**Ch 1**

**八个思想**1. Moore Law(18个月翻一翻) 2. Abstraction to simplify抽象分层 3. Common thing fast 4. Parallelism 5. Pipeline 6. Prediction 7. Mem分层 8. redundancy

**Amdahl law**  %的程序提速k倍，加速,

**Response time(exec)** 一项任务的总时间(含I/O…)

**Throughout(bandwidth)** 单位时间内完成任务数

时钟周期clock cyele time(CC) = 1/clock rate

**clock rate=clock cycles/CPU time** 主频↑一项任务耗时↓(rate与CPI无关)

**CPI** cycle per instr执行一条指令需要的时钟周期数 CPU time = IC\*CPI\*CC IC由编译器决定，CPI and Cycle time由CPU决定

CPI= clock cycle/IC speed up= /



compiler的作用是把高级程序语言转化为汇编

Linker -> Executable Program；Static Linking将所有的库文件都链接到一个可执行文件中；Dynamic Linking在运行时再链接使用到的库文件函数

**Ch 6**

**可靠性Reliability MTTF** mean time to fail平均无故障时间 **MMTR**平均修复时间**MTBF**between fail=MTTF+MTTR平均故障间隔时间

可用性 **Availability=MTTF/(MTTF+MTTR)**

数据传输 **pooling**周期性检查是否到了下一次IO操作 **Interupt**完成操作通知CPU **DMA** (direct mem access)不经过CPU

**RAID0**：没有冗余的条带化，提升n倍性能（n是容器个数），1 bit坏了就是坏了

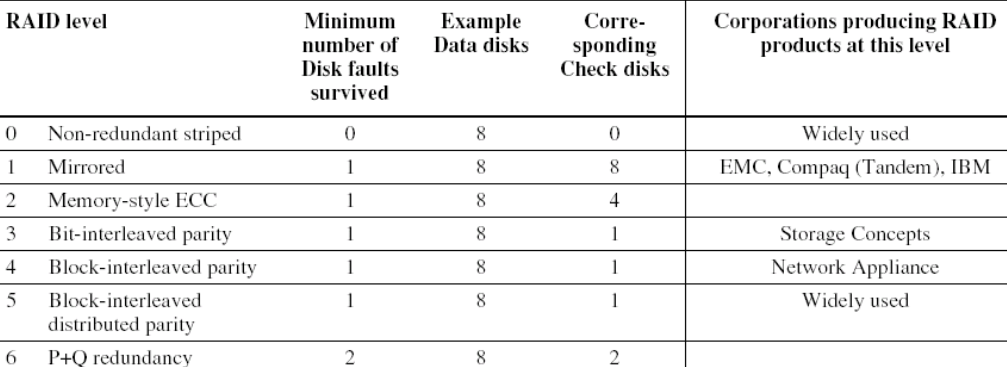
**RAID1**：有镜像磁盘，保证安全性，提升2倍性能.可靠性比2，3，4都好。坏2个可能就不行，但最多可以坏所有的镜像磁盘

**RAID2**：有 Memory-style 的纠错码(ECC校验，汉明码)，只能允许单磁盘错误，并且校验码计算特别麻烦

**RAID3**：bit-interleaved parity 奇偶校验。只用一个盘，它的每个行的数值来存之前每行的序列的就校验码。比RAID2快并且占位小，但也只允许一个disk错。最多坏1块

**RAID4**：按块进行条带化。前几个RAID都是按照bit来条带化的。RAID4B比3方便文件读写。最多坏1块

**RAID5**：RAID4基础上把奇偶校验均匀分布在每个磁盘上。最多坏1块

**RAID6**: P+Q 冗余，纠错码。overhead大于RAID5。允许坏2块，坏3块就异常

**Synchronous bus**：使用时钟和固定协议，速度快、体积小，但每个设备必须以相同的速率运行，时钟偏差要求总线较短

**异步总线**：不使用时钟，而是使用握手

**Handshaking protocol**：用于协调异步总线传输的一系列步骤。

**Polling**：处理器定期检查状态位，以查看是否到了执行下一个 I/O 操作的时间

**Interrupt**：当 IO 想要通知处理器它已完成某些操作或需要注意时，它会导致处理器中断. 当中断驱动的 I/O 设备实际上没有传输时，不需要 CPU 时间, 这是相比polling的优势

**DMA**：设备控制器无需处理器参与，即可直接将数据传输到内存或从内存传输数据

**磁盘**Nonvolatile **Access Time** = Seek time + Rotational Latency + Transfer time + Controller Time Rotational latency: wait for desired sector = **30/v**. v=x

组成：platter，track，sector（最小单位）

每个磁道的容量 = 每个扇区的大小 × 每个磁道上的扇区数（考过）xxRPM

Polling: 轮询操作的时钟周期数为 400，处理器以 500 Mhz 时钟执行。假设您轮询的频率足够高，不会丢失任何数据，并且这些设备可能始终处于繁忙状态。确定鼠标、软盘和硬盘所消耗的 CPU 时间比例。

每秒轮询鼠标 30 次，软盘以 16 位为单位将数据传输到处理器，数据速率为 50 KB/秒。硬盘以四字块为单位传输数据，传输速度为 4 MB/秒

**mouse**: clock cycles per second for polling :

30×400=12,000 cycles

Fraction is 12\*10^3/(500\*10^6)=0.002%

**floppy disk**: the number of polling access per second: 50KB/2B = 25K

clock cycles per second for polling=

25K ×400cycles

Fraction: 10\*10^7/(500\*10^6)=2%

**hard disk**: The number of polling access per second : 4MB/16B = 250K

Clock cycles per second for polling = 250K×400

Fraction: 100\*10^6/(500\*10^6)=20%

Interrupt硬盘和处理器同前，但我们使用的是中断驱动的 I/O。每次传输的开销为 500 个时钟周期

**假设磁盘100%** 的时间都在传输数据。因此，中断率与轮询率相同。

磁盘的每秒周期数为：250K×500=125×106 周期/秒, 传输期间消耗的处理器比例为：

125× 10^6/(500× 10^6)=25%

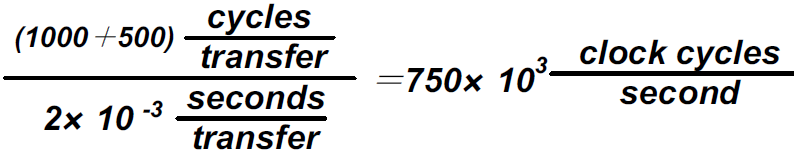
**假设磁盘仅在5%** 的时间内传输数据。平均消耗的处理器时间比例为：25%×5%=1.25%

DMA 传输的初始设置需要处理器的 1000 个时钟周期，并假设 DMA 完成时中断的处理需要CPU的 500 个时钟周期。硬盘的传输速率为 4MB/秒并使用 DMA。磁盘的平均传输量为 8 KB。假设磁盘 100% 都在主动传输

Time for each 8KB transfer is:

8KB/(4MB/second)=2\*10^-3seconds.

It requires the following cycles per second:

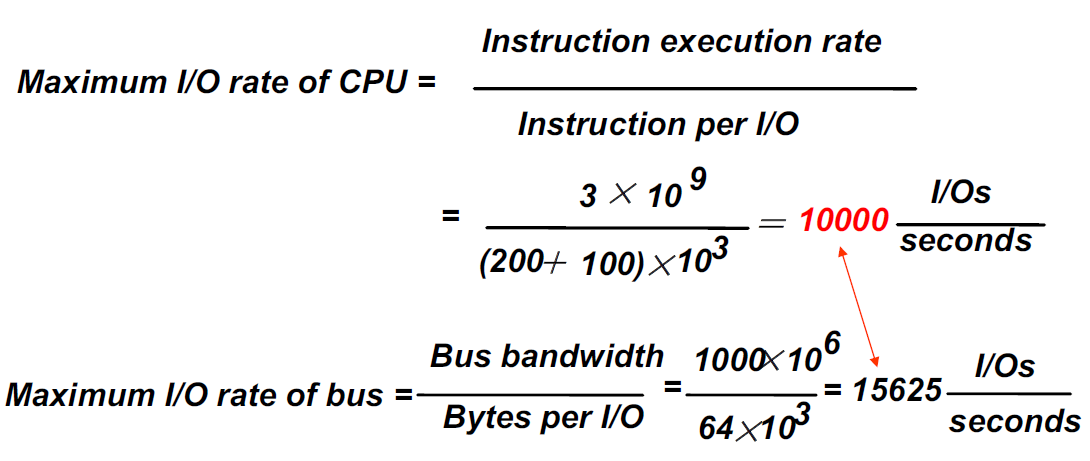


Frac of CPU time=750\*10^3/(500\*10^6)=0.2%

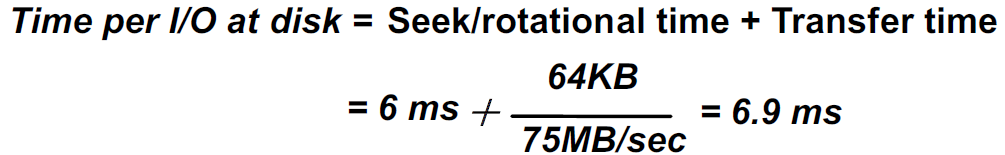
**设计 I/O 系统的一般方法 1.**找到 I/O 系统中最薄弱的环节，即 I/O 路径中会限制设计的组件。工作负载和配置限制都可能决定最薄弱的环节位于何处 2.配置此组件以维持所需的带宽 3.确定系统其余部分的要求并配置它们以支持此带宽

CPU 每秒可执行30亿条指令，操作系统中每次 I/O 平均执行 100,000 条指令。内存背板总线能够维持 1000 MB/秒的传输速率。SCSI-Ultra320 控制器的传输速率为 320 MB/秒，最多可容纳 7 个磁盘。磁盘驱动器的读/写带宽为 75 MB/秒，平均寻道加旋转延迟为 6 毫秒

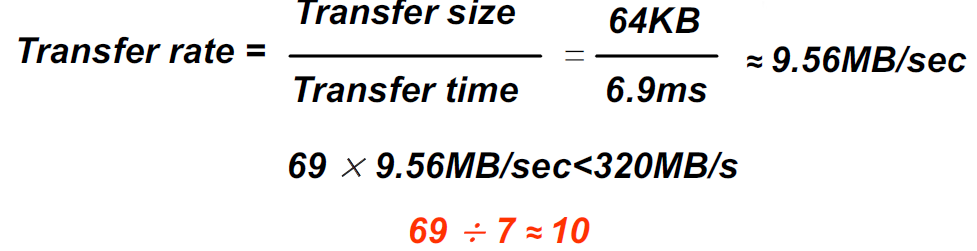
如果工作负载由 64 KB 读取组成（假设数据块在轨道上是连续的），并且用户程序每个 I/O 操作需要 200,000 条指令，请找出最大可持续 I/O 速率以及所需的磁盘和 SCSI 控制器数量。系统的两个固定组件是内存总线和 CPU。让我们首先找到这两个组件可以维持的 I/O 速率，并确定其中哪一个是瓶颈(CPU)



我们首先要确定磁盘上每个 I/O 操作的时间：



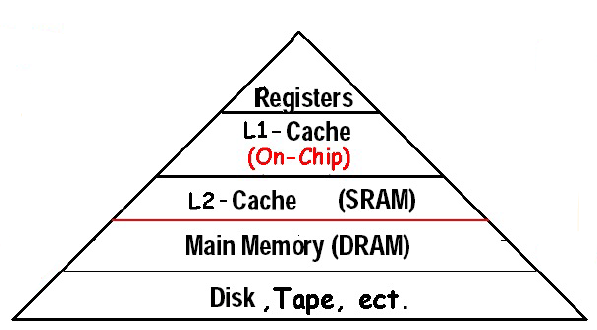
这意味着每个磁盘可以完成 1000ms/6.9ms，或每秒 146 次 I/O。要使总线饱和，系统需要 10000/146≈69 个磁盘. 要计算 SCSI 总线的数量，需要知道每个磁盘的平均传输速率：



假设磁盘访问不是集群的，这样我们就可以使用所有总线带宽，我们可以在每个总线和控制器上放置 7 个磁盘。这意味着我们需要 69/7，即 10 个 SCSI 总线和控制器。

**Ch 5**

二级cache是SRAM，主存是DRAM，二级存储是Disk。**SRAM**晶体管、面积大、一般用于cache，**DRAM**要经常刷新、不刷新数据就没来，慢。**一个 word 是 4 Byte**



**direct**(1-way,pros:简单 cons:替换操作频繁, 命中率比较低.cache利用率低), **fully**(满way，只有tag没有index;pros: 命中率比较高， cache 存储空间利用率高cons: 速度低，成本高，访问相关存储器时，每次都要与全部内容比较), **n-way**(1个set有n个block)， way也称为associative

cache size = item数量\*item大小





block大小为**1 word**则byte offset为2

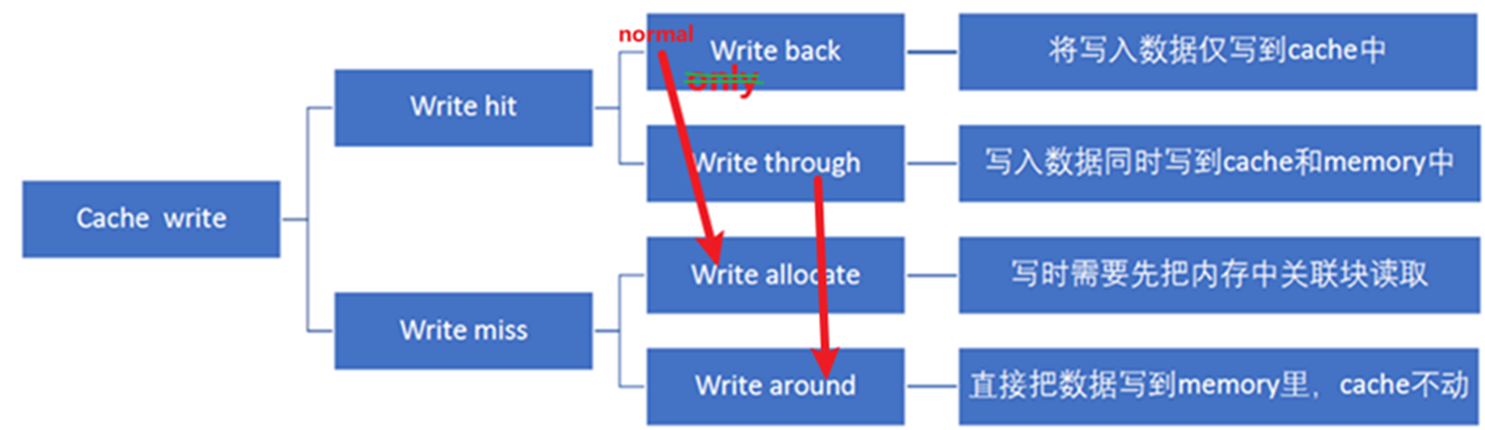
**Write back**好处是bandwidth小(data often overwritten multiple times)

**Write Through** 一般有dirty bit，好处是simple，consistent +Write buffer用于防止write的时候stall，但是没有真正减少stall次数。虚拟内存一般用的write back

**compulsory miss**必定发生的不命中: 初始化缓存的时候cache是空的。增大block size可以缓解这个问题

**capacity miss** 缓存容量有限导致miss

**conflict miss** n-way中的set满了而放不下其他相关数据

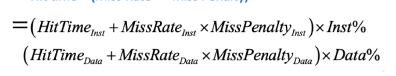


**Read miss**会从 memory 里把对应的 block 拿到 cache，然后读取对应的内容

替换策略: RAND, FIFO LRU最近最少使用

Average miss rate = Inst% \* MRinst. + Data% \* MRdata

Average Memory Access Time(AMAT) = Hit time + (Miss Rate × Miss Penalty)





对策： 1.**降低miss penalty**: multilevel caches, critical word first, read miss before write miss, merging write buffers, and victim caches。最高效的应该是降低block size

2. **降低miss rate**： larger block size, large cache size, higher associativity, way prediction and pseudo-associativity, and compiler optimizations, n way的n更大(但是会造成hit time增加)

3. **降低hit time**：small and simple caches, avoiding address translation, pipelined cache access, and trace caches

**block size↑**,空间局部性增大，miss的概率减少。块数量变少引发pollution，miss变多了。总的来说compulsroy miss↓, miss penalty↑

**associativity↑** conflity miss↓ access time↑

cache size↑ capacity miss↓ access time↑

**多级cache**: 希望一级更快，二级减少miss

虚拟内存

虚拟内存的目的是扩容，由虚拟地址得到物理地址，然后再到cache或者主存里面去找 **write back(**传输比寻址快很多，所以整块整块写比写单个字效率高很多) **fully**

假设一个 page 的大小是 4KB，那么其页内的偏移 page offset 就需要 12 位来表示

translation具体方案：**page table** ，存放在 main memory 中，每个进程都有一个自己的 page

table；同时硬件上有一个 page table register 存page table起始地址。根据 virtual page number 找到对应 **page table entry, PTE** 在 page table 中的偏移，然后与page table register 相加得到对应 entry 的 physical address，从中读取对应的 entry。

常用的PTE放在TLB里面（PT的cache）

当 TLB miss 的时候，处理器去 page table 查找对应的项；如果发现对应项是 valid 的，那么就把他拿到 TLB 里（此时被替换掉的 TLB entry 的 dirty bit 如果是 1，也要写回 page table）；否则就会触发一个 **page fault**

cache里面有的主存里面一定会有(cache和TLB有则page table一定有)

**Page size↓**、页表大小↑、cache↓，传输IO开销↑、局部性↓、page fault数量↑、TLB misses↑

**latency**：CPU读/存取数据的时间，很难减小

**bandwidth**：带宽是指在固定时间可传输的资料数量。二级缓存和更大的块可以提高

**interleave**：合理地控制bank充电时序和各bank传输周期就可以保证所有bank的传输周期相连贯，使bank的充电工作交错进行互不干扰。这样最低限度的减少了充电周期对工作效率的影响，从而得到更高的性能

**还有空间就放一道例题(cache和VM各刷一道**

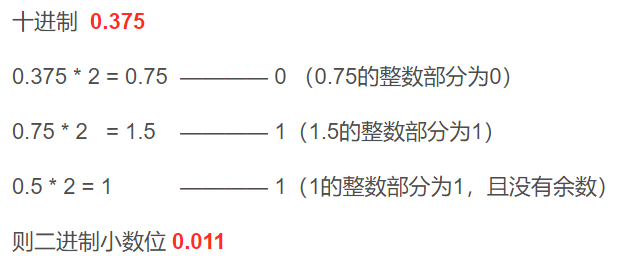
**Ch 3**

原码、反码在数轴上对称，两者都存在+0和-0两个0；补码不对称，0唯一，运算简单，符号位可以和数值位一起运算（加减法可以一起

-1 补码0xBF800000

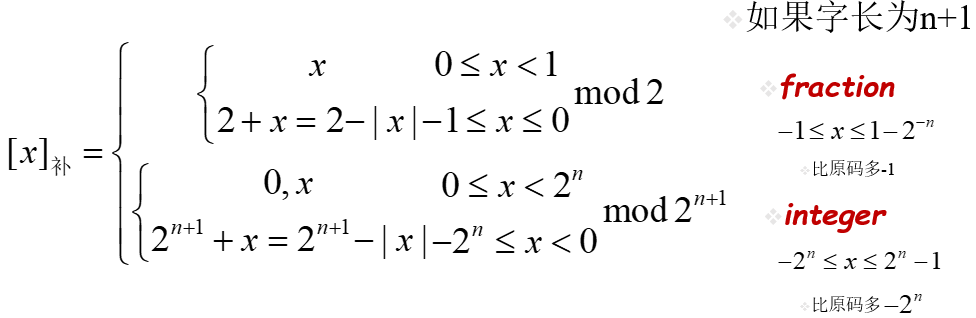
**十进制转二进制**

整数除基取余，小数乘基取整



**真值转换为补码**：对于正数，与原码的方式一样。对于负数，符号位取1，其余各位由真值“各位取反，末位加1”得到

**补码转换为真值**：若符号位为0，与原码的方式一样。若符号位为1，真值的符号为负，数值部分各位由补码“各位取反，末位加1”得到。1’s Complement Code反码和2’s Complement Code补码



例如，若x1=+0.1001, x2=-0.0110，字长为8位，则其补码表示为[x1]=0.1001000，[x2]=2-0.0110=1.1010000

**Overflow**

有符号：用补码进行运算，当前位和进位异或位1，说明溢出

无符号：进位是1，说明溢出

**IEEE754**

Significand里面的1是省略的（规格化）



fraction 的位数越多，浮点数的精度就越高；exponent 的位数越多，浮点数能保存的范围就越大

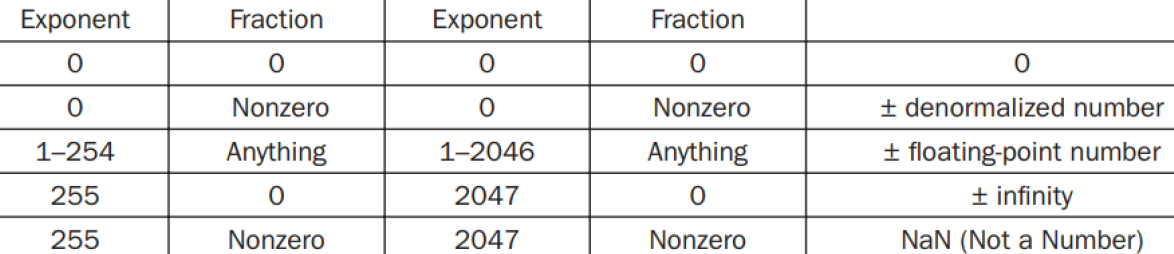
**Single precision** 1bit sign 8 bit exp, 23 bit significand，bias 127

**Double precision** 1bit sign 11 bit exp, 52 bit significand，bias 1023

exponent为00000000和11111111不让用，是特殊情况。如果数字过大不能表示，即 overflow，则

结果置为 inf；如果数字过小不能表示，即 underflow，则结果置为 0

-128补码=10000000



第 1 条表示 0；

第 2 条表示非规格化数，这种数主要是为了用来表示一些很小的数，它的取值为



；但是并非所有机器都支持这种表示，有

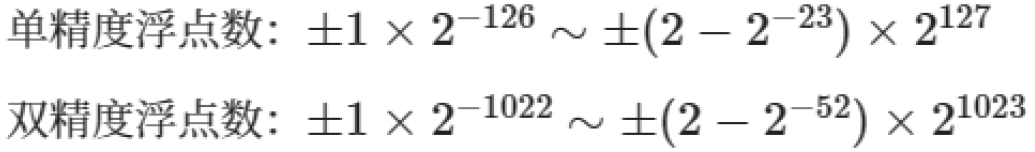
的机器会直接抛出一个 exception。我们不考虑非规格数的存在

第 3 条表示正常的浮点数；

第 4 条表示无穷大或者无穷小，出现在 exponent overflow 或者浮点数运算中非 0 数除以 0 的

情况；

第 5 条表示非数，出现在 0/0, inf / inf, inf - inf, inf \* 0 的情况

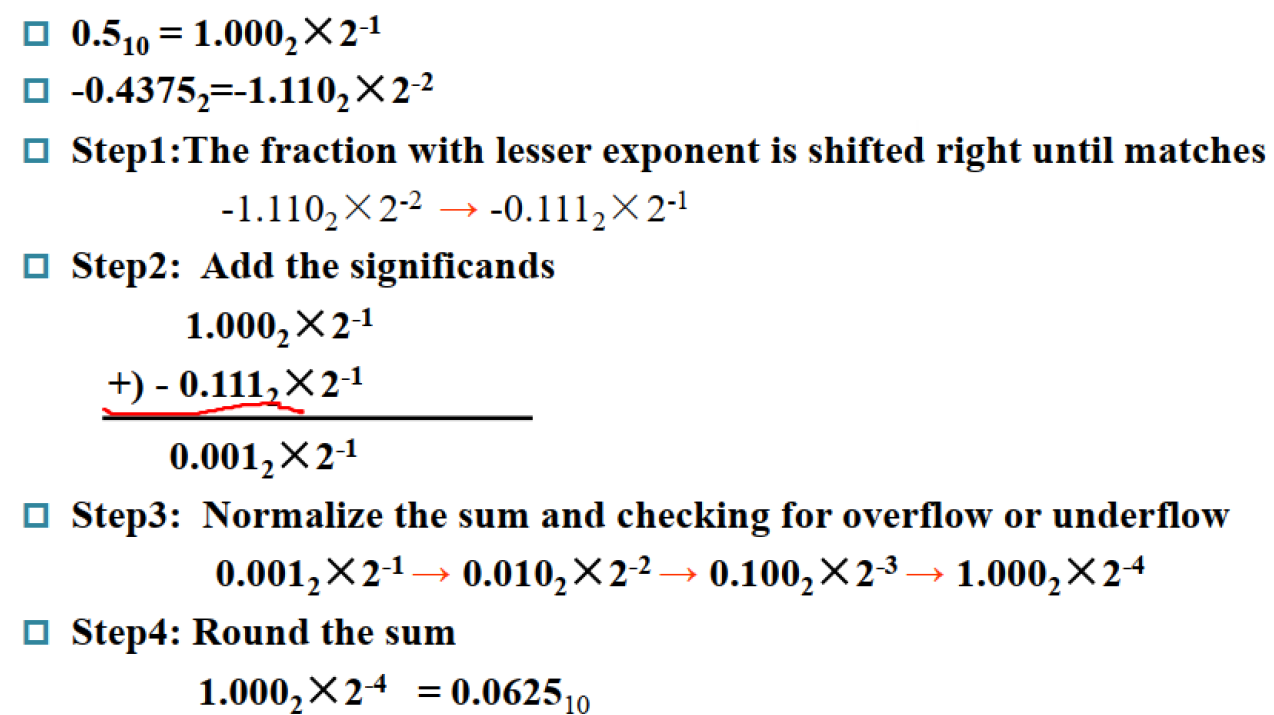


**右规**:尾数右移1位，阶码加1。尾数右移，最高位1被移到小数点前1位作为隐藏位，最后一位移除时，需考虑舍入（右规只要1次）

**左规**：尾数左移1位，阶码减1.左移要多次

右规一次，相当于除2；左规一次，相当于乘2

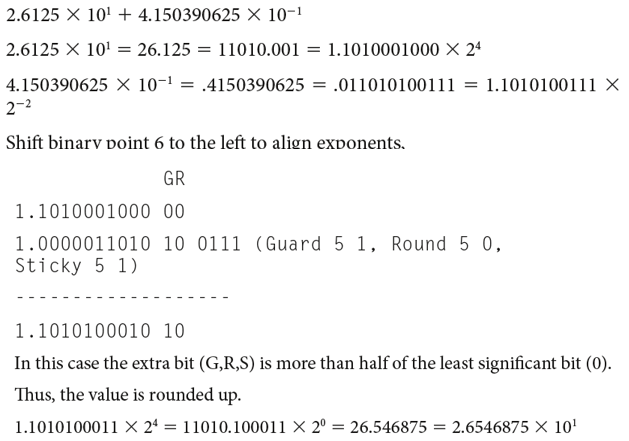
**浮点数加法**



**guard bit**：精确位数右边的一位。先算再四舍五入

**round bit**：精确位数右边的第二位

**sticky bit**：精确位数右边的第三位及以后的位数，只要有一个为1，就置为1



**Ch 2**

**R型**: ①有addi, 没有subi, 因为immediate字段表示的是二进制补码整数，所以addi可以用来做减法。②**逻辑左移=算数左移**，右边统一添0 ；**逻辑右移**，左边统一添0 ；**算数右移**，左边添加的数和符号有关。左移结果出错，右移精度下降 ③RISC-V里面没有NOT指令，用（xor）取代 not：**xori x10,x10,-1** // NOT x10

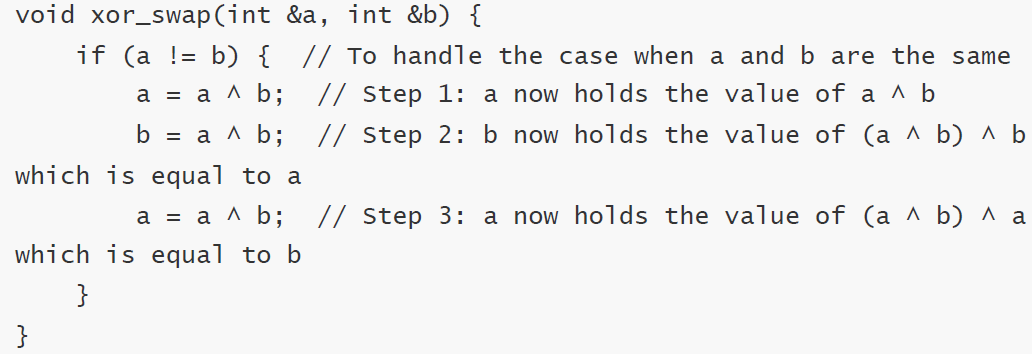
④**0<=x<y**，无符号数比较x<y，在检测x<y的同时，也检测了x是否为负数，比如If (x20>=x11or x20<0) goto IndexOutofBounds

可以写作 bgeu x20, x11, IndexOutofBounds 也可以写作sltu x1, x20, x11 x20 < x11 然后

beq x1, x0, IndexOutofBounds

⑤加载一个大数字： **lui rd, constant**用于将20位常数加载到寄存器的第31位到第12位，低12位填充0，低12位addi或者ori放进去。用**addi**, 把16位立即数按照有符号数拓展，所以最高位是0没有问题，最高位是1就不对了，用**ori** 没有这个问题, 或者先用lui 在第12位添加1，然后再用addi加上低16位。

⑥不用额外寄存器交换两个数

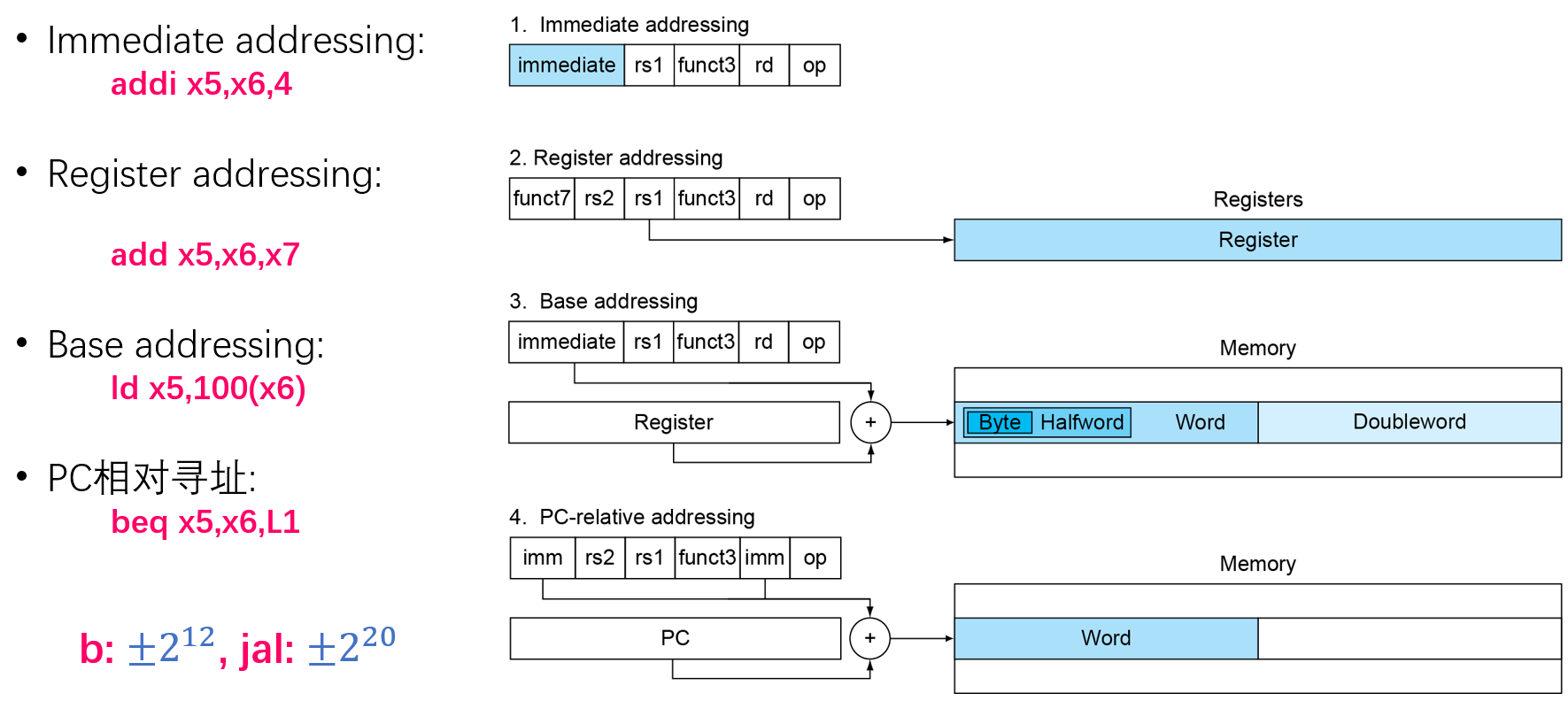


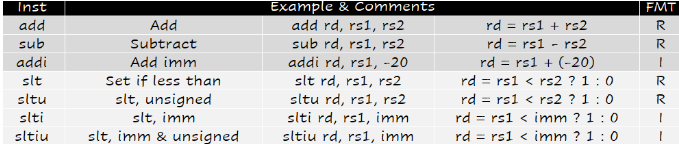
**S型**: Load byte/halfword/word: Sign extend to 64 bits in rd; Load byte/halfword/word unsigned: Zero extend to 64 bits in rd;

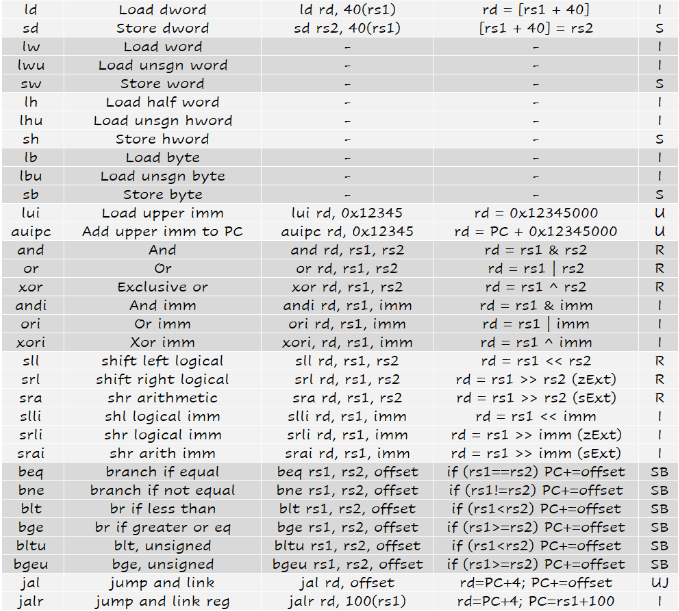
Store byte/halfword/word: Store rightmost 8/16/32 bits

**I+J+UJ型**：jal是无条件跳转,一般是jal x1 L;jalr是PC相对寻址，ld是base，其他是立即数寻址. imm寻址范围为±2^11\*2=2^12. 写成指令的时候就是实际跳的地址，比如jal x0, 2000就是跳2000, 但是imm存的是1000！

因为一条指令宽度不是1，**所以跳转默认乘2**

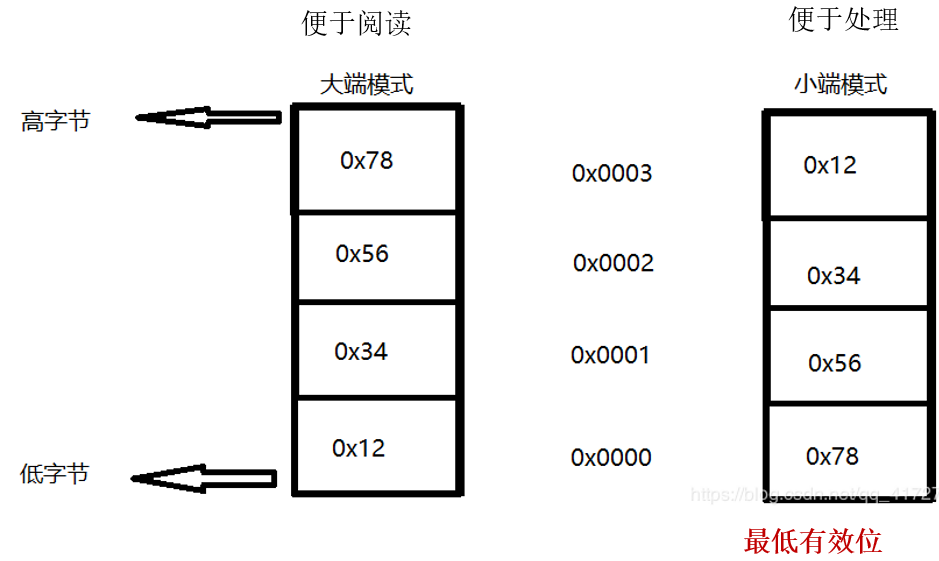






**Big endian**：数据的高字节存放在低地址；Little endian小端模式(主流RISC-V)

E.g. : 32位机器上存放0x12345678，其大小端模式存储如下：



**精度扩展**

M: integer multiply, divide, remainder

A: atomic memory operations

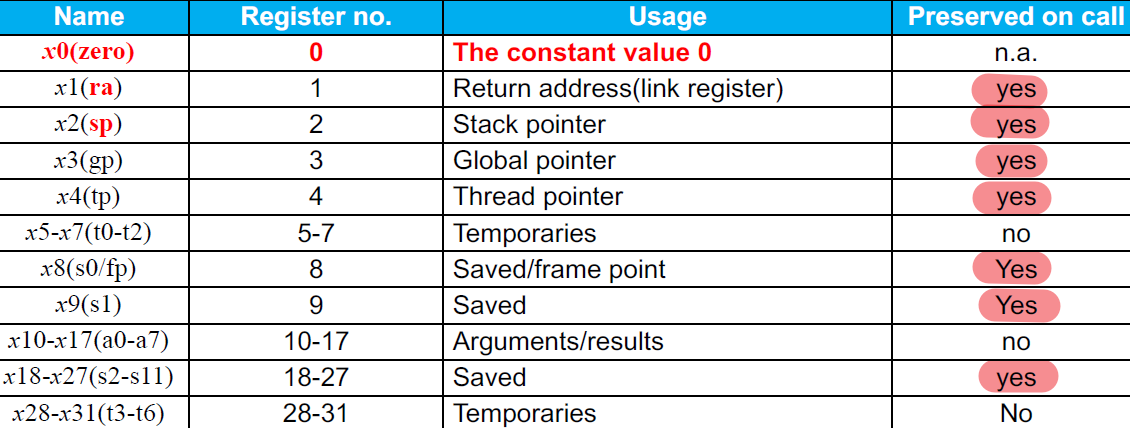
F: single-precision floating point

D: double-precision floating point

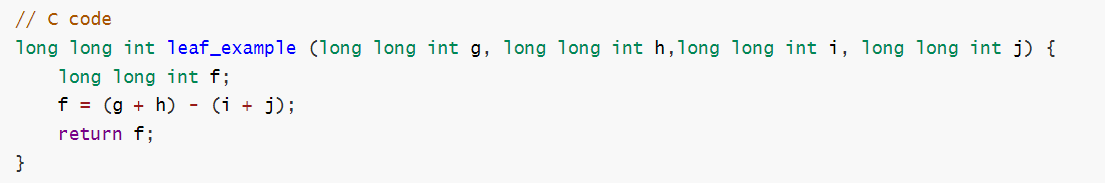
C: compressed instructions(一般float16)

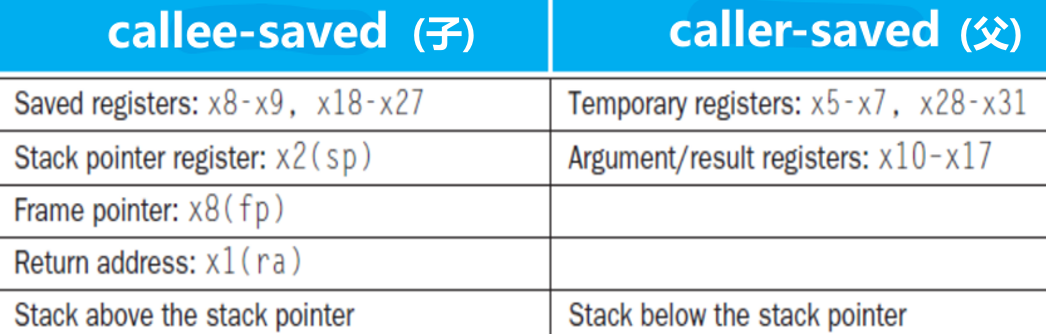
**C转汇编**

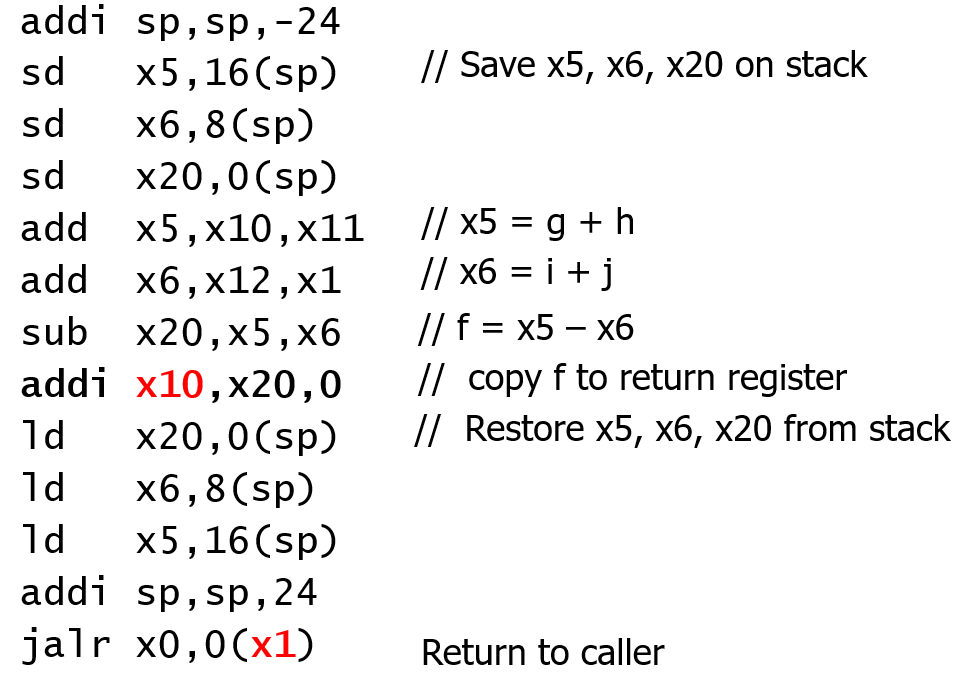
a0 - a7 用于传递参数，**返回值通过 a0 传递**；t0 - t6 为临时变量，可以不用管；s0 - s11 必须保存。x2是**sp. push: sp-8, pop: sp+8**

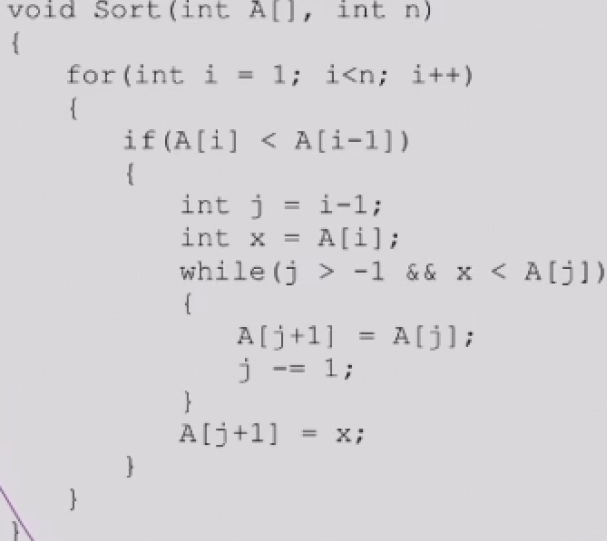


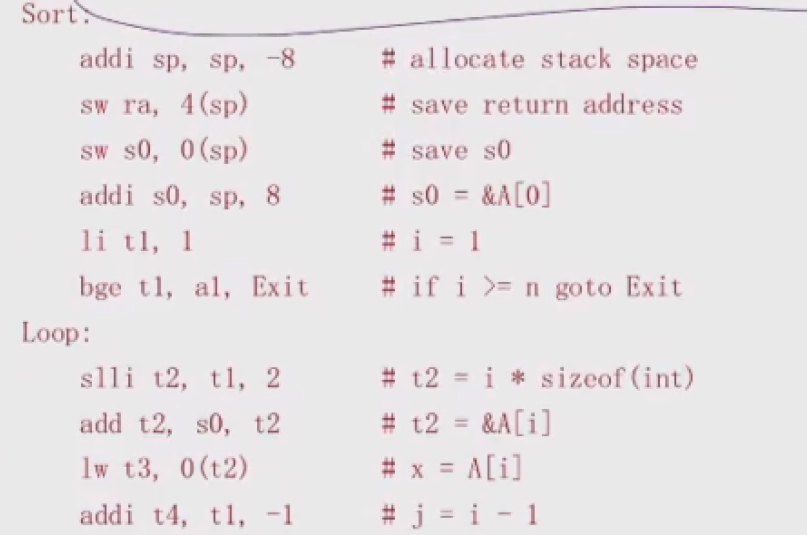
**函数** 要重置下零时寄存器x5,x6,x20的值

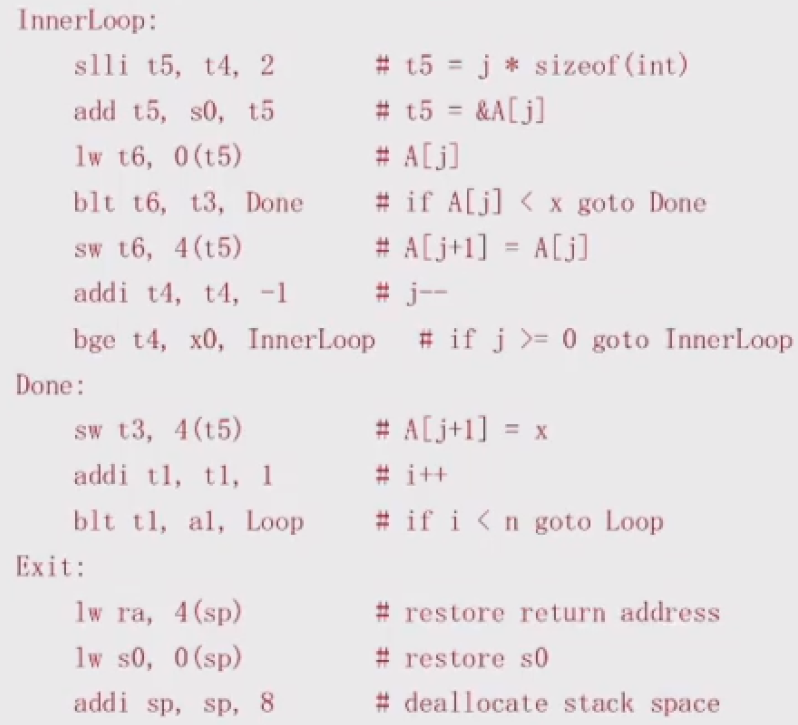
****



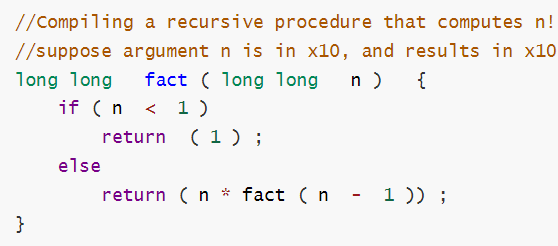




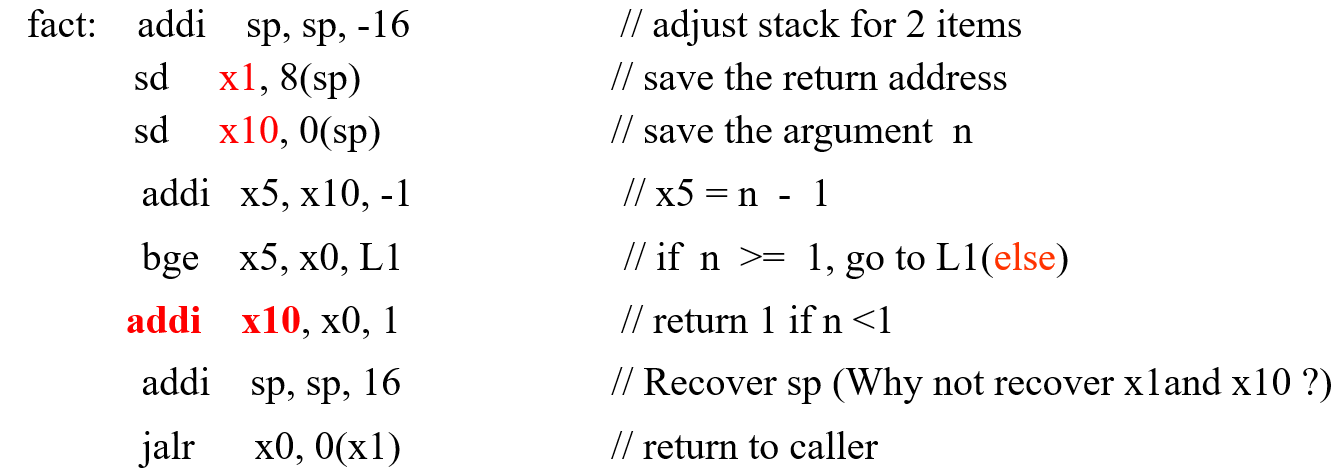


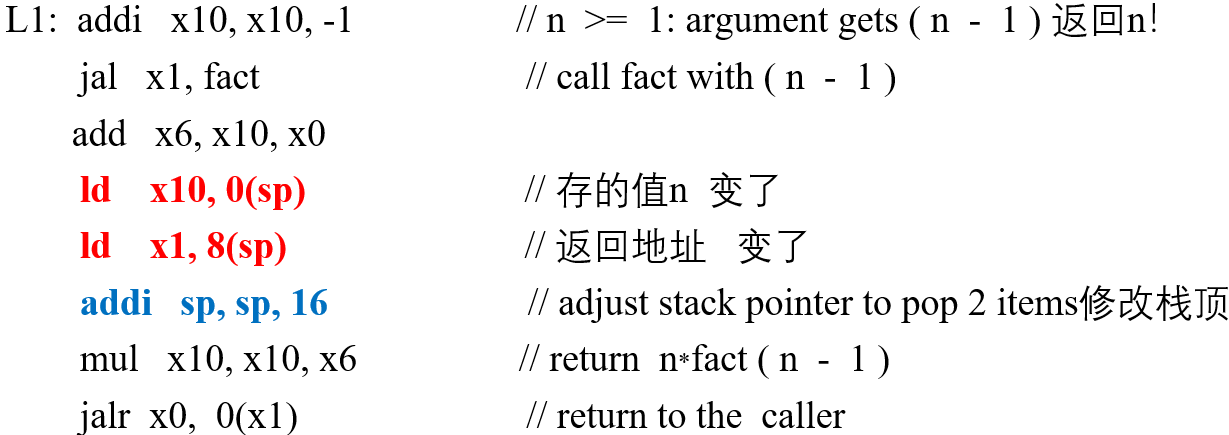


**递归** 无ld x1, ld x10因为没有值没有变，另外递归的效率不如循环。

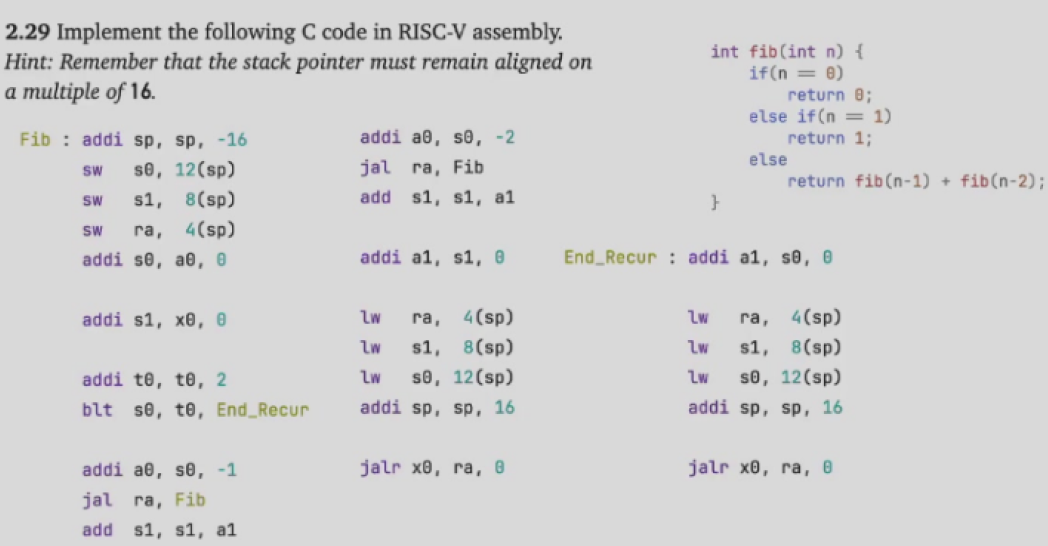


存参数x10和返回地址x1

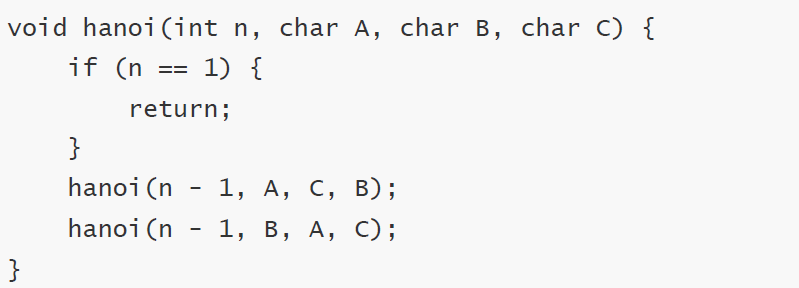


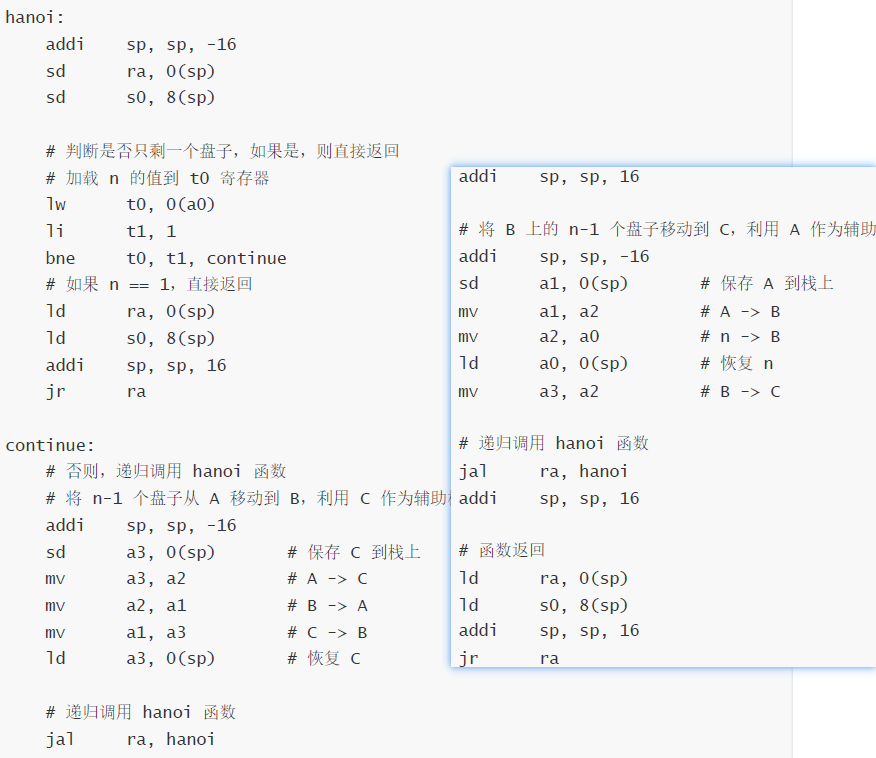


**斐波那契**

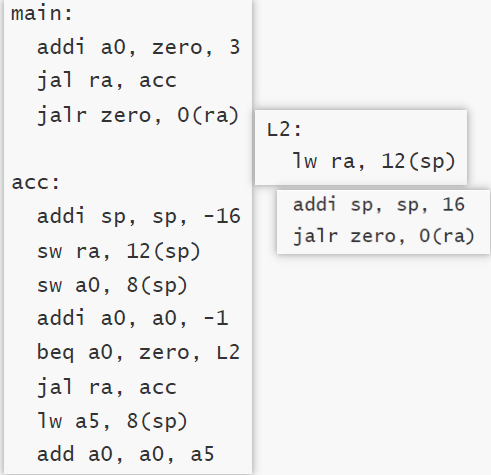
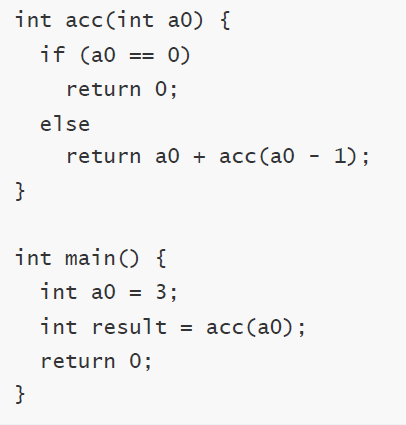


**汉诺塔**





**完整** 从main开始，演示了完整的主程序main调用acc函数，再返回的过程



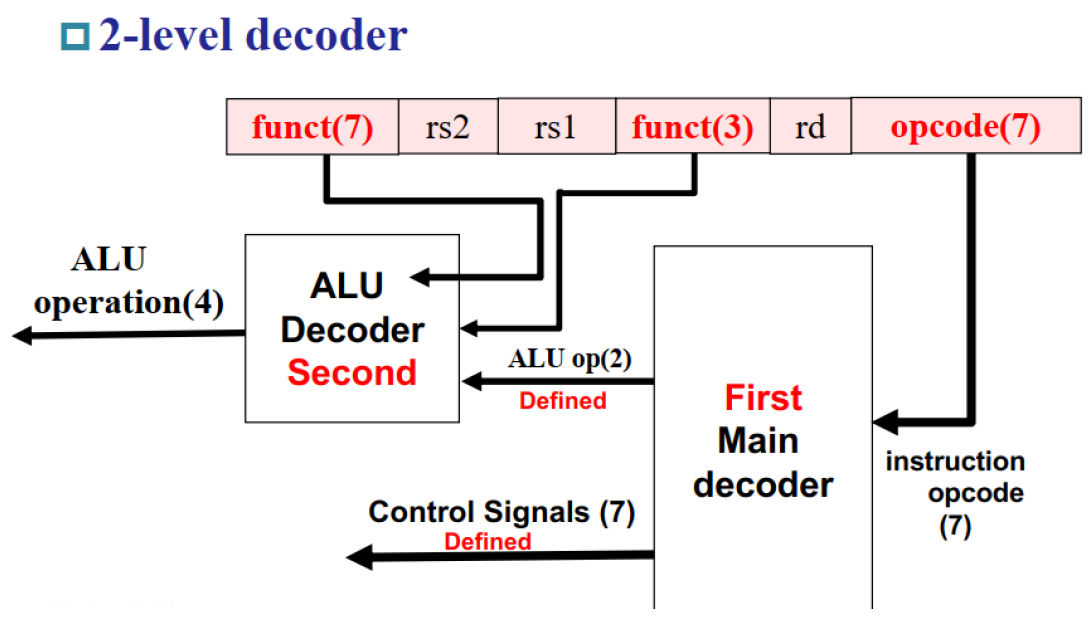
**Ch 4单周期**

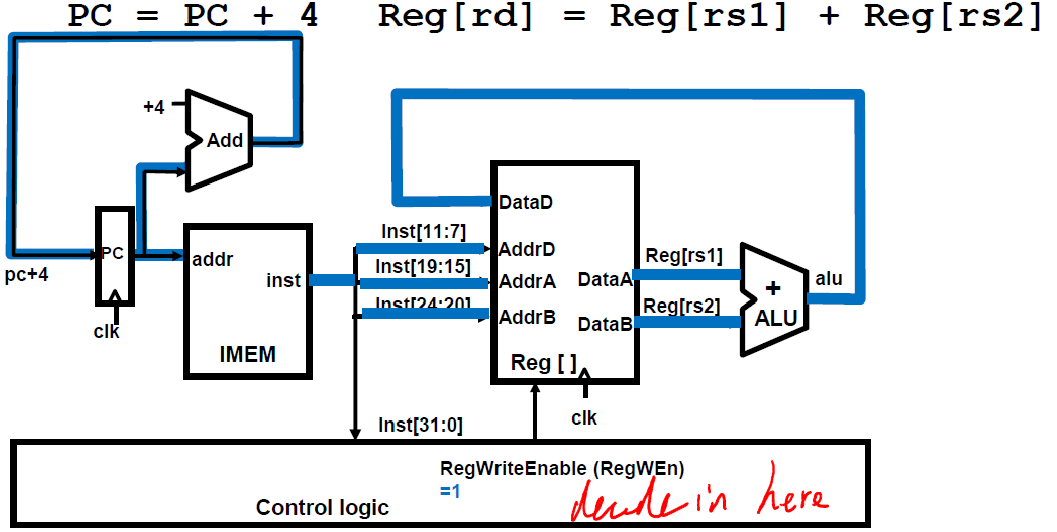
一个指令的所有工作都在一个时钟周期内完成， CPI = 1

64位机器指的是每个word 64位

clocking methodology :时钟周期是时序逻辑时钟加上组合逻辑的传递时间

**只有R型**

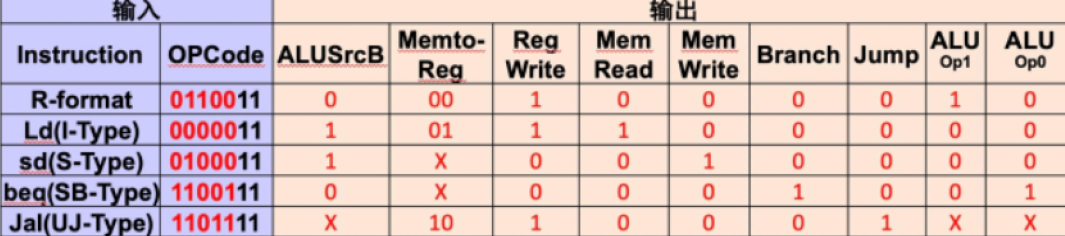




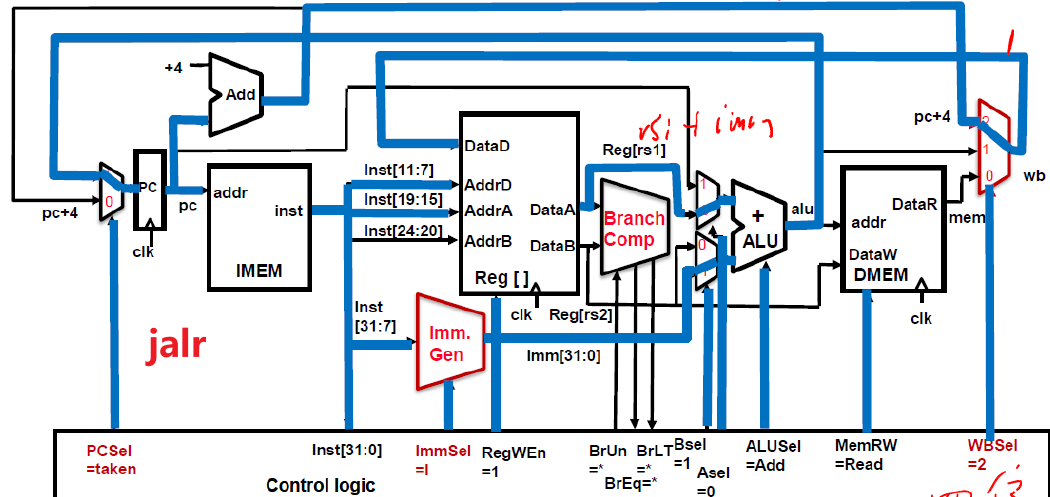
**+I**: ALU (Reg[rs2]) use imm instead; Imm Gen

Load:加信号MemRW来控制MEM

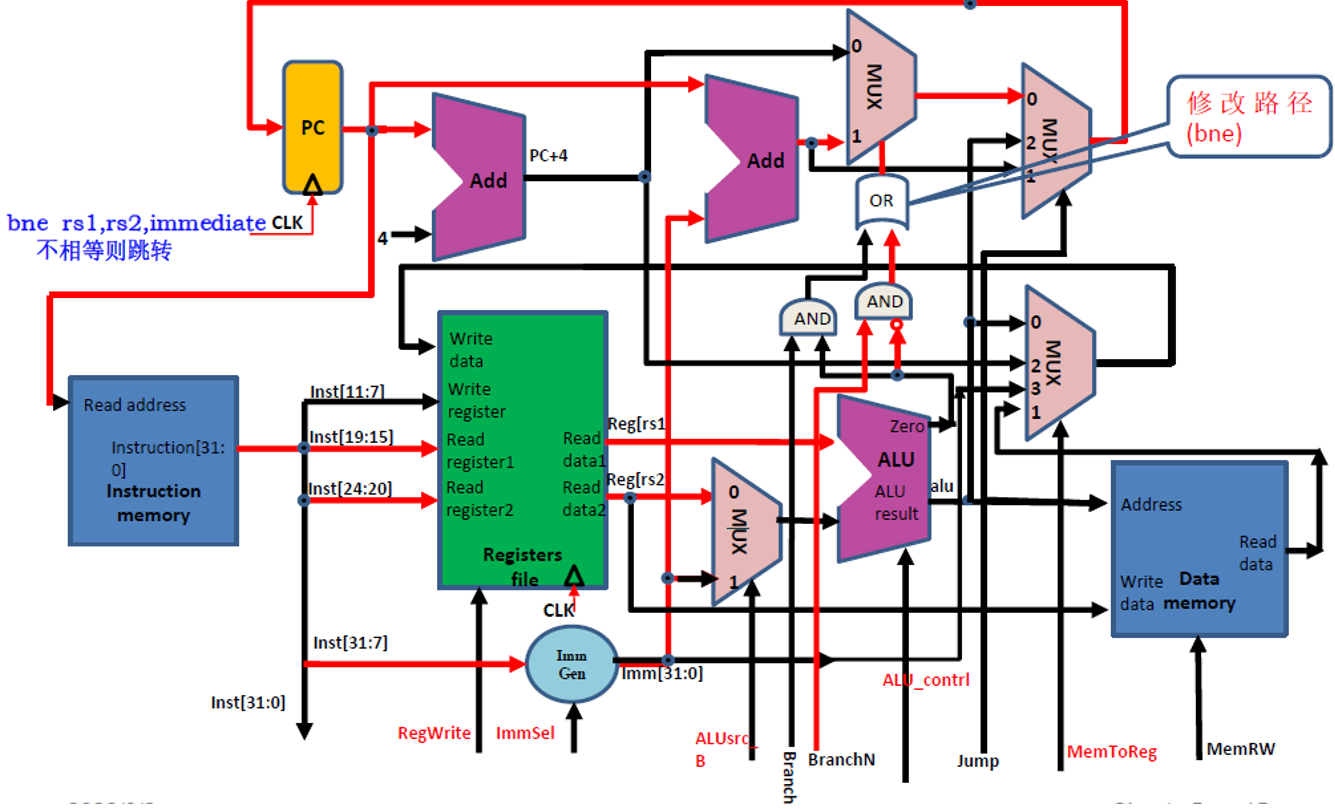
**+J 型** 多一条PC的通路

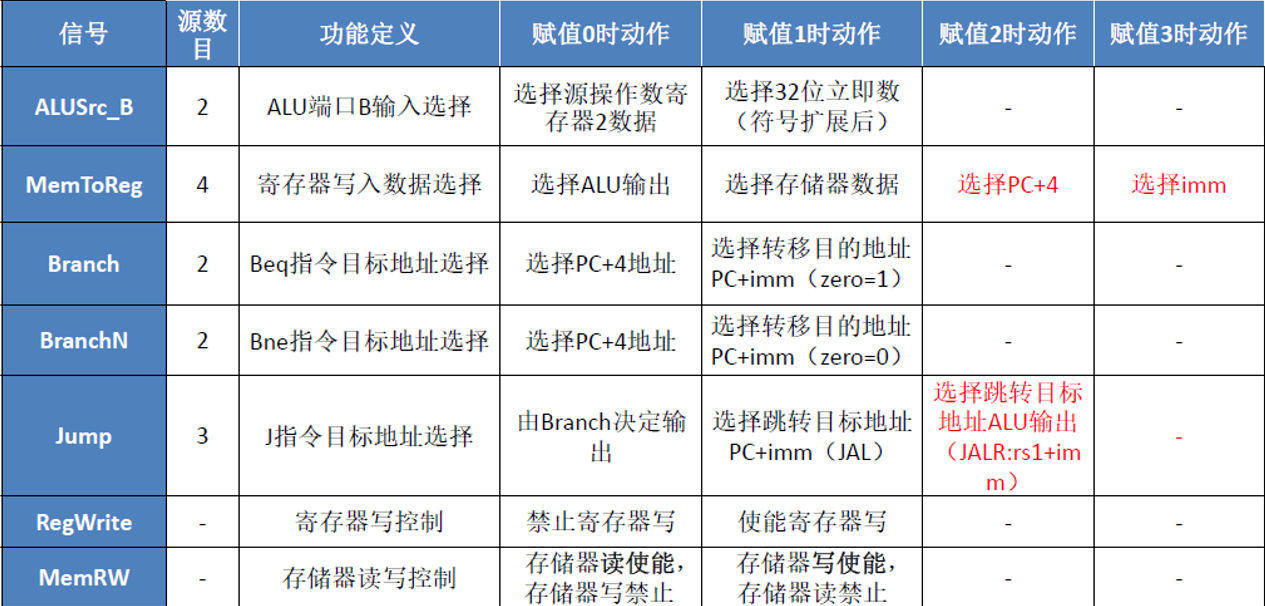


ALUsrc和ALUop在不用的时候不关心，memread不用的时候必须是0（不然读内存可能会read miss），memtoreg不用的时候不关心



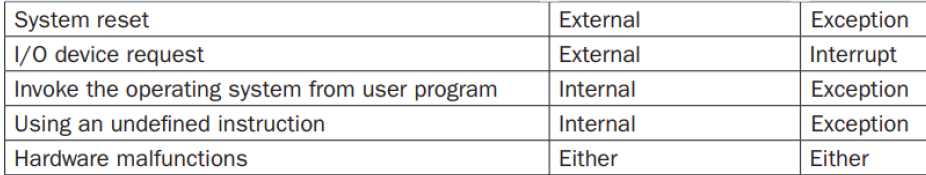
**总体**



Use ALU to calculate①Arithmetic result②Memory address for load/store③Branch comparison

写REG：R-type:ALU results,I-type :memory data

**Exceptions(CPU内部) Interrupts(外部)**



1. 进入处理程序前要把原因放入 cause register（mcause），中断保存当前pc，异常保存pc+4

2. 跳转到 stvec处理

3. 退出异常，mret指令回到mepc

**Ch 4流水线**

流水线通过提高throughput提升效率

PIpeline Speedup: due to increased throughput （CPI和单周期CPI相同 )，所有指令latency相同，没有减少单个指令的延迟

CPU 的时钟周期由耗时最长的指令阶段决定，增加了流水线阶段可以增加时钟速度

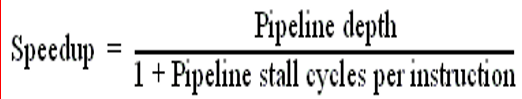
只在 load 或 store 指令中操作 data memory 且不会将存取的结果做进一步运算

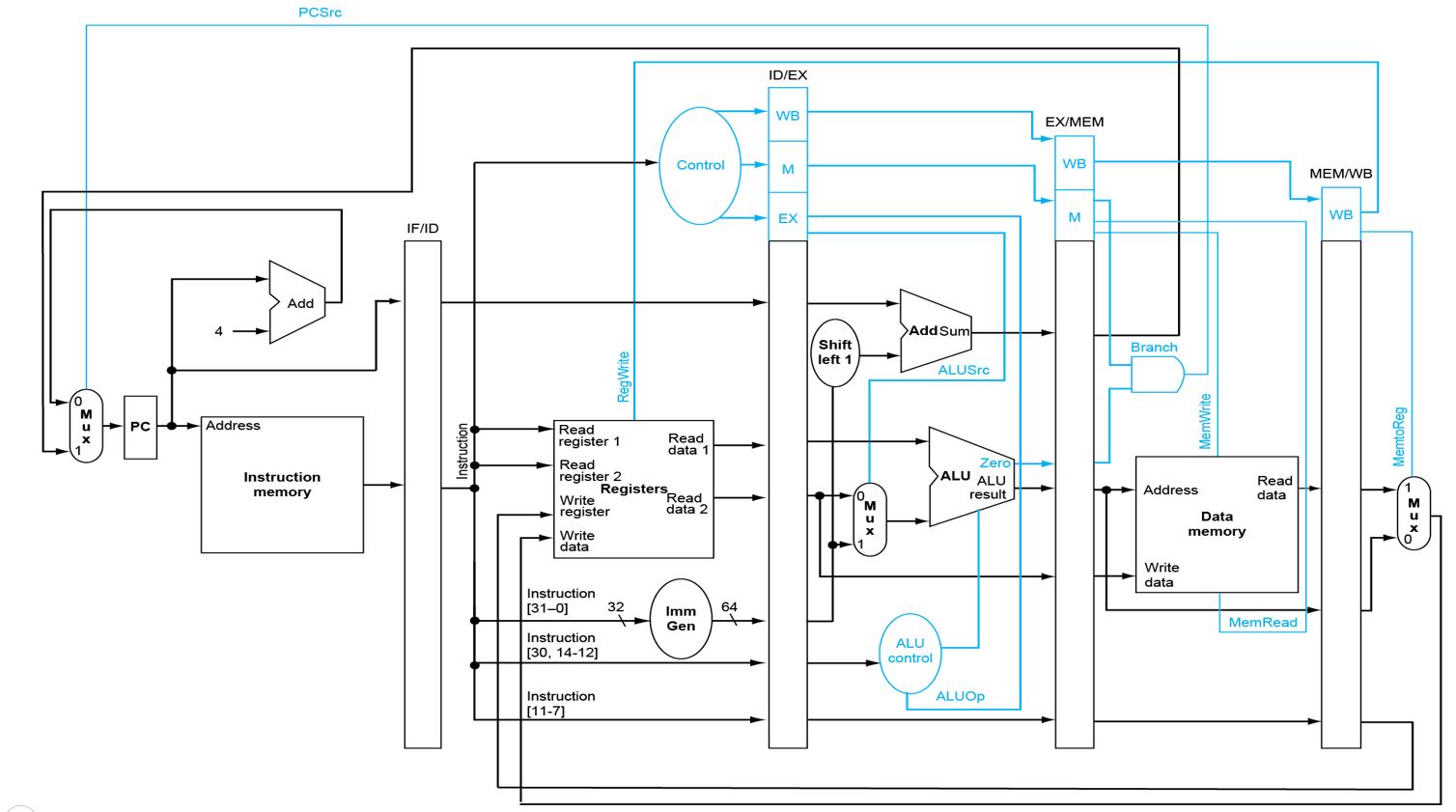
提高资源效率（功能单元）

Ideal speedup is 阶段数, 但不是阶段越多越好，阶段多会引入额外开销（延迟和硬件开销）

单周期 CPI=1 ,clock =5

流水线 CPI=1, clock =1





**Structure hazards**多条指令访问同一个资源比如，Load/store requires data access

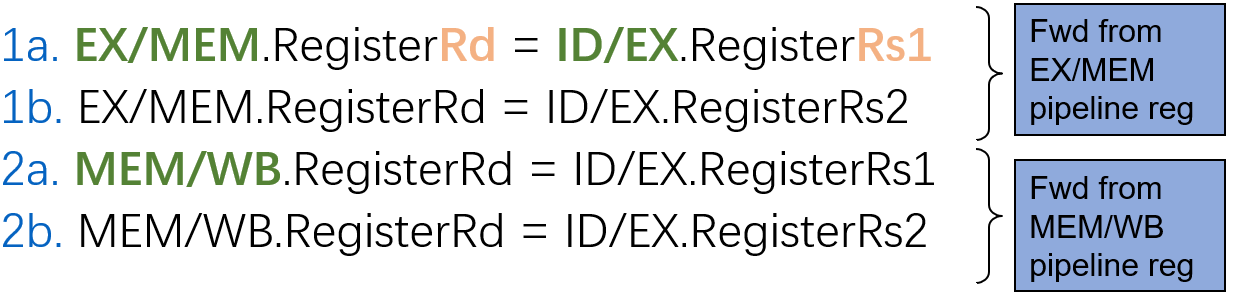
解决：stall 或者多用几个设备, 指令和地址分两个寄存器

**Data hazard**: Double bump(Reg files 和mem的写入均发生在上升沿；而 Pipeline registers 和 PC 的写入均发生在下降沿) 先写后读, Forwarding, Compiler scheduling, Stall

**Control hazard**主要是由一些跳转指令引起的

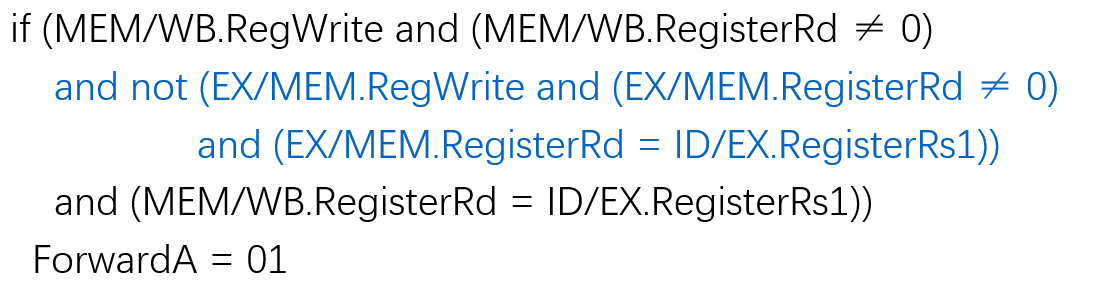
解决：stall, prediction

**Forwarding**



**二次forwad**

add x1,x1,x2  
add x1,x1,x3  
add x1,x1,x4

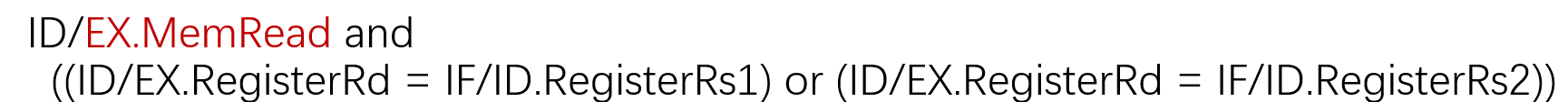


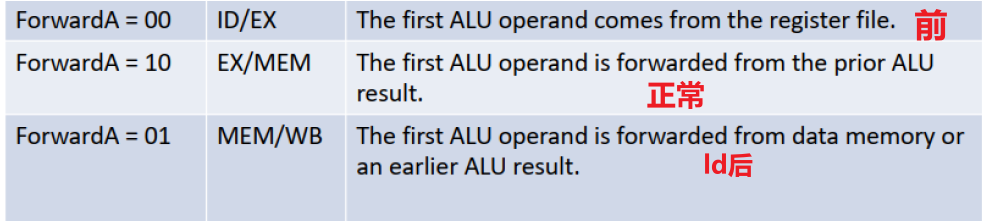
第一行和第二行是说 MEM/WB是写回寄存器的操作，并且这个寄存器不是x0

第三行是EX/MEM的rd不等于ID/EX的rs（因为如果一样就是forward等于10的情况）

最后是ID/EX的rs和MEM/RB的rd一样

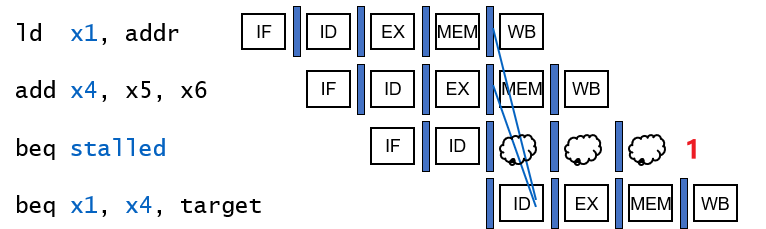
使⽤上⼀条指令**Load**进Memory的结果 ,无法forward，只能插入一个bubble。Load后立刻用

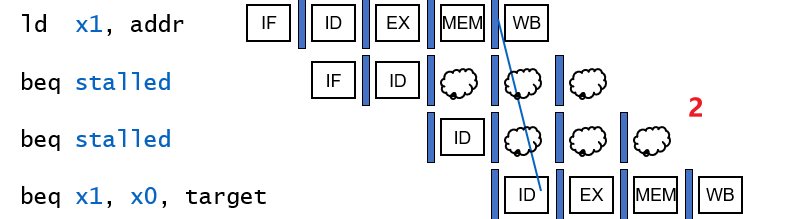




**nop**：当前指令不要产生效果 (清空 RegWrite 和 MemWrite )，让后面的语句不要受到影响 (保留 PC 和 IF/ID 一周期不改)

**动态分⽀预测** :记录上⼀次的结果，下⼀次的branch按照上⼀次的结果进⾏。如果错了再进⾏更改。对于二重循环，使用**2-Bit Predictor** ：连续两次false才预测false





最终的流水线

