Лекция 14

ПРО ПРОЦЕССОРЫ, МАШИННЫЙ КОД
А ТАКЖЕ ПРО ЯЗЫК АССЕМБЛЕРА

rev. 5

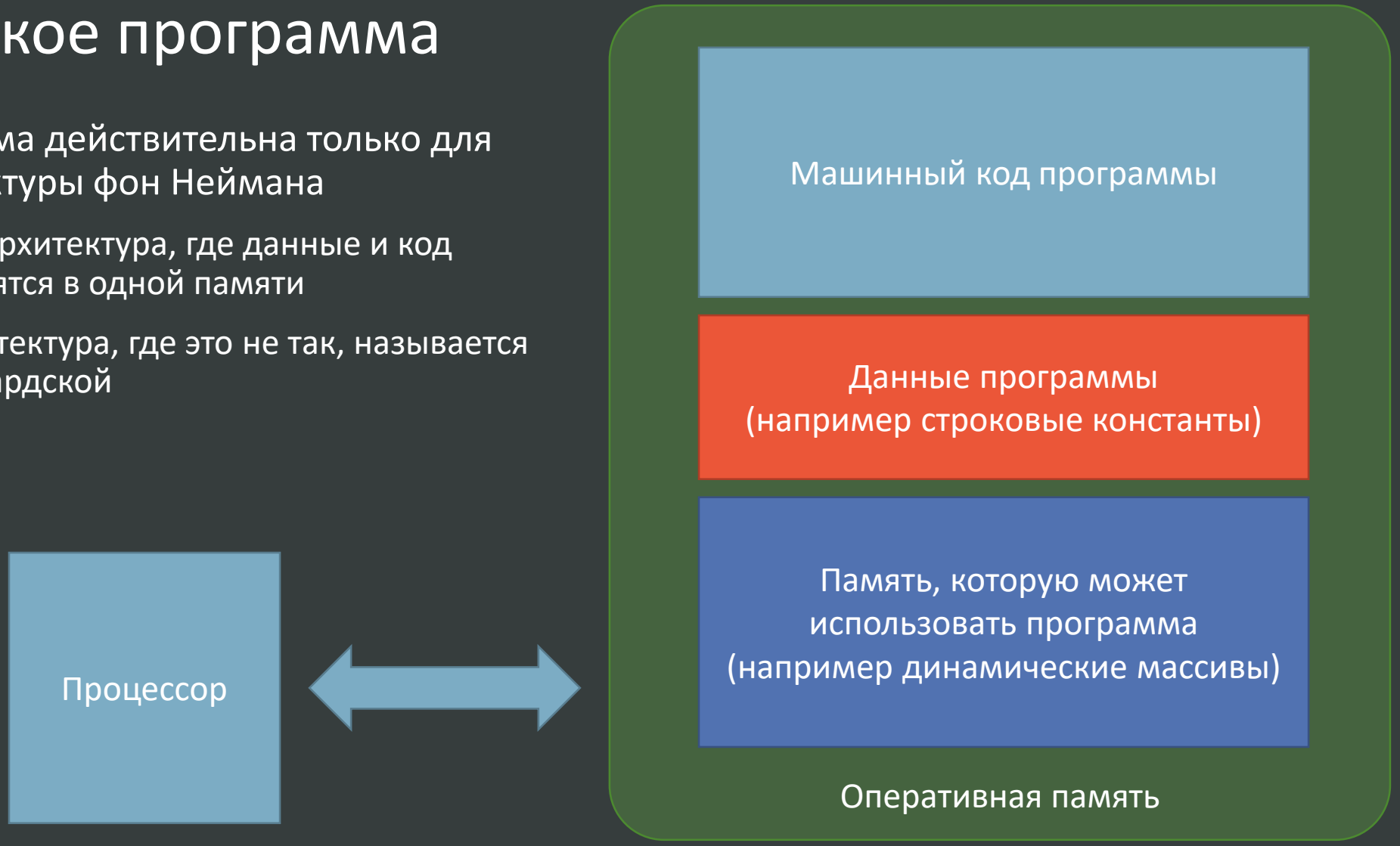
Что такое программа

Что такое программа

- Программа – совокупность инструкций и данных, позволяющая компьютеру решать какие-либо задачи
- Компьютерные инструкции в программе представлены в виде машинного кода, исполняемого напрямую процессором
- В первом приближении можно считать, что программа выполняется исключительно на процессоре (и оперативной памяти)

Что такое программа

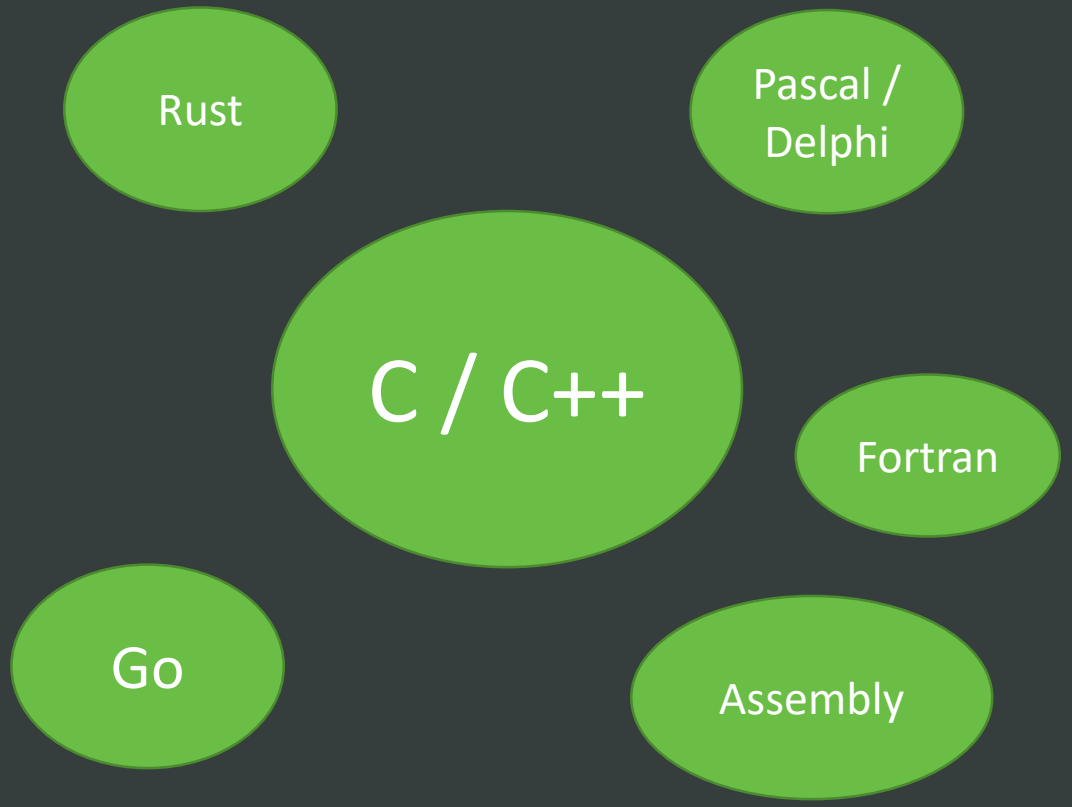
- Эта схема действительна только для архитектуры фон Неймана
  - Это архитектура, где данные и код хранятся в одной памяти
  - Архитектура, где это не так, называется Гарвардской



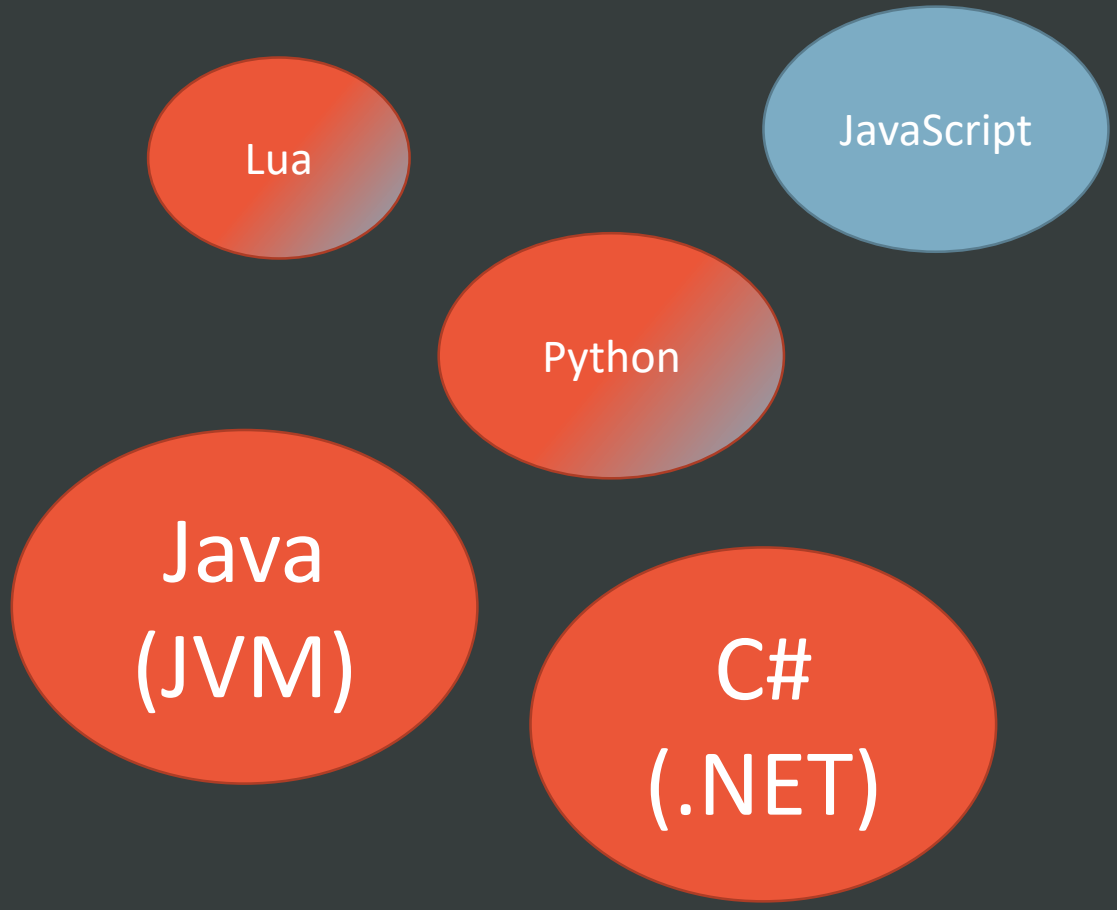
Нативные программы

- Предыдущее определение было дано для так называемых «нативных» программ
- Нативными считаются программы, которые не содержат байт-кода и не подвергаются интерпретации
- Для нативных языков программирования строго обязателен процесс компиляции – преобразования исходного кода в программу
- Файлы .exe или исполняемые файлы Linux – классические программы, соответствующие этому определению
  - Хотя .exe и могут быть написаны на .NET-языках

Нативные языки



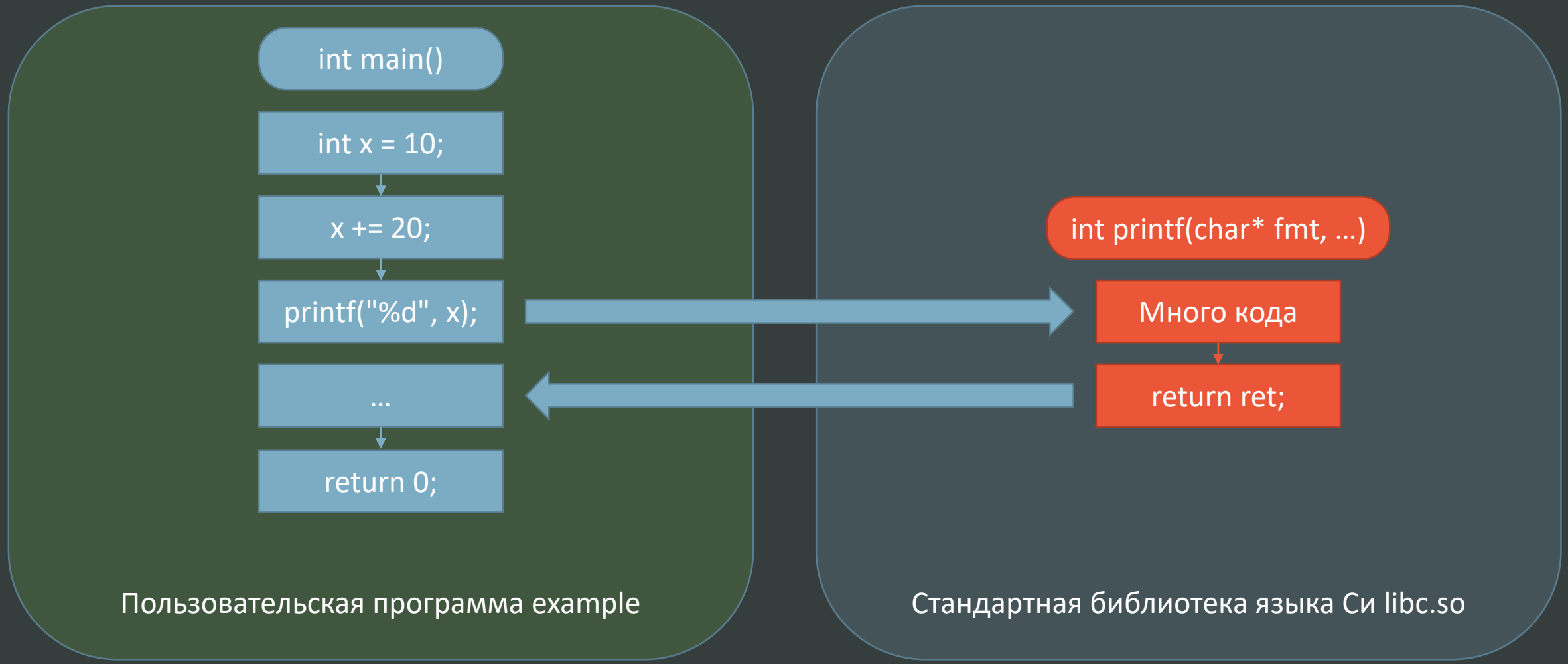
Языки с байткодом или интерпретируемые



Что такое разделяемая библиотека

- Некоторые части программного кода могут быть общими для множества программ, соответственно было бы здорово избавиться от повтора кода в каждой программе
- Разделяемые библиотеки решают эту проблему – они так же как и программы предоставляют данные и машинный код, которые можно использовать в своих программах
  - В Windows и Linux они обычно имеют расширения .dll и .so
- Одним из главных примеров разделяемой библиотеки обычно является libc – стандартная библиотека языка Си
  - Именно в ней хранятся реализации функций вроде printf и malloc
- Более детально с разделяемыми библиотеками мы встретимся ближе к концу семестра

Что такое разделяемая библиотека

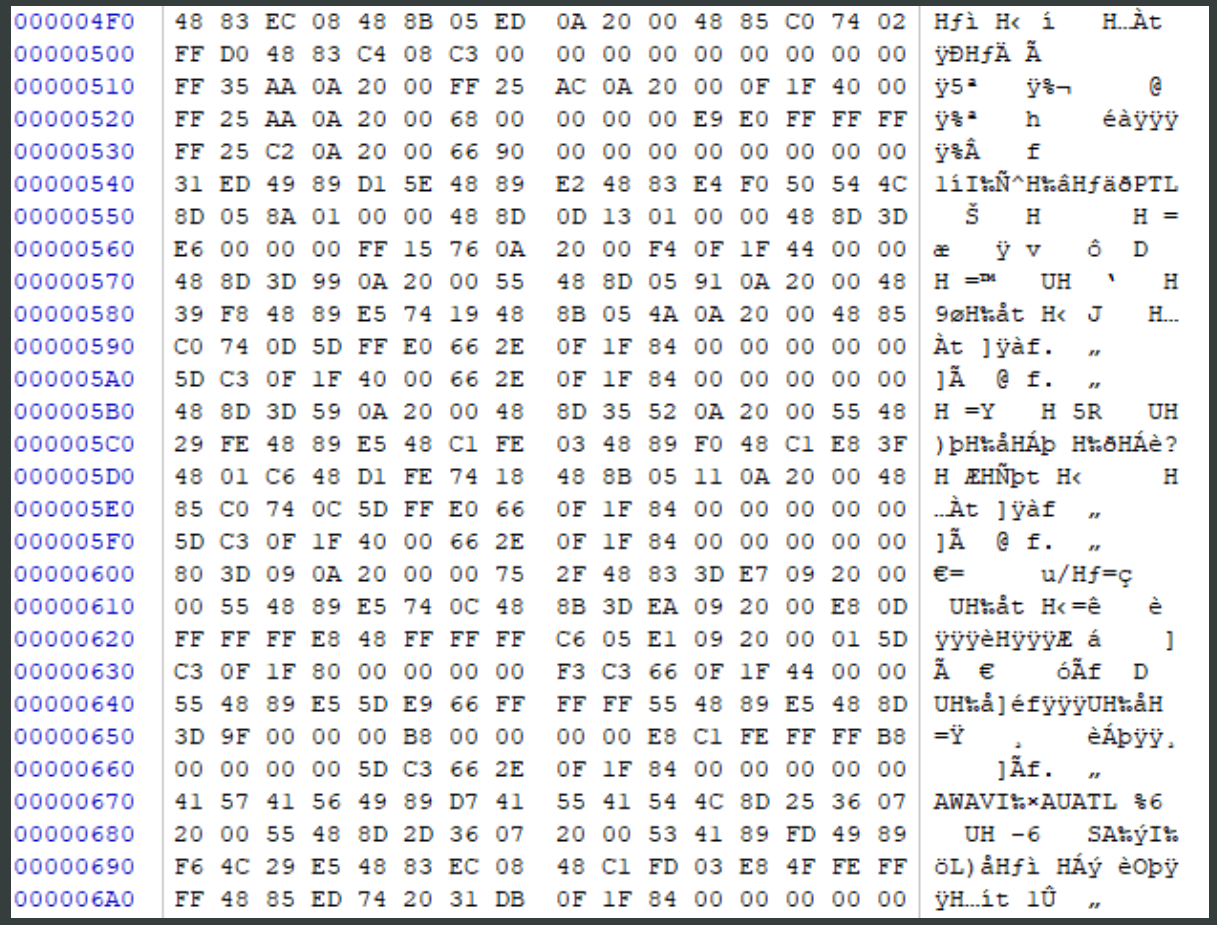


Машинный код и ассемблер

Машинный код

- Самое низкоуровневое представление программного кода
- Обязательно записан в соответствии с какой-то системой команд
  - Обычно у программистов принято отождествлять понятие процессорной архитектуры с системой команд, что мы также далее будем делать
- Не очень удобен для чтения и редактирования, является просто набором байтов

Машинный код



```
000004F0  48 83 EC 08 48 8B 05 ED  0A 20 00 48 85 C0 74 02  Hƒì H‹ í   H…Àt
00000500  FF D0 48 83 C4 08 C3 00  00 00 00 00 00 00 00 00  ÿÐHƒÄ Ã
00000510  FF 35 AA 0A 20 00 FF 25  AC 0A 20 00 0F 1F 40 00  ÿ5ª   ÿ%¬      @
00000520  FF 25 AA 0A 20 00 68 00  00 00 00 E9 E0 FF FF FF  ÿ%ª   h    éàÿÿÿ
00000530  FF 25 C2 0A 20 00 66 90  00 00 00 00 00 00 00 00  ÿ%Â   f
00000540  31 ED 49 89 D1 5E 48 89  E2 48 83 E4 F0 50 54 4C  1íI‰Ñ^H‰âHƒäðPTL
00000550  8D 05 8A 01 00 00 48 8D  0D 13 01 00 00 48 8D 3D    Š   H      H =
00000560  E6 00 00 00 FF 15 76 0A  20 00 F4 0F 1F 44 00 00  æ   ÿ v   ô  D
00000570  48 8D 3D 99 0A 20 00 55  48 8D 05 91 0A 20 00 48  H =™   UH  ‘   H
00000580  39 F8 48 89 E5 74 19 48  8B 05 4A 0A 20 00 48 85  9øH‰åt H‹ J   H…
00000590  C0 74 0D 5D FF E0 66 2E  0F 1F 84 00 00 00 00 00  Àt ]ÿàf.  „
000005A0  5D C3 0F 1F 40 00 66 2E  0F 1F 84 00 00 00 00 00  ]Ã  @ f.  „
000005B0  48 8D 3D 59 0A 20 00 48  8D 35 52 0A 20 00 55 48  H =Y   H 5R   UH
000005C0  29 FE 48 89 E5 48 C1 FE  03 48 89 F0 48 C1 E8 3F  )þH‰åHÁþ H‰ðHÁè?
000005D0  48 01 C6 48 D1 FE 74 18  48 8B 05 11 0A 20 00 48  H ÆHÑþt H‹      H
000005E0  85 C0 74 0C 5D FF E0 66  0F 1F 84 00 00 00 00 00  …Àt ]ÿàf  „
000005F0  5D C3 0F 1F 40 00 66 2E  0F 1F 84 00 00 00 00 00  ]Ã  @ f.  „
00000600  80 3D 09 0A 20 00 00 75  2F 48 83 3D E7 09 20 00  €=     u/Hƒ=ç
00000610  00 55 48 89 E5 74 0C 48  8B 3D EA 09 20 00 E8 0D   UH‰åt H‹=ê   è
00000620  FF FF FF E8 48 FF FF FF  C6 05 E1 09 20 00 01 5D  ÿÿÿèHÿÿÿÆ á    ]
00000630  C3 0F 1F 80 00 00 00 00  F3 C3 66 0F 1F 44 00 00  Ã  €    óÃf  D
00000640  55 48 89 E5 5D E9 66 FF  FF FF 55 48 89 E5 48 8D  UH‰å]éfÿÿÿUH‰åH
00000650  3D 9F 00 00 00 B8 00 00  00 00 E8 C1 FE FF FF B8  =Ÿ   ¸     èÁþÿÿ¸
00000660  00 00 00 00 5D C3 66 2E  0F 1F 84 00 00 00 00 00      ]Ãf.  „
00000670  41 57 41 56 49 89 D7 41  55 41 54 4C 8D 25 36 07  AWAVI‰×AUATL %6
00000680  20 00 55 48 8D 2D 36 07  20 00 53 41 89 FD 49 89    UH -6   SA‰ýI‰
00000690  F6 4C 29 E5 48 83 EC 08  48 C1 FD 03 E8 4F FE FF  öL)åHƒì HÁý èOþÿ
000006A0  FF 48 85 ED 74 20 31 DB  0F 1F 84 00 00 00 00 00  ÿH…ít 1Û  „
```

Ассемблер

- Язык ассемблера является текстовым представлением машинного кода
- Ассемблер – утилита, преобразующая текст программы на языке ассемблера в машинный код
  - Довольно часто (как и в заголовке этого слайда) ассемблером называют и сам язык ассемблера
- Дизассемблер – утилита, осуществляющая обратный процесс: она преобразует машинный код в программу на языке ассемблера
  - Подробнее о дизассемблерах мы поговорим в следующих лекциях
- Как и машинный код, ассемблер сильно зависит от процессорной архитектуры

Ассемблер

- Пример ассемблера и машинного кода:

```
push   rbp                          55
mov    rbp,rsp                      4889E5
lea    rdi,[rip+0x9f]               488D3D9F000000
mov    eax,0x0                      B800000000
call   520 <printf@plt>             E8C1FEFFFF
mov    eax,0x0                      B800000000
pop    rbp                          5D
ret                                 C3
```

Процессорная архитектура

Процессорная архитектура

- В понятие процессорной архитектуры с точки зрения обратной разработки можно включить следующие понятия:
  - Набор регистров
  - Битность
  - Набор инструкций
  - Порядок байтов

Понятие регистра

- Регистр – самая быстрая память процессора, служит для проведения большинства операций над данными (например для сложения чисел)
- Обычно регистров несколько (порядка 16-32)
- Регистры делятся на регистры общего назначения и специальные
  - В регистры общего назначения можно спокойно класть и обрабатывать любые данные
  - У регистров специального назначения обычно есть какой-то особенный смысл
  - Хороший пример регистра специального назначения – Instruction Pointer (счетчик команд), он указывает на текущую исполняемую инструкцию в программе
  - Еще один типичный особый регистр – Flags, хранящий, например, флаг сравнения результата последней операции с нулем или целочисленного переполнения
- Регистры могут быть разных типов – целых и дробных
  - В большинстве случаев для понимания логики программы достаточно только целых

Понятие регистра

Код программы:

```
0: r0 = 10
1: r1 = 20
2: r2 = r0 + r1
3: r2 = r2 * 2
4: r2 = 0
```

Регистры:

r0	r1	r2	rip
10	20	60	4

Исполняемая в данный момент строка (Instruction Pointer)

Битность

- Разрядность машинного слова процессора в битах
- Размер машинного слова обычно совпадает с:
  - Размерами целочисленных регистров
  - Размером указателя (то есть диапазоном адресуемой памяти)
- Большинство современных процессоров – 64-битные или 32-битные

Набор инструкций

- Очень важная для обратной разработки характеристика процессорной архитектуры
  - Для понимания того, что происходит в программе, вам необходимо иметь представление хотя бы о части используемых инструкций
- Набор инструкций – совокупность всех команд, которые вы можете использовать при написании программ на языке ассемблера
- Именно набор инструкций определяет, сможете ли вы, например, выполнить на данном процессоре в одну команду операцию умножения или вам придется изобретать его самостоятельно

Базовый набор инструкций для многих архитектур

- Операции работы с регистрами и памятью:
  - Загрузка константы в регистр
  - Чтение и запись из памяти в регистр
- Арифметические операции:
  - Сложение, вычитание, умножение, деление регистров
  - Побитовые операции (AND, OR, XOR) между регистрами
  - Сравнение регистров
- Управление исполнением
  - Условный или безусловный переход по адресу в программе
  - Вызов функции / возврат

Базовый набор инструкций для многих архитектур

Операция	Пример на ассемблере*	Альтернатива на Си
Загрузка константы в регистр	mov r0, #1337	r0 = 1337;
Чтение из памяти	ldr r0, [r1]	r0 = *r1;
Запись в память	str r0, [r1]	*r1 = r0;
Сложение	add r0, r1, r2	r0 = r1 + r2;
Побитовое AND	and r0, r1, r2	r0 = r1 & r2;
Сравнение	cmp r0, r1	zf = (r0 == r1);
Безусловный переход	jmp r0	goto *r0;
Условный переход	je r0	if (f) goto *r0;
Вызов функции	call r0	(*r0)();
Возврат из функции	ret	return;

* в точности такой архитектуры на самом деле не существует, я немного упростил команды

CISC и RISC архитектуры

- CISC – complex instruction set computer
- Доступно большое множество различных инструкций, у инструкций больше вариантов использования
  - Можно осуществлять арифметические операции напрямую с данными из памяти, без переноса в регистры
  - Инструкции рассчитаны на написание кода вручную: могут присутствовать избыточные инструкции, например полный набор из сложения, вычитания, и смены знака числа
- Зачастую размер инструкции является переменным
  - Длина инструкции определяется из ее вида

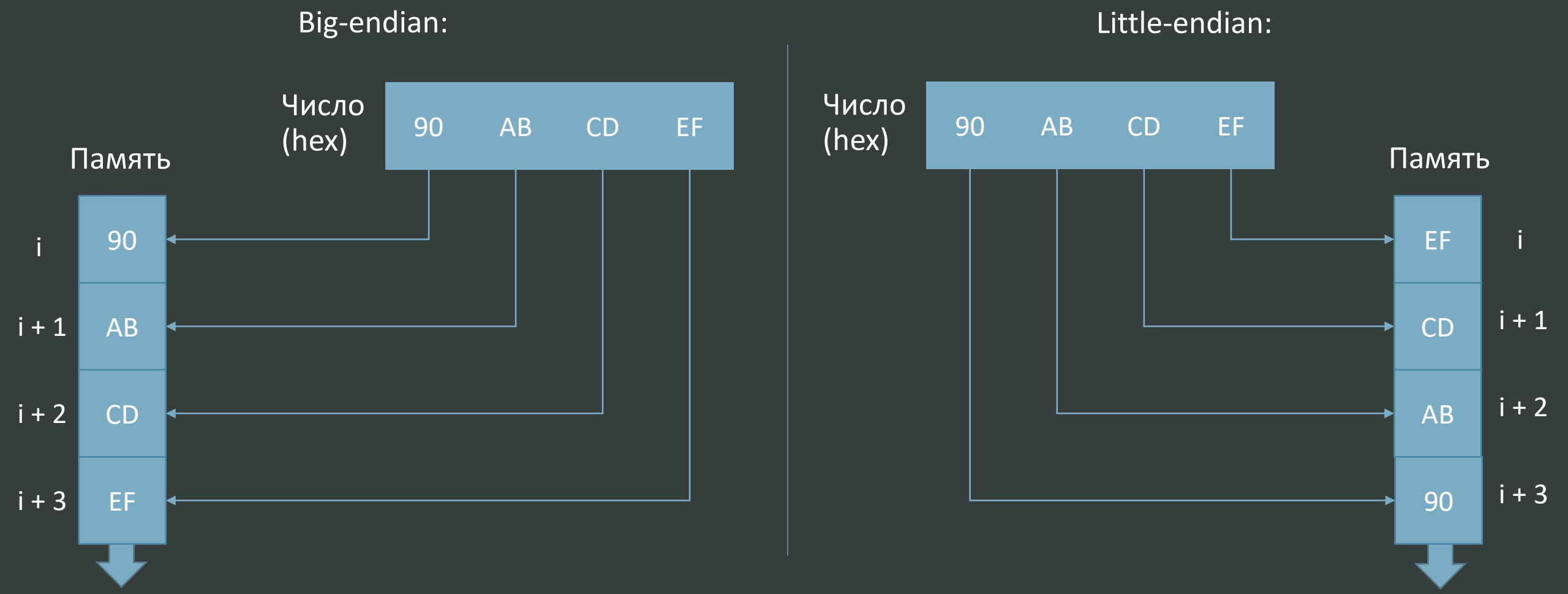
CISC и RISC архитектуры

- RISC – reduced instruction set computer
- Меньшее количество инструкций
  - Арифметические операции осуществляются только между регистрами, данные из памяти загружаются отдельной операцией
  - Инструкции рассчитаны на генерацию кода компиляторами
- Размер инструкции обычно постоянный
  - Это позволяет удобно адресовать инструкции без разбора всей программы

Порядок байтов

- Казалось бы, старший байт в хранимых в памяти числах, как и старшая цифра, должен идти вначале
  - В конце концов, в обычной математике 1002 < 2001
- У вычислительных систем свое мнение на этот счет
- Порядок байтов (endianness) обычно бывает двух видов:
  - Big-endian (от старшего к младшему) – обычный порядок, как в математике
  - Little-endian (от младшего к старшему) – обратный порядок
- Порядок байтов – тоже свойство процессорной архитектуры
  - Некоторые архитектуры могут работать в обоих режимах
- Обратный порядок позволяет удобно приводить типы (особенно к более коротким типам), а также, например, складывать числа, работая с байтами по порядку

Порядок байтов



Порядок байтов

- Это самый простой способ наступить на грабли при попытке перевести байты памяти в числа (или числа в байты, например в строки)
- В little-endian число 0x41424344 будет записано в памяти в виде строки DCBA
- В big-endian число 0x41424344 будет записано в памяти в виде строки ABCD

Популярные архитектуры

Популярные архитектуры

- Наиболее популярная архитектура на ПК и ноутбуках – x86 / x86-64
  - Именно к этой архитектуре принадлежат процессоры, производимые Intel и AMD
  - CISC архитектура, переменный размер инструкции
  - 8 / 16 регистров общего назначения
  - Little-endian порядок байтов
  - Около 1000 различных инструкций (в случае x86-64)
- Именно с этой архитектурой мы будем в основном иметь дело

Популярные архитектуры

- Наиболее популярная архитектура в мобильных устройствах – ARM / Aarch64
  - К этой архитектуре принадлежат процессоры Qualcomm, MediaTek, Samsung, Huawei и Apple
  - RISC архитектура, постоянный размер инструкции
  - 15 / 31 регистров общего назначения
  - Переменный порядок байтов, но в подавляющем большинстве случаев little-endian
- Эта архитектура тоже может оказаться полезной в мобильной разработке

Ассемблер x86

Регистры x86-64

- У регистров x86-64 немного необычные названия:
  - RAX, RBX, и т.д., казалось бы, почему бы не назвать их R0-R15 как у ARM
- Дело в том, что эти регистры появились очень давно, в 16-битном варианте архитектуры в процессорах 8086 – 80286
  - Тогда их звали AX, BX, CX, DX, SP, BP, SI, DI и они были 16-битными
  - К тому же AX-DX можно было использовать по 8-битным половинкам AL/AH - DL/DH
- Затем, когда архитектура стала 32-битной, их расширили до 32-битных, дописав букву E (Extended):
  - EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI
- В конечном итоге их расширили до 64-битных, дописав букву R (Register) вместо E и добавили еще 8 регистров с нормальными названиями (как у ARM):
  - RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8-R15

Регистры x86-64

- Естественно, добрая традиция использования половин регистров никуда не делась:

- Таким образом, для RAX = 0x1234567890ABCDEF:
  - EAX = 0x90ABCDEF
  - AX = 0xCDEF
  - AH = 0xCD
  - AL = 0xEF

Сюрприз

- Если вы решите записать что-нибудь в EAX, то вся остальная часть RAX тоже занулится (остальных регистров это тоже касается)
  - Даже если вы решили просто что-нибудь прибавить к EAX
- Однако, если вы решите записать в AX или AL / AH, то этого не произойдет
- Такое поведение было введено в x86-64 (предположительно) с целью ускорения выполнения операций над 32-битными числами: отбрасывание старшей половины 64-битных регистров позволяет не вычислять / вспоминать, что там было раньше
  - Физические регистры могут вполне не соответствовать логическим, более того половинки регистров тоже могут быть выделены в отдельные физические регистры, что потребует при обращении к полному регистру операции их синхронизации (т.н. "partial register stall")

Инструкции

- Вообще в x86 очень много инструкций, мы пока рассмотрим только основные
  - Причем тоже не все, некоторые будут оставлены на следующие лекции
- Сегодня мы рассмотрим следующие инструкции:
  - MOV
  - ADD, SUB, OR, XOR, AND
  - INC, DEC, NOT
  - MUL, DIV, IMUL, IDIV
  - SHL, SHR, ROL, ROR
  - XCHG
  - CMP, TEST
  - JMP, JE, JNE
  - CALL, RET

Инструкция MOV

- Служит для загрузки из памяти и в память и перемещения регистров
- Для записи чего-нибудь в регистр, пишем следующий код:

```
MOV RAX, <что-нибудь>
```

- В роли чего-нибудь вполне может выступить константа, другой регистр или значение по адресу в памяти:

Код ассемблера	Альтернатива на Си
MOV RAX, 123	RAX = 123;
MOV RAX, RBX	RAX = RBX;
MOV RAX, [RBX]	RAX = *((long long*)RBX);

Инструкция MOV

- Впрочем, можно записать и наоборот из регистра в память:

```
MOV [RAX], <что-нибудь>
```

- В этот раз в роли чего-нибудь может выступить регистр или константа:

Код ассемблера	Альтернатива на Си
MOV [RAX], RBX	*((long long*)RAX) = RBX;
MOV [RAX], BL	*((char*)RAX) = BL;
MOV [RAX], 123	*((long long*)RAX) = 123;

- Впрочем, тут может возникнуть проблема: что если вам нужно записать только один байт, а не сразу 8?
  - В случае с регистрами вы можете просто указать часть регистра (например BL), но что делать с константами?

Инструкция MOV

- Здесь на помощь приходит явное указание размера для записи: byte ptr, word ptr, dword ptr и qword ptr:

```
MOV byte ptr [RAX], 123
```

- Примеры:

Код ассемблера	Альтернатива на Си
MOV byte ptr [RAX], 123	*((char*)RAX) = 123;
MOV word ptr [RAX], 123	*((short*)RAX) = 123;
MOV dword ptr [RAX], 123	*((int*)RAX) = 123;
MOV qword ptr [RAX], 123	*((long long*)RAX) = 123;

Инструкции ADD, SUB, OR, XOR, AND

- Применяются для осуществления арифметических и побитовых операций:

Команда	Операция
ADD	Сложение
SUB	Вычитание
OR	Побитовое ИЛИ
XOR	Побитовое исключающее ИЛИ
AND	Побитовое И

- Операция выполняется с первым аргументом и записывается в него:

Код ассемблера	Альтернатива на Си
ADD RAX, RBX	RAX += RBX;

Инструкции ADD, SUB, OR, XOR, AND

- Правила использования соответствуют MOV (в том числе и фокусы с byte ptr и т.д.):

Код ассемблера	Альтернатива на Си
ADD RAX, 123	RAX += 123;
SUB RAX, RBX	RAX -= RBX;
OR RAX, [RBX]	RAX |= *((long long*)RBX);
XOR [RAX], RBX	*((long long*)RAX) ^= RBX;
AND [RAX], 123	*((long long*)RAX) &= 123;
ADD byte ptr [RAX], 123	*((char*)RAX) += 123;

Инструкции INC, DEC, NOT

- Эти инструкции принимают только один аргумент, так как являются унарными операциями:

Команда	Операция
INC	Увеличение на 1
DEC	Уменьшение на 1
NOT	Побитовое отрицание

- Операция выполняется над единственным аргументом и пишется в него же
- Можно также использовать регистры и адреса в памяти (и byte ptr и т.д.):

Код ассемблера	Альтернатива на Си
INC [RAX]	*((long long*)RAX)++;
NOT RAX	RAX = ~RAX;

Инструкции MUL, DIV

- Осуществляют беззнаковые умножение и деление соответственно
- На удивление принимают только один операнд (хотя казалось бы)
  - В роли первого операнда всегда выступает RAX или RDX:RAX
- RDX:RAX?!
  - Ага, у умножения двух 64-битных чисел 128-битный результат
  - Делить тут тоже можно 128-битные числа на 64-битные (но результат будет 64-битным, другой регистр будет использован для остатка от деления)
  - Старшая часть 128-битного числа лежит в RDX, младшая – в RAX
  - В случае если операнды 32-битные и меньше, результат умножения все равно будет лежать в двух регистрах (например паре EDX:EAX)
  - К счастью, у ARM такого нет
- Единственный аргумент может быть как у INC, DEC, NOT (регистр или адрес)

Инструкции MUL, DIV

Код ассемблера	Альтернатива на Си
MUL RCX	RDX = ((uint128_t) RAX * RCX) >> 64; RAX *= RCX;
DIV RCX	uint128_t Temp = (((uint128_t) RDX << 64) + RAX); RAX = Temp / RCX; RDX = Temp % RCX;

Инструкции IMUL, IDIV

- Осуществляют знаковые умножение и деление соответственно
- Наиболее примечательна инструкция IMUL – в отличие от MUL у нее есть не только варианты с одним аргументом, а и с двумя и даже тремя
- В этих вариантах IMUL не считает старшую половину числа (то есть умножение двух 64-битных чисел считается 64-битным, прямо как обычно в Си), поэтому знаковое умножение тут становится эквивалентным беззнаковому
  - Эффекты от знаковости заметны только в старшей половине результата, а ее как раз нет
  - Поэтому компиляторы предпочитают именно IMUL для беззнакового умножения
- Вариант с двумя аргументами позволяет перемножить регистр и регистр / память / константу и положить результат в регистр первого аргумента
- Вариант с тремя аргументами позволяет перемножить регистр / память и константу, положив результат в регистр результата

Инструкции IMUL, IDIV

Код ассемблера	Альтернатива на Си
IMUL RCX, RBX	RCX *= RBX;
IMUL RCX, RBX, 123	RCX = RBX * 123;
IDIV RCX	int128_t Temp = (((int128_t) RDX << 64) + RAX); RAX = Temp / RCX; RDX = Temp % RCX;

Инструкция XCHG

- Меняет местами два регистра или регистр и память:

Код ассемблера	Альтернатива на Си (xorswap)
XCHG RAX, RBX	RAX ^= RBX; RBX ^= RAX; RAX ^= RBX;

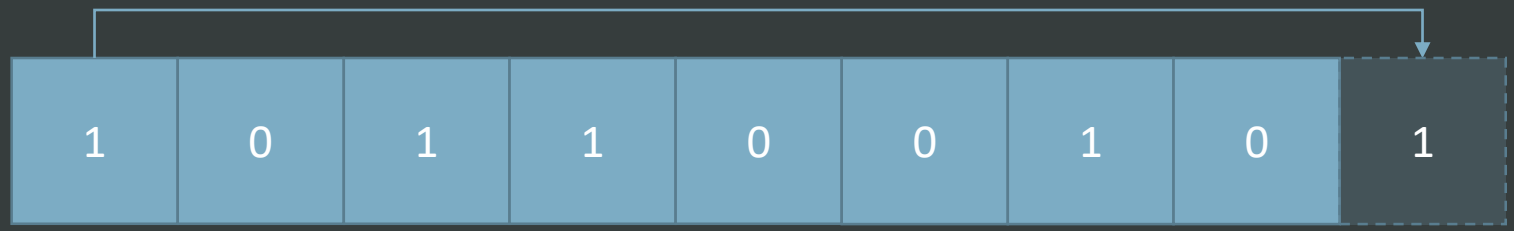
Инструкции SHL, SHR

- Осуществляют двоичные сдвиги влево и вправо соответственно
- Также у них есть и варианты SAL / SAR, осуществляющие арифметический сдвиг (для чисел со знаком), (SAL == SHL)
- Сдвиг может применяться к регистрам и памяти, размер сдвига может быть как константным, так и зависеть от регистра CL

Код ассемблера	Альтернатива на Си
SHL RAX	RAX <<= 1;
SHR RAX, 10	RAX >>= 10;
SHR RAX, CL	RAX >>= CL;

Инструкции ROL, ROR

- Также, однако, присутствует и другая операция сдвига, гораздо менее знакомая программистам на высокоуровневых языках — циклический сдвиг
- Циклический сдвиг очень похож на логический, однако он не теряет выдвинутые из регистра биты, а задвигает их с другой стороны
- На x86 циклический сдвиг реализуется инструкциями ROL и ROR
  - Их использование аналогично SHL и SHR
- Эти функции часто можно встретить в криптографии, потому что они не теряют информацию



Инструкции CMP, TEST

- Обе эти инструкции могут использоваться для сравнения
  - Правда, вторая в основном с нулем
- Инструкции являются полными аналогами SUB и AND, но не сохраняют результат
  - Основная их цель – установка флагов процессора, ранее мы их игнорировали
- В частности, для равных аргументов CMP мы получим значение флага ZF=1
  - Потому что при их вычитании получится ноль, а ZF – zero flag
- TEST обычно используется с одним и тем же аргументом дважды
  - Например TEST EAX, EAX
  - В результате получим ZF=1 если EAX=0

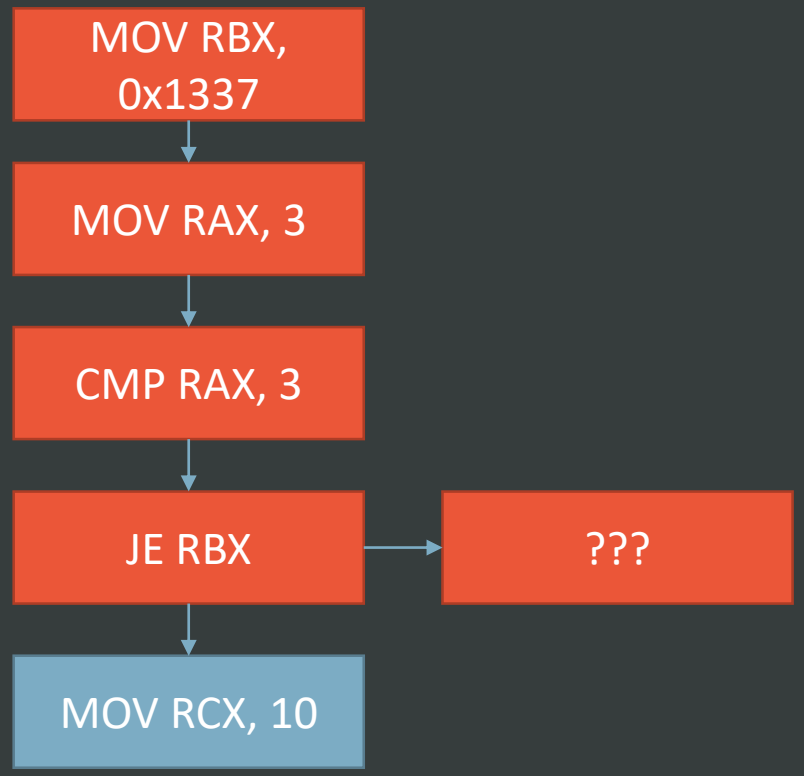
Инструкция JMP

- Эта инструкция используется для перехода на другую часть программы
  - Это один из немногих способов влиять на регистр RIP (Instruction Pointer) на x86
- В качестве аргумента могут выступать:
  - Абсолютные числа, тогда прыжок будет относительным (считается относительно следующей инструкции)
  - Регистры и значения памяти по регистрам, тогда прыжок будет абсолютным

Код ассемблера	Альтернатива на Си
JMP 10	RIP += 10 + sizeof(jmp_instr);
JMP RAX	RIP = RAX;
JMP [RAX]	RIP = *((long long*)RAX);

Инструкции JE, JNE

- Эти инструкции – то же самое, что и JMP, но зависят от значения флага ZF
- Также у этих инструкций есть вторые имена: JZ и JNZ
- В случае выполнения условия, осуществляется прыжок по адресу, в противном случае – переход к следующей инструкции
  - Для JE таким условием является ZF = 1
  - Для JNE таким условием является ZF = 0



Метки

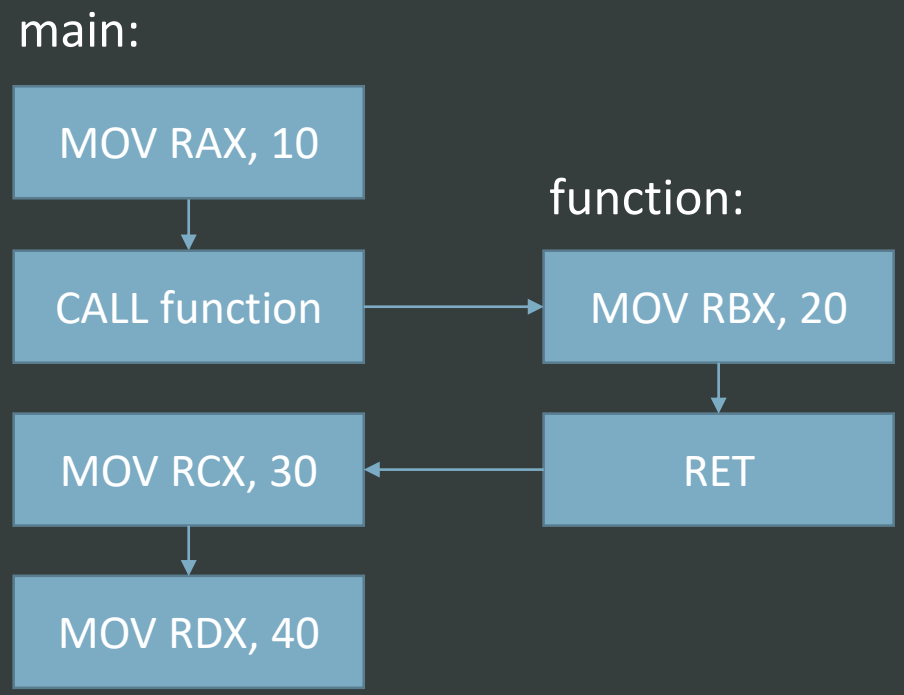
- Для того чтобы облегчить указание места, куда прыгнуть, ассемблер предлагает такую сущность как метки
  - Метки не выполняются процессором, это абстракция времени компиляции
- Метки могут использоваться по имени, а создаются в любом месте программы указанием имени с двоеточием на конце:
- Пример:

```
        jmp label
        mov eax, 10
label:
        mov ebx, 30
```

- В этом примере строка "mov eax, 10" будет перепрыгнута
  - А "label" будет являться меткой

Инструкции CALL и RET

- CALL – то же самое, что и JMP, только сохраняет информацию об адресе возврата
  - Используется при создании функций
  - Варианты, зависящие от флагов (как у JMP / JE / JNE) отсутствуют
- Сохранение адреса возврата позволяет вернуться к следующей инструкции командой RET
- Аргументы функции можно передавать разными способами, на Linux x86-64 это принято делать через регистры (RDI, RSI, RDX, RCX и т.д.)
- Куда именно сохраняется адрес возврата будет рассказано в следующей лекции



Некоторые директивы GNU Assembler

- .byte, .short, .int, .quad, .octa – позволяют прямо в текущем месте программы записать число (или несколько, через запятую), принимаемое в качестве аргумента
  - Так можно создавать массивы с данными или же писать прямо в машинных кодах
- .(l)comm имя, длина, [выравнивание] – позволяет создать массив определенной длины с опциональным выравниванием адреса
- .ascii, .asciz – позволяет создать в текущем месте программы ASCII строку, asciz еще и добавит на конце нулевой символ (как в Си, очень удобно)
- .text, .data, .bss, .section имя – позволяет переключиться в определенную секцию (кода, данных и т.д.), что это такое мы узнаем в следующих лекциях
- offset – позволяет получить адрес метки или символа в программе, часто используется для того чтобы положить соответствующий адрес в регистр

Время задач

Simple CrackMe

Категория: Lesson 14 / Assembly basics

Решивших: 0

Время: 00:00:02

- Доступ к задачам можно получить как обычно на nsuctf.ru

Очередные грабли

- У x86-ассемблера существует несколько синтаксисов
- По умолчанию у GNU Assembler (и GCC) на x86 используется синтаксис AT&T
- Во всех примерах и задачах этой лекции, однако, я использовал синтаксис Intel
  - Также он используется дизассемблером IDA, о котором мы позже узнаем
  - Считается, что синтаксис Intel более популярен в Windows-среде, а AT&T – в Linux-среде
- Самым фундаментальным отличием синтаксисов является порядок операндов:
  - Чтобы поместить в Intel-синтаксисе 0 в регистр EAX, можно использовать mov eax, 0
  - В AT&T-синтаксисе, однако, это будет выглядеть как mov $0x0,%eax
- Включить синтаксис Intel в GNU AS можно директивой .intel_syntax noprefix

Очередные грабли

x86 (AT&T)

```
push   %rbp
mov    %rsp,%rbp
lea    0x9f(%rip),%rdi
mov    $0x0,%eax
callq  520 <printf@plt>
mov    $0x0,%eax
pop    %rbp
retq
```

x86 (Intel)

```
push   rbp
mov    rbp,rsp
lea    rdi,[rip+0x9f]
mov    eax,0x0
call   520 <printf@plt>
mov    eax,0x0
pop    rbp
ret
```

ARM

```
push    {fp, lr}
add     fp, sp, #4
ldr     r0, [pc, #12]
bl      102e0 <printf@plt>
mov     r3, #0
mov     r0, r3
pop     {fp, pc}
```

Полезные программы

- Для сборки программ, в том числе и на языке ассемблера, можно использовать GCC – компилятор по умолчанию в большинстве дистрибутивов Linux
  - Или, если у вас есть дух приключений, напрямую ассемблер GNU AS
- Для того чтобы преобразовать ваш код на языке Си в ассемблер, можно использовать флаг -S этого компилятора:
  - gcc main.c -S создаст файл main.s с листингом на языке ассемблера
- Также для просмотра ассемблерного кода, генерируемого различными компиляторами, можно использовать сайт Compiler Explorer (https://godbolt.org/)

Спасибо за внимание!
Задачи доступны на

nsuctf.ru

- Пожалуйста, используйте имя пользователя формата "Фамилия Имя"
  - e-mail можно забить любой, сервером он не проверяется
- Для вопросов по задачам рекомендую присоединиться к @NSUCTF в Telegram
  - Только, пожалуйста, без спойлеров