Лекция 14
Ассемблер x86/x64

Инструкция mov

- Пересылка данных
  movSFX SRC, DST
- Куда пересылаем – второй аргумент!
- SFX – размер пересылаемых данных:
  - 'movb – байт
  - 'movw – слово (16 бит)
  - 'movl – двойное слово (32 бит)
  - 'movq – 64 бит
- Типы пересылок:
  - Регистр-регистр
  - Регистр-память
  - Память-регистр

Методы адресации

- Возможные типы аргументов операции определяются поддерживаемыми процессором методами адресации
- Методы адресации:
  - Регистровый – указывается имя регистра
    movl %esp, %ebp
  - Непосредственный (immediate) – аргумент задается в инструкции – знак $
    movb **$16**, %cl
  - Прямой (direct) – адрес ячейки памяти задается в инструкции
    movl %eax, var1

Преобразования целых

- Расширение нулями:
  ```
  movzbl    var, %eax   // 8 → 32 бита
  movzwl    var, %eax    // 16 → 32 бита
  ```
- Расширение знаковым битом:
  ```
  movsbl    var, %eax
  movswl    var, %eax
  cdq                          // eax → eax:edx
  ```

Арифметика

- Арифметические инструкции:

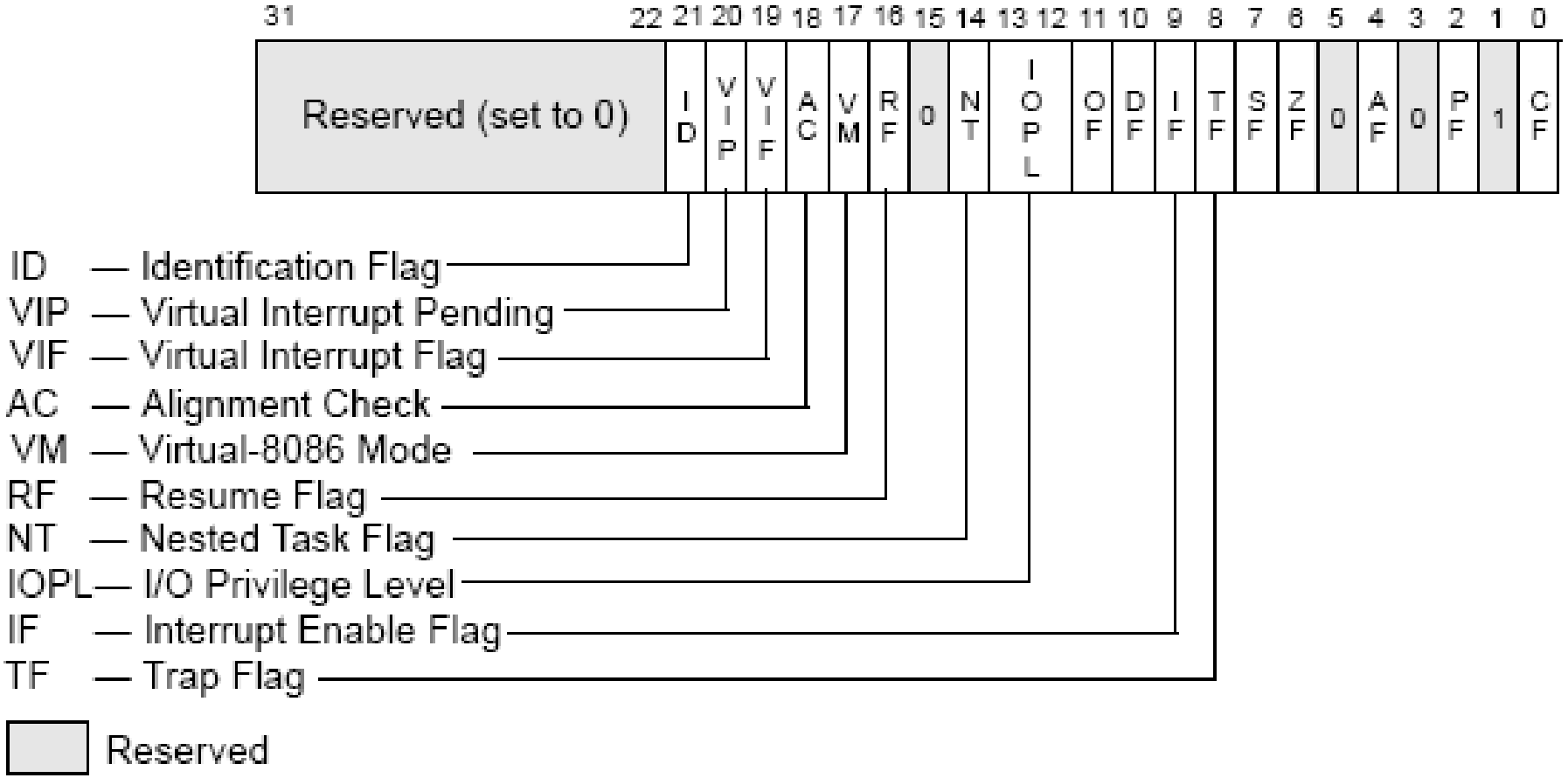
```
add  SRC, DST      // DST += SRC
sub  SRC, DST      // DST -= SRC
cmp  SRC1, SRC2    // SRC2 – SRC1
and  SRC, DST      // DST &= SRC
or SRC, DST        // DST |= SRC
xor  SRC, DST      // DST ^= SRC
test SRC1, SRC2    // SRC1 & SRC2
not  DST           // DST = ~DST
neg  DST           // DST = -DST
inc  DST           // ++DST
dec  DST           // --DST
```

Флаги результата операции

- Регистр EFLAGS содержит специальные биты-флаги результата операции
- Для x86 они называются: ZF, SF, CF, OF
  - ZF (бит 6) – флаг нулевого результата
  - SF (бит 7) – флаг отрицательного результата
  - CF (бит 0) – флаг переноса из старшего бита
  - OF (бит 11) – флаг переполнения

Регистр EFLAGS

- Нас интересуют: CF, ZF, SF, OF



Примеры

1(1) + 2(2) = 3(3), ZF=0, SF=0, CF=0, OF=0

0(0) + 0(0) = 0(0), ZF=1, SF=0, CF=0, OF=0

130(-126)+0(0)=130(-126), ZF=0,SF=1,CF=0, OF=0

130(-126)+126(126)=0(0), ZF=1, SF=0, CF=1, OF=0

127(127)+127(127)=254(-2), ZF=0,SF=1,CF=0,OF=1

Сдвиги

- Арифметические сдвиги влево/вправо

```
sal   %eax          // %eax <<= 1
sal   $2, %eax      // %eax <<= 2
sal   %cl, %eax     // %eax <<= %cl & 0x1F
sar   %eax          // %eax >>= 1
sar   $5, %eax      // ...
sar   %cl, %eax     // ...
```

- Логические сдвиги влево/вправо

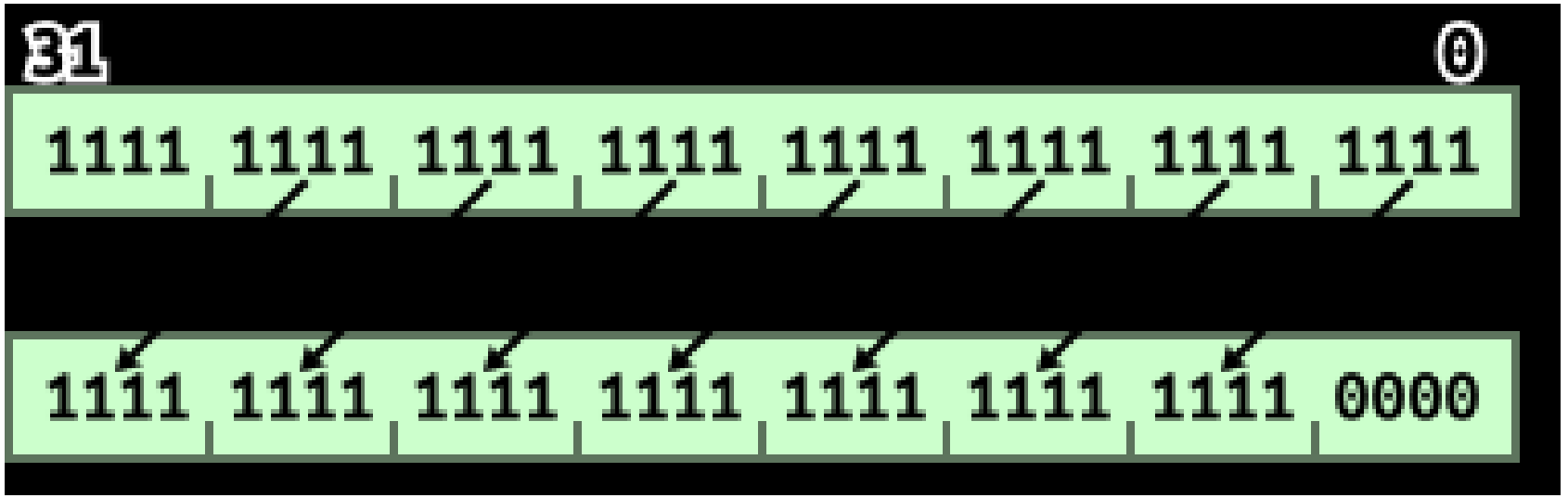
```
shl   [CNT, ] DST // сдвиг влево
shr   [CNT, ] DST // сдвиг вправо
```

Вращения

- Вращение влево/вправо
  rol [CNT,] DST
  ror [CNT,] DST
- Вращение через CF влево/вправо
  rcl [CNT,] DST
  rcr [CNT,] DST

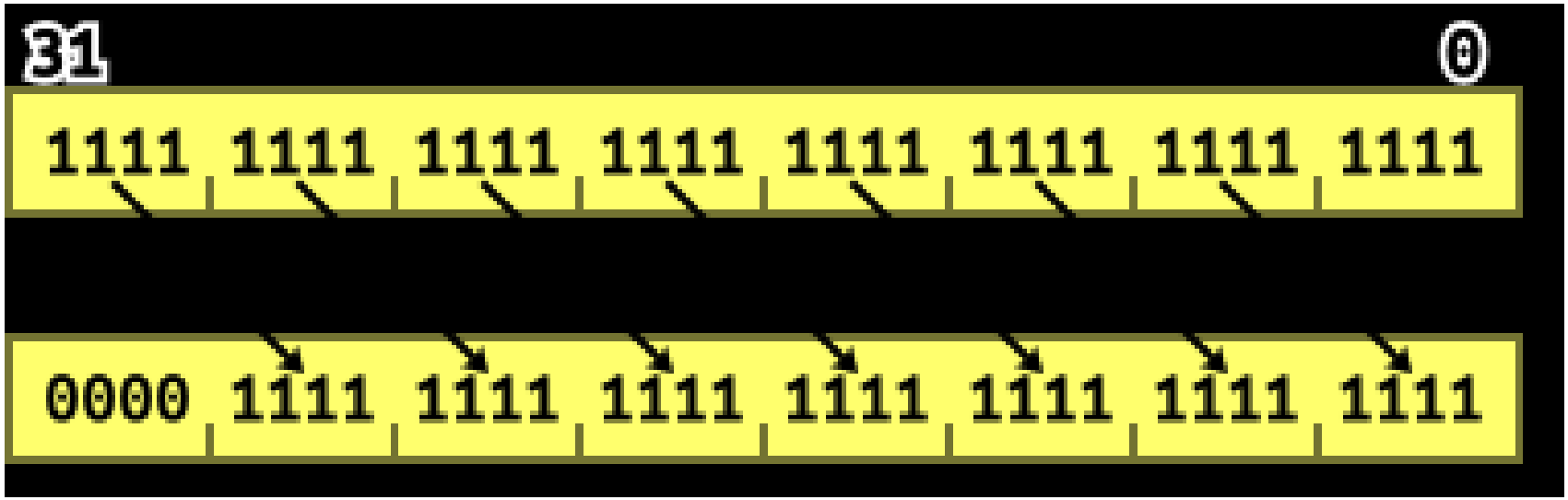
asl/lsl

- asll $4, %eax



lsr

- LSRL $4, %eax

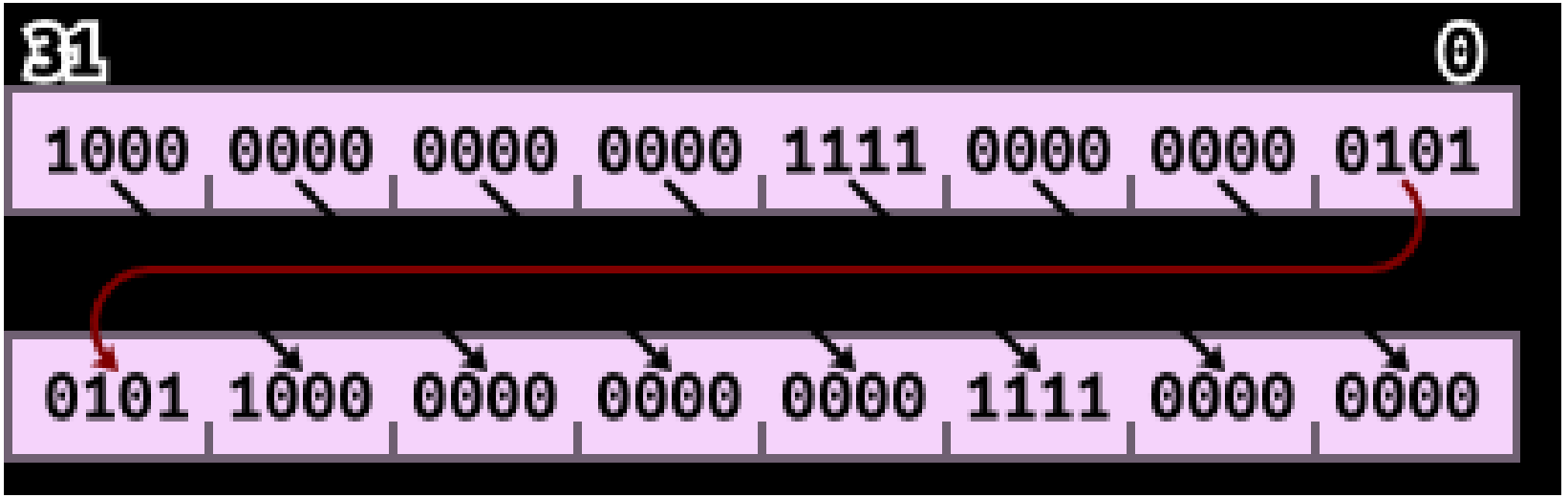


Arithmetical Shift Right

- ASRL $4, %eax

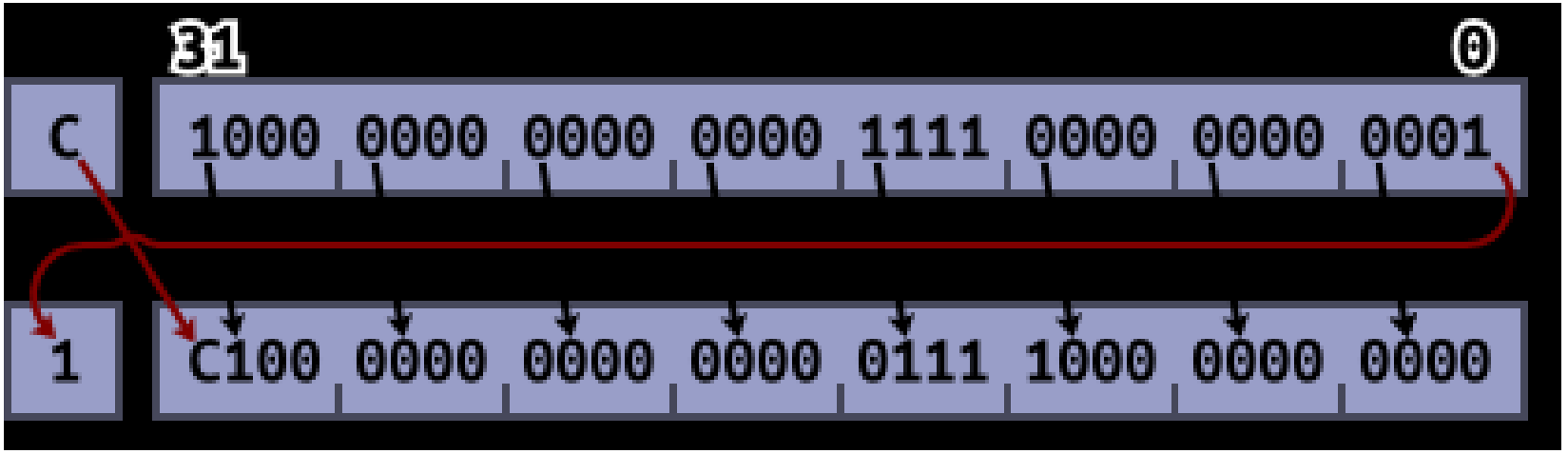
ror

- RORL $4, %eax



rcr

- RCRL %eax


31
0
C
1000 0000 0000 0000 1111 0000 0000 0001
1
C100 0000 0000 0000 0111 1000 0000 0000

Условные переходы

- Переход на метку выполняется только если установлена соответствующая комбинация флагов результата
- Условные переходы по равенству/неравенству
  JE / JZ переход если == или 0
  JNE / JNZ переход если != или не 0

Условные переходы

- Для операций с беззнаковыми числами
  JA / JNBE переход если >
  JAE / JNB / JNC переход если >=
  JB / JNAE / JC переход если <
  JBE / JNA переход если <=
- Для операция со знаковыми числами
  JG / JNLE переход если >
  JGE / JNL переход если >=
  JL / JNGE переход если <
  JLE / JNG переход если <=

Условные переходы

- Специальные случаи

  JO переход если OF == 1
  JNO переход если OF == 0
  JS переход если SF == 1
  JNS переход если SF == 0

Влияние на флаги процессора

- Разные инструкции по-разному влияют на состояние флагов процессора
  - Инструкции ADD, SUB, CMP, INC устанавливает SZOC в зависимости от результата
  - IMUL устанавливает OC в зависимости от представимости результата 32 битами, Z – неопределен, S – старший бит младших 32 битов
  - LEA, MOV – не изменяет флаги
  - AND, TEST, OR, XOR – обнуляют O, C, устанавливают S и Z в зависимости от результата
- Задокументировано для каждой инструкции (например http://www.felixcloutier.com/x86/)

Отображение if

```
int a, b;

if (a >= b) {
  // body
}
```

```
If (a – b < 0) goto done;

// body

done: ...
```

```
mov a, %eax
cmp b, %eax // a – b
jl done
  // body
done:

// используем знаковый
// переход
// меняем условие на
// противоположное
```

Отображение if

```
unsigned a, b;

if (a >= b) {
  // body
}


If (a < b) goto done;
...
done:
```

```
mov a, %eax
cmp b, %eax // a – b
jb done
  // body
done:

// используем
// беззнаковый
// переход
```

Отображение if-else

```
int a, b;

if (a >= b) {
  // if-body
} else {
  // else-body
}
```

```
If (!(a >= b)) goto else_label;
// if-body
Goto done_label;
else_label:
// else-body
done_label:
```

```
mov a, %eax
cmp b, %eax // a – b
jl else_label
  // if-body
jmp done_label
else_label:
  // else-body
done_label:
```

Отображение &&

```
int i;
int *p;

if (i >= 0 && p[i] > 0) {
  // if-body
}
```

```
mov i, %eax
test %eax, %eax
jl out_if
mov p(,%eax,4), %eax
test %eax, %eax
jle out_if
// if-body
out_if:
```

Отображение ||

```
int i;
int *p;

if (i < 0 || p[i] == 0) {
  // if-body
}
```

```
mov i, %eax
test %eax, %eax
jl if_body
mov p(,%eax,4), %eax
test %eax, %eax
jne out_if
if_body:
// if-body
out_if:
```

&& и ||

- Хотя && и || - “логические” связки, в сгенерированном коде им соответствуют условные переходы
- && и || - это варианты оператора if, а не логические операции:

```
if (a && b) {
}
```

это

```
if (a) {
  if (b) {
  }
}
```

Специальные варианты if

```
a  = b;
if (b > c) a = c;
```

```
mov b, %eax
mov c, %ecx
cmp %ecx, %eax // b-c
cmovg %ecx, %eax
mov %eax, a
// нет условных
// переходов!
```

Преобразование к булевскому

```
int a;
_Bool b;

b = a;
```

```
mov a, %eax
test %eax, %eax
setnz b
```

```
If (a) b = 1; else b = 0;
```

```
mov a, %eax
test %eax, %eax
jz if_body
mov $1, b
jmp out_if
if_body: mov $0, b
out_if:
```

Цикл while

```
struct Foo *p = head;

while (p) {
  // body
  p = p->next;
}
```

```
mov head, %ebx
test %ebx, %ebx
jz out_loop
loop:
// body
mov (%ebx), %ebx
test %ebx, %ebx
jnz loop
out_loop:
```

Метод адресации памяти

- Методы адресации – способы получения адреса операндов инструкции в памяти
- Общий вид обращения к памяти:
  OFFSET(BREG, IREG, SCALE)
  адрес вычисляется по формуле:
  BREG + OFFSET + IREG * SCALE
- BREG – базовый регистр (общего назн.)
- IREG – индексный регистр (общего назн.)
- SCALE – {1, 2, 4, 8}, по умолчанию 1
- OFFSET – базовый адрес в памяти или смещение

Обращения к памяти

- Примеры:
  (%eax) // адрес находится в %eax
  16(%esi) // адрес равен %esi + 16
  array(,%eax) // адрес равен array + %eax
  array(,%eax,4) // адрес равен array + %eax*4
  (%ebx,%eax,2) // адрес: %ebx + %eax * 2
  -4(%ebx,%eax,8) // адрес: %ebx-4+%eax*8

Примеры использования

- Разыменование указателя
  char *p; int c = *p;

  (если p загружен в %eax)
  movsbl (%eax), %eax // в %eax будет c
- Доступ к глобальному массиву
  unsigned short array[N]; int x = array[i];
  (если i загружено в %esi)
  movzwl array(,%esi,2), %eax // результат в %eax

Примеры использования

- Доступ к массиву
  int *p; int i;
  x = p[i];

  // пусть p находится в %ebx, i в %esi
  movl (%ebx, %esi, 4), %eax // x – в %eax
- Если размер элемента массива не 1, 2, 4, 8, потребуется операция умножения или несколько сложений и сдвигов

Примеры использования

- Доступ к полю структуры
  struct Str { int f1; int f2; };
  struct Str *p;
  int x = p->f2;

  // пусть p находится в %ebx
  movl 4(%ebx), %eax // x в %eax
- Любой доступ к памяти может быть представлен как комбинация разыменования, доступа к элементу массива, доступа к полю структуры

Инструкция lea

- Вместо обращения к памяти и сохранения в регистре значения из памяти в регистре сохраняется адрес

```
leal   (%eax, %eax, 8), %eax
// %eax = %eax * 9
```

Типизация в ассемблере

- В ассемблере целое число (32 бит) может быть:
  - Знаковым целым числом
  - Беззнаковым целым числом
  - Указателем любого типа
- Тип никак не привязан к ячейке/регистру, в котором хранится число
- Интерпретация числа зависит от выполняющейся инструкции

Структура адресного пространства

- Код программы и данные, загружаемые из исполняемого файла (образ программы)
  - Содержит разные секции исполняемого кода, в том числе .text, .data, .bss
- Основной стек процесса
- Область динамической памяти (куча)