浙江大学

# 本科实验报告

| | |
|---|---|
| 课程名称: | 计算机组成与设计 |
| 姓　　名: | |
| 学　　院: | 信息与电子工程学院 |
| 专　　业: | 电子科学与技术 |
| 学　　号: | |
| 指导教师: | 屈民军 唐奕 |

2023 年 12 月 3 日

专业：电子科学与技术
姓名：
学号：
日期：2023.12.3
地点：教十一 400

# 浙江大学实验报告

课程名称：计算机组成与设计　指导老师：屈民军 唐奕　成绩：_____

实验名称：基于 RV32I 指令集的 RISC-V 微处理器设计　实验类型：设计实验

## 一、实验目的

（1）熟悉RISC-V 指令系统.
（2）了解提高CPU性能的方法。
（3）掌握流水线RISC-V 微处理器的工作原理。
（4）理解数据冒险、控制冒险的概念以及流水线冲突的解决方法。
（5）掌握流水线RISC-V微处理器的测试方法。
（6）了解软件实现数字系统的方法。

## 二、实验任务与要求

（一）基本要求
设计一个流水线RISC-V微处理器。要求如下：
（1）至少运行下列RV32I核心指令。
①算数运算指令：add、sub、addi
②逻辑运算指令：and、or、xor、slt、sltu、andi、ori、xori、slti、sltiu
③移位指令：sll、srl、sra、slli、srli、srai
④条件分支指令：beq、bne、blt、bge、bltu、begu
⑤无条件跳转指令：jal、jalr
⑥数据传送指令：lw、sw、lui、auipc
⑦空指令：nop
（2）采用5级流水线技术，对数据冒险实现转发或阻塞功能。
（3）在Nexys Video开发系统中实现RISC-V微处理器，要求CPU的运行速度大于25MHz。
（二）扩展要求
（1）要求设计的微处理器还能运行 lb、lh、ld、lbu、lhu、lwu、sb、sh 或 sd 等字节、半字和双字数据传送指令。
（2）要求设计的 CPU 增加异常（exception）、自陷（trap）、中断（interrupt）等处理方案。

# 三、设计原理及代码说明

## 1、指令译码模块（ID）的设计

指令译码模块的主要作用是从机器码中解析出指令，并根据解析结果输出各种控制信号。ID 模块主要由指令译码（Decode）、寄存器堆（Registers）、冒险检测、分支检测和加法器等组成。ID 模块的接口信息如表 30.3 所示。

表 30.3 ID 模块的输入/输出引脚说明

| 引 脚名称 | 方向 | 说明 |
|---|---|---|
| clk | Input | 系统时钟 |
| Instruction_id[31:0] | | 指令机器码 |
| PC_id[31:0] | | 指令指针 |
| RegWrite_wb | | 寄存器写允许信号，高电平有效 |
| rdAddr_wb[4:0] | | 寄存器的写地址。 |
| RegWriteData_wb[31:0] | | 写入寄存器的数据 |
| MemRead_ex | | 冒险检测的输入 |
| rdAddr_ex[4:0] | | |
| MemtoReg_id | Output | 决定回写的数据来源（0：ALU；1：存储器） |
| RegWrite_id | | 寄存器写允许信号，高电平有效 |
| MemWrite_id | | 存储器写允许信号，高电平有效 |
| MemRead_id | | 存储器读允许信号，高电平有效 |
| ALUCode_id[3:0] | | 决定 ALU 采用何种运算 |
| ALUSrcA_id | | 决定 ALU 的 A 操作数的来源（0：rs1；1：pc） |
| ALUSrcB_id[1:0] | | 决定 ALU 的 B 操作数的来源(2'b00：rs2；2'b01：imm；2'b10：常数 4) |
| Stall | | ID/EX 寄存器清空信号，高电平表示插入一个流水线气泡 |
| Branch | | 条件分支指令的判断结果，高电平有效 |
| Jump | | 无条件分支指令的判断结果，高电平有效 |
| IFWrite | | 阻塞流水线的信号，低电平有效 |
| BranchAddr[31:0] | | 分支地址 |
| Imm_id[31:0] | | 立即数 |
| rdAddr_id[4:0] | | 回写寄存器地址 |
| rs1Addr_id[4:0] | | 两个数据寄存器地址 |
| rs2Addr_id[4:0] | | |
| rs1Data_id[31:0] | | 寄存器两个端口输出数据 |
| rs2Data_id[31:0] | | |

须知此处的 BranchAddr 就是别处的 JumpAddr 信号，用作表示分支地址。

### （1）寄存器堆（Registers）子模块的设计

RV32I 系统的寄存器结构采用标准的 32 位寄存器堆，共 32 个宽度为 32bit 的寄存器，标号为 0~31。其中，第 0 寄存器永远为常数 0。这些寄存器通过寄存器号进行读写存取。寄存器堆的原理框图如图 30.8 所示。因为读取寄存器不会更改其内容，故只需提供寄存号即

可读出该寄存器内容。读取端口采用数据选择器即可实现读取功能。

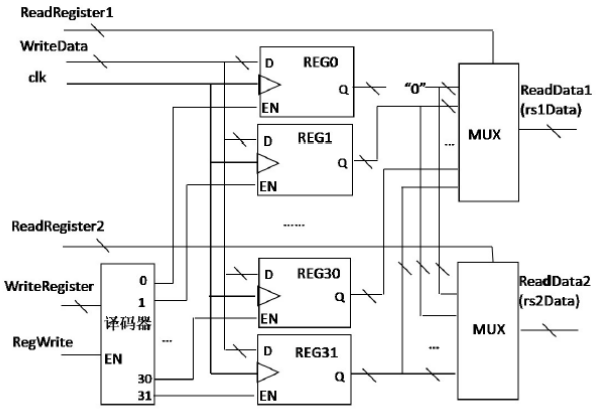
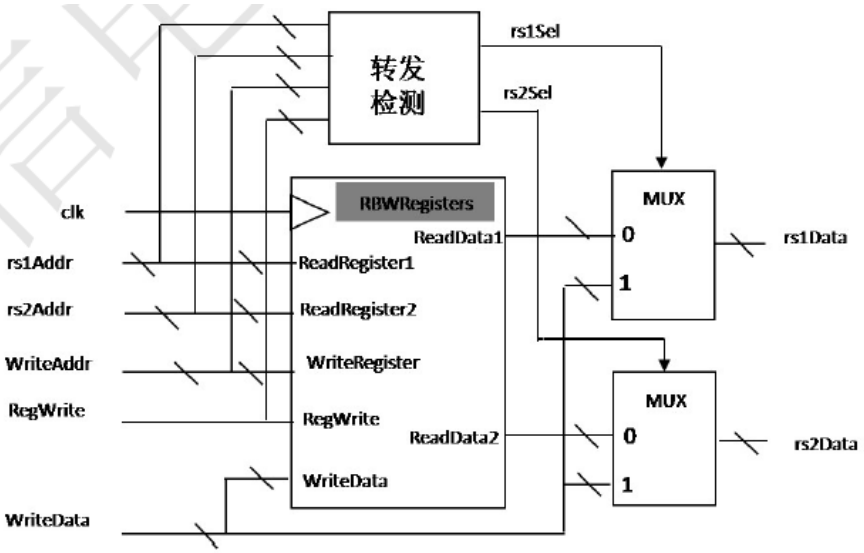



图 30.8　寄存器堆的原理框图

对于往寄存器里写数据，需要目标寄存器号(WriteRegister)、 待写入数据(WriteData)、 写允许信号(RegWrite)三个变量。图 30.8 中 5 位二进制译码器完成地址译码，其输出控制目标寄存器的写使能信号 EN，决定将数据 WriteData 写入哪个寄存器。

```
//Read before write Registers
module RBWRegisters (
    input [4:0] ReadRegister1,ReadRegister2, WriteRegister,
    input clk, RegWrite,
    input [31:0] WriteData,
    output [31:0] ReadData1, ReadData2
);
reg [31:0] regs [31:0]; //define a 32*32 registers
assign ReadData1= (ReadRegister1==5'b0)?32'b0:regs[ReadRegister1]; //rs1Data
assign ReadData2= (ReadRegister2==5'b0)?32'b0:regs[ReadRegister2]; //rs2Data
always @(posedge clk) begin
    if(RegWrite)
        regs[WriteRegister]<=WriteData; //act as a decoder
end
endmodule
```

上面代码给出了一个 Read Before Write 寄存器的代码。在流水线型 CPU 设计中，寄存器堆设计还应解决三阶数据相关的数据转发问题。当满足三阶数据相关条件时，寄存器具有 Read After Write 特性。设计时，只需要在 Read Before Write 寄存器堆的基础上添加少量电路就可实现 Read After Write 特性：




图中转发检测电路的输出表达式为：

rs1Sel= RegWrite && (WriteAddr !=0) && (WriteAddr ==rs1Addr)　　(30.1)

rs2Sel= RegWrite && (WriteAddr !=0) && (WriteAddr ==rs2Addr)　　(30.2)

因此寄存器堆模块代码为：

```
//Read after write registers
module Registers (
    input clk, RegWrite,
    input [4:0] rs1Addr, rs2Addr, WriteAddr,
    input [31:0] WriteData,
    output [31:0] rs1Data, rs2Data
);
wire rs1Sel, rs2Sel;
wire [31:0] ReadData1, ReadData2;
RBWRegisters RBWr(
    .ReadRegister1(rs1Addr), .ReadRegister2(rs2Addr), .WriteRegister(WriteAddr),
    .clk(clk), .RegWrite(RegWrite),
    .WriteData(WriteData),
    .ReadData1(ReadData1), .ReadData2(ReadData2)
);
//forward detction
assign rs1Sel= RegWrite && (WriteAddr!=0) && (WriteAddr==rs1Addr);
assign rs2Sel= RegWrite && (WriteAddr!=0) && (WriteAddr==rs2Addr);
//output
assign rs1Data= (rs1Sel==1'b0)?ReadData1:WriteData;
assign rs2Data= (rs2Sel==1'b0)?ReadData2:WriteData;
endmodule
```

**（2）指令译码（包含立即数产生电路）子模块的设计**

该子模块主要作用是根据指令确定各个控制信号的值，同时产生立即数 Imm 和偏移量

offset。该模块是一个组合电路。

RISC-V 将指令分为 R、I、S、SB、U、UJ 等六类。因此，设置 R_type、I_type、SB_type、LW、JALR、SW、LUI、AUIPC 和 JAL 等变量来表示指令类型，各变量的值由式 30.3 决定。

$$\begin{cases} R\_type = (op == R\_type\_op) \\ I\_type = (op == I\_type\_op) \\ SB\_type = (op == SB\_type\_op) \\ LW = (op == LW\_op) \\ JALR = (op == JALR\_op) \\ SW = (op == SW\_op) \\ LUI = (op == LUI\_op) \\ AUIPC = (op == AUIPC\_op) \\ JAL = (op == JAL\_op) \end{cases} \tag{30.3}$$

```
wire R_type, I_type, SB_type, LW, JALR, SW, LUI, AUIPC, JAL;
//Instruction judgment
assign R_type= (op==R_type_op);     assign I_type= (op==I_type_op);
assign SB_type= (op==SB_type_op);   assign LW= (op==LW_op);
assign JALR= (op==JALR_op);         assign SW= (op==SW_op);
assign LUI= (op==LUI_op);           assign AUIPC= (op==AUIPC_op);
assign JAL= (op==JAL_op);
```

① 只有 LW 指令读取存储器且回写数据取自存储器，所以有

$$MemtoReg\_id=LW \tag{30.4}$$

$$MemRead\_id =LW \tag{30.5}$$

② 只有 SW 指令会对存储器写数据，所以有

$$MemWrite\_id =SW \tag{30.6}$$

③ 需要进行回写的指令类型有 R_type、I_type、LW、JALR、LUI、AUIPC 和 JAL。所以有

$$RegWrite\_id = R\_type \,||\, I\_type \,||\, LW \,||\, JALR \,||\, LUI \,||\, AUIPC \,||\, JAL \tag{30.7}$$

④ 只有 JALR 和 JAL 两条无条件分支指令，所以有

$$Jump=JALR \,||\, JAL \tag{30.8}$$

```
assign MemtoReg= LW;
assign MemRead= LW;
assign MemWrite= SW;
assign RegWrite= R_type || I_type || LW || JALR || LUI || AUIPC || JAL;
assign Jump= JALR || JAL;
```

⑤ 操作数 A 和 B 的选择信号的确定

分析各类指令，可得到表 30.4 操作数选择的功能表。

表 30.4 操作数选择信号的功能表

| 类型 | ALUSrcA_id | ALUSrcB_id[1:0] | 说明 |
|---|---|---|---|
| R_type | 0 | 2'b00 | rd=rs1 op rs2 |
| I_type | 0 | 2'b 01 | rd=rs1 op imm |
| LW | 0 | 2'b 01 | rs1 + imm |
| SW | 0 | 2'b 01 | rs1 + imm |
| JALR | 1 | 2'b 10 | rd=pc + 4 |
| JAL | 1 | 2'b 10 | rd=pc + 4 |
| LUI | 1'bx | 2'b 01 | rd= imm |
| AUIPC | 1 | 2'b 01 | rd=pc + imm |

从表 30.4 可获得 ALUSrcA_id 和 ALUSrcB_id[1:0]表达式。

ALUSrcA_id＝JALR || JAL || AUIPC (30.9)

ALUSrcB_id [1] = JAL JALR

ALUSrcB_id[0] = ~(R _ type || JAL || JALR） (30.10)

```
assign ALUSrcA= JALR || JAL || AUIPC;
assign ALUSrcB[1]= JAL || JALR;
assign ALUSrcB[0]= ~(R_type || JAL || JALR);
```

⑥ ALUCode 的确定

除了条件分支指令，其它指令都需要 ALU 执行运算，共有 11 种不同运算， ALUCode 信号需用 4 位二进制表示。最主要为加法运算，设为默认算法，ALUCode 的功能表如表 30.5 所示。注意：表中 funct7[6]与 funct6[5]在指令中为同一位置，即 instruction[30]。

表 30.5 ALUCode 的功能表

| R_type | I_type | LUI | funct3 | funct7[6] (funct6[5]) | ALUCode | 备注 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 3'o0 | 0 | 4'd 0 | 加 |
| 1 | 0 | 0 | 3'o0 | 1 | 4'd 1 | 减 |
| 1 | 0 | 0 | 3'o1 | 0 | 4'd 6 | 左移 A << B |
| 1 | 0 | 0 | 3'o2 | 0 | 4'd9 | A<B?1:0 |
| 1 | 0 | 0 | 3'o3 | 0 | 4'd10 | A<B?1:0（无符号数） |
| 1 | 0 | 0 | 3'o4 | 0 | 4'd 4 | 异或 |
| 1 | 0 | 0 | 3'o5 | 0 | 4'd 7 | 右移 A >> B |
| 1 | 0 | 0 | 3'o5 | 1 | 4'd 8 | 算术右移 A >>> B |
| 1 | 0 | 0 | 3'o6 | 0 | 4'd 5 | 或 |
| 1 | 0 | 0 | 3'o7 | 0 | 4'd 3 | 与 |
| 0 | 1 | 0 | 3'o0 | x | 4'd 0 | 加 |
| 0 | 1 | 0 | 3'o1 | x | 4'd 6 | 左移 |
| 0 | 1 | 0 | 3'o2 | x | 4'd9 | A<B?1:0 |
| 0 | 1 | 0 | 3'o3 | x | 4'd10 | A<B?1:0（无符号数） |
| 0 | 1 | 0 | 3'o4 | x | 4'd 4 | 异或 |
| 0 | 1 | 0 | 3'o5 | 0 | 4'd 7 | 右移 A >> B |
| 0 | 1 | 0 | 3'o5 | 1 | 4'd 8 | 算术右移 A >>> B |
| 0 | 1 | 0 | 3'o6 | x | 4'd 5 | 或 |
| 0 | 1 | 0 | 3'o7 | x | 4'd 3 | 与 |
| 0 | 0 | 1 | x | x | 4'd 2 | 送数:ALUResult=B |
| 其它 | | | | | 4'd 0 | 加 |

```
always @(*) begin
    if((R_type==1 && I_type==0 && LUI==0)||(R_type==0 && I_type==1 && LUI==0))begin
        case (func3)
            ADD_funct3:begin
                if(R_type==1 && I_type==0 && LUI==0)
                    ALUCode= (func7==1'b0)?4'd0:4'd1;
                else
                    ALUCode= 4'd0;
            end
            SLL_funct3: ALUCode= 4'd6;
            SLT_funct3: ALUCode= 4'd9;
            SLTIU_funct3: ALUCode= 4'd10;
            XOR_funct3: ALUCode= 4'd4;
            SRL_funct3: ALUCode=(func7==1'b0)? 4'd7:4'd8;
            ORI_funct3: ALUCode= 4'd5;
            ANDI_funct3: ALUCode= 4'd3;
        endcase
    end
    else if(R_type==0 && I_type==0 && LUI==1)
            ALUCode= 4'd2;
    else ALUCode= 4'd0;
end
```

⑦ 立即数产生电路（ImmGen）设计

I_type、SB_type、LW、JALR、SW、LUI、AUIPC 和 JAL 这几类指令均用到立即数。由于 I_type 的算术逻辑运算与移位运算指令的立即数构成方法不同，这里再设定一个变量 Shift 来区分两者。Shift=1 表示移位运算，否为则算术逻辑运算。Shift 值由式(30.11)计算。

Shift=（funct3==1）||（funct3==5） (30.11)

立即数构成和扩展方法如表 30.6 所示，表中的 inst 即 instruction。

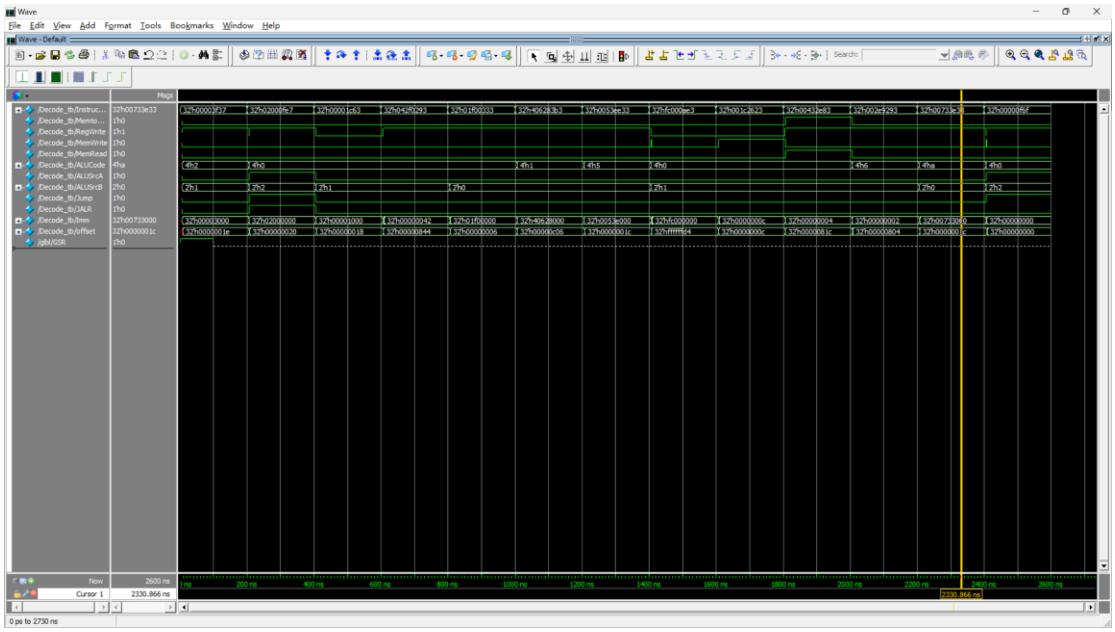
表 30.6 立即数产生方法

| 类别 | Shift | Imm | offset |
|---|---|---|---|
| I_type | 1 | {26'd0,inst[25:20]} | - |
| I_type | 0 | {{20{inst[31]}},inst[31:20]} | - |
| LW | x | | - |
| JALR | x | - | {{20{inst[31]}},inst[31:20]} |
| SW | x | {{20{inst[31]}},inst[31:25],inst[11:7]} | - |
| JAL | x | - | {{11{inst[31]}}, inst[31], inst[19:12] , inst[20], inst[30:21], 1'b0} |
| LUI | x | {inst[31:12], 12'd0} | - |
| AUIPC | x | | - |
| SB_type | x | - | {{19{inst[31]}}, inst[31], inst[7],inst[30:25] , inst[11:8], 1'b0} |

```
wire shift= (func3==3'd1) || (func3==3'd5);
always @(*) begin
    if(I_type)begin
        immediate= (shift==1'b1)?{26'd0, inst[25:20]}:{{20{inst[31]}}, inst[31:20]};
        offset= 32'bz;
    end
    else if(LW)begin
        immediate= {{20{inst[31]}}, inst[31:20]};
        offset= 32'bz;
    end
    else if(JALR)begin
        immediate= 32'bz;
        offset= {{20{inst[31]}}, inst[31:20]};
        end
    else if(SW)begin
        offset= 32'bz;
```

```
        immediate= {{20{inst[31]}}, inst[31:25], inst[11:7]};
    end
    else if(JAL)begin
        offset= {{11{inst[31]}}, inst[31], inst[19:12], inst[20], inst[30:21], 1'b0};
        immediate= 32'bz;
    end
    else if(LUI || AUIPC)begin
        immediate= {inst[31:12], 12'd0};
        offset= 32'bz;
    end
    else if(SB_type)begin
        offset= {{19{inst[31]}}, inst[31], inst[7], inst[30:25], inst[11:8], 1'b0};
        immediate= 32'bz;
    end
end
```

本译码模块（Decode.v）的仿真结果如下：



Decode 模块仿真结果的分析见报告“实验结果及分析”部分。

**（3）分支检测（Branch Test）电路的设计**

分支检测电路主要用于判断分支条件是否成立，在 Verilog HDL 可以用比较运算符号“>”、“==” 和“<”描述，但要注意符号数和无符号数的处理方法不同。在这里，我们用加法器来实现。

① 用一个 32 位加法器完成 rs1Data+（~rs2Data）+1（即 rs1Data-rs2Data），设结果为 sum [31:0] 。

② 确定比较运算的结果。对于比较运算来说，如果最高位不同，即 rs1Data [31]≠rs2Data [31]，可根据 rs1Data [31]、rs2Data [31]决定比较结果，但是应注意符号数、无符号数的最高位 rs1Data [31]、rs2Data [31]代表意义不同。若两数最高位相同，则两数之差不会溢出，所

以比较运算结果可由两个操作数之差的符号位 sum[31]决定。

在符号数比较运算中，rs1Data<rs2Data 有以下两种情况：

a) rs1Data 为负数、rs2Data 为 0 或正数: rs1Data[31]&&(~rs2Data[31])

b) rs1Data、rs2Data 符号相同，sum 为负: (rs1Data[31]~^rs2Data[31]) &&| sum[31]

因此，符号数 rs1Data<rs2Data 比较运算结果为

isLT=rs1Data[31]&&(~rs2Data[31]) Il (rs1Data[31]~^rs2Data[31])&& sum[31] (30. 12)

同样地，无符号数比较运算中，rs1Data<rs2Data 有以下两种情况：

a) rs1Data 最高位为 0、rs2Data 最高位为 1：(~rs1Data[31] )&& rs2Data[31]

b) rs1Data、rs2Data 最高位相同，sum 为负：(rs1Data[31]~^rs2Data[31]) && sum[31]

因此，无符号数比较运算结果为

isLTU= (~rs1Data[31] )&& rs2Data[31] || (rs1Data[31] ~^rs2Data[31]) && sum[31] (30.13)

最后用数据选择器完成式(30.14)即可完成分支检测。

$$
Branch = \begin{cases} \sim(|sum[31:0]); & SB\_type \ \&\& \ (funct3 == beq\_funct3) \\ |sum[31:0]; & SB\_type \ \&\& \ (funct3 = bne\_funct3) \\ isLT; & SB\_type \ \&\& (funct3 = blt\_funct3) \\ \sim isLT; & SB\_type \ \&\& \ (funct3 = bge\_funct3) \\ isLTU; & SB\_type \ \&\& \ (funct3 = bltu\_funct3) \\ \sim isLTU; & SB\_type \ \&\& \ (funct3 = bgeu\_funct3) \\ 0 & others \end{cases} \tag{30.14}
$$

```
wire [31:0] sum= rs1Data+(~rs2Data)+1;
wire isLT= (rs1Data[31] && (~rs2Data[31])) ||((rs1Data[31]~^rs2Data[31]) && sum[31]);
wire isLTU= ((~rs1Data[31]) && rs2Data[31]) ||((rs1Data[31]~^rs2Data[31]) && sum[31]);

parameter beq_func3= 3'o0;    parameter bne_func3= 3'o1;
parameter blt_func3= 3'o4;    parameter bge_func3= 3'o5;
parameter bltu_func3= 3'o6;   parameter bgeu_func3= 3'o7;
always @(*) begin
    if(SB_type) begin
        if(func3==beq_func3)
            Branch= ~(|sum[31:0]);
        else if(func3==bne_func3)
            Branch= |sum[31:0];
        else if(func3==blt_func3)
            Branch= isLT;
        else if(func3==bge_func3)
            Branch= ~isLT;
        else if(func3==bltu_func3)
            Branch= isLTU;
        else if(func3==bgeu_func3)
            Branch= ~isLTU;
    end
    else
        Branch= 1'b0;
end
```

**（4）冒险检测功能电路（Hazard Detector）的设计**

由前面分析可知，冒险成立的条件为：

① 上一条指令必须是 lw 指令（MemRead_ex=1）；

② 两条指令读写同一个寄存器（rdAddr_ex=rs1Addr_id 或 rdAddr_ex=rs2Addr_id）。

当冒险成立应清空 ID/EX 寄存器并且阻塞流水线 ID 级、IF 级流水线，所以有

Stall=((rdAddr_ex==rs1Addr_id) || (rdAddr_ex==rs2Addr_id)) && MemRead_ex (30.15)

IFWrite= ~Stall (30.16)

在用 VerilogHDL 描述 ID 模块时，冒险检测功能电路（Hazard Detector）等功能单元比较简单的电路，直接在 ID 顶层描述。

**（5）ID 顶层模块的设计**

在 ID 模块，除了将各子模块连接，还需要根据 JALR 信号决定基地址，并通过 BranchAddr=base_addr+offset 来计算跳转地址，为 SB 型指令使用。

完整代码如下：

```
module ID (
    input clk, RegWrite_wb, MemRead_ex,
    input [31:0] Instruction_id, PC_id, RegWriteData_wb,
    input [4:0] rdAddr_wb, rdAddr_ex,
    output MemtoReg_id, RegWrite_id, MemWrite_id, MemRead_id,
           ALUSrcA_id, Stall, Branch, Jump, IFWrite,
    output [3:0] ALUCode_id,
    output [1:0] ALUSrcB_id,
    output [31:0] BranchAddr, Imm_id, rs1Data_id, rs2Data_id,
    output [4:0] rdAddr_id, rs1Addr_id, rs2Addr_id
);
//Registers
Registers Regs(
    .clk(clk), .RegWrite(RegWrite_wb),
    .rs1Addr(rs1Addr_id), .rs2Addr(rs2Addr_id), .WriteAddr(rdAddr_wb),
    .WriteData(RegWriteData_wb),
    .rs1Data(rs1Data_id), .rs2Data(rs2Data_id)
);

//Decode(ImmGen)
Decode Decode1(
    .Instruction(Instruction_id), .MemtoReg(MemRead_id),
    .RegWrite(RegWrite_id), .MemWrite(MemWrite_id),
    .MemRead(MemRead_id), .ALUCode(ALUCode_id), .ALUSrcA(ALUSrcA_id),
    .ALUSrcB(ALUSrcB_id), .Jump(jump), .JALR(JALR),
    .Imm(Imm_id), .offset(offset)
);
assign BranchAddr= offset + (JALR==1'b0)?PC_id:rs1Data_id; //JumpAddr
```

```
//Branch Test
BranchTest BT(
    .SB_type(SB_type), .func3(func3),
    .rs1Data(rs1Data_id), .rs2Data(rs2Data_id),
    .Branch(Branch)
);

//Hazard Detector
assign Stall= ((rdAddr_ex==rs1Addr_id) || (rdAddr_ex==rs2Addr_id)) && (MemRead_ex);
assign IFWrite= ~Stall;

assign rdAddr_id= Instruction_id[11:7];
assign rs1Addr_id= Instruction_id[19:15];
assign rs2Addr_id= Instruction_id[24:20];
endmodule
```

## 2、执行模块（EX）的设计

执行模块主要由 ALU 子模块、数据前推电路（Forwarding）及若干数据选择器组成。执行模块的接口信息如表 30. 7 所示。

表 30. 7　EX 模块的输入/输出引脚说明

| 引脚名称 | 方向 | 说明 |
|---|---|---|
| ALUCode_ex[3:0] | Input | 决定 ALU 采用何种运算 |
| ALUSrcA_ex | | 决定 ALU 的 A 操作数的来源（rs1、PC） |
| ALUSrcB_ex[1:0] | | 决定 ALU 的 B 操作数的来源(rs2、imm 和常数 4) |
| Imm_ex[31:0] | | 立即数 |
| rs1Addr_ex[4:0] | | rs1 寄存器地址 |
| rs2Addr_ex[4:0] | | rs2 寄存器地址 |
| rs1Data_ex[31:0] | | rs1 寄存器数据 |
| rs2Data_ex[31:0] | | rs2 寄存器数据 |
| PC_ex[31:0] | | 指令指针 |
| RegWriteData_wb[31:0] | | 写入寄存器的数据 |
| ALUResult_mem[31:0] | | ALU 输出数据 |
| rdAddr_mem[4:0] | | 寄存器的写地址 |
| rdAddr_wb[4:0] | | |
| RegWrite_mem | | 寄存器写允许信号 |
| RegWrite_wb | | |
| ALUResult_ex[31:0] | Output | ALU 运算结果 |
| MemWriteData_ex[31:0] | | 存储器的回写数据 |
| ALU_A [31:0] | | ALU 操作数，测试时使用 |
| ALU_B [31:0] | | |

### （1）ALU 子模块的设计

算术逻辑运算单元（ALU）提供 CPU 的基本运算能力，如加、减、与、或、比较、移位等。具体而言，ALU 输入为两个操作数 A、B 和控制信号 ALUCode，由控制信号 ALUCode 决定采用何种运算，运算结果为 ALUResult。整理表 30.5 所示的 ALUCode 的功能表，可

得到 ALU 的功能表，如表 30.8 所示。

表 30.8　ALU 的功能表

| ALUCode | ALUResult |
|---|---|
| 4'b0000 | A + B |
| 4'b0001 | A-B |
| 4'b0010 | B |
| 4'b0011 | A&B |
| 4'b0100 | A ^ B |
| 4'b0101 | A \| B |
| 4'b0110 | A << B |
| 4'b0111 | A >> B |
| 4'b1000 | A>>>B |
| 4'b1001 | A<B? 1:0，其中 A、B 为有符号数 |
| 4'b1010 | A<B? 1:0，其中 A、B 为无符号数 |

如表 30.8 所示，ALU 需执行多种运算，为了提高运算速度，本设计可同时进行各种运算，再根据 ALUCode 信号选出所需结果。ALU 的基本结构如图 30.10 所示。

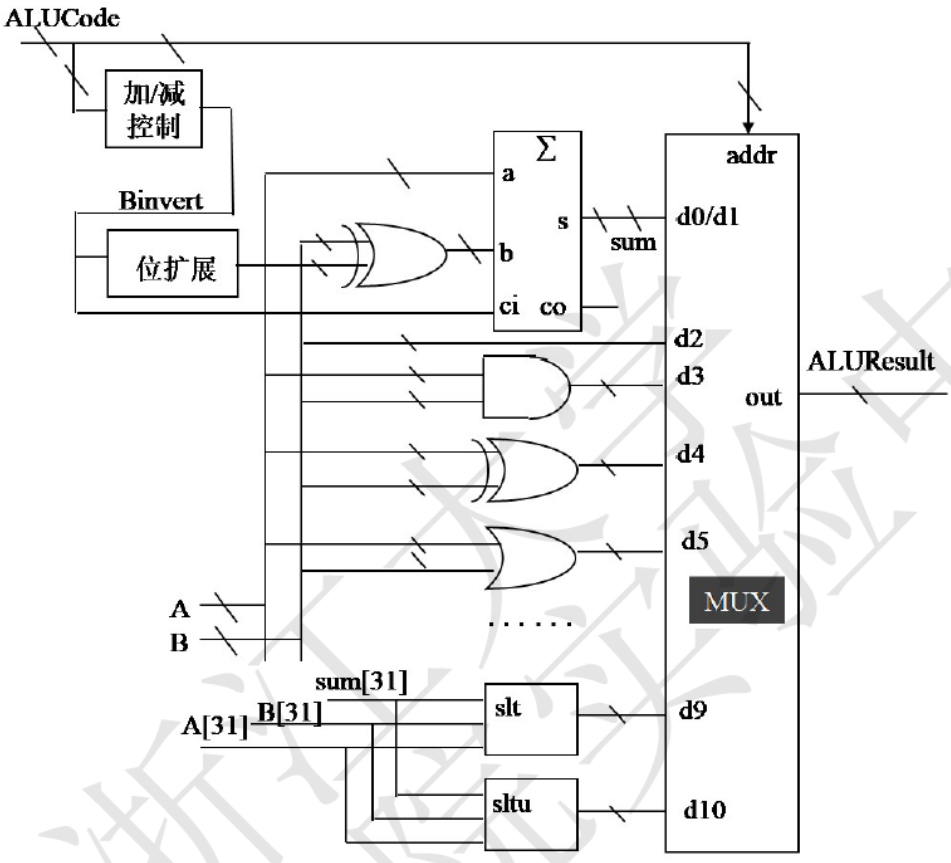



图 30.10　ALU 结构框图

①　加、减电路的设计考虑

减法、比较(slt、sltu)均坷用加法器和必要辅助电路来实现。图 30.10 中的 Binvert 信号控制加减运算；若 Binvert 信号为低电平，则实现加法运算：sum=A+B；若 Binvert 信号为高

电平，则电路为减法运算 sum=A-B。除加法外，减法、比较和分支指令都应使电路工作在减法状态，所以：

$$Binvert=\sim (ALUCode==0) \tag{30.17}$$

② 比较电路的设计考虑

比较电路的设计方法已在分支检测电路介绍参考式(30.12)和式(30.13)可确定 slt 和 sltu 两条件的比较结果。

③ 算术右移运算电路的设计考虑

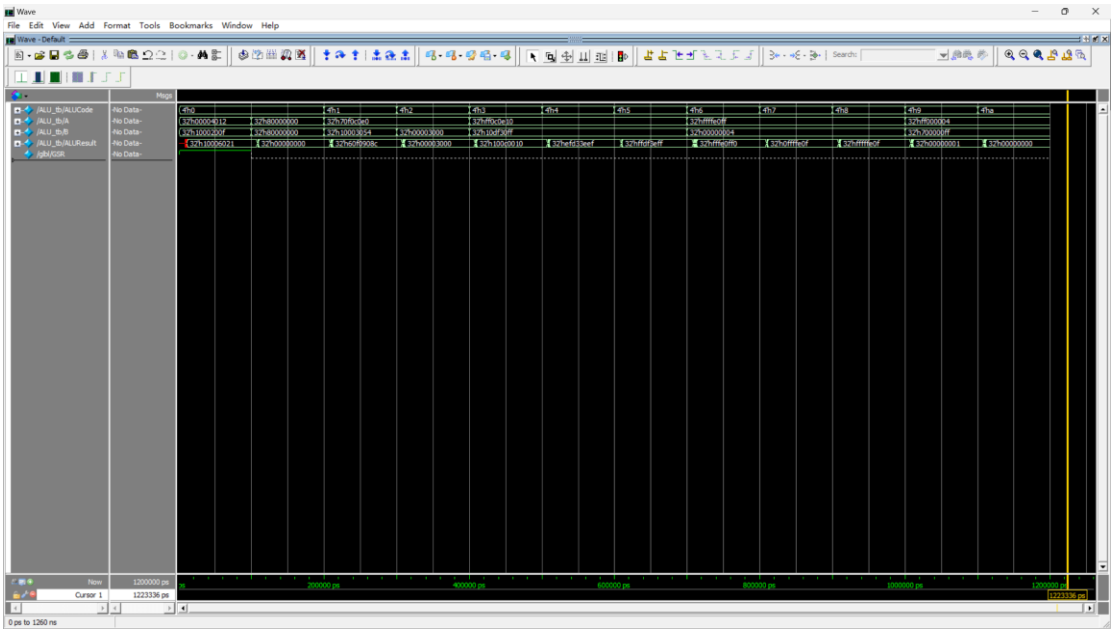
Verilog HDL 的算术右移的运算符是“>>>”。要实现算术右移应注意，被移位的对象必须定义是 reg 类型，但是在 sra 指令，被移位的对象操作数 A 为输入信号，不能定义为 reg 类型。因此，必须引入 reg 类型中间变量 A_ reg，相应的 Verilog HDL 语句为

```
reg signed[31:0] A_ reg;
always @ (*)  begin A_ reg=A;  end
```

引入 reg 类型的中间变量 A_ reg 后，就可对 A_ reg 进行算术右移操作。

```
wire Binvert= ~(ALUCode==0);
wire [31:0] b= {32{Binvert}}^B;
reg signed[31:0] A_reg;
wire [31:0] sum;
adder_32bits adder(
    .a(A), .b(b), .ci(Binvert), .s(sum)
);
always @(*) begin
    A_reg= A;
end
always @(*) begin
    case(ALUCode)
        4'b0000: ALUResult= sum;
        4'b0001: ALUResult= sum;
        4'b0010: ALUResult= B;
        4'b0011: ALUResult= A&B;
        4'b0100: ALUResult= A^B;
        4'b0101: ALUResult= A|B;
        4'b0110: ALUResult= A_reg<<B;
        4'b0111: ALUResult= A_reg>>B;
        4'b1000: ALUResult= A>>>B;
        4'b1001: ALUResult= (A[31]&&(~B[31]))||((A[31]~^B[31]) && sum[31]);
        4'b1010: ALUResult= ((~A[31])&&B[31])||((A[31]~^B[31]) && sum[31]);
    endcase
end
```

ALU 模块（ALU.v）的仿真结果如下：



ALU 仿真结果的分析见报告“实验结果及分析”部分。

**（2）数据前推电路的设计**

操作数 A 和 B 分别由数据选择器决定，数据选择器地址信号 ForwardA、ForwardB 的含义如表 30.9 所示。

表 30.9　前推电路输出信号的含义

| 地　　址 | 操作数来源 | 说　　明 |
|---|---|---|
| ForwardA= 2’b00 | rs1Data_ex | 操作数 A 来自寄存器堆 |
| ForwardA=2’b01 | RegWriteData_wb | 操作数 A 来自二阶数据相关的转发数据 |
| ForwardA= 2’b10 | ALUResult_mem | 操作数 A 来自一阶数据相关的转发数据 |
| ForwardB= 2’b00 | rs2Data_ex | 操作数 B 来自寄存器堆 |
| ForwardB= 2’b01 | RegWriteData_wb | 操作数 B 来自二阶数据相关的转发数据 |
| ForwardB=2’b10 | ALUResult_mem | 操作数 B 来自一阶数据相关的转发数据 |

由前面介绍的一、二阶数据相关判断条件，不难得到

$$\begin{cases} \text{ForwardA}[0] = \text{RegWrite\_wb} \,\&\&\, (\text{rdAddr\_wb} != 0) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_mem} != \text{rs1Addr\_ex}) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_wb} == \text{rs1Addr\_ex}) \\ \text{ForwardA}[1] = \text{RegWrite\_mem} \,\&\&\, (\text{rdAddr\_mem} != 0) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_mem} == \text{rs1Addr\_ex}) \end{cases} \tag{30.18}$$

$$\begin{cases} \text{ForwardB}[0] = \text{RegWrite\_wb} \,\&\&\, (\text{rdAddr\_wb} != 0) \,\&\&\, \\ \qquad\qquad (\text{rd\_mem} != \text{rs2Addr\_ex}) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_wb} == \text{rs2Addr\_ex}) \\ \text{ForwardB}[1] = \text{RegWrite\_mem} \,\&\&\, (\text{rdAddr\_mem} != 0) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_mem} == \text{rs2Addr\_ex}) \end{cases} \tag{30.19}$$

EX 模块通过一系列数据选择器决定 ALU 的两个源操作数，并且 ALU 运算结果输出到 ALUResult_ex 中。

```
module Forwarding (
    input RegWrite_wb, RegWrite_mem,
    input [4:0] rdAddr_wb, rs1Addr_ex, rdAddr_mem, rs2Addr_ex,
    output [1:0] ForwardA, ForwardB
);
    assign ForwardA[0]= RegWrite_wb && (rdAddr_wb!=0) && (rdAddr_mem!=rs1Addr_ex)
                      && (rdAddr_wb==rs1Addr_ex);
    assign ForwardA[1]= RegWrite_mem && (rdAddr_mem!=0) && (rdAddr_mem==rs1Addr_ex);
    assign ForwardB[0]= RegWrite_wb && (rdAddr_wb!=0) && (rdAddr_mem!=rs2Addr_ex)
                      && (rdAddr_wb==rs2Addr_ex);
    assign ForwardB[1]= RegWrite_mem && (rdAddr_mem!=0) && (rdAddr_mem==rs2Addr_ex);

endmodule
```

**（3）EX 顶层模块的设计**

EX 顶层模块中，除了 ALU 和 Forwarding 子模块，还有一个对 ALU 的两操作数进行选择的选择器电路。

```
module EX (
    input [3:0] ALUCode_ex,
    input ALUSrcA_ex, RegWrite_mem, RegWrite_wb,
    input [1:0] ALUSrcB_ex,
    input [31:0] Imm_ex, rs1Data_ex, rs2Data_ex, PC_ex, RegWriteData_wb,
                 ALUResult_mem,
    input [4:0] rs1Addr_ex, rs2Addr_ex, rdAddr_mem, rdAddr_wb,
    output [31:0] ALUResult_ex, MemWriteData_ex,
    reg [31:0] ALU_A, ALU_B
);
// deciding ALU_A and ALU_B
reg [31:0] A_0, B_0;
wire [1:0] ForwardA, ForwardB;
always @(*) begin
    case (ForwardA)
        2'b00: A_0= rs1Data_ex;
        2'b01: A_0= RegWriteData_wb;
        2'b10: A_0= ALUResult_mem;
    endcase
    case (ForwardB)
        2'b00: B_0= rs2Data_ex;
        2'b01: B_0= RegWriteData_wb;
        2'b10: B_0= ALUResult_mem;
    endcase
end
always @(*) begin
    ALU_A= (ALUSrcA_ex==1'b0)?A_0:PC_ex;
    ALU_B= (ALUSrcB_ex==2'b00)?B_0:((ALUSrcB_ex==2'b01)?Imm_ex:32'd4);
end
```

```
ALU ALU(
    .A(ALU_A), .B(ALU_B),
    .ALUCode(ALUCode_ex), .ALUResult(ALUResult_ex)
);

Forwarding Forwarding(
    .RegWrite_wb(RegWrite_wb), .RegWrite_mem(RegWrite_mem),
    .rdAddr_wb(rdAddr_wb), .rdAddr_mem(rdAddr_mem),
    .rs1Addr_ex(rs1Addr_ex), .rs2Addr_ex(rs2Addr_ex),
    .ForwardA(ForwardA), .ForwardB(ForwardB)
);

assign MemWriteData_ex= B_0;

endmodule
```

### 3、数据存储器模块（DataRAM）的设计

数据存储器可用 Xilinx 的 IP 内核实现。考虑到 FPGA 的资源，数据存储器可设计为容量为 64x32bit 的单端口 RAM，输出采用组合输出(Non Registered)。

由于数据存储器容量为 64x32bit，故存储器地址共 6 位，与 ALUResult_mem[7:2]连接。

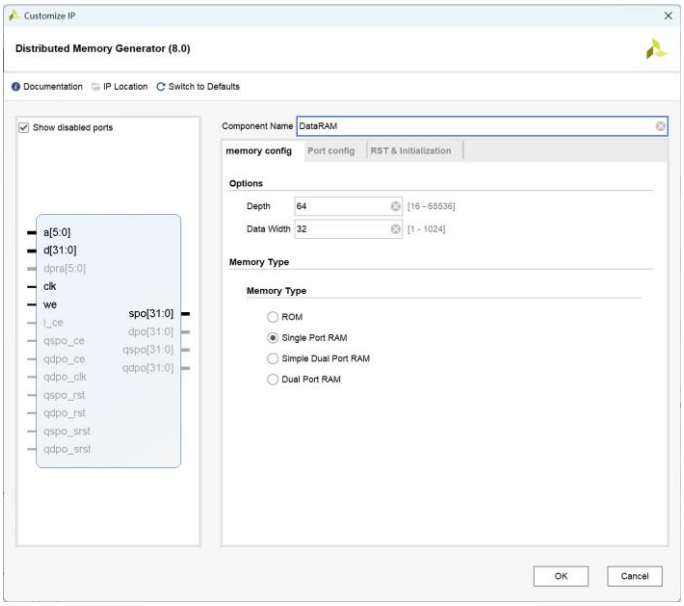
步骤如下：

（1） 打开 PROJECT MANAGER 栏中的 IP Catalog，在 RAMs & ROMs 中选择 Distributed Memory Generator，进入配置界面。

（2） 在第一列 Memory config 中进行如下配置：

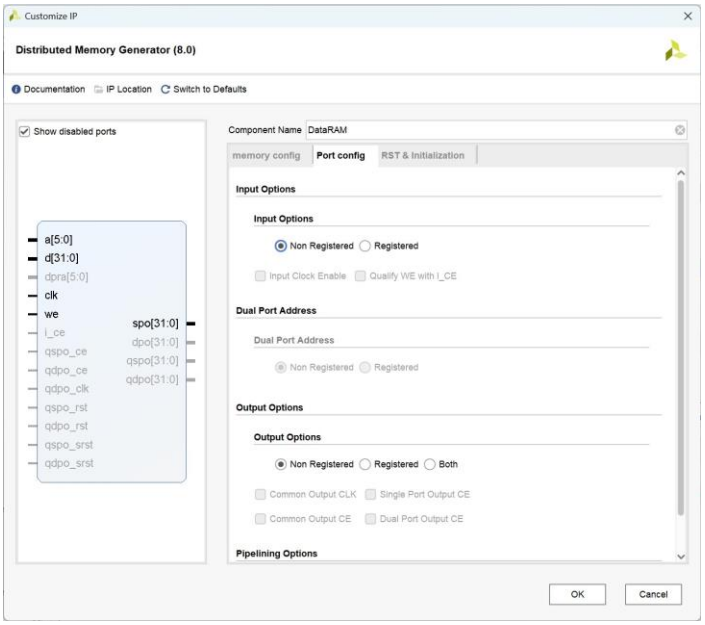
Options 中深度（Depth）设置为 64，Data Width 为 32 位。

Memory Type 中选择 Single Port RAM。




（3） 在第二列 Port Config 中进行如下配置：

Input Options 和 Output Options 中选择 Non Registered。




## 4、取指令级模块（IF）的设计

IF 模块由指令指针寄存器(PC)、指令存储器子模块(Instruction ROM)、指令指针选择器(MUX)和一个 32 位加法器组成，IF 模块接口信息如表 30.10 所示。

表 30.10　IF 模块的输入/输出引脚说明

| 引脚名称 | 方向 | 说明 |
|---|---|---|
| clk | Input | 系统时钟 |
| reset | | 系统复位信号，高电平有效 |
| Branch | | 条件分支指令的条件判断结果 |
| Jump | | 无条件分支指令的条件判断结果 |
| IFWrite | | 流水线阻塞信号 |
| JumpAddr[31:0] | | 分支地址 |
| Instruction [31:0] | Output | 指令机器码 |
| IF_flush | | 流水线清空信号 |
| PC [31:0] | | PC 值 |

由图 30.3 可看出，指令存储器为组合存储器，可用 Verilog HDL 设计一个查找表阵列 ROM。考虑到 FPGA 的资源，该 ROM 容量可设计为 64×32bit。实验提供了一个指令存储器模块 InstructionROM.v，ROM 内存放一段简单测试程序的机器码，对应的测试程序为：

```
        lui X30, 0x3000
        jalr X31, later(X0)
earlier:sw X28, 0x0C(X0)
        lw X29, 4(X6)
        slli X5, X29, 2 //数据冒险
        lw X28, 4(X6)
        sltu X28, X6,X7
done:   jal X31, done
later:  bne X0, X0, end // 分支条件不成立
        addi X5, X30, 0x42
        add X6, X0, X31
        sub X7, X5, X6 //操作 A 二阶数据相关，操作 B 一阶数据相关
        or X28, X7, X5 //操作 A 一阶数据相关，操作 B 三阶数据相关
        beq X0, X0, earlier // 分支条件成立
end:    nop
```

结合数据通路知识并查看相关原理图可以得知，当指令不发生跳转，即无 Branch 或 Jump 信号，IF 模块直接从当前 PC 所指向的 Instruction ROM 内存地址取指令，完成取指令操作后 PC+4；如果发生跳转，则需要先修改 PC 值为 PC＝JumpAddr，再从 PC 所指向的 Instruction ROM 内存地址取指令，王城取指令操作后，PC+4。

此外发生跳转时还需要清空当前流水线，确保之前按顺序取得的指令无效。

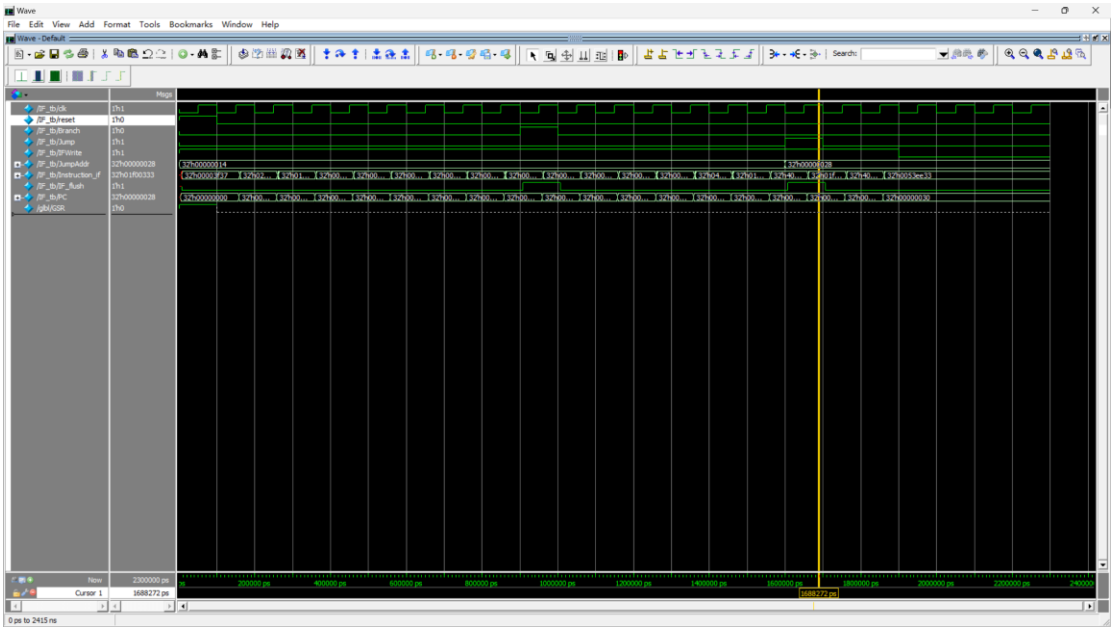
还需注意如果发生阻塞（IFWrite=0），PC 的值应该保持。

```
module IF (
    input clk, reset, Branch, Jump, IFWrite,
    input [31:0] JumpAddr,
    output [31:0] Instruction_if,
    output [31:0] PC,
```

```
    output IF_flush
);
wire [31:0] CurrentPC, NextPC, NextPC_if;
InstructionROM IROM(
    .addr(NextPC[7:2]),
    .dout(Instruction_if)
);

PCRegister PCR(
    .clk(clk), .rst(reset), .PC(CurrentPC),
    .NextPC(NextPC), .En(IFWrite)
);
assign NextPC_if= NextPC+32'd4;
assign PC= NextPC;
assign IF_flush= Branch || Jump;
assign CurrentPC= (IF_flush == 1'b0)?NextPC_if:JumpAddr;
endmodule
```

IF 级模块的仿真结果如下：



IF 仿真结果的分析见报告“实验结果及分析”部分。

## 5、流水线寄存器的设计

流水线寄存器负责将流水线的各部分分开，共有 IF/ID、ID/EX、EX/MEM、MEM/WB 四组，对四组流水线寄存器要求不完全相同，因此设计也有不同考虑。

EX/MEM、MEM/WB 两组流水线寄存器只是普通的 D 型寄存器：

```
// EX/MEM Register
always @(posedge clk) begin
    MemtoReg_mem<=MemtoReg_ex; RegWrite_mem<=RegWrite_ex;
    MemWrite_mem<=MemWrite_ex;
    ALUResult_mem<=ALUResult_ex; MemWriteData_mem<=MemWriteData_ex;
    rdAddr_mem<=rdAddr_ex;
end
```

```
// MEM/WB Register
always @(posedge clk) begin
    MemtoReg_wb<=MemtoReg_mem; RegWrite_wb<=RegWrite_mem;
    MemDout_wb<=MemDout_mem; ALUResult_wb<=ALUResult_mem;
    rdAddr_wb<=rdAddr_mem;
end
```

当流水线发生数据冒险时，需要清空 ID/EX 流水线寄存器而插入一个气泡；若跳转指令或分支成立，则还需要清空 ID/EX 流水线寄存器。因此 ID/EX 流水线寄存器是一个带同步清零功能的 D 型寄存器：

```
// ID/EX Register
always @(posedge clk) begin
    if((Stall|reset)==1'b1) begin
        MemtoReg_ex<=0; RegWrite_ex<=0; MemWrite_ex<=0;
        MemRead_ex<=0; ALUSrcA_ex<=0;
        ALUCode_ex<=4'b0;  ALUSrcB_ex<=2'b0;
        rdAddr_ex<=5'b0;
        rs1Addr_ex<=5'b0; rs2Addr_ex<=5'b0;
        Imm_ex<=32'b0; PC_ex<=32'b0;
        rs1Data_ex<=32'b0; rs2Data_ex<=32'b0;
    end
    else begin
        MemtoReg_ex<=MemtoReg_id; RegWrite_ex<=RegWrite_id;
        MemWrite_ex<=MemWrite_id; MemRead_ex<=MemRead_id;
        ALUSrcA_ex<=ALUSrcA_id; ALUSrcB_ex<=ALUSrcB_id;
        ALUCode_ex<=ALUCode_id;
        rdAddr_ex<=rdAddr_id;
        rs1Addr_ex<=rs1Addr_id; rs2Addr_ex<=rs2Addr_id;
        rs1Data_ex<=rs1Data_id; rs2Data_ex<=rs2Data_id;
        Imm_ex<=Imm_id;  PC_ex<=PC_id;
    end
end
```

当流水线发生数据冒险时，需要阻塞 IF/ID 流水线寄存器。因此，IF/ID 流水线寄存器除

同步清零功能外，还需要具有保持功能（即具有使能 EN 信号输入）：

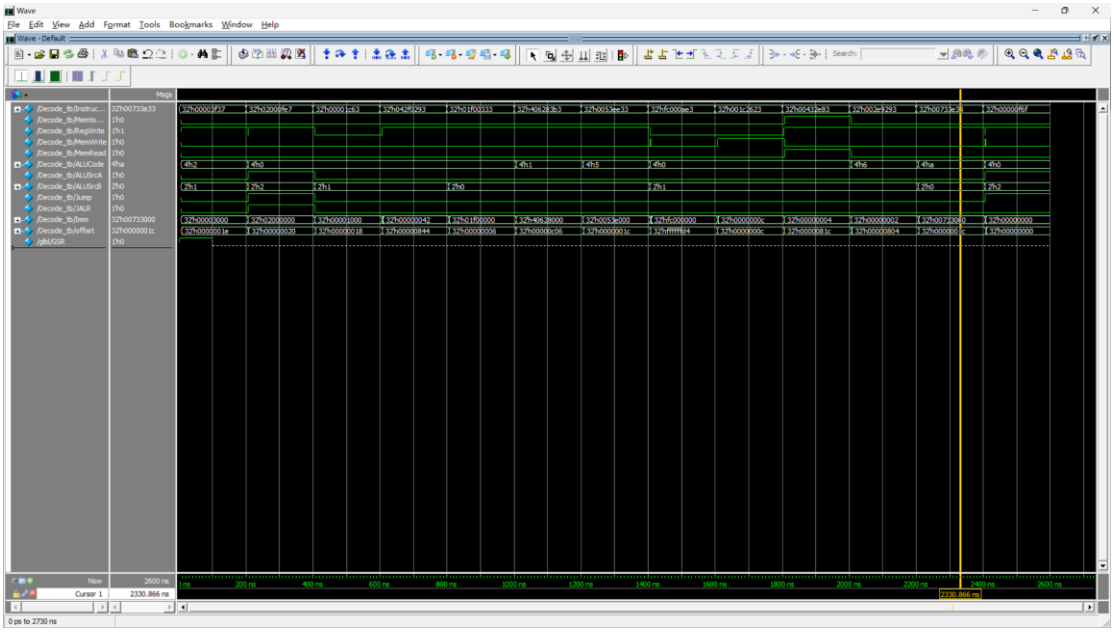
```
// IF/ID Register
always @(posedge clk) begin
    if((IF_flush|reset)==1'b1)begin
        PC_id<= 32'b0;
        Instruction_id<= 32'b0;
    end
    else if(IFWrite==1'b1) begin
        PC_id<= PC;
        Instruction_id<= Instruction_if;
    end
    else begin
        PC_id<= PC_id;
        Instruction_id<= Instruction_id;
    end
end
```

## 6、顶层文件的设计

按照图 30.3 所示的原理框图连接各模块即可。为了测试方便，可将关键变量输出，关键变量有:指令指针 PC、指令码 Instruction_ id. 流水线插入气泡标志 Stall、分支标志 JumpFlag 即{Jump, Branch}、ALU 输入输出(ALU_A、ALU_B、ALUResult_ex)和数据存储器的输出 MemDout_mem。

# 四、实验结果及分析

## 1、Decode 模块仿真结果及分析

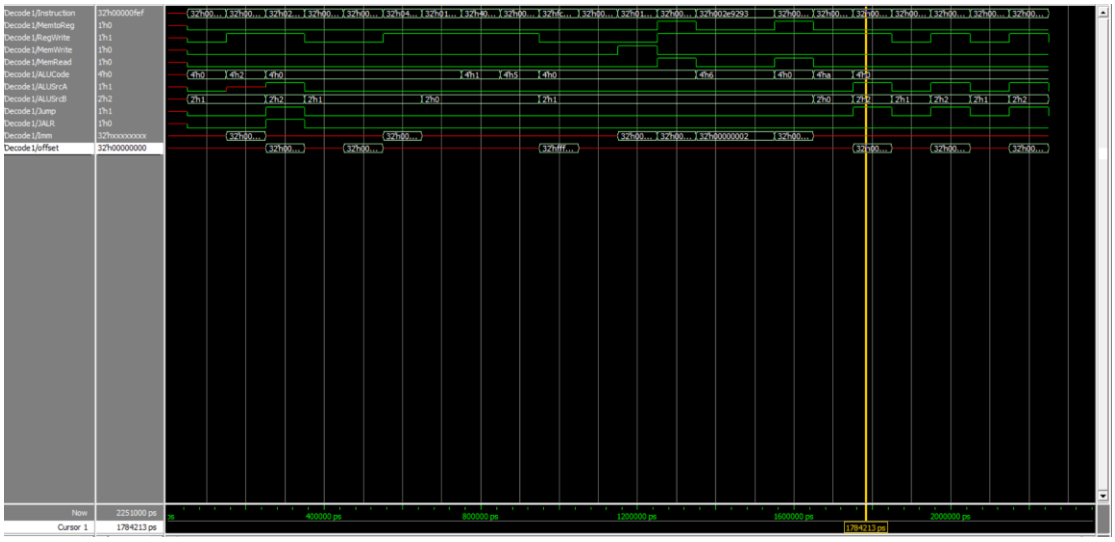


指令相应输出可以参考下表：

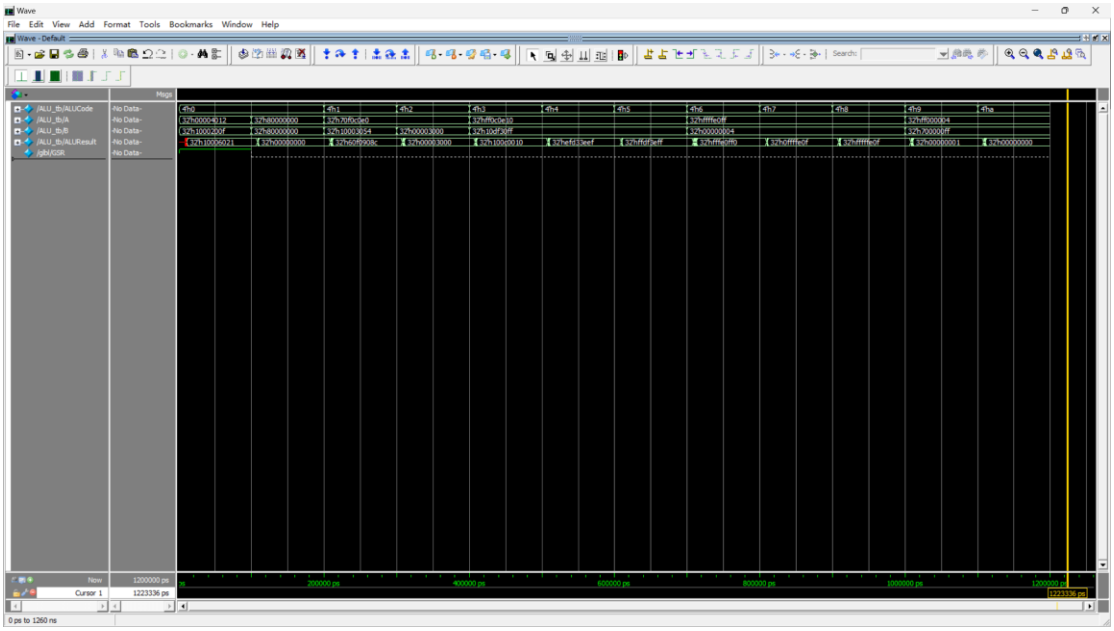
| Instruction | MemRead | MemtoReg | MemWrite | RegWrite | Jump | JALR | ALUSrcA | ALUSrcB | ALUCode | Imm | offset |
|---|---|---|---|---|---|---|---|---|---|---|---|
| lui X30, 0x3000 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2‘h1 | 4‘h2 | 32‘h00003000 | - |
| jalr X31, later(X0) | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2’h2 | 4’h0 | - | 32’h00000020 |
| bne X0, X0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2’h1 | 4’h0 | - | 32’h00000018 |
| addi X5, X30, 42 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2’h1 | 4’h0 | 32’h00000042 | - |
| add X6, X0, X31 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2’h0 | 4’h0 | - | - |
| sub X7, X5, X6 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2’h0 | 4’h1 | - | - |
| or X28, X7, X5 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2’h0 | 4’h5 | - | - |
| beq X0, X0, earlier | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2’h1 | 4’h0 | - | 32’hfffffffd4 |
| sw X28, 0C(X0) | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 2’h1 | 4’h0 | 32’h0000000c | - |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| lw X29, 04(X6) | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 2'h1 | 4'h0 | 32'h00000004 | - |
| sll X5, X29 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2'h1 | 4'h6 | 32'h00000002 | - |
| sltu X28, X6,X7 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 2'h0 | 4'ha | - | - |
| jal X31, done | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 2'h1 | 4'h2 | - | 32'h00000000 |

可以发现，MemRead、MemtoReg、MemWrite、RegWrite、Jump、JALR、ALUSrcA、ALUSrcB、ALUCode 信号始终正确；而 Imm、offset 在需要输出时是正确的。在不需要输出的时候，代码设定是 32'bx，但由于 vivado 的 Implementation 设置并不允许 x 的出现，因而呈现了随机数，但是并不影响系统的工作。下图为流水线 CPU 仿真中 Decode 的运行情况，可见 Imm 和 offset 在不需要输出时确实是 32'bx。



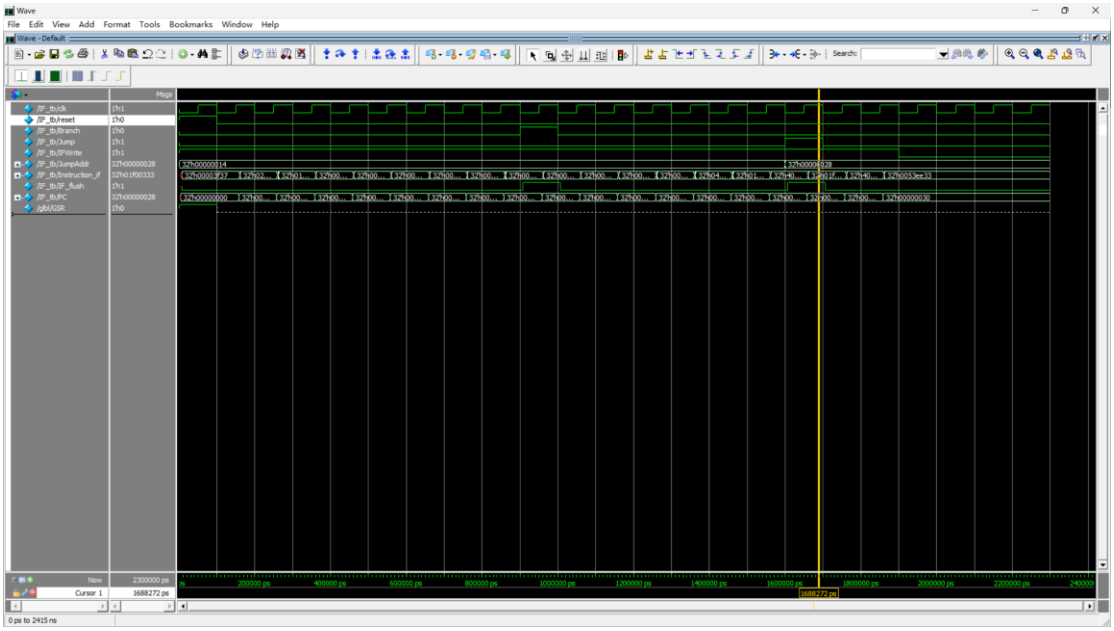
## 2、ALU 模块仿真结果及分析



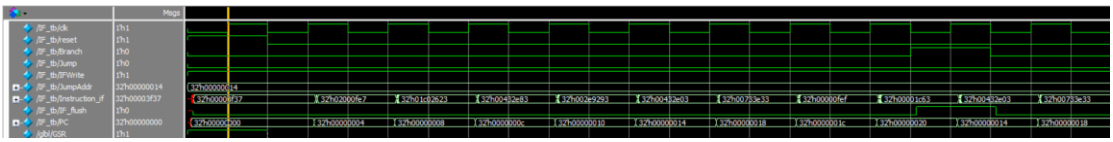
参考下表可知，ALU 功能正确。

| A | ALUCode | B | ALUResult |
|---|---|---|---|
| 32’h00004012 | 0(+) | 32’h1000200f | 32’h10006021 |
| 32’h80000000 | 0(+) | 32’h80000000 | 32’h00000000 |
| 32’h70f0c0e0 | 1(-) | 32’h10003054 | 32’h60f0908c |
| 32’h70f0c0e0 | 2(B) | 32’h00003000 | 32’h00003000 |
| 32’hff0c0e10 | 3(&) | 32’h10df30ff | 32’h100c0010 |
| 32’hff0c0e10 | 4(^) | 32’h10df30ff | 32’hefd33eef |
| 32’hff0c0e10 | 5(\|) | 32’h10df30ff | 32’hffdf3eff |
| 32’hffffe0ff | 6(<<) | 32’h00000004 | 32’hfffe0ff0 |
| 32’hffffe0ff | 7(>>) | 32’h00000004 | 32’h0ffffe0f |
| 32’hffffe0ff | 8(>>>)(算数右移) | 32’h00000004 | 32’hfffffe0f<br>(保持符号位不变) |
| 32’hff00000004 | 9(A<B?1:0)( 有 符 号 数) | 32’h700000ff | 32’h00000001<br>(此时 A 为负数, B 为正数) |
| 32’hff00000004 | 10(A<B?1:0) ( 无 符 号数) | 32’h700000ff | 32’h00000000 |

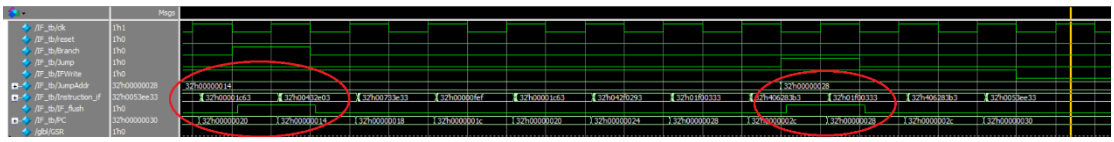
## 3、IF 模块仿真结果及分析



将波形展开，可见 PC 在每个时钟周期正常+4：



当 IF_flush=1 时，PC 在下一个时钟周期跳转为 JumpAddr 的值：



因此 IF 模块仿真结果正确。

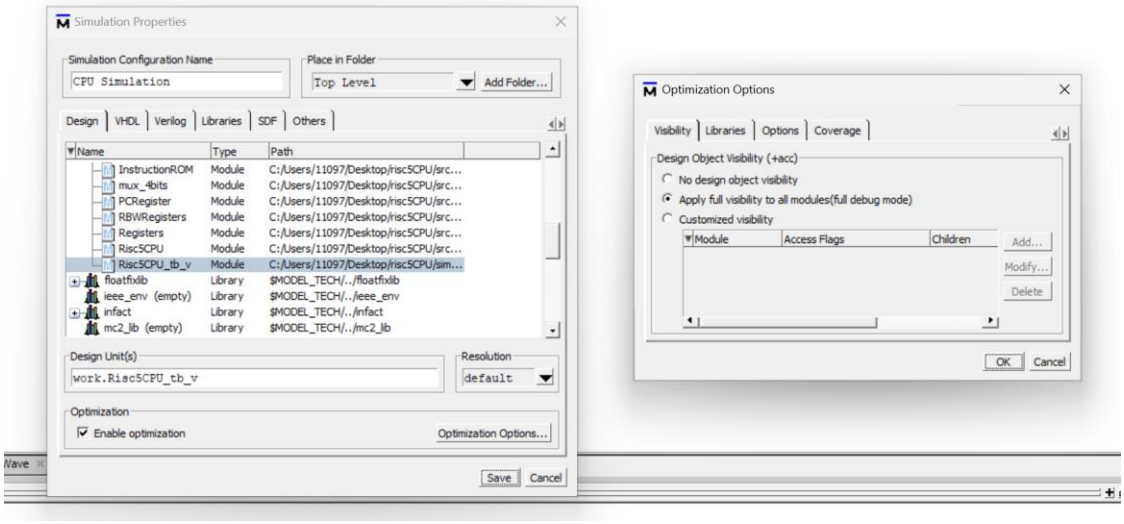
## 4、Risc-V CPU 仿真结果及分析

（1） CPU 仿真配置

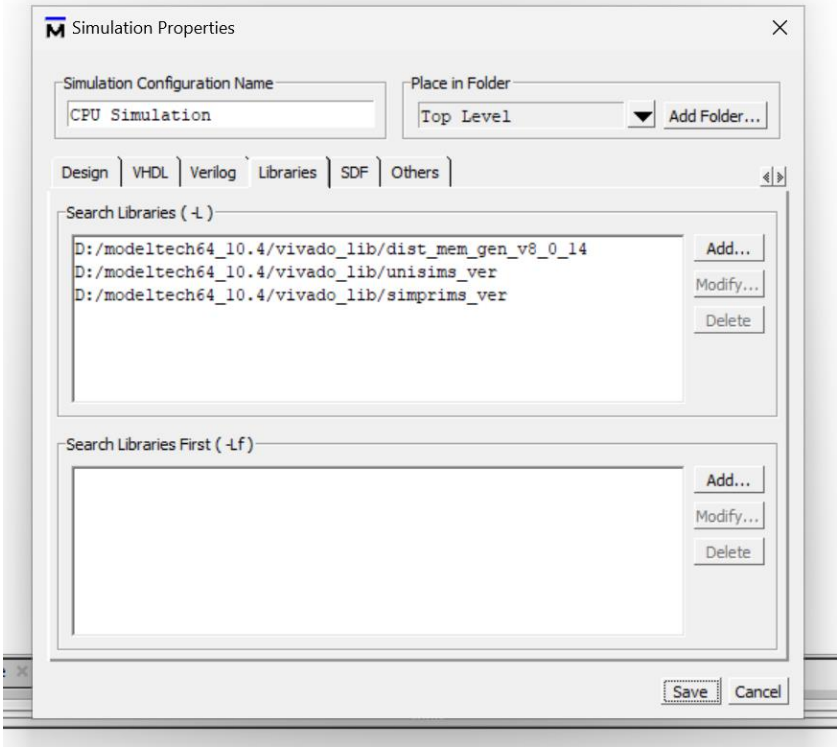
Simulation Configuration：

将 Risc5CPU_tb.v 设为 Design Unit。

Optimization Options 中的 Design Object Visibility 选中 Apply full visibility to all modules（full debug mode）。

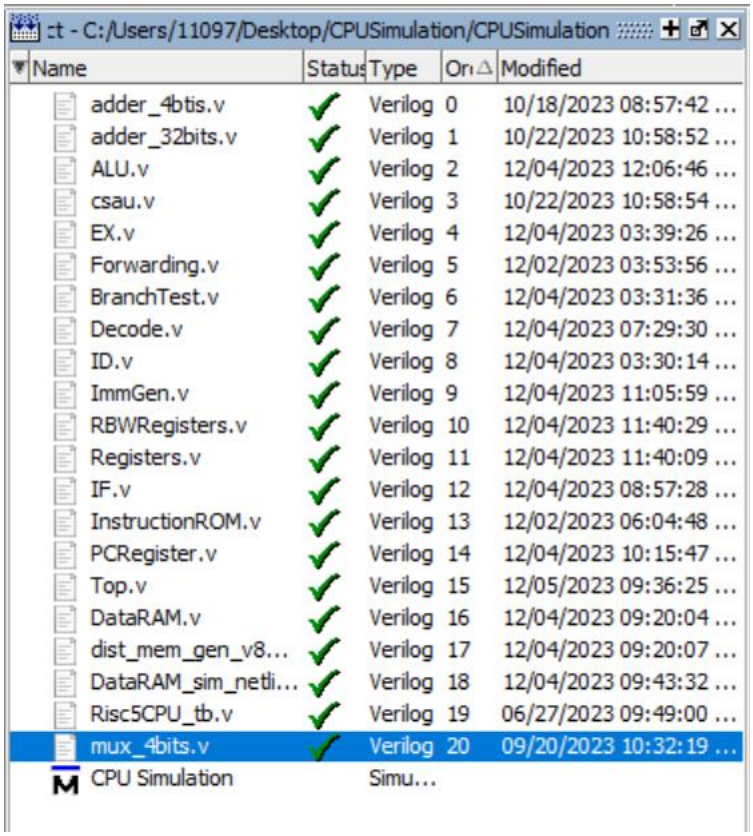



在 Libraries 中添加如下库：

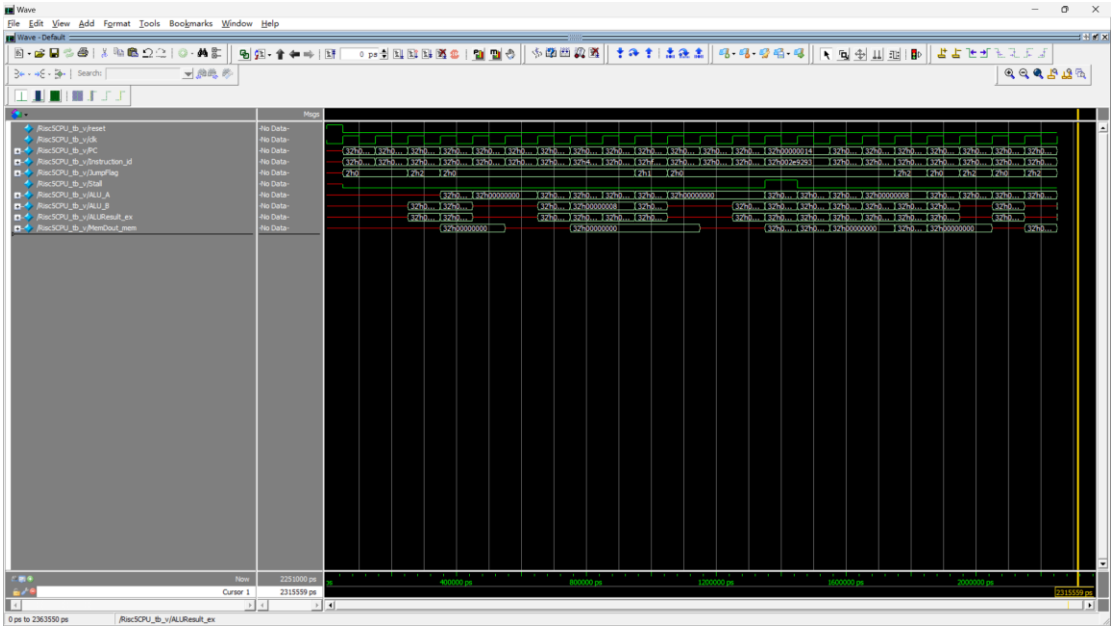


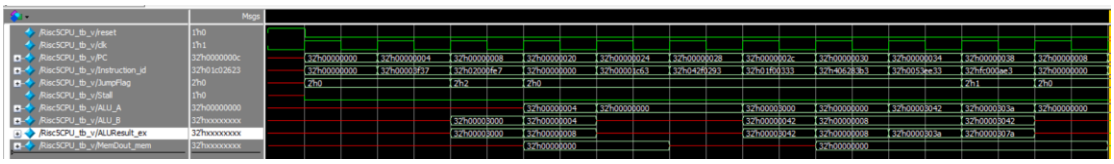

需要注意的是，为了仿真成功，还需要在 Vivado 工程中找到 DataRAM.v，dist_mem_gen_v8_0.v 和 DataRAM_sim_netlist.v，并添加到 ModelSim 工程中，否则会报错 glbl 模块找不到。

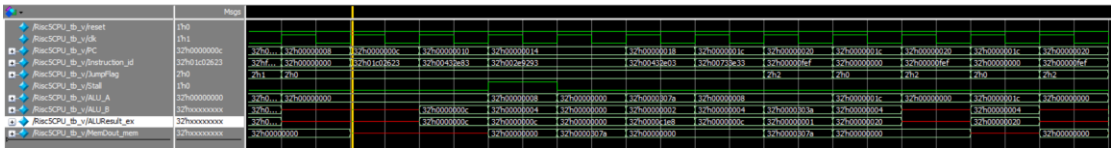
Project 中的文件如下：


:t - C:/Users/11097/Desktop/CPUSimulation/CPUSimulation
Name
Status
Type
Or...
Modified
adder_4btis.v Verilog 0 10/18/2023 08:57:42 ...
adder_32bits.v Verilog 1 10/22/2023 10:58:52 ...
ALU.v Verilog 2 12/04/2023 12:06:46 ...
csau.v Verilog 3 10/22/2023 10:58:54 ...
EX.v Verilog 4 12/04/2023 03:39:26 ...
Forwarding.v Verilog 5 12/02/2023 03:53:56 ...
BranchTest.v Verilog 6 12/04/2023 03:31:36 ...
Decode.v Verilog 7 12/04/2023 07:29:30 ...
ID.v Verilog 8 12/04/2023 03:30:14 ...
ImmGen.v Verilog 9 12/04/2023 11:05:59 ...
RBWRegisters.v Verilog 10 12/04/2023 11:40:29 ...
Registers.v Verilog 11 12/04/2023 11:40:09 ...
IF.v Verilog 12 12/04/2023 08:57:28 ...
InstructionROM.v Verilog 13 12/02/2023 06:04:48 ...
PCRegister.v Verilog 14 12/04/2023 10:15:47 ...
Top.v Verilog 15 12/05/2023 09:36:25 ...
DataRAM.v Verilog 16 12/04/2023 09:20:04 ...
dist_mem_gen_v8... Verilog 17 12/04/2023 09:20:07 ...
DataRAM_sim_netli... Verilog 18 12/04/2023 09:43:32 ...
Risc5CPU_tb.v Verilog 19 06/27/2023 09:49:00 ...
mux_4bits.v Verilog 20 09/20/2023 10:32:19 ...
CPU Simulation Simu...

（2） CPU 仿真结果







参照下表，各项数值正确。

表 30.11　测试程序的运行结果

| reset | clk | PC | Instruction（ID） | JumpFlag | Stall | ALU_A | ALU_B | ALUResult | MemDout（MEM） |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | - |
| 0 | 2 | 4 | 00003f37 (lui) | 0 | 0 | - | - | - | - |
| | 3 | 8 | 02000fe7 (jalr) | 2 | 0 | - | 3000 | 3000 | - |
| | 4 | 20 | 0 | 0 | 0 | 4 | 4 | 8 | - |
| | 5 | 24 | 00001c63 (bne) | 0 | 0 | -- | - | - | - |
| | 6 | 28 | 042f0293 (addi) | 0 | 0 | - | - | - | - |
| | 7 | 2c | 01f00333 (add) | 0 | 0 | 3000 | 42 | 3042 | - |
| | 8 | 30 | 406283b3 (sub) | 0 | 0 | 0 | 8 | 8 | - |
| | 9 | 34 | 0053ee33 (or) | 0 | 0 | 3042 | 8 | 303a | - |
| | 10 | 38 | fc000ae3 (beq) | 1 | 0 | 303a | 3042 | 307a | - |
| | 11 | 8 | 0 | 0 | 0 | - | - | - | - |
| | 12 | c | 01c02623 (sw) | 0 | 0 | - | - | - | - |
| | 13 | 10 | 00432e83 (lw) | 0 | 0 | 0 | c | c | - |
| | 14 | 14 | 002e9293 (sll) | 0 | 1 | 8 | 4 | c | - |
| | 15 | | | 0 | 0 | - | - | - | 307a |
| | 16 | 18 | 00432e03 (lw) | 0 | 0 | 307a | 2 | c1e8 | - |
| | 17 | 1c | 00733e33 (sltu) | 0 | 0 | 8 | 4 | c | - |
| | 18 | 20 | 0000fef (jal) | 2 | 0 | 8 | 303a | 1 | 307a |
| | 19 | 1c | 0 | 0 | 0 | 1c | 4 | 20 | - |
| | 20 | 20 | 0000fef (jal) | 2 | 0 | - | - | - | - |
| | 21 | 1c | 0 | 0 | 0 | 1c | 4 | 20 | - |

# 五、个人心得

（1）　要注意=赋值和<=赋值的区别：

这两种赋值“=”用于阻塞式赋值；“<=”用于非阻塞式赋值中。阻塞赋值：阻塞赋值语句是在这句之后所有语句执行之前执行的，即后边的语句必须在这句执行完毕才能执行，所以称为阻塞，实际上就是顺序执行。非阻塞赋值：非阻塞赋值就是与后边相关语

句同时执行，即就是并行执行。所以一般时序电路使用非阻塞赋值，assign 语句一般使用=阻塞赋值；组合逻辑电路使用阻塞赋值。

而我在写寄存器的时候用了阻塞赋值，导致输入输出信号同周期变化，给我的 debug 造成了很大的困扰。

（2） 要注意变量名的一致性：

实验中，ID 模块输出的 BranchAddr 在别的模块叫做 JumpAddr，这给我带来一个一定的困惑。

（3） 要注意信号的位宽：

实验中涉及的信号很多，有可能一不小心就在子模块的输入或输出中忘记定义信号的位宽了，导致多比特信号变为一比特信号。当然调试中也很容易能够发现，也可以在 Vivado 的 synthesis messages 中发现。

（4） 利用好 Vivado 的 messages：

实验中有很多没有顾及到的小问题都可能导致最后仿真的差错。Verilog 的代码很多问题不会直接报 error，而是通过 warning 呈现，比如重定义、信号未利用、信号位宽有误等，所以我们应该养成不忽略每一个 warning 的习惯，细心排查。

（5） Modelsim Debug 技巧：

在流水线 CPU 的仿真 debug 中，应该对应指令按照模块一个时钟一个时钟地 debug，而不是看某一个信号的时序变化。

## 六、思考题

如下面两条指令，条件分支指令试图读取上一条指令的目标寄存器，插入气泡或数据转发都无法解决流水线冲突问题。为什么在大多 CPU 架构中，都不去解决这一问题？这一问题应在什么层面中解决？

```
lw X28, 04(X6)
beq X28,x29,Loop
```

执行两个空操作能解决这个问题，因为在该情况下，写回阶段在时钟节拍的前半部分进行，而寄存器文件读取在后半部分进行，因此它们不会发生冲突。这样，条件分支指令读取的是已经由 lw 指令更新过的 X28 寄存器的值。

然而，在某些高频设计中，不支持在一个时钟节拍中进行先写后读的操作。在这种情况下，执行三个空操作仍然可以解决这个问题。