Chapter 04 处理器体系架构

- (P249) CISC和RISC的区别

Y86描述:

- **寄存器**共15个(相比x86少了 `%r15`), 分别编码为 `0x0` ~ `0xE` , 而 `0xF` 表示不访问任何寄存器.
  从而一个寄存器需要4bit来编码.
  并且约定, 如果一个指令只需要一个内存引用(irmovq, pushq, popq), 则另一个寄存器指示符设为 `0xF` .
- **内存引用**仅允许 Imm(%reg) 这种类型. 从而一个内存引用需要 4bit+8Byte 来编码.
  由于立即数也需要8Byte, 简便起见, **不允许** `内存引用之间` 以及 `立即数到内存` 的数据传输指令.
- **指令类型**用第1个字节标识,
  其中高4位为 `类型码` , 同时**明确了这条指令的总长度**;
  低4位为 `功能码` , 用于区分同一大类下的不同指令.
  寄存器类型用第2个字节标识(如果指令类型需要寄存器的话),
  其中前4位表示 `rA` , 后4位表示 `rB` .
- `rrmovq` 可以看做是"无条件的" `cmov` ,
  `jmp` 可以看做是"无条件的" `jCC` .
  它们的 `功能部分` 都编码为 `0x0` .
- 在Y86中, `jCC` 和 `callq` 的目的地址 `Dest` 只能是用立即数表示的 `直接寻址` , 而且是 `绝对寻址` . (而x86是PC相对寻址)

  > PC相对寻址使得代码更换在内存中的位置时, 不需要更改所有的Dest编码, 更为优雅. 然而Y86中简便起见, 采用绝对寻址.

- 指令中的整数采用小端法编码.
- 只要能明确序列的起始字节, 则字节编码具有唯一的解释. (当然, 如果不知道起始位置, 就不太容易解释了...)
  这是因为每条指令的第一个字节给出了指令类型, 从而整条指令的长度, 以及每个bit的含义都是明确的!
- **异常**: `stat` 是程序员可见状态. 包含以下4种可能:
  - **`AOK`** 1 (AllOK)唯一表示正常操作. 其余情况都代表某种异常, 并会使得Y86处理器停止.
  - **`HLT`** 2 (HaLT) 遇到器执行 `halt` 指令.
  - **`ADR`** 3 (ADdRess) 遇到非法地址, 比如超过了最大地址值, 或取指令/读写内存时使用了非法的地址.
  - **`INS`** 4 (INStruction)遇到非法的指令代码.

- **HCL** (Hardware Control Language): 硬件控制语言
  类似C语言的语法, 但既然HCL是"描述语言", 因此就不像C是"顺序执行"的, 而是"同时起作用"的.
  `bool` 布尔(bit)类型, 变量名建议小写, 如 a, b, c

`word` 整数(word)类型, 变量名建议大写, 如 A, B, C

> 简便起见, 所有字级信号(无论4bit的寄存器还是64bit的整数)都声明为 `word` , 而不指定大小.

- **HDL (Hardware Description Language) 硬件描述语言**, 最常用的语言是Verilog.
- 逻辑合成(Logic Synthesis)程序可以根据HDL描述, 产生物理电路的设计.
  而利用一些工具, 可以将HCL翻译成HDL, 进而最终产生实际的处理器.
  所以**我们将只专注于**使用HCL语言表达硬件设计的**控制**部分.
- **逻辑门**总是**活动的(active)**, 即一旦输入发生变化, 输出会**很快地**响应输入, 产生变化.
- 逻辑门被视作二元运算符, 但常见的是具有n路输入(n>2)的情形.
- 将多个逻辑门组合起来, 可以构建**组合电路(Combinational Circuit)**.
  注意这个电路网络必须是无环的.
- **多路复用器(Multiplexor)**
- 画字级电路时, `实线` 表示携带字的线路, `虚线` 表示布尔信号线路.
- **ALU**(算术/逻辑单元), 包括3个输入: 2个 `数据` 输入和1个 `控制` 输入(0/1/2/3, 分别对应于指令的功能码)

- 由于组合电路本身不存储任何信息, 为了产生**时序电路(Sequential Circuit)**, 需要引入存储信息的相关设备:
  - **时钟**是一个周期性信号, 决定何时把新值加载到设备中.
  - **时钟寄存器**: 存储单个bit或字, `时钟信号` 决定 `时钟寄存器` 何时加载输入值.
    具体地, 假设寄存器当前状态为x, 此时产生了一个新的输入y .
    只要时钟还是低电位的, 寄存器的状态就不会变化;
    而当时钟信号变成高电位时, 输入信号就会加载到寄存器中, 成为寄存器的新的状态, 直到下一个时钟上升沿都不再改变.
  - **随机访问存储器**: 包括 `虚拟内存系统` 和 `寄存器文件` .
- 典型的寄存器文件有2个读端口和1个写端口.
  虽然寄存器文件不是组合电路, 因为它有内部存储,
  但是从寄存器文件读数据, 就好像寄存器文件是一个以地址为输入, 以数据为输出的组合逻辑块.
- 向寄存器文件写入字是由时钟信号控制的, 每次时钟上升时, 输入(到写端口)的 `valW` 的值会被写入到 `dstW` 所指示的 `程序寄存器` . 如果 `dstW` 是 `0xF` , 则表示不写入任何程序寄存器.
- **SEQ(Sequential, 顺序的)**的6个基本阶段(通用框架).
  1. **取指(Fetch)**:
     1. 以PC值为地址, 从内存中读取指令字节:
        - 第一个字节的高低4位分别称为 `icode` 和 `ifun` .
        - 还可能取出一个寄存器指示符字节, 指明一个或两个寄存器指示符 `rA` `rB` .
        - 还可能取出一个8字节常数字 **`valC`** . (C for Constant)
     2. 计算下一条指令的地址 `valP` . (P for PC)

     > 由于在一开始的取指阶段就计算了 valP , 所以指令访问到的PC是更新后的下一条指令的地址 `valP` .

2. **译码(Decode)**: 从寄存器文件读入最多两个操作数 `valA` `valB` .
   即把 `作为输入的寄存器的代号` Decode成实际的操作数.
   注意: 涉及寄存器的指令并非一定译码, 比如 `irmovq` 中的 `rA` 仅用于指明写寄存器的目的位置, 所以译码阶段什么都不做.
3. **执行(Execute)**: ALU执行 `ifun` 指明的操作, 得到的结果记为 `valE` (Execute) , 并可能设置条件码 / 检查条件码和跳转条件.
4. **访存(Memory)**: 将数据写入内存, 或者从内存读出数据 `valM` .
5. **写回(Write Back)**: 最多写入两个结果到寄存器文件.(比如 `popq %rxx` 指令分别写入 `%rsp` 和 `%rxx`)
6. **更新PC(PC Update)**: 将PC设置为下一条指令的地址.

> **通过这个框架, 可以将 `所有的` Y86-64指令统一地组织成6个阶段.**

- 注意 `访存(Memory)` 是在 `写回(Register)` 之前的.
  原因: 考虑 `mrmovq` 指令, 显然得先访存, 才知道写到寄存器的值应该是什么.
- 由于**Decode**阶段确定了操作数是什么, 所以 `pushq %rsp` 的行为也就不难理解了.
  即当然会把 `%rsp` 的旧值 push 到栈顶.
  而 `栈指针-=8` 的操作则是通过**Execute**阶段中ALU的减法和**WriteBack**阶段对寄存器 `%rsp` 的写完成的.
- 依照Y86(以及x86)惯例, **向内存写**的地址必须是 `valE` , 即ALU的输出值.

  > 这也是合理的, 因为写内存的指令 `pushq` , `rmmovq` 都需要ALU参与计算内存地址(前者是计算rsp-8, 后者是计算D和rB相加)

- 在 `popq` 的阶段中, 在译码阶段读取了两次 `%rsp` 的值.
  看似多余, 不过可以让流程和其它指令更相似.
- `popq` 和 `pushq` 涉及的栈指针±8 应该理解为 `%rsp := (%rsp + 8)` , 而非 `%rsp ±= 8` ,
  而 (%rsp+8)的计算由**Execute**阶段完成, 所以**写回**时是将 `valE` 写入 `%rsp` .

- 读操作沿着相应的单元传播, 就好像它们是"组合逻辑";
  所以由于指令内存只用于"读指令", 可以把这个单元看作"组合逻辑".
- 对 `硬件寄存器` 的写操作是由时钟控制的, 在上升沿时执行写入.

流水线化(Pipeline)

- 流水线提高系统的**吞吐量(throughput)**; 但轻微地增加了**延迟(latency)**, 因为 `流水线寄存器` 有额外的时间开销.
- 以**GIPS(Giga Instructions Per Second)** , 即**十亿(10^9)条指令每秒**为单位描述系统的吞吐量.
- 减缓时钟不会影像流水线的行为; 但加快则可能导致寄存器的输入还是非法的时候就被传递到了输出, 造成错误.
- 时钟的速率是受流水线中**最慢的阶段**的延迟限制的.
- **数据相关(Data Dependency)**

控制相关(Control Dependency)

- **SEQ+: 重新安排计算阶段**:
  移动PC阶段, 使得这一阶段在一个时钟周期开始时执行(而非结束时). 它计算**当前指令**的PC值.
  为此, 我们需要创建状态寄存器, 来保存一条指令执行过程中计算出来的信号.
  从而当一个新的时钟周期开始时, 这些存储着的信号会被用于计算当前指令的PC.
  而且可以注意到, 相比于SEQ, SEQ+不再真正拥有一个硬件的 `PC寄存器` 保存PC值,
  而是根据上一条指令保存的一些状态信息动态地计算新的PC.
- `D_stat` `E_stat` `M_stat` `W_stat` 这种大写前缀分别表示 `流水线寄存器D/E/M/W` 中存储的 **`状态码字段`**;
  而 `f_stat` `m_stat` 这种小写前缀则表示相应阶段内, 由 `控制逻辑块` 产生的 **`状态信号`**.
- 在 **`PIPE-`** 的设计中, 加入了 `SEQ` 和 `SEQ+` 中不存在的新硬件 `SelectA`,
  它的作用是从 `valP` 和 `从寄存器文件A端口读出来的值` 中选一个, 作为流水线寄存器E中的 `valA`.
  这可以减少了流水线寄存器的状态数量(因为把信号 `valP` 去掉了, 合并到 `valA` 去了), 而且**不会影响指令执行**.
  这是因为我们发现:
  只有 `callq` 在访存(M)阶段需要 `valP` 的值,
  只有 `jCC` 在不跳转的情况下需要在执行阶段使用 `valP` 的值.
  而这两类指令又都不会使用从寄存器文件中读出的值.

- **预测下一个PC:**
  除了 `jCC` 和 `ret` 之外, 根据**Fetch**阶段计算的信息, 我们就足以确定下一条指令的地址.
  具体地:
  - 对于 `call` 和 `jmp`, 下一条指令的地址是指令中的常数字段 `valC`
  - 对于 `jCC指令`, 我们可以预测**选择**分支(对应于新PC值为 `valC`); 也可以预测**不选择**分支(对应于新PC值为 `valP`)
    简便起见, 我们采用的策略是: "总预测**选择**分支(对应于新PC值为 `valC`)".
    不过无论预测何种情况, 我们都必须能够处理预测错误的情况, 因为此时已经取出并部分执行了错误的指令.
  - 对于 `ret` 指令, 由于返回地址的可能性是近乎无限的, 我们不去试图预测返回地址, 只是单纯地暂停处理新指令, 直到ret指令到达了**WriteBack**阶段, 此时 `W_valM` 即存储了在上个阶段(**Memory**阶段)从 `内存栈` 中读取到的返回地址.
  - 对于除了上述四种之外的其它指令, 下一条指令的地址就是计算出的 `valP`
- **`Predict`** 组合逻辑会从 `valC` 和 `valP` 中选择一个作为 `预测值`, 存储在 `流水线寄存器F` 的 `predPC` 中.

  > 其实 `流水线寄存器F` 只有 `predPC` 这一个值

- **`SelectPC`** 逻辑会从:
  1. `F_predPC` (对于大多数指令) (存储着上一条指令的 `Predict` 逻辑块计算出的预测值)
  2. `M_valA` (对于不跳转的jCC指令) (回忆这里的 `valA` 是被 `Select_A` 逻辑处理前的 `valP`)

3. `W_valM` (对于ret指令) (这是ret指令经过访存阶段得到的valM)

中选择一个作为这一个时钟周期要取的指令的指令内存地址.

- **流水线冒险(Hazard)**的类型:

  当一条指令要更新后面的指令将要读到的**程序状态**时, 就可能出现 **`冒险`** .

  对于Y86-64, 所谓**程序状态**, 包括: `Register` , `PC` , `Memory` , `CC` , `stat` .

  下面分别讨论这几种程序状态对应的冒险情况:

  - **`Register`** : 由于寄存器的读写在不同的阶段(分别在**Decode**和**WriteBack**), 所以先执行的指令可能还没有**Writeback**, 紧随的指令就在**Decode**阶段需要相应的寄存器. 具体地, 由于**Writeback**在**Decode**后的3个阶段, 所以一条指令的操作数被其前三条指令中的任何一个修改, 都会造成数据冒险.
  - **`PC`** : 当分支预测错误时的处理, 以及对 `ret` 指令的特殊处理.
  - **`Memory`** : 对**数据内存**而言, 由于读和写都是在访存阶段, 所以当一条读内存的指令在到达访存阶段时, 任何之前写入内存的指令也已经完成了! 所以不涉及冒险; 对于**指令内存**而言, 之前的指令的确有可能修改指令内存, 即出现程序代码的"自我修改", 导致当前取出并执行的指令是错误的, 从而导致冒险. 在Y86-64中, 我们只是简单地**假设程序不能修改自身**, 从而不考虑这种冒险.(有些系统有复杂的机制来处理这种冒险)
  - **`CC寄存器`** : `条件传送(cmov)` 会在**Execute**阶段读 `CC寄存器` ; `条件跳转(jCC)` 会在**Memory**阶段读 `CC寄存器` (`SelectPC` 用来判断是否用 `M_valA`).

    由于设置条件码只会发生在 `OPq` 指令的Execute阶段.

    从而"先执行的指令的setCC"一定早于"之后指令的getCC",

    故不会发生冒险.
  - **`stat`** : 流水线中的每条指令都有一个对应的状态码. 当异常发生时, 处理器需要有条理地停止.

- 用**暂停(Stalling)**避免数据冒险:

  1. 将指令阻塞在**Fetch**阶段的方法: 将PC保持不变.
  2. 将指令阻塞在**Decode**阶段的方法: 在**Execute**阶段插入一个"气泡".

     "气泡"类似于一个自动产生的 `nop` 指令, 它不会改变Register/Memory/CC/stat.

     它在这里的作用是代替本该从**Decode**进入**Execute**的指令进入**Execute**.

  > 用暂停避免冒险的效率较低, 因为数据冒险是很常见的情况, 这种插入气泡的方法严重**降低了吞吐量**.

  > Q: `暂停控制逻辑` 如何知道是否发生了数据冒险(从而需要暂停)?
  >
  > cpyA: 检查 `流水线寄存器E/M/W` 的 `dstE` 和 `dstM` , 看是否和 `srcA` 或 `srcB` 中的某个相同. (后面HCL实现处会有专门讨论)

- 用**转发(Forwarding)**来避免数据冒险:

  所谓 **`转发`** , 即在流水线中, 将靠后阶段产生的结果值传到较早阶段.

1. 将**来自ALU**的, 且目标为 `寄存器写端口E` 的值, 转发给**Decode**阶段(的valA valB):
   - 把即将在 **Write_back** 阶段写入的值 `W_valE` 转发给Decode阶段;
   - 把正处于 **Memory** 阶段, (虽然在这一阶段啥都不做, 但)一会要在**Write_back**阶段写入的 `M_valE` 转发给**Decode**阶段;
   - 把正在执行 **Execute** 阶段的ALU的输出值 `e_valE` 转发给**Decode**阶段

     > 这一条看似会有时序问题, 实则不然. 因为**Decode**阶段只需要在**时钟结束前**产生信号 `valA` 和 `valB` 即可(这样在下一个周期开始时, `流水线寄存器E` 就能装载来自**Decode**阶段的值了), 而在此之前ALU的输出 `e_valE` 已经是合法的了!
     > 同理, `m_valM` 也是**Memory**阶段合法的可用值.

2. 将**从内存读出**的, 且目标为 `寄存器写端口W` 的值, 转发到**Decode**阶段:
   - 在**Memory**阶段, 转发 `m_valM`
   - 在**Write_back**阶段, 转发 `W_valM`

> **小结一下:**
> `转发源` 共5个,分别在**Execute/Memory/Writeback**阶段: `e_valE` `m_valM` `M_valE` `W_valM` `W_valE`
> `转发目的` 共2个, 全都在**Decode**阶段: `valA` `valB`
> (其实, 在后面对 `ret` 和 `分支预测错误` 的处理中, 也涉及类似"转发"的东西, 即 `M_valA` 转发到**Fetch**阶段用于处理 `分支预测错误` , 以及 `W_valM` 转发到**Fetch**阶段用于处理 `ret`)

- 基于转发, 我们从 `PIPE-` 升级到 `PIPE` :
  - **逻辑**上, 译码逻辑能够确定是使用来自寄存器文件的值, 还是转发过来的值.
    这仅需要按照一定的优先级, 比对 `流水线寄存器E/M/W` 的 `dstE/dstM` 与 **Decode** 阶段的 `srcA/srcB` 即可.
  - **硬件**上, 多出了从5个转发源到**Decode**阶段的信号反馈线路,
    同时, 新增了 `Sel+Fwd A` 和 `Fwd B` 两个逻辑块. 其中:
    - `Sel+FwdA` 是 `Select A` 的升级版, 负责从 **{** `valP` **,** `R[rA]` **,** `5个转发值` **}** 中选一个作为 `valA` ;
    - `FwdB` 负责从 `R[rB]` 和 `转发值` 中选一个作为 `valB` .

> 但转发不能处理一切数据冒险!
>
> 有一类数据冒险不能单纯靠转发来解决.
>
> 即当涉及**从内存读到寄存器**的数据冒险时,
>
> 由于内存读操作在流水线较晚的阶段. 直接转发是来不及的.
>
> 我们需要⤵️

- 用**加载互锁(Load Interlock)**避免**加载/使用(Load/Use)数据冒险**:
  当指令A需要**使用(Use)**上一条指令B从内存中**读取(Load)**到某个寄存器的值时,
  转发是来不及的, 因为这个值最早也要到指令B进入**Memory**阶段才能拿到, 然而此时指令B还在**Execute**阶段.(差了1个周期)
  所以不得不插入一个 `nop bubble` , 让**Decode**阶段先暂停一个周期, 这样在下一个周期就可以顺利从**Memory**阶段转发 `m_valM` 了.

- **避免控制冒险**: 控制冒险只会发生在 `ret` 和 `跳转指令` .
  - 对于 `ret` , 暂停3个周期(插入3个bubble),
    直到ret从**Memory**阶段拿到地址后, 并进入**Writeback**阶段,
    此时**Fetch**阶段可以得到来自**Writeback**阶段的转发值 `W_valM` , 从而让下一条指令开始取指.
  - 对于 `jCC` , 只有发现预测错误(在我们的设计中, 即发现跳转条件实际不满足) 时才会涉及冒险.
    具体地, 当分支逻辑在**Execute**阶段发现不应该选择分支时, 已经错误地取出了两条指令.
    不过好在这两条指令分别在**Fetch**和**Decode**阶段, 所以并没有改变任何程序员可见状态.
    所以我们只需在下一个周期向**Decode**和**Execute**阶段插入Bubble, 然后设置PC为 `jCC` 后面的那条指令,
    就完成了对预测错误的指令的**指令排除(Instruction Squashing)** .

- **异常处理(Exception Handling)**
  三个细节问题:
  1. 如果流水线中同时发生多个异常, 应该向操作系统报告哪一个?
  2. 如果某条指令导致了异常, 然而后来发现这个指令本不该执行(即因为分支预测错误而被取消), 如何处理?
  3. 一条指令在产生异常时, 它后面的指令或许已经在之前的阶段改变了某个程序员可见状态. 这违背了"异常之后的指令不改变程序员可见状态"的ISA模型. 该怎么办?

  能解决上述问题的异常处理的实现:
  1. 当流水线的一个或者多个阶段出现异常时, 异常信息存放在流水线寄存器的状态字段中, 此时并不终止程序;
     而当异常指令到达**Memory**阶段时, (注意到此时异常指令不可能是要被取消的了), 禁止处于流水线更早阶段的指令更新任何程序员可见状态(CC/Mem) (这里不必有Register, 因为异常指令执行到**Writeback**阶段时, 程序就终止了(见下一条), 后面的指令更没有机会到达**Writeback**阶段修改Register).
  2. 当异常指令到达最后的流水线阶段(即**Writeback**阶段)时, 程序停止执行, 并向操作系统报告此时 `流水线寄存器 W` 中的异常stat. (巧妙地解决了"报告优先级"的问题.)
  3. 如果取出的某条指令被取消, 则关于这条指令的异常状态信息 `stat` 也要被取消.

- **PIPE各个阶段的实现**
  - **Fetch**:
    1. `Select PC` 逻辑计算当前指令的PC值, 产生信号 `f_pc` :

需要从 `M_valA` (对应 `jCC的分支预测错误`) 或 `W_valM` (对应 `ret`) 或 `F_predPC` (对应其余情况)三者中进行选择.

2. **`Predict PC`** 逻辑预测下一个PC值, 产生信号 `f_predPC` :
   需要从 `f_valC` (对应 `jCC/jmp` 和 `call`) 或 `f_valP` (对应其余指令) 二者中进行选择.
3. **`stat`** 逻辑产生信号 `f_stat` , 可能是 `SAOK` , 也可能是异常信号(`SADR` 或 `SINS` 或 `SHLT` , 并且注意:
   无论逻辑上还是规则上, 这三种异常的优先级都是递减的. 而且stat信号只能是4种中的一种, 而不能是某几种的组合!).

- **Decode** & **Write Back**:
  1. **`Sel+Fwd A`** 逻辑产生信号 `d_valA` ,
     它可能来自:
     (1) `jCC` 或 `call` 指令合并过来的 `valP`
     (2)转发过来的值
     (3)直接从寄存器读出来的值 `d_rvalA` (注意与信号 `d_valA` 的区别, 多出来的 `r` 表示 `Register`) .
     由于同一时刻可能有多个转发源, 所以我们要注意选择转发源的优先级:
     e_valE > m_valM > M_valE > W_valM > W_valE .
     其基本原则是"越晚进入流水线的优先级越高",
     不过需要特别强调的是 **m_valM > M_valE** 和 **W_valM > W_valE** 这两个优先级顺序不可颠倒,
     因为虽然在时间上, 两边都是是来自同一流水线阶段,
     不过如果把顺序颠倒, 我们可以构造出违反ISA的反例:
     前者: `popq %rsp; rrmov %rsp %rax` (注意这里涉及Load/Use冒险, 所以在两条指令之间, 流水线会暂停一个周期!)
     后者: `popq %rsp; nop; nop; rrmov %rsp %rax`

     > 构造的思路是考虑到只有 `popq` 指令会同时写入(改变)两个寄存器, 所以只有 `popq` 会关心(**Memory** & **Writeback** 阶段的) `valE` 和 `valM` 的转发顺序. 为了符合

  2. **`Fwd B`** 逻辑同理, 产生信号 `d_valB`
  3. **`Stat`** 逻辑计算 `w_stat` 信号: 如果发现 **WriteBack** 是气泡, 产生 `AOK` 的状态值; 否则沿用之前传进来的状态值 `W_stat` .

- **Execute**:
  1. **`SetCC`** 逻辑以 `W_stat` 和 `m_stat` 为输入, 产生一个信号, 表明是否应该更新 `CC寄存器` .
     这两个输入是意在检查此时的**Memory**或**Writeback**阶段中是否是一个产生异常的指令.
     因为如果是的话, 此时**Execute**阶段的CC更新是应该被禁止的.
- **Memory**:
  1. **`Stat`** 逻辑计算 `m_stat` 信号: 如果 `内存` 输出信号 `dmem_error` 为1, 则 `m_stat` 设置为 `SADR` .

> 这里有疑惑: 如果在之前的阶段已经产生异常 `S某某` 了, 最终汇报的stat会不会被错误地修改成了 `SADR` 呢?

- **特殊控制逻辑**

  > Q: 我们还差什么工作?
  > A: 我们几乎快完成了PIPE的设计, 不过以下4种的情况必须依靠 `特殊控制逻辑` 才能处理,即需要对 `流水线寄存器` 进行特殊控制.

  1. **加载/使用冒险**: 在 `popq` 和 `mrmovq` 指令和紧随其后的读相应寄存器的指令之间, 流水线必须暂停一个周期.
     具体地, 当 `popq` / `mrmovq` 处于**Execute**阶段时, 且正在**Decode**的指令需要对应寄存器,
     则将**Decode**阶段的指令阻塞, 并在下一个周期在**Execute**阶段插入气泡.
     这可以通过将 `流水线寄存器F&D` 的状态保持固定来实现.
  2. **ret**: 流水线必须暂停3个周期, 直到ret指令到达写回阶段

     > 看似到达**Memory**阶段拿到 `m_valM` 就够了, 但可能有时序问题而来不及.
     > 可以这样理解: 如果把 `m_valM` 转发给**Fetch**阶段使用, 则这一个周期进行了两次内存访问, 太费时间了!
     > (即在**Memory**阶段取到指令地址后, 又在**Fetch**阶段从指令内存取指令, 共涉及**两次**内存访问.)

  3. **预测错误的分支**: 取消不该被取出的指令, 并从跳转指令后面的那条指令开始取指.
  4. **异常**:
     - 当发现**Memory**阶段是异常指令时:
       - 禁止**Execute**阶段设置 `CC寄存器` ;
       - 向**Memory**阶段插入气泡, 以禁止 `内存` 写;
     - 当发现**WriteBack**阶段是异常指令时:
       - 暂停**WriteBack**阶段, 暂停流水线.

> Q: 如何**发现**上述4种情况?
>
> A: 通过一些简单的组合逻辑块, 如下:

条件	触发逻辑	触发逻辑的解释
加载/使用冒险	E_icode in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB}	正在**Execute**的是**内存读**指令, 正在Decode的指令却需要对应的寄存器值
处理ret	IRET in {D_icode, E_icode, M_icode}	**Decode**/**Execute**/**Memory**阶段是 `ret` 指令, 即 `ret` 还未进入**WriteBack**阶段
预测错误的分支	E_icode in {IJXX} && !e_Cnd	正在**Execute**的是条件跳转, 并发现不该选择分支
异常	m_stat in {SADR, SINS, SHLT} || W_stat in {SADR, SINS, SHLT}	**Memory**阶段是异常指令 (注意检查的是 `m_stat` 而非 `M_stat` , 因为后者不包含可能出现的 `SADR`), 或**Writeback**阶段是异常指令

> Q: 流水线**控制机制**是什么?
>
> A: 通过对流水线寄存器的两个输入信号 `stall` (暂停)和 `bubble` (气泡)控制流水线寄存器的行为, 具体地:

- 当 `stall` 信号设置为 `1` 时,即时钟上升也不更新状态;
- 当 `bubble` 信号设置为 `1` 时, 时钟上升时产生一个**等效于 `nop` 指令**的状态.

  > 比如向 `流水线寄存器D` 插入气泡时, 要将 `icode` 设置为INOP;
  > 比如向 `流水线寄存器E` 插入气泡时, 要将 `icode` 设置为INOP, 并将 `dstE` `destM` `srcA` 和 `srcB` 设置为RNONE

- 当 `stall` 和 `bubble` 全为 `0` 时, 就是正常工作, 即时钟上升时更新状态.
- 当 `stall` 和 `bubble` 全为 `1` 时, 看作是出错了.
- 三种情况对应的信号如图:

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

> Q: 这图啥意思?
> A: 比如 `Load/use hazard` 行 `E` 列的 `bubble` 的意思是:
> 在这个周期内, 通知 `流水线寄存器E` 进入 `bubble` 状态, 从而当下个周期遇到上升沿的时候, `流水线寄存器E` 要产生气泡, 而不要正常工作.

> 注: Fetch阶段并不能插入气泡. 因为这一阶段的 `流水线寄存器` 上只有一个PredPC, 想想看, 是做不到插入气泡的...

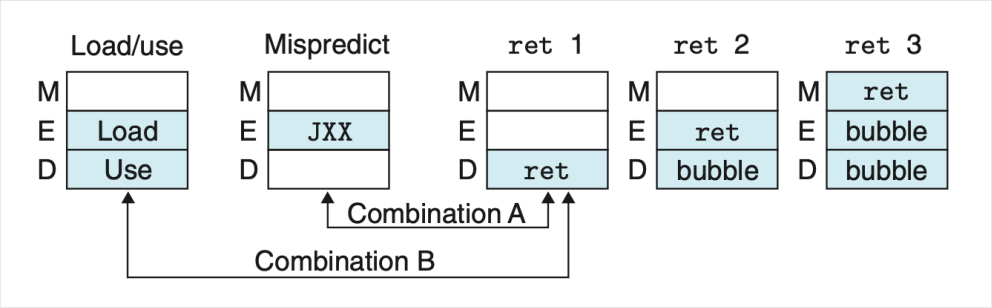
> Q: 上图只考虑了特殊情况单独出现的情形, 如果**多个条件(Condition)组合出现**呢?
>
> A: 下面考虑 `LoadUse` `Mispredict` `retD/retE/retM` 这5种控制条件组合着出现的情形.

事实上, 大多数控制条件是互斥的. 只有两种组合是真正可能出现的:

`[A]`: `jCC` 预测错误后紧随一个 `ret`

`[B]`: 针对 `%rsp` 的Load/Use冒险后紧跟一个 `ret`

(如图所示)



经过验证, 我们发现, [A]的处理是没有问题的, 而[B]的处理有考虑不周的地方:

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble+stall	bubble	normal	normal
Desired	stall	stall	bubble	normal	normal

注意图中的 `bubble+stall`, 这对于流水线寄存器而言, 是一个未定义的非法输入!

那我们希望这个时刻应该是什么呢? 由于涉及Load/Use冒险, 我们当然是希望 `ret` 暂停一个周期, 即暂停在**Decode**阶段. 所以这里我们应该选择 `stall`, 而非 `bubble`. 后者会把暂停着的 `ret` 指令弄丢.

- **`流水线控制逻辑` 的HCL描述**:
  - 明确输入和输出:
    - 输入: 各种 `流水线寄存器的值(大写字母)` 以及 `流水线阶段中产生的信号(小写字母)`
    - 输出: 为各个 `流水线寄存器` 提供 `stall` / `bubble` 信号, 以及决定 `CC寄存器` 是否应该被更新.

    > 这里我们发现 `控制逻辑` 包揽了之前提到的 `setcc` 组合逻辑的功能.

例: `F_stall` 信号:

```
bool F_stall =
    # 1. ret
  # 2. Load/Use冒险
  IRET in {D_icode, E_icode, M_icode}       # ret
                    ||
```

```
# 或者
  E_icode in {IRMMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB};        # Load/Use
```

例: `D_bubble` 信号:

```
bool D_bubble =
    # 1. ret会在Decode阶段插入bubble, 然而如果是紧随Load/Use冒险的ret, 则不该插入bubble
  # 2. 分支预测错误的jCC会在Decode阶段插入bubble.
  IRET in {D_icode, E_icode, M_icode}   # 是ret,
                &&
# 但又..
    !(E_icode in {IPOPQ, IMRMOVQ} && E_dstM in {d_srcA, d_srcB})    # 不是L/U冒险
                ||
  E_icode == IJXX && !e_Cnd     # 分支预测错误
```

例: `setcc` 信号:

```
bool set_cc =
    # 执行阶段是一个OPq指令  &&  Memory阶段不是异常指令  &&  Writeback阶段是异常指令
    E_icode == IOPQ  &&  !m_stat in {SADR, SINS, SHLT}  &&  !W_stat in {SADR, SINS,
SHLT};
```

End Of Note, 585 Lines.