专业：电子科学与技术
姓名：
学号：
日期：
地点：玉泉

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：软件实验　同组学生姓名：无

装 订 线

一、实验目的

熟悉 RISC-V 指令系统；了解提高 CPU 性能的方法；掌握流水线 RISC-V 微处理器的工作原理；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；掌握流水线 RISC-V 微处理器的测试方法；了解用软件实现数字系统的方法。

二、实验任务与要求

设计一个流水线 RISC-V 微处理器，具体要求如下所述。

（1）至少运行下列 RV32I 核心指令。

① 算术运算指令：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、bgeu
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。

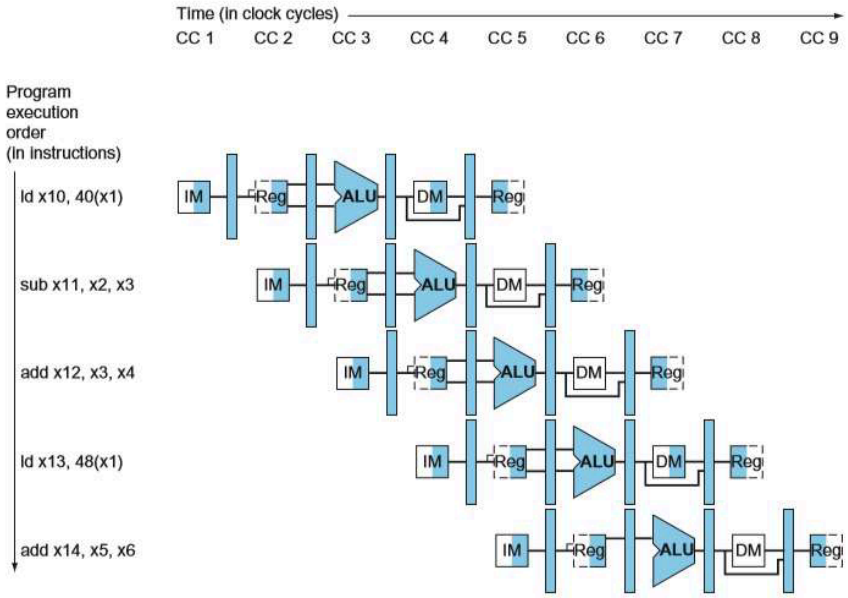


图 1 流水线流水作业示意图

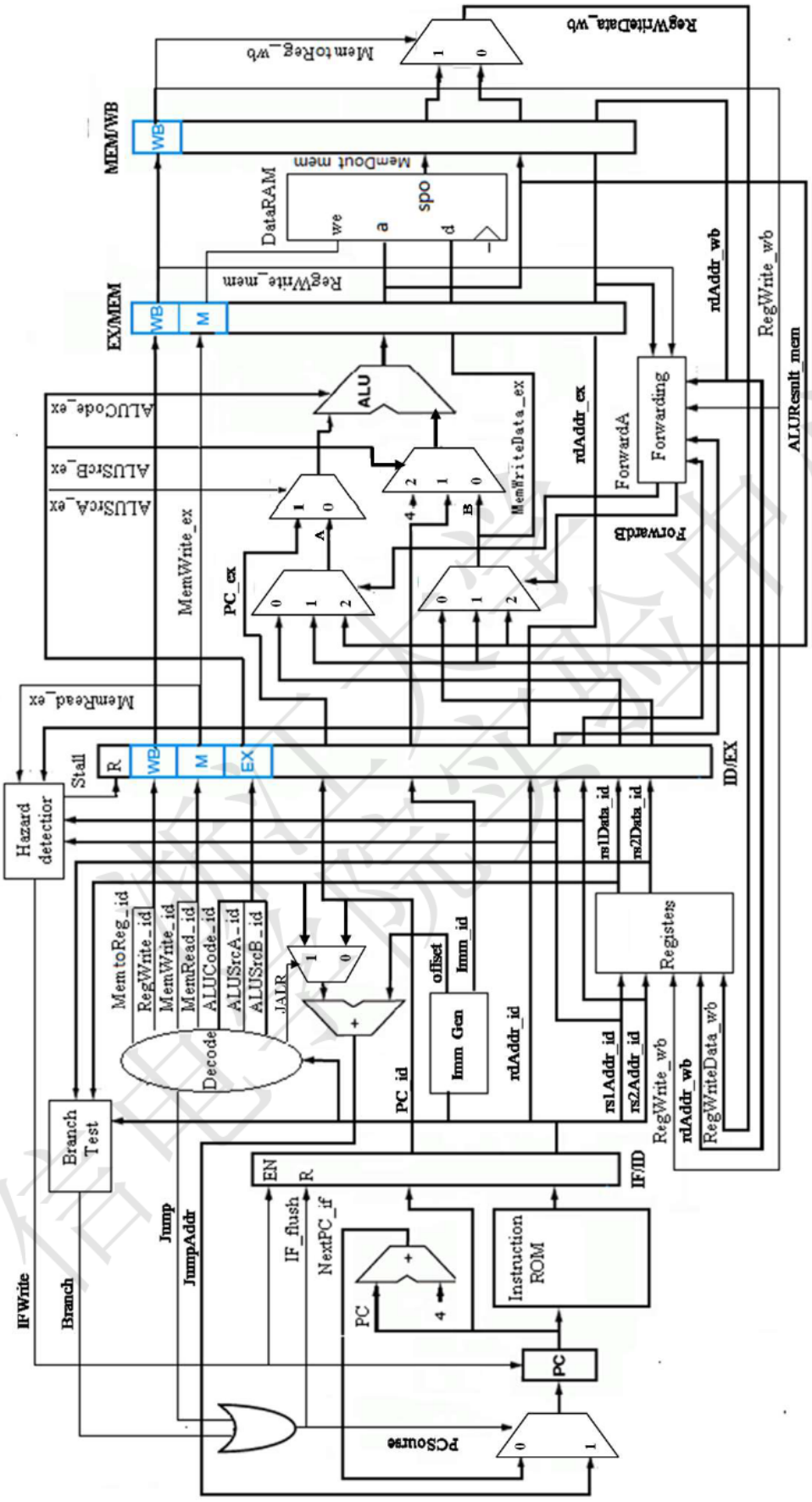


图 2 RISC-V 微处理机流水线原理图

三、实验原理与步骤

（一）总体设计

流水线是数字系统中一种提高系统稳定性和工作速度的方法。根据 RISC-V 处理器指令的特点，将指令整体的处理过程分为取指令（IF）、指令译码（ID）、执行（EX）、存储器访问（MEM）和寄存器回写（WB）五级，如图 1 所示。

1、流水线中的控制信号

（1）IF 级：从 ROM 中读取指令，并在下一个时钟沿到来时把指令送到 ID 级的指令缓冲器中。有 3 个控制信号：

- PCSource：决定下一条指令指针的控制信号，当 PCSource=0 时，顺序执行下一条指令；当 PCSource=1 时，跳转执行。
- IFWrite：IFWrite=0 时阻塞 IF/ID 流水线，同时暂停读取下一条指令。
- IF_flush：IF_flush=1 时清空 IF/ID 寄存器。

（2）ID 级：指令译码级。对来自 IF 级的指令进行译码，并产生相应的控制信号。整个 CPU 的控制信号基本都是在这级上产生。该级自身不需任何控制信号。流水线冒险检测也在该级进行，即当流水线冒险条件成立时，冒险检测电路产生 Stall 信号清空 ID/EX 寄存器，同时冒险检测电路产生低电平 IFWrite 信号阻塞 IF/ID 流水线。即插入一个流水线气泡。

（3）EX 级：执行级。此级进行算术或逻辑操作。此外数据传送指令所用的 RAM 访问地址也是在本级上实现。控制信号有 ALUCode、ALUSrcA 和 ALUSrcB，根据这些信号确定 ALU 操作、并选择两个 ALU 操作数 ALU_A、ALU_B。另外，数据转发也在该级完成。数据转发控制电路产生 ForwardA 和 ForwardB 两组转发控制信号。

（4）MEM 级：存储器访问级。只有在执行数据传送指令时才对存储器进行读写，对其它指令只起到缓冲一个时钟周期的作用。该级只需存储器写操作允许信号 MemWrite。

（5）WB 级：回写级。此级把指令执行的结果回写到寄存器堆中。该级设置信号 MemtoReg 和寄存器写操作允许信号 RegWrite。其中 MemtoReg 决定写入寄存器的数据来源：当 MemtoReg=0 时，回写数据来自 ALU 运算结果；而当 MemtoReg=1 时，回写数据来自存储器。

2、数据相关与数据转发

如果上一条指令的结果还没有写入到寄存器中，而下一条指令的源操作数又恰恰是此寄存器的数据，那么它所获得的将是原来的数据，而不是更新后的数据。这样的相关问题称为数据相关。在设计中，采用数据转发和插入流水线气泡的方法解决数据相关问题。具体而言，数据相关问题可分为以下三类：

（1）一阶数据相关与转发（EX 冒险）

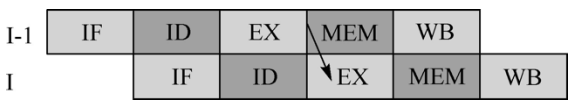


图 3 一阶前推网络示意图

如果源操作寄存器与第 I-1 条指令的目标操作寄存器相重，将导致一阶数据相关。从图 3 可以看出，第 I 条指令的 EX 级与第 I-1 条指令的 MEM 级处于同一时钟周期，且数据转发必须在第 I 条指令的 EX 级完成。因此，导致操作数 A 的一阶数据相关判断的条件为：

①MEM 级阶段必须是写操作（RegWrite_mem=1）

②目标寄存器不是 X0 寄存器（rdAddr_mem!=0）

③两条指令读写同一个寄存器（rdAddr_mem=rs1Addr_ex）。

导致操作数 B 的一阶数据相关成立的条件为：

①MEM 级阶段必须是写操作（RegWrite_mem=1）

②目标寄存器不是 X0 寄存器（rdAddr_mem!=0）。

③两条指令读写同一个寄存器（rdAddr_mem=rs2Addr_ex）。

除了第 I-1 条指令为 lw 外，其它指令回写寄存器的数据均为 ALU 输出，因此当发生一阶数据相关时，除 lw 指令外，一阶数据相关的解决方法是将第 I-1 条指令的 MEM 级的 ALUResult_mem 转发至第 I 条 EX。lw 指令数据相关的处理方法在后续内容中说明。

（2）二阶数据相关与转发（MEM 冒险）

图 4 二阶前推网络示意图

如图 4 所示，如果第 I 条指令的源操作寄存器与第 I-2 条指令的目标寄存器相重，将导致二阶数据相关。导致操作数 A 的二阶数据相关必须满足下列条件：

①WB 级阶段必须是写操作（RegWrite_wb=1）

②目标寄存器不是 X0 寄存器（rdAddr_wb!=0）

③一阶数据相关条件不成立（rdAddr_mem!=rs1Addr_ex）

④两条指令读写同一个寄存器（rdAddr_wb=rs1Addr_ex）。

导致操作数 B 的二阶数据相关必须满足下列条件：

①WB 级阶段必须是写操作（RegWrite_wb=1）

②目标寄存器不是 X0 寄存器（rdAddr_wb!=0）

③一阶数据相关条件不成立（rdAddr_mem!=rs2Addr_ex）

④两条指令读写同一个寄存器（rdAddr_wb=rs2Addr_ex）。

当发生二阶数据相关问题时，解决方法是将第 I-2 条指令的回写数据 RegWriteData_wb 转发至 I 条指令的 EX。

（3）三阶数据相关

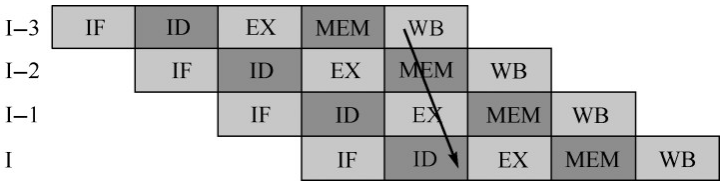


图 5 三阶前推网络示意图

图 5 所示为第 I 条指令与第 I-3 条指令的数据相关问题，即在同一个周期内同时读写同一个寄存器，将导致三阶数据相关。导致操作数 A 的三阶数据相关必须满足下列条件：

①寄存器必须是写操作（RegWrite_wb=1）

②目标寄存器不是 X0 寄存器（rdAddr_wb!=0）

③读写同一个寄存器（rdAddr_wb =rs1Addr_id）

导致操作数 B 的三阶数据相关必须满足下列条件；

①寄存器必须是写操作（RegWrite_wb=1）

②目标寄存器不是 X0 寄存器（rdAddr_wb!=0）

③读写同一个寄存器（rdAddr_wb=rs2Addr_id）。

该类数据相关问题可以通过改进设计寄存器堆的硬件电路来解决，要求寄存器堆具有 Read After Write 特性。

3、数据冒险与数据转发

当第 I 条指令读取一个寄存器，而第 I-1 条指令为 lw，且与 lw 写入为同一个寄存器时，定向转发是无法解决问题的。因此，当 lw 指令后跟一条需要读取它结果的指令时，必须采用相应的机制来阻塞流水线，即还需要增加一个冒险检测单元（Hazard Detector）。它工作在 ID 级，当检测到上述情况时，在 lw 指令和后一条指令之间插入气泡，使后一条指令延迟一个周期执行，这样可将一阶数据冒险问题变成二阶数据冒险问题，就可用转发解决。

冒险检测工作在 ID 级，前一条指令已处在 EX 级，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）。

当上述条件满足时，指令将被阻塞一个周期，Hazard Detector 电路输出的电路输出的 Stall 信号清空信号清空 ID/EX 寄存器，另外一个输出低电平有效的存器，另外一个输出低电平有效的 IFWrite 信号阻塞流水线信号阻塞流水线 ID 级、IF 级，即插入一个流水线气泡

（二）流水线 RISC-V 微处理器的设计

1、指令译码模块（ID）的设计

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。接口信息如表 1 所示。

表 1 ID 模块接口信息

方向	引脚名称	说明
Input	clk	系统时钟
	Instruction_id[31:0]	指令机器码
	PC_id[31:0]	指令指针
	RegWrite_wb	WB 级寄存器写允许信号，高电平有效
	rdAddr_wb[4:0]	WB 级寄存器写地址
	RegWriteData_wb[31:0]	WB 级寄存器写入数据
	MemRead_ex	冒险检测输入
	rdAddr_ex[4:0]	
Output	MemtoReg_id	决定回写数据来源（0:ALU；1:存储器）
	RegWrite_id	ID 译码结果，寄存器写允许信号，高电平有效
	MemWrite_id	ID 译码结果，存储器写允许信号，高电平有效
	MemRead_id	ID 译码结果，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 操作数来源（0:rs1；1:PC）
	ALUSrcB_id[1:0]	决定 ALU 的 B 操作数来源（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

Output	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]	ID 译码结果，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]	

（1）寄存器堆（Registers）子模块的设计

寄存器堆由 32 个 32 位寄存器组成，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 6 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。X0 寄存器为常数 0。

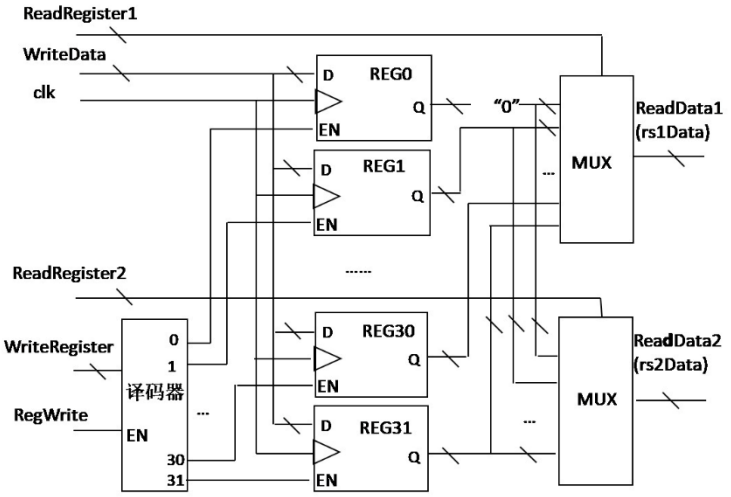


图 6 寄存器堆的原理框图

```
module RBWRegisters(clk, ReadRegister1, ReadRegister2, WriteRegister, RegWrite, WriteData, ReadData1, ReadData2);
    input clk;
    input [4:0] ReadRegister1, ReadRegister2, WriteRegister; //读/写寄存器位置
    input [31:0] WriteData;
    input RegWrite;
    output [31:0] ReadData1, ReadData2;
    reg [31:0]regs [31:0]; //定义 32*32 存储器变量
    assign ReadData1 = (ReadRegister1 == 5'b0)? 32'b0 : regs[ReadRegister1]; //端口 1 数据读出
    assign ReadData2 = (ReadRegister2 == 5'b0)? 32'b0 : regs[ReadRegister2]; //端口 2 数据读出
    always @(posedge clk)
        if(RegWrite) regs[WriteRegister] <= WriteData;
endmodule
```

Code 1 寄存器堆 verilog 代码

根据寄存器堆的原理框图，寄存器堆的 verilog 代码如 Code1 所示。

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。原理框图如图 7 所示

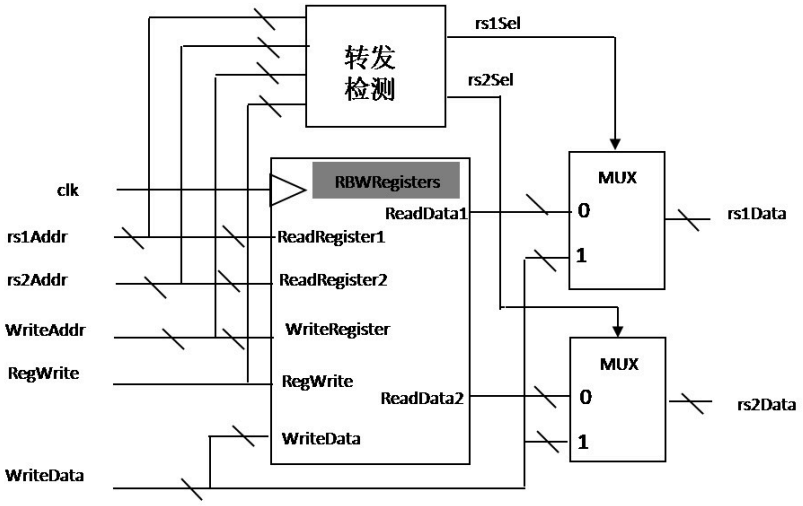


图 7 具有 Read After Write 特性的寄存器堆原理图

根据前文的分析，图中转发检测电路的输出表达式为：

```
assign rs1Sel = RegWrite && (WriteAddr != 0) && (WriteAddr == rs1Addr);
assign rs2Sel = RegWrite && (WriteAddr != 0) && (WriteAddr == rs2Addr);
```

因此，完整的寄存器堆模块 verilog 代码如 Code2 所示。

```
module Registers(clk, rs1Addr, rs2Addr, WriteAddr, RegWrite, WriteData, rs1Data, rs2Data);
    input clk;
    input [4:0] rs1Addr, rs2Addr, WriteAddr;
    input RegWrite;
    input [31:0] WriteData;
    output [31:0] rs1Data, rs2Data;
    wire rs1Sel, rs2Sel;      //转发检测电路输出
    wire [31:0] ReadData1, ReadData2;
    RBWRegisters RBWRegisters_1
    (.clk(clk),.ReadRegister1(rs1Addr), .ReadRegister2(rs2Addr),
    .WriteRegister(WriteAddr),.WriteData(WriteData),
    .RegWrite(RegWrite),.ReadData1(ReadData1),.ReadData2(ReadData2));

    assign rs1Sel = RegWrite && (WriteAddr != 0) && (WriteAddr == rs1Addr);
    assign rs2Sel = RegWrite && (WriteAddr != 0) && (WriteAddr == rs2Addr);
    //output
    assign rs1Data = rs1Sel? WriteData:ReadData1;
    assign rs2Data = rs2Sel? WriteData:ReadData2;
endmodule
```

Code 2 完整的寄存器堆模块代码

（2）指令译码（包含立即数产生电路）子模块设计

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。

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。各变量的值确定如下：

```
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);
```

其中变量 op=Instruction[6:0]为指令的操作码字段，R_type_op 等参数按照 RISC- V 各指令类别的操作码 opcode 字段定义如下：

```
parameter    R_type_op = 7'b0110011;
parameter    I_type_op = 7'b0010011;
parameter    SB_type_op = 7'b1100011;
parameter    LW_op = 7'b0000011;
parameter    JALR_op = 7'b1100111;
parameter    SW_op = 7'b0100011;
parameter    LUI_op = 7'b0110111;
parameter    AUIPC_op = 7'b0010111;
parameter    JAL_op = 7'b1101111;
```

下面确定各使能信号和选择器片选信号。

①只有 LW 指令读取存储器且回写数据取自存储器，所以有：

```
assign MemtoReg = LW;
assign MemRead = LW;
```

②只有 SW 指令会对存储器写数据，所以有：

```
assign MemWrite = SW;
```

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：

```
assign RegWrite = R_type || I_type || LW || JALR || LUI || AUIPC || JAL;
```

④只有 JALR 和 JAL 两条无条件分支指令，所以有：

```
assign Jump = JALR || JAL;
```

下面确定操作数 A 和 B 的选择信号。

分析各类指令，按照表 1 的定义，可得到表 2 操作数选择的功能表。从表 2 中获得 ALUSrcA_id 和 ALUSrcB_id[1:0]的表达式：

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

表 2 ALU 操作数选择信号确定

类型	ALUSrcA_id	ALUSrcB_id[1:0]	说明
R_type	0	2'b00	rd = rs1 op rs2
I_type	0	2'b01	rd = rs1 op imm
LW	0	2'b01	rs1 + imm
SW	0	2'b01	rs1 + imm
JALR	1	2'b10	rd = pc + 4
JAL	1	2'b10	rd = pc + 4
LUI	x	2'b01	rd = imm
AUIPC	1	2'b01	rd = pc + imm

下面确定 ALUCode。除了条件分支指令，其它指令都需要 ALU 执行运算，共 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 3 所示。注意：表中 funct7[6]与 funct6[5]在指令中为同一位置，即 instruction[30]。

表 3 ALUCode 功能表

R_Type	I_Type	LUI	funct3	funct7[6] (funct6[5])	ALUCode	备注
1	0	0	3'o0	0	4'd0	加
			3'o0	1	4'd1	减
			3'o1	0	4'd6	左移 A<<B
			3'o2	0	4'd9	A<B?1:0
			3'o3	0	4'd10	A<B?1:0(无符号)
			3'o4	0	4'd4	异或
			3'o5	0	4'd7	右移 A>>B
			3'o5	1	4'd8	算术右移 A>>>B
			3'o6	0	4'd5	或
			3'o7	0	4'd3	与
0	1		3'o0	x	4'd0	加
			3'o1	x	4'd6	左移 A<<B
			3'o2	x	4'd9	A<B?1:0
			3'o3	x	4'd10	A<B?1:0(无符号)
			3'o4	x	4'd4	异或
			3'o5	0	4'd7	右移 A>>B
			3'o5	1	4'd8	算术右移 A>>>B
			3'o6	x	4'd5	或
			3'o7	x	4'd3	与
	0	1	x	x	4'd2	送数：ALUResult=B
其他					4'd0	加

由表 3，ALUCode 确定过程的 verilog 代码如 Code3 所示。

```
always@(*) begin
    if (LUI)    ALUCode = alu_lui;
    else if(R_type || I_type) begin
        case (funct3)
            3'o0:begin//add or sub
                        if((R_type&&(~funct6_7)) || I_type)    ALUCode = alu_add;
                        else if(R_type&&funct6_7)    ALUCode = alu_sub;
                end
            SLL_funct3: begin
                            if((R_type&&(~funct6_7)) || I_type)    ALUCode = alu_sll;
                        end
            SLT_funct3: begin
                            if((R_type&&(~funct6_7)) || I_type)    ALUCode = alu_slt;
                        end
            SLTU_funct3:begin
                            if((R_type&&(~funct6_7)) || I_type)    ALUCode = alu_sltu;
                        end
            XOR_funct3: begin
                            if((R_type&&(~funct6_7)) || I_type) ALUCode = alu_xor;
                        end
            3'o5: begin      //srl or sra
                    if((R_type&&(~funct6_7)) || (I_type&&(~funct6_7)))    ALUCode = alu_srl;
                    else if((R_type&&funct6_7) || (I_type&&funct6_7))     ALUCode = alu_sra;
                        end
            OR_funct3: begin
                            if((R_type&&(~funct6_7)) || I_type) ALUCode = alu_or;
                      end
            AND_funct3: begin
                            if((R_type&&(~funct6_7)) || I_type) ALUCode = alu_and;
                        end
            default:       ALUCode = alu_add;
        endcase
    end
    else ALUCode = alu_add;
end
```

Code 3 ALUCode **确定过程** verilog **代码**

下面进行立即数产生电路（ImmGen）设计。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。

```
assign Shift = (funct3 == 1) || (funct3 == 5);
```

根据表 4，确定 Imm 和 offset 的 verilog 代码如 Code4 所示。

表 4 立即数产生方法

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

```
always @(*) begin
    if(I_type) begin
            Imm <= Shift?{26'd0, Instruction[25:20]}:{{20{Instruction[31]}}, Instruction[31:20]};
            offset <= 32'bx;    end
    else if(LW) begin
            Imm <= {{20{Instruction[31]}},Instruction[31:20]}; offset <= 32'bx; end
    else if(JALR) begin
            Imm <= 32'bx; offset <= {{20{Instruction[31]}}, Instruction[31:20]}; end
    else if(SW) begin
        Imm <= {{20{Instruction[31]}}, Instruction[31:25], Instruction[11:7]}; offset <= 32'bx;
    end
    else if(JAL) begin
        Imm <= 32'bx;
        offset <= {{11{Instruction[31]}}, Instruction[31], Instruction[19:12], Instruction[20],
    Instruction[30:21], 1'b0};
    end
    else if(LUI || AUIPC) begin
        Imm <= {Instruction[31:12], 12'd0}; offset<= 32'bx;
    end
    else if(SB_type) begin
        Imm <= 32'bx;
        offset <= {{19{Instruction[31]}}, Instruction[31], Instruction[7], Instruction[30:25],
    Instruction[11:8], 1'b0};
    end
    else begin
            Imm <= 32'bx; offset <= 32'bx;
        end
    end
```

Code 4 立即数产生过程 verilog 代码

（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，应注意有符号数和无符号数处理方式的不同。本次设计使用加法器来实现。

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 的比较结果为：

assign isLT = (rs1Data[31]&&(~rs2Data[31])) || ((rs1Data[31]~^rs2Data[31])&&sum[31]);

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 rs1Data<rs2Data 的比较结果为：

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

最后用数据选择器完成分支检测。

该部分的 verilog 代码如 Code5 所示。

```
module BranchTest (Instruction, rs1Data, rs2Data, Branch);
    input[31:0] Instruction, rs1Data, rs2Data;
    output reg Branch;

    parameter SB_opcode=    7'b1100011;
    parameter beq_funct3 = 3'o0;
    parameter bne_funct3 = 3'o1;
    parameter blt_funct3 = 3'o4;
    parameter bge_funct3 = 3'o5;
    parameter bltu_funct3 = 3'o6;
    parameter bgeu_funct3 = 3'o7;

    wire[31:0] sum;
    wire[6:0] op;
    wire[2:0] funct3;
    wire isLT, isLTU, SB_type;

    assign op = Instruction[6:0];
    assign funct3 = Instruction[14:12];
    assign SB_type = (op == SB_opcode);

    Adder32bits adder(.inA(rs1Data), .inB(~rs2Data), .cin(1), .addOut(sum), .cout());
```

```
    assign isLT = (rs1Data[31]&&(~rs2Data[31])) || ((rs1Data[31]~^rs2Data[31])&&sum[31]);
    assign isLTU = ((~rs1Data[31])&&rs2Data[31]) || ((rs1Data[31]~^rs2Data[31])&&sum[31]);

    always@(*) begin
        if(SB_type && (funct3 == beq_funct3)) begin Branch = ~(|sum[31:0]); end
        else if(SB_type && (funct3 == bne_funct3)) begin Branch = |sum[31:0]; end
        else if(SB_type && (funct3 == blt_funct3)) begin Branch = isLT; end
        else if(SB_type && (funct3 == bge_funct3)) begin Branch = ~isLT; end
        else if(SB_type && (funct3 == bltu_funct3)) begin Branch = isLTU; end
        else if(SB_type && (funct3 == bgeu_funct3)) begin Branch = ~isLTU; end
        else begin Branch = 0; end
    end

endmodule
```

Code 5 **分支检测** verilog **代码**

（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

IFWrite= ~Stall

该部分较为简单，故直接在 ID 级顶层中实现。

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 5 所示。

表 5 EX **级接口信息**

方向	引脚名称	说明
Input	ALUCode_ex[3:0]	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]	MEM 级 ALU 输出数据
	rdAddr_mem[4:0]	MEM 级寄存器的写地址

	rdAddr_wb[4:0]	WB 级寄存器的写地址
Input	RegWrite_mem	MEM 级寄存器写允许信号
	RegWrite_wb	WB 级寄存器写允许信号
Output	ALUResult_ex[31:0]	EX 级 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）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 3 所示的 ALUCode 的功能表，可得到 ALU 的功能表，如表 6 所示。

表 6 ALU 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

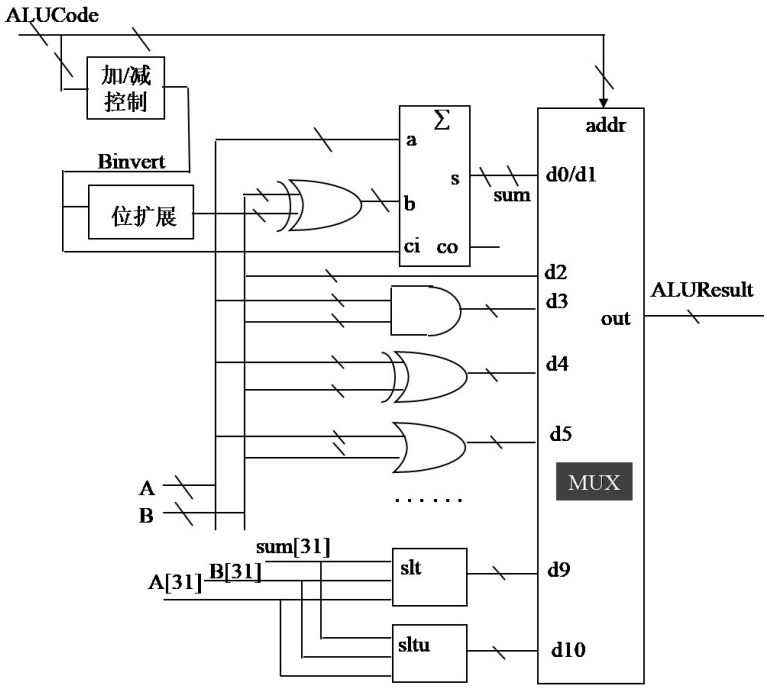


图 8 ALU 结构框图

图中 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。

为了处理有符号数运算，定义有符号寄存器型变量 A_reg（reg signed[31:0] A_reg）。

根据 ALU 结构框图和功能表，ALU 模块的 verilog 代码如 Code6 所示。

```
    wire Binvert;
    reg    [31:0] ALUResult;
    reg signed[31:0] A_reg;
    wire [31:0] sum;
    wire slt_rst, sltu_rst;

    assign Binvert = ~(ALUCode == 0);
    Adder32bits adder1(.inA(A),.inB({32{Binvert}}^B),.cin(Binvert),.addOut(sum),.cout());
    assign slt_rst = A[31]&&(~B[31]) || (A[31]~^B[31])&&sum[31];
    assign sltu_rst = (~A[31])&&B[31] || (A[31]~^B[31])&&sum[31];

    always @(*) begin
        A_reg = A;
        case (ALUCode)
            alu_add: ALUResult = sum;
            alu_sub: ALUResult = sum;
            alu_lui: ALUResult = B;
            alu_and: ALUResult = A&B;
            alu_xor: ALUResult = A^B;
            alu_or:    ALUResult = A|B;
            alu_sll: ALUResult = A<<B;
            alu_srl: ALUResult = A>>B;
            alu_sra: ALUResult = A_reg >>>B;
            //alu_sra: ALUResult = ($signed(A)) >>> B;
            alu_slt: ALUResult = slt_rst?32'd1:32'd0;
            alu_sltu: ALUResult = sltu_rst?32'd1:32'd0;
            default: ALUResult = sum;
        endcase
    end
endmodule
```

Code 6 ALU 模块 verilog 代码

（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 7 所示。

结合一阶、二阶数据相关判断条件，代码设计 Code7。

表 7 数据前推电路输出信号含义

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

```
    wire[1:0] ForwardA, ForwardB;
    wire[31:0] A_rst, B_rst;

    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);
```

Code 7 数据前推模块 verilog 代码

3、MEM 级：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，数据存储器设计为容量为的单端口 RAM，输出采用组合输出（Non Registered）。由于数据存储器容量为 64×32 bit，故存储器地址共 6 位，与 ALUResult_mem[7:2]连接。该部分的代码涉及 IP 内核模块，具体可见工程文件。

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 8 所示。

表 8 IF 级接口信息

方向	引脚名称	说明
Input	clk	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	五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
Output	Instruction[31:0]	指令机器码
	if_flush	流水线清空信号
	PC[31:0]	PC 值

根据图 2，IF_flush 控制信号由 Branch 和 Jump 产生：

assign IF_flush = Branch || Jump;

IF 级整体代码如 Code8 所示。

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

    wire[31:0] NextPC_if, PC_tmp;
    assign IF_flush = Branch || Jump;
    assign PC_tmp = IF_flush?JumpAddr:NextPC_if;
    always@(posedge clk)
    begin
        if(reset) PC <= 0;
        else if(~IFWrite) PC <= PC;
        else PC <= PC_tmp;
    end

    Adder32bits PCadder (.inA(PC),.inB(32'd4),.cin(0),.addOut(NextPC_if),.cout());
    InstructionROM InstROM (.addr(PC[7:2]),.dout(Instruction_if));
endmodule
```

Code 8 IF 级 verilog 代码

5、WB 级设计

由于本次设计中 WB 级仅包含了一个数据选择器，故将其放在 CPU 顶层代码中实现。

6、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，EX/MEM、MEM/WB 两组流水线寄存器只是普通的 D 型寄存器；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，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；当流水线发生数据冒险时，需要阻塞 IF/ID 流水线寄存器；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，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）。

首先设计基本的带参数 D 寄存器单元，verilog 代码如 Code9 所示。在各个流水线寄存器的模块中调用 code9 的 D 寄存器模块即可。IF/ID、ID/EX、EX/MEM、MEM/WB 寄存器的 verilog 代码分别如 Code10~Code13 所示。

```
module dffre(d, en, reset, clk, q);
    parameter n = 1; //寄存器位数，n=1 为 D 触发器
    input en, reset, clk;
    input [n-1:0] d;
    output [n-1:0] q;
    reg [n-1:0] q;
    always @(posedge clk) begin
        if(reset) begin
            q= {n{1'b0}};
        end
        else begin
            q = en? d:q;
        end
    end
endmodule
```

Code 9 带参数 D 寄存器 verilog 代码

```
module IF_ID (clk, en, reset, PC_if, Instruction_if, PC_id, Instruction_id);
    input clk, en, reset;
    input[31:0] PC_if, Instruction_if;
    output [31:0] PC_id, Instruction_id;

    dffre #(.n(32)) PC_dffre (.d(PC_if),.q(PC_id),.clk(clk),.en(en),.reset(reset));
    dffre #(.n(32)) Inst_dffre (.d(Instruction_if),.q(Instruction_id),.clk(clk),.en(en),.reset(reset));

endmodule
```

Code 10 IF/ID 寄存器 verilog 代码

```
module ID_EX (clk, reset,
    MemtoReg_id, RegWrite_id, MemWrite_id, MemRead_id, ALUCode_id, ALUSrcA_id,
ALUSrcB_id, PC_id, Imm_id, rdAddr_id, rs1Addr_id, rs2Addr_id, rs1Data_id, rs2Data_id,
    MemtoReg_ex, RegWrite_ex, MemWrite_ex, MemRead_ex, ALUCode_ex, ALUSrcA_ex,
ALUSrcB_ex, PC_ex, Imm_ex, rdAddr_ex, rs1Addr_ex, rs2Addr_ex, rs1Data_ex, rs2Data_ex);

    input clk, reset, ALUSrcA_id, MemtoReg_id, RegWrite_id, MemWrite_id, MemRead_id;
    input[3:0] ALUCode_id;
    input[1:0] ALUSrcB_id;
    input[31:0] PC_id, Imm_id, rs1Data_id, rs2Data_id;
    input[4:0] rdAddr_id, rs1Addr_id, rs2Addr_id;
```

```
    output ALUSrcA_ex, MemtoReg_ex, RegWrite_ex, MemWrite_ex, MemRead_ex;
    output [3:0] ALUCode_ex;
    output [1:0] ALUSrcB_ex;
    output [31:0] PC_ex, Imm_ex, rs1Data_ex, rs2Data_ex;
    output [4:0] rdAddr_ex, rs1Addr_ex, rs2Addr_ex;

    dffre #(.n(1)) RegWrite_dffre (.d(RegWrite_id),.q(RegWrite_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(1)) MemWrite_dffre (.d(MemWrite_id),.q(MemWrite_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(1)) MemRead_dffre (.d(MemRead_id),.q(MemRead_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(4)) ALUCode_dffre (.d(ALUCode_id),.q(ALUCode_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(1)) ALUSrcA_dffre (.d(ALUSrcA_id),.q(ALUSrcA_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(2)) ALUSrcB_dffre (.d(ALUSrcB_id),.q(ALUSrcB_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(5)) rdAddr_dffre (.d(rdAddr_id),.q(rdAddr_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(5)) rs1Addr_dffre (.d(rs1Addr_id),.q(rs1Addr_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(5)) rs2Addr_dffre (.d(rs2Addr_id),.q(rs2Addr_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(32)) rs1Data_dffre (.d(rs1Data_id),.q(rs1Data_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(32)) rs2Data_dffre (.d(rs2Data_id),.q(rs2Data_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(32)) PC_dffre (.d(PC_id),.q(PC_ex),.clk(clk),.en(1),.reset(reset));
    dffre #(.n(32)) Imm_dffre (.d(Imm_id),.q(Imm_ex),.clk(clk),.en(1),.reset(reset));

endmodule
```

Code 11 ID/EX 寄存器 verilog 代码

```
module EX_MEM (clk,
    MemtoReg_ex, RegWrite_ex, MemWrite_ex, ALUResult_ex, MemWriteData_ex, rdAddr_ex,
    MemtoReg_mem, RegWrite_mem, MemWrite_mem, ALUResult_mem, MemWriteData_mem,
rdAddr_mem
    );
    input clk;
    input MemtoReg_ex, RegWrite_ex, MemWrite_ex;
    input[31:0] ALUResult_ex, MemWriteData_ex;
    input[4:0] rdAddr_ex;

    output reg MemtoReg_mem, RegWrite_mem, MemWrite_mem;
    output reg [31:0] ALUResult_mem, MemWriteData_mem;
    output reg [4:0] rdAddr_mem;

    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

endmodule
```

Code 12 EX/MEM 寄存器 verilog 代码

```
module MEM_WB (clk,
        MemtoReg_mem, RegWrite_mem, MemDout_mem, ALUResult_mem, rdAddr_mem,
        MemtoReg_wb, RegWrite_wb, MemDout_wb, ALUResult_wb, rdAddr_wb
        );
        input clk;
        input MemtoReg_mem, RegWrite_mem;
        input[31:0] MemDout_mem, ALUResult_mem;
        input[4:0] rdAddr_mem;

        output reg MemtoReg_wb, RegWrite_wb;
        output reg [31:0] MemDout_wb, ALUResult_wb;
        output reg [4:0] rdAddr_wb;

        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
endmodule
```

Code 13 MEM/WB 寄存器 verilog 代码

7、顶层文件设计

按照图 2 要求连接各模块，即可完成本次设计

由于部分模块的代码过于冗长，不便在报告中具体展示，上述内容中仅包含部分源代码。具体源代码可参考工程文件。

四、仿真与测试

（一）、Decode 模块仿真

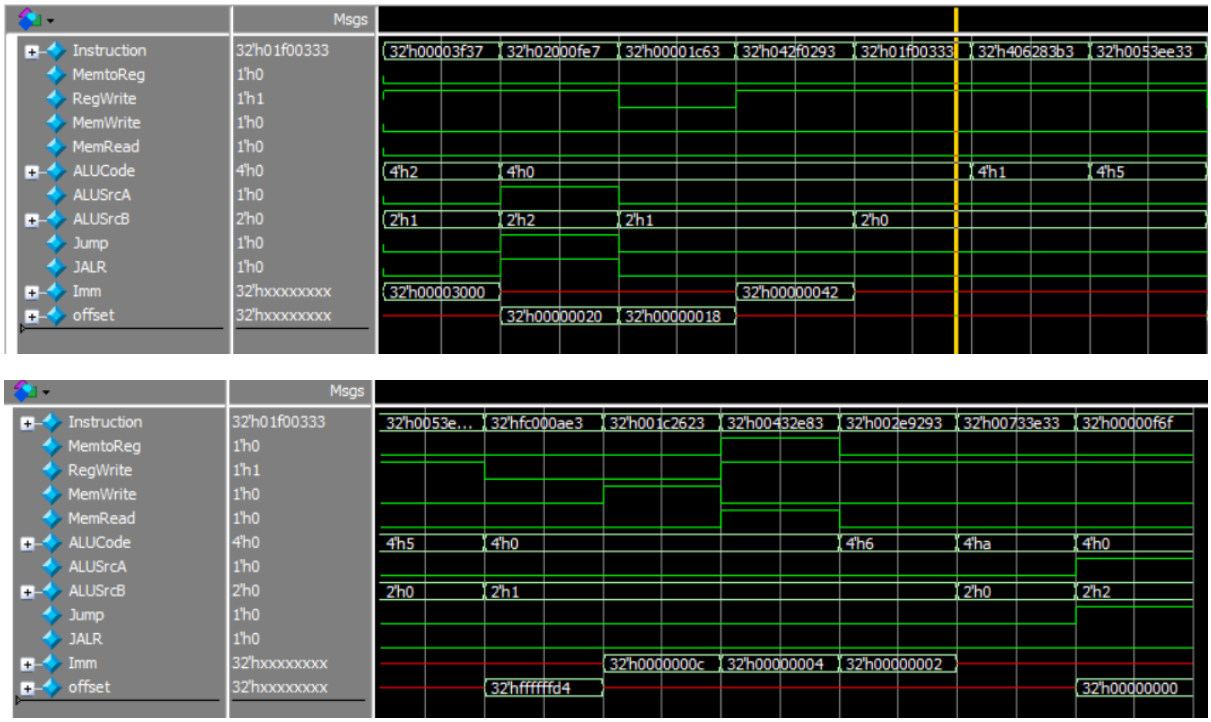


图 9 Decode 模块仿真结果图

Decode 模块仿真结果波形如图 9 所示。现对仿真信号逐一进行分析。

机器码 32’h00003f37 对应指令 lui x30, 0x3000，输出应为 Imm=32’h00003000，offset 无意义 ALUCode=4’d2（送数），ALUSrcA=0，ALUSrcB=2’b01（操作数 B 来源于立即数），RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h02000fe7 对应指令 jalr X31, later(X0)，输出应为 Imm 无意义，offset=32’h00000020，ALUCode=4’d0（加法），ALUSrcA=1（操作数 A 源于 PC），ALUSrcB=2’b10（操作数 B 为 4），RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=1，JALR=1，波形结果正确。

机器码 32’h00001c63 对应指令 bne X0, X0, end，输出应为 Imm 无意义，offset=32’h00000018，ALUCode=4’d0（加法），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b01（操作数 B 来源于立即数），RegWrite=0（不需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h042f0293 对应指令 addi X5, X30, 42H，输出应为 Imm=32’h00000042，offset 无意义，ALUCode=4’d0（加法），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b01（操作数 B 来源于立即数），RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h01f00333 对应指令 add X6, X0, X31，输出应为 Imm 无意义，offset 无意义，ALUCode=4’d0（加法），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b00（操作数 B 来源于 rs2），

RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h406283b3 对应指令 sub X7, X5, X6，输出应为 Imm 无意义，offset 无意义，ALUCode=4’d1（加法），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b00（操作数 B 来源于 rs2），RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h0053ee33 对应指令 or X28, X7, X5，输出应为 Imm 无意义，offset 无意义，ALUCode=4’d5（或），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b00（操作数 B 来源于 rs2），RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’hfc000ae3 对应指令 beq X0, X0, earlier，输出应为 Imm 无意义，offset=32’hfffffd4，ALUCode=4’d0（加法），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b01（操作数 B 来源于立即数），RegWrite=0（不需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h001c2623 对应指令 sw X28, 0C(X0)，输出应为 Imm=32’h0000000c，offset 无意义，ALUCode=4’d0（加法），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b01（操作数 B 来源于立即数），RegWrite=0（不需要回写），MemWrite=1（需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h00432e83 对应指令 lw X29, 04(X6)，输出应为 Imm=32’h00000004，offset 无意义，ALUCode=4’d0（加法），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b01（操作数 B 来源于立即数），选择信号 MemtoReg=1，RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=1（需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h002e9293 对应指令 slli X5, X29,2，输出应为 Imm=32’h00000002，offset 无意义，ALUCode=4’d6（左移），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b01（操作数 B 来源于立即数），RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h00733e33 对应指令 sltu X28, X6,X7，输出应为 Imm 无意义，offset 无意义，ALUCode=4’ha（无符号数大小判断），ALUSrcA=0（操作数 A 源于 rs1），ALUSrcB=2’b00（操作数 B 来源于 rs2），RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=0，JALR=0，波形结果正确。

机器码 32’h0000f6f 对应指令 jal X31, done，输出应为 Imm 无意义，offset=0，ALUCode=4’h0（加法），ALUSrcA=1（操作数 A 源于 PC），ALUSrcB=2’b10（操作数 B 来源于常数 4），RegWrite=1（需要回写），MemWrite=0（不需要写存储器），MemRead=0（不需要读存储器），Jump=1，JALR=0，波形结果正确。

综上，Decode 模块仿真结果正确。

（二）、ALU 模块仿真结果

ALU 模块的仿真波形结果如图 10 所示。将每一个时钟周期的操作和理论结果列为表 9，对照可得 ALU 模块仿真结果正确。

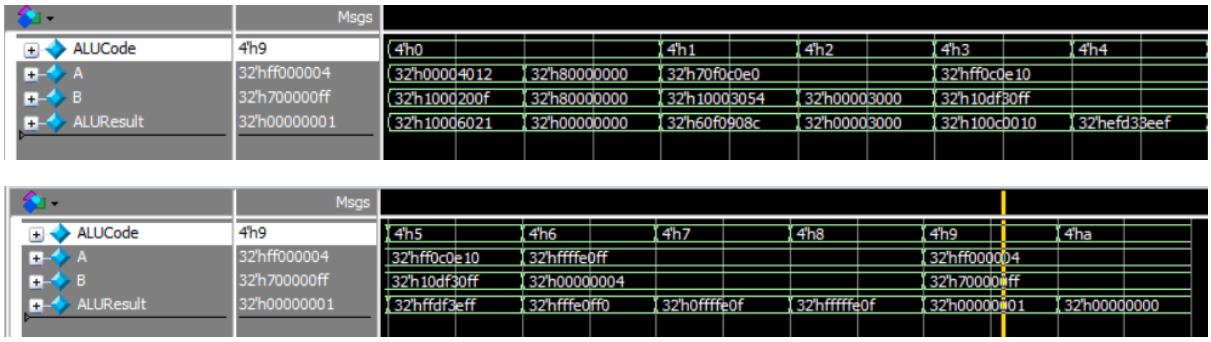
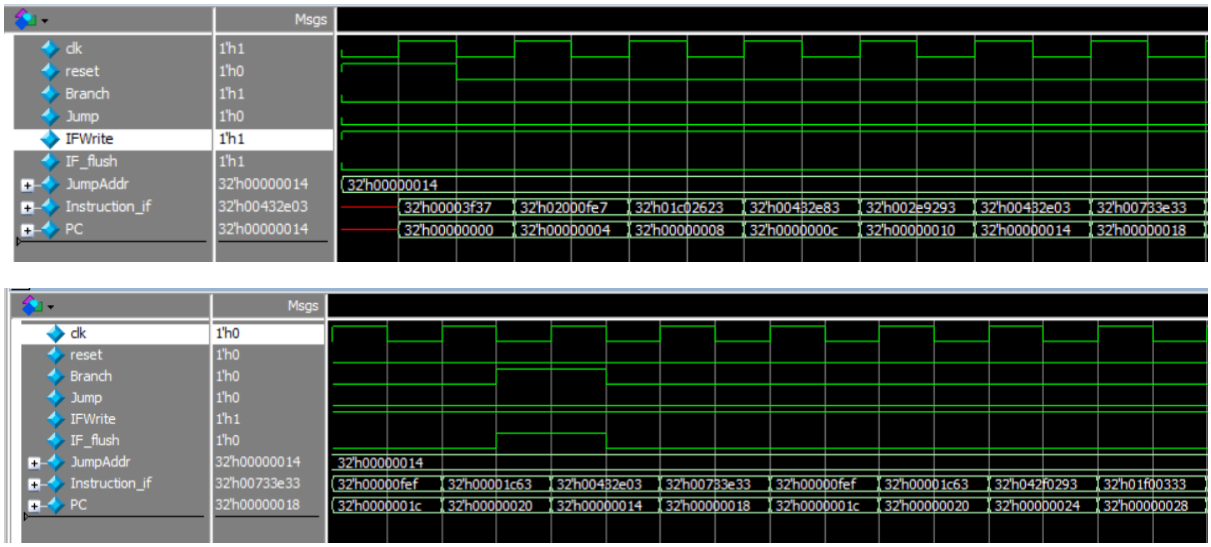


图 10 ALU 模块仿真波形

表 9 ALU 模块仿真使用数据和理论结果

ALUCode	操作	ALU_A	ALU_B	ALUResult 理论值
4’d0	A + B	32'h00004012	32'h1000200F	32’h10006021
4'd0	A + B	32'h80000000	32'h80000000	32’h00000000（溢出）
4'd1	A - B	32'h70F0C0E0	32'h10003054	32’h60F0908C
4'd2	B	32'h70F0C0E0	32'h00003000	32'h00003000
4'd3	A & B	32'hFF0C0E10	32'h10DF30FF	32'h100C0010
4'd4	A ^ B	32'hFF0C0E10	32'h10DF30FF	32'hEFD33EEF
4'd5	A | B	32'hFF0C0E10	32'h10DF30FF	32'hFFDF3EFF
4'd6	A << B	32'hFFFFE0FF	32'h00000004	32'hFFFE0FF0
4’d7	A >> B	32'hFFFFE0FF	32'h00000004	32'h0FFFE0F
4’d8	A >>> B	32'hFFFFE0FF	32'h00000004	32'hFFFFFE0F
4’d9	A < B?（无符号数）	32'hFF000004	32'h700000FF	32’h00000001
4’d10	A < B?（有符号数）	32'hFF000004	32'h700000FF	32’h00000000

（三）、IF 级仿真结果



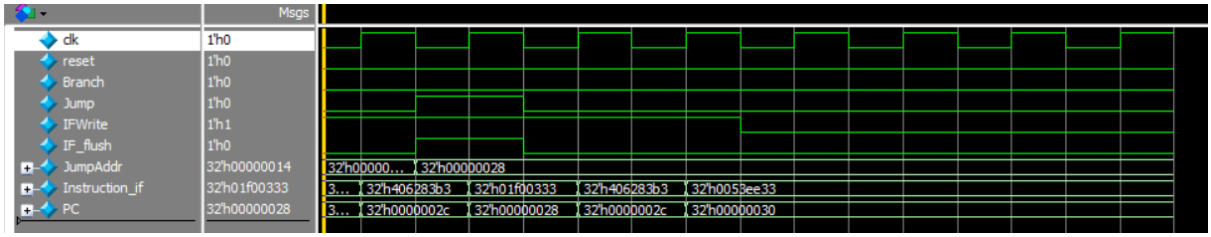


图 11 IF 级仿真波形

如图，第一次跳转时，Branch=1，IF_flush=1，PC 被设置为 JumpAddr 的值 32’h00000014；第二次跳转时，Jump=1，IF_flush=1，PC 被设置为 JumpAddr 的值 32’h00000028。其他情况下各控制指令无效，PC 正常增加。故 IF 级仿真结果正确。

（四）、CPU 顶层仿真结果

图 12 CPU 顶层仿真波形图

CPU 顶层仿真结果如图 12。对照讲义，可知结果正确，CPU 功能正确实现。

五、开发板验证

将程序下载到开发板并进行验证，运行的结果与仿真的结果一致，并通过任课老师验收。由于无法截屏只能手机拍照，记录效果不佳，在此仅做说明。

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

解答：如果加入足够多的气泡，问题可以得到解决，但严重降低 CPU 运行效率。现在的大多数 CPU 都可以做到原子读与原子写，并具有锁的功能。当 lw 的数据结果还未更新时，锁生效，无法访问相应的寄存器，这时可以切换 CPU 工作线程，让 CPU 完成其他指令，待数据更新完毕后再继续执行原指令 beq。存在的这种问题可以通过汇编器调整指令顺序来实现。

七、问题记录

（1）在设计立即数产生模块时，一开始忘记设置最后一层 else 的情况：Imm <= 32'bx; offset <= 32'bx，导致在不需要 Imm 和 offset 时仍然会有不确定值出现。加上这一句后，问题得到解决。

（2）在设计 ALU 模块时，第一次仿真运行的结果不正确。通过查看 ALU 仿真波形，得知是因为在写 always 块的时，引起变化的条件没有使用（*），导致初始化时操作数 A 和操作数 B 的值为高阻态（设定的值没有被正确载入）。将其修正后，问题得到解决。

（3）在使用 Xilinx 的 IP 内核创建 RAM 模块时，第一次选错了 IP 内核模块，选成了 Block Memory Generator，这导致了读出 RAM 数据会有额外一个时钟周期的延迟，最终导致顶层仿真结果错误。查看教材，重新逐步设置 IP 内核，注意选择 Distributed Memory Generator，确定输出结果没有延迟。再次仿真，结果正确。

（4）在连接顶层模块时，由于信号线过多，且很多信号线的名称在数据通路中未标注，导致经常出现信号线未定义或重复定义的情况，最初的连接难以着手。之后参照讲义中每一级的接口信息表，重新细读数据通路，按不同流水线级分开定义信号线。虽然在一个模块中可能会有其他级的信号作为输入，但每一部分只定义该级产生或使用的信号线，如 ID 级就不定义后缀为_wb、_ex 的信号。这样逐步减少重复项，补全遗漏项，问题得到解决。

（5）IF/ID 的重置信号输入，一开始仅仅设置为了 If_flush，导致系统复位时无法将寄存器清零。改为 If_flush | reset 后，问题得到解决。

八、讨论与总结

本次实验设计了流水线 RISC-V 微处理器，使我对 RISC-V 指令系统、流水线工作原理等知识有了更深的理解，很好地巩固了理论课知识。同时也学会了 modesim 的使用和 Xilinx IP 内核使用，加强了我对数字系统的分析、设计与调试的能力。