

# 计算机系统

## 程序的机器级表示：基本

湖南大学

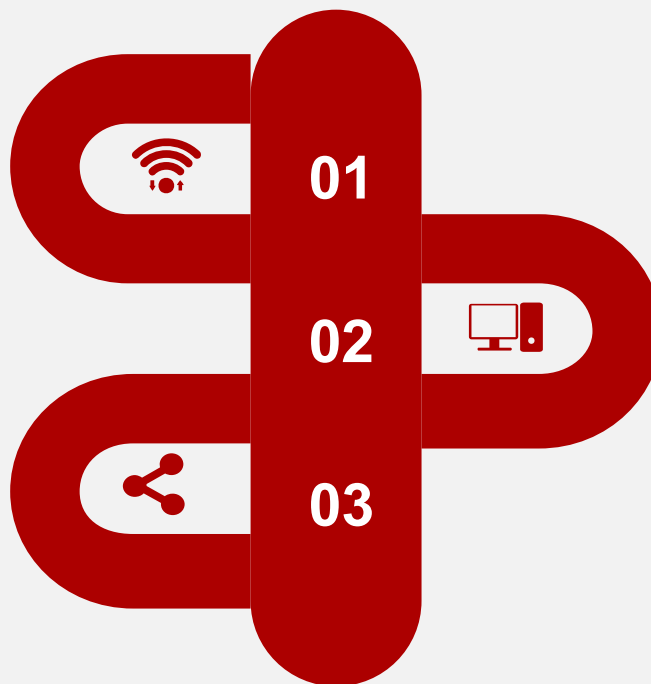
《计算机系统》课程教学组



## 内容提要

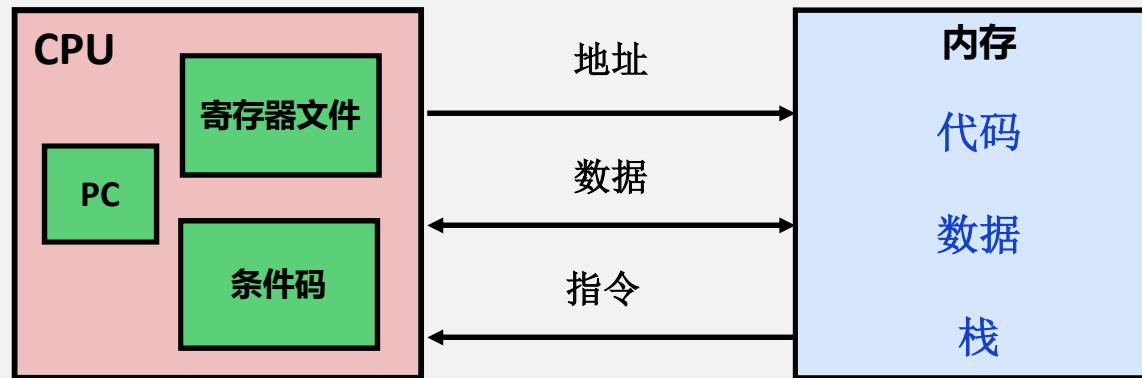
从C代码  
到机器代码

算术操作



数据传送与  
寻址方式

## 程序执行



### • CPU

#### • PC: 程序计数器

- 下一条指令的地址
- 记为 “EIP” (IA32) or “RIP” (x86-64)

#### • 寄存器文件

- 一组寄存器

#### • Condition codes

- 条件码是CPU根据运算结果由硬件设置的位，体现当前指令执行结果的各种状态信息
- 是程序分支和程序循环的依据

### • Condition codes

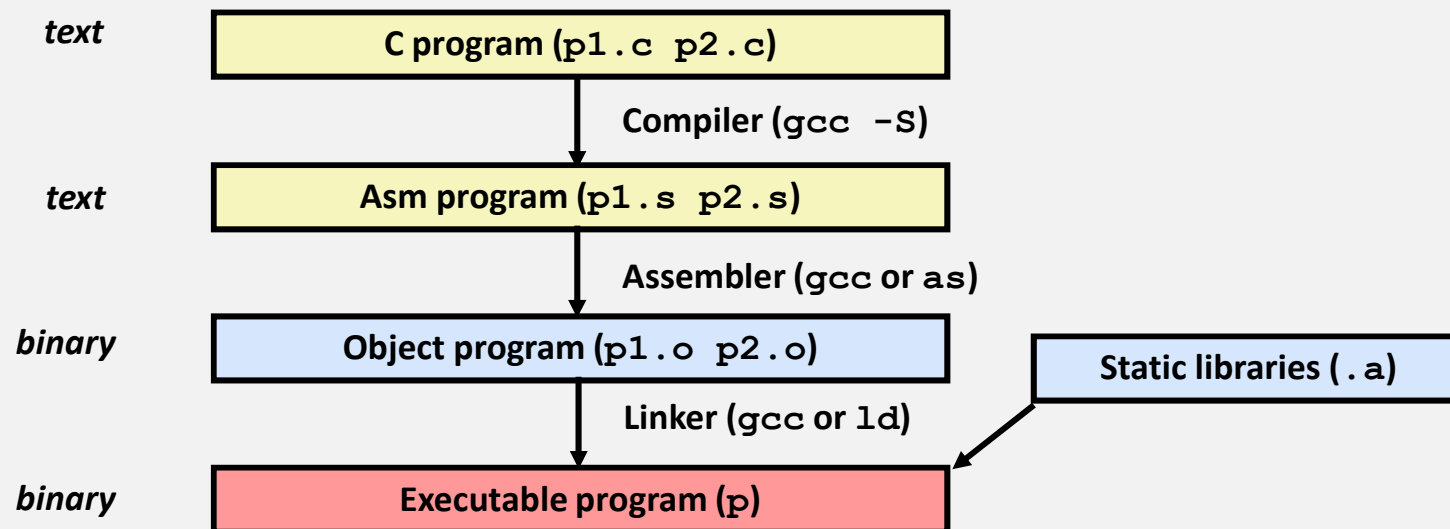
- 条件码是CPU根据运算结果由硬件设置的位，体现当前指令执行结果的各种状态信息
- 是程序分支和程序循环的依据

### • 内存

- 字节数组
- 代码与用户数据
- 支持过程的栈

## 从C到 目标代码

两个源代码文件 `p1.c` 和 `p2.c`，通过编译指令 `gcc -O1 p1.c p2.c -o p` 进行编译，使用最基本的代码优化 (`-O1`)，二进制代码文件 `p`。



## 编译系统

- **编译系统将高级语言变成机器指令**
  - 不仅仅是翻译，还有语法检查、优化、链接等等
  - 编译系统的重要性不亚于操作系统
- **中国独立自主的高性能编译系统**
  - **神威“太湖之光”超级计算机**  
申威64指令集、神威睿智编译器及其工具链、  
神威睿思操作系统.....
  - 中国超算凭借独立成果多次占据世界超算榜首。

- **中国独立自主的桌面操作系统**



廖湘科院士



陈左宁院士



倪光南院士

## 汇编代码

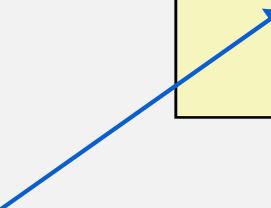
### C Code

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

### Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    popl %ebp
    ret
```

某些编译器使用指令  
"leave"  
来打包出栈指令



## 机器代码

```
int t = x+y;
```



C Code

两个有符号整数相加

```
Addl 8(%ebp), %eax
```



汇编

相似表达式:

```
x+= y
```

更精确的表达:

```
int eax;
```

```
int *ebp;
```

```
eax += ebp[8]
```

- 将两个4字节整数相加
  - 有符号或无符号都是同一指令
- 操作数:
  - **y**: 寄存器            **%eax**
  - **x**: 内存                **M[%ebp+8]**
  - **t**: 寄存器            **%eax**
- 返回值保存在 **%eax**

```
0x80483ca: 03 45 08
```



目标代码

- 3-字节指令
- 保存在地址 **0x80483ca**

## 目标代码

### Code for sum

0x401040 <sum>:

0x55  
0x89  
0xe5  
0x8b  
0x45  
0x0c  
0x03  
0x45  
0x08  
0x5d  
0xc3

目标代码  
code.o

- 一共有 11 个字节
- 每条指令占 1, 2, or 3 个字节
- 起始地址在 0x0401040

使用命令:

**gcc -O1 -c code.c -o code.o**

得到文件: code.o

#### • 汇编器

- 将 .s 文件转成 .o 文件
- 每条指令都编码
- 不同文件之间的联系没有体现

#### • 链接

- 解决文件之间的引用关系
- 与静态链接库整合
  - e.g., code for **malloc**, **printf**
- 有些库是动态链接的 ( *dynamically linked* )
  - 当程序开始执行时才链接

起始地址是动态分配的



## 反汇编

0x401040 <sum>:

0x401040:	55	push	%ebp
0x401041:	89 e5	mov	%esp, %ebp
0x401043:	8b 45 0c	mov	0xc(%ebp), %eax
0x401046:	03 45 08	add	0x8(%ebp), %eax
0x401049:	5d	pop	%ebp
0x40104a:	c3	ret	

反汇编器（**Disassembler**）：将目标代码编译回汇编代码

**objdump -d code.o**

分析目标代码时的有效工具

## 反汇编

### 目标代码

```
0x401040:  
0x55  
0x89  
0xe5  
0x8b  
0x45  
0x0c  
0x03  
0x45  
0x08  
0x5d  
0xc3
```



Dump of assembler code for function sum:

```
0x080483c4 <sum+0>:      push    %ebp  
0x080483c5 <sum+1>:      mov     %esp,%ebp  
0x080483c7 <sum+3>:      mov     0xc(%ebp),%eax  
0x080483ca <sum+6>:      add     0x8(%ebp),%eax  
0x080483cd <sum+9>:      pop     %ebp  
0x080483ce <sum+10>:     ret
```

### 另一种反汇编方式

使用GDB

**gdb p**

**disassemble sum**

**x/11xb sum** 查看从sum函数地址开始的11个字节

## test.s 与 反汇编代码

### 反汇编目标代码的结果

```
1 // test.c
2 #include <stdio.h>
3 int add(int i, int j)
4 {
5     int x = i + j;
6     return x;
7 }
```

gcc -E test.c -o test.i

gcc -S test.i -o test.s

or gcc -S test.c -o test.s

test.s

```
add:
pushl   %ebp
movl    %esp, %ebp
subl    $16, %esp
movl    12(%ebp), %eax
movl    8(%ebp), %edx
leal    (%edx, %eax), %eax
movl    %eax, -4(%ebp)
movl    -4(%ebp), %eax
leave
ret
```

00000000 <add>:

0:	55	push %ebp
1:	89 e5	mov %esp, %ebp
3:	83 ec 10	sub \$0x10, %esp
6:	8b 45 0c	mov 0xc(%ebp), %eax
9:	8b 55 08	mov 0x8(%ebp), %edx
c:	8d 04 02	lea (%edx,%eax,1), %eax
f:	89 45 fc	mov %eax, -0x4(%ebp)
12:	8b 45 fc	mov -0x4(%ebp), %eax
15:	c9	leave
16:	c3	ret

位移量 机器指令

汇编指令

## 两种目标文件 反汇编对比

```
1 // test.c
2 #include <stdio.h>
3 int add(int i, int j )
4 {
5     int x = i +j;
6     return x;
7 }
```

objdump -d test.o

反汇编 test.o 文件

00000000 <add>:

0:	55	push	%ebp
1:	89 e5	mov	%esp, %ebp
3:	83 ec 10	sub	\$0x10, %esp
6:	8b 45 0c	mov	0xc(%ebp), %eax
9:	8b 55 08	mov	0x8(%ebp), %edx
c:	8d 04 02	lea	(%edx,%eax,1), %eax
f:	89 45 fc	mov	%eax, -0x4(%ebp)
12:	8b 45 fc	mov	-0x4(%ebp), %eax
15:	c9	leave	
16:	c3	ret	

objdump -d test

反汇编可执行目标文件 "test"

080483d4 <add>:

80483d4:	55	push ...
80483d5:	89 e5	...
80483d7:	83 ec 10	...
80483da:	8b 45 0c	...
80483dd:	8b 55 08	...
80483e0:	8d 04 02	...
80483e3:	89 45 fc	...
80483e6:	8b 45 fc	...
80483e9:	c9	...
80483ea:	c3	ret

内存地址

机器指令

汇编指令

## 课堂思考

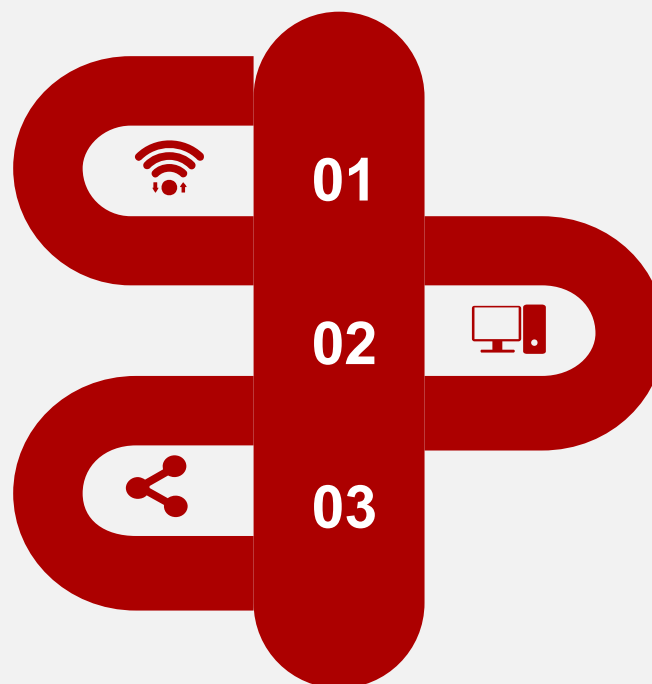


**从目标文件到可执行目标文件，主要的变化有哪些？**

## 内容提要

从C代码  
到机器代码

算术操作



数据传送与  
寻址方式

## 基本数据类型

- “整数” 1, 2, 4, 8 bytes
  - 数据值
  - 地址 (**untyped pointers**)—认真学习课程中心关于指针的讲课视频!
- 浮点数 4, 8, or 10 bytes
  - 单精度 (**float: 4 bytes**)
  - 双精度 (**double: 8 bytes**)
  - 长双精度 (**long double or extended: 10 bytes**)
- 数组与结构
  - 内存中一组连续分配的字节

## 基本操作

- 对寄存器或内存数据进行操作的运算类指令
- 在内存与寄存器中之间传送数据的传送类指令
  - 将数据从内存加载到寄存器中
  - 将寄存器数据保存到内存中
- 决定程序走向的控制类指令
  - 无条件/有条件跳转
  - 分支/循环



## IA32寄存器

通用寄存器

%eax	%ax	%ah	%al
%ecx	%cx	%ch	%cl
%edx	%dx	%dh	%dl
%ebx	%bx	%bh	%bl
%esi	%si		
%edi	%di		
%esp	%sp		
%ebp	%bp		

*accumulate*

*counter*

*data*

*base*

*source  
index*

*destination  
index*

*stack  
pointer*

*base  
pointer*

16-bit 虚拟寄存器  
(为了向下兼容)

Origin  
(mostly obsolete)

## 传送数据 IA32

- **MOV 指令**

- `movl Source, Dest:`

- **操作数类型**

- **Immediate: 立即数**

- 例: `$0x400`, `$-533`
    - 占用 1, 2, or 4 字节

- **Register: 8个整数寄存器之一**

- 例: `%eax`, `%edx`
    - `%esp`与`%ebp`保留作为特殊用途
    - 其他通用寄存器可能会在某些操作时有特定用途

- **Memory: 内存地址**

- 简单例子: (`%eax`)
    - 有非常多的寻址模式

<code>%eax</code>
<code>%ecx</code>
<code>%edx</code>
<code>%ebx</code>
<code>%esi</code>
<code>%edi</code>
<code>%esp</code>
<code>%ebp</code>

## 操作数组合

	Source	Dest	Src,	Dest	C Analog
movl	Imm	Reg	movl <b>\$0x4</b> , %eax		temp = <b>0x4</b> ;
		Mem	movl <b>\$-147</b> , (%eax)		*p = <b>-147</b> ;
	Reg	Reg	movl %eax, %edx		temp2 = temp1;
		Mem	movl %eax, (%edx)		*p = temp;
	Mem	Reg	movl (%eax), %edx		temp = *p;

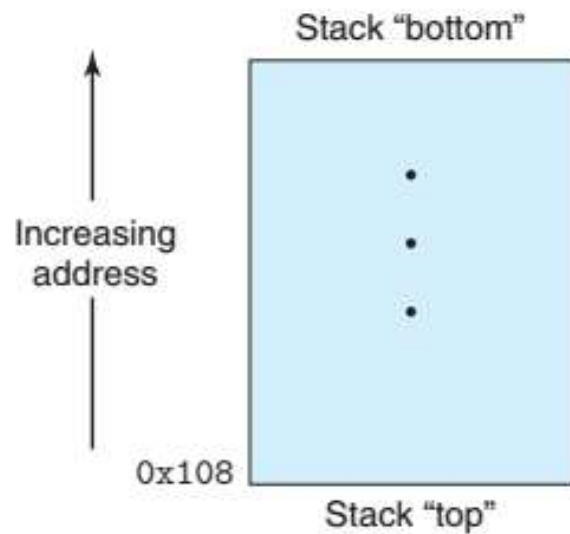
没有内存到内存的数据传送指令

## push 与 pop

数据为int

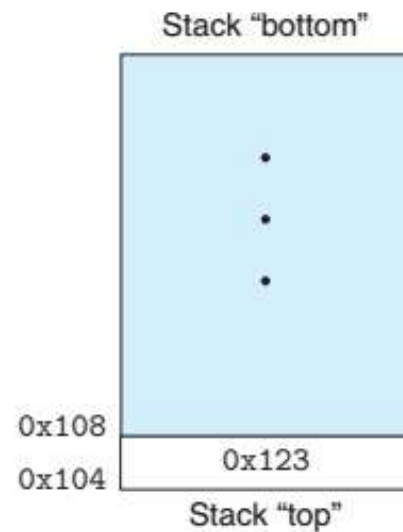
Initially

%eax	0x123
%edx	0
%esp	0x108



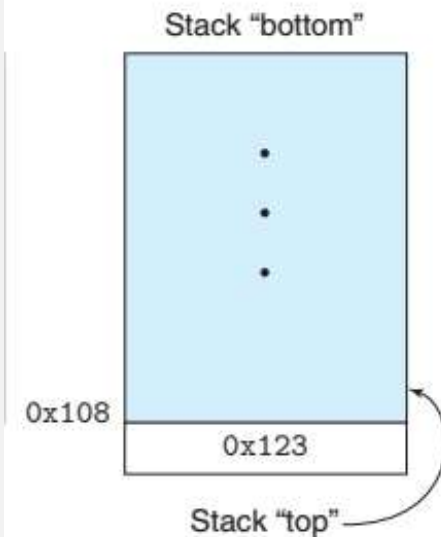
pushl %eax

%eax	0x123
%edx	0
%esp	0x104



popl %edx

%eax	0x123
%edx	0x123
%esp	0x108



## 汇编格式

### ATT格式

`movl (%ecx), %eax`

### Intel格式

`mov eax, [ecx]`

- 区别:

- (1) Intel格式的指令助记符省略了指示数据大小的**后缀**;
- (2) Intel格式省略了寄存器名字前的**%**;
- (3) Intel格式指令中, 源操作数在右边, 目的操作数在左边。 (**与ATT正好相反**)

## 内存寻址

### 寄存器间接寻址(R)

寄存器R中存放了内存地址

$\text{Mem}[\text{Reg}[\text{R}]]$

`movl (%ecx),%eax`

### 基址变址寻址 $\text{D(R)} \quad \text{Mem}[\text{Reg}[\text{R}] + \text{D}]$

寄存器R中存放了内存的起始地址

常数 D 给出了偏移量

`movl 8(%ebp),%edx`

## 内存寻址

Type	Form	Operand value	Name
Immediate	$\$Imm$	$Imm$	Immediate
Register	$E_a$	$R[E_a]$	Register
Memory	$Imm$	$M[Imm]$	Absolute
Memory	$(E_a)$	$M[R[E_a]]$	Indirect
Memory	$Imm(E_b)$	$M[Imm + R[E_b]]$	Base + displacement
Memory	$(E_b, E_i)$	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$Imm(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(, E_i, s)$	$M[Imm + R[E_i] \cdot s]$	Scaled indexed
Memory	$(E_b, E_i, s)$	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

$$EA = Imm + [E_b + E_i * s]$$

**Imm**: 常数 1, 2, or 4 bytes (偏移量)     **$E_b$** : 基址寄存器

**$E_i$** : 变址寄存器 (不要用 %esp)

**$s$** : 比例因子 1, 2, 4, or 8 (why these numbers?)

## 内存寻址

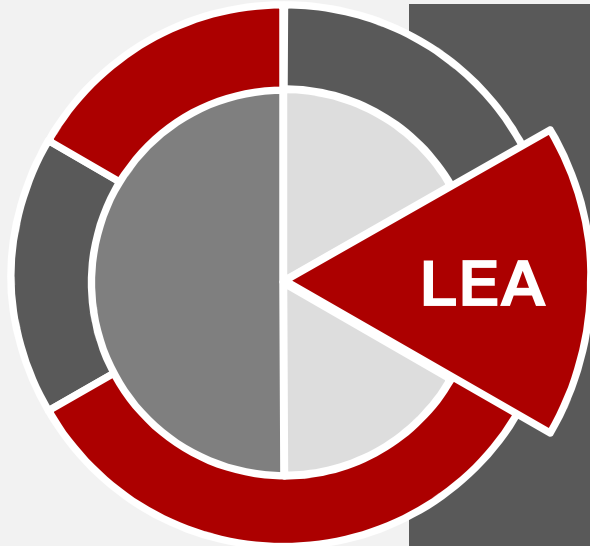
### 地址计算举例

<code>%edx</code>	<code>0xf000</code>
<code>%ecx</code>	<code>0x0100</code>

Expression	Address Computation	Address
<code>0x8(%edx)</code>	<code>0xf000 + 0x8</code>	<code>0xf008</code>
<code>(%edx,%ecx)</code>	<code>0xf000 + 0x100</code>	<code>0xf100</code>
<code>(%edx,%ecx,4)</code>	<code>0xf000 + 4*0x100</code>	<code>0xf400</code>
<code>0x80(,%edx,2)</code>	<code>2*0xf000 + 0x80</code>	<code>0x1e080</code>



## lea 寻址



格式: **leal Src, Dest**

Src : 地址计算表达式

Src的结果保存在Dest中

用途:

- 计算内存地址值本身 (不取内存里的值)  
e.g., translation of  $p = \&x[i]$ ;
- 计算诸如  $x + k*y$  表达式的值  
 $k = 1, 2, 4, \text{ or } 8$

### 举例

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



```
leal (%eax,%eax,2), %eax ;t <- x+x*2
sall $2, %eax             ;return t<<2
```

## 课堂习题 1

参看下图的寄存器信息和存储器信息，若操作数为0xF4( , %edx, 4), 则该寻址方式获得的值为：

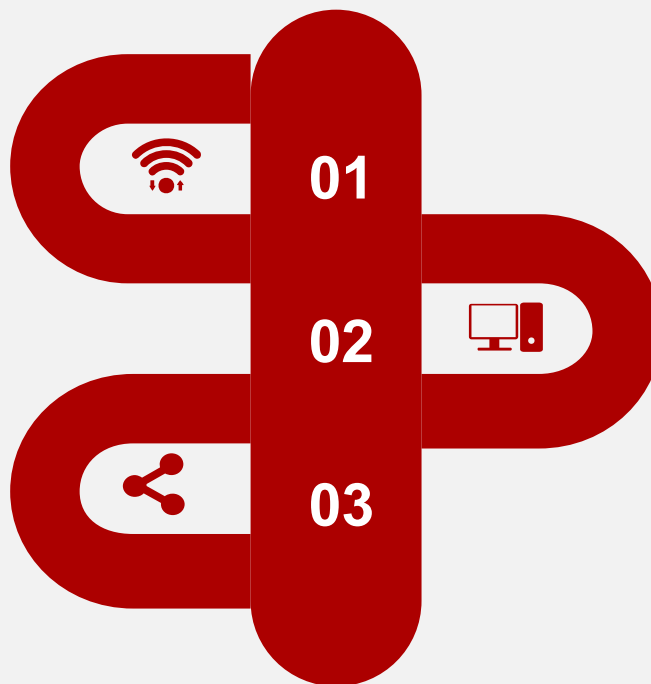
- A. 0XAB
- B. 0X13
- C. 0XFF
- D. 表中无正确值

Address	Value	Register	Value
0x100	0xFF	%eax	0x100
0x104	0xAB	%ecx	0x1
0x108	0x13	%edx	0x3
0x10C	0x11		

## 内容提要

从C代码  
到机器代码

算术操作



数据传送与  
寻址方式

## 算术操作指令

- 双操作数指令:

格式	计算
<b>addl</b> Src, Dest	Dest = Dest + Src
<b>subl</b> Src, Dest	Dest = Dest - Src
<b>imull</b> Src, Dest	Dest = Dest * Src
<b>sall</b> Src, Dest	Dest = Dest << Src (Arithmetic)
<b>shll</b> Src, Dest	Dest = Dest << Src (Logical)
<b>sarl</b> Src, Dest	Dest = Dest >> Src (Arithmetic)
<b>shrl</b> Src, Dest	Dest = Dest >> Src (Logical)
<b>xorl</b> Src, Dest	Dest = Dest ^ Src
<b>andl</b> Src, Dest	Dest = Dest & Src
<b>orl</b> Src, Dest	Dest = Dest   Src

注意操作数的顺序!

## 算术操作指令

- 单操作数指令:

### 格式

**incl** Dest

**decl** Dest

**negl** Dest

**notl** Dest

### 计算

Dest = Dest + 1

Dest = Dest - 1

Dest = -Dest (取补=各位取反后+1)

Dest = ~Dest (取反=各位取反)

## 算术指令示例

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

arith:		
pushl	%ebp	} SetUp
movl	%esp, %ebp	
movl	8(%ebp), %ecx	} Body
movl	12(%ebp), %edx	
leal	(%edx,%edx,2), %eax	
sall	\$4, %eax	
leal	4(%ecx,%eax), %eax	
addl	%ecx, %edx	
addl	16(%ebp), %edx	} Finish
imull	%edx, %eax	
popl	%ebp	
ret		

## 算术运算

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

```
movl 8(%ebp), %ecx
movl 12(%ebp), %edx
leal (%edx,%edx,2), %eax
sall $4, %eax
leal 4(%ecx,%eax), %eax
addl %ecx, %edx
addl 16(%ebp), %edx
imull    %edx, %eax
```

- 汇编指令与C代码有不同的执行顺序
- 有些代码对应多条指令
- 有些指令一次性完成多行代码

```
# ecx = x
# edx = y
# eax = y*3
# eax *= 16 (t4)
# eax = t4 +x+4 (t5)
# edx = x+y (t1)
# edx += z (t2)
# eax = t2 * t5 (rval)
```

## 特殊算术操作

指令	效果	描述
imull <i>S</i>	$(\%edx : \%eax) \leftarrow S \times \%eax$	有符号全64位乘法
mull <i>S</i>	$(\%edx : \%eax) \leftarrow S \times \%eax$	无符号全64位乘法
cld	$(\%edx : \%eax) \leftarrow \text{SignExtend}(\%eax)$	转为四字
idivl <i>S</i>	$\%edx \leftarrow (\%edx : \%eax) \bmod S$ (余数) $\%eax \leftarrow (\%edx : \%eax) \div S$ (商)	有符号全64位除法
divl <i>S</i>	$\%edx \leftarrow (\%edx : \%eax) \bmod S$ (余数) $\%eax \leftarrow (\%edx : \%eax) \div S$ (商)	无符号全64位除法



# 下一节：控制

湖南大学

《计算机系统》课程教学组

