華中科技大學

2017

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：	CS1409
学　　号：	U201414803
姓　　名：	许荣仁
电　　话：	15629069226
邮　　件：	xuduo1017@live.com
完成日期：	2017-03-26

计算机科学与技术学院

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	XOR	异或	指令格式参考 MIPS32 指令集，最终功能以 MARS 模拟器为准。
29	SLTIU	小于立即数置 1 (无符号)	
30	LB	加载字节	
31	BLEZ	小于等于 0 转移	

2 总体方案设计

2.1 单周期 CPU 设计

CPU 整体应该包括指令存储区 ROM，主存 RAM，寄存器文件，ALU，控制单元，多个多路选择器（跳转时修改 PC，立即数与寄存器值的选择，寄存器号的选择，是否读写内存的选择等），位扩展器等。按照指令执行的顺序构建数据通路，需要时增加必要的多路选择器来增加功能。总体结构图如图 2.1 所示。

译码：将指令的 6 位 OP 和 6 位 FUNC 作为输入，24 条指令是否是当前被解释的这一条作为输出，用不同指令的不同 OP 与 FUNC 的对应关系，把每条指令的表达式输入到软件中，自动产生电路。

控制单元：控制单元需要根据不同的指令产生控制信号，比如 ALU 的 op，各个多路选择器的选择信号等。

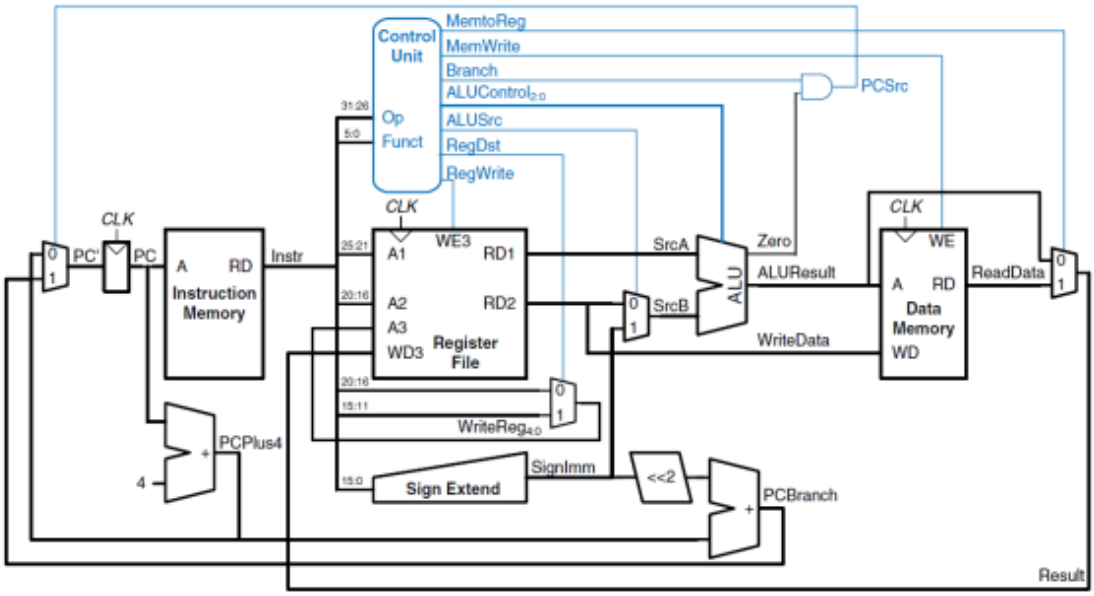

图 2.1 总体结构图

2.1.1 主要功能部件

1. 程序计数器 PC

PC 用一个寄存器保存，基本的运算是每次取出一条指令后其值加 4 再写回到寄

存器，此外还要连接多路选择器，实现跳转、分支和中断的功能，多路选择器的控制信号由指令译码后结合运算的结果得到。

2. 指令存储器 IM

指令存储器由一块 ROM 构成，只读，可以根据输入的地址把对应的指令输出，不需要其他特别的功能。

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

RegFile 理论上应包含 32 个 32 位寄存器，但由于第一个恒为零，所以实际上需要 31 个寄存器，第 0 个用 32 位的常量 0 代替；

每个寄存器都需要读或者写，但读哪两个是由输入信号 R1#和 R2#决定的，所以需要两个多路选择器选择要输出的数据，多路选择器的输入数据端都连接所有的寄存器输出端，选择端分别连接 R1#和 R2#，输出端分别连 reg1 和 reg2；写哪一个寄存器是有 RW#决定的，所以要用两个解复用器把要写入的数据和写使能信号传送给相应的寄存器；

2.1.2 数据通路的设计

一条指令执行的大致执行路径如下，依据指令的不同会走不同的路径。

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，统计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
ALUOP	0~15	为运算器选择正确的运算模式
ext_u	0	为立即数选择符号扩展
	1	为立即数选择无符号扩展
alu_src	0	为 alu 的第二个操作数选择寄存器的值
	1	为 alu 的第二个操作数选择立即数扩展的结果
s_i	0	R1#选择 21-25 位
	1	R1#选择 16-20 位，立即数选择 6-10 位的扩展
reg_dst	0	RW 选择 16-20 位
	1	RW 选择 11-15 位
reg_write	0	不写寄存器
	1	需要写寄存器
mem_write	0	不写内存
	1	需要写内存
mem_to_reg	0	写回结果是运算结果
	1	写回结果是以运算结果为地址取出的内存单元的值

控制信号	取值	说明
j、jal、jr、bne、beq、syscall、LB、ERET、MFTC0、BLEZ	0/1	当前指令是其中的某条指令（1 有效）

2.2 中断机制设计

2.2.1 总体设计

MFC0 指令的格式是 mfc0 rt, rd，指令的寻址方式设计是寄存器寻址，rt 是通用寄存器的编号，rd 是协处理器中寄存器的编号；

MTC0 指令的格式是 mtc0 rt, rd，指令的寻址方式设计是寄存器寻址，rt 是通用寄存器的编号，rd 是协处理器中寄存器的编号；

ERET 指令的格式是 eret，指令的寻址方式设计是寄存器寻址，但寄存器编号并不写在指令中，是事先约定好的；

中断是打断当前程序的运行，进入到中断服务程序，需要由协处理器完成，接收到中断请求时把信号发送给 CPU，

2.2.2 硬件设计

先根据输入的三个中断信号与当前的中断屏蔽字的非进行与操作，然后选择出一个优先的中断，再根据当前是否允许中断输出或不输出中断请求，把当前的中断号写入 CAUSE，PC 值写入 EPC，进入中断服务程序。在中断服务程序中可以读取 EPC、STATUS 和 CAUSE 三个寄存器的值，可以修改 EPC 和 STATUS 的值。

2.2.3 软件设计

中断程序的嵌套和保护现场等操作都是软件实现的。

保护现场：进入中断服务程序后，先关中断，然后保存 EPC 和 STATUS，再把中

断服务需要用到的寄存器的值入栈（把寄存器的值用一个递增的地址写入到内存条中，保护现场），再设置中断屏蔽字，开中断，开始中断程序，程序结束后关中断，恢复现场，恢复 STATUS 和 EPC，再开中断，最后 eret 返回主程序。

中断嵌套：把 STATUS 中的 1-3 位作为屏蔽字，根据当前的中断号设置不同的屏蔽字（001，011，或 111），实现了低优先级中对高优先级中断的嵌套。

2.3 流水 CPU 设计

2.3.1 总体设计

在未加中断的单周期 CPU 基础上进行修改，把一条指令的执行过程分割为 IF，ID，EX，MEM，WB 五个阶段，每两段之间把需要传递的信号用寄存器保存，实现理想流水线；再在理想流水线基础上处理分支、跳转、数据转发和 loaduse 冲突等内容。

2.3.2 流水接口部件设计

每两个阶段由一组寄存器连接，寄存器的输入端为上一阶段的传递过来的信号（或所需的值），输出端连接到下一阶段所用的输入，寄存器的位数由信号的位数决定，寄存器的个数由下一阶段所需要的信号的数量决定，所有寄存器共用时钟和清零信号。

2.3.3 理想流水线设计

理想流水线不用考虑分支跳转数据冒险等问题，只需要把一条指令的执行分为 5 个阶段：IF、ID、EX、MEM 和 WB。

IF 阶段从指令存储器中取出一条指令，并把 PC 加 4，准备取下一条指令。

ID 阶段把 IF 阶段取出的指令进行译码，得到后续阶段所需要的全部控制信号和源操作数等信息（包括从寄存器组中取出所需要的寄存器的值）。

EX 阶段使用 ALU 进行运算，源操作数和控制信号都由 ID 阶段传过来，得出运算结果后传递到下一阶段。

MEM 阶段根据 ID 阶段传过来的信号进行访存，选择把运算结果直接传递给下一阶段或者依据运算结果取出对应内存单元中的值传递给下一阶段。

WB 阶段把 MEM 阶段传过来的结果写回到指定编号的寄存器中。

2.4 数据转发流水线设计

数据转发的需求：在某些情况下，前一条或前两条指令已经计算或取到了正确的结果，但结果还没有写回到当前指令需要用到的寄存器中，即当前指令的 EX 阶段需要用到的数据，还在流水线寄存器中存放着，没有写回到目标寄存器中，这样当前指令用到的数据就是旧的数据，数据转发就是在遇到这种冲突的时候把运算或访存结果转发到 alu 的输入端，再用一些控制信号选择是否使用当前的转发数据。具体过程：

需要进行转发的数据：前一条指令在 EX/MEM 寄存器中存放的 alu 运算结果，前前条指令在写回之前要写入寄存器的数据。

转发条件：当前指令的源操作寄存器编号与前一条指令的目标寄存器编号相同，且前一条指令不是访存指令；当前指令的源操作寄存器编号与前前条指令的目标寄存器编号相同。

根据前面的两个规则可以设计出转发模块。

2.5 气泡式流水线设计

气泡的需求：在某些情况下，转发不能解决数据先写后读的冲突，比如：前一条指令是 load 指令，当前指令需要用到前一条指令的结果，但当前指令在 EX 阶段时，前一条指令还在 MEM 阶段，只有再过一个时钟周期，才能取出数据，但那是当前指令已经完成了 EX 阶段，两者无法仅仅依靠转发解决冲突，就需要让当前指令暂停一个时钟周期，也就是在当前指令与前一条指令之间插入一个气泡。

插入气泡的实现：当产生 loaduse 冲突时，把 ID/EX 的寄存器清除，PC 寄存器和 IF/ID 寄存器的使能设为无效，就在当前指令与前一条指令之间插入了一个气泡。

2.6 分支和跳转流水线设计

跳转是在 EX 阶段完成的，误取深度为 2，判断出是跳转指令时，把计算出的新的 PC 值写入到 PC 寄存器中，并且把之前误取的两条指令清除掉。

分值指令有两种可能的结果，不成功不需要做特殊处理，只要继续执行就可以；而分支成功时，与跳转指令进行的操作相同，更新 PC，清除误取的两条指令。

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) 系统时钟（clk）

① Logism 实现：

使用最初时钟与停机指令的或结果作为系统的时钟，可以实现在不影响正常指令的前提下实现停机功能。比较器的另一个输入是寄存器组$s1 的值，与门的另一个输入是 syscall 信号，当$s1 为 10，且当前是 syscall 指令，与门的输入变为 1，系统时钟不再变化，一直为 1，实现了停机。如图 3.1 所示。

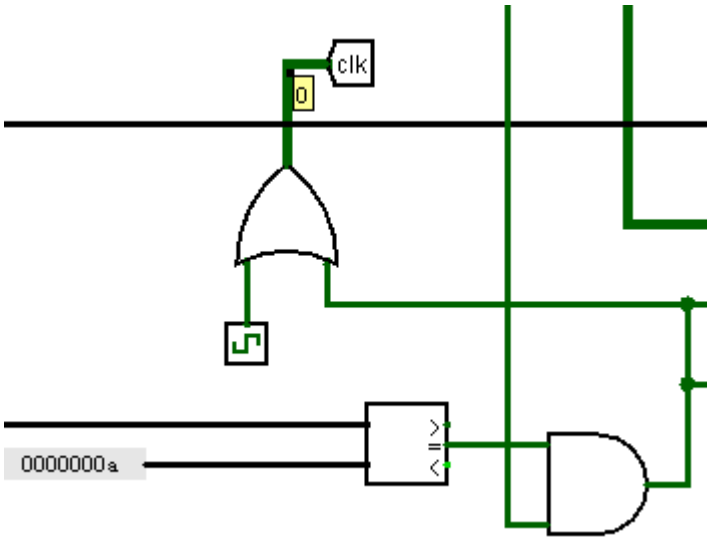

图 3.1 时钟产生

② FPGA 实现：

系统时钟 clk 的 Verilog 代码如下：

```
assign SyscallAddOne = (v0==10 && isSyscall);
assign clkout = (clkin || SyscallAddOne);
```

2) 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，由 clk 控制。如图 3.2 所示。

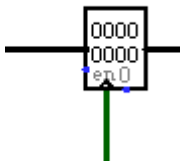

图 3.2 程序计数器（PC）

3) 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 3.3 所示。

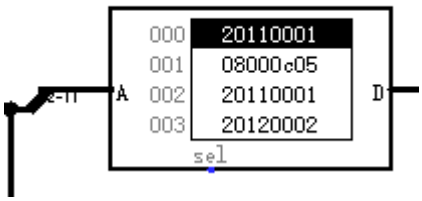

图 3.3 指令存储器（IM）

4) 数据存储器（DM）

① Logism 实现：

使用一个可读写存储器 RAM 实现数据存储器（DM）。设置该存储器的地址位宽为 10 位，数据位宽为 32 位。因为 ALU 运算结果有 32 位，而 ROM 地址线宽度有限，仅为 10 位，故将 32 位地址高位部分（12-31 位）和字节偏移部分（低 2 位）直接屏蔽，使用分线器只取 32 位指令地址的 2-11 位作为存储器的输入地址。还有数据输入端 D（左）和写使能端（str），如图 3.4 所示。

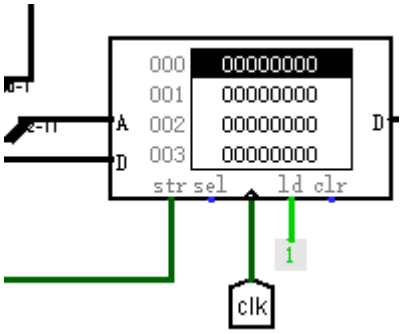

图 3.4 数据存储器（DM）

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			
SLL	PC+4	PC		rt	rd	alu	r2	立即数	0			
SRA	PC+4	PC		rt	rd	alu	r2	立即数	1			
SRL	PC+4	PC		rt	rd	alu	r2	立即数	2			

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			
OR	PC+4	PC	rs	rt	rd	alu	r1	r2	8			
ORI	PC+4	PC	rs		rt	alu	r1	立即数	8			
NOR	PC+4	PC	rs	rt	rd	alu	r1	r2	10			

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，大致如图 3.5 所示。

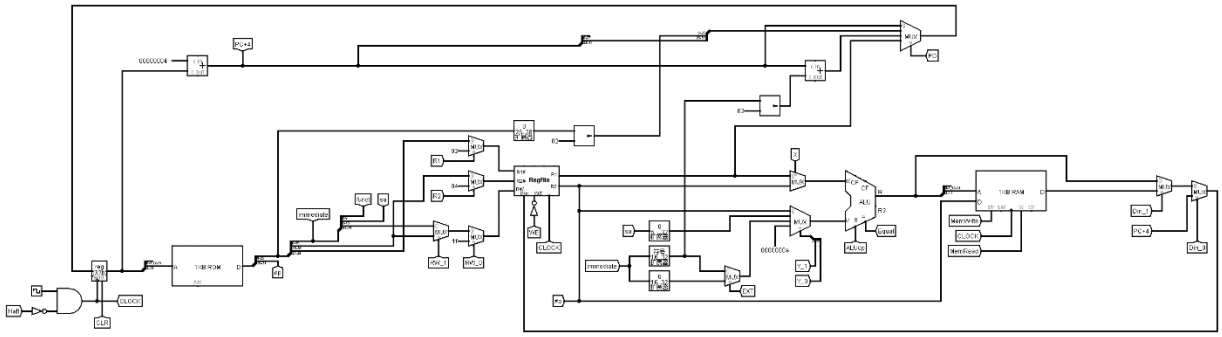

图 3.5 单周期 CPU 数据通路（Logism）

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 Logism 上进行主控制器、Branch 控制器、SYSCALL 控制器的具体实现。

对照表 3.2 所示。

表 3.2 主控制器控制信号

指令	mem_to_reg	reg_dst	reg_write	s_i	ext_u	alu_src	ALUop	MemWrite
ADD	0	0	1	0	0	0	0101	0
ADDI	0	1	1	0	0	1	0101	0
ADDIU	0	1	1	0	0	1	0101	0
ADDU	0	0	1	0	0	0	0101	0

指令	mem_to_reg	reg_dst	reg_write	s_i	ext_u	alu_src	ALUop	MemWrite
AND	0	0	1	0	0	0	0111	0
ANDI	0	1	1	0	1	1	0111	0
SLL	0	0	1	1	0	1	0000	0
SRA	0	0	1	1	0	1	0001	0
SRL	0	0	1	1	0	1	0010	0
SUB	0	0	1	0	0	0	0110	0
OR	0	0	1	0	0	0	1000	0
ORI	0	1	1	0	1	1	1000	0
NOR	0	0	1	0	0	0	1010	0
XOR	0	0	1	0	0	0	1001	0
LW	1	1	1	0	0	1	0000	0
LB	1	1	1	0	0	1	0000	0
SW	0	0	0	0	0	1	0000	1
BEQ	0	0	0	0	0	0	0000	0
BNE	0	0	0	0	0	0	0000	0
SLT	0	0	1	0	0	0	1011	0
BLEZ	0	0	0	0	0	1	1011	0
SLTI	0	1	1	0	0	1	1011	0
SLTU	0	0	1	0	0	0	1100	0
SLTIU	0	1	1	0	0	1	1100	
J	0	0	0	0	0	0	0000	0
JL	0	0	1	0	0	0	0000	0
JR	0	0	0	0	0	0	0000	0
SYSCALL	0	0	0	0	0	0	0000	0

此外 j、jal、jr、bne、beq、syscall、LB、ERET、MFTC0、BLEZ 这些信号直接传递出去，在主电路中作为选择信号使用。

① Logisim 实现

现根据输入的指令的高六位和低六位进行判断当前是哪一条指令（判断 BLEZ 需要额外的 16-20 位），因为每次只能自动生成 12 条电路，所以这一步需要三次使用 Logsim 自动生成电路的功能，再把三次生成的电路放到一个电路中进行手动合并，最后根据不同指令连接到不同的或门上作为输出。此外还要统计不同类型指令的条数，所以还要把指令分类，分别作为对应计数器的使能信号：

最终便可以实现整个主控制器中所有控制信号的生成。在 Logisim 中控制器原理图（部分）如图 3.6 所示。

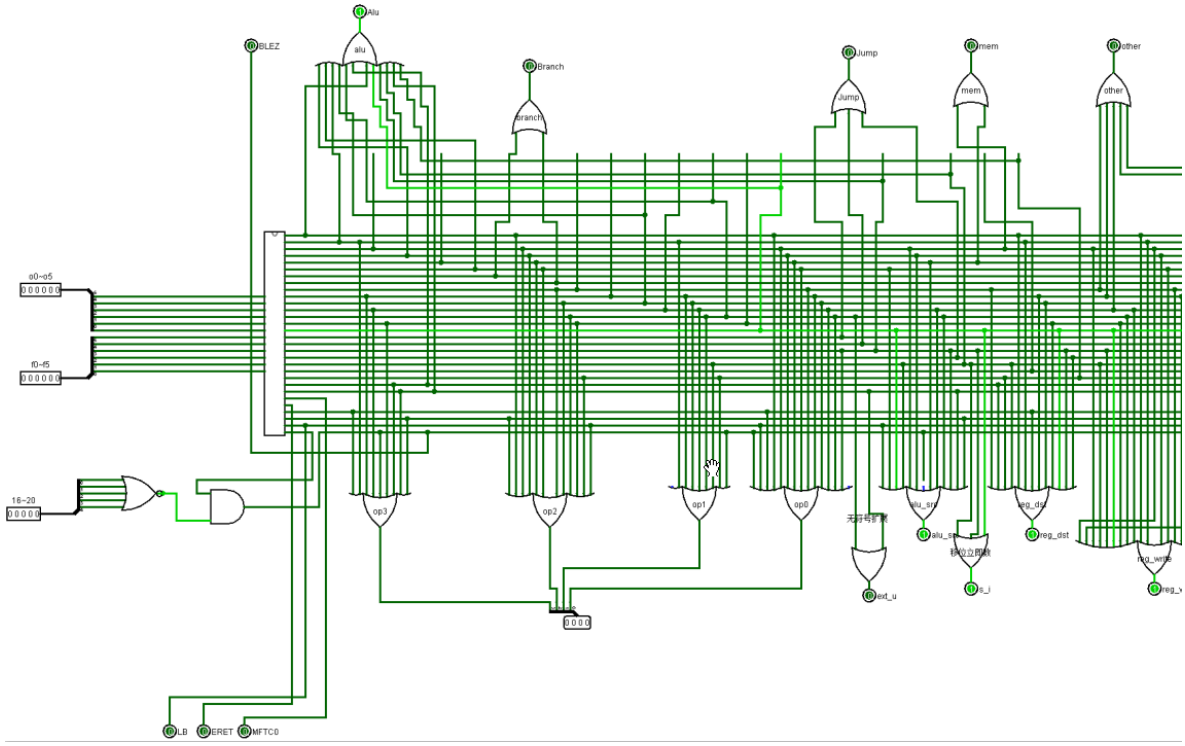

图 3.6 主控制器原理图

3.2 中断机制实现

3.2.1 CP0 整体输入输出

整个 CP0 需要从主电路中接收多个信号，并负责中断的相应判断返回等操作，其输入输出接口如图 3.7 所示：

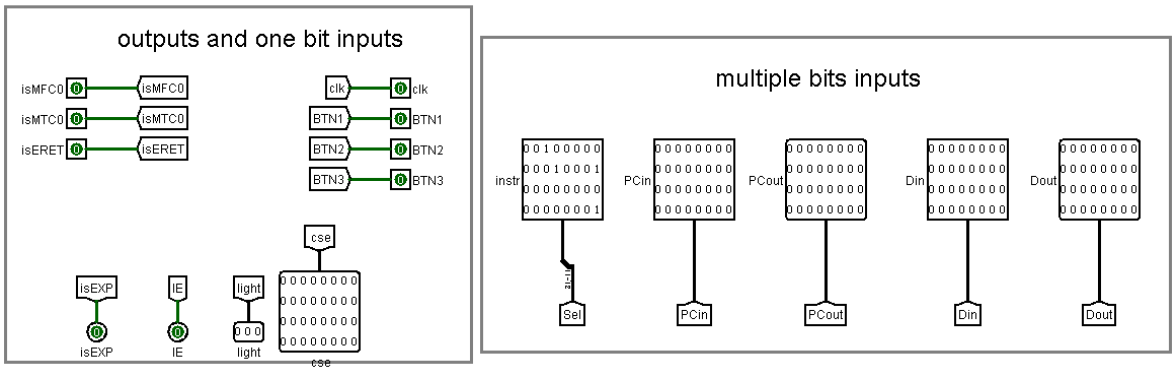

图 3.7 CP0 输入输出接口

3.2.2 中断产生与识别

发生中断信号后，传到 CP0 中，保存到一个 D 触发器中，然后判断是否已经有了相同的中断，有则不再操作，没有则再保存到一个 D 触发器中，然后把全部三个触发器的值与中断屏蔽字的非进行与操作，再把三个结果连接到一个优先编码器，输出中断号 1，2 或 3，最后与是否允许中断（关中断）的信号与操作，最后即为是否产生了中断的信号，把这个信号输出，传给主电路。模块的 Logisim 图如图 3.8 所示：

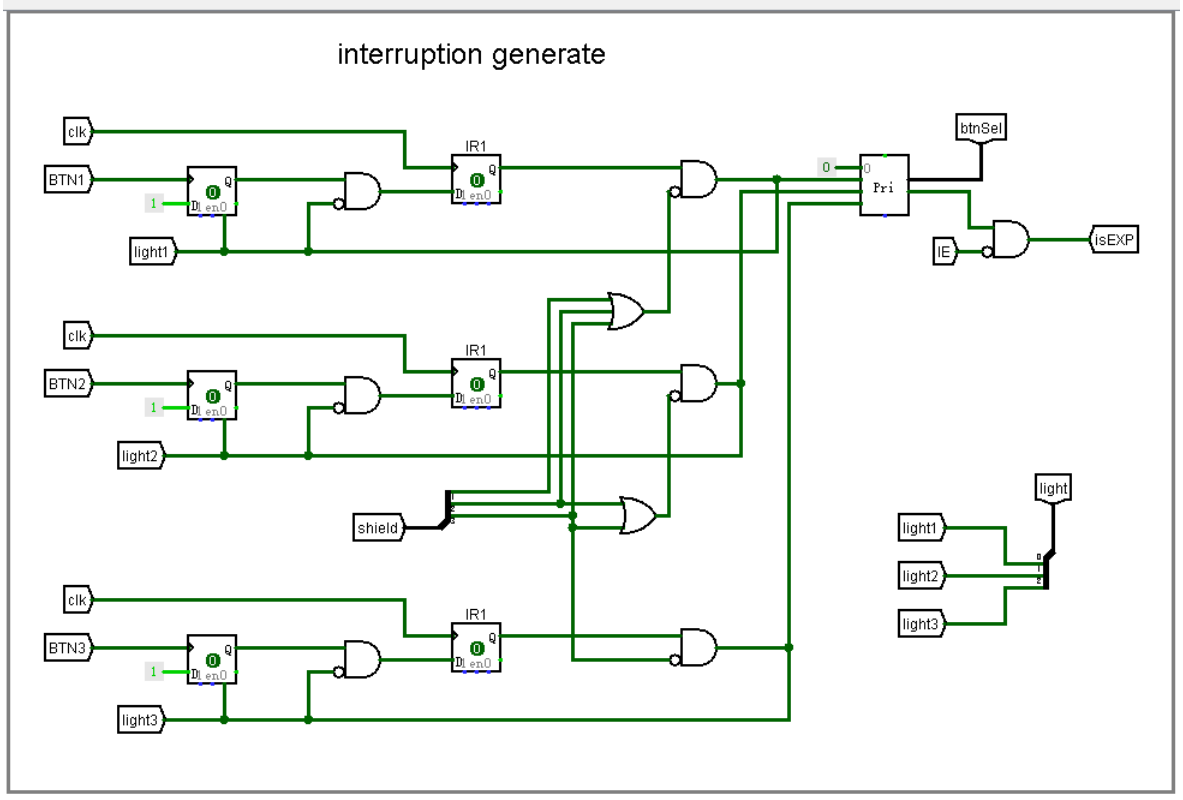

图 3.8 中断产生电路

3.2.3 CP0

CP0 需要在中断发生时把当前 PC 的值保存到 EPC 寄存器，识别出中断号，并把中断号写入到 CAUSE 寄存器，此外还有一个 STATUS 寄存器，保存当前的状态（第 0 位为开关中断，1-3 位为中断屏蔽字），CP0 可以根据指令的不同对 EPC 和 STATUS 寄存器进行写操作，对全部三个寄存器进行读操作并把值输出。PCout 信号始终是与 EPC 相连的，中断返回的时候需要把 EPC 的值写回到主电路的 PC 寄存器中，这样一直相连便可以直接把在判读出是中断返回指令的时候把 PCout 的值写回。此模块的 Logisim 模拟图如 图 3.9 所示：

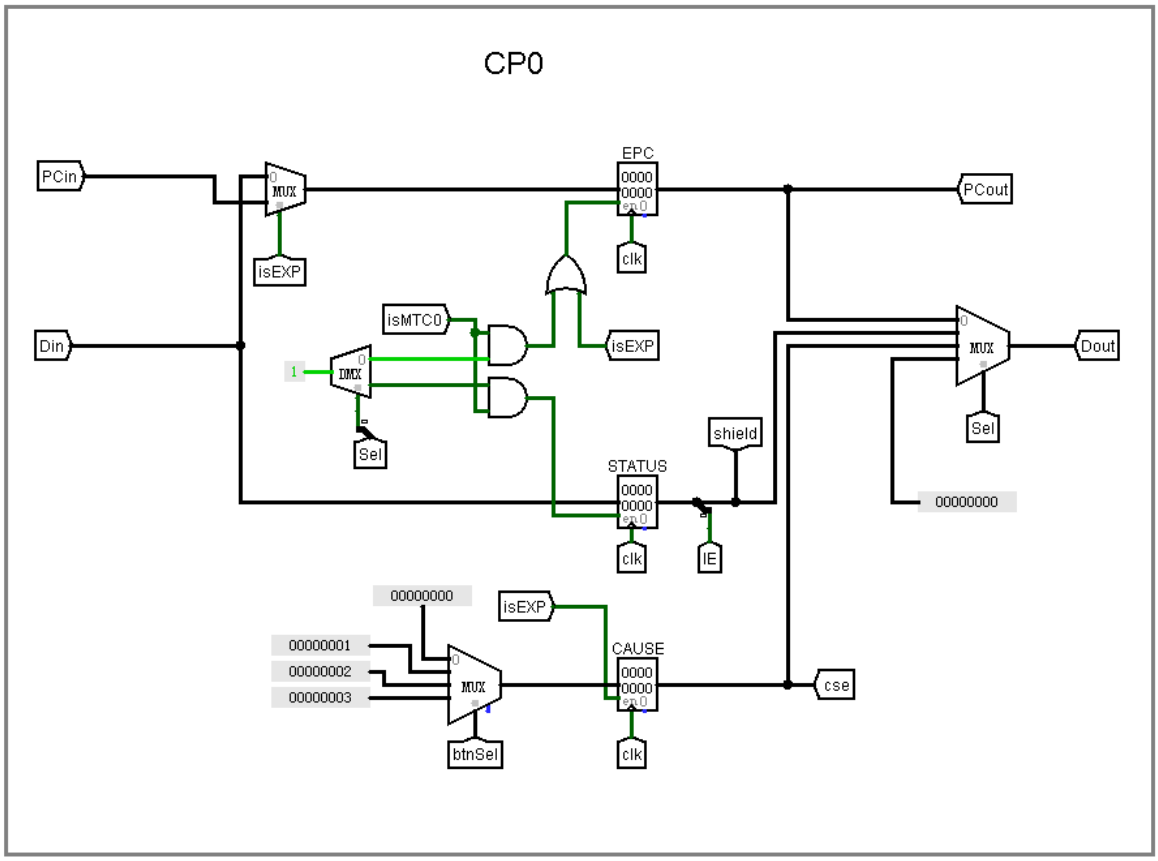

图 3.9 CP0 模块

添加中断后的 ROM 分为了两部分，第一部分是主程序，第二部分存放的是中断服务程序，这样做主要是为了方便加载镜像，区分中断程序与主程序，便于调试。其

整体 Logisim 电路如 图 3.10 所示：

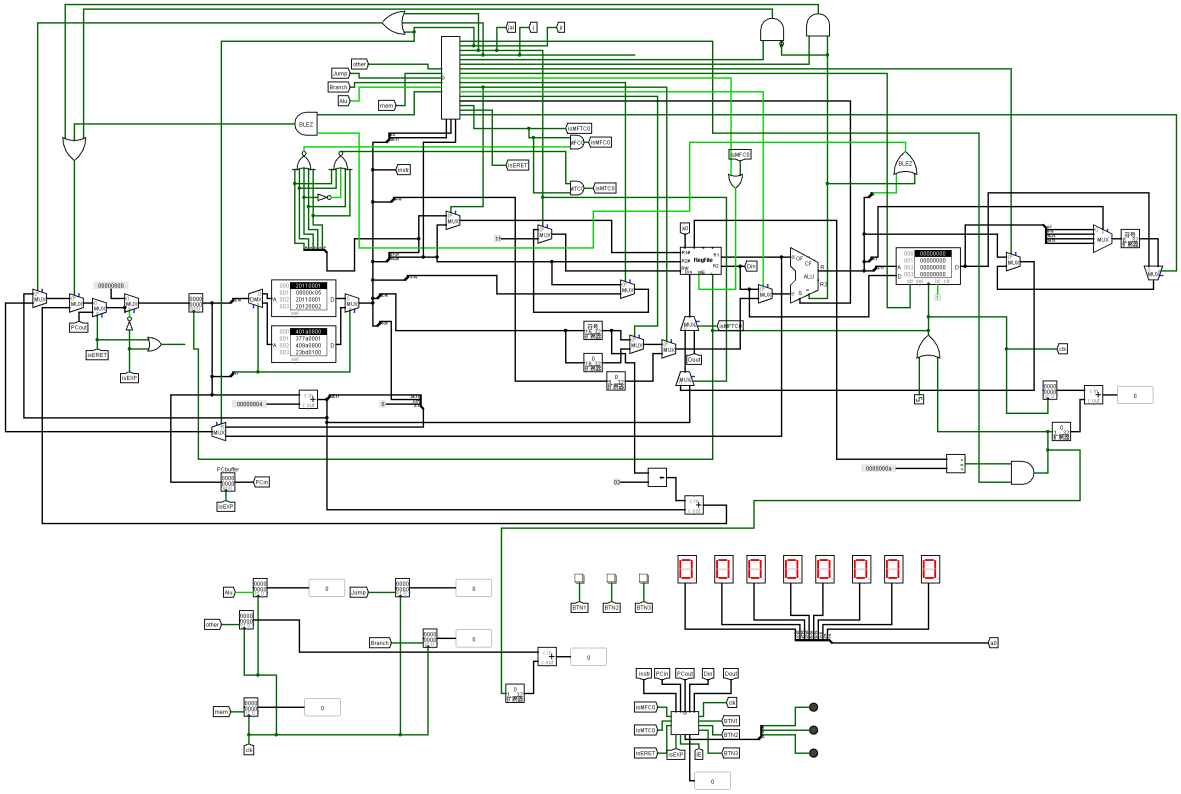

图 3.10 中断单周期 CPU 整体图

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

每两个阶段由一组寄存器连接，寄存器的输入端为上一阶段的传递过来的信号（或所需的值），输出端连接到下一阶段所用的输入，寄存器的位数由信号的位数决定，寄存器的个数由下一阶段所需要的信号的数量决定，所有寄存器共用时钟和清零信号（部分组不需要清零）。

Logisim 实现：把多个寄存器连接起来即可，IF/ID，ID/EX，EX/MEM，MEM/WB 阶段的接口部件基本相同，此处只展示 IF/ID 电路和 ID/EX 的部分电路（电路太长无法全部展示）如图 3.12 和 图 3.11 所示：

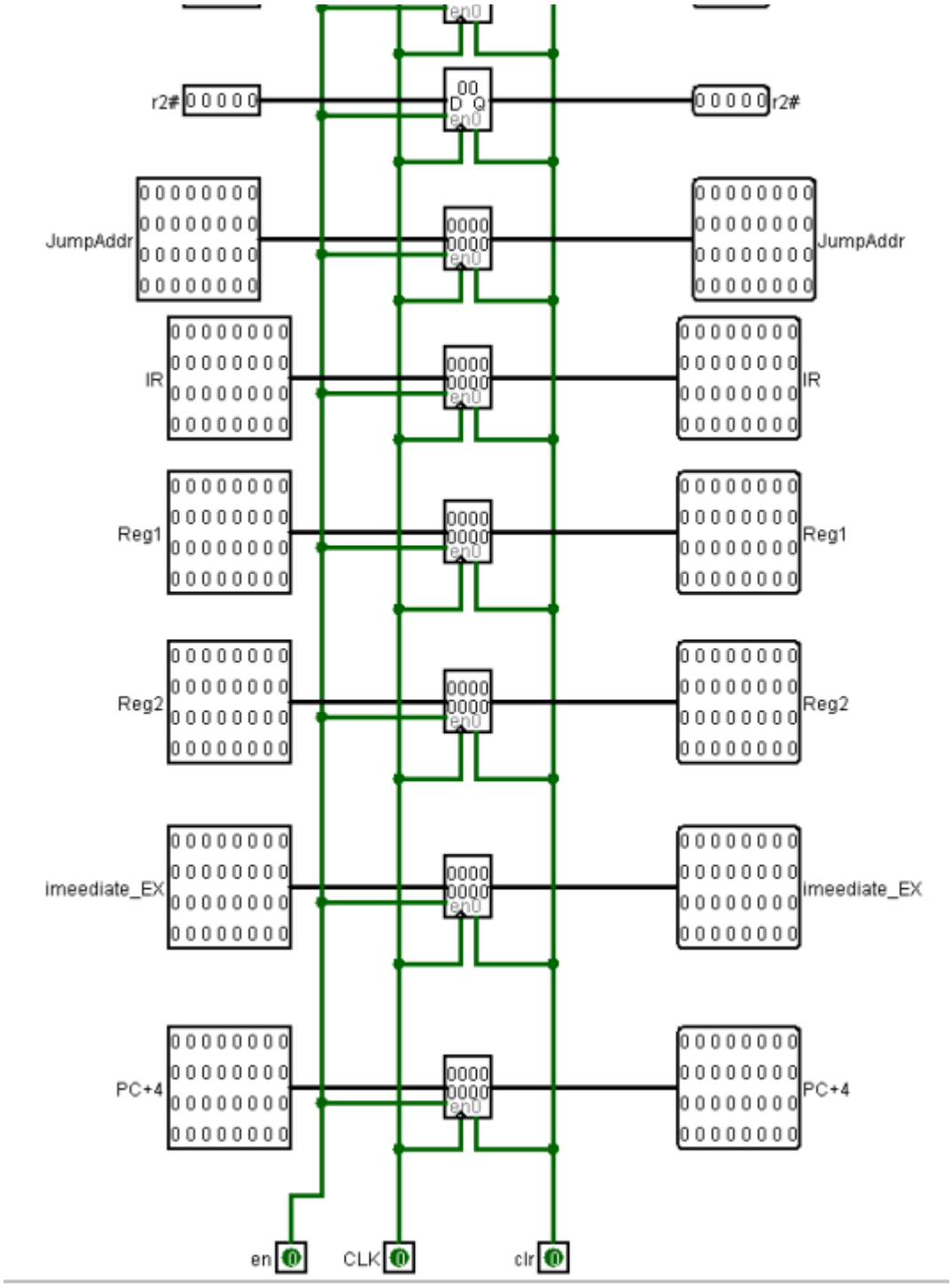
r2#
r2#
JumpAddr
JumpAddr
IR
IR
Reg1
Reg1
Reg2
Reg2
imeediate_EX
imeediate_EX
PC+4
PC+4
en
CLK
clr

图 3.11 ID/EX（部分）

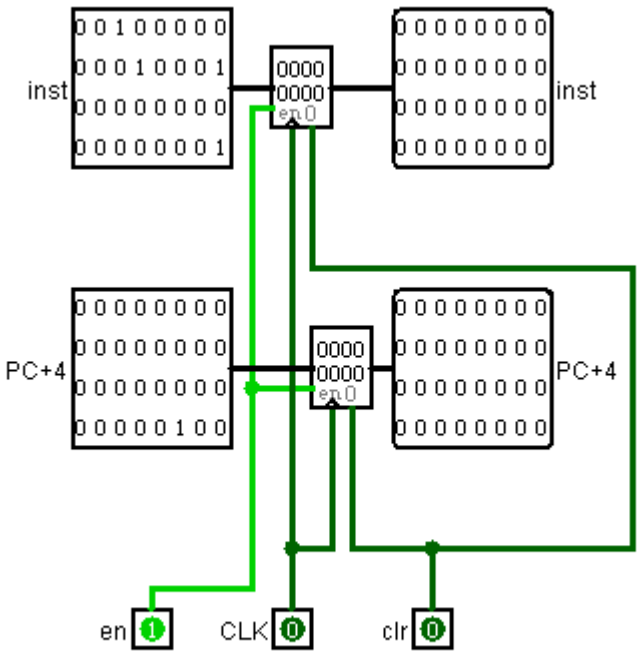

图 3.12 IF/ID

FPGA 实现：四个接口部分的逻辑基本相同，所以仅仅把 IF/ID 的部分主要代码展示一下，采用了在 always 语句中赋值的方法，输出为寄存器类型，不需要显式地实例化一个寄存器。主要代码如下：

```
always @ (posedge clk or posedge reset) begin
    if (reset) begin
        ID_inst = 0;
        ID_PCplus4 = 0;
    end else begin
        if (en) begin
            ID_inst = IF_inst;
            ID_PCplus4 = IF_PCplus4;
        end
    end
end
```

3.3.2 理想流水线实现

不用考虑分支跳转数据冒险等问题，只需要把一条指令的执行分为 5 个阶段：IF、ID、EX、MEM 和 WB；IF 阶段从指令存储器中取出一条指令，并把 PC 加 4，准备取下一条指令；ID 阶段把 IF 阶段取出的指令进行译码，得到后续阶段所需要的全部控制信号和源操作数等信息（包括从寄存器组中取出所需要的寄存器的值）；EX 阶段使用 ALU 进行运算，源操作数和控制信号都由 ID 阶段传过来，得出运算结果后传递到下一阶段；MEM 阶段根据 ID 阶段传过来的信号进行访存，选择把运算结果直接传递给下一阶段或者依据运算结果取出对应内存单元中的值传递给下一阶段；WB 阶段把 MEM 阶段传过来的结果写回到指定编号的寄存器中。

Logisim 实现：把单周期的电路强行分割，再分别把五部分与四个接口部件连接起来，其中 PC 的值直接取的原值加 4，且是在 IF 阶段就更新的，这样才可以实现每个时钟周期都取出一条指令，充分利用流水线。跳转分支这些不需要考虑，直接空着就可以，因为理想流水线不会出现这些指令。

3.4 数据转发流水线实现

根据前面的概要设计规则，可以按照下面的详细规则完成数据转发。

需要进行转发的数据：前一条指令在 EX/MEM 寄存器中存放的 alu 运算结果是一级转发，前前条指令在写回之前要写入寄存器的数据是二级转发。

转发条件：（1）前一条指令需要写寄存器，当前指令的源操作寄存器编号与前一条指令的目标寄存器编号相同，并且前一条指令不是访存指令（访存指令需要插入气泡解决 loaduse 冲突），进行一级转发；（2）前前条指令需要写寄存器，且当前指令的源操作寄存器编号与前前条指令的目标寄存器编号相同，进行二级转发。

Logisim 实现：首先是构建一个检测器，判断是否符合转发的编号相同且前指令不是访存指令的条件，主题就是两个比较器，因为 0 号寄存器恒为 0，不需要进行任何转发，结构如图 3.13 所示；整体转发模块的结构如 图 3.14 所示：

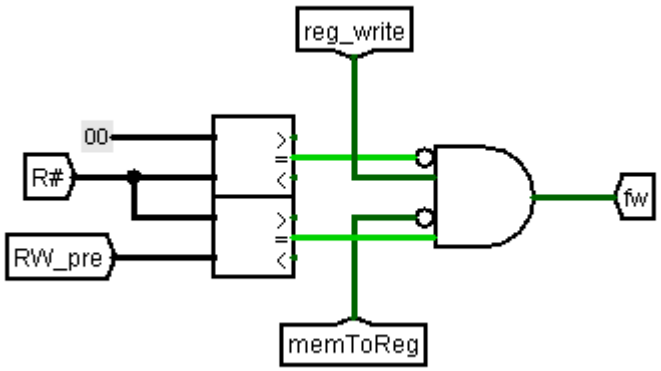

图 3.13 detector

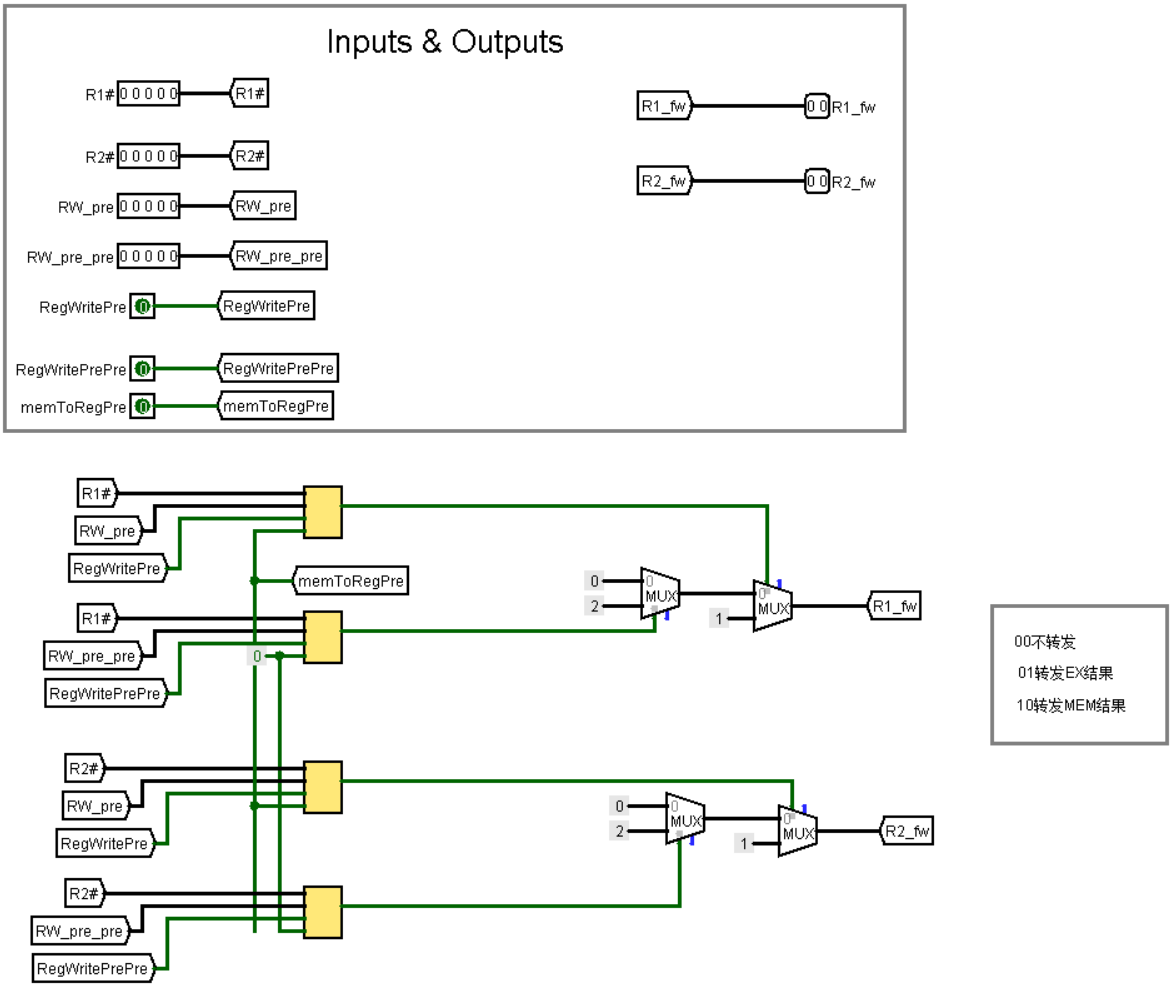

图 3.14 forward

FPGA 实现：根据电路图可以很容易地写出 verilog 代码，先实现 detector，再实例化几个 detector，实现转发控制器，部分主要代码如下：

```
wire ling, sec10, sec11, sec20, sec21;
assign ling = 0;
detector dct10(R1_, RwPre_, wePre, memtoRegPre, sec10);
detector dct11(R1_, RwPrePre_, wePrePre, ling, sec11);
detector dct20(R2_, RwPre_, wePre, memtoRegPre, sec20);
detector dct21(R2_, RwPrePre_, wePrePre, ling, sec21);
assign R1_FW = (sec10) ? 2'b01 : (sec11 ? 2'b10 : 2'b00);
assign R2_FW = (sec20) ? 2'b01 : (sec21 ? 2'b10 : 2'b00);
```

3.5 气泡式流水线实现

气泡流水线在数据转发的基础上插入气泡，解决了 loaduse 冲突，实现方法：当产生 loaduse 冲突时，把 ID/EX 的寄存器清除，PC 寄存器和 IF/ID 寄存器的使能设为无效，就在当前指令与前一条指令之间插入了一个气泡。

Logisim 实现：与数据转发类似，先检测是否产生了 loaduse 冲突，产生冲突需要输出清零信号和 IF 与 ID 的使能信号，阻止流水线的继续传递，具体的模块电路如图 3.15 所示：

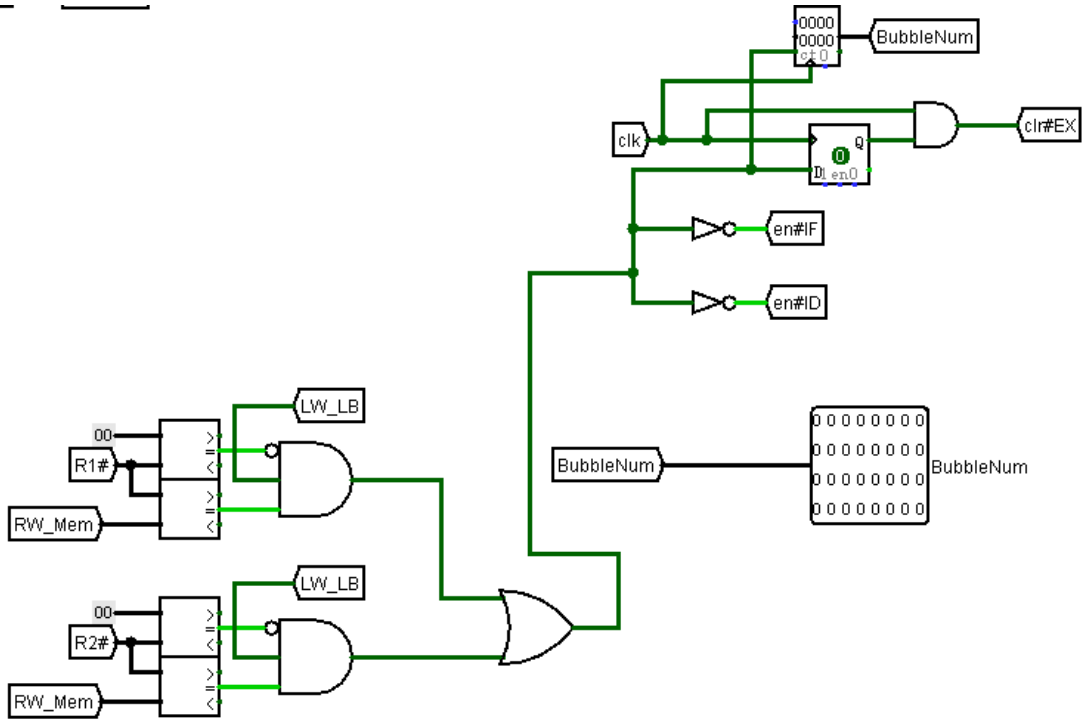

图 3.15 bubble

FPGA 实现：根据 Logisim 的电路图，使用 always 和 assign 语句实现插气泡的模块，主要的部分代码如下：

```
always @ (posedge clk) begin
    if (!reset) begin
        if (RRblb) begin
            count = count+1;
        end
    end else begin
        count = 0;
    end
end

assign EX_clr = (clk & clr_reg);
assign BubbleNum = (count);
assign IF_en = reset ? 1 : (!RRblb);
assign ID_en = reset ? 1 : (!RRblb);
```

3.6 分支和跳转流水线设计

分支和跳转都是在 EX 阶段完成的，判断除当前是否需要跳转（或分支），把跳转（分支）地址写入到 PC 寄存器，再把之前误取的两条指令清除，为了阻止流水线的继续流动产生错误 还要把 IF 和 ID 寄存器段的使能置为无效。

Logisim 实现：如果在 EX 阶段根据 j/jal/jr/beq/bne/blez 信号和 alu 的运算结果判断出需要跳转（分支），就使对应的控制信号为有效，将 EX 段对应的 JumpAddr，BranchResult 或 JrAddr 写入到 PC 寄存器，判断部分电路如 图 3.16 所示：

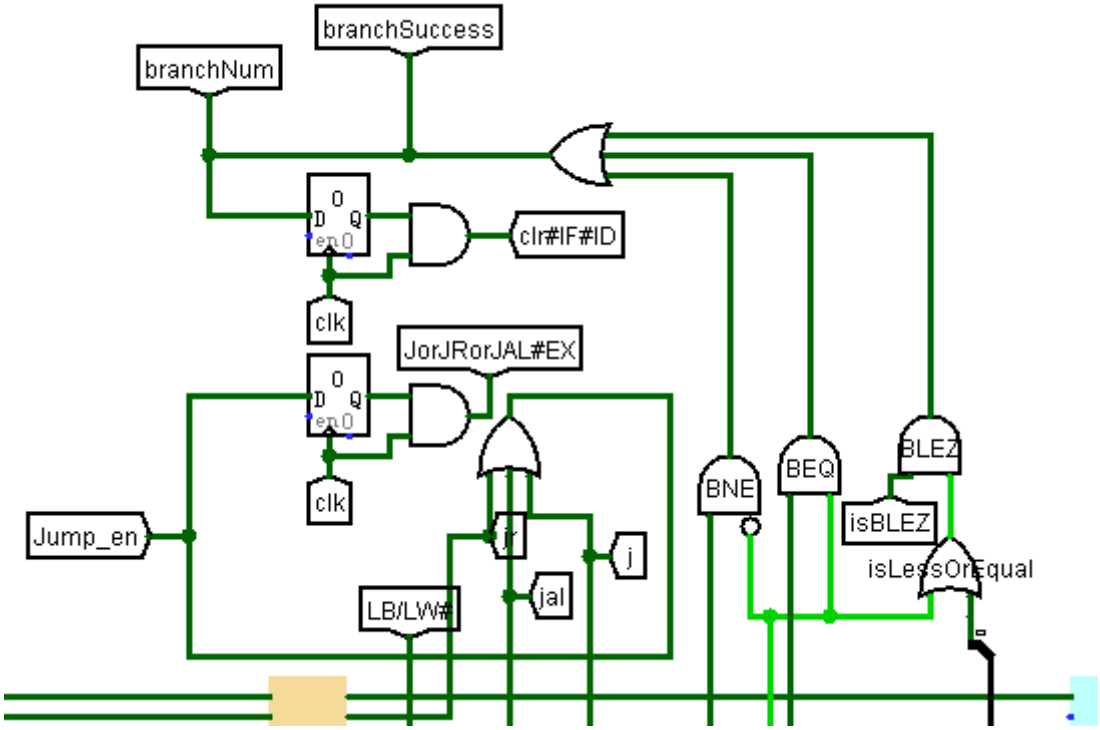
branchSuccess
branchNum
clr#IF#ID
clk
JorJRorJAL#EX
clk
Jump_en
LB/LW#
jr
jal
j
BNE
BEQ
BLEZ
isBLEZ
isLessOrEqual

图 3.16 分支跳转判断

4 实验过程与调试

4.1 测试用例和功能测试

分别对单周期 CPU 进行 benchmark 测试、扩展指令测试和中断测试，对流水线进行 benchmark 测试和扩展指令测试，用例如下：

4.1.1 单周期 benchmark

对单周期 CPU 进行 benchmark 测试，测试过程：在 Mars 中把 asm 文件导出为十六进制镜像文件，加载到 ROM 中，启动系统，运行结果无误，包括了排序走马灯移位等多种功能，最后的指令条数如图 4.1 所示，内存结果如 图 4.3 所示，某个过程中的数码管显示如图 4.2 所示，程序在 Mars 中的运行后的内存结果如图 4.4 所示：

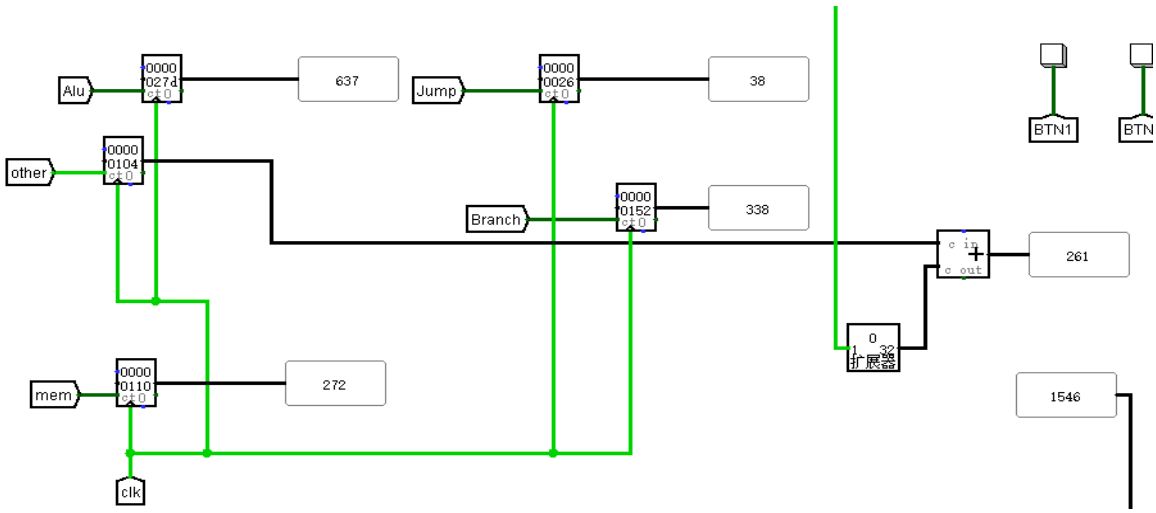

图 4.1 指令条数

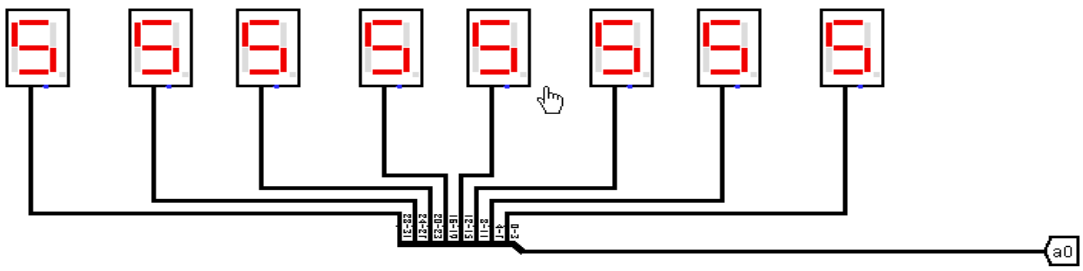

图 4.2 数码管

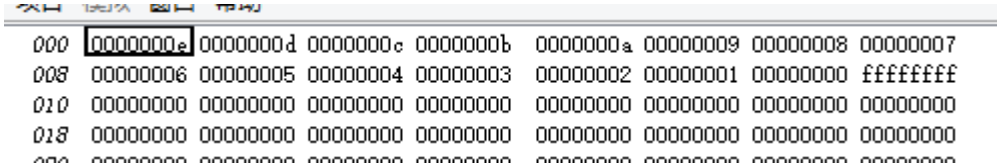

图 4.3 内存（Logisim）

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x0000000e	0x0000000d	0x0000000c	0x0000000b	0x0000000a	0x00000009	0x00000008	0x00000007
0x00000020	0x00000006	0x00000005	0x00000004	0x00000003	0x00000002	0x00000001	0x00000000	0xffffffff
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

图 4.4 内存（Mars）

4.1.2 单周期扩展指令

四条扩展指令分别是 LB，BLEZ，XOR 和 SLTIU，因为 benchmark 测试的完整性很高，所以不方便把这几条与其他的一起测试，就单独测试了，自己编写使用了上述四条指令的汇编代码，分别在 Mars 和 Logisim 中运行，查看最后的结果。在 Mars 中的结果如图 4.5 所示，在 Logisim 中的结果如 图 4.6 所示：

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00007a98	0x00000034	0xffffff98	0x0000007a	0x00000000	0x00000000	0x00000034	0x00000000
0x00000020	0x00000000	0x00000000	0x0000005e	0x00000001	0x00000000	0x00000000	0x00000000	0x0000006a
0x00000040	0x0000006a	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

图 4.5 内存（Mars）

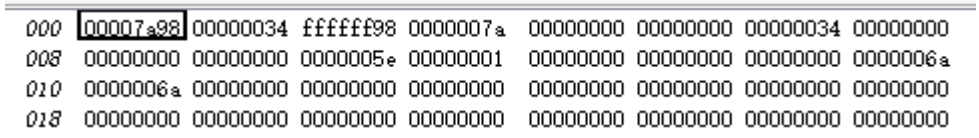

图 4.6 内存（Logisim）

4.1.3 单周期中断

在进行 benchmark 测试的时候，以不同的方式按下按钮，可以的到不同的结果，具体为：在按下按钮 1 后可以再按按钮 2 和按钮 3，按下按钮 2 后可以再按按钮 3，按下按钮 3 后不能再接收其他按钮的信号，所有中断执行完毕后课程返回主

程序继续正常执行，符合中断优先级 1<2<3 的设计，并且实现了中断的嵌套。

4.1.4 流水线 benchmark

对流水线进行 benchmark 测试，观察数码管的显示与之前是相同的，内存如 所示，也与之前单周期相同，时钟周期数和气泡数、分支跳转数等如 图 4.7 所示，与预期结果相同。（误取深度为 2，所以时钟周期数计算方法为：1546+4+(276+38)*2+120=2298）

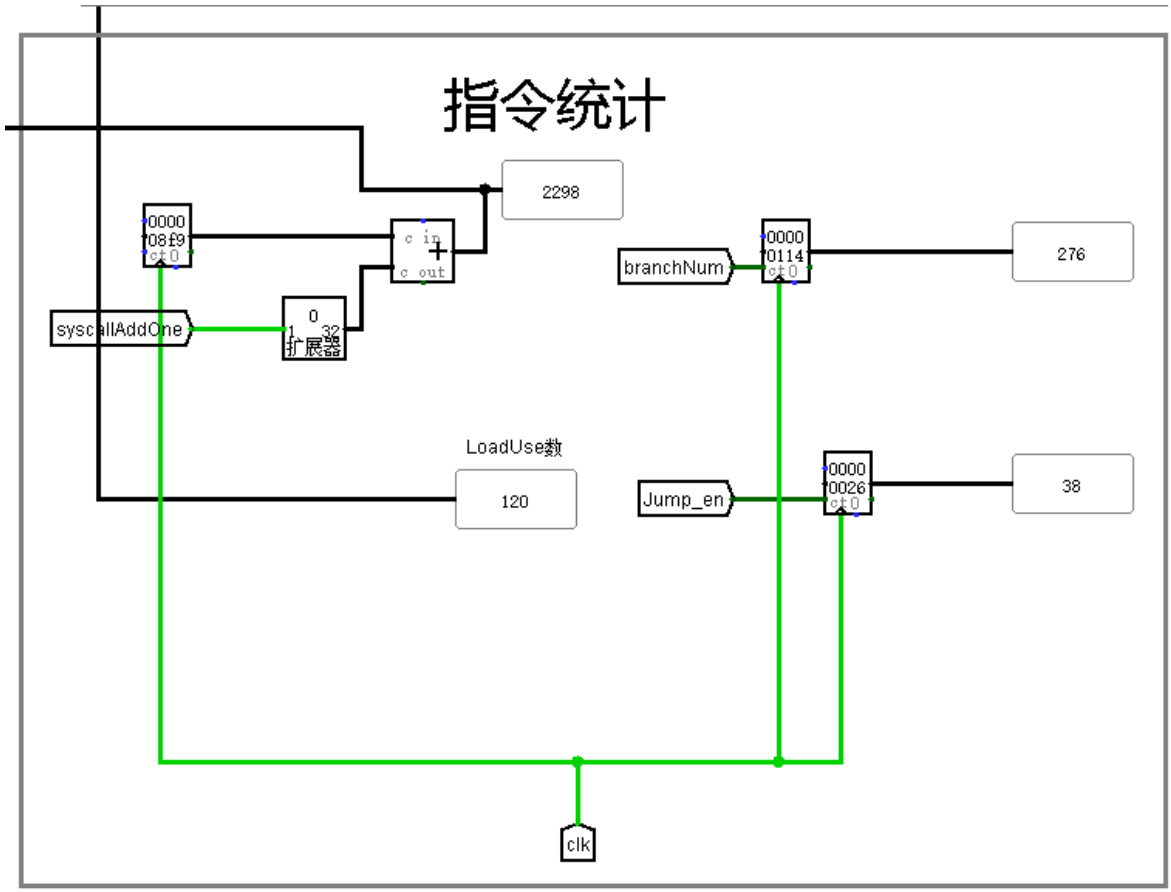

图 4.7 时钟与指令统计

4.1.5 流水线 verilog 仿真

把写好的流水线 CPU 的 verilog 代码进行仿真，仿真结果与在 Logisim 中的运行结果相同，如 图 4.8 所示：

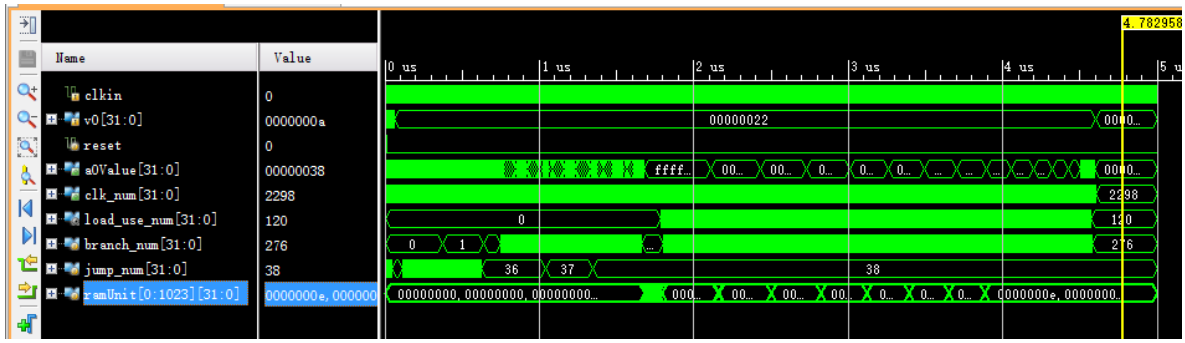

图 4.8 vivado 仿真

4.2 性能分析

不同方案时钟周期数差异：单周期中，时钟周期数就是指令的条数，因为单周期是严格按照每条指令一个时钟周期完成的，都是 1546（benchmark 测试），而在五段流水线测试中，时钟周期数与指令条数之间存在较大的差别，这是因为实际流水线需要处理的事情很多，包括填满流水线（4 个周期）、LoadUse 冲突时插入气泡、分支与跳转时的误取（深度为 2），所以流水线的时钟周期数为上述几个项目与指令条数的和。虽然五段流水线需要更多的时钟周期，但是流水线中每一段的任务也都变得更轻松了，可以在更短的时间内完成，使用更快的时钟频率，所以从实际意义上来说，流水线增加了指令执行的总周期数，但提高了 CPU 的效率。

4.3 主要故障与调试

4.3.1 BLEZ 跳转错误

BLEZ 指令跳转出现错误。

故障现象：BLEZ 指令跳转有问题，即使是在不满足条件时也会跳转。

原因分析：从最开始取到指令开始，跟踪 BLEZ 指令的执行路径，发现是在控制单元中出现了问题，alu_src 信号错误地与 blez 信号线相连，导致运算器的输入操作数不正确，发生了错误跳转。

解决方法：把错连的线删除掉。

4.3.2 addi 指令执行不正确

有时候在执行 addi 的指令时，Logisim 与 Mars 的执行结果不同，但有时又相同。

故障现象：在执行 addi 的指令时，Logisim 与 Mars 的执行结果不同，但有时又相同，产生了不确定性。

原因分析：查看 Mars 中的指令代码，发现在遇到过大的立即数时，Mars 会把一条 addi 的指令分为三条，如 图 4.9 所示，但是分出来的 lui 指令是没有在 logisim 中实现的，所以 logisim 的执行会发生错误，与 Mars 不同。

Code	Basic	Source
0x3c01007a	lui $1,0x0000007a	1: addi $s2,$s0,0x7a9888
0x34219888	ori $1,$1,0x00009888	
0x02019020	add $18,$16,$1	

图 4.9 Mar 对 addi 的分割

解决方法：在使用 addi 指令时，不要使用超过 0x8000 的立即数。

4.3.3 sw 指令出错

故障现象：做了一部分中断工作后，检查之前的 benchmark，发现排序功能错了，程序结束后内存中的数据是乱的，指令条数统计中总指令条数和 memory 类型指令条数错了。

原因分析：分析了一下指令统计中差的数量，发现是 sw 指令改错了，每条 sw 指令占用了 8 个时钟周期，但是与之前对比也没有发现错误在哪里。

解决方法：恢复到提交到 git 的多个版本，依次测试，发现是在一次调整布线的提交后出现了问题，因为连线太过复杂，仍旧没有找到具体的错误出现在哪里，就恢复到了最后一次没有问题的版本，把之后的中断相关的工作再加进来。

4.3.4 中断返回故障

添加中断后中断不能正常返回。

故障现象：嵌套的中断返回不能正常返回，可以返回前一级中断，但返回主程序时会重新执行主程序。

原因分析：进入中断服务程序后，sp 增加了特定的值作为新的栈使用，但在结束

中断时没有修改回去，所以第二次返回时没有读取到保存在栈中的数据，也不能恢复正确的 EPC，恢复到的是 0，从头开始执行。

解决方案：完善汇编指令，返回前把 sp 的值修改回去。

4.3.5 branch 和排序出错

故障现象：benchmark 的执行周期数和跑马灯和 load-use 数等都是正确的，但是排序出错，branch 的周期数错误。

原因分析：原因 1：数据转发的逻辑问题，如果有两条 LW 指令相邻，下面有一条与前条 LW 指令冲突的指令，没有考虑到后一条 LW 指令的 memToReg 信号对转发的影响，直接停止了转发，导致没有取到正确的内容；原因 2：插入气泡问题，检测的数据跟转发时一样，是当前指令的 EX 周期和前一条指令的 MEM 周期，但是这样操作会导致插入气泡的位置不正确，因为 LW 后面的指令已经在执行了。

解决方法：（1）把二级转发检测的 memToReg 输入直接连为 0；（2）检测前移一个周期，也就是在当前指令的 ID 周期和前一条指令（LW）的 EX 周期进行检测，这样才能在 LW 后面插入一个气泡。

4.3.6 没有解析到 LoadUse 冲突

故障现象：排序功能发生错误，并且指令的周期数不对（没有解析到 loaduse 冲突，不会插入气泡）

原因分析：跟踪仿真中的 lw 执行的路径发现 memtoreg 信号一直为 0 才发现的，再由控制单元找到了原因，是 decoder 模块的错误（100011 错写为了 100111）

解决方法：把正确的译码改回去。

4.3.7 转发模块不完整

故障现象：转发模块出现了转发 11 的结果，但合法的结果只有 00，01 和 10。

原因分析：转发模块不完善，仅仅是用分线器把两个转发信号拼接起来的，没有考虑到前两条指令都写本条指令要读的寄存器的情况，导致错误，如 图 4.10 所示。

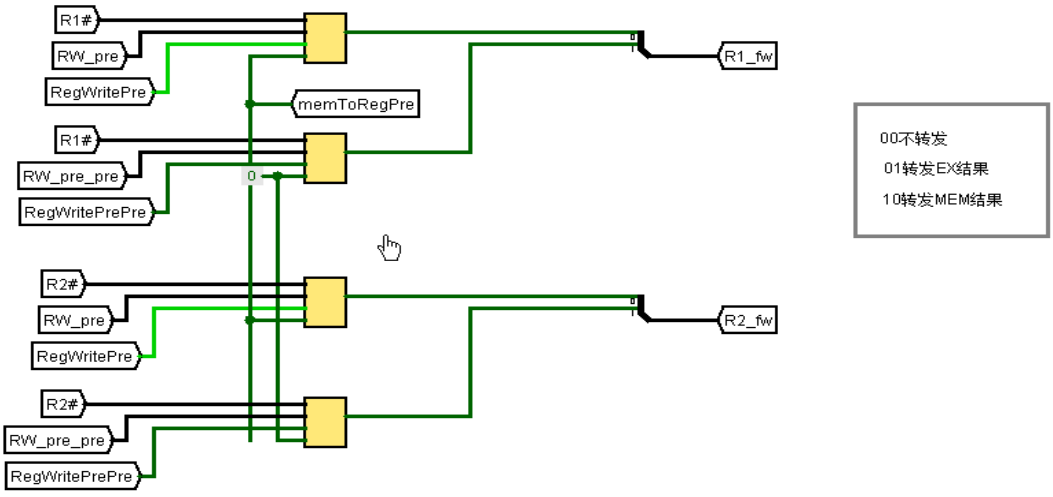

图 4.10 不完整的转发

解决方法：用两个二路选择器用两个信号做选择，选 0，1 或 2。后来又想到了一个更简单的方法，在转发的四路选择器那里把 3 号连上上一条指令的结果。

4.4 实验进度

表 4.1 课程设计进度表

时间	进度
第一天	阅读任务书，扩充部分指令，暂未测试
第二天	完成并测试通过了四条扩充指令，中断有了一点思路
第三天	添加中断的过程中，某个或某些操作把之前的部分指令弄错了（排序错误和 memory 类型指令条数及总指令条数统计错误），正在回退版本并修改，等待修正后再继续中断，目前中断的完成度还比较低，尚未完成单级中断。
第四天	修复了昨天的原有指令的问题，完成了单级中断，多级中断还没有完成
第五天	完成了多级中断的嵌套和优先级设置，开始进行转流水线的工作
第六天	完成了理想流水线并测试通过，正在增加数据重定向功能
第七天	完成了流水线的数据转发和分支功能，正在考虑跳转的实现
第八天	完成了五段流水线的全部功能，开始写 verilog，刚写了 1 个模块
第九天	实现了多个 verilog 的模块，开始进行主电路的编写
第十天	完成了整个 verilog 的仿真测试

5 设计总结与心得

5.1 课设总结

五段流水线 CPU 是对单周期 CPU 的改进，通过把指令不同的功能分割成不同的阶段，实现了多条指令执行在时间上的重叠，为了完成五段流水线的 CPU，作了如下几点工作：

1） 完成方案总结：完成单周期的 CPU；在单周期 CPU 的基础上添加多级嵌套中断支持；在单周期 CPU 基础上完成理想流水线、数据转发流水线、插气泡流水线。

2） 功能总结：单周期 CPU 支持以下功能：27 条运算指令；多级嵌套中断，指令条数统计。

五段流水线 CPU 支持以下功能：27 条运算指令，处理多种冒险，数据转发和插入气泡，程序执行周期和指令统计。

3） 不足之处：线连得不是很规整，模块化程度不够高，整体看上去有点混乱；FPGA 没有完成，仅仅完成了 verilog 的仿真，上板子没有调试成功。

5.2 课设心得

从经过的这三年来看，这次课设可以说是最不水，要求最高，难度最大的一次了，经过了整整两个星期外加周末的时间才完成到了 verilog 仿真的阶段，没有最终完成 FPGA 的工作，回想这两个星期，除了遇到 bug 调不好的抓狂，还有由衷的充实感，从假期的放松，直接到做课设的紧张，对每一小时每一分钟的抓紧，是之前很少体会过的，最后还有完成某一个功能或模块时的欣喜，虽然没有完成全部的任务，但还是有一些成就感。

第一个任务是扩展上学期的课程实验，新增的几条指令的实现方法与上学期的 24 条指令大同小异，所以整个过程比较迅速，很快就扩展完成并测试通过。

第二个任务是添加中断支持，一开始把中断想的太过简单，没有想好实现方法就开始画图了，结果可想而知，处处碰壁，最后拖了很长时间，认真思考实现中断的方法，不断跟同学讨论才磕磕绊绊完成了中断，这是整个课设中遇到的第一个大的挑战

了。

然后是把单周期 CPU 改造成五段流水线 CPU。首先是改造理想流水线，这个过程也比较简单，直接把未添加中断的单周期 CPU 大卸五块，用四个流水线接口把对应的段连接起来；然后逐步在理想流水线的基础上添加对非理想情况的支持，一是数据的转发，两级转发就可以完成，第三级可以通过对寄存器的前半周期写，后半周期读来完成（虽然原理上来说很简单，但实现的时候还是遇到了很多问题，包括转发条件的判断，转发来源，转发的选择信号等，一不小心就是错误），而是 loaduse 冲突查气泡，三是分支和跳转指令的实现，在这里也遇到了问题，其中有一个就是毛刺的问题，很诡异，问了其他同学，有些同学有毛刺，有些同学没有毛刺，解决办法是添加一个寄存器来消除毛刺。经过了抓狂的调 bug 阶段，总算是把五段流水 CPU 的 Logisim 版完成了，真的是调 bug 调到抓狂，要对照 logisim 和 Mars 的执行过程，一步一步查看是哪里的问题，不小心过头了又要重来。

最后是 FPGA，这一步没有最终完成，vivado 的仿真已经通过，但上板子就遇到了很多问题，最终也没有调好，回想很长时间的对 verilog 代码的编写调试，真的是感觉非常遗憾，在 vivado 里调仿真是比 logisim 要容易的，因为不再需要一步一步观察程序的执行，可以直接以时间轴的方式 查看整个执行过程，不管怎样，最后把仿真调好了还是略感欣慰的。

通过这次课程设计，自己的收获确实是很大的，提高了自己调试程序的耐心，对挫败的承受能力，而且在 verilog 阶段，需要编写大量的代码，提高了编程能力和对功能模块化的重要性的认识。

下面是一点点对分组的看法吧，感觉分组在这里的作用并不是很大，每个人都独立完成自己的任务，确实阻止了一般情况下组内一人工作，其他人打酱油的情况的发生，但组内的同学进度不同，很难经常进行讨论交流，还是要跟其他与自己进度相近或比自己快的同学交流才能得到提高，这样组内进度慢的同学就一点点被边缘化，没有达到原本的分组讨论交流共同提高的目的。

最后感谢老师和同学们对自己的帮助，相信组原课设是自己在大学生活中一段美好而又艰辛的回忆。

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] 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.

[5] 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: 许荣仁