

АССЕМБЛЕР

Введение в ассемблер и ассемблирование. Кооперация языка С и
ассемблера целевой архитектуры

К. Владимиров, Yadro, 2024
mail-to: konstantin.vladimirov@gmail.com

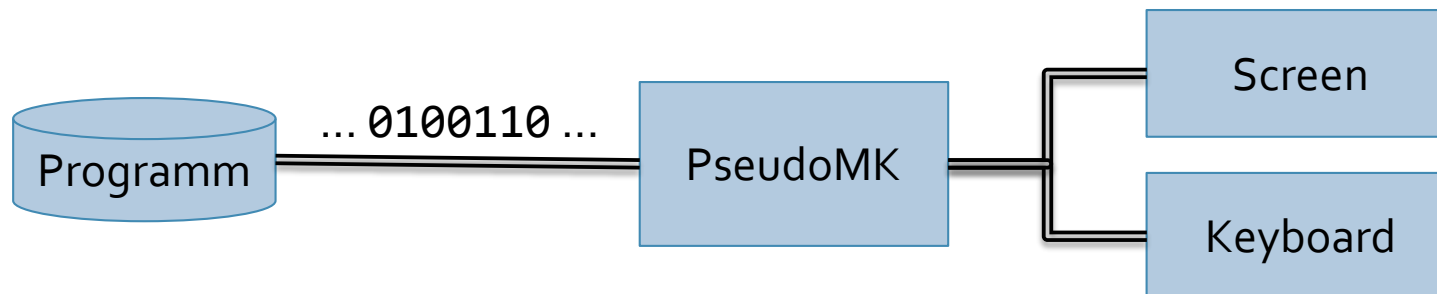
СЕМИНАР 6.1

Изобретаем ассемблер.

Немного о работе компьютеров

- Дедушкой любого компьютера является электронное арифметически-логическое устройство, то есть калькулятор
- Представим гипотетический целочисленный калькулятор PseudoMK, имеющий внутри четыре регистра A, B, C, D на 8 бит каждый
- Как закодировать команды PseudoMK, чтобы подавать их на вход в двоичном виде?

| | |
|------------|----------|
| MOVI D, i | ???????? |
| ADD rx, rs | ???????? |
| SUB rx, rs | ???????? |
| MUL rx, rs | ???????? |
| DIV rx, rs | ???????? |
| IN rd | ???????? |
| OUT rs | ???????? |



Предлагаемая кодировка

- Предложена следующая кодировка
- Регистры A, B, C, D кодируются от 00 до 11
- Каждая команда кодируется **опкодом** от 0 до 6 и каждое из четырёх сочетаний операндов ещё двумя битами
- Прочитайте команды: 0x1F, 0xC3, 0x92, 0xBE
- Приведите пример некорректной команды
- Это не единственная возможная кодировка
- Понимаете ли вы почему здесь выбрана именно она?
- Можете ли вы предложить лучший вариант?

| | |
|------------|----------|
| MOVI D, i | 0IIIIIII |
| ADD rx, rs | 1000RRRR |
| SUB rx, rs | 1001RRRR |
| MUL rx, rs | 1010RRRR |
| DIV rx, rs | 1011RRRR |
| IN rd | 110000RR |
| OUT rs | 110001RR |

Немного о работе компьютеров

- Теперь можно считать простые формулы

$(x / 3 + 5) * y - 2$

```
11000000 // IN A
00000011 // MOVI D, 3
10110011 // DIV A, D
00000101 // MOVI D, 5
10000011 // ADD A, D
11000001 // IN B
10100001 // MUL A, B
00000010 // MOVI D, 2
10010011 // SUB A, D
```

| | |
|------------|----------|
| MOVI D, i | 0IIIIIII |
| ADD rx, rs | 1000RRRR |
| SUB rx, rs | 1001RRRR |
| MUL rx, rs | 1010RRRR |
| DIV rx, rs | 1011RRRR |
| IN rd | 110000RR |
| OUT rs | 110001RR |

- Слева от // у нас перфокарта в виде машинных кодов. Справа [язык ассемблера](#).

Одна из многих программ

- Если использовать 16-ричные числа вместо двоичных, каждая команда займёт два разряда. Вот типичная программа для PseudoМК в [машинном коде](#)

```
0x5f 0xc2 0xae 0xc1 0xb9 0x87 0xc3 0xc0 0xa2 0x81 0xc5 0xad 0x95 0x8b
0x8a 0x9e 0x83 0x4d 0x99 0xbe 0xc6 0x83 0x16 0x83 0xc3 0x8d 0xa9 0x94
0x83 0x64 0x83 0x4d 0xae 0xa0 0x8d 0x83 0xc3 0x99 0xc5 0x81 0x80 0x88
0x81 0x85 0x83 0x09 0x9f 0x8f 0x82 0xc2 0x0b 0x83 0x6a 0x83 0x10 0x83
0xc3 0xc4 0x8b 0xb8 0x82 0xc2 0xbe 0x83 0x6b 0xb3 0xc2 0x8a 0xc1 0xbc
0x99 0x8b 0x90 0x05 0xc1 0xc7 0x93 0xc3 0x97 0xc3 0xa7 0x05 0x83 0x03
0x81 0xc1 0xbc 0x9d 0xc6 0xc4 0xa1 0xa9 0x83 0x6c 0xc7 0x83 0x09 0xc4
0xb2 0xc7 0xc6 0x83 0x02 0xa5 0xc4 0xb7 0xb6 0xb4 0x8f 0xa3 0x9f 0xaa
0x96 0xc5 0xbe 0x83 0xc3 0x8e 0xc6 0x81 0xc1 0xc4 0x83 0x7d 0xb9 0xaa
0xc1 0xb7 0xbb 0xc4 0x81 0xc1 0xc6 0xad 0x83 0xc4 0xa8 0x5d 0xc7 0xc1
0xc5 0x87 0xab 0xb3 0x82 0xc2 0xc4 0x90 0xc5 0x86 0xa3 0x3b 0x83 0xc3
0xc4 0x85 0x8f 0xc6 0x84 0xc0 0xc4 0xc5 0xc7
```

Problem MK – эмулятор калькулятора

- Напишите на языке C эмулятор для этого микрокалькулятора. Вход: файл программы с последовательностью машинных команд и файл ввода с stdin
- Эмулятор должен при обработке каждой команды IN запрашивать ввод с входного потока и выдавать вывод при каждой команде OUT
- Пример:

001.enc: 0x70 0xc7 0xc1 0x87 0x27 0xc5 0x8d 0xc1 0x87 0x6f 0xc5 0xc7

001.in: 104 64

> problem_mk 001.enc < 001.in

112 216 63 111

Problem AS – кодировщик калькулятора

- Вход:

MOVI 112

OUT D

IN B

ADD B, D

- Выход: 0x70 0xc7 0xc1 0x87

Problem AS2 – декодер калькулятора

- Вход:

0x70 0xc7 0xc1 0x87

- Выход:

MOVI 112

OUT D

IN B

ADD B, D

Ассемблер x86: регистры

- Многие регистры имеют задокументированное специальное назначение

| Имя | Специальное значение |
|-----|---------------------------------|
| rax | аккумулятор |
| rbx | указатель на данные в ds* |
| rcx | счётчик цикла или операции |
| rdx | указатель на I/O* |
| rbp | указатель на фрейм |
| rsp | указатель на стек |
| rsi | операнд в строковых операциях |
| rdi | результат в строковых операциях |

| Имя | Специальное значение |
|----------------|-------------------------|
| r8 – r15 | просто регистры |
| eflags | регистр флагов |
| eip | указатель на инструкцию |
| cs | сегмент кода |
| ss | сегмент стека |
| ds, es, fs, gs | сегменты данных |
| cr0, dr0, ... | системная часть |
| mm0, xmm0, ... | расширения |

* в реальности такое назначение не прижилось

Ассемблер x86: имена нижних частей

- Регистры общего назначения в современном x86 64-битные но у них есть отдельные имена для нижних частей.

| |
|----------------|
| rax / rsi / r8 |
|----------------|

| | |
|--|-----------------|
| | eax / esi / r8d |
|--|-----------------|

| | | | |
|--|--|--|---------------|
| | | | ax / si / r8w |
|--|--|--|---------------|

| | | | | |
|--|--|--|----|----|
| | | | ah | al |
|--|--|--|----|----|

 (sil, r8b)

Ассемблер x86: система команд

SUB RAX, 5 // RAX -= 5

| Opcode | Instruction | Op/En | 64-Bit Mode | Compat/Leg Mode | Description |
|------------------|------------------------------|-------|-------------|-----------------|---|
| 2C ib | SUB AL, imm8 | I | Valid | Valid | Subtract imm8 from AL. |
| 2D iw | SUB AX, imm16 | I | Valid | Valid | Subtract imm16 from AX. |
| 2D id | SUB EAX, imm32 | I | Valid | Valid | Subtract imm32 from EAX. |
| REX.W + 2D id | SUB RAX, imm32 | I | Valid | N.E. | Subtract imm32 sign-extended to 64-bits from RAX. |
| 80 /5 ib | SUB r/m8, imm8 | MI | Valid | Valid | Subtract imm8 from r/m8. |
| REX + 80 /5 ib | SUB r/m8 ¹ , imm8 | MI | Valid | N.E. | Subtract imm8 from r/m8. |
| 81 /5 iw | SUB r/m16, imm16 | MI | Valid | Valid | Subtract imm16 from r/m16. |
| 81 /5 id | SUB r/m32, imm32 | MI | Valid | Valid | Subtract imm32 from r/m32. |
| REX.W + 81 /5 id | SUB r/m64, imm32 | MI | Valid | N.E. | Subtract imm32 sign-extended to 64-bits from r/m64. |

Ассемблер x86: условные переходы

- Условный переход происходит как после явного сравнения

```
cmp    edx, 1 // if (x <= 1)
jle    L1      // goto L1;
```

- Так и после обычной арифметики

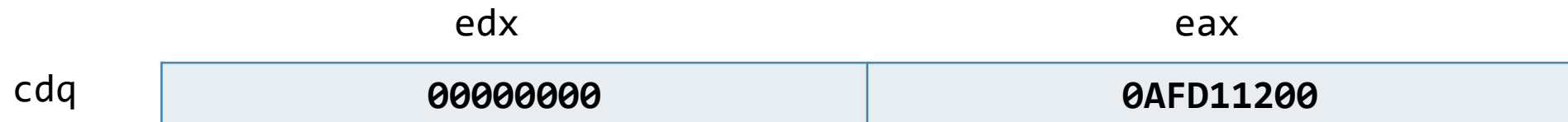
```
sub    edx, 1 // x -= 1;
jne    L3      // if (x != 0)
           // goto L3;
```

- Почти каждая арифметическая операция выставляет флаги

| | | |
|----------|-------------------------|------------------------|
| JE, JZ | if zero == if equal | ZF == 1 |
| JB, JNAE | if less unsigned | CF == 1 |
| JL, JNGE | if less signed | SF != OF |
| JO | if overflow | OF == 1 |
| JS | if sign | SF == 1 |
| JP, JPE | if parity | PF == 1 |
| JGE, JNL | if not less signed | SF == OF |
| JLE, JNG | if not greater signed | ZF == 1 SF != PF |
| JBE, JNA | if not greater unsigned | CF == 1 ZF == 1 |
| JCXZ | if cx is zero | CX == 0 |

Упражнение: что делает somefunc?

```
somefunc:                // somefunc(x, y):
    mov     eax, edi      // eax = x
    mov     edx, esi      // edx = y
L2:
    mov     ecx, edx      // ecx = edx
    cdq                     // sign extend eax to edx:eax
    idiv     ecx          // eax = (edx:eax) / ecx; edx = (edx:eax) % ecx
    mov     eax, ecx      // eax = ecx
    test    edx, edx      // set flags to edx & edx
    jne     L2            // if (edx != 0) goto L3
    ret                   // return eax
```



СЕМИНАР 6.2

Условные переходы и работа с памятью

Обсуждение

- Реалистичные микропроцессоры обычно имеют кроме регистров память и инструкции для работы с ней
- Также они обычно поддерживают отдельные флаговые регистры и **условные переходы** в зависимости от состояния флаговых регистров
- Это делает ассемблерную программу похожей на сишную программу, активно использующую `goto`
- Вот и настало время поговорить о `goto`

Язык С: использование goto

- Нормальная программа

```
int fact(int x) {  
    int acc = 1;  
  
    if (x < 2)  
        return x;  
  
    while (x > 0) {  
        acc = acc * x;  
        x -= 1;  
    }  
  
    return acc;  
}
```

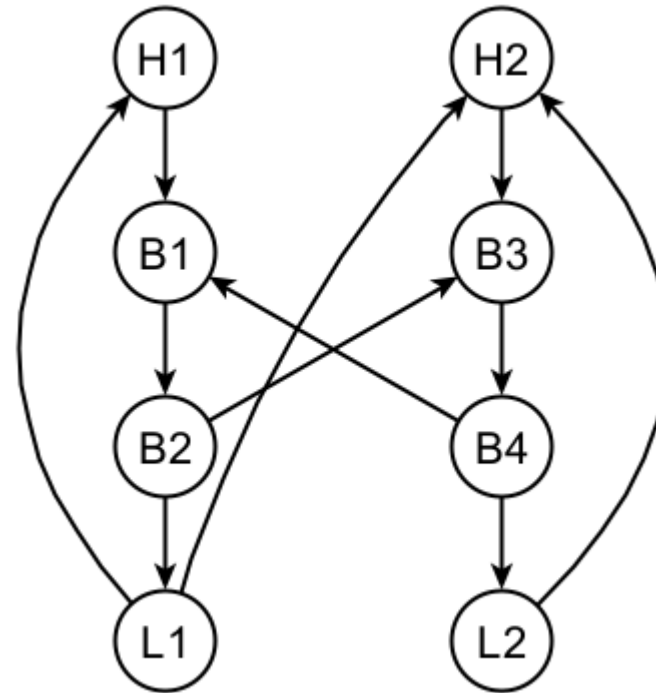
- Программа, близкая к ассемблеру

```
int fact(int x) {  
    int acc = x;  
    x -= 1;  
    if (x < 2) goto ret;  
  
loop:  
    acc = acc * x;  
    x -= 1;  
    if (x > 0) goto loop;  
  
ret:  
    return acc;  
}
```

Внезапный тезис

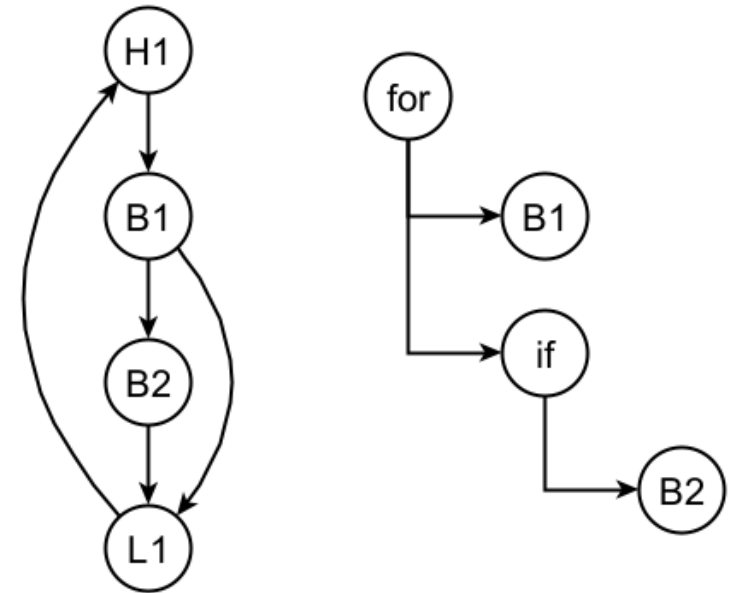
- Переход как концепция куда мощнее, чем циклы и ветвления.

```
for (i = 0; i < UMAX; ++i) {  
    if (upper_cond()) goto lower;  
upper:  
    upper_action();  
}  
  
for (j = 0; j < LMAX; ++j) {  
    if (lower_cond()) goto upper;  
lower:  
    lower_action();  
}
```



goto считать вредным

- Дейкстра обратил внимание на то, что мы не столько пишем текст программы, сколько проектируем процесс её исполнения.
- Далее он поставил задачу сопоставления текста и процесса.
- **Структурная программа** т.е. состоящая только из ветвлений и условных переходов характеризуется предсказуемостью состояния.
- Поэтому при программировании на С мы избегаем goto. А в ассемблере у нас только они и есть.



Ассемблер x86: условные переходы

- Почти каждая арифметическая операция выставляет флаги

`add edx, eax` // `edx += eax` и ставит флаги по `edx`

- Две специальных операции сравнения: `cmp` и `test`

`cmp edx, 1` // ставит флаги как `(edx - 1)`

`test edx, 1` // ставит флаги как `(edx & 1)`

- Регистр флагов содержит пять основных флагов: ZF (zero), SF (sign), OF (overflow), CF (carry), PF (parity).

ZF = [результат операции равен нулю]

SF = [старший бит результата]

OF = [результат не помещается в destination]

Что мы можем сделать из Z, C, O?

ZF = [результат операции равен нулю]

SF = [старший бит результата]

OF = [результат не помещается в destination]

```
cmp    edx, 42    // ставит флаги как (edx - 42)
```

```
je     .L1        // как проверить равенство?
```

```
jnl    .L2        // как проверить знаковое меньше?
```

```
jle    .L2        // как проверить знаковое меньше или равно?
```

```
jg     .L2        // как проверить знаковое больше?
```

0 = 000...000

-1 = 111...111

INT_MAX = 011...111

1 = 000...001

-2 = 111...110

INT_MIN = 100...000

2 = 000...010

-3 = 111...101

Что мы можем сделать из Z, S, O?

ZF = [результат операции равен нулю]

SF = [старший бит результата]

OF = [результат не помещается в destination]

```
cmp    edx, 42    // ставит флаги как (edx - 42)
```

```
je     .L1        // равно ZF == 1
```

```
jnl    .L2        // знаковое меньше SF != OF
```

```
jle    .L2        // их комбинация (SF != OF) || (ZF == 1)
```

```
jg     .L2        // её отрицание (SF == OF) && (ZF == 0)
```

0 = 000...000

-1 = 111...111

INT_MAX = 011...111

1 = 000...001

-2 = 111...110

INT_MIN = 100...000

2 = 000...010

-3 = 111...101

Примеры для jump if less

ZF = [результат операции равен нулю]

SF = [старший бит результата]

OF = [результат не помещается в destination]

```
cmp    edx, 42    // ставит флаги как (edx - 42)
```

```
j1     .L2        // SF != OF
```

| | | |
|----------------|-----------------|---------------------------|
| $5 - (-4) = 9$ | SF = 0, OF = 0, | $(5 < -4) = \text{false}$ |
|----------------|-----------------|---------------------------|

| | | |
|--------------|-----------------|-------------------------|
| $4 - 5 = -1$ | SF = 1, OF = 0, | $(4 < 5) = \text{true}$ |
|--------------|-----------------|-------------------------|

| | | |
|-------------|-----------------|--------------------------|
| $5 - 4 = 1$ | SF = 0, OF = 0, | $(5 < 4) = \text{false}$ |
|-------------|-----------------|--------------------------|

| | | |
|--------------------|-----------------|---------------------------|
| $(-5) - (-4) = -9$ | SF = 1, OF = 0, | $(-5 < -4) = \text{true}$ |
|--------------------|-----------------|---------------------------|

Обсуждение: не забываем test

ZF = [результат операции равен нулю]

SF = [старший бит результата]

OF = [результат не помещается в destination]

```
test  edx, edx    // ставит флаги как (edx & edx)
js    .L4          // jump if SF
```

- Как вы думаете какое условие проверено?
- У нас есть все варианты: jz, jnz, js, jns, jo, jno и т.д. на каждый флаг.
- И не будем забывать про нулевой вариант.

```
jmp    .L1          // jump unconditionally
```


Беззнаковая арифметика

ZF = [результат операции равен нулю]

SF = [старший бит результата]

OF = [результат не помещается в destination]

CF = [перенос в старший разряд или заём из него]

- Новая концепция это carry / borrow flag (CF).
- Перенос / заём MSB на сложении двух unsigned чисел.

$010\dots000 + 010\dots000 = 100\dots000$ // Carry to MSB

$100\dots000 - 010\dots000 = 010\dots000$ // Borrow from MSB

$0 = 000\dots000$ $\text{UINT_MAX} - 2 = 111\dots101$

$1 = 000\dots001$ $\text{UINT_MAX} - 1 = 111\dots110$

$2 = 000\dots010$ $\text{UINT_MAX} = 111\dots111$

Что мы можем сделать из Z, S, O, C?

ZF = [результат операции равен нулю]

SF = [старший бит результата]

OF = [результат не помещается в destination]

CF = [перенос в старший разряд или заём из него]

```
cmp    edx, 42    // ставит флаги как (edx - 42)
```

```
je     .L1        // равно не зависит от знака ZF == 1
```

```
jnl    .L2        // знаковое меньше SF != OF
```

```
jb     .L2        // беззнаковое меньше?
```

```
jae    .L2        // беззнаковое больше или равно?
```

```
0 = 000...000    UINT_MAX - 1 = 111...110
```

```
1 = 000...001    UINT_MAX      = 111...111
```

Что мы можем сделать из Z, S, O, C?

ZF = [результат операции равен нулю]

SF = [старший бит результата]

OF = [результат не помещается в destination]

CF = [перенос в старший разряд или заём из него]

```
cmp    edx, 42    // ставит флаги как (edx - 42)
```

```
je     .L1        // равно не зависит от знака ZF == 1
```

```
jnl    .L2        // знаковое меньше SF != OF
```

```
jb     .L2        // CF == 1 беззнаковое меньше
```

```
jae    .L2        // CF == 0 беззнаковое больше или равно
```

```
ja     .L2        // CF == 0 || ZF == 0 беззнаковое больше
```

```
jbe    .L2        // CF == 1 || ZF == 1 меньше или равно
```

Problem AGF -- распознавание функции

- Вам будет дан код на ассемблере содержащий только арифметику и условные переходы и будет предложено повторить ту же функцию на C.
- Будет указано в каких регистрах лежат аргументы и где лежит возвращаемое значение.

Концепция эффективного адреса

- Представим что у нас простая функция.

```
int test(int *a, int n) { return a[n + 2]; }
```

- Тогда попытка его адресовать имеет три компонента.

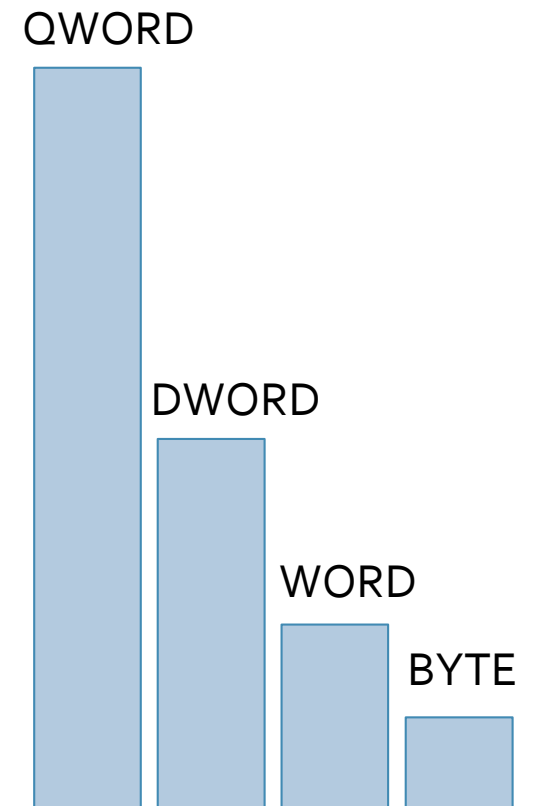
```
a[n + 2] == *((char *)a + n * 4 + 2);
```

EA = **BASE** + **INDEX** * **SCALE** + OFFSET

- Базой адреса или его индексом может быть регистр.

```
mov eax, DWORD PTR [rdi + 8 + rsi * 4]
```

- Для загрузки адреса в регистр служит инструкция lea.



Волшебство инструкции lea

- Формат непрямого доступа.

```
int test(int * a, int n) { return a[n + 2]; }
```

```
mov eax, DWORD PTR [rdi + 8 + rsi * 4]
```

- LEA делает то же, но без доступа к памяти.

```
int * test2(int * a, int n) { return a + n + 2; }
```

```
lea rax, [rdi + 8 + rsi * 4]
```

- Компиляторы очень любят вставлять LEA для обычного сложения.

Распознайте функцию

```
foo:  test  esi, esi           // foo (x, y)
      jle  .L4                // if (y <= 0) goto .L4;
      lea  eax, -1[rsi]        // eax = y - 1;
      lea  rdx, 4[rdi+rax*4]    // rdx = x + rax * 4 + 4;
      xor  eax, eax            // eax = 0;

.L3:  add  eax, DWORD PTR [rdi] // eax += *x;
      add  rdi, 4              // x += 4;
      mov  DWORD PTR -4[rdi], eax // x[-4] = eax;
      cmp  rdi, rdx            // if (x != rdx) goto .L3;
      jne  .L3
      ret                      // return eax;

.L4:  xor  eax, eax            // return 0;
      ret
```

Problem AGM -- опять распознавание

- Вам будет дан код на ассемблере содержащий арифметику, условные переходы и работу с памятью и будет предложено повторить ту же функцию на C.
- Будет указано в каких регистрах лежат аргументы и где лежит возвращаемое значение.

Структура ассемблерного файла (AT&T)

- **Секции**
 - Например секция `text` это код
- **Директивы**
 - Например директива `globl` это внешняя видимость
- **Метки**
 - Используются для вызова функций (метка `fact`) и условных переходов
- **Инструкции**
 - арифметика, логика и переходы
 - см. *[SDM]* для полного списка

```
fact:      .text
           .globl fact

           .cfi_startproc
           movl    4(%esp), %edx
           movl    %edx, %eax
           cmpl    $1, %edx      // if (x == 1)
           jle     L1            // goto L1;
           movl    $1, %eax

L3:         imull   %edx, %eax
           subl    $1, %edx
           jne     L3

L1:         ret
           .cfi_endproc
```

Структура ассемблерного файла (Intel)

- **Секции**
 - Например секция `text` это код
- **Директивы**
 - Например директива `globl` это внешняя видимость
- **Метки**
 - Используются для вызова функций (метка `fact`) и условных переходов
- **Инструкции**
 - арифметика, логика и переходы
 - см. *[SDM]* для полного списка

```
fact:      .text
           .globl fact

           .cfi_startproc
mov     edx, DWORD PTR [esp+4]
mov     eax, edx
cmp     edx, 1      // if (x == 1)
jle     L1          // goto L1;
mov     eax, 1

L3:        imul     eax, edx
           sub     edx, 1
           jne     L3

L1:        ret
           .cfi_endproc
```

Обсуждение

- Какой синтаксис вам больше нравится для `edx = edx - 1`?

1. AT&T: `subl $1, %edx`

2. Intel: `sub edx, 1`

- Какой синтаксис вам больше нравится для `edx = *(esp + 4)`?

1. AT&T: `movl 4(%esp), %edx`

2. Intel: `mov edx, DWORD PTR [esp+4]`

- Если вам больше нравится интеловский синтаксис, подавайте `-masm=intel` для gcc и clang.

Дизассемблер и кодировка

- В дизассемблере вы можете видеть строчки с указанием кодировки

| адрес | кодировка | инструкция | операнды |
|---------|-----------|------------|-----------|
| 4015c6: | 83 ec 20 | sub | esp, 0x20 |

- Как кодируется sub? Сделаем файл subs.s с разными видами вычитаний.

```
$ gcc -c -masm=intel subs.s
```

```
$ objdump -d -M intel subs.o > subs.dis
```

- Что вы можете сказать в результате эксперимента?

кодировка = опкод + операнды

- Попробуйте вычитать большие константы, как изменится опкод инструкции?

Ассемблер x86: кодировка

- Продолжим исследование файла subs.o
- Теперь откроем сам файл любым hex-редактором (можно :%!xxd и далее :%!xxd -r в vim, но лучше WinHex, hiew, dhex, bless или любые иные)

```
00000090: eb04 83ea 0883 ec0c 83ef 1083 e814 83eb  
000000a0: 1883 ea1c 83ec 2083 ef24 9090 2e66 696c
```

- Нет ли тут смутно знакомых последовательностей байт?

Ассемблер x86: кодировка

- Продолжим исследование файла subs.o
- Теперь откроем сам файл любым hex-редактором (можно :%!xxd и далее :%!xxd -r в vim, но лучше WinHex, hiew, dhex, bless или любые иные)

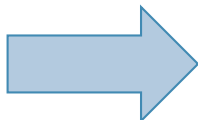
```
00000090: eb04 83ea 0883 ec0c 83ef 1083 e814 83eb
000000a0: 1883 ea1c 83ec 2083 ef24 9090 2e66 696c
```

- Что будет если мы прямо в исполняемом файле что нибудь поменяем?
Например в дизассемблере программы fact видим:

```
401609:      83 7d 08 01      cmp     DWORD PTR [ebp+0x8],0x1
40160d:      7f 13             jg      401622 <_fact+0x26>
```

Главная проблема редактирования кода

- В машинном коде все смещения посчитаны и проставлены

| | | | | | | |
|--|------|-------------------------|-----|----------------|------|-------------------------|
| L3: | mov | edx,DWORD PTR [esp+0x4] | 0: | 8b 54 24 04 | mov | edx,DWORD PTR [esp+0x4] |
| | mov | eax,edx | 4: | 89 d0 | mov | eax,edx |
| | cmp | edx,0x1 | 6: | 83 fa 01 | cmp | edx,0x1 |
| | jle | L1 | 9: | 7e 0d | jle | +13 bytes |
| | mov | eax,0x1 | b: | b8 01 00 00 00 | mov | eax,0x1 |
|  | | | | | | |
| L1: | imul | eax,edx | 10: | 0f af c2 | imul | eax,edx |
| | sub | edx,0x1 | 13: | 83 ea 01 | sub | edx,0x1 |
| | jne | L3 | 16: | 75 f8 | jne | -8 bytes |
| | ret | | 18: | c3 | ret | |

- Если вы измените размер инструкции или вставите новую, вам предстоит вручную менять смещения для всех затронутых переходов

Problem CM: crackme #0

- Используйте файлы `crackme.elf` для Linux и `crackme.exe` для Windows из файлов к семинару (см. zip-архив)
- Вам необходимо дизассемблировать файл, изучить его, после чего используя например `vim` в hex-режиме или любой другой hex-редактор, "сломать" его "защиту". Вывод при успешной модификации файла:
 - > `./cm.out`
 - > `Access granted!`
- Legal disclaimer: УК РФ, статьи 272, 273, 274 явно запрещают нелегальный доступ к компьютерной информации, а также злонамеренную модификацию программного обеспечения. Это не относится к учебным примерам этого семинара, но вы должны иметь это в виду в обычной жизни

СЕМИНАР 6.3

Вызовы функций и ABI

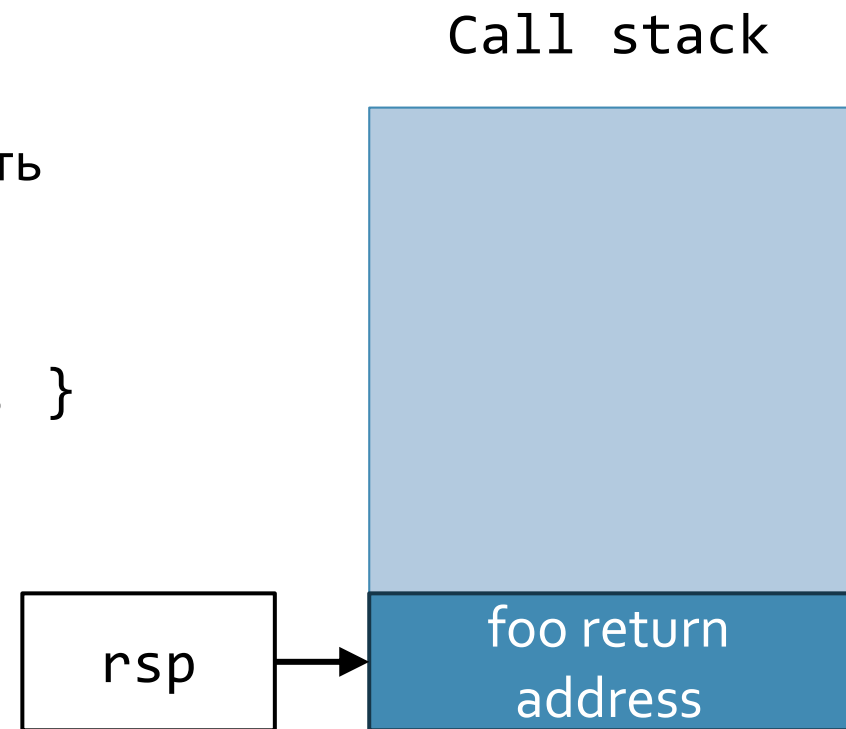
Вызов функций и ABI

- На уровне ассемблера никаких функций не существует, только переходы.
- Поэтому нужно договориться куда складывать параметры и где хранить возвращаемое значение.

```
int foo(int x, int y) { return x + y; }
```

```
lea eax, [rdi + rsi]  
ret
```

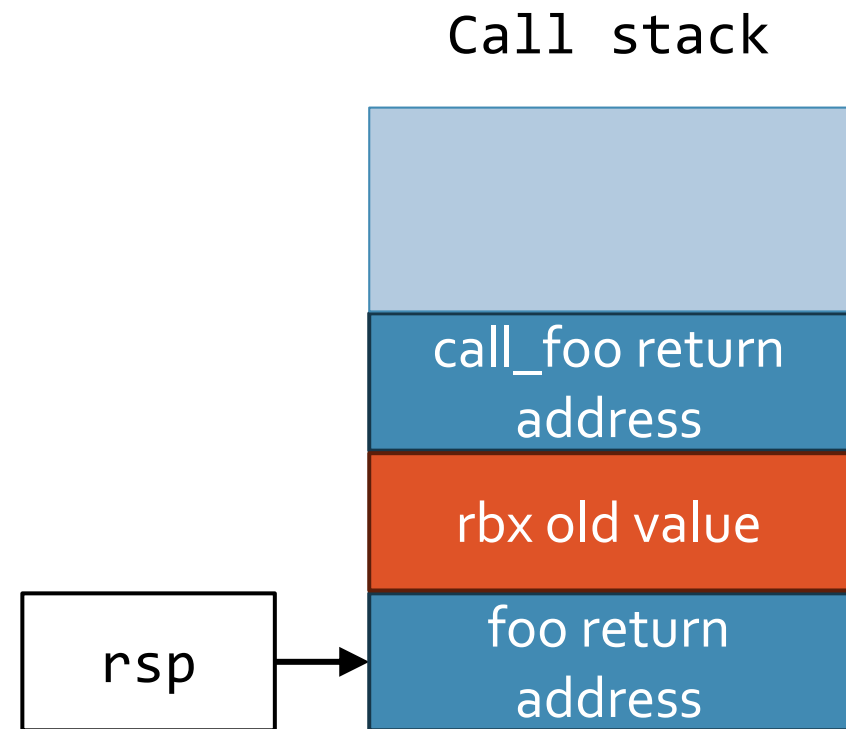
- В x86 обычно аргументы на регистрах, адрес возврата в стеке



Вызов функций и ABI

```
int callfoo(int a, int b) {  
    return foo(b, a) + b;  
}
```

```
callfoo:           // edi = a, esi = b  
    push    rbx  
    mov     ebx, esi // edx = b  
    mov     esi, edi // esi = a  
    mov     edi, ebx // edi = b  
    call    foo      // foo(b, a)  
    add     eax, ebx  // eax = result + b  
    pop     rbx  
    ret
```



call, ret, push и pop

push reg

```
add rsp, size  
mov DWORD PTR [rsp], reg
```

pop reg

```
mov reg, DWORD PTR [rsp]  
sub rsp, size
```

call func

```
push return-address  
jmp func
```

ret

```
pop return-address  
jmp return-address
```

callfoo:

```
push rbx  
mov ebx, esi  
mov esi, edi  
mov edi, ebx  
call foo  
add eax, ebx  
pop rbx  
ret
```

Callee-saved и caller-saved регистры

- Вызывающая функция (caller) обязана вокруг вызова сохранить и восстановить caller-saved регистры которые у неё сейчас активны.
- Вызываемая функция (callee) обязана в прологе и эпилоге сохранить те callee-saved регистры которые она использует.

```
callfoo:
    push rbx // callee-saved
    // .... set(rbx)
    call foo
    // .... use(rbx)
    pop rbx
    ret
```

```
callfoo:
    // .... set(r10)
    push r10 // caller-saved
    call foo
    pop r10
    // .... use(r10)
    ret
```

System V ABI

- Существует довольно много разных конвенций вызова.
- Разумный стандарт де-факто под Unix системами это System V ABI.
- Аргументы закладываются справа налево.
- Очистку передаваемых через стек аргументов делает caller.
- Особое место играет пара rbp / rsp для формирования фрейма.

| | | | |
|-----|------------------|-----|--------|
| rax | ret #1 | r8 | arg #5 |
| rbx | callee | r9 | arg #6 |
| rcx | arg #4 | r10 | |
| rdx | arg #3 ret #2 | r11 | |
| rbp | fp, callee | r12 | callee |
| rsp | sp, callee | r13 | callee |
| rsi | arg #2 | r14 | callee |
| rdi | arg #1 | r15 | callee |

Call sequence

```
0x0000555555555068 <+8>:      mov     esi,0x1
0x000055555555506d <+13>:     xor     edi,edi
=> 0x000055555555506f <+15>:   call    0x555555555190 <callfoo>
```

```
(gdb) x/4x $rsp
```

```
0x7fffffffddfd0: 0x00000000  0x00000000  0xf7dafd90  0x00007fff
```

```
(gdb) stepi 3
```

```
(gdb) x/8x $rsp
```

```
0x7fffffffddde0: 0x00000000  0x00000000  0x55555074  0x00005555
```

```
0x7fffffffddfd0: 0x00000000  0x00000000  0xf7dafd90  0x00007fff
```

Application binary interface experiments

```
long long bar(char a, short b, int c, long d, long long e) {  
    return a + b + c + d + e;  
}
```

```
bar(50, 1500, 18800, (11 << 23), (111 << 46));
```

```
struct S { int x; int y; int z; };
```

```
long long sums(struct S s1, struct S s2, int a) {  
    return s1.x + s2.x + a;  
}
```

```
struct S s1 = {x, x, x}, s2 = {y, y, y};
```

```
sums(s1, s2, 42);
```


Problem AP – дописать часть кода

- В контексте приведена функция наивной проверки числа на простоту на ассемблере (Linux, 64 bit).
- К сожалению, в этом листинге чего-то не хватает, а именно нескольких ассемблерных строчек в указанном месте.
- Допишите их так, чтобы процедура отработала корректно.
- Чтобы облегчить себе задачу попробуйте сделать себе отдельную линковку и сначала написать это на C.

Problem AGS – callers and callees

- Вам задана некая функция caller в System V AMD ABI, вызывающая из себя функцию callee.
- Вам нужно догадаться что это за функция и написать её на языке C.

Мотивация инлайн-ассемблера

- Многие полезные вещи не имеют простого выражения в языке C.
- Пример: подсчёт всех установленных битов в числе.

```
unsigned popcnt(unsigned n) {  
    unsigned mask = 1u << 31, cnt = 0;  
    do {  
        cnt += ((n & mask) == mask);  
    } while ((mask = mask >> 1) != 0);  
    return cnt;  
}
```

- На C сложно придумать что-то лучшее, чем тот или иной цикл. Но в ассемблере у нас есть одна инструкция popcnt.

Инлайн-ассемблер

- Главная проблема использования инлайн-ассемблера программе на С это связать его с остальным кодом.

asm qualifiers (template : output : input : clobber);

```
int myadd(int x, int y) {  
    int res = x;  
    asm ("add %0, %1":"+r"(res):"r"(y));  
    return res;  
}
```

- Констрейнты "+r", "=r", "r", "i", "m" подсказывают респределителю регистров действие.

Вызываем функцию fact

- Пример вызова функции fact.

```
int res;
```

```
// res = fact(y);  
asm("mov edi, %1\n"  
    "\tcall fact\n"  
    "\tmov %0, eax": "=r"(res): "r"(y): "eax");
```

- Клоббер `eax` означает, что этот регистр будет испорчен при вызове.
- Разрывать ассемблерную вставку на три независимых тут является ошибкой т.к. компилятор имеет право таскать код и портить регистры.
- Обратите внимание на конкатенацию литералов.

Инлайн ассемблер: popcnt

- Многие полезные вещи не имеют простого выражения в языке C.
- Пример: подсчёт всех установленных битов в числе.
- Пример использования ассемблерной вставки.

```
unsigned popcnt(unsigned n) {  
    unsigned cnt = 0;  
    asm("popcnt %0, %1" : "=r"(cnt) : "r"(n));  
    return cnt;  
}
```

- Перечислите плюсы и минусы принятого решения.

Обсуждение

- Ассемблерные вставки делают код более оптимальным?
- Увы, это не всегда так. Для достаточно сложной программы вручную обогнать оптимизирующий компилятор – амбициозная задача.
- Хуже того: при эволюции кода ассемблерные вставки навсегда застревают в прошлом и их приходится переписывать, а не перекомпилировать.
- Плюс подумайте о поддержке для разных платформ. Ассемблер сильно отличается даже x86 от x86-64.

Builtins

- Многие полезные вещи не имеют простого выражения в языке C.
- Пример: подсчёт всех установленных битов в числе.

```
unsigned popcnt(unsigned n) {  
    return __builtin_popcount(n);  
}
```

- Идея в том, что компилятор явно умеет некоторые вещи лучше нас, но он **не знает что программист имел в виду**.
- Даже если цикл действительно делает подсчёт установленных бит, об этом сложно догадаться. Билтин же снимает эту неоднозначность.

Оптимизации компилятора

```
unsigned popcnt(unsigned n) {  
    unsigned cnt = 0;  
    while (n) {  
        n &= n - 1;  
        cnt++;  
    }  
    return cnt;  
}
```

```
popcnt:  
    xor     eax, eax  
    xor     edx, edx  
    popcnt  eax, edi  
    test    edi, edi  
    cmovbe  eax, edx  
    ret
```

- Компилятор может сматчить разумную реализацию если узнает её.

```
$ gcc -O2 -march=tigerlake
```

```
$ gcc -O2 -march=native
```

Три альтернативы

- **Компиляторные оптимизации.**
 - Это лучшая альтернатива. Если компилятор может это сматчить и это подпёрто тестом, то вам везёт.
 - Но очень часто вам нужно подать правильный march.
- **Явный builtin.**
 - Тоже неплохо, хотя бы сохраняется кросс-платформенность.
- **Ассемблерная вставка.**
 - Разве что от безысходности.
 - Никогда не пытайтесь руками победить компилятор. Даже если вы тактически выиграете, наказание последует.

Как правильно писать на ассемблере

- Утверждения ниже аннотированы вероятностью того, что они верны
- Вам не надо программировать на ассемблере ($p = 0.8$)
- Вам просто кажется, что надо программировать на ассемблере ($p = 0.16$)
- Возможно вам всё-таки надо что-то написать на ассемблере ($p = 0.04$)
 - Напишите это на C, скомпилируйте и посмотрите на ассемблерный код. После этого скопируйте и при необходимости модифицируйте его ($p = 0.032$)
 - Если этого нельзя написать на C даже через билтины, вам не повезло ($p = 0.008$)
- Не стоит принимать это за догму, это обобщение личного опыта.

СЕМИНАР 6.4

Плавающие числа и стандарт IEEE754. Их поддержка в ассемблере x86. Расширение ABI для плавающих чисел.

Немного о работе компьютера

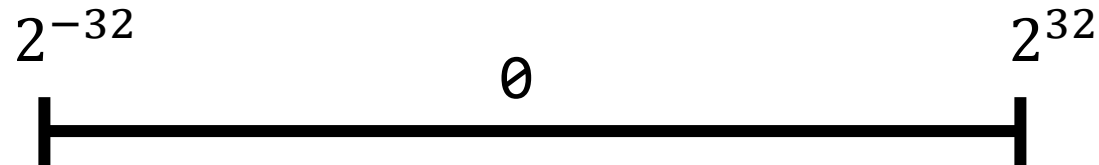
- Память позволяет хранить ограниченное число бит.
- Каждая операция в компьютере производится над ограниченным числом бит.

```
int add(int x, int y) {  
    return x + y;  
                                lea eax, [rdi + rsi]  
                                ret  
}
```

- Естественный способ трактовать ограниченное число бит: как натуральное число. Можно легко закодировать целые.
- Но что делать с вещественными числами?

Обсуждение

- Первая идея кодировка с фиксированной точностью.



- Эта идея иногда находит применение, но в целом она так себе:
 - маленькие диапазоны чисел (64-битное число не больше чем 2^{32}).
 - низкая точность: размер шага не больше, чем 2^{-32} это слишком крупный шаг для многих практических применений.
- Как лучше всего закодировать нечто вроде вещественных чисел с учётом ограниченной точности доступных нам объектов?

Плавающая точность

- Для научных вычислений принято приближать вещественные числа рациональными, используя идею **плавающей точки**.
- Например мы договариваемся, что у нас есть 8 **значащих разрядов**.
- Тогда с плавающей точкой возможны числа:
 1024561 , 102456.1 , 10245.61 , \dots , 10.24561 , 1.024561
- Это было осознано довольно рано и после ряда попыток разной успешности, стандартизовано в 1985 году (см. *[IEEE]*).
- Сейчас это фактический стандарт, от которого крайне редко отступают.

Представление с плавающей точкой

- Представление $\pm 1.frac * 2^{exp-127}$ при этом $(exp > 0) \ \&\& \ (exp < 255)$

| s | exponent | | | | | | | | fractional part of mantissa | | | | | | | | | | | | | | | | | | | | | | |
|----|----------|---|---|---|---|---|---|----|-----------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|--|
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| 31 | | | | | | | | 23 | | | | | | | | | | | | | | | | | | | | | 0 | | |

- Например на рисунке представлено какое-то число
- $exp - 127 = 124 - 127 = -3$
- $mantissa = +1.01b = 1 + \frac{1}{4} = 1.25f$
- Вместе: $1.25 * 2^{-3} = 0.15625f$

Представление с плавающей точкой

- Представление $\pm 1.frac * 2^{exp-127}$ при этом ($exp > 0$) && ($exp < 255$)

| s | exponent | | | | | | | | fractional part of mantissa | | | | | | | | | | | | | | | | | | | | | | |
|----|----------|---|---|---|---|---|---|----|-----------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|--|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| 31 | | | | | | | | 23 | | | | | | | | | | | | | | | | | | | | | 0 | | |

- Например на рисунке представлено какое-то число
- $exp - 127 = 129 - 127 = 2$
- $mantissa = -1.011b = -\left(1 + \frac{1}{4} + \frac{1}{8}\right) = -1.375f$
- Вместе: $-1.375f * 2^2 = -5.5f$

Обсуждение

- У такого формата есть один недостаток: нельзя представить 0
- Найдите самое близкое к нулю число

| s | exponent | | | | | | | | fractional part of mantissa | | | | | | | | | | | | | | | | | |
|----|----------|---|---|---|---|---|---|----|-----------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | 23 | | | | | | | | | | | | | | | | | | 0 |

Обсуждение

- У такого формата есть один недостаток: нельзя представить 0
- Найдите самое близкое к нулю число

| s | exponent | | | | | | | | fractional part of mantissa | | | | | | | | | | | | | | | | | | | | | | |
|----|----------|---|---|---|---|---|---|----|-----------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|--|--|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | |
| 31 | | | | | | | | 23 | | | | | | | | | | | | | | | | | | | | 0 | | | |

- Имеем 2^{-126} это довольно большое число и это точно не ноль

Денормализованные числа

- Специальное значение $\text{exp} = 0$ соответствует денормализованным числам
- Нормализованные: $\pm 1.frac * 2^{\text{exp}-127}$ при этом $\text{exp} > 0$
- Денормализованные: $\pm 0.frac * 2^{-126}$ при этом $\text{exp} == 0$

| s | exponent | | | | | | | | fractional part of mantissa | | | | | | | | | | | | | | | | | | | | | | |
|----|----------|---|---|---|---|---|---|----|-----------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|--|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | |
| 31 | | | | | | | | 23 | | | | | | | | | | | | | | | | | | | | | 0 | | |

- Благодаря этому число, состоящее из всех нулей это ноль, что интуитивно правильно.
- Контринтуитивно здесь то, что возможен -0.0 , отличающийся от $+0.0$ при побитовом сравнении

Бесконечности

- Экспонента, состоящая из всех единиц ($e = 255$) отображает $\pm\infty$

| s | exponent | | | | | | | | fractional part of mantissa | | | | | | | | | | | | | | | | | | | | | | |
|----|----------|---|---|---|---|---|---|----|-----------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| 31 | | | | | | | | 23 | | | | | | | | | | | | | | | | | | | | | | 0 | |

| s | exponent | | | | | | | | fractional part of mantissa | | | | | | | | | | | | | | | | | | | | |
|----|----------|---|---|---|---|---|---|----|-----------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 31 | | | | | | | | 23 | | | | | | | | | | | | | | | | | | | | | 0 |

Концепция not-a-number

- Число, представляющее собой неопределённость (например результат деления нуля на ноль) называется NaN

| s | exponent | | | | | | | | fractional part of mantissa | | | | | | | | | | | | | | | | | | | | | | |
|----|----------|---|---|---|---|---|---|----|-----------------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | |
| 31 | | | | | | | | 23 | | | | | | | | | | | | | | | | | | | | | | 0 | |

- NaN это $\text{exp}=255$ и любая ненулевая fraction
- Первый бит fraction отличает qNaN от sNaN, но это не часть семантики языка
- Сравнение чего угодно с NaN даёт false, в т. ч. NaN не равен сам себе (и поэтому он не число)

Problem EX – работа с битами чисел

- Вам дано число с плавающей точкой типа float (мантисса 23 нижних бита).
- Вы должны получить из него другое число с плавающей точкой, инвертировав все нечётные биты мантиссы (нулевой бит считается чётным) и напечатать с точностью до пятого знака после запятой.

Обсуждение

- Разница между приведением и трактовкой как биты очень велика

```
float x = 1.0f;
```

```
unsigned uval = (unsigned) x; // i == 1
```

```
unsigned u = *(unsigned *) &x; // i == 0x3f800000?
```

- Проблема в том что красным выделена ошибка. Приведение может быть:
 - К байтам: `char*`, `unsigned char*` и т.п.
 - К совместимым типам: от `int` к `unsigned`.
 - Ещё возможно снятие / добавление константности и т.п.
- Всё остальное незаконно, но, к сожалению, компилируется.

Strict aliasing

- Правилами языка запрещён алиасинг двумя разными типами на один объект.

```
float f = 1.0f;  
unsigned *pu = (unsigned *) &f; // strict aliasing violation
```

- Правильный вариант это использовать memcpy.

```
unsigned as_uint(float f) {  
    unsigned u; memcpy(&u, &f, sizeof(unsigned)); return u;  
}  
  
unsigned u = as_uint(f); // ok
```

Упражнения

- Запишите в формате floating-point число `1.0f`
- Число `0.1f` не представимо в формате floating-point точно. Постройте лучшее возможное приближение. Насколько оно хорошо?
- Число $\frac{1}{3}$ тоже нельзя точно представить. Какое будет лучшее приближение этого числа?
- Охарактеризуйте все числа, которые можно представить точно. Например что вы скажете о числе `0.0361328125f`?
- Какое расстояние между самым большим по модулю нормализованным числом и следующим за ним?

Концепция ulp

- "Unit in the last place" это расстояние между двумя последовательными числами с плавающей точкой
- Например посчитаем `ulp(1.0f)`

```
float d0, d1;  
d0 = 1.0f;  
d1 = nextafterf(d0, d0 + 1.0f);  
printf("%.8f", d1 - d0); // на экране 0.00000012
```

- Все вычисления над парой чисел $z = x \text{ (op) } y$ должны быть округлены в пределах $0.5 * \text{ulp}(z)$

Важность округления

- Результат, полученный после арифметической операции внутри `ulr`, должен быть округлён к ближайшему представимому значению.
- Округлять можно вверх, вниз, к нулю и к ближайшему.
- Для выставления метода округления используется функция `fesetround`.

```
float a = 1.0, b = 3.0;
```

```
fesetround(FE_UPWARD);      assert(as_uint(a / b) == 0x3eaaaaaab);  
fesetround(FE_DOWNWARD);    assert(as_uint(a / b) == 0x3eaaaaaaa);  
fesetround(FE_TONEAREST);   assert(as_uint(a / b) == 0x3eaaaaaab);  
fesetround(FE_TOWARDZERO);  assert(as_uint(a / b) == 0x3eaaaaaaa);
```

Problem RP – верхняя и нижняя границы

- Пользователь вводит числитель и знаменатель дроби
- На выходе верхняя и нижняя аппроксимации при представлении в формате `float` как два шестнадцатиричных числа: экспонента и дробная часть мантиссы
- Если возможно точное представление, они должны совпадать

input:

1 3

output:

0x7d 0x2aaaaa 0x7d 0x2aaaab

Разная точность

- В языке C поддержаны три уровня точности: float, double, long double
- double это 64-битные плавающие числа.
- long double часто совпадает с double, но иногда (в таких компиляторах как gcc) оно реализуется через extended-precision 80-битные числа.
- Вопрос в каких регистрах хранить 80-битные числа?

| Тип | bits in exponent | bits in fraction | significant decimal digits |
|--------------|------------------|------------------|----------------------------|
| float | 8 | 23 | 7-8 |
| double | 11 | 52 | 15-16 |
| long double* | 16 | 64 | 20-21 |

Сопроцессор и плавающие числа

- Обработка плавающих чисел на заре x86 происходила в отдельном FPU.
- Сейчас это давно не так, но 32-битный ассемблер остался с тех времён.

```
double foo(double f1,  
           double f2) {  
    double f3;  
    f1 = 1.0 - f1;  
    f1 = f2 * f1;  
    f3 = f1 / 3.0;  
    f2 = f1 + f3;  
    return f2;  
}
```

```
fld1  
fsub QWORD PTR [esp+4]  
fmul QWORD PTR [esp+12]  
fld st(0)  
fdiv DWORD PTR .LC1  
faddp st(1), st
```

st(0)

st(1)

st(2)

st(3)

st(4)

st(5)

st(6)

st(7)

Работа со стеком

f1 = 1.0 - f1;

```
fld1 // push 1.0 to register stack  
fsub QWORD PTR [esp+4] // st(0) -= mem
```

f1 = f2 * f1;

```
fmul QWORD PTR [esp+12] // st(0) *= mem
```

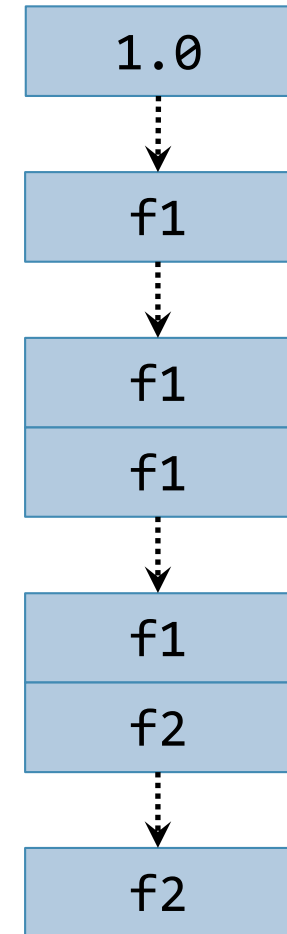
f3 = f1 / 3.0;

```
fld st(0) // push st(0) to stack  
fddiv DWORD PTR .LC1 // st(0) /= mem
```

f2 = f1 + f3;

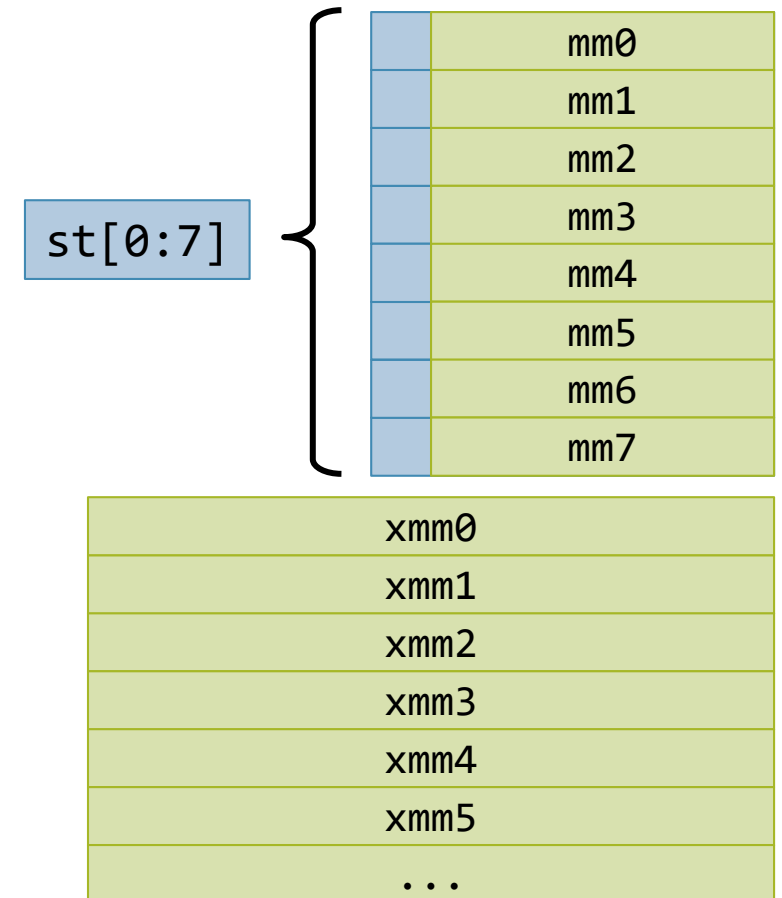
```
faddp st(1), st // st(1) += st(0), pop st
```

return f2;



Расширения регистров: MMX и SSE

- Сопроцессор с отдельным стеком это историческая редкость, конечно
- Для эффективной работы с плавающей точкой, расширение MMX добавило в архитектуру восемь 64-битных регистров MM0-MM7 отображающихся на старый стек сопроцессора, т.е. MM0 это 64-битная часть `st[0]`
- На самом деле, чистый MMX это такая древность, которая уже тоже не встречается
- Расширения SSE, SSE2, SSE3 добавили регистры xmm0-xmm15, отдельные от старого стека, размером в 128 бит



Новый ассемблер с xmm регистрами

```
double foo (double f1, double f2) {  
    double ftmp;  
    f1 = 1.0 - f1; f1 = f2 * f1; ftmp = f1 / 3.0; f2 = f1 + ftmp;  
    return f2;  
}
```

```
fld1  
fsub QWORD PTR [esp+4]  
fmul QWORD PTR [esp+12]  
fld st(0)  
fdiv DWORD PTR .LC1  
faddp st(1), st
```

```
movapd xmm2, xmm0  
movsd xmm0, QWORD PTR .LC0  
subsd xmm0, xmm2  
mulsd xmm1, xmm0  
movapd xmm0, xmm1  
divsd xmm0, QWORD PTR .LC1  
addsd xmm0, xmm1
```

Floating-point добавки в ABI

- Давайте вместе прочитаем System V ABI и попробуем ответить на вопрос как поддержаны fp регистры в x86_64.
- Какие бы вы написали функции чтобы проверить свои предположения?

Problem ARF – распознать FP функцию

```
foo:
    mov     edi, edi
    pxor    xmm1, xmm1
    sub     rsp, 24
    cvtsi2sd xmm1, rdi
    movapd  xmm0, xmm1
    movsd   QWORD PTR [rsp], xmm1
    call    log
    movsd   QWORD PTR [rsp+8], xmm0
    movsd   xmm0, QWORD PTR [rsp]
    call    log
    call    log
    addsd   xmm0, QWORD PTR [rsp+8]
    mulsd   xmm0, QWORD PTR [rsp]
    call    round
    add     rsp, 24
    cvttss2si rax, xmm0
    ret
```

Проблемы FP-оптимизаций

- Из-за проблем с точностью компиляторы вынуждены вести себя консервативно.
- Можно подать `-ffast-math` если точность вас заботит меньше, чем быстродействие.

СЕМИНАР 6.5

Кросс-ассемблеры и различия архитектур.

Kpocc hello world

```
$ cat hello.c
#include <stdio.h>
int main() { printf("Hello, world!\n"); }

$ uname -m
x86_64

$ riscv64-linux-gnu-gcc hello.c -static -o hello.rv.x
$ qemu-riscv64 hello.rv.x
Hello, world!

$ arm-linux-gnueabi-gcc hello.c -static -o hello.arm.x
$ qemu-arm hello.arm.x
Hello, world!
```

Ассемблер ARM

- Регистры x0 ... x31 по 64 бит каждый.
- Регистры w0 ... w31 по 32 бит каждый.
- Интересно что регистр x31 это либо stack pointer либо нулевой регистр в зависимости от инструкции

```
add sp, sp, #0x10
```

```
subs xzr, x0, #0x10
```

- Для удобства лучше пользоваться алиасами xzr и sp.

| Имя | Назначение в ABI |
|---------|--------------------------------|
| x0-x7 | Аргументы |
| x8 | Передача блока памяти |
| x9-x15 | caller-saved |
| x16-x18 | temp |
| x19-x28 | callee-saved |
| x29 | frame pointer |
| x30 | link register |
| x31 | zero register stack pointer |

Факториал

```
fact:                                // fact(int x)
    mov    w1, w0                    // w1 = x;
    mov    w0, 1                     // w0 = 1;
    cmp    w1, w0                    // if (w1 <= 1)
    ble    .L10                      //     goto L10;
.L2:   mov    w2, w1                  // w2 = w1;
    sub    w1, w1, #1                // w1 = w1 - 1;
    mul    w0, w0, w2                // w0 = w0 * w2;
    cmp    w1, 1                     // if (w1 != 1)
    bne    .L2                       //     goto L2;
    ret                                // return w0;
.L10:  mov    w0, w1                  // w0 = w1;
    ret                                // return w0;
```

Режимы адресации в ARM

- Обращение по адресу

`ldr x0, [x1]`

`x0 = *x1;`

- Адресация со смещением

`ldr x19, [sp, 16]`

`x19 = *(sp + 16);`

- Преиндексная адресация

`ldr x1, [x3, 16]!`

`x3 += 16; x1 = *x3;`

- Постиндексная адресация

`ldr x5, [x7], 16`

`x5 = *x7; x7 += 16;`

Режимы адресации

- Адресация со сдвигом

```
ldr    x0, [x1, x2, lsl 16]
```

```
x0 = *(x1 + x2 << 16);
```

- Адресация с расширением

```
ldr    w0, [x0, w1, sextw 2]
```

```
w0 = *(x0 + sext(w1) << 2);
```

- Загрузка сразу двух регистров

```
ldp    x29, x30, [sp], 32
```

```
x29 = *(sp + 32);  
x30 = *(sp + 32 + 8)
```

- PC-relative адресация (загрузка констант с абсолютным адресом)

```
ldr    x1, =num
```

Смысл PC-relative адресации

- В принципе она есть и в x86.

```
int gsym;
```

```
int gret() { return gsym; }
```

- Породит (чтобы влезть в кодировку) попытку дотянуться от `rip`.

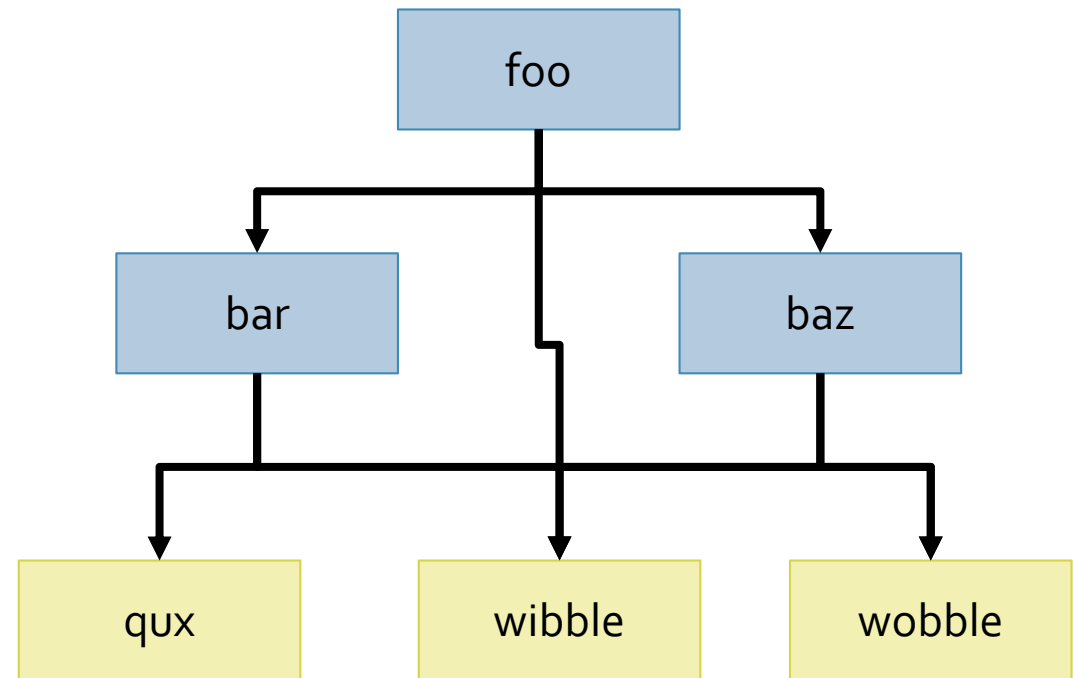
```
mov     eax, DWORD PTR gsym[rip] // pc-relative
```

- Для ARM кодировка плотнее и влезть в неё сложнее.

```
adrp    x0, gsym // загрузка адреса страницы
ldr     w0, [x0, #:lo12:gsym]
```

Идея оптимизации последней функции

- Функции зовут друг друга и должны модифицировать адрес возврата.
- Но листовые функции могли бы брать адрес возврата из спец. регистра.
- Тогда класть этот регистр (он называется линк регистром) на стек нужно только средним функциям.



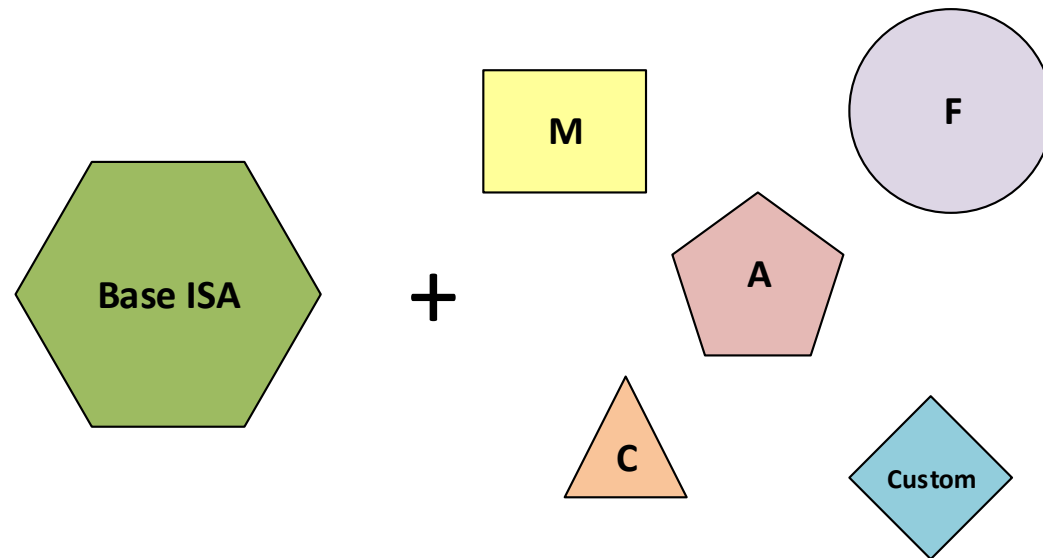
Линк-регистр x30

call_external:

```
    stp    x29, x30, [sp, -32]! // spill x30
    mov    x29, sp
    str    x19, [sp, 16]
    mov    w19, w0
    mov    w0, w1
    mov    w1, w19
    bl     external              // branch and link
                                   // uses x30

    add    w0, w0, w19
    ldr    x19, [sp, 16]
    ldp    x29, x30, [sp], 32    // fill x30
    ret                                // read x30 and return
```

Ассемблер RISC-V



RV32I, RV32E, RV64I, RV128I

| Name | Alias |
|-------------------|---------------|
| x0 | zero |
| x1 | ra |
| x2 | sp |
| x3 | gp |
| x4 | tp |
| x5-x7 | t0-t2 |
| x8-x9 | s0-s1 |
| x10-x15, x16, x17 | a0-a5, a6, a7 |
| x18-x27 | s2-s11 |
| x28-x31 | t3-t6 |

| Jumps & Calls | Loads & Stores | Arithmetics | | | | Special |
|---------------|----------------|-------------|------|-------------|-------|--------------------|
| JAL | LB | ADD | ADDI | ADDW | ADDIW | FENCE |
| JALR | LH | SUB | SUBI | SUBW | SUBIW | ECALL |
| BEQ | LW | OR | ORI | | | EBREAK |
| BNE | LBU | XOR | XORI | | | |
| BLT | LHU | AND | ANDI | | | Upper immediate |
| BGE | SB | SRL | SRLI | SRLW | SRLIW | |
| BLTU | SH | SLL | SLLI | SLLW | SLLIW | LUI |
| BGEU | SW | SRA | SRAI | SRAW | SRAIW | AIUPC |
| | LWU | Data flow | | | | |
| | LD | SLT, SLTU | | SLTI, SLTIU | | |
| | SD | | | | | |

Общая идея data-flow операций

- Data flow в целом тоже есть почти везде.

```
int df(int x, int y) { return (x < y); }
```

```
x86:      setl    al
ARM:      cset    w0, lt
RISC-V:   slt     a0, a0, a1
```

- Внутри ARM и x86 есть ещё select.

```
int df(int x, int y) { return (x < y) ? x : y; }
```

```
x86:      cmovle  eax, esi
ARM:      csel    w0, w1, w0, le
```

Факториал без регистра флагов

```
fact:                                // fact(int x)
    mv      a5, a0                    // a5 = x;
    li      a0, 1                      // a0 = 1;
    li      a3, 1                      // a3 = 1;
    ble     a5, a0, .L8                // if (a5 < a0) goto .L8;
.L2:     mv      a4, a5                // a4 = a5;
    addiw   a5, a5, -1                 // a5 -= 1;
    mulw    a0, a4, a0                 // a0 *= a4;
    bne     a5, a3, .L2                // if (a5 != a3) goto .L2;
    ret                                // return a0;
.L8:     mv      a0, a5                // a0 = a5;
    ret                                // return a0;
```

Problem AGA: угадываем ассемблер ARM

.L6:

```
ldr    w4, [x6, x3, lsl 2]
sub    w5, w2, #1
add    x3, x3, 1
cmp    w4, w0
csinc  w2, w5, w2, ne
cmp    w2, 0
csel   w0, w4, w0, eq
csel   w2, w2, w7, ne
cmp    w1, w3
bgt    .L6
```

Problem AGR: угадываем RISC-V

.L6:

```
lw      a3,0(a5)
add     a2,a6,t1
addiw   a4,a4,-1
ble     a3,t3,.L5
lw      a0,0(a2)
addiw   a7,a7,-1
sw      a3,0(a2)
sw      a0,0(a5)
slli    t1,a7,2
```

.L5:

```
addi    a5,a5,-4
bltu    a1,a4,.L6
```

Обсуждение

- Регистры намного дешевле, чем память.
- Почему бы не делать длинные регистры и не сводить всю программу к вычислениям над их частями?

x86: векторизация на SSE регистрах

```
enum { N = 256 };  
int a[N], b[N], c[N];  
void foo() {  
    int i;  
    for (i = 0; i < N; ++i)  
        a[i] = b[i] + c[i];  
}
```

```
foo:  
    xor     eax, eax  
.L2:  
    movdqa  xmm0, XMMWORD PTR b[rax]  
    paddb   xmm0, XMMWORD PTR c[rax]  
    add     rax, 16  
    movaps  XMMWORD PTR a[ra-16], xmm0  
    cmp     rax, 1024  
    jne     .L2  
    ret
```

Интересная идея для векторизации

```
int find(const int *a, int n, int x) {  
    int i;  
  
    for (i = 0; i < n; i++)  
        if (a[i] == x)  
            return i;  
  
    return -1;  
}
```

- Есть ли у нас идеи что мы тут можем выиграть векторизацией?

Ускорение более чем втрое

```
int find_simd(const int *a, int n, int x) {  
    int i, mainsz = (n / 4) * 4;  
    __m128i v = _mm_set1_epi32(x);  
    for (i = 0; i < mainsz; i += 4) {  
        __m128i u = _mm_loadu_si128(a + i);  
        __mmask8 m = _mm_cmp_epi32_mask(v, u, _MM_CMPINT_EQ);  
        if (m != 0)  
            return i + __builtin_ctz(mask);  
    }  
    // некая обработка хвоста  
}
```

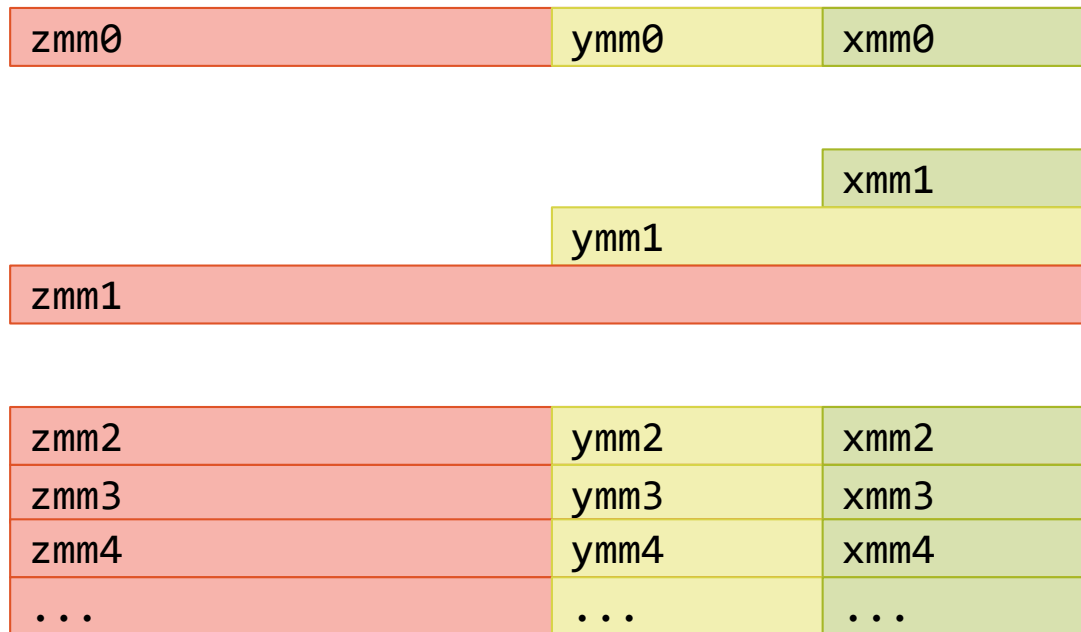

ARM NEON: Q-регистры

- $v0$ то то же, что $q0$, но его можно адресовать частями.

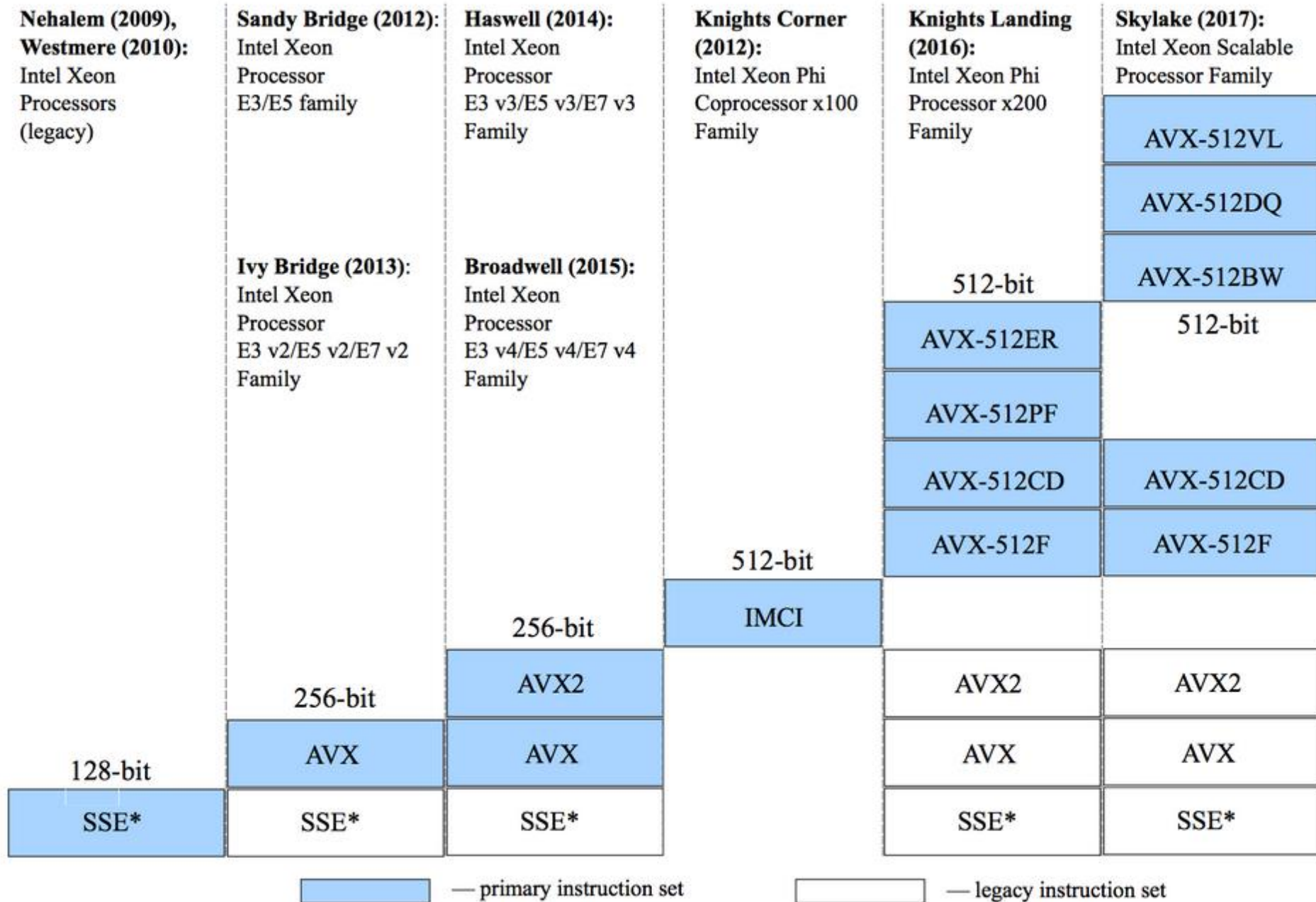
.L2:

```
ldr    q0, [x0, x3]
ldr    q1, [x0, x2]
add    v0.4s, v0.4s, v1.4s
str    q0, [x1, x0]
add    x0, x0, 16
cmp    x0, 1024
bne    .L2
```

Расширения регистров: AVX



- Расширения AVX добавили ymm и zmm регистры, размером 256 и 512 байт соответственно и инструкции для работы с ними



Должны ли вектора алиаситься с fp?

- Для x86 и ARM NEON векторные регистры алиасятся с double precision.
- Для RISC-V векторные регистры вводятся отдельным расширением.

Литература

- [C11] ISO/IEC – "Information technology – Programming languages – C", 2011
- [K&R] Brian W. Kernighan, Dennis Ritchie – The C programming language, 1988
- [SDM] Intel Software Developer Manual: [intel-sdm](#)
- [Linden] Peter van der Linden – Expert C Programming: Deep C Secrets, 1994
- [YH] Юров В., Хорошенко С. – Assembler: учебный курс, 1999
- [ZB] С.В. Зубков – Assembler. Язык неограниченных возможностей, 2007
- [Lomont] Chris Lomont – Introduction to Intel® Advanced Vector Extensions, 2011
- [CAPS] Capabilities of Intel® AVX-512 in Intel® Xeon® Scalable Processors, 2017