Machine-Level Programming I: Basics

15-213/18-213/15-213: Introduction to Computer Systems

Today's Instructor:

Wenbin Yao

程序的转换与机器级表示

- 主要教学目标
  - 了解高级语言、汇编语言、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

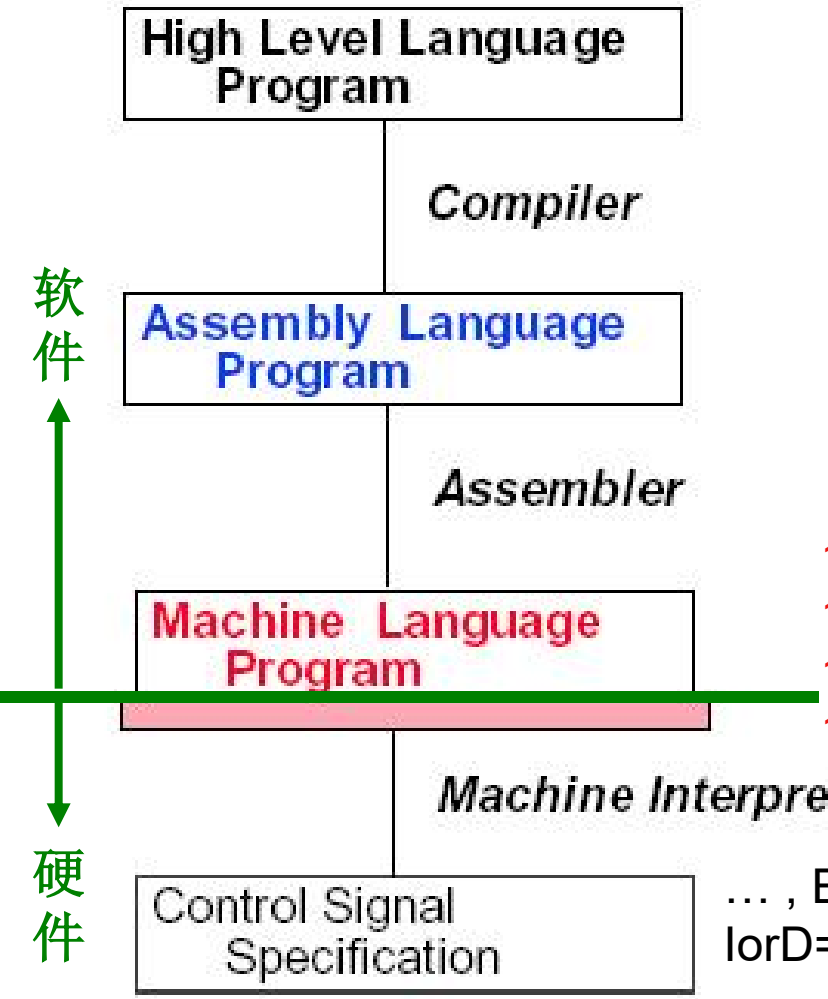
从高级语言程序出发，用其对应的机器级代码以及内存（栈）中信息的变化来说明底层实现

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，属于硬件范畴
- 伪指令是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




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

汇编指令

```
lw $15, 0($2)
lw $16, 4($2)
sw $16, 0($2)
sw $15, 4($2)
```

机器指令

```
1000 1100 0100 1111 0000 0000 0000 0000
1000 1100 0101 0000 0000 0000 0000 0100
1010 1100 0101 0000 0000 0000 0000 0000
1010 1100 0100 1111 0000 0000 0000 0100
```

… , EXTop=1,ALUSelA=1,ALUSelB=11,ALUop=add,
IorD=1,Read,MemtoReg=1,RegWr=1,.....

微指令 → **… 1 1 11 100 1 0 1 1 …**

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。

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

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]←R[cl]

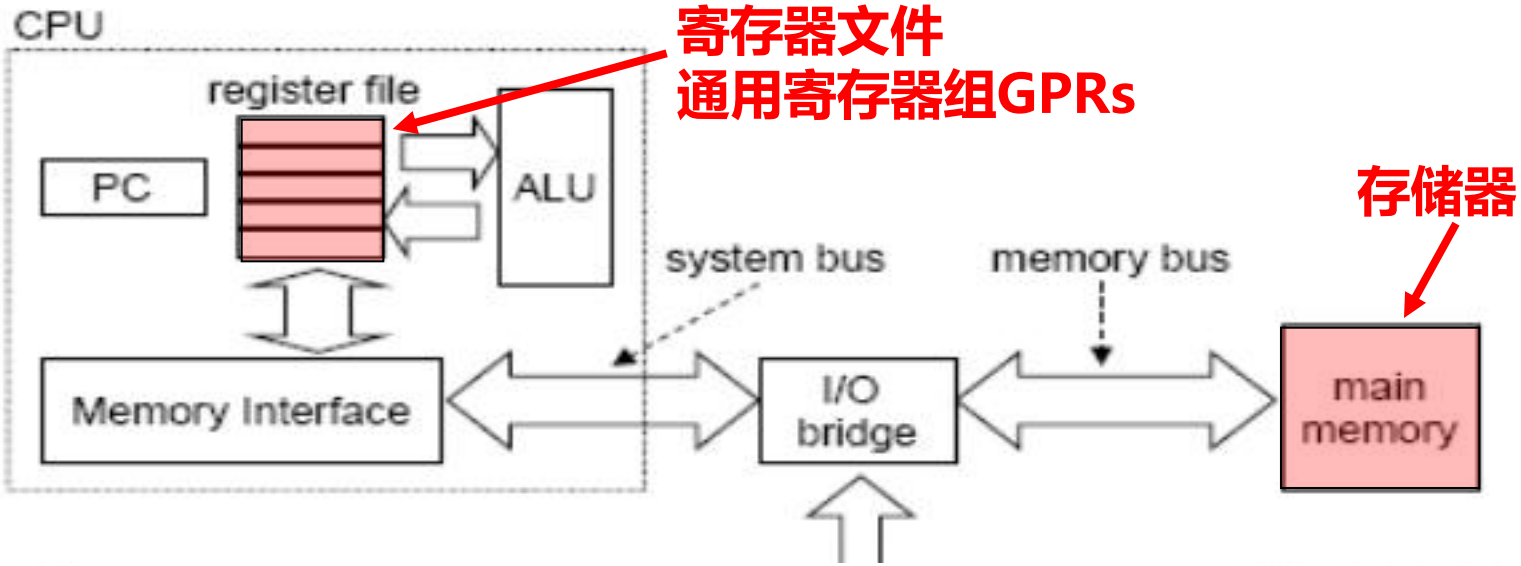
R：寄存器内容

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!

Today: Machine Programming I: Basics

- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code

Our Coverage

- **IA32**
  - The traditional x86

- **x86-64**
  - The standard
  - `bupt1> gcc hello.c`

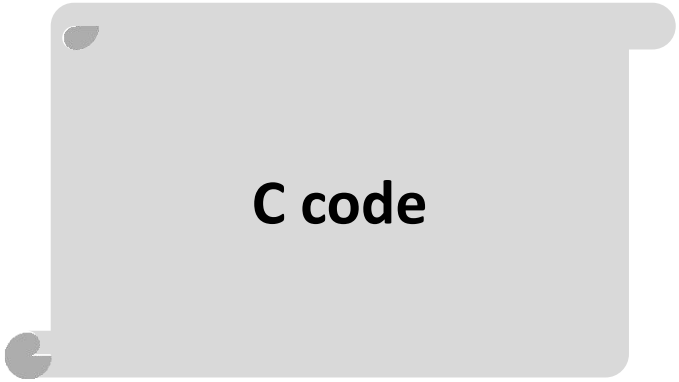
- **Presentation**
  - Book covers x86-64
  - Web aside on IA32
  - We will only cover x86-64

Today: Machine Programming I: Basics

- **Assembly Basics: Registers, operands, move**
- **Arithmetic & logical operations**
- **C, assembly, machine code**

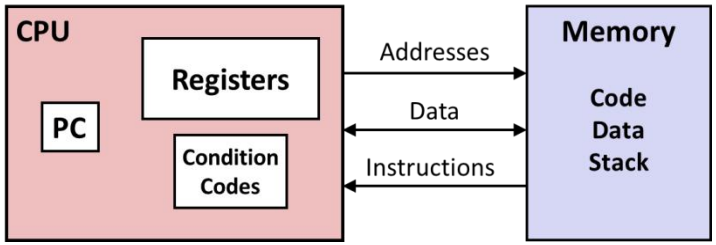
Levels of Abstraction

C programmer



Nice clean layers, but beware…

Assembly programmer



Computer Designer

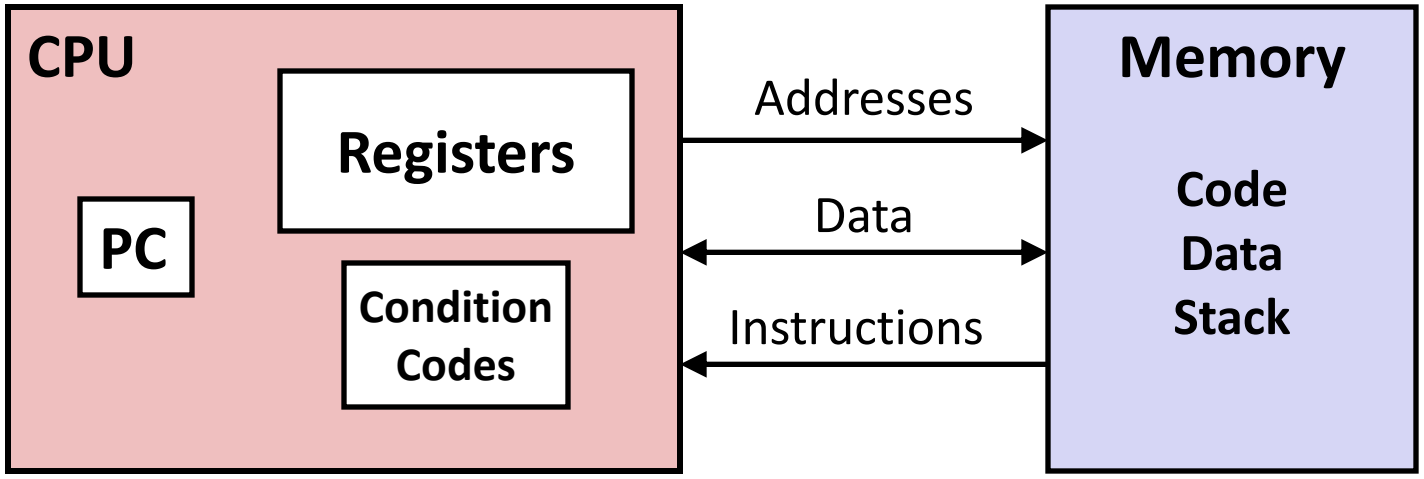
Caches, clock freq, layout, …

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

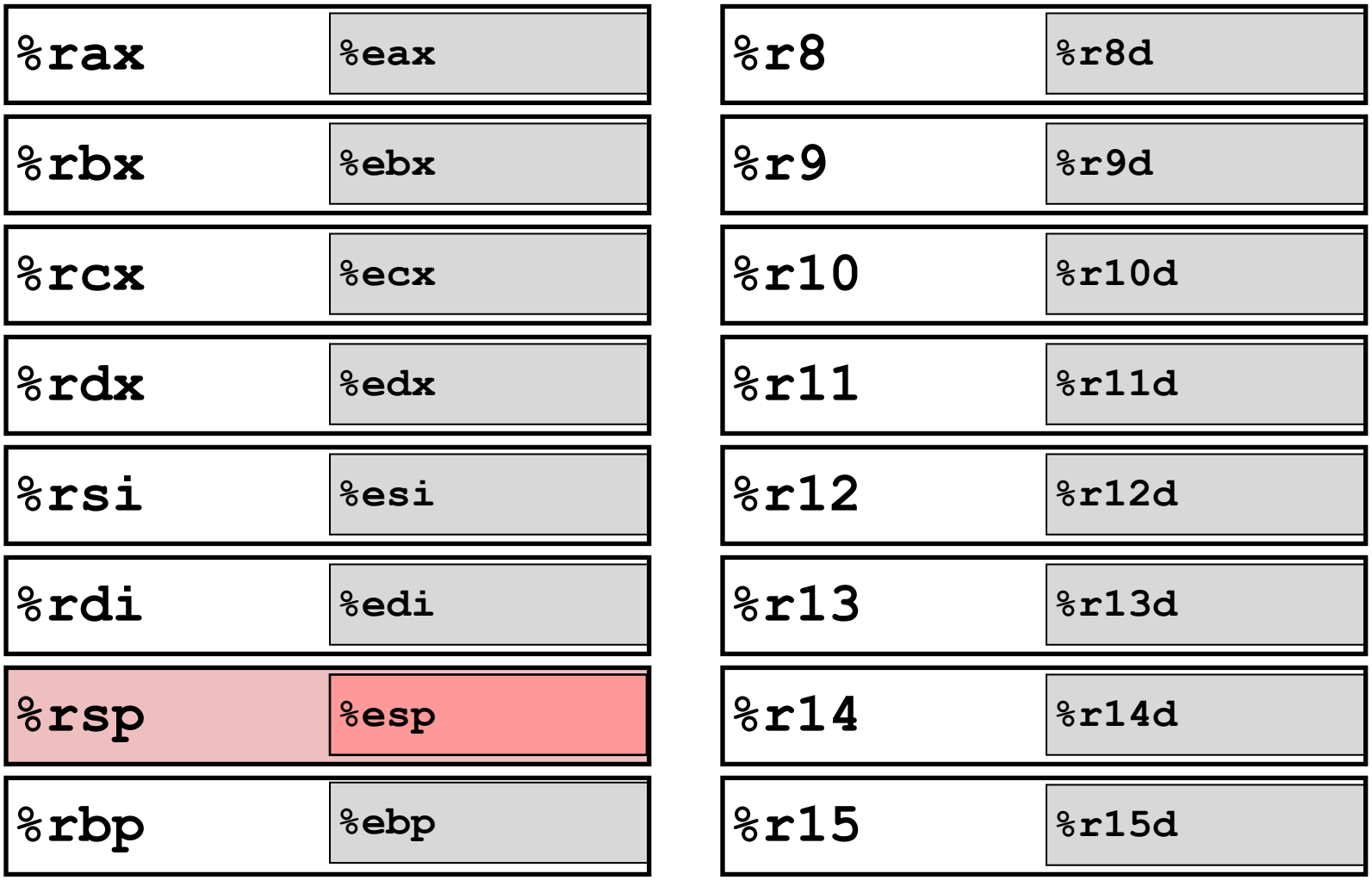
- **PC: Program counter**
  - Address of next instruction
  - Called "RIP" (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

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

Assembly Characteristics: Data Types

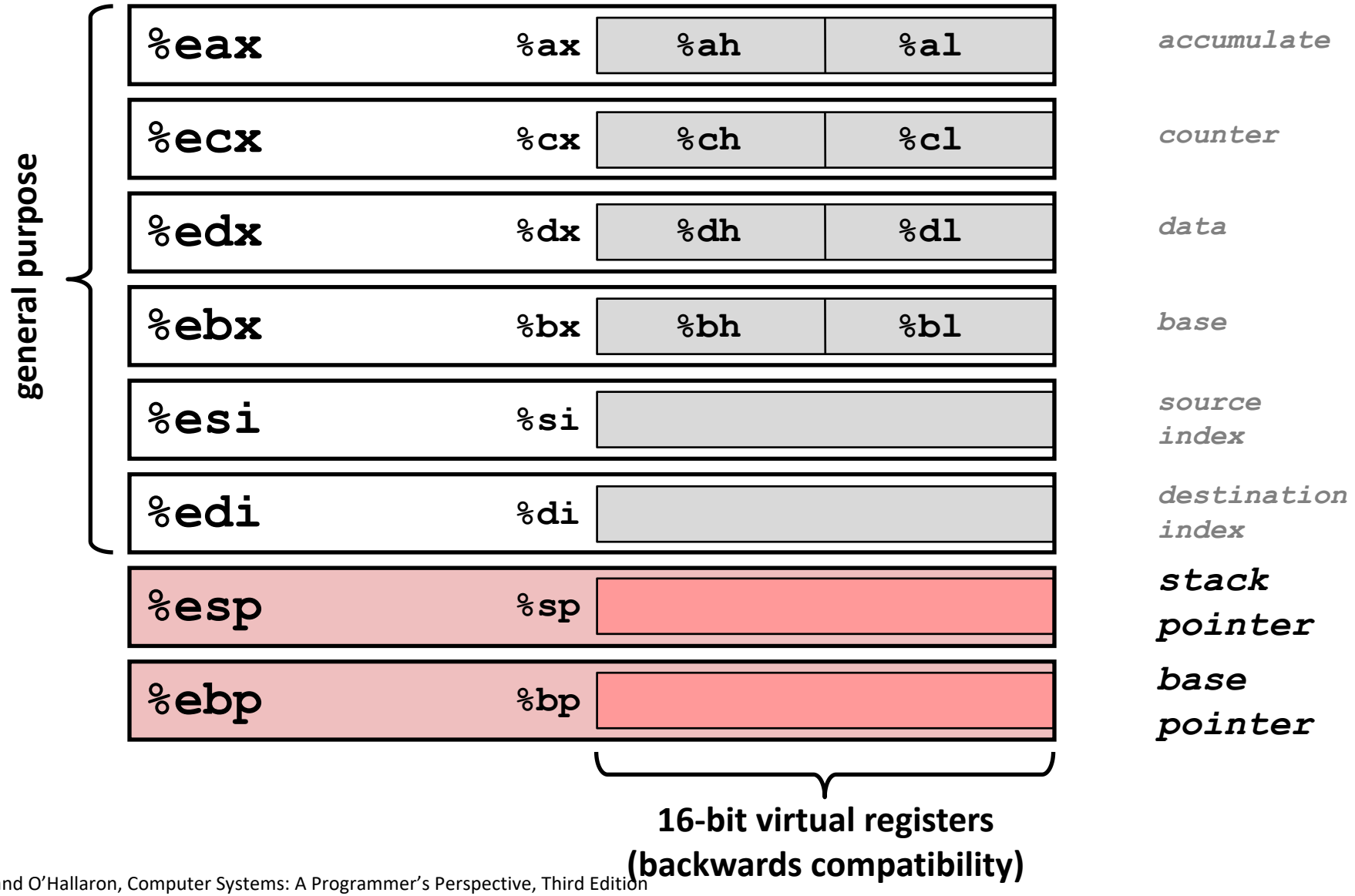
- **"Integer" data of 1, 2, 4, or 8 bytes**
  - Data values
  - Addresses (untyped pointers)
- **Floating point data of 4, 8, or 10 bytes**
- **(SIMD vector data types of 8, 16, 32 or 64 bytes)**
- **Code: Byte sequences encoding series of instructions**
- **No aggregate types such as arrays or structures**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

x86-64 Integer Registers



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

Some History: IA32 Registers



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)<<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)×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)×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+99×4=500

b[i][j]的地址如何计算?

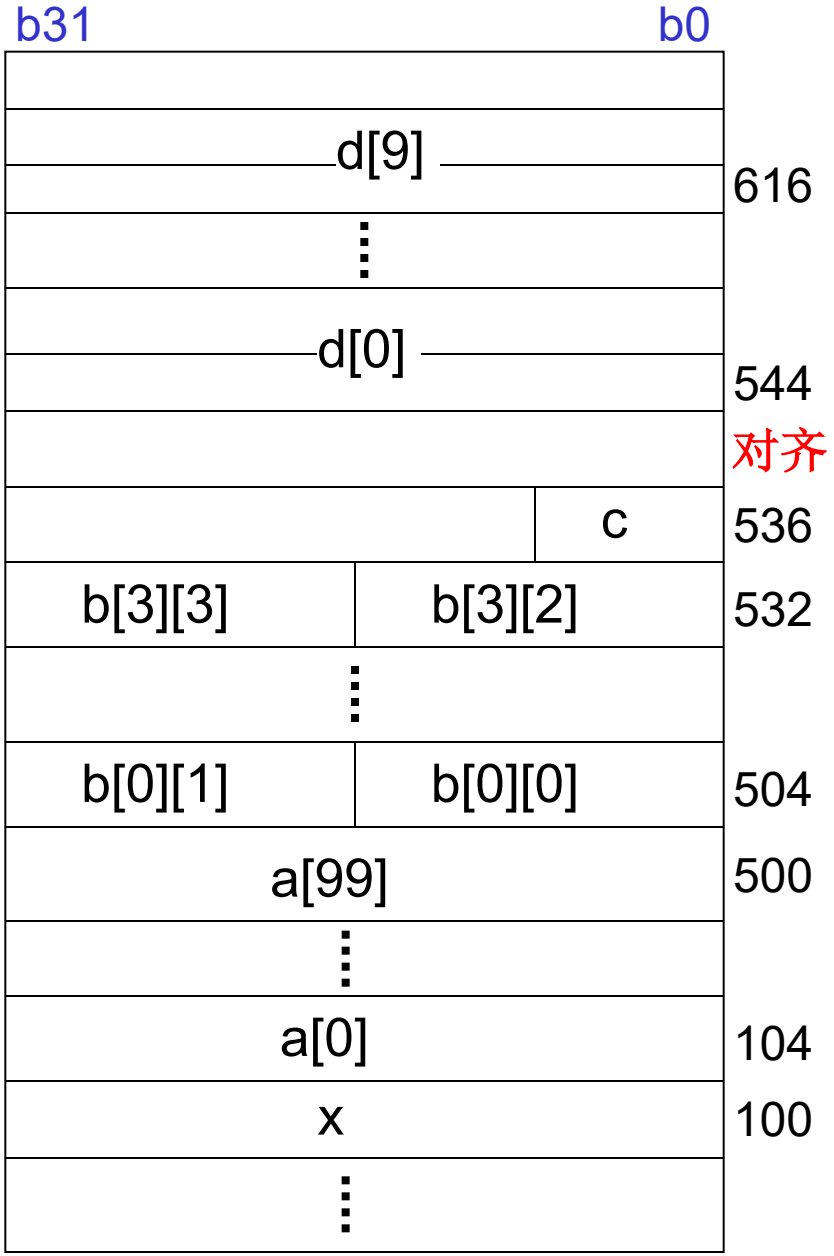
504+i×8+j×2

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+9×8=616



Example Data Representations

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
pointer	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×4，比例变址+位移

d[i]：544+i×8，比例变址+位移

b[i][j]： 504+i×8+j×2，

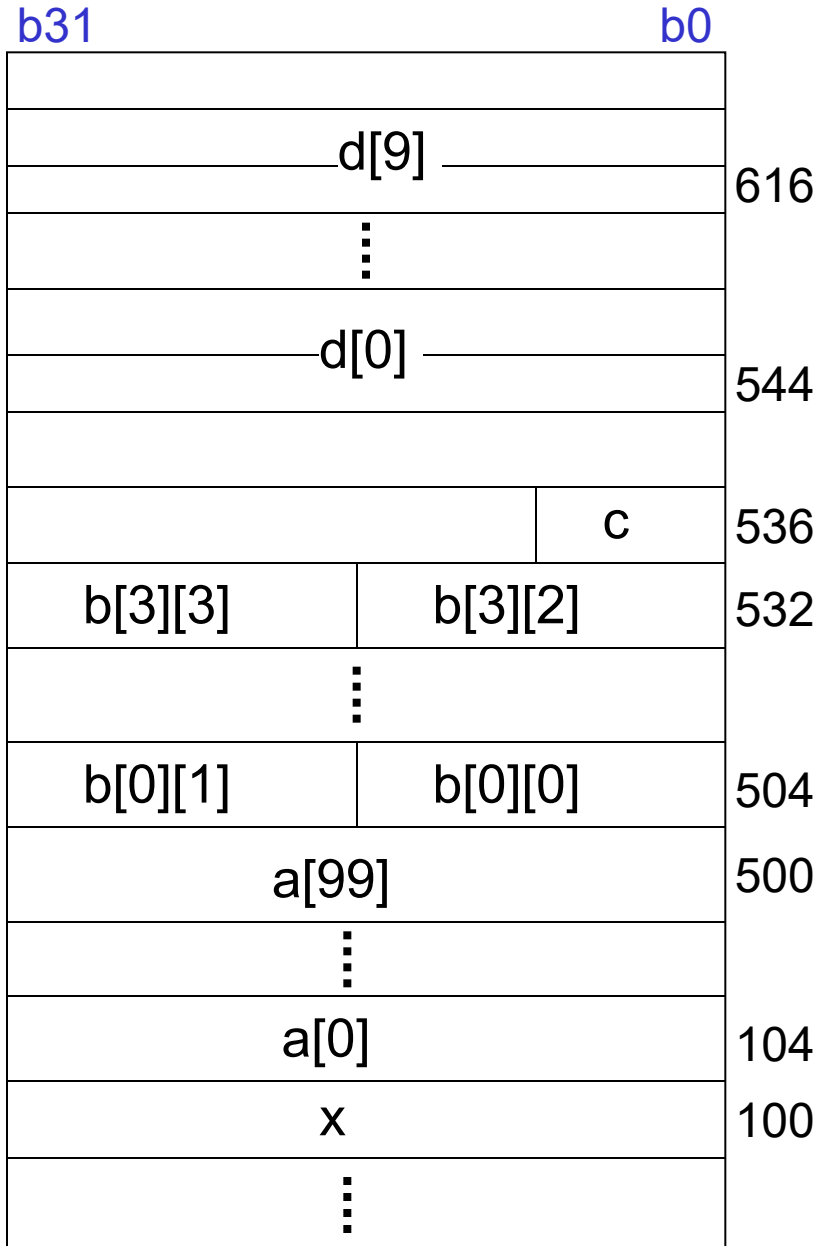
基址+比例变址+位移

将b[i][j]取到EAX中的指令可以是：

“movw 504(%ebp,%esi,2), %eax”

其中， i×8在EBP中，j在ESI中，

2为比例因子



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"

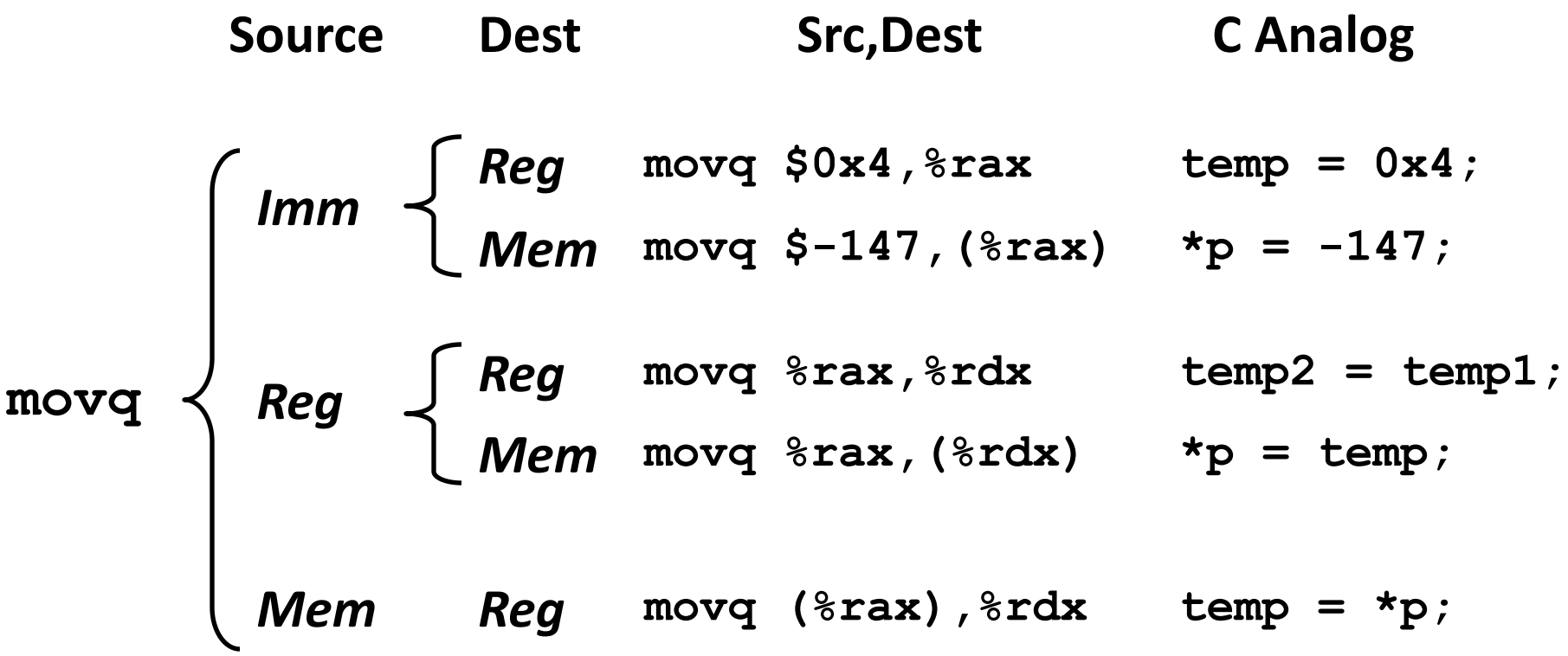
C，C++规定，16进制数必须以 0x开头

`%rax`
`%rcx`
`%rdx`
`%rbx`
`%rsi`
`%rdi`
`%rsp`
`%rbp`

`%rn(n=8~15)`

Warning: Intel docs use mov *Dest, Source*

movq Operand Combinations



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

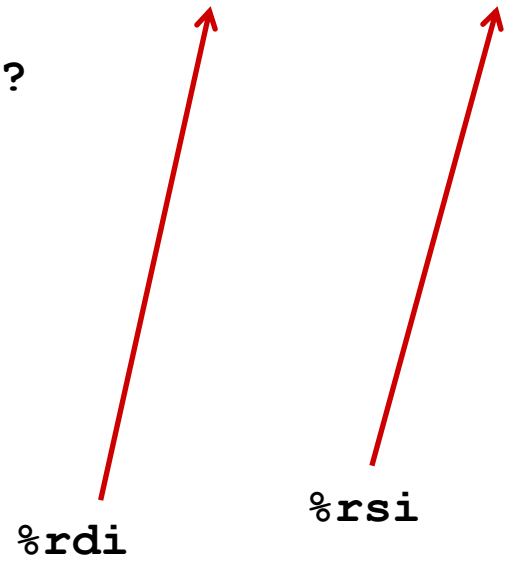
```
movq (%rcx),%rax
```

- **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)
{
      ????
}
```




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

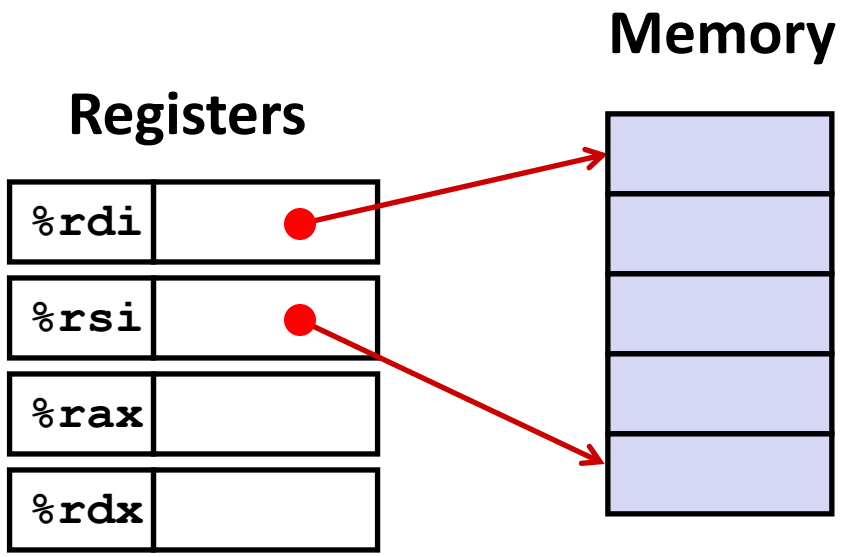
Example of Simple Addressing Modes

```
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()

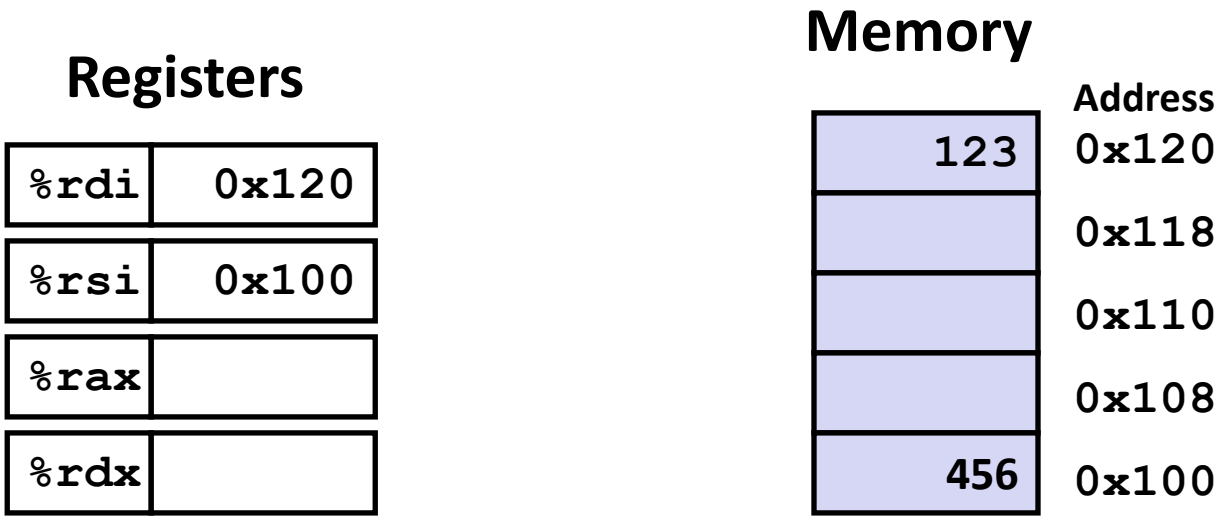
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

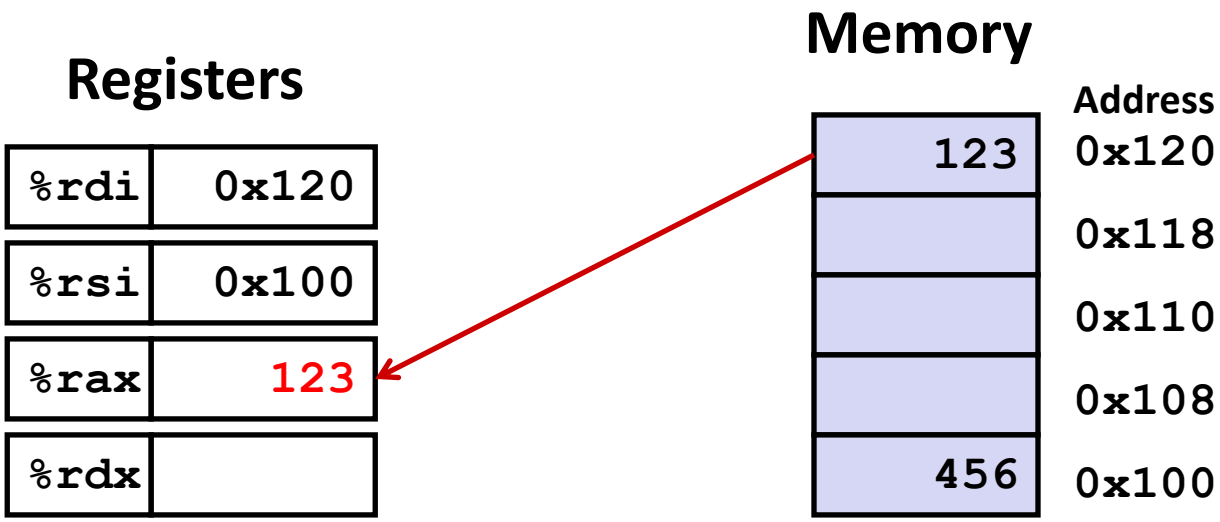
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



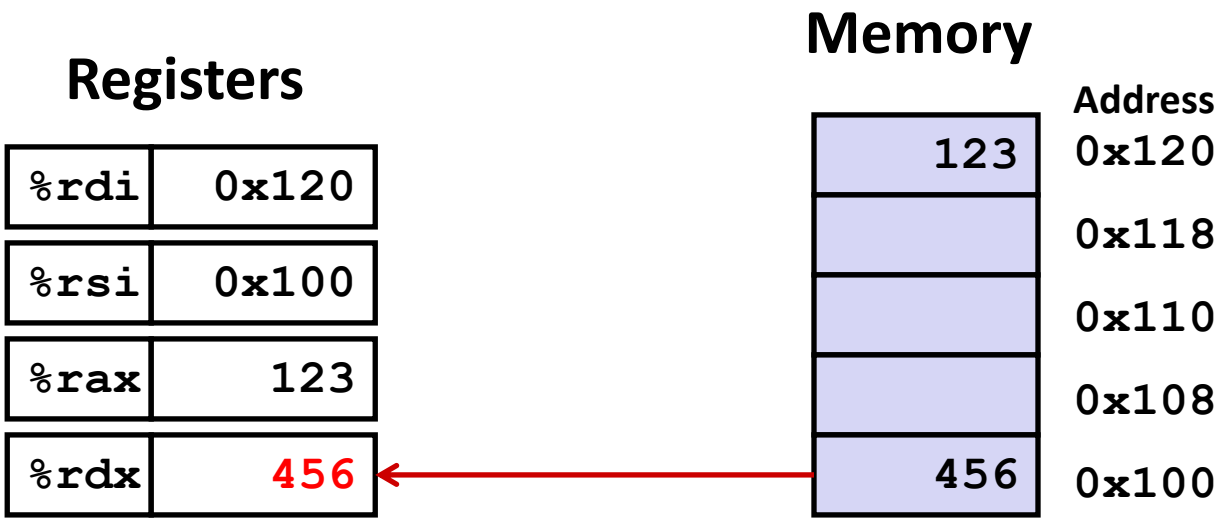

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



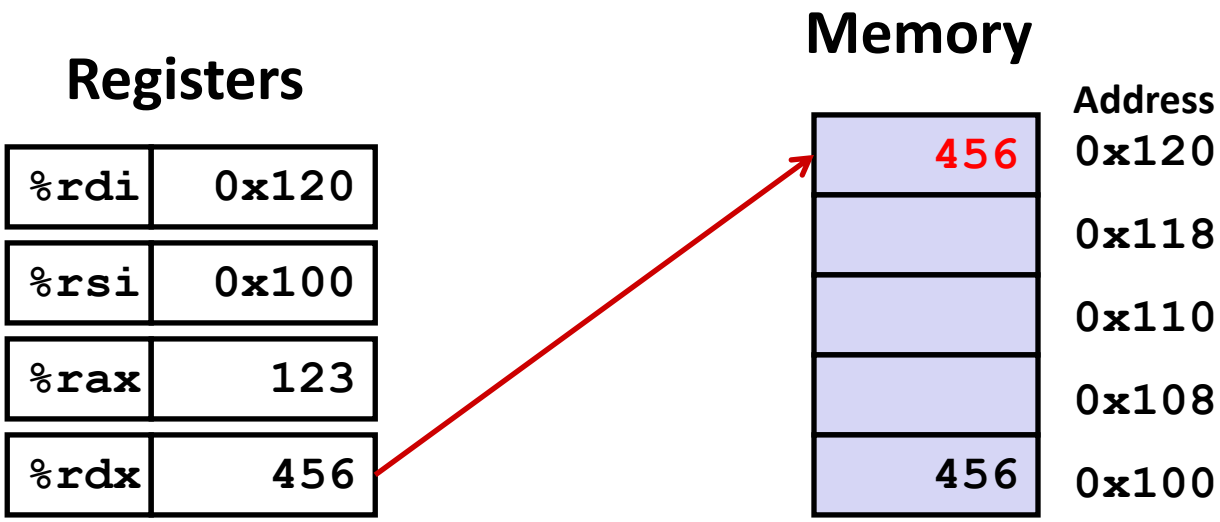

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Understanding Swap()



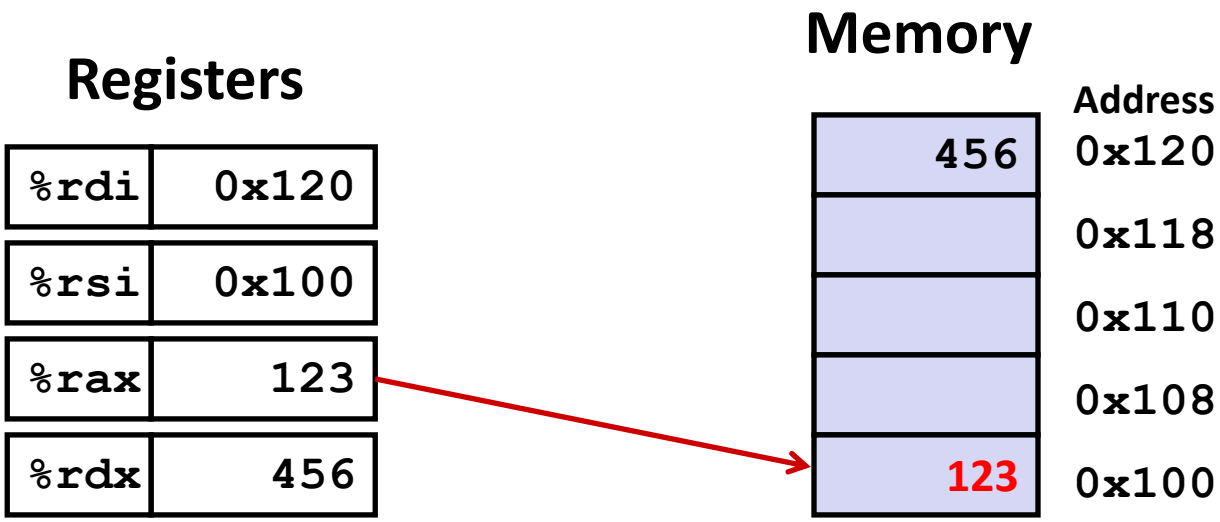

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

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

%rdx	0xf000
%rcx	0x0100

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
0x8(%rdx)		
(%rdx,%rcx)		
(%rdx,%rcx,4)		
0x80(,%rdx,2)		

Today: Machine Programming I: Basics

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

定点算术运算指令汇总（32位）

指令	显式操作数	影响的常用标志	操作数类型	AT&T 指令助记符	对应 C 运算符
ADD	2 个	OF、ZF、SF、CF	无/带符号整数	addb、addw、addl	+
SUB	2 个	OF、ZF、SF、CF	无/带符号整数	subb、subw、subl	−
INC	1 个	OF、ZF、SF	无/带符号整数	incb、incw、incl	++
DEC	1 个	OF、ZF、SF	无/带符号整数	decb、decw、decl	−−
NEG	1 个	OF、ZF、SF、CF	无/带符号整数	negb、negw、negl	−
CMP	2 个	OF、ZF、SF、CF	无/带符号整数	cmpb、cmpw、cmpl	<, <=, >, >=
MUL	1 个	无	无符号整数	mulb、mulw、mull	*
MUL	2 个	无	无符号整数	mulb、mulw、mull	*
MUL	3 个	无	无符号整数	mulb、mulw、mull	*
IMUL	1 个	无	带符号整数	imulb、imulw、imull	*
IMUL	2 个	无	带符号整数	imulb、imulw、imull	*
IMUL	3 个	无	带符号整数	imulb、imulw、imull	*
DIV	1 个	无	无符号整数	divb、divw、divl	/, %
IDIV	1 个	无	带符号整数	idivb、idivw、idivl	/, %

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[eax]←R[edx]+R[eax]，执行前，若R[edx]=*i*,
  R[eax]=*j*，则指令执行后，R[eax]=*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

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, 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
```

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

定点加法指令举例

- 假设R[ax]=FFFAH，R[bx]=FFF0H，则执行Intel格式指令“add ax, bx”后，AX、BX中的内容各是什么？标志CF、OF、ZF、SF各是什么？要求分别将操作数作为无符号数和带符号整数解释并验证指令执行结果。

解：功能：R[ax]←R[ax]+R[bx]，指令执行后的结果如下

R[ax]=FFFAH+FFF0H=FFEAH ，BX中内容不变

CF=1，OF=0，ZF=0，SF=1

若是无符号整数运算，则CF=1说明结果溢出

验证：FFFA的真值为65535-5=65530，FFF0的真值为65520

FFEA的真值为65535-21=65514≠65530+65520，即溢出

若是带符号整数运算，则OF=0说明结果没有溢出

验证：FFFA的真值为-6，FFF0的真值为-16

FFEA的真值为-22=-6+(-16)，结果正确，无溢出

整数乘除指令

- 乘法指令：可给出一个、两个或三个操作数
  - 若给出一个操作数SRC，则另一个源操作数隐含在AL/AX/EAX中，将SRC和累加器内容相乘，结果存放在AX（16位）或DX-AX（32位）或EDX-EAX（64位）中。DX-AX表示32位乘积的高、低16位分别在DX和AX中。
  - 若指令中给出两个操作数DST和SRC，则将DST和SRC相乘，结果在DST中。
  - 若指令中给出三个操作数REG、SRC和IMM，则将SRC和立即数IMM相乘，结果在REG中。
- 除法指令：只明显指出除数，用EDX-EAX中内容除以指定的除数
  - 若为8位，则16位被除数在AX寄存器中，商送回AL，余数在AH
  - 若为16位，则32位被除数在DX-AX寄存器中，商送回AX，余数在DX
  - 若为32位，则被除数在EDX-EAX寄存器中，商送EAX，余数在EDX

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。 BACK

定点乘法指令举例

- 假设R[eax]=000000B4H，R[ebx]=00000011H，M[000000F8H]=000000A0H，请问：

(1) 执行指令“mulb %bl”后，哪些寄存器的内容会发生变化？是否与执行“imulb %bl”指令所发生的变化一样？为什么？请用该例给出的数据验证你的结论。

解：“mulb %bl”功能为 R[ax]←R[al]×R[bl]，执行结果如下

R[ax]=B4H*11H（无符号整数180和17相乘）

R[ax]=0BF4H，真值为3060=180 × 17

```
        1011 0100
   x    0001 0001
-------------------
        1011 0100
   1011 0100
-------------------
0000 1011 1111 0100
```

AH=?　　AL=?

“imulb %bl”功能为 R[ax]←R[al]×R[bl]

R[ax]=B4H*11H（带符号整数-76和17相乘）

若R[ax]=0BF4H，则真值为3060≠-76 × 17

R[al]=F4H, R[ah]=? AH中的变化不一样!

R[ax]=FAF4H, 真值为-1292=-76 × 17

对于带符号乘，若积只取低n位，则和无符号相同；若取2n位，则采用“布斯”乘法

定点乘法指令举例

- 假设R[eax]=000000B4H，R[ebx]=00000011H，M[000000F8H]=000000A0H，请问：

(2) 执行指令“imull $-16, (%eax,%ebx,4), %eax”后哪些寄存器和存储单元发生了变化？乘积的机器数和真值各是多少？

解：“imull -16, (%eax,%ebx,4),%eax”

功能为 R[eax]←(-16)×M[R[eax]+R[ebx]×4]，执行结果如下

R[eax]+R[ebx]×4=000000B4H+00000011H<<2=000000F8H

R[eax]=(-16)×M[000000F8H]

=(-16)× 000000A0H（带符号整数乘）

=FFFFFF60H<<4

=FFFFF600H

EAX中的真值为-2560

SKIP

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。

按位运算指令举例

假设short型变量x被编译器分配在寄存器AX中，R[ax]=FF80H，则以下汇编代码段执行后变量x的机器数和真值分别是多少?

```
movw %ax, %dx
salw   $2, %ax      1111 1111 1000 0000<<2
addl   %dx, %ax     1111 1111 1000 0000+1111 1110 0000 0000
sarw   $1, %ax      1111 1101 1000 0000>>1=1111 1110 1100 0000
```

解：$2和$1分别表示立即数2和1 。

x是short型变量，故都是算术移位指令，并进行带符号整数加。

假设上述代码段执行前R[ax]=x，则执行((x<<2)+x)>>1后，R[ax]=5x/2。算术左移时，AX中的内容在移位前、后符号未发生变化，故OF=0，没有溢出。最终AX的内容为FEC0H，解释为short型整数时，其值为-320。验证：x=-128，5x/2=-320。经验证，结果正确。

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）

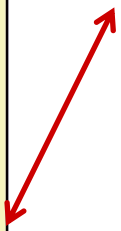
  CALL DST：返回地址RA入栈，转DST处执行

  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。

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

作业

- 练习：3.1、3.2、3.3、3.4、3.6
- 作业：3.58、3.60