中国科学院大学计算机组成原理（研讨课）

实 验 报 告

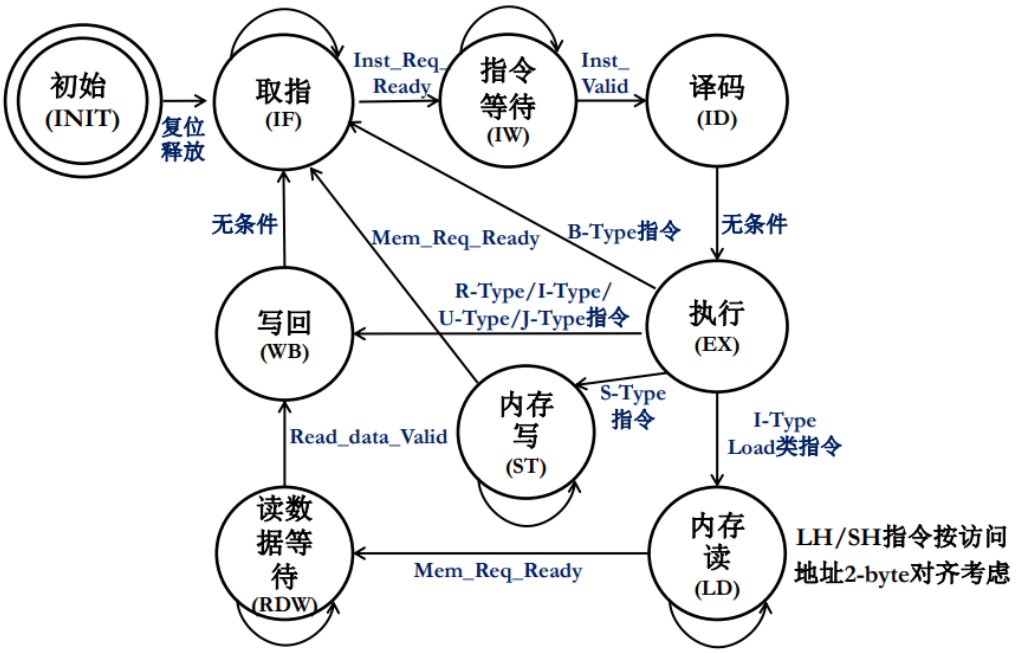
学号：2021K8009929016　　姓名：李金明　专业：计算机科学与技术

实验序号：4　　实验名称：定制 RISC-V 功能型处理器设计

一、 逻辑电路结构及说明

1. 状态转移部分

该部分电路设计使用三段式状态机，代码依照以下的状态转移图进行设计：



状态转移图与定制 MIPS 功能型处理器的基本相同，除了在 EX 状态的跳转条件有稍加改动，这是 MIPS 和 RISC-V 译码方式不同引起的。

状态机前两段设计由以下代码给出：

```
`define S_IF 9'b000000001
`define S_ID 9'b000000010
`define S_EX 9'b000000100
`define S_ST 9'b000001000
`define S_WB 9'b000010000
`define S_LD 9'b000100000
`define S_RDW 9'b001000000
`define S_IW 9'b010000000
`define S_INIT 9'b100000000
    assign state_IF = current_state == `S_IF;
    assign state_ID = current_state == `S_ID;
    assign state_EX = current_state == `S_EX;
    assign state_ST = current_state == `S_ST;
    assign state_WB = current_state == `S_WB;
    assign state_LD = current_state == `S_LD;
    assign state_RDW = current_state == `S_RDW;
    assign state_IW = current_state == `S_IW;
    assign state_INIT = current_state == `S_INIT;
    always @ (posedge clk) begin
            if (rst) begin
                current_state <= `S_INIT;
            end
            else begin
                current_state <= next_state;
            end
    end

    always @ (*) begin
        case (current_state)
            `S_INIT: next_state = `S_IF;
            `S_IF: begin
                if (Inst_Req_Ready) next_state = `S_IW;
                else next_state = `S_IF;
            end
            `S_IW: begin
                if (Inst_Valid) next_state = `S_ID;
                else next_state = `S_IW;
            end
            `S_ID: begin
                next_state = `S_EX;
            end
```

```
`S_EX: begin
                if (S_Type) next_state = `S_ST;
                else if (I_Type_load) next_state = `S_LD;
                else if (R_Type || I_Type && ~I_Type_load || U_Type
|| J_Type) next_state = `S_WB;
                else next_state = `S_IF;
            end
            `S_ST: begin
                if (Mem_Req_Ready) next_state = `S_IF;
                else next_state = `S_ST;
            end
            `S_WB: begin
                next_state = `S_IF;
            end
            `S_LD: begin
                if (Mem_Req_Ready) next_state = `S_RDW;
                else next_state = `S_LD;
            end
            `S_RDW: begin
                if (Read_data_Valid) next_state = `S_WB;
                else next_state = `S_RDW;
            end
            default: next_state = `S_INIT;
        endcase
    end
```

总共有 9 个状态，这 9 个状态使用独热码的方式进行编码。

第一段使用 always 时序逻辑描述状态机的跳转。

第二段使用 always 组合逻辑，根据当前状态和输入信号决定下一状态。

2. 译码部分

在 RISC-V 指令集中，指令被分为 6 大类，R_Type，I_Type，S_Type，B_Type，U_Type 以及 J_Type，其中在实现的指令中 I_Type 可继续划分为计算类、load 类以及跳转类。这些指令可以根据 opcode（指令的 0-6 位）确

定所属类别。确定类别由以下代码给出：

```
`define R_Type_opcode 7'b0110011
`define I_Type_opcode 7'b0010011
`define S_Type_opcode 7'b0100011
`define I_Type_load_opcode 7'b0000011
`define I_Type_jump_opcode 7'b1100111
`define B_Type_opcode 7'b1100011
`define U_Type_opcode 5'b10111
`define J_Type_opcode 7'b1101111

    assign opcode = Instruction_reg[6:0];
    assign R_Type = opcode == `R_Type_opcode;
    assign I_Type = (opcode == `I_Type_opcode | opcode ==
`I_Type_load_opcode | opcode == `I_Type_jump_opcode);
    assign I_Type_load = opcode == `I_Type_load_opcode;
    assign I_Type_jump = opcode == `I_Type_jump_opcode;
    assign S_Type = opcode == `S_Type_opcode;
    assign B_Type = opcode == `B_Type_opcode;
    assign U_Type = opcode[4:0] == `U_Type_opcode;
    assign J_Type = opcode == `J_Type_opcode;
```

剩余指令的拆解由以下的代码给出：

```
    assign funct7 = Instruction_reg[31:25];
    assign rs2 = Instruction_reg[24:20];
    assign rs1 = Instruction_reg[19:15];
    assign rd = Instruction_reg[11:7];
    assign funct3 = Instruction_reg[14:12];
    assign imm_I = Instruction_reg[31:20];
    assign imm_S = {Instruction_reg[31:25], Instruction_reg[11:7]};
    assign imm_B = {Instruction_reg[31], Instruction_reg[7],
Instruction_reg[30:25], Instruction_reg[11:8]};
    assign imm_J = {Instruction_reg[31], Instruction_reg[19:12],
Instruction_reg[20], Instruction_reg[30:21]};
    assign imm_U = {Instruction_reg[31:12]};
```

a. 由于 ALU 和 shifter 是分开实现的，对于 R_Type 和 I_Type 计算类指令而言，还需确定指令是计算还是移位指令，这可以根据 funct3 进行

区分（移位类指令 funct3 低两位时 01，计算类均不是），由以下代码给出：

```
    assign shift = ~I_Type_load & ~I_Type_jump & ~funct3[1] &
funct3[0];
    assign calc = ~I_Type_load & ~I_Type_jump & (funct3[1] |
~funct3[0]);
```

b. 内存读写指令属于 S_Type 或 I_Type load 型指令，在实现的指令中可以根据 funct3 确定需要读取或存储的数据时 1 字节、2 字节还是 4 字节，由以下代码给出：

```
    assign memory_byte = (I_Type_load | S_Type) & (~funct3[1] &
~funct3[0]);
    assign memory_half = (I_Type_load | S_Type) & (~funct3[1] &
funct3[0]);
    assign memory_word = (I_Type_load | S_Type) & (funct3[1] &
~funct3[0]);
```

c. 立即数的获取

在 RISC-V 指令集中，除移位指令外的 I_Type 型指令、S_Type 型指令、U_Type 型指令以及 J_Type 型指令需要进行扩展，其中 I_Type 和 S_Type 指令使用有符号扩展，B_Type，J_Type 以及 U_Type 进行有符号扩展后需要左移 1 位，由以下代码给出：

```
    assign ALUsrc = (I_Type && ~shift || S_Type || U_Type|| J_Type)?
1:0;
    assign extend = (S_Type | I_Type_load | (I_Type & calc))?
signed_extend :sign_left_extend;

    assign signed_extend =
    ({32{I_Type}}) & {{20{imm_I[11]}}, imm_I[11:0]} |
    ({32{S_Type}}) & {{20{imm_S[11]}}, imm_S[11:0]};
    assign sign_left_extend =
    ({32{B_Type}}) & {{19{imm_B[11]}}, imm_B[11:0], 1'b0} |
    ({32{J_Type}} & {{11{imm_J[19]}}, imm_J[19:0], 1'b0}) |
    ({32{U_Type}} & {imm_U[19:0], 12'b0});
```

d. ALUop 及 Shiftop 的编码

```
    assign ALUop = (state_IF | state_ID)? `ALUOP_ADD :
                   ((state_EX)? (((R_Type || I_Type) && calc)?
((~funct3[2] & ~funct3[1])? {funct7[5] & R_Type, 2'b10} :
 ((~funct3[2] & funct3[1])? {~funct3[0], 2'b11} :
{~funct3[1], 1'b0, funct3[1] & ~funct3[0]})) :
 ((B_Type && ~funct3[2])? `ALU_SUB :
 ((B_Type && ~funct3[1])? `ALU_SLT :
 ((B_Type)? `ALU_SLTU :
`ALUOP_ADD)))) : `ALUOP_ADD);

    assign Shiftop =
    ({2{(R_Type || I_Type) && shift && state_EX}}) & {funct3[2],
funct7[5]};
```

相比 MIPS 处理器，主要变化是对 R_Type 和 I-Type 中的 ALUop 编码的重写，仍根据 funct3 确定 ALUop，其中加（减）和算术（逻辑）右移还需要根据 funct7 的第 5 位确定具体指令。

3. PC 的赋值

```
    always @(posedge clk) begin
        if (rst) PC <= 32'b0;
        else if (state_IF) begin
            PC_4 <= ALU_Result;
            PC_reg <= PC;
        end
        else if (state_ID && Instruction_reg == 32'b0) PC <= PC_4;
        else if (state_EX) begin
            if (I_Type_jump || J_Type) PC <= ALU_Result;
            else if (B_Type && ((~funct3[2] && (funct3[0] ^ Zero)) ||
(funct3[2] && (~funct3[0] ^ (ALU_Result == 32'b0)))))
                    PC <= ALUOut;
            else PC <= PC_4;
        end
    end
```

PC 的赋值仅进行了小部分改动：根据译码指令分类的不同 ID 状态更改了

跳转以及分支条件，由于少了 ble，bgt 等指令，该部分判断起来较为简单。

4. 寄存器堆相关操作

```
    assign RF_raddr1 = rs1;
    assign RF_raddr2 = rs2;
    assign RF_wen = (state_WB) & ~(S_Type | B_Type);
    assign RF_waddr = {5{state_WB}} & rd;
    assign RF_wdata =
    ({32{U_Type & opcode[5]}}) & {imm_U[19:0], {12{1'b0}}} |
    {32{(R_Type | I_Type) & (calc | shift) | U_Type & ~opcode[5]}} &
ALUOut |
    {32{(I_Type_jump | J_Type)}} & PC_4 |
    ({32{I_Type_load & memory_byte & ~funct3[2]}} &
{{24{lb_data[7]}}, lb_data}) |
    ({32{I_Type_load & memory_half & ~funct3[2]}} &
{{16{lh_data[15]}}, lh_data}) |
    ({32{I_Type_load & memory_word & ~funct3[2]}} & Read_data_reg) |
    ({32{I_Type_load & memory_byte & funct3[2]}} & {{24{1'b0}},
lb_data}) |
    ({32{I_Type_load & memory_half & funct3[2]}} & {{16{1'b0}},
lh_data});

    assign byte_position = ALUOut[1:0];
    assign lb_data =
    (byte_position[1] && byte_position[0])? Read_data_reg[31:24]:
    ((byte_position[1] && ~byte_position[0])? Read_data_reg[23:16]:
    (~byte_position[1] && byte_position[0]? Read_data_reg[15:8]:
                        Read_data_reg[7:0]));
    assign lh_data = (~byte_position[1] && ~byte_position[0])?
Read_data_reg[15:0]: Read_data_reg[31:16];
```

对于需要对寄存器堆进行操作的指令，其读寄存器编号均为 rs1，rs2，写寄存器均为 rd，其中除 S_Type 和 B_Type 外的指令都需要拉高 RF_wen，写回数据与以往相差不大，需要注意的是 jump 型指令需要写回 PC+4 而不是 PC+8。

5. 向内存写入的数据

```
    assign Address = {ALUOut[31:2], 2'b00};
    assign sb_strb = 4'b1000 >> (~ALUOut[1:0]);
    assign sh_strb = {ALUOut[1], ALUOut[1], ~ALUOut[1], ~ALUOut[1]};
    assign sw_strb = 4'b1111;
    assign Write_strb = ({4{S_Type & memory_byte}} & sb_strb) |
                        ({4{S_Type & memory_half}} & sh_strb) |
                        ({4{S_Type & memory_word}} & sw_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 Write_data = ({32{S_Type & memory_byte}} & sb_data) |
                        ({32{S_Type & memory_half}} & sh_data) |
                        ({32{S_Type & memory_word}} & sw_data);
```

这一部分也与 MIPS 处理器相差不大，仅删去了没有的 lwl 指令和 lwr 指令相关操作。

6. 性能计数器的设计

```
    reg [31:0] cycle_cnt;
    reg [31:0] memory_cnt;
    reg [31:0] instruction_cnt;
    reg [31:0] wait_cnt;
    reg [31:0] load_cnt;
    reg [31:0] instruction_fetch_cnt;
    reg [31:0] instruction_wait_cnt;
    reg [31:0] mem_wait_cnt;
    reg [31:0] rdw_cnt;

    always @(posedge clk) begin
        if (rst) cycle_cnt <= 32'b0;
        else cycle_cnt <= cycle_cnt + 32'b1;
    end
    assign cpu_perf_cnt_0 = cycle_cnt;

    always @(posedge clk) begin
        if (rst) memory_cnt <= 32'b0;
        else if (state_LD || state_ST) memory_cnt = memory_cnt
+32'b1;
    end
    assign cpu_perf_cnt_1 = memory_cnt;

    always @(posedge clk) begin
        if (rst) instruction_cnt <= 32'b0;
        else if (state_IF) instruction_cnt = instruction_cnt +32'b1;
    end
    assign cpu_perf_cnt_2 = instruction_cnt;

    always @(posedge clk) begin
        if (rst) wait_cnt <= 32'b0;
        else if (state_IF && ~Inst_Req_Ready || state_IW &&
~Inst_Valid || (state_LD || state_ST) && ~Mem_Req_Ready || state_RDW
&& ~Read_data_Valid) wait_cnt = wait_cnt +32'b1;
    end
    assign cpu_perf_cnt_3 = wait_cnt;
```

```
always @(posedge clk) begin
        if (rst) load_cnt <= 32'b0;
        else if (state_EX && I_Type_memr) load_cnt <= load_cnt +
32'b1;
    end
    assign cpu_perf_cnt_4 = load_cnt;

    always @(posedge clk) begin
        if (rst) instruction_fetch_cnt <= 32'b0;
        else if (state_IF && ~Inst_Req_Ready) instruction_fetch_cnt =
instruction_fetch_cnt +32'b1;
end
    assign cpu_perf_cnt_5 = instruction_fetch_cnt;

    always @(posedge clk) begin
        if (rst) instruction_wait_cnt <= 32'b0;
        else if (state_IW && ~Inst_Valid) instruction_wait_cnt =
instruction_wait_cnt +32'b1;
    end
    assign cpu_perf_cnt_6 = instruction_wait_cnt;

always @(posedge clk) begin
        if (rst) mem_wait_cnt <= 32'b0;
        else if ((state_LD || state_ST) && ~Mem_Req_Ready)
mem_wait_cnt = mem_wait_cnt +32'b1;
    end
    assign cpu_perf_cnt_7 = mem_wait_cnt;

    always @(posedge clk) begin
        if (rst) rdw_cnt <= 32'b0;
       else if (state_RDW && ~Read_data_Valid) rdw_cnt = rdw_cnt
    +32'b1;
    end
    assign cpu_perf_cnt_8 = rdw_cnt;
```

共设计了 9 个性能计数器，计数内容如下：

cpu_perf_cnt_0：周期计数器；

cpu_perf_cnt_1：访问 memory 次数计数器；

cpu_perf_cnt_2：指令计数器；

cpu_perf_cnt_3：等待时长计数器；

cpu_perf_cnt_4：load 类指令计数器；

cpu_perf_cnt_5：IF 状态等待周期数；

cpu_perf_cnt_6：IW 状态等待周期数；

cpu_perf_cnt_7：LD 及 ST 状态等待周期数；

cpu_perf_cnt_8：RDW 状态等待周期数。

7. 性能计数器打印结果

a. MIPS 处理器

```
Cycles: 52557723
Memory Load/Store: 54020
Instructions: 728305
Cycles Waiting totally: 49089087
Load Instructions: 11595
Cycles Waiting for Fetching Instructions: 0
Cycles Waiting for Instructions: 48248267
Cycles Waiting for Requesting Data: 34725
Cycles Waiting for Reading Data: 806890
```

b. RISC-V 处理器

```
Cycles: 44719087
Memory Load/Store: 52266
Instructions: 619041
Cycles Waiting totally: 41734799
Load Instructions: 11289
Cycles Waiting for Fetching Instructions: 0
Cycles Waiting for Instructions: 40942014
Cycles Waiting for Requesting Data: 33564
Cycles Waiting for Reading Data: 760140
```

可以观察到以 ssort 程序为例，RISC-V 处理器的时钟周期数有了明显的减少，这得益于指令数的大幅减少以及访存次数的减少。

二、 实验过程中遇到的问题、对问题的思考过程及解决方法

该部分的问题主要由笔误和没有正确理解指令导致的。我在 ALU、Shiftop 的编码上存在一些笔误，这导致进行相关操作时出现错误。在立即数扩展部分没有意识到 I_Type 中所有立即数均进行有符号扩展，与以往部分有符号扩展部分无符号扩展相左。

这两个问题均是通过仔细阅读 RISC-V 指令集手册解决的。主要困难在发现问题的过程非常麻烦，因为写入寄存器的数据出错时不会报错，只有后续操作才有可能报错，在这个实验中往往需要回溯好几次有寄存器写回操作的指令才能发现问题所在。有一次程序甚至跑到了 100000+ns 才出错。如果在写回（RF_wen=1）操作时能比较写回数据与金标准是否一致，可以大幅缩短 debug 时间

三、 在课后，你花费了大约 5 小时完成此次实验?

四、 对于此次实验的心得、感受和建议

通过这次实验，我认识到了 RISC-V 指令集与 MIPS 指令集的一些区别。在编码格式上，RISC-V 指令集更为规整，译码难度明显降低，例如对读写寄存器的编码，不像 MIPS 一样有各种特例，而统一为 rs1，rs2 和 rd。另外如计算和分支指令，其区分可由 funct3（有时包括 funct7 的第 5 位）确定，较为简单。另外，RISC-V 在性能上也有所提升。

本次实验行为仿真中矩阵乘及水仙花测试时间依然较长，而且过去的实验从未通过别的实验而在这两个实验挂掉，建议取消这两个测试样例从而大幅缩短测试时间。

非常感谢所有老师的认真讲解及解答问题。