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：2 实验名称：简单功能型处理器设计

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。

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、相应信号的仿真波形和信号变化的说明等）

说明：单周期处理器部分设计共包含四个模块文件，其中 reg_file 直接复用 prj1 所完成即可，alu 需新增 3 个操作，同时新加入 shifter 和 simple_cpu 模块。多周期处理器在单周期的基础上新增了状态机和部分寄存器。以下只对处理器给出逻辑电路结构图和仿真波形说明。

1. alu.v 代码新增操作说明

```
//new ALUop
`define XOR  3'b100
`define NOR  3'b101
`define SLTU 3'b011
...
//new wire virables for direct results
   wire [`DATA_WIDTH -1 :0] R_XOR;
   wire [`DATA_WIDTH -1 :0] R_NOR;
```

```
    wire [`DATA_WIDTH -1 :0] R_SLTU;
...
    assign R_XOR = A ^ B;
    assign R_NOR = ~R_OR;
...
    assign Inverse = ALUop[1] & (ALUop[2] | ALUop[0]); //sub or slt or
sltu
    assign A_tmp = {1'b0,A};
    assign B_tmp = {1'b0, Inverse ? ~B : B } + Inverse ; //note that the
inverse result exclude the highest bit

//result of OP ADD / SUB / SLT
    assign {flag , R_ADD} = A_tmp + B_tmp;
    assign R_SLTU[`DATA_WIDTH-1 : 0] = {31'b0, ~flag};
```

新增的三个操作中 XOR 异或和 NOR 非或操作基于其语义进行操作即可。SLTU 无符号数比较操作只需将 A、B 增加 1 位表示为补码，进行减法操作，并根据结果符号位判断即可。

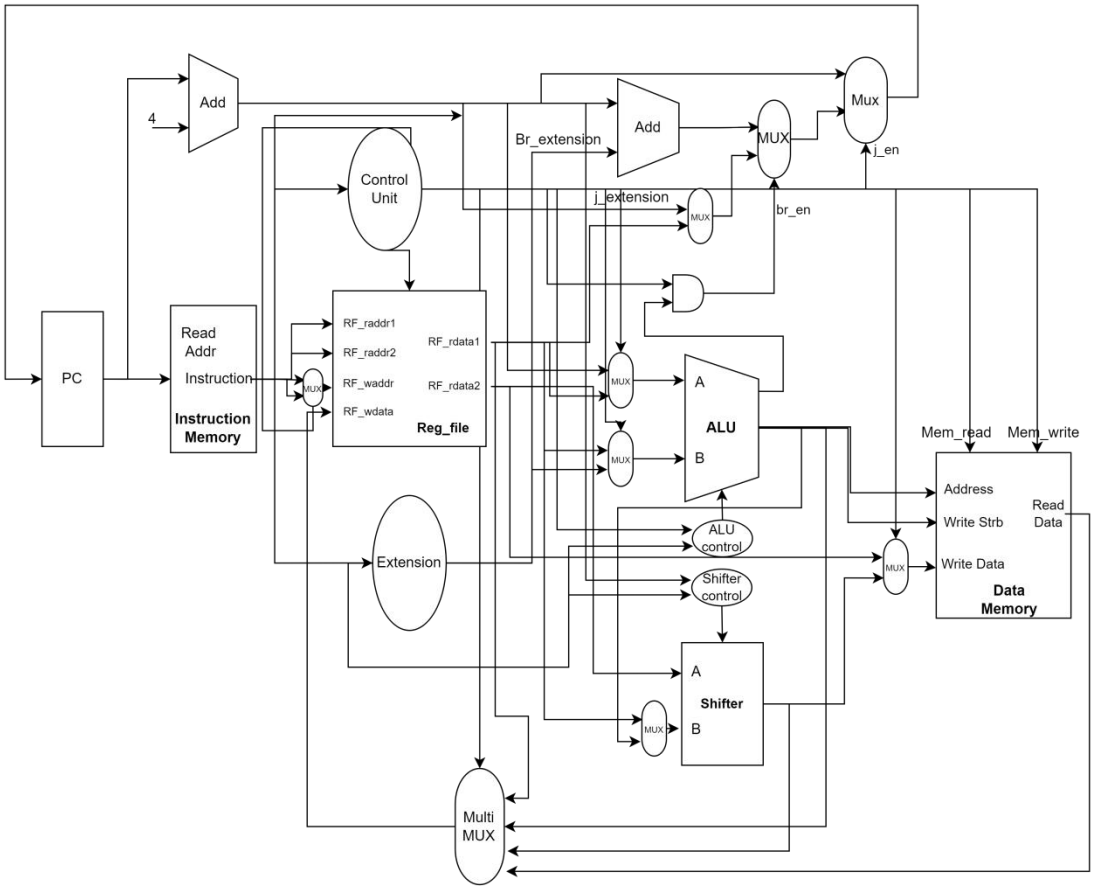
2. shifter.v 代码说明

```
//Shiftop
`define    SLL  2'b00
`define    SRL  2'b10 //set the word right logical
`define    SRA  2'b11 //set the word right arithmetic

//wire variable for direct results
    //SRA: care space left by SRA,so do not use simple shift
    wire [`DATA_WIDTH - 1:0] R_SLL;
    wire [`DATA_WIDTH - 1:0] R_SRL;
    wire [`DATA_WIDTH - 1:0] R_SRA;
    wire [63:0] SRA_tmp;

    assign R_SLL = A << B[4:0];
    assign R_SRL = A >> B[4:0];
    assign SRA_tmp = {{32{A[31]}},A} >> B[4:0];
    assign R_SRA = SRA_tmp[31:0];

    assign Result = ({32{Shiftop == `SLL}} & R_SLL) |
                    ({32{Shiftop == `SRL}} & R_SRL) |
                    ({32{Shiftop == `SRA}} & R_SRA) ;
```

逻辑移位按照语义进行操作即可，对于算数右移，由于高位补充的结果依赖于其符号位，因此先将 A 符号拓展为 64 位，再截取移位后的低 32 位作为算数右移结果即可。

3. 单周期 simple_cpu.v 代码说明

逻辑结构图：其中只含有 PC+4 和 PC+br_extension 两个加法器和 ALU，只含有 Shifter 一个移位器。



省略变量端口声明部分，其余根据逻辑相关程度，可分为 ID 译码部分，ALU 部分，Shifter 部分，Reg_file 部分，PC 部分和 MEM 访存部分 6 个部分，以下将分部分给出代码逻辑说明，并就关键部分给出仿真波形说明。

（1）ID 译码部分

```
//Analyse Instruction code
    assign opcode = Instruction[31:26];
    assign rs     = Instruction[25:21];
    assign rt     = Instruction[20:16];
    assign rd     = Instruction[15:11];
    assign sa     = Instruction[10:6];
    assign func   = Instruction[5:0];
    assign imm    = Instruction[15:0];
    assign index  = Instruction[25:0];

//extension prepared for operation
    assign zero_extension    = { 16'b0 , imm };
    assign sign_extension    = { {16{imm[15]}} ,imm };
    assign br_extension  = { {14{imm[15]}} ,imm , 2'b0 };  //use in Branch
    assign j_extension   = { PC4[31:28] , index , 2'b0 };  //use in Jump
```

通过将指令集进行归纳，可得到 Instruction 的译码各信号对应位置，进行连接即可。同时根据不同指令的操作需求，可事先准备好 4 类拓展，即 imm 的零拓展和符号拓展，以及关于分支和跳转指令的拓展。

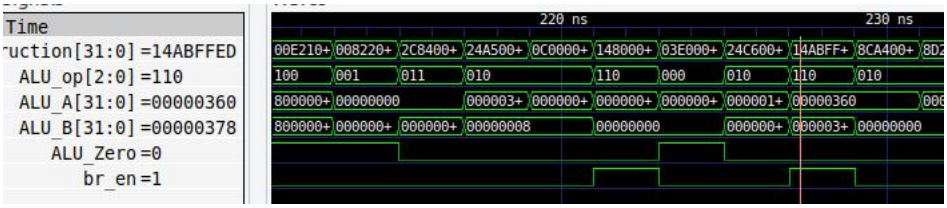
（2）ALU 部分

```
//signals connected to ALU
    //All type below can differ from others
    //Operation related to ALU list as : Type (op_A op_B) num :   feature
        //R calc (rs rt)  8 :           opcode[5:0]=000000 & func[5]=1
        //I calc (rs imm) 6 :           opcode[5:3]=001 & (~&opcode[2:0]) diff from lui
        //REGIMM (rs 0)   2 :           opcode[5:0]=000001 BGEZ BLTZ   GPR[rs]?0   using SLT(sign)
        //Branch (rs rt)  2 :           opcode[5:1]=00010
              //(rs 0)    2 :           opcode[5:1]=00011  using SUB and test ZERO or signflag
        //load and store (rs imm) 12 :  opcode[5]=1 using ADD
            //base(rs)+offset(imm)
        //JAL (PC_reg 8) 1:             opcode[5:0]=000011
        //JALR (PC_reg 8) 1:            opcode[5:0]=000000 & func[5:1]=001001

    //Operation not related to ALU list:
        //shifter 6 :                   opcode[5:0]=000000 & func[5:3]=000
        //jump    2 : J                 opcode[5:0]=000010
                //JR                    opcode[5:0]=000000 & func[5:0]=001000
        //lui   1 :                     opcode[5:0]=001111
        //move (rt 0)  2 :              opcode[5:0]=000000 & func[5:1]=00101
            //move rs to rd,
            //we can use ALU(ADD/SUB) or no, and we should care the condition about Reg_wen
```

```
assign ALU_op = (~|opcode[5:0]) & func[5]==1'b1  ? (func[3:2]==2'b00 ? {func[1],2'b10}      :
                                                    func[3:2]==2'b01 ? {func[1],1'b0,func[0]} :
                                                    func[3:2]==2'b10 ? {~func[0],2'b11}           :
                                                    0 )       :
            opcode[5:3]==3'b001 & (~&opcode[2:0]) ? (opcode[2:1]==2'b00 ? {opcode[1],2'b10}   :
                                                   opcode[2]==1'b1 ? {opcode[1],1'b0,opcode[0]}   :
                                                   opcode[2:1]==2'b01 ? {~opcode[0],2'b11}   :
                                                   0 )       :
            opcode[5:0]==6'b000001                ? 3'b111 :
            opcode[5:2]==4'b0001                  ? 3'b110 :
            opcode[5]  ==1'b1                     ? 3'b010 :
            opcode[5:0]==6'b000011 | ((~|opcode[5:0]) & func[5:0]== 6'b001001)    ? 3'b010 :
            0; //default
assign ALU_A =  opcode[5:0]==6'b000011 | ((~|opcode[5:0]) & func[5:0]==6'b001001) ? PC :
            RF_rdata1;
//ALU_B: 4 / 8/ rt / 0 / sign_extend(imm) /zero_extend(imm)
//imm_extension: I_calc 6 + load and store 12
     //zero_extension : ANDI ORI XORI      opcode[2]=1
     //sign_extension : ADDIU SLTI SLTIU   opcode[2]=0    load and store: opcode[5]==1
assign ALU_B = opcode[5:3]==3'b001 & (~&opcode[2:0]) ? ( opcode[2] ? zero_extension : sign_extension ):
            opcode[5]                                 ? sign_extension                        :
            opcode[5:0]==6'b000001                    ? 32'b0                                 :
            opcode[5:0]==6'b000011 | ((~|opcode[5:0]) & func[5:0]== 6'b001001) ?   32'd8     :
            RF_rdata2;


alu alu_inst(
   .A(ALU_A),
   .B(ALU_B),
   .ALUop(ALU_op),
   .Overflow(ALU_Overflow),
   .CarryOut(ALU_CarryOut),
   .Zero(ALU_Zero),
   .Result(ALU_Result)
);
```

ALU 部分需要根据指令译码得到的结果确定操作类型以及操作数。不同类型指令的条件特征、对应操作数如代码注释所示，且不同类别的特征互斥，并可唯一确定该类指令。每一类中再根据 funct 和 opcode 对应字段得到具体操作类型（优化见第三部分）。特别的，REGIMM 使用 SLT 操作，而其余 branch 类指令使用 SUB 操作。

相应波形说明：



光标所示位置为 BNE 指令，ALU 使用 SUB 操作，所得结果 ALU_Zero 不为 0，表示两寄存器数不等，因此进行跳转。

（3）Shifter 部分

```
//signals connected to Shifter
    //Operation related to Shifter list: 6 + 2(swl swr)
    //B width is 5
    //R shift: opcode[5:0]=000000 & funct[5:3]=000
        //xxx  (rt sa) 3:       fun[2]=0
        //xxxv (rt rs) 3:       fun[2]=1
    //Swl/swr: opcode[5:0]=101x10
        //swl/swr (rt swl_shifter)
    assign Shifter_op = (~|opcode[5:0])&(~|func[5:3])   ? func[1:0]        :
                {opcode[5:3],opcode[1:0]} == 5'b10110  ? {~opcode[2],1'b0}     :
                0;
    assign Shifter_A = RF_rdata2;
    assign Shifter_B = (~|opcode[5:0])&(~|func[5:3])    ? (func[2] ? RF_rdata1[4:0] : sa):
        {opcode[5:3],opcode[1:0]} == 5'b10110 ? (opcode[2] ? swr_shifter : swl_shifter) :
         0;


    shifter shifter_inst(
       .A(Shifter_A),
       .B(Shifter_B),
       .Shiftop(Shifter_op),
       .Result(Shifter_Result)
    );
```

Shifter 部分主要支持两种类型，分别为 I-type 中的 Shifter 类型指令和 Store 类型中的 swl 和 swr，相应特征和操作数如代码注释所示。其中 swl 和 swx 的移动位数根据 ALU_Result 确定。

（4）Reg_file 部分

```
//signals connected to Reg_file
```

```
    assign RF_raddr1 = rs;
    assign RF_raddr2 = rt;
    //Operations related to RF_wen 32
        //Type        Num  addr     data          feature
        //R calc      8    rd       alu_re        opcode[5:0]=000000 & func[5]=1
        //I calc      6    rt       alu_re        opcode[5:3]=001 & (~&opcode[2:0])
        //load        7    rt       loaddata      opcode[5:3]=100
        //shift       6    rd       shift_re      opcode[5:0]=000000 & func[5:3]=000
        //JAL         1    31       PC8(alu_re)   opcode[5:0]=000011
        //JALR        1    rd       PC8(alu_re)   opcode[5:0]=000000 & func[5:0]=001001
        //LUI         1    rt       imm||0^16     opcode[5:0]=001111
        //MOV         2    rd       rdata1        opcode[5:0]=000000 & func[5:1]=00101
            //mov is only use condition to refresh GPR, br or jump refresh PC, so we should
consider whether is 0
    //Operation not related to RF_wen 13
        //Type     Num     feature
        //REGIMM     2     opcode[5:0]=000001
        //Branch     4     opcode[5:2]=0001
        //store      5     opcode[5:3]=101
        //J          1     opcode[5:0]=000010
        //JR         1     opcode[5:0]=000000 & func[5:0]=001000
    assign RF_wen = (opcode[5:0]==6'b000000 & func[5]==1'b1)      |
                    (opcode[5:3]==3'b001 & (~&opcode[2:0]))       |
                    (opcode[5:3]==3'b100)                         |
                    (opcode[5:0]==6'b000000 & func[5:3]==3'b000)  |
                    (opcode[5:0]==6'b000011)                      |
                    (opcode[5:0]==6'b000000 & func[5:0]==6'b001001)   |
                    (opcode[5:0]==6'b001111)                      |
                     ((opcode[5:0]==6'b000000 & func[5:1]==5'b00101) & ((func[0] &
                          (|RF_rdata2))|(~func[0] & (~|RF_rdata2)))) ;
    assign RF_waddr = (opcode[5:3]==3'b100 | opcode[5:3]==3'b001 )    ? rt     :
                      (opcode[5:0]==6'b000011)                        ? 5'd31 :
                       rd ;
   assign RF_wdata = ((opcode[5:0]==6'b000000 & func[5:1]==5'b00101) & ((func[0] &
   (|RF_rdata2))|(~func[0] & (~|RF_rdata2))))   ? RF_rdata1       :
                (opcode[5:3]==3'b100)       ? load_data       : //define below
                (opcode[5:0]==6'b000000 & func[5:3]==3'b000)  ? Shifter_Result  :
                (opcode[5:0]==6'b001111)     ? {imm,16'b0}         :
                ALU_Result ;

    reg_file reg_file_inst(
        .clk(clk),
        .raddr1(RF_raddr1),
        .raddr2(RF_raddr2),
```

```
        .rdata1(RF_rdata1),
        .rdata2(RF_rdata2),
        .waddr(RF_waddr),
        .wdata(RF_wdata),
        .wen(RF_wen)
    );
```

Reg_file 部分除了完成内存读之外，内存写部分主要包含对运算 calc、内存读 load、寄存器数据移动 MOV 和跳转 Link 相关指令。指令类别对应特征及写数据如注释所示。

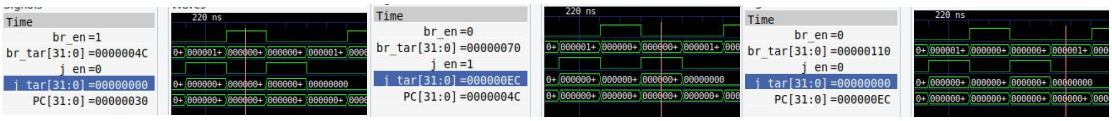
（5）PC 部分

```
//signals connected to PC
        //branch condition  test zero signflag and overflow:
        //branch target is same
            //BEQ,BNE,BLEZ,BGTZ (using SUB): opcode[5:2]=0001   opcode[1:0]=00,01,10,11
            //BLTZ,BGEZ   (using SLT):       opcode[5:0]=000001      rt[0]=0 1
        //jump has no condition
        //jump target differ:
            //J JAL   :   j_extension        opcode[5:1] =00001
            //JR JALR :   rs             opcode[5:0] = 000000 & func[5:1]=00100

    assign br_en = ( (opcode[5:2]==4'b0001)  & (   ( (opcode[1:0]==2'b00) & ALU_Zero)  |
                    ( (opcode[1:0]==2'b01) & ~ALU_Zero)                          |
                    ( (opcode[1:0]==2'b10) & (ALU_Zero  | (ALU_Overflow ^ ALU_Result[31] )))  |
                    ( (opcode[1:0]==2'b11) & (~ALU_Zero & (ALU_Overflow ~^ ALU_Result[31])))  )
                   ) |
                  ( (opcode[5:0]==6'b000001)& ((~rt[0] & ALU_Result[0]) |
                                               (rt[0]  & ~ALU_Result[0]) )
                 ) ;
    assign br_tar = PC4 + br_extension ;
    assign j_en = (opcode[5:1]==5'b00001) | ((~|opcode[5:0])&(func[5:1]==5'b00100)) ;
    assign j_tar  = ({32{(opcode[5:1]==5'b00001)}}           & j_extension ) |
              ({32{((~|opcode[5:0])&(func[5:1]==5'b00100))}} & RF_rdata1);

    always @ (posedge clk) begin
        if (rst) begin
            PC <= 32'b0;
        end
        else begin
            PC <= br_en ? br_tar : j_en ? j_tar : PC4 ;
        end
```

```
    end
```

PC 更新主要可分为三类：顺序更新、条件跳转和无条件跳转。其中条件跳转需要结合 ALU 运算结果，同时因为 mips 指令集分支延迟槽的存在，br_tar 其中一个加数为 PC+4。

相应波形说明：



以上分别为发生条件跳转和无条件跳转时 PC 的更新情况，当对应使能信号拉高时，PC 更新为对应目标。

（6）MEM 访存部分

```
//signals connected to memory (load and store)
    //load opcode[5:3] = 3'b100    store opcode[5:3]=101 using Little_endian
    //mem control
    assign MemRead  = opcode[5] & ~opcode[3];    //100
    assign MemWrite = opcode[5] & opcode[3];    //101
    assign Address  = {ALU_Result[31:2] , 2'b0};

    //load data  mem -> rt
        //byte means vaddr/ALU_result[2:0] memword means Read_data
    assign lb_data =
({ 32{~ALU_Result[1] & ~ALU_Result[0]} } & { {24{Read_data[ 7]}} , Read_data[ 7: 0] } )|
({ 32{~ALU_Result[1] &  ALU_Result[0]} } & { {24{Read_data[15]}} , Read_data[15: 8] } )|
({ 32{ ALU_Result[1] & ~ALU_Result[0]} } & { {24{Read_data[23]}} , Read_data[23:16] } )|
({ 32{ ALU_Result[1] &  ALU_Result[0]} } & { {24{Read_data[31]}} , Read_data[31:24] } );
    assign lh_data = ({ 32{~ALU_Result[1]} } & { {16{Read_data[15]}} , Read_data[15: 0] })|
                ({ 32{ ALU_Result[1]} } & { {16{Read_data[31]}} , Read_data[31:16] } );
    assign lw_data = Read_data ;
    assign lbu_data = {24'b0 , lb_data[ 7:0]} ;
    assign lhu_data = {16'b0 , lh_data[15:0]} ;
    assign lwl_data =
({ 32{~ALU_Result[1] & ~ALU_Result[0]} } & { Read_data[ 7: 0] , RF_rdata2[23: 0] } )|
({ 32{~ALU_Result[1] &  ALU_Result[0]} } & { Read_data[15: 0] , RF_rdata2[15: 0] } )|
({ 32{ ALU_Result[1] & ~ALU_Result[0]} } & { Read_data[23: 0] , RF_rdata2[ 7: 0] } )|
({ 32{ ALU_Result[1] &  ALU_Result[0]} } &  Read_data[31: 0] )            ;
    assign lwr_data = ({ 32{~ALU_Result[1] & ~ALU_Result[0]} } &  Read_data[31: 0] )|
     ({ 32{~ALU_Result[1] &  ALU_Result[0]} } & { RF_rdata2[31:24] , Read_data[31: 8] })|
```

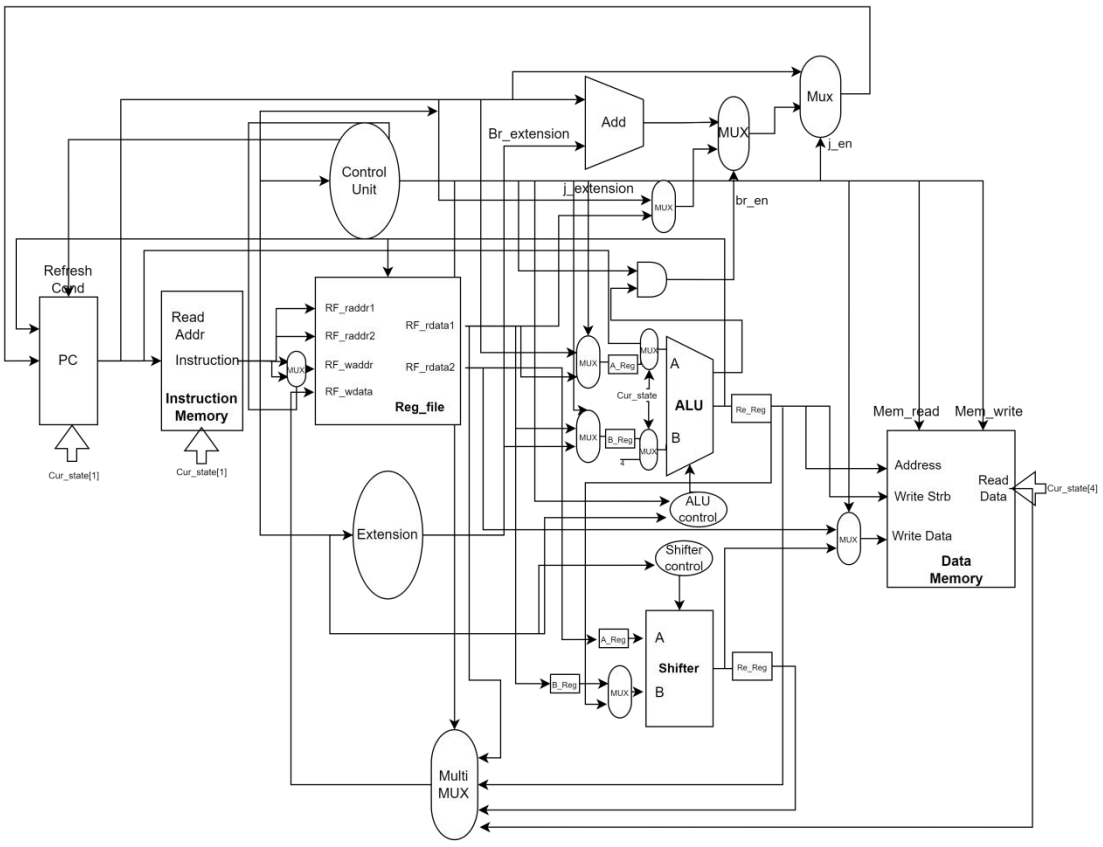
```
    ({ 32{ ALU_Result[1] & ~ALU_Result[0]} } & { RF_rdata2[31:16] , Read_data[31:16] })|
    ({ 32{ ALU_Result[1] &  ALU_Result[0]} } & { RF_rdata2[31: 8] , Read_data[31:24] });
    assign load_data = ( {32{opcode[2:0]==3'b000}} &  lb_data )   |
                       ( {32{opcode[2:0]==3'b001}} &  lh_data )   |
                       ( {32{opcode[2:0]==3'b011}} &  lw_data )   |
                       ( {32{opcode[2:0]==3'b100}} & lbu_data )   |
                       ( {32{opcode[2:0]==3'b101}} & lhu_data )   |
                       ( {32{opcode[2:0]==3'b010}} & lwl_data )   |
                       ( {32{opcode[2:0]==3'b110}} & lwr_data )   ;


    //store data rt->mem
        //strb is signal showing which bytewrite is valid, code by truthtable //byte means
vaddr/ALU_result[2:0]
        // vaddr     swl_strb     swl_shifter  swr_Strb     swr_shifter
        // 00/0      0001         11000/24     1111         00000/0
        // 01/1      0011         10000/16     1110         01000/8
        // 10/2      0111         01000/8      1100         10000/16
        // 11/3      1111         00000/0      1000         11000/24


     assign sb_strb  = { ALU_Result[1]& ALU_Result[0] , ALU_Result[1]& ~ALU_Result[0] ,
     ~ALU_Result[1]& ALU_Result[0] , ~ALU_Result[1]& ~ALU_Result[0]};
    assign sh_strb  = { {2{ALU_Result[1]}} , {2{~ALU_Result[1]}} } ;
    assign sw_strb  = 4'b1111 ;
    assign swl_strb = { &ALU_Result[1:0] , ALU_Result[1] , |ALU_Result[1:0] , 1'b1};
    assign swr_strb = { 1'b1, ~&ALU_Result[1:0] , ~ALU_Result[1], ~|ALU_Result[1:0]} ;
    assign Write_strb = ( {4{opcode[2:0]==3'b000}} & sb_strb )    |
                        ( {4{opcode[2:0]==3'b001}} & sh_strb )    |
                        ( {4{opcode[2:0]==3'b011}} & sw_strb )    |
                        ( {4{opcode[2:0]==3'b010}} & swl_strb )   |
                        ( {4{opcode[2:0]==3'b110}} & swr_strb )   ;


    assign sb_data      = { 4{RF_rdata2[7:0]} } ;
    assign sh_data     = { 2{RF_rdata2[15:0]}} ;
    assign sw_data      = RF_rdata2 ;
    assign swl_shifter = {~ALU_Result[1:0] , 3'b0};
    assign swr_shifter = { ALU_Result[1:0] , 3'b0};
    // Reuse shifter module to save two shifter
    // swl_data = RF_rdata2 >> swl_shifter ;
    // swr_data = RF_rdata2 << swr_shifter ;
    assign Write_data =   ( {32{opcode[2:0]==3'b000}} & sb_data )      |
                          ( {32{opcode[2:0]==3'b001}} & sh_data )      |
                          ( {32{opcode[2:0]==3'b011}} & sw_data )      |
                          ( {32{opcode[1:0]==2'b10}}  & Shifter_Result )   ;
```

Load 和 Store 两类指令均包含不同指令，需对于不同指令确定数据。对 load

类指令，ALU_Result 的后两位确定了新数据覆盖原数据的位置；对 store 类指令，ALU_Result 的后两位确定了掩码，用以向内存传递新数据需要覆盖的位置。同时对于 swl 和 swr 指令，需要对 RF_rdata2 进行移位操作，可复用 Shifter 以节约资源，且将需移动的位数用二进制表示可以逐位用 ALU_Result 表示，减少资源消耗。Load 和 Store 最后根据 opcode 对相应指令得到数据进行选择即可。

4. 多周期 simple_cpu

逻辑结构图：只含有 PC+br_extension 一个加法器和 ALU，只含有 Shifter 一个移位器。相比单周期处理器减少了 PC+4 加法器，在不同阶段间增加了寄存器。



省略变量端口声明和与单周期重复部分，只说明新增及修改代码，大体可分

为状态机部分，寄存器部分，ALU 部分和 Shifter 部分。

(1) 状态机部分

```
//define the state of machine by one-hot
    localparam   RST =9'b000000001,
                 IF  =9'b000000010,
                 ID  =9'b000000100,
                 EX  =9'b000001000,
                 MEM =9'b000010000,
                 WB  =9'b000100000;
//State Machine
    //Part I:
    always @ (posedge clk) begin
       if(rst) begin
           current_state <= IF;
       end
       else begin
           current_state <= next_state;
       end
    end

    //Part II: Combinatorial logic
    always @ (*)
    begin
        case(current_state)
            IF: begin
                    next_state = ID ;
            end
            //NOP: Instr = 32'b0
            ID: begin
                if(~|Instruction_Reg)
                    next_state = IF ;
                else
                    next_state = EX ;
            end
            //EX->IF:
                //REGIMM    2: opcode[5:0]=000001
                //I_branch  4: opcode[5:2]=0001
                //J     1: opcode[5:0]=000010
            //EX->WB
                //R_Type   18: opcode[5:0]=6'b0 include JALR,JR //JR: rs->GPR[0]
                //I_calc    7: opcode[5:3]=001 include LUI
                //JAL       1: opcode[5:0]=000011
```

```
            //EX->MEM
                //load and store 12: opcode[5]=1
            EX: begin
                if((opcode[5:0]==6'b000001) | (opcode[5:2]==4'b0001) |
(opcode[5:0]==6'b000010) )
                    next_state = IF;
                else if((~|opcode)| (opcode[5:3]==3'b001) |
(opcode[5:0]==6'b000011))
                    next_state = WB;
                else if(opcode[5]==1'b1)
                    next_state = MEM;
            end
            //MEM->IF: store: opcode[5:3]=101
            //MEM->WB: load : opcode[5:3]=100
            MEM: begin
                if(opcode[5:3]==3'b101)
                    next_state = IF;
                else if(opcode[5:3]==3'b100)
                    next_state = WB;
            end
            WB: begin
                next_state = IF;
            end
            default: begin
                next_state = IF;
            end
        endcase
    End
```

状态机的第一部分使用时序逻辑进行现态的更新，第二部分使用组合逻辑根据现态和指令确定次态情况。转移条件如代码注释所示。特别的，ID 阶段需要考虑空指令的跳转。

(2) 寄存器部分

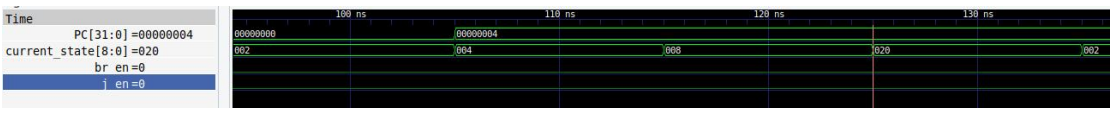
说明：ALU 和 Shift 输入输出相关寄存器见其对应部分

```
    //Part III: deal with output
    //one always module to deal one reg
    always @ (posedge clk) begin
        if(current_state[1]) //IF
            Instruction_Reg <= Instruction ;
    end
```

```
always @ (posedge clk) begin
    if(current_state[4]) //MEM
        Read_data_Reg <=Read_data ;
end

always @ (posedge clk) begin
    if(current_state[1]) //IF
        PC_Reg <= PC;
end

always @ (posedge clk) begin
    if (rst) begin
        PC <= 32'b0;
    end
    else if(current_state[1] & ~rst) begin
        PC <= ALU_Result;
        //all OP refresh in ID, judge by IF
    end
    else if(current_state[3] & ~rst &(|Instruction_Reg))begin
        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 PC
    end
end
```

由于 PC、Instruction 和 Read_data 均只维持一拍，多周期 CPU 新增了相应寄存器将这些值进行寄存。而多周期一条指令过程中 PC 将更新两次，第一次在 IF 阶段对所有指令 PC 均更新加 4，第二次在 EX 阶段根据 ALU 运算结果和指令类别确定是否条件跳转或无条件跳转，否则维持 PC 不变。

相应波形说明：



当不产生跳转时，PC 只在 IF 阶段(002)后更新一次，PC+4;

当产生跳转时，PC 在 IF 阶段(002)后更新一次，PC+4;在 EX 阶段(008)后

再更新一次，跳转到目标值。

(3) ALU 部分

```
//signals connected to ALU
     //IF stage: All instruction: PC+4
     //Other stage: To save adder, reuse ALU when PC+8 (swl swr)
     //All type below can differ from others
     //Operation related to ALU list as : Type (op_A op_B) num :   feature
          //R calc (rs rt)  8 :              opcode[5:0]=000000 & func[5]=1
          //I calc (rs imm) 6 :              opcode[5:3]=001 & (~&opcode[2:0]) diff from lui
          //REGIMM (rs 0)   2 :              opcode[5:0]=000001 BGEZ BLTZ   GPR[rs]?0   using SLT(sign)
          //Branch (rs rt)  2 :              opcode[5:1]=00010
                 //(rs 0)   2 :              opcode[5:1]=00011  using SUB and test ZERO or signflag
          //load and store (rs imm) 12 :     opcode[5]=1 using ADD
               //base(rs)+offset(imm)
          //JAL (PC_reg 8) 1:           opcode[5:0]=000011
          //JALR (PC_reg 8) 1:          opcode[5:0]=000000 & func[5:1]=001001

     //Operation not related to ALU list:
          //shifter 6 :                 opcode[5:0]=000000 & func[5:3]=000
          //jump    2 : J               opcode[5:0]=000010
                    //JR            opcode[5:0]=000000 & func[5:0]=001000
          //lui   1 :               opcode[5:0]=001111
          //move      (rt 0)  2 :           opcode[5:0]=000000 & func[5:1]=00101
               //move rs to rd,
               //we can use ALU(ADD/SUB) or no, and we should care the condition about Reg_wen
     assign ALU_op_origin = (~|opcode[5:0]) & func[5]==1'b1 ? (func[3:2]==2'b00 ? {func[1],2'b10}:
                                                               func[3:2]==2'b01 ? {func[1],1'b0,func[0]}  :
                                                               func[3:2]==2'b10 ? {~func[0],2'b11}        :
                                                                0 )   :
     opcode[5:3]==3'b001 & (~&opcode[2:0])  ? (opcode[2:1]==2'b00 ? {opcode[1],2'b10} :
                                               opcode[2]==1'b1 ? {opcode[1],1'b0,opcode[0]}      :
                                               opcode[2:1]==2'b01 ? {~opcode[0],2'b11}      :
                                               0 )      :
     opcode[5:0]==6'b000001            ? 3'b111 :
     opcode[5:2]==4'b0001              ? 3'b110 :
     opcode[5]  ==1'b1                 ? 3'b010 :
     opcode[5:0]==6'b000011 | ((~|opcode[5:0]) & func[5:0]== 6'b001001)     ? 3'b010 :
     0; //default

     always @(posedge clk) begin
             if(rst)
                     ALU_op_reg <=0;
```

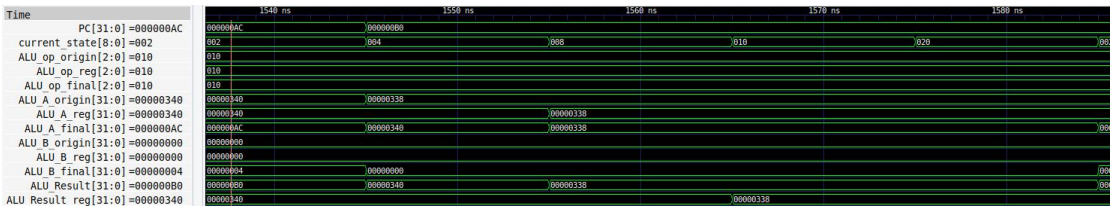
```
            else if(current_state[2]) //ID
                    ALU_op_reg <= ALU_op_origin;
        else
            ALU_op_reg <= ALU_op_reg;
    end

assign ALU_op_final = current_state[1] ? 3'b010 : ALU_op_reg;

    assign ALU_A_origin =  opcode[5:0]==6'b000011 | ((~|opcode[5:0]) & func[5:0]==6'b001001) ? PC_Reg :
                           RF_rdata1;

    always @(posedge clk) begin
            if(rst)
                    ALU_A_reg <= 0;
            else if(current_state[2])//ID
                    ALU_A_reg <= ALU_A_origin;
        else
            ALU_A_reg <= ALU_A_reg;
    end

assign ALU_A_final =  current_state[1]    ? PC : ALU_A_reg;

//ALU_B: 4 / 8/ rt / 0 / sign_extend(imm) /zero_extend(imm)
//imm_extension: I_calc 6 + load and store 12
     //zero_extension : ANDI ORI XORI     opcode[2]=1
     //sign_extension : ADDIU SLTI SLTIU  opcode[2]=0    load and store: opcode[5]==1
 assign ALU_B_origin =   opcode[5:3]==3'b001 & (~&opcode[2:0]) ? ( opcode[2] ? zero_extension :
 sign_extension )    :
       opcode[5]                    ? sign_extension                          :
       opcode[5:0]==6'b000001   ? 32'b0                                   :
       opcode[5:0]==6'b000011 | ((~|opcode[5:0]) & func[5:0]== 6'b001001) ?   32'd8  :
    RF_rdata2;
    always @(posedge clk) begin
            if(rst)
                    ALU_B_reg <= 0;
            else if(current_state[2])
                    ALU_B_reg <= ALU_B_origin;
        else
            ALU_B_reg <= ALU_B_reg;
    end

    assign ALU_B_final =  current_state[1] ? 32'd4    : ALU_B_reg;

    always @(posedge clk) begin
```

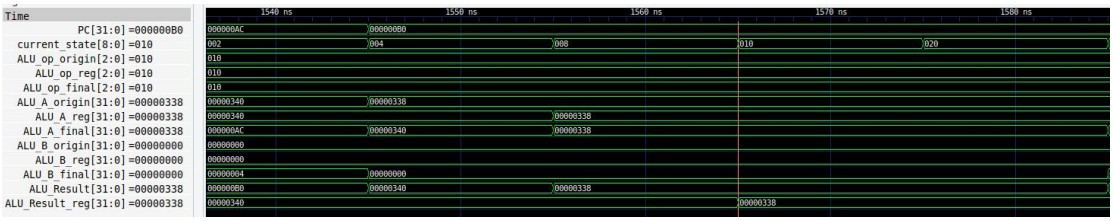
```
            if(rst)
                    ALU_Result_reg <= 0;
            else if(current_state[3])//EX
                    ALU_Result_reg <= ALU_Result;
        else
            ALU_Result_reg <= ALU_Result_reg;
       end


    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)
    );
```

对于多周期下的 ALU，单周期相应的操作只在 EX 阶段进行，其余阶段闲置，可在这些阶段被调用以减少加法器数量、节约电路资源。

对于单周期相应操作，设置 ALU_X_origin 信号，在 EX 的前一阶段 ID 将其寄存，并将结果在 EX 阶段进行寄存。同时考虑到 ALU 可在 IF 阶段复用以得到 PC+4，因此设计数据旁路，通过选择数据旁路值和寄存器得到最终的 ALU 操作数和控制信号。寄存器和数据旁路的设计可以使得 ALU 兼具时效性、稳定性和复用程度高的特点。

相应波形说明：





该条指令在 IF 阶段(002)使用了数据旁路，复用 ALU 获得了 PC+4 的值，在 ID 阶段(004)得到了 ALU 操作数，在 EX 阶段(008)操作数寄存值更新得到运算结果，并在后一阶段得到结果的寄存值。

(4) Shifter 部分

```
//signals connected to Shifter
    //Operation related to Shifter list: 6 + 2(swl swr)
    //B width is 5
    //R shift: opcode[5:0]=000000 & funct[5:3]=000
        //xxx  (rt sa) 3:        fun[2]=0
        //xxxv (rt rs) 3:        fun[2]=1
    //Swl/swr: opcode[5:0]=101x10
        //swl/swr (rt swl_shifter)
    assign Shifter_op = (~|opcode[5:0])&(~|func[5:3])      ? func[1:0]      :
                {opcode[5:3],opcode[1:0]} == 5'b10110 ? {~opcode[2],1'b0}  :
                0;
        always @(posedge clk) begin
                if(current_state[2])
                        Shifter_op_reg <= Shifter_op;
        end

    assign Shifter_A = RF_rdata2;
        always @(posedge clk) begin
                if(current_state[2])
                        Shifter_A_reg <= Shifter_A;
        end

        assign Shifter_B = (~|opcode[5:0])&(~|func[5:3]) ? (func[2] ? RF_rdata1[4:0] :
                                                            sa) : 0 ;
        always @(posedge clk) begin
                if(current_state[2])
                        Shifter_B_reg <= Shifter_B;
        end
        //reuse shifter by swx in MEM
    assign Shifter_B_final =
                current_state[4] & {opcode[5:3],opcode[1:0]} == 5'b10110 ?
                (opcode[2] ? swr_shifter : swl_shifter) :
                 Shifter_B_reg;

        always @(posedge clk) begin
                if(rst)
```

```
                    Shifter_Result_reg <= 0;
            else if(current_state[3])
                    Shifter_Result_reg <= Shifter_Result;
    end

shifter shifter_inst(
    .A(Shifter_A_reg),
    .B(Shifter_B_final),
    .Shiftop(Shifter_op_reg),
    .Result(Shifter_Result)
);
```

与 ALU 相似，Shifter 也可在闲置阶段被复用以节约资源。此处同样设置 origin 处理单周期中的运算指令，并设置数据旁路处理 swl 和 swr 指令所需的数据移位操作。

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）

问题 1：Verilog 位运算语法问题

{}外加数字进行位拓展应当在外侧再加一对{}，查看云平台语法检查 log 发现问题更正。

问题 2：电路资源精简

ALU 和 Shifter 可以在闲置时被复用以节约资源，如 PC+8 的加法操作和 swl、swr 的移位操作都可以进行复用，同时在多周期 CPU 中，由于 IF 阶段 ALU 闲置，可以被复用以进行 PC+4 操作。

问题 3：golden 比对问题

在提前进行实验过程中发现 golden 状态更新逻辑与多周期状态机不符，进行反馈后更正。

问题 4：多周期 CPU 的寄存器使用和协同问题

验收阶段和助教及老师的交流过程中，我理解了 ALU、Shifter 的输入输出寄存器可以将不同状态阶段进行分隔，从而减少最长组合逻辑链长度，允许处理器运行更高的频率。

由于 PC+4 对 ALU 的复用和 swl、swr 对 Shifter 的复用均只在当拍内，无法使用寄存器处理，因此可以设计数据旁路，并在输入 ALU 时选择数据旁路或寄存器，完成寄存器和数据旁路的协同，从而同时具备节约资源和降低延迟的优势。对于 ALU 和 Shifter 的输出，在本状态阶段内使用输出立即值，在不同状态阶段的使用寄存值，使得数据可以具有时效性和稳定性。

问题 5：多周期 CPU 添加寄存器时错误

Debug 过程：ALU_Result 错误 <- ALU_A_origin 错误 <- RF_rdata1 未定义 <- 对应 addr 的 RF_wdata 未定义 <- 写寄存器不同情形 <- Shifter_Result 错误 <- Shifter_B_final default 态错误。通过错误逻辑链向前追溯到错误发生处进行更正。

三、 对讲义中思考题（如有）的理解和回答

ALUop 编码规律和优化

编码规律：

R-type 由最低 4 位映射出 ALUop，I-type 由最低 3 位映射出 ALUop，二者均可以通过两位 func 或 opcode 划分为加减运算、逻辑运算和比较运算三类。R-type 和 I-type 对应指令的最低三位一致（除 lui 外），因此在对应分类中可以使用相同的编码规则。

优化：

由先分类后编码的粗粒度编码转化为逐位分类编码的细粒度编码，减少了选择层数，代码如下所示：

```
R-type:
    ALUop[2] = (~func[3]&func[1])|(func[3]&~func[0])
    ALUop[1] = ~func[2]
    ALUop[0] = (func[2]&func[0])|(~func[2]&func[3])
I-type:
    ALUop[2] = (~opcode[2]&opcode[1]&~opcode[0])|
                ((opcode[2]|~opcode[1])&opcode[1])
    ALUop[1] = ~opcode[2]
    ALUop[0] = (~opcode[2]&opcode[1])|(opcode[2]&opcode[0])
```

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）

心得感受：

本次实验单周期处理器部分考察的主要仍是对指令集的理解，对 Verilog 代

码并未具有太高要求，代码编写过程中主要考察的是对指令的理解归纳以及将逻辑相关代码分模块编写的能力。

多周期处理器部分对时序逻辑的理解和 CPU 处理指令的流程提出了更高要求。一方面需要对只能维持一拍的数据进行寄存，且需关注数据在哪个状态阶段应有效，另一方面需要根据指令确定状态转移规则。同时在验收过程中通过和助教、老师的交流我理解了寄存器在不同阶段中起到的作用（详见第二部分），以及寄存器和数据旁路的协同（如 PC+4 临时借用 ALU），这对后续 prj3 和 pipeline 数据传递的处理具有很大帮助。

建议：

在提前进行实验的过程中我发现多周期 golden 文件的 PC 更新逻辑问题，后续老师进行了修正更新。但由于 PC 更新逻辑可以具有多种写法，而仿真阶段的 golden 写法是确定的，这就可能产生上板可以正常运行但是仿真出错的现象。为支持多种写法，可通过编写多个不同更新逻辑的 golden 文件，在 bhv_sim 阶段使用多个同名任务，或发起多个 pipeline 解决，有一个通过即为通过。

致谢：

感谢陈欲晓助教和张老师在验收过程中对寄存器作用的深入讲解，感谢芦溶民助教和常老师在 golden 更新和云平台相关问题的解答和帮助。