

# 中国科学院大学计算机组成原理实验课

## 实 验 报 告

学号：2020K8009907032 姓名：唐嘉良 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 格式保存在~/COD-Lab/reports 目录下。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 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，可包含但不限于如下内容。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、相应信号的仿真波形和信号变化的说明等）

为完成单周期 CPU 实验，我首先对 reg\_file.v 以及 alu.v 这两份寄存器堆以及运算器代码进行了修改以便复用，然后利用它们和新写的 shifter.v 移位器构造了单周期处理器的代码。

#### 1. reg\_file 相应代码修改

```

home > ucas > COD-Lab > fpga > design > ucas-cod > hardware > sources > reg_file > reg_file.v
1  `timescale 10 ns / 1 ns
2
3  `define DATA_WIDTH 32
4  `define ADDR_WIDTH 5
5  `define REG_NUM 32
6
7  module reg_file(
8      input                clk,
9      input                rst,
10     input  [`ADDR_WIDTH - 1:0] waddr,
11     input  [`ADDR_WIDTH - 1:0] raddr1,
12     input  [`ADDR_WIDTH - 1:0] raddr2,
13     input                wen,
14     input  [`DATA_WIDTH - 1:0] wdata,
15     output [`DATA_WIDTH - 1:0] rdata1,
16     output [`DATA_WIDTH - 1:0] rdata2
17 );
18
19 reg [`DATA_WIDTH-1:0] REG_FILE [`REG_NUM-1:0];
20
21 always @(posedge clk) begin
22     if(wen && (waddr!=5'b0)) begin
23         REG_FILE[waddr] <= wdata;
24     end
25 end
26
27 assign rdata1 = (raddr1==5'b0)?32'b0:REG_FILE[raddr1];
28 assign rdata2 = (raddr2==5'b0)?32'b0:REG_FILE[raddr2];
29
30 endmodule

```

本以为 reg\_file.v 不需要进行修改，谁知在后来云平台的语法检查中出了错误，发现是 reg\_file 模块没有加复位信号 rst 导致的，这是由于在实验一中我的 reg\_file.v 代码并没有设置 rst 接口。因此 reg\_file 的修改仅是在于 rst 端口信号的加入。

后来发现了，是因为我在调用的时候自作主张地加上了 rst 信号端口。其实不需要修改寄存器代码。

## 2. alu 相应代码修改

对于 alu 而言需要进行修改的内容是增加三个 ALUop 信号，也就是按位或非 nor、无符号数比较 sltu 以及按位异或 xor.修改后的关键代码段如下：

```
//translate the ALUop
wire add;
wire sub;
wire orr;
wire andd;
wire slt;
//3 new operations
wire xorr;
wire norr;
wire sltu;
```

```
//the details of combination circuit
assign andd = (!ALUop[2]) & (!ALUop[1]) & (!ALUop[0]); //valid when ALUop=000
assign orr = (!ALUop[2]) & (!ALUop[1]) & (ALUop[0]); //valid when ALUop=001
assign add = (!ALUop[2]) & (ALUop[1]) & (!ALUop[0]); //valid when ALUop=010
assign sub = (ALUop[2]) & (ALUop[1]) & (!ALUop[0]); //valid when ALUop=110
assign slt = (ALUop[2]) & (ALUop[1]) & (ALUop[0]); //valid when ALUop=111
assign xorr = (ALUop[2]) & (!ALUop[1]) & (!ALUop[0]); //valid when ALUop=100
assign norr = (ALUop[2]) & (!ALUop[1]) & (ALUop[0]); //valid when ALUop=101
assign sltu = (!ALUop[2]) & (ALUop[1]) & (ALUop[0]); //valid when ALUop=011
```

**译码信号：**新增三个译码信号 xorr, norr 和 sltu。如此命名是因为 xor 以及 nor 是 verilog 的保留字。

```
/******The onlyyyy adder
assign {ca_out,add_sub_res} = {1'b0,A} + {1'b0,((ALUop[2] | sltu)?~B:B)} + (ALUop[2] | sltu);
```

**加法器信号：**将减法的选择信号从单独的 ALUop[2] 改为 ALUop[2] | sltu，以便在 sltu 的译码信号下加法器执行减法功能。

```
//give the output result
/*to judge the slt, we assume that slt = 1 iff the sub mode of adder gives the result of sign 1 and a sub overflow didn't happen,
or the sub mode of adder gives the result of sign 0 and a sub overflow happened*/
assign Result = ({32{andd}} & {A & B}) |
                ({32{orr}} & {A | B}) |
                ({32{add | sub}} & add_sub_res) |
                ({32{slt}} & {add_sub_res[DATA_WIDTH-1] ^ sub_flow}) |
                ({32{xorr}} & {A ^ B}) |
                ({32{norr}} & {~(A | B)}) |
                ({32{sltu}} & {(A[DATA_WIDTH-1]^B[DATA_WIDTH-1])?~(add_sub_res[DATA_WIDTH-1]^sub_flow):(add_sub_res[DATA_WIDTH-1]^sub_flow)});
```

**Result 信号：**将 Result 这一 output 信号改为上图所示。增加了 xor、nor 以及 sltu 的结果。对于 sltu，考虑 A 与 B 符号位，如果符号相同，则判断规则同 slt，若符号不同，有两种可能：A 小 B 大，结果大且溢出或者小且不溢出；A 大 B 小，结果大且不溢出或者小且溢出。发现这两种情况等价于

add\_sub\_res[31]和 sub\_flow 同号，这是与 slt 相反的。综上，根据同类合并的原则，我用三目运算符给 sltu 下的 Result 进行选择赋值。

### 3. shifter 代码实现

```
shifter.v x
home > ucas > COD-Lab > fpga > design > ucas-cod > hardware > sources > shifter > shifter.v
1 |`timescale 10 ns / 1 ns
2
3 |`define DATA_WIDTH 32
4
5 |module shifter (
6 |    input  [`DATA_WIDTH - 1:0] A,
7 |    input  [          4:0] B,
8 |    input  [          1:0] Shiftop,
9 |    output [`DATA_WIDTH - 1:0] Result
10 |);
11 |//translate the Shiftop
12 |    wire left = (Shiftop == 2'b00);
13 |    wire right_arith = (Shiftop == 2'b11);
14 |    wire right_logic = (Shiftop == 2'b10);
15 |//define the shifter logic
16 |    wire [63:0] signed_extend = {{32{A[`DATA_WIDTH-1]}},A} >> B;
17
18 |    wire [31:0] left_res = A << B;
19 |    wire [31:0] right_logic_res = A >> B;
20 |    wire [31:0] right_arith_res = signed_extend[31:0];
21 |//output the Result
22 |    assign Result = ({32{left}} & left_res) |
23 |                    ({32{right_arith}} & right_arith_res) |
24 |                    ({32{right_logic}} & right_logic_res);
25
26 |endmodule
27
```

与 ALU 类似地，我们利用 left, right\_arith 和 right\_logic 产生译码信号，然后利用 verilog 的移位运算符轻而易举地实现了左移和逻辑右移。关键在于算术右移，这里我采取了一点代码技巧：先将被移位数进行符号扩展，再对符号扩展后的操作数进行逻辑右移，取低 32 位作为算术右移的结果。

Result 的确定方法同 ALU，是简单的，在此不再赘述。

#### 4. simple\_cpu

```
1  `timescale 10ns / 1ns
2  //define the ALUop
3  `define AND 3'b000
4  `define OR 3'b001
5  `define ADD 3'b010
6  `define SUB 3'b110
7  `define STL 3'b111
8  `define XOR 3'b100
9  `define NOR 3'b101
10 `define STLU 3'b011
11 //define the Shifterop
12 `define LEFT 2'b00
13 `define RIGHT_ARITH 2'b11
14 `define RIGHT_LOGIC 2'b10
15 //encode the instruction type
16 `define CAL 4'b0000
17 `define SHIFT 4'b0001
18 `define SHIFTV 4'b0010
19 `define JUMP 4'b0011
20 `define JUMPR 4'b0100
21 `define MOV 4'b0101
22 `define BRANCH 4'b0110
23 `define LOAD 4'b0111
24 `define STORE 4'b1000
25 `define CALI 4'b1001
26 `define LUI 4'b1010
27 `define NO 4'b1011
28
```

**宏定义：**考虑到需要对 ALU 与 shifter 这两个模块进行复用，我们的宏定义中包括了 ALUop、shifterop 的译码；另一方面，考虑将指令译码，我们又定义出 12 种指令译码，以此定义出 Ins\_Type 的宏。



```

47 wire RF_wen;
48 wire [4:0] RF_waddr;
49 wire [31:0] RF_wdata;
50 //The instruction components
51 wire [5:0] opcode = Instruction[31:26];
52 wire [4:0] rs = Instruction[25:21];
53 wire [4:0] rt = Instruction[20:16];
54 wire [4:0] rd = Instruction[15:11];
55 wire [4:0] sa = Instruction[10:6];
56 wire [5:0] func = Instruction[5:0];
57 wire [15:0] imm = Instruction[15:0];
58 wire [31:0] extend_imm = {{16{Instruction[15]}}, Instruction[15:0]}; //sign-extend
59 wire [31:0] two_extend_imm = {{14{Instruction[15]}}, Instruction[15:0], 2'b0}; //sign-extend and shift Left 2 bits
60 //Encode the type of instruction
61 wire [3:0] ins = (opcode[5:3] == 3'b001 && opcode != 6'b001111)? `CALI:
62 ((opcode == 6'b000000 && func[5] == 1)? `CAL:
63 ((opcode == 6'b000000 && func[5:2] == 4'b0000)? `SHIFT:
64 ((opcode == 6'b000000 && func[5:2] == 4'b0001)? `SHIFTV:
65 ((opcode[5:2] == 4'b0001 || (opcode == 6'b000001 && rt == 5'b0000) || (opcode == 6'b000001 && rt == 5'b000000)? `BRANCH:
66 ((opcode[5:1] == 5'b00001)? `JUMP:
67 ((opcode == 6'b000000 && func[5:1] == 5'b00100)? `JUMPR:
68 ((opcode[5:3] == 3'b100)? `LOAD:
69 ((opcode[5:3] == 3'b101)? `STORE:
70 ((opcode == 6'b000000 && func[5:1] == 5'b00101)? `MOV:
71 ((opcode == 6'b001111)? `LUI: `NO))))));

```

**指令译码：**此处我们将 Instruction 信号进行译码，分析指令的组成部分，并且将指令分类。对应相同控制信号的同类指令进行合并，根据事先写出的指令译码表进行译码。这部分工作有点困难且复杂，我一度写错了译码表。这里我采用先将指令分类的方法来进行之后的操作，是因为这样可以方便后续 control signal 和其它端口信号的判断方式统一，便于规范化、统一化操作，也增强了代码的可读性。

```

72 //The control signal
73 wire RegDst = ~(opcode[5:3] == 3'b001 || opcode[5:3] == 3'b100); //not i-type or Load
74 wire Jump = (opcode[5:1] == 5'b00001 || (opcode == 6'b000000 && func[5:1] == 5'b00100));
75 wire Branch = (opcode[5:2] == 4'b0001 || (opcode == 6'b000001 && rt == 5'b00001) || (opcode == 6'b000001 && rt == 5'b000000));
76 assign MemRead = (opcode[5:3] == 3'b100);
77 wire MemtoReg = (opcode[5:3] == 3'b100);
78 wire [2:0] ALUop = (opcode[5:1] == 5'b00100 || (opcode == 6'b000000 && func == 6'b100001) || opcode[5:4] == 2'b10)? `ADD:
79 (((opcode == 6'b000000 && func == 6'b100011) || (ins == `BRANCH && opcode[5:1] == 5'b00010) || (opcode == 6'b000000 && func[5:1] ==
80 ((ins == `CALI && opcode[2:0] == 3'b100) || (opcode == 6'b000000 && func == 6'b100100)? `AND:
81 ((ins == `CALI && opcode[2:0] == 3'b101) || (opcode == 6'b000000 && func == 6'b100101)? `OR:
82 ((ins == `CALI && opcode[2:0] == 3'b110) || (opcode == 6'b000000 && func == 6'b100110)? `XOR:
83 ((ins == `CAL && func[4:0] == 5'b00111)? `NOR:
84 ((ins == `CALI && opcode[2:0] == 3'b010) || (ins == `CAL && func[4:0] == 5'b01010) || (ins == `BRANCH && opcode[5:1] != 5'b0
85 assign MemWrite = (opcode[5:3] == 3'b101);
86 wire ALUSrc = (ins == `CALI || opcode[5:4] == 2'b10);
87 wire RegWrite = ~(ins == `BRANCH || opcode[5:3] == 3'b101 || opcode == 6'b000010 || (ins == `JUMPR && func[0] == 1'b0) || (ins == `MOV && (func[0] == Zero)));
88 wire Branchen = ((Branch && opcode[5:1] == 5'b00010 && ((Zero == 1'b1 && opcode[0] == 1'b0) || (Zero == 1'b0 && opcode[0] == 1'b1))) ||
89 (ins == `BRANCH && opcode[5:2] == 4'b0000 && ((rt == 5'b00000 && (ALU_res == 0 || rdata1 == 32'b0)) || (rt == 5'b00000 && ALU_res !=
90 (ins == `BRANCH && opcode == 6'b000111 && rt == 5'b00000 && ALU_res == 0) ||
91 (ins == `BRANCH && opcode == 6'b000110 && rt == 5'b00000 && (ALU_res != 0 || rdata1 == 32'b0)));
92 wire PCSrc = Branch & Branchen;

```

```

| || (opcode == 6'b000000 && func[5:1] == 5'b00101)? `SUB:
|))? `AND:
|))? `OR:
|))? `XOR:

| (ins == `BRANCH && opcode[5:1] != 5'b00010)? `STL: `STLU))));

; == `MOV && (func[0] == Zero));
|'b1))) ||
|'b0)) || (rt == 5'b00000 && ALU_res != 0)) ||

|);

```

**控制信号：**对于 control unit 的实现，这里我依照助教学姐的意见进行了一定的优化：对于 1bit 宽的信号赋值，不再使用三目运算符，而是直接采取表达式赋值，简化了代码。对照每种指令对应的控制信号表和理论课上讲过的数据通路，我对控制信号依次进行了赋值，这里注意并非每种指令对应唯一的控制信号，对于一些特殊地指令仍然要单独讨论，否则必然会出错。个人认为这是单周期处理器最艰难的部分之一，但是却没有太多的思维含量，只需要总结每条指令的控制信号即可，关键在于复杂程度较高，但是细心分类即可。

```
93 //Other ports
94 wire [31:0] PC_next = PC + 4;
95 wire [31:0] PC_BRANCH;
96 wire [31:0] J_ex = {PC_next[31:28], Instruction[25:0], 2'b0};
97 wire [31:0] J_addr = (ins == `JUMP)? J_ex:rdata1;
98 wire [31:0] zero_extend = {{16{1'b0}}, Instruction[15:0]};
99 wire [31:0] lui_extend = {imm, 16'b0};
100 wire [31:0] extend = ((opcode[5:2] == 4'b0011 && opcode != 6'b001111) || (opcode[5:1] == 5'b10010))? zero_extend:extend_imm;
101 wire [31:0] aluA = (ins == `MOV)? rdata2:rdata1;
102 wire [31:0] aluB = (ALUSrc)? extend:
103     (((ins == `BRANCH && opcode != 6'b000100 && opcode != 6'b000101) || ins == `MOV)? 32'b0:rdata2);
104 wire [31:0] ALU_res;
105 wire Zero;
106 assign Address = {ALU_res[31:2], 2'b00};
```

**其它端口信号：**除却控制信号以外，整个数据通路还有其它很多端口需要定义或赋值。对照 PPT 上的数据通路和组件结构，挨个写出即可，注意同类指令的合并。此时之前的指令分类和宏起到了作用，依照这种清楚、可读性高的写法，在写此部分代码的过程中我的思路非常清晰，一气呵成。

```

107 //Reg_file ports
108 wire [31:0] rdata1;
109 wire [31:0] rdata2;
110 wire [4:0] raddr1 = rs;
111 wire [4:0] raddr2 = rt;
112 assign RF_wen = ~(ins == `BRANCH || opcode[5:3] == 3'b101 || opcode == 6'b000010
113 || (ins == `JUMPR && func == 6'b001000) || (opcode == 6'b000000 && func[5:0] == 6'b001011 && Zero == 1)
114 || (opcode == 6'b000000 && func[5:0] == 6'b001010 && Zero == 0));
115 assign RF_waddr = (opcode == 6'b000011)? 31:((RegDst)? rd:rt);//`jar' needs REG_31, otherwise RegDst controls the waddr.
116 assign RF_wdata = (ins == `CAL || ins == `CALI)? ALU_res:
117 ((ins == `SHIFT || ins == `SHIFTV)? shifter_res:
118 ((opcode == 6'b000011 || (ins == `JUMPR && func[0] == 1'b1)) ? PC + 8:
119 ((opcode[5:3] == 3'b100)? load_res:
120 ((opcode == 6'b000000 && func[5:1] == 5'b00101)? rdata1:lui_extend)));
121 //Shifter ports
122 wire [31:0] shifter_res;
123 wire [31:0] shiftA = rdata2;
124 wire [31:0] shiftB = (ins == `SHIFT)? {{27{1'b0}},sa}: {{27{1'b0}},rdata1[4:0]};
125 wire [1:0] Shiftop = (func[5:3] == 3'b000 && func[1:0] == 2'b00 && opcode == 6'b000000)? `LEFT:
126 ((func[5:3] == 3'b000 && func[1:0] == 2'b11 && opcode == 6'b000000)? `RIGHT_ARITH:
127 ((func[5:3] == 3'b000 && func[1:0] == 2'b10 && opcode == 6'b000000)? `RIGHT_LOGIC);

```

**寄存器堆和移位器端口:**这里大部分端口是显然的(比如 raddr1 和 addr2),但是 RF 的写使能、写地址和写数据都需要进行一番计算、分类和总结。和前面的译码、分类一样,是一个体力活,需要关照每条指令,耗费时间较长且 bug 较多,后来根据波形图成功 debug,在此不再赘述。

```

128 //Mem ports
129 //Load
130 wire [31:0] lb_result = ({32{(ALU_res[1] & ALU_res[0])}} & {{24{Read_data[31]}},Read_data[31:24]}) | //high addr high byte
131 {{32{(ALU_res[1] & ~ALU_res[0])}} & {{24{Read_data[23]}},Read_data[23:16]}} |
132 {{32{(~ALU_res[1] & ALU_res[0])}} & {{24{Read_data[15]}},Read_data[15:8]}} |
133 {{32{(~ALU_res[1] & ~ALU_res[0])}} & {{24{Read_data[7]}},Read_data[7:0]}});
134
135 wire [31:0] lbu_result = {{24{1'b0}},lb_result[7:0]};
136
137 wire [31:0] lh_result = ({32{(~ALU_res[1])}} & {{16{Read_data[15]}},Read_data[15:0]}) |
138 {{32{(ALU_res[1])}} & {{16{Read_data[31]}},Read_data[31:16]}};
139
140 wire [31:0] lhu_result = {{16{1'b0}},lh_result[15:0]};
141
142 wire [31:0] lw_result = Read_data[31:0];
143
144 wire [31:0] lwl_result = ({32{(ALU_res[1] & ALU_res[0])}} & Read_data[31:0]) |
145 {{32{(ALU_res[1] & ~ALU_res[0])}} & {Read_data[23:0],rdata2[7:0]}} |
146 {{32{(~ALU_res[1] & ALU_res[0])}} & {Read_data[15:0],rdata2[15:0]}} |
147 {{32{(~ALU_res[1] & ~ALU_res[0])}} & {Read_data[7:0],rdata2[23:0]}};
148
149 wire [31:0] lwr_result = ({32{(~ALU_res[1] & ~ALU_res[0])}} & Read_data[31:0]) |
150 {{32{(ALU_res[1] & ~ALU_res[0])}} & {rdata2[31:16],Read_data[31:16]}} |
151 {{32{(~ALU_res[1] & ALU_res[0])}} & {rdata2[31:24],Read_data[31:8]}} |
152 {{32{(ALU_res[1] & ALU_res[0])}} & {rdata2[31:8],Read_data[31:24]}};
153
154
155 wire [31:0] load_res = ({32{(opcode == 6'b100000)}} & lb_result[31:0]) |
156 {{32{(opcode == 6'b100001)}} & lh_result[31:0]} |
157 {{32{(opcode == 6'b100011)}} & lw_result[31:0]} |
158 {{32{(opcode == 6'b100100)}} & lbu_result[31:0]} |
159 {{32{(opcode == 6'b100101)}} & lhu_result[31:0]} |
160 {{32{(opcode == 6'b100010)}} & lwl_result[31:0]} |
161 {{32{(opcode == 6'b100110)}} & lwr_result[31:0]};
162

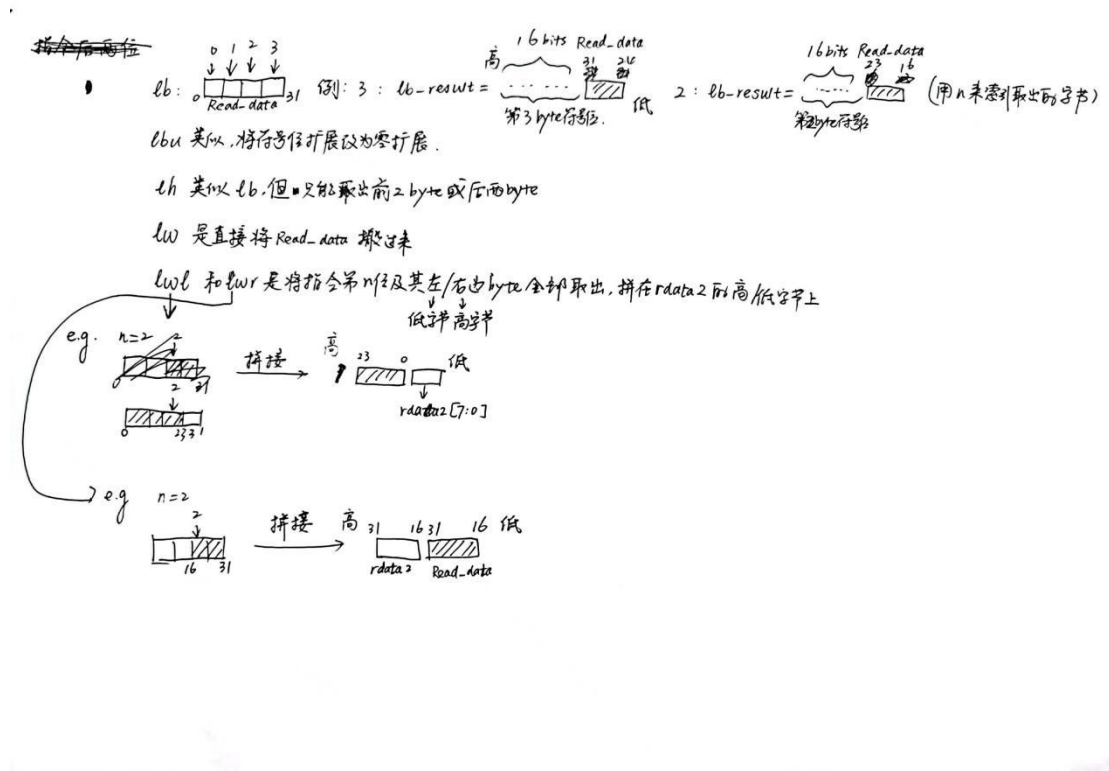
```

**Load 指令:** 这里为方便根据 ALU 运算结果低两位进行赋值,我将它们单独设置在 n 中存储,后面根据 load 指令 (signed 的和 unsigned 的) 来决定



是进行符号扩展还是零扩展，并且根据  $n$  的两位的情况来确定加载位。其中  $lwl$

和  $lwr$  指令较为特殊，根据指令手册，画出示意图如下：

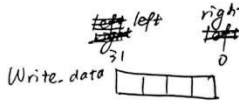


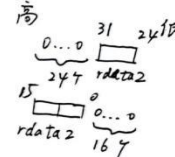
```

163 //Store
164 wire [3:0] sb_strb = ({4{(ALU_res[1] & ALU_res[0])}} & 4'b1000) |
165                      ({4{(ALU_res[1] & ~ALU_res[0])}} & 4'b0100) |
166                      ({4{(~ALU_res[1] & ALU_res[0])}} & 4'b0010) |
167                      ({4{(~ALU_res[1] & ~ALU_res[0])}} & 4'b0001);
168
169 wire [3:0] sh_strb = (ALU_res[1])? 4'b1100: 4'b0011;
170
171 wire [3:0] sw_strb = 4'b1111;
172
173 wire [3:0] swl_strb = {ALU_res[1] & ALU_res[0], ALU_res[1], ALU_res[1] | ALU_res[0], 1'b1};
174
175 wire [3:0] swr_strb = {1'b1, ~ALU_res[1] | ~ALU_res[0], ~ALU_res[1], ~ALU_res[1] & ~ALU_res[0]};
176
177 assign Write_strb = ({4{(opcode == 6'b101000)}} & sb_strb[3:0]) |
178                    ({4{(opcode == 6'b101001)}} & sh_strb[3:0]) |
179                    ({4{(opcode == 6'b101011)}} & sw_strb[3:0]) |
180                    ({4{(opcode == 6'b101010)}} & swl_strb[3:0]) |
181                    ({4{(opcode == 6'b101110)}} & swr_strb[3:0]);
182
183 wire [31:0] sb_data = ({32{(ALU_res[1] & ALU_res[0])}} & {rdata2[7:0], {24{1'b0}}}) |
184                      ({32{(ALU_res[1] & ~ALU_res[0])}} & {{8{1'b0}}, rdata2[7:0], {16{1'b0}}}) |
185                      ({32{(~ALU_res[1] & ALU_res[0])}} & {{16{1'b0}}, rdata2[7:0], {8{1'b0}}}) |
186                      ({32{(~ALU_res[1] & ~ALU_res[0])}} & {{24{1'b0}}, rdata2[7:0]});
187
188 wire [31:0] sh_data = (ALU_res[1])? {rdata2[15:0], {16{1'b0}}} : {16{1'b0}}, rdata2[15:0];
189
190 wire [31:0] sw_data = rdata2[31:0];
191
192
193 wire [31:0] swl_data = ({32{(swl_strb == 4'b0001)}} & {{24{1'b0}}, rdata2[31:24]}) |
194                      ({32{(swl_strb == 4'b0011)}} & {{16{1'b0}}, rdata2[31:16]}) |
195                      ({32{(swl_strb == 4'b0111)}} & {{8{1'b0}}, rdata2[31:8]}) |
196                      ({32{(swl_strb == 4'b1111)}} & rdata2[31:0]);
197
198 wire [31:0] swr_data = ({32{(swr_strb == 4'b1111)}} & rdata2[31:0]) |
199                      ({32{(swr_strb == 4'b1110)}} & {rdata2[23:0], {8{1'b0}}}) |
200                      ({32{(swr_strb == 4'b1100)}} & {rdata2[15:0], {16{1'b0}}}) |
201                      ({32{(swr_strb == 4'b1000)}} & {rdata2[7:0], {24{1'b0}}});
202
203 assign Write_data = ({32{(opcode == 6'b101000)}} & sb_data[31:0]) |
204                    ({32{(opcode == 6'b101001)}} & sh_data[31:0]) |
205                    ({32{(opcode == 6'b101011)}} & sw_data[31:0]) |
206                    ({32{(opcode == 6'b101010)}} & swl_data[31:0]) |
207                    ({32{(opcode == 6'b101110)}} & swr_data[31:0]);

```

**Store 指令：**由于比 load 指令多出一个步骤，我们需要先确定 strb 信号的取值，以此控制内存写的的数据位，swl 和 swr 同样特殊，根据指令示意图，画出示意图如下：


 (注: 此时我们观察 Write-data 的方向是与刚才 load 示意图方向相反, 是  
 为要明瞭地看出取哪几位字节)  
 特殊例, swl-strb 取值为 0001, 0011, 0111, 1111 (b)  
 SWR-strb 取值为 1000, 1100, 1110, 1111 (b)  
 n=0 n=1 n=2 n=3  
 n=3 n=2 n=1 n=0  
 例: swl-strb = 4'b0001  $\Rightarrow$  取 left 高 1 位 byte 排在低位, 并零扩展  
 SWR-strb = 4'b1100  $\Rightarrow$  取 right 低 2 个 byte 排在高位, 并补零

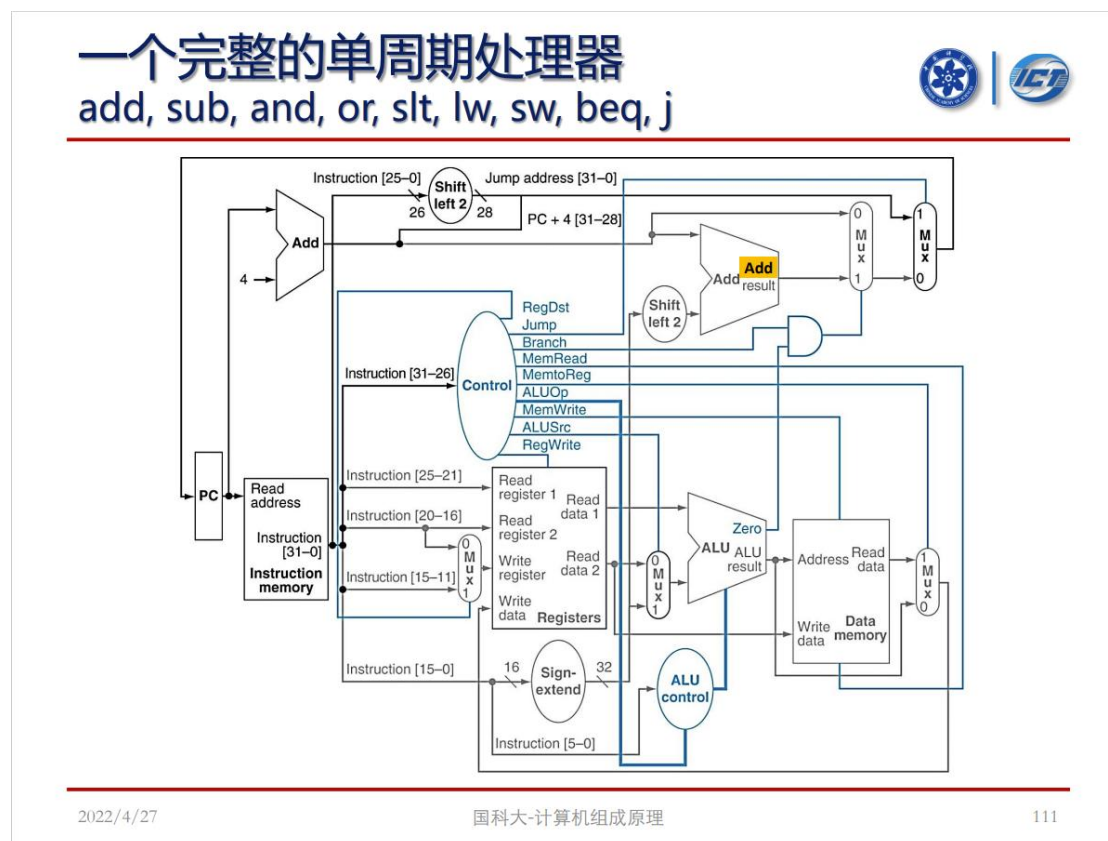

 高 31 24 低  
 0...0 24 7 rdata2  
 16 0...0 16 7  
 rdata2

```

194
195 //Module call: reg_file, shifter, alu*2
196 reg_file reg_file_module(
197     .clk(clk),
198     .rst(rst),
199     .waddr(RF_waddr),
200     .raddr1(raddr1),
201     .raddr2(raddr2),
202     .wen(RF_wen),
203     .wdata(RF_wdata),
204     .rdata1(rdata1),
205     .rdata2(rdata2)
206 );
207
208 shifter shifter_module(
209     .A(shiftA),
210     .B(shiftB),
211     .Shiftop(Shiftop),
212     .Result(shifter_res)
213 );
214
215 alu alu_branch(
216     .A(two_extend_imm),
217     .B(PC_4),
218     .ALUop('ADD'),
219     .Result(PC_BRANCH),
220     .Overflow(),
221     .CarryOut(),
222     .Zero()
223 );
224
225 alu alu_main(
226     .A(aluA),
227     .B(aluB),
228     .ALUop(ALUop),
229     .Result(ALU_res),
230     .Overflow(),
231     .CarryOut(),
232     .Zero(Zero)
233 );
234
235 endmodule
236
  
```

模块调用：调用寄存器堆 reg\_file.v 和移位器 shifter.v，这里注意运算器

alu.v 需要调用两次，因为 PC 需要单独使用一个加法器进行跳转地址的计算。



如图所示即我的代码对应的数据通路以及逻辑电路示意图。唯一的不同点在于最上面的 shifter，我并没有使用这一个 shifter，而是直接拼接，节省了模块调用。

## 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）

首先，因为是第一次写百行量级的 verilog 代码，而且距离上次实验已经很长时间，我对 verilog 语法的掌握变得生疏起来，在最先的多次提交中都因为 rtl\_chk 语法错误而烦恼，后来经过李衍君同学的点醒，我在本地虚拟机安装的

vscode 中找到了检查 verilog 语法的插件 iverilog，并实现了就地检查语法，不必多次提交 serve 等待检查结果。

其次，在行为仿真阶段，一开始我只通过了一个样例，有多达 29 个样例均未通过，后来通过查看仿真波形来逐步 debug。在 debug 的过程中我采取的方法是先将未通过样例的错误波形信号逐个点开，总结哪几个 wire 或者 reg 出了问题，再回头专门检查这几个 wire 或 reg。为此，我从波形图界面点开 Instruction 以及和这几个信号有关的所有信号（数据通路上的相关信号），根据指令本身（在纸上手动译码出指令内容），分析每个信号在该指令下是否正确。个人认为这种方法便捷高效，能非常精确地定位 bug 来源。但是在此过程中出了一个很深的 bug：我在 ALU 的修改中错误地认为按位或|运算符比加法+运算符优先级高，导致我的 ALU 内部信号出错。为此我盯着代码看了两个小时。事实说明，对语法的熟练掌握不仅能提高写代码效率和减少出错，在 debug 时也不会产生如此难以意识到的错误（因为完全没有意识到这个错误，我甚至一度怀疑是查看波形图的软件本身出了故障）。

最后，也是当时比较困扰我的问题出在 bit\_gen 的环节，serve 网站上的报错显示我有 multi-drive 问题，但是苦于我不了解这个问题，在网上查找了半天也没有太多收获，一直到凌晨 2 点才在群里询问助教老师，最终得到了陈欲晓老师的解答，豁然开朗，第二天早晨就修改 simple\_cpu.v 代码消除了多驱动问题——问题原来出在我在模块内部又以 wire 的身份对多个 output 端口进行了赋值，产生了多驱动，事实上直接对其进行 assign 即可。这个问题的根源在于，我本意是对信号都采用 wire 定义的同时赋值，这样节省了 assign 的步骤，大大降低了代码量，但是没有意识到不能对模块 output 做出类似操作。



与 bug 们经此一战，我对 verilog 代码的掌握程度空前提高，并且经过自己的实践、探索和思考，熟悉了一套 debug、解决多驱动等问题的方法，并且养成了良好的代码习惯。只能说，计算机真是一门实践出真知的学科！

### 三、 对讲义中思考题（如有）的理解和回答

#### 1.5.3 ALUOp编码



R-Type 指令	func (6-bit)	ALUOp (3-bit)	ALUOp编码	opcode (6-bit)	I-Type 指令	ALUOp编码
add	10 00 00	010	ADD/SUB: func[3:2] == 2'b00	001 0 00	addi	ADD: opcode[2:1] == 2'b00 ALUOp = {opcode[1], 2'b10}
addu	10 00 01			001 0 01	addiu	
sub	10 00 10	110	ALUOp = {func[1], 2'b10}			
subu	10 00 11					
and	10 01 00	000	逻辑运算: func[3:2] == 2'b01	001 1 00	andi	逻辑运算: opcode[2] == 1'b1
or	10 01 01	001		001 1 01	ori	ALUOp = {opcode[1], 1'b0, opcode[0]}
xor	10 01 10	100	ALUOp = {func[1], 1'b0, func[0]}	001 1 10	xori	
nor	10 01 11	101		001 1 11	lui	非运算类指令，需单独处理
slt	10 10 10	111	比较运算: func[3:2] == 2'b10	001 0 10	slti	比较运算: opcode[2:1] == 2'b01
sltu	10 10 11	011	ALUOp = {~func[0], 2'b11}	001 0 11	sltiu	ALUOp = {~opcode[0], 2'b11}

- 新增3个ALUOp，分别对应XOR、NOR和SLTU操作（表格中橙色字）
- 不允许改变实验项目1中已实现的5种操作编码
- **思考题：**ALUOp的编码有什么规律？表格中的ALUOp编码是否还有优化空间？

2022/4/15

国科大-计算机组成原理（研讨课）

33

**思考：**图中所示 ALUOp 编码的特点在于对于常见和常用的运算（如加减、按位与、按位或），其编码简单且有效位数低。另外，不难察觉到一个事实：类似的运算之间编码几乎都只相差 1bit（这令人不禁联想到格雷码，这样的编码有利于合并相似运算操作，化简逻辑表达式，这是在数字电路课程中提及过的思想），综合其它指令的对应 op 信号判断来看，图中的 ALUOp 编码已经达到了比较优化的情况，可优化空间已经较小。

但是这并非代表不可优化。如果硬要说有什么优化手段，我认为 R-type 和 I-type 指令的共同点在于 func 和 opcode 低三位一致，因此可以定义一个新的 wire 类型的选择信号，用 opcode[3]来选通该选择信号，这样，利用新信号，

结合 func 或 opcode 低三位来确定每个指令对应的 ALUOp 编码。这是一种可行的方案，但是空间成本增加了。只能说这种力度较低的优化可以做但没必要。

四、 在课后，你花费了大约\_\_18\_\_小时完成此次实验。

修改 alu.v 与 reg\_file.v 文件花费了 0.5h，写出 shifter.v 文件花费 1h, 写出 simple\_cpu.v 文件以及最后的 debug 和解决问题花费了 16h 左右(事实上，我有一天坐在座位上从下午 15:00 一直写到了凌晨 2:00 而未休息或者吃饭，只为写 simple\_cpu.v)

五、 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. 本实验过程中遇到的障碍之一是实验课课件上的示意图有错误……实验课课件上所示的数据通路基本不全，而且有时候没有显示关键信号，具有一定迷惑性，刚开始的时候按照实验课课件写，逐渐意识到不对劲，发现错误百出。最后，我根据理论课课件才找到正确且完整的示意图，成功写出正确代码。

2. 课堂验收的时候出了点问题。因为熬了一周的夜，前一天晚上还熬夜复习拓扑，早八期中考试，上了整整一天的课，中午也没睡觉，傍晚实验课验收的时候脑子不太清醒，对助教老师的提问有点懵，脑子没转过来，反应有点慢，而且 jr 指令指错了信号。在此声明：此单周期 CPU 代码为本人原创，验收时表现不佳纯属个人身体状态和熬夜原因。而且写完后在验收前没来得及 remake 一

遍当时写的代码和指令表。

3. 致谢：感谢李衍君同学，在语法检查上提醒了我 vscode 中 iverilog 插件的存在，让我能够快速进行语法错误修改。