浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	黄嘉欣
学　　院：	信息与电子工程学院
系：	信息与电子工程学系
专　　业：	信息工程
学　　号：	3190102060
指导教师：	屈民军、唐奕

2021 年 12 月 5 日

专业：信息工程
姓名：黄嘉欣
学号：3190102060
日期：2021 年 12 月 5 日
地点：教 11-400

浙江大学实验报告

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

一、实验目的

① 熟悉 RISC-V 指令系统；

② 了解提高 CPU 性能的方法；

③ 掌握流水线 RISC-V 微处理器的工作原理；

④ 理解数据冒险、控制冒险的概念以及流水线冲突的解决方法；

⑤ 掌握流水线 RISC-V 微处理器的测试方法；

⑥ 了解用软件实现数字系统的方法。

二、实验任务与要求

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、实验原理与代码设计

① 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，其流水线流水作业示意图如图所示：

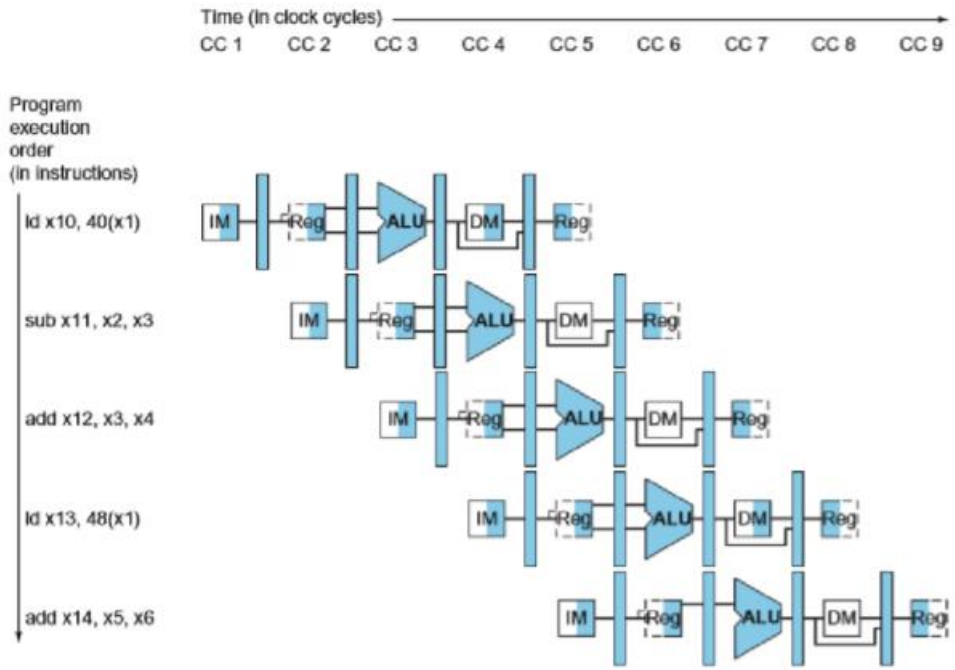


图 3.1.1 流水线流水作业示意图

图 3.1.2 所示为符合设计要求的流水线 RISC-V 微处理器的原理框图，采用五级流水线。其具体设计如下所述。

1) 流水线中的控制信号

a) 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：

- 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：当 IFWrite=0 时，阻塞 IF/ID 流水线，同时暂停读取下一条指令。
- IF_flush：当 IF_flush=1 时，清空 IF/ID 寄存器。

b) 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

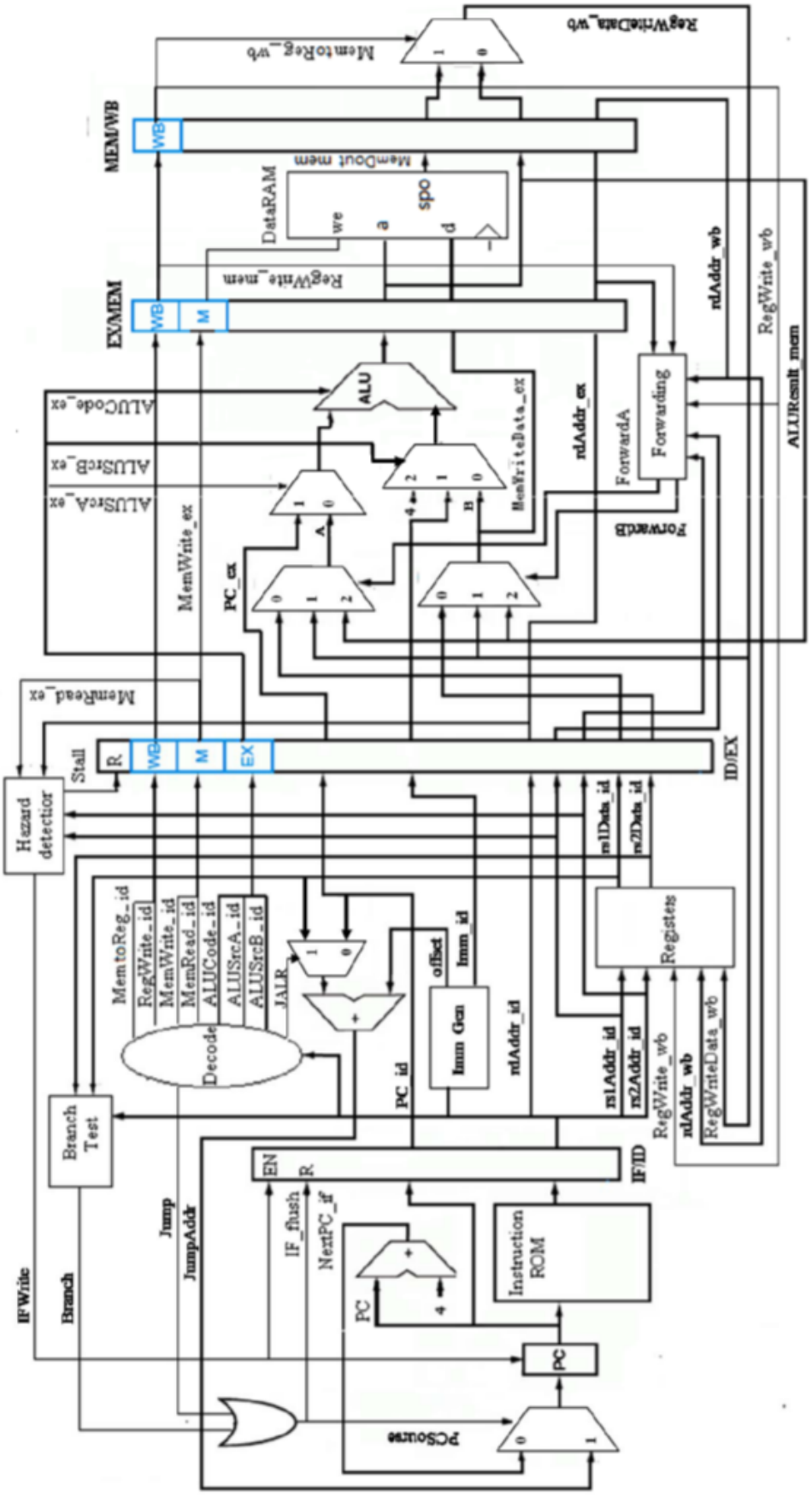


图 3.1.2 五级流水线 RISC-V 微处理器原理框图

的控制信号基本都是在 ID 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。

c) 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。

d) 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。

e) 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，回写数据来自于 ALU 运算结果；而当 MemtoReg=1 时，回写数据来自于存储器。

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。在设计当中，我们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。

a) 一阶数据相关与转发（EX 冒险）

如图 3.1.3 所示，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。此时，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 的一阶数据相关成立的条件为：

- MEM 级阶段必须是写操作（RegWrite_mem=1）；
- 目标寄存器不是 X0 寄存器（rdAddr_mem≠0）；
- 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：

- MEM 级阶段必须是写操作（RegWrite_mem=1）；
- 目标寄存器不是 X0 寄存器（rdAddr_mem≠0）；
- 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，如图 3.2.1 所示。

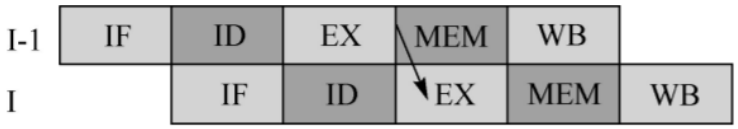


图 3.1.3 一阶前推网络示意图

b) 二阶数据相关与转发（MEM 冒险）

如图 3.1.4 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：

- WB 级阶段必须是写操作（RegWrite_wb=1）；
- 目标寄存器不是 X0 寄存器（rdAddr_wb≠0）；
- 一阶数据相关条件不成立（rdAddr_mem≠rs1Addr_ex）；
- 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：

- WB 级阶段必须是写操作（RegWrite_wb=1）；
- 目标寄存器不是 X0 寄存器（rdAddr_wb≠0）；
- 一阶数据相关条件不成立（rdAddr_mem≠rs2Addr_ex）；
- 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，如图 3.2.2 所示。

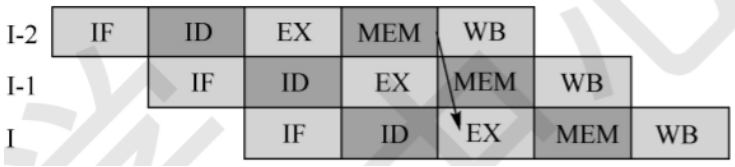


图 3.1.4 二阶前推网络示意图

c) 三阶数据相关

如图 3.1.5 所示，当在同一个周期内同时读写同一个寄存器时，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：

- 寄存器必须是写操作（RegWrite_wb =1）；
- 目标寄存器不是 X0 寄存器（rdAddr_wb≠0）；
- 读写同一个寄存器（rdAddr_wb =rs1Addr_id）；

同样，导致操作数 B 的三阶数据相关必须满足下列条件：

- 寄存器必须是写操作（RegWrite_wb =1）；
- 目标寄存器不是 X0 寄存器（rdAddr_wb≠0）；
- 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，即同一个周期内对同一个寄存器进行读、写操作时，要求读出的值为新写入的数据。

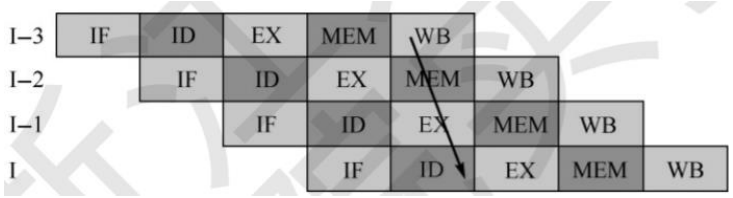


图 3.1.5 三阶前推网络示意图

3) 数据冒险与数据转发

如前分析可知，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：

a) 上一条指令必须是 lw 指令（MemRead_ex=1）；

b) 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 信号将清空 ID/EX 寄存器，另外一个输出低电平有效的 IFWrite 信号将阻塞流水线 ID 级、IF 级，即插入一个流水线气泡。

② 流水线 RISC-V 微处理器的设计

根据流水线不同阶段，将系统划分为 IF、ID、EX 和 MEM 四大模块，WB 部分功能电路非常简单，可直接在顶层文件中设计。另外，系统还包含 IF/ID、ID/EX、EX/MEM、MEM/WB 四个流水线寄存器。

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。其主要由指令译码（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 3.2.1 所示。

表 3.2.1 ID 模块的输入/输出引脚说明

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

a) 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 offet。其为一个组合电路。RISC-V 将指令分为 R、I、S、SB、U、UJ 等六类。从电路设计角度看，我们可以根据操作数的来源和立即数构成方式不同，再次细分指令，即：

- R_type 类：操作码为 7'h33，R 类的所有指令，两操作数分别为 rs1 和 rs2；
- I_type 类：操作码 7'h13，I 类的算术逻辑运算指令和移位指令，两操作数分别为

rs1 和立即数 imm；

- LW 指令：操作码 7'h03，I 类的数据传送指令 lw，两操作数分别为 rs1 和立即数 imm；
- JALR 指令：操作码 7'h67，I 类的无条件分支指令 jalr，两操作数分别为 PC 和 4；
- SW 指令：操作码 7'h23，S 类的数据传送指令 sw，两操作数分别为 rs1 和立即数 imm；
- SB_type 类：操作码 7'h63，SB 类的所有指令，两操作数分别为 PC 和立即数 imm；
- LUI 指令：操作码 7'h37，U 类的数据传送指令 lui，只有一个操作数（立即数 imm）；
- AUIPC 指令：操作码 7'h17，U 类的数据传送指令 auipc，两个操作数分别为 PC 和立即数 imm；
- JAL 指令：操作码 7'h6F，UJ 类的无条件分支指令 jal，两个操作数分别为 PC 和 4。

因此，可设置 R_type、I_type、SB_type、LW、JALR、SW、LUI、AUIPC 和 JAL 等变量来表示指令类型，在已知各类型操作码的情况下，可设计代码如下：

```
wire [6:0] op; // 操作码
assign op = Instruction[6:0];
wire R_type, I_type, LW, JALR, SW, SB_type, LUI, AUIPC, JAL;
assign R_type = (op==R_type_op);
assign I_type = (op==I_type_op);
assign LW = (op==LW_op);
assign JALR = (op==JALR_op);
assign SW = (op==SW_op);
assign SB_type = (op==SB_type_op);
assign LUI = (op==LUI_op);
assign AUIPC =(op==AUIPC_op);
assign JAL = (op==JAL_op);
```

对输出信号，可根据指令操作的要求进行确定，其代码设计如下。

i) 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;
```

ii) 只有 SW 指令会对存储器写数据，所以有：

```
assign MemWrite = SW;
```

iii) 需要回写的指令类型有 R_type、I_type、LW、JALR、LUI、AUIPC 和 JAL，所以有：

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

iv) 只有 JALR 和 JAL 两条无条件分支指令，所以有：

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

v) 操作数 A 和 B 的选择信号的确定

如表 3.2.2，为操作数选择功能表。

表 3.2.2 操作数选择功能表

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

从而可得 ALUSrcA_id 和 ALUSrcB_id[1:0]表达式为：

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

vi) ALUCode 的确定

除了条件分支指令，其它指令都需要 ALU 执行运算，因为共有 11 种不同运算，故 ALUCode 信号需用 4 位二进制表示。将加法运算设为默认算法，其功能表如表 3.2.3 所示。注意：表中 funct7[6]与 funct6[5]在指令中为 instruction[30]。

表 3.2.3 ALUcode 功能表

R_type	I_type	LUI	funct3	funct7[6] (funct6[5])	ALUCode	备注
1	0	0	3'o0	0	4'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

采用分支语句，可完成 ALUcode 的赋值，代码如下：

```
wire           funct6_7;
wire [2:0]     funct3;
assign funct6_7 = Instruction[30];
assign funct3 = Instruction[14:12];
// 确定 ALUCode
always @ (*)
begin
 if (LUI) ALUCode = alu_lui; // 送数
 else if (R_type&&~I_type) // R_type == 1, I_type == 0
 begin
  case (funct3)
     3'b000: begin
                if (~funct6_7) ALUCode = alu_add;  // 加
                else           ALUCode = alu_sub;  // 减
             end
     SLL_funct3:  if (~funct6_7) ALUCode = alu_sll;  // 左移
     SLT_funct3:  if (~funct6_7) ALUCode = alu_slt;  // 跳转
     SLTU_funct3: if (~funct6_7) ALUCode = alu_sltu; // 跳转/U
     XOR_funct3:  if (~funct6_7) ALUCode = alu_xor;  // 异或
     3'b101: begin
                if (~funct6_7) ALUCode = alu_srl;  // 逻辑右移
                else           ALUCode = alu_sra;  // 算术右移
             end
     OR_funct3:   if (~funct6_7) ALUCode = alu_or;   // 或
     AND_funct3:  if (~funct6_7) ALUCode = alu_and;  // 与
     default: ALUCode = alu_add; // 默认为加
  endcase
 end
 else if (~R_type&&I_type) // R_type == 0, I_type == 1
 begin
  case (funct3)
     ADDI_funct3:  ALUCode = alu_add;  // 加
     SLLI_funct3:  ALUCode = alu_sll;  // 左移
     SLTI_funct3:  ALUCode = alu_slt;  // 跳转
     SLTIU_funct3: ALUCode = alu_sltu; // 跳转(无符号数)
     XORI_funct3:  ALUCode = alu_xor;  // 异或
     ORI_funct3:   ALUCode = alu_or;   // 或
     ANDI_funct3:  ALUCode = alu_and;  // 与
     3'b101: begin
```

```
                    if (~funct6_7) ALUCode = alu_srl;  // 逻辑右移
                    else           ALUCode = alu_sra;  // 算术右移
                end
            default: ALUCode = alu_add; // 默认为加
        endcase
    end
    else ALUCode = alu_add; // 加
end
```

vii) 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，可再设定一个变量 Shift 来区分两者。Shift=1 表示移位运算，否则为算术逻辑运算。Shift 值的确定代码为：

```
wire Shift; // 区分 I-type 的算术逻辑运算与移位运算
// 1：移位运算；0：算术逻辑运算
assign Shift = (funct3==1)||(funct3==5);
```

立即数构成和扩展方法如表 3.2.4 所示，表中的 inst 即为 instruction。

表 3.2.4 立即数产生方法

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

采用分支语句，可完成 Imm 和 offset 的赋值，代码如下（表中-用 32'bx 实现）：

```
always @ (*)
begin
   if (I_type)
   begin
     if (Shift)       begin Imm = {26'd0, Instruction[25:20]};
offset = 32'dx; end
     else             begin Imm = {{20{Instruction[31]}},
Instruction[31:20]}; offset = 32'dx; end
   end
```

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

b)　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 3.2.1 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。

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 3.2.1 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。

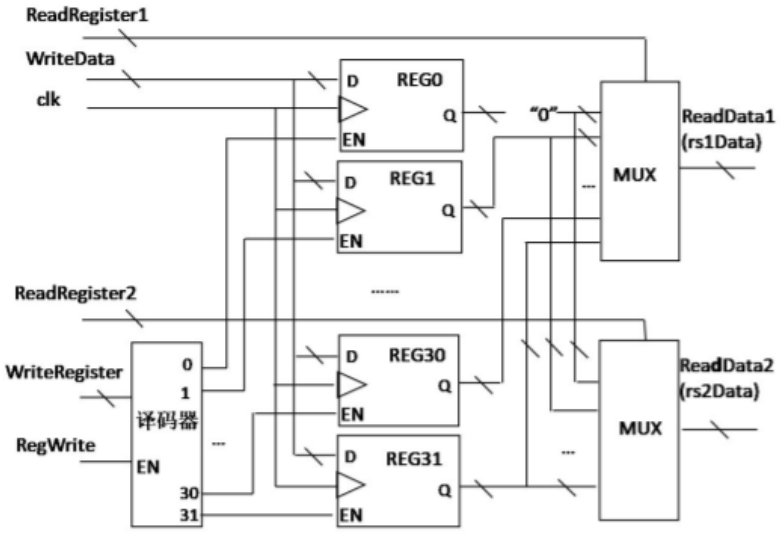


图 3.2.1 寄存器堆原理框图

寄存器堆的设计代码为：

```
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)
begin
    if (RegWrite) regs[WriteRegister] <= WriteData;
end
```

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，其原理图如图 3.2.2 所示。

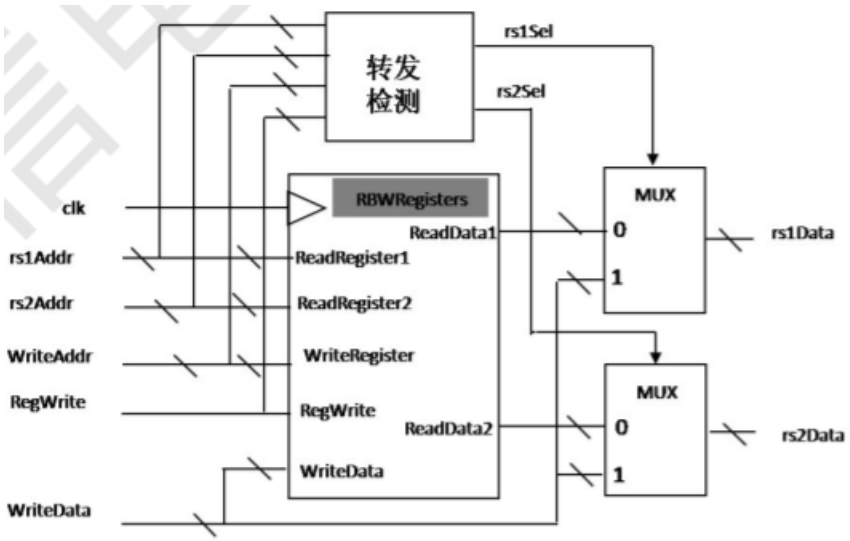


图 3.2.2 具有 Read After Write 特性寄存器堆的原理框图

图中转发检测电路的输出代码设计为：

```
wire rs1Sel, rs2Sel; // 转发检测输出
assign rs1Sel=RegWrite&&(WriteAddr!=0)&&(WriteAddr==rs1Addr);
assign rs2Sel=RegWrite&&(WriteAddr!=0)&&(WriteAddr==rs2Addr);
```

将其与图 3.2.1 所示的 RBW 寄存器堆结合，即可得寄存器堆子模块的设计代码：

```
module Registers(clk, rs1Addr, rs2Addr, WriteAddr, RegWrite,
                 WriteData, rs1Data, rs2Data);
   input clk, RegWrite;
   input [4:0] rs1Addr, rs2Addr, WriteAddr; // 寄存器号
   input [31:0] WriteData; // 数据
   output [31:0] rs1Data, rs2Data;

   wire rs1Sel, rs2Sel; // 转发检测输出
   assign rs1Sel=RegWrite&&(WriteAddr!=0)&&(WriteAddr==rs1Addr);
```

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

    wire [31:0] ReadData1, ReadData2;
    // Read Before Write 寄存器堆
    RBWRegisters RBWReg(.clk(clk), .ReadRegister1(rs1Addr),
                        .ReadRegister2(rs2Addr),
                        .WriteData(WriteData),
                        .WriteRegister(WriteAddr),
                        .RegWrite(RegWrite),
                        .ReadData1(ReadData1),
                        .ReadData2(ReadData2));
    // 选择输出
    mux2 mux_1(.in0(ReadData1), .in1(WriteData), .addr(rs1Sel),
               .out(rs1Data));
    mux2 mux_2(.in0(ReadData2), .in1(WriteData), .addr(rs2Sel),
               .out(rs2Data));
endmodule
```

c) 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 32 位加法器来实现：

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

ii) 确定比较运算的结果。对比较运算，若最高位不同，即 rs1Data[31]≠rs2Data[31]，可根据 rs1Data[31]、rs2Data[31]决定比较结果，但应注意符号数、无符号数的最高位 rs1Data[31]、rs2Data[31]代表意义不同。若两数最高位相同，则两数之差不会溢出，所以比较运算结果可由两个操作数之差的符号位 sum[31]决定。

在符号数比较运算中，若 rs1Data 为负数、rs2Data 为 0 或正数，即 rs1Data[31] &&(~rs2Data[31])；或 rs1Data、rs2Data 符号相同，sum 为负，即(rs1Data[31] ~^rs2Data[31])&&sum[31]，则可得 rs1Data<rs2Data。同理，在无符号数比较运算中，若 rs1Data 最高位为 0、rs2Data 最高位为 1，即(~rs1Data[31])&&rs2Data[31]；或 rs1Data、rs2Data 最高位相同，sum 为负，即(rs1Data[31]~^rs2Data[31])&& sum[31]，则可得 rs1Data<rs2Data。最后根据指令类型和功能码 funct3，利用数据选择器即可完成分支检测。

综上，分支检测电路的设计代码为：

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

    parameter SB_type_op = 7'b1100011; // 7'h63;
    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 [6:0] op; // 操作码
    wire [2:0] funct3; // 功能码
    wire [31:0] sum; // 加法结果
    wire SB_type;
    wire isLT, isLTU; // 比较结果
    assign op = Instruction[6:0];
    assign funct3 = Instruction[14:12];
    assign SB_type = (op==SB_type_op);

    // sum = rs1Data + ~rs2Data + 1
    adder_32bits adder(.a(rs1Data), .b(~rs2Data), .ci(1),
                       .s(sum), .co());

    // 确定比较运算的结果
    assign isLT =
rs1Data[31]&&(~rs2Data)||(rs1Data[31]~^rs2Data[31])&&sum[31]; //
符号数
    assign isLTU =
(~rs1Data[31])&&rs2Data[31]||(rs1Data[31]~^rs2Data[31])&&sum[31];
// 非符号数

    // 数据选择器完成分支检果
    always @ (*)
    begin
        if (SB_type)
        begin
            case (funct3)
                beq_funct3: Branch = ~(|sum[31:0]); // beq
                bne_funct3: Branch = |sum[31:0];    // bne
                blt_funct3: Branch = isLT;          // blt
                bge_funct3: Branch = ~isLT;         // bge
                bltu_funct3: Branch = isLTU;        // bltu
```

```
                bgeu_funct3: Branch = ~isLTU;       // bgeu
                default: Branch = 0;
            endcase
        end
        else Branch = 0;
    end
endmodule
```

d) 冒险检测功能电路（Hazard Detector）的设计

由于冒险检测功能电路比较简单，故可直接在 ID 顶层描述。经分析，冒险成立的条件为：

i) 上一条指令必须是 lw 指令（MemRead_ex=1）；

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

当冒险成立时，应清空 ID/EX 寄存器并且阻塞流水线 ID 级、IF 级流水线，则有：

```
// 冒险检测
assign Stall = ((rdAddr_ex==rs1Addr_id)||(rdAddr_ex==rs2Addr_id))
               &&MemRead_ex;
assign IFWrite = ~Stall;
```

e) ID 模块核心代码设计

综上，根据各子部分的设计代码和 ID 模块结构，可以得到 ID 模块的核心代码为：

```
wire JALR; // 译码输出
wire [31:0] JALRAddr; // 数据选择器输出
wire [31:0] offset; // 立即数产生电路

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

// 指令译码
Decode Decode(.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),
              .Instruction(Instruction_id));

// 寄存器堆
Registers Registers(.clk(clk), .rs1Addr(rs1Addr_id),
```

```
                    .rs2Addr(rs2Addr_id), .WriteAddr(rdAddr_wb),
                    .RegWrite(RegWrite_wb),
                    .WriteData(RegWriteData_wb),
                    .rs1Data(rs1Data_id), .rs2Data(rs2Data_id));

    // 分支检测
    BranchTest BranchTest(.Instruction(Instruction_id),
                          .rs1Data(rs1Data_id), .rs2Data(rs2Data_id),
                          .Branch(Branch));

    // 分支地址
    mux2 mux_1(.in0(PC_id), .in1(rs1Data_id), .addr(JALR),
               .out(JALRAddr));
    adder_32bits adder_1(.a(JALRAddr), .b(offset), .ci(0),
                         .s(JumpAddr), .co());

    // 冒险检测
    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。其接口信息如表 3.2.5 所示。

表 3.2.5 EX 模块的输入/输出引脚说明

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

a) ALU 子模块的设计

算术逻辑运算单元（ALU）提供 CPU 的基本运算能力，如加、减、与、或、比较、移位等。具体而言，ALU 输入为两个操作数 A、B 和控制信号 ALUCode，由控制信号 ALUCode 决定采用何种运算，运算结果为 ALUResult。整理表 3.2.3 所示的 ALUCode 功能表，可得到 ALU 功能表，如表 3.2.6 所示。

表 3.2.6 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

由于 ALU 需执行多种运算，为了提高运算速度，可同时进行各种运算，再根据 ALUCode 信号选出所需结果。ALU 的基本结构如图 3.2.3 所示。

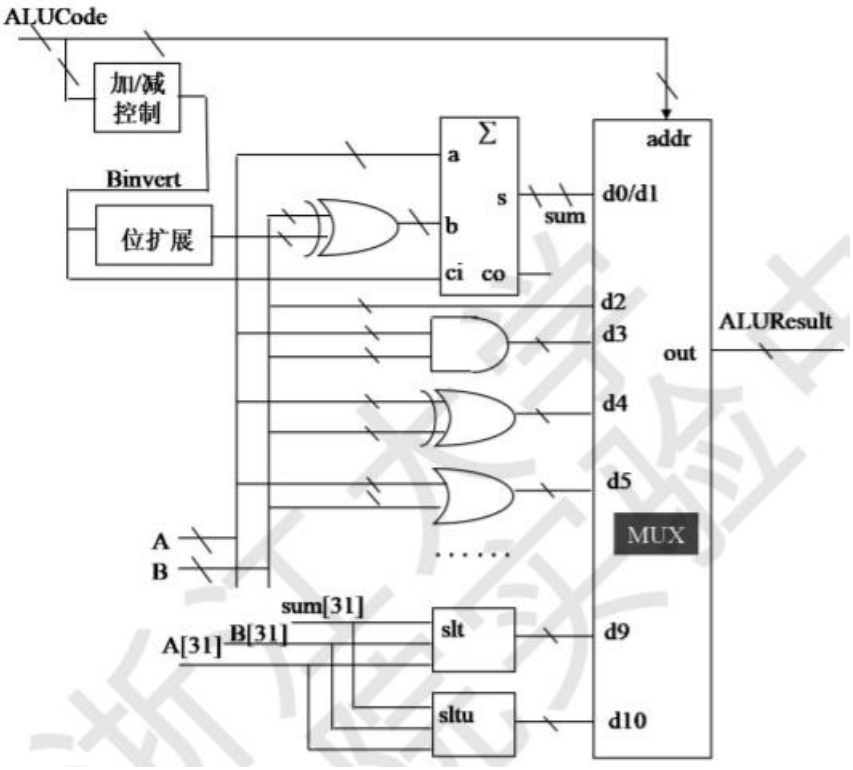


图 3.2.3 ALU 结构图

i) 加、减电路的设计考虑

减法、比较均可用加法器和必要辅助电路来实现。图 3.2.3 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，故有代码：

```
wire Binvert; // 加减运算控制信号
wire [31:0] sum;
// 加减控制
assign Binvert = ~(ALUCode==0);
adder_32bits adder(.a(A), .b(B^{32{Binvert}}), .ci(Binvert),
                   .s(sum), .co());
```

ii) 比较电路的设计考虑

参考分支检测电路的设计，可确定 slt 和 sltu 两条件的比较结果：

```
// 比较控制
assign isLT = A[31]&&(~B)||(A[31]~^B[31])&&sum[31]; // 符号数
assign isLTU = (~A[31])&&B[31]||(A[31]~^B[31])&&sum[31];
// 非符号数
```

iii) 算术右移运算电路的设计考虑

由于在 Verilog HDL 中，实现算术右移的对象须定义是 reg 类型，但在 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 reg，再对 reg 进行算术右移操作。相应的代码设计为：

```
reg signed [31:0] A_reg; // 算术右移中间变量
always @ (*) begin A_reg = A; end
```

iv) ALU 模块核心代码设计

综上，根据各子部分的设计代码和 ALU 功能表，采用分支判断，可以得到 ALU 模块的核心代码为：

```
wire Binvert; // 加减运算控制信号
wire [31:0] sum;
wire isLT, isLTU; // 比较电路
reg [31:0] ALUResult;
reg signed [31:0] A_reg; // 算术右移中间变量

// 加减控制
assign Binvert = ~(ALUCode==0);
adder_32bits adder(.a(A), .b(B^{32{Binvert}}), .ci(Binvert),
```

```
                .s(sum), .co());

    // 比较控制
    assign isLT = A[31]&&(~B)||(A[31]~^B[31])&&sum[31]; // 符号数
    assign isLTU=(~A[31])&&B[31]||(A[31]~^B[31])&&sum[31];
    //非符号数

    // 基本运算
    always @ (*)
    begin
        A_reg = A;
        case (ALUCode)
            alu_add: ALUResult = sum;                   // add
            alu_sub: ALUResult = sum;                   // sub
            alu_lui: ALUResult = B;                     // lui
            alu_and: ALUResult = A&B;                   // and
            alu_xor: ALUResult = A^B;                   // xor
            alu_or:  ALUResult = A|B;                   // or
            alu_sll: ALUResult = A<<B;                  // sll
            alu_srl: ALUResult = A>>B;                  // srl
            alu_sra: ALUResult = A_reg>>>B;             // sra
            alu_slt: ALUResult = isLT?32'd1:32'd0;      // slt
            alu_sltu:ALUResult = isLTU?32'd1:32'd0;     // stlu
            default: ALUResult = sum;
        endcase
    end
```

b) 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 3.2.7 所示。

表 3.2.7 前推电路输出信号的含义

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

由一、二阶数据相关判断条件，可以得到 ForwardA 与 ForwardB 的赋值代码为：

```
wire [1:0] ForwardA, ForwardB; // 数据前推电路
// 数据选择器地址信号
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);
```

c) EX 模块核心代码设计

综上，根据各子部分设计代码和 EX 模块结构，可以得到 EX 模块的核心代码为：

```
wire [1:0] ForwardA, ForwardB; // 数据前推电路
wire [31:0] A, B;

// 数据选择器地址信号
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);

// 数据选择
mux3 mux_0(.in0(rs1Data_ex), .in1(RegWriteData_wb),
           .in2(ALUResult_mem), .addr(ForwardA), .out(A));
mux3 mux_1(.in0(rs2Data_ex), .in1(RegWriteData_wb),
           .in2(ALUResult_mem), .addr(ForwardB), .out(B));
mux2 mux_2(.in0(A), .in1(PC_ex), .addr(ALUSrcA_ex), .out(ALU_A));
mux3 mux_3(.in0(B), .in1(Imm_ex), .in2(32'd4), .addr(ALUSrcB_ex),
           .out(ALU_B));

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

assign MemWriteData_ex = B; // 存储器的回写数据
```

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 64×32 bi 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，其接口信息如表 3.2.8 所示。

表 3.2.8 IF 模块的输入/输出引脚说明

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

由于指令存储器为组合存储器，故可用 Verilog HDL 设计一个查找表阵列 ROM，该 ROM 容量为 64×32bit。根据功能分析，当分支跳转成立时，流水线需要清空，故有：

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

指令指针寄存器中存放有指令地址，其将根据重置信号 reset 和流水线阻塞信号 IF_Write 更改指针值 PC：当 reset=1 时，PC 清 0；当 IFWrite=0 时，流水线阻塞，PC 保持；其余情况 PC 正常传递。确定的 PC 值会被送入 32 位加法器，以求得下条指令地址 NextPC_if。指令指针选择器会根据 IF_flush 信号，即程序是否需要跳转，从跳转地址 JumpAddr 和 NextPC_if 中选取一个存入指令指针寄存器中，由此循环从 Instruction ROM 中读取指令。相关代码设计为：

```
// 确定下条指令地址
always @ (posedge clk)
begin
   if (reset) PC = 0;
   else if (!IFWrite) PC = PC; // 流水线阻塞
   else PC = PC_temp;
end

adder_32bits adder(.a(PC), .b(32'd4), .ci(0), .s(NextPC_if), .co());
mux2 mux(.in0(NextPC_if), .in1(JumpAddr), .addr(IF_flush),
```

```
        .out(PC_temp));

// 取指令机器码，注意 addr 的值
InstructionROM InstructionROM(.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。

a) EX/MEM、MEM/WB 两组流水线寄存器只是普通的 D 型寄存器，只需要在时钟上升沿将变量值从寄存器输入传递到输出；

b) 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 R=1 时，输出的所有信号为 0，否则正常传递变量值。其核心代码为：

```
always @ (posedge clk)
begin
    if (R) // 发生数据冒险，插入气泡
    begin
        MemtoReg_ex = 0; RegWrite_ex = 0;
        MemWrite_ex = 0; MemRead_ex = 0;
        ALUSrcA_ex = 0; ALUCode_ex = 0;
        ALUSrcB_ex = 0; PC_ex = 0;
        Imm_ex = 0; rs1Data_ex = 0;
        rs2Data_ex = 0; rdAddr_ex = 0;
        rs1Addr_ex = 0; rs2Addr_ex = 0;
    end
    else // 正常工作
    begin
        MemtoReg_ex = MemtoReg_id; RegWrite_ex = RegWrite_id;
        MemWrite_ex = MemWrite_id; MemRead_ex = MemRead_id;
        ALUSrcA_ex = ALUSrcA_id; ALUCode_ex = ALUCode_id;
        ALUSrcB_ex = ALUSrcB_id; PC_ex = PC_id;
        Imm_ex = Imm_id; rs1Data_ex = rs1Data_id;
        rs2Data_ex = rs2Data_id; rdAddr_ex = rdAddr_id;
        rs1Addr_ex = rs1Addr_id; rs2Addr_ex = rs2Addr_id;
    end
end
```

c) 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 IF/ID 流水线寄存器。因此，IF/ID 流水线寄存器除同步清零功能外，

还需要具有保持功能（即具有使能 EN 信号输入），其核心代码为：

```
always @ (posedge clk)
begin
    // 跳转或分支成立，清空流水线寄存器
    if (R) begin PC_id = 0; Instruction_id = 0; end
    else if (!EN) // 发生数据冒险，阻塞流水线寄存器
    begin
        PC_id = PC_id; Instruction_id = Instruction_id;
    end
    else // 正常工作
    begin
        PC_id = PC_if; Instruction_id = Instruction_if;
    end
end
```

6) 顶层文件的设计

根据图 3.1.2 所示的原理框图，将各模块连接即可。注意各个模块之间的信号传递以及控制信号的选择。由于信号、端口数量较多，设计代码不便在报告中展示，具体可查看 src/Risc5CPU.v 文件。

四、主要仪器设备

① 装有 Vivado 和 ModelSim SE 软件的计算机；

② Nexys Video 开发板一套；

③ 带有 HDMI 接口的显示器一台。

五、实验内容

① 编写指令译码单元 Decode 模块的 Verilog HDL 代码，并用 ModelSim 软件进行功能仿真；

② 编写寄存器堆 Register 模块的 Verilog HDL 代码；

③ 编写 ID 模块的 Verilog HDL 代码；

④ 编写 ALU 模块的 Verilog HDL 代码，并用 ModelSim 软件进行功能仿真；

⑤ 编写执行单元 EX 模块的 Verilog HDL 代码；

⑥ 编写 IF 模块的 Verilog HDL 代码，并用 ModelSim 软件进行功能仿真；

⑦ 打开 Vivado 文件夹下的 Risc5CPU.xpr 工程，生成符合 CPU 要求的数据存储器 IP 内核；

⑧ 编写 CPU 顶层的 Verilog HDL 代码，并用 ModelSim 进行功能仿真，并对结果进行验证；

⑨ 再次打开 Vivado 文件夹下的 Risc5CPU.xpr 工程，添加流水线 CPU 设计的全部代码，然

后综合、实现和下载至 Nexys Video 开发板；

⑩ 连接带有 HDMI 接口的显示器，进行测试。首先将 SW0 置于低电平，使 RISC-V CPU 工作在“单步”运行模式。复位后，每按一下上边按键，RISC-V CPU 运行一步，记录下显示器上的结果，验证设计是否正确。

六、实验结果与仿真分析

① Decode 模块功能仿真

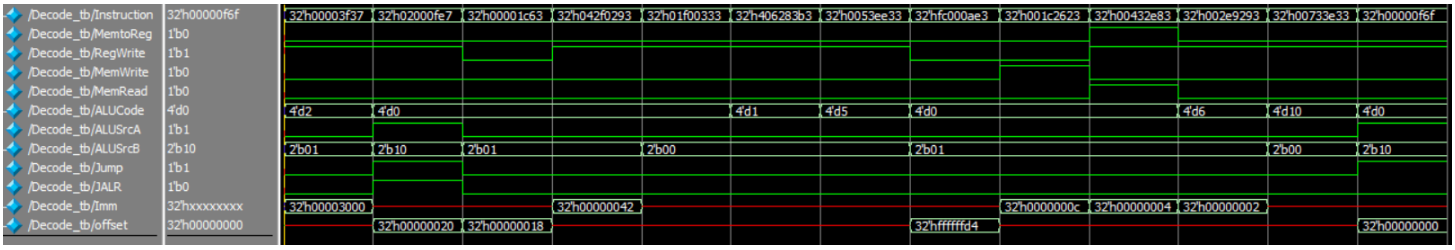


图 6.1.1 Decode 模块完整功能仿真结果

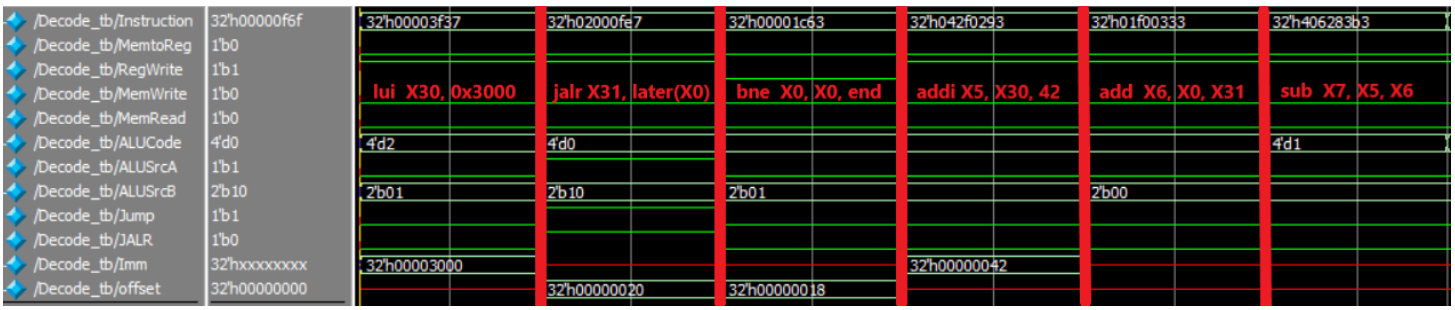


图 6.1.2 Decode 模块前 6 条指令仿真结果

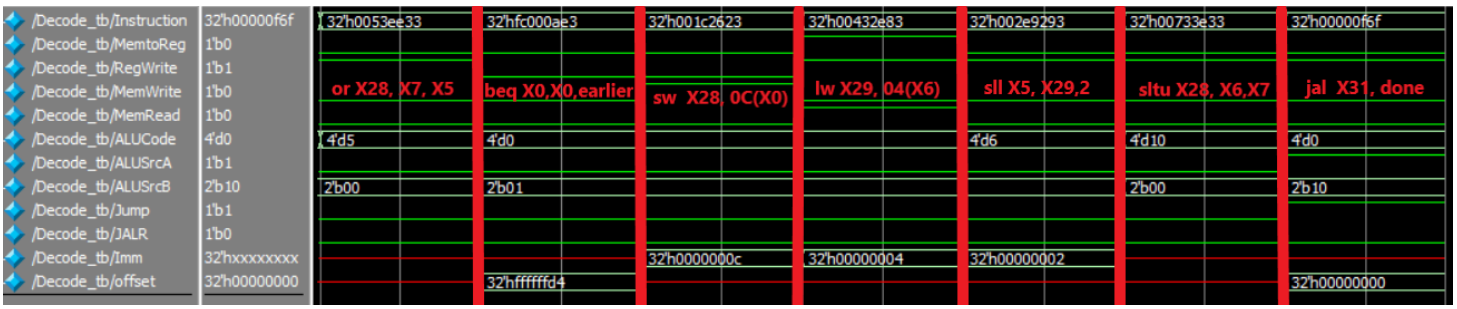


图 6.1.3 Decode 模块后 7 条指令仿真结

分析：根据指令存储器和测试文件，可知各机器码对应指令及其相对位置关系。联系各条指令的功能和实现过程，可得如下理论输出表，其中 ALUCode 可由表 3.2.3 得到：

表 6.1 Decode 模块理论输出表

指令	写寄存器	回写数据来源	写存储器	读存储器	ALUCode	操作数 A 来源	操作数 B 来源	JALR	Jump	立即数 imm	偏移量 offset
lui X30, 0x3000	1	ALU	0	0	4’d2	rs1	imm	0	0	0x3000	——

jalr X31, later(X0)	1	ALU	0	0	4'd0	pc	常数 4	1	1	——	0x0020
bne X0, X0, end	0	——	0	0	4'd0	rs1	imm	0	0	——	0x0018
addi X5, X30, 42	1	ALU	0	0	4'd0	rs1	imm	0	0	0x0042	——
add X6, X0, X31	1	ALU	0	0	4'd0	rs1	rs2	0	0	——	——
sub X7, X5, X6	1	ALU	0	0	4'd1	rs1	rs2	0	0	——	——
or X28, X7, X5	1	ALU	0	0	4'd5	rs1	rs2	0	0	——	——
beq X0, X0, earlier	0	——	0	0	4'd0	rs1	imm	0	0	——	0xfffffffd4
sw X28, 0C(X0)	0	ALU	1	0	4'd0	rs1	imm	0	0	0x000c	——
lw X29, 04(X6)	1	存储器	0	1	4'd0	rs1	imm	0	0	0x0004	——
sll X5, X29, 2	1	ALU	0	0	4'd6	rs1	imm	0	0	0x0002	——
sltu X28, X6, X7	1	ALU	0	0	4'd10	rs1	rs2	0	0	——	——
jal X31, done	1	ALU	0	0	4'd9	rs1	常数 4	0	1	——	0x0000

注：表中数字“1”表示“是”，数字“0”表示“否”；除偏移量 0xfffffffd4 外，其余立即数和偏移量的高 16 位都为 0，表中省略未写。

根据格式约定，可将表中“回写数据来源”、“操作数 A 来源”、“操作数 B 来源”三栏转换为相应数字。由于“写寄存器”、“写存储器”、“读存储器”分别与信号“RegWrite”、“MemWrite”、“MemRead”相对应，可将图中仿真结果与表中理论输出比较，发现两者一致，波形显示正确。故 Decode 模块设计正确。

② ALU 模块功能仿真

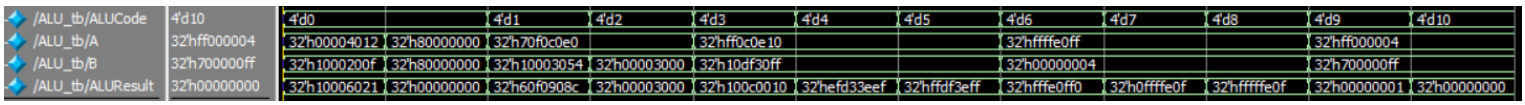


图 6.2.1 ALU 模块完整功能仿真结果

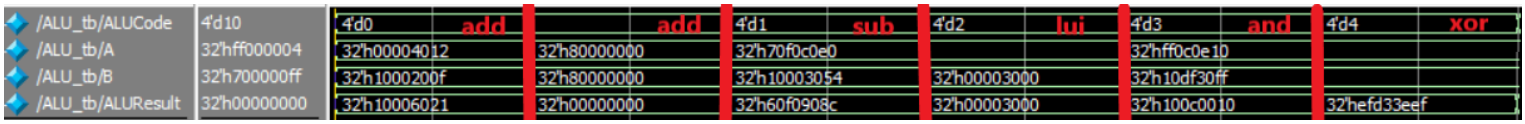


图 6.2.2 ALU 模块前 6 条运算仿真结果

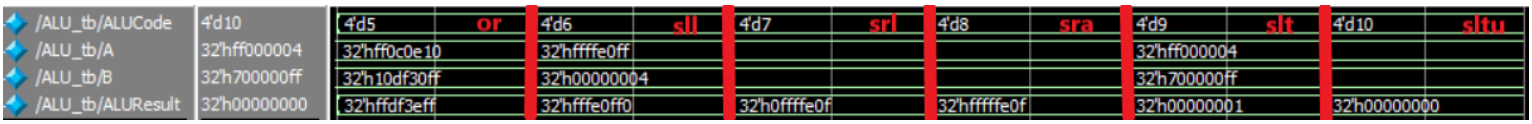


图 6.2.3 ALU 模块后 6 条运算仿真结果

分析：如图，将仿真结果与实际计算结果比较可知，ALU 模块计算正确，符合预期功能。

③ IF 模块功能仿真

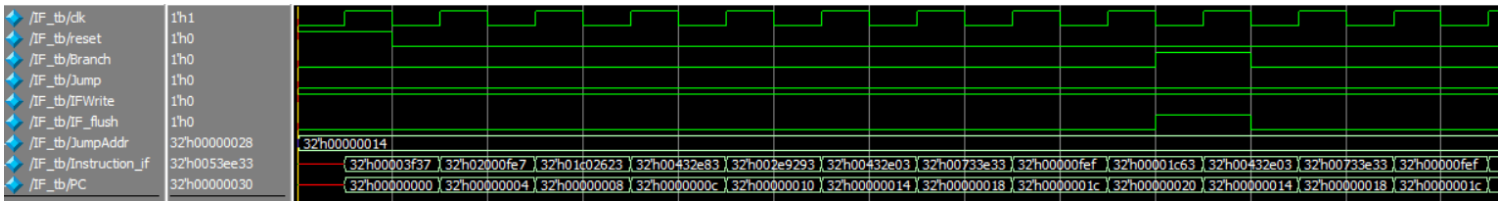


图 6.3.1 IF 模块功能仿真结果（前半部分）

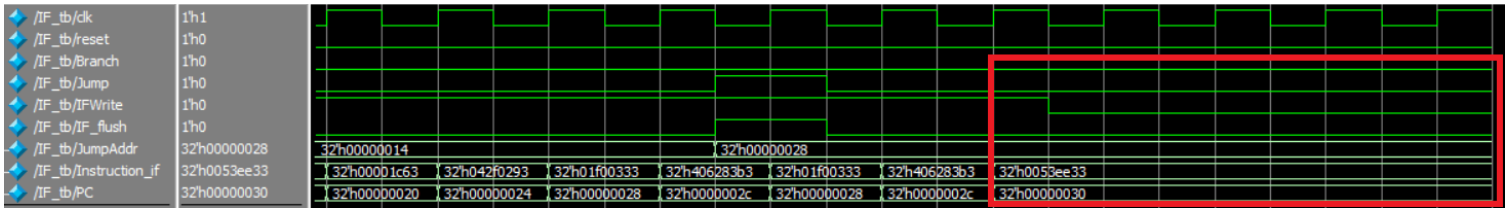


图 6.3.2 IF 模块功能仿真结果（后半部分）

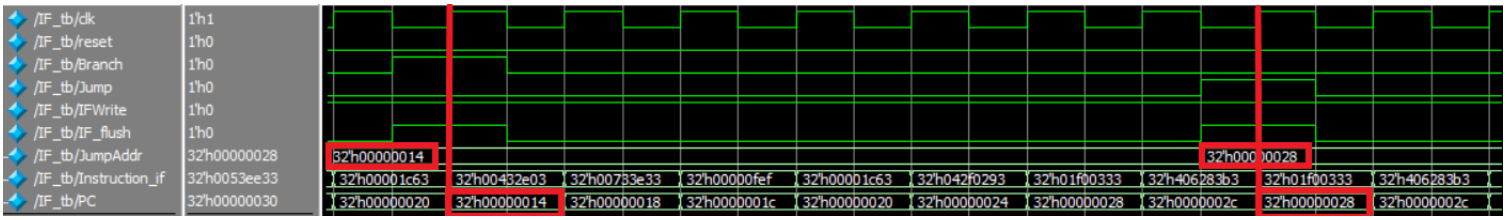


图 6.3.3 IF 模块功能仿真结果（PC 跳转）

分析：如图 6.3.1，当 IFWrite=1 且 IF_flush=0 时，流水线正常运行，每个时钟周期 PC 都在上一个指令地址的基础上加 4；如图 6.3.2，当 IFWrite=0 且 IF_flush=0 时，流水线阻塞，此时 PC 维持不变；如图 6.3.3，当 IF_flush=1 时，发生指令跳转，PC 在时钟上升沿被赋值为指令跳转地址 JumpAddr。综上，仿真输出与理论值一致，IF 模块设计正确。

④ CPU 功能仿真

图 6.3.1 CPU 功能仿真结果（clk：1-5）

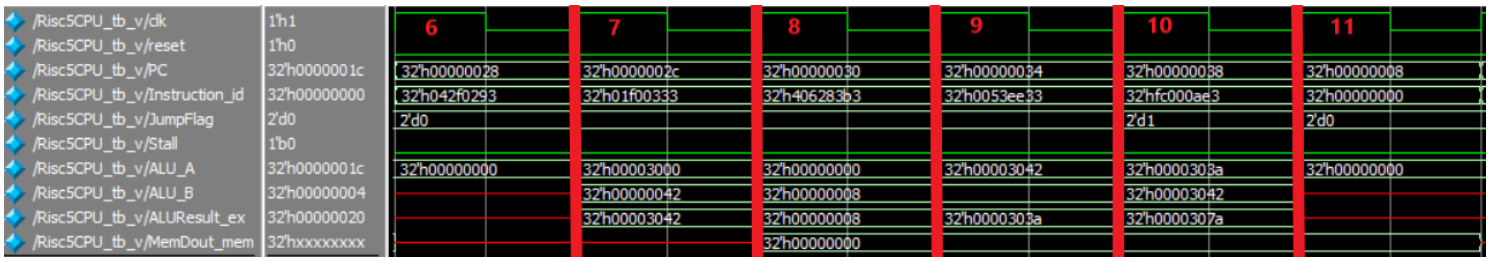


图 6.3.2 CPU 功能仿真结果（clk：6-11）

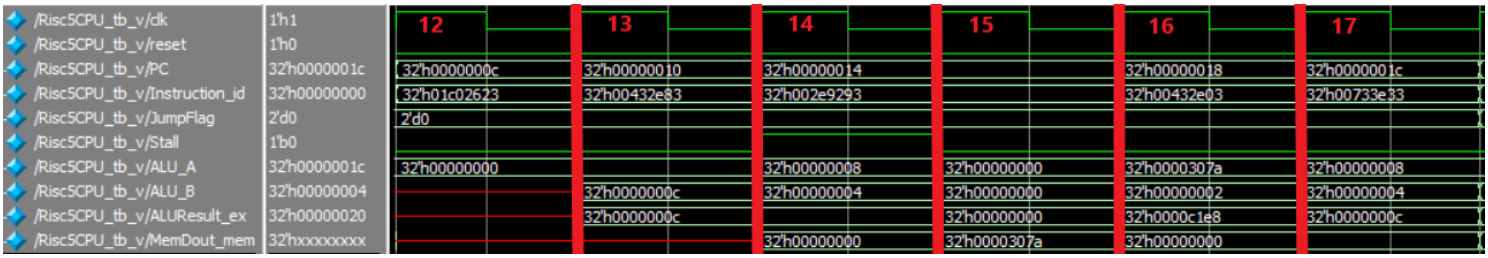
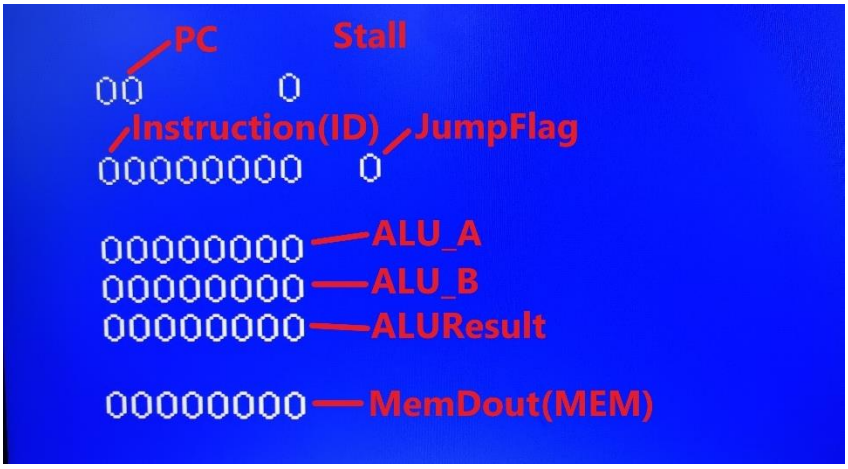


图 6.3.3 CPU 功能仿真结果（clk：12-17）

图 6.3.4 CPU 功能仿真结果（clk：18-22）

分析：如图，将各 clk 下测试程序的运行结果与正确结果相比较，两者完全一致。当程序执行到 clk=18 时，对应指令为：done: jal X31, done，其指令地址为 0x1c 且 PC=0x20。因此，在后续运行过程中，PC 会一直在 0x20 和 0x1c 之间跳转，输出符合预期。综上，CPU 顶层及各模块设计正确。

⑤ 开发板测试结果



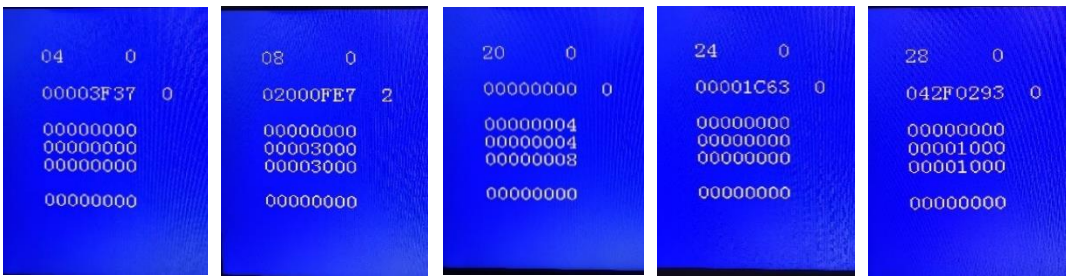


图 6.5 开发板验证结果

分析：如图，开发板上验证结果与理论结果、仿真结果完全一致，CPU 设计正确。

七、思考题

① 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

答：对一般的数据冒险，如以下指令：

lw x28, 04(x6)

add x30, x28, x29

在 lw 指令的执行过程中，x28 的数据在 WB 级才准备好，但 add 指令需要在 EX 级使用 x28 的值，因此，可以在 lw 和 add 之间插入一个气泡，使其变成二阶数据相关，通过转发实现计算。当将 add 指令改为 beq 指令后，如题设中所示，则 x28 的数据在 WB 级才准备好，而 beq 指令需要在 ID 级使用其值，显然，此时需要插入两个气泡，使其变

成三阶数据相关，再利用寄存器堆 Read After Write 的特性解决问题。因此，在硬件上，我们需要冒险检测单元 Hazard Detector 能够检测出 lw 的下一条指令是 beq 还是其他（如 add），再插入相应数量的气泡，这在当前的硬件结构下是无法实现的。为了避免修改硬件，可以在软件层面解决此问题：在汇编时根据指令类型自动插入对应数量的气泡；或调整指令顺序，更易于工程实现。

八、问题与解决方法

① 在 Decode 模块的仿真结果中，对不应产生立即数 imm 或地址偏移 offset 的指令，其会保持上一条指令 imm 或 offset 的值，而非 32'bx。检查后发现由于在 Decode.v 中对 imm 和 offset 赋值时，分支判断缺少了不需产生立即数和地址偏移的指令的情况，导致两值无法做出改变。将代码补充完整后，仿真输出正确。

② 在对 Decode 模块进行仿真时，sub 指令的 ALUCode 始终为 0，但检查代码逻辑却并未发现错误。通过重新阅读功能表、仔细分析，发现在进行分支判断时，我将 ADD_funct3 和 SUB_funct3 都作为了单独的一种情况，但两者值都为 3'b000，显然会引起错误。同理，SRL_funct3 和 SRA_funct3 也存在有类似的问题。将分支修改后重新仿真，结果正确。

③ 在对顶层设计进行功能仿真时，发现指令从第二条开始便全为 32'h0000_0000，同时输出 Stall 始终为高阻，ALU_B、ALUResult_ex、PC、MemDout_mem 等基本为不定。由于 IF 模块已经通过了功能仿真，在对流水线寄存器 IF/ID 检查后，我发现 IF/ID 的输入 R 设置为 IF_flush，这使得当没有使能输入也没有初值时 PC_id 和 Instruction_id 都是不确定的值。因此，将 R 的输入改为 IF_flush|reset 后重新测试，输出有所改变，但 Stall 仍始终为高阻。对 ID 级进行检查，发现我在代码中赋值时错将 Stall 写成了 stall，修改后重新仿真，指令读取仍出现错误：当发生跳转后，PC 变为不定。分析 PC 信号的产生过程，其受到 IFWrite 和 JumpAddr 的影响，由于跳转后，IFWrite=1，显然，PC=PC_temp，因此确定是跳转地址的产生出现了问题。果然，查看 IF 级输入，JumpAddr 恒为不定；进一步查看 ID 级 JumpAddr 的产生，发现当 JALR=1 时，rs1Data_id 为不定。因此，我检查了寄存器堆的代码，发现在做数据选择时，地址 rs2Sel 错被写成了 rs1Se2，修改后，查看寄存器堆的仿真输入和输出，发现 rs1Sel 和 rs2Sel 始终为不定，即 RegWrite 和 WriteAddr 为不定。显然，根据流水线的时钟进程，此问题涉及到信号的初始化。考虑到流水寄存器 IF/ID 清零信号 R 的取值，我将 ID/EX 的重置输入 R 改为了 Stall|reset，重新仿真，问题得到解决。