

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



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

$$\text{CPI} = \text{执行程序所需的时钟周期数} / \text{IC}$$

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 $\times$ 时钟周期时间

= 执行程序所需的时钟周期数 / 时钟频率

= $\text{IC} \times \text{CPI} / \text{时钟频率}$

CPU的性能取决于三个参数: **时钟频率, CPI, IC**

假定处理器 P 的主频为 3GHz。CPI 为 1.5，P 运行某个程序的执行时间为 10s，该程序执行的时钟周期数和指令数分别是 (B)。

1. 时钟周期数 = 主频 × 执行时间

$$= 3 \times 10^9 \times 10$$

$$= 3 \times 10^{10}$$

(A)

(B)

 $3 \times 10^9, 2 \times 10^{10}$

(C)

 $3 \times 10^{10}, 2 \times 10^{10}$

(D)

 $3 \times 10^{10}, 2 \times 10^9$

$$3\text{GHz} = 3 \times 10^9 \text{ Hz}$$

(每秒有 3×10^9 个时钟周期)

$$\text{CPI} = 1.5$$

每条指令平均需 1.5 个时钟周期

2. 指令数

$$B = \frac{\text{时钟周期数}}{1.5} = \frac{3 \times 10^{10}}{1.5} = 2 \times 10^{10}$$

P: 执行时间 10s

提交



(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/O设备的抽象

(4) 进程

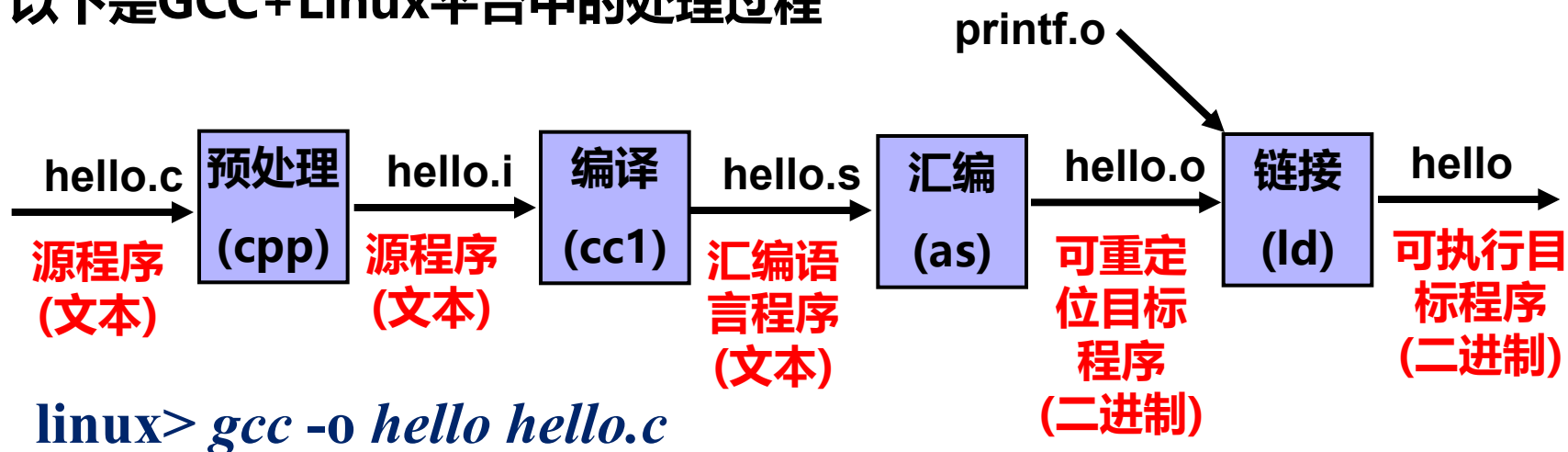
提供正在运行的程序的抽象。包括对处理器、主存和I/O设备的抽象表示

(5) 虚拟机

提供了对整个计算机抽象

高级语言程序开发过程

以下是GCC+Linux平台中的处理过程

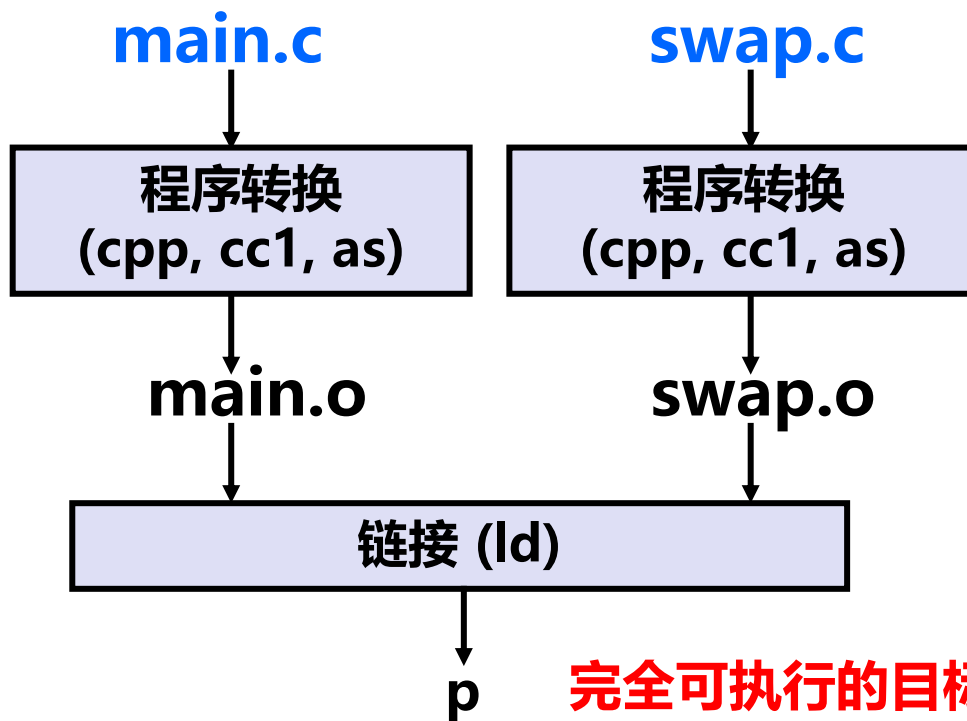




可执行文件的生成

- 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



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

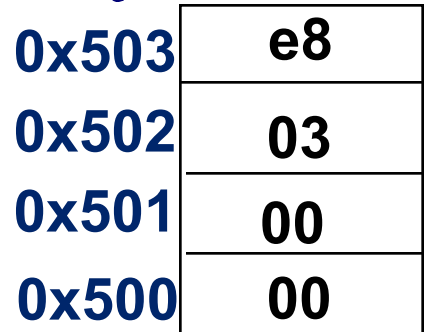
在几乎所有的机器上，多字节对象都被存储为连续的字节序列，对象的地址为所使用字节中最小的地址。

例：unsigned int x=1000; ^{4字节} **&i的值: 0x500**
32位: ^{高位} 0000 0000 0000 0000 0000 0011 1110 1000 ^{低位} (0x000003e8)



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

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 (C)

A 0x12

B 0x34

C 0x56

D 0x78

12	C003	高
34	C002	
56	C001	↓
78	C000	低

提交



unsigned 都正

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。

正数补码 = 原码
负数补码 = 正数码取反 + 1

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:

① 先转+1000 的32位

0000 0000 0000 0000 0000 0011 1110 1000

② 取反

int x=-1000; 1111 1111 1111 1111 1111 1100 0001 0111

③ 加1

32位:

rsp=0xf5dd1550

FF
FF
FA
18

商 12位
↓
低

1111 1111 F 1111 F 1111 F 1111 F 1111 A 1111 8 0001 1000

(0xFFFFFA18)

4个1字节

1个4字节

GDB x/4xb 0xf5dd1550

x/1xw 0xf5dd1550

GDB x/4xb \$rsp

0xf5dd1550 :18 fa ff ff

0xf5dd1550 :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 = 1字节	-128	127
short	16 = 2字节	-32768	32767
int	32 = 4字节	-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）



无符号数和有符号数的转换

计算机系统基础I

-1的32位补码 111 111 111 111 111 111 111 111

考虑以下C代码: 输出: %d: (按有符号int读) 这作为补码 $= -2^{31} + 2^{30} + \dots + 2^0 = -1$

1 int x = -1;

%u: 按无符号读 $= 2^{32} - 1 = 4294967295$

2 unsigned u = 2147483648;

∴有2种答案

3 printf ("x = %u = %d\n" , x, x);

$2^{31} = 214783648$ 这是32位无符号数能求的

4 printf ("u = %u = %d\n" , u, u);

$2^{31} = 1000 \ 0000 \dots 0000$
8位数字

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

输出: 按 %d 有符号读: 最高为1 $\rightarrow$ 补 = $-2^{31} + 0 = -214783648$

按 %u 无符号读: $2^{31} = 214783648$

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; (有) 32位全1
```

```
unsigned ux = 0xffffffff; (无)
```

```
tx = ux; /*Cast to signed */ ☆ 隐式转换, 无符号转有符号
```

```
printf("tx=%d, ux=%u\n", tx, ux);
```

它的输出结果是什么？

$tx = -1, ux = 4294967295$

都是32位全1 但 $\left\{ \begin{array}{l} tx \text{ 有符号} = -1 \\ ux \text{ 无符号} = 4294967295 \end{array} \right.$

C语言：位模式被
只是解释为位



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 定点小数

– **Exponent**指数 E 定点整数

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

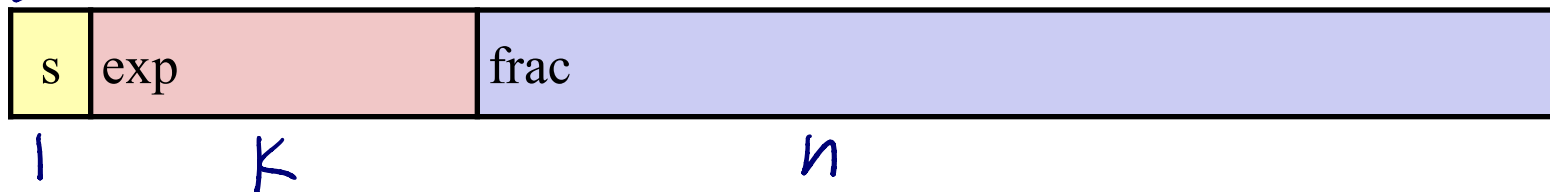
• 编码

– MSB s 是符号位

– exp 编码 E (but is not equal to E)

– frac 编码 M (but is not equal to M)

符号位





C语言中的浮点变量

$$\text{Bias} = 2^{k-1} - 1 \quad (k \text{ 为 } \text{exp} \text{ 位数})$$

$$\text{Bias}_{\text{float}} = 2^7 - 1 = 127$$

$$\text{Bias}_{\text{double}} = 2^{10} - 1 = 1023$$

- 单精度: 32 bits float



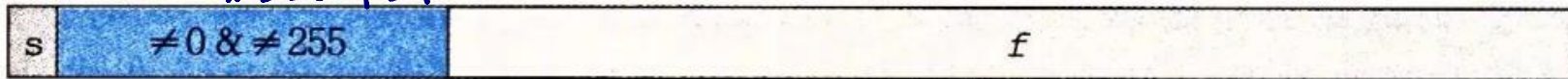
- 双精度: 64 bits double



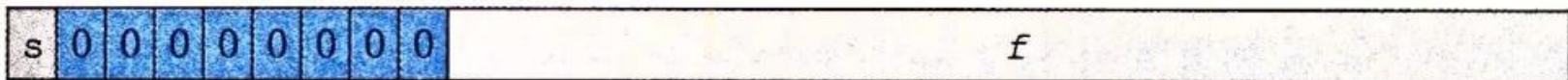


根据 exp 的值，被编码的值可以分成三种不同的情况

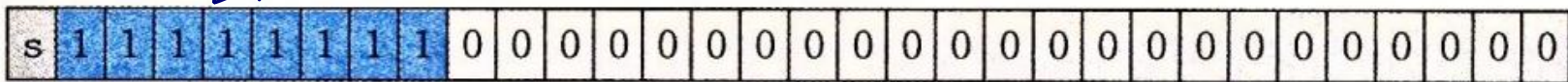
1. 规格化的 ~~非全0也非全1~~



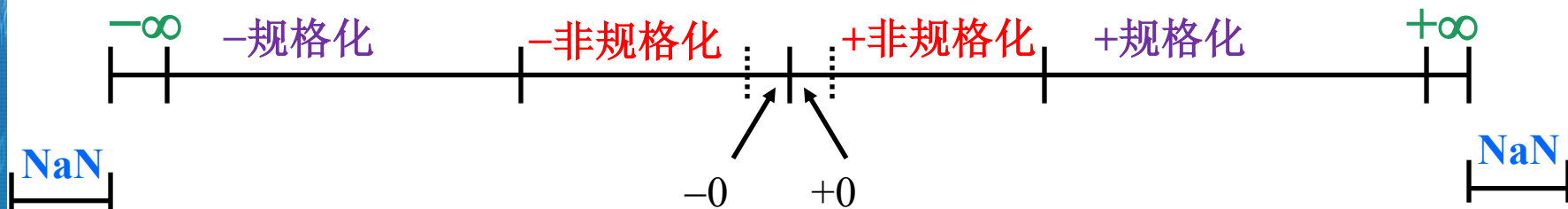
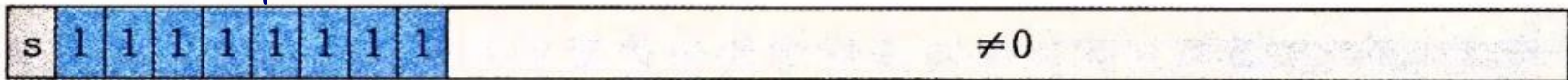
2. 非规格化的 ~~全0~~



3a. 无穷大 ~~全1~~



3b. NaN ~~全1~~





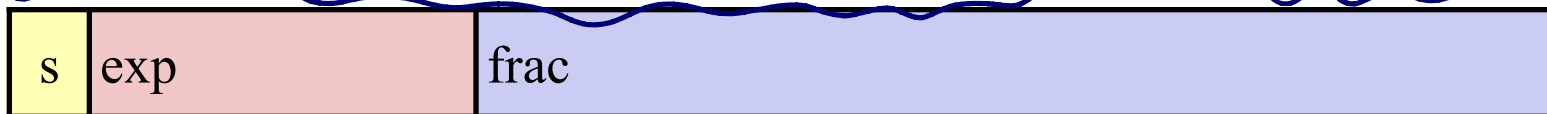
Normalized numbers (规格化数)

计算机系统基础I

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

前面是这样 $\pm 1.xxx$ 后面 $\times 2^E$

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



1

8/11-bits

23/52-bits

- $exp \neq 000...0$ and $exp \neq 111...1$ (非0也非1)
- $exp = E + Bias$ (规格化后二进制指数永远为1)

– exp : 无符号数

– $Bias = 2^{k-1} - 1$, $k=8$ 或 11 是 exp 的位数

exp 范围

- 单精度: 127 (exp : 1...254, E : -126...127)
- 双精度: 1023 (exp : 1...2046, E : -1022...1023)

单精度 $frac$ 用 23 位
但能精度表示 24 位有效数字
双精度 $frac$ 用 52 位
能表示 53 位有效数字

• $M = 1.xxx...x_2$ $xxx...x$: $frac$

– Minimum when $frac=000...0$ ($M = 1.0$)

$\therefore M \in [1.0, 2.0)$

– Maximum when $frac=111...1$ ($M = 2.0 - \epsilon$)

规定: 规格化尾数最高位总是1, 所以隐含表示, 省1位

就不存, 省下给 $frac$ 用一位
实际数 $m = 1.xxx$ 存储 $frac = xxx$



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



1 bit
 8/11-bits
 23/52-bits
exp 范围
↓ 指数
浮点 M (把小数加回, 小数 frac)
Bias
E = exp - Bias

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 $F = 15213.0$;

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

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

$$15213_{10} = 11101101101101_2$$

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

① 先二进制然后格式化成 $1.x \cdot 2^E$

② 根据格式化成 E, frac

③ 算 $\text{exp} = E + \text{bias} \rightarrow$ 二进制

④ frac 补到23位 (左补0)

⑤ 拼接 $s, \text{Exp}, \text{frac}$
4位一分 然得十六进制

• Significand

frac

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

$$\text{frac} = \underline{11011011011010000000000}_2$$

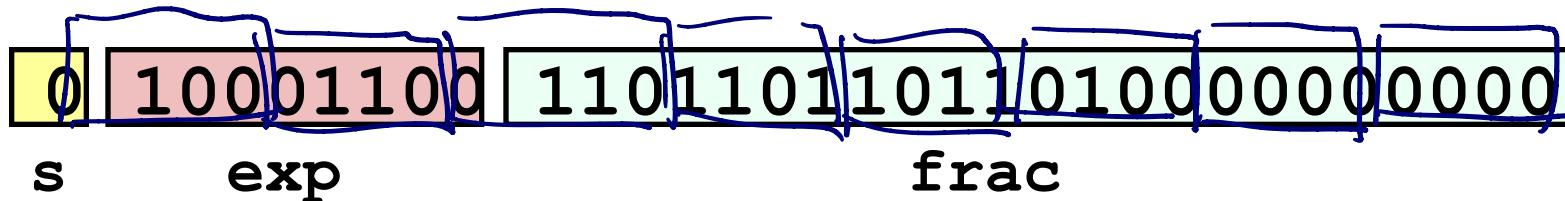
• Exponent

$$E = 13$$

$$\text{Bias} = 127$$

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

• Result: (1+8+23)





Denorm d numbers (非规格化数) 计算机系统基础I

非规格化数: $\boxed{+/-0.\text{xxxxxxxxxxx}_{\text{two}} \times 2^{\text{Exponent}}}$ (补成最小指数0的阶码)



1 8/11-bits

23/52-bits

- Condition: exp = 000...0

- $E = 1 - \text{Bias}$ 注意规格化是 $E = \text{exp} - \text{Bias}$

- 单精度: E: -126

- 双精度: E: -1022

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

– **exp** = 000...0, **frac** = 000...0

- 代表数值 0, 不区分+0和-0

– **exp** = 000...0, **frac** $\neq$ 000...0

- 接近 0.0的小数

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

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

eg 0 00000000 00000000...1
 $s=0$ $\text{frac} = 0.0\cdots1$
 $M = 0.0\cdots1 = 2^{-23}$
 $E = -126$
 $V = 0.00\cdots01 \times 2^{-126}$
 $= 2^{-23} \times 2^{-126}$
 $= 2^{-149}$

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

• $\text{exp} = 111\dots 1$ ^{exp 全1 特殊值} $\left\{ \begin{array}{l} \text{frac 全0} \rightarrow \pm\infty \\ \text{frac 非0} \rightarrow \text{NaN 非数} \end{array} \right.$

• 情况1: $\text{exp} = 111\dots 1$, $\text{frac} = 000\dots 0$

– 代表 ∞

– E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

• 情况2: $\text{exp} = 111\dots 1$, $\text{frac} \neq 000\dots 0$

– Not-a-Number (NaN) ^{不等于任何数, 且任何数与NaN比较仍为NaN}

– E.g., $\text{sqrt}(-1)$, $\infty - \infty$, $\infty \times 0$ ^{数浮上极限}



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。

↓ 下一条指令
 40110a: 对字节
 40110f: 对字节

	高	低
b8	78	56 34 12
bb	78	56 34 12

操作码

告诉CPU这是mov指令
 目标是eax/ebx

mov \$0x12345678,%eax (4字节版)
 mov \$0x12345678,%ebx

把数字 0x12345678 装到寄存器%eax

每个mov指令占5字节 (1字节操作码 + 4字节立即数)

地址

机器码(5字节)

汇编指令(人读)

- 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 寻址方式

- ① – 指令中：立即寻址

eg
\$0x10

- ② – 寄存器中：寄存器寻址

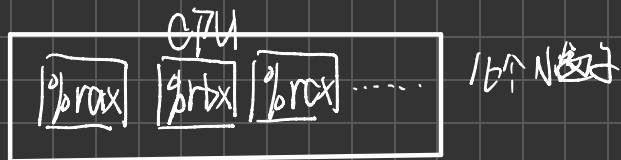
%rax

- ③ – 存储单元中（属于存储器操作数，按字节编址）：
（在内存里）
(%rax), 8(%rax, %rdx/4)

- ④ – 端口中

寄存器

CPU 内部小盒子, 用来临时存数据



每个盒子有一个名字: 比如 %rax

能装 8 个字节的数据

eg %rax 装着: 0x123456789ABCDEF (16位十六进制)

CPU 做运算时只能操作寄存器里的数据 | 所以要经常从内存搬进寄存器, 算完搬回内存)

% 是什么: AT 汇编语法的标记, 告诉这是寄存器名, 区别于普通数

eg: %rax → 寄存器 rax

rax → 不规范写法

0x100 内存地址或数字

#0x100 → 立即数(常数)

一个盒子里



高位 → 低位

不同名字访问不同部分 (eg: 0x123456789ABCDEF)

%rax → 全部 = 0x123456789ABCDEF (8字节)

%eax → 低4字节 = 0x9ABCDEF (低4字节)

%ax → 低2字节 = 0xCDEF (低2字节)

特殊 { %ah → 第2字节 = 0xCD (第2字节)

%al → 最低字节 = 0xEF (最低字节)

指令 命令 CPU

eg: `movq %rbx, %rax`
 ↑ ↑ ↑
 动 源 目标
 词 址 址

二把 %rbx 值复制到 %rax 里
左流向右 (ATT)

四、看到指令怎么“读”

第一步：找动词

Code



```
movq %rbx, %rax    ← mov (搬动)    + q (8字节)
addq %rbx, %rax    ← add (加)       + q (8字节)
subq %rbx, %rax    ← sub (减)       + q (8字节)
```

`decq` 减 1

动词带后缀 (b/w/l/q) 告诉你“操作几字节”:

- b = 1字节
- w = 2字节
- l = 4字节
- q = 8字节



第二步：找操作数

操作数有 3 种形式:

写法	是什么	例子
<code>%xxx</code>	寄存器 (取盒子里的值)	<code>%rax</code>
<code>\$xxx</code>	立即数 (就是这个常数)	<code>\$5</code> 、 <code>\$0x100</code>
<code>(...)</code> 或 <code>数字+(...)</code>	内存 (去内存某地址取值)	<code>(%rax)</code> 、 <code>8(%rax)</code>

第三步：执行

按动词把事情做了。

eg: `movq $5, %rax`

搬 8 字节, 立即数 5 装进 %rax 寄存器

`movq (%rbx), %rax`

↓
去内存里 rbx 指向地方取值

把 rbx 当成地址去内存那个地址读 8 字节 放到 rax

`addq %rbx, %rax`

把 rbx 加到 rax 上 (rbx 不变 rax 变)



x86-64 整数寄存器

16个通用整数寄存器

%rax	32位版本 %eax
------	------------------------------

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

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

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

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

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

普通寄存器
32位版本

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

特殊
(栈指针)

%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)**



存储器操作数寻址

通用公式 $EA = Imm + R[r_b] + R[r_i] \times S$

Imm: 立即数偏移(带符号) S: 比例因子 1, 2, 4, 8 之一

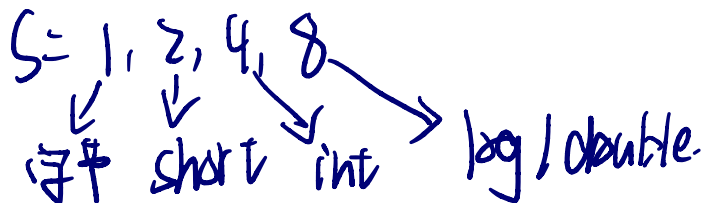
r_b : 基址寄存器

r_i : 变址寄存器

类型

写法	有效地址	寻址方式
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

都是通用写法表示





寻址方式举例

寄存器寻址：直接取寄存器本身
不带偏移也不带标尺 (0x104, %rax)

计算机系统基础I

寄存器寻址 → 寄存器寻址
绝对寻址 → 绝对寻址

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

寄存器	值
rax	0x100
rcx	0x1
rdx	0x3

寄存器寻址：直接取寄存器本身

寄存器寻址：直接取寄存器本身

操作数	值	操作数	值
%rax (寄存器寻址)	0x100	9(%rax,%rdx)	0x11
0x104 (绝对寻址)	0xab	260(%rcx,%rdx)	0x13
\$0x108 (立即寻址)	0x108	0xFC(,%rcx,4)	0xff
(%rax) (寄存器寻址)	0xff	(%rax,%rcx,4)	0xab
4(%rax) (寄存器寻址)	0xab	9(%rax,%rdx)	0x11

rax 地址

取寄存器本身

间接寻址

$M[4 \times 0x100]$
 $M[0x104] =$

$= M[R[ra]] = M[0x100] = 0xff$

① $M[9 + 0x100 + 0x3] = M[0x10c] = 0x11$

② $M[260 + 0x1 + 0x3] = M[0x104 + 0x1 + 0x3] = M[0x108] = 0x13$

转16

$0xFC(,%rcx,4)$
 $= M[0xFC + 0x1 \times 4] = M[0x100] = 0xff$

$(%rax,%rcx,4)$
 $= M[0x100 + 0x1 \times 4] = M[0x104] = 0xab$

$9(%rax,%rdx)$
 $= M[9 + 0x100 + 0x3] = M[0x10c] = 0x11$



↑ 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

16 24 16 8 4 2 0

包括其他指令，如LEA，逻辑运算、移位指令等

指令	目的	值
sarq (除2) 指令		
addq %rcx, (%rax) rcx值加在rax地址值	目的 $R[100] = 0x100$ 内存地址: 0x100	$0x1 + M[0x100] = 0x1 + 0xFF = 0x100$ 0x100
subq %rdx, 8(%rax) 减 被减数	$M[8 + 0x100] = M[0x108] = 0xAB$ 内存地址: 0x108	$0xAB - 0x3 = 0xA8$ 0xA8
imulq \$16, (%rax, %rdx, 8) 乘	$M[0x100 + 8 \times 0x3] = M[0x118] = 0x11$ 内存地址: 0x118	$0x11 \times 16 = 0x110$ 0x110
incq 16(%rax) 有 一个操作数 +1	$M[16 + 0x100] = M[0x110] = 0x13$ 内存地址: 0x110	$0x13 + 1 = 0x14$ 0x14
decq %rcx 减 -1	没拖寄存器 = 0x1 %rcx	$0x1 - 1 = 0x0$ 0x0
subq %rdx, %rax 减	%rax	$0x100 - 0x3 = 0xFFD$ 0xFFD

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

①转二进制

$$ax = 0xFFE8 = 1111 \ 1111 \ 1110 \ 1000$$

$$bx = 0x7FE6 = 0111 \ 1111 \ 1110 \ 0110$$

②: 减法 = 加上 -bx 补码 (CPU不直接加法) $-bx = bx \text{取反} + 1$

-bx 补码 (取反+1)

$$\begin{array}{r} 1000 \ 0000 \ 0001 \ 1001 \\ + 1 \\ \hline 1000 \ 0000 \ 0001 \ 1010 \end{array}$$

$\therefore -bx \text{补码 } 0x801A$

$$\begin{array}{r} 1111 \ 1111 \ 1110 \ 1000 \ (ax) \\ 1000 \ 0000 \ 0001 \ 1010 \ (-bx) \\ \hline 11000 \ 0000 \ 0000 \ 0010 \end{array}$$

$\therefore$ 取低16位 (进位不取)

$$ax = 0x8002$$

CF=0 (加法进) $\therefore$ 减法没进 ZF (零标志) = 0 SF=1 (最高位为1)

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

取反+1
OF=0 (溢出)
ax取 bx正
结果不为0

提交

能是:

② $rsp += 8$ 再把栈顶向上挪8字节

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

② $M[rsp] = rax$ (把 rax 值写到栈顶)

eg

1	0x00000008
2	0x00000000 ← rsp 7 内存地址

假设 max = 0x1234

① $rsp -= 8$
 $rsp / 4 \times 0x00000000 \rightarrow 0x00000008$

② $M[rsp] = max = 0x1234$

1	0x00000008
2	0x1234

提交





跳转指令 → 程序执行流

— 无条件跳转指令

jmp ^{目标} 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 (需要看CPU中条件码寄存器)

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≠OF AND ZF=	带符号整数 $A < B$
16	jle/jng label	SF≠OF OR ZF=1	带符号整数 $A \leq B$

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 把参数从 %rdi 复制到 %rax
2      jmp    .L2         无条件跳转 L2 (第一次跳转, 对应下面第二行 jmp)
3      .L3:
4      sarq    %rax        算术右移 (除以2)
5      .L2:
6      testq   %rax, %rax 测试 %rax 是否为 0 或正
7      lg     .L3         条件跳转, 如果满足 jg 条件, 就跳回上面 .L3 处
8      rep; ret
    
```

指令格式:

(不断/2) ① 赋值 ② 无条件跳 ③ test, %rax > 0 ④ 条件跳

$\left\{ \begin{array}{l} \%rax > 0 \text{ ① 除以2} \\ \%rax \leq 0 \text{ ② 返回} \end{array} \right.$

准备传参 (%rdi 值复制到 %rax)

汇编器产生的“.o”格式的反汇编版本如下:

1	3→5 占2个	0:	48 89 f8	mov	%rdi,%rax	①
2	把下一条指令地址 + 偏移量 = 8	3:	eb 03	无条件	jmp 8 <loop+0x8>	②
3	地址 + 偏移量 (3)	5:	48 d1 f8	sar	(除以2) %rax	③ (A 情况)
4	跳回后	8:	48 85 c0	test	%rax,%rax	③ 值与 0 进行比较
5		b:	7f f8	jg	5 <loop+0x5>	④ (B 情况)
6		d:	f3 c3	repz	retq	④ 如果为 0 跳转, 检查下一条指令

在内存中起始地址 机器码 (操作码) 偏移量

④: $\left\{ \begin{array}{l} A: \%rax > 0, \text{ 满足, 跳转到地址 5} \\ B: \%rax \leq 0, \text{ 不满足, 不发生跳转, 顺序下一条} \end{array} \right.$



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

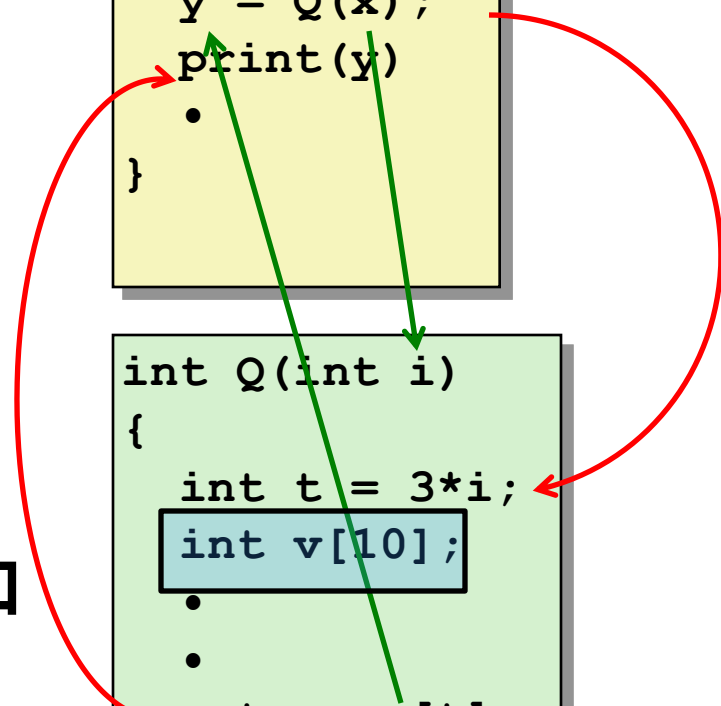
- 内存的分配和释放

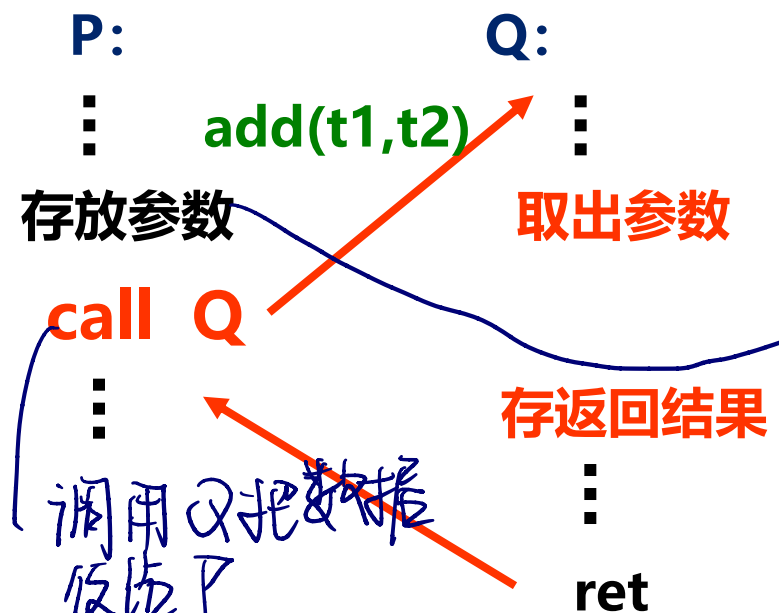
为局部变量分配空间及释放

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

先把传进来的参数放到指定寄存器或堆栈里

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 [P调用传参, 优先用寄存器, 然后堆栈] 计算机系统基础I

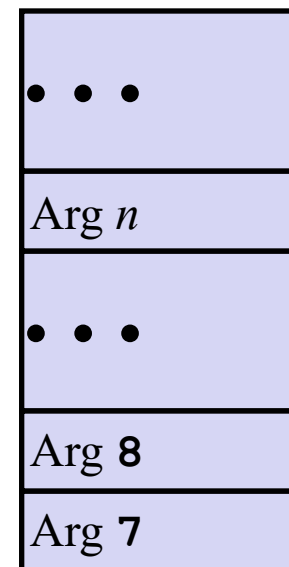
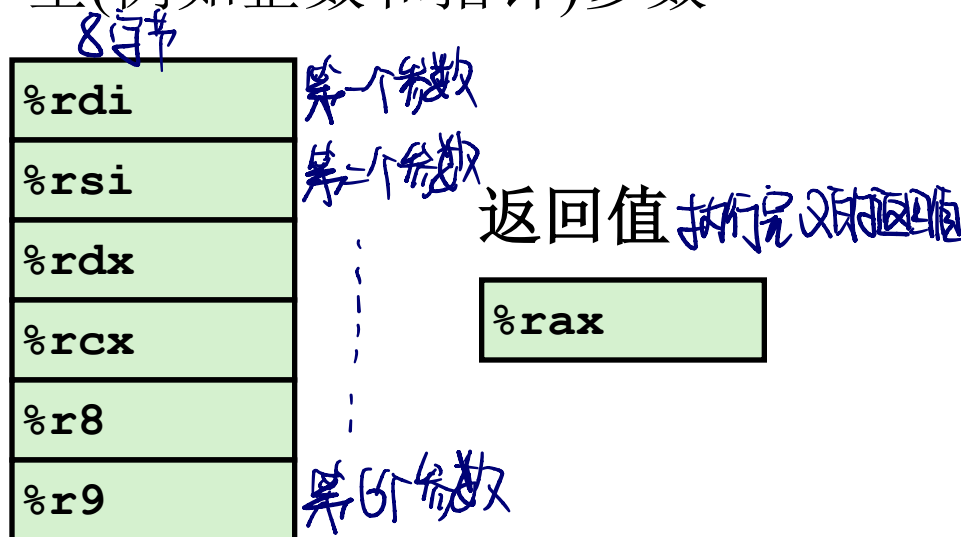
寄存器

~~x86-64~~ 参数

堆栈

- 通过寄存器最多传递6个整型(例如整数和指针)参数

- 需要的时候分配堆栈空间



超过6个有堆栈
1. 必须提前在栈帧里为这些超出的参数分配好空间)

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

栈生长方向：向低地址
寄存器 %rsp (栈指针) 极其关键
永远指向当前栈帧的栈顶(即地址最低)

计算机系统基础I

• 内容

- 返回地址
- 调用另一个过程设置的参数 7~n
- 局部变量

以下三种情况,才在堆栈中分配栈帧)

- (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。eg: $\text{int}^*p = \&x$
 - (3) 局部变量是数组或结构
- 被保存的现场寄存器

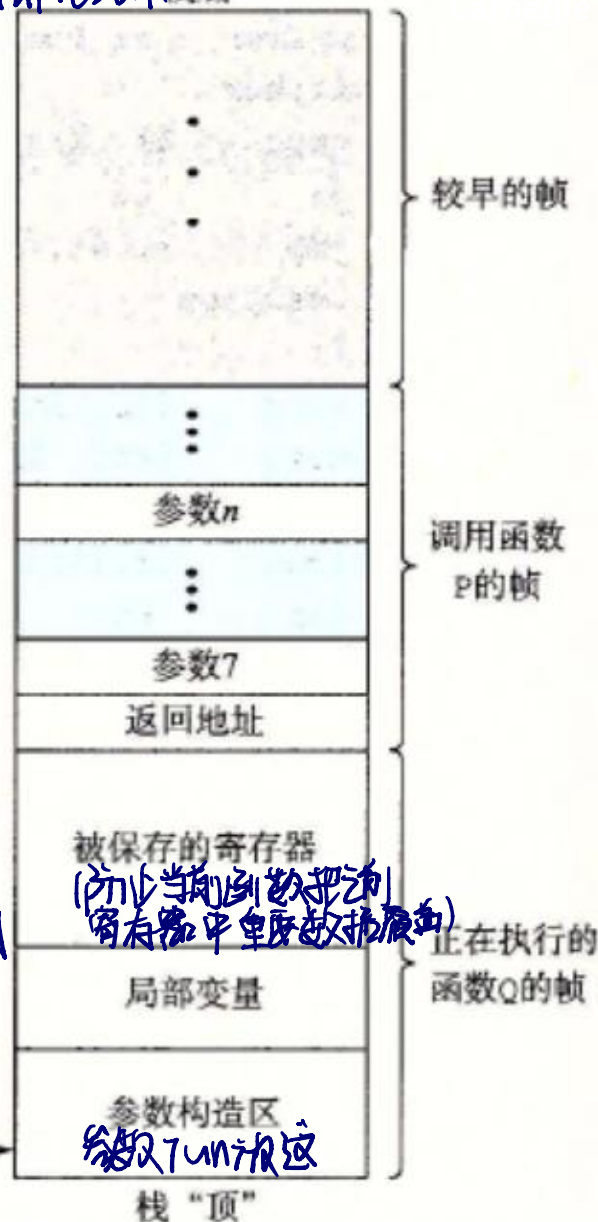
X86-64 过程只分配自己所需要的栈帧部分

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。

叶子函数
内部不调用其他函数

防止当前函数把调用者寄存器中重要数据覆盖掉

在地址 栈指针 %rsp





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 $0 \sim N-1$ 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: $\&A[0] + \text{sizeof}(T) * i$

首地址 一个大N * 第i个

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};
}

```

二维数组

short 占 2 字节
指针 占 4 字节 (32位)

4 字节 指向数组

一维数组

08049300 <num>:

1 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

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

08049310 <pn>: 2 9 -1 (补码) 5 3 8 2 -6

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

08049320 <s>: 08049300

08049300

08049320: 00 00 00 00

中间... 由于编译器
做内存对齐



结构的分配和访问

将不同类型的数据凑一起

机器级管理结构体的原则:

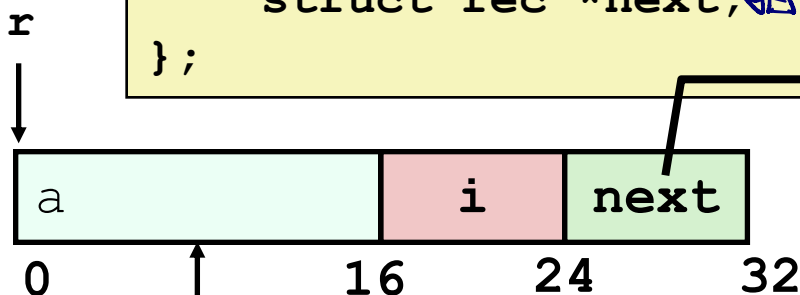
- ① 结构体在内存中也是连续排列的
- ② 通过相对于结构体起始地址的偏移量来访问结构体

• C Code

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

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

数组占 $4 \times 4 = 16$ 字节(偏)
 整型占 4 字节(偏为 16)
 指针(偏格 16+8=24)



Element i

8 字节指针 next 起始地址
 必须 8 位数
 $\therefore$ 本: $16 + 4 = 20$
 但跳 21, 22, 23
 从 24 开始

Register	Value
%rdi	r
%rsi	val

结构体指针 16 字节地址值

```
.L11    jmp .L22
        movslq 16(%rdi), %rax
        movl   %esi, (%rdi,%rax,4)
        movq   24(%rdi), %rdi
.L22:   testq  %rdi, %rdi
        jne    .L11
```

loop:
 # $\rightarrow$ $i = M[r+16]$
 # $\rightarrow$ $M[r+4*i] = val$
 # $\rightarrow$ $r = M[r+24]$
 # $\rightarrow$ Test r
 # $\rightarrow$ if !=0 goto loop

继续到下一个循环

不为 0 代表还未到头



代码注入攻击 Code Injection Attacks

向大小固定的缓冲区写入超出其容量的数据
就会发生缓冲区溢出，由于栈帧上，会破坏栈上的关键
信息，
有变量
控制信息、

Stack after call to `gets()`

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

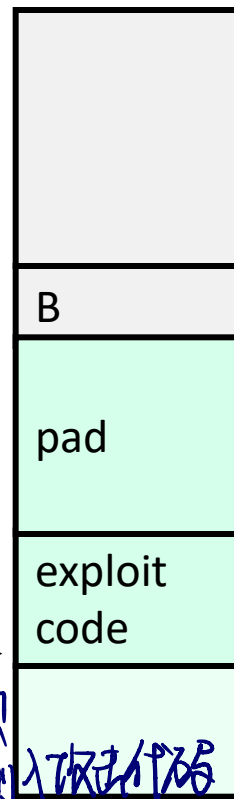
return
address
A

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

64字节的缓冲区
gets() 不受限制输入
长数据，危险。

data written
by `gets()`

返回B值假栈顶
后开始劫持输入攻击代码



P stack frame

Q stack frame

清除方向

get 不设定，多输入
后会有何地址
一路清除跨越
缓冲区边界

- 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x86-64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：^(2个参数)
`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

通用寄存器前两位第一位

通用寄存器第二位

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

x 放在`%xmm0` 中， y 放在`%rdi` 中， 而z 放在`%rsi` 中

本质还是指针



- 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 [经过预处理, 编译, 汇编]
 - 每个.o文件代码和数据地址都从0开始 [不能直接运行, 需和其他.o文件合成一个完整的可执行文件]
- 可执行目标文件 (默认为a.out) [由多个.o文件合并生成]
 - 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) [不需编译时打包进可执行文件]
 - 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)* [动态链接库]



可重定位目标文件格式(.O)

计算机系统基础I

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 静态局部变量

.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 int i (静态)

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

链接器 Linker 把各个 .o 文件合成可执行文件, 做以下两

① 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.

引用 —————> 确定该符号

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

② Step 2. 重定位

.o 文件的 text 节 和 .z, .data 节 分别合并

- 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

回头将写着“相对零地址”的符号引用处全部强行修改为这个
刚算出来的绝对虚拟地址



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：

1 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

2 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

3 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;  
}
```

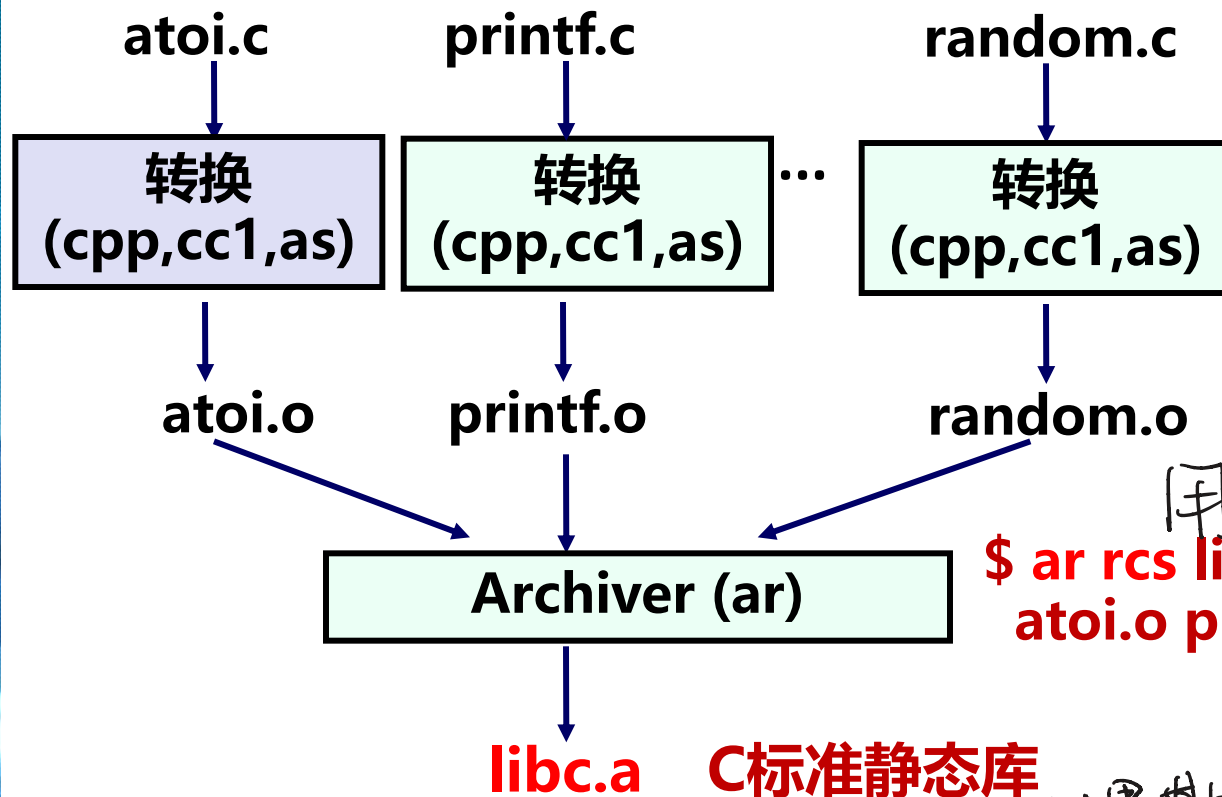
swap.c

```
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？



每个常用的.o文件打包成一个单独的文件，.a结尾



用归档工具ar来打包

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

如果新源文件改了

- 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)



链接顺序问题

静态链接器处理命令行参数，
从左到右扫描文件，维护三个集合

计算机系统基础I

互相独立

- 假设调用关系如下：

① 已解析符号 ② 未解析的符号引用 ③ 包含的模块

func.o ^{引用} libx.a 和 liby.a 中的函数

基本原则：如果一个.o文件引用某个静态库里的符号，那么这个静态库必须放在引用后面。

libx.a ^{又引用} libz.a 中的函数 func.o 必须最后列，libx.a 必须在 libz.a 前面

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：

相互循环依赖

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

计算机系统基础I

把符号解析和重定位都做完

• 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)
(所有地址强行修改为虚拟空间中绝对地址 (不需重定位信息))





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);
}
```




x86-64 反汇编代码

内存地址



机器码
↓

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
```

参数表:
第一个 8字节
%rdi
%rsi
%rdx
%rcx
%r8
%r9

计算机系统基础I

寄存器参数表 (顺序)
输入参数 n: %edi / %rdi
(首个 4字节) (4字节)

返回值 sum: %eax / %rax
(ret前一行绑)

endbr64

```
mov $0x1,%edx %edx = power = 1
mov $0x1,%ecx %ecx = sum = 1
mov $0x0,%eax %eax = i = 0
jmp 401192 <f1+0x10> 条件循环判断
add %edx,%edx power *= %edx = %edx + %edx
add %edx,%ecx sum += power
add $0x1,%eax i++
cmp %edi,%eax 比较 %edi (参数n) 和 %eax (i)
jl 40118b <f1+0x15> 如果 i < n 跳回循环体
mov %ecx,%eax sum 复制到 %eax 作返回值
ret
```

循环

条件判断

C语言指针, long - 8字节 (开头)
int - 4字节 (开头)



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

1. 改变rsp为main开始
2. 24字节的栈帧空间
3. 栈道以保护, 把个随
机数读入rax

4. 把eax和0, 用于后面输入

5. 把栈上移4, 原移

6. 把格式串"0"内地传

7. 调用scanf

8. 调用f1

9. 调用printf

10. 调用scanf

11. 调用f1

12. 调用printf

13. 调用scanf

14. 调用f1

15. 调用printf

16. 调用scanf

17. 调用f1

18. 调用printf

19. 调用scanf

★ 改变%rsp两种方法

① 显式改变: △, 用于栈向下生长, 减去0x18 = 在栈上预留24字节
用于存放临时变量n, 及安全保护数

② 隐式改变: 0, 所有call指令, CPU会自动把当前返回地址压入栈顶,

把sum = %d\n
地址送入rsi



程序部分调试信息如下：

(gdb) i r rip

rip 0x401196

(gdb) i r rsp

rsp 0x7fffffffde18

调用scanf输入变量指定0x4(%rsp)

0x4高4字节对应04

04000000000000000000000000000000

(gdb) x/4xg \$rsp

0x7fffffffde18: 0x00000000000004011cb

0x00000000040000000000000000000000

0x7fffffffde28: 0xfc957cf2acabe600

0x00000000000000000000000000000000

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

第一个输入参数，运行期间被

循环变量i，i=0,1,2,3



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

比较 %eax(i) 与 %edi(n)
j: 若 i < n (i <= n) 则跳回

401194: 7c f5

jl 40118b <f1+0x15>

循环体起始地址
否则退出, 向下继续执行

for 循环的边界条件判断与跳转控制

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？

都有

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？

☆ 因为这两个sum都为局部变量

都被分配在main自身独立堆栈帧空间内

f1() sum 在运行期间被编译器直接映射到 %ecx 中 临时变量

○ 由于它们分别应用完全隔开的内存区域与硬件寄存器资源，具有不同作用域与生命周期，访问时绝不会冲突