中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

实验报告

（实验三）

学 院 名 称 ： 数据科学与计算机学院

专业（班级）：

学 生 姓 名 ：

学 号 ：

时 间 ： 年 月 日

成 绩：

实验二 ： 单周期CPU设计与实现

一. 实验目的

1. 掌握单周期CPU数据通路图的构成、原理及其设计方法；
2. 掌握单周期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. 掌握测试单周期CPU的方法。

二. 实验内容

设计一个单周期 CPU，该 CPU 至少能实现以下指令功能操作。指令与格式如下:

==> 算术运算指令

（1）add rd , rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100000

功能：GPR[rd] ← GPR[rs] + GPR[rt]。

（2）sub rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100010

功能：GPR[rd] ← GPR[rs] - GPR[rt]。

（3）addiu rt , rs ,**immediate**

001001	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] + sign_extend(**immediate**)； **immediate** 做符号扩展再参加“与”运算。

==> 逻辑运算指令

（4）andi rt , rs ,**immediate**

001100	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] and zero_extend(**immediate**)；**immediate** 做 0 扩展再参加“与”运算。

（5）and rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100100

功能：GPR[rd] ← GPR[rs] and GPR[rt]。

（6）ori rt , rs ,**immediate**

001101	rs(5 位)	rt(5 位)	**immediate**(16 位)

功能：GPR[rt] ← GPR[rs] or zero_extend(**immediate**)。

（7）or rd , rs , rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	00000 100101

功能：GPR[rd] ← GPR[rs] or GPR[rt]。

==>移位指令

（8）sll　rd, rt,sa

000000	00000	rt(5 位)	rd(5 位)	sa(5 位)	000000

功能：GPR[rd] ← GPR[rt] << sa。

==>比较指令

（9）　slti　rt, rs,**immediate**　　带符号数

001010	rs(5 位)	rt(5 位)	**immediate**(16 位)

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

==> 存储器读/写指令

（10）sw　rt , **offset** (rs) 写存储器

101011	rs(5 位)	rt(5 位)	**offset**(16 位)

功能：memory[GPR[base] + sign_extend(**offset**)] ← GPR[rt]。

（11) lw　rt , **offset** (rs) 读存储器

100011	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：GPR[rt] ← memory[GPR[base] + sign_extend(**offset**)]。

==> 分支指令

（12）beq　rs,rt, **offset**

000100	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] = GPR[rt]) pc←pc + 4 + sign_extend(**offset**)<<2

else pc ←pc + 4

特别说明：**offset 是从 PC+4 地址开始和转移到的指令之间指令条数。**offset 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 **offset** 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

（13）bne　rs,rt, **offset**

000101	rs(5 位)	rt(5 位)	**offset** (16 位)

功能：if(GPR[rs] != GPR[rt]) pc←pc + 4 + sign_extend(**offset**) <<2

else pc ←pc + 4

（14）bltz　rs, **offset**

000001	rs(5 位)	00000	**offset** (16 位)

功能：if(GPR[rs] < 0) pc←pc + 4 + sign_extend (**offset**) <<2

else pc ←pc + 4。

==>跳转指令

（15）j　addr

000010	**addr**(26 位)

功能：PC ← {PC[31:28] , **addr** , 2’b0}，无条件跳转。

说明：由于 MIPS32 的指令代码长度占 4 个字节，所以指令地址二进制数最低 2 位均为 0，将指令地址放进指令代码中时，可省掉！这样，除了最高 6 位操作码外，还有 26 位

可用于存放地址，事实上，可存放 28 位地址，剩下最高 4 位由 pc+4 最高 4 位拼接上。

==> 停机指令

（16）halt

111111	00000000000000000000000000(26 位)

功能：停机；不改变 PC 的值，PC 保持不变。

三、实验原理

单周期 CPU 指的是一条指令的执行在一个时钟周期内完成，然后开始下一条指令的执行，即一条指令用一个时钟周期完成。电平从低到高变化的瞬间称为时钟上升沿，两个相邻时钟上升沿之间的时间间隔称为一个时钟周期。时钟周期一般也称振荡周期（**如果晶振的输出没有经过分频就直接作为 CPU 的工作时钟，则时钟周期就等于振荡周期。若振荡周期经二分频后形成时钟脉冲信号作为 CPU 的工作时钟，这样，时钟周期就是振荡周期的两倍。**）

CPU 在处理指令时，一般需要经过以下几个步骤：

(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。

单周期 CPU，是在一个时钟周期内完成这五个阶段的处理。

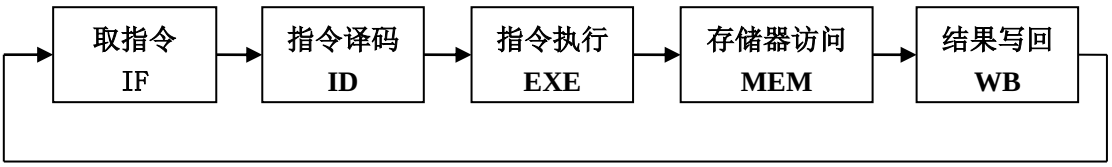
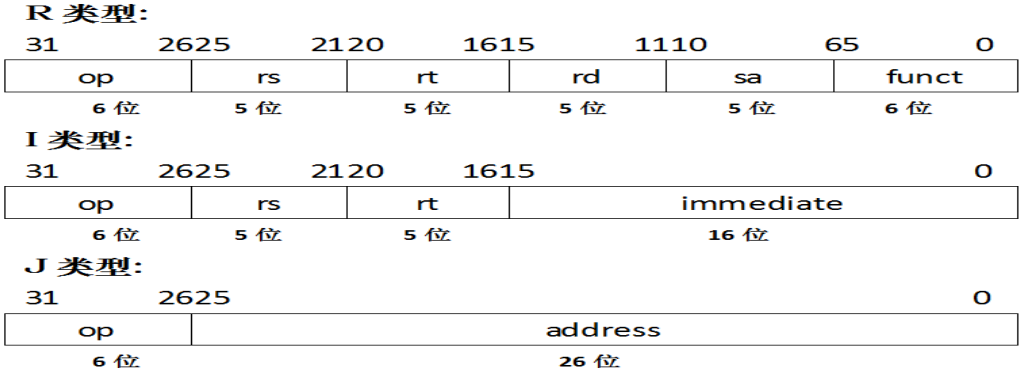


图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：



其中，

op：为操作码；

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

rt：可读可写。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：只写。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与操作码配合使用；

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（Laod）/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。

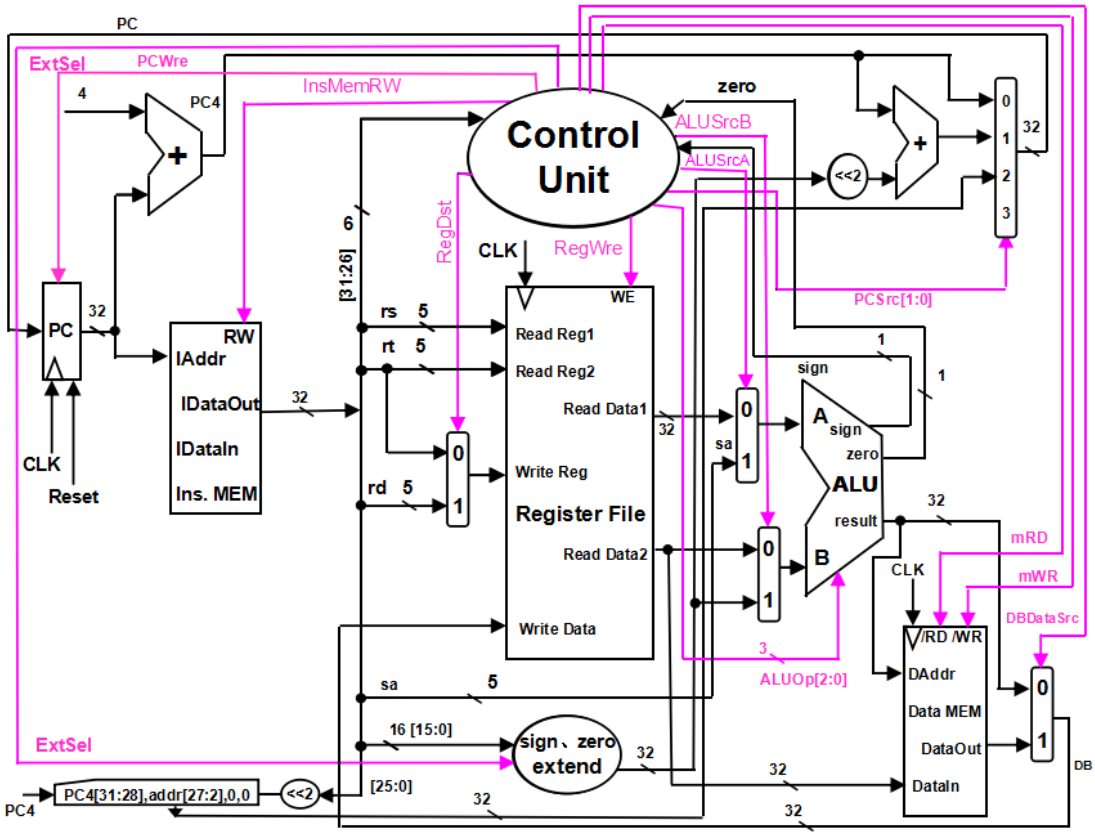


图 2　单周期 CPU 数据通路和控制线路图

图 2 是一个简单的基本上能够在单周期 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。其中指令和数据各存储在不同存储器中，即有指令存储器和数据存储器。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，先给出寄存器地址，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。图中控制信号作用如表 1 所示，表 2 是 ALU 运算功能表。

表 1　控制信号的作用

控制信号名	状态“0”	状态“1”

Reset	初始化 PC 为 0	PC 接收新地址
PCWre	PC 不更改，相关指令：halt	PC 更改，相关指令：除指令 halt 外
ALUSrcA	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	来自移位数 sa，同时，进行(zero-extend)sa，即 {{27{1'b0}},sa}，相关指令：sll
ALUSrcB	来自寄存器堆 data2 输出，相关指令：add、sub、or、and、beq、bne、bltz	来自 sign 或 zero 扩展的立即数，相关指令：addi、andi、ori、slti、sw、lw
DBDataSrc	来自 ALU 运算结果的输出，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll	来自数据存储器（Data MEM）的输出，相关指令：lw
RegWre	无写寄存器组寄存器，相关指令：beq、bne、bltz、sw、halt	寄存器组写使能，相关指令：add、addiu、sub、ori、or、and、andi、slti、sll、lw
InsMemRW	写指令存储器	读指令存储器(Ins. Data)
mRD	输出高阻态	**读数据存储器，相关指令：lw**
mWR	无操作	**写数据存储器，相关指令：sw**
RegDst	写寄存器组寄存器的地址，来自 rt 字段，相关指令：addiu、andi、ori、slti、lw	写寄存器组寄存器的地址，来自 rd 字段，相关指令：add、sub、and、or、sll
ExtSel	(zero-extend)**immediate**(0 扩展)，相关指令：addiu、andi、ori	(sign-extend)**immediate** (符号扩展)，相关指令：slti、sw、lw、beq、bne、bltz
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：未用	
ALUOp[2..0]	ALU 8 种运算功能选择(000-111)，看功能表	

相关部件及引脚说明：

Instruction Memory：**指令存储器，**

Iaddr，指令存储器地址输入端口

IDataIn，指令存储器数据输入端口（指令代码输入端口）

IDataOut，指令存储器数据输出端口（指令代码输出端口）

RW，指令存储器读写控制信号，为 0 写，为 1 读

Data Memory：**数据存储器，**

Daddr，数据存储器地址输入端口

DataIn，数据存储器数据输入端口

DataOut，数据存储器数据输出端口

/RD，数据存储器读控制信号，为 0 读

/WR，数据存储器写控制信号，为 0 写

Register File：寄存器组

Read Reg1，rs 寄存器地址输入端口

Read Reg2，rt 寄存器地址输入端口

Write Reg，将数据写入的寄存器端口，其地址来源 rt 或 rd 字段

Write Data，写入寄存器的数据输入端口

Read Data1，rs 寄存器数据输出端口

Read Data2，rt 寄存器数据输出端口

WE，写使能信号，为 1 时，在时钟边沿触发写入

ALU：　算术逻辑单元

result，ALU 运算结果

zero，运算结果标志，结果为 0，则 zero=1；否则 zero=0

sign，运算结果标志，结果最高位为 0，则 sign=0，正数；否则，sign=1，负数

表 2 ALU 运算功能表

ALUOp[2..0]	功能	描述
000	Y = A + B	加
001	Y = A － B	减
010	Y = B<<A	B 左移 A 位
011	Y = A ∨ B	或
100	Y = A ∧ B	与
101	Y= (A<B) ?1: 0	比较 A 与 B 不带符号
110	Y=(((rega<regb) && (rega[31] == regb[31])) ||((rega[31] ==1 && regb[31] == 0))) ? 1:0	比较 A 与 B 带符号
111	Y = A ⊕ B	异或

需要说明的是以上数据通路图是根据要实现的指令功能的要求画出来的，同时，还必须确定 ALU 的运算功能(当然，以上指令没有完全用到提供的 ALU 所有功能，但至少必须能实现以上指令功能操作)。从数据通路图上可以看出控制单元部分需要产生各种控制信号，当然，也有些信号必须要传送给控制单元。从指令功能要求和数据通路图的关系得出以上表 1，这样，从表 1 可以看出各控制信号与相应指令之间的相互关系，根据这种关系就可以得出控制信号与指令之间的关系表（留给学生完成），再根据关系表可以写出各控制信号的逻辑表达式，这样控制单元部分就可实现了。

四、实验器材

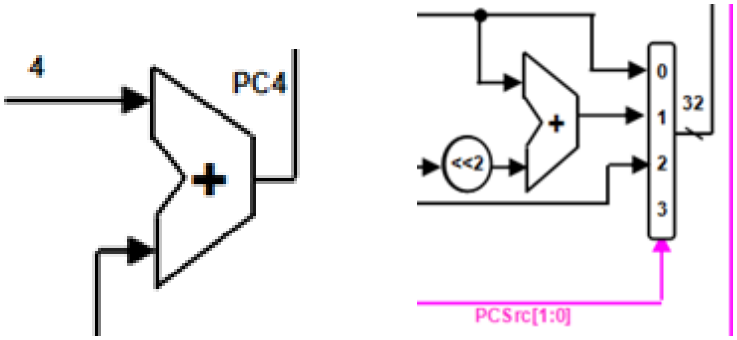
电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五、实验过程与结果

（一）CPU 设计思想与方法

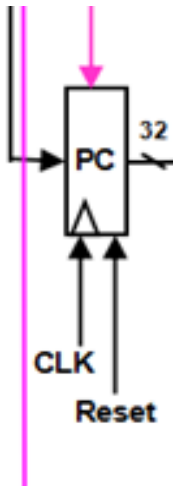
根据图 2 单周期 CPU 数据通路和控制线路图,我们可以把 CPU 分成 12 个模块:CPU，PC、ALU、DataMem、InstrucMem、ControlUnit、RegFile、SignExtend、mux21R、mux21A、mux21B、mux21RW。其中 CPU 为顶层模块。采用至上而下的设计方法。以下对各个模块进行简单说明:

1.CPU：此模块作为顶层模块，将调用另外的模块并将它们的输入输出连接起来，此外还将根据 2 位控制信号 PCSrc 的值完成下一条指令地址的计算（PC+4 或者跳转）。上述关键部分的代码实现如下，



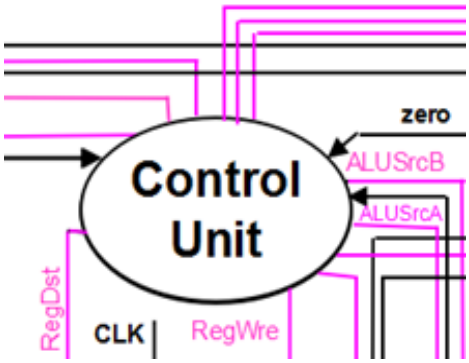

```
assign newAddress = (PCSrc == 2'b01)?{CurPC_4[31:28],instcode[25:0],2'b00}:(PCSrc == 2'b10)?CurPC_4+(Ext_Imm<<2):CurPC_4;//PC更新选择
PC pc(clk,PCWre,Reset,newAddress,CurPC);
ALU alu(ALUOp,ALU_inA,ALU_inB,zero,ALU_res,sign);
Datamem dm(ALU_res,clk,RD,WR,Reg2Value,MemOut);
SignExtend se(instcode[15:0],ExtSel,Ext_Imm);
Mux5 mux21R(RegDst,instcode[20:16],instcode[15:11],WriteRegAddr);
Mux32 mux21A(ALUSrcA,Reg1Value,{27'b000000000000000000000000000,instcode[10:6]},ALU_inA);
Mux32 mux21B(ALUSrcB,Reg2Value,Ext_Imm,ALU_inB);
Mux32 mux21RW(DBDataSrc,ALU_res,MemOut,WriteData);
RegFile rgf(RegWre,clk,instcode[25:21],instcode[20:16],WriteRegAddr,WriteData,Reg1Value,Reg2Value);
ControlUnit cu(PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, RD, WR, RegDst, ExtSel, PCSrc,ALUOp,zero,sign,instcode[31:26]);
InstrucMem im(CurPC,InsMemRW,instcode);
```

2.PC：PC 模块即程序计数器模块，如下图，是一个 32 位寄存器，其功能是读入下一条指令的 32 位地址并输出，即完成指令地址的更新并且存放当前指令的地址。其将输出的指令地址传给指令存储器，由时钟信号 clk 的上升沿触发，且当控制信号 PCWre=1 时，才进行指令地址的更新。PCWre=0 时，PC 不再改变，为停机处理。此外，Reset 是重置信号，PC 接收 Reset 信号，Reset=0 时，初始化 PC 的指令地址为 0x00000000。上述关键部分的代码实现如下，




```
initial begin
    out_pc = 0;
end
always @(posedge clk or negedge Reset) begin
    if(Reset==0)begin
        out_pc = 0;
    end
    else if(PCWre)begin // PCWre为0时为停机处理
        out_pc = in_pc;
    end
```

3.ControlUnit：控制单元，接收来自指令的 opcode 及 ALU 的 zero、sign，输出各个模块所需的控制信号，输出的信号用于控制各个模块的功能，主要的控制信号有 12 个：PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD, mWR, RegDst, ExtSel, PCSrc,ALUOp，各个控制信号具体功能会在各个部分中进行介绍。各个控制信号的值由相应 opcode 确定的指令类型及表 1 控制信号的作用得出。

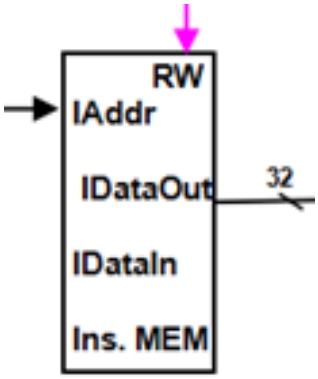


各个控制信号的逻辑表达式整理化简后如下所示，

```
assign ALUSrcA = (op == Sll);
assign ALUSrcB = (op == Addi || op == Andi || op == Ori || op == Slti || op == Sw || op == Lw);
assign PCWre = (op == Halt)?0:1;
assign DBDataSrc = (op == Lw);
assign RegWre = ~(op == Beq || op == Bne || op == Bltz || op == Sw || op == Halt);
assign InsMemRW = 1;
assign mRD = (op == Lw)?1:1'bz;
assign mWR = (op == Sw);
assign RegDst = ~(op == Addi || op == Andi || op == Ori || op == Slti || op == Lw);
assign ExtSel = (op != Andi && op != Ori);
assign PCSrc[0] = (op == J);
assign PCSrc[1] = ((op == Beq&& zero == 1) || (op == Bne && zero == 0) || (op == Bltz) && sign == 1);
assign ALUOp = (op == Sub || op == Beq || op == Bne || op == Bltz)? A_sub: op == Sll? A_sll: op == Or ||
op == Ori? A_or: op == And || op == Andi? A_and: op == Slti? A_slt : A_add;
```

4.InstruMem：即指令存储器，输入端口 IAddr 接收来自 PC 的指令地址，取出相应地址的指令后通过输出端口 IDataOut 输出。接收 RW 读写控制信号，RW=1 时为写指令，RW=0 时为读指令，由于我们将测试指令提前写入指令存储器，故我们将 RW 恒设为 0，即指令存储器相当于是只读的。此处的控制信号 RW 对应 ControlUnit 的 InsMemRW。

在实验中，通过一个 256 大小的 8 位寄存器数组来保存从文件输入的全部指令。然后通过输入的地址，找到相应的指令，输出到 IDataOut。

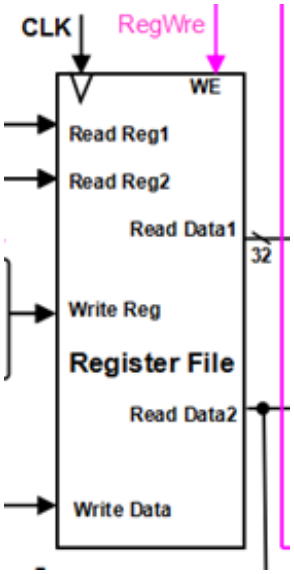


将 txt 文件内的指令写入指令存储器以及读取指令存储器内指令的代码实现如下所示，

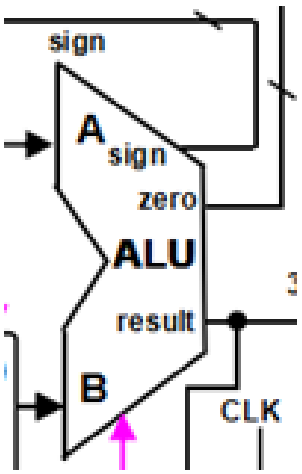
```
initial begin
    $readmemb("C:/Users/86158/Desktop/input.txt",Instmemory);
end
always @(Iaddr or RW)begin
    if(RW)begin
    InDataOut = {Instmemory[Iaddr],Instmemory[Iaddr+1],Instmemory[Iaddr+2],Instmemory[Iaddr+3]};
    end
```

5.RegFile：寄存器堆，是寄存器读写单元。读入两个寄存器号，输出两个寄存器存储的值。其接收时钟信号 clk，写使能 WE（对应 ControlUnit 的 RegWre），写控制信号 WE 由 clk 的下降沿触发，WE=1 时，写操作有效，WriteData 将被写入寄存器 rt 或 rd。

在实验中，通过一个 32 大小的 32 位寄存器数组来模拟寄存器，开始时全部置 0。通过访问寄存器的地址，来获取寄存器里面的值，并进行操作。另外，由于$0 恒为 0，所以写入寄存器的地址不能为 0。



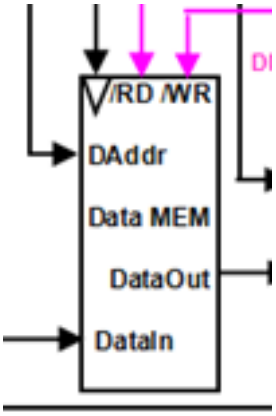
6.ALU：算术逻辑单元。接受两个操作数 A、B，接收来自 ControlUnit 的 3 位操作码 ALUOp 并根据其值完成相应的算术逻辑运算。其中 zero、sign 均为运算结果标志，result 为 0，则 zero=1，否则 zero=0.而 result 最高位为 0，则 sign=0，即表示结果为正，最高位为 1，则 sign=1，结果为负。



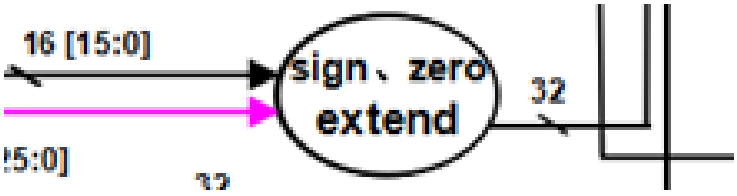
其中参数 A、B 为两个 32 位操作数，zero、sign 为运算结果标识，result 为结果。则根据 ALUOp 的取值来实现相应的功能。通过 case 语句方法逐个产生各指令控制信号，其代码实现如下所示。

```
case(ALUOp)
    3'b000: result = A + B;//加法
    3'b001: result = A - B;//减法
    3'b010: result = B << A;//B左移A位
    3'b011: result = A | B;//或
    3'b100: result = A & B;//与
    3'b101: result = A < B; //比较A<B不带符号
    3'b110: result = (((A<B)&&(A[31] == B[31])) || ((A[31]==1&& B[31] == 0))) ? 1:0;
    3'b111: result = A ^ B;//异或
    default: result = 0;
```

7.DataMem：数据存储器。读入来自 ALU 的运算结果 result 作为 DAddr，接受时钟信号 clk、来自 ControlUnit 的 mRD、mWR，mRD、mWR 对应的输入端口为 RD、WR，RD=1 时，对应指令 lw，表示根据内存地址 DAddr 从存储器中读出相应内存单元的值给 DataOut 并输出，RD = z（高阻态）时，存储器不可读；WR=1 时，对应指令 sw，由 clk 的下降沿触发写信号，将寄存器堆的第二个源寄存器的值写入由 DAddr 确定地址的内存单元。相应地，WR=0 时，存储器不可写。在实验中，采用 255 大小的 8 位寄存器数组模拟内存，并且采用小端模式进行实现。

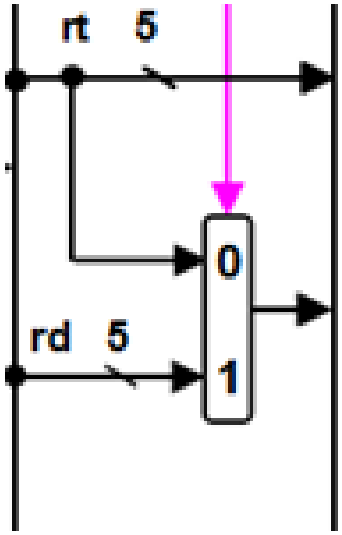


8.SignExtend：符号扩展用于将输入的 16 位立即数扩展成 32 位立即数，输出扩展后的 32 位立即数。接收来自 ControlUnit 的 ExtSel 信号，ExtSel=1 时，对立即数进行有符号扩展，ExtSel=0 时，对立即数进行 0 扩展（逻辑扩展）。

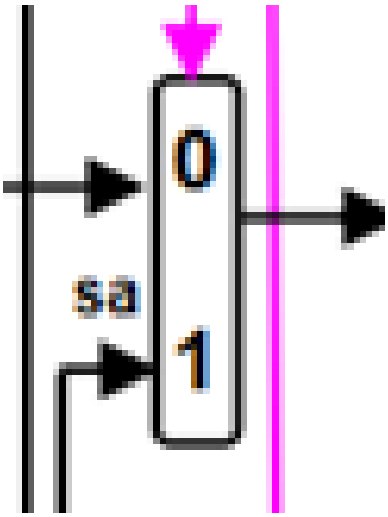



```
assign extendImmediate = {ExtSel&&immediate[15]?16'hffff:16'h0000,immediate};
```

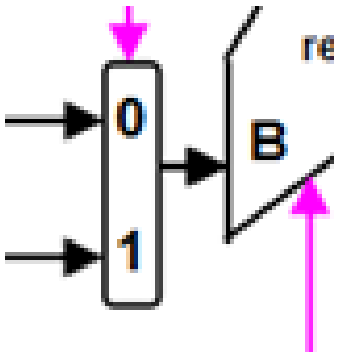
9.mux21R：目的寄存器选择器，接收来自 ControlUnit 的 RegDst 信号，RegDst=1 时，选择来自指令的 rd 字段作为写寄存器的地址。否则，将 rt 字段作为地址。



10.mux21A：2 选 1 选择器，接收来自 ControlUnit 的 ALUSrcA 信号，ALUSrcA=0 时，将选择来自寄存器堆的源寄存器 1 的值 Data1 作为输出。ALUSrcA = 1 时，此时涉及的指令为 sll，选择来自指令的进行了 0 扩展的 10：6 字段 sa 作为输出。



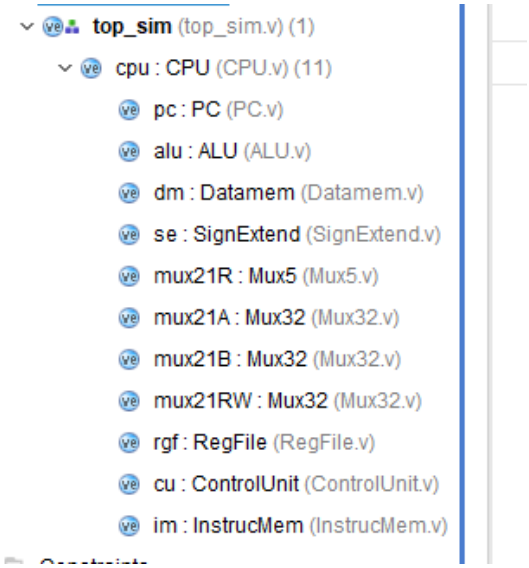
11.mux21B：2 选 1 选择器，接收来自 ControlUnit 的 ALUSrcB 信号，ALUSrcB=0 时，将选择来自寄存器堆的源寄存器 2 的值 Data2 作为输出。ALUSrcB=1 时，将选择进行过扩展的立即数作为输出。



12.mux21RW：2 选 1 选择器，接收来自 ControlUnit 的 DBDataSrc 信号，

DBDataSrc=0 时，选择来自 ALU 的运算结果 result 作为输出。DBDataSrc=1 时，选择来自存储器的输出 DataOut。

按照上述描述对每一个模块进行设计，文件结构如下



仿真文件代码：

```
module top_sim();
    reg clk;   //时钟信号
    reg Reset;    //置零信号


    CPU cpu(clk,Reset);
    initial begin
       clk = 0;
       Reset = 0;    //刚开始设置pc为0

       #50;    //等待Reset完成
       clk = !clk;    //下降沿，使PC先清零
       #50;
       Reset = 1;    //清除保持信号
       forever #50 begin    //产生时钟信号，周期为50s
       clk = !clk;
       end
   end
endmodule
```

1、测试程序段

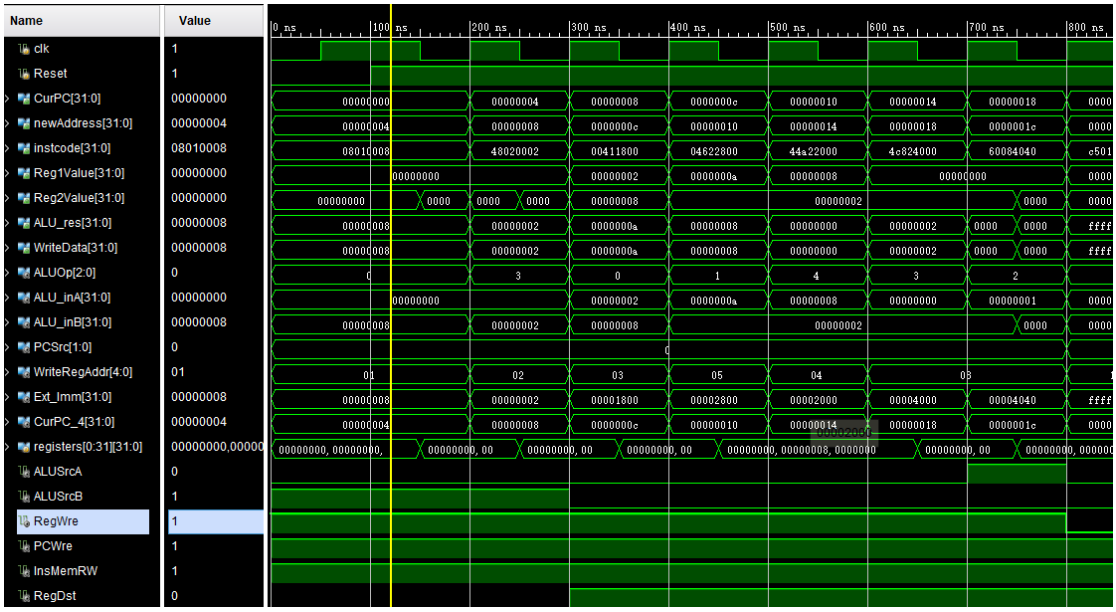
地址	汇编程序	指令代码					
		op（6）	rs(5)	rt(5)	rd(5)/immediate (16)	16 进制数代码	
0x00000000	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008
0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010		48020002
0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000		00411800
0x0000000C	sub $5,$3,$2	000001	00011	00010	0010 1000 0000 0000		04622800
0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0000 0000		44A22000
0x00000014	or $8,$4,$2	010011	00100	00010	0100 0000 0000 0000		4C824000
0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000		60084040
0x0000001C	bne $8,$1,-2 (≠,转 18)	110001	01000	00001	1111 1111 1111 1110		C501FFFE
0x00000020	slti $6,$2,4	011100	00010	00110	0000 0000 0000 0100		70460004
0x00000024	slti $7,$6,0	011100	00110	00111	0000 0000 0000 0000		70C70000
0x00000028	addiu $7,$7,8	000010	00111	00111	0000 0000 0000 1000		08E70008
0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110		C0E1FFFE
0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100		98220004
0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100		9C290004
0x00000038	addiu $10,$0,-2	000010	00000	01010	1111 1111 1111 1110		080AFFFE
0x0000003C	addiu $10,$10,1	000010	01010	01010	0000 0000 0000 0001		094A0001
0x00000040	bltz $10,-2(<0,转 3C)	110010	01010	00000	1111 1111 1111 1110		C940FFFE
0x00000044	andi $11,$2,2	010000	00010	01011	0000 0000 0000 0010		404B0002
0x00000048	j 0x00000050	111000	00000	00000	0000 0000 0001 0100		E0000014
0x0000004C	or $8,$4,$2	010011	00100	00010	0100 0000 0000 0000		4C824000
0x00000050	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

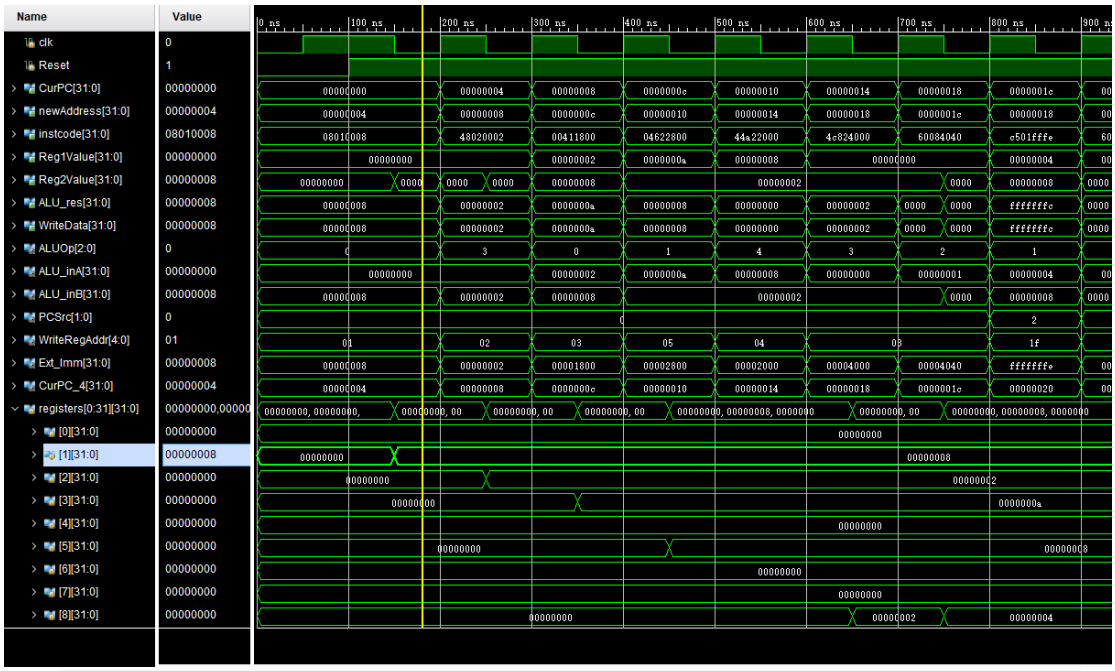
注：该表对应的文件为 tests.txt。

（二）验证 CPU 的正确性，结合仿真波形对每一条指令进行验证。

1.addiu $1, $0, 8

对应 PC 地址为 CurPC=0x00000000，立即数加法，下一条指令为 newAddress 的值，PC 正常+4，故 ALUSrcA=0，ALUSrcB=1，ALUOp=0，RegDst=0，RegWre=1，ExtSel=1，立即数 8 进行有符号扩展，ALU_inA，ALU_inB 分别是 0、8，结果 ALU_res 的值为 0+8=8，写入寄存器$1 值为 8，正确。

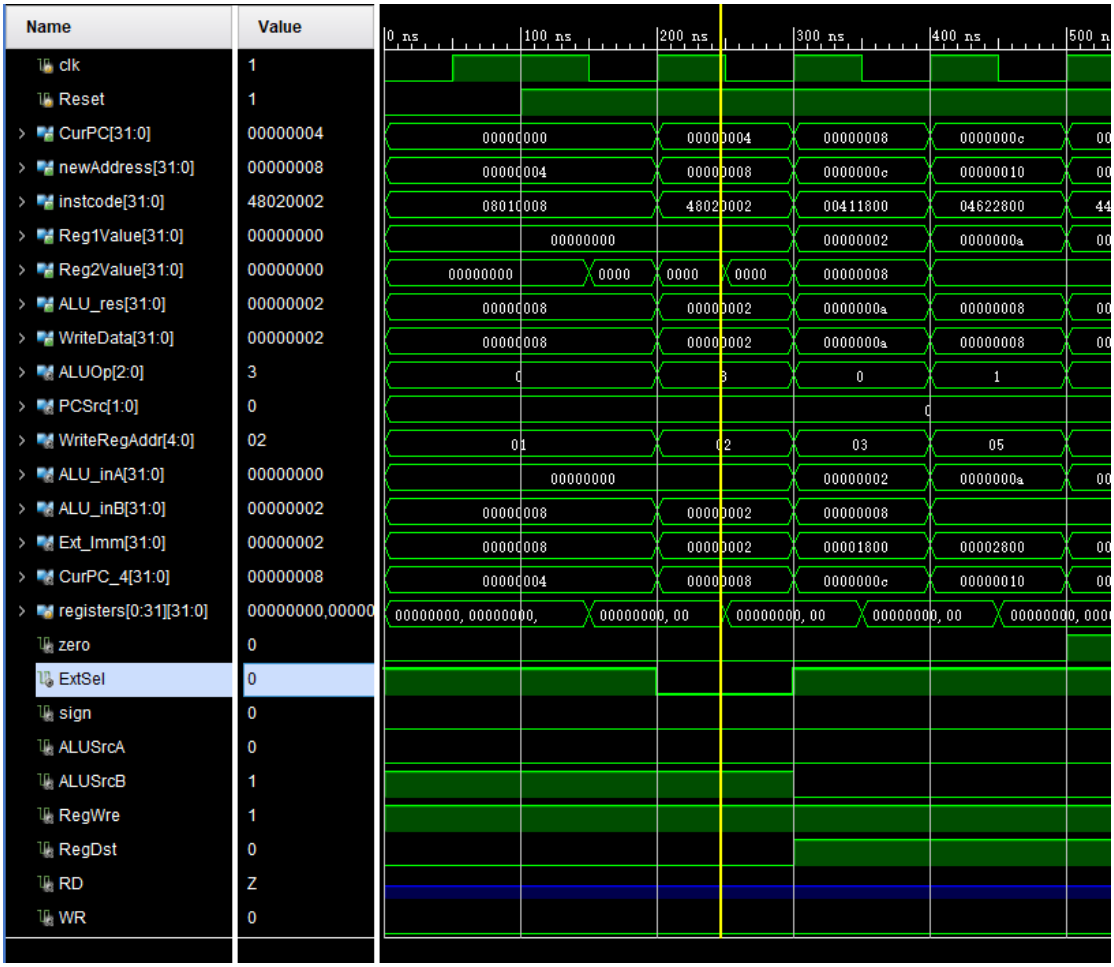


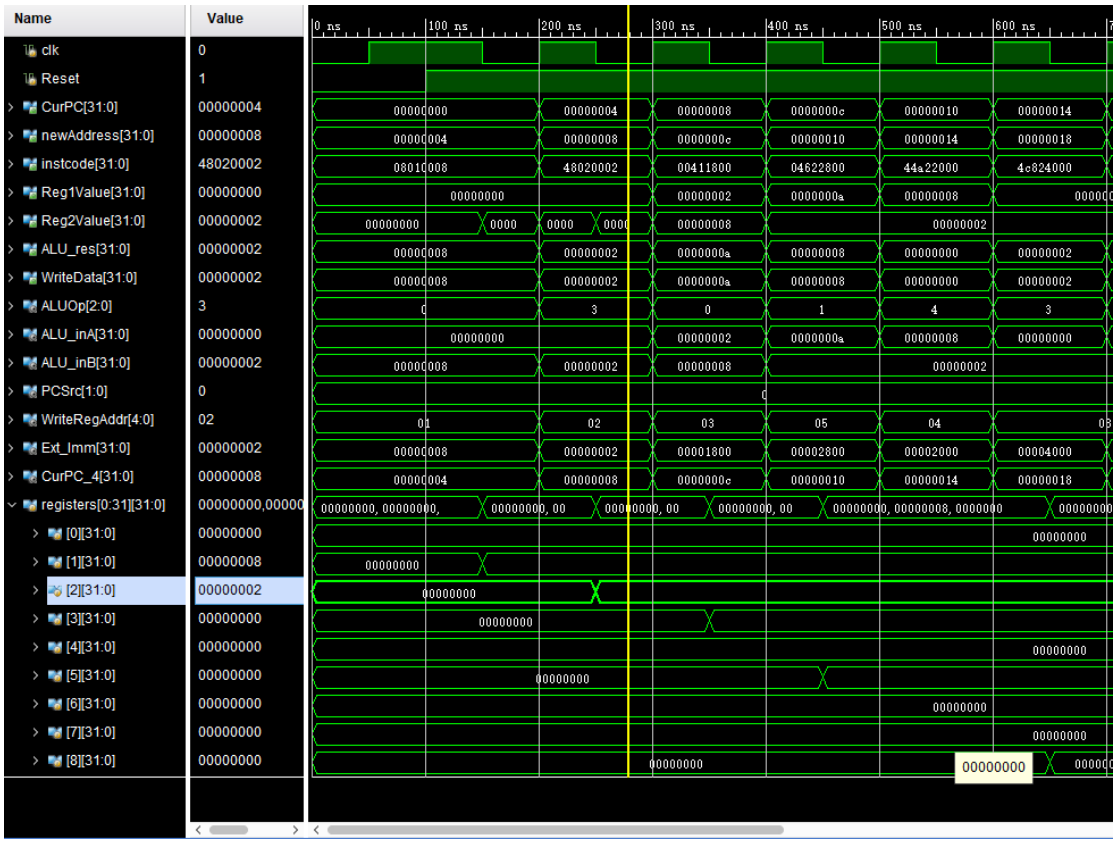


同理可验证得，addiu $7,$7,8　addiu　$10,$0,-2　addiu　$10,$10,1 正确。

2. ori $2,$0,2

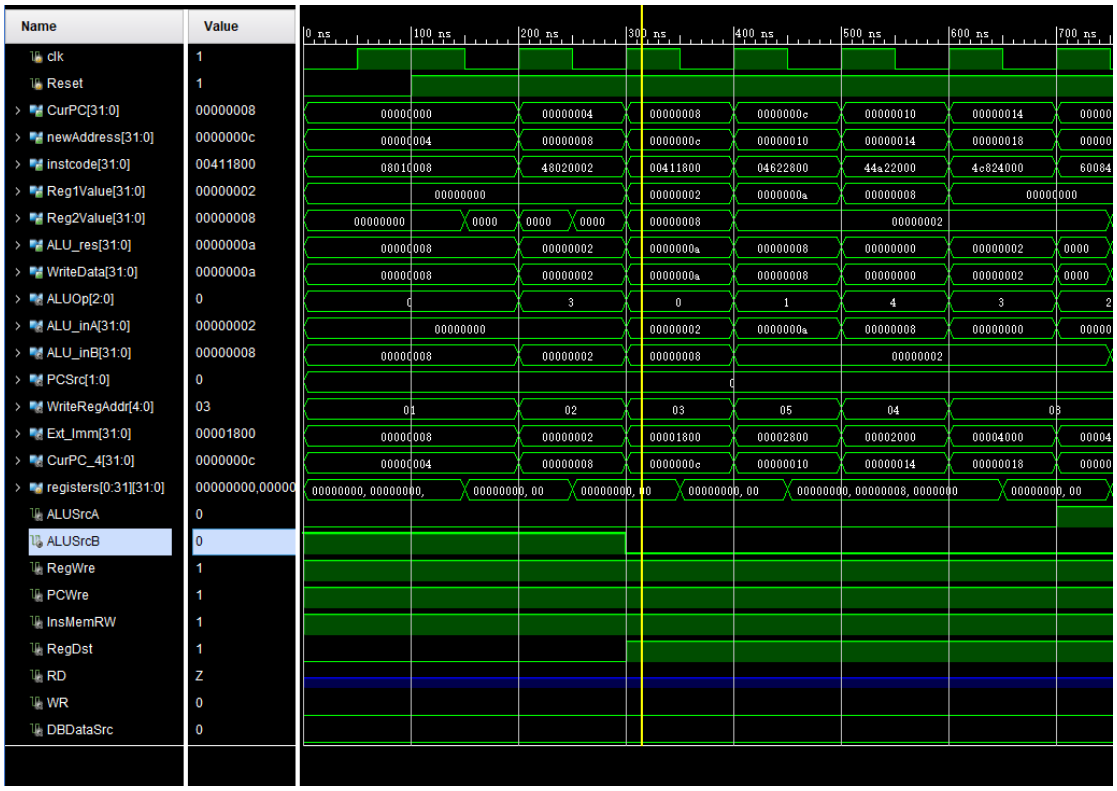
对应 PC 地址为 CurPC=0x00000004，下一条指令为 newAddress 的值，PC 正常+4，立即数或，故 ALUSrcA=0，ALUSrcB=1，ALUOp=3（011），RegDst=0，RegWre=1，ExtSel=0，立即数 2 进行 0 扩展，ALU_inA，ALU_inB 分别是 0、2，0 与 2 相或的值为 2，结果 ALU_res=2，写入寄存器$2 值为 2，正确。

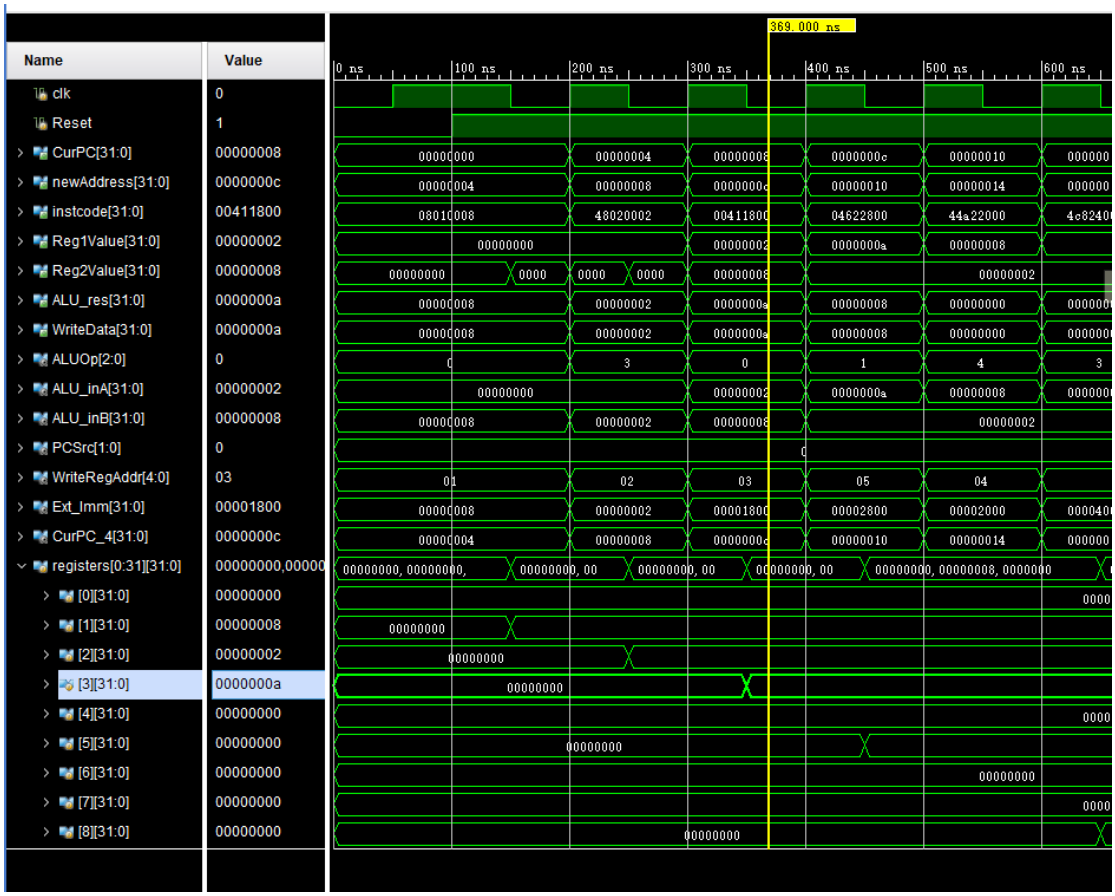

Name
Value
clk 1
Reset 1
CurPC[31:0] 00000004
newAddress[31:0] 00000008
instcode[31:0] 48020002
Reg1Value[31:0] 00000000
Reg2Value[31:0] 00000000
ALU_res[31:0] 00000002
WriteData[31:0] 00000002
ALUOp[2:0] 3
PCSrc[1:0] 0
WriteRegAddr[4:0] 02
ALU_inA[31:0] 00000000
ALU_inB[31:0] 00000002
Ext_Imm[31:0] 00000002
CurPC_4[31:0] 00000008
registers[0:31][31:0] 00000000,00000
zero 0
ExtSel 0
sign 0
ALUSrcA 0
ALUSrcB 1
RegWre 1
RegDst 0
RD Z
WR 0
0 ns
100 ns
200 ns
300 ns
400 ns
500 n


Name
Value
clk 0
Reset 1
CurPC[31:0] 00000004
newAddress[31:0] 00000008
instcode[31:0] 48020002
Reg1Value[31:0] 00000000
Reg2Value[31:0] 00000002
ALU_res[31:0] 00000002
WriteData[31:0] 00000002
ALUOp[2:0] 3
ALU_inA[31:0] 00000000
ALU_inB[31:0] 00000002
PCSrc[1:0] 0
WriteRegAddr[4:0] 02
Ext_Imm[31:0] 00000002
CurPC_4[31:0] 00000008
registers[0:31][31:0] 00000000,00000
[0][31:0] 00000000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 00000000
[4][31:0] 00000000
[5][31:0] 00000000
[6][31:0] 00000000
[7][31:0] 00000000
[8][31:0] 00000000
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns

3. add $3, $2, $1

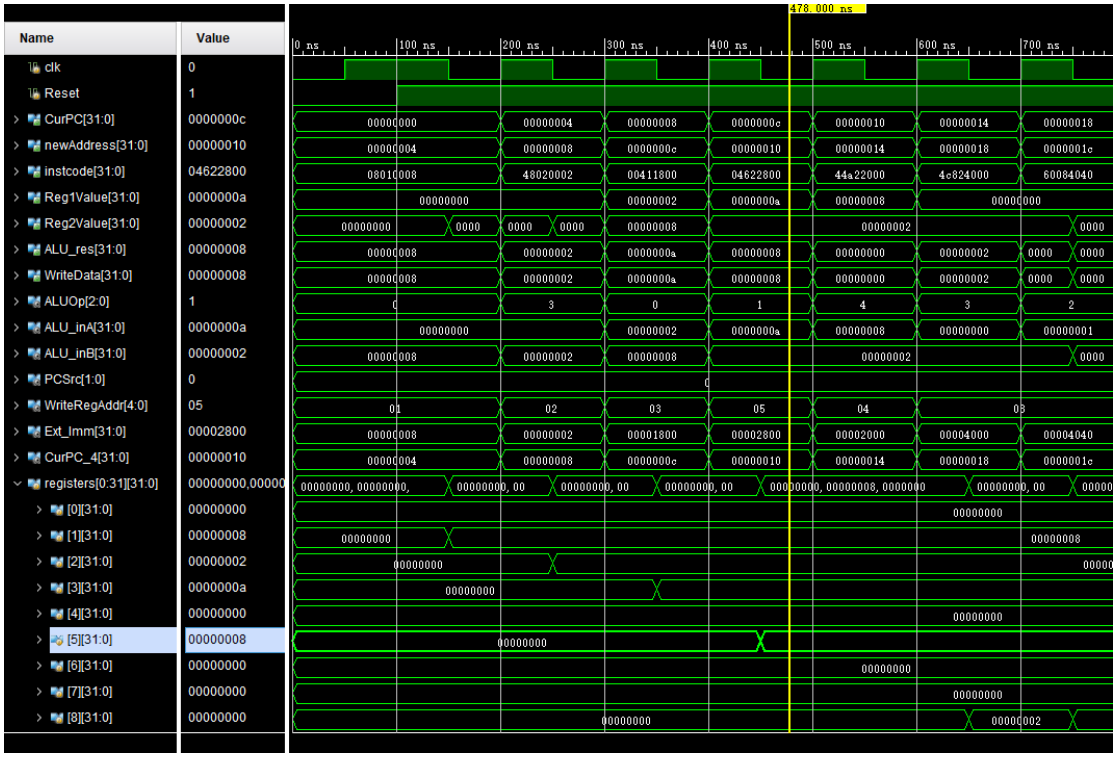
对应的 PC 地址为 CurPC=0x00000008，下一条指令为 newAddress 的值，PC 正常+4，两寄存器相加，故此时 ALUSrcA=0，ALUSrcB=0，ALUOp=0（000），RegDst=1，RegWre=1，ALU_inA，ALU_inB 分别是 2、8，即分别是$2、$1 的值，结果 ALU_res=2+8=10,写入寄存器$3 的值为 10，正确。





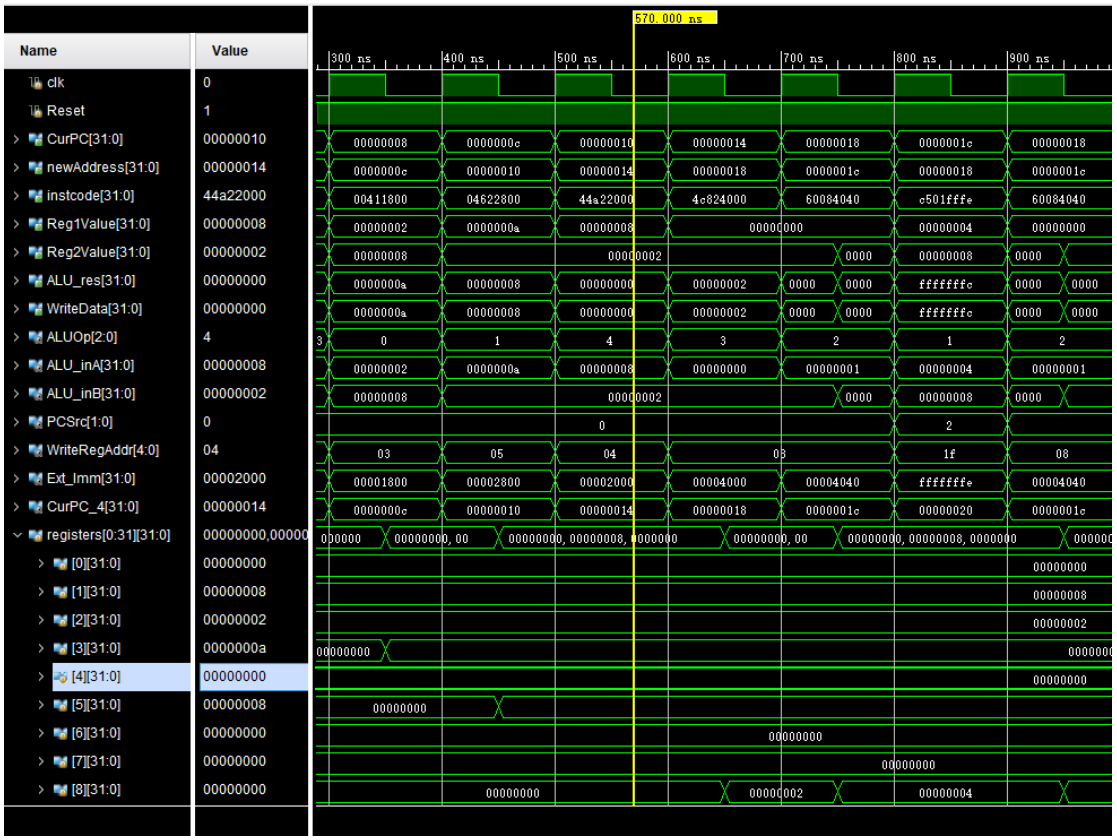
4. sub $5, $3, $2

对应的 PC 地址为 CurPC=0x0000000c，下一条指令为 newAddress 的值，PC 正常+4，两寄存器相减，故此时 ALUSrcA=0，ALUSrcB=0，ALUOp=1（001），RegDst=1，RegWre=1，ALU_inA，ALU_inB 分别是 10、2，即分别是$3、$2 的值，结果 ALU_res=10-2=8,写入寄存器$5 的值为 8，正确。



5. and $4, $5, $2

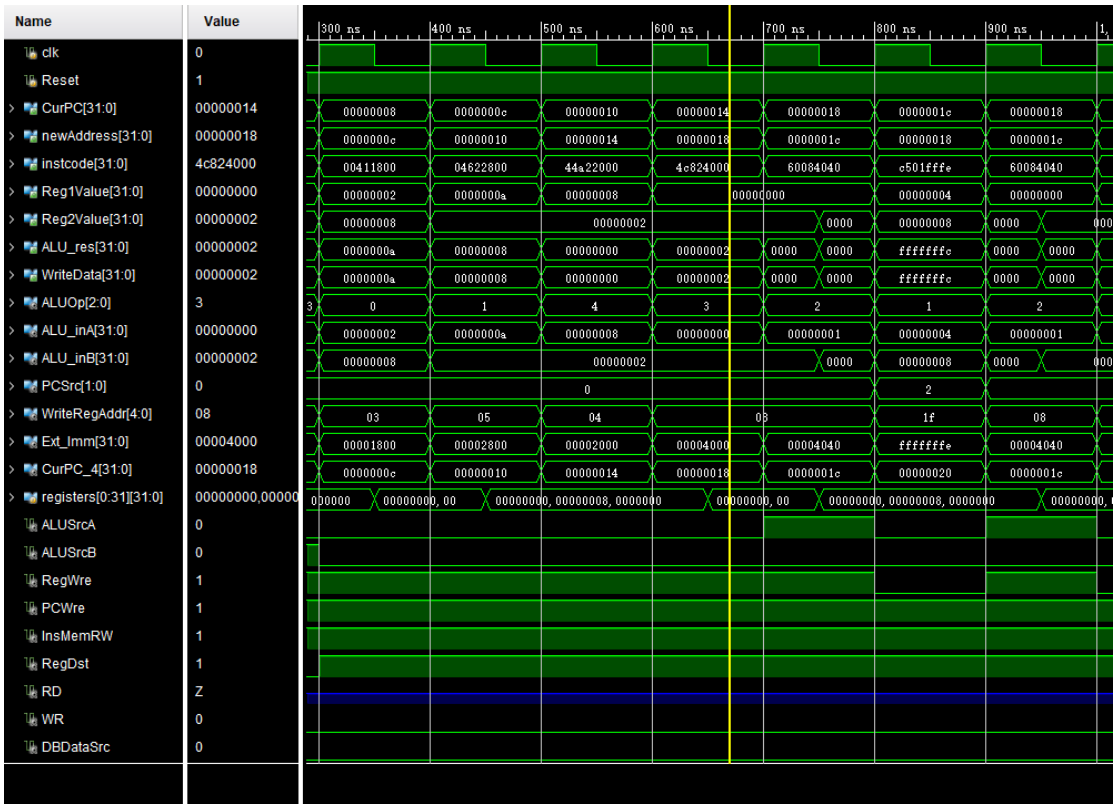
对应的 PC 地址为 CurPC=0x00000010，下一条指令为 newAddress 的值，PC 正常+4，两寄存器相与，故此时 ALUSrcA=0，ALUSrcB=0，ALUOp=4（100），RegDst=1，RegWre=1,ALU_inA,ALU_inB 分别是 8、2,即分别是$5、$2 的值,8 与 2 相与,ALU_res 值为 0，写入寄存器$4 的值为 0，正确。



6. or $8, $4, $2

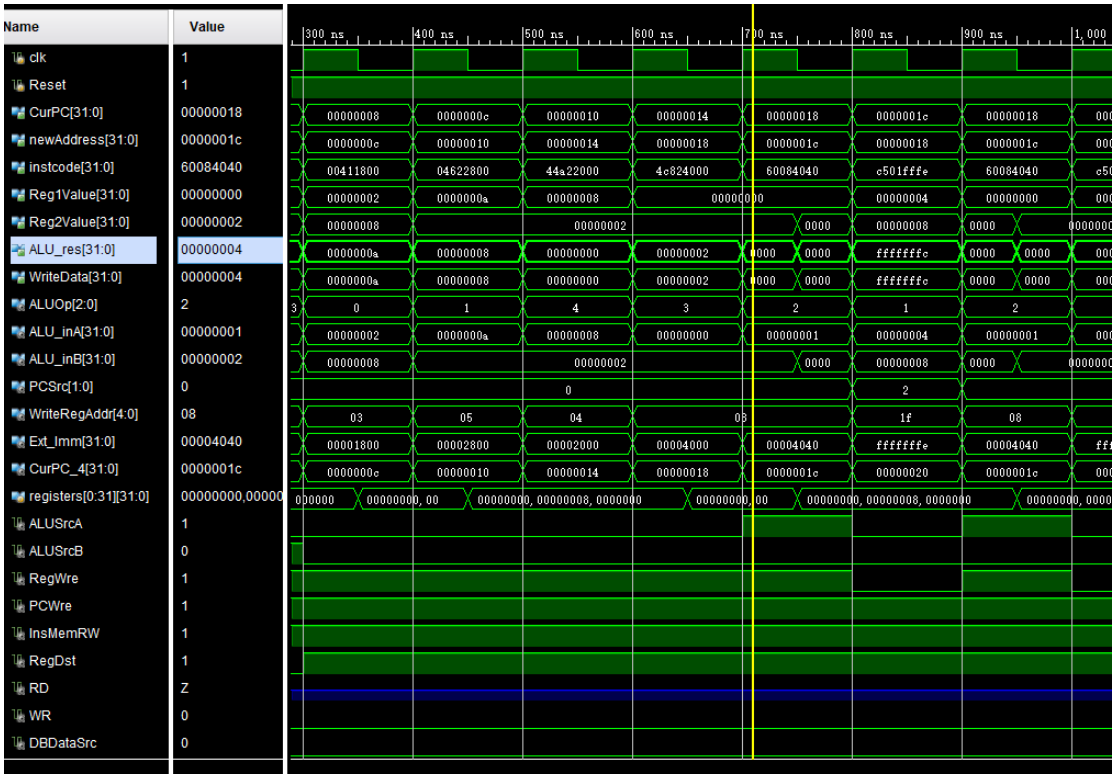
对应的 PC 地址为 CurPC=0x00000014，下一条指令为 newAddress 的值，PC 正常+4，两寄存器相或，故此时 ALUSrcA=0，ALUSrcB=0，ALUOp=3（011），RegDst=1，RegWre=1,ALU_inA,ALU_inB 分别是 0、2,即分别是$4、$2 的值,0 与 2 相或,ALU_res 值为 2，写入寄存器$8 的值为 2，正确。

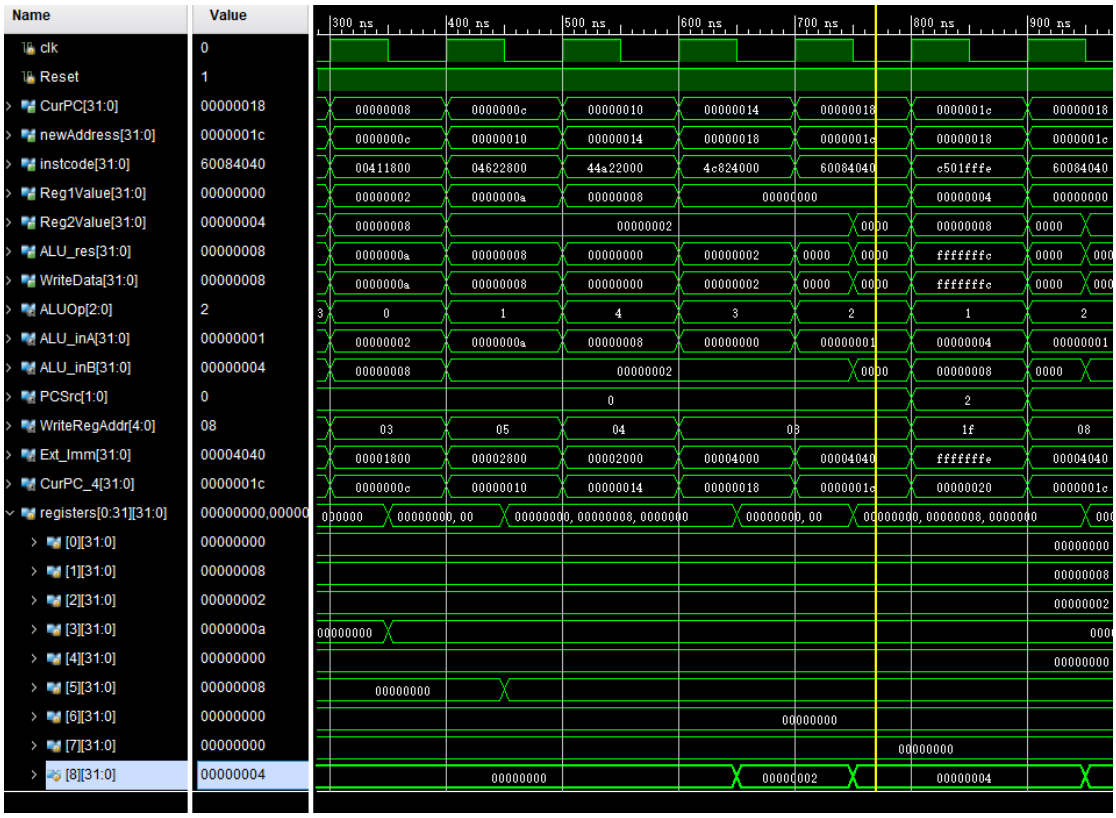
同理可验证得，ori　$2,$0,2 正确。



7. sll $8, $8, 1

对应的 PC 地址为 CurPC=0x00000018，下一条指令为 newAddress 的值，PC 正常+4，左移指令，故此时 ALUSrcA=1，ALUSrcB=0，ALUOp=2 (010)，RegDst=1，RegWre=1，ALU_inA，ALU_inB 分别为 1、2，其中 1 来自指令字段 10：6，移位后结果 ALU_res=4，写入寄存器$8 的值为 4，正确。

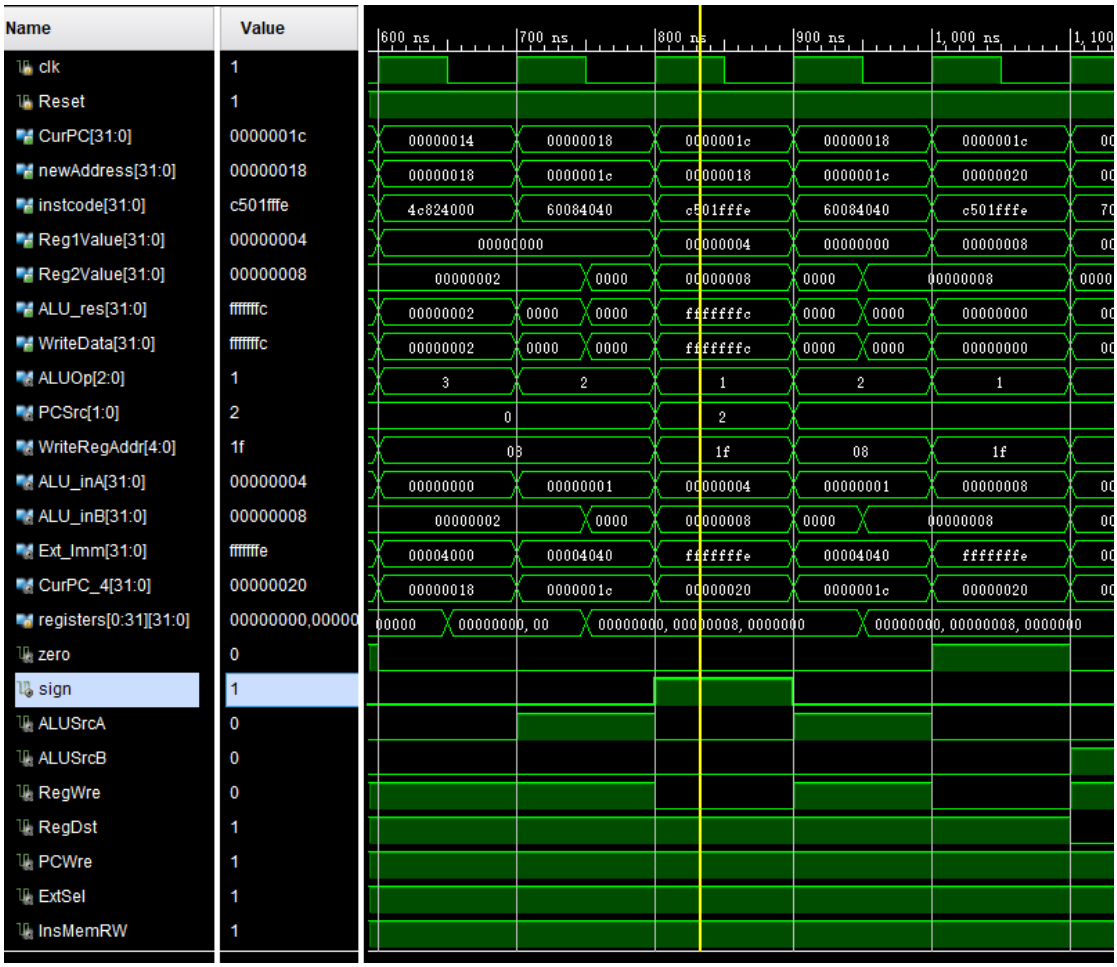




8. bne $8, $1, -2

对应的 PC 地址为 CurPC=0x0000001c，比较指令，则此时 ALUSrcA=0，ALUSrcB=0，ALUOp=1（001），RegWre=0，ALU_inA，ALU_inB 分别为 4、8，即$8、$1 的值，不相等，故 zero=0，又因为 4-8=-4，故 sign=1，则 PCSrc=2（10），CurPC_4 即 PC+4

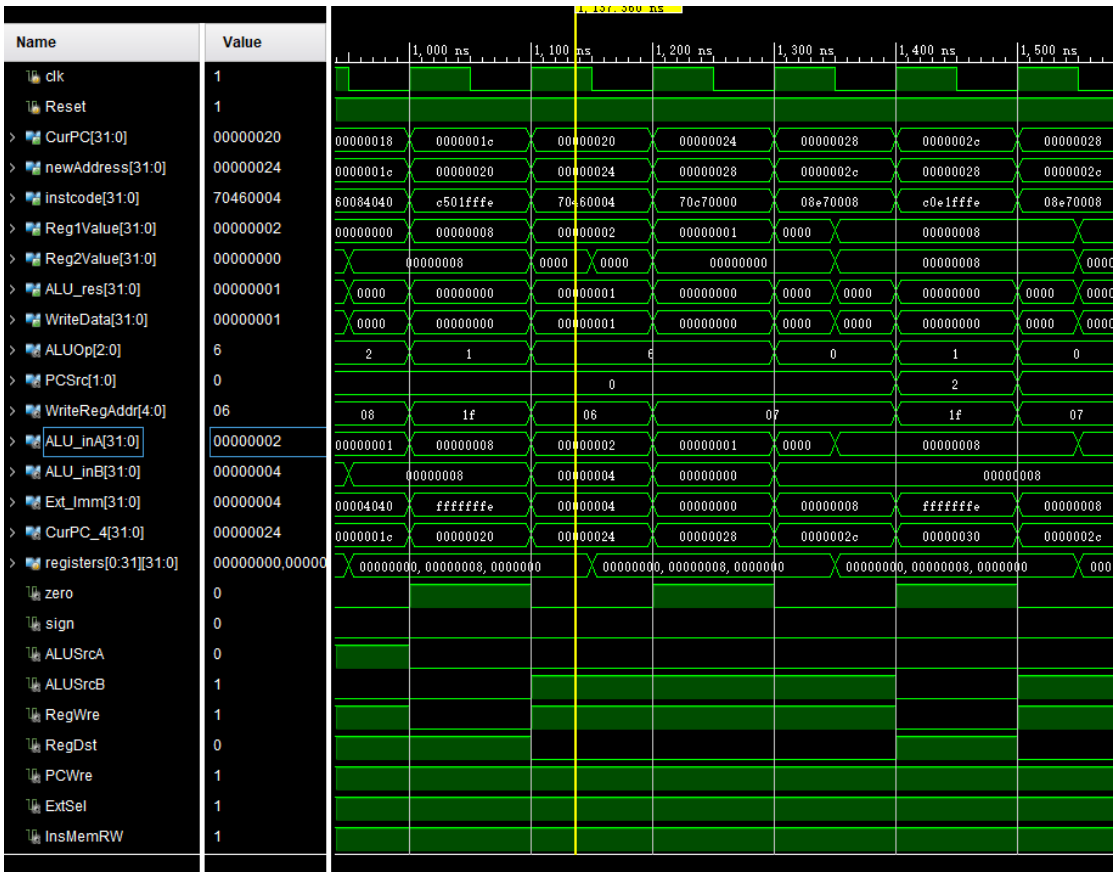
与符号扩展且左移 2 位的立即数-2 相加，得到了下一条指令地址，作为 newAddress 的值，即 0x00000018，正确。

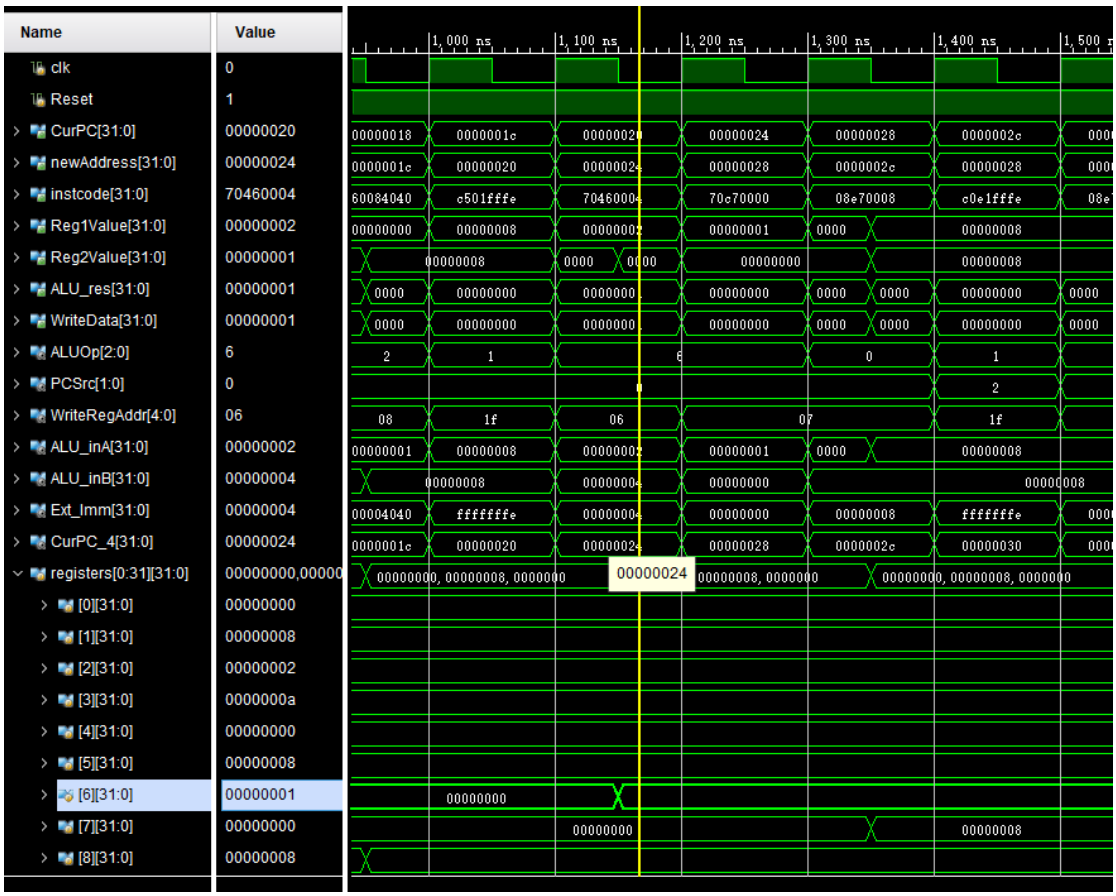


9. slti $6, $2, 4

对应的 PC 地址为 CurPC=0x00000020，下一条指令为 newAddress 的值，PC 正常+4，$2 的值与立即数比较，故此时 ALUSrcA=0，ALUSrcB=1，ALUOp=6 (110)，RegDst=0，RegWre=1，ALU_inA，ALU_inB 分别是 2、4，即$2 和立即数 4 的值，由于 2<4,故 ALU_res=1,写入寄存器$6 的值为 1，正确。

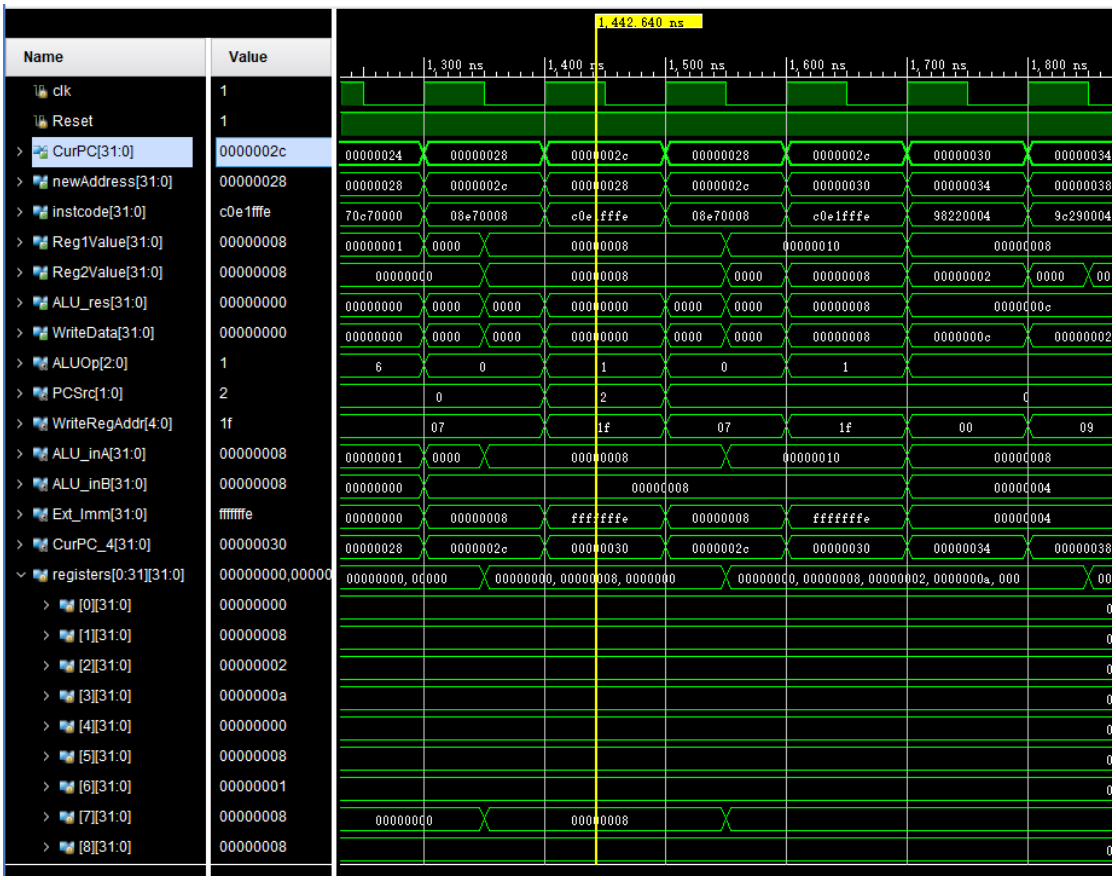
同理可验证得，slti $7,$6,0 正确。


Name
Value
clk 1
Reset 1
CurPC[31:0] 00000020
newAddress[31:0] 00000024
instcode[31:0] 70460004
Reg1Value[31:0] 00000002
Reg2Value[31:0] 00000000
ALU_res[31:0] 00000001
WriteData[31:0] 00000001
ALUOp[2:0] 6
PCSrc[1:0] 0
WriteRegAddr[4:0] 06
ALU_inA[31:0] 00000002
ALU_inB[31:0] 00000004
Ext_Imm[31:0] 00000004
CurPC_4[31:0] 00000024
registers[0:31][31:0] 00000000,00000
zero 0
sign 0
ALUSrcA 0
ALUSrcB 1
RegWre 1
RegDst 0
PCWre 1
ExtSel 1
InsMemRW 1
1,000 ns
1,100 ns
1,200 ns
1,300 ns
1,400 ns
1,500 ns


Name
Value
clk 0
Reset 1
CurPC[31:0] 00000020
newAddress[31:0] 00000024
instcode[31:0] 70460004
Reg1Value[31:0] 00000002
Reg2Value[31:0] 00000001
ALU_res[31:0] 00000001
WriteData[31:0] 00000001
ALUOp[2:0] 6
PCSrc[1:0] 0
WriteRegAddr[4:0] 06
ALU_inA[31:0] 00000002
ALU_inB[31:0] 00000004
Ext_Imm[31:0] 00000004
CurPC_4[31:0] 00000024
registers[0:31][31:0] 00000000,00000
[0][31:0] 00000000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a
[4][31:0] 00000000
[5][31:0] 00000008
[6][31:0] 00000001
[7][31:0] 00000000
[8][31:0] 00000008
00000024
1,000 ns
1,100 ns
1,200 ns
1,300 ns
1,400 ns

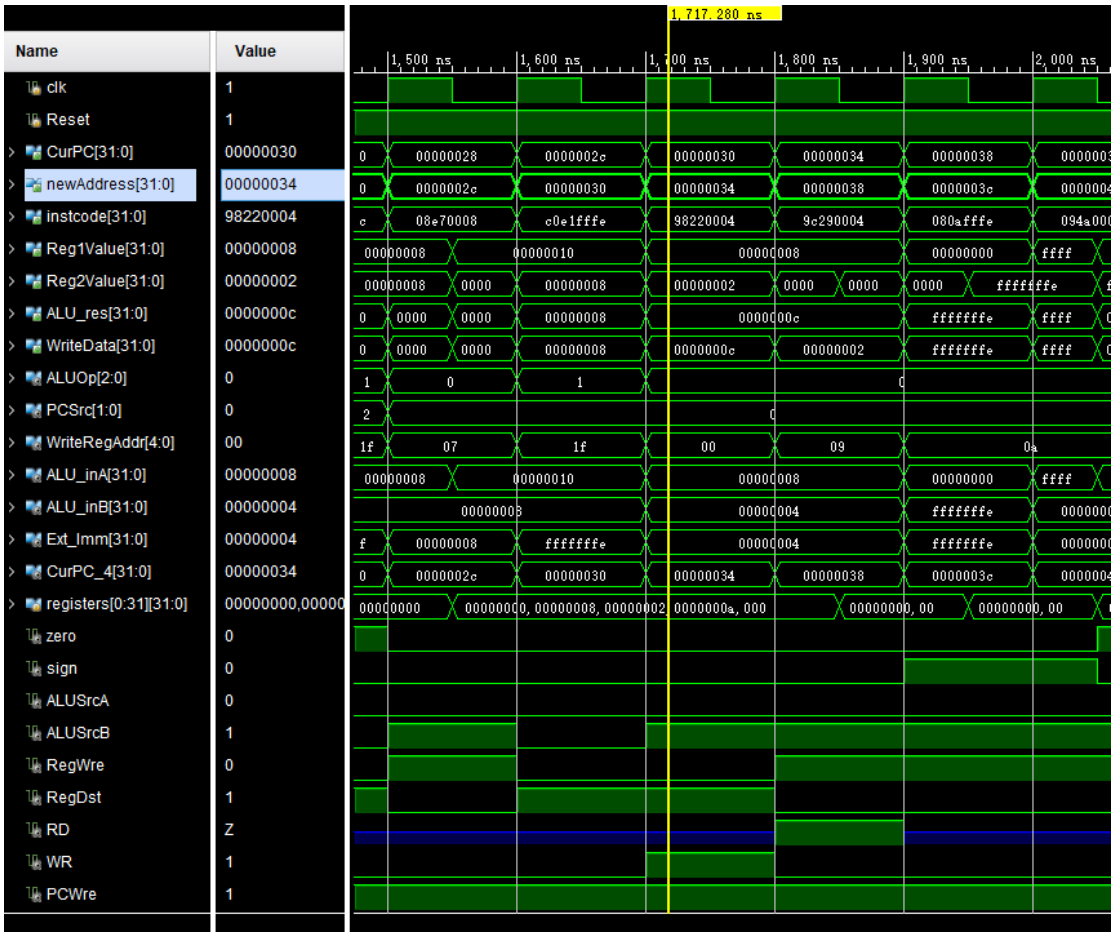
10. beq $7, $1, -2

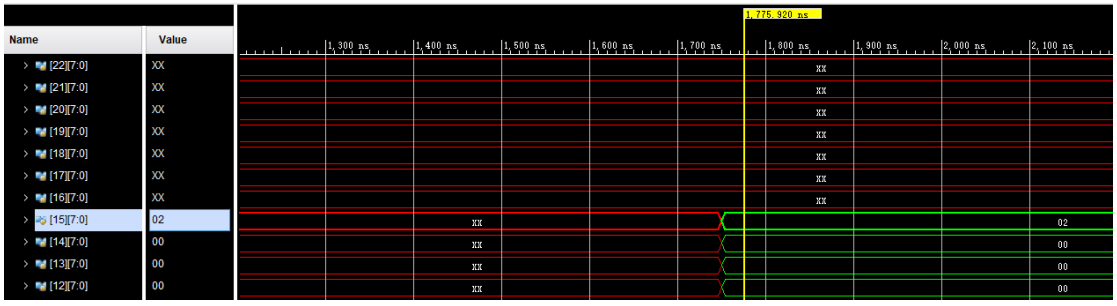
对应的 PC 地址为 CurPC=0x0000002c，比较指令，则此时 ALUSrcA=0，ALUSrcB=0，ALUOp=1（001），RegWre=0，ALU_inA，ALU_inB 分别为 8、8，即$7、$1 的值，相等，故 zero=1，8-8=0，故 sign=0，则 PCSrc=2（10），CurPC_4 即 PC+4 与符号扩展且左移 2 位的立即数-2 相加，得到了下一条指令地址，作为 newAddress 的值，即 0x00000028，正确。



11. sw $2, 4($1)

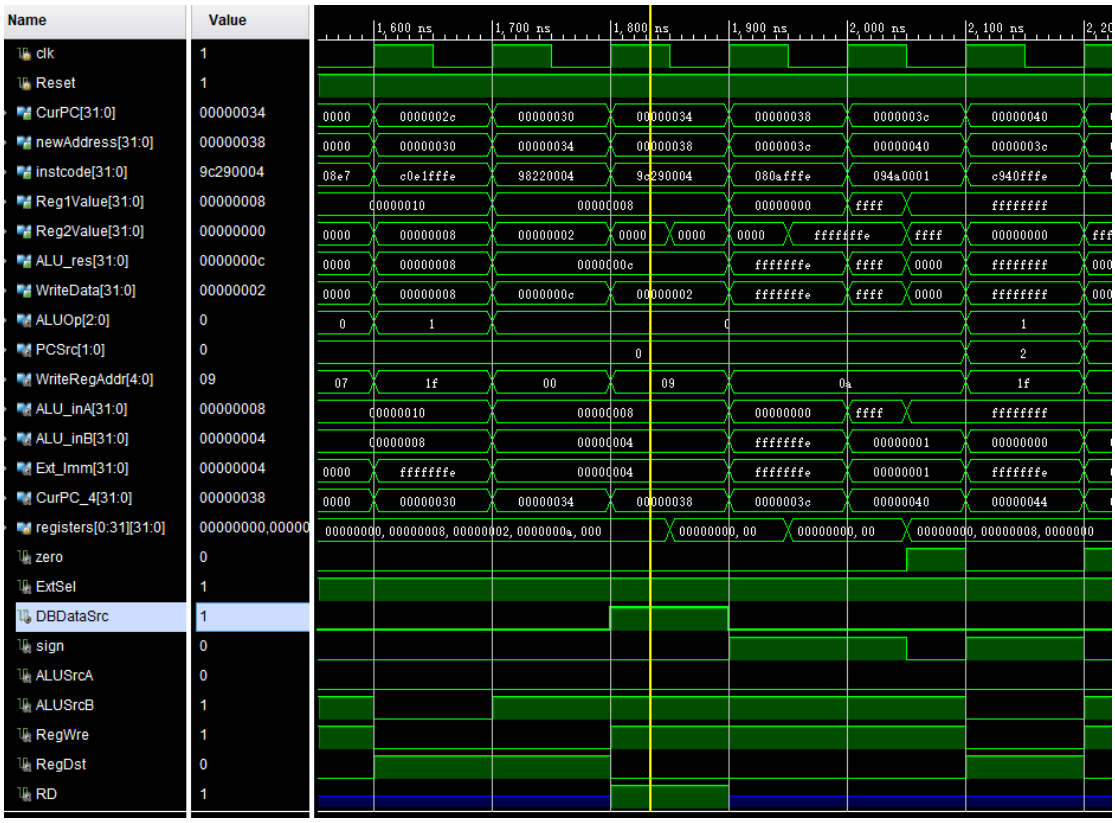
对应的 PC 地址为 CurPC=0x00000030，下一条指令为 newAddress 的值，PC 正常+4，存数指令，故此时 ALUSrcA=0，ALUSrcB=1，ALUOp=0（000），WR=1，RegWre=0，ALU_inA，ALU_inB 分别为 8、4，即$1、立即数 4 的值，8+4=12，此为写入数据存储器的内存地址，相应的起始地址为 12 内存单元的值为 2，即$2 的值，正确。（存储单元采取 8 位 1 小单元的策略。存储 32 位数需要 4 个小单元，4 小单元组成一个内存单元，内存地址即为第一个小单元地址，即起始地址）

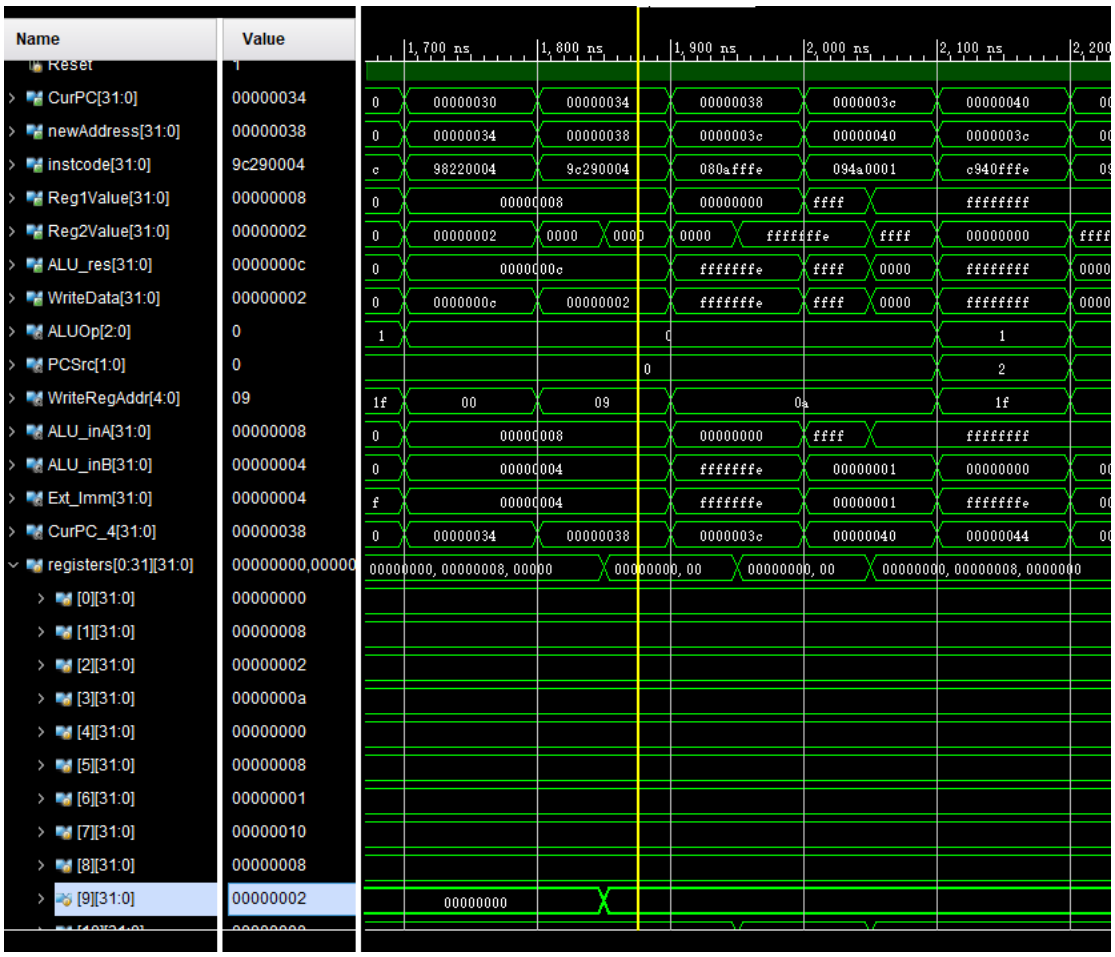




12. lw $9, 4($1)

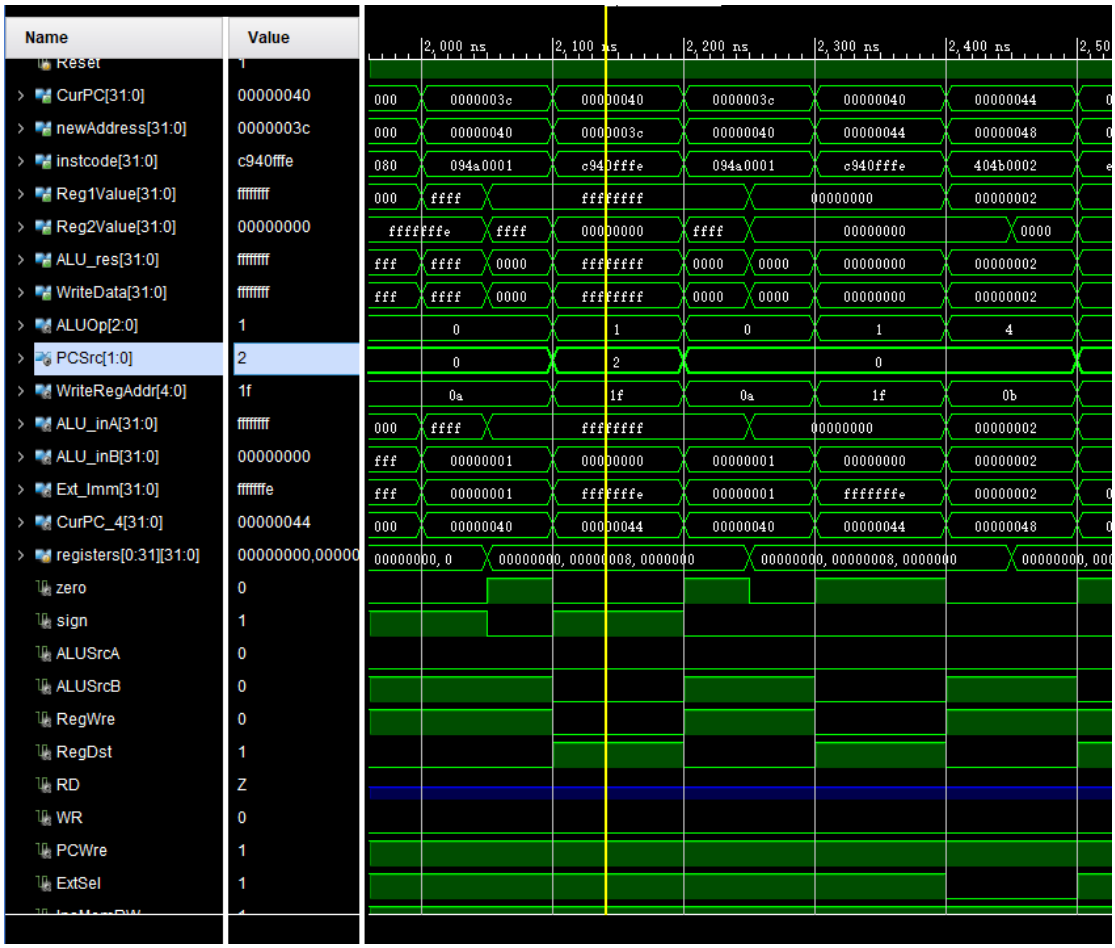
对应的 PC 地址为 CurPC=0x00000034，下一条指令为 newAddress 的值，PC 正常+4，取数指令，则故此时 ALUSrcA=0，ALUSrcB=1，ALUOp=0（000），RD=1，WR=0，RegWre=1，RegDst=0，DBDataSrc=1，ALU_inA，ALU_inB 分别为 8、4，即$1、立即数 4 的值，8+4=12，此为取数据的内存地址，写入寄存器$9 的值为 2，正确。


Name
Value
clk 1
Reset 1
CurPC[31:0] 00000034
newAddress[31:0] 00000038
instcode[31:0] 9c290004
Reg1Value[31:0] 00000008
Reg2Value[31:0] 00000000
ALU_res[31:0] 0000000c
WriteData[31:0] 00000002
ALUOp[2:0] 0
PCSrc[1:0] 0
WriteRegAddr[4:0] 09
ALU_inA[31:0] 00000008
ALU_inB[31:0] 00000004
Ext_Imm[31:0] 00000004
CurPC_4[31:0] 00000038
registers[0:31][31:0] 00000000,00000
zero 0
ExtSel 1
DBDataSrc 1
sign 0
ALUSrcA 0
ALUSrcB 1
RegWre 1
RegDst 0
RD 1
1,600 ns
1,700 ns
1,800 ns
1,900 ns
2,000 ns
2,100 ns


Name
Value
Reset 1
CurPC[31:0] 00000034
newAddress[31:0] 00000038
instcode[31:0] 9c290004
Reg1Value[31:0] 00000008
Reg2Value[31:0] 00000002
ALU_res[31:0] 0000000c
WriteData[31:0] 00000002
ALUOp[2:0] 0
PCSrc[1:0] 0
WriteRegAddr[4:0] 09
ALU_inA[31:0] 00000008
ALU_inB[31:0] 00000004
Ext_Imm[31:0] 00000004
CurPC_4[31:0] 00000038
registers[0:31][31:0] 00000000,00000
[0][31:0] 00000000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a
[4][31:0] 00000000
[5][31:0] 00000008
[6][31:0] 00000001
[7][31:0] 00000010
[8][31:0] 00000008
[9][31:0] 00000002
1,700 ns
1,800 ns
1,900 ns
2,000 ns
2,100 ns

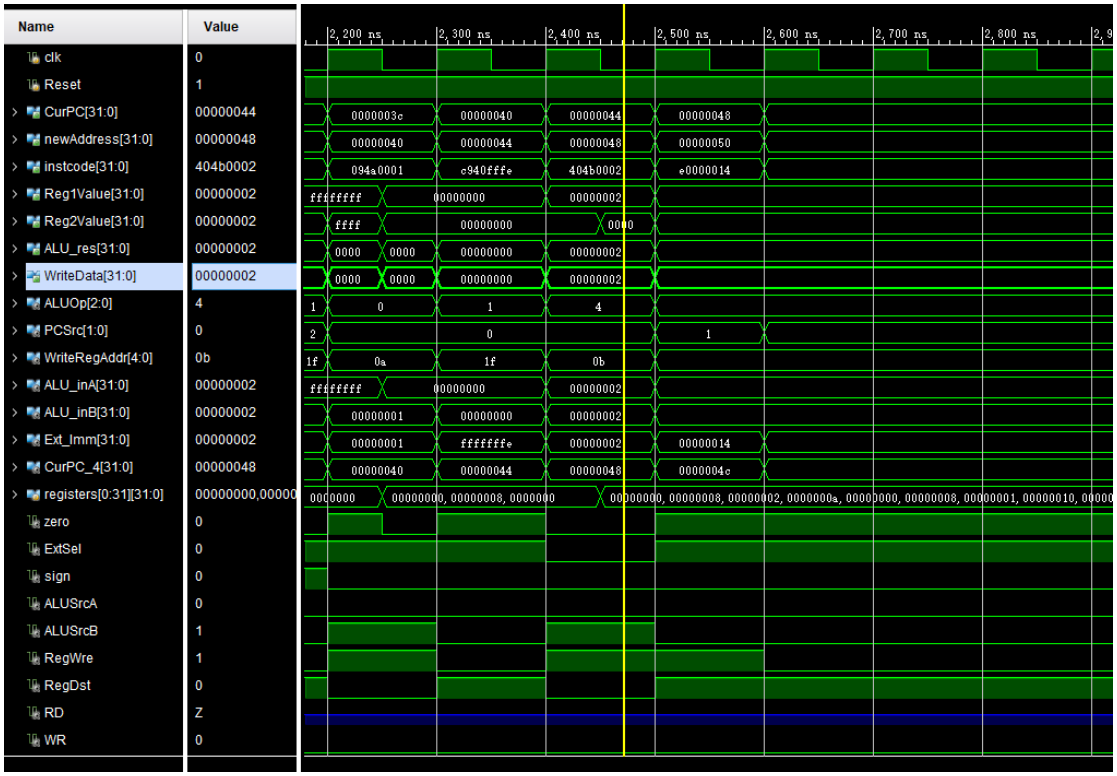
13. bltz $10, -2

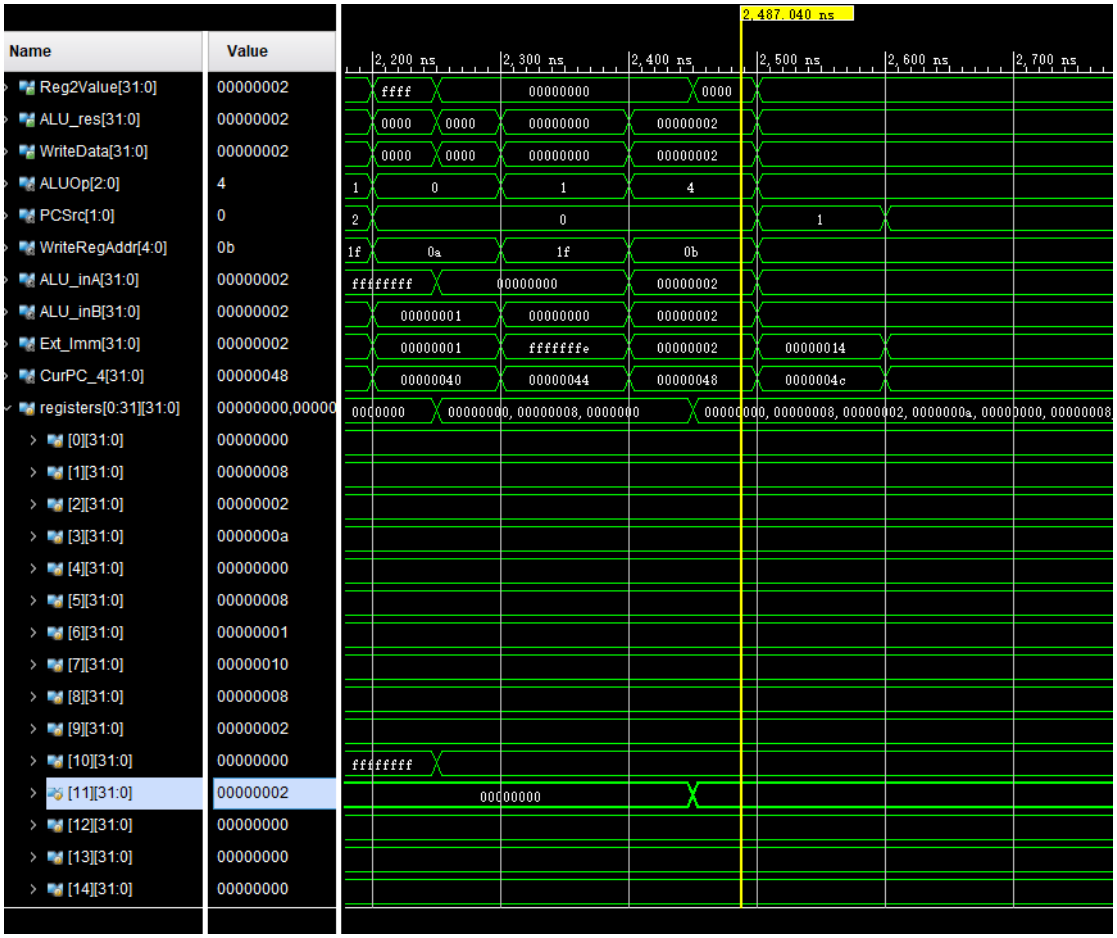
对应的 PC 地址为 CurPC=0x00000040，比较指令，则此时 ALUSrcA=0，ALUSrcB=0，ALUOp=1（001），RegWre=0，ALU_inA，ALU_inB 分别为 ffffffff(-1)、0，即$10、$0 的值，ALU_res=-1，PCSrc=2（10），CurPC_4 即 PC+4 与符号扩展且左移 2 位的立即数-2 相加，得到了下一条指令地址，作为 newAddress 的值，即 0x0000003c，正确。



14. andi $11, $2, 2

对应 PC 地址为 CurPC=0x00000044，下一条指令为 newAddress 的值，PC 正常+4，立即数与，故 ALUSrcA=0，ALUSrcB=1，ALUOp=4（100），RegDst=0，RegWre=1，ExtSel=0，立即数 2 进行 0 扩展，ALU_inA，ALU_inB 分别是 2、2，2 与 2 相与的值为 2，结果 ALU_res=2，写入寄存器$11 值为 2，正确。

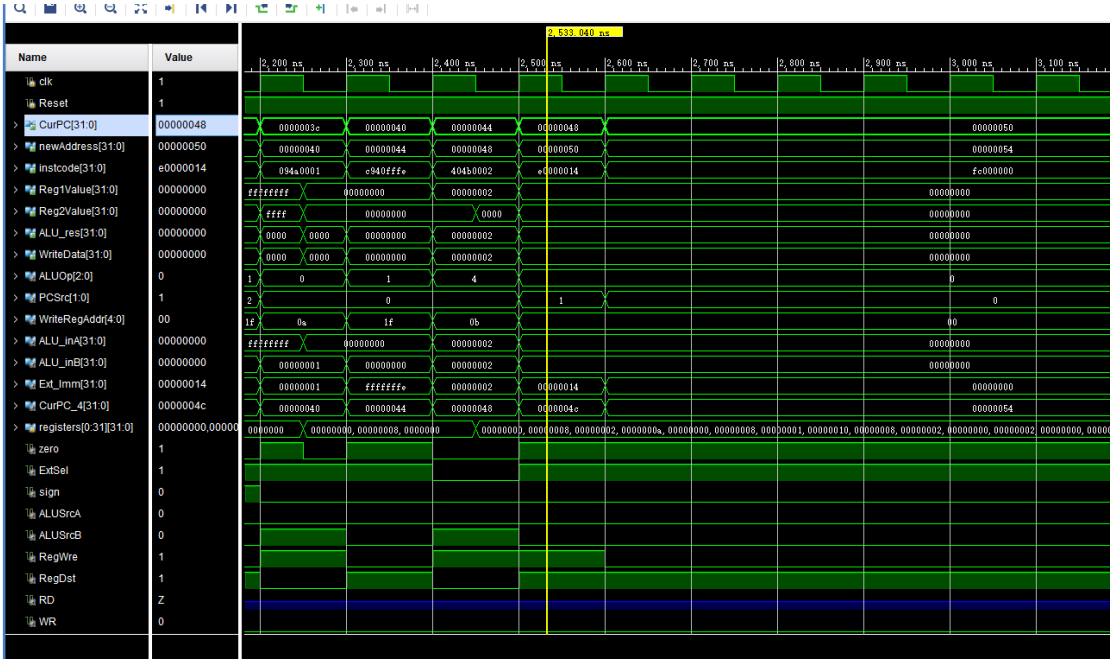

Name
Value
clk 0
Reset 1
CurPC[31:0] 00000044
newAddress[31:0] 00000048
instcode[31:0] 404b0002
Reg1Value[31:0] 00000002
Reg2Value[31:0] 00000002
ALU_res[31:0] 00000002
WriteData[31:0] 00000002
ALUOp[2:0] 4
PCSrc[1:0] 0
WriteRegAddr[4:0] 0b
ALU_inA[31:0] 00000002
ALU_inB[31:0] 00000002
Ext_Imm[31:0] 00000002
CurPC_4[31:0] 00000048
registers[0:31][31:0] 00000000,00000
zero 0
ExtSel 0
sign 0
ALUSrcA 0
ALUSrcB 1
RegWre 1
RegDst 0
RD Z
WR 0
2,200 ns
2,300 ns
2,400 ns
2,500 ns
2,600 ns
2,700 ns
2,800 ns


Name
Value
Reg2Value[31:0] 00000002
ALU_res[31:0] 00000002
WriteData[31:0] 00000002
ALUOp[2:0] 4
PCSrc[1:0] 0
WriteRegAddr[4:0] 0b
ALU_inA[31:0] 00000002
ALU_inB[31:0] 00000002
Ext_Imm[31:0] 00000002
CurPC_4[31:0] 00000048
registers[0:31][31:0] 00000000,00000
[0][31:0] 00000000
[1][31:0] 00000008
[2][31:0] 00000002
[3][31:0] 0000000a
[4][31:0] 00000000
[5][31:0] 00000008
[6][31:0] 00000001
[7][31:0] 00000010
[8][31:0] 00000008
[9][31:0] 00000002
[10][31:0] 00000000
[11][31:0] 00000002
[12][31:0] 00000000
[13][31:0] 00000000
[14][31:0] 00000000
2,487.040 ns
2,200 ns
2,300 ns
2,400 ns
2,500 ns
2,600 ns
2,700 ns

15. j 0x00000050

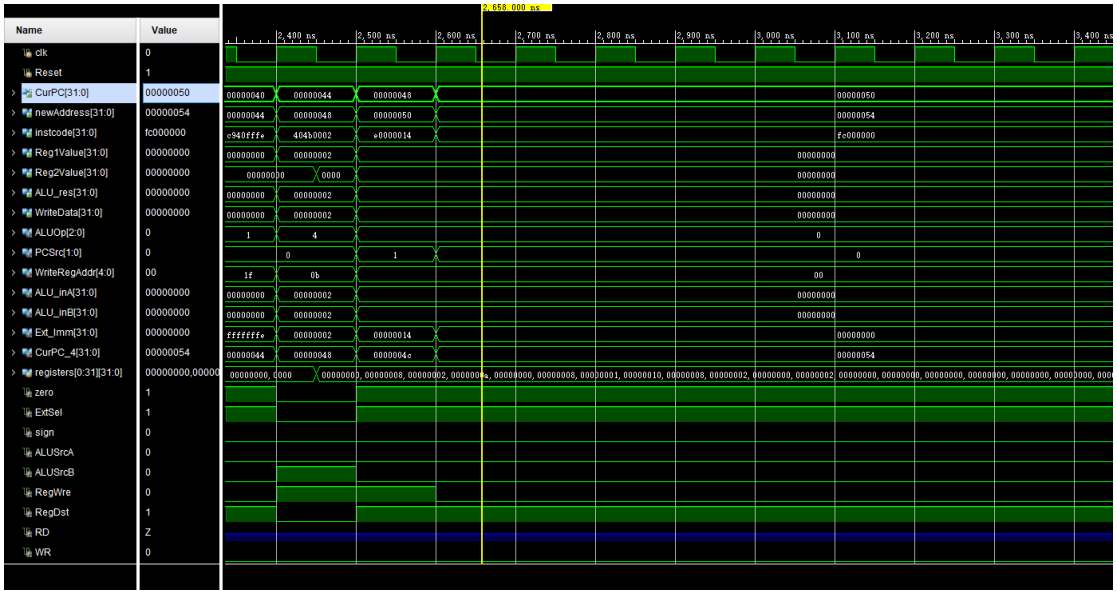
对应的 PC 地址为 CurPC=0x00000040，无条件跳转指令，则 PCSrc=01（1），CurPC_4 即 PC+4 的高四位地址与该指令中的 25:0 字段与低两位的 00 相拼接，得出 newAddress 的值为 0x00000050，正确。相应代码

```
assign newAddress = (PCSrc == 2'b01)?{CurPC_4[31:28],instcode[25:0],2'b00}:(PCSrc == 2'b10)?CurPC_4+(Ext_Imm<<2):CurPC_4;//PC更新选择
```



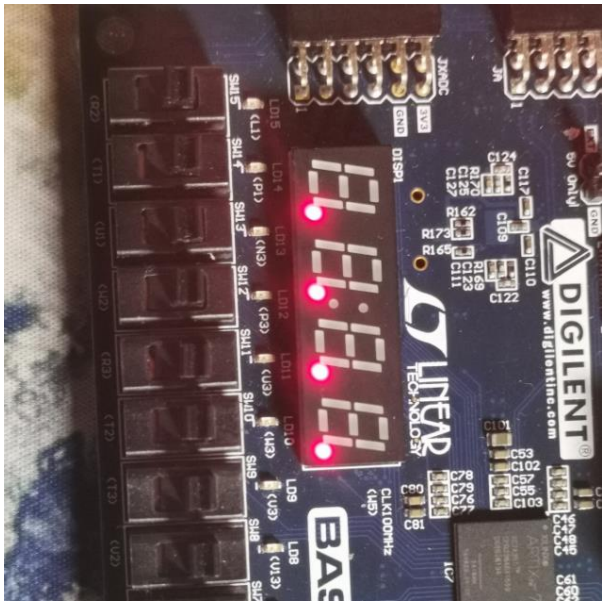
16. halt

对应 PC 地址为 CurPC=0x00000050，停机指令，故 PCWre=0，PC 即 CurPC 不再更新，所示波形图正确!



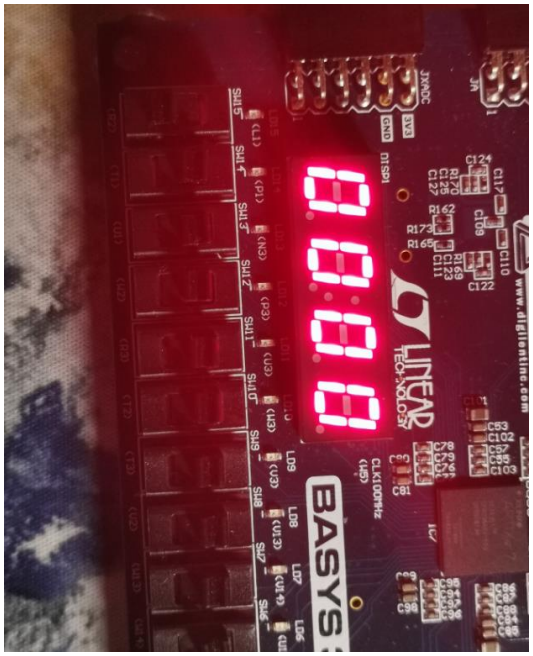
（三）在Basys3板上运行所设计的CPU

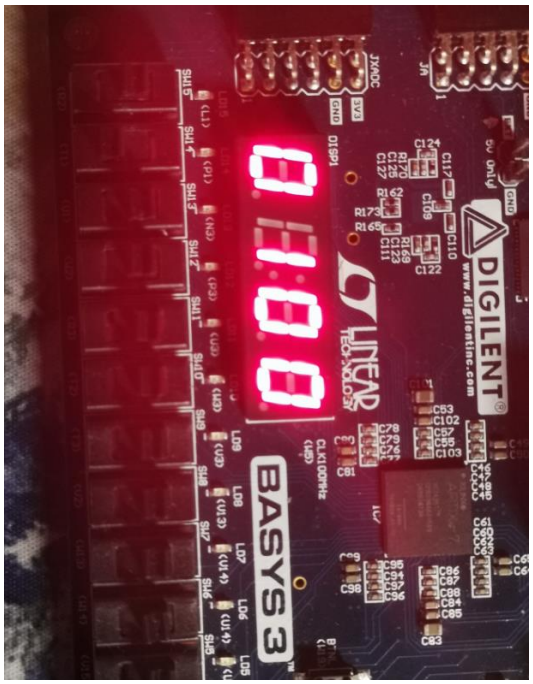
1.初始化：所有寄存器被初始化为 0

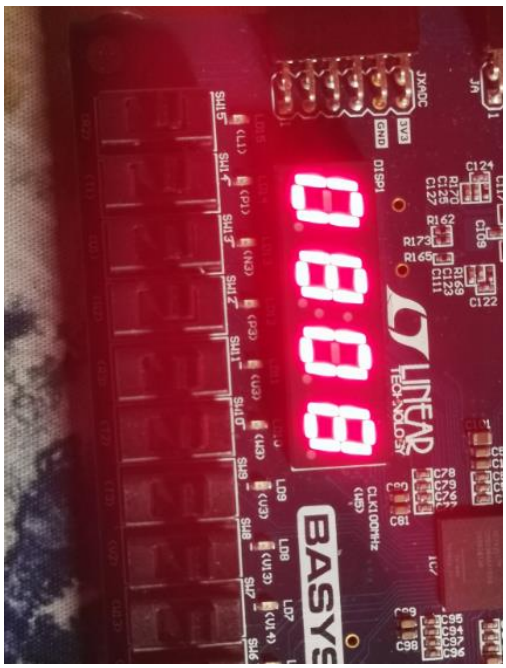


2. 第1条指令addiu $1,$0,8

当前地址为00，下一地址为04。其中，0号寄存器，值为0。1号寄存器，值为0。ALU为8。正确。

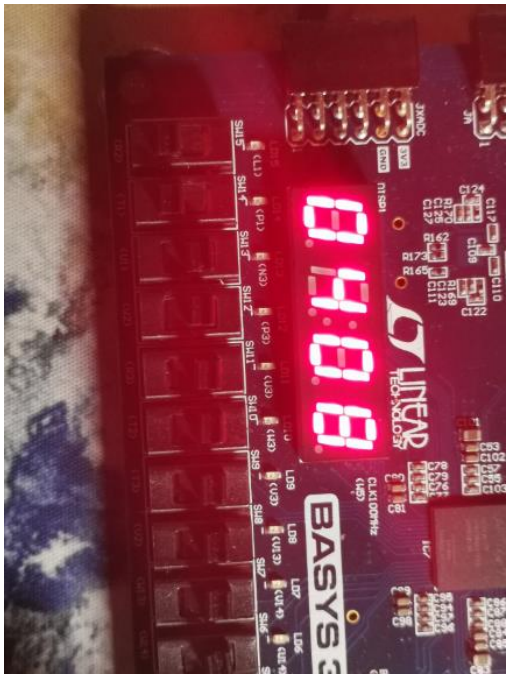

0 0 0 1
BASYS 3

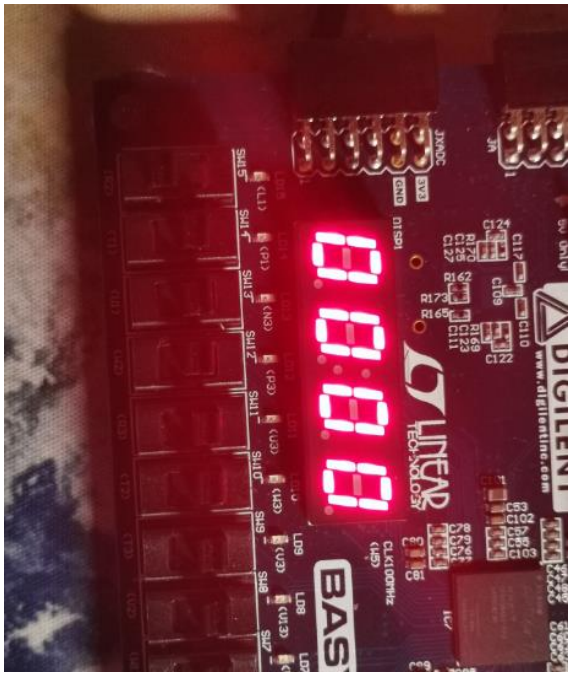

0 0 1 0
BASYS 3

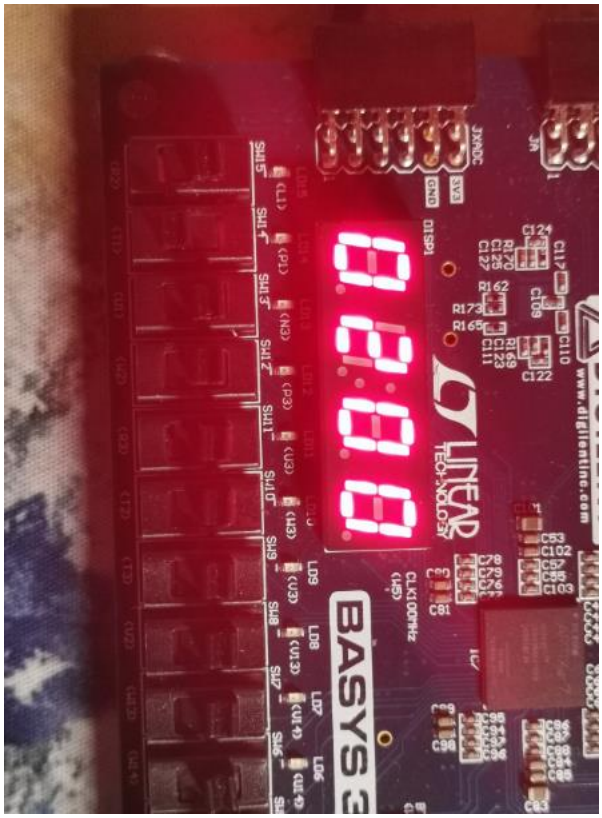


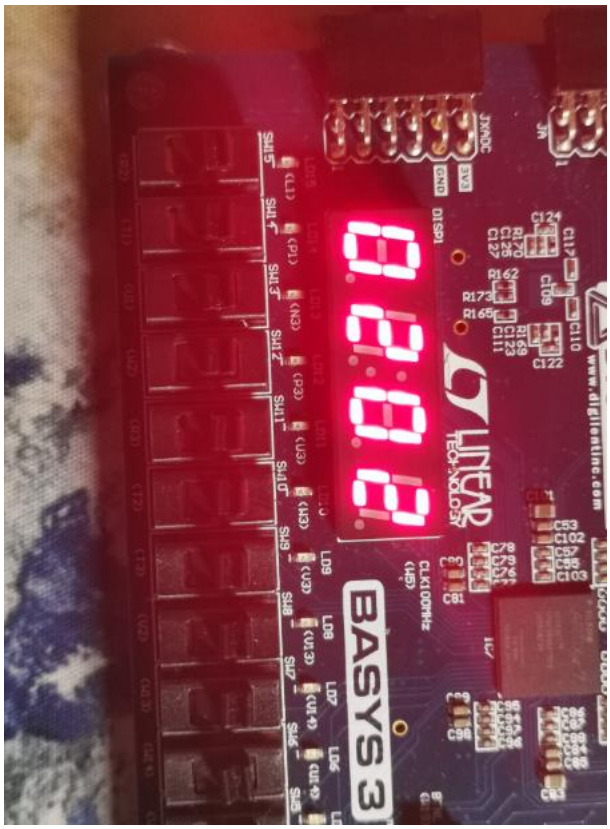
3. 第2条指令ori $2,$0,2

当前地址04，下一地址08。0号寄存器值为0。2号寄存器值为0。ALU结果为2。正确。



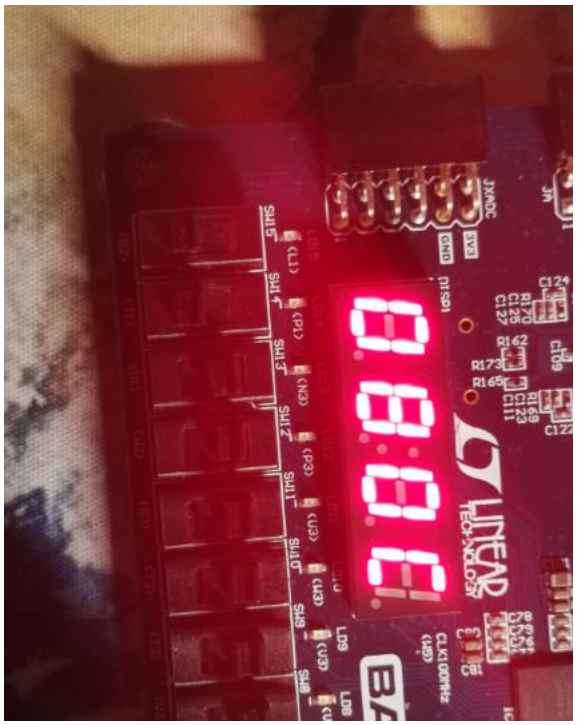


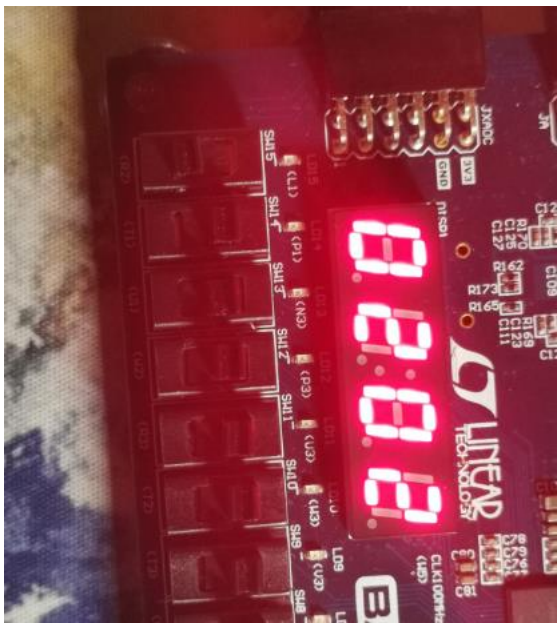


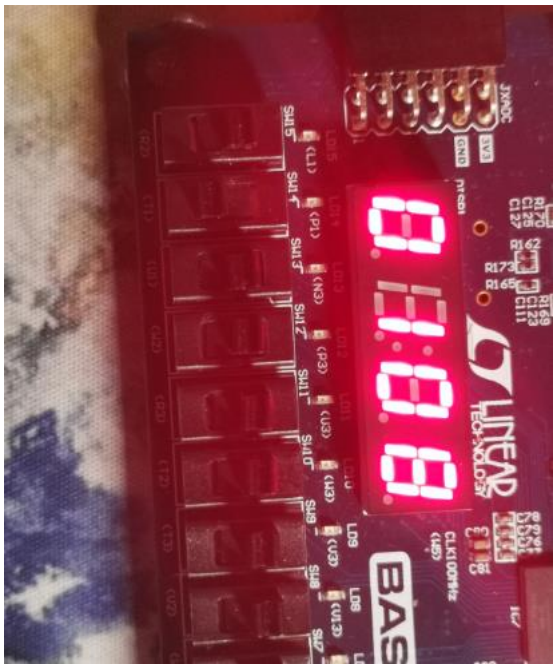


4. 第3条指令add $3,$2,$1

当前地址08，下一地址0c。2号寄存器值为2。1号寄存器值为8。输出结果为0a。正确。

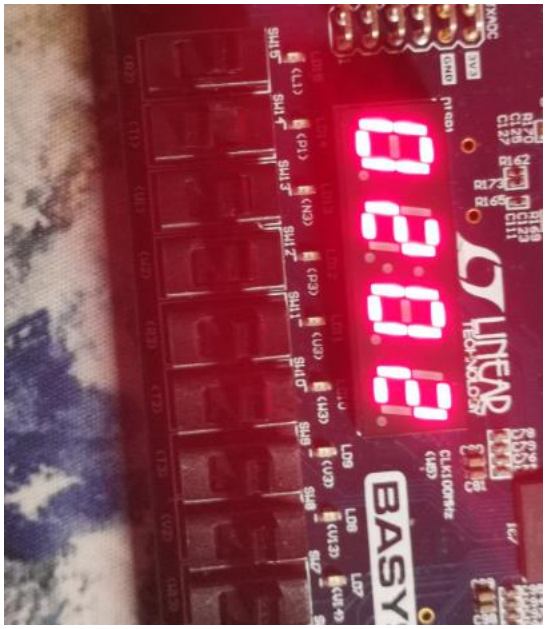






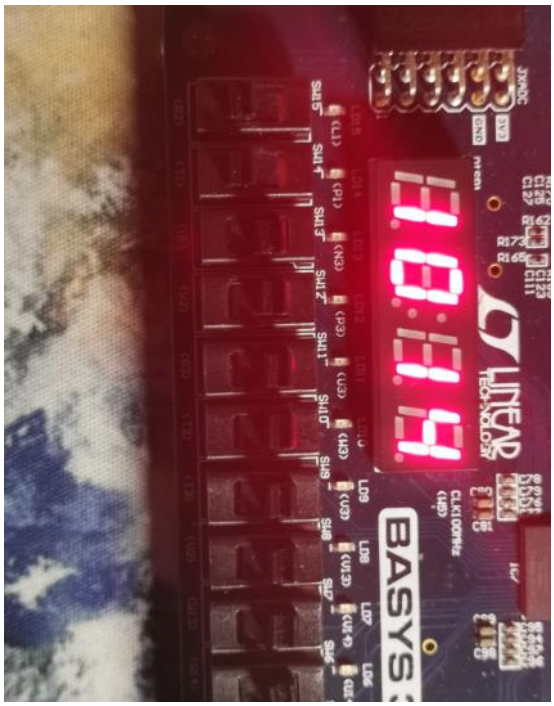
5. 第4条指令sub $5,$3,$2

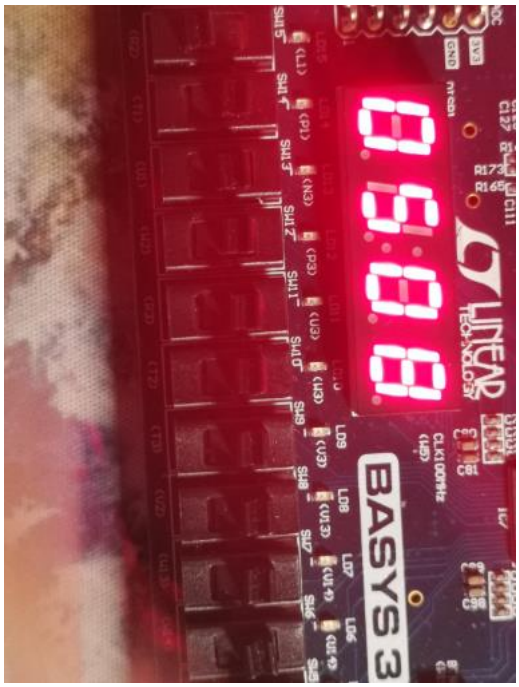
当前地址0c，下一地址10。3号寄存器值为0a。2号寄存器值为2。ALU结果为8。正确。

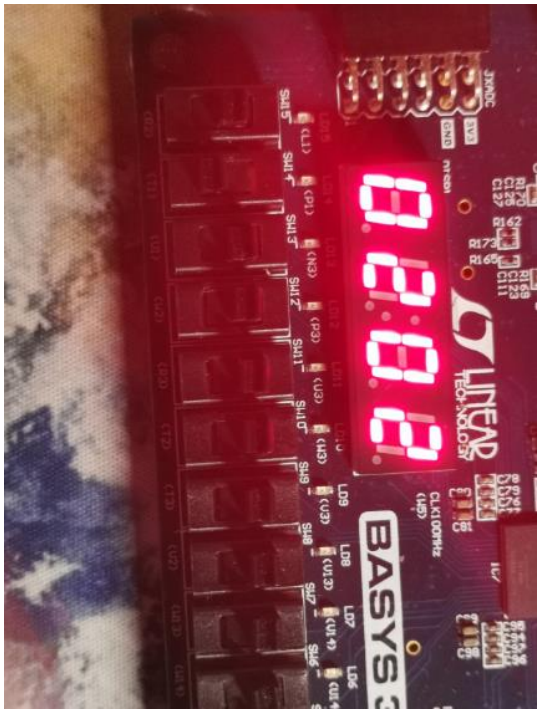


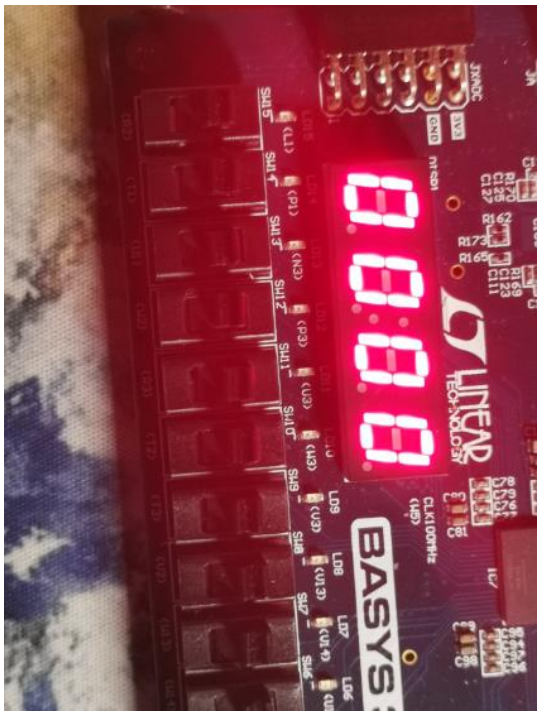
6. 第5条指令and $4,$5,$2

当前地址10，下一地址14。5号寄存器值为3。2号寄存器值为2。ALU结果为0。正确。




BASYS 3


BASYS 3



六、实验心得

本次实验是实现单周期CPU的设计，一开始我觉得很难下手，于是我通过看课本重新梳理所学相关知识，在理解了相应的理论知识后，明确了分模块实现一个单周期CPU的各个功能，最后再组合在一起，构成一个完整的单周期CPU。当专注于一个单一的模块时，其功能实现是简单的，然而，涉及将这些模块组合起来时，却会因为参数太多而显得很混乱，这就是本次实验给我的感受之一，此外也深深感受到严谨细致是设计过程不可或缺的品质。

在实现过程中碰到的问题，大多是对veriog语言的不了解、完全陌生，于是在网上找了一些视频和相关资料学习了一些verilog的基本语法知识，这才有了开展实验的基础。通过学习，我深刻了解到了verilog是一门描述硬件的语言，很多指令都是并行执行的，wire相当于导线，reg是寄存器类型，我们一般是在寄存器传输级使用verilog。

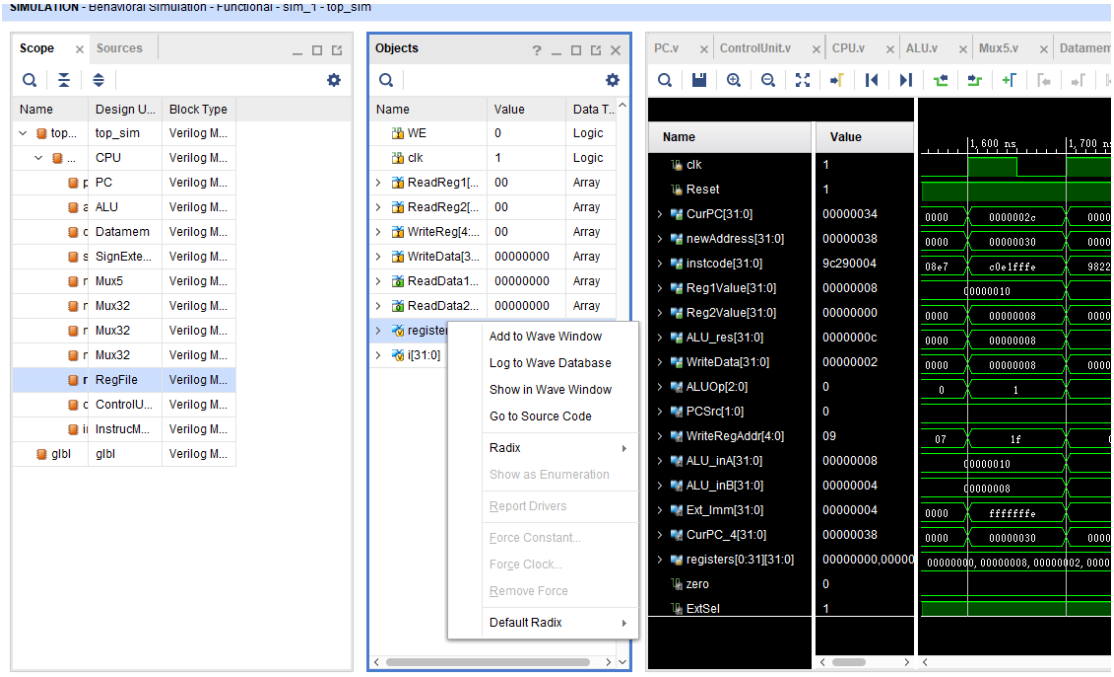
在刚开始设计时，没有重视wire和reg两种变量类型的区别，结果导致后面各种报错。意识到问题后，查找相关资料后归纳总结了wire和reg两种变量类型在使用过程中的区别：

wire主要起信号间连接的作用，例如顶层模块中，需要将各个模块连接起来，这时候只能用wire连接，不能使用reg，wire不保存状态，它的值的随时可以改变，不受时钟信号的影响。而reg则是寄存器的抽象表达，可以用于存储数值，例如指令寄存器和寄存器组以及数据寄存器里面的存储器必须为reg类型，用于保留数据。其次wire类型只能通过assign进行赋值，而reg类型只能在always里面被赋值。另外，reg类型在always中有阻塞赋值和非

阻塞赋值两种方式，当敏感信号为电平信号的时候，采用阻塞赋值(=),而敏感信号为时序信号时，采用非阻塞赋值(<=)。

此外还有debug的问题，因为不熟悉vivado的使用方法，只能肉眼debug，比如开始时仿真失败，最后发现是读入测试指令文件tests.txt的代码写错，readmemd写成了readmed，漏了一个m，等到修改之后，仿真终于成功，却发现波形图不符合预期，最后是在0x00000018、0x0000001c这两条指令来回跳转，通过波形图对第一条指令的执行过程进行分析，我思考会不会是控制单元的某一信号出现了逻辑错误，于是我对控制单元模块进行检查，果不其然，ALUSrcA这一仅在sll指令有效的信号，我写成仅在sll时无效，修改过后终于仿真成功，符合预期结果。

此外，一开始仿真时还遇到了相应关键信息的值如寄存器堆的值没有显示在波形图上等细节问题，这时我通过网上查资料，才知道可以在scope处将相应的值显示到波形图上，如下图，选择Add to Wave Window，重新仿真，就能显示了。



紧接着在烧板的时候，出现了下面的错误：

一开始分析是不是约束问题，代码有违背可综合规则，模块功能问题什么的，都没用。后来通过在网上查找资料发现，如果层模块的接口只有输入信号：比如时钟和复位，没有任何输出，在这种情况下，vivado在进行综合实现时便会将你内部逻辑全部优化掉，所以便

会出现错误：design is empty。而实际上，按照合理的设计顶层模块肯定是会有信号输出的，所以只要有输入输出信号，便不会出现上述错误。了解了问题所在后我针对问题进行修改，将顶层文件有原来的single circle cpu改为了basys3，增加了其输出信号，最终问题得已成功解决。

总的来说，这次实验对我而言是一个比较困难的实验，从开始接触到最后烧板成功的过程仿佛是西天取经了一趟，困难接踵而至。但真的做完以后，也收获了很多。在理论课上学习了关于单周期CPU的设计，但学完之后总感觉浮光掠影般，不能深刻理解。这次实验将理论应用于实践，加深了印象。同时也掌握了更多关于verilog的一些基本知识，对自顶往下，逐层设计的分模块设计方法也有了更为深刻的理解，也算是在困难中有所成长吧。

最后，我想提的建议就是希望以后老师能提供一些关于verilog语言的参考资料或者上课时讲一下verilog相关的基本知识，因为不熟悉verilog是造成实验难以进行的困难之一。而且上学期由于疫情原因没有接触过basys3板，对烧板过程中的具体细节与问题也没有什么处理经验。希望能够在这些方便得到提高。