华中科技大学

2021

计算机组成原理　　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：	CS1806
学　　号：	U201814655
姓　　名：	杨雨鑫
电　　话：	18162318665
邮　　件：	2056404413@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	SRAV	算数右移	
29	XORI	立即数异或	
30	SH	存半字	
31	BGTZ	大于 0 跳转	

2 总体方案设计

2.1 单周期 CPU 设计

本次我们采用的方案是微程序控制，且主、控存分开的方案，即采用微程序控制方式，实现主存储器（MM）和微程序控制存储器（CM）不共用一个存储器的方式完成方案的设计。

总体结构图如图 2.1 所示。

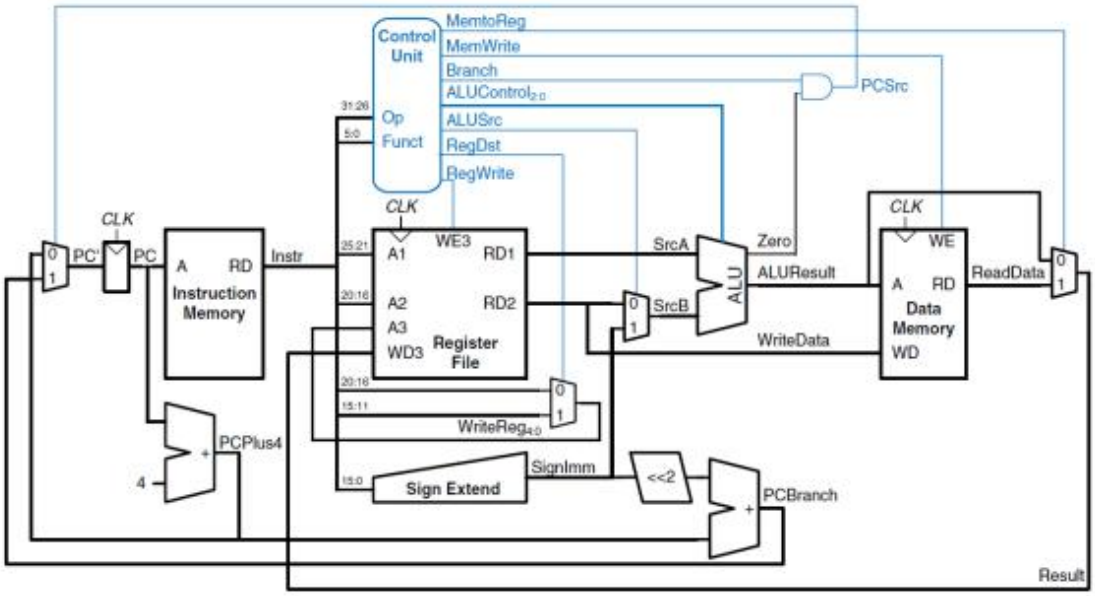

图 2.1 总体结构图

2.1.1 主要功能部件

在电路设计过程中使用的是已经封装好的元器件。

1. 程序计数器 PC

程序计数器直接使用一个计数器实现，实现主存储器（MM）和微程序控制存储器（CM）不共用一个存储器的方式完成方案的设计。选择的触发方式为上升沿触发，PC 数据位宽为 32 位。

2. 指令存储器 IM

指令存储器直接使用一个 ROM 元器件实现，通过加载 hex 格式的文件完成指令的录入，通过输入的地址访问需要的指令，地址位宽为 10，数据位宽为 32.

3. 运算器

表 2.1 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
X	输入	32	操作数 X
Y	输入	32	操作数 Y
ALU_OP	输入	4	运算器功能码，具体功能见下表
Result	输出	32	ALU 运算结果
Result2	输出	32	ALU 结果第二部分，用于乘法指令结果高位或除法指令的余数位，其他操作为零
OF	输出	1	有符号加减溢出标记，其他操作为零
UOF	输出	1	无符号加减溢出标记，其他操作为零
Equal	输出	1	Equal=(x==y)?1:0, 对所有操作有效

4. 寄存器堆 RF

寄存器堆使用封装好的元器件完成，包含了 32 个寄存器。

2.1.2 数据通路的设计

表 2.2 指令系统数据通路框架

指令	PC	IM	RF				ALU			DM		Tube
			R1#	R2#	W#	Din	A	B	OP	Addr	Din	

2.1.3 控制器的设计

首先对于控制信号进行统计，包括各个主要部件所需要输入的控制信号，以及数据通路合并表中所示的具有多输入的主要部件需要进行输入选择的控制信号，并且对各个统计信号的各种取值情况进行定义，得到的控制信号以及说明如表 2.3。

表 2.3 主控制器控制信号的作用说明

控制信号	取值	说明
R1	0	寄存器堆 R1 口读取 rs 字段指示寄存器的值
	1	寄存器堆 R1 口读取 2 号寄存器的值
MemToReg	0	写回 ALU 运算器运算结果
	1	写回数据存储器读取的数据
MemWrite	0	数据存储器写禁止
	1	数据存储器写开放
ALU_SRC	0	寄存器堆 R2 传出的数据送 ALU 运算器的 B 操作数
	1	指令低 16 位的扩展数据送 ALU 运算器的 B 操作数
RegWrite	0	寄存器堆写禁止
	1	寄存器堆写开放
SysCall	0	不使用系统调用
	1	使用系统调用
SignedExt	0	使用符号扩展
	1	使用以 0 扩展
RegDst	0	寄存器堆 W 口写 rt 字段指示寄存器
	1	寄存器堆 W 口写 rd 字段指示寄存器
Beq	0	当前指令不为 BEQ
	1	当前指令为 BEQ
Bne	0	当前指令不为 BNE
	1	当前指令为 BNE
JR	0	当前指令不为 JR

控制信号	取值	说明
	1	当前指令为 JR
JMP	0	当前指令不为 JMP
	1	当前指令为 JMP
JAL	0	当前指令不为 JAL
	1	将下条指令地址（PC+4）写回寄存器堆（寄存器号 0x1F）
SH	0	当前指令不为 SH
	1	当前指令为 SH
BLTZ	0	当前指令不为 BLTZ
	1	当前指令为 BLTZ
SRAV	0	当前指令不为 SRAV
	1	当前指令为 SRAV

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。

2.2 中断机制设计

2.2.1 总体设计

中断处理是在 CPU 执行过程中遇到中断事件，存储当前状态下的 PC 寄存器的值，接着转移到目标地址指令进行中断程序的执行，在中断程序结束后通过加载存储的 PC 寄存器的值返回中断处并继续进行指令的执行。MIPS 中断处理机制的实现需要分为硬件和软件的支持。中断设计要求完成单级中断和多级中断的设计。

1. 单级中断

单级中断指的是在一个中断程序的执行过程中，会产生一个中断信号，接着会发生关中断，阻碍其他的中断请求，PC 寄存器切换到中断子程序设置 PC 中断寄存器存储当前下一步的 PC 指令的地址完成现场保护功能，在执行完中断子程序之后，开中断，返回 ERET 信号，从 PC 中断寄存器中加载中断前的 PC 地址，重新回到主程序执行阶段。

2. 多级中断

多级中断相比于单级中断，首先需要改变的结构是，需要建立一个 PC 中断寄存器堆栈来完成对多级中断的现场恢复。此外，根据实验要求，实现多级中断时需要对三种不同的中断设置优先级从而限制多级中断的嵌套次数以及执行顺序。最后从硬件堆栈中读取出 PC 值恢复现场。

2.2.2 硬件设计

中断硬件支持需要满足以下的几个条件：

1. 生成中断的请求逻辑
2. 完成多级中断的优先级判断，从而确定执行顺序
3. 终端隐指令，完成中断隐周期的设计
4. PC 中断寄存器设计以及 PC 中断寄存器堆栈的设计

2.2.3 软件设计

1.单级中断

根据实验指导书给出的设计构建电路，通过 mars 反汇编得到的十六进制文件，定位到中断子程序入口的偏移地址。通过优先编码器作为多路选择器的选择端选择需要进入的中断子程序入口地址。最后通过检测出 ERET 信号完成对对应的中断信号的清空。

2.多级中断

多级中断的程序代码与单级中断思路基本相同。在硬件方面实现不同，除了同样需要保存现场和恢复之外，需要对 EPC 的内存堆栈进行现场保护。通过开中断指令 MFC0,和关中断指令 MTC0 进行开关中断，将 PC 寄存器的数据进行保护和恢复。

2.3 流水 CPU 设计

2.3.1 总体设计

传统的多周期 CPU 通过复用器件，异步控制，变长指令周期的方式，而指令流水线所实现的多周期 CPU 为多指令并行，可以很大程度上提升性能。五段流水 CPU 将指令执行分为 5 个阶段，如图 2.2 所示：

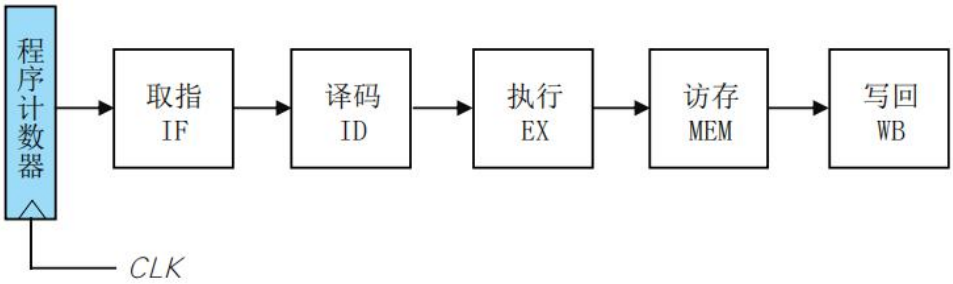

图 2.2 指令运行分段

IF：取指令阶段，以程序计数器 PC 中的数据作为访问地址，从指令存储器中取出指令并放入指令寄存器 IR；同时 PC 值加 4，PC 指向按照顺序的下一条指令。

ID：译码阶段，对 IR 指令寄存器中的内容进行译码，通过分线器得到所需要的位数，并用 IR 中的寄存器地址去访问通用寄存器组，读出所需的操作数。译码操作通过控制器进行，控制器通过硬布线逻辑输出每条指令的对应的控制信号，这些控制信号将和指令一起传送到执行阶段。

EX：执行阶段，不同的指令执行的操作不同，指令从寄存器堆读取需要运算的操作数交给 ALU 运算器，ALU 在控制信号下进行不同的运算操作，得到的结果传入下一个阶段。

MEM：访存阶段，需要对数据存储器进行读写操作，注意到这里的读取方式的地址是字地址而不是字节地址，访存地址由 EX 执行段 ALU 的计算结果给出，根据 ALU 的计算结果和控制信号写使能让数据进行读写，然后将结果送入写回段。

WB：写回阶段，写回指令的执行结果，如果执行的指令是计算类指令，则直接将 ALU 的计算结果进行写回，若指令包含有访存操作，则需要将 MEM 阶段的访存结果写回。写回结果的选择由译码阶段产生的控制信号来完成。

2.3.2 流水接口部件设计

流水接口部件即为五个流水段的设计，需要把整个流水线分成四个流水部件 IF/ID、ID/EX、EX/MEM、MEM/WB 这四个阶段。流水接口本质上就是可以实现同步清零的并排的寄存器来存储相关需要保存的信号的寄存器，所以在设计的阶段中，需要在每个并行的信号前添加一个多路选择器来完成同步清零的功能，不同段根据位置不同，需要保存的信息不同分别进行选择和连接接入即可。

在设计的过程中，注意每一个流水部件之间的封装接口位置要尽量保持一致，便于连线美观，按照提供的实验参考书上的电路实验图完成流水接口部件设计。

2.3.3 理想流水线设计

对于理想流水线的设计，相比于单周期 CPU 设计来说，把指令执行阶段分为五步来执行，通过流水接口部件完成了控制信号的传递，同时各段传输延迟一致，不存在等待现象，取最慢操作步骤的同步。不同阶段之间无共享资源，各段完全并发，不会发生访问冲突。进入流水线的对象不受其他阶段影响，即指令间的相关和依赖都不存在，仅运行无数据相关，无分支相关的程序。

2.4 气泡式流水线设计

通过插入气泡的方式来避免相邻的指令发生数据冲突访问或是不同步的问题。

具体操作：当流水线中执行分支指令时会存在误取的指令，我们对 IF/ID,ID/EX 进行清零操作，即插入了两个气泡，相当于把下一条指令延迟了两个时钟周期，注意在清零的时候必须做到同步清零。下面列出有可能出现的需要插入气泡的情况：

1. 先写后读冲突
2. 先读后写冲突
3. 写侯写冲突

对于上面出现的冲突情况，通过对 RS,RT 代表对应的寄存器，通过数据相关检测逻辑判断得到数据相关逻辑：

$$\begin{aligned}\text{DataHazzard} = {} & \text{RsUsed} \,\&\, (rs\neq 0) \,\&\, \text{EX.RegWrite} \,\&\, (rs==\text{EX.WriteReg\#}) \\ & + \text{RtUsed} \,\&\, (rt\neq 0) \,\&\, \text{EX.RegWrite} \,\&\, (rt==\text{EX.WriteReg\#}) \\ & + \text{RsUsed} \,\&\, (rs\neq 0) \,\&\, \text{MEM.RegWrite} \,\&\, (rs==\text{MEM.WriteReg\#}) \\ & + \text{RtUsed} \,\&\, (rt\neq 0) \,\&\, \text{MEM.RegWrite} \,\&\, (rt==\text{MEM.WriteReg\#})\end{aligned}$$

进一步综合控制冲突处理逻辑表达式可知流水线清空信号的逻辑表达式如下：

$$\text{Stall} = \text{DataHazzard}$$

$$\text{PC.EN} = \sim\text{Stall}$$

$$\text{IF/ID.EN} = \sim\text{Stall}$$

$$\text{IF/ID.CLR} = \text{BranchTaken}$$

$$\text{ID/EX.CLR} = \text{Flush} = \text{BranchTaken} + \text{DataHazzard}$$

2.5 数据转发流水线设计

相比于使用气泡流水线来解决冲突问题，重定向方式很好的节约了时钟周期的利用率，重定向流水线先不考虑 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。

重定向逻辑通过在 EX 阶段添加两个多路选择器，帮助 MEM/WB 阶段的操作直接快速的放入 ALU 进行计算，避免了一些气泡的插入，从而进一步提高了流水线的效率。同样的，和气泡流水线一样，重定向流水线也需要田间数据相关检测模块以及清零信号的生成，具体生成的逻辑如下：

LoadUse = RsUsed & (rs≠0) & EX.MemRead & (rs==EX.WriteReg#)

+ RtUsed & (rt≠0) & EX.MemRead & (rt==EX.WriteReg#)

IF (RsUsed & (rs≠0) & EX.RegWrite & (rs==EX.WriteReg#))

RsFoward = 2

else IF (RsUsed & (rs≠0) & MEM.RegWrite & (rs==MEM.WriteReg#))

RsFoward = 1

else RsFoward = 0

Stall = LoadUse # Load-Use 相关时要暂停 IF、ID 段指令执行

IF/ID.CLR = BranchTaken

ID/EX.CLR = Flush = BranchTaken + LoadUse

PC.EN = ~Stall

2.6 动态分支预测机制

在重定向的基础上为了节约发生动态分支时插入过多的气泡，决定使用动态分支预测技术针对分支跳转指令进行预测。通过对分支指令的跳转历史进行 cache 存储，从而实现命中就发生快速跳转事件，提升了流水线效率。具体 BTB 逻辑设计和 cache 类似，BTB 表放在 IF 段利用 PC 寄存器的值作为关键字进行全相联比较判断是否发生预测命中，决定下一条指令是 PC+4 还是 BTB 中的分支预测地址。如果预测失败，则按照之前的处理方式插入气泡。相关处理逻辑的输出信号以及各流水线寄存器使能信号逻辑如下：

Stall = LoadUse

IF/ID.CLR = PredictErr

ID/EX.CLR = Flush = PredictErr + LoadUse

PC.EN = ~Stall

IF/ID.EN = ~Stall

通过以上的的逻辑可以帮助我们设计出什么时候需要清空流水接口部件，直接分别连接到各自部件的清零端口即可。

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）

① Logism 实现：

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。

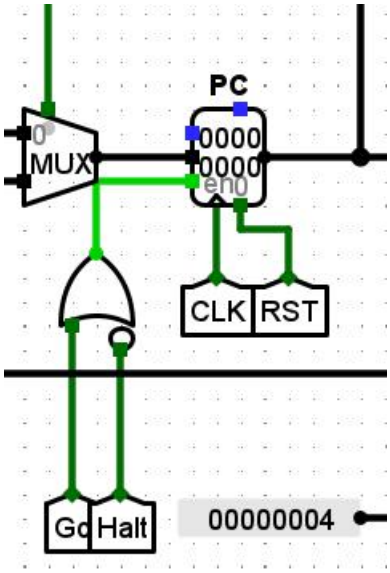

图 3.1 程序计数器（PC）

2) 指令存储器（IM）

① Logism 实现：

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。

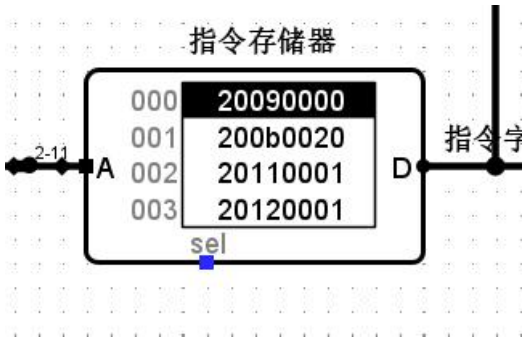

图 3.2 指令存储器（IM）

3.1.2 数据通路的实现

本次课程设计采用的工程化的设计模式，一次性构建所有的数据通路。主要实现方法为，对于每一条指令，将其改写成 RTL（Register Transfer Level），忽略控制类信号，仅保留数据类信号，根据 RTL 功能填写对应指令的数据通路表，描述五大部件之间的连接关系，记录各部件输入端数据来源。

根据总体方案设计中数据通路设计那一小节的详细内容，具体分析每一条指令在执行过程中各个主要部件的输入和输出端口的连接，完成指令系统数据通路表的填写，如表 3.1 所示。

表 3.1 指令系统数据通路表

指令	PC	IM	RF				ALU			DM	
			R1#	R2#	W#	Din	A	B	OP	Addr	Din
SLL	PC + 4	PC		rt	rd	Alu	R2	Imm	0		
SRA	PC + 4	PC		rt	rd	Alu	R2	Imm	1		
SRL	PC + 4	PC		rt	rd	Alu	R2	Imm	2		
ADD	PC + 4	PC	rs	rt	rd	Alu	R1	R2	5		
ADDU	PC + 4	PC	rs	rt	rd	Alu	R1	R2	5		
SUB	PC + 4	PC	rs	rt	rd	Alu	R1	R2	6		
AND	PC + 4	PC	rs	rt	rd	Alu	R1	R2	7		
OR	PC + 4	PC	rs	rt	rd	Alu	R1	R2	8		
NOR	PC + 4	PC	rs	rt	rd	Alu	R1	R2	10		

指令	PC	IM	RF				ALU			DM	
			R1#	R2#	W#	Din	A	B	OP	Addr	Din
SLT	PC + 4	PC	rs	rt	rd	Alu	R1	R2	11		
SLTU	PC + 4	PC	rs	rt	rd	Alu	R1	R2	12		
JR	Reg[rs]	PC	rs						X		
SYSCALL		PC	rs	rt					X		
J	JumpAddr	PC							X		
JAL	JumpAddr	PC			0x1f	PC+4			X		
BEQ	BranchAddr	PC	rs	rt			R1	R2	X		
BNE	BranchAddr	PC	rs	rt			R1	R2	X		
ADDI	PC + 4	PC	rs		rt	Alu	R1	Imm	5		
ANDI	PC + 4	PC	rs		rt	Alu	R1	Imm	7		
ADDIU	PC + 4	PC	rs		rt	Alu	R1	Imm	5		
SLTI	PC + 4	PC	rs		rt	Alu	R1	Imm	11		
ORI	PC + 4	PC	rs		rt	Alu	R1	Imm	8		
LW	PC + 4	PC	rs	rt	rd	Alu	R1	R2	5	R[2:11]	
SW	PC + 4	PC	rs	rt	rd	Alu	R1	R2	5	R[2:11]	R2
XORI	PC + 4	PC	rs		rd	Alu	R1	Imm	9		
SRAV	PC + 4	PC	rs	rt	rt	Alu	R1	R2	1		
SH	PC + 4	PC	rs	rt	rd	Alu	R1	R2	5	R[2:11]	Reg[rt]
BGTZ	BrachAddr	PC	rs	rt	rd	Alu	R1	R2	11		

在完成指令系统数据通路表的填写之后，根据列出的数据通路表，进行多指令数据通路的合并输入数，表，将各个主要功能部件进行连接，根据数据通路合并表的最终结果，对于所有的多输入部件使用多路选择器进行输入选择。最终便可以完成数据通路的搭建。数据通路表如图 3.3 所示：

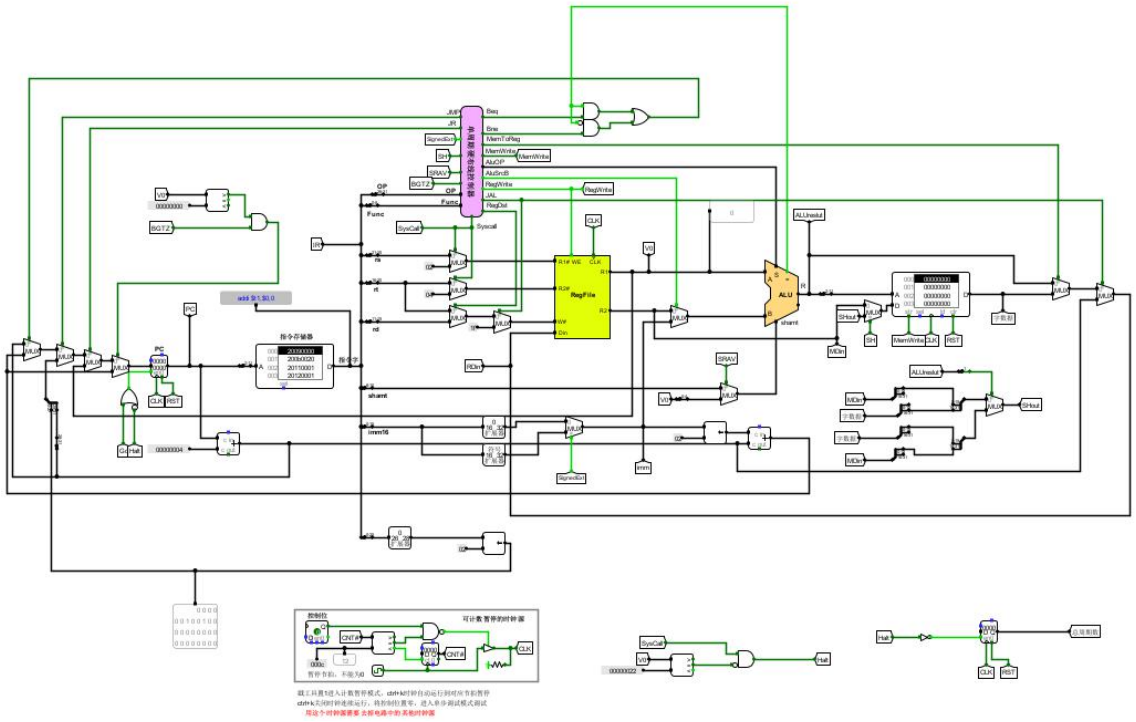

图 3.3 单周期 CPU 数据通路（Logism）

3.1.3 控制器的实现

根据总体方案设计中控制器的设计那一小节的相关内容，分别在 Logism 和 Vivado 上进行主控制器、Branch 控制器、SYSCALL 控制器的具体实现。

主控制器

对照图 3.4 所示。控制器的控制信号都在图中列出，这些信号的在对应不同的指令时的取值不同，上图中的参照各个信号的作用，在这里不对所有的指令进行详解。

指令	OpCode (十进制)	FUNCT (十进制)	ALU_OP	MemtoReg	MemWrite	ALU_SRC	RegWrite	SYSCALL	SignedExt	RegDst	BEQ	BNE	JR	JMP	JAL	RsUsed	RtUsed	ERET	SRAV	SH	BGTZ
SLL	0	0	0				1			1							1				
SRA	0	3	1				1			1							1				
SRL	0	2	2				1			1							1				
ADD	0	32	5				1			1						1	1				
ADDU	0	33	5				1			1						1	1				
SUB	0	34	6				1			1						1	1				
AND	0	36	7				1			1						1	1				
OR	0	37	8				1			1						1	1				
NOR	0	39	10				1			1						1	1				
SLT	0	42	11				1			1						1	1				
SLTU	0	43	12				1			1						1	1				
JR	0	8	X										1	1		1					
SYSCALL	0	12	X					1								1	1				
J	2	X	X											1							
JAL	3	X	X				1							1	1						
BEQ	4	X	X						1		1					1	1				
BNE	5	X	X						1			1				1	1				
ADDI	8	X	5			1	1		1							1					
ANDI	12	X	7			1	1									1					
ADDIU	9	X	5			1	1		1							1					
SLTI	10	X	11			1	1		1							1					
ORI	13	X	8			1	1									1					
LW	35	X	5	1		1	1									1	1				
SW	43	X	5		1	1										1	1				
SRAV	0	7	1				1			1						1	1		1		
XORI	6	X	9			1	1									1					
SH	41	X	5		1	1										1	1			1	
BGTZ	7	X	X						1					1		1	1				1

图 3.4 主控制器控制信号

上图的信号中，最后添加的 SRAV,SH,BGTZ 是针对个人差异化指令而生成的识别辅助信号，因此只有对应的三条指令才有 1 的取值。

3.2 中断机制实现

3.2.1 单级中断

在前文中已经给出了中断逻辑的基本方法，分为硬件和软件两个部分来组合实现，对照图 3.5 给出的单级中断硬件逻辑电路图：

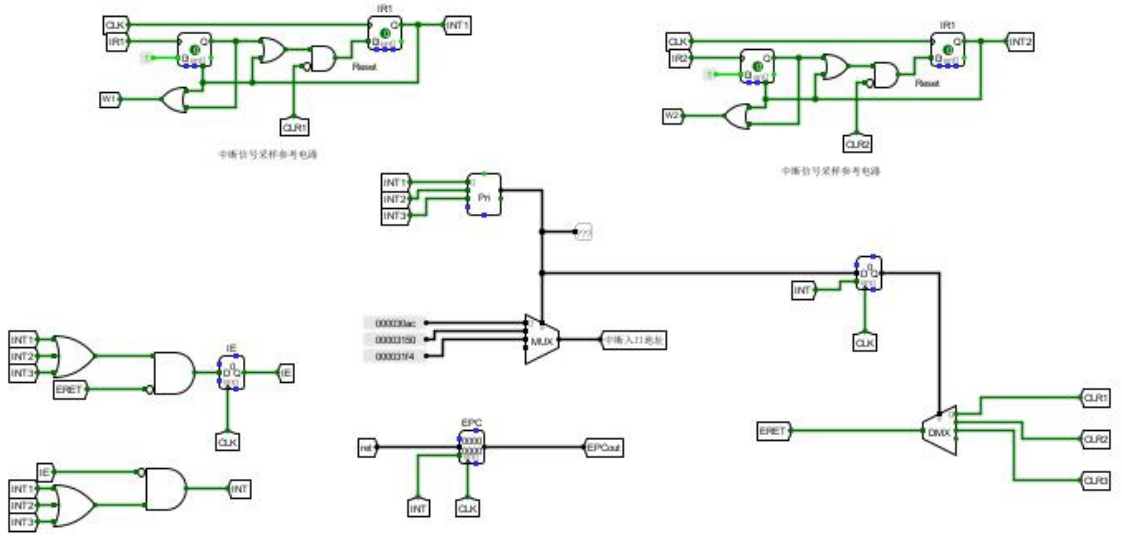

图 3.5 单级中断硬件逻辑

这里给出了中断信号采样电路，提供了中断信号清零功能。通过输入的三个中

断信号 INT1,INT2,INT3 相或并与 ERET 信号的非相与得到 IE，放入 IE 寄存器中，当 IE 信号变为 0 且存在 INT1 这三个中断信号时，表示程序正处在中断程序执行阶段，为了确定三种不同中断的中断子程序入口地址，我首先使用了一个优先编码器对三个中断信号进行了编号，然后通过多路选择器选择出各自的中断程序入口地址，从而进入中断子程序执行阶段，直到程序执行到了 ERET，导致 IE 变为 1，INT 信号重新被置为 0，中断程序退出。注意到这里必须使用一个 IE 寄存器来实现中断隐周期。恢复现场是通过 EPC 寄存器在进入中断时生成的 INT 信号存储 PC 寄存器的值，再等 INT 信号重新变为 0 时把之前存储的 PC 值重新加载到 PC 寄存器中，完成现场恢复。具体恢复逻辑如图 3.6 所示：

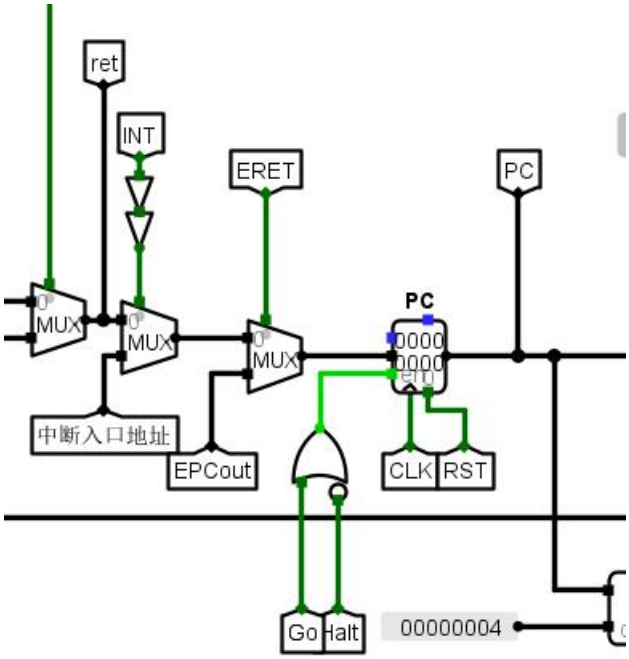

图 3.6 单级中断硬件逻辑

3.2.2 多级中断

多级中断中允许低优先级的中断被高优先级中断打断，所以在打断某个中断之后需要通过开关中断指令，即 MTC0,MFC0 两条数据传输指令，这里我们直接找到了 MTC 和 MFC 的指令 OP 进行对比得出一条指令是否为这两种指令。MFC0 从 CP0 中取出数据放入寄存器中，而 EPC 内存堆栈保护需要将 EPC 的值取出保存到内存中。再用 SW 写入到内存中。

多级中断的实现电路总的如下图 3.7：

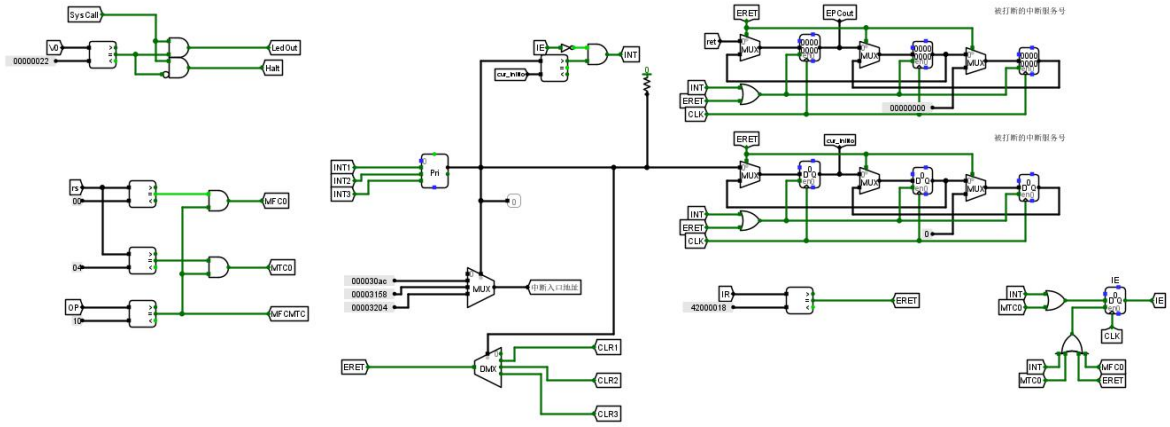

图 3.7 多级中断处理电路

在判断优先级后，对被中断打断的现场 PC 值进入保护堆栈，这里使用三个状态寄存器来存储中断号，当中断号被进入的高优先级中断打断时，中断号就进入第二个寄存器中，同时第二个寄存器的中断号存入第三个寄存器中，是一个硬件实现的堆栈。当一个中断号执行结束后，则取出其之后的寄存器中的中断号，依次向栈顶移动，即出栈操作。

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

对于每一个指令流水接口，需要实现的功能就只有同步清零和寄存器存储。这里给出了一个信号在某个指令流水接口部件中的逻辑电路，如图 3.8：

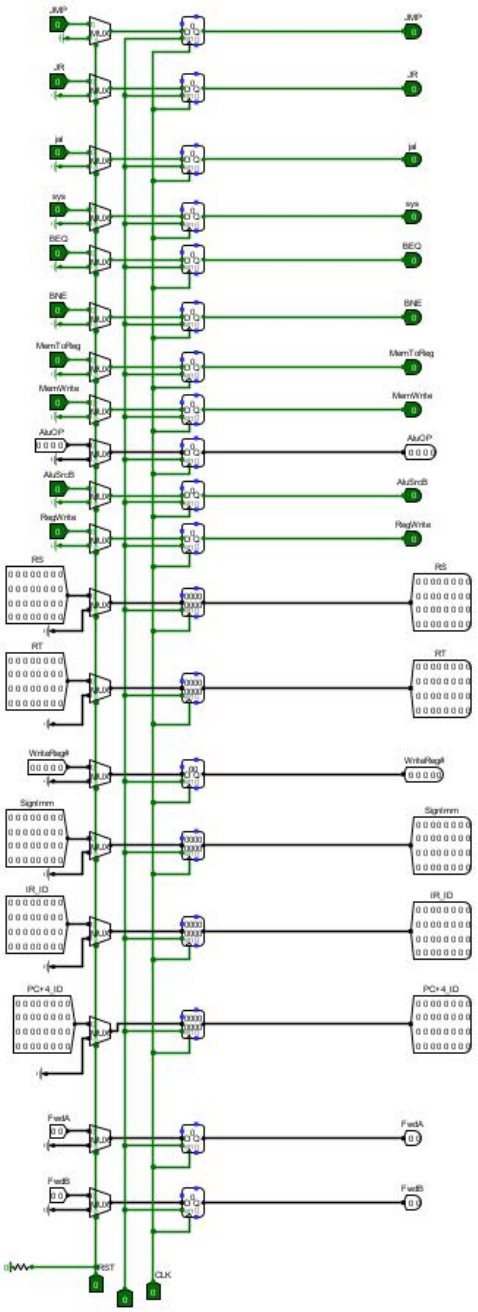

图 3.8 多级中断处理电路

3.3.2 理想流水线实现

理想流水线分为：IF,ID,EX,MEM,WB 这五个步骤，下面分别针对这五个阶段进行设计思路介绍：

1. IF 阶段

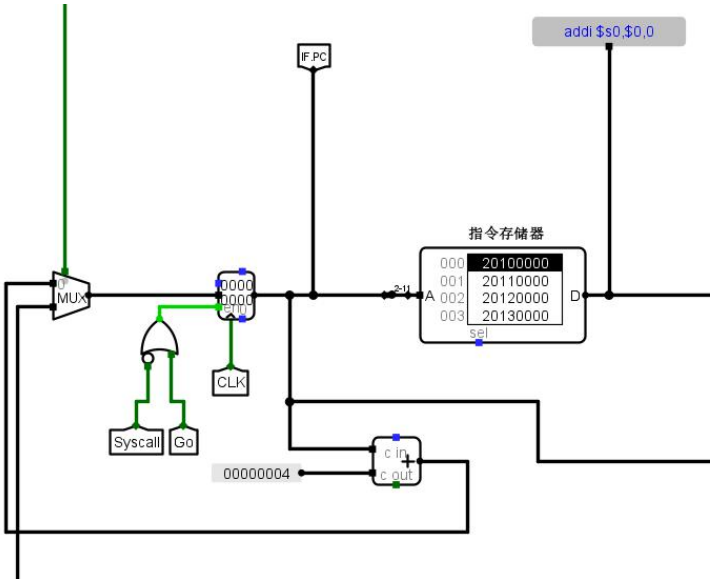

图 3.9 IF 阶段电路逻辑

取指令阶段，只需要根据 PC 寄存器中的值访问指令存储器中的指令并读出来交给 IF/ID 流水接口部件即可。PC 需要进行加四的处理，从而保证在下一个时钟周期里取出的是下一条指令。

2. ID 阶段

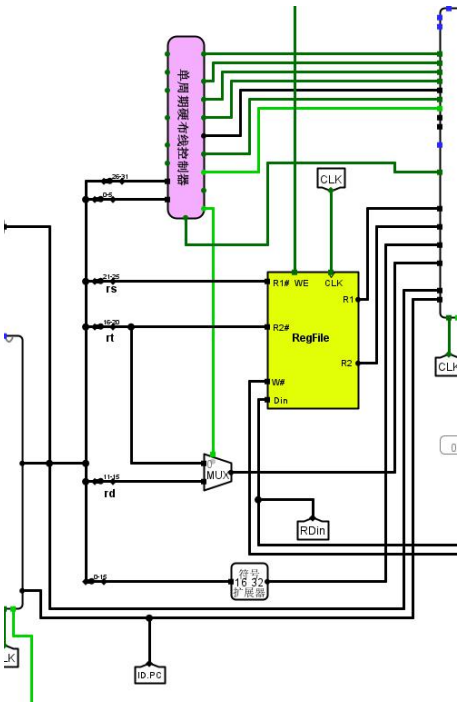

图 3.10 ID 阶段电路逻辑

ID 阶段，通过一个时钟周期，从 IF/ID 指令流水接口中读取出指令，并通过分线器分别提取出 OP,FUNC,RS,RT,RD,IMM 等数据。可以通过 OP 和 FUNC 字段输

入进单周期硬布线控制器得出信号输出。RS 和 RT 作为输入送给 REGFILE 部件中读取出操作数，IMM 通过扩展传入下一个流水部件中。

3. EX 阶段

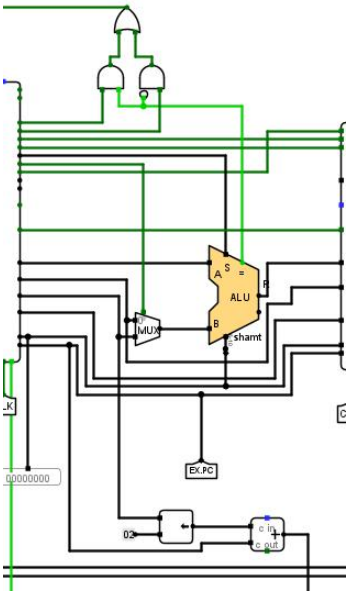

图 3.11 EX 阶段电路逻辑

EX 阶段，通过 ID 阶段得到的操作数送入运算器中进行逻辑运算，同时通过对 J 型指令以及 B 型跳转指令进行逻辑运算后得出 branch 信号，把该信号传回 IF 阶段的多路选择器作为选择信号。

4. MEM 阶段

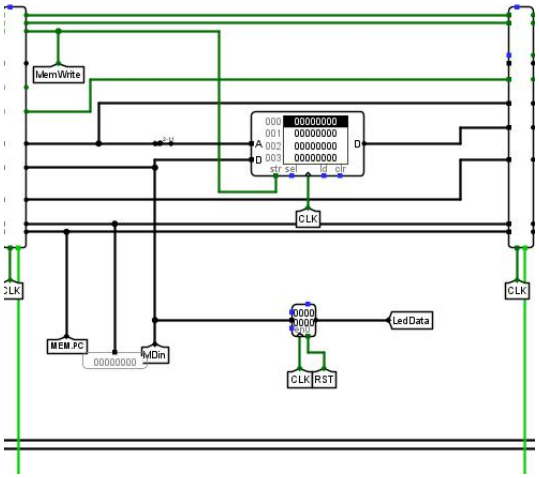

图 3.12 MEM 阶段电路逻辑

MEM 阶段，通过流水接口传递过来的信号和访存地址在存储器中进行写入或者读取操作，把数据传入下一个流水部件接口。这里由于需要进行 LED 显示，因此还需要把 MDin 的数据通过寄存器缓存起来在下一个时钟周期输出。

5. WB 阶段

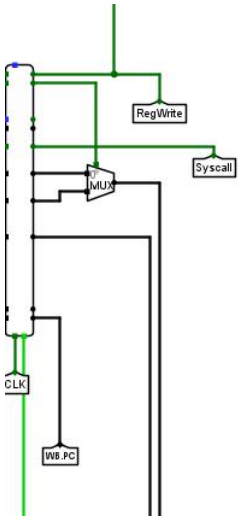

图 3.13 WB 阶段电路逻辑

WB 阶段，需要我们把需要写回的数据传入 ID 阶段配合写使能信号写入寄存器堆中，完成写回操作。

3.4 气泡式流水线实现

气泡流水线相比于理想流水线增加了冲突检测功能，之前已经给出了逻辑表达式，接下来直接给出电路图：

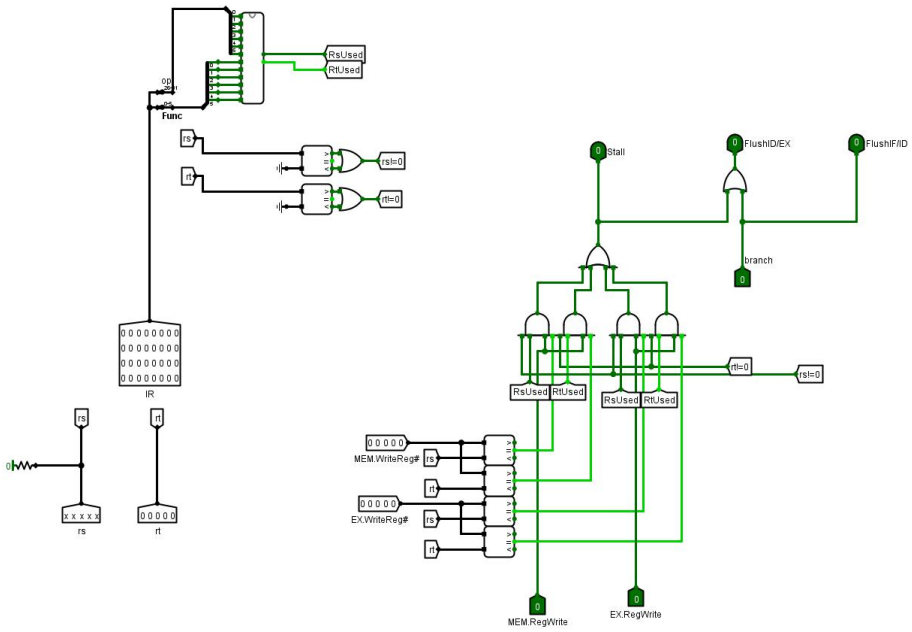

图 3.14 气泡流水线控制逻辑

MIPS 指令包括 0~2 个源操作数，分别是 rs、rt 字段对应的寄存器，其中 0 号

寄存器恒零，不需要考虑相关性。要想确认 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。

有了数据相关检测逻辑，只需考虑如何暂停 IF、ID 段指令的执行以及如何插入气泡的问题，插入气泡可以参考控制相关中的流水清空信号 Flush，当发生数据相关时给 ID/EX 流水寄存器一个同步清空信号 Flush，而要暂停 IF、ID 段指令执行，只须保证程序计数器 PC 的和 IF/ID 流水寄存器的值不变，要做到这一点，只需要控制寄存器使能端即可，当使能端为 1 时，寄存器正常工作，为 0 时则忽略时钟输入，寄存器值保持不变。只需要将数据相关检测逻辑生成的数据相关信号 DataHazzard 作为暂停信号 Stall 取反后送对应的使能端即可。

3.5 数据转发流水线实现

重定向流水线对应指令进入 EX 段可能和前两条指令也就是 MEM、WB 段均存在数据相关，如存在数据相关，EX 段的寄存器操作数 RS、RT 就是错误数据，正确数据应来自于 MEM、WB 段指令的目的操作数，而这些指令已经通过了 EX 段完成了运算，除 Load 类访存指令外，目的操作数都已实际存放在 EX/MEM、MEM/WB 流水寄存器中，可以直接将正确的操作数从其所在位置重定向到 EX 段合适的位置，可以将 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，当然 RT 寄存器也可以采用这种方式处理，重定向方式无需插入气泡，可以解决大部分的数据相关问题，避免插入气泡引起的流水线性能下降，大大优化流水线性能。在控制器部分，与气泡流水线只有清零信号的一点不同。

设计的重定向控制数据通路设计如下：

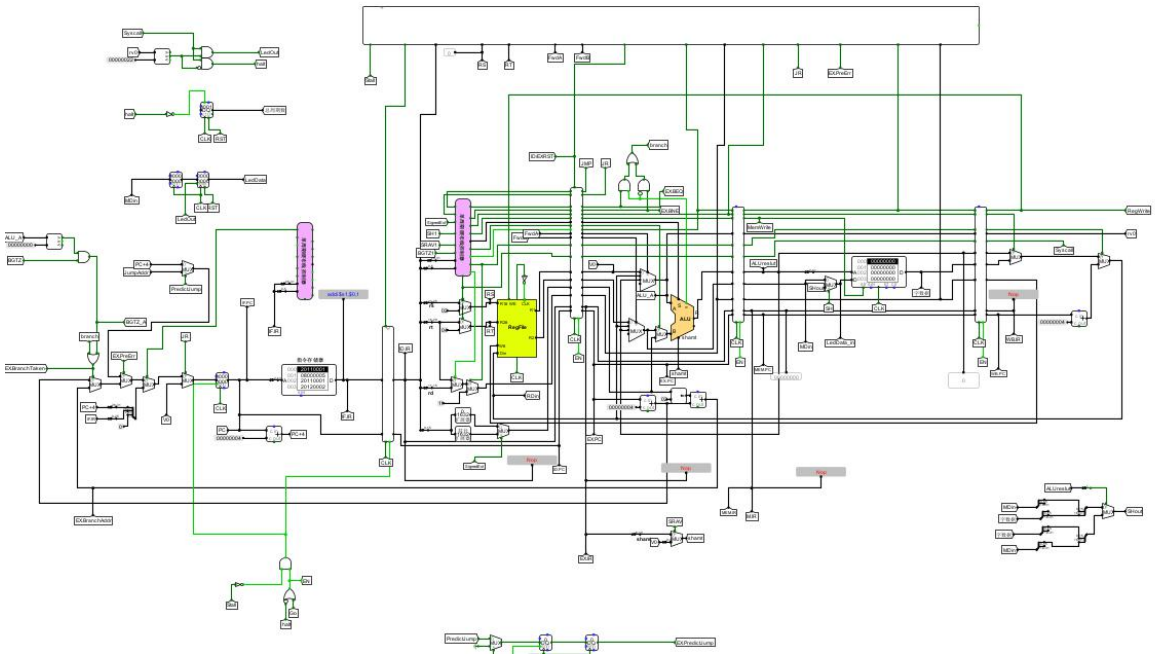

图 3.15 重定向流水线数据通路

3.6 动态分支预测机制实现

动态分支预测是通过在 IF 段利用 PC 寄存器中的值和 BTB 表中进行比较之后一旦发生数据命中，就会根据 BTB 表中的分支预测历史数据的值跳转到预测信息位，同时 BTB 还会提供对应的 PC 寄存器跳转值。EX 段只有在检测到分支预测错误的时候才会清空流水线中的误取的指令。最终相比于气泡流水线和重定向，动态分支预测通过对历史跳转指令的预测来达到直接跳转的目的，减少了气泡插入的数量，提高了流水线效率。

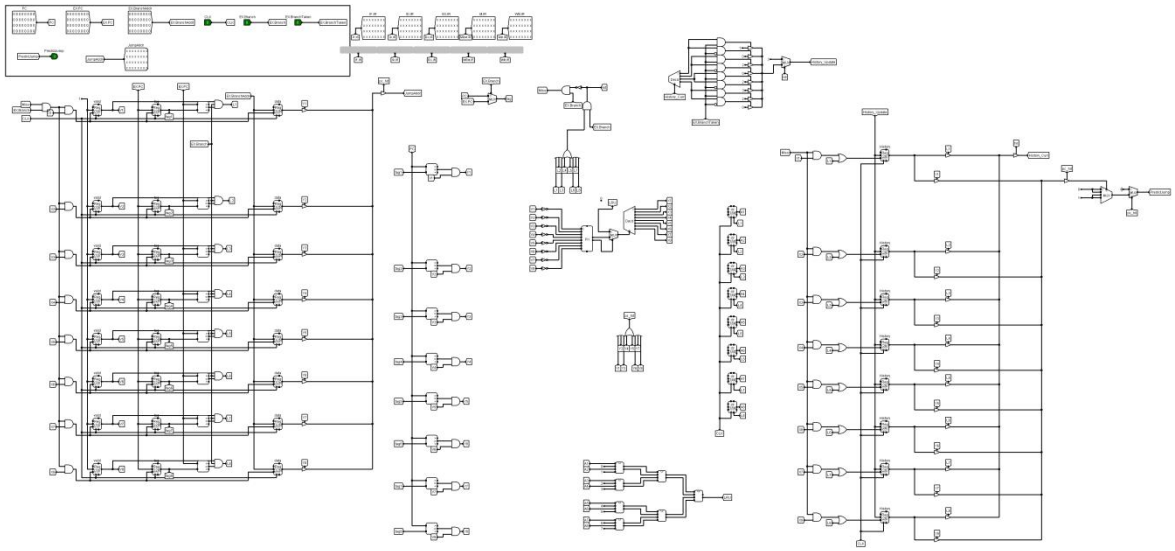

图 3.16 BTB 模块逻辑实现

在 BTB 模块设计中我直接使用了上学期的 cache 存储器的电路设计方案，选区的替换算法是 LRU，相比于 FIFO，两种算法在我的程序中的优化结果是一样的，都是 1750 个周期。最后直接把 BTB 模块加在重定向电路的 IF 段内，修改后的数据通路如下：

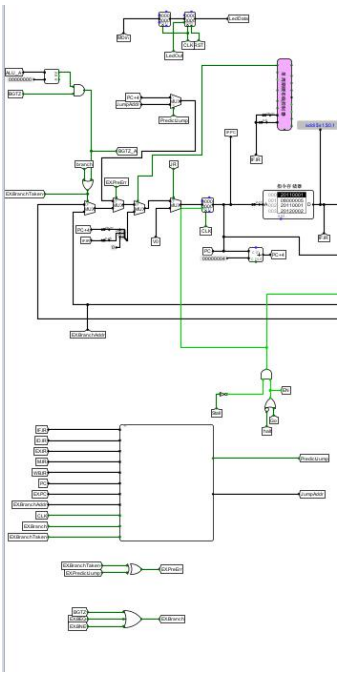

图 3.17 动态分支预测数据通路

4 实验过程与调试

4.1 测试用例和功能测试

测试用例的六个电路主要通过 Benchmark 测试电路，在数码管上显示跑马灯等效果。在这里不再赘述，通过视频演示具体的测试结果即流程。通过对 LED 显示的内容以及周期数，以及流水线的周期时空图对比来测试电路的正确性。

4.1.1 测试用例 1

对于气泡流水线，运行 Benchmark 测试程序的总周期数为 3623，最终的结果显示为 3C，证明电路正确。

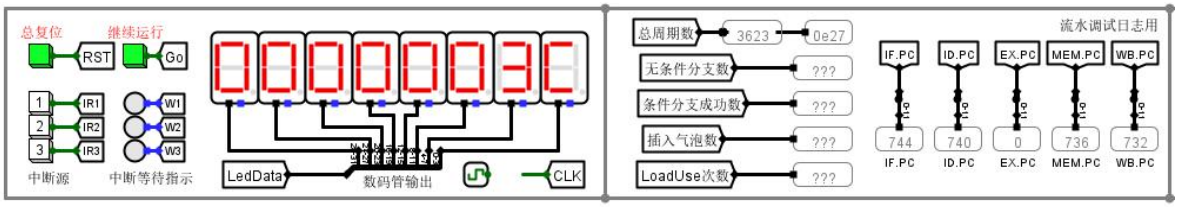

图 4.1 气泡流水线测试结果

对于重定向流水线，我们在其中加入了动态分支预测模块，最终运行 benchmark 程序的总周期数为 1750 相较于之前的 2293 个周期有了很大的提升。

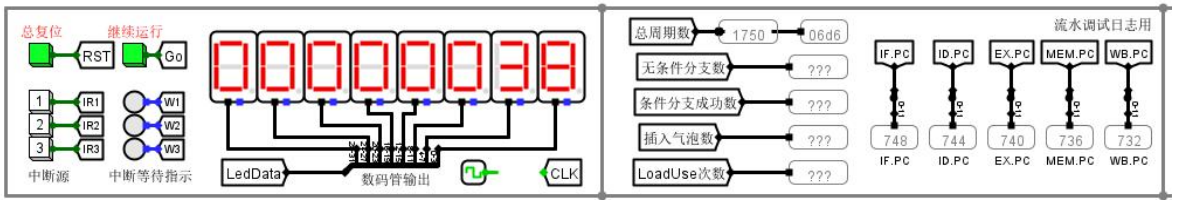

图 4.2 气泡流水线测试结果

接下来测试四条差异化指令：

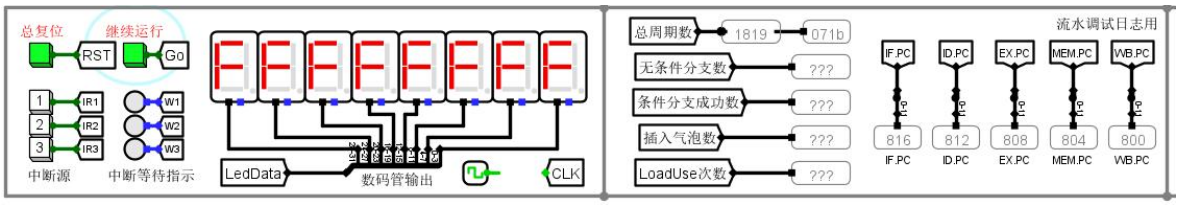

图 4.3 SRAV 指令测试结果

显示器会显示 876 右移一位最后变成全为 F，证明指令成功实现。

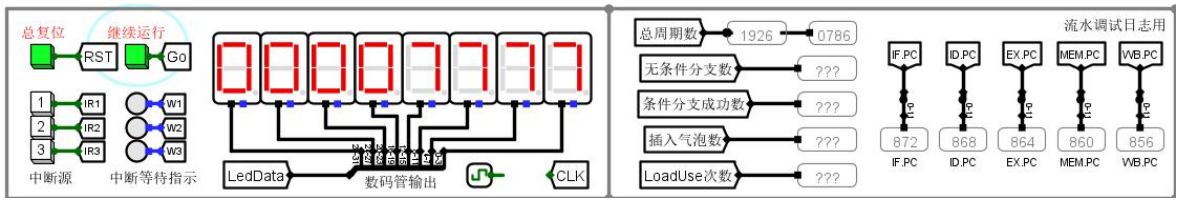

图 4.4 XORI 指令测试结果

显示器会循环显示 7777，8888 重复显示 10 次后停止，证明指令成功实现。

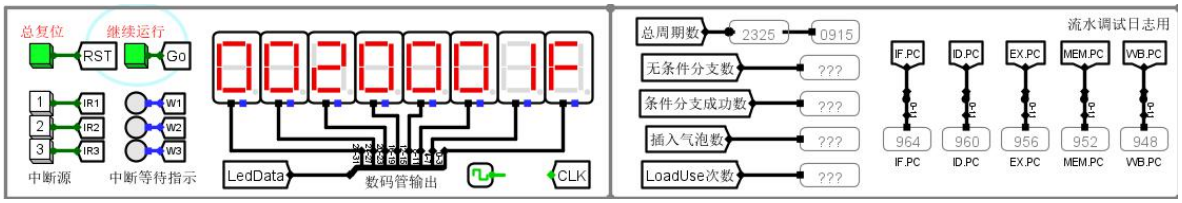

图 4.5 SH 指令测试结果

显示器首先是低四位逐次增加 4，直到达到 20 后会把低四位的数据放到高四位中，最后显示结果为 0020001F，证明指令成功实现。

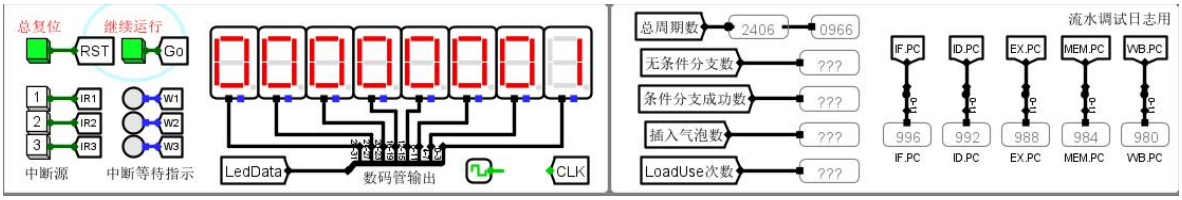

图 4.6 BGTZ 指令测试结果

显示器会从 9 一直每次减 1 直到最后结果为 1，指令成功实现。

4.2 性能分析

通过对不同数据通路执行跑马灯程序的执行结果进行分析我们可以得到气泡流水线需要的时钟周期最多，再进行重定向优化之后，气泡插入的数量大幅减少，继续优化重定向流水线技术，添加 BTB 元器件后，插入的气泡数减少了 543 个，证明 benchmark 程序中有很多重复的跳转指令，通过 BTB 预测直接进行了跳转，提升了 CPU 运行效率。

4.3 主要故障与调试

4.3.1 数据相关故障

理想流水线： syscall 指令不能正常停止。

故障现象：跳转到 syscall 指令发现进入了死循环，数据寄存器里面的值全为 0。

原因分析：单步调试后发现在 syscall 指令执行之后，时钟信号并没有停止计时，从而推断出时钟挂起功能模块出现了问题。后来发现是自己在生成了 halt 信号之后并没有正确连接时钟控制信号让其停下来。之后仍然不行，单步调试后发现不是指导书中写的和 10 比较而是和 0x22 比较。

解决方案：改变比较器中的数据，成功完成挂起，显示数据。

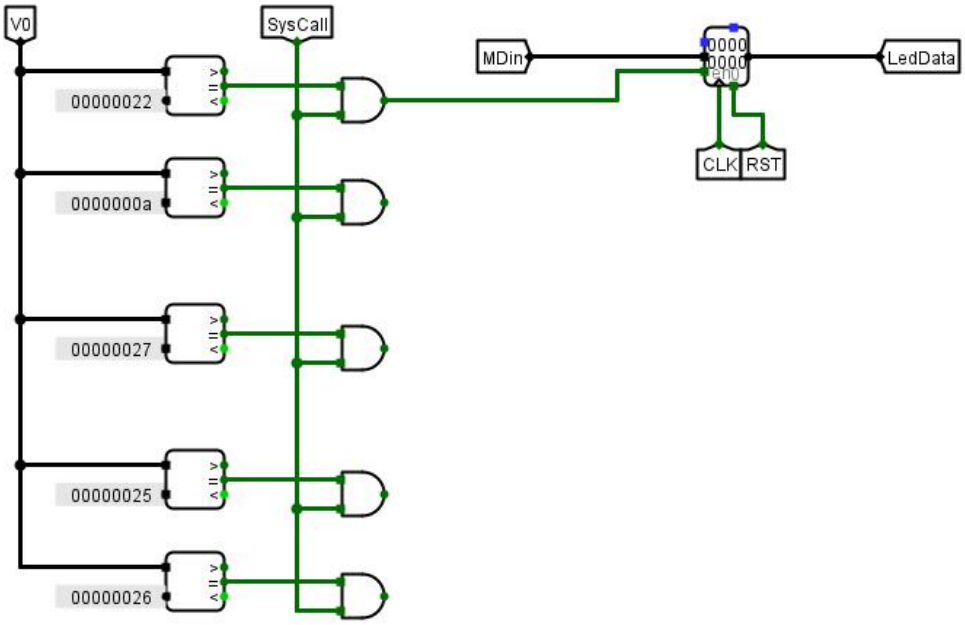

图 4.7 修改比较器的值

4.3.2 Educoder 测试故障

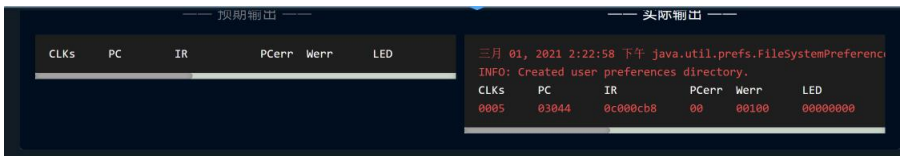

图 4.8 修改比较器的值

单周期 CPU 测试中第五条指令出错，发现自己在设计 jal 指令的时候没有考虑到把数据存回寄存器中，调试过后加上了一个多路选择器成功通关。

图 4.9 修改比较器的值

单周期中断控制电路设计的时候十分顺利，但是在测试的时候 PC 寄存器的值不匹配，发现是需要我们在 INT 信号生成的当前周期就把 PC 寄存器更新，但是我设计的电路需要延迟一个周期才能把数据送入 PC 寄存器中。

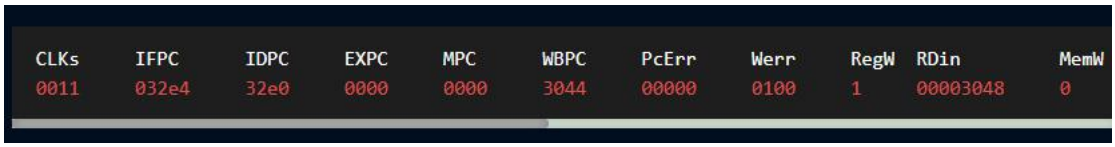

图 4.10 修改比较器的值

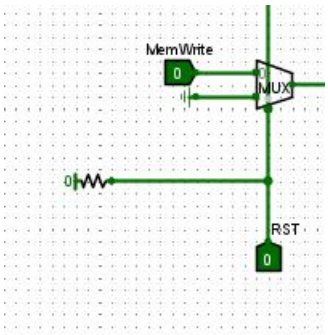

图 4.11 修改比较器的值

在气泡流水线中，最开始我使用的是直接通过清零信号连接到每一个寄存器的清零端实现同步清零，结果发现会产生毛刺，并不能够实现同步清零的功能。最后通过上学期的组原实验经验我选择使用把清零信号与多路选择器的选择段相连，成功完成了同步清零的要求。

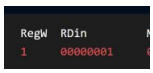

图 4.12 修改比较器的值

重定向电路逻辑的第 22 个周期的 RDin 出现错误，原本这个周期不应该发生指令迁移，但是我却发生了，导致 RDin 为 0，和预期的 1 不符。此外还有第 344 周期报错，最后修改了 writedata 的传输段，错误被成功解决。

4.4 实验进度

表 4.1 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，阅读 MIPS 指令手册，并列出 CPU 各部件的数据通路表，并完成数据通路的基本构建。
第二天	完成单周期 CPU 的控制信号表，使用 Logisim 搭建控制器，实现了单周期 CPU 并且通过了测试。完成部分 Logism 单周期 CPU 故障报告。
第三天	完成 Logism 单周期 CPU 的故障报告，并且通过了 Logism 单周期 CPU 的检查。
第四天	完成 Logisim 单周期中断功能，在 educoder 平台上成功提交。
第五天	完成 Logisim 单周期多级中断功能，在 educoder 平台上成功提交。
第六天	完成 Logisim 理想流水线逻辑设计，在 educoder 平台上提交。
第七天	完成 Logisim 气泡流水线逻辑设计，在 educoder 平台上提交。
第八天	完成 Logisim 重定向流水线设计，在 educoder 平台上提交成功。
第九天	完成冒险处理情形下的动态分支预测，成功运行 benchmark 程序总周期数在 1800 以内。
第十天	完成差异化指令的开发，把动态分支预测和差异化指令组合在重定向流水线中，在 benchmark 中添加差异化指令的测试程序，测试成功通过。

5 团队任务

5.1 团队介绍

我们团队一共有四个同学组成，张鹏同学负责电路实现和汇编代码编写，我负责封装设计以及 ppt 制作，刘金和汪涛负责文档编写。

5.2 团队项目设计与实现

5.2.1 项目内容与特色

我们团队项目是实现一个井字棋游戏，能够交替下棋和判断胜负。项目的特色有：一、综合使用了之前实验用到的汉字显示胜负；二、除了按钮不依赖于特定的硬件，所有的逻辑都通过代码完成，而不是使用电路完成，这增加了整套电路的通用性，加强了 CPU 在其中的作用。

5.2.2 可行性分析

下棋的操作可以使用按钮进行实现，棋局的实现可以使用 CS3410 库中的 LED 进行显示，单周期 CPU 实现的 24 条指令也能够支持完成下棋、绘制、判断胜负等逻辑。

5.2.3 项目设计与实现

CPU 封装：为了整洁和方便调试，将单周期 CPU 作如下封装：

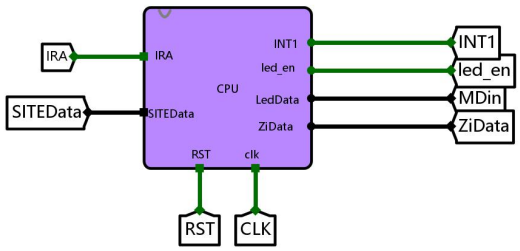

图 5.1 CPU 封装

各引脚定义如表 5.1：

表 5.1 CPU 引脚定义

引脚	I/O 类型	位宽	功能描述	引脚	I/O 类型	位宽	功能描述
IRA	输入	1	中断请求	**RST**	输入	1	复位信号
SITEData	输入	32	按钮位置信	**INT1**	输出	1	是否在中断

引脚	I/O 类型	位宽	功能描述	引脚	I/O 类型	位宽	功能描述
			息				处理过程中
led_en	输出	1	LED 使能	**CLK**	输入	1	时钟信号
ZiData	输出	32	汉字显示数据	**MDin**	输出	32	LED 显示数据

下棋的设计与实现：

1. 设计：使用 9 个按钮代表九宫格上的九个位置，在按下一个按钮之后，在相应的位置绘制图案，绘制过程中，按下其他按钮忽略其他按钮（硬件中断实现），同时要保证图案交替绘制，不能在同一个位置重复绘制图案（软硬件配合实现）。

2. 实现：**硬件上**，如图 5.1 所示，每个按钮作为第一个 D 触发器的时钟端，INT1（表示当前是否正在处理中断）的非作为输入端，输出作为第二个触发器的输入，当时钟到来时，第二个触发器输出 1，同时将第一个触发器清零，INT1 的值为 1，所以其他按钮将会被屏蔽，9 个按钮通过编码器编码，编码作为多路选择器的选择端，选择要写入 0 号内存的信息，每个按钮按下后将在 0 号内存中写入按钮值*4。

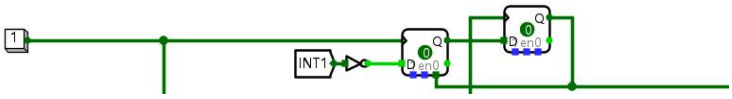

图 5.2 按钮设计

软件上：将按下的按钮保存为数组，读取 0 号寄存器中的值，识别当前是哪个按钮被按下，根据数组中的信息判断该按钮是否在之前被按下，如果是则退出，不予响应，如果是新按钮，则保存然后响应这次按钮按下的程序。

判断胜负逻辑设计与实现：

1. 设计：为两个玩家使用两个寄存器保存状态信息，如果某一个位置被按下，则在该玩家对应寄存器相应二进制位置 1，而井字棋一共有 8 种情况可以赢，只需判断玩家对应寄存器中为 1 的位置是否符合这 8 种情况。

2. 实现：从左到右依次编号为 1 到 9，使用 t8 和 t9 寄存器保存玩家信息，寄存器第 0 位表示是否在位置 1 下棋，依次类推，在每次下棋之后判断该玩家是否获胜，比如判断玩家 1 是否以在 1,2,3 位置下棋的方式获胜，则使用以下两条指令：

```
andi $s0, $t8,7        # 玩家 1 信息保存在$8 中   1 2 3 位置获胜对应低 3 位为 1 即十进制的 7
```

```
beq $s0,7,success1   # 如果以这种方式获胜则转到胜利处理部分
```

显示实现：使用 syscall 指令来调用显示，34 号表示 LCD 显示，35 号表示字符点阵显示，需要显示的数据放在 a0 寄存中，分别从通过 ZiData 和 MDin 引出，具体电路连接如图 5.3 所示：

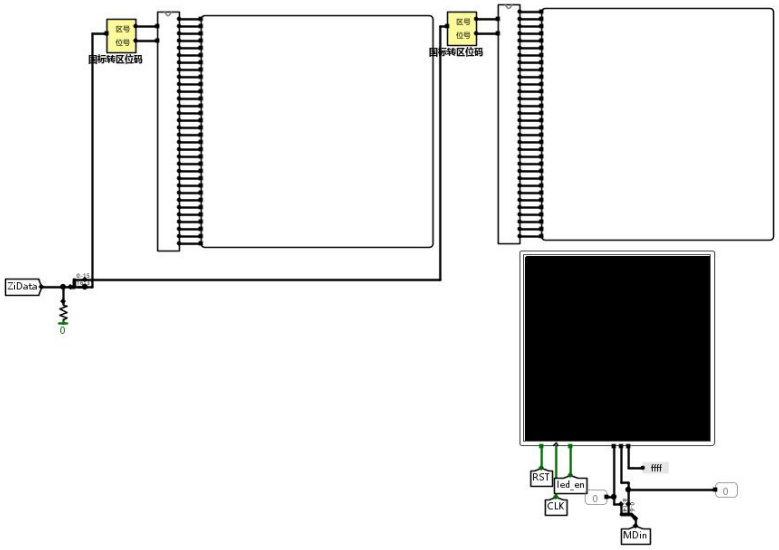

图 5.3 显示电路连接

5.3 团队项目测试

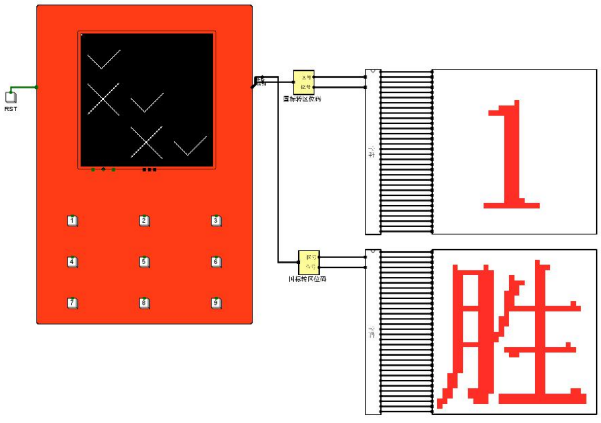

图 5.4 玩家一胜利测试

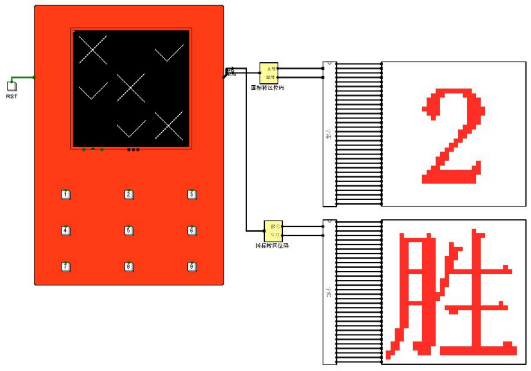

图 5.5 玩家二胜利测试

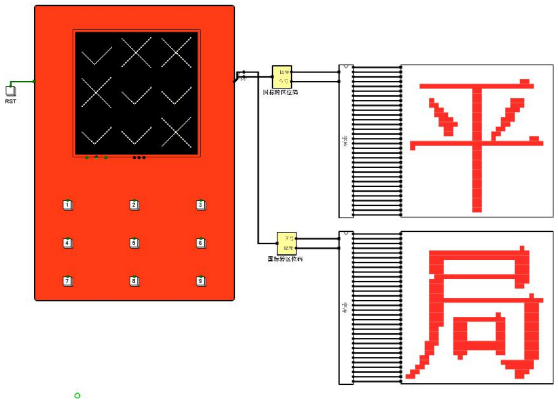

图 5.6 平局测试

5.4 我完成的工作

我负责的部分是设计电路封装并实现，并且在张鹏同学开发过程中一直交流设计思路，最后也是由我来完成 PPT。主要的汇编逻辑部分自己上手写了一个 √ 的显示，其余的都由张鹏同学完成。

6 设计总结与心得

6.1 课设总结

从最开始的单周期 CPU 数据通路开始搭建，到实现单级和多级中断。最后转换到理想流水线，再依次使用气泡流水线，重定向流水线，动态分支预测技术对流水线电路进行改进，最后成功的做出了一整套性能高效的流水线系统。

1） 单周期 CPU 逻辑简单，一个周期完成一条指令的执行，在中断设计时对现场的保护比较注重细节。流水线部分从插入气泡时就需要参考教材上的控制逻辑的表达式，在重定向设计的时侯加上动态分支预测也是参考了教材上的电路逻辑实现的。

2） 最终我实现了单周期 CPU，单级和多级中断功能，理想流水线设计，气泡流水线设计，重定向流水线设计，动态分支预测，差异化指令实现。

6.2 课设心得

本次计算机组成原理课程设计是我花费时间最多的一门课程设计。但是我自己由于之前在设计阶段给自己埋了太多的雷导致我在最后的调试过程中花费了太多的时间，这也反映出了我的很多问题，但是也锻炼了我的耐心和调试能力。

课程设计的第一个实验是 Logisim 设计单周期 CPU，在上学期的组原理论课上已经学的很透彻了，在设计好数据通路后很快的就完成了。单级中断的功能也是和上学期理论课重合很大，也是迅速的完成了。第一次是卡在了多级中断，相比于单级中断，这里需要我检测 MFC 和 MTC 指令，并且还要我添加一个类似于中断寄存器堆栈，最后我是使用了四个寄存器才实现了能够处理三个中断的堆栈。

接下来的流水线 CPU 设计才是我这次课设的重点和难点。理想流水线设计在设计流水接口部件中同步清零的问题困扰了我很久，此外，为了在 educoder 平台上通过测试，我发现我的隧道标签的位置放的不正确也会导致测试无法通过。一些比较小的细节问题比如无符号扩展和符号扩展我之前就没有考虑到，此外由于之前在填信号生成表的时候我对于一些指令的信号产生分析错误导致我在后面的调试中单步跟踪了很久才找出了问题，希望老师以后能够在 excel 表中也添加和上学期一

样的正确检测功能，帮助我们更好的发现自己的问题。 最后就是实现动态分支预测功能了，整体上使用的 cache 存储器借鉴了上学期设计的电路，自己再添加了 hit 和 miss 判断逻辑，最后成功的把运行周期降到了 1750，达到了要求。

然而对于本次课程设计，我还有一些小小的建议和改进。本次课程设计过程时间较长，文档比较杂乱，希望能专门针对实验给出一个指导任务书，能够系统的列出要完成的实验以及实现该功能的资料出处。对于讨论方式，每天在组原群里的消息太多了，我直接屏蔽了消息，导致我在面对一些普遍性的问题的时候不能很好的从群聊天记录中提取出有用的信息，希望能够专门建立一个常见报错表格，能够让我快速找到和我自己错误类似的解决方案。

最后在这里也感谢几位老师对于我在本次课程设计中无数问题的耐心解答，也感谢本组所有成员在课程设计中对于我的帮助和建议。我相信组成原理课程设计必将成为我整个大学生涯中一段无比难忘的回忆。

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4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] 秦磊华，吴非，莫正坤.计算机组成原理. 北京：清华大学出版社，2011年.

[4] 谭志虎，秦磊华，胡迪青.计算机组成原理实践教程.北京：清华大学出版社，2018.

[5] 袁春风编著. 计算机组成与系统结构. 北京：清华大学出版社，2011年.

[6] 张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008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：杨雨鑫