实验四 多周期 CPU

noname

2020 年 6 月 4 日

1 实验目的

1. 理解计算机硬件的基本组成、结构和工作原理
2. 掌握数字系统的设计和调试方法
3. 熟练掌握数据通路和控制器的设计和描述方法

2 逻辑设计

2.1 多周期 CPU(Multi-cycle CPU)

待设计的多周期 CPU 可以执行如下 6 条指令:

add: $rd \leftarrow rs + rt;$ $op = 000000, funct = 100000$

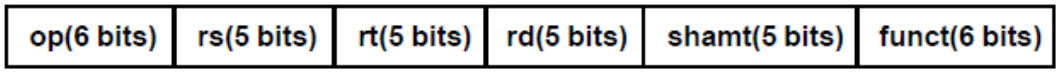

图 1: R-type 指令

addi: $rt \leftarrow rs + imm;$ $op = 001000$
lw: $rt \leftarrow M(rs + addr);$ $op = 100011$
sw: $M(rs + addr) \leftarrow rt;$ $op = 101011$
beq: $if(rs = rt)\ then\ pc \leftarrow pc + 4 + addr << 2$
$else\ pc \leftarrow pc + 4;$ $op = 000100$

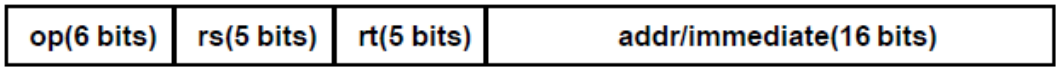

图 2: I-type 指令

j: $pc \leftarrow (pc + 4)[31 : 28]|(add << 2)[27 : 0];$ $op = 000010$

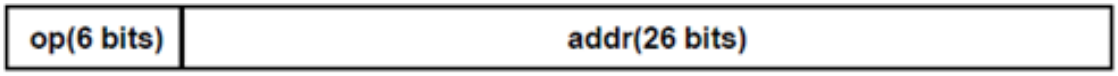

图 3: J-type 指令

满足上述指令的功能，设计多周期 CPU 的数据通路和控制器 (橙色部分) 如图-4 所示，其中控制器的状态图如图-5 所示 (这里加入了 ADDI 指令的状态，对于与代码不符的控制信号值未做修改，看看就好了)。

具体实现时 ALU 和寄存器堆利用实验 1 和实验 2 设计的模块，指令和数据存储共用一个 RAM 存储器，采用 IP 例化实现，容量为 512 x 32 位的分布式存储器

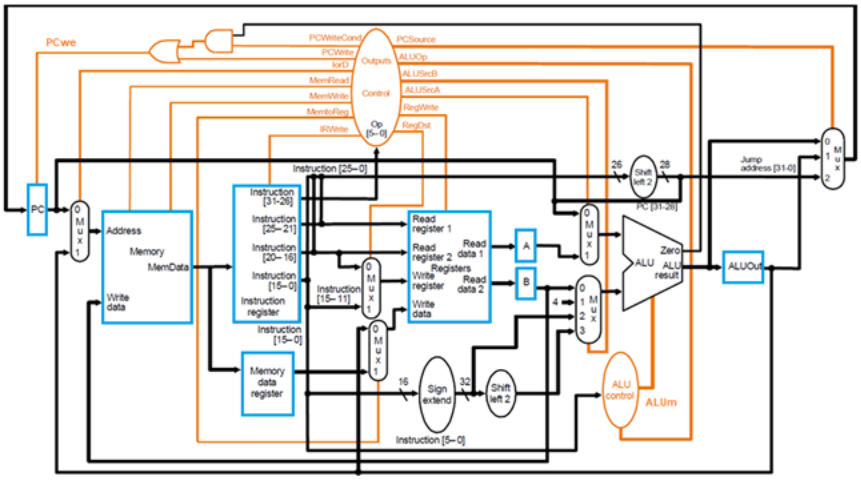

图 4: 数据通路和控制器

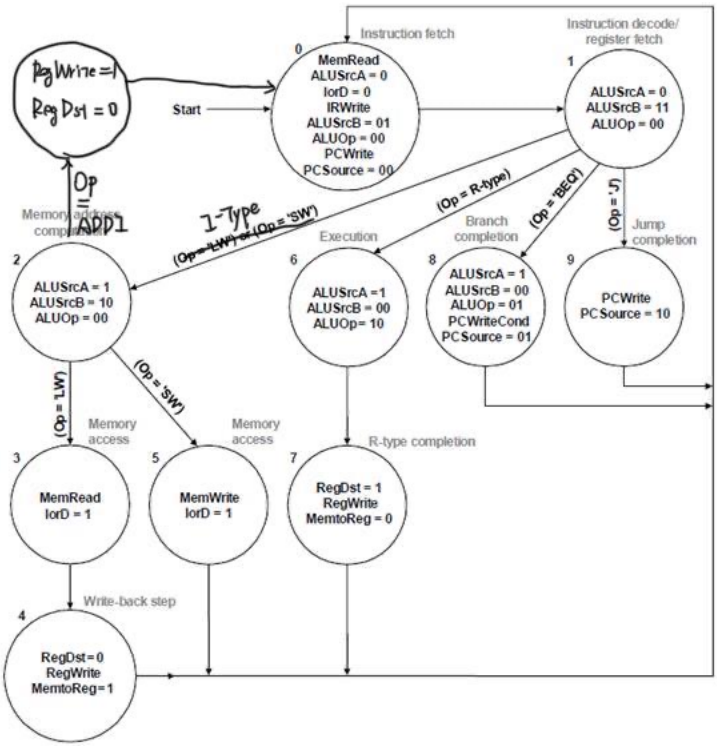

图 5: 状态转移图

多周期 CPU5 个周期内所做操作如下图:

Step	R-Type	lw/sw	beq/bne	j
IF	IR = Mem[PC] PC = PC + 4			
ID	A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUOut = PC + (SE(IR[15-0]) << 2)			
EX	ALUOut = A op B	ALUOut = A +SE(IR[15-0])	If (A==B) then PC = ALUOut	PC = PC[31-28] || (IR[25-0]<<2
MEM	Reg[IR[15-11]] = ALUOut	MDR=Mem[ALUOut] Mem[ALUOut] = B		
WB		Reg[IR[20-16]] = MDR		

图 6: 各周期各个类型指令操作

在介绍各个模块功能，核心代码在后面展示:

1.Mulit-Cycle CPU: 主要负责连接各个模块成为一个完整的 CPU 以及接收 CLK 和 RST 信号
由于没有写 Mux 模块，所以 SingleCPU 内部也实现了多选器的功能用来选择各模块的输入信号

2.InsCut: 指令分割，将 32 位的指令分割成 op,rs,rt,rd,imm,addr,funct 等，便于使用

3.ControlUnit: 控制单元 (译码模块), 输入 opcode 和 CLK 信号, 控制 CPU 状态机跳转和对控制信号赋值

4.PCAdd: 求 NextPC

5.PC: 将求得的 NextPC 根据 PCwe 更新为当前 PC

6.RegFile: Lab2 例化的寄存器堆 (两异步读，一同步写)，用来存取数据，后半个周期写，保证数据的正确性
且增加限制条件使得 0 号寄存器值恒为 0，满足仿真波形的要求 **7.ALU:**Lab1 例化的 ALU，没什么好说的

8.Memory: 分布式 RAM，用来读取指令和读写数据

9.IR: 指令寄存器，根据 IRWrite 来判定指令寄存器是否读入新的指令 (类似的还有 MemDataReg 和 ALUOut)

2.2 核心代码讲解展示

Muilt-Cycle CPU:

端口定义部分不展示，展示各模块连线情况和例化的单元，说明写在注释里

```
1        assign Ex_Imm = immediate[15] ? {16'hffff,immediate} : {16'h0000,
             immediate};
2        //立即数位拓展(有符号数)
3        assign PCwe = PCWrite|(PCWriteCond & zero);                //PCwe控制
             PC更新
4        assign Address = lorD ? ALUOut[10:2] : CurPC[10:2]; //访问Memory选
             择指令或数据
5        assign MemDataReg = lorD ? MemData : MemDataReg;
6        //是否更新MemDataReg的值，与IR单元功能相同
7        assign A = ReadData1,B = ReadData2;
8        //这里应该给个使能信号更新寄存器的，但结果无影响，就没改(
```

```
9         assign ALUA = ALUSrcA ? A : CurPC;
                  //选择ALUA
10        assign ALUB = ALUSrcB[1] ? (ALUSrcB[0] ? (Ex_Imm << 2) : Ex_Imm) :
              (ALUSrcB[0] ? 32'b100 : B);
                                                                  //选择
              ALUB
11
12        assign WriteReg = RegDst ? rd : rt;
                  //寄存器堆写回地址选择
13        assign Reg_WriteData = MemtoReg ? MemDataReg : ALUOut;//寄存器堆写
              回数据选择
14        always@(posedge CLK)
15        begin
16        if(op == 6'b100011 && CUR_STATE == 3'b100)
17        ALUOut = ALUOut;                          //这里的MemDataReg存储数据
              不知为何有问题,增加一个周期延迟使结果正确
18        else
19        ALUOut = ALUresult;                       //正常指令是ALU出结果后下
              个周期写回，则时钟上升沿更新ALUOut
20        end
21        //各个例化的模块调用
22        Memory Memm(.we(MemWrite), .a(Address), .clk(CLK), .d(ReadData2),
              .spo(MemData));
23        RegFile RF(.clk(CLK), .ra0(rs), .ra1(rt), .rd0(ReadData1), .rd1(
              ReadData2),
24        .wa(WriteReg), .we(RegWrite), .wd(Reg_WriteData));
25
26        PC PC(.clk(CLK), .rst(RST), .PCwe(PCwe), .NextPC(NextPC), .CurPC(
              CurPC));
27
28        PCAdd PCAdd(.RST(RST), .addr(addr), .PCSrc(PCSrc), .CurPC(CurPC),
              .ALUresult(ALUresult), .ALUout(ALUOut), .NextPC(NextPC));
29
30        IR IR(.Ins(MemData), .CLK(CLK), .IRWrite(IRWrite), .IRIns(IRIns));
31
32        InsCut IC(.instruction(IRIns), .op(op), .rs(rs), .rt(rt), .rd(rd),
33        .addr(addr), .immediate(immediate));
34
35        Control_Unit CU(.CLK(CLK), .RST(RST), .op(op), .PCSrc(PCSrc), .
              ALUOp(ALUOp), .ALUSrcA(ALUSrcA),
36        .ALUSrcB(ALUSrcB), .RegWrite(RegWrite), .RegDst(RegDst), .
```

```
        PCWriteCond(PCWriteCond),
37      .PCWrite(PCWrite), .lorD(lorD), .MemRead(MemRead), .MemWrite(
            MemWrite),
38      .MemtoReg(MemtoReg), .IRWrite(IRWrite), .CUR_STATE(CUR_STATE));
39
40      ALU ALU(.a(ALUA), .b(ALUB), .ALUOp(ALUOp), .y(ALUresult), .zero(
            zero));
```

Control_Unit:

展示状态机的跳转过程和控制信号的赋值, 说明写在注释里

```
1       module Control_Unit(
2       input CLK,RST,
3       input [5:0] op,      //指令的操作码
4
5       output reg [2:0] CUR_STATE,NEXT_STATE,
6       output reg [1:0] PCSrc,  //控制PC的更新来源, 0:PC+4(IF阶段的PC更
            新, 不进入ALUOut) 1:beq对应的ALUOut 2:Jump指令
7       output reg [2:0] ALUOp,          //ALUOP不解释
8       output reg ALUSrcA,      //选择ALU的A输入 0:选择PC完成PC+4.1:选择
            寄存器堆读出数据
9       output reg [1:0] ALUSrcB,//选择ALU的B输入 0:选择寄存器堆读出数据
            1:4,完成PC+4 2:选择扩展后的立即数 3:Jump指令计算PC时的立即数
10      output reg RegWrite,     //寄存器堆写使能
11      output reg RegDst,       //选择寄存器堆写入地址 0:rt 1:rd
12      output reg PCWriteCond,  //beq指令使用, 如果是beq指令, 和zero与,
            若zero为1则PCwe=1
13      output reg PCWrite,      //选择是否写PC, 优先级更高, 为1PCwe即为1
14      output reg lorD,         //选择读取数据(1)还是指令(0)
15      output reg MemRead,      //数据存储器写使能
16      output reg MemWrite,     //数据存储器读使能 dram还是没用
17      output reg MemtoReg,     //选择写回寄存器堆数据来源 0: ALUOut 1:
            DataMem数据
18      output reg IRWrite       //指令更新, 在IF阶段更新
19      );
20
21      parameter [2:0] INIT = 3'b000,IF = 3'b001,ID = 3'b010,EXE = 3'b011
            ,MEM = 3'b100,WB = 3'b101,HLT = 3'b110;
22      parameter [5:0] ADD = 6'b000000,ADDI = 6'b001000,LW = 6'b100011,
23      SW = 6'b101011,BEQ = 6'b000100,JUMP = 6'b000010;
24      //初始化状态为INIT, 各控制信号为0
25      initial
```

```
26          begin
27          CUR_STATE = INIT;
28          // 2      3       1       2       1       1          1          1
                1       1       1         1         1
29          {PCSrc,ALUOp,ALUSrcA,ALUSrcB,RegWrite,RegDst,PCWriteCond,PCWrite,
                lorD,MemRead,MemWrite,MemtoReg,IRWrite} = 17'b0;
30          end
31
32          //状态机
33          always@(posedge CLK)
34          begin
35          if(RST)                                    //有RST信号归位INIT
36          CUR_STATE <= INIT;
37          else
38          CUR_STATE <= NEXT_STATE; //否则每周期更新至NEXT_STATE
39          end
40
41          //状态转移
42          always@(*)
43          begin
44          case(CUR_STATE)
45          //取指译码阶段各指令相同
46          INIT: NEXT_STATE <= IF;
47          IF:   NEXT_STATE <= ID;
48          ID:   NEXT_STATE <= EXE;
49          //从EXE段开始要考虑不同指令执行跳转情况
50          EXE:
51          begin
52          case(op)
53          //BEQ和JUMP指令3周期结束，跳回IF
54          BEQ: NEXT_STATE <= IF; //beq
55          JUMP: NEXT_STATE <= IF; //Jump
56          default :  NEXT_STATE <= MEM;
57          endcase
58          end
59          MEM:
60          begin
61          case(op)
62          //除LW和3周期结束的BEQ，JUMP指令外其余指令均4周期结束
63          LW: NEXT_STATE <= WB; //lw
64          default:   NEXT_STATE <= IF;
```

```
65          endcase
66          end
67          WB: NEXT_STATE <= IF;
68          default: NEXT_STATE <= INIT;
69          endcase
70          end
71
72          //对各个控制信号赋值
73          always@(*)
74          //首先初始化所有信号为0
75          begin
76          PCWrite = 0; PCSrc = 2'b00; PCWriteCond = 0; ALUSrcA = 0; ALUSrcB
                = 2'b00;
77          ALUOp = 3'b000; MemtoReg = 0; RegWrite = 0; RegDst = 0;
78          lorD = 0; IRWrite = 0; MemRead = 0; MemWrite = 0;
79          case(CUR_STATE)
80          IF://IF阶段完成PC+4和PC = PC+4
81          begin
82          PCWrite = 1;
83          ALUSrcB = 2'b01;
84          IRWrite = 1;
85          MemRead = 1;
86          end
87          ID://ID完成PC + (IMM<<2)以及读寄存器堆内数据(由于AB直接赋值，所以
                没做操作)
88          begin
89          ALUSrcB = 2'b11;
90          end
91          EXE://EXE阶段开始不同
92          begin
93          case(op)
94          ADD://ADD: ALUA和ALUB选择寄存器堆来数据
95          begin
96          ALUSrcA = 1;
97          end
98          //I-type指令 ALUA选择寄存器堆数据，ALUB选择拓展后的立即数
99          ADDI:
100         begin
101         ALUSrcA = 1;      ALUSrcB = 2'b10;
102         end
103         LW:
```

```
104             begin
105             ALUSrcA = 1;      ALUSrcB = 2'b10;
106             end
107             SW:
108             begin
109             ALUSrcA = 1;      ALUSrcB = 2'b10;
110             end
111             BEQ://BEQ: ALUOp = 001,rs，rt做减法。赋PCWriteCond = 1,若zero = 1
112                     //则从PCSrc为01写入BEQ算出的NEXTPC，跳转成功
113             begin
114             PCWrite = 0;      PCSrc = 2'b01;   PCWriteCond = 1;
115             ALUSrcA = 1;      ALUSrcB = 2'b00;      ALUOp = 3'b001;
116             end
117             JUMP://直接跳转，PCSrc也赋为10
118             begin
119             PCWrite = 1;      PCSrc = 2'b10;
120             end
121             endcase
122             end
123             MEM:
124             begin
125             case(op)
126             //MEM阶段大同小异,ADD和ADDI将ALUOut写回rd
127             //SW将ReadData2写回Mem(rs+addr)
128             //LW从Memory内读取数据
129             ADD:
130             begin
131             RegWrite = 1;     RegDst = 1;
132             end
133             ADDI:
134             begin
135             RegWrite = 1;     RegDst = 0;
136             end
137             LW:
138             begin
139             lorD = 1;
140             MemRead = 1;
141             end
142             SW:
143             begin
144             lorD = 1;
```

```
145             MemWrite = 1;
146             end
147             endcase
148             end
149             //只属于LW的WB阶段,将M(rs+addr)写回rt
150             WB:
151             begin
152             MemtoReg = 1;
153             RegWrite = 1;
154             end
155             endcase
156             end
157
158             endmodule
```

其余部分代码在附件内展示

2.3 仿真波形结果分析

具体仿真结果的正确性在录制视频内已经讲过了，可以看出是按照汇编文件逐条执行的，且执行结果均正确

这里只放出仿真波形并进行简要分析

Jump 跳转到 4 条 ADDI 指令

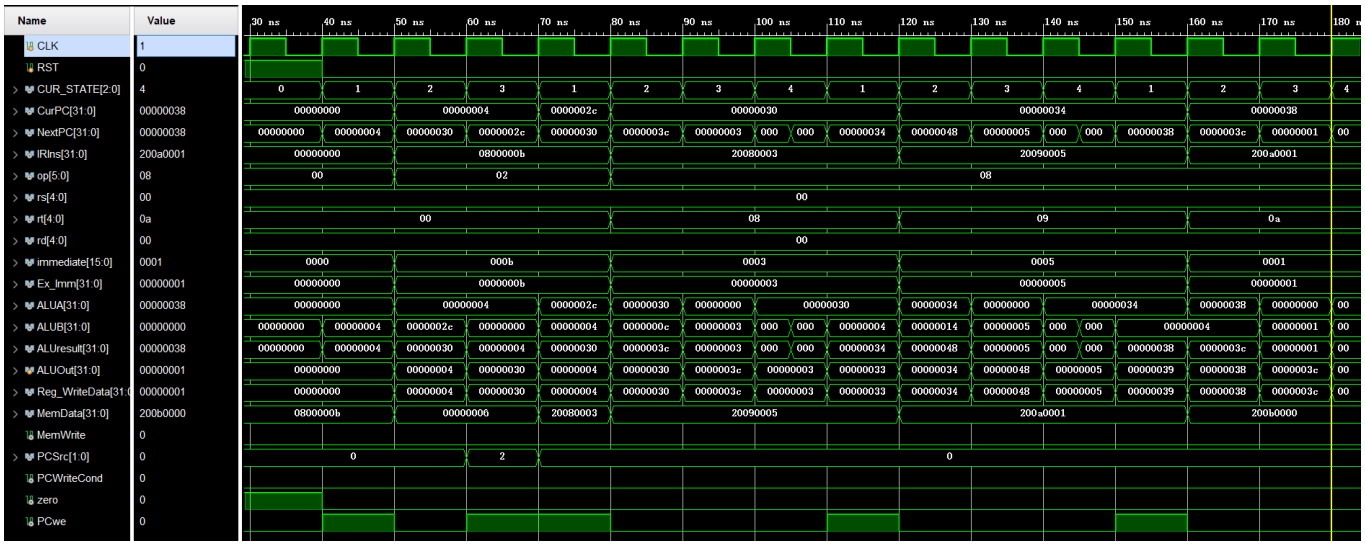

图 7: 波形 1

ADD 和 LW 指令执行

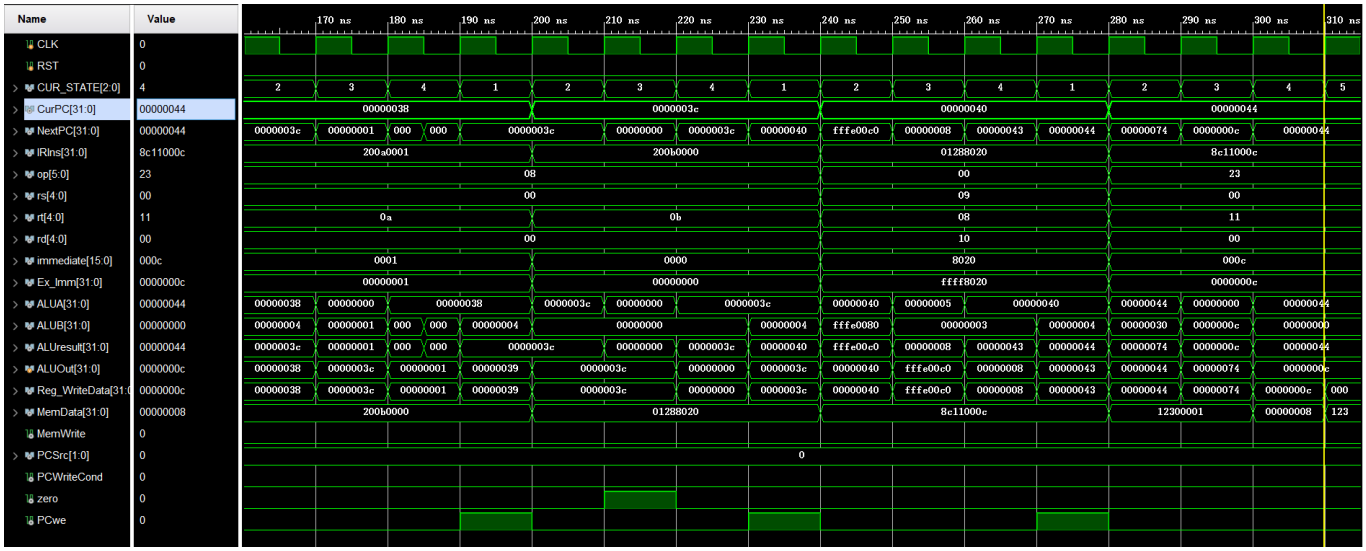

图 8: 波形 2

BEQ 正确跳转后执行两条 LW 指令

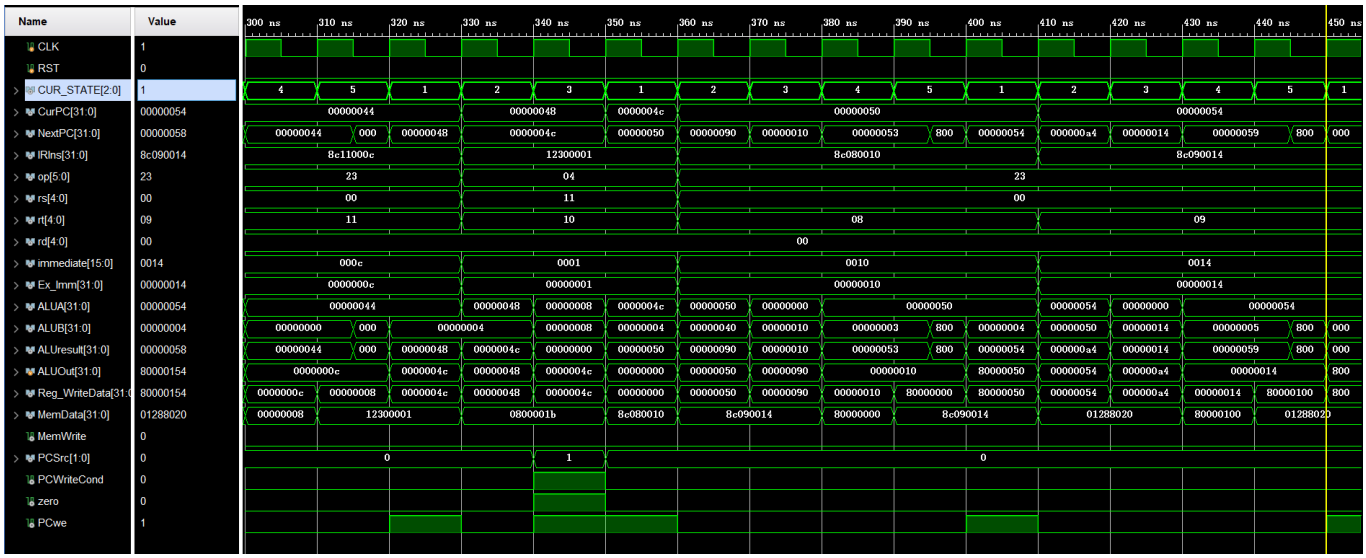

图 9: 波形 3

ADD、LW 紧接一条 BEQ 跳转成功

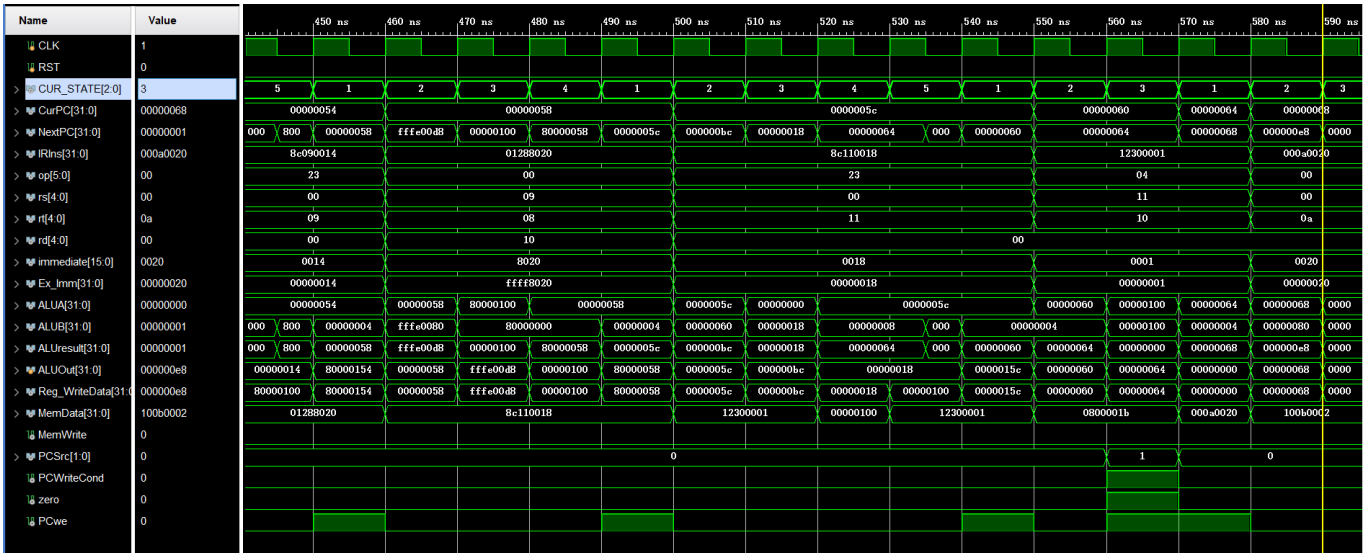

图 10: 波形 4

ADD 接 BEQ 跳转成功，之后 SW 和 JUMP 循环执行

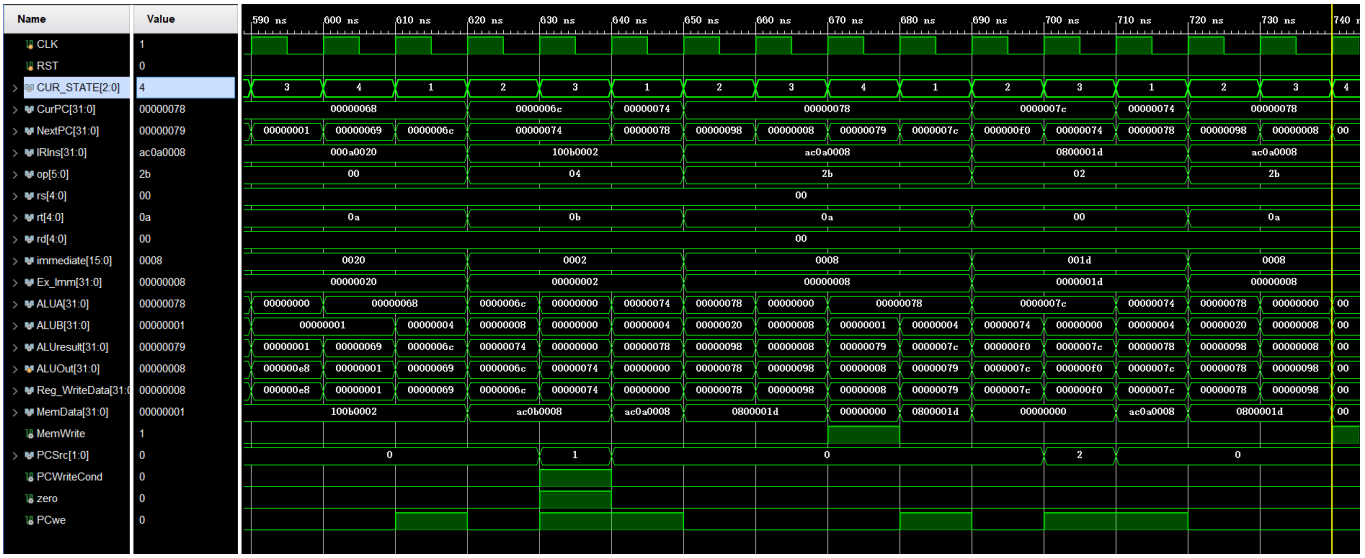

图 11: 波形 5

观察 MEMDATA(存储器内 2 号寄存器，即 0x08) 的值为 0001 发现 SW 指令正确执行

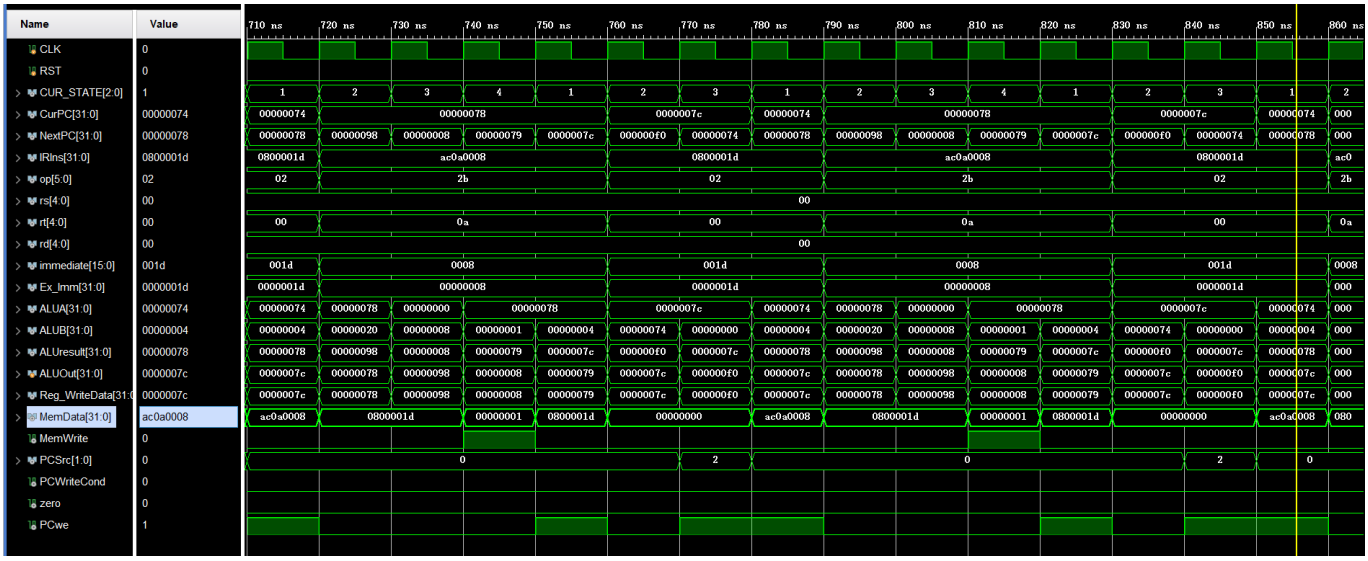

图 12: 波形 6

2.4 调试单元 (Debug Unit)

为了方便下载调试，设计一个调试单元 DBU，该单元可以用于控制 CPU 的运行方式，显示运行过程的中间状态和最终运行结果。DBU 的端口与 CPU 以及 FPGA 开发板外设（拨动/按钮开关、LED 指示灯、7-段数码管）的连接如图 -3 所示。为了 DBU 在不影响 CPU 运行的情况下，随时监视 CPU 运行过程中寄存器堆和数据存储器的内容，可以为寄存器堆和数据存储器增加 1 个用于调试的读端口

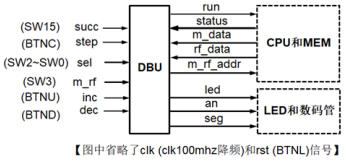

图 13: DBU 逻辑图

控制 CPU 运行方式

succ = 1: 控制 CPU 连续执行指令,run = 1(一直维持)

succ = 0: 控制 CPU 执行一个周期, 每按动 step 一次,run 输出维持一个时钟周期的脉冲

sel = 0: 查看 CPU 运行结果 (存储器或者寄存器堆内容)

m_rf: 1, 查看存储器 (MEM)　0, 查看寄存器堆 (RF)

m_rf_addr:MEM/RF 的调试读口地址 (字地址), 复位时为零

inc/dec:m_rf_addr 加 1 或减 1

rf_data/m_data: 从 RF/MEM 读取的数据字

16 个 LED 指示灯显示 m_rf_addr

8 个数码管显示 rf_data/m_data

sel = 1-7: 查看 CPU 运行状态 (status)

16 个 LED 指示灯 (SW15 SW0) 依次显示控制器的控制信号 (PCSrc, PCwe, lorD, MemWrite, IRWrite, RegDst, MemtoReg, RegWrite, ALUOp, ALUSrcA, ALUSrcB, zero)

8 个数码管显示由 sel 选择的一个 32 位数据

sel = 1：PC, Program_Counter

sel = 2：IR, 指令寄存器数据

sel = 3：MD, MemDataReg 数据

sel = 4：A, 寄存器堆读出寄存器 A

sel = 5：B, 寄存器堆读出寄存器 B

sel = 6：ALUOut, ALU 的运算结果寄存器

首先对单周期 CPU 的模块进行修改以供 DBU 模块使用

1.Mulit_CPU_DBU: 增加了 DBU 内的控制信号输入用来控制 CPU 的运行

2.RegFile: 增加了一个异步读端口，用来读出 m_rf_addr 的内容

3.Memory: 同样的，例化一个双端口 RAM，用来读出 m_rf_addr 的内容

4. 各个凭时钟更新的模块和数据均加入 run 信号，run 为 0 则不更新

5.DBU: 接收输入信号并传递给 CPU, 输出为 LED 和 SW，用来查看运行状态

6.EDG: 增加取边沿电路，使得一次输入仅带来一次改变 (去抖动烧写板子时用，这里不表)

DBU 在思考题会用到，这里不仿真展示 (与单周期基本无异)

3 实验结果

经仿真检验，多周期 CPU 正确实现

4 思考题

4.1 修改数据通路和控制器，扩展对其他 MIPS 指令的支持，并进行功能仿真和下载测试。

若要支持 MIPS 其他指令的执行, 除 R 类型指令中 U 类型的指令 (如 ADDU，SUBU，因为没看懂有无符号数区别), 其余指令只需加入几个状态即可，本人仅实现几条 R-Type 指令并进行仿真，其余指令时间充裕完全可以实现

实现指令有:SUB,AND,OR,XOR,NOR

修改的代码如下：

```
1          //添加指令均为R类型指令，只需EXE阶段考虑funct来控制ALUOp即可
2          EXE:
3          begin
4          case(op)
5          R_TYPE:
6          begin
7          ALUSrcA = 1;
8          case(funct)
9          6'b100000:ALUOp = 3'b000;//add
10         6'b100010:ALUOp = 3'b001;//sub
```

```
11          6'b100100:ALUOp = 3'b010;//and
12          6'b100101:ALUOp = 3'b011;//or
13          6'b100110:ALUOp = 3'b100;//xor
14          6'b100111:ALUOp = 3'b101;//nor
15          endcase
16          end
```

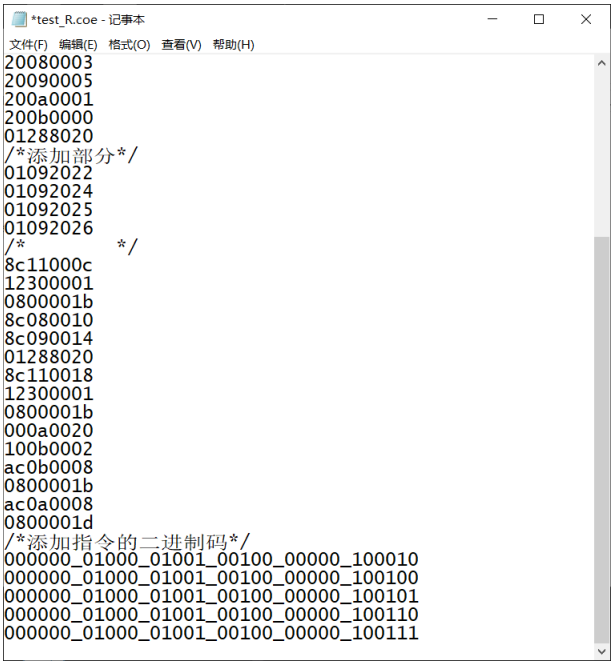

图 14: 修改后 coe 文件

通过 DBU 的 status 查看寄存器堆 4 号寄存器内容可以看出添加指令正确实现

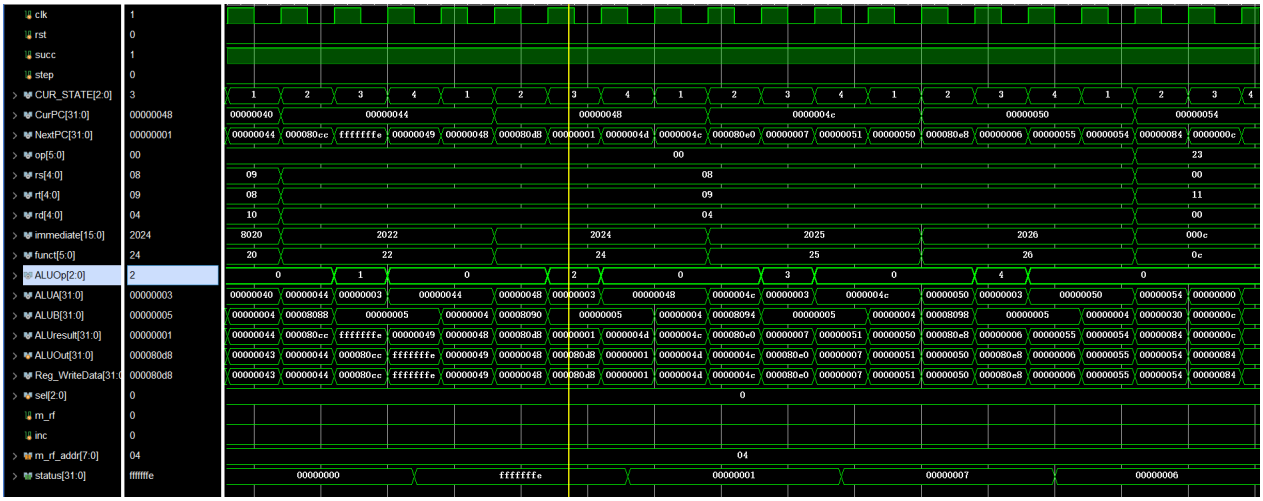

图 15: 新增指令波形

5 意见与建议

本次实验完成难度一般, 单周期 **CPU** 实现时所犯的错误可以很好的避免
但完成了实验还是有一些地方没有完全理解, 可能会成为流水线 **CPU** 实现的障碍

6 附件

本次实验所需提交代码过多, 作为附件上传到 **BB** 系统上 (直接提交多周期 **CPU** 和 **DBU**(思考题版 **CPU**))