浙江大学

本科实验报告

课程名称：　　计算机组成与设计（周一 9，10 节）

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：

2020 年　 11 月　 24　日

浙江大学实验报告

专业：电子科学与技术
姓名：
学号：

课程名称：计算机组成与设计　　指导老师：屈民军、唐奕
实验名称：基于 RV32I 指令集的 RISC-V 微处理器设计

一、实验目的　　二、实验任务与要求
三、实验原理
四、主要仪器设备　　五、实验步骤与过程
六、实验调试、实验数据记录　　七、实验结果和分析处理
八、讨论、心得

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

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 开发系统种实现 RISC-V 微处理器，要求 CPU 的运行速度大于 25MHz。

三、实验原理与模块设计

（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。一个指令的执行需要 5 个时钟周期，每个时钟上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理。

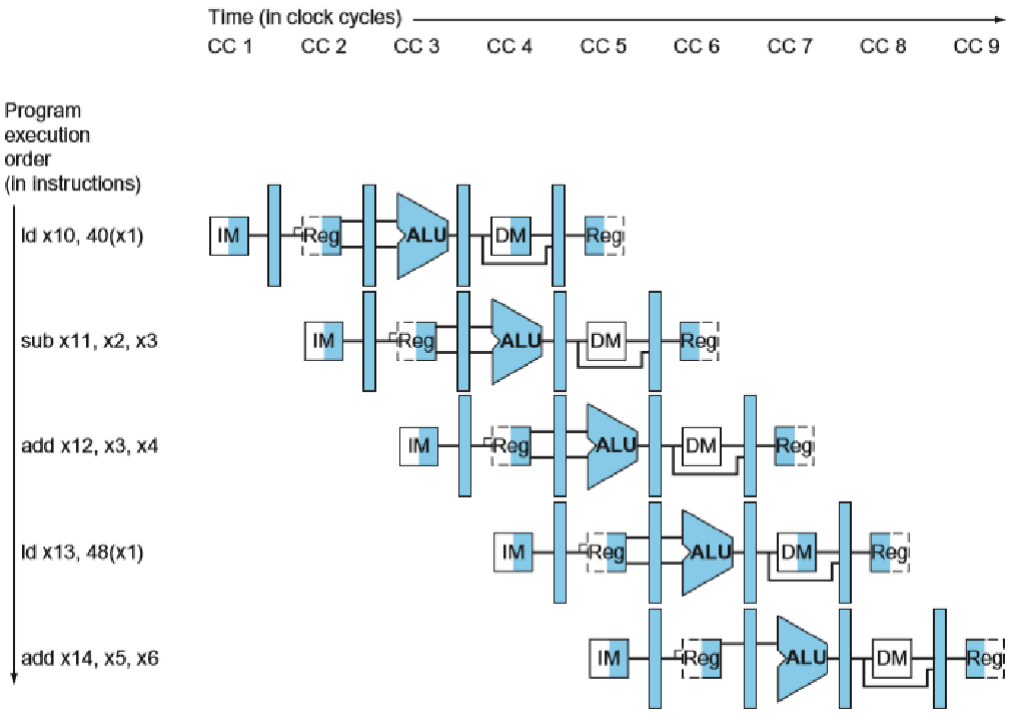


图 1 流水线流水作业示意图

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。

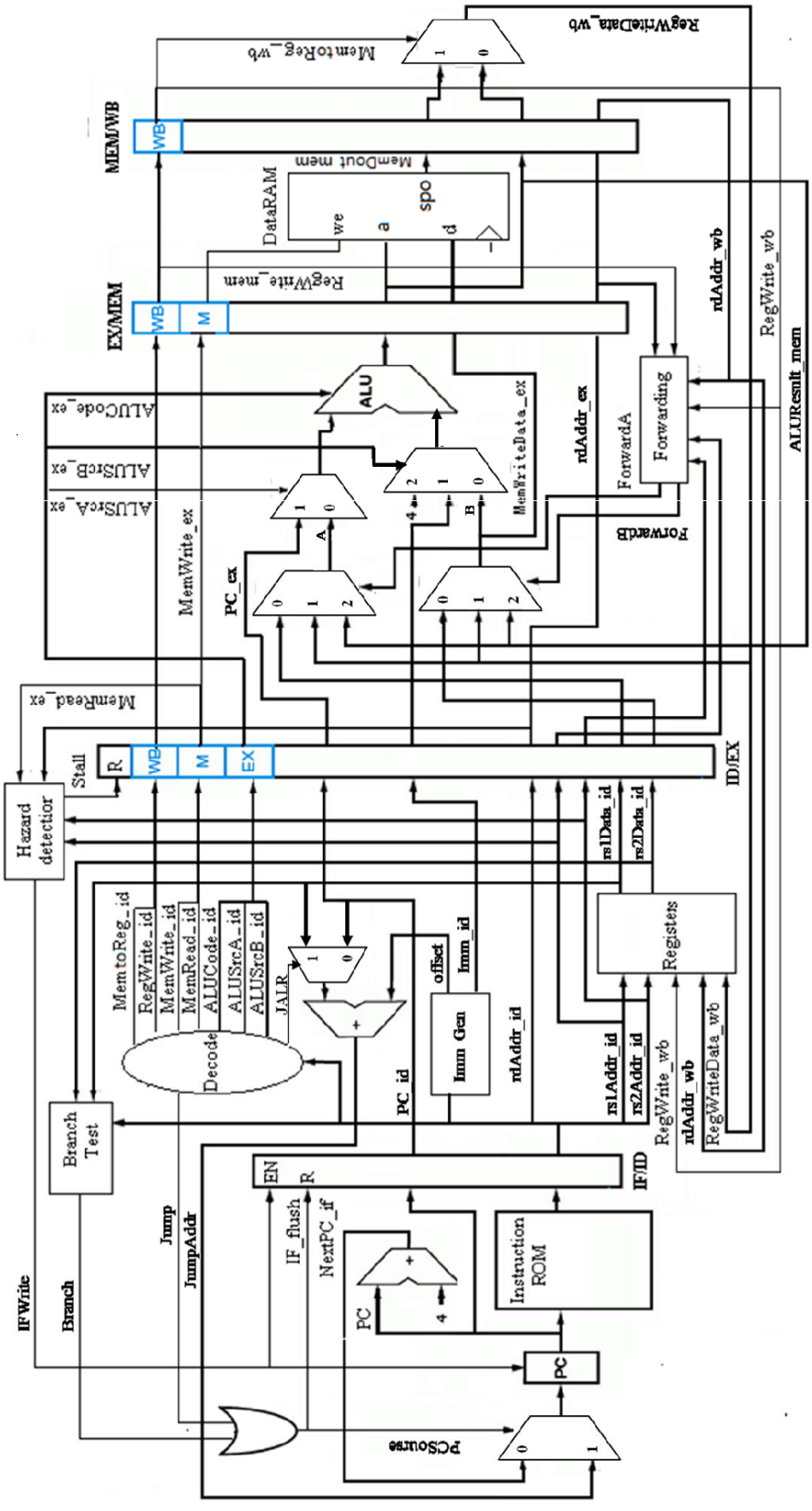


图 2　流水线 RISC-V 微处理器的原理框图

（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

（1）一阶数据相关与转发（EX 冒险）

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：

① MEM 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：

① MEM 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）。

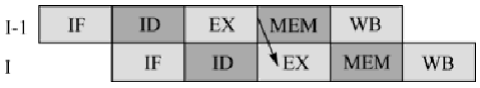


图 3 一阶前推网络示意图

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。

（2）二阶数据相关与转发（MEM 冒险）

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：

① WB 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：

① WB 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）；

图 4 二阶前推网络示意图

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 转发至 I 条指令的 EX。

（3）三阶数据相关

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）；

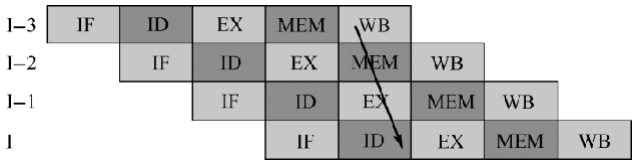


图 5 三阶前推网络示意图

同样，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 Stall 信号清空 ID/EX 寄存器，另外一个输出低电平有效的 IFWrite 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。接口信息如图 6 所示。

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]		

图 6 ID 模块的 I/O 引脚说明

（1）指令译码（Decode）（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。

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);

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;

⑤操作数 A 和 B 的选择信号的确定

类型	ALUSrcA_id	ALUSrcB_id[1:0]	说明
R_type	0	2'b00	rd=rs1 op rs2
I_type	0	2'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

图 7 操作数选择信号的功能表

由图 7 可以确定：

assign ALUSrcA = JALR || JAL || AUIPC;

assign ALUSrcB[1] = JAL || JALR;

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

⑥ALUCode 的确定

由图 8 ALUCode 的功能表，采用 case 语句：

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

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

图 8 ALUCode 的确定

⑦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。

Shift 信号的确定：

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

根据图 9，采用 if-else 语句确定 Imm 和 offset，32’bx 态表示立即数或偏移无效。

```
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
```

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}

图 9 立即数产生方法

（二）寄存器堆子模块设计

先设计 Read Before Write 寄存器堆，寄存器堆由 32 个 32 位寄存器组成，“0”号寄存器为常数 0。根据图 10 寄存器堆的原理框图，设计如下：

```
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; //数据写入
```

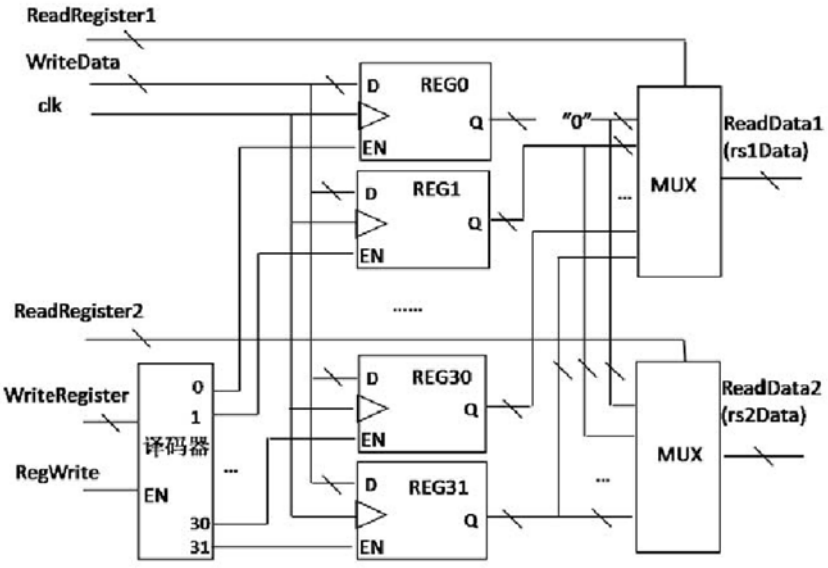


图 10 寄存器堆的原理框图

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 11 所示

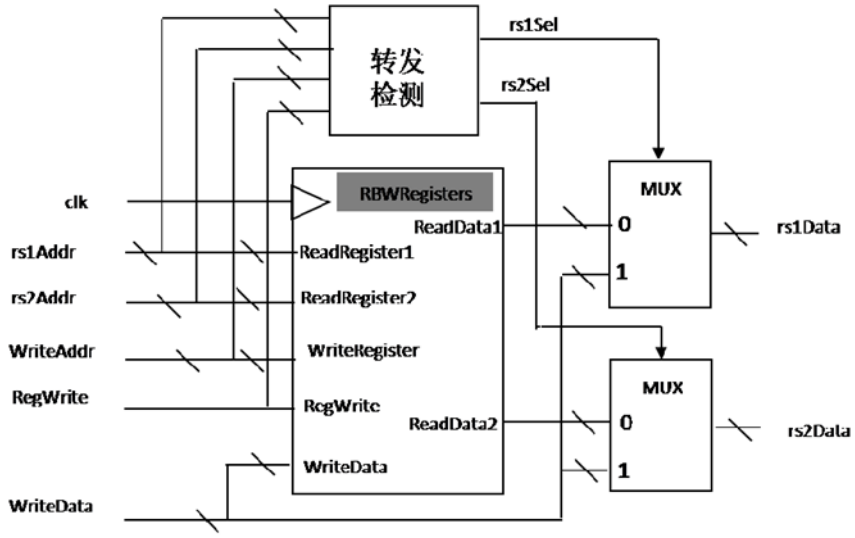


图 11 具有 Read After Write 特性寄存器堆的原理框图

代码设计如下：

```
RBWRegisters RBWRegisters_1(
    .clk(clk),
    .ReadRegister1(rs1Addr),
    .ReadRegister2(rs2Addr),
    .WriteRegister(WriteAddr),
    .WriteData(WriteData),
    .RegWrite(RegWrite),
    .ReadData1(ReadData1),
    .ReadData2(ReadData2)
);
//转发检测电路输出
assign rs1Sel = RegWrite && (WriteAddr!=0) && (WriteAddr==rs1Addr);
assign rs2Sel = RegWrite && (WriteAddr!=0) && (WriteAddr==rs2Addr);
//最终输出
assign rs1Data = rs1Sel?WriteData:ReadData1;
assign rs2Data = rs2Sel?WriteData:ReadData2;
```

（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，符号数和无符号数处理方法不同。这里用 32 位加法器实现。

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]

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]

最后用数据选择器完成分支检测。

代码设计如下：

```
adder_32bits adder(.a(rs1Data), .b({32{1'b1}}^rs2Data), .ci(1), .s(sum), .co());
assign isLT = rs1Data[31]&&(~rs2Data[31]) || (rs1Data[31]~^rs2Data[31])&&sum[31];
assign isLTU = (~rs1Data[31])&&rs2Data[31] || (rs1Data[31]~^rs2Data[31])&&sum[31];
//select Branch's value
always@(*)
begin
    if(SB_type) begin
        case (funct3)
            beq_funct3: Branch = ~(|sum[31:0]);
            bne_funct3: Branch = |sum[31:0];
            blt_funct3: Branch = isLT;
            bge_funct3: Branch = ~isLT;
            bltu_funct3:Branch = isLTU;
            bgeu_funct3:Branch = ~isLTU;
            default:    Branch = 0;
        endcase
    end
    else Branch = 0;
end
```

(4)冒险检测功能电路（Hazard Detector）的设计

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，代码设计如下：

```
assign Stall = ((rdAddr_ex==rs1Addr_id) || (rdAddr_ex==rs2Addr_id)) && MemRead_ex;
assign IFWrite = ~Stall;
```

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图 12 所示。

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]		

图 12　EX 模块的 I/O 引脚说明

（1）ALU 子模块的设计

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。

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 12　ALU 功能表

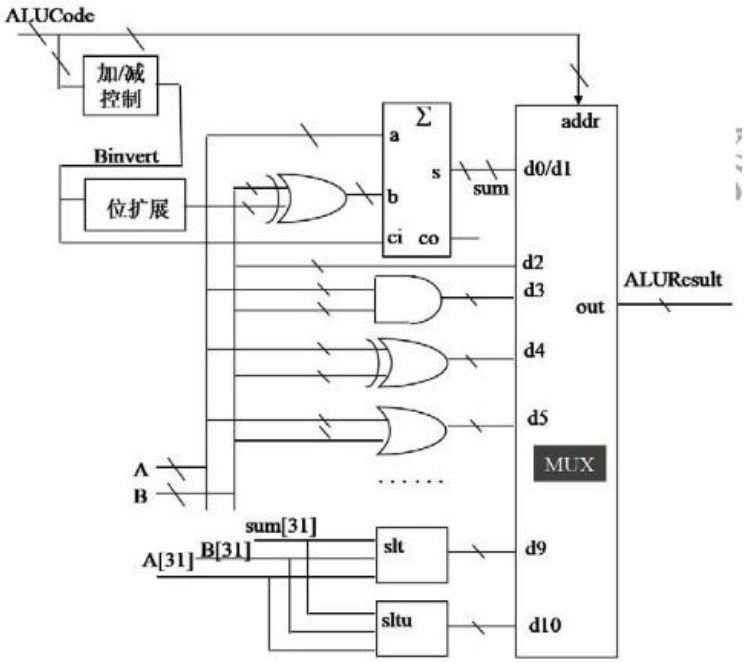


图 13　ALU 结构图

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。

ALU 子模块核心代码设计如下：

```
reg signed[31:0] A_reg;
assign Binvert = ~(ALUCode == 0);
adder_32bits adder(.a(A), .b({32{Binvert}}^B), .ci(Binvert), .s(sum), .co());
assign isLT = A[31]&&(~B[31]) || (A[31]~^B[31])&&sum[31];
assign isLTU = (~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_slt: ALUResult = isLT?32'd1:32'd0;
            alu_sltu: ALUResult = isLTU?32'd1:32'd0;
            default: ALUResult = sum;
        endcase
    end
```

（2）数据前推电路的设计

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

图 14 前推电路输出信号的含义

结合一阶、二阶数据相关判断条件，代码设计如下：

```
    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);
```

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。

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 模块接口信息如图 15 所示。

IF_flush 控制信号由 Branch 和 Jump 产生：

```
    assign IF_flush = Branch || Jump;
```

指令指针选择器根据 IF_flush 信号选择 IFWrite=1 时的 PC 值，设计如下：

```
    mux2 #(.n(32)) m_1(
```

```
        .in0(NextPC_if),
        .in1(JumpAddr),
        .addr(IF_flush),
        .out(PC_tmp)
    );
```

指令指针寄存器 PC 受 IFWrite 信号控制，当 IFWrite=1 时，输出下一个 PC 值，当需要 Stall 时，PC 值保持不变，设计如下：

```
    always@(posedge clk)
    begin
        if(reset) PC <= 0;
        else if(IFWrite) PC <= PC_tmp;
        else PC <= PC;
    end
```

32 位加法器设计如下：

```
    adder_32bits adder_1(
        .a(PC),
        .b(32'd4),
        .ci(0),
        .s(NextPC_if),
        .co()
    );
```

指令存储器子模块（Instruction ROM）根据 PC 值输出指令值，设计如下：

```
    InstructionROM Inst_1(
        .addr(PC[7:2]),
        .dout(Instruction_if)
    );
```

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。

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）。

其中 IF/ID 流水线寄存器核心代码如下：

```
always@(posedge clk)
begin
    if(R) begin PC_id <= 0; Instruction_id <= 0; end
    else if(EN) begin PC_id <= PC_if; Instruction_id <= Instruction_if; end
    else begin PC_id <= PC_id; Instruction_id <= Instruction_id; end
end
```

6、顶层文件设计

按照图 2 的原理框图连接各模块即可。

说明：上述各模块仅列出子模块的核心代码，各模块顶层设计均根据图 2 的原理框图连接各子模块，模块端口数量较多，代码冗长，不便在报告中体现，具体代码可见 src 文件。

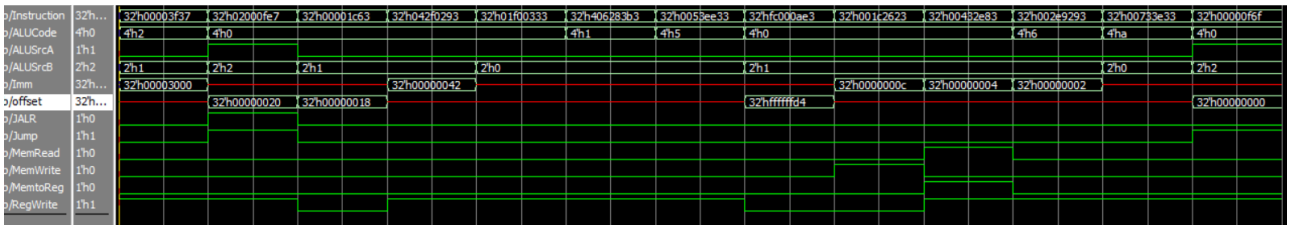
四、主要仪器设备

Modelsim、Vivado

五、实验结果

（1）Decode 模块 ModelSim 功能仿真结果

完整仿真图：



前 6 条指令仿真图：

lui x30, 0x3000，为 U 型，输出应为 Imm = 0x0000_3000，offset 无意义，ALUCode=4'd2，ALUSrcA=0，ALUSrcB=2'b01，LUI 指令需要回写，RegWrite=1，波形显示正确。

jalr X31, later(X0)，为 I 型，输出应为 Imm 无意义，offset=0x0000_0020，ALUCode=4'd0，ALUSrcA=1，ALUSrcB=2'b10，JALR=1，Jump=1，JALR 指令需要回写，RegWrite=1，波形显示正确。

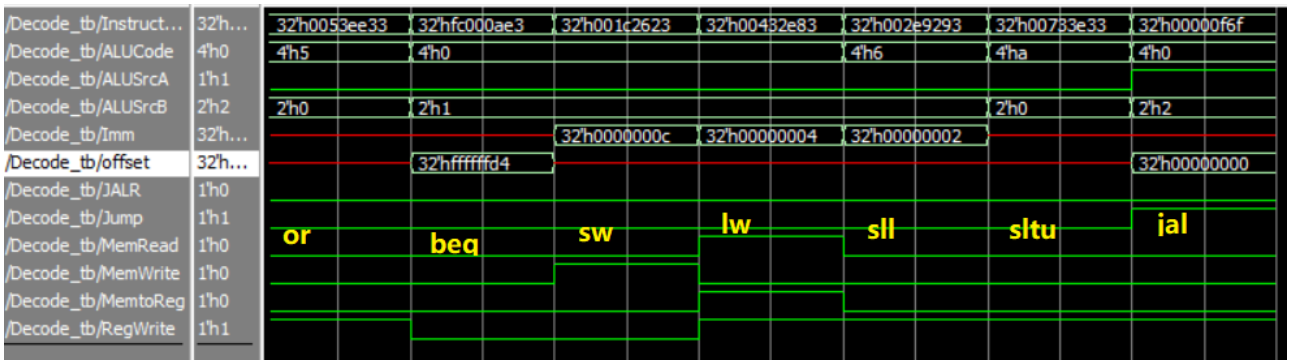
bne X0, X0, end，为 SB 型，输出应为 Imm 无意义，offset=0x0000_0018，ALUCode=4'd0，ALUSrcA=0，ALUSrcB=2'b01，BNE 指令不需要回写，RegWrite=0，波形显示正确。

addi X5, X30, 42，为 I 型，输出应为 Imm =0x0000_0042，offset 无意义，ALUCode=4'd0，ALUSrcA=0，ALUSrcB=2'b01，ADDI 指令需要回写，RegWrite=1，波形显示正确。

add X6, X0, X31，为 R 型，输出应为 Imm 无意义，offset 无意义，ALUCode=4'd0，ALUSrcA=0，ALUSrcB=2'b00，ADD 指令需要回写，RegWrite=1，波形显示正确。

sub X7, X5, X6，为 R 型，输出应为 Imm 无意义，offset 无意义，ALUCode=4'd1，ALUSrcA=0，ALUSrcB=2'b00，SUB 指令需要回写，RegWrite=1，波形显示正确。

后 7 条指令仿真结果图：



or X28, X7, X5，为 R 型，输出应为 Imm 无意义，offset 无意义，ALUCode=4'd5，ALUSrcA=0，ALUSrcB=2'b00，OR 指令需要回写，RegWrite=1，波形显示正确。

beq X0, X0, earlier，为 SB 型，输出应为 Imm 无意义，offset=0xffff_ffd4，ALUCode=4'd0，ALUSrcA=0，ALUSrcB=2'b01，BEQ 指令不需要回写，RegWrite=0，波形显示正确。

sw X28, 0C(X0)，为 S 型，输出应为 Imm =0x0000_000c，offset 无意义，ALUCode=4'd0，ALUSrcA=0，ALUSrcB=2'b01，MemWrite=1，波形显示正确。

lw X29, 04(X6)，为 I 型，输出应为 Imm =0x0000_0004，offset 无意义，ALUCode=4'd0，ALUSrcA=0，ALUSrcB=2'b01，MemRead=1，MemtoReg=1，RegWrite=1，波形显示正确。

sll X5, X29,2，为 I 型，输出应为 Imm =0x0000_0002，offset 无意义，ALUCode=4'd6，ALUSrcA=0，ALUSrcB=2'b01， RegWrite=1，波形显示正确。

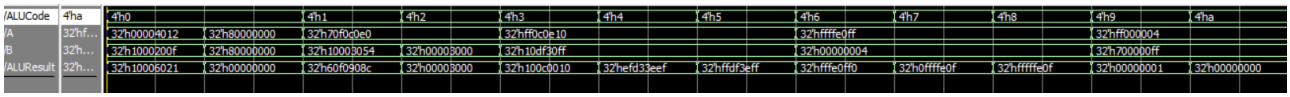
sltu X28, X6,X7，为 R 型，输出应为 Imm 无意义，offset 无意义，ALUCode=4’d10，ALUSrcA=0，ALUSrcB=2’b00， RegWrite=1，波形显示正确。

jal X31, done，为 UJ 型，输出应为 Imm 无意义，offset=0x0000_0000，ALUCode=4’d1，ALUSrcA=1，ALUSrcB=2’b10，SUB 指令需要回写，RegWrite=1，波形显示正确。

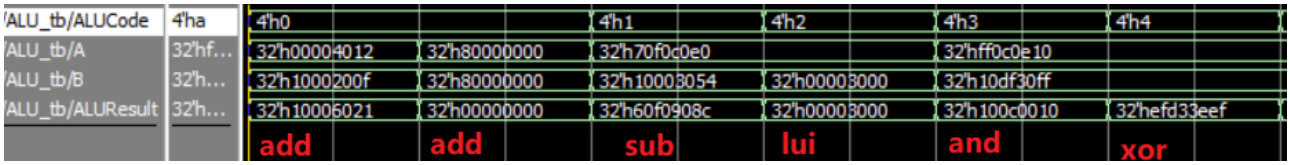
综上，Decode 模块功能实现。

（2）ALU 模块 ModelSim 功能仿真结果

完整仿真图：



前半仿真图：



简单计算，结果正确。

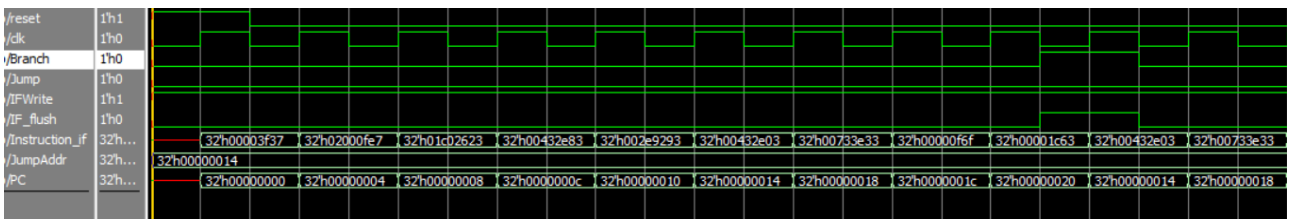
后半仿真图：

简单计算，结果正确。

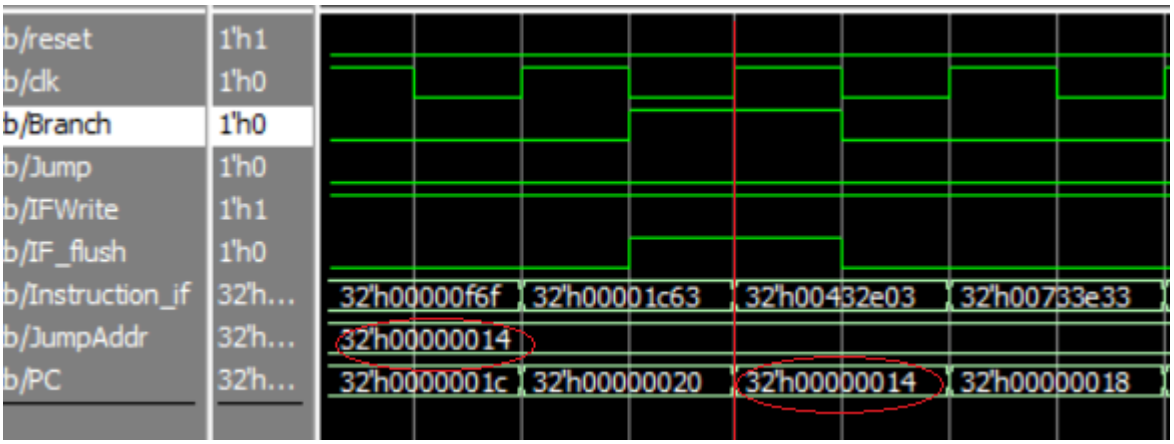
综上 ALU 模块功能实现

（3）IF 模块 ModelSim 功能仿真结果

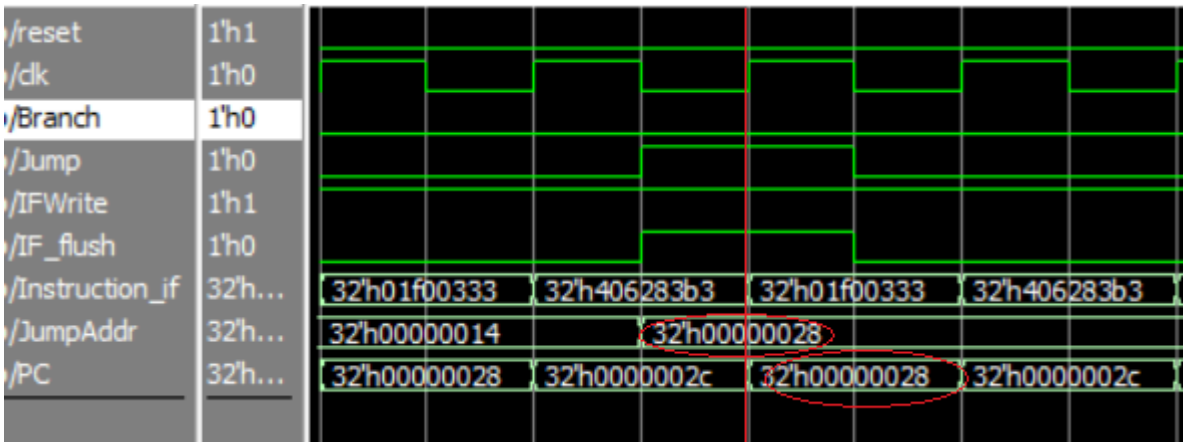
前半仿真图：



局部放大：



如图，Branch=1，则 IF_flush=1，PC 被设置为 JumpAddr 的值 0x0000_0014。

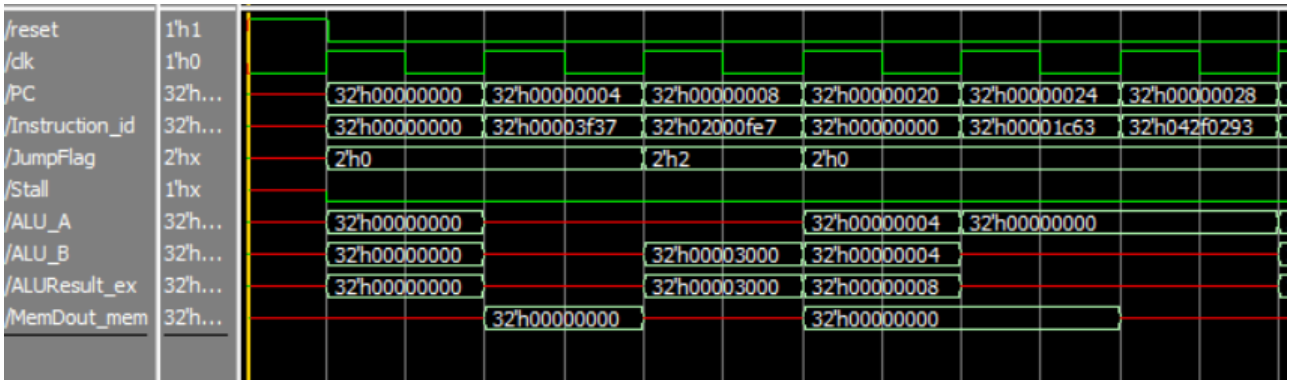


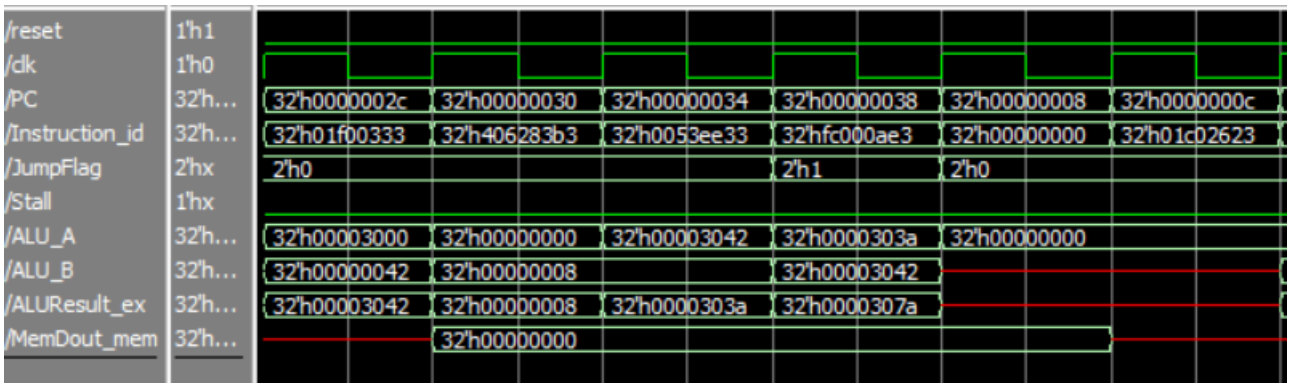
如图，Jump=1，则 IF_flush=1，PC 被设置为 JumpAddr 的值 0x0000_0028。

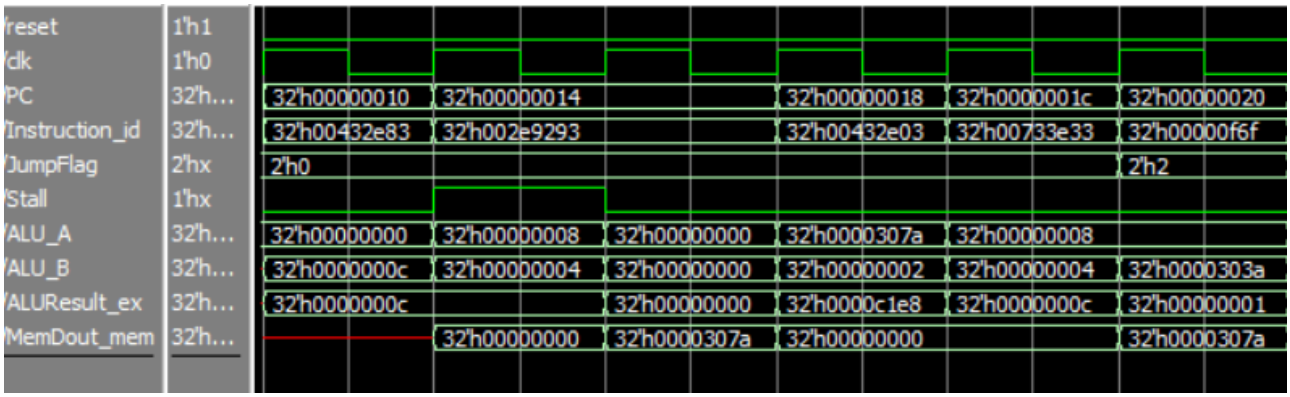
综上，IF 模块功能实现。

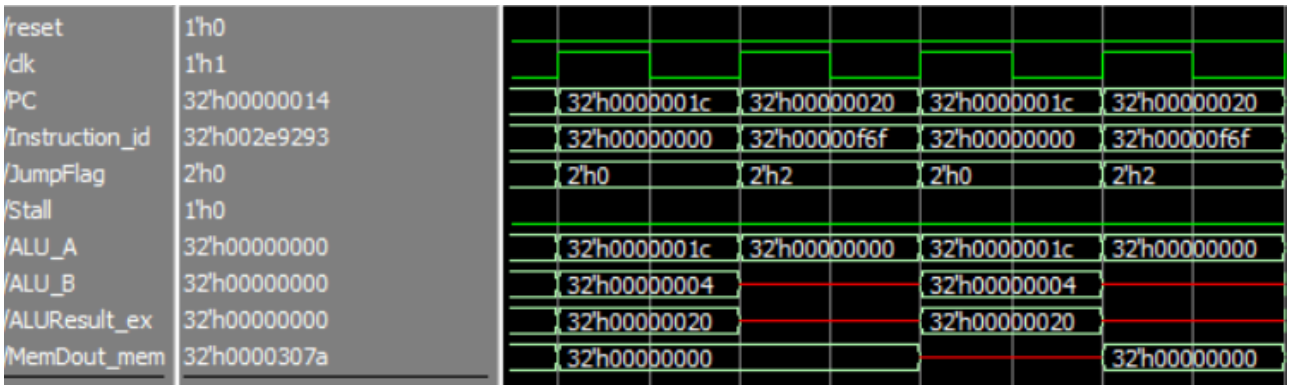
（4）CPU 顶层 ModelSim 功能仿真结果

仿真图（4 张图连续）：



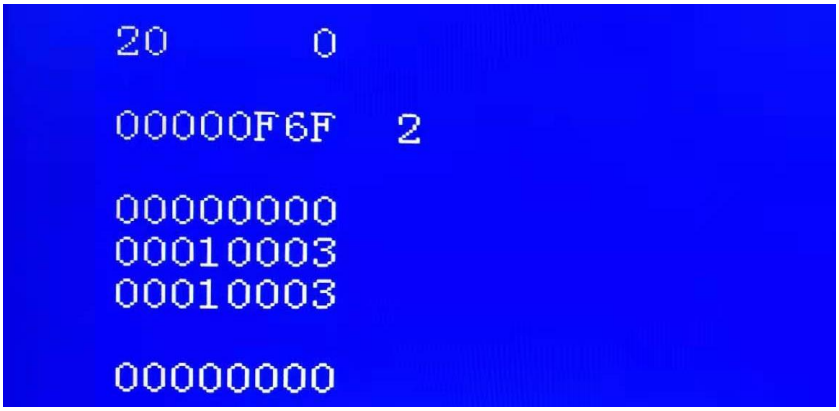






测试程序运行结果正确，顶层模块功能实现。

（5）开发板上验证



六、问题记录

（1）本次实验先做的时 ALU 模块，在设计 ALU 模块的时候碰到了数据选择器实现的问题。因长时间没有接触 Verilog HDL 语言，忘记了其具体的实现方式。最开始想设计一个 MUX 的子模块，但是子模块需要很多个输入输出端口，不便于与运算结合，代码也会不够简洁。后来通过查阅资料，想起来了用 case 语句，合理利用 parameter，实现 MUX 的功能。

（2）在设计 ALU 的时候遇到 Binvert 位扩展的问题，百度得到了结果，应该是{32{Binvert}}形式，最外面一定要有花括号，不能用()。本次实验用到了很多数据位扩展的知识。

（3）设计 Decode 模块的时候，输出 ALUCode 使用了 case 语句，当 R_type=1 或 I_type=1 时，通过 funct3 确定最终的 ALUCode 值，但是波形与手算的总有几个点对不上，反复阅读 ALUCode 的功能表，对照代码，发现 case 没有 default，加上了 default：ALUCode=alu_add。加上了还是不对，又读了几遍功能表，发现 lui 没有单独考虑，把它放进了 case 语句里面，但是 lui 的 funct3 是任何值都可以，应该放到 case 外通过 if 单独判断。lui 单独考虑之后，波形还是不对，然后又读了功能表，发现没有考虑 LUI=0，R_type=0，I_type=0 的情况，这种情况应该让 ALUCode=alu_add。另外立即数产生模块在仿真的时候也出现了问题，我用的是 if-else 实现，R 型指令的 Imm 和 offset 应该都是不确定状态，但是出现了 offset 确定的情况，发现是代码里面少了最后的 else，即 Imm <= 32'bx; offset <= 32'bx;设计 Decode 模块的教训是，case 一定要有 default，if 一定要跟 else，不然会出现意想不到的错误。

（4）设计寄存器堆的时候，我一开始直接通过一个.v 文件实现，最后的顶层仿真结果一直高阻，但那时候所有需要仿真的模块都已实现功能。最初我选择在 ModelSim 里面分实例查看波形，发现各个实例的波形都是一些正确，一些高阻，一些不确定。后来我人为设定实例的输入，例如把 IF 模块的 Branch 值定为 0 或 1，Jump 值定为 0 或 1，IFWrite 的值定为 0 或 1，然后查看波形，发现 IF 实例能够正确输出结果但是顶层模块还是高阻，然后往后排查，这种方法帮助我定位到了寄存器堆的错误。偷懒的方法不可取，还是老老实实按照文档的要求，先设计 Read Before Write 寄存器堆，再设计 Read After Write 寄存器堆，这样基本不会出错。

（5）IF 模块设计是比较难的，主要是文档没有说的很清楚，得自己体会。我最开始一直没有理解，按照原理框图设计的话，最开始的 PC 值哪里来，也没有想明白没有在原理框图里面体现的 reset 有什么用，对原理框图里面的 PC 寄存器也很困惑，不知道怎么设计。只能先设计 IF 模块里的其它子模块，剩下 PC 最后解决。尝试了很多次，才想明白 PC 是个带同步清零功能的 D 寄存器，reset=1 可以给 PC 一个初始值 0。解决了 PC 的问题，仿真便正确了。

（6）顶层模块设计的时候，寄存器堆的问题已经在上文提过了，但是在解决寄存器堆问题后波形还是不对。我先把各个端口的输入输出检查了一遍，发现了一些连线错误后，仿真还是不对。主要是 ID 级

PC 和 Instruction 值有时候不正确，别的波形基本正确。后来的解决方法是把 IF/ID 寄存器的 R 输入改成 IF_flush | reset，我一开始的设计是 R 为 IF_flush，这会使没有使能信号输入也没有初值的时候，寄存器的 PC_id 和 Instruction_id 将为不确定的值。

（7）最好的解决实验遇到的各种问题的方法就是抽一个完整的时间段做完实验。

七、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

插入一个气泡，然后将 MEM/WB 寄存器的数据转发到 EX，这样似乎是可以的。经典的 MIPS Ⅰ使用了负载延迟槽，在 lw 后面加入 nop 但是负载延迟在硬件上非常不可预知，从 RAM 或缓存 load，可能会因资源竞争而变慢。负载延迟使延迟增加，因此大多数 CPU 架构中不去解决这一问题。后来的 MIPS 增加了死锁来避免填充 nop。这一问题主要靠汇编器调整指令顺序来解决。