浙江大学

本科实验报告

课程名称：	计算机组成与设计
姓　　名：	王若鹏
学　　院：	信息与电子工程学院
专　　业：	电子科学与技术
学　　号：	3170105582
指导教师：	屈民军 唐奕
上课时间：	双周周四 1, 2 节

2019 年 11 月 22 日

专业：电子科学与技术
姓名：王若鹏
学号：3170105582
日期：2019.11.22
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：设计型

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、实验要求

设计一个流水线 RISC 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、begu

⑤ 无条件跳转指令：jal、jalr

⑥ 数据传送指令：lw、sw、lui、auipc

⑦ 空指令：nop

(2) 采用 5 级流水线技术，对数据冒险实现转发或阻塞功能。

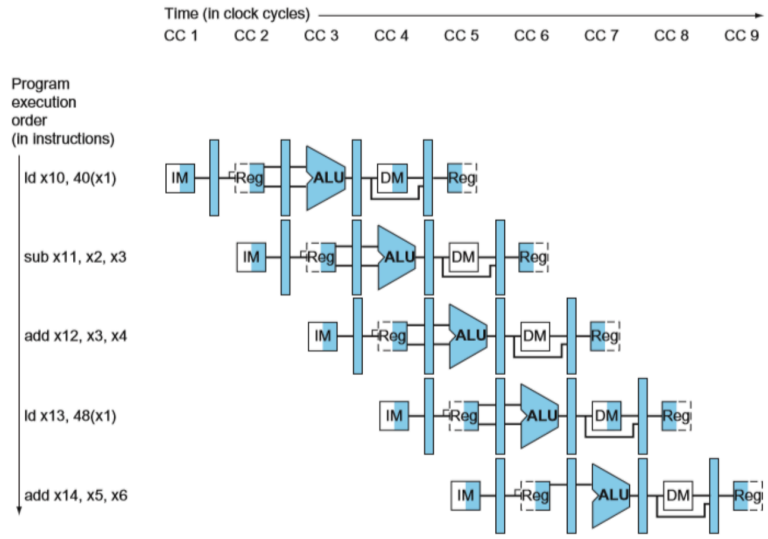
(3) 在 Nexys Video 开发系统中实现 RISC-V 微处理器，要求 CPU 的运行速度大于 25MHz。

三、实验原理

3.1 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，广泛应用于高档 CPU 的

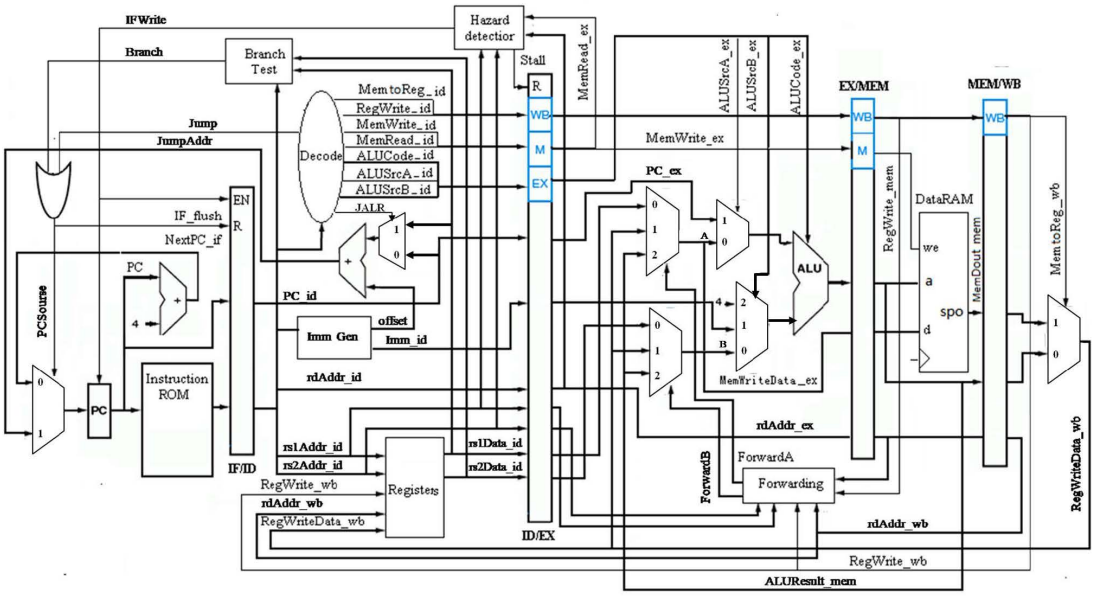
架构中。根据 RISC-V 处理器指令的特点，将指令整体的处理过程分为取指令(IF)、指令译码(ID)、执行(EX)、存储器访问(MEM)和寄存器回写(WB)五级。如图 1 所示，一个指令的执行需要 5 个时钟周期，每个时钟周期的上升沿来临时，此指令所代表的一系列数据和控制信息将转移到下一级处理。



（图 1：流水线流水作业示意图）

3.1.1 原理框图

图 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。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，这样的命名方式起到了很好的识别作用。



（图 2：流水线 RISC-V 微处理器的原理框图）

3.1.2 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：

控制信号	信号作用
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

（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。

（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：

控制信号	信号作用
ALUCode	确定 ALU 操作
ALUSrcA、ALUSrcB	选择两个 ALU 操作数 ALU_A、ALU_B
ForwardA、ForwardB	数据转发控制信号

（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：

控制信号	信号作用
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
RegWrite	寄存器写操作允许信号

3.1.3 数据相关与数据转发

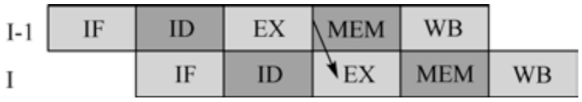
如果上一条指令的结果还没有写入到寄存器中，而下一条指令的源操作数又恰恰是此寄存器的数据，那么它所获得的将是原来的数据，而不是更新后的数据。这样的相关问题称为数据相关。在设计中，采用数据转发和插入流水线气泡的方法解决此类相关问题。

（1）一阶数据相关与转发（EX 冒险）

如图 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。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，导致操作数的一阶数据相关判断的条件为:

判断条件	RegWrite_mem	rdAddr_mem	rdAddr_mem
操作数 A	= 1	≠ 0	= rs1Addr_ex
操作数 B	= 1	≠ 0	= rs2Addr_ex
含义	MEM 级阶段必须是写操作	目标寄存器不是 X0 寄存器	两条指令读写同一个寄存器

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 3 所示。



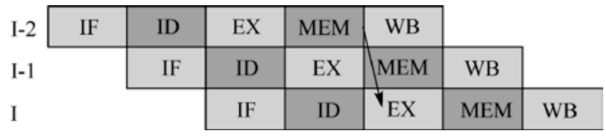
（图 3：一阶前推网络示意图）

（2）二阶数据相关与转发（MEM 冒险）

如图 4 所示，如果第 I 条指令的源操作寄存器与第 I−2 条指令的目标寄存器相重，将导致二阶数据 相关。导致操作数的二阶数据相关必须满足下列条件:

判断条件	RegWrite_wb	rdAddr_wb	rdAddr_mem	rdAddr_wb
操作数 A	= 1	≠ 0	≠ rs1Addr_ex	= rs1Addr_ex
操作数 B	= 1	≠ 0	≠ rs2Addr_ex	= rs2Addr_ex
含义	WB 级阶段必须是写操作	目标寄存器不是 X0 寄存器	一阶数据相关条件不成立	两条指令读写同一个寄存器

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 4 所示。



（图 4：二阶前推网络示意图）

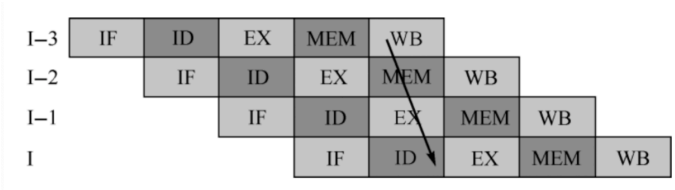
（3）三阶数据相关

图 5 所示为第 I 条指令与第 I−3 条指令的数据相关问题，即在同一个周期内同时读写同

一个寄存器，将导致三阶数据相关。导致操作数的三阶数据相关必须满足下列条件：

判断条件	RegWrite_wb	rdAddr_wb	rdAddr_wb
操作数 A	= 1	≠ 0	= rs1Addr_id
操作数 B	= 1	≠ 0	= rs2Addr_id
含义	寄存器必须是写操作	目标寄存器不是 X0 寄存器	读写同一个寄存器

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 5：三阶前推网络示意图）

3.1.4 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。

3.2 基本模块设计

根据流水线不同阶段，将系统划分为 IF、ID、EX 和 MEM 四大模块。

3.2.1 IF 级（取指令模块）的设计

IF 模块由指令指针寄存器(PC)、指令存储器子模块(Instruction ROM)、指令指针选择器

(MUX)和一个 32 位加法器组成，IF 模块接口信息如图 6 所示。指令存储器为组合存储器，可用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。其中流水线清空信号 IF_flush = Jump || Branch。

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]		分支指令跳转地址
Instruction [31:0]	Output	指令机器码
IF_flush		流水线清空信号
PC [31:0]		PC 值

（图 6：IF 模块接口信息）

3.2.2 ID 级（指令译码模块）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 模块的接口信息如下图所示：

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]		条件分支地址
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]		

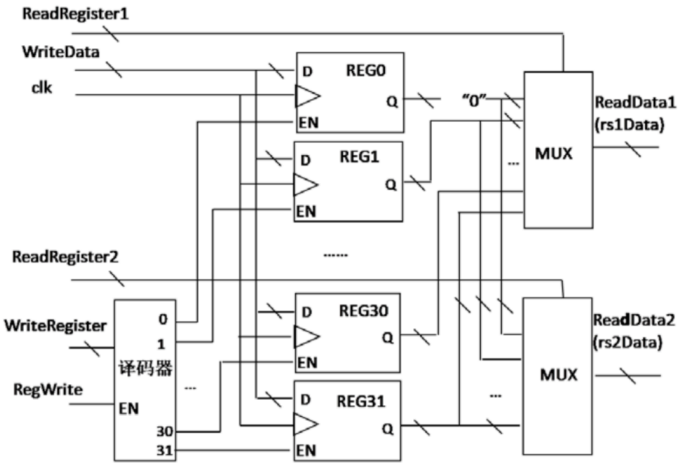
（图 7：ID 模块的接口信息）

（1）寄存器堆子模块的设计（Registers.v）

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 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。

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 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。用 Verilog 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：

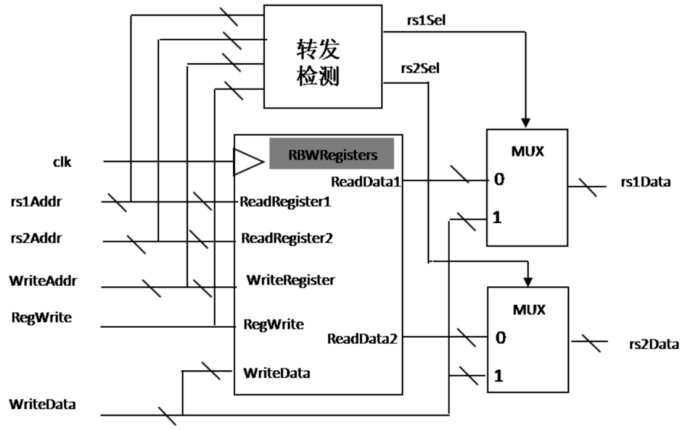
```
reg [31:0]    regs [31:0]; // Memory declaration 32*32
assign ReadData1 = (ReadRegister1 == 5'b0)? 32'b0 : regs[ReadRegister1];
assign ReadData2 = (ReadRegister2 == 5'b0)? 32'b0 : regs[ReadRegister2];
always @ (posedge clk)    if (RegWrite)        regs[WriteRegister] <= WriteData;
```



（图 8：寄存器堆的原理框图）

在流水线型 CPU 设计中，寄存器堆设计还应解决三阶数据相关的数据转发问题。当满足三阶数据相关条件时，寄存器具有 Read After Write 特性。设计时，只需要在图 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 9 所示。图中的 RBW_Registers 模块就是实现图 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);
rs2Sel = RegWrite && (WriteAddr !=0) && (WriteAddr == rs2Addr);
```

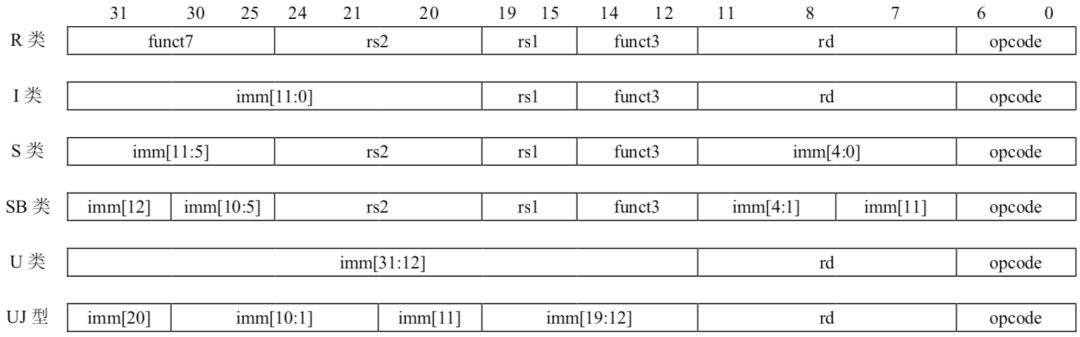


（图 9：具有 Read After Write 特性寄存器堆的原理框图）

（2）指令译码子模块的设计（Decode.v）

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 六类。指令格式中的指令操作码(opcode)、源操作数的寄存器号(rs1、rs2)、目标寄存器号(rd)、 指令操作扩展码(funct3、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。



（图 10：RV32I 指令格式）

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，再次细分指令如下：

指令类型	操作码	对应具体指令	操作数 1	操作数 2
R_type	7’h33	R 类的所有指令	rs1	rs2
I_type	7’h13	I 类的算术逻辑运算指令和移位指令	rs1	imm
LW	7’h03	I 类的数据传送指令 lw	rs1	imm
JALR	7’h67	I 类的无条件分支指令 jalr	PC	4
SW	7’h23	S 类的数据传送指令 sw	rs1	imm
SB_type	7’h63	SB 类的所有指令	PC	imm

LUI	7'h37	U 类的数据传送指令 lui	imm	/
AUIPC	7'h17	U 类的数据传送指令 auipc	PC	imm
JAL	7'h6F	UJ 类的无条件分支指令 jal	PC	4

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;　MemRead_id = LW;

② 只有 SW 指令会对存储器写数据，所以有 MemWrite_id = SW;

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

RegWrite_id = R_type || I_type || LW || JALR || LUI || AUIPC || JAL;

④ 只有 JALR 和 JAL 两条无条件分支指令，所以有 Jump = JALR || JAL;

⑤ 操作数 A 和 B 的选择信号的确定分析各类指令，可得到：

ALUSrcA_id = JALR || JAL || AUIPC;

ALUSrcB_id[1] = JAL || JALR;　ALUSrcB_id[0] = ~(R _ type || JAL || JALR);

⑥ ALUCode 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 的功能表如图 11：

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 11：ALUCode 的功能表）

⑦ 立即数产生电路

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 = (funct3==1) || (funct3==5);

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 12：立即数产生方法）

（3）分支检测电路的设计（Branch_test.v）

分支检测电路主要用于判断分支条件是否成立，可以用比较运算符号描述，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：用一个 32 位加法器完成 rs1Data-rs2Data，设结果为 sum [31:0]；确定比较运算的结果，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：①rs1Data 为负数、rs2Data 为 0 或正数；②rs1Data、rs2Data 符号相同，sum 为负。因此，符号数 rs1Data<rs2Data 比较运算结果为：

isLT = rs1Data[31] && (~rs2Data[31]) || (rs1Data[31]~^rs2Data[31]) && sum[31];

无符号数比较运算中，rs1Data<rs2Data 有以下两种情况：①rs1Data 最高位为 0、rs2Data 最高位为 1；②rs1Data、rs2Data 最高位相同，sum 为负。因此，无符号数比较运算结果为：

isLTU = (~rs1Data[31]) && rs2Data[31] || (rs1Data[31] ~^rs2Data[31]) && sum[31];

最后用数据选择器即可完成分支检测：

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

（4）冒险检测功能电路的设计（Hazard_Detector.v）

冒险成立的条件为：①上一条指令必须是 lw 指令；②两条指令读写同一个寄存器。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;

IFWrite = ~Stall;

3.2.3 EX 级（执行模块）的设计

执行模块主要由 ALU 子模块、数据前推电路(Forwarding)及若干数据选择器组成。执行模块的接口信息如图 13 所示：

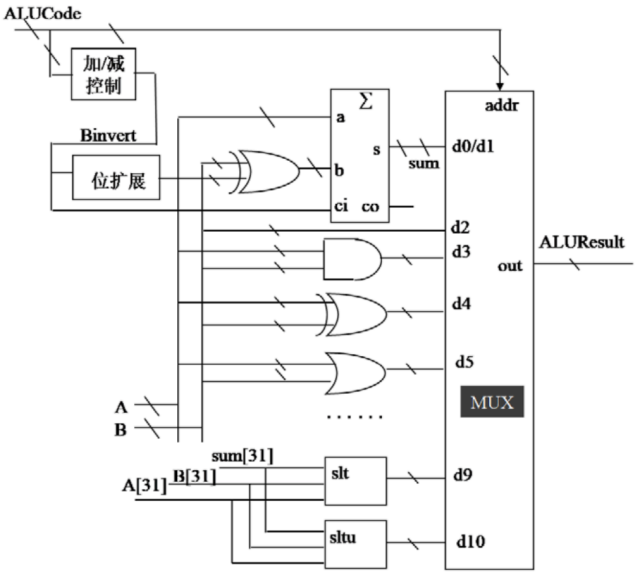
引脚名称	方向	说明
ALUCode_ex[3:0]	Input	决定 ALU 采用何种运算
ALUSrcA_ex		决定 ALU 的 A 操作数的来源（rs1、PC）
ALUSrcB_ex[1:0]		决定 ALU 的 B 操作数的来源(rs2、imm 和常数 4)
Imm_ex[31:0]		立即数
rs1Addr_ex[4:0]		rs1 寄存器地址
rs2Addr_ex[4:0]		rs2 寄存器地址
rs1Data_ex[31:0]		rs1 寄存器数据
rs2Data_ex[31:0]		rs2 寄存器数据
PC_ex[31:0]		指令指针
RegWriteData_wb[31:0]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

（图 13：EX 模块的接口信息）

（1）算术逻辑单元的设计（ALU.v）

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。整理图 11 所示的 ALUCode 的功能表，可得到 ALU 的功能表。

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 14 所示：



（图 14：ALU 结构框图）

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 14 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);

② 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。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：

```
reg signed [31:0]    A_reg;
always @(*)    begin    A_reg=A;    end
```

（2）前推电路的设计（forward.v）

操作数 A 和 B 分别由数据选择器决定，地址信号 ForwardA、ForwardB 的含义如下：

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 15：前推电路输出信号的含义）

由“3.1.3 数据相关与数据转发”中一、二阶数据相关判断条件，不难得到：

$$\begin{cases} ForwardA[0] = RegWrite\_wb \&\& (rdAddr\_wb != 0) \&\& \\ \qquad (rdAddr\_mem != rs1Addr\_ex) \&\& \\ \qquad (rdAddr\_wb == rs1Addr\_ex) \\ ForwardA[1] = RegWrite\_mem \&\& (rdAddr\_mem != 0) \&\& \\ \qquad (rdAddr\_mem == rs1Addr\_ex) \end{cases} \quad \begin{cases} ForwardB[0] = RegWrite\_wb \&\& (rdAddr\_wb != 0) \&\& \\ \qquad (rd\_mem != rs2Addr\_ex) \&\& \\ \qquad (rdAddr\_wb == rs2Addr\_ex) \\ ForwardB[1] = RegWrite\_mem \&\& (rdAddr\_mem != 0) \&\& \\ \qquad (rdAddr\_mem == rs2Addr\_ex) \end{cases}$$

3.2.4 MEM 级（数据存储器模块）的设计

数据存储器 DataRAM 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。

3.2.5 WB 级（写回模块）的设计

WB 级较简单，只有一个数据选择器 RegWriteData_mux，因此可以放在顶层中实现。

3.3 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)。

综上，流水线寄存器的设计要求为：

流水线寄存器	使能信号	清零信号
IF/ID	IFWrite	IF_flush || reset
ID/EX	1	Stall || reset
EX/MEM	1	reset
MEM/WB	1	reset

3.4 顶层设计（Risc5CPU.v）

按照图 2 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。

四、主要仪器设备

（1）装有 Vivado 和 Modelsim SE 软件的计算机

（2）Nexus Video 开发板一套

（3）带有 HDMI 接口的显示器一台

五、实验步骤与内容

（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 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，验证仿真结果.

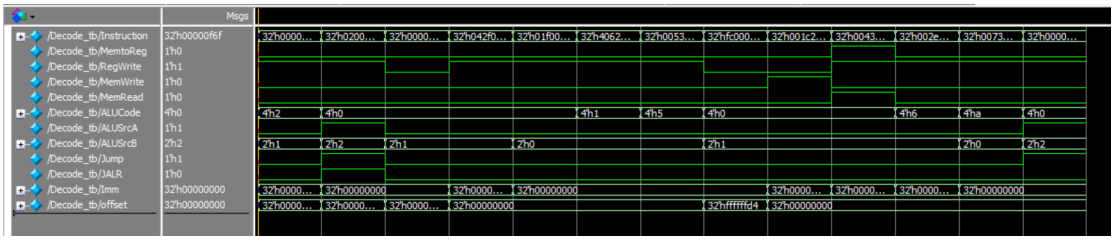
（10）再次打开 Vivado 文件夹下的 Risc5CPU.xpr 工程，添加流水线 CPU 设计的全部代码，然后综合、实现和下载至 Nexys Video 开发板。

（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，对照讲义上的表格验证设计是否正确。

六、实验数据记录与分析

6.1 Decode 仿真分析

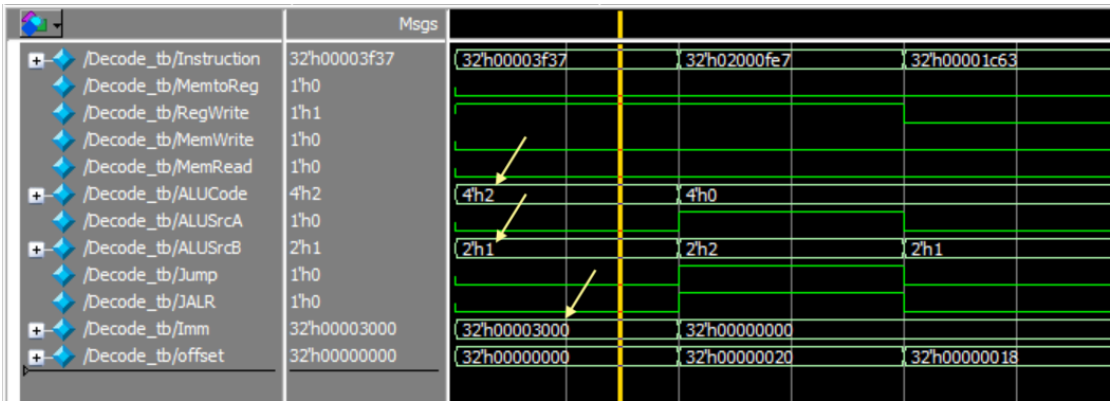
Decode 模块仿真波形总体如下：



（图 16：Decode 仿真波形，整体）

观察具体的指令控制波形，此处以 32'h00003f37（lui　X30, 0x3000）为例进行分析：该指令波形如图所示。该指令的 ALUCode=0010，MemWrite=0,，MemRead=0，Regwrite=1，MemtoReg=0，ALUSrcA=0，ALUSrcB=1，Imm=3000。该指令的译码结果符合要求。同时观察其他指令，结果皆符合要求。

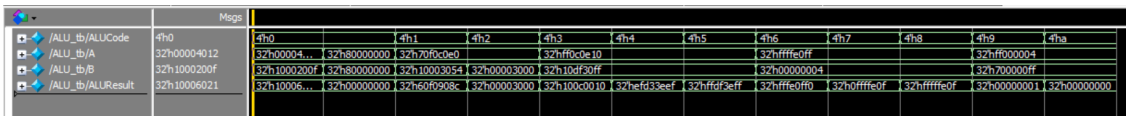
可知 Decode 模块设计成功。



（图 17：Decode 仿真波形，局部）

6.2 ALU 仿真分析

ALU 模块仿真波形如下：



（图 18：ALU 仿真波形）

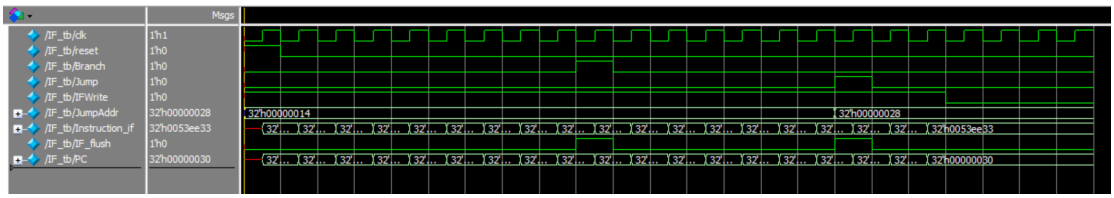
将将测试波形 A 与 B 列成表格，如下表所示，最后一列为手动计算结果。经过与图中的 ALUResult 波形数值相比较，完全符合结果。

可知 ALU 模块设计成功。

ALUCode	指令	A (32'h)	B (32'h)	ALUResult (32'h)
4'h0 (0000)	add	00004012	1000200f	10006021
4'h0 (0000)	add	80000000	80000000	00000000
4'h1 (0001)	sub	70f0c0e0	10003054	60f0908c
4'h2 (0010)	lui	70f0c0e0	00003000	00003000
4'h3 (0011)	and	ff0c0e10	10df30ff	100c0010
4'h4 (0100)	xor	ff0c0e10	10df30ff	efd33eef
4'h5 (0101)	or	ff0c0e10	10df30ff	ffdf3eff
4'h6 (0110)	sll	ffffe0ff	00000004	fffe0ff0
4'h7 (0111)	srl	ffffe0ff	00000004	0ffffe0f
4'h8 (1000)	sra	ffffe0ff	00000004	fffffe0f
4'h9 (1001)	slt	ff000004	700000ff	00000001
4'ha (1010)	sltu	ff000004	700000ff	00000000

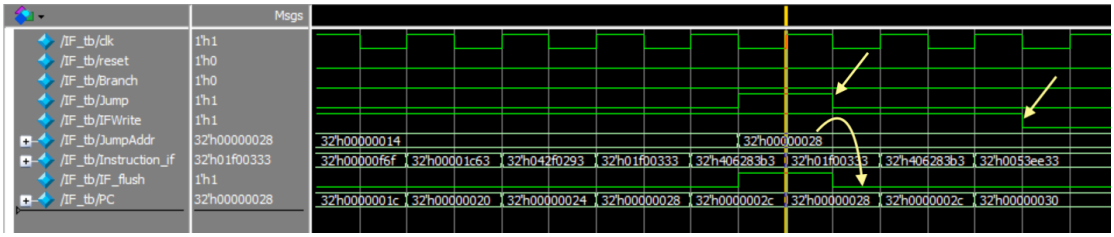
6.3 IF 仿真分析

IF 模块仿真波形如下：



（图 19：IF 模块仿真波形，整体）

为了便于分析，对 Jump=1 和 IFWrite=0 区域进行局部放大：



（图 20：IF 模块仿真波形，局部）

综合以上波形图进行分析:

(1) Instruction_if 里的内容与 Instruction ROM 中 PC 相对应内容相同，功能正常。

(2) reset 高电平时，PC 置零，同时每一轮周期加 4，reset 功能正常。

(3) 在 clk 上升沿，PC 值发生跳变，功能正常。

(4) 在 Jump 或 Branch 分别变为高电平后 IF_flush 变为高电平，PC 值变为对应的 Addr，可见指令指针选择器功能正常。

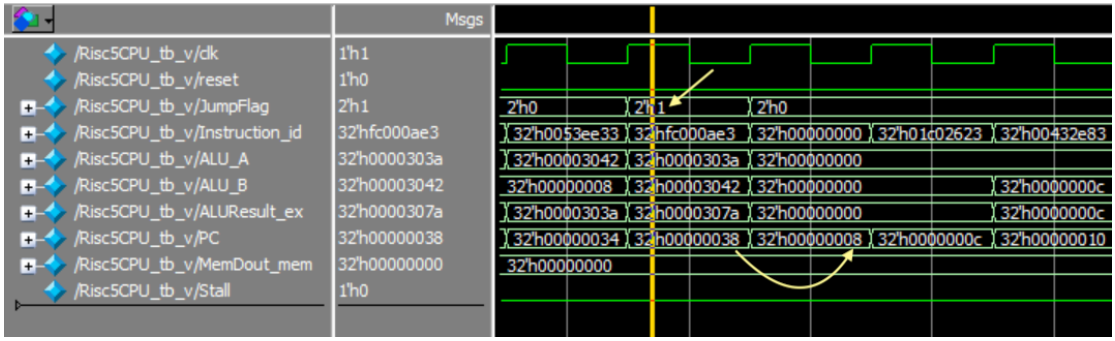
(5) IFWrite 变为低电平后指针不再跳转，阻塞流水线信号功能正常。

可知 IF 模块设计成功。

6.4 顶层仿真分析

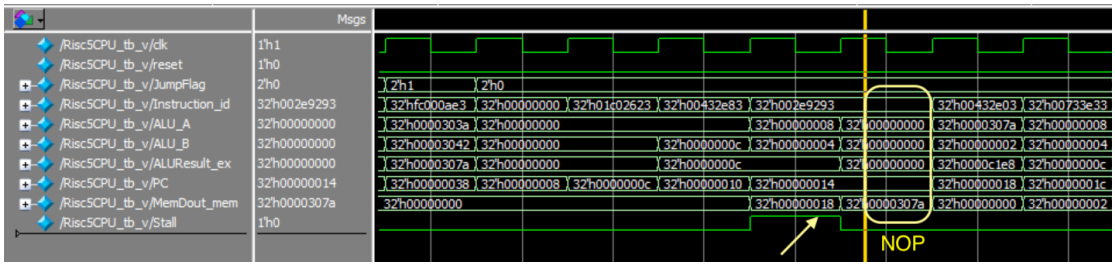
为分析功能是否实现，先列出两幅局部波形。

当运行至“beq X0, X0, earlier”时，beq 条件成立，PC 从 38 跳转至 earlier 处（PC=8）：



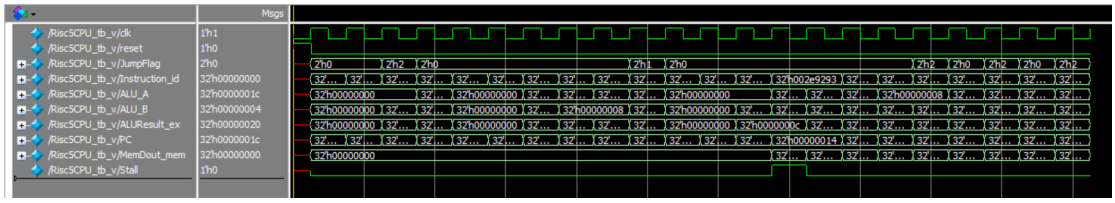
（图 21：顶层仿真波形局部图，跳转）

当运行至“lw X29, 4(X6)；sll X5, X29, 2”时，Stall=1，在流水线中插入空泡 NOP：



（图 22：顶层仿真波形局部图，NOP）

顶级总体波形如图 23 所示。与讲义中表格相比对，运行结果完全相符。可知基于 RV32I 指令集的 RISC-V 微处理器**设计成功**。



（图 23：顶层仿真波形整体图）

七、实验结果

在 Vivado 工程上添加全部代码，综合、实现后，下载至开发板，连接至显示器。经指导老师验收，结果与讲义上的表格完全一致，实验成功完成。

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	001c2623 (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)	0	0	-	-	-	-
	21	1c	0	0	0	1c	4	20	-

（图 24：测试程序运行结果）

八、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

答：可以采用“分支预测”的思想，不进行阻塞而是继续逐行运行，后续再检验预测是否正确，但需要更多的硬件资源；另一方面也可以在代码层面解决，尽量不让这种情况出现。

九、感想与心得

基于 RISC-V 指令集的 CPU 设计实验，这不是结束，而是开始。

首先，是写代码。一开始我先把老师发的讲义通读了几遍，对让我们完成的任务有了整体的框架认识，明确了需要设计的各个模块的功能和信号接口。对照原理框图，我写了 IF、ID、EX、顶层模块及它们的子模块，同时把相关联的文件放到同一个文件夹里，以便于后续的仿真操作。由于我选择在秋学期考试周期间集中精力完成这项大作业，从开始写代码到完成只花了 2 天的时间。

其次，是进行仿真调试。在写完代码之后，我开始了各个模块的仿真与调试。第一次进行仿真，波形往往都是蓝红线交织的。需要仿真的子模块有 decode、IF、ALU，调试过程比较容易。但是顶层仿真就不是这么回事了，第一次顶层仿真从头到尾全是蓝线。于是我把各个子模块的信号打开一一检查，发现是 ALU_B 的传递有问题，解决掉后终于出现了绿线，但还是有很多红线蓝线。Verilog 的项目我已经做过很多了，从数字系统实验的音乐播放器，再到不久前才完成的 32 位快速进位加法器，养成了一套适合自己的调试习惯：把出现问题的波形和信号记录在 txt 文件上，做好日志记录，在下一次打开时便能顺着上次的进程继续调试。我还会通过画信号流图，分析各个信号在不同层级上的传递。这样子顺藤摸瓜，总能找到问题的源头，只是需要耐心。

每次成功解决掉一个 bug，心里总是感叹：要是当时写代码的时候再细心一点就好了。诚然，有很多 bug 都是由于粗心造成的，比如寄存器位数本应是 32 位结果写成了 1 位。当然也有一些错误是由于我对原理的理解不够扎实，比如流水线清零的信号 Stall 我给所有层级都加上了，实际上只需要阻塞 EX 级。值得高兴的是，我的语法错误几乎没有了，这得益于上学期数字系统设计实验给我打下的坚实基础。总体上讲，我 debug 的过程并不是特别困难，总共也就花了十几个小时。

最后，便是验收了。在冬学期的第一次实验课上，我把所有代码在 Vivado 上综合、实现，然后上传到开发板上，连接好显示器。老师按着按钮，对着屏幕检查我的每一轮输出，验收完毕后为我登记，我才发现自己是班里第一个做完的同学，心里成就感十足。

下课了，在走出实验室之后，一只脚还没踏进楼梯，我回首看了看这个地方。或许如果未来从事 Verilog 相关的数字系统设计工作时，我会常常会想起这一帮助我夯实基础的地方。会回想起给予我无数帮助的屈老师、唐老师和其他同学。

这不是结束，而是开始……