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:

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

```
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
dist_mem_gen_0 DMem(...); // DMEM
nextpc nextpclogic(...);  // NEXTPC
always @(posedge clk or posedge rst) ...
// 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:

```
top1 TOP1(...);      // CPU INSIDE
// 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
signaledge(...);     // 各种信号取边沿
always @(*)...;      // 组合逻辑得到out
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

```
always@(*)begin
    if(ra3)
        rd3 = REG[ra3];
    else
        rd3 = 32'h0;
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`

```
module nextpc(
    input [31:0] PC,
    input jump,
    input [25:0] jumpimm,
    input branch,
    input zero,
    input [31:0] sign_ext,
    output [31:0] pcnext
    );
    wire [31:0] pcplus4, pcbeq, pctmp;
    wire [31:0] shleft;
    wire pcsrc;

    assign pcsrc = branch & zero;
    assign shleft = sign_ext<<2;
    assign pcplus4 = PC+4;

    ALU PCOFFSET(.y(pcbeq), .a(shleft), .b(pcplus4), .m(0));
    assign pctmp = pcsrc ? pcbeq : pcplus4;
    assign pcnext = jump ? {{pcplus4[31:28]}, {{2'b00, jumpimm}<<2}} : pctmp;
endmodule
```

control

直接按照对应执行的控制信号值填写

```
module control(
input [5:0] opcode,
output reg memtoreg,
output reg memwrite,
output reg branch,
output reg [2:0] alucontrol,
output reg alusrc,
output reg regdst,
output reg regwrite,
output reg jump
```

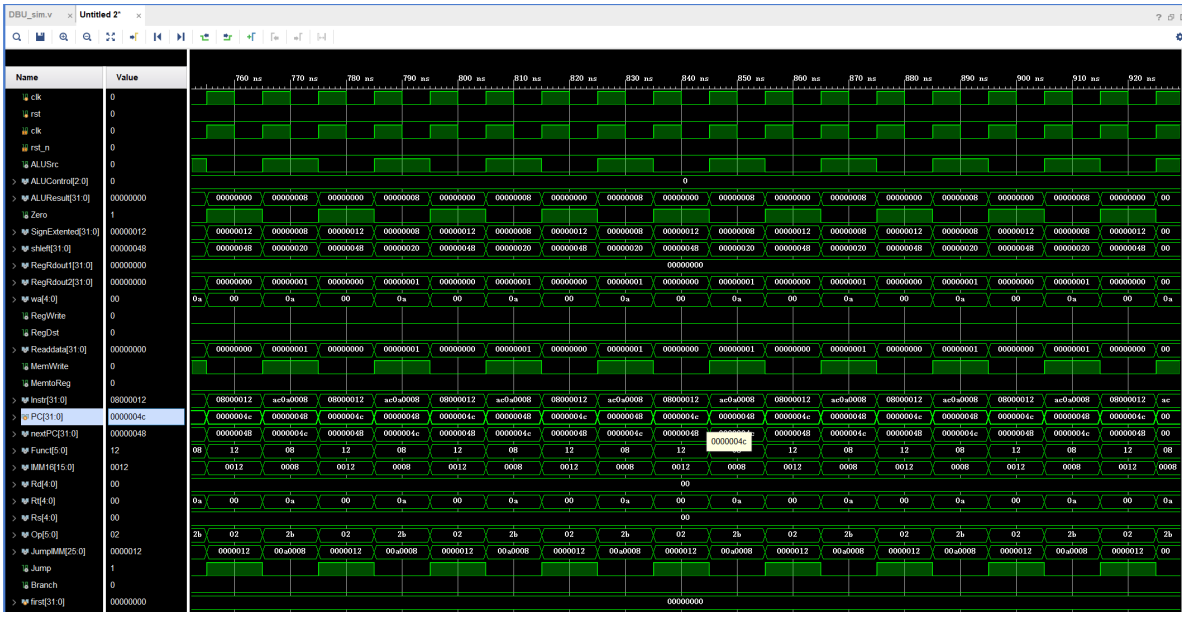
```
);
always@(*)begin
    case(opcode)
        6'b100011: //lw
            begin
                memtoreg = 1;
                memwrite = 0;
                branch = 0;
                alucontrol = 3'h0;  //add
                alusrc = 1;
                regdst = 0;
                regwrite = 1;
                jump = 0;
            end
        6'b101011: //sw
            begin
                memtoreg = 0;
                memwrite = 1;
                branch = 0;
                alucontrol = 3'h0;  //add
                alusrc = 1;
                regdst = 0;
                regwrite = 0;
                jump = 0;
            end
        6'b001000: //addi
            begin
                memtoreg = 0;
                memwrite = 0;
                branch = 0;
                alucontrol = 3'h0;  //add
                alusrc = 1;
                regdst = 0;
                regwrite = 1;
                jump = 0;
            end
        6'b000000: //add
            begin
                memtoreg = 0;
                memwrite = 0;
                branch = 0;
```

```
                alucontrol = 3'h0;
                alusrc = 0;
                regdst = 1;
                regwrite = 1;
                jump = 0;
            end
        6'b000100: //beq
            begin
                memtoreg = 0;
                memwrite = 0;
                branch = 1;
                alucontrol = 3'h1;
                alusrc = 0;
                regdst = 0;
                regwrite = 0;
                jump = 0;
            end
        6'b000010: //jump
            begin
                memtoreg = 0;
                memwrite = 0;
                branch = 0;
                alucontrol = 3'h6;  // undefined
                alusrc = 0;
                regdst = 0;
                regwrite = 0;
                jump = 1;
            end
        default:
            begin
                memtoreg = 0;
                memwrite = 0;
                branch = 0;
                alucontrol = 3'h6;  // undefined
                alusrc = 0;
                regdst = 0;
                regwrite = 0;
                jump = 0;
            end
    endcase
end
```

```
93
94  endmodule
```

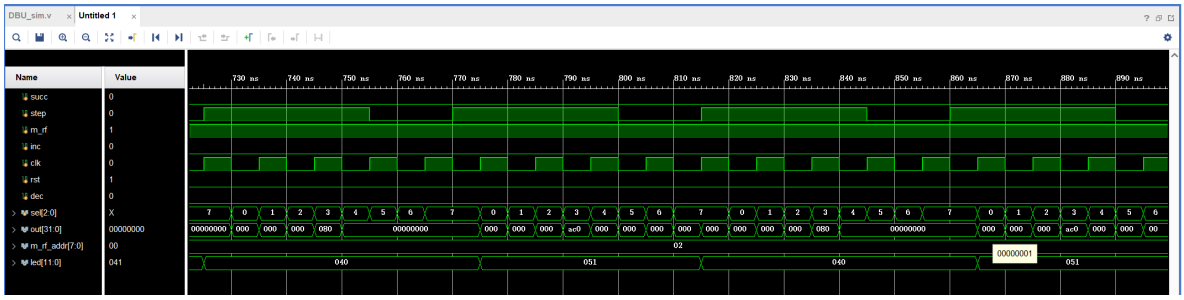
仿真结果

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)