

## Lab3

逻辑设计

CPU

DBU

核心代码

regfile & ALU

nextpc

control

仿真结果

CPU

DBU

结果分析

实验总结

意见建议

思考题

# Lab3

## 逻辑设计

### CPU

基本上按照所给的数据通路图进行连接。考虑到原设计图中的mux大都是2选1, 这种情况下会用? : 语句来替代, SignExtend和shiftleft也直接通过语句实现。top模块大致结构如下:

```
1 assign Func = Instr[5:0]
2 assign Rd = Instr[15:11];
3 assign Rt = Instr[20:16];
4 assign Rs = Instr[25:21];
5 assign Op = Instr[31:26]; // INSTR decode
6 dist_mem_gen_1 IMem(...); // IMEM
7 reg_file REG(...); // regfile
8 ALU ALU1(...); // ALU
9 control Control(...); // control module
```

```

10 ALUcontrol AC(...); // 因为需要实现的6条指令中仅需要
    Op段就能确定ALUOP,所以这个模块未实现. 如果实现只需要对Func做判
    断就行
11 dist_mem_gen_0 DMem(...); // DMEM
12 nextpc nextpclogic(...); // NEXTPC
13 always @(posedge clk or posedge rst) ...
14 // 这一段用来判断PC是否重置并进行新的PC赋值. 如果重置则暂停执行
    一个周期(以避免不必要的错误)

```

## DBU

DBU添加了一个基于top修改的top1模块,引出了一些top的内部变量,并且给regfile添加了一个读端口,双端口DMEM的dpra和dpo端口也被引出来用于读.具体架构如下:

```

1 top1 TOP1(...); // CPU INSIDE
2 // 直接使用succ,clk以及step取边沿后的信号运算得到时钟信号,功能实
    现和要求一样
3 signaledge(...); // 各种信号取边沿
4 always @(*)...; // 组合逻辑得到out
5 always @(p clk or p rst)...; // 得到下一周期的读地址

```

## 核心代码

### regfile & ALU

regfile和原来的实现基本一样,但在DBU里多了一个读端口

```

1 always@(*)begin
2     if(ra3)
3         rd3 = REG[ra3];
4     else
5         rd3 = 32'h0;
6 end

```

ALU实现也一样, of, cf, zf中CPU暂时只用到zf

### nextpc

得到下一个PC的位置, 先通过判断branch得到pctmp, 再通过判断jump得到pcnext

```
1 module nextpc(  
2     input [31:0] PC,  
3     input jump,  
4     input [25:0] jumpimm,  
5     input branch,  
6     input zero,  
7     input [31:0] sign_ext,  
8     output [31:0] pcnext  
9 );  
10 wire [31:0] pcplus4, pcbeq, pctmp;  
11 wire [31:0] shleft;  
12 wire pcsrc;  
13  
14 assign pcsrc = branch & zero;  
15 assign shleft = sign_ext<<2;  
16 assign pcplus4 = PC+4;  
17  
18 ALU PCOFFSET(.y(pcbeq), .a(shleft), .b(pcplus4),  
19 .m(0));  
20 assign pctmp = pcsrc ? pcbeq : pcplus4;  
21 assign pcnext = jump ? {{pcplus4[31:28]}, {{2'b00,  
    jumpimm}<<2}} : pctmp;  
22 endmodule
```

## control

直接按照对应执行的控制信号值填写

```
1 module control(  
2     input [5:0] opcode,  
3     output reg memtoreg,  
4     output reg memwrite,  
5     output reg branch,  
6     output reg [2:0] alucontrol,  
7     output reg alusrc,  
8     output reg regdst,  
9     output reg regwrite,  
10    output reg jump
```

```
11 );
12 always@(*)begin
13     case(opcode)
14         6'b100011: //lw
15             begin
16                 memtoreg = 1;
17                 memwrite = 0;
18                 branch = 0;
19                 alucontrol = 3'h0; //add
20                 alusrc = 1;
21                 regdst = 0;
22                 regwrite = 1;
23                 jump = 0;
24             end
25         6'b101011: //sw
26             begin
27                 memtoreg = 0;
28                 memwrite = 1;
29                 branch = 0;
30                 alucontrol = 3'h0; //add
31                 alusrc = 1;
32                 regdst = 0;
33                 regwrite = 0;
34                 jump = 0;
35             end
36         6'b001000: //addi
37             begin
38                 memtoreg = 0;
39                 memwrite = 0;
40                 branch = 0;
41                 alucontrol = 3'h0; //add
42                 alusrc = 1;
43                 regdst = 0;
44                 regwrite = 1;
45                 jump = 0;
46             end
47         6'b000000: //add
48             begin
49                 memtoreg = 0;
50                 memwrite = 0;
51                 branch = 0;
```

```

52         alucontrol = 3'h0;
53         alusrc = 0;
54         regdst = 1;
55         regwrite = 1;
56         jump = 0;
57     end
58 6'b000100: //beq
59     begin
60         memtoreg = 0;
61         memwrite = 0;
62         branch = 1;
63         alucontrol = 3'h1;
64         alusrc = 0;
65         regdst = 0;
66         regwrite = 0;
67         jump = 0;
68     end
69 6'b000010: //jump
70     begin
71         memtoreg = 0;
72         memwrite = 0;
73         branch = 0;
74         alucontrol = 3'h6; // undefined
75         alusrc = 0;
76         regdst = 0;
77         regwrite = 0;
78         jump = 1;
79     end
80 default:
81     begin
82         memtoreg = 0;
83         memwrite = 0;
84         branch = 0;
85         alucontrol = 3'h6; // undefined
86         alusrc = 0;
87         regdst = 0;
88         regwrite = 0;
89         jump = 0;
90     end
91 endcase
92 end

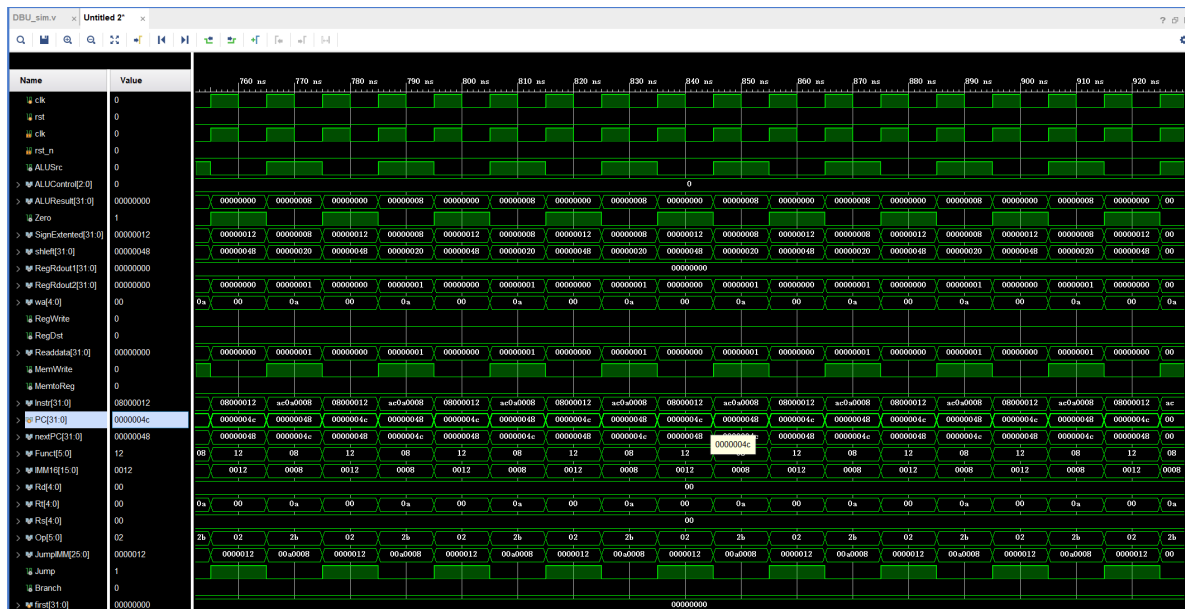
```

93

94 endmodule

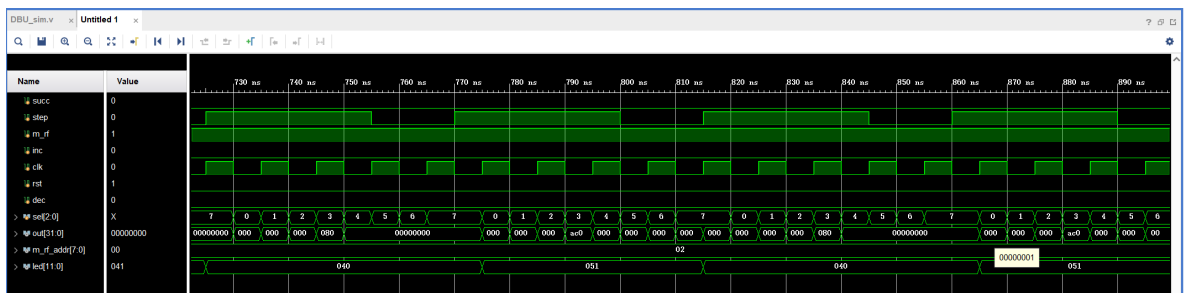
## 仿真结果

### CPU



可见左后PC的值在两个数之间来回跳转, 正好对应的是asm文件中success那两行

### DBU



经过2次 inc 信号后 m\_rf\_addr 的值为2, 可以看到0×08位置上是1

## 结果分析

CPU指令能够正常执行. DBU能控制输出的选项, 控制CPU单步运行. 并且给定机器码后最终可以得到正确的结果.

## 实验总结

- 如果某个模块比较复杂, 可以针对这个模块写一个仿真文件
- 给每条线标好名字(在图像上)可以避免连错
- 信号取边沿时会遇到"两端都碰到clk上升沿"之类的奇怪问题, 注意规避

## 意见建议

---

作为CPU的第一个实验, 难度适中.

## 思考题

---

1. 在 `DMEM` 的读数据端口连一条线到 `ALU` 的输入1(当然需要一个 `mux` 和 `regfile` 的输出1做区别, 同时 `control` 也要多一个控制信号给这么 `mux`)
2. 将 `rs` (做适当的位扩展之后)引到 `DMEM` 的读地址端口(同理, 需要 `mux` 和多一个 `control` 的控制信号)