基于 RV32I 指令集的 RSIC-V 微处理器设计

3190103766　　石滨溥

2021 年 12 月 29 日

目录

1 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。

2 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。

- 算数运算指令：add、sub、addi
- 逻辑运算指令：and、or、xor、slt、sltu、andi、ori、xori、slti、stiu
- 移位指令：s11、sr1、sra、s11i、srli、srai
- 条件分支指令：beq、bne、blt、bge、bltu、begu
- 无条件跳转指令：jal、jalr
- 数据传送指令：1w、sW、lui、auipc
- 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。

3 实验原理与总体设计

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。示意图如图 1 所示。

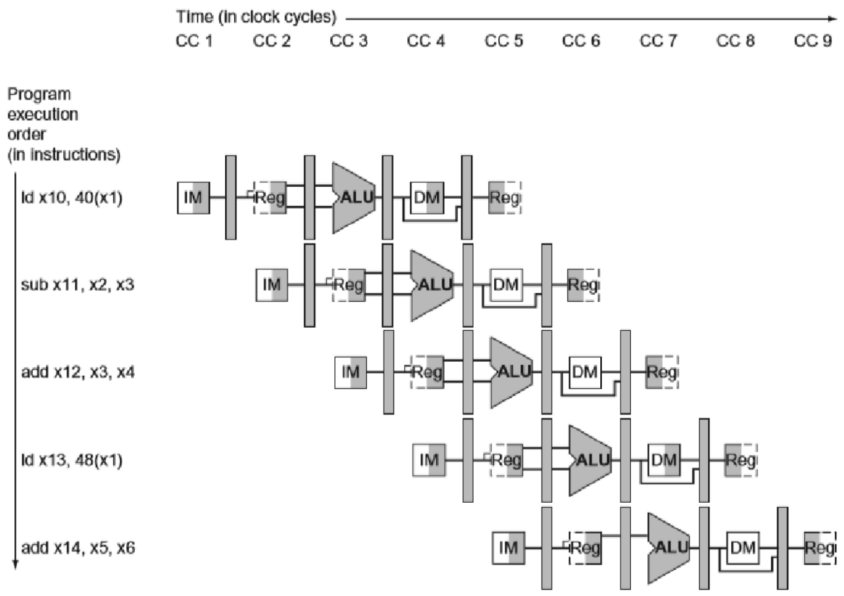

图 1: 流水线流水示意图

3.1 流水线的控制信号

(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 D 级的指令缓冲器中。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。
- IF Write: IF Write=-0 时阻塞 F/ID 流水线，同时暂停读取下一条指令。
- IF flush: IF fush=1 时清空 IF/ID 寄存器

(2) ID 级：指令译码级。对来自 F 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/D 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、ALUSICA 和 ALUSRCB，根据这些信号确定 ALU 操作、并选择两个 ALU 操作数 ALUA、ALUB。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 Mem Write.

(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。

3.2 数据相关和数据转发

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。

3.2.1 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 2 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。

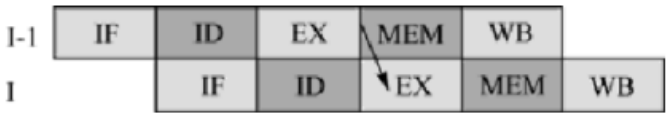

图 2: 一阶前推网络示意图

因此，导致操作数 A 的一阶数据相关判断的条件为：

- 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 的一阶数据相关判断的条件为：

- 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。

3.2.2 二阶数据相关与转发 (MEM 冒险)

如图 3 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。

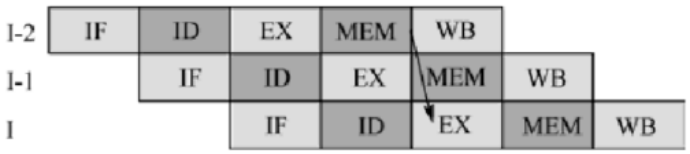

图 3: 二阶前推网络示意图

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 ($rdAddr\_men\ rs2Addr\_ex$);
- 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。

3.2.3 三阶数据相关

图 4 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。

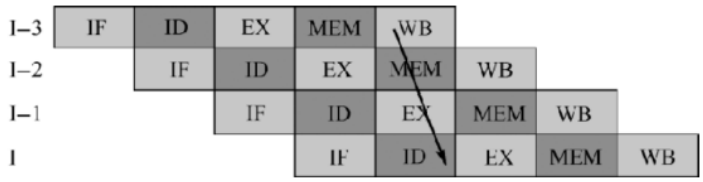

图 4: 三阶前推网络示意图

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 ($rdAddr\_wb = rs1Addr\_id$)。

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。

3.3 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F加一个冒险检测单元 (Hazard Detector)。它工作在 ID 级，当检测到上述情F时，在 lw 指令和后一条指令之间插入气泡，使后一条指令延退一个周期执行，这样可将一阶数据冒险问题变成二阶数据冒险问题，可以用转发来解决。

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：

- 上一条指令必须是 lw 指令 ($Memread\_ex = 1$);
- 两条指令读写同一个寄存器 ($rdAddr\_ex = rs1Addr\_id\ rdAddr\_ex = rs2Addr\_id$)。

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。

4 流水线 RISC-V 微处理器的设计实现

根据流水线不同阶段，将系统划分为 IF、ID、EX 和 MEM 四大模块。另外，系统还包含 IF/ID、ID/EX、EX/MEM、MEM/WB 四个流水线寄存器。

4.1 取指令集模块 (IF) 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 5 所示。

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

图 5: IF 模块的 I/O 引脚说明

4.1.1 指令选择器和加 4 加法器

IF_flush 控制信号由 $Branch$ 和 $Jump$ 产生，指令指针选择器根据 IF_flush 信号选择 $Ifwrite = 1$ 时的 PC 值，指令选择器和加法器的代码实现：

```
assign IF_flush = Jump | Branch;
wire [31:0] NextPC_if = PC + 32'd4;
wire [31:0] PC_select = IF_flush ? JumpAddr : NextPC_if;
```

4.1.2 PC 寄存器

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，PC 寄存器实现如下：

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

4.1.3 指令存储器

指令存储块 ROM 已给出，直接进行调用，输入地址输出指令，实现如下：

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

4.2 指令译码模块 (ID) 设计

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

4.2.1 指令译码

RISC-V 将指令分为 R、I、S、SB、U、UJ 等六类。根据上图指令表，从电路设计角度看，根据操作数的来源和立即数构成方式不同，再次细分指令为 R_type、I_type、LW、$JALR$、SW、SB_type、LUI、$AUIPC$、JAL 这九种。

首先可以通过指令得到 op、$funct3$、以及 $funct6\_7$，实现方式如下：

```
wire [6:0] op = Instruction[6:0];  //op last 7 bits
wire funct6_7 = Instruction[30];
wire [2:0] funct3 = Instruction[14:12];
```

其次，通过 op 可以分辨出该指令是何种类型。具体实现如下：

```
// gain which type
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);
```

之后，可以根据指令类别对一些控制字进行判断。比如，只有 lw 指令读取存储器且回写数据取自存储器；只有 SW 指令会对存储器写数据；需要进行回写的指令类型有 R_type、I_type、LW、$JALR$、LUI、$AUIPC$ 和 JAL；只有 JALR 和 JAL 两条无条件分支指令；以及操作数 A、B 根据图 7 可知。

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: 操作数选择信号功能表

因此，代码实现如下：

```
// control word
assign MemtoReg = LW;
assign MemRead = LW;
```

```
assign MemWrite = SW;
assign RegWrite = R_type||I_type||LW||JALR||LUI||AUIPC||JAL;
assign Jump = JALR || JAL;
assign ALUSrcA = JALR || JAL || AUIPC;
assign ALUSrcB[1] = JAL || JALR;
assign ALUSrcB[0] = ~(R_type || JAL || JALR);
```

同时，根据图 8，可以得到 ALUCode 和各种指令类别的关系。

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 功能表

因此，可以得到相关代码如下：

```
// ALU_Code
wire [5:0] type_RI = {R_type, I_type, funct3, funct6_7};
always @(*) begin
if(LUI) ALUCode = `alu_lui;
else begin
case (type_RI)
    {2'b10, `ADD_funct3, 1'b0}: ALUCode = `alu_add;
    {2'b10, `SUB_funct3, 1'b1}: ALUCode = `alu_sub;
    {2'b10, `SLL_funct3, 1'b0}: ALUCode = `alu_sll;
```

```
        {2'b10, `SLT_funct3, 1'b0}: ALUCode = `alu_slt;
        {2'b10, `SLTU_funct3, 1'b0}: ALUCode = `alu_sltu;
        {2'b10, `XOR_funct3, 1'b0}: ALUCode = `alu_xor;
        {2'b10, `SRL_funct3, 1'b0}: ALUCode = `alu_srl;
        {2'b10, `SRA_funct3, 1'b1}: ALUCode = `alu_sra;
        {2'b10, `OR_funct3, 1'b0}: ALUCode = `alu_or;
        {2'b10, `AND_funct3, 1'b0}: ALUCode = `alu_and;
        {2'b01, `ADDI_funct3, 1'b0}: ALUCode = `alu_add;
        {2'b01, `ADDI_funct3, 1'b1}: ALUCode = `alu_add;
        {2'b01, `SLLI_funct3, 1'b0}: ALUCode = `alu_sll;
        {2'b01, `SLLI_funct3, 1'b1}: ALUCode = `alu_sll;
        {2'b01, `SLTI_funct3, 1'b0}: ALUCode = `alu_slt;
        {2'b01, `SLTI_funct3, 1'b1}: ALUCode = `alu_slt;
        {2'b01, `SLTIU_funct3, 1'b0}: ALUCode = `alu_sltu;
        {2'b01, `SLTIU_funct3, 1'b1}: ALUCode = `alu_sltu;
        {2'b01, `XORI_funct3, 1'b0}: ALUCode = `alu_xor;
        {2'b01, `XORI_funct3, 1'b1}: ALUCode = `alu_xor;
        {2'b01, `SRLI_funct3, 1'b0}: ALUCode = `alu_srl;
        {2'b01, `SRAI_funct3, 1'b1}: ALUCode = `alu_sra;
        {2'b01, `ORI_funct3, 1'b0}: ALUCode = `alu_or;
        {2'b01, `ORI_funct3, 1'b1}: ALUCode = `alu_or;
        {2'b10, `ANDI_funct3, 1'b0}: ALUCode = `alu_and;
        {2'b10, `ANDI_funct3, 1'b1}: ALUCode = `alu_and;
        default: ALUCode = `alu_add;
    endcase
    end
    end
```

4.2.2 立即数产生电路

本文，立即数产生电路是包含在指令译码中的。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。再根据图 9，则可以得到立即数和各种指令类别的关系。

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

因此，立即数产生模块代码如下：

```
// Imm Gen
wire shift = (funct3 == 1) || (funct3 == 5);
always @(*) begin
    if(I_type) begin
        Imm <= shift ? {26'd0, Instruction[25:20]}
        : {{20{Instruction[31]}}, Instruction[31:20]};
        offset <= 32'd0;
    end
    else if(LW) begin
        Imm <= {{20{Instruction[31]}}, Instruction[31:20]};
        offset <= 32'd0;
    end
    else if(JALR) begin
        Imm <= 32'd0;
        offset <= {{20{Instruction[31]}}, Instruction[31:20]};
    end
    else if(SW) begin
        Imm  <= {{20{Instruction[31]}},
        Instruction[31:25], Instruction[11:7]};
        offset <= 32'd0;
    end
    else if(JAL) begin
        Imm <= 32'd0;
        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'd0;
    end
    else if(SB_type) begin
        Imm <= 32'd0;
        offset <= {{19{Instruction[31]}}, Instruction[31],
        Instruction[7], Instruction[30:25],
        Instruction[11:8], 1'b0};
    end
    else begin
        Imm <= 32'd0;
        offset <= 32'd0;
    end
end
```

4.2.3 寄存器堆

首先设置一个 32×32 的寄存器组，再根据读写地址以及写使能进行读写操作。其实现如下：

```
    wire [31:0] ReadData1, ReadData2;
    reg [31:0] registers_32 [31:0];  //32*32
    // load out 2 reg data
    assign ReadData1 = (ReadRegister1 == 5'd0) ?
            32'd0 : registers_32[ReadRegister1];
    assign ReadData2 = (ReadRegister2 == 5'd0) ?
            32'd0 : registers_32[ReadRegister2];
    always @(posedge sys_clk) begin  // positive clk write
    if(RegWrite) registers_32[WriteRegister] <= WriteData;
    end
```

与此同时，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。因此可以构造出如下原理图 (图 10)。

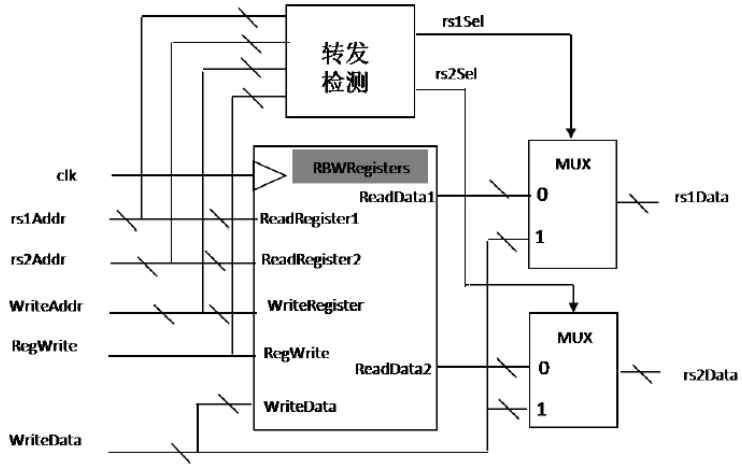

图 10: 具有 Read After Write 特性寄存器堆的原理框图

本文，在同一个寄存器模块中构造这种转发机制，其实现如下：

```
// add forword if WriteAddr == ReadAddr
wire rs1Sel = RegWrite && (WriteRegister != 0)
        && (WriteRegister == ReadRegister1);
wire rs2Sel = RegWrite && (WriteRegister != 0)
        && (WriteRegister == ReadRegister2);
assign regout1 = rs1Sel ? WriteData : ReadData1;
assign regout2 = rs2Sel ? WriteData : ReadData2;
```

4.2.4 分支检测电路

首先确定为 *SB_type*，其次判断两个寄存器输出大小 (无符号和有符号都需要)，这里和手册实现方式不同。

```
wire [6:0] op = Instruction[6:0];
wire [2:0] funct3 = Instruction[14:12];
wire SB_type = (op == `SB_type_op);
wire isLT = ($signed(rs1Data)) < ($signed(rs2Data)) ? 1'b1 : 1'b0;
wire isLTU = rs1Data < rs2Data ? 1'b1 : 1'b0;
wire isequal = (rs1Data == rs2Data ? 1'b1 : 1'b0);
```

然后根据指令的 *funct3* 判断为那种分支指令，其实现如下：

```
// choose whitch branch value
```

```
always @(*) begin
    if(SB_type) begin
        case (funct3)
        `beq_funct3 : Branch <= isequal;
        `bne_funct3 : Branch <= ~isequal;
        `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
```

4.2.5 冒险检测功能电路

由前面分析可知，冒险成立的条件为：

- 上一条指令必须是 lw 指令 ($MemRead\_ex = 1$);
- 两条指令读写同一个寄存器 ($rdAddr\_ex = rs1Addr\_id$ $rdAddr\_ex = rs2Addr\_id$)。

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。因此，代码实现如下：

```
assign Stall = ((rdAddr_ex == rs1Addr)
            || (rdAddr_ex == rs2Addr)) && MemRead_ex;
assign IFWrite = ~Stall;
```

4.3 执行模块 (EX) 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 11。

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 11: EX 模块的 I/O 引脚说明

4.3.1 ALU 模块

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。如图 12 为 ALUCode 对应的运算符。

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

因此，可以构建以下代码实现 ALU 计算。

```
always @(*) begin
case (ALUopcode)
    `alu_add : ALUout = ALUin_a + ALUin_b;
    `alu_sub : ALUout = ALUin_a - ALUin_b;
    `alu_lui : ALUout = ALUin_b;
    `alu_and : ALUout = ALUin_a & ALUin_b;
    `alu_xor : ALUout = ALUin_a ^ ALUin_b;
    `alu_or :  ALUout = ALUin_a | ALUin_b;
    `alu_sll : ALUout = ALUin_a << ALUin_b;
    `alu_srl : ALUout = ALUin_a >> ALUin_b;
    `alu_sra : ALUout = ($signed(ALUin_a)) >>> ALUin_b;
                    //after signed than
    `alu_slt : begin
        if(($signed(ALUin_a)) < ($signed(ALUin_b)))
            ALUout = 32'd1;
        else ALUout = 32'd0;
    end
    `alu_sltu : begin
        if(ALUin_a < ALUin_b) ALUout = 32'd1;
        else ALUout = 32'd0;
    end
    default: ALUout = ALUin_a + ALUin_b;
endcase
```

4.3.2 数据前推电路

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 如图 13 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

图 13: 前推电路输出信号的含义

由前面提到的一、二阶数据相关判断条件，因此可以得到以下代码：

```
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);
```

4.4 数据存储器模块 (MEM) 设计

使用 Xilinx 的 IP 内核实现，数据存储器设计为容量为 64×32*bit* 的单端口 RAM，输出采用组合输出。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] 连接其原理图如图 14 所示：

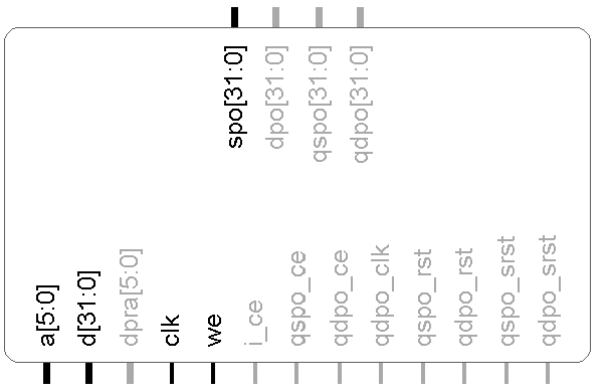

图 14: IP 核构建数据存储器模块 RAM

4.5 寄存器回写模块 (WB) 设计

寄存器回写模块就是一个 MUX，根据 $MemtoReg\_wb$ 使能信号，选择回写 ALU 的结果还是 MEM 中的结果。

其实现如下：

```
assign RegWriteData_wb = MemtoReg_wb ? MemDout_wb : ALUResult_wb;
```

4.6 流水线寄存器设计

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）。

4.6.1 IF/ID 寄存器实现

IF/ID 寄存器实现如下：

```
always @(posedge clk) begin
    if(R) begin                    //if flush
        PC_out_c <= 0;
        Instruction_out_c <= 0;
    end
    else if(EN) begin
        // change status
        PC_out_c <= PC_in_c;
        Instruction_out_c <= Instruction_in_c;
    end
    else begin
        PC_out_c <= PC_out_c;
        //else stay last status
        Instruction_out_c <= Instruction_out_c;
    end
end
```

4.6.2 ID/EX 寄存器实现

ID/EX 寄存器实现如下：

```
always @(posedge clk) begin
    if(R) begin
        {MemtoReg_ex, RegWrite_ex, MemWrite_ex, MemRead_ex} <= 0;
        ALUCode_ex <= 0;
        {ALUSrcA_ex, ALUSrcB_ex} <= 0;
        {rdAddr_ex, rs1Addr_ex, rs2Addr_ex} <= 0;
        {PC_ex, rs1Data_ex, rs2Data_ex, Imm_ex} <= 0;
    end
    else begin
        {MemtoReg_ex, RegWrite_ex, MemWrite_ex, MemRead_ex}
            <= {MemtoReg_id, RegWrite_id, MemWrite_id, MemRead_id};
        ALUCode_ex <= ALUCode_id;
        {ALUSrcA_ex, ALUSrcB_ex} <= {ALUSrcA_id, ALUSrcB_id};
        {rdAddr_ex, rs1Addr_ex, rs2Addr_ex}
            <= {rdAddr_id, rs1Addr_id, rs2Addr_id};
        {PC_ex, rs1Data_ex, rs2Data_ex, Imm_ex}
            <= {PC_id, rs1Data_id, rs2Data_id, Imm_id};
    end
end
```

4.6.3 EX/MEM 寄存器实现

EX/MEM 寄存器实现如下：

```
always @(posedge clk) begin
    {MemtoReg_mem, RegWrite_mem, MemWrite_mem}
                <= {MemtoReg_ex, RegWrite_ex, MemWrite_ex};
    {ALUResult_mem, MemWriteData_mem}
                <= {ALUout, MemWriteData_ex};
    rdAddr_mem <= rdAddr_ex;
end
```

4.6.4 MEM/WB 寄存器实现

MEM/WB 寄存器实现如下：

```
always @(posedge clk) begin
    {MemtoReg_wb, RegWrite_wb} <= {MemtoReg_mem, RegWrite_mem};
    {MemDout_wb, ALUResult_wb} <= {MemDout_mem, ALUResult_mem};
    rdAddr_wb <= rdAddr_mem;
end
```

4.7 顶层模块设计

根据图 15，连接各模块。在本次实验当中，我也将每一级构成了一个整体模块，具体实现请见相关代码。

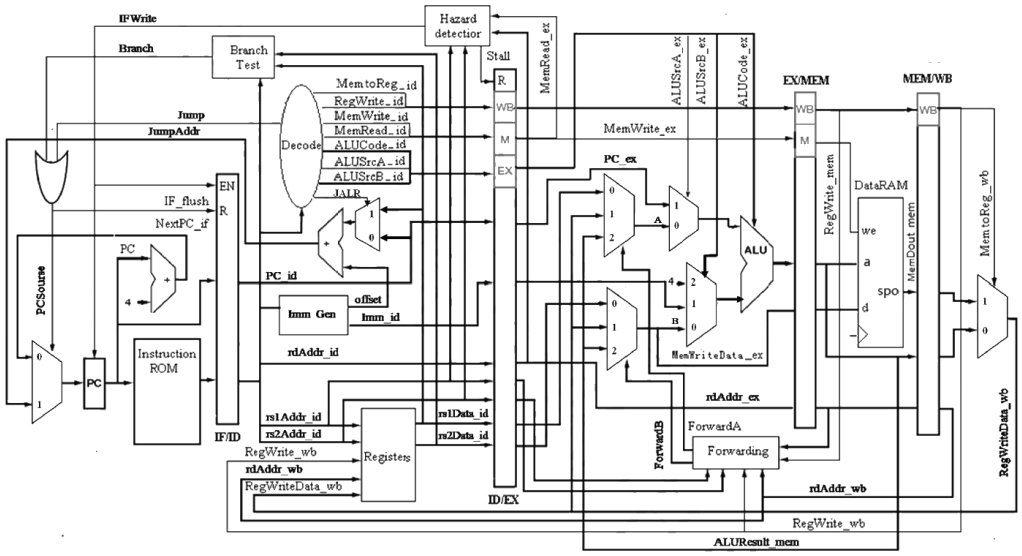

图 15: 流水线 RISC-V 微处理器的原理框图

5 Vivado 仿真

5.1 Decode 模块仿真

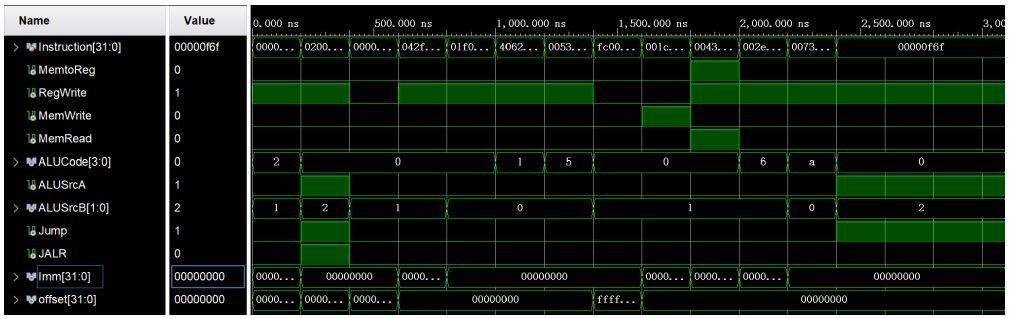

图 16: Decode 模块仿真整体

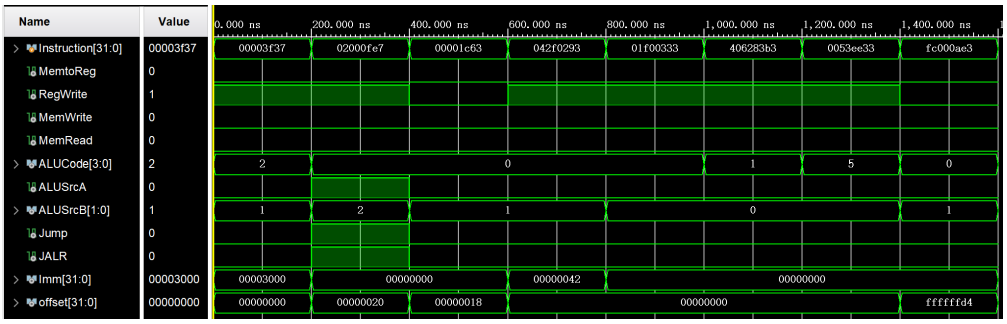

图 17: Decode 模块仿真前 8 条指令

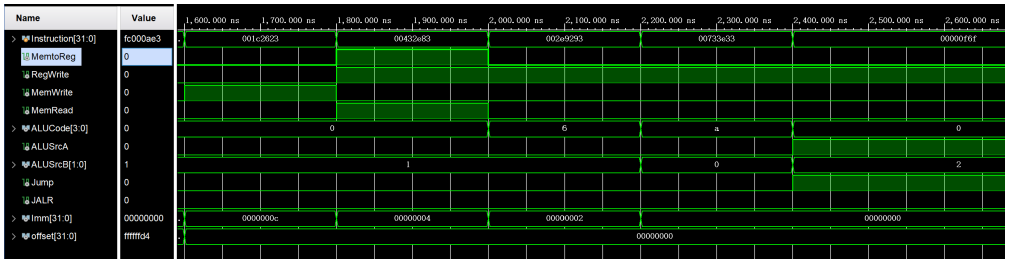

图 18: Decode 模块仿真后 5 条指令

从图 17 可以看出，首先 lui X30, 0x3000，是 U 型指令，输出 Imm 应为 32'o3000，ALUCode 为 4'b2，ALUSrcA 为 0，ALUSrcB 为 1，需要写回故 RegWrite 为 1。波形正确。

第二条为 jalr X31, later(X0)，I 型指令，Imm 无意义，offset 应为 32'o20，ALUCode

为 4'b0，ALUSrcA 为 1，ALUSrcB 为 2，JALR 为 1，Jump 为 1，需要写回故 RegWrite 为 1。波形正确。

第三条为 bne X0, X0, end，SB 型指令，Imm 无意义，offset 应为 32'o18，ALUCode 为 4'b0，ALUSrcA 为 0，ALUSrcB 为 1，不需要写回故 RegWrite 为 0。波形正确。

第四条指令为 addi X5, X30, 42，I 型指令，Imm 应输出 32'o42，offset 无意义，ALUCode 为 4'b0，ALUSrcA 为 0，ALUSrcB 为 1，需要写回故 RegWrite 为 1。波形正确。

第五条指令为 add X6, X0, X31，R 型指令，Imm 无意义，offset 无意义，ALUCode 为 4'b0，ALUSrcA 为 0，ALUSrcB 为 0，需要写回故 RegWrite 为 1。波形正确。

第六条指令为 sub X7, X5, X6，R 型指令，Imm 无意义，offset 无意义，ALUCode 为 4'b1，ALUSrcA 为 0，ALUSrcB 为 0，需要写回故 RegWrite 为 1。波形正确。

第七条指令为 or X28, X7, X5，R 型指令，Imm 无意义，offset 无意义，ALUCode 为 4'b5，ALUSrcA 为 0，ALUSrcB 为 0，需要写回故 RegWrite 为 1。波形正确。

第八条指令为 beq X0, X0, earlier，SB 型指令，Imm 无意义，offset 为 32'bfffffd4(符号扩展)，ALUCode 为 0，ALUSrcA 为 0，ALUSrcB 为 1，不需要写回故 RegWrite 为 0。波形正确。

从图 18 中可以看出，第九条指令为 sw X28, 0C(X0)，S 型指令，Imm 为 32'bc，offset 无意义，ALUCode 为 0，ALUSrcA 为 0，ALUSrcB 为 1，MemWrite 为 1。波形正确。

第十条指令为 lw X29, 04(X6)，I 型指令，Imm 为 32'b4，offset 无意义，ALUCode 为 0，ALUSrcA 为 0，ALUSrcB 为 1，MemRead=l, MemtoReg=1, RegWrite=1。波形正确。

第十一条指令为 sll X5, X29, 2，I 型指令，Imm 为 32'b2，offset 无意义，ALUCode 为 6，ALUSrcA 为 0，ALUSrcB 为 1，需要写回故 RegWrite 为 1。波形正确。

第十二条指令为 sltu X28, X6, X7，R 型指令，Imm 无意义，offset 无意义，ALUCode 为 10，ALUSrcA 为 0，ALUSrcB 为 0，需要写回故 RegWrite 为 1。波形正确。

第十三条指令为 jal X31, done，UJ 型指令，Imm 无意义，offset 为 0，ALUCode 为 1，ALUSrcA 为 1，ALUSrcB 为 2，需要写回故 RegWrite 为 1。波形正确。

综上，Decode 模块仿真正确，功能实现。

5.2 ALU 模块仿真

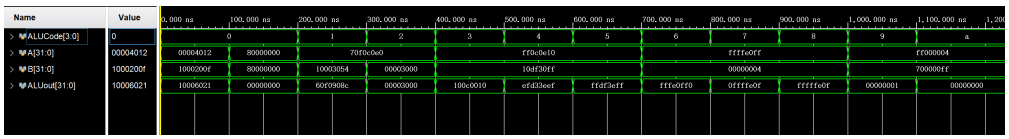

图 19: ALU 模块仿真整体

按照顺序下来每个周期是 add，add，sub，lui，and，xor，or，sll，srl，sra，slt，sltu。经过计算，都符合结果。

综上，ALU 模块功能实现。

5.3 IF 模块仿真

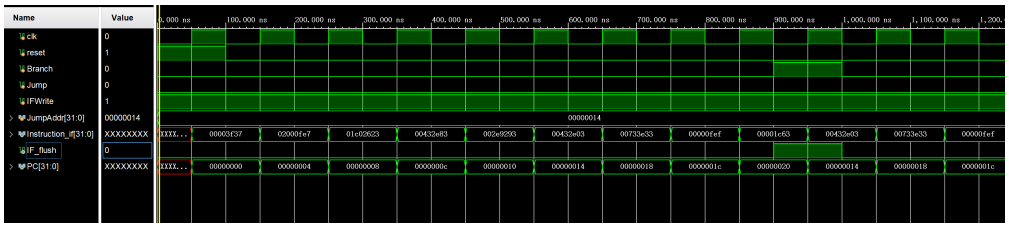

图 20: IF 模块仿真前半段

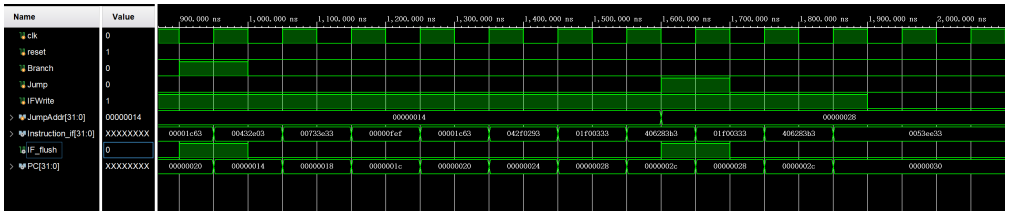

图 21: IF 模块仿真后半段

从图 20 可以看出,第九周期到第十周期输入一个跳转使能 $Branch = 1$,则 $IF\_flush = 1$，故 PC 值从 32'b20 跳转到 32'b14。之后从图 21 也可以看出，这里也有个一个跳转使能 $Jump = 1$，则 $IF\_flush = 1$，使 PC 从 32'b2c 跳转到 32'b28。

综上，IF 模块功能得到实现。

5.4 CPU 顶层模块仿真

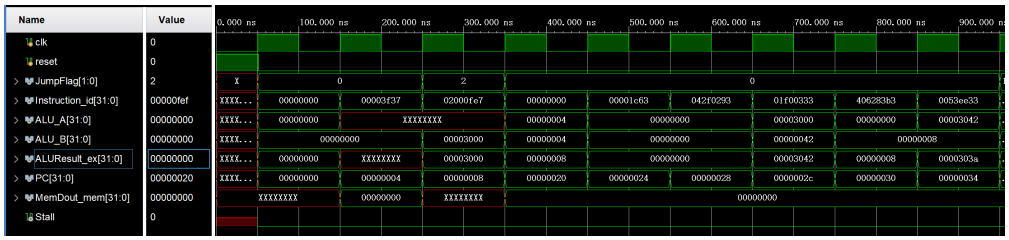

图 22: CPU 顶层模块前半段

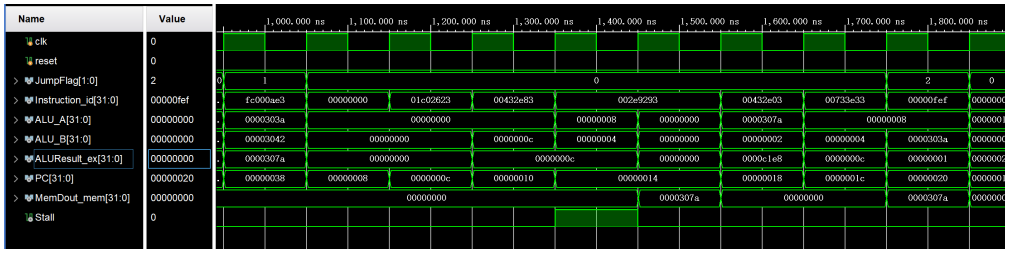

图 23: CPU 顶层模块后半段

综上，测试结果运行正确，CPU 顶层模块功能得到实现。

6 遇到问题及解决

(1) 符号扩展位的实现方法，可以直接使用 {n{Binvert}} 的方式；

(2) 带符号位的数的运算，即将无符号数转化成有符号数。本文使用的方法是使用 verilog 自带的类型转换函数 $signed(x)，即可将 x 看作带符号数；

(3) case 语句当中，有选择是与不定数 x 有关，当时我以为 1'bx 可以既判别 1'b0，又可以判别 1'b1，但是实际上不能。因此，只能多加上一条 case 情况，将 0 和 1 都写一遍，即可解决。

(4) 仿真时出现高阻态，这个原因一般是引脚没对应上。比如一个 4 路的模块输出连接一个 8 路的模块输入，这样就会导致出现高阻态。经过排查，是顶层模块出现问题，连接写错。

(5) 阻塞赋值的问题。仿真 Decode 模块时，当时出现一个周期输出没对上，主要是在 if-else 语句当中采用了阻塞赋值，使输出在原先的输出基础上没有变化。改为非阻塞赋值即可解决问题。

(6) CPU 整体仿真时，也出现过没存进寄存器的原因。可以通过一步步地向前递推找到原因，我当时应该也是引脚连接连错。

7 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

本文实现的 CPU 分支判断是在 ID 级的 Branch 模块实现的，和 MEM 级差了两级，因此至少得插入两个 nop，再通过转发，才能够解决冲突问题。首先，从 RAM load 数据比较慢，会应资源竞争而变慢，导致延迟增加；其次代价太高，空了两个周期 CPU 没运作。因此，这一般是通过**编译器层面**，调整指令顺序，在之间插入其他无关指令来解决。