计算机体系结构 实验二

PB8151866 龚小航

【实验目的】

根据 RISC-V32 数据通路图，在已有的代码框架下完成流水线 CPU 的设计

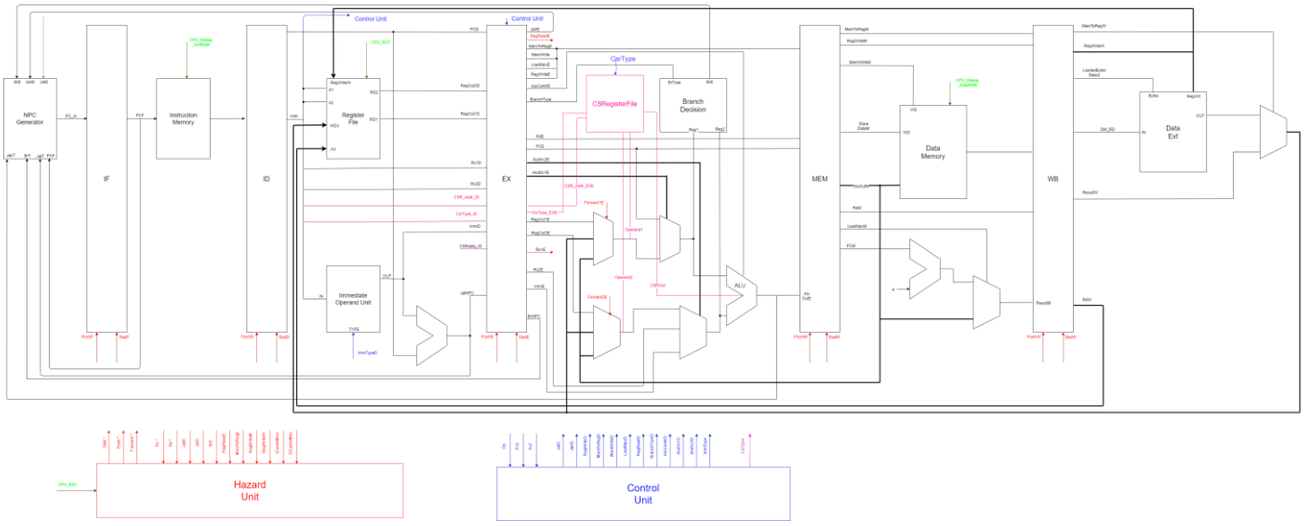
利用给出的仿真文件与数据，验证结果的完整性与正确性

【实验环境】

Xilinx Vivado 2019.1

【实验过程及具体实现】

最终的数据通路如下所示：



本次实验需要实现 RV32 中的各种基础指令。在给出的代码框架中，需要填补的模块主要是控制模块 CU，逻辑运算单元 ALU，立即数扩展单元 ImmOperandUnit，分支处理单元以及处理非字对齐的 load 指令所需的 DataExt 单元，另外还需要处理指令间的冲突或数据相关。

接下来按模块说明补充的部分：

1、NPC_Generator 模块：组合逻辑模块，根据传入的控制信号 BranchE, JalD, JalrE 决定输出的地址。

```
module NPC_Generator(
    input wire [31:0] PCF,JalrTarget, BranchTarget,
JalTarget,
    input wire BranchE,JalD,JalrE,
    output reg [31:0] PC_In
    );
    // 请补全此处代码
    initial PC_In = 32'h00000000;
    always@(*)begin
        if(BranchE) begin
            PC_In = BranchTarget;
        end
        else if(JalrE) begin
            PC_In = JalrTarget;
        end
        else if(JalD) begin
            PC_In = JalTarget;
        end
        else begin
            PC_In = PCF + 32'h00000004; //pc+4
        end
    end
endmodule
```

2、**控制模块代码如下，**引入了外部宏定义 Parameters.v 文件，使代码可读性更强。此模块在译码阶段产生一条指令在各个阶段所需要的所有控制信号，其中一些逻辑较简单的定义为 **wire** 输出类型，直接利用 **assign** 对其进行赋值；另一些控制信号与指令之间的依赖更为复杂，因此对每一条待实现的指令逐条赋值即可。代码示例如下，每一个阶段选取了一条指令的控制信号作为代表，其余指令类似。

```
`include "Parameters.v"
module ControlUnit(
    input wire [6:0] Op,
    input wire [2:0] Fn3,
    input wire [6:0] Fn7,
    output wire JalD,
    output wire JalrD,
    output reg [2:0] RegWriteD,
    output wire MemToRegD,
    output reg [3:0] MemWriteD,
    output wire LoadNpcD,
    output reg [1:0] RegReadD,
    output reg [2:0] BranchTypeD,
    output reg [3:0] AluContrlD,
    output wire [1:0] AluSrc2D,
    output wire AluSrc1D,
    output reg [2:0] ImmType,
    output reg [2:0] CSRType  //CSR类型
    );
    localparam LOAD_TYPE_OP = 7'b0000011, STORE_TYPE_OP = 7'b0100011,
               CSR_TYPE_OP = 7'b1110011;
    assign LoadNpcD = JalD | JalrD;
    assign JalD = ( Op == `JAL_OP )?  1'b1 : 1'b0;
    assign JalrD = ( Op == `JALR_OP )? 1'b1 : 1'b0;
    assign MemToRegD = (Op == LOAD_TYPE_OP || Op == STORE_TYPE_OP)?  1'b1 : 1'b0;
    assign AluSrc1D = (Op == `AUIPC_OP)? 1'b1:1'b0;
    assign AluSrc2D = ( (Op==7'b0010011)&&(Fn3[1:0]==2'b01) )?
            (2'b01):(((Op==7'b0110011)||(Op==7'b1100011))? 2'b00:2'b10);
   always@(*) begin
        if(Op==`SLLI_OP && Fn3==`SLLI_FN3) begin
            RegWriteD = `YES;  MemWriteD = `NOSTORE;  RegReadD = 2'bxx;
            BranchTypeD = `NOBRANCH;  AluContrlD = `SLL;
            ImmType = `ITYPE; CSRType = `NOTCSR_Type;
        end
            ......//其他指令类似，省略
        /*阶段二*/
        else if(Op==`JALR_OP && Fn3==`JALR_FN3) begin
            RegWriteD = `YES;  MemWriteD = `NOSTORE;  RegReadD = 2'bxx;
            BranchTypeD = `NOBRANCH;  AluContrlD = `ADD;
            ImmType = `ITYPE; CSRType = `NOTCSR_Type;
        end
            ......//其他指令类似，省略
        /*阶段三*/
        else if(Op==`CSRRW_OP && Fn3==`CSRRW_FN3) begin
            RegWriteD = `YES;  MemWriteD = `NOSTORE;  CSRType = `CSRRW_Type;
            BranchTypeD = `NOBRANCH;  AluContrlD = `CSR; ImmType = `NOIMM;
        end
            ......//其他指令类似，省略
        else begin   //空指令或 flush
            RegWriteD = `NOREGWRITE;  MemWriteD = `NOSTORE; CSRType = `NOTCSR_Type;
            BranchTypeD = `NOBRANCH;  AluContrlD = `ADD; ImmType = `NOIMM;
        end
    end
endmodule
```

3、**立即数扩展模块**：使用组合逻辑，根据立即数类型从输入的 32 位数据中提取出立即数并扩展。

RISC-V 指令系统的立即数形式如下所示：

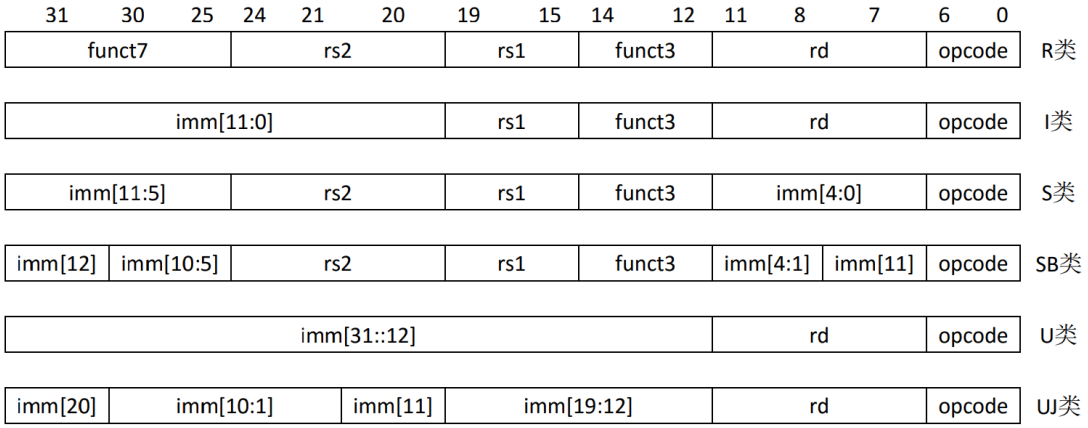


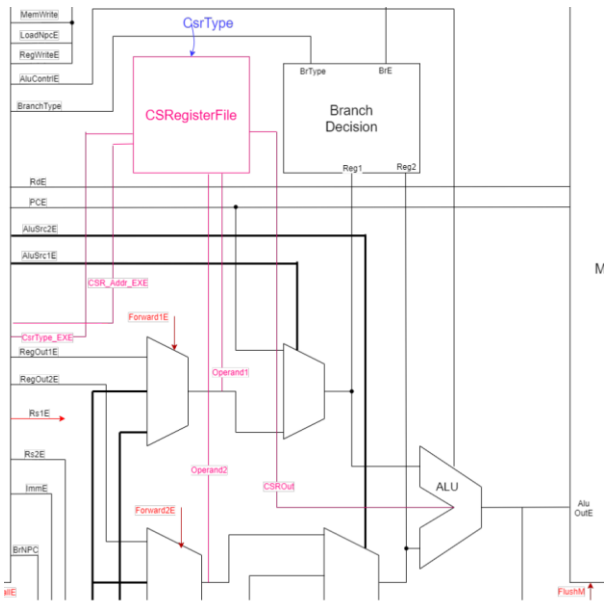
图 2.3：RISC-V 显示了立即数的基本指令格式

根据不同类型的立即数，在该模块内实现对应的提取立即数逻辑即可。代码如下：

```
`include "Parameters.v"
module ImmOperandUnit(
    input wire [31:7] In,
    input wire [2:0] Type,
    output reg [31:0] Out
    );
    always@(*)
    begin
        case(Type)
            `ITYPE: Out = { {21{In[31]}}, In[30:20] };  //符号扩展
            //......                                    //请完善代码
            `STYPE: Out = { {21{In[31]}}, In[30:25] ,In[11:7] };
            `SBTYPE: Out = { {20{In[31]}}, In[7], In[30:25], In[11:8],1'b0};
            `UTYPE: Out = { In[31:12], 12'b0};//低 12 位补 0
            `UJTYPE: Out = {{12{In[31]}}, In[19:12], In[20], In[30:21], 1'b0};
            `CSRITYPE: Out = { 27'b0, In[19:15] };
            default:Out = 32'hxxxxxxxx;
        endcase
    end
endmodule
```

4、**ALU 模块：**组合逻辑电路，根据 ALUControl 对输入的操作数进行处理。

```
`include "Parameters.v"
module ALU(
    input wire [31:0] Operand1,
    input wire [31:0] Operand2,
    input wire [3:0] AluContrl,
    input wire [31:0] CSRDATA,
    output reg [31:0] AluOut
    );
    // 请补全此处代码
    always@(*) begin
        case(AluContrl)
            `SLL: begin
                    AluOut = Operand1 << Operand2[4:0];
                end
            `SRL: begin
                    AluOut = Operand1 >> Operand2[4:0];
                end
            `SRA: begin
                    AluOut = ($signed(Operand1)) >>>  Operand2[4:0];
                end
            `ADD: begin
                    AluOut = Operand1 + Operand2;
                end
            `SUB: begin
                    AluOut = Operand1 - Operand2;
                end
            `XOR: begin
                    AluOut = Operand1 ^ Operand2;
                end
            `OR:  begin
                    AluOut = Operand1 | Operand2;
                end
            `AND: begin
                    AluOut = Operand1 & Operand2;
                end
            `SLT: begin
                    AluOut = ($signed(Operand1) < $signed(Operand2))?  32'h1 : 32'h0;
                end
            `SLTU: begin
                    AluOut = (Operand1 < Operand2)?  32'h1 : 32'h0;
                end
            `LUI:  begin
                    AluOut = Operand2;
                end
            `CSR: begin
                    AluOut = CSRDATA;
                end
           default: AluOut = 0;
       endcase
   end
endmodule
```



此处 CSR 的输出也经过 ALU，当 ALUControl 为 CSR 类型时，直接将 CSR 数据作为 ALUout 输出，共用后面的所有数据通路以及冲突处理，提高了器件复用率也简化了 CPU 设计。

5、分支信号产生模块：组合逻辑电路，根据传入的两个参数以及分支类型，判断当前数据是否满足跳转条件，满足时令分支有效信号 BranchE = 1

代码如下所示：

```
`include "Parameters.v"
module BranchDecisionMaking(
    input wire [2:0] BranchTypeE,
    input wire [31:0] Operand1,Operand2,
    output reg BranchE
    );
    // 请补全此处代码
    always@(*) begin
        case(BranchTypeE)
            `NOBRANCH: BranchE = 1'b0;
            `BEQ: begin
                BranchE = (Operand1 == Operand2)?  1'b1 : 1'b0;
                end
            `BNE: begin
                BranchE = (Operand1 != Operand2)?  1'b1 : 1'b0;
                end
            `BLT: begin
                BranchE = ($signed(Operand1) < $signed(Operand2))?  1'b1 : 1'b0;
                end
            `BLTU: begin
                BranchE = (Operand1 < Operand2)?  1'b1 : 1'b0;
                end
            `BGE: begin
                BranchE = ($signed(Operand1) >= $signed(Operand2))?  1'b1 : 1'b0;
                end
            `BGEU: begin
                BranchE = (Operand1 >= Operand2)?  1'b1 : 1'b0;
                end
            default: BranchE = 1'b0;
        endcase
    end
endmodule
```

6、DataEXT 模块：组合逻辑电路，根据 load 类型与位选信号 LoadedBytesSelect 将从数据存储器取出的数据截取需要的片段并将其扩展。能够实现非字对齐的 load 指令。根据每种情况编写代码即可。实现如下所示：

```
`include "Parameters.v"
module DataExt(
    input wire [31:0] IN,
    input wire [1:0] LoadedBytesSelect,
    input wire [2:0] RegWriteW, //输出到 RF 模块时按位或成为写使能信号,只要不为零就写有效
    output reg [31:0] OUT
    );
    // 请补全此处代码
    always@(*) begin
        case(RegWriteW)
            `NOREGWRITE: begin
                OUT = IN;
                end
            `LB: begin  //读取 8bit，符号扩展为 32 位
                if(LoadedBytesSelect == 2'b00) OUT = { {24{IN[7]}}, IN[7:0] };
                else if(LoadedBytesSelect == 2'b01) OUT = { {24{IN[15]}}, IN[15:8] };
                else if(LoadedBytesSelect == 2'b10) OUT = { {24{IN[23]}}, IN[23:16] };
                else if(LoadedBytesSelect == 2'b11) OUT = { {24{IN[31]}}, IN[31:24] };
                else ;
                end
            `LH: begin
                if(LoadedBytesSelect == 2'b00) OUT = { {16{IN[15]}}, IN[15:0] };
                else if(LoadedBytesSelect == 2'b01) OUT = { {24{IN[24]}}, IN[24:8] };
                else if(LoadedBytesSelect == 2'b10) OUT = { {24{IN[31]}}, IN[31:16] };
                else ;
                end
            `LW: begin
                OUT = IN;
                end
            `LBU: begin
                if(LoadedBytesSelect == 2'b00) OUT = { 24'b0, IN[7:0] };
                else if(LoadedBytesSelect == 2'b01) OUT = { 24'b0, IN[15:8] };
                else if(LoadedBytesSelect == 2'b10) OUT = { 24'b0, IN[23:16] };
                else if(LoadedBytesSelect == 2'b11) OUT = { 24'b0, IN[31:24] };
                else ;
                end
            `LHU: begin
                if(LoadedBytesSelect == 2'b00) OUT = { 16'b0, IN[15:0] };
                else if(LoadedBytesSelect == 2'b01) OUT = { 16'b0, IN[24:8] };
                else if(LoadedBytesSelect == 2'b10) OUT = { 16'b0, IN[31:16] };
                else ;
                end
            default: ;
        endcase
    end
endmodule
```

7、冲突处理模块：

根据不同的情况产生清空段寄存器信号以及 stall 停顿信号，再根据流水线中的指令情况产生两个前推信号 Forward1E，Forward2E，用于多选器选择。代码实现如下所示：

```
module HarzardUnit(
    input wire CpuRst, ICacheMiss, DCacheMiss,
    input wire BranchE, JalrE, JalD,
    input wire [4:0] Rs1D, Rs2D, Rs1E, Rs2E, RdE, RdM, RdW,
    input wire [1:0] RegReadE,
    input wire MemToRegE,
    input wire [2:0] RegWriteM, RegWriteW,
    output reg StallF, FlushF, StallD, FlushD, StallE, FlushE, StallM, FlushM,
StallW, FlushW,
    output reg [1:0] Forward1E, Forward2E
    );
    // 请补全此处代码
      initial StallF = 1'b0;initial   FlushF = 1'b0;
      initial   StallD = 1'b0;initial   FlushD = 1'b0;
      initial   StallE = 1'b0;initial   FlushE = 1'b0;
      initial   StallM = 1'b0;initial   FlushM = 1'b0;
      initial   StallW = 1'b0;initial   FlushW = 1'b0;
      initial   Forward1E = 2'b00;initial   Forward2E = 2'b00;

    always@(*) begin
        StallF = 1'b0;FlushF = 1'b0;
        StallD = 1'b0;FlushD = 1'b0;
        StallE = 1'b0;FlushE = 1'b0;
        StallM = 1'b0;FlushM = 1'b0;
        StallW = 1'b0;FlushW = 1'b0;
     end

    always@(*) begin
        StallF = 1'b0; FlushF = 1'b0; StallD = 1'b0; FlushD = 1'b0; StallE = 1'b0;
        FlushE = 1'b0; StallM = 1'b0; FlushM = 1'b0; StallW = 1'b0; FlushW = 1'b0;
        if(CpuRst) begin
            FlushF = 1'b1; FlushD = 1'b1; FlushE = 1'b1; FlushM = 1'b1;  FlushW = 1'b1;
        end
        else if(BranchE) begin /*分支，EXE 段执行分支，清除 ID,EX 寄存器，IF 取同一条指令*/
            FlushD = 1'b1;  FlushE = 1'b1; FlushM = 1'b1;
        end
        else if(JalrE) begin
            FlushD = 1'b1;  FlushE = 1'b1;
        end
        else if(JalD) begin
            FlushD = 1'b1;
        end
        else if(MemToRegE & ((RdE == Rs1D) || (RdE == Rs2D))) begin
            StallF = 1'b1;
            StallD = 1'b1;
            FlushE = 1'b1;
        end
    end

    //Forward 信号
    always@(*) begin
      /*Forward1E*/
        if((RegWriteM != `NOREGWRITE) && (RdM == Rs1E) && (RdM != 5'b0))
            Forward1E = 2'b10;
        else if((RegWriteW != `NOREGWRITE) && (RdW == Rs1E) && (RdW != 5'b0))
            Forward1E = 2'b01;
        else
            Forward1E = 2'b00;
      /*Forward2E*/
        if((RegWriteM != `NOREGWRITE) && (RdM == Rs2E) && (RdM != 5'b0))
            Forward2E = 2'b10;
        else if((RegWriteW != `NOREGWRITE) && (RdW == Rs2E) && (RdW != 5'b0))
            Forward2E = 2'b01;
        else
            Forward2E = 2'b00;
    end
endmodule
```

8、CSR 模块:

CSR 控制与状态寄存器组共用 12bit 寻址，因此最大支持 4096 个 CSR。所有的六条 CSR 指令都需要实现先读后写的特性，使用阻塞式赋值即可。代码实现如下所示：

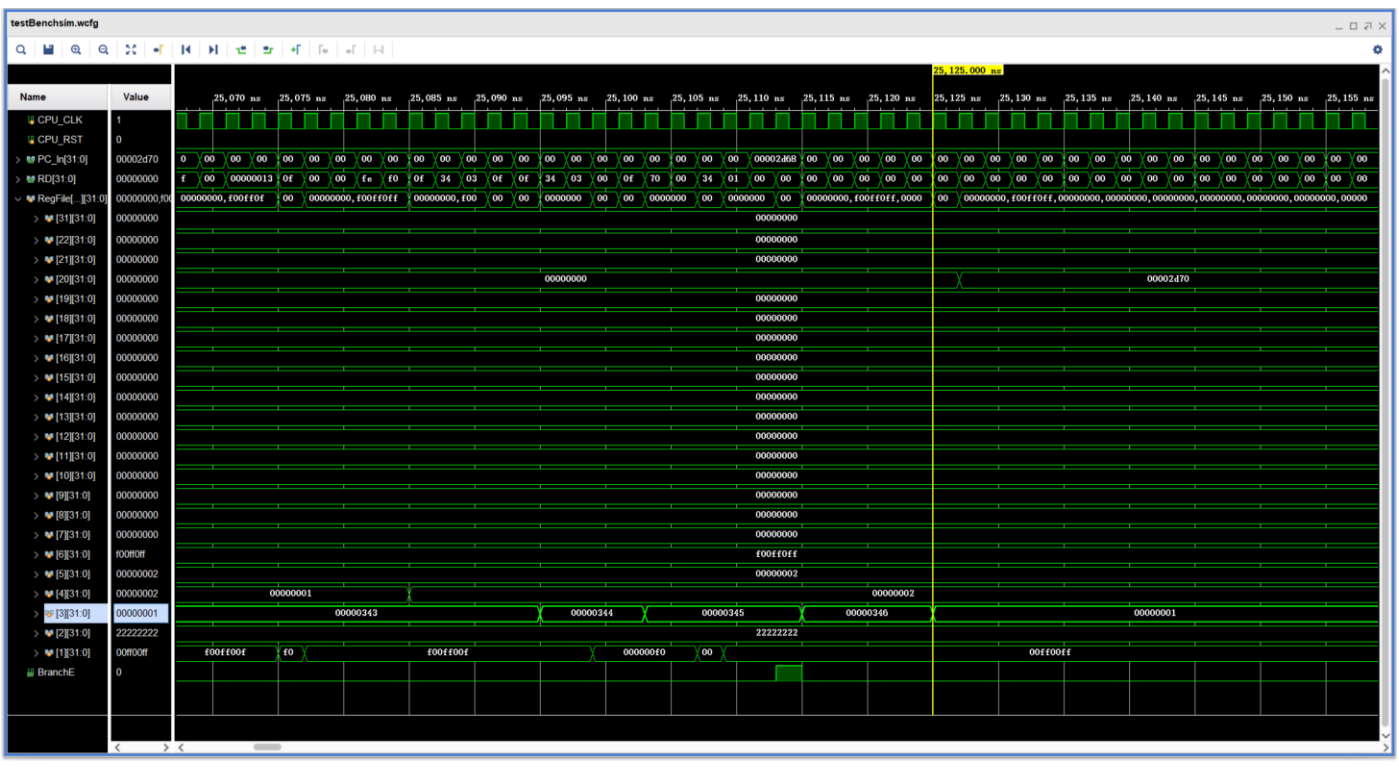
```
`include "Parameters.v"
module CS_RegisterFile(
    input wire clk,
    input wire rst,
    input wire [11:0] CSR_Addr,
    input wire [31:0] RegFileData,
    input wire [31:0] rsdata,
    input wire [2:0] CSRType,

    output wire [31:0] CSRout
);
    reg [31:0] CSReg[4095:0];
    integer i;
    reg [31:0] CSRout_reg;
    initial CSRout_reg = 0;

    always@(negedge clk or posedge rst) begin
        if(rst) begin
            for(i=0;i<4095;i=i+1) begin
                CSReg[i][31:0]<=32'b0;
            end
        end
        else begin
            case(CSRType)
              `CSRRW_Type: begin
                    CSRout_reg = CSReg[CSR_Addr];
                    CSReg[CSR_Addr] = RegFileData;
                end
              `CSRRS_Type: begin
                    CSRout_reg = CSReg[CSR_Addr];
                    CSReg[CSR_Addr] = CSReg[CSR_Addr] | RegFileData;
                end
              `CSRRC_Type: begin
                    CSRout_reg = CSReg[CSR_Addr];
                    CSReg[CSR_Addr] = CSReg[CSR_Addr] & (~RegFileData);
                end
               `CSRRWI_Type: begin
                    CSRout_reg = CSReg[CSR_Addr];
                    CSReg[CSR_Addr] = rsdata;
                end
               `CSRRSI_Type: begin
                    CSRout_reg = CSReg[CSR_Addr];
                    CSReg[CSR_Addr] = CSReg[CSR_Addr] | rsdata;
                end
               `CSRRCI_Type: begin
                    CSRout_reg = CSReg[CSR_Addr];
                    CSReg[CSR_Addr] = CSReg[CSR_Addr] & (~rsdata);
                end
               `NOTCSR_Type:  begin
                    CSRout_reg = 32'b0;
                end
              default: CSRout_reg = 32'b0;
            endcase
        end
    end
    assign CSRout = CSRout_reg;
endmodule
```

【实验结果】

阶段一和阶段二（不包含 CSR 指令）的波形仿真结果如下所示，只需要看 3 号寄存器的取值变化：



阶段三使用的测试文件如下所示：

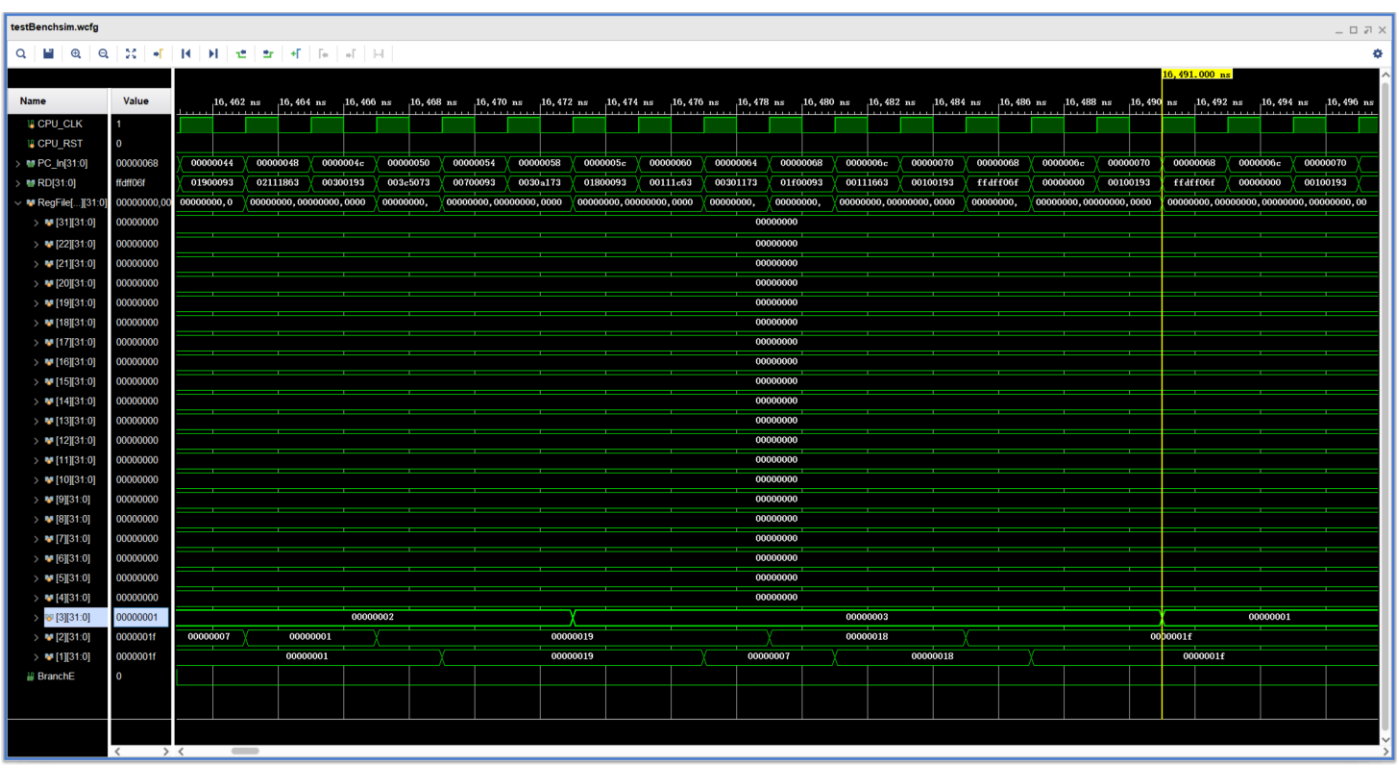
```
00010054 <test_0>:
   10054:   00000193          li  gp,0
   10058:   00f00093          li  ra,15
   1005c:   000090f3          csrw    ustatus,ra
   10060:   00003173          csrrc
sp,ustatus,zero
   10064:   06110063          beq sp,ra,100c4
<failed>
   10068:   000c7073          csrci   ustatus,24
   1006c:   00003173          csrrc
sp,ustatus,zero
   10070:   00700093          li  ra,7
   10074:   04111863          bne sp,ra,100c4
<failed>

00010078 <test_2>:
   10078:   00200193          li  gp,2
   1007c:   00100093          li  ra,1
   10080:   00209073          fsrm    ra
   10084:   002c6173          csrrsi  sp,frm,24
   10088:   02111e63          bne sp,ra,100c4
<failed>
   1008c:   00201173          fsrm    sp,zero
   10090:   01900093          li  ra,25
   10094:   02111863          bne sp,ra,100c4
<failed>

000100bc <success>:
   100bc:   00100193          li  gp,1
   100c0:   ffdff06f          j   100bc <success>

000100c4 <failed>:
   100c4:   0000006f          j   100c4 <failed>
```

以上指令与数据的仿真结果如下所示：



可见各寄存器值的变化情况与预期相符，CSR 功能完整且已解决冲突。