计算机系统基础I复习大纲

题型

- 选择题：20分
- 分析题：30分
- 实验题：15分
- 综合应用题：35分

计算机系统性能指标

CPU时钟周期

机器内部主时钟脉冲信号的宽度，它是CPU工作的最小时间单位。

主频

主频也叫时钟频率，用来表示微处理器的运行速度，主频越高表明微处理器运行越快，主频的单位是MHz。

每条指令平均时钟周期数CPI：Cycles Per Instruction

CPI = 执行程序所需的时钟周期数 / IC

IC：所执行的指令条数

执行一个程序所需的CPU时间

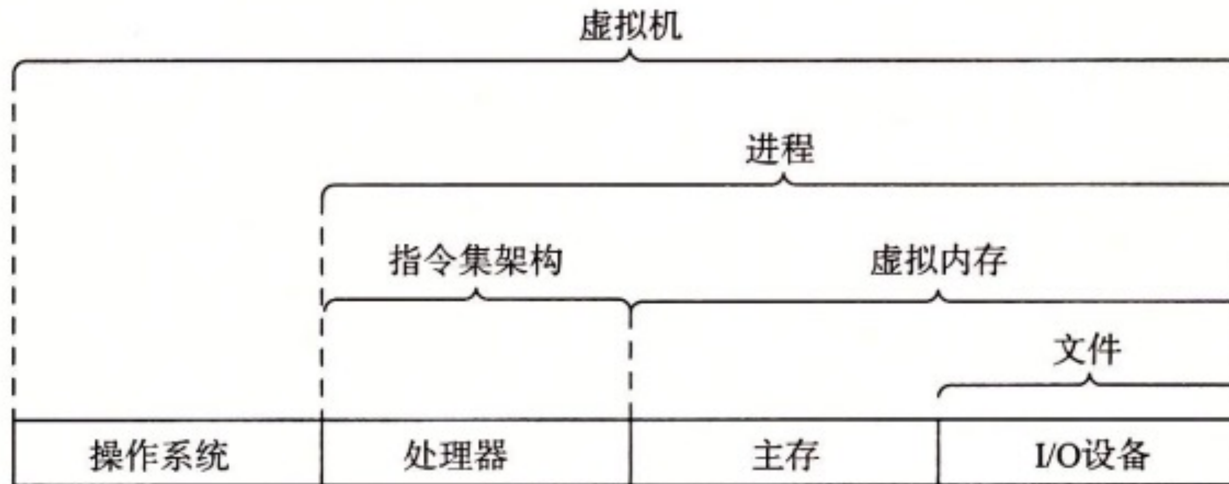
=执行程序所需的时钟周期数×时钟周期时间

=执行程序所需的时钟周期数/时钟频率

=IC× CPI /时钟频率

CPU的性能取决于三个参数：时钟频率，CPI，IC

计算机系统层次结构



（1）指令集架构ISA

提供了对实际处理器硬件的抽象。使用这个抽象，机器代码程序表现得就好像运行在一个一次只执行一条指令的处理器上。

（2）虚拟内存

提供程序存储器的抽象， 包括主存和磁盘I/O设备

（3）文件

提供了对I/0 设备的抽象

（4）进程

提供正在运行的程序的抽象。包括对处理器、主存和1/0 设备的抽象表示

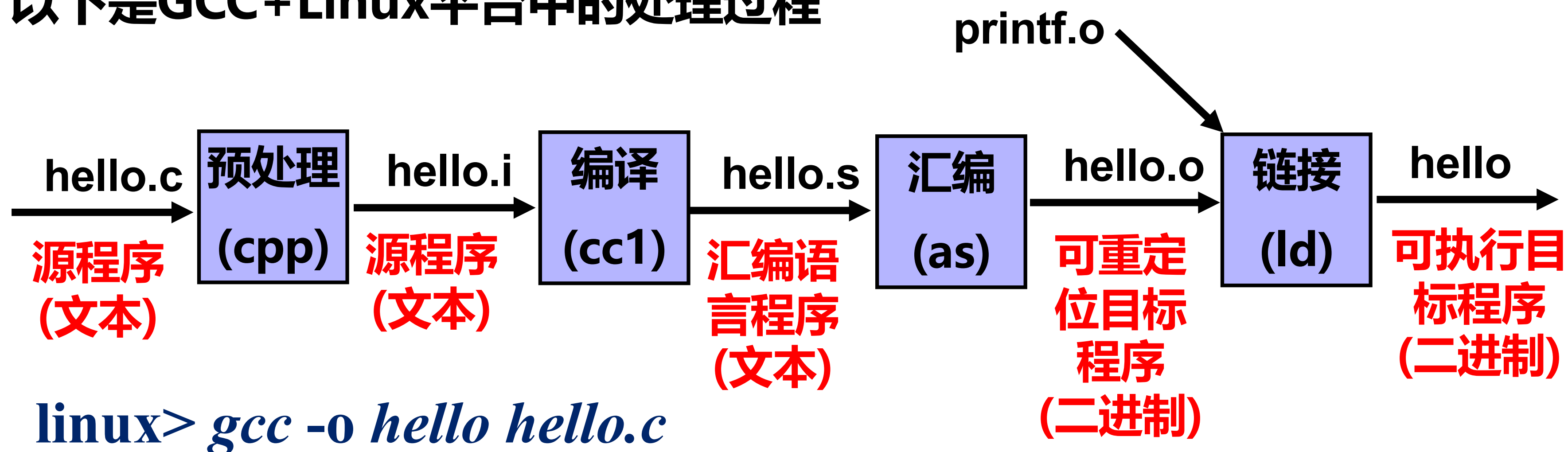
（5）虚拟机

提供了对整个计算机抽象

可执行文件的生成

高级语言程序开发过程

以下是GCC+Linux平台中的处理过程

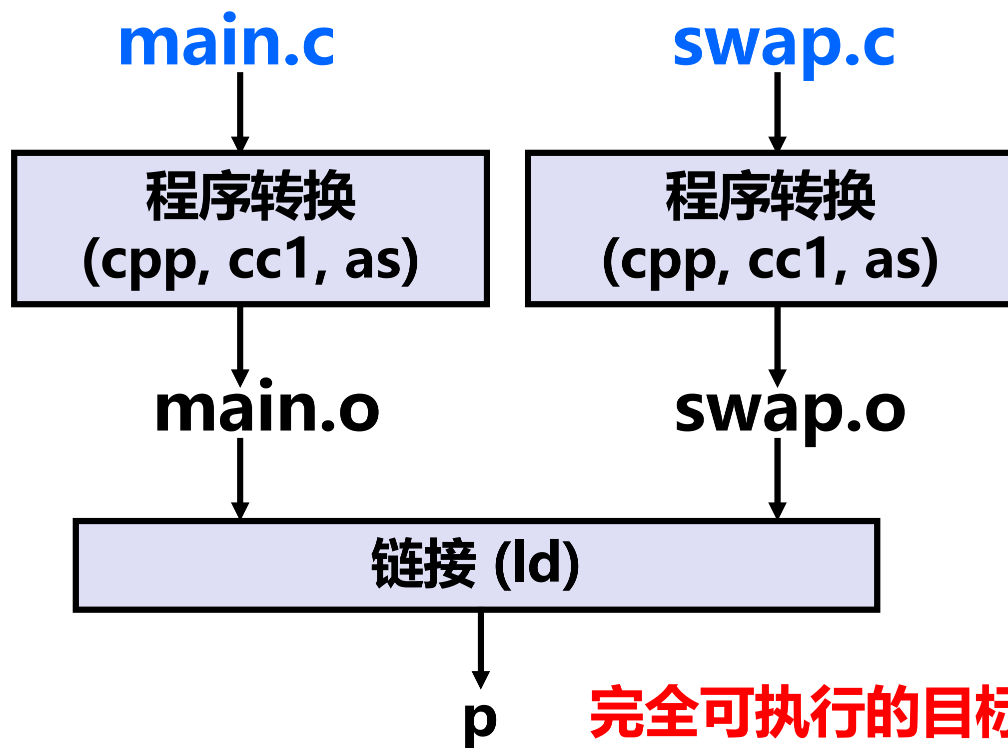


linux> ***gcc*** **-o** ***hello hello.c***

可执行文件的生成

- 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

计算机中数据的编码

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　(0x000003e8)

地址	小端方式
0x503	00
0x502	00
0x501	03
0x500	e8

increasing byte address ↑

地址	大端方式
0x503	e8
0x502	03
0x501	00
0x500	00

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

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

单选题 1分

设置

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

无符号整数编码

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

有符号整数编码

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:

rsp=0xf5dd1550

FF
FF
FA
18

1111 1111 1111 1111 1111 1100 0001 1000

(0xFFFFFA18)

GDB x/4xb 0xf5dd1550 **x/1xw 0xf5dd1550**

GDB x/4xb $rsp

0xf5dd1550 :18 fa ff ff **0xf5dd1550 :ffffa18**

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 147483 648	2147483 647
long	64	-9223 372 036 854 775 808	9223 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）

无符号数和有符号数的转换

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

无符号数和有符号数的转换

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

浮点数表示

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 E

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

- 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)

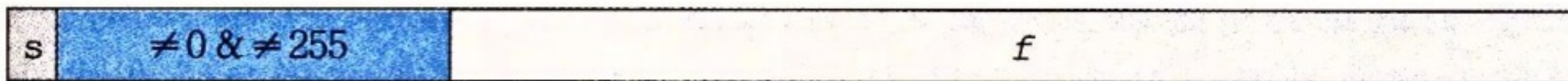
C语言中的浮点变量

- 单精度: 32 bits float

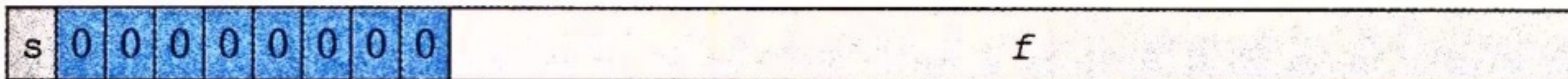
- 双精度: 64 bits double

根据exp 的值，被编码的值可以分成三种不同的情况

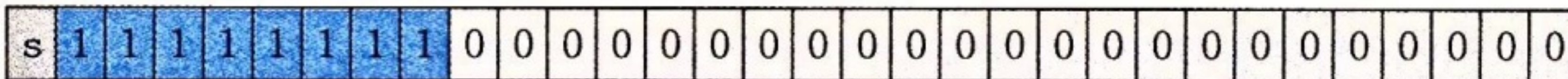
1. 规格化的



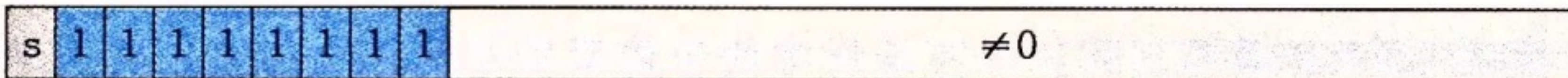
2. 非规格化的

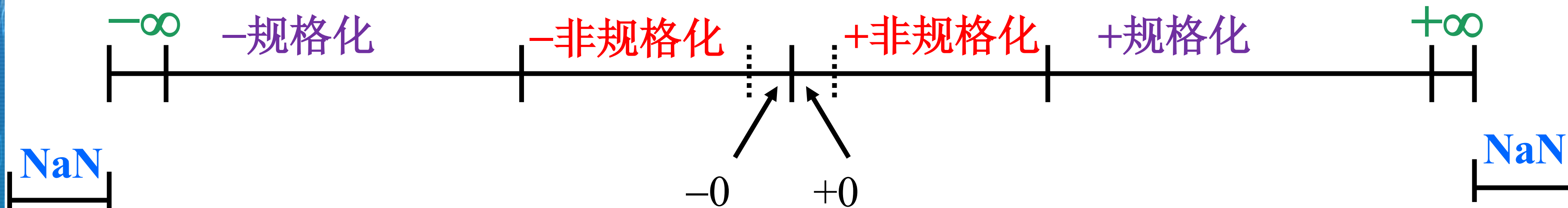


3a. 无穷大



3b. NaN

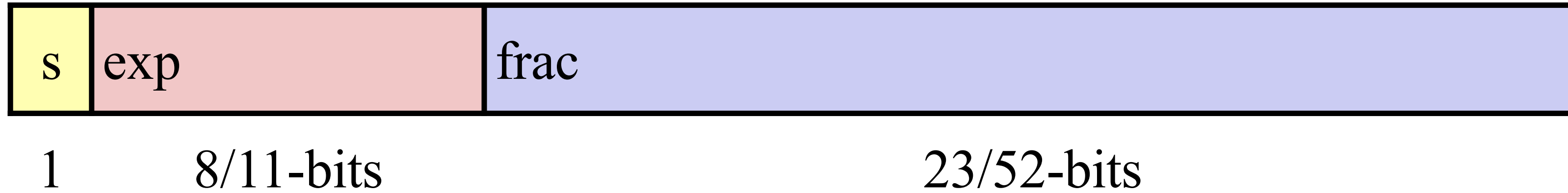




Normalized numbers（规格化数）

规格化数：+/-1.xxxxxxxxxx$_{two}$ x 2^{Exponent}

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



- exp ≠ 000…0 and exp ≠ 111…1
- exp = E+ Bias
  - exp:无符号数
  - 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.xxx…x$_2$ xxx…x: **frac**
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)

规定：规格化尾数最高位总是**1**，所以隐含表示，省**1**位

Normalized numbers（规格化数）

规格化数： $+/-1.xxxxxxxxxx_{two} \times 2^{Exponent}$

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

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

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;`

$15213_{10} = 11101101101101_2$

$= 1.1101101101101_2 \times 2^{13}$

v = $(-1)^s$ M 2^E
E = Exp – Bias

- **Significand**

M = `1.1101101101101`$_2$

`frac` = `11011011011010000000000`$_2$

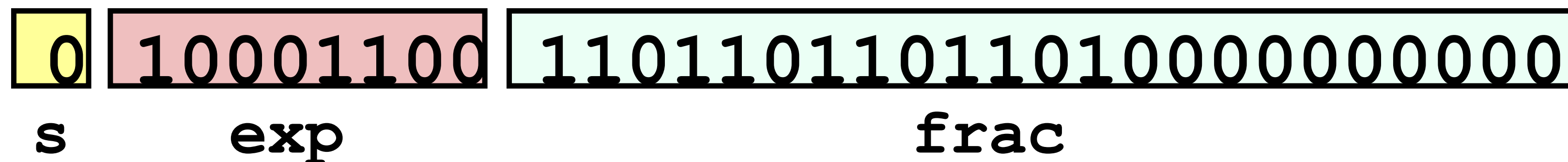
- **Exponent**

E = 13

$Bias$ = 127

Exp = 140 = `10001100`$_2$

- **Result:**



Denorm d numbers（非规格化数）

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

1 8/11-bits 23/52-bits

v = $(-1)^s$ M 2^E
E = 1 – Bias

- Condition: exp = 000…0
- E = 1 – Bias
  - 单精度: E: -126
  - 双精度: E: -1022
- M = 0.xxx…x xxx…x_2 : bits of **frac**
  - **exp** = 000…0, **frac** = 000…0
    - 代表数值 0，不区分+0和-0
  - **exp** = 000…0, **frac** ≠ 000…0
    - 接近 0.0的小数

特殊值

In FP, 除数为0的结果是 +/- ∞, 不是溢出异常.（整数除0为异常）

- **可以利用+∞/-∞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 = +∞, 1.0/−0.0 = −∞
- 情况2: **exp** = **111…1**, **frac** ≠ **000…0**
  - Not-a-Number (NaN)
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

X-86指令概述

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      mov   $0x12345678,%eax
40110f:     bb 78 56 34 12      mov   $0x12345678,%ebx
```

寻址方式

- 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

64-bit	32-bit	64-bit	32-bit
%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

64-bit	32-bit
%rip	%eip

Origin (mostly obsolete)

general purpose

32-bit	16-bit	High byte	Low byte	Origin
%eax	%ax	%ah	%al	accumulate
%ecx	%cx	%ch	%cl	counter
%edx	%dx	%dh	%dl	data
%ebx	%bx	%bh	%bl	base
%esi	%si			source index
%edi	%di			destination index
%esp	%sp			***stack pointer***
%ebp	%bp			***base pointer***
%eip	%ip			

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)	0xab
4(%rax)	0xab	9(%rax,%rdx)	0x11

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

单选题 2分

计算机系 设置

假设R[ax]=FFE8H, R[bx]=7FE6H,
执行指令“subw %bx, %ax”后，
寄存器内容和各标志的变化为

A R[ax]=8002H, OF=0, SF=1
CF=0, ZF=0

B R[ax]=8002H, OF=1, SF=1
CF=0, ZF=0

C R[bx]=8002H, OF=0, SF=1
CF=0, ZF=0

D R[bx]=8002H, OF=1, SF=1
CF=0, ZF=0

提交

单选题 1分

计算机系

设置

指令“popq %rbp”的功能是：

A R[rsp]=R[rsp]+8, R[rbp]=M[R[rsp]]

B R[rbp]=M[R[rsp]] R[rsp]=R[rsp]+8,

C R[rsp]=R[rsp]-8, R[rbp]=M[R[rsp]]

D R[rbp]=M[R[rsp]] R[rsp]=R[rsp]-8,

提交

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⩾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⩽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⩾B
15	jl/jnge label	SF≠OF AND ZF=	带符号整数 A<B
16	jle/jng label	SF≠OF OR ZF=1	带符号整数 A⩽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
```

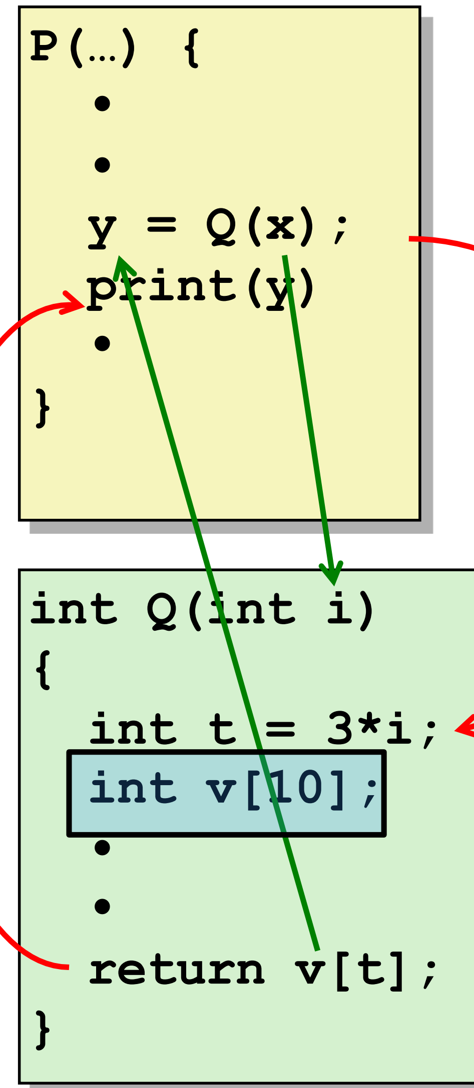
汇编器产生的“.o”格式的反汇编版本如下：

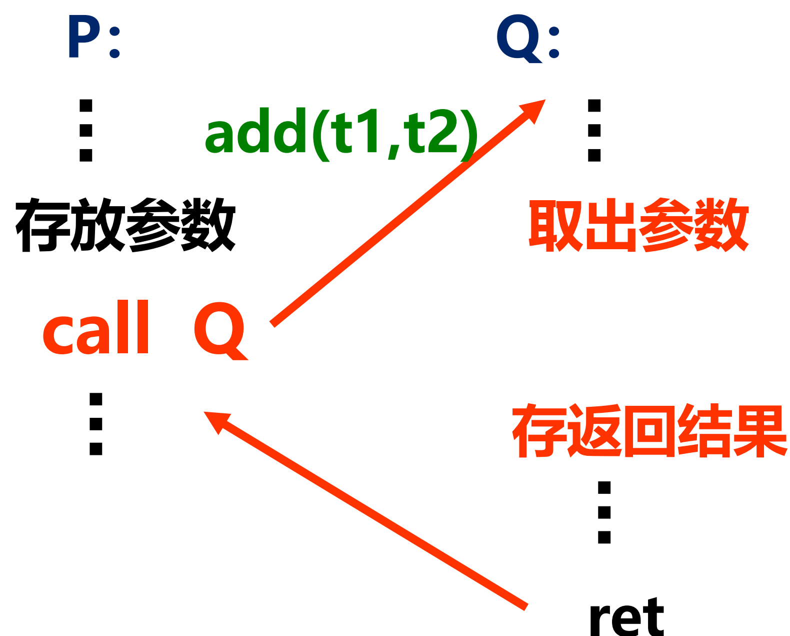
```
1    0:   48 89 f8        mov    %rdi,%rax
2    3:   eb 03           jmp    8 <loop+0x8>
3    5:   48 d1 f8        sar    %rax
4    8:   48 85 c0        test   %rax,%rax
5    b:   7f f8           jg     5 <loop+0x5>
6    d:   f3 c3           repz retq
```

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。





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

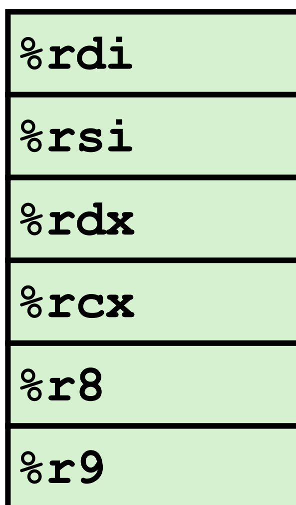
（1）从堆栈栈顶取出P的返回地址

（2）跳转到返回地址，即call Q指令后面的指令继续执行。

数据传送

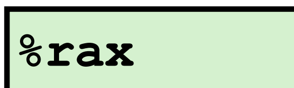
寄存器

- 通过寄存器最多传递6 个整型(例如整数和指针)参数



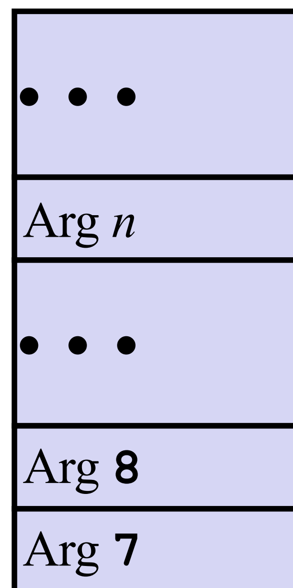
-

返回值



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

x86-64栈帧

- 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。

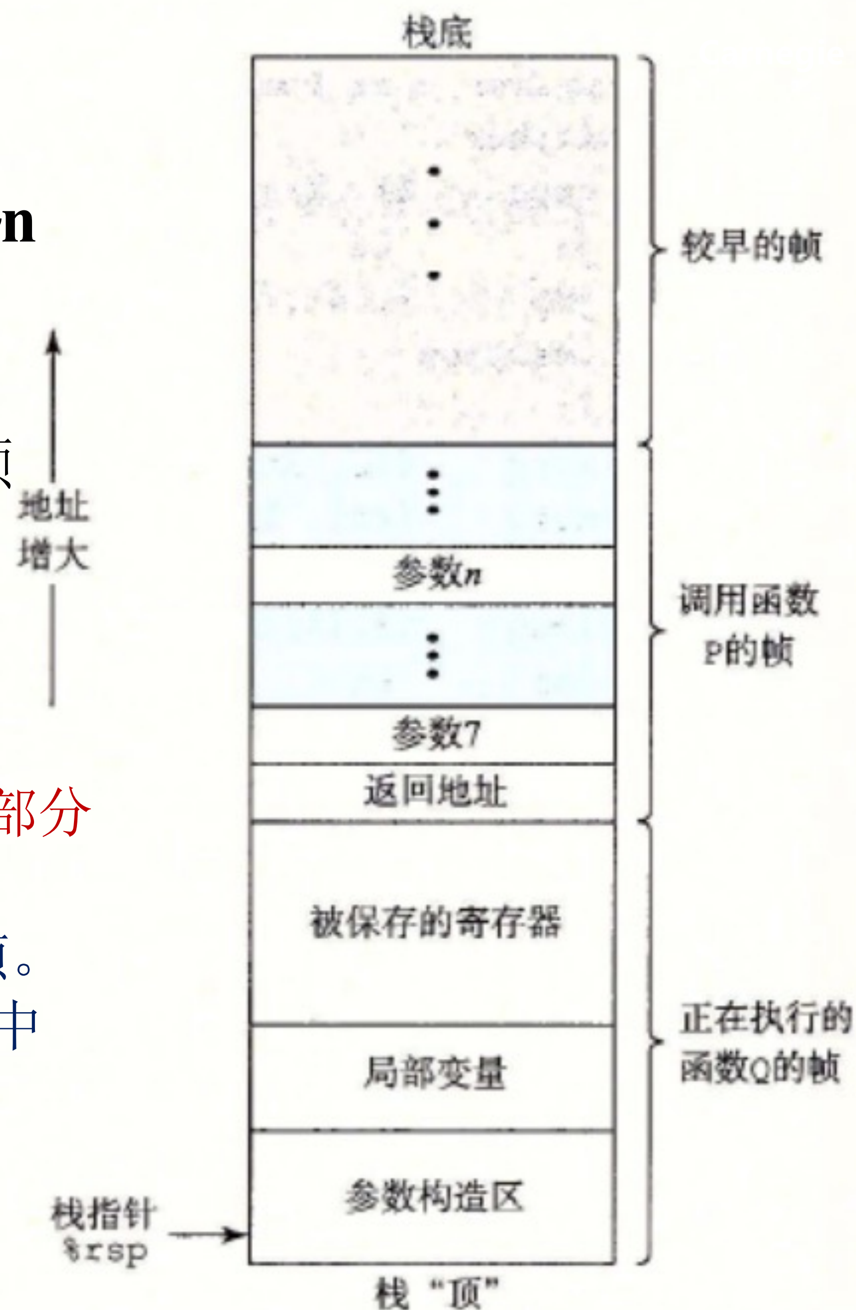
(2)局部变量使用地址运算符‘&'，必须为它产生一个地址。

(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≤i≤3）个元素的地址计算公式:&A[0]+ sizeof(T) *i

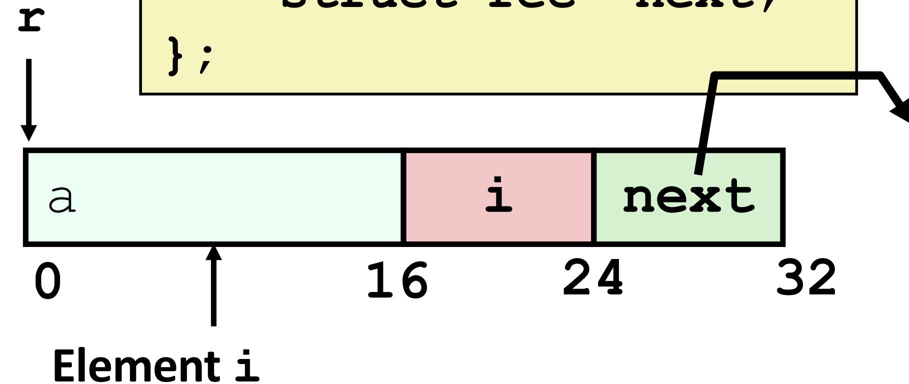
```
main ( )
{
   static short num[ ][4]={ {2, 9, -1, 5},
                           {3, 8, 2, -6}};
   static short *pn[ ]={num[0], num[1]};
   static short s[2]={0, 0};
```

```
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
```

结构的分配和访问

- C Code

```
struct rec {
    int a[4];
    int i;
    struct rec *next;
};
```




```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    int i = r->i;
    r->a[i] = val;
    r = r->next;
  }
}
```

Register	Value
%rdi	r
%rsi	val

```
       jmp .L22                        # loop:
.L11   movslq  16(%rdi), %rax          #   i = M[r+16]
       movl    %esi, (%rdi,%rax,4)     #   M[r+4*i] = val
       movq    24(%rdi), %rdi          #   r = M[r+24]
.L22:  testq   %rdi, %rdi              #   Test r
       jne     .L11                    #   if !=0 goto loop
```

代码注入攻击Code Injection Attacks

```
void P(){
  Q();
  ...
}
```

```
int Q() {
  char buf[64];
  gets(buf);
  ...
  return ...;
}
```


- 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

过程中的浮点代码

在x86 -6 4 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

三类目标文件

- 可重定位目标文件 (.o)
  - 其代码和数据可和其他可重定位文件合并为可执行文件
    - 每个.o 文件由对应的.c文件生成
    - 每个.o文件代码和数据地址都从0开始
- 可执行目标文件 (默认为a.out)
  - 包含的代码和数据可以被直接复制到内存并被执行
  - 代码和数据地址为虚拟地址空间中的地址
- 共享的目标文件 (.so)
  - 特殊的可重定位目标文件，能在装入或运行时被装入到内存并自动被链接，称为共享库文件
  - Windows 中称其为 *Dynamic Link Libraries* (DLLs)

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)

链接操作的步骤

- 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

符号和符号解析

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

符号和符号解析

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**？

静态库的创建

atoi.c → 转换 (cpp,cc1,as) → atoi.o
printf.c → 转换 (cpp,cc1,as) → printf.o
…
random.c → 转换 (cpp,cc1,as) → random.o

atoi.o, printf.o, random.o → Archiver (ar) → libc.a C标准静态库

```
$ ar rcs libc.a \
  atoi.o printf.o … random.o
```

- 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命令行中无需明显指定C标准库libc.a(默认库)

链接顺序问题

- 假设调用关系如下：

  func.o → libx.a 和 liby.a 中的函数

  libx.a → 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 → libx.a 和 liby.a 中的函数

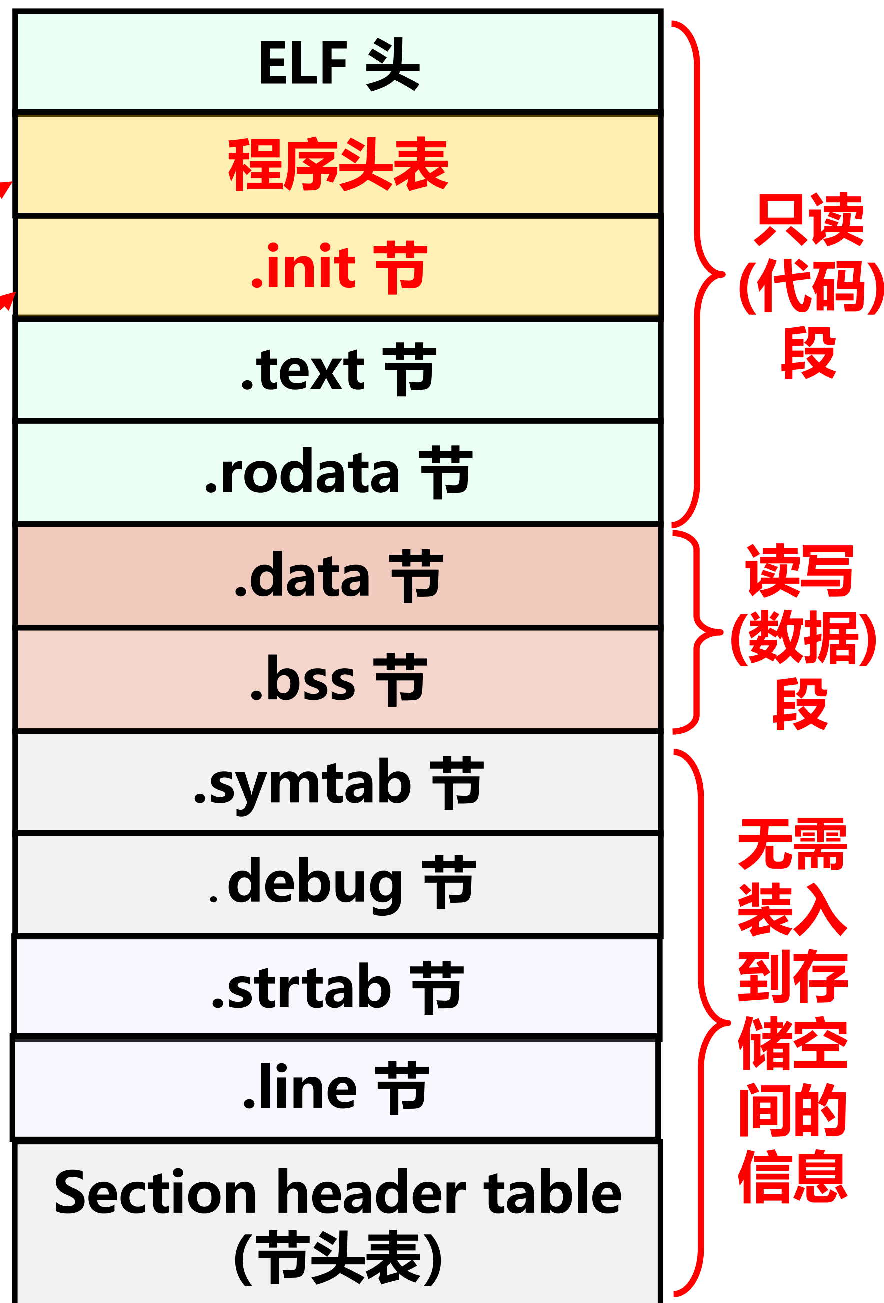
  libx.a → liby.a 同时 liby.a → libx.a

  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

可执行目标文件格式

- **与可重定位文件稍有不同：**
  - **ELF头中字段e_entry给出执行程序时第一条指令的地址，而在可重定位文件中，此字段为0**
  - **多一个程序头表，也称段头表（segment header table），是一个结构数组**
  - **多一个.init节，用于定义_init函数，该函数用来进行可执行目标文件开始执行时的初始化工作**
  - **少两个.rel节（无需重定位）**



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            endbr64
  40117a:        ba 01 00 00 00         mov    $0x1,%edx
  40117f:        b9 01 00 00 00         mov    $0x1,%ecx
  401184:        b8 00 00 00 00         mov    $0x0,%eax
  401189:        eb 07                  jmp    401192 <f1+0x1c>
  40118b:        01 d2                  add    %edx,%edx
  40118d:        01 d1                  add    %edx,%ecx
  40118f:        83 c0 01               add    $0x1,%eax
  401192:        39 f8                  cmp    %edi,%eax
  401194:        7c f5                  jl     40118b <f1+0x15>
  401196:        89 c8                  mov    %ecx,%eax
  401198:        c3                     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: 0x00000000004011cb      0x0000000400000000
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？