

# Machine-Level Programming I: Basics

15-213/18-213/15-213: Introduction to Computer Systems  
5<sup>th</sup> Lecture, September 12, 2017

**Today's Instructor:**

Phil Gibbons

# 程序的转换与机器级表示

- **主要教学目标**
  - 了解高级语言与汇编语言、汇编语言与机器语言之间的关系
  - 掌握有关指令格式、操作数类型、寻址方式、操作类型等内容
  - 了解高级语言源程序中的语句与机器级代码之间的对应关系
  - 了解复杂数据类型（数组、结构等）的机器级实现
- **主要教学内容**
  - 介绍C语言程序与机器级指令之间的对应关系。
  - 主要包括：程序转换概述、IA-64指令系统、C语言中控制语句和过程调用等机器级实现、复杂数据类型（数组、结构等）的机器级实现等。
  - 本章所用的机器级表示主要以汇编语言形式表示为主。

**采用逆向工程方法！**

# 程序的机器级表示

- 分以下五个部分介绍

- **第一讲：程序转换概述**

- 机器指令和汇编指令
- 机器级程序员感觉到的属性和功能特性
- 高级语言程序转换为机器代码的过程

- **第二讲：IA-32 /x86-64指令系统**

- **第三讲：C语言程序的机器级表示**

- 选择语句的机器级表示
- 循环结构的机器级表示
- 过程调用的机器级表示

- **第四讲：复杂数据类型的分配和访问**

- 数组的分配和访问
- 结构体数据的分配和访问
- 联合体数据的分配和访问
- 数据的对齐

- **第五讲：越界访问和缓冲区溢出**

从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

围绕C语言中的语句和复杂数据类型，解释其在底层机器级的实现方法

# “指令”的概念

计算机中的指令有**微指令**、**机器指令**和**伪（宏）指令**之分

**微指令**是微程序级命令，指在微程序控制的计算机中，同时发出的控制信号所执行的一组微操作，属于硬件范畴

**伪指令**是用于对汇编过程进行控制的指令，该类指令并不是可执行指令，没有机器代码，只用于汇编过程中为汇编程序提供汇编信息，由若干机器指令组成的指令序列，属于软件范畴

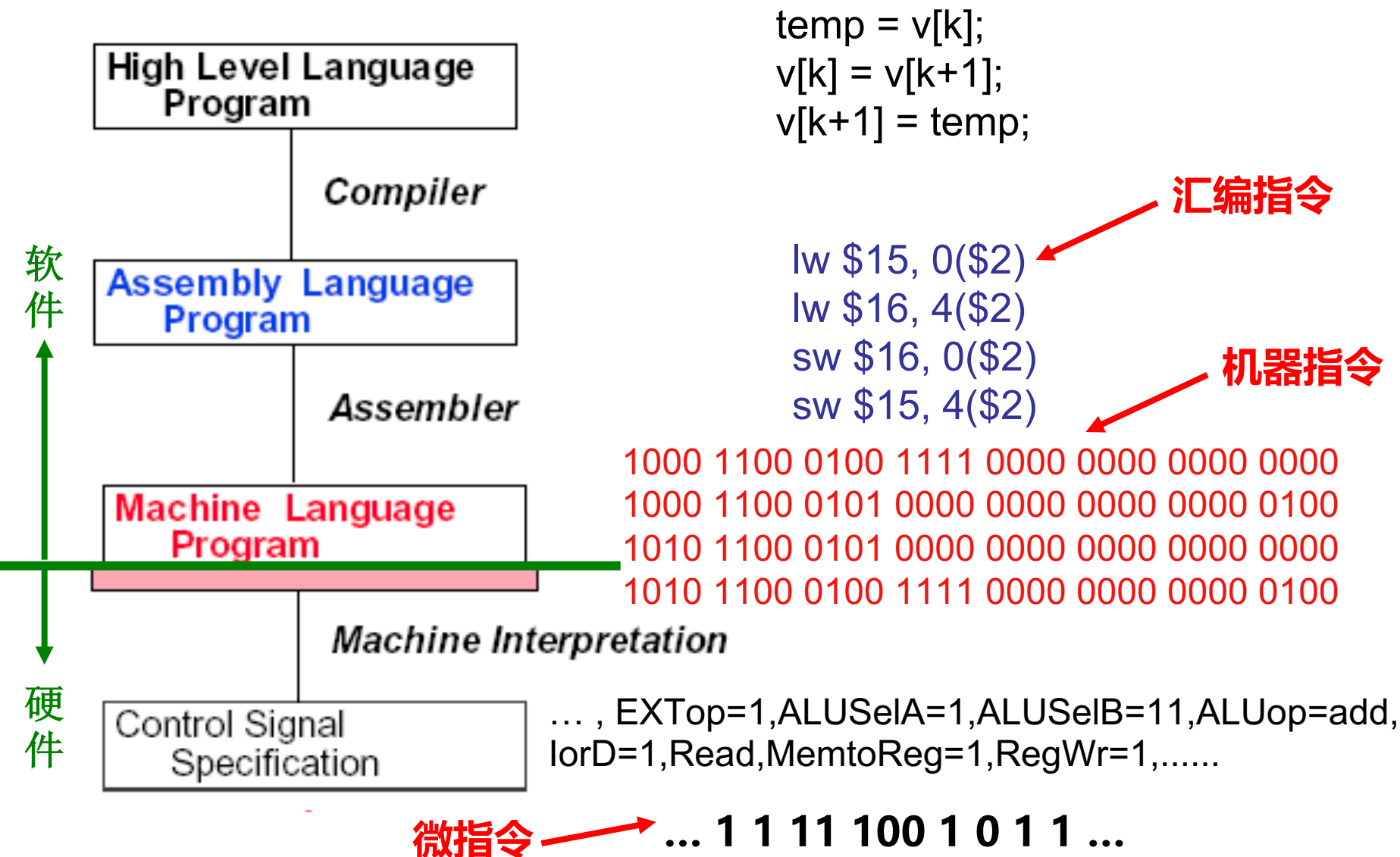
**机器指令**介于二者之间，处于硬件和软件的交界面

本章中提及的指令都指机器指令

**汇编指令**是机器指令的汇编表示形式，即符号表示

机器指令和汇编指令一一对应，它们都与具体机器结构有关，都属于**机器级指令**

# 回顾：Hardware/Software Interface



# 机器级指令

- 机器指令和汇编指令一一对应，都是机器级指令
- 机器指令是一个0/1序列，由若干**字段**组成

补码**11111010**  
的真值为多少？

100010 DW	mod	reg	r/m	disp8
100010 0 0	01	001	001	11111010

操作码

寻址方式

寄存器编号

立即数(位移量)

- 汇编指令是机器指令的符号表示（可能有不同的格式）

**mov [bx+di-6], cl**    或    **movb %cl, -6(%bx,%di)**

Intel格式

AT&T 格式

**mov、movb、bx、%bx**等都是**助记符**

指令的功能为： **$M[R[bx] + R[di] - 6] \leftarrow R[cl]$**

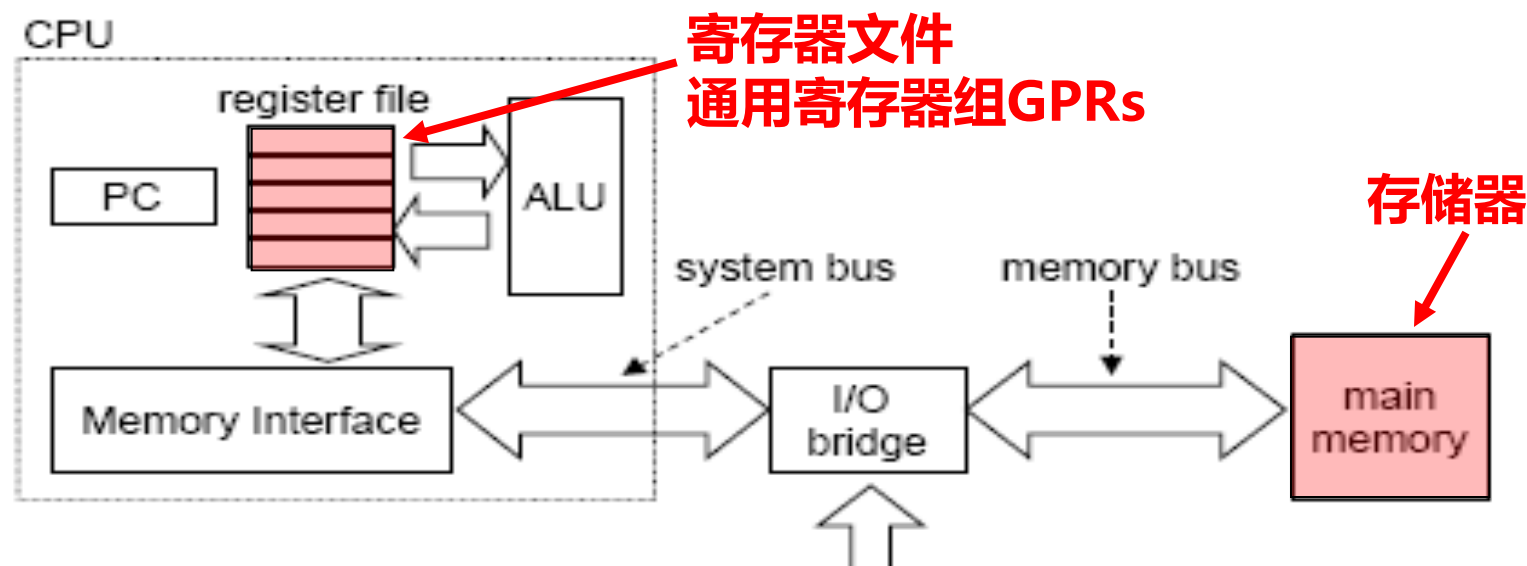
R: 寄存器内容

M: 存储单元内容

**寄存器传送语言 RLT (Register Transfer Language)**

# 计算机中数据的存储

## • 计算机中的数据存放在哪里？



指令中需给出的信息：

操作性质（操作码）

源操作数1 或/和 源操作数2 （立即数、寄存器编号、存储地址）

目的操作数地址 （寄存器编号、存储地址）

存储地址的描述与操作数的数据结构有关！

# 指令集体系结构ISA

- ISA (Instruction Set Architecture) 位于软件和硬件之间
- 硬件的功能通过ISA提供出来
- 软件通过ISA规定的“指令”使用硬件
- ISA规定了：
  - 可执行的指令的集合，包括指令格式、操作种类以及每种操作对应的操作数的相应规定；
  - 指令可以接受的操作数的类型；
  - 操作数所能存放的寄存器组的结构，包括每个寄存器的名称、编号、长度和用途；
  - 操作数所能存放的存储空间的大小和编址方式；
  - 操作数在存储空间存放时按照大端还是小端方式存放；
  - 指令获取操作数的方式，即寻址方式；
  - 指令执行过程的控制方式，包括程序计数器、条件码定义等。



# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

# Intel x86 Processors

- Dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

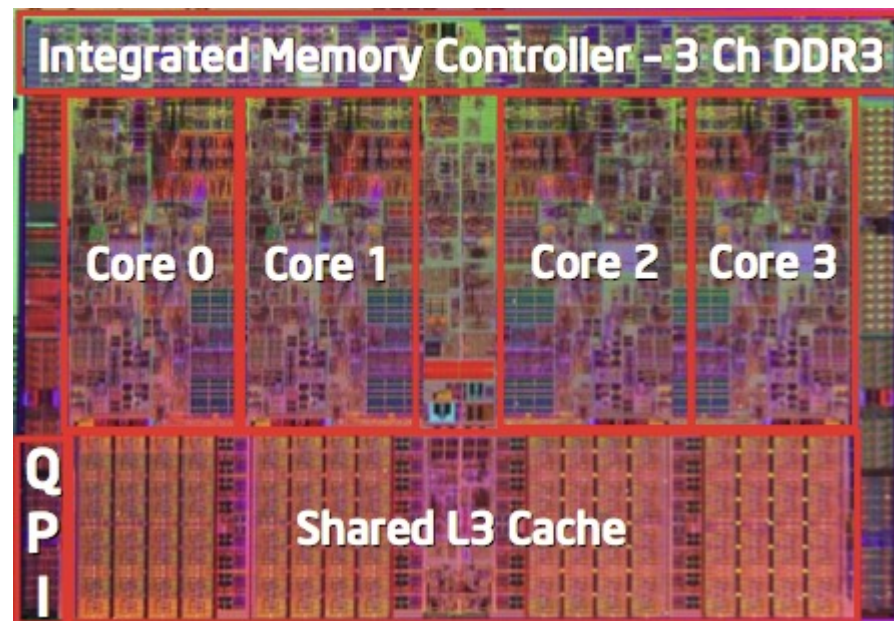
# Intel x86 Evolution: Milestones

<i><b>Name</b></i>	<i><b>Date</b></i>	<i><b>Transistors</b></i>	<i><b>MHz</b></i>
■ <b>8086</b>	<b>1978</b>	<b>29K</b>	<b>5-10</b>
<ul style="list-style-type: none"> <li>First 16-bit Intel processor. Basis for IBM PC &amp; DOS</li> <li>1MB address space</li> </ul>			
■ <b>386</b>	<b>1985</b>	<b>275K</b>	<b>16-33</b>
<ul style="list-style-type: none"> <li>First 32 bit Intel processor , referred to as IA32</li> <li>Added “flat addressing”, capable of running Unix</li> </ul>			
■ <b>Pentium 4E</b>	<b>2004</b>	<b>125M</b>	<b>2800-3800</b>
<ul style="list-style-type: none"> <li>First 64-bit Intel x86 processor, referred to as x86-64</li> </ul>			
■ <b>Core 2</b>	<b>2006</b>	<b>291M</b>	<b>1060-3333</b>
<ul style="list-style-type: none"> <li>First multi-core Intel processor</li> </ul>			
■ <b>Core i7</b>	<b>2008</b>	<b>731M</b>	<b>1600-4400</b>
<ul style="list-style-type: none"> <li>Four cores (our shark machines)</li> </ul>			

# Intel x86 Processors, cont.

## ■ Machine Evolution

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2000	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M



## ■ Added Features

- Instructions to support **multimedia** operations
- Instructions to enable **more efficient conditional** operations
- Transition from 32 bits to 64 bits
- More cores

# Intel x86 Processors, cont.

## ■ Past Generations

### Process technology

■ 1 <sup>st</sup> Pentium Pro	1995	600 nm
■ 1 <sup>st</sup> Pentium III	1999	250 nm
■ 1 <sup>st</sup> Pentium 4	2000	180 nm
■ 1 <sup>st</sup> Core 2 Duo	2006	65 nm

## ■ Recent & Upcoming Generations

1.	Nehalem	2008	45 nm
2.	Sandy Bridge	2011	32 nm
3.	Ivy Bridge	2012	22 nm
4.	Haswell	2013	22 nm
5.	Broadwell	2014	14 nm
6.	Skylake	2015	14 nm
7.	Kaby Lake	2016	14 nm
■	Coffee Lake	2017?	14 nm
■	Cannonlake	2018?	10 nm

**Process technology dimension**  
 = width of narrowest wires  
 (10 nm  $\approx$  100 atoms wide)

# 2017 State of the Art: Skylake

## ■ Mobile Model: Core i7

- 2.6-2.9 GHz
- 45 W

华为mate20的麒麟980处理器于2018年10月国内发布，全球首款7纳米工艺的soc芯片，ARM Cortex-A76内核，主频2133MHz 1平方厘米集成晶体管数量69亿

## ■ Desktop Model: Core i7

- Integrated graphics
- 2.8-4.0 GHz
- 35-91 W

## ■ Server Model: Xeon

- Integrated graphics
- Multi-socket enabled
- 2-3.7 GHz
- 25-80 W

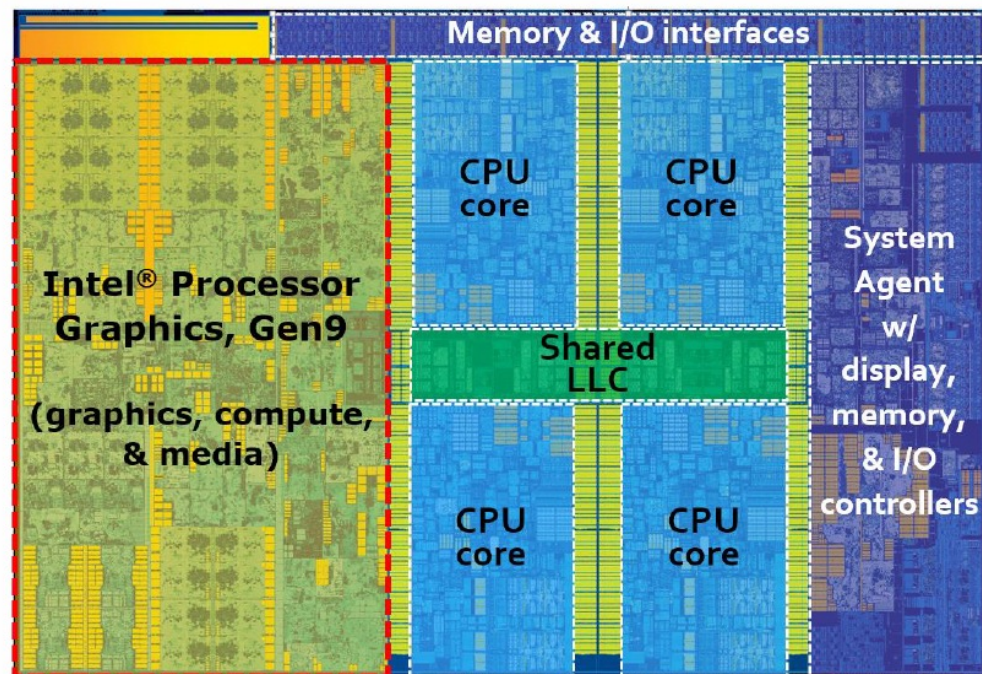


Figure 1: Architecture components layout for an Intel® Core™ i7 processor 6700K for desktop systems. This SoC contains 4 CPU cores, outlined in blue dashed boxes. Outlined in the red dashed box, is an Intel® HD Graphics 530. It is a one-slice instantiation of Intel processor graphics gen9 architecture.

# x86 Clones: Advanced Micro Devices (AMD)

## ■ Historically

- AMD has followed just behind Intel
- A little bit slower, a lot cheaper

## ■ Then

- Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
- Built Opteron: tough competitor to Pentium 4
- Developed x86-64, their own extension to 64 bits

## ■ Recent Years

- Intel got its act together
  - Leads the world in semiconductor technology
- AMD has fallen behind
  - Relies on external semiconductor manufacturer

# Intel's 64-Bit History

- **2001: Intel Attempts Radical Shift from IA32 to IA64**
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Performance disappointing
- **2003: AMD Steps in with Evolutionary Solution**
  - x86-64 (now called “AMD64”)
- **Intel Felt Obligated to Focus on IA64**
  - Hard to admit mistake or that AMD is better
- **2004: Intel Announces EM64T extension to IA32**
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64!
- **All but low-end x86 processors support x86-64**
  - But, lots of code still runs in 32-bit mode



# Our Coverage

## ■ IA32

- The traditional x86

## ■ x86-64

- The standard
- `bupt1> gcc hello.c`

## ■ Presentation

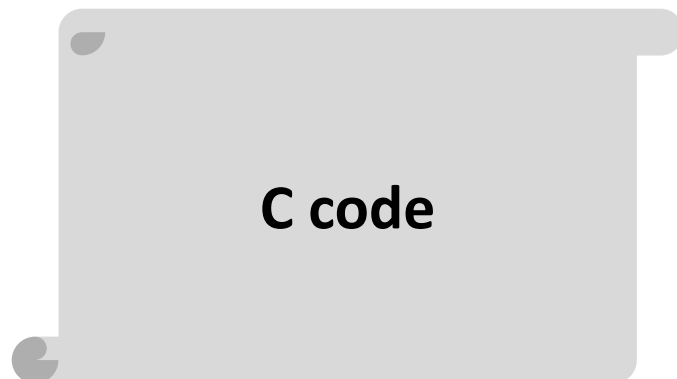
- Book covers x86-64
- Web aside on IA32
- We will only cover x86-64

# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- **Assembly Basics: Registers, operands, move**
- Arithmetic & logical operations
- C, assembly, machine code

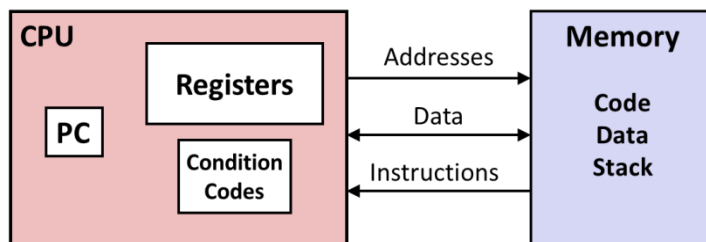
# Levels of Abstraction

C programmer



**Nice clean layers,  
but beware...**

Assembly programmer



Computer Designer

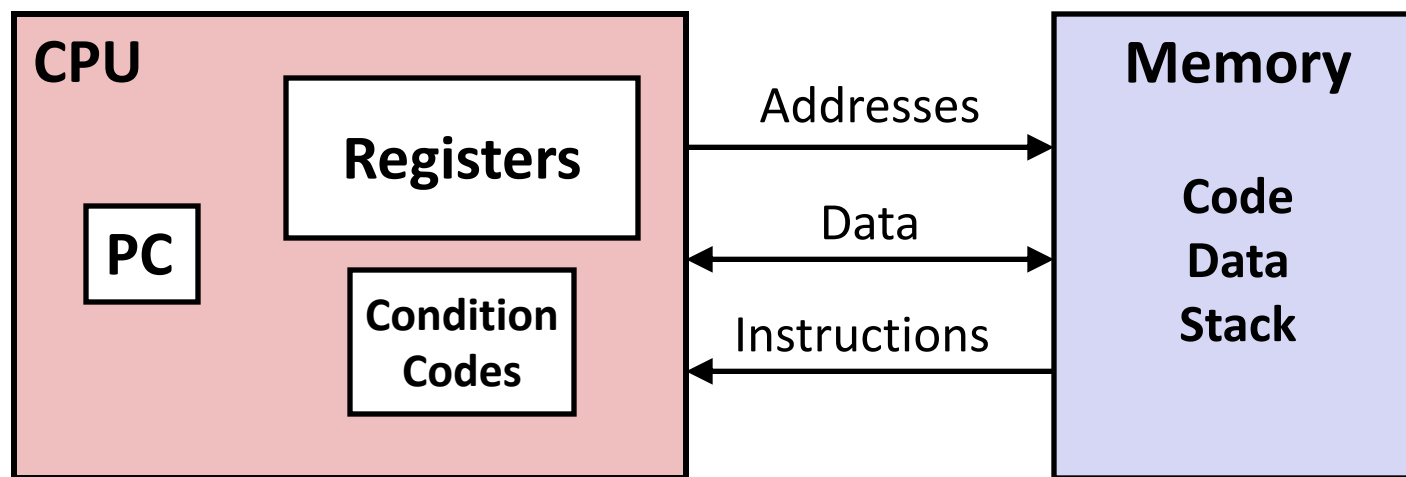
Caches, clock freq, layout, ...

**Of course, you know that: It's why you are taking this course.**

# Definitions

- **Architecture:** (also ISA: instruction set architecture) The parts of a processor design that one needs to understand for writing correct machine/assembly code
  - Examples: instruction set specification, registers
  - **Machine Code:** The byte-level programs that a processor executes
  - **Assembly Code:** A text representation of machine code
- **Microarchitecture:** Implementation of the architecture
  - Examples: cache sizes and core frequency
- **Example ISAs:**
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: Used in almost all mobile phones
  - RISC V: New open-source ISA

# Assembly/Machine Code View



## Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

## ■ Memory

- Byte addressable array
- Code and user data
- Stack to support procedures

# Assembly Characteristics: Data Types

- **“Integer” data of 1, 2, 4, or 8 bytes**
  - Data values
  - Addresses (untyped pointers)
- **Floating point data of 4, 8, or 10 bytes**
- **(SIMD vector data types of 8, 16, 32 or 64 bytes)**
- **Code: Byte sequences encoding series of instructions**
- **No aggregate types such as arrays or structures**
  - Just contiguously allocated bytes in memory

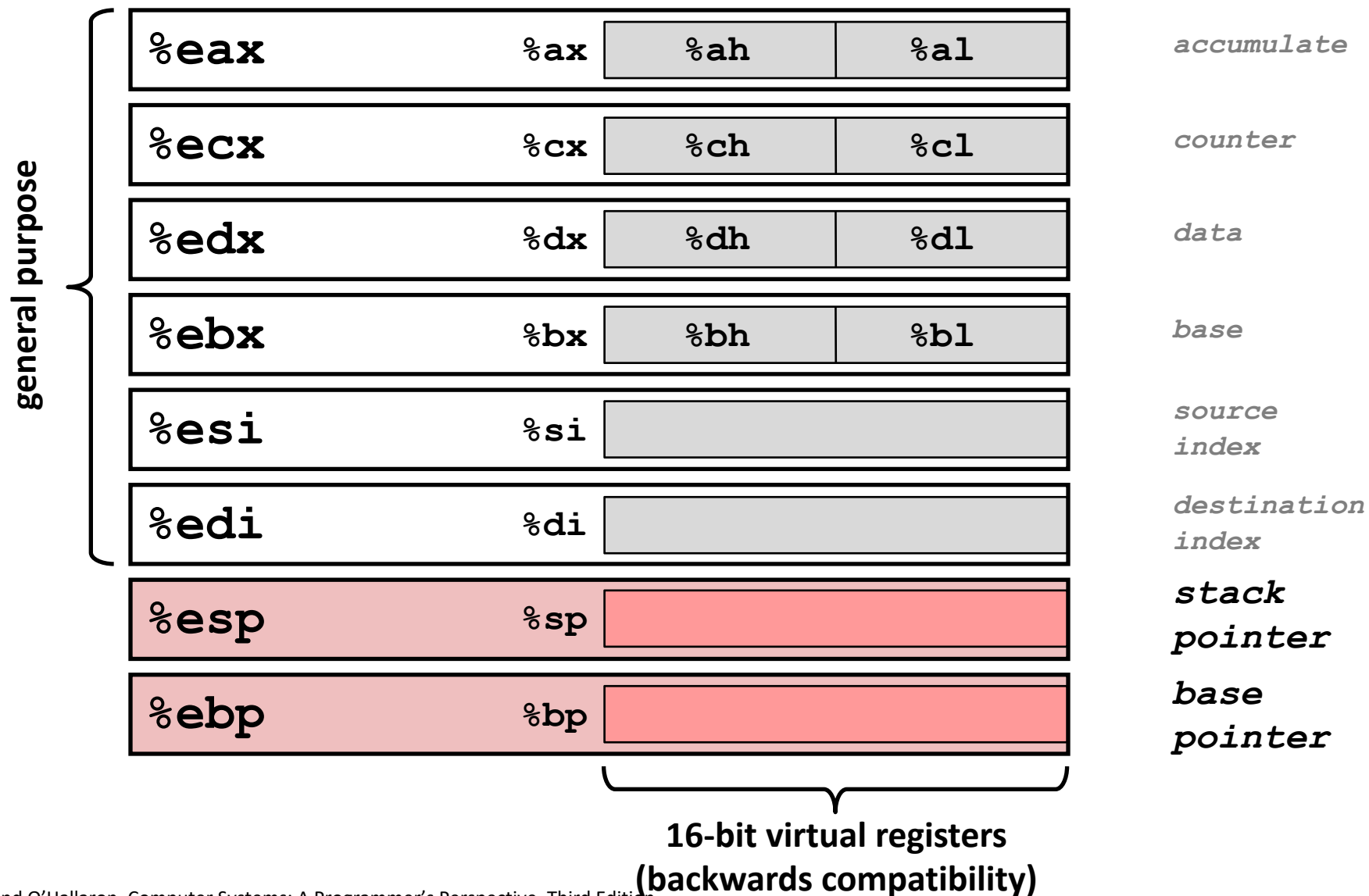
# x86-64 Integer Registers

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

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

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)
- Not part of memory (or cache)

# Some History: IA32 Registers





# IA-32中通用寄存器中的编号

编号	8 位寄存器	16 位寄存器	32 位寄存器	64 位寄存器	128 位寄存器
000	AL	AX	EAX	MM0 / ST(0)	XMM0
001	CL	CX	ECX	MM1 / ST(1)	XMM1
010	DL	DX	EDX	MM2 / ST(2)	XMM2
011	BL	BX	EBX	MM3 / ST(3)	XMM3
100	AH	SP	ESP	MM4 / ST(4)	XMM4
101	CH	BP	EBP	MM5 / ST(5)	XMM5
110	DH	SI	ESI	MM6 / ST(6)	XMM6
111	BH	DI	EDI	MM7 / ST(7)	XMM7

反映了体系结构发展的轨迹，字长不断扩充，指令保持兼容

ST (0) ~ ST (7) 是80位，MM0 ~MM7使用其低64位

# Assembly Characteristics: Operations

- **Transfer data between memory and register**
  - Load data from memory into register
  - Store register data into memory
  
- **Perform arithmetic function on register or memory data**
  
- **Transfer control**
  - Unconditional jumps to/from procedures
  - Conditional branches
  - Indirect branches

# IA-32的寻址方式

- 寻址方式
  - 根据指令给定信息得到操作数或操作数地址
- 操作数所在的位置
  - 指令中：立即寻址
  - 寄存器中：寄存器寻址
  - 存储单元中（属于存储器操作数，按字节编址）：其他寻址方式
- 存储器操作数的寻址方式与微处理器的工作模式有关
  - 两种工作模式：实地址模式和保护模式
- 实地址模式（基本用不到）
  - 为与8086/8088兼容而设，加电或复位时
  - 寻址空间为1MB，20位地址： $(CS) \ll 4 + (IP)$
- 保护模式（需要掌握）
  - 加电后进入，采用虚拟存储管理，多任务情况下隔离、保护
  - 80286以上高档微处理器最常用的工作模式
  - 寻址空间为 $2^{32}B$ ，32位线性地址分段（段基址+段内偏移量）

# 保护模式下的寻址方式

寻址方式	说明
立即寻址	指令直接给出操作数
寄存器寻址	指定的寄存器R的内容为操作数
位移	$LA = (SR) + A$
基址寻址	$LA = (SR) + (B)$
基址加位移	$LA = (SR) + (B) + A$
比例变址加位移	$LA = (SR) + (I) \times S + A$
基址加变址加位移	$LA = (SR) + (B) + (I) + A$
基址加比例变址加位移	$LA = (SR) + (B) + (I) \times S + A$
相对寻址	$LA = (PC) + A$

存储器操作数

跳转目标指令地址

注: LA:线性地址 (X):X的内容 SR:段寄存器 PC:程序计数器 R:寄存器  
A:指令中给定地址段的位移量 B:基址寄存器 I:变址寄存器 S:比例系数

- **SR段寄存器 (间接) 确定操作数所在段的段基址**
- **有效地址给出操作数在所在段的偏移地址**

# 存储器操作数的寻址方式(32位)

int x;

float a[100];

short b[4][4];

char c;

long \*c1;

double d[10];

**a[i]的地址如何计算?**

**104**+**i**×**4**

**i=99**时,  $104 + 99 \times 4 = 500$

**b[i][j]的地址如何计算?**

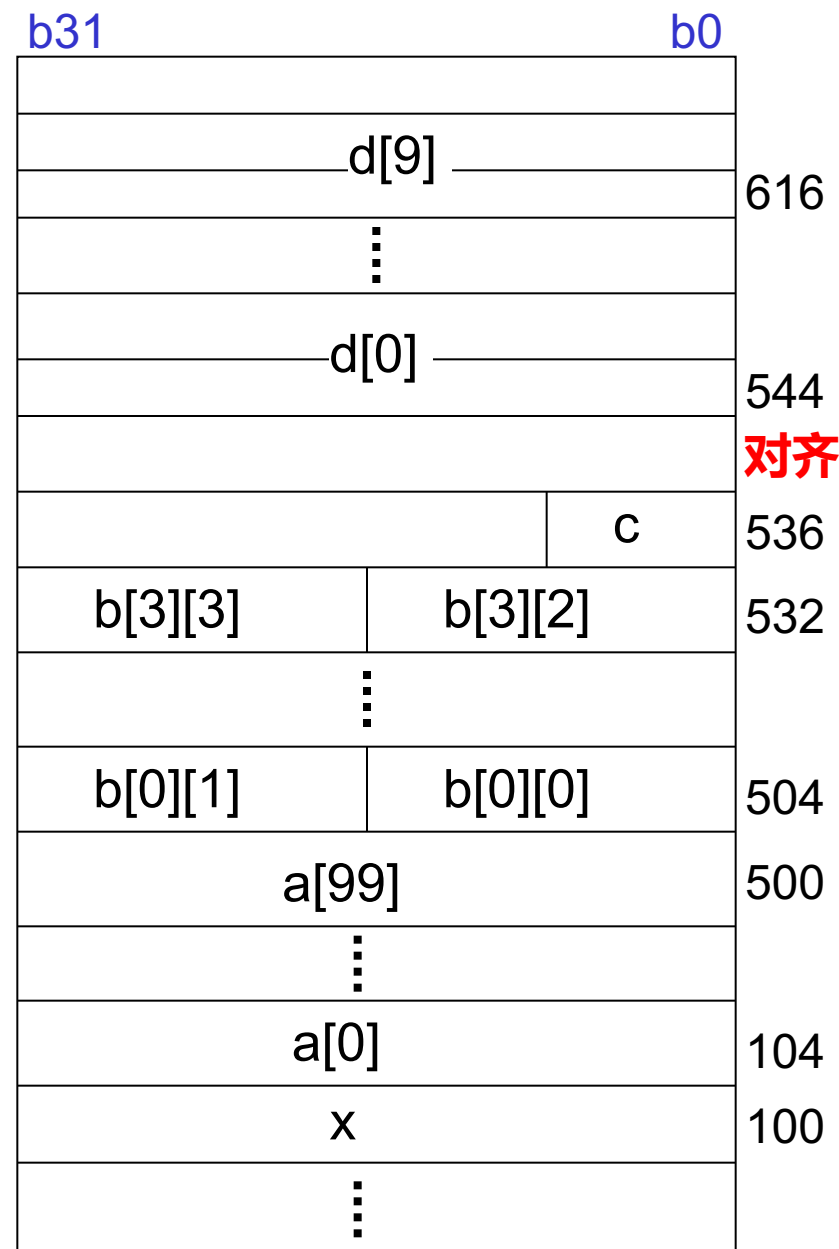
**504**+**i**×**8**+**j**×**2**

**i=3**、**j=2**时,  $504 + 24 + 4 = 532$

**d[i]的地址如何计算?**

**544**+**i**×**8**

**i=9**时,  $544 + 9 \times 8 = 616$



# Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
<code>char</code>	1	1	1
<code>short</code>	2	2	2
<code>int</code>	4	4	4
<code>long</code>	4	8	8
<code>float</code>	4	4	4
<code>double</code>	8	8	8
<code>pointer</code>	4	8	8

# 存储器操作数的寻址方式

```
int x;
float a[100];
short b[4][4];
char c;
long *c1;
double d[10];
```

各变量应采用什么寻址方式？

**x、c:** 位移 / 基址

**a[i]:**  $104 + i \times 4$ , 比例变址 + 位移

**d[i]:**  $544 + i \times 8$ , 比例变址 + 位移

**b[i][j]:**  $504 + i \times 8 + j \times 2$ ,

基址 + 比例变址 + 位移

将b[i][j]取到EAX中的指令可以是：

“**movw**  $504(\%ebp, \%esi, 2), \%eax$ ”

其中,  $i \times 8$ 在EBP中, j在ESI中,

**2为比例因子**

b31		b0	
d[9]			616
⋮			
d[0]			544
		c	536
b[3][3]	b[3][2]		532
⋮			
b[0][1]	b[0][0]		504
a[99]			500
⋮			
a[0]			104
x			100
⋮			

# Moving Data

ATT格式

## ■ Moving Data

`movq Source, Dest`

## ■ Operand Types

- **Immediate:** Constant integer data
  - Example: `$0x400`, `$-533`
  - Like C constant, but prefixed with ``$'`
  - Encoded with **1, 2, or 4** bytes
- **Register:** One of 16 integer registers
  - Example: `%rax`, `%r13`
  - But `%rsp` reserved for special use
  - Others have special uses for particular instructions
- **Memory** 8 consecutive bytes of memory at address given by register
  - Simplest example: `(%rax)`
  - Various other “addressing modes”

C, C++规定, 16进制数必须以 0x开头

`%rax`

`%rcx`

`%rdx`

`%rbx`

`%rsi`

`%rdi`

`%rsp`

`%rbp`

`%rn (n=8~15)`

**Warning: Intel docs use**  
**`mov Dest, Source`**



# movq Operand Combinations

	Source	Dest	Src, Dest	C Analog
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;

**Cannot do memory-memory transfer with a single instruction**

# Simple Memory Addressing Modes

## ■ Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C

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

## ■ Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

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

# Example of Simple Addressing Modes

```
void  
whatAmI (<type> a, <type> b)  
{  
    ???  
}
```

`%rdi`

`%rsi`

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

# Example of Simple Addressing Modes

```
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
```

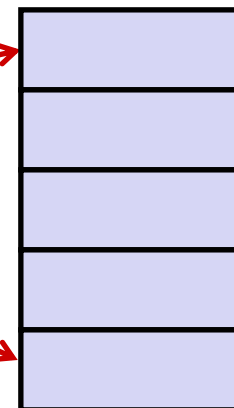
# Understanding Swap()

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

## Registers

%rdi	
%rsi	
%rax	
%rdx	

## Memory



Register	Value
%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
```

# Understanding Swap()

## Registers

<code>%rdi</code>	<code>0x120</code>
<code>%rsi</code>	<code>0x100</code>
<code>%rax</code>	
<code>%rdx</code>	

## Memory

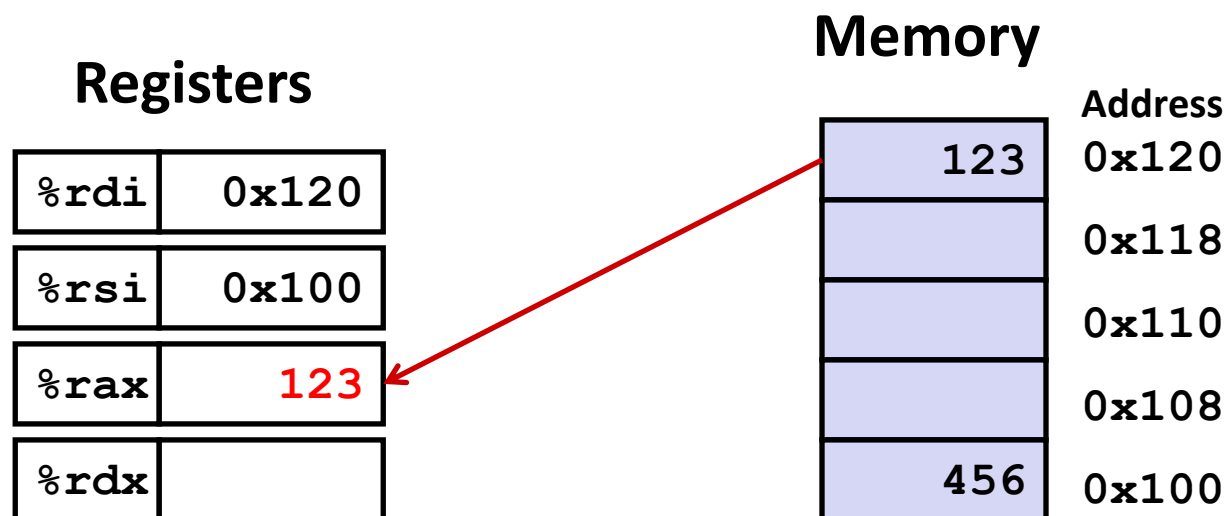
Address	
<code>0x120</code>	123
<code>0x118</code>	
<code>0x110</code>	
<code>0x108</code>	
<code>0x100</code>	456

**swap:**

```

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

# Understanding Swap()



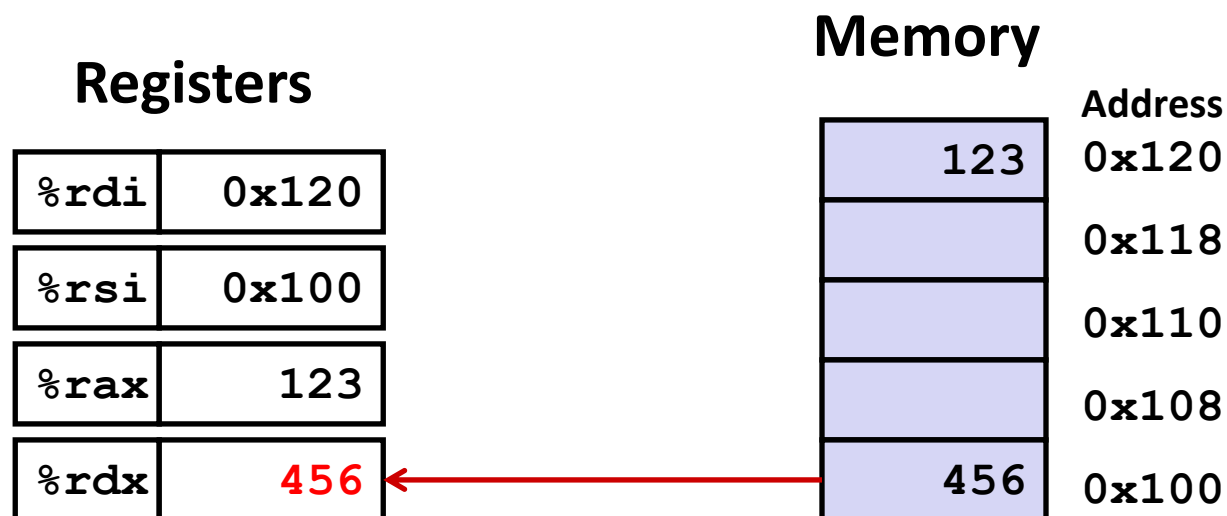
swap:

```

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

```

# Understanding Swap()



**swap:**

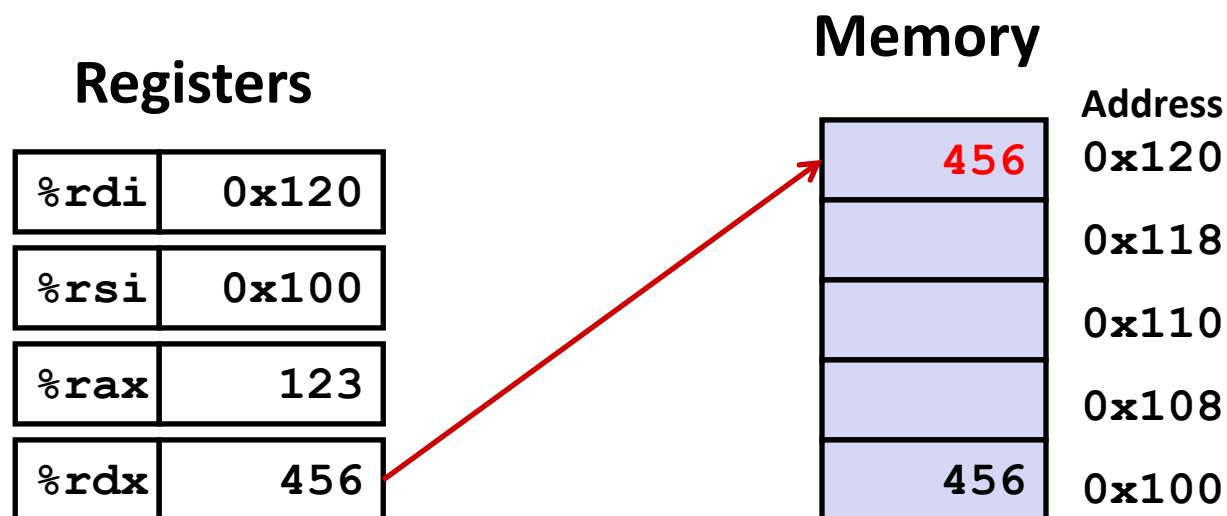
```

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

```



# Understanding Swap()



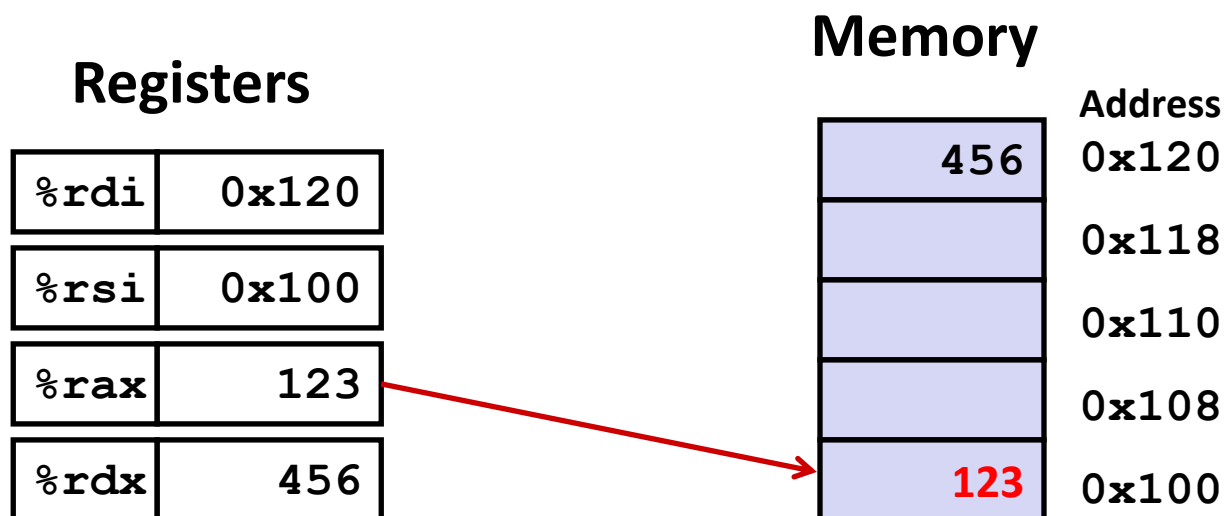
**swap:**

```

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

```

# Understanding Swap()



**swap:**

```

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

```

# Simple Memory Addressing Modes

## ■ Normal (R) Mem[Reg[R]]

- Register R specifies memory address
- Aha! Pointer dereferencing in C

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

## ■ Displacement D(R) Mem[Reg[R]+D]

- Register R specifies start of memory region
- Constant displacement D specifies offset

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

# Complete Memory Addressing Modes

## ■ Most General Form

**D(Rb,Ri,S)**

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

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, **except for %rsp**
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

## ■ Special Cases

**(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]]**

# Address Computation Examples

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

**D(Rb,Ri,S)**

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

- D: Constant “displacement” 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Expression	Address Computation	Address
<code>0x8(%rdx)</code>		
<code>(%rdx,%rcx)</code>		
<code>(%rdx,%rcx,4)</code>		
<code>0x80(,%rdx,2)</code>		

# Today: Machine Programming I: Basics

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- **Arithmetic & logical operations**
- C, assembly, machine code

# IA-32常用指令类型

## (1) 传送指令

### — 通用数据传送指令

MOV: 一般传送, 包括movb、movw和movl等

MOVS: 符号扩展传送, 如movsbw、movswl等

MOVZ: 零扩展传送, 如movzwl、movzbl等

XCHG: 数据交换

PUSH/POP: 入栈/出栈, 如pushl, pushw, popl, popw等

### — 地址传送指令

LEA: 加载有效地址, 如leal (%edx,%eax), %eax” 的功能为

$R[edx] \leftarrow R[edx] + R[edx]$ , 执行前, 若 $R[edx] = i$ ,

$R[edx] = j$ , 则指令执行后,  $R[edx] = i + j$

### — 输入输出指令

IN和OUT: I/O端口与寄存器之间的交换

### — 标志传送指令

PUSHF、POPF: 将EFLAG压栈, 或将栈顶内容送EFLAG

# IA-32常用指令类型

## (2) 定点算术运算指令

- 加 / 减运算 (影响标志、不区分无/带符号)

ADD: 加, 包括addb、addw、addl等

SUB: 减, 包括subb、subw、subl等

- 增1 / 减1运算 (影响除CF以外的标志、不区分无/带符号)

INC: 加, 包括incb、incw、incl等

DEC: 减, 包括decb、decw、decl等

- 取负运算 (影响标志、若对0取负, 则结果为0/CF=0, 否则CF=1)

NEG: 取负, 包括negb、negw、negl等

- 比较运算 (做减法得到标志、不区分无/带符号)

CMP: 比较, 包括cmpb、cmpw、cmpl等

- 乘 / 除运算 (不影响标志、区分无/带符号)

MUL / IMUL: 无符号乘 / 带符号乘

DIV / IDIV: 带无符号除 / 带符号除



# IA-32常用指令类型

## (3) 按位运算指令

- 逻辑运算（仅NOT不影响标志，其他指令OF=CF=0，而ZF和SF根据结果设置：若全0，则ZF=1；若最高位为1，则SF=1）

NOT: 非，包括 notb、notw、notl等

AND: 与，包括 andb、andw、andl等

OR: 或，包括 orb、orw、orl等

XOR: 异或，包括 xorb、xorw、xorl等

TEST: 做“与”操作测试，仅影响标志

- 移位运算（左/右移时，最高/最低位送CF）

SHL/SHR: 逻辑左/右移，包括 shlb、shrw、shrl等

SAL/SAR: 算术左/右移，左移判溢出，右移高位补符

ROL/ROR: 循环左/右移，包括 rolb、rorw、roll等

RCL/RCR: 带循环左/右移，将CF作为操作数一部分循环移位

**以上内容不要死记硬背，遇到具体指令时能查阅到并理解即可。**

# IA-32常用指令类型

## (4) 控制转移指令

指令执行可**按顺序** 或 **跳转到转移目标指令处**执行

- **无条件转移指令**

**JMP DST**: 无条件转移到目标指令DST处执行

- **条件转移**

**Jcc DST**: cc为条件码, 根据标志 (条件码) 判断是否满足条件, 若满足, 则转移到目标指令DST处执行, 否则按顺序执行

- **条件设置**

**SETcc DST**: 将条件码cc保存到DST (通常是一个8位寄存器)

- **调用和返回指令 (用于过程调用)**

**CALL DST**: **返回地址RA**入栈, 转DST处执行

**RET**: 从栈中取出返回地址RA, 转到RA处执行

**以上内容不要死记硬背, 遇到具体指令时能查阅到并理解即可。**

# Address Computation Instruction

## ■ `leaq Src, Dst`

- Src is address mode expression
- Set Dst to address denoted by expression

## ■ Uses

- Computing addresses without a memory reference
  - E.g., translation of `p = &x[i];`
- Computing arithmetic expressions of the form  $x + k*y$ 
  - $k = 1, 2, 4, \text{ or } 8$

## ■ Example

```
long m12(long x)
{
    return x*12;
}
```

```
leaq ($0x1000), %rax    # %rax = 1000H
leaq (%rbx), %rax       # %rax = %rbx
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t = x+2*x
salq $2, %rax            # return t<<2
```

# Some Arithmetic Operations

## ■ Two Operand Instructions:

Format	Computation		
<code>addq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} + \text{Src}$	
<code>subq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} - \text{Src}$	
<code>imulq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} * \text{Src}$	
<code>salq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \ll \text{Src}$	Also called <code>shlq</code>
<code>sarq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \gg \text{Src}$	Arithmetic
<code>shrq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \gg \text{Src}$	Logical
<code>xorq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \wedge \text{Src}$	
<code>andq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest} \& \text{Src}$	
<code>orq</code>	<code>Src, Dest</code>	$\text{Dest} = \text{Dest}   \text{Src}$	

- Watch out for argument order! *Src, Dest*  
(Warning: Intel docs use “op *Dest, Src*”)

- No distinction between signed and unsigned int (why?)

# Some Arithmetic Operations

## ■ One Operand Instructions

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


## ■ See book for more instructions

# Arithmetic Expression Example

```
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
```



## Interesting Instructions

- **leaq**: address computation
- **salq**: shift
- **imulq**: multiplication
  - But, only used once

# Understanding Arithmetic Expression

## Example

```

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

```

Register	Use(s)
%rdi	Argument <b>x</b>
%rsi	Argument <b>y</b>
%rdx	Argument <b>z</b> , t4
%rax	t1, t2, rval
%rcx	t5

# Machine Programming I: Summary

## ■ History of Intel processors and architectures

- Evolutionary design leads to many quirks and artifacts

## ■ C, assembly, machine code

- New forms of visible state: program counter, registers, ...
- Compiler must transform statements, expressions, procedures into low-level instruction sequences

## ■ Assembly Basics: Registers, operands, move

- The x86-64 move instructions cover wide range of data movement forms

## ■ Arithmetic

- C compiler will figure out different instruction combinations to carry out computation