中国科学院大学计算机组成原理（研讨课）

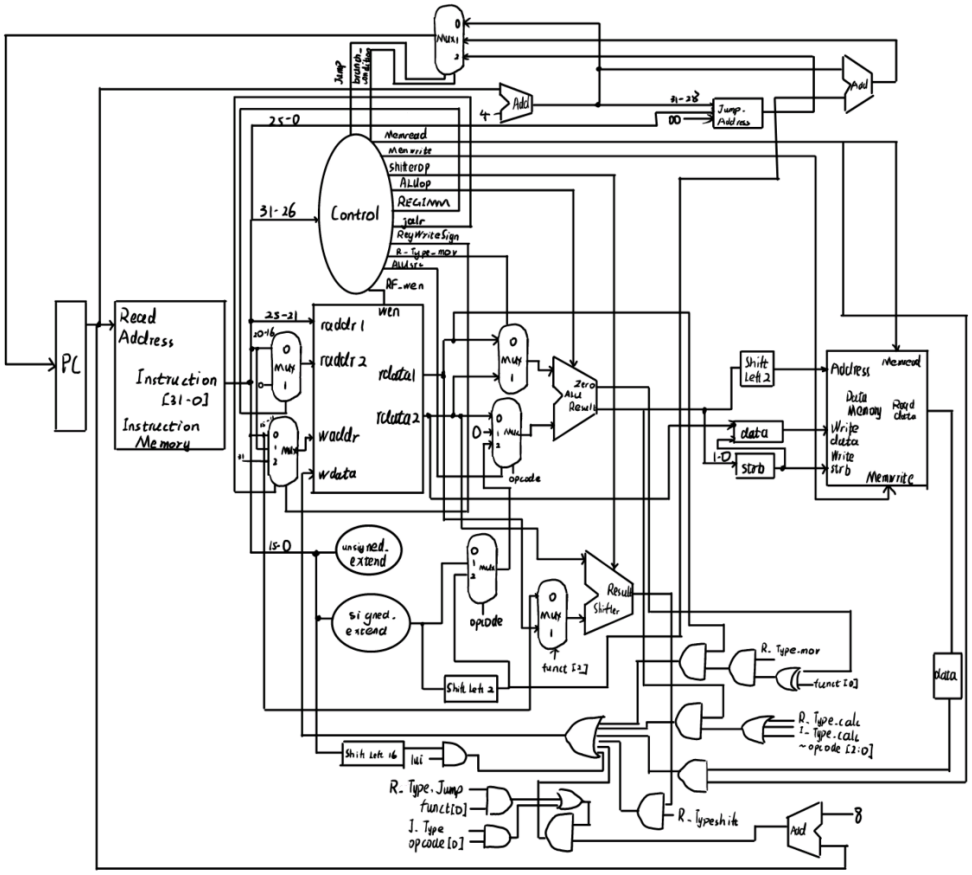
实 验 报 告

学号：2021K8009929016　　姓名：李金明　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

一、 逻辑电路结构及说明

1. 逻辑电路结构图



2. 代码实现部分

a. ALU 代码更改

```
`define ALUOP_XOR 3'b100
`define ALUOP_NOR 3'b101
`define ALU_SLTU 3'b011

wire op_xor = ALUop == `ALUOP_XOR;
wire op_nor = ALUop == `ALUOP_NOR;
wire op_sltu = ALUop == `ALU_SLTU;

wire [`DATA_WIDTH - 1:0]oper_num = {`DATA_WIDTH{op_add}} & B |
{`DATA_WIDTH{op_sub | op_slt | op_sltu}} & (~B);
wire Cin = op_sub | op_slt | op_sltu;

wire [`DATA_WIDTH - 1:0] xor_res = A ^ B;
wire [`DATA_WIDTH - 1:0] nor_res = ~(A | B);

assign Result_tmp = {`DATA_WIDTH{op_and}} & and_res |
                    {`DATA_WIDTH{op_or}} & or_res |
                    {`DATA_WIDTH{op_xor}} & xor_res |
                    {`DATA_WIDTH{op_nor}} & nor_res |
                    {`DATA_WIDTH{op_add | op_sub | op_slt}} & add_res;
assign Result = ~{`DATA_WIDTH{op_slt | op_sltu}} & Result_tmp |
                    {`DATA_WIDTH{op_slt}} & {{`DATA_WIDTH-1{1'b0}},
                   (Overflow ^ Result_tmp[`DATA_WIDTH - 1])} |
                    {`DATA_WIDTH{op_sltu}} & {{`DATA_WIDTH-1{1'b0}},
                   (~CarryOut_init)};
```

在原先 alu 的代码基础上增添实现了异或（xor），或非（nor）和无符号整数比较（sltu）的功能。部分重要更改有：

译码信号：新增 3 个译码信号 xor，nor 和 sltu，分别在对应操作时置为 1。

加法器信号：新增在 sltu 时将第二个操作数置为~B，新增 Cin 在 sltu 时置为 1。

Result 输出：在 sltu 模式下，若出现借位，即 CarryOut 为 1，说明 A<B，此时应输出 0；相反地，若 CarryOut 为 0，应输出 1,。故 sltu 模式下 Result 应为 CarryOut 取反后扩展为 32 位。

b. shifter 代码实现

```
`timescale 10 ns / 1 ns

`define DATA_WIDTH 32

module shifter (
    input  [`DATA_WIDTH - 1:0] A,
    input  [              4:0] B,
    input  [              1:0] Shiftop,
    output [`DATA_WIDTH - 1:0] Result
);

assign Result = {`DATA_WIDTH{~Shiftop[1] & ~Shiftop[0]}} & (A << B) |
                {`DATA_WIDTH{Shiftop[1] & Shiftop[0]}} &
({{`DATA_WIDTH{A[`DATA_WIDTH-1]}}, A} >> B) |
                {`DATA_WIDTH{Shiftop[1] & ~Shiftop[0]}} & (A >> B);

endmodule
```

shifter 功能是在 Shiftop 分别为 00,11,10 时，对 A 逻辑左移、算术右移、逻辑右移 B 位。可以说明的是在算术右移部分，我没有使用“>>>”操作符有关的操作，而是在 A 前面补 32 位 A 的最高位，然后逻辑右移 B 位，最后结果取低 32 位。

以下为单周期 CPU 部分

c. 宏定义

```
`define R_Type_opcode 6'b000000
`define REGIMM_Type_opcode 6'b000001
`define J_Type_opcode 5'b00001
`define I_Type_calc_opcode 3'b001
`define I_Type_branch_opcode 4'b0001
`define I_mem_b_opcode 3'b000
`define I_mem_h_opcode 3'b001
`define I_mem_w_opcode 3'b011
`define I_memr_lbu_opcode 3'b100
`define I_memr_lhu_opcode 3'b101
`define I_mem_wl_opcode 3'b010
`define I_mem_wr_opcode 3'b110
`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 ALU_SUB 3'b110
`define ALU_SLT 3'b111
`define ALU_SLTU 3'b011

    wire R_Type;
    wire R_Type_calc;
    wire R_Type_shift;
    wire R_Type_jump;
    wire R_Type_mov;
    wire REGIMM;
    wire J_Type;
    wire I_Type_branch;
    wire I_Type_calc;
    wire I_Type_memr;
    wire I_Type_memw;
```

这些宏定义及模块中相应的译码信号是为了确定操作指令属于的类别，如 R_Type 类型及其各种划分、REGIMM 类型、J_Type 类型和 I_Type 类型及其各种划分。另外还定义了 ALU 各种操作类型，但实际上并未全部用上。

d. 指令拆解

```
assign rs = Instruction[25:21];
assign rt = Instruction[20:16];
assign rd = Instruction[15:11];
assign funct = Instruction[5:0];
assign shamt = Instruction[10:6];
assign instr_index = Instruction[25:0];
assign imm = Instruction[15:0];
```

这里对指令进行拆解，得到相应的控制信号，对于指令相同部分但含义不同的控制信号，使用其中一个控制信号代替，如 rd 在某些指令中代表寄存器编号，而在 REGIMM 中代表 REG，指代具体是 REGIMM 中的哪条指令。通过拆解和命名，方便了运行时进行操作。

e. 调用 ALU 的部分

```
assign ALUsrc = (I_Type_calc || I_Type_memr || I_Type_memw)? 1:0;
assign ALU_A = (R_Type_mov)? RF_rdata2 : RF_rdata1;
assign ALU_B =({32{R_Type_calc | (I_Type_branch & ~opcode[1])}} &
                    RF_rdata2) |
                (({32{REGIMM | (I_Type_branch & opcode[1]) |
                R_Type_mov}}) & {32{1'b0}}) |
                    ({32{ALUsrc}} & extend);
```

ALU 的第一个操作数仅在 R_Type 指令中的 mov 型指令时是从寄存器堆读出的 RF_rdata2，其余情况均为 RF_data1.

ALU 的第二个操作数在 R_Type 指令中的 cal 型指令和 beq 与 bne（二者需要判断两个操作数是否相等）时为从寄存器堆读出的 RF_rdata2，在 REGIMM 型指令、I_Type 分支指令中的 blez 和 bgtz 指令（opcode 第 1 位为 1）以及 R_Type 指令中的 mov 型指令时为 32 位 0（均需要判断第一个操作数与 0 的关系），在 I_Type 中的计算、内存读和内存写指令中为相应的立即数扩展，具体扩展方式因指令而异，由下面的代码给出：

```
    assign sign_left_extend = (imm[15] == 1'b1)? {{14{1'b1}}, imm[15:0],
2'b00} : {{14{1'b0}}, imm[15:0], 2'b00};
    assign signed_extend = (imm[15] == 1'b1)? {{16{1'b1}}, imm[15:0]} :
{{16{1'b0}}, imm[15:0]};
    assign unsigned_extend = {16'b0, imm[15:0]};
    assign extend = (I_Type_calc && opcode[2])? unsigned_extend :
                    (((I_Type_calc && (opcode[2:1] == 2'b00 ||
opcode[2:1] == 2'b01)) || I_Type_memr || I_Type_memw)? signed_extend :
                    sign_left_extend);
```

在 I_Type 计算指令中的 andi，ori 和 xori 指令（opcode 第 2 位为 1）中，对取到的立即数进行无符号扩展；在 I_Type 计算指令中的 addiu，sltu 和 sltiu 指令（opcode 第 2 位为 0）、内存读和内存写指令中，对取到的立即数进行有符号扩展；在 I_Type 分支指令中，对取到的立即数进行有符号扩展，并逻辑左移两位。

ALUop 的赋值较为复杂，由以下的代码给出：

```
    assign ALUop = (R_Type_calc)? ((funct[3:2] == 2'b00)? {funct[1], 2'b10} :
                        ((funct[3:2] == 2'b01)? {funct[1], 1'b0, funct[0]} :
                        {~funct[0], 2'b11})) :
                   ((I_Type_calc && opcode[2:0] != 3'b111)?
                        ((opcode[2:1] == 2'b00)? {opcode[1], 2'b10} :
                        ((opcode[2])? {opcode[1], 1'b0, opcode[0]} :
                        {~opcode[0], 2'b11})) :
                    ((R_Type_mov || (I_Type_branch && opcode[1] == 1'b0))?
                    `ALU_SUB :
                    ((REGIMM || (I_Type_branch && opcode[1] == 1'b1))?
                    `ALU_SLT :
                    `ALUOP_ADD)));
```

在 R_Type 计算指令和 I_Type 计算指令中，ALUop 编码由下面的表格给出（表格来自课上 PPT）：

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 ALUop = {func[1], 2'b10}	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				
subu	10 00 11					
and	10 01 00	000	逻辑运算：func[3:2] == 2'b01 ALUop = {func[1], 1'b0, func[0]}	001 1 00	andi	逻辑运算：opcode[2] == 1'b1 ALUop = {opcode[1], 1'b0, opcode[0]}
or	10 01 01	001		001 1 01	ori	
xor	10 01 10	100		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 ALUop = {~func[0], 2'b11}	001 0 10	slti	比较运算：opcode[2:1] == 2'b01 ALUop = {~opcode[0], 2'b11}
sltu	10 10 11	011		001 0 11	sltiu	

在 R_Type_mov 指令和 I_Type_branch 中的 beq 和 bne 指令（opcode 第 1 位为 0）中，需要判断两个操作数是否相等，则此时 ALUop 为减；在 REGIMM 和 I_Type_branch 中的 blez 和 bgtz 指令(opcode 第 1 位为 1)中，需要判断所读数与 0 的大小关系，故此时 ALUop 置为有符号比较。

f. Shifter 调用

```
assign Shiftop = ({2{R_Type_shift}}) & funct[1:0] ;
assign Shifter_A = RF_rdata2;
assign Shifter_B = (funct[2])? RF_rdata1 : shamt;
```

Shifter 的调用较为简单，只有 R_Type 移位指令需要调用。在 R_Type 移位指令中 Shiftop 为 funct 的后两位， Shifter_A 始终为 rt 寄存器中读取到的数据。在 sllv，srav 和 srlv 指令（funct 第 2 位为 1）中，Shifter_B 位=为从 rs 寄存器读到数据的低五位，在 sll，sra 和 srl 指令（funct 第 2 位为 0）中，Shifter_B 为从 shamt。

g. PC 值更新

```
    assign PC_4 = PC + 4;
    assign Jump = (J_Type || R_Type_jump)? 1:0;
    assign Jump_Address = ({32{R_Type_jump}} & RF_rdata1) |
                           ({32{J_Type}} & {PC_4[31:28],
instr_index[25:0], 2'b00});
    assign branch = (REGIMM || I_Type_branch)? 1:0;
    assign branch_condition =
    (I_Type_branch) && ((opcode[1:0] == 2'b00) && Zero == 1 ||
                        (opcode[1:0] == 2'b01) && Zero == 0 ||
                        (opcode[1:0] == 2'b10) && (ALU_Result == 32'b1
|| RF_rdata1 == 32'b0) ||
                        (opcode[1:0] == 2'b11) && (ALU_Result == 32'b0
&& ~(RF_rdata1 ==32'b0))) ||
    (REGIMM) && ((rt[0] == 1'b0) && ALU_Result == 32'b1 ||
                  (rt[0] == 1'b1) && ALU_Result == 32'b0);
    alu PC_branch_calc(
        .A(PC_4),
        .B(sign_left_extend),
        .ALUop(`ALUOP_ADD),
        .Result(PC_branch),
        .Overflow(),
        .CarryOut(),
        .Zero()
    );
    assign PC = PC_reg;

    always @(posedge clk) begin
        if (rst) begin
            PC_reg <= 32'b0;
        end
        else begin
            PC_reg <= Jump ? Jump_Address :
                       ((branch_condition) ? PC_branch : PC_4);
        end
    end
```

当输入信号 rst 为 1 时，PC 值会被置为 0；当 rst 为 0 时，PC 值更新主要分为 3 种情况。首先使用 PC_4 这一 wire 型变量存储 PC+4 的值，第一种是 R_Type 跳转指令和 J_Type 指令的跳转类，R_Type 跳转指令会将 PC 值更新

为从 rs 寄存器读到的数据，J_Type 指令中 PC 值更新值的第 31-28 位是 PC_4 的 31-28 位，第 27-2 位是从 Instruction 中拆分出来的 instr_index，最后两位是 0；第二种是 REGIMM 和 I_Type 分支指令组成的分支类，当指令属于这一类且满足分支条件时（如 beq 指令需要判断读取的两个数据是否相等），PC 值更新为 PC_4 加上指令中获取的立即数（由 imm 有符号扩展至 32 位得到）。其余情况（包括第二种指令但不满足分支条件的）均为第三种，此时 PC 值更新为 PC_4。

h. Reg_file 相关操作

```
    assign RF_raddr1 = rs;
    assign RF_raddr2 = (REGIMM)? 32'b0:rt;
    assign RF_wen = ~((R_Type_jump && ~funct[0]) || REGIMM || (J_Type &&
~opcode[0])|| I_Type_memw || I_Type_branch || R_Type_mov && ((~funct[0]
&& ~Zero) || (funct[0] && Zero)));
    assign RegWriteSign = (I_Type_calc || I_Type_memr)? 1:0;
    assign RF_waddr = (J_Type && opcode[0])? 31:((RegWriteSign)? rt:rd);
    assign RF_wdata =
    ({32{R_Type_mov & ((~funct[0] & Zero) | (funct[0] & ~Zero))}} &
(RF_rdata1)) |
    ({32{(J_Type & opcode[0]) | (R_Type_jump & funct[0])}}) & (PC + 8)
|
     ({32{I_Type_calc & (opcode[2:0] == 3'b111)}}) & {imm[15:0],
{16{1'b0}}} |
    {32{R_Type_calc | (I_Type_calc & (opcode[2:0] != 3'b111))}} &
ALU_Result |
    ({32{R_Type_shift}} & Shifter_Result) |
    ({32{I_memr_lb}} & {{24{lb_data[7]}}, lb_data}) |
    ({32{I_memr_lh}} & {{16{lh_data[15]}}, lh_data}) |
    ({32{I_memr_lw}} & Read_data) |
    ({32{I_memr_lbu}} & {{24{1'b0}}, lb_data}) |
    ({32{I_memr_lhu}} & {{16{1'b0}}, lh_data}) |
    ({32{I_memr_lwl}} & lwl_data) |
    ({32{I_memr_lwr}} & lwr_data);
```

1 号读寄存器编码一直为从指令拆分出来的 rs，2 号读寄存器编码在

REGIMM 型指令设置为 0，其余情况均设置为从指令拆分出来的 rt。

写使能信号在以下情况设置为 0：R_Type 跳转指令中的 jr 指令，REGIMM 型指令，J_Type 指令中的 j 指令，I_Type 中的内存写和分支指令，R_Type mov 指令中不符合移动条件的情况。其余情况写使能信号均设置为 1。

写寄存器编号在 jal 指令中设置为 31，在 I_Type 内存读和计算指令中设置为指令分拆出来的 rt，其余情况均设置为指令拆分出来的 rd。

写入的数据如下设置：在 R_Type mov 指令中，若满足跳转条件，则设置为从 1 号寄存器中读到的数据；在 jal 和 jalr 指令中，设置为 PC+8；在 lui 指令中，设置为读到的立即数加上 16 位 0；在 R_Type 计算指令和 I_Type 除 lui 的计算指令中，设置为 ALU 计算结果；在 R_Type 移位指令中，设置为移位器的计算结果；在内存读指令中，依据不同指令进行设置，各指令写入的数据由以下代码给出，较为复杂，不再赘述，可查阅 MIPS 指令集。

```
    assign byte_position = ALU_Result[1:0];
    assign lb_data =
    (byte_position[1] && byte_position[0])? Read_data[31:24]:
    ((byte_position[1] && ~byte_position[0])? Read_data[23:16]:
    (~byte_position[1] && byte_position[0]? Read_data[15:8]:
                      Read_data[7:0]));
    assign lh_data = (~byte_position[1] && ~byte_position[0])?
Read_data[15:0]:Read_data[31:16];
    assign lwl_data =
    (byte_position[1] && byte_position[0])? Read_data[31:0]:
    ((byte_position[1] && ~byte_position[0])? {Read_data[23:0],
RF_rdata2[7:0]}:
    (~byte_position[1] && byte_position[0]? {Read_data[15:0],
RF_rdata2[15:0]}:{Read_data[7:0], RF_rdata2[23:0]}));
    assign
    lwr_data = (byte_position[1] && byte_position[0])?
{RF_rdata2[31:8], Read_data[31:24]}:
    ((byte_position[1] && ~byte_position[0])? {RF_rdata2[31:16],
Read_data[31:16]}:
    (~byte_position[1] && byte_position[0]? {RF_rdata2[31:24],
Read_data[31:8]}:Read_data[31:0]));
```

i. 访存指令相关

```
    assign Address = {ALU_Result[31:2], 2'b00};
    assign sb_strb = 4'b1000 >> (~ALU_Result[1:0]);
    assign sh_strb = {ALU_Result[1], ALU_Result[1], ~ALU_Result[1],
~ALU_Result[1]};
    assign sw_strb = 4'b1111;
    assign swl_strb = {ALU_Result[1] & ALU_Result[0], ALU_Result[1],
ALU_Result[1] | ALU_Result[0], 1'b1};
    assign swr_strb = {1'b1, ~ALU_Result[1] | ~ALU_Result[0],
~ALU_Result[1], ~ALU_Result[1] & ~ALU_Result[0]};
    assign Write_strb = ({4{I_memw_sb}} & sb_strb) |
                        ({4{I_memw_sh}} & sh_strb) |
                        ({4{I_memw_sw}} & sw_strb) |
                        ({4{I_memw_swl}} & swl_strb) |
                        ({4{I_memw_swr}} & swr_strb) ;

    assign sb_data =
    ({32{sb_strb[0]}} & {{24{1'b0}}, RF_rdata2[7:0]}) |
    ({32{sb_strb[1]}} & {{16{1'b0}}, RF_rdata2[7:0], {8{1'b0}}}) |
    ({32{sb_strb[2]}} & {{8{1'b0}}, RF_rdata2[7:0], {16{1'b0}}}) |
    ({32{sb_strb[3]}} & {RF_rdata2[7:0], {24{1'b0}}});
    assign sh_data =
    ({32{sh_strb[0]}} & {{16{1'b0}}, RF_rdata2[15:0]}) |
    ({32{sh_strb[2]}} & {RF_rdata2[15:0], {16{1'b0}}});
    assign sw_data = RF_rdata2;
    assign swl_data =
    ({32{sb_strb[3]}} & RF_rdata2) |
    ({32{sb_strb[2]}} & {{8{1'b0}}, RF_rdata2[31:8]}) |
    ({32{sb_strb[1]}} & {{16{1'b0}}, RF_rdata2[31:16]}) |
    ({32{sb_strb[0]}} & {{24{1'b0}}, RF_rdata2[31:24]});
    assign swr_data =
    ({32{sb_strb[0]}} & RF_rdata2) |
    ({32{sb_strb[1]}} & {RF_rdata2[23:0], {8{1'b0}}}) |
    ({32{sb_strb[2]}} & {RF_rdata2[15:0], {16{1'b0}}}) |
     ({32{sb_strb[3]}} & {RF_rdata2[7:0], {24{1'b0}}});

    assign Write_data = ({32{I_memw_sb}} & sb_data) |
                        ({32{I_memw_sh}} & sh_data) |
                        ({32{I_memw_sw}} & sw_data) |
                        ({32{I_memw_swl}} & swl_data) |
                        ({32{I_memw_swr}} & swr_data);
```

由于已在 ALU 部分设置过相关部分，Address 设置只需使用 ALU 计算结果即可。Write_strb 和 write_data 也根据具体指令的不同需要具体设置，较为复杂，具体可查阅 MIPS 指令集手册。

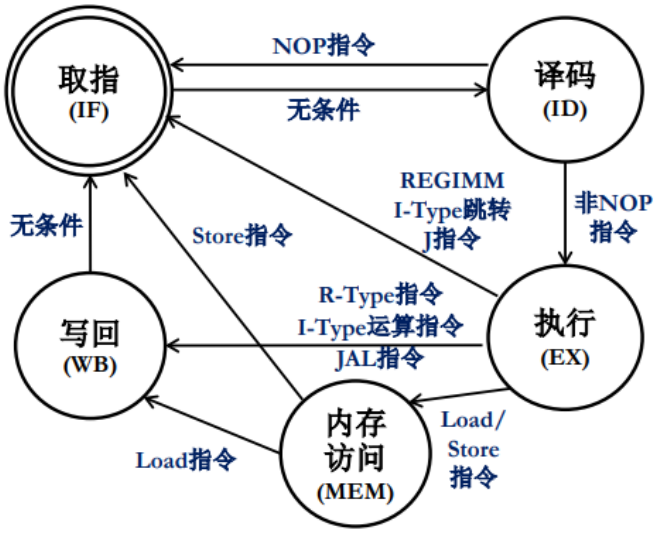
以下为多周期 CPU 部分

j. 状态转移指令

```
    always @ (posedge clk) begin
            if (rst) begin
                current_state <= `S_IF;
            end
            else begin
                current_state <= next_state;
            end
    end

    always @ (*) begin
        case (current_state)
            `S_IF: next_state = `S_ID;
            `S_ID: begin
                if (Instruction_reg == 32'b0) next_state = `S_IF;
                else next_state = `S_EX;
            end
            `S_EX: begin
                if (I_Type_memw || I_Type_memr) next_state = `S_MEM;
                else if (R_Type || I_Type_calc || (J_Type && opcode[0]))
next_state = `S_WB;
                else next_state = `S_IF;
            end
            `S_MEM: begin
                if (I_Type_memr) next_state = `S_WB;
                else next_state = `S_IF;
            end
            `S_WB: begin
                next_state = `S_IF;
            end
            default: next_state = `S_IF;
        endcase
    end
```

该部分电路设计使用三段式状态机，前两段由以上的代码给出，代码依照以下的状态转移图进行设计：



k. ALU_src 寄存器的设置

```
    assign ALU_A = ({32{state_IF | state_ID}} & PC) |
                   ({32{state_EX}} & ALU_A_reg);
    assign ALU_B = ({32{state_IF}} & {{29{1'b0}}, 3'b100}) |
                   ({32{state_ID}} & {sign_left_extend}) |
                   ({32{state_EX}} & ALU_B_reg);
    assign ALUop =
    (state_IF | state_ID)? `ALUOP_ADD :
    ((state_EX)? ((R_Type_calc)? ((funct[3:2] == 2'b00)? {funct[1],
2'b10} :
    ((funct[3:2] == 2'b01)? {funct[1], 1'b0, funct[0]} :
    {~funct[0], 2'b11})) :
    ((I_Type_calc && opcode[2:0] != 3'b111)? ((opcode[2:1] == 2'b00)?
{opcode[1], 2'b10} :
    ((opcode[2])? {opcode[1], 1'b0, opcode[0]} :
    {~opcode[0], 2'b11})) :
    ((R_Type_mov || (I_Type_branch && opcode[1] == 1'b0))? `ALU_SUB :
    ((REGIMM || (I_Type_branch && opcode[1] == 1'b1))? `ALU_SLT :
    `ALUOP_ADD)))) : `ALUOP_ADD);
```

```
    always @(posedge clk) begin
        if (state_ID) begin
            ALUOut <= ALU_Result;
            ALU_A_reg <= (({32{R_Type_mov | (I_Type_branch &
~opcode[1])}} & RF_rdata2) |
                        ({32{R_Type_jump | J_Type}} & PC) |
                        ({32{I_Type_branch & opcode[1]}} & {32{1'b0}}) |
                        ({32{R_Type_calc | REGIMM | I_Type_calc |
I_Type_memw | I_Type_memr}} & RF_rdata1));
            ALU_B_reg <= (({32{R_Type_calc}} & RF_rdata2) |
                        ({32{I_Type_branch & opcode[1] | (I_Type_branch
& ~opcode[1])}} & RF_rdata1) |
                        ({32{REGIMM | R_Type_mov}}) & {32{1'b0}} |
                        ({32{ALUsrc}} & extend) |
                        ({32{R_Type_jump | J_Type}} & {{29{1'b0}},
3'b100}));
            Shifter_A_reg <= RF_rdata2;
            Shifter_B_reg <= {5{funct[2]}} & RF_rdata1 |
                                {5{~funct[2]}} & shamt;
        end
        if (state_EX) begin
            if (R_Type_shift) ALUOut <= Shifter_Result;
            else ALUOut <= ALU_Result;
        end
    end
```

这是多周期 CPU 另一个主要更改的部分：实现 ALU 的复用，具体过程较为复杂，以一条指令（beq）为例进行讲解。Beq 指令是在三个时钟周期（取指、译码、执行）内完成的，在取指阶段，ALU 实现 PC+4 的操作，在译码阶段，ALU 实现 PC+offset 的操作，在执行阶段，ALU 判断 rs 和 rt 寄存器所读到的数据是否相等，若相等，PC 值更新为 PC+offset 的结果，若不相等，则更新为 PC+4 的结果。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

本次实验的问题：

解决方法：

1. 本次实验遇到的问题绝大部分都来源于编写代码时不认真导致的缺漏和失误，其中不乏许多笔误。

   解决方法：通过波形图查找具体出错原因。单周期 CPU 编写过程中查错较为简单，我所遇到的所有问题都是当周期即错即报，可以在波形图中查看相应指令，在程序中进行具体更改。而多周期 CPU 较为复杂，需要进行回溯，后面遇到的大量问题都需要根据 Instruction 值先回溯到 PC 值出错的地方，此时一般是从寄存器堆读到的数据出错，进而向前寻找相应地址寄存器写入的数据出错的地方，从而对相应指令对应的代码进行修改。

2. 对 PC 值修改的各种条件把握不清。

   解决方法：由于 PC 值更新牵扯的指令较多且跳转条件较为复杂，这里花了较长的时间，通过阅读 MIPS 指令集手册和阅读相关资料才解决。最后我将 PC 值更新分为了 3 类，在前文已经叙述过，不再重复叙述。

3. 对 shifter 中算术移位的语法把握不清

   解决方法：由于 verilog 算术移位时区分有符号和无符号数且对这里没有什么了解，最后我选择了将操组数有符号扩展成 64 位后进行逻辑右移再取低 32 位的方法。

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?

对于 R_Type 运算指令，可以根据 funct[3:2]位确定运算类型；对于 I_Type 运算指令，可以根据 opcode[2:1]确定运算类型。对于以上的运算指令，完成分类后可以根据 funct 或 opcode 中的某一位或两位确定具体运算指令。

2. 表格中的 ALUop 编码是否还有优化空间

若允许更改指令编码，可以令 funct(opcode)的 3-1 位表示 ALUop，第 0 位表示是无符号还是有符号操作，这样可以降低译码过程的复杂度，且能提高代码可读性。

四、 在课后，你花费了大约 30 小时完成此次实验?

五、 对于此次实验的心得、感受和建议

本次实验难度适中，但由于以往没有接触过如此大工作量的 verilog 操作，刚上手时有一定的无力感，不知从何下手。后来随着对指令理解的加深，逐渐完成了代码的编写，这里要感谢详细的课上 PPT 资料以及 MIPS 指令集手册。

本次实验测试点测试时间略长，不过由于报错较快，这里没有产生太大的影响。在调试过程中，加深了我对 verilog 用法的了解，极大提高了阅读波形图的能力。

非常感谢张科老师以及助教老师的认真讲解，使得我第一次完成这种较大工作量的 verilog 程序编写。