# 1 实验四　流水线 MIPS 处理器设计

在实验三中已介绍过 MIPS 传统的五级流水线阶段，分别是取指、译码、执行、访存、回写 *(Instruction Fetch, Decode, Execution, Memory Request, Write Back)*，五阶段。

单周期 CPU 虽然 CPI 为 1，但由于时钟周期取决于时间最长的指令 (如 lw、sw) 没有很好的性能，而多周期虽然能提升性能但仍旧无法满足当今处理器的需求。流水线能够很好地解决效率问题，通过分阶段，达到指令的并行执行。同时，在单周期的基础上，能够很容易地使用触发器做阶段分隔，实现流水线。

本次实验将从实验三单周期处理器过渡至五级流水，并将解决冒险 (hazard) 问题。

## 1.1 实验目的

1. 掌握流水线 (Pipelined) 处理器的思想；
2. 掌握单周期处理中执行阶段的划分；
3. 了解流水线处理器遇到的冒险；
4. 掌握数据前推、流水线暂停等冒险解决方式。

## 1.2 实验设备

1. 计算机 1 台 (尽可能达到 8G 及以上内存)；
2. Xilinx Vivado 开发套件 (2019.1 版本)。

注：本次实验为 CPU 软核实验，不涉及开发板外围设备，故不需要开发板

## 1.3 实验任务

### 1.3.1 实验要求

阅读实验原理实现以下模块：

(1) Datapath，所有模块均可由实验三复用，需根据不同阶段，修改 mux2 为 mux3(三选一选择器)，以及带有 enable(使能)、clear(清除流水线) 等信号的触发器，
(2) Controller，其中 main decoder 与 alu decoder 可直接复用，另需增加触发器在不同阶段进行信号传递
(3) 指令存储器 inst_mem(Single Port Ram)，数据存储器 data_mem(Single Port Ram)；同实验三一致，无需改动，
(4) 参照实验原理，在单周期基础上加入每个阶段所需要的触发器，重新连接部分信号。实验给出 top 文件，需兼容 top 文件端口设定。
(5) 实验给出仿真程序，最终以仿真输出结果判断是否成功实现要求指令。

### 1.3.2 实验步骤

1. 在实验三基础上，加入 D 触发器，增加流水级数；
2. 处理流水线冒险问题，加入暂停和前推机制；
3. 导入顶层文件及仿真文件，运行仿真；

## 1.4 实验环境

| | |
|---|---|
| |–top.v | 设计顶层文件，已提供。 |
| |—-mips.v | MIPS 软核顶层文件，将 Controller 与 Datapath 连接，已提供 |
| |———controller.v | 控制器模块，本次实验重点。 |
| |———maindec.v | Main decoder 模块，负责译码得到各个组件的控制信号。 |
| |———aludec.v | ALU Decoder 模块，负责译码得到 ALU 控制信号。 |
| |———datapath.v | 数据通路模块，自行实现 |
| |———pc.v | PC 模块，使用实验二代码 |
| |———alu.v | ALU 模块，使用实验一代码 |
| |———sl2.v | 移位模块，参考《其他组件实现.pdf》 |
| |———signext.v | 有符号扩展模块，参考《其他组件实现.pdf》 |
| |———mux2.v | 二选一选择器，自行实现 |
| |———regfile.v | 寄存器堆，已提供 |
| |———adder.v | 加法器，已提供 |
| |———floprc | 带有 reset 与 clear 的触发器 |
| |———flopenr | 带有 enable 与 reset 的触发器 |
| |———flopenrc | 带有 enable、reset 与 clear 的触发器 |
| |———hazard | 冒险处理模块，自行实现 |
| |—-*inst_ram.ip* | *RAM IP*，通过 *Block memory generator* 进行实例化 |
| |—-*data_ram.ip* | *RAM IP*，通过 *Block memory generator* 进行实例化 |

**表 1:** 实验文件树

# 2 实验原理

## 2.1 单周期改流水线原理

实验三已完成图中单周期的部分，可以看到，流水线处理器的主要改动，是在每个执行阶段加入触发器，使得每个周期执行一个阶段，得到的结果送往下一个周期进行执行，同时下一条指令执行一个阶段，这样能够使指令各阶段并行执行，提升效率。

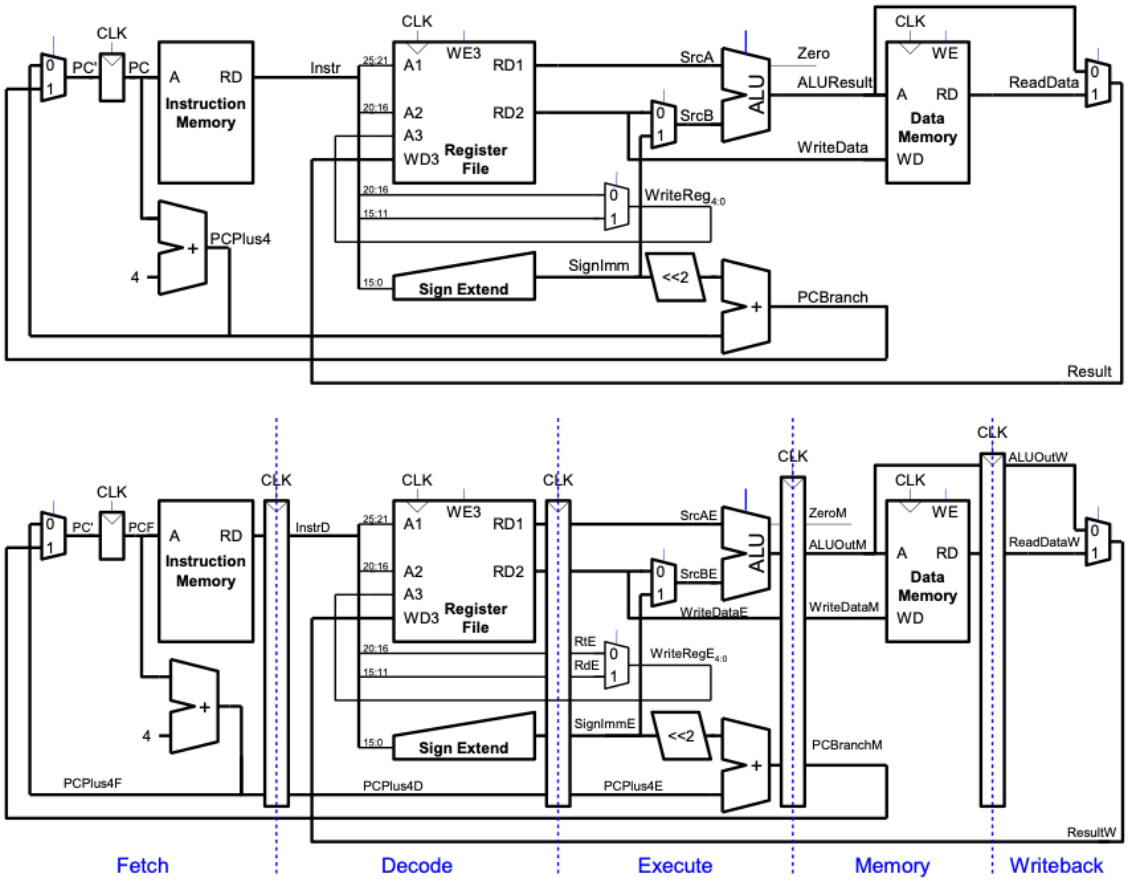


图 1: 单周期和流水线比较

可以看到，单周期中写寄存器堆的地址信号 writereg 需要延迟到 writeback 阶段与回写数据 result 一起写回寄存器堆：

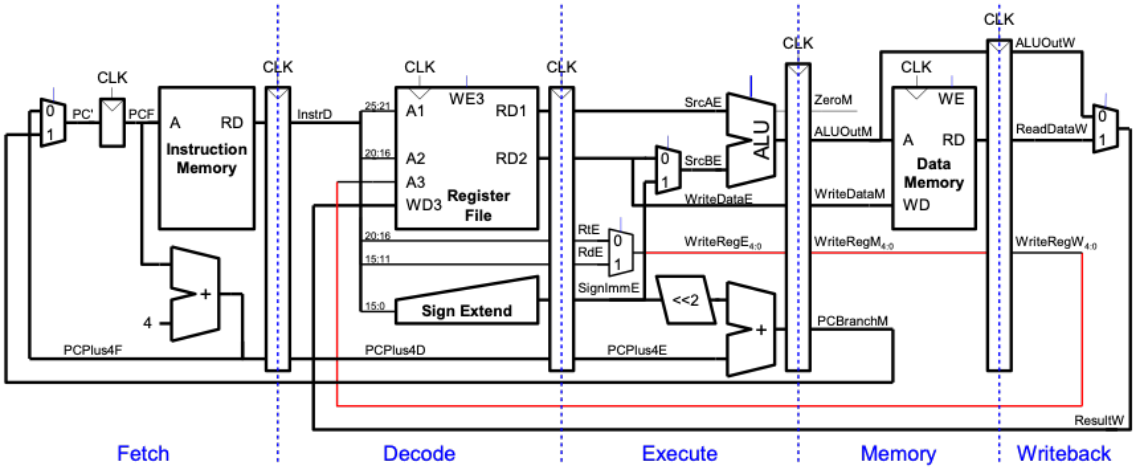


图 2: 修改 writereg 信号

在此基础上，datapath 的基本通路已经形成，下面加入控制器部分。控制器部分与单周期相同，仍然由 main decoder 和 alu decoder 构成，但由于改为五级流水线后，每一个阶段所需要的控制信号仅为一部分，控制器产生信号的阶段为译码阶段，产生控制信号后，依次通过触发器传到下一阶段，若当前阶段需要的信号，则不需要继续传递到下一阶段：

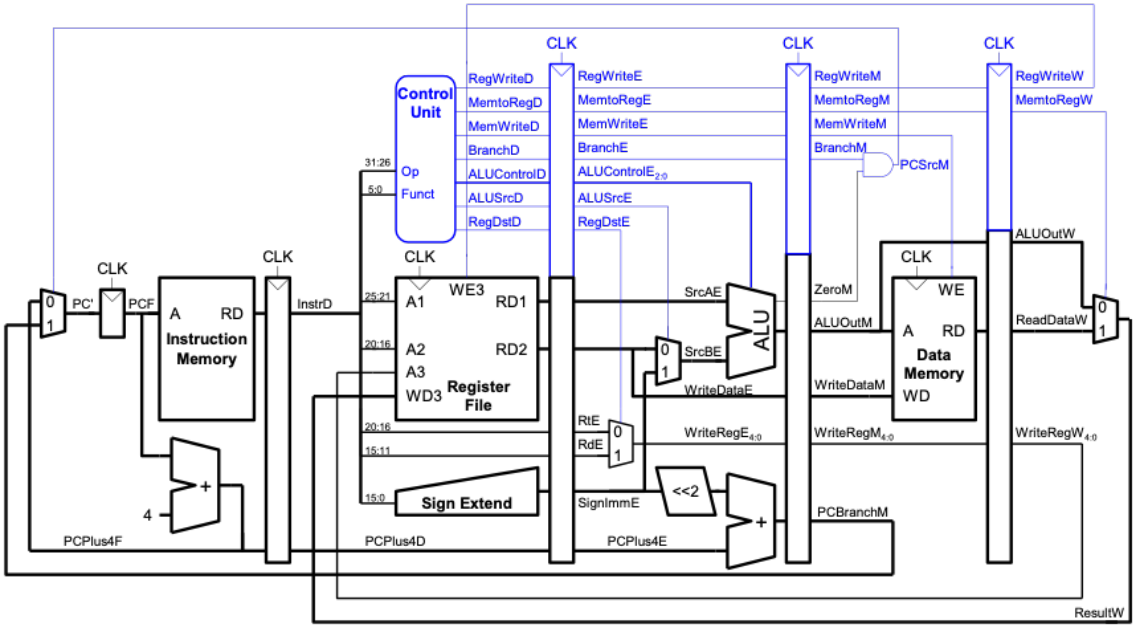


图 3: 加入控制器的流水线示意图

## 2.2 各类型触发器的实现

实验三中已给出触发器 flopr，作为 PC 使用，此外需要在其基础上实现下列触发器，：

- flopr：带有 reset 的触发器
- floprc：带有 reset 与 clear 的触发器
- flopenrc：带有 enable、reset 与 clear 的触发器

flopr 的写法如下：

**Listing 1:** flopr 实现

```
module flopr #(parameter WIDTH = 8)(
  input wire clk, rst,
  input wire[WIDTH-1:0] d,
  output reg[WIDTH-1:0] q
    );

  always @(posedge clk,posedge rst) begin
    if(rst) begin
      q <= 0;
    end else begin
      q <= d;
    end
  end
endmodule
```

floprc 的写法如下：

**Listing 2:** floprc 实现

```
module floprc #(parameter WIDTH = 8)(
  input wire clk,rst,clear,
  input wire[WIDTH-1:0] d,
  output reg[WIDTH-1:0] q
    );

  always @(posedge clk,posedge rst) begin
    if(rst) begin
      q <= 0;
    end else if (clear)begin
      q <= 0;
    end else begin
      q <= d;
    end
  end
endmodule
```

flopenrc 的写法如下：

**Listing 3:** flopenrc 实现

```
module flopenrc #(parameter WIDTH = 8)(
  input wire clk,rst,en,clear,
  input wire[WIDTH-1:0] d,
  output reg[WIDTH-1:0] q
```

```
    );
  always @(posedge clk) begin
    if(rst) begin
      q <= 0;
    end else if(clear) begin
      q <= 0;
    end else if(en) begin
      /* code */
      q <= d;
    end
  end
endmodule
```

注意 **#(parameter WIDTH = 8)** 的写法，这样写，在调用时可以指定宽度：

**Listing 4:** 带参数实例化

```
flopenrc #(32) r1M(clk,rst,~stallM,flushM,srcb2E,writedataM);
flopenrc #(32) r2M(clk,rst,~stallM,flushM,aluout2E,aluoutM);
flopenrc #(5)  r3M(clk,rst,~stallM,flushM,writereg2E,writeregM);
```

## 2.3 冒险 (hazard) 的解决

在流水线 CPU 中，并不是能够完全实现并行执行。在单周期中由于每条指令执行完毕才会执行下一条指令，并不会遇到冒险问题，而在流水线处理器中，由于当前指令可能取决于前一条指令的结果，但此时前一条指令并未执行到产生结果的阶段，这时候，就产生了冒险。

冒险分为：

1. 数据冒险：寄存器中的值还未写回到寄存器堆中，下一条指令已经需要从寄存器堆中读取数据；
2. 控制冒险：下一条要执行的指令还未确定，就按照 PC 自增顺序执行了本不该执行的指令 (由分支指令引起)。

### 2.3.1 数据冒险

分析图 4中指令，and、or、sub 指令均需要使用 $s0 中的数据，然而 add 指令在回写阶段才能写入寄存器堆，此时后续三条指令均已经过或正在执行译码阶段，得到的结果均为错误值。以上就是数据冒险的特点，数据冒险有以下解决方式：

1. 在编译时插入空指令；
2. 在编译时对指令执行顺序进行重排；
3. 在执行时进行数据前推；
4. 在执行时，暂停处理器当前阶段的执行，等待结果。

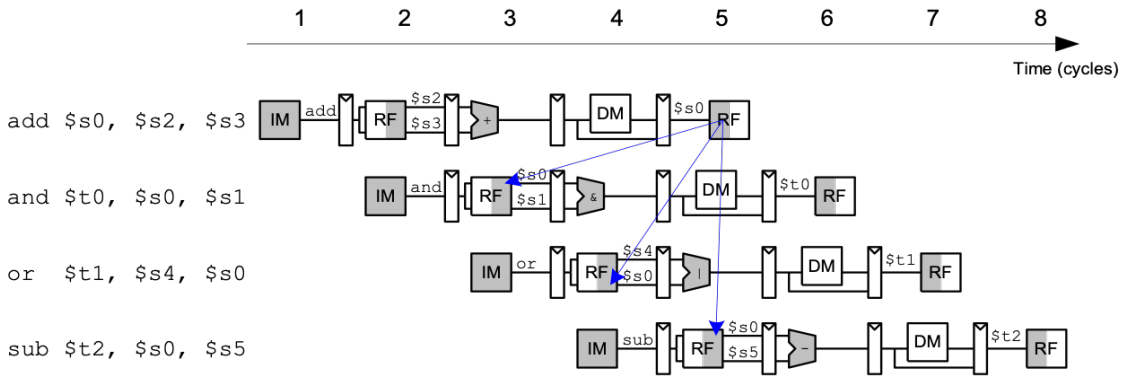


图 4: 数据冒险示例

由于我们未进行编译层的处理，需要在运行时 (run time) 进行解决，故采用 3、4 解决方案。

### 2.3.2 数据前推

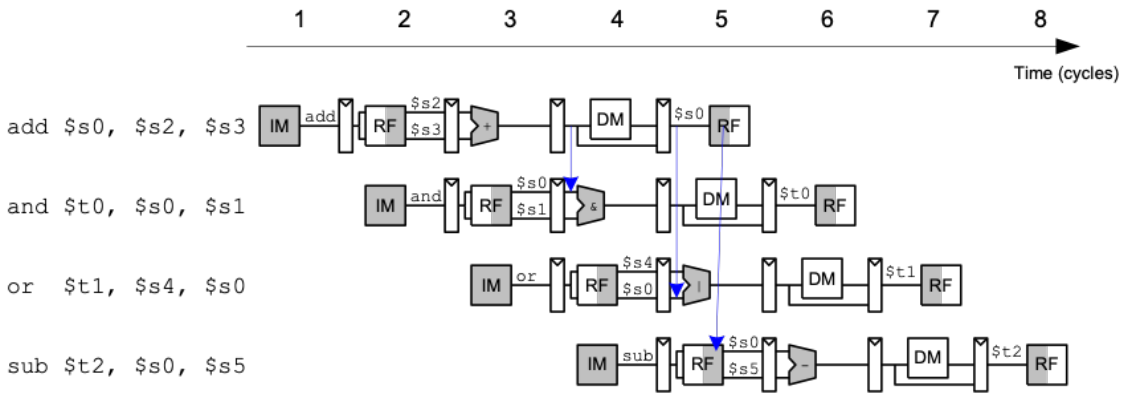


图 5: 数据前推示例

从图 5中可以看到，add 指令的结果在 execute 阶段已经由 ALU 计算得到，此时可以将 alu 得到的结果直接推送到下一条指令的 execute 阶段，同理，后续所有的阶段均已有结果，可以向对应的阶段推送，而不需要等到回写后再进行读取，达到数据前推的目的。

数据前推的实现逻辑如下:

Listing 5: 数据前推的实现逻辑

```
1  if  ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
2    then
3        ForwardAE = 10
4  else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
5    then
6        ForwardAE = 01
7  else
8      ForwardAE = 00
```

结合实现逻辑,观察下图。在 execute 阶段需要判断当前输入 ALU 的地址是否与其他指令在此时执行的阶段要写入寄存器堆的地址相同,如果相同,就需要将其他指令的结果直接通过多路选择器输入到 ALU 中。

此处需要:

1. 增加 rs,rt 的地址传递到 execute 阶段,并与冒险模块连接;
2. Memory 阶段和 writeback 阶段要写入寄存堆的地址与冒险模块连接;
3. Memory 阶段和 writeback 阶段的寄存器堆写使能信号 regwrite 与冒险模块连接;
4. 根据实现逻辑,将生成的 forward 信号输出,控制 mux3 选择器。

数据通路结构如下:

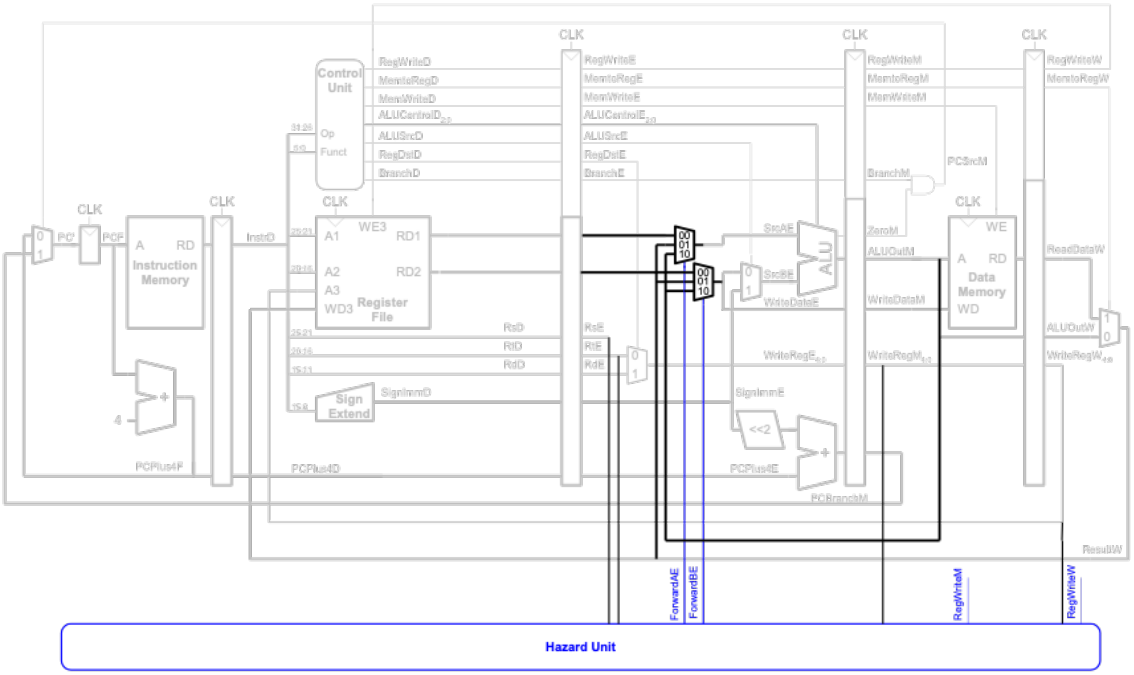


**图 6:** 加入前推的数据通路结构

### 2.3.3 流水线暂停

多数情况下,数据前推能解决很大一部分数据冒险的问题,然而在图中 7,lw 指令在 memory 阶段才能够从数据存储器读取数据,此时 and 指令已经完成 ALU 计算,无法进行数据前推。

如图 8在这种情况下,必须使流水线暂停,等待数据读取后,再前推到 execute 阶段。

流水线暂停的实现逻辑如下:

**Listing 6:** 流水线暂停的实现逻辑

```
1  lwstall = ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE
2  StallF = StallD = FlushE = lwstall
```

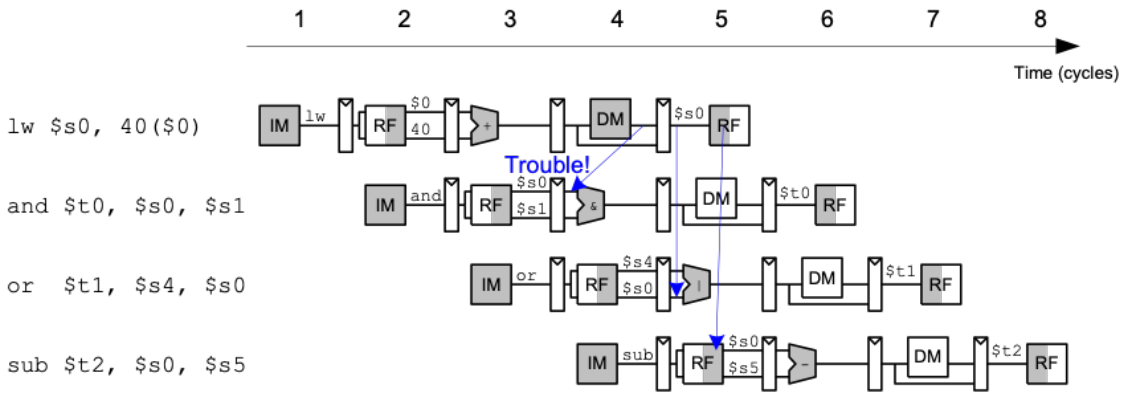


图 7: 数据无法前推的情况

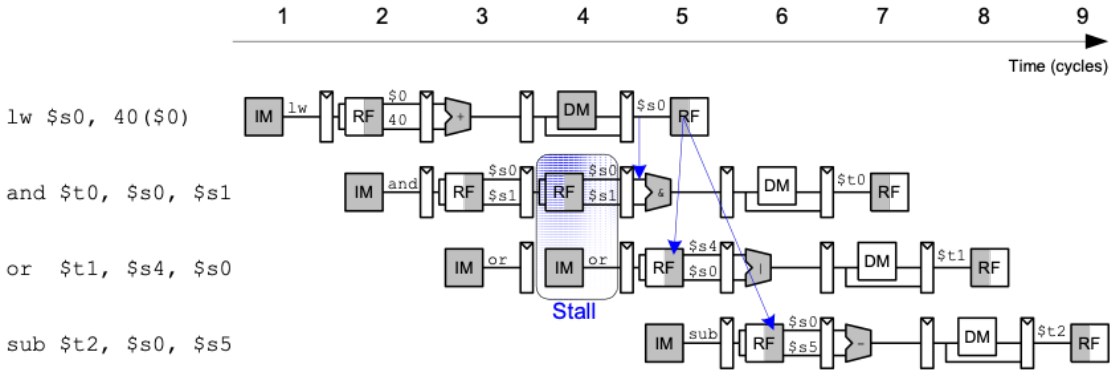


图 8: 流水线暂停示例

结合实现逻辑,需要完成下列功能:

1. 判断 decode 阶段 rs 或 rt 的地址是否是 lw 指令要写入的地址;
2. 设置 PC、fetch->decode 阶段触发器的暂停信号 (触发器使能端 disable);
3. Decode->exexcute 阶段触发器清除 (避免后续阶段的执行,等待完成后方可继续执行后续阶段)。

数据通路结构如下:

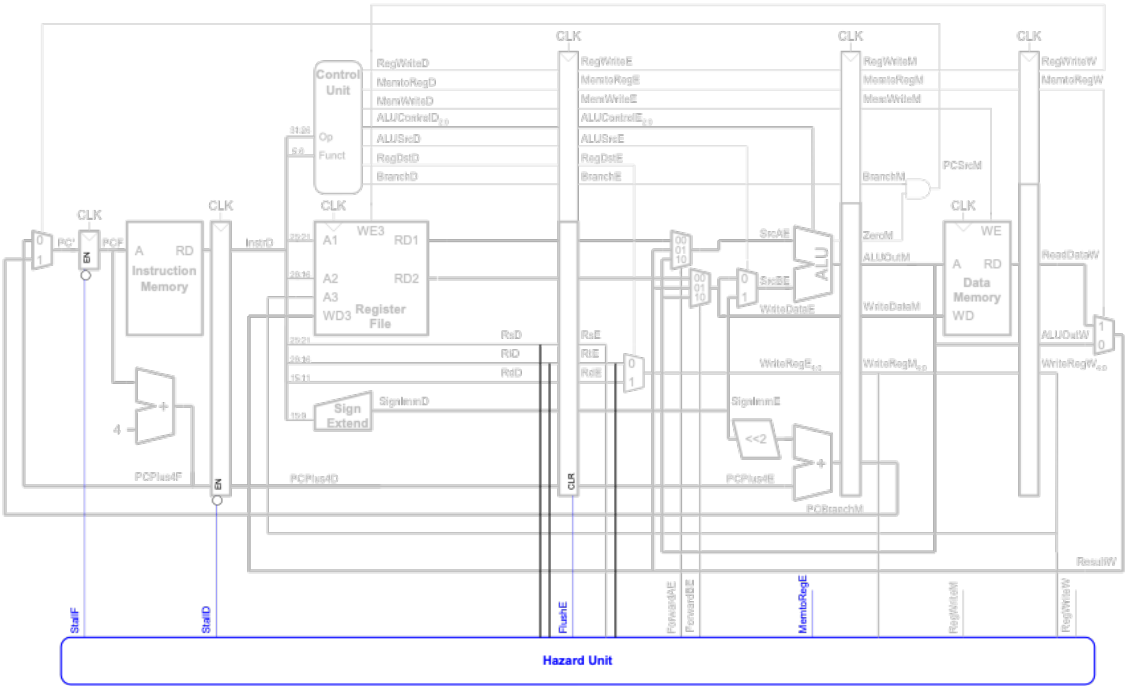


图 9: 加入暂停的数据通路结构

### 2.3.4 控制冒险

控制冒险是分支指令引起的冒险。在五级流水线当中，分支指令在第 4 阶段才能够决定是否跳转；而此时，前三个阶段已经导致三条指令进入流水线开始执行，这时需要将这三条指令产生的影响全部清除。

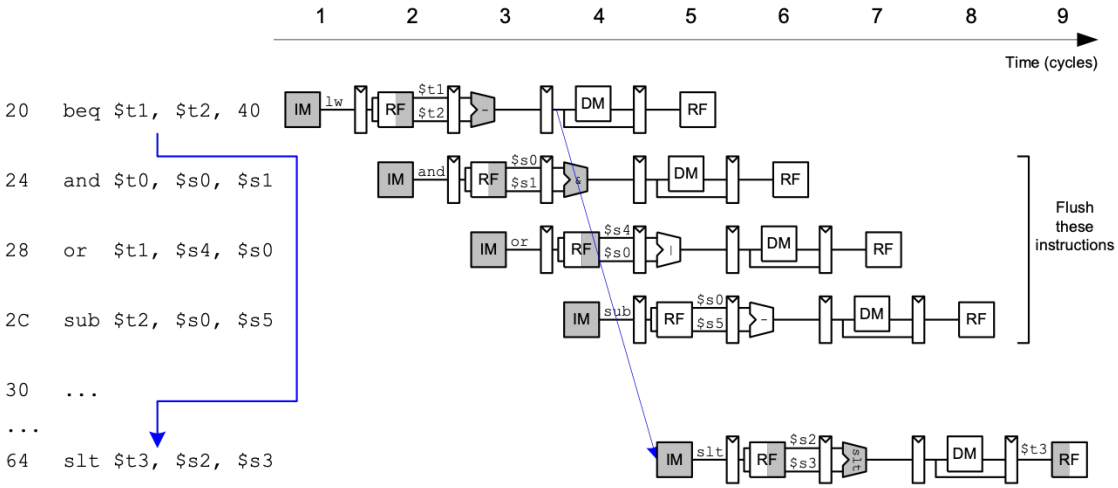


图 10: 控制冒险示例

将分支指令的判断提前至 decode 阶段，此时能够减少两条指令的执行；

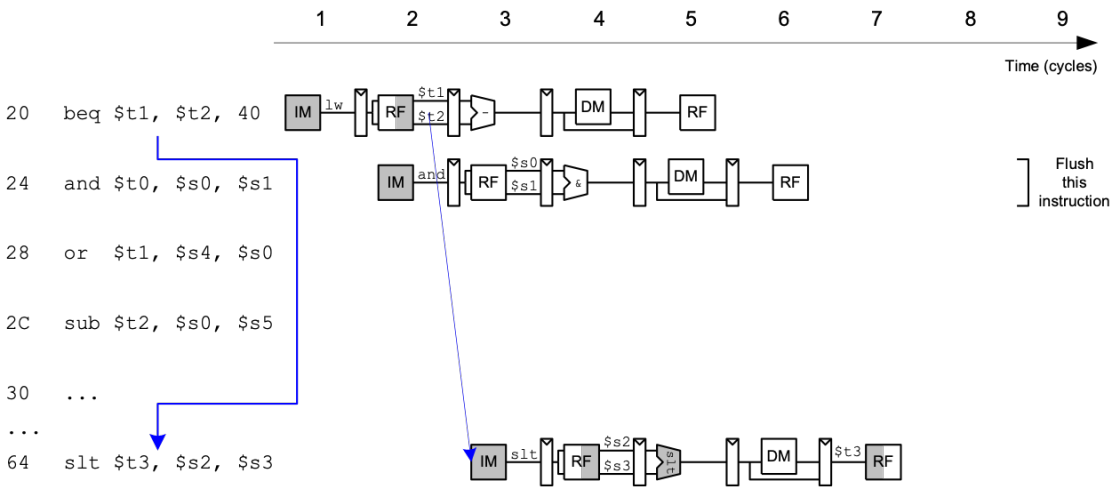


图 11: 提前判断分支

在 regfile 输出后添加一个判断相等的模块，即可提前判断 beq：

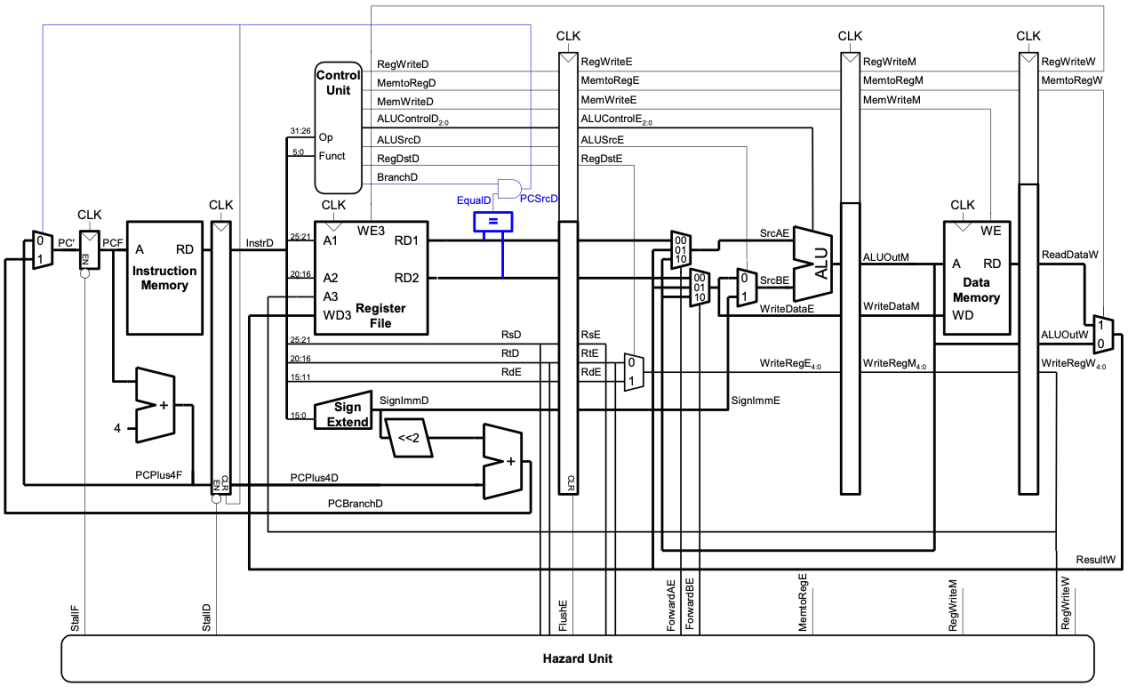


图 12: 提前判断分支的实现

此时又产生了数据冲突问题,需要增加数据前推和流水线暂停模块;

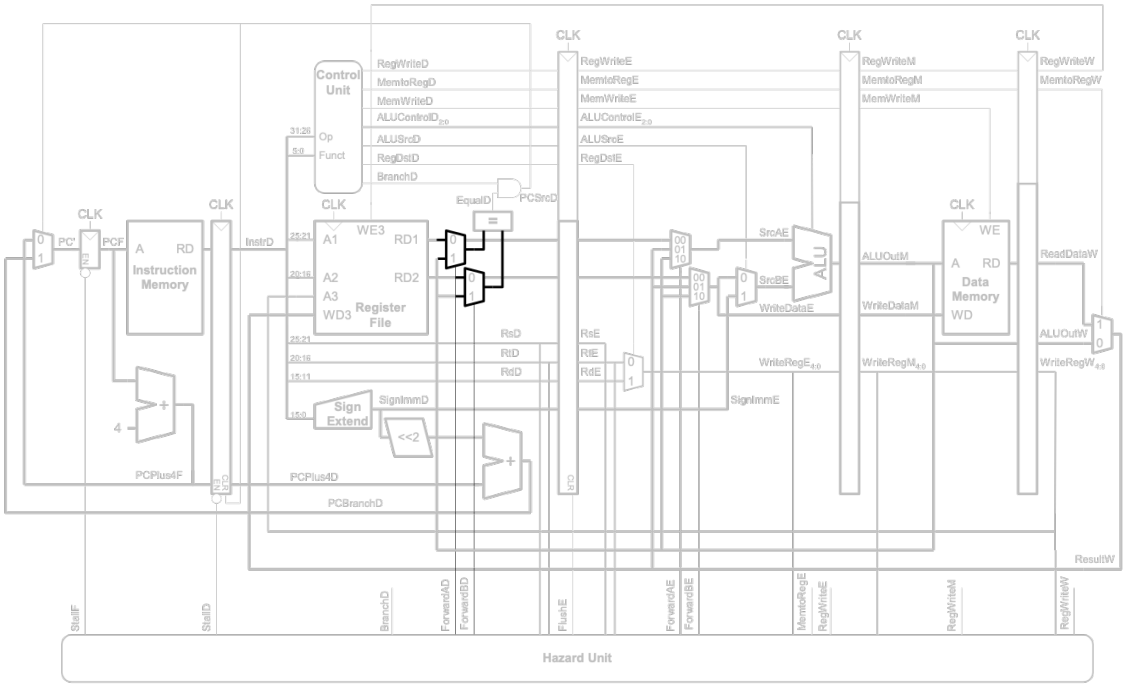


图 13: 解决控制冒险的数据通路

实现逻辑如下:Forwarding logic:

Listing 7: 控制冒险前推的实现逻辑

```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND RegWriteM
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND RegWriteM
```

Stalling logic:

Listing 8: 控制冒险暂停的实现逻辑

```
branchstall = BranchD AND RegWriteE AND
                (WriteRegE == rsD OR WriteRegE == rtD)
              OR BranchD AND MemtoRegM AND
                (WriteRegM == rsD OR WriteRegM == rtD)
StallF = StallD = FlushE = lwstall OR branchstall
```