计算机迭代：

Pre-computer：非电子，不可编程；

First generation：电子管，可编程；ENIAC

Second：晶体管，编程语言开始应用；体积小，速度快，计算更可靠；IBM 1401，UNIVACIII

Third：集成电路，操作系统开始使用；更小更便宜；

Fourth：大规模集成微处理器；

第一台纯电子数字计算设备：atanasoff-Berry Computer（ABC）

接近现代计算机，但不可编程，非图灵完备；

图灵完备：对于一套操作规则而言，如果这套规则可以模仿通用图灵机，执行任何可实现的计算时，便可以被称为图灵完备；

使用二进制，计算与存储分离；

ENIAC 第一台通用计算机

十进制，可编程，图灵完全，没有程序的存储能力；

冯诺依曼结构：

计算与存储分离；

数据与指令集保存在同一存储器中；

Backward compatibility：向前兼容；

Microprocessors：微处理器；

Integrated circuits：集成电路；

RISC (Reduced Instruction Set Computer)：相比之下有更少CPI（cycles per instruction）；

Pervasive：普遍的，扩大的，渗透的；

Manipulates：控制，操控；

Interface：接口；

Volatile：易失性；nonvolatile：非易失性；

Main memory：如dram结构；

Sram结构：静态随机存储器，也是易失性结构；

Second memory：flash结构；

固态键盘，闪存，硬盘；

Firmware：驱动；

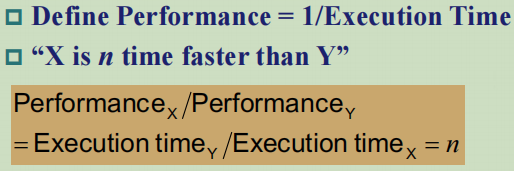
Simultaneously：同时地，并行地；

内存墙（memory wall）问题：处理器和存储器性能提升不成比例；

功耗墙（power wall）问题：单位面积功耗增大，散热问题日趋严重；

工艺发展受限，摩尔定律面临失效；

/\*Throughput（bandwidth）：吞吐率；total work done per unit time；

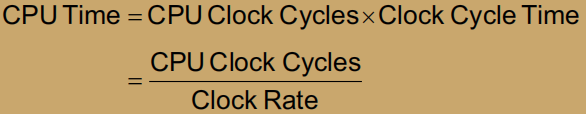
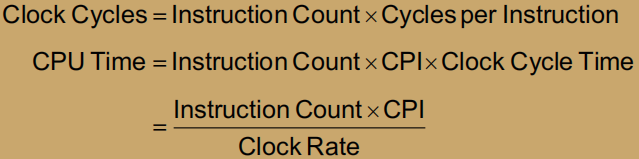


Elapsed time：持续时间；包括I/O，OS操作等时间；

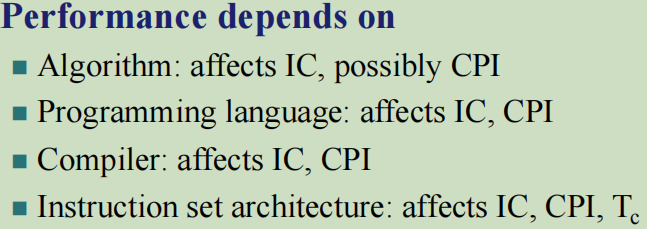
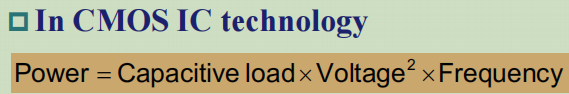
CPU time：只有CPU处理事件的时间；

Clock period: duration of a clock cycle；每一是时钟cycle的对映时间；

Clock frequency (rate): cycles per second：每一秒的时钟周期数；



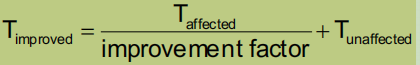
指令总数与程序，ISA及编译器有关；

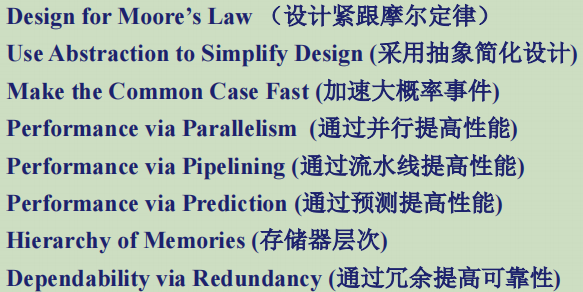


所用功耗 = 电压的平方 \* 频率 \* 电容性负载；

Benchmark：基准/参照；（SPEC CPU Benchmark，用相同的基准比较不同CPU之间的性能差异----计算的是elapsed time虽然I/O可以忽略不计----等效于CPU的表现）；

Amdahl’s Law：只改变CPU的一个方面，很难在全局上做到很大的提升；



八大思想：

\*/

摩尔定律：1.在同样的芯片区域上，可以实现更多的功能；2.使用更少的power；3.相同功能的实现需要更少的空间，带来更低的开销；

Generic：通用的，普通的；

Signed and unsigned numbers；

Signed magnitude：符号位单独占一位，后续按照一般二进制转化即可；（两个零）

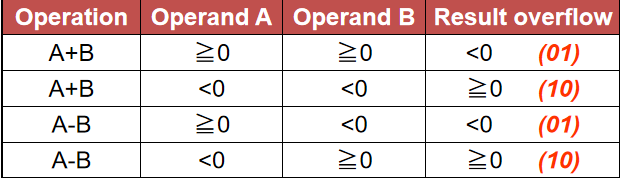
Two’s complement：取反（one’s complement）plus 1；

LSD（Least significant digital）：低位--最右边数值；

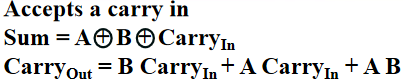
MSD（Most significant digital）：高位--最左边数值；

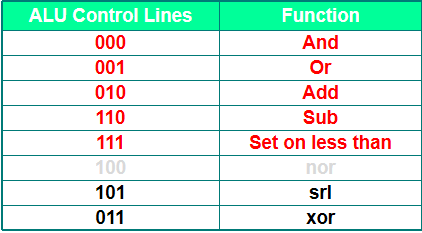
Radix：基数，进制；decimal/binary

二进制计算时，可能出现overflow的现象：

10/01表示最高的两位进位，可以发现，两位进位异或为1时，表示overflow的发生；

/\*全加器公式：

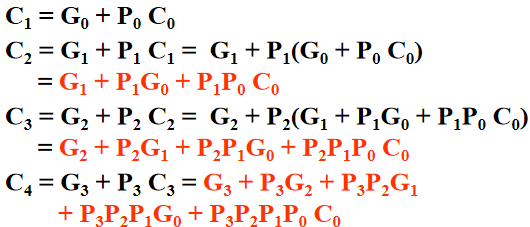
Zero detector：如果每一bit都为0，输出1；若不全为0，输出0；

ALU的control lines与对映的输出;

Ripple carry adder：行波进位加法器，一位位计算加法（简单的将单bit加法器进行整合），虽然所用的级数（level）较少，但有延迟（delay）过高、不断累积的缺点；

Carry look ahead adder（CLA）：所有进位可以以四个一组的方式快速算出，从而降低时耗（虽然电路的层级增加）；

Generate：input都是1，造成的进位是1；

Propagate： input只有一个是1，carry-in是1，造成的进位是1；

\*/

为保证进位的输出在合理的fan-out下，因而一般仅采用四位一group的做法；可以层级的进行嵌套；有C0可以计算所得C1/C2/C3/C4，若考虑层级的概念，则C0可以使用层级概念迅速计算得C4/C8/C12/C16，从而用其分别推导计算剩余的进位情况；

\*\*\*Carry skip adder：进位跳跃加法器；

Carry select adder（CSA）：进位选择加法器；

对于加法的低位，只按照原数计算；高位则均考虑carry为0/1的情况，计算得两个结果；通过最后carry的实际情况选择一个数值进行输出；

二进制乘法的计算--->类似于列竖式；

如A \* B = C；（A为被乘数，B为乘数，C为结果product）

V1：64bit multiplier（乘数）；128bit multiplicand（被乘数）；128bit product（结果）

被乘数实际只有64bit，放置于128的右64位，不断向左移动；乘数64bit防止与64bit区域，不断向右移动（低位不断被截断）；如果对映bit为1，则将被乘数加至result，若为0，则仅进行平移操作；

V2：乘数/被乘数均放置于64bit；product依然为128bit；

移动product（乘法结果），先放置于高64bit，不断向右移动；乘数操作同V1；但是被乘数一直增加至最高的64bit；是否进行加法的规则同V1；

V3：被乘数放置于64bit；product与multiplier共同放置于128bit位置，其中product先放置于高64bit，multiplier放置于低64bit，两者一起向右移，multiplier不断被截断，而product不断增长位数；具体的加法方法同V1；

Booth’s algorithm rule：更多的shift，更少的加法；

用0补齐最右端bit位；从最左端

不断向右移动，注意最高位填补的是符号位，不只是0；

10在左半部减去被乘数；11只进行移位操作；01在左半部加上被乘数；00只进行移位操作；

乘法操作符

Mul：取乘法结果的低64bit（操作数有符号数）；

Mulh：取乘法结果的高64bit（操作说有符号数）；

Mulhu：高64bit作为无符号操作数的乘法结果；

Mulhsu：认为操作数一个为有符号数一个为无符号数；

Division除法：

Divisor除数，dividend被除数；quotient（商），remainder（余数），且余数与被除数同号；！！！

V1：128bit divisor; 128bit remainder; 64bit quotient;

Divisor位于左端64bit，dividend在remainder的最右端；商放置于64bit位置；

使用remainder中低64位减去divisor的128bit，结果放置于remainder中的高64bit；若为负数，则rollback减法；若为整数，则在商的对映位置置1；divisor不断右移，remainder中数值不偏移，商不断左移；通过平移的次数来判定除法的停止条件；remainder中剩下的低位64bit就是余数；

V2：divisor占用64bit；remainder占用128bit并不断向左移动，其中dividend放置于最低64bit；低位补商，每次作差的位置为高64bit；最后高64bit为余数，低64bit为商；最后因为商要补最低位，因而高64bit需要额外向右移动1bit，使数的返回大小一致；

操作符：

Div，rem 有符号数除数、余数；divu，remu 无符号数除数、余数；（不会对除0的情况产生报错---->加速大概率时间）

Significant-->精度/exponent----->范围；

浮点数的表示（IEEE）：符号位（1表示负数） + exponent（注意有bias--2^（m-1）- 1） + fraction（注意有隐藏的1）

Exponent----->浮点数实际值等于二进制表现值 - bias；

Single precision：1 + 8 + 23（bias 127）

Double precision：1 + 11 + 52（bias 1023）

对于exponent而言

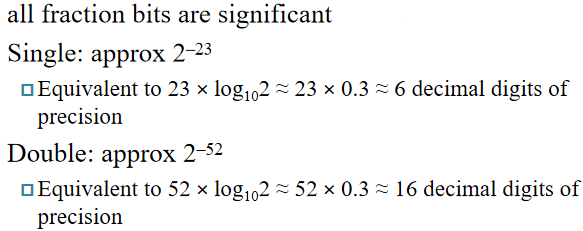
非规格数，对标上述的规格数；（一般而言，最大最小值的计算都限定于规格数中）

全为0最小exponent；表示没有隐藏的1；

全为1最大exponent；若fraction部分均为0，则表示无穷大；若fraction部分不全为0，则表示NaN（not a number）；

Relative precision：

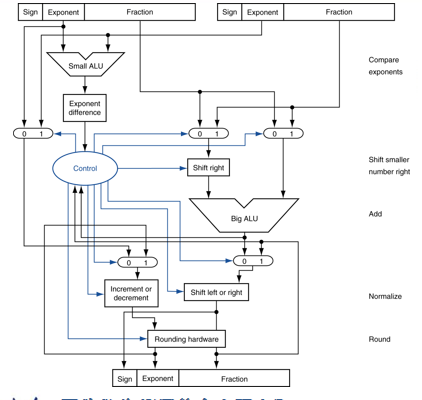
Single/double表示为10进制的位数精度；



浮点数加法的计算：

Exponent小的向大的归约；计算结果，并将结果normalize，转化为符合要求的指数形式；判断是否overflow与underflow；round the result；

不完全示意图如下：

浮点数的乘法；注意exponent相加后需要多减去一个bias；

在浮点数加法时，我们会额外用到两个数位来辅助rounding的执行，使得其可以有更精确的round结果；

Guard位，额外保留位中的第一位；round右侧额外位的第二位（第一第二应该是从左向右数）；sticky bit（粘滞位）用于区别0.5...0与0.5...1间的差距（管理两位之外的数据差距），远处舍去的若不为0，则sticky bit直接置为1；

浮点数使用近邻偶数归约的法则，即24.50 round到24，25.50 round到26（没有sticky bit的使用时）；

Using in the last place（ulp）：计算结果与真实结果之间的有误差的最低bit位置；

Machine language：机器语言；

Assembly language：汇编语言；

High-level programming language：高级编程语言；

Operands：操作数；oeprate：操作；

冯诺依曼结构：1.指令像数据一样存储；2.程序可以像数据一样在内存中读取；

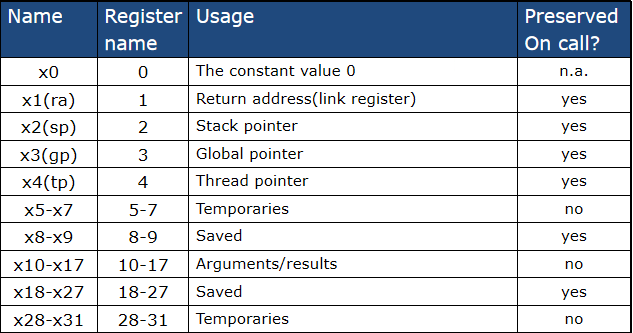
操作的设计：（RISC-V）：一条指令只有一个操作数，一条指令最多操作三个变量；

1.Simplicity favors regularity：规则的指令便于简化；可以以更小的cost介导更好的性能；

2.smaller is faster；

/\*RISC-V使用32个64bit的register；每一个寄存器（register）---->对映于double ward；

各个寄存器的对应说明如下：

一般而言，可以将x5 - x7, x10 - x17（a0 - a7）, x28 - x31, 往往仅作为临时寄存器使用，因而不需要使用stack进行对映的存储，而其余register在使用时，最好进行preserved，防止数据丢失；\*/

Memory is byte addressed；每一个memory格子只占有一个bytes，这意味着一个word可以有四个访问的interfere；

RISC-V is litter endian（小端存储）--- Least-significant byte at least address of a word（数据中的低位存储在memory中的较低地址）；区别于大端存储（big endian）；

RISC-V does not require words to be aligned（对齐） in memory（不要求内存中的word对齐）；但是地址对齐（如新数据的存放起始位一定要为x的倍数）可以更加有利于数据的读取；

注意数组形式的[]中的字节偏移量需要乘以8，转化为绝对偏移量；因为内存中一个location占用一个字节，\*8则占用了八个字节，正好等于64bit，匹配于double ward的需求；在写汇编语言时，则直接给出绝对偏移量即可；

Spill：溢出，洒出，涌出；convention：约定，公约；

保留一个寄存器用于记录0 --->加速一般情况；

/\*Two’s complement signed integers名称可以表示补码；

RISC-V用32bit记录每一条指令；

1. cf890cf0481385d5e7956f80ebb4322format指令：

注意rd（destination register number），rs1（source register number），rs2的分布；

Good design demands good compromises：好的设计需要相互妥协（折中）----->RISC-V使用等长指令；

1. format：

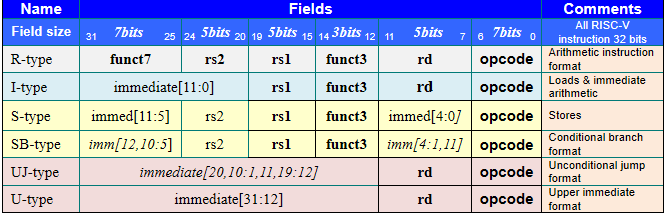
4d585c345c6ba130b8a1b9c4421209f

Immediate表示立即数，一方面需要sign extended，一方面使用2’s complement；

1. format：

5976a06560b838af2894e1002f49887为符合一般指令的结构构成，需要将对映的immediate进行差分；

特别注意rs1，rs2的相对位置；一般而言，介导memory偏移的称为rs1，因而sd指令与其余指令具有不同的排列顺序；



RISC-V中slli与srli左右位移的format；

983177ec6d22f71f87ceb3c2f53162a

RISC-V拥有的操作指令：and，or，xor，andi，ori，xori，sll，srl（逻辑右移，用0填充最高位），sra（算术右移，用符号位填充最高位），slli

Branch指令：

Beq x21, x22, L1 若x21 = x22，则跳转至L1；

Bne branch not equal；不相等即跳转；

Slt set less than；slt x5, x19, x20；若x19小于x20，则将x5置为1；

Blt branch less than；

Bge branch bigger and equal to；

Blt，bge都基于符号数，bltu/begu面向于无符号数的比较；

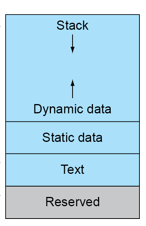
Jalr x1,100(x6)；PC跳转至x6 + 100，并将跳转前的PC + 4置于x1位置；\*\*\*？？？\*\*\*

特别注意，记录的是绝对的偏移量（汇编语言，不同操作数的偏移记录都是绝对偏移量），转化为二进制实际代码时，需要进行相应的转化（除以2）等操作；

Jal（jump and link） x1，xxx；无条件跳转至xxx，把当前的PC + 4记录到x1中，x1称为目标寄存器 ----> rd；

可以用sp模仿stack进行memory内存的认为控制；

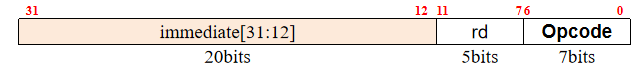
每一个data存储都需要8的byte偏移量---->对映于double ward的64bit偏移；减法开辟空间，从大到小作加利用空间；不要忘记function结束后对利用的空间进行释放；在一个stack结构中，sp往往指向最下端的数据，而fp一直定位于当前function开始时stack的顶端位置，便于function结束后，sp的迅速恢复；

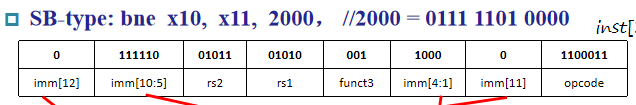
如上图，其中全局变量放置于static data之中，人为分配的内存位于dynamic data结构中自下而上填充，而自动分配的变量则在stack从上至下分配；

对于一般的操作指令而言，加u表示处理无符号数，加b/h/w表示处理byte/half word/word三种不同长度单位的数据；

PPT P70-71有详细的各操作数应用样例及细节；\*/

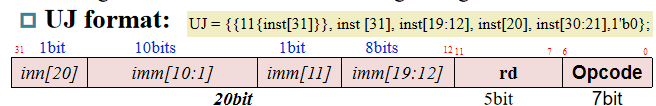
U-format：

lui x19,976；汇编语言中显示的immediate number是真实的高20bit，但在实际的存储中，低12bit还需要全部进行补0；当然，一些难以实现的数字可以考虑少加多减的技巧，配合lui与addi指令使用（特别注意不要超过bit数可以表现的上界）；

SB-format：

汇编语言记录的是真实偏移量，在转化为二进制时，还需要除以2；

UJ-format：

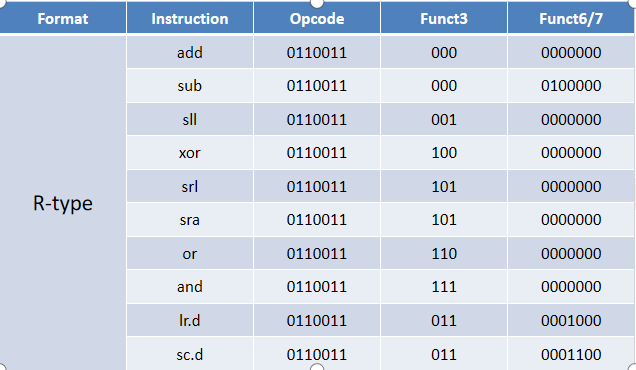
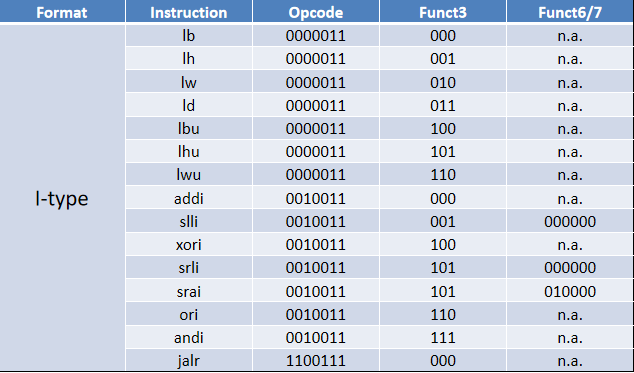
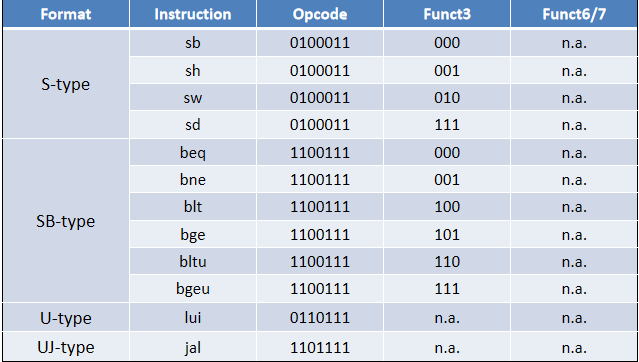
一般对映于jal指令，同样也需要除以2再转化为二进制记录；

寻址方式：

1. immediate addressing 立即数计算；
2. Register addressing 寄存器中数值相加；
3. Base addressing 基于寄存器地址的偏移；
4. PC-relative addressing 基于PC的指令偏移寻址；

一般通过funct7，funct3，opcode来check执行的具体format与操作功能；

指令format集合：

Synchronization：同步，同步化；

Check并行性？

Load reserved：lr.d rd,(rs1)；声明一份地址location，将rs1中数据取至rd；

Store conditional：sc.d rd,(rs1),rs2；将rs2的数据放入rs1中，如果rs1数据在其中未改变，则rd置为0；如果rs1中数据发生了改变（store fail表示store过程本身失败），则在rd置非0数；

文件信息：

文件头信息：文件名称，指令大小，数据大小；

指令部分：指令顺序存放；

数据部分：数据顺序存放；

Relocation information：对会造成地址跳转的语句进行跳转位置分析（一般而言都有标签介导）；

Symbol table：将标签与实际address进行对映；

Leaf procedure：不再调用其他程序的function，不需要进行preserve；递归（自调用）可能不算叶子程序；

Li x3, 5 指令，把立即数5赋给对映的register x3；

Mv x21, x20指令，把x20存储的对映参数copy给x21一份；

MIPS与RISC-V的相同之处：

32bit宽度指令；使用32个通用寄存器，且register 0 always记录0；

只能用ld/sd访问读取读入内存；

X86则相应的指令长短不一；

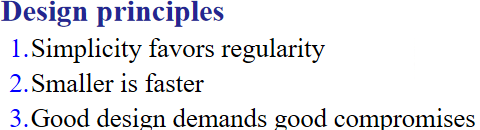
Fallacy谬误：

指令功能并不是越强大越可以有高性能的表现；编译器更愿意执行简单的指令；

并不是使用汇编语言就可以提升程序性能的；汇编语言往往要求更多的代码量，因而会带来更多的错误，生产效率低下；

Backward compatibility（向后兼容）：指令集还是会发生一定的变化，以提升程序的性能；

Pitfalls 陷阱：



1. 指令设计的简洁，有利于更少cost情况下更高性能的实现；
2. Memory每个location都设计的比较小，但因而可以获得给高的程序性能；
3. 折中的设计带来更好的效果，将所有的指令设计成相同的长度，便可以大大提升性能；

执行指令的顺序：

Fetch instruction in memory；

Decode the instruction，read operand；

Executive control，execute ALU operation;

Memory access，read/write data;

Write results to register，写回的来源不同;

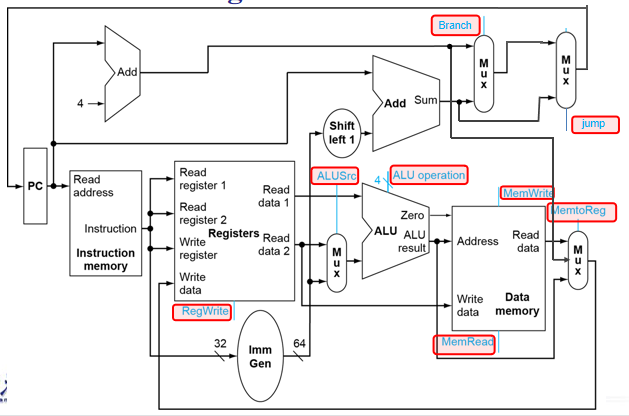
Modify PC，执行branch指令;

记录具体的数据时，使用edge-triggered边沿触发进行更新；可以加入write写信号来控制写入的是否执行；

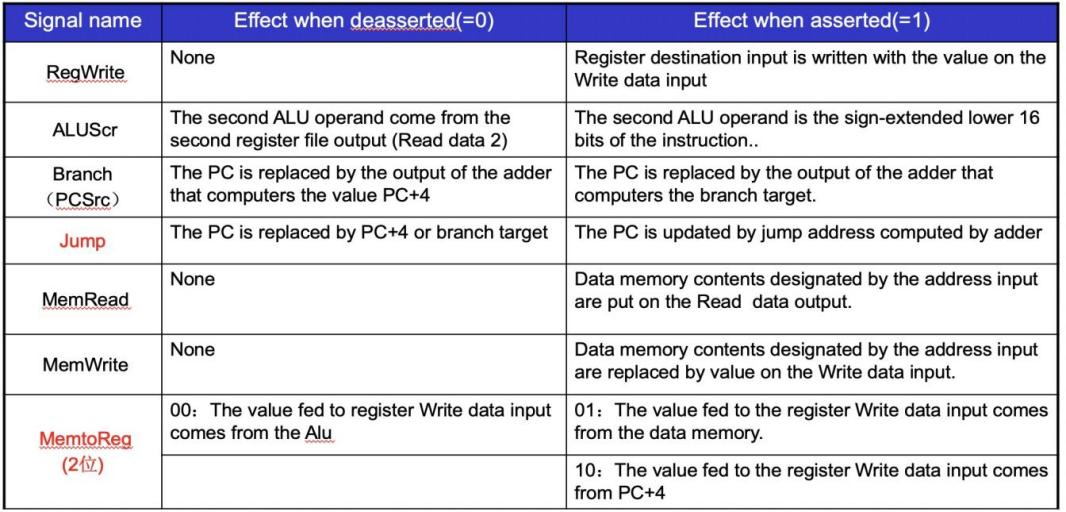
指令在使用立即数时，需要将立即数扩展至64bit（好像只是扩展至32bit诶，可能不是必须的），便于与寄存器中数据进行ALU运算；

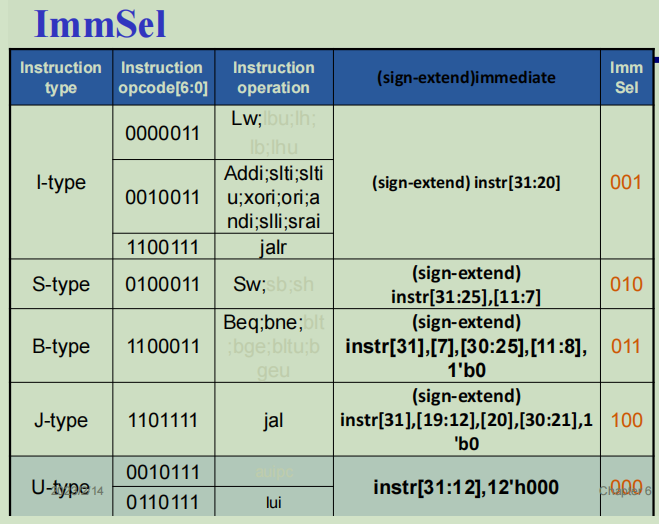
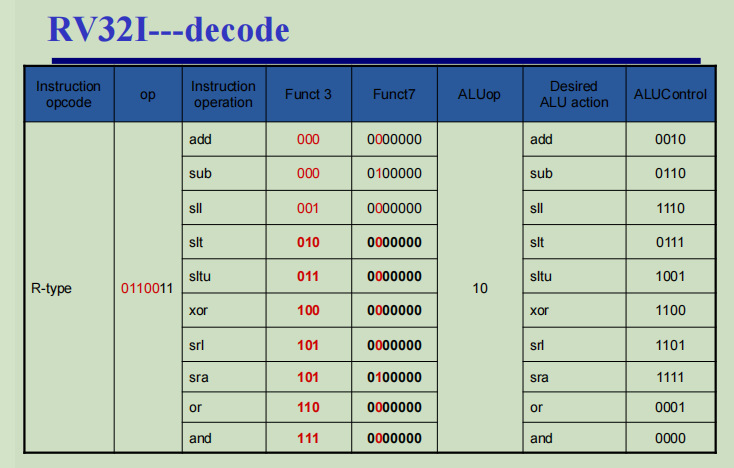
数据通路的详细介绍：CH4-1 P30-41；

There are some control signals：branch，jump，alusrc，alu operation，memwrite，memtoreg，memread，regwrite；

数据通路：

控制信号的具体数值设置情况：





Alusrc 等于0时使用寄存器数值，等于1时使用立即数的扩展；

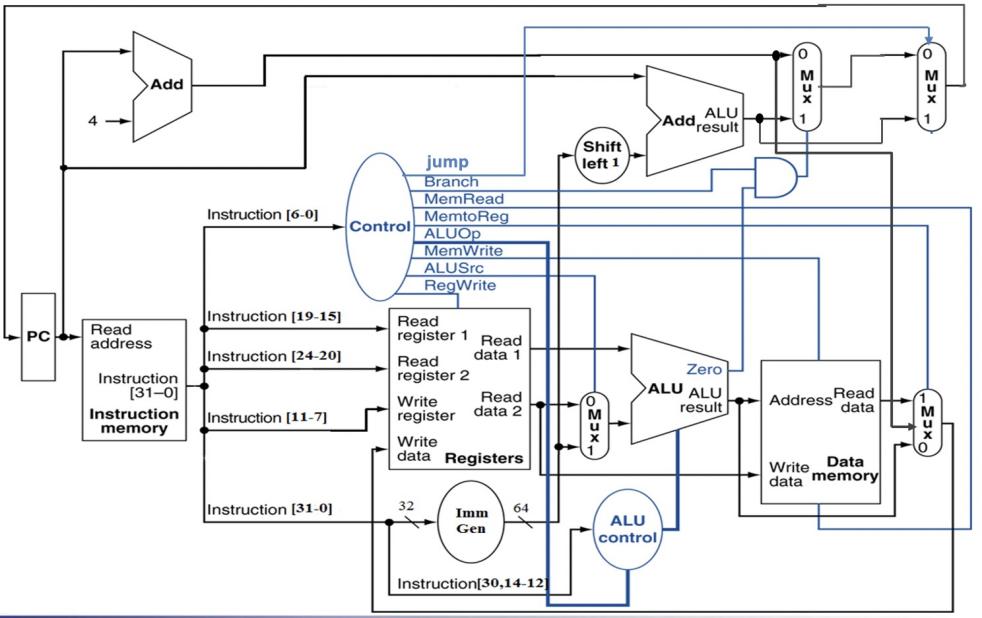
Pcsrc为0时使用PC + 4，为1时使用需要跳转地址的addresss；

Jump为0使用PC + 4，为1使用需要跳转的地址；

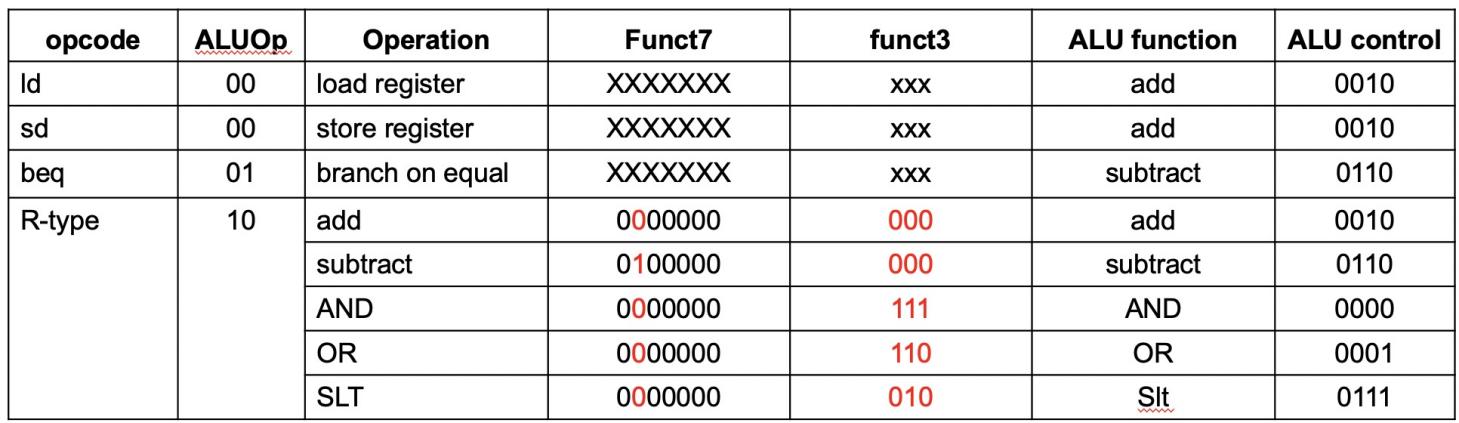
MemtoReg为00时，使用ALU的结果写回寄存器；为01时，使用memory中的结果写回寄存器；为10时，将 PC + 4 的计算结果写回寄存器；

只有opcode就可以决定control signal的具体信息（opcode决定了指令的format），附以funct3与funct7，就可以决定ALU operator的具体使用；

整体性完整记录：

对映的控制信号（X表示无所谓）：



不同指令需要用到的ALU计算单元（根据指令的截断与部分控制信号）：

在pipeline（流水线）中，longest delay determines clock period，往往是load决定最长的时间delay；

Violate：违反；period：周期；feasible：合适的；

RISC-V的具体pipeline执行过程：

IF：获得instruction；

ID：解析instruction，获得对映的register read；

EX：执行对映的ALU计算与地址计算；

MEM：从memory中获得对映的操作数；

WB：将计算结果写回register；

同样，pipeline中每一个阶段需要一个clock cycle，但是每一个clock cycle的相对时间会减少；同时在一个clock cycle中，可以同时执行不同指令间的多个步骤；CPI的计算还是约等于1；

注意对于register file的读取而言，写reg在对映clock cycle的前半周期，读reg在对映clock cycle的后半个周期；

Pipeline并没有改变对映的latency延迟，只是增加了throughput（吞吐量）；

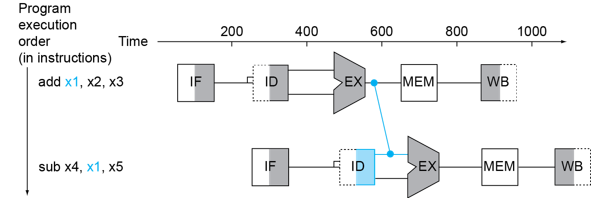
Hazards冒险：

1.Structure hazard：需要使用的资源正处于工作状态；

RISC-V只有一个memory，重叠的操作需要引发stall来过渡因而；pipeline的实现需要separate的指令与数据存储集合；

2.Data hazard：需要等待先前的操作对已有数据的完成，才能继续访问数据（数据的繁忙）；

Forwarding（bypassing）：将一些计算结果在写回register file之前进行旁路传递；

如ld指令，需要在mem环节之后才可以得到相应的数据，因而使用旁路并不能解决问题，因而此时必须插入bubble，造成指令执行的短暂暂停。可以重新排列指令顺序从而减少必须的bubble数量；

3.Control hazard：是否执行分支依赖于后续的计算结果；后续执行的指令依赖于先前指令的计算结果；

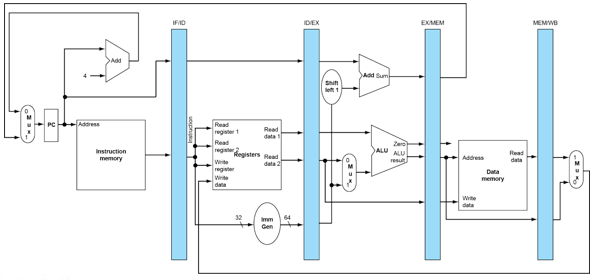
使用stall中断等待 ----> 插入bubble；

Branch也可以使用预测：

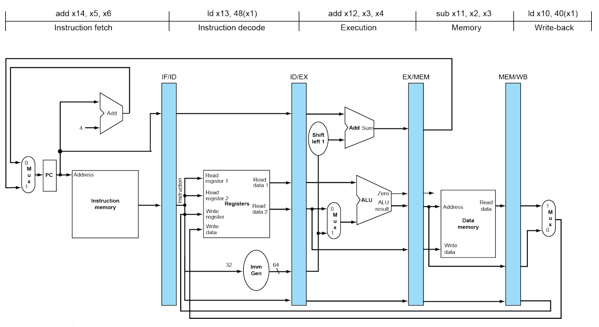
Static branch prediction：基于typical branch 选择，沿用上一次的选择情况；

Dynamic branch prediction：根据统计规律进行预测；

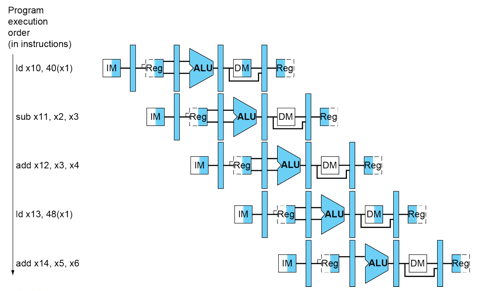
Pipeline需要使用自己特殊的register，用于记录需要向后传递的数据与控制符：



有多种画图方式：

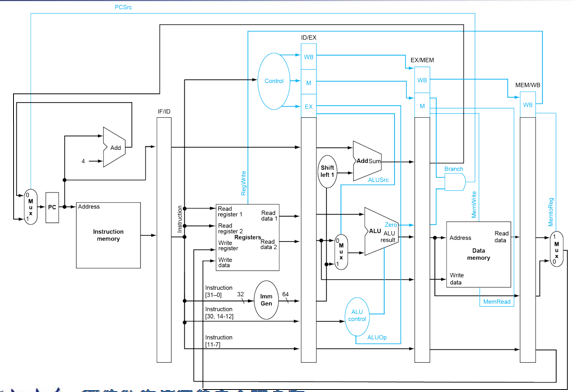
Single-clock-cycle（五阶段的详细截图，每一阶段对映不同的指令）；

Multi-clock-cycle（展现不同指令的全部阶段以及它们的重叠情况）；



Pipeline并没有使用更多的控制信号；

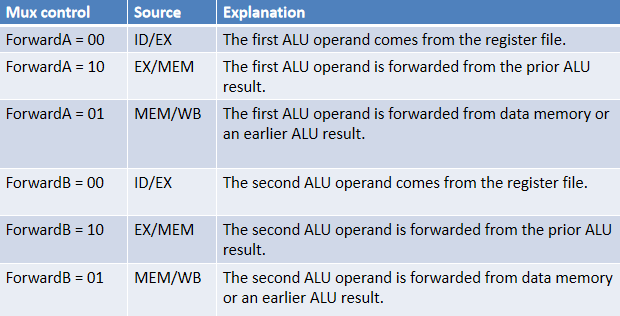
Pipeline控制信号具体的使用：



Detect the data hazards：

首先要存在数据的使用冲突（注意可以是EX/MEM阶段、MEM/RB阶段与ID/EX阶段的冲突），其次计算结果确实要写入register file中，且rd寄存器不为x0，x0始终不会被改写；

一些用于冲突控制的信号的具体数值与作用效力之间的关系：（其中ID/EX与MEM/RB的冲突表现为后续指令已经希望通过ALU计算，但RB需要在同一clock中将数据写回register file，因而正常运行会产生一个clock cycle的时间差距，不能正确执行后续指令）；



需要特殊处理load指令

当ID/EX.MemRead and ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs1))时，说明当前指令为load，且下一条指令对load载入的值进行了使用；因而需要直接插入一个bubble；

在插入bubble时，需要把ID/EX寄存器中的数值均置为0，表示nop（no-operation）的产生；要阻止PC的改变以及IF/ID寄存器的变化传递；

中断会降低性能的表现；但我们必须加入stall去获得正确的答案；编译器也可以将指令重排以达到更少的冒险与中断的需求；

处理branch指令带来的control hazards：如果等待计算结果，需要在EX阶段之后才可以得到对映的分支条件判断结果，这样会导致错误指令fetch与decode的可能性；因而我们要求在ID stage添加一ALU计算单元，可以更早的计算获得对映的跳转条件结果，从而减少必须插入的bubble数量；

使用branch prediction buffer（branch history table）进行动态预测是一种很好的做法；如果预测失败，必须对pipeline CPU的最前阶段进行flush操作；1-bit记录结果不妥当，可以使用2-bit的记录方法，只有两次连续的改变才会使得下一次的预测发生变化；

Exception and interrupts：异常与中断；

解决exception 将下一条instruction对映的PC地址放入SEPC寄存器中保存；记录当前的一些控制信号；跳转至handler进行exception的额外处理；

Vectored interrupts：不同类型的exception解决代码均以完成，可以通过vector table进行相应的地址跳转；

Malfunction：故障；speculation：预测，投机；aligned 校准；

遭遇multiple exception时，可以考虑顺序处理（deal with exception from earliest instruction）的方法，顺序处理往往可以介导precise exception；

一些imprecise的exception解决方法，直接中断程序，并保存对映的状态；或者收集所有中断，让handler自己解决问题；

Instruction-level parallelism（ILP）：指令级并行；

1. 可以使用更多层次的流水线，但不能无限叠加，需要考虑到寄存器的消耗与冲突发生的可能性带来的相应性能下降；
2. 使用多个运行单元，有更多的流水线（x-way），对于此类设计，其CPI往往小于1，因而我们使用IPC来评价其性能（instruction per cycles）；

Static multiple issue：静态多发--->编译器处理，纯编译器帮助检测与规避冒险；

Dynamic multiple issue：动态多发，CPU检测处理冒险；另外，编译器也可以进行对映的帮助；

RISC-V with static dual issue

Loop unrolling：循环展开；

每个clock cycle执行尽可能多的语句，只要保证只有一条语句使用了ALU/branch操作（甚至可以添加一个ALU操作），还有一条指令执行了如ld/sd的操作指令；

只用了packet的概念；very long instruction word（VLIW）将多条语句group打包的概念；x-issue packets（将x条指令进行打包并行处理）；

同样，可以考虑将x-issue的多条语句再进行打包，扩大register的使用个数，使数据使用之间不存在register相同带来的hazard，从而可以更好的提升IPC数值；CH4-part2 P86/88对比图；

使用寄存器重命名（register renaming）来克服anti-dependence（反相关）/name dependence（名字相关）的问题；（数据本身没有复用先后关系，因为被迫使用同一个register，从而导致了冲突的发生）；

动态多发生处理器：

Pipeline CPU 可以乱序计算，但一定只能顺序提交计算结果；先存缓冲区，然后可以进行乱序执行，再将执行结果顺序提交。在缓存区中，如果所有数据及运算单元已准备完成，则指令直接进行运算；若有成分尚未准备完全，则相应指令就继续等待。

动态多发生器的必要性：1.不是所有install pipeline都可以预测，如cache的失效便往往是难以预测的，动态调度允许执行其他指令来隐藏这一问题；2.在编译过程中无法知道指令真实执行速率，仅仅使用预测，会极大降低程序的并发性能；3.不同流水线有不同的延迟与带宽，会改变编译代码的最佳配置。因而只依赖于编译器的优化是不足够的。

在分支精确计算及内存读取内容写回寄存器之前，不能提交相应的branch与load指令，即使当时的预测是正确的；

使用更多更简单的core可能是更好的选择；

谬误：流水线的等结构设计，与工艺存在极大的关联；

ISA的设计与data path与processor的设计相互影响；

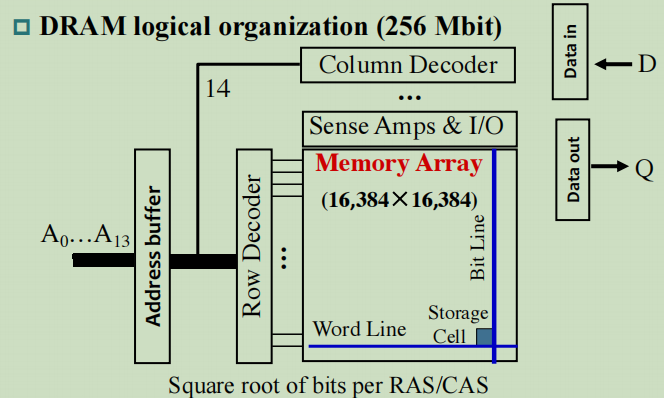
存储器的层级结构：

Memory：使用SRAM结构；速度更快，但是相比于DRAM会占用更多的空间；

SRAM是静态RAM，DRAM是动态RAM；SRAM速度更快，但存储一bit需要6个晶体管，发热量更大，不适合作主存储器，因而常用来作为cache；成本较高，但速度极快；

DRAM是动态RAM，需要不断刷新电路来保持数据，否则内部数据会消失；DRAM只需要一个电容一个晶体管就可以存储1bit因而常用作内存的构成；价格相对低廉，但速度也相对较慢；

DRAM的构造：

SDR：仅在时钟上升沿进行操作；

DDR：时钟的上升下降沿均进行操作，因而双倍速率；

QDR：集合读写操作，上下沿均触发操作于一身（两个接口），因而表现出四倍的速率；

Nonvolatile semiconductor storage：非易失性半导体存储器；

位于DRAM与disk之间；

Each sector records：

1. sector ID；
2. 扇区对映的数据信息；
3. Error correcting code（ECC）；

Access to sector 包括：

1. 等待时间，如果其余access正在操作；
2. Seek寻道时间；
3. Rotational latency，旋转等待时间；
4. Data transfer时间；
5. Controller overhead time；

Temporal locality时间局部性，最近使用数据临近使用可能性较大；（recently）

Spatial locality空间局部性，空间存放较为接近的数据间连续使用性较大；（near）

Hit：upper level还有CPU需求的对映对映数据；

Hit ratio = hits/accesses；命中概率；

Miss：数据必须从lower level中copy获得；

Time taken：miss penalty；目标miss获取代价；

Miss ratio = 1 - hit ratio；

特别注意，即使目标判定为miss，也需要hit的判断时间消耗；

Hit time：判断所需求数据是否在upper level中；

Miss penalty：替换upper level中block数据，并将这些数据传递给处理器所需要的时间消耗；

Memory hierarchy：存储器结构，最上端速度最快，但内存最小；越往下存储空间越大，对映速率越慢；

Direct mapped直接映射：通过mod计算的单个block存储映射；

Memory block address的构成：TAG + index + Byte offset；

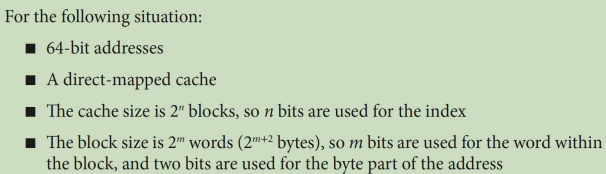
其往往大于cache block address；

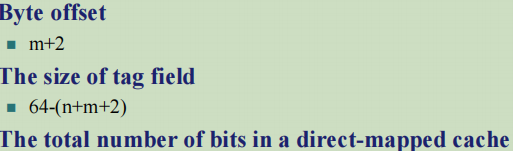
Cache则往往由 valid bit + Tag + Data构成；

其中valid bit表示cache中当前的对映位置是否存在数值；Tag表示memory address的对映寻址；Data表示存储的具体信息；index bit个数决定cache中有2^index条语句；Tag的长度往往为计算而得，判断当前记录行对映的具体内存映射，为disk中的何条特别数据行（disk为多对一映射cache，因而需要进行区别）；

一些长度相关的计算：

注意是byte size，表现为byte搜索，因而最小单元只需记录为byte类型即可；不同要求不同处理；

对映的计算公式：



Miss：分为instruction与data miss；

Write hit分为两种策略：

Write-back：只将数据写入cache中，可能会引起inconsistent；后续在将cache中数据写入memory中，这样会获得较快的写入速度；

Write-through：数据直接被写入cache与memory，保证consistent；但是写入会很慢，可以考虑使用一个buffer；

Write-miss：则需要先把block读到内存块中；

Write allocate：将对映的block读入cache中；（write-back策略常常使用）

Write around：直接将block写入memory中，即block不写入cache中；（write-through策略常常使用）

对于through而言，Write-round不需要don’t fetch the block；write-allocate则需要fetch block；

对于back策略需要fetch the block；

Cache需要考虑的四个方面：

1.block placement

Direct map：每个disk中的block只能对映cache中的一块block；usually address mod numbers of blocks；

Fully associative：block can go anywhere in cache;

Set associative：Block can go in one of a set of places in the cache；A set is a group of blocks in the cache. Always block address MOD Number of sets in the cache；every set have n blocks ---> n-way set associative；

2.block identification

Tag，用于比较disk多对一cache的正确性；

Valid bit，判断cache中当前block是否处于有效读取状态；

对于physical address的format而言：

对于set associative而言，index等于cache中set的个数；对于block associative而言，正常index等于cache block总数；对于fully associative而言，不需要index；

对于byte offset而言， 只要block大小一致，direct map与set associative是同样的长度；无论分割形式，byte offset总和每个block中byte的数量有关；

Fully associative完全依赖于Tag进行cache数据的匹配；

3.strategy of block replacement

random replacement：随机替换block；

Least-recently used（LRU）：最近最少使用替换；

First in first out（FIFO）：最先近来的最先出去；

4.write strategy --- 见write-through，write-back解释；

其中使用write-back策略，需要为cache多设置1bit dirty位，因为cache与memory is inconsistent；当cache与memory之间的写入带宽较窄时建议使用，可以大幅提升速度；

Write stall：CPU必须等待write through的过程，因而发生了中断；

Write buffer：to avoid install，将一些需要写入disk的数据保留在其中；

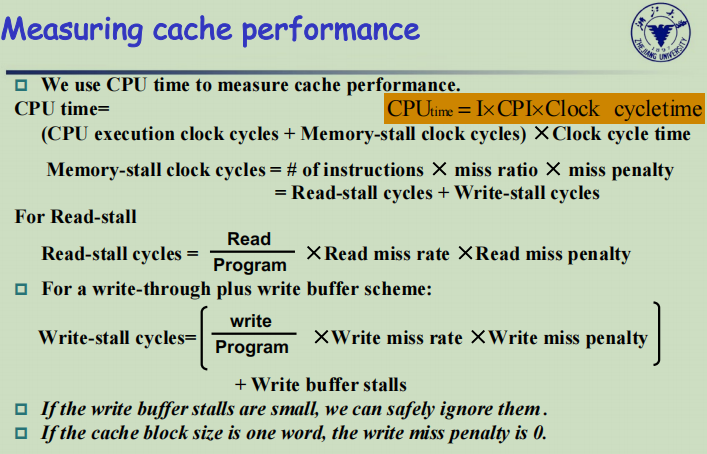
Cache的表现与改进：

1. measuring the performance
2. Reducing cache misses
3. Reducing the miss penalty

Average Memory Assess time = hit time（每次访问，无论hit or miss，都需要一次hit time） + miss time

We use CPU time to measure cache performance.

Cache表现CPU time的具体计算法则；



对于miss penalty而言，我们需要用实际的clock cycle数来对其进行衡量，我们默认对于不同的时钟频率而言，miss penalty的时间开销是相同的；

一般认为只有ld/sd会造成data cache miss，而instruction的 miss则经常发生；

如果减少CPI，即每一clock cycle的instruction数量翻倍，则含有中断的CPI放大倍数反而还会相应的上升；

如果提升时钟频率至原始的双倍；with miss的CPI上升幅度会有一定的下降；

减少cache miss的方法：

使用set associative cache；但是set中含有的block数量也不能过多；在实际获得数据时，我们需要去遍历set中的每block以寻得匹配的data；一般使用2/4-way set associative；

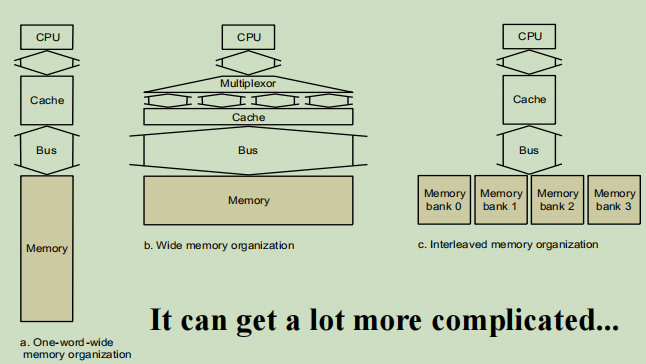
选择合适block进行替换的方法：

最常用的便为LRU策略；如果set中对映的block数量上升的话，替换顺序的管理便会变得十分复杂；

Block size的选择：

larger的block size会有更低的miss rate；但是在cache大小统一的前提下，更大的block会导致更少的blocks，因而发生block冲突的可能性同样上升；且larger block的读取写入开销更大；

优化设计memory system：

如图：（细节在CH5 P88 - 90）

一方面可以细化为the main memory width of x-words（可以同时传输x个word），相比之下，随着width的上升，bandwidth也可以得到显著的提升；

另一方方面，可以使用x banks interleaved memory；（一次寻址后，可以同时访问x块内存，传输仍需要一个block一个block传输）；

使用multilevel cache（多级cache）减少miss penalty：

注意对映的miss概率是单层计算，还是整体结构的miss概率；

因而设计时需要关注第一level的hit time，以及第二level的miss rate；

软件优化blocking：

在block被替换之前，最大化使用block；

Virtual memory:

在virtual memory中block又名page；translation miss is called a page fault；

page fault：不再内存之中，需要到disk中去retrieve（检索）；

注意：因为page较大，有很大的miss penalty；一般使用LRU策略；一般使用write-back 策略而不是write-through策略来减小写回开销；

可以更为高效、安全的在不同程序之间分享内存；程序使用空间不再受到内存大小的限制；

Virtual memory往往大于memory；virtual memory中block部分对映于内存，部分对映于disk；

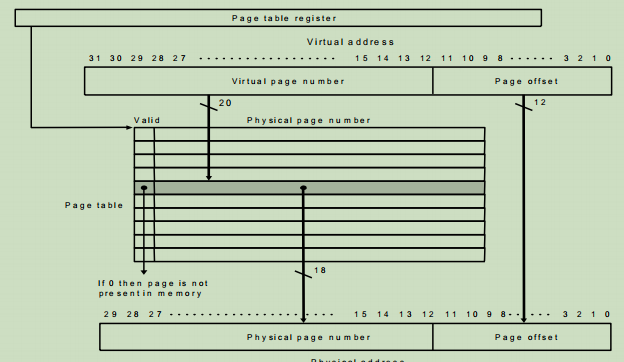
优点：更多的内存空间；便于程序重定位；用于保护；

Page tables：

Page table, Program counter and the page table register, specifies the state of the program. Each process has one page table.

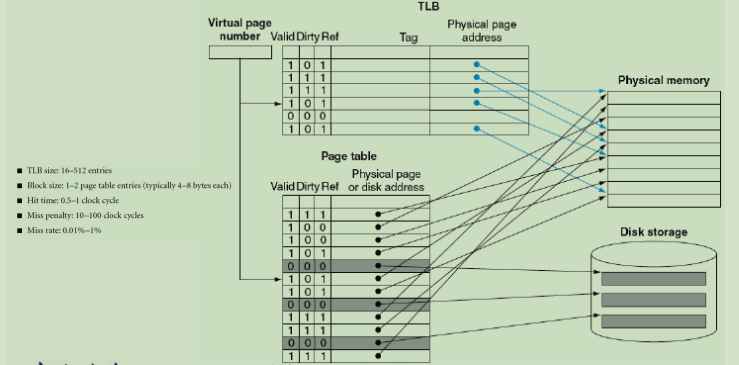
示意图如下，其中virtual page number往往会比physical page number长一些；VA最低位（一般为12位，实际与page大小一致）为page offset，需要直接传递给PA；可以使用VPN到page table中进行PPN的搜索，将获得的PPN与page offset进行拼合后得到PA。

Page table由valid位与PPN的记录信息构成；

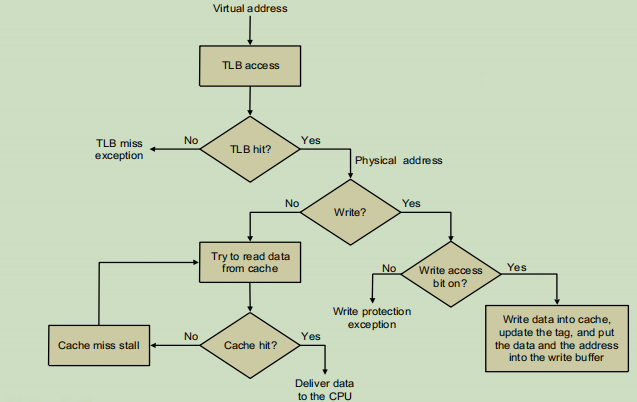
因为使用的往往是write-back策略，所以需要有额外的dirty bit记录是否修改了信息；dirty被置位的page需要写回disk后才可以被替换；

类似于memory与cache的关系，page table也有相应的cache；

Translation - look aside buffer（TLB）：

一般由valid、dirty、ref（各1bit）以及Tag，PPN构成；一般而言根据VPN去比对Tag即可；Tag的长度往往与VPN的长度一致；

TLB与cache的总体使用关系：

Page table中的信息，在memory中一定会有对映；当然另一部分则在disk中；如果明确地知道对映的page在disk中，这对于page table而言也是hit的。

处理器的性能提升往往快于存储器，因而两者间存在“内存墙”；

I/O篇章（appendix）

I/O需要考虑到很多问题：如performance性能；expandability可扩展性以及resilience恢复力/适应力；

I/O三要素：behavior（input只能读/output只能写/storage可以重复读与重复写），partner（在I/O设备另一端人或者机器），data rate（在memory与处理器之间的最大数据传输速率）；

I/O性能的衡量：

Throughput：1.单位时间内该系统中的数据传输量；2.在单位时间内，可以执行多少步I/O操作；

Response time；

Disk storage and dependability（磁盘存储与可靠性）：

Magnetic disk的两种types：

Floppy disk；

Hard disk，较大、更高密度、更高的数据传输速率、有多个platter（磁盘盘面）；

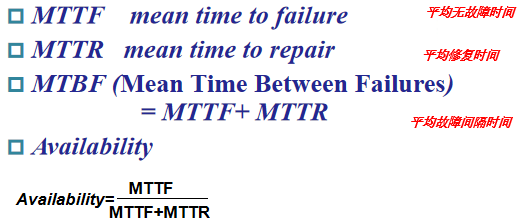
Concentric：同心的；

Disk consists of a collection of platters，each of which has two recordable disk surfaces（每一个platter都是双面可读的）；

时间计算四部分：Seek、rotational latency、transfer、disk controller；

Dependability：系统提供服务的质量，以及服务的可保证性；

正常为Service accomplishment，崩溃（被中断）则为Service interruption；

Availability可用性：

改变MTTF的方法，1.fault avoidance；2.fault tolerance，使用冗余，去容忍忽略disk产生的部分问题；3.fault forecasting；

如果将一块大硬盘换成使用x块小硬盘，则MTTF（该盘的reliability直接下降n倍）；

RAID（redundant arrays of inexpensive disks）：

Level（throughput吞吐率的考虑 ---> 传输速度的考虑，数据量与时间兼顾）

Overhead：开销；

1. 没有任何校验盘，多块盘可以都是被使用；
2. 有同数量的校验盘；
3. ECC需要三个额外盘；
4. bit校验，只有一个多余盘（bit存贮--相当于拆块了），使用奇偶校验法进行数据check与recovery；把数据块且成n块，存放于n个盘上；
5. bit校验改成block校验；1 Logical Write = 2 Physical Reads + 2 Physical Writes；在小规模写上有较好的表现（仅写一块disk的smaller write），写多块则会产生问题；数据块单盘存储；
6. 没有特殊的校验盘，校验结果分布于各个盘；
7. 类似于5，但对于每一块盘都有一个备份；

Bus basics：

Control lines释放控制信号，data lines传递information；

Bus transaction：传递地址与接受数据（包括input与output）；

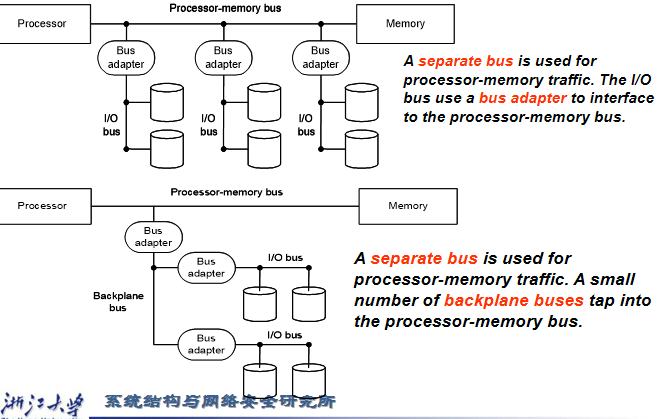
Types of bus：

Processor-memory：断而速度快，定制设计；

Backplane：速度快，往往是标椎化的；

I/O；

两类结构（如下图）：前者每一个I/O器件都与processor/memory连接；后者只有一个接口，且不同I/O设备之间可以直接交互；



Synchronous：同步的，所有设备都需要使用相同的时钟周期与频率；

Asynchronous：异步的，不使用clock而使用handshaking方式建立联系；

异步过程往往要执行七个阶段；CH6 P48 - 49；

BUS master，用于处理不同设备间访问权限的请求，处理已提出的访问请求；

BUS arbitration schemes：bus主线申请的使用模式：

Daisy chain：串联式，设备按一定的顺序连接在总线上，类似于遍历，当一个请求信号发出时，需要通过后续所有的链接设备，用于check是否会发生控制的冲突；较低的控制效率与交长的延迟；

Centralized：集中式，使用专门的控制器来控制权限的分配，将权限申请按照预先设定的优先级与算法来决定设备的总线优先控制顺序；这样做可以有较高的控制效率与较低的延迟；但专门的控制器将成为系统的瓶颈；

Self selection：自选式，分布式仲裁办法；每个设备根据一定的优先级与规则自主选择是否发送控制总线的需求，设备通过检测总线上有无申请信号进行控制权的分配；有较高的控制效率与较低的延迟 ----> 但可能存在冲突与竞争；

Collision detection：冲突检测，设备可以同时发送仲裁请求，如果检测到冲突，则设备会进行相应的处理，如等待一段时间后重新发送请求；这样会带来较高延迟与高低控制效率的问题；

同步与异步分析：

CH6 PPT P54-55，特别注意！异步程序的部分handshaking过程本身就是数据的读取过程（时间消耗只需取最大值即可）；

增大 bandwidth是意味着提升传输速率；可以考虑separate地址传输线与数据传输线，或在每一次传输multiple words；

Give commands to I/O devices：

1. memory-mapped I/O；部分内存地址被分配给I/O程序接口，可以使用ld/sd去访问这些接口，进行I/O操作；
2. Special I/O instructions；使用特殊的指令进行I/O操作；

Communication with the processor：

Polling（轮询）：processor不断去check是否有I/O设备提交的访问控制进程；

Interrupt（中断）：I/O设备在需要控制时，去打断processor的工作，从而获取注意力；

DMA（direct memory access）：设备直接从内存中读取数据，跳过processor的控制；

需要以下三个步骤：

1. processor set up the DMA，设备的识别信息、操作，需要被操作数据的地址与资源，需要传输的byte数量；
2. Starts the operation，开始传输数据与属性；如果存在多个需要传输的目标，则在此过程中DMA可以获取下一内存地址并进行初始化；
3. 当DMA传输结束时，controller 中断processor，去检测数据传输过程是否发生了error；

三者间比较：polling会waste大量的processor time，DMA不需要processor的控制，因而不需要多少processor time；

Polling占用processor time的例题P70 - 76

Designing an I/O system：

I/O rate ---> 每秒I/O操作极限，需要面向I/O system的最弱势部分；

在bottleneck方面，考虑CPU或者memory bus；然后根据给出的disk传输速率决定disk 的使用个数、对映controller个数的等硬件条件；（一方面是disk个数要求，一方面是transfer输出速度要求）；

例题P80 -

