Лекция 0xD

30 марта

Зачем переходить на 64-х разрядную архитектуру?

- Особенности полноценной 64-х разрядной процессорной архитектуры
  - АЛУ оперирует 64-х разрядными данными
  - (Большой) набор 64-х разрядных регистров общего назначения
  - 64-х разрядное (плоское) адресное пространство
- Преимущества 64-х разрядной процессорной архитектуры
  - Эффективнее (быстрее) работаем с 64-х разрядными данными
  - Реже «проливаем» содержимое регистров
  - Огромное пространство адресуемой памяти
    $2^{64} = 16 \times 2^{30} \times 2^{30} =$ 16 Эксбибайт
- Примеры полноценных 64-х разрядных архитектур
  - PowerPC, Sparc, Alpha, IA-64 (Itanium)

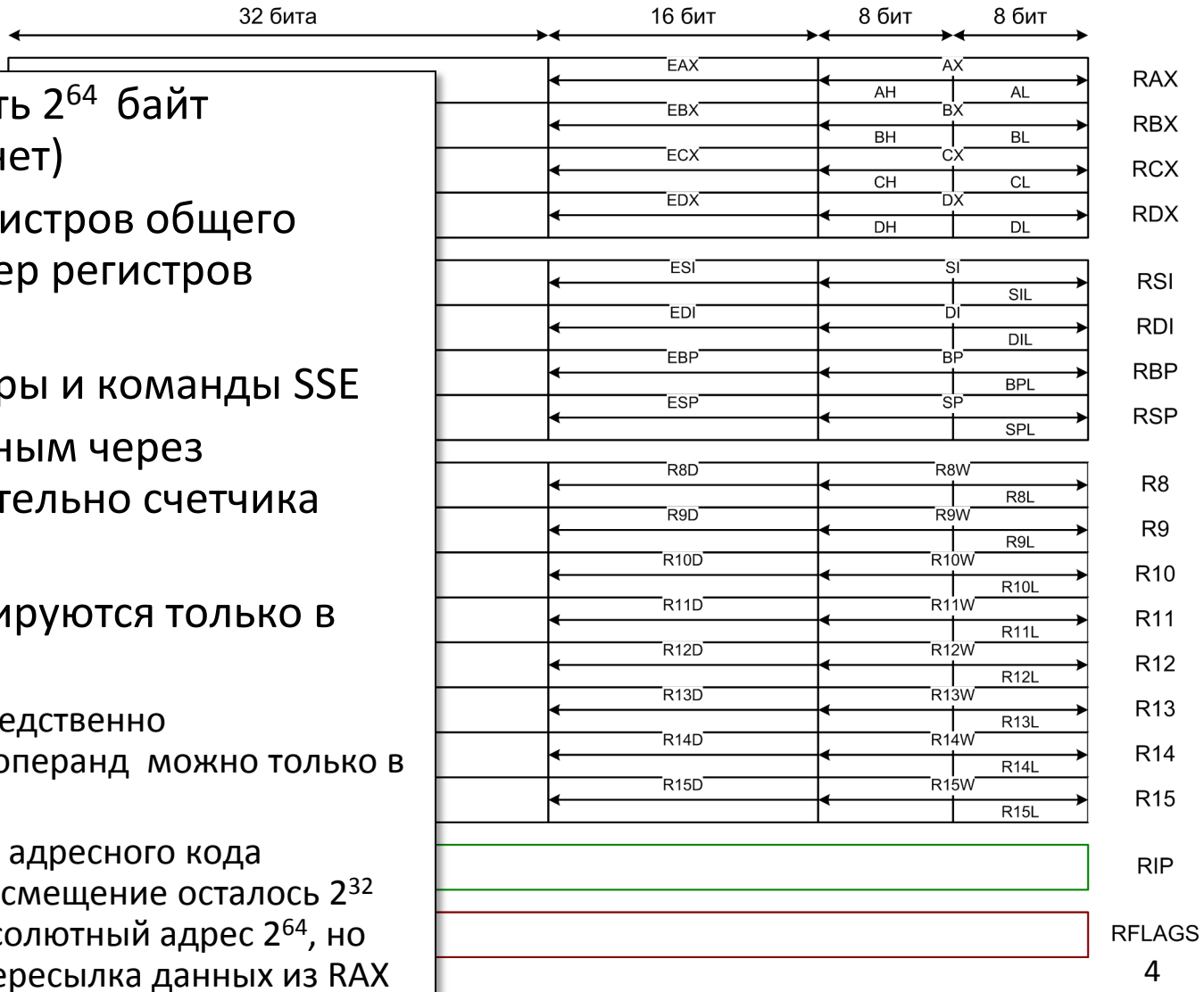
Что такое «Архитектура x86_64»?

- x86_64 едва ли возможно отнести к полноценным 64-х разрядным архитектурам
  - Архитектура x86_64 была получена очередным «эволюционным» расширением ISA IA-32
  - Двоичная кодировка команд IA-32 не изменилась. Работа с 64-х разрядными регистрами и данными реализована через специальные префиксы в коде операции, переопределяющие поведение процессора по умолчанию. Декодирование команд «оптимизировано» для работы с 32-х разрядными командами
  - Доступ к 64-х разрядному адресному пространству реализован через доработанный механизм сегментной памяти IA-32
- Почему AMD/Intel пошли таким путем?
  - Некоторые свойства 64-х разрядных процессоров достигаются, причем переход с 32-х разрядных процессоров получается проще
  - Сохранена работоспособность ISA IA-32, а следовательно – ***всех раннее написанных для данной архитектуры программ***.

The move toward 64-bit computing for mainstream applications, will initially focus on applications that are already constrained by 32-bit memory limitations. … Platforms based on the Intel Xeon processor with Intel EM64T are preferable for general purpose applications, such as Web and mail infrastructure, digital content creation, mechanical computer aided design, and electronic design automation; and for mixed environments in which optimized 32-bit performance remains critical.
Intel. The 64-bit Tipping Point. September 2004 https://software.intel.com/sites/default/files/e1/4f/26944

Архитектура x86-64

- Адресуемая память 2^{64} байт (на самом деле - нет)
- Вдвое больше регистров общего назначения, размер регистров удвоился
- Векторные регистры и команды SSE
- Обращение к данным через смещение относительно счетчика команд
- Величины 2^{64} кодируются только в команде MOV
  - Переслать непосредственно закодированный операнд можно только в регистр
  - В общем формате адресного кода операнда-памяти смещение осталось 2^{32} Можно задать абсолютный адрес 2^{64}, но только если это пересылка данных из RAX



Регистры x86-64: Соглашение по использованию при вызове функций

rax	Возвращаемое значение
rbx	Сохраняется вызванной функцией
rcx	Аргумент #4
rdx	Аргумент #3
rsi	Аргумент #2
rdi	Аргумент #1
rsp	Указатель стека
rbp	Сохраняется вызванной функцией

r8	Аргумент #5
r9	Аргумент #6
r10	Сохраняется вызывающей функцией
r11	Сохраняется вызывающей функцией
r12	Сохраняется вызванной функцией
r13	Сохраняется вызванной функцией
r14	Сохраняется вызванной функцией
r15	Сохраняется вызванной функцией

Единое соглашение для Linux, BSD, OS X, Solaris и компиляторов gcc и icc.

Регистры x86-64

- Аргументы передаются в функцию через регистры
  - Если целочисленных параметров более 6, остальные передаются через стек
  - Регистры-аргументы могут рассматриваться как сохраненные на стороне вызывающей функции
- Все обращения к фрейму организованы через указатель стека
  - Отпадает необходимость поддерживать значения EBP/RBP
- Остальные регистры
  - 6 регистров сохраняется вызванной функцией
  - 2 регистра сохраняется вызывающей функцией
  - 1 регистр для возвращаемого значения
    может рассматриваться как регистр, сохраненный на стороне вызывающей функции
  - 1 выделенный регистр – указатель стека

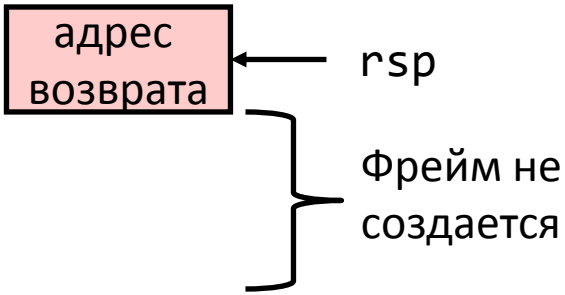
x86-64

Обмен значениями переменных long@x86-64

```
void swap_l(long *xp, long *yp) {
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   mov  rdx,   qword [rdi]
   mov  rax,   qword [rsi]
   mov  qword [rdi],   rax
   mov  qword [rsi],   rdx
   ret
```

- Параметры передаются через регистры
  - Первый параметр (**xp**) был размещен в **rdi**, второй (**yp**) – в **rsi**
  - 64-разрядные указатели
- Никакие команды не работают со стеком (за исключением **ret**)
- Удалось полностью отказаться от использования стека
  - Все локальные данные размещены на регистрах



x86-64

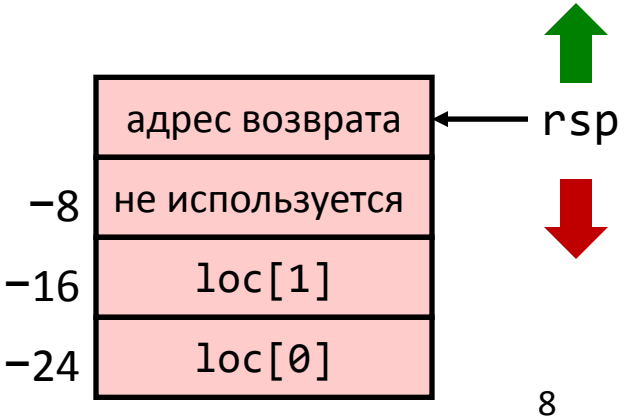
Локальные переменные в «красной зоне»

Листовая функция

```
/*
 * Обмен через локальный массив
 */
void swap_a(long *xp, long *yp) {
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
  mov rax, qword [rdi]
  mov qword [rsp-24], rax
  mov rax, qword [rsi]
  mov qword [rsp-16], rax
  mov rax, qword [rsp-16]
  mov qword [rdi], rax
  mov rax, qword [rsp-24]
  mov qword [rsi], rax
  ret
```

- Обходимся без изменения указателя стека
  - Все данные размещены во «фрейме», неявно организованным под текущим указателем стека
  - Поддержка «красной зоны» явно прописана в ABI x86_64: обработка ***прерываний*** не должна затрагивать содержимое «красной зоны»



Нелистовая функция без организации фрейма

```
/* Обмен a[i] и a[i+1] */
void swap_ele(long a[], int i) {
    swap(&a[i], &a[i+1]);
}
```

- На период работы swap уже никаких значений сохранять на регистрах не требуется
- Не требуется сохранять регистры в качестве вызванной функции
- Команда (префикс) rep используется вместо команды NOP
  - Рекомендации компании AMD…

Источники:
Software Optimization Guide for AMD64 Processors https://support.amd.com/TechDocs/25112.PDF раздел 6.2
Software Optimization Guide for AMD Family 17h Processors https://support.amd.com/TechDocs/55723_SOG_Fam_17h_Processors_3.00.pdf раздел 2.8.1.3.2

```
swap_ele:
   movsx rsi, esi                    ; знаковое расширение i
   lea   rax, [rdi + 8*rsi + 8]      ; &a[i+1]
   lea   rdi, [rdi + 8*rsi]          ; &a[i] первый аргумент
   mov   rsi, rax                    ;       второй аргумент
   call  swap
   rep                               ; пустая команда / НОП
   ret
```

Пример организации фрейма

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
  (long a[], int i) {
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Размещаем значения выражений `&a[i]` и `&a[i+1]` в регистрах, сохраняемых на стороне вызванной функции
- Необходимо сформировать фрейм для сохранения этих регистров

```
swap_ele_su:
   mov   [rsp-16], rbx
   mov   [rsp-8],  rbp
   sub   rsp, 16
   movsx rax, esi
   lea   rbx, [rdi + 8*rax + 8]
   lea   rbp, [rdi + 8*rax]
   mov   rsi, rbx
   mov   rdi, rbp
   call  swap
   mov   rax, [rbx]
   imul  rax, [rbp]
   add   [rel sum], rax
   mov   rbx, [rsp]
   mov   rbp, [rsp+8]
   add   rsp, 16
   ret
```

Для x86-64 может использоваться одна из четырех моделей построения кода
`-mcmodel=[small | medium | large | kernel]`

Как происходит работа с фреймом

```
swap_ele_su:
   mov   [rsp-16], rbx          ; сохраняем rbx
   mov   [rsp-8],  rbp          ; сохраняем rbp
   sub   rsp, 16                ; выделяем на стеке место для фрейма
   movsx rax, esi               ; знаковое расширение i
   lea   rbx, [rdi + 8*rax + 8] ; &a[i+1]
   lea   rbp, [rdi + 8*rax]     ; &a[i]
   mov   rsi, rbx               ; второй аргумент вызова
   mov   rdi, rbp               ; первый аргумент вызова
   call  swap
   mov   rax, [rbx]             ; помещаем в rax a[i+1]
   imul  rax, [rbp]             ; умножаем на a[i]
   add   [rel sum], rax         ; прибавляем к переменной sum
                                ; адрес вычисляется как RIP + sum
   mov   rbx, [rsp]             ; восстанавливаем значение rbx
   mov   rbp, [rsp+8]           ; восстанавливаем значение rbp
   add   rsp, 16                ; освобождаем место занятое фреймом
   ret
```

Особенности работы с фреймом

- Выделение всего фрейма одной командой
  - Обращения к содержимому фрейма используют адресацию относительно `rsp`
  - Уменьшаем значение в указателе стека
  - Выделение памяти может выполняться не сразу, поскольку в определенных временных пределах хранить данные в «красной зоне» безопасно
- Простое освобождение фрейма
  - Увеличиваем значение в указателе стека
  - Указатель фрейма не требуется

Промежуточные итоги
x86-64 : организация вызова функций

- Активное использование регистров
  - Передача параметров
  - Больше регистров – больше возможностей вычислять временные значения и их повторно использовать
- Минимальное использование стека
  - Иногда удается вообще его не использовать
  - Создание/освобождение всего фрейма
- Доступные оптимизации
  - В каком виде будет создан фрейм?
  - Как именно будет выполняться создание?

Далее…

- ***Функции***
  - *Рекурсия*
  - *Выравнивание стека*
  - *Различные соглашения о вызове функций*
    - *cdecl/stdcall/fastcall, отказ от указателя фрейма*
    - *Соглашение вызова для x86-64*
  - ***Переполнение буфера, эксплуатация ошибок, механизмы защиты***
- Динамическая память
- Числа с плавающей точкой

Пример №1 «Заглянуть за горизонт»

```
void f(int i) {
   int a[3] = {1, 2, 3};
   printf("%x\n", a[i]);
}
```

i	Вывод на экран	
0	1	массив a
1	2	
2	3	
3	b7ff1030	«мусор»
4	8049ff4	
5	bffff748	сохраненный ebp
6	804844b	адрес возврата
7	7	фактический аргумент
8	b7fccff4	содержимое фрейма вызывающей функции
9	8048470	

```
f:
   push    ebp
   mov     ebp, esp
   sub     esp, 40
   mov     eax, dword [ebp+8]
   mov     dword [ebp-20], 1
   mov     dword [ebp-16], 2
   mov     dword [ebp-12], 3
   mov     eax, dword [ebp-20+eax*4]
   mov     dword [esp], .LC0
   mov     dword [esp+4], eax
   call    printf
   leave
   ret
```

Пример №2 «Нескучная арифметика»

```
#include <stdio.h>

void f() {
   char buf[5];
   int *ret;

   ret = (int*)(buf + 20);
   (*ret) += 17;
}

void main() {
  int x = 0;

  f();
  x += 1;
  printf("x=%d\n", x);
}
```

gcc 4.9.2

```
f:
   push    ebp
   mov     ebp, esp
   sub     esp, 16
   lea     eax, [ebp-9]
   add     eax, 20
   mov     dword [ebp-4], eax
   mov     eax, dword [ebp-4]
   mov     eax, dword [eax]
   lea     edx, [eax+17]
   mov     eax, dword [ebp-4]
   mov     dword [eax], edx
   leave
   ret
```

```
$gcc -O0 -o retrape
retrape.c
$objdump -d -M intel retrape
$ ./retrape
0
$
```

- Можно менять константы в функции **f** (выделены красным)
- Как добиться того, чтоб программа напечатала «**0**», а не «**1**»?
- Есть, как минимум, два способа

```
08048439 <main>:
 ...
 8048444:       89 e5                    mov    ebp,esp
 8048446:       51                       push   ecx
 8048447:       83 ec 14                 sub    esp,0x14
 804844a:       c7 45 f4 00 00 00 00     mov    DWORD PTR [ebp-0xc],0x0
 8048451:       e8 c5 ff ff ff           call   804841b <f>
 8048456:       83 45 f4 01              add    DWORD PTR [ebp-0xc],0x1
 804845a:       83 ec 08                 sub    esp,0x8
 804845d:       ff 75 f4                 push   DWORD PTR [ebp-0xc]
 8048460:       68 00 85 04 08           push   0x8048500
 8048465:       e8 86 fe ff ff           call   80482f0 <printf@plt>
 ...
```