计算机组成原理

张泽宇 2022012117 zhangzey22@mails.tsinghua.edu.cn

- [x] 比特与位级运算
- [x] 整数表示与运算
- [x] 浮点数
- [x] 过程调用
- [x] Y86指令集
- [ ] Y86顺序实现
- [ ] 高速缓存
- [ ] 高级程序性能优化

2. 信息的表示和处理

2.1 信息存储

- **字节**(byte)：8位的块，作为最小可寻址的内存单位
- **虚拟内存**(virtual memory)：将内存视为一个非常大的字节数组
- **地址**(address)内存的每个字节都由唯一的数字进行标识
- **虚拟地址空间**(virtual address space)：所有可能的地址组成的空间
- 字节的值域：一个字节的值域一般使用16进制表示，为$00-FF_{16}$
  - 在C语言中，以`0x`开头的数为16进制表示
- **字长**(word size)：指针数据的标称大小，假设字长为w bit，则使用该字长的计算机能够访问的虚拟地址范围是$0\rightarrow 2^w-1$
  - 32位计算机的虚拟地址空间约为4GB，64位计算机的虚拟地址空间约为16EB
  - 计算机和编译器支持多种不同方式编码的数字格式
  - 32比特字的地址偏移量是4,64比特字的地址偏移量是8
- 寻址和字节顺序
  - 大多数Intel兼容机使用**小端模式**存储数据，例如`0x12345678`将被存储为`78 56 34 12`
  - 例如，在C语言中，整数`15213`被存储为`6D 3B 00 00`
  - C语言中`%.2x`表示使用至少含有2个数字的16进制格式输出（即上文形式）
  - 字符串的字符编码显示`0x30+i`与字符的顺序相同，而与计算机的小端和大端无关

2.2 C语言中的位级运算

- 布尔预算：与、或、异或、非
- C语言中的位级运算：
  - `OR |`或
  - `AND &`与
  - `NOT ~`非
  - `XOR ^`异或

- C语言中的逻辑运算
  - `LOGIC_OR ||` 逻辑或
  - `LOGIC_AND &&` 逻辑与
  - `LOGIC_NOT !` 逻辑非
    - 注意逻辑运算和按位运算的差别：逻辑运算将所有非`0x00`的数视为TRUE1，而将`0x00`视为FALSE0
- C语言中的移位运算
  - `<<` 左移：在右侧的空余位添加0
  - `>>` 右移：**逻辑右移**：在空出的高位添加0；**算术右移**：将空出的高位添加和之前最高位一样的数字
    - C对于有符号数一般使用算术右移

2.3 整型数据类型

- 计算机中的整数实际上是一种模运算行为
- 无符号整数
  - 范围：$0 \to 2^w - 1$（表示为`0000...0` - `1111...1`）
- 有符号整数：最高位为符号位
  - 范围$-2^{w-1} \to 2^{w-1} - 1$（表示为`1000...0` - `0111...1`）
- 无符号整数和有符号整数之间的强制类型转换：不改变类模式（即二进制编码），只改变解释类模式的方式
  - T->U时负数被转换为大的正数，而在T/U交集范围内的正数保持不变
- C语言中的UT类型转换
  - 可以使用显示类型转换，例如`(int)`或`printf("%d",n)`等
  - 当将一种类型的整型赋给另一种类型时，发生隐式类型转换
  - 当表达式/算式中混合了unsigned和int时，将全部转换为**unsigned**进行计算
- 拓展数字的位表示
  - **零拓展**(zero extension)：将无符号数拓展为位更大的数据类型，只需要在前面添加0
  - **符号拓展**(sign extension)：在前面的所有拓展位添加符号位**sign bit**
  - 截断数字：直接丢弃高位。因此，截断数字是位溢出的一种表现形式

2.4 整数运算

- 相反数(negation)：$\sim x + 1 == -x$
  - 对于**补码**表示的最小整数`1000 0000 ..`，其非就是自己
- 无符号加法：当结果超过2^w时，最高位发生溢出，结果减小2^w
  - 无符号加法溢出检测：$s < x || s < y$
- 符号加法：当结果超过2^{w-1}时，最高位发生正溢出，截断的结果是减小2^w；当结果小于-2^{w-1}时，最高位发生负溢出，截断的结果是增加2^w

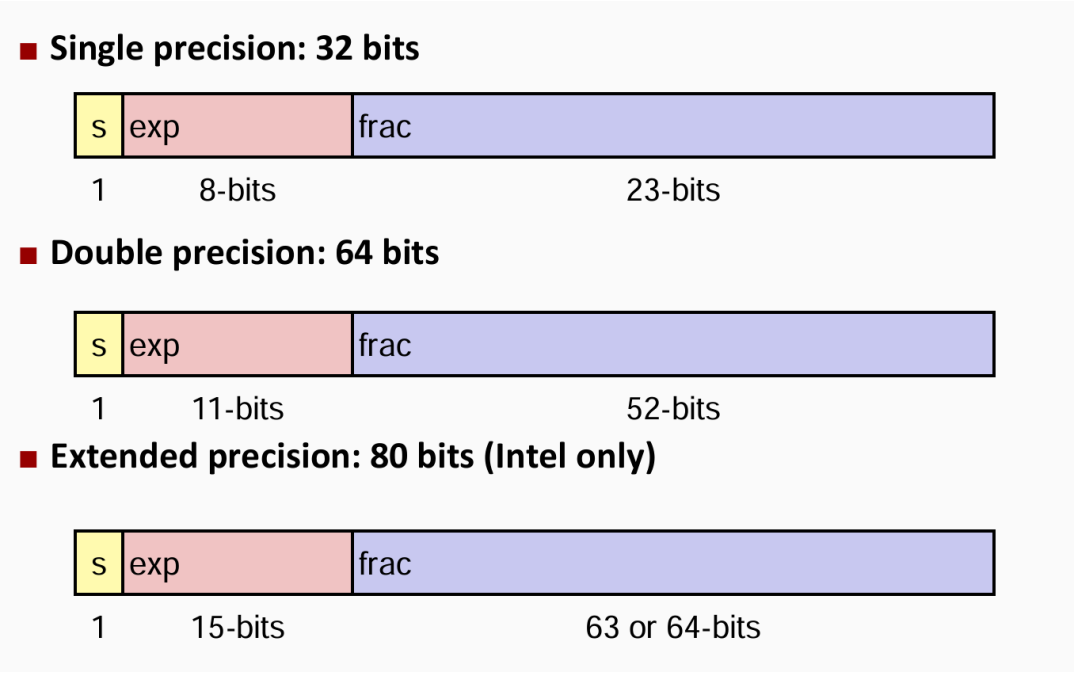
  位运算用 sign x signy signxy 实现
  - 符号加法溢出检测：$(x > 0 \&\& y > 0 \&\& s < 0) || (x < 0 \&\& y < 0 \&\& s \geq 0)$
  - 有符号整数的正数和负数相加，只会发生舍入不会发生溢出
- 乘法：有符号数和无符号数都是进行正数移位

    - 例如 `5*3(U)=101*011= 001100 + 000011 = 001111 == 111 = 7 = 15%8`
    - 例如 `-3*3(T)=101*011= -(011*011) = -(000110 + 000011) = -(001001) = 110111 = 111 = -1 = (-9)%8`
        - 可以看出有符号数和无符号数的结果都是真实的乘积对2^m取模
    - 乘法溢出检测：$m/y! = x$
- 乘以常数
    - `x << w`与$x * 2^w$在计算上等效
        - **左移指令**：`sall $w %eax`
    - `x >> w`与$\lfloor x/2^w \rfloor$在计算上等效
        - **算术右移指令**：`sarl $w %eax`
    - 当x<0时，需要$\lceil x/2^w \rceil = \lfloor (x + 2^w - 1)/2^w \rfloor$，即为`(x+(1<<w)-1)>>w`

2.5 浮点数

- **二进制小数**：小数点右侧的数字代表2的负幂，小数点左侧的数代表2的非负幂
    - 例$0.1_2 = 0.5;\ 0.101_2 = 0.625$
    - 二进制小数点左移一位相当于原数被2除

2.5.1 IEEE浮点数表示

- **基本形式**：$(-1)^s M * 2^E$
    - 符号位(sign, s)：表示这个浮点数是正数还是负数
    - 尾数(significand, M)：二进制小数，范围是$[1, 2) or [0, 1)$（右侧实际上是1或2减去一个很小的值）
    - 阶码(exponent, E)：对浮点数加权，权重是2的E次幂（E可能是负数）
    - 

- **规格化的值**：当E既不全为0，也不全为1时；E被表示为偏置有符号整数（即有偏移量$2^k - 1$的无符号整数）
    - 单精度(8 bit)：`-126-+127`，双精度(13 bit)：`-1022-+1023`
    - 例：1521310 = 11101101101101_2 = 1.1101101101101_2 x 2^13
        - frac部分存小数点后的部分，即`1101101101101`，后部使用0填补

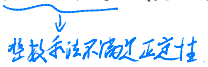
        - exp部分存`13+127=140`，即`10001100`
- **非规格化的值**：
    - **0**：E和M字段全为0，根据符号位决定是`+0.0`还是`-0.0`
    - 非常接近于0的数
- **特殊值**：
    - 无穷：E字段全为1，M字段全为0，可以表示**溢出**的结果 浮点数溢出结果必定为无穷
    - **NaN**（不是一个数）：E字段全为1，M字段不全为0，表示**非实数或无穷**的运算结果
- 浮点数可表示数的分布（以8位float为例）

。

描述	位表示	e字段	E值	2^E	frac字段	M值	值	化简值	十进制
0	0 0000 000	0000	-6	1/64	000	0/8	0	0	0.0
第小正非规格数	0 0000 001	0000	-6	1/64	001	1/8	1/512	1/512	0.001953
最大正非规格数	0 0000 111	0000	-6	1/64	111	7/8	7/512	7/512	0.013672
最小正规格化数	0 0001 000	0001	-6	1/64	000	1+0/8	8/512	1/64	0.015625
1	0 0111 000	0111	0	1	000	1+0/8	1	1	1
最大正规格化数	0 1110 111	1110	7	128	111	1+7/8	1920/8	240	240.0
无穷大	0 1111 000	1111	-	-	000	-	-	∞	-

- 浮点数的精度随靠近0而增加；float**无法表示所有的**int值
- 假设k位阶码和n为尾数
  - 最小正非规格数: $\frac{1}{2^{2^{k-1}-2}*2^n}$
  - 最大正非规格数：$\frac{2^n-1}{2^{2^{k-1}-2}*2^n}$

        - 最小正规格化数：$\frac{1}{2^{2^{k-1}-2}}$
        - 最大正规格化数：$\frac{2^{2^{k-1}-1}*(2^{n+1}-1)}{2^n}$
- 浮点数的舍入
    - **向偶数舍入**(round-to-even)：默认的舍入方式，也被称为向最近的值舍入
        - 符合统计学规律
        - 在二进制小数中，实现为**尽量使最低位有效数字为0**
    - 向零舍入：向绝对值较小的方向舍入
    - 向上舍入：无论正数还是负数均向上方舍入
    - 向下舍入：无论正数还是负数均向下方舍入
- 浮点运算：
    - 浮点数加法**具有单调性**（除去无穷和NaN），**不具有结合性**（因为未对齐的浮点数在加法运算会有舍入）
    - 浮点数乘法满足**单调性**（除去无穷和NaN），满足**正定性**（除去无穷和NaN），不满足**分配律和结合律**



- C浮点数
    - 所有的C语言版本提供了两种不同的浮点数类型：`double`（双精度浮点数）和`float`（单精度浮点数）
        - I2F：不会溢出，但可能被舍入
        - I2D，F2D：可以保留精确数值
        - D2F：可能溢出，可能舍入
        - F2I，D2I：可能溢出，可能（向0）舍入
        - double只能保证在int范围内的精度表示。如果使用的整数类型是long long，同样会发生精度损失。

3. 程序的机器级表示

3.1 算数与逻辑操作 （略）

- x86汇编的数据结构
    - 整型数据(1,2,4 bytes)
    - 浮点类型数据(4,8,10 bytes)
    - 动态性类型数据：无固定大小，如数组、结构等
- x86的汇编指令
    - 对寄存器或内存数据执行算术函数
    - 在内存和寄存器之间交换数据
    - 过程控制

3.1.1 汇编基础

- IA32寄存器
    - 通用寄存器

0-32bit	0-16bit	8-16bit	0-8bit	Origin
%eax	%ax	%ah	%al	累加结果/返回值
%ecx	%cx	%ch	%cl	计数器
%edx	%dx	%dh	%dl	数据
%ebx	%bx	%bh	%bl	数组指针
%esi	%si	-	-	源索引
%edi	%di	-	-	目的索引
    - 栈寄存器
      - **%esp**：栈指针寄存器，指向系统最上面一个栈帧的栈顶
      - **%ebp**：帧指针寄存器，指向系统最上面一个栈帧的栈底
    - x86_64的寄存器：%r8/%r9/%r10/%r11/%r12/%r13/%r14/%r15
- 操作数指示符
    - **移动/赋值指令**：`movl source dest`
      - `movb/movw/movl/movq`分别对应移动1,2,4,8字节
    - 操作数：
类型	格式	操作的数值	名称	示例
立即数	$Imm	Imm	立即数寻址	movl $0x400, %eax
寄存器	%ra	R[ra]	寄存器寻址	movl %eax, %edx
存储器	Imm	M[Imm]	绝对寻址（通过数）	movl 0x400, %eax
存储器	(ra)	M[R[ra]]	间接寻址（通过寄存器的值）	movl (%eax), %ebx
存储器	Imm(ra)	M[R[ra]+Imm]	基址+偏移量寻址	movl 4(%esp), %eax
      - x86不支持**内存到内存**的数据移动
      - x86还支持其他的寻址方式。$D(Rb,Ri,S) \rightarrow Mem[Reg[Rb]+Reg[Ri]*S+D]$
        - 总体而言，对于()内部，第一个参数和第二个参数均为基址或线性偏移量，第三个参数需要和第二个参数相乘
        - ()外部的Imm一定是线性偏移量
        - 例如，Rb为数组头，Ri为引用变量（数组下标），S为数组对象的长度，D为地址偏移量（如++,--等操作）
      - **访存式计算指令**：`leal source dest`（`leaq`指令没有其他大小的变种）
        - 例`leal (%eax, %eax, 3) %eax`对应的计算是`x=x+x*3`
        - leal不涉及内存的计算

3.2 控制（略）

3.2.1 计算指令

- **加法指令**：`addl src, dest`：dest = dest + src
- **减法指令**：`subl src, dest`：dest = dest - src
- **乘法指令**：`imull src, dest`：dest = dest * src
- **左移指令**：`sall $w, dest`
- **算术右移指令**：`sarl`
- **逻辑右移指令**：`shrl`
- **按位异或指令**：`xorl src, dest`：dest = dest ^ src
- **按位与指令**：`andl src, dest`：dest = dest & src
- **按位或指令**：`orl src, dest`： dest = dest | src
- **自增指令**：`incl dest`：dest = dest + 1
- **自减指令**：`decl dest`：dest = dest - 1
- **取负指令**：`negl dest`：dest = -dest
- **取反指令**：`notl dest`：dest = ~dest

3.2.2 过程状态

- **指令寄存器**：**%eip**：存储正在执行的指令的地址
- 条件码
  - **CF**：发生无符号数进位溢出时，置为1
  - **ZF**：指令计算结果为0时，置为1
  - **SF**：指令计算结果为负数时，置为1
  - **OF**：发生双精度有符号数溢出时，置为1
- **比较指令**：`cmpl src, dest`：隐式计算`dest-src`，当`dest==src`时，ZF置为1，当`dest<src`时SF置为1
- **查零指令**：`test src1, src2`：隐式计算`src1&src2`，当与结果为0时，ZF置为1（因此`test a, a`可用于检查a是否为0）
- **访问条件码指令**：`setX dest`：使用X代表的条件码组合的值设置dest的值

3.2.3 分支与跳转

- 跳转
  - **无条件跳转指令**：`jmp .Ls`：运行到该指令时无条件跳转到代码段Ls
  - **条件跳转指令**：`jX .Ls`：运行到该指令时隐式调用条件码，如果满足条件则跳转到Ls
- **条件赋值指令**：`cmovX src, dest`：隐式调用条件码，如果条件码满足X条件，进行move操作
- 循环结构：`{循环体|跳出循环指令|开始循环指令}`

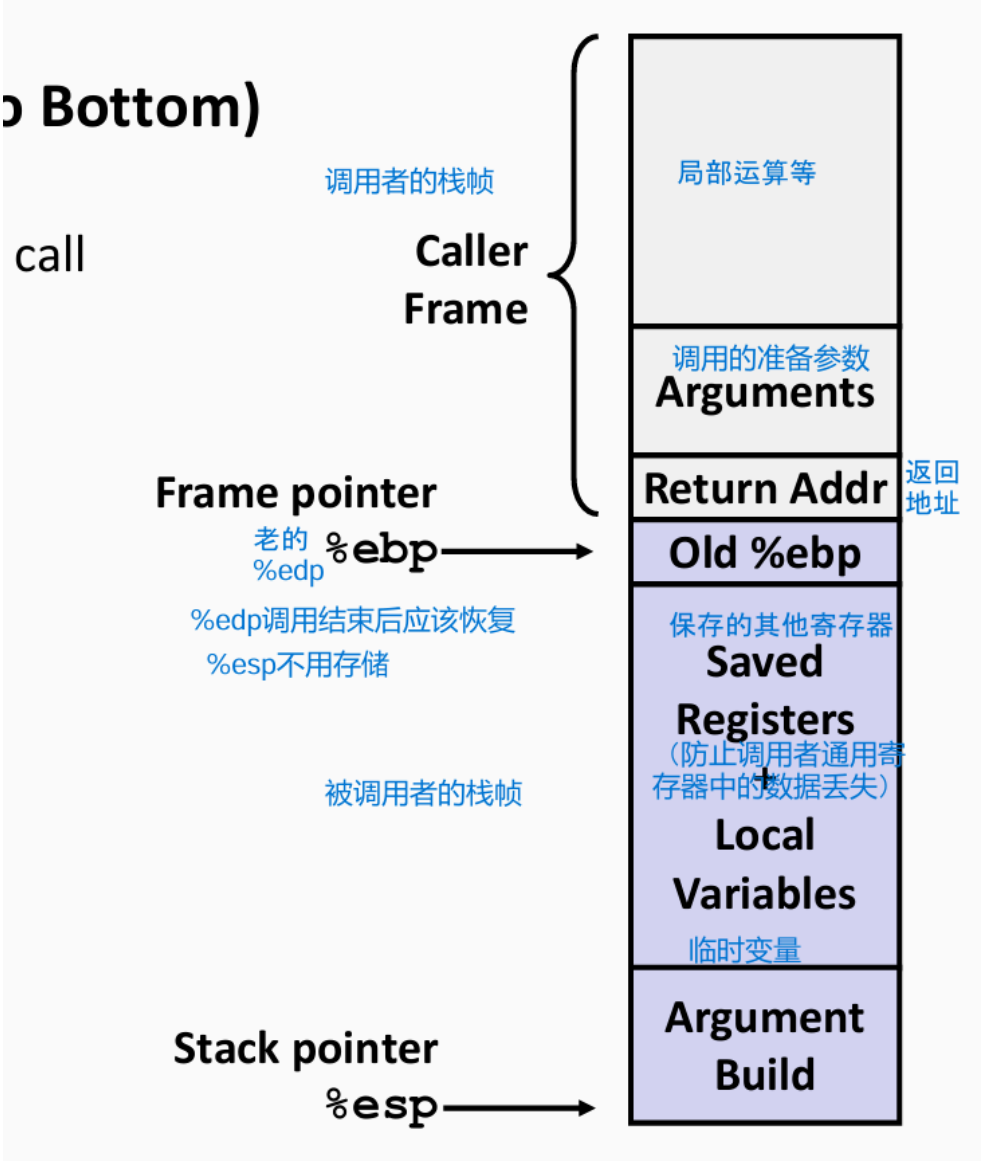
3.3 过程调用

3.3.1 IA32运行时栈结构

- **入栈指令**：`push src`
  - 将栈顶指针**%esp**减少**4**（即将栈的内容扩大32位）
  - 将`src`指向的内容写入%esp指向的地址
- **出栈指令**：`pop dest`
  - 将%esp指向的内容写入dest
  - 将栈顶指针**%esp**增加**4**（即将栈的内容减少32位）

3.3.2 转移控制

- **转移指令**：`call Procedure`
  - 将**下一条指令的地址**压入栈，作为返回地址
  - 将%eip设置为Procedure的入口指令
    - ```
      call next ;call 将next的地址压入栈顶，此时%eip为栈顶元素
      next:
      popl %eax ;将栈顶元素移入%eax
      ```
    - 这是唯一获取%eip内容的方式
- **返回指令**：`ret`
  - 从栈中弹出**返回地址**，将%eip设置为返回地址
- 传递参数
  - x86中传递参数是通过寄存器和栈实现的
    - 如果调用者使用`cdecl`约定或`stdcall`约定，调用者会将所有参数压入栈（则在return address+4/+8/...，依次类推）
    - 如果调用者使用`fastcall`约定，调用者会优先按固定的顺序将参数转移至空闲的寄存器，并将多余的参数压入栈
- 栈帧
  - x86中过程使用的栈是以**栈帧**(stack frame)为单位在运行时栈上划分的。
    - 每个过程拥有自己独立的栈帧，每递归一次增加一个栈帧
    - 栈帧的底部地址存储在**%ebp**中，栈帧的顶部地址存储在**%esp**中
    - 栈帧的底部4个字节存储**旧的%edp**，即该过程的调用者所拥有的栈帧的%ebp
      ```



- 调用者的数据准备

```
call_swap:
...
   subl    $8, %esp ;给栈上新增存储两个32位数据的空间
   movl    $course2,   4(%esp) ;放入第二个参数
   movl    $course1,   (%esp) ;放入第一个参数
   call    swap ;准备完毕，压入返回地址并调用进程
```

- 被调用者的准备

```
swap:
   pushl   %ebp ;将老%ebp内容压入自己栈帧
   movl    %esp,   %ebp ;同步%esp,%ebp指向的地址，使自己的栈帧中只有老%ebp一个元素
   pushl   %ebx ;将被调用者保存的寄存器压入自己的栈帧

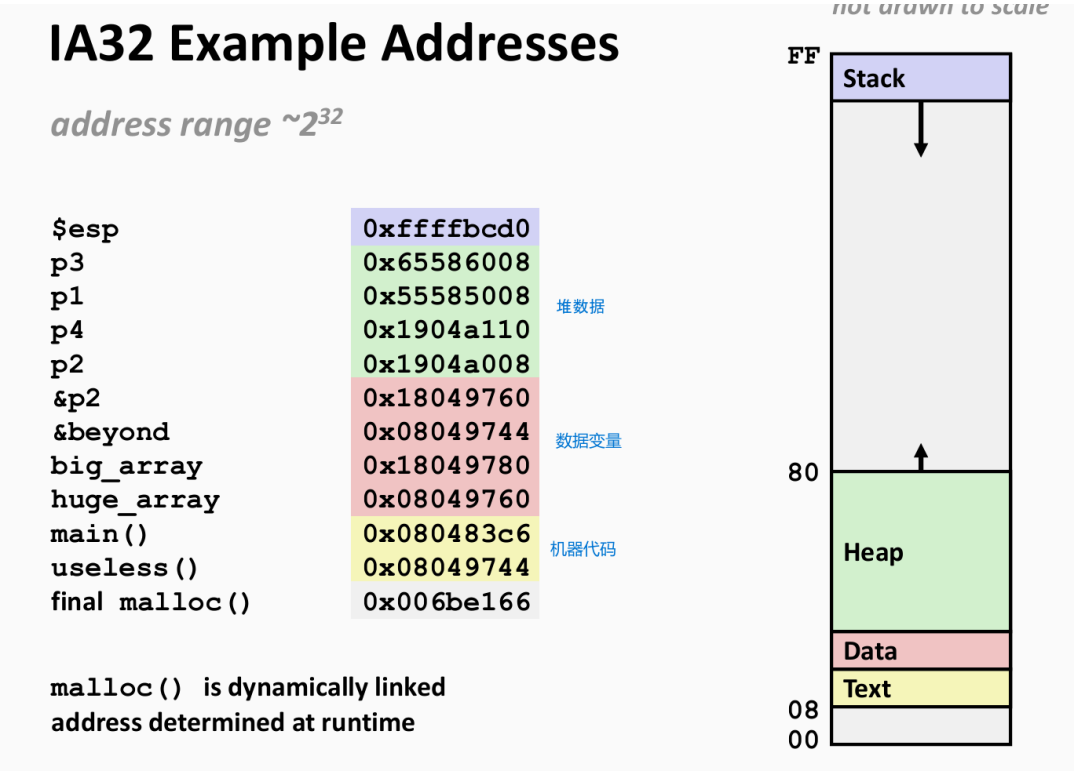
   movl    8(%ebp),    %edx ;拿数据course1（因为ret和老%ebp各占4字节，所以偏移量为8字节）
   movl    12(%ebp),   %ecx ;拿数据course2
   movl    (%edx), %eax
   movl    (%ecx), %ebx
   movl    %ebx,   (%edx)
   movl    %eax,   (%ecx)

```

```
13    popl    %ebx ;将老%ebx弹入%ebx
14    popl    %ebp ;将老%ebp内容弹入%ebp（由于popl指令自动移动%esp，现在%esp已经指向call_swap栈帧的返回地址）
15    ret ;准备就绪，返回被调用者
```

    - 在正常情况下，被调用者返回后，其栈帧内的内容应该正确弹出，各寄存器恢复原状
- 寄存器使用
    - **调用者保存**：%eax/%edx/%ecx
    - **被调用者保存**：%ebx/%esi/%edi
    - **被调用者负责恢复原状**：%esp/%ebp

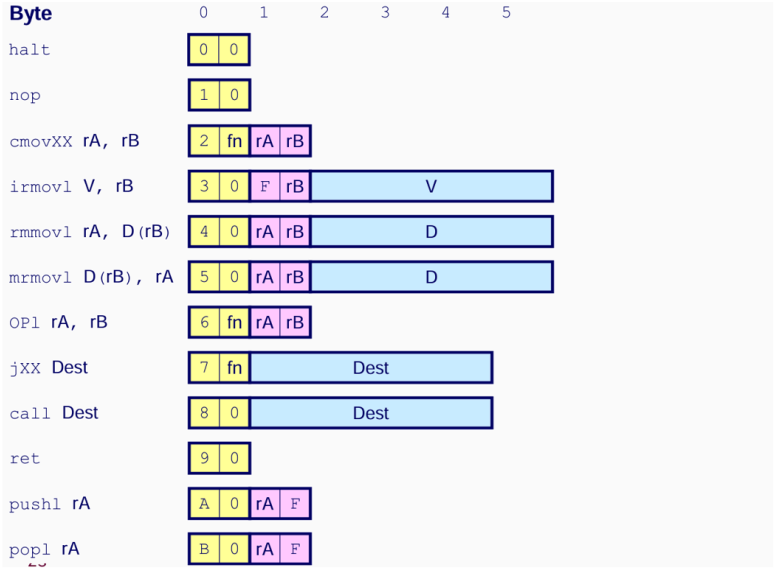
3.4 内存越界应用和缓冲区溢出

- IA32的动态数据结构
    - 

- 缓冲区溢出
    - 通常，在栈中分配某个字符数组来保存一个字符串，但是字符串的长度超过了为数组分配的空间
    - 例子：假设栈上开辟了4字节的buf[4]空间，存储4字节的老%ebx，4字节的老%ebp
        - 输入`abc\0`（注意C语言会在字符串末尾自动添加EOF）：无事发生
        - 输入`abcdefg\0`：缓冲区溢出4字节，覆盖%ebx，程序不会立即崩溃（取决于%ebx的功能）
        - 输入`abcdefghijk\0`：缓冲区溢出8字节，覆盖%ebp，栈帧无法恢复为原位置
        - 输入`abcdefghijkl\0`：缓冲区溢出9字节，覆盖return address，程序无法正常返回

4 处理器体系结构 ISA

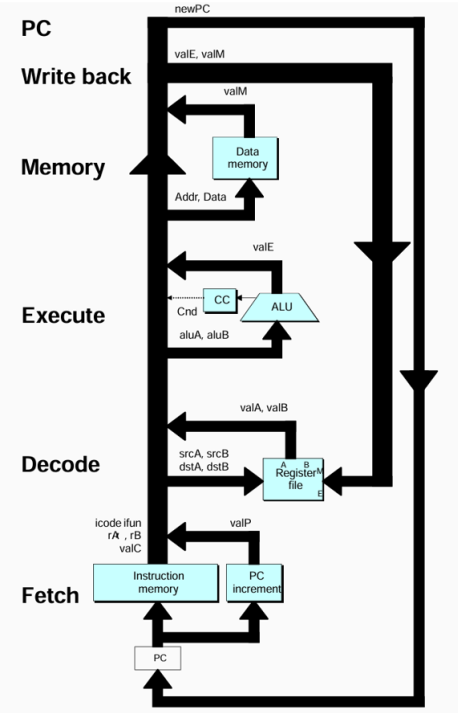
4.1 y86指令集

- y86系统的状态
  - 寄存器：通用寄存器的栈指针寄存器与x86的相同
  - 条件码：ZF，SF，OF
  - y86使用虚拟地址引用内存位置
  - y86使用stat状态码展示程序的运行状态
- y86指令集
  - 

  - **条件传送指令**(cmovXX)：共6条，当条件码满足约束时，更新目的寄存器的值
  - **移动/赋值指令**(xxmovl)：共4条，首部的两个字母表示数据的源和目的地
  - **整数操作指令**(opl)：共4条，包含加、减、按位与和异或，只对寄存器进行操作
  - **跳转指令**(jXX)：共7条，根据分支指令的类型和条件代码的设置选择分支
  - **调用/返回指令**(call/ret)：call将dest放入栈，ret弹出地址并返回
- y86指令编码：每条指令需要1-6个字节不等
  - **指令类型**（第1字节）：
    - 代码部分（0-4bit）
    - 功能部分（4-8bit）
  - **寄存器指示符**（如果有，第2字节）：指定寄存器的标识符（0x0-0xE）
    - 指定两个寄存器：`rA|rB`
    - 指定一个寄存器：`0xF|rB` || `rA|0xF`
  - **常数字**（如果有，第3-6字节或第2-5字节）：使用小端编码的32位常数
    - `irmovl` 指令移入寄存器的立即数
    - `rmmovl` 访问内存的偏移量
    - `mrmovl` 访问内存的偏移量
    - `jXX|call` 分支跳转地址
- y86的过程栈和过程调用：基本和x86相同

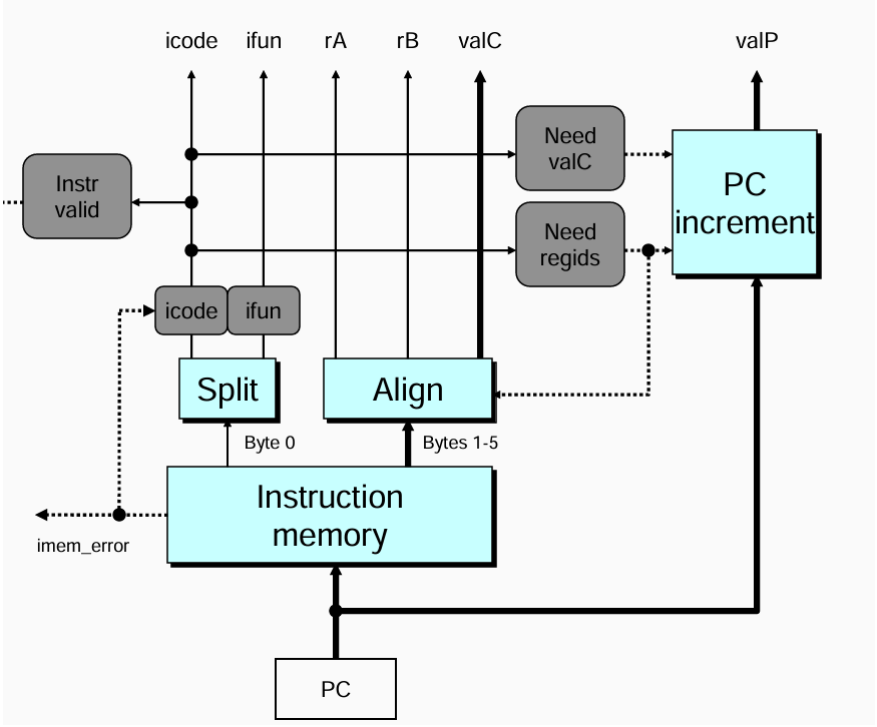
- y86的其它指令
  - **跳过指令**(nop)：跳过，执行下一条指令
  - **终止指令**(halt)：结束指令操作

4.2 y86顺序处理器



- y86顺序处理器的阶段
  - **取指**(Fetch)：将程序计数器寄存器作为地址，指令内存读取指令的字节
    - icode/ifun：指令的代码部分和指令的功能部分，访问固定为`M[PC]`
    - rA/rB：读取操作数的寄存器，如果存在访问固定为`M[PC+1]`
    - ValC：读取指令中的常数字，如果存在访问形式可能是`M[PC+1]`或`M[PC+2]`
    - ValP：下一条指令的地址，一般为`M[PC+正在处理指令的字节数]`
  - **译码**(Decode)：从寄存器的读端口读取寄存器值
    - valA<--srcA：操作数A，更新形式为`valA<--R[rA]`
    - valB<--srcB：操作数B，更新形式为`valB<--R[rB]`
  - **执行**(Execute)：将算术/逻辑单元用于指令的运算操作（包括计算操作数，给栈指针+4-4，计算偏移量）
    - valE：计算得到的值，一般计算形式为`valE<--function(valA,valB,valC,valP)`
  - **访存**(Memory)：数据内存读出或写入一个内存字
    - valM：从内存中读出的数据。如果valE是偏移量，访问形式为`valM<--M[valE]`
  - **写回**(Write back)：向寄存器文件的写端口写入数据
    - dstE<--valE：向寄存器E端口写入valE，写入形式为`R[要写入的寄存器]<--valE`
    - dstM<--valM：向寄存器M端口写入valM，写入形式为`R[要写入的寄存器]<--valM`
  - **更新PC**(PC)：更新下一条指令的地址
    - PC：下一条指令的地址，一般更新形式为`PC<--valM`

	OPI	RMMOVE	POPL	JMP
取址	$icode:ifun \longleftarrow M_1[PC]$ $rA:rB \longleftarrow M_1[PC+1]$ $valP = PC+2$	$icode:ifun \longleftarrow M_1[PC]$ $rA:rB \longleftarrow M_1[PC+1]$ $valC \longleftarrow M_4[PC+4]$ $valP \longleftarrow PC+6$	$icode:ifun \longleftarrow M_1[PC]$ $rA:rB \longleftarrow M_1[PC+1]$ $valP \longleftarrow PC+2$	$icode:ifun \longleftarrow M_1[PC]$ $valC \longleftarrow M_4[PC+1]$ $valP \longleftarrow PC+5$
译码	$valA \longleftarrow R[rA]$ $valB \longleftarrow R[rB]$	$valA \longleftarrow R[rA]$ $valB \longleftarrow R[rB]$	$valA \longleftarrow R[\%esp]$ $valB \longleftarrow R[\%esp]$	
执行	$valE \longleftarrow valA\ OP\ valB$ set CC	$valE \longleftarrow valB+valC$	$valE \longleftarrow valB+4$	$Cnd \longleftarrow Cond(CC,ifun)$
访存		$M_4[valE] \longleftarrow valA$	$valM \longleftarrow M_4[valA]$	
写回	$R[rB] \longleftarrow valE$		$R[\%esp] \longleftarrow valB$ $R[rA] \longleftarrow valM$	
PC更新	$PC \longleftarrow valP$	$PC \longleftarrow valP$	$PC \longleftarrow valP$	$PC \longleftarrow Cnd?valC:valP$

- y86顺序处理器的实现
  - 取址（例子）
    - 

      - 先取出所有数据，再使用函数逻辑判断是否需要这些数据
  - 其他模块的逻辑都是相同的，均为使用**指令判断**函数指导电路实现相应的功能

5 存储器

5.1 存储器结构

- 随机访问存储器
  - 静态RAM(SRAM)：使用双稳态的存储器单元存储数据
  - 动态RAM(DRAM)：使用电容充电存储位数据
    - DRAM使用d个含w个位存储单元的**超单元**存储数据（因此DRAM总共存储的数据为dw位）
    - DRAM对超单元寻址为(r,c)，对超单元内部寻址为(i,j)
      - 因此，DRAM对位信息读取是将整行读入缓冲区，再从行缓冲区中读取列

        - DRAM存储也是按行存储，例如对8个8M*8DRAM芯片，一个64位字将被拆成8个长度为8的行（字节），按顺序存入8个DRAM超单元中
- 非易失性存储器
    - 磁盘
        - 容量：`扇区字节数 * 磁道平均扇区数 * 表面磁道数 * 2 * 磁盘盘片数`
        - 访问时间：`寻道时间 * 旋转时间 * 传送时间`

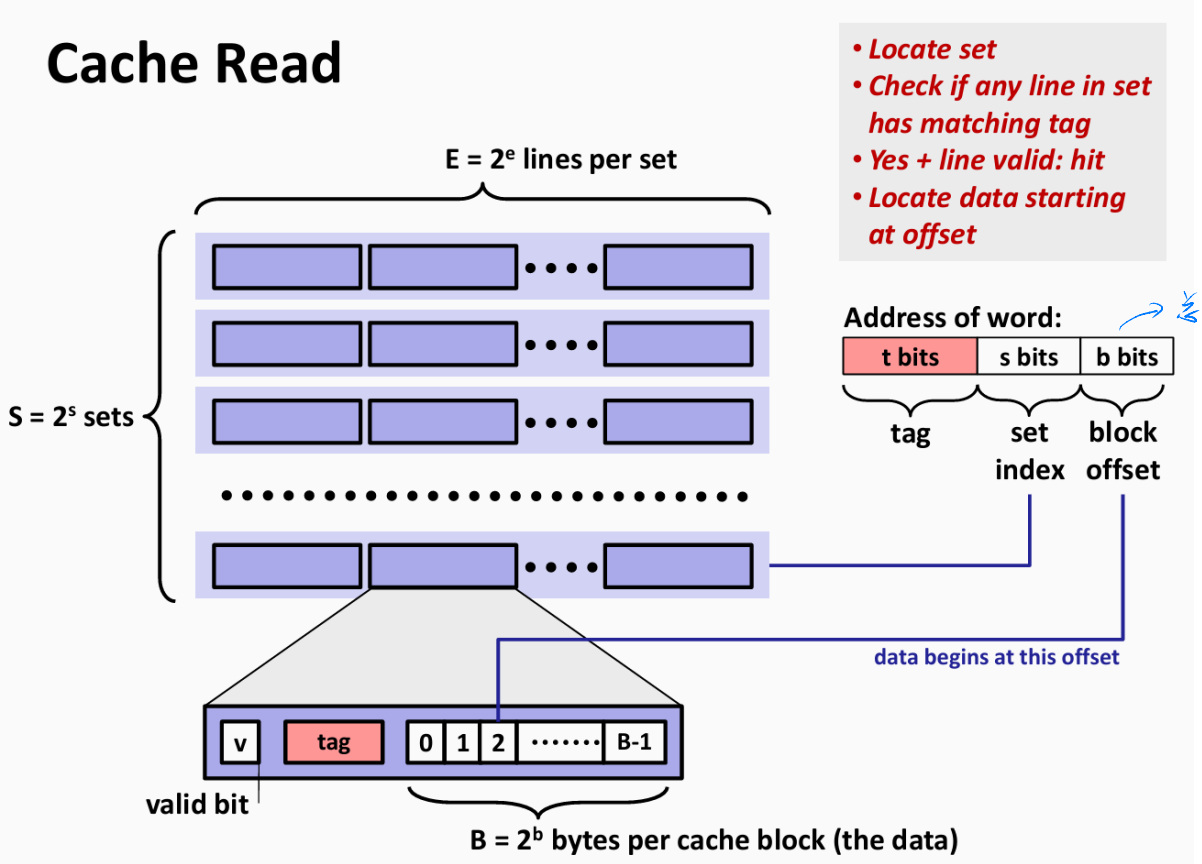
5.2 局部性

- **时间局部性**：被引用过一次的内存位置可能在不久的将来被再次引用
- **空间局部性**：如果一个内存位置被引用了一次，那么程序很可能在不远的将来引用附近的一个内存位置
- 对程序数据引用的局部性
    - 线性表：步长为1的引用模式
    - 多维表：按照**行优先顺序**读取元素，即读取完第一行的元素，再读取第二行的元素

5.3 存储器结构和高速缓存

- 存储器结构：寄存器--L1--L2--L3（高速缓存）--主存（DRAM）--本地二级存储（磁盘）--远程二级存储
- **高速缓存**：对于每个k，位于k层的更快更小的存储设备作为位于k+1层的更慢更大的存储设备的缓存
    - 数据以**块**为传送单元在第k层和第k+1层之间来回复制，一般k越小，k和k+1层时间的块大小越小
- 缓存命中：
    - **缓存命中**：当程序需要k+1层中的某个数据块d时，在k层中查找并读取了数据块d
    - **缓存不命中**：当程序需要k+1层中的某个数据块d时，在k层中没有缓存数据块d
        - **替换块**：从k+1层中取出包含d的块，并覆盖k层中现有的一个块
        - 覆盖现有块取决于缓存的替换策略（随机替换/LRU）
    - 缓存不命中的种类
        - **冷不命中**：第k层高速缓存是空的
        - **冲突不命中**：由于限制性的放置策略（例如将k+1层的某些块固定映射到k层的一个块中）导致的不命中
        - **容量不命中**：由于程序在某一阶段的工作集超过了缓存的大小导致的不命中

5.4 高速缓存结构



- 高速缓存的组织结构：**四元组(S, E, B, m)**
  - **物理地址位数**m：按顺序有t位标记，s位组索引和b位块偏移
  - **组数**S：高速缓存中组的数量。每组对应m中的一个组索引
  - **行数**E：高速缓存中每个组中含有行的数量。每行由一个tag作为索引，这个tag对应m中的t位标记
  - **数据块字节数**B：高速缓存中每一行内数据块的字节数。每个具体的字节对应m中的b位块偏移量
- 高速缓存的容量：$S * E * B$（单位是字节）
- 直接映射高速缓存：$E = 1$
  - 标记位和索引唯一标识了内存中的每个块。如`0 00 0`和`0 00 1`这两个地址都位于块`0 00`
  - 2个块映射到同一个缓存组，比如`0 00 1`和`1 00 0`这两个地址分别位于块0：`0 00`和块4：`1 00`但是都映射到高速缓存组`00`
    - 映射到同一个缓存组的块使用标志位标识。例如，当高速缓存组`00`中的块是`1 00`时，地址`1 00 1`命中，但`0 00 1`不命中
  - 地址偏移量代表字节。
    - 比如，当缓存组`00`中存放块`1 00`，`1 00 0`表示第1个字节，`1 00 1`表示第二个字节
- 组相连高速缓存：$e \geq 1, E \geq 2$
  - 多个可映射到同一个缓存组，并有E个块能同时放置在同一个缓存组中
    - 比如e=1，E=2，t=2，假如`00`缓存组中现有`00 00`块，则`10 00`块也能放置在缓存组`00`中
    - 当需要进行替换，比如`11 00`块到达，需要映射进缓存组`00`时（此时`00`中已经有2个块），使用LFU或LRU替换5.

5.5 高速缓存的使用

5.5.1 存储器山（记住样子就行）

5.5.2 优化循环

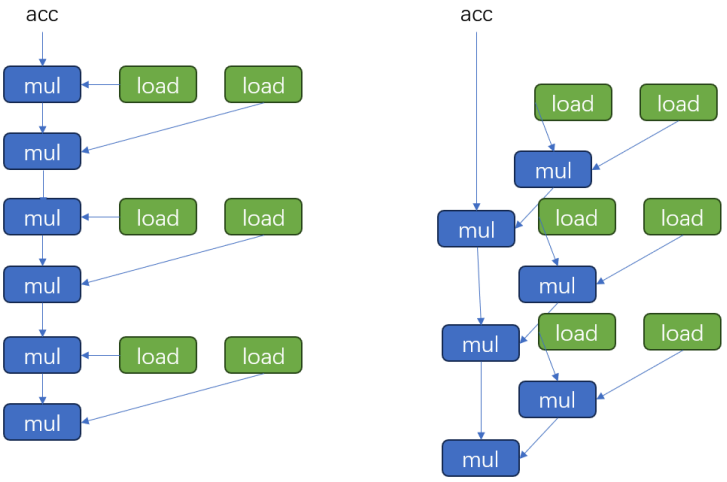
- 访问2维数组时每次更换后面的下标

5.5.3 分块优化

- 处理大型数据块（例如矩阵乘法）时分块处理（例如划分为更小的矩阵）往往具有更高的效率

6 程序性能优化

- 编译器优化方法
  - 代码移动：识别要执行多次（例如在循环内）但计算结果不会改变的运算，将运算外提至循环外
  - 替代运算：将花费较高的运算替换成花费较低的运算（如将乘法替换成位运算）
  - 减少过程调用：如使用内联函数
  - 减少存储器调用：如使用临时变量
- 简单**循环展开**
  - 设循环展开因子为k，循环总长度为n，则最大循环索引为$n-k+1$
    - 未达到最大循环索引时，每次处理k个元素；每次迭代后循环索引i+=2
    - 达到最大循环索引时，每次处理1个元素，直到处理完全部的索引
  - 循环展开无法消除关键路径上的计算
  - 循环展开的优势在于减少了**循环开销**
    - **降低循环迭代次数**：通过循环展开，循环的迭代次数从n减少到n/2。这样，相关的循环控制指令（如比较、跳转、计数器更新等）也相应减少了一半，降低了整体的循环开销。
    - **减少分支跳转**：较少的循环次数意味着分支跳转的次数减少，减轻了处理器的分支预测压力，提高了分支预测的准确性，减少因预测错误带来的性能损失。
  - 循环展开的优势在于提高**指令并行性**
    - **增加指令并行度**：通过同时处理多个操作，处理器可以更好地利用其超标量能力，同时执行多个指令，提高执行效率。
  - 循环展开的优势在于充分利用**缓存命中率**
    - **优化内存访问模式**：通过设计循环展开因子，可以使循坏体内部的数据和指令获得较高的缓存命中率
  - 循环展开可以减少分支预测错误的概率并充分利用现代处理器的特性
- 简单的**提高并行性**
  - 多个累计变量的并行计算
    - 例：无依赖累加的两路并行

- ```
  int data[n] = {...};
  long limit = n-1;
  long i;
  for (i = 0; i < limit; i+=2){
      acc0 += data[i];
      acc1 += data[i+1];
  }
  for(; i< n; i++){
      acc0 += data[i];
  }
  dest = acc0+acc1;
  ```
    - 并行可以减少关键路径上的计算
    - 对于**延迟**为L，**容量**为C的操作而言，当$k \geq CL$时可以达到最大吞吐量
- 重新结合变换
  - 重新结合变换的关键在于消除运算之间的迭代关系以实现并行的效果
  - 例：重新结合乘法
  - 

  - 通过重新结合乘法运算，顺序执行的乘法由6次降为3次
  ```