中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2017K8009929044 姓名： 李昊宸 专业： 计算机科学与技术

实验序号：5 实验名称： RISC-V 指令集处理器

注 1：请在实验项目个人本地仓库中创建顶层目录 doc。撰写此 Word 格式实验报告后以 PDF 格式保存在 doc 目录下。文件命名规则：学号-prjN.pdf，其中学号中的字母“K”为大写，“-”为英文连字符，“prj”和后缀名“pdf”为小写，“N”为 1 至 5 的阿拉伯数字。例如：2015K8009929000-prj1.pdf。PDF 文件大小应控制在 5MB 以内。

注 2：使用 git add 命令将 doc 目录下的实验报告 PDF 文件添加到本地仓库，然后 git push 推送提交。

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、相应信号的仿真波形和信号变化的说明等）

三段式状态机：

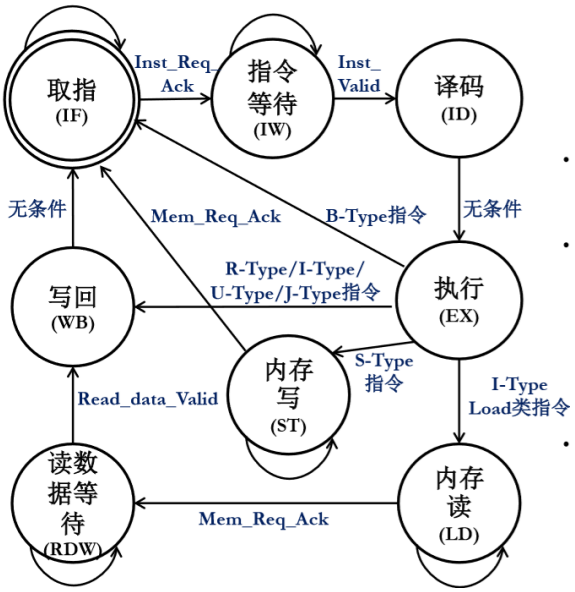
第一段：时序逻辑控制状态跳转

```
always@(posedge clk)                //control unit
if(rst)                 state <= 5'd0;   //IF
else
                        state <= nextstate;
```

三段式的第一段叙述跳转条件为每一时钟上升沿触发。

第二段：组合逻辑控制 nextstage 跳转信号

```
always@(*)
begin
case(state)
5'd0:               if(Inst_Req_Ack) nextstate = 5'd1;   //IW
                    else             nextstate = 5'd0;
5'd1:               if(Inst_Valid)   nextstate = 5'd7;   //ID
                    else             nextstate = 5'd1;
5'd7:                                nextstate = 5'd2;   //EX
5'd2:               if(op == 7'b1100011)     nextstate = 5'd0; //B-type
                    else if(op == 7'b0000011)   nextstate = 5'd3; //I-type load
                    else if(op_SB||op_SH||op_SW)                  nextstate = 5'd4; //S-type store
                    else                                                  nextstate = 5'd5; //Write reg_file WB
5'd3:               if(Mem_Req_Ack)  nextstate = 5'd6; //RDW
                    else             nextstate = 5'd3;
5'd4:               if(Mem_Req_Ack)  nextstate = 5'd0;
                    else             nextstate = 5'd4;
5'd5:               nextstate = 5'd0;
5'd6:               if(Read_data_Valid)  nextstate = 5'd5;
                    else             nextstate = 5'd6;
default:            nextstate = 5'd0;
endcase
end
```



状态机的状态转移表写在状态机的第二段中。采用握手原理，对应路径的 Valid 和 Ack 信号同时拉高时进行数据传输完成的确认，跳转至下一状态。

此外，与实验 4 不同的是，riskv 的指令集编码系统非常方便指令类型的译码判定，具有很良好的书写体验。建议以后实验先进行 riscv 的设计，再进行 mips 的设计。

第三段：按当前状态机所处状态对各信号赋值。

由于本实验没有 reg 型寄存器的使用，赋值部分全部采用 assign 阻塞赋值，故将 always 块省略。下面是与单周期处理器中有修改的控制信号的 RTL 代码。

```
assign MemRead  =   (state == 5'd3)?1:0;    //LOAD
```

MemRead 在 LD 状态被拉高

```
assign MemWrite =   (state == 5'd4)?1:0; //STORE
```

MemWrite 在 ST 状态被拉高

```
assign RegWrite =   (state == 5'd5)?1:0;
```

RegWrite 在 EX 段被拉高

```
assign Inst_Req_Valid = (RST)?0:(state == 5'd0)?1:0;
assign Inst_Ack       = (RST)?1:(state == 5'd1)?1:0;
assign Read_data_Ack  = (state == 5'd6)?1:0;
```

三个控制信号，分别表示指令请求有效信号，指令接收信号，内存数据读取接收信号。其中为了防止出现“锁死”，在 rst 归零信号拉高时 Inst_Reg_Valid 信号

需要持续为低，Inst_Ack 需要持续拉高。另外，为了代码规范化，在赋值逻辑

中没有直接出现 rst 信号，取而代之用另一信号 RST 代替之。

```
        assign ALUSrc    =   (op_AUIPC||op_JAL||op_JALR||op_LB||op_LH||op_LW||
op_LBU||op_LHU||op_SW||op_SB||op_SH||op_ADDI||op_SLTI||op_SLTIU||op_XORI||op_ORI||
op_ANDI)?1:0;
```

ALUSrc 为第二计算信号来源代码

```
 assign MemToReg =   (op == 7'b0000011)?1:0;
```

MemToReg 为 load 类型指令码

```
 assign Branch   =   (op == 7'b1100011)?1:0;
```

Branch 为 branch 类型指令码

值得一提的是，在 riskv 架构中，取消了延迟槽的设计。

```
        assign ALUOp     =   (op_LUI||op_AUIPC||op_JAL||op_JALR||(op ==
7'b0000011)||(op == 7'b0100011)||op_ADDI||op_ADD)?3'b010:
                             (op_BNE||op_BEQ||op_SUB)?3'b110:

                             (op_SRAI||op_SRA)?3'b100:


                             (op_ORI||op_OR)?3'b001:

                             (op_SLT||op_SLTI||op_BLT||op_BGE)?3'b111:

                             (op_SLTIU||op_SLTU||op_BLTU||op_BGEU)?3'b101:

                             (op_XORI||op_XOR)?3'b011:

                             3'b000;
```

ALUOp 控制信号

```
 assign Write_strb =
 (op_SB)?
                    (Result[1:0]==2'b00)?4'b0001:
                    (Result[1:0]==2'b01)?4'b0010:
                    (Result[1:0]==2'b10)?4'b0100:
                    (Result[1:0]==2'b11)?4'b1000:0
 :(op_SH)?
                    (Result[1]==1'b0)?4'b0011:
                    (Result[1]==1'b1)?4'b1100:0


 :4'b1111;
```

写入逻辑，对应写入内存时的数据位置。与 mips 相同。

```
always@(posedge clk)
begin
if(rst)
PC<=32'b0;
else if(state == 5'd2) PC<=PC2;
else PC <= PC;
end

always@(posedge clk)
begin
if(rst)
p <= 32'd0;
else if(state == 5'd7) p <= PC;
else p <= p;
end

assign PC3 = PC + 32'd4;
assign PCJUMP = (op_JAL||op_JALR)?{Result[31:1],1'b0}:(PC+Sign_extend);


assign PC2 = (op_JAL||op_JALR||(op_BEQ&&Zero)||(op_BNE&&!Zero)||((op_BLT||
op_BLTU)&&!Zero)||((op_BGE||op_BGEU)&&Zero))?PCJUMP:PC3;
```

时钟时序逻辑：

rst 为高时持续为零，在状态机进行到 EX 状态后跳转至下一指令地址。

```
always@(posedge clk)
if(rst)
instruction = 32'd0;
else if(Inst_Valid && Inst_Ack)
instruction = Instruction;
//else instruction <= instruction;

always@(posedge clk)
if(rst)
Read_data_continue = 32'd0;
else if(Read_data_Valid && Read_data_Ack )
Read_data_continue = Read_data;
//else Read_data_continue <= Read_data_continue;
```

两个用于存放当前指令信号和内存读取信号的寄存器。由于这两个信号只会存在一个时钟周期，但其数据需要在整个指令执行期间被使用，所以设置这两个寄存器是有必要的。

主要的运算逻辑：

```
reg_file reg_file1(clk,rst,RF_waddr,rs1,rs2,RF_wen,RF_wdata,data1,data2);

assign RF_waddr = rd;
```

```
        assign Sign_extend = (op_LUI||op_AUIPC)?{instruction[31:12],12'd0}:
                             (op_JAL)?(instruction[31]==1'b1)?
{11'b11111111111,instruction[31],instruction[19:12],instruction[20],instruction
[30:21],1'd0}:
                                                                {11'd0,instruction
[31],instruction[19:12],instruction[20],instruction[30:21],1'd0}:
                             (op_JALR||(op == 7'b0000011)||op_ADDI||op_SLTI||
op_SLTIU||op_XORI||op_ORI||op_ANDI)?(instruction[31]==1'b1)?{20'hfffff,instruction
[31:20]}:

{20'd0,instruction[31:20]}:
                             (op == 7'b1100011)?(instruction[31]==1'b1)?
{19'h7ffff,instruction[31],instruction[7],instruction[30:25],instruction
[11:8],1'b0}:

{19'd0,instruction[31],instruction[7],instruction[30:25],instruction
[11:8],1'b0}:

                             (instruction[31]==1'b1)?{20'hfffff,instruction
[31:25],instruction[11:7]}:
                                                     {20'd0,instruction
[31:25],instruction[11:7]};
```

riscv 有着令人着迷的立即数处理方式，（据说）这样设计和硬件调用优化有关。

```
        assign data = (op_JAL)?PC:(op_AUIPC)?p:data1;
        assign Mux_ALU = (op_SRAI)?rs2:(ALUSrc)?Sign_extend:data2;

        alu alu1(data,Mux_ALU,ALUOp,Overflow,CarryOut,Zero,Result);

        assign Address = (op_LB||op_LBU||op_LH||op_LHU||op_SB||op_SH)?{Result
[31:2],2'b00}:Result;

        assign Write_data =
        (op_SB)?
        (Write_strb == 4'b0001)?{24'd0,data2[7:0]}:
        (Write_strb == 4'b0010)?{16'd0,data2[7:0],8'd0}:
        (Write_strb == 4'b0100)?{8'd0,data2[7:0],16'd0}:
        (Write_strb == 4'b1000)?{data2[7:0],24'd0}:0
        :(op_SH)?
        (Write_strb == 4'b0011)?{16'd0,data2[15:0]}:
        (Write_strb == 4'b1100)?{data2[15:0],16'd0}:0

        :data2;

        assign RF_wen = RegWrite;

       assign RF_wdata =
       (op_SLLI)?(data1<<rs2)
       :(op_SLL)?(data1<<{data2[4:0]})
       :(op_SRLI)?(data1>>rs2)
       :(op_SRL)?(data1>>{data2[4:0]})
       //:(op_SRAI)?
       //:(op_SRA)?


       :(op_LB||op_LBU||op_LH||op_LHU)?Rf_wdata

       :(op_JAL||op_JALR)?p+32'd4
       :(op_LUI)?{instruction[31:12],12'd0}
       :(MemToReg)?Read_data_continue
       :Result;
```

```
        assign Rf_wdata = (op_LB)?
                          (Result[1:0]==2'b00)?(Read_data_continue[7]==1)?
{24'hffffff,Read_data_continue[7:0]}:{24'd0,Read_data_continue[7:0]}:
                          (Result[1:0]==2'b01)?(Read_data_continue[15]==1)?
{24'hffffff,Read_data_continue[15:8]}:{24'd0,Read_data_continue[15:8]}:
                          (Result[1:0]==2'b10)?(Read_data_continue[23]==1)?
{24'hffffff,Read_data_continue[23:16]}:{24'd0,Read_data_continue[23:16]}:
                          (Result[1:0]==2'b11)?(Read_data_continue[31]==1)?
{24'hffffff,Read_data_continue[31:24]}:{24'd0,Read_data_continue[31:24]}:0
                          :(op_LBU)?
                          (Result[1:0]==2'b00)?$unsigned(Read_data_continue[7:0]):
                          (Result[1:0]==2'b01)?$unsigned(Read_data_continue[15:8]):
                          (Result[1:0]==2'b10)?$unsigned(Read_data_continue
[23:16]):
                          (Result[1:0]==2'b11)?$unsigned(Read_data_continue
[31:24]):0
                          :(op_LH)?
                          (Result[1]==1'b0)?(Read_data_continue[15]==1)?
{16'hffff,Read_data_continue[15:0]}:{16'd0,Read_data_continue[15:0]}:
                          (Result[1]==1'b1)?(Read_data_continue[31]==1)?
{16'hffff,Read_data_continue[31:16]}:{16'd0,Read_data_continue[31:16]}:0    //
CONCERNNED!
                          :(op_LHU)?
                          (Result[1]==1'b0)?$unsigned(Read_data_continue[15:0]):
                          (Result[1]==1'b1)?$unsigned(Read_data_continue
[31:16]):0


                          :0;
```

整个过程和 mips 基本一致，在此不过多叙述。

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，仿真、本地上板及云平台调试过程中的难点等）

在该实验中遇到的主要问题是 RTL 代码的可综合性与硬件的支持性。比如在该实验中，时序电路的代码书写经常出现的问题是仿真波形正确，bit_stream 正确生成，查看 log 文件也没有错误，但是上板全部 bad trap。经分析后发现可能是 verilog 会将 always 过程块内阻塞赋值的自赋值行为看作可综合，但是在硬件逻辑优化中可能会将其优化删除，这导致了在硬件上该部分逻辑出现时数据信号没有自保存导致错误，这些部分在网上没有查阅到相关资料，为代码的书写和纠错造成了极大的麻烦，

三、 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）

作为最后一个实验，riskv 确实在已经写过 mips 的基础上实现的较快。还是想感慨一下，riskv 的编码真是太舒服了！强烈建议明年课程先写 riskv，把 mips 放进选作。