中山大學
SUN YAT-SEN UNIVERSITY

《计算机组成原理实验》

实验报告

（实验二）

学 院 名 称	：	数据科学与计算机学院
专业（班级）	：	17 软工 2 班
学 生 姓 名	：	张莉斌
学　　号	：	16340290
时　　间	：	2018　年　11　月　18　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；
5. 掌握单周期CPU的实现方法。

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)	reserved

功能：rd←rs + rt。reserved 为预留部分，即未用，一般填“0”。

（2）sub rd , rs , rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt。

（3）addiu rt , rs ,immediate

000010	rs(5 位)	rt(5 位)	immediate(16 位)

功能：rt←rs + (sign-extend)immediate；immediate 符号扩展再参加“加”运算。

==> 逻辑运算指令

（4）andi rt , rs ,immediate

010000	rs(5 位)	rt(5 位)	immediate(16 位)

功能：rt←rs & (zero-extend)immediate；immediate 做“0”扩展再参加“与”运算。

（5）and rd , rs , rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs & rt；逻辑与运算。

（6）ori rt , rs ,immediate

010010	rs(5 位)	rt(5 位)	immediate(16 位)

功能：rt←rs | (zero-extend)immediate；immediate 做“0”扩展再参加“或”运算。

（7）or rd , rs , rt

010011	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

==>移位指令

（8）sll rd, rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa(5 位)	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa。

==>比较指令

（9） slti rt, rs,immediate 带符号数

011100	rs(5 位)	rt(5 位)	immediate(16 位)

功能：if (rs< (sign-extend)immediate) rt =1 else rt=0, 具体请看表 2 ALU 运算功能表，带符号。

==> 存储器读/写指令

（10）sw rt ,immediate(rs) 写存储器

100110	rs(5 位)	rt(5 位)	immediate(16 位)

功能：memory[rs+ (sign-extend)immediate]←rt；immediate 符号扩展再相加。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

（11）lw rt , immediate(rs) 读存储器

100111	rs(5 位)	rt(5 位)	immediate(16 位)

功能：rt ← memory[rs + (sign-extend)immediate]；immediate 符号扩展再相加。

即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

==> 分支指令

（12）beq rs,rt,immediate

110000	rs(5 位)	rt(5 位)	immediate(16 位)

功能：if(rs=rt) pc←pc + 4 + (sign-extend)immediate <<2 else pc ←pc + 4

特别说明：immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。immediate 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 immediate 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,immediate

110001	rs(5 位)	rt(5 位)	immediate(16 位)

功能：if(rs!=rt) pc←pc + 4 + (sign-extend)immediate <<2 else pc ←pc + 4

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

（14）bltz rs,immediate

110010	rs(5 位)	00000	immediate(16 位)

功能：if(rs<$zero) pc←pc + 4 + (sign-extend)immediate <<2 else pc ←pc + 4。

==>跳转指令

（15）j addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，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。

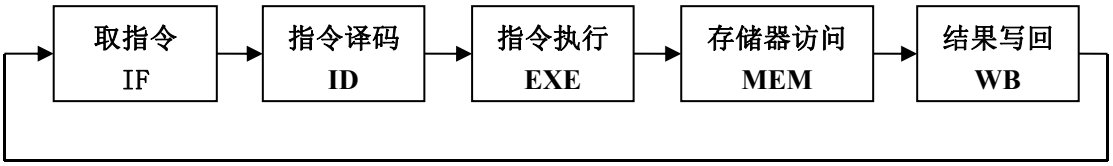


图 1　单周期 CPU 指令处理过程

MIPS 指令的三种格式：

R 类型:

31	2625	2120	1615	1110	65	0
op	rs	rt	rd	sa	funct	
6 位	5 位	5 位	5 位	5 位	6 位	

I 类型:

31	2625	2120	1615	0
op	rs	rt	immediate	
6 位	5 位	5 位	16 位	

J 类型:

31	2625	0
op	address	
6 位	26 位	

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。

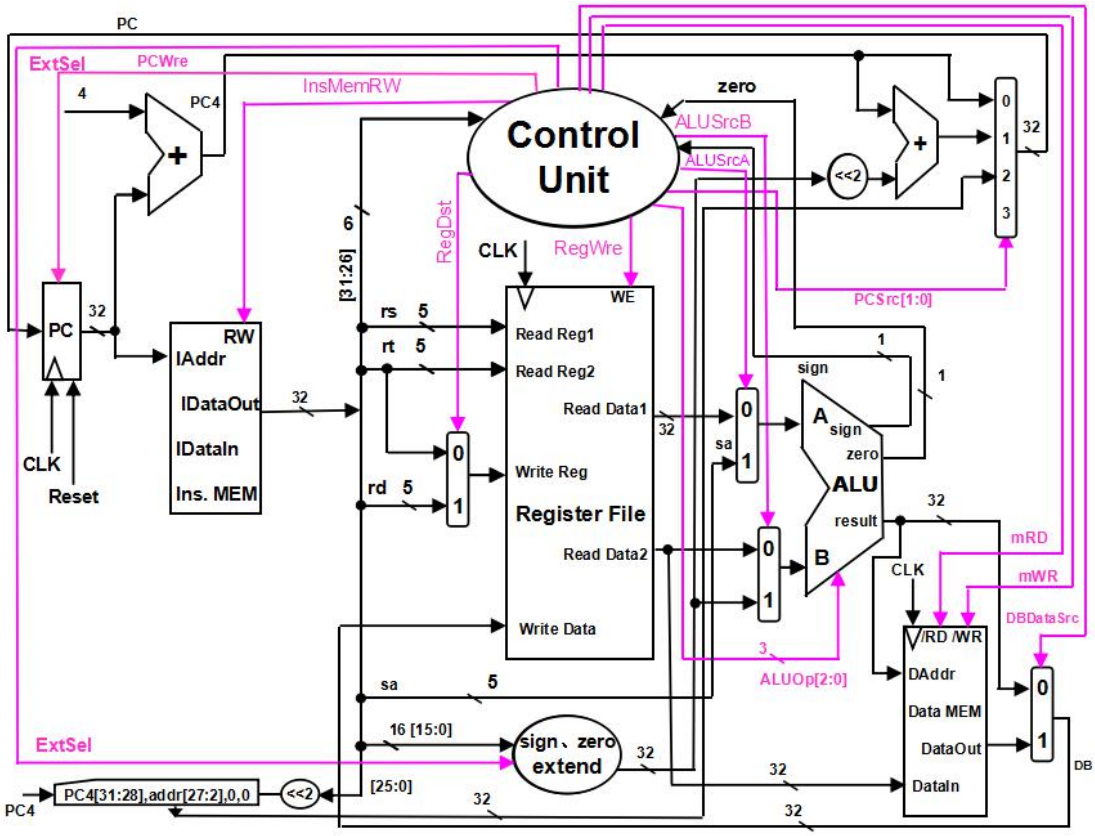


图 2　单周期 CPU 数据通路和控制线路图

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

相关部件及引脚说明：

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：andi、ori	(sign-extend)immediate（符号扩展），相关指令：addiu、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<<2，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	

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<B 不带符号
110	Y=(((A<B)&&(A[31] == B[31])) ||((A[31]==1&& B[31] == 0))) ? 1:0	比较 A<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。

指令执行的结果总是在时钟下降沿保存到寄存器和存储器中，PC 的改变是在时钟上升沿进行的，这样稳定性较好。另外，值得注意的问题，设计时，用模块化的思想方法设计，关于 ALU 设计、存储器设计、寄存器组设计等等，也是必须认真考虑的问题。

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

1.设计的方法、思想

设计CPU的基本思想是自顶向下，分模块实现。先完成数据通路，把数据通路的每个模块功能所需的输入输出端口统一。数据通路图2所示，接下来就是各个模块的实现。再用顶层文件完成各个模块的连接。

最终结构：

（1）PC模块

PC
Input： CLK 时钟 Reset 重置 PCWre 读取使能 PB 返回地址

PC是一个CPU的指挥部，需要用CLK时钟来保证它的有序行进，还需要配上Reset来重置它和PCWre来执行停机指令

```
module PC(
    input CLK,          //时钟
    input Reset,        //为0时重置
    input PCWre,        //为0时停机，PC不更改
    input [31:0] inputAddress,//下一个地址
    output reg [31:0] outputAddress//当前地址
    );

    initial begin
        outputAddress <= 0;//初始化PC地址为0
    end

    always@(posedge CLK or negedge Reset) begin
        if (Reset == 0) outputAddress <= 0;
        else begin
            if (PCWre == 1) outputAddress <= inputAddress;
            else if(PCWre == 0)
                outputAddress = outputAddress;//地址不变
        end
    end
Endmodule
```

（2）ROM 指令存储模块

ROM
Input： PC 当前地址 Output： Ins 指令代码 Memory： rom[0:127] 指令存储

ROM的实现是关键在于初始化写入指令和读取的实现。其中初始换使用$readmemb指令，找到写有指令的文本文件，如 $readmemb ("C:/Users/Administrator/Desktop/CPU/instruction.txt", rom); // 数据文件 rom_data（.coe或.txt）。未指定，就从0地址开始存放。

读取指令则使用大端方式，即指令高位在存储的地址小端。同时根据always@来不断实现读取。

```
initial begin // 加载数据到存储器rom。注意：必须使用绝对路径，如：
E:/Xlinx/VivadoProject/ROM/（自己定）
    $readmemb ("C:/Users/Administrator/Desktop/CPU/instruction.txt", rom);
    IDataOut = 0;
end
always @( InsMemRW or IAddr ) begin
        IDataOut[31:24] = rom[IAddr];
        IDataOut[23:16] = rom[IAddr+1];
        IDataOut[15:8] = rom[IAddr+2];
        IDataOut[7:0] = rom[IAddr+3];
End
```

（3）CtrlUnit控制单元模块

CtrlUnit
Input： Opcode 操作码 Output： PCWre PC 读取使能 ALUSrcA 操作数 1 选择 ALUSrcB 操作数 2 选择 DBDataSrc 写会数据选择 RegWre 寄存器写入使能 InsMemRW 指令读取使能 RD 存储器读取使能 WR 存储器写入使能 RegDst 目的寄存器选择 ExtSel 扩展选择 PCSrc PC 选择 ALUOp 运算选择

控制单元是不仅是所有模块的控制中心，还是联通所有模块的重要通路。通过六位操作码得到12个控制输出，有两种实现方法，一种是通过操作码内的运算确定各个控制输出，一种是根据每个操作码来确定控制输出，通过case实现。本次实验室通过后者来实现。部分代码如下：

```
initial begin
        PCWre = 1;
        ALUSrcA = 0;
        ALUSrcB = 1;
        DBDataSrc = 0;
        RegWre = 1;
        InsMemRW = 0;
        mRD = 1;
        mWR = 1;
        RegDst = 0;
        ExtSel = 0;
```

```
        PCSrc = 0;
        ALUOp = 0;
    End

    always @(zero or op) begin
        case (op)
             6'b000000:  //1 add指令
        begin
            PCWre = 1;
            ALUSrcA = 0;
            ALUSrcB = 0;
            DBDataSrc = 0;
            RegWre = 1;
            InsMemRW = 1;
            mRD = 1;
            mWR = 1;
            RegDst = 1;
            ExtSel = 1;
            PCSrc = 2'b00;
            ALUOp = 3'b000;
        end
```

除了要考虑操作码，控制输出还与zero、sign标志位有关，所以always@语句中加入标志位zero 和 sign。与标志位相关的是跳转指令相关的PCSrc，实现方式用if语句判断标志位，代码如下：

```
             ExtSel = 1;
             if(zero) PCSrc = 2'b00; else PCSrc = 2'b01;
             ALUOp = 1;
```

（4）RegFile寄存器模块

RegFile
Input： CLK RegWre 寄存器写能 ReadReg1 读寄存器号 ReadReg2 读寄存器号 DB 写入的数据 Output： ReadData1 ReadData2 Memory： ram[31:1] 31 个寄存器

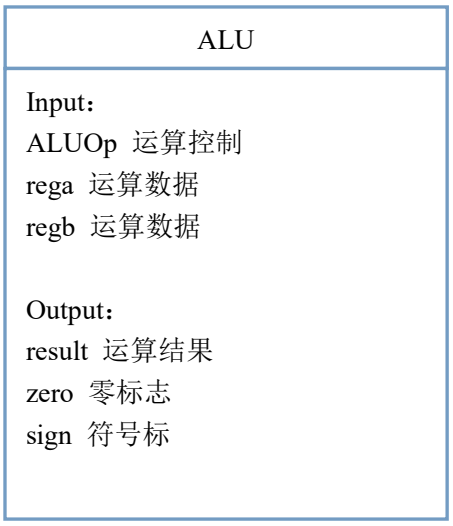
代码入下：

```
reg [31:0] regFile[0:31]; // 寄存器定义必须用reg 类型
 integer i;
 initial begin
     for (i = 0; i < 32; i = i + 1) regFile[i] = 0;
 end

 assign ReadData1 = (!ReadReg1) ? 0 : regFile[ReadReg1]; // 读寄存器
数据
 assign ReadData2 = (!ReadReg2) ? 0 : regFile[ReadReg2];
 always @ (negedge CLK) begin // 必须用时钟边沿触发
     if(RegWre && WriteReg) // WriteReg != 0，$0 寄存器不能修改
         regFile[WriteReg] = WriteData; // 写寄存器
 end
```

零号寄存器不用分配存储空间，当寄存器号为零时直接输出零即可。寄存器写入需要时钟来控制，用下降沿触发来避免与PC读取相冲突，而写入寄存器时又不会写入存储单元，这样两种写操作都可以用下降沿触发而不冲突。

（5）ALU运算单元



该模块的标志位可以用三目运算符来直接选择，至于运算的操作可以用case语句来实现选择。部分代码如下

```
    assign zero = (result == 0) ? 1 : 0;
    assign sign = (result[31] == 0) ? 1 : 0;
always @( ALUopcode or rega or regb ) begin
        case (ALUopcode)
            3'b000 : result = rega + regb;
            3'b001 : result = rega - regb;
            3'b010 : result = regb << rega;
            3'b011 : result = rega | regb;
            3'b100 : result = rega & regb;
            3'b101 : result = (rega < regb)?1:0; // 不带符号比较
            3'b110 : begin                       // 带符号比较
                if(rega < regb && (rega[31] == regb[31]))result = 1;
                else if (rega[31] == 1 && regb[31] == 0) result = 1;
                else result = 0;
                end
            3'b111 : result = rega ^ regb;
            default : begin
                result = 32'h00000000;
                $display (" no match");
```

End

对ALUopcode 的判断是根据需求的指令来决定的，老师给出的指令基本上都覆盖了MIPS指令类型，所以直接把所有的操作都写出来不用针对的单独某一指令考虑了

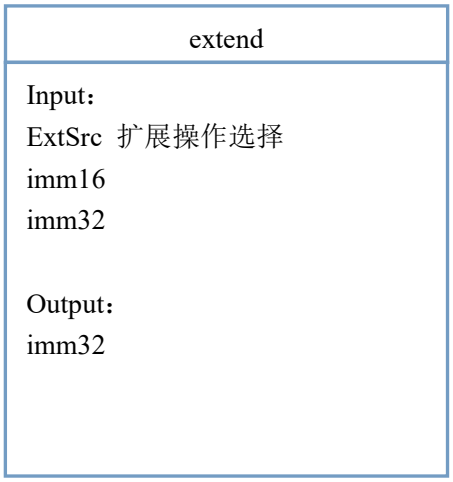
（6）RAM 数据存储模块

RAM
Input： RD 读使能 WR 写使能 writeData 写入数据 Output： DataOut 读取数据 Memory： ram[0:60] 数据存储

该模块存储数据，所以首先赋予一定的存储空间。如果nRD为1是输出高阻态，直接用assign语句可以实现。而数据写入时必须使用时钟下降沿触发，这样可以避免同时写入不同数据不同地址。代码如下：

```
reg [7:0] ram [0:60]; // 存储器定义必须用reg类型
    // 读
    assign Dataout[7:0] = (nRD==0)?ram[DAddr + 3]:8'bz; // z 为高阻态
    assign Dataout[15:8] = (nRD==0)?ram[DAddr + 2]:8'bz;
    assign Dataout[23:16] = (nRD==0)?ram[DAddr + 1]:8'bz;
    assign Dataout[31:24] = (nRD==0)?ram[DAddr ]:8'bz;
    // 写
    always@( negedge CLK ) begin // 用时钟下降沿触发写存储器，个例
        if( nWR==0 ) begin
            ram[DAddr] <= DataIn[31:24];
            ram[DAddr + 1] <= DataIn[23:16];
            ram[DAddr + 2] <= DataIn[15:8];
            ram[DAddr + 3] <= DataIn[7:0];
        end
```

（7）extend 扩展模块



扩展方式有两种，零扩展和符号为扩展，可以用三目选择运算符实现，也可以用case实现，下面的代码使用的是三目运算符。在JUMP指令连接时直接使用拼接符{ }来实现。重点在于符号扩展可以用 16 {imm[15]}来做符号位。

```
assign extendImmediate[15:0] = immediate;
    assign extendImmediate[31:16] = (ExtSel == 1) ? (immediate[15] ? 16'hffff : 16'h0000) : 16'h0000;
```

（8）余下的几个选择器都差不多，只拿其中一个来举例：

这是ALU模块的输入A一部分，比B多了一步操作就是0扩展，因为输入的SA是五位的

```
module Multiplexer32ForA(       //用于选择ALU的第一个输入的2选1选择器
    input select,
    input [31:0]DataIn1,
    input [4:0]DataIn2,          //输入的sa只有5位，要做0扩展
    output [31:0]DataOut
    );

    wire [31:0]extendSa;
     assign extendSa = {27'b0, DataIn2};

    assign DataOut = (!select) ? DataIn1:extendSa  ;
endmodule
    assign B =(!ALUSrcB ) ? ReadData2 : extendImmediate;
```

endmodule

（9）其他模块差不多都和上面的类型差不多,实现方法一样,用三目选择运算符或case选择语块实现。

2.验证CPU的正确性

先列出测试程序的代码表格，再写出程序的测试文件，进行仿真，得到仿真波形图，通过分析波形图中当前PC值和返回PC值、指令、控制单元、寄存器值和存储器值等来判断程序的争取性。测试文件中时钟和置位端口初始化及行进模块如下：

```
initial begin
   CLK = 0;
   Reset = 0;
   #50;
    CLK = 1;
   #50;
    Reset = 1;

   forever #50 begin
    CLK = !CLK;
    end
End
```

由此可得到理想的波形图。

顶层文件要把数据通路图里所有的线都声明一遍，然后各个再初始化

如下

```
wire [2:0]ALUOp;
wire [1:0]PCSrc;
wire [31:0]address_immediate, jAddress, newAddress;
wire [31:0]extendImmediate;
wire [4:0]WriteReg;
wire [31:0]A, B;
wire zero,sign, PCWre, ALUSrcA, ALUSrcB, DBDataSrc, RegWre, InsMemRW, mRD, mWR, RegDst, E

assign opcode = instruction[31:26];
assign rs = instruction[25:21];
assign rt = instruction[20:16];
assign rd = instruction[15:11];

ControlUnit ControlUnit(zero,sign, instruction[31:26], PCWre, ALUSrcA, ALUSrcB, DBDataSrc
PC PC(CLK, Reset, PCWre, newAddress, PCAddress);
PC4 PC4(PCAddress,PC4Address);
InstrucionMemory InstrucionMemory(InsMemRW, PCAddress, instruction);
RegFile RegFile(CLK, RegWre, instruction[25:21], instruction[20:16], WriteReg, DB, ReadDa
ALU ALU(ALUOp, A, B, ALUresult, zero,sign);
SignZeroExtend SignZeroExtend(ExtSel, instruction[15:0], extendImmediate);
DataMemory DataMemory(CLK, ALUresult, ReadData2, mRD, mWR, DataOut);
Multiplexer5 Multiplexer5(RegDst, instruction[20:16], instruction[15:11], WriteReg);
Multiplexer32ForA Multiplexer32ForA(ALUSrcA, ReadData1, instruction[10:6], A);
Multiplexer32ForB Multiplexer32ForB(ALUSrcB, ReadData2, extendImmediate, B);
Multiplexer32ForDB Multiplexer32ForDB(DBDataSrc, ALUresult, DataOut, DB);
JUMP JUMP(PC4Address, instruction[25:0], jAddress);
Multiplexer32For4 Multiplexer32For4(PCSrc, PC4Address, extendImmediate, jAddress, newAddr
```

然后就是测试，首先需要明确的是实现的指令有：add sub addiu andi and ori or sll slti sw lw beq bne bltz j halt 这些指令，然后根据这些指令得出控制信号与指令之间的关系表如下：

控制信号与指令关系表

	控制信号												
指令	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMemRW	RD	WR	RegDst	ExtSel	PCSrc	ALUOp	OP
add	1	0	0	0	1	1	1	1	1	1	00	000	
andi	1	0	1	0	1	1	1	1	0	0	00	000	
addiu	1	0	1	0	1	1	1	1	0	1	00	000	
sub	1	0	0	0	1	1	1	1	1	1	00	001	
ori	1	0	1	0	1	1	1	1	0	0	00	011	
and	1	0	0	0	1	1	1	1	1	1	00	100	
or	1	0	0	0	1	1	1	1	1	1	00	011	

sll	1	1	0	0	1	1	1	1	1	1	00	010	
slti	1	0	1	0	1	1	1	1	0	1	00	110	
sw	1	0	1	0	0	1	1	0	1	1	00	000	
lw	1	0	1	1	1	1	0	1	0	1	00	000	
beq	1	0	0	0	0	1	1	1	1	1	0zero	001	
bne	1	0	0	0	0	1	1	1	1	1	0!zero	001	
bltz	1	0	0	0	0	1	1	1	1	1	0sign	001	
j	1	0	0	0	0	1	1	1	1	1	10	000	
halt	0	0	0	0	0	1	1	1	1	1	00	000	

通过细致的分析建立汇编程序指令顺序，再据此得出指令代码，包括二进制代码和十六进制代码，复制粘贴到程序初始化RAM时所访问的文档文件。

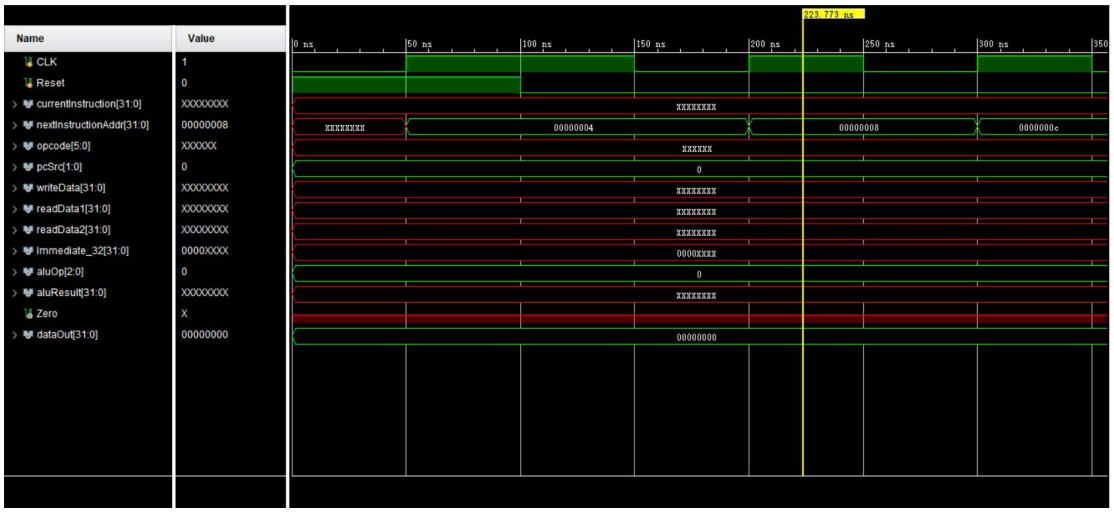
然后将需要的测试文件写好：

地址		汇编程序	指令代码						结果
			op（6）	rs(5)	rt(5)	rd(5)/immediate (16)		16 进制数代码	
1	0x00000000	addiu $1,$0,8	000010	00000	00001	0000 0000 0000 1000	=	08010008	2
2	0x00000004	ori $2,$0,2	010010	00000	00010	0000 0000 0000 0010	=	48020002	10
3	0x00000008	add $3,$2,$1	000000	00010	00001	0001 1000 0000 0000	=	411800	8
4	0x0000000C	sub $5,$3,$2	000001	00011	00010	0010 1000 0010 0010	=	4622822	0
5	0x00000010	and $4,$5,$2	010001	00101	00010	0010 0000 0010 0100	=	44a22024	2
6	0x00000014	or $8,$4,$2	010011	00100	00010	0100 0000 0010 0101	=	4c824025	4,8
7	0x00000018	sll $8,$8,1	011000	00000	01000	0100 0000 0100 0000	=	60084040	4,8
8	0x0000001C	bne $8,$1,-2 (≠,转18)	110001	00001	01000	1111 1111 1111 1110	=	c428fffe	1
9	0x00000020	slti $6,$2,4	011100	00010	00110	0000 0000 0000 0100	=	70460004	0

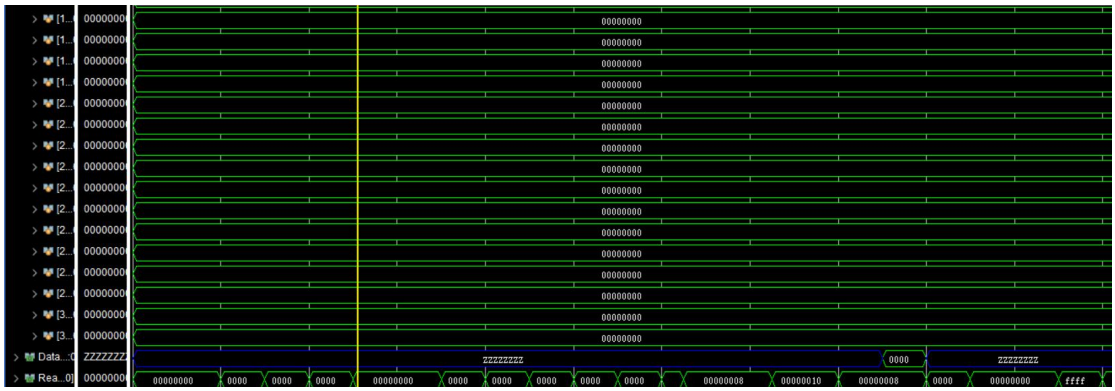
10	0x00000024	slti $7,$6,0	011100	00110	00111	0000 0000 0000 0000	=	70c70000	8,16
11	0x00000028	addiu $7,$7,8	000010	00111	00111	0000 0000 0000 1000	=	8e70008	8,16
12	0x0000002C	beq $7,$1,-2 (=,转 28)	110000	00111	00001	1111 1111 1111 1110	=	c0e1ffffe	
13	0x00000030	sw $2,4($1)	100110	00001	00010	0000 0000 0000 0100	=	98220004	2
14	0x00000034	lw $9,4($1)	100111	00001	01001	0000 0000 0000 0100	=	9c290004	0
15	0x00000038	addiu $10,$0,-2	000010	00000	01010	1111 1111 1111 1110	=	80afffe	
16	0x0000003C	addiu $10,$10,1	000010	01010	01010	0000 0000 0000 0001	=	94a0001	
17	0x00000040	bltz $10,-2(<0 ,转 3C)	110010	00000	01010	1111 1111 1111 1110	=	c80afffe	
18	0x00000044	andi $11,$2,2	010000	00010	01011	0000 0000 0000 0010	=	404b0002	
19	0x00000048	j 0x00000050	111000	00000	00000	0000 0000 0001 0100	=	e0000014	
20	0x0000004C	or $8,$4,$2	010011	00100	00010	0100 0000 0010 0101	=	4c824025	
21	0x00000050	halt	111111	00000	00000	0000 0000 0000 0000	=	fc000000	

3，**根据数据通路图写好各小模块，最后加上顶层模块test,开始调试波形debug**

一开始波形全是高阻态~



我把每个小模块都仔细的看了几遍，就是找不到什么逻辑上的问题，所以猜测是顶层文件出了问题，然后检查顶层模块的写法时发现有一些（挺多的）小BUG，修改后有了部分的波形，都不再是高阻态，但是有些有数据，有些是未知X和全是0。



但是肉眼并不能确定全是0是对还是错，继续修改。

而且一直有个问题是，ROM 模块的输出应该是我编写的测试代码的十六进制形式，可是没一个是对的，很奇怪，问了同学后才知道“每 8 个字符分一个空格或回车”而我为了美观一行 32 个 01 字符，每 16 个多加了一个 TAB~修改后就有了现在的波形：

终于和我的测试文件一样的输入了。

现在开始一条一条指令检查，进入下一个阶段。

4，然后根据汇编程序要求的指令依次分析波形如下：

波形分析

然后根据汇编程序要求的指令依次分析波形如下：

（1）addiu $1,$0,8

addiu rt ， rs ,immediate

000010	rs(5 位)	rt(5 位)	immediate(16 位)

功能：rt←rs + (sign-extend)immediate；immediate 符号扩展再参加“加”运算。

该波形图表示执行addiu $1,$0,8指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000000，返回PC指向0x00000004。

b. 输出为 

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b000，表示执行加法运算，下一条指令跳转至

PC+4。

```
always @( ALUopcode or rega or regb ) begin
    case (ALUopcode)
        3'b000 : result = rega + regb;
        3'b001 : result = rega - regb;
        3'b010 : result = regb << rega;
        3'b011 : result = rega | regb;
        3'b100 : result = rega & regb;
        3'b101 : result = (rega < regb)?1:0; // 不带符
        3'b110 : begin                       // 带符号
            if(rega < regb && (rega[31] == regb[31]))r
            else if (rega[31] == 1 && regb[31] == 0)
            else result = 0;
            end
        3'b111 : result = rega ^ regb;
```

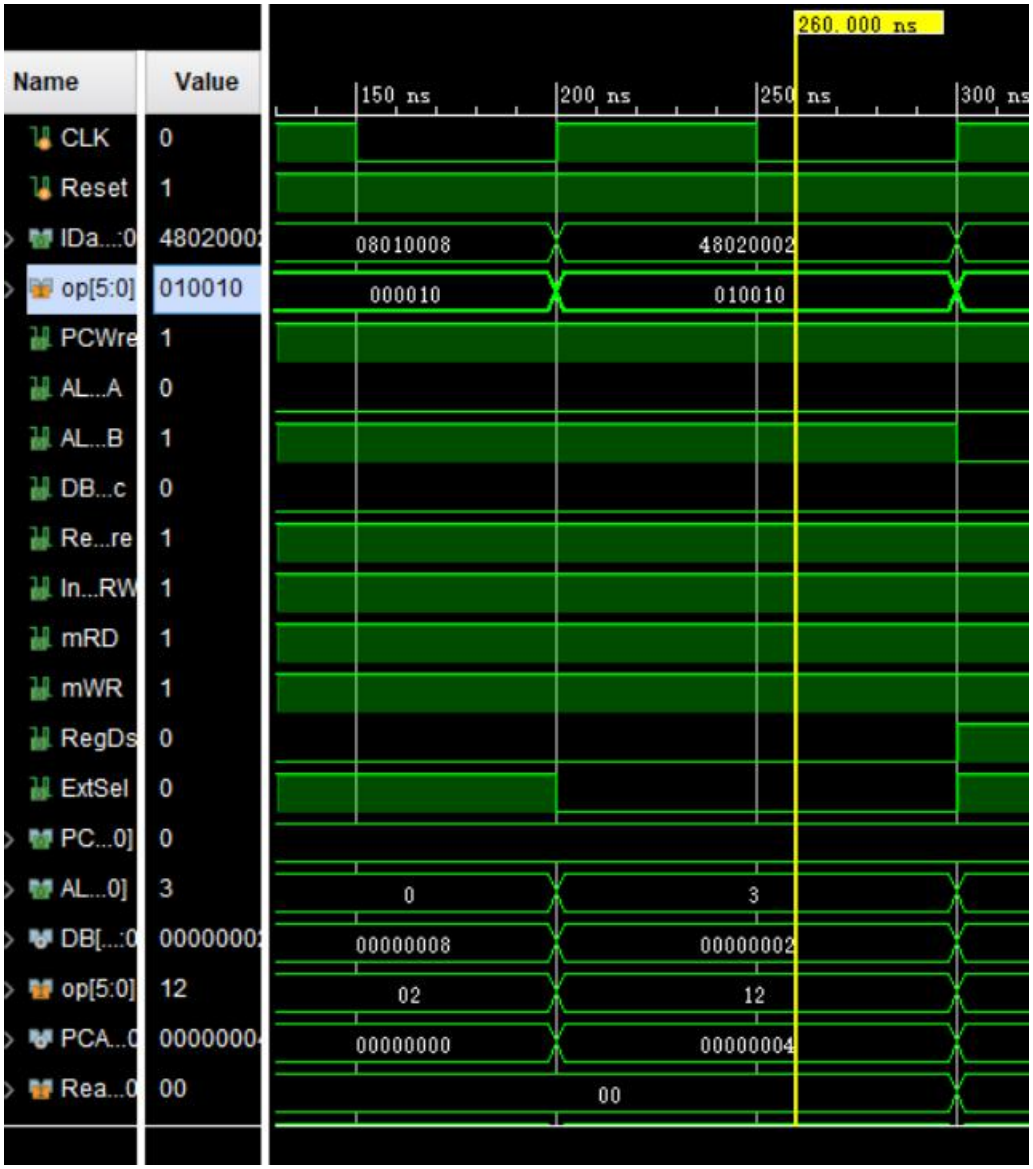
d.寄存器堆中读出 $0 = 0，加上立即数8，写入 $1 = 8。

（2）ori $2,$0,2

ori rt , rs ,immediate

010010	rs(5 位)	rt(5 位)	immediate(16 位)

功能：rt←rs | (zero-extend)immediate；immediate 做“0”扩展再参加“或”运算。



该波形图表示执行 ori　$2,$0,2 指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000004，返回PC指向0x00000008。

b. 输出为 

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b011，表示执行逻辑或运算，下一条指令跳转至PC+4。

d.寄存器堆中读出 $0 = 0，写入 $2 = 2。

（3）add　$3,$2,$1

add　rd , rs, rt

000000	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs + rt。reserved 为预留部分，即未用，一般填“0”。

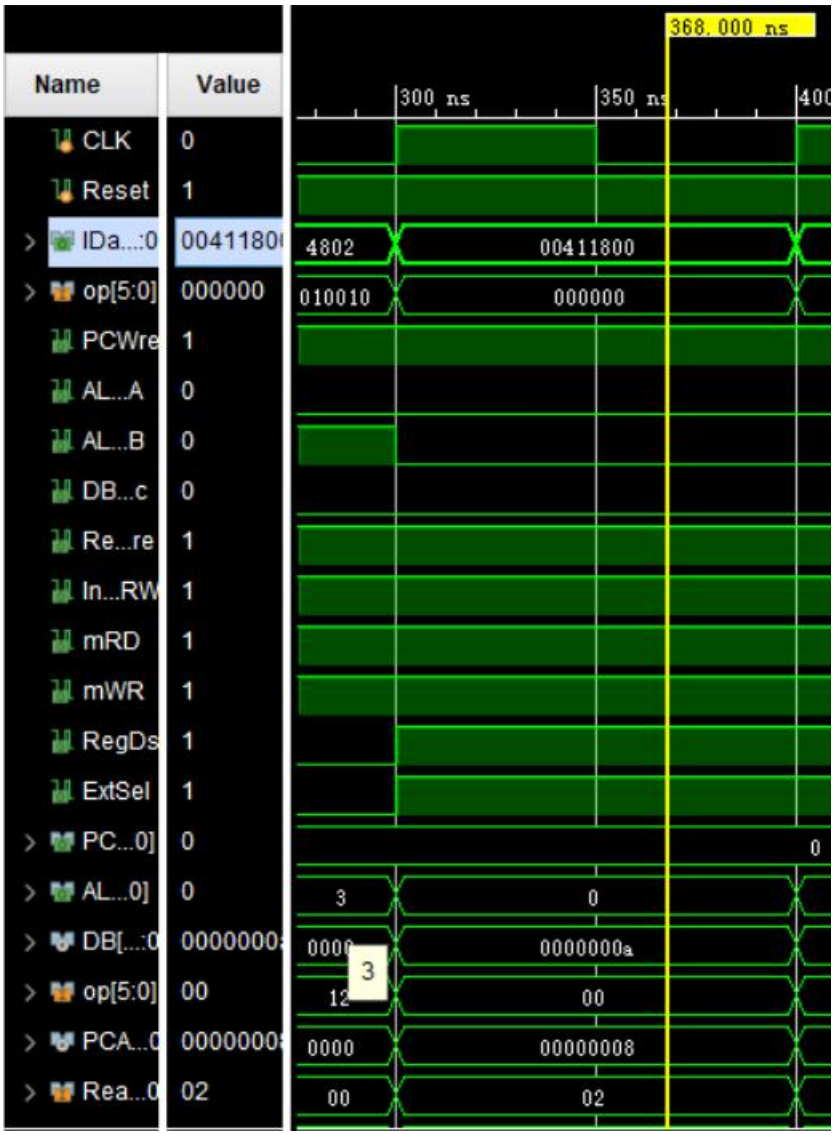
该波形图表示执行add　$3,$2,$1指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000008，返回PC指向0x0000000c。

b. 输出为 

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b000，表示执行加法运算，下一条指令跳转至PC+4。

d.寄存器堆中读出 $1 = 8、$2 = 2，写入 $3 = 10。



（4）sub $5,$3,$2

sub rd , rs , rt

000001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs - rt。

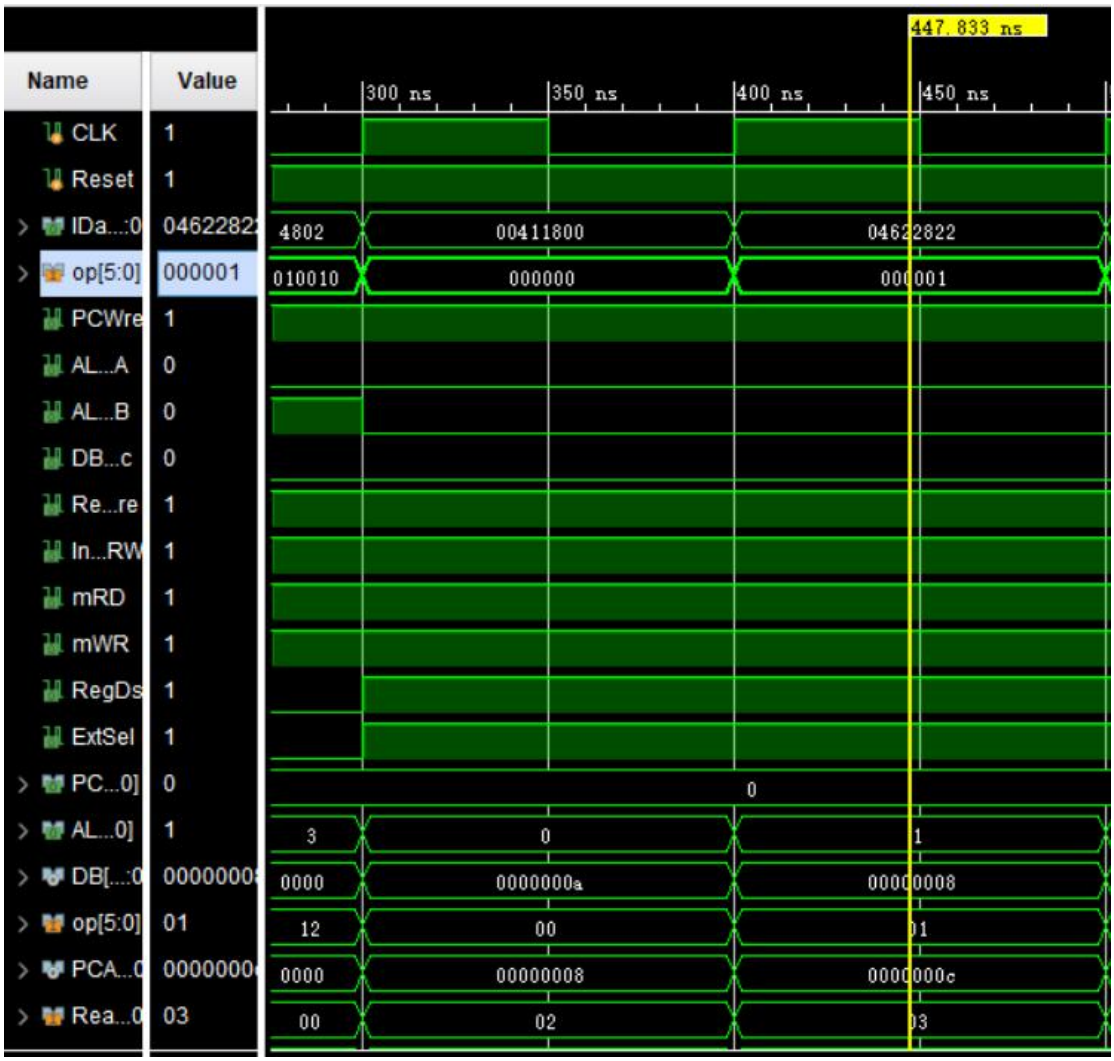
该波形图表示执行sub $5,$3,$2指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x0000000c，返回PC指向0x00000010。

b. 输出为 

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b001，表示执行减法运算，下一条指令跳转至PC+4。

d.寄存器堆中读出 $3 = a、$2 = 2，写入 $5 = 8。



（5）and $4,$5,$2

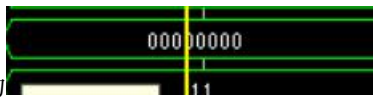
and rd , rs , rt

010001	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs & rt；逻辑与运算。

该波形图表示执行and $4,$5,$2指令的结果。指令执行效果正确，表现在：

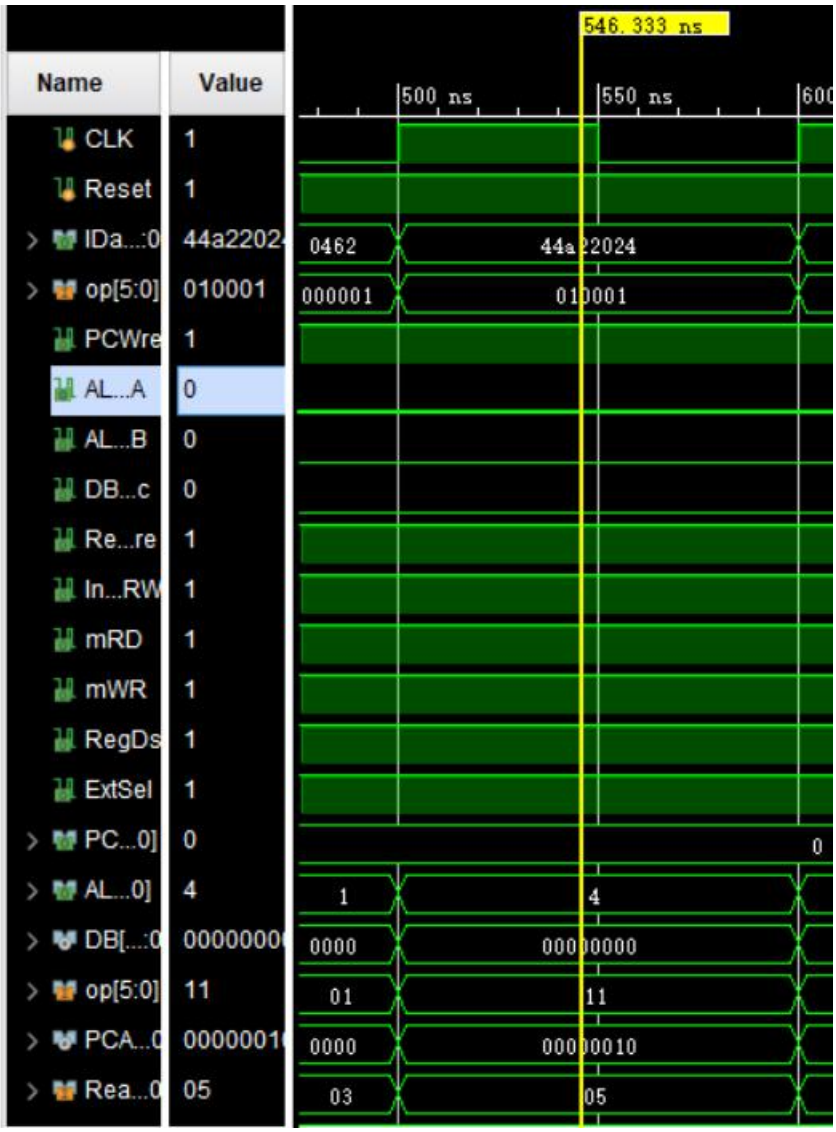
a. 当前PC指向0x00000010，返回PC指向0x00000014。

b. 输出为 

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b100，表示执行逻辑与运算，下一条指令跳转

至PC+4。

d.寄存器堆中读出 $5 = 8、$2 = 2，写入 $4 = 0。



（6）or $8,$4,$2

or rd , rs , rt

010011	rs(5 位)	rt(5 位)	rd(5 位)	reserved

功能：rd←rs | rt；逻辑或运算。

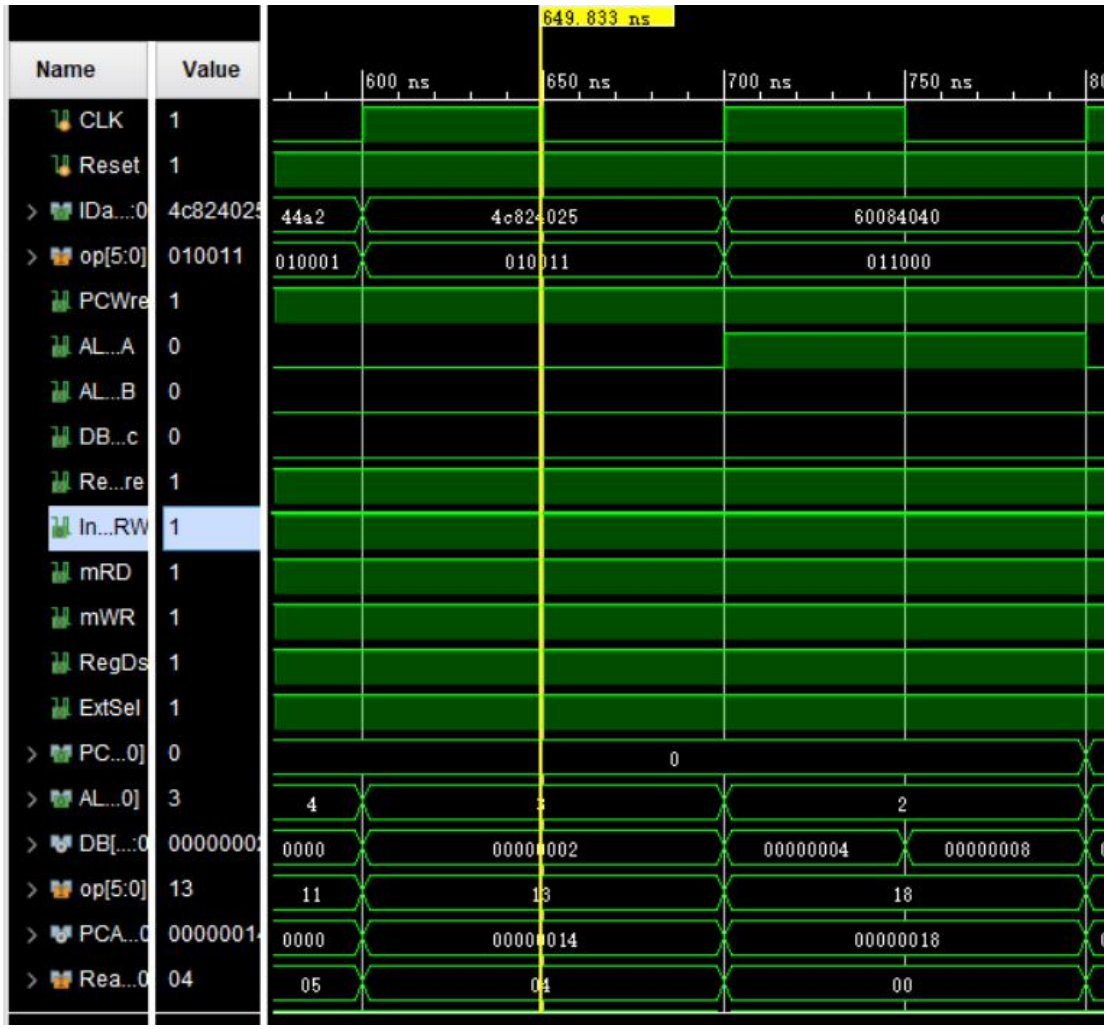
该波形图表示执行or $8,$4,$2指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000014，返回PC指向0x00000018。

b. 输出为 

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b011，表示执行逻辑与运算，下一条指令跳转至PC+4。

d.寄存器堆中读出 $4 = 0、$2 = 2，写入 $8 = 2。



（7）sll　$8,$8,1

sll　rd，rt,sa

011000	未用	rt(5 位)	rd(5 位)	sa(5 位)	reserved

功能：rd<－rt<<(zero-extend)sa，左移 sa 位 ，(zero-extend)sa

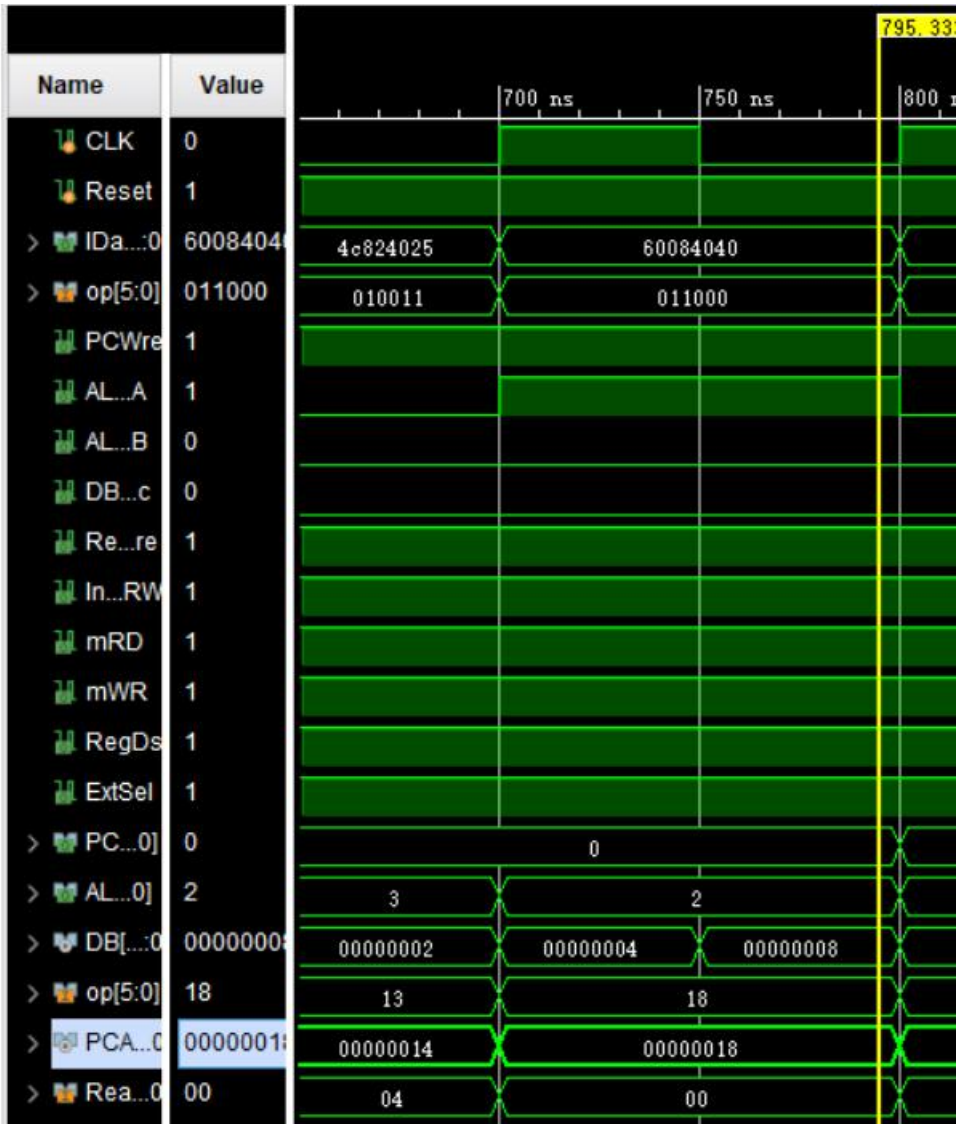
该波形图表示执行sll　$8,$8,1指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000018，返回PC指向0x0000001c。

b. 输出为 

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b010，表示执行移位运算，下一条指令跳转至PC+4。

d.寄存器堆中读出 $8 = 2，写入 $8 = 4。



（8）bne $8,$1,-2 （≠,转 18)

bne rs,rt,immediate

110001	rs(5 位)	rt(5 位)	immediate(16 位)

功能：if(rs!=rt) pc←pc + 4 + (sign-extend)immediate <<2 else pc ←pc + 4

特别说明：与 beq 不同点是，不等时转移，相等时顺序执行。

该波形图表示执行bne $8,$1,-2 (≠,转18)指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x0000001c，返回PC指向0x00000018。

b. 输出为 

c.控制单元中PCSrc = 2’b01，ALUOp = 3’b010，表示执行移位运算，下一条指令跳转至 PC+4 - (-2) * 4。

d.寄存器堆中读出 $8 = 4。

（9）slti $6,$2,4

slti rt, rs,immediate 带符号数

011100	rs(5 位)	rt(5 位)	immediate(16 位)

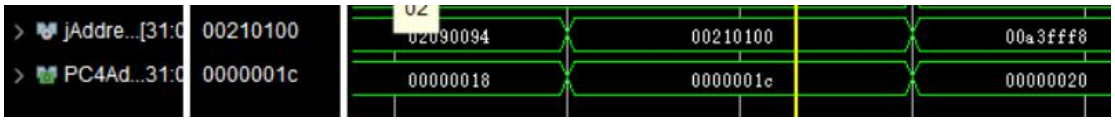
功能：if (rs< (sign-extend)immediate) rt =1 else rt=0, 具体请看表 2 ALU 运算功能表，带符号。

该波形图表示执行 slti $6,$2,4 指令的结果。指令执行效果正确，表现在：

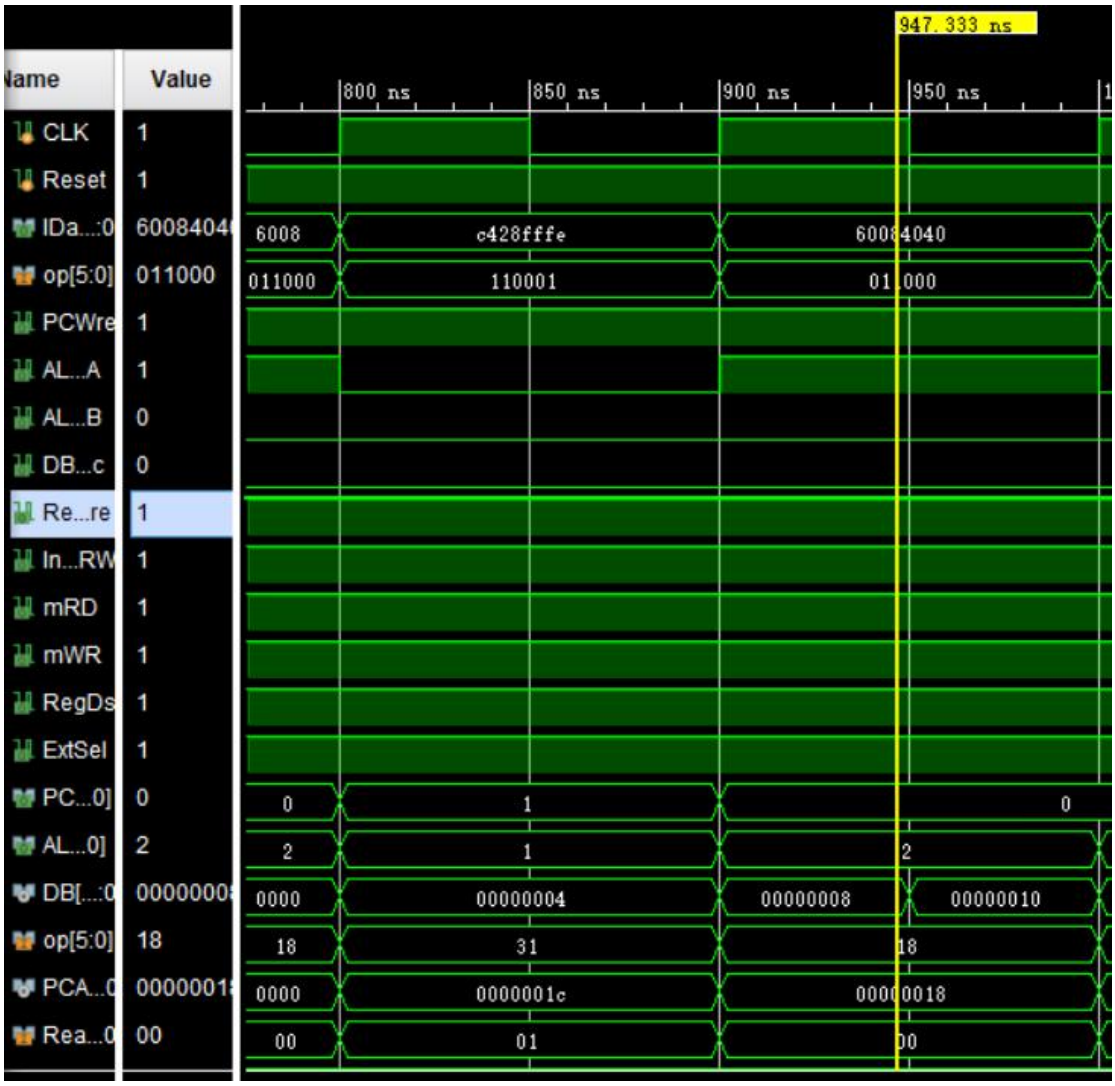
a. 当前PC指向0x0000001c，返回PC指向0x00000018。

b. 输出为 

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b110，表示执行比较运算，下一条指令跳转至

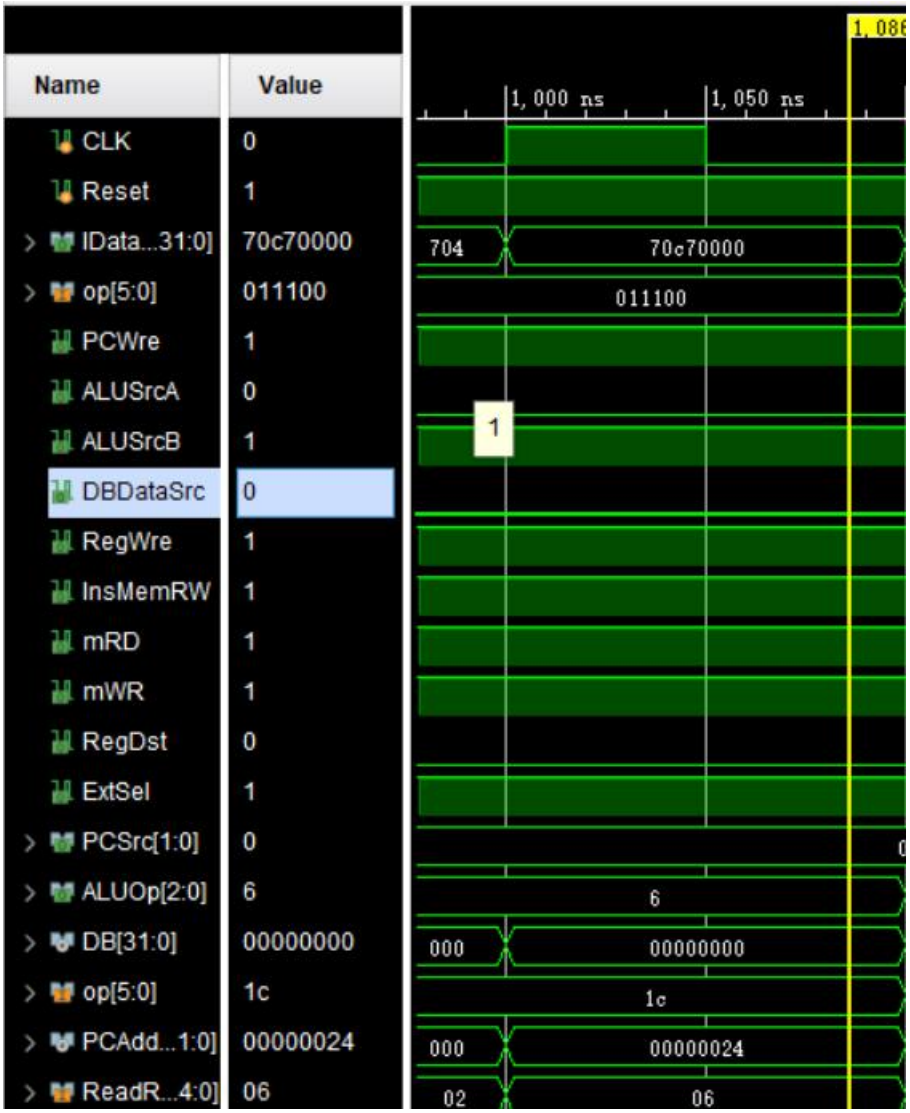


d.寄存器堆中读出 $6 = 8。



（10）slti　$7,$6,0

和上面类似



（11）addiu　$7,$7,8

同第一条差不多

（12）beq $7,$1,-2 (=,转 28)

beq rs,rt,immediate

110000	rs(5 位)	rt(5 位)	immediate(16 位)

功能：if(rs=rt) pc←pc + 4 + (sign-extend)immediate <<2 else pc ←pc + 4

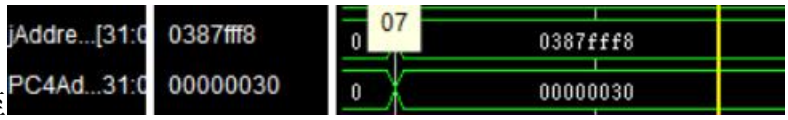
特别说明：immediate 是从 PC+4 地址开始和转移到的指令之间指令条数。immediate 符号扩展之后左移 2 位再相加。为什么要左移 2 位？由于跳转到的指令地址肯定是 4 的倍数（每条指令占 4 个字节），最低两位是“00”，因此将 immediate 放进指令码中的时候，是右移了 2 位的，也就是以上说的“指令之间指令条数”。

该波形图表示执行beq $7,$1,-2 (=,转28)指令的结果。指令执行效果正确，表现在：

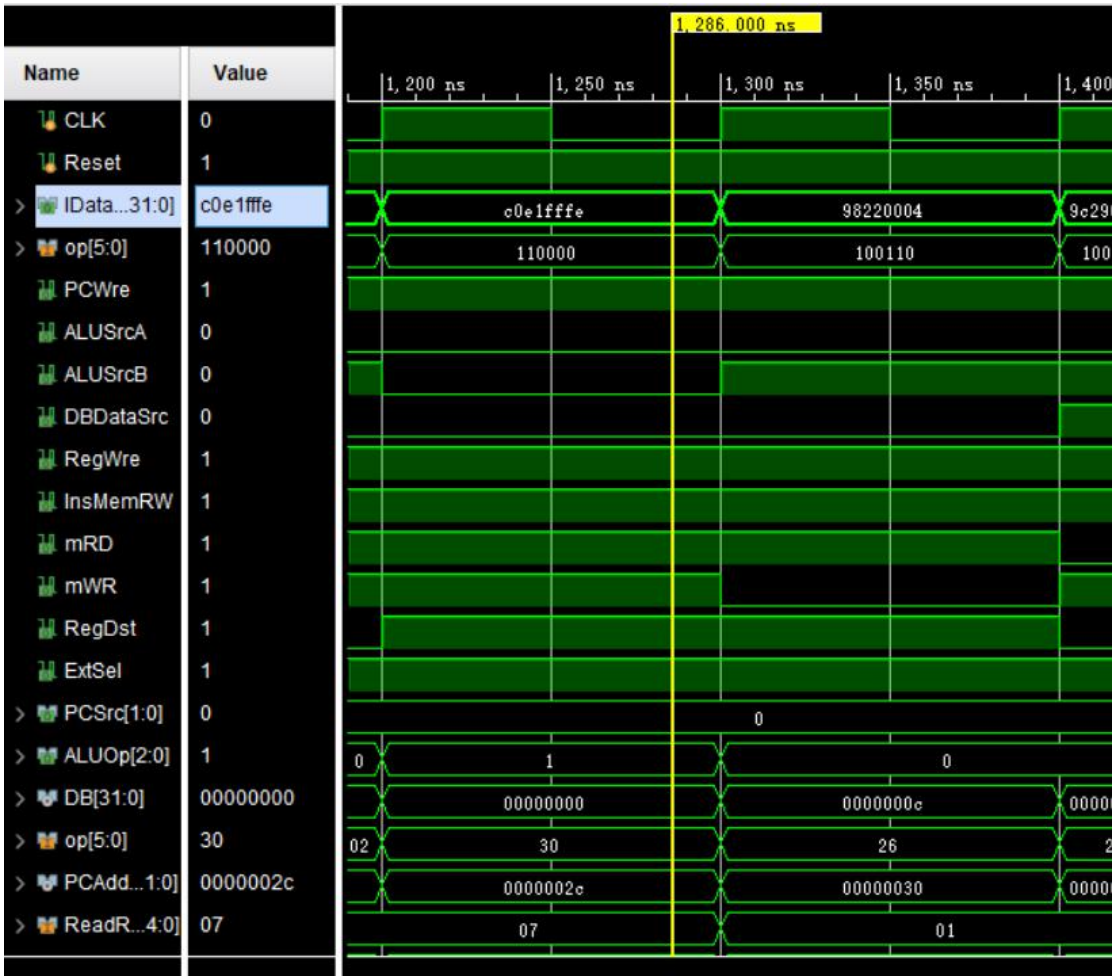
a. 当前PC指向0x0000002c，返回PC指向0x00000030。

b. 输出为0

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b110，表示执行判断大小运算，下一条指令跳转至



d.寄存器堆中读出$7 = 8、 $1 = 8。



（13）sw $2,4($1)

sw rt ,immediate(rs) 写存储器

100110	rs(5 位)	rt(5 位)	immediate(16 位)

功能：memory[rs+ (sign-extend)immediate]←rt；immediate 符号扩展再相加。即将 rt 寄存器的内容保存到 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中。

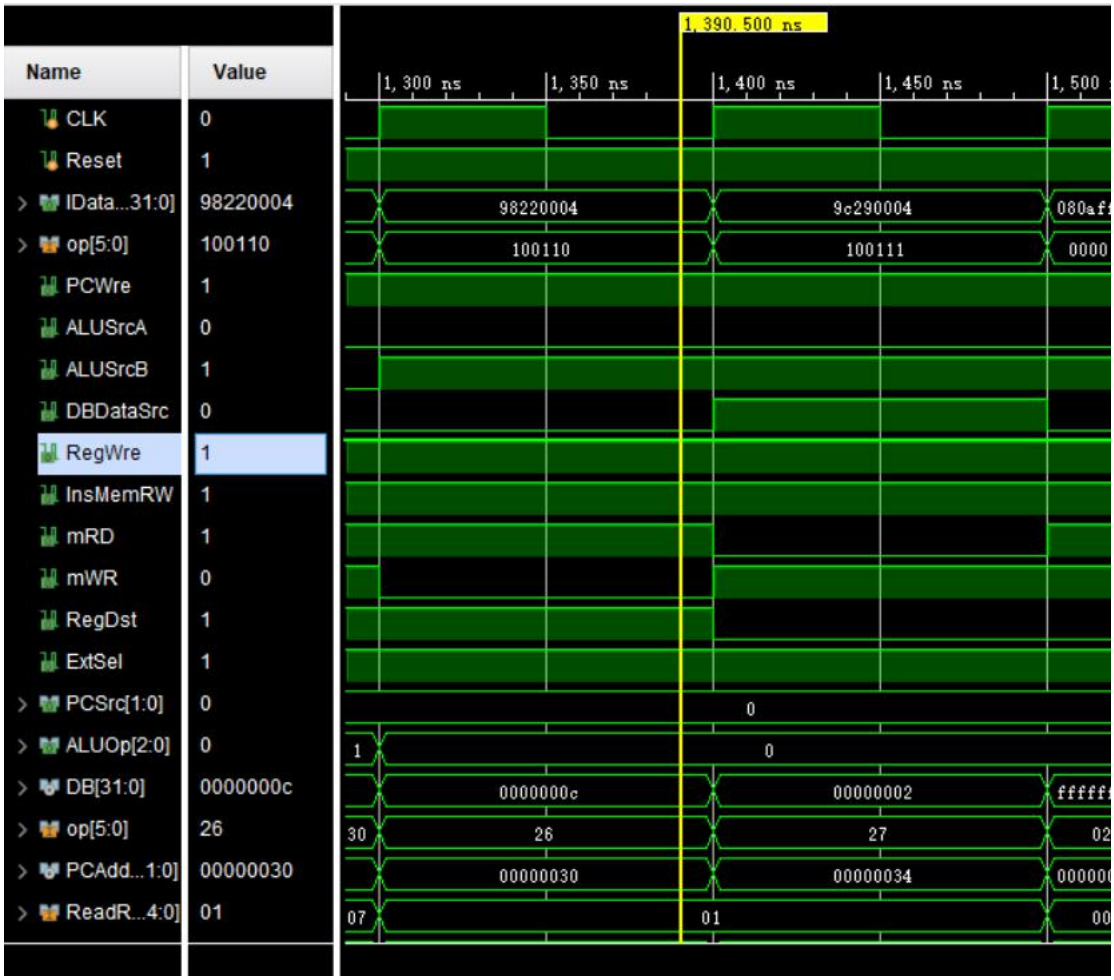
该波形图表示执行sw $2,4($1)指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000030，返回PC指向0x0000034。

b. 输出为0x 0000000c。

c.控制单元中WR = 0，PCSrc = 2’b00，ALUOp = 3’b000，表示执行地址加法运算，写入存储器，下一条指令跳转至PC+4。

d.寄存器堆中读出$2 = 2、 $1 = 8。存储器堆中写入ram[12] = 2。



（14）lw $9,4($1)

lw rt , immediate(rs) 读存储器

100111	rs(5 位)	rt(5 位)	immediate(16 位)

功能：rt ← memory[rs + (sign-extend)immediate]；immediate 符号扩展再相加。

即读取 rs 寄存器内容和立即数符号扩展后的数相加作为地址的内存单元中的数，然后保存到 rt 寄存器中。

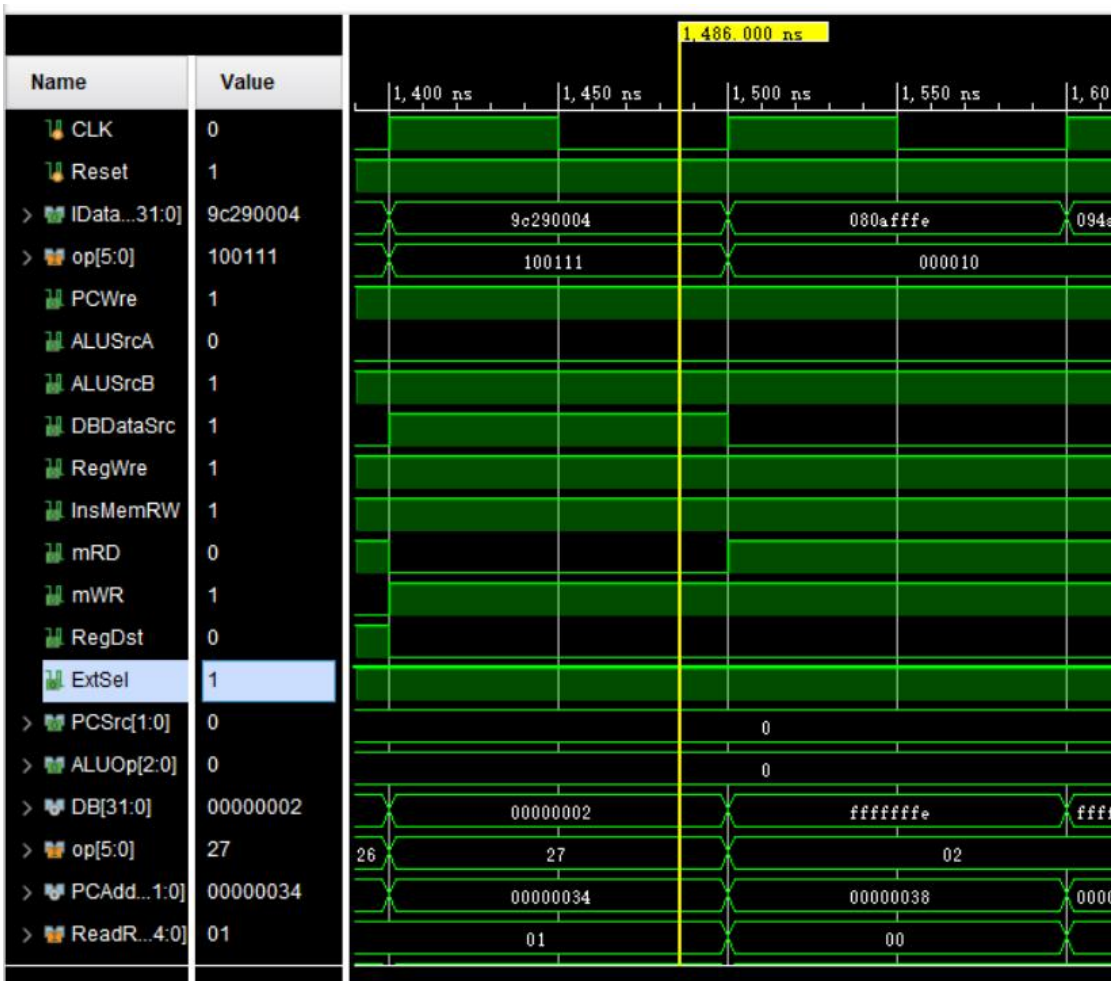
该波形图表示执行lw $9,4($1)指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000034，返回PC指向0x0000038。

b. 输出指令为0x9C290004。

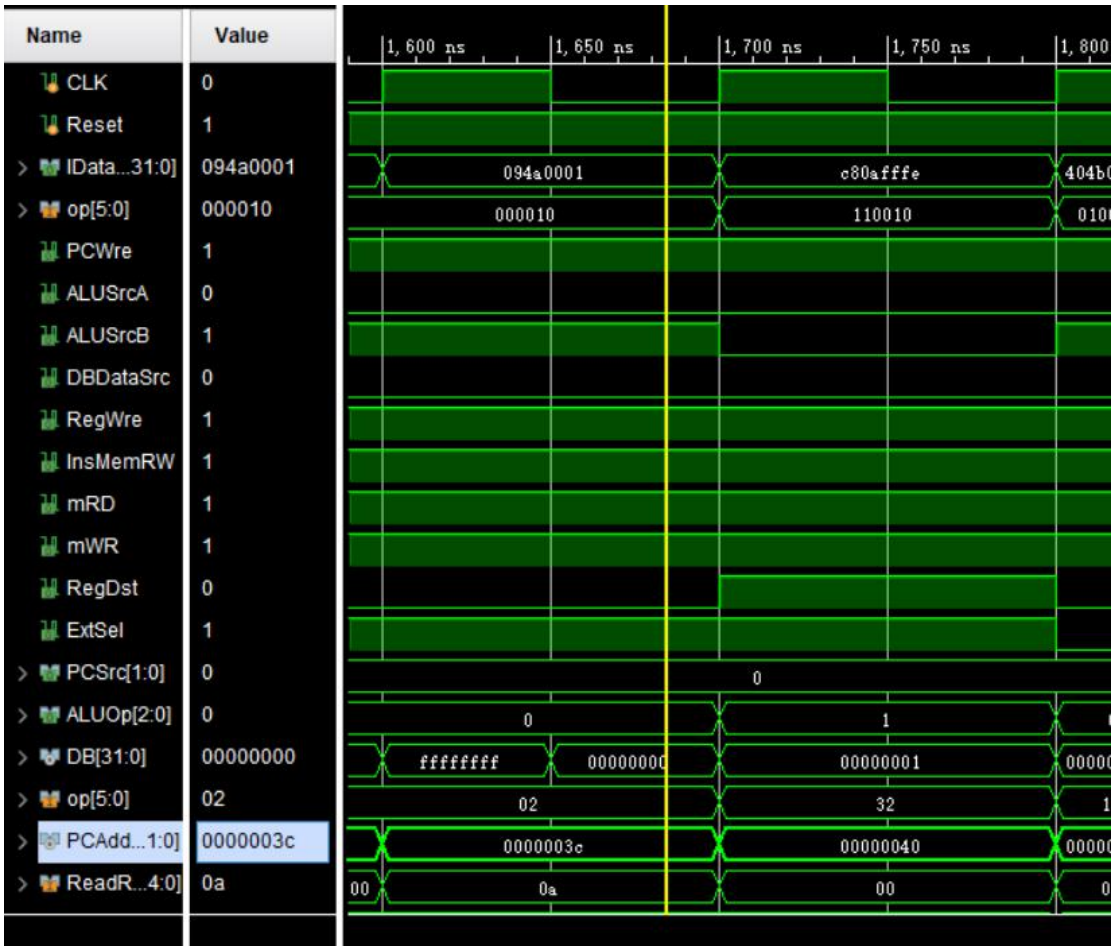
c.控制单元中WR = 1，RD = 0，DBDataSrc = 1，PCSrc = 2’b00，ALUOp = 3’b000，表示执行地址加法运算，读取存储器，选择存储器读取的值写会寄存器，下一条指令跳转至PC+4。

d.写会数据寄存器堆中 $1 = 8，写入$9 = 2。存储器堆中读取ram[12] = 2，写回值DB = 2。



（15）addiu　$10,$0,-2

（16）addiu　$10,$10,1



（17）bltz $10,-2(<0,转 3C)

bltz rs,immediate

110010	rs(5 位)	00000	immediate(16 位)

功能：if(rs<$zero) pc←pc + 4 + (sign-extend)immediate <<2 else pc ←pc + 4。

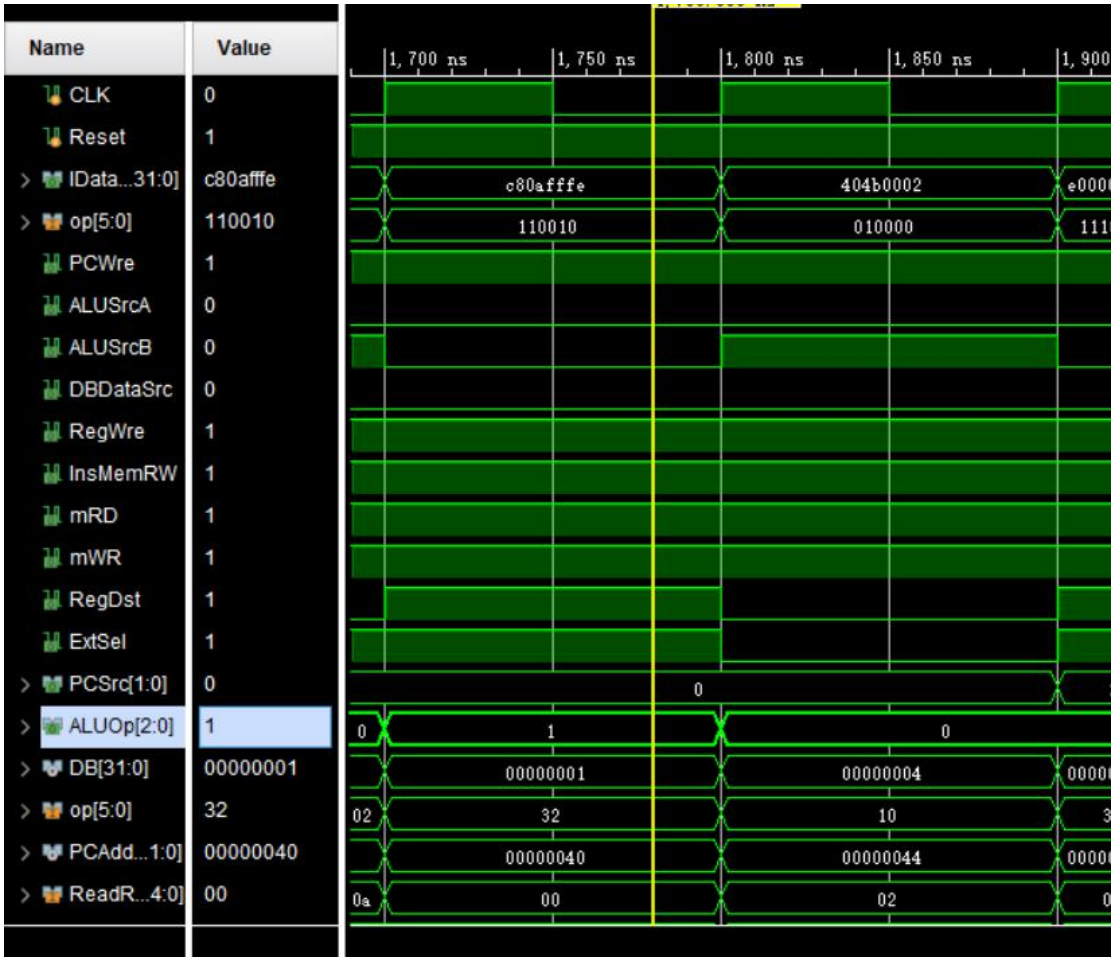
该波形图表示执行bltz $10,-2(<0,转3C)指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000040，返回PC指向0x0000044。

b. 输出为0x 00000001。

c.控制单元中PCSrc = 0sign，ALUOp = 3’b001，sign = 1表示执行减法运算，下一条指令跳转至PC+4+0*4。

d.寄存器堆中读出$10 = 1。



（18）andi　$11,$2,2

andi　rt ， rs ,immediate

010000	rs(5 位)	rt(5 位)	immediate(16 位)

功能：rt←rs & (zero-extend)immediate；immediate 做“0”扩展再参加“与”运算。

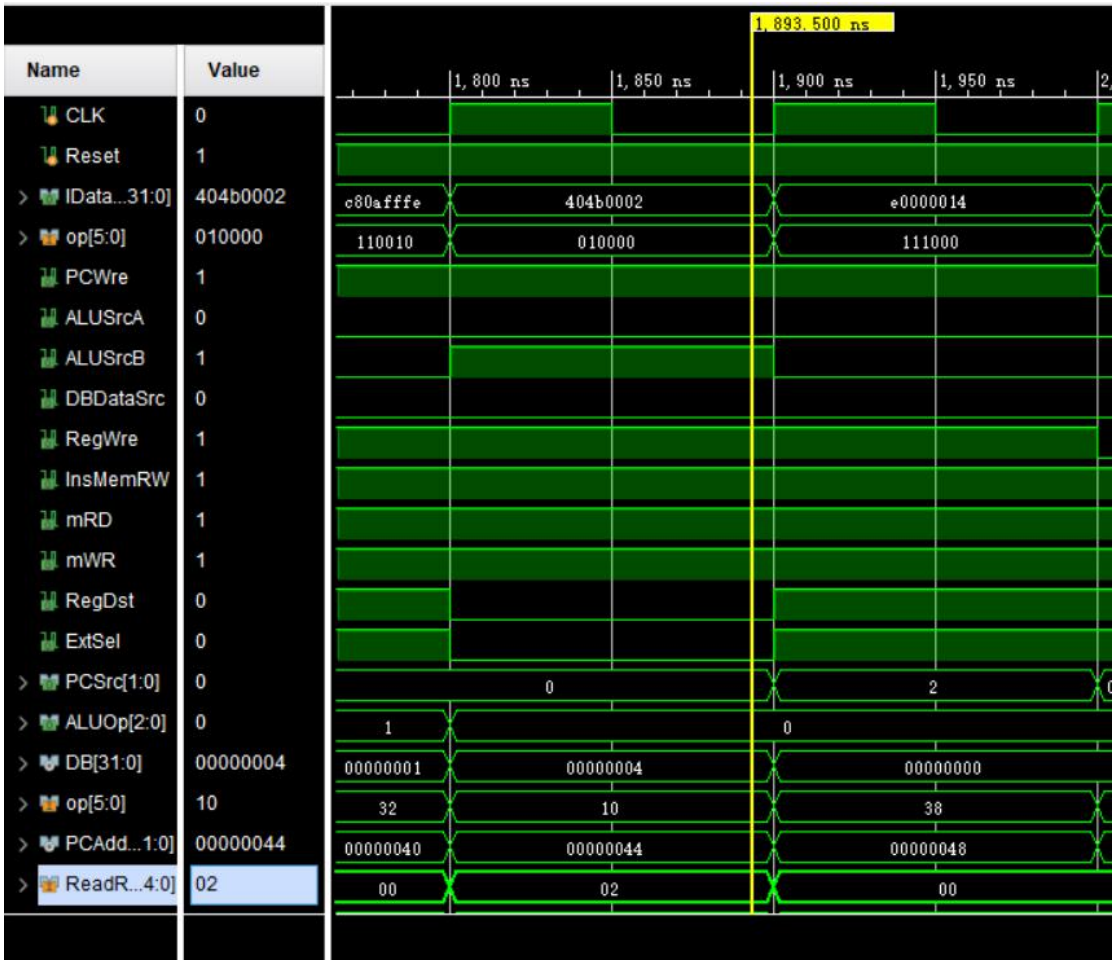
该波形图表示执行andi　$11,$2,2指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000044，返回PC指向0x00000048。

b. 输出为0x00000004。

c.控制单元中PCSrc = 2’b00，ALUOp = 3’b100，表示执行逻辑与运算，下一条指令跳转至PC+4。

d.寄存器堆中读出 $2 = 2，写入 $11= 4。



（19）j　0x00000050

j　addr

111000	addr[27:2]

功能：pc <－{(pc+4)[31:28],addr[27:2],2'b00}，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。

该波形图表示执行j　0x00000038指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000048，返回PC指向0x0000050。

b. 输出为0x 00000000。

c.控制单元中PCSrc = 2’b10，下一条指令跳转至0x50。

（20）or　$8,$4,$2

这个直接被跳转过去了，没有执行

（21）Halt

halt

111111	00000000000000000000000000(26 位)

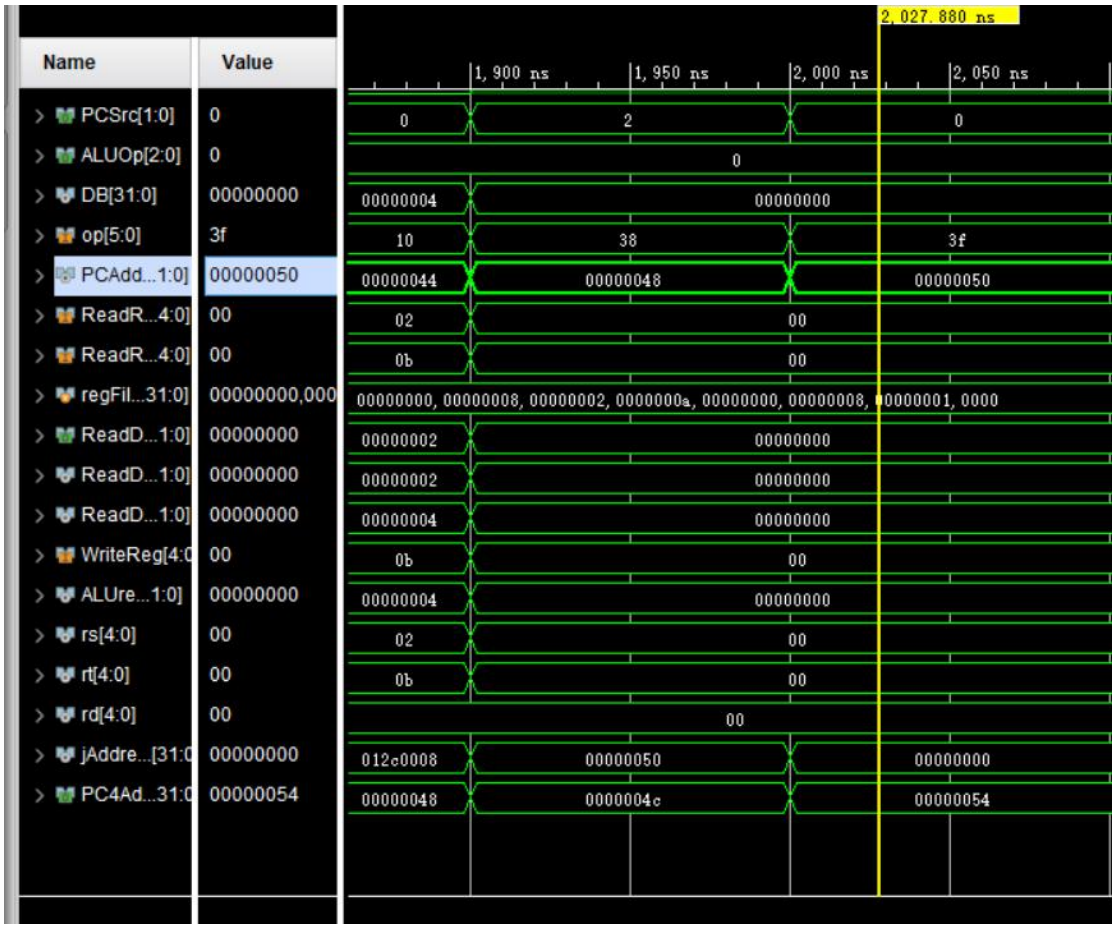
功能：停机；不改变 PC 的值，PC 保持不变。

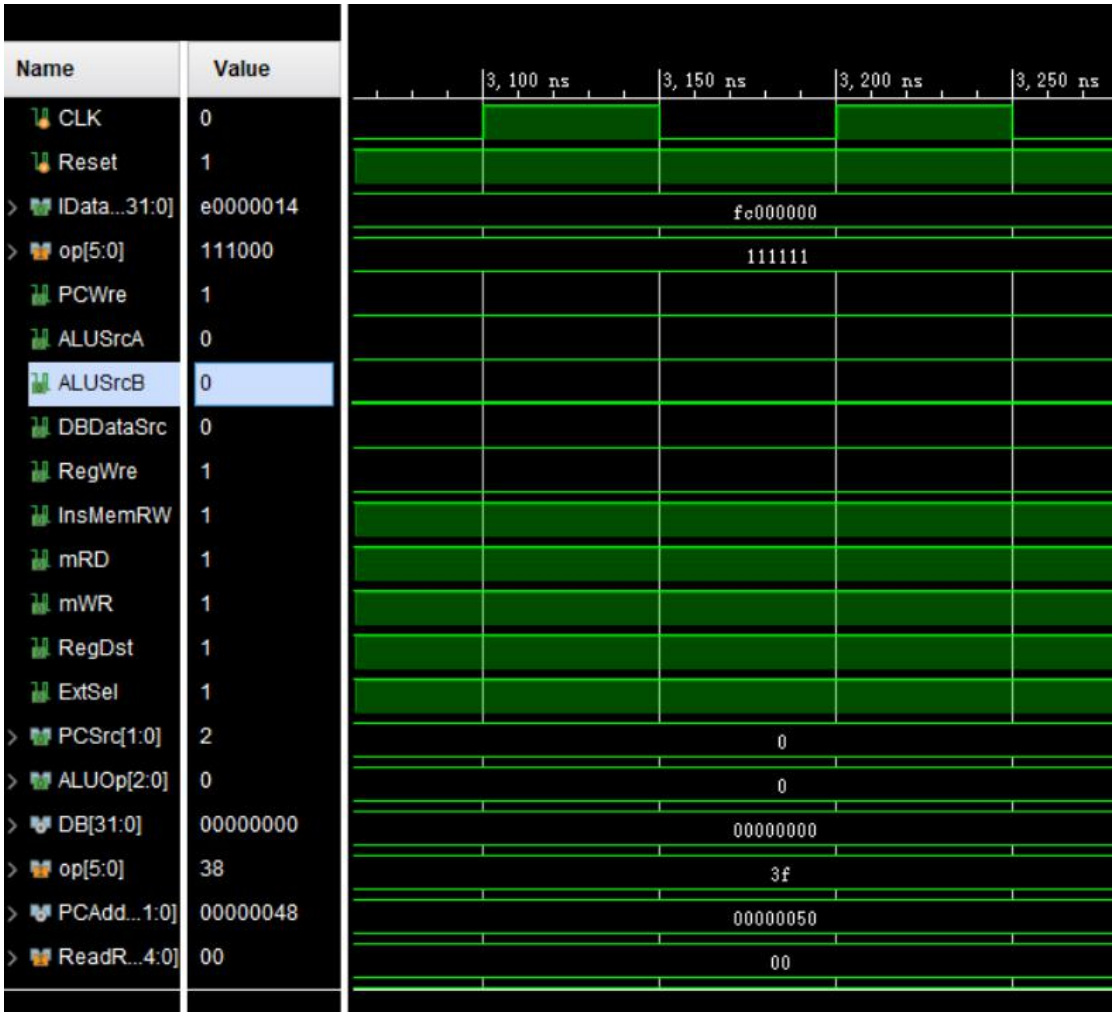
该波形图表示执行halt指令的结果。指令执行效果正确，表现在：

a. 当前PC指向0x00000050，返回PC指向0x0000000。

b. 输出为0x 00000000。

c.控制单元中PCWre表示PC不改变，指令不发生变化 。


Name
Value
PCSrc[1:0]
0
ALUOp[2:0]
0
DB[31:0]
00000000
op[5:0]
3f
PCAdd...1:0]
00000050
ReadR...4:0]
00
ReadR...4:0]
00
regFil...31:0]
00000000,000
ReadD...1:0]
00000000
ReadD...1:0]
00000000
ReadD...1:0]
00000000
WriteReg[4:0]
00
ALURe...1:0]
00000000
rs[4:0]
00
rt[4:0]
00
rd[4:0]
00
jAddre...[31:0]
00000000
PC4Ad...31:0]
00000054
1,900 ns
1,950 ns
2,000 ns
2,050 ns
2,027.880 ns
00000000, 00000008, 00000002, 0000000a, 00000000, 00000008, 00000001, 0000
012c0008
0000004c



六、实验心得

CPU单周期实验老师很早就开始讲了，检查是从上周开始，上周我已经写好了代码，但是有很多错误就没有检查，然后因为电脑的问题，有一次代码无故失踪，最后重新写了一遍，但是VIVADO又没保存下来新更改的，前几天原本写完了波形检查都对了要动手写实验报告时，我清理了一下内存垃圾，结果电脑一下蓝屏，新更改的代码又没了，只留下了原来那个满满的是BUG的，所以这次的代码算是敲了三遍，最后一遍我怕保存不了，这几天也就没敢关机。每一遍都很心酸，但是更加熟悉了各个模块。单独模块其实挺简单的，根据数据通路图就可以写出来。我写好小模块后写了顶层文件，然后就是仿真找BUG。

因为上学期数电对HDL语言并不是很熟悉，所以这次实验基本就是重新学了一遍。关于结构，主要就是对那个数据通路图的熟悉，各条指令怎么一步一步被执行，数据分配等等，第一节课老师带着大家把所有指令的走向都分析了一遍，听完基本就有很清晰的概念了。

下面说一下我遇到的BUG：

一开始时是变量名的问题，同一条线连在不同模块接口名字不一样，所以导致有点混乱，这个在仿真时会有信息提示可以去找。

然后就是一个细节错误：那几个选择器看起来差不多，所以也就直接copy了，没注意到A里面的信号sa是五位的，而不和DA,B一样是32位的，这一个小细节，虽然看起来很简单，但是往往显示错误不会在这里报错，在别的地方报错后一个模块一个模块检查最后才排查到这里，其实早该想到的，只是一开始觉得这么短的一个模块不会有问题就先去排查了别的模块。

下一个是JUMP里面的问题，26位地址移位变28位，余下的四位由PC+4的高四位补上，我写的时候不小心把31:28,写成了31:29，然后这个小错误就很难发现。

粗心犯错的还有总是把input打成inout,但是vivado貌似没有发现这个，这个是偶然看到自己居然打错了这个。

然后最大的一个 BUG 就是，一开始为了美观，我把测试文件的代码把空格用 TAN 代替了，四位一个 TAB 键，结果出来了波形，但是首先一个 ROM 的指令输出都是错的，没一个对的，然后把所有的模块检查好多遍，就是没想到是这里的问题。最后问同学，才知道每 8 个字符分一个空格或回车，然后一下子大半指令波形都对了。接着就是一遍写报告一边按照老师给的汇编程序分析每一条波形，检查出来 SLTI 指令的 PCSRC 信号我写错了，改了之后后面的指令就都对了。JUMP 指令因为第一次丢代码时注意到了在重写时就注意了。还有最后一条 OR 指令，一开始没有被执行我没意识到 JUMP 指令，把每一条指令都当作单独的一条指令来分析的，所以没考虑到这点。

差不多就是这些问题了，波形分析上，找源头确实很麻烦，开始我只看了错误的那条指令，却没想到有可能问题是上一条指令带来的，因为一开始检查时没注意到这点，也是花了不少时间。

最后的心得体会就是，一定要冷静，代码丢了后我又急又气甚至想砸了电脑，辛辛苦苦写了几遍，都莫名其妙失踪，同学们都说没遇到过我的问题，我网上也找不到答案，所以最后的解决办法就是赶快把代码一份一份复制了，免得又丢了，只是这样子搞得一大堆文件有点混乱。而且分析波形一定要冷静下来，不然很容易看到波形头晕眼花。一开始写那个测试文件时，我足足花了两个多小时，看着看着一堆01我就一下子眼花了，一遍又一遍重新对着数。最后波形显示指令全部写对了一遍过了真的很开心。

做完实验，不仅熟悉了VIVADO的基本操作，弥补了上学期的遗憾，而且调试波形也开始得心应手，最后也对计组理论的理解有了很大的帮助。