《计算机组成原理实验》

实验报告

学 院 名 称 ：	计算机学院
专业（班级）：	计算机科学与技术(超算方向)
学 生 姓 名 ：	林天皓
学　　号 ：	18324034
时　　间 ：	2020 年 1 月 4 日

成 绩 ：

实　验 ： 流水线CPU设计与实现

一. 实验目的

1. 了解流水线CPU基本功能部件的设计与实现方法，
2. 了解提高CPU性能的方法。
3. 掌握流水线MIPS微处理器的工作原理。
4. 理解数据冒险、控制冒险的概念以及流水线冲突的解决方法。
5. 掌握流水线MIPS微处理器的测试方法。

二. 实验内容

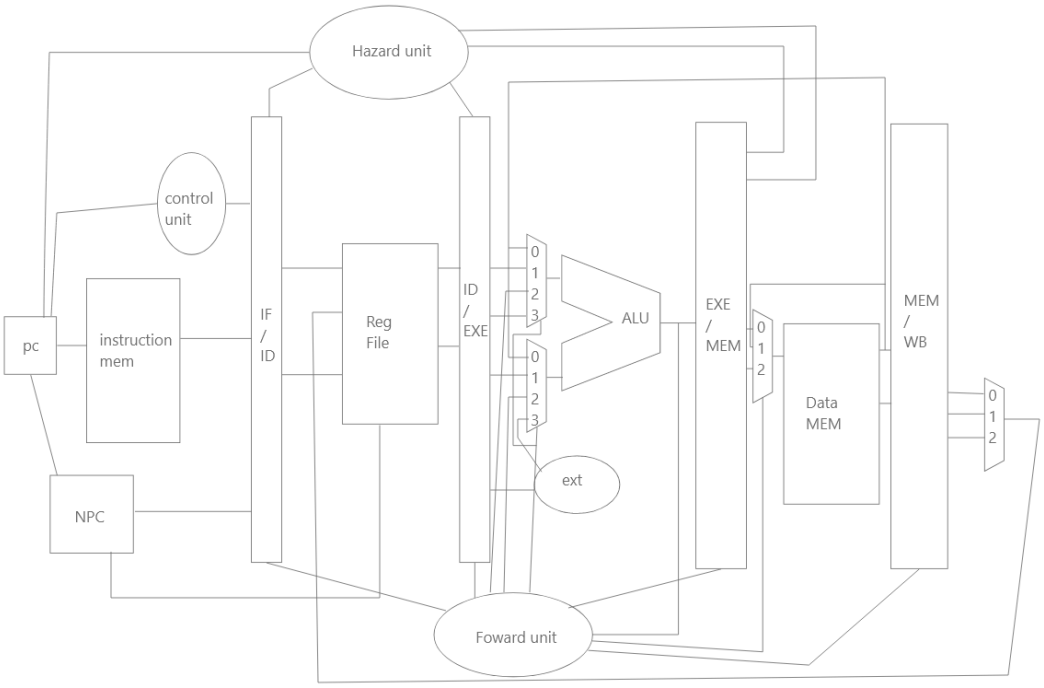
至少支持 add、sub、and、or、addi、andi、ori、lw、sw、beq、bne 十一条指令。采用 5 级流水线技术（可选:具有数据前推机制）,寄存器堆的写操作提前半个时钟周期，下降沿读取，上升沿写入数据。

三. 实验原理

流水线是数字系统中一种提高系统稳定性和工作速度的方法，广泛应用于现代CPU的架构中。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。

流水线CPU的重点在于对冒险的处理，如果没有处理冒险情况，在运行过程中会出现运算错误的情形，因此，需要将冒险检测单元与转发模块加入CPU中，使得流水线CPU能正确运算结果。

本次实现的流水线CPU的整体结构图如下，



流水线CPU数据通路图

所实现的指令如下:

MIPS 的 31 种指令

助记符	指令格式						示 例	示例含义	操作及解释
BIT #	31..26	25..21	20..16	15..11	10..6	5..0			
R-类型	op	rs	rt	rd	shamt	func			
add	000000	rs	rt	rd	00000	100000	add $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1
addu	000000	rs	rt	rd	00000	100001	addu $1,$2,$3	$1=$2+S3	(rd)←(rs)+(rt); rs=$2,rt=$3,rd=$1,无符号数
sub	000000	rs	rt	rd	00000	100010	sub $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1
subu	000000	rs	rt	rd	00000	100011	subu $1,$2,$3	$1=$2-S3	(rd)←(rs)-(rt); rs=$2,rt=$3,rd=$1,无符号数
and	000000	rs	rt	rd	00000	100100	and $1,$2,$3	$1=$2&S3	(rd)←(rs)&(rt); rs=$2,rt=$3,rd=$1
or	000000	rs	rt	rd	00000	100101	or $1,$2,$3	$1=$2|S3	(rd)←(rs) | (rt); rs=$2,rt=$3,rd=$1
xor	000000	rs	rt	rd	00000	100110	xor $1,$2,$3	$1=$2^S3	(rd)←(rs)^(rt); rs=$2,rt=$3,rd=$1
nor	000000	rs	rt	rd	00000	100111	nor $1,$2,$3	$1= ~($2 | S3)	(rd)←~((rs) | (rt)); rs=$2,rt=$3,rd=$1
slt	000000	rs	rt	rd	00000	101010	slt $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1
sltu	000000	rs	rt	rd	00000	101011	sltu $1,$2,$3	if($2<$3) $1=1 else $1=0	if (rs< rt) rd=1 else rd=0;rs＝$2，rt=$3, rd=$1, 无符号数
sll	000000	00000	rt	rd	shamt	000000	sll $1,$2,10	$1=$2<<10	(rd)←(rt)<<shamt,rt=$2,rd=$1,shamt=10
srl	000000	00000	rt	rd	shamt	000010	srl $1,$2,10	$1=$2>>10	(rd)←(rt)>>shamt, rt=$2, rd=$1, shamt=10, (逻辑右移)
sra	000000	00000	rt	rd	shamt	000011	sra $1,$2,10	$1=$2>>10	(rd)←(rt)>>shamt, rt=$2, rd=$1, shamt=10, (算术右移，注意符号位保留)

sllv	000000	rs	rt	rd	00000	000100	sllv $1,$2,$3	$1=$2<<$3	(rd)←(rt)<<(rs), rs=$3,rt=$2,rd=$1
srlv	000000	rs	rt	rd	00000	000110	srlv $1,$2,$3	$1=$2>>$3	(rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (逻辑右移)
srav	000000	rs	rt	rd	00000	000111	srav $1,$2,$3	$1=$2>>$3	(rd)←(rt)>>(rs), rs=$3,rt=$2,rd=$1, (算术右移，注意符号位保留)
jr	000000	rs	00000	00000	00000	001000	jr $31	goto $31	(PC)←(rs)
I-类型	op	rs	rt	immediate					
addi	001000	rs	rt	immediate			addi $1,$2,10	$1=$2+10	(rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2
addiu	001001	rs	rt	immediate			addiu $1,$2,10	$1=$2+10	(rt)←(rs)+(sign-extend)immediate,rt=$1,rs=$2
andi	001100	rs	rt	immediate			andi $1,$2,10	$1=$2&10	(rt)←(rs)&(zero-extend)immediate,rt=$1,rs=$2
ori	001101	rs	rt	immediate			ori $1,$2,10	$1=$2|10	(rt)←(rs)|(zero-extend)immediate,rt=$1,rs=$2
xori	001110	rs	rt	immediate			xori $1,$2,10	$1=$2^10	(rt)←(rs)^(zero-extend)immediate,rt=$1,rs=$2
lui	001111	00000	rt	immediate			lui $1,10	$1=10*65536	(rt)←immediate<<16 & 0FFFF0000H，将 16 位立即数放到目的寄存器高 16 位，目的寄存器的低 16 位填 0
lw	100011	rs	rt	offset			lw $1,10($2)	$1=Memory[$2+10]	(rt)←Memory[(rs)+(sign_extend)offset], rt=$1,rs=$2
sw	101011	rs	rt	offset			sw $1,10($2)	Memory[$2+10] =$1	Memory[(rs)+(sign_extend)offset]←(rt), rt=$1,rs=$2
beq	000100	rs	rt	offset			beq $1,$2,40	if($1=$2) goto PC+4+40	if ((rt)=(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2), rs=$1, rt=$2
bne	000101	rs	rt	offset			bne $1,$2,40	if($1≠$2) goto PC+4+40	if ((rt)≠(rs)) then (PC)←(PC)+4+((Sign-Extend) offset<<2) , rs=$1, rt=$2
slti	001010	rs	rt	immediate			slti $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Sign-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
sltiu	001011	rs	rt	immediate			sltiu $1,$2,10	if($2<10) $1=1 else $1=0	if ((rs)<(Zero-Extend)immediate) then (rt)←1; else (rt)←0, rs=$2, rt=$1
J-类型	op	address							
j	000010	address					j 10000	goto 10000	(PC)←((Zero-Extend) address<<2), address=10000/4
jal	000011	address					jal 10000	$31=PC+4 goto 10000	($31)←(PC)+4; (PC)←((Zero-Extend) address<<2), address=10000/4

除了上述基本指令之外，本次实验还实现了其他的扩展指令(8 条)

I-类型	op	rs	rt	immediate			
Bgez	000100	Rs	00001	offset	bgez $s, offset	if($s>=0)pc+=4+(offset << 2)	if $s == $t advance_pc (offset << 2)); else advance_pc (4);
bgtz	000111	rs	00000	offset	bgtz $s, offset	if($s>0)pc+=4+(offset << 2)	if $s > 0 advance_pc (offset << 2)); else advance_pc (4);
blez	000110	rs	00000	offset	blez $s, offset	if($s<=0)pc+=4+(offset << 2)	if $s <= 0 advance_pc (offset << 2)); else advance_pc (4);

bltz	000001	rs	00000	offset	bltz $s, offset	if($s<0)pc+=4+(offset << 2)	if $s < 0 advance_pc (offset << 2)); else advance_pc (4);
sh	101001	rs	rt	offset	sh $t,offset($s)	mem[$s+8]=$t	MEM[$s + offset] = $t; advance_pc (4);
sb	101000	rs	rt	offset	sb $t,offset($s)	mem[$s+8]=$t	MEM[$s + offset] = $t; advance_pc (4);
lh	100001	rs	rt	offset	lh $t,offset($s)	$t =mem[$s+8]	$t = MEM[$s + offset]; advance_pc (4);
lb	100000	rs	rt	offset	lb $t,offset($s)	$t =mem[$s+8]	$t = MEM[$s + offset]; advance_pc (4);

本次实验中，实现了一个能执行39条指令的，具有冒险检测与转发功能的流水线CPU。

四. 实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五. 实验过程与结果

在本次实验中，对原本实现的单周期CPU进行了非常大的更改，同时在流水线CPU的转发模块与竞争检测模块上通过梳理各种指令之间的依赖关系才能完成关于冒险和竞争的检测，从而合理转发数据。下面我们先从竞争检测和转发模块开始分析。

一、 冒险检测处理模块

冒险检测模块通过实现了根据指令判断是否发生了冒险现象。

可以分为两种情况

1.对于lw指令进行到EXEMEM阶段时候，如果后面紧跟的指令有数据相关，此时不能通过转发来解决这一个冲突，必须要产生一个stall信号，使得pc，IFID阶段，ID-EXE阶段暂停一个时钟周期，才能通过后面的转发模块将正确的数据转发进入alu中进行运算。

2.对于branch，jump指令，我们通过优化判断是否跳转的位置，在ID-EXE阶段就可以判断是否发生了跳转情况，即下文总共npc不等于pc+4的情况，此时我们需要将已经进入IFID阶段的下一条指令进行flush，以防止执行了错误的指令。

完整实现代码如下。

```
module Hazard ( pc, npc, EXEMEM_DMRd, EXEMEM_rd, rs,
                //output
                IFID_stall, IFID_flush, IDEXE_stall, PC_stall, jump );
```

```
    input   [31:0]  pc;
    input   [31:0]  npc;
    input   [3:0]   EXEMEM_DMRd;
    input   [4:0]   EXEMEM_rd;
    input   [4:0]   rs;

    output reg      IFID_stall;
    output reg      IFID_flush;
    output reg      IDEXE_stall;
    output reg      PC_stall;
    output reg      jump;


initial
begin
    IFID_stall  <= 1'b0;
    IFID_flush  <= 1'b0;
    IDEXE_stall <= 1'b0;
    PC_stall    <= 1'b0;
    jump        <= 1'b0;
end

always @(*)
    begin
        //lw指令 和 其他指令的冲突 暂停一个周期
        if( EXEMEM_DMRd != `DMRd_NOP && (EXEMEM_rd == rs) )
            begin
                PC_stall    <= 1'b1;
                IFID_stall  <= 1'b1;
                IDEXE_stall <= 1'b1;
            end
        //直接通过判断npc和pc的对比来判断是否需要进行跳转指令，因为前文已经统一了跳转指令在ID-EXE阶段
        //判断，这里只需要统一flush一条指令。
        else if( npc != pc + 4 )
                    begin
                        PC_stall    <= 1'b0;
                        jump        <= 1'b1;
                        IFID_stall  <= 1'b0;
                        IFID_flush  <= 1'b1;
                    end

        else
            begin
                IFID_stall  <= 1'b0;
```

```
                IFID_flush  <= 1'b0;
                IDEXE_stall <= 1'b0;
                PC_stall    <= 1'b0;
                jump        <= 1'b0;
            end
    end
endmodule
```

二、转发处理模块

转发处理模块通过实现了根据控制模块生成的信号来选择转发信号，达到流水线征程运行的目的。

在本次实验中，一共实现了5种转发的情况。

1.对于jr指令的转发，如果jr指令和上一条指令有数据依赖，则不能直接通过寄存器堆的读取来获取错误的值，而是直接从ALU的运算结果来获取已经计算完毕但是尚未写入寄存器堆得到的值。

2.对于alu获取操作数的转发

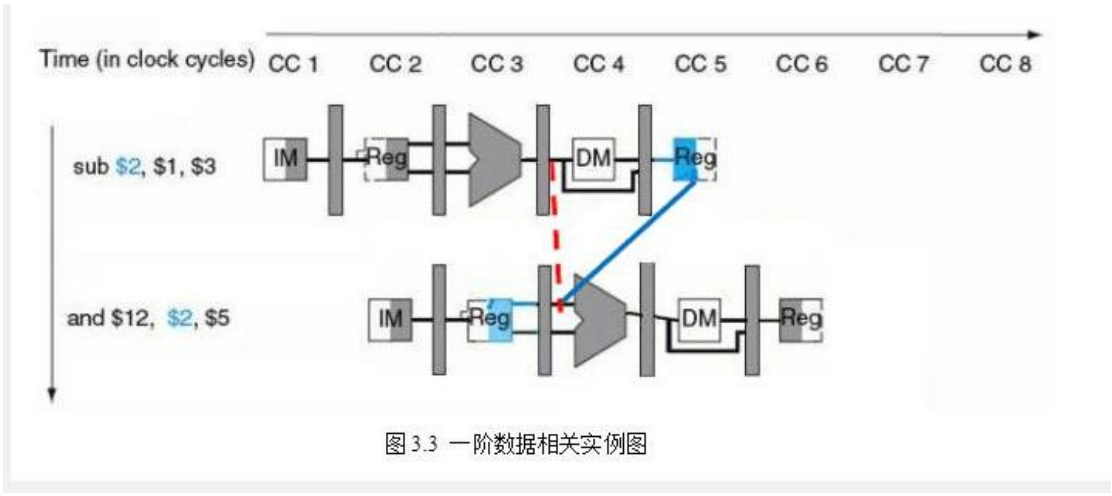


图 3.3 一阶数据相关实例图

图-一阶数据依赖

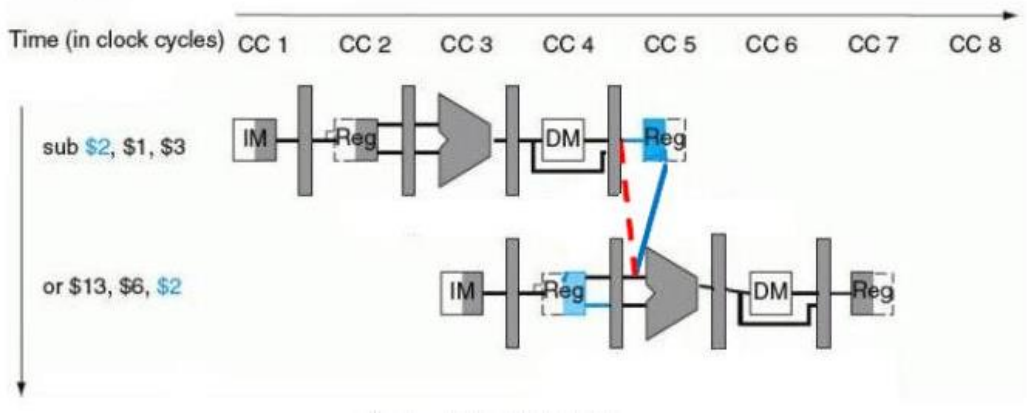


图-二阶数据依赖

根据分析，ALU的操作数的来源有1.寄存器读取值2.指令中的立即数3.由于数据依赖需要从上一个指令的alu运算结果来提前获取值（一阶数据依赖）4.从MEMWB阶段储存的需要写回寄存器堆的值（二阶数据依赖）

3.sw指令数据与lw指令的一阶数据依赖

如果一个sw指令后方紧跟着一个lw指令，如果不进行转发，则lw会获取错误的数据，此时我们需要在lw的时候直接获取sw指令需要存入数据内存中的值，而不是从数据内存中再次读取。（一阶数据依赖）

4.sw指令与前面普通指令的二阶数据依赖

这时候sw指令需要提前获取来转发自alu运算结果的数据，而不是从寄存器堆中获取错误的数据。

5. sw指令与前面lw的二阶数据依赖

这时候sw指令需要储存的值需要提前获取来转发来自在数据内存中读取获得的值，而不是从寄存器堆中获取错误的数据。

实现代码如下

```
module ForwardingUnit( EXEMEM_RFWr, MEMWB_RFWr, MEMWB_DMWr, EXEME
M_DMWr, EXEMEM_DMRd,
                       EXEMEM_rd,   MEMWB_rd,   IFID_DMWr,  MEMWB
_DMRd,
                       IFID_rs,     IFID_rt,    IDEXE_rd,   IDEXE
_RFWr,  IDEXE_DMWr,
                       IDEXE_rs,    IDEXE_rt,   ALU_A,      ALU_B
,       DMdata_ctrl,
                       NPC_F1,      NPC_F2,     SW_ctrl,
```

```
                        ALUSrc1,     ALUSrc2);

    input        EXEMEM_RFWr;
    input        MEMWB_RFWr;
    input        IDEXE_RFWr;
    input [1:0]  IFID_DMWr;
    input [1:0]  IDEXE_DMWr;
    input [1:0]  MEMWB_DMWr;
    input [3:0]  MEMWB_DMRd;
    input [3:0]  EXEMEM_DMRd;
    input [1:0]  EXEMEM_DMWr;
    input        ALUSrc1;
    input        ALUSrc2;
    input [4:0]  EXEMEM_rd;
    input [4:0]  MEMWB_rd;
    input [4:0]  IFID_rs;
    input [4:0]  IFID_rt;
    input [4:0]  IDEXE_rs;
    input [4:0]  IDEXE_rt;
    input [4:0]  IDEXE_rd;

    output reg [1:0] ALU_A;
    output reg [1:0] ALU_B;
    output reg       NPC_F1;
    output reg       NPC_F2;
    output reg [1:0] SW_ctrl;
    output reg       DMdata_ctrl;

initial
    begin
        ALU_A       <= 2'b00;//no foward
        ALU_B       <= 2'b00;//no foward
        NPC_F1      <= 1'b0;
        NPC_F2      <= 1'b0;
        DMdata_ctrl <= 1'b0;
        SW_ctrl     <= 2'b0;
    end

always @(*)
    begin
        ALU_A       <= 2'b00;//no foward
        ALU_B       <= 2'b00;//no foward
        NPC_F1      <= 1'b0;
        NPC_F2      <= 1'b0;
```

```
        DMdata_ctrl <= 1'b0;
        SW_ctrl     <= 2'b0;

        //pc 是否跳转的判断全部放在 ID-EXE 阶段就完成了，这个时候只需要
flush 一条指令
        if(IDEXE_RFWr && (IDEXE_rd != 0) && (IDEXE_rd != 31))
            begin
                if( (IFID_rs == IDEXE_rd) ) NPC_F1 <= 1'b1;
                if( (IFID_rt == IDEXE_rd) ) NPC_F2 <= 1'b1;
            end

        if(EXEMEM_RFWr && (EXEMEM_rd != 0) && (EXEMEM_rd != 31))
            begin
                if( (IDEXE_rs == EXEMEM_rd) ) ALU_A <=  2'b01;//f
oward from exe;
                if( (IDEXE_rt == EXEMEM_rd) && ALUSrc2 == 1'b0 &&
 (IDEXE_DMWr == 3'b000) ) ALU_B <=  2'b01;//foward from exe;
            end


        if(MEMWB_RFWr && (MEMWB_rd != 0) && (MEMWB_rd != 31))
            begin
                if( !(EXEMEM_RFWr && (EXEMEM_rd != 0) && (EXEMEM_
rd != 31) && (EXEMEM_rd == IDEXE_rs)) && MEMWB_DMRd == 3'b000
                    && (MEMWB_rd == IDEXE_rs) )
                    ALU_A <=  2'b10;//foward from mem

                if( !(EXEMEM_RFWr && (EXEMEM_rd != 0) && (EXEMEM_
rd != 31) && (EXEMEM_rd == IDEXE_rt)) && MEMWB_DMRd == 3'b000
                    && (MEMWB_rd == IDEXE_rt) && (IDEXE_DMWr == 3
'b000) )
                    ALU_B <=  2'b10;//foward from mem

            end

        //sw 指令的上一条没有写回 reg，需要转发
        if( EXEMEM_DMWr != 3'b000
            && (MEMWB_rd == EXEMEM_rd) && (MEMWB_rd != 0) && (MEM
WB_rd != 31) && MEMWB_DMWr == 3'b000)
            begin
            DMdata_ctrl <= 1'b1; //MEM2MEM
            end
```

```
        //分别对应 将 alu 运算结果转发成为写入内存的数据和将刚从内存中读
取的数据作为写入内存的数据
        if( IFID_DMWr != 3'b000 && IFID_rt == EXEMEM_rd && (EXEME
M_rd != 0) && (EXEMEM_rd != 31)
            && EXEMEM_DMWr == 3'b000 )
                if( EXEMEM_DMRd == 3'b000 )
                    SW_ctrl <= 2'b01; //other, xxx, SW
                else
                    SW_ctrl <= 2'b10; //LW, xxx, SW
    end
endmodule
```

三、NPC模块

该模块负责处理pc的下一次值，根据指令，npc的状态可以分为以下几种

1.正常运行状态，NPC=PC+4

2.无条件跳转状态，NPC=跳转值

3.条件跳转状态，NPC=PC+偏移量

4.JR指令，NPC=寄存器读取值

在这个模块的处理中，相比单周期也做了较大的改动，在单周期CPU的实现过程中，我们将条件跳转判断是否跳转的状态放在exe阶段之后，而在流水线CPU中，为了统一跳转的时机在ID-EXE阶段，我们将条件跳转的条件判断放在该阶段中，如果发生跳转则只需要flush一条指令，提高了流水线CPU的IPC性能。

具体实现代码如下：

```
module NPC( clk, pc, NPCOp, IMM, Rt, RD1, RD2, npc );
   input          clk;
   input  [31:0] pc;
   input  [2:0]  NPCOp;
   input  [25:0] IMM;       // imme
   input  [4:0]  Rt;        // bltz, bgez
   input  [31:0] RD1;       // jr
   input  [31:0] RD2;
   output reg [31:0] npc;
   wire [31:0] PCPLUS4;
   assign PCPLUS4 = pc + 4; // pc + 4
  initial
    begin
      npc <= 32'h0000_0004;
    end
```

```
   always @(negedge clk) begin
      case (NPCOp)
          3'b000:  npc = PCPLUS4;
          3'b001:    npc = (RD1 == RD2) ? (PCPLUS4 + {{14{IMM[15]
}}, IMM[15:0], 2'b00}) : (PCPLUS4);
          3'b010:    npc = (RD1 == RD2) ? (PCPLUS4) : (PCPLUS4 +
{{14{IMM[15]}}, IMM[15:0], 2'b00});
          3'b111:   npc = {PCPLUS4[31:28], IMM[25:0], 2'b00};

          3'b011:    npc = RD1;
          3'b100:   npc = (RD1 ==  0 || RD1[31] == 1) ? (PCPLUS4
+ {{14{IMM[15]}}, IMM[15:0], 2'b00}) : (PCPLUS4);
          3'b101:   npc = (RD1 !=  0 && RD1[31] == 0) ? (PCPLUS4
+ {{14{IMM[15]}}, IMM[15:0], 2'b00}) : (PCPLUS4);
          3'b110: begin
                case(Rt)
                    5'b00000: npc = (RD1 != 0 && RD1[31] == 1) ?
(PCPLUS4 + {{14{IMM[15]}}, IMM[15:0], 2'b00}) : (PCPLUS4);
                    5'b00001: npc = (RD1 == 0 || RD1[31] == 0) ?
(PCPLUS4 + {{14{IMM[15]}}, IMM[15:0], 2'b00}) : (PCPLUS4);
                endcase
          end
          default:     npc = PCPLUS4;
      endcase
   end

endmodule
```

四、PC模块

Pc模块负责产生pc传入IFID模块

与单周期CPU中的PC模块有一点微小的不同，在于流水线CPU中，有可能遇到指令暂停的情况，在该模块中，通过一个PC_STALL信号来控制PC的流向。

模块实现如下

```
module PC( clk, rst, PC_stall, jump, npc, pc );

  input             clk;
  input             rst;
  input             jump;
  input             PC_stall;
  input      [31:0] npc;
```

```
  output reg  [31:0] pc;

  initial
    begin
      pc <= 32'h0000_0000;
    end

  always @(posedge clk, posedge rst)
  begin
    if (rst)
      pc <= 32'h0000_0000;
    else
      begin
        if(!PC_stall)
          if(jump)
              pc <= npc;
          else
              pc <= pc + 4;
        else
          pc <= pc;
      end
   end

endmodule
```

五、数据内存模块

处理lw，sw,lh,sh等指令，根据DMRD的控制信号来选择写入的方式，实现代码如下。

```
module DM( clk, DMWr, DMRd, DMaddr, DMdata, DMout );
    input                clk;
    input       [1:0]    DMWr;
    input       [3:0]    DMRd;
    input       [31:0]   DMaddr;
    input       [31:0]   DMdata;
    output reg  [31:0]   DMout;

    reg[7:0] RAM [31:0];

  integer i;
    initial
    begin
        for (i = 0; i <= 31; i = i + 1)
            RAM[i] <= 0;
    DMout <= 32'b0;
    end
```

```
always@(*) begin
    if( DMRd != 3'b000 ) begin
        case(DMRd)
            3'b001:
              DMout <= { RAM[DMaddr+3], RAM[DMaddr+2], RAM[DMaddr+1], R
AM[DMaddr] };//LW
            3'b010:
               DMout <= { {16{RAM[DMaddr+1][7]}} , RAM[DMaddr+1], RAM[D
Maddr] };//LH
            3'b011:
               DMout <= { {24{RAM[DMaddr][7]}}  , RAM[DMaddr] };//LB
        endcase
    end
end

always@(posedge clk) begin
    if( DMWr != 3'b000 ) begin
        case(DMWr)
            3'b001:
                { RAM[DMaddr+3], RAM[DMaddr+2], RAM[DMaddr+1], RAM[DMad
dr] } <= DMdata[31:0];
            3'b010:
               { RAM[DMaddr+1], RAM[DMaddr] } <= DMdata[15:0];
            3'b011:
               { RAM[DMaddr] } <= DMdata[7:0];
        endcase
    end
end
```

六、指令内存模块

通过读取文件的方式写入CPU将要执行的指令。

```
`include "ctrl_encode_def.v"
module IM( pc, ins );
    input    [9:0]    pc;
    output   [31:0]   ins;

    reg [7:0]   imem [1023:0];
    reg [31:0]  filedata     [254:0];
    reg [31:0]  temp;

    integer  i, addr ;
```

```
    initial begin
      addr = 0;
      $readmemh("D:\\programme\\Computer-Organization-Experiment\\ex14p
ipelineCPU\\test.txt", filedata);
      for (i = 0; i < 254; i= i + 1) begin
        temp = filedata[i];
        imem[addr]   = temp[7:0];
        imem[addr+1] = temp[15:8];
        imem[addr+2] = temp[23:16];
        imem[addr+3] = temp[31:24];
        addr = addr + 4;
      end
    end

    assign  ins = { imem[pc+3], imem[pc+2], imem[pc+1], imem[pc] };

endmodule
```

其中的文件内容如下

```
24010008
34020002
00411820
00622822
00a22024
00824025
00084040
1501fffe
28460004
28c70000
24e70008
10e1fffe
ac220004
8c290004
240afffe
254a0001
0540fffe
304b0002
08000014
00824025
00000000
3c0a000a
0c000017
200c0002
1980fffd
```

00cc6804
200e0074
01c00008
00000000
000c6043
1d80fffe
200fffff
05e1fffc
a22f0000
a62f0002
82300000
86300002
01e0802a
2df0fffb
01e0802b
20080008
00084042
02084006
010d4821
010d4823
010d4827
010d4826
01a84807
390dffff
fc000000

七、寄存器堆模块

实现CPU中的寄存器读写，注意判断写入寄存器的地址是否为零，如果为零，则读取数据始终为0

实现代码如下：

```
module RF(   input    clk,
             input    rst,
             input         RFWr,
             input  [4:0]  A1, A2, A3,
             input  [31:0] WD,
             output [31:0] RD1, RD2
             );

reg [31:0] regfile[31:0];
integer i;
always @(negedge clk, posedge rst)
  if (rst) begin
    for (i=0; i<32; i=i+1)
```

```
      regfile[i] <= 0;
  end
  else
    if (RFWr) begin
      regfile[A3] <= WD;
    end
assign RD1 = (A1 != 0) ? regfile[A1] : 0;
assign RD2 = (A2 != 0) ? regfile[A2] : 0;

endmodule
```

八、控制模块

负责读取指令，同时产生各种控制信号，传递给ID-EXE阶段。

完整实现代码较长，打包在文件中，此处仅仅展示前50行。

```
module ctrl_unit( opcode, func,
                  RegDst, NPCOp, DMRd, toReg, ALUOp, DMWr, ALUSrc1, ALU
Src2, RFWr, EXTOp );

   input      [5:0] opcode;
   input      [5:0] func;

   output reg ALUSrc1;             // reg shamt imm
   output reg ALUSrc2;
   output reg RFWr;

   output reg [1:0] RegDst;    //Rt Rd R31
   output reg [2:0] NPCOp;
   output reg [3:0] DMRd;      //lw lh lb
   output reg [1:0] toReg;     //PC2Reg Mem2Reg
   output reg [3:0] ALUOp;
   output reg [1:0] DMWr;      //sw sh sb
   output reg [1:0] EXTOp;     //zero sign


   always @(*) begin
      NPCOp = 3'b000;
      RegDst = 2'b01;
      ALUSrc1 = 1'b0;
      ALUSrc2 = 1'b0;
      toReg = 2'b00;
      RFWr = 1'b0;
      DMRd = 3'b000;
      DMWr = 3'b000;
      ALUOp = 4'b0000;
```

```
    EXTOp = 2'b00;

    case (opcode)
        `R_type: begin
            NPCOp = 3'b000;
            RegDst = 2'b01;
            toReg = 2'b00;
            RFWr = 1'b1;
            DMRd = 3'b000;
            DMWr = 3'b000;

            case (func)

                6'b100000: begin
                    ALUSrc1 = 1'b0;
                    ALUSrc2 = 1'b0;
                    ALUOp   = 4'b0001;//ADD
                end

                6'b100010: begin
                    ALUSrc1 = 1'b0;
                    ALUSrc2 = 1'b0;
                    ALUOp   = 4'b0010;//SUB
                end

                6'b100100: begin
                    ALUSrc1 = 1'b0;
                    ALUSrc2 = 1'b0;
                    ALUOp   = 4'b0011;//AND
                end
```

九、扩展模块

负责实现立即数扩展，与单周期CPU不同的是，这里把LUI指令对立即数的处理没有放在alu中，而是放在了扩展模块中。

```
module EXT( Imm16, EXTOp, Imm32 );
   input       [1:0]   EXTOp;
   input       [15:0]  Imm16;
   output reg  [31:0]  Imm32;
   always @(*) begin
      case (EXTOp)
         2'b00:
            Imm32 = {{16{Imm16[15]}}, Imm16};
```

```
        2'b01:
            Imm32 = {16'd0, Imm16};
        2'b10:
            Imm32 = {Imm16, 16'd0};//LUI
        default:
            Imm32 = {{16{Imm16[15]}}, Imm16};
      endcase
   end
endmodule
```

十、IF-ID模块

负责从pc模块获取当前的pc值，传递PC给控制模块，同时考虑是否有stall信号需要暂停pc单元的改变，同时在遇到flush信号的时候需要清除自身流水线中的信号。

实现代码如下：

```
module IF_ID ( clk, rst, IFID_stall, IFID_flush,
               pc, ins,
               IFID_pc, IFID_ins );
  input              clk;
  input              rst;
  input              IFID_stall;
  input              IFID_flush;
  input       [31:0] pc;
  input       [31:0] ins;
  output reg  [31:0] IFID_pc;
  output reg  [31:0] IFID_ins;
    initial
    begin
        IFID_pc  <= 32'b0;
        IFID_ins <= 32'b0;
    end
    always @(posedge clk) begin
        if (rst) begin
            IFID_pc  <= 32'b0;
            IFID_ins <= 32'b0;
        end
      else if(!IFID_stall)
        begin
            if(IFID_flush)
                begin
                    IFID_pc  <= 32'b0;
                    IFID_ins <= 32'b0;
```

```
                end
            else
                begin
                    IFID_pc  <= pc;
                    IFID_ins <= ins;
                end
        end
    end
endmodule
```

十一、ID-EXE模块

在这个阶段已经获取了来自控制模块生成的控制信号，将获取到的信号在下一个时钟周期传递给alu执行以及继续下传信号，同时也需要判断是否有读内存的数据依赖导致的stall需要暂停执行。由于所有的分支判断均在exe阶段之前即可完成，因此id-exe阶段已经不需要flush指令了。

具体实现代码如下

```
module ID_EXE(  clk, rst, IDEXE_stall, RD1, RD2, rd,
Imm32,  ins, pc, RegDst, NPCOp, DMRd, toReg, ALUOp, DMWr, ALUSrc1, ALUSr
c2, RFWr,
IDEXE_RD1, IDEXE_RD2, IDEXE_Imm32, IDEXE_rd,IDEXE_ins, IDEXE_pc, IDEXE_
RegDst, IDEXE_NPCOp, IDEXE_DMRd, IDEXE_toReg, IDEXE_ALUOp,  IDEXE_DMWr,
 IDEXE_ALUSrc1, IDEXE_ALUSrc2, IDEXE_RFWr);

    input clk, rst, IDEXE_stall;
    input ALUSrc1;
    input ALUSrc2;
    input RFWr;
    input [31:0] RD1;
    input [31:0] RD2;
    input [4:0]  rd;
    input [31:0] Imm32;
    input [31:0] ins;
    input [31:0] pc;
    input [1:0]  RegDst;
    input [2:0]  NPCOp;
    input [3:0]  DMRd;
    input [1:0]  toReg;
    input [3:0]  ALUOp;
    input [1:0]  DMWr;

    output reg IDEXE_ALUSrc1;
    output reg IDEXE_ALUSrc2;
```

```
    output reg IDEXE_RFWr;
    output reg [31:0] IDEXE_RD1;
    output reg [31:0] IDEXE_RD2;
    output reg [4:0]  IDEXE_rd;
    output reg [31:0] IDEXE_Imm32;
    output reg [31:0] IDEXE_ins;
    output reg [31:0] IDEXE_pc;
    output reg [1:0]  IDEXE_RegDst;
    output reg [2:0]  IDEXE_NPCOp;
    output reg [3:0]  IDEXE_DMRd;
    output reg [1:0]  IDEXE_toReg;
    output reg [3:0]  IDEXE_ALUOp;
    output reg [1:0]  IDEXE_DMWr;
initial
begin
    IDEXE_ALUSrc1 <= 1'b0;
    IDEXE_ALUSrc2 <= 1'b0;
    IDEXE_RFWr    <= 1'b0;
    IDEXE_RD1     <= 32'b0;
    IDEXE_RD2     <= 32'b0;
    IDEXE_rd      <= 5'b0;
    IDEXE_Imm32   <= 32'b0;
    IDEXE_ins     <= 32'b0;
    IDEXE_pc      <= 32'b0;
    IDEXE_RegDst  <= 2'b01;
    IDEXE_NPCOp   <= 3'b000;
    IDEXE_DMRd    <= 3'b000;
    IDEXE_toReg   <= 2'b00;
    IDEXE_ALUOp   <= 4'b0000;
    IDEXE_DMWr    <= 3'b000;
end

always @(posedge clk)
begin
    if(rst)
    begin
        IDEXE_ALUSrc1 <= 1'b0;
        IDEXE_ALUSrc2 <= 1'b0;
        IDEXE_RFWr    <= 1'b0;
        IDEXE_RD1     <= 32'b0;
        IDEXE_RD2     <= 32'b0;
        IDEXE_rd      <= 5'b0;
        IDEXE_Imm32   <= 32'b0;
        IDEXE_ins     <= 32'b0;
```

```
        IDEXE_pc      <= 32'b0;
        IDEXE_RegDst  <= 2'b01;
        IDEXE_NPCOp   <= 3'b000;
        IDEXE_DMRd    <= 3'b000;
        IDEXE_toReg   <= 2'b00;
        IDEXE_ALUOp   <= 4'b0000;
        IDEXE_DMWr    <= 3'b000;
    end
    else if(!IDEXE_stall)
    begin
            IDEXE_ALUSrc1 <= ALUSrc1;
            IDEXE_ALUSrc2 <= ALUSrc2;
            IDEXE_RFWr    <= RFWr;
            IDEXE_RD1     <= RD1;
            IDEXE_RD2     <= RD2;
            IDEXE_rd      <= rd;
            IDEXE_Imm32   <= Imm32;
            IDEXE_ins     <= ins;
            IDEXE_pc      <= pc;
            IDEXE_RegDst  <= RegDst;
            IDEXE_NPCOp   <= NPCOp;
            IDEXE_DMRd    <= DMRd;
            IDEXE_toReg   <= toReg;
            IDEXE_ALUOp   <= ALUOp;
            IDEXE_DMWr    <= DMWr;
    end
end
endmodule
```

十二、EXE-MEM模块

该模块负责接收alu的结果，并且将以前获取到的控制信号继续传递给mem模块和下面的MEM-WB模块，来控制内存的写入，同时在这个阶段，已经不再会有flush和stall的出现了。

实现代码如下

```
module EXE_MEM( clk, rst,
                ins, pc, rd, toReg, DMRd, DMWr, RFWr, DMdata, ALUout,
                //output
                EXEMEM_ins, EXEMEM_pc, EXEMEM_rd, EXEMEM_toReg,
                EXEMEM_DMRd, EXEMEM_DMWr, EXEMEM_RFWr, EXEMEM_DMdata, E
XEMEM_ALUout );

    input  clk, rst;
    input  RFWr;
```

```
    input  [31:0]  ins;
    input  [31:0]  pc;
    input  [31:0]  DMdata;
    input  [31:0]  ALUout;
    input  [4:0]   rd;
    input  [1:0]   toReg;
    input  [3:0]   DMRd;
    input  [1:0]   DMWr;
    output reg EXEMEM_RFWr;
    output reg [4:0]   EXEMEM_rd;
    output reg [31:0]  EXEMEM_ins;
    output reg [31:0]  EXEMEM_pc;
    output reg [1:0]   EXEMEM_DMWr;
    output reg [3:0]   EXEMEM_DMRd;
    output reg [1:0]   EXEMEM_toReg;
    output reg [31:0]  EXEMEM_DMdata;
    output reg [31:0]  EXEMEM_ALUout;
initial
begin
    EXEMEM_RFWr    <= 1'b0;
    EXEMEM_rd      <= 5'b0;
    EXEMEM_ins     <= 32'b0;
    EXEMEM_pc      <= 32'b0;
    EXEMEM_DMWr    <= 3'b000;
    EXEMEM_DMRd    <= 3'b000;
    EXEMEM_toReg   <= 2'b00;
    EXEMEM_DMdata  <= 32'b0;
    EXEMEM_ALUout  <= 32'b0;
end
always @(posedge clk)
begin
    if(rst)
        begin
            EXEMEM_RFWr    <= 1'b0;
            EXEMEM_rd      <= 5'b0;
            EXEMEM_ins     <= 32'b0;
            EXEMEM_pc      <= 32'b0;
            EXEMEM_DMWr    <= 3'b000;
            EXEMEM_DMRd    <= 3'b000;
            EXEMEM_toReg   <= 2'b00;
            EXEMEM_DMdata  <= 32'b0;
            EXEMEM_ALUout  <= 32'b0;
        end
    else
```

```
        begin
            EXEMEM_RFWr    <= RFWr;
            EXEMEM_rd      <= rd;
            EXEMEM_ins     <= ins;
            EXEMEM_pc      <= pc;
            EXEMEM_DMWr    <= DMWr;
            EXEMEM_DMRd    <= DMRd;
            EXEMEM_toReg   <= toReg;
            EXEMEM_DMdata  <= DMdata;
            EXEMEM_ALUout  <= ALUout;
        end
end
endmodule
```

十三、MEM-WB模块

该模块和EXE-MEM模块类似，同样没有flush和stall需要处理，只需要继续将EXE-MEM模块传递过来的信号继续传递，为寄存器堆的写入提供数据和地址。

实现代码如下

```
module MEM_WB ( clk, rst,
                ins, pc, rd, toReg, RFWr, DMRd, DMWr, DMout, ALUout,
                MEMWB_ins,  MEMWB_pc,   MEMWB_rd,   MEMWB_toReg,
                MEMWB_DMRd, MEMWB_DMWr, MEMWB_RFWr, MEMWB_DMout, MEMWB_
ALUout );
    input  clk, rst;
    input  RFWr;
    input  [31:0]  ins;
    input  [31:0]  pc;
    input  [31:0]  DMout;
    input  [31:0]  ALUout;
    input  [4:0]   rd;
    input  [1:0]   toReg;
    input  [1:0]   DMWr;
    input  [3:0]   DMRd;

    output reg MEMWB_RFWr;
    output reg [4:0]   MEMWB_rd;
    output reg [31:0]  MEMWB_ins;
    output reg [31:0]  MEMWB_pc;
    output reg [3:0]   MEMWB_DMWr;
    output reg [3:0]   MEMWB_DMRd;
    output reg [1:0]   MEMWB_toReg;
    output reg [31:0]  MEMWB_DMout;
```

```
    output reg [31:0]  MEMWB_ALUout;
initial
begin
    MEMWB_RFWr    <= 1'b0;
    MEMWB_rd      <= 5'b0;
    MEMWB_ins     <= 32'b0;
    MEMWB_pc      <= 32'b0;
    MEMWB_DMWr    <= 3'b000;
    MEMWB_DMRd    <= 3'b000;
    MEMWB_toReg   <= 2'b00;
    MEMWB_DMout  <= 32'b0;
    MEMWB_ALUout  <= 32'b0;
end
always @(posedge clk)
begin
    if(rst)
        begin
            MEMWB_RFWr    <= 1'b0;
            MEMWB_rd      <= 5'b0;
            MEMWB_ins     <= 32'b0;
            MEMWB_pc      <= 32'b0;
            MEMWB_DMWr    <= 3'b000;
            MEMWB_DMRd    <= 3'b000;
            MEMWB_toReg   <= 2'b00;
            MEMWB_DMout  <= 32'b0;
            MEMWB_ALUout  <= 32'b0;
        end
    else
        begin
            MEMWB_RFWr    <= RFWr;
            MEMWB_rd      <= rd;
            MEMWB_ins     <= ins;
            MEMWB_pc      <= pc;
            MEMWB_DMWr    <= DMWr;
            MEMWB_DMRd    <= DMRd;
            MEMWB_toReg   <= toReg;
            MEMWB_DMout   <= DMout;
            MEMWB_ALUout  <= ALUout;
        end
end
endmodule
```

十四、顶层模块

实现了以上所有模块之后，通过将模块组装形成最终的顶层模块。

实现代码如下：

```
module top( clk, rst );
    input           clk;
    input           rst;
    wire    IFID_stall;
    wire    IFID_flush;
    wire    [31:0]   IFID_ins;
    wire    [31:0]   IFID_pc;
    IF_ID    IF_ID( .clk(clk), .rst(rst), .IFID_stall(IFID_stall), .IFI
D_flush(IFID_flush),
                    .pc(pc),    .ins(ins),

                    .IFID_pc(IFID_pc),    .IFID_ins(IFID_ins));

    wire    [31:0]  npc;
    wire            PC_stall;
    wire    [31:0]  pc;
    wire            jump;
    PC      PC      ( .clk(clk), .rst(rst), .PC_stall(PC_stall), .jump(
jump), .npc(npc), .pc(pc) );
    wire    [31:0] ins;
    IM      IM      ( .pc(pc), .ins(ins) );

    wire    [2:0]   NPCOp;
    NPC     NPC     ( .clk(clk),   .pc(IFID_pc),    .NPCOp(NPCOp),   .I
MM(IFID_ins[25:0]), .Rt(IFID_ins[20:16]),
                      .RD1(NPC_1),   .RD2(NPC_2),       .npc(npc) );
    //NPC forward
    wire    [31:0]  NPC_1;
    wire    [31:0]  NPC_2;
    wire    [31:0]  ALUout;
    wire    [31:0]  EXEMEM_ALUout;
    mux2    NPC1  ( .d0(RD1), .d1(ALUout), .s(NPC_F1), .y(NPC_1) );
    mux2    NPC2  ( .d0(RD2), .d1(ALUout), .s(NPC_F2), .y(NPC_2) );

    wire [31:0]     RD1;
    wire [31:0]     RD2;
    wire            MEMWB_RFWr;
    wire [4:0]      MEMWB_rd;
    wire [31:0]     WD;

    RF      RF      ( .clk(clk), .rst(rst),
                      .RFWr(MEMWB_RFWr), .A1(IFID_ins[25:21]), .A2(IFID
_ins[20:16]), .A3(MEMWB_rd), .WD(WD),
```

```
                    .RD1(RD1), .RD2(RD2) );

    wire    [1:0]   EXTOp;
    wire    [31:0]  Imm32;
    EXT     EXT     ( .Imm16(IFID_ins[15:0]), .EXTOp(EXTOp), .Imm32(Imm
32) );

    wire    [4:0]   rd;
    wire    [1:0]   RegDst;
    mux4    RDmux  ( .d0(IFID_ins[20:16]), .d1(IFID_ins[15:11]), .d2(31
), .d3(0), .s(RegDst), .y(rd) );
    //SW  forward
    wire    [1:0]   SW_ctrl;

    wire    [31:0]  RD2final;
    wire    [31:0]  DMout;
    mux4    sw_foward   ( .d0(RD2),  .d1(EXEMEM_ALUout), .d2(DMout),  .
d3(0), .s(SW_ctrl), .y(RD2final) );


    //IDEXE
    wire    [31:0]  IDEXE_RD2;
    wire    [31:0]  IDEXE_RD1;
    wire    [1:0]   IDEXE_DMWr;
    wire    [31:0]  IDEXE_ins;
    wire    [4:0]   IDEXE_rd;
    wire    [31:0]  IDEXE_Imm32;
    wire            IDEXE_ALUSrc2;
    wire    [3:0]   IDEXE_DMRd;
    wire    [2:0]   IDEXE_NPCOp;
    wire            IDEXE_RFWr;
    wire            IDEXE_ALUSrc1;
    wire    [31:0]  IDEXE_pc;
    wire    [1:0]   IDEXE_RegDst;
    wire    [1:0]   IDEXE_toReg;
    wire    [3:0]   IDEXE_ALUOp;

    //ctrl_unit
    wire    [3:0]   ALUOp;
    wire    [1:0]   DMWr;
    wire    [3:0]   DMRd;
    wire    [1:0]   toReg;
    wire            ALUSrc1;
    wire            ALUSrc2;
```

```
    ctrl_unit ctrl_unit( .opcode(IFID_ins[31:26]), .func(IFID_ins[5:0])
,
                         .RegDst(RegDst),     .NPCOp(NPCOp),    .DMRd(D
MRd), .toReg(toReg),
                         .ALUOp(ALUOp),       .DMWr(DMWr),      .ALUSrc
1(ALUSrc1),
                         .ALUSrc2(ALUSrc2),   .RFWr(RFWr),      .EXTOp(
EXTOp) );

    ID_EXE  ID_EXE( .clk(clk), .rst(rst), .IDEXE_stall(IDEXE_stall),
                    .RD1(RD1), .RD2(RD2final), .rd(rd),
                    .Imm32(Imm32),
                    .ins(IFID_ins), .pc(IFID_pc), .RegDst(RegDst), .NPC
Op(NPCOp), .DMRd(DMRd), .toReg(toReg), .ALUOp(ALUOp),
                    .DMWr(DMWr), .ALUSrc1(ALUSrc1), .ALUSrc2(ALUSrc2),
.RFWr(RFWr),
                    .IDEXE_RD1(IDEXE_RD1), .IDEXE_RD2(IDEXE_RD2), .IDEX
E_Imm32(IDEXE_Imm32),   .IDEXE_rd(IDEXE_rd),
                    .IDEXE_ins(IDEXE_ins), .IDEXE_pc(IDEXE_pc),   .IDEX
E_RegDst(IDEXE_RegDst), .IDEXE_NPCOp(IDEXE_NPCOp), .IDEXE_DMRd(IDEXE_DM
Rd),
                    .IDEXE_toReg(IDEXE_toReg), .IDEXE_ALUOp(IDEXE_ALUOp
), .IDEXE_DMWr(IDEXE_DMWr),
                    .IDEXE_ALUSrc1(IDEXE_ALUSrc1), .IDEXE_ALUSrc2(IDEXE
_ALUSrc2), .IDEXE_RFWr(IDEXE_RFWr) );
    wire    [31:0] ALUSrcAout;
    mux4    ALU_A_f   ( .d0(ALUSrcAout), .d1(EXEMEM_ALUout),.d2(WD), .d
3(0),
                        .s(ALU_A),.y(A) );
    mux2    ALU_A     ( .d0(IDEXE_RD1), .d1({27'b0,IDEXE_ins[10:6]}),
                        .s(IDEXE_ALUSrc1), .y(ALUSrcAout) );
    wire    [1:0]  ALU_A;
    wire    [31:0] A;

    wire    [31:0] ALUSrcBout;
    mux4    ALU_B_f   ( .d0(ALUSrcBout), .d1(EXEMEM_ALUout), .d2(WD), .
d3(0),
                        .s(ALU_B),.y(B) );
    mux2    ALU_B     ( .d0(IDEXE_RD2), .d1(IDEXE_Imm32),
                        .s(IDEXE_ALUSrc2), .y(ALUSrcBout) );
    wire    [1:0]  ALU_B;
    wire    [31:0] B;
    alu      alu    ( .A(A), .B(B), .ALUOp(IDEXE_ALUOp), .C(ALUout), .Z
ero(Zero) );
```

```
    wire            Zero;



    wire    [31:0]  EXEMEM_ins;
    wire    [1:0]   EXEMEM_toReg;
    wire    [31:0]  EXEMEM_DMdata;
    wire    [31:0]  EXEMEM_pc;

    EXE_MEM EXE_MEM( .clk(clk), .rst(rst),
                     .ins(IDEXE_ins), .pc(IDEXE_pc), .rd(IDEXE_rd), .to
Reg(IDEXE_toReg),
                     .DMRd(IDEXE_DMRd), .DMWr(IDEXE_DMWr), .RFWr(IDEXE_
RFWr), .DMdata(IDEXE_RD2), .ALUout(ALUout),
                     .EXEMEM_ins(EXEMEM_ins), .EXEMEM_pc(EXEMEM_pc), .E
XEMEM_rd(EXEMEM_rd), .EXEMEM_toReg(EXEMEM_toReg),
                     .EXEMEM_DMRd(EXEMEM_DMRd), .EXEMEM_DMWr(EXEMEM_DMW
r), .EXEMEM_RFWr(EXEMEM_RFWr),
                     .EXEMEM_DMdata(EXEMEM_DMdata), .EXEMEM_ALUout(EXEM
EM_ALUout) );
    wire    [31:0]  DMdata;

    DM      DM      ( .clk(clk), .DMWr(EXEMEM_DMWr), .DMRd(EXEMEM_DMRd)
,
                      .DMaddr(EXEMEM_ALUout), .DMdata(DMdata), .DMout(D
Mout) );
    mux2    Dd_mux ( .d0(EXEMEM_DMdata), .d1(WD), .s(DMdata_ctrl), .y(D
Mdata) );

    wire    [1:0]   MEMWB_toReg;
    wire    [31:0]  MEMWB_DMout;
    wire    [31:0]  MEMWB_pc;
    wire    [31:0]  MEMWB_ins;
    wire    [31:0]  MEMWB_ALUout;
    mux4 Rdmux( .d0(MEMWB_ALUout), .d1(MEMWB_DMout), .d2(MEMWB_pc + 4),
 .d3(0),
                  .s(MEMWB_toReg),   .y (WD));
    MEM_WB  MEM_WB ( .clk(clk), .rst(rst),
                     .ins(EXEMEM_ins), .pc(EXEMEM_pc), .rd(EXEMEM_rd),
.toReg(EXEMEM_toReg),
                     .RFWr(EXEMEM_RFWr), .DMRd(EXEMEM_DMRd), .DMWr(EXEM
EM_DMWr), .DMout(DMout), .ALUout(EXEMEM_ALUout),
                     .MEMWB_ins(MEMWB_ins), .MEMWB_pc(MEMWB_pc), .MEMWB
_rd(MEMWB_rd), .MEMWB_toReg(MEMWB_toReg),
```

```
                    .MEMWB_DMRd(MEMWB_DMRd), .MEMWB_DMWr(MEMWB_DMWr),
                    .MEMWB_RFWr(MEMWB_RFWr), .MEMWB_DMout(MEMWB_DMout)
, .MEMWB_ALUout(MEMWB_ALUout) );
    //Hazard
    wire    [3:0]   EXEMEM_DMRd;
    wire    [4:0]   EXEMEM_rd;
    Hazard Hazard  ( .pc(IFID_pc), .npc(npc), .EXEMEM_DMRd(EXEMEM_DMRd)
, .EXEMEM_rd(EXEMEM_rd), .rs(IFID_ins[25:21]),
                    .IFID_stall(IFID_stall), .IFID_flush(IFID_flush),
.IDEXE_stall(IDEXE_stall),
                    .PC_stall(PC_stall),     .jump(jump) );
    //Forward
    wire    EXEMEM_RFWr;
    wire    [1:0] EXEMEM_DMWr;
    wire    [3:0] MEMWB_DMRd;
    wire    DMdata_ctrl;
    wire    [1:0] MEMWB_DMWr;

    ForwardingUnit ForwardingUnit( .EXEMEM_RFWr(EXEMEM_RFWr),   .EXEMEM
_rd(EXEMEM_rd),     .IDEXE_rs(IDEXE_ins[25:21]), .IDEXE_rt(IDEXE_ins[20
:16]),
                                   .MEMWB_RFWr(MEMWB_RFWr),     .MEMWB_
rd(MEMWB_rd),       .IDEXE_rd(IDEXE_rd),         .IDEXE_RFWr(IDEXE_RFWr
),
                                   .NPC_F1(NPC_F1),             .NPC_F2
(NPC_F2),           .IFID_rs(IFID_ins[25:21]),   .IFID_rt(IFID_ins[20:1
6]),
                                   .ALU_A(ALU_A),.ALU_B(ALU_B), .DMdata
_ctrl(DMdata_ctrl), .IDEXE_DMWr(IDEXE_DMWr),     .IFID_DMWr(DMWr),
                                   .MEMWB_DMWr(MEMWB_DMWr),     .EXEMEM
_DMWr(EXEMEM_DMWr), .SW_ctrl(SW_ctrl),           .MEMWB_DMRd(MEMWB_DMRd
),
                                   .ALUSrc1(ALUSrc1),           .ALUSrc
2(IDEXE_ALUSrc2),   .EXEMEM_DMRd(EXEMEM_DMRd));
endmodule
```

接下来进行仿真测试

用于仿真测试的指令表如下

关于测试流水线 CPU 的简单方法

（特别说明：本表每个同学都必须建立，检查实验时，必须提供！）

1、测试程序段

地址	汇编程序	指令代码	运行结果

		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码		
0x00000000	addiu $1,$0,8	001001	00000	00001	0000 0000 0000 1000	=	24010008	$1=8
0x00000004	ori $2,$0,2	001101	00000	00010	0000 0000 0000 0010		34020002	$2=2
0x00000008	add $3,$2,$1	000000	00010	00001	00011 00000 100000		00411820	$3=10
0x0000000C	sub $5,$3,$2	000000	00011	00010	00101 00000 100010		00622822	$5=8
0x00000010	and $4,$5,$2	000000	00101	00010	00100 00000 100100		00a22024	$4=0
0x00000014	or $8,$4,$2	000000	00100	00010	01000 00000 100101		00824025	$8=2
0x00000018	sll $8,$8,1	000000	00000	01000	01000 00001 000000		00084040	$8=4,=8
0x0000001C	**bne $8,$1,-2 (≠,转 18)**	000101	01000	00001	1111111111111110		1501fffe	跳转 18,不跳转
0x00000020	slti $6,$2,4	001010	00010	00110	0000000000000100		28460004	$6=1
0x00000024	slti $7,$6,0	001010	00110	00111	0000000000000000		28c70000	$7=0
0x00000028	addiu $7,$7,8	001001	00111	00111	0000000000001000		24e70008	$7=8, $7=16
0x0000002C	**beq $7,$1,-2 (=, 转 28)**	000100	00111	00001	1111111111111110		10e1fffe	跳转 28，不跳转
0x00000030	sw $2,4($1)	101011	00001	00010	0000000000000100		ac220004	M[12:15]=2
0x00000034	lw $9,4($1)	100011	00001	01001	0000000000000100		8c290004	$9=2
0x00000038	addiu $10,$0,-2	001001	00000	01010	1111111111111110		240afffe	$10 = -2
0x0000003C	addiu $10,$10,1	001001	01010	01010	0000000000000001		254a0001	$10 = -1, $10 = 0
0x00000040	**bltz $10,-2(<0,转 3C)**	000001	01010	00000	1111111111111110		0540fffe	转 3c,不转
0x00000044	**andi $11,$2,2**	001100	00010	01011	0000000000000010		304b0002	$11 = 2
0x00000048	**j 0x00000050**	000010	00000	00000	0000000000010100		08000014	跳转 50
0x0000004C	or $8,$4,$2	000000	00100	00010	01000 00000 100101		00824025	跳过
0x00000050	**NOP**	000000	00000	00000	0000000000000000	=	00000000	NO OP

补充测试

地址	汇编程序	指令代码						运行结果
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码		
0x00000054	lui $10,10	001111	00000	01010	0000000000001010		3c0a000a	$10=0x000a0000
0x00000058	jal 0x1b	000011	00000	00000	0000000000010111		0c000017	Pc=5c
0x0000005C	addi $12,$0,2	001000	00000	01100	0000000000000010		200c0002	$12=2
0x00000060	blez $12 -3	000110	01100	00000	1111111111111101		1980fffd	Pc=60
0x00000064	sllv $13,$12,$6	000000	00110	01100	01101 00000 000100		00cc6804	$13=4
0x00000068	addi $14,$0,116	001000	00000	01110	00000 00001 110100		200e0074	$14=0x74
0x0000006C	jr $14	000000	01110	00000	00000 00000 001000		01c00008	Pc=0x74
0x00000070	Nop	000000	00000	00000	0000000000000000		00000000	被跳过
0x00000074	sra $12,$12,1	000000	00000	01100	01100 00001 000011		000c6043	$12=1 $12=0
0x00000078	bgtz $12,-2	000111	01100	00000	1111111111111110		1d80fffe	跳转 74 不跳转
0x0000007C	addi $15,$0,-1	001000	00000	01111	1111111111111111		200fffff	$15=-1
0x00000080	bgez $15,-4	000001	01111	00001	1111111111111100		05e1fffc	不跳转
0x00000084	sb $15,0($s1)	101000	10001	01111	0000000000000000		a22f0000	mem[0]=-1
0x00000088	sh $15,2($s1)	101001	10001	01111	0000000000000010		a62f0002	mem[2:3]=-1
0x0000008C	lb $16,0($s1)	100000	10001	10000	0000000000000000		82300000	$16=-1
0x00000090	lh $16,2($s1)	100001	10001	10000	0000000000000010		86300002	$16=-1
0x00000094	slt $16,$15,$0	000000	01111	00000	10000 00000 101010		01e0802a	$16=1
0x00000098	sltiu $16,$15,-5	001011	01111	10000	1111111111111011		2df0fffb	$16=0
0x0000009C	sltu $16,$15,$0	000000	01111	00000	10000 00000 101011		01e0802b	$16=0
0x000000A0	addi $8,$0,8	001000	00000	01000	00000 00000 001000		20080008	$8=8

0x000000A4	srl $8,$8,1	000000	00000	01000	01000 00001 000010		00084042	$8=4
0x000000A8	srlv $8,$8,$16	000001	01000	10001	00000 00000 000000		02084006	$8=4
0x000000AC	addu $9,$8,$13	000000	01000	01101	01001 00000 100001		010d4821	$9=8
0x000000B0	subu $9,$8,$13	000000	01000	01101	01001 00000 100011		010d4823	$9=0
0x000000B4	nor $9,$8,$13	000000	01000	01101	01001 00000 100111		010d4827	$9= -5
0x000000B8	xor $9,$8,$13	000000	01000	01101	01001 00000 100110		010d4826	$9=-0
0x000000BC	srav $9,$8,$13	000000	01000	01101	01001 00000 000111		01a84807	$9= 0
0x000000C0	xori $13,$8,-1	001110	01000	01101	1111111111111111		390dffff	$13 = -5
0x000000C4	**halt**	111111	00000	00000	0000000000000000	=	FC000000	停机

编写仿真测试文件，实现代码如下。

```
`timescale 1ns/1ns

module top_sim;
reg clk, rst;
initial begin
    clk = 0;
    rst = 1;
    #1000 rst = 0;
end

    always #100 clk = ~clk;
    top top1 ( .clk(clk), .rst(rst) );

endmodule
```

接下来进行仿真测试

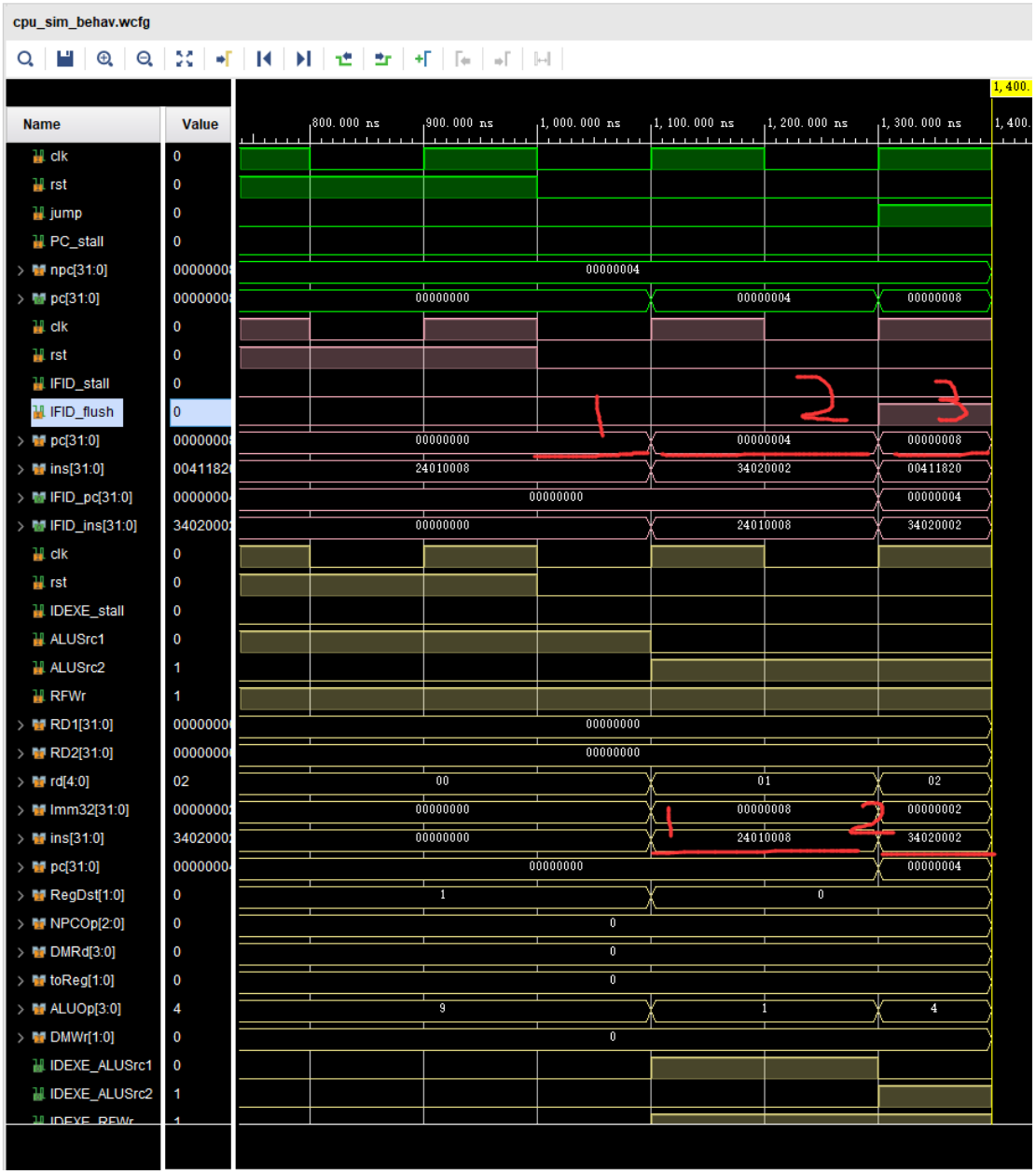
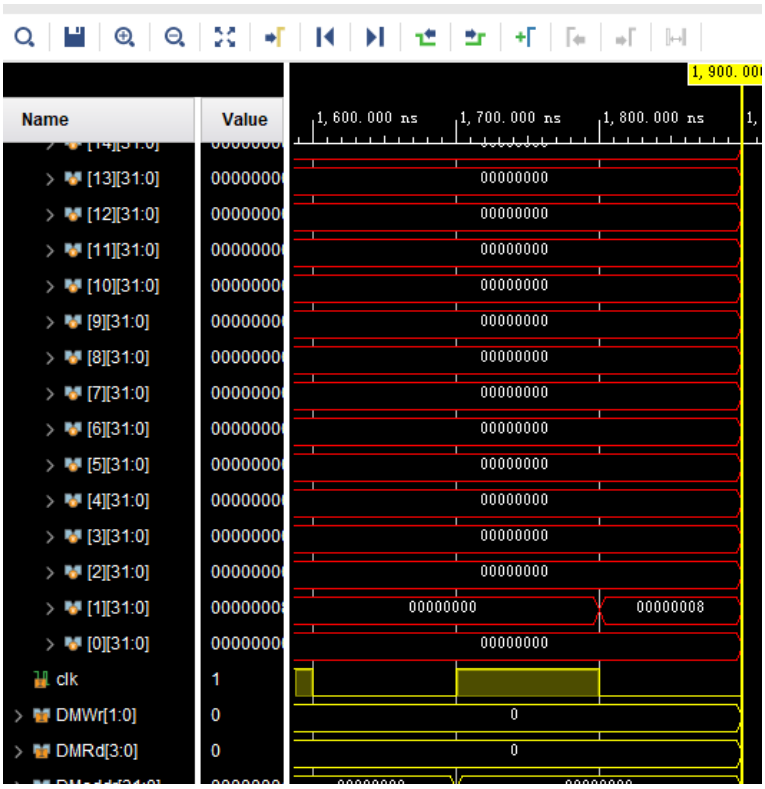


图-流水线指令运行

指令1：

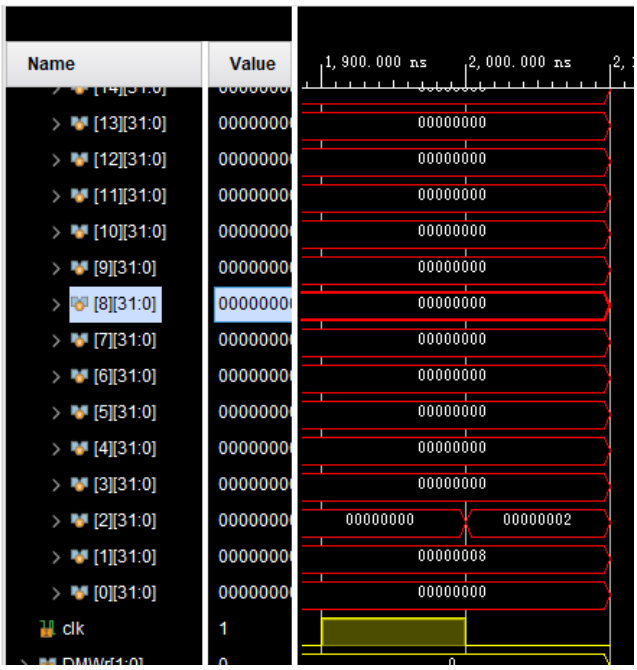
地址	汇编程序	16进制代码	预期结果
0x00000000	addiu　$1,$0,8	24010008	$1=8



在第四个时钟周期结束之后，寄存器1成功写入8，完成以第一条指令。

指令2：

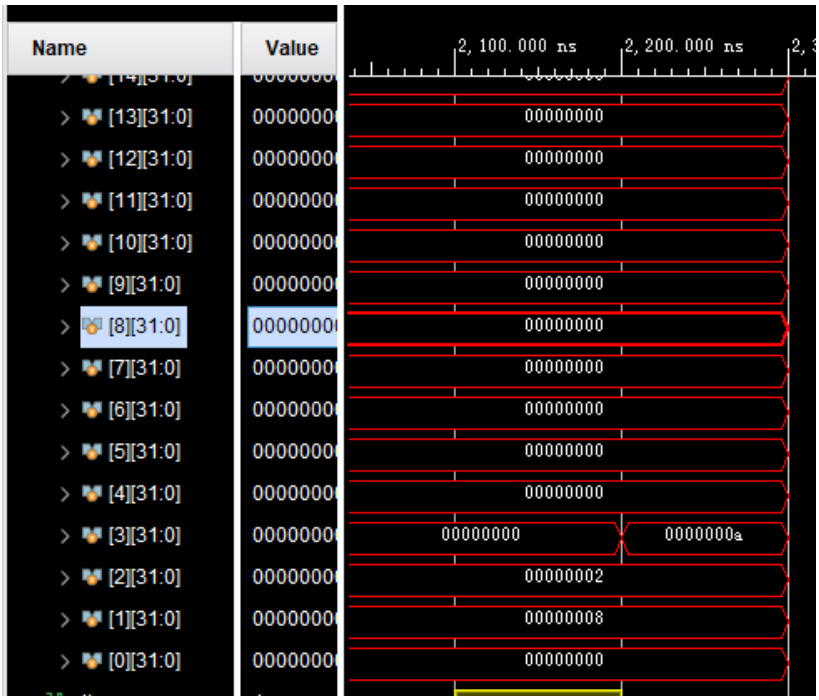
地址	汇编程序	16进制代码	预期结果
0x00000004	ori　$2,$0,2	34020002	$2=2



在第五个时钟周期结束之后，寄存器2成功写入2，完成以第二条指令。

指令3：

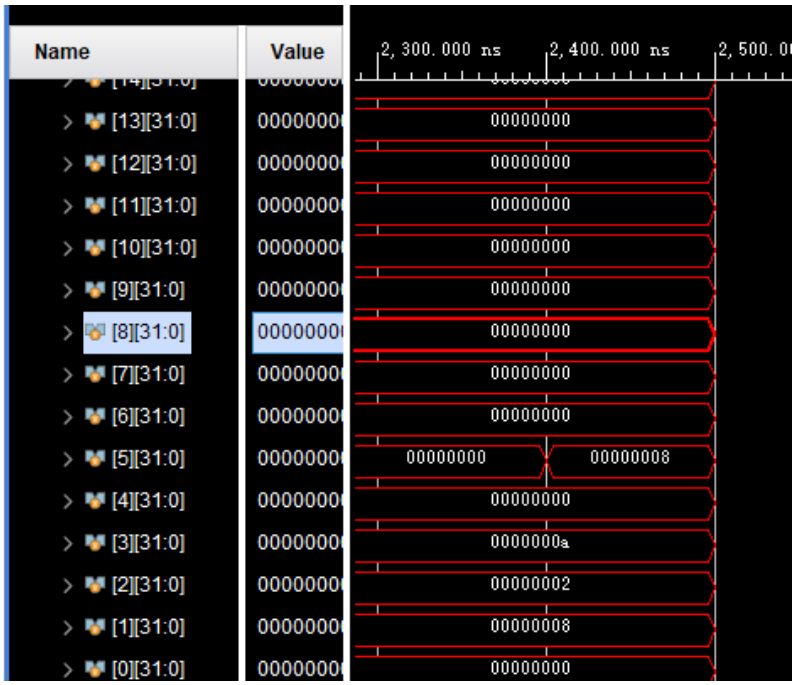
地址	汇编程序	16进制代码	预期结果
0x00000008	add $3,$2,$1	00411820	$3=10



第三条指令完成，$3=0x0a

指令4：

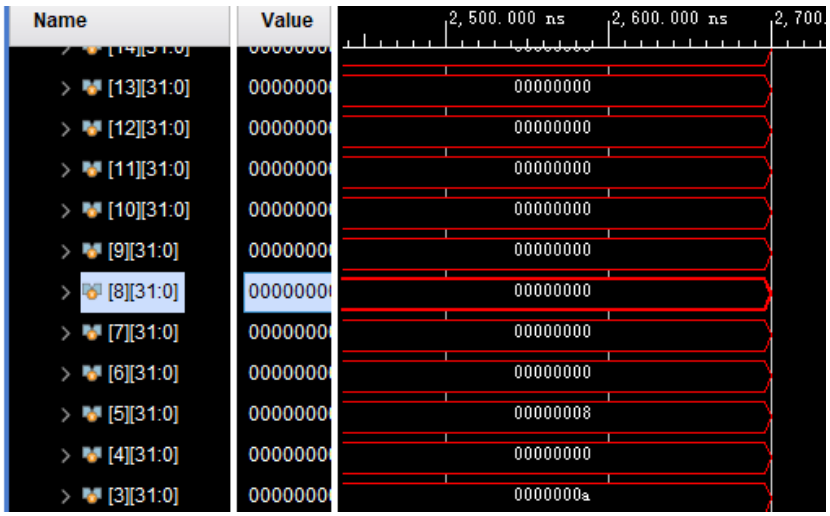
地址	汇编程序	16进制代码	预期结果
0x0000000C	sub $5,$3,$2	00622822	$5=8



第四条指令完成，$5=8

指令5：

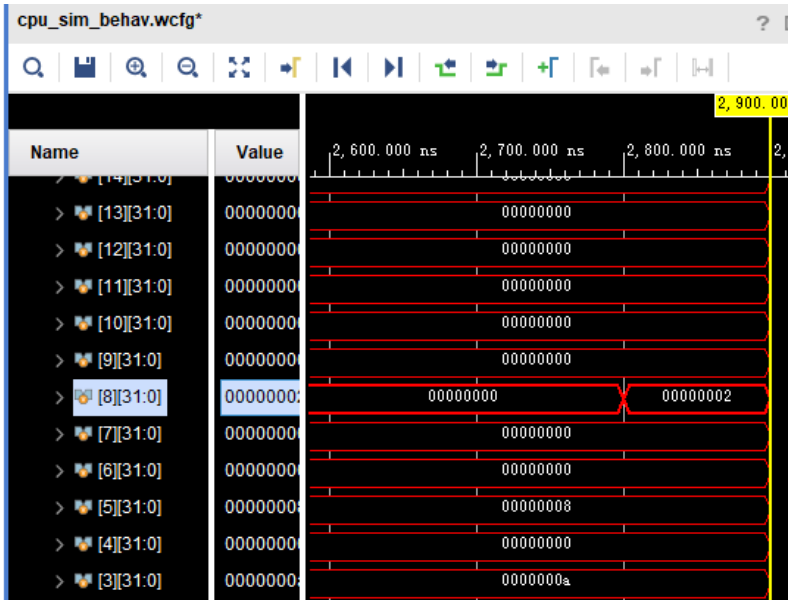
地址	汇编程序	16进制代码	预期结果
0x00000010	and $4,$5,$2	00a22024	$4=0



第五条指令完成，$4=0

指令6：

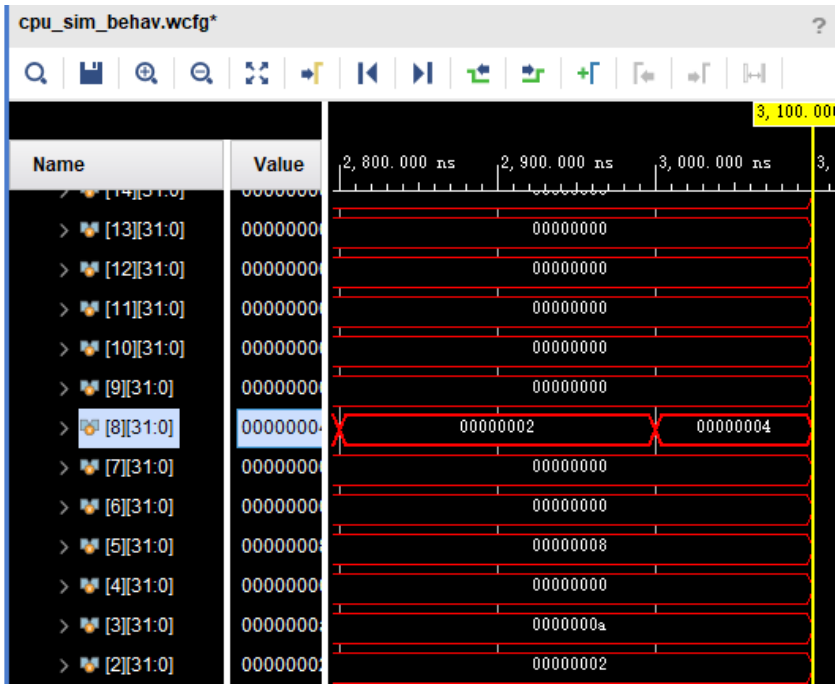
地址	汇编程序	16进制代码	预期结果
0x00000014	or $8,$4,$2	00824025	$8=2



第六条指令完成，$8=2

指令7：

地址	汇编程序	16进制代码	预期结果
0x00000018	sll $8,$8,1	00084040	$8=4

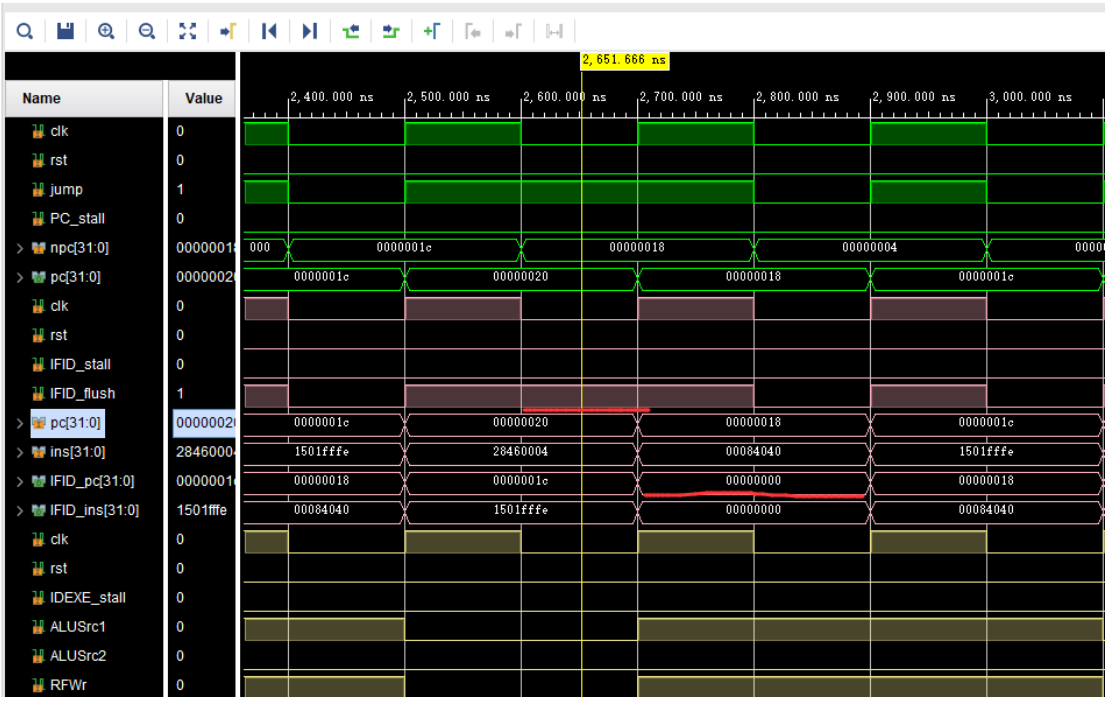


第七条指令完成，$8=4

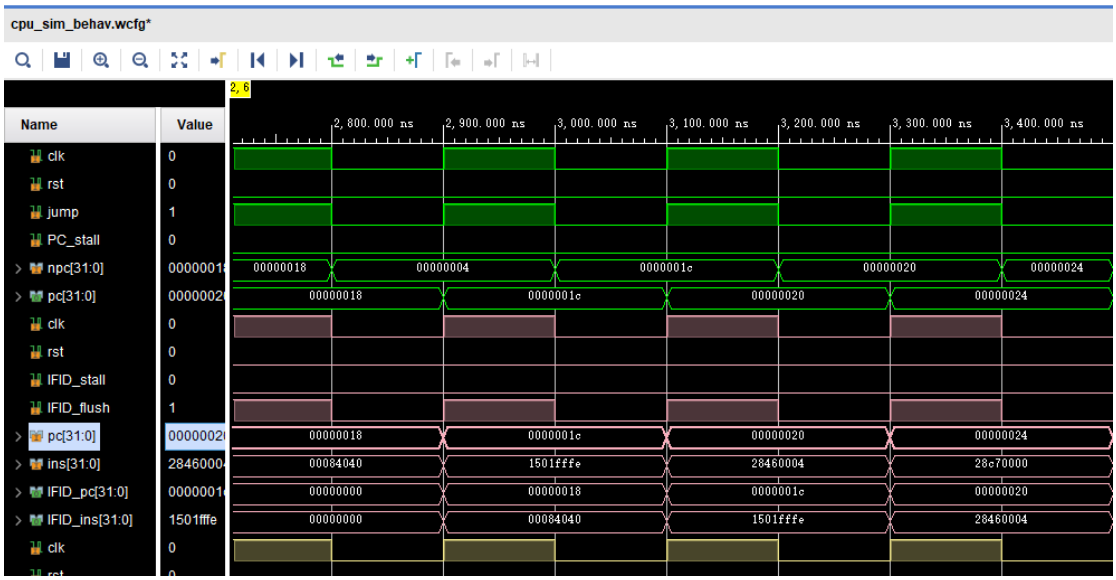
指令8：

地址	汇编程序	16进制代码	预期结果
0x0000001C	**bne $8,$1,-2 (≠, 转18)**	1501fffe	nextpc=0x18

第八条指令为bne指令，由于此时具有数据依赖，所以通过转发机制获取$8的值，判断此时需要执行分支指令，产生了一个flush信号



然后IF-ID阶段成功flush，下一个时钟周期的pc为零



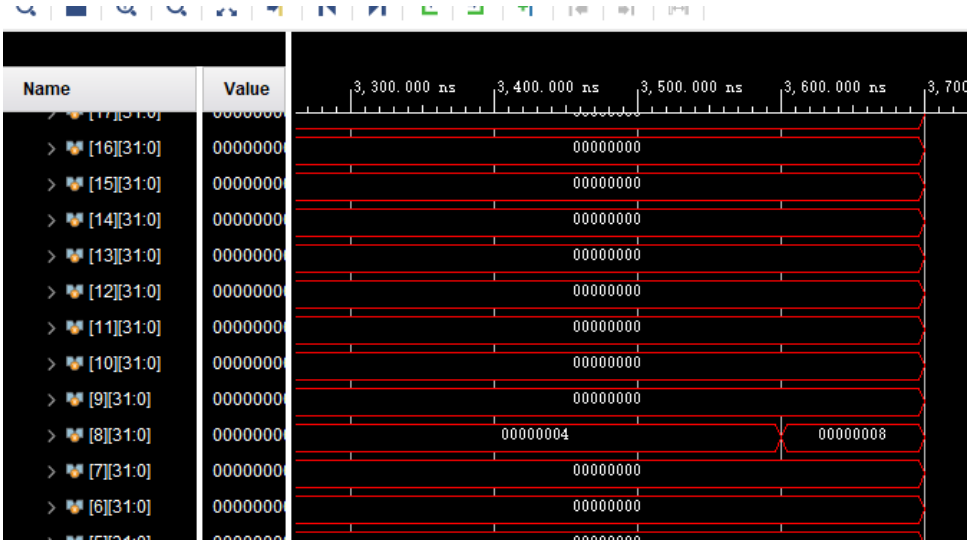
接下来CPU恢复正常执行，pc按照预期增长，第八条bne指令执行成功。

指令9：

地址	汇编程序	16进制代码	预期结果
0x00000018	sll $8,$8,1	00084040	$8=8

指令10：

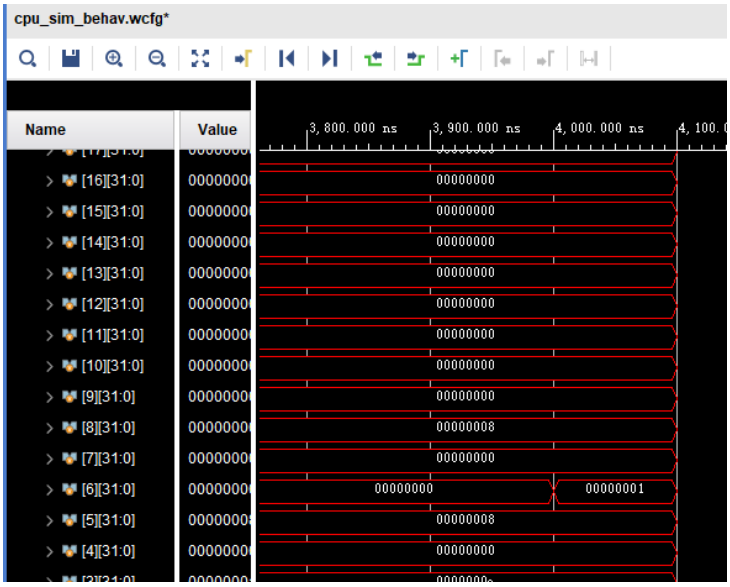
为之前指令8的bne指令，此处不符合跳转条件，继续正常执行



最终结果$8=8

指令11：

地址	汇编程序	16进制代码	预期结果
0x00000020	slti $6,$2,4	28460004	$6=1



第十一条指令完成，$6=1

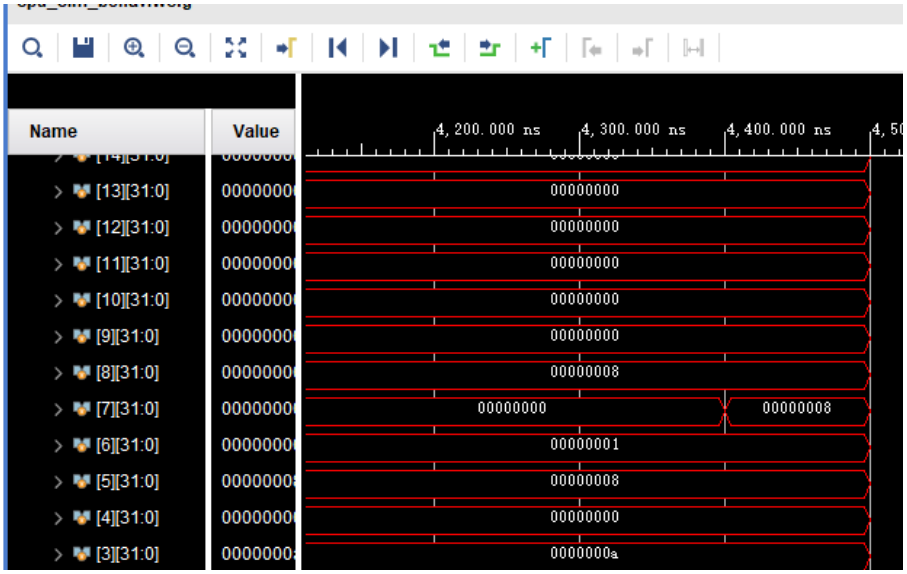
指令12：

地址	汇编程序	16进制代码	预期结果
0x00000024	slti $7,$6,0	28c70000	$7=0

第十二条指令完成，$7=0

指令13：

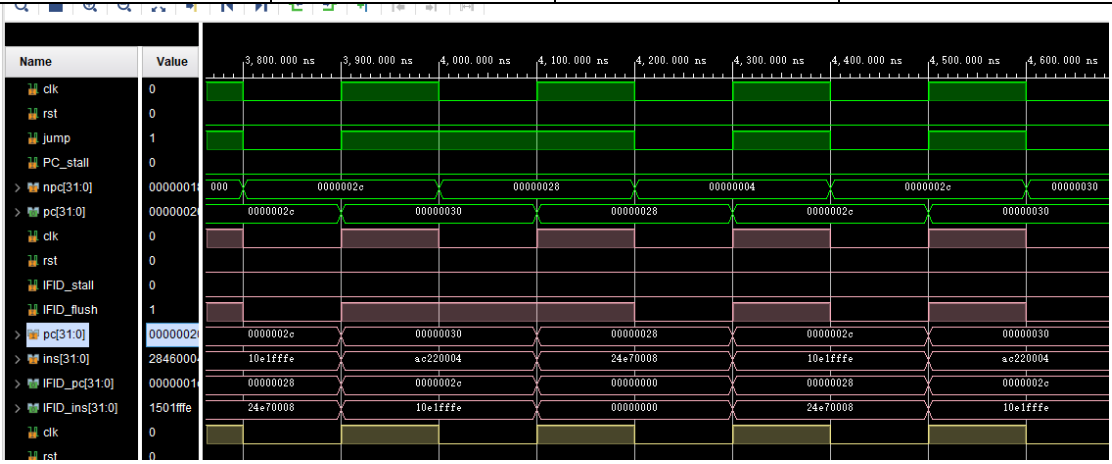
地址	汇编程序	16进制代码	预期结果
0x00000028	addiu $7,$7,8	24e70008	$7=8



第十三条指令完成，$7=8

指令14：

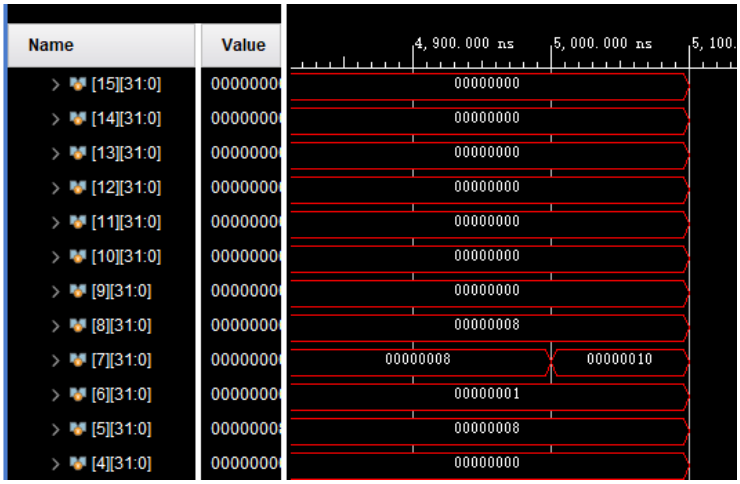
地址	汇编程序	16进制代码	预期结果
0x0000002C	**beq $7,$1,-2 (=, 转28)**	10e1fffe	nextpc=28



IF-ID的pc地址从28-2c-00-28成功跳转并恢复运行，成功产生flush信号。

指令15：

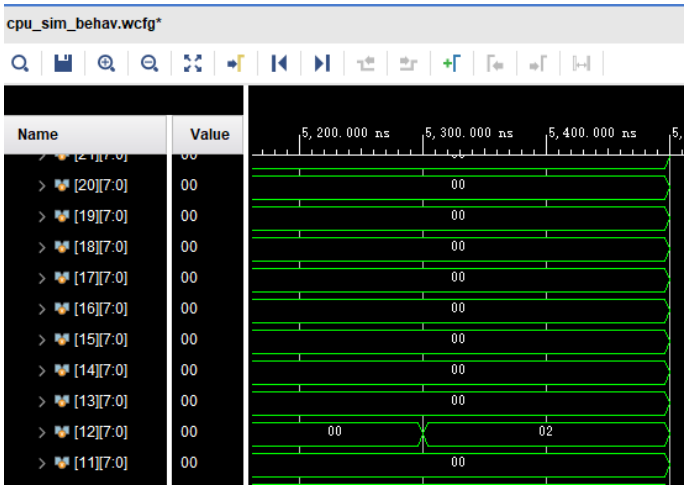
地址	汇编程序	16进制代码	预期结果
0x00000028	addiu $7,$7,8	24e70008	$7=16



第十五条指令完成，$7=16

指令16：

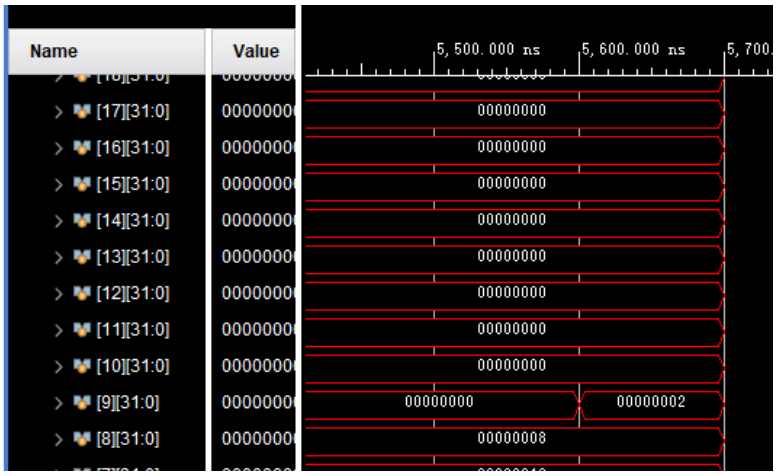
地址	汇编程序	16进制代码	预期结果
0x00000030	sw $2,4($1)	ac220004	M[12:15]=2



第十六条指令完成，M[12:15]=2

指令17：

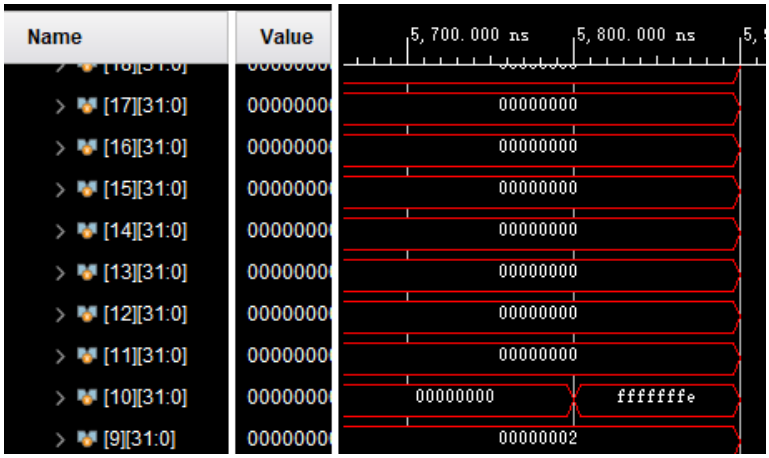
地址	汇编程序	16进制代码	预期结果
0x00000034	lw $9,4($1)	8c290004	$9=2



第十七条指令完成，$9=2

指令18：

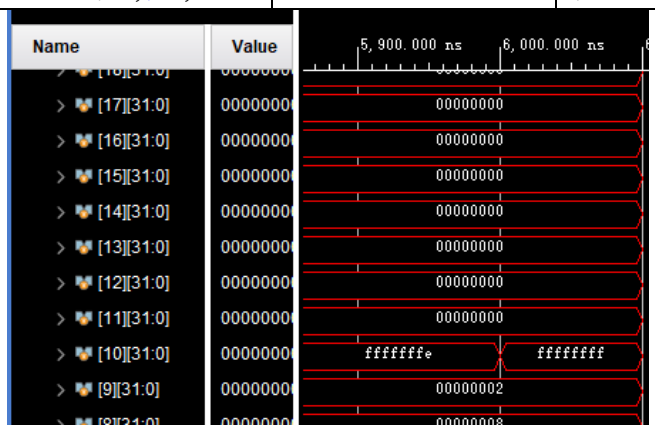
地址	汇编程序	16进制代码	预期结果
0x00000038	addiu $10,$0,-2	240afffe	$10 = -2



第十八条指令完成，$10=-2

指令19：

地址	汇编程序	16进制代码	预期结果
0x0000003C	addiu $10,$10,1	254a0001	$10 = -1



第十九条指令完成，$10=-1

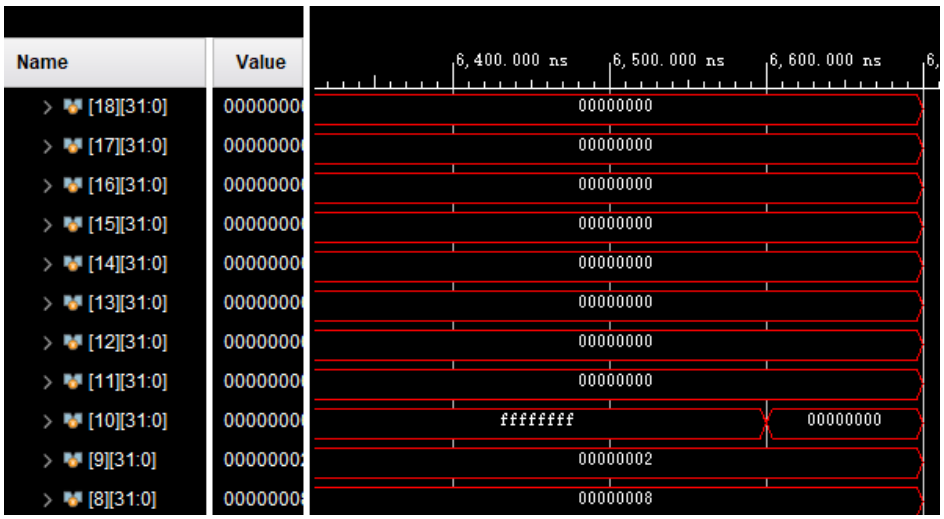
指令20：

地址	汇编程序	16进制代码	预期结果
0x00000040	**bltz $10,-2(<0,转 3C)**	0540fffe	nextpc=0x3c

第二十条指令完成,可见IF-ID的PC从3c-40-00-3c-40,完成了跳转的功能

指令21：

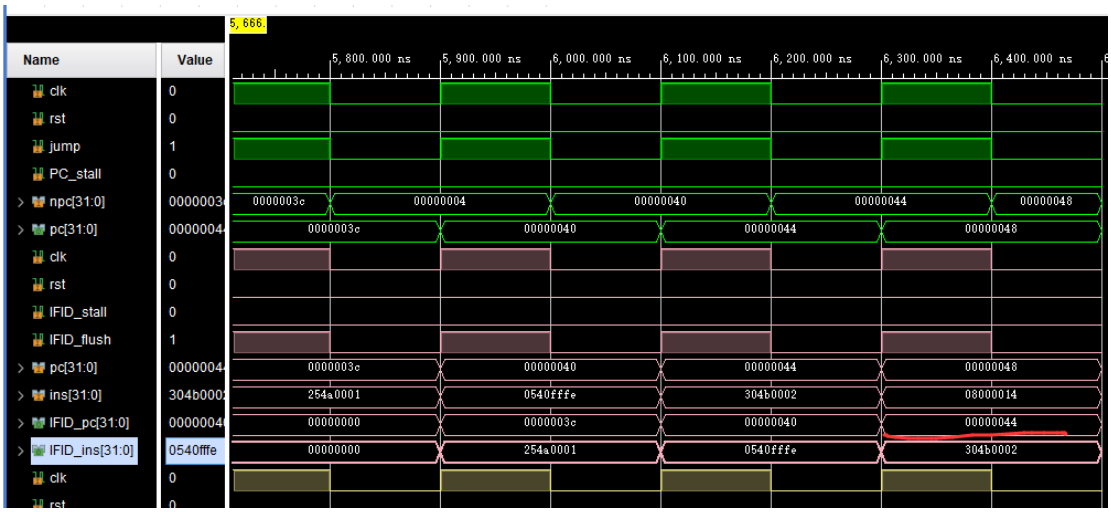
地址	汇编程序	16进制代码	预期结果
0x0000003C	addiu $10,$10,1	254a0001	$10 = 0



第二十一条指令完成，$10=0

指令22：

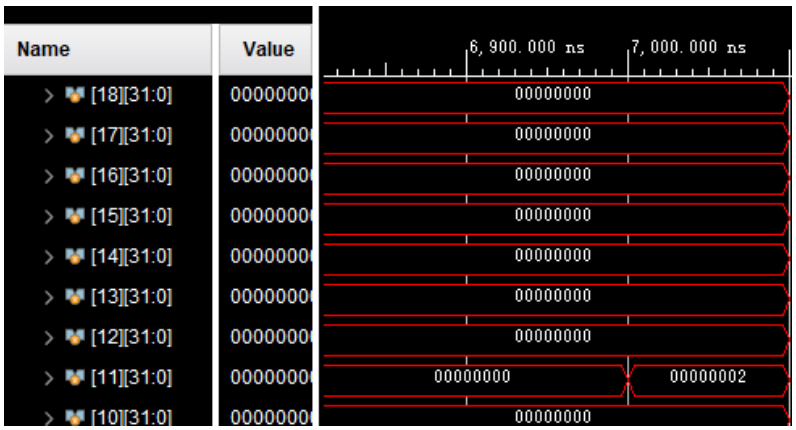
地址	汇编程序	16进制代码	预期结果
0x00000040	**bltz $10,-2**	0540fffe	nextpc=44



第二十二条指令执行完成，如图，非跳转，继续运行

指令23：

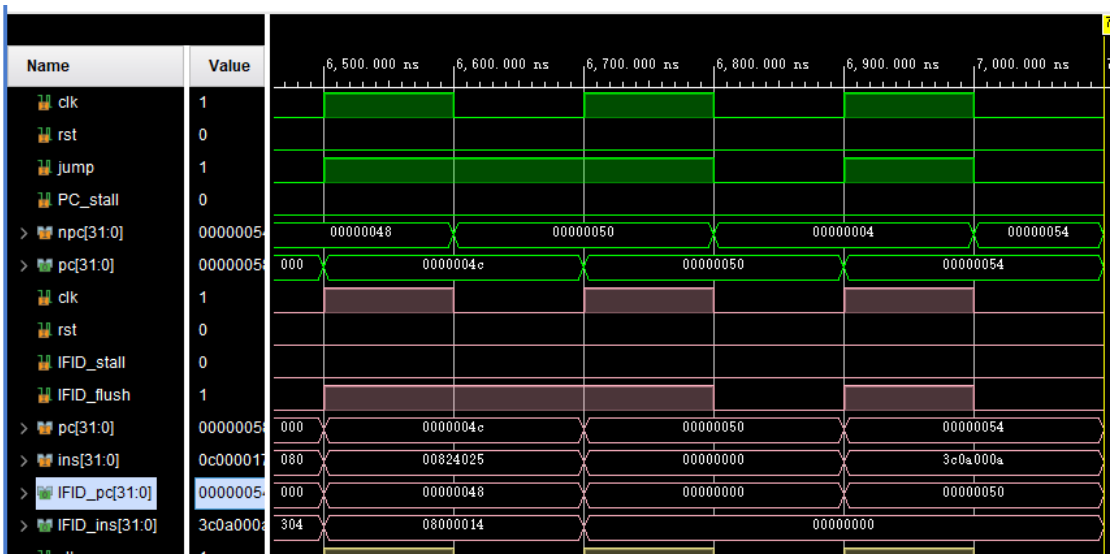
地址	汇编程序	16进制代码	预期结果
0x00000044	**andi $11,$2,2**	304b0002	$11 = 2



第二十三条指令完成，$11=2

指令24：

地址	汇编程序	16进制代码	预期结果
0x00000048	**j 0x00000050**	08000014	nextpc=50



成功执行跳转指令，IF-ID的pc从48-00-50完成了这个无条件跳转。

指令25：

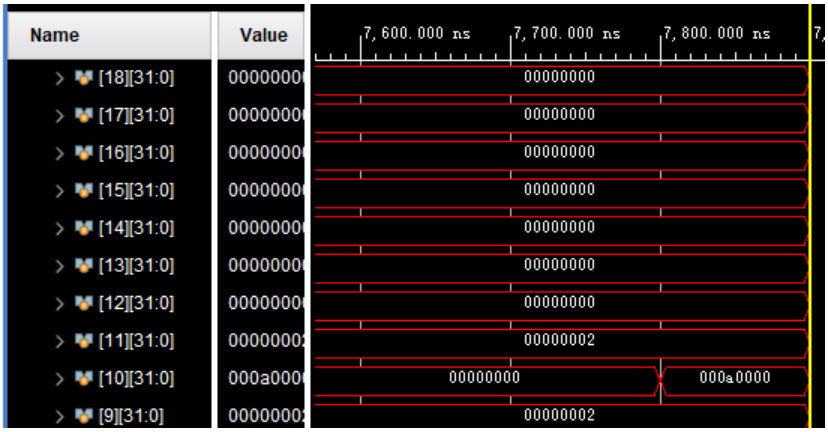
地址	汇编程序	16进制代码	预期结果
0x0000004C	or $8,$4,$2	00824025	该条指令被跳过

指令26：

地址	汇编程序	16进制代码	预期结果
0x00000050	**NOP**	00000000	无

指令27：

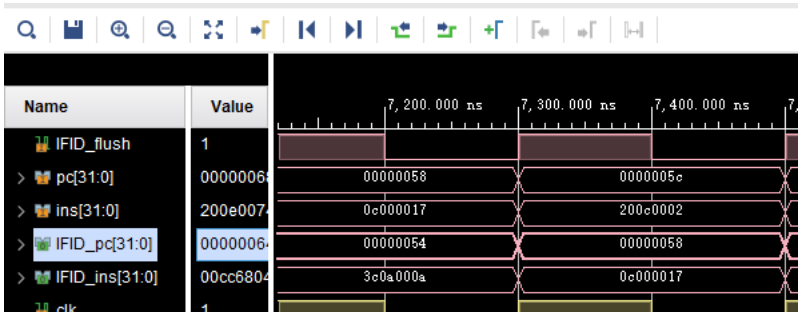
地址	汇编程序	16进制代码	预期结果
0x00000054	lui $10,10	3c0a000a	$10= 0x000a0000



第二十七条指令完成，$10= 0x000a0000

指令28

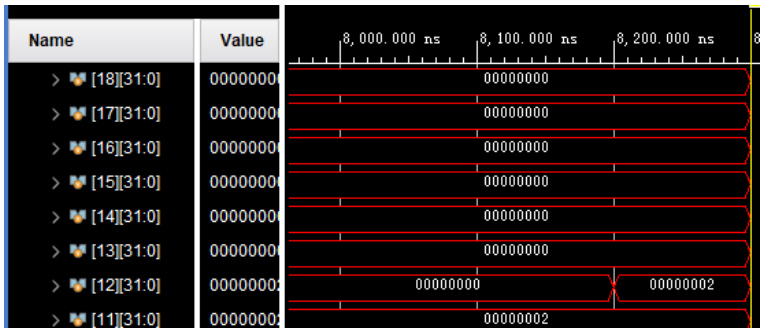
地址	汇编程序	16进制代码	预期结果
0x00000058	jal 0x1b	0c000017	Pc=5c



第二十八条指令完成，pc=5c

指令29

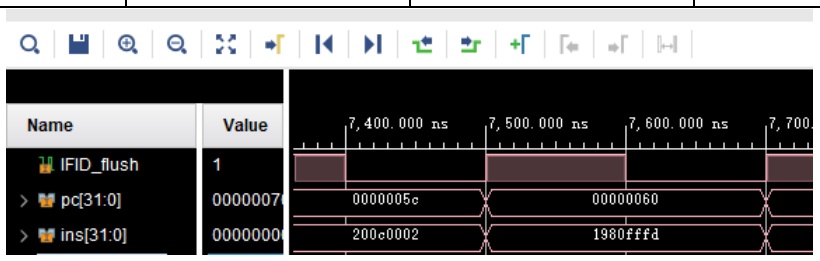
地址	汇编程序	16进制代码	预期结果
0x0000005C	addi $12,$0,2	200c0002	$12=2



第二十九条指令完成，$12=2

指令30

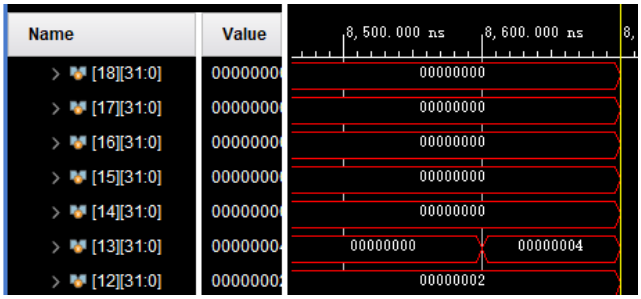
地址	汇编程序	16进制代码	预期结果
0x00000060	blez $12 -3	1980fffd	Pc=60



第三十条指令完成，pc=60

指令31

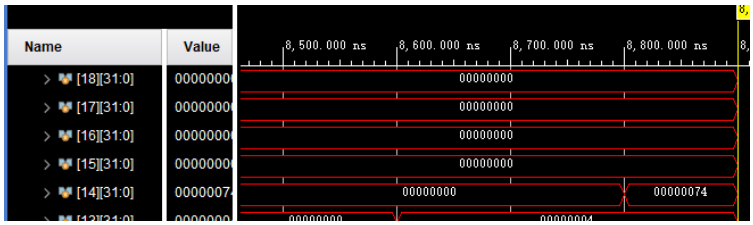
地址	汇编程序	16进制代码	预期结果
0x00000064	sllv $13,$12,$6	00cc6804	$13=4



第三十一条指令完成，$13=4

指令32

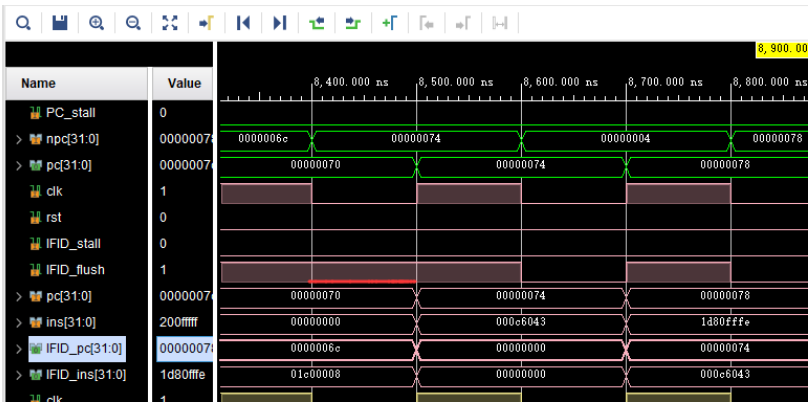
地址	汇编程序	16进制代码	预期结果
0x00000068	addi $14,$0,116	200e0074	$14=0x74



第三十二条指令完成，$14=0x74

指令33

地址	汇编程序	16进制代码	预期结果
0x0000006C	jr $14	01c00008	Pc=0x74



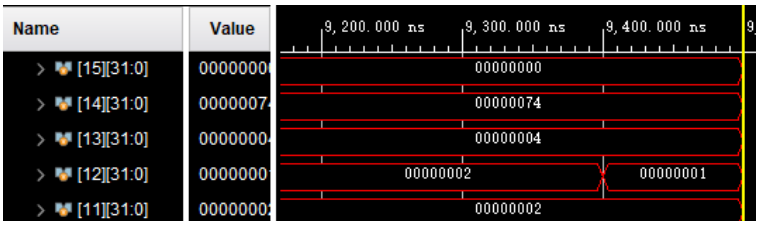
为跳转指令，IF-ID 阶段成功 flush，pc 由 6c-00-74 完成跳转

指令34

地址	汇编程序	16进制代码	预期结果
0x00000070	Nop	00000000	被跳过

指令35

地址	汇编程序	16进制代码	预期结果
0x00000074	sra $12,$12,1	000c6043	$12=1



第三十五条指令完成，$12=1

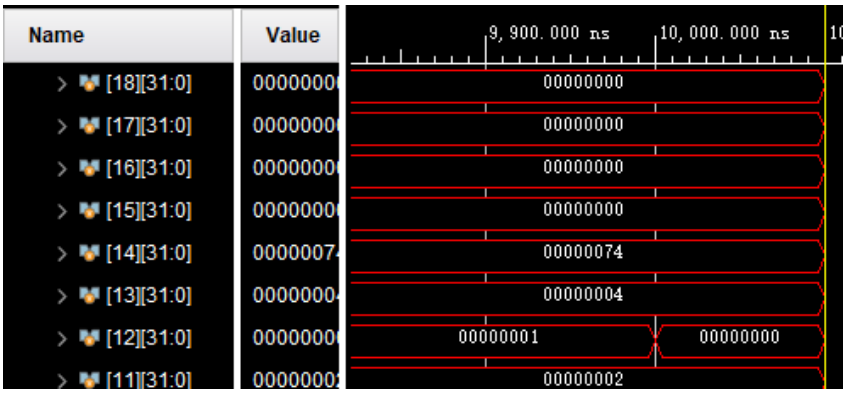
指令36

地址	汇编程序	16进制代码	预期结果
0x00000078	bgtz $12,-2	1d80fffe	跳转 74

未达到跳转条件，继续执行

指令37

地址	汇编程序	16进制代码	预期结果
0x00000074	sra $12,$12,1	000c6043	$12=0



第三十七条指令完成，$12=0

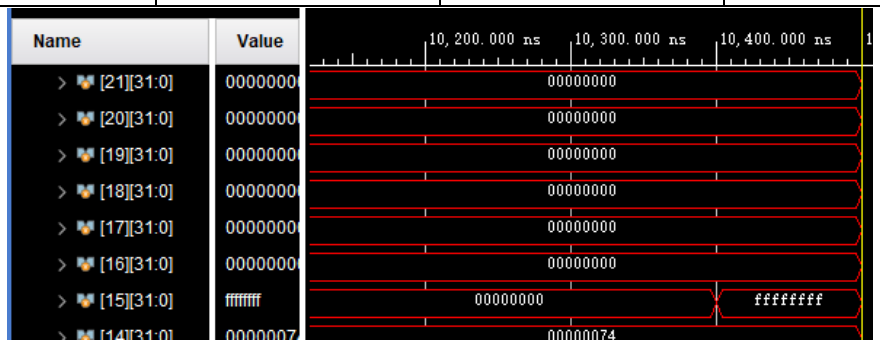
指令38

地址	汇编程序	16进制代码	预期结果
0x00000078	bgtz $12,-2	1d80fffe	不跳转

未达到跳转条件，继续执行

指令39

地址	汇编程序	16进制代码	预期结果
0x0000007C	addi $15,$0,-1	200fffff	$15=-1



第三十九条指令完成，$15=-1

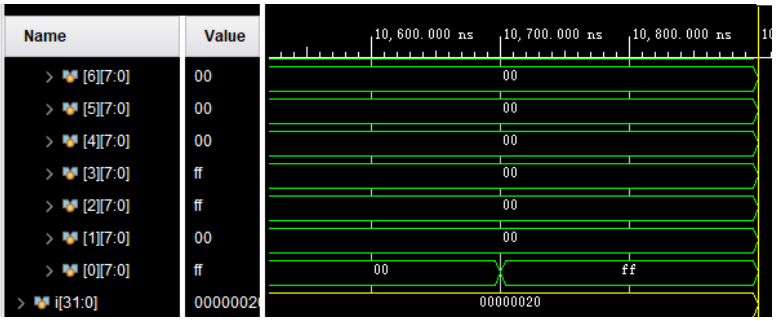
指令40

地址	汇编程序	16进制代码	预期结果
0x00000080	bgez $15,-4	05e1fffc	不跳转

未达到跳转条件，继续执行

指令41

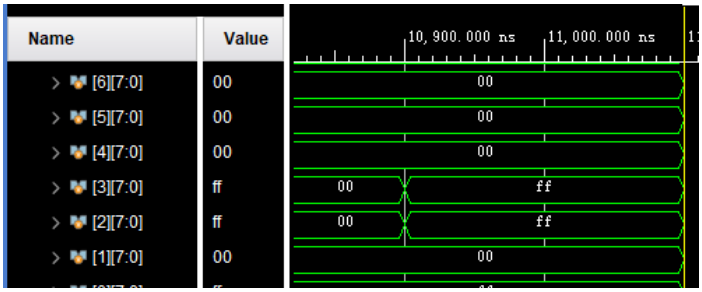
地址	汇编程序	16进制代码	预期结果
0x00000084	sb $15,0($s1)	a22f0000	mem[0]=-1



第四十一条指令完成，mem[0]=-1

指令42

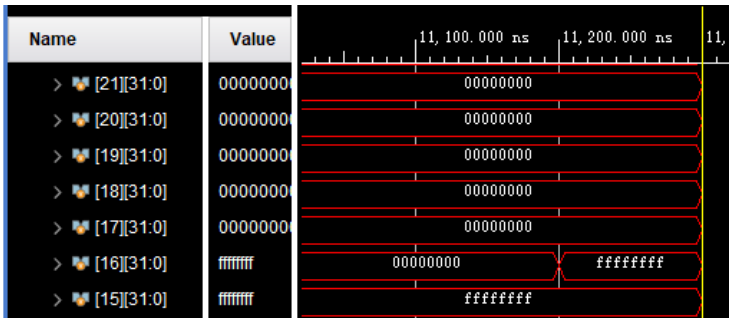
地址	汇编程序	16进制代码	预期结果
0x00000088	sh $15,2($s1)	a62f0002	mem[2:3]=-1



第四十二条指令完成，mem[2:3]=-1

指令 43

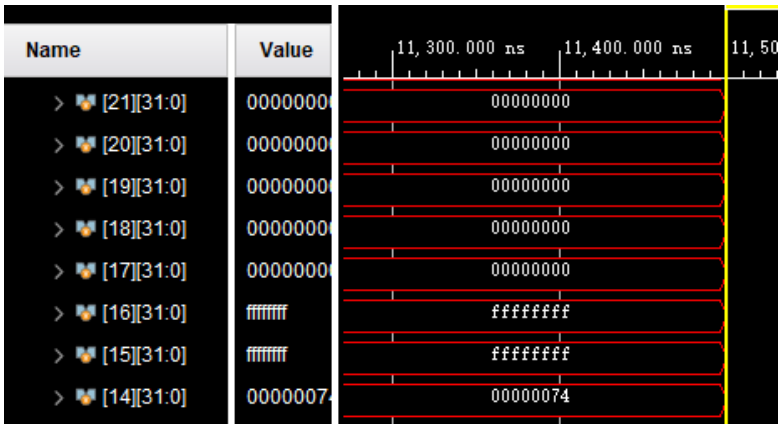
地址	汇编程序	16进制代码	预期结果
0x0000008C	lb $16,0($s1)	82300000	$16=-1



第四十三条指令完成，$16=-1

指令 44

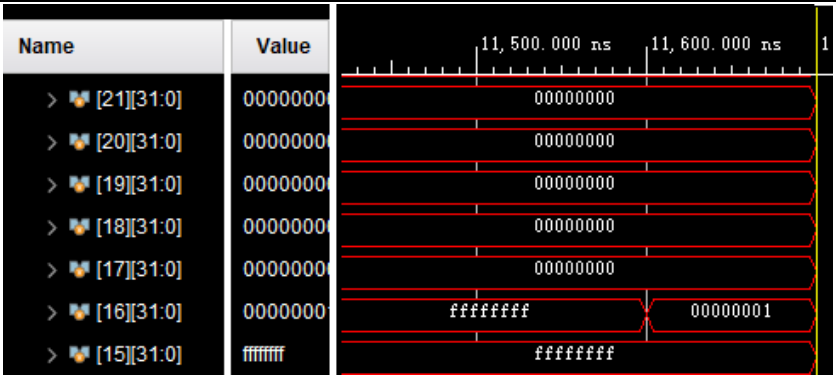
地址	汇编程序	16进制代码	预期结果
0x00000090	lh $16,2($s1)	86300002	$16=-1



第四十四条指令完成，$12=1

指令45

地址	汇编程序	16进制代码	预期结果
0x00000094	slt $16,$15,$0	01e0802a	$16=1

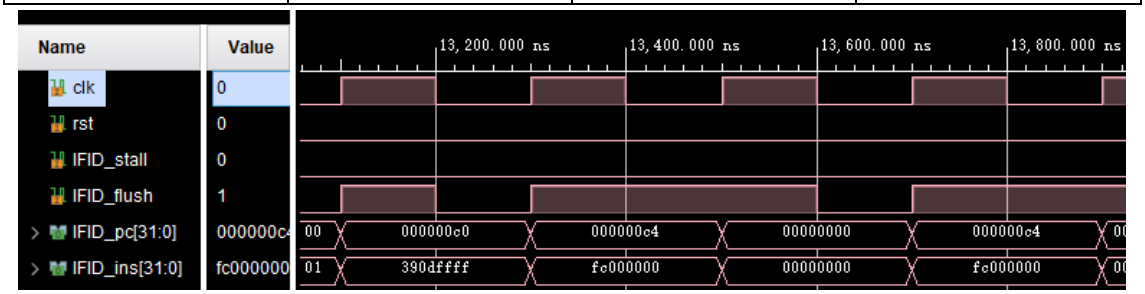


第四十五条指令完成，$12=1

这些内容过于重复，中间略去一部分，经过查验均与预期结果一致。

指令57

地址	汇编程序	16进制代码	预期结果
0x000000C4	**halt**	FC000000	停机



停机完成，IF-ID pc在fc-00之间循环，不再增加

仿真测试结束，寄存器写回和各种分支与跳转结果均符合预期结果。

五、实验心得

本次实验中，实现了一个能执行39条指令的，具有冒险检测与转发功能的流水线CPU。

整体来说，如果没有冒险现象，CPU会简易很多，只需要将每个信号和中间结果直接按照原始的方式转发到下一个模块的中间寄存器，每个流程运行的模块只需要关注自己那部分的信号和数据，逻辑清晰。但是考虑到跳转指令和数据依赖整个cpu就变的十分复杂，这也是当前cpu工业中对分支预测发展的重要性。在查阅资料的过程中，理解了如果需要进一步改进，需要改进哪些方向，例如多发射可以使得CPU具有多条流水线，超标量乱序执行可以使得cpu在发生跳转的时候不用flush而通过改变执行指令的顺序而正常运行该运行的指令，有的将传统的MIPS五级流水线的IF阶段单独拆分出来，取指令和发射两个阶段分开，多加了一级流水线成为六级流水线，龙芯杯cpu设计比赛优秀参赛作品的资料给了我很多启发。同时我也进一步理解了CPU和GPU之间的区别，GPU运算不支持跳转指令，可以加入很多的alu单元在一次运算中运行更多的操作，GPU对于浮点数的处理有更多特殊的硬件。

与单周期CPU相比，其中各个部件之间的耦合程度更高，直接使用原始单周期CPU结构进行改进会使得控制变的十分不合理，例如在单周期CPU中我们在alu运算执行完毕之后才判断是否发生了分支指令，但是如果在流水线CPU中也这样做的话，一是需要flush更多的指令，二是在无条件跳转和条件分支跳转的判断状态不同，有多种跳转的时机，为flush的控制加入了更多的无序性，也不利于编写易理解的代码。所以在流水线CPU中，将无条件跳转指令和条件分支指令的判断全部统一在ID-EXE阶段完成，这样仅仅需要flush一条

指令，提高了CPU的IPC性能。

对于转发模块和冒险检测模块编写也十分困难，这个问题不在于算法有多复杂，而在于这两个模块互相之间的耦合性，例如如果我一开始并没能梳理出完整的需要转发的信号，在实际的仿真测试中才发现了数据错误的bug（例如jr指令这样既涉及到跳转又涉及到寄存器读取的指令，需要转发到NPC模块），这样反过来去添加一条转发条件的时候，往往还需要更改冒险检测模块（添加了转发之后不需要加入stall指令了），还需要在顶层模块中添加或者更改数据选择器，有时候数据选择器的控制位数不够了还要继续更改，整个过程耦合度高，牵一发动全身，修改地十分困难。