такими векторами як єдиним цілим, тобто здійснювати ту саму операцію (складання, множення, віднімання, ділення) з кожним з компонентів двох векторів одночасно. За рахунок цього при використанні розширень SSE досягається значний виграш в продуктивності. Як приклад розглянемо операцію складання двох трикомпонентних векторів: A і B (четвертий компонент векторів в даному випадку не використовується).

```
VECTOR4F A, B;
__asm
{
   movaps   xmm0, xmmword ptr[A]
   movaps   xmm1, xmmword ptr[B]
   addps    xmm0, xmm1
   movaps    xmmword ptr[A], xmm0
}
```

З цього прикладу видно, що для здійснення операції додавання необхідна всього одна інструкція addps (за умови, що як початкові вектори, так і результат складання зберігаються в XMM-регістрах, тобто операції завантаження-вивантаження даних не потрібні). Для здійснення подібної операції за допомогою FPU треба було б використовувати набагато більшу кількість операцій:

```
VECTOR4F A, B;
__asm
{
   fld      dword ptr[A]
   fadd     dword ptr[B]
   fld      dword ptr[A+4]
   fadd     dword ptr[B+4]
   fld      dword ptr[A+8]
   fadd     dword ptr[B+8]
   fxch     st(2)
   fstp     dword ptr[A]
   fstp     dword ptr[A+4]
   fstp     dword ptr[A+8]
}
```

Такий самий виграш в швидкості при використанні SSE можна отримати і в будь-яких інших операціях з векторами, які здійснюються поелементно, іншими словами, вертикально (тобто коли та сама арифметична операція застосовується по відношенню до кожного *i*-го елемента обох векторів: A і B). Проте у векторній алгебрі відомий і ряд інших операцій з векторами, здійснюваних з різними елементами того самого вектора, або горизонтально. Найяскравіший тому приклад – скалярний добуток двох векторів або обчислення довжини вектора (квадратного кореня зі скалярного добутку вектора самого на себе).

Команди розширення SSE3 дозволяють отримати серйозний виграш в продуктивності.

21.8. SSSE3-доповнення

Ці команди є доповненням до SSE3-розширення. Команди SSSE3 доступні для процесорів Xeon 5100, а також для процесорів Intel Core 2.

Новими в SSSE3, в порівнянні з SSE3, є 16 команд, що працюють з упакованими цілими числами. Кожна з них може працювати як з 64-розрядними (MMX), так і з 128- розрядними (XMM) регістрами.

PABSB, **PABSW**, **PABSD** – абсолютна величина
Input – { A0, A1… }
Output – { A0 * sign(A0), A1 * sign (A1)… }

Кожне поле результату є абсолютна величина відповідного поля з src1. Фактично це ті ж операції PSIGNB, PSIGNH, PSIGNW, у яких обидва аргументи один і той же регістр.

PSIGNB, PSIGNW, PSIGND – упакований знак
Input – { A0, A1… }, { B0, B1… }

Output – { A0 * sign(B0), A1 * sign (B1)… }

Кожне поле результату є добуток поля з src1 на {-1,0,1} залежно від знака відповідного поля в src2 (множення на 0, коли поле в src2 дорівнює нулю).

PALIGNR – пакування двох регістрів в один

Input – { A0, A1 }, { B0, B1 }, imm8
Output – { B1_B0_A1_A0 >> (imm8 * 8) }

Два регістри операнда розглядаються як один беззнаковий регістр подвоєної розмірності, з якого витягується 64-/128-бітове значення, починаючи з байта вказаного в безпосередньому аргументі-константі команди.

PSHUFB – змішування байтів

Input – { A0, A1, A2,.. A7/A15 }, { B0, B1, B2,.. B7/B15 }
Output – { [AB0 AB1 AB2 …] }

Команда виконує перестановку байтів. Кожен байт результату є байт з першого аргументу, визначуваний по відповідному байту з другого аргументу (якщо байт негативний, то в байт результат прописується нуль, інакше використовуються молодші 3 або 4 біти як номер байта в першому аргументі).

PMULHRSW – перемножити послівно (16 бітів) з округленням і масштабуванням

Input – { A0, A1… }, { B0, B1… }
Output – { A0 * B0, A1 * B1… }

Аргументи A та B розглядаються як вектори 16-бітових знакових чисел з фіксованою комою, поданих в діапазоні [-1,+1) (тобто 0x4000 – це 0,5, а 0x000 – це -0,75 й т. д.), які перемножуються один з одним з коректним округленням.

Команда застосовується для множення двох пікселів (8-бітової компоненти кольору) і замінює код:

```
; mm0 *= mm1
movq   mm2, mm0
pmulhuw mm0, mm1     ; hi = (a * b) >> 16
pmullw  mm2, mm1     ; lo = (a * b) & 0xFFFF
psllw   mm0, 1       ; hi <<= 1
psrlw   mm2, 14      ; lo >>= 14
paddw   mm2, mm4     ; lo += 1 (mm4 = 4x0001)
psrlw   mm2, 1       ; lo >>= 1
paddw   mm0, mm2     ; hi += lo
```

Алгоритм роботи цієї інструкції можна описати як (http://forum.ixbt.com/topic.cgi?id=8:20531):

1) помножити signed word (16 бітів) з операнда -1 на signed word з операнда-2;

2) результат множення (32-бітове число) зсунути управо на 14 розрядів;

3) до кожного отриманого 18-бітового числа додати 1;

4) узяти з отриманого результату 16 бітів (починаючи з біта 1 по біт 16 – якщо вважати від 0), і помістити це значення на місце відповідного signed word операнда-1.

Наприклад, є два числа (12.34 и 56.78). Якщо floating point не підтримується, то числа можна представити як цілі: 1234 и 5678 (ці числа в 100 разів

більші, ніж реальні значення). При складанні: 1234 + 5678 = 6912 (12.34 + 56.78 = 69.12). Тепер спробуємо провести множення: 1234 * 5678 = 7006652 (тоді як 12.34 * 56.78 = 700.6652). Очевидно, що для коректування, результат множення потрібно зсунути на 2 розряди. При цьому "зайві" 2 розряди можна викинути, тобто провести звичайний зсув і отримати 70066 (аналог 700.66). А можна провести округлення 70067 (аналог 700.67).

PMADDUBSW – побайтове перемноження
Input – { A0, A1, A2, A3,.. }, { B0, B1, B2, B3,.. }
Output – { (A0*B0+A1*B1), (A2*B2+A3*B3), … }
Проводиться побайтове перемножування векторів A та B, проміжні 16-бітові результати попарно складаються між собою з насиченням і видаються як результат.

PHSUBW, PHSUBD — горизонтальне віднімання цілих 16/32-бітових значень
Input – { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
Output – { A0-A1 A2-A3 … B0-B1 B2-B3 …}

PHSUBSW – горизонтальне віднімання цілих 16-бітових полів з насиченням.
Input – { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
Output – { A0-A1 A2-A3 B0-B1 B2-B3 }

PHADDW, PHADDD – горизонтальне додавання цілих 16/32-бітових значень
Input – { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
Output – { A0+A1 A2+A3 … B0+B1 B2+B3 …}

PHADDSW – горизонтальне додавання цілих 16-бітових полів з насиченням
Input – { A0, A1, A2, A3 }, { B0, B1, B2, B3 }
Output – { A0+A1 A2+A3 … B0+B1 B2+B3 …}

21.9. Лабораторна робота "SSE3-команди"

Мета заняття:

– поглибити і закріпити знання з архітектури МП платформи x86 і навички його програмування;

– набути практичних навичок складання, налагодження і виконання програм з використанням команд SSE3, написаних мовою асемблера для програмування МП платформи x86.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати варіант свого завдання та написати на асемблері програму за допомогою SSE3-команд, ***вивести на екран варіант завдання, його вміст, результат, прізвище та e-mail*** автора програми.

Завдання

1. Написати програму дослідження команди PABS та вивести результат на екран.
2. Написати програму дослідження команди PSIGN та вивести результат на екран.
3. Написати програму дослідження команди PALIGNR та вивести результат на екран.
4. Написати програму дослідження команди PSHUFB та вивести результат на екран.
5. Написати програму дослідження команди PMULHRSW та вивести результат на екран.
6. Написати програму дослідження команди PHSUB та вивести результат на екран.
7. Написати програму дослідження команди PHSUBSW та вивести результат на екран.
8. Написати програму дослідження команди PHADDSW та вивести результат на екран.
9. Написати програму дослідження команди PMADDUBSW та вивести результат на екран.
10. Написати програму дослідження команди PHADDSW та вивести результат на екран.

Зміст звіту

1. Постановка задачі для конкретного варіанта завдання.
2. Блок-схема алгоритму виконання прикладу з детальним коментарем та описом роботи.
3. Лістинг програми з виведенням даних на екран монітора з використанням API-функцій та з детальним коментарем і описом роботи.
4. Print screen екрана 32-розрядного налагоджувача з виконаною програмою.
5. Короткий опис виконання програми.
6. Висновки за результатами роботи.

Контрольні запитання

1. Яке призначення команд SSE3-розширення?
2. Який формат даних SSE3-розширень?
3. На які групи поділяються команди SSE3-розширень?
4. Які особливості використання команд SSE3-розширення?

22. SSE4-РОЗШИРЕННЯ

Набір команд SSE4, що був анонсований 27.09.2006 року, складається з 54 інструкцій, 47 з яких належать до SSE4.1 (вони є в процесорах Penryn+), 7 – до SSE4.2 (http://www.intel.com/software/penryn/). Усі команди призначені для 128-розрядних регістрів XMM[0 – 15]. Компілювати ці команди можливо на C++ з 10-ї версії компілятора з використанням опції –QxS (http://cracklab.ru/).

22.1. SSE4.1-розширення

22.1.1. Прискорення відео

Команда **MPSADBW** xmm1, xmm2/m128, imm8 – обчислення восьми сум абсолютних значень різниць (SAD) зміщених 4-байтових беззнакових груп:

Input – {A_0, A_1, ... A_{14}}, {B_0, B_1, ... B_{15}}, Shiftmode

Output – {SAD_0, SAD_1, SAD_2, SAD_7}

Розташування операндів для 16-бітних SAD визначається 3-ма бітами безпосереднього аргументу imm8:

s1 = imm8[2]*4

s2 = imm8[1:0]*4

$SAD_0 = |A_{(s1+0)}-b_{(s2+0)}| + |A_{(s1+1)}-b_{(s2+1)}| + |A_{(s1+2)}-b_{(s2+2)}| + |A_{(s1+3)}-b_{(s2+3)}|$

$SAD_1 = |A_{(s1+1)}-b_{(s2+0)}| + |A_{(s1+2)}-b_{(s2+1)}| + |A_{(s1+3)}-b_{(s2+2)}| + |A_{(s1+4)}-b_{(s2+3)}|$

$SAD_2 = |A_{(s1+2)}-b_{(s2+0)}| + |A_{(s1+3)}-b_{(s2+1)}| + |A_{(s1+4)}-b_{(s2+2)}| + |A_{(s1+5)}-b_{(s2+3)}|$

...

$SAD_7 = |A_{(s1+7)}-b_{(s2+0)}| + |A_{(s1+8)}-b_{(s2+1)}| + |A_{(s1+9)}-b_{(s2+2)}| + |A_{(s1+10)}-b_{(s2+3)}|$

Команда **PHMINPOSUW** xmm1, xmm2/m128 – пошук серед 16-бітових беззнакових полів A_0...A_7 такого, який має мінімальне значення (і позицію з меншим номером, якщо таких полів декілька). Повертається 16-бітове значення та його позиція:

Input – {A_0, A_1,… A_7}

Output – {MinVal, MinPos, 0, 0…}

Команда **PMOV{SX,ZX}B,W,D}** xmm1, xmm2/m{64,32,16} – група з 12-ти інструкцій для розширення формату упакованих полів. Упаковані 8-, 16-, або 32-бітові поля з молодшої частини аргументу розширюються (із знаком або без) в 16-, 32- або 64-бітові поля результату (табл. 22.1).

Таблиця 22.1

Вхідний формат			Результуючий формат
8 бітів	16 бітів	32 біти	
PMOVSXBW PMOVZXBW			16 бітів
PMOVSXBD	PMOVSXWD		32 біти

PMOVZXBD	PMOVZXWD		
PMOVSXBD PMOVZXBD	PMOVSXWD PMOVZXWD	PMOVSXWQ PMOVZXWQ	64 біти

22.1.2. Векторні примітиви

Команда **P{MIN,MAX}{SB,UW,SD,UD}** xmm1, xmm2/m128 – обчислення min/max значення полів двох аргументів.

Кожне поле результату є мінімальним/максимальним значенням відповідних полів двох аргументів. Байтові поля розглядаються тільки як числа із знаком, 16-бітові – тільки як числа без знака. Для 32-бітових упакованих полів передбачений варіант як із знаком, так й без нього.

Команда **PMULDQ** xmm1, xmm2/m128 – перемноження 32-бітових полів із знаком із видачею повних 64 бітів результату (дві операції множення над 0 та 2 полями аргументів):

Input — $\{A_0, A_1, A_2, A_3\}$, $\{B_0, B_1, B_2, B_2\}$

Output — $\{A_0*B_0, A_2*B_2\}$

Команда **PMULLD** xmm1, xmm2/m128 – перемноження 32-бітних полів із видачею молодших 32-бітових результатів (чотири операції множення над всіма полями аргументів):

Input — $\{A_0, A_1, A_2, A_3\}$, $\{B_0, B_1, B_2, B_2\}$

Output — $\{low32(A_0*B_0), low32(A_1*B_1), low32(A_2*B_2), low32(A_3*B_3)\}$

Команда **PACKUSDW** xmm1, xmm2/m128 – упаковка 32-бітових полів із знаком в 16-бітові поля без знака з насиченням.

Команда **PCMPEQQ** xmm1, xmm2/m128 – перевірка 64-х бітових полів на рівняння з видачею 64-бітових масок.

22.1.3. Вставки/витягання

Команда **INSERTPS** xmm1, xmm2/m32, imm8 – вставка 32-бітового поля з xmm2 (можливо вибрати будь-який з 4 полів цього регістра) або 32-бітового елемента пам'яті в довільне поле результату. Крім того, для кожного з полів результату можна задати скидання його в +0.0.

Команда **EXTRACTPS** r/m32, xmm, imm8 – витягання 32-бітового поля із xmm-регістра; номер поля указується в молодших 2 бітах imm8. Якщо як результат вказаний 64-бітовий регістр, то його старші 32 біти скидаються (розширення без знака).

Команда **PINSR{B,D,Q}** xmm, r/m*, imm8 – вставка 8-, 32-, або 64-бітового значення у вказане поле xmm регістра (решта полів не змінюється).

Команда **PEXTR{B,W,D,Q}** r/m*, xmm, imm8 – витягання 8-, 16-, 32-, 64-бітового поля з вказаного в imm8 поля xmm регістра. Якщо як результат вказаний регістр, то його старша частина скидається (розширення без знака).

22.1.4. Скалярне множення векторів

Команда **DPPS** xmm1, xmm2/m128, imm8 – скалярне перемноження одинарної точності.

Команда **DPPD** xmm1, xmm2/m128, imm8 – скалярне перемноження подвійної точності.

За допомогою бітової маски в imm8 вказується, які добутки полів повинні додаватися та що слід прописати в кожне поле результату: суму вказаних добутків або +0.0.

22.1.5. Змішування

Команда **BLENDV{PS,PD}** xmm1, xmm2/m128, <xmm0> – змішування одинарних значень або подвійної точності.

Вибір кожного 32-/64-бітового поля результату здійснюється залежно від знака такого ж поля в неявному аргументі xmm0 або з першого, або з другого аргументу.

Команда **BLEND{PS,PD}** xmm1, xmm2/m128, imm8 – змішування відповідно до маски.

Бітова маска (4 або 2 біти) в imm8 вказує, з якого аргументу слід взяти кожне 32/64-бітове поле результату.

Команда **PBLENDVB** xmm1, xmm2/m128, <xmm0> – змішування відповідно до знака.

Вибір кожного байтового поля результату здійснюється залежно від знака байта такого ж поля в неявному аргументі xmm0 або з першого, або з другого аргументу.

Команда **PBLENDW** xmm1, xmm2/m128, imm8 – змішування упакованих слів залежно від маски.

Бітова маска (8 біт) в imm8 вказує, з якого аргументу слід узяти кожне 16-бітове поле результату.

22.1.6. Перевірка бітів

Команда **PTEST** xmm1, xmm2/m128 – логічне порівняння.

Встановити прапорець ZF, якщо в xmm2/m128 всі біти помічені нульовою маскою із xmm1. Якщо всі не помічені біти дорівнюють нулю, то встановити прапорець CF. Решта прапорців (AF, OF, PF, SF) завжди скидається. Інструкція не модифікує xmm1.

22.1.7. Округлення

Команда **ROUND{PS, PD}** xmm1, xmm2/m128, imm8 – округлення чисел одинарної чи подвійної точності.

Округлення всіх 32/64-бітових полів. Режим округлення (4 варіанти) вибирається або з MXCSR.RC, або задається безпосередньо в imm8. Також можна "подавити" генерацію виключення втрати точності.

Команда **ROUND{SS, SD}** xmm1, xmm2/m128, imm8 – округлення скалярних чисел (одинарного чи подвійного значення).

Округлення тільки молодшого 32/64- бітового поля (решта біт залишається незмінними).

22.1.8. Читання WC пам'яті

Команда **MOVNTDQA** xmm1, m128 – (Load Double Quadword Non-Temporal Aligned Hint). Ця команда читання дозволяє прискорити (до 7,5 разу) роботу з write-combining областями пам'яті.

22.2. SSE4.2-розширення

22.2.1. Обробка рядків

Ці інструкції виконують арифметичні порівняння між всіма можливими парами полів (64 або 256 порівнянь) з обох рядків, заданих вмістом xmm1 та xmm2/m128. Потім булеві результати порівнянь обробляються для отримання потрібних результатів. Безпосередній аргумент imm8 керує розміром (байтові рядки або unicode, до 16/8 елементів кожний), знаковістю полів (елементів рядків), типом порівняння та інтерпретацією результатів.

Ними можна проводити в рядку (області пам'яті) пошук символів із заданого набору чисел або в заданих діапазонах. Можна порівнювати рядки (області пам'яті) або проводити пошук підрядків.

Всі вони роблять вплив на прапорці процесора: прапорець SF встановлюється, якщо в xmm1 не повний рядок, ZF – якщо в xmm2/m128 не повний рядок, CF – якщо результат ненульовий, OF – якщо молодший біт результату ненульовий. Прапорці AF та PF скидаються.

Команда **PCMPESTRI** <ecx>, xmm1, xmm2/m128, <eax>, <edx>, imm8 – явне завдання розміру рядків в <eax>, <edx> (береться абсолютна величина регістрів з насичення до 8/16, залежно від розміру елементів рядків. Результат в регістрі ecx.

Команда **PCMPESTRM** <xmm0>, xmm1, xmm2/m128, <eax>, <edx>, imm8 – явне завдання розміру рядків в <eax>, <edx> (береться абсолютна величина регістрів з насичення до 8/16, залежно від розміру елементів рядків. Результат в регістрі xmm0.

Команда **PCMPISTRI** <ecx>, xmm1, xmm2/m128, imm8 – неявне завдання розміру рядків (проводиться пошук нульових елементів до кожного з рядків). Результат в регістрі ecx.

Команда **PCMPISTRM** <xmm0>, xmm1, xmm2/m128, imm8 – неявне завдання розміру рядків (проводиться пошук нульових елементів до кожного з рядків). Результат в регістрі xmm0.

22.2.2. Підрахунок контрольної суми CRC32

Команда **CRC32** r32, r/m – підрахунок CRC32. Команда виконує накопичення значення контрольної суми CRC-32C (інші позначення CRC-32/ISCSI CRC-32/CASTAGNOLI) для 8-, 16-, 32- або 64-бітового аргументу (використовується поліном 0x11EDC6F41).

22.2.3. Підрахунок популяції одиничних бітів

Команда **POPCNT** r, r/m – підрахунок числа одиничних бітів. Три варіанти інструкції: для 16-, 32- та 64-бітових регістрів. Команда присутня в SSE4A від фірми AMD.

22.2.4. Векторні примітиви

Команда **PCMPGTQ** xmm1, xmm2/m128 – перевірка 64-х бітових полів на "більше ніж" та видача 64-бітових масок.

22.3. Формати даних SSE 1-4 команд

Формати даних SSE бувають двух форм:

– формати з плаваючою точкою (floating-Point). В 128 розрядах можуть розміщуватися 4 32-розрядних числа (4 × DWORD – Single-Precision), або 2 64 -розрядних числа (2 × QWORD – Double-Precision);

– формати з фіксованою точкою (Integer). В 128 розрядах можуть розміщуватися один з типів:

16 × BYTE;
8 × WORD;
4 × DWORD;
2 × QWORD.

Завдання для практичного використання

Сформулювати задачу з використанням команди однієї з групи команд згідно з варіантом:

1. Прискорення відео.

2. Векторні примітиви.
3. Вставки/витягання.
4. Скалярне множення векторів.
5. Змішування.
6. Перевірка біт.
7. Округлення.
8. Обробка рядків.
9. Підрахунок контрольної суми CRC32.
10. Векторні примітиви.

Контрольні запитання

1. Яке призначення команд SSE4-розширення?
2. Який формат даних SSE4-розширень?
3. На які групи поділяються команди SSE4-розширень?

http://blogs.kpi.kharkov.ua/v2/asm/knigi/

23. AVX-РОЗШИРЕННЯ

23.1. Особливості використання команд AVX-розширення

Спочатку 30 августа 2007 г. фірма AMD запропонувала розширення системи команд, яку назвали SSE5 (англ. Streaming SIMD Extensions version 5) як доповнення до 128-бітних команда ядра в архітектурі AMD64.

AMD вирішила не реалізовувати SSE5 так, як пропонувалося спочатку. У травні 2009 AMD замінила SSE5 трьома меншими наборами команд, названими XOP, FMA4 та CVT16, які зберігають запропоновану функціональність SSE5, але по-іншому кодують інструкції для кращої сумісності з набором команд AVX, запропонованим компанією Intel. Ці команди використовуються в ядрах з 32 нм технологічним процесом.

Команди AVX (Advanced Vector Extensions) використовують для інтенсивних обчислень з плаваючою точкою в мультимедіа, наукових і фінансових завданнях. AVX-розширення збільшує ступінь паралелізму і пропускну спроможність в дійсних SIMD обчисленнях, зменшує тиск (навантаження) на регістри завдяки неруйнуючим трьохоперандним операціям.

Для використання AVX потрібна його підтримка не тільки мікропроцесором, але і операційною системою (починаючи з Windows 7).

Поліпшення (http://habrahabr.ru/post/99367/):

– нова схема кодування інструкцій;

– розмір векторних регістрів SIMD збільшується з 128 (XMM) до 256 біт (**регістри YMM0 — YMM15**). 128-бітні SSE інструкції використовують молодшу половину регістрів YMM, не змінюючи старшу частину. Для роботи з YMM регістрами додані нові 256-бітні AVX інструкції. В майбутньому можливе розширення векторних регістрів SIMD до 512 або 1024 біт;

– неруйнуючі операції. Набор AVX інструкцій використовує трьохоперандний синтаксис. Наприклад, замість a = a + b можна використовувати c = a + b, при цьому регістр a залишається незміненим, наприклад:

```
VPADDW   xmm1, xmm2, xmm3;
```

– для більшості нових інструкцій відсутні вимоги до вирівнювання операндів в пам'яті. Проте, рекомендується стежити за вирівнюванням на розмір операнда, щоб уникнути значного зниження продуктивності;

– набір інструкцій AVX містить в собі аналоги 128-бітних SSE інструкцій для дійсних чисел. При цьому, на відміну від оригіналів, збереження 128-бітного результату обнуляє старшу половину YMM регістра. 128-бітні AVX інструкції зберігають інші переваги AVX, такі як, нова схема кодування, трехоперандний синтаксис і невирівняний доступ до пам'яті. Рекомендується відмовитися від старих SSE інструкцій на користь нових 128-бітних AVX інструкцій, навіть якщо досить два операнди. Наприклад, команда з використанням 256-розрядних регыстрыв YMM:

```
VADDPS   ymm1, ymm2, ymm3.
```

Для визначення можливості підтримки команд AVX необхідно в 64-розрядній ОС викликати CPUID з eax = 1 і переконатися, що після цього в 28-му біті регістра ecx знаходиться одиниця.

Лістинг 23.1. Перевірка підтримки команд AVX:

```
        .686                    ; директива определения типа микропроцессора
        .model  flat, stdcall ; задание линейной модели памяти и соглашения ОС Windows
        option casemap:none ; отличие строчных и прописных букв
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32
.data
   titl db "Проверка микропроцессора на поддержку команд AVX",0 ; название окна
   szInf db "Команды AVX ПОДДЕРЖИВАЮТСЯ!!!",0  ;
.code
_start:
; при использования 64-разрядной ОС
      mov   EAX, 1
      cpuid ; по содержимому eax производится идентификация микропроцессора
      and  ecx,10000000h ; ecx:= ecx Ù 1000 0000h  (28 розряд)
      jnz  exit1       ; перейти на exit, если не нуль
invoke MessageBox,0,chr$("Команды AVX микропроцессором НЕ поддержи-
ваются"),chr$("Проверка поддержки команд AVX"),MB_OK
      jmp exit2
exit1:
invoke MessageBox, 0, addr szInf, addr titl, MB_ICONINFORMATION;+180000h ; вид окна
exit2:
      invoke ExitProcess, 0 ; возвращение управления ОС и освобождение ресурсов
      END   _start          ; окончание программы с именем _start
```

23.2. Схема кодування

Схема кодування інструкцій AVX використовує VEX префікс. Зараз існують два префікса, довжиною 2 та 3 байти. Для 2-байтного префікса перший байт дорівнює 0xC5, для 3-байтного 0xC4. В 64-бітному режимі перший байт префікса унікальний. В 32-бітному режимі виникає конфлікт з інструкціями LES та LDS, який вирішується старшим бітом другого байта. Він має значення тільки в 64-бітному режимі, через непідтримувані форми інструкцій LES и LDS. Довжина AVX інструкцій, разом з префіксом, не перевищує 11 байт. У наступній версії AVX2 з'явилися довші інструкції.

23.3. Набор команд AVX

Всі команди AVX описані в довіднику, який можна знайти на сайті Intel (http://software.intel. com/en-us/intel-isa-extensions). Команди AVX є розширенням наборів команд SSE і розширюють 128-бітові регістри SSE до 256 бітів. Нові 256-бітові регістри позначаються як ymm0-ymm15 (для 32- бітової програми доступні тільки ymm0-ymm7). При цьому 128-бітові SSE регістри

xmm0-xmm15 посилаються на молодші 128 бітів відповідного AVX регістра (http://habrahabr.ru/post/99367/).

Більшість команд є лише зміненими версіями інструкцій SSE. Для цього кожна інструкція з SSE має в AVX свій аналог з префіксом v.

SSE2-код

```
movdqa xmm2, xmm0
punpcklbw xmm0, xmm1
punpckhbw xmm2, xmm1
```

може бути переписаний з AVX як

```
vpunpckhbw xmm2, xmm0, xmm1
vpunpcklbw xmm0, xmm0, xmm1.
```

При цьому команди з префіксом v зануляют старші 128 бітів AVX регістра, в який вони пишуть. Наприклад, інструкція vpaddw xmm0, xmm1, xmm2 занулить старші 128-біт регістра ymm0.

Крім того, деякі SSE- інструкції були розширені в AVX для роботи з 256-бітовими регістрами. До таких інструкцій відносяться всі команди, що працюють з числами з плаваючою точкою. Наприклад наступний AVX код

```
vmovapd ymm0, [esi]
vmulpd ymm0, ymm0, [edx]
vmovapd [edi], ymm0
```

обробляє відразу 4 double.

AVX- розширення включає деякі нові інструкції:

vbroadcastss/vbroadcastsd/vbroadcastf128 – заповнення AVX регістра одним і тим же завантаженим значенням;

vmaskmovps/vmaskmovpd – умовне завантаження/збереження float/double чисел в AVX регістр залежно від знаку чисел в іншому AVX регістрі;

vzeroupper - обнулення старших 128 бітів всіх AVX регістрів;

vzeroall – повне обнулення всіх AVX регістрів;

vinsertf128/vextractf128 – вставка/отримання будь-якої 128-бітової частини 256-бітового регістра AVX;

vperm2f128 – перестановка 128-бітових частин 256-бітового регістра AVX. Параметр перестановки задається статично;

vpermilps/vpermilpd – перестановка float/double чисел усередині 128-бітових частин 256-бітового регістра AVX. При цьому параметри перестановки беруться з іншого AVX регістра;

vldmxcsr/vstmxcsr – завантаження/збереження параметрів AVX, що управляють;

xsaveopt – отримання підказки про те, які AVX-регистри містять дані. Ця команда зроблена для розробників ОС і допомагає їм прискорити перемикання контексту.

23.4. Використання AVX-инструкций

Особливості використання AVX-інструкцій:

✓ небажано змішувати SSE- та AVX-інструкції. Щоб перейти від виконання AVX-інструкцій до SSE-інструкцій процесор зберігає в спеціальному кеші верхні 128 бітів AVX регістрів, на що може піти півсотні тактів. Після виконання SSE-інструкцій процесор знову повернеться до виконання AVX-інструкцій. Він відновить верхні 128 бітів AVX регістрів, на що піде ще півсотні тактів;

✓ уникати збереження верхньої частини AVX регістрів при переході до SSE-коду. Цього можна досягти, якщо занулити верхні 128 бітів AVX регістрів за допомогою команди vzeroupper або vzeroall. Ці команди працюють дуже швидко;

✓ вирівнювати на 16 байт перед виконанням команд завантаження/збереження вирівняних даних vmovaps/vmovapd/vmovdqa;

✓ підпрограма на Windows x64 не повинна змінювати регістри xmm6-xmm15 (або відповідні ним регістри ymm6-ymm15). Перед виконанням підпрограми необхідно їх зберегти в стеку і перед виходом з підпрограми відновити їх із стека.

Для того, щоб перевірити виконання програм на мікропроцесорі, який підтримує команди AVX використання середовища masm32 v11r недостатньо! Треба використовувати файл ml.exe, який розуміє команди AVX (починаючи з Sandy Bridge, 2011). Потрібний файл файл ml.exe може бути лише в Visual Studio версії 2010 та вище (див. підрозділ 20.8).

Приклад додавання двох чисел за допомогою команди vaddpd набору AVX наведено в лістингу 23.2.

Лістинг 23.2. Приклад додавання двох чисел за допомогою команди vaddpd:

```
        .686p  ; директива определения типа микропроцессора
        .xmm
        .model  flat, stdcall ;
        option casemap:none ; отличие строчных и прописных букв
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32,masm32
.data
        a dd 8 dup(0f0h)
        b dd 8 dup(0fh)
        res dd 8 dup(0)
        titl1 db "Исследование",0
        buf dd 8 dup(0),0
        frmt dd " %d ",0
.code
_st:
        vmovups ymm0, a
        vmovups ymm1, b
        vaddpd ymm0,ymm0,ymm1
        vmovups res,ymm0
        invoke ExitProcess,0
```

```
end _st
```

Команда vaddpd додає два числа: одне з регістру ymm0, а друге – з ymm1. Результат додавання записується до регістру ymm0.

Приклад циклічного додавання командою vpaddd з виведенням результату через MessageBox наведено в лістингу 23.3.

Лістинг 23.3. Приклад циклічного додавання двох масивів чисел за допомогою команди vpaddd:

```
.686
.xmm
.model flat, stdcall ;
option casemap:none ;
include \masm32\include\windows.inc
include \masm32\macros\macros.asm
uselib kernel32, user32,masm32
.data
mas1 dd 268, 175, 357, 657
mas2 dd  27, 674, 354, 874
res dd 4 dup(0),0
buf dd 4 DUP(0),0
_info db "Пример работы команды AVX vpaddd", 0Ah, 0Dh,
   "(циклическое сложение).", 0Ah, 0Dh,
   "Результат работы: ",
   4 dup("%d "),0
_title db "Мельников А.С. КИТ-12",0
.code
_start:
  vmovups xmm1, mas1 ; переслати невирівняні упаковані значення
  vmovups xmm2, mas2 ; переслати невирівняні упаковані значення
vpaddd xmm0, xmm1, xmm2 ; операція циклічного складання
; перший операнд - для результату, останні два - початкові дані
    vunpcklpd xmm4, xmm4, xmm0 ; розпаковування молодших половин
    vunpckhpd xmm4, xmm4, xmm5 ; розпаковування старших половин
    vunpckhpd xmm5, xmm5, xmm0 ; распаковывание старших половин
    vunpckhpd xmm5, xmm5, xmm6 ; розпаковування старших половин
vcvtdq2pd xmm0, xmm4 ; перетворення чисел на числа з подвійною точністю
vcvtdq2pd xmm1, xmm5 ; перетворення чисел на числа з подвійною точністю
    vunpcklpd xmm4, xmm4, xmm0 ; розпаковування молодших половин
    vunpckhpd xmm4, xmm4, xmm6 ; розпаковування старших половин
    vunpckhpd xmm5, xmm5, xmm0 ; розпаковування старших половин
    vunpckhpd xmm5, xmm5, xmm4 ; розпаковування старших половин
vcvttsd2si eax, xmm4 ; перетворення числа подвійної точності
vcvttsd2si ebx, xmm5 ; перетворення числа подвійної точності
mov dword ptr res, eax ;
mov dword ptr res[4], ebx ;
    vunpcklpd xmm4, xmm4, xmm1 ; розпаковування молодших половин
    vunpckhpd xmm4, xmm4, xmm6 ; розпаковування старших половин
    vunpckhpd xmm5, xmm5, xmm1 ; розпаковування старших половин
    vunpckhpd xmm5, xmm5, xmm4 ; розпаковування старших половин
vcvttsd2si eax, xmm4 ; перетворення числа подвійної точності
vcvttsd2si ebx, xmm5 ; перетворення числа подвійної точності
mov dword ptr res[8], eax ;
mov dword ptr res[12], ebx ;
    invoke wsprintf,ADDR buf, ADDR _info, res, res[4], res[8], res[12] ;
    invoke MessageBox, 0, addr buf, addr _title, 0 ;
```


http://blogs.kpi.kharkov.ua/v2/asm/knigi/

```
        invoke ExitProcess, 0 ;
        end _start
```

В програмі виконується додавання двох чисел командою vpaddd. Для виведення результату через функцію MessageBox командами vunpcklpd та vunpckhpd виконується розпакування упакованого результату. А командою vcvttsd2si виконується скалярне перетворення дійсного числа подвійної точності в ціле 32-розрядне число. Результат виконання програми навдено на рис. 23.1.

Приклад додавання двох масивів чисел командою vaddpd набору AVX vpaddd з виведенням результату через MessageBox наведено в лістингу 23.4.

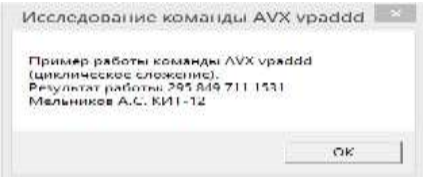


Рис. 23.1. Результат виконання програми

Лістинг 23.4. Приклад додавання двох масивів чисел командою vaddpd набору AVX з виведенням результату через MessageBox:

```
        .686
        .xmm
        .model flat, stdcall ;
        option casemap:none
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32,masm32
.data
        a1 dd 8,16,1,20,9,15,36,25,68,12,54,36,10,12,74,52    ;16 чисел
        b1 dd 2,20,21,5,3,5,6,5,14,12,85,65,12,30,42,72
        len1 EQU ($-b)/type b1
        res1 dd 16 dup(0),0
        frmt db "Использование команды vaddpd",0dh,0dh,
        "8 16 1  20 9 15 36 25 68 12 54 36 10 12 74 52",0dh,0dh,
        "2 20 21 5  3 5  6  5  14 12 85 65 12 30 42 72",0dh,0dh
        buf db 100 dup(0),0
        buf1 db " %d ",0
        titl1 db "Автор:              , НТУ ХПИ",0
.code
_st:
        mov eax,len1 ;
        mov ebx,8    ; 32 x 8 = 256
        div ebx,       ;
        mov ecx,eax
        lea esi,a1
        lea edi,b1
        lea ebx,res
m1:     vmovups ymm0, [esi]
        vmovups ymm1, [edi]
        vaddpd ymm0,ymm0,ymm1
```

```
        vmovups [ebx],ymm0
        add edi,32
        add esi,32
        add ebx,32
    loop m1
        cmp edx,0h    ; порівняння edx із нулем
        jz exit1          ; перейти, якщо нуль
        mov ecx, edx ; занесення вмісту edx до ecx
m2:     mov eax, [esi]       ; занесення елемента mas1
        vaddss  eax, [edi]        ; елемента mas2
        mov dword ptr [ebx], eax ; занесення вмісту у res1
        add esi,4   ; зміщення вказівника на mas1
        add edi,4   ; зміщення вказівника на mas2
        add ebx,4  ; зміщення вказівника на res1
loop m2
exit1:  lea esi, dword ptr res1 ; занесення до esi адреси res1
        xor ebx, ebx      ; заповнення нулями ebx
        mov ecx, len1 ; занесення len1 у ecx
        @@:    mov eax, [esi]    ; занесення елемента res1
               push ecx        ; занесення ecx у стек
               push esi        ; занесення esi у стек
               push ebx        ; занесення ebx у стек
        invoke wsprintf,ADDR buf[ebx],ADDR frmt,eax
               pop ebx                  ; витягнення ebx зі стеку
               add ebx,4; збільшення ebx
               mov buf[ebx],20h ; занесення до буфера пробілу
               add ebx,1
               pop esi
               add esi,2
               pop ecx
        loop @b
        invoke MessageBox, 0, ADDR frmt, ADDR titl1, MB_OK
        invoke ExitProcess,0
        end _st
```

Результат виконання програми навдено на рис. 23.2.

Рис. 23.2. Результат виконання програми

В лістингу 23.5 наведено програму витягання 2-го числа з регістру xmm.

Лістинг 23.5. Програма витягання 2-го числа з регістру xmm:

```
        .686
        .xmm
        .model flat, stdcall ;
        option casemap:none ;
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32,masm32
.data
        b dd 10, 8, 9, 6
        titl1 db "Исследование",0
```

```
        buf db ?
        frmt db "%d",0
.code
_st:
        vmovups xmm2, b
        vpextrd eax,xmm2,2                  ; витягання 2-го числа з xmm
        invoke wsprintf, addr buf, addr frmt, eax
        invoke MessageBox, 0, addr buf,  addr titl1, MB_OK
        invoke ExitProcess,0
        end _st
```

В програмі виконується витягання 2-го числа з регістру xmm. Функція wsprintf виконує перетворення числа в символ. А функція MessageBox виводить результат у спрощене віконце.

Результат виконання програми навдено на рис. 23.3.

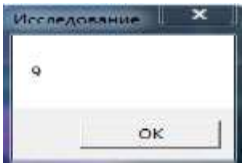


Рис. 23.3. Результат виконання програми

В лістингу 23.6 наведено програму порівняння на “більше”

Лістинг 23.6. Програма порівняння на “більше”:

```
        .686
        .xmm
        .model flat, stdcall ;
        option casemap:none ;
        include \masm32\include\windows.inc
        include \masm32\macros\macros.asm
        uselib kernel32, user32,masm32
.data
        mas1 dd 1, 10, 3, 4
        mas2 dd  5, 6, 7, 8
        res dd 4 dup(0),0
        buf dd 4 DUP(0),0
  _info db "Пример работы команды AVX vpcmpgtb",10,13,
      "(сравнение на больше).",10,13,
      "Первый массив: %d, %d, %d, %d",10,13,
      "Второй массив: %d, %d, %d, %d",10,13,
      "Результат работы (0 - больше, 255 - нет): ",10,13,
      4 dup("%d "),0
  _title db "Ольшевская Д.С. КИТ-22б",0
.code
_start:
        vmovups xmm1, mas1
        vmovups xmm2, mas2
      vpcmpgtb xmm0, xmm2, xmm1
        vunpcklpd xmm4, xmm4, xmm0
        vunpckhpd xmm4, xmm4, xmm5
        vunpckhpd xmm5, xmm5, xmm0
        vunpckhpd xmm5, xmm5, xmm6
        vcvtdq2pd xmm0, xmm4
        vcvtdq2pd xmm1, xmm5
        vunpcklpd xmm4, xmm4, xmm0
```

```
        vunpckhpd xmm4, xmm4, xmm6
        vunpckhpd xmm5, xmm5, xmm0
        vunpckhpd xmm5, xmm5, xmm4
      vcvttsd2si eax, xmm4
      vcvttsd2si ebx, xmm5
        mov dword ptr res, eax
        mov dword ptr res[4], ebx
        vunpcklpd xmm4, xmm4, xmm1
        vunpckhpd xmm4, xmm4, xmm6
        vunpckhpd xmm5, xmm5, xmm1
        vunpckhpd xmm5, xmm5, xmm4
      vcvttsd2si eax, xmm4
      vcvttsd2si ebx, xmm5
        mov dword ptr res[8], eax
        mov dword ptr res[12], ebx
invoke wsprintf, ADDR buf, ADDR _info, mas1, mas1[4], mas1[8], mas1[12], mas2, mas2[4],
mas2[8], mas2[12], res, res[4], res[8], res[12]
        invoke MessageBox, 0, addr buf, addr _title, 0
        invoke ExitProcess, 0
        end _start
```

Результат виконання програми навдено на рис. 23.4.

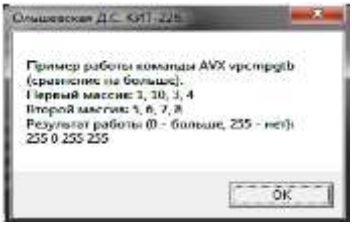


Рис. 23.4. Результат виконання програми

23.5. Команди AVX-512

AVX-512 розширює AVX до векторів завдовжки 512-біт за допомогою кодування з префіксом EVEX. Розширення AVX-512 вводить 32 векторних регістра (ZMM), кожен по 512 бітів, 8 регістрів масок, 512-розрядні упаковані формати для цілих та дробових чисел і операції над ними, управління режимами округлення (дозволяє перевизначити глобальні настройки), операції broadcast, подавлення помилок в операціях з дрібними числами, операції gather/scatter, швидкі математичні операції, компактне кодування великих зсувів. Регістри AVX (YMM0–YMM15) відображаються на молодші частини регістрів AVX-512 (ZMM0–ZMM15) по аналогії з SSE і AVX регістрами (AVX3.1).

23.6. Лабораторна робота
“AVX-команди”

Мета заняття:

набути практичних навичок складання, налагодження і виконання програм з використанням команд AVX, написаних мовою асемблера masm.

Постановка задачі

Згідно з останньою цифрою номера студента в групі вибрати свій варіант задачі та написати на асемблері за допомогою AVX-команд програму обчислення одного з виразів, ***виведенням на екран варіанту завдання, його вмісту, результату, прізвища та e-mail*** автора програми.

Завдання :

За допомогою команд AVX виконати рівняння і вивести результат через MessageBox. Числа задані в масивах.

1. ab – c;
2. a – 2c;
3. a/b – 3;
4. 4 + a/c;
5. a b +5;
6. a + 6/b;
7. a – bc;
8. a – b +5;
9. 9a + c;
10. a +10b

Приклад.

```
        .686 ; директива определение типа микропроцессора
        .model flat, stdcall
        .XMM
        option casemap:none ; отличие строчных и прописных букв
        include\masm32\include\windows.inc
        include\masm32\include\kernel32.inc
        include\masm32\include\fpu.inc
        include\masm32\include\user32.inc
        includelib\masm32\lib\kernel32.lib
        includelib\masm32\lib\fpu.lib
        includelib  \masm32\lib\user32.lib
.data
        mas1  DD 0.0, 2.,4.,6.,7.,8.,9.,2.              ; mas1 - 8 элементов (a)
        mas2  DD 1.,3.,5.,7.,24.,21.,23.,33.            ; mas2 - 8 элементов (b)
        mas3 DD 8.,9.,10.,11.,20.,30.,40.,50.           ; mas3 - 8 элементов (c)
        res  DD 8 DUP(0); резервирования памяти для результирующих 8х значений
s   DB "No AVX        ",0 ; для вывода в MessageBox
DB "              ",0 ; 8 строк под 8 результатов (в каждой строке 15 символов и '\0' в конце)
   DB "              ",0
   DB "              ",0
   DB "              ",0
   DB "              ",0
   DB "              ",0
   DB "              ",0
   Author  DB "Chelak V.V.",0 ; Разработал Челак Виктор Владимирович, КИТ23б
.code
_unzero:   ; меняет первый символ 00h в строке [ESI] на 20h (т.е. пробел)
        pusha ; поместить eax и в стек
        cld
@@:
        lodsb    ;загрузка строки: al <- [ESI]
        or al,al ; если получаем 00h, то выход из цикла (00h мы в любом случае обнаружим
так как в конце каждой строки стоит "0")
```

http://blogs.kpi.kharkov.ua/v2/asm/knigi/

```
        jnz @B          ; если не равно нулю (00h), то прыжок на @@
        dec esi         ; меняем указатель на 1 назад
        mov BYTE PTR [ESI],20h  ; записываем в строку вместо 00h - 20h
        popa    ; вернем в eax его старое значение из стека
        ret     ; возвращаемся в точку входа
_start:
        mov EAX,1
        cpuid                   ; проверка на наличие AVX: получаем EDX и ECX
        and  ECX,10000000h      ; bit28 - AVX
        or ECX,ECX              ;  1 - поддерживается, 0 - не поддерживается
        jz quit                 ;
vmovups  YMM0, mas1 ; пересилання 8   32-розрядних чисел масиву mas1
vmovups  YMM1, mas2 ; пересилання 8   32-розрядних чисел масиву mas2
vmovups  YMM2, mas3 ; пересилання 8   32-розрядних чисел масиву mas3
vmulps  YMM1, YMM1, YMM2            ; b*c  (произведение элементов двух массивов)
vsubps  YMM0, YMM0, YMM1            ; a-(b*c)
vmovups  res, YMM0          ; запам'ятовування результату
        mov  EBX, offset res    ; запам'ятовування адреси масиву результату
        xor  EDI,EDI            ; подготовка EDI
        mov ECX,8               ; заносим в счетчик количество результатов. Здесь их 8.
lp:
        fld DWORD PTR [EBX+EDI]         ; загружаем float в fpu
        lea ESI,[s+4*EDI]               ; каждая итерация - следующая строка
; форматируем в строку по адресу s+(16*индекс)
invoke FpuFLtoA,NULL,6,ESI,SRC1_FPU or SRC2_DIMM
     call  _unzero          ; убираем 00h в конце строки для вывода всех 8-х значений
        mov BYTE PTR [ESI+15],0Ah       ; ставим 0Ah (перевод строки)
        add EDI,4               ; переходим на след. елемент результата
     loop  lp
        mov BYTE PTR [ESI+15],00h ; меняем последний 0Ah обратно на 00h
quit:                           ; Метка для компьютера, не содержащий расширение AVX
;Выводим результаты, если AVX нет - выведет 7 пустых строк и строку "No AVX"
invoke MessageBox, NULL, offset s, offset Author, MB_ICONINFORMATION
        invoke ExitProcess,NULL
        end _start
```

Результат виконання програми навдено на рис. 23.5.

Рис. 23.5. Результат виконання програми

Контрольні запитання

1. Назовіть особливості набору AVX.
2. Для яких цілей використовується VEX префікс?
3. Який набір команд успадковують команди AVX?
4. Перерахуйте особливості використання AVX-інструкцій.

СПИСОК ЛІТЕРАТУРИ

1. Ирвин Кип. Язык ассемблера для процессоров Intel / Ирвин Кип. – 4-е изд.; пер. с англ. – М. : Издат. дом Вильямс, 2005. – 912 с.

2. Кравець В. О. Системне програмування. Програмування на мові асемблера: навч. посіб. / В.О.Кравець, О.М. Рисований. – Х. : НТУ «ХПІ», 2007. – 448 с.

3. Кравець В. О. Системне програмування. Асемблер під Win32 API : навч. посіб. / В. О. Кравець, О. М. Рисований. – Х. : НТУ «ХПІ», 2008. – 512 с.

4. Крупник А. Ассемблер. Самоучитель / А. Крупник. – С.Пб. : Питер, 2005. – 235 с.

5. Магда Ю. С. Ассемблер для процессоров Intel Pentium / Ю.С. Магда. – С.Пб. : Питер, 2006. – 410 с.

6. Пирогов В. Ю. Ассемблер и дизасемблирование / В.Ю. Пирогов. – С.Пб. : БХВ-Петербург, 2006. – 464 с.

7. Рисований О. М. Системне програмування. Основи асемблера під Win32 API: навч. посіб. / О.М. Рисований. – Х. : Сучасний Друк, 2008. – 552 с.

8. Рисований О. М. Цифрові пристрої і мікропроцесори. Архітектура і програмне забезпечення: навч. посіб. / О. М. Рисований, М. В. Грушенко. – Х. : ХУПС, 2005. – 384 с.

9. Рисований О. М. Системне програмування [Текст]: підручник для студентів напряму "Комп'ютерна інженерія" вищих навчальних закладів / О. М. Рисований. – Х. : НТУ «ХПІ», 2010. – 912 с.

10. Методичні вказівки до виконання та оформлення курсового проекту з курсу «Системне програмування» для студентів спеціальностей: 7.05010201 «Комп'ютерні системи та мережі», 7.05010202 «Системне програмування», 7.05010203 «Спеціалізовані комп'ютерні системи» / уклад. О. М. Рисований – Х. : НТУ «ХПІ», 2013. – 184 с.

11. Методичні вказівки до виконання лабораторних робіт з курсу «Системне програмування» для студентів спеціальностей: 7.05010201 «Комп'ютерні системи та мережі», 7.05010202 «Системне програмування», 7.05010203 «Спеціалізовані комп'ютерні системи» / уклад. О. М. Рисований. – Х. : НТУ «ХПІ», 2013. – 216 с.

12. Рогожина С. І. Системне програмування та операційні системи : навч. посіб. Ч. ІІ / С. І. Рогожина, О. М. Рисований, В. М. Федорченко. – Х. : ХУПС, 2007. – 200 с.

13. Щупак Ю. А. Win32 API. Эффективная разработка приложений / Ю.А. Щупак. – С.Пб. : Питер, 2007. – 572 с.

Навчальне видання

РИСОВАНИЙ Олександр Миколайович

СИСТЕМНЕ ПРОГРАМУВАННЯ
Том 1

Підручник

для студентів напрямку “Компютерна інженерія”
вищих навчальних закладів

Роботу до видання рекомендував *В.Д. Дмитрієнко*
Редактор *Л.Л. Яковлева*

План 2009 р., поз. №14 / 30-10

Підписано до друку 22.02.2010. Формат 60x84 1/16. Папір офісний. Riso-друк.
Гарнітура Times. Ум. друк. арк. 36. Обл.-вид. арк. 56,7.
Зам. № Ціна договірна.

Видавничий центр НТУ «ХПІ»
Свідоцтво про державну реєстрацію ДК № 3657 від 24.12.2009 р.
61002, Харків, вул. Фрунзе, 21

Відруковано в друкарні «Слово»
Адреса: 61002, Харків, вул. Лермонтовська, 27.