中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929034　姓名：侯汝垚　专业：计算机科学与技术

实验序号：　2　实验名称：简单功能型处理器设计——基于 MIPS 32 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

1. 单周期 CPU

a) RTL 代码段

```
// split the instruction in to diffrent parts
wire [ 5:0] op, func;
wire [ 4:0] rs, rt, rd, sa;
wire [15:0] imm = {rd, sa, func};
wire [25:0] instr_index = {rs, rt, rd, sa, func};
wire R_type, J_type, REGIMM;
assign {op, rs, rt, rd, sa, func} = Instruction;
```

将指令分成多个字段方便分析。同时声明 R_type，J_type 和 REGIMM 信号方便判断指令类型。

```
// PC module ***********************************************************
wire [31:0] PC_plus = PC + 4;

// the PC result of branch instruction
wire [31:0] PC_branch = PC_plus + {{14{imm[15]}}, imm, 2'b00};;
wire branchValid = (REGIMM             & (Zero ^ ~rt[0])
                  | (~op[1] && ~REGIMM) & (Zero ^ op[0])
                  | op[1]               & (Zero & (|RF_rdata1) ^ ~op[0])) & branch ;

// the PC result of J-type instruction or jr and jalr
wire [31:0] PC_jmp = {32{ J_type}} & {PC[31:28], instr_index, 2'b00}
                   | {32{~J_type}} & RF_rdata1;
// select the next PC
wire [31:0] PC_next = {32{branchValid}}          & PC_branch
                    | {32{jmp}}                  & PC_jmp
                    | {32{~jmp && ~branchValid}} & PC_plus;
PC_mod my_PC(
        .clk(clk),
        .rst(rst),
        .PC_next(PC_next),
        .PC(PC)
);
```

```
module PC_mod(
        input clk,
        input rst,
        input [31:0] PC_next,
        output [31:0] PC
);

        reg [31:0] PC_reg;

        assign PC = PC_reg;
        always @(posedge clk) begin
                if (rst) begin
                        PC_reg <= 32'b0;
                end
                else begin
                        PC_reg <= PC_next;
                end
        end

endmodule
```

PC_next 有三种情况，正常加四、分支和跳转，分别计算三种情况，然后根据信号进行选择。PC_mod 即为 PC 寄存器。

```
// CU module ************************************************************
// generate this control signals and some signals about the type of instruction
wire RegDst, ALUSrc, RegWrite, MemtoReg, branch, lui, ImmDst;
wire [2:0] ALUcontrol, ALUop;
CU my_CU(
        .J_type(J_type),
        .R_type(R_type),
        .REGIMM(REGIMM),
        .op(op),
        .RegDst(RegDst),
        .ALUSrc(ALUSrc),
        .MemtoReg(MemtoReg),
        .branch(branch),
        .RegWrite(RegWrite),
        .MemRead(MemRead),
        .MemWrite(MemWrite),
        .ALUop(ALUop),
        .link(link),
        .lui(lui),
        .ImmDst(ImmDst)
);

// mov
wire mov;
wire movValid = ((|RF_rdata2) ^ ~func[0]) && mov;
assign mov = R_type && (func[5:1] == 5'b00101);
// link
wire link = J_type && op[0] || R_type && (func == 6'b001001);
// jmp
wire jmp = J_type || R_type && (func[5:1] == 5'b00100);
```

```
module CU(
        input [5:0] op,
        output RegDst, ALUSrc, MemtoReg, branch, RegWrite, MemRead, MemWrite,
        output link, lui, ImmDst,
        output J_type, R_type, REGIMM,
        output [2:0] ALUop0
);

        wire I_type_branch = (op[5:2] == 4'b0001);

        assign J_type = (op[5:1] == 5'b00001);
        assign R_type = ~(|op);
        assign REGIMM = (op == 6'b000001);

        assign lui = (op == 6'b001111);
        assign RegDst   = R_type;
        assign ImmDst =  ALUSrc;
        assign ALUSrc   = op[5] | op[3];
        assign MemtoReg = op[5];
        assign RegWrite = (op[5] ^ op[3]) || R_type || (J_type && op[0]);
        assign branch   = I_type_branch | REGIMM;
        assign MemRead  = op[5] & ~op[3];
        assign MemWrite = op[5] &  op[3];

        // decode to generate ALUop0
        // 110表示Rtype运算类型 111表示branch(slt) 010表示l和s(add)
        // 010表示立即数加 000表示立即数与 001表示立即数或 100表示立即数异或
        // 111表示立即数比较 011表示无符号立即数比较
        // 即只要不是110就按给定的ALUop输出 否则需要结合funct进一步译码
        assign ALUop0[2] = ~op[5] && (~op[3] || op[1] && ~op[0]);
        assign ALUop0[1] = op[5] || (op[3] ^ op[2]) || REGIMM;
        assign ALUop0[0] = R_type || REGIMM || ~op[5] && (~op[2] && op[1] || ~op[3] && op[2] && op[1] || op[3] && op[2] && op[0]);

endmodule
```

```
module ALU_controler(
        input [2:0] ALUop0,
        input [5:0] func,
        output [2:0] ALUOp
);
        wire R_type = (ALUop0 == 3'b101);
        wire [2:0] funcOP;
        assign funcOP[2] =  func[1] && (~func[3] || ~func[0]);
        assign funcOP[1] = ~func[2] && (~func[3] || func[1]);
        assign funcOP[0] =  func[2] && func[0] || func[3] && func[1];

        assign ALUOp = {3{ R_type}} & funcOP
                     | {3{~R_type}} & ALUop0;

endmodule
```

	1	0
RegDst	输出写到rd寄存器	无
ImmDst	输出写到rt寄存器	无
ALUSrc	ALU第二个操作数为imm	ALU第二个操作数为RF_rdata2
MemtoReg	写数据来自内存	写数据来自ALU
RegWrite	寄存器堆写使能（需结合功能码进一步判断）	无
branch	分支（I型分支或者REGIMM）	无
MemRead	内存读	无
MemWrite	内存读	无
lui	是lui指令	无

CU 模块根据 opcode 产生大部分控制信号，和 func 功能码相关的控制信号在 CU 外部定义。各信号定义如上图，CU 产生的 ALUOp 的第一级译码结果 ALUOp0，ALUcontrol 模块根据 func 和 ALUOp0 进一步翻译出真正的 ALUOp。

```
// judge if we should select ALU_result to write in register
wire ALUValid = (R_type && func[5] || ~op[5] && op[3]) && ~lui;
wire CarryOut, Overflow, Zero;
wire [31:0] ALU_A, ALU_B, ALU_result;

assign ALU_A = RF_rdata1;
assign ALU_B = {32{~ALUSrc}} & RF_rdata2
             | {32{ ALUSrc && ~op[2]}} & {{16{imm[15]}}, imm}
             | {32{ ALUSrc &&  op[2]}} & {16'b0, imm};

alu my_ALU(
        .ALUop0(ALUOp),
        .A(ALU_A),
        .B(ALU_B),
        .Result(ALU_result),
        .Zero(Zero),
        .CarryOut(CarryOut),
        .Overflow(Overflow)
);

// shifter module ******************************************************************
wire [31:0] Shfter_result, Shfter_A;
wire [ 1:0] Shfter_op = func[1:0];
wire [ 4:0] Shfter_B = {5{ saValid}} & sa
                     | {5{~saValid}} & ALU_A[4:0];
wire ShfterValid = R_type & ~func[5] & ~func[3] && ~lui;
wire saValid = ~func[2];
assign Shfter_A = ALU_B;

shifter my_shifter(
        .A(Shfter_A),
        .B(Shfter_B),
        .Result(Shfter_result),
        .Shiftop(Shfter_op)
);
```

ALU 和 shifter 模块类似，定义 Valid 信号表示结果有效，根据信号选择操作数，ALU 的第二个操作时可能是 RF_rdata2 或者立即数的符号拓展和零拓展，shifter 的第二个操作数可能来自 sa 字段或者寄存器。

```
// RegFile module ********************************************************************
wire [31:0] Result, RF_rdata1, RF_rdata2;
wire [4:0] RF_raddr1 = rs;
wire [4:0] RF_raddr2 = {5{ REGIMM}} & 5'b0
                     | {5{~REGIMM}} & rt;

assign RF_waddr = {5{RegDst && ~mov || movValid}} & rd
                | {5{ImmDst}}                     & rt
                | {5{J_type && op[0]}}            & {5'd31};

assign RF_wdata = {32{ MemtoReg}} & MemRdata
                | {32{~MemtoReg}} & Result;

assign Result = {32{ShfterValid}} & Shfter_result
              | {32{link}}        & PC + 8
              | {32{lui}}         & {imm, 16'b0}
              | {32{movValid}}    & RF_rdata1
              | {32{mov}}         & 32'b0
              | {32{ALUValid}}    & ALU_result;

wire jr = R_type && (func == 6'b001000);
assign RF_wen = RegWrite && ~jr && ~mov || movValid;
reg_file my_rf(
        .clk(clk),
        .raddr1(RF_raddr1),
        .raddr2(RF_raddr2),
        .wen(RF_wen),
        .waddr(RF_waddr),
        .wdata(RF_wdata),
        .rdata1(RF_rdata1),
        .rdata2(RF_rdata2)
);
```

RegFile 第一个读地址永远是 rs，而第二个读地址只有在 REGIMM 类型指令时设置为 0，其余情况为 rt， 根据 CU 的译码结果，选择 RegFile 的写地址、写使能和写数据，其中 MemRdata 是内存读后正确处理的数据，而 Result 则是非访存指令时得到的数据，除了可能来自 ALU 和 Shifter 外，如果是 jal 和 jalr 指令则为 PC+8，如果 mov 类指令并且有效则为 RF_rdata1，lui 指令也要特殊处理。RF_wen 结合 RegWrite 外还要考虑 jr 指令不需要寄存器写，还有当 mov 信号条件不满足时不能写。

```
// load
wire [3:0] offset;
assign Address = {ALU_result[31:2], 2'b0};
assign offset[0] = (ALU_result[1:0] == 2'b00);
assign offset[1] = (ALU_result[1:0] == 2'b01);
assign offset[2] = (ALU_result[1:0] == 2'b10);
assign offset[3] = (ALU_result[1:0] == 2'b11);

// generate mask by op[1:0]
wire [31:0] mask, validRdata;
wire wr = (op[2:1] == 2'b11);
wire wl = op[2:0] == 3'b010;
wire ailgn = ~(op[1:0] == 2'b10);
assign mask[ 7: 0] =  8'hFF;
assign mask[15: 8] =  8'hFF   & { 8{op[0]}};
assign mask[31:16] = 16'hFFFF & {16{op[1]}};

wire [31:0] MemRdata, extenMemRdata, lwr_data, lwl_data;
assign validRdata = lwr_data & mask;

// get RdataSignal by op[0]
wire [31:0] RdataSignal = {32{~op[0]}} & {{24{validRdata[ 7]}},  8'b0}
                        | {32{ op[0]}} & {{16{validRdata[15]}}, 16'b0};

assign extenMemRdata = {32{op[1] || op[2]}}   &  validRdata
                     | {32{~op[1] && ~op[2]}} & (validRdata | RdataSignal);

assign lwl_data = {32{offset[0]}} & {Read_data[ 7: 0],RF_rdata2[23:0]}
                | {32{offset[1]}} & {Read_data[15: 0],RF_rdata2[15:0]}
                | {32{offset[2]}} & {Read_data[23: 0],RF_rdata2[ 7:0]}
                | {32{offset[3]}} & {Read_data[31: 0]};

assign lwr_data = {32{offset[0]}} & {Read_data}
                | {32{offset[1]}} & {RF_rdata2[31:24], Read_data[31: 8]}
                | {32{offset[2]}} & {RF_rdata2[31:16], Read_data[31:16]}
                | {32{offset[3]}} & {RF_rdata2[31: 8], Read_data[31:24]};

assign MemRdata = {32{wl  }} & lwl_data
                | {32{wr  }} & lwr_data
                | {32{ailgn}} & extenMemRdata;
```

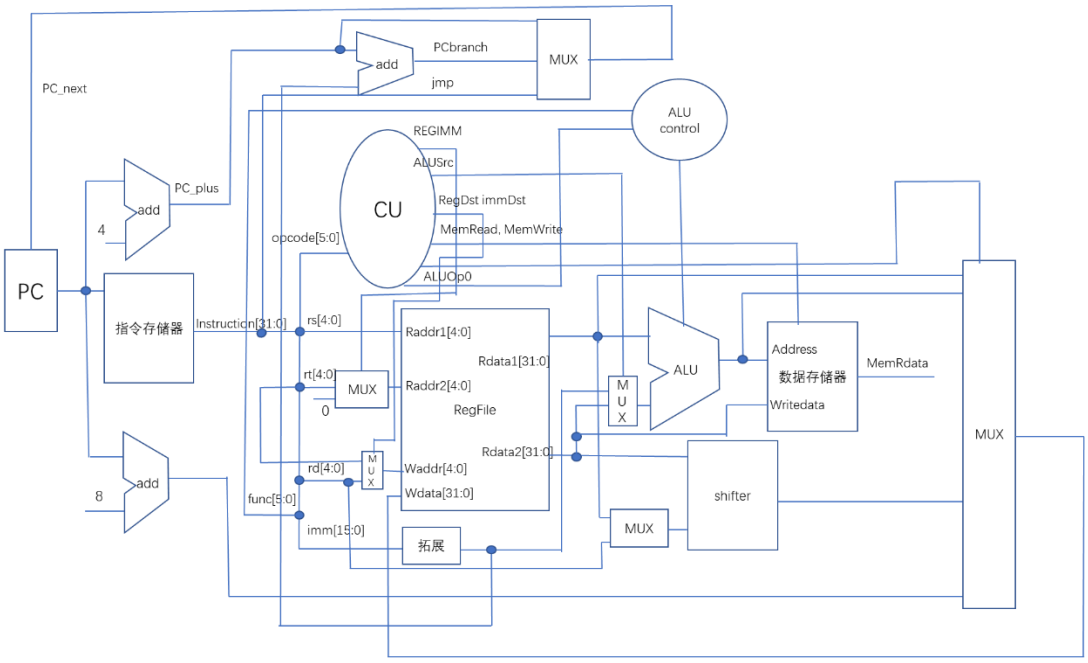
根据访存地址后两位可以定义 offset 信号判断偏移位数，lwl 和 lwr 指令单独处理，对齐情况下读取数据的排列与 lwr 对应的有效数据相同，只需要产生掩码 mask 将无效数据掩盖然后选择符号拓展或者零拓展即可。

```
// Store
wire [31:0] validWdata, swl_data;
wire [3:0] WriteWidth, Write_strb_l, Write_strb_r, Write_strb_ailgn;
assign WriteWidth[0] = 1'b1;
assign WriteWidth[1] = op[0] || op[1];
assign WriteWidth[3:2] = {2{op[1]}} & 2'b11;

assign Write_strb_ailgn = {4{offset[0]}} & {WriteWidth}
                        | {4{offset[1]}} & {WriteWidth[2:0], 1'b0}
                        | {4{offset[2]}} & {WriteWidth[1:0], 2'b0}
                        | {4{offset[3]}} & {WriteWidth[0]  , 3'b0};

assign Write_strb_l[0] = 1'b1;
assign Write_strb_l[1] = ALU_result[0] || ALU_result[1];
assign Write_strb_l[2] = ALU_result[1];
assign Write_strb_l[3] = ALU_result[0] && ALU_result[1];

assign Write_strb_r = {Write_strb_l[0], ~Write_strb_l[3], ~Write_strb_l[2], ~Write_strb_l[1]};

assign Write_strb = {4{wl   }} & Write_strb_l
                  | {4{wr   }} & Write_strb_r
                  | {4{ailgn}} & Write_strb_ailgn;

assign validWdata = {32{offset[0]}} & {RF_rdata2}
                  | {32{offset[1]}} & {RF_rdata2[23: 0],  8'b0}
                  | {32{offset[2]}} & {RF_rdata2[15: 0], 16'b0}
                  | {32{offset[3]}} & {RF_rdata2[ 7: 0], 24'b0};

assign swl_data = {32{offset[0]}} & {24'b0, RF_rdata2[31:24]}
                | {32{offset[1]}} & {16'b0, RF_rdata2[31:16]}
                | {32{offset[2]}} & { 8'b0, RF_rdata2[31: 8]}
                | {32{offset[3]}} & RF_rdata2;

assign Write_data = {32{ wl}} & swl_data
                  | {32{~wl}} & validWdata;
```

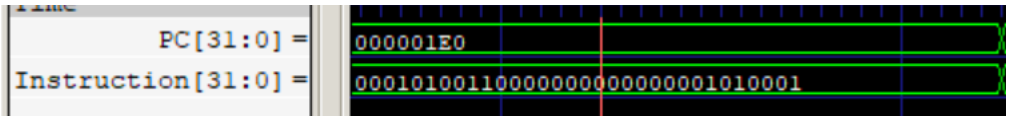
Store 指令同样将对齐和非对齐情况分开考虑，对于 lwl，列出 Write_str 和 ALU_result[1:0]（访存地址后两位）真值表后可以轻松写出 Write_str，lwr 情况的 Write_str 只需要根据 lwl 情况略做修改。对齐情况的 Write_str 则需要结合偏移和位宽才能得到。同样对齐情况的写数据和 swr 相同，只是 Write_str 不同。

b) 逻辑电路图

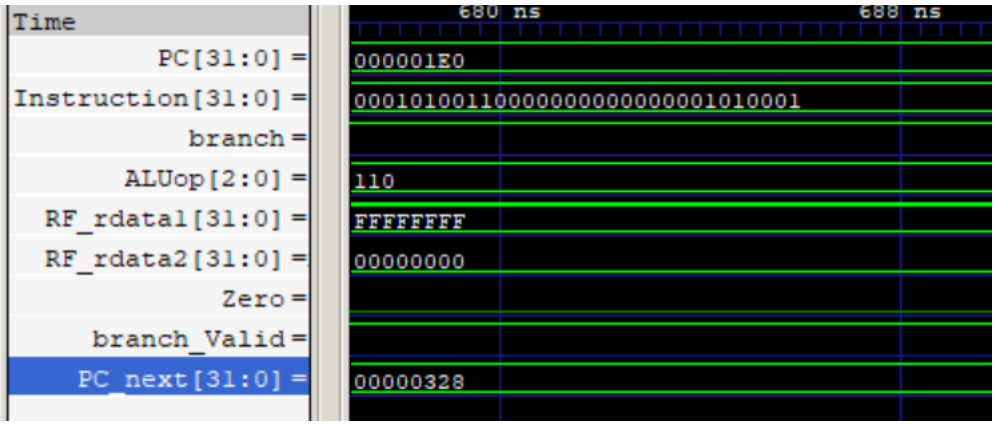
部分控制信号及电路较复杂，因此省略。



c) 波形图示例



以该条指令 00010100110000000000000000101001 为例，由 000101 知为 bne 指令，应拉高 branch 信号，ALUop 为 110（减法），rs 为 00110，rt 为 0，并且需要根据 Zero 信号看 rdata1 和 rdata2 是否相等。



显然两者不等，因此 Zero 信号为 0，分支有效，branchValid 拉高，PC_next 为 PC 加上 16 位立即数的符号拓展。

2. 多周期 CPU

a) RTL 代码段

```
// IR
reg [31:0] IR;
always @(posedge clk) begin
        if (IRWrite) begin
                IR <= Instruction;
        end
        else begin
                IR <= IR;
        end
end

// MDR
reg [31:0] MDR;
always @(posedge clk) begin
        MDR <= MemRdata;
end
```

```
reg [31:0] ALUOut;
always @(posedge clk) begin
        ALUOut <= ALU_result;
end
```

```
    reg [31:0] ShifterOut;
    always @(posedge clk) begin
            ShifterOut <= Shifter_result;
    end
```

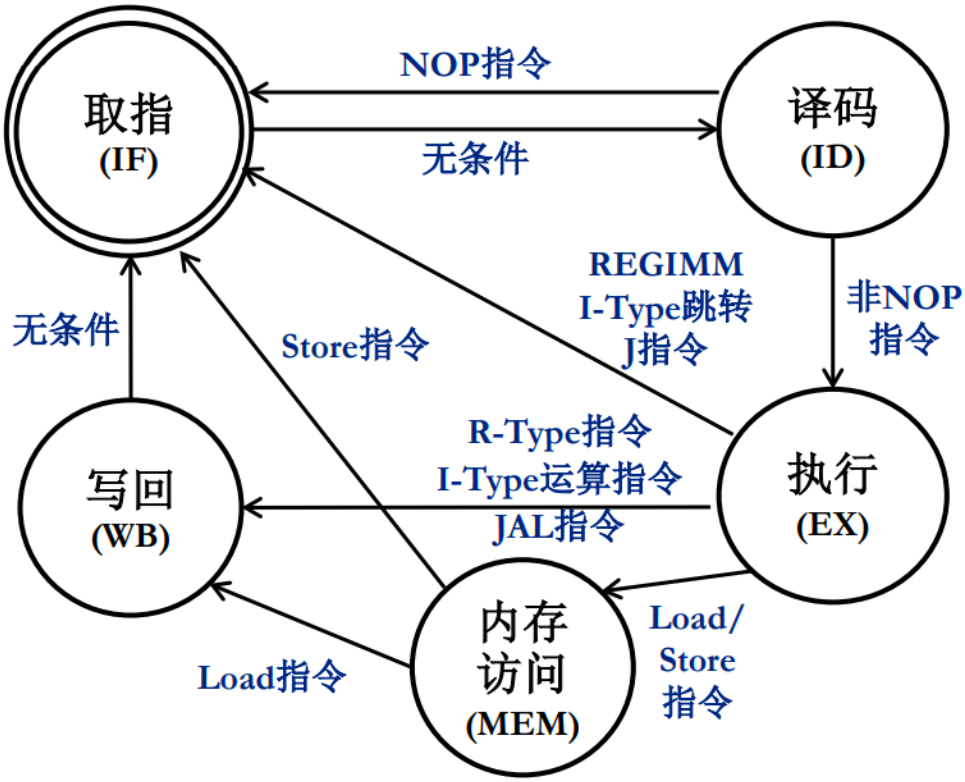
```
reg [31:0] A, B;
always @(posedge clk) begin
        A <= RF_rdata1;
        B <= RF_rdata2;
end
```

对于多周期 CPU，因为该周期产生的信号可能需要下个周期使用，因此添加了 IR、MDR、A、B、ALUOut 和 ShifterOut 寄存器，并将单周期 CPU 对于的变量替换为寄存器变量。

```
// FSM ***************************************************************
reg [4:0] current_state;
reg [4:0] next_state;
localparam IF  = 5'b00001,
           ID  = 5'b00010,
           EX  = 5'b00100,
           MEM = 5'b01000,
           WB  = 5'b10000;

// FSM 1.
always @(posedge clk) begin
        if (~rst)
                current_state <= next_state;
        else
                current_state <= IF;
end
```

多周期变化中的 CU 变为一个状态机，参考 ppt 讲义，分为取指、译码、执行、访存和写回五个状态。采用三段式描述。

```
// FSM 2.
always @(*) begin
        case(current_state)
                IF : next_state = ID;
                ID : begin
                        if (NOP) begin
                                next_state = IF;
                        end
                        else begin
                                next_state = EX;
                        end
                end
                EX : begin
                        if (J_type && ~link || REGIMM || I_type_branch) begin
                                next_state = IF;
                        end
                        else if (op[5]) begin
                                next_state = MEM;
                        end
                        else begin
                                next_state = WB;
                        end
                end
                MEM: begin
                        if (op[3]) begin
                                next_state = IF;
                        end
                        else begin
                                next_state = WB;
                        end
                end
                WB : next_state = IF;
                default: next_state = IF;
        endcase
end
```



第二段描述各个状态的转移规则。

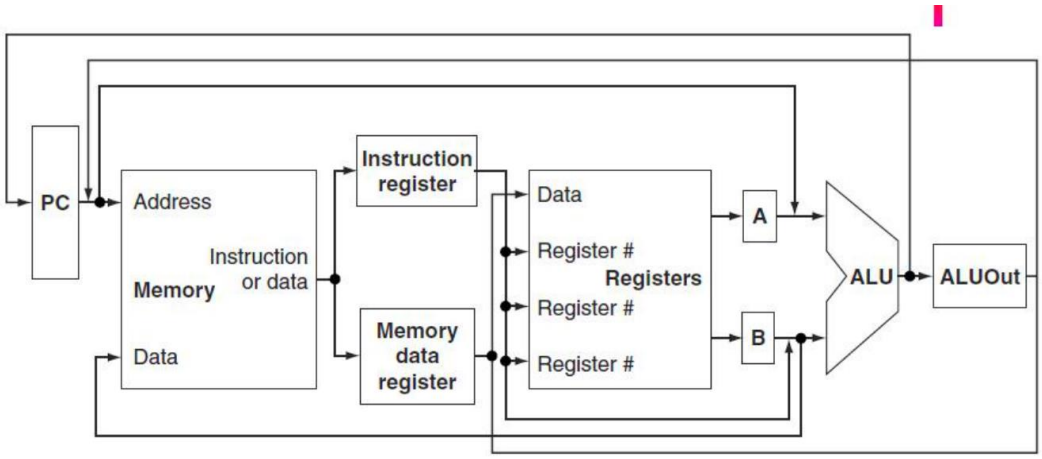
```
// FSM 3.
assign PCWrite     = current_state[0] || current_state[2] && jmp;
assign PCWriteCond = current_state[2] && (REGIMM || I_type_branch);
assign MemRead     = current_state[3] && ~op[3];
assign MemWrite    = current_state[3] &&  op[3];
assign PCsource[1] = current_state[2] && jmp;
assign PCsource[0] = current_state[2] && I_type_branch;
assign ALUSrcB[1]  = current_state[3] || current_state[2] && (op[3] || op[5]) || current_state[1] && branch;
assign ALUSrcB[0]  = current_state[3] || current_state[0] || current_state[2] && link || current_state[1] && branch;
assign ALUSrcA     = (current_state[2] && ~link) || (current_state[1] && ~branch);
assign RegWrite    = current_state[4];
assign IRWrite     = current_state[0];
assign MemtoReg    = current_state[4] && op[5] && ~op[3];
assign RegDst      = current_state[4] && R_type;
assign ImmDst      = current_state[4] && (op[5] || op[3]);

wire [2:0] ALUEx;

assign ALUEx[2] = ~op[5] && (~op[3] || op[1] && ~op[0]);
assign ALUEx[1] = op[5] || (op[3] ^ op[2]) || REGIMM;
assign ALUEx[0] = R_type || REGIMM || ~op[5] && (~op[2] && op[1] || ~op[3] && op[2] && op[1] || op[3] && op[2] && op[0]);

assign ALUop = (current_state[0] || current_state[1] || current_state[2] && link) ? 3'b010
                                                                                  : ALUEx;
```

第三段用组合逻辑描述各控制信号。其中与单周期不同的地方在于添加了 PCWrite 和 IRWrite 控制 PC 和 IR 的写使能，用 PCSource 选择 PC 的值来自 ALUOut、指令还是 ALU_result（正常加 4），用 ALUSrcA 选择 ALU 第一个操作数为 PC 还是 rdata1（A），ALUSrcB

选择第二个 ALU 操作数为 rdata2（B）、4 或者立即数。

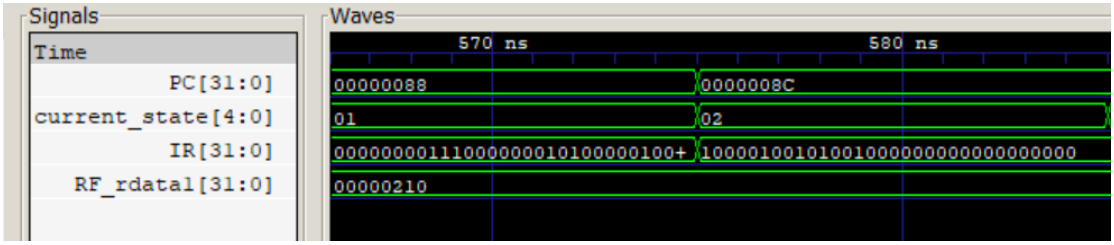
其余部分代码与单周期 CPU 差别不大。

b) 逻辑电路图

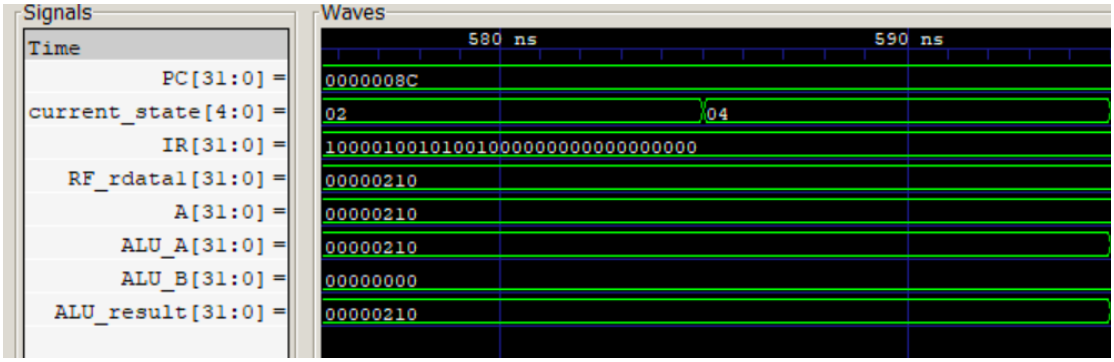


只需采用一个 ALU 即可完成所有计算。

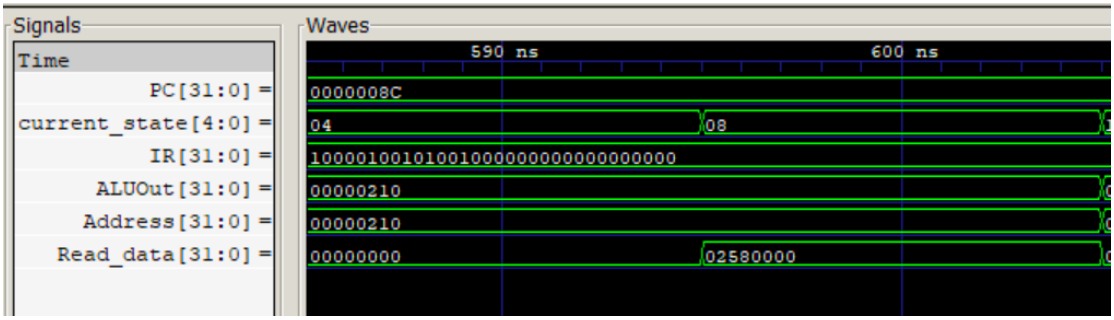
c) 波形图示例



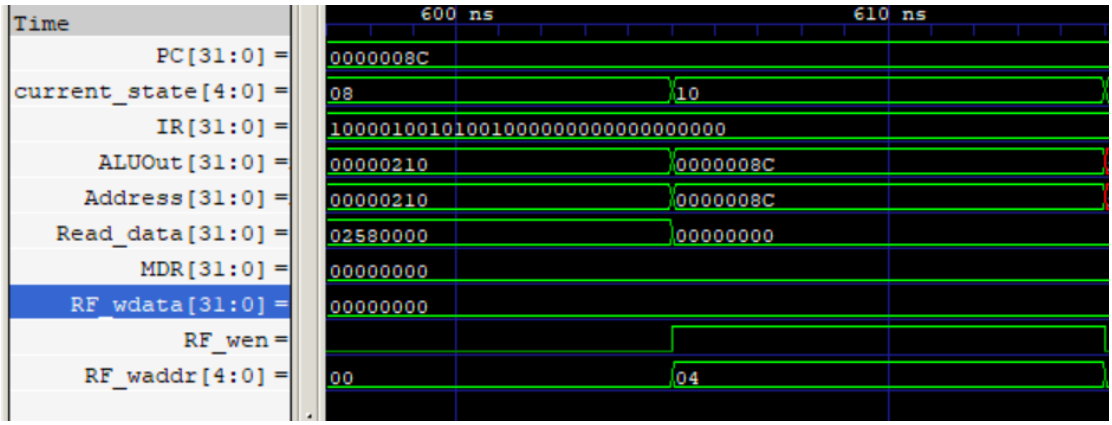
以该过程为例，取指（状态 01）后进入译码状态（02），IR 寄存器更新为 10000100101001000000000000000000，PC 加 4，当前指令为 lh 指令，并得到 rdata1 为 00000210 H。



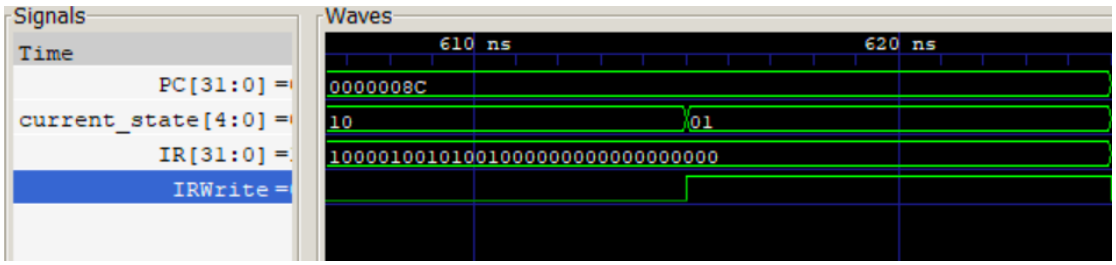
进入执行状态（04），计算出访存地址并写入 ALUOut。



进入访存状态（08），根据 ALUout 获得 Read_data，处理后写入 MDR。



进入写回（10），Read_data 地位半字为 0000，故 MDR 为 00000000，RF_wen 拉高，数据写入对应寄存器。



IRWrite 拉高，获取下一条指令。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

1. 如何做到在一个周期内读和写?

PC 的写和寄存器堆的写都是时序逻辑，只有当 clk 上升沿来的时候才会写入，而读取以及计算是组合逻辑，在 clk 上升沿来之前以及生成正确的结

果，因此可以在一个周期内完成读和写的操作。

2. PC 存放当前指令地址还是下一条指令地址?

理论课之前提到 PC 取指后指向下一条指令，但是对于我们的单周期 CPU，取指令这一步应该可以看作组合逻辑，指令随 PC 的变化而变化，因此在该周期内 PC 应该不能改变，必须指向当前执行指令的地址。

3. 怎么产生各种控制信号?

参考课本，根据指令类型及各类指令对应的所有控制信号的值可以列出真值表，然后根据真值表化简得到各个信号的表达式。因为实验实现的指令比课本上多，因此还需要添加一些其他控制信号。

4. 对访存指令的理解较为困难。

刚开始不清楚访存获取的数据以及需要写入内存的数据到底应该如何处理，读取数据是否已经根据地址偏移? 存储的数据需不需要偏移? 读取的数据是不是低地址存在寄存器低位? 过多次试错才逐渐理解 ppt 上说的意思。读取数据按字节对齐，相当于默认地址低两位为 00，然后需要根据地址以及指令来对读取的数据进行选择。而对于存储则需要根据地址和指令将数据偏移到对应位置并且生成对应的 write_str。

5. 非对齐访存指令 lwl，lwr，swl，swr 比较混乱。

最开始看到这四条指令感到非常晕，不知道靠左对齐到底是寄存器的左还是地址的左，不知道大端小端的区别在哪，也不知道取哪部分数据放在寄存器的哪部分。多次试错后逐渐明白，对于小端来说，向左对齐就是向低地址对齐并将数据放在寄存器高位，向右对齐就是向高地址对齐并将数据放在寄存器地位。

6. 算术移位失效。

在 shifter 模块中使用>>>算术右移无效，依然是逻辑右移，但是假如重新定义信号就可以实现算术右移，查询资料发现 Verilog 中有有符号数和无符号数的区别，对无符号数来说逻辑右移和算术右移等效，猜测可能是连接到模块内的信号被认为是无符号数且无法更改，但是模块内的信号根据上下文被解释为有符号数，因此可以实现算术右移。

7. 为什么 jal 和 jalr 返回地址为 PC + 8 而不是 PC + 4

理论上返回地址应该是当前指令的下一条指令，也就是 PC + 4，但是根据指令手册，存入寄存器的值应该为 PC + 8，查询资料后发现，这是因为紧随 jal 指令的指令位于"分支延迟槽"（delay slot）中，这种技术手段主要用在早期没有分支预测的流水线 RISC 上。

三、 对讲义中思考题（如有）的理解和回答

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？

<table>
<tr><th>R-Type 指令</th><th>func（6-bit）</th><th>ALUop（3-bit）</th><th>ALUop编码</th><th>opcode（6-bit）</th><th>I-Type 指令</th><th>ALUop编码</th></tr>
<tr><td>add</td><td>10 00 00</td><td rowspan="2">010</td><td rowspan="3">ADD/SUB：func[3:2] == 2'b00
ALUop = {func[1], 2'b10}</td><td>001 0 00</td><td>addi</td><td rowspan="2">ADD：opcode[2:1] == 2'b00
ALUop = {opcode[1], 2'b10}</td></tr>
<tr><td>addu</td><td>10 00 01</td><td>001 0 01</td><td>addiu</td></tr>
<tr><td>sub</td><td>10 00 10</td><td rowspan="2">110</td><td></td><td></td><td></td></tr>
<tr><td>subu</td><td>10 00 11</td><td></td><td></td><td></td></tr>
<tr><td>and</td><td>10 01 00</td><td>000</td><td rowspan="4">逻辑运算：func[3:2] == 2'b01
ALUop = {func[1], 1'b0, func[0]}</td><td>001 1 00</td><td>andi</td><td rowspan="3">逻辑运算：opcode[2] == 1'b1
ALUop = {opcode[1], 1'b0, opcode[0]}</td></tr>
<tr><td>or</td><td>10 01 01</td><td>001</td><td>001 1 01</td><td>ori</td></tr>
<tr><td>xor</td><td>10 01 10</td><td>100</td><td>001 1 10</td><td>xori</td></tr>
<tr><td>nor</td><td>10 01 11</td><td>101</td><td>001 1 11</td><td>lui</td><td>非运算类指令，需单独处理</td></tr>
<tr><td>slt</td><td>10 10 10</td><td>111</td><td rowspan="2">比较运算：func[3:2] == 2'b10
ALUop = {~func[0], 2'b11}</td><td>001 0 10</td><td>slti</td><td rowspan="2">比较运算：opcode[2:1] == 2'b01
ALUop = {~opcode[0], 2'b11}</td></tr>
<tr><td>sltu</td><td>10 10 11</td><td>011</td><td>001 0 11</td><td>sltiu</td></tr>
</table>

从图中可以看出，对于加减运算，ALUop 后两位为 10，第一位为 func[1] 或者 opcode[1]，对于逻辑运算，ALUop 中间一位为 0，最高位为 func[1]

或 opcode[1]，最低位为 func[0]或者 opcode[0]，对于比较运算，ALUop 低两位为 11，高位为~func[0]或者~opcode[0]，无论是 R-type 和 I-type，某种运算下的 ALUop 的生成模式对于 func 和 opcode 的格式是一样的，区别只是 R-type 里用 func，I-type 里用 opcode，因此可以用同一套逻辑电路生成 ALUop，用一个选择器选择输入 opcode 或者 func。

表格中 ALUop 的生成方式虽然对于 func 和 opcode 是一样的，但是判断做何种运算的方法不同，同时还要考虑其他指令对于的 ALUop，比如 beq 指令对于 ALLUop 为 110，我参考课本上的多级译码方式，在 CU 中根据 opcode 先翻译出 ALUop0，若 ALUOp0 不为 101（I-type 里没有 nor 指令）则 ALUop 就是 ALUop0，若 ALUOp0 为 101，说明是 R-type，再根据 func 用 ppt 上的方式进一步译码，这样 CU 中只需要输入 opcode 而不需要输入 func，缩小主控制单元的规模。

四、 在课后，你花费了大约 40 小时完成此次实验。

单周期 25 小时，多周期 15 小时

五、 对于此次实验的心得、感受和建议

本次实验相比上一个实验来说难度提升很大，工作量也是成倍增长，刚开始的时候几乎无从下手。由于实验课和理论课进度不一，实验初期无法下手，很多工作都拖到了实验中后期，导致时间不是很充裕。但是随着理论课的深入以及在试错总结的经验，对单周期 CPU 逐渐有了一个清晰的理解，主要设计工作其实在于通过 CU 产生的各种控制信号来对各个信号进行选择。虽然难度大，但是收获也很丰富，当看到自己写的 CPU 成功跑出那么多程序时内心也是非常激动。

我认为访存是该实验最难的部分，在理论课上也并没有对这部分细讲，只能通过一次次试错来理解。希望下一届的课程可以对访存指令执行过程进行更详细的讲解。另外在最后一节课的验收环节助教才演示了如何根据波形图 debug，虽然大部分同学都探索出了如何通过波形图 debug 的技能，但是还是建议在实验中期就把一些用到的方法讲解一下，毕竟这个实验难度还是不小。