# 信息的表示与处理

## 信息储存

### 大小端序

x86 x64采取小端序，低字节存放低位。

常见的ARM采取大端序，低字节存放高位。

### 整数表示

#### 无符号

$$x=\sum_{i=0}^{w-1}(x_i2^i)$$

#### 有符号(补码)

$$x=-x_{w-1}2^{w-1}+\sum_{i=0}^{w-2}(x_i2^i)$$

#### 有无符号转换

```
int x;
```

$$(unsigned)x=\begin{cases}x & x\geq 0\\ x+2^w & x<0\end{cases}$$

### IEEE浮点数

float=1Sign+9Exp+23Frac

double=1Sign+11Exp+52Frac

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

s为符号，M称为尾数，E称为阶码。

#### 1.规格化的值

当Exp即非全1，也非全0时，浮点数表示一个规格化的值。

此时，

$E=Exp-Bias$，其中$Bias=\begin{cases}127 & \text{单精度}\\ 1023 & \text{双精度}\end{cases}$

Frac被解释为小数$f,0\leq f<1$，那么$M=1+f$

#### 2.非规格化的值

当Exp为全0时，浮点数表示非规格化的值，此时

$E = 1 - Bias$

$M = f$

### 3.特殊值

当Exp为全1时，浮点数表示一些特殊值。

如果Frac全为0，表示无穷，即

$$float = \begin{cases} +\infty & Sign = 0 \\ -\infty & Sign = 1 \end{cases}$$

其它时候，float表示的值被称为NaN(Not a Number)

# 信息处理

## 移位运算

### 左移

总在右侧补0。

### 算术右移

根据需求补0或补1，实现

$$x >> k = x/2^k$$

的期待算术功能。

### 逻辑右移

总在左侧补0，实现逻辑功能。

## 位扩展

### 零扩展

对于无符号数，我们只需要简单地在左侧补零即可。

### 符号扩展

根据符号左侧补0或补1，这样可以保证有符号数的值不变。

## 截断数字

### 无符号数截断

$$x' = x \mod 2^k$$

无符号数直接截断。

### 有符号数截断

有符号数先转化为无符号数，截断后再转换回来。

## 整数加法

### 无符号加法

$$x +_w^u y = \begin{cases} x+y & x+y < 2^w & \text{正常} \\ x+y-2^w & 2^w \le x+y < 2^{w+1} & \text{溢出} \end{cases}$$

### 无符号加法逆元

$$-_w^u x = \begin{cases} x & x = 0 \\ 2^w - x & x > 0 \end{cases}$$

### 补码加法

$$x +_w^t y = \begin{cases} x+y-2^w & 2^{w-1} \le x+y & \text{正溢出} \\ x+y & -2^{w-1} \le x+y < 2^{w-1} & \text{正常} \\ x+y+2^w & 2^w \le x+y < -2^{w-1} & \text{负溢出} \end{cases}$$

### 补码加法逆元

$$-_w^u x = \begin{cases} x & x = TMin_w \\ -x & x > TMin_w \end{cases}$$

## 整数乘法

### 无符号乘法

$$x *_w^u y = (x \cdot y) \mod 2^w$$

### 补码乘法

$$x *_w^t y = U2T_w((x \cdot y) \mod 2^w)$$

补码乘法和无符号乘法在位级是等价的。

## 浮点数的舍入

IEEE浮点数采取向偶数舍入的方式，自动向最接近的值舍入，即

待社区的部分低于新的最低位的一半时舍去，大于时则进位；

如果恰好等于最低位的一半，我们应该让新的最低位为0(前最低为1则进位，否则舍去)

# 程序的机器级表示

## 数据格式

| 类型 | 汇编代码后缀 | 大小(Bytes) |
| --- | --- | --- |
| Byte | b | 1 |
| Word | w | 2 |
| Double Words | l | 4 |
| Quad Words | q | 8 |
| Single | s | 4 |
| Double | l | 8 |

# 访问信息

## 寄存器

| 64位名称 | 32位名称 | 16位名称 | 8位名称 | 特殊作用/被谁保存 |
|---|---|---|---|---|
| rax | eax | ax | al | 返回值 |
| rbx | ebx | bx | bl | 被调用者保存 |
| rcx | ecx | cx | cl | 参数4 |
| rdx | edx | dx | dl | 参数3 |
| rsi | esi | si | sil | 参数2 |
| rdi | edi | di | dil | 参数1 |
| rbp | ebp | bp | bpl | 被调用者保存 |
| rsp | esp | sp | spl | 栈指针 |
| r8 | r8d | r8w | r8b | 参数5 |
| r9 | r9d | r9w | r9b | 参数6 |
| r10 | r10d | r10w | r10b | 调用者保存 |
| r11 | r11d | r11w | r11b | 调用者保存 |
| r12 | r12d | r12w | r12b | 被调用者保存 |
| r13 | r13d | r13w | r13b | 被调用者保存 |
| r14 | r14d | r14w | r14b | 被调用者保存 |
| r15 | r15d | r15w | r15b | 被调用者保存 |

## 操作数操作符(AT&T)

| 格式 | 操作数值 | 名称 |
|---|---|---|
| $Imm | Imm | 立即数 |
| r | R[r] | 寄存器 |
| Imm | M[Imm] | 直接寻址 |
| (r) | M[R[r]] | 间接寻址 |
| Imm(r) | M[R[r]+Imm] | 间接寻址 |
| $(r_a, r_b)$ | M[R$[r_a]$+R$[r_b]$] | 变址寻址 |
| $Imm(r_a, r_b)$ | M[R$[r_a]$+R$[r_b]$+Imm] | 变址寻址 |
| $(, r_a, s)$ | M$[s * R[r_a]]$ | 比例变址寻址 |
| $Imm(, r_a, s)$ | M$[s * R[r_a] + Imm]$ | 比例变址寻址 |

| 格式 | 操作数值 | 名称 |
| --- | --- | --- |
| $(r_a, r_b, s)$ | M[$R[r_a]+s*R[r_b]$] | 比例变址寻址 |
| $Imm(r_a, r_b, s)$ | M[$R[r_a]+s*R[r_b]+Imm$] | 比例变址寻址 |

# 控制

## 条件码

### 条件码寄存器

CPU还维持着一组单个位的条件码构成的寄存器。

| 条件码 | 名称 | 意义 | 执行t=a+b时的C表述 |
| --- | --- | --- | --- |
| CF | 进位标志 | 最高位进位 | (unsigned)t<(unsigned)a |
| ZF | 零标志 | 结果为0 | t==0 |
| SF | 符号标志 | 结果为负数 | t<0 |
| OF | 溢出标志 | 结果发生补码溢出 | (a<0==b<0)&&(t<0!=a<0) |

### 条件指令后缀

| 后缀 | 同义后缀 | 对应条件码 | 对于表达式a<b的条件 |
| --- | --- | --- | --- |
| e | z | ZF | 相等 |
| ne | nz | ~ZF | 不等 |
| s | | SF | 负数 |
| ns | | ZF | 非负数 |
| g | nle | ~(SF^OF)& ~ZF | 有符号> |
| ge | nl | ~(SF^OF) | 有符号>= |
| l | nge | SF^OF | 有符号< |
| le | ng | (SF^OF)\|ZF | 有符号<= |
| a | nbe | ~CF& ~ZF | 无符号> |
| ae | nb | ~CF | 无符号>= |
| b | nae | CF | 无符号< |
| be | na | CF\|ZF | 无符号<= |

# 抽象结构的实现

## 条件分支结构

#### if和三目运算符

使用条件传送(cmov__)和条件跳转(j__)可实现三目运算符(A?B:C)和条件分支(if...else if...else...)

### switch

当开关情况较多，且分布密集时，编译器将使用跳转表来翻译switch。

跳转表是一个由跳转地址构成的数组，那么switch可以根据被选择的值，确定到这个数组的某一位，来完成跳转。

对于default等情况，只需要单独判断即可。

## 循环结构

### do-while

对于

```
do{
    body-statement
}while(test-expr)
```

总能转化为

```
loop:
    body-statement
if (test-expr) goto loop;
```

进而转化为汇编

```
.Loop
    body-statement-asm
    calculate-test-expr-asm
    j__ .Loop
```

### while

对于

```
while(test-expr){
    body-statement
}
```

总能转化为

```
    goto test;
loop:
    body-statement
test:
    if (test-expr) goto loop;
```

进而转化为汇编

```
    jmp .Test
.Loop
    body-statement-asm
.Test
    calculate-test-expr-asm
    j__ .Loop
```

这种翻译方式被称为jump-to-middle。

还有一种翻译方式被称为guard-do方式，它将有更高的执行效率。

源代码总能转化为

```
    if (!(test-expr)) goto done;
loop:
    body-statement
    if (test-expr) goto loop;
done:
```

进而转化为汇编

```
    calculate-test-expr-asm
    j__ .Done
.Loop
    body-statement-asm
    calculate-test-expr-asm
    j__ .Loop
.Done
```

**for**

对于

```
for(init-expr;test-expr;update-expr){
    body-statement
}
```

总能转化为

```
init-expr
while(test-expr){
    body-statement
    update-expr
}
```

也就可以使用while的两种翻译方式。

## 栈帧结构

| **上一个过程的栈帧** | ***输入参数*(7+)** | |
|---|---|---|
| .. | ..... | |
| .. | *返回地址* | |
| **当前栈帧** | *上一个过程的帧指针* | ←rbp |
| .. | ***本过程的变量*** | |
| .. | ..... | |
| .. | ***要传递的参数***(7+) | |
| .. | ..... | |
| .. | *返回地址* | ←rsp |

# 汇编指令

## 数据传送指令

```
mov S,D; 把S的值复制到D
;movq, movl, movw, movb，除了movl外，只更新指定的字节，movl还会将高4字节置0
movabsq S,D
;对于movq，其源只能为32位补码数字，而movabsq可以使用64位立即数，但是只能以寄存器作为目标。
movz S,D;把S零扩展后的值复制到D
;movzbw,movzbl,movzbq,movzwl,movzwq,movl相当于movzlq
movs S,D;把S符号扩展后的值复制到D
;movsvw,movsbl,movsbq,movswl,movswq,movslq
cltq ;把eax符号扩展为rax
cmov__ S,D;'__'表示条件后缀，条件传送S到D
```

## 压入/弹出栈指令

x86-64的栈向下增长，栈顶数据位于低地址。

```
pushq S;栈指针-8，然后S入栈
popq D;弹出栈顶到D，然后栈指针+8
```

## 算术/逻辑指令

### 取地址指令

```
lea S,D;取S的地址到D
```

对于$Imm(r_a,r_b,s)$这样的S，实际上是计算算术运算$R[r_a]+Imm+s*R[r_b]$并赋值到D，编译器经常使用这样的奇怪用法。lea的目的操作数必须是寄存器。

### 单操作数指令

```
inc D;D++
dec D;D--
neg D;D=-D
not D;D=~D
```

### 算术指令

```
add S,D;D+=S
sub S,D;D-=S
IMUL S,D;D*=S
```

### 逻辑指令

```
xor S,D;D^=S
and S,D;D&=S
or S,D;D|=S
SAL k,D;D<<=k
SHL k,D;D<<=k
SAR k,D;D=((signed)D)>>k
SHR k,D;D=((unsigned)D)>>k
```

### 特殊算术指令

以下指令对128位数提供有限支持。

```
imulq S;有符号乘法
mulq S;无符号乘法
clto ;符号扩展为八字
idivq S;有符号除法
divq S;无符号除法
```

imulq/mulq将S和rax的两个64位整数相乘，结果储存到rdx:rax组成的128位寄存器中。

clto则将rax中的值符号扩展为rdx:rax的128位值

idivq/divq将rdx:rax的128位值除以S，然后商保存到rax，余数保存到rdx

## 条件码访问指令

```
set D
cmp B,A
test A,B
```

set+<条件后缀>将对应的条件值赋值给D。

cmp指令根据对A和B的比较情况设置条件码。

test指令根据A&B的情况设置条件码

## 跳转指令

```
jmp D
jmp *D
j__ D
j__ *D
```

跳转指令分为直接跳转和间接跳转，直接跳转是跳转到某个程序指定的位置，间接跳转则是跳转到D指向的位置。

jmp指令为无条件跳转，而j+<条件后缀>的指令则为条件跳转。

利用条件跳转可以实现程序的条件分支(if...else if...else...)

### 控制转移指令

```
call D
call *D
ret
```

call指令调用过程D，或者D指向的过程(*D)。执行该指令，会将返回地址入栈，然后改变程序计数器的值。

ret指令表示从当前过程返回到调用者，即从栈中弹出地址，然后设置PC为该地址。

# 处理器体系结构

## Y86-64

### 架构状态

#### 寄存器

除了r15以外的所有x86-64寄存器。

r15用于指令不需要寄存器时的占位。

#### 程序状态

条件码CC: ZF,SF,OF

程序状态: Stat

### 指令集与编码

| 指令 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| halt | 0 | 0 | | | | | | | | |
| nop | 1 | 0 | | | | | | | | |
| cmovxx rA,rB | 2 | fn | rA | rB | | | | | | |
| irmovq V,rB | 3 | 0 | F | rB | V | .. | .. | .. | .. | .. |
| rmmovq rA,D(rB) | 4 | 0 | rA | rB | D | .. | .. | .. | .. | .. |
| mrmovq D(rB),rA | 5 | 0 | rA | rB | D | .. | .. | .. | .. | .. |

| 指令 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| OPq rA,rB | 6 | fn | rA | rB | | | | | | |
| jXX Dest | 7 | fn | Dest | .. | .. | .. | .. | .. | .. | .. |
| call Dest | 8 | 0 | Dest | .. | .. | .. | .. | .. | .. | .. |
| ret | 9 | 0 | | | | | | | | |
| pushq rA | A | 0 | rA | F | | | | | | |
| popq rA | B | 0 | rA | F | | | | | | |

## 特点

1-10字节的信息，从内存读取

根据第一字节可判断指令长度

比x86-64的指令类型少

比x86-64的编码简单

每次存取更改程序状态的一些部分

## fn编码

**cmovXX**

| 指令 | fn |
|---|---|
| rrmovq | 0 |
| cmovle | 1 |
| cmovl | 2 |
| cmove | 3 |
| cmovne | 4 |
| cmovge | 5 |
| cmovg | 6 |

**OPq**

| 指令 | |
|---|---|
| addq | 0 |
| subq | 1 |
| andq | 2 |
| xorq | 3 |

**jXX**

| 指令 | fn |
| --- | --- |
| jmp | 0 |
| jle | 1 |
| jl | 2 |
| je | 3 |
| jne | 4 |
| jge | 5 |
| jg | 6 |

# CISC和RISC

## CISC

基于栈的指令集，程序计数器。

明确的出、入栈指令

算术指令可以访存，地址计算复杂。

条件码是算术和逻辑运算的副作用

## RISC

更多的寄存器，没有条件码。

更少更简单的指令

MISP寄存器

# HCL

## 字相等

```
bool Eq=(A==B)
```

## 位多路复用器

```
bool out=(s&&a)||(!s&&b)
```

## 字多路复用器

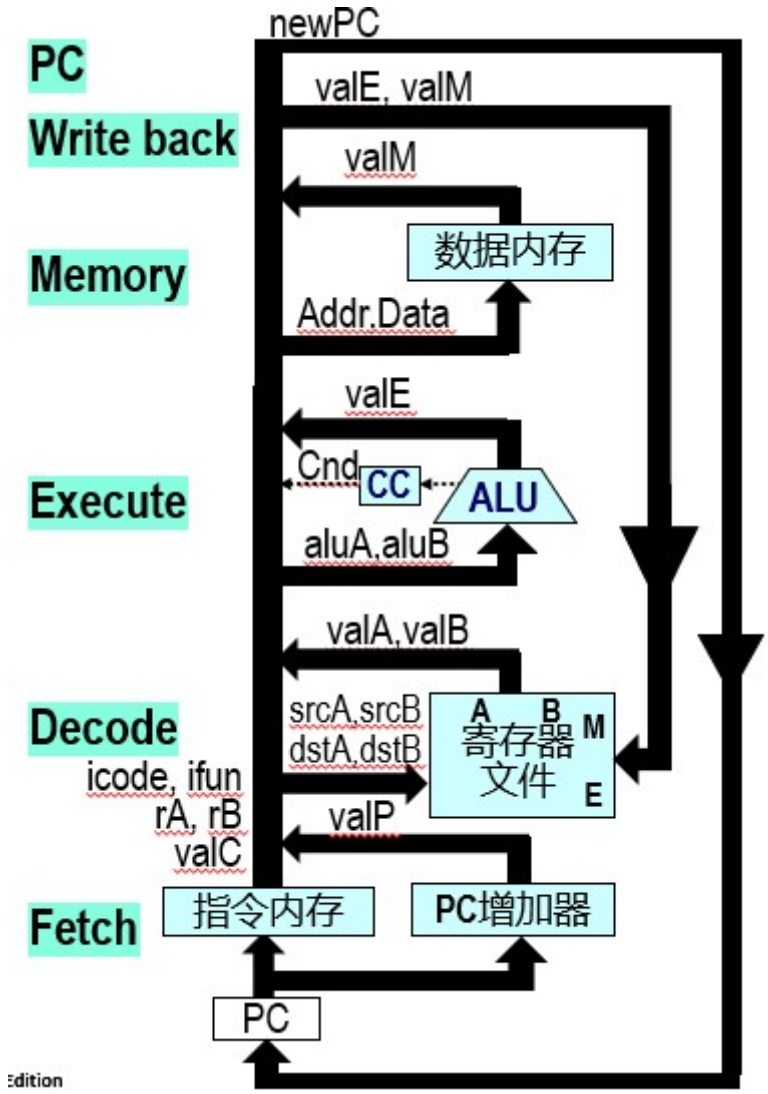
```
int Out=[
    s:A;
    1:B;
]
```

情况表达式(case语句)

一系列二元组"布尔表达式:整数表达式", 第一个求值为1 的情况会被选中

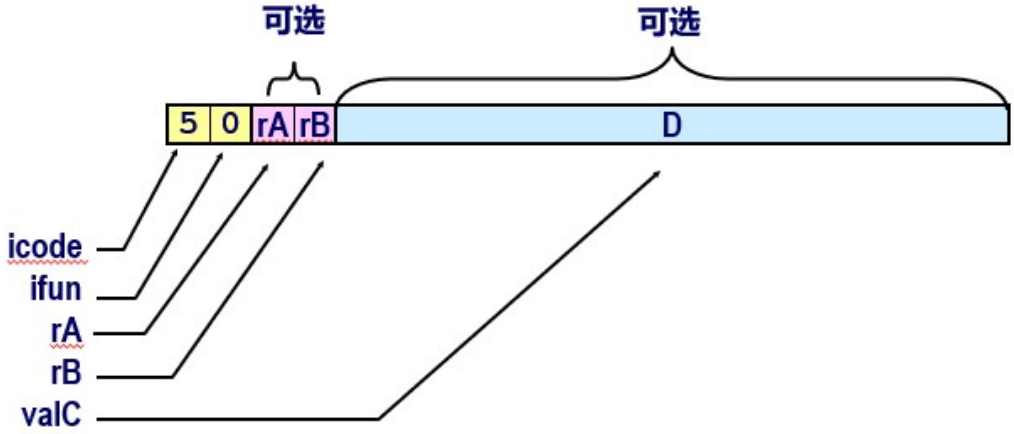
# 顺序执行的处理器

## SEQ硬件结构




| 阶段 | 嘤语 | 作用 |
| --- | --- | --- |
| 取指 | Fetch | 从指令储存器读取指令 |
| 译码 | Decode | 读程序寄存器 |
| 执行 | Execute | 计算数值/地址 |
| 访存 | Memory | 读/写内存数据 |
| 写回 | Write Back | 写程序寄存器 |
| 更新PC | PC | 更新程序计数器 |

## 指令编码的分析




指令字节 icode:ifun

可选的寄存器字节 rA:rB

可选的常数字 valC

# 指令的执行

## OPq

| | **OPq rA,rB** |
|---|---|
| Fetch | $icode : ifun \leftarrow M_1[PC]$ |
| .. | $rA : rB \leftarrow M_1[PC+1]$ |
| .. | $valP \leftarrow PC+2$ |
| Decode | $valA \leftarrow R[rA]$ |
| .. | $valB \leftarrow R[rB]$ |
| Execute | $valE \leftarrow valA\ OP\ valB$ |
| .. | $Set\ CC$ |
| Memory | |
| WriteBack | $R[rB] \leftarrow valE$ |
| .. | |
| PC | $PC \leftarrow valP$ |

## cmov/rrmovq

| | **cmovXX/rrmovq rA,rB** |
|---|---|
| Fetch | $icode : ifun \leftarrow M_1[PC]$ |
| .. | $rA : rB \leftarrow M_1[PC + 1]$ |
| .. | $valP \leftarrow PC + 2$ |
| Decode | $valA \leftarrow R[rA]$ |
| .. | $valB \leftarrow 0$ |
| Execute | $valE \leftarrow valA + valB$ |
| .. | $if(!Cond(CC, ifun))\ rB \leftarrow 0xF$ |
| Memory | |
| WriteBack | $R[rB] \leftarrow valE$ |
| .. | |
| PC | $PC \leftarrow valP$ |

用ALU传递数据。
如果传送条件不满足，则将端口设为0xF来阻止数据传输。

## irmovq

| | **irmovq V,rB** |
|---|---|
| Fetch | $icode : ifun \leftarrow M_1[PC]$ |
| .. | $rA : rB \leftarrow M_1[PC + 1]$ |
| .. | $valC \leftarrow M_8[PC + 2]$ |
| .. | $valP \leftarrow PC + 10$ |
| Decode | |
| .. | |
| Execute | $valE \leftarrow 0 + valC$ |
| .. | |
| Memory | |
| WriteBack | $R[rB] \leftarrow valE$ |
| .. | |
| PC | $PC \leftarrow valP$ |

## rmmovq, mrmovq

| | **rmmovq rA,D(rB)** |
|---|---|
| Fetch | $icode : ifun \leftarrow M_1[PC]$ |
| .. | $rA : rB \leftarrow M_1[PC + 1]$ |
| .. | $valC \leftarrow M_8[PC + 2]$ |
| .. | $valP \leftarrow PC + 10$ |
| Decode | $valA \leftarrow R[rA]$ |
| .. | $valB \leftarrow R[rB]$ |
| Execute | $valE \leftarrow valB + valC$ |
| .. | |
| Memory | $M_8[valE] \leftarrow valA$ |
| WriteBack | |
| .. | |
| PC | $PC \leftarrow valP$ |

| | **mrmovq D(rB),rA** |
|---|---|
| Fetch | $icode:ifun \leftarrow M_1[PC]$ |
| .. | $rA:rB \leftarrow M_1[PC+1]$ |
| .. | $valC \leftarrow M_8[PC+2]$ |
| .. | $valP \leftarrow PC+10$ |
| Decode | |
| .. | $valB \leftarrow R[rB]$ |
| Execute | $valE \leftarrow valB+valC$ |
| .. | |
| Memory | $valM \leftarrow M_8[valE]$ |
| WriteBack | |
| .. | $R[rA] \leftarrow valM$ |
| PC | $PC \leftarrow valP$ |

利用ALU计算内存的有效地址

## popq/pushq

| | **pushq rA** |
|---|---|
| Fetch | $icode:ifun \leftarrow M_1[PC]$ |
| .. | $rA:rB \leftarrow M_1[PC+1]$ |
| .. | $valP \leftarrow PC+2$ |
| Decode | $valA \leftarrow R[rA]$ |
| .. | $valB \leftarrow RSP$ |
| Execute | $valE \leftarrow valB-8$ |
| Memory | $M_8[valE] \leftarrow valA$ |
| WriteBack | $RSP \leftarrow valE$ |
| .. | |
| PC | $PC \leftarrow valP$ |

| | popq rA |
|---|---|
| Fetch | $icode : ifun \leftarrow M_1[PC]$ |
| .. | $rA : rB \leftarrow M_1[PC+1]$ |
| .. | $valP \leftarrow PC+2$ |
| Decode | $valA \leftarrow RSP$ |
| .. | $valB \leftarrow RSP$ |
| Execute | $valE \leftarrow valB+8$ |
| Memory | $valM \leftarrow M_8[valA]$ |
| WriteBack | $RSP \leftarrow valE$ |
| .. | $R[rA] \leftarrow valM$ |
| PC | $PC \leftarrow valP$ |

利用ALU增减栈指针，更新两个寄存器。

## jXX

| | jXX Dest |
|---|---|
| Fetch | $icode : ifun \leftarrow M_1[PC]$ |
| .. | $valC \leftarrow M_8[PC+1]$ |
| .. | $valP \leftarrow PC+9$ |
| Decode | |
| .. | |
| Execute | |
| .. | $Cnd \leftarrow Cond(CC, ifun)$ |
| Memory | |
| WriteBack | |
| .. | |
| PC | $PC \leftarrow Cnd?valC : valP$ |

计算两个地址(跳转或者不跳转的地址)

根据分支条件做出选择

## call

| | **call Dest** |
|---|---|
| Fetch | $icode : ifun \leftarrow M_1[PC]$ |
| .. | $valC \leftarrow M_1[PC+1]$ |
| .. | $valP \leftarrow PC+9$ |
| Decode | |
| .. | $valB \leftarrow RSP$ |
| Execute | $valE \leftarrow valB-8$ |
| .. | |
| Memory | $M_8[valE] \leftarrow valP$ |
| WriteBack | $RSP \leftarrow valE$ |
| .. | |
| PC | $PC \leftarrow valC$ |

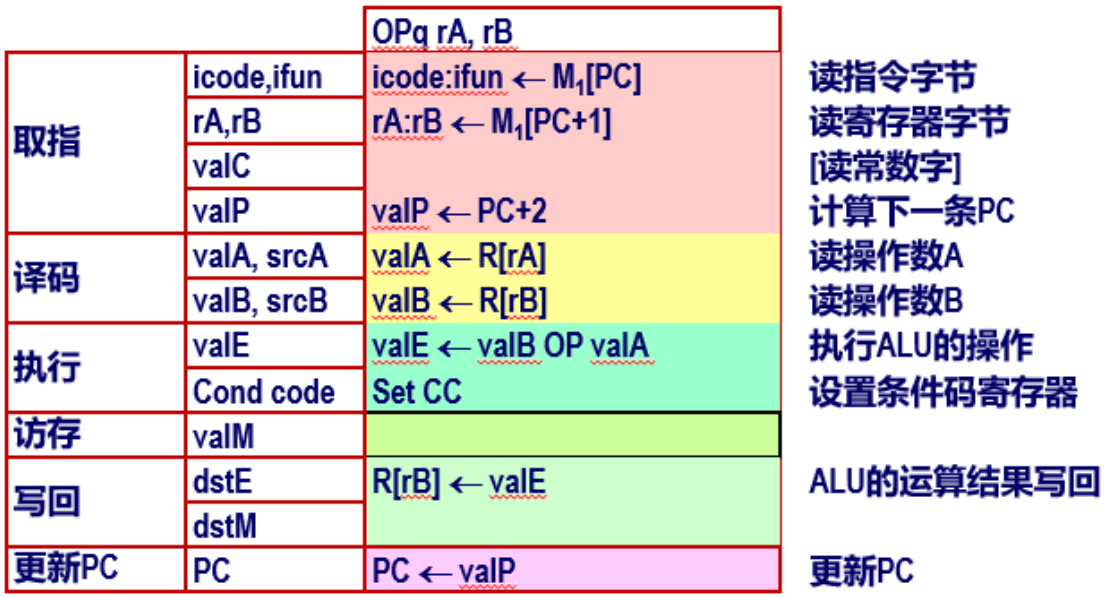
利用ALU减少栈指针, 储存增加后的PC。

### ret

| | **ret** |
|---|---|
| Fetch | $icode : ifun \leftarrow M_1[PC]$ |
| .. | $valP \leftarrow PC+1$ |
| Decode | $valA \leftarrow RSP$ |
| .. | $valB \leftarrow RSP$ |
| Execute | $valE \leftarrow valB+8$ |
| .. | |
| Memory | $valM \leftarrow M_8[valA]$ |
| WriteBack | $RSP \leftarrow valE$ |
| .. | |
| PC | $PC \leftarrow valM$ |

利用ALU增加栈指针，从内存读取新的PC值

## 通用操作序列

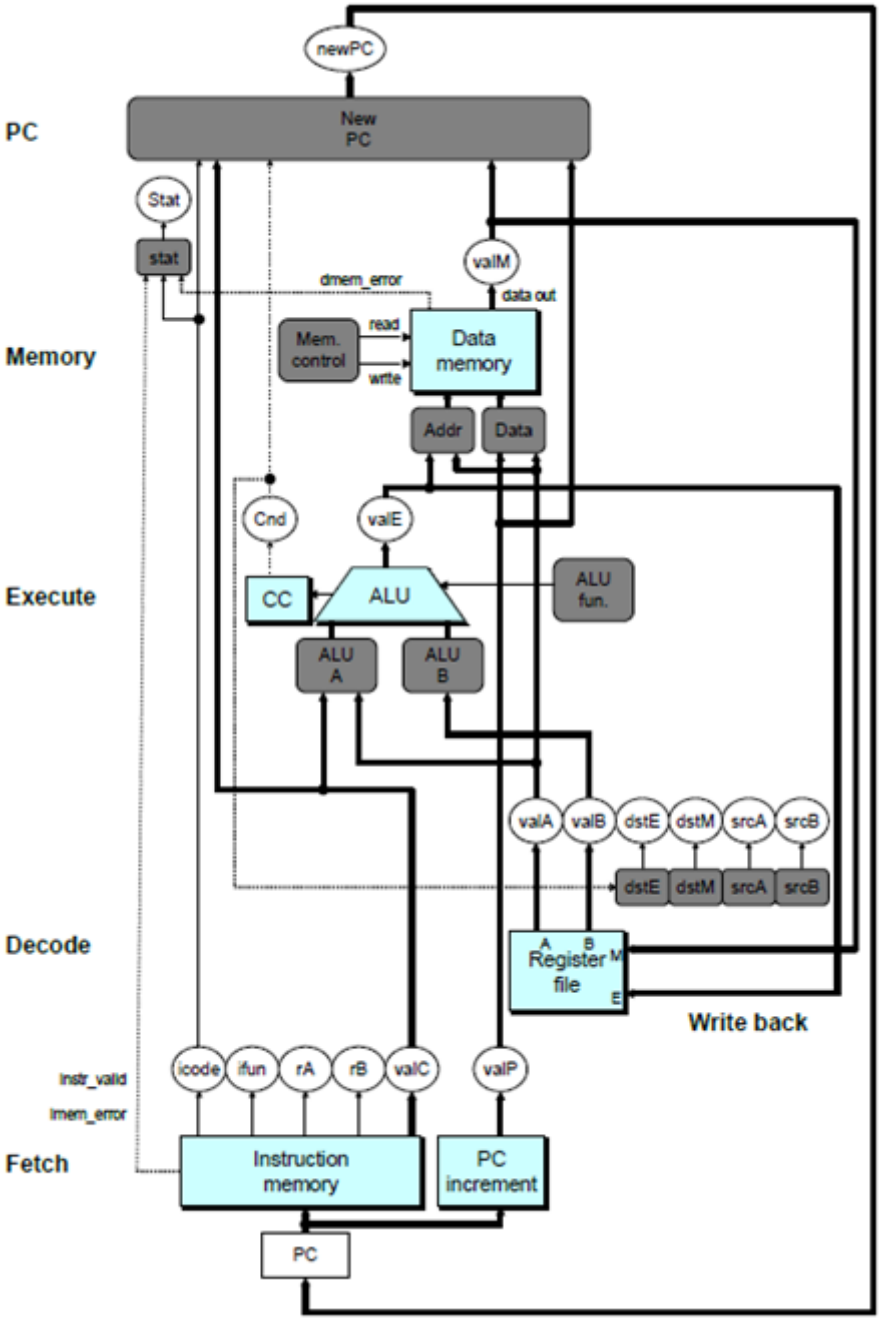



所有指令有相同的格式，但是每一步计算的内容有区别。

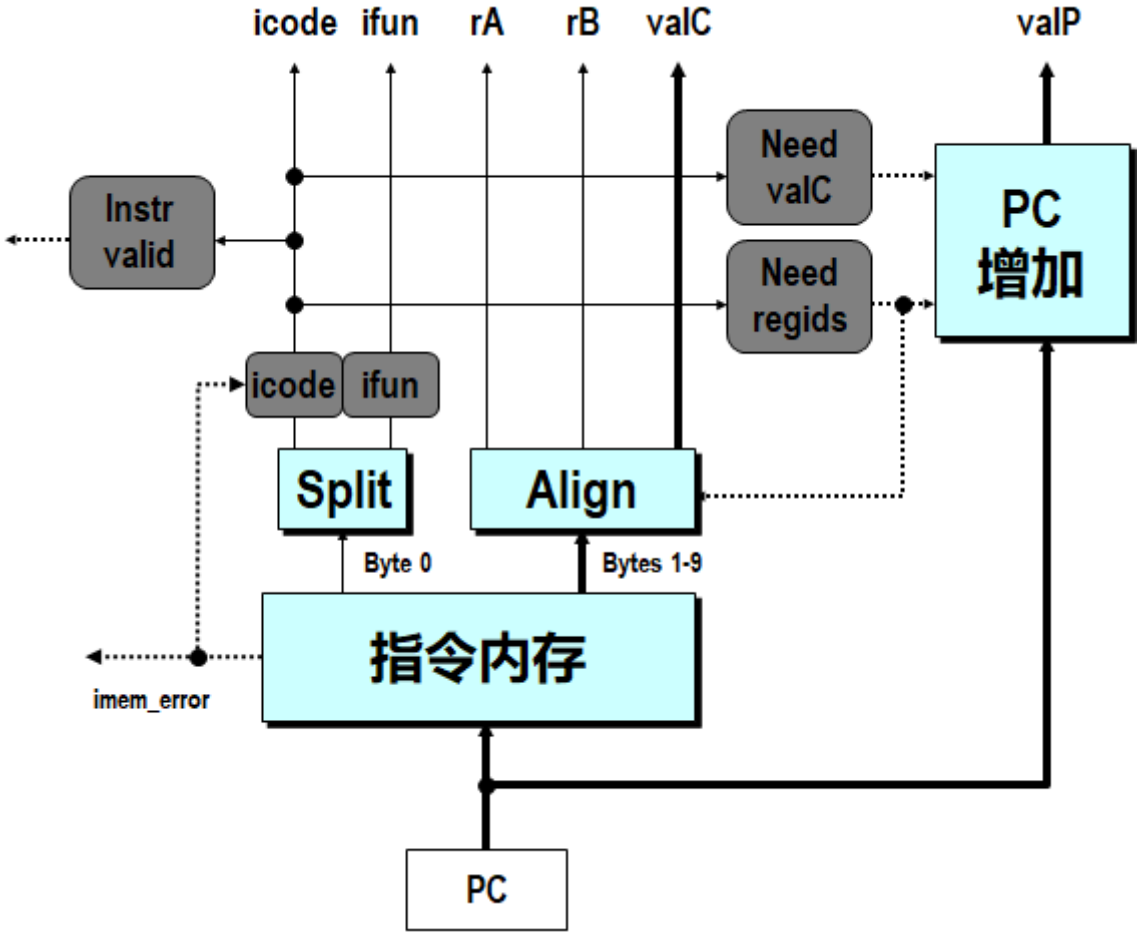
***操作产生的信号：***

| 阶段 | 信号名 | 意义 |
| --- | --- | --- |
| Fetch | icode | 指令码 |
| .. | ifu | 指令功能 |
| .. | rA | 指令中的寄存器A |
| .. | rB | 指令中的寄存器B |
| .. | valC | 指令中的常数 |
| .. | valP | 增加PC |
| Decode | srcA | 寄存器ID A |
| .. | srcB | 寄存器ID B |
| .. | dstE | 目的寄存器E |
| .. | dstM | 目的寄存器M |
| .. | valA | 寄存器值A |
| .. | valB | 寄存器值B |
| Execute | valE | ALU运算结果 |
| .. | Cnd | 分支或转移标识 |
| Memory | valM | 内存中的数值 |

## 细化的SEQ硬件结构


newPC
PC
New PC
Stat
stat
valM
dmem_error
data out
Memory
Mem. control
read
write
Data memory
Addr
Data
Cnd
valE
Execute
CC
ALU
ALU fun.
ALU A
ALU B
valA
valB
dstE
dstM
srcA
srcB
Decode
Register file
Write back
instr_valid
imem_error
icode
ifun
rA
rB
valC
valP
Fetch
Instruction memory
PC increment
PC

## Fetch




Split将第一字节拆分icode和ifun

Align将剩余的字节输入寄存器与常数

**控制逻辑**

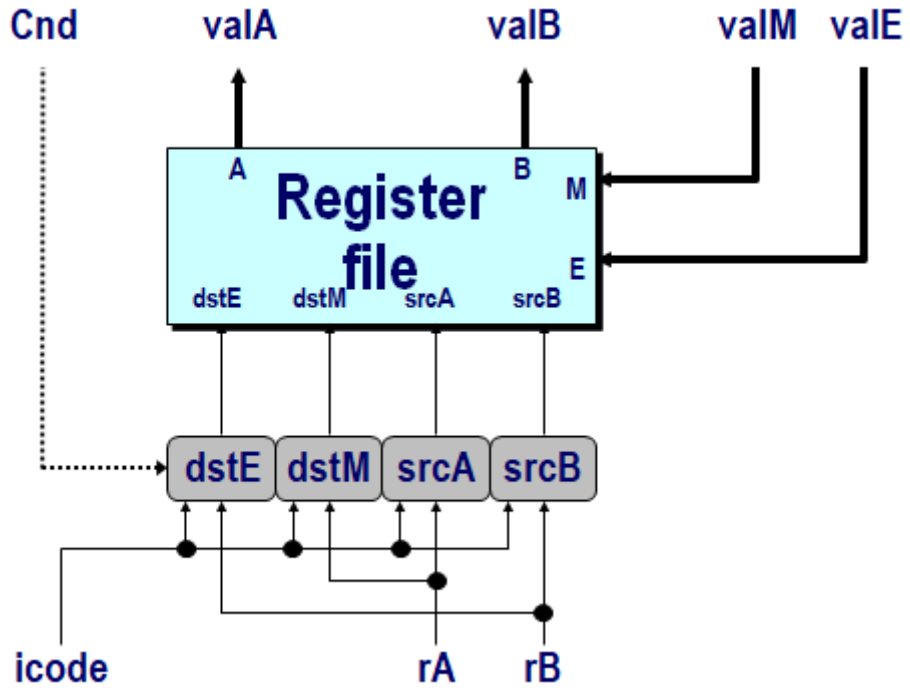
Instr. Valid: 指令是否有效? icode, ifun: 指令地址无效时生成no-op指令

Need regids: 指令是否有寄存器字节?

Need valC: 指令是否有常数字?

```
bool need_regids = icode in {
    irrmovq, iopq, ipushq, ipopq,
    iirmovq, irmmovq, imrmovq
};
bool instr_valid = icode in {
    inop, ihalt, irrmovq, iirmovq,
    irmmovq, imrmovq, iopq, ijxx,
    icall, iret, ipushq, ipopq
};
bool need_valC = icode in {
    iirmovq, irmmovq, imrmovq, ijxx, icall
};
```

## Decode/WriteBack




**信号**

Cnd:触发条件

**控制逻辑**

srcA, srcB: 读端口地址

dstE,dstM: 写端口地址

```
int srcA = [
    icode in {
        irrmovq, irmmovq, iopq, ipushq
    }:rA;
    icode in {
        ipopq, iret
    }:rRsp;
    1:rNone;
];
int srcB = [
    icode in {
        imrmovq, iopq, irmmovq,
    }:rB;
    icode in {
        ipopq, ipushq, icall, iret
    }:rRsp;
    1:rNone;
];
int dstE = [
    icode in {
        irrmovq
    } && Cnd:rB;
    icode in {
        iirmovq, iopq
    }:rB;
    icode in {
        ipushq, ipopq, icall, iret
```

```
    }:rRsp;
    1:rNone
];
int dstM = [
    icode in {
        popq, mrmovq
    }:rA;
    1:rNone;
];
```

## Execute

### 控制逻辑

Set CC: 是否更新条件码寄存器

ALU A: 数据A送ALU

ALU B: 数据B送ALU

ALU fun: ALU执行哪个功能

```
int aluA = [
    icode in {
        irrmovq, iopq
    }:valA;
    icode in {
        iirmovq,irmmovq,imrmovq
    }:valC;
    icode in {
        icall, ipushq
    }:-8;
    icode in {
        iret, ipopq
    }:8;

];
int aluB = [
    icode in {
        iopq, irmmovq, imrmovq, ipushq,
        ipopq, icall, iret
    }:valB;
    icode in {
        irrmovq, iirmovq
    }:0;

];
bool set_CC = icode in {iopq};
int alu_fun = [
    icode == iopq:ifun;
    1:ALUADD;
]
```

## Memory

### 控制逻辑

stat: 指令状态是什么?

Mem. read: 是否读数据字?

Mem. write: 是否写数据字?

Mem. addr.: 选择地址

Mem. data.: 选择数据

```
int mem_addr = [
    icode in {
        irmmovq, ipushq, icall, imrmovq
    }:valE;
    icode in {
        ipopq, iret
    }:valA;
];
int mem_data = [
    icode in {
        irmmovq, ipushq
    }:valA;
    icode in {
        icall
    }:valP;

];
bool mem_read = icode in {
    imrmovq, ipopq, iret
};
bool mem_write = icode in {
    irmmovq, ipushq, icall
};
int Stat = [
    imem_error || dmem_error : SADR;
    !instr_valid:SINS;
    icode==ihalt:SHLT;
    1:SAOK;
];
```

### PC

#### 控制逻辑

new_pc:下一个pc值

```
int new_pc = [
    icode==icall:valC;
    icode==ijxx&&Cnd:valC;
    icode==iret:valM;
    1:valP;
];
```

# 优化程序性能

函数调用的副作用，和内存的别名，使得编译器优化的能力是有极限的。

程序性能的表示采用每元素的周期数(CPE)表示。它表示每轮迭代所需的时钟周期数。

# 示例定义

## 数据定义和基本函数

来自vec.h

```
/* Create abstract data type for vector */
typedef struct {
    long len;
    data_t *data;
} vec_rec, *vec_ptr;

/* Create vector */
vec_ptr new_vec(long len);

/* Free storage used by vector */
void free_vec(vec_ptr v);

/*
 * Retrieve vector element and store in dest.
 * Return 0 (out of bounds) or 1 (successful)
 */
int get_vec_element(vec_ptr v, long index, data_t *dest);

/* Macro version */
#define GET_VEC_ELEMENT(v,index,dest) \
    !((index) < 0 || (index) >= (v)->len) && \
    *(dest) = (v)->data[(index)], 1;


data_t *get_vec_start(vec_ptr v);

/*
 * Set vector element.
 * Return 0 (out of bounds) or 1 (successful)
 */

int set_vec_element(vec_ptr v, long index, data_t val);

/* Get vector length */
long vec_length(vec_ptr v);

/* Set length of vector.  If > allocated length, will reallocate */
void set_vec_length(vec_ptr v, long newlen);
```

## 原始版本

```
void combine1(vec_ptr v, data_t *dest)
{
    *dest = IDENT;
    for (long i = 0; i < vec_length(v); i++)
    {
        data_t val;
        get_vec_element(v, i, &val);
        *dest = *dest OP val;
    }
}
```

# 消除循环带来的低效率

消除循环中可能出现重复的过程调用、不必要的内存引用。这些会造成低效率的累加。

```
void combine4(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;
    for (long i = 0; i < length; i++)
    {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

# 利用现代处理器结构带来优化界限

超标量，流水线，并行。

## 延迟界限

延迟是完成运算所需要的总时间。

发射时间表示连续两个相同类型运算之间需要的最小时钟周期数。

容量表示所有能够执行该运算的功能单元数。

| 运算 | 延迟 | 发射时间 | 容量 |
| --- | --- | --- | --- |
| Int+ | 1 | 1 | 4 |
| Int* | 3 | 1 | 1 |
| Int/ | 3-30 | 3-30 | 1 |
| float+ | 3 | 1 | 1 |
| float* | 5 | 1 | 2 |
| float/ | 3-15 | 3-15 | 1 |

## 吞吐量界限

功能元件产生结果的最大速率，是CPE的最小界限。

# 并行与循环展开

## k*1展开

```
/*
 * 2*1 循环展开
 * 2 表示每轮循环运算2次
 * 1 表示每次只有一个缓冲变量。
 */
void combine5(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length - 1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;
    for (i = 0; i < limit; i += 2)
    {
        acc = (acc OP data[i]) OP data[i + 1];
    }
    //剩余项
    for (; i < length; i++)
    {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

这种展开无法突破延迟界限。

## k*k展开

```
/*
 * 2*2 循环展开
 * 2 表示每轮循环运算2次
 * 2 表示每次有2个缓冲变量。
 */
void combine5(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length - 1;
    data_t *data = get_vec_start(v);
    data_t acc0 = IDENT;
    data_t acc1 = IDENT;
    for (i = 0; i < limit; i += 2)
    {
        acc0 = acc0 OP data[i];
        acc1 = acc1 OP data[i + 1];
    }
    //剩余项
    for (; i < length; i++)
    {
```

```
        acc0 = acc0 OP data[i];
    }
    *dest = acc0 OP acc1;
}
```

k取得够大时，程序性能能逼近吞吐量界限。但是当k过大，可能造成寄存器溢出，反而低效。

## k*1a展开

这种写法产生的汇编和k*k展开更接近，因此也可以逼近吞吐量界限。

```
/*
 * 2*1a 循环展开
 * 2 表示每轮循环运算2次
 * 1 表示每次只有一个缓冲变量。
 * a 表示元素结合方式的改变为右优先结合
 */
void combine7(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    long limit = length - 1;
    data_t *data = get_vec_start(v);
    data_t acc = IDENT;
    for (i = 0; i < limit; i += 2)
    {
        acc = acc OP (data[i] OP data[i + 1]);
    }
    //剩余项
    for (; i < length; i++)
    {
        acc = acc OP data[i];
    }
    *dest = acc;
}
```

# 优化的限制因素

## 寄存器溢出

当循环并行度超过了寄存器数量，并行反而会造成效率的下降。

## 分支预测和预测错误处罚

不要过分关心可预测的分支：

预测分支的错误不会对程序执行中关键路径的指令的取指和处理产生太大影响

书写适合用条件传送实现的代码。

# 储存器层次结构

# 程序的局部性

## 时间局部性

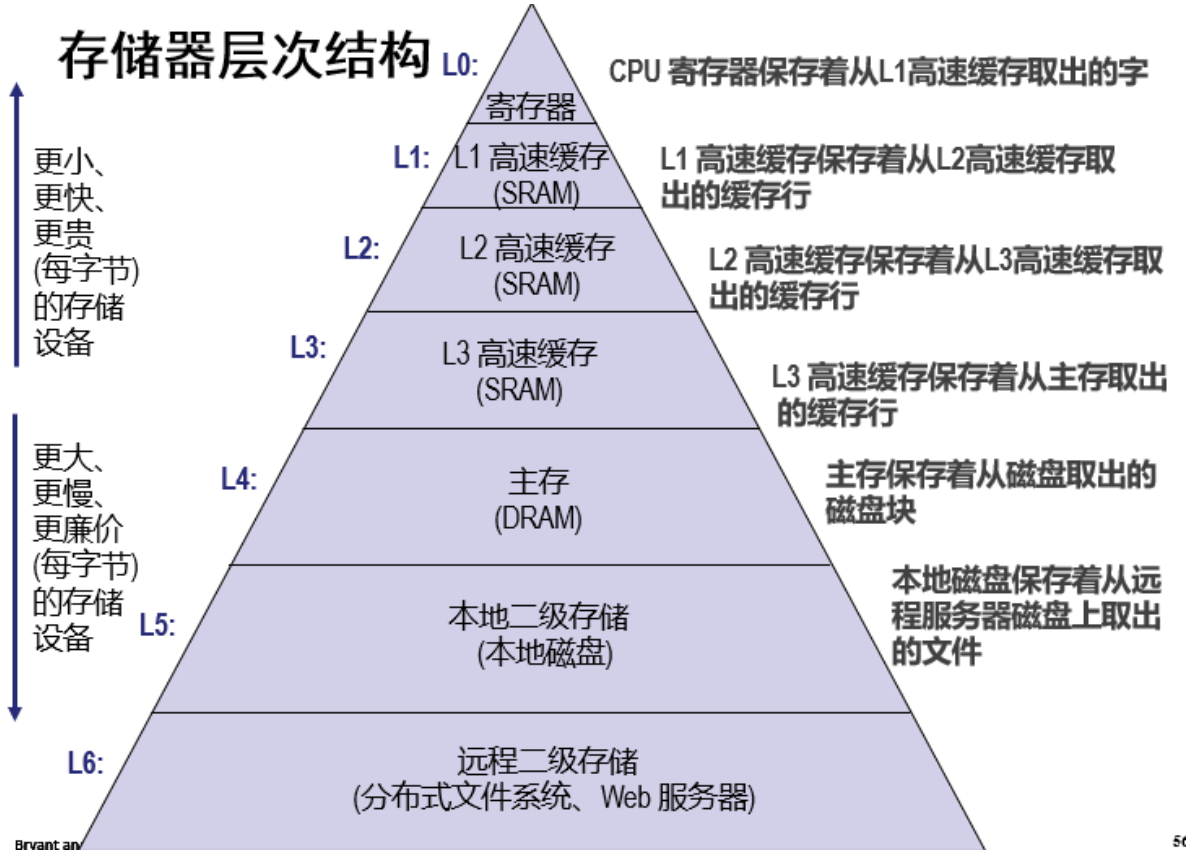
程序最近访问过的信息，很可能在近期还会再次访问。

## 空间局部性

地址接近的数据项，被使用的时间也倾向于接近

# 高速缓存

## 储存器层次结构




高速缓存将一种更小、速度更快的存储设备，作为更大、更慢存储设备的缓存区。

不同层之间，数据以块为单位进行复制传输。

## 缓存不命中

如果k层中，未找到请求的数据，则发生缓存不命中。

### 冷不命中/强制不命中

当缓存为空时，任何数据请求都不会命中。

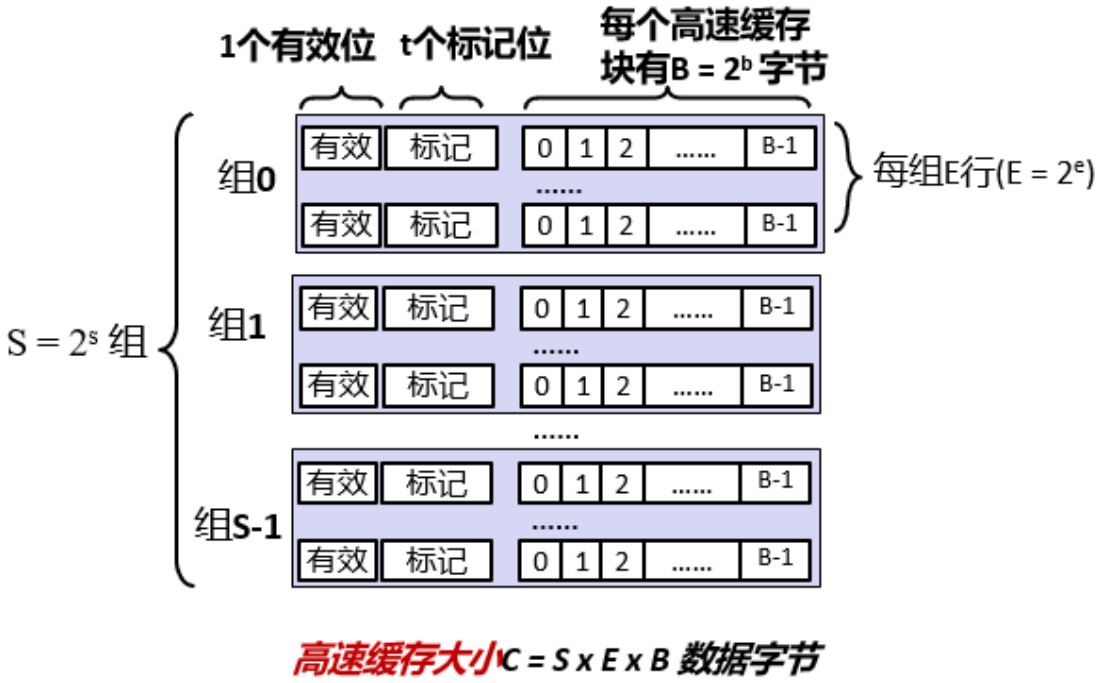
### 冲突不命中

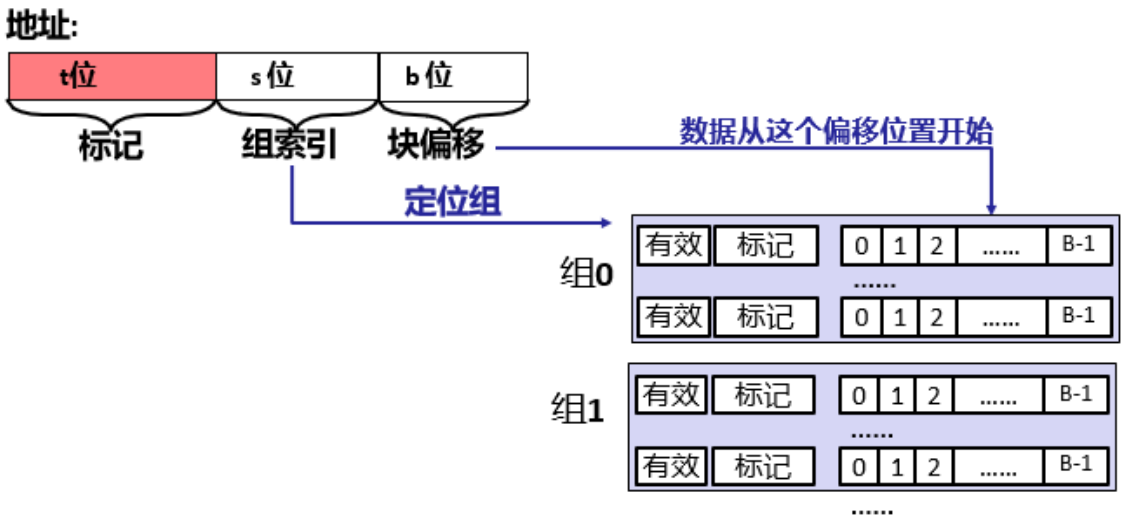
大部分缓存将k+1层的某个块限制在k层块的一个子集里。当缓存足够大，但是被引用的对象都映射到同一缓存块中，就发生冲突不命中。

### 容量不命中

当工作集大小超过缓存大小时，会发生容量不命中。

# 高速缓存的结构和读写




## 高速缓存读

1. 根据组索引定位组
2. 坚持组中有无匹配的标记
3. 如果存在匹配标记&&行有效，那么缓存命中
4. 定位从偏移开始的数据

**直接映射高速缓存**

每一组只有一行的高速缓存，速度很快。但是很容易冲突不命中。

**E路组相联高速缓存**

魅族有E行的高速缓存。

当缓存不命中发生时，在组中选择一行用于驱逐和替换。替换策略包含随机策略和LRU(最近最少使用)策略。

**全相联高速缓存**

只有一组的高速缓存，速度较慢。

### 高速缓存写

#### 写命中时

##### 直写

直接写入k+1层储存器

##### 写回

推迟写入，直到行要被替换(需要额外维护一个修改位)。

#### 写不命中时

##### 写分配

加载行到缓存，然后更新这个缓存行；常配合写回使用。

##### 非写分配

直接写入下一级存储；常和直写配合使用。

## 高速缓存的性能指标

### 不命中率

不命中的频率。

### 命中时间

获取一行所需的时间。

### 不命中处罚

不命中时所需要的额外的时间。

# 链接

# 静态链接

源程序被分开编译成可重定位的目标文件(*.o)，然后被链接成一个完整的可执行文件。

链接使得程序模块化，这样可以提供代码重用率（构建公共函数库）

同时，这样也可以提高效率，当只更改了一个源文件时，其它文件不再需要重新编译。这样也节省了空间。

## 符号解析

程序定义全局变量和函数，这些被称为符号。

汇编器在目标文件的符号表中记录符号，符号表的每个条目包含符号的名称、大小和位置；

在符号解析中，链接器将每个符号引用与一个确定的符号定义关联起来。

## 重定位

连接器将多个单独的代码节和数据节合并为单个节；把符号从相对位置重定位到最终的绝对内存位置，并以此更新这些符号的引用。

# 目标文件/模块

## 分类

可重定位目标文件(.o)

包含代码和数据，其形式能与其它可重定位目标文件相结合，以形成可执行的目标文件

可执行目标文件(.exe,无扩展名)

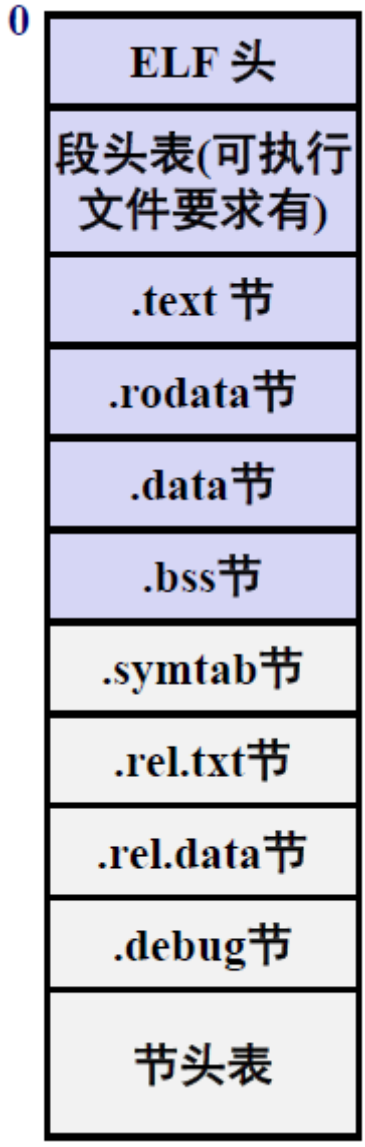
可以直接执行的文件

共享目标文件/动态链接库(.so,.dll)

可以在加载或运行时，动态地加载到内存中并链接。

## 可执行与链接格式 (ELF)

目标文件的标准二进制格式。适用于Linux和Unix




### ELF头

这部分包含16字节说明的字大小、字节序

还包含文件类型，机器类型，节头表位置等内容。

### 段头表/程序头表

可执行文件需要有的部分。说明页面大小，虚拟地址内存段，段大小

### .init

初始化代码

### .text

代码段

### .rodata

只读数据，如printf的格式串，跳转表。

### .data

可读写的数据。已初始化的全局或者静态变量。

### .bss

未初始化/初始化为0的全局和静态变量；仅有节头，节本身不占用磁盘空间

### .symtab

符号表，记录函数和全局/静态变量名，节名称和位置。

### .rel.text

可重定位代码；.text节的可重定位信息。

在可执行文件中需要修改的指令地址和需要修改的指令。

### .rel.data

可重定位数据，.data节的可重定位信息。表示在合并后的可执行文件中需要修改的指针数据的地址。

### .debug

调试符号表，为符号调试的信息。

### .line

### .strtab

### 节头表

每个节的偏移量和大小，描述可重定位的目标文件。

# 符号

## 符号的分类

### 全局符号

由模块m定义，可由其它模块引用的符号，主要包含非静态函数和非静态全局变量。

### 外部符号

由其他模块定义，被模块m引用的符号。

### 本地/局部符号

由模块m定义，仅有m引用的符号。主要包含静态的函数和全局变量以及静态变量。本地连接器符号不是程序的局部变量，非静态局部变量储存在栈上。

## 符号处理规则

### 符号的强弱

函数和初始化的全局变量是强符号；未初始化的全局变量是弱符号。

### 处理规则

1. 不允许有多个同名强符号，否则会产生链接错误
2. 若有一个强符号和多个弱符号同名，则选择强符号解析
3. 如果有多个弱符号，则任选其一。

### 避免错误的方法

- 如果可以的话，使用static修饰
- 定义时初始化他
- 使用extern声明引用的外部全局符号

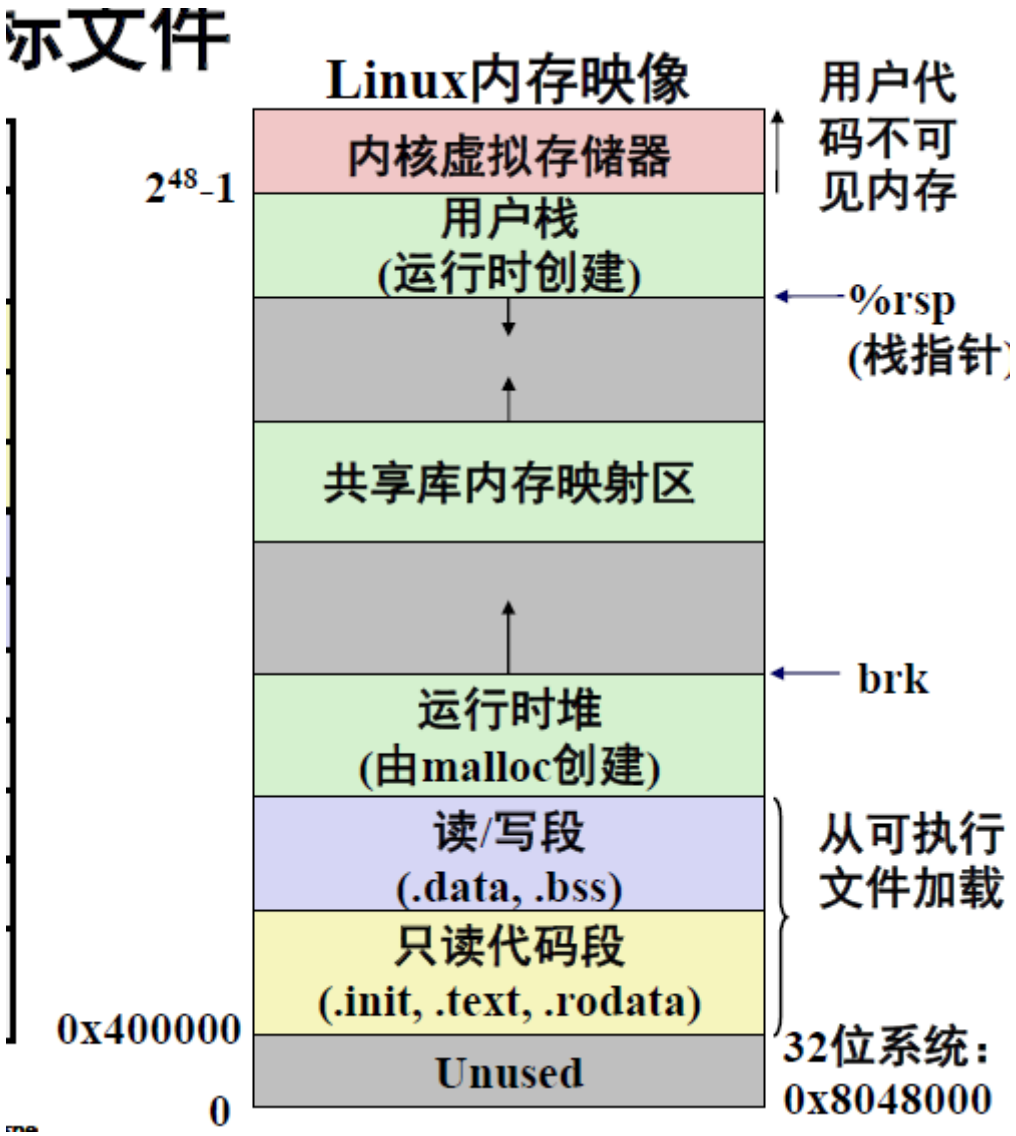
## 重定位

# 重定位PC相对引用sum

- 重定位条目信息
  - r.offset = 0xf
  - r.symbol = sum
  - r.type = R_X86_64_PC32
  - r.addend = -4
- 假设链接器已经确定
  - ADDR(s)=ADDR(.text)=0x4004d0
  - ADDR(r.symbol) = ADDR(sum)=0x4004e8
- 重定位
  - refaddr = ADDR(s) + r.offset = 0x4004d0 + 0xf = 0x4004df
  - *refptr = (unsigned)(ADDR(r.symbol) + r.addend - refaddr)
    =(unsigned)(0x4004e8 + (-4) - 0x4004df)
    =(unsigned)(0x5)

# 重定位绝对引用array

- 重定位条目信息
  - r.offset = 0xa
  - r.symbol = array
  - r.type = R_X86_64_32
  - r.addend = 0
- 假设链接器已经确定
  - ADDR(s)=ADDR(.text)=0x4004d0
  - ADDR(r.symbol) = ADDR(array)=0x601018
- 重定位
  - *refptr = (unsigned)(ADDR(r.symbol) + r.addend)
    =(unsigned)(0x601018 + 0 )
    =(unsigned)(0x601018)




# 链接库

## 静态库

*.a文件。将相关的可重定位目标文件连接到一个带有索引的存档文件中。增强连接器通过查找一个/多个存档文件中的符号来解析外部引用。如果存档中的一个成员文件解析了符号引用，就把它链接。

应该将库放在命令行的末尾。

### 缺点

1. 可执行文件的储存存在重复
2. 可执行文件的运行存在重复
3. 系统库的小错误要求每个程序显式重新链接

## 动态库

*.so, *.dll文件，包含代码和数据的目标文件，在加载或者运行时，被动态加载、链接。

共享库例程可以被多个进程共享。

### 加载时链接

当可执行文件首次加载和运行时进行动态链接；linux一般由动态链接器ld-linux.so自动处理。

c标准库libc.so一般进行动态链接。

### 运行时链接

程序开始运行后，进行动态连接；linux中一般通过调用dlopen()接口完成。一般用于分发软件、高性能服务器、运行时库打桩。

# 异常控制流

## 异常

异常是指为响应某个事件，将控制权交给操作系统内核的情况。

异常处理依赖异常表，这是一张跳转表。任何时间都有一个唯一的异常号。

### 中断

唯一的异步异常，由处理器外部I/O设备引起，由处理器的中断引脚指示；中断处理程序返回下一条指令处。

### 陷阱

有意的异常，是执行指令的结果。如系统调用等。

陷阱处理程序返回下一条指令处。

### 故障

可能被修复的异常错误，同步异常。如，缺页故障，保护故障，浮点异常。

处理程序要么重新执行引起故障的指令，要么终止。

### 终止

不可恢复的致命错误。如非法指令，奇偶校验错误，机器检查等。

终止当前程序。

## 进程

进程是一个正在运行的程序实例。

### 基本概念

#### 逻辑控制流

每个程序似乎独占地使用CPU；

这个是由OS内核通过上下文切换机制提供。

#### 私有地址空间

每个程序似乎独占的使用；

这是由OS内核的虚拟内存机制提供的。

### 上下文切换

进程由常驻内存的操作系统代码块（内核）管理。内核不是一个单独的进程，而是现有进程的一部分。

OS装载保存的寄存器，切换地址空间。

通过上下文切换，控制流从一个进程传递到另一个进程。

# 进程的状态

## 运行

进程要么正在执行，要么在等待被执行，且最终会被操作系统内核调度。

## 挂起/停止

进程的执行被挂起且不会被调度，直到收到新的信号

## 终止

进程永远的停止了。

# 进程的终止

进程会因为以下三种原因终止

- 从主函数返回
- 收到了一个默认行为是终止进程的信号
- 调用exit函数

```
void exit(int status);
```

exit函数以status退出来终止进程。

正常返回状态0，否则返回非0.和主函数中return一个值等效。

exit在运行过程中只能被调用一次，且不会返回。

# 获取进程ID

```
pid_t getpid(void);
//获取当前进程id
pid_t getppid(void);
//获取父进程id
```

# 创建进程

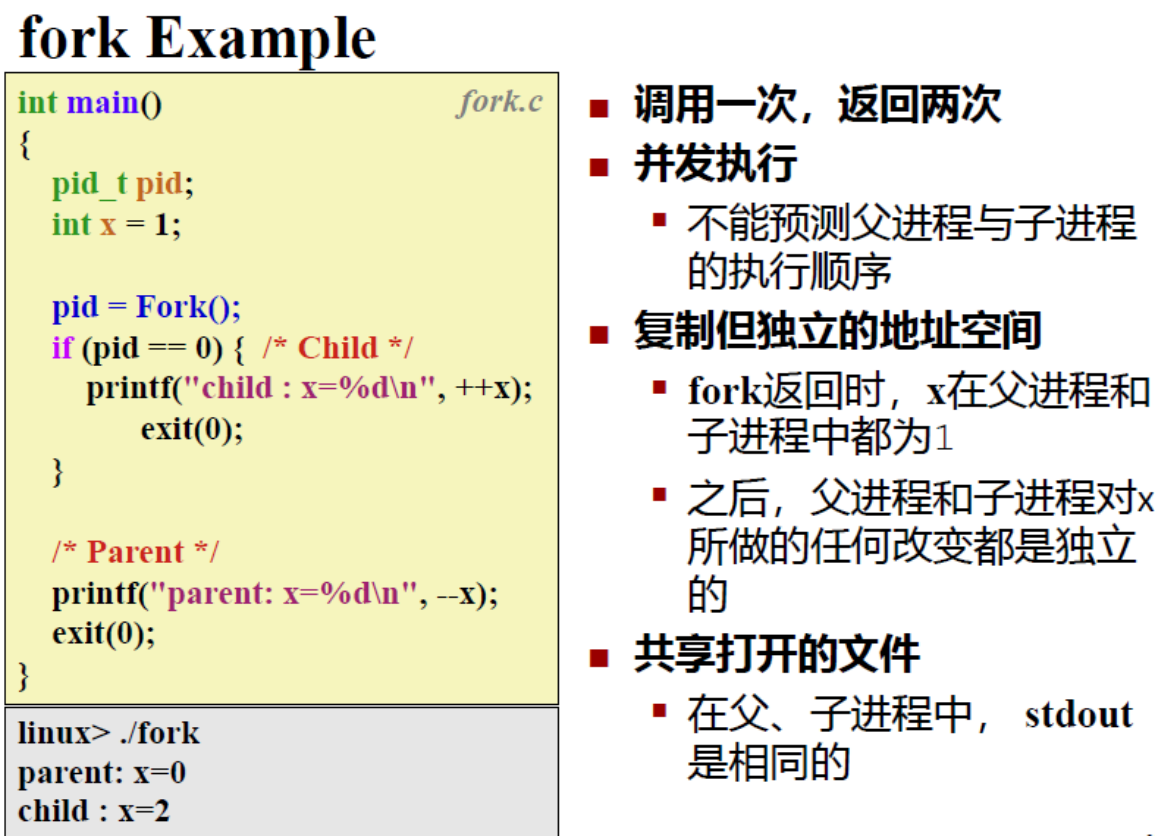
父进程可以通过调用fork函数创建一个新的、处于运行状态的子进程。

```
int fork(void);
```

该函数被调用一次，返回两次：子进程返回0，父进程返回子进程的id。

创建的子进程几乎与父进程完全相同。虚拟地址空间相同，但是实际映射的不是同一个；子进程pid与父进程也不同。

父子进程共享打开的文件(包括stdin，stdout，stderr)




## 子进程的回收

子进程终止后，仍消耗系统资源，称为僵尸进程。

父进程需要使用wait或waitpid回收子进程，当父进程收到子进程的退出状态后，内核将删掉僵死的子进程。

如果父进程没有回收子进程就终止了，那么内核将安排init进程去回收它们。

## 与子进程同步

使用wait函数可以与子进程同步。该函数将挂起当前进程的执行，直到它的一个子进程终止。wait函数返回终止的子进程的pid，并设置child_status指向的变量为推出信息。

```
pid_t wait(int* child_status);
```

waitpid函数提供了更强大的支持，有多种选项。

```
pid_t waitpid(pid_t pid, int* status, int options);
```

## 加载并执行程序

```
int execve(char* filename, char** argv, char** envp);
```

在当前进程中载入并执行程序。

filename是可执行文件，包括脚本和目标文件。

argv是参数列表，默认argv[0]==filename。

envp是环境变量列表。

产生的新进程将覆盖当前进程的代码、数据、栈，但是具有相同的PID，继承已打开的文件描述符和信号上下文。

除非指定文件不存在，否则调用一次不再返回。

## 系统调用错误的处理

当Linux系统级函数遇到错误时，一般返回-1并设置全局整数变量errno来标示出错误原因。

因此必须检查每个系统级函数的返回状态。

# 虚拟内存

## 寻址

主存的每一个单元拥有一个唯一的物理地址。那么CPU访问内存的最自然方式就是物理寻址。

虚拟寻址指的是CPU通过一个生成的虚拟地址来访问主存，这个虚拟地址在使用前会被转换成物理地址，这一过程称为地址翻译。这一过程需要CPU硬件(地址翻译单元MMU)与OS配合完成，利用位于主存的查询表进行地址动态翻译，该表内容由OS管理。

## 利用虚拟内存进行缓存

虚拟内存被组织成一个由存放在磁盘上N个连续的字节大小的单元组成的数组，每字节有一个唯一的虚拟地址。磁盘上的数据被拆分为虚拟页(VP)来传送到主存。虚拟页大小$P = 2^p$，同时，物理内存也被分割为物理页(PP)，大小相同。

任意时刻，虚拟页都有一个三划分：

1. 未分配的，还未创建的页，不占用磁盘空间；
2. 缓存的，已经缓存在物理内存的已分配页；
3. 未缓存的，未缓存在物理内存中的已分配页。

DRAM缓存采用全相联和写回。并采用更复杂的替换策略。

## 页表

页表储存虚拟页到物理页的映射，每次地址翻译硬件将虚拟地址转换为物理地址时，都会读取页表。操作系统负责维护页表的内容，并在磁盘和DRAM之间传送页。

页表就是一个页表条目(PTE)的数组。我们假设每个PTE由一个有效位和n位地址字段构成。有效位表示当前虚拟页是否被缓存。

## 地址翻译

### 符号

#### 基本参数

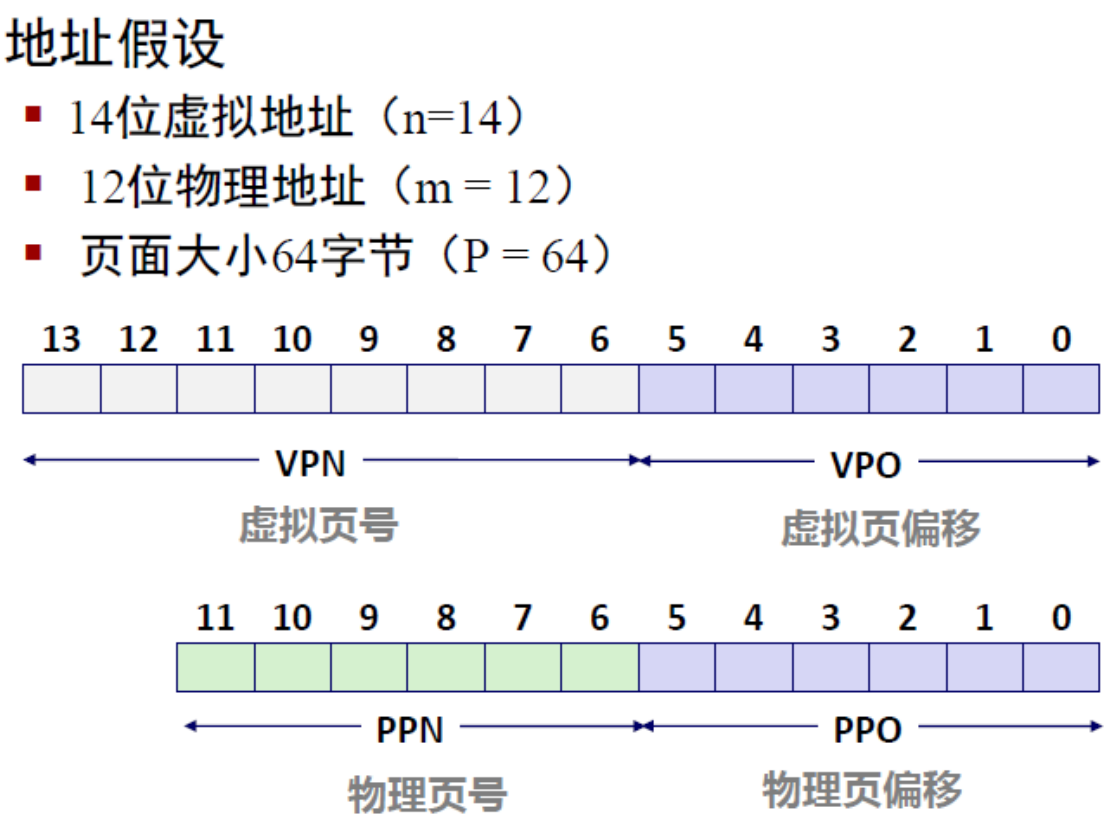
| 符号 | 描述 |
| --- | --- |
| $N = 2^n$ | 虚拟地址空间中的地址数量 |
| $M = 2^M$ | 物理地址空间中的地址数量 |
| $P = 2^p$ | 页的大小 |

**虚拟地址(VA)的组成部分**

| 符号 | 描述 |
| --- | --- |
| VPO | 虚拟页面偏移量/字节 |
| VPN | 虚拟页号 |
| TLBI | TLB索引 |
| TLBT | TLB标记 |

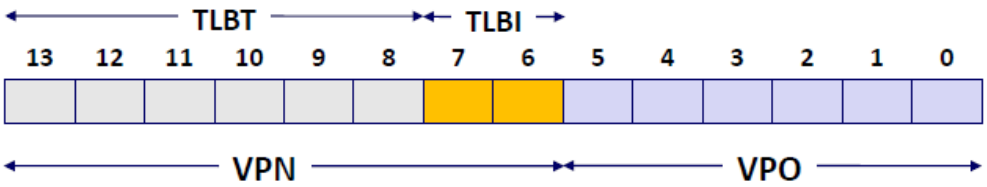
**物理地址(PA)的组成部分**

| 符号 | 描述 |
| --- | --- |
| PPO | 物理页面偏移量/字节 |
| PPN | 物理页号 |
| CO | Cache的字节偏移量 |
| CI | 高速缓存索引 |
| CT | 高速缓存标记 |



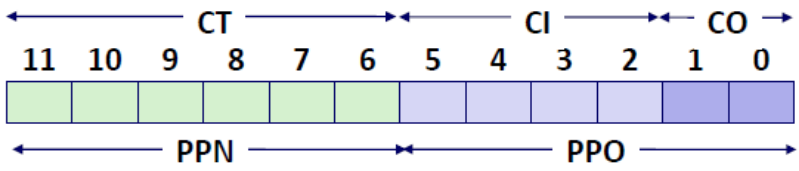
# 1. 小内存系统的 TLB

- 16个条目（ 16 entries ）
- 4路组相联(4-way associative)




# 3. 小内存系统的 Cache

- 16个组，每块为4字节
- 通过物理地址中的字段寻址
- 直接映射

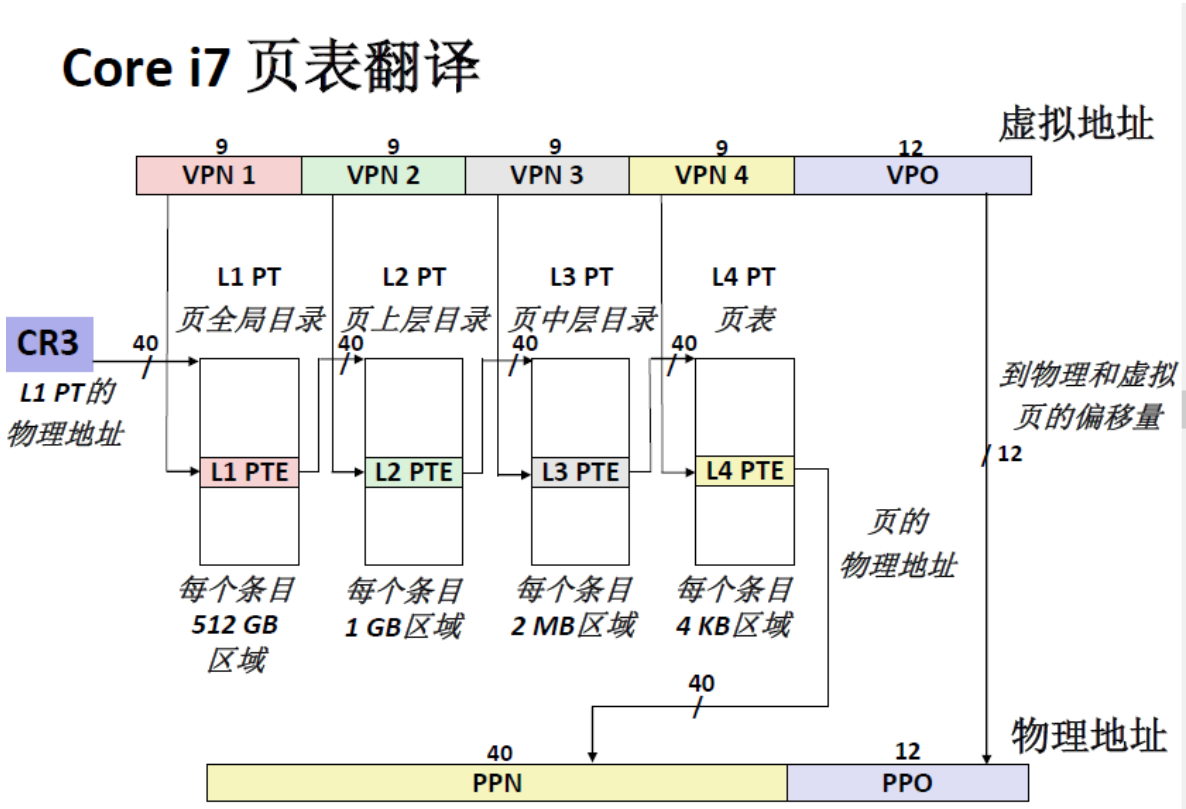



**TLB(翻译后备缓冲器)**

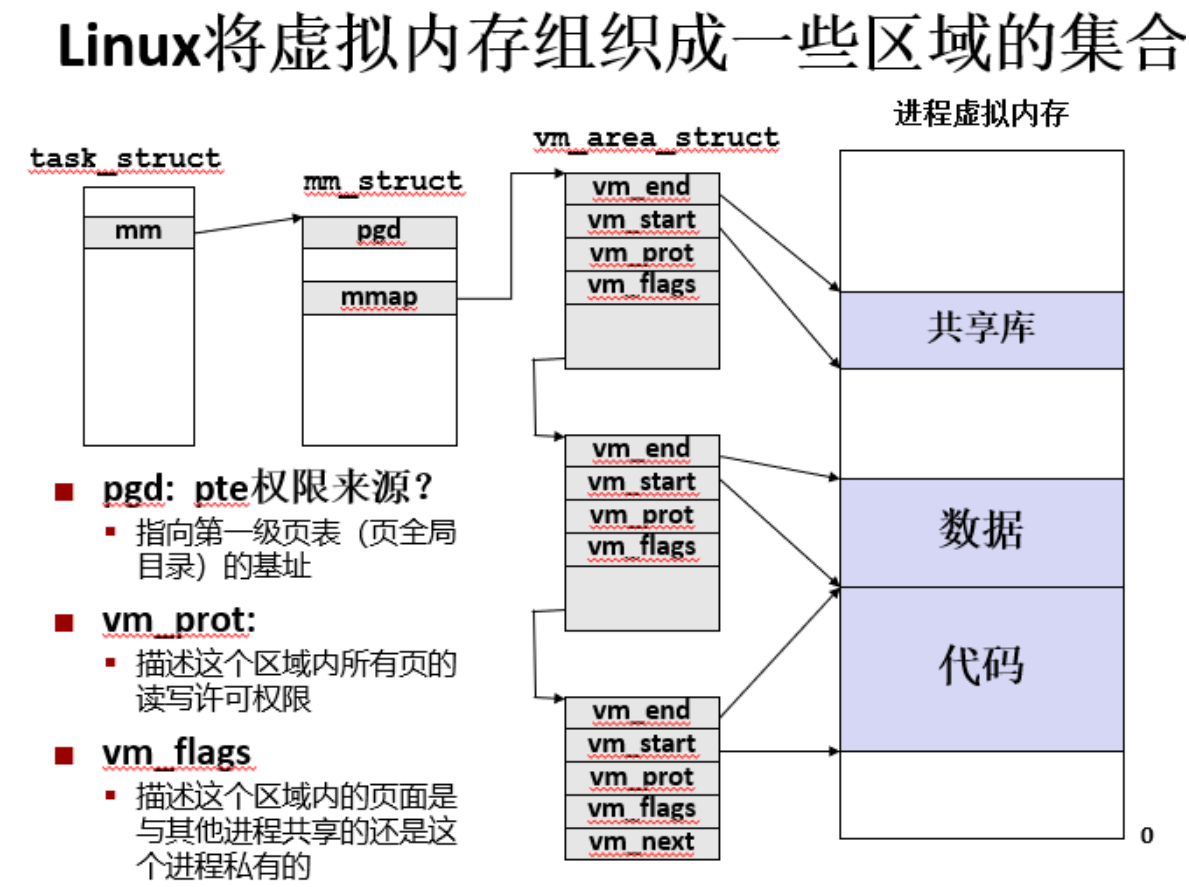
和Chache相似，只是储存的数据为PPN

**多级页表**

页表的条目不再指向某一页，而是某个页表。




## Linux的虚拟内存组织




# 内存映射

将虚拟内存区域与磁盘上的对象关联起来以初始化这个区域的内容，这一过程称为内存映射。

## 映射到的对象

可以是：

1.普通文件

文件区被分成页大小的片，对虚拟页面初始化。

2.匿名文件

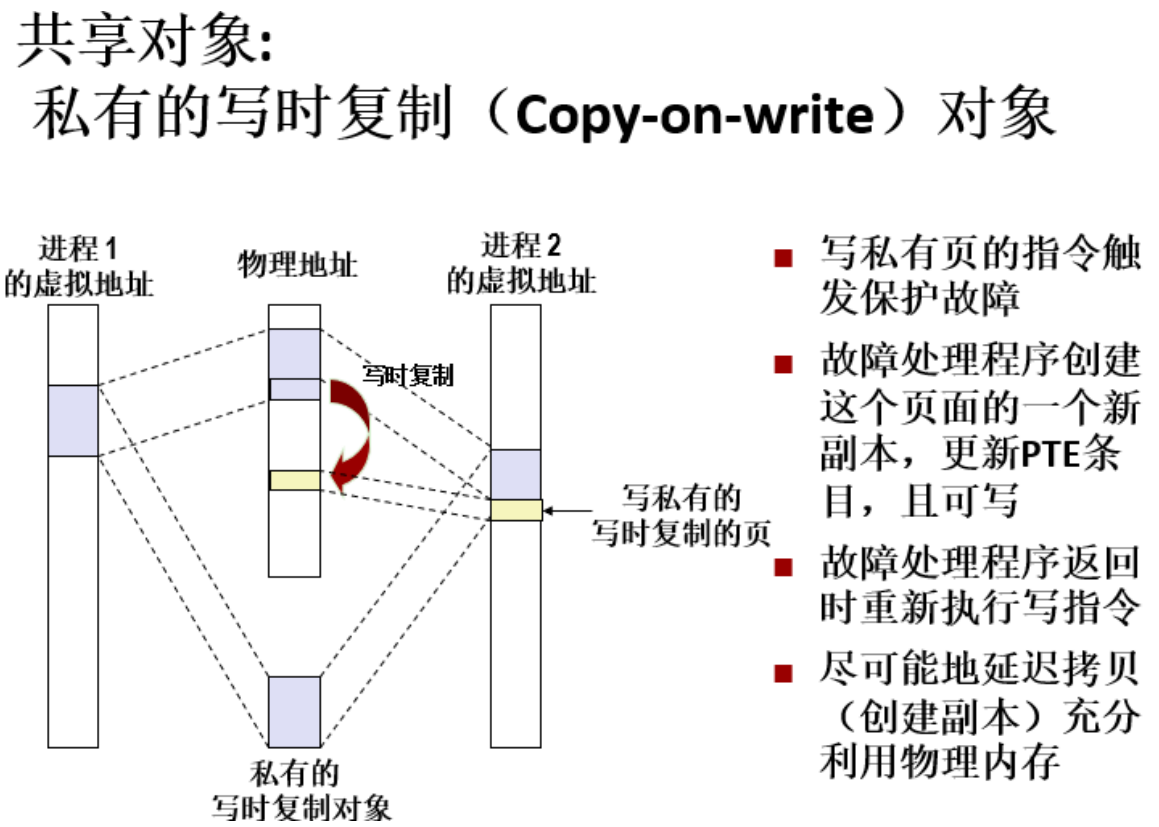
内核创建，全是二进制0.

第一次引用该区域内的虚拟页面时分配一个全是0的物理页；

一旦被修改，即和其它页面一样。

### 共享对象

对于一个物理地址的共享对象，可以拥有不同的虚拟地址。对于堆栈、数据、堆、寄存器等进行写时复制。




## 虚拟内存的其他作用

### 内存管理

虚拟内存保证了每个程序都可以有独立的地址空间。

这简化了共享和内存分配。

内存保护在PTE上添加额外的许可位来表示该页的访问权限。

## 动态内存分配

程序运行时，程序员使用动态内存分配器(malloc)获得虚拟内存。动态内存分配器维护着堆。

显示分配器，隐式分配器(垃圾收集器)

### GC

堆被视为有向图，根节点在堆外部则不是垃圾。

# 碎片

碎片化导致内存利用率低。

## 内部碎片

有效载荷小于块大小时会产生内部碎片。

产生于：

- 维护数据结构
- 增加块大小以满足对齐
- 显式的策略决定

特点是只取决于以前请求的模式；易于量化

## 外部碎片

当空闲内存合计起来足够满足一个分配请求，但是没有一个独立的空闲块足够大可以来处理这个请求时发生。

外部碎片难以量化。

# 问题的实现需要

## 知道释放多少

在块前面的word中记录该块的长度；每个分配块都需要一个这样的word。

## 记录空闲块

#### 隐式空闲链表

通过头部中的大小字段，隐含得连接所有块。

#### 显式空闲链表

在空闲块中使用指针记录下一个可用的块。

#### 分离的空闲列表

按照大小分类，构成不同大小的空闲内存。

#### 平衡树

在空闲块中维护一个带指针的平衡树，并使用长度作为权值。

# 隐式空闲链表

## 状态储存

块的大小和分配状态储存在一个Word中。

因为对其的块中，低阶地址位总是0，那么我们不储存这些0位，而是使用它作为一个分配状态标记；但是读取大小字段时，必须屏蔽它。

## 查找空闲块

#### 首次适配

从头开始搜索空闲链表，选择**第一个**合适的空闲块。

总块数的线性时间；在靠近链表起始处留下小空闲块的碎片。

**下一次适配**

从链表中上一次查询结束的地方开始，快于首次适配，避免了对无用块的扫描。

**最佳适配**

查询链表，选择最好的空闲块，剩余最少的空闲时间。运行速度较慢。

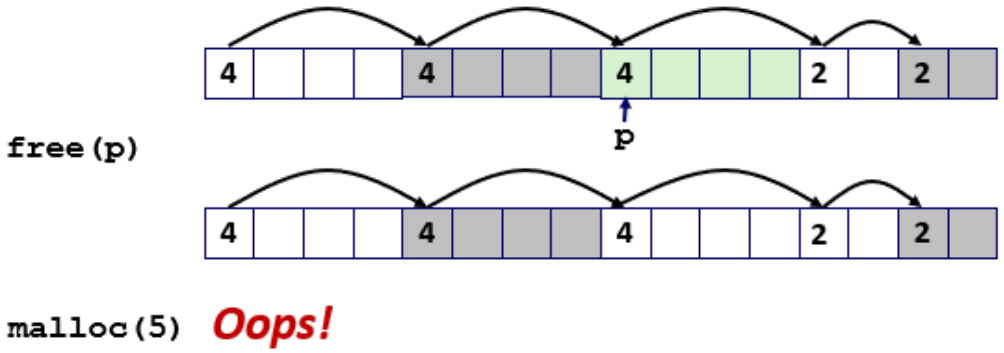
## 释放一个块

清除分配标记即可。但是会产生假碎片！

- 最简单的实现:
  - 清除“已分配 (allocated)”标志

```
void free_block(ptr p) { *p = *p & -2 }
```

  - 但有可能会产生“假碎片 (false fragmentation)”




**块的合并**

解决假碎片，只需进行合并相邻的空闲块。

**和下一块合并**

只需相加即可

```
void free_block(ptr p) {
    *p = *p & -2;
    // clear allocated flag
    next = p + *p;
    // find next block
    if ((*next & 1) == 0)
        *p = *p + *next;
    // add to this block not allocated
}
```

**双向合并**

在空闲块的底部添加标记(大小和分配状况)，允许我们反向查找链表。但是这样会造成内部碎片，小块浪费空间。

**四种合并情况**


用A(Allocated)和E(Empty)记作AA，AE，EA和EE记作四种情况。