

Машинный уровень I: Основы

Основы информатики

Компьютерные основы программирования

u.to/DbCmFA

На основе CMU 15-213/18-243:

Introduction to Computer Systems

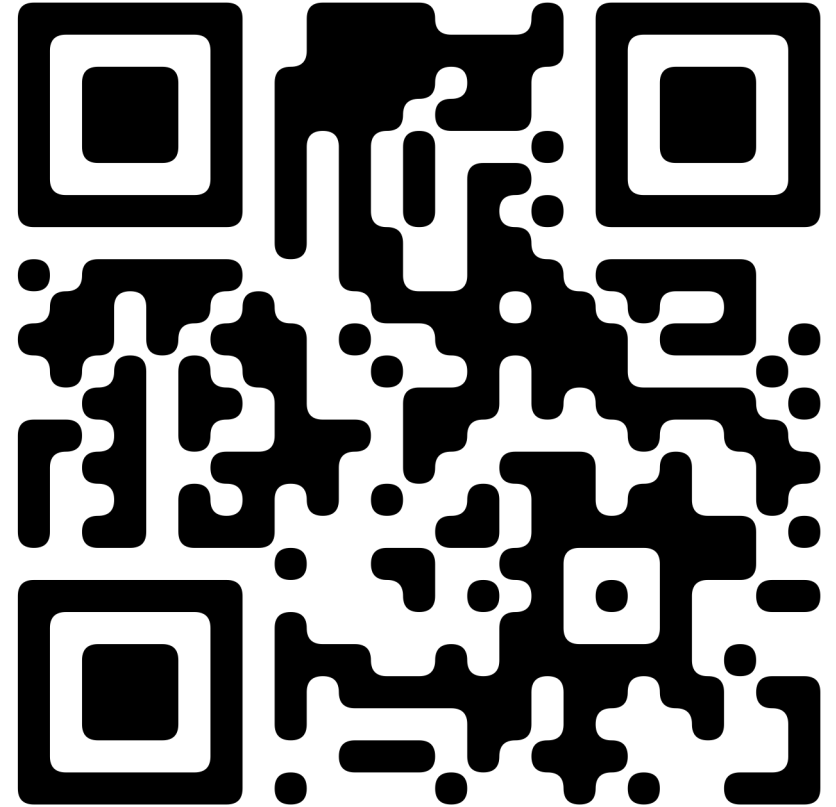
u.to/XoKmFA

Лекция 4, 26 февраля, 2024

Лектор:

Дмитрий Северов, кафедра информатики 608 КПП

cs.mipt.ru/wp/?page_id=346



Машинный уровень 1: Основы

- Краткая история изделий Интел
- Си, ассемблер, машинный код
- Основы ассемблера: регистры, операнды, пересылки
- Арифметические и логические операции

Процессоры Intel x86

- Подавляющее доминирование на рынках настольных ПК, ноутбуков, серверов
- Эволюционное конструирование
 - Обратно совместимы до 8086, выпущенного в 1978
 - Функции добавляются с ходом времени
- **Complex instruction set computer (CISC)**
 - Много команд во многих форматах
 - Но, лишь немногие встречаются в Linux программах
 - Трудно конкурировать по быстродействию с Reduced Instruction Set Computers (RISC)
 - Однако, Intel сделал именно это!
 - В терминах быстродействия. Но не энергопотребления.

Вехи эволюции Intel x86

<i>Имя</i>	<i>Дата</i>	<i>Транзисторов</i>	<i>МГц</i>
■ 8086	1978	29K	5-10
<ul style="list-style-type: none">■ 1-й 16-битный процессор. Основа для IBM PC & DOS■ 1MB адресного пространства			
■ 386	1985	275K	16-33
<ul style="list-style-type: none">■ 1-й 32-битный процессор, известен как IA32■ Добавлена “плоская адресация”, способен исполнять Unix			
■ Pentium 4F	2004	125M	2800-3800
<ul style="list-style-type: none">■ 1-й 64-битный процессор, известен как x86-64			
■ Core 2	2006	291M	1060-3500
<ul style="list-style-type: none">■ First multi-core Intel processor			
■ Core i7	2008	731M	2667-3333
<ul style="list-style-type: none">■ Почти современные машины			

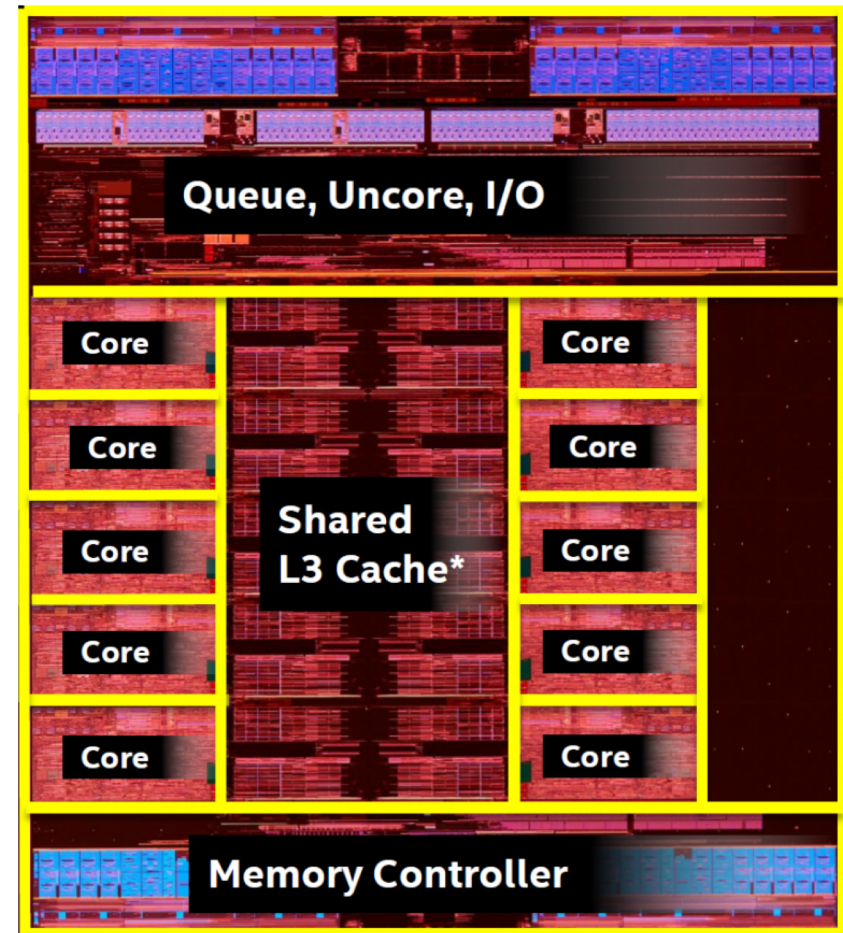
Процессоры Intel x86, далее

■ Эволюция машин

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M
■ Core i7	2016	2.6-3.6B

■ Добавленные возможности

- Команды обработки данных мультимедиа(multimedia), криптографии
- Команды для более эффективной условной обработки
- Переход от 32-битного слова к 64-битному
- Больше ядер, больше памяти



Состояние 2015

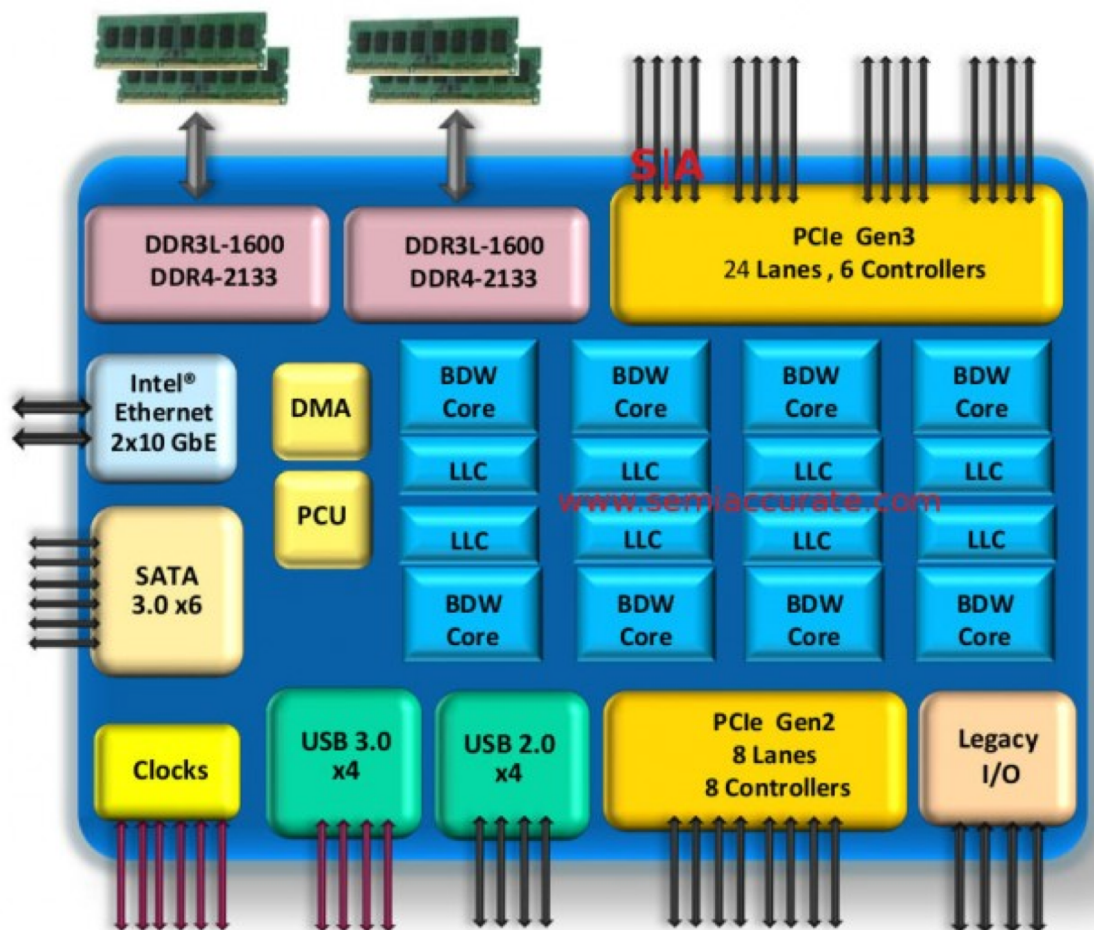
- Core i7 Broadwell 2015

■ Настольная модель

- 4 ядра
- Встроенная графика
- 3.3-3.8 ГГц
- 65Вт

■ Серверная модель

- 8 ядер
- Встроенный ввод/вывод
- 2-2.6 ГГц
- 45Вт



<https://en.wikichip.org/wiki/intel/microarchitectures>

Клон x86: Advanced Micro Devices (AMD)

■ Сначала

- AMD догоняла Intel
- Немного медленнее, много дешевле

■ Потом

- Привлекла лучших аппаратчиков из Digital Equipment Corp. И других компаний
- Создала Opteron: сильного соперника Pentium 4
- Разработала x86-64, собственное 64-битное расширение

■ Последние годы

- Intel вертикально интегрированная компания
 - Один из лидеров полупроводниковых технологий
- AMD опирается на сторонних изготовителей ИС
 - Существенно отстаёт от Intel в целом, иногда обгоняя, в т.ч. сейчас

64-битный Intel

- **2001: Intel попытался радикально сменить IA32 на IA64**
 - Полностью другая архитектура (Itanium)
 - Исполнение кода IA32, только как устаревшего
 - Разочаровывающее быстрое действие
- **2003: AMD вышла с эволюционным решением**
 - x86-64 (сейчас известна как “AMD64”)
- **Intel нарушил обязательство сосредоточения на IA64**
 - Трудно признавать ошибки и превосходство AMD
- **2004: Intel анонсировал EM64T расширение к IA32**
 - 64-битная технология расширенной памяти
 - Почти идентична x86-64!
- **Сейчас все “low-end” процессоры x86 содержат x86-64**
 - Однако много кода всё ещё в 32-битном режиме

Мы затронем

- **IA32**

- Традиционная x86

- **x86-64/EM64T**

- Современный стандарт

- **Разделы в книге**

- 3.1—3.12 об IA32,
 - 3.13 об x86-64

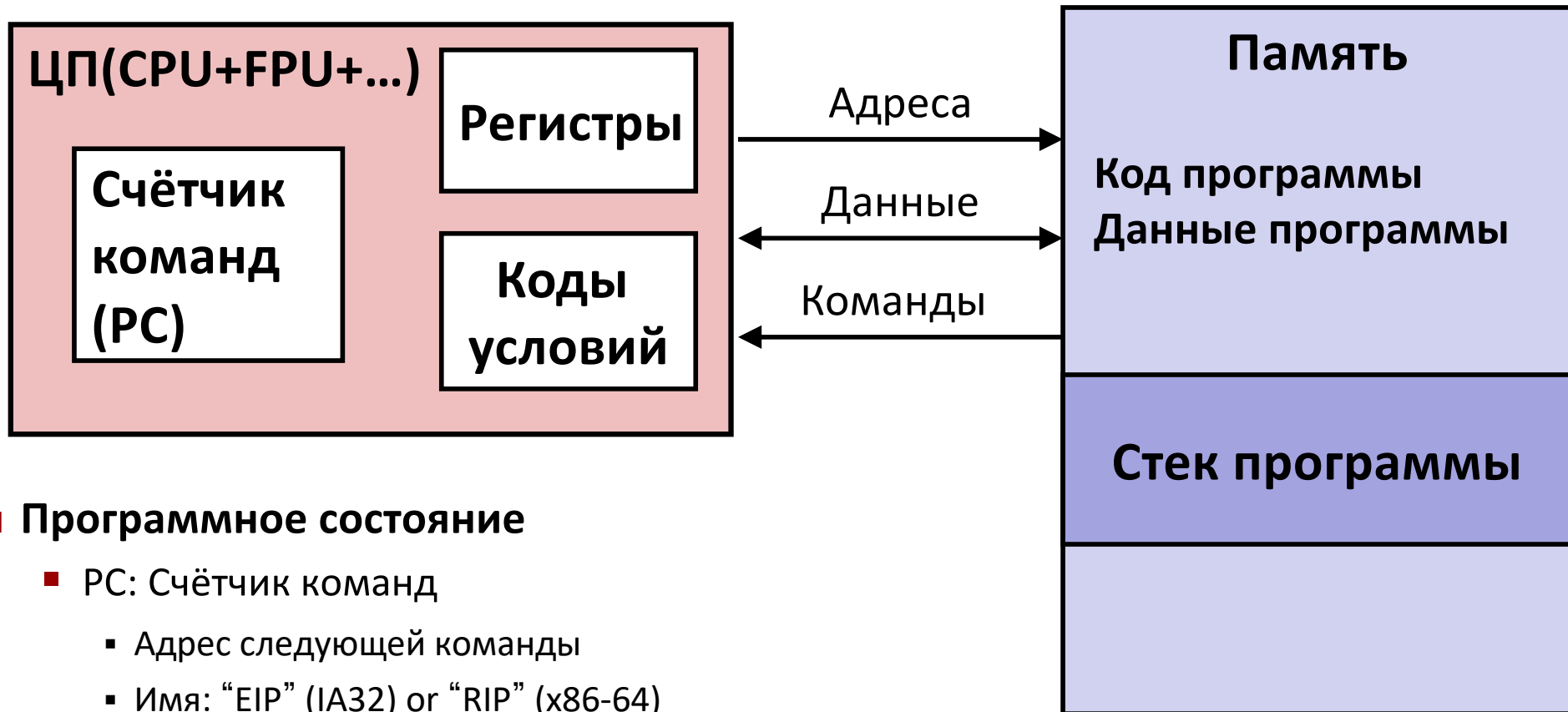
Машинный уровень 1: Основы

- Краткая история изделий Интел
- Си, ассемблер, машинный код
- Основы ассемблера: регистры, операнды, пересылки
- Арифметические и логические операции

Определения

- **Архитектура(системы команд),** также instruction set architecture: ISA. Часть конструкции процессора которую надо знать для понимания ассемблерного кода.
 - Примеры: описание набора команд, регистры.
- **Микроархитектура:** Реализация архитектуры.
 - Примеры: размер кеша и частота ядра.
- **Формы кода:**
 - **Машинный код:** содержимое байтов, исполняемое процессором
 - **Ассемблерный код:** текстовое представление машинного кода
- **Примеры ISA (Intel):** x86, IA, IPF

Программная модель ассемблера



■ Программное состояние

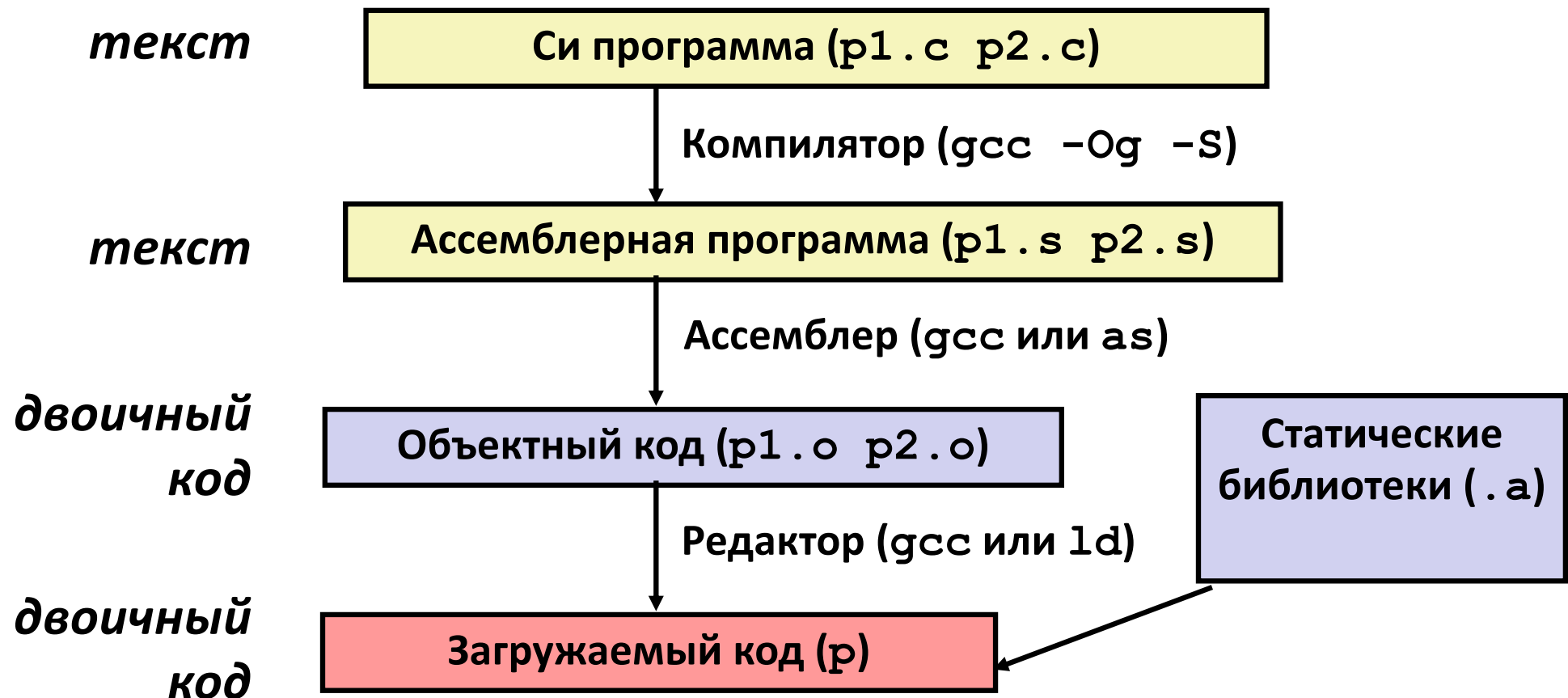
- PC: Счётчик команд
 - Адрес следующей команды
 - Имя: "EIP" (IA32) or "RIP" (x86-64)
- Набор регистров
 - Наиболее используемые данные
- Коды условий
 - Хранят информацию о состоянии самой последней арифм. команды
 - Используются для условных переходов

■ Память

- Массив адресуемых байт
- Код, данные пользователя
- Включая стек для поддержки процедур

Трансляция Си кода в объектный

- Код в файлах `p1.c` `p2.c`
- Компилируем командой: `gcc -Og p1.c p2.c -o p`
 - Используем минимальную оптимизацию (`-Og`)
 - Помещаем результирующий двоичный код в файл `p`



Компиляция в ассемблер

Си код (sum.c)

```
long plus(long x, long y);  
  
void sumstore(long x, long y,  
              long *dest)  
{  
    long t = plus(x, y);  
    *dest = t;  
}
```

Сгенерированный код ассемблера x86-64

```
sumstore:  
    pushq    %rbx  
    movq     %rdx, %rbx  
    call     plus  
    movq     %rax, (%rbx)  
    popq     %rbx  
    ret
```

Получается командой

```
gcc -Og -S sum.c
```

Создаёт файл `sum.s`

Внимание: даёт сильно различные результаты в различных окружениях из-за различных версий gcc и различных параметров компилятора.

“Типы данных” ассемблера

- **1-, 2-, 4- или 8-байтные “целочисленные” данные**
 - Значения данных
 - Адреса (нетипизированные указатели)
- **4-, 8-, или 10- байтные данные с плавающей точкой**
- **Код: последовательности байт, кодирующие последовательность команд**
- **Никаких сложных типов как массивы или структуры**
 - Просто подряд расположенные в памяти байты

Операции ассемблера

- **Арифметические операции с данными в регистрах или в памяти**
- **Передача данных между памятью и регистрами**
 - Загружают данные из памяти в регистры
 - Выгружают данные из регистров в память
- **Передача управления**
 - Безусловные переходы
 - Условные переходы
 - Вызов процедур и возврат из них

Объектный код

Код `sum`

`0x0400595:`

`0x53`

`0x48`

`0x89`

`0xd3`

`0xe8`

`0xf2`

`0xff`

`0xff`

`0xff`

`0x48`

`0x89`

`0x03`

`0x5b`

`0xc3`

- **Всего 14 байт**
- **Каждая команда 1, 2 или 3 байта**
- **Старт по адресу `0x0400595`**

■ Ассемблер

- Транслирует `.S` в `.o`
- Кодировывает команды двоичным кодом
- Почти готовый к загрузке код
- Без кода других файлов, со ссылками на него

■ Редактор (связей)

- Использует ссылки между файлами кода
- Компонуется со статическими библиотеками
 - Например, код для **`malloc`**, **`printf`**
- Некоторые библиотеки *связываются динамически*
 - Связывание происходит при загрузке, исполнении

Пример машинных команд

```
*dest = t;
```

```
movq %rax, (%rbx)
```

```
0x40059e:  48 89 03
```

■ Си-код

- Сохранить значение **t** в память обозначенную **dest**

■ Ассемблер

- Переместить 8-байтное значение в память
 - Четверное слово в терминах x86-64
- Операнды:
 - t:** регистр **%rax**
 - dest:** регистр **%rbx**
 - *dest:** память **M[%rbx]**

■ Объектный код

- 3-байтная команда
- по адресу **0x40059e**

Дизассемблирование объектного кода

Дизассемблированный код

```
0000000000400595 <sumstore>:  
400595: 53                push    %rbx  
400596: 48 89 d3          mov     %rdx,%rbx  
400599: e8 f2 ff ff ff    callq   400590 <plus>  
40059e: 48 89 03          mov     %rax, (%rbx)  
4005a1: 5b                pop     %rbx  
4005a2: c3                retq
```

■ Дизассемблер

`objdump -d sum`

- Полезный инструмент для анализа объектного кода
- Анализирует битовые последовательности наборов команд
- Приблизительно воссоздаёт ассемблерный код
- Может обрабатывать файлы `a.out` (загрузочные) или `.o`

Вариант дизассемблирования

Объектный код

Дизассемблированный код

0x0400595:

0x53

0x48

0x89

0xd3

0xe8

0xf2

0xff

0xff

0xff

0x48

0x89

0x03

0x5b

0xc3

Dump of assembler code for function sumstore:

0x0000000000400595 <+0>: push %rbx

0x0000000000400596 <+1>: mov %rdx,%rbx

0x0000000000400599 <+4>: callq 0x400590 <plus>

0x000000000040059e <+9>: mov %rax, (%rbx)

0x00000000004005a1 <+12>: pop %rbx

0x00000000004005a2 <+13>: retq

■ С помощью отладчика gdb

`gdb sum`

`disassemble sumstore`

■ Дизассемблируемая процедура

`x/14xb sum`

■ Разобрать 14 байт, начиная с `sumstore`

Что удастся дизассемблировать?

```
% objdump -d WINWORD.EXE
```

```
WINWORD.EXE:      file format pei-i386
```

```
No symbols in "WINWORD.EXE".
```

```
Disassembly of section .text:
```

```
30001000 <.text>:
```

```
30001000:
```

```
30001001:
```

```
30001003:
```

```
30001005:
```

```
3000100a:
```

Обратная разработка нарушает
лицензионное соглашение
Microsoft

- Всё, что имеет смысл исполняемых команд
- Дизассемблер разбирает указанные байты как машинный код и представляет как ассемблерный код

Машинный уровень 1: Основы

- Краткая история изделий Интел
- Си, ассемблер, машинный код
- Основы ассемблера: регистры, операнды, пересылки
- Арифметические и логические операции

Целочисленные регистры x86-64

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

- В сравнении с IA32 расширены существующие регистры. Добавлены 8 новых.
- **%ebp/%rbp** сделан регистром общего назначения, в отличие от IA32

Почти история: регистры IA32

Общего назначения	%eax	%ax	%ah	%al	Мнемоника (устаревшая) <i>Accumulate</i> аккумулятор
	%ecx	%cx	%ch	%cl	<i>Counter</i> счётчик
	%edx	%dx	%dh	%dl	<i>Data</i> данные
	%ebx	%bx	%bh	%bl	<i>Base</i> База
	%esi	%si			<i>Source index</i> Индекс источника
	%edi	%di			<i>Destination index</i> индекс назначения
	%esp	%sp			<i>Stack pointer</i> указатель стека
	%ebp	%bp			<i>Base pointer</i> указатель базы
16-битные поименованные части регистров (для обратной совместимости)					

Пересылки данных

■ Перенос данных

`movq` *Источник, Результат:*

■ Типы операндов

- **Непосредственные:** константные, в командах
 - Пример: `$0x400`, `$-533`
 - Как Си константы, но с префиксом ‘\$’
 - Кодировются 1, 2, 4 или 8 байтами
- **Регистровые:** только 16 (целочисленных)
 - Пример: `%rax`, `%r13`
 - `%rsp` зарезервирован
 - Другие особо используются некоторыми командами
- **В памяти:** 8 последовательных байт памяти по адресу из регистра
 - Простейший пример: `(%rax)`
 - Несколько различных “вариантов адресации”

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rN`

movq Комбинации операндов

	Откуда	Куда	Src, Dest	Си-аналог
movq	Imm	Reg	movq \$0x4, %rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Mem	Reg	movq (%rax), %rdx	temp = *p;

Нельзя передать из памяти в память одной командой

Простые адресации памяти

■ Базовая (R) Mem[Reg[R]]

- Регистр R содержит адрес памяти
- Ага! Раскрытие указателя в Си

```
movq (%rcx) , %rax
```

■ Со смещением D(R) Mem[Reg[R]+D]

- Регистр R содержит адрес начала фрагмента памяти
- Константа D обозначает сдвиг от начала фрагмента

```
movq 8(%rbp) , %rdx
```

Пример использования простых адресаций

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

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

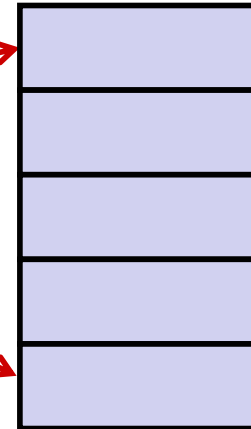

Разбираем swap()

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

Регистры

%rdi	
%rsi	
%rax	
%rdx	

Память



Регистр	Значение
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Разбираем swap()

Регистры

%rdi	0x120
%rsi	0x100
%rax	
%rdx	

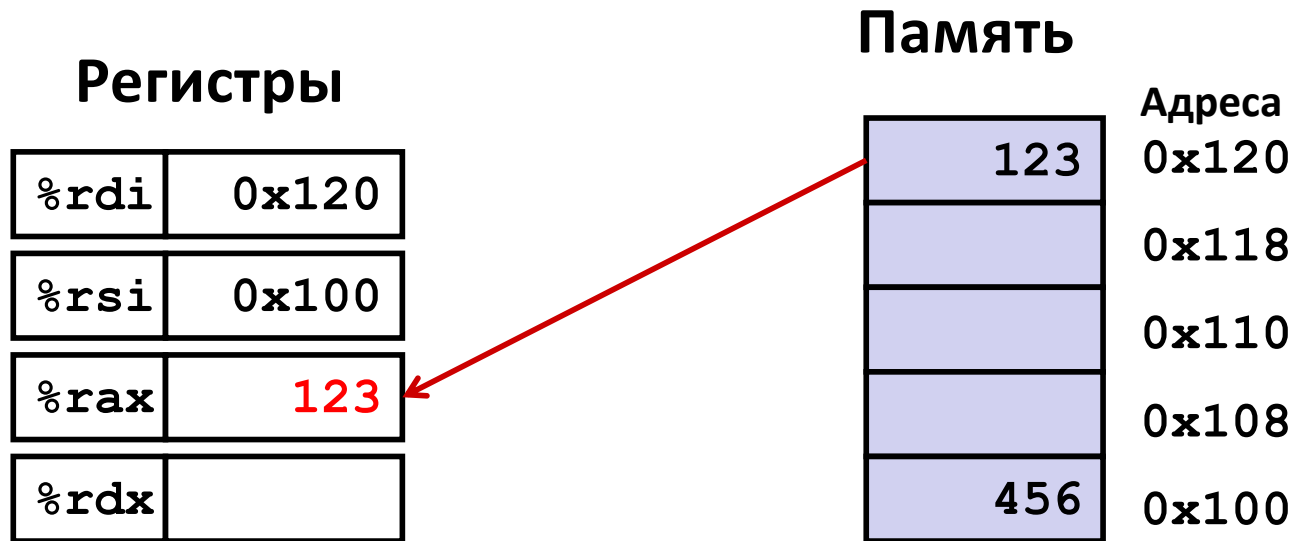
Память

Адреса	
123	0x120
	0x118
	0x110
	0x108
456	0x100

swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

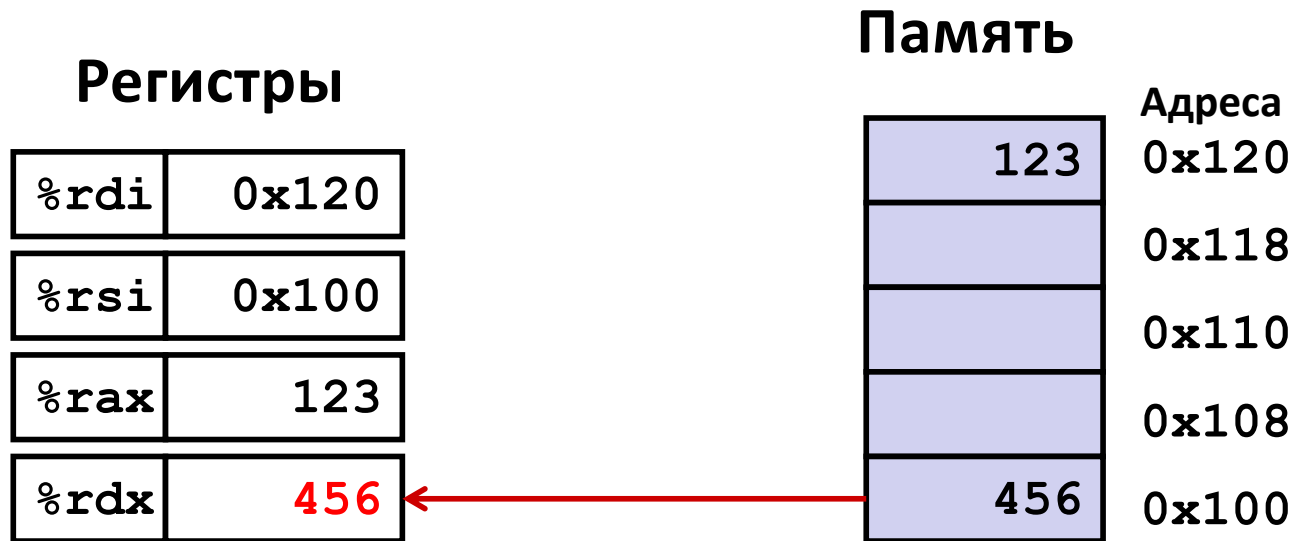
Разбираем swap()



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

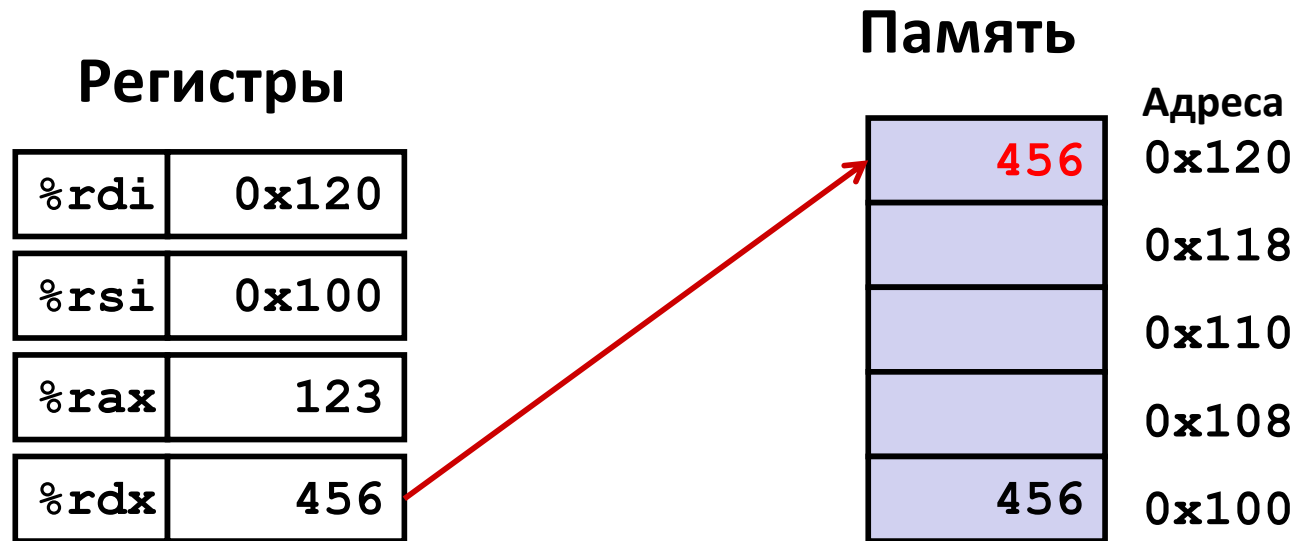
Разбираем swap()



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

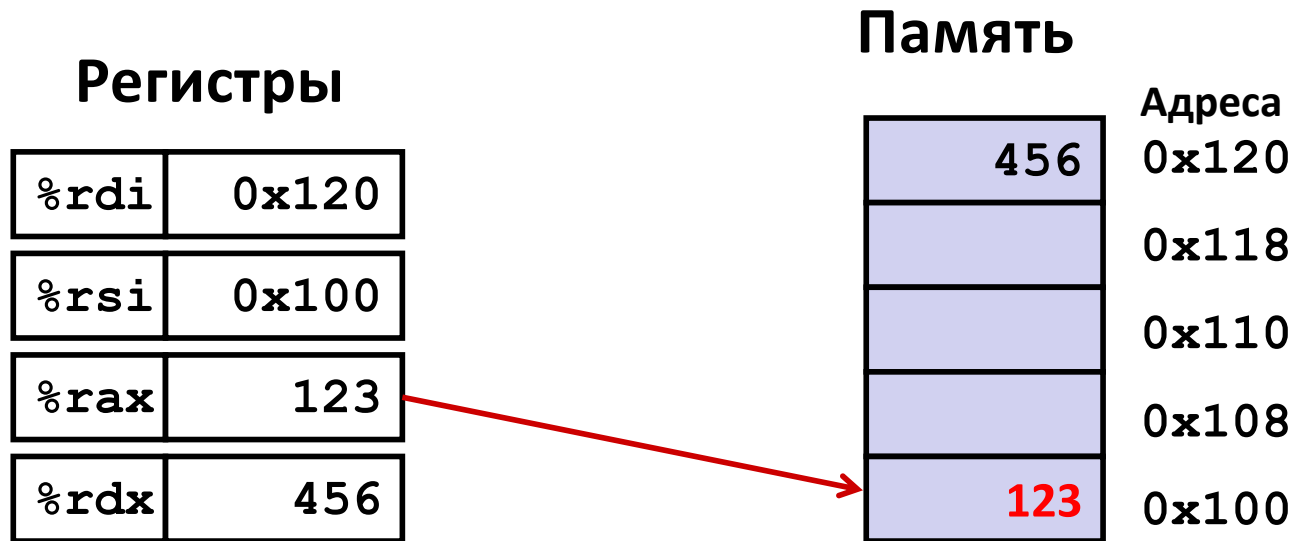
Разбираем swap()



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Разбираем swap()



swap:

```
movq    (%rdi), %rax    # t0 = *xp
movq    (%rsi), %rdx    # t1 = *yp
movq    %rdx, (%rdi)    # *xp = t1
movq    %rax, (%rsi)    # *yp = t0
ret
```

Простые адресации памяти

■ Базовая (R) Mem[Reg[R]]

- Регистр R содержит адрес памяти
- Ага! Раскрытие указателя в Си

```
movq (%rcx) , %rax
```

■ Со смещением D(R) Mem[Reg[R]+D]

- Регистр R содержит адрес начала фрагмента памяти
- Константа D обозначает сдвиг от начала фрагмента

```
movq 8(%rbp) , %rdx
```

Полная адресация

■ Наиболее общая форма

$D(Rb, Ri, S)$

$Mem[Reg[Rb] + S * Reg[Ri] + D]$

- **D:** 1-, 2-, or 4-байтное константное “смещение”
- **Rb:** Базовый регистр: любой из 8 целочисленных регистров
- **Ri:** Индексный регистр: любой, кроме `%rsp`
- **S:** Масштаб: 1, 2, 4, или 8 (*а почему эти числа?*)

■ Специальные случаи

(Rb, Ri)

$Mem[Reg[Rb] + Reg[Ri]]$

$D(Rb, Ri)$

$Mem[Reg[Rb] + Reg[Ri] + D]$

(Rb, Ri, S)

$Mem[Reg[Rb] + S * Reg[Ri]]$

Пример вычисления адресов

<code>%rdx</code>	<code>0xf000</code>
<code>%rcx</code>	<code>0x0100</code>

Выражение	Вычисление адреса	Адрес
<code>0x8(%rdx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%rdx,%rcx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%rdx,%rcx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%rdx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>

Машинный уровень 1: Основы

- Краткая история изделий Интел
- Си, ассемблер, машинный код
- Основы ассемблера: регистры, операнды, пересылки
- Арифметические и логические операции

Команда вычисления адреса

■ `leaq Src, Dest`

- Src адресное выражение
- Присваивает Dest адрес вычисленный по выражению

■ Используется

- Для вычисления адресов без обращения к памяти
 - например, трансляции `p = &x[i];`
- Для вычисления выражений вида `x + k*y`,
 - где `k = 1, 2, 4, или 8`

■ Пример

```
long mul12(long x)
{
    return x*12;
}
```

Результат компиляции в ассемблер:

```
leaq (%rax,%rax,2), %rax ;t <- x+x*2
salq $2, %rax           ;return t<<2
```

Некоторые арифметические команды

■ Двухоперандные команды:

Формат

Вычисления

<code>addq</code>	Src, Dest	$\text{Dest} = \text{Dest} + \text{Src}$	
<code>subq</code>	Src, Dest	$\text{Dest} = \text{Dest} - \text{Src}$	
<code>mulq</code>	Src, Dest	$\text{Dest} = \text{Dest} * \text{Src}$	Беззнаковые
<code>imulq</code>	Src, Dest	$\text{Dest} = \text{Dest} * \text{Src}$	Знаковые
<code>salq</code>	Src, Dest	$\text{Dest} = \text{Dest} \ll \text{Src}$	Синоним: <code>shlq</code>
<code>sarq</code>	Src, Dest	$\text{Dest} = \text{Dest} \gg \text{Src}$	Арифметический
<code>shrq</code>	Src, Dest	$\text{Dest} = \text{Dest} \gg \text{Src}$	Логический
<code>xorq</code>	Src, Dest	$\text{Dest} = \text{Dest} \wedge \text{Src}$	
<code>andq</code>	Src, Dest	$\text{Dest} = \text{Dest} \& \text{Src}$	
<code>orq</code>	Src, Dest	$\text{Dest} = \text{Dest} \text{Src}$	

■ Следите за порядком аргументов!

■ Почти не различаются `signed` и `unsigned int` (почему?)

Некоторые арифметические команды

■ Однооперандные команды

<code>incq</code>	Dest	$\text{Dest} = \text{Dest} + 1$
<code>decq</code>	Dest	$\text{Dest} = \text{Dest} - 1$
<code>negq</code>	Dest	$\text{Dest} = -\text{Dest}$
<code>notq</code>	Dest	$\text{Dest} = \sim\text{Dest}$

■ Больше информации – в книге

Пример арифметических выражений

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

arith:

```
leaq    (%rdi,%rsi), %rax
addq    %rdx, %rax
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx
leaq    4(%rdi,%rdx), %rcx
imulq   %rcx, %rax
ret
```

Интересные команды

- **leaq**: вычисление адресов
- **salq**: сдвиг
- **imulq**: умножение
 - используется лишь однажды

Разбираем arith

```
long arith
(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

arith:

```
leaq    (%rdi,%rsi), %rax    # t1
addq    %rdx, %rax          # t2
leaq    (%rsi,%rsi,2), %rdx
salq    $4, %rdx            # t4
leaq    4(%rdi,%rdx), %rcx   # t5
imulq   %rcx, %rax          # rval
ret
```

%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	t1, t2, rval
%rdx	t4
%rcx	t5

Машинный уровень 1: Основы. Сводка

■ Эволюция процессоров и архитектур Intel

- Эволюционное конструирование приводит к странностям и неестественным свойствам

■ Си, ассемблер, машинный код

- Новые формы видимого состояния: счётчик команд, регистры, ...
- Компилятор должен преобразовать операторы, выражения, процедуры в последовательности низкоуровневых команд

■ Основы ассемблера: регистры, операнды, пересылки

- Команда **move** x86-64 обеспечивает множество вариантов пересылки

■ Арифметика

- Си компилятор будет выдавать различные комбинации команд для реализации одинаковых вычислений