中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009925006　姓名：冯浩瀚　专业：计算机科学与技术

实验序号：02　实验名称：简单功能型处理器设计

注 1：撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 /home/serve-ide/cod-lab/reports 目录下（注意：reports 全部小写）。文件命名规则：prjN.pdf，其中“prj”和后缀名“pdf”为小写，“N”为 1 至 4 的阿拉伯数字。例如：prj1.pdf。PDF 文件大小应控制在 5MB 以内。此外，实验项目 5 包含多个选做内容，每个选做实验应提交各自的实验报告文件，文件命名规则：prj5-projectname.pdf，其中“-”为英文标点符号的短横线。文件命名举例：prj5-dma.pdf。具体要求详见实验项目 5 讲义。
注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 SERVE GitLab 远程仓库 master 分支（具体命令详见实验报告）。
注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

一、 逻辑电路结构与仿真波形的截图及说明（比如关键 RTL 代码段{包含注释}及其对应的逻辑电路结构图{自行画图，推荐用 PPT 画逻辑结构框图，复制到 word 中}、相应信号的仿真波形和信号变化的说明等）

（1） 宏定义

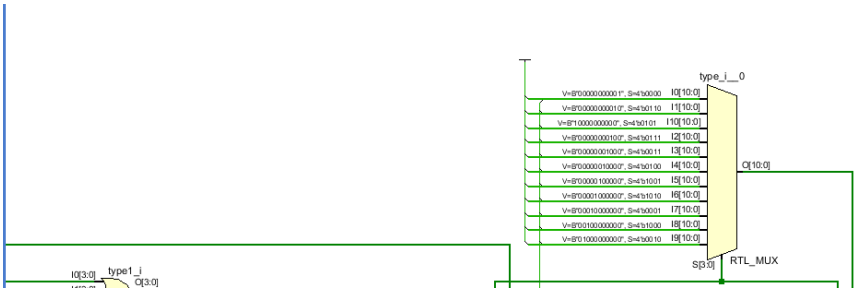
```
`define SPECIAL 6'b000000
`define ALUOP_AND  3'b000
`define ALUOP_OR   3'b001
`define ALUOP_XOR  3'b100
`define ALUOP_NOR  3'b101
`define ALUOP_ADD  3'b010
`define ALUOP_SUB  3'b110
`define ALUOP_SLT  3'b111
`define ALUOP_SLTU 3'b011 // define ALU operation code
// instruction type
`define RAlu 4'b0000
`define RShift 4'b0001
`define RJump 4'b0010
`define RMove 4'b0011
`define REGIMM 4'b0100
`define JType 4'b0101
`define IBranch 4'b0110
`define IAlu 4'b0111
`define lui 4'b1000
`define ILoad 4'b1001
`define IStore 4'b1010 // define instruction type code
```

目的是提高代码的可读性

(2) type 分类

对指令进行分类，方便后续电路的逻辑控制

```
wire [3:0] type = {4{opcode == `SPECIAL && func[5] == 1'b1}} & `RAlu |
                  {4{opcode == `SPECIAL && func[5:3] == 3'b000}} & `RShift |
                  {4{opcode == `SPECIAL && {func[5:3], func[1]} == 4'b0010}} & `RJump |
                  {4{opcode == `SPECIAL && {func[5:3], func[1]} == 4'b0011}} & `RMove |
                  {4{opcode == 6'b000001}} & `REGIMM |
                  {4{opcode[5:1] == 5'b00001}} & `JType |
                  {4{opcode[5:2] == 4'b0001}} & `IBranch |
                  {4{opcode[5:3] == 3'b001 && opcode != 6'b001111}} & `IAlu |
                  {4{opcode == 6'b001111}} & `lui |
                  {4{opcode[5:3] == 3'b100}} & `ILoad |
                  {4{opcode[5:3] == 3'b101}} & `IStore; // decode the opcode
```

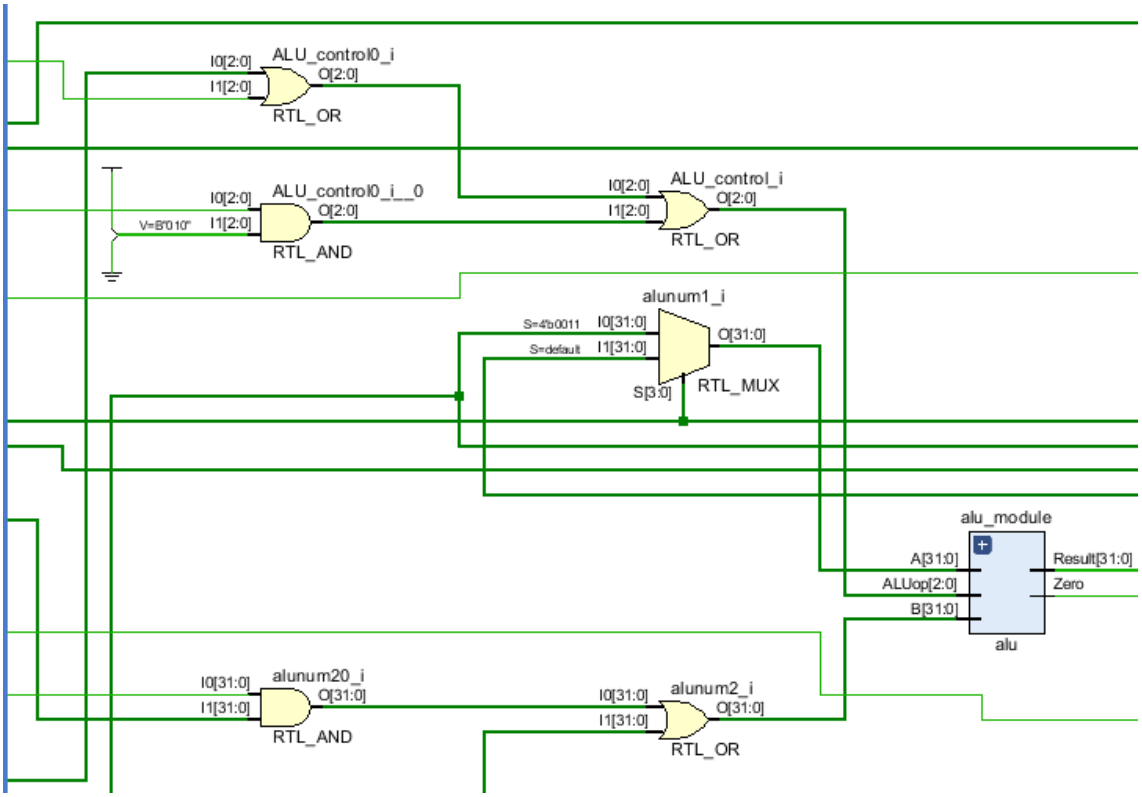


(3) Alu Control

根据不同的指令所需的运算类型以及数据来源，使用选择器对 alu 模块的输

入进行选择，输出运算结果和 Zero 指标

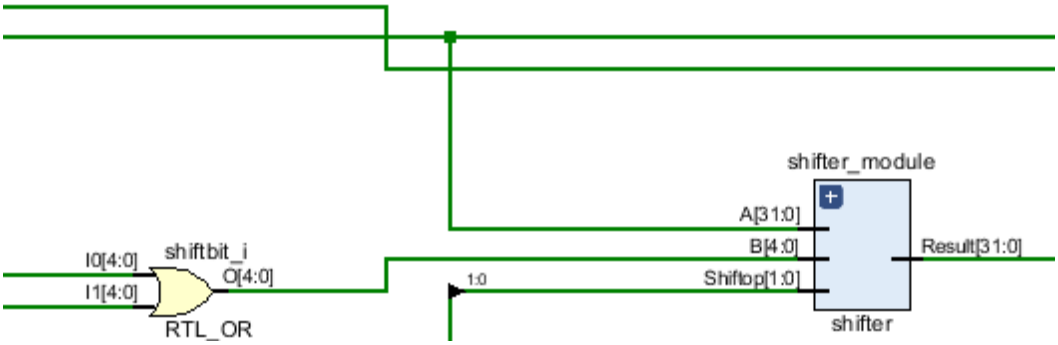
```
// ALU control
wire [31:0] alunum1 = (type == `RMove)? RF_rdata2: RF_rdata1;
wire [31:0] alunum2 = {{32{ALUSrc}} & extend} |
            {{32{type == `RAlu || type == `IBranch}} & RF_rdata2} |
            {{32{type == `RMove || type == `REGIMM}} & 32'b0} ;
wire [2:0] ALU_control = {{3{type == `RAlu && func[3:2] == 2'b00}} & {func[1], 2'b10}} | // addu & subu
                         {{3{type == `IAlu && opcode[2:1] == 2'b00}} & {opcode[1], 2'b10}} | // addiu
                         {{3{type == `RAlu && func[3:2] == 2'b01}} & {func[1], 1'b0, func[0]}} | // and & or & xor & nor
                         {{3{type == `IAlu && opcode[2] == 1'b1}} & {opcode[1], 1'b0, opcode[0]}} | // andi & ori & xori & nori
                         {{3{type == `RAlu && func[3:2] == 2'b10}} & {~func[0], 2'b11}} | // slt & sltu
                         {{3{type == `IAlu && opcode[2:1] == 2'b01}} & {~opcode[0], 2'b11}} | // slti & sltiu
                         {{3{type == `RMove}} & `ALUOP_SUB} | // movz & movn
                         {{3{type == `REGIMM}} & `ALUOP_SLT} | //bltz & bgez
                         {{3{type == `IBranch && opcode[1] == 1'b0}} & `ALUOP_SUB} | // beq & bne
                         {{3{type == `IBranch && opcode[1] == 1'b1}} & `ALUOP_SLT} | // blez & bgtz
                         {{3{type == `ILoad || type == `IStore}} & `ALUOP_ADD}; // lw & sw
wire Zero;
wire [31:0] ALU_result;
alu alu_module(
    .A(alunum1),
    .B(alunum2),
    .ALUop(ALU_control),
    .Result(ALU_result),
    .Overflow(),
    .CarryOut(),
    .Zero(Zero)
); // ALU module
```



(4) Shifter control

根据不同的指令所需的移位类型以及数据来源，使用选择器对 shifter 模块的输入进行选择，输出移位结果。

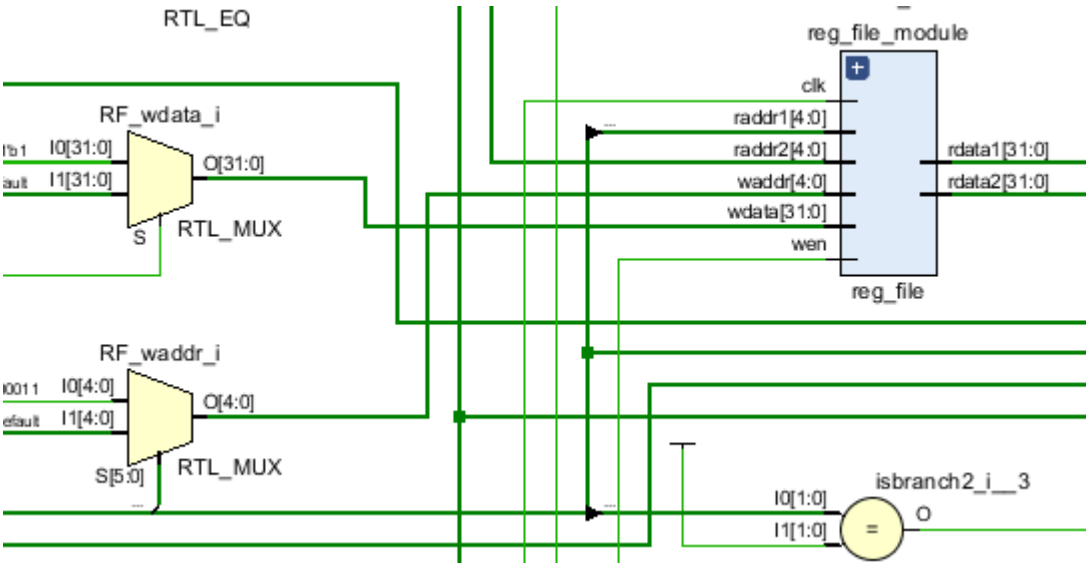
```
// shift control
`define SHIFTOP_SLL  2'b00
`define SHIFTOP_SRL  2'b10
`define SHIFTOP_SRA  2'b11 // define SHIFT operation code

wire [4:0] shiftbit = {{5{type == `RShift && func[5:2] == 4'b0000 }} & shamt} |
                      {{5{type == `RShift && func[5:2] == 4'b0001}} & RF_rdata1[4:0]};
wire [1:0] SHIFT_control = func[1:0];
wire [31:0] SHIFT_result;
shifter shifter_module(RF_rdata2, shiftbit, SHIFT_control, SHIFT_result); // shifter module
```



(5) Reg_File Control

根据不同的指令所需读出或写入的寄存器位置，对 reg_file 模块进行例化，输出读出数据或进行写入操作。

```
// reg_file control unit
assign RF_wen = (type ==`IStore || (type ==`JType && opcode[0] == 0) || type == `IBranch || type == `REGIMM || type == `RJump && func[0] == 0 || type == `RMove && ~func[0] ^ Zero)? 1'b0:1'b1;
assign RF_waddr = (opcode == 6'b000011)? 31:((RegDst)? rd:rt); // jal writes r31
// assign RF_wdata = {{32{type == `RAlu || type == `IAlu}} & ALU_result} |
//      {{32{type == `RShift}} & SHIFT_result} |
//      {{32{type == `RJump && func[0] == 1 || type == `JType && opcode[0] == 1}} & (PC[31:0] + 8) } | // jal & jalr
//      {{32{type == `RMove && func[0] ^ Zero}} & RF_rdata1} |
//      {{32{type == `lui}} & lui_extend}; // write data to register file

assign RF_wdata = (type == `RAlu || type == `IAlu)? ALU_result:
                  ((type == `RShift)? SHIFT_result:
                  ((opcode == 6'b000011 || (type == `RJump && func == 6'b001001))? PC + 8:
                  ((type == `ILoad)? load_result:
                  ((type == `RMove)? RF_rdata1:lui_extend))));

assign RF_raddr1 = rs;
assign RF_raddr2 = (type == `REGIMM)? 5'b0:rt;
reg_file reg_file_module(
    .clk(clk),
    .waddr(RF_waddr),
    .raddr1(RF_raddr1),
    .raddr2(RF_raddr2),
    .wen(RF_wen),
    .wdata(RF_wdata),
    .rdata1(RF_rdata1),
    .rdata2(RF_rdata2)
    ); // register file module
```



(6) Main Control

一些关键的控制信号，控制 alu, PC, Reg_File 等模块

```
// main control unit
wire RegDst = (type == `IAlu || type == `ILoad || type == `lui)? 1'b0:1'b1;
wire branch = (type == `IBranch || type == `REGIMM)? 1'b1:1'b0;
assign MemRead = (type == `ILoad)? 1'b1:1'b0;
wire MemtoReg = (type == `ILoad)? 1'b1:1'b0;
assign MemWrite = (type == `IStore)? 1'b1:1'b0;
wire ALUSrc = (type == `IAlu || type == `ILoad || type == `IStore)? 1'b1:1'b0;
```

这些信号在逻辑电路结构图中分布较散，不一一截图

(7) sign extend

处理 MIPS 指令中存在一些对立即数的位拓展操作

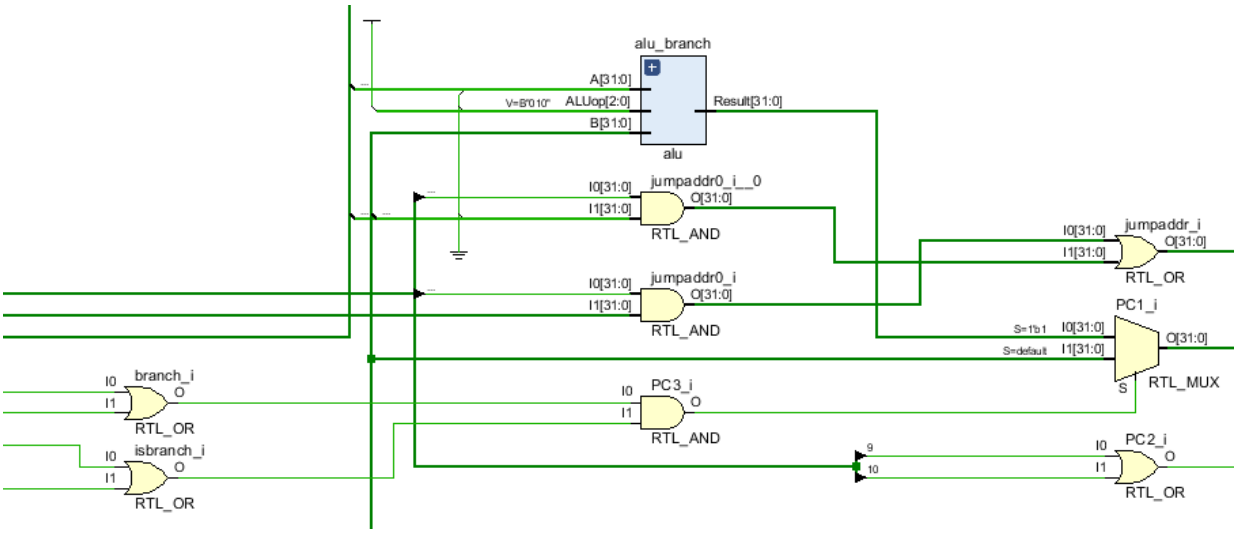
```
// sign extend
wire [31:0] lui_extend = {{16{type == `lui}} & imm, 16'b0};
wire [31:0] sign_extend = {{16{imm[15]}}, imm};
wire [31:0] zero_extend = {16'b0, imm};
wire [31:0] offset_two_extend = {{14{imm[15]}}, imm, 2'b0};
wire [31:0] instr_two_extend = {PC_next[31:28], Instr_index[25:0], 2'b0};
wire [31:0] extend = (type == `IAlu && opcode[2] == 1 || type == `ILoad && opcode[5:1] == 5'b10010)? zero_extend : sign_extend;
```

(8) jump & branch

这两类指令的共同点在于需要更改 PC 的值

```
// jump control
wire [31:0] jumpaddr = {{32{type == `RJump}} & RF_rdata1} |
                       {{32{type == `JType}} & instr_two_extend};

// branch control
wire isbranch = ((type == `REGIMM && REG[0] ^ ~Zero) ||
                 (type == `IBranch && opcode[1] == 0 && opcode[0] ^ Zero) ||
                 (type == `IBranch && opcode[1:0] == 2'b10 && (~Zero || RF_rdata1 == 32'b0)) ||
                 (type == `IBranch && opcode[1:0] == 2'b11 && ~Zero) )? 1'b1:1'b0;
```

```
if (rst) begin
    PC <= 32'b0;
end
else begin
    PC <= (type == `RJump || type == `JType) ? jumpaddr:((branch & isbranch)?branch_PC:PC_next);
end
```

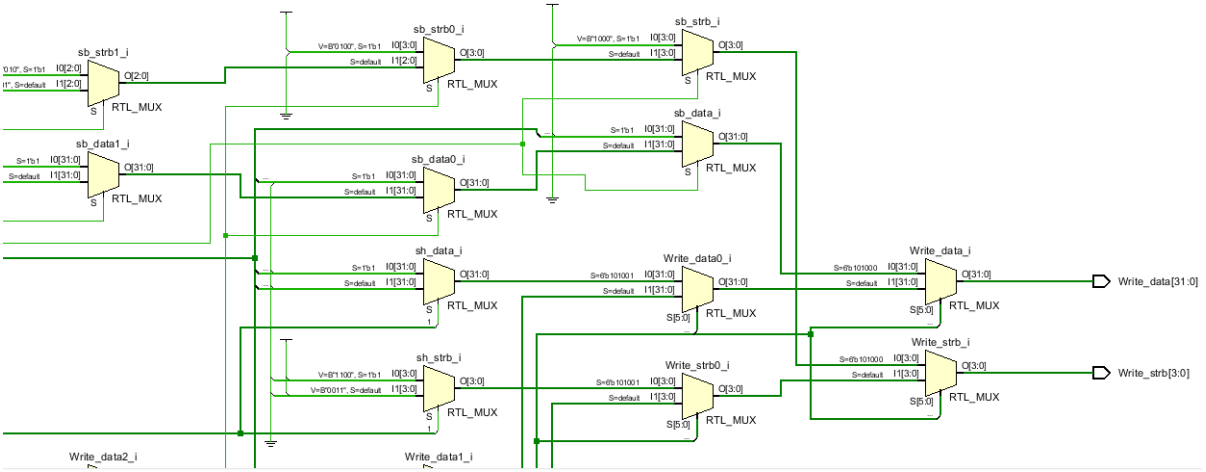


(9) load & store

这两类指令的代码难点在于对数据灵活地进行分割、连接等操作

```
// load control
wire [1:0] n = ALU_result[1:0];

wire [31:0] lb_result = (n[1] & n[0])? {{24{Read_data[31]}},Read_data[31:24]}:
                        ((n[1] & ~n[0])? {{24{Read_data[23]}},Read_data[23:16]}:
                        ((~n[1] & n[0])? {{24{Read_data[15]}},Read_data[15:8]}:{{24{Read_data[7]}},Read_data[7:0]}));
wire [31:0] lbu_result = {{24{1'b0}},lb_result[7:0]};

wire [31:0] lh_result = (~n[1])? {{16{Read_data[15]}},Read_data[15:0]}:{{16{Read_data[31]}},Read_data[31:16]};
wire [31:0] lhu_result = {{16{1'b0}},lh_result[15:0]};

wire [31:0] lw_result = Read_data[31:0];
wire [31:0] lwl_result = (n[1] & n[0])? Read_data[31:0]:
                         ((n[1] & ~n[0])? {Read_data[23:0],RF_rdata2[7:0]}:
                         ((~n[1] & n[0])? {Read_data[15:0],RF_rdata2[15:0]}:{Read_data[7:0],RF_rdata2[23:0]}));

wire [31:0] lwr_result = (~n[1] & ~n[0])? Read_data[31:0]:
                         ((~n[1] & n[0])? {RF_rdata2[31:24],Read_data[31:8]}:
                         ((n[1] & ~n[0])? {RF_rdata2[31:16],Read_data[31:16]}:{RF_rdata2[31:8],Read_data[31:24]}));


wire [31:0] load_result = (opcode == 6'b100000)? lb_result:
                          ((opcode == 6'b100001)? lh_result:
                          ((opcode == 6'b100011)? lw_result:
                          ((opcode == 6'b100100)? lbu_result:
                          ((opcode == 6'b100101)? lhu_result:
                          ((opcode == 6'b100010)? lwl_result: lwr_result)))));
```

```
//store control
assign Address = {ALU_result[31:2],2'b00};//aligned address

wire [3:0] sb_strb = (n[1] & n[0])? 4'b1000:
                     ((n[1] & ~n[0])? 4'b0100:
                     ((~n[1] & n[0])? 4'b0010: 4'b0001));
wire [3:0] sh_strb = (n[1])? 4'b1100: 4'b0011;
wire [3:0] sw_strb = 4'b1111;

wire [3:0] swl_strb = {ALU_result[1] & ALU_result[0], ALU_result[1], ALU_result[1] | ALU_result[0], 1'b1};
wire [3:0] swr_strb = {1'b1, ~(ALU_result[1] & ALU_result[0]), ~ALU_result[1], ~ALU_result[1] & ~ALU_result[0]};

assign Write_strb = (opcode == 6'b101000)? sb_strb:
                    ((opcode == 6'b101001)? sh_strb:
                    ((opcode == 6'b101011)? sw_strb:
                    ((opcode == 6'b101010)? swl_strb:
                    ((opcode == 6'b101110)? swr_strb:swr_strb))));

wire [31:0] sb_data = (n[1] & n[0])? {RF_rdata2[7:0],{24{1'b0}}}:
                      ((n[1] & ~n[0])? {{8{1'b0}},RF_rdata2[7:0],{16{1'b0}}}:
                      ((~n[1] & n[0])? {{16{1'b0}},RF_rdata2[7:0],{8{1'b0}}}: {{24{1'b0}},RF_rdata2[7:0]}));

wire [31:0] sh_data = (n[1])? {RF_rdata2[15:0],{16{1'b0}}}: {{16{1'b0}},RF_rdata2[15:0]};

wire [31:0] sw_data = RF_rdata2[31:0];

wire [31:0] swl_data = (swl_strb == 4'b0001)? {{24{1'b0}},RF_rdata2[31:24]}:
                       ((swl_strb == 4'b0011)? {{16{1'b0}},RF_rdata2[31:16]}:
                       ((swl_strb == 4'b0111)? {{8{1'b0}},RF_rdata2[31:8]}:RF_rdata2[31:0]));

wire [31:0] swr_data = (swr_strb == 4'b1111)? RF_rdata2[31:0]:
                       ((swr_strb == 4'b1110)? {RF_rdata2[23:0],{8{1'b0}}}:
                       ((swr_strb == 4'b1100)? {RF_rdata2[15:0],{16{1'b0}}}:{RF_rdata2[7:0],{24{1'b0}}}));

assign Write_data = (opcode == 6'b101000)? sb_data:
                    ((opcode == 6'b101001)? sh_data:
                    ((opcode == 6'b101011)? sw_data:
                    ((opcode == 6'b101010)? swl_data:
                    ((opcode == 6'b101110)? swr_data: swr_data))));
```



二、 实验过程中遇到的问题、对问题的思考过程及解决方法（比如 RTL 代码中出现的逻辑 bug，逻辑仿真和 FPGA 调试过程中的难点等）

（1） 控制信号的选择器的条件不完善，漏掉了某些指令的情况。根据发生错误时的指令类型以及其他信号的波形推断出错的位置。这类错误发生的最多，因为一次性将 45 条指令的操作类型、操作条件、有关寄存器的位置等代码编写完整很困难，只能一步一步修缮完整，所以这类 bug 是本次实验的主旋律。

（2） 编写 shifter 模块时算术右移不能使用“>>>”，应该采用一种将原操作数进行逻辑右移之后再与 11…100…0（1 的个数与移位位数相同）按位或的方法，手动移位。

三、 在课后，你花费了大约 20 小时完成此次实验

四、 对于此次实验的心得、感受和建议（比如实验是否过于简单或复杂，是否缺少了某些你认为重要的信息或参考资料，对实验项目的建议，对提供帮助的同学的感谢，以及其他想与任课老师交流的内容等）

（1） 本次实验较困难，首先需要对单周期处理器的原理、流程图完全理解，之后对代码框架的搭建也考验了我的大局观。我认为这个项目应该从整体入

手，将 alu，reg_file，shifter，PC 等模块写出来，然后从细节入手，参考指令集处理这些模块的输入与 cpu 的输入之间的关联。

(2) 本次实验的参考资料较为齐全，但是在一些细微之处存在一些误导性的话语，不过因此导致的程序 bug 很好找出。例如，“P03-简单功能型处理器设计-v6.pdf”中的第 25 页指出，“所有 R-Type 指令都会产生寄存器写（wen=1)”。然而笔者发现 bhv_sim 中要求 jr 和 movz 与 movn 的 wen 并不总是 1。

(3) 总之，感谢老师们的辛勤付出。