计算机体系结构 总复习

第一章，Introduction

计算机体系结构的定义：
1. The attribute of a system
2. Conceptual structure and functional behavior
3. The logic design
4. The physical implementation
5. 硬件和软件的接口
6. design of the abstraction layers between Application and Physics

在OS和微体系结构（称为计组）之间ISA（Instruction Set Architecture）

计算机体系结构：
1. 指令集
2. 内存管理和保护
3. 中断和陷阱（？）
4. 浮点数标准

计算机组成（微体系结构）：
1. number/location of function units
2. pipeline/cache configuration
3. datapath connections

计算机的实现：
1. 低级的元件

计算机体系结构的好处：
1. 避免重复错误在微处理层出现
2. 软硬件分离的实现

第二章，从历史上的计算机看ISA

Ada
第一个高级语言：ALGOL
中国的：DJS-100 series

第三章，CISC vs RISC - 两种体系结构的

历史演变

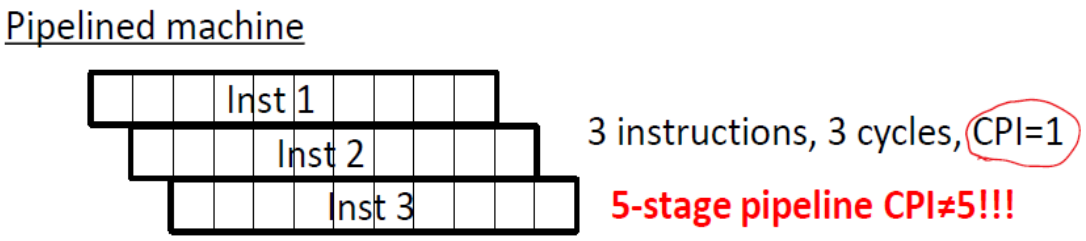
Pages:

1. 标题
2. 上集回顾。计算机体系结构刚开始的130年，从babbage到IBM360，从计算到全能化编程机器；从电子机械到纯电子。软件开发对体系结构有很大的约束，需要软件可兼容性。IBM系列机在不同的机器上跑着相同的ISA，但是底层的实现不同。
3. ISA。是软硬件之间的桥梁。传统地被定义为程序员可见的，状态（寄存器加内存）以及对状态的操作。IBM 360首先实现了ISA的定义与实现的分离。一些ISA的例子。以 Berkeley RISC-V 2.0
4. 控制vs数据通道。数据通道是数字储存、计算的地方，操作是一系列的操作作用于数据通道上。最初的计算机设计难题在于控制单元的正确性。Maurice Wilkes发明了微程序来设计控制单元。
5. 50s时微编码（microcode）出现，导致不同的技术出现：
    a. Logic：Vacuum Tubes 真空管
    b. Main Memory：Magnetic cores 磁芯
    c. ROM：Diode matrix，punched metal cards 卡... ROM比RAM便宜又快。
6. 微编码的CPU图
7. 单总线数据通道
8. RISC-V指令执行阶段：取指令，解码指令，取寄存器，ALU操作，储存器操作（可选），寄存器回写（可选），计算下一指令地址。
9. 微编码的例子
10. 微编码的例子
11. 纯ROM实现的控制器，重点在于计算位宽。带问号的位表示bool值。ROM宽=2^|address| * |data|。|address| = |wPC| + |opcode| + 1 + 1；|data| = |wPC| + |control signals|
12. 纯ROM控制器的运行例子。
13. 一个计算的例子。
14. 上一个例子的带size版。
15. 如何减小ROM control的大小，用外部逻辑编码输入、操作码成组来共享编码，以减小address，ROM height。对微PC重新编码，对控制信号进行编码，减小data宽度，ROM width。
16. 最终形态的RISC-V设计，使用了附加的逻辑电路，微PC jump。
17. 微PC Jump分成五种，next 增加微PC，spin 等待内存，fetch 开始指令fetch，dispatch 开始解码操作组，fture/ffalse 判断fetch的cond。
18. ROM编码的内容，原来的next PC已经变成了PC Jump的五种指令。
19. 其实这样的编码是比较难实现的，需要对datapath进行修改。
20. 水平和垂直的微代码。什么两种不同的伪代码对微指令的不同的作用？？？说是要结合两者的优点？？？变成了什么nanocoding？？？
21. 微代码ROM通过nanoaddress解析到nanoinstruction ROM，再执行。
22. IBM 360最初的实现。
23. 一个表格，Microprogramming在IBM 360最初的实现。
24. Microcode Emulation 讲了一堆IBM和当年的死对头的历史。
25. Microprogramming 在70s称为了最成功的设计方式。

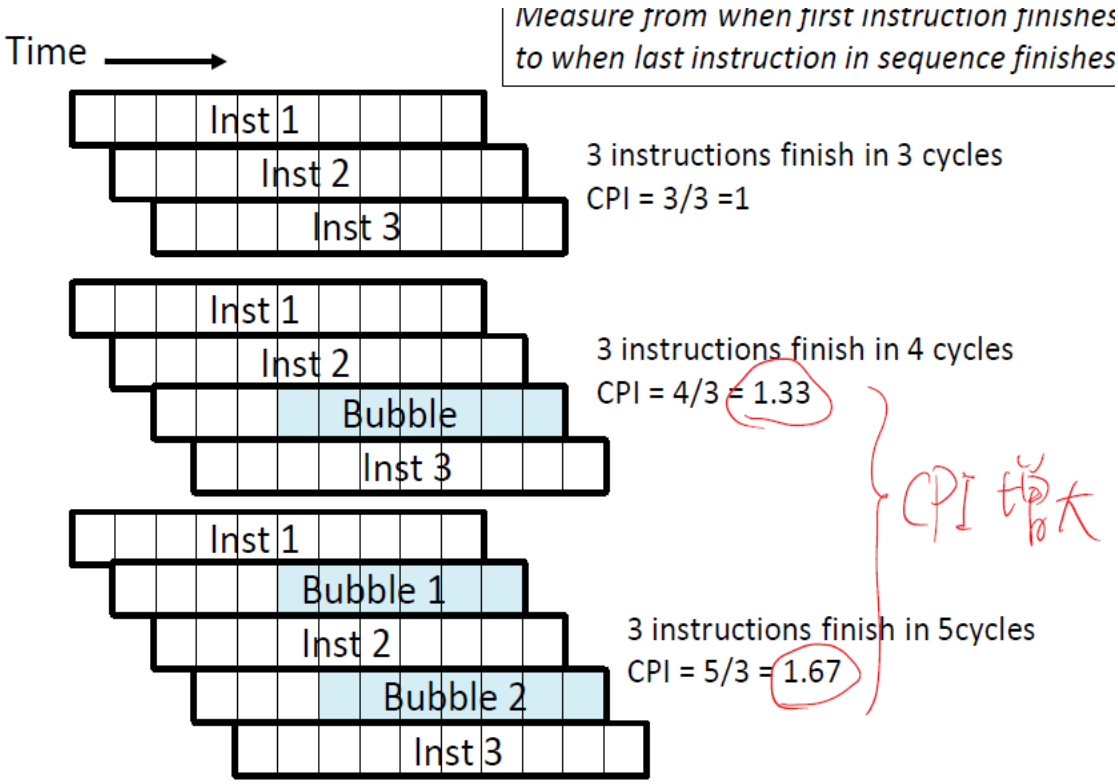
26. 处理器能力的“Iron Law”：Time/Program = Instructions/Program * Cycles/Instructions * Time/Cycle。其中：Instructions per Program和源代码、编译技术以及ISA有关；Cycle per instructions(CPI)与ISA和微体系结构有关，这个比较能优化；Time per Cycle与微体系结构和基础技术相关。
27. 求CPI时，求个平均值就好了。
28. 世界上第一个微处理器，Intel 4004， 1971
    a. 70s年代的微处理器。最初的目标是嵌入式的控制。8位微处理器用在了PC上。
29. 一些用了微处理器的机器的介绍。
30. 70s年代的DRAM，由Intel提出。
31. 微处理器的发展：balabalabala
32. IBM PC, 1981
33. 同上
34. 80s年代初期，为了动态debug，提出了Writable Control Store（WCS）。
35. WCS替换到原来的ROM控制器，提供了修改微代码的可能。但其实除此之外没啥用。
36. 一些对微代码机器的分析。历史。
37. 同上
38. 后面都是一些夸奖Nanocoding、RISC、Microprogramming。

第四章，pipeline

1. 标题
2. 上集回顾。微编码是一种有效的的技术去管理控制单元的复杂性。在逻辑单元、主存、RM使用不同底层技术实现的时代被发明出来。ROM和RAM之间的速度不同推动了更多的复杂指令的引入。技术发展导致了fast SRAM假设的破灭，复杂的指令阻碍了并行与流水的实现，而简化的指令集机器具有更好的效率，也导致了微处理器的发展与流行。
3. Iron Law。其中最重要的是CPI，它与微体系结构和基础技术相关。有个表，Microcoded的CPI>1，但cycle time是short；Single-cycle unpipelined的CPI为1，但cycle time是long。Pipeline的CPI也是1，而且cycle time是short。所以可以看出pipeline真强。
4. 经典的5状态RISC pipeline。Fetch，Decode，EXecute，Memory，Writeback。同步读写。
5. 每个Inst都要经过那五个步骤，但是！！！pipeline可以并行地执行多个inst。因为每个步骤的功能单元供一个inst使用，但是多个inst只要处于不同的状态，就能同时使用一个pipeline上的不同功能单元。



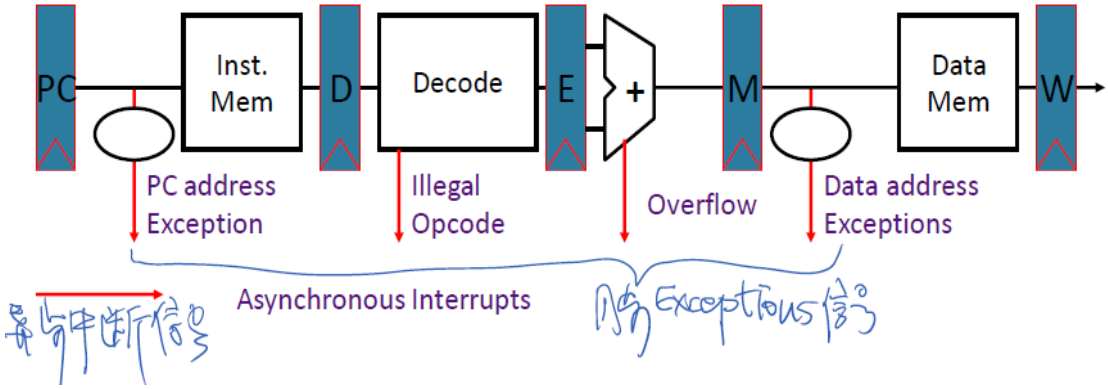
6. Insts之间的相互依赖，会导致：功能单元被使用，structural hazard；数据依赖，data hazard；地址依赖，控制依赖，control hazard。解决方法：在pipeline李加入bubbles，使CPI变大。
7. 加入bubbles的例子，其实就是加入了等待的时间。



8. 解决结构性的冲突。出现的原因是两个insts同时要用同一个硬件资源。可以通过等待来解决（其实那就是没有解决），还可以通过增加硬件资源（不可持续发展）。纯整数计算的流水一般没有结构性冲突，多周期单元容易引起结构性冲突。
9. 数据依赖，分为三种，RAW/WAR/WAW，后两种在一般的pipeline中一般没有，但是在复杂的pipeline，例如OoO中可能有。必须要保证program ordering的读写。
10. 三种解决data hazard的方式：Interlock互锁stall，Bypass绕行/回避forwarding，Speculate推测。
11. Interlocking，要等待好几个cycle；bypass，在X阶段直接传值给下一个指令的X阶段，不用等待。
12. bypass做了那么逆天的事情，肯定是要对datapath进行修改的。
13. 一个完整的bypass数据路径。
14. 与其等待RAW，不如猜它！可以用在：分支预测、栈指针更新、储存器地址更新
15. 控制hazard。各种跳转指令。PC的重要性。
16. 在F/D/X阶段都需要一些control flow info。
17. RISC-V 无条件的PC相对跳转。用一个BUBBLE来废除一条指令（就是跳转的顺序后一条条）。
18. 上一页的例子。j target指令，在D阶段就知道了要跳转到哪里，用fkill吧第二条指令废除，而用PCSd选取要跳转的目标。
19. 早期的RISC在跳转的时候会先执行跳转语句的下一局再进行跳转，因此要再跳转语句的下一句增加一个NOP，或者是有效的指令。
20. Post-1990 RISC ISA就没有那个delay slots了。但是它把微体系结构的东西引入到了ISA，不符合ISA的抽象。影响了执行的效率，与cache相关。使更高级的微结构变得复杂，需要更好的分支预测。
21. RISC-V 条件分支（IF-THEN）：在X阶段产生跳转条件，因此在D和F阶段都要被kill。
22. 上文的例子。一个跳转导致两个阶段被kill，因此产生两个bubbles。
23. Jump Register（JR X1为例）：其实也是有条件判断，所以也是在X阶段才产生跳转，所以要kill两个阶段，所以有两个BUBBLES。与J target不同。
24. 为何指令不会在每个CYCLE被调度：

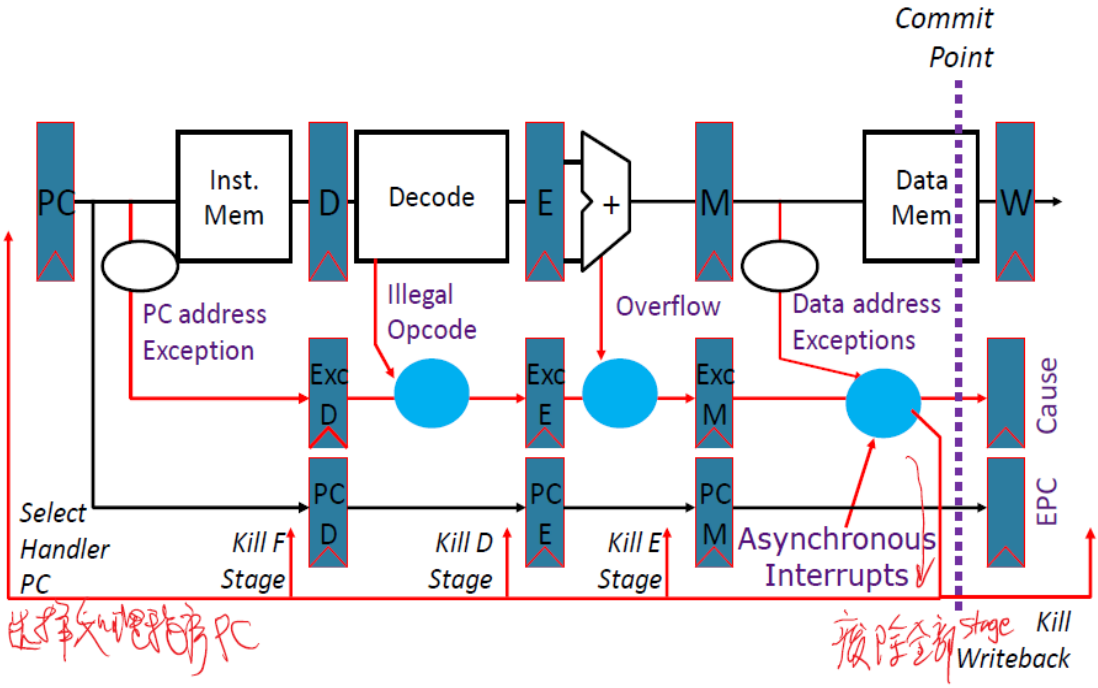
a. 全BYPASSING的datapath太贵了，一般不会去实现。
b. 装入指令有两个时钟的时延。
c. 跳转/条件分支会造成bubbles。

25. Traps and Interrupts。
    a. Exception异常：不正常的内部事件，比如缺页、溢出等等。
    b. Trap陷阱：由异常引起的控制转换（传递）到管态（supervisor）。
    c. Interrrupt中断：程序执行之外的事件引起控制传递到管态。
    d. 陷阱和中断通常在流水线上得到相同机制的处理。
26. 异常处理的历史。
27. 异步中断。由一个有优先级的中断请求线向处理器请求中断。当处理器接受并处理该请求的时候，会把当前的inst停止，执行完其之前的insts。保存PC到EPC寄存器，disables interrupts，进入核心态。
28. 中断处理程序。
    a. 保存EPC在允许中断，保证不会有嵌套中断。
    b. 读取状态寄存器发现中断的原因。
    c. 使用一个特殊的SRET指令（中断返回指令）来返回。它会：允许中断、把处理器交回用户态、读取硬件状态和控制状态（应该就是寄存器从栈里取回）。
29. 同步陷阱。由特定的指令或异常导致。一般来说，异常的指令需在异常处理后重新启动，需要恢复一个或多个指令的执行。系统调用时，该指令要先被执行完。
30. 一张图区分同步和异步的异常在pipeline上的出现时机的不同。



两个问题：1. 如何处理在流水线上同时出现的不同异常。2. 如何处理外部中断。

31. 一张图，说明同步异常和异步异常如何在pipeline上处理。



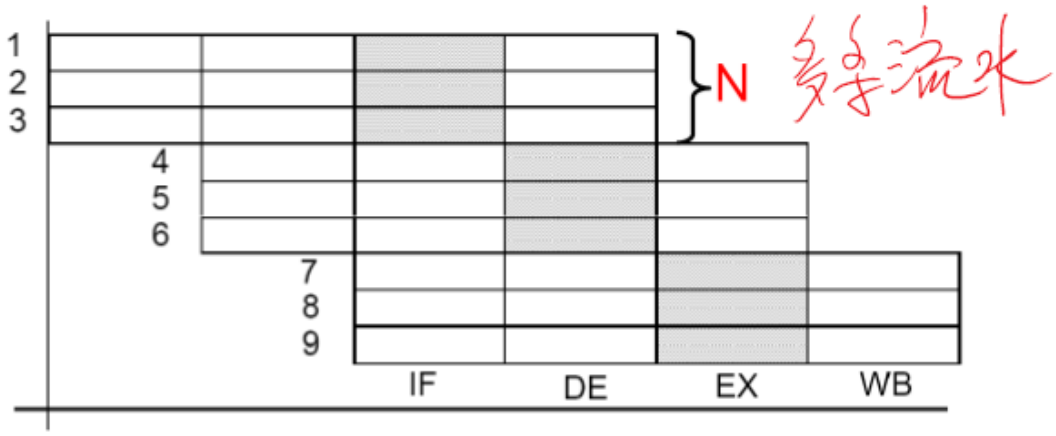
32. 对上一张图的解释：
    a. 直到commit point之前，保持exception flags。
    b. 在给定的inst中，处于前面状态的异常会覆盖后面的（或者是移动到后面）。
    c. 在commit point注入外部异常（覆盖其他）
    d. 当异常到达commit point时，更新Cause和EPC，kill所有其他状态，长如异常处理程序的PC到Fetch stage。
33. 异常推测。
    a. 预测无异常
    b. 指令执行后检测异常
    c. 恢复机制。只有当inst执行到commit point的时候才写状态，遇到异常的时候就可以直接丢掉。
    d. Bypassing允许把未commit的指令结果给随后的指令使用。
34. 在复杂的pipeline control中，Issues很重要：
    a. 如果FPU或者memory单元不是pipelined，并且使用了多于一个cycle，会导致结构冲突。
    b. 写回状态由于不同功能单元的数据延迟导致结构冲突。
    c. OoO也有因不同功能单元的数据延迟导致的结构冲突。
    d. 如何处理异常？
35. W stage有一个真的很长的延迟，会把整个pipeline都拖慢。还有一些非一个cycle的操作，比如除法、缓存misses。如何防止增加写回时间导致减慢单个cycle的整数运算？BYPASSING！
36. 超标量pipeline。每个cycle Fetch两条指令，如果一个是整数操作一个是浮点数操作，那么同时dispatch它们。是一个不贵的方式来增加吞吐量。同样的方法可以用于拓宽issue，用增加功能单元来实现，但是regfile端口和bupassing的开销会大很多。

第六章，ILP和动态执行

1. 标题
2. ILP，即Instruction Level Parallelism。
3. 开发ILP的基础：标量流水线的局限性。指令并行度上限为1（IPC = 1）；为XX Hazard

引入的stalls。

4. 一些术语：
    a. Instruction parallelism 指令并行度，通知执行的指令数量。
    b. Operation Latemcy 操作延迟，可提供的（available）结果出现的时候。
    c. Peak IPC（Inst per clock） 每个cycle可并行执行的指令数。
5. 对于Scalar Pipeline来说：
    a. Instruction parallelism = D = 有多少种状态
    b. Operation Latency = 1
    c. Peak IPC = 1
6. Superscalar Pipelined：
    a. IP = D*N　N为同时几条流水数
    b. OL = 1 baseline cycle
    c. Peak IPC = N per baseline cycle
    d. 可以看出是提高了IP和IPC



7. 有序执行的真正问题：
    a. 引起指令依赖的距离成倍增长，CPI下降。
    b. Forwarding不再有效。
8. In-order在哪里丢掉了速度？给了个例子，RAW和ippeline hazard（就是同一个stage只有一个inst，structural hazard？）
9. In-order pipeline的问题：
    a. 有structural hazard。
    b. OoO就可以跳跃（pass）功能，消除structural hazard。
10. 新的pipeline定义：F,D,X,W（multi-cycle X includes M）。变量时延为：整数1cycle，FP 3-cycle。
11. ILP（inst-level parallelism）。Avg ILP = no.inst / no.cyc required。给了两个例子，有依赖的不能同时执行，没有依赖的可以。
12. Purported Limits on ILP。看不懂，好像和人名、年份有关。
13. ILP分析的scope。就是把一段代码相互依赖的分成一个scope，那么整段代码有几个scope就是ILP等于几。
14. 指令窗口尺寸的大小，与inst issues per cycle的关系。总的来说，指令窗口越小，每个cycle能issue的inst就越少。同时，它与编译器又有关系。
15. 动态调度，OoO X。
    a. 动态调度，完全是在硬件上，又叫做OoO X。
    b. 同时fetch多条指令到inst window。
        i. 使用分支预测

ii. 误预测的时候flush pipeline

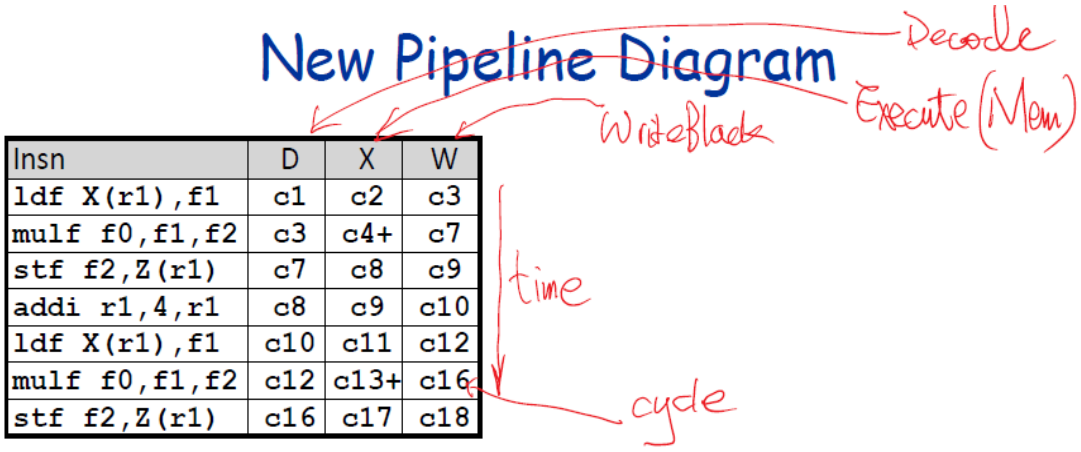
c. 对变量重命名，防止伪冲突（WAW，WAR）

d. 尽快地执行inst。寄存器依赖是已知的。内存依赖比较难处理。

e. 按序commit insts。如果commit之前发生了什么异常，直接flush。

f. 当前机器地调度窗口100以上。

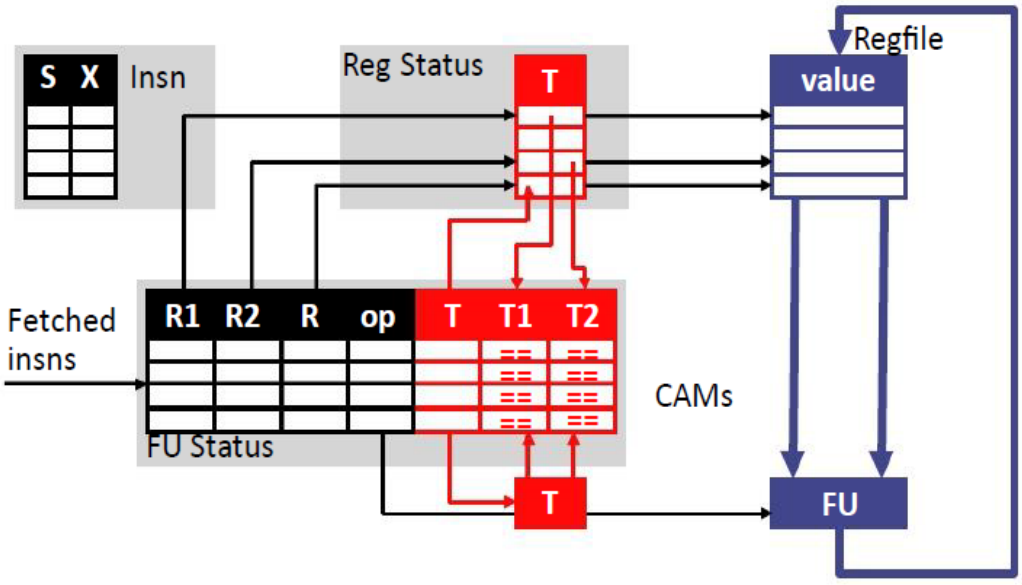
16. 动态调度的动机。
    a. 非顺序执行。减少RAW hazard；增加pipeline和功能单元的利用率；更多并行issue指令的机会。
    b. 但仍然保持顺序执行的语义！很重要，但难。
17. 动态执行的大图。在D步骤和S(issue)步骤（后文说是把D阶段分成两part，后一part被称为issue）之间有个inst buffer，当指令被fetch/decoded/renamed之后，就会被放到那个buffer里面，又被称为inst window。每个cycle，指令都会检查一下ready bits，如果ready就X。
18. 关于OoO的基础解剖。标题页。
19. 新pipeline的定义，和前面一样的。
20. 新pipeline的执行图。Fetch完马上Decode，所以二者合为一个D，X和M合并（新pipeline），W为最后一个阶段。时间是由上到下，其实是按照cycle的no.来决定。这种表示方式有利于OoO。



Insn	D	X	W
ldf X(r1),f1	c1	c2	c3
mulf f0,f1,f2	c3	c4+	c7
stf f2,Z(r1)	c7	c8	c9
addi r1,4,r1	c8	c9	c10
ldf X(r1),f1	c10	c11	c12
mulf f0,f1,f2	c12	c13+	c16
stf f2,Z(r1)	c16	c17	c18

21. OoO的buffer分析。Insn buffer（指令缓冲），一堆用于储存insns的锁存器（latches）。把D阶段分成了两个部分，前面fetch和decoded还是in-order的，但是后面的buffer把insns发送到后面的pipeline是OoO的。
22. 两个part的D。前面的part，称为D（Dispatch）。把指令放到insn buffer里，引起新的结构冲突（缓存满了），是in-order的过程。后面的part，称为S，send insn from insn buffer to execution units，老指令的等待不会让新指令也等待，不然就不叫OoO了对吧。
23. 一张图，Dispathch和Issue过程with浮点数处理部件。从图中可以看出，浮点数乘法、加法、除法所需要的cycle数分别为3、2、1。
24. 动态调度算法。
    a. Register scheduler，寄存器（依赖性）调度。
    b. 书中提到的两种Register scheduler：
        i. Scoreboard：没有寄存器重命名，被用在一些GPU里。
        ii. Tomasulo：有寄存器重命名，更加灵活，更好的表现。
    c. 在课堂中讨论的时候，为了简化内存调度，默认寄存器算法都知道内存依赖。
25. 关键的OoO设计特性：（Issue的政策和逻辑）
    a. 如果多个insn都ready，要选哪个？policy。最老的最先？（安全），最长时延的先？（有可能有更好的表现）

    b. 选择逻辑，实现issue policy。大多数项目使用random。
26. 所以！最重要的就是算法了。算法调度方法可以分为：没有寄存器重命名的Scoreboard；寄存器重命名来解决一些伪冲突；用tomasulo算法来实现OoO。处理包括：precise state and speculation，memory dependencies。
27. 第一个调度算法：Scoreboard。
    a. 集中控制方案：insn的状态被显式追踪。
    b. insn buffer：Functional Unit Status Table（FUST）
    c. 最先在CDC 6600中被实现。
    d. 我们的例子，简单的Scoreboard：
        i. 5 FU: 1 ALU, 1LOAD, 1 STORE, 2 FP(3-CYCLE PIPELINED)
28. Scoreboard的数据结构：
    a. FU Status Table（功能单元之状态表）
        i. FU, busy, op, R, R1, R2: 目的或源的寄存器名
        ii. T: destination reg tag (FU producing the value)
        iii. T1, T2: source reg tags (FU producing the values)
    b. Register Status Table（寄存器状态表）
        i. T: tag (FU that will write this register)
    c. Tags interpreted as ready-bits （TAG）
        i. Tag == 0 -> Value is ready in register file
        ii. Tag != 0 -> Value is not ready, will be supplied by T
    d. Insn status table （指令状态表）
        i. S,X bits for all active insns.
29. 图解上面的结构



Insn fields and status bits; Tags; Values.

30. 新的pipeline结构：F, D, S, X, W:
    a. F (Fetch) 和原来一样
    b. D (Dispatch) 有Structural/WAW hazard，stall：申请Scoreboard entry。
    c. S (issue)，RAW hazard? wait： 数据冲突，读。
    d. X (execute)，执行操作，做完了会通知scoreboard。
    e. W (writeback)，WAR hazard? wait：写寄存器，释放scoreboard entry。
        i. W 和 RAW-dependent S 可以在同个cycle
        ii. W 和 structural-dependent D 可以在同个cycle

31. Scoreboard Dispatch：Stall：WAW / structural (Scoreboard, FU) hazards
    a. 申请scoreboard entry
    b. 拷贝输入寄存器状态
    c. 设置输出寄存器状态
32. Scoreboard Issue：Wait：RAW
    a. 读取寄存器，可能要wait
33. Issue的政策和逻辑：和前面一样的介绍，政策是理论，逻辑是实现。
34. Scoreboard Execute：执行insn咯。
35. Scoreboard Writeback：Wait：WAR
    a. 把值写进regfile里，清除Reg Status entry。（wait）
    b. 与等待中的input tags进行比较，如果相同，清除哪个input tag（不太懂
    c. Free scoreboard entry

36 ~ 46. 过一遍scoreboard的流程，以下是一些注意的点：

- 尽量贪心地去用cycle，在每个阶段有不同的注意事项，会导致stall或者wait的情况不同。
- 一定要注意，保持in-order执行的语义，不能代码结果乱了。
- 没有哪个阶段结束后一定能进入下一阶段（好像S阶段可以...），都会有被wait的可能。
- 在D阶段，考虑两种，WAW引起的，可以看要被写的寄存器的status是否为空。Structure引起的，scoreboard没有空位来存，也就是FU被占用了。
- 在S阶段，注意它的含义，issue，是从buffer里把指令取出来去执行。这一步会对source寄存器进行读取，因此会有RAW hazard。
- 在X阶段，似乎没有什么原因能被wait，但是不同的Float FU执行的cycle数不同。乘法3，减法2，加法1。
- 在W阶段，要把东西写回寄存器，如果是写回内存的，不用等待。如果是写回寄存器的，有WAR hazard。
- W阶段，不仅是写回寄存器，还要free掉reg status和FU status。并且，在同一个cycle，可以立马执行刚被释放而无依赖的insn。
- 例如stf f2, Z(r1)指令，其实f2和r1都是源操作寄存器，目的寄存器没有，而是写入内存。

47.Scoreboard真的有快很多很多吗？？？并没有。因为一个很奇怪的WAR hazard（具体问题具体分析）

48.Cache Miss会导致更多的时延。略过。

49.Scoreboard Redux。

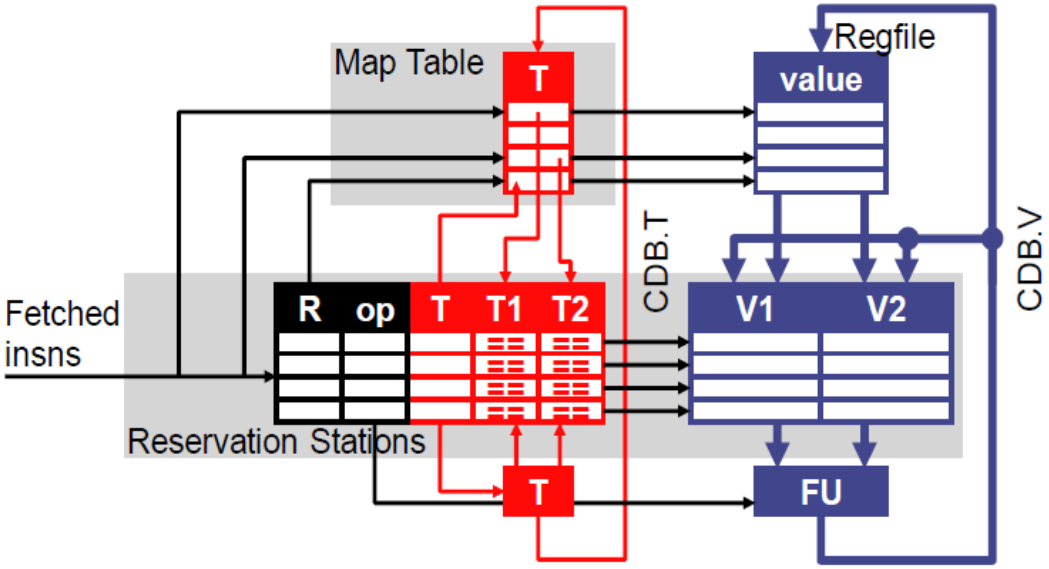
- 好处有，比较简单的、便宜的硬件。比较好的表现。
- 不是很好的地方有：
    - 没有Bypassing，可能是一个基础的问题。
    - 被限制的调度域，Structural / WAW hazards delay dispatch。
    - 拖慢issue的真・时延（RAW），WAR时延拖慢writeback。
    - 用硬件寄存器重命名来解决！

第7章，Tomasulo implementation of OoO execution

1. 标题。
2. 大纲。通过寄存器重命名来解决伪依赖；Tomasulo算法的实现。控制precise state和

speculation。控制内存依赖。

3. 标题，通过寄存器重命名来解决伪依赖。
4. 真依赖的例子。RAW是真的依赖，必须按序执行。我觉得，所谓的真依赖和伪依赖，其实是看代码语义上是否依赖，而不是硬件或者结构上的依赖。
5. 伪依赖 / 名字依赖。WAW依赖，其实是可以一起执行的（？）。其实WAW依赖是在D阶段出现，而在W阶段才会真正影响结果，所以其实是可以一起在D阶段dispatch（？）。WAR依赖，一个重命名就行了。
6. 一个例子，只有RAW是真依赖。
7. 一个更长的例子，还是只有RAW是真依赖。WAW和WAR都是伪依赖。
8. 上面那个例子，两段明显无关的代码（可能是循环展开），不应该隔着那么远去依赖去wait。还是一句话，改名就行。
9. 改名的规则。从第一个写开始到下一次写之前，这一段的同一个寄存器要用同样的名字，此外其他的名字可以改。
10. 改名后的代码，可以看出已经分成两个不依赖的部分了。
11. 寄存器改名的方法。
    - 每次 architected register 被写的适合，我们给它分配一个 physical register。
    - 直到下一次这个architeched register被写之前，都是指向这个物理寄存器。
    - 对于RAW依赖不做修改。
    - 一些很简单的例子。把所有的寄存器都给改名了，全部从r变成p，有一个映射表专门用来映射，还有一个freelist，是architected register的。
12. 又是一个例子。全部被写的寄存器都改名了。
13. Tomasulo 算法的标题。
14. Tomasulo算法包括：
    a. Reservation stations（RS）：instruction buffer 指令保留站
    b. Common data bus（CDB）：broadcasts results to RS 通用数据总线
    c. Register renaming：removes WAR/WAW hazards 消除假依赖
    d. 在IBM 360/91 中首次实现。只给FP units进行调度，有bypassing。
    e. 我们课件中讨论简化版的，给所有步骤进行调度，没有bypassing，5个功能单元。
15. Tomasulo的数据结构：
    a. Reservation Stations(RS#) 其实和scoreboard的那个表很像。
        i. FU, busy?, op, R: destination register name
        ii. T: destination register tag (RS# of this RS)
        iii. T1, T2: source register tags (RS# of RS that will produce value)
        iv. V1, V2: source register values
    b. Rename Table/ Map Table/ RAT 重命名表。
        i. T: tag (RS#) that will write this register
    c. Common Data Bus (CDB)
        i. Broadcasts <RS#, value> of completed insns指令完成后广播RS#和值
    d. Tags interpreted as ready-bits ++ （和scoreboard一样，可以解释）
        i. T == 0 value ready
        ii. T != 0 value 还没好，等CDB广播
16. 图解上面的结构。

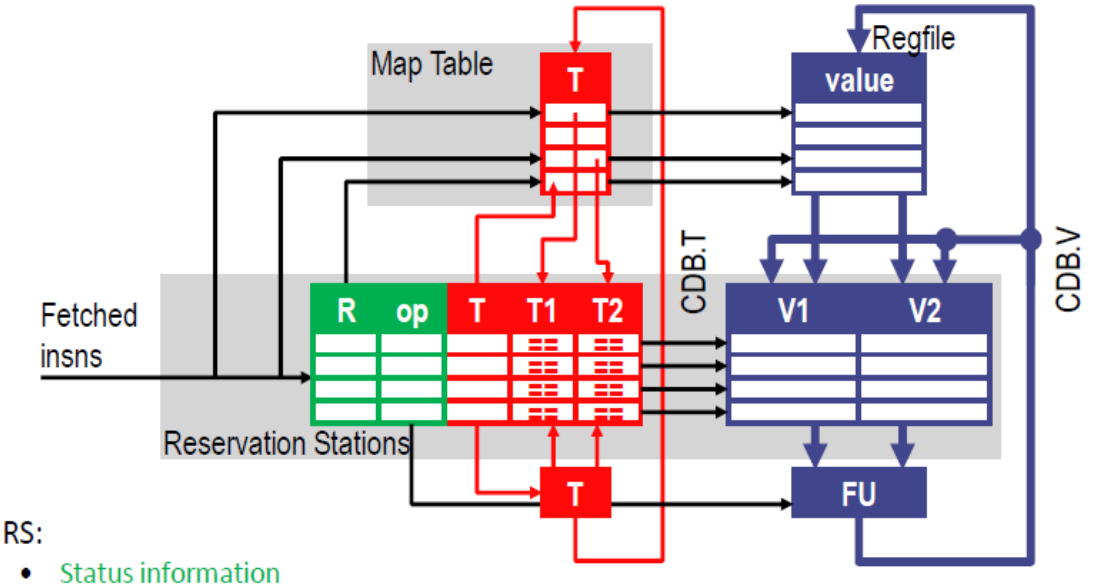


- Insn fields and status bits
- Tags
- Values

数据结构

17. Tomasulo pipeline:
    a. D(dispatch), structural hazard only.
    b. S(issue), RAW hazard will wait for CDB broadcast.
    c. W(writeback), free RS entry. 其他可以同时操作的stage和scoreboard一样。
18. Tomasulo Dispatch
    a. 申请RS entry
    b. 输入的reg是否ready? 读取value到RS: 读取tag到RS
    c. Rename output register to RS #
19. Tomasulo Issue
    a. Wait for RAW hazards
20. Tomasulo Execute
21. Tomasulo Writeback
    a. Wait for structural CDB hazards
    b. if Map Table rename still matches? Clear mapping, write result to regfile
    c. CDB broadcast to RS: tag match? clear tag, copy value
    d. Free RS entry
22. 如何实现register renaming?
    a. Value copies in RS(V1, V2)
    b. insn store correct input values in its own RS entry
    c. Future insns can overwrite master copy in regfile, doesn't matter
23. Tomasulo 的寄存器重命名有被称为value-based或者copy-based。
    a. 重命名的名字，被称为体系结构寄存器。
    b. 储存的位置有register file和RS，一个值可以同时存在于两个地方，但是regfile保存着master values。
    c. RS的拷贝消除了WAR hazard。
    d. 储存位置是用RS# tags来指向。Reg table把名字翻译成tags。tag为零，说明值在regfile里，不为零，说明还没算出来。
    e. CDB广播<tag, value>对。

24. 一张详细的图解。



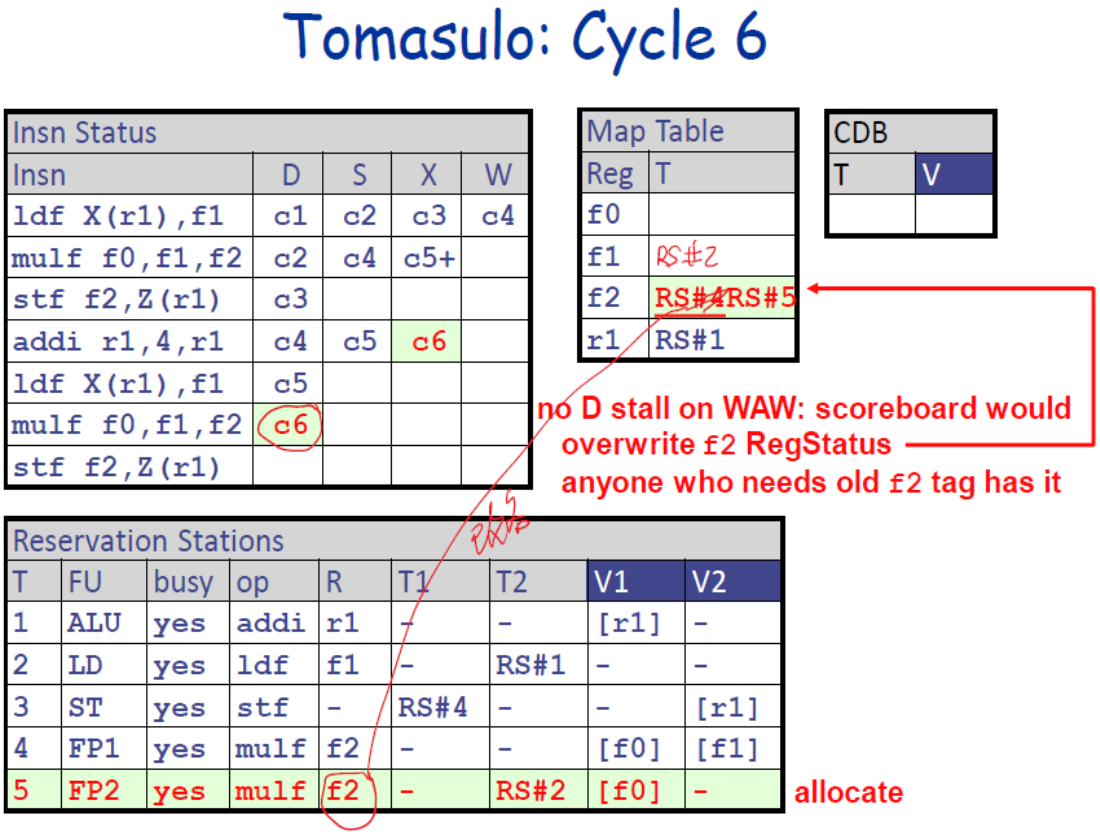
RS:

- Status information
  - R: Destination Register
  - op: Opcode (add, etc.)
- Tags
  - T1, T2: source operand tags
- Values
  - V1, V2: source operand values

- Map table (also RAT: Register Alias Table)
  - Maps registers to tags
- Regfile (also ARF: Architected Register File)
  - Holds value of register if no value in RS

一些不同，T1，T2不再直接装值，装的是tags。

25. 开始例子：
    - RS#的含义：例如RS #2，表示RS的T==2的行，它是一个tag。
    - CDB每次只广播一个值，所有拥有同个tag的RS都会得到那个值。
    - D阶段只要判断被写的寄存器有没有tag，就知道它之前有没有被写。但是！！！不用stall for WAW，只需要重新给一个重命名在Map Table里，而且！已有的RS里的tag不用改。到了原来的tag值被计算出来广播的时候，匹配的RS会得到值，不匹配的Map Table不会被清空。



- WAR不用等待，因为需要Read的值在D阶段就装入到RS的V里了。

36.能否加上bypassing？可以的，但是会很复杂。调度必须在计算之前开始；需要知道指令的时延，很难；准确的bypass在随后十年是很前沿的技术。

37.加上一个超标量。设超标量宽度为N，窗口大小（RS大小）为W。

What do we need for an N-by-W Tomasulo?

- RS: N tag/value w-ports (D), N value r-ports (S), 2N tag CAMs (W)
- Select logic: W→N priority encoder (S)
- MT: 2N r-ports (D), N w-ports (D)
- RF: 2N r-ports (D), N w-ports (W)
- CDB: N (W)
- Which are the expensive pieces?

38.超标量选择的逻辑。加了超标量的结构比较复杂，可以选择不同的RS设计：

split design：拆！

FIFO design

39.动态调度的总结：

- 动态调度就是为了OoO。
  - 高利用率的pipeline/FU
  - 用硬件实现比让进实现要简单而且高效
- Insn buffer：多个 F/D 锁存器
- 两个动态调度算法，没有renaming的scoreboard，有重命名的copy-based的Tomasulo。

40.收工了吗？没有，没有指令可选的时候，可能会出错，所以需要一个很好的分支预测（？）；异常！再也分辨不出RS里的相关顺序。

41.Issue可以看成两个步骤：wakeup和select。RS发现它自己可以执行了，是wakeup。

RS被告知有相关的计算单元，是select。

第8章，Vector Supercomputers

1. 标题，向量计算机。
2. 超级计算机的应用领域。一堆一堆，大数据集计算，CDC6600，70s-80s 超级计算机等于向量计算机。
3. 向量超级计算机的重要代表，Cray-1，1976。Scalar Unit（标量单元，存取结构），Vector Extension（向量扩展，向量的寄存器和向量指令）。实现是由硬件控制、高级流水线功能单元，交织（interleaved）内存系统，没有数据缓存，没有虚拟内存。
4. 向量编程模型（ISA）。有专门的向量寄存器（一行好多个的），有专门的向量加法指令，向量装载指令。
5. 向量代码的例子。要注意setvlr x4，是设置向量的长度，这样装载和加法指令都知道该加多长了。
6. Cray-1是向量计算机最经典的代表。
7. 向量指令集的好处
    a. 紧凑
    b. 表达能力强，一条指令就可以告诉机器做一堆事情
    c. 可扩展性强，更多的流水通道
8. 向量体系结构的流水执行。
    a. 深度流水，多stages，fast clock
    b. 简化控制
9. 多流水的好处图，同样的时间可以跑更多的运算。
10. Interleaved（交织的）向量内存系统，一个内存基地址加偏移量，加上一开始设置的向量长度，就可以同时存取多个值。
11. 向量单元的结构。

$$\text{Speedup}_{\text{overall}} = \frac{1}{\left(1 - \text{Fraction}_{\text{parallel}}\right) + \frac{\text{Fraction}_{\text{parallel}}}{\text{Speedup}_{\text{parallel}}}}$$

总加速度