华中科技大学

2019

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：	CSIE1701
学　　号：	U201710080
姓　　名：	周易
电　　话：	15623608870
邮　　件：	U201710080@hust.edu.cn

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

(1) 支持表 1.1 前 27 条基本 32 位 MIPS 指令；

(2) 支持教师指定的 4 条扩展指令；

(3) 支持多级嵌套中断，利用中断触发扩展指令集测试程序；

(4) 支持 5 段流水机制，可处理数据冒险，结构冒险，分支冒险；

(5) 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。

(6) 能运行教师提供的标准测试程序，并自动统计执行周期数

(7) 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	
13	NOR	或非	

#	指令助记符	简单功能描述	备注
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	算术可变右移	
29	XOR	异或	
30	LB	加载字节	
31	BLTZ	小于 0 转移	

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硬布线控制，控制信号直接由各种类型的逻辑门电路组合生成。所有的电路在 Logisim 软件中进行搭建。

总体结构图如图 2.1 所示。

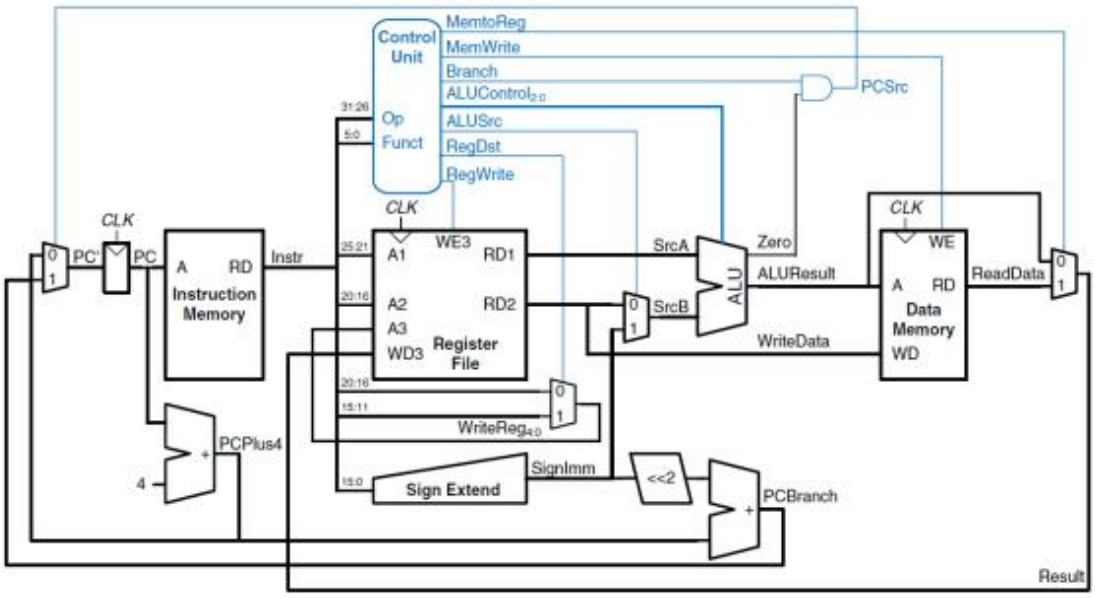

图 2.1 总体结构图

2.1.1 主要功能部件

1. 程序计数器 PC

该部件用来存储当前执行指令的地址。每经过一个时钟周期，该部件存储的指令地址都会发生改变。

2. 指令存储器 IM

该部件存储程序的所有指令，它与 PC 相连。根据 PC 存储的指令地址，指令存储器 IM 会输出地址对应的指令编码（二进制机器码）。

3. 运算器

运算器的输入端有两个源操作数，控制器与运算器相连。运算器将根据控制器的控制信号决定进行哪种运算，然后将运算结果输出。运算器的各种引脚与功能描述如表 2.1 所示。

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
shamt	输入	5	移位操作数
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
Less	输出	1	Less = (x < y)? 1 : 0，对所有操作有效
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 RF 整合了所有 32 个寄存器，它可以接受两个源寄存器的编号，然后输出它们存储的数据；也可以接受一个目的寄存器的编号，当写使能为高电平时，一旦接受到时钟脉冲，就把要写的数据写入目的寄存器。

2.1.2 数据通路的设计

不同的指令经过的数据通路并不完全相同，对于不同指令，按照指令的功能，将各部件连接起来。最后，将所有指令对应的连接路径结合起来，就形成了数据通路。一条指令采用数据通路的哪一条路径由控制单元进行控制。不同指令的详细路径参见§3.1.2。

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，统计得到的控制信号以及说明如表 2.2 所示。

表 2.2 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
R2	0	寄存器堆 R2 口读取 rt 字段指示寄存器的值
	1	寄存器堆 R2 口读取 4 号寄存器的值
W#	00	寄存器堆 W#口读取 rt 字段指示寄存器的编号
	10	寄存器堆 W#口读取 rd 字段指示寄存器的编号
	1	寄存器堆 W#口接收 0x1f
PC	000[1]	PC 输入端读取 PC+4
	100	PC 输入端读取 JMP 指令的跳转地址
	10	PC 输入端读取寄存器 R1 的值
	1	PC 输入端读取分支地址
B	0	运算器 B 口读取 R2 口的值
	01	运算器 B 口读取无符号扩展的立即数
	11	运算器 B 口读取有符号扩展的立即数
Din	00	寄存器堆 Din 口读取运算器的运算结果
	010	寄存器堆 Din 口读取存储器输出的数据
	X110	寄存器堆 Din 口读取经符号扩展的存储器输出的低 8X ~ 8X + 7 位数据，X 可以取 0，1，2，3
	1	寄存器堆 Din 口读取 PC + 4 的值
SignedExt	0, 1	是否为有符号扩展
JR	0, 1	当前指令是否为 JR 指令

[1] 多位取值代表输入端信号需要多个多路选择器选择。低位代表 Logisim 电路图中靠近输入端的选择器选择信号，反之代表远离输入端的选择器选择信号。

控制信号	取值	说明
JMP	0, 1	当前指令是否为 JMP 指令
Beq	0, 1	当前指令是否为 Beq 指令
Bne	0, 1	当前指令是否为 Bne 指令
MemToReg	0	寄存器堆 Din 口读取运算器的运算结果
	1	寄存器堆 Din 口读取来自存储器的数据
MemWrite	0, 1	数据存储器的写使能
AluOP	0	Result = X << Y　逻辑左移　(Y 取低五位)　Result2=0
	1	Result = X >>>Y　算术右移　(Y 取低五位)　Result2=0
	2	Result = X >> Y　逻辑右移　(Y 取低五位)　Result2=0
	3	Result = (X * Y)[31:0];　Result2 = (X * Y)[63:32]　无符号乘法
	4	Result = X/Y;　Result2 = X%Y　无符号除法
	5	Result = X + Y　(Set OF/UOF)
	6	Result = X - Y　(Set OF/UOF)
	7	Result = X & Y　按位与
	8	Result = X | Y　按位或
	9	Result = X ⊕ Y　按位异或
	10	Result = ~(X |Y)　按位或非
	11	Result = (X < Y) ? 1 : 0　符号比较
	12	Result = (X < Y) ? 1 : 0　无符号比较

根据统计信息，我们在各种主要部件的输入端口添加多路选择器形成控制电路，以根据控制信号确定各种端口的输入信号。其中，主控制器、Branch 控制器和 SYSCALL 控制器是三个主要的控制模块。主控制器获取指令的 funct 和 op 段编码，从根本上产生一系列控制信号，然后这些控制信号或者是直接控制主要部件端口的输入，或者再去控制其他控制模块产生下一层的控制信号。主控制器模块如图 2.2 主控制器所示。

对照主控制器产生的所有控制信号，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，取值一律为 0，以简化控制器电路的设计。这样就形成了一张关于控制信号与指令的真值表（详见§3.1.3），

根据真值表就可以完成从 funct 和 op 段编码到主控制信号的逻辑电路。

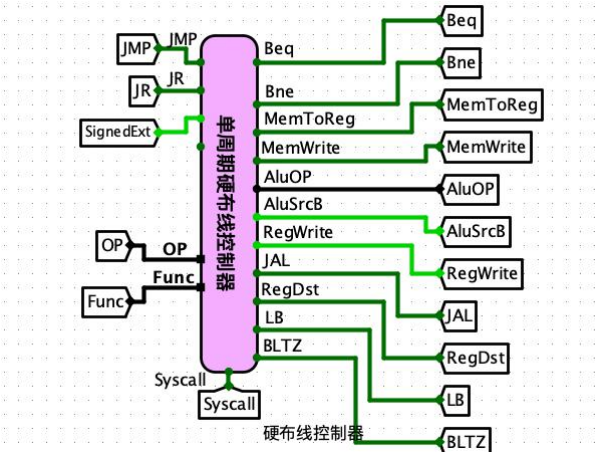

图 2.2 主控制器

2.2 中断机制设计

2.2.1 总体设计

考虑到外部设备与 CPU 处理速度的差异，我们需要设计一个中断机制。该机制让 CPU 在没有外部设备请求的条件下，正常执行现有任务，一旦外部设备发出中断请求，CPU 就暂停当前程序，转而执行中断号对应的子程序。等到子程序执行完毕，再回到原来的程序，继续执行。这样的总体设计需要我们从硬件和软件两个方面进行考虑。

2.2.2 硬件设计

MIPS CPU 接受到外部信号后，按照处理顺序，需要的硬件模块包含以下几部分：

- 中断请求生成逻辑——外部设备向 CPU 发送信号后，需要生成中断请求；
- 中断优先级仲裁——如果接收到不同的外部信号，由 CPU 选择响应哪一个中断；
- 中断响应周期——在进入子程序之前，首先保存原先程序要执行的指令地址，方便以后回到此处，然后跳转到中断子程序入口地址；
- 与中断相关的寄存器——负责保存恢复信息，如保存原先程序要执行的指令地址。

2.2.3 软件设计

MIPS 指令集需要设计具有开中断、关中断、中断返回功能的指令，用于支持中断功能。在中断子程序末尾，加上中断返回指令，意味着子程序执行完毕，返回到原程序；如果执行了关中断，CPU 将屏蔽一切中断请求，直到再次开中断为止。

一个符合常规的中断子程序，为了让 CPU 回到原程序时，所有数据都保持原样，需要在开头加上保护现场的指令，对应地，在结尾增加恢复现场的指令。在程序需要退出时，加上中断返回指令用来退出。

2.3 流水 CPU 设计

2.3.1 总体设计

CPU 的时钟周期受到关键路径的影响，在单周期 CPU 中，一个主要问题是关键路径需要很长的时钟周期。因此，在流水 CPU 中，我们将 CPU 的数据通路分割成 5 段，分别是取指 IF、译码 ID、执行 EX、访存 MEM、写回 WB。

2.3.2 流水接口部件设计

为了使前一阶段的数据能够传到下一阶段，我们必须在段与段之间设置一个流水接口部件，以便保存前一阶段的数据。考虑到寄存器具有保存数据的特性，我们在段与段之间的每条传输路径上都加一个寄存器，这样就完成了流水接口部件的构造。

2.3.3 理想流水线设计

理想流水线不需要我们考虑数据冲突、结构冲突和分支冲突等等问题，因此我们只需要简单的用寄存器连接彼此相邻的两段即可。

2.4 气泡式流水线设计

气泡式流水线通过插入气泡的方式消除指令之间的数据冲突和分支冲突。

分支冲突是指分支指令还没来得及跳转，紧跟其后的若干指令就进入了流水线，导致执行出错。以指令 BEQ 为例，假设紧随其后的是 ADD 指令和 SUB 指令，在本次实验设计的 CPU 中，BEQ 指令需要到 EX 段才能判断是否需要跳转。可是

如果 BEQ 进入 EX 段并且判断分支成功, ADD 与 SUB 指令早已进入 ID 段和 IF 段, 它们被称为误取指令。气泡式流水线的解决方案是在下一个时钟周期到来时，把 ADD 指令与 SUB 指令清空，从而不会影响执行结果（如图 2.3 所示）。

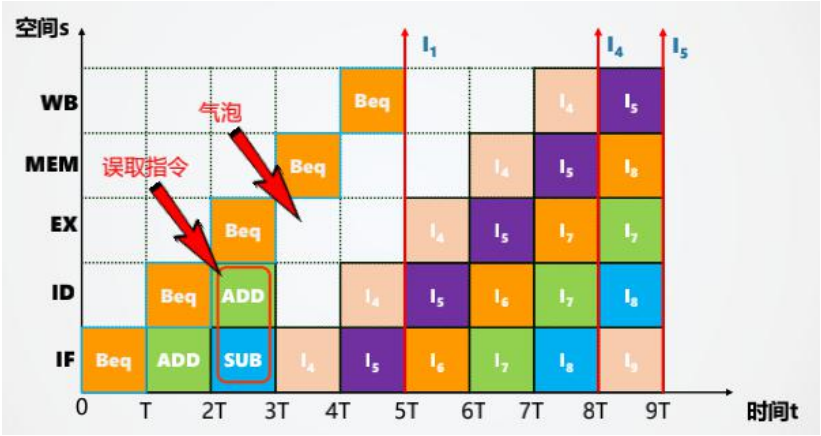

图 2.3 分支冲突的气泡式流水线解决方案

数据冲突是指后面的指令需要读取的寄存器与前面的指令需要写入的寄存器相同，可是前面的指令还没有来得及把新数据写入寄存器。这时，我们需要暂停后面指令，让前面的指令继续执行，直到写入新的数据才继续执行。在两个指令之间插入气泡。

2.5 数据转发流水线设计

对于数据冲突，气泡流水线的解决方案是在数据相关的两条指令之间插入气泡，消除冲突。可是这带来的问题是会增加若干条无用指令。数据转发流水线解决了这个弊端，如果两条指令之间存在数据冲突，数据转发流水线不会让后一条指令等待前一条指令将数据写入寄存器，而是直接将数据送到后一条指令需要的位置，这样既消除了数据冲突，又不会增加额外的时间开销。

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，触发方式为上升沿触发，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，将此控制信号连接到寄存器的使能端，当需要进行停机时，Halt 控制信号为 0，使整个电路停机。如图 3.1 所示。

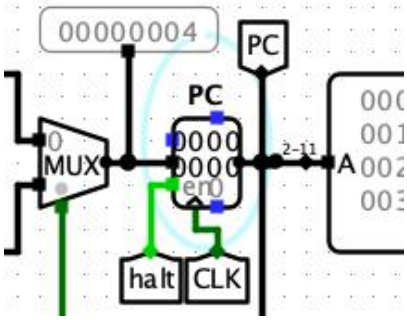

图 3.1 程序计数器（PC）

2. 指令存储器（IM）

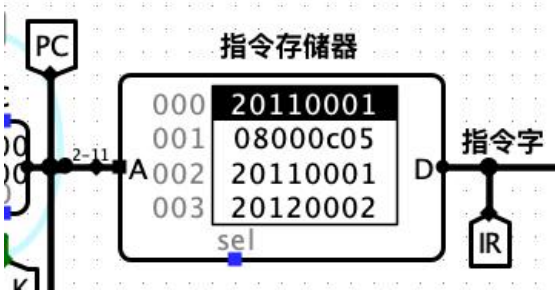

图 3.2 指令存储器（IM）

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。

3. 运算器

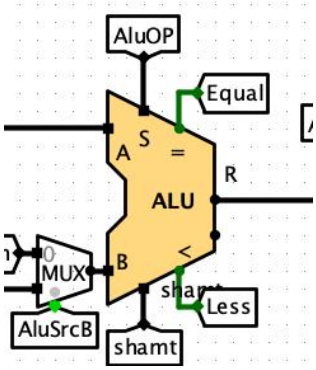

图 3.3 运算器

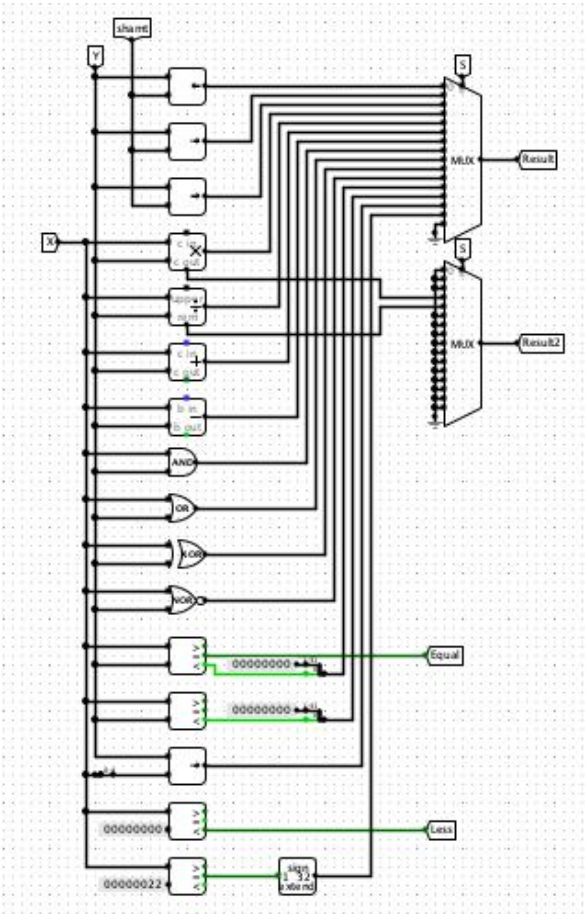

图 3.4 运算器内部结构

运算器支持多种类型的运算，这些运算连接到选择器上，由 AluOP 控制信号选择运算方式，然后输出结果，封装后如图 3.3 所示。

在运算器内部，提供了所有的指令运算可能用到的运算功能。每个运算功能由具有对应功能的逻辑部件实现，比如加法运算用加法器实现，移位运算用移位器实现。具体的内部结构如图 3.4 所示。

4. 寄存器堆 RF

顾名思义，寄存器堆是一堆寄存器的结合。本次实验提供了寄存器堆 RF 的逻辑组件，只需要连接逻辑组件的各个端口即可。寄存器堆 RF 的结构如图 3.5 所示。

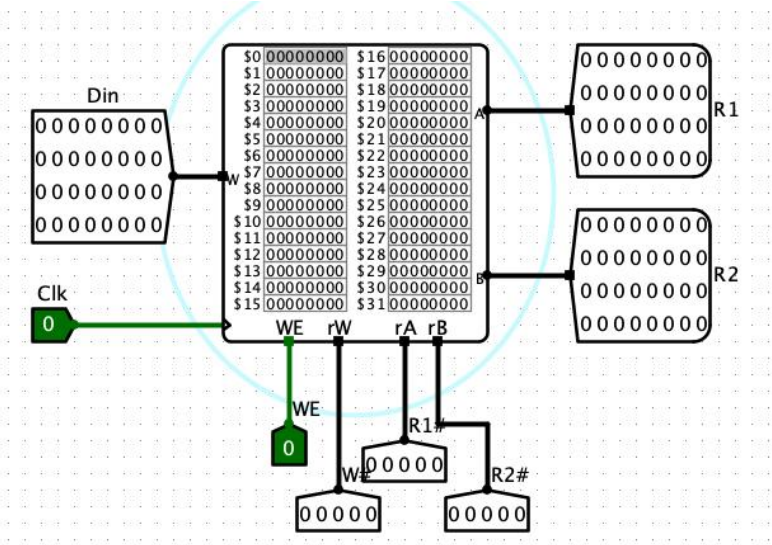

图 3.5 寄存器堆（RF）

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		Tube
			R1#	R2#	W#	Din	A	B	OP	Addr	Din	
ADD	PC+4	PC	rs	rt	rd	alu	r1	r2	5			
ADDI	PC+4	PC	rs		rt	alu	r1	立即数	5			
ADDIU	PC+4	PC	rs		rt	alu	r1	立即数	5			
ADDU	PC+4	PC	rs	rt	rd	alu	r1	r2	5			
AND	PC+4	PC	rs	rt	rd	alu	r1	r2	7			
ANDI	PC+4	PC	rs		rt	alu	r1	立即数	7			
BEQ	PC+4 / PCBranch[1]	PC	rs	rt			r1	r2	X			
BLTZ	PC+4 / PCBranch	PC	rs				r1		X			
BNE	PC+4 / PCBranch	PC	rs	rt			r1	r2	X			
J	JumpAddr	PC										
JAL	JumpAddr	PC			0x1f	PC+4						
JR	r1	PC	rs									
LB	PC+4	PC	rs		rt	MData[2]	r1	立即数	5	Result		
LW	PC+4	PC	rs		rt	MData	r1	立即数	5	Result		
NOR	PC+4	PC	rs	rt	rd	alu	r1	r2	10			
SLL	PC+4	PC		rt	rd	alu	r2	立即数	0			
SLT	PC+4	PC	rs	rt	rd	alu	r1	r2	11			
SLTI	PC+4	PC	rs		rt	alu	r1	立即数	11			
SLTU	PC+4	PC	rs	rt	rd	alu	r1	r2	12			
SRA	PC+4	PC		rt	rd	alu	r2	立即数	1			
SRAV	PC+4	PC	rs	rt	rd	alu	r1	r2	13			
SRL	PC+4	PC		rt	rd	alu	r2	立即数	2			

[1] 将由控制信号决定输入哪个数据，下面以斜线分割的情况均是如此。
[2] 这里的 MData 是从内存中取出某个字节然后扩展成 32 位得到的，电路参见图 3.6。

指令	PC	IM	RF				ALU			DM		Tube
			R1#	R2#	W#	Din	A	B	OP	Addr	Din	
SUB	PC+4	PC	rs	rt	rd	alu	r1	r2	6			
SW	PC+4	PC	rs	rt			r1	立即数	5	result	r2	
SYSCALL	PC+4	PC	0x2	0x4		alu						
OR	PC+4	PC	rs	rt	rd	alu	r1	r2	8			
ORI	PC+4	PC	rs		rt	alu	r1	立即数	8			
XOR	PC+4	PC	rs	rt	rd	alu	r1	r2	9			

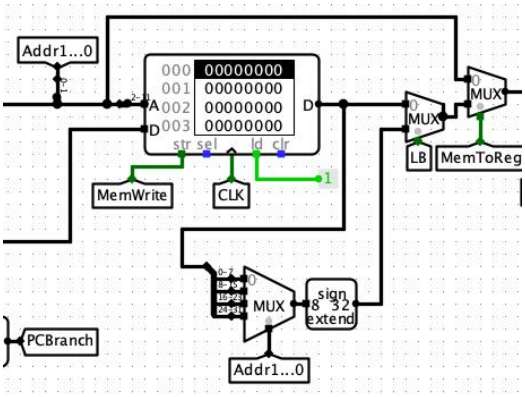

图 3.6 从内存中取出一个字节数据

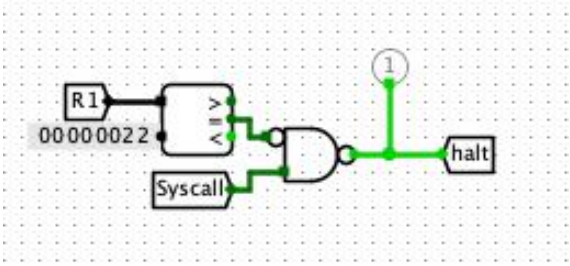

图 3.7 控制停机电路

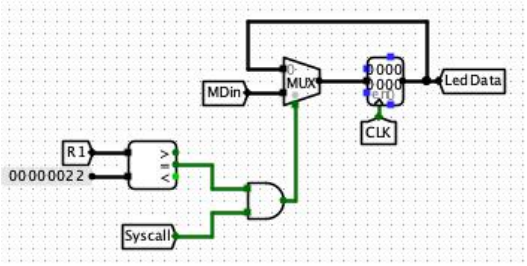

图 3.8 数码管显示电路

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。

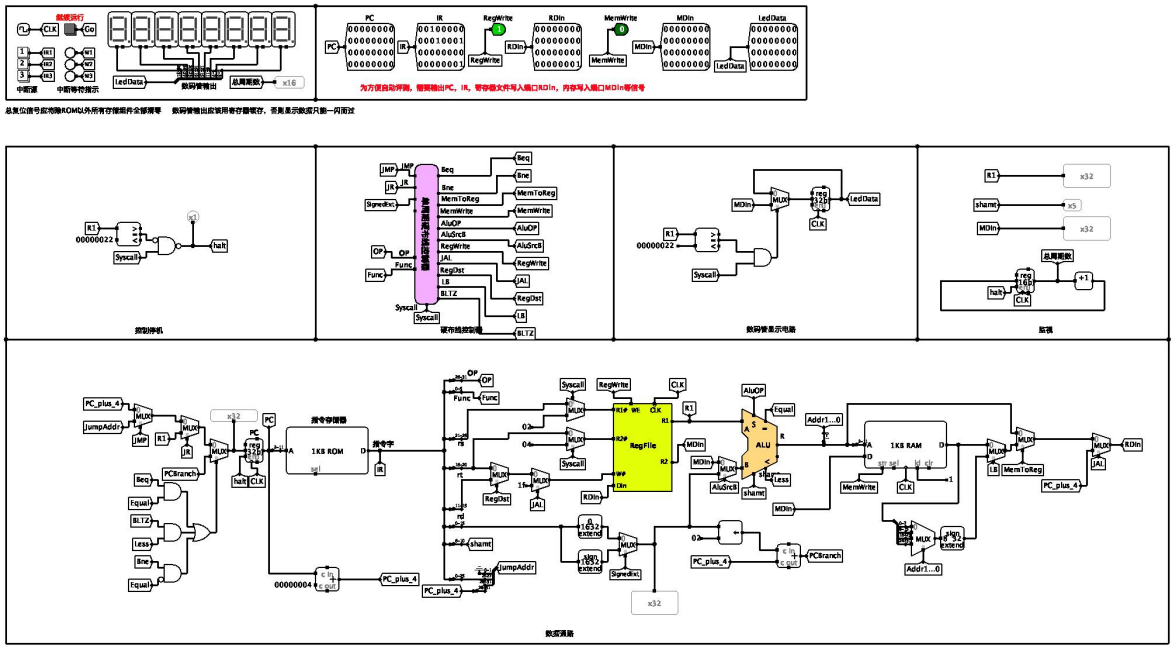

图 3.9 单周期 CPU 数据通路（Logism）

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，在 Logisim 上进行主控制器、Branch 控制器、SYSCALL 控制器的具体实现。

1. 主控制器

表 3.2 给出了以指令 op 和 funct 段为输入，控制信号为输出的真值表。

表 3.2 主控制器控制信号

指令	AluOP	MemToReg	MemWrite
SLL	0		
SRA	1		
SRL	2		
ADDU	5		
ADD	5		
SUB	6		
AND	7		
OR	8		
NOR	10		
SLT	11		
SLTU	12		
JR	X		
SYSCALL	14		
J	X		
JAL	X		
BEQ	X		
BNE	X		
ADDI	5		
ANDI	7		
ADDIU	5		
SLTI	11		
ORI	8		
LW	5	1	
SW	5		1
SRAV	13		
XOR	9		
LB	5	1	
BLTZ	X		
ERET	X		

指令	ALU_SRC	RegWrite	SYSCALL	SignedExt	RegDst	BEQ	BNE	JR	JMP	JAL	LB	BLTZ	rs	rt	eret
SLL		1			1								1	1	
SRA		1			1								1	1	
SRL		1			1								1	1	
ADD		1			1								1	1	
ADDU		1			1								1	1	
SUB		1			1								1	1	
AND		1			1								1	1	
OR		1			1								1	1	
NOR		1			1								1	1	
SLT		1			1								1	1	
SLTU		1			1								1	1	
JR								1					1		
SYSCALL			1												
J									1						
JAL		1							1	1					
BEQ				1		1							1	1	
BNE				1			1						1	1	
ADDI	1	1		1									1		
ANDI	1	1											1		
ADDIU	1	1		1									1		
SLTI	1	1		1									1		
ORI	1	1											1		
LW	1	1		1									1		
SW	1			1									1	1	
SRAV		1			1								1	1	
XOR		1			1								1	1	
LB	1	1		1							1		1		
BLTZ				1								1	1		
ERET															1

根据真值表，可以写出每一个控制信号的逻辑表达式，从而每个信号都有一个逻辑电路。将表中的指令输入进这个逻辑电路，就能得到正确的输出。将这些逻辑电路全部封装在一个芯片中，就形成了主控制器模块。

2. Branch 控制器

Branch 控制器选择下一条程序指令输入到 PC 寄存器中（如图 3.10 所示）。它由多个多路选择器组成，每个多路选择器通过由主控制器生成的信号进行选择。它

的大致原理是：如果某一种分支的情况成功了，就跳转到分支指向的指令；否则，如果指令本身是跳转指令（J, JAL, JR），那么就跳转到它们指向的指令；否则，下一条指令仍然是 PC+4 指向的指令。

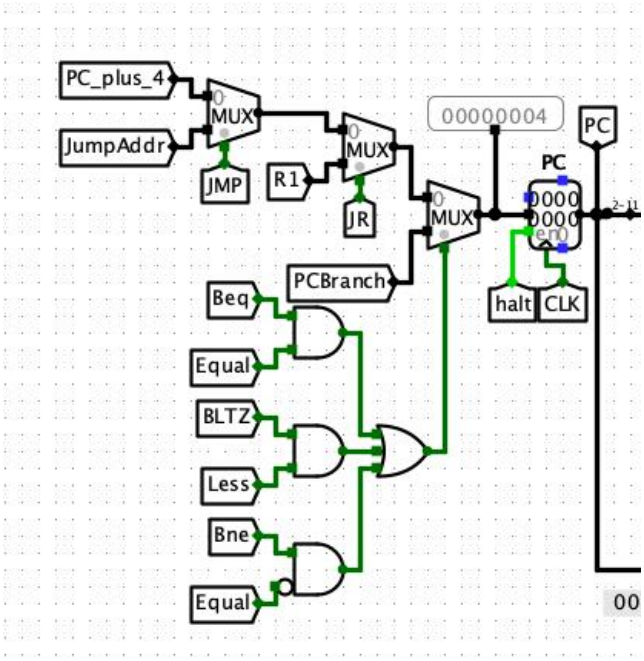

图 3.10 Branch 控制器

3. SYSCALL 控制器

SYSCALL 指令从寄存器堆取出 2 号寄存器和 4 号寄存器的值，2 号寄存器的值送入控制停机电路（结构见图 3.7），4 号寄存器的值送入数码管显示电路（结构见图 3.8）。如果 2 号寄存器的值是 0x22，产生停机信号 halt（低电平有效）；如果 2 号寄存器的值不是 0x22，在数码管上显示 4 号寄存器的值。

3.2 中断机制实现

3.2.1 单周期 MIPS + 单级中断

首先是中断请求生成电路，这部分电路在实验包中已经给出。图 3.11 是该模块的电路结构，图 3.12 是封装图。

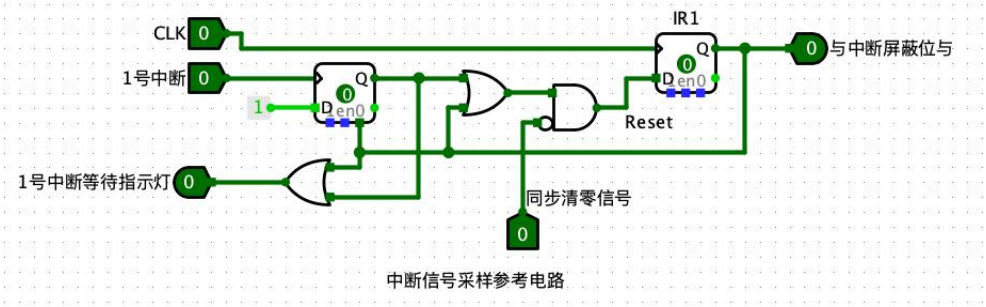

图 3.11 中断请求生成电路

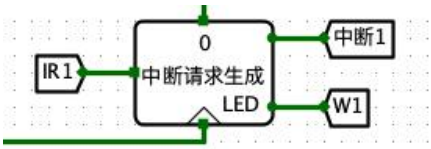

图 3.12 中断请求生成封装图

以 1 号中断为例，外部请求信号（IR1）传入 1 号中断端口后，指示灯（W1）亮起。下一个时钟信号到来后，“与中断屏蔽位与”端口输出高电平信号（中断 1），交给优先级仲裁电路。

优先级仲裁电路使用元件 Priority Encoder，当有中断请求时从封装芯片上方端口输出中断信号（低电平有效），并且输出优先级最高的中断号（如图 3.13 所示）。

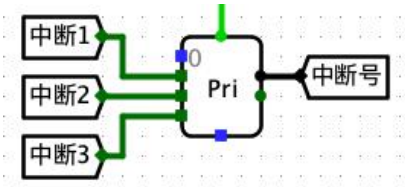

图 3.13 中断电路中的 Priority Encoder 元件

有了中断信号和要处理的中断号后，进入中断响应周期。当且仅当程序处于中断响应周期时，我们才能把原程序将要执行的指令地址压入寄存器 EPC，而进入中断响应周期的标志是中断信号为高电平，因此将中断信号连接到 EPC 的使能端（如图 3.14 所示）。

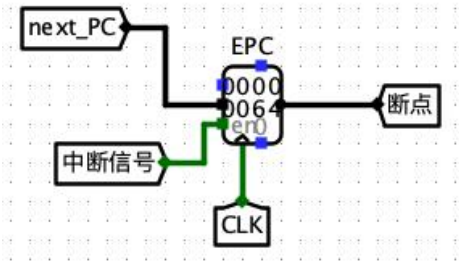

图 3.14 EPC 寄存器

中断响应周期要做的另一件事，是将中断号锁存。这样在中断返回指令到来时（即 ERET 指令），才能确定哪一个中断请求生成电路需要清除中断请求信号（如图 3.15 所示）。

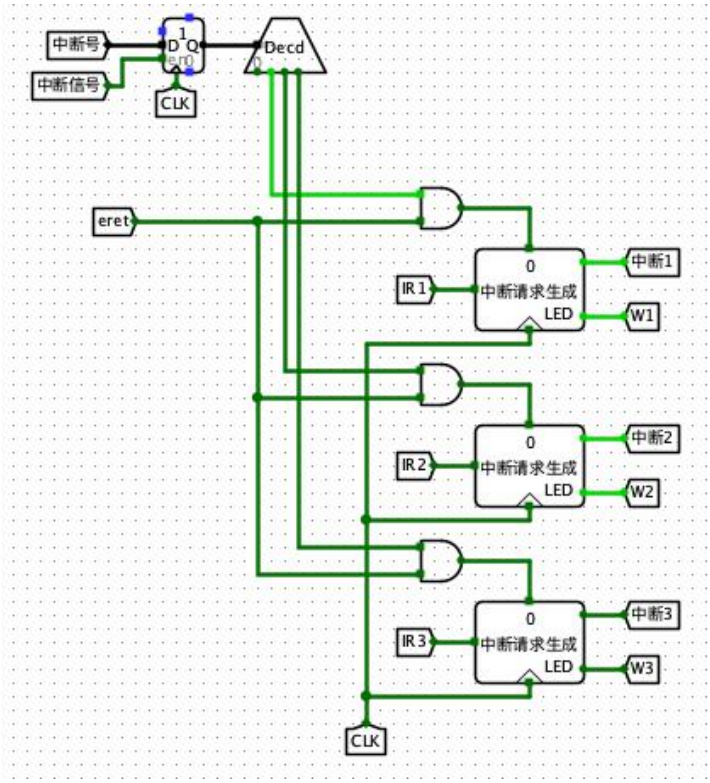

图 3.15 中断子程序退出时，需要清除指定的中断请求信号

中断响应周期结束后，我们需要清除中断信号。回顾图 3.13，我们假设中断信号是由 Priority Encoder 的顶部端口直接生成的，但是中断 1 信号必须在子程序退出时才被清除。试想一下，如果我们在中断响应周期结束时，通过清除中断 1 信号来清除中断信号，那么进入 1 号中断子程序后，它还可能被其他中断打断，这在单级中断里，是不被允许的。因此我们对原先假设的中断信号再做些处理（如图 3.16 所示）。由 IE 寄存器输出的$\overline{Q}$信号初始状态下为 1，这样中断信号还是由 Priority Encoder 的顶部端口决定，中断信号变为高电平后，二路选择器将选择常量 1 作为寄存器的输入。这样在下一个时钟到来后，$\overline{Q}$信号变为 0，中断信号被清除。退出中断子程序后，$\overline{Q}$信号恢复初始状态，中断信号又由 Priority Encoder 的顶部端口决定。这样的电路结构既可以保证中断响应周期结束后清除中断信号，又可以在 CPU 执行中断子程序时屏蔽其他中断。

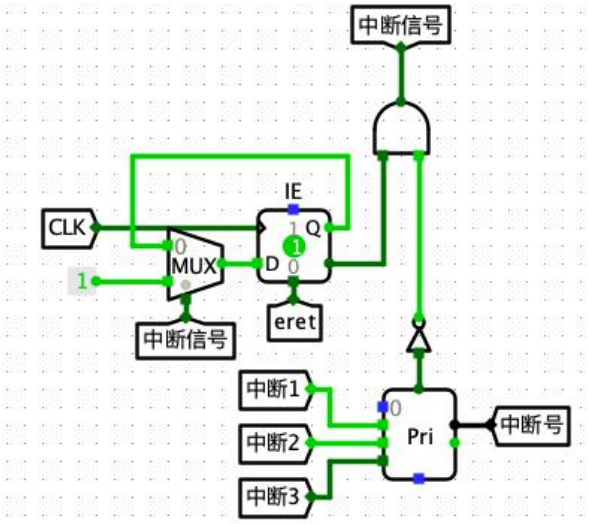

图 3.16 生成与清除中断信号

3.2.2 单周期 MIPS + 多级中断

与单级中断不同，对于多级中断机制来说，进入中断子程序后，如果执行了指令 mfc0（开中断），仍然可以被更高优先级的中断请求打断，直到执行了指令 mtc0（关中断）为止。这里面包含两点变化：1. 中断子程序只能被更高优先级的中断请求打断；2. 支持指令 mfc0 和 mtc0。

为了满足第一点变化，在优先级仲裁电路中，将每个外部中断请求与中断屏蔽字相与之后，才能输入 Priority Encoder 元件（如图 3.17 所示）。1 号中断优先级最低，因此只要有中断请求，1 号屏蔽字就会变成高电平。2 号中断优先级其次，如果有 2 号中断请求或 3 号中断请求，2 号屏蔽字就会变成高电平。而 3 号中断优先级最高，因此只有接收到 3 号中断请求时，3 号屏蔽字才会变成高电平。如果没有接收到可以使屏蔽字变为 1 的中断请求，那么屏蔽字还保持原先的值（如图 3.18 所示）。

为了满足第二点变化，首先我们设计了由指令 op 和 rs 段生成 mfc0 与 mtc0 控制信号的电路。在单级中断机制的实现中，我们已经知道通过 IE 寄存器的输出可以控制开关中断（如图 3.16 所示）。对于多级中断，我们只要让 mfc0 与 mtc0 也可以对 IE 寄存器的输入进行控制，就可以实现指令层次的开关中断（如图 3.17 所示）。

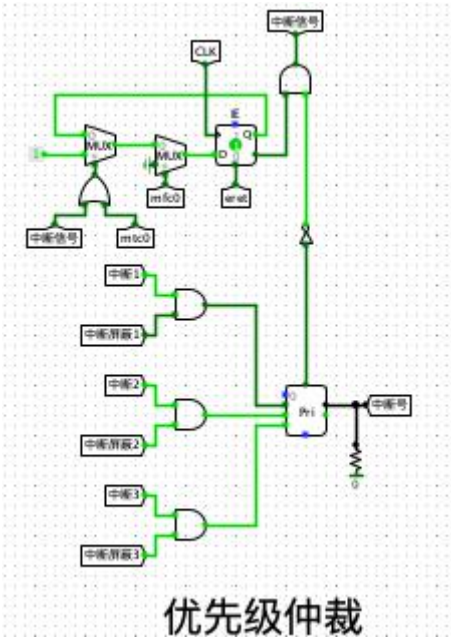

图 3.17 多级中断机制的优先级仲裁电路

最后，我们还要实现一个硬件堆栈，用于存储屏蔽字、中断号和每一层的next_PC。对于本实验的多级中断，中断请求最多可以嵌套三层，因此要使用三个寄存器。每个寄存器的输入端插入一个 2 路选择器，用 eret 信号控制硬件堆栈的压栈与弹出动作。寄存器使能端由“中断信号∨ eret 信号”控制。以屏蔽字堆栈为例，如图 3.18 所示。

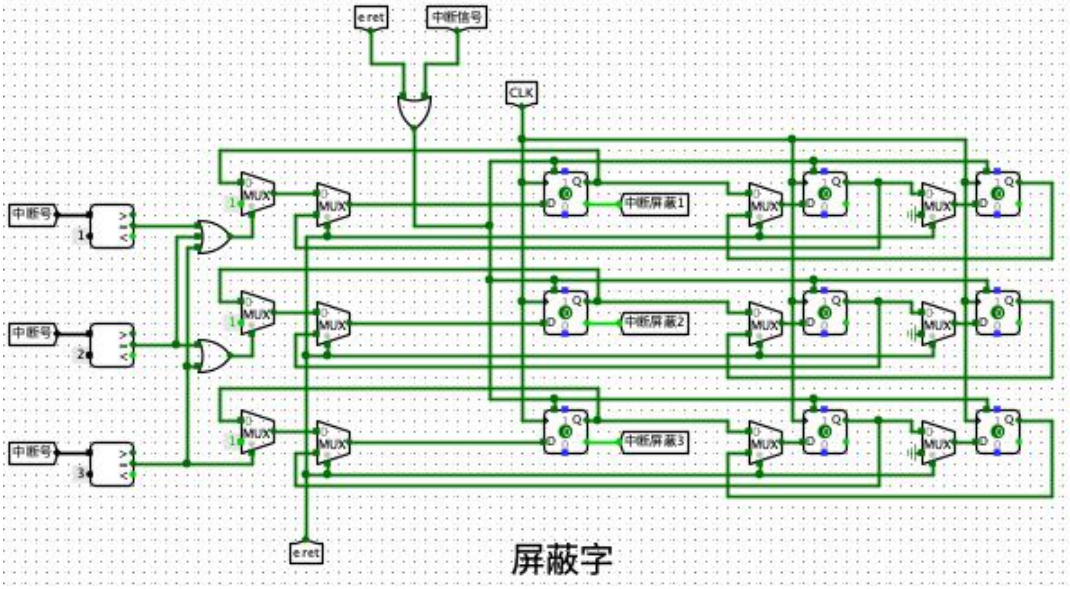

图 3.18 屏蔽字堆栈

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

流水接口部件将数据通路彼此相邻的段连接起来，两个段之间，每有一条路径连接，流水接口部件就增加一个输入和输出段子，以便将该路径上的数据保存起来，供下一个流水段使用。将流水接口部件封装后，大体的效果如图 3.19 所示。至于流水接口部件内部的实现，取决于流水线的种类，将在下面几节详细描述。

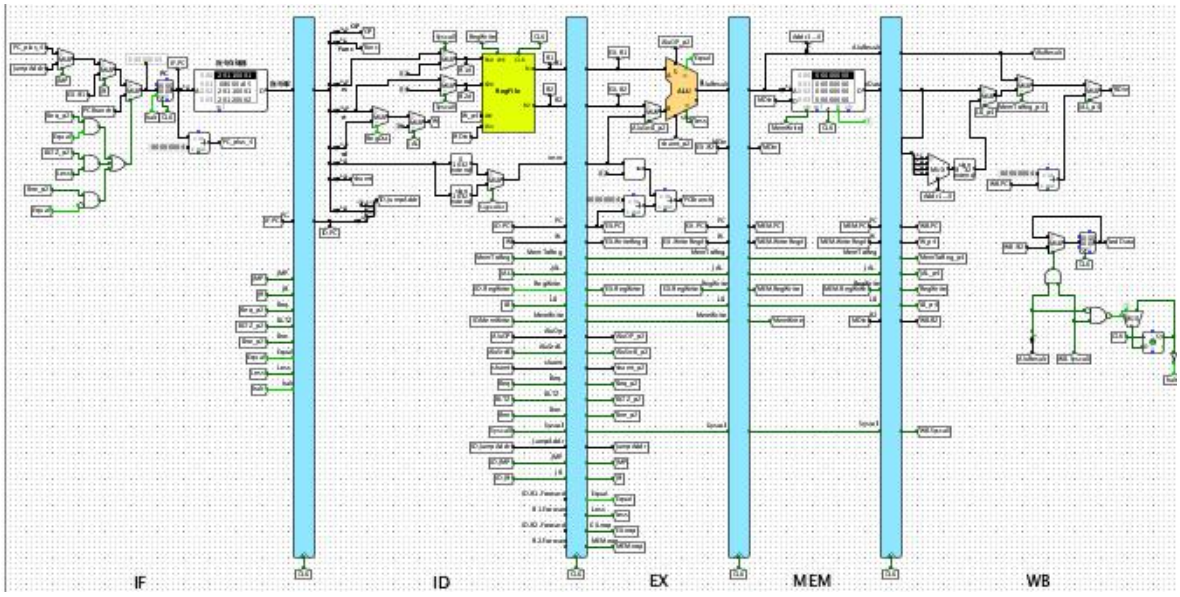

图 3.19 增加了流水接口部件的数据通路

3.3.2 理想流水线实现

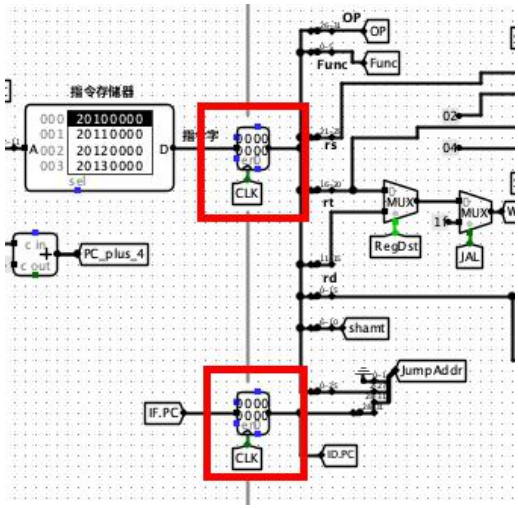

图 3.20 理想流水线（IF / ID）

理想流水线不考虑任何可能的冲突，它的流水接口部件唯一作用就是存储数据，所以在相邻两段相连的路径上分别插入一个寄存器，即可完成理想流水线。以 IF 与 ID 段为例，两段之间的连接情况如图 3.20 所示。

3.4 气泡式流水线实现

在理想流水线中，对于两个相邻流水段，每经过一个时钟周期，前一阶段的值总是直接存储在流水接口部件的寄存器中。而现在，我们需要用插入气泡的方法避免分支冲突和数据冲突，如何实现这个思路，将是本节需要讲解的内容。

1. 分支冲突

在理想流水线中，条件分支指令是在 EX 段判断分支是否成功，而无条件分支则是在 ID 段就进行跳转。如果仍然采用这一方案，对于条件分支冲突，我们需要插入两个气泡，而对于无条件分支冲突，我们却只需要插入一个气泡，这为电路设计造成了不便，因此我们将无条件分支的控制信号也传递到 EX 段（如图 3.21 所示）。

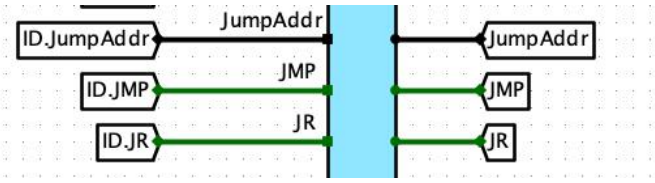

图 3.21 将无条件分支指令的控制信号传到 EX 段

现在，所有的跳转指令都在 EX 段执行了，所以可能产生误取指令的只有前面两段（IF 段和 ID 段）。假设有一条跳转指令在 EX 段判断分支成功了，此时 IF 段和 ID 段各有一条误取指令，IF 段的误取指令即将进入 IF、ID 之间的流水寄存器，ID 段的误取指令即将进入 ID、EX 之间的流水寄存器，为了防止这个现象，我们在流水寄存器的输入端口之前插入一个二路选择器，选择信号为 0 时输入前一流水段的数据，否则输入空数据，而这个选择信号当然就是判断跳转成功的控制信号。这样，当分支跳转时，误取指令就不再会进入流水寄存器，也就不会流经接下来的数据通路。图 3.22 是增加了二路选择器之后，IF 与 ID 之间的流水接口部件。

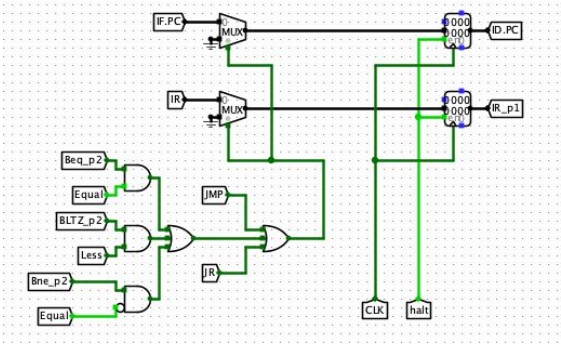

图 3.22 在流水寄存器前插入多路选择器，由判断分支跳转的信号进行选择

2. 数据冲突

当后面指令需要读取前面未执行完的指令即将写入的寄存器时，前一条指令可能正处于 EX 段、MEM 段或 WB 段。如果处于 WB 段，只需要将写入寄存器堆 RF 的触机改为下跳沿，就可以保证前面指令先写数据，后面指令再读数据。而对于其他两种情况，需要实现让后面指令暂停，并在两条指令间插入气泡。

首先我们需要检测后面指令是否需要读取寄存器，只要给出后面指令是什么，我们就可以判断该指令是否要读寄存器，究竟是从 R1 读，还是 R2 读。而我们根据 op 段和 func 段就可以判断指令是什么，因此我们可以建立一个以 op 段和 func 段为输入，R1_used 和 R2_used 为输出的逻辑电路，达到检测目的。

接着，我们可以生成向 EX 段和 MEM 段插入气泡的控制信号 EX.nop 和 MEM.nop。只有满足以下条件时，EX.nop 才能为 1:

- EX 段指令的 RegWrite 为 1（说明前一条指令要写寄存器）
- EX.WriteReg#不为 0（前一条指令的写寄存器不是$0）
- 如果后一条指令 R1_used 为 1，那么 EX.WriteReg#要与 R1#相同；如果后一条指令 R2_used 为 1，那么 EX.WriteReg#要与 R2#相同；如果后一条指令为 syscall 指令，那么 EX.WriteReg#要么是$v0，要么是$a0。总之，前一指令的写寄存器要与后一指令的读寄存器相同。

MEM.nop 的生成条件与 EX.nop 类似，不再赘述。一旦需要插入气泡，意味着要停机，因此 halt 信号要生效。最终生成 EX.nop, MEM.nop 和 halt 信号的电路如图 3.23 所示。

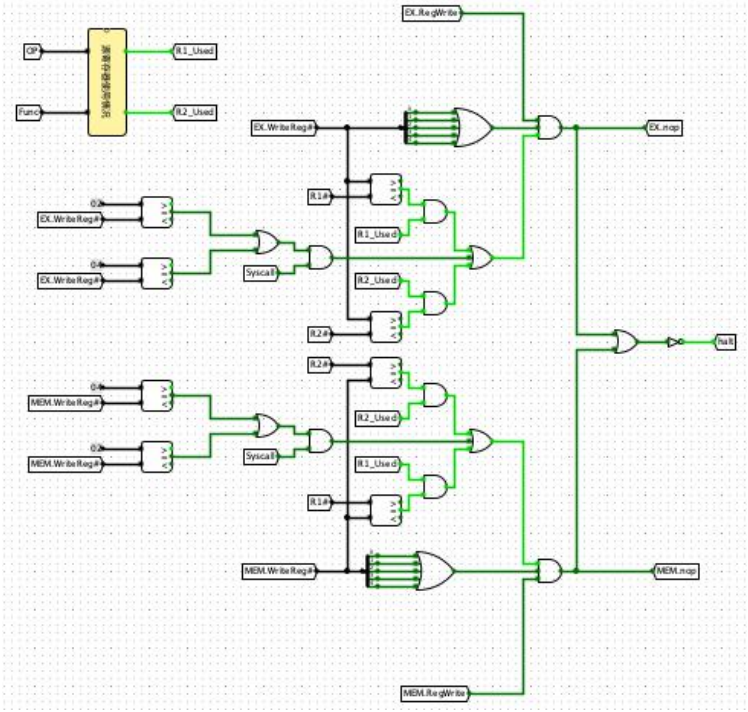

图 3.23 数据相关检测逻辑

最后，将 halt 信号连接到 PC 寄存器的使能端，而 EX.nop 和 MEM.nop 用来选择输入流水寄存器的数据，就可插入气泡。

3.5 数据转发流水线实现

错误! 未定义书签。这一节已经告诉我们，我们可以把前一条指令的数据直接送到后一条指令需要的位置。而后一条指令只有在 EX 段的 R1 和 R2 以及 MEM 段的 R2 处，才可能用到前面指令的寄存器数据。但是 MEM 段的 R2 是由 EX 段的 R2 传送过来的，因此我们只要在 EX 段的 R1 和 R2 处增加几条数据旁路，并且分别用 R1.Forward 和 R2.Forward 信号进行选择就行。

那么前一条指令的哪些地方有可能产生需要的数据然后写入寄存器呢？答案是，这个数据要么由 ALU 输出，要么由存储器输出。但是当后面需要该寄存器数据的指令执行到 EX 段时，前面写该寄存器的指令可能位于 MEM 段，也可能位于 WB 段。这就分为了四种情况。其中，前面该指令位于 MEM 段从存储器输出的情况，我们不考虑，后面将解释原因。

图 3.24 用红色箭头标注了后面指令需要数据的位置，蓝色箭头标注了前面指令可以给出该数据的位置。

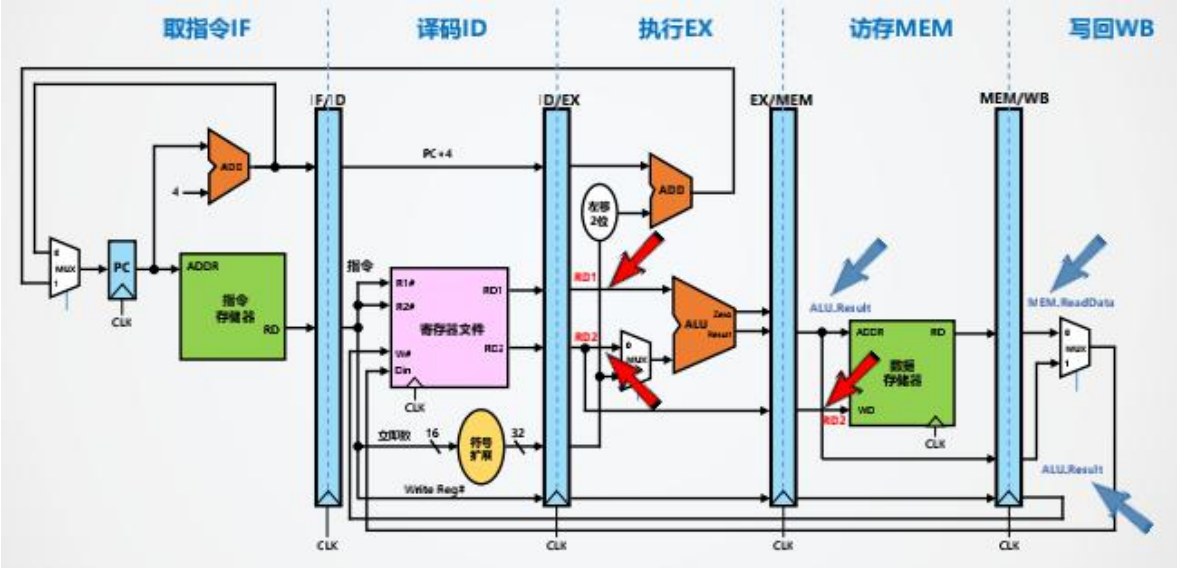

图 3.24 后面指令的红色箭头处可能需要前面指令蓝色箭头指向的数据

建立数据旁路，并用多路选择器通过 Forward 信号选择数据后，建立的局部电路如图 3.25 所示。

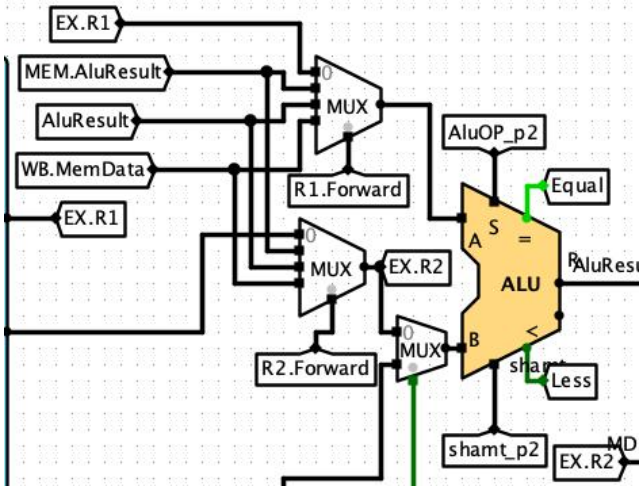

图 3.25 建立数据旁路

如何产生 Forward 信号？选择由多路选择器的哪一路输出数据，其实就是判断前面的指令是从哪一路写的，以及后面的指令使用的哪一个寄存器与之相同。以选择 R1 的数据线路为例，如果要选择 WB.MemData，那么当后面指令位于 ID 段时，前面指令的 MemToReg 和 RegWrite 信号一定为高电平（前面指令要写寄存器并且从内存写），然后用**§错误! 未定义书签。**中数据冲突一节介绍的方法检测两条指令是否存在数据冲突。如果条件满足了，就将 3 传给 R1.Forward 信号，否则再考虑其他数据线路是否满足条件。图 3.26 给出了 R1.Forward 信号生成电路的一部分。

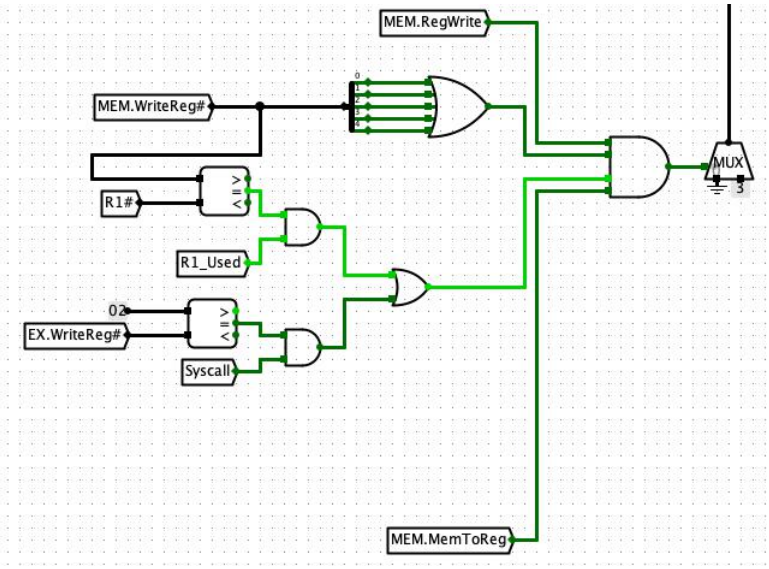

图 3.26 选择 R1.Forward 信号

前面指令从 MEM 段的存储器输出需要的数据的情况，称为 Load-Use 相关。这种情况无法进行数据转发。由于访存时间太长，如果为这种情况建立数据旁路，线路将变成关键路径，CPU 性能就会下降，因此仍然采用气泡式流水线的方法解决这种冲突。

4 实验过程与调试

4.1 测试用例和功能测试

4.1.1 测试用例 1

使用 Educoder 平台上的测试用例测试，通过了关卡

- 单周期 CPU（24 条指令）
- 理想流水线设计
- 气泡流水线设计（EX 段分支 3624 版本）
- 重定向流水线（EX 段分支 2298 版本）
- 单周期 MIPS + 单级中断
- 多级嵌套中断（EPC 硬件堆栈保存）

4.1.2 测试用例 2

```
   #SRAV 移位测试 revise date:2018/3/12 tiger
   #依次输出 0x87600000 0xf8760000 0xff876000 0xfff87600 0xffff8760 0xfffff876
0xffffff87 0xfffffff8 0xffffffff

   .text

   addi $t0,$zero,1 #sllv 移位次数
   addi $t1,$zero,3 #sllv 移位次数
   addi $s1,$zero, 0x876 #
   sll $s1,$s1,20 #

   add $a0,$0,$s1
   addi $v0,$zero,34 # system call for print
   syscall # print

   addi $t3,$zero,8

   srav_branch:
   srav $s1,$s1,$t0 #先移 1 位
   srav $s1,$s1,$t1 #再移 3 位
   add $a0,$0,$s1
   addi $v0,$zero,34 # system call for print
   syscall # print
   addi $t3,$t3, -1
   bne $t3,$zero,srav_branch #循环 8 次

   addi $v0,$zero,10 # system call for exit
   syscall # we are out of here.
```

4.1.3 测试用例 3

```
   #XOR 测试    revise date:2018/3/12 tiger
   # 0x00007777 xor   0xffffffff =  0xffff8888
   # 0xffff8888 xor   0xffffffff =  0x00007777
   #依次输出 0x00007777 0xffff8888 0x00007777 0xffff8888 0x00007777 0xffff8888
0x00007777 0xffff8888 0x00007777 0xffff8888 0x00007777 0xffff8888 0x00007777
0xffff8888 0x00007777 0xffff8888 0x00007777

   .text

   addi $t0,$zero,-1    #
   addi $s1,$zero, 0x7777    #

   add $a0,$0,$s1
   addi $v0,$zero,34        # system call for print
   syscall               # print

   addi $t3,$zero, 0x10

   xor_branch:
   xor $s1,$s1,$t0    #先移 1 位
   add $a0,$0,$s1
   addi $v0,$zero,34        # system call for print
   syscall               # print
   addi $t3,$t3, -1
   bne $t3,$zero,xor_branch   #循环 8 次

   addi   $v0,$zero,10        # system call for exit
   syscall               # we are out of here.
```

4.1.4 测试用例 4

```
    #LB 测试    revise date:2018/3/12 tiger
    #依次输出 0xffffff81 0xffffff82 0xffffff83 0xffffff84 0xffffff85 0xffffff86
0xffffff87 0xffffff88 0xffffff89 0xffffff8a 0xffffff8b 0xffffff8c 0xffffff8d
0xffffff8e 0xffffff8f 0xffffff90 0xffffff91 0xffffff92 0xffffff93 0xffffff94
0xffffff95 0xffffff96 0xffffff97 0xffffff98 0xffffff99 0xffffff9a 0xffffff9b
0xffffff9c 0xffffff9d 0xffffff9e 0xffffff9f 0xffffffa0
    .text
    addi $t1,$zero,0    #init_addr
    addi $t3,$zero,16    #counter

    #预先写入数据，实际是按字节顺序存入 0x81,82,84,86,87,88,89.......等差数列
    ori $s1,$zero, 0x8483  #
    addi $s2,$zero, 0x0404  #
    sll $s1,$s1,16
    sll $s2,$s2,16
    ori $s1,$s1, 0x8281  #    注意一般情况下 MIPS 采用大端方式
    addi $s2,$s2, 0x0404  #   init_data= 0x84838281 next_data=init_data+ 0x04040404
    lb_store:
    sw $s1,0x10010000($t1)
    add $s1,$s1,$s2   #data +1
    addi $t1,$t1,4    # addr +4
    addi $t3,$t3,-1   #counter
```

```
    bne $t3,$zero,lb_store

    addi $t3,$zero,32   #循环次数
    addi $t1,$zero,0    # addr
    lb_branch:
    lb $s1,0x10010000($t1)        #测试指令
    add $a0,$0,$s1
    addi $v0,$zero,34        #输出
    syscall
    addi $t1,$t1, 1
    addi $t3,$t3, -1
    bne $t3,$zero,lb_branch



    addi   $v0,$zero,10        # system call for exit
    syscall               # we are out of here.
```

4.1.5 测试用例 5

```
    #bltz 测试    小于 0 跳转   累加运算，从负数开始向零运算 revise date:2018/3/12 tiger
    #依次输出 0xffffffff1 0xffffffff2 0xffffffff3 0xffffffff4 0xffffffff5 0xffffffff6
0xffffffff7 0xffffffff8 0xffffffff9 0xfffffffa 0xfffffffb 0xfffffffc 0xfffffffd
0xfffffffe 0xffffffff
    .text
    addi $s1,$zero,-15      #初始值
    bltz_branch:
    add $a0,$0,$s1
    addi $v0,$zero,34
    syscall               #输出当前值
    addi $s1,$s1,1
    bltz $s1,bltz_branch     #当前指令


    addi   $v0,$zero,10
    syscall               #暂停或退出
```

4.2 性能分析

使用 benchmark 对单周期、气泡流水线和重定向流水线 MIPS CPU 分别进行测试，最后测试的周期数分别为 1546、3624、2298。

单周期 CPU 尽管周期数最少，但是由于没有细分关键路径，因此时钟周期一般比后面两种流水线 CPU 长得多，性能可能最差。

对于流水线 CPU，程序开始执行时，需要几个额外的时钟周期让指令进入流水段。一旦出现指令冲突，气泡流水线就会插入若干气泡，因此周期数是最多的；而重定向流水线只有在 Load-Use 冲突的情况插入气泡，因此它的额外开销相比气泡

流水线要少。

4.3 主要故障与调试

4.3.1 流水线 syscall 故障

所有流水线： syscall 停机失败。

故障现象：在流水线 CPU 中，当执行到 syscall 指令，并且$v0 == 10 的情况下，CPU 仍然不停机。

原因分析：halt 信号只连接到 PC 寄存器的使能端，遇到 syscall 指令时，流水寄存器仍然在运作，这样 syscall 指令经过若干时钟周期，又从流水线上被排出了。

解决方案：在 IF / ID 的流水寄存器使能端也连上 halt 信号，这样运行到 syscall 指令时，PC 寄存器和流水寄存器均停止工作，才能保证 CPU 真正停机。

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。
第三天	实现了单周期 CPU，并且通过了 Logism 单周期 CPU 的检查。
第四天	实现了理想流水线 CPU 的数据通路，并且通过了理想流水线的检查。
第五天	实现了源寄存器和数据相关检测的电路逻辑。
第六天	完成了气泡流水线，通过了气泡流水线的检查。
第七天	完成了可以生成 Forward 信号的重定向逻辑。
第八天	完成了重定向流水线，通过了重定向流水线的检查。
第九天	完成了单周期 MIPS+单级中断，并通过了相应测试。
第十天	实现了多周期 MIPS+多级嵌套中断，并通过了相应测试。

5 设计总结与心得

5.1 课设总结

计算机组成原理课设完成的任务是对 MIPS CPU 进行由简到繁的设计。主要作了如下几点工作:

1) 设计了单周期 MIPS CPU。
2) 设计了理想流水线 MIPS CPU。
3) 设计了气泡流水线 MIPS CPU。
4) 设计了重定向流水线 MIPS CPU。
5) 设计了单周期 MIPS + 单级中断。
6) 设计了多周期 MIPS + 多级嵌套中断。
7) 完成了团队任务，可以用自己设计的 CPU 演示团队任务。

5.2 课设心得

本次课程设计可以说是迄今为止所有实验以及课程设计中难度最大也最有价值的项目之一。单周期 CPU 的设计和理想流水线的设计中规中矩，难度不算太大。后面的气泡流水线、重定向流水线以及中断电路的设计难度比较大，老师详细的视频讲解让我有了很多思路, 引导我自己完成了这些模块的设计, 这让我很有成就感。

虽然能够自己动手完成这些流水线电路，实现相应的功能，然后通过测试，但总觉得距离商用 MIPS CPU 有比较大的差距。做完课设之后, 很想知道上市的 MIPS CPU 到底是如何设计的, 如果在课设结束时, 老师能提供一些开放的商用 MIPS CPU 的设计电路及讲解视频，给我们参考参考，我们可以将自己的 CPU 与它们进行比对，反思自己的不足在哪里。

现在的主流 CPU 除了采用 Intel 的 CISC 架构以外, 还有 ARM 这样的体系结构, 而 RISC-V 架构则是新兴的一种体系结构。相信过不久，咱们的课程设计也会转而面向这些架构来进行吧。

6 团队任务

我属于小萝卜生产队，我们的团队任务是做一个简单的文字编辑器，并且有程序启动动画。想要实现这一功能，必须使硬件与软件协同工作。这其中，我负责了电路设计、软件框架设计以及程序动画设计这些部分。

电路设计上，我们通过在键盘上按键，向 CPU 发送信号，CPU 收到信号后，发生中断相应，根据键盘发送的 ascii 码，跳转到对应的中断子程序。这个电路主要是在单周期 MIPS + 单级中断电路的基础上加以改进。

如何根据 ascii 码选择正确的中断程序是我遇到的一个问题, 一开始计划使用多路选择器，由 ascii 码选择相应字符的子程序地址。但是因为字符太多，我们没有这么大的选择器，所以最后改用只读存储器。以 ascii 码为地址，去相应的存储单元读取入口地址，然后传给 PC 寄存器（如图 6.1 所示）。

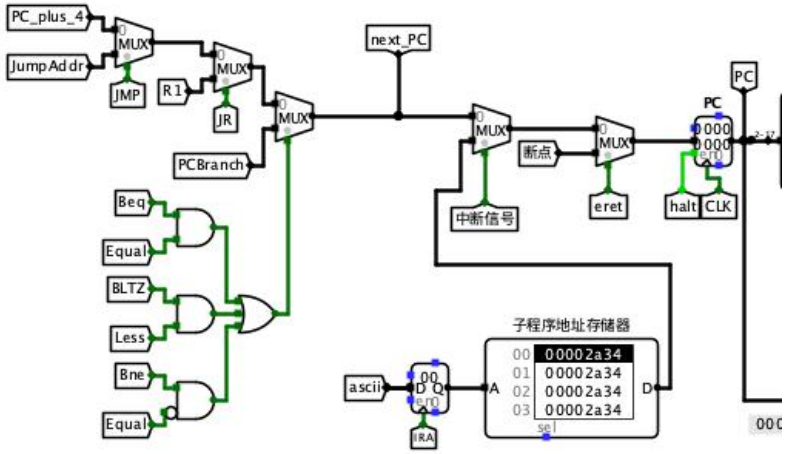

图 6.1 如何保存所有中断程序入口

软件方面需要我们设计中断程序。我们用$a1, $a2 两个寄存器存储要下一个打印的字符的位置。每次打印字符时将位置、颜色、复位等信息转存到$a0，然后使用指令 syscall，把这些信息传给屏幕。打印完一个字符后，更新$a1, $a2 的值，然后再打印光标。

打印开机动画是我遇到的另一个问题。我的实现方法是先用 matlab 程序从 png 图像中提取出一个矩阵，矩阵的每个元素是一个 rgb 值。然后将矩阵线性化存储到内存中，打印图像时 CPU 通过 lw 指令把矩阵取出来复现图像。

·指导教师评定意见·

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: 周易