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

op(6 bits)	rs(5 bits)	rt(5 bits)	rd(5 bits)	shamt(5 bits)	funct(6 bits)

图 1: R-type 指令

 $\begin{array}{lll} \textbf{addi:} & rt \leftarrow rs + imm; & op = 001000 \\ \textbf{lw:} & rt \leftarrow M(rs + addr); & op = 100011 \\ \textbf{sw:} & M(rs + addr) \leftarrow rt; & op = 101011 \\ \end{array}$

 $\mathbf{beq:} \quad if(rs=rt) \ then \ pc \leftarrow pc + 4 + addr << 2$

 $else\ pc \leftarrow pc + 4; \\ op = 000100$

op(6 bits) rs(5 bits	s) rt(5 bits)	addr/immediate(16 bits)
----------------------	---------------	-------------------------

图 2: I-type 指令

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

op(6 bits) addr(26 bits)

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 512×32 位的分布式存储器

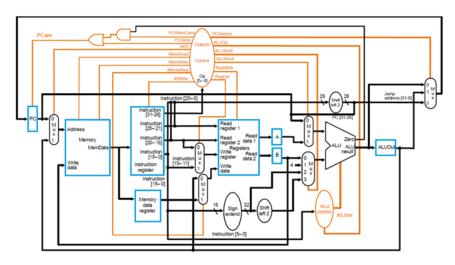


图 4: 数据通路和控制器

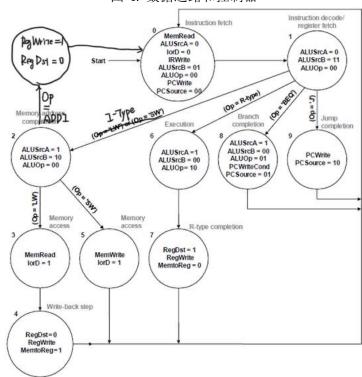


图 5: 状态转移图

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[F PC = PC +							
ID		A = Reg[IR[25 B = Reg[IR[20 ALUOut = PC + (SE(IR	-16]]						
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	1						

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 \};
         //立即数位拓展(有符号数)
2
         assign PCwe = PCWrite | (PCWriteCond & zero);
                                                           //PCwe控制
3
           PC更新
         assign Address = lorD ? ALUOut[10:2] : CurPC[10:2]; //访问Memory选
4
            择指令或数据
         assign MemDataReg = lorD ? MemData : MemDataReg;
5
         //是否更新MemDataReg的值,与IR单元功能相同
6
         assign A = ReadData1,B = ReadData2;
7
         //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(
8
```

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

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

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

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

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

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

```
145
            MemWrite = 1;
            end
146
147
             endcase
            end
148
            // 只属于LW的WB阶段,将M(rs+addr)写回rt
149
            WB:
150
            begin
151
            MemtoReg = 1;
152
            RegWrite = 1;
153
            end
154
            endcase
155
            end
156
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	Jump	跳转到	4 条	ADDI	指今
----------------------	------	-----	-----	------	----

Name	Value	30 ns	40 ns	50 ns	60 ns	70 ns	80 ns	90 ns	100 ns	110 ns	120 ns	130 ns	140 ns	150 ns	160 ns	170 ns	180
₩ CLK	1																j
 RST	0																
₩ CUR_STATE[2:0]		0	1	2	3	1	2	3	4	1	2	3	4	1	2	3	
♥ CurPC[31:0]	00000038	00000	1000	0000	0004	0000002c		0000	00030		X	000	00034		\	00000038	
W NextPC[31:0]	00000038	00000000	00000004	00000030	0000002c	00000030	0000003c	00000003	000 000	00000034	00000048	00000005	000 000	00000038	0000003c	00000001	00
⊌ IRIns[31:0]	200a0001	00000	1000		0800000Ъ		\	2008	80003		X	200	90005		1	200 a0001	
⊌ op[5:0]	08	00		/	02							08					
⊌ rs[4:0]	00								00								
⊌ rt[4:0]	0a			00			<u> </u>		18		X		09		<u> </u>	0a	
⊌ rd[4:0]	00								00								
⊌ immediate[15:0]	0001	0000		000Ъ			<u> </u>	00	003		<u> </u>	0	0001				
⊌ Ex_lmm[31:0]	00000001	00000000		0000000Р		00000003			03			00005		00000001			
■ ALUA[31:0]	00000038	00000	000	0000	0004	0000002c	00000030	00000000	000	00030	00000034	00000000	000	000034	00000038	00000000	01
₩ ALUB[31:0]	00000000	00000000	00000004	0000002c	00000000	00000004	0000000c	00000003	000 000	00000004	00000014	00000005	000 000	000	00004	00000001	0
■ ALUresult[31:0]	00000038	00000000	00000004	00000030	00000004	00000030	0000003c	00000003	000 000	00000034	00000048	00000005	000 000	00000038	0000003c	00000001	00
■ ALUOut[31:0]	00000001	00000	1000	00000004	00000030	00000004	00000030	0000003c	00000003	00000033	00000034	00000048	00000005	00000039	00000038	0000003c	00
■ Reg_WriteData[31:0	00000001	00000	000	00000004	00000030	00000004	00000030	0000003c	00000003	00000033	00000034	00000048	00000005	00000039	00000038	0000003c	00
₩ MemData[31:0]	200b0000			0000	0006	20080003		2009	00005			200	a0001		1	2001-0000	
1⊌ MemWrite																	
⊌ PCSrc[1:0]			0		2	<u> </u>					0						
I PCWriteCond																	
¼ zero																	
l PCwe																	

图 7: 波形 1

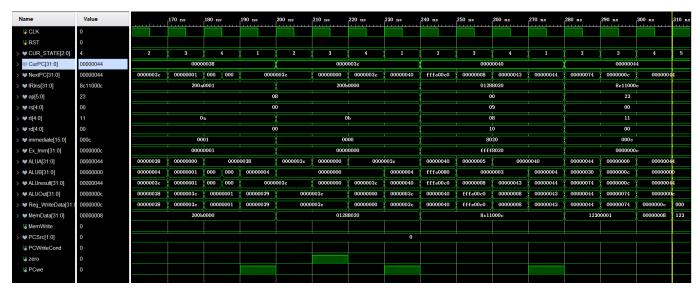


图 8: 波形 2

BEQ 正确跳转后执行两条 LW 指令

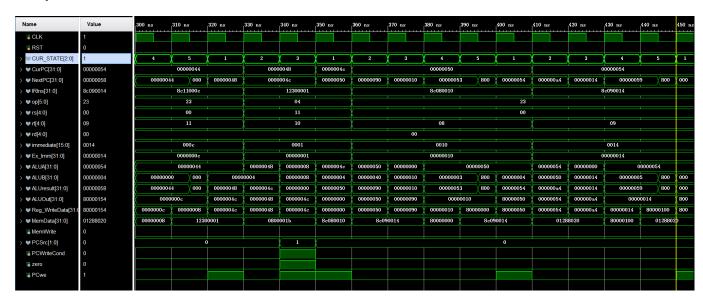


图 9: 波形 3

ADD、LW 紧接一条 BEQ 跳转成功

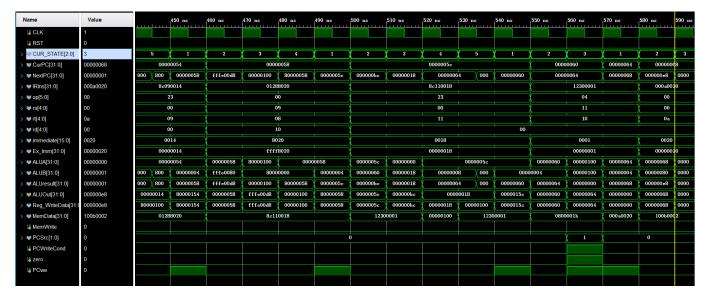


图 10: 波形 4

ADD 接 BEQ 跳转成功, 之后 SW 和 JUMP 循环执行

Name	Value	590 ns	600 ns	610 ns	620 ns	630 ns	640 ns	650 ns	660 ns	670 ns	680 ns	690 ns	700 ns	710 ns	720 ns	730 ns
¼ CLK	1															
₩ RST	0															
© CUR_STATE[2:0]	4	3	4	1	2	3	1	2	3	4	1	2	3	1	2	3
♥ CurPC[31:0]	00000078		8900000		0000	0006c	00000074		0000	00078		000	0007c	00000074	\	0000078
■ NextPC[31:0]	00000079	00000001	00000069	0000006c	0000	00074	00000078	00000098	80000000	00000079	0000007c	01000000	00000074	00000078	00000098	80000000
■ IRIns[31:0]	ac0a0008		000a0020		X	100ь0002			ac0:	a0008		X	0800001d			ac0a0008
♥ op[5:0]	2b		00		X	04				2Ъ		X	02		X	2ь
₩ rs[4:0]	00								00							
₩ rt[4:0]	0a		0a		X	ОЪ) a		X	00		Χ	0a
₩ rd[4:0]	00								00							
w immediate[15:0]	0008		0020		0002				0008			X	001d		X	8000
■ Ex_lmm[31:0]	80000000		00000020		00000002			X		00000008		0000001d			00000008	
₩ ALUA[31:0]	00000078	00000000	000	89000	0000006c	00000000	00000074	00000078	00000000	0000	00078	000	0007c	00000074	00000078	00000000
₩ ALUB[31:0]	00000001	0000	0001	00000004	00000008	00000000	00000004	00000020	00000008	00000001	00000004	00000074	00000000	00000004	00000020	80000000
■ ALUresult[31:0]	00000079	00000001	00000069	0000006c	00000074	00000000	00000078	00000098	00000008	00000079	0000007c	0300000	0000007c	00000078	00000098	80000000
■ ALUOut[31:0]	80000000	000000e8	00000001	00000069	0000006c	00000074	00000000	00000078	00000098	00000008	00000079	0000007c	000000f0	0000007c	00000078	00000098
■ Reg_WriteData[31:0	80000000	000000e8	00000001	00000069	0000006c	00000074	00000000	00000078	00000098	00000008	00000079	0000007c	000000£0	0000007c	00000078	00000098
■ MemData[31:0]	00000001		100ь0002		ac0h	0008	ac0a0008	080	0001d	00000000	0800001d	000	00000	ac0a0008	080	0001d
MemWrite	1															
▶ PCSrc[1:0]	0		()		1	X			ó			2	X	0	
□ PCWriteCond	0															
V zero	0															
¼ PCwe	0															

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

Name	Value	710 ns	720 ns	730 ns	740 ns	750 ns	760 ns	770 ns	780 ns	790 ns	800 ns	810 ns	820 ns	830 ns	840 ns	850 ns		86
¼ CLK	0																	
 RST	0																	
♥ CUR_STATE[2:0]	1	1	2	3	4	1	2	3	1	2	3	4	1	2	3	1		Х
♥ CurPC[31:0]	00000074	00000074	X	0000	00078		0000	1007c	00000074	*	0000	00078		000	0007с	00000	074	χ·
■ NextPC[31:0]	00000078	00000078	00000098	00000008	00000079	0000007c	000000f0	00000074	00000078	00000098	80000000	00000079	0000007c	000000f0	00000074	00000	078	X
■ IRIns[31:0]	0800001d	0800001d	X	ac0:	0008		X	0800001d		*	ac0a	0008			0800001d			Х
₩ op[5:0]	02	02	X		2Ь		X	02		*	2	2Ь			02			Х
₩ rs[4:0]	00								00									
₩ rt[4:0]	00	00	X)a		X	00		X	d)a			00			Х
₩ rd[4:0]	00								00							-		
₩ immediate[15:0]	001d	001d	X	01	008		X	001d		X	00	008			001d	'		X
₩ Ex_lmm[31:0]	0000001d	0000001d	X	0000	80000		/	0000001d		/	0000	80000			0000001d	-		χ
₩ ALUA[31:0]	00000074	00000074	00000078	00000000	0000	00078	0000	1007c	00000074	00000078	00000000	000	00078	000	0007c	00000	074	Х
₩ ALUB[31:0]	00000004	00000004	00000020	00000008	00000001	00000004	00000074	00000000	00000004	00000020	00000008	00000001	00000004	00000074	00000000	00000	004	χ
₩ ALUresult[31:0]	00000078	00000078	00000098	80000000	00000079	0000007c	000000f0	0000007c	00000078	00000098	80000000	00000079	0000007c	01000000	0000007c	00000	078	X
■ ALUOut[31:0]	0000007c	0000007c	00000078	00000098	80000000	00000079	0000007с	01000000	0000007c	00000078	00000098	80000000	00000079	0000007c	000000f0	00000	07c	χ
₩ Reg_WriteData[31:0	0000007c	0000007c	00000078	00000098	80000000	00000079	0000007c	000000£0	0000007c	00000078	00000098	80000000	00000079	0000007c	000000£0	00000	07c	Ÿ
₩ MemData[31:0]	ac0a0008	ac0a0008	0800	0001d	00000001	0800001d	0000	10000	ac0a0008	0800	001d	00000001	08000014	000	00000	ac0a0	800	Ÿ
¼ MemWrite	0																	ı
₩ PCSrc[1:0]	0	X T			0			2	<u> </u>			0			2	X	0	
□ PCWriteCond	0																	ı
¼ zero	0																	
1⊌ PCwe																		г

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

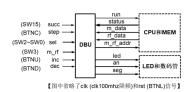


图 13: DBU 逻辑图

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)

3 实验结果 13

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:

```
//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
1
          EXE:
2
          begin
3
          case (op)
4
          R TYPE:
5
          begin
6
7
          ALUSrcA = 1;
          case (funct)
8
          6'b100000 : ALUOp = 3'b000; //add
9
          6'b100010:ALUOp = 3'b001;//sub
10
```

4 思考题 14

```
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
```

```
🌉 *test_R.coe - 记事本
                                                                                                              文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
 20080003
20090005
200a0001
200b0000
01288020
/*添加部分*/
01092022
01092024
01092025
01092026
8c11000c
12300001
0800001b
 8c080010
8c090014
01288020
8c110018
12300001
 0800001b
000a0020
100b0002
ac0b0008
0800001b
0800001b
ac0a0008
0800001d
/*添加指令的二进制码*/
000000_01000_01001_00100_00000_100010
000000_01000_01001_00100_00000_100100
000000_01000_01001_00100_00000_100101
000000_01000_01001_00100_00000_100110
 000000_01000_01001_00100_00000_100111
```

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

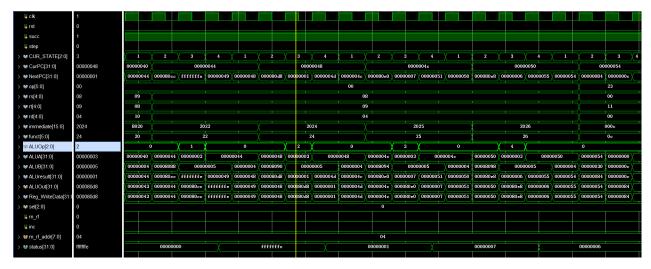


图 15: 新增指令波形

5 意见与建议 15

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))