# 实验报告二 流水线CPU

报告人：胡博岩 2019302785

[实验报告二 流水线CPU 1](#_Toc26490)

[1 实验要求 1](#_Toc15523)

[2 实验过程 1](#_Toc21722)

[2.1 不考虑冒险的流水线CPU 2](#_Toc23436)

[2.2 EXE级数据冒险 6](#_Toc22049)

[2.3 ID级控制冒险 12](#_Toc1110)

[2.4 ID级数据冒险 17](#_Toc525)

[2.5 改进 19](#_Toc2386)

[3 实验总结 20](#_Toc26104)

[3.1 在本次实验遇到的问题： 20](#_Toc23960)

[3.2 对流水线CPU设计的理解： 20](#_Toc24369)

[3.3比较分析流水线CPU和单周期CPU的性能： 21](#_Toc11628)

[3.4对这次实验的感受： 21](#_Toc25687)

1. **实验要求**

1.1：在实验一设计的单周期CPU的基础上增加四个流水线寄存器，实现五级流水线CPU框架，使它能够执行一段没有数据冒险和控制冒险的程序。

1.2：处理三种EXE级数据冒险：寄存器的新值是ALU的计算结果；寄存器新值是从DM中读出的数据；寄存器在WB周期结束时才能写入引起的数据冒险

1.3：处理两种控制冒险：J, JR, JAL指令控制冒险；beq指令（ID级比较）控制冒险

1.4：处理一种ID级数据冒险：beq指令的比较操作提前到ID级引入了新的ID级数据冒险。

1. **实验过程**
   1. **不考虑冒险的流水线CPU**

**原理分析：**

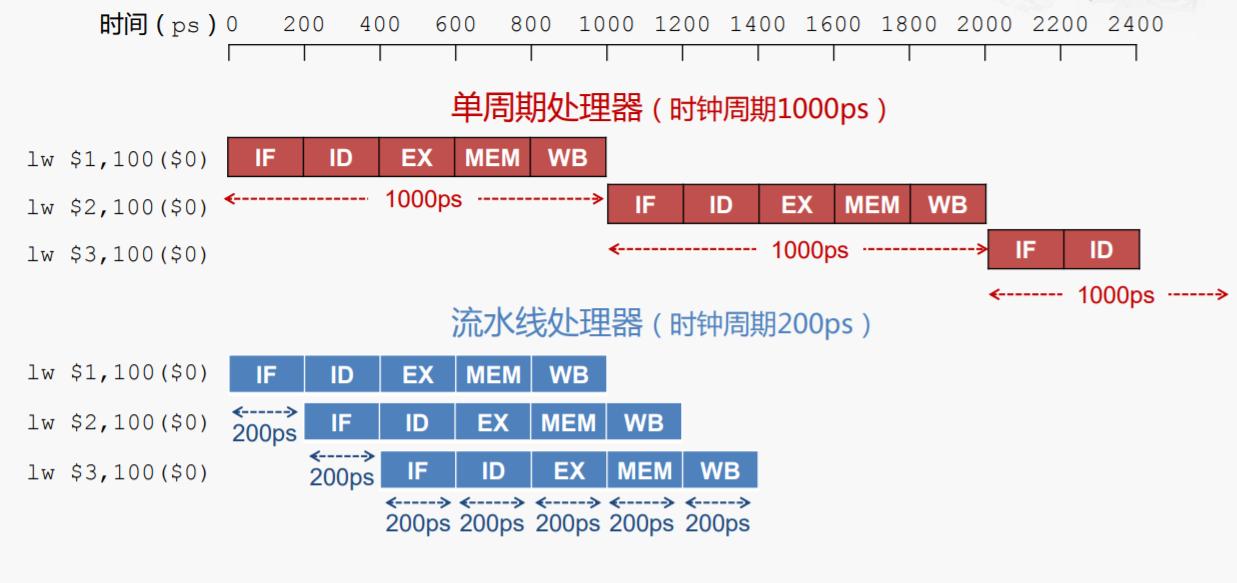
cpu[流水线技术](https://baike.baidu.com/item/%E6%B5%81%E6%B0%B4%E7%BA%BF%E6%8A%80%E6%9C%AF" \t "https://baike.baidu.com/item/cpu%E6%B5%81%E6%B0%B4%E7%BA%BF/_blank)是一种将指令分解为多步，并让不同指令的各步操作重叠，从而实现几条指令[并行处理](https://baike.baidu.com/item/%E5%B9%B6%E8%A1%8C%E5%A4%84%E7%90%86/8983963" \t "https://baike.baidu.com/item/cpu%E6%B5%81%E6%B0%B4%E7%BA%BF/_blank)，以加速程序运行过程的技术。指令的每步有各自独立的电路来处理，每完成一步，就进到下一步，而前一步则处理后续指令。

根据MIPS处理器的特点，将整体的处理过程分为取指令（IF）、指令译码（ID）、执行（EX）、存储器访问（MEM）和寄存器会写（WB）五级，对应多周期的五个处理阶段。一个指令的执行需要5个时钟周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理。

经典的五级流水线：IF -> ID -> EX -> MEM -> WB



采用[流水线技术](https://baike.baidu.com/item/%E6%B5%81%E6%B0%B4%E7%BA%BF%E6%8A%80%E6%9C%AF" \t "https://baike.baidu.com/item/cpu%E6%B5%81%E6%B0%B4%E7%BA%BF/_blank)后，并没有加速单条指令的执行，每条指令的操作步骤一个也不能少，只是多条指令的不同操作步骤同时执行，因而从总体上看加快了[指令流](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E6%B5%81" \t "https://baike.baidu.com/item/cpu%E6%B5%81%E6%B0%B4%E7%BA%BF/_blank)速度，缩短了程序执行时间。

****

**数据通路：**



**代码实现思路：**

在实验一、二设计的单周期CPU的基础上，增添四个流水线寄存器模块IF\_ID、ID\_EXE、EXE\_MEM、MEM\_WB，由于现在由流水线寄存器隔开的5段数据通路在同一时刻执行的是5条不同的指令，故我们需要利用上述4个寄存器实现不同指令的传输，即根据不同指令产生的各信号作用的模块位置来决定是否储存和传输各阶段不同指令产生的各信号。

**模块代码如下：**

if\_id.v

module if\_id(npc\_1,instruction\_1,npc,instruction,clock,reset);

//输入、输出略

input clock,reset;

always@(posedge clock or negedge reset) begin

if(reset) begin

npc\_1<=npc;instruction\_1<=instruction;end

else begin

npc\_1<=32'b0; instruction\_1<=32'b0; end

end

endmodule

id\_exe.v

module id\_exe(npc\_2,data\_1\_2,data\_2\_2,imm\_32\_2,num\_write\_2,s\_b\_2,mem\_write\_2,reg\_write\_2,s\_data\_write\_2,aluop\_2,npc\_1,data\_1,data\_2,imm\_32,num\_write,s\_b,mem\_write,reg\_write,s\_data\_write,aluop,clock,reset);

//输入、输出略

always@(posedge clock or negedge reset) begin

if(reset) begin

npc\_2<=npc\_1;data\_1\_2<=data\_1;data\_2\_2<=data\_2;imm\_32\_2<=imm\_32;num\_write\_2<=num\_write;s\_b\_2<=s\_b;mem\_write\_2<=mem\_write;s\_data\_write\_2<=s\_data\_write;aluop\_2<=aluop;reg\_write\_2<=reg\_write; end

else begin

npc\_2<=31'b0;data\_1\_2<=31'b0;data\_2\_2<=31'b0;imm\_32\_2<=31'b0;num\_write\_2<=5'b0;s\_b\_2<=1'b0;s\_data\_write\_2<=2'b0;mem\_write\_2<=1'b0;aluop\_2<=4'b0;reg\_write\_2<=1'b0; end

end

endmodule

exe\_mem.v

module exe\_mem(npc\_3,c\_3,data\_2\_3,num\_write\_3,mem\_write\_3,reg\_write\_3,s\_data\_write\_3,npc\_2,c,data\_2\_2,num\_write\_2,mem\_write\_2,reg\_write\_2,s\_data\_write\_2,clock,reset);

//输入、输出略

always@(posedge clock or negedge reset) begin

if(reset) begin

npc\_3<=npc\_2;c\_3<=c;data\_2\_3<=data\_2\_2;num\_write\_3<=num\_write\_2;mem\_write\_3<=mem\_write\_2;s\_data\_write\_3<=s\_data\_write\_2;reg\_write\_3<=reg\_write\_2; end

else begin

npc\_3<=31'b0;c\_3<=31'b0;data\_2\_3<=31'b0;num\_write\_3<=5'b0;mem\_write\_3<=1'b0;s\_data\_write\_3<=2'b0;reg\_write\_3<=1'b0; end

end

endmodule

mem\_wb.v

module mem\_wb(npc\_4,c\_4,data\_out\_4,num\_write\_4,reg\_write\_4,s\_data\_write\_4,npc\_3,c\_3,data\_out,num\_write\_3,reg\_write\_3,s\_data\_write\_3,clock,reset);

//输入、输出略

always@(posedge clock or negedge reset) begin

if(reset) begin

npc\_4<=npc\_3;c\_4<=c\_3;data\_out\_4<=data\_out;num\_write\_4<=num\_write\_3;s\_data\_write\_4<=s\_data\_write\_3;reg\_write\_4<=reg\_write\_3; end

else begin

npc\_4<=31'b0;c\_4<=31'b0;data\_out\_4<=31'b0;num\_write\_4<=5'b0;s\_data\_write\_4<=2'b0; reg\_write\_4<=1'b0; end

end

endmodule

pipeline\_cpu.v

module pipeline\_cpu(clock,reset);

//输入、输出、赋值略

add ADD\_2(beq\_addr\_1,npc,imm\_32<<2);//生成equal时的beq跳转地址加法器

mux MUX\_5(beq\_addr,npc,beq\_addr\_1,zero);//beq指令跳转地址beq\_addr选择器

mux\_4 MUX\_4(n\_pc,beq\_addr,reg\_addr,abs\_addr,npc,s\_pc);//下一个地址n\_pc选择器

pc PC(pc,clock,reset,n\_pc);//pc模块

add ADD\_1(npc,pc,4);//加法器模块

im IM(instruction,pc);//指令存储器

if\_id IF\_ID(npc\_1,instruction\_1,npc,instruction,clock,reset);//流水线寄存器（1）

ctrl CTRL(aluop,reg\_write,s\_num\_write,s\_b,s\_extend,s\_data\_write,mem\_write,s\_pc,op,funct);

//控制模块

extend EXTEND(imm\_32,imm,s\_extend);//扩展模块

mux\_3 #(5) MUX\_1(num\_write,rt,rd,5'b11111,s\_num\_write);//写寄存器num\_write选择器

gpr GPR(data\_1,data\_2,clock,reg\_write\_4,num\_write\_4,rs,rt,data\_write);//通用寄存器

id\_exe ID\_EXE(npc\_2,data\_1\_2,data\_2\_2,imm\_32\_2,num\_write\_2,s\_b\_2,mem\_write\_2,reg\_write\_2,s\_data\_write\_2,aluop\_2,npc\_1,data\_1,data\_2,imm\_32,num\_write,s\_b,mem\_write,reg\_write,s\_data\_write,aluop,clock,reset);//流水线寄存器（2）

mux MUX\_2(b,data\_2\_2,imm\_32\_2,s\_b\_2);//b选择器

alu ALU(zero,c,a,b,aluop\_2);//alu

exe\_mem EXE\_MEM(npc\_3,c\_3,data\_2\_3,num\_write\_3,mem\_write\_3,reg\_write\_3,s\_data\_write\_3,npc\_2,c,data\_2\_2,num\_write\_2,mem\_write\_2,reg\_write\_2,s\_data\_write\_2,clock,reset);

dm DM(data\_out,clock,mem\_write\_3,address,data\_2\_3);//数据存储器

mem\_wb MEM\_WB(npc\_4,c\_4,data\_out\_4,num\_write\_4,reg\_write\_4,s\_data\_write\_4,npc\_3,c\_3,data\_out,num\_write\_3,reg\_write\_3,s\_data\_write\_3,clock,reset);//流水线寄存器（3）

mux\_3 MUX\_3(data\_write,npc\_4,c\_4,data\_out\_4,s\_data\_write\_4);//写数据data\_write选择器

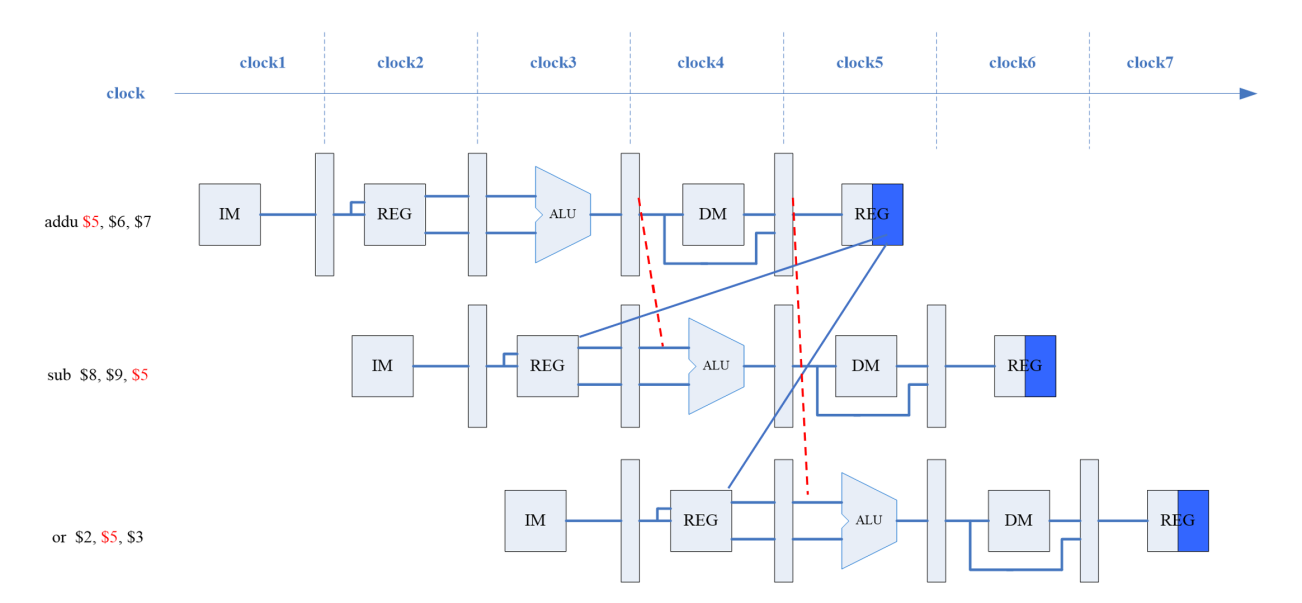
endmodule

* 1. **EXE级数据冒险**

数据冒险：也称为流水线数据冒险（pipeline data hazard），即因无法提供指令执行所需数据而导致指令不能在预定的时钟周期内执行的情况。

**2.2.1 EXE级数据冒险（1）——寄存器的新值是ALU的计算结果**

**原理分析：**



R型指令和I型指令在EXE级需要从寄存器堆读数据，如果此时MEM级或者WB级的指令有将数据写回寄存器的操作而且写回的寄存器编号与EXE级指令的读寄存器编号相同，R型指令和I型指令在EXE级读取时数据不是最新数据，就会产生数据冒险。

为解决此类数据冒险，我们使用旁路将ALU的计算结果提前给出即可。

**数据通路：**



**代码实现思路：**

为解决此类数据冒险，我们增添exe\_data\_hazard\_1模块和两个旁路（选择器）。

对于exe\_data\_hazard\_1模块，我们利用rs,rt,num\_write\_2,num\_write\_3信号对是否出现此类EXE级数据冒险进行判定并给出两个旁路的控制信号s\_forwardA和s\_forwardB。

对于两条旁路，注意旁路位于EXE级，故需要将exe\_data\_hazard\_1模块给出的控制信号s\_forwardA和s\_forwardB从ID级传输到EXE，再根据传输后的控制信号s\_forwardA\_2和s\_forwardB\_2分别对c\_3,c\_4,data\_1\_2和c\_3,c\_4,data\_2\_2进行选择，以解决数据冒险。

**模块代码如下：**

exe\_data\_hazard\_1.v

module exe\_data\_hazard\_1(s\_forwardA,s\_forwardB,rs,rt,num\_write\_2,reg\_write\_2,num\_write\_3,reg\_write\_3);

output reg [1:0] s\_forwardA,s\_forwardB;

input [4:0] rs,rt,num\_write\_2,num\_write\_3;

input reg\_write\_2,reg\_write\_3;

always@(\*) begin

if(rs==num\_write\_2 && reg\_write\_2) s\_forwardA<=2'b00;

else if(rs==num\_write\_3 && reg\_write\_3) s\_forwardA<=2'b01;

else s\_forwardA<=2'b10;

if(rt==num\_write\_2 && reg\_write\_2) s\_forwardB<=2'b00;

else if(rt==num\_write\_3 && reg\_write\_3) s\_forwardB<=2'b01;

else s\_forwardB<=2'b10;

end

endmodule

pipeline\_cpu.v增加代码如下

exe\_data\_hazard\_1 EXE\_DH\_1(s\_forwardA,s\_forwardB,rs,rt,num\_write\_2,num\_write\_3);

mux\_3 MUX\_6(a,c\_3,c\_4,data\_1\_2,s\_forwardA\_2);

mux\_3 MUX\_7(data\_2\_2\_out,c\_3,c\_4,data\_2\_2,s\_forwardB\_2)

**波形分析：**

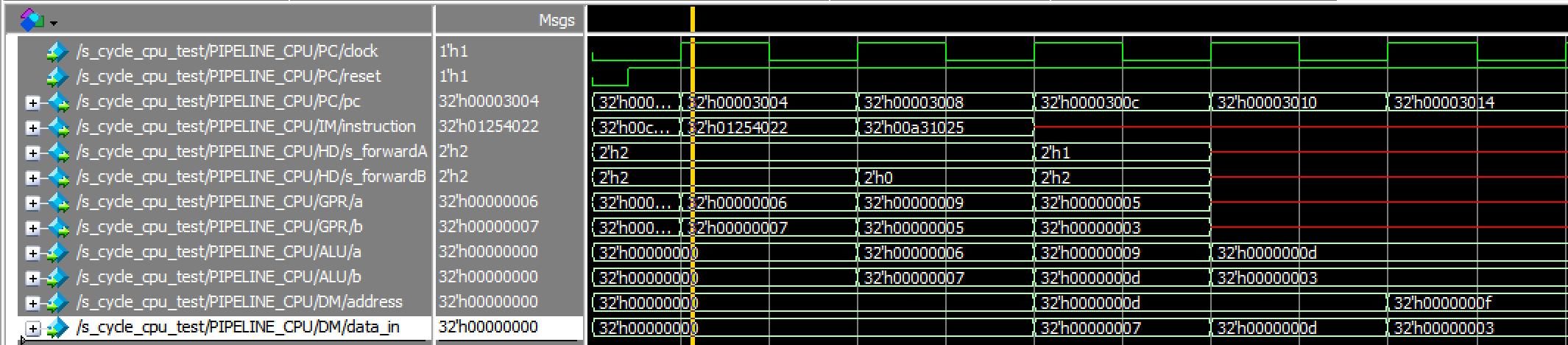
addu $5,$6,$7

sub $8,$9,$5

or $2,$5,$3

第三条指令和第一条、第二条指令都存在数据冒险，该冒险解决后，旁路将第二条指令的计算结果前递，第三条指令的源操作数$5号寄存器的值应该是第二条指令的计算结果。

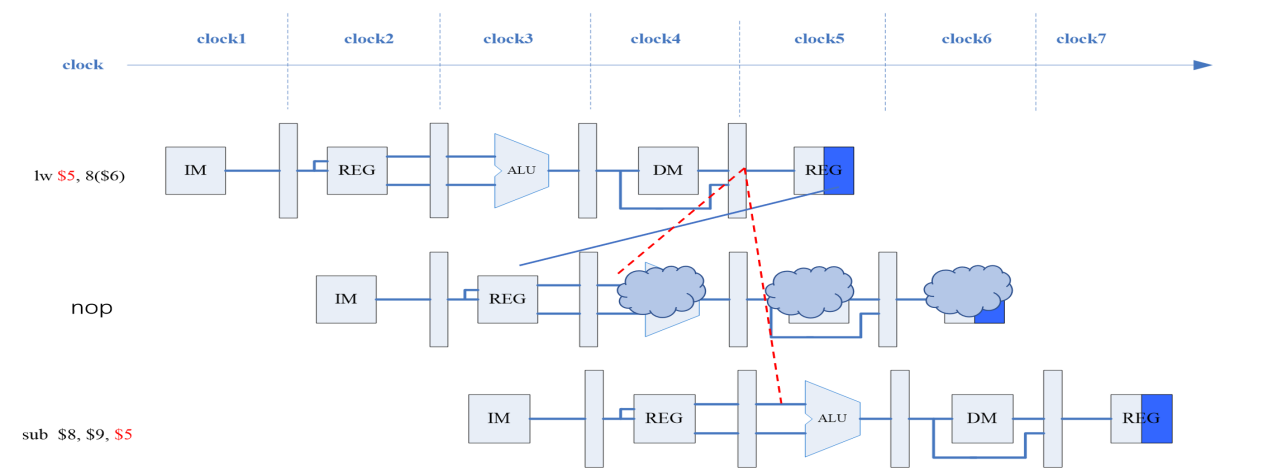
仿真波形如下：



clock3时，可以看到s\_forwardA,s\_forwardB信号的改变，仿真结果正确。

**2.2.2 EXE级数据冒险（2）——寄存器新值是从DM中读出的数据**

**原理分析：**



R型指令和I型指令在EXE级需要从寄存器堆读数据，如果此时上一条指令为lw指令，并且lw指令load的地址即等于R型指令和I型指令从寄存器读数据的地址，R型指令和I型指令在EXE级读取时数据不是最新数据，就会产生数据冒险。

为了解决该类数据冒险，我们阻塞一个周期进行即可。

**数据通路：**



**代码实现思路：**

为了解决该类数据模型，我们增添exe\_data\_hazard\_2模块，并对PC寄存器、IF\_ID流水线寄存器、ID\_EXE流水线寄存器增添控制信号pc\_write,IF\_ID\_write,ID\_EXE\_flush。

对于exe\_data\_hazard\_2模块，我们利用rs,rt,num\_write\_2,op,op\_2信号对是否出现此类EXE级数据冒险进行判定，并且输出PC寄存器、IF\_ID流水线寄存器、ID\_EXE流水线寄存器的控制信号pc\_write,IF\_ID\_write,ID\_EXE\_flush。

对于PC寄存器，pc\_write是写使能信号，当发生此类数据冒险时，pc\_write被置0，PC寄存器不更新。

对于IF\_ID流水线寄存器，控制信号IF\_ID\_write也是写使能信号，当发生此类数据冒险时，IF\_ID\_write被置0，IF\_ID流水线寄存器不更新。

对于ID\_EXE流水线寄存器，ID\_EXE\_flush信号为清零信号，当发生此类数据冒险时，ID\_EXE\_flush被置1，把ID\_EXE流水线寄存器中控制信号都置0（也可以全部置0），这样当前指令无效 。

**模块代码如下：**

exe\_data\_hazard\_2.v

module exe\_data\_hazard\_2(pc\_write,IF\_ID\_write,ID\_EXE\_flush,rs,rt,num\_write\_2,op,op\_2);

output reg pc\_write,IF\_ID\_write,ID\_EXE\_flush;

input [5:0] op,op\_2;

input [4:0] rs,rt,num\_write\_2;

always@(\*) begin

if(op\_2!=6'b100011) begin //上一条是否是lw指令

pc\_write<=1'b1; ID\_EXE\_flush<=1'b0; IF\_ID\_write<=1'b1; end

else begin

if(op==6'b000000) begin //这一条是否是R型指令

if(rs==num\_write\_2 || rt==num\_write\_2) begin

pc\_write<=1'b0; ID\_EXE\_flush<=1'b1; IF\_ID\_write<=1'b0; end

else begin

pc\_write<=1'b1; ID\_EXE\_flush<=1'b0; IF\_ID\_write<=1'b1; end

end

else begin //这一条是否是I型指令

if(rs==num\_write\_2) begin

pc\_write<=1'b0; ID\_EXE\_flush<=1'b1; IF\_ID\_write<=1'b0; end

else begin

pc\_write<=1'b1; ID\_EXE\_flush<=1'b0; IF\_ID\_write<=1'b1; end

end

end

end

endmodule

**波形分析：**

addu $5,$6,$7

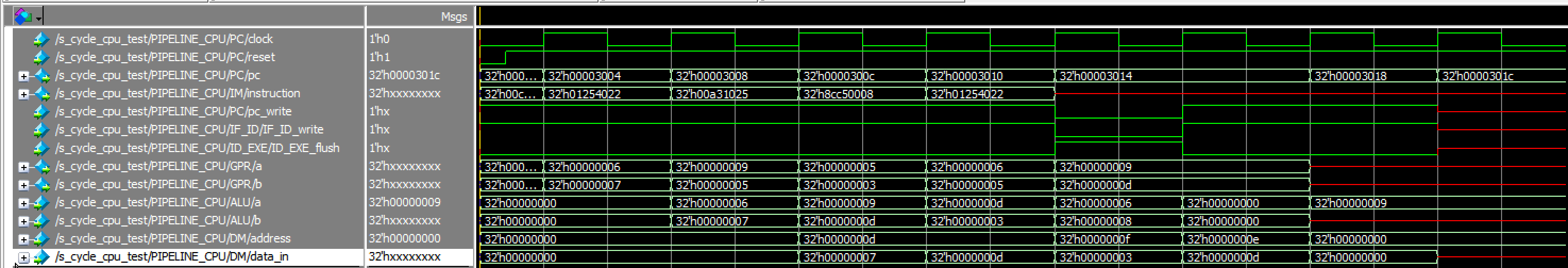
sub $8,$9,$5

or $2,$5,$3

lw $5,8($6)

sub $8,$9,$5

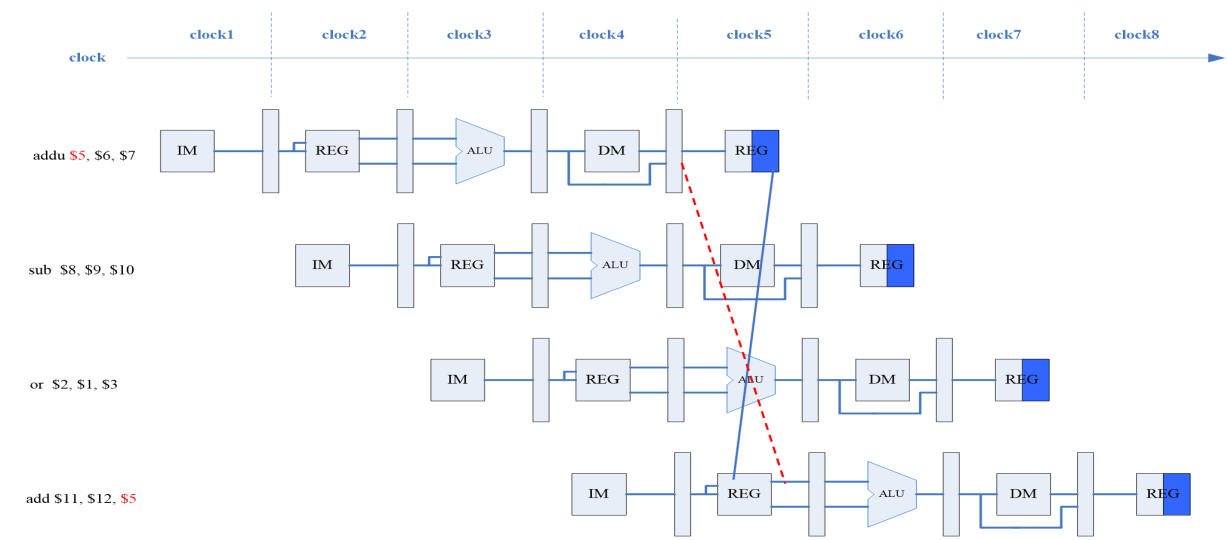
第五条指令和第四条指令存在数据冒险，该冒险解决后，第五条指令延迟一个周期执行，其源操作数$5号寄存器的值应该是第二条指令的计算结果。



clock6时，可以看到pc\_write,IF\_ID\_write,ID\_EXE\_flush信号的改变，仿真结果正确。

**2.2.3 EXE级数据冒险（3）——寄存器在WB周期结束时才能写入**

**原理分析：**



R型指令和I型指令在EXE级需要从寄存器堆读数据，如果此时在WB级执行的指令写回的寄存器编号等于R型指令和I型指令从寄存器读数据的地址R型指令和I型指令在EXE级读取时数据不是最新数据，就会产生数据冒险。

为了解决该类数据冒险，我们使用旁路提前将写回寄存器的数据给出即可。

**数据通路：**



**代码实现思路：**

为了解决该类数据模型，我们增添exe\_data\_hazard\_3模块和两条旁路（选择器）。

对于exe\_data\_hazard\_3模块，我们利用op,rt,rs,reg\_write\_4,num\_write\_4信号对是否出现此类EXE级数据冒险进行判定并给出两个旁路的控制信号s\_forwardA2和s\_forwardB2。

对于两条旁路，注意旁路位于ID级，根据控制信号s\_forwardA2和s\_forwardB2分别对c\_3,c\_4,data\_1\_2和c\_3,c\_4,data\_2\_2进行选择，以解决数据冒险。

**模块代码如下：**

exe\_data\_hazard\_3.v

//EXE级数据冒险（3）——寄存器在WB周期结束时才能写入引起的数据冒险

module exe\_data\_hazard\_3(s\_forwardA2,s\_forwardB2,op,rt,rs,reg\_write\_4,num\_write\_4);

output reg s\_forwardA2,s\_forwardB2;

input [5:0] op;

input [4:0] rt,rs,num\_write\_4;

input reg\_write\_4;

always@(\*) begin

if(!reg\_write\_4) begin //WB

s\_forwardA2<=1'b1; s\_forwardB2<=1'b1; end

else begin

if(op==6'b000000) begin //R

if(rs == num\_write\_4) s\_forwardA2<=1'b0;

else s\_forwardA2<=1'b1;

if(rt == num\_write\_4) s\_forwardB2<=1'b0;

else s\_forwardB2<=1'b1;

end

else if(op==6'b001000 || op==6'b001001 || op==6'b001100 || op==6'b001101) begin //I

if(rs == num\_write\_4) begin s\_forwardA2<=1'b0; s\_forwardB2<=1'b1; end

else begin s\_forwardA2<=1'b1; s\_forwardB2<=1'b1; end

end

else begin

s\_forwardA2<=1'b1; s\_forwardB2<=1'b1; end

end

end

endmodule

pipeline\_cpu.v增加代码如下

exe\_data\_hazard\_3 EXE\_DH\_3(s\_forwardA2,s\_forwardB2,op,rt,rs,reg\_write\_4,num\_write\_4);

mux MUX\_8(data\_1\_out,data\_write,data\_1,s\_forwardA2);

mux MUX\_9(data\_2\_out,data\_write,data\_2,s\_forwardB2);

**波形仿真：**

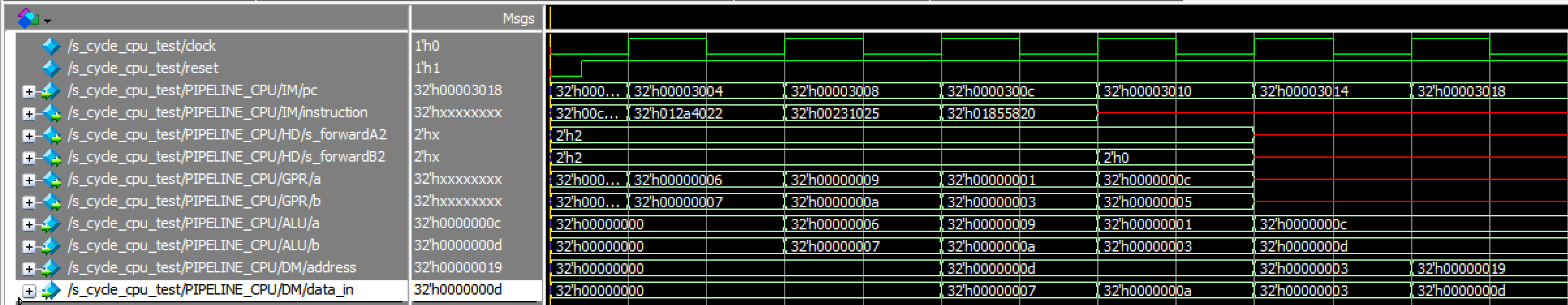
addu $5,$6,$7

sub $8,$9,$10

or $2,$1,$3

add $11,$12,$5

第四条指令和第一条指令存在数据冒险，该冒险解决后，旁路将第一条指令的计算结果前递，第四条指令的源操作数应该是第一条指令的计算结果。



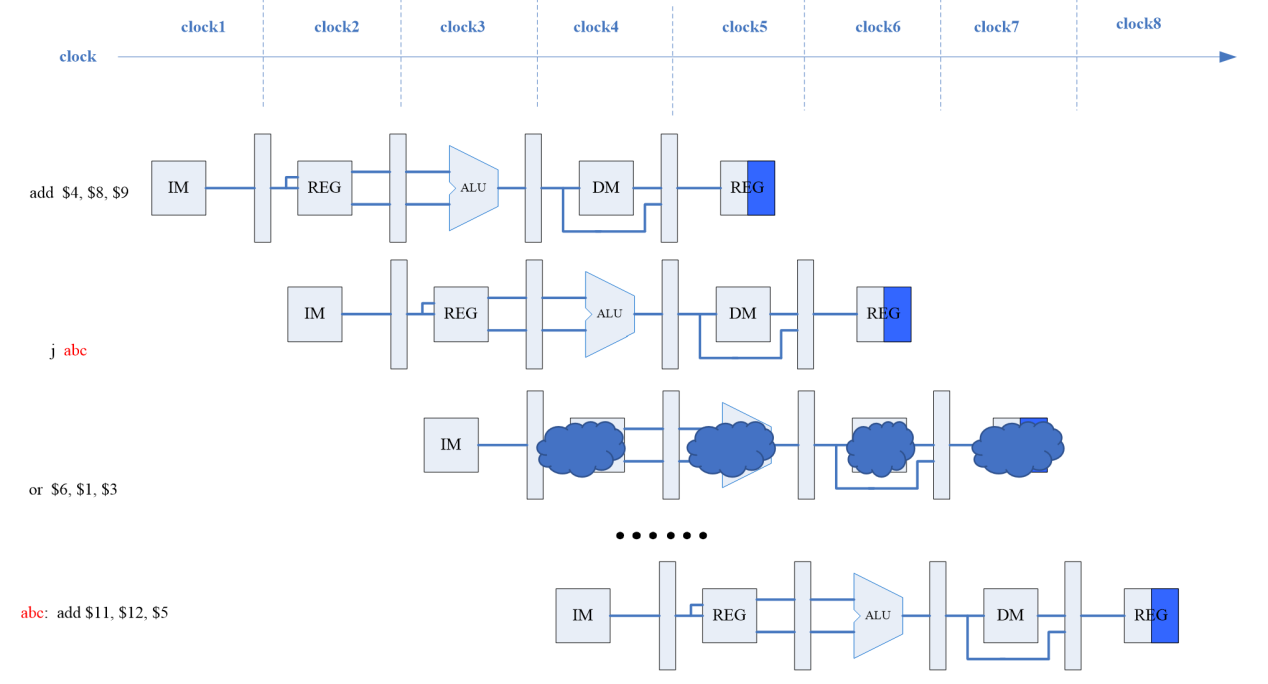
clock5时，可以看到s\_forwardA2、s\_forwardB2信号发生变化，仿真结果正确。

* 1. **ID级控制冒险**

控制冒险：也称分支冒险（control hazard）。因为取到的指令并不是所需要的（或者说指令地址的变化并不是流水线所预期的）而导致指令不能在预定的时钟周期内执行。

**2.3.1 J, JR, JAL指令控制冒险**

**原理分析：**



J,JR,JAL指令会在ID级确定跳转地址，但是此时下一条指令在IF级已经完成取指，即下一条指令其实也进入流水线并将继续执行，该指令并不是流水线所预期的，即产生了控制冒险。

为了解决此类控制冒险，在上述跳转指令在ID级执行后对IF\_ID流水线寄存器进行清零即阻塞一个周期即可。

**数据通路：**



**代码实现思路：**

为了解决此类控制冒险，我们增添id\_control\_hazard模块和IF\_ID流水线寄存器的控制信号IF\_ID\_flush。

对于id\_control\_hazard模块，我们利用s\_npc即可判断ID级执行指令是否是J,JR,JAL指令，并以此为根据产生IF\_ID\_flush信号。

对于IF\_ID流水线寄存器，控制信号IF\_ID\_flush是清零信号，当发生此类控制冒险时，IF\_ID\_flush被置1，把IF\_ID流水线寄存器中控制信号都置0（也可以全部置0），这样当前指令无效 。

**模块代码如下：**

id\_control\_hazard.v

//J, JR, JAL指令控制冒险

`include"head.v"

module id\_control\_hazard\_1(IF\_ID\_flush,s\_npc);

output reg IF\_ID\_flush;

input [1:0] s\_npc;

always@(\*) begin

if(s\_npc==ABS\_S || s\_npc==REG\_S) IF\_ID\_flush<=1'b1;

else IF\_ID\_flush<=1'b0;

end

endmodule

**波形分析：**

add $4,$8,$9

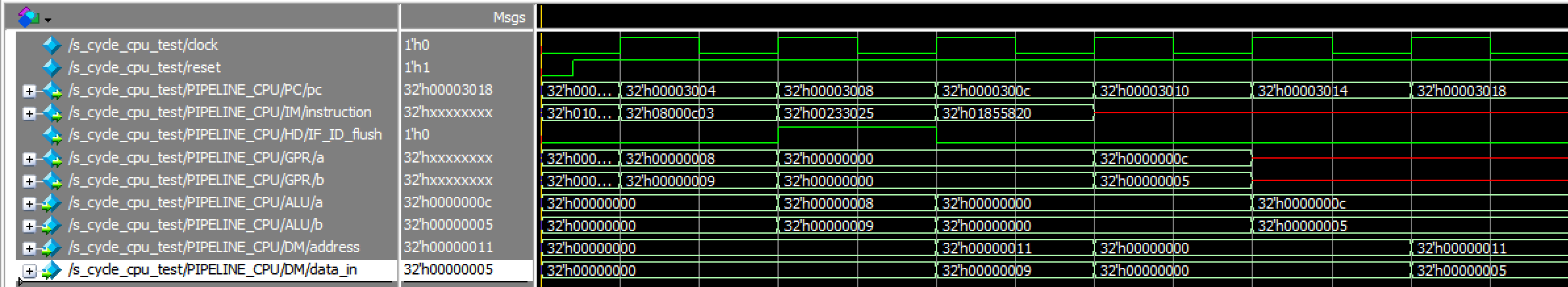
j abc

or $6,$1,$3

abc:

add $11,$12,$5

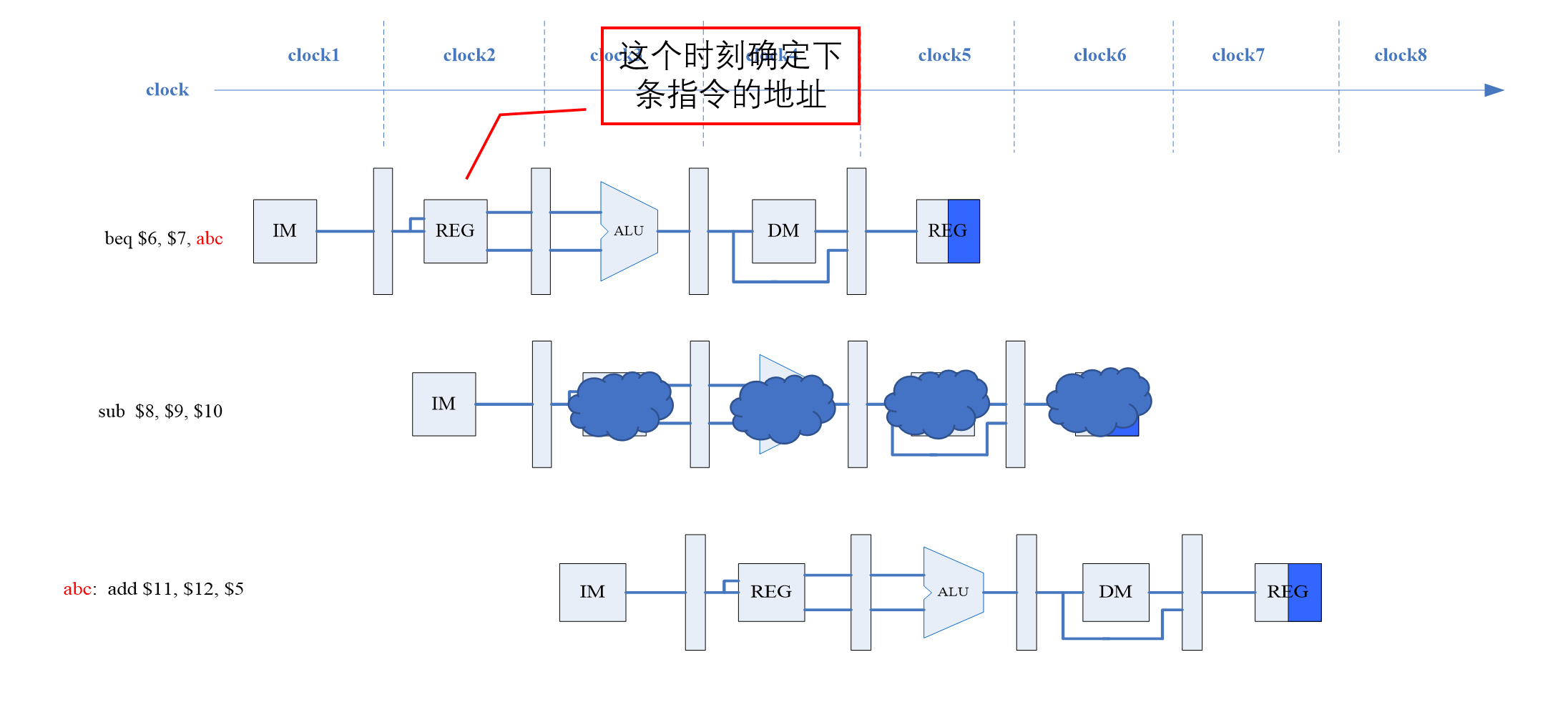
第三条指令与第二条指令存在控制冒险，该冒险解决后，第三条指令在取指后被清除。



clock3时，可以看到IF\_ID\_flush信号发生变化，仿真结果正确。

**2.3.2 BEQ（ID级比较）指令控制冒险**

**原理分析：**



对于BEQ指令，本来应该在EXE级比较以确定跳转的地址，但是如果在EXE级比较，我们需要阻塞两个周期才能确定下条指令的地址，这样比较麻烦。于是，我们想到增加硬件，将比较提前到ID级，减少阻塞时间，这样只需要阻塞一个周期。

在ID级比较的情况下，BEQ指令会在ID级确定跳转地址，但是此时下一条指令在IF级已经完成取指，即下一条指令其实也进入流水线并将继续执行，该指令并不是流水线所预期的，即产生了控制冒险。

为了解决此类控制冒险，在BEQ指令在ID级执行后对IF\_ID流水线寄存器进行清零即阻塞一个周期即可。

**数据通路：**



**代码实现思路：**

为了将BEQ'指令比较提前至ID级并解决此类控制冒险，我们在pipeline\_cpu.v中对BEQ指令相关模块重新进行连线并对id\_control\_hazard模块进行修改。

对于BEQ指令的比较信号zero，原本zero信号由EXE级的ALU模块产生，我们将其提前到ID级，直接由data\_1\_out和data\_2\_out比较得出zero信号。

对于id\_control\_hazard模块，我们增加考虑s\_npc为BEQ\_S和zero为0的情况。

**模块代码如下：**

id\_control\_hazard.v

//控制冒险

`include"head.v"

module id\_control\_hazard\_1(IF\_ID\_flush,s\_npc,zero);

output reg IF\_ID\_flush;

input [1:0] s\_npc;

input zero;

always@(\*) begin

if(s\_npc==`ABS\_S || s\_npc==`REG\_S || (s\_npc==`BEQ\_S && zero==1'b0))

IF\_ID\_flush<=1'b1;

else

IF\_ID\_flush<=1'b0;

end

endmodule

**波形分析：**

add $6,$2,$5

addu $5,$6,$7

sub $8,$9,$10

or $2,$1,$3

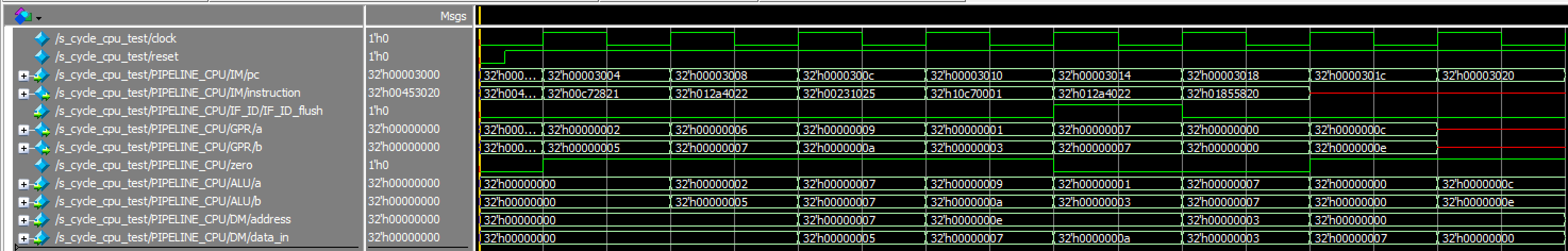
beq $6,$7,abc

sub $8,$9,$10

abc:

add $11,$12,$5

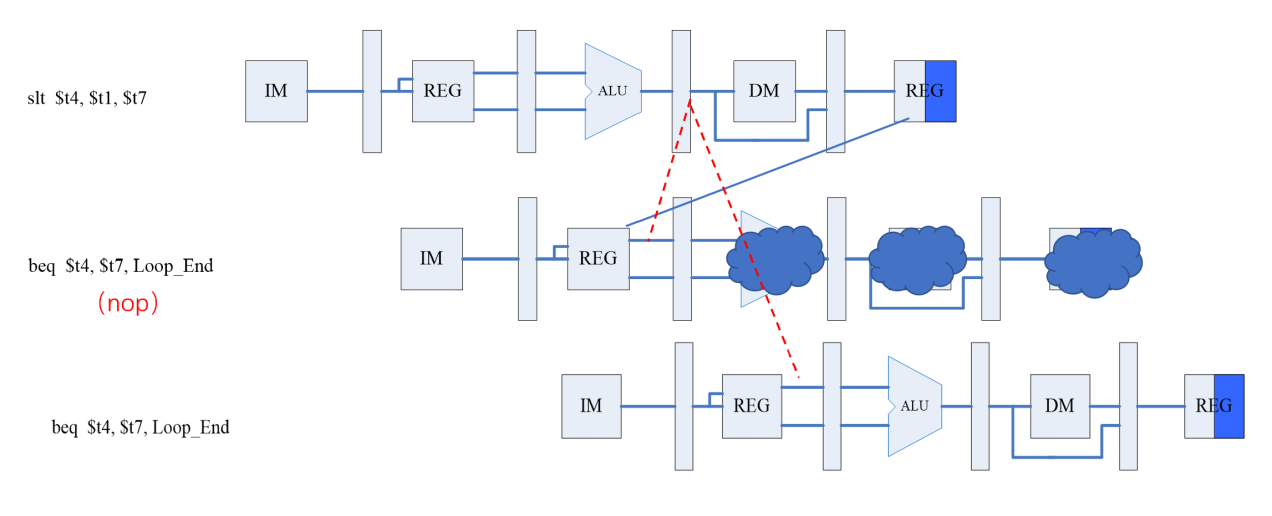
第六条指令与第五条指令存在控制冒险，该冒险解决后，第六条指令在取指后被清除。



clock6时，可以看到IF\_ID\_flush信号发生变化，仿真结果正确。

* 1. **ID级数据冒险**

**原理分析：**



beq指令在ID级进行比较时需要从寄存器堆读数据，如果此时上一条指令为R型或者I型指令，并且R型或者I型指令写回寄存器的地址即等于beq指令从寄存器读数据的地址，beq指令在ID级读取时数据不是最新数据，就会产生数据冒险。

为了解决该类数据冒险，我们阻塞一个周期进行即可。

**数据通路：**



**代码实现思路：**（参考2.1.2）

为了解决此数据冒险，我们增添id\_data\_hazard模块。

对于id\_data\_hazard模块，我们利用op,rs,rt,num\_write\_2,reg\_write\_2判断是否出现此类ID级数据冒险，并产生pc\_write、IF\_ID\_write、ID\_EXE\_flush控制信号。

对于PC寄存器，pc\_write是写使能信号，当发生此类数据冒险时，pc\_write被置0，PC寄存器不更新。

对于IF\_ID流水线寄存器，控制信号IF\_ID\_write也是写使能信号，当发生此类数据冒险时，IF\_ID\_write被置0，IF\_ID流水线寄存器不更新。

对于ID\_EXE流水线寄存器，ID\_EXE\_flush信号为清零信号，当发生此类数据冒险时，ID\_EXE\_flush被置1，把ID\_EXE流水线寄存器中控制信号都置0（也可以全部置0），这样当前指令无效 。

**增加模块代码如下：**

id\_data\_hazard.v

//由于BEQ比较提前到ID级而引起的ID级数据冒险

module id\_data\_hazard(pc\_write,IF\_ID\_write,ID\_EXE\_flush,op,rs,rt,num\_write\_2,reg\_write\_2);

output reg pc\_write2,IF\_ID\_write2,ID\_EXE\_flush2;

input [5:0] op;

input [4:0] rs,rt,num\_write\_2;

input reg\_write\_2;

always@(\*) begin if(op!=6'b000100 || !reg\_write\_2)

//本条指令不是BEQ指令or上条指令不写回寄存器

begin pc\_write2<=1'b1; ID\_EXE\_flush2<=1'b0; IF\_ID\_write2<=1'b1; end

else begin

if(rs==num\_write\_2 || rt==num\_write\_2) begin pc\_write<=1'b0; ID\_EXE\_flush<=1'b1; IF\_ID\_write<=1'b0; end

else begin pc\_write<=1'b1; ID\_EXE\_flush<=1'b0; IF\_ID\_write<=1'b1; end

end

end

endmodule

**波形分析：**

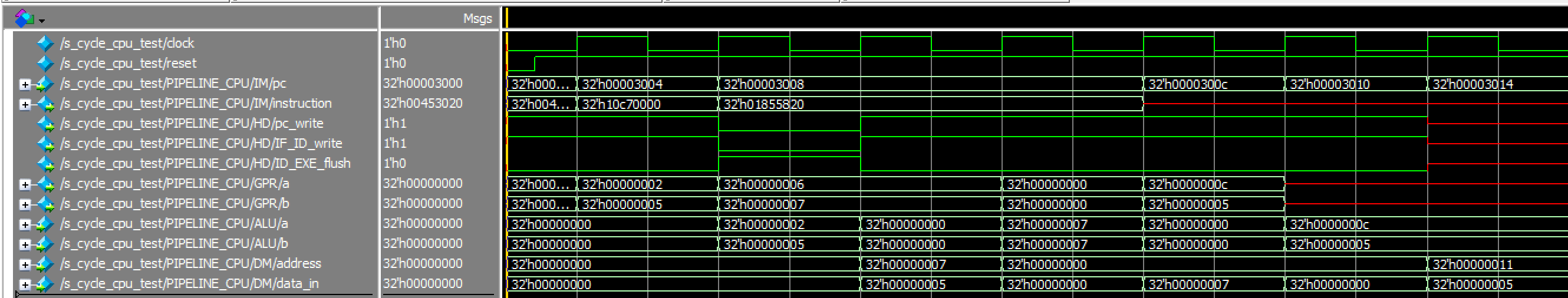
add $6,$2,$5

beq $6,$7,abc

abc:

add $11,$12,$5

第二条指令和第一条指令存在数据冒险,该冒险解决后，第二条指令延迟一个周期执行。



clock3时，可以看到pc\_write,IF\_ID\_write,ID\_EXE\_flush信号发生变化，仿真结果正确。

* 1. **改进**

**思考：**

做完所有实验提交正确后，我重新来审视我写的verilog代码，我发现了不少的所谓不足之处：

1. 每次处理冒险时都新建一个模块造成了负责处理各冒险的模块多而冗杂。
2. 一直大量使用if-else语句的嵌套，导致代码量大且冗杂。

于是，我决定将我前面编写的各个数据冒险和控制冒险处理模块合并成一个模块，并且将其中的if-else语句替换为三目运算符。

**改进结果：**

hazard\_detector.v

//冒险处理

`include"head.v"

module hazard\_detector(pc\_write,IF\_ID\_write,ID\_EXE\_flush,IF\_ID\_flush,s\_forwardA,s\_forwardB,s\_forwardA2,s\_forwardB2,op\_2,rt,rs,num\_write\_2,num\_write\_3,num\_write\_4,s\_npc,zero,reg\_write\_2,reg\_write\_3,reg\_write\_4);

//输入、输出略

assign s\_forwardA=(rs==num\_write\_2 && rs!=0 && reg\_write\_2)?2'b00:

(rs==num\_write\_3 && rs!=0 && reg\_write\_3)?2'b01:2'b10;

assign s\_forwardB=(rt==num\_write\_2 && rt!=0 && reg\_write\_2)?2'b00:

(rt==num\_write\_3 && rt!=0 && reg\_write\_3)?2'b01:2'b10;

assign s\_forwardA2=(rs==num\_write\_4 && rs!=0 && reg\_write\_4)?2'b00:

(rs==num\_write\_3 && rs!=0 && reg\_write\_3)?2'b01:2'b10;

assign s\_forwardB2=(rt==num\_write\_4 && rt!=0 && reg\_write\_4)?2'b00:

(rt==num\_write\_3 && rt!=0 && reg\_write\_3)?2'b01:2'b10;

assign pc\_write=(op\_2==6'b100011 && (rs==num\_write\_2 || rt==num\_write\_2))?1'b0:

(rs==num\_write\_2 && rs!=0 && s\_npc==`BEQ\_S && reg\_write\_2)?1'b0:

(rt==num\_write\_2 && rt!=0 && s\_npc==`BEQ\_S && reg\_write\_2)?1'b0:1'b1;

assign IF\_ID\_write=pc\_write;

assign ID\_EXE\_flush=~pc\_write;

assign IF\_ID\_flush=(s\_npc==`ABS\_S || s\_npc==`REG\_S || (s\_npc==`BEQ\_S && zero==1'b0))?1'b1:1'b0;

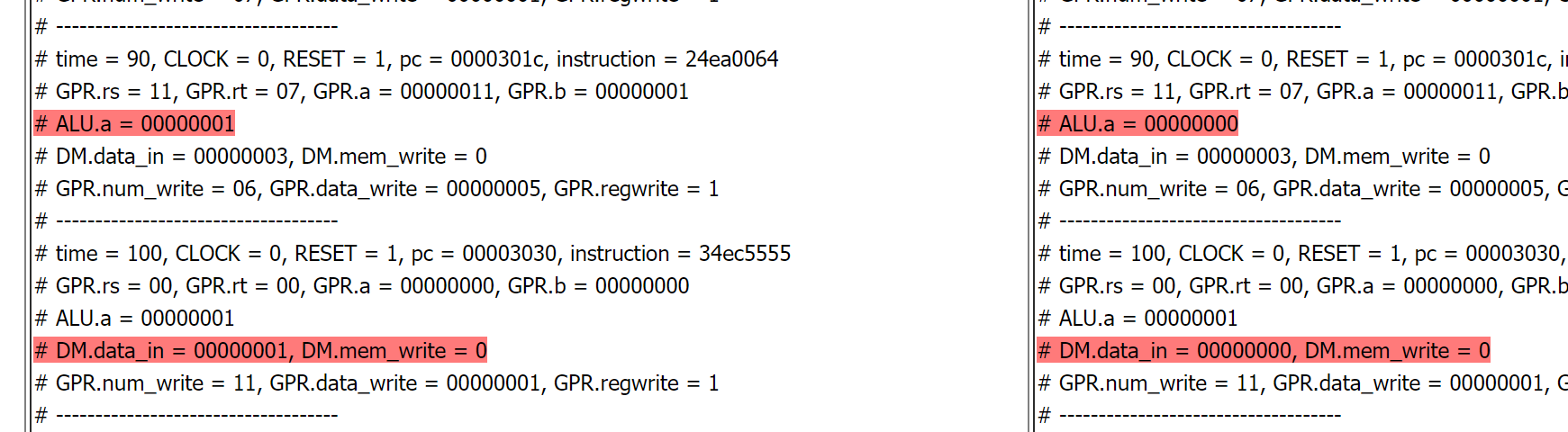
endmodule

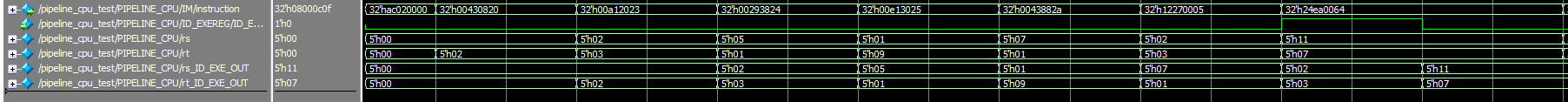
1. **实验总结**

**3.1 在本次实验遇到的问题：**

1.在2.3.1阶段，我发现虽然J、JAL、JR指令的控制冒险已经解决，但是仍出现alu.a数据错误的情况，于是我经过代码排查和波形仿真，最终发现在处理第一类数据冒险时，我考虑不够周全，单凭rs、rt和number\_write\_2、number\_write\_3来判断是否发生该类数据冒险，要想完善对该类数据冒险是否发生的判断，还应该引入考虑写寄存器信号reg\_write\_2和reg\_write\_3。

2.在2.4阶段，我发现虽然ID级控制冒险已经解决，但是仍出现alu.a数据错误的情况，于是我经过代码排查和波形仿真，最终发现在ID\_EXE流水线寄存器模块中，当ID\_EXE\_flush信号为1时，我之前只将ID\_EXE流水线寄存器中传输的控制信号全部赋值为0，但实际上，我们应该将寄存器读数据也赋值为0，并且将s\_forwardA\_2和s\_forwardB\_2信号赋值为2b‘10，以避免出现下图中的数据错误。





**3.2 对流水线CPU设计的理解：**

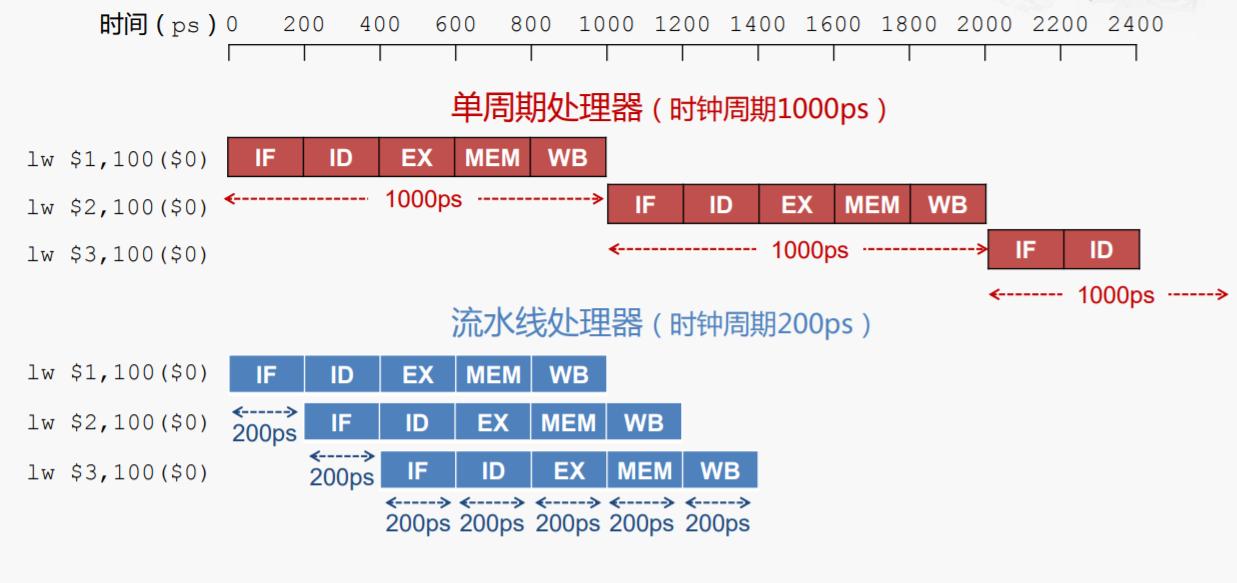
在单周期CPU的基础上，要设计一个流水线CPU：

首先就是要设计好它的流水线寄存器，要注意哪些信号需要从哪一级传输到哪一级，不同的流水线寄存器需要传输的信号并不相同，需要根据设计需求确定；

其次是处理数据和控制冒险，要考虑各个冒险涉及的变量，并以此为根据判断冒险是否发生，然后产生控制信号，并且，为了解决冒险可以利用旁路和阻塞；

**3.3比较分析流水线CPU和单周期CPU的性能：**

当比较流水线CPU和单周期CPU时，我们发现，采用[流水线技术](https://baike.baidu.com/item/%E6%B5%81%E6%B0%B4%E7%BA%BF%E6%8A%80%E6%9C%AF" \t "https://baike.baidu.com/item/cpu%E6%B5%81%E6%B0%B4%E7%BA%BF/_blank)并没有加速单条指令的执行，每条指令的操作步骤还是一个也不能少，但是多条指令的不同操作步骤同时执行，从总体上看加快了[指令流](https://baike.baidu.com/item/%E6%8C%87%E4%BB%A4%E6%B5%81" \t "https://baike.baidu.com/item/cpu%E6%B5%81%E6%B0%B4%E7%BA%BF/_blank)速度，缩短了程序执行时间，故流水线CPU的性能要强于单周期CPU。

****

**3.4对这次实验的感受：**

首先，我感觉这次实验首先使我加深了对流水线cpu各方面上的理解，使我能够更好地学习计算机组成与系统结构的理论课程中第四章的内容；

其次，经过这次实验，我对verilog语言和quastasim软件的使用越来越得心应手，特别是学会了利用三目运算符代替if-else语句来精简代码；

最重要的是，这次实验极大程度上考验了我的耐心、细心和意志力，让我明白坚持对于成功的意义。