浙江大学

本科实验报告

RISC-V CPU 设计

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：	
指导老师：	

January 22, 2024

目　录

一、实验目的 **3**

二、实验任务 **3**

1. 基本要求 . . . 3
2. 扩展要求 . . . 3

三、实验原理 **3**

1. 流水线中的控制信号 . . . 3
2. 数据相关与数据转发 . . . 4
3. 数据冒险与数据转发 . . . 5

四、分块设计及代码 **5**

1. 指令译码模块（ID）的设计 . . . 5
   (1) 寄存器堆（Registers）子模块的设计 . . . 5
   (2) 指令译码（包含立即数产生电路）子模块的设计 . . . 7
   (3) 分支检测（Branch Test）电路的设计 . . . 11
   (4) ID 顶层描述 . . . 12
2. 执行模块（EX）的设计 . . . 14
   (1) ALU 子模块的设计 . . . 14
   (2) 数据前推电路的设计 . . . 17
3. 数据存储器模块 (DataRAM) 的设计 . . . 19
4. 取指令级模块 (IF) 的设计 . . . 19
5. 流水线寄存器的设计 . . . 20
   (1) IF/ID . . . 20
   (2) ID/EX . . . 21
   (3) EX/MEM . . . 22
   (4) MEM/WB . . . 22
6. 顶层文件的设计 . . . 23

五、仿真与结果分析 **26**

1. ALU 仿真结果 . . . 26
2. Decode 模块仿真结果 . . . 27
3. IF 模块仿真结果 . . . 27
4. CPU 顶层模块仿真结果 . . . 28

六、思考题 **29**

七、遇到的问题及解决方案 **29**

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）了解用软件实现数字系统的方法。

二、 实验任务

1. 基本要求

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：
a. 算术运算指令：add、sub、addi
b. 逻辑运算指令：and、or、xor、slt、sltu、andi、ori、xori、slti、sltiu
c. 移位指令：sll、srl、sra、slli、srli、srai
d. 条件分支指令：beq、bne、blt、bge、bltu、bgeu
e. 无条件跳转指令：jal、jalr
f. 数据传送指令：lw、sw、lui、auipc
g. 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。

2. 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、 实验原理

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，广泛应用于高档 CPU 的架构中。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，每个时钟周期的上升治来临时，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。

由于在流水线中，数据和控制信息将在时钟周期的上升沿转移到下一级，所以规定流水线转移的变量命名遵守如下格式: 名称流水线级名称。如，在 ID 级指令译码电路（decode）产生的寄存器写允许信号 RegWrite 在 ID 级、EX 级、MEM 级和 WB 级上的命名分别为 RegWrite_id,RegWrite_ex ,RegWrite_mem 和 RegWrite_wb。在顶层文件中，类似的变量名称有近百个，这样的命名方式起到了很好的识别作用。

1. 流水线中的控制信号

(1) IF 级: 取指令级。

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。该级共有三个控制信号:

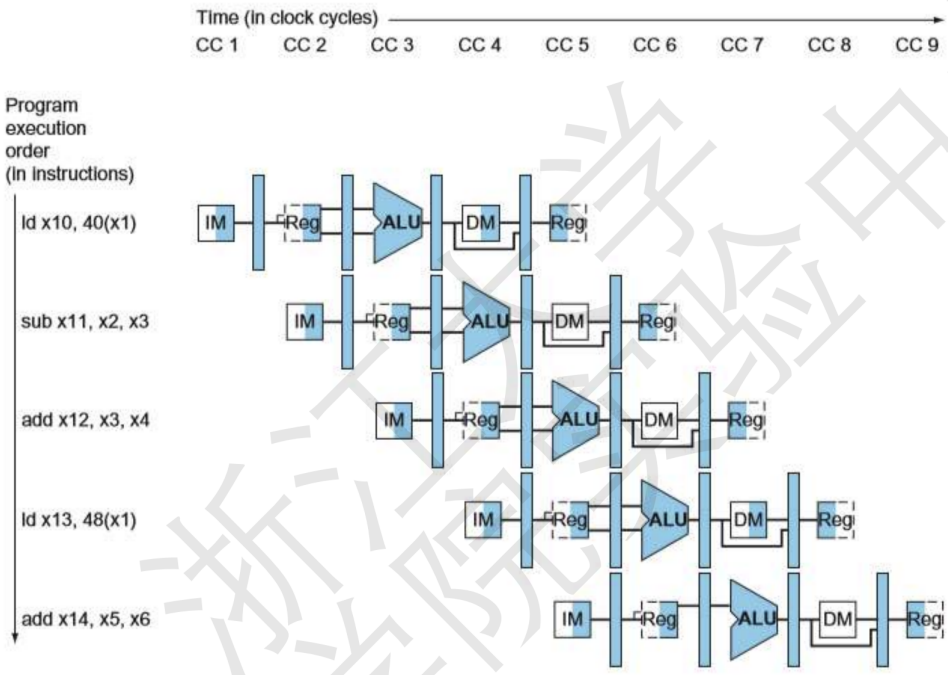

Figure 1: 流水线 CPU 工作示意图

a.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: 而当 PC-Source=1l 时，跳转执行。

b.IFWiite: IFWrite=0 时阻塞 IFMID 流水线，同时暂停读取下一条指令。

c.IF_flush: IF_ftush=1l 时清空 IFMID 寄存器。

(2) ID 级: 指令译码级。对来自正级的指令进行译码，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 IDAEX 寄存器，同时冒险检测电路产生低电平下 Write 信号阻塞 IEAD 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。

(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，对其他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此类相关问题。

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。

四、 分块设计及代码

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。

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]		

Figure 2: ID 模块的输入输出引脚说明

(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。应注意的是，“0”号寄存器为常数 0。

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。

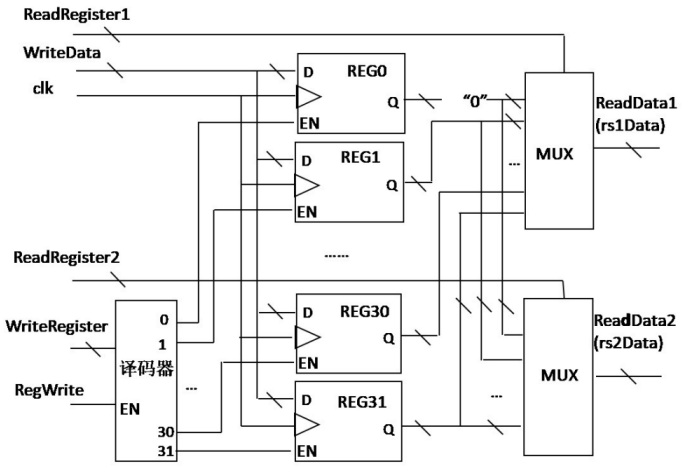

Figure 3: 寄存器堆原理图

```
module RBWRegisters (clk, ReadRegister1, ReadRegister2, WriteRegister, WriteData, RegWrite,
    ReadData1, ReadData2);
input clk;
input[4:0] ReadRegister1, ReadRegister2, WriteRegister;
input[31:0] WriteData;
input RegWrite;
output[31:0] ReadData1, ReadData2;

reg[31:0] regs[31:0];
assign ReadData1 =(ReadRegister1 ==5'b0)?32'b0:regs[ReadRegister1];
assign ReadData2 =(ReadRegister2 ==5'b0)?32'b0:regs[ReadRegister2];
always @(posedge clk) if(RegWrite) regs[WriteRegister] <=WriteData;
endmodule
```

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，只需要在图 3设计寄存器堆的基础上添加少量电路就可实现 Read After Write 特性，如图 4所示。

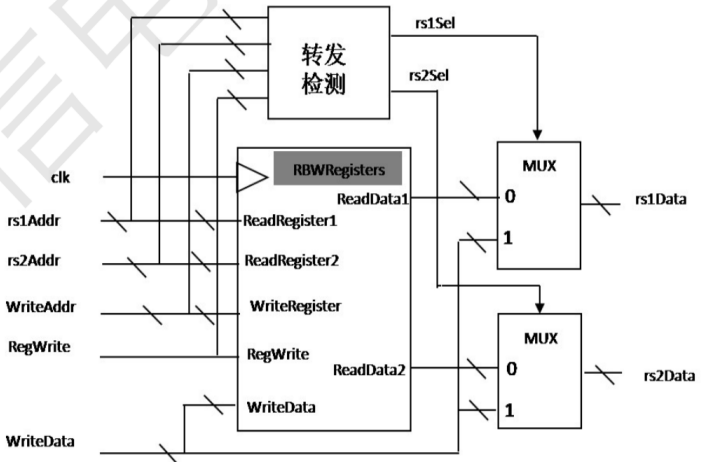

Figure 4: 具有 RAW 特性的寄存器堆原理图

其中，转发检测电路的输出逻辑为：

$$rs1Sel = RegWrite\&\&(WriteAddr! = 0)\&\&(WriteAddr == rs1Addr)$$

$$rs2Sel = RegWrite\&\&(WriteAddr! = 0)\&\&(WriteAddr == rs2Addr)$$

根据该图可以得到以下代码来填入转发检测电路，从而实现 RAW 功能:

```
module registers (clk, rs1Addr, rs2Addr, RegWrite, WriteAddr, WriteData, rs1Data, rs2Data);
input clk, RegWrite;
input[4:0] rs1Addr, rs2Addr, WriteAddr;
input[31:0] WriteData;
output[31:0] rs1Data, rs2Data;

wire[31:0] ReadData1, ReadData2;
RBWRegisters RBWRegisters0(
// Inputs
.clk(clk),
.ReadRegister1(rs1Addr),
.ReadRegister2(rs2Addr),
.WriteRegister(WriteAddr),
.WriteData(WriteData),
.RegWrite(RegWrite),
// Outputs
.ReadData1(ReadData1),
.ReadData2(ReadData2)
);

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

assign rs1Data =rs1Sel?WriteData:ReadData1;
assign rs2Data =rs2Sel?WriteData:ReadData2;
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，其值由下式决定。

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

Figure 5: 指令译码表达式

根据该表达式，可以得到相应的指令判别代码：

```
//指令类型判别
wire R_type, I_type, SB_type, LW, JALR, SW, LUI, AUIPC, JAL;
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);
```

a. 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
MemRead_id =LW
b. 只有 SW 指令会对存储器写数据，所以有
MemWrite_id =SW
c. 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
d. 只有 JALR 和 JAL 两条无条件分支指令，所以有
Jump=JALR || JAL
e. 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

Figure 6: 操作数功能表

对以上表达进行编程，即得到：

```
 //LW 指令读取存储器且回写数据取自存储器
assign MemtoReg=LW;
assign MemRead=LW;

//SW 指令会对存储器写数据
assign MemWrite=SW;

//需要进行回写的指令类型有 R_type、I_type、LW、JALR、LUI、AUIPC 和 JAL
assign RegWrite =R_type ||I_type ||LW ||JALR ||LUI ||AUIPC ||JAL;

//JALR 和 JAL 两条无条件分支指令
assign Jump=JALR ||JAL;

//操作数 A 和 B 的选择信号的确定
assign ALUSrcA =JALR ||JAL ||AUIPC;
assign ALUSrcB[1] =JAL ||JALR;
assign ALUSrcB[0] =~(R_type ||JAL ||JALR);
```

f.ALUCode 的确定

确定 ALUcode 的功能表如图所示：

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

Figure 7: ALUcode 功能表

```
//ALUCode 的确定
always@(*) begin
   if (LUI&(~R_type)&(~I_type))
      ALUCode =alu_lui;
   else if((R_type|I_type)&(~LUI))
      begin
      case (funct3)
      ADD_funct3: begin
         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
      SRL_funct3: begin
         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
```

g. 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。其表达式如下：

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

立即舒产生方法如图所示：

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}

Figure 8: 立即数产生方法

因此，代码结构为：

```
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
```

```
26      else if(SB_type) begin
27         Imm =32'bx;
28         offset ={{19{Instruction[31]}}, Instruction[31], Instruction[7], Instruction[30:25],
               Instruction[11:8], 1'b0};
29      end
30      else begin
31         Imm =32'bx;
32         offset =32'bx;
33      end
34   end
```

(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。

a. 用一个 32 位加法器完成 rs1Data+（ rs2Data）+1（即 rs1Data-rs2Data），设结果为 sum [31:0] 。备注：此处使用 lab8 中编写的 32 位进位选择加法器。

b. 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] $\neq$ rs2Data，可根据 rs1Data、rs2Data 决定比较结果，但是应注意符号数、无符号数的最高位 rs1Data、rs2Data 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、rs2a) 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] ŝs2Data[31]) &&| sum[31]

因此，符号数 rs1Data<rs2Data 比较运算结果为

$$isLT = rs1Data[31]\&\&(\ rs2Data[31])Il(rs1Data[31]\ rs2Data[31])\&\&sum[31]$$

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] ŝs2Data[31]) && sum[31]

因此，无符号数比较运算结果为

$$isLTU = (\ rs1Data[31])\&\&rs2Data[31]||(rs1Data[31]\ rs2Data[31])\&\&sum[31]$$

```
1   module BranchTest(
2   input[31:0] Instruction,
3   input[31:0] rs1Data,
4   input[31:0] rs2Data,
5   output reg Branch
6   );
7
8   parameter SB_type_op =7'b1100011;
9   parameter BEQ_FUNCT3 =3'o0;
10  parameter BNE_FUNCT3 =3'o1;
11  parameter BLT_FUNCT3 =3'o4;
12  parameter BGE_FUNCT3 =3'o5;
13  parameter BLTU_FUNCT3 =3'o6;
14  parameter BGEU_FUNCT3 =3'o7;
15
16  wire[31:0] sum;
17  Add_32bit adder_BT(
18  .a(rs1Data),
19  .b(~rs2Data),
20  .ci(1),
21  .sum(sum),
22  .c_out()
```

```
    );

    wire isLT, isLTU;
    assign isLT =rs1Data[31]&&(~rs2Data[31]) ||(rs1Data[31]~^rs2Data[31])&&sum[31];//绗～彿鑲版瘪权?
    assign isLTU =(~rs1Data[31])&&rs2Data[31] ||(rs1Data[31]~^rs2Data[31])&&sum[31];//鏃犵□鍙锋暟姣旇緝

    wire[6:0] op;
    wire[2:0] funct3;
    wire SB_type;

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

    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

    endmodule
```

(4)　ID 顶层描述

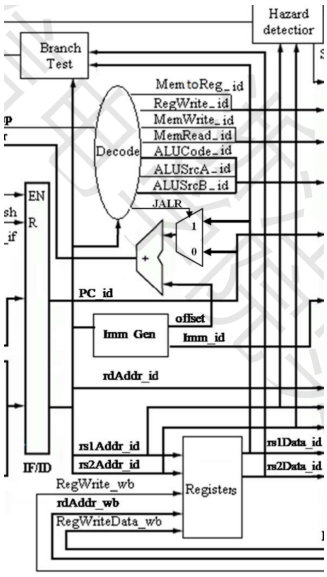

Figure 9: ID 模块原理图

根据该原理图，我们可以将各模块链接形成 ID 的顶层设计：

```
module ID(clk,Instruction_id, PC_id, RegWrite_wb, rdAddr_wb, RegWriteData_wb, MemRead_ex,
rdAddr_ex, MemtoReg_id, RegWrite_id, MemWrite_id, MemRead_id, ALUCode_id,
ALUSrcA_id, ALUSrcB_id, Stall, Branch, Jump, IFWrite, JumpAddr, Imm_id,
rs1Data_id, rs2Data_id,rs1Addr_id,rs2Addr_id,rdAddr_id);

input clk;
input [31:0] Instruction_id;
input [31:0] PC_id;
input RegWrite_wb;
input [4:0] rdAddr_wb;
input [31:0] RegWriteData_wb;
input MemRead_ex;
input [4:0] rdAddr_ex;
output MemtoReg_id;
output RegWrite_id;
output MemWrite_id;
output MemRead_id;
output [3:0] ALUCode_id;
output ALUSrcA_id;
output [1:0]ALUSrcB_id;
output Stall;
output Branch;
output Jump;
output IFWrite;
output [31:0] JumpAddr;
output [31:0] Imm_id;
output [31:0] rs1Data_id;
output [31:0] rs2Data_id;
output[4:0] rs1Addr_id,rs2Addr_id,rdAddr_id;

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

registers registers_ID(
// Inputs
.clk(clk),
.rs1Addr(rs1Addr_id),
.rs2Addr(rs2Addr_id),
.RegWrite(RegWrite_wb),
.WriteAddr(rdAddr_wb),
.WriteData(RegWriteData_wb),
// Outputs
.rs1Data(rs1Data_id),
.rs2Data(rs2Data_id)
);


wire JALR;
wire[31:0] offset;

Decode Decode0(
.Instruction(Instruction_id),
.MemtoReg(MemtoReg_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)
);

BranchTest BranchTest0(
```

```
    .Instruction(Instruction_id),
    .rs1Data(rs1Data_id),
    .rs2Data(rs2Data_id),
    .Branch(Branch)
    );

    wire[31:0] base_addr;
    assign base_addr =JALR?rs1Data_id:PC_id;

    Add_32bit adder_ID(
    .a(offset),
    .b(base_addr),
    .ci(0),
    .sum(JumpAddr),
    .c_out()
    );

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

    endmodule
    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。其输入输出信号如下所示。

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]		

Figure 10: EX 模块输入输出信号

(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。

对于在 ALU 中所使用的 32bit 加法器，直接使用实验 8 中的超进位加法器。

```
    module Add_4bit(
    input[3:0] a,
    input[3:0] b,
    input c_in,
```

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

Figure 11: ALU 功能表

```
    output[3:0] sum,
    output c_out
    );
    wire[3:0] g,p,c;

    assign p=a^b;
    assign g=a&b;
    assign c[0]=g[0]|(p[0]&c_in);
    assign c[1]=g[1]|(p[1]&(g[0]|(p[0]&c_in)));
    assign c[2]=g[2]|(p[2]&(g[1]|(p[1]&(g[0]|(p[0]&c_in)))));
    assign c[3]=g[3]|(p[3]&(g[2]|(p[2]&(g[1]|(p[1]&(g[0]|(p[0]&c_in)))))));
    assign c_out=c[3];

    assign sum[0]=p[0]&(!g[0])^c_in;
    assign sum[1]=p[1]&(!g[1])^c[0];
    assign sum[2]=p[2]&(!g[2])^c[1];
    assign sum[3]=p[3]&(!g[3])^c[2];

    endmodule


    module Add_middle(
    input [3:0] a,
    input [3:0] b,
    input c_in,

    output reg [3:0] sum,
    output c_out
    );

    wire [3:0] sum1, sum2;
    wire c_out1, c_out2;
    reg c1=0;
    reg c2=1;
    Add_4bit A41(.a(a), .b(b), .c_in(c1), .sum(sum1), .c_out(c_out1));
    Add_4bit A42(.a(a), .b(b), .c_in(c2), .sum(sum2), .c_out(c_out2));
    assign c_out=c_in &c_out2 |c_out1;

    always @(*)begin
    if(c_in ==0) sum=sum1;
    else sum=sum2;
    end

    endmodule


    module Add_32bit(
    input [31:0] a,
    input [31:0] b,
    input ci,
```

```
57      output [31:0]sum,
58      output c_out
59      );
60      wire c1, c2, c3, c4, c5, c6, c7;
61      Add_4bit A40(.a(a[3:0]), .b(b[3:0]), .c_in(ci), .sum(sum[3:0]), .c_out(c1));
62      Add_middle AM1(.a(a[7:4]), .b(b[7:4]), .c_in(c1), .sum(sum[7:4]), .c_out(c2));
63      Add_middle AM2(.a(a[11:8]), .b(b[11:8]), .c_in(c2), .sum(sum[11:8]), .c_out(c3));
64      Add_middle AM3(.a(a[15:12]), .b(b[15:12]), .c_in(c3), .sum(sum[15:12]), .c_out(c4));
65      Add_middle AM4(.a(a[19:16]), .b(b[19:16]), .c_in(c4), .sum(sum[19:16]), .c_out(c5));
66      Add_middle AM5(.a(a[23:20]), .b(b[23:20]), .c_in(c5), .sum(sum[23:20]), .c_out(c6));
67      Add_middle AM6(.a(a[27:24]), .b(b[27:24]), .c_in(c6), .sum(sum[27:24]), .c_out(c7));
68      Add_middle AM7(.a(a[31:28]), .b(b[31:28]), .c_in(c7), .sum(sum[31:28]), .c_out(c_out));
69      endmodule
```

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：

$$Binvert = (ALUCode == 0)$$

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。

根据以上分析，即可对 ALU 模块进行描述：

```
1       module ALU (
2       // Outputs
3       ALUResult,
4       // Inputs
5       ALUCode, A, B);
6       input [3:0] ALUCode;       // Operation select
7       input [31:0] A, B;
8       output reg [31:0] ALUResult;
9
10      // Decoded ALU operation select (ALUsel) signals
11      parameter alu_add= 4'b0000;
12      parameter alu_sub= 4'b0001;
13      parameter alu_lui= 4'b0010;
14      parameter alu_and= 4'b0011;
15      parameter alu_xor= 4'b0100;
16      parameter alu_or =4'b0101;
17      parameter  alu_sll= 4'b0110;
18      parameter alu_srl= 4'b0111;
19      parameter alu_sra= 4'b1000;
20      parameter alu_slt= 4'b1001;
21      parameter alu_sltu= 4'b1010;
22
23      wire c1, c2;
24      wire [31:0]result_add, result_sub, result_slt, result_sltu;
25      wire [31:0]B_sub;
26      wire signed [31:0] A_sign;
27      assign A_sign=A;
28
29      assign B_sub=~B;
30      Add_32bit ADD_alu1(.a(A), .b(B), .ci(0), .sum(result_add), .c_out(c1));
31      Add_32bit ADD_alu2(.a(A), .b(B_sub), .ci(1), .sum(result_sub), .c_out(c2));
32
33      always@(*) begin
34         case(ALUCode)
35         alu_add: ALUResult=result_add;
36         alu_sub: ALUResult=result_sub;
37         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_sign>>>B;
        alu_slt: ALUResult= (c2==1)? 1:0;
        alu_sltu:
           begin
           if(A<B) ALUResult=1;
           else ALUResult=0;
           end
        default: ALUResult=result_add;
        endcase
    end

    endmodule
```

(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 的含义如表所示。

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

Figure 12: 前推电路输出信号的含义

$$ForwardA[0] = RegWrite\_wb\&\&(rdAddr\_wb! = 0)\&\&(rdAddr\_mem! = rs1Addr\_ex)\&\&(rdAddr\_wb == rs1Addr\_ex)$$

$$ForwardA[1] = RegWrite\_mem\&\&t(rdAddr\_mem! = 0)\&\&(rdAddr\_mem == rs1Addr\_ex)$$

$$ForwardB[0] = RegWrite\_wb\&\&(rdAddr\_wb! = 0)\&\&(rd\_mem! = rs2Addr\_ex)\&\&(rdAddr\_wb == rs2Addr\_ex)$$

$$ForwardB[1] = RegWrite\_mem\&\&(rdAddr\_mem! = 0)\&\&(rdAddr\_mem == rs2Addr\_ex)$$

数据前推电路直接在 EX 的顶层实现。并根据 EX 模块原理图进行顶层连接。

```
    module EX(ALUCode_ex, ALUSrcA_ex, ALUSrcB_ex,Imm_ex, rs1Addr_ex, rs2Addr_ex, rs1Data_ex,
    rs2Data_ex, PC_ex, RegWriteData_wb, ALUResult_mem,rdAddr_mem, rdAddr_wb,
    RegWrite_mem, RegWrite_wb, ALUResult_ex, MemWriteData_ex, ALU_A, ALU_B);

    input [3:0] ALUCode_ex;
    input ALUSrcA_ex;
    input [1:0]ALUSrcB_ex;
    input [31:0] Imm_ex;
    input [4:0] rs1Addr_ex;
    input [4:0] rs2Addr_ex;
    input [31:0] rs1Data_ex;
    input [31:0] rs2Data_ex;
    input [31:0] PC_ex;
    input [31:0] RegWriteData_wb;
    input [31:0] ALUResult_mem;
    input [4:0]rdAddr_mem;
    input [4:0] rdAddr_wb;
    input RegWrite_mem;
```

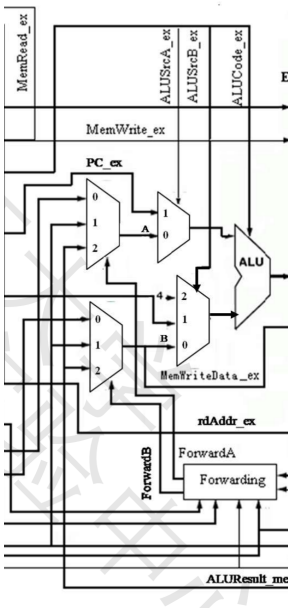

Figure 13: EX 模块原理图

```
19      input RegWrite_wb;
20      output [31:0] ALUResult_ex;
21      output [31:0] MemWriteData_ex;
22      output [31:0] ALU_A;
23      output [31:0] ALU_B;
24
25      wire[1:0] ForwardA, ForwardB; //mux input
26      wire [31:0] A; // mux1 output
27      wire [31:0] B; // mux2 output
28
29      assign ForwardA[0] =RegWrite_wb &&(rdAddr_wb!=0) &&(rdAddr_mem!=rs1Addr_ex) &&(
            rdAddr_wb==rs1Addr_ex);
30      assign ForwardA[1] =RegWrite_mem &&(rdAddr_mem!=0) &&(rdAddr_mem==rs1Addr_ex);
31      assign ForwardB[0] =RegWrite_wb &&(rdAddr_wb!=0) &&(rdAddr_mem!=rs2Addr_ex) &&(
            rdAddr_wb==rs2Addr_ex);
32      assign ForwardB[1] =RegWrite_mem &&(rdAddr_mem!=0) &&(rdAddr_mem==rs2Addr_ex);
33
34      mux mux_ex1(.a(rs1Data_ex), .b(RegWriteData_wb), .c(ALUResult_mem), .select(ForwardA), .
            result(A));
35      mux mux_ex2(.a(rs2Data_ex), .b(RegWriteData_wb), .c(ALUResult_mem), .select(ForwardB), .
            result(B));
36
37      //mux2 output赋值为MemWriteData，写入mem
38      assign MemWriteData_ex =B;
39
40      //2->1数据选择器 output
41      assign ALU_A =ALUSrcA_ex? PC_ex:A;
42
43      mux mux_ex3(.a(B), .b(Imm_ex), .c(32'd4), .select(ALUSrcB_ex), .result(ALU_B));
44
45      ALU ALU_ex(.ALUCode(ALUCode_ex), .A(ALU_A), .B(ALU_B), .ALUResult(ALUResult_ex));
46
47
48      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。

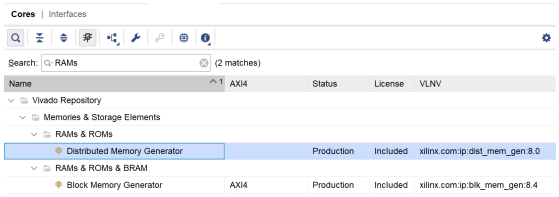

Figure 14: DataRAM1

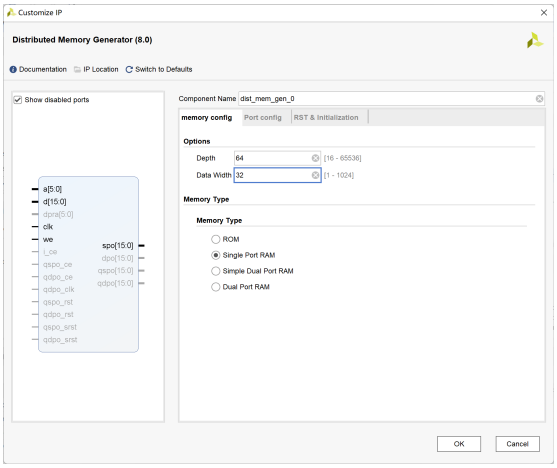

Figure 15: DataRAM2

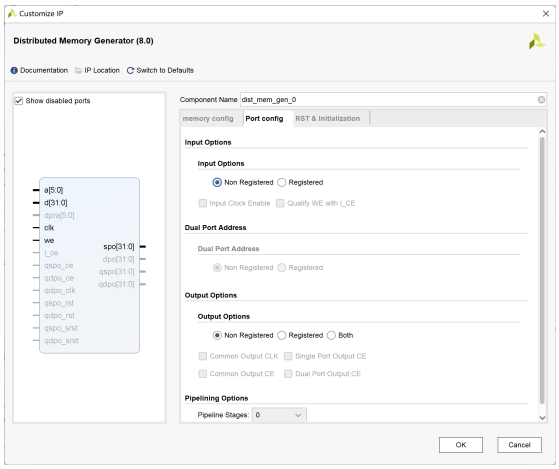

Figure 16: DataRAM3

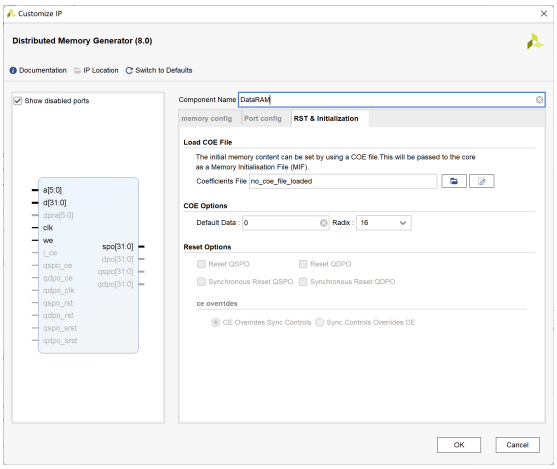

Figure 17: DataRAM4

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.

由表可看出，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。

借用提供的存储器模块 InstructionROM，对 IF 模块进行描述。

```
1       module IF(clk, reset, Branch,Jump, IFWrite, JumpAddr,Instruction_if,PC,IF_flush);
2
3       input clk;
```

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

Figure 18: IF 输入输出引脚说明

```
    input reset;
    input Branch;
    input Jump;
    input IFWrite;
    input [31:0] JumpAddr;
    output [31:0] Instruction_if;
    output [31:0] PC;
    output IF_flush;

    wire[31:0] NextPC_if, PCSource;
    reg[31:0] PC;

    assign IF_flush =Branch ||Jump;

    assign PCSource =IF_flush?JumpAddr:NextPC_if;

    always@(posedge clk) begin
       if(reset)
          PC <=32'd0;
       else if(IFWrite)
          PC <=PCSource;
       else
          PC <=PC;
    end

    //PC+4
    Add_32bit adder_IF(.a(PC), .b(32'd4), .ci(0), .sum(NextPC_if), .c_out());

    //指令存储模块调用
    InstructionROM InstROM0(.addr(PC[7:2]), .dout(Instruction_if));

    endmodule
```

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。

(1) IF/ID

```
    module IFID(
    input clk,
    input en,
    input reset,
    input[31:0] PC_if,
    input[31:0] Instruction_if,

```

```
    output reg [31:0]PC_id,
    output reg [31:0]Instruction_id
    );

    always@(posedge clk) begin
       if(reset) 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

    endmodule
```

(2) ID/EX

```
    module IDEX(
    // Inputs
    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,
    // Outputs
    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;
    input MemtoReg_id, RegWrite_id, MemWrite_id, MemRead_id;
    input[3:0] ALUCode_id;
    input ALUSrcA_id;
    input[1:0] ALUSrcB_id;
    input[4:0] rdAddr_id, rs1Addr_id, rs2Addr_id;
    input[31:0] rs1Data_id, rs2Data_id, PC_id, Imm_id;

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

    always@(posedge clk) begin
       if(reset) begin
          MemtoReg_ex <=0;
          RegWrite_ex <=0;
          MemWrite_ex <=0;
          MemRead_ex <=0;
          ALUCode_ex <=0;
          ALUSrcA_ex <=0;
          ALUSrcB_ex <=0;
          PC_ex    <=0;
          Imm_ex   <=0;
          rdAddr_ex <=0;
          rs1Addr_ex <=0;
          rs2Addr_ex <=0;
          rs1Data_ex <=0;
          rs2Data_ex <=0;
       end
       else begin
```

```
            MemtoReg_ex <=MemtoReg_id;
            RegWrite_ex <=RegWrite_id;
            MemWrite_ex <=MemWrite_id;
            MemRead_ex <=MemRead_id;
            ALUCode_ex <=ALUCode_id;
            ALUSrcA_ex <=ALUSrcA_id;
            ALUSrcB_ex <=ALUSrcB_id;
            PC_ex    <=PC_id;
            Imm_ex   <=Imm_id;
            rdAddr_ex <=rdAddr_id;
            rs1Addr_ex <=rs1Addr_id;
            rs2Addr_ex <=rs2Addr_id;
            rs1Data_ex <=rs1Data_id;
            rs2Data_ex <=rs2Data_id;
         end
      end

      endmodule
```

(3) EX/MEM

```
      module EXMEM(
      // Inputs
      clk, MemtoReg_ex, RegWrite_ex, MemWrite_ex, ALUResult_ex, MemWriteData_ex, rdAddr_ex,
      // Outputs
      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
```

(4) MEM/WB

```
      module MEMWB(
      // Inputs
      clk, MemtoReg_mem, RegWrite_mem, MemDout_mem, ALUResult_mem, rdAddr_mem,
      // Outputs
      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;
```

```
13      output reg [4:0] rdAddr_wb;
14
15      always@(posedge clk) begin
16         MemtoReg_wb <=MemtoReg_mem;
17         RegWrite_wb <=RegWrite_mem;
18         MemDout_wb  <=MemDout_mem;
19         ALUResult_wb <=ALUResult_mem;
20         rdAddr_wb   <=rdAddr_mem;
21      end
22
23      endmodule
```

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。

根据原理图，即可对各模块进行连接。

```
1       module Risc5CPU(clk, reset, JumpFlag, Instruction_id, ALU_A,
2       ALU_B, ALUResult_ex, PC, MemDout_mem,Stall);
3
4       input clk;
5       input reset;
6       output[1:0] JumpFlag;
7       output [31:0] Instruction_id;
8       output [31:0] ALU_A;
9       output [31:0] ALU_B;
10      output [31:0] ALUResult_ex;
11      output [31:0] PC;
12      output [31:0] MemDout_mem;
13      output Stall;
14
15      wire IFWrite, IF_flush;
16      wire[31:0] JumpAddr;
17      wire[31:0] Instruction_if, PC;
18
19      wire MemtoReg_id, RegWrite_id, MemWrite_id, MemRead_id;
20      wire[3:0] ALUCode_id;
21      wire ALUSrcA_id;
22      wire[1:0] ALUSrcB_id;
23      wire[4:0] rdAddr_id, rs1Addr_id, rs2Addr_id;
24      wire[31:0] rs1Data_id, rs2Data_id, PC_id, Imm_id;
25
26      wire MemtoReg_ex, RegWrite_ex, MemWrite_ex, MemRead_ex;
27      wire [3:0] ALUCode_ex;
28      wire ALUSrcA_ex;
29      wire [1:0] ALUSrcB_ex;
30      wire [4:0] rdAddr_ex, rs1Addr_ex, rs2Addr_ex;
31      wire [31:0] rs1Data_ex, rs2Data_ex, PC_ex, Imm_ex, MemWriteData_ex, ALUResult_ex;
32
33      wire MemtoReg_mem, RegWrite_mem, MemWrite_mem;
34      wire [31:0] ALUResult_mem, MemWriteData_mem;
35      wire [4:0] rdAddr_mem;
36
37      wire MemtoReg_wb, RegWrite_wb;
38      wire [31:0] MemDout_wb, ALUResult_wb, RegWriteData_wb;
39      wire [4:0] rdAddr_wb;
40
41
42      IF IF_1(
43      // Inputs
44      .clk(clk),
```

```
45      .reset(reset),
46      .Branch(JumpFlag[0]),
47      .Jump(JumpFlag[1]),
48      .IFWrite(IFWrite),
49      .JumpAddr(JumpAddr),
50      // Outputs
51      .Instruction_if(Instruction_if),
52      .PC(PC),
53      .IF_flush(IF_flush)
54      );
55
56      IFID IF_ID_1(
57      // Inputs
58      .clk(clk),
59      .en(IFWrite),
60      .reset(IF_flush|reset),
61      .PC_if(PC),
62      .Instruction_if(Instruction_if),
63      // Outputs
64      .PC_id(PC_id),
65      .Instruction_id(Instruction_id)
66      );
67
68
69      ID ID_1(
70      // Inputs
71      .clk(clk),
72      .Instruction_id(Instruction_id),
73      .PC_id(PC_id),
74      .RegWrite_wb(RegWrite_wb),
75      .rdAddr_wb(rdAddr_wb),
76      .RegWriteData_wb(RegWriteData_wb),
77      .MemRead_ex(MemRead_ex),
78      .rdAddr_ex(rdAddr_ex),
79      // Outputs
80      .MemtoReg_id(MemtoReg_id),
81      .RegWrite_id(RegWrite_id),
82      .MemWrite_id(MemWrite_id),
83      .MemRead_id(MemRead_id),
84      .ALUCode_id(ALUCode_id),
85      .ALUSrcA_id(ALUSrcA_id),
86      .ALUSrcB_id(ALUSrcB_id),
87      .Stall(Stall),
88      .Branch(JumpFlag[0]),
89      .Jump(JumpFlag[1]),
90      .IFWrite(IFWrite),
91      .JumpAddr(JumpAddr),
92      .Imm_id(Imm_id),
93      .rs1Data_id(rs1Data_id),
94      .rs2Data_id(rs2Data_id),
95      .rs1Addr_id(rs1Addr_id),
96      .rs2Addr_id(rs2Addr_id),
97      .rdAddr_id(rdAddr_id)
98      );
99
100     IDEX ID_EX_1(
101     // Inputs
102     .clk(clk),
103     .reset(Stall|reset),
104     .MemtoReg_id(MemtoReg_id),
105     .RegWrite_id(RegWrite_id),
106     .MemWrite_id(MemWrite_id),
107     .MemRead_id(MemRead_id),
108     .ALUCode_id(ALUCode_id),
109     .ALUSrcA_id(ALUSrcA_id),
110     .ALUSrcB_id(ALUSrcB_id),
111     .PC_id(PC_id),
```

```
112         .Imm_id(Imm_id),
113         .rdAddr_id(rdAddr_id),
114         .rs1Addr_id(rs1Addr_id),
115         .rs2Addr_id(rs2Addr_id),
116         .rs1Data_id(rs1Data_id),
117         .rs2Data_id(rs2Data_id),
118         // Outputs
119         .MemtoReg_ex(MemtoReg_ex),
120         .RegWrite_ex(RegWrite_ex),
121         .MemWrite_ex(MemWrite_ex),
122         .MemRead_ex(MemRead_ex),
123         .ALUCode_ex(ALUCode_ex),
124         .ALUSrcA_ex(ALUSrcA_ex),
125         .ALUSrcB_ex(ALUSrcB_ex),
126         .PC_ex(PC_ex),
127         .Imm_ex(Imm_ex),
128         .rdAddr_ex(rdAddr_ex),
129         .rs1Addr_ex(rs1Addr_ex),
130         .rs2Addr_ex(rs2Addr_ex),
131         .rs1Data_ex(rs1Data_ex),
132         .rs2Data_ex(rs2Data_ex)
133         );
134 
135         EX EX_1(
136         // Inputs
137         .ALUCode_ex(ALUCode_ex),
138         .ALUSrcA_ex(ALUSrcA_ex),
139         .ALUSrcB_ex(ALUSrcB_ex),
140         .Imm_ex(Imm_ex),
141         .rs1Addr_ex(rs1Addr_ex),
142         .rs2Addr_ex(rs2Addr_ex),
143         .rs1Data_ex(rs1Data_ex),
144         .rs2Data_ex(rs2Data_ex),
145         .PC_ex(PC_ex),
146         .RegWriteData_wb(RegWriteData_wb),
147         .ALUResult_mem(ALUResult_mem),
148         .rdAddr_mem(rdAddr_mem),
149         .rdAddr_wb(rdAddr_wb),
150         .RegWrite_mem(RegWrite_mem),
151         .RegWrite_wb(RegWrite_wb),
152         // Outputs
153         .ALUResult_ex(ALUResult_ex),
154         .MemWriteData_ex(MemWriteData_ex),
155         .ALU_A(ALU_A),
156         .ALU_B(ALU_B)
157         );
158 
159         EXMEM EX_MEM_1(
160         // Inputs
161         .clk(clk),
162         .MemtoReg_ex(MemtoReg_ex),
163         .RegWrite_ex(RegWrite_ex),
164         .MemWrite_ex(MemWrite_ex),
165         .ALUResult_ex(ALUResult_ex),
166         .MemWriteData_ex(MemWriteData_ex),
167         .rdAddr_ex(rdAddr_ex),
168         // Outputs
169         .MemtoReg_mem(MemtoReg_mem),
170         .RegWrite_mem(RegWrite_mem),
171         .MemWrite_mem(MemWrite_mem),
172         .ALUResult_mem(ALUResult_mem),
173         .MemWriteData_mem(MemWriteData_mem),
174         .rdAddr_mem(rdAddr_mem)
175         );
176 
177         DataRam DataRAM_1(
178         // Inputs
```

```
179       .a(ALUResult_mem[7:2]),
180       .d(MemWriteData_mem),
181       .clk(clk),
182       .we(MemWrite_mem),
183       // Outputs
184       .spo(MemDout_mem)
185       );
186
187
188       MEMWB MEM_WB_1(
189       // Inputs
190       .clk(clk),
191       .MemtoReg_mem(MemtoReg_mem),
192       .RegWrite_mem(RegWrite_mem),
193       .MemDout_mem(MemDout_mem),
194       .ALUResult_mem(ALUResult_mem),
195       .rdAddr_mem(rdAddr_mem),
196       // Outputs
197       .MemtoReg_wb(MemtoReg_wb),
198       .RegWrite_wb(RegWrite_wb),
199       .MemDout_wb(MemDout_wb),
200       .ALUResult_wb(ALUResult_wb),
201       .rdAddr_wb(rdAddr_wb)
202       );
203
204       assign RegWriteData_wb =MemtoReg_wb?MemDout_wb:ALUResult_wb;
205
206
207       endmodule
```

五、 仿真与结果分析

1. ALU 仿真结果

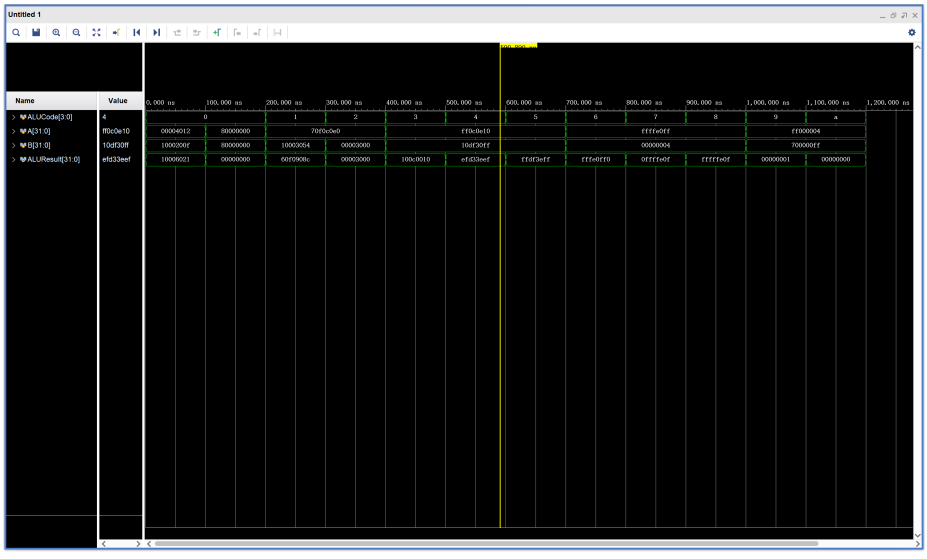

Figure 19: ALU 仿真结果

同预期结果进行对比，可以认定 ALU 仿真结果正确。

2. Decode 模块仿真结果

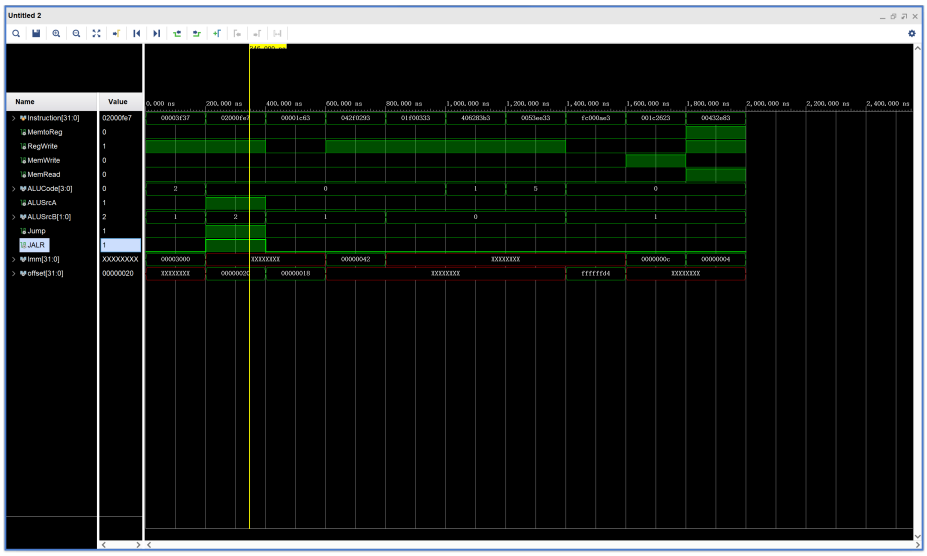

Figure 20: Decode 仿真结果

同预期结果进行对比，可以认定 Decode 仿真结果正确。

3. IF 模块仿真结果

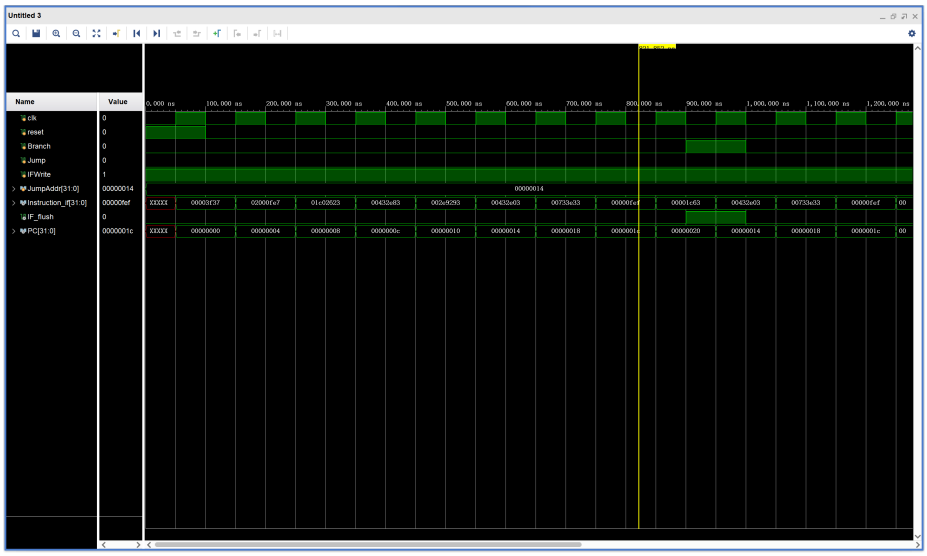

Figure 21: IF 仿真结果 1

同预期结果进行对比，可以认定 IF 模块仿真结果正确。

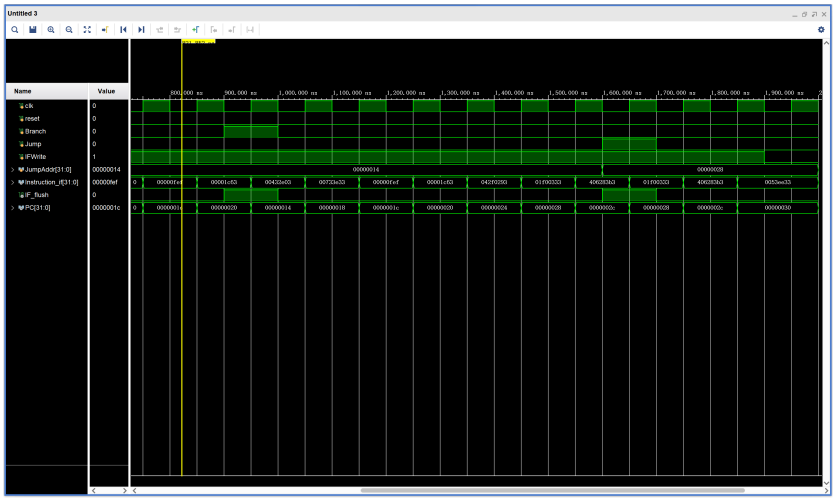

Figure 22: IF 仿真结果 2

4. CPU 顶层模块仿真结果

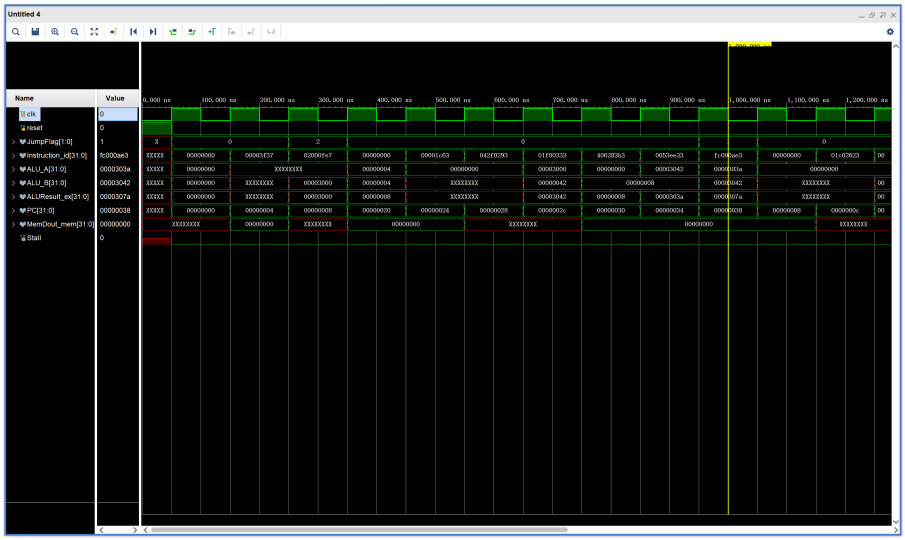

Figure 23: CPU 顶层仿真 1

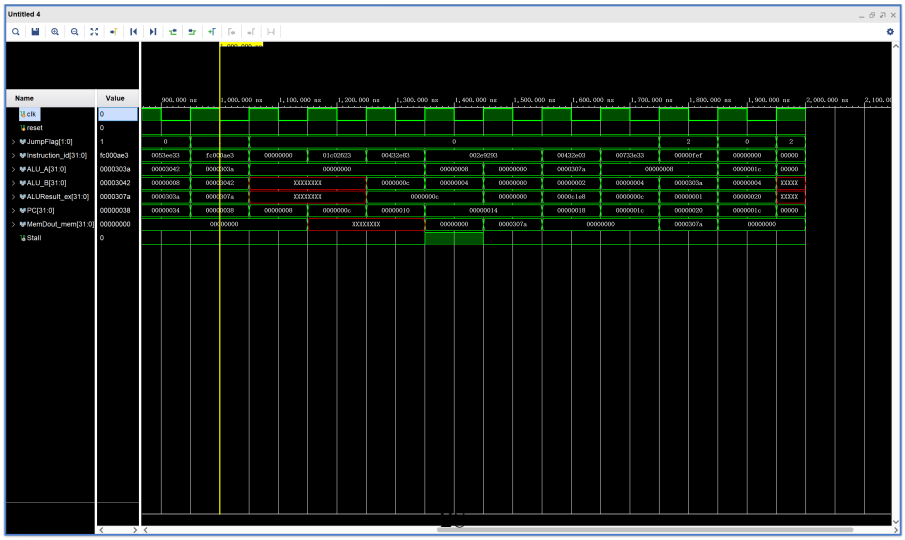

Figure 24: CPU 顶层仿真 2

通过仿真结果，可以看出 CPU 能够完成设计指令，并有异常（exception）、自陷（trap）、中断（interrupt）等处理方案。

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

例子中的两条指令发生了数据冲突，两条指令都对 x28 寄存器发起了访问请求，且后一条指令依赖于前一条指令的值。对于这一问题，一般通过数据旁路、暂停等待或分时访问的方式解决。所以在指令间添加空指令就可以解决该问题，在硬件层面，可以添加检测模块，在发生流水线冲突时，跳转进入空指令，在值写入 memory 后进行下一指令。

很多 CPU 架构并不解决这个问题，可能是因为跳转进入空指令也会浪费很多时间，例子中需要多花费 3 个空指令的时间，且额外的检测模块会占用不少的资源，相较之下并不划算。而直接使这两条指令顺序执行，也可以使指令正常运行，且花费的时间相差不大，所以选择不去进行优化。

七、 遇到的问题及解决方案

在 CPU 的设计中，我遇到了不少的问题。

首先，在开始时，由于模块较多，且一个顶层模块下会包含多个单元电路，导致不知道从哪里入手，后在老师的提示之下，从简单的 ALU 设计中开始入手，逐渐熟练，一层一层的完成设计。

之后对于 Decode 的设计中，ALUcode 的逻辑较为复杂，根据功能表得到的逻辑仿真出来出现了错误，之后经过反复的查验，通过同时使用 if-else 和 case 语句，而不是一味地使用 if-else 进行条件地判别，逐步理清逻辑从而得到正确地结果。

最后，在对 CPU 完成了顶层设计后，忽略了顶层仿真库地填入，导致仿真虽然能够通过，但是始终无法生成烧录板子地二进制流。在老师的提示和指导之下，重新搭建仿真平台，并将缺失的仿真文件添加至正确的位置之后，方才成功烧录入板子完成上板验证。