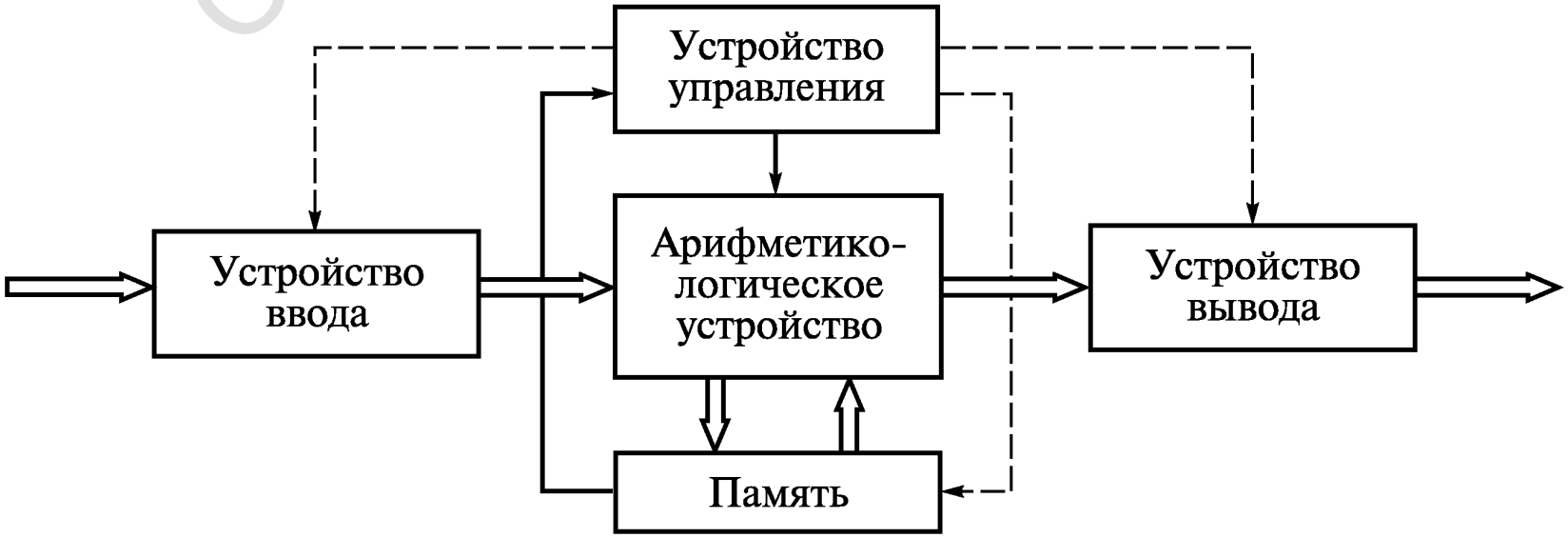
12. Введение в программирование на языке ассемблера. Часть 1

Разделы:

- Основы программирования на языке ассемблера с использованием GNU *Assembler*
- Система команд с примерами использования
- Подпрограммы и их вызов

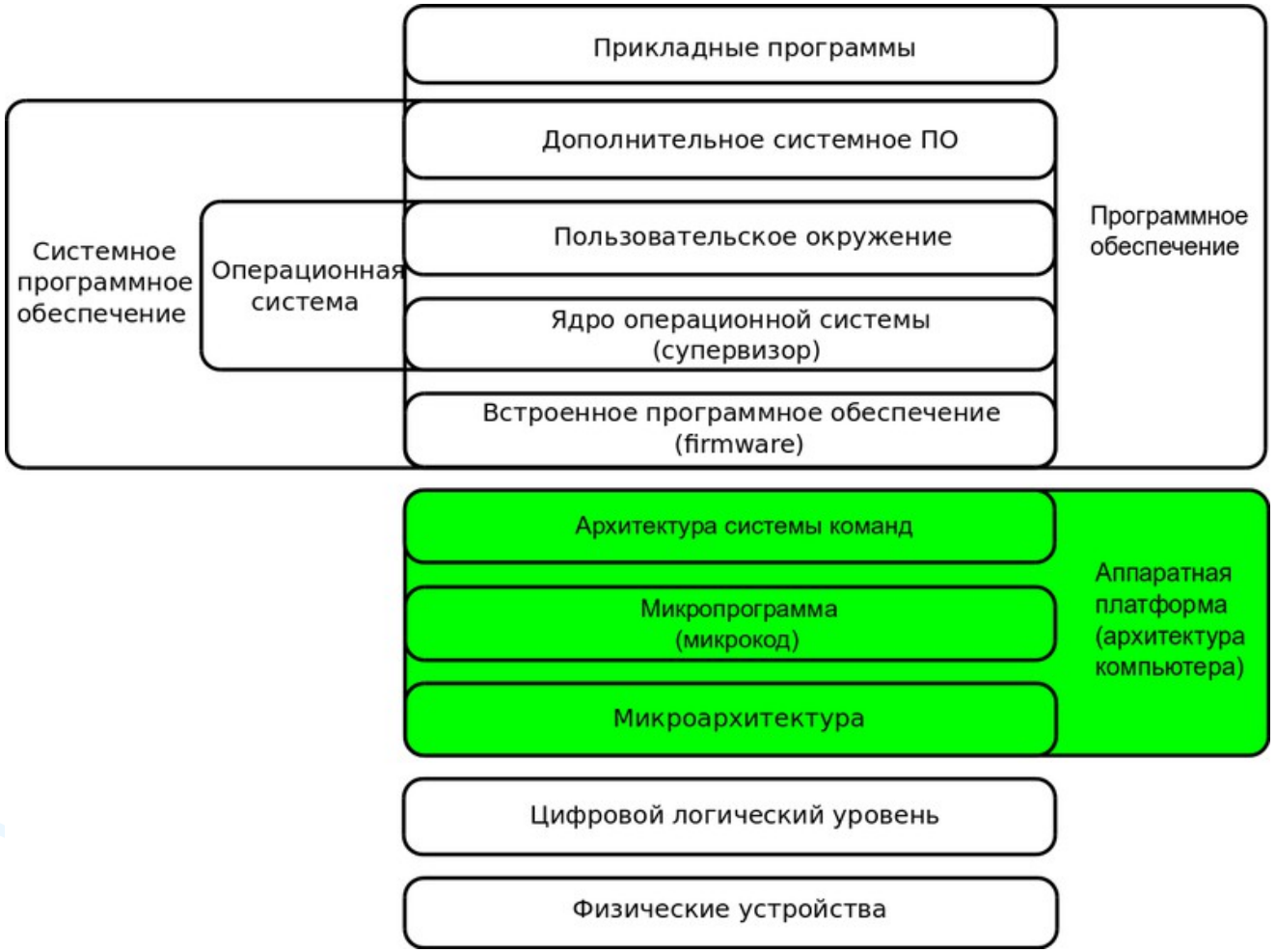
Архитектура вычислительных машин



⇒ линии связи для данных; → линии связи для команд; — — → линии связи для управления

- Источник — книга Мелехина В.Ф. и Павловского Е.Г. «Вычислительные машины, системы и сети» (доступен из корпоративной сети СФУ) - http://lib3.sfu-kras.ru/ft/lib2/elib/u004/i-705826.djvu?

Архитектура вычислительных машин



- Источник — https://ru.wikipedia.org/wiki/Файл:GeneralizedLayered

Программирование на ассемблере для x86

- Зачем изучать Ассемблер и программировать на нем?
  - Большую часть задач решает “plain” C
  - Очень небольшая часть ядра Linux и весьма специфическая часть системных библиотек написана на Ассемблере
- Но:
  - программист, не понимающий, как работает процессор на уровне команд, «не ведает, что творит»
  - отсюда получаются офисные приложения, не помещающиеся в отведенные 4 Гб
- Целевая архитектура 1 - x86, синтаксис Ассемблера – AT&T, вместо принятого в Windows – Intel-синтаксиса, рабочий инструмент – *gas*
- Целевая архитектура 2 - MIPS32, рабочий инструмент – текстовый редактор + симулятор SPIM

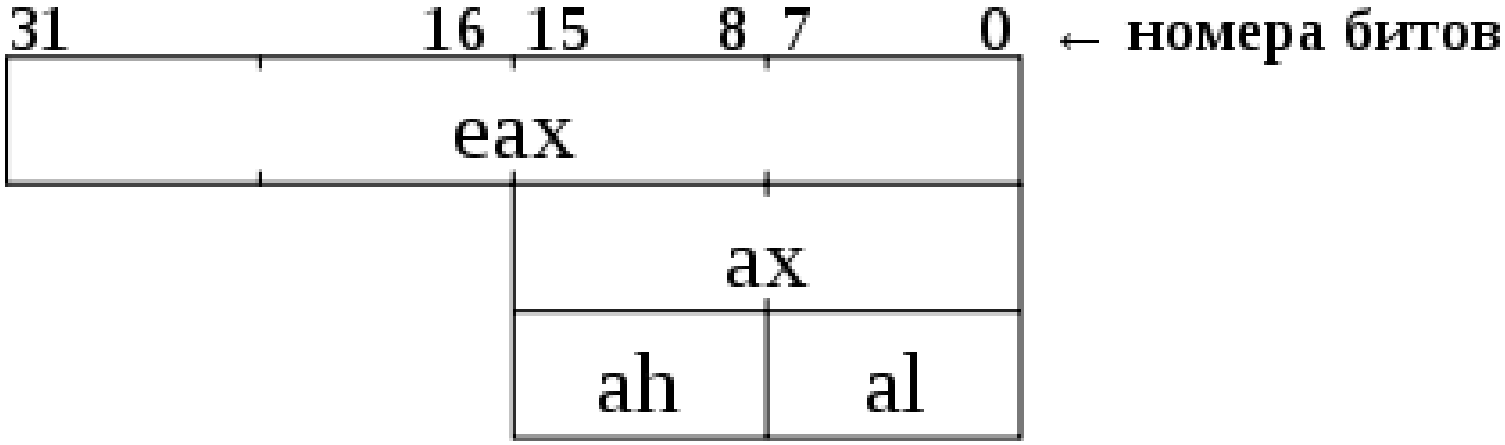
Программирование на ассемблере для x86

- Регистры общего назначения (Размер – 32 бита)
  - %eax: Accumulator register
  - %ebx: Base register
  - %ecx: Counter register
  - %edx: Data register
  - %esp: Stack pointer register
  - %ebp: Base pointer register
  - %esi: Source index register
  - %edi: Destination index register
  - регистры математического сопроцессора
  - регистры MMX, SSE, SSE2, SSE3 etc.

Программирование на ассемблере для x86

- К младшим 16 битам регистра %*eax* можно обратиться как %*ax*
- В свою очередь, %*ax* содержит две однобайтовых части, которые могут использоваться как самостоятельные регистры: старший %*ah* и младший %*al*

Аналогично можно обращаться к
%ebx/%bx/%bh/%bl %ecx/%cx/%ch/%cl
%edx/%dx/%dh/%dl %esi/%si %edi/%di



Программирование на ассемблере для x86

- Сегментные регистры:
  - %*cs*: Code segment
  - %*ds*: Data segment
  - %*ss*: Stack segment
  - %*es*: Extra segment
  - %*fs*: F segment
  - %*gs*: G segment
- ОС предоставляет специальные средства - системный вызов *modify_ldt*()
- Они позволяют описывать нестандартные сегменты и работать с ними
- Однако такая потребность возникает редко

Программирование на ассемблере для x86

- Флаги:
  - cf: *carry flag*, флаг переноса:
    - 1 — во время арифметической операции был произведён перенос из старшего бита результата;
    - 0 — переноса не было;
  - zf: *zero flag*, флаг нуля:
    - 1 — результат последней операции нулевой;
    - 0 — результат последней операции ненулевой;
  - of: *overflow flag*, флаг переполнения:
    - 1 — во время арифметической операции произошёл перенос в/из старшего (знакового) бита результата;
    - 0 — переноса не было;
  - df: *direction flag*, флаг направления. Указывает направление просмотра в строковых операциях:
    - 1 — направление «назад», от старших адресов к младшим;
    - 0 — направление «вперёд», от младших адресов к старшим.

Программирование на ассемблере для x86

- Указатель команды *eip* (*instruction pointer*)
- Размер — 32 бита
- Содержит указатель на следующую команду
- Напрямую недоступен, изменяется неявно командами условных и безусловных переходов, вызова и возврата из подпрограмм

Программирование на ассемблере для x86

```
/* Компилировать: $gcc hello.s -o hello */
.data                           /* поместить следующее в сегмент данных
                                                                    */
hello_str:                      /* наша строка                     */
        .string "Hello, goodbye and farewell!\n"
                                /* длина строки                    */
        .set hello_str_length, . - hello_str - 1
.text                           /* поместить следующее в сегмент кода */
.globl  main                    /* main - глобальный символ, видимый
                                   за пределами текущего файла     */
.type   main, @function         /* main - функция (а не данные)    */
main:
        movl    $4, %eax        /* поместить номер системного вызова
                                   write = 4 в регистр %eax        */
        movl    $1, %ebx        /* первый параметр - в регистр %ebx;
                                   номер файлового дескриптора
                                   stdout - 1                      */
        movl    $hello_str, %ecx  /* второй параметр - в регистр %ecx;
                                     указатель на строку           */
        movl    $hello_str_length, %edx /* третий параметр - в регистр
                                           %edx; длина строки      */
        int     $0x80           /* вызвать прерывание 0x80         */
        movl    $1, %eax        /* номер системного вызова exit - 1   */
        movl    $0, %ebx        /* передать 0 как значение параметра  */
        int     $0x80           /* вызвать exit(0)                 */
        .size   main, . - main    /* размер функции main           */
```

Программирование на ассемблере для x86

- В роли операнда ассемблерной команды могут выступать:
  - Конкретное значение, известное на этапе компиляции, — например, числовая константа или символ
    - Записываются при помощи знака $, например: $0xf1, $10, $*hello_str*
    - Эти операнды называются *непосредственными*
  - Регистр (перед именем регистра ставится знак %, например: %*eax*, %*bx*, %*cl*)
  - Указатель на ячейку в памяти
  - Неявный операнд

Программирование на ассемблере для x86

```
Формат ассемблерной команды:
Операция    Источник, Назначение
Пример:
movl    $4, %eax
```

- Суффикс *l* после *mov* указывает на то, что команда работает с операндами длиной в 4 байта (*long*)
- Все суффиксы:
  - *b* (*byte*) — 1 байт
  - *w* (*word*) — 2 байта
  - *l* (*long*) — 4 байта
  - *q* (*quadword*) — 8 байт
- **Студентам**: как записать значение 41 в регистр %*al*?

Программирование на ассемблере для x86

`смещение(база, индекс, множитель)`

- Вычисленный адрес = база + индекс × множитель + смещение
- Множитель может принимать значения 1, 2, 4 или 8
- Например:

  (%*ecx*) адрес операнда находится в регистре %*ecx*;

  4(%*ecx*) адрес операнда равен %*ecx* + 4;

  -4(%*ecx*) адрес операнда равен %*ecx* – 4;

  kuku(, %*ecx*, 4) адрес операнда равен *kuku* + %*ecx* × 4, где *kuku* — некоторый адрес

Программирование на ассемблере для x86

- Директивы:
  - *.byte* — размещает каждое выражение как 1 байт;
  - *.short* — 2 байта;
  - *.long* — 4 байта;
  - *.quad* — 8 байт.
- Примеры:

```
.byte   0x10, 0xf5, 0x42, 0x55
.long   0xaabbaabb
.short  -123, 456
```

Программирование на ассемблере для x86

- Существуют директивы для размещения в памяти строковых литералов:

.*ascii* "STR" размещает строку STR, нулевых байтов не добавляет

.*string* "STR" размещает строку STR, после которой следует нулевой байт (как в языке *C*)

- У директивы .*string* есть синоним .*asciz* (*z* указывает на добавление нулевого байта)
- Строка-аргумент этих директив может содержать стандартные *escape*-последовательности языка C, например, \n, \r, \t, \\, \" и т.д.

Программирование на ассемблере для x86

- Данные нужно помещать в секцию данных директивой *.data*:

```
.data
        .string "Hello, goodbye and farewell\n"
        ...
```

- Если некоторые данные не предполагается изменять в ходе выполнения программы, их можно поместить в специальную секцию данных только для чтения директивой *.section .rodata*:

```
.section .rodata
        .string "program version 0.419"
```

Программирование на ассемблере для x86

- Для соблюдения выравнивания в распоряжении программиста есть директива *.p2align*

```
.p2align степень_двойки, заполнитель, максимум
```

- Второй и третий аргументы – не обязательны
- Примеры:

```
.data
        .string "Hello, goodbye and farewell\n"
/* мы вряд ли захотим считать, сколько символов
  занимает эта строка, и является ли следующий адрес
  выровненным                                    */
        .p2align 2          /* выравниваем по
  границе 4 байта    для следующего .long          */
        .long 123456
```

Программирование на ассемблере для x86

- Мы не присвоили имён нашим данным
- Как же к ним обращаться?
- Ответ: нужно поставить метку
- **Метка** — это константа, значение которой — адрес

```
hello_str:
        .string "Hello, goodbye and farewell!\n"
```

- А если нам нужна константа с каким-то другим значением?
- Тогда мы приходим к более общему понятию «символа»

Программирование на ассемблере для x86

- Просмотреть символы в объектном модуле можно, используя утилиту *nm*
- Например:

```
$ nm hello.o
00000000 d hello_str
0000000e a hello_str_length
00000000 T main
$
```

Программирование на ассемблере для x86

- Для создания нового символа используется директива *.set*
- Синтаксис:

```
.set    символ, выражение
```

- Определим символ *kookoo* = 42:

```
.set kookoo, 42
```

- Ещё пример из *hello.s*:

```
hello_str:
        .string "Hello, goodbye and farewell!\n"
                /* наша строка  */
        .set    hello_str_length, . - hello_str - 1
/* длина строки */
```

Программирование на ассемблере для x86

- Для неинициализированных данных используются специальные директивы:

```
.space  количество_байт
.space  количество_байт, заполнитель
```

- Директива *.space* резервирует *количество_байт* байт.
- Можно использовать для размещения инициализированных данных, для этого существует параметр заполнитель — этим значением будет инициализирована память, например:

```
.bss
long_var_1:             /* по размеру как .long                    */
        .space 4

buffer:                 /* какой-то буфер в 1024 байта             */
        .space 1024

struct:                 /* какая-то структура размером 20 байт     */
        .space 20
```

Программирование на ассемблере для x86

```
.data
some_var:
        .long 0x00000072

other_var:
        .long 0x00000001, 0x00000002, 0x00000003

.text
.globl main
main:
        movl  $0x48, %eax       /* поместить число 0x00000048 в %eax */

        movl  $some_var, %eax   /* поместить в %eax значение метки
                                   some_var, то есть адрес числа в
                                   памяти; например, у нас
                                   содержимое %eax равно 0x08049589  */

        movl  some_var, %eax    /* обратиться к содержимому переменной;
                                   в %eax теперь 0x00000072          */

        movl  other_var + 4, %eax  /* other_var указывает на 0x00000001
                                   размер одного значения типа long — 4
                                   байта; значит, other_var + 4
                                   указывает на 0x00000002;
                                   в %eax теперь 0x00000002          */
```

Программирование на ассемблере для x86

```
movl  $1, %ecx          /* поместить число 1 в %ecx          */

movl  other_var(,%ecx,4), %eax  /* поместить в %eax первый
                                   (нумерация с нуля) элемент массива
                                   other_var, пользуясь %ecx как
                                   индексным регистром              */

movl  $other_var, %ebx  /* поместить в %ebx адрес массива
                                   other_var                        */

movl  4(%ebx), %eax     /* обратиться по адресу %ebx + 4;
                                   в %eax снова 0x00000002          */

movl  $other_var + 4, %eax  /* поместить в %eax адрес, по
                                   которому расположен 0x00000002
                                   (адрес массива плюс 4 байта --
                                   пропустить нулевой элемент)      */

movl  $0x15, (%eax)     /* записать по адресу "то, что записано
                                   в %eax" число 0x00000015         */
```

Программирование на ассемблере для x86

```
inc   операнд
dec   операнд
neg   операнд
add   источник, приёмник
sub   источник, приёмник
mul   множитель_1
imul  множитель_1
div   делитель
idiv  делитель
```

Команда	Второй сомножитель	Результат
mulb	%al	16 бит: %ax
mulw	%ax	32 бита: младшая часть в %ax, старшая в %dx
mull	%eax	64 бита: младшая часть в %eax, старшая в %edx

Программирование на ассемблере для x86

- Примеры:

```
.text
        movl   $72, %eax
        incl   %eax                 /* в %eax число 73 */
        decl   %eax                 /* в %eax число 72 */

        movl   $48, %eax
        addl   $16, %eax            /* в %eax число 64 */

        movb   $5, %al
        movb   $5, %bl
        mulb   %bl              /* в регистре %ax
                                   произведение %al × %bl = 25 */
```

Программирование на ассемблере для x86

```
        movb  $0,   %ah          /* %ah = 0                              */
        movb  $252, %al          /* %al = 252                            */
        addb  $8,  %al          /* %al = %al + 8
                                    происходит переполнение,
                                    устанавливается флаг cf;
                                    в %al число 4                        */
        jnc   no_carry           /* если переполнения не было, перейти
                                    на метку                             */
        movb  $1,   %ah          /* %ah = 1                              */
no_carry:
        /* %ax = 260 = 0x0104 */
```

Программирование на ассемблере для x86

- Сравнение:

```
cmp   операнд_2, операнд_1
```

- Переход:

```
jcc   метка
```

- Варианты:

Мнемоника	Английское слово	Смысл	Тип
e	equal	равенство	любые
n	not	инверсия условия	любые
g	greater	больше	со знаком
l	less	меньше	со знаком
a	above	больше	без знака
b	below	меньше	без знака

jmp – безусловный переход

Программирование на ассемблере для x86

```
.text
        /* Тут пропущен код, который получает некоторое значение в
   %eax.
           Пусть нас интересует случай, когда %eax = 15 */

        cmpl  $15, %eax          /* сравнение */

        jne   not_equal         /* если операнды не равны, перейти
                                    на метку not_equal */
         /* сюда управление перейдёт только в случае,
            когда переход не сработал, а значит, %eax = 15 */

not_equal:
        /* а сюда управление перейдёт в любом случае */
```

Память

- Число 0x04030201 можно записать в виде байтовой последовательности:
  - начиная со старшего байта: 0x04 0x03 0x02 0x01 — *big-endian*
  - начиная с младшего байта: 0x01 0x02 0x03 0x04 — *little-endian*
- Адрес всего слова в памяти – адрес первого байта последовательности

Методы адресации

- Пространство памяти предназначено для хранения кодов команд и данных, для доступа к которым имеется богатый выбор методов адресации
- Операнды могут находиться во внутренних регистрах процессора
- Они могут располагаться в системной памяти
- Они могут находиться в устройствах ввода/вывода
- Определение местоположения операндов производится кодом команды

Методы адресации

- Прямая или абсолютная адресация
- Физический адрес операнда содержится в адресной части команды, формально:
- Операнд$_i$ = (A_i)
- A_i – код, содержащийся в i-м адресном поле команды.

```
.data
num:
        .long    0x12345678

.text
main:
        movl    (num), %eax    /* Записать в регистр %eax операнд,
                                  который содержится в оперативной
                                  памяти по адресу метки num          */

        addl    (num), %eax    /* Сложить с регистром %eax операнд,
                                  который содержится в оперативной
                                  памяти по адресу метки num и записать
                                  результат в регистр %eax            */

        ret
```

Методы адресации

- Непосредственная адресация
- В команде содержится не адрес операнда, а непосредственно сам операнд
- Операнд$_i$ = A$_i$

```
.text
main:
        movl    $0x12345, %eax        /* загрузить константу 0x12345 в
                                         регистр %eax.                 */
```

Методы адресации

- Косвенная (базовая) адресация
- Адресная часть команды указывает адрес ячейки памяти или регистр, в котором содержится адрес операнда
- Операнд$_i$ = ((A_i))

```
.data
num:
        .long   0x1234

.text
main:
        movl    $num, %ebx      /* записать адрес метки в регистр
                                   адреса %ebx                      */

        movl    (%ebx), %eax    /* записать в регистр %eax операнд из
                                   оперативной памяти, адрес которого
                                   находится в регистре адреса %ebx  */
```

Методы адресации

- Регистровая адресация
- Предполагается, что операнд находится во внутреннем регистре процессора

```
.text
main:
        movl    $0x12345, %eax  /* записать в регистр константу 0x12345
                                                                   */
        movl    %eax, %ecx      /* записать в регистр %ecx операнд,
                                   который находится в регистре %eax */
```

Методы адресации

- Относительная адресация
- Используется тогда, когда память логически разбивается на блоки — **сегменты**

$Операнд_i = (база_i + смещение_i)$

- Пример 1
  - адресное поле команды состоит из двух частей, в одной указывается номер регистра, хранящего базовое значение адреса, а в другой задается смещение, определяющее положение ячейки относительно начала сегмента
  - Это собственно относительная адресация

Методы адресации

- Относительная адресация
- Используется тогда, когда память логически разбивается на блоки — **сегменты**
- $Операнд_i = (база_i + смещение_i)$
- Пример 2
  - Первая часть адресного поля команды также определяет номер базового регистра, а вторая содержит номер регистра, в котором находится смещение
  - Это базово-индексная адресация

Команды для адресов и стека

```
lea  источник, назначение


.data
some_var:
        .long 0x00000072

.text
        leal  0x32, %eax        /* аналогично movl $0x32, %eax       */
        leal  some_var, %eax    /* аналогично movl $some_var, %eax   */

        leal  $0x32, %eax       /* вызовет ошибку при компиляции,
                                   так как $0x32 - непосредственное
                                   значение                          */
        leal  $some_var, %eax   /* аналогично, ошибка компиляции:
                                   $some_var - это непосредственное
                                   значение, адрес                   */

        leal  4(%esp), %eax     /* поместить в %eax адрес предыдущего
                                   элемента в стеке;
                                   фактически, %eax = %esp + 4       */
```

Команды для адресов и стека

```
Вычисленный адрес = база + индекс × множитель + смещение

movl  $10, %eax
movl  $7, %ebx

leal  5(%eax)        ,%ecx  /* %ecx = %eax + 5 = 15                    */
leal  -3(%eax)       ,%ecx  /* %ecx = %eax - 3 = 7                     */
leal  (%eax,%ebx)    ,%ecx  /* %ecx = %eax + %ebx × 1 = 17             */
leal  (%eax,%ebx,2)  ,%ecx  /* %ecx = %eax + %ebx × 2 = 24             */
leal  1(%eax,%ebx,2),%ecx   /* %ecx = %eax + %ebx × 2 + 1 = 25         */
leal  (,%eax,8)      ,%ecx  /* %ecx = %eax × 8 = 80                    */
leal  (%eax,%eax,2)  ,%ecx  /* %ecx = %eax + %eax × 2 = %eax × 3 = 30  */
leal  (%eax,%eax,4)  ,%ecx  /* %ecx = %eax + %eax × 4 = %eax × 5 = 50  */
leal  (%eax,%eax,8)  ,%ecx  /* %ecx = %eax + %eax × 8 = %eax × 9 = 90  */
```

Команды для адресов и стека

- Две специальные команды: *push* (поместить в стек) и *pop* (извлечь из стека)

- Синтаксис:

```
push   источник
pop    назначение
```

- **Студентам**: как отреагирует компилятор, если задать команду *pushb* 0x34?

Команды для адресов и стека

```
.text
        pushl $0x10             /* поместить в стек число 0x10       */
        pushl $0x20             /* поместить в стек число 0x20       */
        popl  %eax              /* извлечь 0x20 из стека и записать в
                                   %eax                              */
        popl  %ebx              /* извлечь 0x10 из стека и записать в
                                   %ebx                              */

        pushl %eax              /* странный способ сделать           */
        popl  %ebx              /* movl %eax, %ebx                   */

        movl  $0x00000010, %eax
        pushl %eax              /* поместить в стек содержимое %eax  */
        popw  %ax               /* извлечь 2 байта из стека и
                                   записать в %ax                    */
        popw  %bx               /* и ещё 2 байта и записать в %bx    */
                                /* в %ax находится 0x0010, в %bx
                                   находится 0x0000; такой код сложен
                                   для понимания, его следует избегать
                                                                     */

        pushl %eax              /* поместить %eax в стек; %esp
                                   уменьшится на 4                   */
        addl  $4, %esp          /* увеличить %esp на 4; таким образом,
                                   стек будет приведён в исходное
                                   состояние                         */
```

Подпрограммы: общие принципы

- **Вызов подпрограммы** - это передача управления по адресу начала подпрограммы с одновременным запоминанием в стеке адреса возврата
- Адрес возврата - адрес команды, непосредственно следующей за командой вызова)
- Процессоры семейства x86_32 предусматривают для этой цели команду *call*
- Она похожа на *jmp*
- Возврат из подпрограммы производится командой *ret*

Подпрограммы: общие принципы

```
.data
printf_format:   /* форматная строка для вывода заполненного байтового массива */
        .string "%s\n"

.bss
array:  .space 80 /* Байтовый массив для заполнения */

.text
.globl main
main:
        mov $array, %edi
        movl $80, %ecx
        movb $0x41, %al
        call fill_memory
/*
 * следующий код выводит число в %eax на экран и завершает программу
 */
        pushl $array
        pushl $printf_format
        call  printf
        addl  $8, %esp

        movl $0, %eax
        ret
```

Подпрограммы: общие принципы

```
/* Подпрограмма fill memory
 * (edi = адрес, ecx = размер, al = байт-
  заполнитель)
 */
fill_memory:
        jecxz fm_q
fm_lp:
        movb %al, (%edi)
        incl %edi
        loop fm_lp
fm_q:   ret
```

Организация стековых фреймов

- Обычно параметры в подпрограммы передаются через стек
- Там же размещают локальные переменные
- Параметры в стеке размещает вызывающая программа
- Затем при вызове подпрограммы в стек заносится адрес возврата, а после этого сама вызванная подпрограмма резервирует в стеке место под локальные переменные
- Все это в совокупности образует **стековый фрейм**

Организация стековых фреймов

- Если в стек занести три четырехбайтных параметра, а потом вызвать подпрограмму, то адрес возврата будет лежать в памяти по адресу (%*esp*), а параметры окажутся доступными по адресам 4(%*esp*), 8(%*esp*) и 12(%*esp*)
- Если разместить в стеке локальные 4-байтные переменные, то они окажутся доступными по адресам: -4(%*esp*), -8(%*esp*) и т.д.

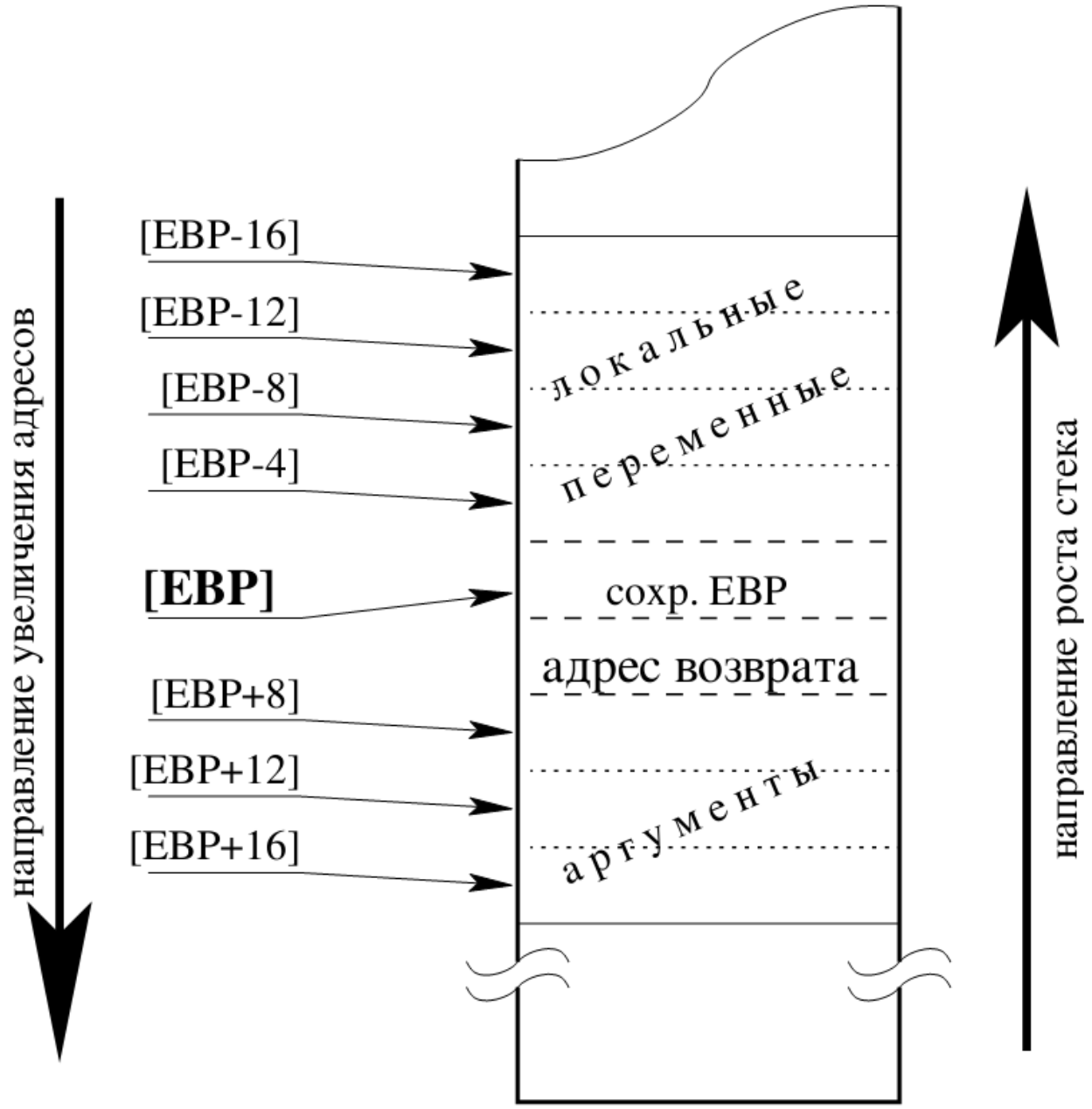
Организация стековых фреймов

- Первым своим действием подпрограмма обычно сохраняет значение регистра %*esp* в каком-то другом регистре, обычно %*ebp*
- Он используется для доступа к параметрам и локальным переменным
- В то же время %*esp* продолжает играть роль указателя стека, изменяясь по мере необходимости
- Перед возвратом из подпрограммы его обычно восстанавливают в исходном значении за счет пересылки в него значения из %*ebp*, чтобы оно снова указывал на адрес возврата

Организация стековых фреймов

- Каждая подпрограмма должна сама сохранять старое значение %*ebp* и восстановить его перед возвратом управления
- Для сохранения значения %*ebp* тоже используется стек (*push %ebp* сразу после получения управления)
- Старое значение %*ebp* помещается в стек непосредственно после адреса возврата из подпрограммы
- Следующая команда – *movl %esp, %ebp*

Организация стековых фреймов



Организация стековых фреймов

```
/* пролог */
pushl %ebp
movl %esp, %ebp
subl $16, %esp


/* эпилог */
movl %ebp, %esp
popl %ebp
ret
```

See also

- Магда, Ю.С. Ассемблер для процессоров Intel Pentium/ Ю.С. Магда. – СПб.: Питер, 2006. – 416 с.
- Робачевский, А. Операционная система Unix, 2 изд./ А.Робачевский, С.Немнюгин, О.Стесик. – СПб.: БХВ-Петербург, 2010. – 656 с.
- Столяров, А.В. Программирование на языке ассемблера NASM для ОС UNIX: учеб.пособие. – М.: Макс, 2011. – 188 с. – Доступ: http://www.stolyarov.info/books/asm_unix
- flat assembler - http://www.flatassembler.net/
- The Netwide Assembler: NASM - http://nasm.us/
- Intel® 64 and IA-32 Architectures Optimization Reference Manual, 3.5.1.3 Using LEA.
- Intel® 64 and IA-32 Architectures Software Developer's Manual, 4.1 Instructions (N-Z), PUSH
- Ассемблер в Linux для программистов С – http://ru.wikibooks.org/wiki/Ассемблер_в_Linux_для_програ