中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929010 姓名：贾城昊 专业：计算机科学与技术

实验序号：2 实验名称：简单功能性处理器设计划(基于 MIPS32 位指令集)

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。

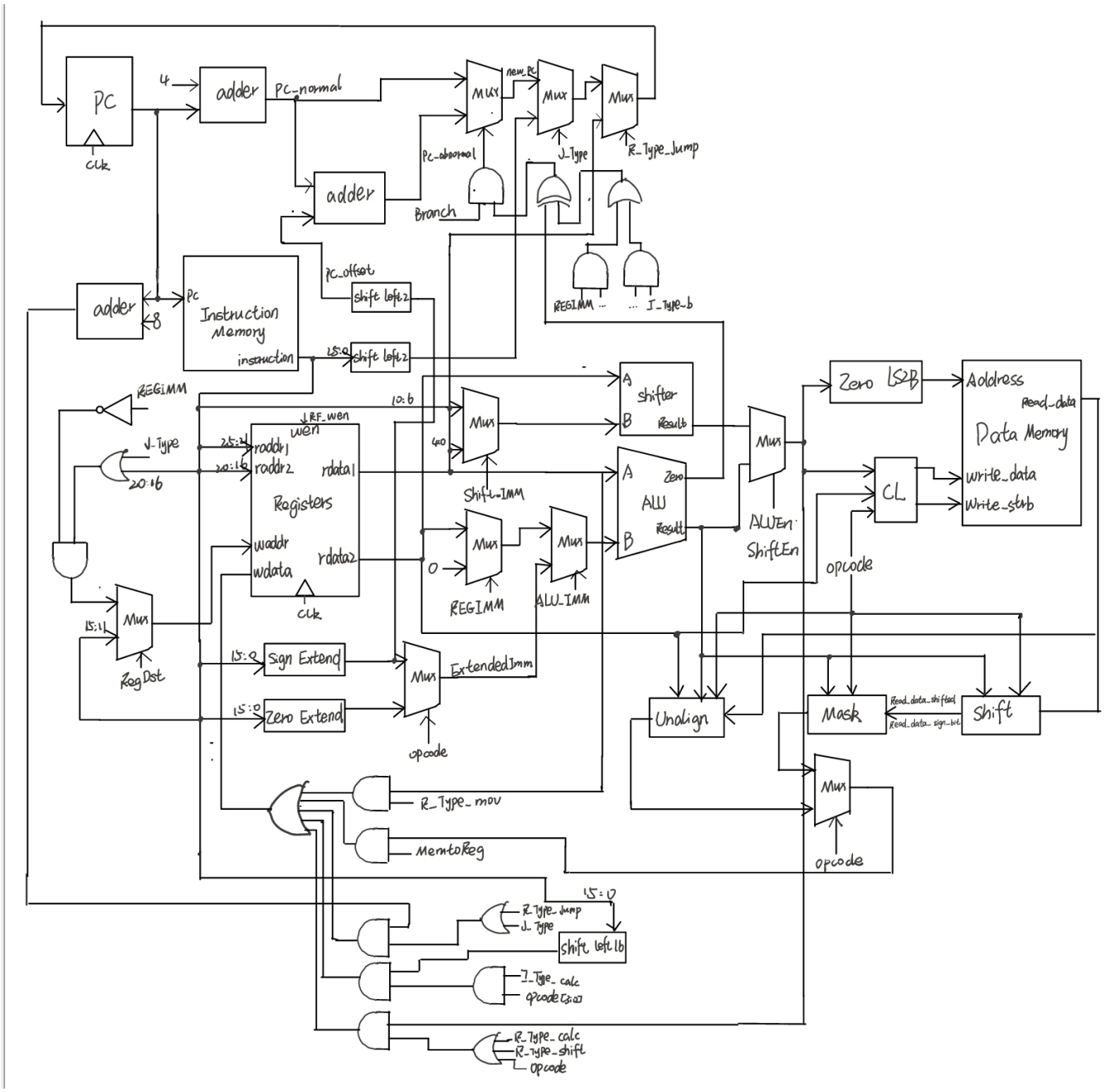
注 2：使用 git add 及 git commit 命令将实验报告 PDF 文件添加到本地仓库 master 分支，并通过 git push 推送到 SERVE GitLab 远程仓库 master 分支（具体命令详见实验报告）。

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．单周期处理器:

1） 逻辑电路图



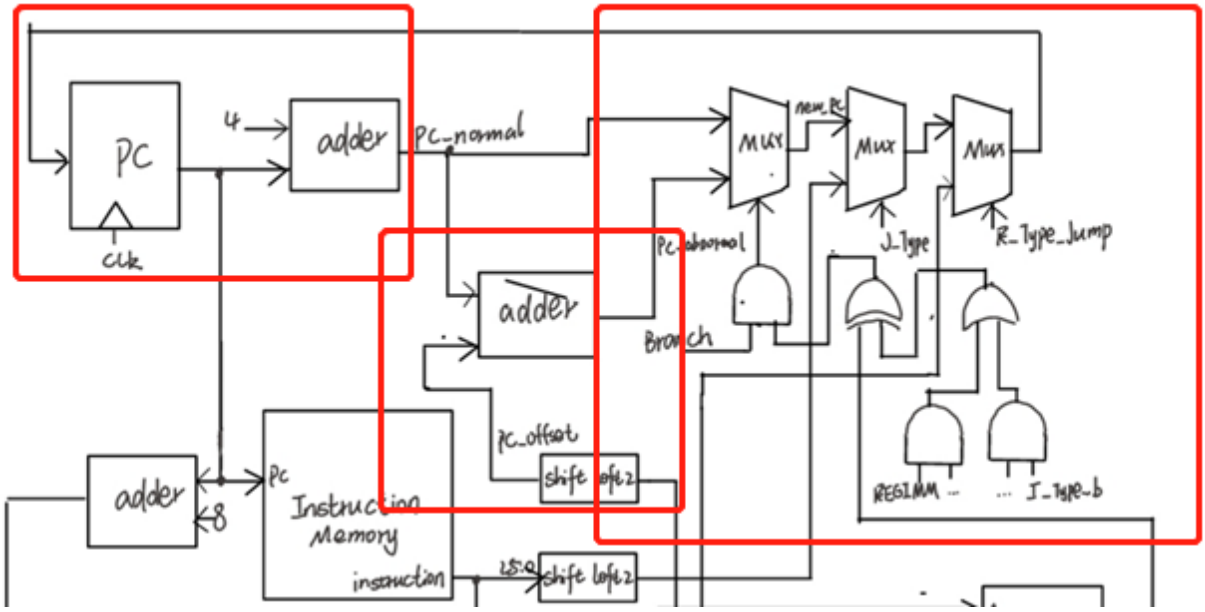
2） 关键 RTL 代码段：

a） PC 更新

```
//PC
reg  [31:0] PC;
wire [31:0] new_PC;
wire [31:0] PC_normal;
wire [31:0] PC_abnormal;
assign PC_normal = PC + 4;
assign new_PC = (Branch & (Zero ^ (
        REGIMM & ~Instruction[16] |
        | I_Type_b & (opcode[0] ^ (opcode[1] & (|RF_rdata1))))))) ? PC_abnormal : PC_normal;//是否branch

always @(posedge clk) begin
    if(rst) begin
        PC <= 32'd0;
    end
    else if(R_Type_jump) begin
        PC <= RF_rdata1;
    end
    else if(J_Type) begin
        PC <= {PC_normal[31:28], Instruction[25:0], 2'b00};
    end
    else begin
        PC <= new_PC;
    end
end
```

```
//PC_abnormal
wire [31:0] PC_offset;
assign PC_offset        = {SignExtend[29:0], 2'b00}; //offset*4
assign PC_abnormal      = PC_normal + PC_offset;
```

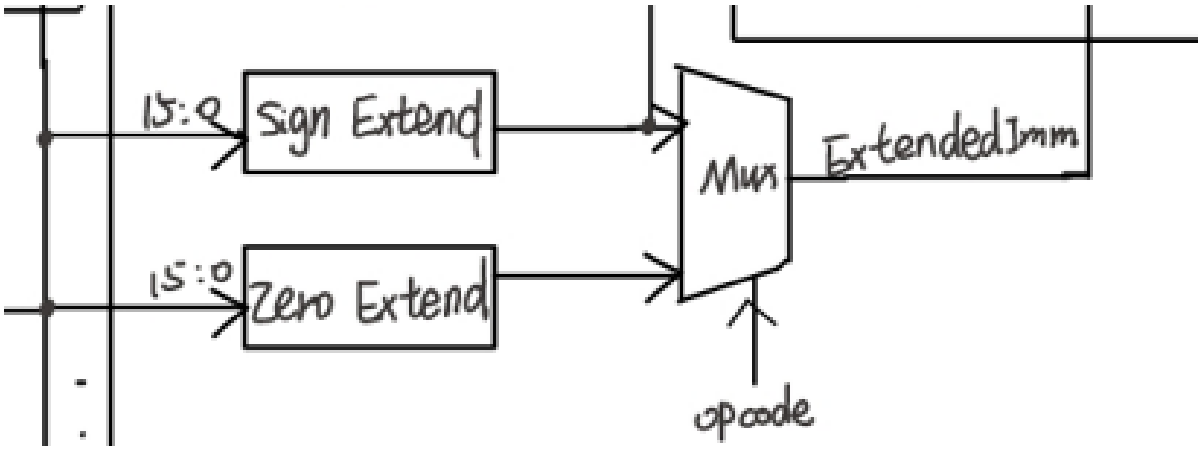
如图，对于跳转指令，如果是 R_Type 的跳转，直接由寄存器的值确定，如果是 J_Type，则是由 PC_normal（PC+4），Instruction 拼接而成；对于分支指令，如果条件满足，PC 的值更新为 PC_normal+offset 的 PC_abnormal；对于其它指令，则直接更新为 PC_normal（PC+4），结构图如下：



b) 立即数扩展

```
//EXTEND_IMM
wire [31:0] SignExtend;
// sign extend
assign SignExtend = {{(16){Instruction[15]}}, Instruction[15:0]};
//Zero extend for andi, ori, xori
wire [31:0] ZeroExtend;
assign ZeroExtend = {16'b0, Instruction[15:0]};
//Select
wire [31:0] ExtendedImm;
assign ExtendedImm = (opcode[5:2] == 4'b0011) ? ZeroExtend : SignExtend;
```

如图，根据指令的要求对立即数进行零扩展或者符号位扩展，其中只有 andi，ori，xori 才需要零扩展，其它均需要符号位扩展，结构图如下：



c) 控制信号生成

```
wire [5:0] opcode;
wire [4:0] rs;
wire [4:0] rt;
wire [4:0] rd;
wire [4:0] shamt;
wire [5:0] func;
assign {opcode,rs,rt,rd,shamt,func} = Instruction;
```

```
assign R_Type          = opcode[5:0] == 6'b000000;
assign R_Type_calc     = R_Type & func[5];
assign R_Type_shift    = R_Type & (func[5:3]==3'b000);
assign R_Type_jump     = R_Type & ({func[5:3],func[1]}==4'b0010);
assign R_Type_mov      = R_Type & ({func[5:3],func[1]}==4'b0011);
assign REGIMM          = opcode[5:0] == 6'b000001;
assign J_Type          = opcode[5:1] == 5'b00001;
assign I_Type_b        = opcode[5:2] == 4'b0001;
assign I_Type_calc     = opcode[5:3] == 3'b001;
assign I_Type_mr       = opcode[5:3] == 3'b100;
assign I_Type_mw       = opcode[5:3] == 3'b101;

wire RegDst,Branch,MemtoReg,ALUEn,ShiftEn,ALU_IM,Shift_IM;

assign RegDst   = R_Type;
assign Branch   = REGIMM | I_Type_b;
assign ALUEn    = R_Type_calc | REGIMM | I_Type_b | I_Type_calc | I_Type_mr | I_Type_mw;
assign ShiftEn  = R_Type_shift;
assign ALU_IM   = I_Type_calc | I_Type_mr | I_Type_mw; //立即数计算
assign Shift_IM = ~func[2]; //立即数移位
assign MemWrite = I_Type_mw;
assign MemRead  = I_Type_mr;
assign MemtoReg = I_Type_mr;
assign RF_wen   = (R_Type &
                  ~(R_Type_mov & (func[0] ^ (|RF_rdata2))) & //move指令条件满足才写
                  ~(R_Type_jump & ~func[0])) | //jr指令RF_wen不能位1
                  J_Type & opcode[0] | I_Type_calc | I_Type_mr;
```

如图，首先根据 instruction 确定指令类型，然后根据指令类型(R_Type，J_Type，REGIMM 以及各类 I_Type)，生成各类控制信号。

d) ALU 和 Shifter

```
assign ALUop = {(3){R_Type_calc}} & {func[1] & ~(func[3] & func[0]), ~func[2], func[3] & ~func[2] & func[1] | func[2] & func[0]} |
               {(3){I_Type_calc}} & {opcode[1] & ~(opcode[3] & opcode[0]), ~opcode[2],opcode[3] & ~opcode[2] & opcode[1] | opcode[2] & opcode[0]} |
               {(3){REGIMM}} |
               {(3){I_Type_b}}  & {2'b11, opcode[1]} | // slt 111 sub 110
               {(3){I_Type_mr | I_Type_mw}} & 3'b010;

assign Shiftop = func[1:0];
```

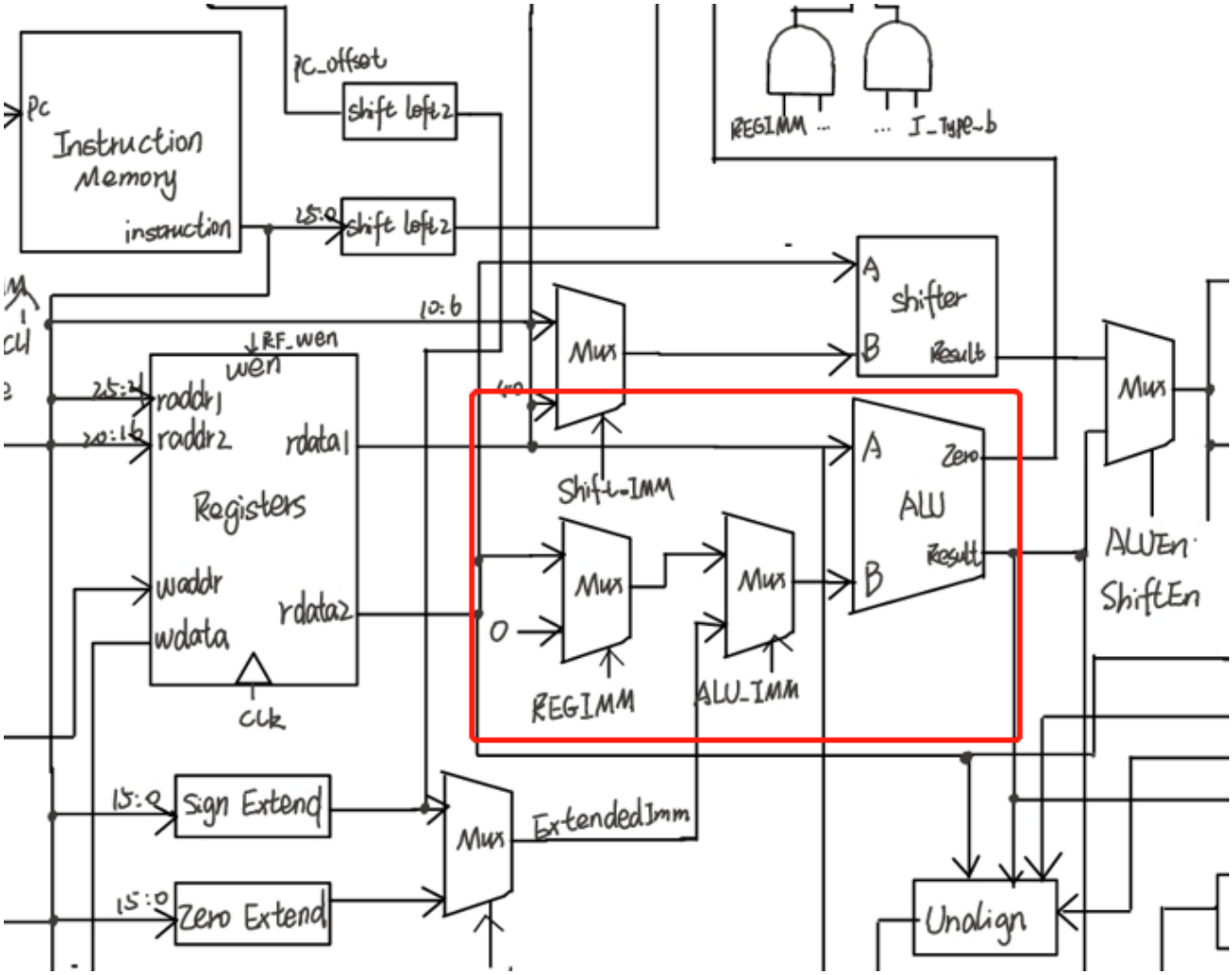
```
wire Overflow;
wire CarryOut,Zero;
wire [31:0] Result;
wire [31:0] ALUResult;
wire [31:0] ShifterResult;

//ALU
alu ALU(
        .A              (RF_rdata1),
        .B              (ALU_IM ? ExtendedImm : REGIMM ? 32'b0 :RF_rdata2),
        .ALUop          (ALUop),
        .Overflow       (Overflow),
        .CarryOut       (CarryOut),
        .Zero           (Zero),
        .Result         (ALUResult)
);

shifter Shifter(
        .A              (RF_rdata2),
        .B              (Shift_IM ? shamt : RF_rdata1[4:0]),//移位对32取模
        .Shiftop        (Shiftop),
        .Result         (ShifterResult)
);
```

如图，ALU 的 A 接口就是寄存器第一读端口的值，而 B 则是根据指令类型进行选择，如果 ALU_IM 的信号为 1，则为扩展的立即数，若为 REGIMM 类型指令，则为全 0，否则为寄存器第二读端口的值。

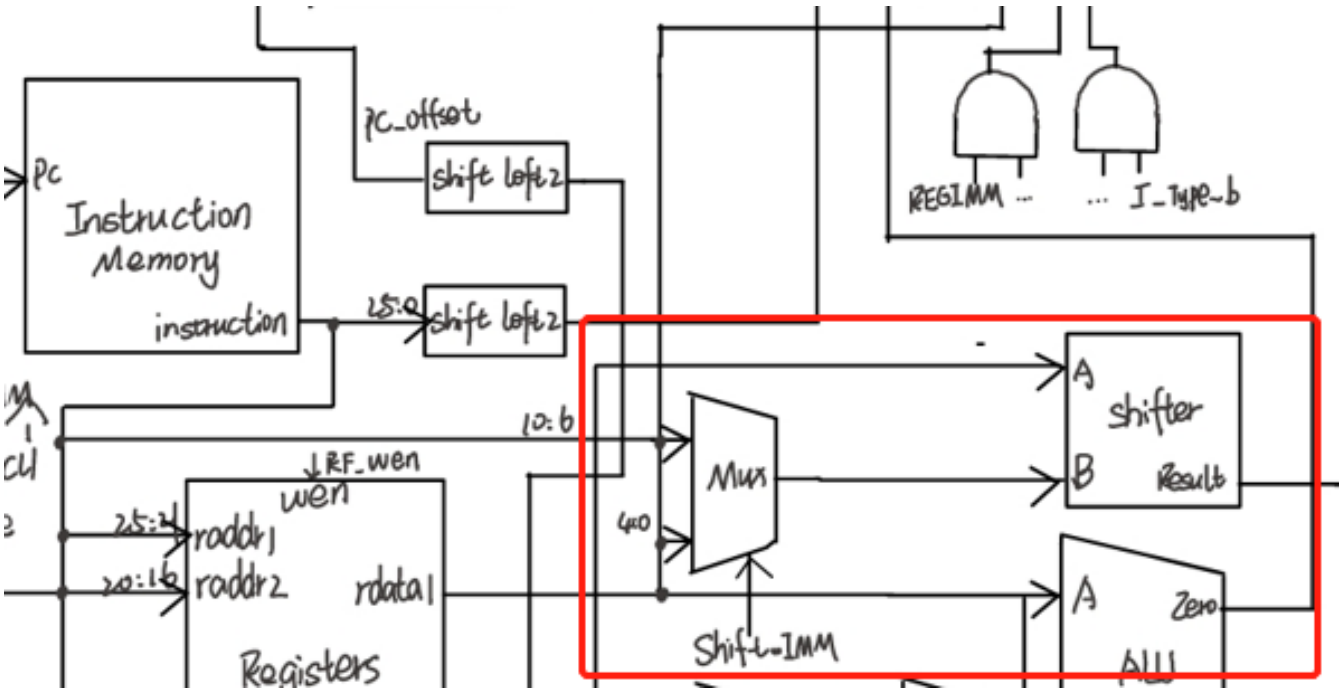
Shifer 的 A 接口就是寄存器第一读端口的值，而 B 则也同样需要根据指令类型进行选择，如果 Shift_IM 的信号为 1，则为 shamt，否则则为寄存器第二读端口对 32 取模的值（取低五位）。

ALUop 的生成是一个难点，根据 PPT 和指令集手册分指令类型可以发现 ALUop 需要根据不同的指令类型进行编码。而 Shitop 的生成则比较简单，取

func[1:0]即可。结构图如下：



ALU 结构图

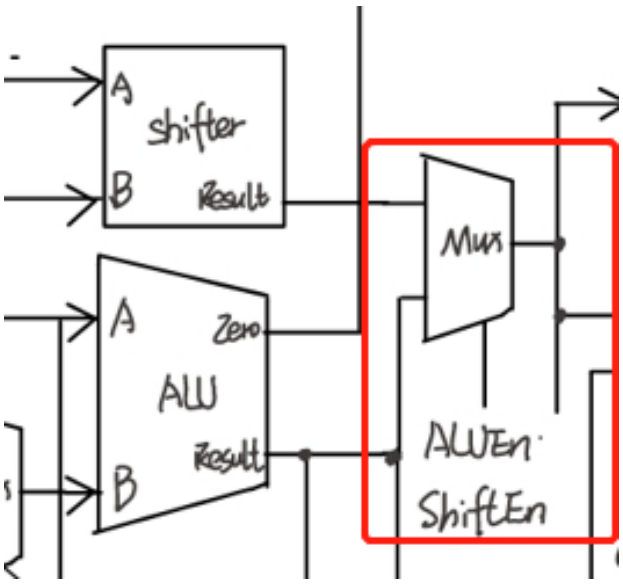


Shifter 结构图

e) 结果选择

```
//Result
assign Result = {(32){ALUEn}} & ALUResult |
                {(32){ShiftEn}} & ShifterResult;
```

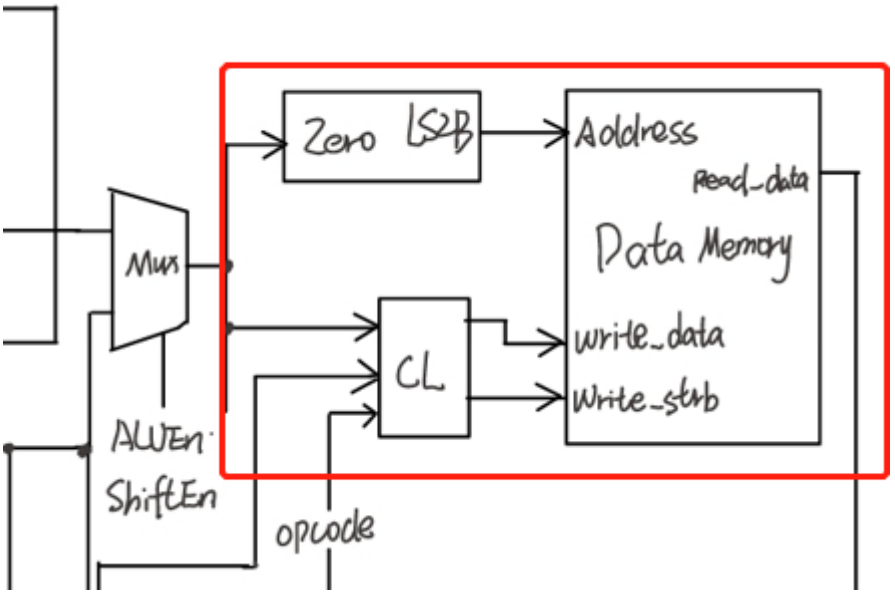
根据 ALUEn 和 ShiftEn 对结果进行选择，如下所示：



f)　内存写地址和写入数据

```
//Data Memory
assign Address          = Result & ~32'b11; //对齐
assign Write_data       = (opcode[2:0] == 3'b010) ? RF_rdata2 >> {~Result[1:0], 3'b0} : RF_rdata2 << {Result[1:0], 3'b0}; //除swl外移位规则相同

assign Write_strb       = {(4){~opcode[2] & opcode[1] & ~opcode[0]}} & {Result[1] & Result[0], Result[1], Result[1] | Result[0], 1'b1} | //swl
                          {(4){ opcode[2] & opcode[1] & ~opcode[0]}} & {1'b1, ~(Result[1] & Result[0]), ~Result[1], ~(Result[1] | Result[0])} | //swr
                          {(4){~opcode[1] | opcode[0]}} &{( Result[1] | opcode[1]) & ( Result[0] | opcode[0]),
                                                          ( Result[1] | opcode[1]) & (~Result[0] | opcode[0]),
                                                          (~Result[1] | opcode[1]) & ( Result[0] | opcode[0]),
                                                          (~Result[1] | opcode[1]) & (~Result[0] | opcode[0])};
```

如图，首先地址需要对齐（后两位为 0），然后由于地址的对齐，写入的数据也需要进行移位，而通过对指令集研究可以发现，除了 swl 外，其它指令的移位规则相同（如图所示），而 Write_strb 需要对 swl 和 swr（非对齐写入）指令单独处理。结构图如下（Zero LS2B 指将最低两位变为 0，而 CL 的具体逻辑，在上面代码段已经做了展示）：

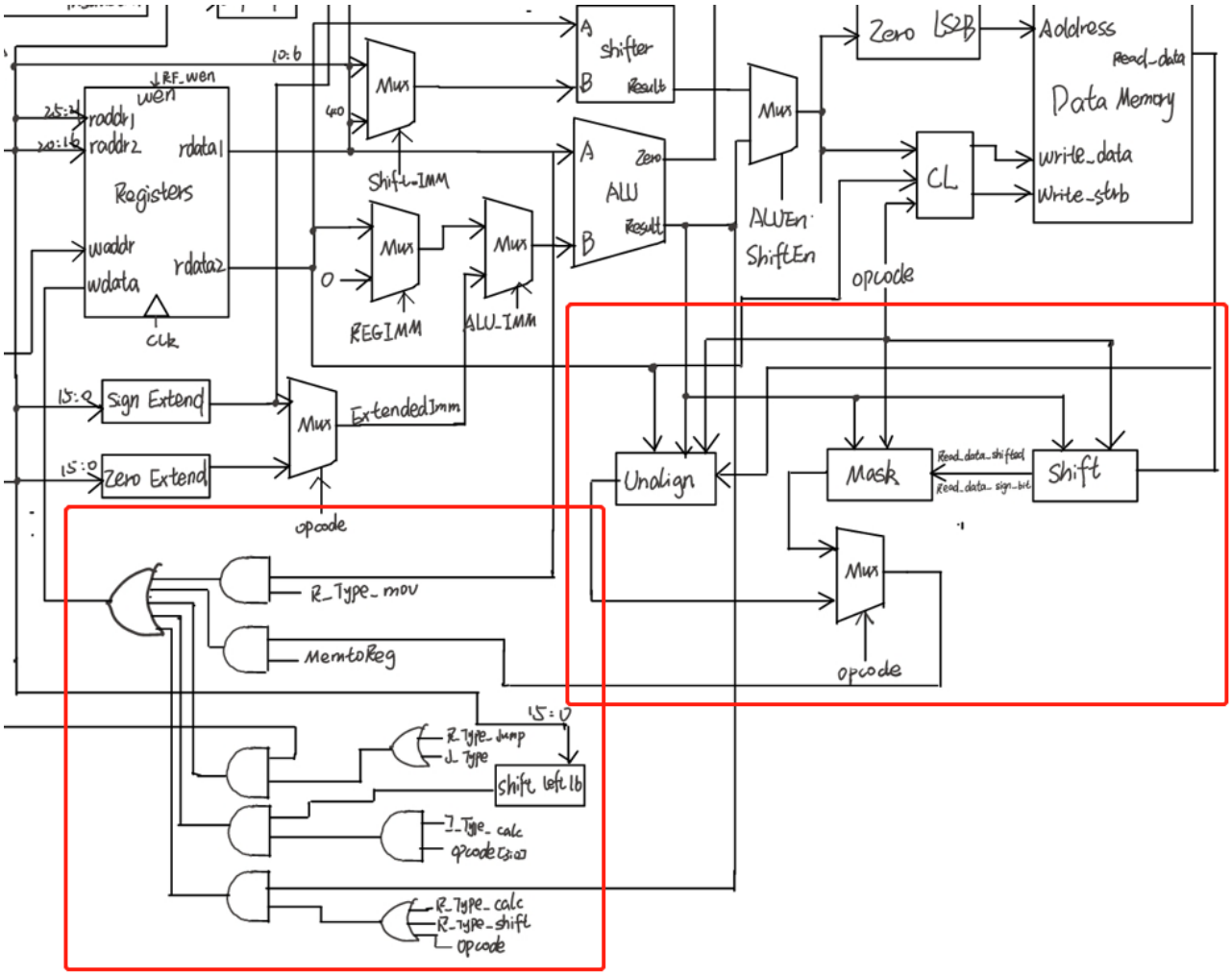


g)　内存读数据与寄存器堆的写数据

```
//WB
wire [31:0] Read_data_shifted;
wire [31:0] Read_data_masked;
wire Read_data_sign;
wire [31:0] Read_data_unaligned;
assign Read_data_shifted        = Read_data >> {Result[1:0], 3'b0};
assign Read_data_sign           = Read_data_shifted[(opcode[1:0]==2'b01) ? 15 : 7];//符号位

assign Read_data_masked         = Read_data_shifted & {{(16){opcode[1]}}, {(8){opcode[0]}}, {(8){1'b1}}} |
                                  {(32){~opcode[2] & Read_data_sign}} & ~{{(16){opcode[1]}}, {(8){opcode[0]}}, {(8){1'b1}}};//字节掩码进行有符号扩展

assign Read_data_unaligned      = {(32){~opcode[2]}} & ((Read_data << {~Result[1:0], 3'b0}) | RF_rdata2 & ({(32){1'b1}} >> { Result[1:0], 3'b0})) |
                                  {(32){opcode[2]}} & ((Read_data >> { Result[1:0], 3'b0}) | RF_rdata2 & ({(32){1'b1}} << {~Result[1:0], 3'b0}));//字节掩码保留

assign RF_wdata = {(32){MemtoReg & (~opcode[1] |  opcode[0])}} & Read_data_masked |
                  {(32){MemtoReg & ( opcode[1] & ~opcode[0])}} & Read_data_unaligned | //访存指令的数据选取
                  {(32){R_Type_mov}} & RF_rdata1  | //move指令
                  {(32){R_Type_jump | J_Type}} & PC + 8 | //跳转指令
                  {(32){I_Type_calc & (&opcode[3:0])}} & {Instruction[15:0], 16'd0} | //lui单独处理
                  {(32){R_Type_calc | R_Type_shift | (I_Type_calc & ~(&opcode[3:0]))}} & Result;
```

如图，需要对非对齐读数据指令单独处理，通过移位和掩码保存寄存器读出数据的其它位的值。而对齐读数据需要通过字节掩码进行符号位扩展。

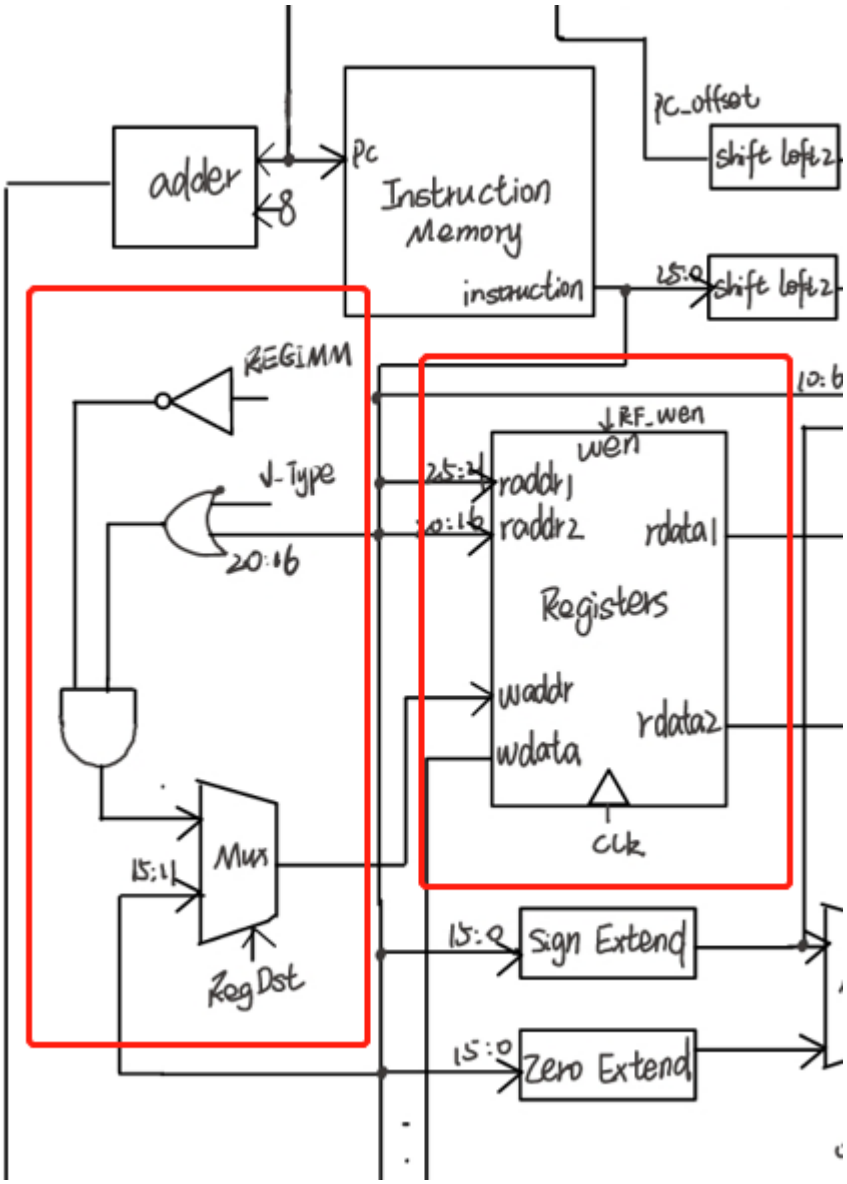
而寄存器堆的写回数据则需要根据指令类型进行选择，且在这对 lui 指令单独处理。结构图如下（shift，Mask，Unalign 单元的具体逻辑在上面代码段进行了展示）：


RF_wen
wen
Registers
raddr1
raddr2
rdata1
rdata2
waddr
wdata
clk
Mux
Shift_IMM
REGIMM
ALU_IMM
shifter
ALU
Result
Zero
ALUEn
ShiftEn
Zero LSB
Address
Read_data
Data Memory
write_data
Write_stb
CL
opcode
Sign Extend
Zero Extend
ExtendedImm
Unalign
Mask
Shift
Read_data_shifted
Read_data_sign_bit
R_Type_mov
MemtoReg
R_Type_jump
J_Type
Shift left 16
J_Type_calc
R_Type_calc
R_Type_shift

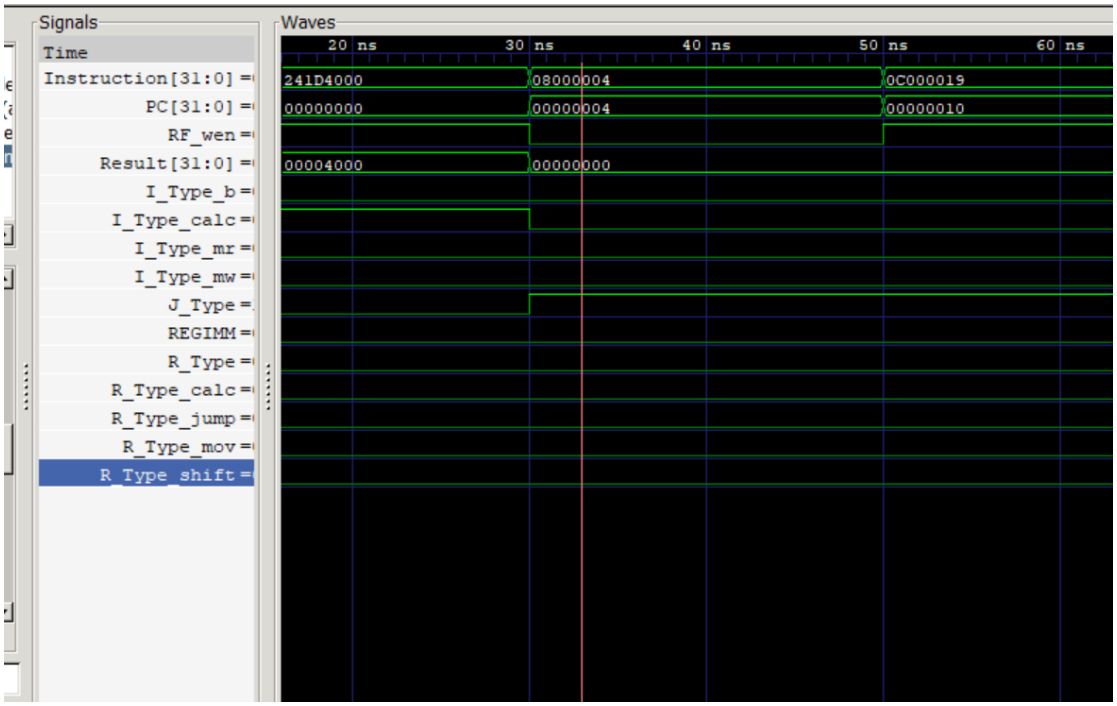
h)　寄存器堆

```
//Registers
wire [31:0] RF_rdata1;
wire [31:0] RF_rdata2;

assign RF_waddr = {(5){R_Type}} & rd |
                  {(5){I_Type_mr | I_Type_calc}} & rt |
                  {(5){J_Type}} & 5'b11111;


reg_file Registers(
        .clk    (clk),
        .waddr  (RF_waddr),
        .raddr1 (rs),
        .raddr2 (rt),
        .wen    (RF_wen),
        .wdata  (RF_wdata),
        .rdata1 (RF_rdata1),
        .rdata2 (RF_rdata2)
);
```

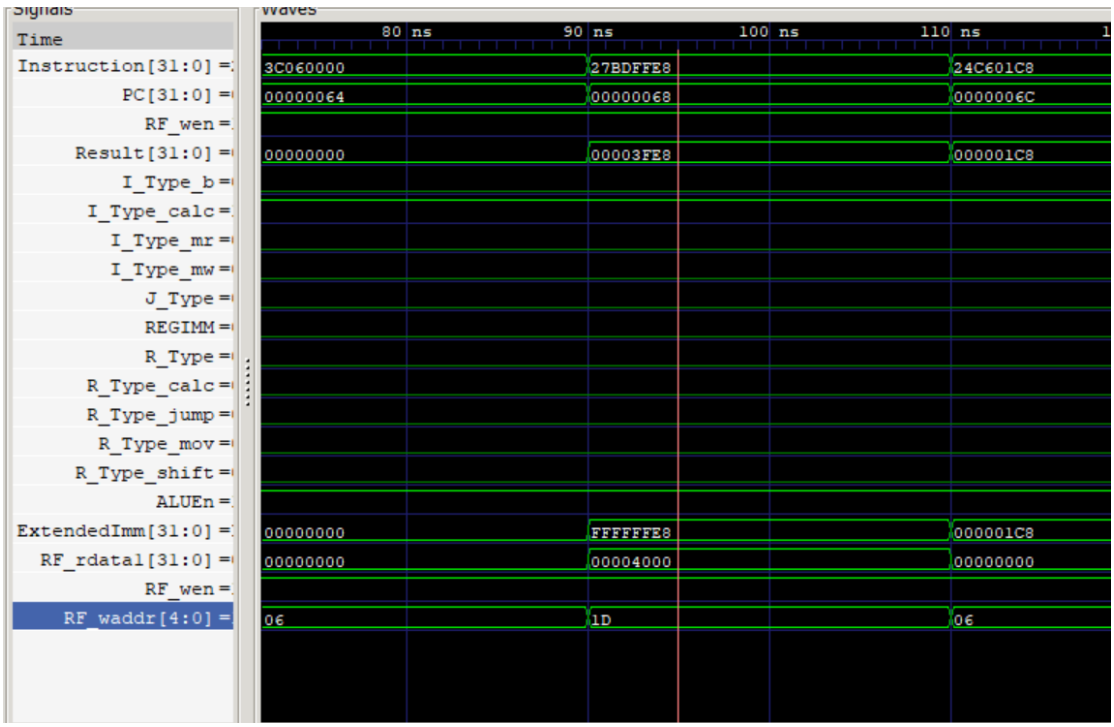
RF_waddr 需要根据指令类型选择是 rd 还是 rt 还是 31,结构图如下:


adder
8
Pc
Instruction Memory
instruction
pc_offset
shift left 2
25:0
shift left 2
REGIMM
J-Type
20:16
RF_wen
wen
25:21
raddr1
20:16
raddr2
rdata1
Registers
waddr
wdata
rdata2
clk
Mux
15:1
RegDst
15:0
Sign Extend
15:0
Zero Extend

3） 波形图示例

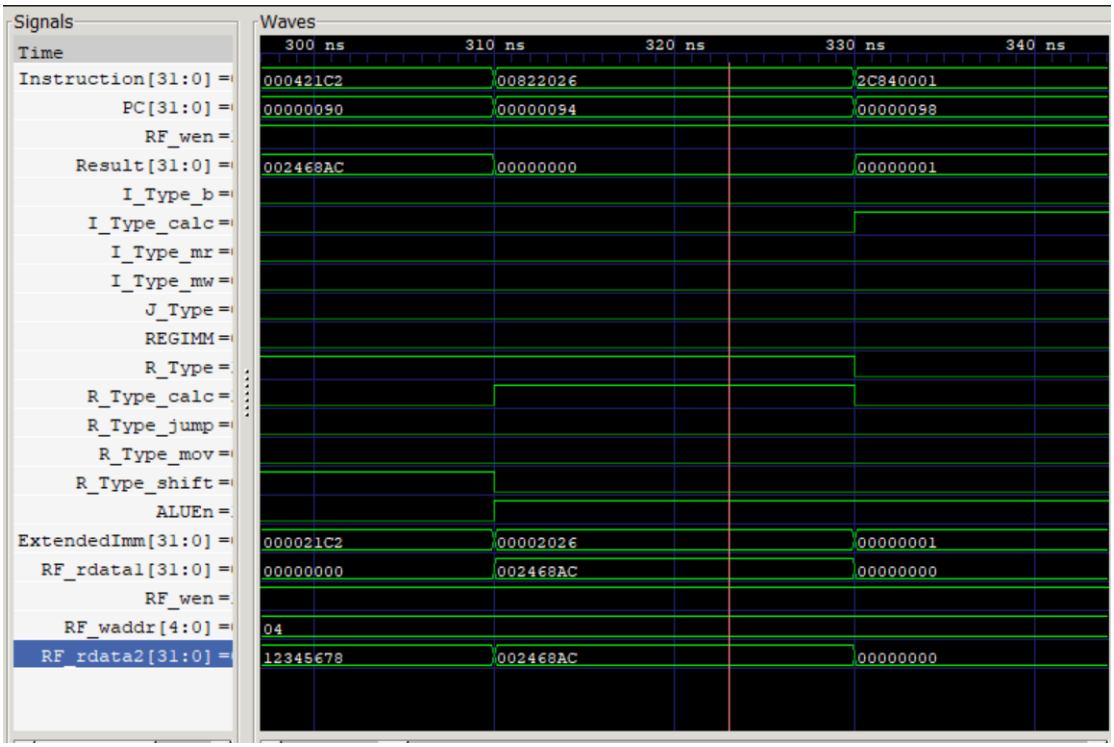


如上述图中，指令编码对应的 j 指令，而目标地址就是 00000004，最后 PC 的值符合，且 J_Type 信号拉高，表明译码正确

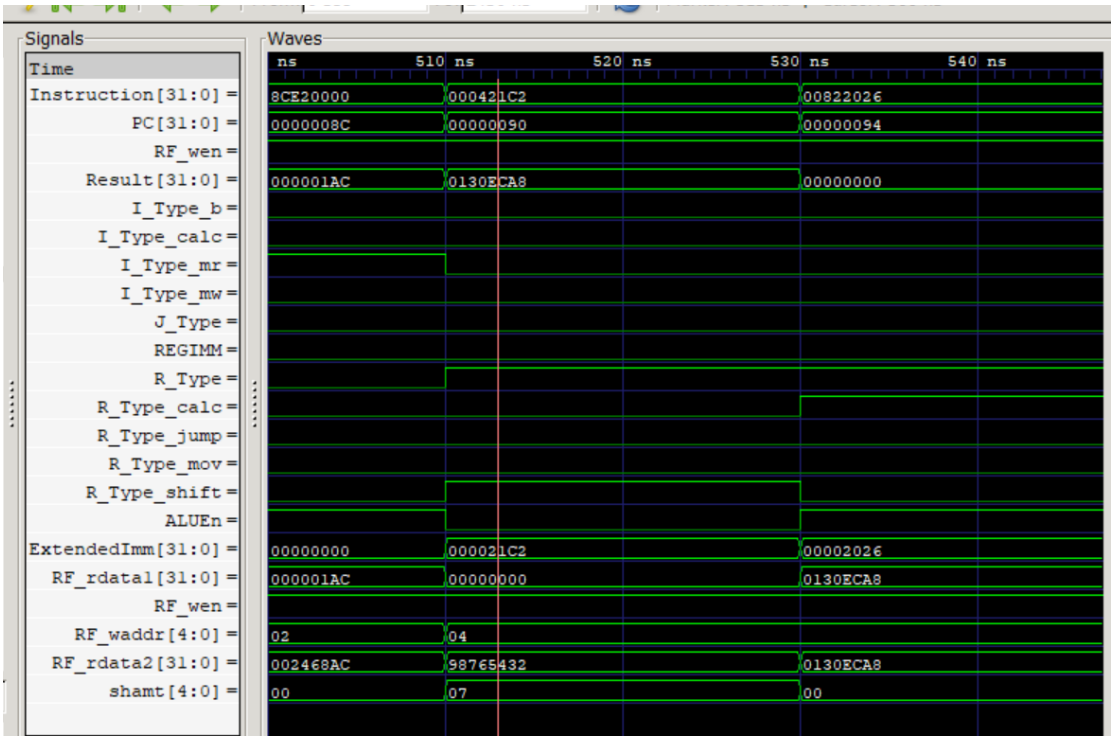


如上述图中，指令编码对应的 addiu 指令，立即数扩展算出来的结果是 FFFFFFE0，寄存器读出的数据是 00004000，最后 Result

的值为 00003FE0，符合，且 I_Type_calc 信号拉高，表明译码正确，然后 RF_wen 拉高，RF_waddr 是 1D，均正确。



如上述图中，指令编码对应的 xor 指令，寄存器读出的数据均是 002468AC，最后 Result 的值为 00000000，符合，且 I_Type_calc 信号拉高，表明译码正确，然后 RF_wen 拉高，RF_waddr 是 04，均正确。



如上述图中，指令编码对应的 srl 指令，寄存器读出的数据为 98765432，shamt 数据为 07，最后 Result 的值为 0130ECA8，符合，且 R_Type_shift 信号拉高，表明译码正确，然后 RF_wen 拉高，RF_waddr 是 04，均正确。

2. 多周期处理器

多周期的设计与单周期很多部分是相似的，下面主要展示不同的地方。

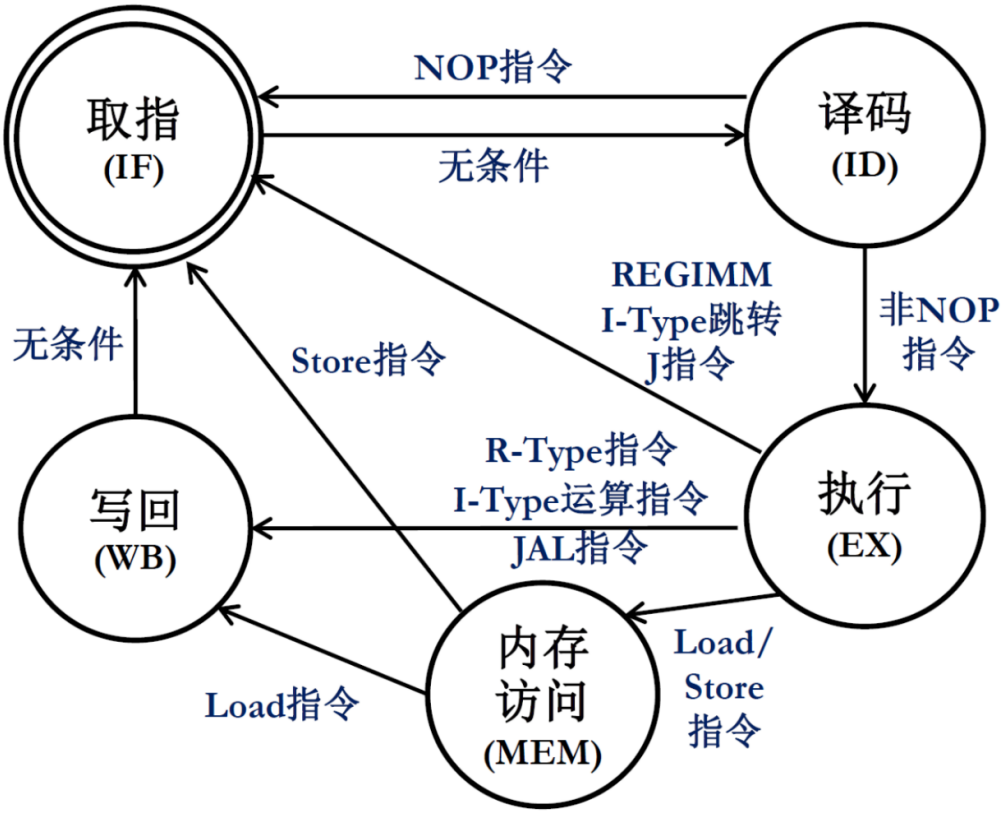
1） 关键 RTL 代码段：

a） 状态转移

```
reg [4:0] current_state;
reg [4:0] next_state;

localparam IF   = 5'b00001;
localparam ID   = 5'b00010;
localparam EX   = 5'b00100;
localparam MEM  = 5'b01000;
localparam WB   = 5'b10000;
```

```
always @(*) begin
        case (current_state)
                IF: next_state <= ID;
                ID: begin
                        if(|reg_Instruction) begin
                                next_state <= EX;
                        end else begin
                                next_state <= IF;
                        end
                end
                EX: begin
                        if(R_Type | I_Type_calc | J_Type & opcode[0]) begin
                                next_state <= WB;
                        end else if(I_Type_mr | I_Type_mw) begin
                                next_state <= MEM;
                        end else begin
                                next_state <= IF;
                        end
                end
                MEM: begin
                        if(I_Type_mr) begin
                                next_state <= WB;
                        end else begin
                                next_state <= IF;
                        end
                end
                WB: next_state <= IF;
                default: next_state <= ID;
        endcase
end
```

这里的转移与 PPT 上给的状态转移图相一致，如下：




```
//状态转移状态机
always @ (posedge clk) begin
        if(rst) begin
                current_state <= IF;
        end else begin
                current_state <= next_state;
        end
end
```

始终上沿到来时，如果复位信号有效，则当前状态更新为取值状态，否则当前状态更新为下一状态。

b） PC 更新的变化（复用 ALU）

```
reg [31:0] PC;
reg [31:0] PC_normal;

always @(posedge clk) begin
        if (current_state == ID) begin
                PC_normal <= PC;
        end
end

always @(posedge clk) begin
        if (rst) begin
                PC <= 32'd0;
        end
        else if (current_state == IF) begin
                PC <= ALUResult;
        end
        else if (current_state == EX) begin
                if (R_Type_jump) begin
                        PC <= RF_rdata1;
                end
                else if (J_Type) begin
                        PC <= {PC_normal[31:28], reg_Instruction[25:0], 2'b00};
                end
                else if ((Zero ^ (REGIMM & ~reg_Instruction[16] |
                             I_Type_b & (opcode[0] ^ (opcode[1] & |RF_rdata1)))) & Branch) begin //branch
                        PC = Result; // ID阶段算出的PC
                end
        end
end
```

首先 ALU 的更新需要根据当前状态来进行了。而且 PC+4 的计算使用了 ALU，并在取指阶段进行更新，而 branch 指令的目标地址在译码阶段就计算出来并存在寄存器中，在执行阶段进行 branch 指令和 jump 指令的目标地址的更新。这样保证了除了 ALU 外，没有其它的加法器。

c) ALU 的变化（用于计算 PC 值）

```
assign ALUop = (current_state == EX) ?
                ({(3){R_Type_calc}} & {func[1] & ~(func[3] & func[0]), ~func[2], func[3] & ~func[2] & func[1] | func[2] & func[0]} |
                {(3){I_Type_calc}} & {opcode[1] & ~(opcode[3] & opcode[0]), ~opcode[2],opcode[3] & ~opcode[2] & opcode[1] | opcode[2] & opcode[0]}
                {(3){REGIMM}} |
                {(3){I_Type_b}}  & {2'b11, opcode[1]} | // slt 111 sub 110
                {(3){I_Type_mr | I_Type_mw}} & 3'b010)
                : 3'b010;
```

```
assign ALU_A = {(32){(current_state == IF) | (current_state == ID)}} & PC |
               {(32){current_state == EX}} & RF_rdata1;
assign ALU_B = {(32){(current_state == IF)}} & {29'b0, 3'b100} |
               {(32){current_state == ID}} & {{{(14){reg_Instruction[15]}}, reg_Instruction[15:0]}, 2'b00} |
               {(32){current_state == EX}} & (ALU_IM ? ExtendedImm : REGIMM ? 32'b0 :RF_rdata2);

//ALU
alu ALU(
        .A              (ALU_A),
        .B              (ALU_B),
        .ALUop          (ALUop),
        .Overflow       (Overflow),
        .CarryOut       (CarryOut),
        .Zero           (Zero),
        .Result         (ALUResult)
);
```

对于 ALUop，执行阶段的编码逻辑与单周期相同，但取指和译码阶段，均是 010（加法），对于 ALU 的端口 A 和 B，执行阶段与单周期相同，但取指阶段，A 为 PC，B 为 4（计算 PC+4），译码阶段，A 为 PC，B 为 offset，计算（PC+offset，即分支指令的目标地址）。

d) 增加的寄存器

```
reg [31:0] reg_Instruction;

always @(posedge clk) begin
        if (current_state == IF) begin
                reg_Instruction <= Instruction;
        end
end
```

```
reg [31:0] PC_normal;

always @(posedge clk) begin
        if (current_state == ID) begin
                PC_normal <= PC;
        end
end
```

```
always @(posedge clk) begin
        if (current_state == ID) begin
                SignExtend <= {{(16){reg_Instruction[15]}}, reg_Instruction[15:0]};
                ZeroExtend <= {16'b0, reg_Instruction[15:0]};
        end
end
```

```
always @(posedge clk) begin
        if (current_state == ID) begin
                RF_rdata1 <= wire_RF_rdata1;
                RF_rdata2 <= wire_RF_rdata2;
        end
end
```

```
//Result
always @(posedge clk) begin
        if ((current_state == EX) || (current_state == ID)) begin
                Result <= {(32){ALUEn}} & ALUResult |
                        {(32){ShiftEn}} & ShifterResult; //Choose Result
        end
end
```

```
//WB
reg [31:0] Read_data_reg;
always @(posedge clk) begin
         if (current_state == MEM) begin
                 Read_data_reg <= Read_data;
         end
end
```

主要包括指令的寄存器，寄存器堆数取数寄存器，执行结果寄存器，内存读数寄存器，立即数扩展的寄存器。

e) 控制信号的变化

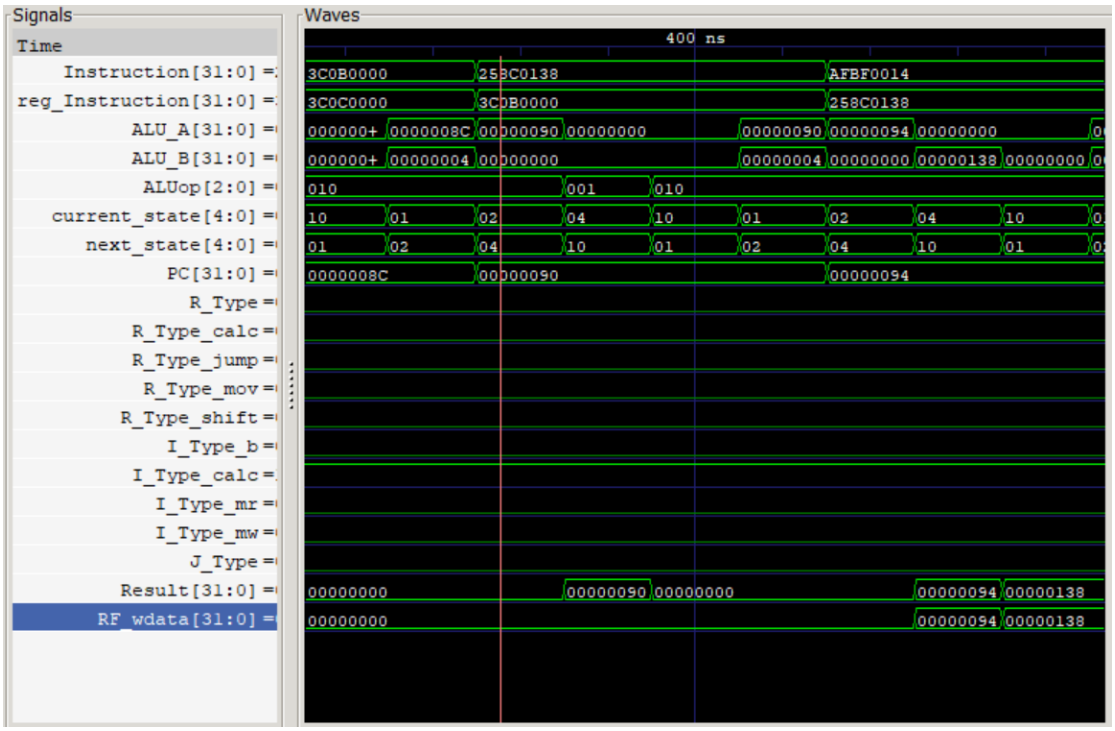
```
wire [5:0] opcode;
wire [4:0] rs;
wire [4:0] rt;
wire [4:0] rd;
wire [4:0] shamt;
wire [5:0] func;
assign {opcode,rs,rt,rd,shamt,func} = reg_Instruction;
```

```
assign RegDst   = R_Type;
assign Branch   = REGIMM | I_Type_b;
assign ALUEn    = R_Type_calc | REGIMM | I_Type_b | I_Type_calc | I_Type_mr | I_Type_mw | (current_state == IF) | (current_state == ID);
assign ShiftEn  = R_Type_shift;
assign ALU_IM   = I_Type_calc | I_Type_mr | I_Type_mw; //立即数计算
assign Shift_IM = ~func[2]; //立即数移位
assign MemWrite = (current_state==MEM) & I_Type_mw;
assign MemRead  = (current_state==MEM) & I_Type_mr;
assign MemtoReg = I_Type_mr;
assign RF_wen   = (current_state==WB) & (R_Type &
                ~(R_Type_mov & (func[0] ^ (|RF_rdata2))) & //move指令条件满足才写
                ~(R_Type_jump & ~func[0])) | //jr指令RF_wen不能位1
                J_Type & opcode[0] | I_Type_calc | I_Type_mr;
```

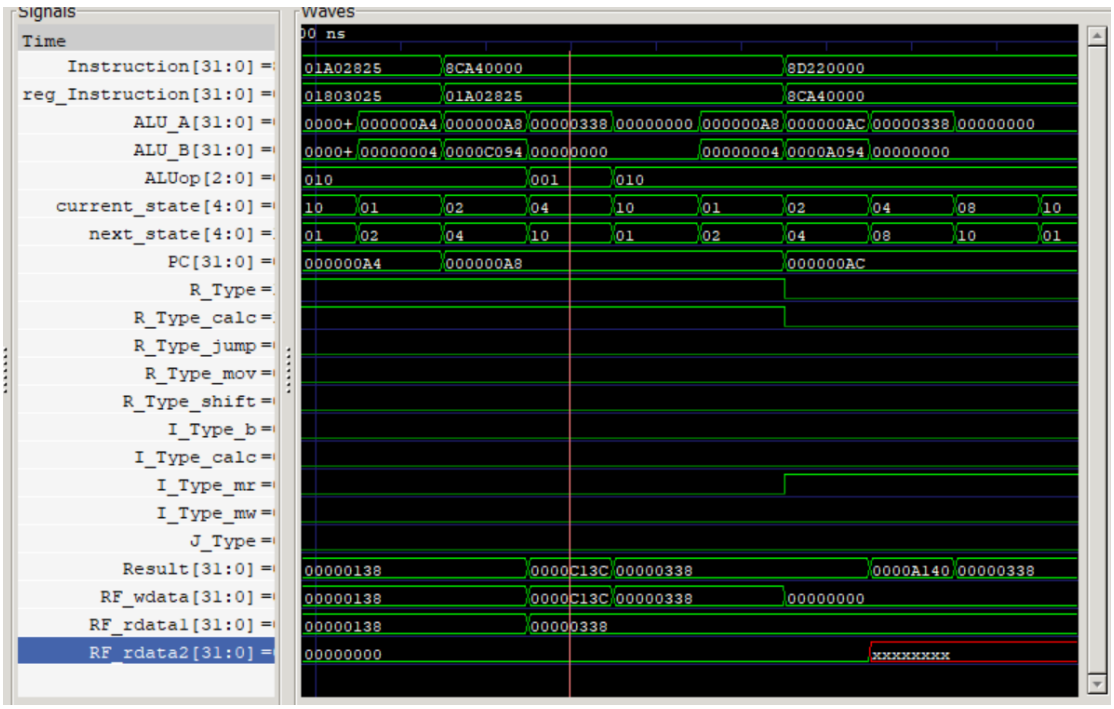
明显改变控制信号有 MemWrite，MemRead，RF_wen，ALUEn。

这几个信号需要对当前状态进行考虑。如 MemWrite 只能在访存阶段拉高，RF_wen 只能在写回阶段拉高，而 ALUEn 需要在取指和译码阶段无条件拉高（因为 PC 更新复用 ALU 的原因）。但同时，其它的译码信号均是根据 reg_Instruction 生成

2） 波形图示例



如上图所示，通过 reg_Instruction 译码后是 lui 指令。有取指，译码，执行，写回阶段，然后取指之后 reg_Instruction 才更新，然后取指，译码 ALUop 的都是 010（加法），然后取指之后，PC 便加 4，取指时，ALU_A 的值是 PC 的值，ALU_B 的值是 4，译码阶段 ALU_A 的值是 PC+4 后的值，满足复用 ALU 要求。然后最后寄存器写数据为 00000000，也符合指令的要求。



Reg_Instruction 译码后是 or 指令。有取指，译码，执行，写回阶段，然后取指之后 reg_Instruction 才更新，然后取指，译码 ALUop 的都是 010（加法），然后取指之后，PC 便加 4，取指时，ALU_A 的值是 PC 的值，ALU_B 的值是 4，译码阶段 ALU_A 的值是 PC+4 后的值，满足复用 ALU 要求。然后执行阶段，从寄存器读出的数据为 00000338 和 00000000 最后寄存器写数据为 00000338，也符合指令的要求。

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） 单周期处理器

本次实验的难点在于对 45 条 MIPS 指令实现过程中，代码的缺失、遗漏和失误，而解决办法只能是对着波形图取寻找错误（本次实验提交次数不下于 45 次），而我认为主要的难点如下：

难点：

1. 指令类型与控制信号的译码逻辑
2. 寄存器堆、ALU 和 Shifter 输入信号的选择（尤其是寄存器堆写数据的选择）
3. PC 的更新
4. 内存的读写（本人认为是实验中最难的部分）

对难点的思考：

1. 指令类型与控制信号的译码逻辑

   首先，之所以把这个列为难点，是因为译码需要对指令集的分类有足够的了解，且要保证译码逻辑的尽可能的简洁，不过好在实验课和理论课的 PPT 给了充分的提示，将主要的指令分类和需要的控制信号都描述的很清晰，所以这部分实现没有想象中的那么麻烦，但这个确实是在本次实验中耗费本人时间比较多的一个点。本人在这次实验中是先根据 PPT 上的提示，译码出指令的类型，并在此基础上对 R_Type 进行了分类，然后再根据译码出来的指令类型来译码控制信号就比较容易了。

2. 寄存器堆、ALU 和 Shifter 输入信号的选择（尤其是寄存器堆写数据的选择）

   这个的难点我认为主要是 ALUop 和寄存器堆的写数据。相比而言 Shifter 比较需要注意的点就是移位的位数需要把寄存器中读取的数据需要对 32 取模，也就是取其低 5 位。而 ALUop

是一个比较麻烦的点，需要兼容所有的需要 ALU 的指令，不过好在实验课 PPT 讲述的也很详细，最后我也是在 PPT 给的描述下进行补充的。

然后寄存器堆写数据的选择是一个非常麻烦的点，首先是从内存读出的数据需要进行一系列操作，如掩码，移位，扩展然后才能写入，且需要对非对齐读指令单独处理。然后通过分类，发现其它指令大部分直接取 Shifter 和 ALU 的结果即可,但是 lui，跳转指令，move 指令需要单独处理，这样就得到了寄存器堆的写数据的选择逻辑，如上面代码所示。

同时我认为本实验中比较坑的一个点就是，并不是所有的 R_Type 指令的 RF_wen 信号需要拉高（虽然实验课 PPT 上是这么写的），比如 jr 指令不能拉高，然后 move 指令条件不满足时也不能拉高。

3. PC 的更新

PC 的更新难点我认为主要是 branch 条件的判断，因为 blez 和 bgtz 是需要判断大于 0 和小于等于 0,不能简单通过 ALU 的 Zero 进行判断（只能判断大于等于和小于）。本人的做法是同时判断寄存器堆读的数据是不是全 0，这样实现比较简洁。其它的方面主要是需要把跳转，分支的具体含义弄清楚就行。

4. 内存的读写（本人认为是实验中最难的部分）

内存的读写本人认为是这次实验最难的部分，也是耗费时间最多的部分。比如由于写入地址需要对齐，让其最低两位为 0，

这样为了保证写数据的正确性，则需要对其进行移位，同时使得Write_strb 的值也跟算出的原始地址低两位有关，而更麻烦的是还需要对 swl 和 swr 进行考虑。

对于 Write_strb，本人主要是对 swl 和 swr 单独处理，且对于不同的分类，对 Write_strb 各位的逻辑通过卡诺图进行的化简，然后再进行合并，最后得到的结果还算是比较简洁。而对于 Write_data 的处理，本人则是对着指令集手册，发现在小尾端下除了 swl 外移位规则相同（指令集手册对 swl 和 swr 给了图进行解释，比较直观）。且其移位量取决于原始地址末两位乘 8，这样就可以通过拼接完成。

而从内存读出的数据，则需要对 lwr 和 lwl 单独处理，而其它指令则需要对读的数据进行符号位扩展。这个操作的实现，本人受到 PPT 的启发，使用了字节掩码的方式进行实现，使得不用继续分类讨论，且代码比较简洁。而 lwr 和 lwl 则需要保存相应寄存器的其它位的值，而这个操作也是通过字节掩码进行实现。

遇到的重要 bug 以及解决过程：

1. Shifter 中算术移位的错误（对>>>了解不够）

解决过程：本人首先的算术右移使用的是 A>>>B，后面发现这样写的结果等价于逻辑右移，然后经过上网查询，发现是因为没有申明 A 是有符号数的原因，所以就改成了 $signed(A)>>>B，但实践证明这种方法也不行，所以最后本人

决定自己写的算术移位逻辑，表达式如下：

((A>>B) | ~((~32'b0)>>B) & {(32){A[31]}})

这种实现方式证明是可以的，不过后面在与他人的讨论后和研究研讨课群里助教发的资料得知：只用声明 wire signed [31:0] A，就可以实现算术移位，也算是进一步了解了 verilog 语法。

2．RF_wen 的信号没有考虑 jr 指令

解决过程：由于 PPT 的误导，本人 RF_wen 最初的译码对于所有的 R_Type 均是拉高的状态，后面发现 move 指令条件不满足和 jr 指令，RF_wen 信号不能拉高。

其它的 bug 主要是因为对指令的理解不够透彻，或对某些指令的考虑有所遗漏而导致的，如跳转指令对小于等于和大于的判断，如访存的读数据，写数据的移位逻辑，然后就是 ALU 操作扩展所导致的一些错误。对于这些错误主要是通过研究波形图与 MIPS 指令集手册，来定位出错地点，然后进行修改，这里就不赘述了。

2） 多周期处理器

多周期处理器的逻辑大部分地方跟单周期相同，主要的是添加几个寄存器，部分译码逻辑的改变和 PC 更新对 ALU 的复用（本人认为的最难点）。具体如下：

难点：

1．需要新添哪些寄存器

2．部分译码逻辑的改变

3．计算 PC 时候对 ALU 的复用（不再用额外的加法器）

对难点的思考：

1．需要新添哪些寄存器

哪些地方需要添加寄存器，总的规则就是如果数据需要保存，则需要添加寄存器，而需要新添的大部分寄存器在理论课也给了，而本人新添了对于立即数有符号和无符号扩展的结果添加了寄存器进行储存，以及对 PC+4 的值用一个寄存器进行保存，以免其发生变化。

综上，新添的寄存器主要包括指令的寄存器，寄存器堆数取数寄存器，执行结果寄存器，内存读数寄存器，立即数扩展的寄存器。

2．部分译码逻辑的改变

不是所有的译码逻辑均需要改变（不过单周期是通过输入的指令直接译码，多周期肯定是通过指令的寄存器的值进行译码）。而通过研究，发现主要有改变控制信号有 MemWrite，MemRead，RF_wen，ALUEn。这几个信号需要对当前状态进行考虑。例如 MemWrite 只能在访存阶段拉高，RF_wen 只能在写回阶段拉高，而 ALUEn 需要在取指和译码阶段无条件拉高（因为 PC 更新复用 ALU 的原因）。

3．计算 PC 时候对 ALU 的复用（不再用额外的加法器）

这个本人认为是多周期处理器的难点，不过理论课上对原理讲的很清楚，但相应的，这样的改变会导致很多地方的变化，这个是本人在这次实验中主要的 bug 来源。最直接的，与 ALU 相关的输入都要发生变化。如 ALUop 和 ALUEn 需要进行对应的调整，ALUEn 改变上一点已经讲了，ALUop 在取指和译码阶段均为 010（加法操作）。然后 ALU 的输入 A，B 也要有相应的变化，取指阶段是 PC 和 4，译码阶段是 PC 和对应的 offset，执行阶段与单周期相同。

然后 PC 的变化也导致一些其它地方逻辑的变化，例如对于 jal，jalr 的写回数据也要改变（因为 PC 会发生变化），而本人的做法是用了一个 PC_normal 寄存器来存 PC+4 的值。再就是计算分支指令目标地址时候的立即数扩展的值也要有相应的改变。以上这些都是本次实验的 bug 主要来源。

遇到的重要 bug 以及解决过程：

多周期处理器遇到的 bug 大多都是在实现对 PC 的复用上，jal，jalr 的写回数据要改变，计算分支指令目标地址时候的立即数扩展的值不能直接用立即数扩展单元算的值，ALUop 和 ALUEn 需要进行对应的调整，ALU 的输入 A，B 也要有相应的变化。这些都在设计时出现了 bug，而调整的过程也是对多周期处理器加深理解的过程，具体的方法在讲难点的部分也都有所提及，这里就不进行赘述了。然后新添的寄存器在什么状态进行更

新也是出错的点。不过这部分看波形图很快就能发现。

三、 对讲义中思考题（如有）的理解和回答

1. ALUop 的编码有什么规律?

   a. 对于 R_Type 指令,使用 func[3:2]就可以确定需要进行的操作（是加减运算、逻辑运算还是比较运算）

   b. 对于 I_Type 指令，使用 opcode[2:1]就可以确定需要进行的操作

   c. 当确定进行的操作后，R_Type 和 I_Type 指令可以根据同一套译码逻辑来生成 ALUop 的编码，这样就可以简化电路逻辑。

2. 表格中的 ALUop 编码是否还有优化空间?

   a. 如果 ALU 实现逻辑，ALUop 与指令编码均不能改变的话，这套 ALUop 的编码生成逻辑已经是最优的了。当然可以把 func[3:0]与 opcode[3:0]作为输入,对 ALUop 的每一位的生成逻辑用卡诺图进行化简。但实践证明，这样做并没有减少电路延迟与逻辑器件的数量，反而让代码的可读性变得更差，得不偿失。

   b. 如果可以修改指令编码，本人认为还是有优化空间的，比如可以把指令的 func[3:1]或 opcode[3:1]表示对于的指令，并用最低位区分使用相同 ALUop 的不同指令（在这个 45 条 MIPS 指令集情况下，相同的 ALUop 对于 R_Type 和 I_Type 只有两种不同的对应指令，例如 addi 和 addiu，sub 和 subu)。这样就可以省去译码的逻辑，直接用 func[3:1]和 opcode[3:1]作为 ALUop，但

是显然，这种实现方法当指令集指令数目增多的时候，就不能满足了，因为可能有很多条的指令对应的 ALUop 相同，这样上述对指令的编码办法就没法实现了。

c. 如果可以修改 ALU 实现逻辑与 ALUop，本人认为也是可以优化的。例如可以对 ALUop 多加一位，然后对每个 ALU 运算类型指令的 func 与 opcode 使用相同的译码逻辑,这样就不用首先判断 ALU 运算类型了

3. 用 current_state 作为判断条件或赋值变量，尽量不使用 next_state（why？）

a. 首先同一个 next_state 会对应不同的 current_state，比如当前状态是译码阶段、写回阶段、访存阶段或者执行阶段，下一个状态均可能是取指阶段。这样根据 next_state 判断显然是不可能的。

b. 状态机的更新顺序：在 Verilog 中，状态机的更新顺序是并行的，即同时更新所有状态。因此，在当前时钟周期中，如果 next_state 和 current_state 同时用于状态机的行为描述，会出现多种情况。例如，在某些组合逻辑中，next_state 的值在当前时钟周期末才被更新，但此时 current_state 已经在逻辑中被使用了。因此，使用 current_state 可以保证逻辑在当前时钟周期内得到正确的状态值。

c. 模块化和重用性：使用 current_state 可以使得状态机的行为描述更加模块化，因为它不依赖于任何其他模块或组件。同时，使用 current_state 也可以提高状态机的重用性，因为它只关注当前状态

的行为，而不需要考虑下一个状态的行为。

d. 动态行为描述：在某些情况下，状态机的下一个状态可能会根据输入信号的变化而发生变化。这种情况下，使用 current_state 作为判断条件可以更好地描述状态机的动态行为，因为它可以及时地更新状态。

总之，在 Verilog 中，使用 current_state 作为判断条件或赋值变量可以保证状态机的正确性和可读性，并且可以提高状态机的模块化和重用性。

四、 在课后，你花费了大约___40__小时完成此次实验。

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）

这次实验不是特别复杂，难度适中，资料也特别完善。唯一的问题可能是测试样例跑的时间太长了，例如水仙花数。

同时，PPT 上有一些地方可能会有些误导，比如 PPT 上表示 R_Type 均会让 RF_wen 拉高，但实际上 jr 就不能拉高，但它属于 R_Type，且 move指令条件不满足时也要么RF_wen拉高，要么寄存器写地址为31。希望以后老师能把 PPT 改进一下。

通过这次实验,我受益匪浅：一方面，对单周期处理器与多周期处理器有了更深的了解。另一方面，通过这次实验，我进一步熟悉了很多 Verilog 语法上的特性，例如对 verilog 中的算术移位和逻辑移位的实现方式有了进一步的理解，例如对如何对一个多位的数据逐位逻辑运算(例如|RF_rdata1)，例如如何实现字节掩码。

另外，感谢助教在单周期处理器和多周期处理器设计上的启发与帮助。