

# Лекция 12

# 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
- Задokumentировано для каждой инструкции (например <http://www.felixcloutier.com/x86/>)

# Работа с 64-битными целыми

- 64-битные целые требуют по несколько инструкций для обработки
  - 64-битное значение в паре регистров (напр. %eax и %edx)
  - Логические операции – отдельно для младшей и старшей половины
  - Сложение: ADD для младшей половины, ADC для старшей
  - Вычитание: SUB для младшей половины, SBB для старшей
  - Умножение, деление: вспомогательные функции (находятся в libgcc или аналогичной библиотеке)

# Отображение if

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

```
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  
}
```

```
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  
}
```

```
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
```

# 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:
```

# Оператор switch

- В зависимости от количества и группировки значений:
  - Линейная цепочка if
  - Дерево if
  - Таблица переходов

# Таблица переходов

```
int a;  
switch (a) {  
case 1: // block-1  
    break;  
case 2: // block-2  
    break;  
case 3: // block-3  
    break;  
}
```

```
mov a, %eax  
sub $1, %eax  
cmp $2, %eax  
ja out_switch  
jmp *swtab(,%eax,4)  
swtab: .int blk1, blk2, blk3  
blk1: jmp    out_switch  
blk2: jmp    out_switch  
blk3: jmp    out_switch  
out_switch:
```

# Floating point

- Устройство вычислений с плавающей точкой может отсутствовать на простых процессорах (low-end микроконтроллерах)
  - Заменяется фиксированной точкой или программным вычислением
- X86/x64 два разных (!) устройства для вычислений с плавающей точкой

# X86 Legacy - FPU

- 8 регистров размером 80 бит (то есть тип long double), организованных в стек: %st(0), %st(1), ..., %st(7)
- Загрузка из памяти: варианты fld
- Сохранение в память: варианты fst
- Разные вычисления: fadd, fsub, ..., fsin



# X86 calling convention

- Если подпрограмма возвращает результат типа float или double, он возвращается в %st(0)
- Результат должен быть удален из стека FPU, даже если не используется (ответственность вызывающей стороны)
- Регистры %st(0), ..., %st(7) – scratch

# SSE и расширения

- SSE, SSE2, ..., - набор SIMD команд, или векторные расширения
- SIMD (single instruction multiple data) – классификация Флинна обработки данных
  - SIMD (single instruction multiple data)
  - SISD (single instruction single data)
  - ...
- Другой пример SIMD – вычисления на GPU
- Одна инструкция обрабатывает сразу несколько однотипных наборов данных (вектор)

# Регистры SSE

- X86: 8 регистров %xmm0 - %xmm7 – 128 бит
- X64: 16 регистров %xmm0 - %xmm15
- %mxcsr – регистр статуса
- Все регистры – scratch
- На x64 в %xmm передаются параметры, в %xmm0 возвращается результат

# Упаковка/распаковка

- Один регистр %xmm может содержать:
  - 2 числа типа double
  - 4 числа типа float
  - 2 64-битных целых числа
  - 4 32-битных
  - 8 16-битных
  - 16 8-битных
- Операции выполняются над всеми числами за раз (как правило)

# Выравнивание аргументов

- MOVDQA – загружает значение XMM из памяти целиком (128 бит)
- PADDD – складывает два вектора из 4 32-битных значений в XMM регистрах или памяти
- У подобных инструкций адрес в памяти должен быть **выровнен по 16**
- Иначе – Segmentation Fault

# Дополнение к x86/x64 CC

- Стек должен быть выровнен по 16
- При вызове подпрограммы адрес начала области аргументов в стеке (то есть адрес первого аргумента для x86) должен находиться по адресу, кратному 16
- То есть, если  $f(a, b, c)$ , то  $\&a$  должен быть  $0xNNNNNNN0$  (последняя 16-ричная цифра – 0)
- Соответственно, адрес ячейки стека, в которой хранится адрес возврата, заканчивается на  $0xC$

# Сравнение floating point

- Инструкция COMISD сравнивает две младших половины (double) регистров или памяти  
`comisd %xmm1, %xmm0 // xmm0 – xmm1`
- Устанавливаются флаги PF (unordered – то есть при операциях с NaN); CF, ZF
- Можно использовать беззнаковые переходы:  
`ja // переход, если %xmm0 > %xmm1`

# Где почитать

- <https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf>
- <http://x86.renejeschke.de/>
-