**Лекция 16**

<https://godbolt.org/> - чудо сайт, позволяет посмотреть ассемблерный код программы

**Ассемблер** — язык команд процессора.

В процессоре есть ячейки памяти — **регистры**.

**Основные из них:**

**%rax** — аккумулятор: регистр, который обычно используется для хранения значений арифметических операций.

**%rbx** — base: optionally used as base pointer.

**%rcx** — counter: обычно используется как счетчик цикла.

**%rdx** — data: для данных.

rax — 64-х битный регистр a (ax — 16-битное представление, eax — 32-х битное)

**%rsp** — указатель на вершину стека

**%rbp** — указывает на начало фрейма стека

**%rsi** — используется, чтобы передать второй аргумент в функцию

**%rdi** — используется, чтобы передать первый аргумент в функцию

Грубо говоря, есть основные 4 ячейки памяти, и какие-то указатели на стек.

**Рассмотрим функцию:**

void f() {

}

Вход в функцию и выход из нее генерирует некоторые ассемблерные команды:

push rbp

mov rbp, rsp

nop

pop rbp

ret

При заходе в функцию обновляем stack pointer и base pointer.

**push** — положить на стек

**mov** — положить значение первой ячейки во вторую ячейку

**pop** — снять со стека

**ret** — выйти из функции

При выходе из функции, удаляем rbp из стека и выходим.

Заведем переменную:

|  |  |
| --- | --- |
| void f() {  int x = 1;  } | push rbp  mov rbp, rsp  mov DWORD PTR [rbp-4], 1  nop  pop rbp  ret |

Положили число 1 в ячейку, адрес которой равен rbp - 4. (стек растет вниз!!!)

Если завести еще несколько переменных, то каждый раз будем сдвигать адрес, куда положим число, еще на 4.

Для того, чтобы завести число типа double, понадобятся некоторые другие регистры (например **xmm0**).

**add** — добавляет к первой ячейке значение из второй ячейки

mov eax, DWORD PTR [rbp-8]

add DWORD PTR [rbp-4], eax

сначала в eax сохранили значение y = 2; затем к x прибавили y.

Потому что можно только из памяти класть в регистр и из регистра в память, но не из памяти в память.

**DWORD PTR** — обозначение того, что указатель 4-х байтный

**QWORD** — 8-байтный

**cdqe** — команда, которая берет значение в регистре eax и конвертирует его в 64-х битное представление.

Пример:

|  |  |
| --- | --- |
| void f() {  long long x = 1;  int y = 2;  int z = 3;  x+=y;  } | push rbp  mov rbp, rsp  mov QWORD PTR [rbp-8], 1  mov DWORD PTR [rbp-12], 2  mov DWORD PTR [rbp-16], 3  mov eax, DWORD PTR [rbp-12]  cdqe  add QWORD PTR [rbp-8], rax  nop  pop rbp  ret |

**sub** — вычитание

**imul** — умножение (integer nultiplication)

**idiv** — деление

|  |  |
| --- | --- |
| void f() {  int x = 1;  int y = 2;  int z = 3;  x += y;  x -= y;  x \*= z;  z /= y;  } | push rbp  mov rbp, rsp  mov DWORD PTR [rbp-4], 1  mov DWORD PTR [rbp-8], 2  mov DWORD PTR [rbp-12], 3  mov eax, DWORD PTR [rbp-8]  add DWORD PTR [rbp-4], eax  mov eax, DWORD PTR [rbp-8]  sub DWORD PTR [rbp-4], eax  mov eax, DWORD PTR [rbp-4]  imul eax, DWORD PTR [rbp-12]  mov DWORD PTR [rbp-4], eax  mov eax, DWORD PTR [rbp-12]  cdq  idiv DWORD PTR [rbp-8]  mov DWORD PTR [rbp-12], eax  nop  pop rbp  ret |

Деление происходит так: записываем в регистр делимое, расширяем его представление с 4-х байтного до 8-байтного, затем делаем idiv: эта команда принимает 1 аргумент — делитель, затем делим, и в результате неполное частное лежит в eax, а остаток в edx, затем результат обратно в переменную записываем.

Попробуем сделать x\*=5. Получим такой набор команд:

mov edx, DWORD PTR [rbp-4]

mov eax, edx

sal eax, 2

add eax, edx

mov DWORD PTR [rbp-4], eax

Когда умножаем на константу, неоптимально пользоваться imul, так как это дорогая операция. Цена каждой операции измеряется в тактах процессора. Таким образом, дешевле сделать побитовый сдвиг на 2 (умножить на 4), и затем прибавить x еще 1 раз.

**sal** — побитовый сдвиг влево.

**shr** — побитовый сдвиг вправо.

**inc, dec** – ассемблерные инструкции для инкремента и декремента

Рассмотрим такой пример:

|  |  |
| --- | --- |
| void f() {  int x = 32767;  int y = 2;  int z = 3;  x /= 3;  } | push rbp  mov rbp, rsp  mov DWORD PTR [rbp-4], 32767  mov DWORD PTR [rbp-8], 2  mov DWORD PTR [rbp-12], 3  mov eax, DWORD PTR [rbp-4]  movsx rdx, eax  imul rdx, rdx, 1431655766  mov rcx, rdx  shr rcx, 32  cdq  mov eax, ecx  sub eax, edx  mov DWORD PTR [rbp-4], eax  nop  pop rbp  ret |

Чем один раз выполнить команду div, дешевле будет сделать умножение на какое-то большое число, чтобы с учетом переполнения получилось такое число, которое должно было получиться, когда поделим x на 3.

**movsx** — инструкция, которая одновременно и копирует регистр в ячейку памяти, и расширяет регистр до 64-х битного представления.

**Как работают метки?** (которые goto и прыгаем на помеченный участок кода)

|  |  |
| --- | --- |
| void f() {  int x = 32767;  int y = 2;  int z = 3;  mylabel:  x /= 3;  goto mylabel;  } | push rbp  mov rbp, rsp  mov DWORD PTR [rbp-4], 32767  mov DWORD PTR [rbp-8], 2  mov DWORD PTR [rbp-12], 3  .L2:  mov eax, DWORD PTR [rbp-4]  movsx rdx, eax  imul rdx, rdx, 1431655766  mov rcx, rdx  shr rcx, 32  cdq  mov eax, ecx  sub eax, edx  mov DWORD PTR [rbp-4], eax  jmp .L2 |

С точки зрения ассемблера начало функции — тоже метка.

**jmp** — передать исполнение на метку (**безусловный переход**)

**Условный переход:**

|  |  |
| --- | --- |
| void f() {  int x = 32767;  int y = 2;  int z = 3;  mylabel:  x /= 3;  if (x > 4) {  goto mylabel;  }  } | f():  push rbp  mov rbp, rsp  mov DWORD PTR [rbp-4], 32767  mov DWORD PTR [rbp-8], 2  mov DWORD PTR [rbp-12], 3  .L2:  mov eax, DWORD PTR [rbp-4]  movsx rdx, eax  imul rdx, rdx, 1431655766  mov rcx, rdx  shr rcx, 32  cdq  mov eax, ecx  sub eax, edx  mov DWORD PTR [rbp-4], eax  cmp DWORD PTR [rbp-4], 4  jle .L4  jmp .L2  .L4:  nop  pop rbp  ret |

**cmp** — команда сравнения. Результат сравнения записывается в специальный регистр %eflags. Он содержит специальные биты, хранящие состояния процессора и результаты различных последних операций.

**jle** (jump less equal) — проверяет, чему равен соответствующий бит последнего сравнения в регистре флагов, и в зависимости от этого делает прыжок на метку или нет.

**Пример с for:**

|  |  |
| --- | --- |
| void f() {  int x = 32767;  int y = 2;  int z = 3;  for (int i = 0; i < 100; ++i) {  x += 3;  y += x;  }  } | push rbp  mov rbp, rsp  mov DWORD PTR [rbp-4], 32767  mov DWORD PTR [rbp-8], 2  mov DWORD PTR [rbp-16], 3  mov DWORD PTR [rbp-12], 0  jmp .L2  .L3:  add DWORD PTR [rbp-4], 3  mov eax, DWORD PTR [rbp-4]  add DWORD PTR [rbp-8], eax  add DWORD PTR [rbp-12], 1  .L2:  cmp DWORD PTR [rbp-12], 99  jle .L3  nop  nop  pop rbp  ret |

Компилятор имеет право в рамках стекового фрейма (в пределах одной локальной области видимости) перекладывать переменные как ему удобно, поэтому переменную i он положил на стек раньше, чем z, чтобы x, y, i лежали рядом и ими можно было быстро оперировать. В целом он не обязан все переменные класть на стек, какие-то из них компилятор может положить в регистр.

Далее будем писать на **nasm** - Netwide Assembler.

Рассмотрим **функцию проверки числа на простоту** на ассемблере:

section .text  
global is\_prime  
  
is\_prime:  
 cmp eax, 2  
 jb .not\_prime  
 je .is\_prime  
  
 test eax, 1  
 jz .not\_prime  
  
 mov ecx, 3  
 mov ebx, eax ; ebx = n  
  
.check\_loop:  
 mov eax, ecx  
 mul eax ; eax \*= eax  
 cmp eax, ebx ; counter^2 < n ?  
 jg .is\_prime  
  
 mov edx, 0  
 mov eax, ebx  
 div ecx ; eax = n / i, edx = n % i  
 cmp edx, 0  
 je .not\_prime  
  
 add ecx, 2  
 jmp .check\_loop  
  
.is\_prime:  
 mov eax, 1  
 ret  
  
 .not\_prime:  
 xor eax, eax  
 ret

*В nasm комментарии пишутся через ;*

Хотим, чтобы функция принимала аргумент. Где его брать? По соглашению он передается через регистр eax. Результат функции тоже будет записываться в eax.

Выход из функции осуществляется с помощью функций is\_prime и not\_prime: если оно простое, в is\_prime записываем 1 в регистр eax и выходим из функции. Если нет, то в not\_prime зануляем eax с помощью xor с самим собой и выходим из функции.

Рассмотрим функцию is\_prime:

Сначала сравниваем число в регистре eax на равенство с 2. Если да, то прыгаем на is\_prime, которая обеспечит выход из функции. Если нет, прыгаем на not\_prime.

Далее: test eax 1 — проверяем на равенство 1. Если равно, прыгаем на not\_prime.

ecx — счетчик цикла. Начинаем проверять с 3, и каждую итерацию цикла увеличиваем счетчик на 2.

В ebx сохранили изначальное проверяемое число.

В цикле проверяем делители до корня: идем по циклу, пока счетчик в квадрате меньше, чем число. Если счетчик в квадрате оказался больше числа, говорим, что оно простое. Если нет, пробуем делить с остатком. Если остаток равен 0, число не простое, иначе идем дальше по циклу.

С помощью nasm можно превратить наш ассемблерный код в объектный файл:

nasm -f elf64 is\_prime.asm

**Попробуем вызвать эту функцию:**

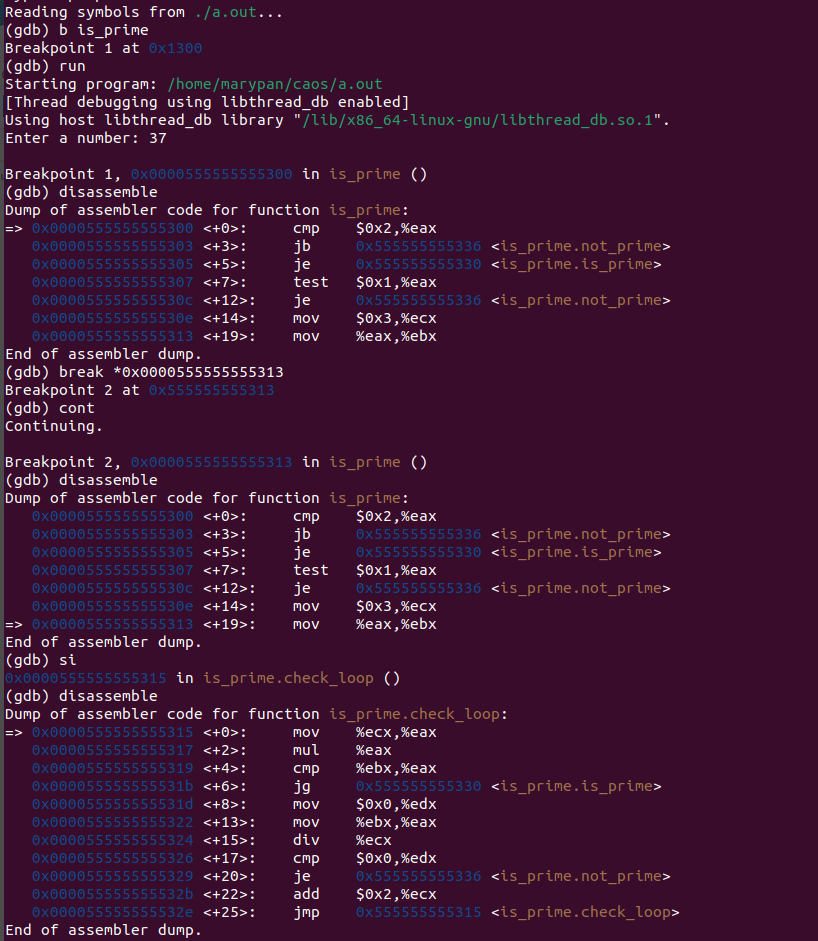
#include <iostream>  
  
// Declare the external assembly function  
extern "C" int is\_prime(int n);  
  
int main() {  
 int number;  
 std::cout << "Enter a number: ";  
 std::cin >> number;  
  
 // Call the assembly function  
 int result = is\_prime(number);  
  
 // Display the result  
 if (result) {  
 std::cout << number << " is a prime number " << std::endl;  
 } else {  
 std::cout << number << " is not a prime number " << std::endl;  
 }  
}

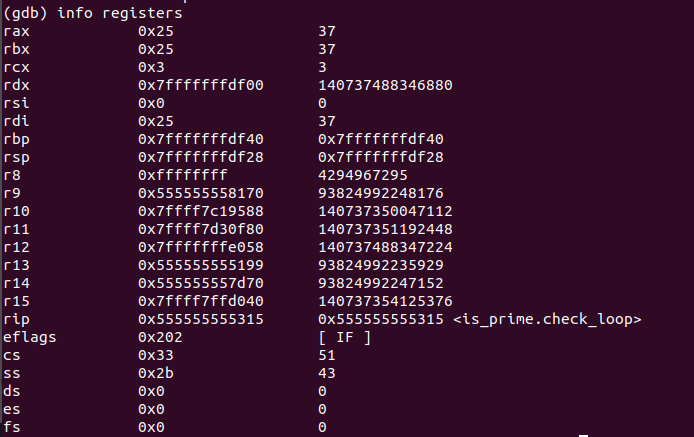
Компилируем вместе с объектным файлом: *g++ is\_prime.cpp is\_prime.o*

Можем зайти дебагером в эту функцию и посмотреть по шагам, как она выполняется.

В gdb можно:

* **disassemble** — смотреть ассемблерный код (там будет синтаксис gnu ассемблера)
* **info registers** — посмотреть чему равны значения регистров
* **break \*0x0000555555555313** (break \*адрес) — ставить брейкпоинты на ассемблерные инструкции
* **si (stepi)** — шагать по инструкциям ассемблера





Все ассемблерные команды кодируются числами. Бинарный код — ассемблер, переведенный в числа.

<https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf> — System V Application Binary Interface: файл, который будет нам помогать изучать ассемблер, это такой стандарт для процессоров AMD64, то есть более менее отражающий всю суть и общие тенденции независимо от процессора.