



《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) : 16 计算机类 4 班

学生姓名 : 张家豪

学号 : 16337303

时 间 : 2017 年 12 月 15 日

成 绩 :

实验三：多周期CPU设计与实现

目 录

一、 实验目的	2
二、 实验内容	3
三、 实验原理	7
四、 实验器材	29
五、 实验过程与结果	30
六、 实验心得	35
附录：	37

一、 实验目的

- (1) 认识和掌握多周期数据通路原理及其设计方法；
- (2) 掌握多周期 CPU 的实现方法，代码实现方法；
- (3) 编写一个编译器，将 MIPS 汇编程序编译为二进制机器码；（详见附录）
- (4) 掌握多周期 CPU 的测试方法；
- (5) 掌握多周期 CPU 的实现方法。

二、实验内容

设计一个多周期 CPU，该 CPU 至少能实现以下指令功能操作。需设计的指令与格式如下：

==>算术运算指令

(1) add rd, rs, rt

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

功能： $rd \leftarrow -rs + rt$

(2) sub rd, rs, rt

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

完成功能： $rd \leftarrow -rs - rt$

(3) addi rt, rs, immediate

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

功能： $rt \leftarrow -rs + (\text{sign-extend})\text{immediate}$

==>逻辑运算指令

(4) or rd, rs, rt

010000	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能： $rd \leftarrow -rs | rt$

(5) and rd, rs, rt

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

功能： $rd \leftarrow -rs \& rt$

(6) ori rt, rs, immediate

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

功能: $rt \leftarrow -rs \mid (\text{zero-extend})\text{immediate}$

==>移位指令

(7) sll rd, rt, sa

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

功能: $rd \leftarrow -rt \ll (\text{zero-extend})sa$, 左移 sa 位, ($\text{zero-extend})sa$

==>比较指令

(8) slt rd, rs, rt 带符号数

100110	rs(5 位)	rt(5 位)	rd(5 位)	reserved
--------	---------	---------	---------	----------

功能: if ($rs < rt$) $rd = 1$ else $rd = 0$, 具体请看表 2 ALU 运算功能表, 带符号

(9) slti rt, rs, immediate 带符号

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

功能: if ($rs < (\text{sign-extend})\text{immediate}$) $rt = 1$ 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)

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

功能: $\text{memory}[rs + (\text{sign-extend})\text{immediate}] \leftarrow -rt$ 。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)

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

功能: $rt \leftarrow \text{memory}[rs + (\text{sign-extend})\text{immediate}]$ 。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 (说明: immediate 从 pc+4 开始和转移到的指令之间间隔条数)

110100	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: $\text{if}(rs=rt) pc \leftarrow pc + 4 + (\text{sign-extend})\text{immediate} \ll 2 \text{ else } pc \leftarrow pc + 4$

(13) bne rs,rt, immediate (说明: immediate 从 pc+4 开始和转移到的指令之间间隔条数)

110101	rs(5 位)	rt(5 位)	immediate(16 位)
--------	---------	---------	-----------------

功能: $\text{if}(rs!=rt) pc \leftarrow pc + 4 + (\text{sign-extend})\text{immediate} \ll 2 \text{ else } pc \leftarrow pc + 4$

(14) bgtz rs,immediate

110110	rs(5 位)	00000	immediate
--------	---------	-------	-----------

功能: $\text{if}(rs>0) pc \leftarrow pc + 4 + (\text{sign-extend})\text{immediate} \ll 2 \text{ else } pc \leftarrow pc + 4$

=>跳转指令

(15) j addr

111000	addr[27:2]
--------	------------

功能: $pc \leftarrow \{(pc+4)[31:28],addr[27:2],0,0\}$, 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。

(16) jr rs

111001	rs(5 位)	未用	未用	reserved
--------	---------	----	----	----------

功能: $pc \leftarrow rs$, 跳转。

=>调用子程序指令

(17) jal addr

111010	addr[27..2]
--------	-------------

功能: 调用子程序, $pc \leftarrow \{(pc+4)[31:28],addr[27:2],0,0\}$; $\$31 \leftarrow pc+4$, 返回地址设置; 子程序返回, 需用指令 $jr \$31$ 。跳转地址的形成同 j $addr$ 指令。

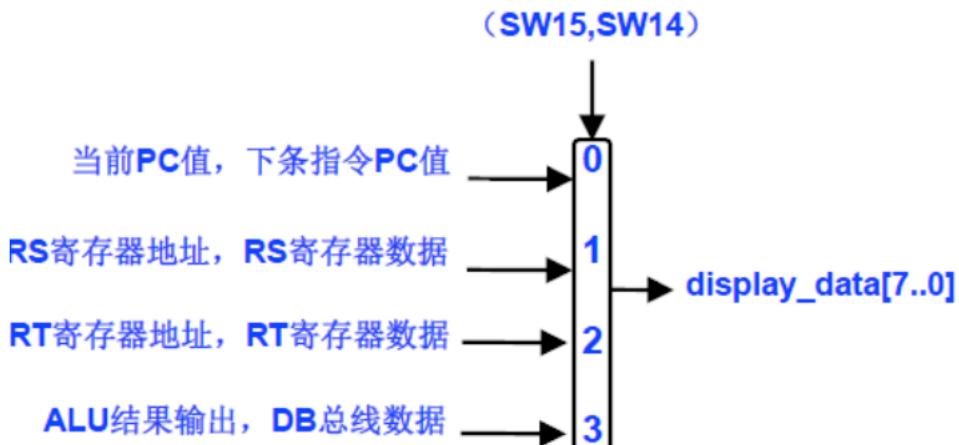
=>停机指令

(18) halt (停机指令)

111111	00000000000000000000000000(26 位)
--------	----------------------------------

不改变 pc 的值, pc 保持不变。

2、构造完单周期 CPU 后，按照要求实现 Basys3 板的烧制，效果图如下。



有 4 种输出模式，指令采用单步执行控制

三、实验原理

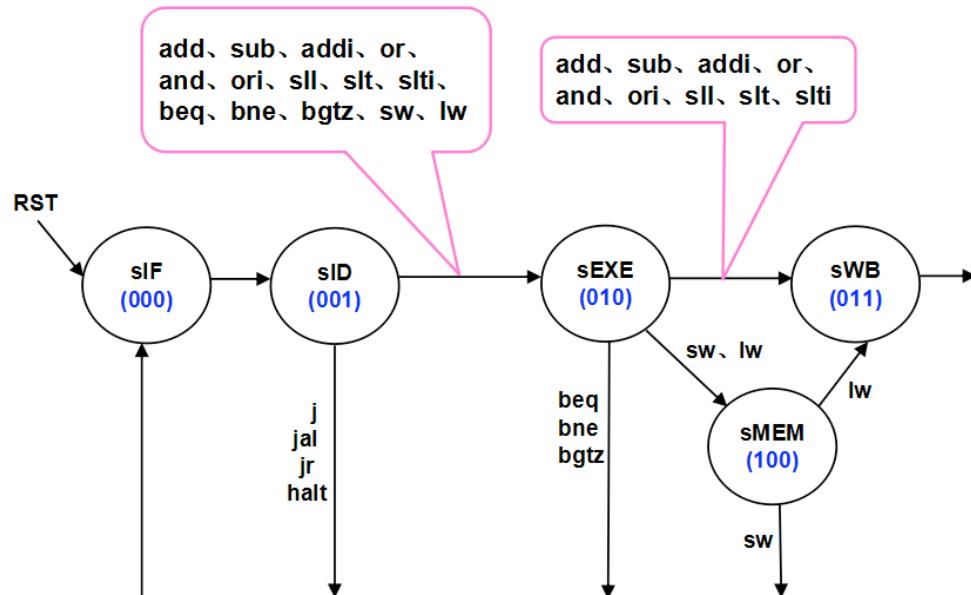
多周期 CPU 指的是将整个 CPU 的执行过程分成几个阶段，每个阶段用一个时钟去完成，然后开始下一条指令的执行，而每种指令执行时所用的时钟数不尽相同，这就是所谓的多周期 CPU。CPU 在处理指令时，一般需要经过以下几个阶段：

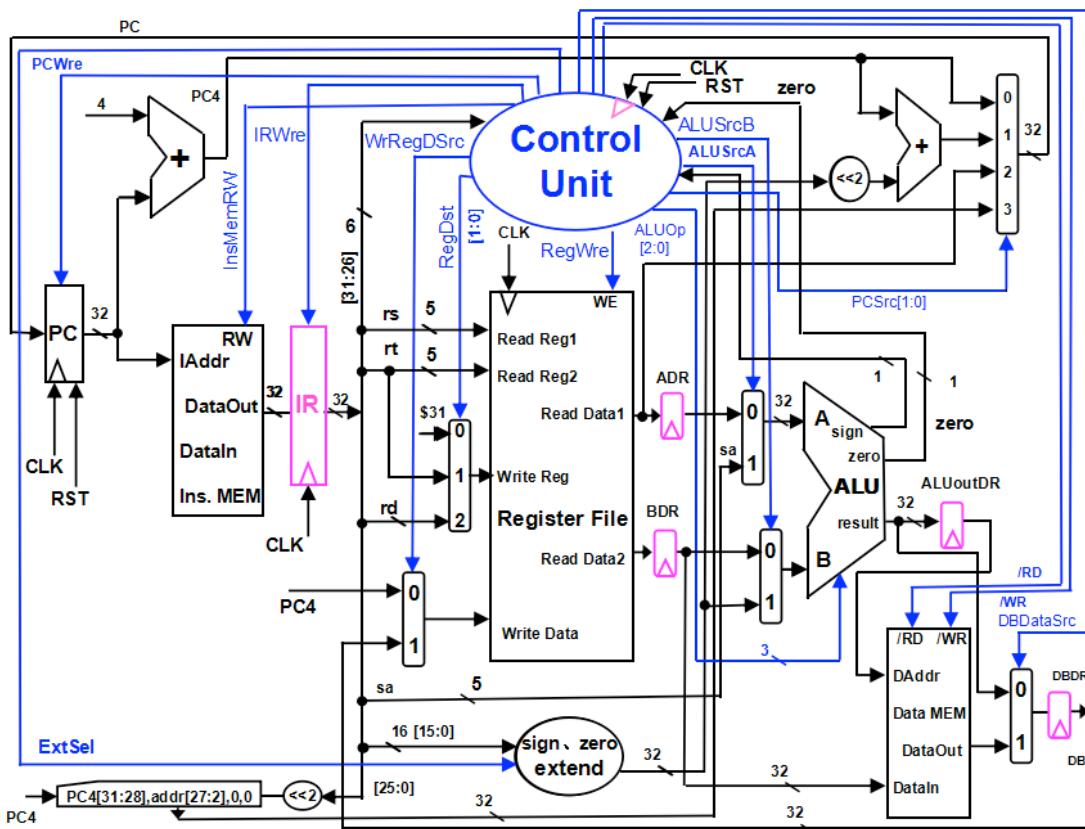
- (1) 取指令(IF): 根据程序计数器 pc 中的指令地址，从存储器中取出一条指令，同时，pc 根据指令字长度自动递增产生下一条指令所需要的指令地址，但遇到“地址转移”指令时，则控制器把“转移地址”送入 pc，当然得到的“地址”需要做些变换才送入 pc。
- (2) 指令译码(ID): 对取指令操作中得到的指令进行分析并译码，确定这条指令需要完成的操作，从而产生相应的操作控制信号，用于驱动执行状态中的各种操作。
- (3) 指令执行(EXE): 根据指令译码得到的操作控制信号，具体地执行指令动作，然后转移到结果写回状态。
- (4) 存储器访问(MEM): 所有需要访问存储器的操作都将在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

给出存储器的数据地址，把数据写入到存储器中数据地址所指定的存储单元或者从存储器中得到数据地址单元中的数据。

(5) 结果写回(WB): 指令执行的结果或者访问存储器中得到的数据写回相应的寄存器中。

状态转移与指令的关系如下图所示。





由图可知可以将单周期CPU分为七大模块，分别是PC模块、InstructionMemory

指令存储器模块、RegisterFile寄存器堆模块、SignZeroExtend符号拓展模块、ControlUnit控制单元、ALU模块和DataMemory数据内存模块，此外还需要组合逻辑电路构成的以下器件：1、32位4选1数据选择器用来生成next_PC（下一个PC的值）；2、5位3选1数据选择器来选择写入哪个寄存器；3、4个2选1数据选择器分别用来选择写回寄存器的值、送去ALU进行运算的第一个操作数的值（对应图中的A）、送去ALU进行运算的第二个操作数的值（对应图中的B）以及选择返回寄存器堆的值来自于ALU的计算结果还是来自于内存（Load指令）。最后还需要5个时钟信号触发的寄存器进行延时操作，确保每个周期只做对应的事情。

而后烧板需要消抖debounce模块和显示display模块。消抖是为了消除按键按下过程中电平短时间内的多次跳动，显示模块负责输出Basys3板上的位选信号和数码管显示信号。

最后通过top模块调用CPU模块和消抖模块以及显示模块完成本次实验。

下面逐一介绍各模块功能：

1、PC模块：接收输入时钟信号clk、重置信号Reset、写PC信号PCWre、和由代表下一条PC地址的32位pc_in信号。输出信号pc_out根据是时序触发，在clk的上升沿或Reset信号的下降沿触发。

代码如下：

```
'timescale 1ns / 1ps
module PC(
    input clk, Reset, PCWre,
    input [31:0] pc_in,
    output reg [31:0] pc_out
);
initial begin
    pc_out = 32'h00000000;
end
always @(posedge clk or negedge Reset) begin
if(Reset == 0) begin
    pc_out <= 0;
end
else if(PCWre)
    pc_out <= pc_in;
end
endmodule
```

2、Pcselector模块：接收2位PCSrc来决定输出pc_out(next_PC)的来源，next_PC输入信号有4个来源，分别是PC+4、immediate立即数+4、ReadData1来自寄存器堆的数据和j_address跳转信号。

代码如下：

```
'timescale 1ns / 1ps
```

```

module PCselector(
    input [1:0] PCSrc,
    input [31:0] pc4,
    input [31:0] immediate,
    input [31:0] ReadData1,
    input [25:0] j_address,
    output [31:0] pc_out
);

    assign pc_out = (PCSrc == 2'b00) ? pc4 : (PCSrc == 2'b01)? pc4 + immediate * 4 :
    (PCSrc == 2'b10)? ReadData1 : {pc4[31:28],j_address[25:0],2'b00} ;

endmodule

```

3、InstructionMemory指令存储器模块（在这里命名为ROM模块，因为指令在本次实验中只读不修改）：输入信号InsMemRW恒为0（只读状态），和32位addr信号（PC值），通过PC找到指令后输出32位指令信号dataOut。

此次代码预存指令如下：

地址	汇编程序	指令代码					16 进制数代码
		op (6)	rs(5)	rt(5)	rd(5)/immediate (16)		
0x00000000	addi \$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	or \$3,\$2,\$1	010000	00010	00001	00011 000 0000 0000		40411800
0x0000000C	sub \$4,\$3,\$1	000001	00011	00001	00100 000 0000 0000		04612000
0x00000010	and \$5,\$4,\$2	010001	00100	00010	00101 000 0000 0000		44822800
0x00000014	sll \$5,\$5,2	011000	00000	00101	00101 00010 00 0000		60052880
0x00000018	beq \$5,\$1,-2(=, 转 14)	110100	00101	00001	1111 1111 1111 1110		D0A1FFE
0x0000001C	jal 0x00000048	111010	00000	00000	0000 0000 0001 0010		E8000012
0x00000020	slt \$8,\$12,\$1	100110	01100	00001	01000 000 0000 0000		99814000

0x000000024	addi \$14,\$0,-2	000010	00000	01110	1111 1111 1111 1110		080EFFFE
0x000000028	slt \$9,\$8,\$14	100110	01000	01110	01001 000 0000 0000		990E4800
0x00000002C	slti \$10,\$9,2	100111	01001	01010	0000 0000 0000 0010		9D2A0002
0x000000030	slti \$11,\$10,0	100111	01010	01011	0000 0000 0000 0000		9D4B0000
0x000000034	add \$11,\$11,\$8	000000	01011	01000	01011 000 0000 0000		01685800
0x000000038	bne \$11,\$2,-2 (≠, 转 34)	110101	01011	00010	1111 1111 1111 1110		D562FFFE
0x00000003C	addi \$2,\$2,-1	000010	00010	00010	1111 1111 1111 1111		0842FFFF
0x000000040	bgtz \$2,-2 (>0, 转 3C)	110110	00010	00000	1111 1111 1111 1110		D840FFFE
0x000000044	j 0x00000054	111000	00000	00000	0000 0000 0001 0101		E0000015
0x000000048	sw \$2,4(\$1)	110000	00001	00010	0000 0000 0000 0100		C0220004
0x00000004C	lw \$12,4(\$1)	110001	00001	01100	0000 0000 0000 0100		C42C0004
0x000000050	jr \$31	111001	11111	00000	0000 0000 0000 0000		E7E00000
0x000000054	halt	111111	00000	00000	0000000000000000	=	FC000000

代码如下：

```

`timescale 1ns / 1ps

module ROM (InsMemRW, addr, dataOut);

    input InsMemRW ;
    input [31:0] addr;
    output [31:0] dataOut;
    reg[7:0] rom[99:0];

initial begin
    $readmemb("E:/rom_data.txt",rom);
end

assign dataOut[31:24] = (InsMemRW == 0) ? rom[addr] : 8'h00;
assign dataOut[23:16] = (InsMemRW == 0) ? rom[addr + 1] : 8'h00;
assign dataOut[15:8] = (InsMemRW == 0) ? rom[addr + 2] : 8'h00;
assign dataOut[7:0] = (InsMemRW == 0) ? rom[addr + 3] : 8'h00;

endmodule

```

3、 IR指令寄存器模块（具有译码功能）：输入信号有时钟信号clk、IRWre写控制

信号和32位指令 Instruction 信号，所有的输出都是clk下降沿触发，从 Instruction 信号分出6位操作码 opCode 信号、3个寄存器序号信号 rs、rt、rd 和16位立即数 immediate 信号、26位 j_address 跳转指令的跳转地址信号以及 移位指令时要用到的32位 sa 偏移位数信号。

代码如下：

```

`timescale 1ns / 1ps

module IR( clk, IRWre, Instruction, opCode, rs, rt, rd, immediate, j_address, sa);

    input clk;
    input IRWre;
    input [31:0] Instruction;
    output [5:0] opCode;
    output [4:0] rs, rt, rd;
    output [15:0] immediate;
    output [25:0] j_address;
    output [31:0] sa;

    reg [31:0] operation;

    always @ (negedge clk) begin
        if(IRWre)
            operation <= Instruction;
    end

    assign j_address = operation[25:0];
    assign sa[31:8] = 24'h000000;
    assign sa[7:5] = 3'b000;
    assign sa[4:0] = operation[10:6];
    assign opCode = operation[31:26];

```

```

assign rs = operation[25:21];
assign rt = operation[20:16];
assign rd = operation[15:11];
assign immediate = operation[15:0];

endmodule

```

4、 ControlUnit控制单元模块：为了实现多周期的不同阶段的控制信号实时变化，这里的输入信号包括了时钟信号clk和3位过程变量state来决定各个控制信号的发出。此外，输入信号还有指令存储器输出的6位信号opCode和来自ALU模块的输出zero和sign信号，这三个信号也参与决定其他控制信号的输出。

代码如下：

```

`timescale 1ns / 1ps
module controlUnit(
    input [5:0] opcode,
    input zero, sign, clk, Reset,
    output reg InsMemRW,
    output reg ExtSel, PCWre, IRWre, WrRegDSrc, RegWre, ALUSrcA, ALUSrcB, DBDataSrc, RD, WR,
    output reg [1:0] RegDst, PCSrc,
    output reg [2:0] ALUOp,
    output reg [2:0] state
);

parameter [2:0]      sif = 3'b000,
                     sid = 3'b001,
                     sexe = 3'b010,
                     swb = 3'b011,
                     smem = 3'b100;

parameter [5:0]      add = 6'b000000,
                     sub = 6'b000001,
                     addi = 6'b000010,
                     Or = 6'b010000,
                     And = 6'b010001,
                     ori = 6'b010010,
                     sll = 6'b011000,
                    slt = 6'b100110,
                     slti = 6'b100111,
                     sw = 6'b110000,
                     lw = 6'b110001,
                     beq = 6'b110100,
                     bne = 6'b110101,
                     bgtz = 6'b110110,
                     j = 6'b111000,
                     jr = 6'b111001,

```

```

        jal = 6'b111010,
        halt = 6'b111111;

// reg [2:0] state, next_state;

initial begin
    PCWre = 1;
    InsMemRW = 0;
    IRWre = 0;
    WrRegDSrc = 0;
    RegWre = 0;
    ALUSrcB = 0;
    DBDataSrc = 0;
    RD = 0;
    WR = 0 ;
    ExtSel = 2'b11;
    RegDst = 2'b11;
    PCSrc = 2'b00;
    ALUOp = 0;
    state = swb;
end
fuyjyvyvijv 20:46:01
always @(posedge clk or negedge Reset) begin
    if (Reset == 0) begin
        state <= sif;
    end else begin
        case(state)
            sif: state <= sid;
            sid: begin
                case (opcode[5:3])
                    3'b111: state <= sif; // j, jal, jr, halt
                    default: state = sexe;//other operation
                endcase
            end
            sexe: begin
                case (opcode[5:2])
                    4'b1101: state <= sif;
                    4'b1100: state <= smem;
                    default: state <= swb;
                endcase
            end
            smem: begin
                case (opcode[0])
                    1'b1: state <= swb;
                    1'b0: state <= sif;
                endcase
            end
            swb: state <= sif;
            default: state <= sif;
        endcase
    end
end
always @(state) begin
    if ( (state == sid && opcode[5:2] == 4'b1110) || (state == sexe && opcode[5:2] == 4'b1101) ||
(state == smem && opcode[0] == 0) || state == swb) PCWre <= 1;
    else PCWre <= 0;

    InsMemRW <= 0;

    if (state == sif) IRWre <= 1;
    else IRWre <= 0;

    if (opcode == jal) WrRegDSrc <= 0;
    else WrRegDSrc <= 1;

    if (state == swb || opcode == jal ) RegWre <= 1;
    else RegWre <= 0;

```

```

if (opcode == sll) ALUSrcA <= 1;
else ALUSrcA <= 0;

ALUSrcB <= 1;
if (opcode == addi || opcode == ori || opcode == sw || opcode == lw || opcode == slti)
    else ALUSrcB <= 0;

if (state == smem && opcode == sw) WR <= 0;
else WR <= 1;

if (state == smem && opcode == lw) RD <= 0;
else RD <= 1;

if (state == swb && opcode == lw) DBDataSrc <= 1;
else DBDataSrc <= 0;

if (opcode == ori) ExtSel <= 0;
else ExtSel <= 1;

if (opcode == jal) RegDst <= 2'b00;
else if (opcode == addi || opcode == ori || opcode == lw || opcode == slti) RegDst <= 2'b01;
else RegDst <= 2'b10;

case(opcode)
    j: PCSrc <= 2'b11;
    jal: PCSrc <= 2'b11;
    jr: PCSrc <= 2'b10;
    beq: begin
        if (zero) PCSrc <= 2'b01;
        else PCSrc <= 2'b00;
    end
    bne: begin
        if(zero == 0) PCSrc <= 2'b01;
        else PCSrc <= 2'b00;
    end
    bgtz: begin
        if(zero == 0 && sign == 0)      PCSrc <= 2'b01;
        else PCSrc <= 2'b00;
    end
    default: PCSrc <= 2'b00;
endcase

case(opcode)
    sub: ALUOp <= 3'b001;
    Or: ALUOp <= 3'b101;
    And: ALUOp <= 3'b110;
    ori: ALUOp <= 3'b101;
    slt: ALUOp <= 3'b011;
    slti: ALUOp <= 3'b011;
    sll: ALUOp <= 3'b100;
    beq: ALUOp <= 3'b001;
    bne: ALUOp <= 3'b001;
    bgtz: ALUOp <= 3'b001;
    default: ALUOp <= 3'b000;
endcase

if (state == sif) begin
    RegWre <= 0;
end
end
endmodule

```

5、 signZeroExtend立即数拓展模块：输入信号有16位立即数immediate信号，而输入信号ExtSel决定拓展类型，如果是ExtSel为0（ori指令）则输出信号out前16位补0，如果ExtSel为1则根据immediate的最高位补全32位输出信号out。

代码如下：

```
'timescale 1ns / 1ps

module signZeroExtend(immediate, ExtSel, out);
    input [15:0] immediate;
    input ExtSel;
    output [31:0] out;

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

endmodule
```

6、 Regfile寄存器堆模块：接收clk时钟信号（写操作时下降沿触发），接收RST信号（下降沿触发，用于清零寄存器堆），接收RegWre信号决定是否进行进行寄存器写操作（高电平工作），ReadReg1输入信号来自于指令的rs字段，ReadReg2输入信号来自于rt字段，WriteReg来自于5位2选1选择器输出的rt或rd字段。同时输入ReadReg1信号对应的寄存器的数据ReadData1和ReadReg2信号对应的寄存器的数据ReadData2。

代码如下：

```
'timescale 1ns / 1ps

module RegFile(CLK, RST, RegWre, ReadReg1, ReadReg2, WriteReg, WriteData,
```

```
ReadData1, ReadData2);

input CLK;
input RST;
input RegWre;
input [4:0] ReadReg1;
input [4:0] ReadReg2;
input [4:0] WriteReg;
input [31:0] WriteData;
output [31:0] ReadData1;
output [31:0] ReadData2;

reg [31:0] regFile[0:31];
integer j;
initial begin
  for( j = 0; j < 32 ; j = j + 1)
    regFile[j] <= 0;
end
integer i;

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

8、5位3选1数据选择器模块：

```
'timescale 1ns / 1ps

module DataSelect_3(signal, data1, data2, target);
    input [1:0] signal;
    input [4:0] data1;
    input [4:0] data2;
    output [4:0] target;

    assign target = (signal == 2'b00) ? 5'b11111 : (signal == 2'b01) ? data1 : data2;

endmodule
```

9、32位2选1数据选择器模块：

```
'timescale 1ns / 1ps

module DataSelect_2 (signal, data1, data2, target);
    input signal;
    input [31:0] data1;
    input [31:0] data2;
    output [31:0] target;

    assign target = (signal == 0)? data1 : data2;

endmodule
```

10、ALU模块：输入信号有来自选择器的2个32位操作数信号A和B和来自控制单元决定运算类型的3位ALUopcode信号，运算结果在输出信号result中，同时输出的还有zero和sign信号，这两个信号将返回controlUnit模块决定分支转移指令是否执行。

ALU功能表如图所示。

ALUOp[2..0]	功能	描述
000	$Y = A + B$	加
001	$Y = A - B$	减
010	$Y = (A < B) ? 1 : 0$	比较 A 与 B 不带符号
011	<pre>if (A < B && (A[31] == B[31])) Y = 1; else if (A[31] == 1 && B[31] == 0) Y = 1; else Y = 0;</pre>	比较 A 与 B 带符号
100	$Y = B \ll A$	B 左移 A 位
101	$Y = A \vee B$	或
110	$Y = A \wedge B$	与
111	$Y = A \Box B$	异或

代码如下：

```
'timescale 1ns / 1ps
module ALU32(ALUopcode, A, B, result, zero, sign);
    input [2:0] ALUopcode;
    input [31:0] A;
    input [31:0] B;
    output reg [31:0] result;
```

```

        output zero;
        output sign;
    always @( ALUopcode or A or B )
        begin
            case (ALUopcode)
                3'b000 : result = A + B;
                3'b001 : result = A - B;
                3'b100 : result = B << A;
                3'b101 : result = A | B;
                3'b110 : result = A & B;
                3'b010 : result = (A < B) ? 1 : 0 ;
                3'b011 : begin
                    if (A < B &&(( A[31] == 0 && B[31]==0) ||(A[31] == 1 &&
B[31]==1)))
                        result = 1;
                    else if (A[31] == 0 && B[31]==1)
                        result = 0;
                    else if ( A[31] == 1 && B[31]==0)
                        result = 1;
                    else
                        result = 0;
                end
                3'b111 : result = ((~A) & B) | ((~B) & A);
            endcase
        end
        assign zero = (result == 0) ? 1 : 0;
        assign sign = (result[31] == 0) ? 0 : 1;
    endmodule

```

11、DataMemory数据内存模块：输入信号有clk（写内存操作时下降沿触发），来自ALU的Daddr信号（ALU模块的result）决定了要操作的内存的地址，信号DataIn决定了写入的内容，信号RD和WR分别是读信号和写信号，输出信号DataOut是要写回寄存器堆的内容。

代码如下：

```
`timescale 1ns / 1ps

module DataMemmory(Daddr, DataIn, RD, WR, DataOut);

    input [31:0] Daddr;
    input [31:0] DataIn;
    input RD;
    input WR;
    output reg [31:0] DataOut;

    reg [7:0] ram [0:60];

    always@( RD or WR or Daddr or DataIn ) begin
        if( WR == 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
        else if(RD == 0) begin
            DataOut[7:0] <= ram[Daddr + 3];
            DataOut[15:8] <= ram[Daddr + 2];
            DataOut[23:16] <= ram[Daddr + 1];
            DataOut[31:24] <= ram[Daddr];
        end
    end
endmodule
```

12、DataLate数据延时模块：

```
`timescale 1ns / 1ps
```

```
module DataLate (input [31:0] i_data,
                 input clk,
                 output reg [31:0] o_data);

    always @(posedge clk) begin
        o_data = i_data;
    end
endmodule
```

13、display显示模块：输入信号有4个4位信号A，B，C，D，四个信号决定各个数码管上显示的数字，还有一个输入信号clk分频后用来决定位选信号的更新频率，输出是4位的位选信号pos和8位的数码管显示信号 dispcode。

代码如下：

```
`timescale 1ns / 1ps
module display(
    input [3:0] A,
    input [3:0] B,
    input [3:0] C,
    input [3:0] D,
    input clk,
    output reg [3:0] pos,
    output reg [7:0] dispcode
);

    reg [3:0] num;
    integer i;
    integer t;
    initial begin
        i = 0;
        t = 0;
    end
```

```
always @(posedge clk) begin
    if (t==1<<18) begin
        t = 0;
        i = (i+1)%4;
        case (i)
            0:begin
                pos = 4'b0111;
                num = A;
            end
            1:begin
                pos = 4'b1011;
                num = B;
            end
            2:begin
                pos = 4'b1101;
                num = C;
            end
            3:begin
                pos = 4'b1110;
                num = D;
            end
        endcase
        case (num)
            4'b0000 : dispcode = 8'b1000_0001; //0; '0'-亮灯，'1'-熄灯
            4'b0001 : dispcode = 8'b1100_1111; //1
            4'b0010 : dispcode = 8'b1001_0010; //2
            4'b0011 : dispcode = 8'b1000_0110; //3
            4'b0100 : dispcode = 8'b1100_1100; //4
            4'b0101 : dispcode = 8'b1010_0100; //5
            4'b0110 : dispcode = 8'b1010_0000; //6
            4'b0111 : dispcode = 8'b1000_1111; //7
            4'b1000 : dispcode = 8'b1000_0000; //8
            4'b1001 : dispcode = 8'b1000_0100; //9
            4'b1010 : dispcode = 8'b1000_1000; //A
            4'b1011 : dispcode = 8'b1110_0000; //b
            4'b1100 : dispcode = 8'b1011_0001; //C
            4'b1101 : dispcode = 8'b1100_0010; //d
            4'b1110 : dispcode = 8'b1011_0000; //E
            4'b1111 : dispcode = 8'b1011_1000; //F
            default : dispcode = 8'b1111_1111; //不亮
        endcase
    end
    else begin
```

```
t = t+1;  
end  
end  
endmodule
```

14、消抖模块：输入信号有系统时钟信号sclk和按键信号button，输出信号clk将用作CPU模块的clk，消抖的做法是每隔20ms才根据button的值更新clk信号。

代码如下：

```
'timescale 1ns / 1ps  
module xiaodou(  
    input  sclk,  
    input button,  
    output reg clk  
)  
    reg [19:0] delay;  
    initial delay = 0;  
    initial clk = 0;  
  
    always @ (negedge sclk)begin  
  
        if(delay == 20'b1111_0010_0010_0000)begin  
            delay <=0;  
            clk <= button;  
        end  
        else begin  
            delay <=delay+1;  
        end  
    end  
  
endmodule
```

15、CPU顶层模块：

调用关系如图所示

```

`v u0 - CPU (CPU.v) (18)
`v pc - PC (PC.v)
`v pselector - PCselector (PCselector.v)
`v rom - ROM (ROM.v)
`v ir - IR (IR.v)
`v controlunit - controlUnit (controlUnit.v)
`v u6 - signZeroExtend (signZeroExtend.v)
`v u0 - DataSelect_3 (DataSelect_3.v)
`v u5 - RegFile (RegFile.v)
`v Adelay - DataLate (DataLate.v)
`v Bdelay - DataLate (DataLate.v)
`v uu - DataSelect_2 (DataSelect_2.v)
`v u7 - DataSelect_2 (DataSelect_2.v)
`v u8 - DataSelect_2 (DataSelect_2.v)
`v u9 - ALU32 (ALU32.v)
`v Rdelay - DataLate (DataLate.v)
`v u10 - DataMemory (DataMemory.v)
`v u11 - DataSelect_2 (DataSelect_2.v)
`v WDdelay - DataLate (DataLate.v)

```

代码如下：

```

module CPU(
    input clk,Reset,
    output wire [5:0] opCode,
    output wire [4:0] rs, rt, rd, WriteReg,
    output wire [31:0] Instruction, ReadData1, ReadData2, result, WriteData, nextPC,
    output wire [31:0] cur_pc
);
    wire PCWre, IRWre,WrRegDSrc,DataMemRW, ALUSrcA, ALUSrcB, DBDataSrc,
    RegWre, InsMemRW, RD, WR, ExtSel;
    wire [2:0] state_out;
    wire [1:0] RegDst;
    wire [1:0] PCSrc;
    wire [2:0] ALUOp;
    wire zero,sign;
    wire [15:0] immediate;
    wire [25:0] j_address;
    wire [31:0] sa, A, B , Extseled, DataOut, ADR, BDR, InsLate ,WD, WDLate ,pc4, RDR;

    assign pc4 = cur_pc+ 4;

    PC pc(clk,Reset, PCWre, nextPC, cur_pc);
    PCselector pselector( PCSrc, pc4, Extseled, ReadData1, j_address, nextPC);

```

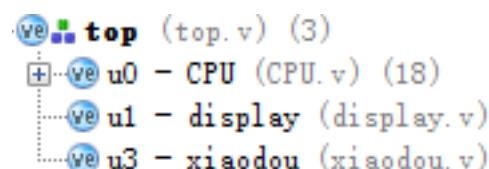
```

ROM rom(InsMemRW, cur_pc, Instruction);
IR ir(clk, IRWre, Instruction, opCode, rs, rt, rd, immediate, j_address, sa);
controlUnit controlunit(opCode , zero, sign, clk, Reset, InsMemRW, ExtSel, PCWre,
IRWre, WrRegDSrc, RegWre, ALUSrcA, ALUSrcB, DBDataSrc, RD, WR, RegDst, PCSrc,
ALUOp, state_out);
signZeroExtend u6(immediate, ExtSel, Extseled);
DataSelect_3 u0(RegDst, rt, rd, WriteReg);
RegFile u5(clk, Reset, RegWre, rs, rt, WriteReg, WriteData, ReadData1, ReadData2);
DataLate Adelay(ReadData1, clk, ADR);
DataLate Bdelay(ReadData2, clk, BDR);
DataSelect_2 uu(WrRegDSrc, pc4, WD , WriteData);
DataSelect_2 u7(ALUSrcA, ADR, sa, A);
DataSelect_2 u8(ALUSrcB, BDR, Extseled, B);
ALU32 u9(ALUOp, A, B, result, zero, sign);
DataLate Rdelay(result, clk, RDR);
DataMemory u10(RDR, ReadData2, RD, WR, DataOut);
DataSelect_2 u11(DBDataSrc, result ,DataOut,  WD);
DataLate WDdelay(WD, clk, WDLate);
endmodule

```

16、top顶层模块：

调用关系如图所示：



代码如下：

```

`timescale 1ns / 1ps
module top(
    input button,
    input sclk,
    input reset,
    input [1:0] select,

```

```

output [7:0] digit,
output [3:0] pos
);
wire clk190;
reg [31:0] PCoutr,PCnextr,ReadData1r,ReadData2r,resultr,WriteDatar;
reg [4:0] rsr,rtr;
reg [3:0] A,B,C,D;
wire clk;
wire [5:0] opCode;
wire [4:0] rs,rt,rd,WriteReg;
wire [31:0] Instruction,result,WriteData,ReadData1,ReadData2,cur_pc,nextPC;

CPU u0(
    .clk(clk),.Reset(reset),
    .opCode(opCode),
    .cur_pc(cur_pc), .Instruction(Instruction),.result(result), .WriteData(WriteData),
    .rs(rs),.rt(rt),.rd(rd),.WriteReg(WriteReg),
    .ReadData1(ReadData1),.ReadData2(ReadData2),
    .nextPC(nextPC)
);

display u1(
    .A(A),
    .B(B),
    .C(C),
    .D(D),
    .clk(sclk),
    .pos(pos),
    .dispcode(digit)
);

xiaodou u3(sclk,button,clk);

always @(posedge sclk)
begin
    if (clk == 0) begin
        PCoutr <= cur_pc;
        PCnextr <= nextPC;
        rsr <= rs;
        rtr <= rt;
        ReadData1r <= ReadData1;
        ReadData2r <= ReadData2;
        resultr <= result;
        WriteDatar <= WriteData;
    
```

```
end

case (select)
  0: begin
    A <= PCoutr[7:4];
    B <= PCoutr[3:0];
    C <= PCnextr[7:4];
    D <= PCnextr[3:0];
  end
  1: begin
    A <= {3'b000,rsr[4]};
    B <= rsr[3:0];
    C <= ReadData1r[7:4];
    D <= ReadData1r[3:0];
  end
  2: begin
    A <= {3'b000,rtr[4]};
    B <= rtr[3:0];
    C <= ReadData2r[7:4];
    D <= ReadData2r[3:0];
  end
  3: begin
    A <= resultr[7:4];
    B <= resultr[3:0];
    C <= WriteDatar[7:4];
    D <= WriteDatar[3:0];
  end
endcase
end
endmodule
```

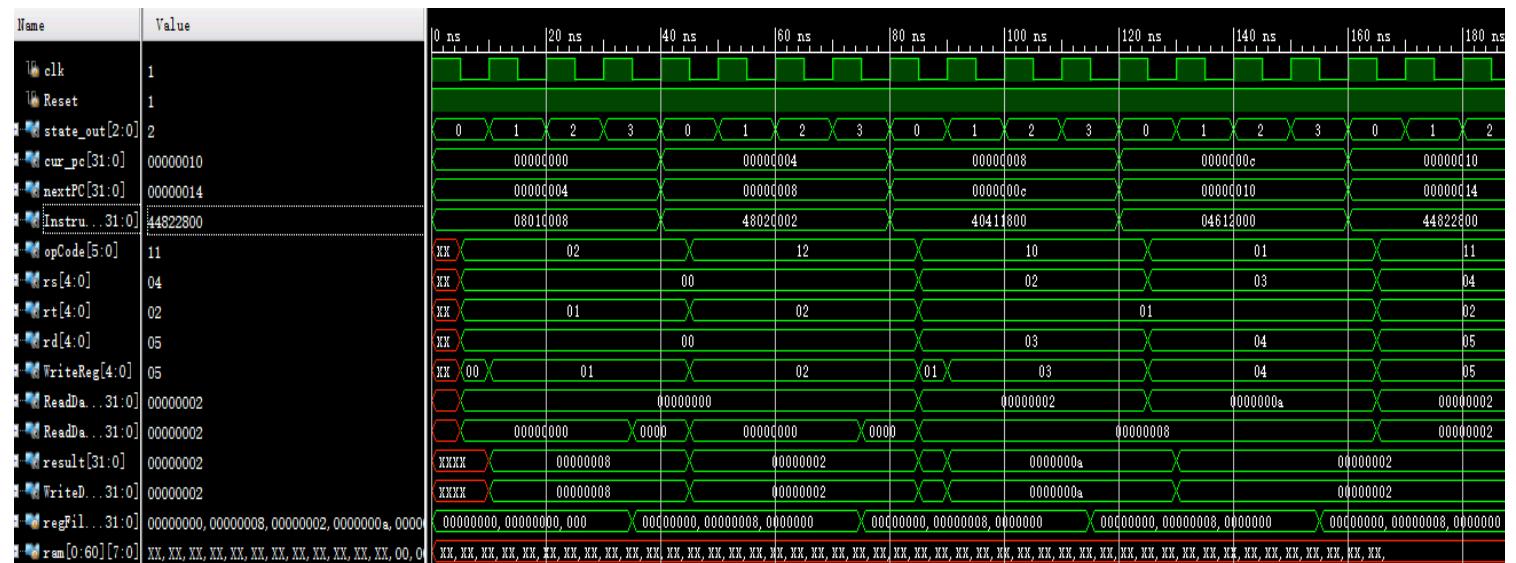
四、实验器材

电脑一台，Xilinx Vivado 软件一套，Basys3板一块。

五、实验过程与结果

(由于此次实验指令过多，所以重点讲解每张图中容易出错的点以及其正确性的验证)

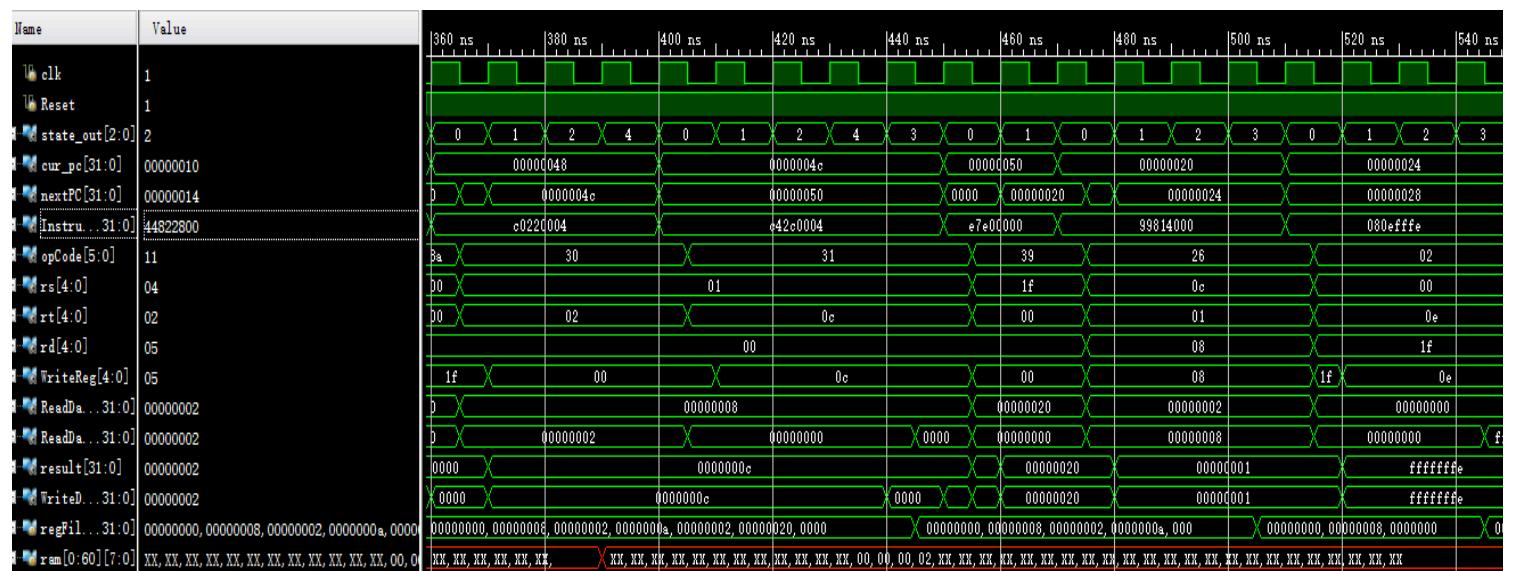
波形图展示：



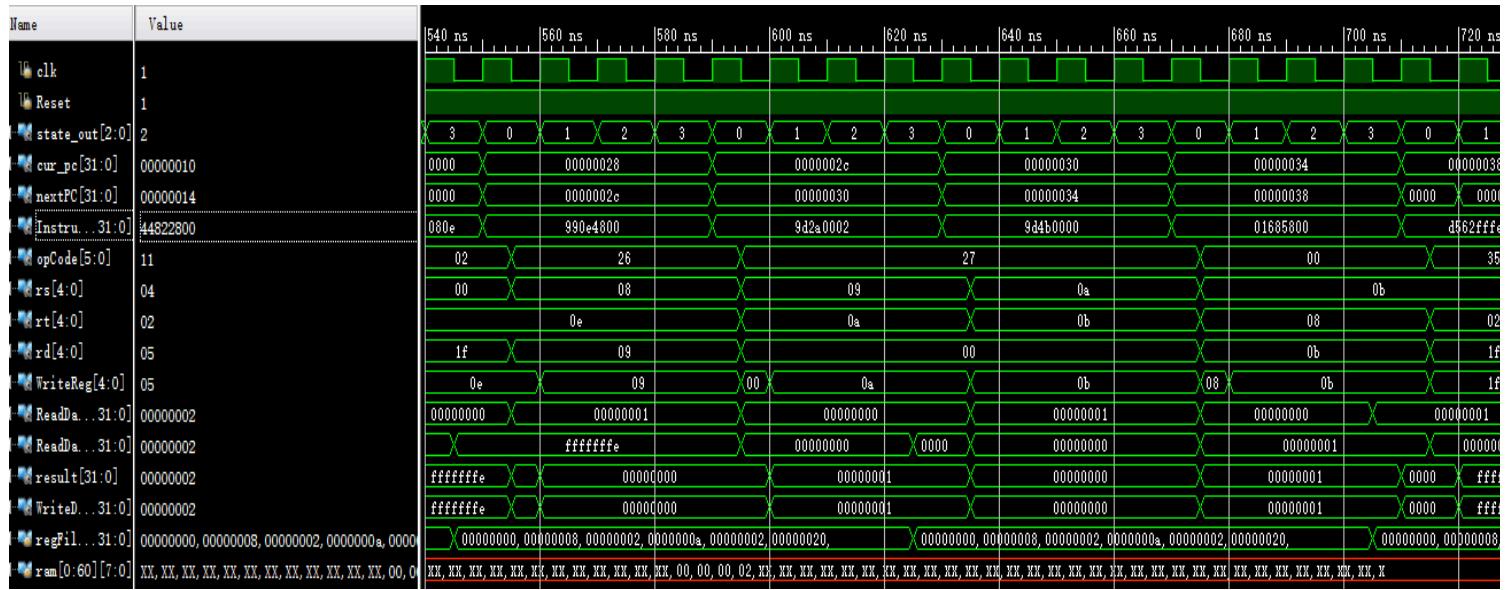
如图所示，执行第一条指令addi \$1,\$0,8时，指令表示为0x08010008，正确，ALU的result为8，在第4个时钟周期下降沿写回1号寄存器，也正确。



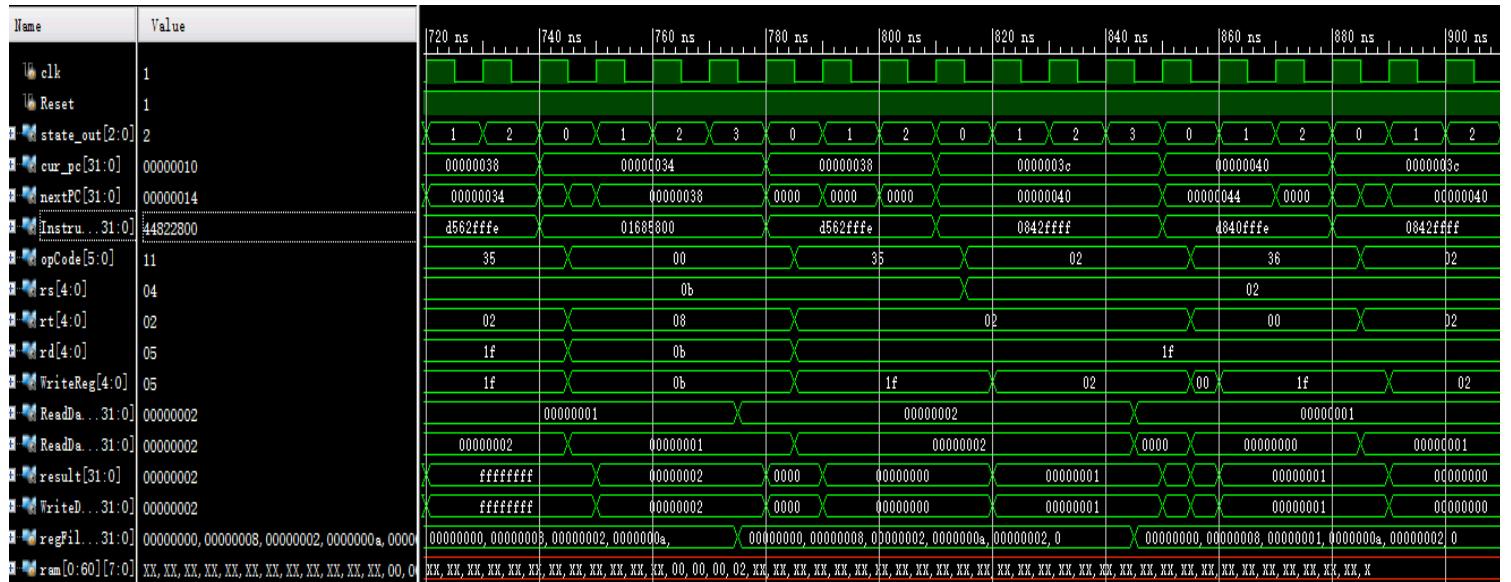
如图所示，第一次执行`beq $5,$1,-2(=,转14)`时，5号寄存器的值为8，与1号寄存器的值相等，于是PC跳转到0x14，正确，第二次执行`beq $5,$1,-2`时，5号寄存器的值变为0x20，不再与1号寄存器相等，于是PC跳转到0x1c，正确。



如图所示，执行jal 0x0000048后，PC跳转到0x48，正确（此时31号寄存器的值变为0x20，图中并未反映），执行jr \$31时，PC用31号寄存器取出的值，于是PC跳转到0x20，正确。



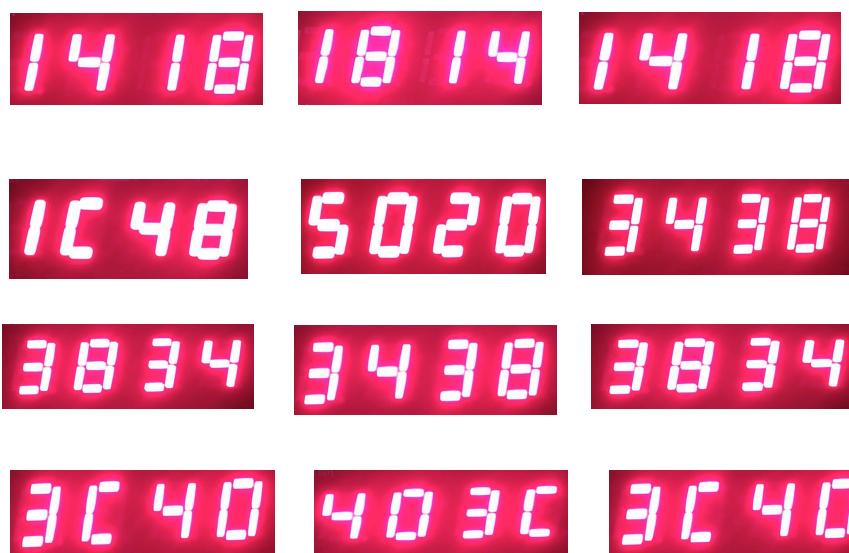
如图所示，执行 **slt \$9,\$8,\$14** 时，ALU的result信号为0；执行 **slt \$10,\$9,2** 时，
ALU的result信号为1；执行 **slt \$11,\$10,0** 时，ALU的result信号为0，三条指令
的运算结果都正确。



如图所示，第一次执行 **bne \$11,\$2,-2 (#转34)** 时，11号寄存器的值为1，与2号寄
存器的值不相等，于是PC跳转到0x34，正确，第二次执行 **bne \$11,\$2,-2** 时，11
号寄存器的值变为2，与2号寄存器相等，于是PC跳转到0x3c，正确。

如图所示，执行 **j 0x0000054** 后，PC跳转到0x54，执行halt停机指令，此后PC的值不变，CPU不进行任何有效操作。然而当Reset信号从1变为0的瞬间，PC和寄存器堆重置，PC变为0，各寄存器内容清零。（DataMemory的值不变因为在*这里将数据存储器视作外存*）

烧板结果展示：（同样由于图片过多，这里仅展示关键图片，即部分PC跳转过程）



The image shows two separate digital displays. The left display shows the number 4454, and the right display shows the number 5458. Both displays have a red background and white digits.

汇编器结果展示：

```
please input the assembly language(enter 'q' to quit)
addi $1,$0,8
ori $2,$0,2
or $3,$2,$1
sub $4,$3,$1
and $5,$4,$2
sll $5,$5,2
beq $5,$1,-2(=,转14)
jal 0x0000048
slt $8,$12,$1
addi $14,$0,-2
slt $9,$8,$14
slti $10,$9,2
slti $11,$10,0
add $11,$11,$8
bne $11,$2,-2 (=,转34)
addi $2,$2,-1
bgtz $2,-2 (>0,转3C)
j 0x0000054
sw $2,4($1)
lw $12,4($1)
jr $31
halt
000010000000000010000000000000001000
01001000000000001000000000000000000010
0100000001000001000110000000000000000
0000010001100001001000000000000000000
0100010010000010001010000000000000000
01100000000001010010100010000000000000
11010000000100101111111111111110
11101000000000000000000000000000000000
1001000110000001010000000000000000000
00001000000011101111111111111110
100100010000111001001000000000000000
100111010010101000000000000000000000
1001110101001011000000000000000000000
000000010110100001011000000000000000
1101010001001011111111111111110
00001000010000101111111111111111
11011000010000001111111111111110
111000000000000000000000000000000000000
11000000001000100000000000000000000000
11000100001011000000000000000000000000
111001111100000000000000000000000000000
1111110000000000000000000000000000000000
```

六、实验心得

1、有时候重新写会比修改更快，bug更少。我在写controlUnit控制单元的时候，想着从单周期CPU那里抄过来，然后加入state（状态信号）小改一下，但是我发现这样很耽误事，改动的地方还要一个地方一个地方的看，不能有缺漏，然而controlUnit那么大一个模块其实要发现哪里漏改了是相当费劲的一个事情，（例如这次老师给出的ALUop其实是和上次不一样的，看了好久才发现）更何况当初因为写controlUnit的时候就有一些不合理的地方。后来深刻理解了多周期CPU是如何运作之后，便推翻重做了，其实更快解决了问题，以后也要借鉴这次敢破敢立的精神，同时也要避免陷入定式思维。觉得改一下就行。

2、实验前理清指令的工作顺序，分配好要改变的信号是上升沿触发还是下降沿触发。这次实验做的时候一开始将PC设置为下降沿触发，PCWre信号在IF取指令阶段变为1，结果IR寄存器在要执行ID译码阶段还没取到指令，许多指令后面又还有延时的操作，这样一来后面的周期都不能正常的执行工作，于是到处小改，最后甚至想“取巧”，不要了IR寄存器，但是被老师一番指教之后才恍然大悟，可以在IF阶段之前就设置为1，然后让PC上升沿触发，这样后面边有充足的时间执行剩下的事情。

3、删除word文档中不常用的字体能够显著降低word文档崩溃的概率（针对word for Mac）。因为我在用苹果电脑打报告，感觉word文档的兼容性很差，而且我还加了很多图片进去，导致打报告的时候经常突然崩溃。在打单周期CPU的实验报告的时候就因为程序崩溃而“痛不欲生”，多周期的情况更甚。在网上怎么都找不到解决办法。后来我发现好像是字体的问题，每次改变字体再保存就容易崩溃。

我发现word for Mac每次都是重新加载mac系统的字体，由于字体容量太大，所以导致崩溃。知道后，通过删除不常用的字体显著得降低了程序崩溃的概率。

本次实验其实内容还是挺多的，加上很多模块也不能照搬照抄，还要加些模块进去，而且烧板确实不是一件简单事，前前后后也弄了很长时间，但是好在在做单周期CPU的时候打好了基础，所以对vivado的操作足够熟悉，对verilog的语法也大概了解，各种调试技巧也知道，所以感觉做多周期CPU时遇到的很多错误还是能够很快的发现的，深刻得感受到打好基础的重要性。

学期总结：

1、学习体验：本学期前半学期还是跟得很好的，但是后半学期因为自身原因，准备体育比赛而松懈下来了，多周期开始做时一知半解，找老师检查那天其实自己还是一知半解的，那天被老师说了很久，才豁然开朗，回去自己又细细体会了一些点，感觉简单的CPU设计还是学的不错的。

2、技能熟练度：从实验一到实验三，包括平时上的计组理论课，感觉自己对verilog和汇编语言的掌握还是相对不错的，语法规则和基本的操作都会用了。关于vivado的使用也比上学期数电课熟练很多，调看波形、单步执行来排除bug也会了，对于组合逻辑电路和时序电路有了更深刻的理解。

感觉几个星期的努力还是让自己受益匪浅的，体会到了“一分耕耘一分收获”这句话的魅力。

附录：

a、汇编器代码

```

//  

// main.cpp  

// 汇编语言翻译器  

//  

// Created by 张家豪 on 2018/1/2.  

// Copyright © 2018年 张家豪. All rights reserved.  

//  

// 实现一个简单的汇编器  

#include <iostream>  

#include <sstream>  

#include <string>  

#include <map>  

#include <stdio.h>  

using namespace std;  

map<string,int> mymap;  

void p1(string a){  

    size_t _1 = a.find('$');//第一个"$"  

    size_t _2 = a.find('$', _1 + 1);//第二个"$"  

    size_t _3 = a.find('$', _2 + 1);//第三个"$"  

    size_t __1 = a.find(',')//第一个", "  

    size_t __2 = a.find(',', __1 + 1);//第二个", "  

    string rd = a.substr(_1 + 1, __1 - _1 - 1);  

    string rs = a.substr(_2 + 1, __2 - _2 - 1);  

    string rt = a.substr(_3 + 1);  

    int d,s,t;  

    sscanf(rd.c_str(), "%d" , &d);  

    sscanf(rs.c_str(), "%d" , &s);  

    sscanf(rt.c_str(), "%d" , &t);  

    cout << (bitset<5>)s << (bitset<5>)t << (bitset<5>)d << "000000000000" << endl;
}  

void p2(string a){  

    size_t _1 = a.find('$');//第一个"$"  

    size_t _2 = a.find('$', _1 + 1);//第二个"$"  

    size_t __1 = a.find(',')//第一个", "  

    size_t __2 = a.find(',', __1 + 1)//第二个", "
}

```

```

string rt = a.substr(_1 + 1, _1 - _1 - 1);
string rs = a.substr(_2 + 1, _2 - _2 - 1);
string immediate = a.substr(_2 + 1);
int t,s,immediate_;
sscanf(rt.c_str(), "%d", &t);
sscanf(rs.c_str(), "%d", &s);
sscanf(immediate.c_str(), "%d", &immediate_);
cout << (bitset<5>)s << (bitset<5>)t << (bitset<16>)immediate_ << endl;
}

void p3(string a){
    size_t _1 = a.find('$');//第一个"$"
    size_t _2 = a.find('$', _1 + 1);//第二个"$"
    size_t __1 = a.find(',');
    size_t __2 = a.find(',', __1 + 1);
    string rd = a.substr(_1 + 1, __1 - _1 - 1);
    string rt = a.substr(_2 + 1, __2 - _2 - 1);
    string sa = a.substr(__2 + 1);
    int d,t,sa_;
    sscanf(rd.c_str(), "%d", &d);
    sscanf(rt.c_str(), "%d", &t);
    sscanf(sa.c_str(), "%d", &sa_);
    cout << "00000" << (bitset<5>)t << (bitset<5>)d << (bitset<5>)sa_ << "000000" <<
endl;
}

void p4(string a){
    size_t _1 = a.find('$');
    size_t _2 = a.find('$', _1 + 1);
    size_t __1 = a.find(',');
    size_t __2 = a.find(')');
    string rt = a.substr(_1 + 1, __1 - _1 - 1);
    string rs = a.substr(_2 + 1, __2 - _2 - 1);
    string immediate = a.substr(__2 + 1, __1 - __2 - 1);
    int s,t,immediate_;
    sscanf(rt.c_str(), "%d", &t);
    sscanf(rs.c_str(), "%d", &s);
    sscanf(immediate.c_str(), "%d", &immediate_);
    cout << (bitset<5>)s << (bitset<5>)t << (bitset<16>)immediate_ << endl;
}

void p5(string a){
}

```

```

size_t _1 = a.find('$');//第一个"$"
size_t __1 = a.find(',');
string rs = a.substr(_1 + 1, __1 - _1 - 1);
string immediate = a.substr(_1 + 1);
int s,immediate_;
sscanf(rs.c_str(), "%d", &s);
sscanf(immediate.c_str(), "%d", &immediate_);
cout << (bitset<5>)s << "00000" << (bitset<16>)immediate_ << endl;
}

void p6(string a){
    size_t _1 = a.find('x');//第一个"x"
    string addr = a.substr(_1 + 1);
    int addr_;
    sscanf(addr.c_str(), "%d", &addr_);
    cout << (bitset<26>)addr_ << endl;
}

void p7(string a){
    size_t _1 = a.find('$');//第一个"$"
    string rs = a.substr(_1 + 1);
    int rs_;
    sscanf(rs.c_str(), "%d", &rs_);
    cout << (bitset<5>)rs_ << (bitset<21>)0 << endl;
}

int main()
{
    mymap["add"] = 0;
    mymap["sub"] = 1;
    mymap["or"] = 16;
    mymap["and"] = 17;
    mymap["slt"] = 36;
    //p1

    mymap["addi"] = 2;
    mymap["ori"] = 18;
    mymap["slti"] = 39;
    mymap["beq"] = 52;
    mymap["bne"] = 53;
    //p2

    mymap["sll"] = 24;
    //p3
}

```

```
mymap["sw"] = 48;
mymap["lw"] = 49;
//p4

mymap["bgtz"] = 54;
//p5

mymap["j"] = 56;
mymap["jal"] = 58;
//p6

mymap["jr"] = 57;
//p7

mymap["halt"] = 63;
//p8

string temp;
cout << "please input the assembly language(enter 'q' to quit)" << endl;
while((getline(cin, temp)) && temp != "q"){
    size_t found = temp.find(' ');
    string opcode = temp.substr(0, found);
    cout << (bitset<6>)mymap[opcode];
    string reg = temp.substr(found + 1);
    if(mymap[opcode] == 0 || mymap[opcode] == 1 || mymap[opcode] == 16 ||
mymap[opcode] == 17 || mymap[opcode] == 36)
        p1(reg);
    else if(mymap[opcode] == 2 || mymap[opcode] == 18 || mymap[opcode] == 39 ||
mymap[opcode] == 52 || mymap[opcode] == 53)
        p2(reg);
    else if(mymap[opcode] == 24)
        p3(reg);
    else if(mymap[opcode] == 48 || mymap[opcode] == 49)
        p4(reg);
    else if(mymap[opcode] == 54)
        p5(reg);
    else if(mymap[opcode] == 56 || mymap[opcode] == 58)
        p6(reg);
    else if(mymap[opcode] == 57)
        p7(reg);
    else if(mymap[opcode] == 63)
        cout << (bitset<26>)0 << endl;
    else
```

```
    cout << "Invaild input" << endl;
}
}
```