

АССЕМБЛЕР

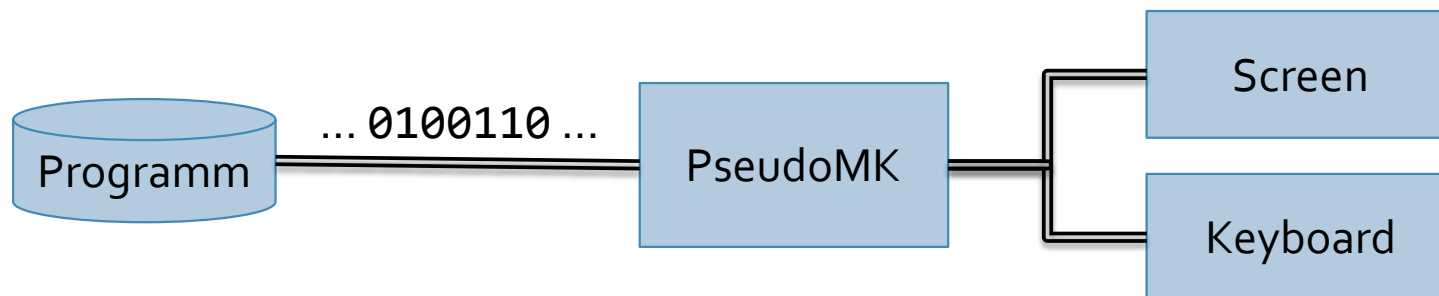
Введение в ассемблер и ассемблирование. Кооперация языка С и ассемблера целевой архитектуры

К. Владимиров, Intel, 2019
mail-to: konstantin.vladimirov@gmail.com

Немного о работе компьютеров

- Дедушкой любого компьютера является электронное арифметически-логическое устройство, то есть калькулятор
- Представим гипотетический целочисленный калькулятор PseudoМК, имеющий внутри четыре регистра A, B, C, D на 8 бит каждый
- Как закодировать команды PseudoМК, 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

- Напишите на языке C кодек (encoder/decoder), то есть программу, берущую на вход последовательность ассемблерных инструкций рассмотренного микрокалькулятора и выдающую машинные коды или наоборот

```
> codec -d 00000011
> MOVDI D, 3
> codec -c "MOVDI D, 3"
> 00000011
```

- Кодировщик должен поддерживать аргументы позволяющие закодировать целый файл

```
> codec -c -f file.asm -o file.bin
```

Ассемблер x86: регистры

- Разумеется, современные микропроцессоры куда сложнее игрушечного калькулятора, рассмотренного ранее
- Регистры общего назначения в современном x86 64-битные но у них есть отдельные имена для нижних частей

<code>rax / rsi / r8</code>

	<code>eax / esi / r8d</code>
--	------------------------------

			<code>ax / si / r8w</code>
--	--	--	----------------------------

			<code>ah</code>	<code>al / sil / r8b</code>
--	--	--	-----------------	-----------------------------

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

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;
}
```

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

```
.text
.globl fact

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

```
.text
.globl fact

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 = MEM[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`

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 следующий код?

```
somefunc:                                     // somefunc(x, y):
    mov     eax, DWORD PTR [esp+4]           // mov eax, x
    mov     edx, DWORD PTR [esp+8]           // mov edx, y
L3:
    mov     ecx, edx
    cmp     eax, edx
    jnb     L2
    mov     ecx, eax
    mov     eax, edx
L2:
    xor     edx, edx
    div    ecx
    mov     eax, ecx
    test    edx, edx
    jne     L3
    ret                                       // return eax
```


Ассемблер x86: кодировка

- В дизассемблере вы можете видеть строчки с указанием кодировки

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

- Как кодируется sub? Посмотрите 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?
- И ключевой вопрос: что будет если мы прямо в исполняемом файле что-нибудь поменяем? Например в дизассемблере программы fact видим:

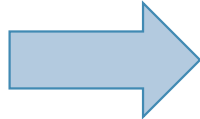
```
401609:      83 7d 08 01      cmp     DWORD PTR [ebp+0x8],0x1
40160d:      7f 13           jg     401622 <_fact+0x26>
```

- Если мы теперь прямо внутри исполняемой программы сменим jg на, скажем, jnge, это изменит логику программы?

Главная проблема редактирования кода

- В машинном коде все смещения посчитаны и проставлены

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



- 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

Глобальные данные

- Ассемблерные файлы разделены на секции (кода, данных и т.д.)
- Любые глобальные данные размещаются в секции данных

```
#include <stdio.h>                                .section .rdata,"dr"
                                                    LC0:
int                                                 .ascii "Hello, world!\0"
main()                                             // .... и так далее
{
    puts("Hello, world!");                        .text
}                                                 // .... тут много кода
                                                    mov    DWORD PTR [esp], OFFSET FLAT:LC0
```

- Видно, что глобальная переменная на уровне ассемблера это метка

Вызов функций и ABI

- На уровне ассемблера никаких функций не существует, только переходы
- Поэтому мы должны договориться куда складывать параметры

```
int x = fact(5);
```

```
4015ce: c7 04 24 05 00 00 00  mov     DWORD PTR [esp], 0x5
4015d5: e8 22 00 00 00        call   4015fc <fact>
4015da: 89 44 24 1c          mov     DWORD PTR [esp+0x1c], eax
```

- В этом фрагменте из fact.gds видно, что передача в 32-битном режиме происходит через стек
- Инструкция call сохраняет на стек адрес возврата и делает переход

Application binary interface

- Регламентирует как именно раскладываются по регистрам и стеку аргументы и как приходит возвращаемое значение
- Поэкспериментируйте с функцией

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

- Вызовите её из main и посмотрите как легли в ассемблер аргументы

```
int res = sumall(50, 1500, 18800, (11 << 23), (111 << 46));
```

Problem KG: keygen #0

- Используйте файлы `crackme.elf` для Linux и `crackme.exe` для Windows из файлов к семинару (см. zip-архив)
- Вам необходимо исследовать логику работы но на этот раз не взломать изменением кода, а вычислить с какими параметрами запустить, чтобы доступ был выдан
- Это задание можно взять на дом, оно потребует анализа ABI для функции `main`

Инлайн-ассемблер в сишных функциях

- Зная конвенции вызова можно написать программу частично на ассемблере

```
int main () {  
    // mov    DWORD PTR [esp], 5  
    // call  _fact  
    // mov    DWORD PTR [esp+4], eax  
    // mov    DWORD PTR [esp], OFFSET FLAT:LC0  
    // call  _printf  
    // mov    eax, 0  
}
```

- Для этого мы просто используем расширение языка:
- `asm ("mov DWORD PTR [esp], 5" : out : in : clobber); // GCC`
- `__asm mov DWORD PTR [esp], 5; // MSVS`

Инлайн-ассемблер в сишных функциях

- Зная конвенции вызова можно написать программу частично на ассемблере

```
int main () {  
    asm("mov  DWORD PTR [esp], 5");  
    // call _fact  
    // mov  DWORD PTR [esp+4], eax  
    // mov  DWORD PTR [esp], OFFSET FLAT:LC0  
    // call _printf  
    // mov  eax, 0  
}
```

- Проблема: до компиляции мы не можем быть уверены добавить ли компилятор лидирующее подчёркивание
- Решение: синтаксис с явными констрейнтами

Инлайн-ассемблер в сишных функциях

- Зная конвенции вызова можно написать программу частично на ассемблере

```
int main () {  
    asm("mov  DWORD PTR [esp], 5");  
    asm("call %0"::"P0"(fact)); // instr : in : out : clobber  
    asm("mov  DWORD PTR [esp+4], eax");  
    // mov  DWORD PTR [esp], OFFSET FLAT:LC0  
    // call _printf  
    // mov  eax, 0  
}
```

- Проблема: между строчками компилятор волен что-нибудь вставить
- Чтобы решить эту проблему используется конкатенация литералов препроцессором

Инлайн-ассемблер в сишных функциях

- Зная конвенции вызова можно написать программу частично на ассемблере

```
int main () {
    asm("mov  DWORD PTR [esp], 5\n"
        "call %0\n"
        "mov  DWORD PTR [esp+4], eax\n"::"P0"(fact));
    // mov  DWORD PTR [esp], OFFSET FLAT:LC0
    // call _printf
    // mov  eax, 0
}
```

- Проблема: до компиляции мы не можем быть уверены куда компилятор назначит строковый литерал

Инлайн-ассемблер в сишных функциях

- Мы явно выносим строку и используем аргумент в памяти и явное `lea`

```
const char *name = "Hello, world!\n"; // ok
```

```
int main () {  
    asm("mov DWORD PTR [esp], 5\n"  
        "call %P0\n"  
        "mov DWORD PTR [esp+4], eax\n"  
        "lea eax, %1\n"  
        "mov DWORD PTR [esp], eax\n"  
        "call %P2\n"::"i"(fact), "m"(name[0]), "i"(printf));  
}
```

- Всё равно ничего не работает. Дело в том, что компилятор распределяет `name` в `eax`, но вызов функции внутри тихо портит (клобберит) именно `eax`.

Инлайн-ассемблер в сишных функциях

- Используем клоббер-лист и в итоге всё срастается

```
const char *name = "Hello, world!\n"; // ok
```

```
int main () {  
    asm("mov DWORD PTR [esp], 5\n"  
        "call %P0\n"  
        "mov DWORD PTR [esp+4], eax\n"  
        "lea eax, %1\n"  
        "mov DWORD PTR [esp], eax\n"  
        "call %P2\n"::"i"(fact), "m"(name[0]), "i"(printf):"eax");  
}
```

- Как видно из проведённого исследования хорошая вставка на ассемблере нетривиальна. Подробнее: <https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>

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

Оптимизации компилятора

- Ключи оптимизации контролируют какой именно ассемблерный код получится у компилятора. Обычные уровни оптимизации
- -O0 (по умолчанию) – здесь компилятор старается сделать ассемблер наиболее точно воспроизводящий написанное программистом
- -O1 (базовые оптимизации) – применяются простые оптимизации
- -O2 (продвинутые оптимизации) – применяются все основные оптимизации
- -O3 (агрессивные оптимизации) – применяются дополнительные и небезопасные оптимизации, включая векторизацию
- -Os (оптимизировать размер) – применяются оптимизации размера вместо быстродействия

Как правильно писать на ассемблере

- Утверждения ниже аннотированы вероятностью того, что они верны
- **Не надо программировать на ассемблере** ($p = 0.8$)
- **Никогда не 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, вам не повезло** ($p = 0.008$)
- Ну что же, кажется настало время немного попрограммировать на ассемблере...

Problem 1A: подсмотреть регистры

- Распечатайте значения регистров `eax`, `ebx`, `ecx`, `edx` в начале функции `main`
- Обратите особое внимание, чтобы распечатать **именно те значения**, которые были в момент входа в `main`. Проверьте себя в `gdb`

```
> problem_ia
eax = 0x1
ebx = 0x1d2bb4
ecx = 0x759d6bba
edx = 0x1d1c10
```

- Есть соблазн вызвать функцию `printf` тоже из ассемблера, но это довольно сложно.
- Постарайтесь написать минимальную ассемблерную вставку

Problem LG – шпионская функция

- У вас есть бинарный файл и вам надо его модифицировать таким образом, чтобы вставить в начало функции interesting вызов функции spume, причём тело этой функции вы сами должны вставить в бинарник
- Сейчас ограничьтесь распечаткой инкрементирующегося счётчика
- После модификации вывод должен быть такой:

```
> problem_lg  
1 2 3 4 5 6 7 8
```

- Понятно, что так можно не только счётчик распечатывать, но и кредитки воровать. Но мы же честные люди (кроме того см. legal disclaimer к Problem CM)

Builtins

- Многие полезные вещи не имеют простого выражения в языке C
- Пример: подсчёт всех установленных битов в числе. На C сложно придумать что-то лучшее, чем цикл

```
unsigned mask = 1;
int cnt = 0;
for (;;) {
    if ((n & mask) == mask)
        cnt += 1;
    if (mask == (1u << 31))
        break;
    mask = mask << 1;
}
```

Builtins

- Многие полезные вещи не имеют простого выражения в языке C
- Пример: подсчёт всех установленных битов в числе. На C сложно придумать что-то лучшее, чем цикл
- Но весь этот цикл можно легко заменить на один интринсик

```
int cnt = __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 это способ сказать компилятору явно чего от него нужно. И тогда, например в ARM, он может просто использовать инструкцию cnt, которая делает это на аппаратном уровне

Сравнение ассемблера ARM

Использование __builtin_popcount

```
uxtw x0, w0
fmov d0, x0
cnt v0.8b, v0.8b
addv b0, v0.8b
umov w0, v0.b[0]
```

Просто цикл

```
mov w3, w0
mov w2, 32
mov w0, 0
mov w1, 1
.L4:
bics wzr, w1, w3
lsl w1, w1, 1
cinc w0, w0, eq
subs w2, w2, #1
bne .L4
```

Обсуждение

- А что компилятор сделает, если инструкции просто нет?
- Например в x86 нет простой возможности подсчитать установленные биты

Сравнение ассемблера x86

Использование __builtin_popcount

```
mov eax, edi
shr eax
and eax, 1431655765
sub edi, eax
mov eax, edi
and eax, 858993459
shr edi, 2
and edi, 858993459
add edi, eax
mov eax, edi
shr eax, 4
add eax, edi
and eax, 252645135
imul eax, eax, 16843009
shr eax, 24
```

Просто цикл

```
mov ecx, 32
xor eax, eax
mov edx, 1
jmp .L4
.L6:
add edx, edx
.L4:
mov esi, edx
and esi, edi
cmp esi, edx
sete sil
movzx esi, sil
add eax, esi
sub ecx, 1
jne .L6
```


Ассемблер x86, march=nehalem*

Использование __builtin_popcount

```
popcnt eax, edi
```

Просто цикл

```
mov ecx, 32
xor eax, eax
mov edx, 1
jmp .L4
.L6:
add edx, edx
.L4:
mov esi, edx
and esi, edi
cmp esi, edx
sete sil
movzx esi, sil
add eax, esi
sub ecx, 1
jne .L6
```

Сопроцессор и плавающие числа

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

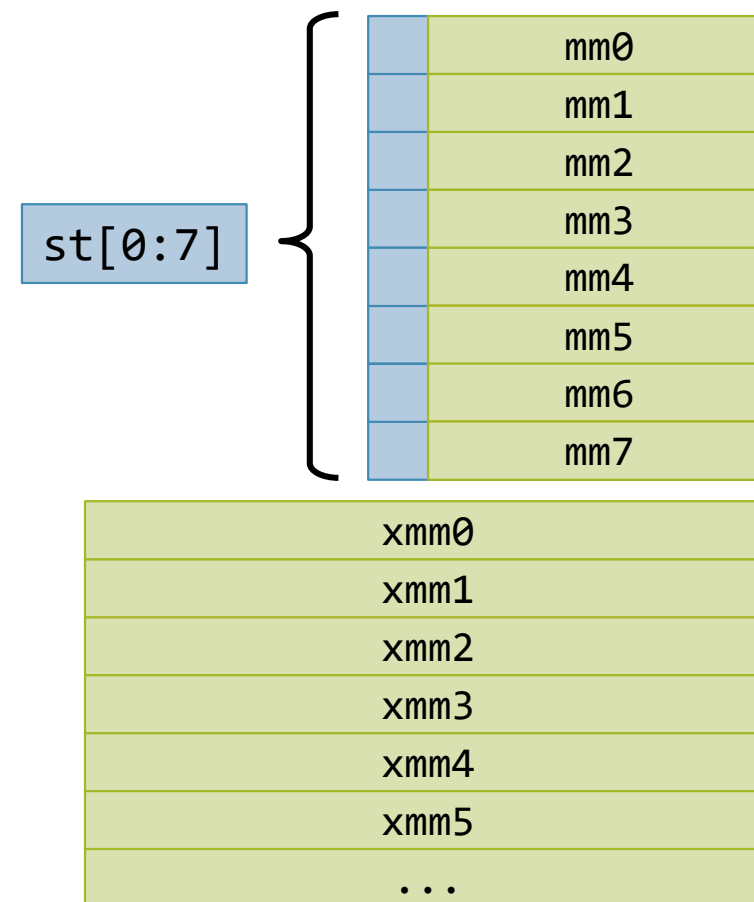
```
double
foo (double f1,
    double f2) {
    double ftmp;
    f1 = 1.0 - f1;
    f1 = f2 * f1;
    ftmp = f1 / 3.0;
    f2 = f1 + ftmp;
    return f2;
}

fld    QWORD PTR [esp+4]
fld    st(0)
fsubr  DWORD PTR LC0
fld    QWORD PTR [esp+12]
fmulp  st(1), st
fxch   st(1)
fdiv   DWORD PTR LC1
faddp  st(1), st
```

st[0]
st[1]
st[2]
st[3]
st[4]
st[5]
st[6]
st[7]

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

- Исходник

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

- Было

```
fld    QWORD PTR [esp+4]
fld    st(0)
fsubr  DWORD PTR LC0
fld    QWORD PTR [esp+12]
fmulp  st(1), st
fxch   st(1)
fdiv   DWORD PTR LC1
faddp  st(1), st
```

- Стало

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

- Важен также тот факт, что больше не надо думать о стеке сопроцессора

Problem RA – инверсный корень снова

- Реальный замер инверсного корня из Problem RI даёт на современных машинах неутешительные результаты
- Ваша задача: заменить всю функцию ассемблерной вставкой с использованием специальной инструкции PFRSQRT
- Обратите внимание: эта сравнительно новая инструкция работает только с xmm регистрами

Векторизация на SSE регистрах

- Наличие широких команд приводит к тому, что циклы выгодно **векторизовать**

.LBB0_1:

```
movdqa xmm0, xmmword ptr [rdx + 4*rax]
movdqa xmm1, xmmword ptr [rdx + 4*rax + 16]
movdqa xmm2, xmmword ptr [rdx + 4*rax + 32]
movdqa xmm3, xmmword ptr [rdx + 4*rax + 48]
padd   xmm0, xmmword ptr [r8 + 4*rax]
padd   xmm1, xmmword ptr [r8 + 4*rax + 16]
padd   xmm2, xmmword ptr [r8 + 4*rax + 32]
padd   xmm3, xmmword ptr [r8 + 4*rax + 48]
movdqa xmmword ptr [rcx + 4*rax], xmm0
movdqa xmmword ptr [rcx + 4*rax + 16], xmm1
movdqa xmmword ptr [rcx + 4*rax + 32], xmm2
movdqa xmmword ptr [rcx + 4*rax + 48], xmm3
add    rax, 16
cmp    rax, 256
jne    .LBB0_1
```

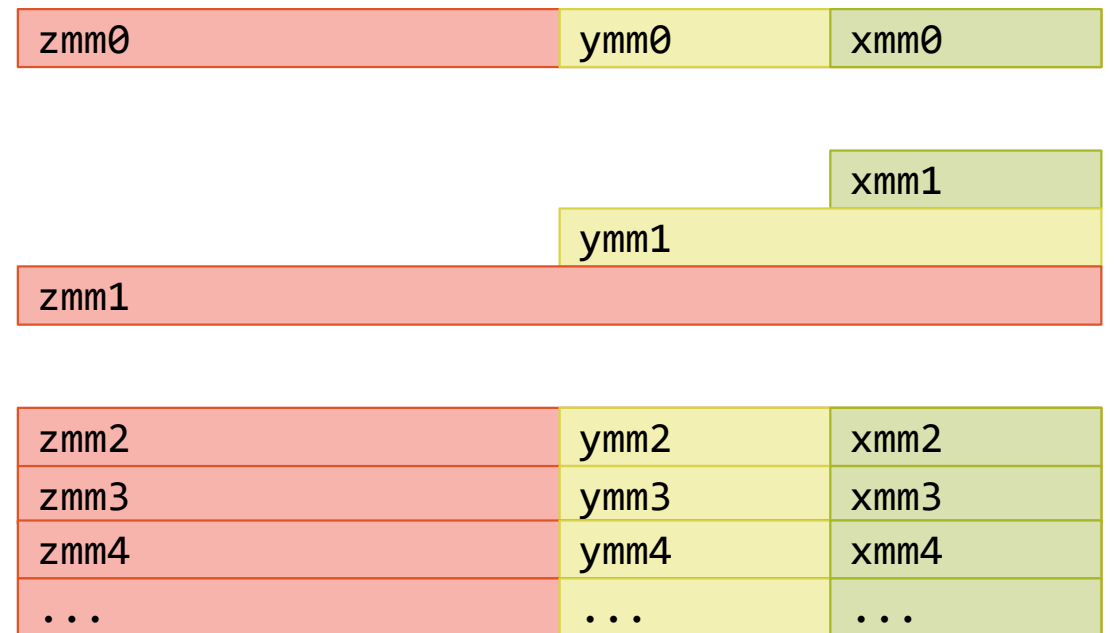
```
int a[256], b[256], c[256];
```

```
void foo () {
    int i;
    // этот цикл может быть векторизован
    // и копировать массивы
    // блоками по 64 байта
    // используя xmm регистры
    for (i=0; i<256; i++){
        a[i] = b[i] + c[i];
    }
}
```

Расширения регистров: AVX

- Расширения AVX добавили ymm и zmm регистры, размером 256 и 512 байт соответственно и инструкции для работы с ними
- Фрагмент векторизации того же кода

```
vmovdqu ymm1, ymmword ptr [rip + b]
vmovdqu ymm2, ymmword ptr [rip + b+32]
vmovdqu ymm3, ymmword ptr [rip + b+64]
vmovdqu ymm8, ymmword ptr [rip + b+96]
vmovdqu ymm4, ymmword ptr [rip + c]
vmovdqu ymm5, ymmword ptr [rip + c+32]
vmovdqu ymm6, ymmword ptr [rip + c+64]
vmovdqu ymm9, ymmword ptr [rip + c+96]
```



**Nehalem (2009),
Westmere (2010):**
Intel Xeon
Processors
(legacy)

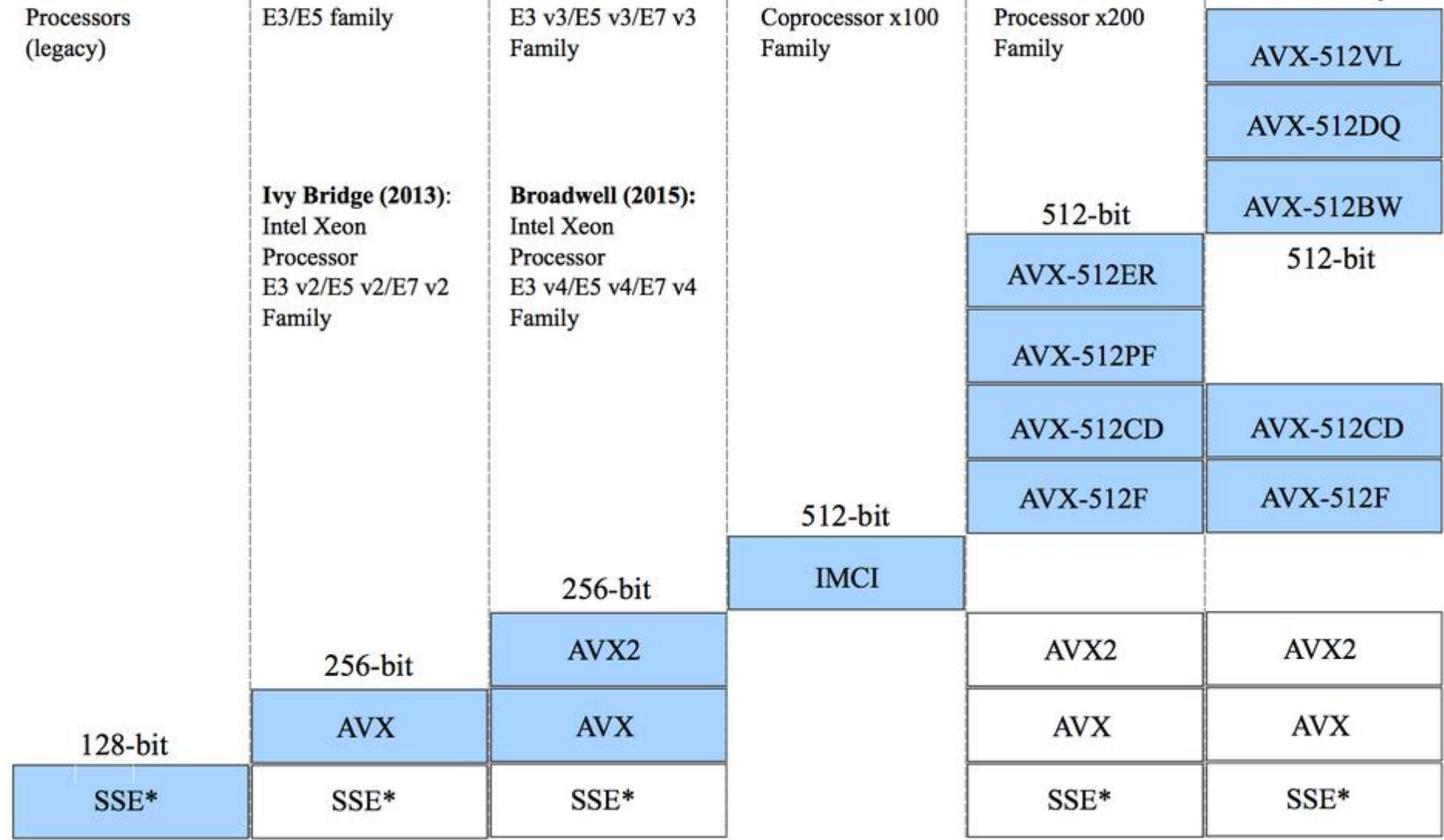
Sandy Bridge (2012):
Intel Xeon
Processor
E3/E5 family

Haswell (2014):
Intel Xeon
Processor
E3 v3/E5 v3/E7 v3
Family

**Knights Corner
(2012):**
Intel Xeon Phi
Coprocesor x100
Family

**Knights Landing
(2016):**
Intel Xeon Phi
Processor x200
Family

Skylake (2017):
Intel Xeon Scalable
Processor Family



— primary instruction set

— legacy instruction set

Домашняя работа HWV – векторизация

- Исследуйте применение векторизации в умножении вектора на матрицу

```
for (i = 0; i < XSZ; ++i) {  
    int acc = 0;  
    for (j = 0; j < YSZ; ++j)  
        acc += vect[j] * matr[j][i];  
    result[i] = acc;  
}
```

- Может ли этот код быть эффективно векторизован?
- Исследуйте его быстроедействие с разными уровнями оптимизациями и размерами векторов и матриц

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