Компьютерные основы программирования
Представление программ
часть1

Лекция 5, 16 марта 2017

Лектор: Чуканова Ольга Владимировна

Кафедра информатики

602 АК

ovcha@mail.ru

Машинный уровень

- Краткая история изделий Интел
- Си, ассемблер, машинный код
- Основы ассемблера: регистры, операнды, пересылки
- Немного об x86-64
- Полная адресация, вычисление адреса (leal)
- Арифметические операции
- Управление: флаги условий
- Условные переходы и пересылки
- Циклы
- Операторы переключения
- Процедуры IA 32
  - Структура стека
  - Соглашения вызова процедур
  - Рекурсия и указатели

Целочисленные регисты (IA32)

Форматы данных

Описание в C	Тип данных Intel	Суффикс GAS	Размер (в байтах)
char	Байт	b	1
short	Слово	w	2
int	Двойное слово	l	4
unsigned	Двойное слово	l	4
long int	Двойное слово	l	4
unsigned long	Двойное слово	l	4
char *	Двойное слово	l	4
float	Одинарная точность	t	4
double	Двойная точность	l	8
long double	Повышенная точность	t	10/12

Формы операндов

Тип	Форма	Значение операнда	Название
Непосредств.	*$Imm*	*Imm*	Непосредственное
Регистр	E_a	R[E_a]	Регистр
Память	*Imm*	M[*Imm*]	Абсолютное
Память	(E_a)	M[R[E_a]]	Косвенное
Память	*Imm*(E_b)	M[*Imm*] + R[E_b]]	База + смещение
Память	(E_b, E_i)	M[R[E_b]] + R[E_i]]	Индексированное
Память	*Imm*(E_b, E_i)	M[*Imm* + R[E_b]] + R[E_i]]	Индексированное
Память	(, E_i, s)	M[R[E_i]·s]	Нормированно-идексированное
Память	*Imm*(, E_i, s)	M[*Imm* + R[E_i]·s]	Нормированно-идексированное
Память	(E_b, E_i, s)	M[R[E_b] + R[E_i]·s]	Нормированно-идексированное
Память	*Imm*(E_b, E_i, s)	M[*Imm* + R[E_b] + R[E_i]·s]	Нормированно-идексированное

Memory Operands.

Операнды памяти, как замечено выше отличаются также. В синтаксисе Intel индексный регистр в '[' и']', тогда как в синтаксисе AT&T в '(' и')'.

Example:

Intel Syntax		AT&T Syntax	
mov	eax,[ebx]	movl	(%ebx),%eax
mov	eax,[ebx+3]	movl	3(%ebx),%eax

Форма AT&T для инструкций, включающих сложные операции, очень неясен по сравнению с синтаксисом Intel. Форма синтаксиса Intel - [base+index*scale+ offset]. Форма синтаксиса AT&T - смещение (база, индекс, масштаб).

Непосредственные используемые данные не должны , иметь префикс '$' в AT&T, когда используется для адресации

Example:

Intel Syntax		AT&T Syntax	
instr	[base+index*scale+offset]	instr	offset(base,index,scale),foo
mov	eax,[ebx+20h]	movl	0x20(%ebx),%eax
add	eax,[ebx+ecx*2h]	addl	(%ebx,%ecx,0x2),%eax
lea	eax,[ebx+ecx]	leal	(%ebx,%ecx),%eax
sub	eax,[ebx+ecx*4h-20h]	subl	-0x20(%ebx,%ecx,0x4),%eax

As you can see, AT&T is very obscure. [base+index*scale+disp] makes more sense at a glance than disp(base,index,scale).

Как Вы видите, AT&T очень неясен. [base+index*scale+disp] имеет больше смысла сразу, чем смещение (основа, индекс, масштаб).

Перенос данных: IA32

- **Перенос данных**

  `movl` ***Источник, Результат*:**
  mov Результат, Источник

- **Типы операндов**
  - ***Непосредственные:*** константные
    - Пример: **`$0x400, $-533`**
    - Как Си константы, но с префиксом '**`$`**'
    - Кодируются в 1-м, 2-мя, или 4-мя байтами
  - ***Регистровые:*** только 8 целочисленных
    - Пример: **`%eax, %edx`**
    - **`%esp`** и **`%ebp`** зарезервированы
    - Другие особо используются некоторыми командами
  - ***В памяти:*** 4 последовательных байта памяти по адресу, находящемуся в регистре
    - Простейший пример: **`(%eax)`**
    - Несколько различных "вариантов адресации"

`%eax`
`%ecx`
`%edx`
`%ebx`
`%esi`
`%edi`
`%esp`
`%ebp`

`movl` Комбинации операндов

	Откуда	Куда	Src,Dest	Си аналог
`movl`	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

Нельзя передать из памяти в память одной командой

Простые адресации памяти

- **Базовая (R) Mem[Reg[R]]**
  - Регистр **R** содержит адрес памяти

```
movl  (%ecx),%eax
```

```
mov     eax, [ecx]
```

- **Базовая со смещением D(R) Mem[Reg[R]+D]**
  - Регистр **R** содержит адрес начала фрагмента памяти
  - Константа **D** обозначает сдвиг от начала фрагмента

```
movl  8(%ebp),%edx
```

```
mov  edx, [ebp + 8]
```

```
1    movl    $0x4050,%eax          Непосредственное значение - Регистр
2    movl    %ebp,%esp             Регистр – Регистр
3    movl    (%edi,%ecx),%eax      Память - Регистр
4    movl    $-17,(%esp)           Непосредственное значение - Память
5    movl    %eax,-12(%ebp)        Регистр - Память
```

```
mov  eax, 4050h
mov  esp, ebp
mov  eax, [edi+ecx]
mov [esp], -17
mov  [ebp – 12], eax
```

Использование простых адресаций

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  pushl %ebp
  movl  %esp,%ebp
  pushl %ebx

  movl  8(%ebp), %edx
  movl  12(%ebp), %ecx
  movl  (%edx), %ebx
  movl  (%ecx), %eax
  movl  %eax, (%edx)
  movl  %ebx, (%ecx)

  popl  %ebx
  popl  %ebp
  ret
```

Пролог

Тело

Эпилог

```
mov  edx, dword ptr [ebp + 8]
mov  ecx, dword ptr [ebp + 12]
mov  ebx,  dword ptr [edx]
mov  eax,  dword ptr [ecx]
mov  dword ptr [edx], eax
mov  dword ptr [ecx],  ebx
```

Разбираем Swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

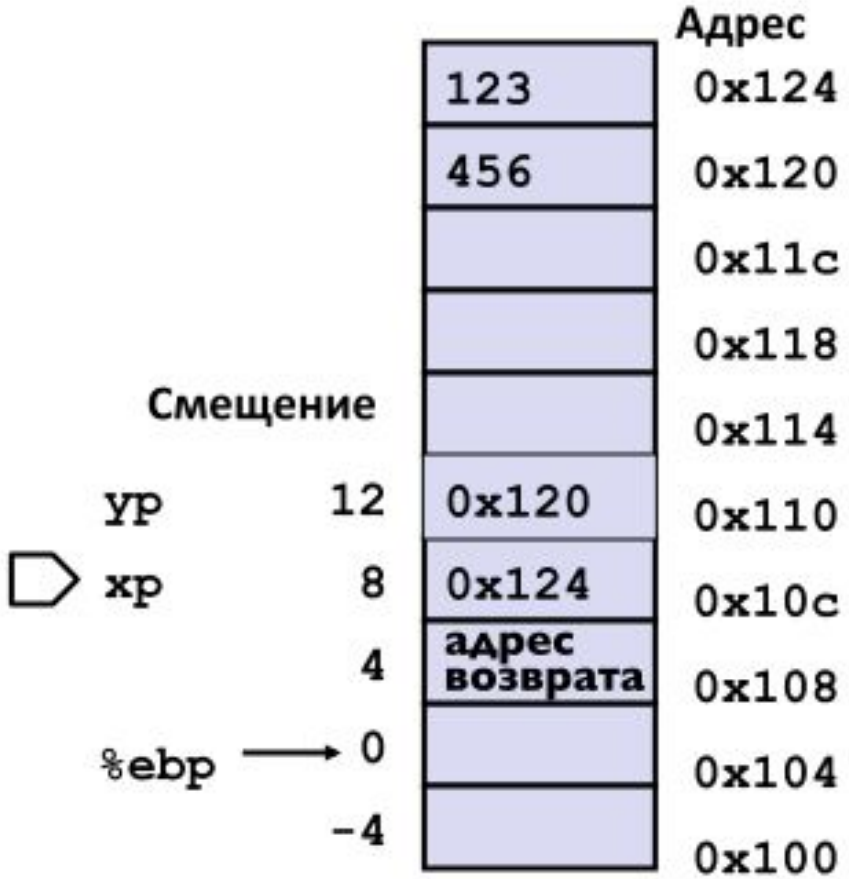
Регистр	Значение
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

```
movl  8(%ebp), %edx    # edx = xp
movl  12(%ebp), %ecx   # ecx = yp
movl  (%edx), %ebx     # ebx = *xp (t0)
movl  (%ecx), %eax     # eax = *yp (t1)
movl  %eax, (%edx)     # *xp = t1
movl  %ebx, (%ecx)     # *yp = t0
```

Разбираем Swap

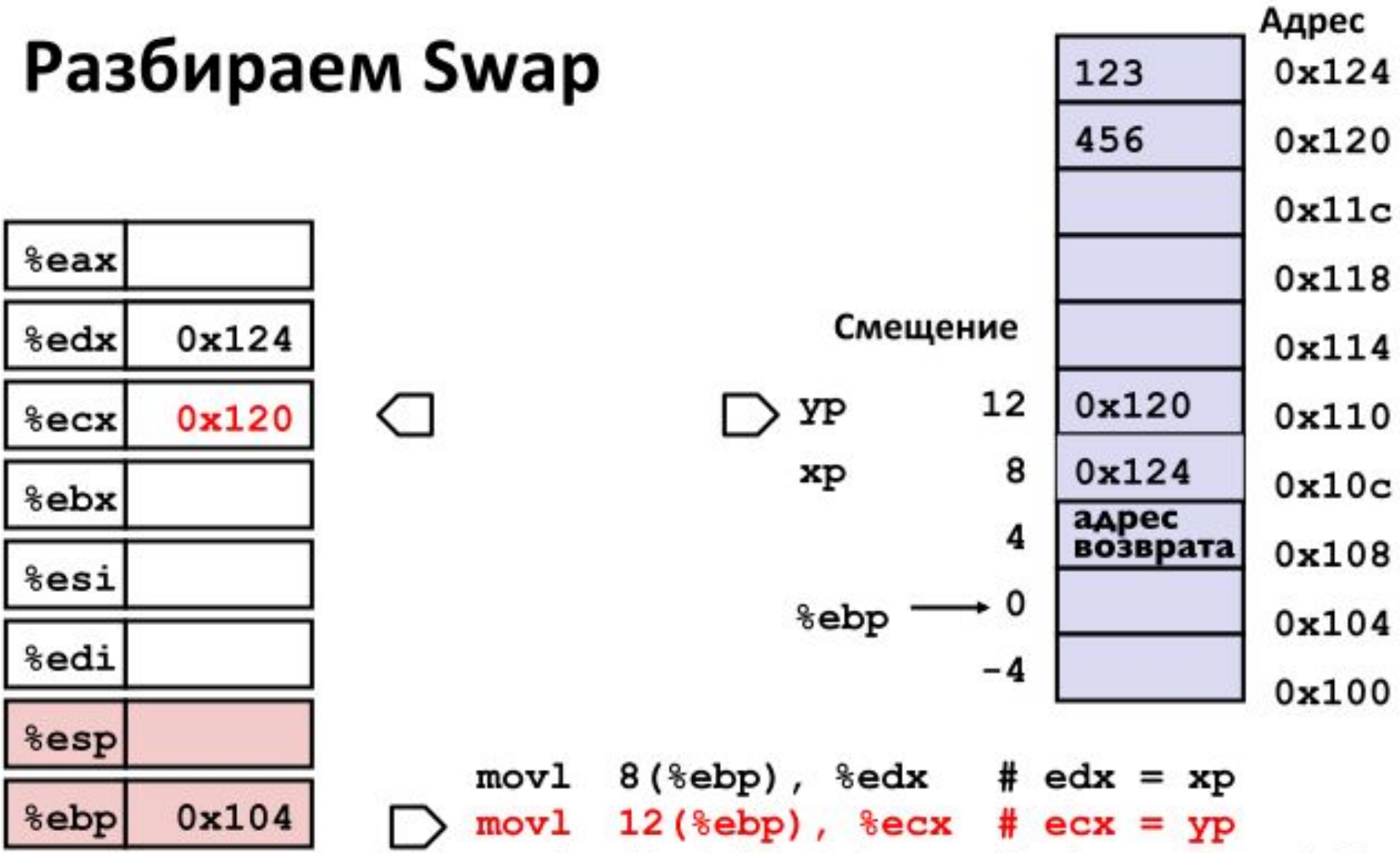

```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

Разбираем Swap




```
movl  8(%ebp), %edx    # edx = xp
movl  12(%ebp), %ecx   # ecx = yp
movl  (%edx), %ebx     # ebx = *xp (t0)
movl  (%ecx), %eax     # eax = *yp (t1)
movl  %eax, (%edx)     # *xp = t1
movl  %ebx, (%ecx)     # *yp = t0
```

Разбираем Swap

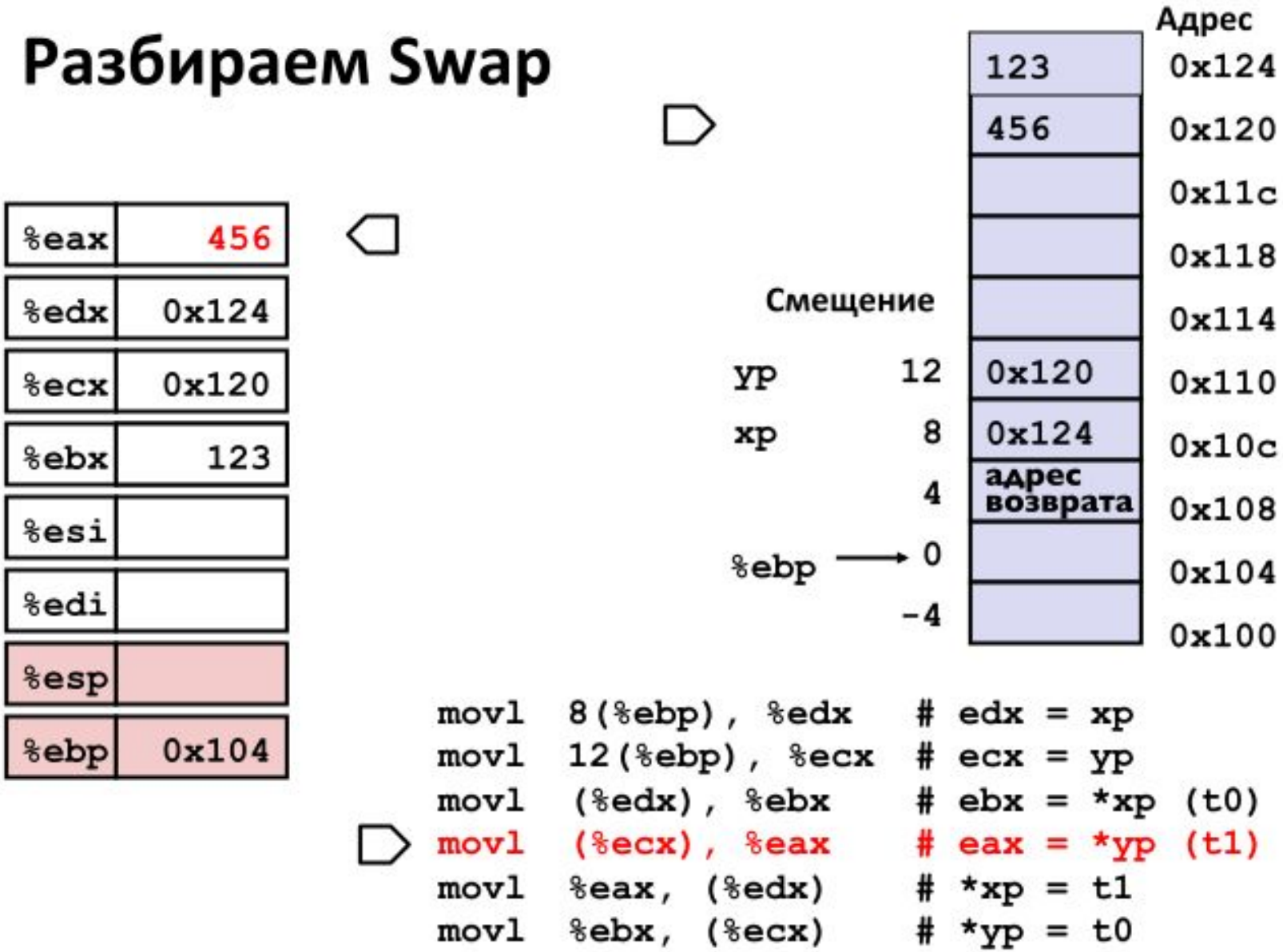



```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

Разбираем Swap


```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

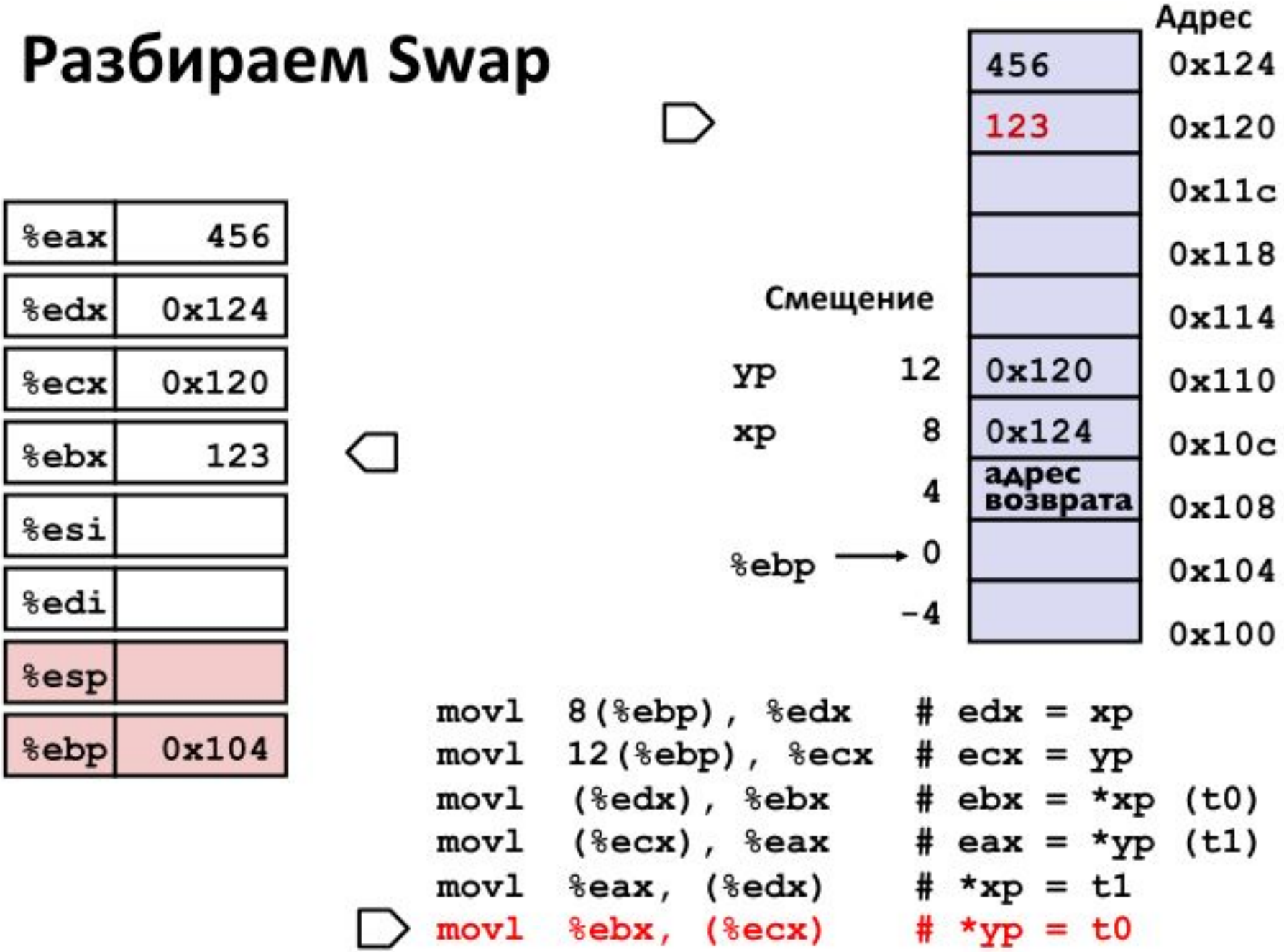
Разбираем Swap



Разбираем Swap


```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

Разбираем Swap




```
movl   8(%ebp), %edx    # edx = xp
movl   12(%ebp), %ecx   # ecx = yp
movl   (%edx), %ebx     # ebx = *xp (t0)
movl   (%ecx), %eax     # eax = *yp (t1)
movl   %eax, (%edx)     # *xp = t1
movl   %ebx, (%ecx)     # *yp = t0
```

Полная адресация

- **Наиболее общая форма**

  D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  - **D**: 1-, 2-, or 4-байтное константное "смещение"
  - **Rb**: Базовый регистр: любой из 8 целочисленных регистров
  - **Ri**: Индексный регистр: любой, кроме **`%esp`**
    - Также нежелательно использовать **`%ebp`**
  - **S**: Масштаб: 1, 2, 4, или 8 (*а почему эти числа?*)

- **Специальные случаи**

  (Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]]**

  D(Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]+D]**

  (Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]]**

Целочисленные регистры x86-64

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Расширены существующие регистры. Добавлены 8 новых.
- **%ebp/%rbp** сделан регистром общего назначения

Команды

- **Длинное слово `l` (4 байта) ↔ Четверное слово `q` (8 байт)**

- **Новые инструкции:**
  - **`movl` → `movq`**
  - **`addl` → `addq`**
  - **`sall` → `salq`**
  - и т.п.

- **32-битные инструкции выдают 32-битный результат**
  - Устанавливают биты старшей половины регистра-результата в `0`
  - Пример: **`addl`**

32-битный код swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
  pushl %ebp
  movl  %esp,%ebp
  pushl %ebx

  movl  8(%ebp), %edx
  movl  12(%ebp), %ecx
  movl  (%edx), %ebx
  movl  (%ecx), %eax
  movl  %eax, (%edx)
  movl  %ebx, (%ecx)

  popl  %ebx
  popl  %ebp
  ret
```

Пролог

Тело

Эпилог

64-битный код swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
                                    } Пролог
movl   (%rdi), %edx
movl   (%rsi), %eax
movl   %eax, (%rdi)                 } Тело
movl   %edx, (%rsi)
ret                                 } Эпилог
```

- **Операнды переместились в регистры (а в чём польза?)**
  - Первый (**xp**) в **%rdi**, второй (**yp**) в **%rsi**
  - 64-битные указатели
- **Не требуются операции со стеком**
- **32-битные данные**
  - Данные в регистрах **%eax** and **%edx**
  - Операции **movl**

64-битный код long int swap

```
void swap(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap_1:
                                    } Пролог
movq    (%rdi), %rdx
movq    (%rsi), %rax                } Тело
movq    %rax, (%rdi)
movq    %rdx, (%rsi)
ret                                 } Эпилог
```

- **64-битные данные**
  - Данные в регистрах **%rax** и **%rdx**
  - операции **movq**
    - "**q**" означает «четверное слово»

Полная адресация

- **Наиболее общая форма**

  D(Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]+ D]**

  - **D**: 1-, 2-, or 4-байтное константное "смещение"
  - **Rb**: Базовый регистр: любой из 8 целочисленных регистров
  - **Ri**: Индексный регистр: любой, кроме **`%esp`**
    - Также нежелательно использовать **`%ebp`**
  - **S**: Масштаб: 1, 2, 4, или 8 (*а почему эти числа?*)

- **Специальные случаи**

  (Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]]**

  D(Rb,Ri) **Mem[Reg[Rb]+Reg[Ri]+D]**

  (Rb,Ri,S) **Mem[Reg[Rb]+S*Reg[Ri]]**

Предположим, что следующие значения хранятся в памяти по указанным адресам и в регистрах:

Адрес	Значение
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Регистр	Значение
%eax	0x100
%ecx	0x1
%edx	0x3

Заполните следующую таблицу, показав значения указанных операндов:

Операнд	Значение
%eax	0x100
0x104	0xAB
$0x108	0x108
(%eax)	0xFF
4(%eax)	0xAB
9(%eax,%edx)	0x11
260(%ecx,%edx)	0x13
0xFC(,%ecx,4)	0xFF
(%eax,%edx,4)	0x11

Команда вычисления адреса

- **`leal Src,Dest`**
  - Src адресное выражение
  - Присваивает Dest адрес вычисленный по выражению

```
lea Dest, Scr
```

- **Используется**
  - Для вычисления адресов без обращения к памяти
    - например, трансляции **`p = &x[i];`**
  - Для вычисления выражений вида **x + k*y**,
    - где k = 1, 2, 4, или 8

- **Пример**

```
int mul12(int x)
{
  return x*12;
}
```

Результат компиляции в ассемблер:

```
leal (%eax,%eax,2), %eax  ;t <- x+x*2
sall $2, %eax             ;return t<<2
```

```
lea eax, [eax+eax*2]
sal eax, 2
```

```
mov ax,word ptr [bp+6]
    mov dx,12
    imul  dx
```

Некоторые арифметические команды

- **Двухоперандные команды:**

Формат		Вычисления	
addl	Src,Dest	Dest = Dest + Src	
subl	Src,Dest	Dest = Dest - Src	
imull	Src,Dest	Dest = Dest * Src	
sall	Src,Dest	Dest = Dest << Src	Синоним: **shll**
sarl	Src,Dest	Dest = Dest >> Src	Арифметический
shrl	Src,Dest	Dest = Dest >> Src	Логический
xorl	Src,Dest	Dest = Dest ^ Src	
andl	Src,Dest	Dest = Dest & Src	
orl	Src,Dest	Dest = Dest | Src	

- **Следите за порядком аргументов!**
- **Не различаются signed и unsigned int (почему?)**

```
add Dest,Scr
sub Dest,Scr
imul Dest,Scr
sal (синоним shl)
sar
shr
xor
and
or
inc  Dest      Dest=Dest+1
dec Dest       Dest=Dest-1
```

Пример арифметических выражений

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
arith:
  pushl  %ebp
  movl   %esp, %ebp

  movl   8(%ebp), %ecx
  movl   12(%ebp), %edx
  leal   (%edx,%edx,2), %eax
  sall   $4, %eax
  leal   4(%ecx,%eax), %eax
  addl   %ecx, %edx
  addl   16(%ebp), %edx
  imull  %edx, %eax

  popl   %ebp
  ret
```

Пролог

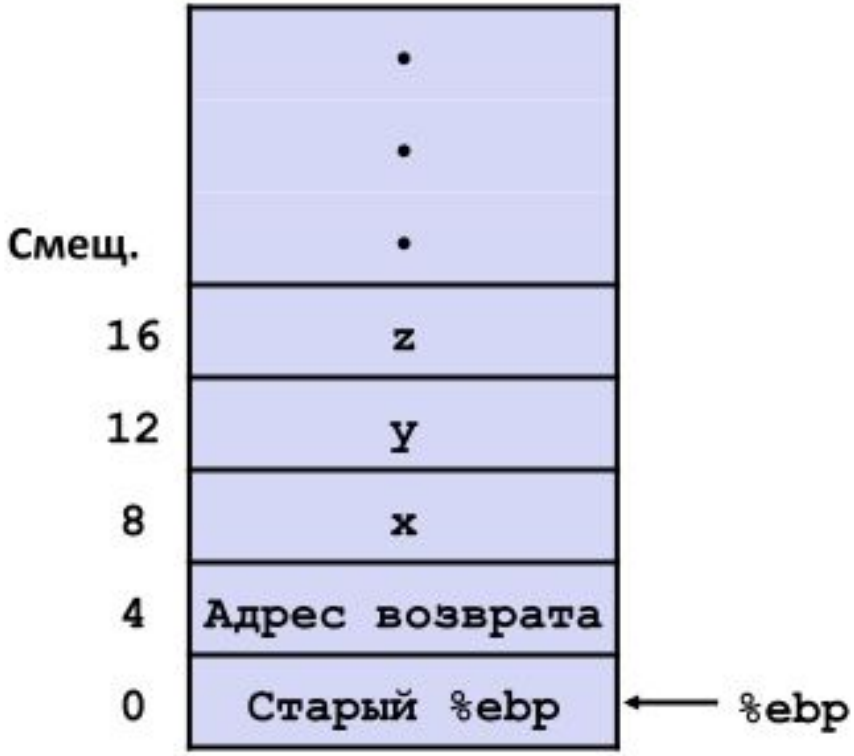
Тело

Эпилог

```
mov   ecx, [ebp + 8]
mov   edx, [ebp +12]
lea     eax, [edx + edx * 2]
sal     eax, 4
lea     eax, [ecx + eax + 4]
add     edx, ecx
imul    eax, edx
```

Разбираем arith

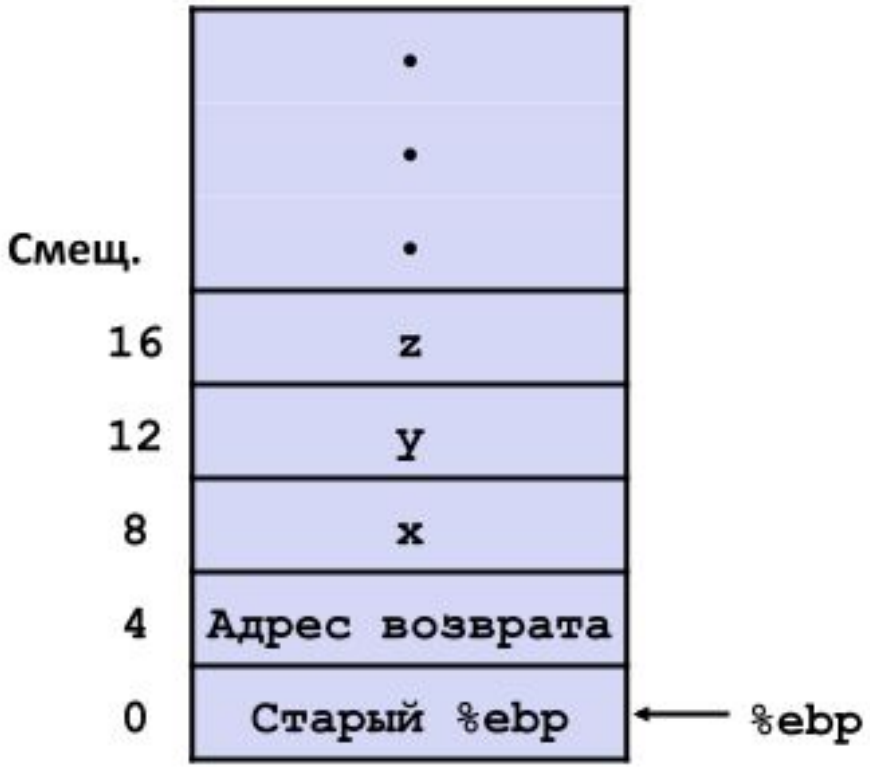
```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```




```
movl   8(%ebp), %ecx          # ecx = x
movl   12(%ebp), %edx         # edx = y
leal   (%edx,%edx,2), %eax    # eax = y*3
sall   $4, %eax               # eax *= 16 (t4)
leal   4(%ecx,%eax), %eax     # eax = t4 +x+4 (t5)
addl   %ecx, %edx             # edx = x+y (t1)
addl   16(%ebp), %edx         # edx += z (t2)
imull  %edx, %eax             # eax = t2 * t5 (rval)
```

Разбираем arith

```
int arith(int x, int y, int z)
{
1  int t1 = x+y;
2  int t2 = z+t1;
3  int t3 = x+4;
4  int t4 = y * 48;
3  int t5 = t3 + t4;
5  int rval = t2 * t5;
   return rval;
}
```




```
   movl   8(%ebp), %ecx          # ecx = x
   movl   12(%ebp), %edx         # edx = y
4  leal   (%edx,%edx,2), %eax    # eax = y*3
4  sall   $4, %eax               # eax *= 16 (t4)
3  leal   4(%ecx,%eax), %eax     # eax = t4 +x+4 (t5)
1  addl   %ecx, %edx             # edx = x+y (t1)
2  addl   16(%ebp), %edx         # edx += z (t2)
5  imull  %edx, %eax             # eax = t2 * t5 (rval)
```

Важно в `arith`

```
 int arith(int x, int y, int z)
 {
1  int t1 = x+y;
2  int t2 = z+t1;
3  int t3 = x+4;
4  int t4 = y * 48;
3  int t5 = t3 + t4;
5  int rval = t2 * t5;
   return rval;
 }
```

- Порядок команд иной, чем в Си-программе
- Часть Си-выражений требуют несколько команд
- Часть команд реализуют несколько Си-выражений
- Выдает точно такой же код при компиляции :

`(x+y+z)*(x+4+48*y)`

```
   movl   8(%ebp), %ecx          # ecx = x
   movl   12(%ebp), %edx         # edx = y
4  leal   (%edx,%edx,2), %eax    # eax = y*3
4  sall   $4, %eax               # eax *= 16 (t4)
3  leal   4(%ecx,%eax), %eax     # eax = t4 +x+4 (t5)
1  addl   %ecx, %edx             # edx = x+y (t1)
2  addl   16(%ebp), %edx         # edx += z (t2)
5  imull  %edx, %eax             # eax = t2 * t5 (rval)
```

Ещё пример

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

Пролог

Тело

Эпилог

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y          (t1)
sarl $17,%eax          # eax = t1>>17       (t2)
andl $8185,%eax        # eax = t2 & mask    (rval)
```

Ещё пример

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

Пролог

Тело

Эпилог

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y          (t1)
sarl $17,%eax          # eax = t1>>17       (t2)
andl $8185,%eax        # eax = t2 & mask    (rval)
```

Ещё пример

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

2

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

2

Пролог

Тело

Эпилог

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y          (t1)
sarl $17,%eax          # eax = t1>>17       (t2)
andl $8185,%eax        # eax = t2 & mask    (rval)
```

Ещё пример

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

3 3

```
logical:
    pushl %ebp
    movl %esp,%ebp

    movl 12(%ebp),%eax
    xorl 8(%ebp),%eax
    sarl $17,%eax
    andl $8185,%eax

    popl %ebp
    ret
```

3

Пролог

Тело

Эпилог

$2^{13} = 8192, 2^{13} - 7 = 8185$

```
movl 12(%ebp),%eax     # eax = y
xorl 8(%ebp),%eax      # eax = x^y          (t1)
sarl $17,%eax          # eax = t1>>17       (t2)
andl $8185,%eax        # eax = t2 & mask    (rval)
```

```
int log(int x, int y){
 int t1,t2,mask, rval;
t1=x^y;
t2=t1>>5;
mask=(1<<8) - 7;
rval=t2 & mask;
return t2;}
```

```
log  proc      far
     push      bp
     mov       bp,sp
     sub sp,6
     mov       ax,word ptr [bp+6]
     xor  ax,word ptr [bp+8]
     mov       cl,5
     sar  ax,cl
     mov       word ptr [bp-4],249
     and ax,word ptr [bp-4]
     mov       sp,bp
     pop bp
     ret
```

Специальные арифметические операции

Команда	Результат	Описание
`imull` S	R[%edx] : R[%eax] $\leftarrow S \times$ R[%eax]	Полное умножение со знаком
`mull` S	R[%edx] : R[%eax] $\leftarrow S \times$ R[%eax]	Полное умножение без знака
`cltd` S	R[%edx] : R[%eax] $\leftarrow$ SignExtend(R[%eax])	Преобразование в четверное слово
`idivl` S	R[%edx] $\leftarrow$ R[%edx] : R[%eax] mod S R[%eax] $\leftarrow$ R[%edx] : R[%eax] $\div S$	Деление со знаком
`divl` S	R[%edx] $\leftarrow$ R[%edx] : R[%eax] mod S R[%eax] $\leftarrow$ R[%edx] : R[%eax] $\div S$	Деление без знака

Примеры

```
адрес x есть %ebp+8, адрес y есть %ebp+12
1   movl    8(%ebp),%eax    Поместить x в %eax
2   imull   12(%ebp)        Умножить на y
3   pushl   %edx            Затолкнуть в стек старшие 32 разряда
4   pushl   %eax            Затолкнуть в стек младшие  32 разряда
```

```
Адрес x есть %ebp+8, адрес y есть %ebp+12
1   movl 8(%ebp),%eax       Поместить x в %eax
2   cltd                    Расширение знака в %edx
3   idivl 12(%ebp)          Разделить на y
4   pushl %eax              Протолкнуть в стек x / y
5   pushl %edx              Протолкнуть в стек x % y
```

Состояние процессора (IA32, частично)

- **Информация о непосредственно исполняемой программе**
  - Промежуточные данные (`%eax`, …)
  - Положение стека (`%ebp`,`%esp`)
  - Положение текущей команды (`%eip`, …)
  - Результаты последних проверок (CF, ZF, SF, OF)

Флаги условий (неявная установка)

- **Однобитные регистры- флаги**
  - CF перенос (для unsigned) SF знак (для signed)
  - ZF ноль OF переполнение (для signed)
- **Выставляются неявно арифметическими оерациями (как побочный результат)**

  Пример: **addl / addq Src,Dest** ↔ `t = a+b`

  CF=1, если перенос из старшего бита или заём в него (беззнаковое переполнение)

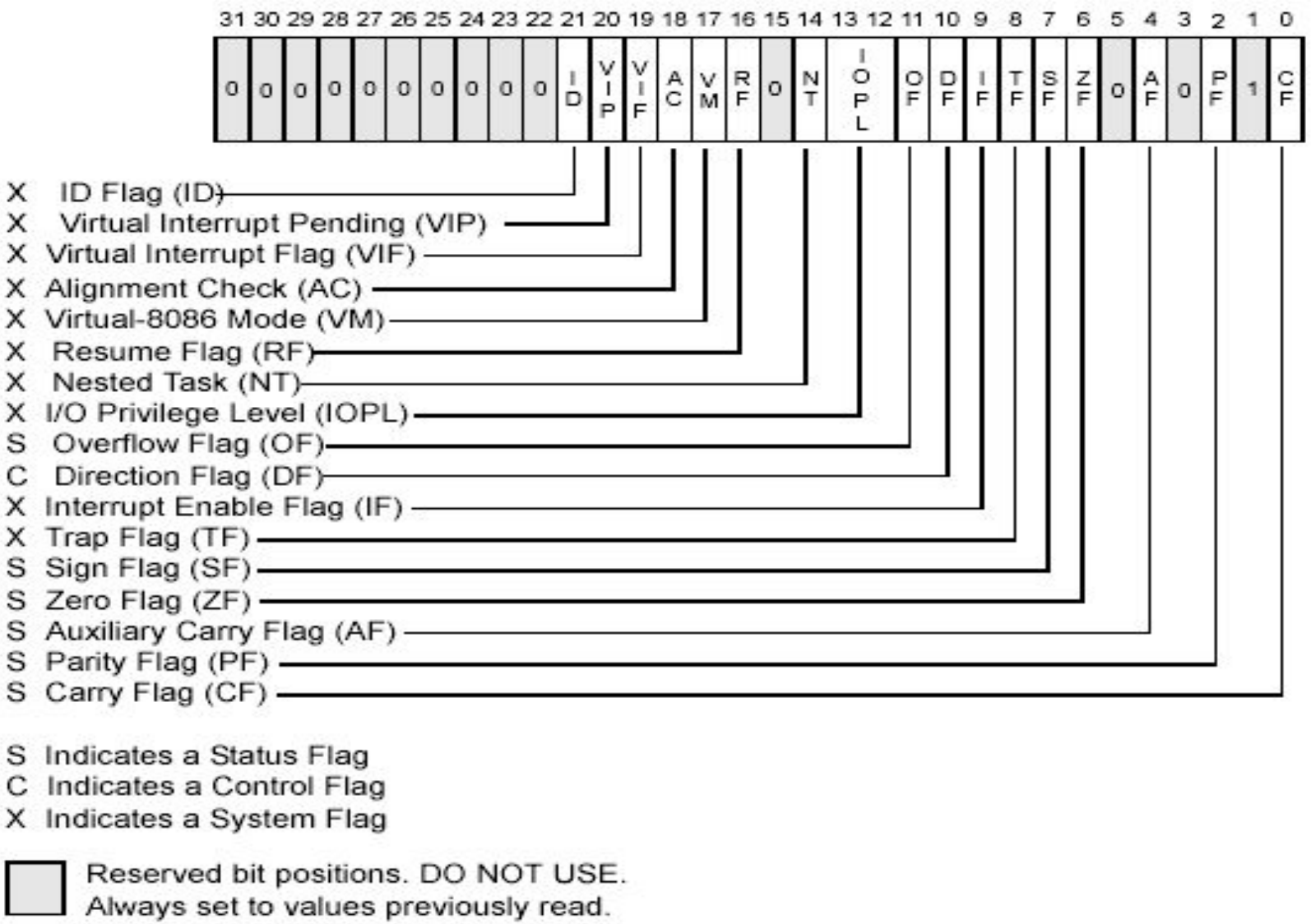
  ZF=1, если `t == 0`

  SF=1, если `t < 0` (как знаковое), SF == MSB

  OF=1, если переполнился доп.код (знаковый)

  ```
  (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
  ```
- **Не устанавливается командой `lea`!**

Регистр флагов (EFlags)



Флаги состояния (Status Flags)

№ бита	Мнемоника	Флаг	Содержание и назначение
0	cf	Carry Flag	1 - арифметическая операция произвела перенос из старшего бита результата. Старшим является 7-й, 15-й или 31-й бит в зависимости от размерности операнда; 0 - переноса не было.
2	pf	Parity Flag	Этот флаг - только для 8 младших разрядов операнда любого размера. 1, когда 8 младших разрядов результата содержат четное число единиц; 0, когда 8 младших разрядов результата содержат нечетное число единиц.
4	af	Auxiliary carry Flag	Только для команд, работающих с BCD-числами: 1- в результате операции сложения был произведен перенос из разряда 3 в старший разряд или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде; 0-переносов не было.
6	zf	Zero Flag	1 - результат нулевой; 0 - результат ненулевой.
7	sf	Sign Flag	Отражает состояние старшего бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно).
11	of	Overflow Flag	Фиксирует факт потери значащего бита при арифметических операциях. 1 - произошел перенос(заем) из(в) старшего или знакового бита; 0 – произошел перенос(заем) из(в) старшего и знакового бита или переноса не было.
12 13	iopl	Input/Output Privilege Level	Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода-вывода в зависимости от уровня привилегий задачи.
14	nt	Nested Task	Используется в защищенном режиме работы микропроцессора для фиксации того факта, что одна задача вложена в другую.

Примеры:

```
xor     ax,ax
mov     al,64
add     al,64
```

01000000
01000000
10000000
cf=0 pf=0 zf=0 sf =1 of=1

```
xor     ax,ax
mov     al,128
add     al,128
```

10000000
10000000
(1)00000000
cf=1 pf=1 zf=1 sf=0 of=0

```
xor     ax,ax
mov     al,192
add     al,192
```

11000000
11000000
(1)10000000
cf=1 pf=0 zf=0 sf=1 of=1

Флаги условий (явная установка сравнением)

- **Явная установка командами сравнения**
  - **`cmpl/cmpq` Src2, Src1**
  - **`cmpl b,a`** как вычисление **`a-b`** без сохранения результата
  - CF=1, если перенос из старшего бита или заём в него (используется при беззнаковых сравнениях)
  - ZF=1, если **`a == b`**
  - SF=1, если **`(a-b) < 0`** (как знаковые)
  - OF=1, переполнение в дополнительном коде (знаковое)

```
(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)
```

Флаги условий (явная установка проверкой бит)

- **Явная установка командой test**
  - **`testl/testq` Src2, Src1**
    `testl b,a` как вычисление `a&b` без сохранения результата
  - Устанавливает флаги в зависимости от Src1 & Src2
  - Удобно, если один из операндов - маска
  - ZF=1, если **`a&b == 0`**
  - SF=1, если **`a&b < 0`** (старший бит == 1)

Команды установки байта по условию

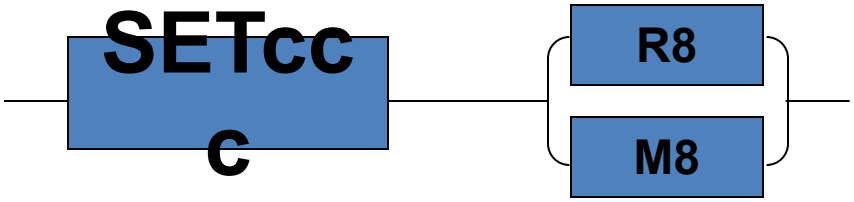
SETccc операнд

Команды проверяют условие, заданное модификатором ccc в коде операции (фактически, состояние флагов) и устанавливают операнд логическим значением 1 или 0 в зависимости от истинности условия. Команды Setccc можно использовать после любой команды, изменяющей флаги, при необходимости анализа результата изменения. Если проанализировать условия для команд Jccc, то обнаружится их полное соответствие с условиями, обрабатываемыми командами Jccc.

Чтение флагов условий IA32 - 1

- **Команды set***
  - Устанавливают один байт в 1 или 0 в зависимости от флагов

Команда	Условие	Описание
`sete`	`ZF`	Равно / Ноль
`setne`	`~ZF`	Неравно / Не ноль
`sets`	`SF`	Отрицательно
`setns`	`~SF`	Неотрицательно
`setg`	`~(SF^OF)&~ZF`	Больше (знаковое)
`setge`	`~(SF^OF)`	Больше или равно (знаковое)
`setl`	`(SF^OF)`	Меньше (знаковое)
`setle`	`(SF^OF)|ZF`	Меньше или равно (знаковое)
`seta`	`~CF&~ZF`	Выше (беззнаковое)
`setb`	`CF`	Ниже (unsigned)



Команда	Условие	Команда	Условие
seta/setnbe	cf=0 и zf=0	setle/setng	zf=1 или sf≠of
setae/setnb	cf=0	setnc	cf=0
setb/setnae	cf=1	setne/setnz	zf=0
setbe/set	cf=1 или zf=1	setno	of=0
setc	cf=1	setnp/setpo	pf=0
sete/setz	zf=1	setns	sf=0
setg/setnle	zf=0 или sf=of	seto	of=1
setge/setnl	sf=of	setp/setpe	pf=1
setl/setnge	sf≠of	sets	sf=1

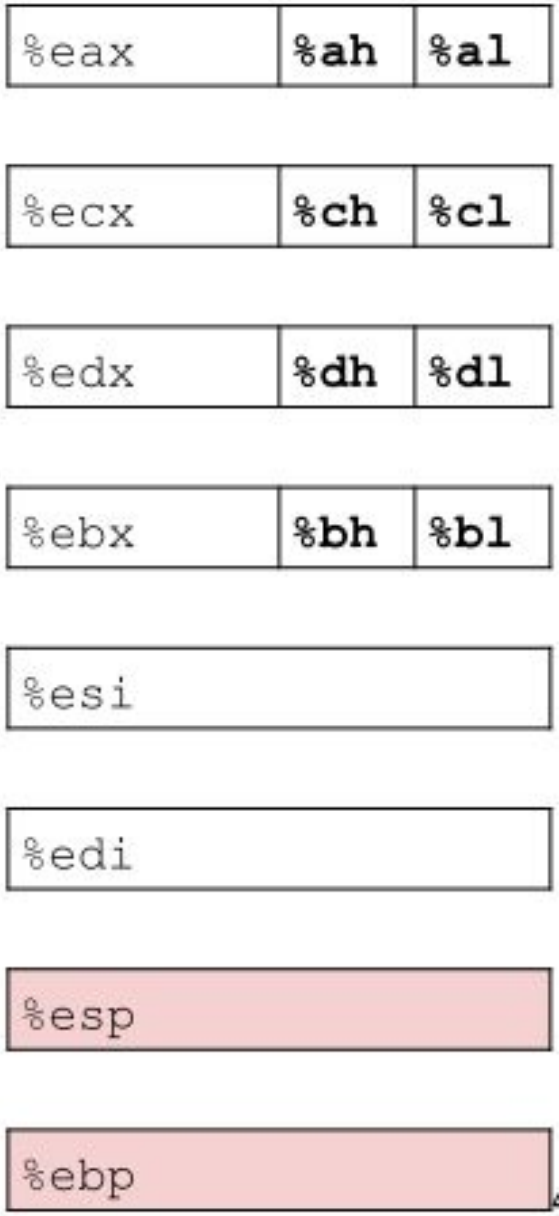
Чтение флагов условий IA32 - 2

- **Команды set***
  - Устанавливают один байт в 1 или 0 в зависимости от флагов
- **Один из 8 байтовых регистров**
  - Не изменяются остальные 3 байта
  - Для зануления обычно используется `movzbl`

```
int gt (int x, int y)
{
  return x > y;
}
```

Тело функции

```
movl 12(%ebp),%eax    # eax = y
cmpl %eax,8(%ebp)     # Сравнить x и y
setg %al              # al = x > y
movzbl %al,%eax       # Зануление остатка
```



Чтение флагов условий x86-64

- **Команды set***
  - Устанавливают один байт в 1 или 0 в зависимости от флагов
  - Не меняют оставшиеся 3 байта 4-х байтного регистра

```
int gt (long x, long y)
{
  return x > y;
}
```

```
long lgt (long x, long y)
{
  return x > y;
}
```

Тела функций

```
cmpl %esi, %edi
setg %al
movzbl %al, %eax
```

```
cmpq %rsi, %rdi
setg %al
movzbl %al, %eax
```

Нулевые ли старшие 32 бита 64-битного регистра `%rax`?
Да: 32-ная команда устанавливает старшие 32 бита в 0!

```
_qt proc   far
 push  bp
 mov   bp,sp
 mov   ax,word ptr [bp+6]
 cmp   ax,word ptr [bp+8]
 jle  short @1@86
 mov   ax,1
 jmp    short @1@114
@1@86:
 xor     ax,ax
@1@114:
 pop    bp
 ret
```

Переходы

- **Команды j***
  - Передача управления по адресу в зависимости от флагов

j*	Условия	Описание
`jmp`	`1`	Безусловный
`je`	`ZF`	Равно / Ноль
`jne`	`~ZF`	Неравно / Не ноль
`js`	`SF`	Отрицательно
`jns`	`~SF`	Неотрицательно
`jg`	`~(SF^OF)&~ZF`	Больше (знаковое)
`jge`	`~(SF^OF)`	Больше или равно (знаковое)
`jl`	`(SF^OF)`	Меньше (знаковое)
`jle`	`(SF^OF)|ZF`	Меньше или равно (знаковое)
`ja`	`~CF&~ZF`	Выше (беззнаковое)
`jb`	`CF`	Ниже (unsigned)

Пример условного перехода

```
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
      result = x-y;
    } else {
      result = y-x;
    }
    return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Пролог

Тело1 «если»

Тело2a «то»

Тело2b «иначе»

Эпилог

Пример условного перехода

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

- **Си допускает “goto” для передачи управления**
  - Стиль кодирования «ближе к машине»
- **В общем считается плохим стилем**

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Пролог

Тело1 «если»

Тело2a «то»

Тело2b «иначе»

Эпилог

Пример условного перехода

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Пролог

Тело1 «если»

Тело2a «то»

Тело2b «иначе»

Эпилог

Пример условного перехода

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Пролог

Тело1 «если»

Тело2a «то»

Тело2b «иначе»

Эпилог

Пример условного перехода

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp
    ret
```

Пролог

Тело1 «если»

Тело2a «то»

Тело2b «иначе»

Эпилог

```
_absd    proc     far
  push    bp
  mov     bp,sp
  mov     dx,word ptr [bp+6]
  mov     bx,word ptr [bp+8]
  cmp     dx,bx
  jle   short @1@86
  mov     ax,dx
  subax,bx
  jmpshort @1@114
@1@86:
  mov     ax,bx
  subax,dx
@1@114:
  pop     bp
  ret
```

Трансляция условного выражения

Си код

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

"goto" версия

```
  nt = !Test;
  if (nt) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Test – целочисленное выражение
  - = 0 интерпретируется как ложь
  - ≠ 0 интерпретируется как истина
- Создаёт раздельные фрагменты кода для **Then_Expr** и **Else_Expr**
- Исполняется один их двух

```
_absd1  proc    far
 push   bp
 mov    bp,sp
 subsp,2
 mov    dx,word ptr [bp+6]
 mov    bx,word ptr [bp+8]
 cmp    dx,bx
 jle  short @1@86
 mov    ax,dx
 subax,bx
 jmpshort @1@114
@1@86:
 mov    ax,bx
 subax,dx
@1@114:
 mov    sp,bp
 pop    bp
 ret
```