实验 30　基于 RV32I 指令集的 RISC-V 微处理器设计

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

（一）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。

① 算术运算指令：add、sub、addi

② 逻辑运算指令：and、or、xor、slt、sltu、andi、ori、xori、slti、sltiu

③ 移位指令：sll、srl、sra、slli、srli、srai

④ 条件分支指令：beq、bne、blt、bge、bltu、bgeu

⑤ 无条件跳转指令：jal、jalr

⑥ 数据传送指令：lw、sw、lui、auipc

⑦ 空指令：nop

（2）采用 5 级流水线技术，对数据冒险实现转发或阻塞功能。

（3）在 Nexys Video 开发系统中实现 RISC-V 微处理器，要求 CPU 的运行速度大于 25MHz。

（二）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、RV32I 基本整数指令集简介

RSIC-V 指令集是 UC Berkeley 研发的一款 CPU 指令集，该指令集最初的目的是为了支持计算机体系结构的研发和教学，但是后来 Berkeley 希望该指令集能够成为一个可以被产业界真正实现的开源架构。

RSIC-V 的设计使得它具有良好的扩展性和定制化。在基本的整数指令上可以添加一个或者多个可选的指令集扩展，但是基础的整数指令不能修改。为了支持更通用的软件开发工作，RISC-V 提供了一些标准的扩展模块，这些扩展模块提供了整数乘法/除法，原子操作，单精度浮点和双精度浮点算术运算等。其中 I 架构（RV32I/RV64I/RV128I）指令集定义为标准整数指令集，包括的主要指令有：整数计算指令，整数加载，整数存储，控制流指令，这部分指令对于所有的 RSIC-V 实现都是完全一样的。

1、寄存器堆

RV32I 系统的寄存器结构采用标准的 32 位寄存器堆，共 32 个宽度为 32 比特的寄存器，标号为 0～31。其中，第 0 寄存器永远为常数 0。寄存器的含义规定如表 30.1 所示。

注意：RISC-V 架构还定义了一些控制和状态寄存器（Control and Status Register，CSR），用于配置或记录一些运行的状态。由于本实验基本要求没有涉及到 CSR，这里不作介绍。

表 30.1 寄存器堆使用规范

寄存器编号	助记符	用途
X0	zero	硬件连线 0
X1	ra	Return address 返回地址
X2	sp	堆栈指针
X3	gp	全局指针
X4	tp	Thread（线程）指针
X5～X7	t0～t2	临时变量
X8	s0/fp	临时变量/帧指针
X9	s1	临时变量，过程调用时不需要恢复
X10～X17	a0~a7	函数参数，其中 a0、a1 用于存放函数返回值
X18～X27	s2~s11	保留
X28～X31	t3～t6	临时变量

2、RV32I 指令格式

在 RV32I 指令集中，有四种核心格式：为 R 型、I 型、S 型和 U 型，另外，基于立即数的不同处理，S 型和 U 型还各有一种变型格式：SB 型和 UJ 型。指令格式如图 30.1 所示，所有指令都是固定的 32 位长度。

指令格式中的指令操作码（opcode，简称 op）、源操作数的寄存器号（rs1 和 rs2）、目标寄存器号（rd）、指令操作扩展码（funct3 和 funct7）等字段都固定在相同位置上。但立即数字段（imm）在不同的指令类型中存放位置是不同的，这一点应引起特别重视。

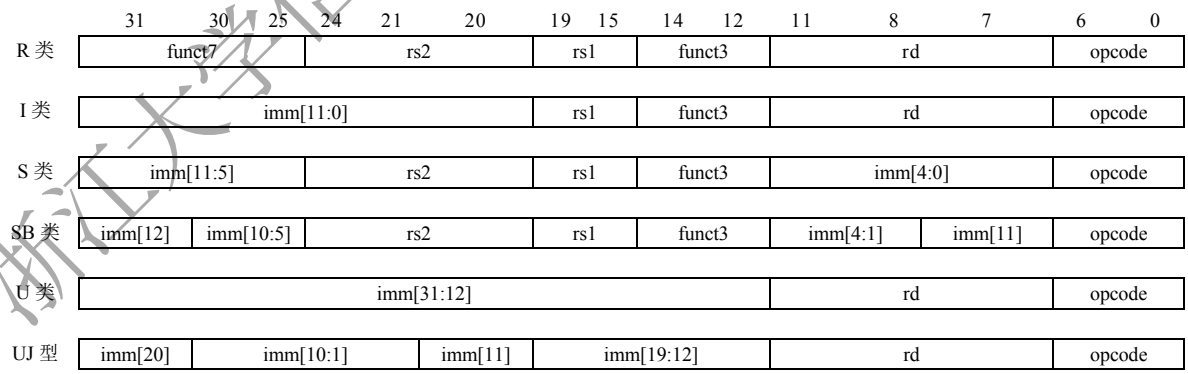

图 30.1　RV32I 指令格式

3、RV32I 指令表

下面通过表格简单介绍本实验使用的 RISC-V 核心指令。表 30.2 列出 RV32I 基本整数指令的类型、指令操作码、指令操作扩展码等信息。

表 30.2　RV32I 指令格式

类　别	指令使用格式	功能	类型	opcode	funct3	funct6/7
算术或逻辑运算	add　rd,rs1,rs2	加法：rd= rs1+rs2	R	7'h33	3'o0	7'h00
	sub　rd,rs1,rs2	减法：rd= rs1-rs2		7'h33	3'o0	7'h20
	slt　rd,rs1,rs2	数值大小判断：rd=(rs1<rs2)		7'h33	3'o2	7'h00
	sltu　rd,rs1,rs2	无符号数判断：rd=(rs1<rs2)		7'h33	3'o3	7'h00
	xor　rd,rs1,rs2	按位异或: rd= rs1 ^ rs2		7'h33	3'o4	7'h00
	or　rd,rs1,rs2	按位或: rd= rs1 | rs2		7'h33	3'o6	7'h00
	and　rd,rs1,rs2	按位与: rd= rs1 & rs2		7'h33	3'o7	7'h00
	addi　rd,rs1,imm	立即数符号扩展，加法运算	I	7'h13	3'o0	n.a.
	andi　rd,rs1,imm	立即数符号扩展，与运算		7'h13	3'o7	n.a.
	ori　rd,rs1,imm	立即数符号扩展，或运算		7'h13	3'o6	n.a.
	slti　rd,rs1,imm	立即数 rs1<imm 判断		7'h13	3'o2	n.a.
	sltiu　rd,rs1,imm	立即无符号数 rs1<imm 判断		7'h13	3'o3	n.a.
	xori　rd,rs1,imm	立即数符号扩展，异或运算		7'h13	3'o4	n.a.
移位	sll　rd,rs1,rs2	左移：rd=rs1<<rs2	R	7'h33	3'o1	7'h00
	srl　rd,rs1,rs2	右移：rd=rs1>>rs2		7'h33	3'o5	7'h00
	sra　rd,rs1,rs2	算术右移：rd=rs1>>>rs2		7'h33	3'o5	7'h20
	slli　rd,rs1,sa	左移：rd=rs1<<sa	I	7'h13	3'o1	6'h00
	srli　rd,rs1,sa	右移：rd=rs1>>sa		7'h13	3'o5	6'h00
	srai　rd,rs1,sa	算术右移：rd=rs1>>>sa		7'h13	3'o5	6'h10
数据传送	lw　rd,imm(rs1)	rd=memory[rs1+imm]	I	7'h03	3'o2	n.a.
	lb	Byte from memory to register		7'h03	3'o0	n.a.
	lh	Halfword from memory to register		7'h03	3'o1	n.a.
	ld	Doubleword from memory to register		7'h03	3'o3	n.a.
	lbu	Unsigned byte from memory to register		7'h03	3'o4	n.a.
	lhu	Unsigned halfword from memory to registe		7'h03	3'o5	n.a.
	lwu	Unsigned word from memory to register		7'h03	3'o6	n.a.
	sw　rs2, imm(rs1)	memory[rs1+imm]=rs2	S	7'h23	3'o2	n.a.
	sb	Byte from register to memory		7'h23	3'o0	n.a.
	sh	Halfword from register to memory		7'h23	3'o1	n.a.
	sd	Doubleword from register to memory		7'h23	3'o7	n.a.
	lui rd,imm	rd={imm[31:12],12'd0}	U	7'h37	n.a.	n.a.
	auipc rd,imm	rd={imm[31:12],12'd0}+pc		7'h17	n.a.	n.a.
无条件分支	jal rd,offset	rd=PC+4; go to PC+offset	UJ	7'h6F	n.a.	n.a.
	jalr rd,offset(rs1)	rd=PC+4; go to rs1+ offset	I	7'h67	3'o0	n.a.
条件分支	beq　rs1,rs2,offset	if(rs1==rs2) goto PC+offset	SB	7'h63	3'o0	n.a.
	bne　rs1,rs2,offset	if(rs1!=rs2) goto PC+offset		7'h63	3'o1	n.a.
	blt　rs1,rs2,offset	if(rs1<rs2) goto PC+offset		7'h63	3'o4	n.a.
	bge　rs1,rs2,offset	if(rs1>=rs2) goto PC+offset		7'h63	3'o5	n.a.
	bltu　rs1,rs2,offset	if(rs1<rs2) goto PC+offset		7'h63	3'o6	n.a.
	bgeu　rs1,rs2,offset	if(rs1>=rs2) goto PC+offset		7'h63	3'o7	n.a.

对于 RV32I 指令集有两点值得说明一下：

（1）I 型指令类型中，立即数移位指令格式与图 3.1 所示 I 型格式略有不同，imm[11:0]（即

instruction[31:20]）分两部分：高 6 位为指令操作扩展码（funct6）；低 6 位为无符号移位操作数 sa。

（2）本实验定义空指令 nop 指令码为 32'd0，对寄存器堆和存储器不进行任何读写操作。

四 、实验原理

（一）总体设计

流水线是数字系统中一种提高系统稳定性和工作速度的方法，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。如图30. 2 示，一个指令的执行需要 5 个时钟周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理。

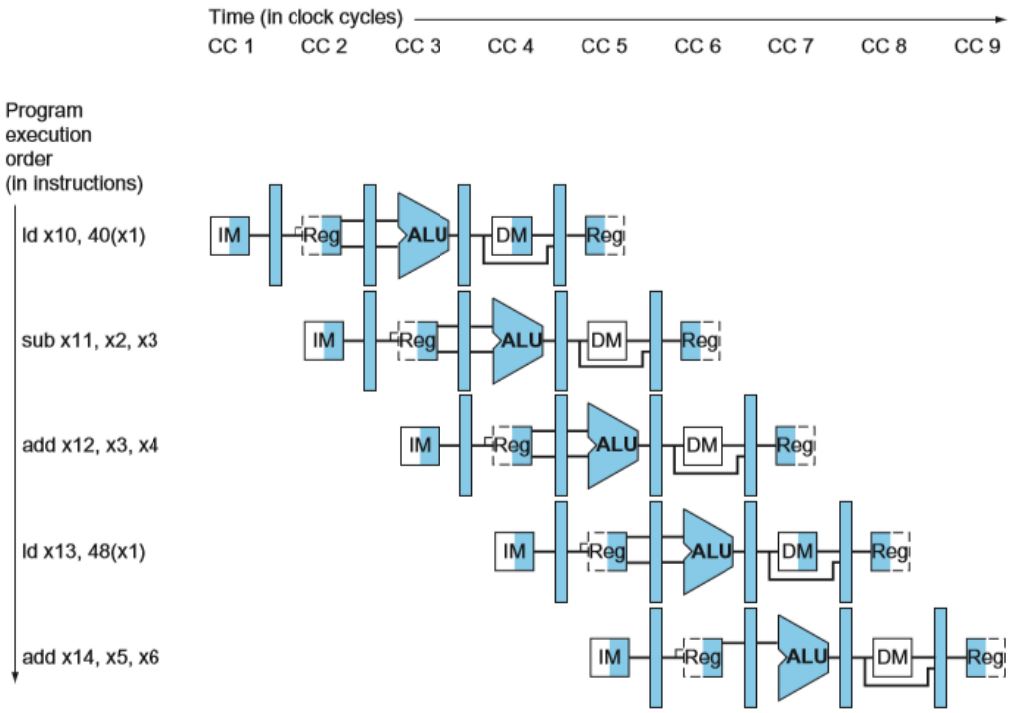

图 30.2 流水线流水作业示意图

图 30.3 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。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。

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

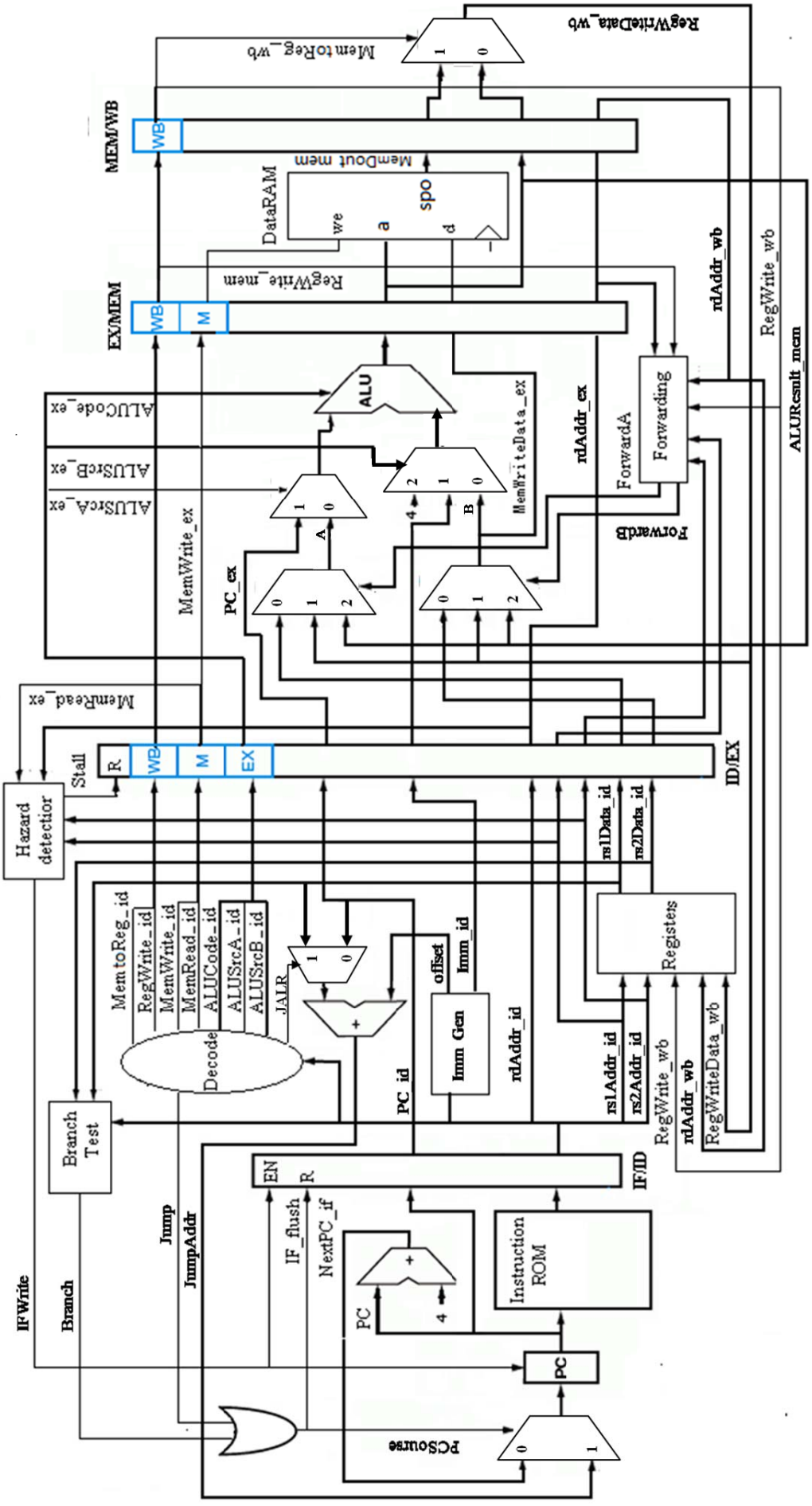

图 30.3　流水线 RISC-V 微处理器的原理框图

（2）ID 级：指令译码级。对来自 IF 级的指令进行译码，并产生相应的控制信号。整个 CPU 的控制信号基本都是在这级上产生。该级自身不需任何控制信号。

流水线冒险检测也在该级进行，即当流水线冒险条件成立时，冒险检测电路产生 Stall 信号清空 ID/EX 寄存器，同时冒险检测电路产生低电平 IFWrite 信号阻塞 IF/ID 流水线。即插入一个流水线气泡。

（3）EX级：执行级。此级进行算术或逻辑操作。此外数据传送指令所用的 RAM 访问地址也是在本级上实现。控制信号有 ALUCode、ALUSrcA 和 ALUSrcB，根据这些信号确定 ALU 操作、并选择两个 ALU 操作数 ALU_A、ALU_B。

另外，数据转发也在该级完成。数据转发控制电路产生 ForwardA 和 ForwardB 两组转发控制信号。

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

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

2．数据相关与数据转发

如果上一条指令的结果还没有写入到寄存器中，而下一条指令的源操作数又恰恰是此寄存器的数据，那么，它所获得的将是原来的数据，而不是更新后的数据。这样的相关问题称为数据相关。如图 30.4 所示的五级流水结构，当前指令与前三条指令都构成数据相关问题。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。

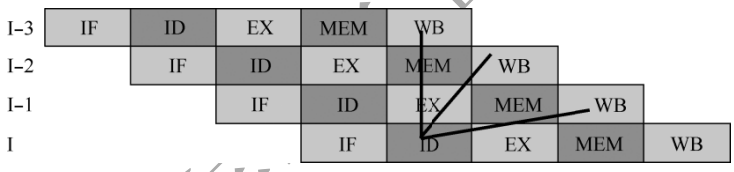

图 30.4　数据相关性问题示意图

（1）一阶数据相关与转发（EX 冒险）

如图30.5 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。从图30.5 可以看出，第 I 条指令的 EX 级与第 I–1 条指令的 MEM 级处于同一时钟周期，且数据转发必须在第 I 条指令的 EX 级完成。因此，导致操作数 A 的一阶数据相关判断的条件为：

① MEM 级阶段必须是写操作（RegWrite_mem=1）；

② 目标寄存器不是 X0 寄存器（rdAddr_mem ≠ 0）；

③ 两条指令读写同一个寄存器（rdAddr_mem=rs1Addr_ex）。

导致操作数 B 的一阶数据相关成立的条件为：

① MEM 级阶段必须是写操作（RegWrite_mem=1）；

② 目标寄存器不是 X0 寄存器（rdAddr_mem≠0）。

③ 两条指令读写同一个寄存器（rdAddr_mem=rs2Addr_ex）。

除了第 I–1 条指令为 lw 外（注意：本设计不考虑 lb、lh、ld、lbu、lhu 和 lwu 数据传送指令），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 级，如图30.5 所示。

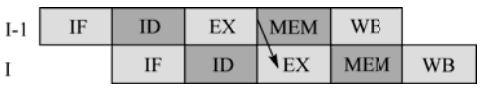

图 30.5　一阶前推网络示意图

lw 指令发生一阶数据相关的处理方法在后面介绍。

（2）二阶数据相关与转发（MEM 冒险）

如图30.6 所示，如果第 I 条指令的源操作寄存器与第 I–2 条指令的目标寄存器相重，将导致二阶数据相关。导致操作数 A 的二阶数据相关必须满足下列条件：

① WB 级阶段必须是写操作（RegWrite_wb=1）；

② 目标寄存器不是 X0 寄存器（rdAddr_wb≠0）；

③ 一阶数据相关条件不成立（rdAddr_mem≠rs1Addr_ex）；

④两条指令读写同一个寄存器（rdAddr_wb=rs1Addr_ex）。

导致操作数 B 的二阶数据相关必须满足下列条件：

① WB 级阶段必须是写操作（RegWrite_wb=1）；

② 目标寄存器不是 X0 寄存器（rdAddr_wb≠0）；

③ 一阶数据相关条件不成立（rdAddr_mem≠rs2Addr_ex）；

④ 两条指令读写同一个寄存器（rdAddr_wb=rs2Addr_ex）。

当发生二阶数据相关问题时，解决方法是将第 I–2 条指令的回写数据 RegWriteData 转发至 I 条指令的 EX 级，如图30.6 所示。

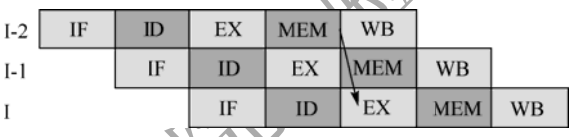

图 30.6 二阶前推网络示意图

（3）三阶数据相关

图30.7 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。

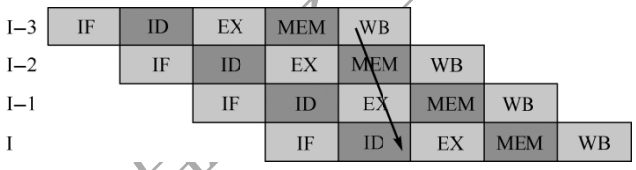

图 30.7 三阶前推网络示意图

导致操作数 A 的三阶数据相关必须满足下列条件：

① 寄存器必须是写操作（RegWrite_wb =1）；

② 目标寄存器不是 X0 寄存器（rdAddr_wb≠0）；

③ 读写同一个寄存器（rdAddr_wb =rs1Addr_id）；

同样，导致操作数 B 的三阶数据相关必须满足下列条件；

① 寄存器必须是写操作（RegWrite_wb =1）；

② 目标寄存器不是 X0 寄存器（rdAddr_wb≠0）；

③ 读写同一个寄存器（rdAddr_wb =rs2Addr_id）。

该类数据相关问题可以通过改进设计寄存器堆的硬件电路来解决，要求寄存器堆具有 Read After Write 特性，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。具体设计方法将在后面介绍。

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：

① 上一条指令必须是 lw 指令（MemRead_ex=1）；

② 两条指令读写同一个寄存器（rdAddr_ex=rs1Addr_id 或 rdAddr_ex=rs2Addr_id）。

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

（二）流水线 RISC-V 微处理器的设计

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。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 模块的接口信息如表 30.3 所示。

表 30.3 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]		

（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30.8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。

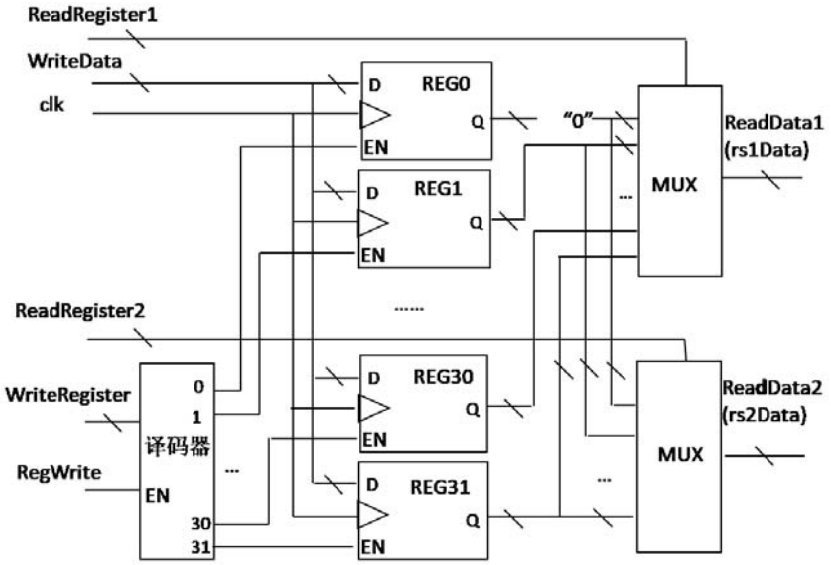

图 30.8　寄存器堆的原理框图

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。

注意：用 Verilog HDL 设计描述寄存器堆时，用存储器变量定义 32 个 32 位寄存器更为方便。下面为描述寄存器堆核心语句。

```
reg [31:0]regs [31:0];//定义 32*32 存储器变量
assign ReadData1= (ReadRegister1 == 5'b0) ? 32'b0 : regs[ReadRegister1];//端口 1 数据读出
assign ReadData2= (ReadRegister2 == 5'b0) ? 32'b0 : regs[ReadRegister2];//端口 2 数据读出
always @ (posedge clk)    if (RegWrite)    regs[WriteRegister] <= WriteData; //数据写入
```

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 30.8 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30.9 所示。图中的 RBW_Registers 模块就是实现图 30.8 的 Read Before Write 寄存器堆。图中转发检测电路的输出表达式为

rs1Sel= RegWrite && (WriteAddr !=0) && (WriteAddr ==rs1Addr)　　(30.1)

rs2Sel= RegWrite && (WriteAddr !=0) && (WriteAddr ==rs2Addr)　　(30.2)

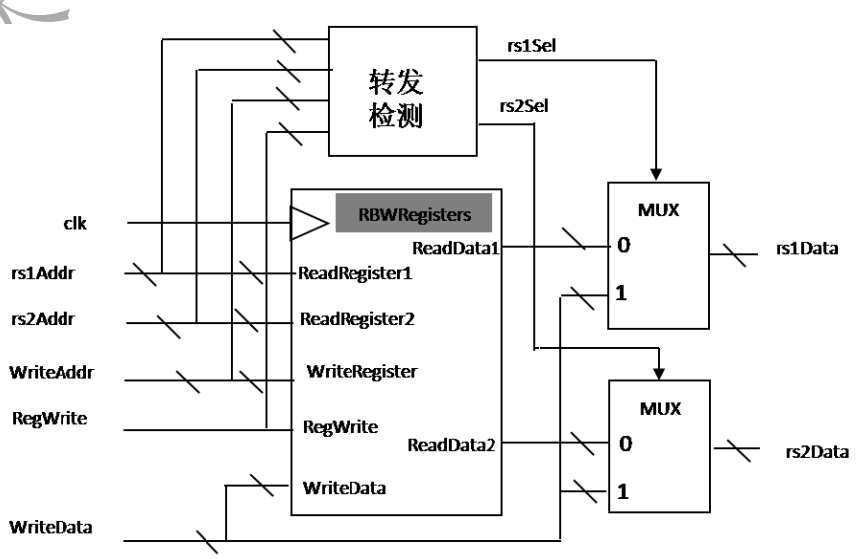

图 30.9 具有 Read After Write 特性寄存器堆的原理框图

（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 offet。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。仔细分析表 30.2 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。具体做法：

- R_type 类：操作码（opcode，简称 op）为 7’h33，R 类的所有指令，两个操作数分别为 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，两个操作数分别为 rs1 和立即数 imm；
- LW 指令：操作码 7’h03，I 类的数据传送指令 lw，两个操作数分别为 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，两个操作数分别为 PC 和常数 4；
- SW 指令：操作码 7’h23，S 类的数据传送指令 sw，两个操作数分别为 rs1 和立即数 imm；
- SB_ type 类：操作码 7’h63，SB 类的所有指令，两个操作数分别为 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 和常数 4。

因此，设置 R_type、I_type、SB_ type、LW、JALR、SW、LUI、AUIPC 和 JAL 等变量来表示指令类型，各变量的值由式 30.3 决定。

$$\begin{cases} R\_type = (op == R\_type\_op) \\ I\_type = (op == I\_type\_op) \\ SB\_type = (op == SB\_type\_op) \\ LW = (op == LW\_op) \\ JALR = (op == JALR\_op) \\ SW = (op == SW\_op) \\ LUI = (op == LUI\_op) \\ AUIPC = (op == AUIPC\_op) \\ JAL = (op == JAL\_op) \end{cases} \tag{30.3}$$

①只有 LW 指令读取存储器且回写数据取自存储器，所以有

$$MemtoReg\_id=LW \tag{30.4}$$

$$MemRead\_id =LW \tag{30.5}$$

② 只有 SW 指令会对存储器写数据，所以有

$$MemWrite\_id =SW \tag{30.6}$$

③ 需要进行回写的指令类型有 R_type、I_type、LW、JALR、LUI、AUIPC 和 JAL。所以有

$$RegWrite\_id =R\_type \,||\, I\_type \,||\, LW \,||\, JALR \,||\, LUI \,||\, AUIPC \,||\, JAL \tag{30.7}$$

④ 只有 JALR 和 JAL 两条无条件分支指令，所以有

$$Jump=JALR \,||\, JAL \tag{30.8}$$

⑤操作数 A 和 B 的选择信号的确定

分析各类指令，可得到表 30.4 操作数选择的功能表。

表 30.4　操作数选择信号的功能表

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

从表 30.4 可获得 ALUSrcA_id 和 ALUSrcB_id[1:0]表达式。

$$\text{ALUSrcA\_id} = \text{JALR} \ \| \ \text{JAL} \ \| \ \text{AUIPC} \tag{30.9}$$

$$\begin{cases} \text{ALUSrcB\_id}[1] = \text{JAL} \| \text{JALR} \\ \text{ALUSrcB\_id}[0] = \sim(\text{R\_type} \| \text{JAL} \| \text{JALR}) \end{cases} \tag{30.10}$$

⑥ALUCode 的确定

除了条件分支指令，其它指令都需要 ALU 执行运算，共有 11 种不同运算， ALUCode 信号需用 4 位二进制表示。最主要为加法运算，设为默认算法，ALUCode 的功能表如表 30.5 所示。注意：表中 funct7[6]与 funct6[5]在指令中为同一位置，即 instruction[30]。

表 30.5　ALUCode 的功能表

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

⑦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。Shift 值由式(30.11)计算。

Shift=（funct3==1）||（funct3==5）　　(30.11)

立即数构成和扩展方法如表 30.6 所示，表中的 inst 即 instruction。

表 30.6　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}

（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。

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

②确定比较运算的结果。对于比较运算来说，如果最高位不同，即 rs1Data [31]≠ rs2Data [31]，可根据 rs1Data [31]、rs2Data [31]决定比较结果，但是应注意符号数、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，rs1Data<rs2Data 有以下两种情况：

a) rs1Data 为负数、rs2Data 为 0 或正数：rs1Data[31]&&(~rs2Data[31])

b) rs1Data、rs2Data 符号相同，sum 为负：(rs1Data[31]~^rs2Data[31]) && sum[31]

因此，符号数 rs1Data<rs2Data 比较运算结果为

$$isLT=rs1Data[31]\&\&(\sim rs2Data[31]) \| (rs1Data[31]\sim\hat{}\,rs2Data[31])\&\& sum[31] \tag{30.12}$$

同样地，无符号数比较运算中，rs1Data<rs2Data 有以下两种情况：

a) rs1Data 最高位为 0、rs2Data 最高位为 1：(~rs1Data[31])&& rs2Data[31]

b) rs1Data、rs2Data 最高位相同，sum 为负：(rs1Data[31]~^rs2Data[31]) && sum[31]

因此，无符号数比较运算结果为

$$isLTU= (\sim rs1Data[31])\&\& rs2Data[31] \| (rs1Data[31] \sim\hat{}\,rs2Data[31]) \&\& sum[31] \tag{30.13}$$

最后用数据选择器完成式(30.14)即可完成分支检测。

$$Branch = \begin{cases} \sim(|sum[31:0]); & SB\_type \ \&\& \ (funct3 == beq\_funct3) \\ |sum[31:0]; & SB\_type \ \&\& \ (funct3 = bne\_funct3) \\ isLT; & SB\_type \ \&\& (funct3 = blt\_funct3) \\ \sim isLT; & SB\_type \ \&\& \ (funct3 = bge\_funct3) \\ isLTU; & SB\_type \ \&\& \ (funct3 = bltu\_funct\ 3) \\ \sim isLTU; & SB\_type \ \&\& \ (funct3 = bgeu\_funct\ 3) \\ 0 & others \end{cases} \tag{30.14}$$

(4) 冒险检测功能电路（Hazard Detector）的设计

由前面分析可知，冒险成立的条件为：

① 上一条指令必须是 lw 指令（MemRead_ex=1）；

② 两条指令读写同一个寄存器（rdAddr_ex=rs1Addr_id 或 rdAddr_ex=rs2Addr_id）。

当冒险成立应清空 ID/EX 寄存器并且阻塞流水线 ID 级、IF 级流水线，所以有

Stall=((rdAddr_ex==rs1Addr_id) || (rdAddr_ex==rs2Addr_id)) && MemRead_ex (30.15)

IFWrite= ~Stall (30.16)

在用 VerilogHDL 描述 ID 模块时，冒险检测功能电路（Hazard Detector）等功能单元比较简单，建议直接在 ID 顶层描述。

2. 执行模块（EX）的设计

执行模块主要由 ALU 子模块、数据前推电路（Forwarding）及若干数据选择器组成。执行模块的接口信息如表 30. 7 所示。

表 30. 7　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]		

（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。整理表 30.5 所示的 ALUCode 的功能表，可得到 ALU 的功能表，如表 30.8 所示。

表 30.8　ALU 的功能表

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

如表 30.8 所示，ALU 需执行多种运算，为了提高运算速度，本设计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30.10所示。

①加、减电路的设计考虑

减法、比较（slt、sltu）均可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) (30.17)

最后要强调的是，32 位加法器的运算速度决定了 RISC-V 微处理器的时钟信号频率的高低，因此设计一个高速的 32 位加法器尤为重要。32 位加法器可采用实验 8 介绍的进位选择加法器。

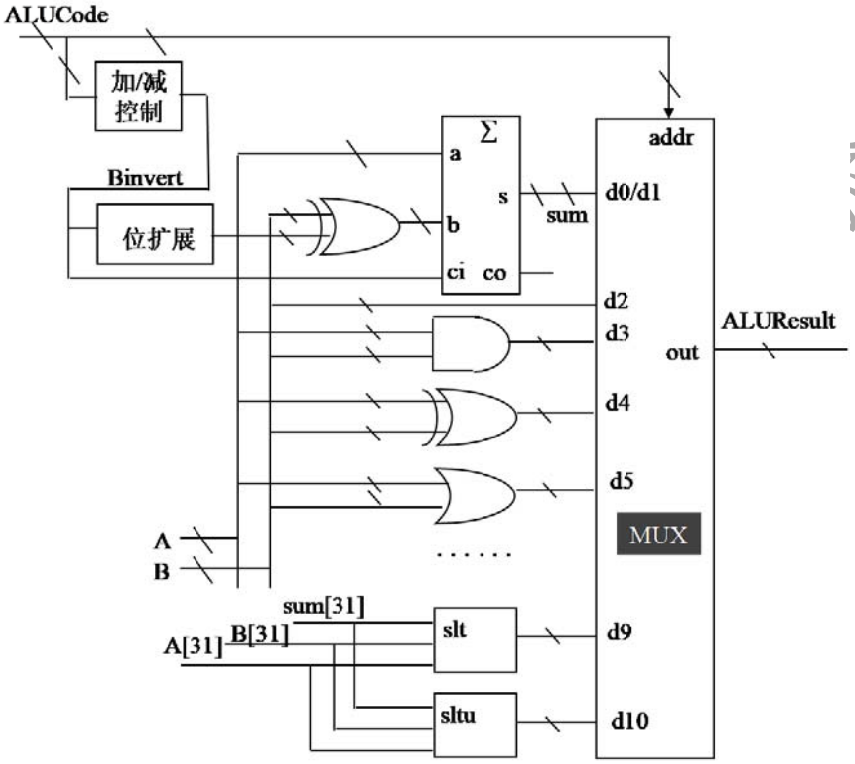

图 30.10 ALU 结构框图

②比较电路的设计考虑

比较电路的设计方法已在分支检测电路介绍，参考式(30.12)和式(30.13)可确定 slt 和 sltu 两条件的比较结果。

③算术右移运算电路的设计考虑

算术右移对有符号数而言，移出的高位补符号位而不是 0。每右移一位相当于除以 2。例如，有符号数负数 10100100(-76)算术右移两位结果为 11101001(-19)，正数 01100111(103)算术右移一位结果为 00110011(51)。

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。

（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 30.9 所示。

表 30.9　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

由前面介绍的一、二阶数据相关判断条件，不难得到

$$\begin{cases} \text{ForwardA}[0] = \text{RegWrite\_wb} \,\&\&\, (\text{rdAddr\_wb} != 0) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_mem} != \text{rs1Addr\_ex}) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_wb} == \text{rs1Addr\_ex}) \\ \text{ForwardA}[1] = \text{RegWrite\_mem} \,\&\&\, (\text{rdAddr\_mem} != 0) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_mem} == \text{rs1Addr\_ex}) \end{cases} \tag{30.18}$$

$$\begin{cases} \text{ForwardB}[0] = \text{RegWrite\_wb} \,\&\&\, (\text{rdAddr\_wb} != 0) \,\&\&\, \\ \qquad\qquad (\text{rd\_mem} != \text{rs2Addr\_ex}) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_wb} == \text{rs2Addr\_ex}) \\ \text{ForwardB}[1] = \text{RegWrite\_mem} \,\&\&\, (\text{rdAddr\_mem} != 0) \,\&\&\, \\ \qquad\qquad (\text{rdAddr\_mem} == \text{rs2Addr\_ex}) \end{cases} \tag{30.19}$$

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 bit 的单端口 RAM，输出采用组合输出（Non Registered）。

由于数据存储器容量为 64×32 bit，故存储器地址共 6 位，与 ALUResult_mem[7:2]连接。

4．取指令级模块（IF）的设计

IF 模块由指令指针寄存器（PC）、指令存储器子模块（Instruction ROM）、指令指针选择器（MUX）和一个 32 位加法器组成，IF 模块接口信息如表 30.10 所示。

表 30.10　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

由图 30.3 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。作者提供一个指令存储器模块 InstructionROM.v， ROM 内存放一段简单测试程序的机器码，对应的测试程序为：

```
          lui  X30, 0x3000
          jalr X31, later(X0)
earlier:sw    X28, 0x0C(X0)
          lw   X29, 4(X6)
          slli  X5,  X29, 2       //数据冒险
          lw   X28, 4(X6)
          sltu X28, X6,X7
done:     jal  X31, done
later:    bne  X0,  X0, end       // 分支条件不成立
          addi X5,  X30, 0x42
          add  X6,  X0, X31
          sub  X7,  X5, X6        //操作 A 二阶数据相关，操作 B 一阶数据相关
          or   X28, X7, X5        //操作 A 一阶数据相关，操作 B 三阶数据相关
          beq  X0,  X0, earlier   // 分支条件成立
end:      nop
```

5．流水线寄存器的设计

流水线寄存器负责将流水线的各部分分开，共有 IF/ID、ID/EX、EX/MEM、MEM/WB 四组，对四组流水线寄存器要求不完全相同，因此设计也有不同考虑。

EX/MEM、MEM/WB 两组流水线寄存器只是普通的 D 型寄存器。

当流水线发生数据冒险时，需要清空 ID/EX 流水线寄存器而插入一个气泡，因此 ID/EX 流水线寄存器是一个带同步清零功能的 D 型寄存器。

当流水线发生数据冒险时，需要阻塞 IF/ID 流水线寄存器；若跳转指令或分支成立，则还需要清空 ID/EX 流水线寄存器。因此，IF/ID 流水线寄存器除同步清零功能外，还需要具有保持功能（即具有使能 EN 信号输入）。

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。顶层文件的端口列表描述如下。

```
module Risc5CPU(clk, reset, JumpFlag, Instruction_id, ALU_A,
                ALU_B, ALUResult_ex, PC, MemDout_mem,Stall);
    input    clk;
    input    reset;
    output[1:0] JumpFlag;
    output [31:0] Instruction_id;
    output [31:0] ALU_A;
    output [31:0] ALU_B;
    output [31:0] ALUResult_ex;
    output [31:0] PC;
    output [31:0] MemDout_mem;
    output Stall;
```

四、提供的文件

1. 指令译码模块（decode）的测试代码 Decode_tb.v、ALU 测试代码 ALU_tb.v、取指令级模块（IF）的测试代码 IF_tb.v 和 CPU 的顶层测试代码 Risc5CPU_tb .v。

2.指令存储器（InstructionROM）的 Verilog HDL 代码 InstructionROM.v。

3. 硬件测试 CPU 的 Vivado 架构，在该架构中，用带有 HDMI 接口的显示器显示 CPU 运行的内部重要变量。测试架构将程序指针的低 8 位 PC[7:0]、指令机器码 Instruction_id、流水线气泡插入标志 Stall、跳转标记 JumpFlag、ALU 操作数 A 及 B、ALU 结果 ALUResult 和数据存储输出 MemDout_mem 等送入带有 HDMI 接口的显示器，显示格式如图30.11 所示。

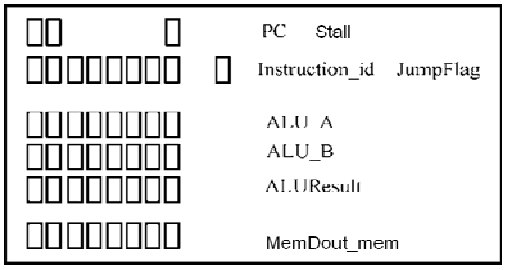

图 30.11 VGA 显示格式

当完成整个流水线 CPU 的设计与仿真测试，打开 Vivado 文件夹下的 Risc5CPU.xpr 工程，添加流水线 CPU 设计代码（包括子模块代码），然后综合、实现和下载至开发板，连接带有 HDMI 接口的显示器，就可进行测试。

五、实验设备

（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。

六、预习内容

（1）查阅相关书籍，掌握流水线 RISC-V CPU 工作原理。
（2）查阅相关书籍或资料，理解流水线数据相关和数据冒险的概念，掌握数据相关和数据冒险的解决方法。

七、实验内容

（1）从网络下载相关文件。
（2） 编写指令译码单元 Decode 模块的 Verilog HDL 代码，并用 ModelSim 进行功能仿真。
（3）编写寄存器堆 Register 模块的 Verilog HDL 代码。
（4）编写 ID 模块 Verilog HDL 代码，ID.v 文件已给端口列表。
（5）编写 ALU 模块的 Verilog HDL 代码并用 ModelSim 进行功能仿真。
（6）编写执行单元 EX 模块的 Verilog HDL 代码，EX.v 文件已给端口列表。
（7）编写 IF 模块的 Verilog HDL 代码并用 ModelSim 进行功能仿真。
（8）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。
（9）编写 CPU 顶层的 Verilog HDL 代码，并用 ModelSim 进行功能仿真。注意：由于存在 IP 内核，仿真时，需加仿真库，方法参考实验 3.

根据表 30.11 验证仿真结果。表格中显示的数据均为十六进制，表格中“-”表示此处值无意义。

表 30.11　测试程序的运行结果

reset	clk	PC	Instruction（ID）	JumpFlag	Stall	ALU_A	ALU_B	ALUResult	MemDout（MEM）
1	1	0	0	0	0	0	0	0	-
0	2	4	00003f37 (lui)	0	0	-	-	-	-
	3	8	02000fe7 (jalr)	2	0	-	3000	3000	-
	4	20	0	0	0	4	4	8	-
	5	24	00001c63 (bne)	0	0	--	-	-	-
	6	28	042f0293 (addi)	0	0	-	-	-	-
	7	2c	01f00333 (add)	0	0	3000	42	3042	-
	8	30	406283b3 (sub)	0	0	0	8	8	-
	9	34	0053ee33 (or)	0	0	3042	8	303a	-
	10	38	fc000ae3 (beq)	1	0	303a	3042	307a	-
	11	8	0	0	0	-	-	-	-
	12	c	01c02623 (sw)	0	0	-	-	-	-
	13	10	00432e83 (lw)	0	0	0	c	c	-
	14	14	002e9293 (sll)	0	1	8	4	c	-
	15			0	0	-	-	-	307a
	16	18	00432e03 (lw)	0	0	307a	2	c1e8	-
	17	1c	00733e33 (sltu)	0	0	8	4	c	-
	18	20	0000f6f (jal)	2	0	8	303a	1	307a
	19	1c	0	0	0	1c	4	20	-
	20	20	0000f6f (jal)	2	0	-	-	-	-
	21	1c	0	0	0	1c	4	20	-

（10）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。

（11）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，对照表 30.11 验证设计是否正确。

注意：一般设计都采用同步复位，因此在“单步”运行模式时，如需要复位 CPU，应先按住“复位”按键（中间按键），再按上边按键才能复位。

八、实验报告要求

（1）写出设计原理、列出 Verilog HDL 代码并对设计作适当说明。

（2）记录 ModelSim 仿真波形，并对仿真波形作适当解释，分析是否符合预期功能。

（3）记录实验结果，分析设计是否正确。

（4）记录实验中碰到的问题和解决方法。

九、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
```