南京大学计算机系
组成原理实验

Lab6
MIPS流水线CPU设计
设计手册

作者	张瑞祎
学号	121220139
项目组成员	张瑞祎，徐栋，李乾科
班级	4班
报告日期	2014年6月6日

目录

1　设计目标

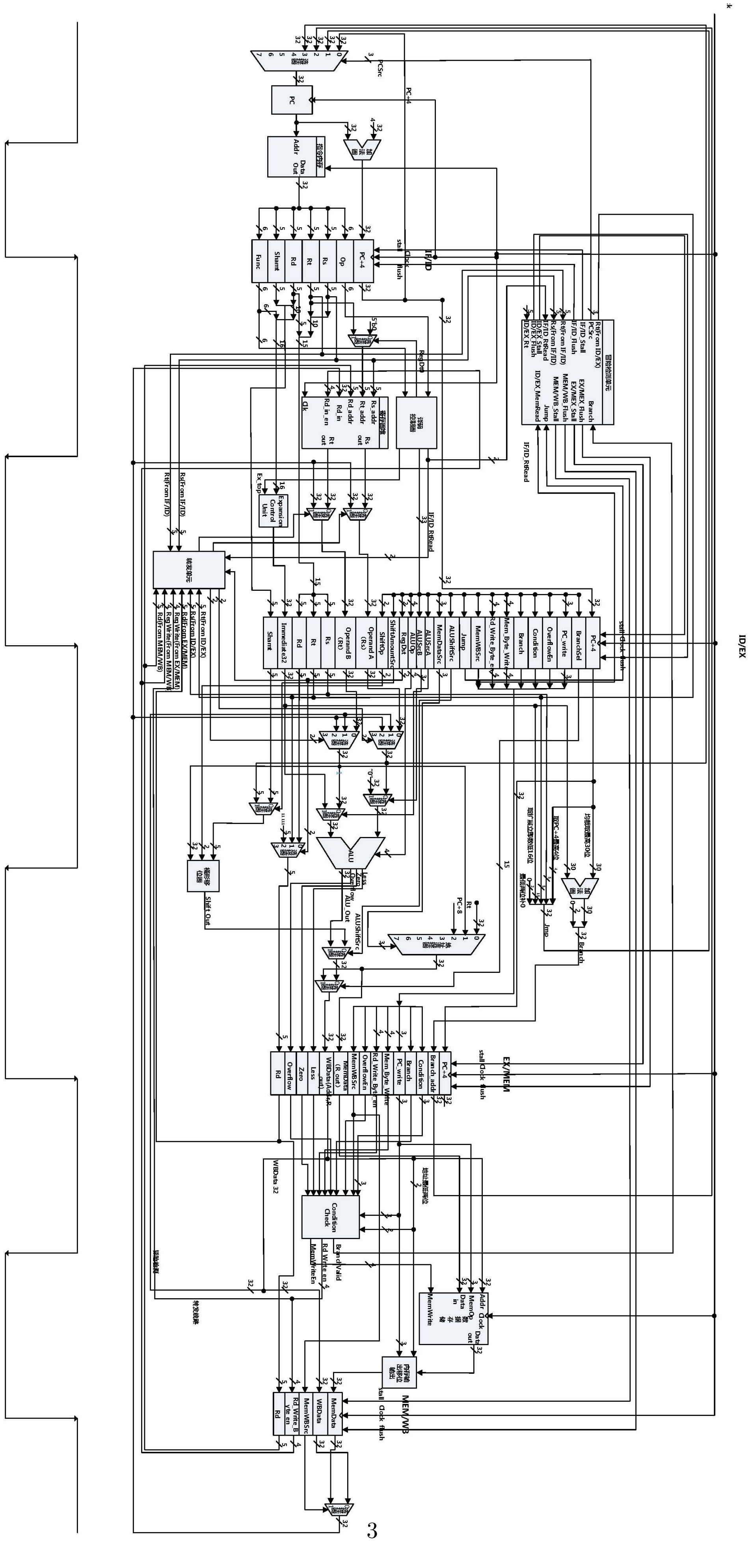
（1）设计一个具备基本功能的MIPS流水线的软核微处理器。
（2）理解数据冒险、控制冒险的原理并在设计中予以解决。
（3）对流水线MIPS微处理器进行的指令测试以及程序测试。

2　MIPS架构简介

MIPS架构20年多前由斯坦福大学开发，是一种简洁、优化、具有高度扩展性的R ISC架构。它的基本特点是：包含大量的寄存器、指令数和字符、可视的管道延时时隙，这些特性使MIPS架构能够提供最高的每平方毫米性能和当今SoC设计中最低的能耗。

3　设计原理图

设计原理图如图1所示。



IF/ID
ID/EX
EX/MEM
MEM/WB
PC
ALU

4 MIPS CPU各部件设计

MIPS流水线CPU采用经典的五段流水设计，分别是取指，译码，执行，访存和写回五个阶段。对于五个阶段具体描述如下：

(1)Ifetch(取指,Insturction Fetch)，从指令高速缓存(I-cache)获取下一条指令。
(2)Reg/Dec(取数和译码,Register Read,Decoder)，读取该指令的源寄存器域指定的CPU寄存器的内容。
(3)Ex (执行,Execute)，在一个时钟周期内ALU和桶形移位器完成算术或者逻辑操作。
(4)MEM(访问内存,Memory)，从数据存储器中读或向内存中写入数据。
(5)WB(写回寄存器,Write Back)，将操作结果值写到寄存器堆中。

每个阶段执行的时间均为一个时钟周期，当一个时钟上升沿到来时，我们将此段的内容暂存到段寄存器中并使其流向下一执行阶段。

4.1 IF段的设计

IF段主要包括两个部件，即PC（指令计数器）以及指令内存。PC与指令内存在始终下降沿写入，

比较需要特殊说明的设计处为：PC没有stall信号，但是整个段是可以被stall的，因为设计中，每次将IF/ID段中的PC+4值传到顺序转移的PC入口处，这样在stall有效时，将会使IF/ID段暂停，从而间接也使PC不在加4运算。毕竟阻塞PC目标就是阻塞IF/ID段。

IF段接收其他阶段的PC传入值，经选择后传入一个到PC中，指令内存写有时钟控制，读为纯组合电路，在这里，指令内存因为不会更改，人为将使能接口置为无效，Data_in赋值为32'b0；

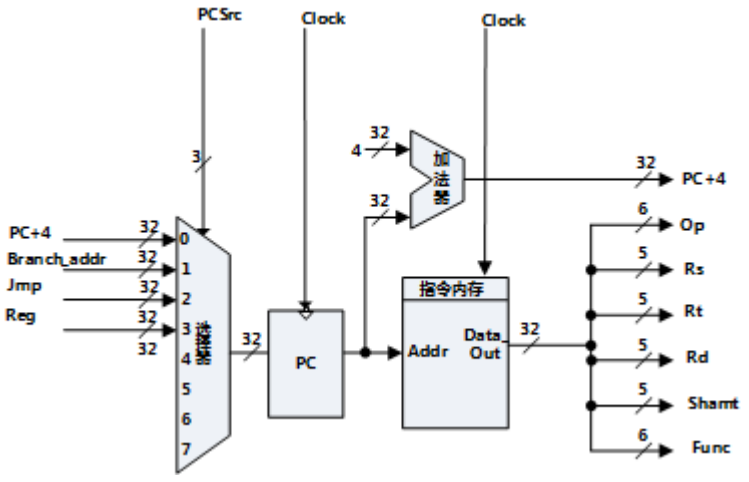


Figure 1: IF段

4.2 ID段的设计

ID段主要包括译码控制器，寄存器堆，立即数拓展三个模块。寄存器堆在始终下降沿时写入，读取为完全的组合电路。译码控制器为纯组合电路。

译码控制器根据指令生成控制信号，提供给冒险监测单元，同时在时钟上升沿，将其写入到ID/EX都寄存器。

寄存器堆包括2个读口和1个写口，写使能由四位使能信号控制，分别对应四个字节。

立即数拓展有三种方式，基本的模式为符号拓展，逻辑拓展。还包括左移16位并在低位补0，以用于lui指令。

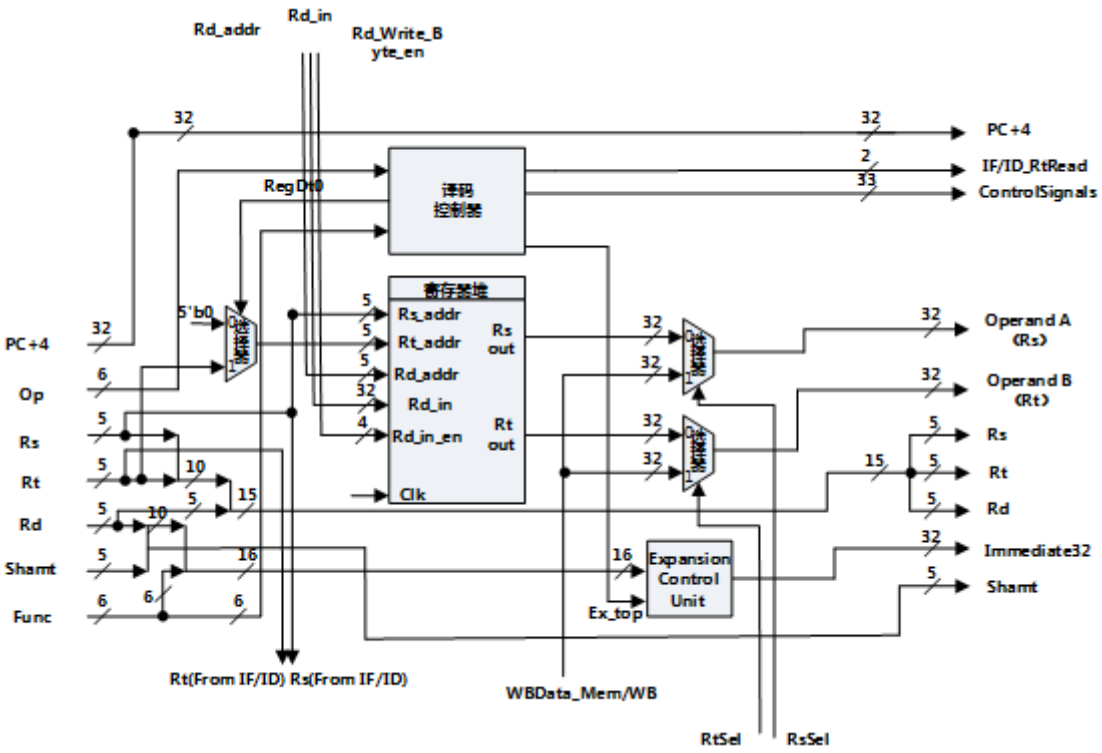


Figure 2: ID段

4.3 EX段的设计

EX段主要运算部件为桶形移位器以及算术逻辑运算单元（ALU），同时还有对转移指令地址的运算模块，ALU以及桶形移位器的结果经选择后存入WBData中，转发选择部分为3选一，由转发模块来控制，数据分别来自MEM以及WB段的寄存器值。

ALU运算数的选择包括两个控制信号，A_in可能为0号寄存器或者Rs字段值，B_in来自寄存器输出或者立即数，ALU产生的标志位存入到段寄存器中，用于下一阶段。

Rd的选择有控制信号RegDst来控制，从Rd，Rt，以及过程返回的31号寄存器中选择一个。

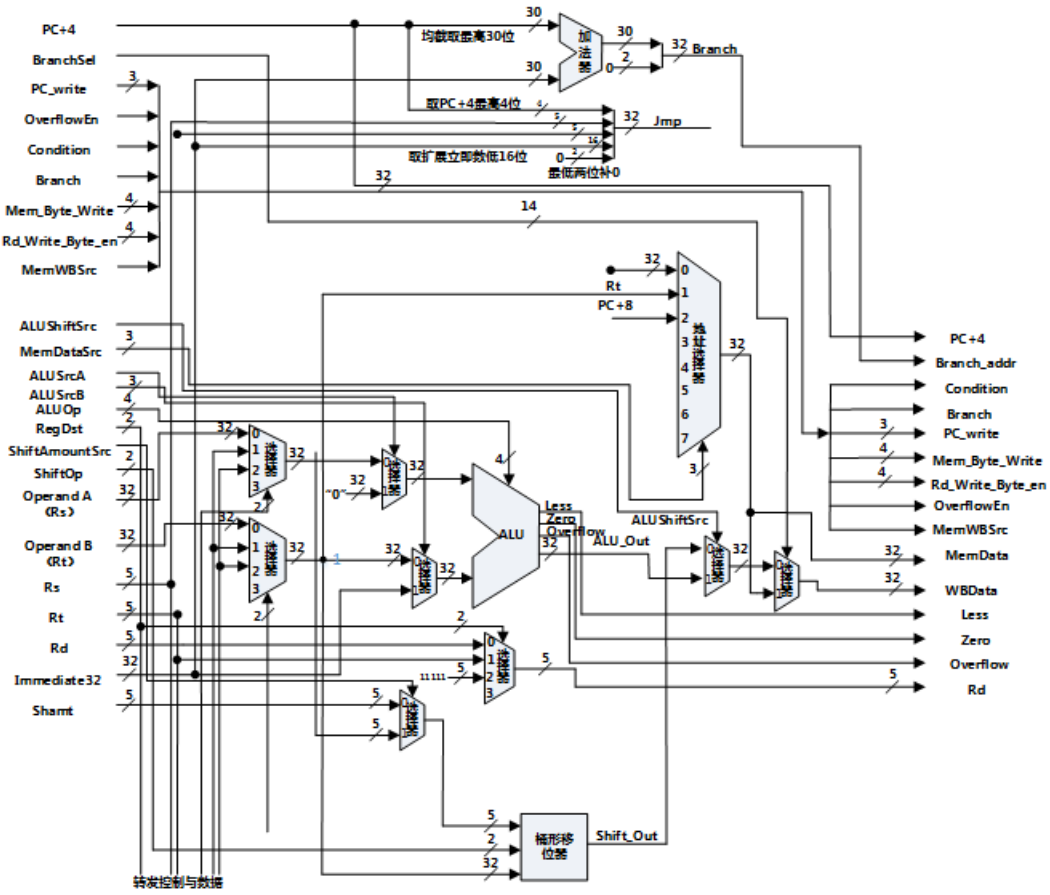


Figure 3: EX段

4.4 MEM段的设计

MEM段主要包括三个模块，分别为条件检测模块，数据存储器模块，以及内存移位输出模块。其中数据存储器模块内部包含寄存器移位输出模块。

条件检测模块根据输入的信号确定寄存器，内存的写使能以及是否执行跳转。数据存储器在时钟下降沿写入，输出经内存移位输出后存入段寄存器内。

4.4.1 条件检测模块

条件检测模块中BranchValid由ALU计算出的标志位来决定，这里设置了7种比较结果判定方式，只要更改传入condition三位，然后就会计算出相应的结果。

寄存器写使能根据溢出与否和指令类型来确定。

溢出判断，根据两个控制信号，如果需要溢出判断，则OverflowEn为1，一旦根据ALU计算的结果Overflow有效，那么就会发生溢出。

对于lwl等类型的指令，寄存器写使能需要结合地址低两位来判断。

4.5 WB段的设计

根据控制信号，选择写入寄存器的数据。

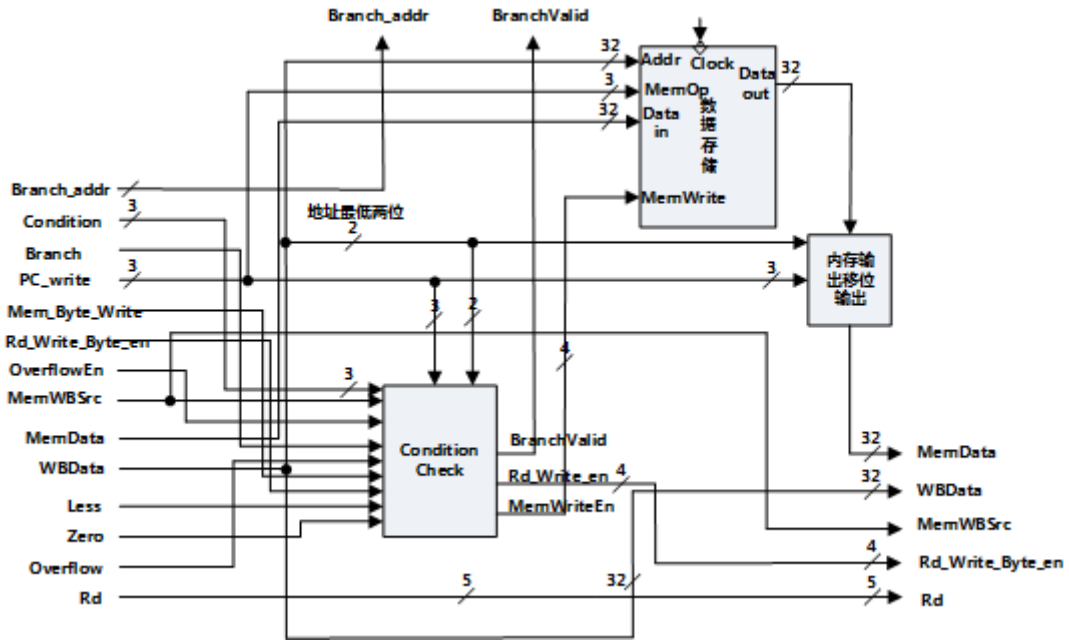


Figure 4: MEM段

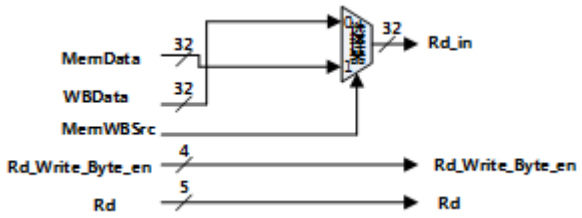


Figure 5: WB段

4.6 冒险模块的设计

4.6.1 结构冒险

结构冒险，也称为硬件资源冲突，是指同一个部件同时被不同指令所使用。

现象：

1.如果只有一个存储器，则在Load指令取数据同时又取指令的话，则发生冲突!

2.如果不对寄存器堆的写口和读口独立设置的话，则发生冲突!

如下图所示:

解决要求：

1.一个部件每条指令只能使用1次，且只能在特定周期使用。

2.设置多个部件，以避免冲突。

解决方案：

1.将Instruction Memory(Im)和Data Memory(Dm)分开。

2.将寄存器读口和写口独立开来。

如下图所示:

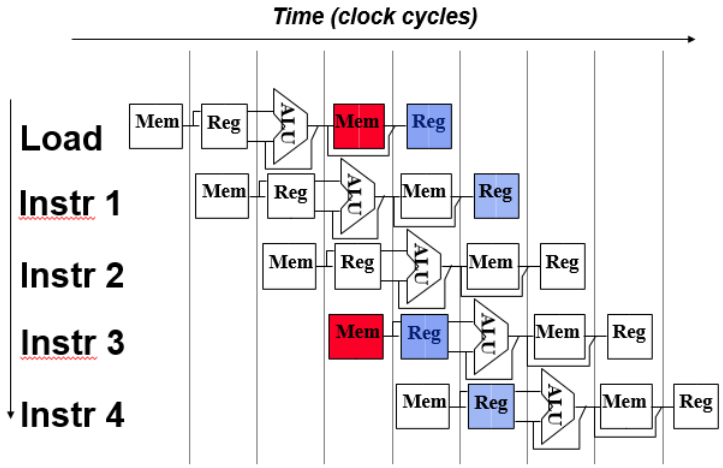


Figure 6: 结构冒险1

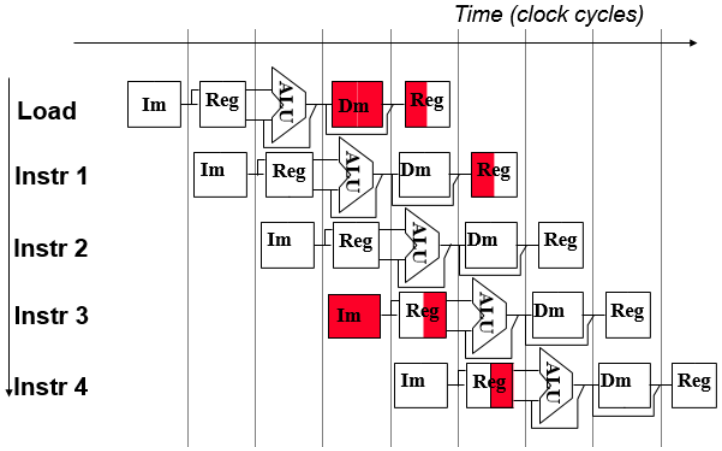


Figure 7: 结构冒险2

4.6.2 数据冒险

数据冒险是指后面指令用到前面指令结果时，前面指令结果还没产生的现象。我们的MIPS流水线采用采用转发(Forwarding/Bypassing)技术予以解决，Load-use冒险需要一次阻塞(Stall)。

4.6.3 第一条指令的目的寄存器是后一条指令的源寄存器

出现这一情况则需要将ALU的运算结果从EX/MEM段寄存器转发到下一条指令的EX段执行之前，这样才可以保证执行的正确性。如图所示： 逻辑设计：需要首先判断两条指令目的寄存器与源寄存器的值是否相等，以及写使能是否有效，如果不有效的话，则直接忽略掉转发,公

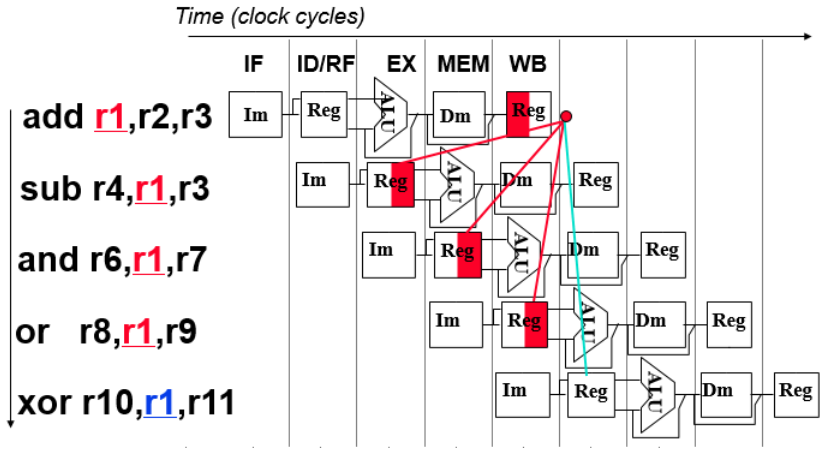


Figure 8: 数据冒险1

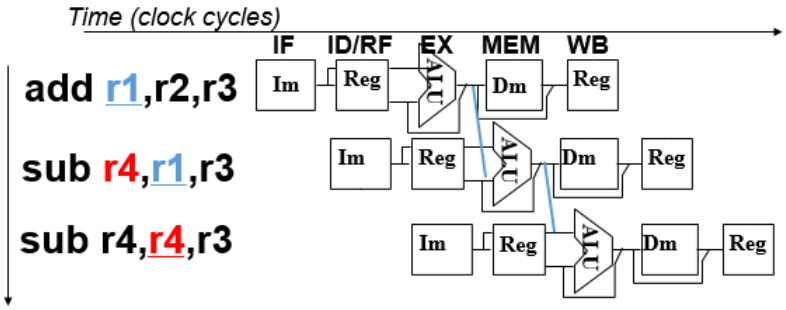


Figure 9: 数据冒险2

式如下：

$$\begin{aligned} Rs\_From\_ID\_EX === Rd\_From\_EX\_MEM; \\ RegWrite\_From\_EX\_MEM! == 4'b0000; \\ Rt\_From\_ID\_EX === Rd\_From\_EX\_MEM; \\ RegWrite\_From\_EX\_MEM! == 4'b0000; \end{aligned} \tag{1}$$

4.6.4 第一条指令的目的寄存器是后第二条指令的源寄存器

出现这一情况则需要将MEM段的结果从MEM/WB段寄存器转发到下一条指令的EX段执行之前，这样才可以保证执行的正确性。

如图数据冒险3所示：

逻辑设计：需要首先判断两条指令目的寄存器与源寄存器的值是否相等，以及写使能是否有效，如果不有效的话，则直接忽略掉转发,公式如下：

$$\begin{aligned} Rs\_From\_ID\_EX === Rd\_From\_MEM\_WB; \\ RegWrite\_From\_MEM\_WB! == 4'b0000; \\ Rt\_From\_ID\_EX === Rd\_From\_MEM\_WB; \\ RegWrite\_From\_MEM\_WB! == 4'b0000; \end{aligned} \tag{2}$$

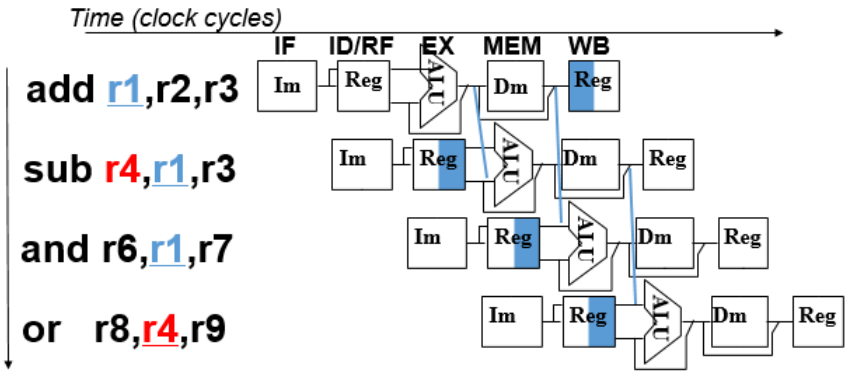


Figure 10: 数据冒险3

4.6.5 第一条指令的目的寄存器是后第三条指令的源寄存器

寄存器前半周期写，后半周期读就可以解决好这一冒险问题。这里我们也采用了转发，这样可以节省时间，是此处不会成为处理器频率的瓶颈。

如图所示： 逻辑设计：需要首先判断两条指令目的寄存器与源寄存器的值是否相等，以及

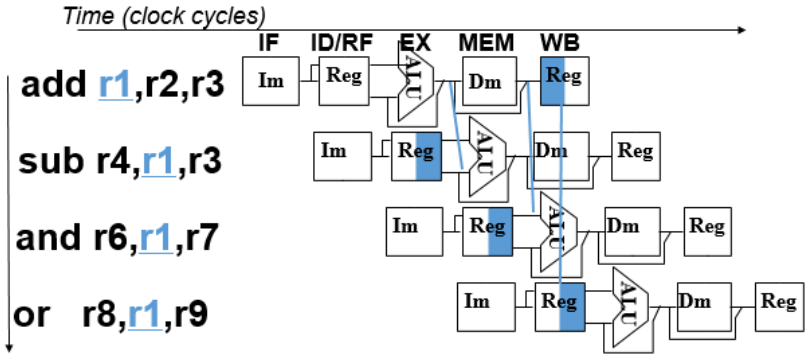


Figure 11: 数据冒险4

写使能是否有效，如果不有效的话，则直接忽略掉转发,公式如下：

$$\begin{aligned} Rs_From\_IF\_ID === Rd\_From\_MEM\_WB; \\ RegWrite\_From\_MEM_WB! == 4'b0; \\ Rt_From\_IF\_ID === Rd\_From_MEM\_WB; \\ RegWrite\_From\_MEM\_WB! == 4'b0; \end{aligned} \tag{3}$$

4.6.6 Load所产生的数据冒险

Load指令之后，如果下一条指令的源寄存器与Load的目的寄存器相同，就会有LoudUse冒险。一旦触发LoadUse冒险，就会冲刷段寄存器，插入指令Nop。 LoadUse冒险发生在Load指令的EX段，Load下条指令的ID段，判断条件如下:

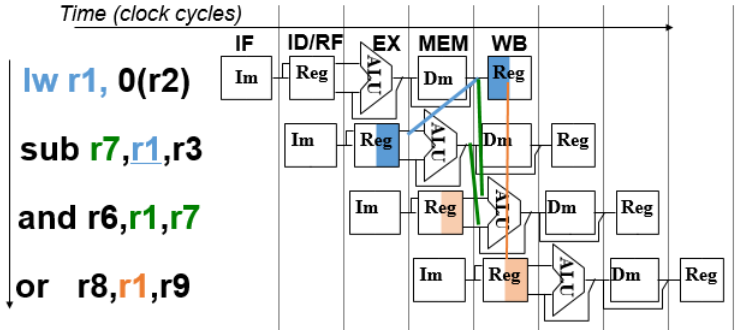


Figure 12: 数据冒险5

$$\begin{aligned}&((RsRead === 1'b1 \quad Rt\_From\_ID\_EX === Rs\_From\_IF\_ID)\\ &Or(RtRead === 1'b1 \quad Rt\_From\_ID\_EX === Rt\_From\_IF\_ID)) \\ &And(MemWBSrc === 1'b1)\end{aligned} \tag{4}$$

RsRead信号表明下条指令会读取Rs寄存器，RtRead类似，而EX段的MemWBSrc信号值为1则表明当前指令为Load指令。

一旦触发LoadUse冒险，就会在下一周期冲刷ID/EX段寄存器，同时向IF/ID段寄存器发出Stall指令，从而相当于插入了一条Nop指令，Load后的指令被延时一个周期执行。如果是其后第二条指令，在经过上述步骤处理后，则会满足转发的第二个条件，从MEM/WB段寄存器转发到下一条指令的EX段执行之前。至此，LoadUse冒险得以解决。

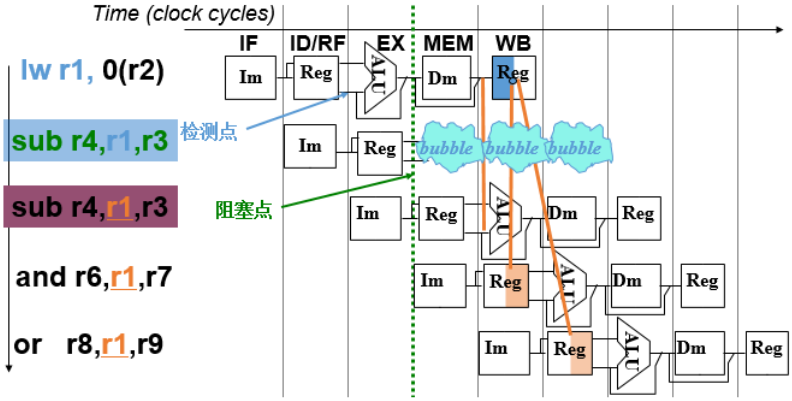


Figure 13: 数据冒险6

4.6.7 一些需要注意的细节问题

1.并不是所有的指令Rs字段含义为Rs寄存器的地址，比如Rotr。

2.许多指令的Rt字段并非表示Rt寄存器的地址，如I型指令，Jump指令。

对于1，2两类情况，不加约束可能会导致错误的转发，如果指令还在使用ALU，就可能引发不可预知的错误。

3.要注意传入目的寄存器的写使能位，只有写使能有有效位时，才会回去转发，否则不进行转发操作。

4.注意条件判断要完全符合，在流水线CPU执行时，前几条指令的转发条件因为使用===符号判断才能有效解决，否则转发信号为xx，就会是数据通路被阻塞。

5.当同时发生前两种冒险时，优先选择后一种冒险转发，即转发最近指令的数据。

6.**加入两个寄存器Rs，Rt读标志信号，以防止因为字段误当作寄存器号而引发的问题。**

4.6.8 控制冒险

采用静态分支预测，总是预测不转移，一旦转移，就必须冲刷寄存器。对于Jump和Branch两类跳转情形是不同的。Jump在EX段就可以确定转移，此时需要冲刷ID/EX段寄存器。而对于Branch指令，必须在ALU进行条件计算后，根据结果来判断是否需要转移。在EX/MEM段可以确定是否转移，一旦确定转移，则地址被送到PC，然后冲刷掉ID/EX，EX/MEM两个段寄存器。

逻辑设计：

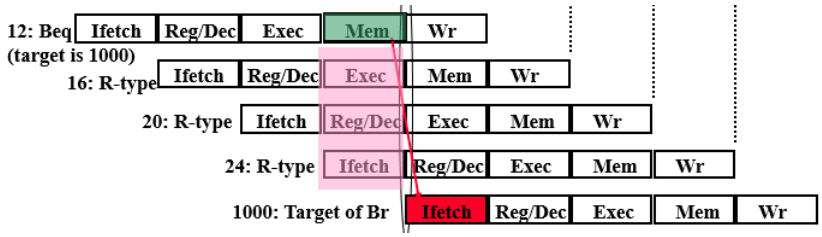


Figure 14: 控制冒险1

ID段，一旦发现Jump有效，就会触发Jump控制冒险；

在EX阶段，一旦发现Branch有效，则就会触发Branch控制冒险。

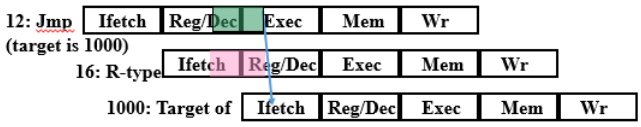


Figure 15: 控制冒险2

4.7 段寄存器的设计

段寄存器主要由三个控制信号Clk，Stall以及Flush控制，其中在时钟信号Clk上升沿到来时，根据Stall，Flush来改变寄存器内的值。如果段寄存器内Stall有效，则会保持之前的值不会改变，从而达到延迟一个周期的目的。当Flush有效，将会将所有寄存器内的值清0，从而达到冲刷寄存器的目的。

5 特别鸣谢

感谢张泽生老师在流水线以及之前实验中给出的指导，他在最后实验中给出的流水线参考图对我们的设计起到了非常大的帮助和参考作用。