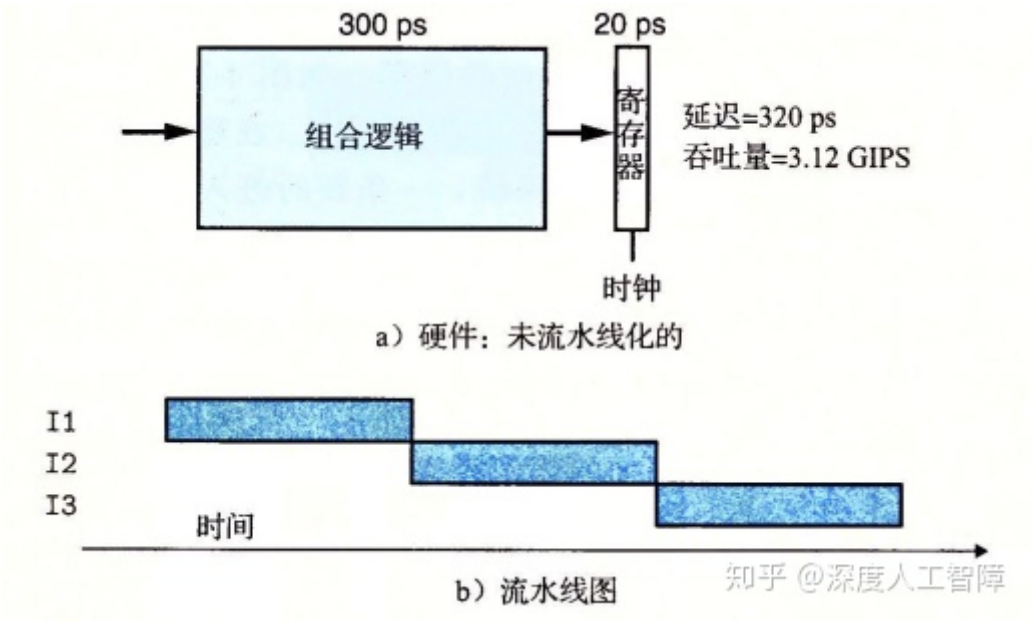
[读书笔记]CSAPP：13[B]处理器体系结构：流水线

1 流水线的通用原理



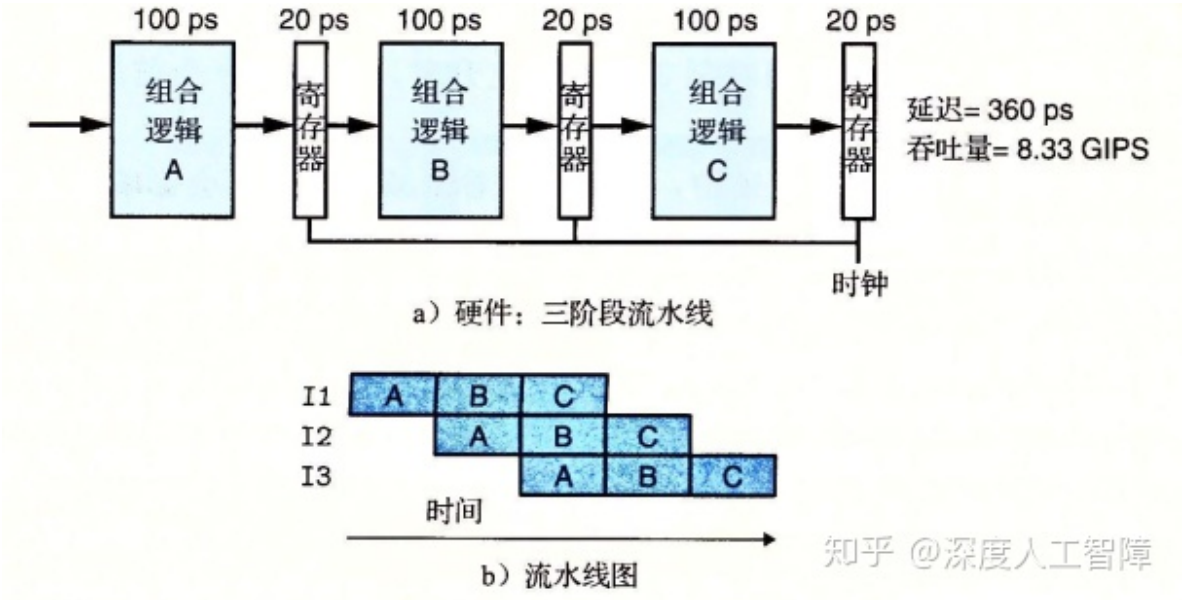
如上图所示是一个非流水线化的计算硬件。当信号输入到组合逻辑中时，通过一系列逻辑门经过300ps获得输出信号，然后经过20ps将结果加载到寄存器中，由于时钟周期控制存储器写入的频率，为了保证当时钟变为高电平之前，能够得到将计算好的结果放到寄存器的输入端口，则这里的时钟周期设定为300+20=320ps。

我们将从头到尾执行一条指令所需的时间称为**延迟（Delay）**，则这里延迟为320ps。我们将系统在单位时间内能执行的指令数目称为**吞吐量（Throughput）**，则

$$Throughput = \frac{1I}{320ps} \cdot \frac{1000ps}{1ns} = 3.12GIPS$$

意味着一秒能执行3.12G条指令。

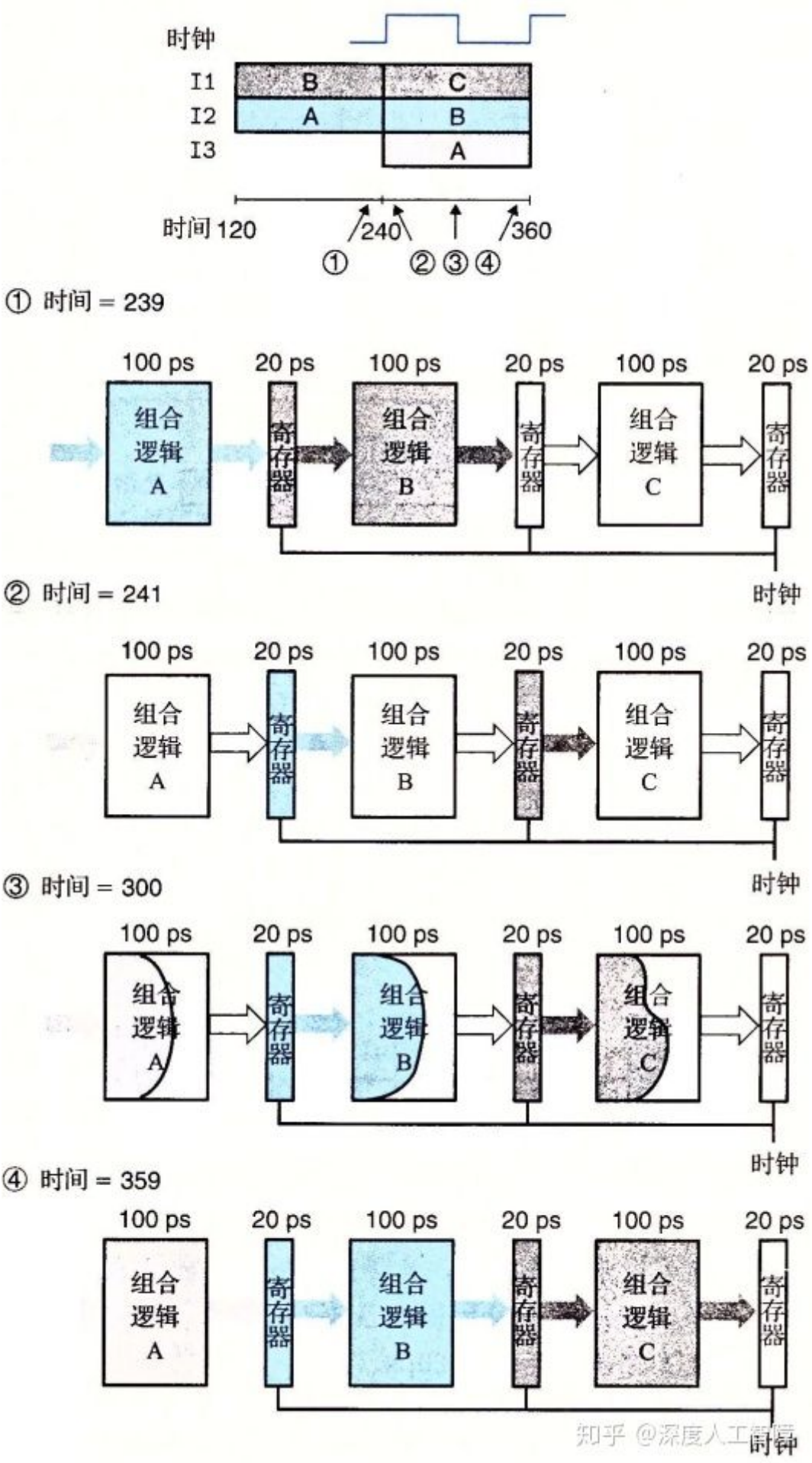
由于这个是非流水线化的计算硬件，所以从流水线图中可以看到在开始下一条指令之前必须完成上一条指令。如果我们将组合逻辑根据不同功能，通过**流水线寄存器（Pipline Register）**划分成独立的三阶段，就能得到简易的流水线化计算硬件。



由于每阶段的组合逻辑实现独立的功能，并且能通过流水线寄存器来控制进入下一阶段的时机，所以如上图的流水线图所示，只需要通过流水线寄存器控制每个阶段只执行一条指令，就能流水线化地执行指令。

对于每个阶段，我们需要100ps的组合逻辑计算时间以及20ps加载到寄存器的时间，所以我们这里能将时钟周期设定为120ps。并且我们可以发现每过一个时钟周期就有一条指令完成，所以吞吐量变为了8.33GIPS，提高了2.67倍。但是每条指令需要经过3个时钟周期，所以延迟为360ps，变为原来的1.12倍。

所以**流水线特点**为：提高系统的吞吐量，但是会轻微增加延迟。



上图是其中一段时间详细过程。

1. 239ps时，`I2`经过组合逻辑A的计算到达寄存器A，`I1`经过组合逻辑B的计算到达寄存器B，此时时钟还处于低电平，则流水线寄存器还未读取组合逻辑计算的结果，还保持着原来的值。

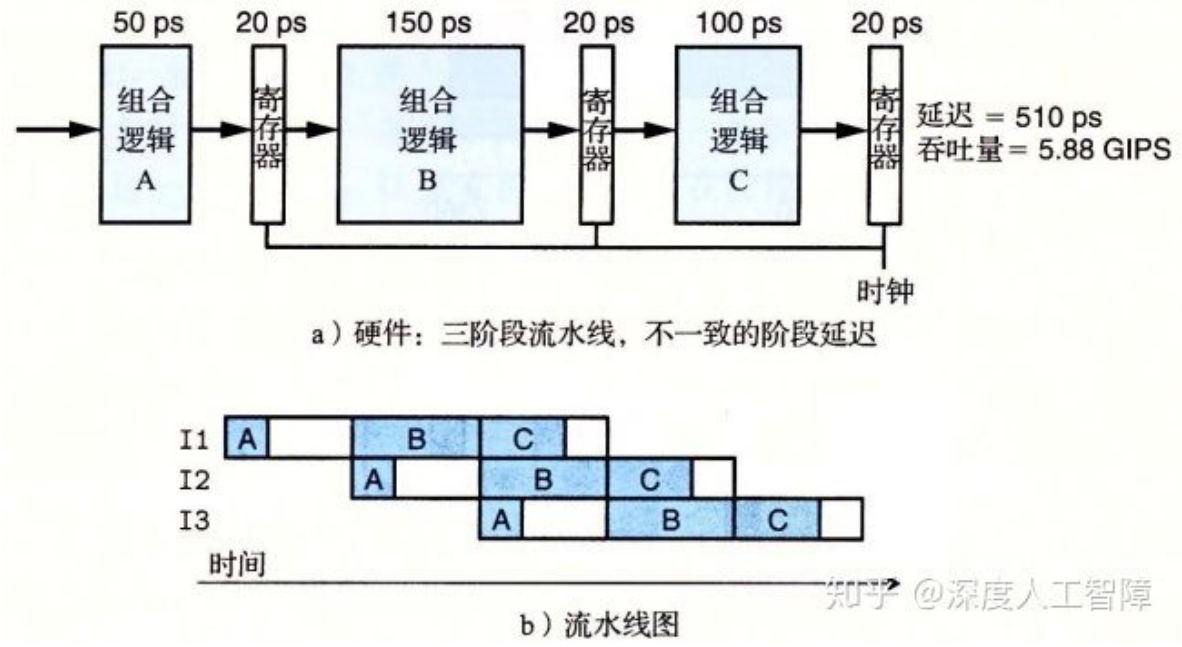
2. 241ps时，时钟已经变成高电平了，此时寄存器就会读取组合逻辑计算的结果，将其保存到寄存器中。则寄存器A保存`I2`在组合逻辑A中计算的结果，寄存器B保存`I1`在组合逻辑B中计算的结果。
3. 300ps时，寄存器中的值已经输入到下一阶段的逻辑电路一段时间了，则该输入信号会以不同的速率在逻辑电路中传播，形成了图中所示的**波阵面（Curved Wavafront）**。
4. 359ps时，又重复到了1中相似的状态。

时钟周期的影响：时钟周期用来控制流水线寄存器的读取频率，用来将不同阶段分隔开来，互不干扰。如果时钟周期太快，组合逻辑的计算还未完成，就会使得非法的值保存到寄存器中。如果时钟周期太慢，不会导致计算错误，只是效率会比较低。

1.1 流水线的局限性

1.1.1 不一致的划分

处理器中的某些硬件单元，比如ALU或内存，是无法划分成多个延迟较小的单元的，这使得我们划分的不同阶段的组合逻辑具有不同的延迟。

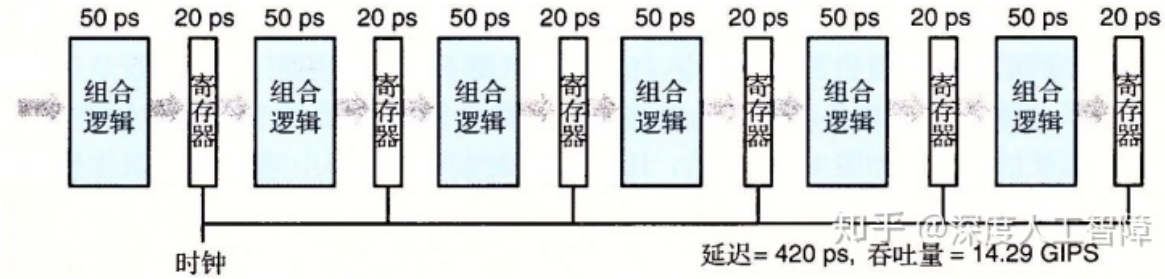


如上图所示，其中组合逻辑B需要150ps进行计算。由于整个系统共用一个时钟周期，为了保证组合逻辑B能在一个时钟周期内计算出正确结果，使得保存到流水线寄存器中，我们就需要将时钟周期设定为150+20=170ps，这使得系统吞吐量变为5.88GIPS，而运行一条指令需要的延迟为3*170=510ps。

注意：想要吞吐量最大，我们需要使得时钟周期尽可能小，而时钟周期受到最慢的组合逻辑的限制，所以我们可以将最小的组合逻辑的时间加上一个寄存器的时延作为时钟周期。想要延迟最小，就不使用流水线。

1.1.2 流水线过深，收益下降

我们将每个组合逻辑进一步划分成更小的部分，构建更深的流水线



这里时钟周期变为70ps，则吞吐量为14.29GIPS。这里我们可以发现，虽然我们将组合逻辑分成了更小的单元，使得组合逻辑的时延缩小了两倍，但是吞吐量的性能并没有提升两倍。这是由于更深的流水线，会扩大寄存器时延的影响，在70ps的时钟周期中，寄存器的时延就占了28.6%，意味着更深的流水线的吞吐量会依赖于寄存器时延的性能。

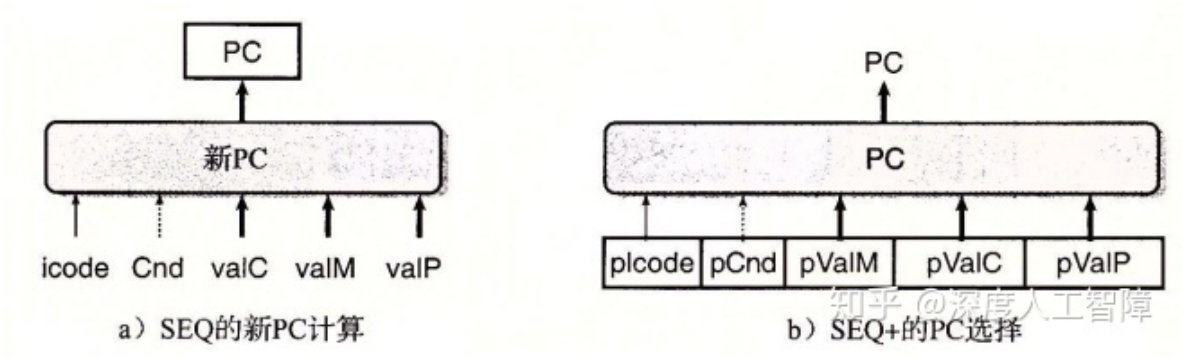
1.2 指令相关

我们之前考虑流水线时，只有当指令之间是不相关时才是完全正确的。但是真实系统中，指令之间存在两种形式的相关：**数据相关（Data Dependency）**，下一条指令会用到这条指令计算出来的结果；**控制相关（Control Denpendency）**，一条指令要确定下一条指令的位置。这些相关可能会导致流水线产生计算错误，称为**冒险（Hazard）**，包括：**数据冒险（Data Hazard）**和**控制冒险（Control Hazard）**。

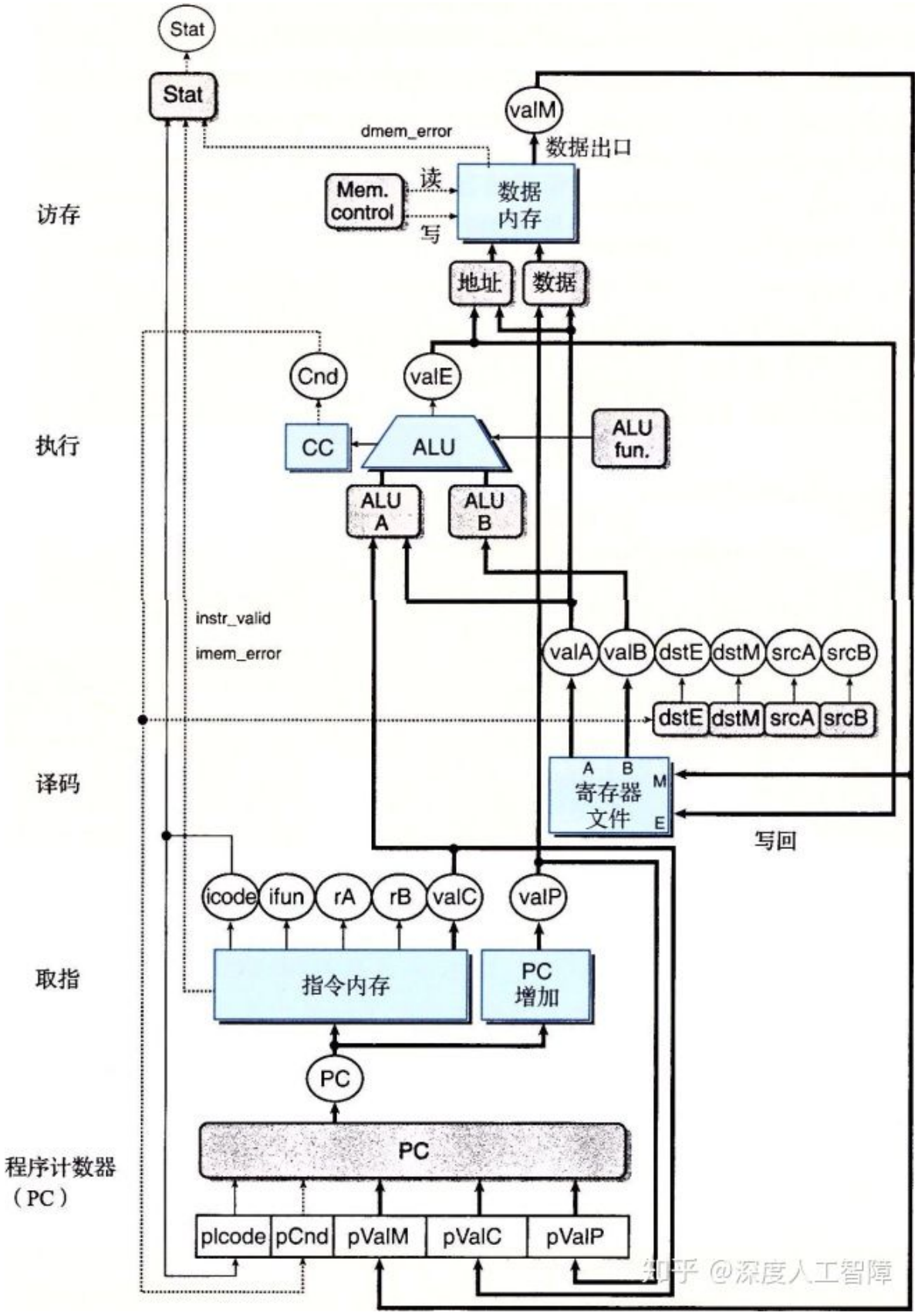
2 Y86-64流水线实现

2.1 SEQ+和PIPE-

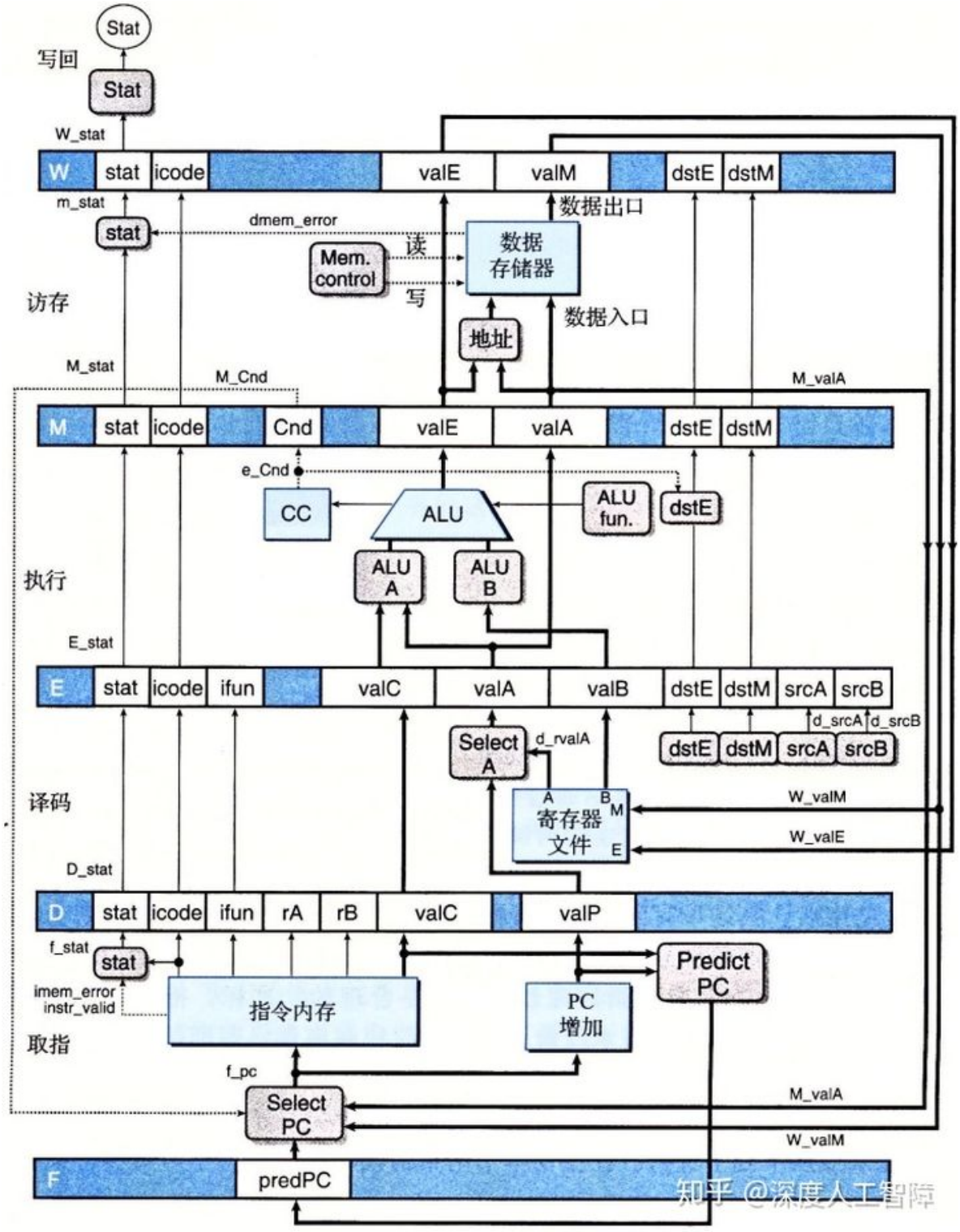
为了平衡一个流水线系统各个阶段的延迟，需要使用**电路重定时（Circuit Retiming）**在不改变逻辑行为的基础上，修改系统的状态表示。如下图所示，顺序实现的SEQ中，更新PC阶段是在时钟周期结束时才执行的，通过组合电路计算得到的`icode`、`Cnd`、`valC`、`valM`和`valP`通过组合电路计算得到新的PC，将其保存到PC的时钟寄存器中。但是这些值是在不同阶段中计算出来的，所以SEQ+新增了一系列状态寄存器来保存之前计算出来的结果，然后将更新PC阶段放到了时钟周期开始执行，这样在每个阶段时钟周期变成高电平时就会将该阶段计算出来的值保存到状态寄存器中，然后PC逻辑电路就能根据当前的状态寄存器的值来预测下一步的PC值。



对应的SEQ+硬件结构如下图所示，可以发现将更新PC阶段移到了时钟周期开始的位置。



我们可以在各个阶段中加入流水线寄存器，并将信号重新排列来将SEQ+转换成初步的流水线处理器PIPE-，硬件结构如下图所示



- 分别插入了5个流水线寄存器用来保存后续阶段所需的信号，编号为`F`、`D`、`E`、`M`和`W`。我们可以发现在`D`和`E`中都有`stat`信号，分别表示为`D_stat`和`E_stat`。在取指阶段和访存阶段都有通过逻辑计算得到`stat`信号，分别表示为`f_stat`和`m_stat`。
- 在SEQ+中，在译码阶段通过逻辑电路计算得到`dstE`和`dstM`，会直接将其连接到寄存器文件的写端口的地址输入，当计算出`valE`和`valM`时直接写回到对应寄存器中。但是`dstE`和`dstM`是在译码阶段计算出来的，而`valE`是在执行阶段计算得到，`valM`是在访存阶段获得的，在流水线系统PIPE-中各个阶段是相互独立的，当某条指令运行到写回阶段时，得到了`valE`和`valM`，但是当前的`dstE`和`dstM`是处于译码阶段的指令计算出来的，会出现错误，所以需要将`dstE`和`dstM`一直保存到后续的流水线寄存器中。**通用规则：**我们要保存处于一个流水线阶段中的指令的所有信息。
- 我们可以发现，只有`call`指令需要将`valP`保存到内存中，即我们为了`call`指令需要将取指阶段得到的`valP`一直保存到后续的流水线寄存器中，直到访存阶段将其保存到内存中。但是我们发现`call`指令只使用`valB`保存`%rsp`的值，并不会使用`valA`，所以我们可以通过PIPE-中的`selectA`模块将`valP`保存到`valA`，由此就不需要保存`valP`了。同理条件跳转指令，当不选择跳转分支时，后面也需要`valP`，也可

以将其保存到 valA 中，由此也不需要保存 valP 了。**通用规则：**通过合并信号来减少寄存器状态和线路的数量。

2.2 处理控制相关

对于 call 和 jmp 指令，下一条指令的地址就是 valC，而除了条件分支和 ret 指令外，下一条指令的地址就是 valP，这些指令不存在控制相关，使得流水线处理器能够每个时钟周期就处理一条指令。如果出现了条件分支，则需要该指令运行到执行阶段后才知道是否选择该分支，如果出现了 ret 指令，则需要该指令运行到访存阶段，才知道返回地址，此时就存在了控制相关，使得处理器要经过几个时钟周期才知道要运行的下一条指令的地址，所以**控制冒险只会出现在条件分支和 ret 指令中**，我们可以通过预测下一条PC来处理这个问题。

- **条件分支：**我们可以通过**分支预测**技术来预测分支方向，并根据预测开始取值。常见的技术包括：
- - **总是选择（always taken，AT）：**总是预测处理器选择了条件分支，因此预测PC值为 valC，成功率大约为60%。
  - **从不选择（never taken，NT）：**总是预测处理器不选择条件分支，因此预测PC值为 valP，成功率大约为40%。
  - **反向选择、正向不选择（backward taken, forward not-taken，BTFNT）：**条件分支通常用于循环操作，当跳转地址比下一条指令地址小，说明进入了循环，否则退出循环，而循环通常会执行多次，因此当跳转地址比下一条指令地址低就选择分支，否则就不选择分支， 成功率大约为65%。
- ret 指令：常见的技术包括
- - 暂停处理新指令，直到 ret 指令通过写回阶段知道下一条指令的地址
  - 在取指单元中放入一个硬件栈，保存过程调用指令产生的返回地址

当预测PC出现错误时出现控制冒险，会执行错误的指令，所以会极大影响流水线处理器的性能，后面再讨论这个问题。

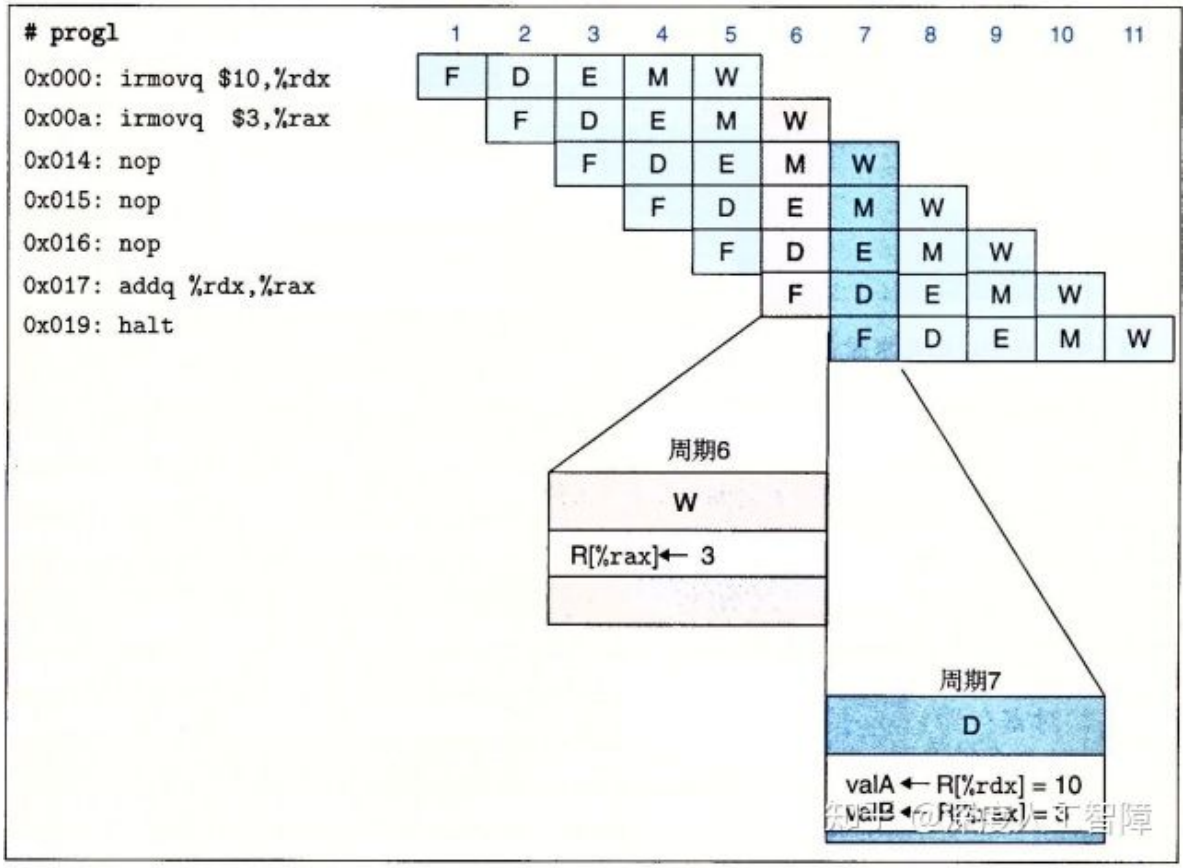
在本文中，条件分支使用AT策略，ret 指令使用第一条策略。从PIPE-硬件结构中可知，在取值阶段首先根据 icode 从 valP 和 valC 中选出预测的PC值，对于 call 和分支使用 valC，其他指令使用 valP。然后 Select PC 逻辑电路再从 predPC、M_valA 和 W_valM 中进行选择。我们推测为什么是这样的

- **条件分支：**首先条件分支在取指阶段会直接选择条件分支，使得 predPC 为 valC，则当条件分支执行到译码阶段时，valC 对应的指令就会在取指阶段开始执行。当条件分支执行到执行阶段时，可以通过 CC 知道是否真的要选择条件分支，如果真的选择分支，则继续执行，否则条件分支的下一条指令地址应该是 valP，此时该条件分支对应的 valP 保存在 M_valA 中，所以可以让 Select PC 选择 M_valA 来重新执行条件分支的部分。
- **ret 指令：**当执行 ret 指令时，会暂停传入新的指令，知道 ret 指令执行到访存阶段时，才从内存中读取了下一条指令的返回地址，保存在 W_valM 中，所以 Select PC 可以选择 W_valM 来执行返回地址对应的指令。

2.3 流水线冒险

流水线冒险主要包含数据冒险和控制冒险，当程序状态的读写**不处于同一阶段**，就可能出现数据冒险，当出现分支预测错误或 ret 指令时，会出现控制冒险。

在Y86-64中，程序状态包含程序寄存器、内存、条件码寄存器和状态寄存器。程序寄存器的读取处于译码阶段，而写入处于写回阶段，因此程序寄存器会出现数据冒险的可能，以以下代码为例



我们在代码中插入了三行 `nop` 指令，则当 `addq %rdx, %rax` 处于译码阶段读取寄存器时，第一行和第二行指令已经完成了对寄存器 `%rdx` 和 `%rax` 的写入操作，因此该代码不会出现数据冒险，但是如果减少 `nop` 指令，第一行和第二行指令还没完成对寄存器的写入操作时，`addq %rdx, %rax` 已经处于译码阶段读取寄存器了，此时就会读取到错误的值而出现数据冒险。由于读取操作和写入操作相差3个时钟周期，所以如果一条指令的操作数被它前面三条指令中的任何一条修改时，就会出现数据冒险。

而内存的读写都处于访存阶段、条件码寄存器的读写都处于执行阶段因此它们不会出现数据冒险的情况，而我们为每个阶段都在流水线寄存器中保留了stat值，所以当异常发生时，处理器就能有条理地停止。

所以这里我们主要探讨程序寄存器数据冒险和控制冒险。

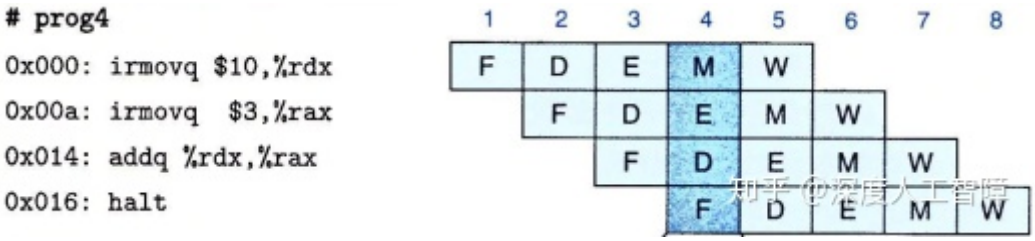
2.3.1 用暂停来避免数据冒险

我们可以在执行阶段中插入一段自动产生的 `nop` 指令，来保持寄存器、内存、条件码和程序状态不变，使得当前指令停在译码阶段，并且会控制程序计数器不变，使得下一条指令停在取指阶段，直到产生指令的源操作数的指令通过了写回阶段。（相当于产生类似上图的指令序列）

该方法指令要停顿最少一个最多三个时钟周期，严重降低整体的吞吐量。

2.3.2 用转发来避免数据冒险

对于以下代码我们可以发现，`I1` 处于访存阶段而 `I2` 处于执行阶段，都还没有将 `valE` 保存在 `%rdx` 和 `%rax` 中，所以 `I3` 的译码阶段无法从寄存器文件中读取到正确的 `%rax` 和 `%rdx`。



但是即使还没有将 `valE` 保存到对应的寄存器文件中，其实 `I1` 在执行阶段已经将 `%rdx` 的值保存到流水线寄存器M中 `M_valE`，而 `I2` 在执行阶段通过ALU计算得到了 `%rax` 的值 `e_valE`，所以即使没有写入对应的寄存器文件中，已经能从 `M_valE` 和 `e_valE` 得到 `%rax` 和 `%rbx` 的值了，所以 `I3` 的译码阶段可以从以下形式

```
valA = R[%rdx]
valB = R[%rax]
```

变成

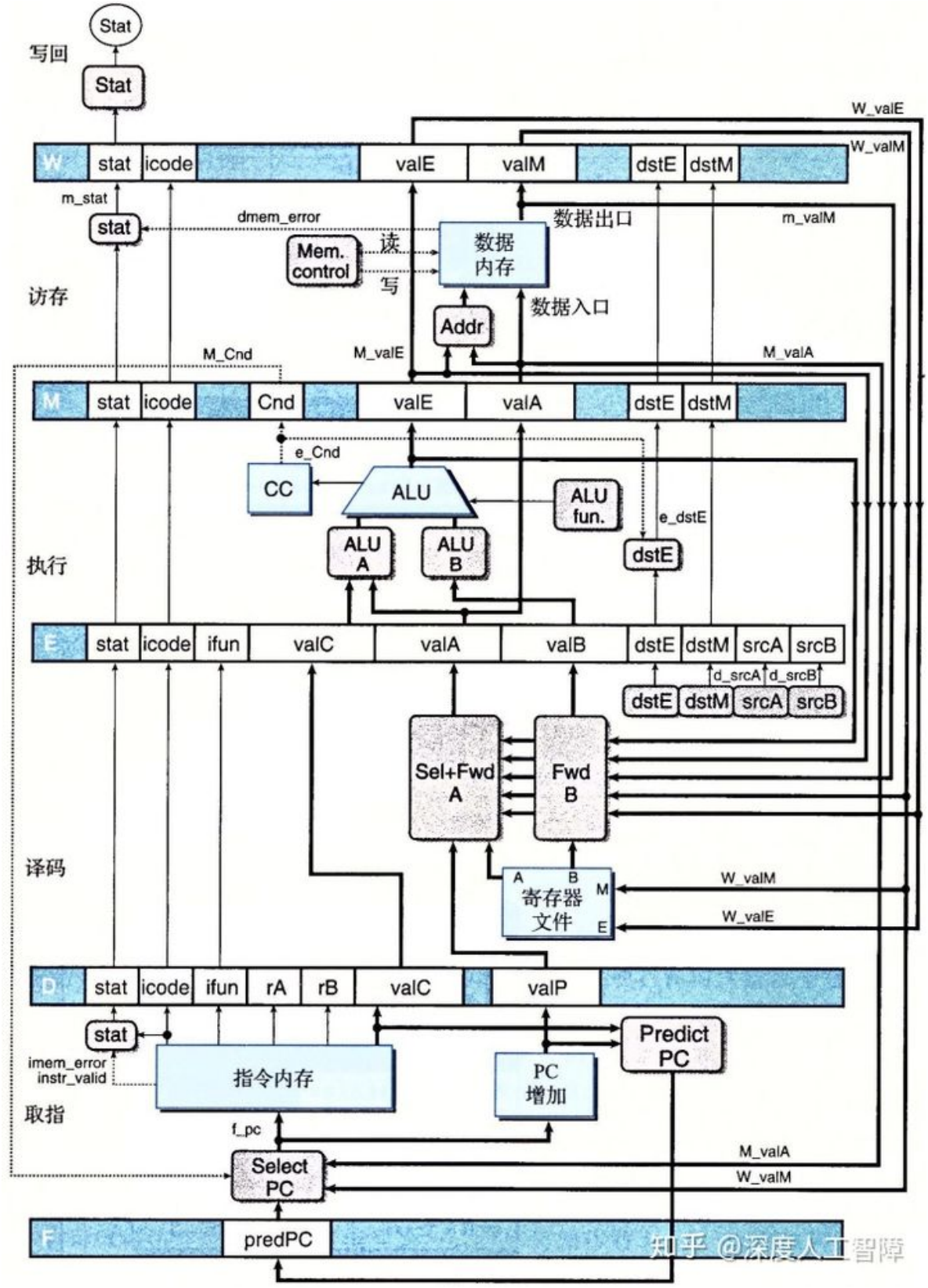
```
valA = M_valE
valB = e_valE
```

此时就不存在数据冒险，以及暂停了。

除了通过ALU的计算结果来转发，还能通过内存来进行转发，并且通过当前阶段的`dstE`和`dstM`与目标指令的`srcA`和`srcB`进行判断来决定是否转发。在处理器中，`valA`和`valB`一共有5个转发源：

- `e_valE`：在执行阶段，ALU中计算得到的结果`valE`，通过`E_dstE`与`d_srcA`和`d_src_B`进行比较决定是否转发。
- `M_valE`：将ALU计算的结果`valE`保存到流水线寄存器M中，通过`M_dstE`与`d_srcA`和`d_src_B`进行比较决定是否转发。
- `m_valM`：在访存阶段，从内存中读取的值`valM`，通过`M_dstM`与`d_srcA`和`d_src_B`进行比较决定是否转发。
- `W_valM`：将内存中的值`valM`保存到流水线寄存器W中，通过`W_dstM`与`d_srcA`和`d_src_B`进行比较决定是否转发。
- `W_valE`：将ALU计算的结果`valE`保存到流水线寄存器W中，通过`W_dstE`与`d_srcA`和`d_src_B`进行比较决定是否转发。

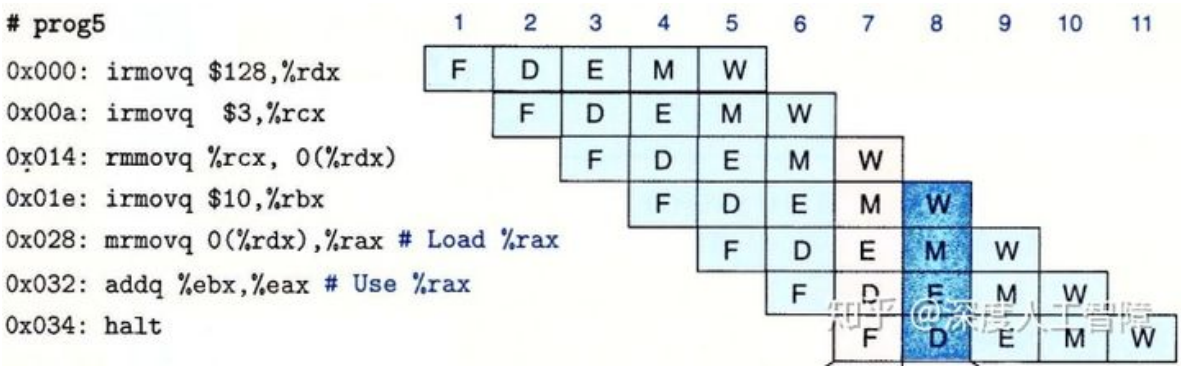
数据转发需要在基本的PIPE-的硬件结构基础上增加一些额外的数据连接和控制逻辑，则PIPE的硬件结构如下图所示



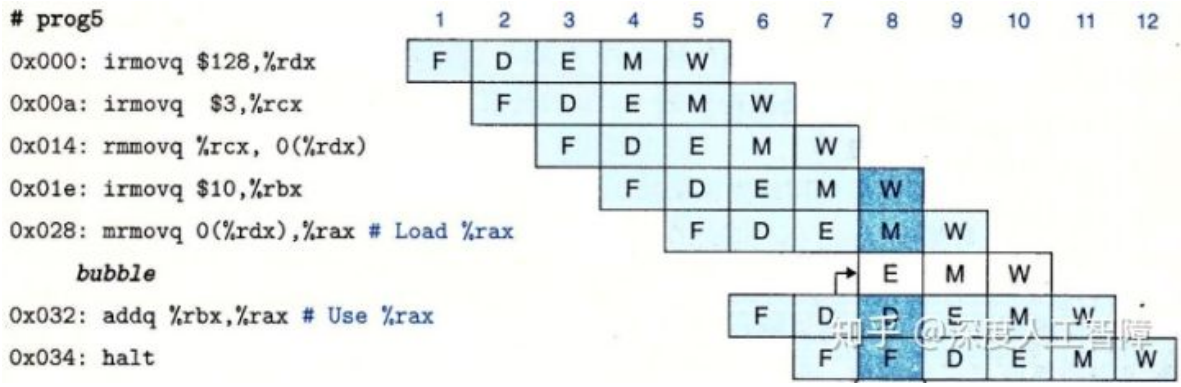
其中 `Fwd B` 负责 `valB` 的转发逻辑， `Sel+Fwd A` 负责是 `Select A` 模块加上 `valA` 的转发逻辑。

2.3.3 加载/使用数据冒险

我们考虑以下代码，可以发现在执行 `0x032` 指令的译码阶段时， `%rbx` 的值通过转发技术可以从 `M_valE` 中获得，但是 `%rax` 的值需要 `0x028` 指令执行到访存阶段，才能从内存中读取到 `%rax` 的值，但是当前 `0x028` 指令处于执行阶段，所以无法通过转发技术来解决这个数据冒险。



我们可以通过**加载互锁（Load Interlock）**方法来处理这种加载/使用数据冒险，其实就是引入了暂停，如下图所示，当 `0x032` 指令执行到译码阶段时，对该指令暂停一个时钟周期，此时 `0x028` 指令就能执行到访存阶段，此时就能从 `m_valM` 中获得 `%rax` 的值。



结合加载互锁和转发技术足以解决所有类型的数据冒险，并且对模型的吞吐量不会造成很大的影响。

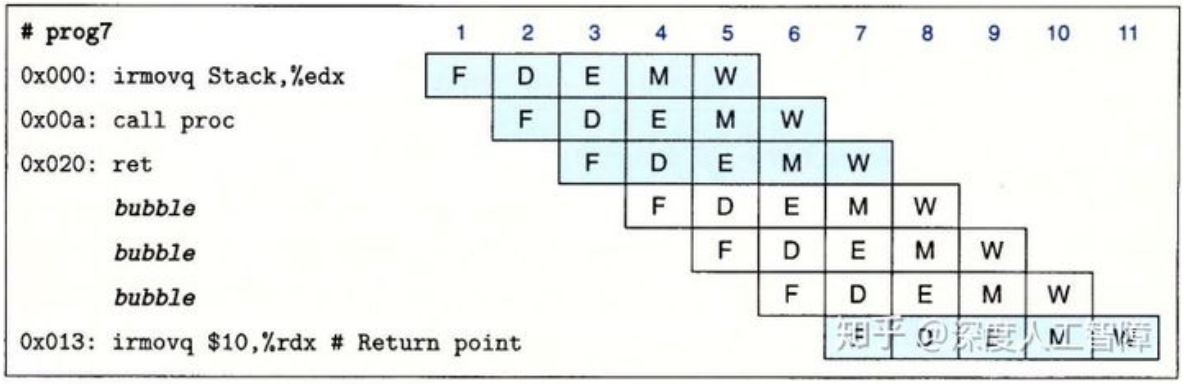
2.3.4 避免控制冒险

控制冒险只会出现在 `ret` 指令和跳转指令预测错方向时产生。

- `ret` 指令

```
0x000:     irmovq stack,%rsp  #   Initialize stack pointer
0x00a:     call proc          #   Procedure call
0x013:     irmovq $10,%rdx    #   Return point
0x01d:     halt
0x020: .pos 0x20
0x020: proc:                  # proc:
0x020:     ret                #   Return immediately
0x021:     rrmovq %rdx,%rbx   #   Not executed
0x030: .pos 0x30
0x030: stack:                 # stack: Stack pointer
```


对于以上代码，对应的流水线图为

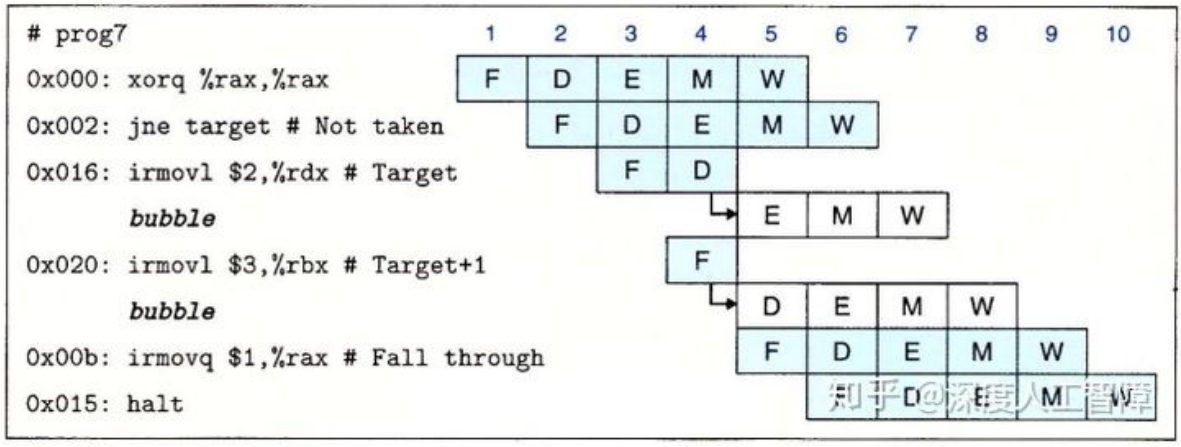


可以发现，当执行`call proc`时，在取指阶段就能获得`valC`表示下一条指令的地址，所以会取到`ret`指令。而`ret`指令只有运行到访存阶段时才能获得返回地址`valM`，并且在写回阶段的时钟电平变高时，才会写入PC寄存器中，所以需要在`ret`指令后添加3个bubble。

- 跳转指令

```
0x000:     xorq %rax,%rax
0x002:     jne  target          # Not taken
0x00b:     irmovq $1, %rax      # Fall through
0x015:     halt
0x016: target:
0x016:     irmovq $2, %rdx      # Target
0x020:     irmovq $3, %rbx      # Target+1
0x02a:     halt
```


对于以上代码，对应的流水线图为



首先对于跳转分支，我们采用AT策略，所以在执行`jne target`的取指阶段时获得的`valC`会直接作为下一条指令的地址。当跳转指令运行执行阶段时，就会通过`CC`和`ifun`得知是否预测正确，此时已经将下一条指令运行到译码阶段，第二条指令运行到了取指阶段，如果预测错误，就会分别插入两个bubble，避免运行到后续阶段，改变程序员可见状态，会浪费两个时钟周期。

2.4 异常处理

异常可以由程序执行从内部产生，也可以由某个外部信号从外部产生。当前的ISA包含三种内部产生的异常：1. halt指令；2. 非法指令码和功能码组合的指令；3. 取值或数据读写访问非法地址。外部产生的异常包括：接收到一个网络接口受到新包的信号、点击鼠标的信号等等。

在我们的ISA中，希望处理器遇到异常时，会停止并设置适当的状态码。**要求：**异常指令之前的所有指令已经完成，后续的指令都不能修改条件码寄存器和内存。流水线系统包含以下问题：

值	名字	含义
1	AOK	正常操作
2	HLT	遇到器执行 halt 指令
3	ADR	遇到非法地址
4	INS	遇到非法指令

1. 当同时多条指令引起异常时，处理器应该向操作系统报告哪个异常？**基本原则：**由流水线中最深的指令引起的异常，优先级最高，因为指令在流水线中越深的阶段，表示该指令越早执行。
2. 在分支预测中，当预测分支中出现了异常，而后由于预测错误而取消该指令时，需要取消异常。

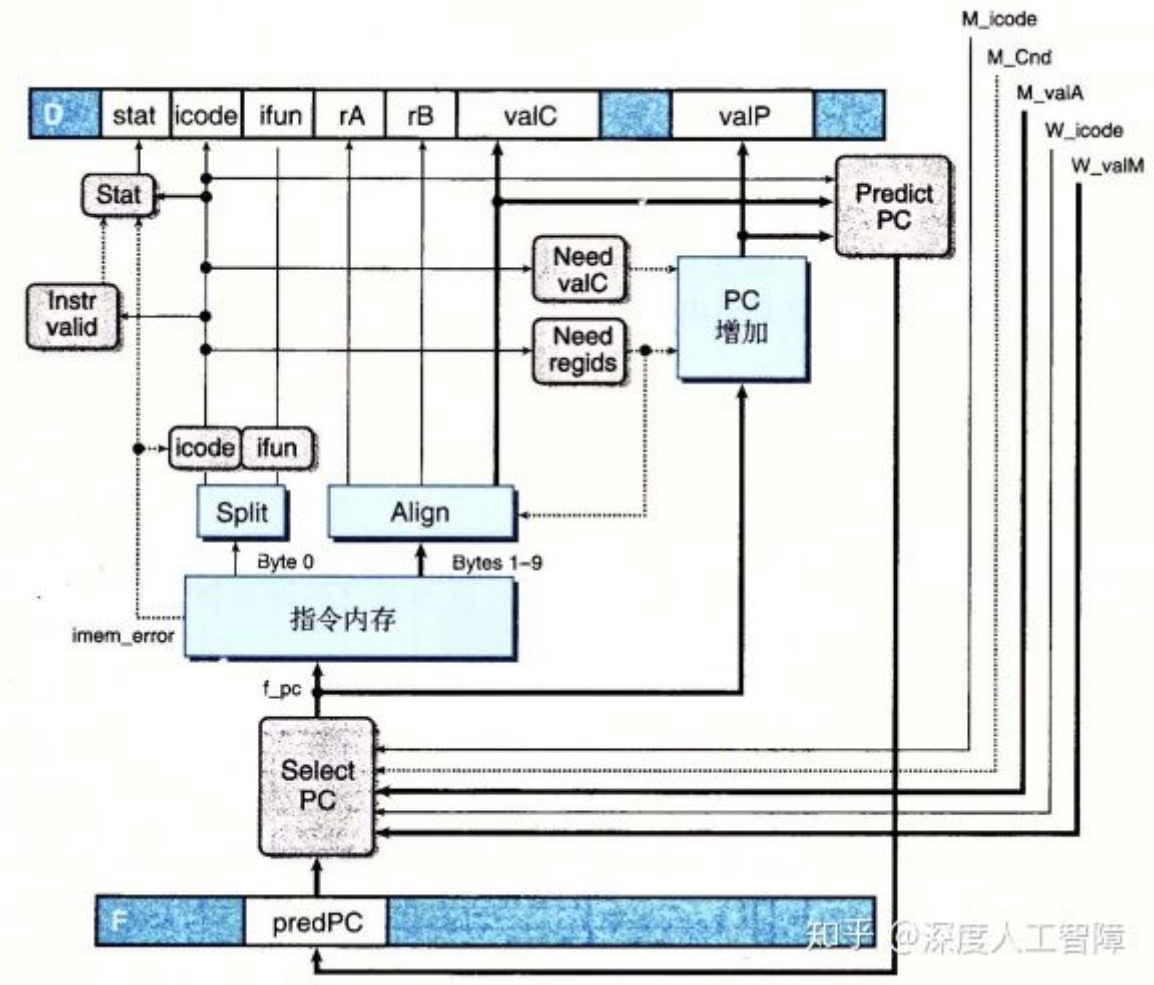
在PIPE硬件架构中，我们对每个流水线寄存器中都设置了一个`stat`信号，用来保存当前阶段的异常信号，随着流水线的进行，就能解决以上问题：

1. `stat`信号只是简单存放在流水线寄存器的状态字段中，不会对流水线中的指令流有任何影响，保证了异常指令之前的指令都能完成，但是要进制流水线中后面的指令不能更新条件码寄存器和内存。
2. 当出现异常的指令到达写回阶段时，由于流水线中的指令是顺序执行的，所以能保证当前异常是最早出现的异常。
3. 当条件分支预测错误时，直接取消该指令后，`stat`信号就不会保存下去了。
4. 最终流水线寄存器W中的`stat`信号会被记录为程序状态。

3 PIPE的HCL代码

接下来我们看看各个逻辑块的设置。

3.1 PC选择和取指阶段



当前阶段需要完成选择程序计数器的值，并且预测下一个PC值。

预测PC值时，对于`call`指令时，会直接将`valC`作为下一个PC值，对于条件分支指令，我们选择AT策略，所以也会直接将`valC`作为下一个PC值，其他除了`ret`指令外，都是使用`valP`作为下一个PC值。在图中为`Predict PC`模块，对应的HCL代码为

```
word f_predPC = [
  f_icode in {IJXX, ICALL} : f_valC;
  1                        : f_valP;
];
```

注意：这里需要用前缀表明使用了哪个阶段的值，比如`f_valC`表示使用了取指阶段中计算出来的`valC`，如果是`D_valC`表示保存在流水线寄存器D中的`valC`值。

选择PC值时主要分以下三种情况：

1. 当条件分支运行到执行阶段时，会知道是否出现预测错误，如果出现预测错误，则需要将PC值设置为`valP`值，而当前的`valP`值保存在`M_valA`中
2. 当出现`ret`指令时，会暂停后续指令，直到`ret`指令运行到访存阶段时，从内存读取出了返回地址才是PC值，而当前返回地址保存在`W_valM`中
3. 对于其他指令，直接使用预测的PC值`F_predPC`就行了

注意：`f_predPC`表示当前阶段预测的下一个PC值，而`F_predPC`表示前一条指令预测的当前指令的PC值。

在图中为`Select PC`模块，对应的HCL代码为：

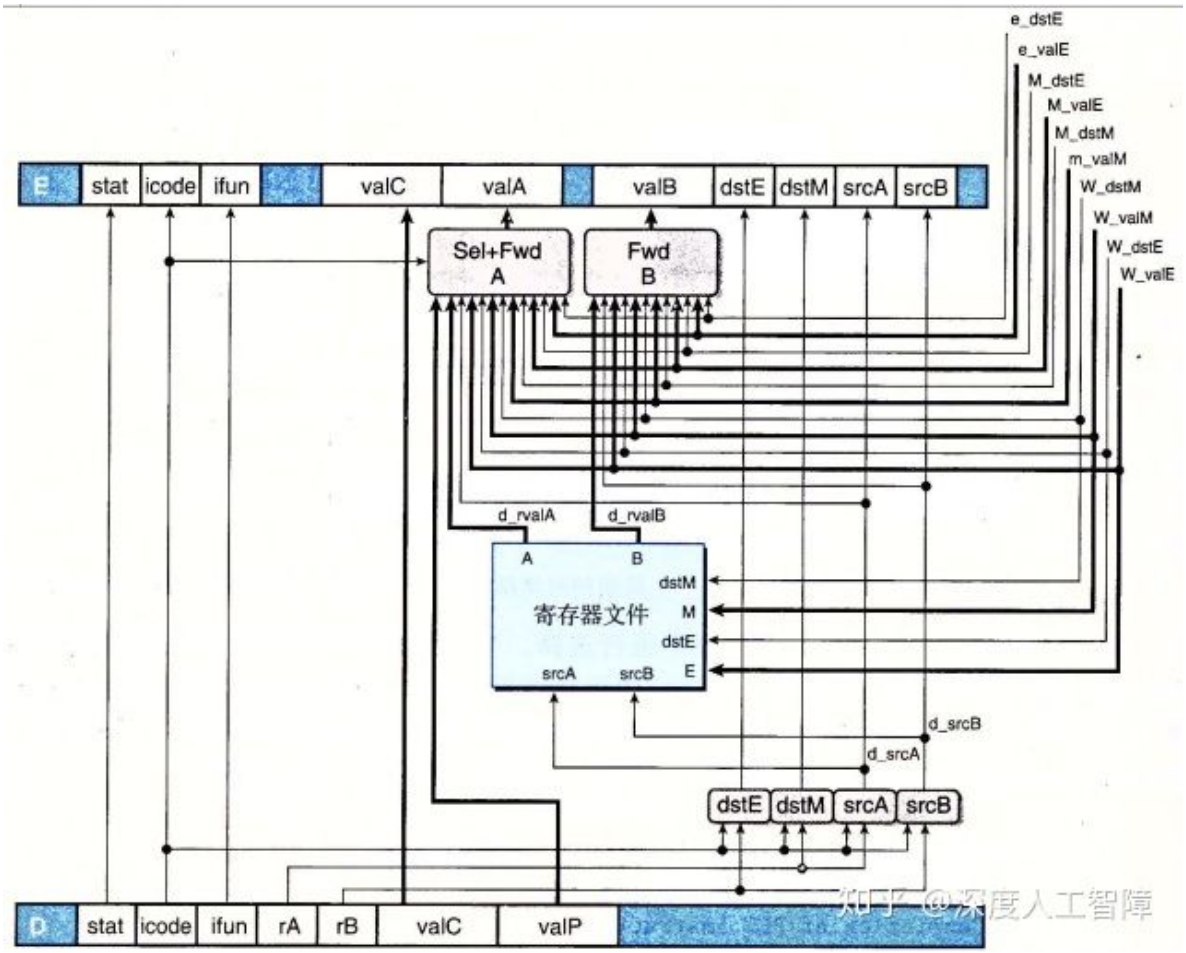
```
word f_pc = [
  #通过M_icode知道是否为条件分支指令，并且在执行阶段会根据ifun和计算结果设置信号Cnd
  M_icode == IJXX && !M_Cnd : M_valA;
  #通过W_icode知道是否为ret指令
  W_icode == IRET           : W_valM;
  #默认
  1                         : F_predPC;
];
```

其他部分和SEQ的HCL代码类似

```
#确定指令的icode
word f_icode = [
  imem_error : INOP;
  1          : imem_icode;
];

#确定指令的ifun
word f_ifun = [
  imem_error :INOP;
  1          : imem_ifun;
];

#判断指令是否合法
bool instr_valid = f_icode in{
  INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
  IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ
};

#判断取指阶段的状态
word f_stat = [
  imem_error       : SADR;
  !instr_valid     : SINS;
  f_icode == IHALT : SHLT;
```

```
  1                : SAOK;
];

#判断是否需要寄存器
bool need_regids = f_icode in {
  IRRMOVQ, IOPQ, IPUSHQ, IPOPQ, IIRMOVQ, IRMMOVQ, IMRMOVQ
};

#判断是否包含valC
bool need_valC = f_icode in {
  IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL
};
```

3.2 译码和写回阶段



在译码阶段，比较复杂的逻辑单元与转发逻辑和合并信号相关

- **合并信号**：我们可以发现只有`call`指令和条件跳转指令在后面阶段需要`valP`，前者用于压入栈中，后者用于预测错误时跳回，并且该两种指令都不需要从寄存器文件A端口读取数据，所以对于这两种指令，可以将`valA`的值设置为`valP`的值，减少流水线寄存器需要保存的信号。
- **转发逻辑**：由于对寄存器文件的读写不在同一阶段，所以可能会造成数据冒险，可以直接使用流水线寄存器中保存的或者每个阶段中计算出来的值，而无需要求从寄存器文件中读取。一共包含5中转发源：`e_valE`、`m_valM`、`M_valE`、`W_valM`和`W_valE`，为了保证能读取到最新指令的结果，应该设置转发源满足从左到右依次降低的优先级。

由此我们可以完成译码阶段的HCL代码

```
#设置从寄存器文件中读取的源
word d_srcA = [
  D_icode in {IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ} : D_rA; #这些指令需要从寄存器rA中读取数据
  D_icode in {IPOPQ, IRET}                    : RRSP; #需要设置栈顶指针，所以需要读取栈值
  1                                           : RNONE;
```

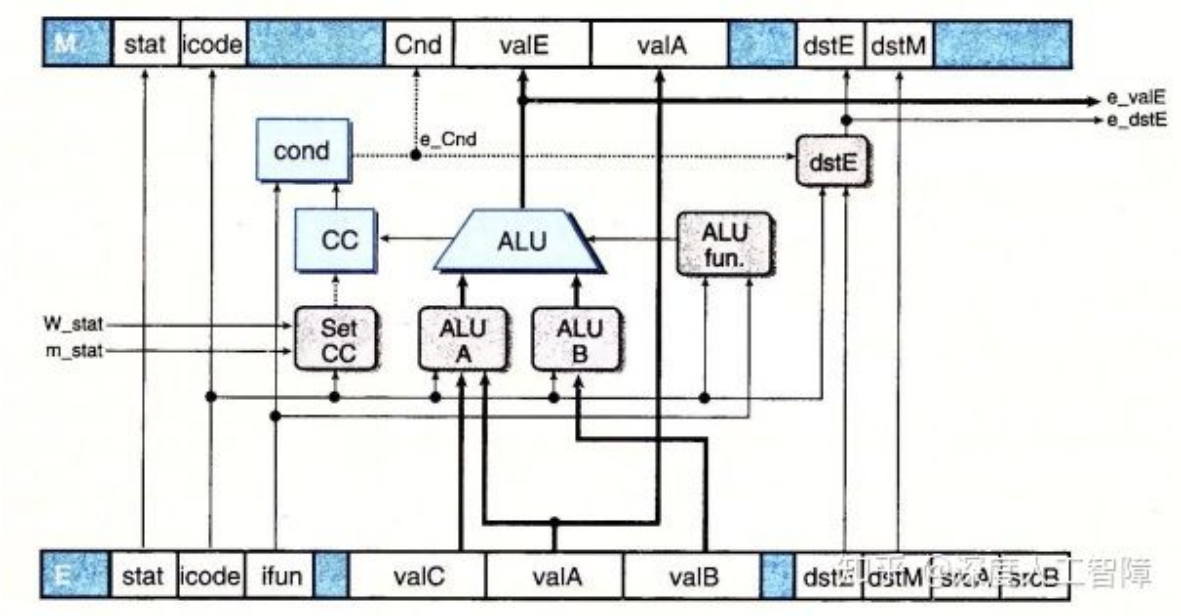
```
];
word d_srcB = [
  D_icode in {IOPQ, IRMMOVQ, IMRMOVQ}     : D_rB; #从内存中读取时需要从寄存器中读取偏移量
  D_icode in {IPUSHQ, IPOPQ, ICALL, IRET} : RRSP;
  1                                       : RNONE;
];
#设置写入寄存器文件的目的
#注意：在译码阶段并不会进行写入，只是先计算出当前指令需要的目的寄存器地址，保存到流水线寄存器中，而后在写回阶段才使用
word d_dstE = [
  D_icode in {IRRMOVQ, IIRMOVQ, IOPQ}     : D_rB;
  D_icode in {IPUSHQ, IPOPQ, ICALL, IRET} : RRSP;
  1                                       : RNONE;
];
word d_dstM = [
  D_icode in {IMRMOVQ, IPOPQ} : D_rA;
  1                           : RNONE;
];

#通过合并信息和转发机制设置valA的值
word d_valA = [
  #合并信息
  D_icode in {ICALL, IJXX} : D_valP;
  #按照转发源的优先级设置转发源
  d_srcA == e_dstE         : e_valE;
  d_srcA == M_dstM         : m_valM;
  d_srcA == M_dstE         : M_valE;
  d_srcA == W_dstM         : W_valM;
  d_srcA == W_dstE         : W_valE;
  #默认都是从寄存器文件中读取的
  1                        : d_rvalA;
];
#通过转发机制设置valB的值
word d_valB = [
  d_srcB == e_dstE         : e_valE;
  d_srcB == M_dstM         : m_valM;
  d_srcB == M_dstE         : M_valE;
  d_srcB == W_dstM         : W_valM;
  d_srcB == W_dstE         : W_valE;
  1                        : d_rvalB;
];
```

注意：在写回阶段时，我们要写入的寄存器文件目的是`W_dstE`和`W_dstM`的值。

当指令运行到最终的写回阶段时，因为流水线寄存器中的`stat`信号表示最近完成指令的处理器状态，所以就可以用来设置处理器的状态

```
word Stat = [
  W_stat == SBUB : SAOK; #特殊情况，当写回阶段时bubble时
  1              : W_stat;
];
```

3.3 执行阶段



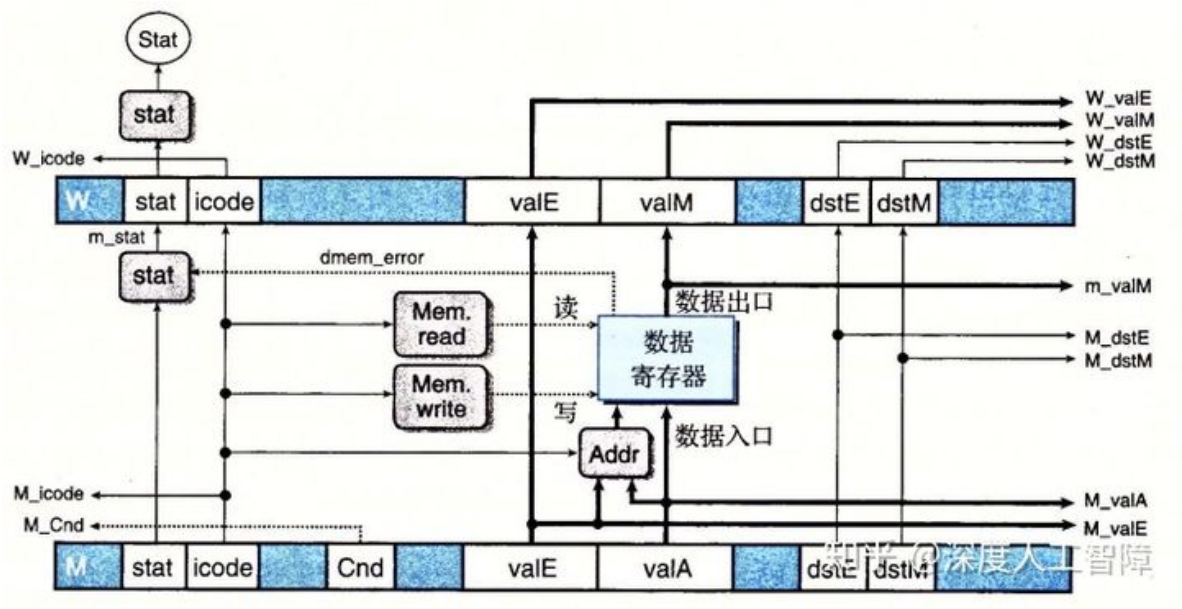
在执行阶段中和SEQ的类似，只是要注意，当前面的指令出现异常时，需要保证条件码寄存器不进行修改，对应的HCL代码为

```
bool set_cc = E_icode == IOPQ && #首先要保证当前指令是算数指令，才会设置CC
  !m_stat in {SADR, SINS, SHLT} && #保证上一条处在访存阶段的指令没有出现异常
  !W_stat in {SADR, SINS, SHLT}; #保证上两条指令没有出现异常
```

其他部分与SEQ的类似

```
#设置进入ALU的值
word aluA = [
  E_icode in {IRRMOVQ, IOPQ}              : E_valA; #直接从寄存器中读取的值
  E_icode in {IIRMOVQ, IRMMOVQ, IMRMOVQ} : E_valC; #包含立即数的值
  E_icode in {ICALL, IPUSHQ}              : -8; #入栈，修改栈顶指针
  E_icode in {IRET, IPOPQ}                : 8; #出栈，修改栈顶指针
];
word aluB = [
  E_icode in {IRMMOVQ, IMRMOVQ, IOPQ, ICALL, IPUSHQ, IRET, IPOPQ } : E_valB;
  E_icode in { IRRMOVQ, IIRMOVQ }                                   : 0;
];

#设置ALU进行的运算
word alufun = [
  E_icode == IOPQ : E_ifun;
  1               : ALUADD;
];

#根据ALU计算的结果设置条件转移
word e_dstE = [
  E_icode == IRRMOVQ && !e_Cnd :RNONE; #当为寄存器之间移动时，这里时条件转移
  1                            : E_dstE;
];

#call指令需要的valP保存在valA中，在访存阶段需要使用
#并且在取指阶段，还会通过当前的M_Cnd来判断跳转指令预测是否正确，再使用M_valA返回
#所以需要将e_val传递下去
word e_valA = E_valA;
```

3.4 访存阶段



这部分的HCL代码和SEQ的类似

```
#设置内存地址
word mem_addr = [
  M_icode in {IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ} : M_valE;
  M_icode in {IPOPQ, IRET}                     : M_valA;
];

#设置读写的控制信号
bool mem_read = M_icode in {IMRMOVQ, IPOPQ, IRET};
bool mem_write = M_icode in {IRMMOVQ, IPUSHQ, ICALL};

#由于当前阶段可能出现内存地址错误，所以还需要设置状态
word m_stat = [
  dmem_error :SADR;
  1          : M_stat;
];

#将其他需要的值传递下去
word w_dstE = W_dstE;
word w_valE = W_valE;
word w_dstM = W_dstM;
word w_valM = W_valM;
```

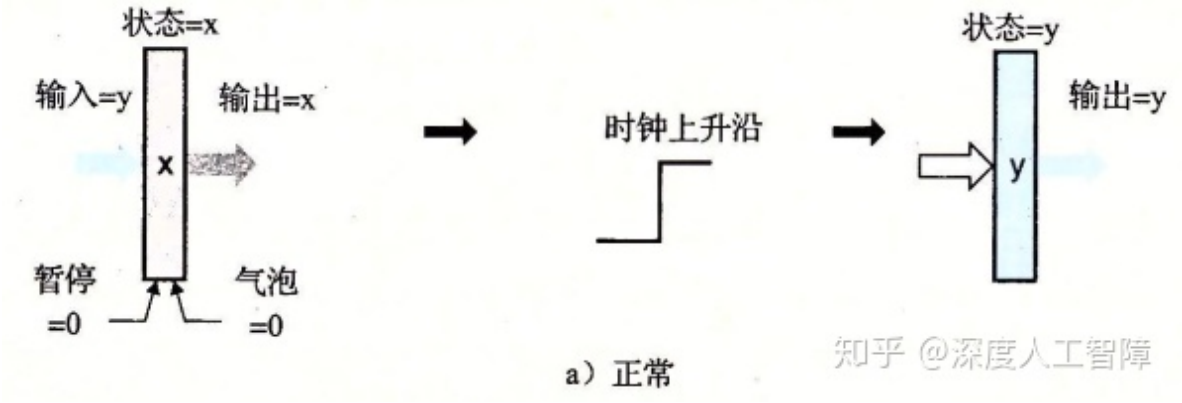
4 流水线控制逻辑

会讨论流水线中低级机制，使得流水线控制逻辑能将指令阻塞在流水线寄存器或往流水线中插入一个气泡。并且在流水线中，还有些特殊情况是其他机制不能处理的，包括：加载/使用冒险、处理`ret`、预测错误的分支、异常等情况。

4.1 暂停和气泡

暂停和气泡是流水线中低级的机制，**暂停**能将指令阻塞在某个阶段，往流水线中插入**bubble**能使得流水线继续运行，但是不会改变当前阶段的寄存器、内存、条件码或程序状态。这两个状态决定了当时钟电平变高时，如何修改流水线寄存器。

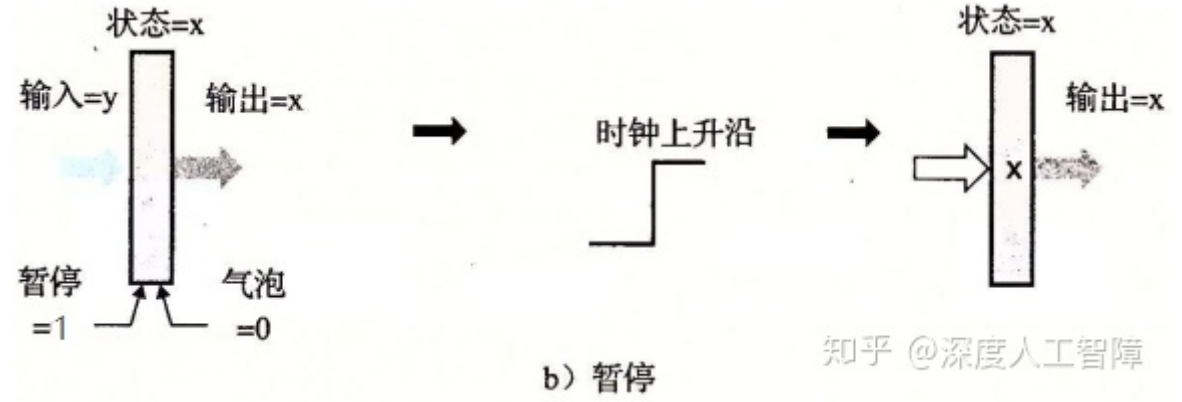
对于正常状态，即不是用暂停和bubble时，只要时钟电平变高，就会将流水线寄存器的状态修改为输入值，并作为新的输出。



- **暂停**

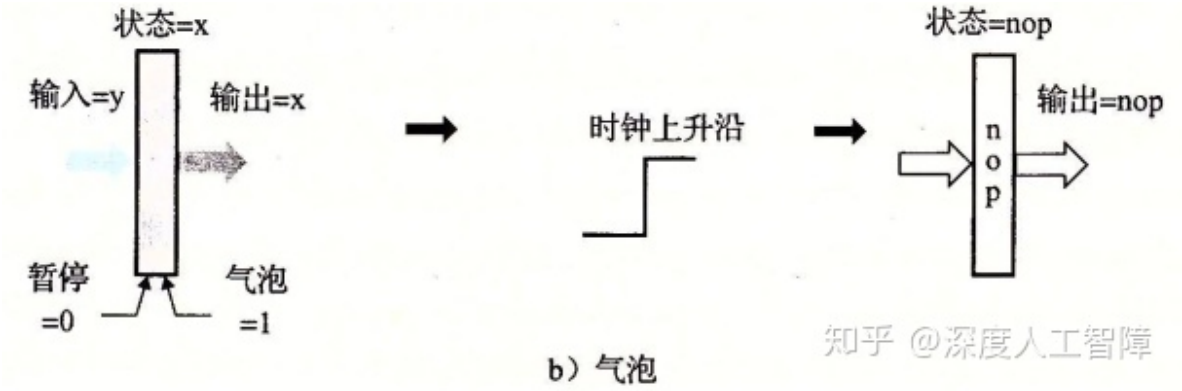
通过加入流水线寄存器，我们将指令的执行划分成了不同的阶段，并且每个阶段的输入就是流水线寄存器中的内容，所以如果我们想要将指令暂停在某个阶段时，我们可以直接将该阶段的流水线寄存器固定不变，使得该阶段的输入信息保持不变，就能在该阶段反复地执行指令，就是的指令阻塞在当前阶段了。

所以将指令暂停在某个阶段，就是当时钟电平变高时，保持该阶段的流水线寄存器的状态不变



- **bubble**

当时钟电平变高时，上一阶段指令的执行结果会保存到当前阶段的流水线寄存器，执行当前阶段后就会修改程序员可见状态，当我们想要保持程序员可见状态不变，可以插入一个bubble，使得寄存器状态设置成某个固定的复位配置，得到一个等效于`nop`指令的状态，相当于取消指令的运行

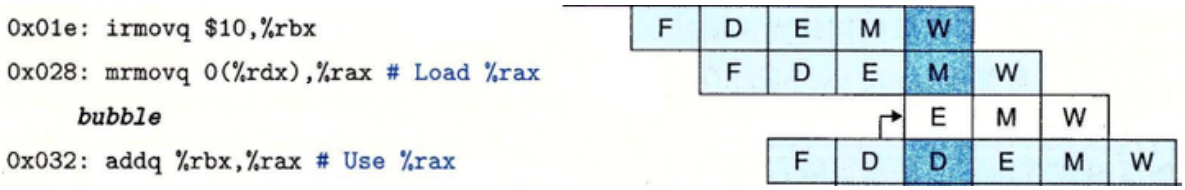


4.2 加载/使用冒险

`mrmovq`和`popq`指令`I1`会从内存中读取值保存到寄存器中，但是是在访存阶段才会读取到内存的值，所以如果下一条指令`I2`会读取这个寄存器的值，就会出现加载/使用冒险，因为当`I2`处于译码阶段读取寄存器值时，`I1`还是处于执行阶段，所以无法读取到内存的值。**触发条件**为

```
E_icode in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB}
```

理想处理方式为：固定流水线寄存器D和F，使得指令`I2`和下一条指令`I3`能分别阻塞在译码阶段和取指阶段，然后在译码阶段后面插入一个时钟周期的bubble，使得`I1`和前面的指令可以继续向后执行一个时钟周期，则`I1`此时处于访存阶段，就能读取到内存的值了。



所以当触发了加载/使用冒险时，流水线寄存器会如下设置一个时钟周期

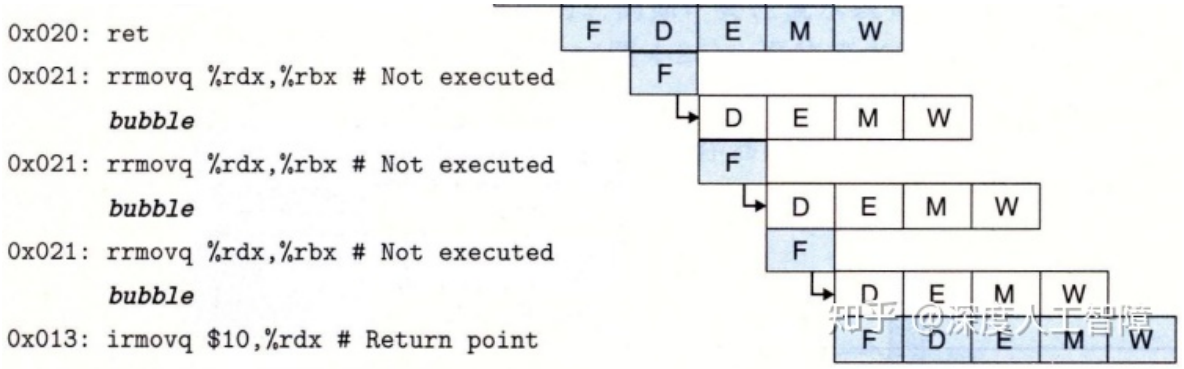
F	D	E	M	W
暂停	暂停	Bubble	正常	正常

4.2 处理`ret`指令

执行`ret`指令时，会从栈中读取返回地址作为下一条指令的地址，所以当`ret`执行到访存阶段时，才能读取到下一条指令的地址，然后在写回阶段的时钟电路变成高电平时，才会将其写入流水线寄存器M中，然后将`M_valM`传回去到`Select PC`逻辑模块。**触发条件**为：

```
IRET in {D_icode, E_icode, M_icode}
```

理想处理方式为：当`ret`执行到译码阶段时，会触发触发条件，此时就固定流水线寄存器F，就能保持不断读取下一条指令`I2`，并且后面在译码阶段插入3个时钟周期的bubble（根据取指阶段的HCL，会不断执行`valP`的错误指令，但是通过插入bubble，使得它只能执行到取指阶段），使得`ret`指令能向后执行3个时钟周期到达写回阶段，此时就能直接通过`W_valM`获得下一个PC的地址。



所以当触发了`ret`指令时，流水线寄存器会如下设置3个时钟周期

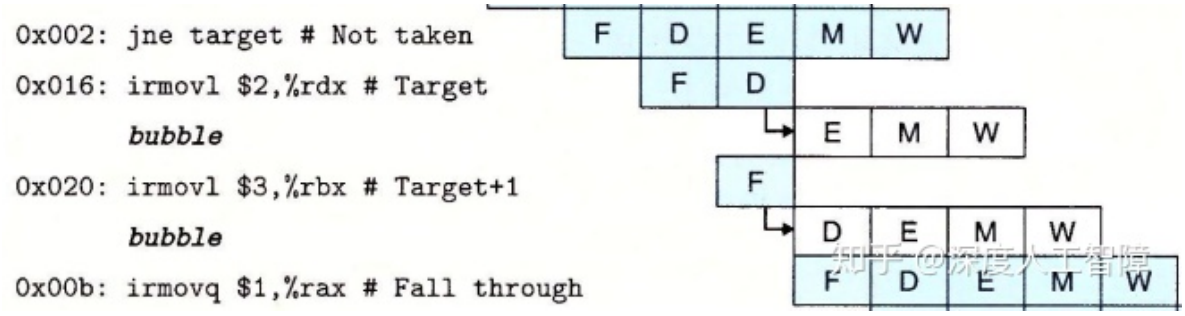
F	D	E	M	W
暂停	Bubble	正常	正常	正常

4.3 预测错误的分支

我们采用AT分支预测策略，所以当遇到条件分支指令`I1`时，会直接跳转到对应的地址开始执行，只有当`I1`执行到执行阶段时，才能通过`e_Cnd`判断是否发生跳转，此时已经执行了后续的两个指令`I2`和`I3`，分别处于译码阶段和取指阶段。预测错误的**触发条件**为：

```
E_icode == IJXX && !e_Cnd
```

当出现预测错误时，说明我们并不需要执行已经执行了的`I2`和`I3`指令，**理想的处理方式**为：直接在译码阶段插入bubbl中断`I3`，在执行阶段插入bubble中断`I2`，然后将正确的指令放入取指阶段开始执行，所以分支预测错误最多损耗两个时钟周期。



所以当触发了预测错误的分支时，流水线寄存器就会如下设置一个时钟周期

F	D	E	M	W
正常	Bubble	Bubble	正常	正常

4.4 异常指令

当出现`halt`指令、错误的指令码和函数码组合的指令或内存地址错误时，就会出现异常，所以异常通常在取指阶段和访存阶段被发现，对于异常**理想的处理方式**为：异常指令之前的指令都能完成，之后的指令都不会修改程序员可见状态，异常指令到达写回阶段时停止执行。

但是存在以下**困难**：异常在取指阶段和访存阶段被发现，程序员可见状态在执行阶段、访存阶段和写回阶段被修改。

我们首先在所有阶段的流水线寄存器中都包含一个程序状态信号`stat`，即使出现了异常，也只是将其当做普通信号传到下一阶段。当异常指令到达访存阶段时，后续的三条指令分别处于执行阶段、译码阶段和取指阶段，只有处于执行阶段的指令会修改条件码寄存器，所以要禁止执行阶段中的指令设置条件码。并且在访存阶段插入bubble，使得异常指令执行到写回阶段时，下一条指令就阻塞在执行阶段，不会到达访存阶段来修改内存。由于流水线处理器是按顺序处理指令的，所以第一次在写回阶段检测到异常指令就是最新的异常，所以只要在写回阶段检测到异常指令，就暂停写回，并暂停流水线。

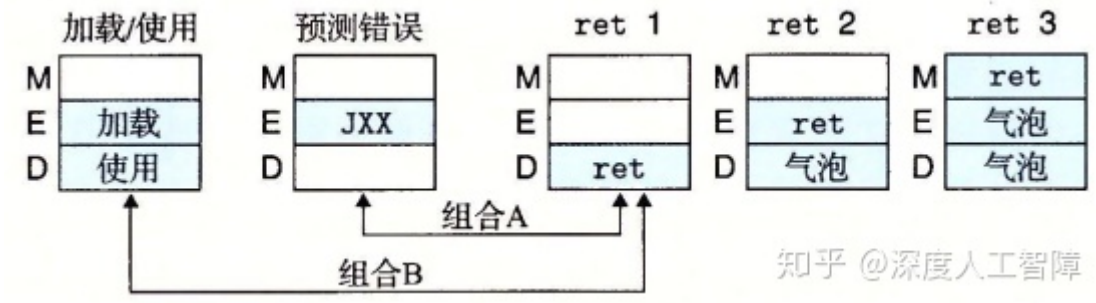
触发条件为：

```
m_stat in {SADR, SINS, SHLT} || W_stat in {SADR, SINS, SHLT}
```

4.5 特殊情况组合

以上讨论都假设了一个时钟周期只发生一种特殊情况，我们还需要讨论在一个时钟周期中是否可能发生同时出现多个特殊情况。

下图为出现各种特殊情况是流水线寄存器的情况，我们可以大多数情况中特殊情况的触发条件都是互斥的，但是有两种组合可能会出现。



- **组合A**：

预测错误的触发条件为

```
E_icode == IJXX && !e_Cnd
```

`ret`的触发条件为

```
IRET in {D_icode, E_icode, M_icode}
```

所以预测错误可能和`ret1`同时出现。

当执行阶段为条件分支，而译码阶段为`ret`时，由于预测分支采用了AT策略，说明`ret`指令是条件分支跳转后的第一条指令。此时由于预测错误，说明我们不应该执行`ret`指令，则理想的处理方式是对执行阶段插入bubble来取消`ret`的执行。并且当预测错误时，PC选择逻辑会选择`E_valA`，所以不用考虑流水线寄存器F采取的动作。

我们组合两种特殊情况的流水线控制动作，可以发现在E中使用了bubble符合理想的处理方式，兵器在取指阶段使用暂停不会造成影响。

条件	流水线寄存器				
	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
预测错误的分支	正常	气泡	气泡	正常	正常
组合	暂停	气泡	气泡	正常	正常

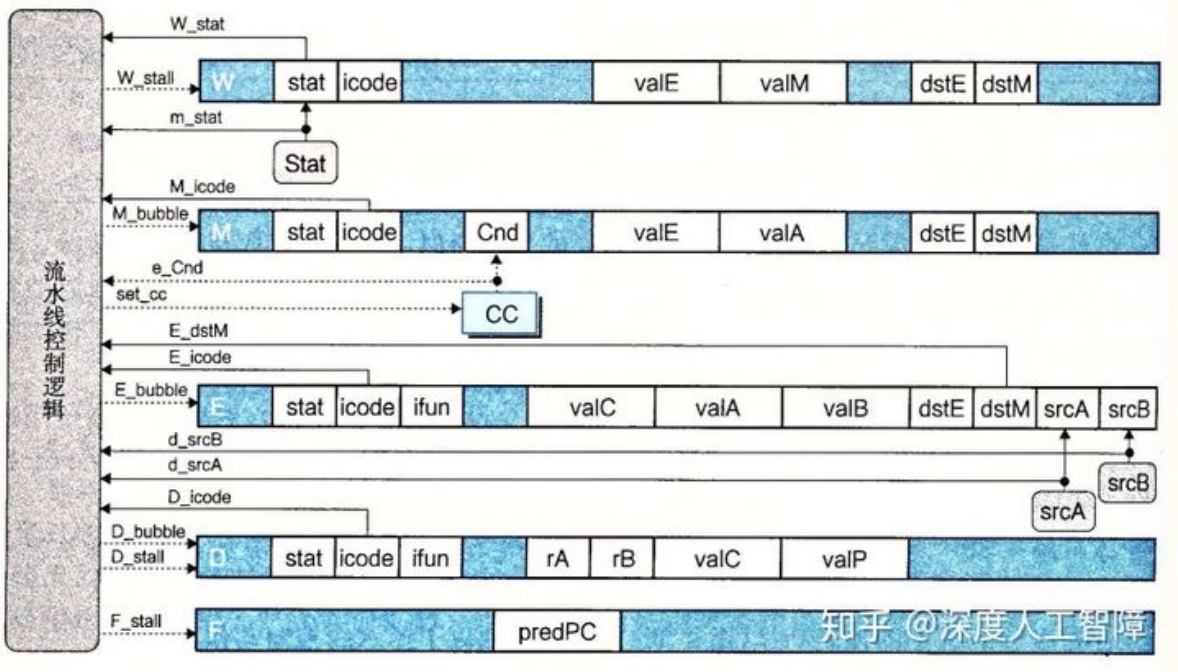
- **组合B**:

`ret`指令在译码阶段会读取`%rsp`的值，如果我们上一条指令是从内存中读取值来设置`%rsp`，则会出现加载/使用冒险和`ret1`的组合。

我们需要优先保证处理加载/使用冒险，来对`%rsp`进行设置，然后再处理`ret`指令，所以需要将`ret`指令阻塞在译码阶段，所以对流水线寄存器D需要使用暂停。我们组合两种特殊情况的流水线控制动作

条件	流水线寄存器				
	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
加载/使用冒险	暂停	暂停	气泡	正常	正常
组合	暂停	气泡＋暂停	气泡	正常	正常
期望的情况	暂停	暂停	气泡	正常	正常

4.6 流水线控制逻辑的HCL代码



如上图所示是流水线控制逻辑的整体结构，我们依次来说明对应的HCL代码

```
#取指阶段
##是否对F进行暂停
bool F_stall =
  #出现加载/使用冒险时
  E_icode in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB} || #加载/使用冒险的触发条件
  #出现ret指令时
  IRET in {D_icode, E_icode, M_icode}; #ret的触发条件
##F是不会插入bubble的
bool F_bubble = 0;

#译码阶段
##是否对D进行暂停
bool D_stall =
  #出现加载/使用冒险时
  E_icode in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB};
##是否对D插入bubble
bool D_bubble =
  #出现预测错误时
  (E_icode == IJXX && !e_Cnd) || #预测错误的触发条件
  #出现了ret，但是没有出现加载/使用冒险，（组合B）
  !(E_icode in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB}) &&
  IRET in {D_icode, E_icode, M_icode};

#执行阶段
##因为没有特殊情况对E进行暂停，所以直接为0
bool E_stall = 0;
##是否对E插入bubble
bool E_bubble =
  #出现预测错误时
  (E_icode == IJXX && !e_Cnd) ||
  #出现加载/使用冒险时
  E_icode in {IMRMOVQ, IPOPQ} && E_dstM in {d_srcA, d_srcB};
##当出现异常时，我们需确保后续指令不会对条件码寄存器进行修改
bool set_cc = E_icode == IOPQ && #首先要保证当前指令是算数指令，才会设置CC
  !m_stat in {SADR, SINS, SHLT} && #保证上一条处在访存阶段的指令没有出现异常
  !W_stat in {SADR, SINS, SHLT}; #保证上两条指令没有出现异常

#访存阶段
##没有指令会在访存阶段进行暂停
bool M_stall = 0;
##当之前的指令出现异常时，我们需要插入bubble来取消指令，防止对内存进行修改
bool M_bubble = m_stat in {SADR, SINS, SHLT} || W_stat in {SADR, SINS, SHLT};

#写回阶段
#没有指令会插入bubble
bool W_bubble = 0;
#当出现异常时，会在写回阶段暂停
bool W_stall = W_stat in {SADR, SINS, SHLT};
```

4.7 性能分析

我们通过计算PIPE执行一条指令所需的平均时钟周期来衡量系统的性能，该指标为**CPI（Cycles Per Instruction）**。而影响该指标的因素是流水线中插入bubble的数目，因为插入bubble时就会损失一个流水线周期。

我们可以在处理器中运行某个基准程序，然后统计执行阶段中运行的指令数 C_i 和bubble数 C_b ，就得到对应的CPI指标

$$CPI = \frac{C_i + C_p}{C_i} = 1 + \frac{C_p}{C_i}$$

由于只有三种特殊请款（加载/使用冒险、预测错误、`ret`指令）会插入bubble，所以我们可以将惩罚项 $\frac{C_b}{C_i}$ 分为三部分，`lp`表示加载/使用冒险插入bubble的平均数，`mp`表示预测错误插入bubble的平均数，`rp`表示`ret`指令插入bubble的平均数，则CPI可变为

$$CPI = 1.0 + lp + mp + rp$$

我们可以根据指令出现的频率以及出现特殊情况的频率对CPI进行计算

- `mrmovq`和`popq`占所有执行指令的25%，其中20%会导致加载/使用冒险
- 条件分支指令栈所有执行指令的20%，使用AT策略会有60%的成功率
- `ret`指令栈所有执行指令的2%

由此我们可以得到三种特殊情况的惩罚

原因	名称	指令频率	条件频率	气泡	乘积
加载/使用	lp	0.25	0.20	1	0.05
预测错误	mp	0.20	0.40	2	0.16
返回	rp	0.02	1.00	3	0.06
总处罚					0.27

5 额外内容

5.1 多周期指令

我们提供的Y86-64指令集只有简单的操作，在执行阶段都能在一个时钟周期内完成，但是如果要实现整数乘法和除法以及浮点数运算，我们首先要增加额外的硬件来执行这些计算，并且这些指令在执行阶段通常都需要多个时钟周期才能完成，所以执行这些指令时，我们需要平衡流水线各个部分之间的关系。

实现多周期指令的简单方法是直接暂停取指阶段和译码阶段，直到执行阶段执行了所需的时钟周期后才恢复，这种方法的性能通常比较差。

常见的方法是使用独立于主流水线的特殊硬件功能单元来处理复杂的操作，通常会有一个功能单位来处理整数乘法和除法，还有一个功能单位来处理浮点数运算。在译码阶段中遇到多周期指令时，就可以将其发射到对应的功能单元进行运算，而主流水线会继续执行其他指令，使得多周期指令和其他指令能在功能单元和主流水线中并发执行。但是如果不同功能单元以及主流水线的指令存在数据相关时，就需要暂停系统的某部分来解决数据冒险。也同样可以使用暂停、转发以及流水线控制。

5.2 与存储系统的接口

我们假设了取指单元和数据内存都能在一个时钟周期内读写内存中的任意位置，但是实际上并不是。

1. 处理器的存储系统是由多种硬件存储器和管理虚拟内存的操作系统共同组成的，而存储系统包含层次结构，最靠近处理器的一层是**高速缓存（Cache）存储器**，能够提供对最常使用的存储器位置的快速访问。典型系统中包含一个用于读指令的cache和一个用于读写数据的cache，并且还有一个**翻译后备缓冲器（Translation Look-aside Buffer，TLB）**来提供从虚拟地址到物理地址的快速翻译。将TLB和cache结合起来，大多数时候能再一个时钟周期内读指令并读写数据。
2. 当我们想要的引用位置不在cache中时，则出现高速缓存**不命中（Miss）**，则流水线会将指令暂停在取指阶段或访存阶段，然后从较高层次的cache或处理器的内存中找到不命中的数据，然后将其保存到cache中，就能恢复指令的运行。这通常需要3~20个时钟周期。
3. 如果我们没有从较高层次的cache或处理器的内存中找到不命中的数据，则需要从磁盘存储器中寻找。硬件会先产生一个**缺页（Page Fault）**异常信号，然后调用操作系统的异常处理程序代码，则操作系统会发起一个从磁盘到主存的传送操作，完成后操作系统会返回原来的程序，然后重新执行导致缺页异常的指令。其中访问磁盘需要数百万个时钟周期，操作系统的缺页中断处理程序需要数百个时钟周期。

5.3 当前的微处理器设计

我们采用的五阶段流水线设计，吞吐量都现在最多一个时钟周期执行一条指令，CPI测量值不可能小于1.0。较新的处理器支持**超标量（Superscalar）**操作，能够在取值、译码和执行阶段并行处理多条指令，使得CPI测量值小于1.0，通常会使用IPC（每个周期执行的指令数）来测量性能。更新的处理器会支持**乱序（Out-of-order）**技术，对指令执行顺序进行打乱来并行执行多条指令。