Зміст

1. Що таке Асемблер? 5

2. Що обов’язково повинен знати програміст на Асемблері на відміну від програміста мови високого рівня? 5

3. Для чого потрібен Асемблер? 6

4. Які джерела інформації для програмістів на Асемблері можна вважати найважливішими? 6

5. З чого складається базове середовище виконання програм з точки зору програміста на Асемблері? 6

6. Вісім регістрів загального призначення 6

7. 16 регістрів загального призначення 6

8. Які ресурси середовища IA-32 для виконання програм 7

9.Які ресурси середовища Intel64 для виконання програм 7

10. Режимы работы процессоров IA-32 (Режими роботи процесорів архітектури x86) 8

11) Який адресний простір архітектури IA-32 і чим він визначається? 9

12 Який адресний простір архітектури Intel64 і чим він визначається? 9

13. Позиційні системи числення. Поняття та основні формули 10

14.Способи переведення у іншу систему числення. Приклади 12

15. Способи представлення від'ємних чисел 13

16. Додатковий код. Приклади 14

17. Зміщений код. Приклади 15

18 Формати з плаваючою точкою. Стандарти 15

19 Стандартний двійковий 32-бітовий формат з плаваючою точкою 16

20 тандартний двійковий 64-бітовий формат з плаваючою точкою 16

21.Стандартний двійковий 80-бітовий формат з плаваючою точкою 16

22. Порівняння основних стандартних двійкових форматів з плаваючою точкою 17
23. Фізична память. Як узнати розрядність адрес? 17
24. Моделі пам'яті у процесорах архітектури x86 17
25. Сегментована пам'ять 18
26. Flat - модель пам'яті 18
27. Сторінкова пам'ять 18
28. Регістри процесора архітектури IA-32 19
29. Регістри процесора архітектури Intel64 20
30. Регістри загального призначення. 20
31. Сегментні регістри 21
32. Регістр EFALGS 21
33. Регістр ESP 22
34 Регістр EIP 22
35 Операнди команд та способи адресації операндів 23
36 Безпосередні операнди команд 24
37 Регістрові операнди команд 25
38 Способи вказування комірок пам'яті у якості операндів команд 25
39. Що таке опосередкована адресація стосовно вказування операнду команди? 26
40. Що таке база, індекс, зміщення, масштаб у операнді команди? 27
41Полное определение 27
42 PUSH 28
43 POP 29
44. Структура вихідного тексту програми на Асемблері 29

45.Директиви Асемблера. Найпопулярніші на вашу думку директиви 30

46.Директиви вказування типу процесора та моделі пам'яті 31

47. Директиви створення даних 33

48. Директива INVOKE. Приклади 33

49. Директива ALIGN. Приклади 33

50. Базові типи даних архітектури процесорів x86 33

51. Числові типи даних 34

52.Упаковані SIMD типи даних 35

53. Основні групи команд процесорів архітектури x86 36

54. Команди копіювання даних 36

55. Команди додавання цілих чисел 37

56. Команди віднімання цілих чисел 37

57. Команди множення цілих чисел 38

58. Команди ділення цілих чисел 39

59. Побітові логічні команди 39

60. Програмування читання та запису окремих бітів 40

61. Команди зсувів 40

62. Команди циклічного зсуву 41

63 Команды перехода 41

64 Реализация IF-THEN-ELSE 43

65, 66 Програмування циклів на асемблері? 44

67 Цикл з постумовою (рис. 2) 45

68. Програмування циклу зі збереженням біту переносу CF 46

69. Програмування циклу на основі команд LOOP 46

70. Програмування вкладених циклів 46
71. Команди обробки рядків даних 47
72. Команди STOS, STOSx 47
73. Команди MOVS,MOVSX. 48
74. Префікс повторення REP та його різновиди. 49
75.Що таке структурованість та модульність програм. Наведіть приклади. 50
76. Макроси. Приклади програмування 51
77. Процедури. Визначення, виклик. Приклади програмування 53
78. Способи передавання значень параметрів процедурам 53
79. Локальні дані процедур 54
80. Пролог та епілог процедури 55
81. Стековий кадр процедури 56
82. Написання модульних програм на Асемблері 57
83. Можливості використання мов високого рівня сумісно з Асемблером 57
84. Конвенції виклику процедур 58
85. Конвенція cdecl. Приклад програмування. 59
86. Конвенція stdcall. Приклад програмування. 61
87. Асемблерні вставки у мовах високого рівня. 62
88. Середовище x87 FPU 63
89. Команди x87 FPU. Приклад програми. 63
90. Стек даних x87 FPU. Приклад програми. 66
91. команди SIMD 67
92. Розширення MMX 69
93. Розширення SSE 70
94. Типи даних SSE 71

95. Векторны команди SSE 71

96 Горизонтальне додавання 72

97. Команди MOVUPS та MOVAPS. Приклад використання 73

98. Векторні команди множення SSE. Приклад використання 76

99. Векторні команди додавання SSE. Приклад використання 76

100. Команда SHUFPS та її застосування 77

101. Розширення AVX 77

102. Типи даних AVX 78

103. Команды AVX 78

104. Особенности команд AVX по отношению к SSE 79

105. Эволюция SIMD-расширений системы команд процессоров архитектуры x86 80

1. Що таке Асемблер?

Язик Ассе́мблер - машинно-ориентированный язык низкого уровня с командами, обычно соответствующими командам машины, который может обеспечить дополнительные возможности вроде макрокоманд.

Перевод программы на языке ассемблера в исполнимый машинный код (вычисление выражений, раскрытие макрокоманд, замена мнемоник собственно машинными кодами и символьных адресов на абсолютные илиотносительные адреса) производится *ассемблером* — программой-транслятором, которая и дала языку ассемблера его название.

Команды языка ассемблера один к одному соответствуют командам процессора. Фактически, они и представляют собой более удобную для человека символьную форму записи — *мнемокоды* — команд и их аргументов. При этом одной команде языка ассемблера может соответствовать несколько вариантов команд процесора.

2. Що обов’язково повинен знати програміст на Асемблері на відміну від програміста мови високого рівня?

-архітектуру процесора,систему комп'ютера

-організацію памяті

-режим роботи

- з'являється можливість хорошої оптимізації программи

- можливість читати інші програми не маючи їх лістинга

- та багато іншого

3. Для чого потрібен Асемблер?

а)Для вивчення і розуміння того, як працює процесор і програма.Assembler завжди буде потрібен, а може і ні.

б)Для написання компактних, швидкодіючих програм,драйверів.

в)Для написання програм, щоб в них виконувалось те, що потрібно програмісту

г)Для створення прогр. забезпечення для особливих процесорів,для яких недоступні мови високого рівня

4. Які джерела інформації для програмістів на Асемблері можна вважати найважливішими?

На мою думку, це документації від виробників процесорів,наприклад:64-ia-32-architectures-software-developer

5. З чого складається базове середовище виконання програм з точки зору програміста на Асемблері?

Кожна програма отримує набір ресурсів для виконання команд, зберігання даних ті отримання інформації про стани.Базове середовище надає ресурси як системни так і прикладним програмам.Також середовище має підтримувати базовий набір директив ассемблера.{конспект}

6. Вісім регістрів загального призначення

EAX, EBX, ECX, EDX, EBP,ESP,ESI,EDI

Сегментні регістри : 16-бітні CS DS ES SS FS GS

32 бітний регістр вказівника команд: EIP

X87 FPU 8 80-бітних регістрів R0..R7

MMX 8 64 бітних регістрів

XMM 8 128-бітних регістрів XMM0..XMM7

7. 16 регістрів загального призначення

RAX, RCX, RDX, RBX, RSP, RBP, RSI, RDI, R8 — R15

Сегментні регістри : 16-бітні CS DS ES SS FS GS

64 бітний регістр вказівника команд: RIP

X87 FPU 8 80-бітних регістрів R0..R7

MMX 8 64 бітних регістрів

XMM 8 128-бітних регістрів XMM0..XMM7

8. Які ресурси середовища IA-32 для виконання програм

Базовая среда :

- регистры общего назначения (32-битные, 8 шт.) :

  EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI

  Сегментные регистры (16-битные, 6шт.) :

  CS, DS, SS, ES, FS, GS

  - 32-битный регистр флажков EFIAGS
  - 32-битный регистр указателя команд EIP
  - X87 FPU
  - 8 шт. 80-битных регистров (R0-R7)
  - управляющие регистры
  - MMX (64-битные, 8шт.)
  - XMM (128-битные, 8 шт. : XMM0-XMM7);

Адресное пространство : программа может использовать до 4 Гб линейного пространства и до 64 Гб физического пространства

Регистр EIP непосредственно программисту не доступен – хранит адрес следующей команды и заполняется процессором автоматически.

Регистр флажков – набор битов, сигнализирующих о состоянии процессора и исполнении программ.

MMX – первая и не особо совершенная версия SIMD-команд (векторных)

SIMD – команды AVX (256-битные регистры) YMM0-YMM7

9.Які ресурси середовища Intel64 для виконання програм

Архитектура Intel® 64 в сочетании с соответствующим программным обеспечением поддерживает 64-разрядные вычисления во встраиваемых устройствах1. Архитектура Intel® 64 обеспечивает повышение производительности, позволяя системам использовать более 4 ГБ виртуальной и физической памяти.

Архитектура Intel® 64 поддерживает следующие возможности:

- 64-разрядное сплошное пространство виртуальных адресов
- 64-разрядные указатели
- 64-разрядные регистры общего назначения
- 64-разрядная поддержка вычислений с целыми числами
- До 1 ТБ адресного пространства платформы

10. Режимы работы процессоров IA-32 (Режими роботи процесорів архітектури x86)

1 Реальный режим

После инициализации (системного сброса) МП находится в реальном режиме (Real Mode). В реальном режиме МП работает в режиме эмуляции 8086 с возможностью использования 32-битных расширений.Механизм адресации, размеры памяти и обработка прерываний (с их последовательными ограничениями) МП 8086 полностью совпадают с аналогичными функциями других МП IA-32 в реальном режиме.

Имеется две фиксированные области в памяти, которые резервируются в режиме реальной адресации:

- область инициализации системы,
- область таблицы прерываний.

2 Режим системного управления

Режим системного управления (SMM – System Management Mode) предназначен для выполнения некоторых действий с возможностью их полной изоляции от прикладного программного обеспечения и даже от операционной системы.

Микропроцессор переходит в этот режим только аппаратно. Никакой программный способ не предусмотрен для перехода в этот режим. МП возвращается из режима системного управления в тот режим, при работе в котором был получен сигнал о переходе

3 Защищенный режим

В защищенном режиме программа оперирует с адресами, которые могут относиться к физически отсутствующим ячейкам памяти, поэтому такое адресное пространство называетсявиртуальным. Размер виртуального адресного пространства программы может превышать емкость физической памяти и достигать 64 Тбайт.

Основным режимом работы МП является защищенный режим.

Ключевые особенности защищенного режима:

- виртуальное адресное пространство,
- защита,
- многозадачность.

11) Який адресний простір архітектури IA-32 і чим він визначається?

Будь-яка програма в IA-32 може використовувати до 4Гб та до 64Гб фіз. простору

для процессоров i80286, i80386, i80486, Pentium — 2(в 32 степені)-1 (4 Гб)

для процессоров Pentium Pro, Pentium II, Pentium III, Pentium IV — 2(в 36 степені)-1 (64 Гб). определяется разрядностью счетчика команд микропроцесора.

12 Який адресний простір архітектури Intel64 і чим він визначається?

Адресное пространство

Хотя 64-битный процессор теоретически может адресовать 16 экзабайт памяти (2^64), Win64 в настоящий момент поддерживает 16 терабайт (2^44). Этому есть несколько причин. Текущие процессоры могут обеспечивать доступ лишь к 1 терабайту (2^40) физической памяти. Архитектура (но не аппаратная часть) может расширить это пространство до 4 петабайт (2^52). Однако в этом случае необходимо огромное количество памяти для страничных таблиц, отображающих память.

Помимо перечисленных ограничений, объем памяти, который доступен в той или иной версии 64-битной операционной системе Windows зависит также от коммерческих соображений компании Microsoft. Различные версии Windows имеют различные ограничения, представленные в таблице.

64-битные версии Winows	память
Windows XP Professional	128 Гбайт
Windows Server 2003, Standard	32 Гбайт
Windows Server 2003, Enterprise	1 Тбайт
Windows Server 2003, Datacenter	1 Тбайт
Windows Server 2008, Datacenter	2 Тбайт
Windows Server 2008, Enterprise	2 Тбайт
Windows Server 2008, Standard	32 Гбайт
Windows Server 2008, Web Server	32 Гбайт
Vista Home Basic	8 Гбайт
Vista Home Premium	16 Гбайт
Vista Business	128 Гбайт
Vista Enterprise	128 Гбайт
Vista Ultimate	128 Гбайт
Windows 7 Home Basic	8 Гбайт
Windows 7 Home Premium	16 Гбайт
Windows 7 Professional	192 Гбайт
Windows 7 Enterprise	192 Гбайт
Windows 7 Ultimate	192 Гбайт

13. Позиційні системи числення. Поняття та основні формули

Системою числення, або нумерацією**,** називається сукупність правил і знаків, за допомогою яких можна відобразити (кодувати) будь-яке невід'ємне число. У позиційних системах числення одна і та ж цифра (числовий знак) у записі числа набуває різних значень залежно від своєї позиції. Здебільшого вага кожної позиції кратна деякому натуральному числу b, b>1, яке називається основою системи числення. Позиційні системи числення розподіляються на два великих класи — однорідні (з рівною довжиною чисел і основою в вигляді натурального числа) і неоднорідні (з рівною та нерівною довжиною чисел і більш складною основою, ніж натуральні числа).

У позиційній системі числення з основою *b* число подають у вигляді лінійної комбінації степенів числа *b*:

$x = \sum_{k=0}^{n} a_k b^k$, де a_kі k— цілі, $0 \leq |a_k| < |b|$

Для запису чисел системи числення з основою до 36 включно у якості цифр використовуються арабські цифри (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) а потім букви латинського алфавіту (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z). При цьому, a = 10, b = 11 і т.д.

Позиційна система числення має такі властивості:

1. Основа системи числення у ній самій завжди записується як 10. Наприклад, у двійковій системі 10_2означає число 2_{10}.
2. Для запису числа *x* у системі числення з основою *b* потрібно $[log_b(x)] + 1$ цифр, де$[\cdot]$— ціла частина числа.
3. Порівняння чисел. Порівняємо два числа 516 і 561. Для цього зліва направо порівнюємо цифри, які стоять на однакових позиціях: 5 = 5 — результат порівняння не визначений; 1 < 6 — перше число менше незалежно від цифр, що залишились.
4. Додавання чисел. Для цього справа наліво додаємо цифри, що стоять на однакових позиціях таким чином можна додавати числа довільної довжини.

Джерела: http://uk.wikipedia.org/wiki/Позиційні_системи_числення
http://uk.wikipedia.org/wiki/Система_числення

14.Способи переведення у іншу систему числення. Приклади

Переведення довільної позиційної системи числення до десяткової

Якщо число у системі числення з основою *b* дорівнює

$$a_1\ a_2\ a_3 \ldots a_n,$$

то для переведення його до десяткової системи обчислюють наступну суму:

$$\sum_{i=1}^{n} a_i \cdot b^{n-i}$$

Для переведення із десяткової до довільної потрібно ділити число із залишком на основу системи числення допоки частка не стане меншою за основу.

44_{10} переведемо до двійкової системи

```
44 ділимо на 2; частка 22, залишок 0
22 ділимо на 2; частка 11, залишок 0
11 ділимо на 2; частка  5, залишок 1
 5 ділимо на 2; частка  2, залишок 1
 2 ділимо на 2; частка  1, залишок 0
```

Частка менша двох, ділення закінчено. Тепер записуємо останню частку від ділення і усі залишки, починаючи з останнього, зліва направо, отримаємо число 101100_2.

Переведення із двійкової у вісімкову і шістнадцяткову системи і навпаки. Для цього типу операцій існує спрощений алгоритм.

Для вісімкової — розбиваємо числа на триплети, перетворюючи триплети згідно з таблицею

```
000 0  100 4
001 1  101 5
010 2  110 6
011 3  111 7
```

Для шістнадцяткової — розбиваємо на квартети, перетворюючи згідно з таблицею

```
0000 0  0100 4  1000 8  1100 C
0001 1  0101 5  1001 9  1101 D
0010 2  0110 6  1010 A  1110 E
0011 3  0111 7  1011 B  1111 F
```

Перетворимо 101100_2

у вісімкову — 101 100 → 54_8

у шістнадцяткову — 0010 1100 → $2C_{16}$

Перетворимо до двійкової системи

$76_8 \rightarrow$ 111 110
$3E_{16} \rightarrow$ 0011 1110

Джерела: http://uk.wikipedia.org/wiki/Позиційні_системи_числення

15. Способи представлення від'ємних чисел

Для представлення ж від’ємних чисел слід виділити один біт для знаку. Як правило, це старший біт. Якщо один біт в числі виділяється під його знак, таке число називається знаковим. Як правило, 0 у старшому (крайньому зліва) біті відповідає додатнім числам, а 1 - від’ємним.

Представлення від’ємних чисел залежить від кількості байтів, яка відводиться на число. Для визначеності будемо розглядати однобайтові знакові числа.

Виділяють три основних способи представлення від’ємних чисел:

- *прямий код*, який утворюється з коду відповідного додатного числа шляхом встановлення знакового біта в 1;
- *обернений код*, який утворюється шляхом заміни значення кожного біта на протилежне (інверсне);
- *додатковий код*, який утворюється шляхом додавання 1 до молодшого біта оберненого коду.

Приклад.
Розглянемо число -3. Двійковим еквівалентом відповідного додатного числа 3 є 00000011**.**
Прямий код. Встановимо знаковий біт в 1 (нагадаємо, що 1 в старшому біті знакового числа сигналізує про його від’ємність). Всі інші біти залишаються без змін. В результаті вийде 10000011.
Обернений код. Замінимо кожний біт на протилежний (1 на 0; 0 на 1); результатом буде 11111100.
Додатковий код. Додамо 1 до оберненого коду; в результаті вийде 11111101. Зверніть увагу, що якщо розглядати послідовність 11111101 як беззнакове, а не як знакове число, вона інтерпретується як додатнє число 253.

Джерело:
http://idndist.lp.edu.ua/moodle/library/books/SHEMOTEHNIKA%20IDN/l01/ml1.html

16. Додатковий код. Приклади

Дополнительный код (англ. *two's complement*, иногда *twos-complement*) — наиболее распространённый способ представления отрицательных целых чисел в компьютерах. Он позволяет заменить операцию вычитания на операцию сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел, чем упрощает архитектуру ЭВМ.

Преобразование числа из прямого кода в дополнительный осуществляется по следующему алгоритму.

1. Если число, записанное в прямом коде, положительное, то к нему дописывается старший (знаковый) разряд, равный 0, и на этом преобразование заканчивается;
2. Если число, записанное в прямом коде, отрицательное, то все разряды числа инвертируются, а к результату прибавляется 1. К получившемуся числу дописывается старший (знаковый) разряд, равный 1.

Преимущества

- Общие инструкции (процессора) для сложения, вычитания и левого сдвига для знаковых и беззнаковых чисел (различия только во арифметических флагах которые нужно проверять для контроля переполнения в результате).
- Более удобная упаковка чисел в битовые поля.
- Отсутствие числа «минус ноль».

Недостатки

- Представление отрицательного числа не читается по обычным правилам, для его восприятия нужен особый навык или вычисления
- В некоторых представлениях (например, двоично-десятичный код) или их составных частях (например, мантисса числа с плавающей запятой) дополнительное кодирование неудобно
- Модуль наибольшего числа не равен модулю наименьшего числа. Например, для восьмибитного целого со знаком, максимальное число: $127_{10} = 01111111_2$, минимальное число: $-128_{10} = 10000000_2$. Соответственно, не для любого числа существует противоположное. Операция изменения знака может потребовать дополнительной проверки.
- Сравнение. В отличие от сложения, числа в дополнительном коде нельзя сравнивать как беззнаковые, или вычитать без расширения разрядности. Один из методов состоит в сравнении как беззнаковых исходных чисел с инвертированным знаковым битом.

Джерело: http://ru.wikipedia.org/wiki/ Дополнительный_код_(представление_числа)

17. Зміщений код. Приклади

Смещенный код, или код с избытком, используется в ЭВМ дял упрощения операций над порядками чисел с плавающей запятой. Он формируется следующим образом. Сначала выбирается длина разрядной сетки - n и записываются последовательно все возможные кодовые комбинации в обычной двоичной системе исчисления. Затем кодовая комбинация с единицей в старшем разряде, имеющая значение 2^{n-1}, выбирается для представления числа 0. Все последующие комбинации с единицей в старшем разряде будут представлять числа 1, 2, 3,… соответственно, а предыдущие комбинации в обратном порядке – числа -1, -2, -3, …

Так, числа 3 и -3 в формате со смещенным для 3-разрядной сетки будут иметь представление 111 и 001 соответственно, в формате со смещением для 4-разрядной сетки 1011 и 0101 соответственно. Нетрудно заметить, что различие между двоичным кодом с избытком и двоичным дополнительным кодом состоит в противоположности значений знаковых битов, а разность значений кодовых комбинаций в обычном двоичном коде и двоичном коде с избытком для 3- и 4-разрядных сеток равна соответственно 4 и 8.

18 Формати з плаваючою точкою. Стандарти

Существуют стандартные форматы для двоичных чисел с плавающей точкой

IEEE 754 – 1985,2008

$V= (-1)^S * 2^E * M$

S - знак

E - экспонента

M – мантисса (состоит из целой и дробной части)

M = 1*F, где F – дробная часть

E = e + сдвиг

Три основные формата :

- 32-битный (Single Precisiom) : S – 1 бит; e – 8 бит; F – 23 бита. Хитрость : целая часть в памяти не записывается, поскольку она в мантиссе всегда 1 («спрятанный бит»). Для этого формата сдвиг – 127. Диапазон чисел +- (2^{-126} …2^{+126})

- 64-битный (Double Precision) : S – 1 бит; e – 11 бит; F – 42 бита. Для этого формата сдвиг – 1023. Диапазон чисел +- (2^{-1022} …2^{+1022})

- 80-битный (Extended Precision) : S – 1 бит; e – 15 бит; F – 63 бита, j (M = j*F, где j - бит целой части, 0 или 1) – 1 бит; сдвиг – 16383. Диапазон чисел +- (2^{-16382} …2^{+16382}). Этот формат используется для промежуточных вычислений тригонометрических функций и самим процессором как внутренний.

В версии стандарта 2008 года появились новые разновидности :

- 16-битный (Half Precision) : S – 1 бит; e – 5 бит; F – 10 бит; сдвиг : 15. Этот формат центральный процессор не использует, но он популярен среди процессоров видеокарт.

Диапазон чисел : (-14…15)

19 Стандартний двійковий 32-бітовий формат з плаваючою точкою

Широко поширений комп'ютерний формат представлення дійсних чисел, що займає в пам'яті 32 біт а (4 байта). Як правило, під ним розуміють формат числа з плаваючою комою стандарту IEEE 754.
Числа одинарної точності з плаваючою комою забезпечують відносну точність 7-8 десяткових цифр в діапазоні від 10^{-38} до приблизно 10^{38}.
У сучасних комп'ютерах обчислення з числами з плаваючою комою підтримуються апаратним співпроцесором (FPU - Floating Point Unit). Однак у багатьох обчислювальних архітектурах немає апаратної підтримки чисел з плаваючою комою і тоді робота з ними здійснюється програмно.

Експонента записується у зміщеному коді: $e = E+127$.

Діапазон для експоненти E: від $Emin = -126$ до $Emax=+127$. Виходячи з цього, можна оцінити діапазон представлення чисел: ±(від 2-126 до 2+127).

Мантиса має вигляд $M = 1.F$, тобто ціла частина завжди дорівнює одиниці. Оскільки це заздалегідь відомо, то для заощадження пам’яті біт цілої частини в пам’ять не записується – цей біт зветься "схованим бітом".

1	8	23
S	**e**	**F**

http://uk.wikipedia.org/wiki/Число_одинарної_точності

20 тандартний двійковий 64-бітовий формат з плаваючою точкою

Число́ подвійної то́чності (**Double precision**, **Double**) — компьютерний формат представлення чисел, що займає у пам’яті дві послідовні комірки з точністю мантиси залежно від розрядності процесора (комп’ютерне слово; у випадку 32-бітного комп’ютеру – 64 біта або 8 байт). Як правило, під ним розуміють формат числа з плаваючою комою стандарту IEEE 754

Кодоване значення експоненти: $e = E + 1023$. Експонента E від $Emin = -1022$ до $Emax=+1023$, тому діапазон представлення чисел: ±(від 2-1022 до 2+1023). Мантиса записується так само, як і у 32-бітовому форматі: $M = 1.F$. Враховуючи схований біт цілої частини, мантиса загалом має 53 біти.

В компьютерах, которые имеют 64-разрядные с плавающей запятой арифметические единицы, большинство численных вычислений осуществляется в двойной точности с плавающей запятой, поскольку использование чисел одинарной точности обеспечивает почти такую же производительность.

1	11	52
S	**e**	**F**

http://ru.wikipedia.org/wiki/Число_двойной_точности

21.Стандартний двійковий 80-бітовий формат з плаваючою точкою

Кодоване значення експоненти: e = E + 16383.

Діапазон експоненти *E*: від *Emin* = -16382 до *Emax*=+16383.
Діапазон представлення (для нормалізованих чисел): ±(від 2-16382 до 2+16383).

У цьому форматі мантиса записується інакше, аніж для 32-, та 64-бітових форматів, і має наступний вигляд: *M = j.F,* де *j* – це біт цілої частини, який може бути 0 або 1. Цей біт записується у пам'ять, як і решта бітів. Це зроблено тому, що у розширеному форматі передбачено виконання операцій також і над ненормалізованими числами, у яких старші біти мантиси можуть бути нулями.

1	15	1	63
S	**e**	**j**	**F**

Джерело – лаб 3 Сис прог.

22. Порівняння основних стандартних двійкових форматів з плаваючою точкою

Критерій порівняння	32-бітовий формат	64-бітовий формат	80-бітовий формат
К-ть бітів для представлення	**32**	**64**	**80**
Діапазон представлення	±(від 2^-126 до 2^+127)	±(від 2^-1022 до 2^+1023)	±(від 2^-16382 до 2^+16383).
Формат запису мантиси	*M* = 1.*F*		*M = j.F, j = 0,1*
Производительность	**Одинаковая**		**Меньшая производительность**

Висновок: В ЕОМ з 64-бітовою архітектурою краще використовувати 64-бітове представлення чисел з плаваючою комою, так як швидкодія однакова з 32-бітовим форматом, а діапазон представлення є більшим, що забезпечує більшу точність.
Якщо необхідно мати ще більшу точність обчислення, доцільно використовувати 80-бітовий формат представлення чисел з плаваючою комою, але він займатиме більше пам'яті та займатиме більше ресурсів.

23. Фізична память. Як узнати розрядність адрес?

Дійсна або фізична пам'ять – пам'ять, спосіб адресації до якої відповідає фізичному розташуванню її даних.
Дізнаемось розрядність адрес
N – ємність пам'яті(у бітах)
M – комірка пам'яті(у бітах)
K – розрядність
K = $log_2(N/M)$,N/M округлюється в більшу сторону.

24. Моделі пам'яті у процесорах архітектури x86

Найпростішу організацією має плоска модель пам'яті (flat): уся пам'ять представляється єдиною лінійною послідовністю байт, де зберігаються і дані, і коди програми. Відповідальність за коректне використання пам'яті лягає цілком на прикладного програміста - він повинний піклуватися про те, щоб дані не "затерли" коди або на них не "наїхав" зростаючий стек. Щоб одержати плоску модель пам'яті, по суті, досить зробити так, щоб усі сегментні регістри вказували на ту саму область пам'яті.

Протилежністю плоскої моделі є сегментована захищена модель - основна модель пам'яті, яка використовується в захищеному режимі. Кожній програмі в будь-який момент часу надаються кодовий сегмент, сегмент стека і до чотирьох сегментів даних. Сегменти спеціальним образом вибираються з таблиць, підготовлених операційною системою. Некоректні звертання до пам'яті блокуються системою захисту, що керує операційною системою.

Проміжне положення між названими моделями займає модель пам'яті реального режиму. Тут також пам'ять організується у виді сегментів, але незалежності і захищеності сегментів немає.

25. Сегментована пам'ять

Сегментована пам'ять зображає логічний адресний простір як сукупність незалежних блоків змінної довжини, які називають сегментами. Кожен сегмент містить дані одного призначення. Кожен сегмент має ім'я і довжину. Логічна адреса складається з номера сегмента і зсуву всередині сегмента; з такими адресами працює прикладна програма.

Компілятори створюють окремі сегменти для різних даних програми (сегмент коду, сегмент даних, сегмент стека). Під час завантаження програми у пам'ять створюють таблицю дескрипторів сегментів процесу, кожний елемент якої відповідає одному сегменту і складається із базової адреси, значення межі та прав доступу.

Під час формування адреси її сегментна частина вказує на відповідний елемент таблиці дескрипторів сегментів процесу.

Якщо зсув більший, ніж задане значення межі (або якщо права доступу процесу не відповідають правам, заданим для сегмента), то апаратне забезпечення генерує помилку. Коли ж усе гаразд, сума бази і зсуву в разі чистої сегментації дасть у результаті фізичну адресу в основній пам'яті.

26. Flat - модель пам'яті

Модель пам’яті flat (або плоска модель пам'яті) має найпростішу організацію.

При використанні моделі програма оперує єдиним безперервним адресним простором - лінійним адресним простором. У ньому містяться і код, і стек, і дані програми, адресовані зміщенням в межах від 0 до 232-1. Такий 32-бітовий зсув називається лінійною адресою.

Така модель є не захищеною і уся відповідальність за коректне використання пам'яті лягає цілком на прикладного програміста - він повинний піклуватися про те, щоб дані не "затерли" коди або на них не "наїхав" зростаючий стек. Щоб одержати плоску модель пам'яті, по суті, досить зробити так, щоб усі сегментні регістри вказували на ту саму область пам'яті.

27. Сторінкова пам'ять

Головна ідея — розподіл пам'яті блоками фіксованої довжини, що називають сторінками. Фізичну пам'ять розбивають на блоки фіксованої довжини — фрейми, або сторінкові блоки. Логічну пам'ять, у свою чергу, розбивають на блоки такої самої довжини — сторінки. Коли процес починає виконуватися, його сторінки завантажуються в доступні фрейми фізичної пам'яті з диска або іншого носія. Сторінкова організація пам'яті має апаратну підтримку. Кожна адреса, яку генерує процесор, ділиться на дві частини: номер сторінки і зсув сторінки. Номер сторінки використовують як індекс у таблиці сторінок.

Для кожного процесу створюють його власну таблицю сторінок. Коли процес починає своє виконання, ОС розраховує його розмір у сторінках і кількість фреймів у фізичній пам'яті. Кожну сторінку завантажують у відповідний фрейм, після чого його номер записують у таблицю сторінок процесу.

На логічному рівні для процесу вся пам'ять зображується неперервним блоком і належить тільки цьому процесові, а фізично вона розосереджена по адресному простору мікросхеми пам'яті, чергуючись із пам'яттю інших процесів. Процес не може звернутися до пам'яті, адреса якої не вказана в його таблиці сторінок (так реалізований захист пам'яті).

28. Регістри процесора архітектури IA-32

Регістри загального призначення:

- AX EAX (Accumulator) Регістр для виконання арифметичних операцій.
  BX EBX (Base) Використовується для адресації за базою
- CX ECX (Counter) Лічильник кроків циклу
  DX EDX (Data) Зберігає “довгі” результати операцій (які не вміщуються в AX)
- SI ESI (Source Index) Індекс поточного елемента області, з якої пересилаються дані
- DI EDI (Destination Index) Індекс поточного елемента області, куди пересилаються дані
- BP EBP (Base Pointer) Адреса бази стека
- SP ESP (Stack Pointer) Адреса вершини стека

При цьому вони поділяються на:

32 біта EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D-R15D

16 біт AX, BX, CX, DX, DI, SI, BP, SP, R8W-R15W (молодші половинки відповідних регістрів)

8 біт AL, BL, CL, DL AH, BH, CH, DH или DIL, SIL, BPL, SPL, R8L-R15L(старший/молодший байт 16 бітових половинок)

Регістр – вказівник команд:

- IP EIP (Instruction Pointer) Містить адресу команди, яку процесор виконує на поточному кроці своєї роботи

Сегментні регістри:

- CS — Сегмент коду. Використовується для вибірки команд програми;
- DS — Сегмент даних. Використовується за замовчуванням для доступу до даних;
- ES — Додатковий сегмент. Є отримувачем даних в командах обробки рядків;
- SS — Сегмент стеку. Використовується для розміщення програмного стеку;
- FS — Додатковий сегментний регістр. Спеціального призначення не має. З'явився в процесорі 80386;
- GS — Аналогічно попередньому, але в нових процесорах с 64-бітною архітектурою має особливий статус: може використовуватись для швидкого переключення контекстів.

FLAGS - це регістр стану процесора (PSW).

- CF - прапор переносу зі старшого розряду
- OF - прапор переповнення
- AF - прапор додатк. переносу (для десяткового перенесення).
- SF - прапор знака.
- ZF - прапор нуля.
- PF - прапор парності.

- PF = 1, якщо кількість одиниць біт парне, прапори IF, DF, TF - прапори управління.
- IF - прапор дозволу переривання (може встановлюватися програмно)
- DF - прапор напрямки при обробці строкових операндів.
- TF - прапор трасування (мікропроцесор виконує 1-у команду і зупиняється).

Цілочисельні регістри MMX-розширення MMXO, MMXl, MMX2, MMX3, MMX4, MMX5, MMX6, MMX7

Регістри MMX-розширення з плаваючою точкою XMMO, XMM1, XMM2, XMM3,, XMM4, XMM5, XMMб, XMM7

Регістри співпроцесора ST (0), ST (1), ST (2), ST (3), ST (4), ST (5), ST (6), ST (7) призначені для написання програм, що використовують тип даних з плаваючою точкою

Регістри розширення x87 FPU: R0, R1, R2, R3, R4, R5, R6, R7

29. Регістри процесора архітектури Intel64

16 целочисленных 64-битных регистра общего назначения (RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8 — R15),
8 80-битных регистров с плавающей точкой (ST0 — ST7),
8 64-битных регистров Multimedia Extensions (MM0 — MM7, имеют общее пространство с регистрами ST0 — ST7),
16 128-битных регистров SSE (XMM0 — XMM15),
64-битный указатель RIP и 64-битный регистр флагов RFLAGS.

Джерело: http://habrahabr.ru/company/intel/blog/93831/

30. Регістри загального призначення.

Особливість цих регістрів полягає в тому, що можлива адресація їх як одного цілого чи слова як однобайтової частини. Лівий байт є старшою частиною (high), а правий - молодшою частиною (low).

1. Регістр AX. Регістр AX є основним суматором і застосовується для всіх операцій уведення-висновку, деяких операцій над рядками і деяких арифметичних операцій.

2. Регістр BX. Регістр BX є базовим регістром. Це єдиний регістр загального призначення, що може використовуватися в якості "індексу" для розширеної адресації. Інше загальне застосування його - обчислення.

3. Регістр CX. Регістр CX є лічильником. Він необхідний для керування числом повторень циклів і для операцій зрушення уліво або вправо.

4. Регістр DX. Регістр DX є регістром даних. Він застосовується для деяких операцій уведення/висновку і тих операцій множення і розподілу над великими числами, які використовують реєстрову пару DX і AX.

Будь-які регістри загального призначення можуть використовуватися для додаваня і вирахування як 8-ми, так і 16-ти бітових значень.

Джерело: http://bibliofond.ru/view.aspx?id=445652

31. Сегментні регістри

У програмній моделі мікропроцесора є шість сегментних регістрів: cs, ss, ds, es, gs, fs. Їх існування обумовлено специфікою організації і використовування оперативної пам'яті мікропроцесорами Intel. Вона полягає в тому, що мікропроцесор апаратний підтримує структурну організацію програми у вигляді трьох частин, званих сегментами. Для того, щоб вказати на сегменти, до яких програма має доступ в конкретний момент часу, і призначені сегментні регістри. В цих регістрах містяться адреси пам'яті з яких починаються відповідні сегменти. Логіка обробки машинної команди побудована так, що при вибірці команди, доступі до даних програми або до стека неявно використовуються адреси в цілком певних сегментних регістрах. Мікропроцесор підтримує наступні типи сегментів:

1. Сегмент коду. Містить команди програми. Для доступу до цього сегменту служить регістр cs (code segment register) — сегментний регістр коду. Він містить адресу сегменту з машинними командами, до якого має доступ мікропроцесор.

2. Сегмент даних. Містить оброблювані програмою дані. Для доступу до цього сегменту служить регістр ds (data segment register) — сегментний регістр даних, який береже адресу сегменту даних поточної програми.

3. Сегмент стека. Цей сегмент є областю пам'яті, званою стеком. Роботу із стеком мікропроцесор організовує за наступним принципом: останній записаний в цю область елемент вибирається першим. Для доступу до цього сегменту служить регістр ss (stack segment register) — сегментний регістр стека, що містить адресу сегменту стека.

Джерело: http://kit.znu.edu.ua/Meth/arch1.pdf

32. Регістр EFALGS

EFLAGS — регістр прапорів. Розрядність eflags/flags — 32/16 біт. Окремі біти даного регістра мають певне функціональне призначення і називаються прапорами. Молодша частина цього регістра повністю аналогічна регістру flags для i8086.

Виходячи з особливостей використовування, прапори регістра eflags/flags можна розділити на три групи:

8 прапорів полягання

Ці прапори можуть змінюватися після виконання машинних команд. Прапори полягання регістра eflags відображають особливості результату виконання арифметичних або логічних операцій. Це дає можливість аналізувати полягання обчислювального процесу і реагувати на нього за допомогою команд умовних переходів і викликів підпрограм.

1 прапор управління

Позначається df (Directory Flag). Він знаходиться в 10-у біті регістра eflags і використовується ланцюжковимі командами.

5 системних прапорів

Керують уведення-виведенням, маскованими перериваннями, відладкою, перемиканням між задачами і віртуальним режимом 8086.

Джерело: http://kit.znu.edu.ua/Meth/arch1.pdf

33. Регістр ESP

Регістр ESP / SP завжди вказує на вершину стека , тобто містить зсув , по якому в стек був занесений останній елемент . Команди роботи зі стеком неявно змінюють цей регістр так , щоб він вказував завжди на останній записаний в стек елемент . Якщо стек порожній , то значення esp одно адресою останнього байта сегмента , виділеного під стек . При занесенні елемента в стек процесор зменшує значення регістра esp , а потім записує елемент за адресою нової вершини. При отриманні даних з стека процесор копіює елемент, розташований за адресою вершини , а потім збільшує значення регістра покажчика стека esp . Таким чином, виходить , що стек росте вниз , у бік зменшення адрес.

Джерело: http://otvet.mail.ru/question/69008232

34 Регістр EIP

EIP

Начиная с процессора 80386 была введена 32-разрядная версия регистра-указателя — EIP (англ. Extended Instruction Pointer). Принцип работы EIP в целом схож с работой регистра IP. Основная разница состоит в том, что в защищённом режиме, в отличие от реального режима, регистр CS является селектором.

IP (англ. Instruction Pointer) — регистр, содержащий адрес-смещение следующей команды, подлежащей исполнению, относительно кодового сегмента CS в процессорах семейства x86.

Регистр IP связан с CS в виде CS:IP, где CS является текущим кодовым сегментом, а IP — текущим смещением относительно этого сегмента.

Принцип работы

Например, CS содержит значение 2CB5[0]H, в регистре IP хранится смещение123H.

Адрес следующей инструкции, подлежащей исполнению, вычисляется путем суммирования адреса в CS (сегменте кода) со смещением в регистре IP:

2CB50H + 123H = 2CC73H – адрес следующей инструкции для исполнения

При выполнении текущей инструкции процессор автоматически изменяет значение в регистре IP

http://ru.wikipedia.org/wiki/Регистр_процессора

35 Операнди команд та способи адресації операндів

Команда, як послідовність деяких дій над даними, виконується по тактам (мікропрограма команди). І, у залежності від формату опе¬ран¬дів, кількість тактів може бути різна. Процесор працює з командами та даними, де дані є не просто масивами бітових рядків, а операндами команд.

Спосіб адресації Запис команди Ефективна адреса Розгорнутий запис Застосування

Абсолютна (пряма)

ADD R1,@#1000 M[1000] R1:=R1+M[1000] Коли відома абсолютна (пряма) адреса операнда

Безпосередня

ADD R1,#4 - R1:=R1+4 Один з операндів — константа (арифм. операції, перевірки умов)

Регістрова

ADD R1,R2 R2 R1:=R1+R2 Усі операнди в регістрах

Непряма регістрова ADD R1,(R2) M[R2] R1:=R1+M[R2] Доступ до даних за попередньо обчисленою адресою, визначення адреси, на яку посилається вказівник (адреса вказівника — в R2)

Непряма ADD R1,@(R2) M[M[R2]] R1:=R1+M[M[R2]] Робота з вказівниками: якщо в R2 — адреса вказівника p, то ефективна адреса — це значення *p

За зсувом (базова, індексна)

ADD R1,30(R2) M[R2+30] R1:=R1+M[R2+30] Один з основных способів. Застосовується для організації переміщуваних програм (фіксація «початку відліку (база)» в R2), для роботи з масивами (адреса початку — в R2, змінною зсуву отримаємо доступ до різних комірок масиву)

Масштабування (індексна регістрова непряма) ADD R1,(R2)[R3] M[x*R3+R2] R1:=R1+M[x*R3+R2]

x — різне в залежності від типу операндів Робота з масивами

Непряма регістрова з автоінкрементом ADD R1,(R2)+ M[R2] R1:=R1+M[R2];

R2:=R2+1 Робота з масивами у циклах. R2 початково вказує на початок масиву, кожна нова ітерація супроводжується позиціонуванням на наступний елемент

Непряма регістрова з автодекрементом ADD R1,-(R2) M[R2-1] R2:=R2-1;

R1:=R1+M[R2] Аналогічно попередньому способу

Ri — регістр з порядковим номером i.

M[j] — комірка пам'яті з абсолютною адресою j.

M[Ri]- комірка пам'яті з адресою, яка розташована в регістрі Ri

http://satr.unicyb.kiev.ua/apx_eom/arhit_eom_lab_6.htm

http://uk.wikipedia.org/wiki/Способи_адресації_пам'яті

36 Безпосередні операнди команд

 Постійні чи безпосередні операнди — число, рядок, чи ім'я вираження, що мають деяке фіксоване значення. Ім'я не повинне бути переміщуваним, тобто залежати від адреси завантаження програми в пам'ять. Наприклад, воно може бути визначено операторами equ чи =.

```
num     equ     5

imd = num-2

mov     al,num  ;еквівалентно mov al,5 ;5 тут безпосередній операнд

add     [si],imd        ; imd=3 - безпосередній операнд

mov     al,5            ;5 - безпосередній операнд
```

http://www.unicyb.kiev.ua/~boiko/ass/2.htm

37 Регістрові операнди команд

Регістр команд (англ. Instruction Register - IR) — складова частинапроцесора, що відповідає за приймання та збереження двійкового кодукоманди впродовж машинного циклу, поки вона не буде виконана чи дешифрована[1]. У простих мікроконтролерах необхідності в цьому функціональному блоці немає, проте в складних обчислювальних пристроях, що містять конвеєр команд, інструкції можуть перебувати на різному етапі виконання, тому виникає потреба їх зберігати.

Регістр команди, яка в даний момент інтерпретується. Містить спочатку саму команду для подальшого розбору, зокрема виділення операндів та приведення їх до потрібного для операції робочого вигляду, бо на рівні команди операнди можуть представлятися:

- явно у команді (літералом);
- номером/ім'ям регістра;
- адресою у ОП;
- косвеною адресою, тобто адресою поля ОП чи регістра, в якому зберігається адреса значення операнда.

http://uk.wikipedia.org/wiki/Регістр_команд

Джерело http://satr.unicyb.kiev.ua/apx_eom/arhit_eom_lab_6.htm

38 Способи вказування комірок пам'яті у якості операндів команд

В фон-нейманівських машинах кожна комірка пам'яті має власну адресу й проблема визначення місця розташування потрібних даних зводиться до визначення цієї адреси. В перших ЕОМ адреса або номер комірки необхідно було вказувати явно, що виявилося незручним. Це спричинило до розробки нових методів.

Пік винахідництва в цій галузі припав на час панування CISC-архітектур «регістр-пам'ять», які дозволяли безпосередньо використовувати в якості одного з операндів комірку пам'яті. RISC-архітектури типу «регістр-регістр», в яких доступ до пам'яті регламентується значно більш жорстко, мають у порівнянні з CISC, дуже скромний набір методів адресації, і у найрадикальніших представниках цієї ідеології зведений до єдиного.

CISC-архітектура

Абсолютна (пряма) – застосовуємо Коли відома абсолютна (пряма) адреса операнда

Безпосередня – застосовуємо Коли Один з операндів — константа (арифм. операції, перевірки умов)

Регістрова – засосовуємо коли Усі операнди в регістрах

Непряма регістрова – застосовуємо коли Доступ до даних за попередньо обчисленою адресою, визначення адреси, на яку посилається вказівник (адреса вказівника — в R2)

Непряма – застосовуємо коли Робота з вказівниками: якщо в R2 — адреса вказівника p, то ефективна адреса — це значення *p

За зсувом (базова, індексна) – Застосовується для організації переміщуваних програм, для роботи з масивами

Масштабування (індексна регістрова непряма) – застосовується для роботи з масивами

Непряма регістрова з автоінкрементом -- Робота з масивами у циклах. R2 початково вказує на початок масиву, кожна нова ітерація супроводжується позиціонуванням на наступний елемент

Непряма регістрова з автодекрементом – аналогічно до попереднього

RISC-архітектура

Спосіб адресації

За зсувом

Регістрово-індексна

Регістрова з поновленням регістра

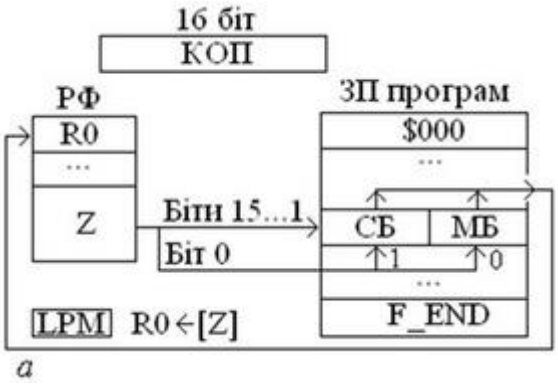
Регістрово-індексна з поновленням регістра

Регістрова (похідна від адресації за зсувом)

Абсолютна (похідна від адресації за зсувом)

39. Що таке опосередкована адресація стосовно вказування операнду команди?

Такой тип адресации, когда ячейка памяти задается ее именем, плюс некоторая константа, называется непосредственной (прямой) адресацией. Хотя непосредственная адресация - это хороший метод, она не отличается достаточной гибкостью, поскольку обращение выполняется каждый раз по одному и тому же адресу памяти.



приклад_опосередкованої_адресації_до_констант_пам'яті_програм

40. Що таке база, індекс, зміщення, масштаб у операнді команди?

Операнды — это объекты, над которыми или при помощи которых выполняются действия, задаваемые инструкциями или директивами.

4. Байт масштаб – индекс – база (байт sib). Используется для расширения возможностей адресации операндов. Байт sib состоит из трех полей:

1) поля масштаба ss. В этом поле размещается масштабный множитель для индексного компонента index, занимающего следующие 3 бита байта sib;

2) поля index. Используется для хранения номера индексного регистра, который применяется для вычисления эффективного адреса операнда;

3) поля base. Используется для хранения номера базового регистра, который также применяется для вычисления эффективного адреса операнда.

5. Поле смещения в команде.

8-, 16– или 32-разрядное целое число со знаком, представляющее собой, полностью или частично (с учетом вышеприведенных рассуждений), значение эффективного адреса операнда.

http://www.e-reading.ws/chapter.php/99319/50/Cvetkova_-_Informatika_i_informacionnye_tehnologii.html

41Полное определение

Стек – это область памяти, специально выделяемая для временного хранения данных программы. Важность стека определяется тем, что для него в структуре программы предусмотрен отдельный сегмент. Для работы со стеком предназначены три регистра:

1) ss – сегментный регистр стека;

2) sp/esp – регистр указателя стека;

3) bp/ebp – регистр указателя базы кадра стека.

Размер стека зависит от режима работы микропроцессора и ограничивается 64 Кбайтами (или 4 Гбайтами в защищенном режиме).

2) по мере записи данных в стек последний растет в сторону младших адресов. Эта особенность заложена в алгоритм команд работы со стеком;

В общем случае стек организован так, как показано на рисунке 23.

Рис. 23. Концептуальная схема организации стека

Для работы со стеком предназначены регистры SS, ESP/SP и EBP/BP. Эти регистры используются комплексно, и каждый из них имеет свое функциональное назначение.

Регистр ESP/SP всегда указывает на вершину стека, т. е. содержит смещение, по которому в стек был занесен последний элемент. Команды работы со стеком неявно изменяют этот регистр так, чтобы он указывал всегда на последний записанный в стек элемент. Если стек пуст, то значение esp равно адресу последнего байта сегмента, выделенного под стек. При занесении элемента в стек процессор уменьшает значение регистра esp, а затем записывает элемент по адресу новой вершины. При извлечении данных из стека процессор копирует элемент, расположенный по адресу вершины, а затем увеличивает значение регистра указателя стека esp. Таким образом, получается, что стек растет вниз, в сторону уменьшения адресов.

42 PUSH

push источник – запись значения *источник* в вершину стека.

Интерес представляет алгоритм работы этой команды, который включает следующие действия (рис. 24):

1) (sp) = (sp) – 2; значение sp уменьшается на 2;

2) значение из источника записывается по адресу, указываемому парой ss: sp.

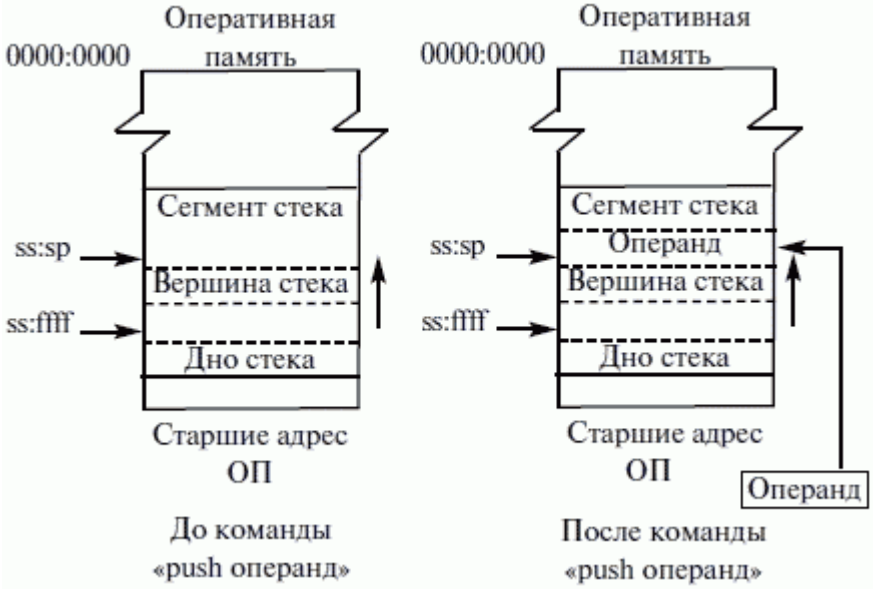


Рис. 24. Принцип работы команды push

43 POP

pop назначение – запись значения из вершины стека по месту, указанному операндом *назначение.* Значение при этом «снимается» с вершины стека. Алгоритм работы команды pop обратен алгоритму команды push (рис. 25):

1) запись содержимого вершины стека по месту, указанному операндом *назначение;*

2) (sp) = (sp) + 2; увеличение значения sp.

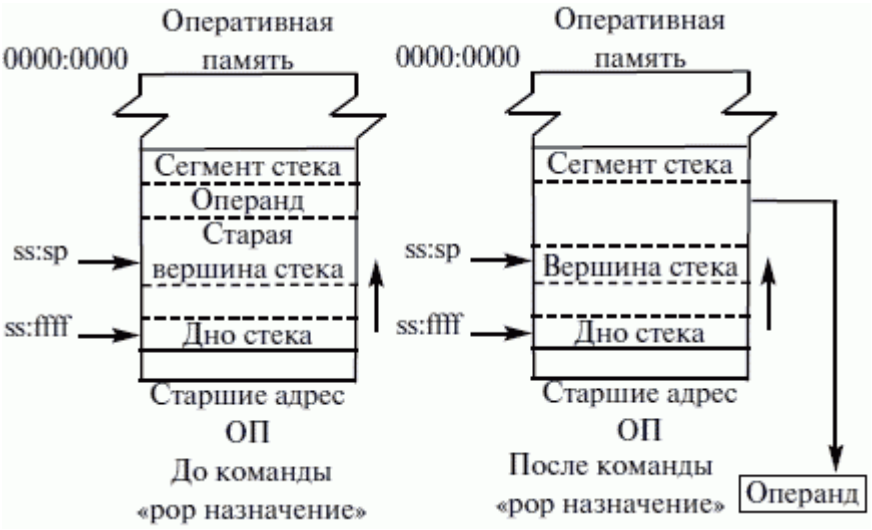


Рис. 25. Принцип работы команды pop

44. Структура вихідного тексту програми на Асемблері

Программа на асссемблере может состоять из нескольких частей, называемых модулями, в каждом из которых могут быть определены один или несколько сегментов данных, стека и кода.

Любая законченная программа на ассемблере должна включать один главный, или основной (main), модуль, с которого начинается ее выполнение. Основной модуль может содержать программные сегменты, сегменты данных и стека, объявленные при помощи упрощенных директив. Кроме того, перед объявлением сегментов нужно указать модель памяти при помощи директивы .MODEL.

СТРУКТУРА ПРОГРАММЫ:

- ДЕКЛАРАТИВНАЯ часть - описание символических имен данных и адресов, которые используют в программе, а также директивы выделения памяти для переменных и указания значений констант (посредством директивы Ассемблера);

- блок ИНИЦИАЛИЗАЦИИ - настройка портов и блоков периферийных функций на необходимые режимы работы, инициализация стека (посредством команд);

- блок РЕАЛИЗАЦИИ алгоритмов и функций управления (посредством команд).

Каждая команда представляет собой строку такой конструкции:

[МЕТКА:] мнемокод операции операнд(ы) [; комментарии]

[ ] - поле может отсутствовать.

МЕТКА - символическое имя ячейки памяти, начиная с которой размещается в памяти данная команда.

В качестве ОПЕРАНДОВ могут использоваться числа (адреса и данные), зарезервированные и определенные символические имена.

Для указания системы счисления, в которой задается число, используют буквенные индексы после самого числа: **B** - двоичная, **Q** - восьмеричная, **D** или ничего - десятичная, **H** - шестнадцатеричная.

КОММЕНТАРИИ - любые символы.

Для разработки программы для модели flat перед директивой .model flat следует разместить одну из директив: .386, .486, .586 или .686. Желательно указывать тот тип процессора, который используется в машине.

45.Директиви Асемблера. Найпопулярніші на вашу думку директиви

Директивы управления листингом являются:
.LIST – включить в листинг все строки исходного текста программы;
.NOLIST – исключить из листинга исходный текст;
.SUMS – включить в листинг таблицу символов;
.NOSUMS – исключить из листинга таблицу символов и т. д.

Директивы указания типа процессора :

.8086 – разрешает только команды микропроцессора 8086 (работает по умолчанию)

.286 и .286P – разрешает команды 286-го микропроцессора (P – защищённый режим)

.386 и .386P – разрешает команды 386-го микропроцессора

.486 и .486P – разрешает команды 486-го микропроцессора

.586 и .586P – разрешает команды P5 (Pentium)

.686 и .686P – разрешены команды P6 (Pentium Pro, Pentium II)

Директивы секционирования программы:

.code – открывает описание сегмента кодов команд;
.data – начинает описание сегмента данных;
.stack – описывает сегмент стека, в качестве аргумента этой директивы

Директивы .macro, .endmacro (.endm), определяющие начало и конец макроса соответственно.

Синтаксис написания:

.macro {имя макроса}

.endm

Директивы резервирования данных:

db (define byte) – резервирует область для байтовых данных;
dw (define word) – резервирует область для двухбайтовых данных;
dd (define double word) – резервирует область для четырехбайтовых данных;
df (define float) – резервирует область для шестибайтовых данных;
dq (define quakeук word) – резервирует область для восьмибайтовых данных;
dt (define ten bytes) – резервирует область для десятибайтовых данных

46.Директиви вказування типу процесора та моделі пам'яті

- .MODEL (.model) модель_памяти [,соглашение_о_вызовах] [,тип_ОС] [,параметр_стека] — определяет модель памяти, используемую программой. Директива должна находиться перед любой из директив объявления сегментов. Она связывает определенным образом различные сегменты программы, определяемые ее параметрами tiny, small, compact, medium, large, huge или flat. Параметр модель_памяти является обязательным.

Модель памяти	Адресация кода	Адресация данных	Операционная система	Чередование кода и данных
TINY	NEAR	NEAR	MS-DOS	Допустимо
SMALL	NEAR	NEAR	MS-DOS, Windows	Нет
MEDIUM	FAR	NEAR	MS-DOS, Windows	Нет
COMPACT	NEAR	FAR	MS-DOS, Windows	Нет
LARGE	FAR	FAR	MS-DOS, Windows	Нет
HUGE	FAR	FAR	MS-DOS, Windows	Нет
FLAT	NEAR	NEAR	Windows NT, Windows 2000, Windows XP, Windows 2003	Допустимо

Все семь моделей памяти поддерживаются всеми компиляторами MASM, начиная с версии 6.1.

- **small** поддерживает один сегмент кода и один сегмент данных. Данные и код при использовании этой модели адресуются как near (ближние).

- **large** поддерживает несколько сегментов кода и несколько сегментов данных. По умолчанию все ссылки на код и данные считаются дальними (far).

- **medium** поддерживает несколько сегментов программного кода и один сегмент данных, при этом все ссылки в сегментах программного кода по умолчанию считаются дальними (far), а ссылки в сегменте данных — ближними (near).

- **compact** поддерживает несколько сегментов данных, в которых используется дальняя адресация данных (far), и один сегмент кода с ближней адресацией (near).

- **huge** практически эквивалентна модели памяти large.

- **tiny** работает только в 16-разрядных приложениях MS-DOS. В этой модели все данные и код располагаются в одном физическом сегменте модель

- **flat** предполагает несегментированную конфигурацию программы и используется только в 32-разрядных операционных системах. Эта модель подобна модели tiny в том смысле, что данные и код размещены в одном сегменте, только 32-разрядном. Хочу напомнить, что многие примеры из этой книги разработаны именно для модели flat.

Для разработки программы для модели flat перед директивой .model flat следует разместить одну из директив: .386, .486, .586 или .686. Желательно указывать тот тип процессора, который используется в машине, хотя на машинах с Intel Pentium можно указывать директивы .386 и .486. Операционная система автоматически инициализирует сегментные регистры при загрузке программы, поэтому модифицировать их нужно, только если необходимо смешивать в одной программе 16- и 32-разрядный код

.8086 – разрешает только команды микропроцессора 8086 (работает по умолчанию)

.186 – разрешает команды 186-го микропроцессора

.286 и .286P – разрешает команды 286-го микропроцессора (P – защищённый режим)

.386 и .386P – разрешает команды 386-го микропроцессора

.486 и .486P – разрешает команды 486-го микропроцессора

.586 и .586P – разрешает команды P5 (Pentium)

.686 и .686P – разрешены команды P6 (Pentium Pro, Pentium II)

.8087 – разрешены команды арифметического процессора 8087

.287 – разрешены команды арифметического процессора 287

.387 – разрешены команды арифметического процессора 387

.MMX – разрешены команды расширения MMX

47. Директиви створення даних

Для створення таких даних у асемблері є директиви:
DB (*define byte*) – визначає перемінну розміром у 1 байт;
DW (*define word*) – визначає перемінну розміром у 2 байти (слово);
DD (*define double word*) – визначає перемінну розміром у 4 байти (подвійне слово);
DQ (*define quad word*) – визначає перемінну розміром у 8 байтів (квадрослово);
DT (*define ten bytes*) – визначає перемінну розміром у 10 байтів.
Усі ці директиви можуть бути використані як для створення простих перемінних та констант, так і для створення масивів. Масиви для рядків символів зазвичай створюють директивою DB. Формат запису директиви для створення перемінної або константи такий:
<ім'я> D* <операнд> [, <операнд>]

де позначка D* означає одну з директив DB, DW, DD, DF, DQ або DT.

Джерело: Перша лаба

48. Директива INVOKE. Приклади

INVOKE фактически создает код для вызова функции. Необходимо определить функцию ранее PROC, EXTERNDEF, TYPEDEF или оператор PROTO. Далее представлен синтаксис для ВЫЗОВА:

```
   INVOKE expression [,arguments]
```

Потому что ассемблер знает ожидается в отношении аргументов и соглашение о вызовах функции, его можно принимать аргументы, переданные в операторе ВЫЗОВА и помещают их в стек в правильном порядке, вызова функции по имени требуемую функцию и очистку стека после этого (если требуется использовать соглашение о вызовах).

Джерело: http://support.microsoft.com/kb/73407/ru

49. Директива ALIGN. Приклади

ALIGN – директива вирівнювання данних у пам*яті, тобто розташування об*єктів по адресам кратним байту, слову

Приклад

Align 4

Var1 db 255 ; address 0040400h;

Джерело: Конспект лекцій

50. Базові типи даних архітектури процесорів x86

Фундаментальные типы данных

Фундаментальными типами для архитектуры IA-32 являются:

байт (byte)	-	8 бит
слово (word)	-	16 бит
двойное слово (doubleword)	-	32 бит
четверное слово (quadword)	-	64 бит

Джерело : http://dims.karelia.ru/x86/env_5.shtml

51. Числові типи даних

Целые типы данных

Целое число может считаться знаковым или беззнаковым. Фундаментальный тип кодирует беззнаковое число, если его значение интерпретируется как целое число от 0 до 2^n-1, где n - разрядность типа.

Байт

Символ со знаком -128...+127
(signed char)

символ без знака 0...255
(unsigned char)

Слово
(word)

короткое со знаком -32768...+32767
(signed short)

короткое без знака 0...65535
(unsigned short)

Двойное слово

целое со знаком -2147483648...+2147483647
(signed int)

целое без знака 0...4294967295
(unsigned int)

Вещественные типы данны

Старший разряд двоичного представления вещественного числа всегда кодирует знак числа. Остальная часть разбивается на две части: экспонента и мантисса. Вещественное число вычисляется как: $(-1)^S*2^E*M$, где S - знаковый бит числа, E - экспонента, M - мантисса.

Тип: вещественное ординарной точности
(single precision) - 32 бит

Диапазон : $1.18*10^{-38}$... $3.40*10^{38}$

Тип: вещественное двойной точности
(double precision) - 64 бит

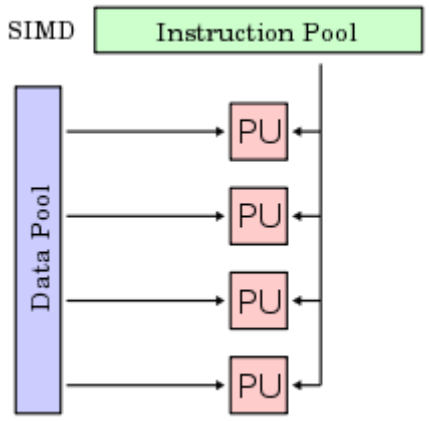
Диапазон : $2.23*10^{-308}$... $1.79*10^{308}$

Тип: вещественное расширенной точности
(extended precision) - 80 бит

Диапазон : $3.37*10^{-4932}$... $1.18*10^{4392}$

52.Упаковані SIMD типи даних

SIMD (англ. *single instruction, multiple data* — **одиночний потік команд, множинний потік даних**) - це елемент класифікації згідно з таксономією Флінна для паралельних процесорів, де до багатьох елементів даних виконується одна або однакові команди. SIMD - це одна з головних умов, котра гарантує можливість паралельного виконання алгоритмів.



Використання

При обробці мультимедійної інформації, наприклад накладання фільтрів, необхідно виконувати однакові дії над кожним пікселем зображення — саме тому дана архітектура дуже широко використовується при обробці мультимедійної інформації.

- Векторний процесор — процесор, в якому операндами деяких команд можуть слугувати впорядковані масиви даних — вектори.
- GPU - орієнтація відеокарт — це обробка мультимедійної відео інформації, яка найкраще відповідає представленню SIMD архітектури.
- Архітектура MAJC — багатоядерний та багатопоточний мікропроцесор, від компанії Sun Microsystems, котрий був орієнтований на обробку мультимедійної інформації у мережі.
- Векторні розширення центрального процесора X86 та x86_64: в центральні процесори додавалися набори інструкцій для роботи з мультимедійними даними:

    1. **MMX**(Multimedia Extensions)- комерційна назва додаткового набору інструкцій, що виконують характерні для процесів кодування / декодування потокових аудіо / відео даних дії за одну машинну інструкцію.
    2. **3DNow!** — розширення для набору інструкцій платформи X86, яку розробила компанія Advanced Micro Devices(AMD), що дозволяє виконувати прості векторні операції,.
    3. **SSE**(Streaming SIMD Extensions)— набір інструкцій, розроблених Intel, як відповідь на аналогічний набір інструкцій 3DNow! відAMD
    4. **AVX**(Advanced Vector Extensions)— розширення системи команд X86 для мікропроцесорів Intel та AMD, запропоноване компанією Intel у березні 2008.

- Векторні розширення NEON ARM процесорів — набір з 64- та 128-бітових SIMD інструкцій, що надає стандартизоване прискорення для засобів медіа та сигнальної обробки прикладних програм.

Реалізація блоків SIMD виконується розпаралелюванням обчислювального процесу між даними. Тобто коли через один блок даних проходить по черзі багато потоків даних.

Джерело http://uk.wikipedia.org/wiki/SIMD

53. Основні групи команд процесорів архітектури x86

Систему команд процессора 8086 образуют 113 базовых команд, многие из которых допускают использование разнообразных режимов адресации.

По функциональному назначению выделяют спедующие группы команд:

- команды передачи данных
Dst, src, mem, reg, sreg, ac, data, mov.
- команды арифметических операций
Add, adc, inc,- команды прибавления
Sub, sbb, dec, neg - команды вычитания
Cmp – команда сравнения
Mul, imul – команды умножения
Div, idiv – команды деления
Cbw, cwd – команды преобразования
- команды логических операций и сдвигов
AND, OR, XOR и TEST - команды логических операций
Rol , ror , rcl , rcr, shl , shr , sal , sar – команды сдвигов
- команды передачи управления
Команды передачи управления процессора 8086 подразделяются на команды безусловных переходов, условных переходов, вызовов, возвратов, управления циклами и команды прерываний.
- цепочечные команды
-команды управления микропроцессором - Команды данной группы обеспечивают программное управление различными функциями процессора. Они делятся на две подгруппы: команды установки флагов и команды синхронизации.

ДЖЕРЕЛО http://www.emanual.ru/download/www.eManual.ru_1255.html

54. Команди копіювання даних

MOV

Ця команда використовується для копіювання значення з одного місця в інше. Це 'місце' може бути регістр, комірка пам'яті або безпосереднє значення (тільки як вихідне значення). Синтаксис команди: mov приймач, джерело. Можна копіювати значення з одного регістра в інший. Розмір джерела і приймача повинні бути однаковими.

ДЖЕРЕЛО http://www.unicyb.kiev.ua/~boiko/ass/3.htm

XCHG Обмін данімі між операндами

Команда xchg пересилає значення першого операнда в другій, а іншого - в перший. Як будь-який операнд можна указуваті регістр (окрім сегментного) або елемент пам'яті, проте не допускається візначаті обідва операнді одночасно як елементі пам'яті. Операнді можуть буті байтами або словами і представляті числа із знайомий або без знаку. Команда не вплівае на прапорі процесора.

Приклад

```
mov Ax,off01h
mov Si,1000h
xchg Ax,si ;AX=01000h, Si=ff01h
```

ДЖЕРЕЛО http://www.znannya.org/?view=asm-teams30

55. Команди додавання цілих чисел

Нехай потрібно додати 128-бітові числа. Представимо кожне 128-бітове число чотирма 32-бітовими групами: Ai та Bi. Спочатку треба додати молодші групи бітів: A0+B0 . Результат додавання буде 33-бітовим – 32 молодші біти та перенос. Молодші 32 біти вже є бітами результату, а перенос треба додати при обчисленні суми наступних груп: A1+B1+перенос. І так далі. Біт переносу результату додавання старших груп A3+B3 потрібно зберегти, якщо у цьому є потреба.
При реалізації додавання у програмах на асемблері використовується команда ADD для додавання молодших груп бітів A0+B0, і команда ADC – для додавання у вигляді: Ai+Bi+перенос. При виконанні додавання значення переносу автоматично записується у біт CF регістру EFLAGS.

56. Команди віднімання цілих чисел

Віднімання виконується так само, як і додавання – послідовно від молодших груп бітів. При відніманні наступних груп бітів потрібно враховувати результат віднімання попередньої групи.
Позичання виникає при відніманні більшого числа від меншого. Позичання

означає вимогу розповсюдити групу одиниць від старшого до поточного біту.

Цю групу одиниць достатньо закодувати одним бітом, який зветься бітом позичання (borrow).

При реалізації у програмах на асемблері для віднімання молодших груп бітів (A0 – B0) використовується команда SUB, а віднімання усіх наступних груп бітів (Ai – Bi) робиться командою SBB (subtract with borrow). Біт позичання записується у біт CF регістру EFLAGS.

57. Команди множення цілих чисел

(источник - http://looch-disasm.narod.ru/refe01.htm)

mul op2 – беззнаковое умножение.

imul op2 – знаковое умножение.

Алгоритм умножения

Умноження двух 96-битових операндов A та B.

Результат будет состоять из 192 бітов.

Алгоритм выполнения умножения групами по 32 бита состоит в умножении одного 96-битного операнда (A) на группы 32 битов другого операнда (Bi).

Алгоритм для данного случая будет выполнен за 3 шага — будет 3 группы по 64 бита.

Потом эти частичные произведения будут прибавлены друг к другу по мере их расположения в разрядной сетке.

Или

A * B = A * B0 + A * B1 + A * B2

58. Команди ділення цілих чисел

(источник - http://looch-disasm.narod.ru/refe01.htm)

div op2 – беззнаковое целочисленное деление,

idiv op2 – знаковое целочисленное деление.

Алгоритм деления

Рассмотрим выполнение операции D = A / B , где A и B цели двоичные числа без знака .

Одним из самых простых алгоритмов является деление " в столбик ".

Алгоритм деления n - битового числа A 4-битное B

можно записать так :

1 . Установить i = n - 4.

2 . Взять четыре старшие биты числа A. Отметить это как R = { an -1 , an -2 , an -3 , an -4 }

3 . Если R больше или равна B ,

то : i - и цифра (бит) результата di равна 1 , вычитаем R = R - B

иначе : i - и цифра результата di равна 0

4 . Уменьшить i на единицу. Если i меньше нуля , то конец работы .

5 . Умножить R на два. Такое умножение означает смещение битов на одну позицию

влево. Добавить в R в младший разряд i -й бит числа A , т.е. ai .

6 . Переход на п. 2 .

59. Побітові логічні команди

(источник - http://looch-disasm.narod.ru/refe01.htm)

ADD – сложение.

OR – логическое включающее «ИЛИ»

ADC – сложение с переносом.

SBB – вычитание с заёмом

AND – логическое «И».

SUB - вычитание

XOR – логическое исключающее «ИЛИ»

CMP – сравнение двух операндов.

TEST – логическое сравнение.

NEG – получение дополнительного кода (изменение знака).

NOT – инвертирование

60. Програмування читання та запису окремих бітів

(источник - http://looch-disasm.narod.ru/refe01.htm)

BSF – сганирование бита вперед

BSR – сканирование бита назад

BT – проверка бита

BTS – проверка бита и установка в единицу

BTC – проверка бита и сброс в нуль

61. Команди зсувів

(источник - http://looch-disasm.narod.ru/refe01.htm)

SHL – обычный сдвиг влево (по направлению к старшему биту)

SHR – обычный сдвиг вправо (по направлению к младшему биту)

SAL – арифмитический сдвиг влево (по направлению к старшему биту)

SAR – арифметический сдвиг вправо (старший бит размножается — если в старшем бите 0, то сдвинутые биты забиваются нулями, если 1 - единицами)

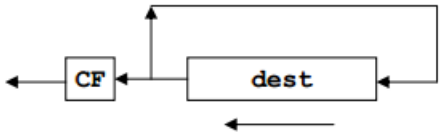
SAL и SHL работают одинаково.

62. Команди циклічного зсуву

Команди циклічного зсуву

Команда ROL. Циклічний зсув бітів вліво (у напрямку старшого біту). Старші біти записуються на молодші позиції.

```
rol dest, count
```



Наприклад:

```
mov ax, 7215h        ; 0111 0010 0001 0101
rol ax, 4            ; 0010 0001 0101 0111
```

Команда ROR. Циклічний зсув бітів вправо (у напрямку молодшого біту). Старші біти записуються на молодші позиції.

```
ror dest, count
```

Наприклад:

```
mov ax, 7215h        ; 0111 0010 0001 0101
ror ax, 4            ; 0101 0111 0010 0001
```

63 Команды перехода

1) **jmp** метка – безусловный переход на метку. Например: **jmp** @exit; Используется очень часто. По сути выполняет то же самое действие, что и goto в Паскале

2) **jcc** метка (**jcc** – одна из команд в следующей таблице) – условный переход. Переход выполняется, если соответствующее условие выполнено. Реально каждое условие является каким-либо состоянием флагов. Обычно перед командой условного перехода выполняется команда cmp, хотя это совсем не обязательно. В приведённом ниже примере мы хотим найти модуль числа eax:

```
   cmp eax, 0;
   jge @NoNEG; {если больше или равно, то знак инвертировать не надо}
   neg eax;
@NoNEG:
```

Если вы разберётесь с флагами, и поймёте, какие флаги реально изменяет

каждая команда, то сможете писать условные переходы, например, после команды dec. Это иногда позволяет получить некоторый выигрыш в производительности.Команды условного перехода:

Код команды	Реальное условие	Условие для CMP	Код команды	Реальное условие	Условие для CMP
JA	CF=0 и ZF=0	если выше	**JG**	ZF=0 и SF=OF	если больше
JAE **JNC**	CF=0	если выше или равно если нет переноса	**JGE**	SF=OF	если больше или равно
JB **JC**	CF=1	если ниже если перенос	**JL**	SF<>OF	если меньше
JBE	CF=1 | ZF=1	если ниже или равно	**JLE**	ZF=1 | SF<>OF	если меньше или равно
JE **JZ**	ZF=1	если равно если ноль	**JNE** **JNZ**	ZF=0	если не равно если не ноль
JO	OF=1	если есть переполнение	**JNO**	OF=0	если нет переполнения
JS	SF=1	если есть знак	**JNS**	SF=0	если нет знака
JP	PF=1	если есть четность	**JNP**	PF=0	если нет четности

3) **jecxz** метка – переход, если ecx=0

4) **loop** метка – цикл. Уменьшает регистр ecx на 1 и выполняет переход типа short (не дальше, чем на 127 байт) на метку, если ecx<>0. Например, в следующем фрагменте команда **add** выполнится 10 раз:

```
    mov ecx, 10;
@loop_start:
    add eax, ecx;
    loop @loop_start;
```

Команда **loop** эквивалентна паре команд dec ecx; jnz метка, но не меняет флаги

5) **loope|loopz** метка – цикл пока равно|ноль
loopne|loopnz метка – цикл пока не_равно|не_ноль
Все перечисленные команды уменьшают ecx на 1, после чего выполняют переход типа short, если ecx<>0, и если выполняется условие. Для команд **loope|loopz** условие – равенство ZF=1, для loopne|loopnz – равенство ZF=0. Сами команды**loopcc** не изменяют значений флагов, так что ZF должен быть определён предыдущей командой. Например, следующий фрагмент копирует строку из esi в edi пока не кончится строка (ecx=0), или пока не встретится символ с ASCII-кодом 13 (конец строки):

```
    mov ecx, lenght_string;
@loop0:
    lodsb;
    stosb;
    cmp al, 13;
@loopnz loop0;
```

64 Реализация IF-THEN-ELSE

Конструкция IF-THEN-ELSE в ассемблере реализуется с помощью условных переходов

Код команды	Реальное условие	Условие для CMP	Код команды	Реальное условие	Условие для CMP
JA	CF=0 и ZF=0	если выше	**JG**	ZF=0 и SF=OF	если больше
JAE JNC	CF=0	если выше или равно если нет переноса	**JGE**	SF=OF	если больше или равно
JB JC	CF=1	если ниже если перенос	**JL**	SF<>OF	если меньше
JBE	CF=1 | ZF=1	если ниже или равно	**JLE**	ZF=1 | SF<>OF	если меньше или равно
JE JZ	ZF=1	если равно если ноль	**JNE JNZ**	ZF=0	если не равно если не ноль
JO	OF=1	если есть переполнение	**JNO**	OF=0	если нет переполнения
JS	SF=1	если есть знак	**JNS**	SF=0	если нет знака
JP	PF=1	если есть четность	**JNP**	PF=0	если нет четности

К примеру мы должны сравнить ax и bx

Конструкция в ПАСКАЛЕ

```
If(условие1)then
        Какие то действия 1

goto
        Какие то действия 2
```

Соответствие в Ассемблере

```
cmp ax,bx;

jz/js/jo/… @label1; – не условие1 (потому что по условию один мы перейдем, а нам это не нужно)

        ;Какие то действия 1

@label1

        ;Какие то действия 2
```

Если мы хотим реализовать классическую IF-THEN-ELSE типа

```
If(условие)then begin
        Какие то действия 1
```

```
end else begin
        Какие то действия 2

end
```

Вот лично мое решение

```
…

Jnz @if ; безусловный переход – как к примеру goto в Паскале

@then

        ;Какие то действия1

Jnz @ifend; безусловный переход – как к примеру goto в Паскале

@if

Cmp ax,bx

Jz/jo/… @then; - условие (то же что и в примере на паскале)

        ;Какие то действия 2

@ifend

…
```

65, 66 Програмування циклів на асемблері?

Нехай потрібно щось виконати 10 разів. Це "щось" будемо називати "тілом циклу". Тіло циклу може містити деяку множину рядків програмного коду на асемблері. Розглянемо варіант алгоритму циклу з постумовою (рис. 2)

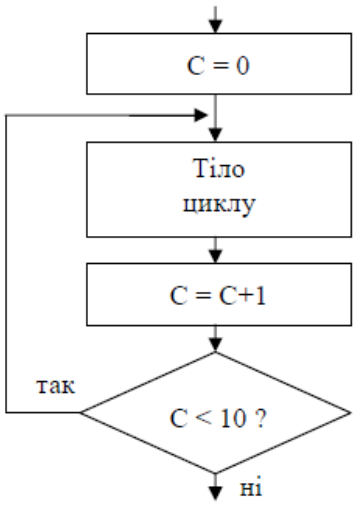


Рис. 2. Блок-схема алгоритму циклу з постумовою

На асемблері такий цикл можна запрограмувати наступним чином:
mov ecx, 0 ; обнулюємо лічильник – регістр ECX
cycle:
. . . ; тіло циклу
inc ecx ; збільшуємо лічильник на 1
cmp ecx, 10 ; порівнюємо лічильник з 10

jl cycle ; якщо лічильник менше – перехід на мітку cycle
Необхідно відзначити, що такий варіант організації циклу є неприйнятний для послідовної обробки груп бітів з розповсюдженням переносу, оскільки команда CMP сама записує біт CF.
Інший варіант циклу – у лічильник спочатку записується потрібна кількість повторень, а на кожній ітерації значення лічильника зменшується на одиницю:
mov ecx, 10 ; у регістр ECX записуємо кількість повторень
cycle:
. . . ; тіло циклу
dec ecx ; зменшуємо лічильник на 1
jnz cycle ; якщо лічильник не 0, то перехід на мітку cycle

Джерело: лекції Порєва

67 Цикл з постумовою (рис. 2)

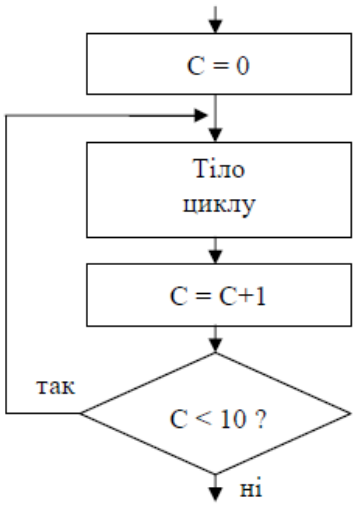


Рис. 2. Блок-схема алгоритму циклу з постумовою

На асемблері такий цикл можна запрограмувати наступним чином:
mov ecx, 0 ; обнулюємо лічильник – регістр ECX
cycle:
. . . ; тіло циклу
inc ecx ; збільшуємо лічильник на 1
cmp ecx, 10 ; порівнюємо лічильник з 10
jl cycle ; якщо лічильник менше – перехід на мітку cycle

Джерело: лекції Порєва

68. Програмування циклу зі збереженням біту переносу CF

mov ecx, 10 ; у регістр ECX записуємо кількість повторень

cycle:

. . . ; тіло циклу

dec ecx ; зменшуємо лічильник на 1

jnz cycle ; якщо лічильник не 0, то перехід на мітку cycle

Такий варіант циклу може бути використаний для наших цілей, оскільки ані

команда DEC, ані JNZ не змінюють біт CF регістру EFLAGS.

69. Програмування циклу на основі команд LOOP

Для организации цикла предназначена команда LOOP. У этой команды один операнд — имя метки, на которую осуществляется переход. В качестве счётчика цикла используется регистр CX.
Команда LOOPвыполняет декремент CX, а затем проверяет его значение. Если содержимое CX не равно нулю, то осуществляется переход на метку, иначе управление переходит к следующей после LOOP команде.
Содержимое CX интерпретируется командой как число без знака. В CX нужно помещать число, равное требуемому количеству повторений цикла. Понятно, что максимально может быть 65535 повторений. Ещё одно ограничение связано с дальность перехода. Метка должна находиться в диапазоне -127…+128 байт от команды LOOP (если это не так, FASM сообщит об ошибке).

http://asmworld.ru/uchebnyj-kurs/013-cikly-i-komanda-loop/

70. Програмування вкладених циклів

Алгоритм со структурой вложенных циклов(циклический алгоритм) – это алгоритм, в котором внутри одного цикла называемого *внешним* (глобальным) расположен другой цикл называемый *внутренним* (локальным). Для того, чтобы организовать внутренний цикл можно использовать любой из способов реализации внешнего цикла, но с одной особенностью. Для того, скажем, чтобы организовать внутренний цикл на основе команды Loop необходимо сохранить значение CX перед началом вложенного цикла и восстановить после его завершения (перед командой LOOP внешнего цикла). Сохранить значение можно в другой регистр, во временную переменную или в стек, то есть нужно слідить, тобы вложеный цикл не менял данные, необходимые внешнему циклу для корректной работы.

http://asmworld.ru/uchebnyj-kurs/013-cikly-i-komanda-loop/

71. Команди обробки рядків даних

Кроме привычного всем понятия массивов в ассемблере существует структура называемая цепочкой. Цепочка - непрерывная последовательность байт, слов или двойных слов, обрабатываемая как единое целое. Основное отличие цепочек от массивов состоит в способе доступа к элементам: для массивов - произвольный доступ, для цепочек - только последовательный (от начала цепочки к концу или от конца к началу). Цепочечные команды - команды для обработки цепочек. Особенностью всех цепочечных команд (кроме обработки очередного элемента цепочки) является автоматическое продвижение к следующему элементу цепочки.

Цепочечные команды:

Название	Команды	Действие
пересылка цепочки	`movs <адр. приемника>, <адр. источника>` `movsb, movsw, mowsd`	копирует один элемент цепочки из операнда источника в операнд приемник
сравнение цепочек	`cmps <адр. приемника>, <адр. источника>` `cmpsb, cmpsw, cmpsd`	сравнивает элементы цепочек из операнда источника и операнда приемника
сканирование цепочки	`scas <адр. приемника>` `scasb, scasw, scasd`	сканирует цепочку приёмник на присутствие некоторого элемента (задаётся в регистре аккумуляторе)
загрузка элемента из цепочки	`lods <адр. источника>` `lodsb, lodsw, lodsd`	загрузить элемент из цепочки источника в регистр аккумулятор
сохранение элемента в цепочке	`stos <адр. приемника>` `stosb, stosw, stosd`	восстановить элемент из регистра аккумулятора в цепочку
получение элемента цепочки из порта ввода/вывода	`ins <адр. приемника>, <номер порта>` `insb, insw, insd`	загрузить элемент в цепочку приемник из указанного порта ввода/вывода
вывод элементов цепочки в порт ввода/вывода	`outs <номер порта>, <адр. источника>` `outbs, outws, outds`	переслать элемент из цепочки источника в указанный порт ввода/вывода

http://archkomp.narod.ru/lab09.htm

72. Команди STOS, STOSx

Эти команды сохраняют элемент из регистров al/ax/eax в ячейку памяти. Перед командой stos можно указать префикс повторения rep, в этом случае появляется возможность работы с блоками памяти, заполняя их значениями в соответствии с

содержимым регистра *ecx/cx*. Команда STOS обеспечивает повторение пока не достигнут конец цепочки, т.е. пока содержимое регистра CX не достигнет нуля. После каждого повторения производится декремент CX, поэтому его необходимо инициализировать на требуемое число. Тип записываемой информации определяется по мнемонике команды: символ B (STOSB) означает

запись байта, W (STOSW)- запись слова, D (STOSD) – запись двойного слова.

http://www.kolasc.net.ru/cdo/programmes/assembler/stos.html

73. Команди MOVS,MOVSX.

MOVS

MOVS (MOVS,MOVSB,MOVSW) Пересылка данных из строки в строку

Команды предназначены для операций над строками (строкой называется последовательность байтов или слов памяти с любым содержимым). Они пересылают по одному элементу строки, который может быть байтом или словом. Первый операнд (приемник) адресуется через ES:DI, второй (источник) - через DS:SI. Операцию пересылки можно условно изобразить следующим образом:

(DS:SI) -> (ES:DI)

После каждой операции пересылки регистры SI и DI получают положительное (если флаг DF=0) или отрицательное (если флаг DF=1) приращение. Величина приращения составляет 1 или 2 в зависимости от размера пересылаемых элементов. Вариант команды movs имеет формат:

movs строка_1, строка_2

*MOVSX (**MOVe and Sign eXtension**)*

Команда преобразует операнд со знаком в эквивалентный ему операнд со знаком большей размерности. Для этого содержимое операнда источник начиная с младших разрядов записывается в операнд приемник. Старшие биты операнда приемник заполняются значением знакового разряда операнда источник.

Синтаксис	**`MOVSX приемник, источник`**

Машинный код	`OF BE/r — MOVSX r16,r/m8`
	`OF BE/r — MOVSX r32,r/m8`
	`OF BF/r — MOVSX r32,r/m16`

Алгоритм работы:
считать содержимое источника;
записать содержимое операнда источника в операнд приемник, начиная с младших разрядов источника;

распространить значение знакового разряда источника на свободные старшие разряды операнда назначения.

Источник-Куча сайтов, везде одно и то же.

74. Префікс повторення REP та його різновиди.

Один из возможных типов префиксов — это *префиксы повторения*. Они предназначены для использования цепочечными командами.

Префиксы повторения имеют свои мнемонические обозначения:

rep
repe или **repz**
repne или **repnz**

Эти префиксы повторения указываются перед нужной цепочечной командой в поле метки. Цепочечная команда без префикса выполняется один раз. Размещение префикса перед цепочечной командой заставляет ее выполняться в цикле.

Отличия приведенных префиксов в том, на каком основании принимается решение о циклическом выполнении цепочечной команды: *по состоянию регистра ecx/cx или по флагу нуля zf*:

- префикс повторения **rep** (REPeat). Этот префикс используется с командами, реализующими операции-примитивы пересылки и сохранения элементов цепочек — соответственно, **movs** и **stos**.

  Префикс **rep** заставляет данные команды выполняться, пока *содержимое в ecx/cx не станет равным 0*.
  При этом цепочечная команда, перед которой стоит префикс, *автоматически уменьшает содержимое ecx/cx на единицу*. Та же команда, но без префикса, этого не делает;

- префиксы повторения **repe** или **repz** (REPeat while Equal or Zero). Эти префиксы являются абсолютными синонимами.

  Они заставляют цепочечную команду выполняться до тех пор, пока *содержимое ecx/cx не равно нулю или флаг zf равен 1*.
  Как только одно из этих условий нарушается, управление передается следующей команде программы. Благодаря возможности анализа флага zf, наиболее эффективно эти префиксы можно использовать с командами cmps и scas для поиска отличающихся элементов цепочек.

- префиксы повторения **repne** или **repnz** (REPeat while Not Equal or Zero). Эти префиксы также являются абсолютными синонимами. Их действие на цепочечную команду несколько отличается от действий префиксов repe/repz. Префиксы repne/repnz заставляют цепочечную команду циклически выполняться до тех пор, пока *содержимое ecx/cx не равно нулю или флаг zf равен нулю*.

При невыполнении одного из этих условий работа команды прекращается.
Данные префиксы также можно использовать с командами cmps и scas, но для поиска совпадающих элементов цепочек.

http://kit.znu.edu.ua/eDoc/Arch/assembl/guide/Lesson/Lesson11/Les_11.htm

75.Що таке структурованість та модульність програм. Наведіть приклади.

Структорность программы: Структурой программы является последовательность содержимого ее кода, разбитого на определенные части. Для удобности и взаимопонимания программистов, структура программы имеет общие правила и каноны по которым строится код. Для разных языков программирования структуры разные. Например для Ассемблера структура программы имеет вид:

```
;--- Шапка программы ---
    list p=16f628a
   __config b'11111100110001'
   CBLOCK 0x20
     variable1   ; первая запись
     variable2   ; вторая запись
     variable3   ; третья запись
     variable4   ; четвёртая запись
   ENDC
     Const1  equ  .1
     Const2  equ  .5
     TRISB   equ  06h (1-й банк)
     PORTB   equ  06h (0-й банк)
     Status  equ  03h
     Z       equ  02h
```

шапка программы

```
;--- Тело программы ---
         org 0h
  ; можно выполнить 3 команды
  ; основной программы
    goto  start
         org 4h
  ; подпрограмма обработки
  ; прерываний
start ; продолж. основной программы
      ; инициализация
      ; решение задачи
     end
```

Тело программы

В шапке указывается различная служебная информация для компилятора. То есть, практически всё, что находится в шапке, служит инструкциями не для контроллера, а для компилятора (кроме слова конфигурации).

2. Тело программы.

В теле программы пишутся те инструкции, которые будут исполняться контроллером, то есть это как раз и есть сама

программа для контроллера. Хотя, здесь тоже встречаются директивы, предназначенные для компилятора, например для правильного размещения участков программы в памяти контроллера.

Модульность в языках программирования — принцип, согласно которому программное средство (ПС, программа, библиотека, веб-приложение и др.) разделяется на отдельные именованные сущности, называемые модулями. Модульность часто является средством упрощения задачи проектирования ПС и распределения процесса разработки ПС между группами разработчиков. При разбиении ПС на модули для каждого модуля указывается реализуемая им функциональность, а также связи с другими модулями.[1]

Роль модулей могут играть структуры данных, библиотеки функций, классы, сервисы и др. программные единицы, реализующие некоторую функциональность и предоставляющие интерфейс к ней.

Пример модульности:

Программный код часто разбивается на несколько файлов, каждый из которых компилируется отдельно от остальных. Такая модульность программного кода позволяет значительно уменьшить время перекомпиляции при изменениях, вносимых лишь в небольшое количество исходных файлов, и упрощает групповую разработку.

Так жето возможность замены отдельных компонентов (таких как jar-файлы, so или dll библиотеки) конечного программного продукта, без необходимости пересборки всего проекта (например, разработка плагинов к уже готовой программе).

Источники: Википедия и http://radiohlam.ru/teory/progr_asm_3.htm

76. Макроси. Приклади програмування

Определение:

Макрос (*от англ. macros, мн.ч. от macro*) — программный объект, при обработке «развёртывающийся» в последовательность действий и/или команд. Корректный перевод термина с английского — «макрокоманда», слово же «макрос» получило распространение благодаря использованию в локализованных продуктах американской корпорации Microsoft.

Макросы в программировании

В языке ассемблера, а также в некоторых других языках программирования, макрос — символьное имя, заменяемое при обработке препроцессором на последовательность программных инструкций. Для каждого компилятора (ассемблера) существует специальный синтаксис

объявления и вызова макросов. При этом внутри макроса могут быть условные операторы препроцессора, многие компиляторы поддерживают при вызове макросов передачу аргументов. В этом случае один и тот же макрос может «разворачиваться» в различные последовательности инструкций при каждом вызове — в зависимости от сработавших разветвлений внутри макроса и переданных ему аргументов.

```
.586
.model flat, c
.code
;процедура StrHex_MY записує текст шістнадцятькового коду
;перший параметр - адреса буфера результату (рядка символів)
;другий параметр - адреса числа
;третій параметр - розрядність числа у бітах (має бути кратна 8)
StrHex_MY proc
push ebp
mov ebp,esp
mov ecx, [ebp+8] ;кількість бітів числа
cmp ecx, 0
jle @exitp
shr ecx, 3 ;кількість байтів числа
mov esi, [ebp+12] ;адреса числа
mov ebx, [ebp+16] ;адреса буфера результату
@cycle:
mov dl, byte ptr[esi+ecx-1] ;байт числа - це дві hex-цифри
mov al, dl
shr al, 4 ;старша цифра
call HexSymbol_MY
mov byte ptr[ebx], al
mov al, dl ;молодша цифра
call HexSymbol_MY
mov byte ptr[ebx+1], al
mov eax, ecx
cmp eax, 4
jle @next
dec eax
and eax, 3 ;проміжок розділює групи по вісім цифр
cmp al, 0
jne @next
mov byte ptr[ebx+2], 32 ;код символа проміжку
inc ebx
@next:
add ebx, 2
dec ecx
jnz @cycle
mov byte ptr[ebx], 0 ;рядок закінчується нулем
@exitp:
pop ebp
ret 12
StrHex_MY endp
;ця процедура обчислює код hex-цифри
;параметр - значення AL
;результат -> AL
HexSymbol_MY proc
and al, 0Fh
add al, 48 ;так можна тільки для цифр 0-9
cmp al, 58
jl @exitp
add al, 7 ;для цифр A,B,C,D,E,F
@exitp:
ret
```

```
HexSymbol_MY endp
end
```

77. Процедури. Визначення, виклик. Приклади програмування

Процедура – группа команд для решения конкретной подзадачи.

Синтаксис процедуры:

имя_процедуры PROC [[модификатор_языка] язык] [расстояние]

команды

[имя_процедуры] ENDP

Пример

```
model small
.stack 100h
.data
.code
my_proc procnear
ret
my_proc endp
start:
end start
```

процедуру можно располагать в конце программы либо вкладывать в другую. Во втором случае неоходимо предусмотреть обход тела процедуры, ограниченного директивами PROC и ENDP, с помощью JMP/

Команда CALL осуществляет вызов процедуры (подпрограммы). Синтаксис команды:
call [модификатор] имя_процедуры
Подобно команде JMP команда CALL передает управление по адресу с символическим именем имя_процедуры, но при этом в стеке сохраняется адрес возврата (то есть адрес команды, следующей после команды CALL).
http://volgota.com/yaroslavz/assembler-14-procedury-uslovnye-perehody-i-cikly

78. Способи передавання значень параметрів процедурам

Допускается два различных способа передачи переменных процедуре или функции: *по ссылке* и *по значению.* Если переменная передается по ссылке, то это означает, что процедуре или функции будет передан адрес этой переменной в памяти. При этом происходит отождествление формального аргумента процедуры и переданного ей фактического параметра. Тем самым вызываемая процедура может изменить значение фактического параметра: если будет изменен формальный аргумент процедуры, то это скажется на значении переданного ей при вызове фактического параметра. Если же фактический параметр передается по значению, то формальный аргумент вызываемой процедуры или функции получает только значение фактического параметра, но не саму переменную, используемую в качестве этого параметра. Тем самым все изменения

значения формального аргумента не сказываются на значении переменной, являющейся фактическим параметром.
Способ передачи параметров процедуре или функции указывается при описании ее аргументов: имени аргумента может предшествовать явный описатель способа передачи. Описатель ByRef задает передачу по ссылке, а ByVal — по значению. Если же явное указание способа передачи параметра отсутствует, то по умолчанию подразумевается передача по ссылке.

79. Локальні дані процедур

Давайте посмотрим что мы знаем о локальных переменных, они могут быть использованы лишь в пределах локальной переменной и не занимают места, так как уничтожаются при завершении процедуры создателя. Что ещё может произвольно изменяться на протяжении процедуры и всегда возвращается к своему изначальному значению в конце процедуры? Конечно это стек. То есть для того чтоб создать локальную переменную нам достаточно лишь уменьшить стек на нужное нам количество байт, адресовать же переменную мы можем через **ebp**, создав переменные сразу после создание стекового кадра. Давайте рассмотрим простой пример процедуры с двумя параметрами и одной локальной переменной:

```
procedure proc
push ebp
mov ebp,esp
sub esp,4
.....
leave
ret 8
procedure endp
```

В этой процедуре возможно обращение к следующим переменным:

dword ptr [ebp+08h] - параметр 1

dword ptr [ebp+0Ch] - параметр 2

dword ptr [ebp-04h] - локальная переменная

Также мы видим, что в конце процедуры вместо команды **ret** находится команда **ret 8**. Дело в том,

что процедура должна удалять параметры, переданные ей в стеке, поэтому цифра после **ret** является количеством удаляемых из стека байт после завершения процедуры. Последняя деталь, стоящая обсуждения, является получение адреса локальных переменных или параметров для передачи их процедурам. Тут, к сожалению, не обойтись без какого-либо регистра, например **eax**. Для вычисления адреса используется команда **lea**:

lea eax,[ebp+08h]

80. Пролог та епілог процедури

Приклад процедури

```
. data
  varA dd 2014
  varB dd ?
  ...

. code

; ця процедура має два параметри, які передаються через стек
DoSomething proc
  push ebp                ;пролог
  mov ebp, esp

  mov eax, [ebp+12]       ;перший параметр - звичайне число ;якось його
  add eax, 8              використовуємо

  mov ebx, [ebp+8]        ;другий параметр - вказівник
  mov [ebx], eax          ;запис значення EAX у пам'ять по вказівнику

mov esp, ebp              ;епілог процедури - відновлюємо стек ;у стеку були два
pop ebp
ret 8                     параметри - 8 байтів
DoSomething endp

                          ;перший параметр - значення перемінної varA ;другий
  push varA push          параметр - адреса перемінної varB
  offset varB call
  DoSomething
```

Пролог - создание стекового кадра, что значит сохранить текущее значение стека в **ebp** (или любой другой регистр), предварительно сохранив его в тот же стек, и адресовать наши переменные уже через этот регистр.

push ebp

mov ebp,esp

Епілог - восстановление **ebp** и **esp**,

mov esp,ebp

pop ebp

81. Стековий кадр процедури

Процедура это определённая часть кода, которая вызывается командой **call**, при этом в стеке ей передаётся адрес следующей после **call** команды, что и позволяет командой **ret** перейти на эту метку. В досе каждая программа имела собственный сегмент, поэтому при вызове функций через прерывания параметры передавались в регистрах. Однако в Windows стек наследуем, то есть при вызове определённой функции она получает в своё распоряжение тот же стек, что и вызывающая его программа. Это даёт возможность передавать параметры через стек. И тут возникает вопрос как к ним обращаться, если единственная возможность обратиться к ним является обращением через **esp**. Но там где одна проблема там и другая, так как при адресации через определённый регистр мы должны сохранять его значение, в нашем случае не пользоваться стеком. Тогда самым простым решением является создание стекового кадра, что значит сохранить текущее значение стека в **ebp** (или любой другой регистр), предварительно сохранив его в тот же стек, и адресовать наши переменные уже через этот регистр.

push ebp

mov ebp,esp

Теперь нам нужно получить доступ к переменным переданным процедуре, так как по адресу **[ebp]** у нас находится сохранённый **ebp**, а по адресу **[ebp+04h]** находится адрес возврата процедуры, то первым переданным нам параметром будет **[ebp+08h]**, вторым - **[ebp+0Ch]** и т.д. Причём надо заметить, что параметр, загруженный в стек последним будет именно **[ebp+08h]**.

Сохранение **ebp** является обязательным, так как если вызывающая процедура уже имеет стековой кадр, то изменяя его мы разрушаем стековой кадр вызывающей процедуры. Также обязательно восстановление **ebp** и **esp**,

mov esp,ebp

pop ebp

82. Написання модульних програм на Асемблері

1. Створити новий проект

2. Додати у проект головний файл програмного коду.

3. Додати у проект модуль з ім'ям **module.asm (можна використовувати будь-яке ім'я). У** файл module.asm помістити текст процедури procedura і наданий нижче:

```
.586
.model flat, c.code
    procedura
    /текст процедури
    procedura endp
    end
```

Окрім файлу **module.asm** потрібно у робочу папку проекту записати файл заголовку модуля **module.inc**. У файлі заголовку вказуються директивою EXTERN імена процедур, які можуть викликатися іншими модулями. Поки що тільки одна така процедура у цьому модулі, тому файл **module.inc** містить один рядок:

```
EXTERN procedura : proc
```

4. Як **викликати процедуру procedura? Для цього спочатку потрібно у тексті головного файлу програми записати рядок**

```
include module.inc
```

Тепер можна використовувати процедуру procedura. Щоб її правильно викликати, необхідно передати потрібні параметри.

83. Можливості використання мов високого рівня сумісно з Асемблером

Інколи постає необхідність використання асемблера при програмування на мовах високого рівня, часто при зверненні до апаратного забезпечення. Убагатьох мовах програмування високого рівня є

зв'язок із асемблером. Наступний приклад демонструє додавання двох чисел на мові C з використанням вставки мови асемблера у середовищі Microsoft Visual C++[1]:

```
main ()
{
	int a = 1; // оголошуємо змінну a і кладемо туди значення 1
	int b = 2; // оголошуємо змінну b і кладемо туди значення 2
	int c; // оголошуємо змінну c, але не ініціалізуємо її
	// Початок асемблерної вставки
	__asm{
		mov eax, a // завантажуємо значення змінної a в регістр EAX
		mov ebx, b // завантажуємо значення змінної b в регістр EBX
		add eax, ebx // додаємо EAX з EBX, записуючи результат в EAX
		mov c, eax // завантажуємо значення EAX у змінну c
	}
	// Кінець асемблерної вставки
	// Виводимо вміст c на екран
	// За допомогою звичної функції printf
	printf ("a + b =% x +% x =% x \ n", a, b, c);
}
```

84. Конвенції виклику процедур

Соглашение о вызове (англ. *Calling convention*) — часть двоичного интерфейса приложений, которая регламентирует технические особенности вызова подпрограммы, передачи параметров, возврата из подпрограммы и передачи результата вычислений в точку вызова.

Соглашение вызова определяет следующие особенности процесса использования подпрограмм:

- Расположение входных параметров подпрограммы и возвращаемых ею значений. Наиболее распространённые варианты:
  - в регистрах;
  - в стеке;
  - в регистрах и стеке.
- Порядок передачи параметров. При использовании для параметров стека определяет, в каком порядке параметры должны быть помещены в стек, при использовании регистров — порядок сопоставления параметров и регистров. Варианты:
- Кто возвращает указатель стека на исходную позицию:
  - вызываемая подпрограмма — это сокращает объём команд, необходимых для вызова подпрограммы, поскольку команды восстановления указателя стека записываются только один раз, в конце подпрограммы;

- вызывающая программа — в этом случае вызов становится сложнее, но облегчается использование подпрограмм с переменным количеством и типом параметров.
- Какой командой вызывать подпрограмму и какой — возвращаться в основную программу.
- Содержимое каких регистров процессора подпрограмма обязана восстановить перед возвратом.

Соглашения вызова зависят от архитектуры целевой машины и компилятора.

Джерело:
http://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B3%D0%BB%D0%B0%D1%88%D0%B5%D0%BD%D0%B8%D0%B5_%D0%BE_%D0%B2%D1%8B%D0%B7%D0%BE%D0%B2%D0%B5

85. Конвенція cdecl. Приклад програмування.

Cdecl- тип викликів, який походить з мови програмування С і використовується за замовчуванням в компіляторах C/C++ на архітектурі x86[1]. Параметри в підпрограму передаються через стек, куди поміщаються у зворотному порядку (справа наліво). 32-бітний результат повертається в регістрі процесора EAX. Чисткою стека займається зовнішня підпрограма.

Приклад.

Функція на асемблері виробляє підсумовування двох цілих чисел і повертає цілий результат.

1. файл c_asm.cpp (bc)

```
#include <iostream.h>

int p1,p2;

int extern cdecl addint (int,int);

void main()

{

cout<<"\n p1=";

cin>>p1;

cout<<"\n p2=";

cin>>p2;

cout<<"\n сумма="<<addint(p1,p2);
```

```
}
```

файл c_asm.c (tc++)

```
# include <stdio.h>
int p1,p2;
int extern cdecl addint(int,int);
void main()
{
printf("\n Введите значение p1 : ");
scanf("%d",&p1);
printf("\n Введите значение p2 : ");
scanf("%d",&p2);
printf("\n Результат Y= ");
printf("%d",addint(p1,p2));
}
```

2. функция addint (asm_c.asm)

```
.MODEL SMALL
.CODE
PUBLIC _addint ; добавляется _

_addint proc far ; добавляется _
; используется атрибут процедуры far, если в C
; задана модель памяти LARGE
push bp
mov bp,sp
mov ax,[bp+6]
add ax,[bp+8]
pop bp
ret
```

```
_addint endp

end
```

Джерело:

http://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B3%D0%BE%D0%B4%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F_%D0%B2%D0%B8%D0%BA%D0%BB%D0%B8%D0%BA%D1%96%D0%B2

http://webcache.googleusercontent.com/search?q=cache:_L5DprTiGGwJ:phys.onu.edu.ua/files/student/3course/1term/assembler/lab_14.doc+&cd=5&hl=ru&ct=clnk&gl=ua

86. Конвенція stdcall. Приклад програмування.

stdcall - є стандартним типом виклику для Microsoft Win32 API і Open Watcom C++. Викликана підпрограма залишається відповідальною за очищення стека, але параметри поміщаються в стек у зворотному (справа-наліво, як і в cdecl). Регістри EAX, ECX, EDX призначені для використання в межах функції. Значення, що повертається, зберігається в регістрі EAX.

Приклад. Програма виводить повідомлення в невеликому windows-вікні.

```
.386

.model flat, stdcall

option casemap:none

include \masm32\include\kernel32.inc

include \masm32\include\user32.inc

include \masm32\include\windows.inc

includelib \masm32\lib\kernel32.lib

includelib \masm32\lib\user32.lib

.data

Caption db "Лаб1", 0

Text db "Здоровенькі були", 0

.code

start:

invoke MessageBoxA, 0,  ADDR Text, ADDR Caption, MB_ICONINFORMATION
```

invoke ExitProcess, 0

end start

Джерело:

http://uk.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B3%D0%BE%D0%B4%D0%B6%D0%B5%D0%BD%D0%BD%D1%8F_%D0%B2%D0%B8%D0%BA%D0%BB%D0%B8%D0%BA%D1%96%D0%B2

87. Асемблерні вставки у мовах високого рівня.

Фундаментальною перевагою асемблера перед мовами високого рівня є швидкість. Більшість програм, які працюють у режимі реального часу, або написані на мовою асемблер, або використовують у критичних ділянках програмного коду асемблерні модулі. Асемблер, вбудований у мови високого рівня, використовують для досягнення високої продуктивності роботи програм.

Зручність, вигідність та легкість використовування вбудованого асемблера важко переоцінити, оскільки він не потребує окремого компілювання, компунуванян блоків асемблерного коду. Це є надто важливо для швидкого розробляння та налогоджування програм.

Схематично блок команд на асемблері для програми розробленій на C++ виглядає так:

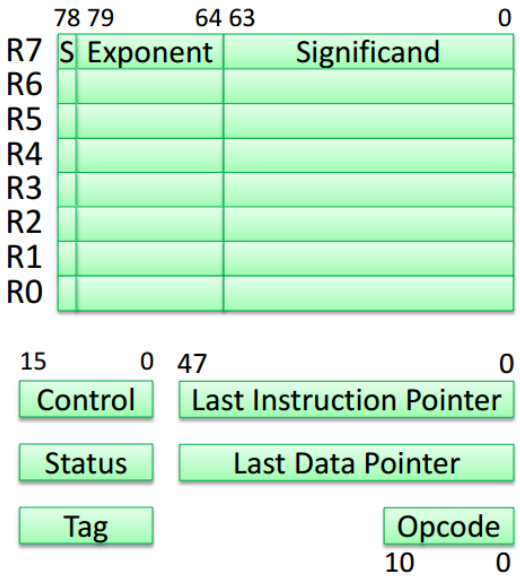
```
asm
{ асемблерні команди
}
```

88. Середовище x87 FPU

школа ПРОГРАММИСТА

Среда исполнения x87 FPU

- Восемь 80-разрядных регистров R0-R7 предназначены для хранения данных
- Status Register (регистр состояния) хранит информацию о текущем состоянии сопроцессора
- Control Register (регистр управления) управляет точностью вычислений и округлениями
- Tag Register хранит информацию о содержимом регистров данных
- Остальные регистры хранят информацию о последней исполненной команде



3

Джерело: http://frogs.digdes.ru/netcat_files/1162_49.pdf

89. Команди x87 FPU. Приклад програми.

Команды пересылки данных

Формат данных			Операция
Floating Point	Integer	Packed BCD	
FLD m(32,64,80)	FILD m(16,32,64)	FBLD m80	Загрузить в стек
FST m(32,64)/ST(i)	FIST m(16,32)		Сохранить ST(0)
FSTP m(32,64, 80)/ST(i)	FISTP m(16,32,64)	FBSTP m80	Вытолкнуть из стека
FXCH ST(i)			Обменять содержимое с ST(0)

Некоторые управляющие команды

FINIT	Инициализация FPU. Слово состояния обнуляется, слово управления устанавливается в 037F (округление к ближайшему, мантисса 64 бита, все ошибки замаскированы)
FLDCW/FSTCW mem16	Загрузить/Сохранить регистр управления
FSTSW mem16/AX	Сохранить регистр состояния в памяти/регистре AX
FINCSTP/FDECSTP	Увеличить/уменьшить на единицу указатель вершины стека
FNOP	Отсутствие операции

Сложение и вычитание

FADD m32	Прибавить к ST(0)
FADD m64	Прибавить к ST(0)
FADD ST(0), ST(i)	Прибавить к ST(0)
FADD ST(i), ST(0)	Прибавить к ST(i)
FADDP ST(i),ST(0)	Прибавить к ST(i) и вытолкнуть
FADDP	То же, но i=1
FIADD m32	Прибавить к ST(0) целое
FIADD m64	Прибавить к ST(0) целое

FSUB m32	Вычесть из ST(0)
FSUB m64	Вычесть из ST(0)
FSUB ST(0), ST(i)	Вычесть из ST(0)
FSUB ST(i), ST(0)	Вычесть из ST(i)
FSUBP ST(i),ST(0)	Вычесть из ST(i) и вытолкнуть
FSUBP	То же, но i=1
FISUB m32	Вычесть из ST(0) целое
FISUB m64	Вычесть из ST(0) целое
FSUBR/FSUBRP/ FISUBRP	Обратное вычитание
Например: FSUBR m32	Вычесть ST(0) из m32 и сохранить в ST(0)

Арифметические команды

FMUL/FMULP/FIMUL	Умножение
FDIV/FDIVP/FIDIV	Деление
FDIVR/FDIVPR/FIDIVR	Обратное деление
FABS/FCHS	Вычисление модуля/Смена знака ST(0)
FSQRT	Квадратный корень ST(0)
FPREM/FPREM1	Вычисление остатка от деления ST(0) на ST(1). ST(0) ← ST(0)-(Q*ST(1)), где Q – результат округления частного ST(0)/ST(1) до ближайшего целого в сторону нуля (FPREM1 соответствует стандарту IEEE 754)
FRNDINT	Округление ST(0) до целого в соответствии с установленными правилами округления (в слове управления)
FSIN/FCOS/FSINCOS	Вычисление синуса/косинуса/одновременно ST(0)
FPTAN/FPATAN	Вычисление тангенса/арктангенса ST(0)
FYL2X/FIL2XP1	ST(1) ← ST(1)log2(ST(0)) / ST(1) ← ST(1)log2(ST(0)+1) и вытолкнуть
F2XM1	ST(0) ← 2ST(0)-1
FXTRACT	Разложение ST(0) на порядок ST(1) и мантиссу ST(0)
FSCALE	Действие, обратное FXTRACT. ST(0) ←ST(0)* $2^{ST(1)}$

Приклад обчислення суми двох пар добутків Res = A×B + C×D

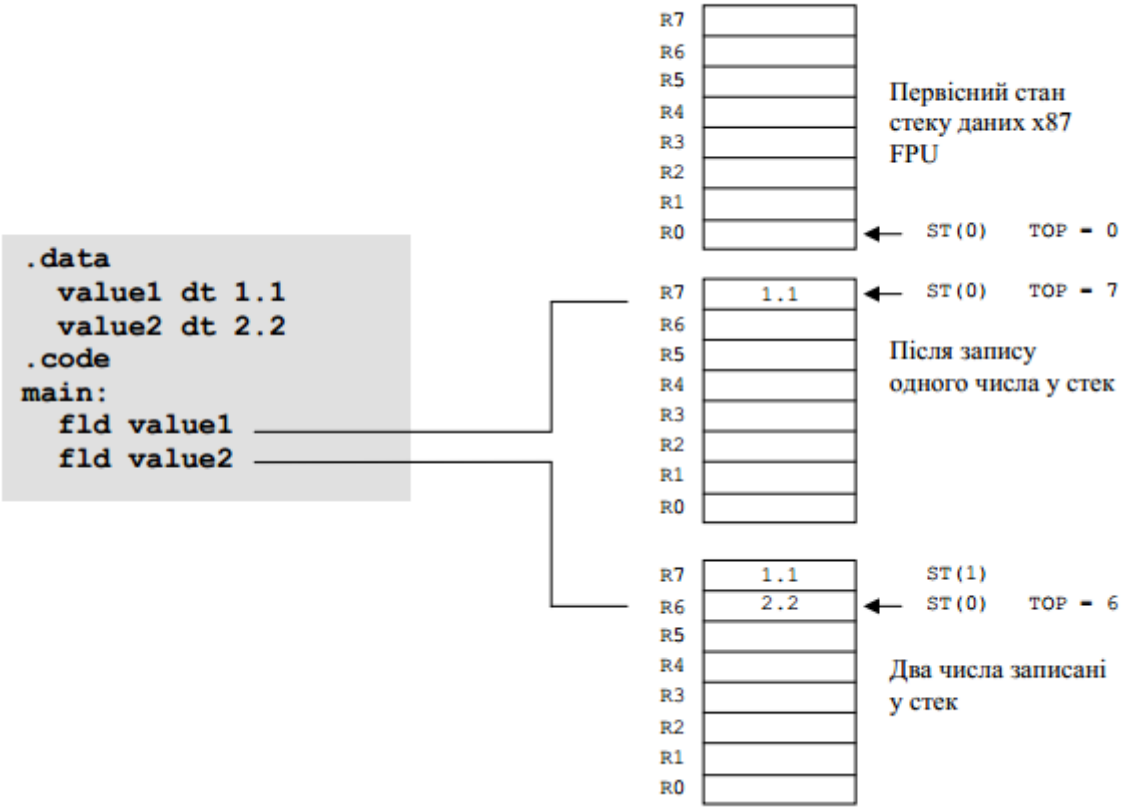
```
.data
  valA dq 5.6     ;A
  valB dq 2.4     ;B
  valC dq 3.8     ;C
  valD dq 10.3    ;D
  Res dq ?

.code
main:
  fld valA
  fmul valB
  fld valC
  fmul valD
  faddp st(1), st(0)
  fstp res
```

90. Стек даних x87 FPU. Приклад програми.

Приклад завантаження у стек FPU двох чисел:

```
.data
  value1 dt 1.1
  value2 dt 2.2
.code
main:
  fld value1
  fld value2
```



Стек FPU може зберігати одночасно не більше восьми значень. Вважається, що цього достатньо для обчислення будь-яких числових виражень. Адреса вершини стеку зберігається у регістрі статусу FPU (рис. 7)

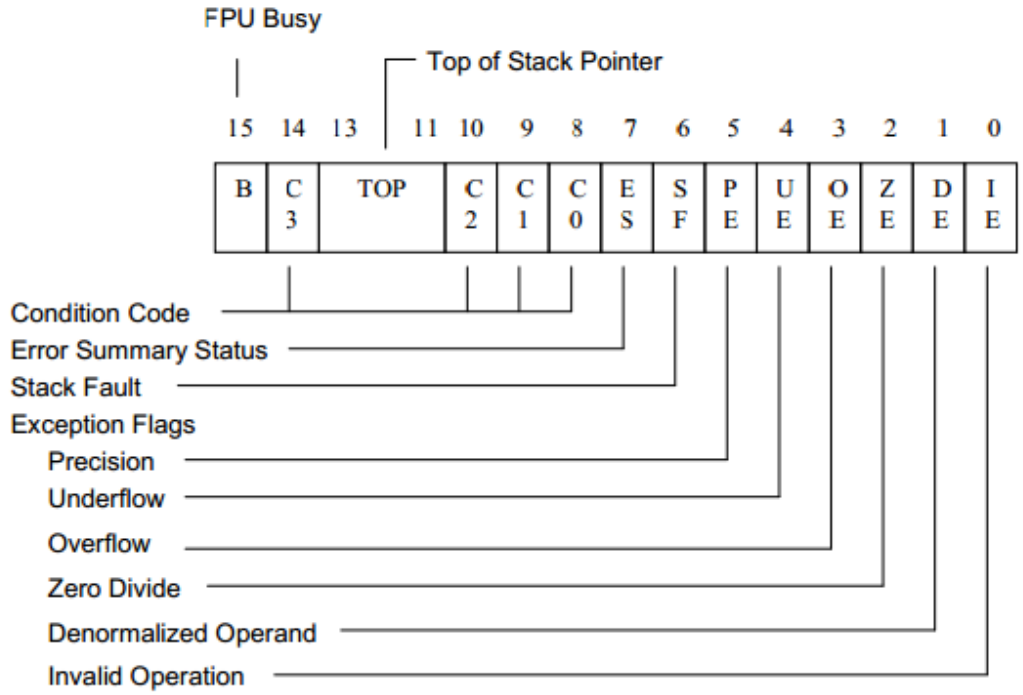


Рис. 7. Біти регістру статусу FPU

91. команди SIMD

Расширения потоковой обработки (Streaming SIMD Extensions) были введены в архитектуру процессоров Intel, начиная с процессора Pentium III. Эти расширения предназначены для использования в программах обработки мультимедийной информации наряду с более ранними мультимедиа-расширениями MMX.

Набор команд SIMD первоначально включает около 70 команд, которые могут оперировать данными как в MMX-регистрах, так и в новых SIMD-регистрах процессора. Специальные форматы данных введены для обеспечения обработки информации, представленной в форме вещественных значений (это необходимо, например, при 3D-моделировании).

При разработке SIMD-команд фирма Intel учла все нововведения, появившиеся в наборе команд 3DNow! компании AMD и практически включила этот набор команд в свое SIMD-расширение.

ADDPS сложить упакованные короткие вещественные значения

ADDSS Сложить одиночные короткие вещественные значения

ANDNPS Логическое НЕ-И над 128-битными операндами

ANDPS Логическое И над 128-битными операндами

CMPPS Сравнить упакованные короткие вещественные значения

CMPSS Сравнить одиночные короткие вещественные значения

COMISS Сравнить упорядоченно одиночные короткие вещественные значения и установить EFLAGS по результатам сравнения

CVTPI2PS Преобразовать упакованные знаковые целые двойные слова в упакованные короткие вещественные значения

CVTPS2PI Преобразовать упакованные короткие вещественные значения в упакованные знаковые целые двойные слова

CVTSI2SS Преобразовать одиночное знаковое двойное целое слово в одиночное короткое вещественное значение

CVTSS2SI Преобразовать одиночное короткое вещественное значение в одиночное знаковое целое двойное слово

CVTTPS2PI Преобразовать упакованные короткие вещественные значения в упакованные знаковые целые двойные слова без точного округления

CVTTSS2SI Преобразовать одиночное короткое вещественное значение в одиночное знаковое целое двойное слово без точного округления

DIVPS Делить упакованные короткие вещественные значения

DIVSS Делить одиночные короткие вещественные значения

LDMXCSR Загрузить регистр управления/статуса SIMD

MASKMOVQ Переслать 64-битные данные из MMX-регистра с маскированием

MAXPS Найти наибольшее для упакованных вещественных значений

MAXSS Найти наибольшее для одиночных вещественных значений

MINPS Найти наименьшее для упакованных вещественных значений

MINSS Найти наименьшее для одиночных коротких вещественных значений

MOVAPS Переслать выровненные упакованные короткие вещественные значения

MOVHLPS Переслать два старших упакованных коротких вещественных значения в младшие

MOVHPS Переслать два старших упакованных коротких вещественных значения в/из памяти

MOVLHPS Пересылать два младших упакованных коротких вещественных значения в старшие

MOVLPS Переслать два младших упакованных коротких вещественных значения в/из памяти

MOVMSKPS Переслать маску состояния упакованных коротких вещественных значений в целочисленный регистр

MOVNTPS Переслать выровненные упакованные короткие вещественные значения без использования КЭШ

MOVNTQ Переслать 64-битные данные из MMX-регистра в память без использования КЭШ

MOVSS Переслать одиночное короткое вещественное значение

MOVUPS Переслать невыровненные упакованные короткие вещественные значения

MULPS Перемножить упакованные короткие вещественные значения

MULSS Перемножить одиночные короткие вещественные значения

ORPS Логическое ИЛИ над 128-битными операндами

PAVGB Усреднить беззнаковые упакованные байты

PAVGW Усреднить беззнаковые упакованные слова

PEXTRW Развернуть слово

PINSRW Вставить слово

PMAXSW Найти наибольшее для упакованных знаковых целых слов

PMAXUB Найти наибольшее для упакованных беззнаковых целых байтов

PMINSW Найти наименьшее для упакованных знаковых целых слов

PMINUB Найти наименьшее для упакованных беззнаковых целых байтов

PMOVMSKB Переслать маску состояния упакованных байтов в целочисленный регистр

PMULHUW Перемножить упакованные беззнаковые слова

PREFETCH Упреждающая выборка данных

PSADBW Вычислить абсолютное отклонение упакованных беззнаковых байтов

PSHUFW Перестановка упакованных слов

RCPPS Найти обратные величины для упакованных коротких вещественных значений

RCPSS Найти обратную величину для одиночного короткого вещественного значения

RSQRTPS Найти обратные величины квадратных корней упакованных коротких вещественных значений

RSQRTSS Найти обратную величину квадратного корня одиночного короткого веществennго значения

SFENCE Защита операций записи

SHUFPS Перестановка коротких вещественных значений

SQRTPS Найти квадратные корни от упакованных коротких вещественных значений

SQRTSS Найти квадратный корень от одиночного короткого вещественного значений

STMXCSR Сохранить регистр управления/статуса SIMD

SUBPS Вычитание упакованных коротких вещественных значений

SUBSS Вычитание одиночных коротких вещественных значений

UCOMISS Сравнить неупорядоченно одиночные короткие вещественные значения и установить EFLAGS по результатам сравнения

UNPCKHPS Распаковать старшие упакованные короткие вещественные значения

UNPCKLPS Распаковать младшие упакованные короткие вещественные значения

XORPS Логическое ИСКЛЮЧАЮЩЕЕ ИЛИ над 128-битными операндами

92. Розширення MMX

MMX (Multimedia Extensions — мультимедийные расширения) — коммерческое название дополнительного набора инструкций, выполняющих характерные для процессов кодирования/декодирования потоковых аудио/видео данных действия за одну машинную инструкцию. Впервые появился в процессорах Pentium MMX. Разработан в лаборатории Intel в Хайфе, Израиль, в первой половине 1990-х.

Аббревиатура MMX происходит от выражения MultiMedia eXtension — расширение для мультимедиа, которое реализовано фирмой Intel в своей новой серии процессоров MMX с тактовой частотой 166 МГц и более.

Процессор Pentium MMX отличается от обычного Pentium по шести основным пунктам:

1. добавлено 57 новых команд обработки данных;
2. увеличен в два раза объем внутреннего кэша (16 кб для команд и столько же — для данных);
3. увеличен объем буфера адресов перехода (Branch Target Buffer — BTB), используемого в системе предсказания переходов (Branch Prediction);

4. оптимизирована работа конвейера (Pipeline);
5. увеличено количество буферов записи (Write Buffers);
6. введено так называемое двойное электропитание процессора.

Набор из 57 новых команд и является основным отличием; остальные пять — не более, чем сопутствующие изменения. Хотя увеличенный объем кэша и внутренних буферов и оптимизированный конвейер несколько ускоряют работу любых приложений, однако основное увеличение производительности — до 60 % — возможно только при использовании программ, правильно применяющих технологию MMX в обработке данных.

93. Розширення SSE

SSE (англ. *Streaming SIMD Extensions*, потоковое SIMD-расширение процессора) — это SIMD (англ. *Single Instruction, Multiple Data*, Одна инструкция — множество данных) набор инструкций, разработанный Intel и впервые представленный в процессорах серии Pentium III как ответ на аналогичный набор инструкций 3DNow! отAMD, который был представлен годом раньше. Первоначально названием этих инструкций было KNI — *Katmai New Instructions* (Katmai — название первой версии ядра процессора Pentium III).

SSE включает в архитектуру процессора восемь 128-битных регистров и набор инструкций, работающих со скалярными и упакованными типами данных.

Преимущество в производительности достигается в том случае, когда необходимо произвести одну и ту же последовательность действий над разными данными. В таком случае блоком SSE осуществляется распараллеливание вычислительного процесса между данными.

В SSE добавлены восемь (шестнадцать для x86-64) 128-битных регистров, которые называются xmm0 — xmm7 (-xmm15).

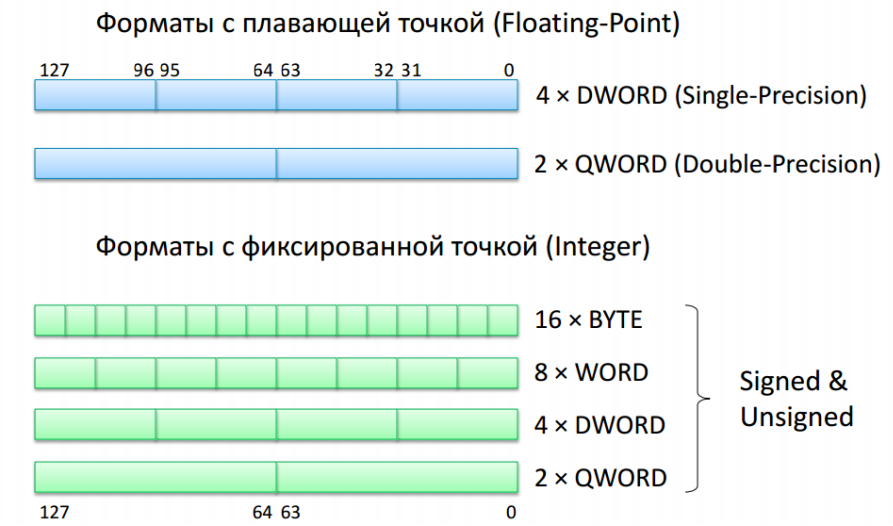
Каждый регистр может содержать четыре 32-битных значения с плавающей точкой одинарной точности.

Особенности:

- 8 (в x86-64 - 16)128-битных регистров XMM.
- 32-битный (в x86-64 - 64) регистр флагов (MXCSR).
- 128-битный упакованный тип данных с плавающей точкой одинарной точности.
- Инструкции над вещественными числами одинарной точности.
- Инструкции явной предвыборки данных, контроля кэширования данных и контроля порядка операций сохранения.

94. Типи даних SSE

Основной тип данных – упакованные числа с плавающей запятой одинарной точности. В одном 128-битном регистре размещаются сразу 4 таких числа: в битах 127—96 число 3, в битах 95—64 число 2, в битах 63—32 число 1, в битах 31—0 число 0. Целочисленные команды SSE могут работать с упакованными байтами, словами или двойными словами. Но эти команды оперируют данными, находящимися в регистрах MMX



95. Векторны команди SSE

Streaming SIMD Extensions (SSE) – это векторные команды с плавающей запятой, выполняемые процессором в специальном блоке. Это развитие системы команд MMX (MultiMedia eXtensions – мультимедийные расширения). MMX предлагает работу с целочисленными векторами с количеством элементов от 1 до 8. При этом используются 64-разрядные регистры MMX, физически размещаемые в регистрах сопроцессора с плавающей запятой.

Команды для чисел с плавающей точкой

- Команды пересылки
  - Скалярные типы – MOVSS
  - Упакованные типы – MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS
- Арифметические команды
  - Скалярные типы – ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS
  - Упакованные типы – ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS
- Команды сравнения
  - Скалярные типы – CMPSS, COMISS, UCOMISS
  - Упакованные типы – CMPPS
- Перемешивание и распаковка
  - Упакованные типы – SHUFPS, UNPCKHPS, UNPCKLPS
- Команды для преобразования типов
  - Скалярные типы – CVTSI2SS, CVTSS2SI, CVTTSS2SI

    - Упакованные типы – CVTPI2PS, CVTPS2PI, CVTTPS2PI
- Битовые логические операции
    - Упакованные типы – ANDPS, ORPS, XORPS, ANDNPS

Команды для целых чисел

- Арифметические команды
    - PMULHUW, PSADBW, PAVGB, PAVGW, PMAXUB, PMINUB, PMAXSW, PMINSW
- Команды пересылки
    - PEXTRW, PINSRW
- Другие
    - PMOVMSKB, PSHUFW

Другие команды

- Работа с регистром MXCSR
    - LDMXCSR, STMXCSR
- Управление кэшем и памятью
    - MOVNTQ, MOVNTPS, MASKMOVQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE

96 Горизонтальне додавання

Для отримання кінцевого результату скалярного добутку потрібно обчислити суму чотирьох елементів регістру XMM0, тобто виконати так зване "горизонтальне" додавання (Рис. 2):

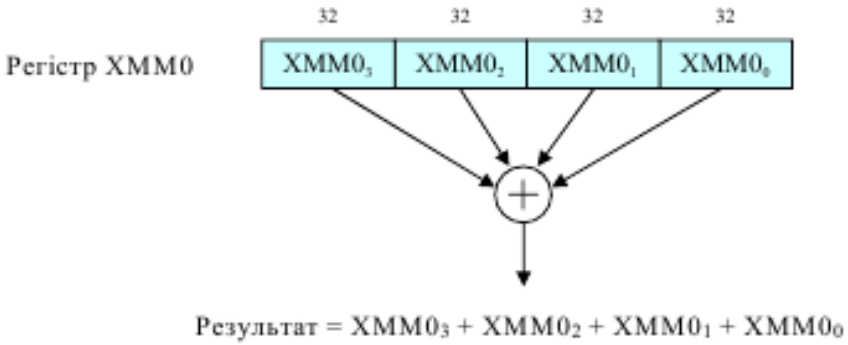


Рис. 2. "Горизонтальне" додавання елементів регістру

Знаходження суми усіх чвертинок регістру XMM0 можна виконати двома командами haddps:

haddps xmm0, xmm0

haddps xmm0, xmm0

Одна команда haddps працює так:

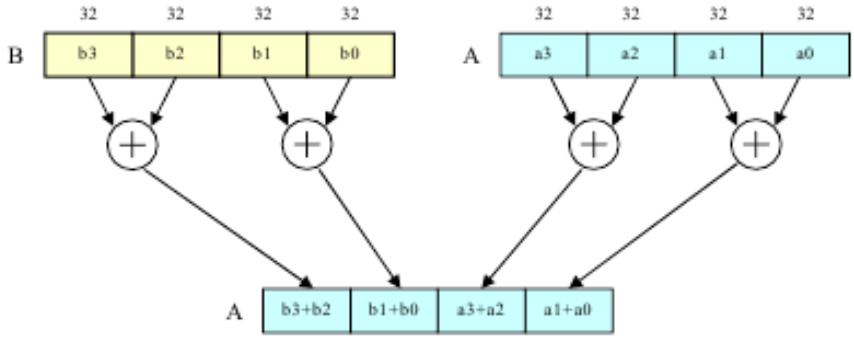


Рис. 3. Схема виконання команди **haddps A,B**

Команда haddps належіть до складу команд SSE3, тому якщо процесор не підтримує це розширення, то горизонтальне додавання ускладнюється – його можна запрограмувати на основі команд shufps.

97. Команди MOVUPS та MOVAPS. Приклад використання

MOVAPS

Переслать выровненные упакованные короткие вещественные значения

I	D	Z	O	U	P

Код	Команда	Описание	Проц.	Пример
0F 28 */r*	MOVAPS *xmm1,xmm2/m128*	*xmm1 = xmm2/m128*	Pentium III	movaps xmm2,xmm3
0F 29 */r*	MOVAPS *xmm2/m128,xmm*	*xmm2/m128 = xmm1*	Pentium III	movaps [ebx],xmm4

```
IF (destination = DEST) THEN
  IF (SRC = m128)THEN (* Команда загрузки *)
     DEST[127-0] = m128;
  ELSE(* Команда переноса *)
     DEST[127=0] = SRC[127-0];
  FI;
```

```
ELSE
  IF (destination = m128)THEN (* Команда сохранения *)
     m128 = SRC[127-0];
  ELSE(* Команда переноса *)
     DEST[127-0] = SRC[127-0];
  FI;
FI;
```

Команда MOVAPS служит для пересылки 128-битных данных между SIMD-регистрами общего назначения или между SIMD-регистром и памятью. Данные располагаются в памяти так, что младший байт 128-битного значения располагается по эффективному адресу, задаваемому командой. При обращении к памяти данные пересылаются порциями по 16-бит.

Использование команды MOVAPS возможно только с данными в памяти выровненными по 16-битным границам. Такое выравнивание рекомендуется для всех SIMD-данных, поскольку работа процессора оптимизируется именно под него. Для пересылки невыровненных операндов может использоваться команда MOVUPS.

Использование префикса REP/REPNE (F3h/F2h) и префикса размера операнда (66h) с командой MOVAPS является зарезервированной недокументированной функцией. Различные модели процессоров по разному реагируют на эту ситуацию. Не рекомендуется применять в программах комбинацию команды MOVAPS и указанных префиксов, т.к. в дальнейших моделях процессоров эта функция может быть изменена или исключена.

при некорретном эффективном адресе операнда в памяти в сегментах CS, DS, ES, FS или GS, и для всех сегментов если операнд в памяти не выровнен по 16-байтным границам.
при использовании некорректного эффективного адреса в сегменте SS.
при страничной ошибке.
, если CR0.TS = 1.
, если CR0.EM = 1 или CR4.OSFXSR = 0, а также при встрече незамаскированных SIMD-исключений, когда CR4.OSXMMEXCPT = 0.
при встрече незамаскированных SIMD-исключений, когда CR4.OSXMMEXCPT = 1.

, если любая часть операнда находится вне пространства эффективных адресов от 0 до 0FFFFh.
, если CR0.TS = 1.
, если CR0.EM = 1 или CR4.OSFXSR = 0, а также при встрече незамаскированных SIMD-исключений, когда CR4.OSXMMEXCPT = 0.
при встрече незамаскированных SIMD-исключений, когда CR4.OSXMMEXCPT = 1.

Такие же, как и в режиме реальной адресации.
при страничной ошибке.

http://www.club155.ru/x86cmdsimd/MOVAPS

MOVUPS

Переслать невыровненные упакованные короткие вещественные значения

I	D	Z	O	U	P

Код	Команда	Описание	Проц.	Пример
0F 10 */r*	MOVUPS *xmm1,xmm2/m128*	*xmm1 = xmm2/m128*	Pentium III	movups xmm2,xmm3
0F 11 */r*	MOVUPS *xmm2/m128,xmm*	*xmm2/m128 = xmm1*	Pentium III	movups [ebx],xmm4

```
IF (destination = xmm) THEN
  IF (SRC = m128)THEN(* Команда загрузки *)
     DEST[127-0] = m128;
  ELSE (* Команда переноса *)
     DEST[127-0] = SRC[127-0];
  FI;
ELSE
  IF (destination = m128) THEN(* Команда сохранения *)
     m128 = SRC[127-0];
  ELSE (* Команда переноса *)
     DEST[127-0] = SRC[127-0];
  FI;
FI;
```

Команда MOVUPS служит для пересылки 128-битных данных между SIMD-регистрами общего назначения или между SIMD-регистром и памятью. Данные располагаются в памяти так, что младший байт 128-битного значения располагается по эффективному адресу, задаваемому командой. При обращении к памяти данные пересылаются порциями по 16-бит.

Использование команды MOVUPS возможно с любыми (выровненными и невыровненными) данными в памяти. Для всех SIMD-данных рекомендуется использовать выравнивание по 16-битным границам, поскольку работа процессора оптимизируется именно под него. Для пересылки выровненных операндов может использоваться команда MOVAPS.

Использование префикса REP/REPNE (F3h/F2h) и префикса размера операнда (66h) с командой MOVUPS является зарезервированной недокументированной функцией. Различные модели процессоров по разному реагируют на эту ситуацию. Не рекомендуется применять в программах комбинацию команды MOVUPS и указанных префиксов, т.к. в дальнейших моделях процессоров эта функция может быть изменена или исключена.

при некорретном эффективном адресе операнда в памяти в сегментах CS, DS, ES, FS или GS.
при использовании некорректного эффективного адреса в сегменте SS.
при страничной ошибке.
, если CR0.TS = 1.
при невыровненных обращениях к памяти (когда CR0.AM = 1, EFLAGS.AC = 1 и CPL = 3).
, если CR0.EM = 1.

, если любая часть операнда находится вне пространства эффективных адресов от 0 до 0FFFFh.
, если CR0.TS = 1.
, если CR0.EM = 1.

Такие же, как и в режиме реальной адресации.
при страничной ошибке.
при невыровненных обращениях к памяти (когда CR0.AM = 1 и EFLAGS.AC = 1).

98. Векторні команди множення SSE. Приклад використання

SSE – это векторные команды с плавающей запятой, выполняемые процессором в специальном блоке. В системе команд SSE (а также 2, 3 и 4) используются 128-битные специальные регистры XMM и отдельные операционные устройства. Допускается как обработка с плавающей, так и с фиксированной запятой.
mulps — параллельное умножение, выходной операнд всегда XMM-регистр
mulss — скалярное умножение, входной/выходной операнд память-32 и XMM-регистр
пример: mulps xmm1, xmm0

99. Векторні команди додавання SSE. Приклад використання

SSE – это векторные команды с плавающей запятой, выполняемые процессором в специальном блоке. В системе команд SSE (а также 2, 3 и 4) используются 128-битные специальные регистры XMM и отдельные операционные устройства. Допускается как обработка с плавающей, так и с фиксированной запятой.

addps — параллельное сложение, выходной операнд всегда XMM-регистр
addss — скалярное сложение, входной/выходной операнд
память-32 и XMM-регистр
пример: addps xmm1, xmm0

100. Команда SHUFPS та її застосування

Команда SHUFPS осуществляет пересылку любых двух из четырех коротких вещественных значений, упакованных в операнде-источнике команды (SIMD-регистр или операнд в памяти) в младшие позиции в операнде-назначении (SIMD-регистр), а также любых двух из четырех коротких вещественных значений, упакованных в операнде-назначении команды (SIMD-регистр) в старшие позиции этого же операнда. Для каждой позиции в операнде-назначении третий операнд команды (*imm8*) задает номер копируемого в нее слова из операнда-источника (для двух младших полей) или из операнда-назначения (для двух старших полей). Биты 0,1 параметра *imm8* задают номер копируемого слова для младшего 16-битного поля операнда-назначения, биты 2, 3 — для следующего, 4,5 — для третьего и 6, 7 — для самого старшего. Эти биты кодируются обычными двоичными кодами, например код 11b в битах 2, 3 означает, что в соответствующую позицию будет скопировано самое старшее слово из операнда-источника.
При задании в SHUFPS в качестве операнда-источника и в качестве операнда-назначения одного и того же регистра, эта команда может возвращать любые последовательности четырех коротких вещественных значений, упакованных во входном операнде.
Использование префикса REP/REPNE (F3h/F2h) и префикса размера операнда (66h) с командойSHUFPS является зарезервированной недокументированной функцией.
Различные модели процессоров по разному реагируют на эту ситуацию. Не рекомендуется применять в программах комбинацию команды SHUFPS и указанных префиксов, т.к. в дальнейших моделях процессоров эта функция может быть изменена или исключена.

Джерело: http://www.club155.ru/x86cmdsimd/SHUFPS

101. Розширення AVX

Advanced Vector Extensions (AVX) — расширение системы
команд x86 для микропроцессоров Intel и AMD, предложенное Intel в марте 2008.AVX предоставляет различные улучшения, новые инструкции и новую схему кодирования машинных кодов.

УЛУЧШЕНИЯ:

1)кодирования инструкций VEX

2)Ширина векторных регистров SIMD увеличивается со 128 (XMM) до 256 бит (регистры YMM0 — YMM15). Существующие 128-битные SSE инструкции будут использовать младшую половину новых YMM регистров, не изменяя старшую часть. Для работы с YMM регистрами добавлены новые 256-битные AVX инструкции. В будущем возможно расширение векторных регистров SIMD до 512 или 1024 бит. Например, процессоры с архитектурой Larrabee уже имеют векторные регистры (ZMM) шириной в 512 бит, и используют для работы с ними SIMD команды с MVEX и VEX префиксами, но при этом они не поддерживают AVX.

3)Неразрушающие операции. Набор AVX инструкций использует трёхоперандный синтаксис. Например, вместо $a = a + b$ можно использовать $c = a + b$, при этом регистр a остаётся неизменённым. В случаях, когда значение a используется дальше в вычислениях, это повышает производительность, так как избавляет от необходимости сохранять перед

вычислением и восстанавливать после вычисления регистр, содержавший a, из другого регистра или памяти.

4)Для большинства новых инструкций отсутствуют требования к выравниванию операндов памяти. Однако рекомендуется следить за выравниванием на размер операнда, во избежание значительного снижения производительности.

5)Набор инструкций AVX содержит в себе аналоги 128-битных SSE инструкций для вещественных чисел. При этом, в отличие от оригиналов, сохранение 128-битного результата будет обнулять старшую половину YMM регистра. 128-битные AVX инструкции сохраняют прочие преимущества AVX, такие как новая схема кодирования, трехоперандный синтаксис и невыровненный доступ к памяти. Рекомендуется отказаться от старых SSE инструкций в пользу новых 128-битных AVX инструкций, даже если достаточно двух операндов.

ПРИМЕНЕНИЯ: подходит для интенсивных вычислений с плавающей точкой в мультимедиа программах и научных задачах. Там, где возможна более высокая степень параллелизма, увеличивает производительность с вещественными числами.

Джерело: http://ru.wikipedia.org/wiki/AVX

102. Типи даних AVX

Для поддержки технологии AVX в языке C

1) __**m256** - 8×32-bit с плавающей точкой
2) __**m256d** - 4×64-bit с плавающей точкой
3) __**m256i** - 32×8-bit, 16×16-bit, 8×32-bit, 4×64-bit целых

• Определены в <immintrin> как объединения:

__m256	m256_f32[8]	-С плавающей точкой
__m256d	m256d_f64[4]	-С плавающей точкой
__m256i	m256_u64[4]	-Беззнаковое целое
	m256_u32[8]	-Беззнаковое целое
	m256_u16[16]	-Беззнаковое целое
	m256_u8[32]	-Беззнаковое целое
	m256_i64[4]	-Знаковое целое
	m256_i32[8]	-Знаковое целое
	m256_i16[16]	-Знаковое целое
	m256_i8[32]	-Знаковое целое

Джерело: http://frogs.digdes.ru/netcat_files/1162_23.pdf

103. Команды AVX

Инструкция	Описание
VBROADCASTSS, VBROADCASTSD,	Копирует 32-х, 64-х или 128-ми битный операнд из памяти во все элементы векторного регистра XMM или YMM.

VBROADCASTF128	
VINSERTF128	Замещает младшую или старшую половину 256-ти битного регистра YMM значением 128-ми битного операнда. Другая часть регистра-получателя не изменяется.
VEXTRACTF128	Извлекает младшую или старшую половину 256-ти битного регистра YMM и копирует в 128-ми битный операнд-назначение.
VMASKMOVPS, VMASKMOVPD	Условно считывает любое количество элементов из векторного операнда из памяти в регистр-получатель, оставляя остальные элементы несчитанными и обнуляя соответствующие им элементы регистра-получателя. Также может условно записывать любое количество элементов из векторного регистра в векторный операнд в памяти, оставляя остальные элементы операнда памяти неизменёнными
VPERMILPS, VPERMILPD	Переставляет 32-х или 64-х битные элементы вектора согласно операнду-селектору (из памяти или из регистра).
VPERM2F128	Переставляет 4 128-ми битных элемента двух 256-ти битных регистров в 256-ти битный операнд-назначение с использованием непосредственной константы (imm) в качестве селектора.
VZEROALL	Обнуляет все YMM регистры и помечает их как неиспользуемые. Используется при переключении между 128-ми битным режимом и 256-ти битным.
VZEROUPPER	Обнуляет старшие половины всех регистров YMM. Используется при переключении между 128-ми битным режимом и 256-ти битным.

104. Особенности команд AVX по отношению к SSE

Набор инструкций AVX содержит в себе аналоги 128-битных SSE инструкций для вещественных чисел. При этом, в отличие от оригиналов, сохранение 128-битного результата будет обнулять старшую половину YMM регистра. 128-битные AVX инструкции сохраняют прочие преимущества AVX, такие как новая схема кодирования, трехоперандный синтаксис и невыровненный доступ к памяти. Рекомендуется отказаться от старых SSE инструкций в пользу новых 128-битных AVX инструкций, даже если достаточно двух операндов.

105. Эволюция SIMD-расширений системы команд процессоров архитектуры x86

MMX — Multimedia Extensions. Коммерческое название дополнительного набора инструкций, выполняющих характерные для процессов кодирования/декодирования потоковых аудио/видео данных действия за одну машинную инструкцию. Впервые появился в процессорах Pentium MMX.

MMX Extended — расширенный набор инструкций MMX, используемый в процессорах AMD и Cyrix.

3DNow! — расширение набора команд MMX процессоров AMD, начиная с AMD K6-2.

3DNow! Extended — расширение набора команд 3DNow! процессоров AMD, начиная с AMD Athlon.

SSE — набор инструкций, разработанный Intel, и впервые представленный в процессорах серии Pentium III

SSE2 — набор инструкций, разработанный Intel, и впервые представленный в процессорах серии Pentium 4.

SSE3 — третья версия SIMD-расширения Intel, потомок SSE, SSE2 и x87. Представлен 2 февраля 2004 года в ядре Prescott процессора Pentium 4.

SSSE3 — набор SIMD-инструкций, используемый в процессорах Intel Core 2 Duo.

SSE4 — новая версия SIMD-расширения Intel. Анонсирован 27 сентября 2006 года. Представлен в 2007 году в процессорах серии Penryn.

AVX — анонсированная версия SIMD-расширения Intel, которая представлена в 2010 году в процессорах архитектуры Sandy Bridge.