



# 《计算机组成原理实验》

## 实验报告

(实验四)

学院名称 : 数据科学与计算机学院

专业(班级) : 18计教学三班

学生姓名 : 田蕊

学号 : 18340161

时间 : 2019 年 12 月 17 日

# 成 绩 :

---

## 实验四：流水线CPU设计与实现

---

### 一. 实验目的

- (1) 认识和掌握流水线 CPU 数据通路图的构成、原理及其设计方法；
- (2) 掌握流水线 CPU 的实现方法，代码实现方法；
- (3) 认识和掌握指令与 CPU 的关系；
- (4) 掌握流水线 CPU 的测试方法。

### 二. 实验内容

设计一个流水线CPU，该CPU在单周期指令集的基础上增加实现以下指令功能操作。

本次实验中需要实现运算操作的溢出判断：ALU运算操作溢出时，ALU需给出一位溢出信号（部分指令可能需要用到该信号。对于溢出发生时，需要能检测识别出，且不写回溢出错误结果，但不需要设计异常处理功能）。需设计的指令与格式如下，指令的具体描述和功能以mips官方文档为准：

#### ==>逻辑运算指令

- (1) ADDI rt, rs, immediate

|        |        |        |                |
|--------|--------|--------|----------------|
| 001000 | rs(5位) | rt(5位) | immediate(16位) |
|--------|--------|--------|----------------|

功能：GPR[rt]  $\leftarrow$  GPR[rs] + sign\_extend(immediate); immediate 做符号扩展再参加“加”运算。

#### ==>比较指令

- (1) SLT rd, rs, rt

|        |        |        |        |              |
|--------|--------|--------|--------|--------------|
| 000000 | rs(5位) | rt(5位) | rd(5位) | 00000 101010 |
|--------|--------|--------|--------|--------------|

功能：if (GPR[rs] < GPR[rt]) GPR[rd] = 1 else (GPR[rd] = 0)。

- (2) MOVN rd, rs, rt

|        |        |        |        |              |
|--------|--------|--------|--------|--------------|
| 000000 | rs(5位) | rt(5位) | rd(5位) | 00000 001011 |
|--------|--------|--------|--------|--------------|

功能：if GPR[rt]  $\neq$  0 then GPR[rd]  $\leftarrow$  GPR[rs]。

#### ==>访存指令

- (1) LHU rt, offset(base)

|        |          |        |             |
|--------|----------|--------|-------------|
| 100101 | base(5位) | rt(5位) | offset(16位) |
|--------|----------|--------|-------------|

功能：GPR[rt]  $\leftarrow$  memory[GPR[base] + offset]。

**==>跳转指令**

(1) JR rs

|        |        |            |    |        |
|--------|--------|------------|----|--------|
| 000000 | rs(5位) | 0000000000 | 未用 | 001000 |
|--------|--------|------------|----|--------|

功能:  $PC \leftarrow GPR[rs]$ , 跳转。**==>调用子程序指令**

(1) JAL addr

|        |            |
|--------|------------|
| 000011 | addr[27:2] |
|--------|------------|

功能: 调用子程序,  $PC \leftarrow \{PC[31:28], addr, 2'b0\}$ ;  $GPR[$31] \leftarrow pc+4$ , 返回地址设置; 子程序返回, 需用指令  $jr \$31$ 。跳转地址的形成同  $j$   $addr$  指令。**三. 实验原理****(一)、概述**

流水线 CPU 是将指令分解为多步，并让不同的指令的各步操作重叠，从而实现及条指令的并行处理。指令的每步有各自独立的电路来处理，每完成一步，就进到下一步，而前一步则处理后续指令。CPU 在处理指令时，一般需要经过以下几个阶段：

- (1) 取指令(IF): 根据程序计数器 pc 中的指令地址，从存储器中取出一条指令，同时，pc 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 pc，当然得到的“地址”需要做些变换才送入 pc。
- (2) 指令译码(ID): 对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。
- (3) 指令执行(EXE): 根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。
- (4) 存储器访问(MEM): 所有需要访问存储器的操作都将在这个步骤中执行，该步骤给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。
- (5) 结果写回(WB): 指令执行的结果或者访问存储器中得到的数据写回相应的目的寄存器中。



图 1 流水线 CPU 一条指令处理过程

MIPS 指令的三种格式：

**R 类型：**

|     |       |       |       |       |       |   |
|-----|-------|-------|-------|-------|-------|---|
| 31  | 26 25 | 21 20 | 16 15 | 11 10 | 6 5   | 0 |
| op  | rs    | rt    | rd    | sa    | funct |   |
| 6 位 | 5 位   | 5 位   | 5 位   | 5 位   | 6 位   |   |

**I 类型：**

|     |       |       |           |   |
|-----|-------|-------|-----------|---|
| 31  | 26 25 | 21 20 | 16 15     | 0 |
| op  | rs    | rt    | immediate |   |
| 6 位 | 5 位   | 5 位   | 16 位      |   |

**J 类型：**

|     |         |   |
|-----|---------|---|
| 31  | 26 25   | 0 |
| op  | address |   |
| 6 位 | 26 位    |   |

其中，

**op**: 为操作码；

**rs**: 为第 1 个源操作数寄存器，寄存器地址（编号）是 00000~11111, 00~1F；

**rt**: 为第 2 个源操作数寄存器，或目的操作数寄存器，寄存器地址（同上）；

**rd**: 为目的操作数寄存器，寄存器地址（同上）；

**sa**: 为位移量 (shift amt)，移位指令用于指定移多少位；

**funct**: 为功能码，在寄存器类型指令中 (R 类型) 用来指定指令的功能；

**immediate**: 为 16 位立即数，用作无符号的逻辑操作数、有符号的算术操作数、数据加载 (Load) / 数据保存 (Store) 指令的数据地址字节偏移量和分支指令中相对程序计数器 (PC) 的有符号偏移量；

**address**: 为地址。

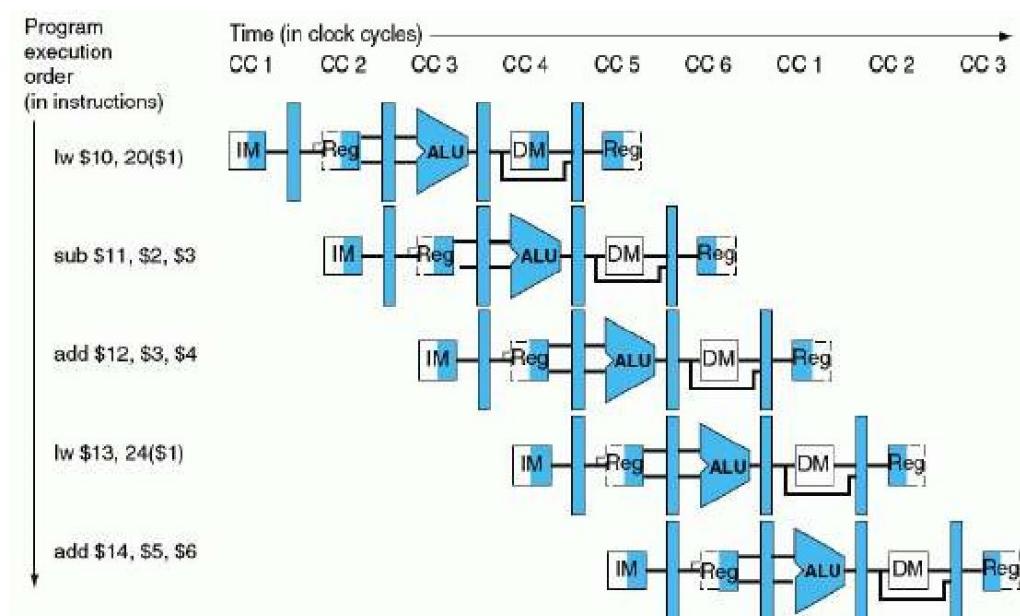


图 2 五级流水线 CPU 指令处理过程

## (二)、数据通路

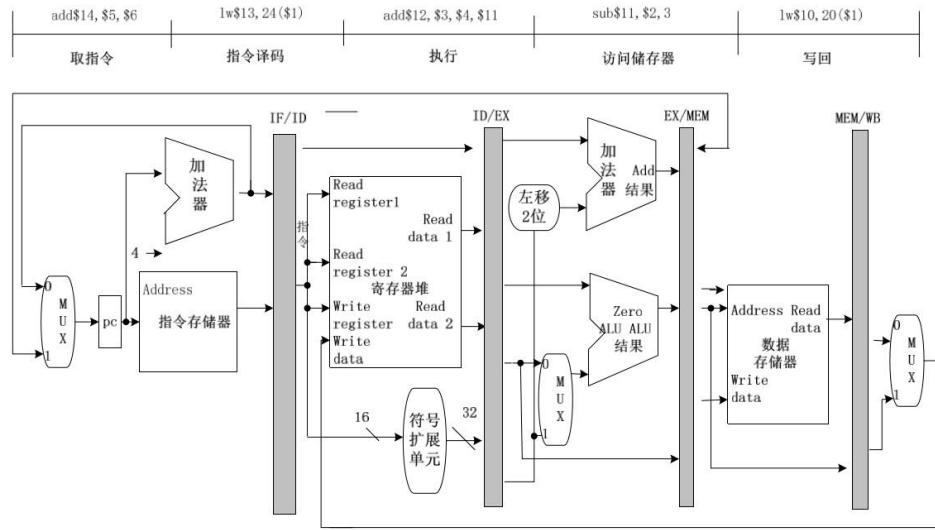


图 3 五级流水线 CPU 同一个时钟周期内五条指令的数据通路

图 3 是一个简单的基本上能够在流水线 CPU 上完成所要求设计的指令功能的数据通路和必要的控制线路图。其中各段的数据和控制信号需要保存到流水段寄存器中。访问存储器时，先给出内存地址，然后由读或写信号控制操作。对于寄存器组，给出寄存器地址（编号），读操作时不需要时钟信号，输出端就直接输出相应数据；而在写操作时，在 WE 使能信号为 1 时，在时钟边沿触发将数据写入寄存器。

## (三)、控制信号的产生

本实验涉及到的控制信号包括Reset, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRw, mRD, mWR, RegDst, ExtSel, PCSrc[1:0], ALUop[2:0].

### 1、Reset:

状态为0: 初始化PC为0

状态为1: PC接受新地址

### 2、PCWre:

状态为0: PC不更改，相关指令: halt

状态为1: PC更改，相关指令: 除指令halt外

### 3、ALUSrcA:

状态为0: 来自寄存器堆data1输出，相关指令: add、sub、addiu、or、and、andi、ori、slti、beq、bne、bltz、sw、lw

状态为1: 来自移位数sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令: sll

### 4、ALUSrcB:

状态为0: 来自寄存器堆data2输出, 相关指令: add、sub、or、and、beq、bne、bltz

状态为1: 来自sign或zero扩展的立即数, 相关指令: addi、andi、ori、slt、sw、lw

#### 5、DBDataSrc:

状态为0: 来自ALU运算结果的输出, 相关指令: add、addiu、sub、ori、or、and、andi、slt、sll

状态为1: 来自数据存储器 (Data MEM) 的输出, 相关指令: lw

#### 6、RegWre:

状态为0: 无写寄存器组寄存器, 相关指令: beq、bne、bltz、sw、halt

状态为1: 寄存器组写使能, 相关指令: add、addiu、sub、ori、or、and、andi、slt、sll、lw

#### 7、WrRegDSrc[1:0]

状态为00: 写入寄存器组的数据来自于PC+4

状态为01: 写入寄存器组的数据来自于存储器

状态为10: 写入寄存器组的数据来自于寄存器

状态为11: 写入寄存器组的数据来自于ALU

#### 8、InsMemRW:

状态为0: 写指令存储器

状态为1: 读指令存储器(Ins. Data)

#### 9、mRD:

状态为0: 输出高阻态

状态为1: 读数据存储器, 相关指令: lw

#### 10、mWR:

状态为0: 无操作

状态为1: 写数据存储器, 相关指令: sw

#### 11、RegDst[1:0]:

状态为 00:  $pc < -pc + 4$ , 相关指令: add、addiu、sub、and、andi、ori、xori、slt、slt、sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0);

状态为 01:  $pc < -pc + 4 + (\text{sign-extend}) \times 4$ , 相关指令: beq(zero=1)、bne(zero=0)、bltz(sign=1);

状态为 10: pc<-rs, 相关指令: jr;

状态为11: pc<-{pc[31:28],addr[27:2],2'b00}, 相关指令: j、jal;

#### 12、ExtSel:

状态为0: (zero-extend)immediate (0扩展), 相关指令: addiu、andi、ori

状态为1: (sign-extend)immediate (符号扩展), 相关指令: slti、sw、lw、beq、  
bne、bltz

#### 13、PCSrc[1..0]:

00: pc<-pc+4, 相关指令: add、addiu、sub、or、ori、and、andi、slti、  
sll、sw、lw、beq(zero=0)、bne(zero=1)、bltz(sign=0);

01 : pc<-pc+4+(sign-extend)immediate , 相关指令 : beq(zero==1)、  
bne(zero==0)、bltz(sign==1);

10: pc<-{(pc+4)[31:28],addr[27:2],2'b00}, 相关指令: j;

11: pc<-{pc[31:28],addr[27:2],2'b00}, 相关指令: j、jal;

#### 14、IRWre

状态为0: IR(指令寄存器)不更改

状态为1: IR寄存器写使能。向指令存储器发出读指令代码后, 这个信号也接着发出, 在时钟上升沿, IR接收从指令存储器送来的指令代码。与每条指令都相关。

#### 15、ALUOp[2..0]:

ALU 8种运算功能选择(000-111)

| ALUOp[2..0] | 功能   | 描述               |
|-------------|--|------------------|
| 000         | $Y = A + B$  | 加                |
| 001         | $Y = A - B$  | 减                |
| 010         | $Y = B \ll A$  | B 左移 A 位         |
| 011         | $Y = A \vee B$   | 或                |
| 100         | $Y = A \wedge B$   | 与                |
| 101         | $Y = (A < B) ? 1: 0$   | 比较 A 与 B<br>不带符号 |
| 110         | $Y = (((reg_a < reg_b) \&\& (reg_a[31] == reg_b[31]))    ((reg_a[31] == 1 \&\& reg_b[31] == 0))) ? 1: 0$ | 比较 A 与 B<br>带符号  |
| 111         | $Y = (B == 0)$   | 将 B 和 0 比较       |

## 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

### (一)、CPU设计的思想，方法

#### 1、综述

CPU设计的总体思路是根据各个部件的输入，输出及其内部功能确定各个连线之间的内部逻辑，再用硬件描述语言将这个逻辑表达出来。CPU的设计我们将其分为各个功能部件分别设计，这样在不同的功能部件之间只需要考虑接口的设计，而不需要关心内部的连线实现与关系。对于流水线CPU的设计，设计的过程需要添加流水段寄存器，各个寄存器保存这一条指令的数据和控制信号，寄存器的数值随着指令的流动而改变。流水线CPU设计的过程，就是将一条指令再详细剖分的过程，在这个过程里各个段相互并行，设计的过程中要考虑各个部件的控制信号需要由那一段的数据控制。

#### 2、流程图

根据流程图我们可以分析出各个部件之间的接口联系，根据这些接口我们可以确定各个部件在设计的时候所需要的输入和输出。在设计好各个部件接口的输入和输出之后，再根据流程图将各个部件连接起来，就可以得到我们最终的CPU。

#### 3、控制信号表

控制信号表的主要作用是帮助我们对Control Unit进行控制。根据控制信号表的内容确定各个指令对应控制信号的取值。控制信号是根据各个指令的op和func部分得到的。对于R型指令，我们根据func部分推断出各个控制信号的取值；对于非R型指令，我们根据op部分推断出各个控制信号的取值。得到控制信号，我们就相当于得到了一张真值表，根据真值表的内容我们可以设计出ALU的逻辑，更新PC的数据来源，DataMemory的来源，RegisterFile的数据来源和目的寄存器的地址等等一些列的部件的控制。同时，在分析出控制信号的取值之后，控制单元也可以根据控制信号的取值逻辑来表示。

|                | add                | sub                | addiu              | andi | and | ori | or  | sll | sw  | lw     | beq | bne | bltz | j   | halt | addi | slt                | movn                   | lh <u>u</u>        | jr  | jal |      |
|----------------|--------------------|--------------------|--------------------|------|-----|-----|-----|-----|-----|--------|-----|-----|------|-----|------|------|--------------------|------------------------|--------------------|-----|-----|------|
| Reset          | 1                  | 1                  | 1                  | 1    | 1   | 1   | 1   | 1   | 1   | 1      | 1   | 1   | 1    | 1   | 1    | 1    | 1                  | 1                      | 1                  | 1   | 1   |      |
| PCWre          | 1                  | 1                  | 1                  | 1    | 1   | 1   | 1   | 1   | 1   | 1      | 1   | 1   | 1    | 1   | 0    | 1    | 1                  | 1                      | 1                  | 1   | 1   |      |
| ALUSrcA        | 0                  | 0                  | 0                  | 0    | 0   | 0   | 0   | 0   | 0   | 0      | 0   | 0   | 0    | 0   | 0    | X    | 0                  | 0                      | X                  | X   | X   |      |
| ALUSrcB        | 0                  | 0                  | 1                  | 1    | 0   | 1   | 0   | 0   | 1   | 1      | 1   | 0   | 0    | 0   | X    | X    | 1                  | 0                      | X                  | 1   | X   |      |
| ReqWre         | SF=0->1<br>SF=1->0 | SF=0->1<br>SF=1->0 | SF=0->1<br>SF=1->0 | 1    | 1   | 1   | 1   | 1   | 1   | 1      | 1   | 0   | 0    | 0   | 0    | X    | SF=0->1<br>SF=1->0 | zero=1->1<br>zero=0->0 | SF=0->1<br>SF=1->0 | 0   | 1   |      |
| WrRegDsrc[1:0] | 11                 | 11                 | 11                 | 11   | 11  | 11  | 11  | 11  | 11  | 11     | 11  | 1   | 0    | 1   | 0    | X    | X                  | 11                     | 11                 | 10  | 01  | X 00 |
| InsMemRw       | 1                  | 1                  | 1                  | 1    | 1   | 1   | 1   | 1   | 1   | 1      | 1   | 1   | 1    | 1   | 1    | X    | X                  | 1                      | 1                  | 1   | 1   | 1    |
| mRD            | 0                  | 0                  | 0                  | 0    | 0   | 0   | 0   | 0   | 0   | 0      | 0   | 0   | 0    | 0   | 0    | 0    | 0                  | 0                      | 0                  | 1   | 0   | 0    |
| mWR            | 0                  | 0                  | 0                  | 0    | 0   | 0   | 0   | 0   | 0   | 0      | 0   | 0   | 0    | 0   | 0    | 0    | 0                  | 0                      | 0                  | 0   | 0   | 0    |
| lRVre          | 1                  | 1                  | 1                  | 1    | 1   | 1   | 1   | 1   | 1   | 1      | 1   | 1   | 1    | 1   | 1    | 1    | 1                  | 1                      | 1                  | 1   | 1   | 1    |
| ExtSel         | X                  | X                  | 1                  | 0    | X   | 0   | X   | 0   | 1   | 1      | 1   | 1   | 1    | 1   | 1    | X    | X                  | 1                      | X                  | X   | 1   | X X  |
| PCSrc[1:0]     | 00                 | 00                 | 00                 | 00   | 00  | 00  | 00  | 00  | 00  | 00     | 00  | 00  | 00   | 00  | 00   | 00   | 00                 | 00                     | 00                 | 00  | 10  | 11   |
| ReqDst         | 10                 | 10                 | 01                 | 01   | 10  | 01  | 10  | 10  | 01  | X 01   | X   | X   | X    | X   | X    | 01   | 10                 | 10                     | 01                 | X   | 00  |      |
| ALUop[2:0]     | 000                | 001                | 000                | 100  | 100 | 011 | 001 | 010 | 110 | 000000 | 001 | 001 | 001  | XXX | XXX  | 000  | 110                | 111                    | 000                | XXX | XXX |      |

图 4 控制信号的取值

#### 4、各个部件的设计

对于流水线CPU有许多部件都可以对单周期的部件进行复用，包括Ins\_Memory, Instruction\_divided, SignZeroExtend, Register\_File, Data\_Memory。对于重复的部件我们在这里不再赘述，重点讲述做出了修改和新增的部件。

##### (1)、ALU部件

对于这次的指令设计由于要求实现判溢出功能，所以此次ALU不仅需要计算出数据的结果，而且需要增加OF信号。OF信号判断ALU运算的过程中是否发生了数据溢出，如果发生了数据溢出则OF位置1。

判断溢出的方式有很多，包括通过进位进行异或运算来判断等等。我们这里采用比较简单的一种方式。由于只有同号相加，异号相减和移位的时候才有可能发生数据溢出，所以只在这加法和减法情况下，判断操作数和结果是否满足操作数异号而结果与被加（减）数也异号，则说明此种情况下发生了溢出。对于移位，只需判断移位前的最高位和移位后的最高位是否一致，若不一致，则发生了溢出，OF位置1。需要特别注意的是，因为这里判溢出是根据计算后的结果来判断的，所以这里必须用阻塞赋值。

```

case(ALUop)
 3'b000 :
  begin
    result = A + B;
    OF = (A[31]^B[31]) ? 1'b0 : (result[31]^A[31]) ? 1'b1 : 1'b0;
    zero = (result==0);
  end
 3'b001 :
  begin
    result = A - B;
    OF = (A[31]==B[31]) ? 1'b0 : (result[31]!=A[31]) ? 1'b1 : 1'b0 ;
    zero = (result==0);
  end
 3'b010 :
  begin
    result = B << A;
    OF = B[31]^result[31];
    zero = (result==0);
  end

```

图 5 加法，减法和移位部分的 ALU 设计

## (2)、PC

PC部分根据PCWre信号对指令进行更新，与单周期CPU不同的是，由于流水新啊CPU可能会有数据冒险和控制冒险，所以需要输入一个keep信号，来决定这里是否要有阻塞，阻塞时即将PC的之保持不变而不更新。同时PC仍然是每个周期都要根据时钟信号和PCWre信号进行判断，同时根据Reset信号的取值决定是否要清零。需要注意的是Always内部的触发信号不仅要有时钟的下降沿，同时还要有Reset的下降沿。

```

begin
  if(Reset==0)//如果要清零的话，PC值全部赋为0
    Instr_Addr[31:0] <= 32'h00000000;
  else if(PCWre && !keep)//如果PC的写使能为真，那么就要写入新的PC
    Instr_Addr[31:0] <= newPC;
end

```

图 6 PC 部件核心部分

## (3)、控制单元部件

控制单元是控制信号产生的部件，在这个部分内我们通过对指令的op段和func段对指令进行译码。译码出的控制信号主要是根据之前列出的控制信号表产生的。为了方便代码的阅读，我们先定义一些常量来表示各个指令，如果是R型指令，则用func部分定义，如果不是R型

指令，则用op部分定义。

```

parameter ADD = 6'b100000;
parameter SUB = 6'b100010;
parameter AND = 6'b100100;
parameter OR = 6'b100101;
parameter SLL = 6'b000000;
parameter ADDIU = 6'b001001;
parameter ANDI = 6'b001100;
parameter ORI = 6'b001101;
parameter SLTI = 6'b001010;
parameter SW = 6'b101011;
parameter LW = 6'b100011;
parameter BEQ = 6'b000100;
parameter BNE = 6'b000101;
parameter BLTZ = 6'b000001;
parameter J = 6'b000010;
parameter HALT = 6'b111111;
parameter ADDI = 6'b001000;
parameter SLT = 6'b101010;
parameter MOVN = 6'b001011;
parameter LHU = 6'b100101;
parameter JR = 6'b001000;
parameter JAL = 6'b000011;

```

图 7 常量的定义

由于此次新增了JAL, MOVN等指令，使得写入寄存器中的数值不一定再是存储器中的数值和ALU结果，还有可能是PC+4的值和寄存器组中的数值，所以此次我们直接将WrRegDSrc设置成两位位宽而不再需要DBDataSrc控制信号。同时对于RegWre信号，因为需要判断是否溢出，还需要根据OF信号来决定取值。

```
RegWre = (func==JR) ? 1'b0 : (func==AND||func==OR||func==SLT) ? 1'b1 : (OF!=0) ? 1'b0 : 1'b1;
```

图 8 R型指令能够 RegWre 信号的生成

```
RegWre = (op==ANDI||op==ORI||op==SLTI||op==LW||op==SLT||op==JAL||op==ADDI)?1'b1:(op==SW||op==BEQ||op==BNE||op==BLTZ||op==J||op==HALT)?1'b0:(OF==0)?1'b1:1'b0;
```

图 9 非 R 类型指令能够 RegWre 信号的生成

#### (4)、IF\_ID 流水段寄存器

先在就到了我们流水线 CPU 最核心的部分啦啦啦啦啦！流水段寄存器最主要需要考虑的一个问题就是有哪些信号和值需要存入流水段寄存器。这个内容是需要根据各部分实现的内容和功能进行判断的，对于 IF\_ID 段寄存器，由于 IF 段所实现的功能很简单，所需要的控制信号不多，而且此时指令还没有译码，指令的控制信号还没有产生也就自然不需要向下传递，所以这一部分所需要传入和保存的数据内容相对较少，只需要保存当前 PC+4 后的值和根据当前 PC 值取出的指令。但是需要注意的是，为了解决流水线 CPU 的数据冒险和控制冒险，我们这里还传入了 keep 信号和 br 信号，我们还要根据这些信号来决定是不是要保存流水段寄存器的值不变。

```

always@ (posedge CLK)
begin
    if (!Reset || BR) begin
        PC_add4_ID <= 32'h0;
        Instruction_ID <= 32'b0;
    end
    else if (stall || keep) begin           //一旦阻塞，立刻信号保持
        PC_add4_ID <= PC_add4_ID;
        Instruction_ID <= Instruction_ID;
    end
    else if (IRWre) begin                 //如果不阻塞，那就写入
        PC_add4_ID <= PC_add4_IF;
        Instruction_ID <= Instruction_IF;
    end
    else begin                           //不阻塞也不写入，就保持
        PC_add4_ID <= PC_add4_IF;
        Instruction_ID <= Instruction_IF;
    end
end

```

图 10 IF\_ID 段寄存器核心部分

#### (5)、ID\_EX 流水段寄存器

IF\_ID 段寄存器保存的是 IF 段产生的值，而 ID\_EX 段寄存器保存的则是 ID 段产生的值，因为在 ID 段指令已经进行了译码，所以此时需要存入段寄存器的信号就变得很多，这里我们需要自行判断在后续部分需要用到的控制信号然后将其保存到流水段寄存器中，同时还要讲从 Register\_File 中取得的数据保存在流水段寄存器中。同时仍需要传入 BR, keep, stall 信号来决定是否需要把这个流水段寄存器的数值保存。

```
module ID_EX(
    input CLK,
    input Reset,
    input BR,
    input keep,
    input [31:0] DataA_ID,
    input [31:0] DataB_ID,
    input [31:0] PC_Addr4_ID,
    input [31:0] imm_ID,
    input [25:0] j_Addr_ID,
    input stall,
    input ALUSrcA_ID,
    input ALUSrcB_ID,
    input RegWre_ID,
    input mRD_ID,
    input mWR_ID,
    input [1:0] PCSrc,
    input [1:0] WrRegDSrc_ID,
    input [2:0] ALUop_ID,
    input [4:0] DstReg_ID,
    input [4:0] sa_ID,
    input [4:0] Rs_ID,
    input [4:0] Rt_ID,
    input [5:0] op_ID,
    output reg [31:0] DataA_EX,
    output reg [31:0] DataB_EX,
    output reg [31:0] PC_Addr4_EX,
    output reg [31:0] imm_EX,
    output reg [25:0] j_Addr_EX,
    output reg ALUSrcA_EX,
    output reg ALUSrcB_EX,
    output reg RegWre_EX,
    output reg mRD_EX,
    output reg mWR_EX,
    output reg [1:0] PCSrc_EX,
    output reg [2:0] ALUop_EX,
    output reg [1:0] WrRegDSrc_EX,
    output reg [4:0] DstReg_EX,
    output reg [4:0] sa_EX,
    output reg [4:0] Rs_EX,
    output reg [4:0] Rt_EX,
    output reg [5:0] op_EX,
    output reg [1:0] PCSRC_EX_NOT_CLEAR
);
```

图 11 ID\_EX 段寄存器数据接口

### (6)、EX\_MEM 段寄存器

与前面的类似，EX\_MEM 段寄存器仍然需要保存后面指令执行的过程需要用到的信号的值和数据，但是这个流水段寄存器不需要再传入 stall 信号，因为跳转指令最早在 EX 段可以判断出是否需要跳转，这时最少取错了两条指令。所以影响到的只有前两个段寄存器，而第三个和第四个段寄存器是不受影响的，所以 EX\_MEM 段寄存器和 EX\_WB 段寄存器是不需要加入 stall 信号来控制它阻塞的。

```
module EX_MEM(
    input CLK,
    input keep,
    input Reset,
    input RegWre_EX,
    input mRD_EX,
    input mWR_EX,
    input [1:0] WrRegDSrc_EX,
    input [4:0] DstReg_EX,
    input [31:0] DataA_EX,
    input [31:0] DataB_EX,
    input [31:0] Jump_PC_EX,
    input [31:0] ALUResult_EX,
    input [31:0] PC_Addr4_EX,
    input Zero_EX,
    input Sign_EX,
    input OF_EX,
    output reg RegWre_MEM,
    output reg mRD_MEM,
    output reg mWR_MEM,
    output reg [1:0] WrRegDSrc_MEM,
    output reg [4:0] DstReg_MEM,
    output reg [31:0] DataA_MEM,
    output reg [31:0] DataB_MEM,
    output reg [31:0] Jump_PC_MEM,
    output reg [31:0] ALUResult_MEM,
    output reg [31:0] PC_Addr4_MEM,
    output reg Zero_MEM,
    output reg Sign_MEM,
    output reg OF_MEM
```

图 12 EX\_MEM 段寄存器数据接口

### (7)、MEM\_WB 段寄存器

到 MEM\_WB 段寄存器部分所需要保存的信号就有一次重新变少了，只需要保存寄存器写使能信号，目的寄存器，写入寄存器的数据来源，从寄存器组中读出的 Rs 寄存器中的内容（为了实现 MOVN 指令），ALU 的运算结果，从数据存储器中读出的内容和 PC+4 的值，以及 OF 标志位。具体的实现过程和之前的流水段寄存器都是类似的，都是根据 Reset 信号，keep 信号和时钟来决定寄存器的写入与否和写入时间。正如前面所说，这一部分的寄存器是不需要传入 stall 信号的。

### (8)、Forward\_Unit

这一部分是控制数据转发的部件。由于流水线 CPU 会出现数据冒险的情况，所以需要一个特殊的控制单元我们这里将其称为 Forward\_Unit，来判断这条指令是否有产生数据冒险，判断条件即根据上一条指令的目的寄存器和这一条指令的源操作数寄存器来决定是否需要实行数据的转发。值得注意的是，这一部分 Forward\_Unit，只能解决 LW 指令意外的数据冒险，对于 LW 指令产生的数据冒险，是不能通过转发来解决的。这一部分的单元输出两个信号，分别是 ForwardA 和 ForwardB，这两个信号将传入 ALUSrcA 和 ALUSrcB 两个数据的选择单元中，并根据 ForwardA 和 ForwardB 的值决定数据来源。具体的控制信号如下

| 多选器控制         | 源      | 解释                                |
|---------------|--------|-----------------------------------|
| ForwardA = 00 | ID_EX  | 第一个 ALU 操作数来自寄存器                  |
| ForwardA = 01 | EX_MEM | 第一个 ALU 操作数由上一个 ALU 运算结果旁路获得      |
| ForwardA = 10 | MEM_WB | 第一个 ALU 操作数从数据存储器或前面的 ALU 结果中旁路获得 |
| ForwardB = 00 | ID_EX  | 第二个 ALU 操作数来自寄存器                  |
| ForwardB = 01 | EM_MEM | 第二个 ALU 操作数由上一个 ALU 运算结果旁路获得      |
| ForwardB = 10 | MEM_WB | 第二个 ALU 操作数由数据存储器或前面的 ALU 结果旁路获得  |

下面给出检测冒险条件以及解决冒险的控制信号

```

always@(*) begin
    if(RegWre_MEM && DstReg_MEM!=0 && DstReg_MEM==Rs_EX && !OF_MEM)
        forwardA = 2'b10;
    else if(RegWre_WB && DstReg_WB!=0 &&
        !(RegWre_MEM && (DstReg_MEM!=0) && (DstReg_MEM == Rs_EX)) &&
        DstReg_WB == Rs_EX && !OF_MEM)
        forwardA = 2'b01;
    else
        forwardA = 2'b00;
end

```

图 13 ForwardA 部分的控制逻辑

```

always@(*) begin
    if(RegWre_MEM && DstReg_MEM!=0 && DstReg_MEM==Rt_EX&&!OF_MEM)
        forwardB = 2'b10;
    else if(RegWre_WB && DstReg_WB!=0 &&
        !(RegWre_MEM && (DstReg_MEM!=0) && (DstReg_MEM == Rt_EX)) &&
        DstReg_WB == Rt_EX&&!OF_MEM)
        forwardB = 2'b01;
    else
        forwardB = 2'b00;
end

```

图 14 ForwardB 部分的控制逻辑

## (9)、br\_control

这一部分是跳转控制单元，这一部分是根据 EX 段 ALU 算出的结果和指令来决定是否要跳转，如果需要跳转那么 BR 信号置一，否则为零。

```

always@(*) begin
    if(PCSsrc_EX == 2'b01) begin
        BR = (op_EX==BEQ&&Zero_EX==1)?1'b1:(op_EX==BNE&&Zero_EX==0)?1'b1:(op_EX==BLTZ&&Sign_EX==1)?1'b1:1'b0;
    end
    else if(PCSsrc_EX != 2'b00) begin
        BR = 1'b1;
    end
    else begin
        BR = 1'b0;
    end
end

```

图 15 Br\_Control 核心部分

## (二)、各条指令的波形截图

由于是流水线 CPU，一个周期同时执行多条指令，但是每一条指令从开始到结束都需要五个时钟周期，所以我们这里将每次截图五个时钟周期的信号，其中 EX 后缀的信号就是当前正在执行的指令。

### (1)、addiu \$1,\$0,5

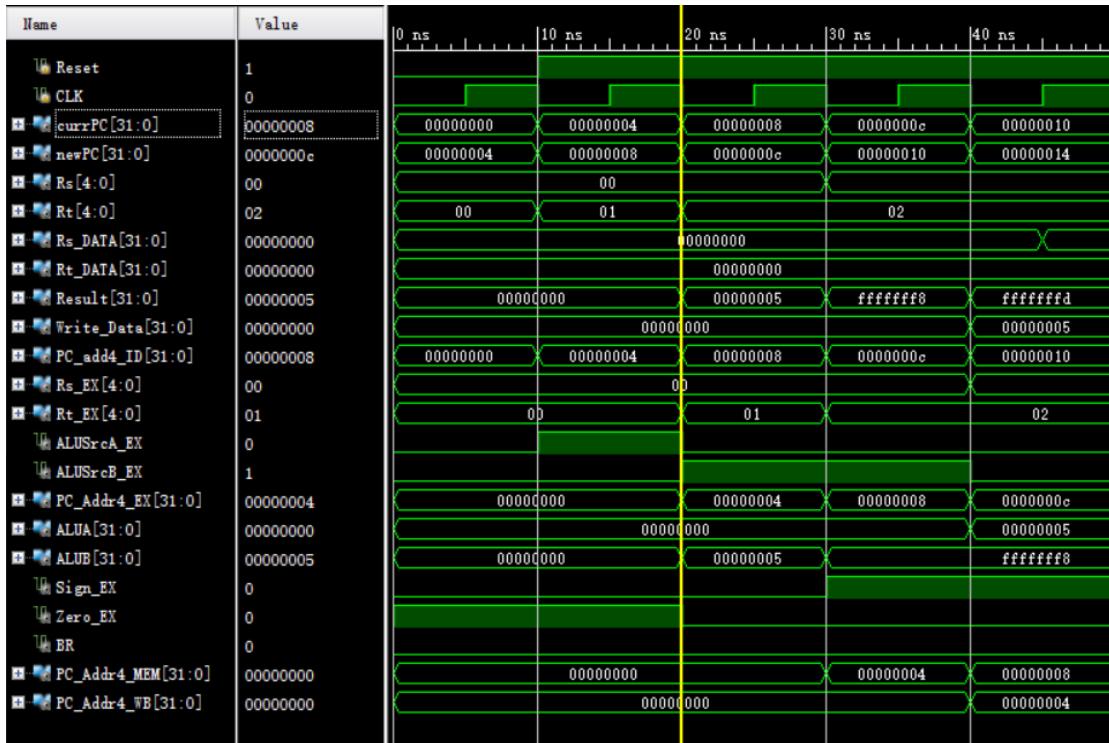


图 16 addiu \$1,\$0,5

如上即为 addiu \$1,\$0,5 的波形图，可以看到 Rs\_EX 和 Rt\_EX 分别为 0 和 1，即为指令中的寄存器号码。这是一条 I 型指令，我们可以看见第一个操作数是零号寄存器中的数值，即为 0，第二个操作数为常数 5，正如波形图中 ALUA 和 ALUB 中的值，分别为 0 和 5。运算结果为 5，正如波形图中 Result 中的值所示，同时可以看到第五个时钟周期的时候，Write\_Data 中的值为 5，即写入目的寄存器中的数据为 5。结果正确。

### (2)、add \$1,\$1,\$2

这一条指令为 add \$1,\$1,\$2，根据波形图可以看出此时 EX 段的 RS 寄存器和 RT 寄存器分别为一号寄存器和二号寄存器，和指令中的寄存器一致。这是一条 R 型指令，第一个操作数为一号寄存器中的内容第二个操作数为二号寄存器中的内容，而由于在此之前的前两条指令分别将数据写入一号和二号寄存器，所以在该条指令这里发生了数据冒险，所以这个时候根据波形图可以看出 forwardA 取值

为 1, forwardB 取值为 2, 由前面的控制信号表可以知到, 第一个 ALU 操作数由上一个 ALU 运算结果旁路获得, 第二个 ALU 操作数由数据存储器或前面的 ALU 结果旁路获得, 再根据波形图中的 ALUA 和 ALUB 的数值一个为 5 另一个为 -8 可以知道答案正确。

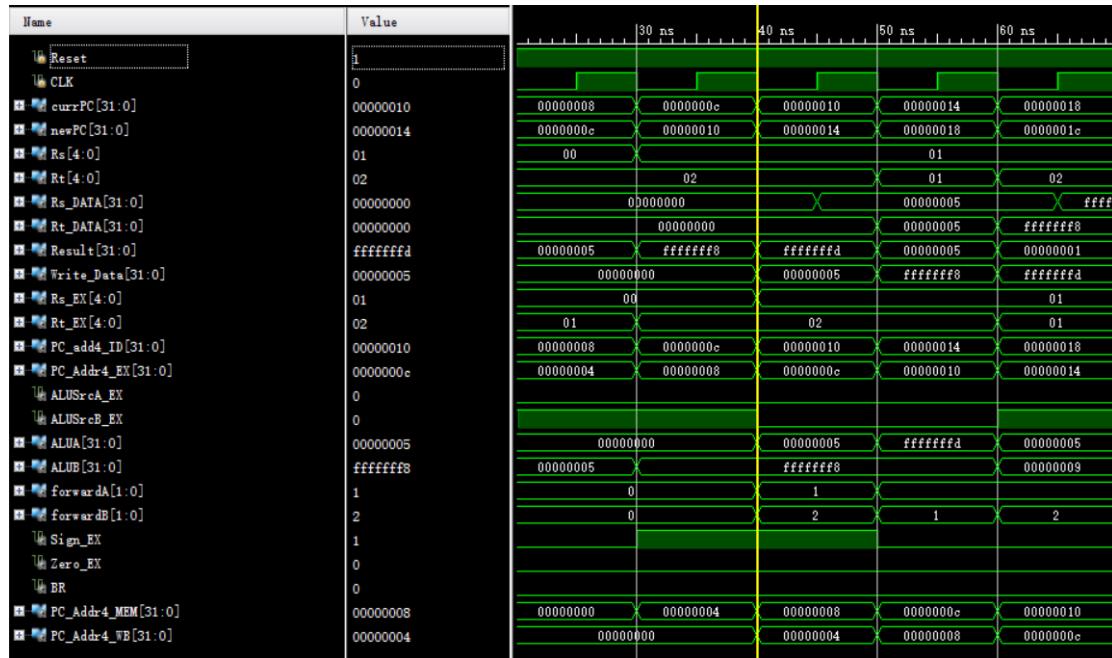


图 17 add \$1,\$1,\$2

(3) sub \$1, \$1, \$2

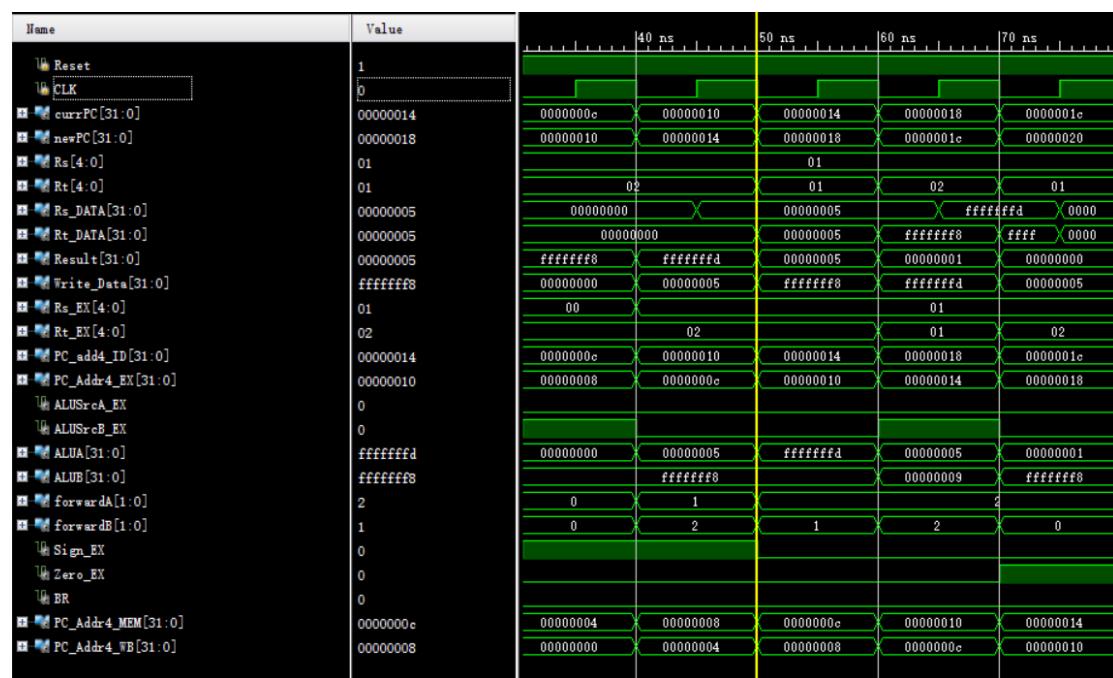


图 18 sub \$1, \$1, \$2

这一条指令为 sub \$1,\$1,\$2，操作是将一号寄存器中的数值和二号寄存器中的数值相减后的结果保存到一号寄存器中，根据波形图我们可以看出此时 EX 段的 RS 寄存器和 RT 寄存器分别为 1 和 2，与指令内容相符。由于这条指令也有数据冒险，由 forwardA 和 forwardB 的值可以看出，第二个 ALU 操作数由上一个 ALU 运算结果旁路获得，第一个 ALU 操作数由数据存储器或前面的 ALU 结果旁路获得。根据 ALUA 和 ALUB 的值可以看出两个操作数分别为 -3 和 -8，结果正确，说明旁路转发机制正常。再根据波形图中的 ALU 运算结果为 5，运算正确。

(4)、andi \$1,\$1,9

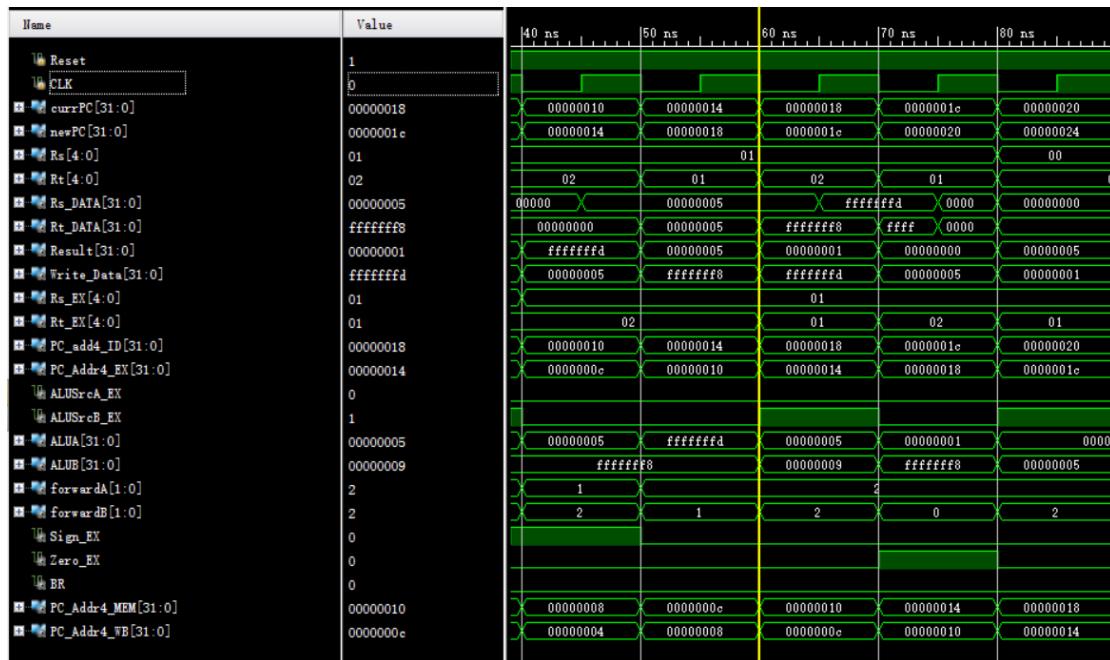


图 19 andi \$1,\$1,9

这一条指令为 andi \$1,\$1,9，操作是将一号寄存器和常数九相加，结果保存在一号寄存器中。这是一条 I 型指令，操作数分别为一号寄存器中的内容和常数 9，根据 ALUSrcA 和 ALUSrcB 分别为 0 和 1，我们可以知道操作数一个来自寄存器一个来自立即数。由于这一条指令也存在数据冒险，所以，ForwardA 为 2，数据来自上一条指令 ALU 旁路运算结果。这里虽然 forwardB 也为 2，但是只有在 ALUSrcB 为零时我们才会用到 forwardB 的数值，而这里 ALUSrcB 为 1，所以 forwardB 的数值是不重要的，也就是我们不 care。根据波形图可以看出两个操作数分别为 5 和 9，运算结果为 1，指令正确。

(5)、and \$1, \$1, \$2

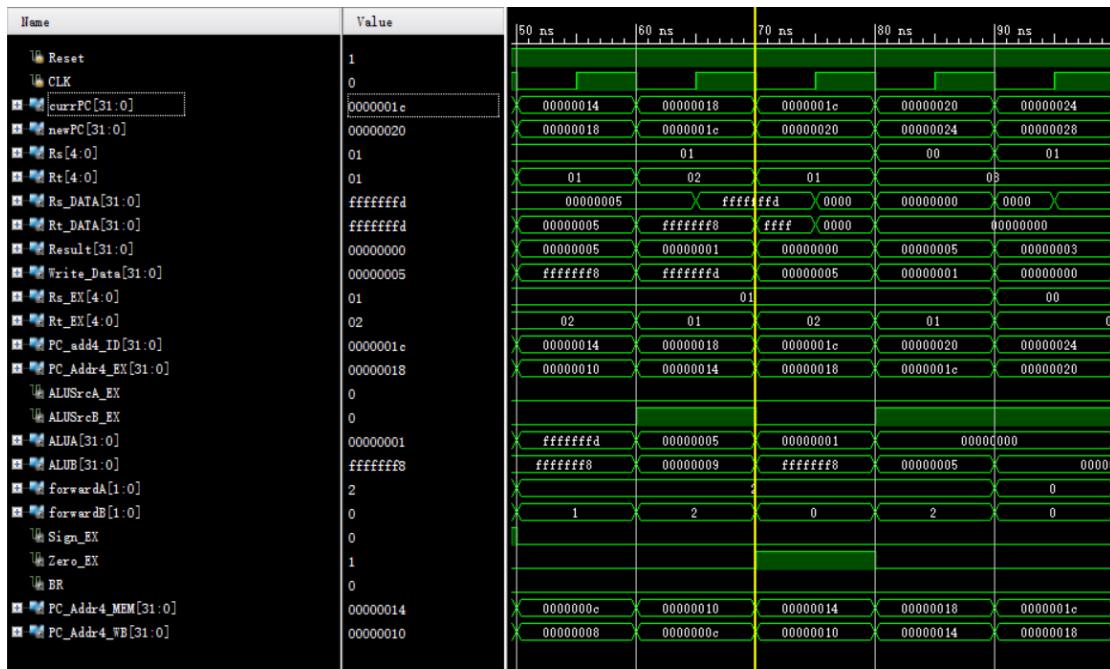


图 20 and \$1, \$1, \$2

这一条指令是将一号寄存器和二号寄存器的内容相与，之后将结果保存到一号寄存器中去。由于前面的指令，这里一号寄存器仍然存在数据冒险，而二号寄存器不存在数据冒险，所以这里 forwardA 为 2，ALU 操作数来自上一条指令 ALU 的旁路，而 forwardB 为 0，ALU 操作数直接来自二号寄存器。根据波形图中的 RS\_EX 和 RT\_EX，可以看出处于 EX 阶段的指令两个源寄存器分别为一号和二号，符合逻辑，而两个操作数分别为 1 和 -8，也符合逻辑，运算结果为 0，符合逻辑。所以指令正确。

(6)、ori \$1, \$1, 5

这条指令的内容为或运算，即将一号寄存器中的内容和立即数 5 进行或操作。我们可以看到在 Rs\_EX 取值为 1，说明译码结果正确。由于这条指令也存在数据冒险，所以 forwardA 为 2，表示操作数来自上一条指令的 ALU 运算结果。ALUA 和 ALUB 分别为 0 和 5，符合指令逻辑，运算结果为 5，逻辑正确。所以此条指令的逻辑正确。

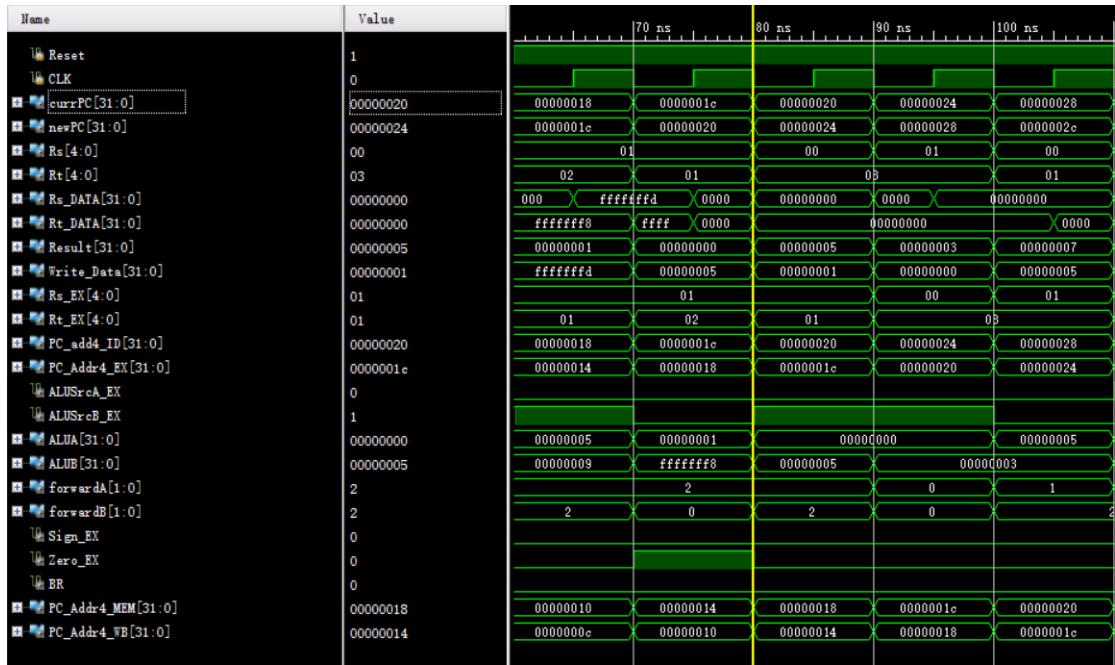


图 21 ori \$1, \$1, \$5

(7)、or \$1, \$1, \$3

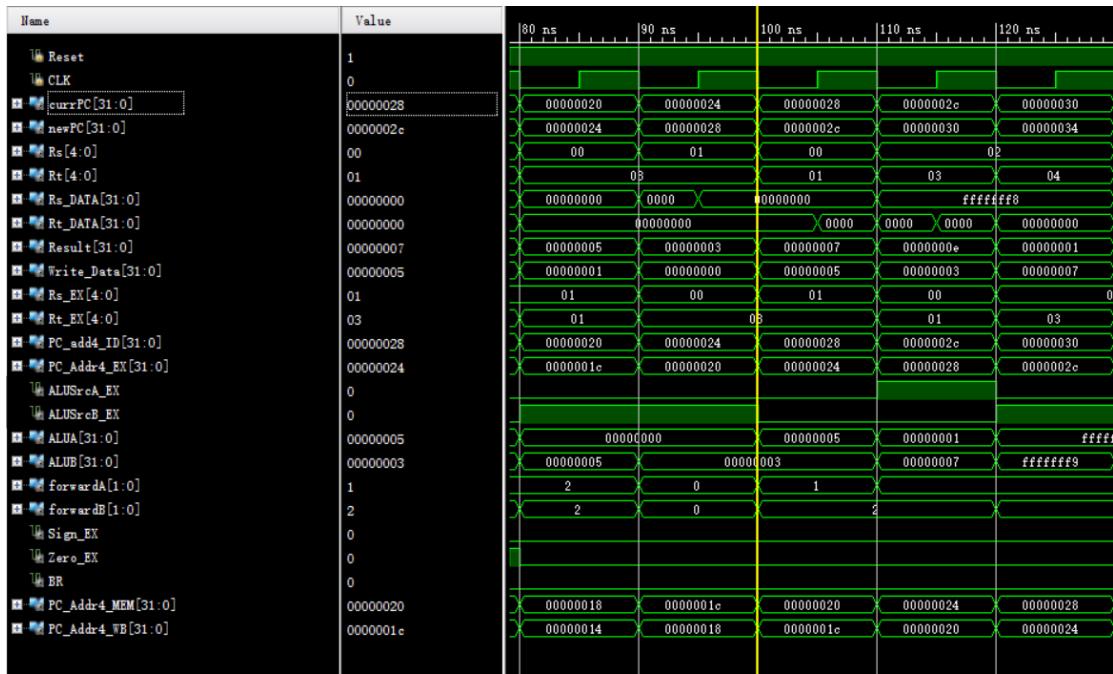


图 22 or \$1, \$1, \$3

此条指令的含义为将一号寄存器中的内容和三号寄存器中的内容或，运算结果保存在一号寄存器中。这条指令同样存在数据冒险，所以这个时候根据波形图可以看出 forwardA 取值为 1，forwardB 取值为 2，由前面的控制信号表可以知道，第一个 ALU 操作数由上一个 ALU 运算结果旁路获得，第二个 ALU 操作数由数据存储器或前面的 ALU 结果旁路获得。观察到 Rs\_EX 和 Rt\_EX 的值分别为 1 和 3，说明处于执行

阶段的指令的两个源操作数分别为一号寄存器和三号寄存器中的数据，两个数据相或后，运算结果为 7，观察到波形图中 result 结果为 7，所以逻辑正确。

(8)、sll \$1, \$1, 1

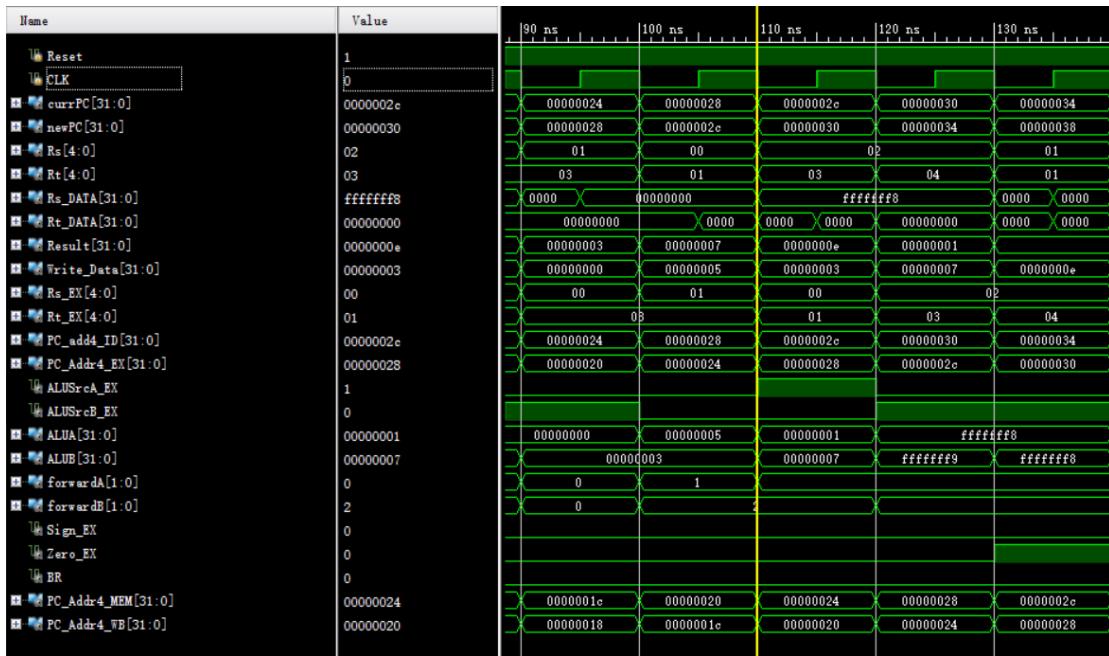


图 23 sll \$1, \$1, 1

这一条指令的含义是移位运算，即将一号寄存器中的内容左移一位，结果保存在一号寄存器中，这一条指令仍然存在数据冒险，我们可以看见 forwardB 的数值为 2，说明运算结果来自 ALU 的旁路，所以这条指令即为将数字 7 的二进制表示左移一位，即相当于将数字 7 扩大二倍，所以运算结果应为 14，根据波形图我们可以看出运算结果为 14，符合运算结果。

(9)、slti \$3, \$2, -7

这一条指令的含义是将二号寄存器中的内容和-7 比较，如果小于则将三号寄存器置一，否则置零。这一条指令没有数据冒险，我们可以看见 Rs\_EX 和 Rt\_EX 的内容分别为 2 和 3. 两个操作数的取值分别为-8 和-7，由于 $-8 < -7$ ，所以三号寄存器中的数据也应该置一，我们可以看见 result 中的内容为 1，说明运算结果正确。再根据后续周期中的数据可以看出各个时期的数据通路取值都正确，所以运算结果正确。

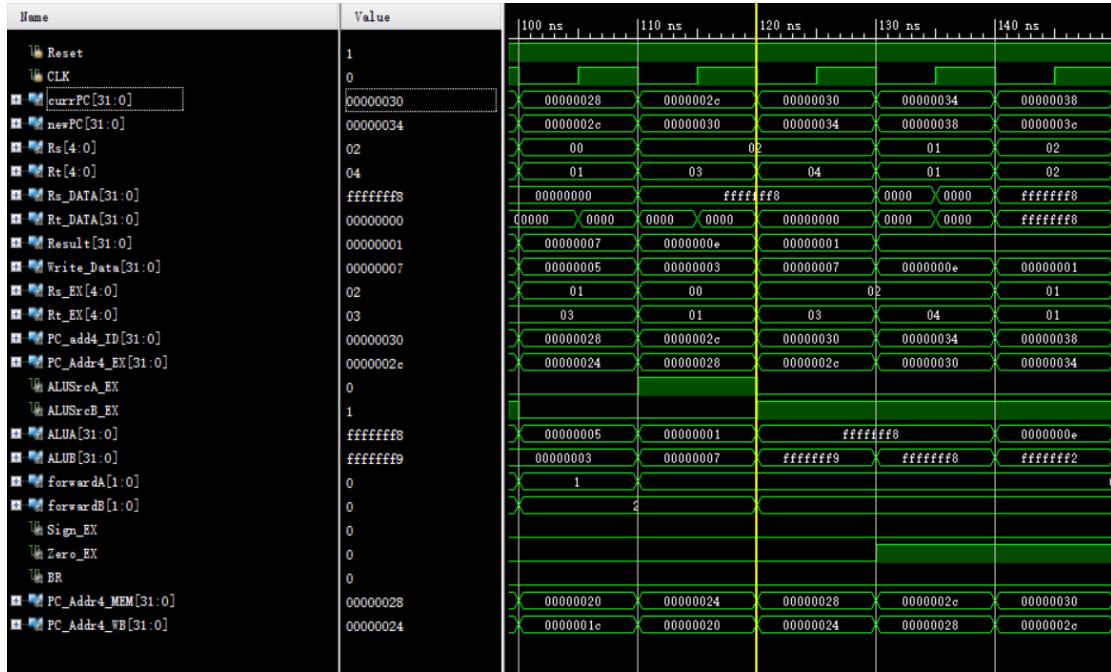
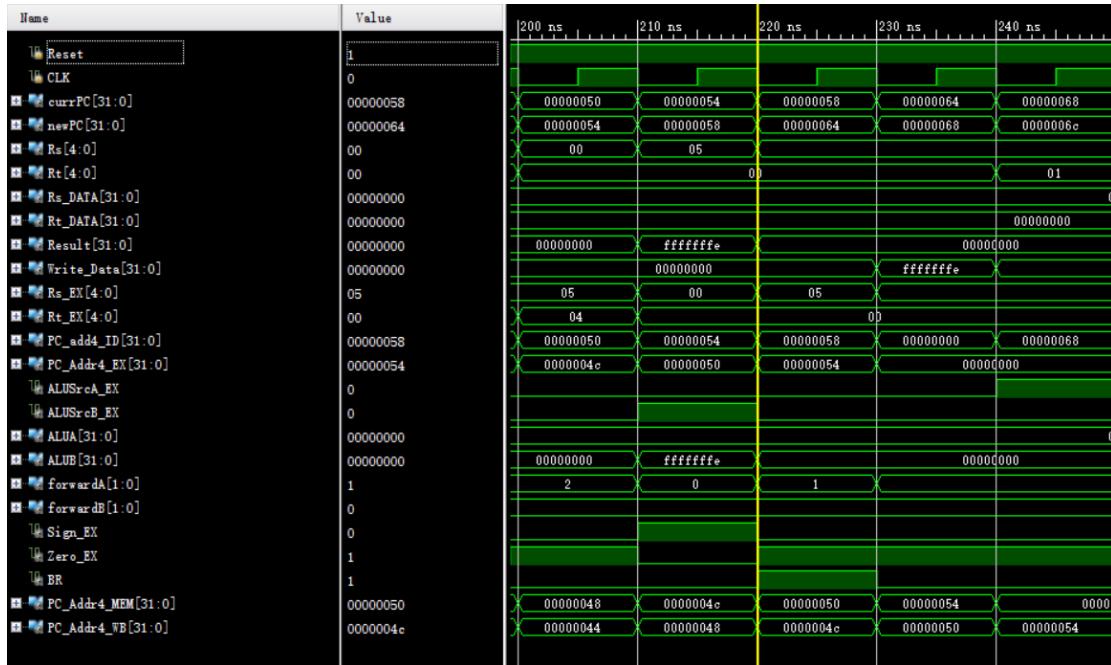


图 24 sli \$3, \$2, -7

(10)、`beq $5,$0,1`图 25 `beq $5,$0,1`

这条指令的含义是将零号寄存器中的数据和五号寄存器中的数据比较，如果相等就跳转到当前  $PC + 4 + immediate * 4$  指令地址处。由于五号寄存器和零号寄存器中的数据都是零，所以相等，发生跳转，我们可以看见 `Br` 信号此时为一，所以此时信号正确。由于在这个 CPU 的设计过程中，没能实现 stall，所以在实

际的指令序列中插入了 nop，在这条指令的之后几条指令都为 nop 指令来解决控制冒险。

(11)、sw \$2, 0(\$1)

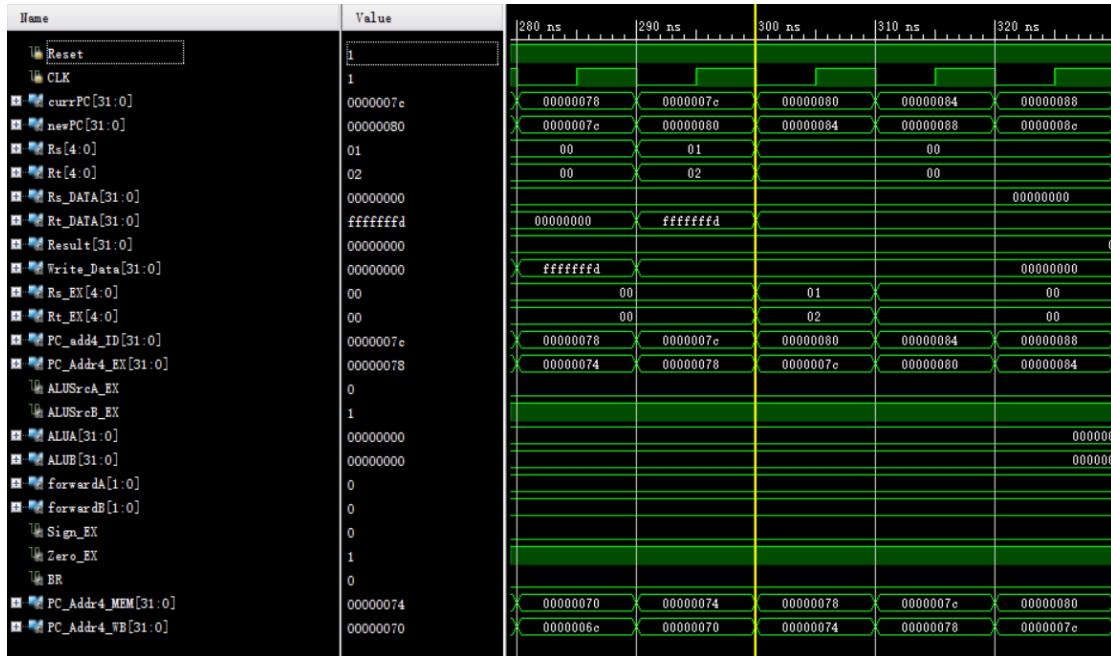


图 26 sw \$2,0(1)

这条指令的含义是将二号寄存器中的值保存在数据存储器中一号寄存器中的数值加上 0 之后得到的地址处。由于一号寄存器中的数据为 0，所以相加之后结果为 0，可以看见 result 结果为 0，运算正确。在之后的时钟周期中，可以看见写回的数据也为 0，所以运算结果正确，指令操作正确。

(12)、lw \$1, 0(\$1)

这条指令的含义是在数据寄存器中将一号寄存器中保存的地址数值加 0 后的得到的地址处的数据保存到一号寄存器中。由于一号寄存器中的数据为 0，所以所取到的数据即为数据存储器中零号单元中的数据，由于上一条指令中将二号寄存器中的内容存到了数据存储器中零号的位置，所以此时数据存储器中零号地址处保存的数据为 -3，之后将数据写回一号寄存器，可以看到在这条指令的第五个时钟周期即写回时钟周期处的 Write\_Data 的值为 -3，所以逻辑正确。

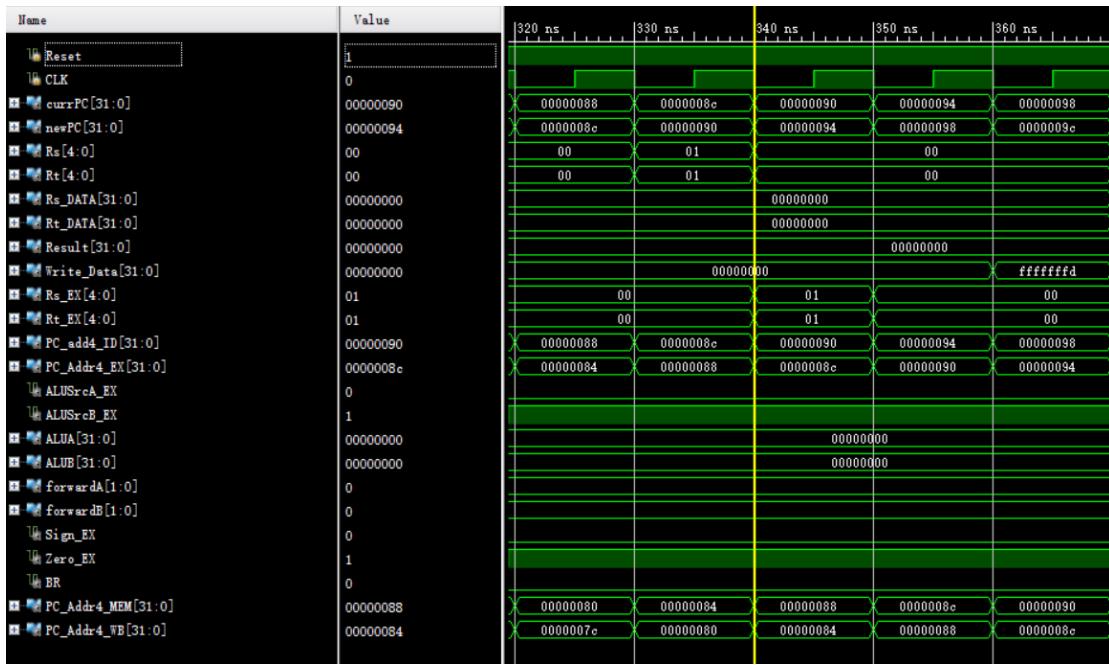


图 27 lw \$1,0(1)

(13)、bne \$1,\$2, 3

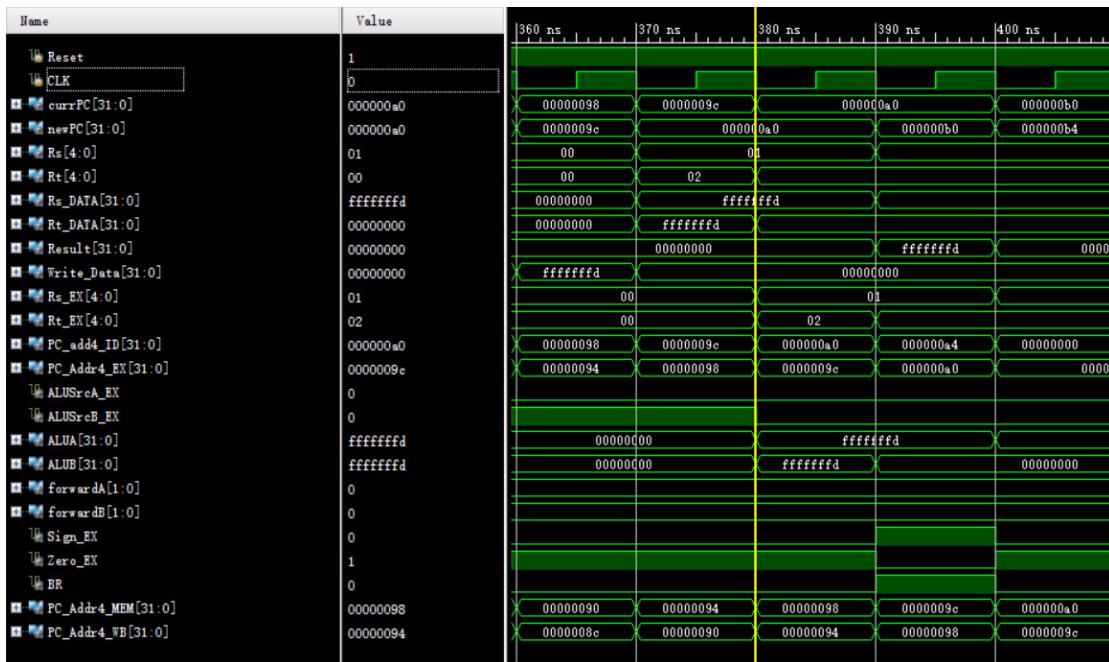


图 28 bne \$1,\$2,3

这一条指令的含义是将一号寄存器和二号寄存器中的数值比较，如果不相等就跳转到当前  $PC + 4 + imm * 4$  的地址处的指令处。由于之前的操作一号寄存器和二号寄存器中的数据都为-3，即一号寄存器和二号寄存器中的数据是相等的，所以此处指令不会发生跳转，可以看见 `zero_EX` 信号为 1，说明相等。`Br` 信号为 0，指令不发生跳转，所以逻辑正确。

(14)、bltz \$1, 1

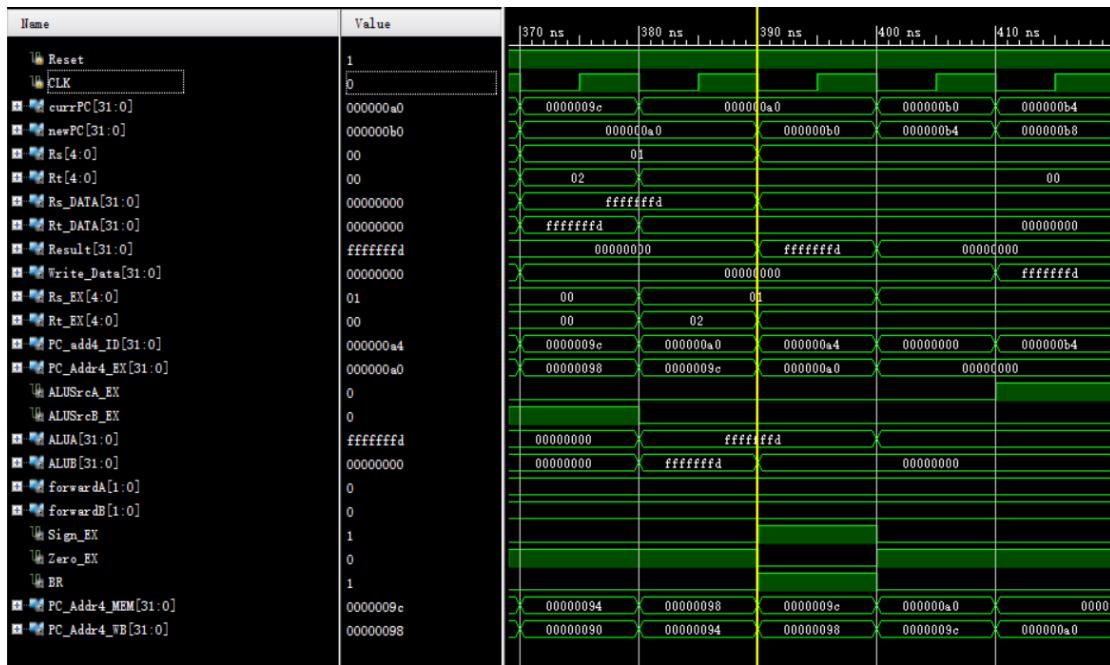


图 29 bltz \$1, 1

这一条指令的含义是将一号寄存器中的内容和零比较，如果一号寄存器中的内容比 0 小，那么就跳转，跳转规则和所有 branch 类型的指令跳转规则一致。由于我们之前就已经知道一号寄存器中的内容为 -3，ALU 所作的操作实际为减法操作，可以看到 sign\_EX 为 1，说明运算结果为负数，所以此时一号寄存器中的数值小于 0，符合逻辑。观察到波形图中 BR 信号取值为 1，所以发生跳转。此条指令逻辑正确。

(15)、j 0x31

这条指令是跳转指令，含义为条件跳转到所给出的地址处。事实上给出的测试指令为 j 0x1f，但是由于在 CPU 的设计中插入了 nop，所以指令的地址有所改变，相应之下此处的跳转地址就应该改为 0x31。根据伪直接寻址的运算方法可知此处跳转的地址应该为 0xc4，根据波形图我们可以看见计算出的跳转地址为 0xc4，符合逻辑。且下一条指令地址，即 newPC 值变为 c4，指令逻辑正确，所以跳转指令逻辑正确。特别一提的都是，因为是流水线 CPU，所以指令在计算出跳转地址的时候，已经预取了两条指令，这时已经发生了控制冒险，但是由于在设计的过程中，没有实现 hazard\_detect 单元，所以没有办法有效解决控制冒险，只能在指令序列里面插入 nop，所以在 jump 指令处于 EX 段时，取入的两条指令都是空指令，即不做任何操作。

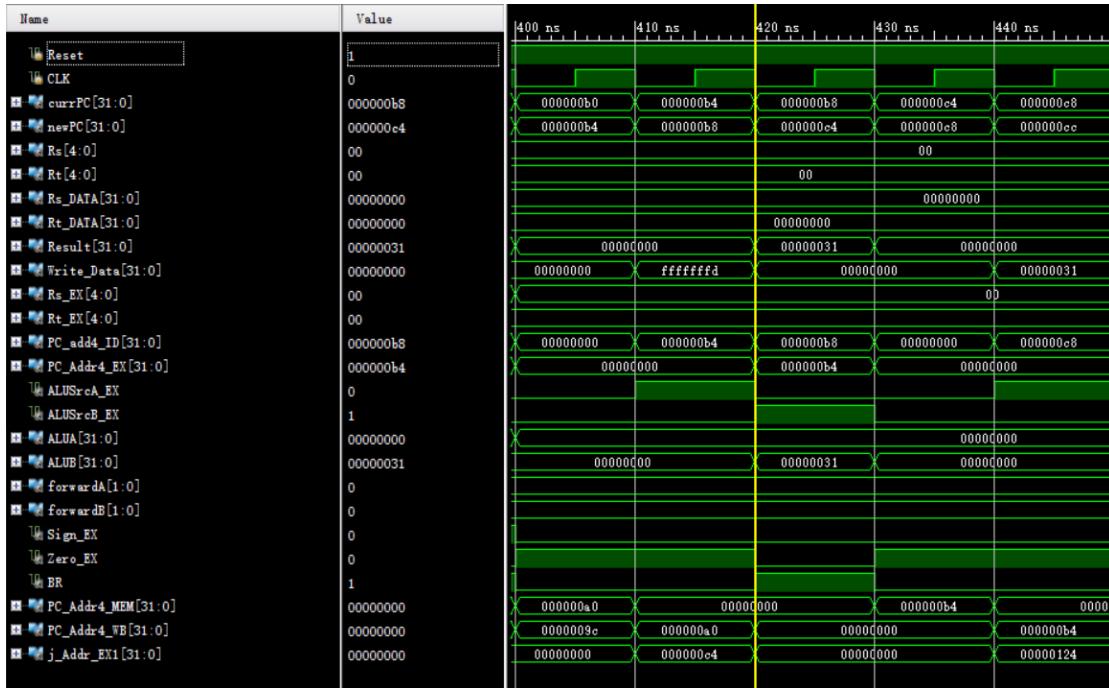


图 30 j 0x31

(16)、jal 0x49

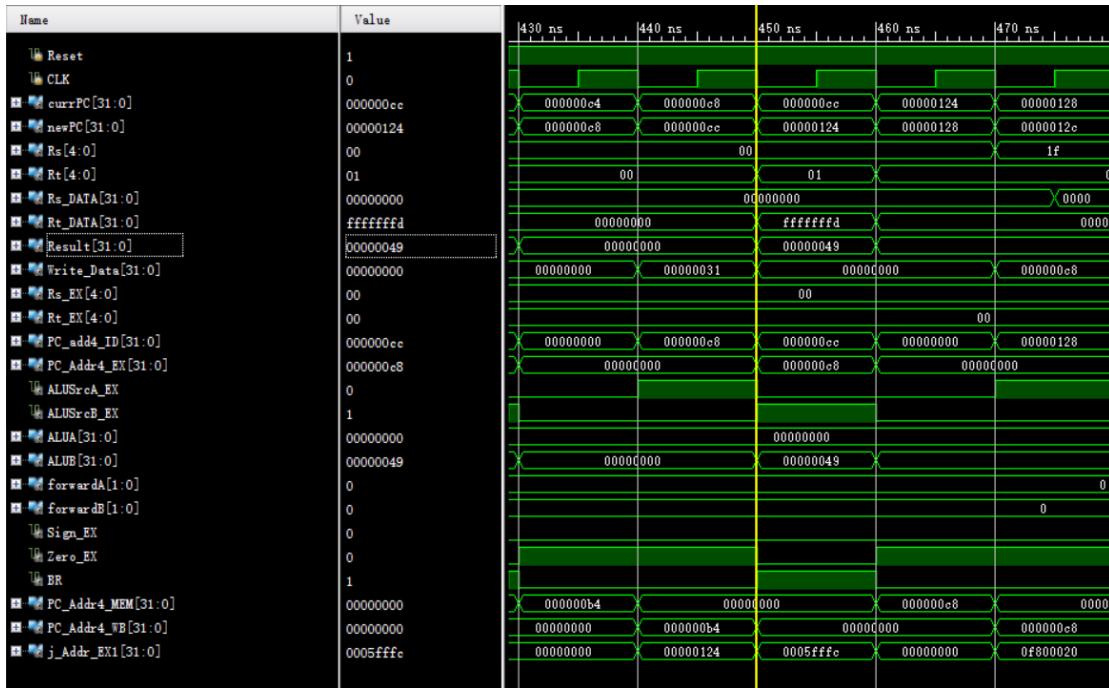


图 31 jal 0x49

这条指令的含义是调用子程序，跳转到指令中直接给出的地址，并将当前 PC+4 的值保存到 31 号寄存器当中。根据伪直接寻址的地址计算规则，可以知道跳转地址应该为 0X124 处，根据波形图可知计算正确。同时要将当前 PC 值+4 后的结果保存到 31 号寄存器当中，根据波形图中第五个时钟周期，也就是这一条

指令的写回周期可以看到写回数据为 0xc8，即当前 PC 值加 4 的结果，所以逻辑正确。

(17)、jr \$31

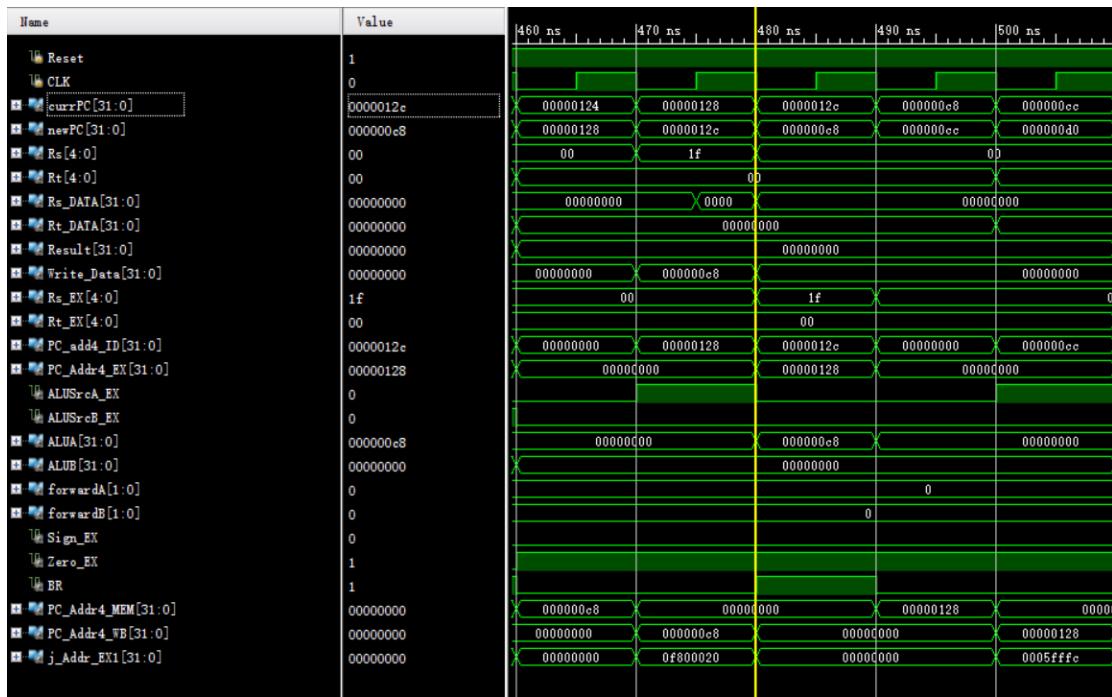


图 32 jr \$31

这条指令的含义是将 PC 值更改为 31 号寄存器中的数据，实际上也是一条跳转指令。这条指令 ALU 所做的操作实际上是加法操作，而另一个操作数恒为 0. 可以看见波形图中两个操作数分别为 0xc8 和 0. 所以指令运算逻辑正确。根据波形图中数据可以看出下一条指令的地址为 0xc8，所以跳转地址也符合逻辑。

(18)、addi \$1,\$1, 32767

这条指令是将一号寄存器的值和指令中给出的立即数相加，相加的结果保存在一号寄存器中。由于此次 CPU 有判溢出的要求，所以如果发生溢出状况需要求运算结果不写回寄存器组。此时一号寄存器中的内容为 0x7fff7fff，再加上 0x7fff 运算结果为 0x7fffffe，一个正数和另一个正数相加此时结果仍然是一个正数，此时不发生溢出。所以结果需要写回寄存器组，根据后面周期里面的写回数据可以看出运算结果写回了寄存器，所以运算正确。

| Name               | Value    | 520 ns   | 530 ns   | 540 ns   | 550 ns   | 560 ns   |
|--------------------|----------|----------|----------|----------|----------|----------|
| Reset              | 1        |          |          |          |          |          |
| CLK                | 0        | 1        | 1        | 1        | 1        | 1        |
| currPC[31:0]       | 000000dc | 00000044 | 000000d8 | 000000dc | 000000e0 | 000000e4 |
| newPC[31:0]        | 000000e0 | 00000048 | 000000dc | 000000e0 | 000000e4 | 000000e8 |
| Rs[4:0]            | 01       |          | 01       |          | 02       |          |
| Rt[4:0]            | 02       | 01       |          | 02       |          | 0        |
| Rs_DATA[31:0]      | 00007fff | fffff000 | fffff000 | fffff000 | fffff000 | fffff000 |
| Rt_DATA[31:0]      | fffff000 | fffff000 | fffff000 | fffff000 | fffff000 | fffff000 |
| Result[31:0]       | 7fffffff | 7ffff000 | 7ffff000 | 7ffff000 | 80000000 |          |
| Write_Data[31:0]   | 7ff00000 | 00000000 | 00007fff | 7ff00000 | 7ff00000 | 7ff00000 |
| Rs_EX[4:0]         | 01       | 00       |          | 01       |          | 0        |
| Rt_EX[4:0]         | 01       |          | 01       |          | 02       |          |
| PC_addr4_ID[31:0]  | 000000dc | 00000044 | 000000d8 | 000000dc | 000000e0 | 000000e4 |
| PC_addr4_EX[31:0]  | 000000d8 | 00000040 | 000000d4 | 000000d8 | 000000dc | 000000e0 |
| ALUSrcA_EX         | 0        |          |          |          |          |          |
| ALUSrcB_EX         | 1        |          |          |          |          |          |
| ALUA[31:0]         | 7ffff000 | 00000010 | 7ff00000 | 7ffff000 | 7ffffffe |          |
| ALUB[31:0]         | 00007fff |          | 00007fff |          | 00000000 | 00000002 |
| forwardA[1:0]      | 2        | 0        |          | 2        |          |          |
| forwardB[1:0]      | 2        |          | 2        |          | 0        | 2        |
| Sign_EX            | 0        |          |          |          |          |          |
| Zero_EX            | 0        |          |          |          |          |          |
| BR                 | 0        |          |          |          |          |          |
| PC_Addr4_MEM[31:0] | 000000d4 | 000000cc | 000000d0 | 00000044 | 000000d8 | 000000dc |
| PC_Addr4_NB[31:0]  | 000000d0 | 00000000 | 000000cc | 000000d0 | 000000d4 | 000000d8 |
| j_addr_EX1[31:0]   | 00880000 | 0085ffff | 00880000 | 01080008 | 01080008 | 010460a8 |

图 32 addi \$1,\$1,32767

| Name               | Value    | 540 ns   | 550 ns   | 560 ns   | 570 ns   | 580 ns   |
|--------------------|----------|----------|----------|----------|----------|----------|
| Reset              | 1        |          |          |          |          |          |
| CLK                | 0        | 1        | 1        | 1        | 1        | 1        |
| currPC[31:0]       | 0000004  | 000000dc | 000000e0 | 0000004  | 000000e8 | 000000ec |
| newPC[31:0]        | 0000008  | 000000e0 | 000000e4 | 000000e8 | 000000ec | 000000f0 |
| Rs[4:0]            | 02       | 01       | 02       | 01       | 02       |          |
| Rt[4:0]            | 01       | 02       | 01       | 02       | 01       |          |
| Rs_DATA[31:0]      | fffff000 | 00007fff | fffff000 | fffff000 | fffff000 | 00000000 |
| Rt_DATA[31:0]      | 7ffff000 | fffff000 | fffff000 | fffff000 | 7ffffffe |          |
| Result[31:0]       | 80000000 | 7ffffffe | 80000000 |          | 00000001 |          |
| Write_Data[31:0]   | 7ff00000 | 7ff00000 | 7ff00000 | 7ff00000 | 7ff00000 | 80000000 |
| Rs_EX[4:0]         | 02       | 01       | 02       | 01       | 02       |          |
| Rt_EX[4:0]         | 02       | 01       | 02       | 01       | 02       |          |
| PC_addr4_ID[31:0]  | 000000e4 | 000000dc | 000000e0 | 0000004  | 0000008  | 000000ec |
| PC_addr4_EX[31:0]  | 000000e0 | 000000d8 | 000000dc | 000000e0 | 0000004  | 000000e8 |
| ALUSrcA_EX         | 0        |          |          |          |          |          |
| ALUSrcB_EX         | 1        |          |          |          |          |          |
| ALUA[31:0]         | 7ffff000 | 7ffff000 | 7ffffffe | fffff000 | fffff000 |          |
| ALUB[31:0]         | 00000002 | 00000000 | 00000002 | 00000002 | 7ffffffe | 00000001 |
| forwardA[1:0]      | 2        | 2        | 0        | 2        |          |          |
| forwardB[1:0]      | 2        | 2        | 0        | 2        |          |          |
| Sign_EX            | 1        |          |          |          |          |          |
| Zero_EX            | 0        |          |          |          |          |          |
| BR                 | 0        |          |          |          |          |          |
| PC_Addr4_MEM[31:0] | 000000dc | 000000d4 | 00000048 | 000000dc | 000000e0 | 000000e4 |
| PC_Addr4_NB[31:0]  | 000000d0 | 000000d0 | 00000044 | 000000d8 | 000000dc | 000000e0 |
| j_addr_EX1[31:0]   | 010460a8 | 00880000 | 01080008 | 010460a8 | 00400004 | 00080008 |

图 32 addi \$2,\$2,2

在执行完上一条指令后，又将一号寄存器中的值赋值到了二号寄存器当中，所以此时二号寄存器中的值也为 0x7ffffffe，此时再将这个数值加 2，两个正数相加结果变成了负数，发生了溢出，所以结果不写回寄存器组，虽然此时显示的写回数据为 0x8000000，但是此时的 REGWRE 的结果为 0，所以数据不会发生写回。指令逻辑正确。

(19)、movn \$1,\$3,\$0

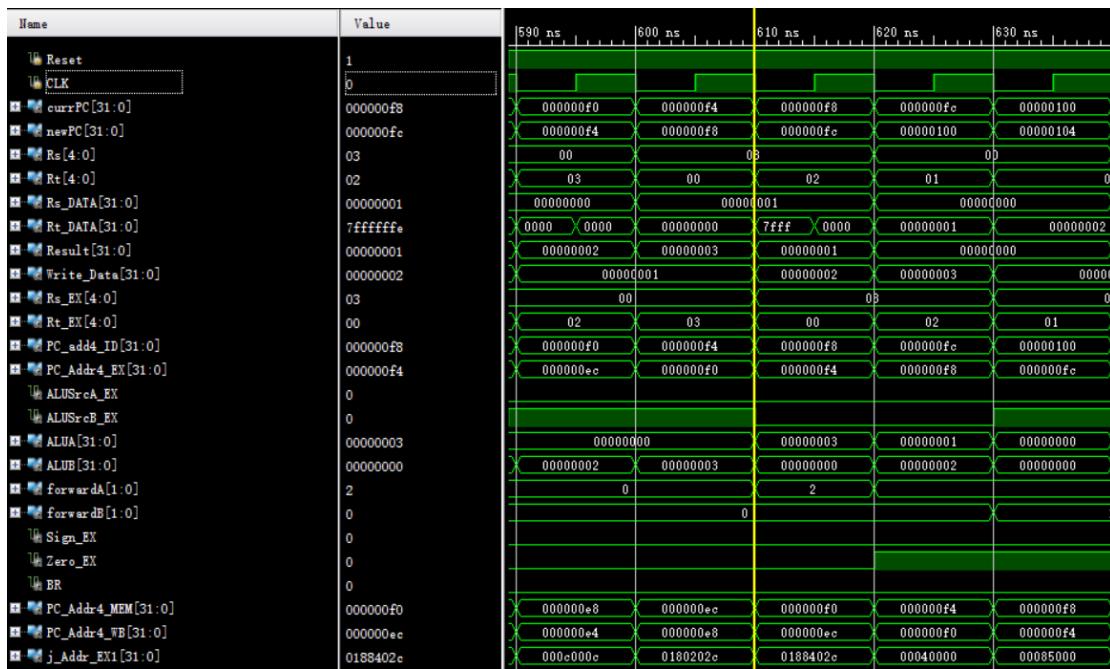


图 33 movn \$1,\$3,\$0

这条指令的含义是如果零号寄存器中的数据不为零，就将三号寄存器中的数据复制到一号寄存器当中。此时 ALU 所做的运算是将寄存器的内容和零比较。由于零号寄存器中的内容永远为 0，所以此时应该将三号寄存器中的数据复制到一号寄存器当中，根据第五个始终周期的写回数据为 0，的值结果正确。

(20)、lhu \$3,10(\$1)

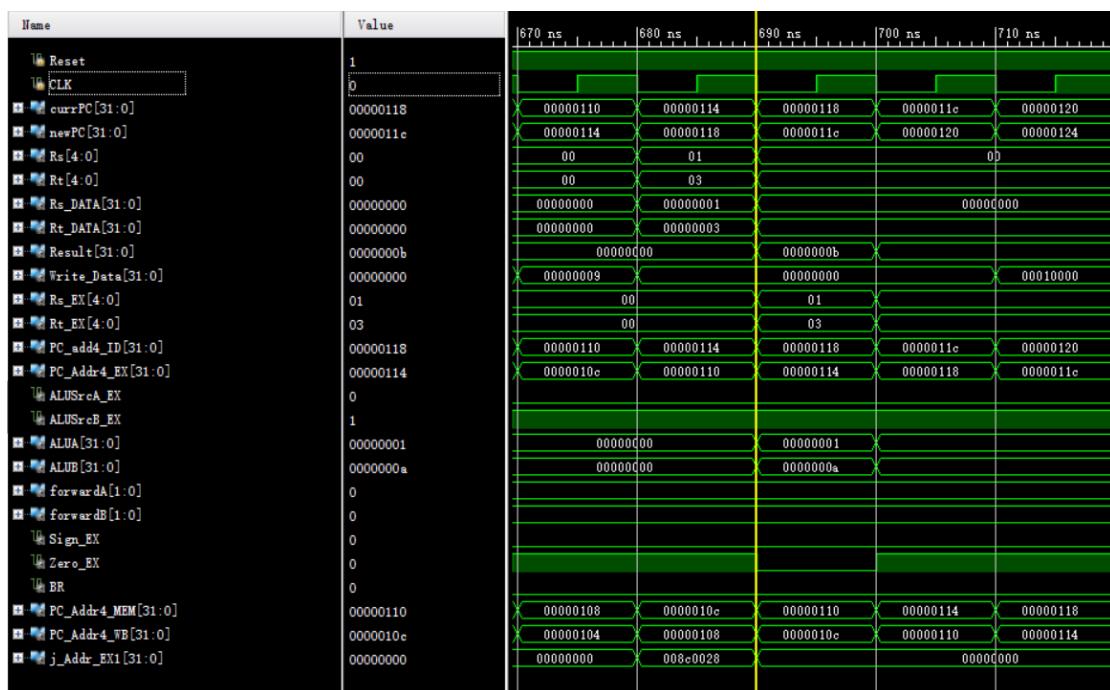


图 34 lhu \$3,10(\$1)

这条指令的含义是将一号寄存器中的地址值偏移是个单位后的结果在数据存储器中寻址，再将该地址处的数据保存到三号寄存器中。由于一号寄存器中的数据为 0，偏移十个地址后得到的地址即为 10，之后在数据存储器中寻址，数据的内容为 0，所以写回到三号寄存器中的内容也就为 0，根据波形图显示可知逻辑正确。

(21)、slt \$3,\$2,\$1

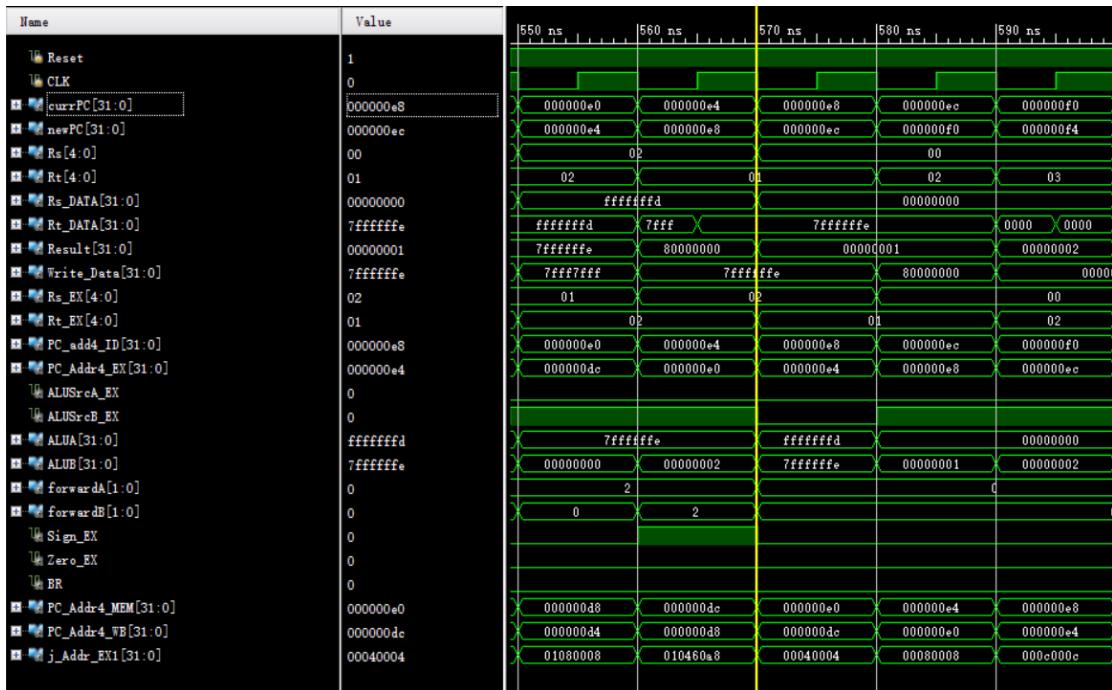


图 35 sll \$3,\$2,\$1

这条指令的含义是将二号寄存器中的数值和一号寄存器中的数值比较，如果小于就将三号寄存器中的数据置 1，否则清零。根据波形图可以看出二号寄存器中的内容为 0，一号寄存器中的内容为 0x7fffffff，所以比较结果为 1，可以看在后面的周期写回的数据也为 1，指令逻辑正确

(22)、halt

这条指令的含义是停机指令，所有的控制信号清零，PC 的值不再更改，根据波形图中的数据可以看出指令执行正确，所有信号清零，PC 值保持不变。值得一提的是，此处的后面我们仍然插入了几条 NOP 指令，因为流水线会向下继续取指令，这样如果后面不再有指令，而 halt 指令还没有到达执行阶段，所有的信号都会变成高阻态，所以此处的后面仍然插入了 nop 以保证指令的正确执行。

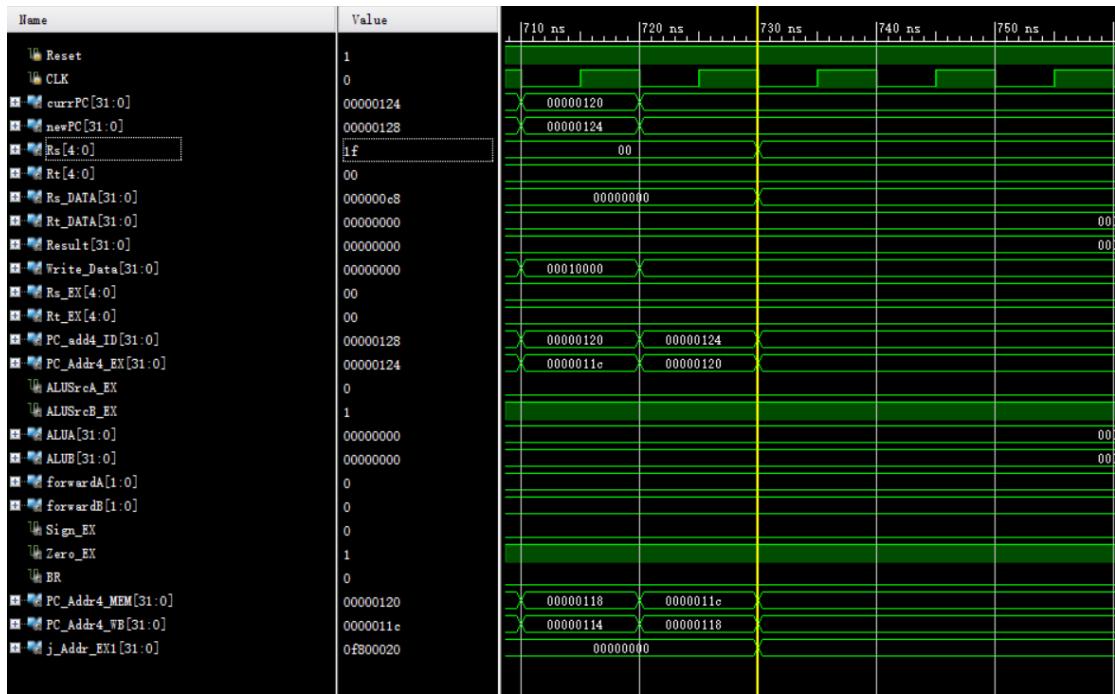


图 36 halt

### (三)、basys3 实验板

本实验的文件组织如下所示

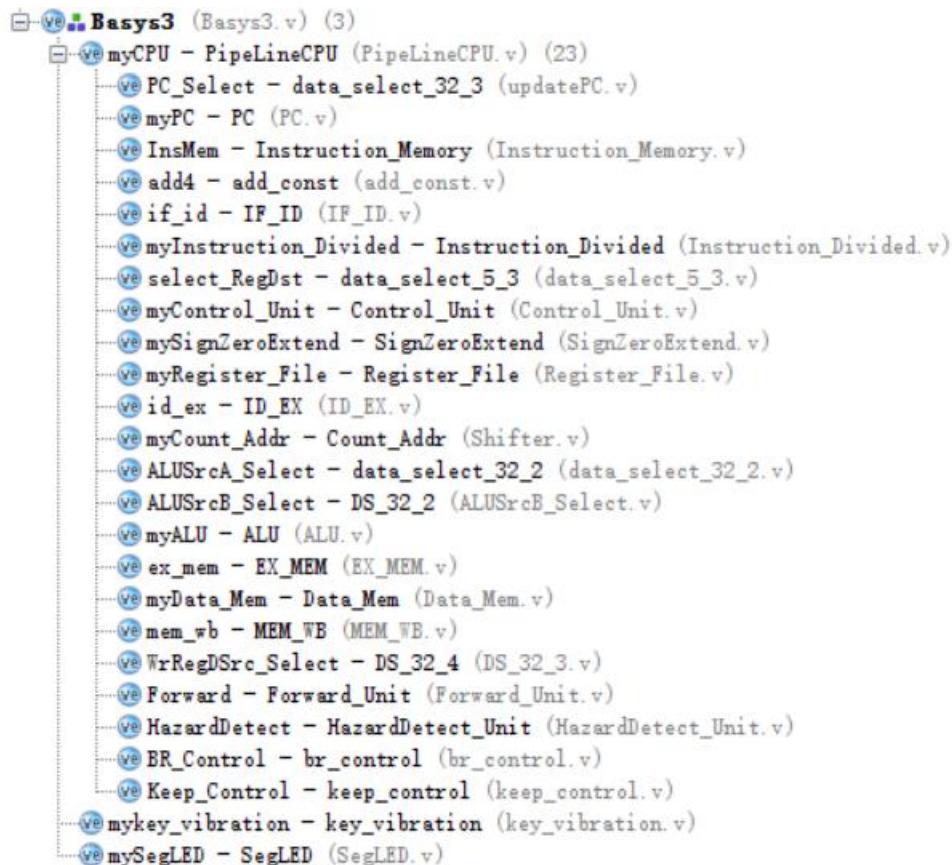


图 37 本实验的文件组织

也就是说在烧板的部分，还单独添加了一个数码管的译码文件和按键消抖的文件，同时在顶层文件中将这几个模块实例化。

那么在烧板的过程中我们需要解决一些问题，其中包括：

- (1)、如何实现数码管的位选
- (2)、如何实现拨动开关控制数据的显示
- (3)、如何实现按键的消抖
- (4)、各个模块之间是如何连接的

我们接下来将分别讨论这四个问题的解决方法。

### 1、如何实现数码管的位选

数码管的显示我们仍然采用上个学期所学习过的扫描式显示。这就需要对时钟进行分频操作，防止时钟频率过高，数码管不能正确显示。同时，由于数码管扫描式显示利用人眼的视觉暂留效果，如果时钟频率过低仍然不能达到同时显示的视觉效果，所以时钟分频要选取一个适当的分频时间。

之前的分频操作都是直接用封装好的分频器 IP，这次自己实现其实原理也很简单。设置一个常量表示 CLK\_DIV 表示分频后得到的每个时钟周期的内部时钟周期数，这里我们选取的数值是 10000。对于位选部分具体的实现，就是每次一个机器时钟的下降沿到来，计数变量就自增，当计数变量和 CLK\_DIV 相等时，就是到了我们理想的分频周期，此时数码管的位选改变。对于数码管位选的改变，只需要一个 case 语句，使得数码管的位选信号顺序变化即可，事实上位选是不需要顺序变化的，只要是每一位都会平均的取到就可以了，不过我们又何必自找麻烦呢～

```
//这一部分是位选
always@(negedge CLK) begin
    if(Reset==1) begin//如果不是清零信号
        cnt <= cnt + 1;//来一个CLK记一次数
        if(cnt==CLK_DIV) begin
            cnt <= 0; //当来到了CLK_DIV这么多的始终的时候，先
            case(AL)
                4'b0111: AL <= 4'b1011;
                4'b1011: AL <= 4'b1101;
                4'b1101: AL <= 4'b1110;
                4'b1110: AL <= 4'b0111;
                default: AL <= 4'b1111;
            endcase
        end
    end
end
```

图 38 位选部分代码

### 2、如何实现拨动开关控制数据的显示

我们使用两个拨动开关来控制四组数据的显示，所以表示拨动开关的变量要设置成两位。在拨动开关控制数据显示这一部分，要注意需要两层 case 语句嵌

套，这两层 case 语句分别控制位选和数据。即在位选选到第 K 位时根据波动开关的取值选取第 K 位应该输出的数据。这一部分拨动开关的显示不是独立自主确定的，是要结合位选所选通的位置的。这里是位选 case 在外层还是拨动开关 case 在外层是没有关系的，我们的程序选择了位选 case 在外层

```

always@( key_en ) begin
    if(Reset==1) begin
        case(A)
            4'b0111:begin
                case(SW_in)
                    2'b00: display_data = Instr_Addr[7:4];
                    2'b01: display_data = {3'b000,rs[4]};
                    2'b10: display_data = {3'b000,rt[4]};
                    2'b11: display_data = result[7:4];
                endcase
            end
            4'b1011:begin
                case(SW_in)
                    2'b00: display_data = Instr_Addr[3:0];
                    2'b01: display_data = rs[3:0];
                    2'b10: display_data = rt[3:0];
                    2'b11: display_data = result[3:0];
                endcase
            end
            4'b1101:begin
                case(SW_in)
                    2'b00: display_data = newPC[7:4];
                    2'b01: display_data = Read_Data1[7:4];
                    2'b10: display_data = Read_Data2[7:4];
                    2'b11: display_data = result[7:4];
                endcase
            end
            4'b1110:begin
                case(SW_in)
                    2'b00: display_data = newPC[3:0];
                    2'b01: display_data = Read_Data1[3:0];
                    2'b10: display_data = Read_Data2[3:0];
                    2'b11: display_data = result[3:0];
                endcase
            end
        endcase
    end

```

图 38 拨动开关控制数据输出部分代码

### 3、按键消抖部分

按键消抖部分其实是为了提高 CPU 的稳定性，经过资料的查询，按键消抖的方法是有很多的，我们这里采用最简单的一种按键消抖方法，就是延时的方式。根据 Basys3 实验板手册，可以知道 Basys3 板子的按键是高电平有效，但是按键按下时电平信号如下所示

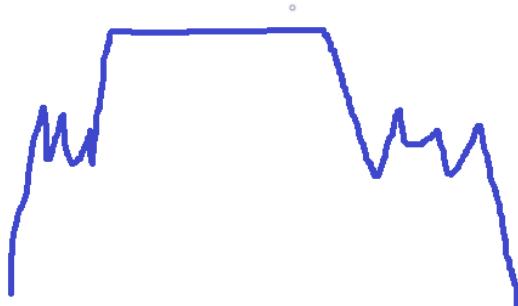


图 39 按下按键时的电平信号

如上所示，在按键刚刚按下时，信号是不稳定的，那么我们用延时的方法，等到信号稳定的时候再选取信号，即如下所示

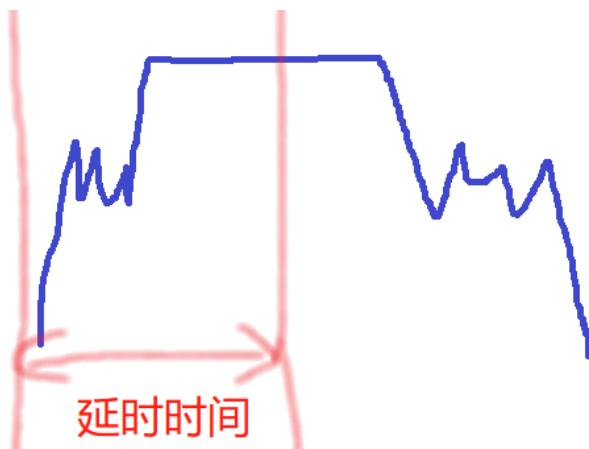


图 39 按键延迟时间

和数码管的位选一样，这个延时时间不能过长也不能过短，如果过短信号还没有稳定就捕捉信号，如果过长可能捕捉不到信号。对于延时的实现思路，就是定义一个 DURATION 的常量，当时钟周期走过 DURATION 这么多的个数时，捕捉信号。这就相当于将信号捕捉时间与按键真正按下的时间延迟了 5000 个时钟周期。在每个时钟到来的时候，如果计数变量和 DURATION - 1 相等，那么就采集键使能信号，否则不采集。具体实现代码如下：

```

always@(posedge CLK)
begin
    if(key==1) begin//表示如果按键按下去了
        if(cnt==DURATION)
            cnt <= cnt;
        else
            cnt <= cnt + 1;
    end
    else
        cnt <= 16'b0;
end

always@(posedge CLK) begin//每个时钟来的时候都判断现在是否键使能了
    if(key) key_en <= (cnt==DURATION-1'b1)?1'b1:1'b0;
end

```

图 40 按键消抖部分主体代码

## 4、各个模块之间的连接

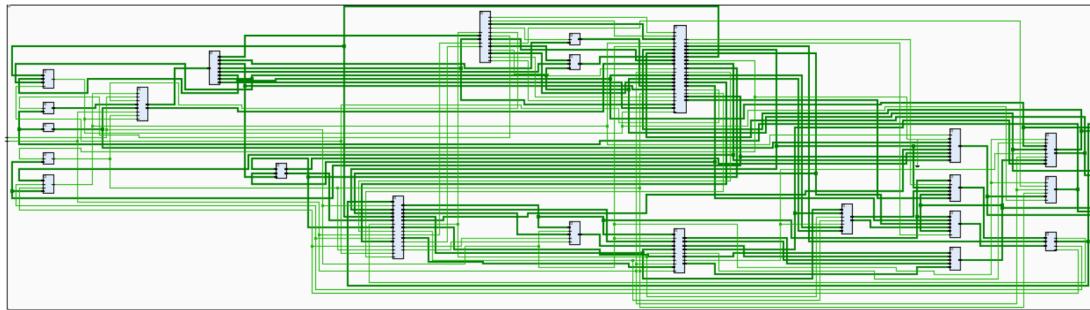


图 41 CPU 内部各个部件的连接

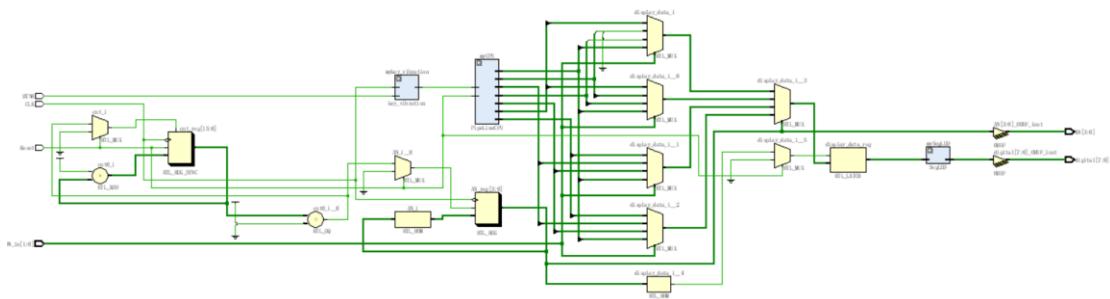


图 41 外部各个部件之间的连接图

在Basys3顶层模块中实例出CPU，按键消抖，数码管译码部分，由于数码管的位选部分在顶层文件中直接实现而没有再设计模块，所以接口连接中存在很多的数据选择器。对于整个CPU的设计，只包含四个输入和两个输出端口。

Basys3上连续的指令(以取指 (IF) 阶段为准):

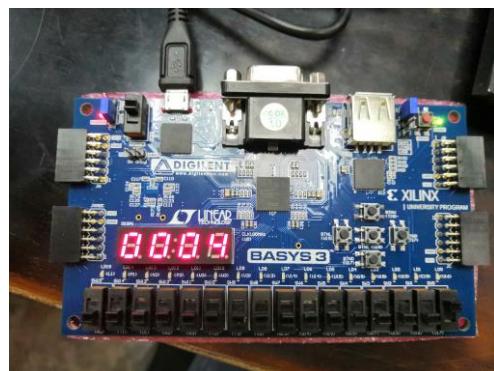


图 42 第一条指令 addiu \$1,\$0,5-> 当前 PC: 下一条 PC

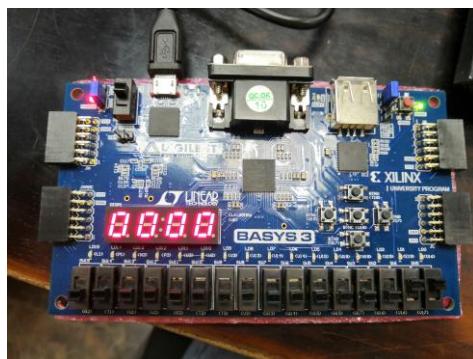


图 42 第一条指令 addiu \$1,\$0,5-> rs 寄存器地址: rs 寄存器数据



图 42 第一条指令 addiu \$1,\$0,5-> rt 寄存器地址: rt 寄存器数据

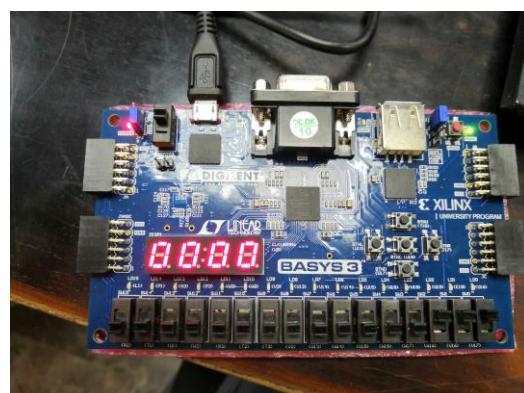


图 43 第一条指令 addiu \$1,\$0,5-> ALU 运算结果: DB 总线数据

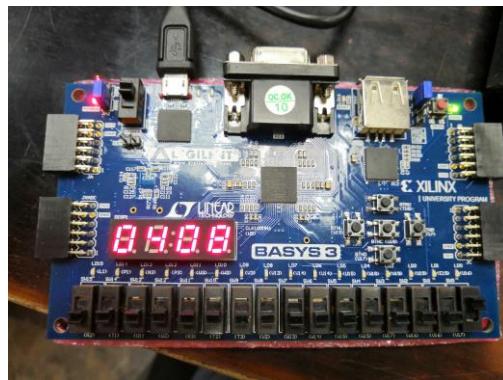


图 44 第二条指令 addiu \$2,\$0,-8-> 当前 PC: 下一条 PC

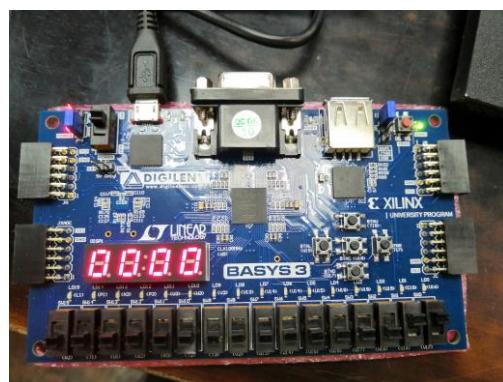


图 45 第二条指令 addiu \$2,\$0,-8-> rs 寄存器地址: rs 寄存器数据

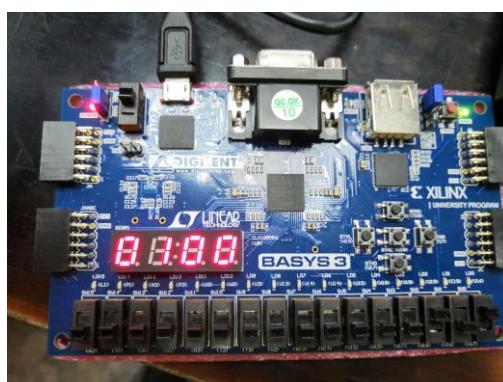


图 46 第二条指令 addiu \$2,\$0,-8-> rt 寄存器地址: rt 寄存器数据

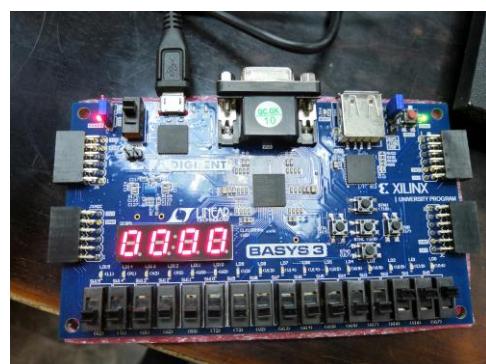


图 47 第二条指令 addiu \$2,\$0,-8->ALU 运算结果: DB 总线数据

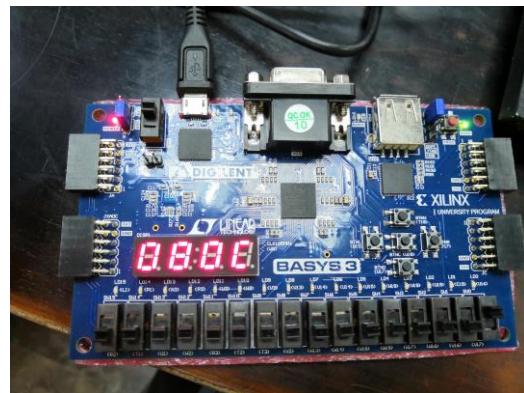


图 48 第三条指令 add \$1,\$1,\$2-> 当前 PC: 下一条 PC

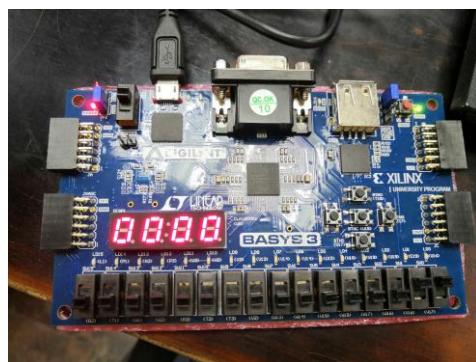


图 49 第三条指令 add \$1,\$1,\$2-> rs 寄存器地址: rs 寄存器数据

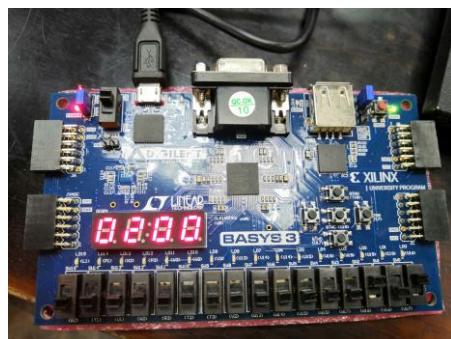


图 50 第三条指令 add \$1,\$1,\$2-> rt 寄存器地址: rt 寄存器数据

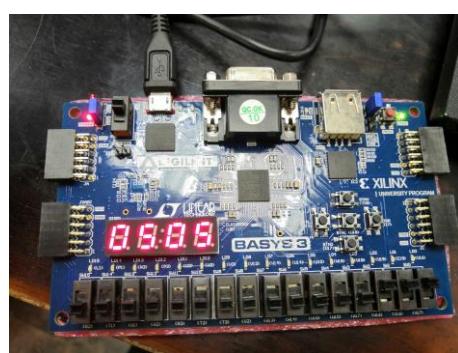


图51 第三条指令add \$1,\$1,\$2->ALU运算结果: DB总线数据

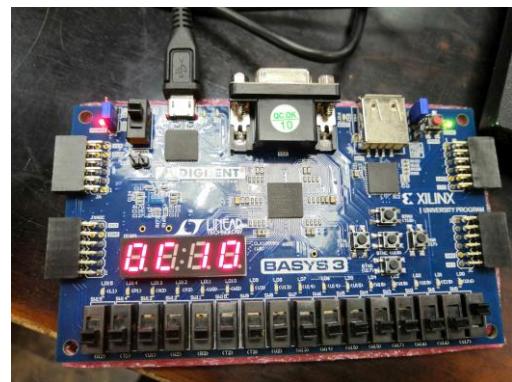
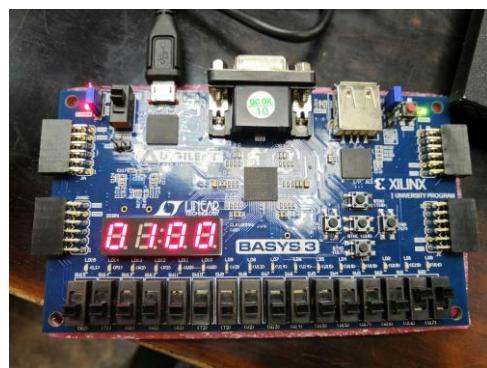
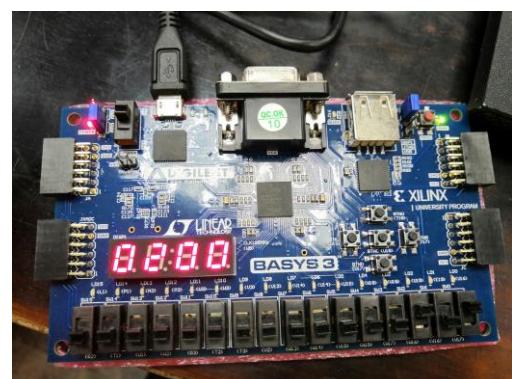


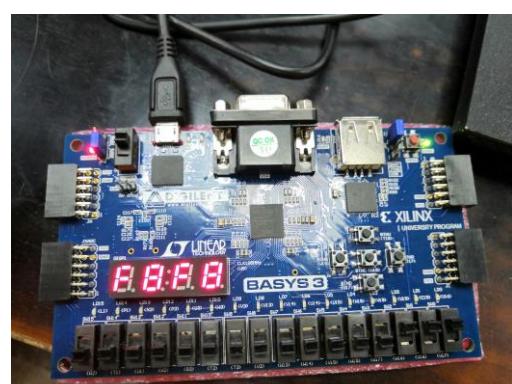
图 52 第四条指令  $sub \$1,\$1,\$2 \rightarrow$  当前 PC: 下一条 PC



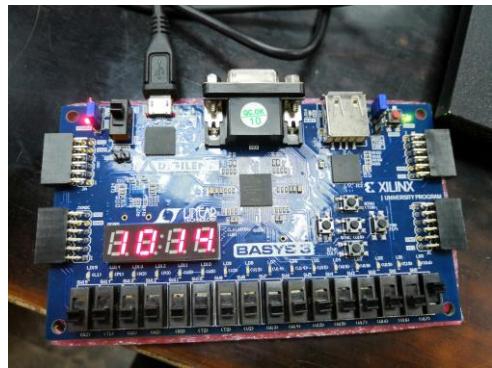
第四条指令  $sub \$1,\$1,\$2 \rightarrow$  rs 寄存器地址: rs 寄存器数据



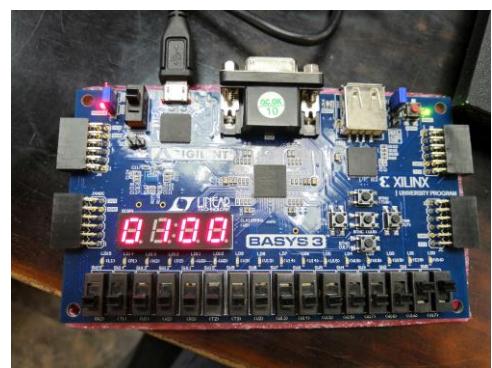
第四条指令  $sub \$1,\$1,\$2 \rightarrow$  rt 寄存器地址: rt 寄存器数据



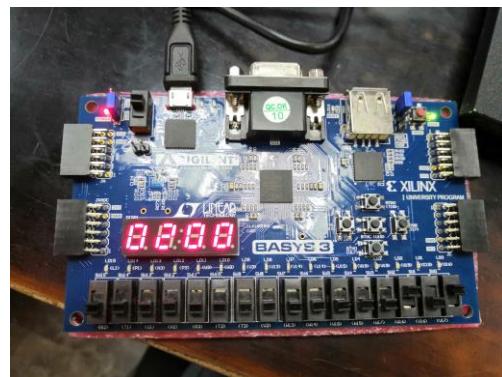
第四条指令  $sub \$1,\$1,\$2 \rightarrow$  ALU 运算结果: DB 总线数据



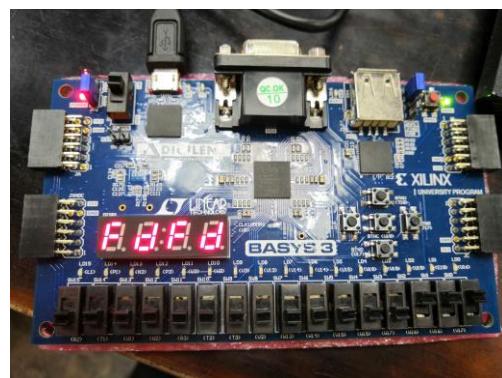
第五条指令 andi \$1,\$1,9-> 当前 PC: 下一条 PC



第五条指令andi \$1,\$1,9-> rs寄存器地址: rs寄存器数据



第五条指令 andi \$1,\$1,9-> rt 寄存器地址: rt 寄存器数据



第五条指令 andi \$1,\$1,9->ALU 运算结果: DB 总线数据

## 六. 实验心得

### (一)、不足

1、对于Control Unit的设计，没能完美地利用控制信号表。许多直接利用控制信号表就可以直接生成并进行控制的信号没有正确地实现，而是新增加了专门的信号判断单元来决定控制信号的生成。使得设计出的CPU有一些冗余的部件，这些其实是不必要的。

2、由于对数据通路理解的不够透彻，在设计过程中流水段寄存器中是传入了很多多余的数据的，这样不仅使整个程序都变得复杂，而且CPU的设计也变得很繁琐。

3、没能解决LW数据冒险，对于插入stall的流水线CPU没能实现。曾经尝试着去实现这一部分，但是由于stall部分信号是由ID段和EX段的数据控制产生的，而stall信号又会保持IF\_ID段寄存器和ID\_EX段寄存器的数据不变，这样就形成了一个死循环，就是说stall信号一旦为一那么将一直为1，这样程序就会在这里卡死，本来希望通过事中控制stall只保持一个时钟周期，但是不知道该如何实现，无奈之下只能通过插入NOP指令。

4、没有解决控制冒险，所以只能选择最简单的在指令中插入nop序列的方式来防止错取指令。这样的结果就是指令序列长度很长。而且中间有大量的nop指令。这是很不合理的。

5、因为插入了nop指令，所以使得整个指令序列很长，所以指令存储器没用使用256个寄存器来模拟，而是使用了512个，对于金贵的寄存器来说，这样大规模的使用是一种浪费。

### (二)、收获

1、在单周期CPU的设计中，ALU部分的sign信号的产生是用result的最低位作为判断的，这是符合逻辑的但是是不符合我们日常的习惯的。这样做出的程序和普遍的指令兼容性是很差的，所以在流水线CPU这里我们做出了改进，sign通过做差后result的最高位产生。这样的设计符合MIPS指令的要求，可以和更多的指令有兼容性。

2、对流水线的实现过程中总结了单周期CPU的经验教训，曾经犯过的问题没有再一次犯错。

3、这一次对阻塞赋值和非阻塞赋值有了一个更加什么的理解，更加能够清楚的体会到什么时候要用阻塞赋值什么时候要用非阻塞赋值。

这学期的所有实验到这里就全部结束啦！计组实验果然是很有成就感的！这次写流水线有了单周期的基础，这次写起来明显就顺手了很多！不过以后也还要继续加油鸭~而且这里还要特别感谢一下刘芳老师和TA们的帮助，真的这学期计组的TA是我见过最好的TA啦~