《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 计教学 2 班

学生姓名 ：王程钥

学号 ：17341146

时间 ：2018 年 11 月 23 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)	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。

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。

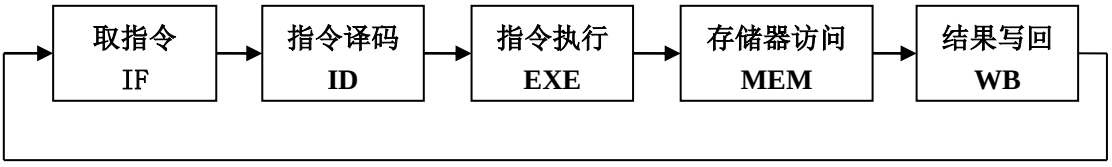


图 1　单周期 CPU 指令处理过程

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。

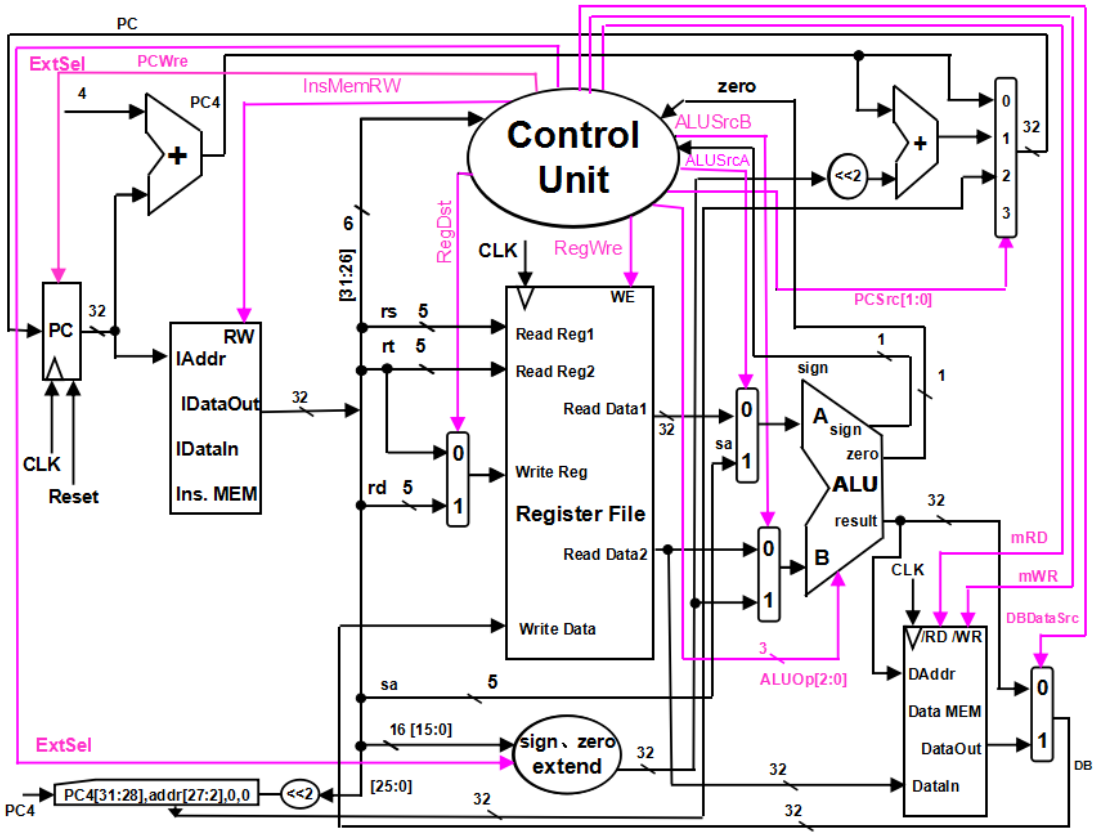


图 2　单周期 CPU 数据通路和控制线路图

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)。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（关系表将在下面控制译码部分给出），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

本次实验的代码主要分为CPU部分，仿真部分和写板部分。

1、CPU部分。

CPU部分包括了地址跳转（PC），指令内存（Instruction Memory）,控制信号译码（control），符号拓展（Sign Extend），寄存器（Regfile），计算器（ALU），选择器（MUX），数组内存（Data Memory）。

1）地址跳转（PC）

输入当前的地址，输出下一步要跳转的地址。根据Reset信号决定是否要清空地址，根据PCSrc信号决定是不跳转直接PC+4，还是条件跳转，还是无条件跳转。

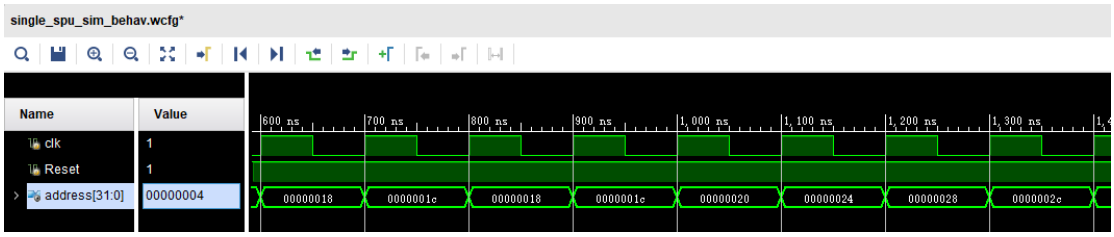
输入：clk,Reset,PCSrc,immediate,address

输出：NewAddress

核心代码：

```
reg [31:0] new;
always @(negedge clk)begin
    if(Reset==0)NewAddress=0;
    else begin
        new=address+4;
        case (PCSrc)
           2'b00: NewAddress=new;
           2'b01: NewAddress=new+immediate*4;
           2'b10: NewAddress={ new[31:28],immediate[27:2],2'b00 };
        endcase
    end
end
```

仿真结果：



2）指令内存（Instruction Memory）

指令从文件“input.txt”读入，开一个数组模拟内存存储所有指令。每次调用的时候根据地址从指令内存数组中读取指令，得到当前的指令，并将当前指令相应的部分（OP,rs,rt,rd,sa,immediate）赋值。

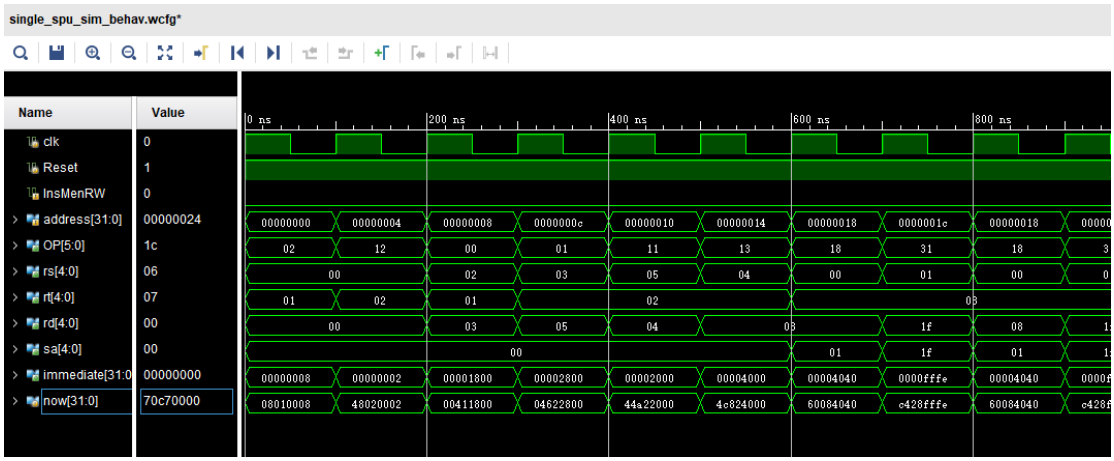
输入：InsMenRW,address

输出：OP,rs,rt,rd,sa,immediate

核心代码：

```
reg [7:0] Memory [255:0];
reg [31:0] now;
initial begin
    $readmemb("D:/input.txt", Memory);
end
always@(address or InsMenRW)begin
    if(!InsMenRW)begin
        now={ Memory[address] , Memory[address+1] , Memory[address+2] , Memory[address+3] };
        OP= now [31:26];
        rs= now [25:21];
        rt= now [20:16];
        rd= now [15:11];
        sa= now [10:6];
        immediate= now [15:0];
    end
end
```

仿真结果：



3）译码部分（control）

根据指令的类型，即op域，确定各种控制信号的值，完成对控制信号的译码。

输入：OP,zero

输出：PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre,InsMenRW,mRD,mWR,RegDst,ExtSel,PCSrc,ALUOp

	zero	Reset	PCWre	ALUSrcA	ALUSrcB	DBDataSrc	RegWre	InsMenRW
add	x	1	1	0	0	0	1	0
sub	x	1	1	0	0	0	1	0
and	x	1	1	0	0	0	1	0
or	x	1	1	0	0	0	1	0
addiu	x	1	1	0	1	0	1	0
andi	x	1	1	0	1	0	1	0
ori	x	1	1	0	1	0	1	0
sll	x	1	1	1	x	0	1	0
slti	x	1	1	0	1	0	1	0
bne	0	1	1	0	0	x	0	0
	1	1	1	0	0	x	0	0
beq	0	1	1	0	0	x	0	0
	1	1	1	0	0	x	0	0
bltz	0	1	1	0	0	x	0	0
	1	1	1	0	0	x	0	0
sw	x	1	1	0	1	x	0	0
lw	x	1	1	0	1	1	1	0
j	x	1	1	x	x	x	x	0
halt	x	1	0	x	x	x	x	0

	mRD	mWR	RegDst	ExtSel	PCSrc[1]	PCSrc[0]	ALUOp[2]	ALUOp[1]	ALUOp[0]
add	0	0	1	x	0	0	0	0	0
sub	0	0	1	x	0	0	0	0	1
and	0	0	1	x	0	0	1	0	0
or	0	0	1	x	0	0	0	1	1
addiu	0	0	0	1	0	0	0	0	0
andi	0	0	0	0	0	0	1	0	0
ori	0	0	0	0	0	0	0	1	1
sll	0	0	1	x	0	0	0	1	0
slti	0	0	0	1	0	0	1	1	0
bne	0	0	x	1	0	1	0	0	1
	0	0	x	1	0	0	0	0	1
beq	0	0	x	1	0	0	0	0	1
	0	0	x	1	0	1	0	0	1
bltz	0	0	x	1	0	0	1	1	0
	0	0	x	1	0	1	1	1	0
sw	0	1	x	1	0	0	0	0	0
lw	1	0	0	1	0	0	0	0	0
j	0	0	x	x	1	0	x	x	x
halt	x	x	x	x	x	x	x	x	x

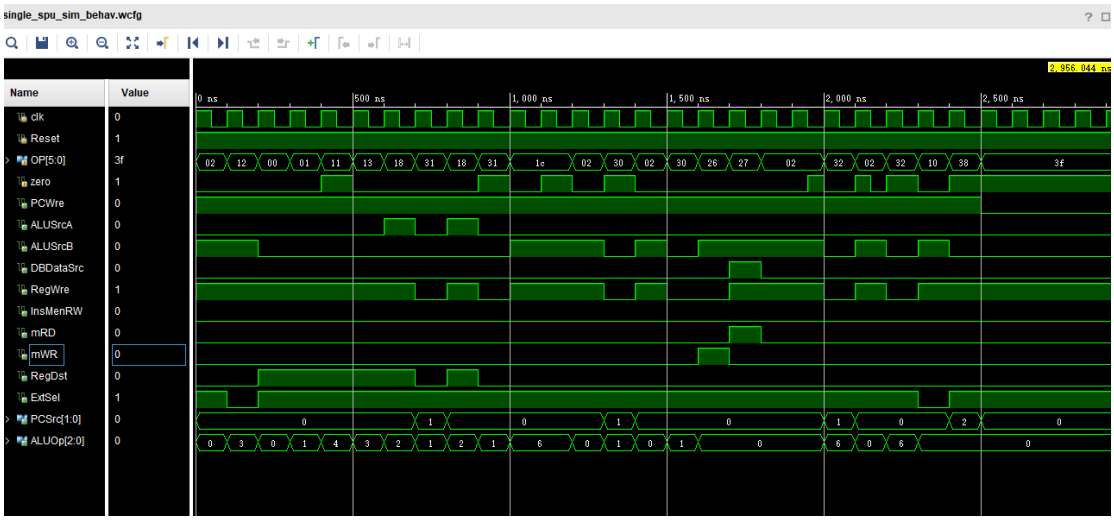
如图为本次实验需要用到的指令对应的控制信号的真值表。（X表示不影响结果）

核心代码：

```
always@(OP or zero)begin
    PCWre=( OP==6'b111111 )?0:1;
    ALUSrcA=( OP==6'b011000 )?1:0;
    ALUSrcB=( OP==6'b000010 || OP==6'b010000 || OP==6'b010010 ||
    OP==6'b011100 || OP==6'b100110 || OP==6'b100111 )?1:0;
    DBDataSrc=( OP==6'b100111 )?1:0;
    RegWre=( OP==6'b110000 || OP==6'b110001 || OP==6'b110010 ||
    OP==6'b100110 )?0:1;
    InsMenRW=0;
    mRD=( OP==6'b100111 )?1:0;
    mWR=( OP==6'b100110 )?1:0;
    RegDst=( OP==6'b000000 || OP==6'b000001 || OP==6'b010001 ||
    OP==6'b010011 || OP==6'b011000 )?1:0;
    ExtSel=( OP==6'b010000 || OP==6'b010010 )?0:1;
    PCSrc[1]=( OP==6'b111000 )?1:0;
    PCSrc[0]=( (OP==6'b110000 && zero==1) || (OP==6'b110001 && zero==0) ||
    (OP==6'b110010 && zero==0) )?1:0;
```

```
    ALUOp[2]=( OP==6'b010001 || OP==6'b011100 || OP==6'b110010 ||
    OP==6'b010000 )?1:0;
    ALUOp[1]=( OP==6'b010011 || OP==6'b010010 || OP==6'b011000 ||
    OP==6'b011100 || OP==6'b110010 )?1:0;
    ALUOp[0]=( OP==6'b000001 || OP==6'b010011 || OP==6'b010010 ||
    OP==6'b110000 || OP==6'b110001 )?1:0;
end
```

仿真结果：



4）符号拓展（Sign Extend）

根据控制信号ExtSel决定拓展的类型，将16位立即数拓展为32位立即数

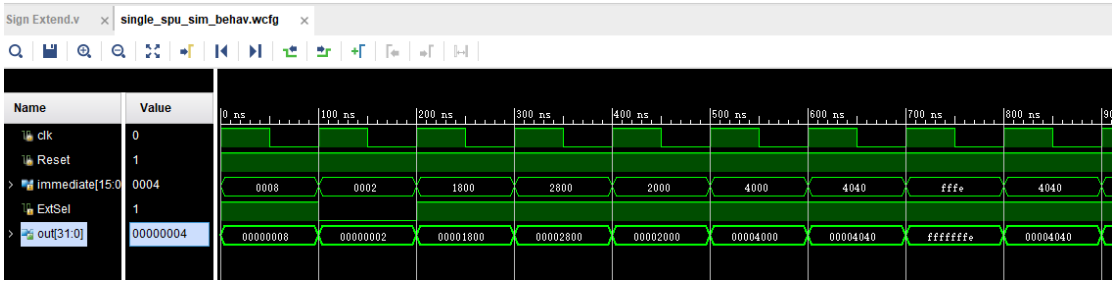
输入：immediate，ExtSel

输出：out

核心代码：

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

仿真结果：



5)寄存器（Regfile）

寄存器分为读写数据两个部分。

读操作从寄存器中读取对应位置的数据，两个地址为ReadReg1和ReadReg2。

写操作只有在时钟下降沿才会进行，在对应的内存写入数据，写入地址为WriteReg，写入的数据为WriteData。

Regfile数组模拟寄存器，数据存储在该数组里。

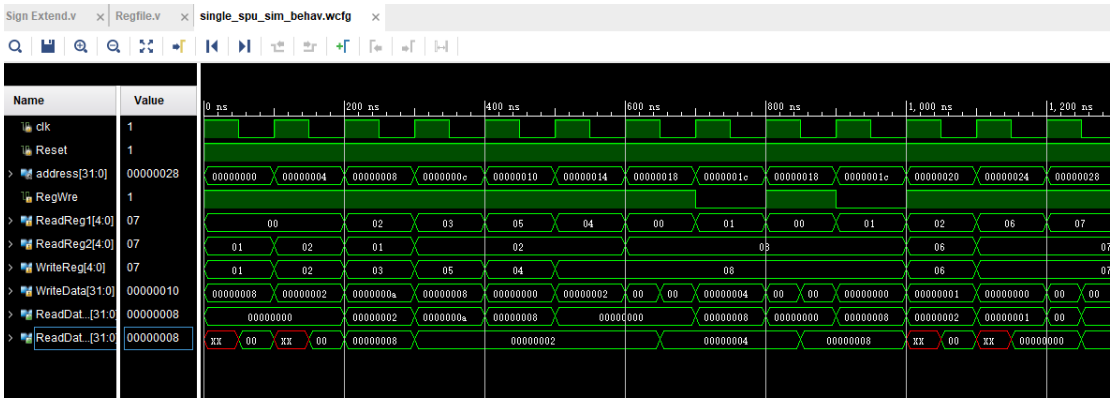
输入：CLK,RST,RegWre,ReadReg1,ReadReg2,WriteReg, WriteData,

输出：ReadData1,ReadData2

核心代码：

```
reg [31:0] regFile[1:31];
integer i;
assign ReadData1 = (ReadReg1 == 0) ? 0 : regFile[ReadReg1];
assign ReadData2 = (ReadReg2 == 0) ? 0 : regFile[ReadReg2];
always @ (negedge CLK or negedge RST) begin
    if (RST==0) begin
        for(i=1;i<32;i=i+1) regFile[i] <= 0;
    end
    else if(RegWre == 1 && WriteReg != 0)
        regFile[WriteReg] <= WriteData;
end
```

仿真结果：



6）计算部分（ALU）

根据输入的ALUopcode信号决定计算的类型，并将rega和regb两个数据进行计算，结果输出到result里。Zero位用于判断结果是否为0，用于条件跳转指令。

输入：ALUopcode,rega,regb

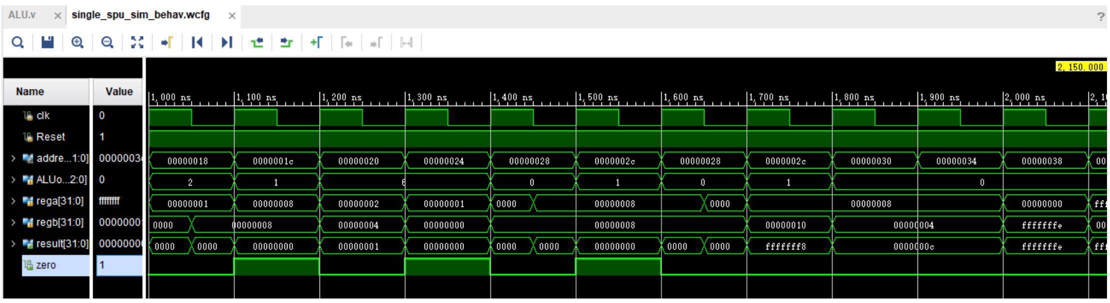
输出：result,zero

核心代码：

```
assign zero = (result==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
           endcase
        end
```

仿真结果



7）数据内存（Data Memory）

数据内存分为数据读写两个部分。读数据部分，将内存中的数据写到Dataout数组中。写数据只在下降沿写，将writeData的数据写入内存。Ram数组模拟数据内存。

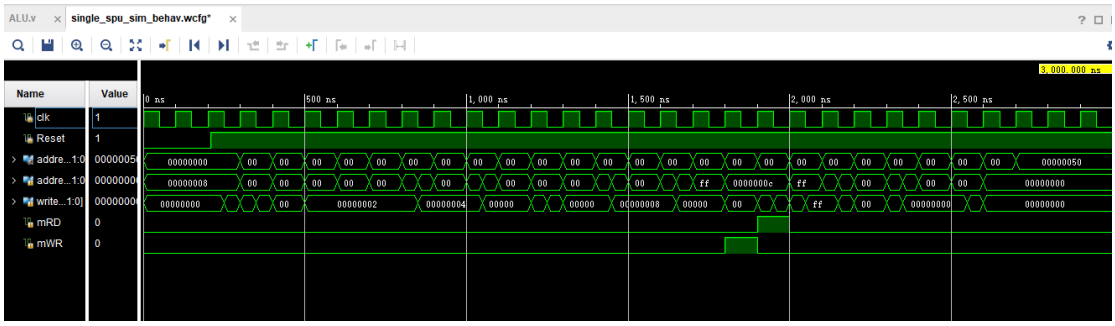
输入：clk,address,writeData,mRD,mWR,

输出：Dataout

核心代码：

```
reg [7:0] ram [0:60];
//读
always @(address or mRD) begin
    Dataout[7:0] = (mRD==1)?ram[address + 3]:8'bz;
    Dataout[15:8] = (mRD==1)?ram[address + 2]:8'bz;
    Dataout[23:16] = (mRD==1)?ram[address + 1]:8'bz;
    Dataout[31:24] = (mRD==1)?ram[address ]:8'bz;
end
// 写
always@( negedge clk ) begin
    if( mWR==1 ) begin
        ram[address] <= writeData[31:24];
        ram[address+1] <= writeData[23:16];
        ram[address+2] <= writeData[15:8];
        ram[address+3] <= writeData[7:0];
    end
end
```

仿真结果：



8）选择器（MUX）

根据需求需要建立三种选择器，分别为两个五进制（用于选择寄存器写入地址），两个32进制（用于选择ALU中第二个数据和寄存器写入数据），一个5进制一个32进制（用于选择ALU中第一个数据）。

输入两个数据(data0,data1)和选择控制信号(op)，输出其中一个数据。

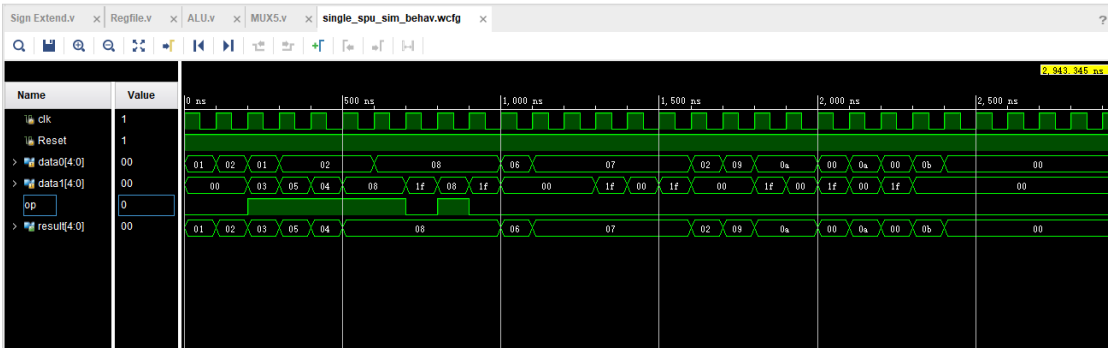
输入：data0,data1,op

输出：result

核心代码：

```
always@(op or data0 or data1)begin
    result=(op==0)?data0:data1;
end
```

仿真结果：



9）顶层文件

顶层文件将上述底层文件串联在一起，构成单周期CPU。

因为verilog支持并行，所以调用底层文件的顺序并没有什么影响。

将各种控制信号和数据存储在顶层文件中。

输入：clk,Reset

输出：address,result,data3,data4,rs,rt

核心代码：

```
PC pc(clk,Reset,PCWre,NewAddress,address);
//pc
InstructionMemory ins(InsMenRW,address,OP,rs,rt,rd,sa,immediate);
//取指
control
cont(OP,zero,PCWre,ALUSrcA,ALUSrcB,DBDataSrc,RegWre,InsMenRW,mRD,mWR,Re
gDst,ExtSel,PCSrc,ALUOp);
//译码
```

```
SignExtend SE(immediate,ExtSel,Extend);
//符号拓展
RegFile RF(clk,Reset,RegWre,rs,rt,RegWrite,writeData,data3,data4);
//读写寄存器
ALU ALU(ALUOp,data1,data2,result,zero);
//计算
DataMemory Data(clk,Reset,result,data4,mRD,mWR,Dataout);
//数据内存读写
MUX5 M1(rt,rd,RegDst,RegWrite);
//选择数据寄存器的写入地址
MUX532 M2(data3,sa,ALUSrcA,data1);
//选择ALU的data1
MUX32 M3(data4,Extend,ALUSrcB,data2);
//选择ALU的data2
MUX32 M4(result,Dataout,DBDataSrc,writeData);
//选择写入内存数字
GetAddress get(clk,Reset,PCSrc,Extend,address,NewAddress);
//算地址
```

2、仿真部分

设立虚拟时钟clk初值为1，Reset初值为1，每50ps刷新一次，即clk=!clk。

核心代码：

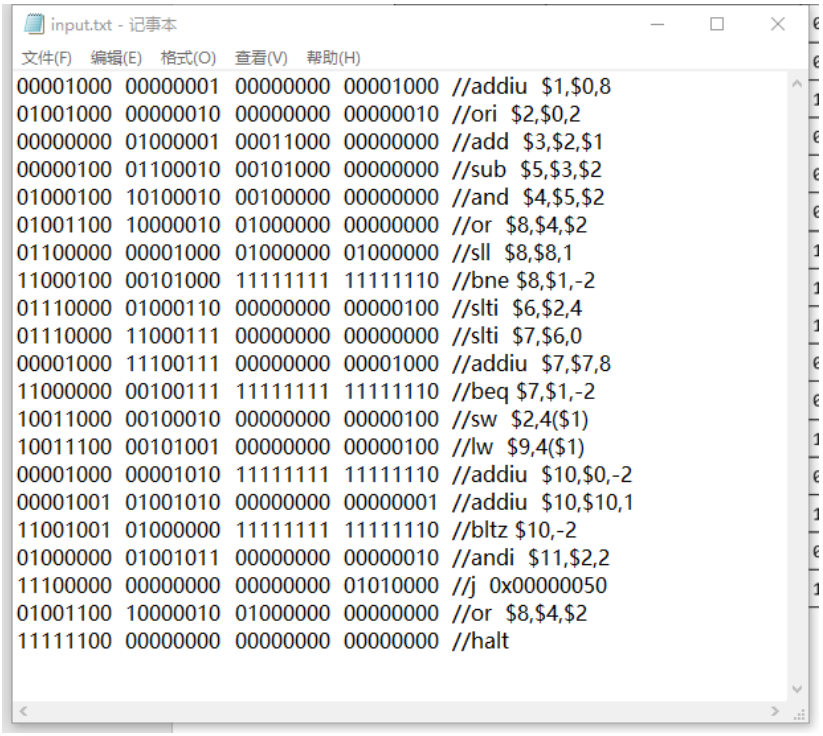
```
Single_CPU cpu(
    .clk(clk),
    .Reset(Reset),
    .address(address),
    .result(result),
    .data1(data1),
    .data2(data2),
    .data3(data3),
    .data4(data4),
    .OP(OP),
    .rs(rs),
    .rt(rt),
    .rd(rd),
    .RegWrite(RegWrite),
    .writeData(writeData),
    .ALUOp(ALUOp),
    .PCSrc(PCSrc)
);

initial begin
    Reset=1;clk=1;
end
always #50 clk=!clk;
```

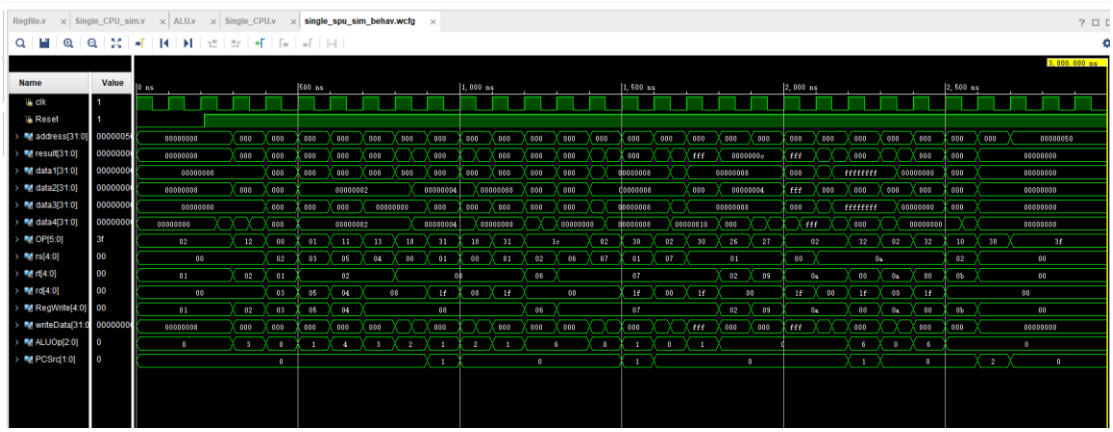
仿真结果：

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	00011 00000 000000	=	00411800
0x0000000C	sub $5,$3,$2	000001	00011	00010	00101 00000 000000	=	04622800
0x00000010	and $4,$5,$2	010001	00101	00010	00100 00000 000000	=	84A22000
0x00000014	or $8,$4,$2	010011	00100	00010	01000 00000 000000	=	4C824000
0x00000018	sll $8,$8,1	011000	00000	01000	01000 00001 000000	=	60084040
0x0000001C	bne $8,$1,-2 (≠,转 18)	110001	00001	01000	1111 1111 1111 1110	=	C428FFFE
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	00001	00111	1111 1111 1111 1110	=	C027FFFE
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 0101 0000	=	E0000050
0x0000004C	or $8,$4,$2	010011	00100	00010	01000 00000 000000	=	4C824000
0x00000050	halt	111111	00000	00000	0000 0000 0000 0000	=	FC000000

上图为本次实验运行的指令集

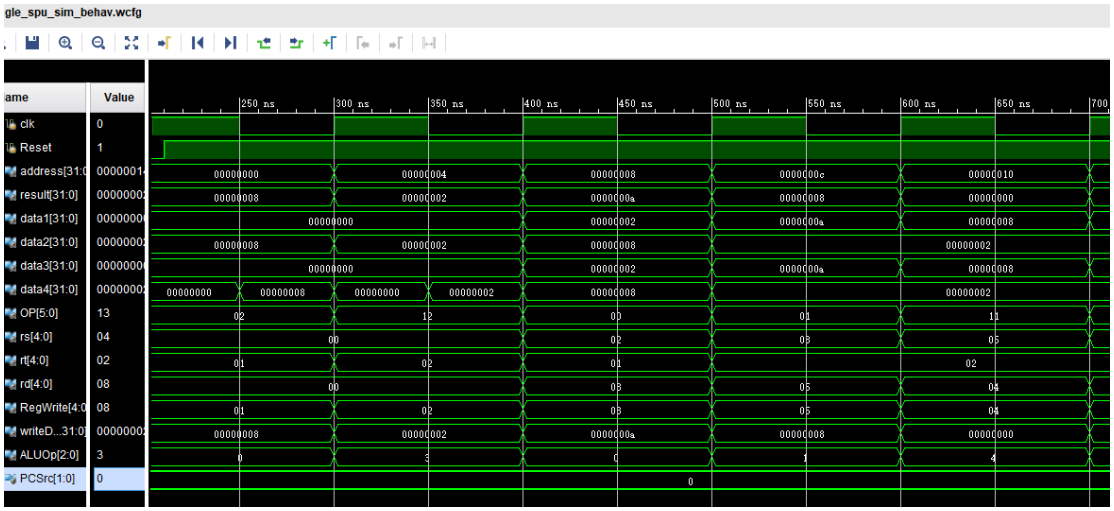


上图为指令集的输入文件



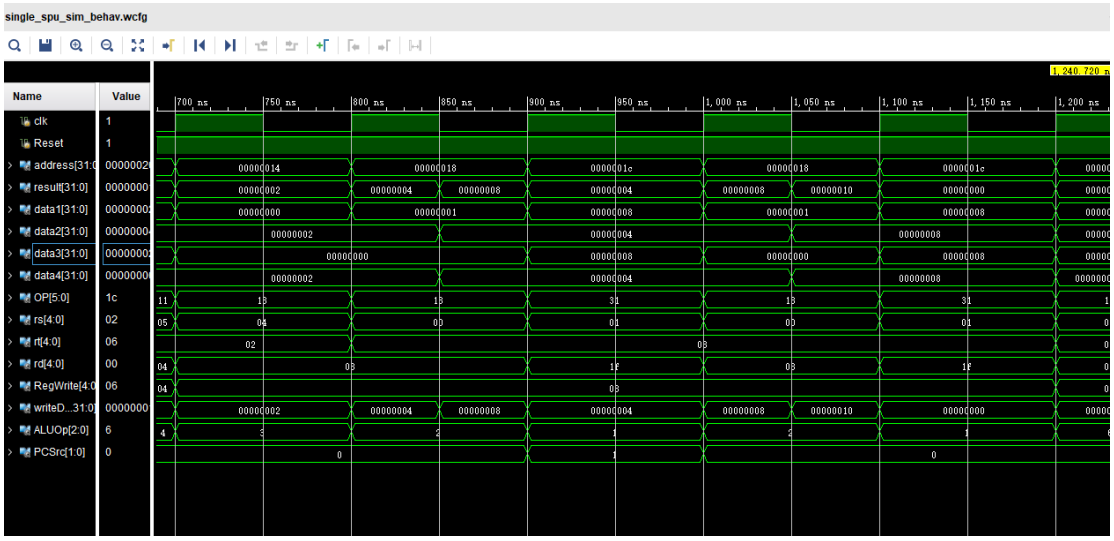
上图为运行上述16条指令构成的波形图。

以下为具体的波形图



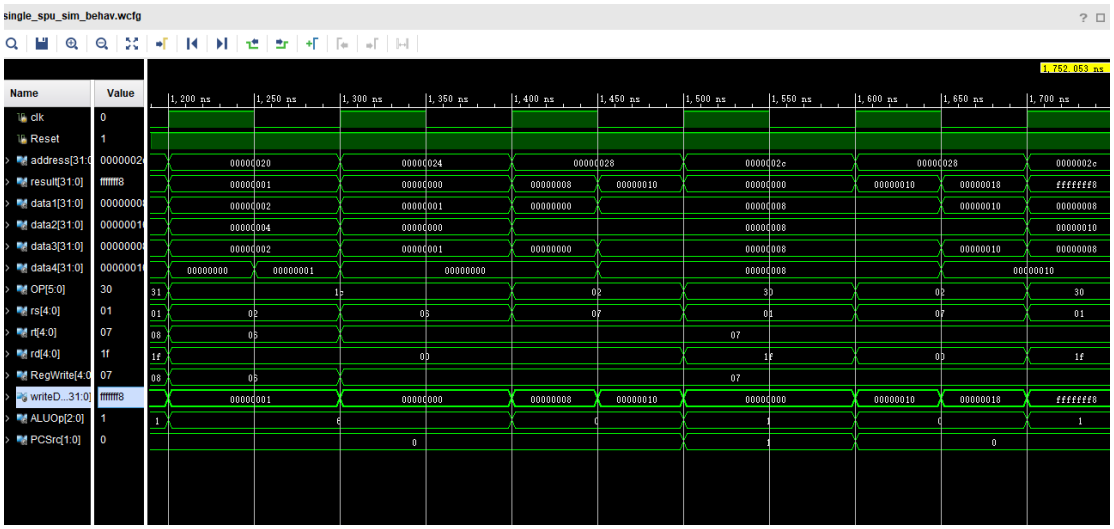
1-5 条指令

```
addiu $1,$0,8      $1=0+8=8;
ori $2,$0,2        $2=0|2=2;
add $3,$2,$1       $3=8+2=10;
sub $5,$3,$2       $5=10-2=8;
and $4,$5,$2       $4=8&2=0;
```



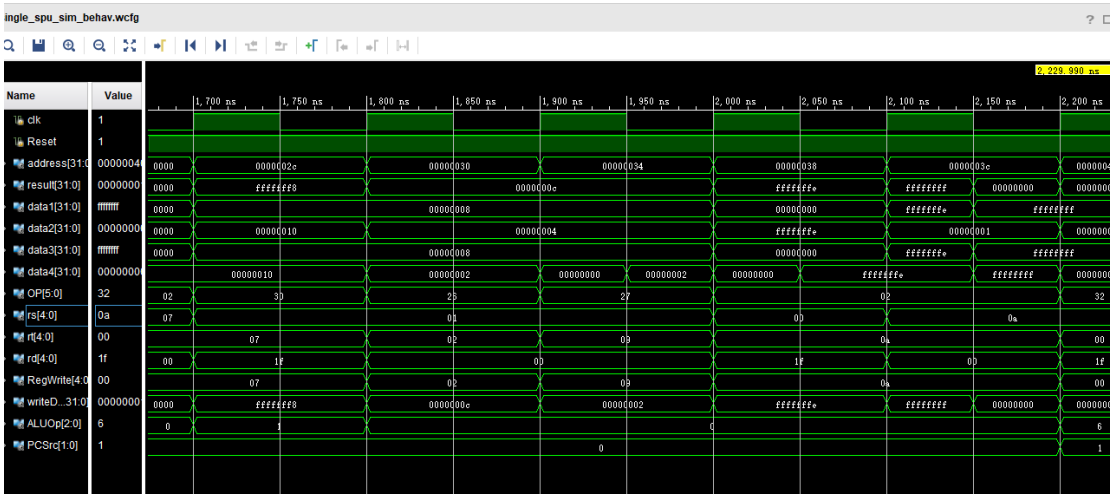
6-10条指令

```
or $8,$4,$2        $8=0|2=2;
sll $8,$8,1;       $8=2<<1=4;
bne $8,$1,-2;      $8=4,$1=8,4!=8,跳转到 0x18。
sll $8,$8,1;       $8=4<<1=8;
bne $8,$1,-2;      $8=4,$1=8,8==8,不跳转。
```



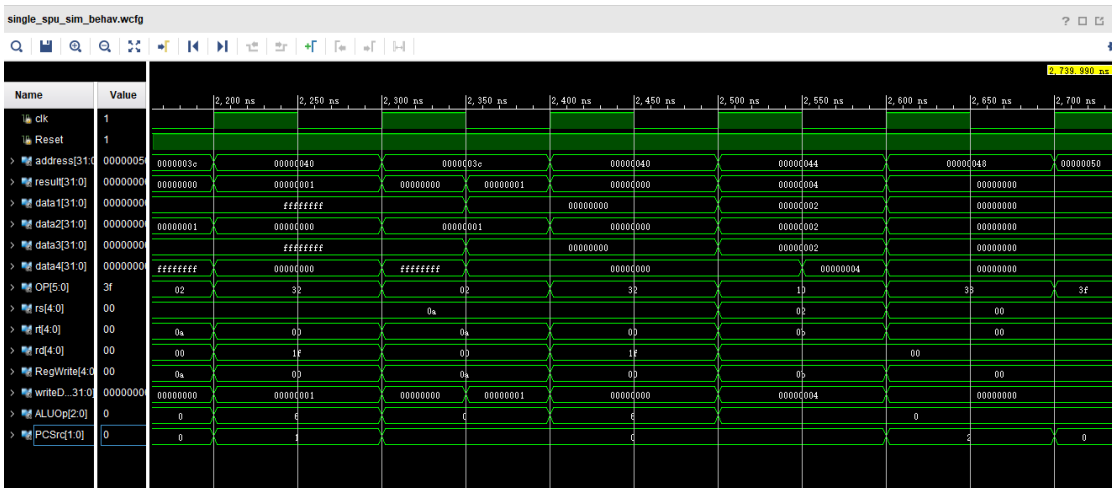
11-15条指令

```
slti $6,$2,4       $6=(2<4)=1;
slti $7,$6,0;      $7=(1<0)=0;
addiu $7,$7,8;     $7=0+8=8;
beq $7,$1,-2;      $7=8,$1=8,8==8,跳转到 0X28;
addiu $7,$7,8;     $7=8+8=16;
```



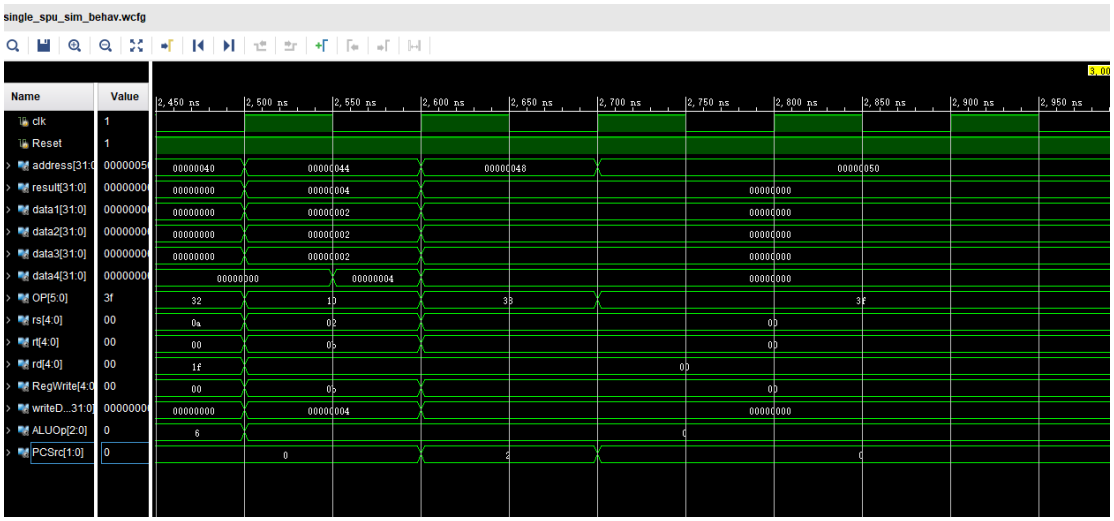
16-20条指令

```
beq $7,$1,-2        $7=16,$1=8,16!=8,不跳转;
sw $2,4($1)         内存$5=$2=2;
lw $9,4($1)         $9=内存$5=2;
addiu $10,$0,-2     $10=0-2=-2;
addiu $10,$10,1     $10=-2+1=-1;
```



21-25条指令

```
bltz $10,-2         $10=-1,-1<0,跳转到 0x3C
addiu $10,$0,-2     $10=-1+1=0;
bltz $10,-2         $10=0,0==0,不跳转
andi $11,$2,2       $11=2+2=4;
j 0x00000050        直接跳转到 0x50;
```



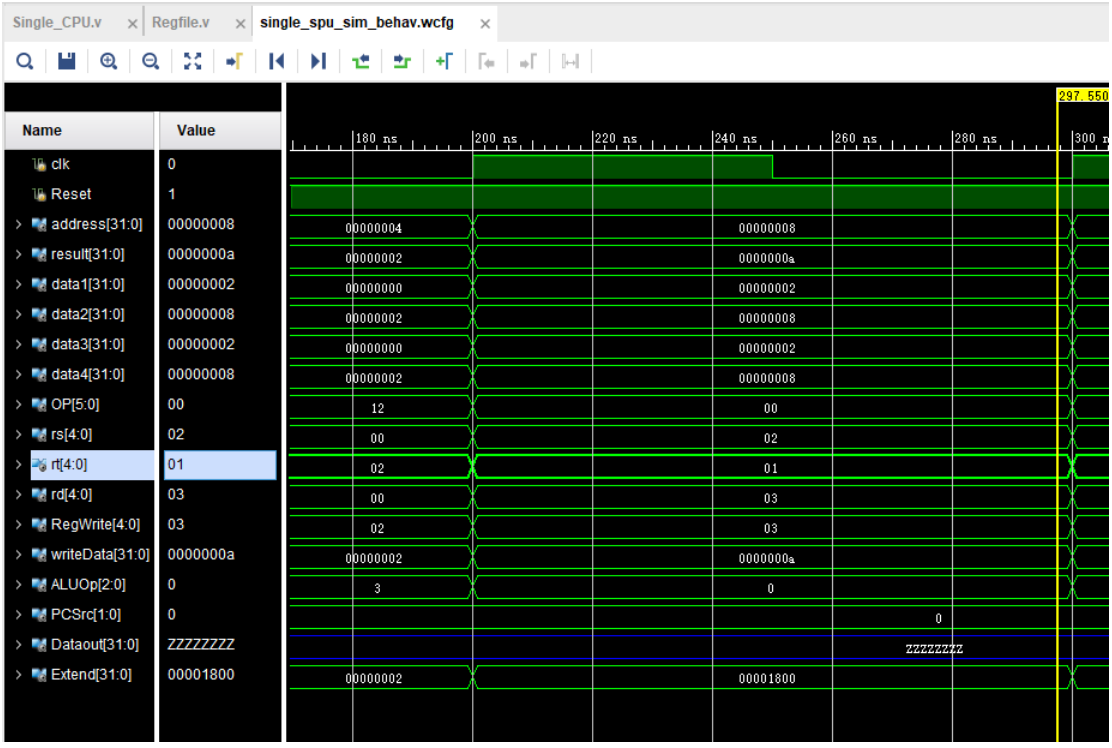
25-26条指令。

`j 0x00000050` 直接跳转到 0x50;

`halt` 停机;

下面以其中几条指令为例展示仿真的结果。

`add $3,$2,$1`



如图为add指令执行时的波形图

取指：add指令的地址为0x8，

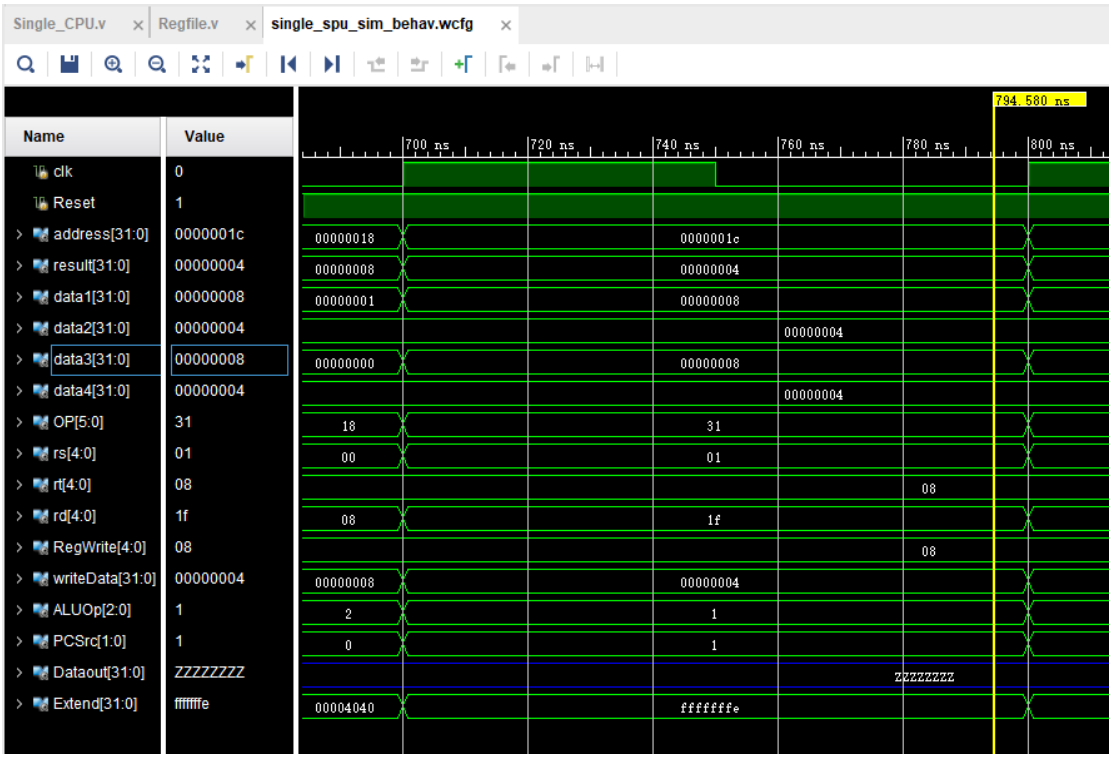
译码：rs为$2，rt为$1，rd为$3，op为000000，PCSrc为0。

寄存器读：从rs寄存器中取出的值为2，存在data3域，从rs寄存器中取出的值为3，存在data4域。

计算： $2的值被选入data1，$1的值被选入data2。ALUOp为0，即进行加法计算。data1和data2在ALU中进行计算，2+8=10，存入result。

寄存器写：result的数据通过选择器存入RegWrite，并写入rd，即$3。

bne $8,$1,-2



如图为bne指令执行时的波形图

取指：bne指令的地址为0x1c，

译码：rs为$1，rt为$8，op为111111，PCSrc为0。

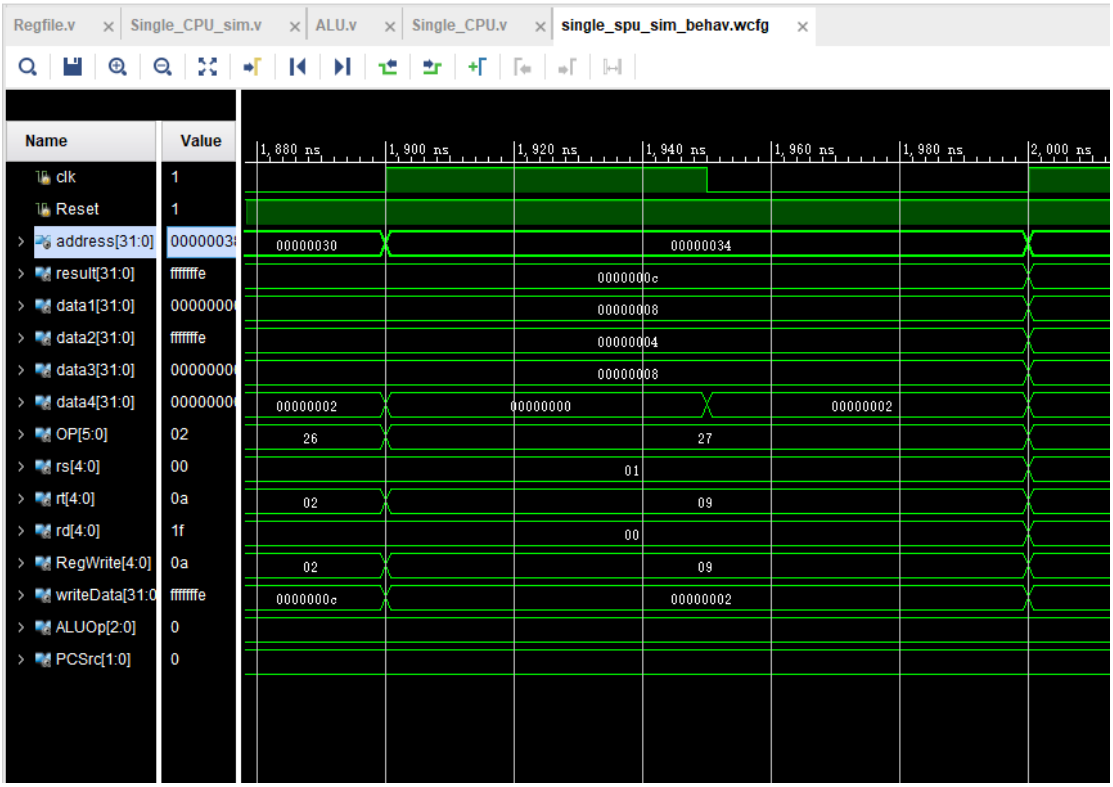
寄存器读：从rs寄存器中取出的值为8，存在data3域，从rs寄存器中取出的值为4，存在data4域。

计算： $1的值被选入data1，$8的值被选入data2。ALUOp为1，即进行减法计算。data1和data2在ALU中进行计算，8-4=4，存入result，zero=0，不为0。

地址操作：zero=0，两数不相等，进行跳转，PCSrc=1，PC=PC+4-2*4=PC-4。

地址操作：PCSrc=0，PC=PC+4。

Lw $9,4($1)



如图为执行lw指令时的波形图。

取指：Lw指令的地址为0x34。

译码：rs为$1，rt为$9，op为011011，PCSrc为0。

寄存器读：从rs寄存器中取出的值为8，存在data3域，rt寄存器在执行操作前还没有被赋值，取值为undefined，存在data4域。

计算：立即数为4，经过符号拓展后通过选择器被选入data2，而$1的值被选入data1。ALUOp为0，即进行加法计算。data1和data2在ALU中进行计算，8+4=12，存入result。

内存读写：lw指令只读内存不写内存，内存读的地址为ALU计算的结果，即result。从内存中去取出值2，存入Dataout。

寄存器写：dataout的数据通过选择器存入RegWrite，并写入rt，即$9。

地址操作：PCSrc=0，PC=PC+4。

3、写板部分

写板部分将CPU的运行结果写到basy3板上。根据要求，需要根据输入输出当前地址，下一步地址，rs，$rs，rt，$rt，ALU结果，数组总线等数据。写板的代码可以分为4个部分，分别为CPU部分，写板数据获取部分，时钟处理部分和数码管显示部分。

CPU部分上面已经介绍，接下来介绍其他部分。

1）写板数据获取

根据CPU运行的数据，和输入的输出类型信号(type)，生成四个数码管对应的输出数据A,B,C,D。$rs和$rt的数据在下降沿后可能会被修改，所以需要提前写入并保存。

输入：clk,type,NowAddress,NxtAddress,result,datas,datat,rs,rt, writeData。

输出：A,B,C,D。

核心代码：

```
always @(negedge clk)begin
    DS=datas;
    DT=datat;
end
always @( type or NowAddress or NxtAddress or rs or rt or result ) begin
    case (type)
        2'b00: begin
           A=NowAddress[7:4];B=NowAddress[3:0];
           C=NxtAddress[7:4];D=NxtAddress[3:0];
        end
        2'b01: begin
            A={3'b000,rs[4:4]};B=rs[3:0];
            C=DS[7:4];D=DS[3:0];
        end
        2'b10: begin
            A={3'b000,rt[4:4]};B=rt[3:0];
            C=DT[7:4];D=DT[3:0];
```

```
        end
        2'b11: begin
            A=result[7:4];B=result[3:0];
            C=writeData[7:4];D=writeData[3:0];
        end
    endcase
end
```

2） 时钟处理

根据要求，需要维护两个时钟，CPU时钟和写板扫描时钟。本部分需要根据系统时钟和按键输入得到上述两个时钟clk1，clk2。针对按键输入需要进行按键消抖处理。当连续4个时钟周期按键输入tag为1时，认为出现上升沿，clk=1。当连续100个时钟周期按键输入tag为0时，认为出现下降沿，clk=0。

输入：clk,tag,Reset,

输出：clk1,clk2

```
always @(posedge clk) begin
    cnt=cnt+1;
    if(cnt>50)begin
        clk2=!clk2;cnt=0;
    end
    if(tag==0)sum0=sum0+1;
    else sum0=0;
    if(tag==1)sum1=sum1+1;
    else sum1=0;
    if(sum1>4)now=1;
    if(sum0>100)now=0;
end
assign clk1=now;
```

3） 数码管显示

根据数码管显示时钟确定数码管显示的数值和扫描电路选通的数码管编号。后调用

SegLED得到8端数码管输出值。

输入：clk,A,B,C,D

输出：dispcode,dispseg

核心代码：

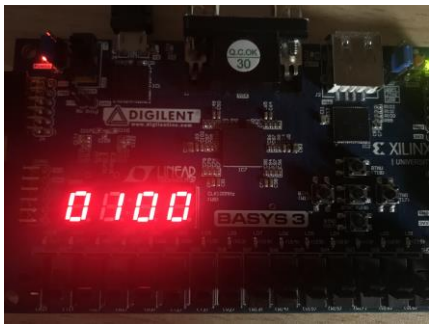
```
always @(posedge clk) begin
    i=i+1;
    if(i>3)i=0;
    case (i)
        0: begin
            tmp=A; dispseg=4'b0111;
        end
        1: begin
            tmp=B; dispseg=4'b1011;
        end
        2: begin
            tmp=C; dispseg=4'b1101;
        end
        3: begin
            tmp=D; dispseg=4'b1110;
        end
    endcase
end
SegLED seg(tmp,dispcode);
Endmodule
```

写板运行结果如下：

1、addiu $1,$0,8

当前PC=0，下条指令PC=4

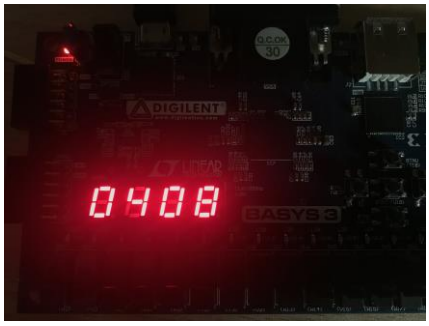
rs=0,$rs=0



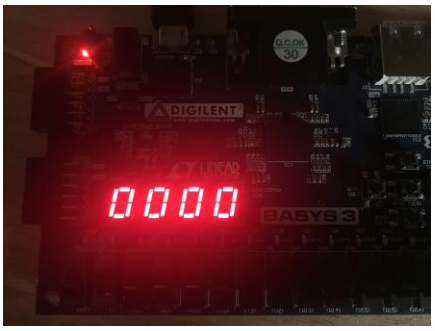
rt=1,$rt=0(undefine,初值赋0)

ALU输出=8，DB总线数据=8

2、ori $2,$0,2

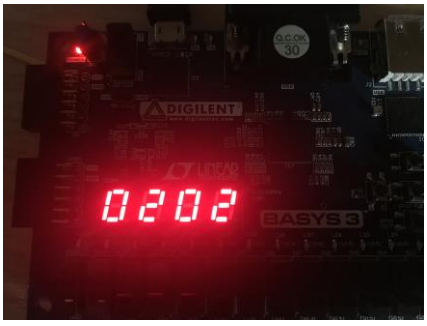


当前PC=4，下条指令PC=8



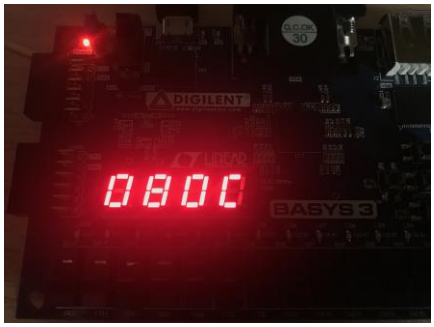
rs=0,$rs=0

rt=2,$rt=0(undefine,初值赋0)



ALU输出=2，DB总线数据=2

3、add $3,$2,$1



当前PC=8，下条指令PC=12（0xC）

rs=2,$rs=2

rt=1,$rt=8

ALU输出=10(0xA)，DB总线数据=10(0xA)

4、sub $5,$3,$2

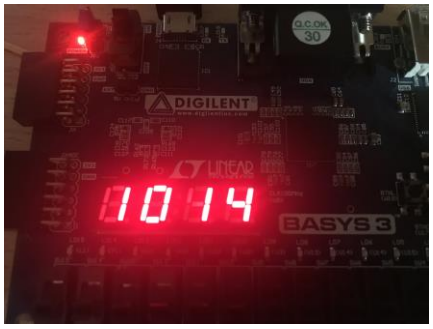
当前PC=12，下条指令PC=16(0x10)

rs=3,$rs=10(0xA)

rt=2,$rt=2

ALU输出=8，DB总线数据=8

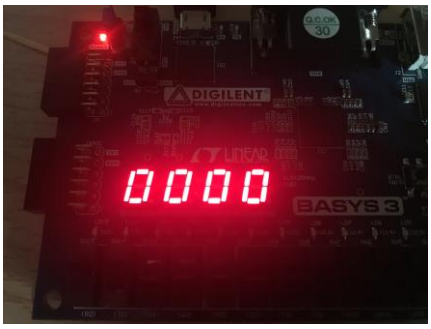
5、and $4,$5,$2



当前PC=16，下条指令PC=20(0x14)

rs=5,$rs=8

rt=2,$rt=2



ALU输出=0，DB总线数据=0

六. 实验心得

本次实验要求使用verilog语言和vivado软件，设计一个简单版的单周期CPU，使其能够运行规定的指令。实验主要分为两个部分，一是完成CPU设计，运行行为仿真得到波形图，二是将相关运行数据输出到basy3板上。

CPU的设计部分要求通过代码设计出一套能够运行相关指令的单周期CPU系统。主要需要完成指令地址处理，寄存器存取，内存存取，数值计算等部分。这部分相对比较简单，只要知道单周期CPU的原理，分模块将代码写出并调试即可。这部分的参考代码之前基本都有给出，可以借鉴参考代码，根据要求进行修改。

烧板部分需要从CPU的运行结果中根据实验要求提取相关数据，并将其显示到basy3板上。烧板部分没有参考代码，需要自己摸索。在这部分我遇到了一些问题。

烧板和仿真不同，仿真是在纯理论情况下的运行，而烧板则需要考虑实际情况。纯理论情况和现实情况有很多的不同，一些不是很正规的写法在仿真情况下运行正常，但是一烧板就会出现各种错误。

一开始我直接将按钮的输入接到CPU时钟，结果怎么按都没有反应。试了很久，突然想起来按键的时钟是不稳定的，按下后会在01之间长期波动。因此，我加了一段按键消抖

的代码，解决了这个问题。

烧板要求输出rs和rt在计算时的值，而直接输出的话，由于vivado的always语句只要满足条件就会运行，所以一些输出的值可能会是修改后的值。我将输出数据的写入调整到了下降沿，同时把寄存器写和内存写也调整到了下降沿，这样就避免了数据写入后再次执行对输出结果的影响。

而且由于vivado软件的特性，跑一次综合+实验需要花费近10分钟的时间。因此调试过程非常缓慢，耗费了大量的等待时间。

当然这次设计的CPU也有一些缺陷，比如只能执行部分的功能，比如只是单周期CPU，运行速度不够快。接下来还要做多周期CPU，还可以对这些缺陷进行一些改进。

通过本次实验，我熟悉了verilog语言的语法，熟悉了vivado软件的使用，也对单周期CPU的结构和功能有了更深入地认知。