

# Целочисленные регистры (IA32)

Общего назначения

<b>%eax</b>	<b>%ax</b>	<b>%ah</b>	<b>%al</b>
<b>%ecx</b>	<b>%cx</b>	<b>%ch</b>	<b>%cl</b>
<b>%edx</b>	<b>%dx</b>	<b>%dh</b>	<b>%dl</b>
<b>%ebx</b>	<b>%bx</b>	<b>%bh</b>	<b>%bl</b>
<b>%esi</b>	<b>%si</b>		
<b>%edi</b>	<b>%di</b>		
<b>%esp</b>	<b>%sp</b>		
<b>%ebp</b>	<b>%bp</b>		

Мнемоника  
(устаревшая)

*Accumulate*  
аккумулятор

*Counter*  
счётчик

*Data*  
данные

*Base*  
База

*Source index*  
Индекс источника

*Destination index*  
индекс назначения

*Stack pointer*  
указатель стека

*Base pointer*  
указатель базы

16-битные виртуальные регистры  
(для обратной совместимости)

# Сравнение языков ассемблера Intel and AT&T

В синтаксисе Intel нет никаких префиксов регистра или префиксов констант. В AT&T, однако, регистры помечены '%', и константы помечены с '\$'. В синтаксисе intel шестнадцатеричные или двоичные данные имеют suffix с 'h' и 'b' соответственно. Также, если первая шестнадцатеричная цифра - буква тогда, перед ней ставится '0'.

Пример:Example:

Intel Syntax	AT&T Syntax
<u>mov</u> <u>eax, 1</u>	<u>movl</u> \$1, %eax
<u>mov</u> <u>ebx, 0ffh</u>	<u>movl</u> \$0xff, %ebx
<u>int</u> 80h	<u>int</u> \$0x80

Направление операндов в синтаксисе Intel противоположно от того из синтаксиса AT&T. В синтаксисе Intel первый операнд - место назначения, и второй операнд - источник, тогда как в синтаксисе AT&T первый операнд - источник, и второй операнд - место назначения.

Example:

Intel Syntax	AT&T Syntax
<u>instr</u> <u>dest, source</u>	<u>instr</u> <u>source, dest</u>
<u>mov</u> <u>eax, [ecx]</u>	<u>movl</u> (%ecx), %eax

## 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
<code>mov    eax, [ebx]</code>	<code>movl   (%ebx), %eax</code>
<code>mov    eax, [ebx+3]</code>	<code>movl   3(%ebx), %eax</code>

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
<code>instr    [base+index*scale+offset]</code>	<code>instr    offset(base, index, scale), foo</code>
<code>mov      eax, [ebx+20h]</code>	<code>movl     0x20(%ebx), %eax</code>
<code>add      eax, [ebx+ecx*2h]</code>	<code>addl     (%ebx, %ecx, 0x2), %eax</code>
<code>lea      eax, [ebx+ecx]</code>	<code>leal     (%ebx, %ecx), %eax</code>
<code>sub      eax, [ebx+ecx*4h-20h]</code>	<code>subl     -0x20(%ebx, %ecx, 0x4), %eax</code>

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).

## Suffixes.

Как Вы, возможно, заметили, мнемоника синтаксиса AT&T имеет суффикс. Значение этого суффикса - значение размера операнда. 'l' long, 'w' word, и 'b' для байта. У синтаксиса intel есть подобные директивы для использования с операндами памяти, т.е. `byte ptr`, `word ptr`, `dword ptr`.

Example:

### Intel Syntax

```
mov    al,bl  
mov    ax,bx  
mov    eax,ebx  
mov    eax, dword ptr [ebx]
```

### AT&T Syntax

```
movb   %bl,%al  
movw   %bx,%ax  
movl   %ebx,%eax  
movl   (%ebx),%eax
```

# Сравнение языков ассемблера Intel and AT&T

- How does “mov (%ebx,%ecx,4),%eax” work? How does “mov (%ebx,%ecx,4),%eax” work?
- The complete memory addressing mode format in AT&T assembly is:
  - offset(base, index, width)
  - смещение (база, индекс, масштаб)
  - offset = 0
  - base = ebx
  - index = ecx
  - width = 4
  - $eax = *(uint32\_t *)((uint8\_t *)ebx + ecx * 4 + 0)$
- \_\_mov eax, [ebx+ecx\*4] in NASM mov eax, [ebx+ecx\*4] \_\_mov eax, [ebx+ecx\*4]

1.  $x == (\text{int}) (\text{float}) x$

Нет. Например, когда  $x$  —  $TMax$ .

2.  $x == (\text{int}) (\text{double}) x$

Да, поскольку `double` обладает большей точностью и диапазоном, нежели `int`.

3.  $f == (\text{float}) (\text{double}) f$

Да, поскольку `double` обладает большей точностью и диапазоном, нежели `float`.

4.  $d == (\text{float}) d$

Нет. Например, когда  $d$  равно  $1e400$ , в правой части получаем  $+\infty$ .

5.  $f == -(-f)$

Да, число с плавающей точкой отрицается простой инверсией знака.

6.  $2/3 == 2/3.0$

Нет, значение в левой части будет целым значением 0, тогда как значение в правой части будет приближением с плавающей точкой, равное  $\frac{2}{3}$ .

7.  $(d >= 0.0) \mid \mid ((d*2) < 0.0)$

Да, поскольку умножение монотонно.

8.  $(d+f) -d == f$

Нет. Например,  $d$  равно  $+\infty$ , а  $f$  — 1, тогда левая часть будет `NaN`, а правая 1.