浙江大学

本科实验报告

基于 RV32I 指令集的微处理器设计

课程名称：	计算机组成与设计
姓　　名：	周灿松
学　　院：	信息与电子工程学院
系：	
专　　业：	信息工程
学　　号：	3190105055
指导老师：	屈民军、唐奕

2021 年 12 月 20 日

浙江大学实验报告

专业：信息工程
姓名：周灿松
学号：3190105055
日期：2021 年 12 月 20 日
地点：教 11-400

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 指令集的微处理器设计　实验类型：设计实验　同组学生姓名：

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。

二　实验任务

1. 基本要求

设计一个流水线 RISC-V 微处理器，具体要求如下所述。

1.1　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

1.2 采用 5 级流水线技术，对数据冒险实现转发或阻塞功能。

1.3 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

2.1 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。

2.2 要求设计的 CPU 增加异常（exception）、自陷（trap）、中断（interrupt）等处理方案。

三 实验原理与各模块设计

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，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（D)、执行〈EX)、存储器访问 (MEM) 和寄存器回写 (WB）五级。如图 1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。

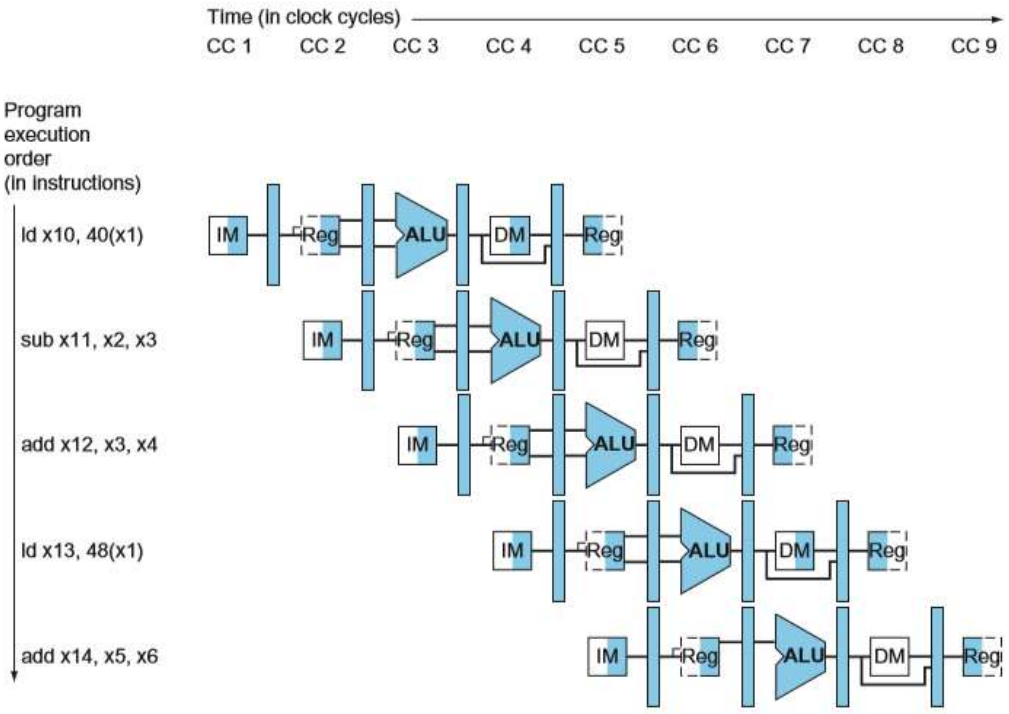


图 1: 流水线流水作业示意图

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。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。

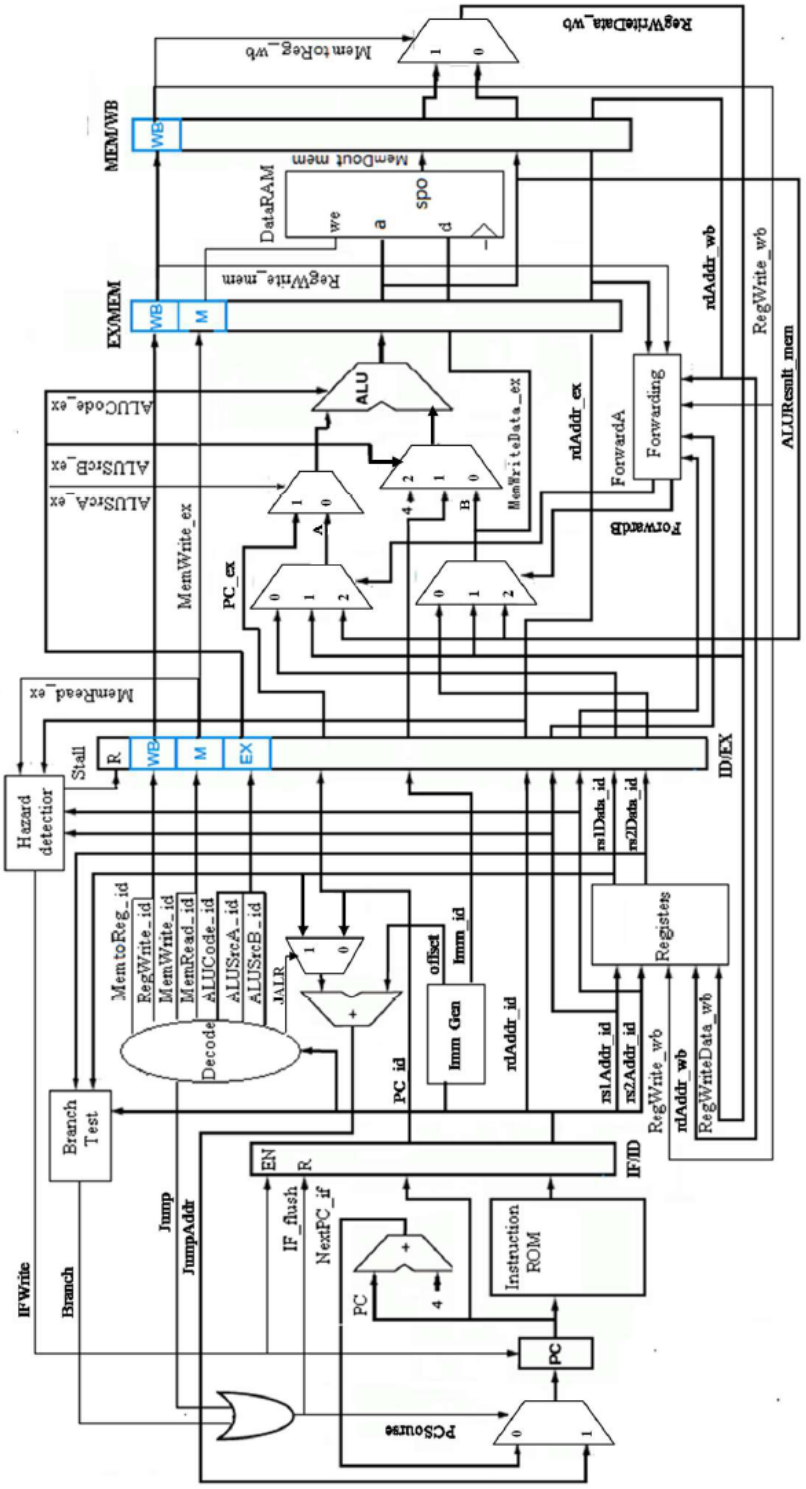


图 2: 流水线原理框图

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。

2. 指令译码 ID 模块的设计

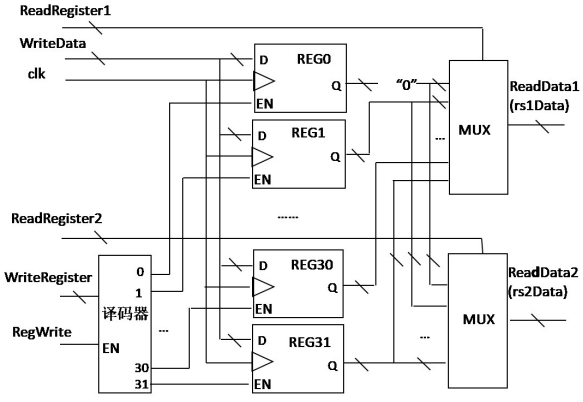
指令译码模块的主要作用是从机器码中解析出指令，并根据解析结果输出各种控制信号。ID 模块主要由指令译码（Decode）、寄存器堆（Registers）、冒险检测、分支检测和加法器等组成。ID 模块的接口信息如图 3所示。

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]		

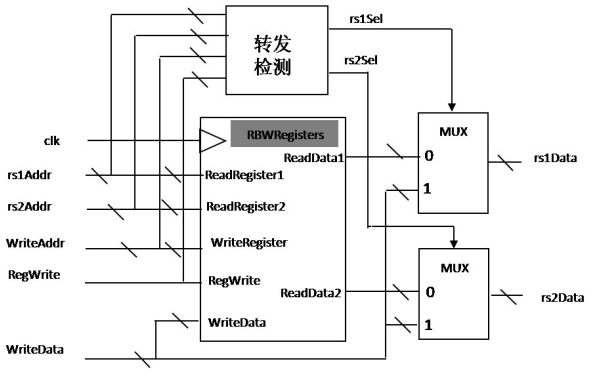
图 3: ID 模块的输入/输出引脚说明

2.1 寄存器堆（Registers）子模块的设计

寄存器堆由 32 个 32 位宽的寄存器组成，并且寄存器 x0 的值始终为 0 值。同时因为需要解决三阶数据相关的数据转发问题，所以需要令其具有 Read After Write 特性。具体的实现电路如图 1所示



寄存器堆的原理框图



具有 Read After Write 特性寄存器堆的原理框图

图 4: 寄存器堆

如图 1书写寄存器堆模块的 Verilog 代码，具体实现请参见 Solution 中 RAWRegisters.v 文件，核心语句如代码 1所示：

Listing 1: 寄存器堆

```
//assign the value of temprary readData
assign ReadData1 = (rs1Addr == 5'b0)?31'b0:regs[rs1Addr];
assign ReadData2 = (rs2Addr == 5'b0)?31'b0:regs[rs2Addr];

//assign the values of select signal
assign rs1Sel = RegWrite & (WriteAddr != 0) & (WriteAddr ...
   == rs1Addr);
assign rs2Sel = RegWrite & (WriteAddr != 0) & (WriteAddr ...
   == rs2Addr);

//write data into regs
always @(posedge clk) begin
    if(RegWrite) regs[WriteAddr] <= WriteData;
end

//if need forward , choose WriteData as output,else ...
   choose the data from regs
assign rs1Data = (rs1Sel == 1)?WriteData:ReadData1;
assign rs2Data = (rs2Sel == 1)?WriteData:ReadData2;
```

2.2 指令译码 (包括立即数产生电路) 子模块的设计

这一模块主要是依据输入的指令产生对应的控制信号，同时产生应有的立即数 Imm 以及偏移量 offset 供流水线的控制使用。这一模块采用组合电路进行设计。

实现思路主要为先将指令按域进行划分，然后依据 op 的值判断出输入指令的类型，随后按照不同指令类型的定义产生应有的控制信号以及立即数（偏移量）。

代码主要见 Solution 中 Decode.v 文件。

2.3 分支检测电路的设计

此模块主要是为了判断出分支条件是否成立，虽然在 Verilog 中可以采用运算符号进行描述，但是需要注意有符号数和无符号数的处理是有着不同的，此次实验中我们采用 32 位快速加法器进行实现。实现步骤如下：

首先利用加法器实现 rs1Data+(rs2Data)+1(即 rs1Data-rs2Data), 设结构为 sum[31:0], 列出真值表我们可以得出代码 2中的表达式, 分支判断模块的其余代码请参见 Solution 中的 testBranch.v 文件。

Listing 2: 分支判断核心语句

```
    isLT = rs1Data[31] && (¬rs2Data[31]) || ...
        (rs1Data[31]¬^rs2Data[31]) && sum[31];//有符号数
2   isLTU = (¬rs1Data[31]) && rs2Data[31] || ...
        (rs1Data[31]¬^rs2Data[31]) && sum[31];//无符号数
```

2.4 冒险检测功能电路的设计

分析可得，冒险成立的条件有以下两条：

(1) 前一条指令为 lw 指令

(2) 两条指令读写同一个寄存器

满足以上两个条件是应当清空 ID/EX 寄存器并且阻塞流水线 ID 级、IF 级流水线，所以会产生如下两个控制信号。

```
    assign Stall = ((rdAddr_ex == rs1Addr_id)||(rdAddr_ex == ...
        rs2Addr_id)) && MemRead_ex;
2   assign IFWrite = ¬Stall;
```

3. 执行模块设计

执行模块主要由 ALU 子模块，数据前推电路以及数据选择器构成，具体数据流向可以参考图 2中的 EX 模块。执行模块的输入输出信息如图 5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 5: EX 模块的输入输出引脚说明

3.1 ALU 子模块的设计

ALU 模块提供 CPU 的基础运算能力，比如加、减、移位、与或、比较等。

此模块接收两个操作数以及一个控制信号 ALUCode，控制信号用于控制 ALU 执行什么操作。输出为一个 32 位的 ALUResult。

为了提高运算速度，一方面首先我们让 ALU 并行执行所有的运算类型，然后根据 ALUCode 的不同选择不同的运算结果。另一方面我们提高 32 位加法器的运算速度，此次实验中我选择了进位选择加法器。

此模块中尤其需要注意的为进行算术右移操作时，Verilog 中的算术右移操作符'»>' 被移位的对象只能够是 reg 类型，所以需要声明一个中间变量对其进行算术右移，代码如下：

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

此子模块的代码详情可见 Solution 中的 ALU.v 文件。

3.2 数据前推电路

此部分电路主要由数据选择器以及产生数据选择器控制地址信号的 ForwardA 与 ForwardB 确定。首先按照实验手册中的要求产生正确的 ForwardA 与 ForwardB 信号，然后按照图 2中 EX 模块中的数据前推部分连接信号。

4. 数据存储模块 (DataRAM) 的设计

此模块采用 Xilinx 的 IP 内核实现，此处我们生成了一个空间为 64×32bit 的单端口 RAM，输出为组合输出。因为容量为 64×32bit，所以存储器地址共 6 位，所以将地址位与 ALUResult_mem[7:2] 连接。

5. 取指令模块 (IF) 的设计

IF 模块主要由指令指针寄存器 (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 模块的接口信息如图 6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 6: IF 模块的输入/输出引脚说明

其中指令指针寄存器用于存储当前 PC 值，由一个 D 触发器构成，以控制信号 IFWrite 作为触发器的使能信号确认是否需要更新 PC 值。

指令存储器模块用于存储代码的所有指令。

指令指针选择器根据 Decode 模块解码出来的控制信号确认选择原 PC 值加 4 或是选择跳转之后的 PC 值，以此方式实现指令的跳转。

32 位加法器用于实现 PC 值的自增。

此模块的具体实现请参加 Solution 中的 IF.v 文件，此处不再赘述。

6. 流水线寄存器的设计

流水线寄存器的作用为将流水线的各个部分却分开来，一共有 IF/ID,ID/EX,EX/MEM,MEM/WB 四组。

其中 IF/ID 这一组寄存器要求带有同步清零功能，并且因为在发生数据冒险时需要阻塞 IF/ID 寄存器，所以其还需要带有保持功能（具有使能 EN 信号输入）

ID/EX 流水线寄存器需要带有同步清理功能，因为在发生数据冒险时需要将其清空。

而 EX/MEM,MEM/WB 两个流水线寄存器则只是普通的 D 型触发器。

因为每一组寄存器都有许多变量，所以如果采用单输入的 D 触发器会使编程工作变得繁琐起来，所以我在书写这一部分时选择了每一组寄存器都实例化一个多输入的 D 触发器。

7. 顶层设计

顶层设计只需要将前面各个模块按照图 2进行连接即可。连接的过程中注意按照信号所属层级进行命名，以此方便区分不同阶段的信号。

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 实验结果

1. Decode 模块

1.1 仿真图

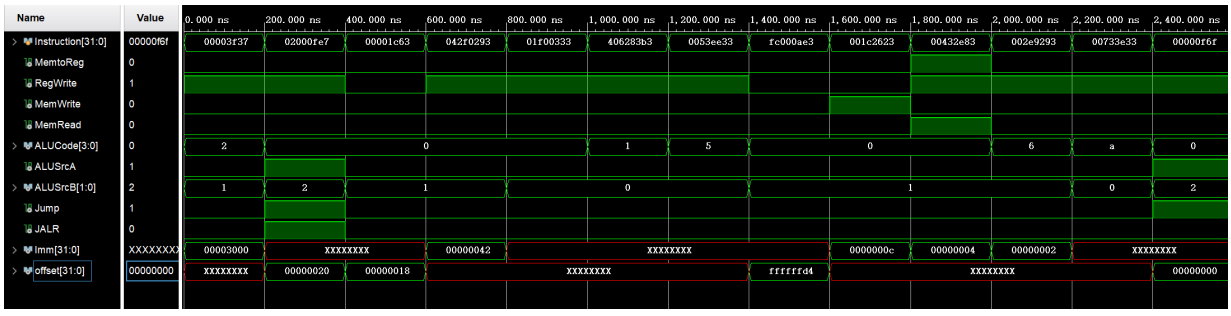


图 7: Decode 模块仿真图

1.2 仿真结果分析

从左到右依次手动译码每条指令得到以下结果:

(1) lui x30, 0x3000, 为 U 型, 输出应为 Imm = 0x0000_3000,offset 无意义, ALUCode=4'd2,ALUSrcA=0, ALUSrcB=2'b01, LUI 指令需要回写,RegWrite=1, 波形显示正确。

(2) ja1r X31,1ater(X0),为 I 型,输出应为 Imm 无意义,offset=0x0000_0020,ALUCode=4'd0,ALUSrcA=1, ALUSrcB=2'b10，JALR=1，Jump=1，JALR 指令需要回写，RegWrite=1，波形显示正确。

(3) bne X0，X0，end , 为 SB 型, 输出应为 Imm 无意义, offset=0x0000_0018 , ALUCode=4'd0, ALUSrcA=0 , ALUSrcB=2'b01 , BNE 指令不需要回写, RegWrite=0 , 波形显示正确。

(4) addi X5,X30,42, 为 I 型, 输出应为 Imm =0x0000_0042,offset 无意义,ALUCode=4'd0,ALUSrcA=0, ALUSrcB=2'b01，ADDI 指令需要回写，RegWrite=1，波形显示正确。

(5) add X6，X0，X31，为 R 型，输出应为 Imm 无意义，offset 无意义，ALUCode=4'd0，ALUSrcA=0, ALUSrcB=2'b00，ADD 指令需要回写，RegWrite=1，波形显示正确。

(6) sub X7，X5，X6，为 R 型，输出应为 Imm 无意义，offset 无意义，ALUCode=4'd1，ALUSrcA=0, ALUSrcB=2'b00，SUB 指令需要回写，RegWrite=1，波形显示正确。

(7) or X28，X7，X5，为 R 型，输出应为 Imm 无意义，offset 无意义，ALUCode=4'd5，ALUSrcA=0, ALUSrcB=2'b00，OR 指令需要回写，RegWrite=1，波形显示正确。

(8) beq X0，X0，ear1ier，为 SB 型，输出应为 Imm 无意义，offset=0xffff_ffd4，ALUCode=4'd0, ALUSrcA=0，ALUSrcB=2'b01，BEQ 指令不需要回写，RegWrite=0，波形显示正确。

(9) sw X28，0C(X0), 为 S 型, 输出应为 Imm=0x0000_000c, offset 无意义, ALUCode=4'd0 , ALUSrcA=0, ALUSrcB=2'b01，MemWrite=1，波形显示正确。

(10) lw X29,04(X6), 为 I 型，输出应为 Imm=0x0000_0004, offset 无意义，ALUCode=4'd0, ALUSrcA=0, ALUSrcB=2'b01, MemRead=1，MemtoReg=1，RegWrite=1，波形显示正确。

(11) sll X5,X29,2，为 I 型，输出应为 Imm=0x0000_0002，offset 无意义, ALUCode=4'd6 , ALUSrcA=0 , ALUSrcB=2'b01 , RegWrite=1，波形显示正确。

(12) sltu X28,X6,X7，为 R 型，输出应为 Imm 无意义，offset 无意义，ALUCode=4'd10，ALUSrcA=0, ALUSrcB=2'b00 , RegWrite=1，波形显示正确。

(13) jal X31,done , 为 UJ 型, 输出应为 Imm 无意义, offset=0x0000_0000 , ALUCode=4'd1 , ALUSrcA=1 ， ALUSrcB=2'b10 ， SUB 指令需要回写，RegWrite=1，波形显示正确。

由上述分析可知，Decode 模块仿真结果正常。

2. ALU 模块

2.1 完整仿真图

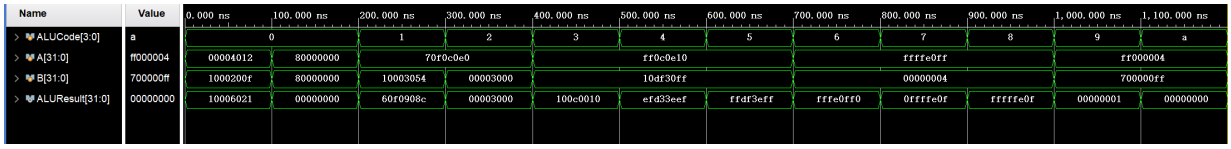


图 8: ALU 完整仿真图

2.2 仿真图分析

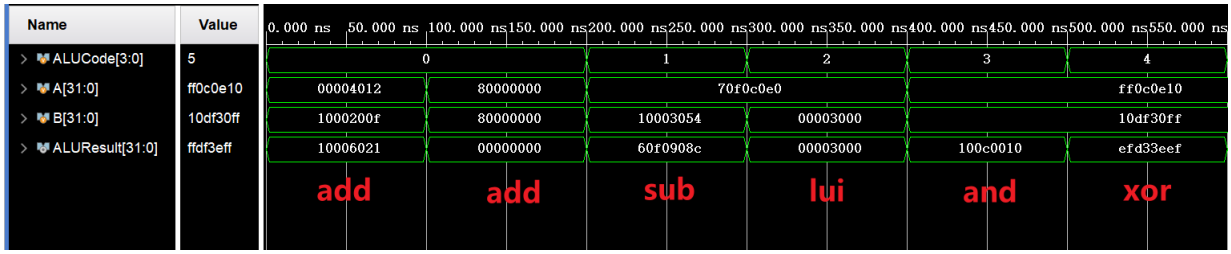


图 9: ALU 前半部分

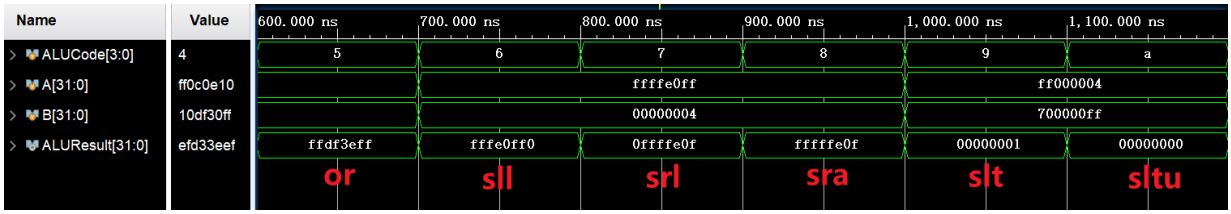


图 10: ALU 后半部分

在仿真图中已经标出了对应时间段的 ALUCode 对应操作，按照操作进行简单的计算即可证明 ALU 部分工作正常。

3. IF 模块

3.1 完整仿真结果

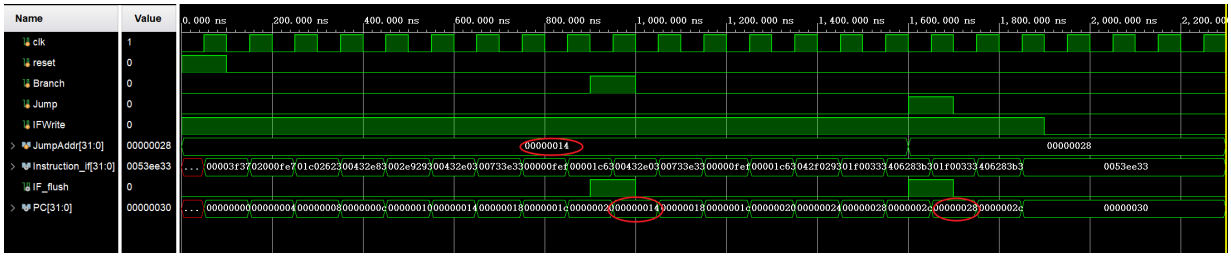
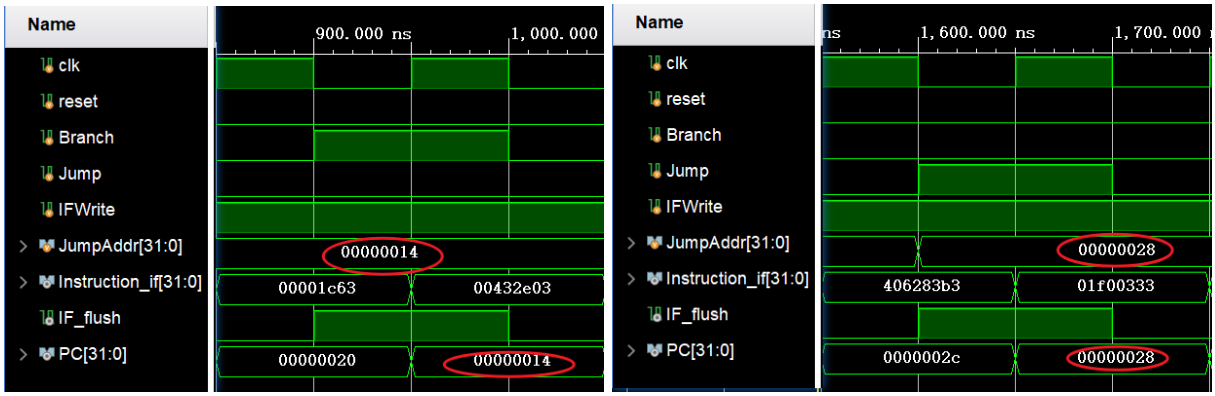


图 11: IF 模块仿真结果

3.2 结果分析



IF 局部图 1 IF 局部图 2

图 12: IF 局部分析

通过分析上述图 12中两个两个局部图可以看出，当控制信号 Branch 或者 Jump 为 1 是，PC 值能够实现正确跳转，而当两个控制信号均为 0 是，PC 值则依次递增，说明 IF 模块运行正常。

4. CPU 顶层

4.1 仿真图像

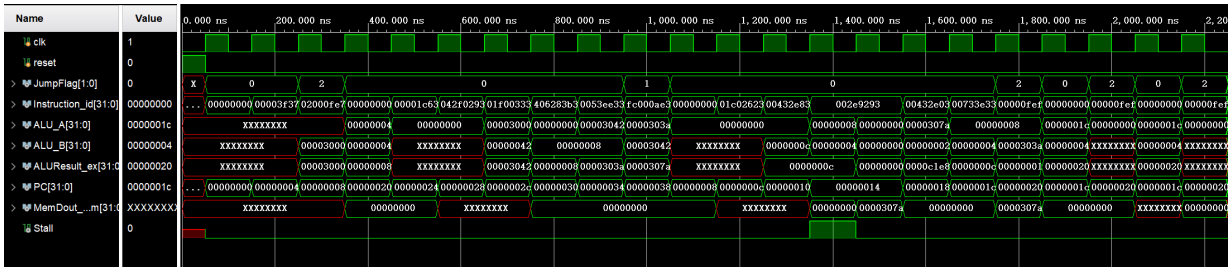


图 13: CPU 顶层仿真结果

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	0000fef (jal)	2	0	8	303a	1	307a
	19	1c	0	0	0	1c	4	20	-
	20	20	0000fef (jal)	2	0	-	-	-	-
	21	1c	0	0	0	1c	4	20	-

图 14: 应有的测试结果

4.2 结果分析

依次对比图 13与图 14, 可以发现仿真结果中每个周期内各个信号的取值与应有结果相同，由此可以说明仿真结果正确。

5. 上板验证

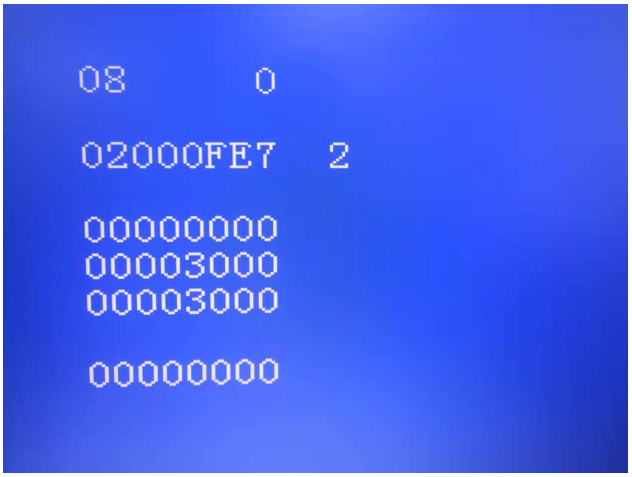


图 15: 上板结果

六 问题记录

(1) 文件没有做好备份，在实验做到一半的时候因为某些原因永久性删除了工程代码，导致最后耗费了比较多的时间重新写大作业。在第二次写的时候我吸取了上一次的教训，每写完一部分就将本地的代码与 Github 云端进行一次备份。

(2) 在设计 ALU 的时候遇到 Binvert 位扩展的问题，百度得到了结果，应该是 32Binvert 形式，最外面一定要有花括号。本次实验用到了很多数据位扩展的知识。

(3) IF/ID 级寄存器的重置清零端传入的信号我最初单纯认为是 reset, 最后通过查看图 2后发现应该传入的值为 IF_flush|reset。只有这样才能够达到在发生数据冒险时清空寄存器的目的。

(4) 寄存器堆的接线错误。这一错误并不会直接在最顶层的仿真结果中反映出来，其表现为 PC 信号更新错误，在一步步观察信号之后才定位到是寄存器堆出了问题。

(5) 很多时候问题的成因并不是那么明显，解决问题最好的方法是对照着电路图一步一步地分析一个信号的错误是有那些信号导致的，最终才能够找到真正出问题的地方。

七 思考题

插入一个气泡，然后将 MEM/WB 寄存器的数据转发到 EX，这样似乎是可以的。在 lw 后面加入 nop 但是负载延迟在硬件上非常不可预知，从 RAM 或缓存 load，可能会因资源竞争而变慢。负载延迟使延迟增加，因此大多数 CPU 架构中不去解决这一问题。后来的 MIPS 增加了死锁来避免填充 nop。

这一问题主要靠汇编器调整指令顺序来解决。