中国科学院大学计算机组成原理实验课

实 验 报 告

学号： 2020K8009926006 姓名： 游昆霖 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

注1：撰写此Word格式实验报告后以PDF格式保存在~/COD-Lab/reports目录下。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。

1. 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、相应信号的仿真波形和信号变化的说明等）

说明：除custom\_cpu.v外软硬件代码均可复用prj3，同时逻辑电路图也与mips处理器的基本一致，以下只对custom\_cpu.v中针对RISC-V指令集进行的改动进行说明。

（1）状态机第二部分

|  |
| --- |
| //Part II: Combinatorial logic          always @ (\*)          begin                  case(current\_state)                          RST: begin                                  next\_state = IF ;                          end                          IF:     begin                                  if (Inst\_Req\_Ready)                                          next\_state = IW;                                  else                                          next\_state = IF;                          end                          IW: begin                                  if (Inst\_Valid)                                          next\_state = ID;                                  else                                          next\_state = IW;                          end                          //NOP: Instr = 32'b0                          ID: begin                                  next\_state = EX ;  //no need to care NOP                          end                          //EX->IF:                                  //B\_type                          //EX->WB                                  //R\_type / I\_calc / I\_jalr / U\_type / J\_type                          //EX->LD                                  //I\_load                          //EX->ST                                  //S\_type                          EX: begin                                  if(B\_type)                                          next\_state = IF;                                  else if(R\_type | I\_calc | I\_jalr | U\_type | J\_type )                                          next\_state = WB;                                  else if(I\_load)                                          next\_state = LD;                                  else if(S\_type)                                          next\_state = ST;                                  else                                          next\_state = RST;                          end                          LD: begin                                  if(Mem\_Req\_Ready)                                          next\_state = RDW;                                  else                                          next\_state = LD;                          end                          ST: begin                                  if(Mem\_Req\_Ready)                                          next\_state = IF;                                  else                                          next\_state = ST;                          end                          RDW: begin                                  if(Read\_data\_Valid)                                          next\_state = WB;                                  else                                          next\_state = RDW;                          end                          WB: begin                                  next\_state = IF;                          end                          default: begin                                  next\_state = RST;                          end                  endcase          end |

状态机的状态与prj3相同，跳转条件有所区别，但对应的指令功能基本一致。值得注意的是，本实验中RISC-V指令集无条件跳转指令(Jal和Jalr)均需要进行写寄存器操作，且ID到EX状态转移不再需要考虑为非NOP指令（RISCV中的空指令用addi x0,x0,0实现）。  
  
 （2）PC更新逻辑

|  |
| --- |
| always @ (posedge clk) begin                  if(current\_state[1])//IF                          PC\_tmp <= PC;          end            always @ (posedge clk) begin                  if (rst) begin                          PC <= 32'b0;                  end                  else if(current\_state[2] & Inst\_Valid & ~rst) begin                          PC <= ALU\_Result; //PC+4 IW, consider the cycle before ID                  end                  else if(current\_state[4] & ~rst)begin                          if (br\_en)                                  PC <= ALU\_Result\_tmp; //br\_tar                          else if(j\_en)                                  PC <= j\_tar;                          else                                  PC <= PC;                          //PC <= br\_en ? br\_tar : j\_en ? j\_tar : PC ;                          //j or br OP refresh in next IF, judge by EX                          //note that the default result is PC4                  end          end |

虽然RISCV不使用分支延迟槽，因此求PC条件跳转地址的时候使用的是原PC值，但为提高ALU复用率及操作简便性，仍在IW阶段将PC进行+4，后续求条件跳转地址时使用PC的寄存值PC\_tmp作为操作数。  
  
 （3）译码部分

|  |
| --- |
| //Analyse Instruction code          //imm\_extension:                  //I\_type: unsigned: SLTIU(011)  signed：other OP  shamt: shift   note that I\_jalr: {rs1+signed(offset)}[31:1],0                  //S\_type: signed                  //B\_type: signed  in multiples of 2                  //U\_type: fill low 12 bit with 0                  //J\_type: signed  in multiples of 2          assign opcode   = Instruction\_tmp[6:0];          assign rd       = Instruction\_tmp[11:7];          assign rs1      = Instruction\_tmp[19:15];          assign rs2      = Instruction\_tmp[24:20];          assign shamt    = Instruction\_tmp[24:20];          assign funct3   = Instruction\_tmp[14:12];          assign funct7   = Instruction\_tmp[31:25];          assign I\_imm[11:0]  = Instruction\_tmp[31:20];          assign I\_imm[31:12] = funct3 == 3'b011 ? {20'b0} : {20{Instruction\_tmp[31]}};          assign S\_imm    = { {20{Instruction\_tmp[31]}}, Instruction\_tmp[31:25], Instruction\_tmp[11:7]};          assign B\_imm    = { {19{Instruction\_tmp[31]}}, Instruction\_tmp[31], Instruction\_tmp[7], Instruction\_tmp[30:25], Instruction\_tmp[11:8], 1'b0};          assign U\_imm    = { Instruction\_tmp[31:12],12'b0};          assign J\_imm    = { {11{Instruction\_tmp[31]}}, Instruction\_tmp[31], Instruction\_tmp[19:12], Instruction\_tmp[20], Instruction\_tmp[30:21], 1'b0};    //Differ type by one-bit signals          assign  R\_type  = opcode == 7'b0110011;          assign  I\_calc  = opcode == 7'b0010011;          assign  I\_load  = opcode == 7'b0000011;          assign  I\_jalr  = opcode == 7'b1100111;          assign  I\_type  = I\_calc | I\_load | I\_jalr;          assign  S\_type  = opcode == 7'b0100011;          assign  B\_type  = opcode == 7'b1100011;          assign  U\_lui   = opcode == 7'b0110111;          assign  U\_auipc = opcode == 7'b0010111;          assign  U\_type  = U\_lui | U\_auipc;          assign  J\_type  = opcode == 7'b1101111; |

由于RISCV指令集更为简洁齐整，操作数位置较为固定，且分类可固定由最低7位决定。因此在译码部分获得指令类别，操作数内容，以及不同类别指令对应的立即数拓展。

（4）ALU部分

|  |
| --- |
| //Channel to ALU          //IW stage: use add to save adder of PC+4          //ID stage: use add to save adder of PC\_tmp+B\_imm          //WB stage: use add to save adder of PC\_reg+4 by I\_jalr and J\_type                  //note that ALU is also used in EX\_stage to get PC+imm by J\_type or rs1+imm by I\_jalr          //Operations related to ALU:          //TYPE                          op1     op2     OP          //R(exclude shift)              rs1     rs2     ...          //I\_calc(exclude shift)         rs1     I\_imm   ...          //I\_load I\_jalr                 rs1     I\_imm   ADD          //S\_type                        rs1     S\_imm   ADD          //B\_type                  //BEQ,BNE               rs1     rs2     SUB                  //other                 rs1     rs2     SLT/SLTU          //U\_auipc                       PC\_tmp  U\_imm   ADD          //J\_type                        PC\_tmp  J\_imm   ADD            assign ALU\_op\_origin =  ({3{R\_type & ~(funct3==3'b001 | funct3 == 3'b101)}} & ( {3{funct3 == 3'b000}}     & {funct7[5],2'b10}   |                                                                                          {3{funct3[2:1] == 2'b01}} & {~funct3[0],2'b11}  |                                                                                          {3{funct3 == 3'b100}}     & funct3              |                                                                                          {3{funct3[2:1] == 2'b11}} & ~funct3             )                                  ) |                                  ({3{I\_calc & ~(funct3==3'b001 | funct3 == 3'b101)}} & ( {3{funct3 == 3'b000}}     & 3'b010              |                                                                                          {3{funct3[2:1] == 2'b01}} & {~funct3[0],2'b11}  |                                                                                          {3{funct3 == 3'b100}}     & funct3              |                                                                                          {3{funct3[2:1] == 2'b11}} & ~funct3             )                                  ) |                                  ({3{I\_load | I\_jalr | S\_type | U\_auipc | J\_type }}  &  3'b010 ) |                                  ({3{B\_type}}                                        & ( funct3[2:1]==2'b00 ? 3'b110 : {~funct3[1], 2'b11}) );          always @(posedge clk) begin                  //if(current\_state[3]) //ID                          ALU\_op\_tmp <= ALU\_op\_origin;                  // else if(current\_state[4] & (I\_jalr | J\_type)) //EX before WB                  //         ALU\_op\_tmp <= 3'b010;          end          assign ALU\_op\_final =   current\_state[2] | current\_state[3] ? 3'b010 :                                  current\_state[8] & (I\_jalr | J\_type) ? 3'b010 :                                  ALU\_op\_tmp;            assign ALU\_A\_origin =   U\_auipc | J\_type ? PC\_tmp :                                  RF\_rdata1;          always @(posedge clk) begin                  //if(current\_state[3]) //ID                          ALU\_A\_tmp <= ALU\_A\_origin;                  // else if(current\_state[4] & (I\_jalr | J\_type)) //EX before WB                  //         ALU\_A\_tmp <= PC\_tmp;          end          assign ALU\_A\_final =    current\_state[2]  ? PC :                                  current\_state[3]  ? PC\_tmp :                                  current\_state[8] & (I\_jalr | J\_type) ? PC\_tmp :                                  ALU\_A\_tmp;          assign ALU\_B\_origin =   {32{(I\_calc & ~(funct3==3'b001 | funct3 == 3'b101)) | I\_load | I\_jalr}}  & I\_imm        |                                  {32{S\_type}}                                                             & S\_imm        |                                  {32{U\_auipc}}                                                            & U\_imm        |                                  {32{J\_type}}                                                             & J\_imm        |                                  {32{R\_type & ~(funct3==3'b001 | funct3 == 3'b101) | B\_type}}             & RF\_rdata2    ;          always @(posedge clk) begin                  //if(current\_state[3]) //ID                          ALU\_B\_tmp <= ALU\_B\_origin;                  // else if(current\_state[4] & (I\_jalr | J\_type)) //EX before WB                  //         ALU\_B\_tmp <= 32'd4;          end          assign ALU\_B\_final =    current\_state[2] ? 32'd4 :                                  current\_state[3] ? B\_imm :                                  current\_state[8] & (I\_jalr | J\_type) ? 32'd4 :                                  ALU\_B\_tmp;          always @(posedge clk) begin                          ALU\_Result\_tmp <= ALU\_Result;          end          alu alu\_inst(             .A(ALU\_A\_final),             .B(ALU\_B\_final),             .ALUop(ALU\_op\_final),             .Overflow(ALU\_Overflow),             .CarryOut(ALU\_CarryOut),             .Zero(ALU\_Zero),             .Result(ALU\_Result)          ); |

EX阶段各类型指令对应操作数及操作类型如注释所示，即得ALU\_XX\_origin，将其寄存得到了ALU\_XX\_tmp。由于在IW阶段复用ALU得到PC+4，ID阶段得到PC\_tmp+B\_imm，WB阶段得到PC\_tmp+4。设计数据旁路，并以状态进行选择寄存值和旁路数据，从而得到与ALU相连的信号ALU\_XX\_final。

（5）Shifter部分

|  |
| --- |
| //Channel to Shifter          //Operations related to Shifter          //R\_type shift  rs1 rs2          //I\_calc shift  rs1 shamt          assign Shifter\_op = (R\_type | I\_calc) & (funct3==3'b001 | funct3 == 3'b101) ? {funct3[2],funct7[5]} : 0;          assign Shifter\_A  = RF\_rdata1;          assign Shifter\_B  = R\_type ? RF\_rdata2[4:0] : shamt;          always @(posedge clk)begin                  //if(current\_state[3]) //ID                          Shifter\_op\_tmp <= Shifter\_op;                          Shifter\_A\_tmp  <= Shifter\_A;                          Shifter\_B\_tmp  <= Shifter\_B;          end          always @(posedge clk)begin                  //if(current\_state[4])//EX                          Shifter\_Result\_tmp <= Shifter\_Result;          end          shifter shifter\_inst(             .A(Shifter\_A\_tmp),             .B(Shifter\_B\_tmp),             .Shiftop(Shifter\_op\_tmp),             .Result(Shifter\_Result)          ); |

由于RISCV中无swl和swr指令，不需复用Shifter得到写内存的数，考虑运算类型指令即可。

（6）Reg\_file部分  
 与MIPS逻辑基本一致，特别的，无条件跳转指令也需写寄存器。  
 （7）访存部分  
 减少了swl和swr指令，其余逻辑基本一致。  
 （8）性能计数器部分  
 与prj3使用的性能计数器相同，在更新条件处针对RISCV指令略作更改即可。

1. 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）

问题1：性能计数结果不符合预期  
 取值次数统计inst\_cnt和访存次数统计mem\_cnt远大于预期值，检查代码发现应当考虑内存延迟导致的状态停滞。添加对应ready信号使的仅统计该状态的最后一个周期，修改后性能计数结果符合预期。

**以下问题均通过硬件仿真工具发现：**

问题2：硬件仿真工具第一阶段报错  
 查看报错日志，显示next\_state产生了latch。将状态机补充default，if补充else，使得状态机覆盖所有情形即可。  
问题3：ALU\_B\_origin遗漏情形  
 仿真通过，但上板错误，使用硬件仿真加速工具得到产生错误的时钟周期，结合硬件仿真的金标准信号以及出错benchmark的反汇编文件，发现ALU\_B\_origin遗漏了I\_jalr和I\_load的情形。

1. 对讲义中思考题（如有）的理解和回答

MIPS和RISC-V性能对比：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 指令集 | 时钟周期数 | 指令总数 | 访存次数 | 内存延迟周期数 | 跳转次数 | 顺序更新次数 | CPI |
| 15pz | mips | 330924220 | 5287727 | 3231803 | 91736634 | 631860 | 4639604 | 62.58 |
| riscv | 324238832 | 5224477 | 3228762 | 90004486 | 625713 | 4598777 | 62.06 |
| bf | mips | 28358029 | 559065 | 100480 | 3987321 | 83994 | 420153 | 50.72 |
| riscv | 23775610 | 452850 | 100484 | 3956925 | 83991 | 368872 | 52.50 |
| dinic | mips | 1047512 | 19342 | 6525 | 198150 | 1190 | 17159 | 54.16 |
| riscv | 906059 | 16687 | 5566 | 171982 | 1277 | 15423 | 54.30 |
| fib | mips | 109644831 | 2525738 | 5389 | 188194 | 387757 | 2122364 | 43.41 |
| riscv | 110676382 | 2549521 | 5303 | 183886 | 478083 | 2071451 | 43.41 |
| md5 | mips | 249557 | 5243 | 577 | 20371 | 267 | 4661 | 47.60 |
| riscv | 236203 | 4911 | 642 | 22927 | 359 | 4565 | 48.10 |
| qsort | mips | 437705 | 8355 | 2463 | 66179 | 1415 | 6517 | 52.39 |
| riscv | 483070 | 9476 | 2464 | 65405 | 1415 | 8074 | 50.98 |
| queen | mips | 4240606 | 80872 | 26771 | 627369 | 6233 | 70042 | 52.44 |
| riscv | 4284106 | 81486 | 26772 | 637557 | 5589 | 75910 | 52.57 |
| sieve | mips | 737234 | 16494 | 472 | 14654 | 1671 | 14653 | 44.70 |
| riscv | 456038 | 10191 | 470 | 14335 | 1401 | 8803 | 44.75 |
| ssort | mips | 32122266 | 728305 | 19305 | 506651 | 62245 | 621492 | 44.11 |
| riscv | 27431823 | 619041 | 18714 | 514622 | 67292 | 551762 | 44.31 |
| 平均结果 | mips | 56417995.56 | 1025682.33 | 377087.22 | 10816169.22 | 130736.89 | 879627.22 | 50.23 |
| riscv | 54720902.56 | 996515.56 | 376575.22 | 10619125.00 | 140568.89 | 855959.67 | 50.33 |

由上表可知，从性能计数器统计结果上看，对于同一benchmark，RISCV指令集下，时钟周期数和指令数均略小于MIPS指令集，而CPI略大于MIPS指令集。

从代码实现和电路资源配置角度上看，RISCV指令集分类更加清晰简洁，相同的控制信号在不同类型指令中位置的重复率也较高，因此可以用更少的电路资源完成译码部分的电路配置。

1. 在课后，你花费了大约\_\_\_\_\_10\_\_\_\_\_\_小时完成此次实验。
2. 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）

心得感受：  
 在充分了解prj3中实现的MIPS型处理器的设计逻辑后，本次实验所需完成的RISCV型处理器难度不大，且由于RISCV指令集更加简洁规整的特点，在译码和相关控制信号上分类也更加清晰，容易减少潜在bug的产生。本次实验的重点还是在于体会不同指令集对应处理设计的异同，以及进行不同指令集的性能评估比较。  
  
建议：  
 在prj4实验中可以提高根据反汇编文件debug的能力要求，鼓励同学们减少对于金标准文件的依赖，增加自主检验错误能力。  
  
致谢：  
 感谢陈欲晓助教和芦溶民助教在仿真加速工具上的帮助，感谢常老师对于代码规范和优化的建议。