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：4 实验名称：定制 RISC-V 功能性处理器设计

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）。

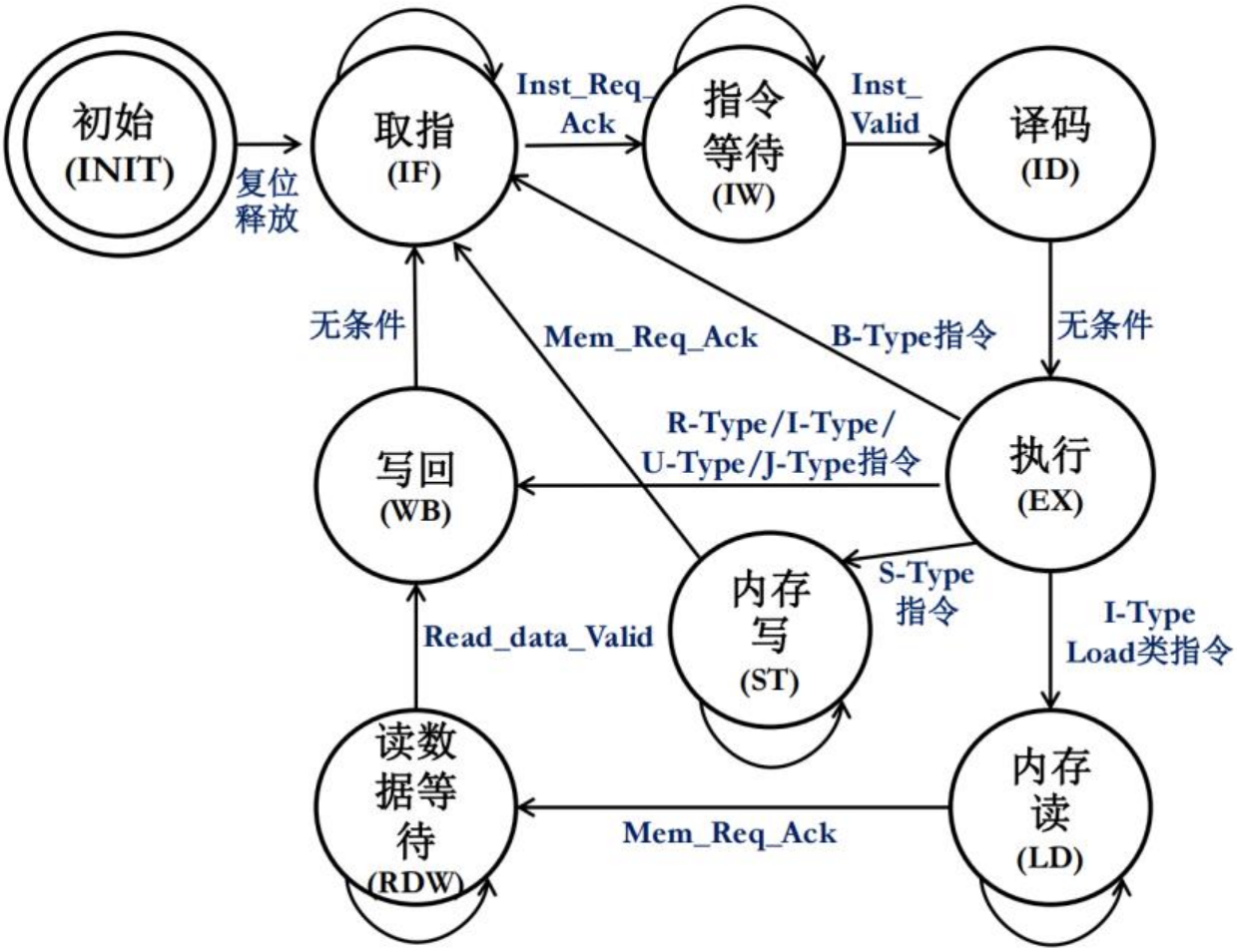
注 3：实验报告模板下列条目仅供参考，可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

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） 关键 RTL 代码段：

a） 基于 RISCV 指令集的多周期状态机

状态转移图与课件 PPT 上给出的一致：



如图所示，一共 9 个状态，用独热码对状态进行编码，如下：

```
localparam INIT =9'b000000001;
localparam IF   =9'b000000010;
localparam IW   =9'b000000100;
localparam ID   =9'b000001000;
localparam EX   =9'b000010000;
localparam ST   =9'b000100000;
localparam LD   =9'b001000000;
localparam RDW  =9'b010000000;
localparam WB   =9'b100000000;
```

状态机采用“三段式”描述方法：

I. “第一段”采用always时序逻辑，描述状态寄存器的同步跳转。

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

II. “第二段”采用 always 组合逻辑，根据当前状态机状态和输入信号，描述下一状态的计算逻辑。

```
always @(*) begin
        case (current_state)
                INIT: next_state <= IF;
                IF: begin
                        if (Inst_Req_Ready) begin
                                next_state <= IW;
                        end
                        else begin
                                next_state <= IF;
                        end
                end
                IW: begin
                        if (Inst_Valid) begin
                                next_state <= ID;
                        end
                        else begin
                                next_state <= IW;
                        end
                end
                ID: next_state <= EX;
```

```
        EX: begin
                if (R_Type | I_Type & ~LOAD | U_Type | J_Type) begin
                        next_state <= WB;
                end
                else if (LOAD) begin
                        next_state <= LD;
                end
                else if (S_Type) begin
                        next_state <= ST;
                end
                else if (B_Type) begin
                        next_state <= IF;
                end
                else begin
                        next_state <= INIT;
                end
        end
        ST: begin
                if (Mem_Req_Ready) begin
                        next_state <= IF;
                end else begin
                        next_state <= ST;
                end
        end
```

```
            LD: begin
                    if (Mem_Req_Ready) begin
                            next_state <= RDW;
                    end else begin
                            next_state <= LD;
                    end
            end
            RDW: begin
                    if (Read_data_Valid) begin
                            next_state <= WB;
                    end else begin
                            next_state <= RDW;
                    end
            end
            WB: next_state <= IF;
            default: next_state <= INIT;
        endcase
end
```

III. “第三段”采用 always 时序逻辑或 assign 组合逻辑，根据当前状态机的状态，描述不同输出信号的变化。

下面是第三段的 always 时序逻辑部分：

```
//PC
always @(posedge clk) begin
        if (current_state == IW && Inst_Valid) begin
                PC_normal <= ALUResult; //ALU算出的PC+4
        end
end

always @(posedge clk) begin
        if (rst) begin
                PC <= 32'd0;
        end
        else if (current_state == EX) begin
                if (JAL) begin
                        PC <= Result; // ID阶段算出的PC(复用ALU)
                end
                else if (JALR) begin
                        PC <= Result & {~31'b0, 1'b0};  // ID阶段算出的PC(末尾清0)
                end
                else if (Branch_or_not) begin //branch
                        PC <= Result; // ID阶段算出的PC
                end
                else begin
                        PC <= PC_normal;
                end
        end
end
```

```
//reg_Instruction
always @(posedge clk) begin
        if (current_state == IW && Inst_Valid) begin
                reg_Instruction <= Instruction;
        end
end


//register
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
```

```
//register
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
always @(posedge clk) begin
        if (current_state == RDW && Read_data_Valid) begin
                Read_data_reg <= Read_data;
        end
end
```

```
//Read_Data
always @(posedge clk) begin
        if (current_state == RDW && Read_data_Valid) begin
                Read_data_reg <= Read_data;
        end
end
```

（注：其中重要部分在后面可能会再次提到）

b） 基于真实内存的多周期访存逻辑（与 pj3 相似）

```
//Handshake Signal
assign Inst_Req_Valid           = current_state == IF;
assign Inst_Ready               = current_state == IW || current_state == INIT;
assign MemWrite                 = current_state == ST;
assign MemRead                  = current_state == LD;
assign Read_data_Ready          = current_state == RDW || current_state == INIT;
```

如代码所示，在 IF 状态拉高 Inst_Req_Valid，在接收到 Inst_Req_Ready 有效时进入 IW 状态等待指令。在 IW 状态拉高

Inst_Ready，在接收到 Inst_Valid 有效时进入 ID 状态。对于 Store 指令，在 ST 状态拉高 MemWrite,在接收到 Mem_Req_Valid 有效时返回 IF 状态。对于 Load 指令,在 LD 状态拉高 MemRead,在接收到 Mem_Req_Valid 有效时进入 RDW 等待读数,当 Read_data_Ready 有效时返回 IF 状态。

c) PC 更新（复用 ALU）

```
//PC
always @(posedge clk) begin
        if (current_state == IW && Inst_Valid) begin
                PC_normal <= ALUResult; //ALU算出的PC+4
        end
end

always @(posedge clk) begin
        if (rst) begin
                PC <= 32'd0;
        end
        else if (current_state == EX) begin
                if (JAL) begin
                        PC <= Result; // ID阶段算出的PC(复用ALU)
                end
                else if (JALR) begin
                        PC <= Result & {~31'b0, 1'b0};  // ID阶段算出的PC(末尾清0)
                end
                else if (Branch_or_not) begin //branch
                        PC <= Result; // ID阶段算出的PC
                end
                else begin
                        PC <= PC_normal;
                end
        end
end
```

```
//Branch_or_not
assign Branch_or_not = (Zero ^ funct3[2] ^ funct3[0]) & Branch;
```

如图，对于 PC 的更新，本人仍然尝试复用 ALU 而不用额外的加法器，最后也确实做到了这点。

在 IW 阶段且 Inst_Valid 有效时，用 ALU 算出 PC+4，并保存给 PC_normal，然后在 ID 阶段，算出跳转指令和分支指令的目标地址，保存在 Result 寄存器里。

在 EX 阶段对 PC 进行更新，如果是跳转或者分支指令，则直接把 Result 的值赋给 PC，否则把 PC_normal（即 PC+4）的值赋值给 PC（注：JALR 指令需要对末尾清 0）。

需要注意的是,由于本次实验所实现的 RISC-V 处理器没有 MIPS 中分支延迟槽的设计，即使是 NOP 指令也有 EX 状态，因此我选择在 EX 状态更新所有指令的 PC 值。

（对于如何复用 ALU 的细节将在后面进行具体展示）

d) 立即数扩展

```
//Extend Immediate (要实现的RISCV32中的指令的立即数扩展均是有符号扩展,包括与操作，或操作，异或操作)
assign imm = {(32){U_Type}} & {reg_Instruction[31:12], 12'b0} |
             {(32){J_Type}} & {{(12){reg_Instruction[31]}}, reg_Instruction[19:12], reg_Instruction[20], reg_Instruction[30:21],1'b0} |
             {(32){B_Type}} & {{(20){reg_Instruction[31]}}, reg_Instruction[7], reg_Instruction[30:25], reg_Instruction[11:8], 1'b0} |
             {(32){I_Type}} & {{(20){reg_Instruction[31]}}, reg_Instruction[31:20]} |
             {(32){S_Type}} & {{(20){reg_Instruction[31]}}, reg_Instruction[31:25], reg_Instruction[11:7]};
```

如图，根据指令的要求对立即数进行符号位扩展（要实现的 RISCV 指令集的指令中的立即数扩展均是符号位扩展，包括与操作，或操作，异或操作）。根据不用的指令，可以得到不用的立即数扩展规则。

e) 指令解码与控制信号的生成

```
//Instruction Decode
assign {funct7, rs2, rs1, funct3, rd, opcode} = reg_Instruction;

assign OP_IMM   = opcode[6:0] == 7'b0010011;
assign LUI      = opcode[6:0] == 7'b0110111;
assign AUIPC    = opcode[6:0] == 7'b0010111;
assign OP_REG   = opcode[6:0] == 7'b0110011;
assign JAL      = opcode[6:0] == 7'b1101111;
assign JALR     = opcode[6:0] == 7'b1100111;
assign BRANCH   = opcode[6:0] == 7'b1100011;
assign LOAD     = opcode[6:0] == 7'b0000011;
assign STORE    = opcode[6:0] == 7'b0100011;

assign R_Type = OP_REG;
assign I_Type = OP_IMM | JALR | LOAD; //JALR指令和LOAD类型的指令是I_Type的格式
assign S_Type = STORE;
assign B_Type = BRANCH;
assign U_Type = LUI | AUIPC; //LUI和AUIPC是U_Type的格式
assign J_Type = JAL;
```

```
//Control Signal
assign Branch   = B_Type;
assign MemtoReg = LOAD;
assign ALUSrc   = I_Type | S_Type | U_Type; //I_Type和S_Type的ALU或Shifter操作来源是立即数
assign ShiftSrc = I_Type | S_Type;
assign RF_wen   = (current_state == WB) & (J_Type | I_Type | R_Type | U_Type); //只有S_Type和B_Type不用写回

assign ALUEn    = (OP_REG | OP_IMM) & (funct3[1] | ~funct3[0]) | JALR | LOAD | S_Type | B_Type |
                  (current_state == ID); //ID阶段Result需要对ALU的结果进行选择
assign ShiftEn  = (OP_REG | OP_IMM) & (~funct3[1] & funct3[0]) &
                  ~(current_state == ID); //ID阶段Result需要对ALU的结果进行选择
```

如图，首先根据 reg_Instruction 确定指令类型，然后根据指令类型（R_Type，J_Type，S_Type，B_Type，U_Type 以及 I_Type）和这类指令下的具体分类（如 OP_IMM，JALR，LOAD 等）以及当前状态，生成各类控制信号。

f) ALU

```
//ALU
assign ALU_A = {(32){(current_state == IW && Inst_Valid)}} & PC | //算PC+4
               {(32){current_state == ID}} & (JALR ? wire_RF_rdata1 : PC) | //算跳转和分支指令的目标地址
               {(32){current_state == EX}} & (U_Type ? PC :RF_rdata1);//AUIPC指令的操作数之一为PC

assign ALU_B = {(32){(current_state == IW && Inst_Valid)}} & {29'b0, 3'b100} | //算PC+4
               {(32){current_state == ID}} & imm | //算跳转和分支指令的目标地址
               {(32){current_state == EX}} & (ALUSrc ? imm :RF_rdata2);
```

由于本人想要复用 ALU，所以对 ALU 的两个操作数的输入逻辑如上图所示：

I. 对于 IW 阶段且 Inst_Valid 有效时，A 为 PC，B 为 4，用以计算 PC+4 的值。

II. 对于 ID 阶段，如果是 JALR 指令，A 为 wire_RF_rdata1（寄存器堆读出来的值），B 为 imm（扩展的立即数）；否则 A 为 PC，B 为 imm，用以计算跳转和分支指令的目标地址。

III. 对于执行阶段，如果是 U_Type，A 为 PC（避免 AUIPC 指令需要使用额外加法器），否则 A 为 RF_rdata1（寄存器堆读出的数据）；如果 ALUSrc 有效，B 为 imm，否则 B 为 RF_rdata2。

```
assign ALUop = (current_state == EX) ?
               (({(3){OP_IMM | OP_REG}} & {
                     (funct3 == 3'b100) | (funct3 == 3'b010) | (funct3 == 3'b000 && funct7[5] && opcode[5]), //xor slt sub类型操作首位为1，其它为0
                     ~funct3[2],                                                                             //and, or, xor类型操作第二位为0，其它为1
                     funct3[1] & ~(funct3[0] & funct3[2])                                                    //or, slt, sltu类型操作第三位为1，其它为0
               } |
               {(3){STORE | LOAD | JALR | U_Type}} & 3'b010 |                                  //add 010
               {(3){BRANCH}} & {~funct3[1], 1'b1, funct3[2]})                                  //sub 110 slt 111 sltu 011
               : 010;
```

ALUop 的生成是一个难点，根据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。

根据当前指令类型,将 ALUop 分为三类:

I. OP_IMM 或 OP, 此时对 ALUop 根据 funct3, funct7, opcode 信息进行逐位编码。

II. STORE 或 LOAD 或 JALR,此时需要对寄存器数和立即数做加法得到最终.的地址,因此直接输入 010 (add)。

III. BRANCH，此时需要判断两个寄存器数的相对大小，并根据结果选择分支，因此需要根据指令要求做减法、比较运算。

IV. 上面是对于 EX 阶段的处理，而对于 EX 以外的阶段，均为 010（加法操作），这是为了复用 ALU，而不用额外加法器。

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

上图为 ALU 的实例化。

由上面的讨论可知，RISCV 指令集实现 ALU 的复用比 MIPS 指令集实现 ALU 复用要稍微复杂一点。

g) Shifter

```
//Shifter
assign Shiftop = {funct3[2],funct7[5]}; //funct3[2]代表左移右移,funct7[5]代表逻辑移位还是算术移位

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

根据 funct3 和 funct5 进行 Shiftop 的生成，其中 funct3[2]代表左移

右移,funct7[5]代表逻辑移位还是算术移位。

然后，如果 ShiftSrc 有效，则 B 传入立即数，否则传入 RF_rdata2，传入 B 的数据均需对 32 取模，即取低五位。

h) 内存写地址和写入数据

```
//Write
assign Address          = Result & ~32'b11; //对齐
assign Write_data       = RF_rdata2 << {Result[1:0], 3'b0};
assign Write_strb       = {( Result[1] | funct3[1]) & ( Result[0] | funct3[0] | funct3[1]),
                           ( Result[1] | funct3[1]) & (~Result[0] | funct3[0] | funct3[1]),
                           (~Result[1] | funct3[1]) & ( Result[0] | funct3[0] | funct3[1]),
                           (~Result[1] | funct3[1]) & (~Result[0] | funct3[0] | funct3[1])};
```

如图，由于没有非对齐写的指令，RISCV 内存写数据的逻辑要比 MIPS 简单许多。

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 Write_strb 的逻辑与 MIPS 指令集中的对齐写回的逻辑相似，这里不作过多赘述。

i) 内存读数据与寄存器堆的写数据

```
//Write Back
assign Read_data_shifted        = Read_data_reg >> {Result[1:0], 3'b0};
assign Read_data_sign_bit       = Read_data_shifted[funct3[1:0]==2'b01 ? 15 : 7];//符号位

assign Read_data_masked         = Read_data_shifted & {{(16){funct3[1]}}, {(8){funct3[0] | funct3[1]}}, {(8){1'b1}}} |
                                  {(32){~funct3[2] & Read_data_sign_bit}} & ~{{(16){funct3[1]}}, {(8){funct3[0] | funct3[1]}}, {(8){1'b1}}};
                                  //字节掩码进行数据选择并进行符号位或者零扩展

assign RF_wdata                 = {(32){LUI}}           & imm |
                                  {(32){AUIPC}}         & Result | //避免使用额外的加法器,AUIPC指令的结果也使用ALU进行计算
                                  {(32){JAL | JALR}}    & PC_normal | //PC_normal存的是PC+4
                                  {(32){LOAD}}          & Read_data_masked |
                                  {(32){OP_REG | OP_IMM}}       & Result;
```

如图，由于没有非对齐写的指令，RISCV 内存读数据的逻辑要比 MIPS 简单许多。只需要对读数据通过字节掩码进行符号位扩展或者零扩展即可。

而寄存器堆的写回数据则需要根据指令类型进行选择，对于 LUI 指令直接传入扩展立即数，对于 AUIPC 指令则传入 Result 寄存器的数据，对于 JAL 和 JALR 则写入 PC_normal（即 PC+4）,对于 LOAD 指令则传入处理之后的读数据，否则对于 R_Type 和 I_Type 中的计算指令，传入 Result 中的值。

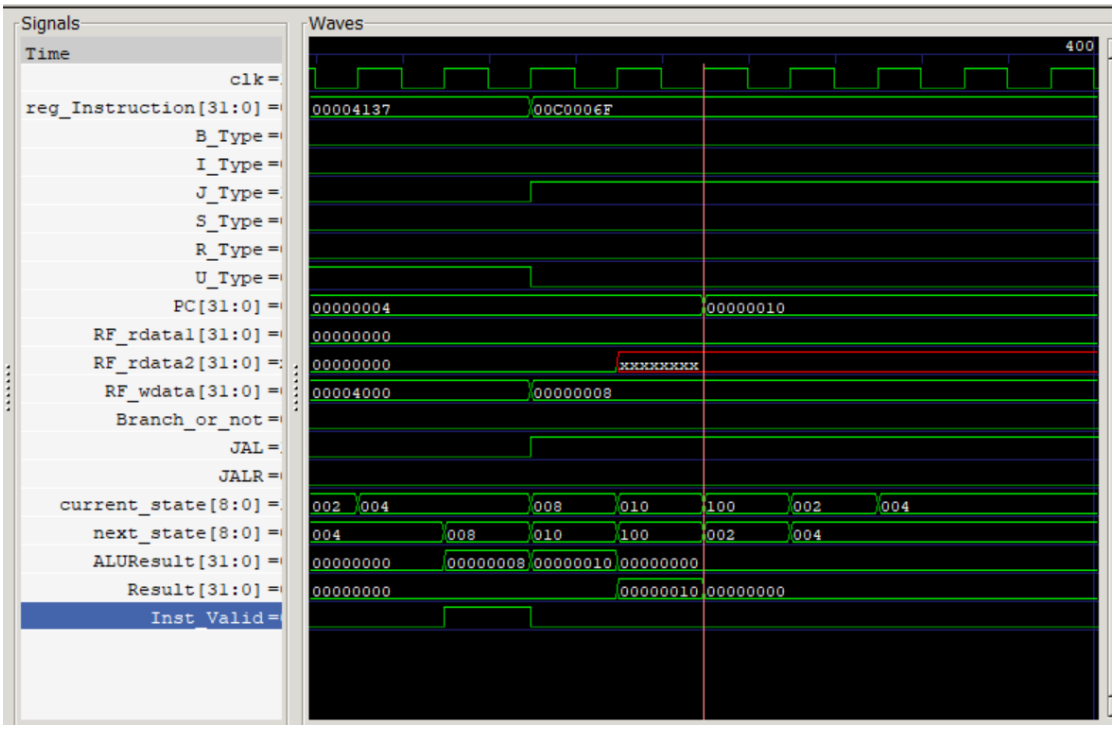
j) 寄存器堆

```
//Registers
reg_file Registers(
        .clk    (clk),
        .waddr  (RF_waddr),
        .raddr1 (rs1),
        .raddr2 (rs2),
        .wen    (RF_wen),
        .wdata  (RF_wdata),
        .rdata1 (wire_RF_rdata1),
        .rdata2 (wire_RF_rdata2)
);

assign RF_waddr = rd;
```

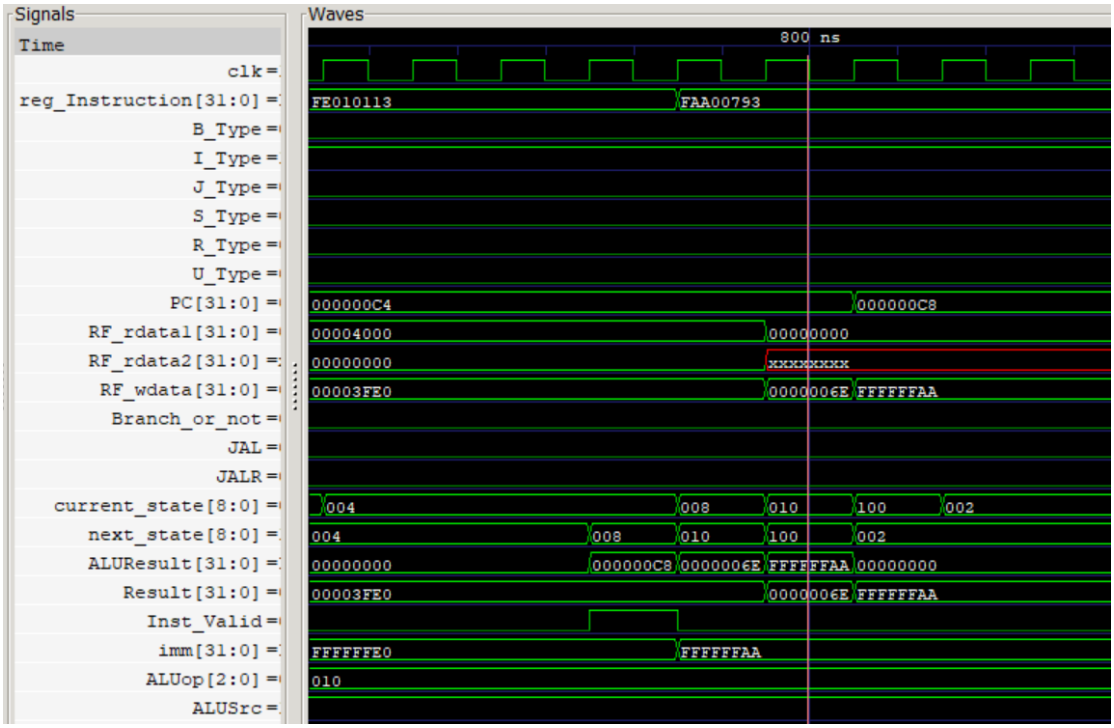
RISCV 的寄存器堆的写地址比较简单，所有需要实现的指令都是 rd。

2) 波形图示例

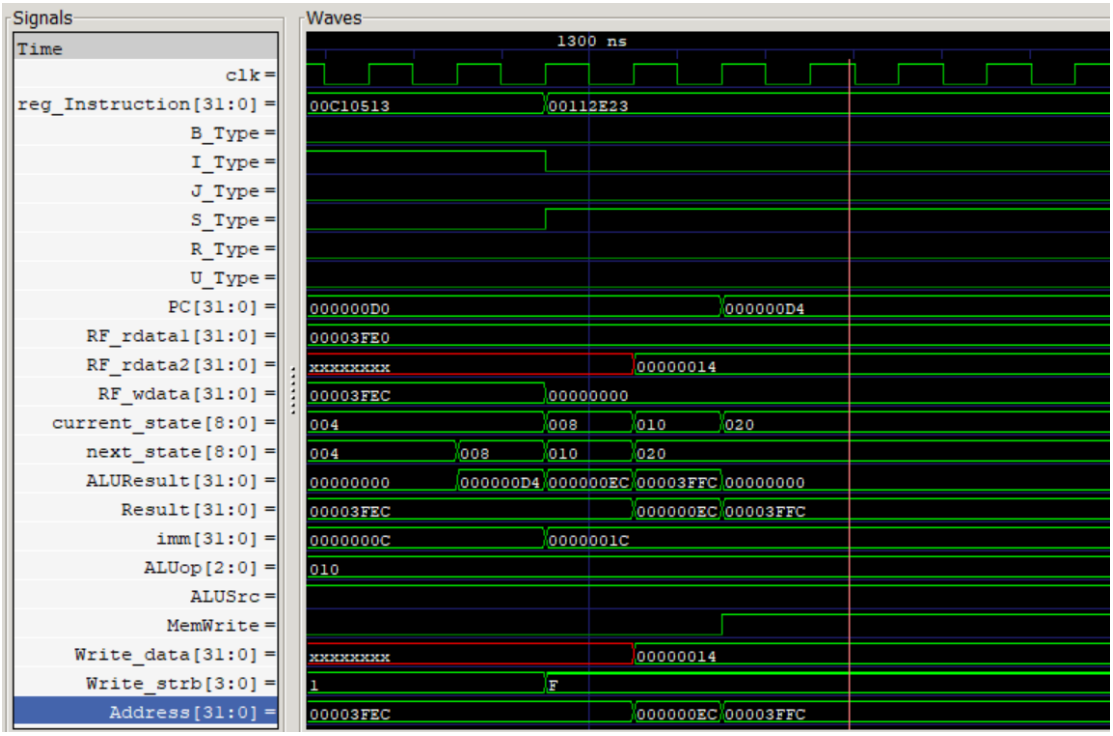


如上述图中，指令编码对应的 JAL 指令，而目标地址就是 00000010，且 J_Type 信号拉高，表明译码正确。最后执行阶段所更新的 PC 的值为 00000010，符合；而最后 RF_wdata 为 00000008，正是 PC+4 的值，表明执行与写回正确。

同时上图很好的体现了 ALU 的复用，如图，当 current_state 为 IW 且 Inst_Valid 为 1 时，ALU 算出了 PC+4 的值 00000008，然后当 current_state 为 ID 时，ALU 算出 PC 目标跳转地址 00000010，且存在了 Result 里面，



如上述图中，指令编码对应的 ADDI 指令，ALUSrc 拉高（表明 ALU 需要传入立即数），立即数扩展算出来的结果是 FFFFFFAA，寄存器读出的数据是 00000000，ALUop 译码出来是 010（加法），最后 Result 的值为 FFFFFFAA，符合，且 I_Typ 信号拉高，均正确。



如上述图中，指令编码对应的 SW 指令，S_Type 拉高，表明译码正确。寄存器堆读出的数据是 00003FE0 和 00000014，其中 00003FE0 用于计算写入地址，00000014 作为写入数据。

立即数扩展后的数为 0000001C，ALUop 为 010（加法），最后 ALU 加法算出来的写入地址为 00003FFC，符合。最后 Write_data 为 00000014，符合，Address 为 00003FFC 也符合，Write_strb 为 F（即 1111），满足。所以该指令执行正确。

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）

本次实验的整体逻辑与上个实验类似，但需要重新实现 RISCV 指令集的指令。有了前几次实验的经验，这次实验完成的还算顺利。但整个过

程中仍然有不少的难点，也花费了一定的时间，而我认为主要的难点如下：

难点：

1. 基于 RISCV 指令集的指令类型与控制信号的译码逻辑
2. ALUop 的编码逻辑
3. 复用 ALU 计算跳转和分支指令目标地址，而不用额外的加法器

对难点的思考：

1. 基于 RISCV 指令集的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 PPT 给了一定的提示，且 RISCV 的指令较为工整，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了本人一定的时间，比如如何对指令类型进行进一步细分，从而是执行部分的逻辑更加清楚简洁（比如 I_Type 包括了 LOAD 类型指令和 JALR 指令，但其与其它的 I_Type 指令需要完成的操作有很大的差别，所以本人在 I_Type 基础了，将其单独译码出来，方便处理)。

控制信号其实与 MIPS 逻辑相似，本人在这次实验中仍然是先根据 PPT 上的提示以及指令集手册，译码出指令的类型，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. ALUop 的编码逻辑

由于本次实验中，PPT 并没有给出具体的 ALUop 的编码逻辑，所以这部分也是本次实验一个很棘手的点。本人是首先将指令类型进行一次大的分类，然后在每个分类里面分别进行编码。而进行编码的时候，首先是理解每条指令需要的 ALUop 编码，然后分析不同操作的指令 funct3 和 funct7 的不同之处。如果简单就可以直接写出来，否则则利用卡诺图进行辅助。本人在编码过程中是对每一位进行编码，然后连接起来，这样可以方便进行卡诺图处理。

3. 复用 ALU 计算跳转和分支指令目标地址，而不用额外的加法器

这个本人认为也是本次实验的一个难点，不过大致逻辑跟 MIPS 多周期处理器设计相似，但对于 RISCV 指令集，不仅仅分支指令会进行加法操作，跳转指令也会进行加法操作，而且 AUIPC 指令也需要进行加法操作，但其也需要 PC 作为输入。所以为了不使用额外的加法器，实现 ALU 复用也要比 MIPS 更加麻烦。

要实现 ALU 复用，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 需要在 ID 阶段无条件拉高，ALUop 在 IW 且 Inst_Valid 为 1 和 ID 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 或者 RF_rdata1 和对应的扩展立即数 imm，而执行阶段也需要对

AUIPC 指令单独处理一下（这部分在前文对 ALU 的讲解中以及讲的很详细了，这里不再赘述了）。

遇到的重要 bug 以及解决过程：

1. 复用 ALU 时，ALUEn 和 ShiftEn 逻辑错误：

解决过程：ALUEn 和 ShiftEn 是对 ALU 和 Shifter 结果进行选择的信号，但是本人在复用 ALU 的过程中，在 ID 阶段没有把 ALUEn 拉高，ShiftEn 拉低，导致出错。最后在发现出错后，在 ALUEn 和 ShiftEn 的生成逻辑中，增添根据当前状态是否为 ID 而拉高或者拉低。

2. 复用ALU时,JALR目标地址计算出错(错误地将RF_data1的值传给了 ALU 的 A 接口)

解决过程： 复用 ALU 时，针对 JALR 指令，本人错误地将 RF_data1 的值传给了 ALU 的 A 接口，但此时 RF_data1 的值并没有被更新（下个周期来了才更新），所以最后把接入 A 接口的数据改成了 wire_RF_data1,这样由于寄存器堆是异步读出，所以就能正确计算目标地址了。

3. 误以为和 MIPS 一样具有分支延迟槽，导致 PC 更新逻辑出错

解决过程：由于刚开始没有认真阅读指令集手册,将 MIPS

处理器中的 PC 更新逻辑部分用在了这次实验上，导致分支指令目标地址计算出错（多了一个 4），后面阅读指令集手册，发现原因后将 PC 更新统一放在了 EX 阶段，问题得到解决。。

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 ALUop 编码逻辑出错，ALUEn 编码逻辑出错， ALU 的 A、B 接口在不同阶段的值出错等。对于这些错误主要是通过研究波形图与 RISC-V 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。

三、 对讲义中思考题（如有）的理解和回答

1．RISC-V/MIPS 指令集性能分析对比

a. RISC-V 指令集较 MIPS 指令集更规整

例如 RISC-V 指令中立即数的生成复用了各类型指令共有的部分,相较 MIPS 少用了一些数据选择器,成本更低。由此带来了更为简单优美的硬件电路设计。

比如，寄存器堆的写回地址，RISC-V 都为 rd 字段，且立即数扩展均为符号位扩展，省去了多路选择器进行选择。再比如，RISC-V 没有非对齐的写指令和读指令，这样使得访存逻辑变得比 MIPS 简单许多，少了很多的数据选择器。

再例如 opcode 统一在指令的末 7 位，立即数的最高位都在

指令的最高位，编码也更适合进行符号扩展，以及操作数的位置相对固定，这些都为译码带来了更多的便利。

b. RISC-V32 指令集未包含非对齐访存指令，编码正交性更高

RISC-V32 指令集不支持非对齐访存指令,这使得对编码空间的利用更高效。而 MIPS 指令集包含了非对齐访存指令，这使得其硬件电路较为复杂，会用到较多的选择器以及移位，性能较差。与此同时,RISC-V 由于不支持非对齐访存,也使得其编码的正交性更高。

c. RISC-V 指令集取消了分支延迟槽的设计，使得总指令数一般较 MIPS 集少

分支延迟槽主要用于提高多周期 CPU 流水线的性能。考虑到我们目前实现的多周期 CPU 都并不是流水结构,分支延迟槽并不能带来性能的提升,反而造成了汇编代码的冗长,带来不必要的指令开销（编译器往往会增添一些无意义的 nop 指令，从而使得指令条数变多，运行周期数边长）

d. RISC-V 指令集中的分支指令没有大于跳转和小于等于跳转，硬件实现上更为简洁

RISC-V 指令集中的分支指令不支持大于跳转和小于等于跳转，这与 pj1 的 ALU 设计更加吻合，从而针对 pj1 的 ALU 设计，更方便实现分支条件的判断。而 MIPS 支持 blez 和 bgtz，这除了

需要考虑 ALU 的 Zero 输出，还需要考虑寄存器读出的数据是否为全 0，从而电路更加复杂。

e. 此外, MIPS 具有固定的 16 位和 32 位编码,而对于 RISC-V 而言,其被设置为模块化的,尽管以 32 位为基准,但仍有 16 位、64 位的变种,同时人们正在寻找 128 位（用于百亿分之一的计算）的扩展，这为 RISC-V 带来了更多的可能性与更广阔的未来。

总之，整体对 RISC-V 的实现下来，RISC-V 的硬件设计确实比 MIPS 要简单一些，所以本人猜测其性能相较于 pj3 实现的 MIPS 要好上一点。综合以上几点和这几次实验的实现过程，就我个人而言,我更加喜欢 RISC-V。而下面则是同一程序在 MIPS 和 RISC-V 指令集上运行的性能计数器的结果:

I. ssort 程序:

MIPS 运行结果:

```
[ssort] Suffix sort: * Passed.
Result:
        Cycles: at least 52493029
        Instruction Count: at least 728307
        Memory Read Instruction Count: at least 11595
        Memory Write Instruction Count: at least 7711
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 48207856
        Memory Read Request Delay Cycles: at least 11599
        Read Data Delay Cycles: at least 782363
        Memory Write Request Delay Cycles: at least 7711
        Branch Instruction Count: at least 106130
        Jump Instruction Count: at least 693
benchmark finished
```

RISC-V 运行结果：

```
[ssort] Suffix sort: * Passed.
Result:
        Cycles: at least 44743001
        Instruction Count: at least 619044
        Memory Read Instruction Count: at least 11289
        Memory Write Instruction Count: at least 7426
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 40961428
        Memory Read Request Delay Cycles: at least 11294
        Read Data Delay Cycles: at least 764316
        Memory Write Request Delay Cycles: at least 7426
        Branch Instruction Count: at least 132631
        Jump Instruction Count: at least 869
benchmark finished
```

II. bf 程序：

MIPS 运行结果：

```
[bf] Brainf**k interpreter: * Passed.
Result:
        Cycles: at least 46345825
        Instruction Count: at least 559067
        Memory Read Instruction Count: at least 94508
        Memory Write Instruction Count: at least 5973
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 37004864
        Memory Read Request Delay Cycles: at least 94512
        Read Data Delay Cycles: at least 6462106
        Memory Write Request Delay Cycles: at least 5973
        Branch Instruction Count: at least 106688
        Jump Instruction Count: at least 32234
benchmark finished
```

RISC-V 运行结果：

```
[bf] Brainf**k interpreter: * Passed.
Result:
        Cycles: at least 38813686
        Instruction Count: at least 452853
        Memory Read Instruction Count: at least 94511
        Memory Write Instruction Count: at least 5974
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 29882315
        Memory Read Request Delay Cycles: at least 94516
        Read Data Delay Cycles: at least 6458206
        Memory Write Request Delay Cycles: at least 5974
        Branch Instruction Count: at least 91039
        Jump Instruction Count: at least 47906
benchmark finished
```

上面四张图的程序计数器，前两张是 ssort 程序运行结果，后两张是 bf 程序运行的结果。第 1.3 张是基于 MIPS 指令集，第 2.4 张是基于 RISC-V 指令集。可以看到,无论是 ssort 程序还是 sieve 程序,RISC-V 的总指令数较 MIPS 更少，且相应地,总周期数也较 MIPS 更少。从这便可以看出，RISC-V 的性能要比 MIPS 更好一点。

同时,还可以注意到, ssort 程序和 bf 程序除了周期数和指令数，其它性能寄存器的值都较为相似，而且可以注意到 Branch 指令数与 RISC-V 指令集下的指令数和 MIPS 指令集下指令数的差大概在同一水平。综合以上信息，可以合理猜测是由于分支延迟槽的取消，使得 Branch 指令后紧跟的延迟指令无需执行，由此带来了总指令数的减少，且减少量大致约为 Branch 指令的数量。

对于基于 MIPS 和 RISC-V 指令集所编译出来的汇编代码，我们可以发现，MIPS 中有较多的 nop 指令，且大都是在 Branch 指令后。但 RISC-V 由于没有分支延迟槽而少了很多。本人猜测，这应该也是 RISC-V 在某些程序上性能明显好于 MIPS 的一个重要原因。

但并非对于所有程序 RISC-V 的性能都比 MIPS 好，比如针对 fib 程序，其运行结果如下图所示：

III. fib 程序：

MIPS 运行结果:

```
[fib] Fibonacci number: * Passed.
Result:
        Cycles: at least 179410037
        Instruction Count: at least 2525740
        Memory Read Instruction Count: at least 4411
        Memory Write Instruction Count: at least 979
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 166893415
        Memory Read Request Delay Cycles: at least 4415
        Read Data Delay Cycles: at least 302473
        Memory Write Request Delay Cycles: at least 979
        Branch Instruction Count: at least 398164
        Jump Instruction Count: at least 5220
benchmark finished
```

RISC-V 运行结果：

```
[fib] Fibonacci number: * Passed.
Result:
        Cycles: at least 181090027
        Instruction Count: at least 2549524
        Memory Read Instruction Count: at least 4324
        Memory Write Instruction Count: at least 980
        Instruction Fetch Request Delay Cycle: at least 0
        Instruction Fetch Delay Cycles: at least 168610278
        Memory Read Request Delay Cycles: at least 4329
        Read Data Delay Cycles: at least 295249
        Memory Write Request Delay Cycles: at least 980
        Branch Instruction Count: at least 577540
        Jump Instruction Count: at least 12122
benchmark finished
```

可以看出，在 fib 程序下，MIPS 指令集下的指令数和周期数均 RISC-V 指令集下的指令数和周期数少，而进一步观察可以得知，RISC-V 指令集下的分支指令数要远大于 MIPS 指令集下的分支指令数。

针对这个现象，本人认为可能的原因是，RISC-V 指令集中 Branch 指令没有大于跳转和小于等于跳转，而 MIPS 中有 blez 和 bgtz，可能是因为这个原因导致 RISC-V 指令集下 fib 程序的 Branch 指令数较多，从而导致周期数和指令数都比 MIPS 多上一点。当时这一点本人也不是很确定。

四、 在课后，你花费了大约___20__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。由于之前写过基于 MIPS 的 CPU，加上 RISC-V 指令集本身易于实现的特性，本次实验整体花费时间并不长。但 RISC-V 指令手册的可读性比起 MIPS 较差。比如指令手册就没有给出公式化的指令实现描述。这导致需要投入较多精力研究 RISC-V 标准的具体含义。而且实验课的 PPT 对于 RISC-V 指令集的讲解相较于 pj2 也不算特别多（当然，到了实验四，确实不需要讲的太具体了），所以弄清楚各个指令的含义还是花费了一定的时间。

通过这次实验,我受益匪浅：一方面，对 RISC-V 指令集有了更深的了解，对 RISC 和 CISC 指令集的区别有了更深的理解。另一方面，通过这次实验，我进一步熟悉了状态机的三段式描述，对 verilog 语法也更加熟悉。

最后，感谢助教在定制 RISC-V 功能处理器的设计和实现上的启发与帮助。