1 实验要求

1)在单周期 CPU 的基础上增加流水线寄存器，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。

2) 处理两种 EXE 级数据冒险。

3) 继续逐步处理其他各种冒险，使实现的五级流水线 CPU 能够执行 fibonacci 程序。

4) 比较分析单周期 CPU 和流水线 CPU 的性能。

2 实验过程

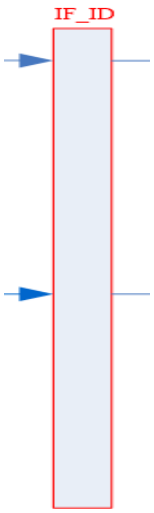
2.1 流水线 CPU(不考虑冒险)

2.1.1 功能描述

增加流水线寄存器，在不考虑冒险的情况下实现能够执行 addu,add,addi,subu,and,or,slt,addiu,andi,ori,lui,lw,sw 等指令的流水线 CPU。

2.1.2 主要模块及代码实现

IF/ID 寄存器：



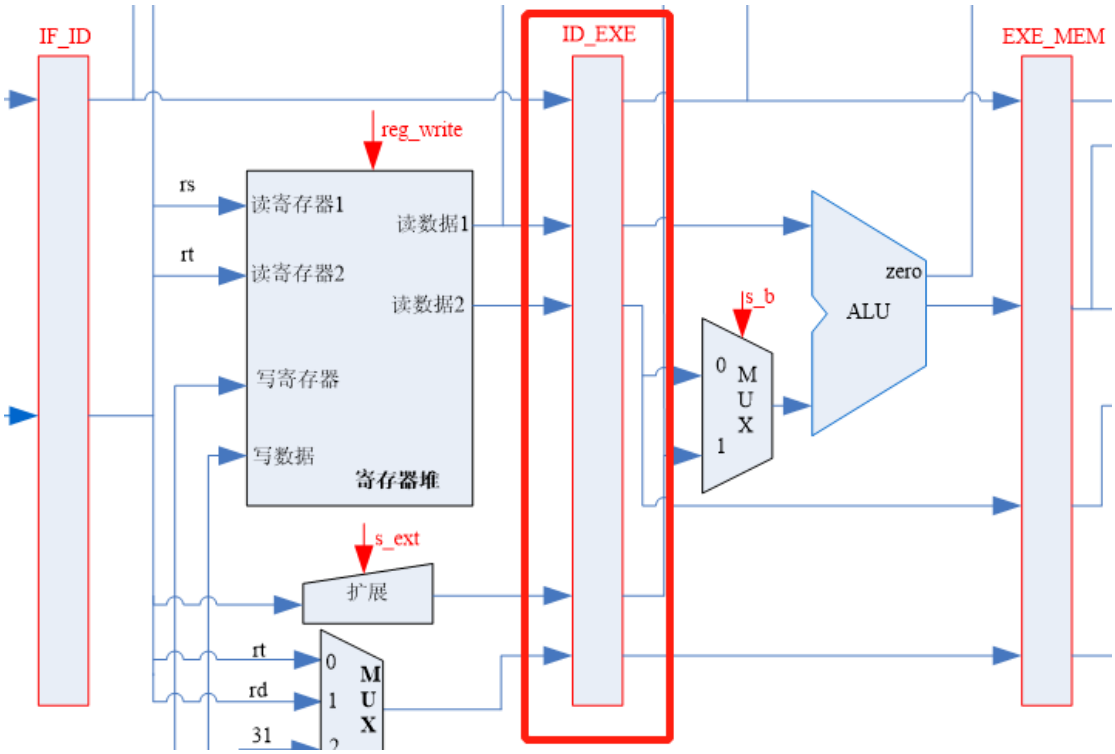
代码实现：

```
`include "pipe_reg.v"
module if_id(pc4, inst, clock, reset, dpc4, dinst);
    input [31:0] pc4;
    input [31:0] inst;
    input   clock;
    input   reset;
    output [31:0] dpc4;
    output [31:0] dinst;

    pipe_reg nextpc(pc4,clock,reset,dpc4);
    pipe_reg id(inst,clock,reset,dinst);
endmodule
```

分析：pipe_reg 的功能是时钟上升沿的时候传输，reset 的功能是异步复位，具体如下：

```
module pipe_reg(
        input [31:0] in,
        input clock,
        input reset,
        //input op,
        output reg [31:0]out
 );

    always @(posedge clock,negedge reset)
    begin
    if(!reset)
        out <= 32'h0000_0000;
    else
        out <= in;
    end
endmodule
```

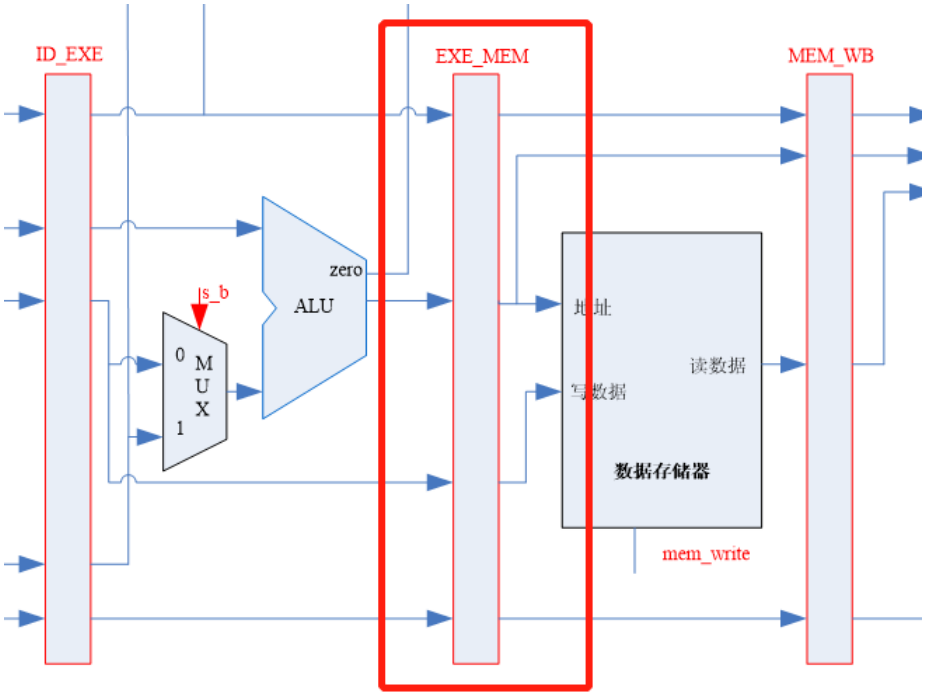
ID_EXE 寄存器：



代码实现：

```
`include "pipe_reg.v"
`include "pipe_reg_5.v"
`include "pipe_reg_1.v"
`include "pipe_reg_6.v"
`include "pipe_reg_2.v"
module id_exe(
    input clock,
    input reset,
    input [31:0]dpc4,
    input [31:0]data1,
    input [31:0]data2,
    input [31:0]ext_imm,
    input [4:0] rw,
    input [4:0] aluop,
    input s_b,
    input reg_write,
    input mem_write,
    input [1:0] s_data_write,
    output  [31:0] ddpc4,
    output  [31:0] ddata1,
    output  [31:0] ddata2,
    output  [31:0] dext_imm,
    output  [4:0]  drw,
    output  [4:0] naluop,
    output ns_b,
    output nreg_write,
    output nmem_write,
    output [1:0]ns_data_write
);

    pipe_reg pc_1(dpc4,clock,reset,ddpc4);
    pipe_reg reg_data1(data1,clock,reset,ddata1);
    pipe_reg reg_data2(data2,clock,reset,ddata2);
    pipe_reg ext_1(ext_imm,clock,reset,dext_imm);
    pipe_reg_5 rw_1(rw,clock,reset,drw);
    pipe_reg_5 aluop_1(aluop,clock,reset,naluop);
    pipe_reg_1 s_b_1(s_b,clock,reset,ns_b);
    pipe_reg_1 reg_write_1(reg_write,clock,reset,nreg_write);
    pipe_reg_1 mem_write_1(mem_write,clock,reset,nmem_write);
    pipe_reg_2 s_data_write_1(s_data_write,clock,reset,ns_data_write);
endmodule
```

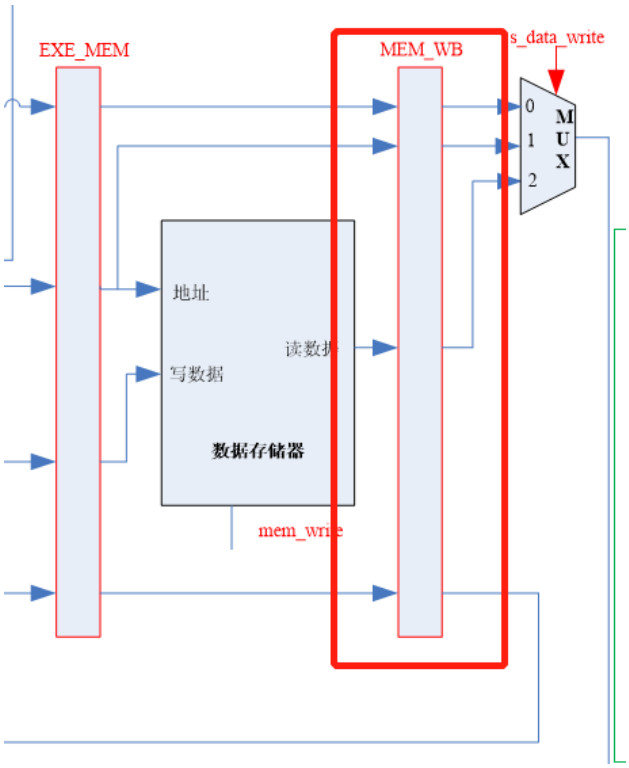
EXE_MEM 寄存器：



代码实现：

```
`include "pipe_reg.v"
`include "pipe_reg_5.v"
`include "pipe_reg_1.v"
`include "pipe_reg_6.v"
`include "pipe_reg_2.v"
module  exe_mem(
    input clock,
    input reset,
    input [31:0] dpc4,
    input [31:0] busw,
    input [31:0] data2,
    input [4:0] rw,
    input nreg_write,
    input nmem_write,
    input [1:0] ns_data_write,
    output [31:0] ddpc4,
    output [31:0] dbusw,
    output [31:0] ddata2,
    output [4:0] drw,
    output nnreg_write,
    output nnmem_write,
    output [1:0]nns_data_write
    );

    pipe_reg pc_2(dpc4,clock,reset,ddpc4);
    pipe_reg busw_1(busw,clock,reset,dbusw);
    pipe_reg reg_data3(data2,clock,reset,ddata2);
    pipe_reg_5 rw_2(rw,clock,reset,drw);
    pipe_reg_1 reg_write_2(nreg_write,clock,reset,nnreg_write);
    pipe_reg_1 mem_write_2(nmem_write,clock,reset,nnmem_write);
    pipe_reg_2 s_data_write_2(ns_data_write,clock,reset,nns_data_write);

endmodule
```

MEM_WB 寄存器：



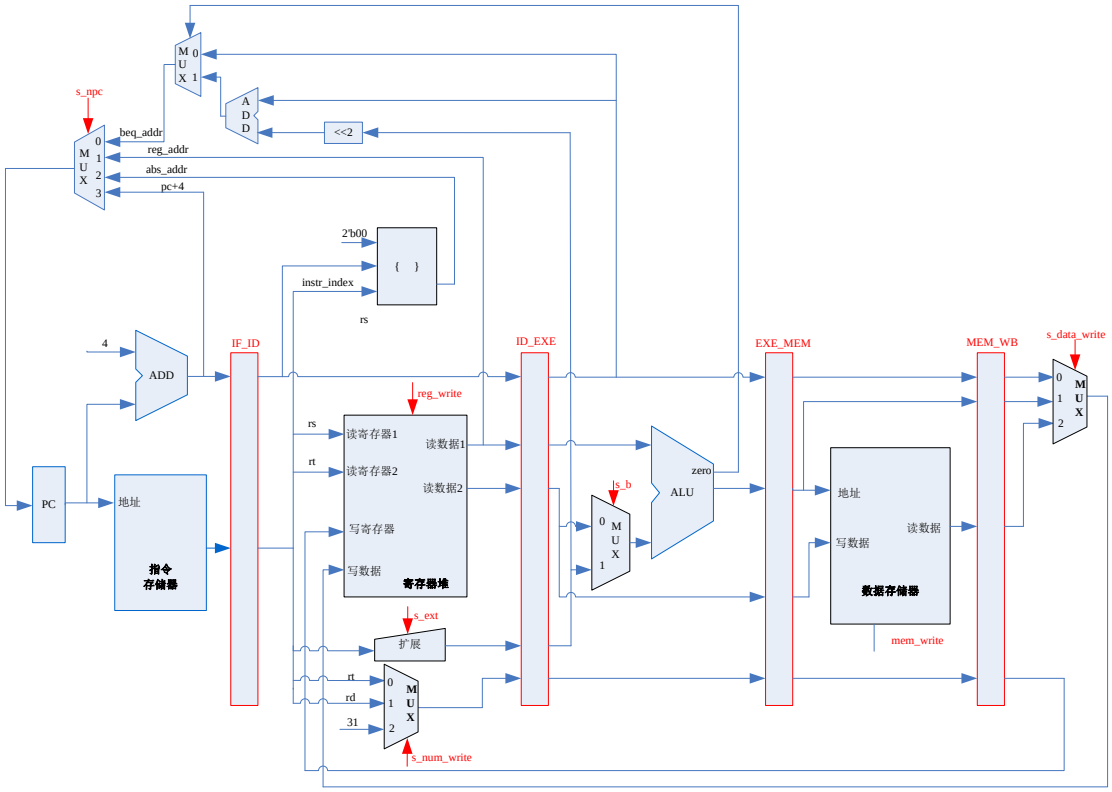
代码实现：

```
`include "pipe_reg.v"
`include "pipe_reg_5.v"
`include "pipe_reg_1.v"
`include "pipe_reg_6.v"
`include "pipe_reg_2.v"

module  mem_wb(
    input clock,
    input reset,
    input [31:0] dpc4,
    input [31:0] busw,
    input [31:0] data,
    input [4:0] rw,
    input nnreg_write,
    input [1:0]nns_data_write,
    output [31:0] ddpc4,
    output [31:0] dbusw,
    output [31:0] ddata,
    output [4:0] drw,
    output nnnreg_write,
    output [1:0]nnns_data_write
    );

    pipe_reg pc_3(dpc4,clock,reset,ddpc4);
    pipe_reg busw_2(busw,clock,reset,dbusw);
    pipe_reg reg_data4(data,clock,reset,ddata);
    pipe_reg_5 rw_3(rw,clock,reset,drw);
    pipe_reg_1 reg_write_3(nnreg_write,clock,reset,nnnreg_write);
    pipe_reg_2 s_data_write_3(nns_data_write,clock,reset,nnns_data_write);
endmodule
```

顶层模块：

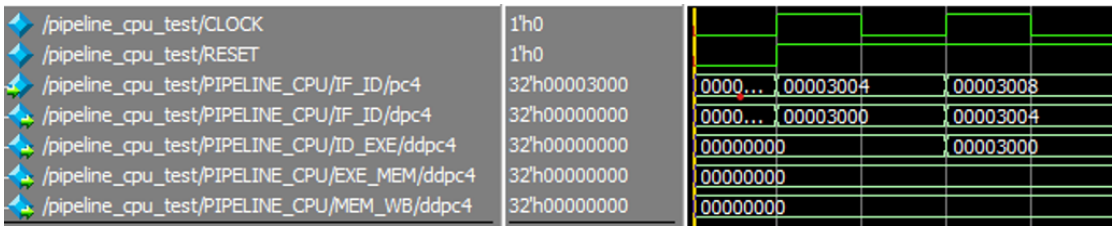
```
pc PC(pc,clock,reset,npc);
npc NPC(pc,Ext_imm32,imm26,a,Npcop,zero,npc);
im IM(instruction,pc);
if_id IF_ID(pc,instruction,clock,reset,dpc,dinst);
ctrl CTRL(opcode,func,aluop,reg_write,Extop,s_b,s_num_write,mem_write,s_data_write,Npcop);
mux3x5  MUX1(rt,rd,thirty_one,s_num_write,num_write);
ext EXT(imm16,Extop,Ext_imm32);
gpr GPR(a,b,clock,nnnreg_write,dinst,dddrw,data_write);
id_exe ID_EXE(clock,reset,dpc,a,b,Ext_imm32,num_write,aluop,s_b,reg_write,mem_write,s_data_write,
ddpc,ddata1,ddata2,dext_imm,drw,naluop,ns_b,nreg_write,nmem_write,ns_data_write);
mux2x32 MUX2(ddata2,dext_imm,ns_b,busb);
alu ALU(busw,zero,ddata1,busb,naluop);
exe_mem EXE_MEM(clock,reset,ddpc,busw,ddata2,drw,nreg_write,nmem_write,ns_data_write,dddpc,dbusw,dddata2,ddrw,nnreg_write,nnmem_write,nns_data_write);
dm DM(data_out,clock,nnmem_write,dbusw,dddata2);
mem_wb MEM_WB(clock,reset,dddpc,dbusw,data_out,ddrw,nnreg_write,nns_data_write,ddddpc,ddbusw,ddata,dddrw,nnnreg_write,nnns_data_write);
mux3x32 MUX3(ddbusw,ddata,ddddpc,nnns_data_write,data_write);
```

分析：整体的数据通路图如下：



增加流水线寄存器，把数据通路分成五部分。值得注意的是，一个信号可能需要传递多个阶段，因此需要注意命名的合理性，并且在同一时刻，由流水线寄存器分开的数据通路执行的是不同的指令。

以下是 testbench 部分截图以及仿真截图：

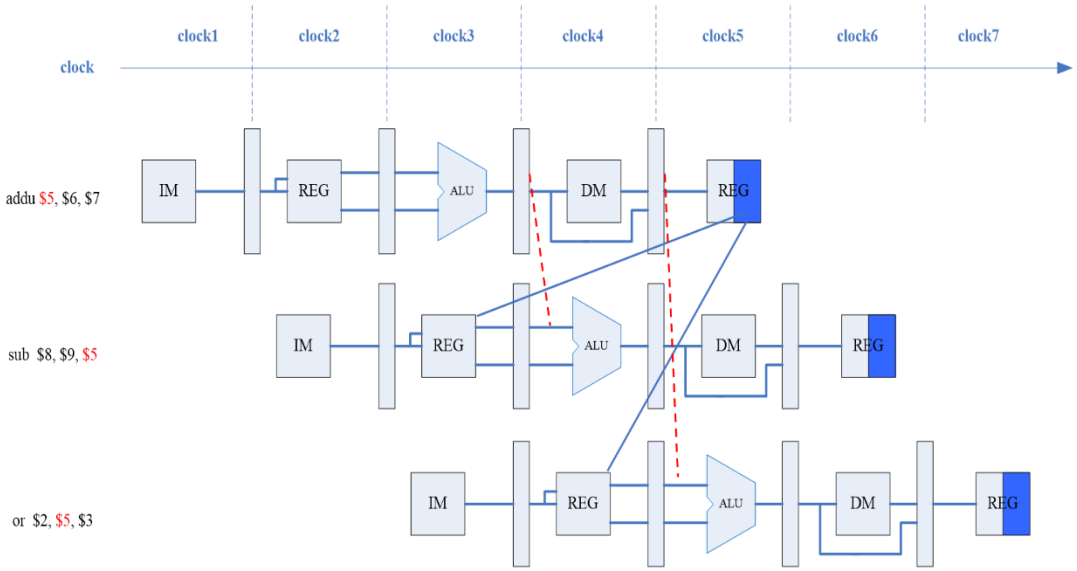


分析：在不考虑分支冒险和数据冒险时得到如上波形，易知在每个上升沿时，pc 值和流水线寄存器的值都会更新，并且前一个 pc 值会流向后一个流水线寄存器。

2.2 流水线 CPU(EXE 级数据冒险 1)

2.2.1 功能描述

在上一个实验（流水线 CPU(不考虑冒险)）的基础上处理以下 EXE 级数据冒险（包括双重数据冒险）：



2.2.2 主要模块及代码实现

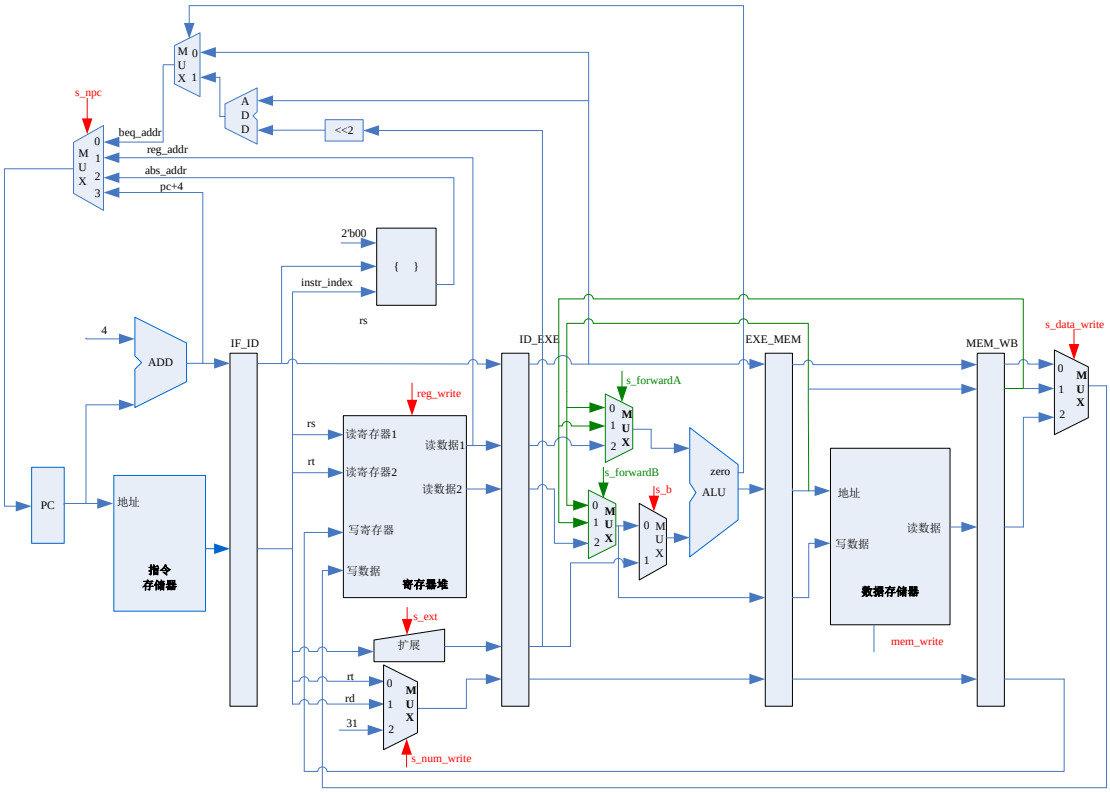
CTRL2

```
module ctrl2(
    input [4:0] dddrw,
    input [4:0] ddrw,
    input [4:0] nrt,
    input [4:0] nrd,
    output reg [1:0] s_forwardA,
    output reg [1:0] s_forwardB
    );

    always@(*)
    begin
        if(nrt == ddrw)
            s_forwardA = 2'b00;
        else if(nrt == dddrw)
            s_forwardA = 2'b01;
        else
            s_forwardA = 2'b10;
    end

    always@(*)
    begin
        if(nrd == ddrw)
            s_forwardB = 2'b00;
        else if(nrd == dddrw)
            s_forwardB = 2'b01;
        else
            s_forwardB = 2'b10;
   end
endmodule
```

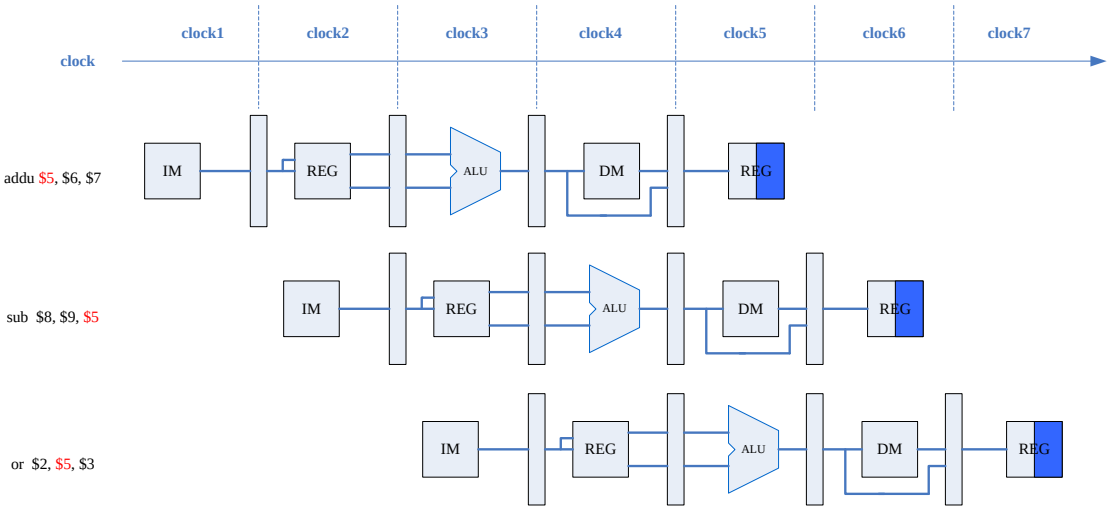
顶层模块：

```
pc PC(pc,clock,reset,npc);
npc NPC(pc,Ext_imm32,imm26,a,Npcop,zero,npc);
im IM(instruction,pc);
if_id IF_ID(pc,instruction,clock,reset,dpc,dinst);
ctrl CTRL(opcode,func,aluop,reg_write,Extop,s_b,s_num_write,mem_write,s_data_write,Npcop);
mux3x5  MUX1(rt,rd,thirty_one,s_num_write,num_write);
ext EXT(imm16,Extop,Ext_imm32);
gpr GPR(a,b,clock,nnnreg_write,dinst,dddrw,data_write);
id_exe ID_EXE(clock,reset,dpc,a,b,Ext_imm32,num_write,aluop,s_b,reg_write,mem_write,s_data_write,rs,rt,
ddpc,ddata1,ddata2,dext_imm,drw,naluop,ns_b,nreg_write,nmem_write,ns_data_write,nrs,nrt);
ctrl2 CTRL2(dddrw,ddrw,nrs,nrt,s_forwardA,s_forwardB);
mux3x32 MUX4(dbusw,ddbusw,ddata1,s_forwardA,ddata1_s);
mux3x32 MUX5(dbusw,ddbusw,ddata2,s_forwardB,ddata2_s);
mux2x32 MUX2(ddata2_s,dext_imm,ns_b,busb);
alu ALU(busw,zero,ddata1_s,busb,naluop);
exe_mem EXE_MEM(clock,reset,ddpc,busw,ddata2_s,drw,nreg_write,nmem_write,ns_data_write,dddpc,dbusw,dddata2,ddrw,nnreg_write,nnmem_write,nns_data_write);
dm DM(data_out,clock,nnmem_write,dbusw,dddata2);
mem_wb MEM_WB(clock,reset,dddpc,dbusw,data_out,ddrw,nnreg_write,nns_data_write,ddddpc,ddbusw,ddata,dddrw,nnnreg_write,nnns_data_write);
mux3x32 MUX3(ddbusw,ddata,ddddpc,nnns_data_write,data_write);
```

分析：整体的数据通路图如下：



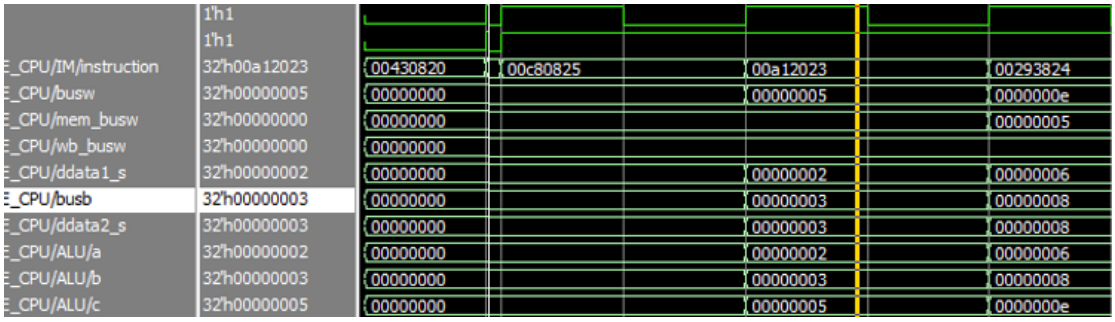
EXE 级数据冒险（1）——寄存器的新值是 ALU 的计算结果的示意图如下：



需要注意的是，EXE 级需要从寄存器堆读数据，如果读取时数据不是最新数据，就会产生数据冒险。$5 号寄存器的值在 clock4 就可以得到。如果 ALU 源操作数$0 号寄存器，则不需要旁路。数据通路图中增加了一个旁路单元，产生旁路控制信号 s_forwardA 和 s_forwardB。

以下是 testbench 部分截图以及仿真截图：
用到的汇编：

```
1:0x00430820 :  add $1 ,$2 ,$3
2:0x00c80825  :  or $1, $6, $8
3:0x00a12023  :  subu $4 ,$ 5, $1
4:0x00293824  :  and $7 ,$1 ,$ 9
```



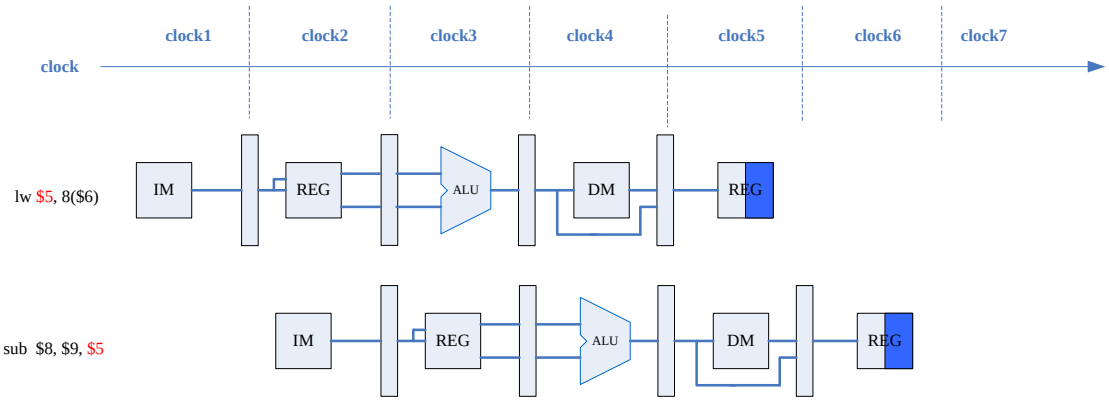
分析：

通过分析波形图可知，前两条指令都能够正常运算。而第三条指令和第四条指令中的 alu 分别成功替换成计算结果，这说明能够正常的进行冒险的运算，数据传输正常。困难之处在于判断是否发生数据冒险和传输正确的数据。

2.3 流水线 CPU（EXE 级数据冒险 2）

2.3.1 功能描述

EXE 级数据冒险（2）——寄存器新值是从 DM 中读出的数据，在 EXE 级数据冒险 1 的基础上处理以下数据冒险：



2.3.2 主要模块及代码实现

IF_ID 寄存器：

较之前的更改之处已在图中标出

```
`include "pipe_reg_wrn.v"
module if_id(pc4, inst, clock, reset, wrn,dpc4, dinst);
    input [31:0] pc4;
    input [31:0] inst;
    input   clock;
    input   reset;
    input   wrn;
    output [31:0] dpc4;
    output [31:0] dinst;

    pipe_reg_wrn nextpc(pc4,clock,reset,wrn,dpc4);
    pipe_reg_wrn id(inst,clock,reset,wrn,dinst);
endmodule
```

ID_EXE 寄存器：

```
`include "pipe_reg.v"
`include "pipe_reg_5.v"
`include "pipe_reg_1.v"
`include "pipe_reg_6.v"
`include "pipe_reg_2.v"
module id_exe(
    input clock,
    input reset,
    input [31:0]dpc4,
    input [31:0]data1,
    input [31:0]data2,
    input [31:0]ext_imm,
    input [4:0] rw,
    input [4:0] aluop,
    input s_b,
    input reg_write,
    input mem_write,
    input [1:0] s_data_write,
    input [4:0] rs,
    input [4:0] rt,
    input  memtoreg,
    input  wrn,
    output  reg [31:0] ddpc4,
    output  reg [31:0] ddata1,
    output  reg [31:0] ddata2,
    output  reg [31:0] dext_imm,
    output  reg [4:0]  drw,
    output  reg [4:0] naluop,
    output reg ns_b,
    output reg nreg_write,
    output reg nmem_write,
    output reg [1:0]ns_data_write,
    output reg [4:0] nrs,
    output reg [4:0] nrt,
    output reg ememtoreg
    );
```

```
always@(posedge clock , negedge reset)
begin
    if((!wrn)||(!reset))
    begin
        ddpc4 = 32'd0;
        ddata1 = 32'd0 ;
        ddata2 =32'd0 ;
        dext_imm = 32'd0;
        drw = 5'd0;
        naluop = 5'd0;
        ns_b =1'd0;
        nreg_write = 1'd0;
        nmem_write = 1'd0;
        ns_data_write = 2'd0;
        nrs = 5'd0;
        nrt = 5'd0;
        ememtoreg = 1'd0;
    end
    else
    begin
        ddpc4 = dpc4;
        ddata1 = data1 ;
        ddata2 = data2 ;
        dext_imm = ext_imm;
        drw = rw;
        naluop = aluop;
        ns_b = s_b;
        nreg_write = reg_write;
        nmem_write = mem_write;
        ns_data_write = s_data_write;
        nrs = rs;
        nrt = rt;
        ememtoreg = memtoreg;
    end
end
```

此部分改动较大，当 reset 或 wrn 为 0，需要令通过它的所有信号赋 0，也就是说，不仅要增加使能信号来传递阻塞信息，还需注意让 exe、wb、mem 分时序停止工作。

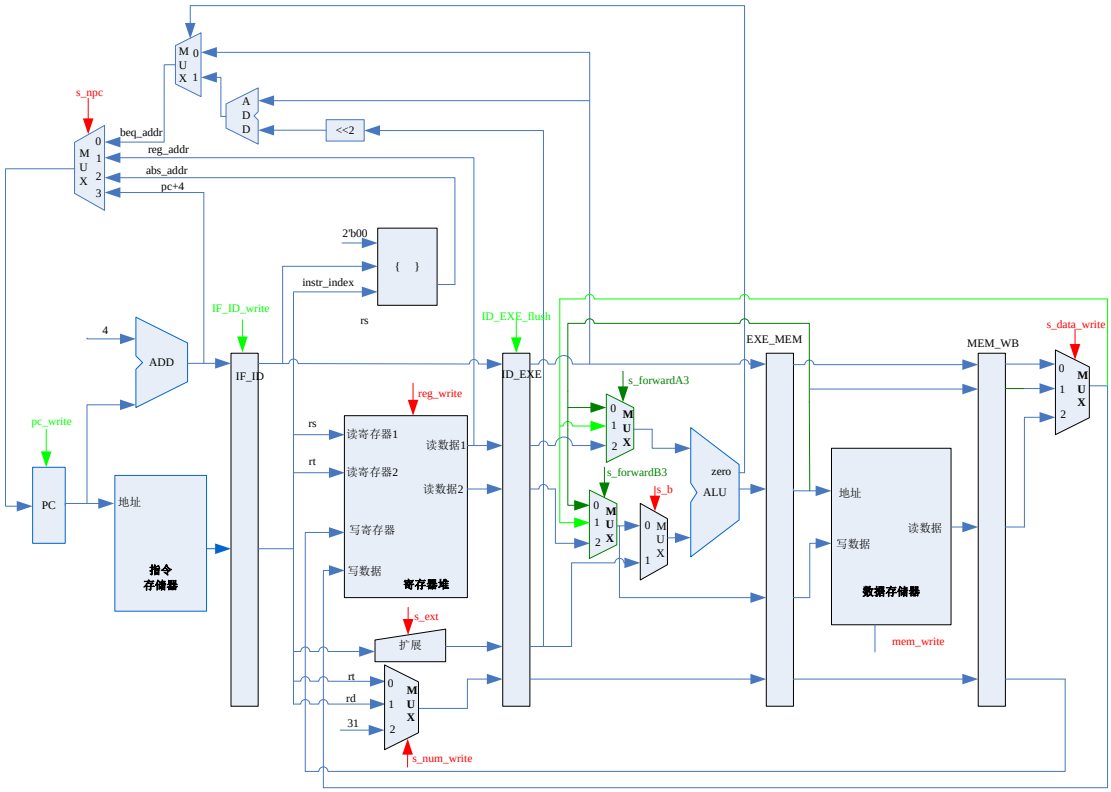
MEM_WB 寄存器：

```
`include "pipe_reg.v"
`include "pipe_reg_5.v"
`include "pipe_reg_1.v"
`include "pipe_reg_6.v"
`include "pipe_reg_2.v"
module  mem_wb(
    input clock,
    input reset,
    input [31:0] dpc4,
    input [31:0] busw,
    input [31:0] data,
    input [4:0] rw,
    input nnreg_write,
    input [1:0]nns_data_write,
    input wrn,
    output [31:0] ddpc4,
    output [31:0] dbusw,
    output [31:0] ddata,
    output [4:0] drw,
    output nnnreg_write,
    output [1:0]nnns_data_write
    );

    pipe_reg pc_3(dpc4,clock,reset,wrn,ddpc4);
    pipe_reg busw_2(busw,clock,reset,wrn,dbusw);
    pipe_reg reg_data4(data,clock,reset,wrn,ddata);
    pipe_reg_5 rw_3(rw,clock,reset,wrn,drw);
    pipe_reg_1 reg_write_3(nnreg_write,clock,reset,wrn,nnnreg_write);
    pipe_reg_2 s_data_write_3(nns_data_write,clock,reset,wrn,nnns_data_write);

endmodule
```

顶层模块：

```
pc PC(pc,clock,reset,wrn,npc);
npc NPC(pc,Ext_imm32,imm26,a,Npcop,zero,npc);
im IM(instruction,pc);
if_id IF_ID(pc,instruction,clock,reset,wrn,dpc,dinst);
ctrl CTRL(opcode,func,aluop,reg_write,Extop,s_b,s_num_write,mem_write,s_data_write,Npcop,memtoreg);
mux3x5  MUX1(rt,rd,thirty_one,s_num_write,num_write);
ext EXT(imm16,Extop,Ext_imm32);
gpr GPR(a,b,clock,nnnreg_write,dinst,dddrw,data_write);
id_exe ID_EXE(clock,reset,dpc,a,b,Ext_imm32,num_write,aluop,s_b,reg_write,mem_write,s_data_write,rs,rt,memtoreg,wrn,
ddpc,ddata1,ddata2,dext_imm,drw,naluop,ns_b,nreg_write,nmem_write,ns_data_write,nrs,nrt,ememtoreg);
ctrl2 CTRL2(dddrw,ddrw,drw,nrs,nrt,rs,rt,nreg_write,nnreg_write,ememtoreg,memtoreg,s_forwardA,s_forwardB,wrn);
mux3x32 MUX4(dbusw,data_write,ddata1,s_forwardA,ddata1_s);
mux3x32 MUX5(dbusw,data_write,ddata2,s_forwardB,ddata2_s);
mux2x32 MUX2(ddata2_s,dext_imm,ns_b,busb);
alu ALU(busw,zero,ddata1_s,busb,naluop);
exe_mem EXE_MEM(clock,reset,ddpc,busw,ddata2_s,drw,nreg_write,nmem_write,ns_data_write,ememtoreg,mwrn,
                dddpc,dbusw,dddata2,ddrw,nnreg_write,nnmem_write,nns_data_write,mmemtoreg);
dm DM(data_out,clock,nnmem_write,dbusw,dddata2);
mem_wb MEM_WB(clock,reset,dddpc,dbusw,data_out,ddrw,nnreg_write,nns_data_write,mwrn,
                ddddpc,ddbusw,ddata,dddrw,nnnreg_write,nnns_data_write);
mux3x32 MUX3(ddbusw,ddata,ddddpc,nnns_data_write,data_write);
```

分析：总体数据通路图如下：



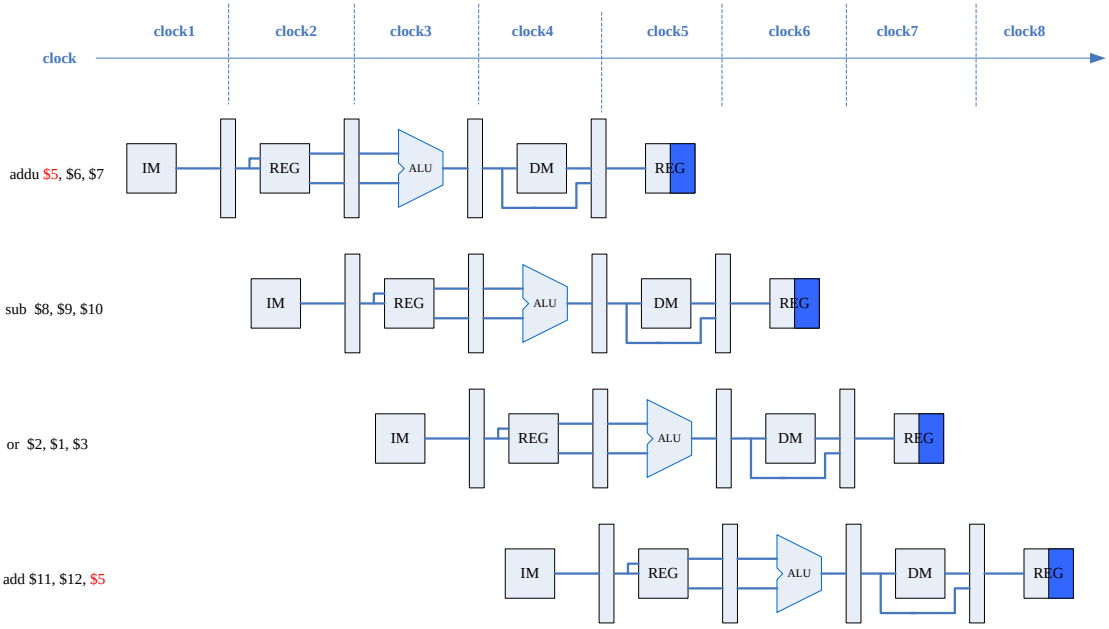
根据数据通路图可知，需要注意的是$5 号寄存器的值是 DM 的输出，它的值在 clock5 时才能够得到，sub 指令必须延迟一个周期再执行，也就是说流水线必须阻塞一个周期。，并且 sub 指令在 clock4 时需要最新的$5 号寄存器的值。

对于阻塞问题，由于 IF_ID 流水线寄存器不更新，这样 ID 级译码的指令就还是同一条指令，所以需要增加写使能信号 IF_ID_write。同时 PC 寄存器也不更新，所以同理也需要增加写使能信号 pc_write。而 ID_EXE 流水线寄存器增加了 flush 信号，把控制信号都改为 0，这样当前指令无效 。

2.4 流水线 CPU（EXE 级数据冒险 3）

2.4.1 功能描述

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。



2.4.2 主要模块及代码实现

EXE_MEM 模块：

```
module  exe_mem(clock,reset,dpc4,busw,data2,rw,nreg_write,nmem_write,ns_data_write,
ememtoreg,mem_r,ddpc4,dbusw,ddata2,drw,nnreg_write,nnmem_write,nns_data_write,
mmemtoreg);
    input clock,reset;
    input [31:0] dpc4,busw,data2;
    input [4:0] rw;
    input nreg_write,nmem_write,ememtoreg,mem_r;
    input [1:0] ns_data_write;
    output reg [31:0] ddpc4,dbusw,ddata2;
    output reg [4:0] drw;
    output reg  nnreg_write,nnmem_write,mmemtoreg;
    output reg [1:0]nns_data_write;

    always @(posedge clock,negedge reset)
    begin
        if(~reset) {ddpc4,dbusw,ddata2,drw,nnreg_write,nnmem_write,nns_data_write,
        mmemtoreg}={32'h0000_0000,32'h0000_0000,32'h0000_0000,5'b00000,1'b0,1'b0,2'b00
        ,1'b0};
        else if(mem_r) {ddpc4,dbusw,ddata2,drw,nnreg_write,nnmem_write,nns_data_write,
        mmemtoreg}={dpc4,busw,data2,rw,nreg_write,nmem_write,ns_data_write,ememtoreg};
    end

endmodule
```

EXT 模块：

```
module ext(imm16,Extop,Ext_imm32);
  input [15:0] imm16;
  input Extop;
  output reg [31:0]Ext_imm32;



always @(*)
begin
  if(Extop)
    if($signed(imm16) >= 0)
      Ext_imm32 =  {16'h0000,imm16};
    else
      Ext_imm32 =  {16'hffff,imm16};
  else Ext_imm32 = {16'h0000,imm16};

end
```

ID_EXE 模块：

```
module id_exe(
    input clock,reset,
    input [31:0]dpc4,data1,data2,ext_imm,
    input [4:0] rw, aluop,
    input s_b,reg_write,mem_write,
    input [1:0] s_data_write,
    input [4:0] rs,rt,
    input  memtoreg,pc_write,pc_write2,flush,
    output  reg [31:0] ddpc4,ddata1,ddata2,dext_imm,
    output  reg [4:0]  drw,naluop,
    output reg ns_b,nreg_write,nmem_write,
    output reg [1:0]ns_data_write,
    output reg [4:0] nrs,[4:0] nrt,
    output reg ememtoreg,mem_flush
);

always@(posedge clock , negedge reset)
 begin
  if((~pc_write)||(~reset)||(~pc_write2)){ddpc4,ddata1,ddata2,dext_imm,drw,naluop,ns_b
  ,nreg_write,nmem_write,ns_data_write,nrs,nrt,ememtoreg}={32'd0,32'd0,32'd0,32'd0,
  5'd0,5'd0,1'd0,1'd0,1'd0,1'd0,2'd0,5'd0,5'd0,1'd0};
  else {ddpc4,ddata1,ddata2,dext_imm,drw,naluop,ns_b,nreg_write,nmem_write,
  ns_data_write,nrs,nrt,ememtoreg,mem_flush}={dpc4,data1,data2,ext_imm,rw,aluop,s_b,
  reg_write,mem_write,s_data_write,rs,rt,memtoreg,flush};
 end

endmodule
```

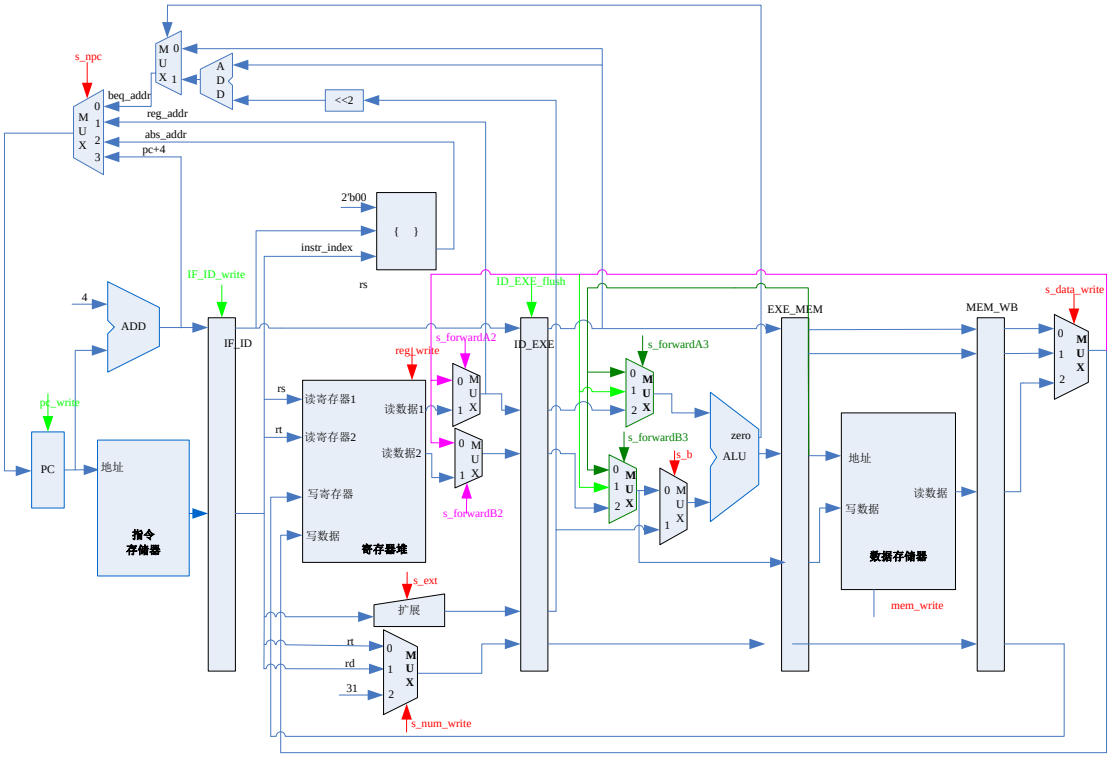
MEM_WB 模块：

```
module  mem_wb(
    input clock,reset,
    input [31:0] dpc4,busw,data,
    input [4:0] rw,
    input nnreg_write,
    input [1:0]nns_data_write,
    input [31:0] mem_data2,
    input mem_r,
    output reg [31:0] ddpc4,dbusw,ddata,
    output reg [4:0] drw,
    output reg nnnreg_write,
    output reg [1:0]nnns_data_write,
    output reg [31:0] wb_data2
    );


    always @(posedge clock,negedge reset)
    begin
        if(~reset) {ddpc4,dbusw,ddata,drw,nnnreg_write,wb_data2,nnns_data_write}={
        32'h0000_0000,32'h0000_0000,32'h0000_0000,5'b00000,1'b0,32'd0,2'b00};
        else if(mem_r) {ddpc4,dbusw,ddata,drw,nnnreg_write,wb_data2,nnns_data_write}={
        dpc4,busw,data,rw,nnreg_write,mem_data2,nns_data_write};
    end

endmodule
```

顶层模块：

```
pc PC(pc,clock,reset,pc_write,pc_write_2,npc);//ok
npc NPC(pc,id_ext_num,Ext_imm32,id_ins[25:0],for_id_reg1Data,npc_op,zeroone,npc);
//ok
im IM(if_ins,pc);//ok
if_id IF_ID(pc,if_ins,clock,reset,IF_ID_flu,pc_write,pc_write_2,EXE_flu,
id_ext_num,id_ins);//ok
ctrl CTRL(id_ins[31:26],id_ins[5:0],aluop,id_reg_write,exSer,id_s_b,s_num_write,
id_mem_write,s_data_write,npc_op,mem_to_reg);//ok
ext EXT(id_ins[15:0],exSer,Ext_imm32);//ok
gpr GPR(for_id_reg1Data,for_id_reg2Data,clock,wb_reg_write,id_ins,wb_rw,
data_write); //ok

id_exe ID_EXE(clock,reset,id_ext_num,for_ex_reg1Data,for_ex_reg2Data,Ext_imm32,
num_write,aluop,id_s_b,id_reg_write,id_mem_write,s_data_write,rs,rt,mem_to_reg,
pc_write,pc_write_2,EXE_flu,
exe_pc,ex_reg1Data,ex_reg2Data,exe_ext_imm,exe_rw,exe_alu_op,exe_s_b,
exe_reg_write,exe_mem_write,exe_s_data_write,exe_rs,exe_rt,exe_mtor,mem_flu);
forward CTRL2(wb_rw,mem_rw,exe_rw,exe_rs,exe_rt,rs,rt,for_ex_reg1Data,
for_ex_reg2Data,exe_reg_write,mem_reg_write,wb_reg_write,exe_mtor,mem_mtor,npc_op
,EXE_flu,mem_flu,mem_mem_write,
s_forwardA,s_forwardB,s_forwardA2,s_forwardB2,wst,IF_ID_flu,zeroone,pc_write_2,
pc_write);

alu ALU(ex_alu_res,id_reg_data,alu_result_b,exe_alu_op);//ok
exe_mem EXE_MEM(clock,reset,exe_pc,ex_alu_res,id_reg_data2,exe_rw,exe_reg_write,
exe_mem_write,exe_s_data_write,exe_mtor,mem_pc_write,
            mem_pc,mem_alu_res,mem_reg2Data,mem_rw,mem_reg_write,mem_mem_write,
            mem_s_data_write,mem_mtor);
dm DM(mem_mem_data,clock,mem_mem_write,mem_alu_res,mem_reg2Data);//ok
mem_wb MEM_WB(clock,reset,mem_pc,mem_alu_res,mem_mem_data,mem_rw,mem_reg_write,
mem_s_data_write,mem_reg2Data,mem_pc_write,
            wb_pc,wb_alu_result,wb_mem_data,wb_rw,wb_reg_write,wb_s_data_write,
            wb_data2);//ok


mux3  MUX3(wb_alu_result,wb_mem_data,s_wb_pc,wb_s_data_write,data_write);//ok
mux4 #(5)  MUX1(rt,rd,32'b1111_1111_1111_1111_1111_1111_1111_1111,5'b00000,
s_num_write,num_write);//ok
mux3  MUX4(mem_alu_res,data_write,ex_reg1Data,s_forwardA,id_reg_data);//ok
mux3  MUX5(mem_alu_res,data_write,ex_reg2Data,s_forwardB,reg_tmpdata);//ok
mux2  MUX8(reg_tmpdata,data_write,wst,id_reg_data2);//ok
mux2  MUX2(id_reg_data2,exe_ext_imm,exe_s_b,alu_result_b);//ok
mux3  MUX6(data_write,mem_alu_res,for_id_reg1Data,s_forwardA2,for_ex_reg1Data);
//ok
mux3  MUX7(data_write,mem_alu_res,for_id_reg2Data,s_forwardB2,for_ex_reg2Data);
//ok
```

分析：总体数据通路图如下：

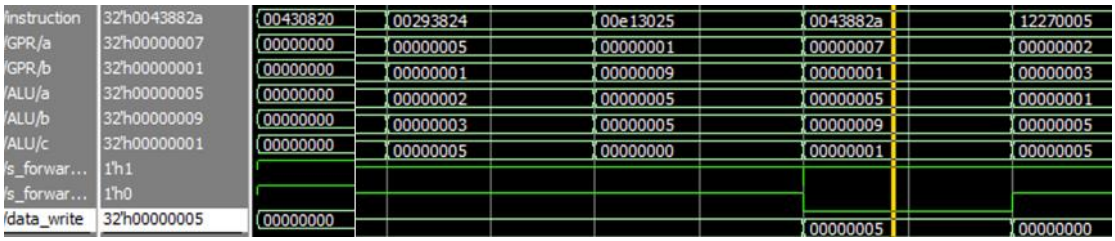


理想化的寄存器堆采用前半个周期写后半个周期读，所以不存在这种数据冒险。而在实际实现中，寄存器堆在 WB 级的结束时才写入值，所以 add 指令在 clock5 读取的$5 号寄存器的值是旧值，存在数据冒险。从图中可以看出我们需要在 MEM 阶段的 out 与 REG 阶段的从寄存器中取出操作数进行选择。同时增加两个选择信号：s_forwardA2 和 s_forwardB2

写寄存器的问题：%50 点是信号在转变过程中电压处于高电平和低电平之间中间点的位置，输出相应输入的改变需要一定的时间。

以下是 testbench 部分截图以及仿真截图：
用到的汇编：

```
1:00430820:   add     $1,$2,$3
2:00a12023:   subu    $6,$7,$1
3:00293824:   and     $8,$1,$4
4:00e13025:   or      $5,$8,$1
```

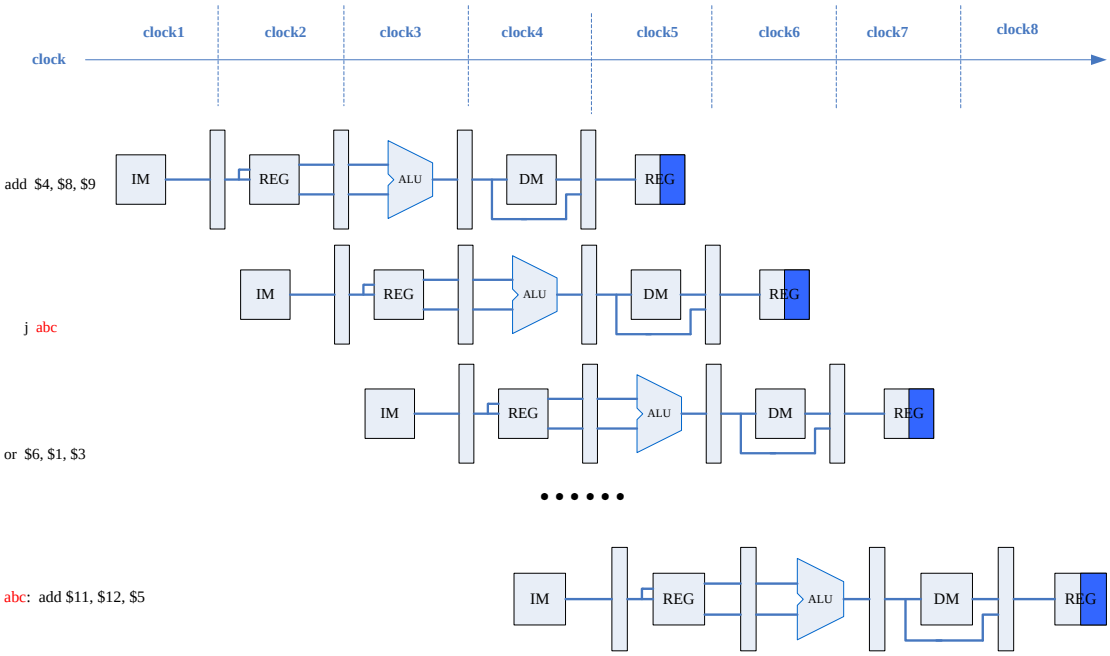


通过分析图像可知，在 or 指令的 ID 阶段的 data_write 为 add 指令的结果，之后将其作为操作数 2，下一个 clock cycle 可以看到 alu.b 的值也换成了 data_write 的值，所以可以证明数据冒险能够准确传输数据。

2.5 J, JR, JAL 指令控制冒险

2.5.1 功能描述

处理 J, JR, JAL 指令引入的控制冒险。



2.5.2 主要模块及代码实现

EXE_MEM 模块：

```
module  exe_mem(clock,reset,dpc4,busw,data2,rw,nreg_write,nmem_write,ns_data_write,
ememtoreg,mem_r,ddpc4,dbusw,ddata2,drw,nnreg_write,nnmem_write,nns_data_write,
mmemtoreg);
    input clock,reset;
    input [31:0] dpc4,busw,data2;
    input [4:0] rw;
    input nreg_write,nmem_write,ememtoreg,mem_r;
    input [1:0] ns_data_write;
    output reg [31:0] ddpc4,dbusw,ddata2;
    output reg [4:0] drw;
    output reg  nnreg_write,nnmem_write,mmemtoreg;
    output reg [1:0]nns_data_write;

    always @(posedge clock,negedge reset)
    begin
        if(~reset) {ddpc4,dbusw,ddata2,drw,nnreg_write,nnmem_write,nns_data_write,
        mmemtoreg}={32'h0000_0000,32'h0000_0000,32'h0000_0000,5'b00000,1'b0,1'b0,2'b00
        ,1'b0};
        else if(mem_r) {ddpc4,dbusw,ddata2,drw,nnreg_write,nnmem_write,nns_data_write,
        mmemtoreg}={dpc4,busw,data2,rw,nreg_write,nmem_write,ns_data_write,ememtoreg};
    end

endmodule
```

ID_EXE 模块：

```
module id_exe(
    input clock,
    input reset,
    input [31:0]dpc4,
    input [31:0]data1,
    input [31:0]data2,
    input [31:0]ext_imm,
    input [4:0] rw,
    input [4:0] aluop,
    input s_b,
    input reg_write,
    input mem_write,
    input [1:0] s_data_write,
    input [4:0] rs,
    input [4:0] rt,
    input  memtoreg,
    input  pc_write,
    input flush,
    output  reg [31:0] ddpc4,
    output  reg [31:0] ddata1,
    output  reg [31:0] ddata2,
    output  reg [31:0] dext_imm,
    output  reg [4:0]  drw,
    output  reg [4:0] naluop,
    output reg ns_b,
    output reg nreg_write,
    output reg nmem_write,
    output reg [1:0]ns_data_write,
    output reg [4:0] nrs,
    output reg [4:0] nrt,
    output reg ememtoreg,
    output reg mem_flush

);
always@(posedge clock , negedge reset)
 begin
  if((~pc_write)||(~reset)) {ddpc4,ddata1,ddata2,dext_imm,drw,naluop,ns_b,nreg_write,
  nmem_write,ns_data_write,nrs,nrt,ememtoreg}={32'd0,32'd0,32'd0,32'd0,5'd0,5'd0,1'd0,
  1'd0,1'd0,2'd0,5'd0,5'd0,1'd0};
  else {ddpc4,ddata1,ddata2,dext_imm,drw,naluop,ns_b,nreg_write,nmem_write,
  ns_data_write,nrs,nrt,ememtoreg,mem_flush}={dpc4,data1,data2,ext_imm,rw,aluop,s_b,
  reg_write,mem_write,s_data_write,rs,rt,memtoreg,flush};
  end

endmodule
```

MEM_WB 模块：

```
module  mem_wb(clock,reset,dpc4,busw,data,rw,nnreg_write,nns_data_write,mem_r,ddpc4,
dbusw,ddata,drw,nnnreg_write,nnns_data_write);
    input clock,reset;
    input [31:0] dpc4,busw,data;
    input [4:0] rw;
    input nnreg_write,mem_r;
    input [1:0]nns_data_write;
    output reg [31:0] ddpc4,dbusw,ddata;
    output reg [4:0] drw;
    output reg nnnreg_write;
    output reg [1:0]nnns_data_write;


    always @(posedge clock,negedge reset)
    begin
        if(~reset) {ddpc4,dbusw,ddata,drw,nnnreg_write,nnns_data_write}={32'h0000_0000
        ,32'h0000_0000,32'h0000_0000,5'b00000,1'b0,2'b00};
        else if(mem_r) {ddpc4,dbusw,ddata,drw,nnnreg_write,nnns_data_write}={dpc4,busw
        ,data,rw,nnreg_write,nns_data_write};
    end

endmodule
```

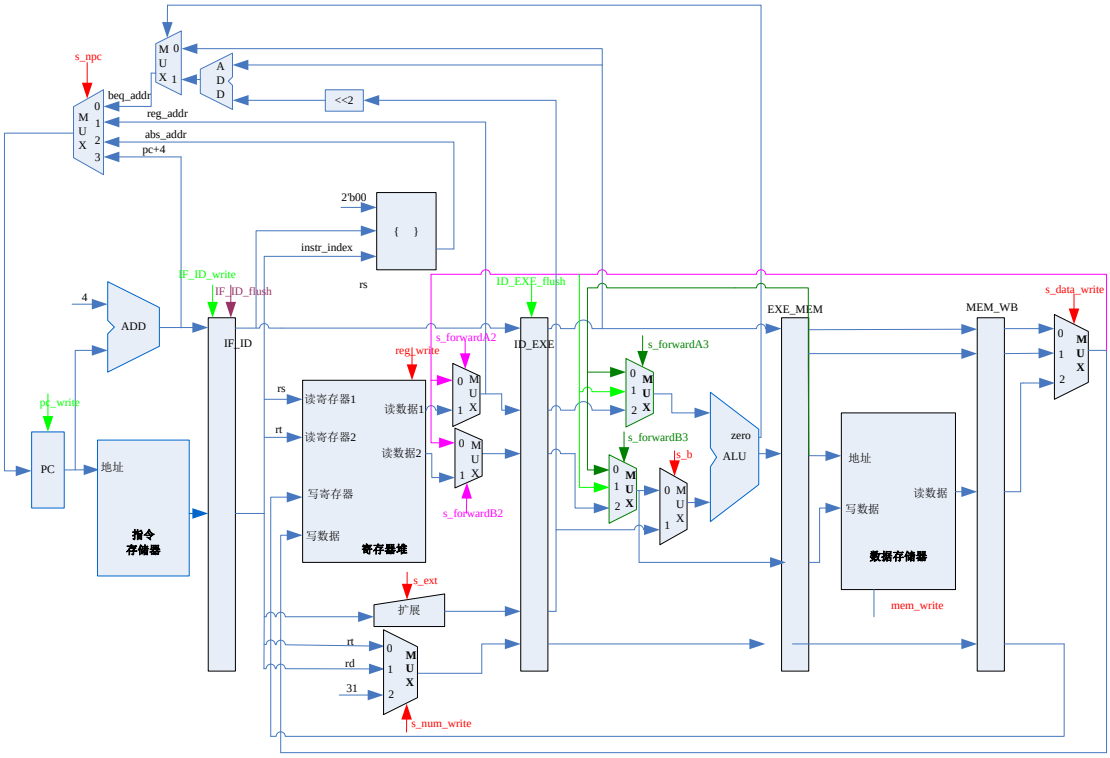
IF_ID 模块：

```
module if_id(pc4, ins, clock, reset,IF_ID_f, pc_write,EXE_f,id_ext_num, nst);

    input [31:0] pc4,ins;
    input clock,reset,IF_ID_f,pc_write;
    output reg  EXE_f;
    output reg [31:0] id_ext_num,nst;

    always @(posedge clock,negedge reset) begin
        if(!reset|| IF_ID_f) {id_ext_num,nst,EXE_f}<={32'h0000_0000,32'h0000_0000,1'b1
        };
        else if(pc_write) {id_ext_num,nst,EXE_f}<={pc4,ins,1'b0};
    end


endmodule
```

顶层模块：

```
    pc PC(pc,clock,reset,pc_write,npc);//ok
    npc NPC(pc,id_ext_num,Ext_imm32,id_ins[25:0],for_id_reg1data,npc_op,zero,npc);//ok
    im IM(if_ins,pc);//ok
    if_id IF_ID(pc,if_ins,clock,reset,IF_ID_flush,pc_write,EXE_flush,id_ext_num,
    id_ins);//ok
    ctrl CTRL(id_ins[31:26],id_ins[5:0],aluop,id_reg_write,exSer,id_s_b,s_num_write,
    id_mem_write,s_data_write,npc_op,id_mem_to_reg);//ok
    ext EXT(id_ins[15:0],exSer,Ext_imm32);//ok
    gpr GPR(for_id_reg1data,for_id_reg2data,clock,wb_reg_write,id_ins,wb_rw,
    data_write); //ok
    id_exe ID_EXE(clock,reset,id_ext_num,id_reg1Data,id_reg2Data,Ext_imm32,num_write,
    aluop,id_s_b,id_reg_write,id_mem_write,s_data_write,rs,rt,id_mem_to_reg,pc_write,
    EXE_flush,
    exe_pc,ex_reg1Data,ex_reg2Data,exe_ext_imm,exe_rw,exe_aluop,exe_s_b,exe_reg_write
    ,exe_mem_write,exe_s_data_write,exe_rs,exe_rt,exe_mtor,mem_flush);
    pipe_ctrl2 CTRL2(wb_rw,mem_rw,exe_rw,exe_rs,exe_rt,rs,rt,exe_reg_write,
    mem_reg_write,wb_reg_write,exe_mtor,mem_mtor,npc_op,EXE_flush,mem_flush,
    s_forwardA,s_forwardB,s_forwardA2,s_forwardB2,IF_ID_flush,pc_write);
    alu ALU(ex_alu_result,zero,for_ex_reg1Data,alu_result_b,exe_aluop);//ok
    exe_mem EXE_MEM(clock,reset,exe_pc,ex_alu_result,for_ex_reg2Data,exe_rw,
    exe_reg_write,exe_mem_write,exe_s_data_write,exe_mtor,mem_pc_write,
                mem_pc,mem_alu_result,mem_mem_data,mem_rw,mem_reg_write,
                mem_mem_write,mem_s_data_write,mem_mtor);
    dm DM(mem_reg2Data,clock,mem_mem_write,mem_alu_result,mem_mem_data);//ok
    mem_wb MEM_WB(clock,reset,mem_pc,mem_alu_result,mem_reg2Data,mem_rw,mem_reg_write
    ,mem_s_data_write,mem_pc_write,
                wb_pc,wb_alu_result,wb_mem_data,wb_rw,wb_reg_write,wb_s_data_write);
                //ok
    mux3 #(32) MUX3(wb_alu_result,wb_mem_data,s_wb_pc,wb_s_data_write,data_write);//ok
    mux3 #(5)  MUX1(rt,rd,32'b1111_1111_1111_1111_1111_1111_1111_1111,s_num_write,
    num_write);//ok
    mux3 #(32) MUX4(mem_alu_result,data_write,ex_reg1Data,s_forwardA,for_ex_reg1Data
    );//ok
    mux3 #(32) MUX5(mem_alu_result,data_write,ex_reg2Data,s_forwardB,for_ex_reg2Data
    );//ok
    mux2  MUX2(for_ex_reg2Data,exe_ext_imm,exe_s_b,alu_result_b);//ok

   mux2  MUX6 (data_write,for_id_reg1data,s_forwardA2,id_reg1Data);//ok
    mux2  MUX7(data_write,for_id_reg2data,s_forwardB2,id_reg2Data);//ok

endmodule
```

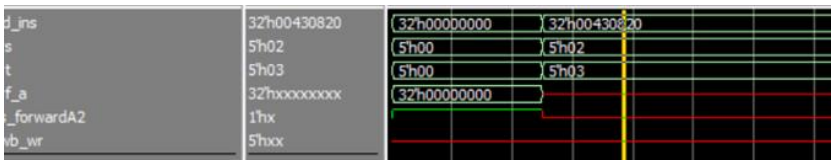
分析：完整数据通路图如下：



需要注意的是，在 clock3 时 j 指令确定了跳转地址。这时 or 指令已经被取指，因此我们必须要将它清除掉，同时，IF_ID 流水线寄存器需要增加一个 flush 信号 IF_ID_flush，并且在 clock3 时，正确的下条指令被取指执行。

以下是 testbench 部分截图以及仿真截图：

Address	Code	Basic	
0x00003000	0x014b4820	add $9,$10,$11	1: add $t1,$t2,$t3
0x00003004	0x08000c05	j 0x00003014	2: j abc
0x00003008	0x014b4825	or $9,$10,$11	3: or $t1,$t2,$t3
0x0000300c	0x01ae6025	or $12,$13,$14	4: or $t4,$t5,$t6
0x00003010	0x00c72825	or $5,$6,$7	5: or $a1,$a2,$a3
0x00003014	0x02538820	add $17,$18,$19	6: abc: add $s1,$s2,$s3



一开始观察 gpr 发现 J 指令没有产生作用，原来是由于之前没有跳转指令，而选择模块用 exe 级控制信号去选择，所以也有问题，修改后得以解决，开始的时候旁路模块写的也有问题，导致一些不是冒险的指令也被当作冒险被处理。完成此题需要在单周期跳转指令实现后增加阻塞信号，让流水线信号替代单周期信号，这里如果提完不全面就可能会出现无限循环清空寄存器一直 flush 等问题。

2.6 能够执行 addu 指令的单周期 CPU

2.6.1 功能描述

1)尽量提前确定下条指令的地址：如果在 EXE 级比较，需要阻塞两个周期才能确定下条指令的地址。可以增加硬件，将比较提前到 ID 级，减少阻塞时间。

2)预测（假设分支不发生）：只有在预测错误时才需要清除执行错误的指令。

2.6.2 主要模块及代码实现

EXE_MEM 模块：

```
module  exe_mem(clock,reset,pc_write,
                pc,ex_alu_result,data2,rw,reg_write,mem_write,s_data_write,ememtoreg,
                pc_out,alu_result_out,data2_out,rw_out,reg_write_out,mem_write_out,
                s_data_write_out,mmemtoreg);
    input clock;
    input reset;
    input pc_write;
    input [31:0] pc;
    input [31:0] ex_alu_result;
    input [31:0] data2;
    input [4:0] rw;
    input reg_write;
    input mem_write;
    input [1:0] s_data_write;
    input ememtoreg;
    output reg [31:0] pc_out;
    output reg [31:0] alu_result_out;
    output reg [31:0] data2_out;
    output reg [4:0] rw_out;
    output reg  reg_write_out;
    output reg mem_write_out;
    output reg [1:0]s_data_write_out;
    output reg mmemtoreg;


    always @(posedge clock,negedge reset)
    begin
        if(!reset)
        begin
            pc_out = 32'h0000_0000;
            alu_result_out = 32'h0000_0000;
            data2_out = 32'h0000_0000;
            rw_out = 5'b00000;
            reg_write_out = 1'b0;
            mem_write_out = 1'b0;
            s_data_write_out = 2'b00;
            mmemtoreg = 1'b0;
        end
        else if(pc_write)
        begin
            pc_out = pc;
            alu_result_out = ex_alu_result;
            data2_out = data2;
            rw_out =rw;
            reg_write_out = reg_write;
            mem_write_out = mem_write;
            s_data_write_out = s_data_write;
            mmemtoreg = ememtoreg;
        end
    end
```

MEM_WB 模块：

```
module  mem_wb(clock,reset,pc_write,
        mem_pc,mem_alu_result,data,rw,reg_write,s_data_write,
        wb_pc, wb_alu_result, ddata, drw, out_reg_write, out_s_data_write);
    input clock;
    input reset;
    input pc_write;
    input [31:0] mem_pc;
    input [31:0] mem_alu_result;
    input [31:0] data;
    input [4:0] rw;
    input reg_write;
    input [1:0]s_data_write;

    output reg [31:0] wb_pc;
    output reg [31:0] wb_alu_result;
    output reg [31:0] ddata;
    output reg [4:0] drw;
    output reg out_reg_write;
    output reg [1:0]out_s_data_write;

    always @(posedge clock,negedge reset)
    begin
        if(!reset)
        begin
            wb_pc = 32'h0000_0000;
            wb_alu_result = 32'h0000_0000;
            ddata = 32'h0000_0000;
            drw = 5'b00000;
            out_reg_write = 1'b0;
            out_s_data_write = 2'b00;
        end
        else if(pc_write)
        begin
            wb_pc = mem_pc;
            wb_alu_result = mem_alu_result;
            ddata = data;
            drw = rw;
            out_reg_write = reg_write;
            out_s_data_write = s_data_write;
        end

    end

endmodule
```

IF_ID 模块：

```
module if_id(clock, reset, pc_write, pc4, inst,IF_ID_flush, id_pc, dinst,EXE_flush);
    input   clock;
    input   reset;
    input   pc_write;
    input [31:0] pc4;
    input [31:0] inst;   //输入要保存的信号 pc4和inst
    input IF_ID_flush;
    output reg [31:0] id_pc;
    output reg [31:0] dinst;  //pc4转为dpc4, inst转为dinst
    output reg  EXE_flush;
    always @(posedge clock,negedge reset) begin
        if(!reset || IF_ID_flush)
        begin
            id_pc <= 32'h0000_0000;
            dinst <= 32'h0000_0000;
            EXE_flush <= 1'b1;
        end
        else if(pc_write)
        begin
            id_pc <= pc4;
            dinst <= inst;
            EXE_flush <= 1'b0;
        end
    end


endmodule
```

ID_EXE 模块：

```
module id_exe(clock,reset,pc_write,
        id_pc,data1,data2,ext_imm, rw,aluop,id_s_b, id_reg_write,id_mem_write,
        s_data_write,rs,rt,id_mem_to_reg,flush,
        ex_pc,ddata1,ddata2,dext_imm,drw,naluop,ns_b,nreg_write,nmem_write,
        ns_data_write,nrs,nrt,ex_mem_to_reg,mem_flush);
    input clock;
    input reset;
    input pc_write;
    input [31:0]id_pc;
    input [31:0]data1;
    input [31:0]data2;
    input [31:0]ext_imm;
    input [4:0] rw;
    input [4:0] aluop;
    input id_s_b;
    input id_reg_write;
    input id_mem_write;
    input [1:0] s_data_write;
    input [4:0] rs;
    input [4:0] rt;
    input  id_mem_to_reg;
    input flush;
    output  reg [31:0] ex_pc;
    output  reg [31:0] ddata1;
    output  reg [31:0] ddata2;
    output  reg [31:0] dext_imm;
    output  reg [4:0]  drw;
    output  reg [4:0] naluop;
    output reg ns_b;
    output reg nreg_write;
    output reg nmem_write;
    output reg [1:0]ns_data_write;
    output reg [4:0] nrs;
    output reg [4:0] nrt;
    output reg ex_mem_to_reg;
    output reg mem_flush;
```

```
always@(posedge clock , negedge reset)
begin
    if((!pc_write)||(!reset))
    begin
        ex_pc = 32'd0;
        ddata1 = 32'd0 ;
        ddata2 =32'd0 ;
        dext_imm = 32'd0;
        drw = 5'd0;
        naluop = 5'd0;
        ns_b =1'd0;
        nreg_write = 1'd0;
        nmem_write = 1'd0;
        ns_data_write = 2'd0;
        nrs = 5'd0;
        nrt = 5'd0;
        ex_mem_to_reg = 1'd0;
    end
    else
    begin
        ex_pc = id_pc;
        ddata1 = data1 ;
        ddata2 = data2 ;
        dext_imm = ext_imm;
        drw = rw;
        naluop = aluop;
        ns_b = id_s_b;
        nreg_write = id_reg_write;
        nmem_write = id_mem_write;
        ns_data_write = s_data_write;
        nrs = rs;
        nrt = rt;
        ex_mem_to_reg = id_mem_to_reg;
        mem_flush = flush;
    end
end
endmodule
```

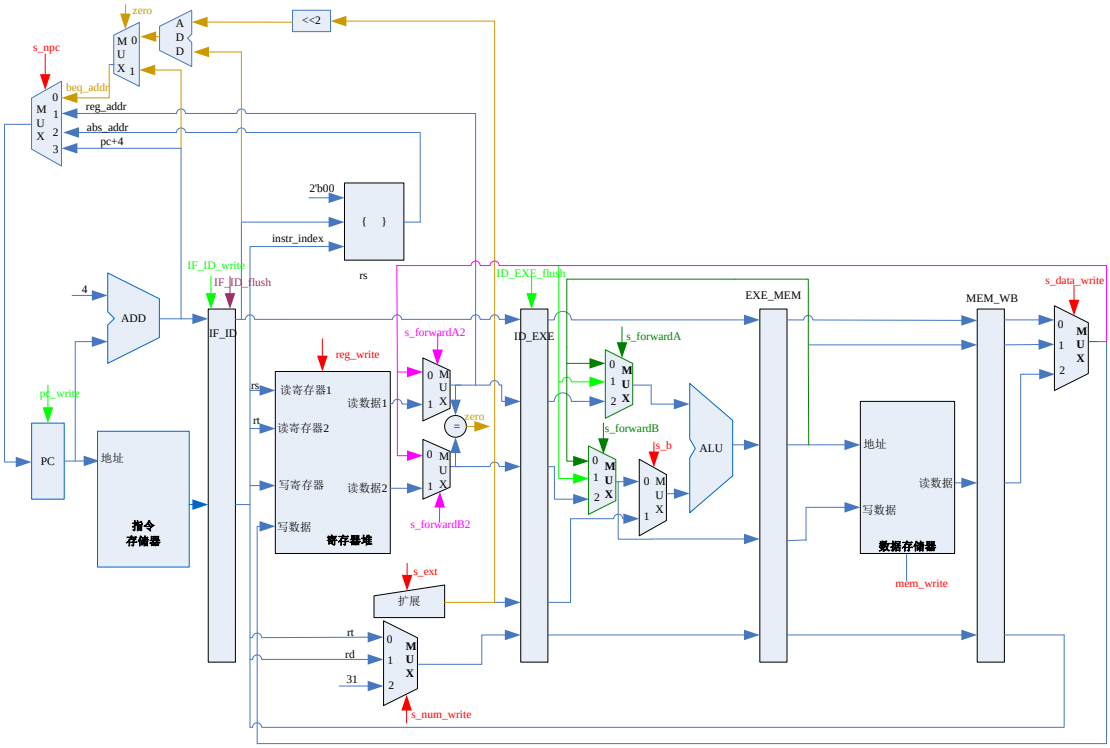
EXTENDER 模块：

```
module extender(imm,exSer,id_ext_num);
    input [15:0] imm;
    input exSer;
    output reg [31:0]id_ext_num;


always @(*)
begin
   if(exSer == 0)
      id_ext_num = {16'b0000000000000000,imm};
   else if(exSer == 1)
      if($signed(imm) >= 0)
        id_ext_num =  {16'b0000000000000000,imm};
      else
        id_ext_num =  {16'b1111111111111111,imm};
   else
        id_ext_num = 32'bxxxxxxxxxxxxxxxx;
end

endmodule
```

顶层模块：

```
    pc PC(pc,clock,reset,pc_write,npc);
    npc NPC(pc,id_pc,id_ext_num,id_ins[25:0] ,for_id_reg1Data,npc_op,zero_flag_2,npc);
    im IM(if_ins,pc);
    ctrl CTRL(id_ins[31:26],id_ins[5:0],exSer,id_reg_write,id_s_b,id_mem_write,aluop,
    s_num_write,s_data_write,npc_op,id_mem_to_reg);
    extender EXTENDER(id_ins[15:0],exSer,id_ext_num);
    gpr GPR(clock,wb_reg_write,id_ins,wb_rw,data_write,id_reg1Data,id_reg2Data);
    mux3 #(32) Hazard1_forward1 (mem_alu_result,data_write,ex_reg1Data,s_forwardA,
    for_ex_reg1Data);
    mux3 #(32) Hazard1_forward2 (mem_alu_result,data_write,ex_reg2Data,s_forwardB,
    for_ex_reg2Data);
    mux2 #(32) Hazard2_forward1 (data_write,id_reg1Data,s_forwardA2,for_id_reg1Data);
    mux2 #(32) Hazard2_forward2 (data_write,id_reg2Data,s_forwardB2,for_id_reg2Data);
    forward CONTROL(
        wb_rw,mem_rw,ex_rw,exe_rs,exe_rt,rs,rt,ex_reg_write,mem_reg_write,wb_reg_write,
        ex_mem_to_reg,mem_mem_to_reg,npc_op,EXE_flush,mem_flush,id_reg1Data,id_reg2Data,
        pc_write,s_forwardA,s_forwardB,s_forwardA2,s_forwardB2,IF_ID_flush,zero_flag_2);
    alu ALU(ex_alu_result,for_ex_reg1Data,alu_result_b,exe_alu_op,zero_flag);
    dm DM(clock,mem_mem_write,mem_alu_result,mem_reg2Data,mem_mem_data);
    if_id IF_ID(
        clock,reset,pc_write,
        pc,if_ins,IF_ID_flush,
        id_pc,id_ins,EXE_flush
        );
    id_exe ID_EXE(
        clock,reset,pc_write,
        id_pc,for_id_reg1Data,for_id_reg2Data,id_ext_num, num_write,aluop,     id_s_b,
        id_reg_write,id_mem_write,s_data_write,   rs,   rt,     id_mem_to_reg,EXE_flush,
        ex_pc,ex_reg1Data,    ex_reg2Data,    exe_ext_imm,ex_rw,    exe_alu_op,exe_s_b,
        ex_reg_write,ex_mem_write,ex_s_data_write,exe_rs,exe_rt,ex_mem_to_reg,mem_flush
        );

    mux3 #(5)  RegToWrite(rt,rd, 32'b1111_1111_1111_1111_1111_1111_1111_1111,s_num_write,
    num_write);
    mux2 #(32) ALUToInput(for_ex_reg2Data,exe_ext_imm,exe_s_b,alu_result_b);
    //选第二个运算数据，注意是用传的数据选择，这意味着选择信号也要传

    exe_mem EXE_MEM(
        clock,reset,mem_pc_write,
        ex_pc, ex_alu_result, for_ex_reg2Data, ex_rw, ex_reg_write, ex_mem_write,
        ex_s_data_write, ex_mem_to_reg,
        mem_pc,mem_alu_result,mem_reg2Data,    mem_rw,mem_reg_write,mem_mem_write,
        mem_s_data_write,mem_mem_to_reg
        );
    mem_wb MEM_WB(
        clock,reset,mem_pc_write,
        mem_pc,mem_alu_result,mem_mem_data,mem_rw,mem_reg_write,mem_s_data_write,
        wb_pc, wb_alu_result, wb_mem_data, wb_rw, wb_reg_write, wb_s_data_write
        );

    mux3 #(32) MemToWrite(wb_alu_result,wb_mem_data,s_wb_pc,wb_s_data_write,data_write);
```

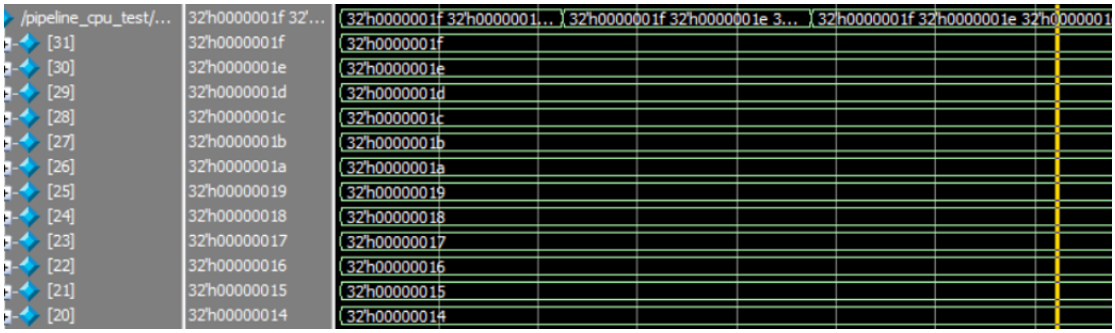
分析：完整的数据通路图如下：



经分析可知，把比较提前到 ID 级，只需要阻塞一个周期。预测分支不发生，所以只有分支发生时才需要阻塞一个周期。在 REG 时刻确定下一条指令的地址。

以下是 testbench 部分截图以及仿真截图：

Address	Code	Basic	
0x00003000	0x00a62021	addu $4, $5, $6	1: addu $a0, $a1, $a2
0x00003004	0x10850002	beq $4, $5, 0x00000002	2: beq $a0, $a1, loop1
0x00003008	0x0338b821	addu $23, $25, $24	3: addu $s7, $t9, $t8
0x0000300c	0x108b0001	beq $4, $11, 0x00000001	4: beq $a0, $t3, loop2
0x00003010	0x00831021	addu $2, $4, $3	6: addu $v0, $a0, $v1
0x00003014	0x00971021	addu $2, $4, $23	8: addu $v0, $a0, $s7

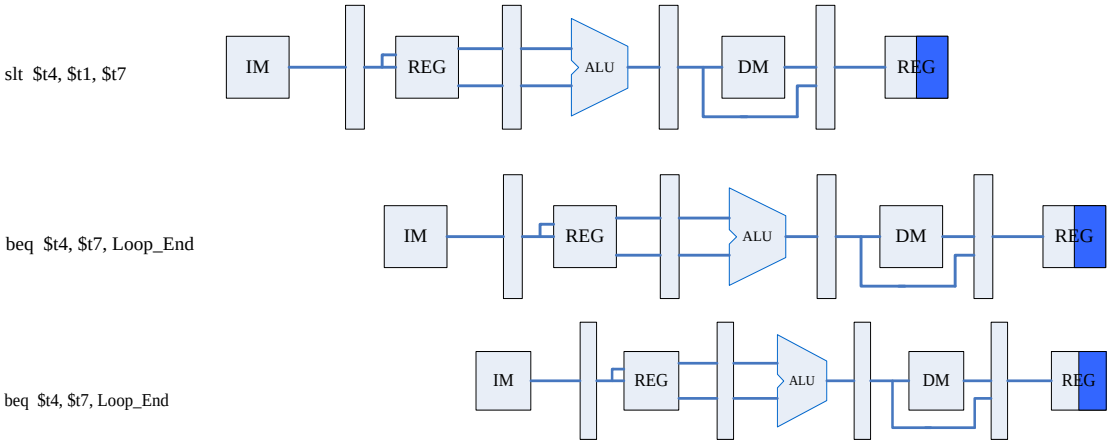


此题我们要解决新增的 ID 级比较器后信号传递的过程，通过分析波形图可知，beq 成功发生跳转，说明 beq 指令合理实现。

2.7 ID 级数据冒险(1)

2.7.1 功能描述

在上一个实验的基础上实现 ID 级数据冒险。



2.7.2 主要模块及代码实现

EXE_MEM 模块：

```
module  exe_mem(clock,reset,dpc4,busw,data2,rw,nreg_write,nmem_write,ns_data_write,ememtoreg,
mem_r,ddpc4,dbusw,ddata2,drw,nnreg_write,nnmem_write,nns_data_write,mmemtoreg);
    input clock,reset;
    input [31:0] dpc4,busw,data2;
    input [4:0] rw;
    input nreg_write,nmem_write,ememtoreg,mem_r;
    input [1:0] ns_data_write;
    output reg [31:0] ddpc4,dbusw,ddata2;
    output reg [4:0] drw;
    output reg  nnreg_write,nnmem_write,mmemtoreg;
    output reg [1:0]nns_data_write;

    always @(posedge clock,negedge reset)
    begin
        if(~reset) {ddpc4,dbusw,ddata2,drw,nnreg_write,nnmem_write,nns_data_write,mmemtoreg}={
        32'h0000_0000,32'h0000_0000,32'h0000_0000,5'b00000,1'b0,1'b0,2'b00,1'b0};
        else if(mem_r) {ddpc4,dbusw,ddata2,drw,nnreg_write,nnmem_write,nns_data_write,
        mmemtoreg}={dpc4,busw,data2,rw,nreg_write,nmem_write,ns_data_write,ememtoreg};
    end

endmodule
```

ID_EXE 模块：

```
module id_exe(
    input clock,reset,
    input [31:0]dpc4,data1,data2,ext_imm,
    input [4:0] rw, aluop,
    input s_b,reg_write,mem_write,
    input [1:0] s_data_write,
    input [4:0] rs,rt,
    input  memtoreg,pc_write,pc_write2,flush,
    output  reg [31:0] ddpc4,ddata1,ddata2,dext_imm,
    output  reg [4:0]  drw,naluop,
    output reg ns_b,nreg_write,nmem_write,
    output reg [1:0]ns_data_write,
    output reg [4:0] nrs,[4:0] nrt,
    output reg ememtoreg,mem_flush
);

always@(posedge clock , negedge reset)
 begin
  if((~pc_write)||(~reset)||(~pc_write2)){ddpc4,ddata1,ddata2,dext_imm,drw,naluop,ns_b,
  nreg_write,nmem_write,ns_data_write,nrs,nrt,ememtoreg}={32'd0,32'd0,32'd0,32'd0,5'd0,5'd0,
  1'd0,1'd0,1'd0,2'd0,5'd0,5'd0,1'd0};
  else {ddpc4,ddata1,ddata2,dext_imm,drw,naluop,ns_b,nreg_write,nmem_write,ns_data_write,nrs,
  nrt,ememtoreg,mem_flush}={dpc4,data1,data2,ext_imm,rw,aluop,s_b,reg_write,mem_write,
  s_data_write,rs,rt,memtoreg,flush};
 end

endmodule
```

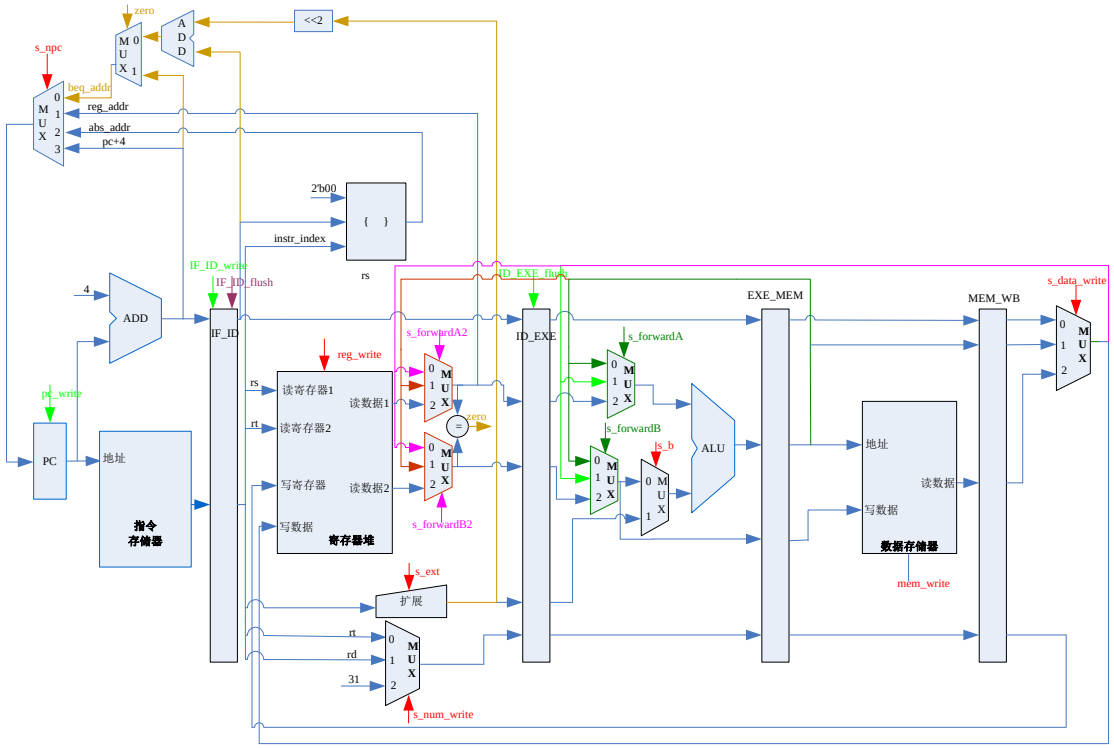
MEM_WB 模块：

```
module  mem_wb(
    input clock,reset,
    input [31:0] dpc4,busw,data,
    input [4:0] rw,
    input nnreg_write,
    input [1:0]nns_data_write,
    input [31:0] mem_data2,
    input mem_r,
    output reg [31:0] ddpc4,dbusw,ddata,
    output reg [4:0] drw,
    output reg nnnreg_write,
    output reg [1:0]nnns_data_write,
    output reg [31:0] wb_data2
    );


    always @(posedge clock,negedge reset)
    begin
        if(~reset) {ddpc4,dbusw,ddata,drw,nnnreg_write,wb_data2,nnns_data_write}={
        32'h0000_0000,32'h0000_0000,32'h0000_0000,5'b00000,1'b0,32'd0,2'b00};
        else if(mem_r) {ddpc4,dbusw,ddata,drw,nnnreg_write,wb_data2,nnns_data_write}={dpc4,
        busw,data,rw,nnreg_write,mem_data2,nns_data_write};
    end

endmodule
```

顶层模块：

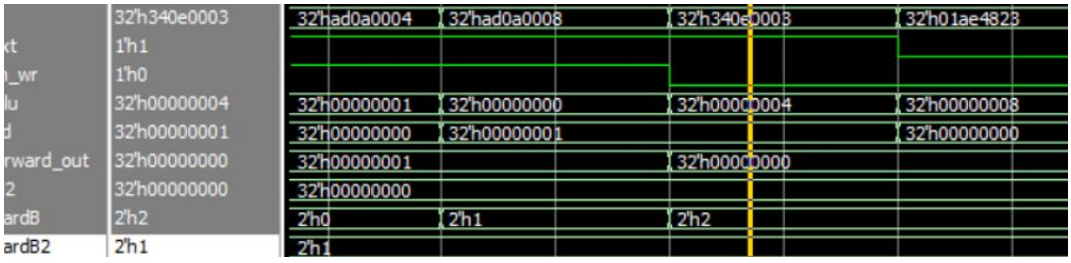
```
    pc PC(pc,clock,reset,pc_write,pc_write_2,npc);//ok
    npc NPC(pc,id_ext_num,Ext_imm32,id_ins[25:0],for_id_reg1Data,npc_op,zeroone,npc);//ok
    im IM(if_ins,pc);//ok
    if_id IF_ID(pc,if_ins,clock,reset,IF_ID_flu,pc_write,pc_write_2,EXE_flu,id_ext_num,id_ins
    );//ok
    ctrl CTRL(id_ins[31:26],id_ins[5:0],aluop,id_reg_write,exSer,id_s_b,s_num_write,
    id_mem_write,s_data_write,npc_op,mem_to_reg);//ok
    ext EXT(id_ins[15:0],exSer,Ext_imm32);//ok
    gpr GPR(for_id_reg1Data,for_id_reg2Data,clock,wb_reg_write,id_ins,wb_rw,data_write); //ok

    id_exe ID_EXE(clock,reset,id_ext_num,for_ex_reg1Data,for_ex_reg2Data,Ext_imm32,num_write,
    aluop,id_s_b,id_reg_write,id_mem_write,s_data_write,rs,rt,mem_to_reg,pc_write,pc_write_2,
    EXE_flu,
    exe_pc,ex_reg1Data,ex_reg2Data,exe_ext_imm,exe_rw,exe_alu_op,exe_s_b,exe_reg_write,
    exe_mem_write,exe_s_data_write,exe_rs,exe_rt,exe_mtor,mem_flu);
    forward CTRL2(wb_rw,mem_rw,exe_rw,exe_rs,exe_rt,rs,rt,for_ex_reg1Data,for_ex_reg2Data,
    exe_reg_write,mem_reg_write,wb_reg_write,exe_mtor,mem_mtor,npc_op,EXE_flu,mem_flu,
    mem_mem_write,
    s_forwardA,s_forwardB,s_forwardA2,s_forwardB2,wst,IF_ID_flu,zeroone,pc_write_2,pc_write);

    alu ALU(ex_alu_res,id_reg_data,alu_result_b,exe_alu_op);//ok
    exe_mem EXE_MEM(clock,reset,exe_pc,ex_alu_res,id_reg_data2,exe_rw,exe_reg_write,
    exe_mem_write,exe_s_data_write,exe_mtor,mem_pc_write,
                mem_pc,mem_alu_res,mem_reg2Data,mem_rw,mem_reg_write,mem_mem_write,
                mem_s_data_write,mem_mtor);
    dm DM(mem_mem_data,clock,mem_mem_write,mem_alu_res,mem_reg2Data);//ok
    mem_wb MEM_WB(clock,reset,mem_pc,mem_alu_res,mem_mem_data,mem_rw,mem_reg_write,
    mem_s_data_write,mem_reg2Data,mem_pc_write,
                wb_pc,wb_alu_result,wb_mem_data,wb_rw,wb_reg_write,wb_s_data_write,wb_data2
                );//ok


    mux3  MUX3(wb_alu_result,wb_mem_data,s_wb_pc,wb_s_data_write,data_write);//ok
    mux4 #(5)  MUX1(rt,rd,32'b1111_1111_1111_1111_1111_1111_1111_1111,5'b00000,s_num_write,
    num_write);//ok
    mux3  MUX4(mem_alu_res,data_write,ex_reg1Data,s_forwardA,id_reg_data);//ok
    mux3  MUX5(mem_alu_res,data_write,ex_reg2Data,s_forwardB,reg_tmpdata);//ok
    mux2  MUX8(reg_tmpdata,data_write,wst,id_reg_data2);//ok
    mux2  MUX2(id_reg_data2,exe_ext_imm,exe_s_b,alu_result_b);//ok
    mux3  MUX6(data_write,mem_alu_res,for_id_reg1Data,s_forwardA2,for_ex_reg1Data);//ok
    mux3  MUX7(data_write,mem_alu_res,for_id_reg2Data,s_forwardB2,for_ex_reg2Data);//ok


endmodule
```

分析：完整的数据通路图如下：



因为 beq 指令的比较操作提前到 ID 级，所以引入了新的 ID 级数据冒险。需要注意的是，ID 级旁路数据选择器从 2 选 1 变为 3 选 1。

以下是 testbench 部分截图以及仿真截图：

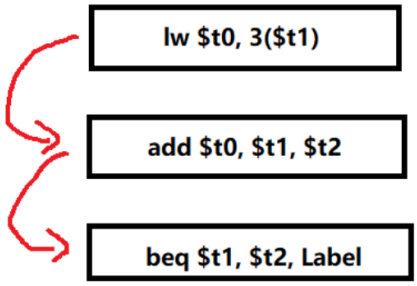


在检测到 beq 并与前一条指令发生数据冒险后，从 ID_EXE 开始阻塞，在第二个时钟周期时用阻塞构建了冒险所需通路条件，通过分析波形图可知成功跳转到正确地址。这一部分只需要利用号仿真波形，让上述的代码实现部分新增的信号传输到正确的位置，并在正确的时间和合理的条件下发挥作用。

3 实验总结

单周期 CPU 和流水线 CPU 的区别：

单周期：



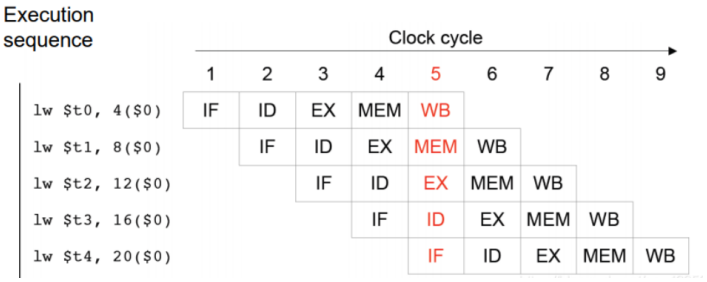
1)CPI = 1

2)以指令为单位，对于每条指令而言，都需要耗费 1 个时钟周期

3)在物理设计上，cycle time 必须统一标准，因此 cycle time 一般取决于耗时最长的指令 lw(lw 需要干的事情最多)，当指令中只有少数 lw 指令时，就会产生时间上的浪费

4)单位指令执行时间 = 1 * T（T 为耗时最长指令的 cycle time）

流水线：



1)在多周期处理器的基础上进一步优化

2)平均 CPI 为 1（当指令数量很大 > 10000 时，最开始 4 个 cycle 的指令填充和 最后 4 个 cycle 的指令排空可以忽略不计，可以近似认为执行 m 条指令消耗 m（ m >> 8 ）个时钟周期，因此平均 CPI = 1）

3)不再以指令为单位。与多周期不同，流水线以每个阶段为单位，一个阶段内可能有多条指令在同时执行（并行）。例如当一条指令在 EX 阶段时，它的上一条指令可能在 MEM 阶段，而下一条指令可能在 ID 阶段，理论上它们是同时执行的

4)cycle time 取决于耗时最长的阶段（一般是 MEM）

5)单位指令执行时间 = 1 *（ T / N ）

6)可能存在三类冒险问题：结构冒险（ Structural Hazard ）、数据冒险（ Data Hazard ）以及控制冒险（ Control Hazard ）

总结：

本次实验的难点在于对流水线 CPU 的整体把握，以及顶层仿真中的纠错工作。流水线 CPU 速度更快，多条指令同时进行，这就要求我们对不同指令的运行和数据的传递有一定的认识。在理论课程中已经对流水线 CPU 有了一定的了解学习，通过实验可以更直观更透彻的学习其具体流程和实现方式。

回顾整个实验过程，最重要的时严谨细致的态度。笔误这种错误最容易避免，

也最容易发生，排查起来却相当的费力。只要再细心一点，实验就能更轻松一嗲你。对于仿真软件的应用也更为熟练，学习到很多新的方便的小功能，对于分析波形很有帮助。

计组实验已经结束，感觉收获良多，通过这两次与流水线 cpu 相关的实验，让我对 cpu 内部数据流通的理解更进一层，从开始的单周期，到后来的数据冒险，在自己发现问题、解决问题的过程中不断地得到了锻炼，在写汇编、看波形、找错误源头的过程中提升了自己的理论功底，我对 verilog 语言的使用也愈加娴熟，代码中的逻辑错误大大减少，为今后的学习打下了良好的基础。