

计算机系统基础复习大纲



C声明		字节数	
有符号	无符号	32位	64位
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

基本C 数据类型的典型大小(以字节为单位)。
分配的字节数受程序是如何编译的影响而变化。



多字节数据的存放规则

在几乎所有的机器上，多字节对象都被存储为连续的字节序列，对象的地址为所使用字节中最小的地址。

例：unsigned int x=1000;

&i的值: 0x500

32位: 0000 0000 0000 0000 0000 0011 1110 1000 (0x0000003e8)

0x503	00
0x502	00
0x501	03
0x500	e8

↑
increasing
byte
address

小端方式

低位字节存放在低地址单元
高位字节存放在相邻的高地址单元

0x503	e8
0x502	03
0x501	00
0x500	00

↑

大端方式

高位字节存放在低地址单元
低位字节存放在相邻的高地址单元

设intel机器中，有一个int型变量x的地址为0xFFFFC000,x=0x12345678,则内存单元0xFFFFC001中存放的内容的十六进制表示为（ ）

- ☐ A 0x12
- ☐ B 0x34
- ☒ C 0x56
- ☐ D 0x78

提交



n位无符号数, 其可表示的值范围为: $0 \sim +2^n - 1$

C数据类型	位数	最小值	最大值
unsigned char	8	0	255
unsigned short	16	0	0-65535
unsigned int	32	0	4294967295
unsigned long	64	0	18446744 073709551615



补码:

- 补码表示的有符号数，对于正数来说同原码、反码一样，但负数的数值位部分为其绝对值按位取反后末位加1所得。

例如： 8位机器数，则：

+23的补码为 00010111

-23的补码为 11101001

结论1： 一个负数的补码等于对应正数补码的“各位取反、末位加1”



求补码另一种方法

结论2：一个负数的补码等于模（ 2^n ）减该负数的绝对值。

- 23的补码为 100000000

-00010111

11101001



例:

int x=-1000;

32位:

0x503

FF
FF
FA
18

1111 1111 1111 1111 1111 1100 0001 1000
(0xFFFFFA18)

GDB x/4xb

0x503:18 fa ff ff

x/1xw

0x503:ffffffa18

b表示单字节，h表示双字节，w表示四字节，g表示八字节

x表示十六进制，d表示有符号十进制，u表示无符号十进制



n位补码所能表示的真值范围为： $-2^{n-1} \sim +2^{n-1} - 1$

C数据类型	位数	$TMin_w$ 最小值	$TMax_w$ 最大值
char	8	-128	127
short	16	-32768	32767
int	32	-2 147 483 648	2 147 483 647
long	64	-9 223 372 036 854 775 808	9 223 372 036 854 775 807

C语言中char类型的宽度为1个字节，可表示一个字符（非数值数据），也可表示一个8位的整数（数值数据）



假定机器数有n位

$$\textcircled{1} [-2^{n-1}]_{\text{补}} = 2^n - 2^{n-1} = 10\dots0 \text{ (n-1个0)}$$

$$\textcircled{2} [-1]_{\text{补}} = 2^n - 0\dots01 = 11\dots1 \text{ (n个1)}$$

$$\textcircled{3} [+0]_{\text{补}} = [-0]_{\text{补}} = 00\dots0 \text{ (n个0)}$$

没有负零的补码，或者说负零的补码与正零的补码相同。



数	字长 w			
	8	16	32	64
$UMax_w$	0xFF 255	0xFFFF 65 535	0xFFFFFFFF 4 294 967 295	0xFFFFFFFFFFFFFFFF 18 446 744 073 709 551 615
$TMin_w$	0x80 -128	0x8000 -32 768	0x80000000 -2 147 483 648	0x8000000000000000 -9 223 372 036 854 775 808
$TMax_w$	0x7F 127	0x7FFF 32 767	0x7FFFFFFF 2 147 483 647	0x7FFFFFFFFFFFFFFF 9 223 372 036 854 775 807
-1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

图 2-14 重要的数字。图中给出了数值和十六进制表示

补码所对应的真值范围是不对称的，负数个数比正数多一个。
注意：-1 和UMax 有同样的位表示——一个全1 的串
数值0在两种表示方式中都是全0 的串



考虑以下C代码:

```
1 int x = -1;  
2 unsigned u = 2147483648;  
3 printf ( "x = %u = %d\n" , x, x);  
4 printf ( "u = %u = %d\n" , u, u);
```

在机器上运行上述代码时, 它的输出结果是什么? 为什么?

$x = 4294967295 = -1$

$u = 2147483648 = -2147483648$

◆ 2^{31} 的无符号数表示为 "100...0", 被解释为32位带符号整数时, 其值为最小负数: $-2^{32-1} = -2^{31} = -2\ 147\ 483\ 648$ 。

处理同样字长的有符号数和无符号数之间相互转换的一般规则是:数值可能会改变, 但是位模式不变;

强制类型转换的结果保持位值不变, 只是改变了解释这些位的方式



考虑以下C代码：

C语言隐式类型转换

```
int tx ;  
unsigned ux=0xffffffff;  
tx=ux;    /*Cast to signed */  
printf( "tx=%d , ux=%u\n" , tx , ux);  
它的输出结果是什么？
```

tx = -1, ux = 4294967295



IEEE浮点标准 $(-1)^s M 2^E$

– **Sign**符号位 s 代表这个数的正负

– **Significand**尾数 M

– **Exponent**指数（阶码） E

$$(-1)^0 1.011101 \times 2^3$$

• 编码

– MSB s 是符号位 s

– exp 编码 E (but is not equal to E)

– frac 编码 M (but is not equal to M)





C语言中的浮点变量

- 单精度: 32 bits float



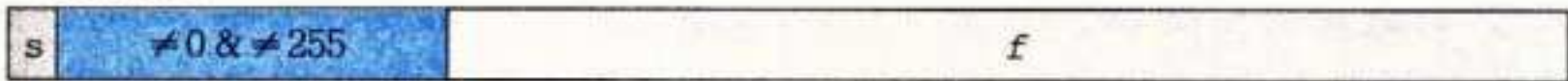
- 双精度: 64 bits double



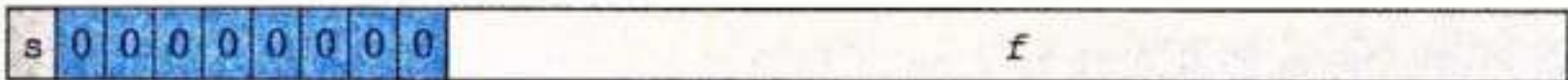


根据exp 的值，被编码的值可以分成三种不同的情况

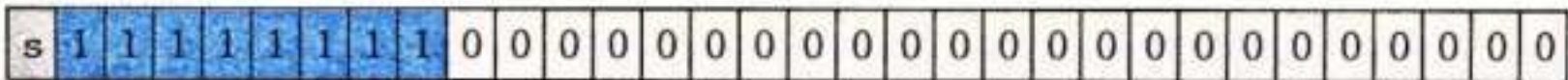
1. 规格化的



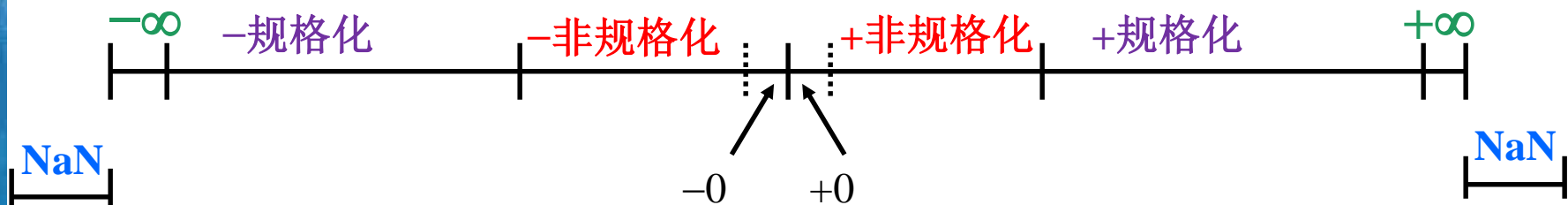
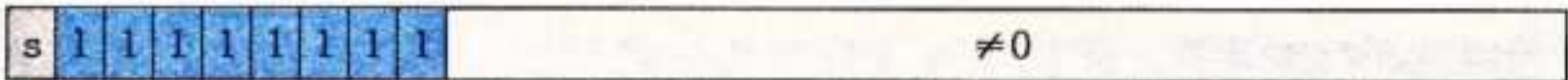
2. 非规格化的



3a. 无穷大



3b. NaN





规格化数: $\pm 1.\text{xxxxxxxxxx}_{\text{two}} \times 2^{\text{Exponent}}$

$$v = (-1)^s M 2^E$$



- $\text{exp} \neq 000\dots 0$ and $\text{exp} \neq 111\dots 1$
- **$\text{exp} = E + \text{Bias}$**
 - exp : 无符号数
 - $\text{Bias} = 2^{k-1} - 1$, $k=8$ 或 11 是 exp 的位数
 - 单精度: 127 (exp : 1...254, E : -126...127)
 - 双精度: 1023 (exp : 1...2046, E : -1022...1023)
- **$M = 1.\text{xxx}\dots\text{x}_2$ $\text{xxx}\dots\text{x}$: frac**
 - Minimum when $\text{frac}=000\dots 0$ ($M = 1.0$)
 - Maximum when $\text{frac}=111\dots 1$ ($M = 2.0 - \epsilon$)

规定: 规格化尾数最高位总是1, 所以隐含表示, 省1位



规格化数: $\pm 1.\text{xxxxxxxxxx}_{\text{two}} \times 2^{\text{Exponent}}$



单精度: $(-1)^s \times (1 + \text{frac}) \times 2^{(\text{exp}-127)}$

双精度: $(-1)^s \times (1 + \text{frac}) \times 2^{(\text{exp}-1023)}$

32位浮点数格式的规格化数的表示范围

$$v = (-1)^s M 2^E$$



规格化数举例

- Value: float $F = 15213.0$;

$$\begin{aligned} 15213_{10} &= 11101101101101_2 \\ &= 1.1101101101101_2 \times 2^{13} \end{aligned}$$

$$\begin{aligned} V &= (-1)^S M 2^E \\ E &= \text{Exp} - \text{Bias} \end{aligned}$$

- Significand

$$M = 1.\underline{1101101101101}_2$$

$$\text{frac} = \underline{1101101101101}0000000000_2$$

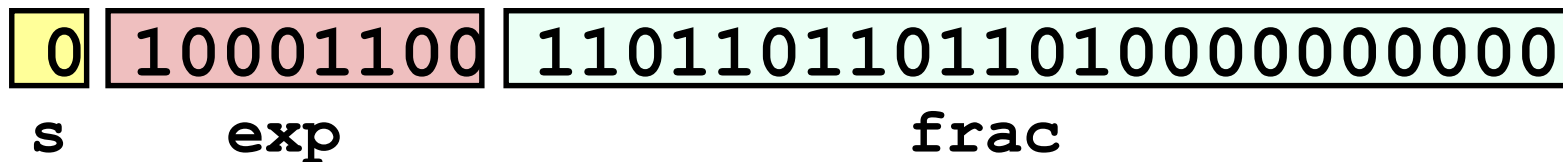
- Exponent

$$E = 13$$

$$\text{Bias} = 127$$

$$\text{Exp} = 140 = 10001100_2$$

- Result:





Denorm d numbers (非规格化数) 计算机系统基础I

非规格化数: $+/-0.\text{xxxxxxxxxxx}_{\text{two}} \times 2^{\text{Exponent}}$



1 8/11-bits

23/52-bits

- Condition: $\text{exp} = 000\dots 0$

- $E = 1 - \text{Bias}$

- 单精度: E: -126
- 双精度: E: -1022

- $M = 0.\text{xxx}\dots\text{x} \quad \text{xxx}\dots\text{x}_2 : \text{bits of } \mathbf{frac}$

– $\mathbf{exp} = 000\dots 0, \mathbf{frac} = 000\dots 0$

- 代表数值 0, 不区分+0和-0

– $\mathbf{exp} = 000\dots 0, \mathbf{frac} \neq 000\dots 0$

- 接近 0.0的小数

$$V = (-1)^s M 2^E$$

$$E = 1 - \text{Bias}$$



In FP, 除数为0的结果是 $\pm\infty$, 不是溢出异常. (整数除0为异常)

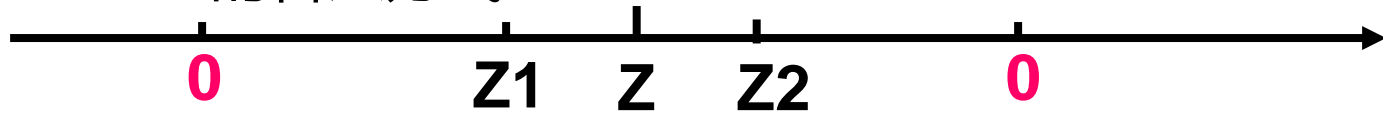
- 可以利用 $+\infty/-\infty$ 作比较。 例如: $X/0 > Y$ 可作为有效比较
- **exp = 111...1**
- 情况1: **exp = 111...1, frac = 000...0**
 - 代表 ∞
 - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$
- 情况2: **exp = 111...1, frac \neq 000...0**
 - Not-a-Number (NaN)
 - E.g., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$



IEEE 754的舍入方式的说明

计算机系统基础I

IEEE 754的舍入方式



(Z1和Z2分别是结果Z的最近的可表示的左、右两个数)

(1) 就近舍入：舍入为最近可表示的数

非中间值：0舍1入；

中间值：强迫结果为偶数-慢

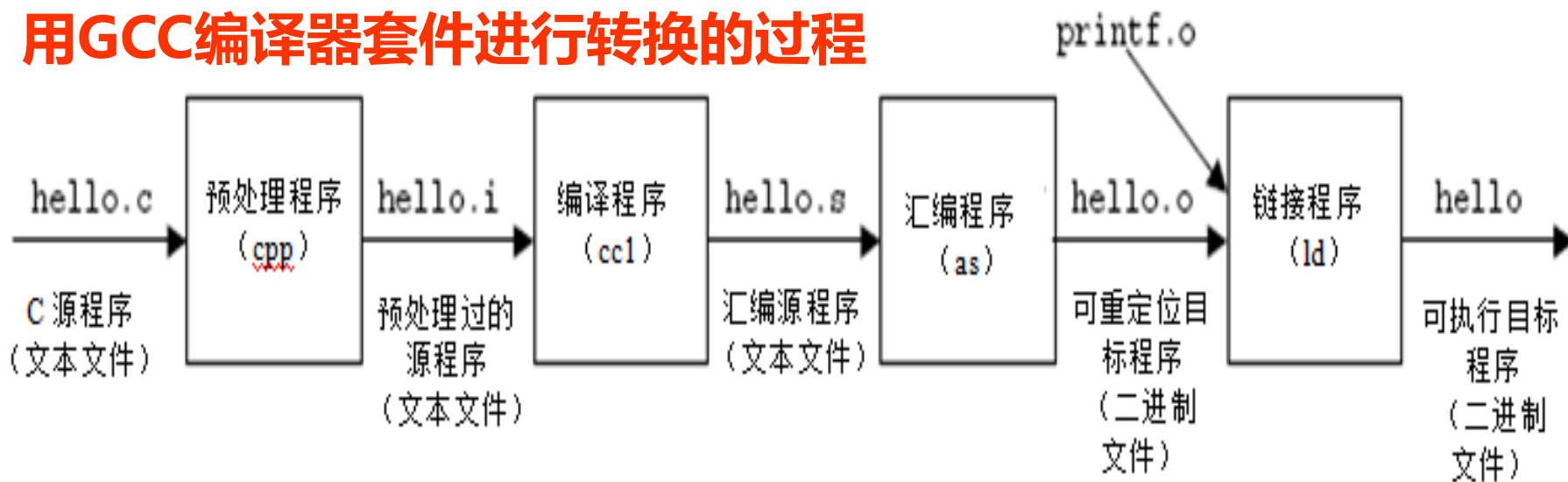
例如：附加位为

01：舍

11：入

10：(强迫结果为偶数)

用GCC编译器套件进行转换的过程



预处理：在高级语言源程序中插入所有用`#include`命令指定的文件和用`#define`声明指定的宏。

编译：将预处理后的源程序文件编译生成相应的汇编语言程序。

汇编：由汇编程序将汇编语言源程序文件转换为可重定位的机器语言目标代码文件。

链接：由链接器将多个可重定位的机器语言目标文件以及库例程（如`printf()`库函数）链接起来，生成最终的可执行目标文件。



指令存放

多字节操作数连续存放，顺序依据小端法规则（Little Endian），
即：低位字节存放在低地址单元，高位字节存放在相邻的高地址单元。

40110a: b8 78 56 34 12

40110f: bb 78 56 34 12

mov \$0x12345678,%eax

mov \$0x12345678,%ebx



- 寻址方式
 - 根据指令给定信息得到操作数
- 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）：
 - 端口中



x86-64 整数寄存器

计算机系统基础I

%rax	%eax
-------------	-------------

%rbx	%ebx
-------------	-------------

%rcx	%ecx
-------------	-------------

%rdx	%edx
-------------	-------------

%rsi	%esi
-------------	-------------

%rdi	%edi
-------------	-------------

%rsp	%esp
-------------	-------------

%rbp	%ebp
-------------	-------------

%rip	%eip
-------------	-------------

%r8	%r8d
------------	-------------

%r9	%r9d
------------	-------------

%r10	%r10d
-------------	--------------

%r11	%r11d
-------------	--------------

%r12	%r12d
-------------	--------------

%r13	%r13d
-------------	--------------

%r14	%r14d
-------------	--------------

%r15	%r15d
-------------	--------------



general purpose

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

accumulator
Origin
(mostly obsolete)

counter

data

base

source
index

destination
index

stack
pointer

base
pointer

16-bit virtual registers
(backwards compatibility)



Imm	$M[Imm]$	绝对寻址
(r_a)	$M[R[r_a]]$	间接寻址
$Imm(r_b)$	$M[Imm+R[r_b]]$	(基址 + 偏移量) 寻址
(r_b, r_i)	$M[R[r_b]+R[r_i]]$	变址寻址
$Imm(r_b, r_i)$	$M[Imm+R[r_b]+R[r_i]]$	变址寻址
(r_i, s)	$M[R[r_i] \cdot s]$	比例变址寻址
$Imm(r_i, s)$	$M[Imm+R[r_i] \cdot s]$	比例变址寻址
(r_b, r_i, s)	$M[R[r_b]+R[r_i] \cdot s]$	比例变址寻址
$Imm(r_b, r_i, s)$	$M[Imm+R[r_b]+R[r_i] \cdot s]$	比例变址寻址



寻址方式举例

地址	值
0x100	0xff
0x104	0xab
0x108	0x13
0x10c	0x11

寄存器	值
rax	0x100
rcx	0x1
rdx	0x3

操作数	值	操作数	值
%rax	0x100	9(%rax,%rdx)	0x11
0x104	0xab	260(%rcx,%rdx)	0x13
\$0x108	0x108	0xFC(,%rcx,4)	0xff
(%rax)	0xff	(%rax,%rcx,4)	
4(%rax)	0xab	9(%rax,%rdx)	



↑ 假设下面的值存放在指定的内存地址和寄存器中：

地址	值
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

寄存器	值
%rax	0x100
%rcx	0x1
%rdx	0x3

包括其他指令，如**LEA**，逻辑运算、移位指令等

指令	目的	值
addq %rcx, (%rax)	内存地址: 0x100	0x100
subq %rdx, 8(%rax)	内存地址: 0x108	0xA8
imulq \$16, (%rax, %rdx, 8)	内存地址: 0x118	0x110
incq 16(%rax)	内存地址: 0x110	0x14
decq %rcx	%rcx	0x0
subq %rdx, %rax	%rax	0xfd



- 假设 $R[ax]=FFFAH$ ， $R[bx]=FFF0H$ ，则执行以下指令后

`"addw %bx, %ax"`

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： $R[ax] \leftarrow R[ax] + R[bx]$ ，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 $65535-21=65514 \neq 65530+65520$ ，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



- **lea Src, Dst**

- 计算有效地址

- E.g., translation of **p = &x[i];**

- 算术运算表达式 形式为: $x + k*y$

- $k = 1, 2, 4, \text{ or } 8$

- 例 `leaq 7(%rdx,%rdx,4),%rax` $\%rax = 5 * \%rdx + 7$

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

Converted to ASM by compiler:

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




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 $((x \ll 2) + x) \gg 1$ 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。

逆向工程：从汇编指令退出高级语言程序代码



跳转指令

– 无条件跳转指令

jmp DST: 无条件转移到目标指令DST处执行

(1) 直接跳转

jmp 标号

(2) 间接跳转

例如: jmp *%rdx jmp *(%rdx)

– 条件跳转指令

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



分三类：

(1)根据单个标志的值转移

(2)按无符号整数比较转移

(3)按带符号整数比较转移

序号	指令	转移条件	说明
1	jc label	CF=1	有进位/借位
2	jnc label	CF=0	无进位/借位
3	je/jz label	ZF=1	相等/等于零
4	jne/jnz label	ZF=0	不相等/不等于零
5	js label	SF=1	是负数
6	jns label	SF=0	是非负数
7	jo label	OF=1	有溢出
8	jno label	OF=0	无溢出
9	ja/jnbe label	CF=0 AND ZF=0	无符号整数 $A > B$
10	jae/jnb label	CF=0 OR ZF=1	无符号整数 $A \geq B$
11	jb/jnae label	CF=1 AND ZF=0	无符号整数 $A < B$
12	jbe/jna label	CF=1 OR ZF=1	无符号整数 $A \leq B$
13	jg/jnle label	SF=OF AND ZF=0	带符号整数 $A > B$
14	jge/jnl label	SF=OF OR ZF=1	带符号整数 $A \geq B$
15	jl/jnge label	SF \neq OF AND ZF=	带符号整数 $A < B$
16	jle/jng label	SF \neq OF OR ZF=1	带符号整数 $A \leq B$

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。



跳转指令的编码

```
1    movq    %rdi, %rax
2    jmp     .L2
3    .L3:
4    sarq    %rax
5    .L2:
6    testq   %rax, %rax
7    jg      .L3
8    rep; ret
```

汇编器产生的“`.o`”格式的反汇编版本如下：

1	0:	48 89 f8	mov	%rdi,%rax
2	3:	eb 03	jmp	8 <loop+0x8>
3	5:	48 d1 f8	sar	%rax
4	8:	48 85 c0	test	%rax,%rax
5	b:	7f f8	jg	5 <loop+0x5>
6	d:	f3 c3	repz retq	



```
long pcount_for
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

```
    movl    $0x0,%edx
    movl    $0x0,%ecx
    jmp     L1
L2:  movq    %rdi,%rax
    shrq    %cl,%rax
    andl    $0x1,%eax
    addq    %rax,%rdx
    addq    $0x1,%rcx
L1:  cmpq    $63,%rcx
    jle     L2
    movq    %rdx,%rax
    ret
```



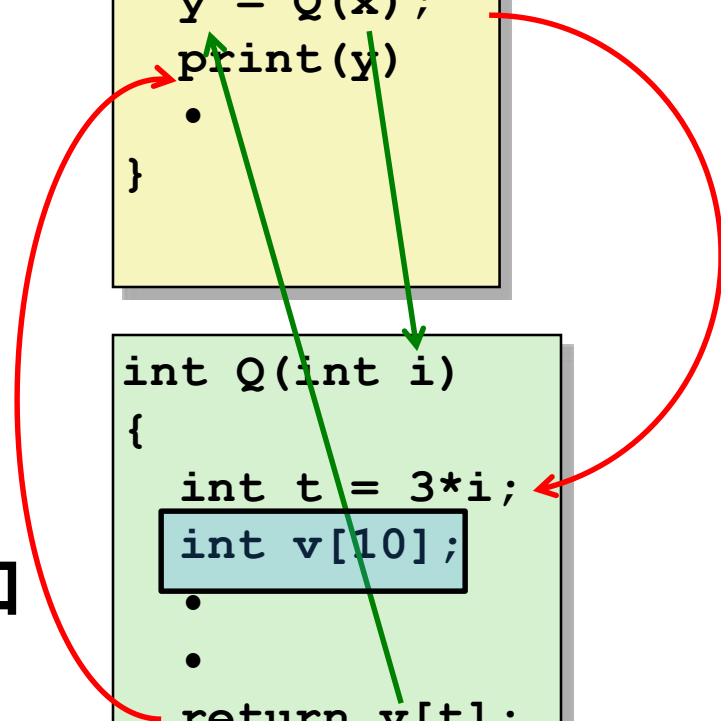
过程调用的机器级表示

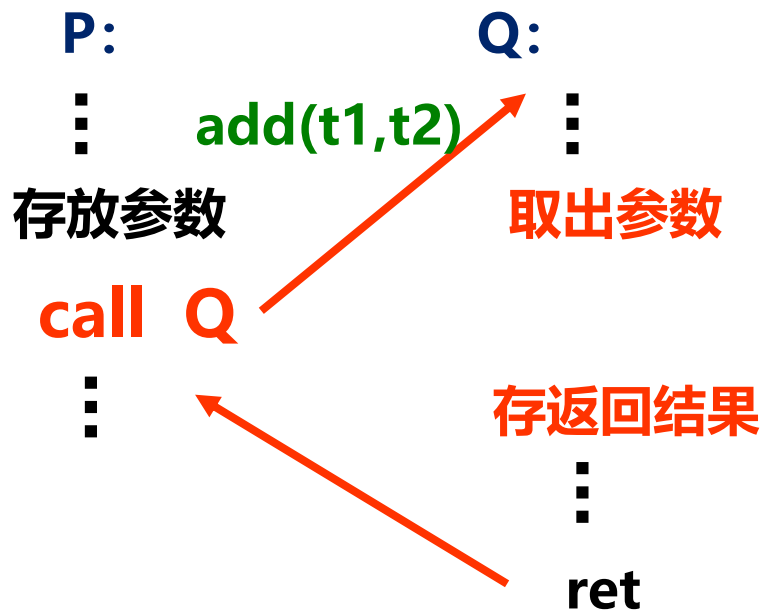
- 传递控制
过程的调用和返回
- 传递数据
参数的传递和返回值
- 内存的分配和释放
为局部变量分配空间及释放

x86-64 的过程实现包括过程调用和返回指令和一些对机器资源(例如寄存器和程序内存)使用的约定规则。

```
P (...) {  
    •  
    •  
    y = Q(x);  
    print(y)  
    •  
}
```

```
int Q(int i)  
{  
    int t = 3*i;  
    int v[10];  
    •  
    •  
    return v[t];  
}
```





直接调用: call 标号

间接调用: call *operand

过程调用call指令

- (1) P保存返回地址
- (2) 跳转到Q

过程返回ret指令

- (1) 从堆栈栈顶取出P的返回地址
- (2) 跳转到返回地址, 即call Q指令后面的指令继续执行。



寄存器

- 通过寄存器最多传递6个整型(例如整数和指针)参数

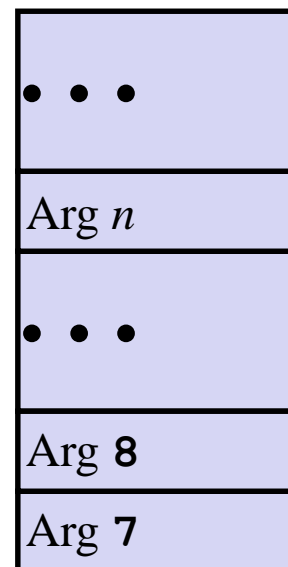
%rdi
%rsi
%rdx
%rcx
%r8
%r9

返回值

%rax

堆栈

- 需要的时候分配堆栈空间



操作数宽度 (字节)	入口参数						返回 参数
	1	2	3	4	5	6	
8	RDI	RSI	RDX	RCX	R8	R9	RAX
4	EDI	ESI	EDX	ECX	R8D	R9D	EAX
2	DI	SI	DX	CX	R8W	R9W	AX
1	DIL	SIL	DL	CL	R8B	R9B	AL



- 内容
 - 返回地址
 - 调用另一个过程设置的参数7~n
 - 局部变量

(1)寄存器不足够存放所有的本地数据。

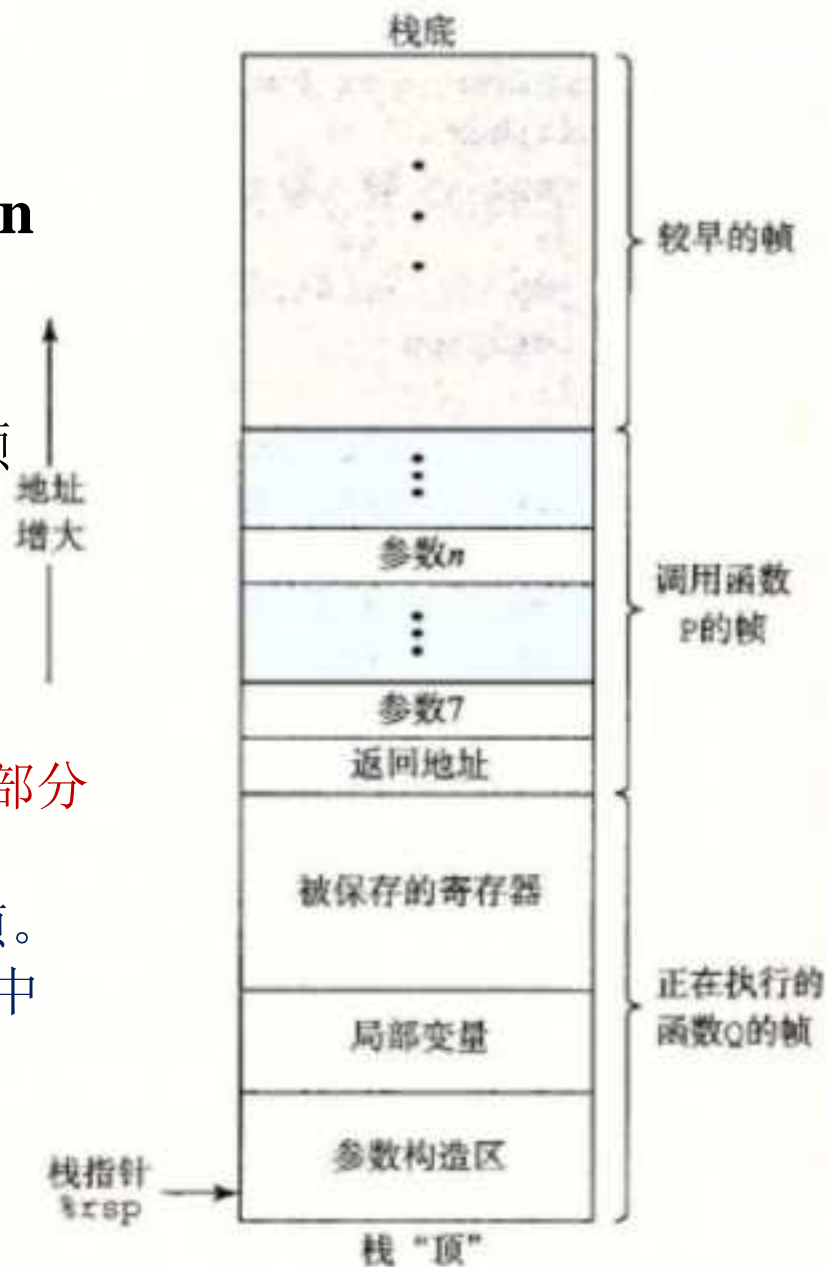
(2)局部变量使用地址运算符 '&'，必须为它产生一个地址。

(3)局部变量是数组或结构

- 被保存的现场寄存器

X86-64 过程只分配自己所需要的栈帧部分

实际上，许多函数甚至根本不需要栈帧。若所有的局部变量都可以保存在寄存器中，而且该函数不会调用任何其他函数。





数组的分配和访问

$T\ A[L];$

- 用标识符A 来作为指向数组开头的指针: Type T^*
- 用0~N-1 的整数索引来访问该数组元素
- 第 i ($0 \leq i \leq 3$) 个元素的地址计算公式: $\&A[0] + \text{sizeof}(T) * i$

main ()

{

static short num[][4]={ {2, 9, -1, 5},
 {3, 8, 2, -6}};

static short *pn[]={num[0], num[1]};

static short s[2]={0, 0};

08049300 <num>:

08049300: 02 00 09 00 ff ff 05 00 03 00 08 00 02 00 fa ff

08049310 <pn>:

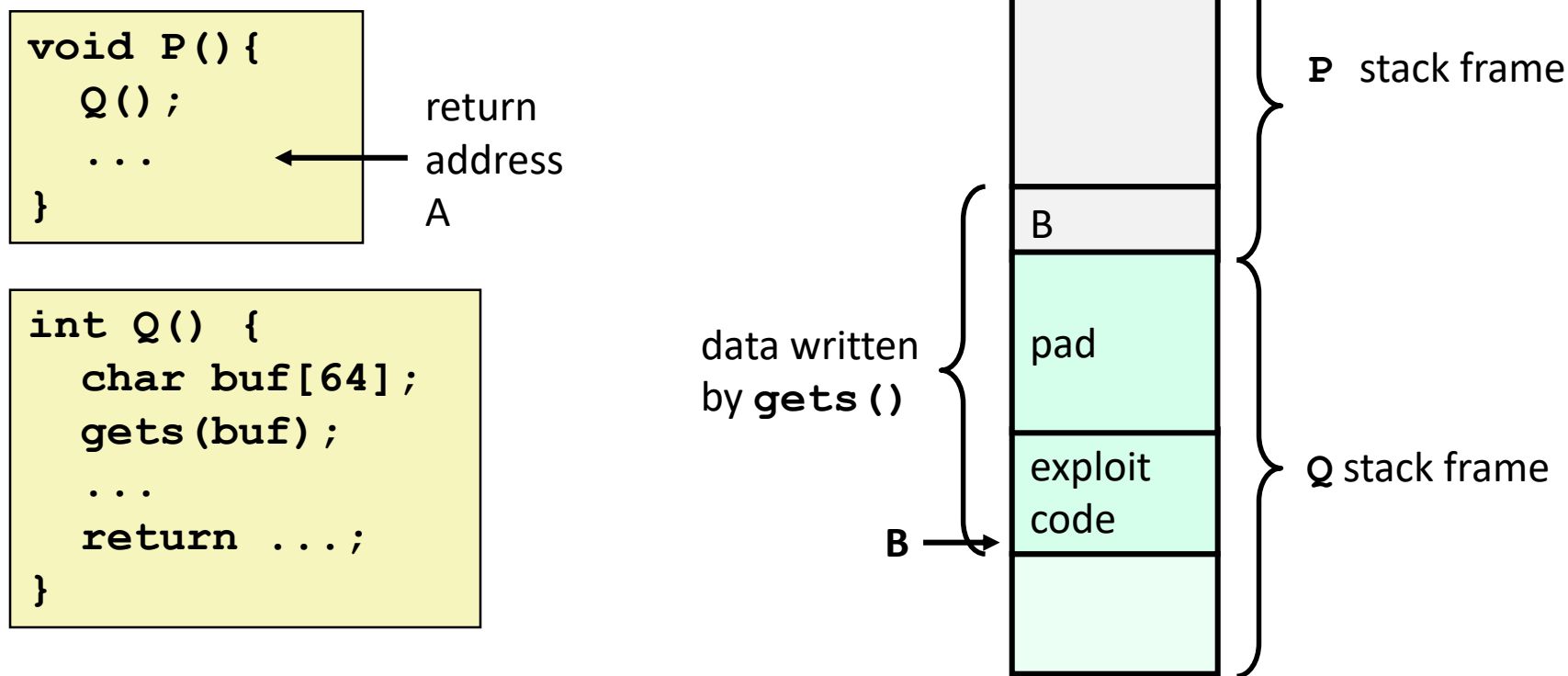
08049310: 00 93 04 08 00 00 00 00 08 93 04 08 00 00 00 00

08049320 <s>:

08049320: 00 00 00 00



代码注入攻击Code Injection Attacks



- 输入给程序一个字符串，这个字符串包含一些可执行代码的字节编码，称为攻击代码(exploit code)，
- 攻击代码的地址覆盖原返回地址。
- 执行ret 指令跳转到攻击代码



1. 程序员角度防范——在代码中避免

- 使用带限制长度的库函数
 - **fgets** instead of **gets**等

2、系统级别的防范

栈随机化

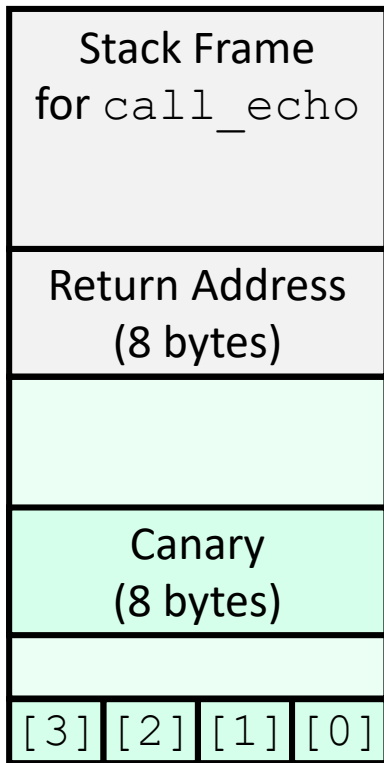
限制可执行代码区域

栈破坏检测



Setting Up Canary

Before call to gets



`buf ← %rsp`

```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small! */  
    gets(buf);  
    puts(buf);  
}
```

```
echo:  
    . . .  
    movq    %fs:40, %rax    # Get canary  
    movq    %rax, 8(%rsp)  # Place on stack  
    xorl    %eax, %eax     # Erase canary  
    . . .
```



过程中的浮点代码

在x86 -64 中，XMM 寄存器用来向函数传递浮点参数，以及从函数返回浮点值。具体规则：

(1) XMM 寄存器%xmm0~ %xmm7 最多可以传递8 个浮点参数。按照参数列出的顺序使用这些寄存器。可以通过栈传递额外的浮点参数。

(2) 函数使用寄存器%xmm0来返回浮点值。

(3) 当函数包含指针、整数和浮点数混合的参数时，指针和整数通过通用寄存器传递，而浮点值通过XMM 寄存器传递

例：double fl(int x , double y , long z);

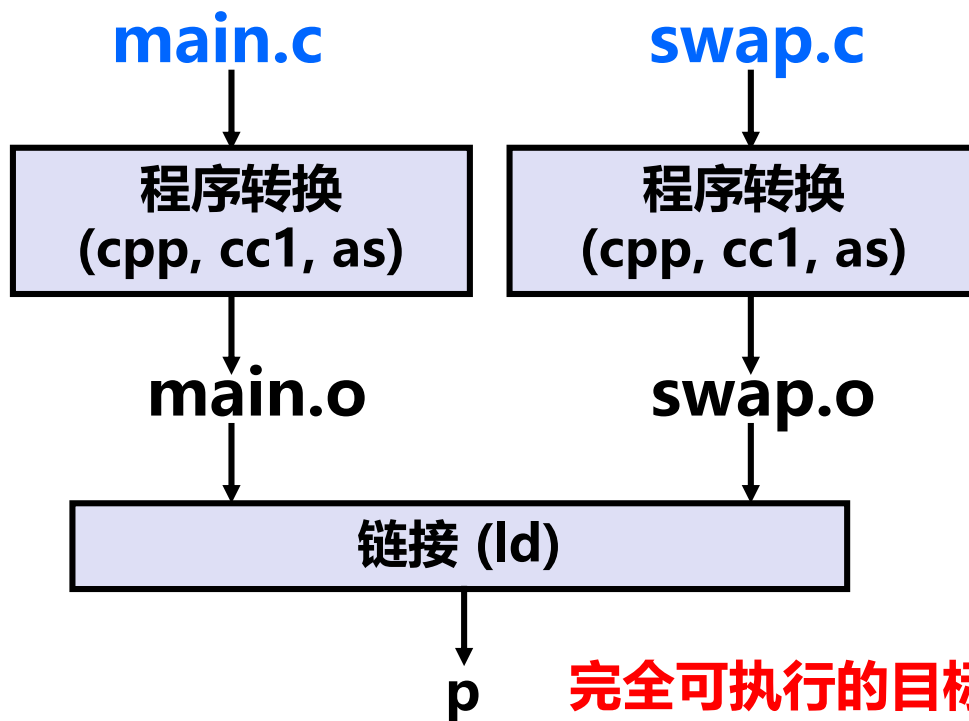
x 存放在%edi 中， y 存放在%xmm0 中，而z存放在%rsi 中

double fl(float x , double *y , long *z);

x 放在%xmm0 中， y 放在%rdi 中，而z 放在%rsi 中

- 使用GCC编译器编译并链接生成可执行程序P:
 - \$ gcc -o p main.c swap.c
 - \$./p

**GCC
编译
器的
静态
链接
过程**



源程序文件

分别转换（预处理、编译、汇编）为可重定位目标文件

完全可执行的目标文件



可重定位目标文件格式

ELF 头

- ✓ 定义了ELF魔数、版本、小端/大端、操作系统平台、目标文件的类型、机器结构类型、节头表的起始位置和长度等

.text 节

- ✓ 编译后的代码部分

.rodata 节

- ✓ 只读数据，如 printf 格式串、switch 跳转表等

.data 节

- ✓ 已初始化的全局变量

.bss 节

- ✓ 未初始化全局变量，仅是占位符，不占据任何实际磁盘空间。区分初始化和非初始化是为了空间效率

ELF 头
.text 节
.rodata 节
.data 节
.bss 节
.symtab 节
.rel.txt 节
.rel.data 节
.debug 节
.line 节
.strtab 节
Section header table (节头表)



- Step 1. 符号解析 (Symbol resolution)
 - 程序中有定义和引用的符号 (包括变量和函数等)
 - `void swap() {...} /* 定义符号swap */`
 - `swap(); /* 引用符号swap */`
 - `int *xp = &x; /* 定义符号 xp, 引用符号 x */`
 - 编译器将定义和引用的符号存放在一个符号表 (symbol table) 中.
 - 链接器将每个符号的引用都与一个确定的符号定义建立关联
- Step 2. 重定位
 - 将多个代码节与数据节分别合并为一个单独的代码段和数据段
 - 计算每个定义的符号在虚拟地址空间中的绝对地址
 - 将可执行文件中符号引用处的地址修改为重定位后的地址信息



每个可重定位目标模块m都有一个符号表，它包含了在m中定义的符号。
有三种链接器符号：

- **Global symbols** (模块内部定义的全局符号)

- 由模块m定义并能被其他模块引用的符号。例如，非static函数和非static的全局变量（指不带static的全局变量）

如，main.c 中的全局变量名buf

- **External symbols** (外部定义的全局符号)

- 由其他模块定义并被模块m引用的全局符号

如，main.c 中的函数名swap

- **Local symbols** (本模块的局部符号)

- 仅由模块m定义和引用的本地符号。例如，在模块m中定义的带static的函数和全局变量

如，swap.c 中的static变量名bufp1

链接器局部符号不是指程序中的局部变量（分配在栈中的临时性变量），链接器不关心这种局部变量



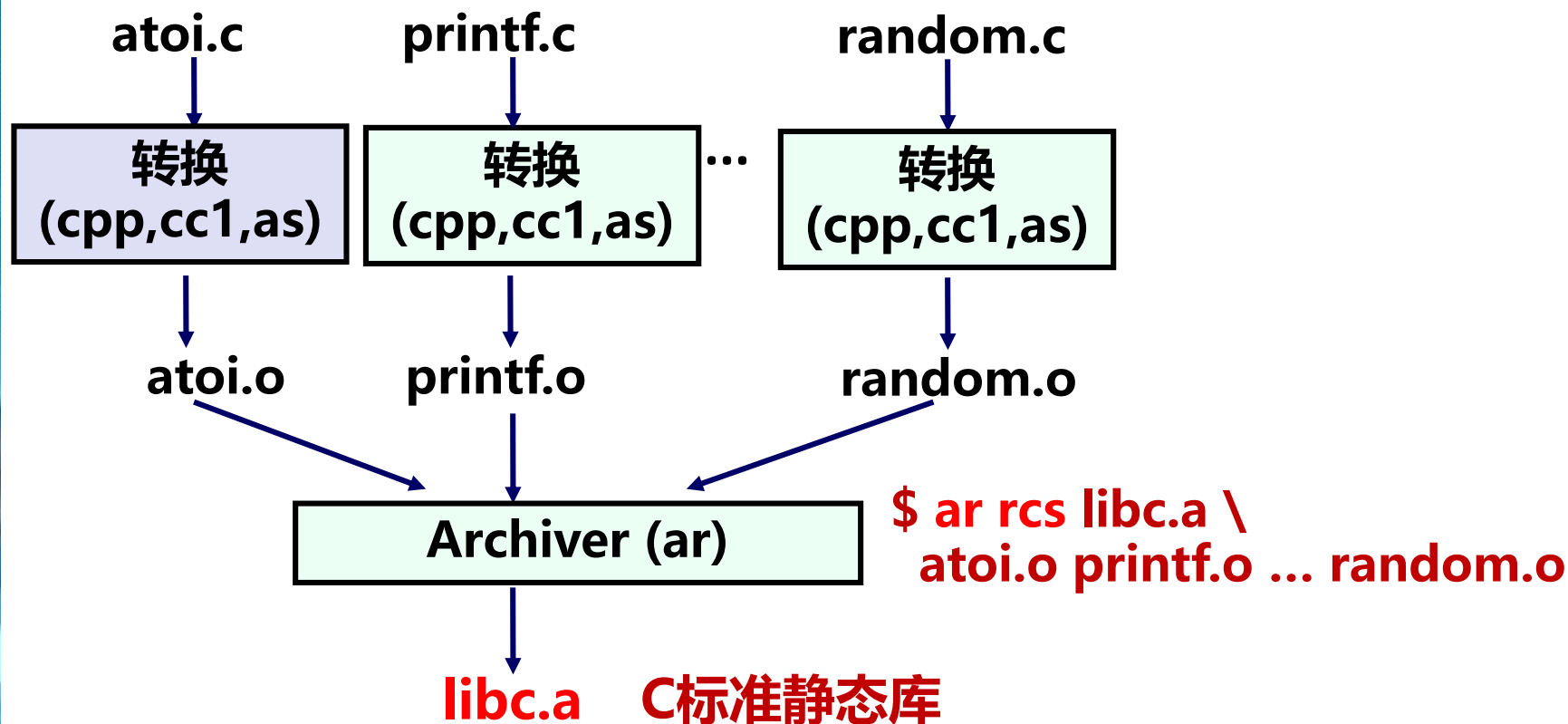
main.c

```
int buf[2] = {1, 2};  
extern void swap();  
  
int main()  
{  
    swap();  
    return 0;  
}
```

swap.c

```
extern int buf[];  
  
int *bufp0 = &buf[0];  
static int *bufp1;  
  
void swap()  
{  
    int temp;  
  
    bufp1 = &buf[1];  
    temp = *bufp0;  
    *bufp0 = *bufp1;  
    *bufp1 = temp;  
}
```

哪些是全局符号？ 哪些是外部符号？ 哪些是局部符号？



- Archiver (归档器) 允许增量更新, 只要重新编译需修改的源码并将其.o文件替换到静态库中。

在gcc命令中无需明显指定C标准库libc.a(默认库)



- 假设调用关系如下：

func.o \rightarrow libx.a 和 liby.a 中的函数

libx.a \rightarrow libz.a 中的函数

libx.a 和 liby.a 之间、liby.a 和 libz.a 相互独立

则以下几个命令行都是可行的：

- gcc -static -o myfunc func.o libx.a liby.a libz.a
- gcc -static -o myfunc func.o liby.a libx.a libz.a
- gcc -static -o myfunc func.o libx.a libz.a liby.a

- 假设调用关系如下：

func.o \rightarrow libx.a 和 liby.a 中的函数

libx.a \rightarrow liby.a 同时 liby.a \rightarrow libx.a

则以下命令行可行：

- gcc -static -o myfunc func.o libx.a liby.a libx.a
- gcc -static -o myfunc func.o liby.a libx.a liby.a



`readelf -r main.o` 可重定位信息在可重定位目标 (.o) 文件中!

Relocation section '.rel.text' at offset 0x190 contains 1 entry:

Offset	Info	Type	Sym. Value	Sym. Name + Addend
000000000000e	0004000000004	R_X86_64_PLT32	0000000000000000	swap - 4

汇编器遇到引用时，生成一个重定位条目数据引用的重定位条目在.rel_data节中
指令中引用的重定位条目在.rel_text节中

ELF重定位条目的结构如下：

```
typedef struct
```

```
{ long offset; // 需要被修改的节偏移  
  int type; // 重定位类型  
  int symbol; // 被修改引用指向的符号  
  long addend; // 重定位使用的调整偏移值  
} Elf64_Rela;
```

重定位类型：

- **R_X86_64_64(32)**：重定位一个使用64位绝对地址的引用
- **R_X86_64_PC32**：重定位一个使用32位PC相对地址的引用。
- **R_X86_64_PLT32**：过程链接表延迟绑定。



• 假定：

- 可执行文件中main函数对应机器代码从0x401106开始
- swap紧跟main后

• 则swap起始地址为

- 0x401106+0x1C

• 则重定位后call指令

- 转移目标地址=PC+ **偏移量(重定位值)**,

- PC= **ADDR(s) ++r.offset - r.addend**
= 0x401106+0x0e- (-4) =0x401118

- **重定位值**=转移目标地址-PC=0x4x1122-0x401118=0xa

- call指令的机器代码为 "e8 0a 00 00 00"

main.o中text节的地址

PC相对地址方式下，重定位值计算公式为：

$$\text{ADDR(r.sym)} - ((\text{ADDR(.text)} + \text{r.offset}) - \text{r.addend})$$

引用目标处地址

call指令下条指令地址

即当前PC的值

Disassembly of section .text:

00000000 <main>:

.....

d: e8 **00 00 00 00** call 12 <main+0x12>

e: R_X86_64_PLT32 swap-0x4

12: b8 00 00 00 00 mov \$0x0,%eax



ELF头信息举例

\$ readelf -h main

可执行目标文件的ELF头

ELF Header:

Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00

Class: ELF64

Data: 2's complement, little endian

Version: 1 (current)

OS/ABI: UNIX - System V

ABI Version: 0

Type: EXEC (Executable file)

Machine: Advanced Micro Devices X86-64

Version: 0x1

Entry point address: 0x401020

Start of program headers: 64 (bytes into file)

Start of section headers: 14000 (bytes into file)

Flags: 0x0

Size of this header: 64 (bytes)

Size of program headers: 56 (bytes)

Number of program headers: 13

Size of section headers: 64 (bytes)

Number of section headers: 28

Section header string table index: 27

13x56B

28x40B

ELF 头

程序头表

.init 节

.text 节

.rodata 节

.data 节

.bss 节

.symtab 节

.debug 节

.strtab 节

.line 节

Section header table
(节头表)



加载时动态链接

调用关系:

main→add
vec→printf

main2.c vector.h

```
unix> gcc -shared -o libvector.so \
      addvec.c multvec.c
```

Translators
(cpp, cc1, as)

libc.so
libvector.so

*Relocatable
object file*

main2.o

*Relocation and symbol
table info*

Linker (ld)

prog2l

*Partially linked
executable object file*

Loader
(execve)

libc.so
libvector.so

Code and data

*Fully linked
executable
in memory*

Dynamic linker (ld-linux.so)

加载 prog2l 时, 加载器发现在其程序头表中有 .interp 段, 其中包含了动态链接器路径名 ld-linux.so, 因而加载器根据指定路径加载并启动动态链接器运行。动态链接器完成相应的重定位工作后, 再把控制权交给 procg, 启动其第一条指令执行。



- 动态链接用到一个重要概念：
 - 位置无关代码 (Position-Independent Code, PIC)
 - GCC选项-fPIC指示生成PIC代码
- 共享库代码是一种PIC
 - 共享库代码的位置可以是不确定的
 - 即使共享库代码的长度发生变化, 也不影响调用它的程序
- 引入PIC的目的
 - 链接器无需修改代码即可将共享库加载到任意地址运行
- 过程调用、跳转, 变量的访问

要实现动态链接,
必须生成PIC代码

所有引用情况

- (1) 模块内的过程调用、跳转, 采用PC相对偏移寻址
- (2) 模块内数据访问, 如模块内的全局变量和静态变量
- (3) 模块外的过程调用、跳转
- (4) 模块外的数据访问, 如外部变量的访问



C程序如下所示，其中f1(n)函数用于计算 $f(n) = \sum_{i=0}^n 2^i = 2^{n+1} - 1$

=11.....1B
(有n+1个1)

```
#include "stdio.h"
int f1( int n )
{   int i;
    int sum = 1, power = 1;
    for ( i = 0; i <= n - 1; i++ )
    { power *= 2;
      sum += power;
    }
    return sum;
}
void main()
{   int n,sum;
    scanf("%d",&n);
    sum=f1(n);
    printf("sum=%d\n",sum);
}
```



00401176 <f1>:

```
401176:      f3 0f 1e fa
40117a:      ba 01 00 00 00
40117f: b9 01 00 00 00
401184:      b8 00 00 00 00
401189:      eb 07
40118b:      01 d2
40118d:      01 d1
40118f: 83 c0 01
401192:      39 f8
401194:      7c f5
401196:      89 c8
401198:      c3
```

```
endbr64
mov    $0x1,%edx
mov    $0x1,%ecx
mov    $0x0,%eax
jmp    401192 <f1+0x1c>
add    %edx,%edx
add    %edx,%ecx
add    $0x1,%eax
cmp    %edi,%eax
jl     40118b <f1+0x15>
mov    %ecx,%eax
ret
```



0401199 <main>:

401199:	f3 0f 1e fa	endbr64
40119d:	48 83 ec 18	sub \$0x18,%rsp
4011a1:	64 48 8b 04 25 28 00	mov %fs:0x28,%rax
4011a8:	00 00	
4011aa:	48 89 44 24 08	mov %rax,0x8(%rsp)
4011af: 31 c0		xor %eax,%eax
4011b1:	48 8d 74 24 04	lea 0x4(%rsp),%rsi
4011b6:	48 8d 3d 47 0e 00 00	lea 0xe47(%rip),%rdi
4011bd:	e8 be fe ff ff	call 401080
<__isoc99_scanf@plt>		
4011c2:	8b 7c 24 04	mov 0x4(%rsp),%edi
4011c6:	e8 ab ff ff ff	call 401176 <f1>
4011cb:	89 c2	mov %eax,%edx
4011cd:	48 8d 35 33 0e 00 00	lea 0xe33(%rip),%rsi
4011d4:	bf 01 00 00 00	mov \$0x1,%edi
4011d9:	b8 00 00 00 00	mov \$0x0,%eax

.....



程序部分调试信息如下：

```
(gdb) i r rip
```

```
rip      0x401196
```

```
(gdb) i r rsp
```

```
rsp      0x7fffffffde18
```

```
(gdb) x/4xg $rsp
```

```
0x7fffffffde18: 0x0000000000004011cb
```

```
0x0000000040000000
```

```
0x7fffffffde28: 0xfc957cf2acabe600
```

```
0x0000000000000000
```

1. 基于调试信息，输入的n的值是多少？（十进制表示）

4

2. 此时eax, edi,ecx寄存器的值分别是多少？（十进制表示）

4 4 31



3. 执行下述两条指令的作用是什么？（对照C语句解释）

```
401192: 39 f8          cmp    %edi,%eax
401194: 7c f5          jl     40118b <f1+0x15>
```

4. 在给出的main（）的部分反汇编代码中，哪些指令会改变rsp寄存器的内容？选取其中一条指令，基于调试信息，说明该指令执行后rsp寄存器的内容是什么？

5. 在main（）和f1（）中都定义了sum变量，为什么这两个sum变量的访问没有冲突？