华中科技大学

2024

硬件综合训练　　课程设计报告

题　　目：	5 段流水 CPU 设计
专　　业：	计算机科学与技术
班　　级：	CS2208
学　　号：	U202215628
姓　　名：	方子豪
电　　话：	19546890096
邮　　件：	3299488768@qq.com

目 录

1 课程设计概述

1.1 课设目的

计算机组成原理是计算机专业的核心基础课。该课程力图以“培养学生现代计算机系统设计能力”为目标，贯彻“强调软/硬件关联与协同、以 CPU 设计为核心/层次化系统设计的组织思路，有效地增强对学生的计算机系统设计与实现能力的培养”。课程设计是完成该课程并进行了多个单元实验后，综合利用所学的理论知识，并结合在单元实验中所积累的计算机部件设计和调试方法，设计出一台具有一定规模的指令系统的简单计算机系统。所设计的系统能在 LOGISIM 仿真平台和 FPGA 实验平台上正确运行，通过检查程序结果的正确性来判断所设计计算机系统正确性。

课程设计属于设计型实验，不仅锻炼学生简单计算机系统的设计能力，而且通过进行中央处理器底层电路的实现、故障分析与定位、系统调试等环节的综合锻炼，进一步提高学生分析和解决问题的能力。

1.2 设计任务

本课程设计的总体目标是利用 FPGA 以及相关外围器件，设计五段流水 CPU，要求所设计的流水 CPU 系统能支持自动和单步运行方式，能正确地执行存放在主存中的程序的功能，对主要的数据流和控制流通过 LED、数码管等适时的进行显示，方便监控和调试。尽可能利用 EDA 软件或仿真软件对模型机系统中各部件进行仿真分析和功能验证。在学有余力的前提下，可进一步扩展相关功能。

1.3 设计要求

（1） 根据课程设计指导书的要求，制定出设计方案；

（2） 分析指令系统格式，指令系统功能。

（3） 根据指令系统构建基本功能部件，主要数据通路。

（4） 根据功能部件及数据通路连接，分析所需要的控制信号以及这些控制信号的有效形式；

（5） 设计出实现指令功能的硬布线控制器；

（6） 调试、数据分析、验收检查；

（7） 课程设计报告和总结。

1.4 技术指标

（8） 支持表 1.1 前 27 条基本 32 位 MIPS 指令；

（9） 支持教师指定的 4 条扩展指令；

（10） 支持多级嵌套中断，利用中断触发扩展指令集测试程序；

（11）支持 5 段流水机制，可处理数据冒险，结构冒险，分支冒险；

（12） 能运行由自己所设计的指令系统构成的一段测试程序，测试程序应能涵盖所有指令，程序执行功能正确。

（13） 能运行教师提供的标准测试程序，并自动统计执行周期数

（14） 能自动统计各类分支指令数目，如不同种类指令的条数、冒险冲突次数、插入气泡数目、load-use 冲突次数、动态分支预测流水线能自动统计预测成功与失败次数。

表 1.1 指令集

#	指令助记符	简单功能描述	备注
1	ADD	加法	指令格式参考 MIPS32 指令集，最终功能以 MARS 模拟器为准。
2	ADDI	立即数加	
3	ADDIU	无符号立即数加	
4	ADDU	无符号数加	
5	AND	与	
6	ANDI	立即数与	
7	SLL	逻辑左移	
8	SRA	算数右移	
9	SRL	逻辑右移	
10	SUb	减	
11	OR	或	
12	ORI	立即数或	

#	指令助记符	简单功能描述	备注
13	NOR	或非	
14	LW	加载字	
15	SW	存字	
16	BEQ	相等跳转	
17	BNE	不相等跳转	
18	SLT	小于置数	
19	STI	小于立即数置数	
20	SLTU	小于无符号数置数	
21	J	无条件转移	
22	JAL	转移并链接	
23	JR	转移到指定寄存器	If $v0==10 halt(停机指令)
24	SYSCALL	系统调用	else 数码管显示$a0 值
25	MFC0	访问 CP0	中断相关，可简化，选做
26	MTC0	访问 CP0	中断相关，可简化，选做
27	ERET	中断返回	异常返回，选做
28	AUIPC	PC 加立即数储存到寄存器	
29	LUI	设置寄存器值	
30	SH	半字存储到内存	
31	BLT	有符号比较小于时跳转	

2 总体方案设计

2.1 单周期 CPU 设计

我们采用**硬布线控制器**的方式来设计我们的单周期 CPU。主要有五个重要功能部件组成，分别是程序计数器 PC，指令寄存器 IM，寄存器堆 RF，运算器 ALU 以及数据存储器 MEM。各个部件通过数据通路进行连接最终实现一个单周期 CPU。

总体结构图如图 2.1 所示。

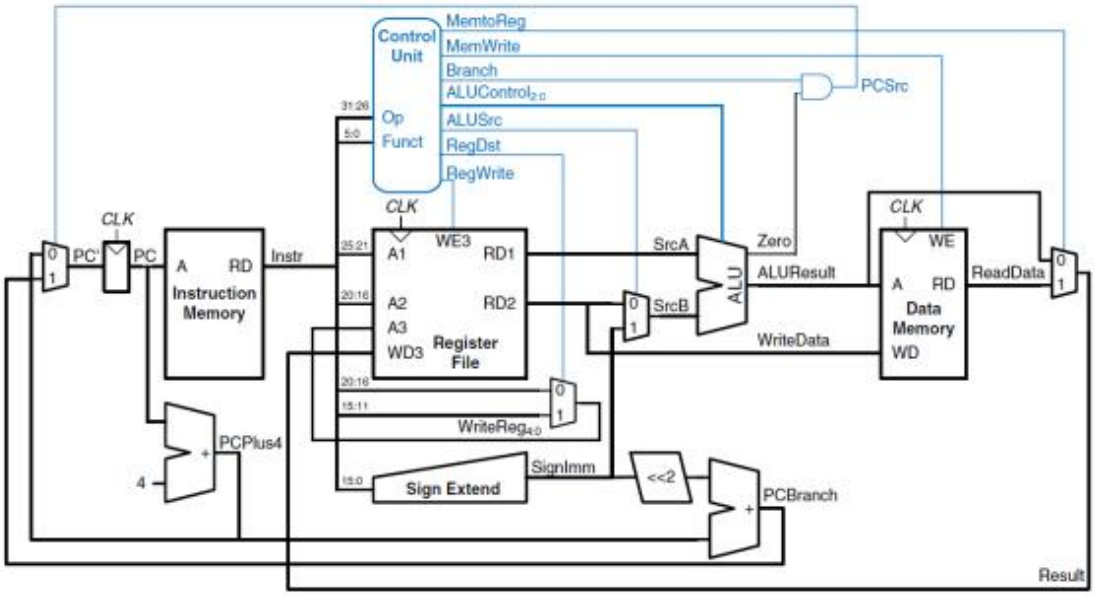

图 2.1 单周期 CPU 总体结构图

2.1.1 主要功能部件

1. 程序计数器 PC

在设计单周期 CPU 时，程序计数器是一个核心组件，其主要作用是指示当前指令的地址，并控制指令的执行顺序。在具体实现中，PC 所存储的是**下一条指令的地址。**其输入为多个可能地址的其中之一，输出为最终根据程序具体运行情况选择的地址。输入输出位宽均为 32 位。

2. 指令存储器 IM

指令存储器用于存储指令内容，由 Logisim 中自带 ROM 部件实现，输入输出格式及引脚见下表 2.1

表 2.1 指令存储单元引脚与功能描述

引脚	输入/输出	位宽	功能描述
A	输入	10	指令地址
sel	输入	1	片选信号
D	输出	32	具体指令内容

3. 运算器

运算器用于执行各种算数和逻辑操作。按照要求进行运算并输出结果至对应端口中。在这里选用已经 Logisim 中已经设计好的运算部件。输入输出格式及引脚见下表 2.2

表 2.2 算术逻辑运算单元引脚与功能描述

引脚	输入/输出	位宽	功能描述
A	输入	32	操作数 A
B	输入	32	操作数 B
S	输入	4	ALUOP，运算操作
=	输出	1	比较 AB 是否相等。是则输出 1 反之为 0.
<	输出	1	（A<B）？1:0
>=	输出	1	（A>=B）？1:0
Result	输出	32	ALU 运算结果
Result2	输出	32	ALU 结果第二部分，多用于乘法指令高位或出发指令的余数位，其他操作为 0。

4. 寄存器堆 RF

寄存器堆用于模拟 RISC-V 中所有寄存器。具体输入输出引脚见表 2.3

表 2.3 寄存器堆单元引脚与功能描述

引脚	输入/输出	位宽	功能描述
R1#	输入	5	R1 编号
R2#	输入	5	R2 编号
W#	输入	5	写入地址
Din	输出	32	写入内容
WE	输出	1	写使能
CLK	输出	1	时钟信号
R1	输出	32	R1 内容
R2	输出	32	R2 内容

5. 数据存储器

用于模仿 RISC-V 中的主存，由 Logisim 中 RAM 实现。具体引脚见表

表 2.4 数据存储器单元引脚与功能描述

引脚	输入/输出	位宽	功能描述
A	输入	32	操作数 A
D	输入	32	操作数 B
S	输入	4	ALUOP，运算操作
=	输出	32	比较 AB 是否相等。是则输出 1 反之为 0.
<	输出	32	（A<B）？1:0
>=	输出	1	（A>=B）？1:0
Result	输出	1	ALU 运算结果
Result2	输出	1	ALU 结果第二部分，多用于乘法指令高位或出发指令的余数位，其他操作为 0。

2.1.2 数据通路的设计

由于整体实验要求设计五段流水线 CPU，因此在单周期 CPU 的数据通路设计时，我们就要考虑到流水线阶段的数据通路设计问题。为此，我们将数据通路分为以下五个模块：取指令阶段（IF），指令解码阶段（ID），执行阶段（EX），访存阶段（MEM），写回阶段（WB）。虽然在单周期 CPU 中这些阶段都在**一个周期内完成**，但是基于此思想设计的数据通路在流水线部分可以发挥作用。同样在数据通路设计中除了 2.1.1 节中的五个主要功能部件外，我们还需要**控制单元、扩展单元、加法器、多路选择器**等额外模块。

（1）取指令阶段：

模块参与：**程序计数器 PC（主要）**，运算器，指令存储器。

数据流：PC 提供当前指令地址，指令存储器根据地址读出指令，加法器计算下一条指令地址 PC+4.

具体流程：每当一个时钟周期到来时，根据该 PC 值做出对应操作。在此之后我们取出 PC 并+4，但这不一定就是我们下一条指令的地址。因为我们还要考虑跳转指令等可能。根据上一条指令的类型，我们判断下一条指令的地址是 PC+4 还是在上一条指令中直接跳转或者是通过 ALU 进行计算之后获得的地址。最后将正确地址输入 PC 为下一个时钟周期做准备。

（2）指令解码阶段：

模块参与：指令存储器输出指令，控制单元生成控制信号，寄存器堆读取寄存器值，扩展单元对立即数进行扩展。

数据流：解码指令获取操作码、寄存器号和立即数，根据控制单元的信号读取寄存器或扩展立即数。

具体流程：当完成取指令之后，我们将指令字 IR 输入已经设计好的硬布线控制器中，得到对应的控制信号。对于不同类型的指令分别进行不同的操作。比如对于 R 型指令，我们从 IR 中提取 RS1，RS2 和 RD。对于 I 型和 S 型指令，我们对应提取出运算的立即数并且扩展成 32 位。这里的具体操作跟指令的设计有关。

（3）执行阶段：

模块参与：运算器、加法器、中断处理

数据流：ALU 执行算数运算、逻辑运算或比较操作。如果是分支指令，计算跳

转目的地址。

具体流程：对于结果的计算，我们首先根据 AluSrcB 控制信号确定 ALU 的 B 引脚是否为立即数，接着根据 AluOp 控制信号确定要进行的计算类型。最后输出 ALU 的计算结果即可。对于系统中断的处理，我们需要设置一个寄存器来保存要显示在 LED 上的数据。我们首先确定 A 引脚输入的值是否是 34。如果是，则将寄存器设置成要显示在 LED 上的数值；否则将 halt 设置为 1，引发时钟暂停。

（4）访存与写回阶段：

模块参与：数据存储器、寄存器堆、多路选择器

具体流程：对 ALU 计算结果进行对应处理（比如选取高位，具体处理与指令相关），执行将数据写入数据存储器中或者将数据存储器中对应地址的值取出。并将 ALU 计算结果或者数据存储器读取的值写入目标寄存器。

2.1.3 控制器的设计

在设计单周期 CPU 时，我们采用硬布线控制器进行总体的调用和控制。对于每一条指令，我们需要确定它所有的控制信号以及其对应的处理的地址或立即数等数据，使得该指令所设计的数据在正确的数据通路上流动。在具体设计中，我们用运算器控制器和控制信号控制生成器两个器件共同组成硬布线控制器。

（1）运算控制器设计

运算控制信号用 AluOp 表示。不同的控制信号会使得运算器完成不同的运算功能。对于每一条指令，我们需要确定它的 AluOp，以保证它可以正确地使用 ALU 完成它的既定功能。由于 ALU 有 13 种不同的运算功能，所以 AluOp 使用 4 个比特即可。对于每一中 ALU 运算分配一个对应 AluOp 即可。

（2）控制信号生成

排除运算控制信号之后剩余的所有信号均可以用一个 bit 为来进行标识，对于每一条指令，我们需要确定它的所有控制信号，使得该指令所设计的数据在正确的数据通路上流动。

（3）硬布线控制器信号说明

综合以上两种控制器的设计，我们设计出了我们的硬布线控制器，其中具体的控制信号以及说明如下表 2.5

表 2.5 硬布线控制器控制信号的作用说明

控制信号	取值	说明
ALU_OP	0-12	ALU 功能选择
MemToReg	0/1	是否从内存写入寄存器
MemWrite	0/1	是否需要写入内存
ALU_SRC	0/1	操作数 B 是否为立即数
RegWrite	0/1	指令是否需要写入寄存器
S_type	0/1	是否是 S 型指令
Beq	0/1	是否是 beq 指令
Bne	0/1	是否是 bne 指令
Jarl	0/1	是否是 jarl 指令
jal	0/1	是否是 jal 指令
half	0/1	是否是半字处理
CSR	0/1	是否是特权指令
auipc	0/1	是否是 auipc 指令
lui	0/1	是否是 lui 指令
sh	0/1	是否是 sh 指令
blt	0/1	是否是 blt 指令
ecall	0/1	是否是 ecall 指令

对照所有控制信号，依次分析各条指令，分析该指令执行过程中需要哪些控制信号，对于与本条指令无关的控制信号，控制信号的取值一律为 0，以简化控制器电路的设计。该控制信号表的框架如表 2.6 所示。

表 2.6 控制器控制信号框架

指令	AluOp	需要的控制信号	指令	AluOp	需要的控制信号
add	5	RegWrite	Lw	5	RegWrite,MemtoReg
sub	6	RegWrite	Sw	5	MemWrite,ALU_Src, S_Type

And	7	RegWrite	Ecall		Ecall
Or	8	RegWrite	Beq		Beq
Slt	11	RegWrite	Bne		Bne
Sltu	12	RegWrite	Jal		RegWrite
Addi	5	RegWrite, ALU_Src	Jalr	5	RegWrite,jal
Andi	7	RegWrite, ALU_Src	Csrrsi	8	lui
Ori	8	RegWrite, ALU_Src	Csrrci	7	lui
Xori	9	RegWrite, ALU_Src	Uret		ecall
Slti	11	RegWrite, ALU_Src	Auipc		RegWrite,auipc
Slli	0	RegWrite, ALU_Src	Lui		RegWrite,lui
Srli	2	RegWrite, ALU_Src	Sh	5	MemWrite,Alu_Src, S_Type,sh
Srai	1	RegWrite, ALU_Src	Blt	11	blt

2.2 中断机制设计

2.2.1 总体设计

中断机制是现代计算机系统中处理异步事件的重要手段，通过允许 CPU 暂停当前的指令执行，转而处理中断请求，来实现对外部设备、内部异常等事件的快速响应。在 CPU 设计中，中断机制的引入对性能、灵活性以及复杂性都有重要影响。中断可分为单级中断和多级中断。对于单级中断，CPU 在执行中断服务程序时，不允许任何中断请求打断当前的中断服务程序；对于多级中断，高优先级的中断可以打断正在执行的低优先级中断的中断服务程序。

在本实验中我们使用**EPC 硬件堆栈保护**方式实现了单**周期的多级中断**和**流水线的单级中断**，对于 EPC 硬件堆栈保护的中断，CPU 会将执行中断前的 PC 值放入 EPC 寄存器中进行保存。而我们选择使用 Logisim 中自带的寄存器来模拟这种情况。同时设置多个中断硬件位来判断当前中断状态。

2.2.2 硬件设计

（1）CPU 单级中断硬件设计

- INTR：中断响应标识，中断号，（包含 INRT1，INTR2，INTR3，INTR）
- IE：中断使能位，用于开关中断
- IR：中断请求寄存器（IRA，IRB，IRC）
- IR.PC：中断处理程序地址
- MEPC：保存中断断点地址

（2）CPU 多级中断硬件设计

多级中断在单级中断的基础上，支持了中断嵌套，也就是更高级别的中断能够中断比较低的级别的中断。为此我在中断号和断点地址保存处进行了修改。使用了三个串联的寄存器，分别标识三种中断的处理状态。

（3）CPU 流水线单级中断硬件设计

基于单周期 CPU 单级中断设计即可。

2.3 流水 CPU 设计

2.3.1 总体设计

实验目的是设计一个**五段流水 CPU**，也就是将单周期 CPU 数据通路分为五个阶段，每个阶段有对应的功能部件完成。在 2.1.2 中我们设计单周期 CPU 的数据通路时已经对这五个部件进行了讨论，**在这里我们按照上述 2.1.2 章节中的分段继续进行设计即可。**也就是取指令、指令解码、指令执行、内存访问和写回五个阶段。总体的逻辑框架如下图 2.2：

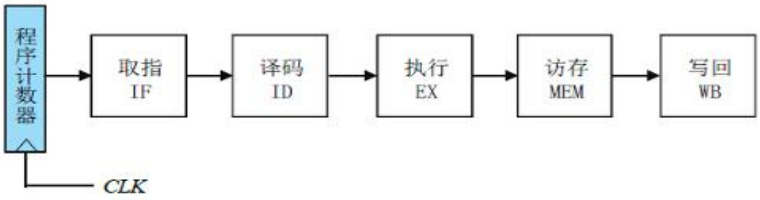

图 2.2 流水线 CPU 逻辑框架

2.3.2 流水接口部件设计

五段流水线每个阶段之间的传输需要间隔一个周期。所以需要利用寄存器或者存储的方式将上一个阶段的数据或者结果传输到下一个阶段。在本实验中，我选择添加了四个流水寄存器作为流水接口部件来完成数据的传输功能。逻辑框架如下：

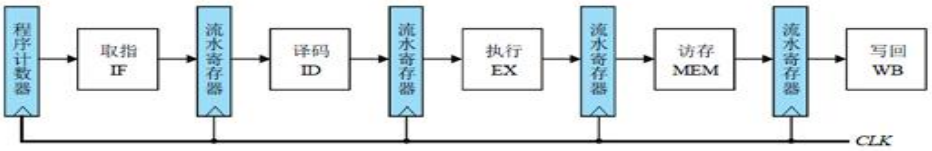

图 2.3 指令流水线逻辑框架

四个流水接口部件的设计有许多相似之处，具体的输入输出引脚如下：

表 2.7 流水接口及其引脚含义

流水接口部件	引脚	引脚含义	流水接口部件	引脚	引脚含义
公共部件	Clk	时钟信号	ID/EX	Sh	Sh 指令
	Rst	复位信号		Blt	Blt 指令
	Flush	刷新信号		Auipc	Auipc 指令
	En	使能信号		Lui	Lui 指令
	IR	当前指令内容		Beq	Beq 指令
	IMM	PC+4		Bne	Bne 指令
	PC	PC		MemToReg	内存写入寄存器
	Writing（IF/ID 无）	写标志		MemWrite	写内存运算编号
	Jal/Jalr（J）	跳转标志		AluOP	是否使用立即数
	MemWrite（MEM/WB 无）	写内存标志		AluScrB	写寄存器
	REGWRITE	写寄存器标志		RegWrite	中断指令
	MemToReg（IF/ID 无）	写内存入寄存器		ecall	
IF/ID	无多余引脚	无	MEM/WB	Half	取半字节指令
EX/MEM	HALT	中断指令		HALT	中断相关
	Half	取半字节指令		high	取高 bit 指令
	Sh	Sh 指令		ALURESULT	ALU 运算结果
	auipc/lui	U 型指令		auipc/lui	U 型指令

2.3.3 理想流水线设计

理想流水线就是将我们设计的单周期CPU拆分成五个流水阶段并用我们设计的流水接口部件**连接**起来即可。但是这仅仅是**理想情况**。由于分支指令需要多个周期才能经过ALU回到PC，所以当我们涉及到分支指令时，如果我们需要进行跳转，那么此时我们的跳转指令已经在ALU之后了。这时候我们PC值变更为跳转地址，但是ID段和EX段还保留着跳转指令之后的指令的两个时钟周期，而这部分就是误取指令。为了解决这个数据冲突问题，提升效率，我们进而实现气泡式流水线等流水线的设计。

2.4 气泡式流水线设计

为了解决分支指令引起的数据冲突问题，气泡式流水线是一个很优秀的方式。解决数据冲突的问题需要将ALU之前的IF/ID，ID/EX段清空。而气泡式流水线就通过在理想流水线的基础上增加**气泡冲突处理**模块来解决这个问题。根据当前CPU状态来确定是否发生冲突。当发生冲突时，插入气泡（也就是给IF/ID和ID/EX流水寄存器一个同步清空信号Flush，将其清空）即可解决问题。而对应的控制模块可以封装成一个对应的气泡冲突处理模块，输入输出引脚如下表 2.8：

表 2.8 气泡处理模块引脚信息

引脚	输入/输出	位宽	引脚含义
ID.IR	输入	32	解码阶段指令
EX.WRITEREG	输入	5	执行阶段寄存器写的编号
EX.REGWRITE	输入	1	执行阶段寄存器写信号
MEM.WRITETEG	输入	5	存储阶段寄存器写的编号
MEM.REGWRITE	输入	1	存储阶段寄存器写信号
DATAZZARD	输出	1	气泡添加逻辑

2.5 重定向流水线设计

虽然气泡流水线能够处理数据冲突的问题并且能够正确执行我们的指令。但是由于插入气泡会浪费过多的周期导致流水线CPU的效率降低。为了提高流水线CPU

的效率，我们修改部分气泡流水线的逻辑，将数据相关的两条指令进行重定向来减少气泡的插入从而提高流水线 CPU 的效率。注意，当两条指令数据相关且上一条指令时访存指令时，我们**依然需要插入气泡来避免冲突。**对应的重定向处理模块的输入输出引脚信息如下:

表 2.9 重定向处理模块引脚信息

引脚	输入/输出	位宽	引脚含义
ID.IR	输入	32	解码阶段指令
EX.MEMTOREG	输入	1	执行阶段内存写入寄存器信号
EX.REGWRITE	输入	1	执行阶段寄存器写信号
MEM.WRITETEG	输入	5	存储阶段寄存器写的编号
MEM.REGWRITE	输入	1	存储阶段寄存器写信号
R1FOWARD	输出	2	重定向控制信号 1
LOADUSE	输出	1	发生访存，依然需要插入气泡
R2FOWARD	输出	2	重定向控制信号 2

具体的重定向定义则是，在取操作数时，先不考虑 ID 段所取的寄存器操作数是否正确，而是等到指令实际使用这些寄存器操作数时再考虑正确性问题。如存在数据相关，EX 段的寄存器操 作数 RS、RT 就是错误数据，可以直接将 EX/MEM 流水寄存器中的 AluResult 或 WB 段的 WriteBackData 直接送到 EX 段的 RS 处，作为 SrcA 送 ALU 参与运算，所以接口部件 ID/EX 需要增加两个流水接口。详细通路如图 2.4 所示：

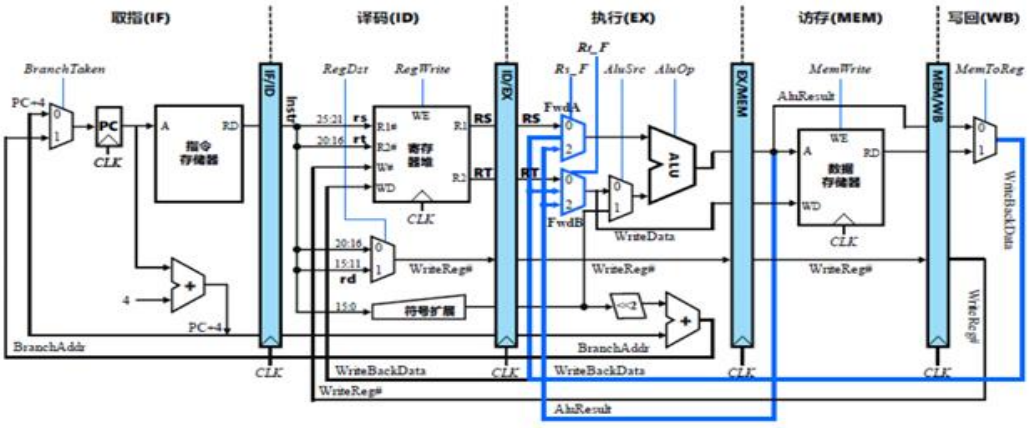

图 2.4 数据重定向数据通路改造

2.6 动态分支预测机制

通过观察理想流水线和气泡流水线、重定向流水线之间的关联，我们发现影响一个流水线运行效率的重要原因就是其**气泡插入数量**。为了进一步提高流水线 CPU 效率，我们引入**动态分支预测机制。**因为程序中的分支指令（如条件跳转和循环）会导致指令执行路径的**不确定性**，分支预测机制的主要目标是尽可能准确地预测程序的执行路径，从而减少流水线暂停和性能损失。我们使用一个分支历史表来记录曾经出现的指令对应的跳转地址。并且用 LUR 算法更新分支历史表。在取指阶段时我们便可以查询 BTB 表判断是否曾经处理过该指令。如果命中则直接预测将要跳转到该地址，这样我们就可以在取指阶段率先预测分支的结果，大大提高了流水线 CPU 的效率。具体的分支历史表设计如下：

表 2.10 分支历史表的输入输出引脚

引脚	输入/输出	位宽	引脚含义
CLK	输入	1	时钟信号
IF.PC	输入	32	取指令阶段 PC 值
BRANCHTAKEN	输入	1	当前是否是分支指令
EX.BRANCH	输入	1	执行阶段是否是分支指令（用于更新）
EX.PC	输入	32	执行阶段的 PC 值
EX.BRANCHADDR	输入	32	执行阶段分支指令跳转的地址（用于更新）
PRIDICTJUMP	输出	1	分支预测地址是否使用
PRIDICTADDR	输出	32	分支预测地址

动态分支预测的预测过程应放在重定向流水线 CPU 的 IF 阶段，更新过程放在重定向流水线 CPU 的 EX 阶段。若执行阶段发现预测正确，则说明此时取指阶段的预取指令是正确的，无需再插入气泡；否则说明预取指令错误，需要插入气泡并重新预取指令。注意到如果预测错误我们所承担的风险也不过是重新插入气泡。但是如果预测正确可以大大减少插入气泡的数量。所以对于分支预测表的设计优劣很大情况下可以决定流水线 CPU 最后的运行效率。

3 详细设计与实现

3.1 单周期 CPU 实现

3.1.1 主要功能部件实现

1) 程序计数器（PC）

使用一个 **32 位寄存器**实现程序计数器 PC，触发方式为下降沿触发，输入为下一条将要执行的指令的地址，输出为当前执行指令的地址。Halt 为停机信号，将此控制信号通过非门取反之后和时钟相与，当需要进行停机时，Halt 控制信号为 1，经过非门之后为 0，与时钟信号相与，屏蔽时钟信号，使整个电路停机。如图 3.1 所示。

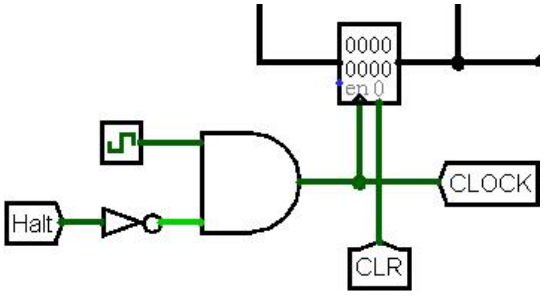

图 3.1 程序计数器（PC）

2) 指令存储器（IM）

使用一个**只读存储器 ROM** 实现指令存储器（IM）。设置该只读存储器的地址位宽为 10 位，数据位宽为 32 位。因为 PC 中存储的指令地址有 32 位，而 ROM 地址线宽度有限，仅为 10 位，故将 32 位指令地址高位部分和字节偏移部分直接屏蔽，使用分线器只取 32 位指令地址的 2-11 位作为指令存储器的输入地址。如图 3.2 所示。

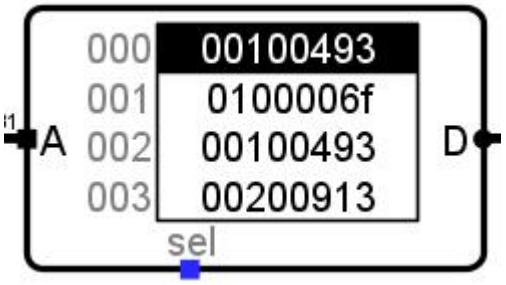

图 3.2 指令存储器（IM）

3) 寄存器堆

将三十二个三十二位寄存器封装成一个寄存器堆，输入位当前想要进行的操作以及操作的寄存器编号，分别用不同的引脚指代，由于只有三十二个寄存器，所以输入编号位宽为五位。输出引脚有两个，分别输出对应的寄存器的值，其位宽与寄存器位宽为 32 位。

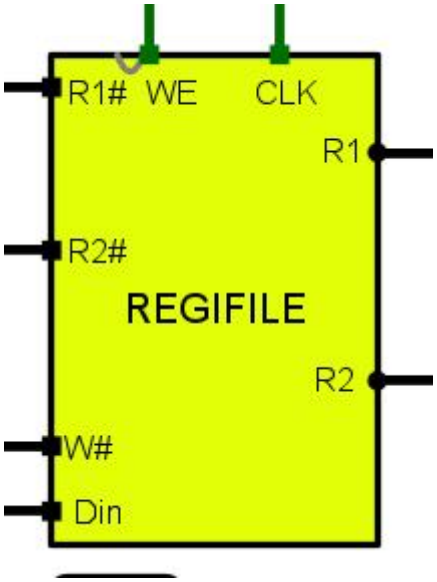

图 3.3 寄存器堆（RF）

4) 运算器（ALU）

运算器用来进行计算，使用 Logisim 中已经封装好的运算器即可。输入为两个想要进行运算的数据和一个 ALU_OP 运算方式。输出为运算结果（加减乘除）以及比较结果（大于小于等于）。操作数和结果均为 32 位。

具体实现如下：

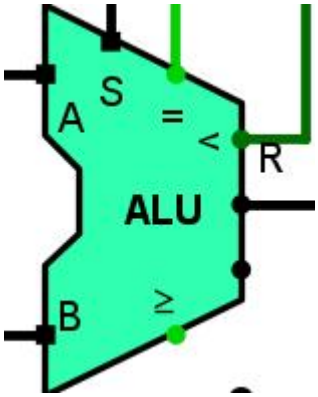

图 3.4 运算器（ALU）

5) 数据存储器

数据存储器用来模拟主存存储数据。输入为 32 位地址及若干控制信号，输出为 32 位地址中储存的内容，其位宽也为 32 位。我们使用 Logisim 中自带的存储器来实现即可。具体如下：

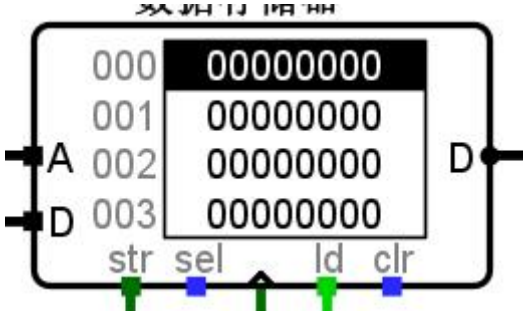

图 3.5 数据存储器

6) 硬布线控制器

硬布线控制器用来总体调控所有器件的具体实施以及数据的流向问题。使得指令在正确的道路上流通，我们将单周期硬布线控制器用一个组合逻辑电路实现，并将其封装为下所示：

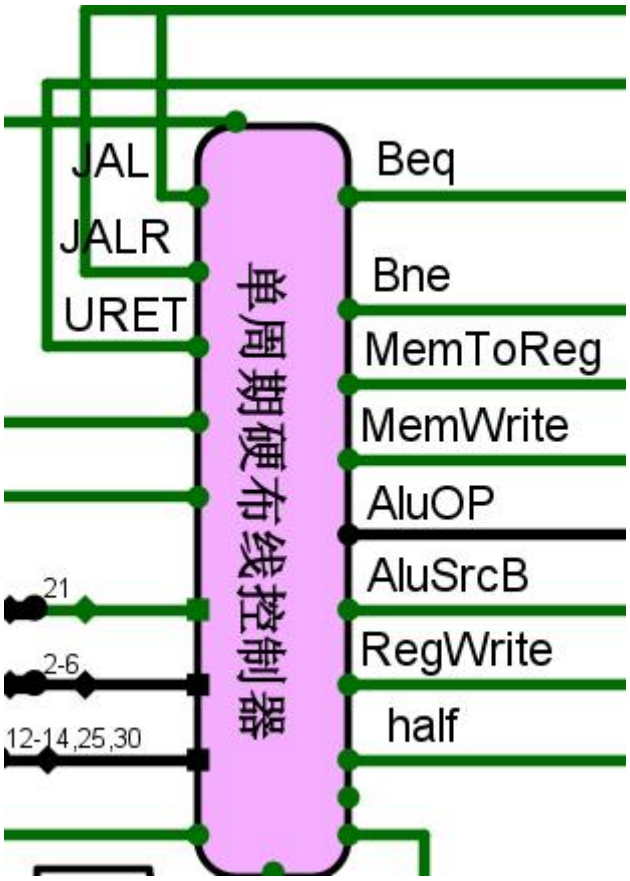

图 3.6 硬布线控制器结构图

3.1.2 数据通路的实现

具体的数据通路的设计已经在 2.1.2 节总体数据通路方案设计中已经提及了，现在基于该章节中的思路对数据通路进行实现。单周期 CPU 的数据通路仿照流水线 CPU，也分为成取指、译码、执行、存储和写回五个 阶段。我们通过选择合适的逻辑门、多路选择器、优先编码器等元件，实现了单 周期 CPU 的数据通路，如所示

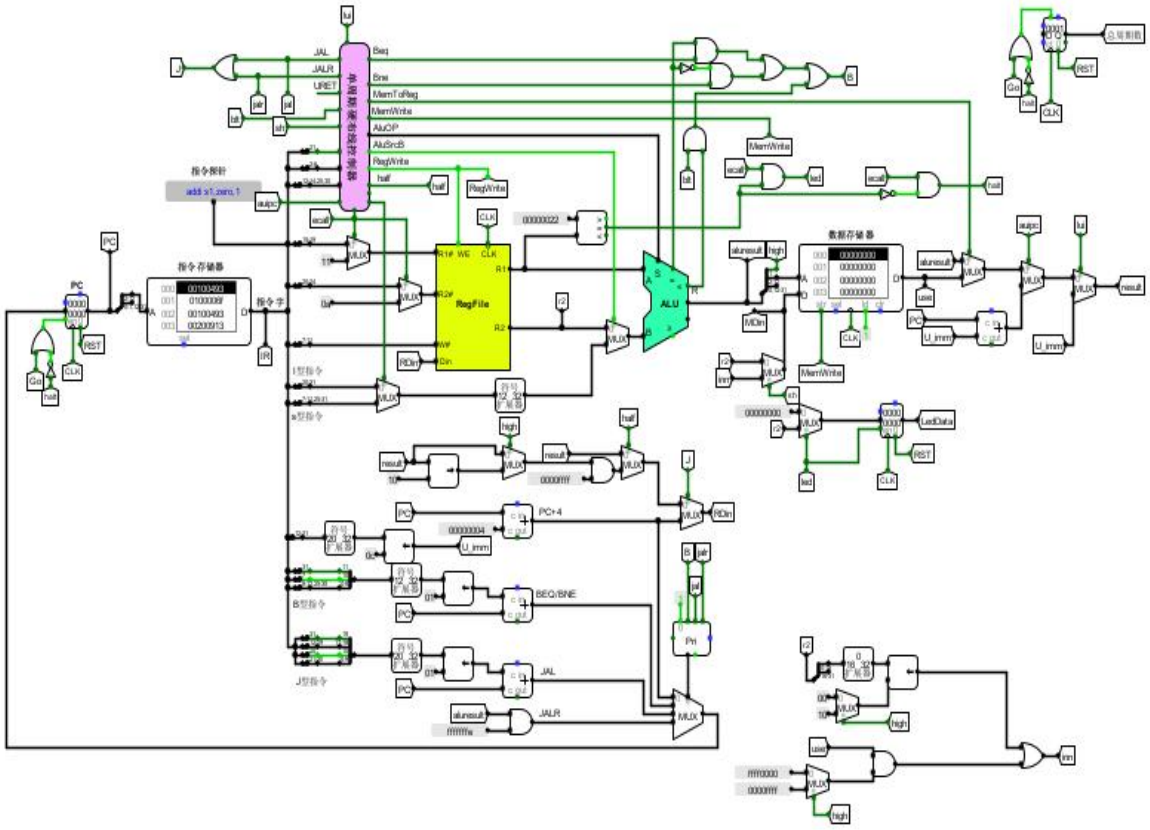

图 3.7 单周期 CPU 数据通路

3.1.3 控制器的实现

根据 2.1.3 中控制器的总体设计方案，我们在 Logisim 上对硬布线控制器进行了实现。首先我们将硬布线控制器分为了控制信号部分和运算控制部分。前者控制运算器 ALU 执行的操作，后者控制 ALU 之外得到其他所有部件的接口的开关、传输和数据流通方向。但是二者是一体的，ALU 的运算结果流向哪个部件也与控制信号部分息息相关。因此，我们第一步首先是通过填写 Excel 表来获取对应指令与输出控制信号之间的逻辑关系，具体填写情况如下图 3.8 所示：

#	指令	Funct7（十进制）	Funct3（十进制）	OpCode（十六进制）	ALU_OP	MemtoReg	MemWrite	ALU_Src	RegWrite	ecall	S_Type	BEQ	BNE	Jal	jalr	auipc	LUI	SH	BLT
1	add	0	0	c	5				1										
2	sub	32	0	c	6				1										
3	and	0	7	c	7				1										
4	or	0	6	c	8				1										
5	slt	0	2	c	11				1										
6	sltu	0	3	c	12				1										
7	addi		0	4	5			1	1										
8	andi		7	4	7			1	1										
9	ori		6	4	8			1	1										
10	xori		4	4	9			1	1										
11	slti		2	4	11			1	1										
12	slli	0	1	4	0			1	1										
13	srli	0	5	4	2			1	1										
14	srai	32	5	4	1			1	1										
15	lw		2	0	5	1		1	1										
16	sw		2	8	5		1	1			1								
17	ecall	0	0	1c						1									
18	beq		0	18								1							
19	bne		1	18									1						
20	jal			1b					1					1					
21	jalr		0	19	5			1	1						1				
22	CSRRSI		6	1c	8			1									1		
23	CSRRCI		7	1c	7			1									1		
24	URET		0	1c						1									
25	AUIPC			5					1							1			
26	LUI			0d					1								1		
27	SH		1	8	5		1	1			1							1	
28	BLT		4	18	11														1

图 3.8 硬布线控制器逻辑表达式

当获取了对应的指令与控制信号之间的逻辑关系之后，在 Logisim 中可以使用表达式生成电路，从而得到我们的控制信号生成电路。同理我们运算控制器也可以用这个方法生成。最终将这两个电路合并到一起成为我们的硬布线控制器。由于具体的两个电路电路图非常复杂，所以仅展示最终的硬布线控制器电路，如图 3.9 所示：

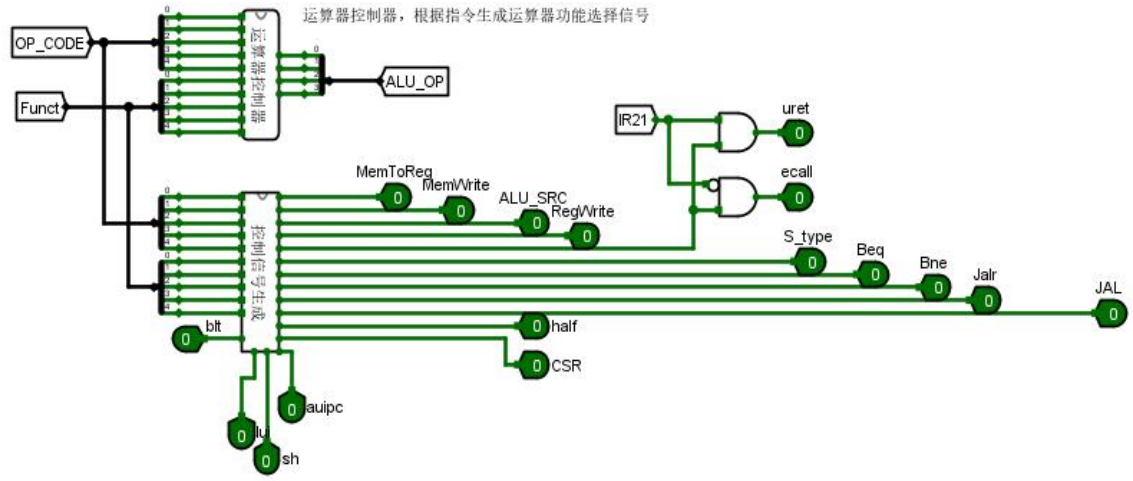

图 3.9 硬布线控制器电路

当完成了上述章节的具体设计之后，我们已经基本上实现了一个简单的单周期 CPU。接下来将对其进行功能扩展。

3.2 中断机制实现

3.2.1 单周期 CPU 单级中断实现

在设计单级中断机制时，我们需要构建几个关键电路来处理中断请求、响应、使能状态、断点保存以及返回操作。以下是每一步的具体实现思路：

（1）中断信号的检测

在我们设计的单级中断 CPU 中，能够引发中断的类别都是外部中断，在实际设计时我们设计了三个中断源 IRA、IRB、IRC，中断级别依次递增（不过单周期终端**不考虑中断级别的问题**，因为一个中断不能中断其他中断）。当发生中断时通过中断信号检测逻辑得出对应的中断号并且进入中断处理程序。具体的电路设计如下：

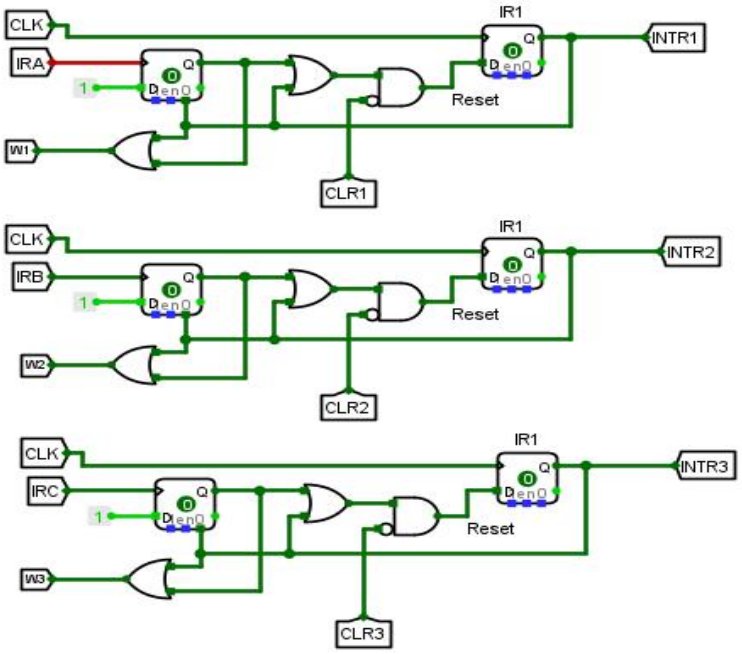

图 3.10 中断识别电路

（2）中断响应实现

在获取到对应的中断号之后，我们需要实现中断响应逻辑，也就是能够进入中断处理程序，在此步骤中需要将中断标志位置 1 通知系统进入中断状态。具体的电路设计如下：

图 3.11 中断响应电路

（3）ISR 地址的确定

不同编号的中断对应着不同的中断处理程序，也对应着不同的入口地址。在具体设计时可以使用一个多路选择器来完成 ISR 的地址确定工作：

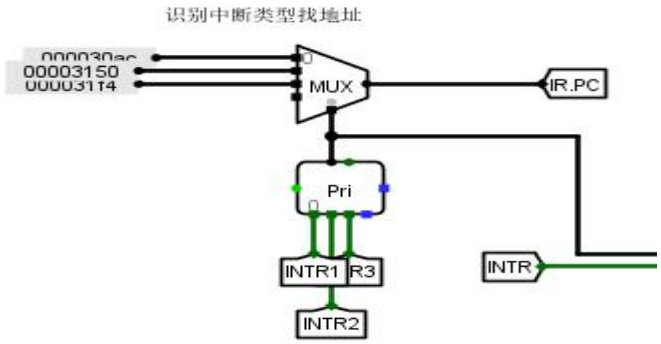

图 3.12 中断地址确定

（4）中断使能与状态保存

单周期 CPU 这一部分的实现比较简单，中断使能寄存器 IE、异常程序计数器 EPC 都使用寄存器来实现即可。

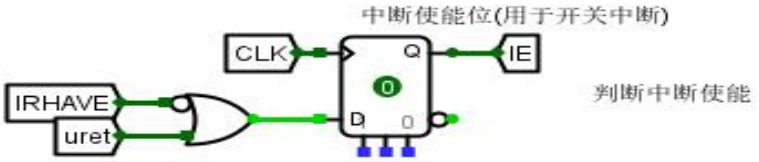

图 3.13 中断使能实现

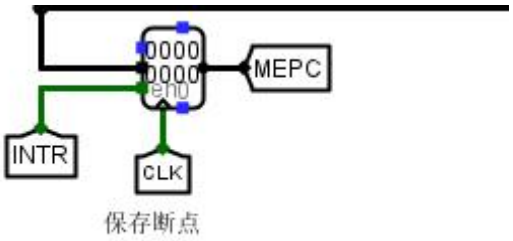

图 3.14 断点保存实现

（5）中断返回

由于中断返回也是修改 PC 值，所以在 PC 的取值之前加一个多路选择器即可：

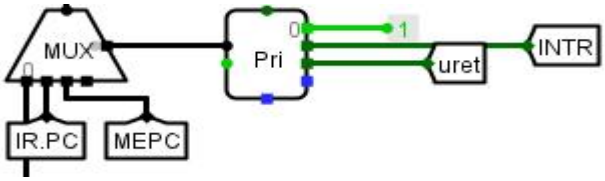

图 3.15 中断返回地址选择

3.2.2 单周期 CPU 多级中断实现

在单周期 CPU 中实现多级中断机制，需要在单级中断机制的基础上扩展，以支持**中断优先级**和**中断嵌套**的功能。首先要明确多级中断机制的基本概念。多级中断机制允许多个中断源按照优先级处理，并能在处理中断时响应更高优先级的中断。这种机制适用于需要实时响应多个事件的系统。而实现多级中断的关键特性在于：

- 中断优先级：每个中断源都有对应的优先级，CPU 在处理中断时，如果有更高优先级的中断发生，会抢占当前中断。
- 嵌套中断：允许中断嵌套，但需确保嵌套深度有限，以免栈溢出。在本课程设计中只有三层中断，所以该问题无须考虑。

为了实现上述额外功能，需要对 3.1.1 中设计的电路进行修改。其中，中断信号检测逻辑，ISR 地址确定逻辑，中断响应逻辑，中断返回逻辑无须修改。重点修改中断使能逻辑和中断地址保护逻辑。

（1）中断使能管理

在多级中断中，硬件电路负责在中断响应时自动关闭中断（关中断），并在中断返回时重新开启中断（开中断）。这是通过直接控制中断使能寄存器（IE）来实现的。除了硬件控制外，支持通过软件指令动态调整中断使能状态，CSRRSI 指令用于打开中断，CSRRCI 指令用于关闭中断。为了确保高优先级中断能够优先处理，引入了屏蔽字来过滤低优先级的中断信号。当有新的高优先级中断请求时，屏蔽字会更新，允许更高优先级的中断通过。

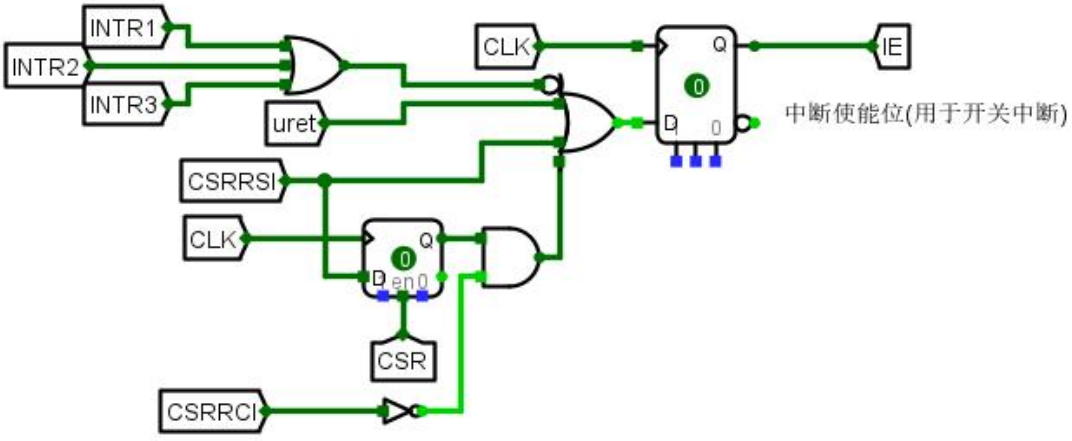

图 3.16 中断使能管理逻辑更新

（2）中断地址保护逻辑

由于中断嵌套的存在，我们也不能仅仅只简单用一个寄存器就保存现场地址了。由于较低级的中断能够被更高等级的中断所中断，我们假设**最坏情况**，也就是三个中断依次发生，那么我们需要依次入栈三个入口地址。所以我们需要设计**中断号保存逻辑**（这一点在单级中断中不需要考虑，因为仅考虑眼前的中断，所以可以很简单的得到当前中断号）和**断点地址保存逻辑**，具体实现如下：

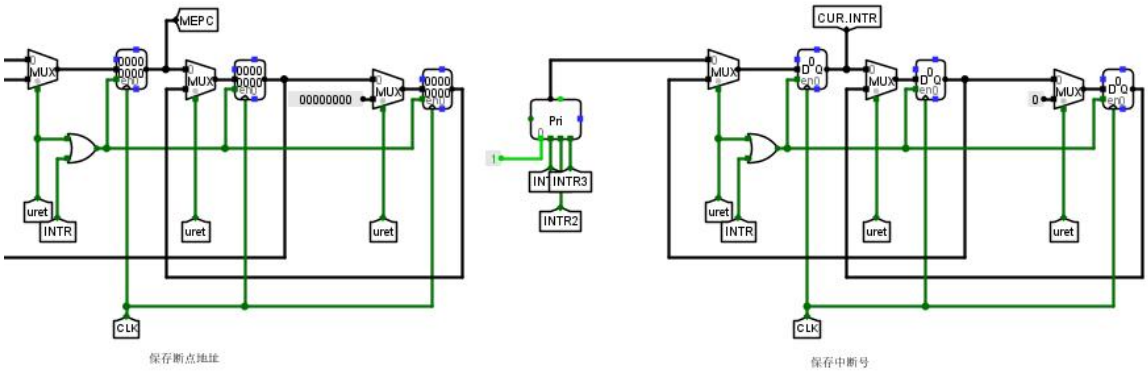

图 3.17 中断保存逻辑更新

3.2.3 流水线 CPU 单级中断实现

流水线 CPU 单级中断的设计与单周期 CPU 单级中断基本一致。注意一点就是保存的断点可能是 ID 段的 PC 值，IF 段的 PC 值（URET 置 1）或者是 EX 段要跳转的地址值，将断点地址保存逻辑进行修改，其他部分按照单周期 CPU 中断设计即可。

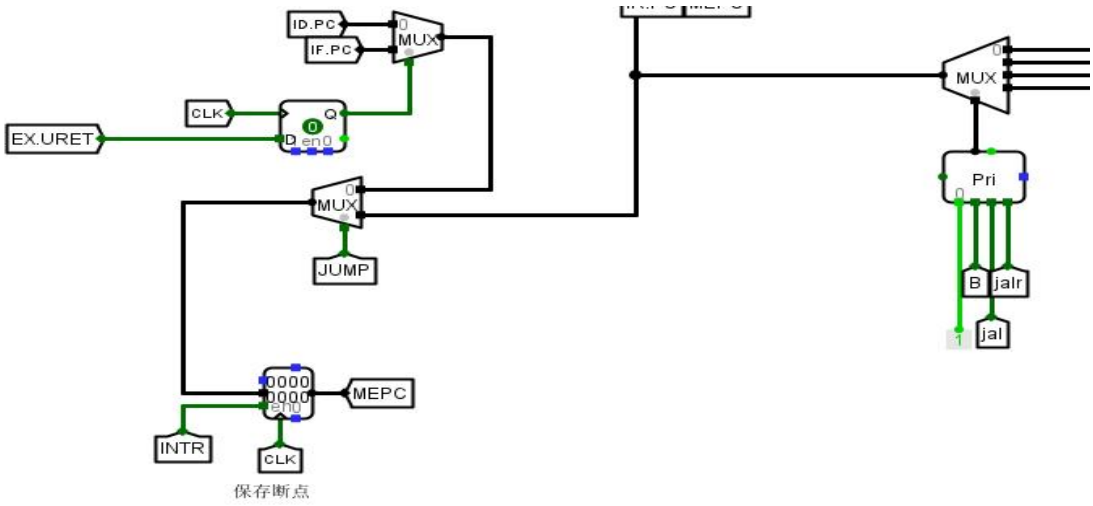

图 3.18 断点保存逻辑修改

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

流水接口部件本质是通过多个并联寄存器和同一的时间调度，实现数据流水传输。我们在 Logisim 中一共实现了四段流水部件 IF/ID，ID/EX，EX/MEM，MEM/WB，而每段部件的结构和遵循的逻辑基本相同，所以仅用其中一段流水接口部件 IF/I 的实现举例。对于每一个输入，使用一个与其位宽一致的寄存器来储存其值。输出均为下一阶段的对应输出。为了支持同步逻辑和支持清空，所有寄存器均使用同一的时钟信号 CLK，和同一清空逻辑 RST。**不过为了体现个人 CCAB 的差异化**，我在设计流水接口部件时将个人的四个 CCAB 指令的流水接口部件**单独提取出来**与原本的 24 条指令的流水接口部件区分，本质上二者都属于同一流水部件。

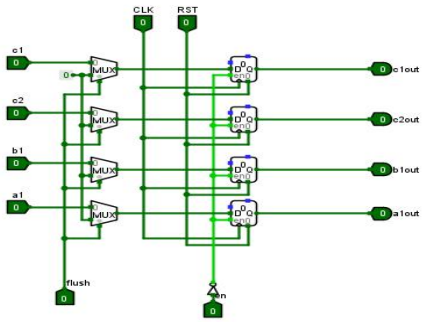

图 3.19 CCAB 逻辑单独实现

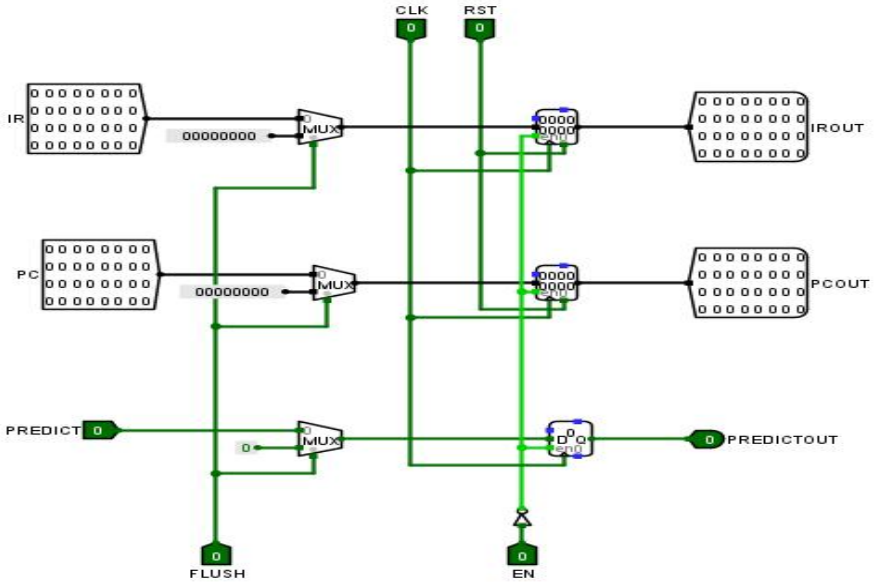

图 3.20 IF/ID 段流水线部件

3.3.2 理想流水线实现

按照 2.3.3 中的设计思路，因为我们是按照取指、译码、执行、存储和写回这五个阶段设计并实现单周期 CPU 的，所以设计理想流水线时仅需将这五个阶段划分开，并且用上文设计的四个流水接口部件相连即可。注意我的流水部件**将 CCAB 部分单独提取出来**体现区分度，不过本质上他们还是一体的，具体的 Logisim 电路如下图 3.21：

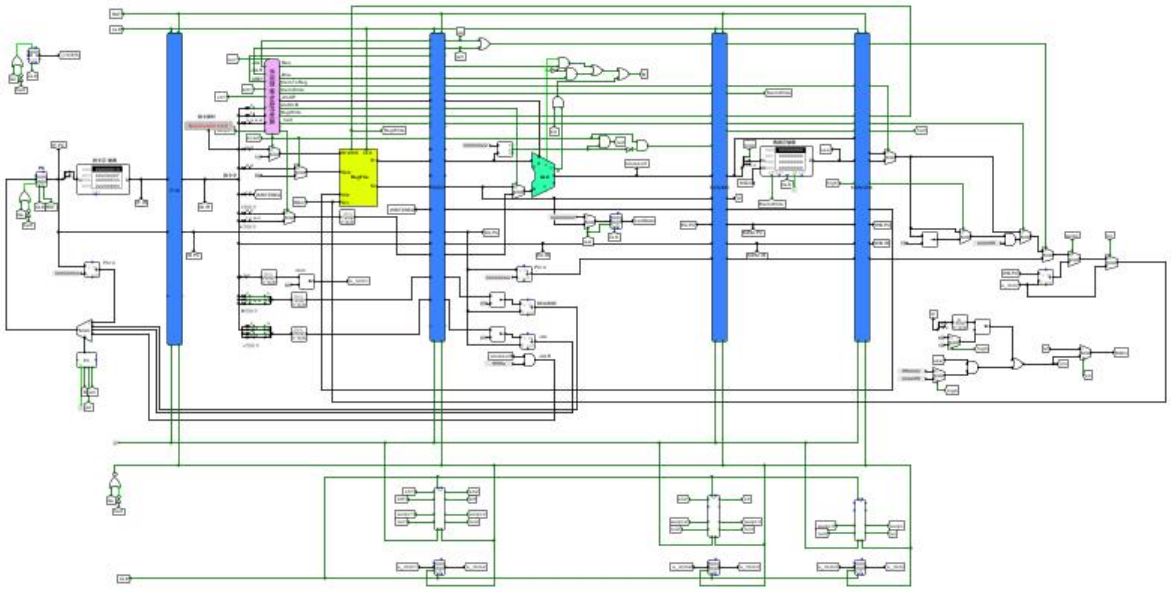

图 3.21 理想流水线电路设计

3.4 气泡式流水线实现

根据总体设计方案，我们需要再理想流水线上加以改进。首先需要设计一个气泡冲突处理模块，由**气泡冲突检测单元**和**气泡插入单元**组成然后将其插入到我们的理想流水线 CPU 电路中即可。

而气泡插入的实现则比较简单，当气泡冲突检测逻辑检测到冲突时，在 ID/EX 段实现 flush 操作刷新即可，同时在 IF/ID 端关闭使能一个周期即可，相当于清空了这两段的内容。只需在原先理想流水线中加入一个 DATAHAZZARD 信号即可，具体的设计如下：

图 3.22 气泡插入逻辑

气泡冲突检测单元的设计则是一个组合逻辑电路。重点就是在于判断出 ID 段指令使用的源寄存器是否在前两条指令中写入，只需要检查 EX、MEM 段的寄存器堆的写入控制信号 RegWrite 是否为 1，且写寄存器编号 WriteReg#是否和源寄存器编号相同即可。具体实现如下：

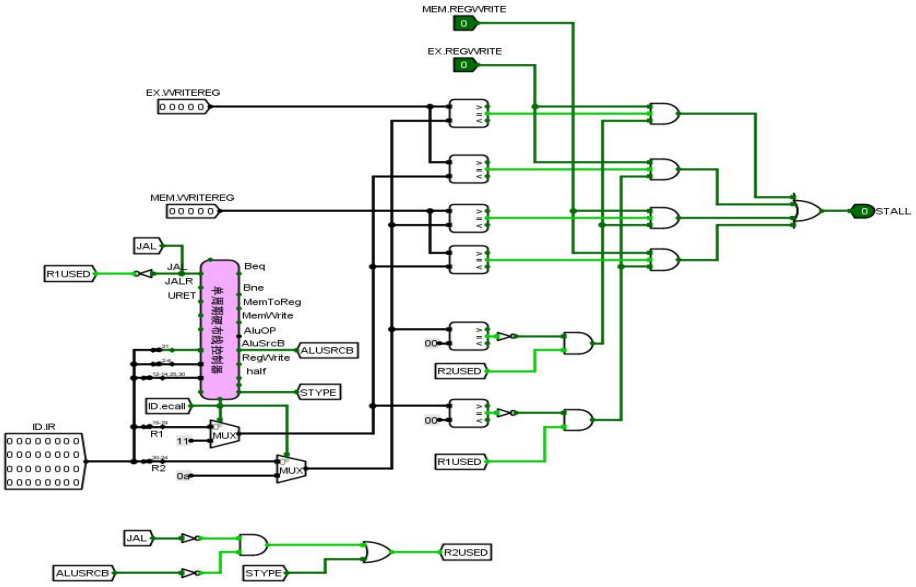

图 3.23 气泡冲突检测逻辑

最终设计完成后的气泡流水线如下：

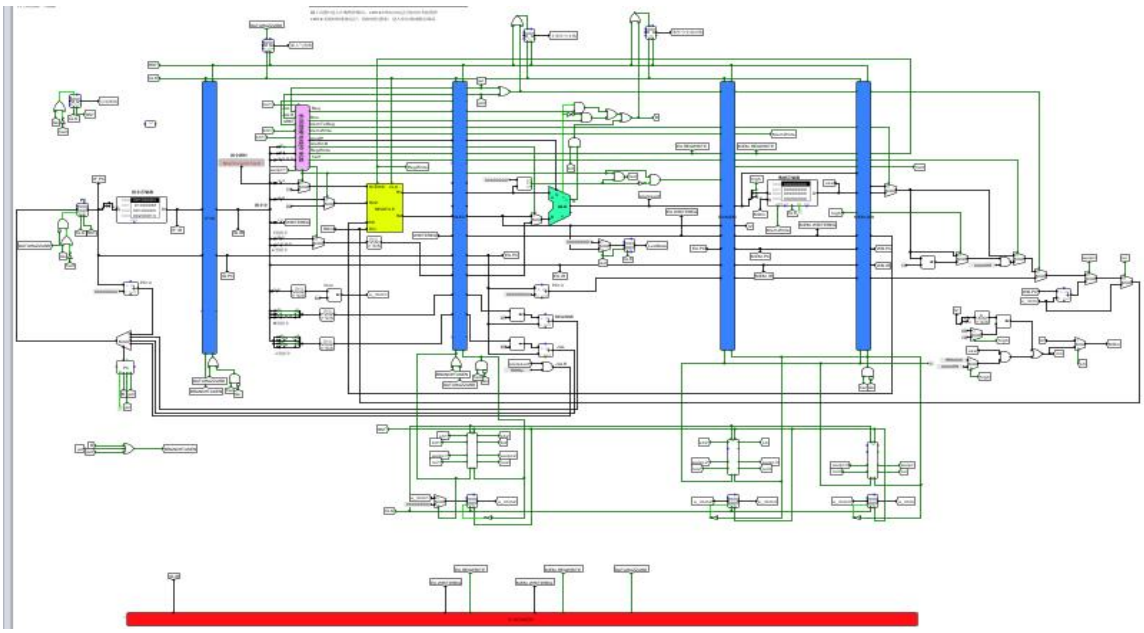

图 3.24 气泡流水线设计图

3.5 重定向流水线实现

采用重定向机制后，流水线中的相关处理逻辑必须进行适当的修订，由于重定向中 LoadUse 数据相关仍然需要通过插入气泡方式进行消除，所以相关处理逻辑应该**能检测出 LoadUse 相关**。其他方式下的数据相关都可以通过重定向的方式解决，相关处理逻辑只需要 在 ID 段生成两个重定向选择信号 R1Foward、R2Foward 传输给 ID/EX 流水寄存器即可。具体这两个信号的逻辑已经在 2.5 节中讲述，这里不过多赘述，设计完毕后的电路图如下图 3.25：

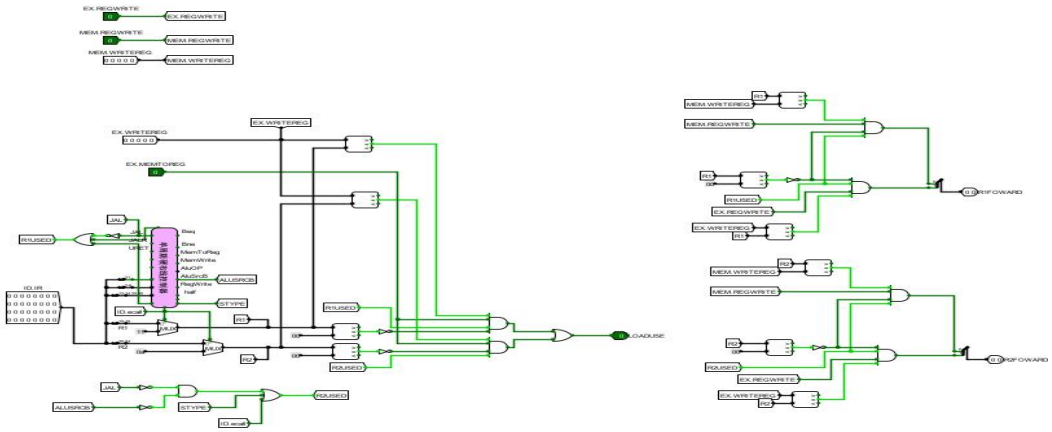

图 3.25 重定向处理逻辑实现

最终实现的重定向流水线如下图 3.26：

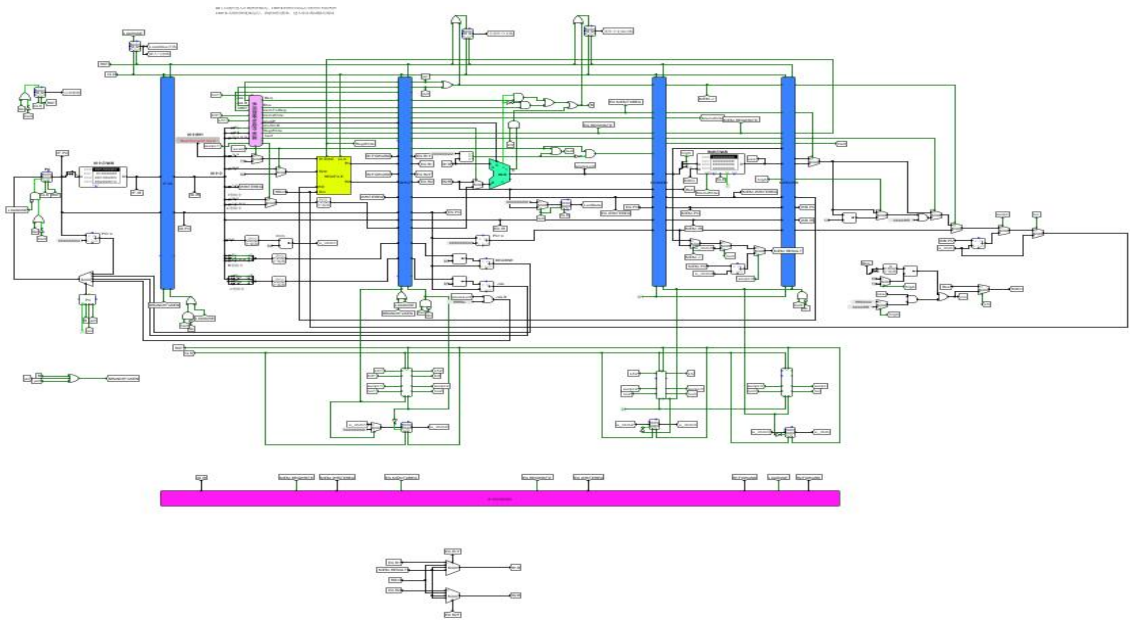

图 3.26 重定向流水线实现

3.6 动态分支预测机制实现

为了在 Logisim 上实现动态分支预测机制，我们首先需要实现分支历史表。根据总体方案设计中的内容，我们设计的 BTB 表如下：

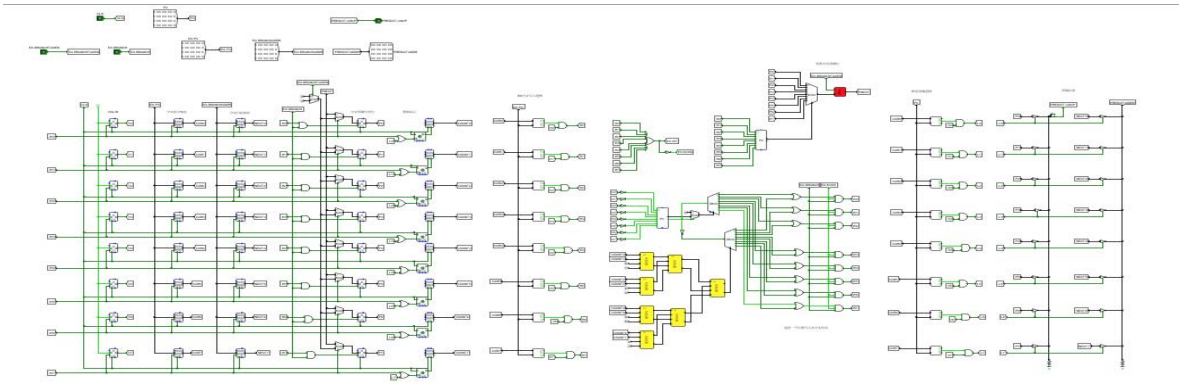

图 3.27 BTB 表电路设计

将分支预测逻辑加入重定向流水线中，每次当读取新 PC 时，增加一个选项。如果当前指令时跳转指令并且曾经读取过这个指令，我们便可以得到其预测的跳转地址，提前将其作为下一个周期的地址，这样可以省略后几个周期的流水等待时间。如果预测正确就大大提高效率，如果预测错误就插入气泡重新设置 PC 即可。具体设计如下：

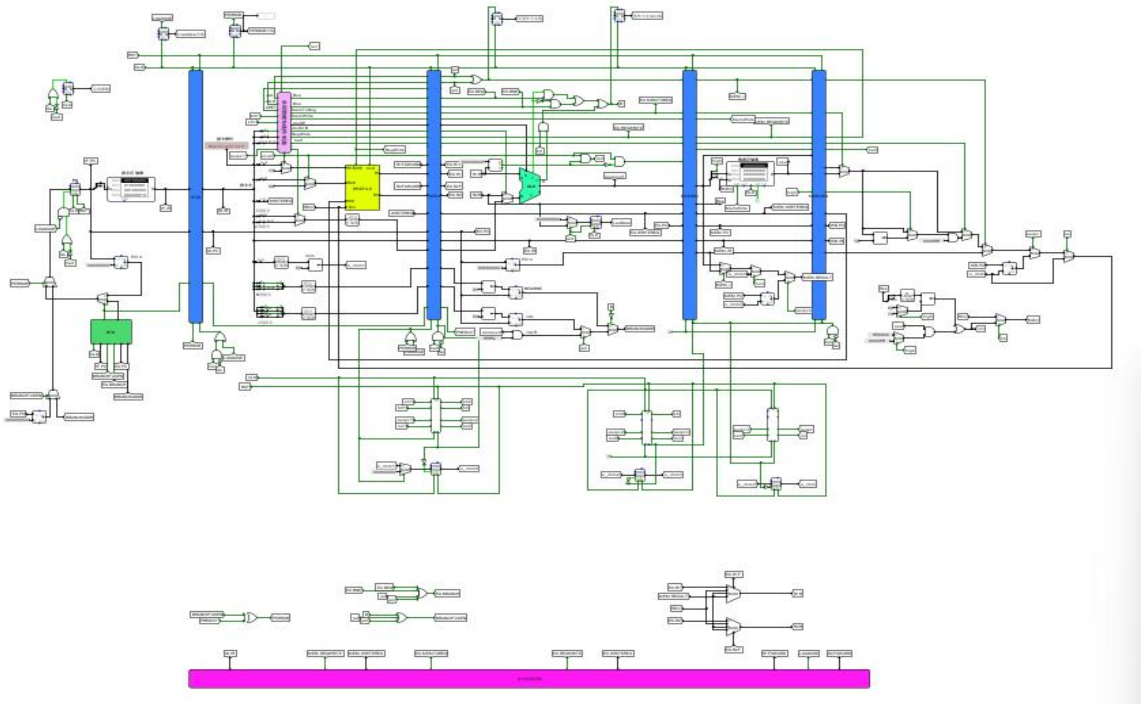

图 3.28 分支预测流水线设计

4 实验过程与调试

4.1 测试用例和功能测试

本实验共使用两种测试用例对 Logisim 上的各种 CPU 进行测试。同时 24 条指令 CPU 的设计也通过了头哥的测试。具体的测试用例如下：

（1）risc-v-benchmark_ccab：测试基本的 24 条指令和 4 条 CCAB 指令。

（2）risc-v 多级中断测试(EPC 硬件堆栈保护)：测试多级中断

（3）risc-v 单级中断测试

以上测试样例均通过测试。具体测试结果见下文。为了节省篇幅，其中 **CCAB 指令仅在分支预测中展示结果。**

4.1.1 单周期 CPU 测试

在单周期 CPU 中运行 risc-v-benchmark_ccab.hex 文件，正确完成预期功能，周期数为 1545，测试结果如下：

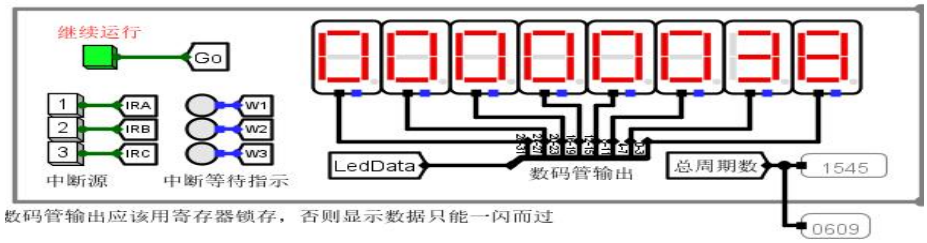

图 4.1 单周期 CPU 测试结果

4.1.2 中断测试

正确加载 hex 文件进入指令存储器中，并运行对应程序。

（1）单周期 CPU 单级中断

首先按下按钮 1，进入 1 号中断处理程序，满足我们的需求：

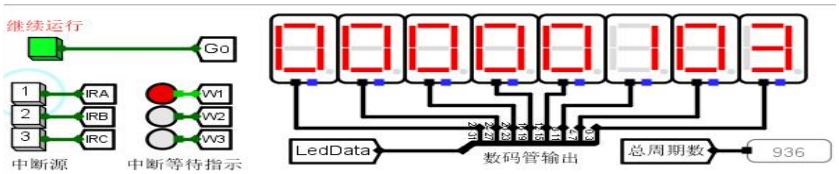

图 4.2 单级中断测试（按钮 1）

然后重新运行程序，首先按下按钮 1，当进入一号中断处理程序的时候按下按钮 2、3，紧接着会执行三号中断处理程序再执行二号中断处理程序：

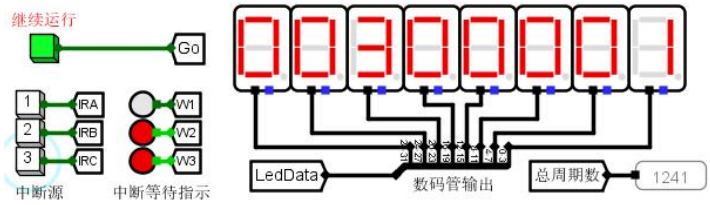

图 4.3 单级中断测试（按钮 1/2/3）

（2）单周期 CPU 多级中断

然后重新运行程序，首先按下按钮 1，当进入一号中断处理程序的时候按下按钮 2、3，这时会首先进入三号中断处理程序，然后再执行二号中断处理程序，最后执行一号处理程序，这也是多级中断与单级中断不同的一点，多级中断能够支持中断嵌套，也就是一个中断能够中断另一个中断：

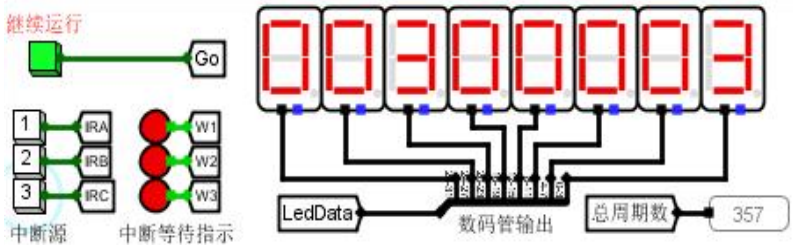

图 4.4 单周期多级中断测试（按钮 1/2/3）

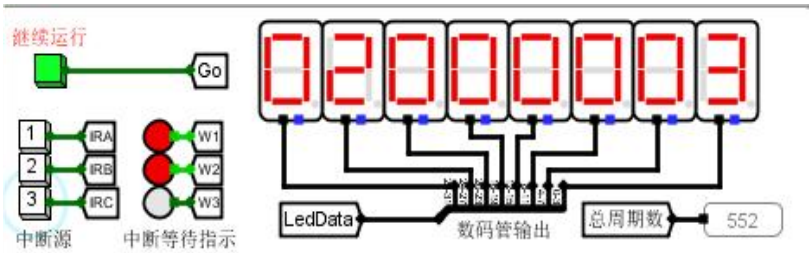

图 4.5 单周期多级中断测试（按钮 1/2/3）

（3）流水线单级中断

最终测试结果与单周期单级中断相同，已通过线下检查，故不再展示。

4.1.3 理想流水线测试

理想流水线时后续流水线的基础，已经包含在后续测试中，故不再多余测试。

4.1.4 气泡流水线和重定向流水线测试

分别对气泡流水线和重定向流水线加载 risc-v-benchmark_ccab.hex 文件，两种流水线均能够**正确完成预期功能**，最终在 CCAB 指令前暂停时结果如下：

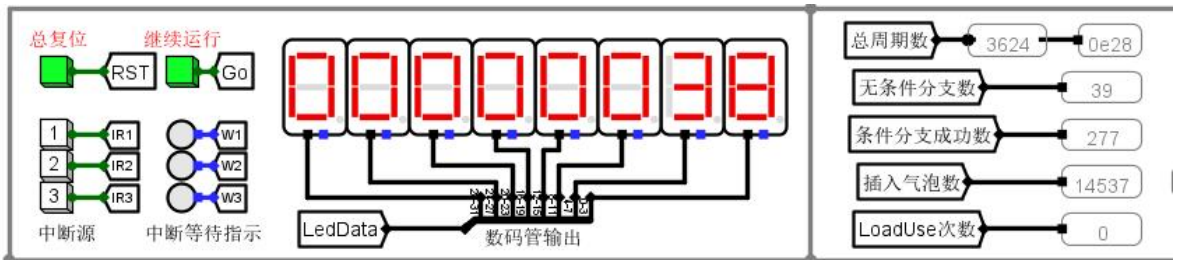

图 4.6 气泡流水线测试结果

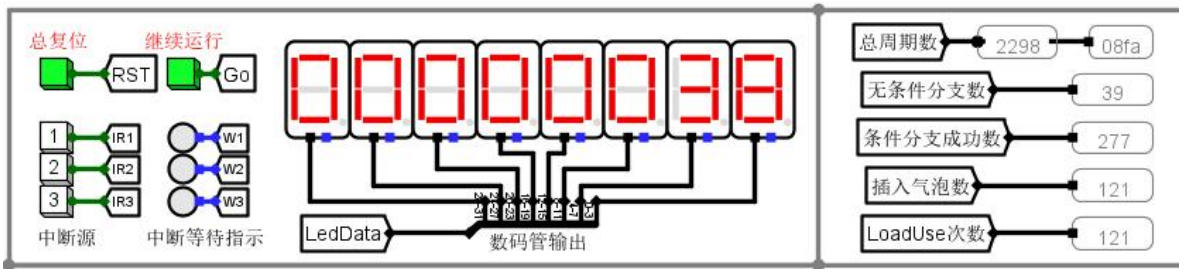

图 4.7 重定向流水线测试结果

通过比较可以发现重定向流水线插入气泡数大大减少。

4.1.5 分支预测流水线测试

对分支预测流水线加载加载 risc-v-benchmark_ccab.hex 文件并且记录其每一次暂停前后的输出值，验证得到该分支预测流水线能够正确完成 24 条指令和 CCAB 的额外指令功能，满足我们的需求：

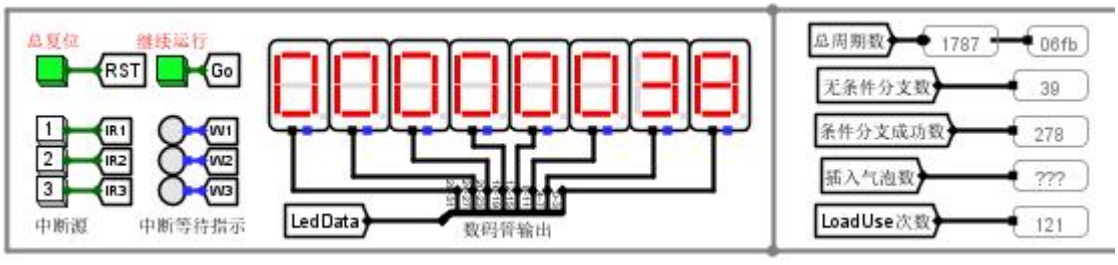

图 4.8 分支预测流水线测试结果

由于我们的文件已经添加了属于我们个人的 CCAB 指令，所以当我们继续按下 GO 按钮的时候，程序会继续执行我们的下一条 C 指令，每次按按钮都会往后执行一条

指令，我们的四条指令分别是 AUIPC，LUI，SH 和 BLT 指令，检验结果如下：

（1）AUIPC 指令

AUIPC 指令的预期结果为：依次输出 0x00430004 0x00430014 0x00430024 0x00430034 0x00430044 0x00430054 0x00430064 0x00430074。但是由于 AUIPC 指令是当前 PC 值加上一个立即数，所以我们还要考虑 PC 值的影响。由于具体实验中 PC 值无法读取，所以我们只能判断输出之间的间隔来测试该指令是否正确。经过验证发现 AUIPC 指令从 0x004582d4 开始一直运行到 0x00458354，符合增长要求，并且把 AUIPC 指令单独拿出来测试结果符合预期输出，我们可以认为该指令设计正确。

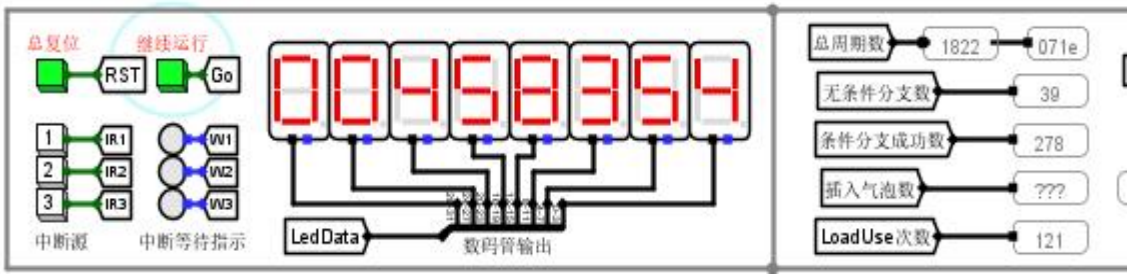

图 4.9 AUIPC 指令运行结束

（2）LUI 指令

LUI 指令预期结果为 0xfedcffff 0x0ba98000 0x07654000 0x03210000 …… 0xfedcffff 0x0ba98000 0x07654000 0x03210000 ，最终测试属于与预期相同：

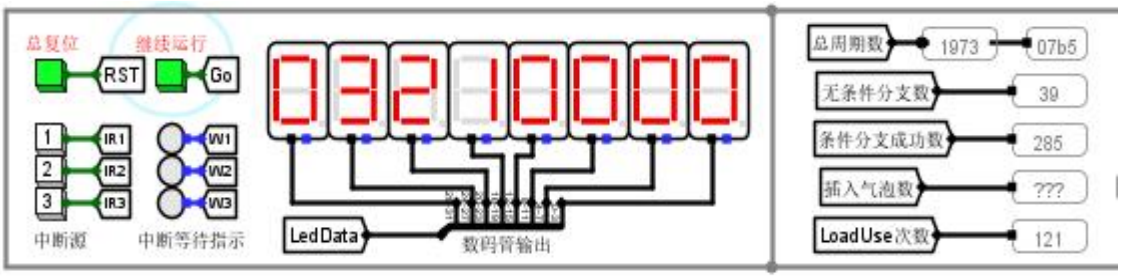

图 4.10 LUI 指令测试

（3）SH 指令

SH 指令预期结果为依次输出 0x00000001 0x00000002 0x00000003 …… 0x0000001f 0x00000020 0x00020001 …… 0x001e001d 0x0020001f，最终结果与预期相同：

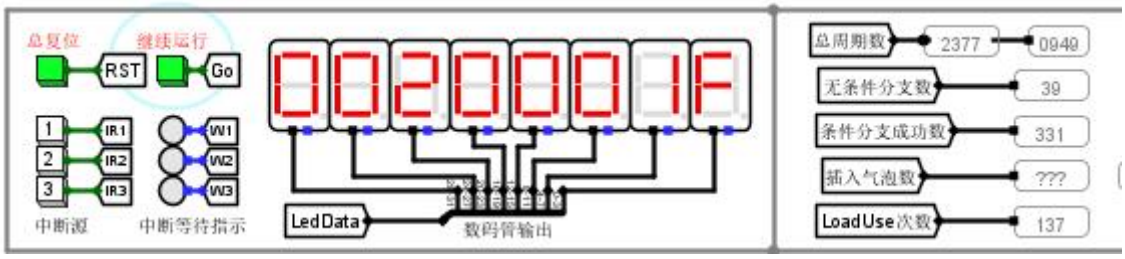

图 4.11 SH 指令测试

（4）BLT 指令

BLT 指令预期结果为依次输出 0xfffffff1 0xfffffff2 0xfffffff3 0xfffffff4 0xfffffff5 0xfffffff6 0xfffffff7 0xfffffff8 0xfffffff9 0xfffffffa 0xfffffffb 0xfffffffc 0xfffffffd 0xfffffffe 0xffffffff，最终结果与预期相同：

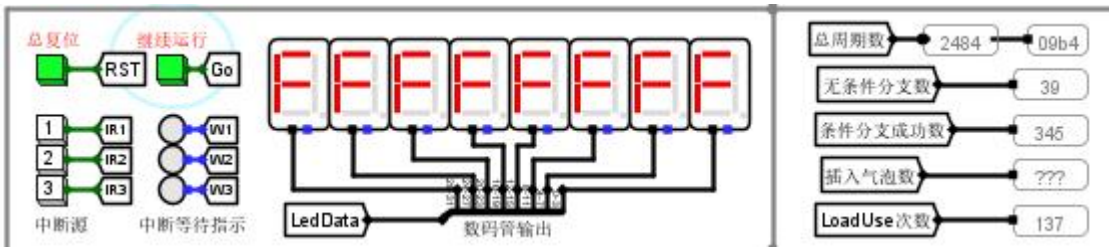

图 4.12 BLT 指令测试结果

由于分支预测流水线的设计基于以上几种流水线和单周期 CPU 的设计，所以仅在分支预测流水线中展示 CCAB 四条指令的输出。经过验证，所有待测 CPU 均通过了对应的测试用例，CPU 设计符合要求。

4.2 性能分析

分别统计不同类别的 CPU 上运行 risc-v-benchmark.hex 程序所需要的周期数，统计结果如下表 4.1 所示。

表 4.1 四种 CPU 运行周期表

CPU 类型	单周期 CPU	气泡流水线	重定向流水线	动态分支预测流水线
周期数	1545	3624	2298	1787

单周期 CPU 每一条指令都在一个周期内完成，不涉及复杂的流水线控制逻辑和冲突处理，因此花费周期数最短。

剩下三种流水线 CPU 运行 risc-v-benchmark.hex 所需的时钟周期数依次降低，说明这三种 CPU 的性能依次提高，符合预期。

4.3 主要故障与调试

4.3.1 重定向流水线运行故障

故障现象：重定向流水线无法正确运行 AUIPC 指令

原因分析：重定向流水线的处理逻辑是根据气泡流水线进行修改的，而在一开始的修改中没有考虑 CCAB 的问题，也就是判断 R1USED 时没有考虑 AUIPC 指令的关系。因为气泡流水线只需要判断当前 R1，R2 是否被使用即可，而重定向流水线在此基础上还得多判断一个 LOADUSE 信号。即使 R1 和 R2 满足重定向的条件，但是由于 LOADUSE 信号的原因还是得插入气泡。如果不对 R1USED 进行修改，那么很有可能本来无法正确重定向的部分被我们误以为能够重定向从而引发错误。

解决方案：在重定向冲突处理逻辑中吧 AUIPC 指令添加进去即可

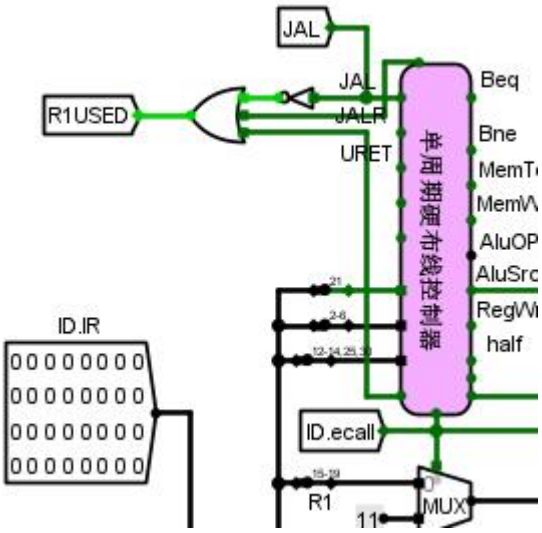

图 4.13 修改 R1USED 逻辑

4.3.2 气泡流水线条件分支跳转错误

故障现象：气泡流水线条件分支判断错误，程序未能正确跳转

原因分析：在判断 BLT 指令时，应当是 ALU<部分输出为 1 才能确定为一条 B 型指令，但是原本的测试中为考虑 BLT 指令的存在性，即直接将 ALU 运算器的< 接口连接到或门上，使得误判 B 型指令从而导致跳转错误

解决方案：同时判断当前指令为 BLT 指令和 ALU 输出<信号为 1 即可。

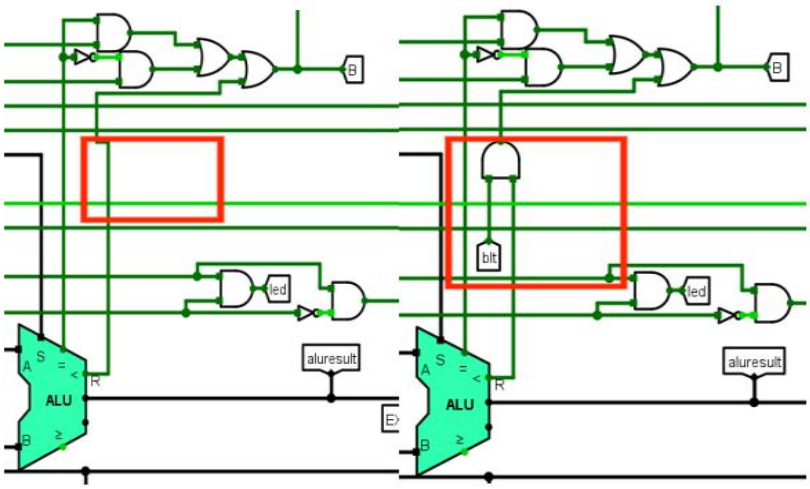

图 4.14 blt 指令判断修改前后

4.4 实验进度

表 4.2 课程设计进度表

时间	进度
第一天	复习组成原理 CPU 相关理论知识，阅读课设任务书和阅读 RISC-V 指令手册。完成运算控制器自动生成和控制信号自动生成电路 EXCEL 表格填写。
第二天	基于 EXCEL 表格完成运算控制器和控制信号电路生成并完成硬布线控制器
第三天	完成单周期 CPU 的设计和 CCAB 的设计，开始阅读流水线 CUP 相关知识
第四天	对单周期 CPU 进行修改，尝试按照五个模块为标准设计单周期 CPU
第五天	学习 5 段流水 CPU 架构:深入理解五段流水线的工作原理,完成 5 段流水 CPU 的接口部件实现。
第六天	将修改后的单周期 CPU 拆分成五个段并且用流水接口部件连接，初步实现理想流水线。
第七天	在理想流水线的基础上加入气泡机制，实现气泡流水线，学习重定向流水线和分支预测流水线的知识。
第八天	在理想流水线的基础上修改，实现重定向流水线和分支预测流水线。
第九天	在单周期 CPU 的基础上实现单级中断和多级中断设计并实现 EPC 寄存器堆栈和中断号栈，确保中断处理过程中的断点保存和恢复。
第十~十二天	在气泡流水线上实现单级中断，并对所有 CPU 进行测试，修复可能存在的问题，保证设计的可靠性

5 团队任务

5.1 团队任务简介

5.1.1 项目背景

本项目在传统井字棋的基础上进行了创新设计，将经典的 3×3 **棋盘**扩展为 5×5 **棋盘**，并引入了 **三人对战**模式。游戏规则保持与传统井字棋一致：三位玩家轮流在棋盘上落子，最先在横、竖或对角线上连续放置 5 **个棋子**的玩家获胜。此项目的设计不仅增加了游戏的趣味性与挑战性，也进一步体现了简单 CPU 在更复杂场景中的应用能力。

5.1.2 游戏规则

（1）棋盘扩展

棋盘大小由传统的 3×3 扩展为 5×5，提供了更大的游戏空间和更多策略选择。

（2）三人对战

三位玩家分别使用三种不同标记（如 X、O、V）轮流落子，每轮游戏按照固定顺序进行，确保公平性。

（3）胜负判定

首位在棋盘的横、竖或对角线上连续放置 3 个相同标记的玩家获胜。若棋盘填满且无玩家获胜，则游戏判定为平局。

5.1.3 实现方式

（1）硬件设计

基于 Logisim 平台设计的简单 CPU，扩展了内存容量以支持更大的棋盘状态存储。增加了逻辑判断模块，用于检测 3×3 棋盘上的连线情况。

（2）软件实现

编写了用于控制游戏逻辑的汇编代码，包括玩家输入处理、棋盘更新和胜负判定等功能。使用 CPU 的输入输出接口与玩家交互，实现了对三位玩家轮流输

入的支持。

（3）玩家交互

通过简单的输入设备（具体为 Logisim 中的按钮）实现玩家的坐标输入。棋盘状态通过输出设备（如 LED 矩阵或字符显示器）实时展示，便于玩家直观了解当前局势。

5.1.4 游戏特色

三人对战模式增加了游戏的策略深度，使得玩家需要同时考虑两位对手的行动，游戏过程更加紧张刺激。在更大的棋盘上，玩家需要更精准的布局与预判，对抗性更强。游戏的运行成功验证了简单 CPU 在更复杂场景中的运算能力和稳定性。

5.2 团队任务具体分工

我们小组由四名成员组成，成员名称和分工如下：

（1）田清林 CS2208

编写汇编代码，负责图案绘制模块，实现不同图案的绘制逻辑。

（2）方子豪 CS2208

设计硬件接口，连接汇编代码和 CPU、LCD 显示屏。将代码移植到硬件模块。实现中断处理的硬件原理，同时确保一一对应中断号和中断处理程序。

（3）刘彦哲 CS2203

编写汇编代码，负责棋局状态检测模块，实现状态检测与程序跳转逻辑。

（4）刘柯佟 CS2204

软硬件联调测试，完善状态检测逻辑，并完成 PPT 制作与视频录制。

5.3 硬件设计实现

接下来我具体讲一下我负责的任务的具体实现。

（1）像素显示模式

经过讨论我们决定使用 CS3410 组件中能够显示彩色图像的 LCD Video 部件来完成游戏的显示功能。如果在 CLK 的上升沿时 WE 为高电平，则在 LCD 屏幕上给定由 7 位无符号 X 和 Y 坐标指定的位置写入一个像素。像素颜色由 16 位 RGB 输入指定。RST 输入用于重置 LCD 屏幕。

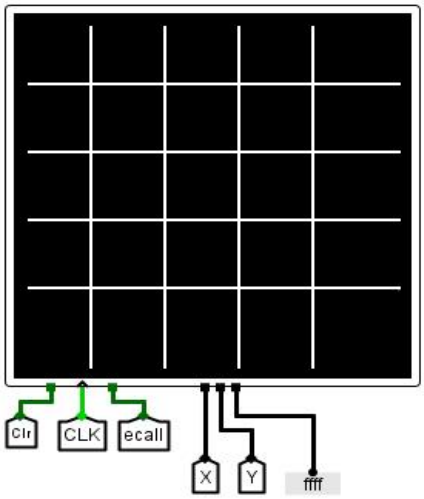

图 5.1 LCD 屏幕

（2）单级中断 CPU

团队任务中使用的 CPU 与我们个人任务中的 CPU 并无过大区别，不过由于负责软件编写的队友将程序的汇编代码导出后发现有**四万多行**，所以将取指令的地址扩大至 2-21 位，共可以容纳 104 万条指令（为了扩展其他功能铺垫）。

（3）键盘模拟中断

在一般的程序运行中，如果我们想写一个下棋的函数，当我们选择在一个位置下棋之后，应当会进入棋子绘制的函数，当棋子绘制完毕之后，再执行其他指令。在 Logisim 为了方便我们使用**按钮**来模拟函数调用。而为了实现函数调用，我们利用了单周期 CPU 的**中断机制**。即当按下按钮的时候，会出发对应编号的中断，从而将程序导向对应编号的中断处理程序，当然这里的中断处理程序也是我们编写的**棋子绘制代码**。这样就可以巧妙地通过中断机制来实现棋子的绘制了。

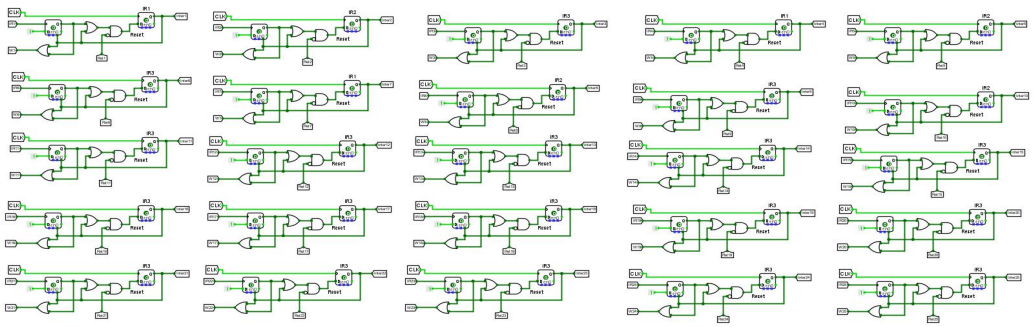

图 5.2 键盘模拟中断

（4）中断处理及恢复

首先根据不同的按键，选择不同的地址输入到 interAddr 中，执行相应的中断处理程序并关中断。此时，开始在棋盘上绘制图像，图像绘制结束后开中断，等待下一次按键。

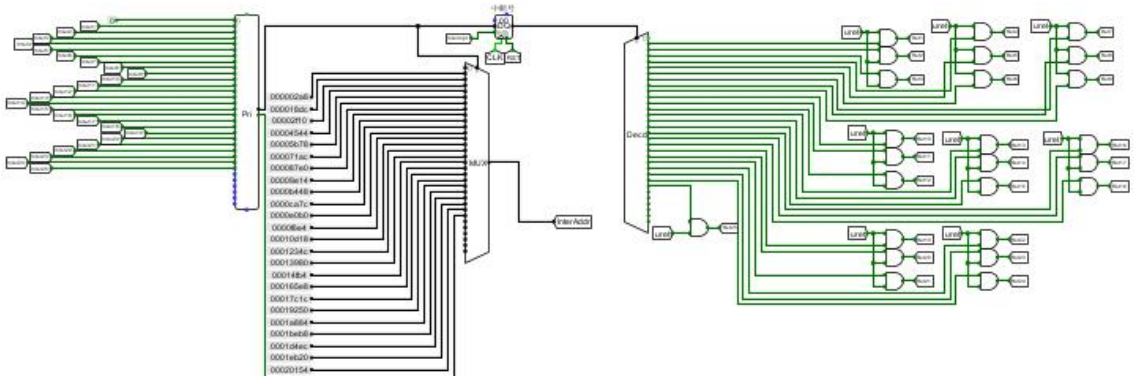

图 5.3 中断处理程序断点

5.4 软件设计实现

由于运行机器性能不足，需要尽可能地减少运行开销，因此决定将全局状态变量和各角色状态变量全部存放在寄存器中，对棋局进行状态压缩，从 0~24 位依次表示 1~25 个格子。使用三个变量分别存储三名角色的下棋记录。事先对寄存器的使用进行规定如下表 5.1：

表 5.1 寄存器使用规定

寄存器	用途
a1	当前下棋角色(0, 1, 2)
a2	当前回合数(0-25)
a3	记录当前游戏状态(进行，获胜，平局)
a7	控制系统调用（停机或绘制像素）
t2	是否清空界面
t6	用于开关中断
ra/sp	写入 LCD 的坐标(X、Y)
S2,s3	当前棋子的中心位置坐标(X、Y)
t0,t1,t3	不同角色棋局状态(X、O、 √)

程序总共分为三个模块，初始化模块，玩家操作模块和状态判断模块。下面依次介绍其功能：

（1）初始化模块：

考虑到 cpu 的性能，本程序的变量都存储至寄存器中。在程序的初始化阶段，要给部分存储全局变量的寄存器赋初始值。本项目的棋盘设计为 5×5 的格子，那么横竖分别要绘制四条。由于屏幕尺寸为 128×128，那么每条线的范围为[3,124]，利用循环来依次绘制四条横线与四条竖线。

```
addi t6,zero,1   ##初始化关中断
addi a5,zero,4   ##横竖各画4条线
addi a6,zero,124 ##线从3到124号的像素
addi a7,zero,0   ##初始a7
addi a4,zero,0
addi ra,zero,0  ##绘制4条横线
Hengxian:
addi ra,ra,24
addi sp,zero,3
Heng_Dian:
addi sp,sp,1
ecall            ##画像素点
bne sp,a6,Heng_Dian
addi a4,a4,1
bne a4,a5, Hengxian
addi a4,zero,0
addi sp,zero,0  ##绘制4条竖线
Shuxian:
addi sp,sp,24
addi ra,zero,3
Shu_Dian:
addi ra,ra,1
ecall            ##画像素点
bne ra,a6,Shu_Dian
addi a4,a4,1
bne a4,a5, Shuxian
```

图 5.4 初始化模块部分代码

（2）玩家操作模块：

首先需要判断玩家回合，这可以通过寄存器 a1 的值进行判断，0 表示 X 回合，1 表示 O 回合，2 表示 √ 回合，根据 a1 的值来判断当前属于谁的回合，并进行跳转。由于 beq/bne 指令跳转范围有限([-4096, 4094]字节)，此处使用 j 指令(jal,包含 20 位的有符号偏移量)进行跳转。

判断回合后，设置棋盘状态变量的某一位，之后便需要在棋盘上绘制对应的标志，使用系统调用来绘制像素点。

对于不同的图形，首先确定中心点位置，之后计算相对位置绘制图形。具体来说，对于 X，先确定交叉点的横纵坐标，然后绘制两条斜线，同理，O 是一个八边形，√是一个直角。

```
#####################
####按键2中断程序#####
#####################
Interupt_2:  ##中断2入口
addi a2,a2,1
addi a4,zero,1
 ##为O下时，跳到对应处理位置
bne a1,a4,skip_O_jump_2
j O_jump_2
skip_O_jump_2:
 ##为D下时，跳到对应处理位置
addi a4,zero,2
bne a1,a4,skip_D_jump_2
j D_jump_2
skip_D_jump_2:
addi s2,zero,36 ##设置Xx中心
addi s3,zero,12 ##设置Xy中心
##画X
addi ra,s2,-10
addi sp,s3,-10
addi a4,zero,20
Draw_X1_2:
ecall
addi ra,ra,1
addi sp,sp,1
addi a4,a4,-1
bne a4,zero,Draw_X1_2
```

图 5.5 中断跳转代码展示

（3）状态判断模块：

首先为了判断是否三字连成一条线，以每个格子为基准，依次判断其八个方向是否有三个相同棋子相连的情况，将所有可能的情况穷举出来，如果发现相连，则跳转至将 a3 赋值后返回。比如(t1&(t1<<1)&(t1<<2)&(1<<2))== 1<<2 就是判断横着的时候是否有三个棋子相连的情况。

至于判断谁是赢家，根据 a3 的值，来判断哪个玩家最终赢得比赛。具体来说，为 0 时表示正在下棋，为 1 时表示 X 胜利，为 2 时表示○胜利，3 表示平局，4 表示 D 胜利。

5.5 结果展示

下面展示一下正常游戏结束 X 获胜的图片和玩家获得平局的结果图片：

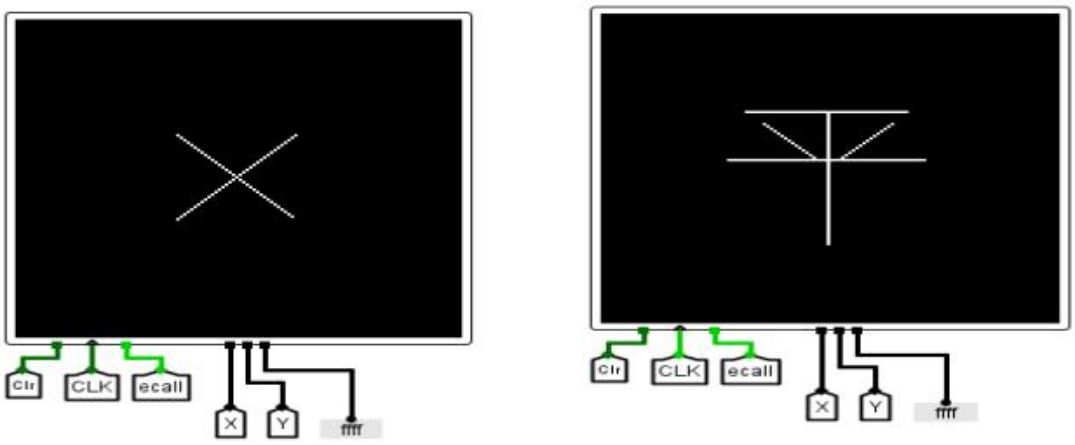

图 5.6 X 胜利和双方平局截图

6 设计总结与心得

6.1 课设总结

本次实验中，完成了如下工作：

1） 单周期 CPU 设计与实现：设计并实现了支持 24 条基础指令和 4 条扩展指令的单周期 CPU，确保其功能完整性和正确性。

2） 中断机制设计与实现：设计并实现了带单级中断和多级中断的单周期 CPU，同时在流水线中集成了单级中断机制，确保系统能够正确处理中断请求。

3） 流水线 CPU 的设计与实现：设计并实现了理想流水线 CPU，气泡流水线 CPU，重定向流水线 CPU，动态分支预测流水线 CPU 等类别 CPU。

4） 平台测试与检查

5） 团队任务（三人扩展井字棋）

6） 课程报告撰写

6.2 课设心得

在硬件综合训练课程中，我完成了从单周期 CPU 到五段流水线 CPU，再到带分支预测和中断功能的流水线 CPU 的设计与实现，最后还完成了团队任务。这段学习过程让我受益匪浅，不仅加深了对计算机体系结构的理解，也锻炼了实际动手能力和解决问题的能力。

课程初期，通过 Logisim 平台实现单周期 CPU，我对指令集架构、数据通路设计和控制信号生成等核心概念有了直观的认识。这一步骤强化了我对理论知识的掌握，让我清晰地认识到硬件设计中每一个模块的功能和作用。当任务升级到五段流水线 CPU 时，问题的复杂性明显提高。我不仅需要处理指令的并行执行，还需要解决数据冒险和控制冒险等问题。通过查阅资料和反复调试，我深刻体会到流水线技术在提升 CPU 性能中的关键作用。

课程过程中，我与同学们进行了广泛的交流。与同学交流其中的思想还是能让我受益匪浅。他们处理一些指令的方法也为我的个人任务打开了新世界的大门，提供了新的思路，对我的帮助很大。

通过这门课程，我不仅掌握了 CPU 的设计方法和关键技术，还培养了严谨的逻辑思维和独立解决问题的能力。这段经历让我更加认识到硬件设计的重要性，以及团队合作在项目开发中的价值。

总的来说，“硬件综合训练”是一门极具挑战性和收获的课程，它为我将来的学习和研究奠定了坚实的基础，也激励我在计算机硬件领域继续探索和进步。

但是我还是要对课程提出几点建议：

1. 课设任务书过于冗长，而且没有切入重点，直接将压缩包丢给学生一开始学生可能会云里雾里不知道要干什么。可以适当的进行修改删减一些内容。

2. 课程设计检查应当多安排几次，时间最好一开始就规定好，不然临时规定有的同学可能很难调出时间检查，特别是期末周的时候，复习期间去花费几个小时排队检查是很难受的。

3. CCAB 指令在最好带上其对应的功能以及预期输出，同时提醒同学有的指令输出值并不固定，比如我的 AUIPC 指令与 PC 值相关，所以是不固定的，这可能会导致部分同学认为自己电路设计错误。

参考文献

[1] DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第5版).北京：机械工业出版社.

[2] David Money Harris(美).数字设计和计算机体系结构（第二版）. 机械工业出版社

[3] 谭志虎，秦磊华，吴非，肖亮.计算机组成原理. 北京：人民邮电出版社，2021年.

[4] 谭志虎，周健，周游.计算机组成原理实验指导（基于RISC-V在线实训）. 北京：人民邮电出版社，2024年.

[5] 曹强，施展.计算机系统结构（微课版）. 北京：人民邮电出版社，2024年.

一、原创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签字：方子豪