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：	校交 1601
学　号：	U201610270
姓　名：	蒋苡杭
电　话：	17701522656
邮　件：	947002697@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	
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	SRLV	逻辑右移	
29	XOR	异或操作	
30	LBU	无符号字节加载	
31	BGTZ	大于 0 分支跳转	

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硬布线控制，且主、控存分开的方案，实现数据存储器和指令存储器不共用一个存储器的方式完成方案的设计。指令存储器由 PC 寄存器的值获取指令地址，输出 MIPS32 位指令。指令用于控制硬布线信号的产生，寄存器的存取以及为 ALU 提供待运算内容。ALU 的运算结果用于输入数据存储器作为数据地址取出相应位置的数据或者作为待存放入数据存储器的地址。整个单周期 CPU 由时钟统一控制。同时在实施的过程中，采用先将电路用 logisim 方式实现，接着在 logisim 的基础上用 Verilog 语言实现。

总体结构图如图 2.1 所示。

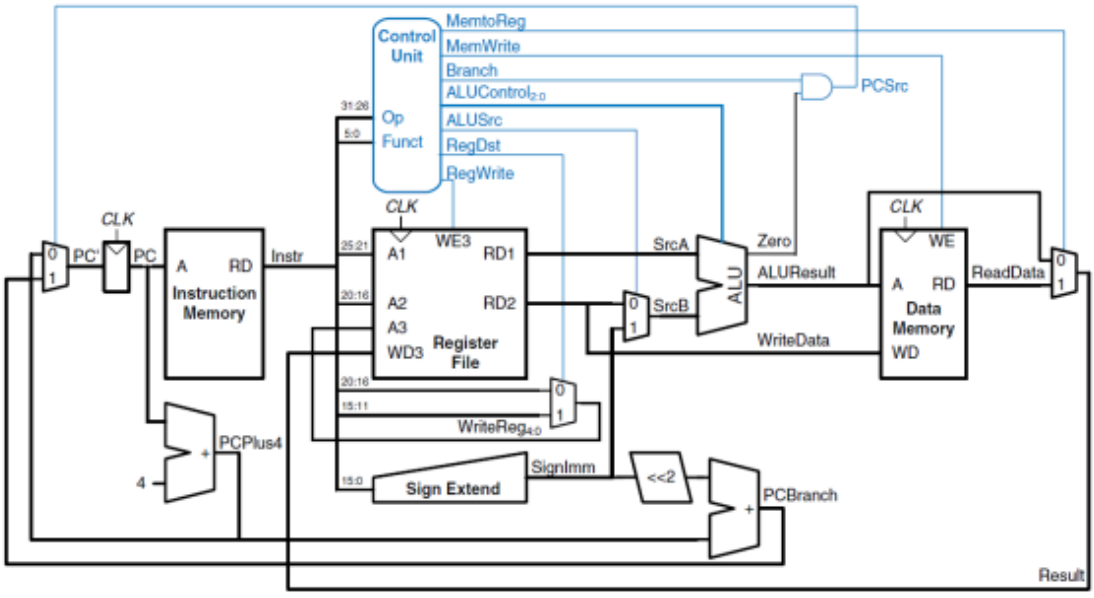

图 2.1 总体结构图

2.1.1 主要功能部件

单周期 CPU 可以划分为以下几个模块：程序计数器、指令存储器、数据存储器、运算器、寄存器堆、单周期控制器，各个模块的设计思想如下：

1. 程序计数器 PC

为了分工工作量较为均等，我们组将程序计数器和分支指令的控制信号包含在程序计数器模块内实现。PC 的输入有四个来源：顺序存取获得的下一条指令、由指令解析出的有条件分支指令的跳转地址、无条件分支指令中 JR 指令对应的寄存器中取出的地址以及无条件分支指令中 jal 指令和 jmp 指令对应的指令直接解析出的地址。PC 的输出将送往指令存储器的地址段。

2. 指令存储器 IM 和数据存储器 DM

指令存储器：在 logisim 中指令存储器的实现是基于组件 ROM，输入为指令地址寄存器 PC 的输出值，输出则为取出指令值。在 verilog 程序中，switch-case 语句实现输入地址值和输出值寄存器的对应。switch-case 语句的生成是通过 python 程序解析.hex 文件自动生成的。

数据存储器：在 logisim 中数据存储器的实现是基于组件 RAM，输入为 ALU 计算得到的 2-21 位数据，作为取数据的地址。输出为数据存储器的内容，信号 MemWrite 有效时需要向数据存储器输入数据，该数据的来源是 R2 寄存器的输出。当 MemToReg 信号有效时，数据存储器可以出去，同时多路选择器的选择信号有效，选择数据存储器的内容作为输出，在 verilog 程序中，全部使用寄存器二维数组实现对于数据存储器的模拟。根据存储信号 store 控制数据存入值，reset 信号来临时将二维数组清空，数据输出值根据输入的地址值作为数据索引，获取对应数组的值。

3. 运算器

运算器用于完成操作数的各类运算，包括：逻辑移位，算术移位，无符号除法和乘法，按位与，按位或，按位异或，按位或非以及比较操作。引脚与功能描述如表 2.1 所示。

X 操作数来源于寄存器堆 R1 的输出、Y 操作数来源于 R2 的输出和指令解析而得的立即数，Y 的来源由多路选择器确定，多路选择器的控制信号为硬布线控制器的 AluSrcB 信号。ALU 的输入还包括运算器功能码，运算器功能码如表 2.2 所示和

移位量值，功能码由硬布线控制器生成，移位量值由 SRLV 的多路选择器进行选择。当 SRLV 信号有效时，移位量为 rs 字段的值，否则移位量为指令的 6-10 位的值。ALU 的输出值为运算所得的 Result 以及 X、Y 相等时的 eq 信号。

表 2.1 ALU 引脚与功能描述表

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	操作数移位量
Result	输出	32	ALU 运算结果
Equal	输出	1	Equal=(x==y)?1:0, 对所有操作有效

表 2.2 ALU_OP 运算功能表

ALU_OP	十进制	运算功能
0000	0	Result = X << Y 逻辑左移 （Y 取低五位） Result2=0
0001	1	Result = X >>>Y 算术右移 （Y 取低五位） Result2=0
0010	2	Result = X >> Y 逻辑右移 （Y 取低五位） Result2=0
0011	3	Result = $(X * Y)_{[31:0]}$;
0100	4	Result = X/Y; Result2 = X%Y 无符号除法
0101	5	Result = X + Y (Set OF/UOF)
0110	6	Result = X - Y (Set OF/UOF)
0111	7	Result = X & Y 按位与
1000	8	Result = X | Y 按位或
1001	9	Result = X⊕Y 按位异或
1010	10	Result = ~(X |Y) 按位或非
1011	11	Result = (X < Y) ? 1 : 0 符号比较
1100	12	Result = (X < Y) ? 1 : 0 无符号比较

4. 寄存器堆 RF

32 个寄存器组成的寄存器堆，输入包括：R1 寄存器号和 R2 寄存器号、写入寄存器号、写入寄存器的值，写使能信号，输出包括 R1、R2 寄存器的值。由 RegFile 模块实现。

在 verilog 中，该寄存器堆通过 32 个 32 位二维数组实现。其中特殊的寄存器为 0#寄存器，该寄存器无法写入并且始终保持 0 为其内容。模块的输入为写使能信号，写寄存器号，R1 寄存器号，R2 寄存器号和写入数据值。模块的输出为 R1 和 R2 寄存器内容。R1 和 R2 寄存器内容的输出由 R1 寄存器号，R2 寄存器号作为索引，在二维数组中直接获取。当写使能号和写入寄存器号非 0 时，可以将写入数据值送入相应的写寄存器号作为索引的寄存器。

2.1.2 数据通路的设计

根据单周期数据通路需要的各类功能部件，在数据通路框架中，不同指令需要考虑的数据来源如表 2.3。包括 PC 来源，立即数来源，寄存器堆待取数据的寄存器号，写入寄存器号和写入内容，ALU 的操作数和操作码以及数据存储器的数据地址和写入数据。

表 2.3 指令系统数据通路框架

指令	PC	IM	RF				ALU			DM	
			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 2.4。

表 2.4 主控制器控制信号的作用说明

控制信号	取值	说明
Sys	0	寄存器堆 R1，R2 寄存器号分别读取 rs 和 rt 指示的寄存器号的值
	1	寄存器堆 R1，R2 寄存器号分别读取 2 号和 4 号寄存器的值
JMP	1	无条件跳转有效，送入 PC 的值为 26 位立即数
	0	根据 branch 信号判断跳转目标
RegDst	1	写寄存器编号为 rt
	0	写寄存器编号为 rd
Branch	1	有条件跳转成功，送入 PC 的值为左移两位后的 16 位立即数
	0	送入 PC 的值为 PC+4
AluSrcB	1	ALU 操作数 Y 的选择为立即数扩展的结果
	0	选择寄存器堆的 R2 输出
AluOP	0000-1100	控制 ALU 的操作码
MemWrite	1	数据存储器可以写入
	0	数据存储器不可以写入
MemToReg	1	写回寄存器的数据来自于数据存储器
	0	写回寄存器的数据不来自于数据存储器
SignedExt	1	立即数扩展有有符号位实现
	0	立即数扩展由无符号位实现
LBU	1	写入寄存器堆内容为扩展字节
	0	写入寄存器堆和字节无关
SRLV	1	偏移量为寄存器堆的 R1 输出
	0	偏移量来自命令解析
JR	1	JR 指令有效，送入 PC 的值为寄存器堆 R1 内容
	0	根据 JMP 信号和 branch 信号判断跳转目标
JAL	1	JAL 指令有效，送入 PC 的值为 nextPC||instr_index||0^2
	0	根据 JR 信号判断跳转目标

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 2.5 所示。

表 2.5 主控制器控制信号框架

指令	R	RW	WE	X	EXT	Y	ALUop	MemWrite	MemRead	Din	Branch	SYSCALL

2.2 中断机制设计

2.2.1 总体设计

在中断请求到来时，保存断点，也就是当前 PC 的值。接着中断识别 并且跳转，根据中断源找到中断服务入口程序地址，此处将使用直接地址选择的方式进入中断而不是通过中断向量表的形式，地址的获取可以通过 Mars 直接读取。然后利用堆栈保存现场并运行中断服务程序。最后恢复现场和中断返回。注意多级中断还需要考虑中断优先级的问题，只有高优先级中断才可以打断低优先级中断。

2.2.2 硬件设计

1) 新指令的支持

为支持 MTC0，MFC0 和 ERET 指令，需要扩展硬布线控制器，通过操作码和 FUNCT 字段的比较实现指令信号的产生。MTC0、MFC0 指令用于将数据从特殊寄存器向普通寄存器传输。ERET 指令用于返回中断之前的程序。

2) EPC 的保护和设置

1#寄存器用于存放 EPC，当恢复现场或者保存现场时，需要用这两条指令对 1#寄存器进行修改（为了简化起见，本次实验未物理上实现 CP0）。ERET 指令用于返回原有的指令，需要将 EPC 恢复。

3) 中断优先级控制

优先级的控制是通过优先编码器来实现的。当中断同时到来时，优先编码器可以选择优先级较高的编码，结合多路选择器，选出相应的中断程序地址送入 PC 运行。

2.2.3 软件设计

软件主要实现的功能是开关中断和保护现场，恢复现场，通过汇编代码实现对于硬件的帮助。例如，关中断的操作通过 addiu 指令和 mtc0 指令实现修改中断使能寄存器 IE，实现中断功能。因此若需要关中断，则利用 addiu 指令将立即数 1 和 0#寄存器相加，送往指定寄存器，如 k0。紧接着通过 mtc0 指令的信号，将 k0 的值写回 0#寄存器，从而完成对 IE 开关的控制。

而保护现场和恢复现场则通过 SW 指令和 LW 指令实现。通过 SP 堆栈指针寄存器，每保存一个值就将 SP 移动以保存下一个值。待恢复现场时，则根据 SP 的值将对应地址中的数据加载回通用寄存器中。

2.3 流水 CPU 设计

2.3.1 总体设计

流水线的设计目的是将单周期的时钟周期通过分段使得多条指令并行执行从而缩短整个程序的运行时间。将单周期 CPU 的数据通路分为 5 段：取指令段 IF，译码指令阶段 ID，执行指令阶段 EX，访存阶段 MEM 以及写回阶段 WB。为实现数据通路的划分，需要加入流水接口部件，各个部件之间就是相应的流水加工段，每个流水加工段接收上一个流水加工段输出的数据（存放于流水接口部件的寄存器中），接着向下一个流水接口部件输出待处理的数据（存放于下一个流水接口部件的寄存器中）。

2.3.2 流水接口部件设计

流水接口部件的主要功能是缓存上一个流水加工段的输出数据，为下一个流水加工段提供更新后的数据。因此流水接口部件的主要组成部分应该是寄存器组，每个寄存器保存需要传递的信号，每个流水接口部件的具体寄存器设置见流水线设计部分。

除了寄存器，该部件需要以下信号来确保信号的同步及冲突问题的解决：

(1) 统一的时钟同步信号；

(2) 清零信号：该信号在理想流水线中是同步清零信号，但是在进行分支相关问题的处理时不同的流水接口部件需要设置不同的清零信号以确保分支指令的运行正

确性。

(3) 使能信号：和清零信号一样，在理想流水线中使能信号由 Syscall 信号统一决定，当 syscall 信号无效时，使能信号统一有效。在气泡流水线和重定向流水线中，使能信号收到数据相关和 load-use 相关影响，需要加入另外的控制信号。

2.3.3 理想流水线设计

理想流水线各段需要完成的工作如下：

IF 段：获取指令地址和指令内容，通过流水接口部件 IF/ID 送往 ID 段。地址内容和指令内容在写回段仍然需要用到需要回传至寄存器堆解析写入寄存器号，因此需要通过每一个流水寄存器部件。

ID 段：解析指令，生成相应控制信号，将控制信号和寄存器所取数据 R1 和 R2 通过 ID/EX 流水接口部件送往 EX 段。

EX 段：计算 ALU 的输出结果，将控制信号和 ALU 运算结果通过 EX/MEM 接口部件送往 MEM 段，其中控制信号 ALU_OP 仅作用于 ALU 上，ALU_SRC 仅作用于 ALU 第二个操作数的选择上，因此无需后传。

MEM 段：获取数据存储器内容或者直接由 ALU 的计算内容传入 WB 段。另外将控制信号和 MEM/WB 接口部件送往 WB 段。其中控制信号 MEM_Write 仅作用于 MEM 段的数据存储器控制内存写功能，无需后传。

WB 段：写回段的操作和单周期中类似，为寄存器堆提供写使能信号，写回指定寄存器号的内容。

2.4 气泡式流水线设计

气泡流水线解决的问题是分支相关和数据相关的问题。分支相关问题产生的原因是当指令执行到 EX 段时，由于产生分支跳转，之前两个顺序取出流水段的指令即作废，需要插入气泡消除。数据相关问题产生的原因是一条指令在译码段取的寄存器堆中的数据由于流水段的延迟，并不能取到最新的数据，需要暂停并且插入气泡。

2.4.1 分支相关问题解决

当无条件分支信号或者有条件分支信号有效时，将 IF/ID 和 ID/EX 段清空，相

当于插入两个气泡，从而实现将作废指令清空。

2.4.2 数据相关问题解决

译码段和写回段的数据冲突解决：通过将存储器时钟控制端改为下跳沿有效，这样译码段需要数据时，写回段已经将数据传入寄存器。

访存段和译码段的数据冲突解决：当出现此种数据冲突时，需要将 PC 的使能端暂停，IF/ID 段接口部件同样暂停，在 ID/EX 接口部件处使得清零信号有效，这样当下一个时钟周期来临时访存段的指令就到了写回段，访存段是一条空指令。回到了译码段和写回段冲突的情况。

译码段和执行段相关：处理方式与访存段和译码段冲突类似，经过 PC 使能端和 IF/ID 接口部件的暂停后，回到访存段和译码段的数据冲突的情况。

接口部件使能和清零信号和数据相关信号的生成会在 3.4 的气泡流水线实现中说明。

2.5 重定向流水线设计

2.5.1 重定向总体设计

重定向在解决分支相关问题的基础上解决数据相关问题，该方案和气泡式流水线相比，可以减少插入气泡带来的周期数的损失，提升流水线的效率。

重定向根据执行段的信号，判断执行段的读寄存器操作是否需要用到访存阶段和写回阶段的写寄存器操作，如果需要就存在着数据冲突问题。

重定向流水线中还有一个重要的待解决问题是如果前一条指令是访存指令时出现数据相关不可以进行重定向而需要进行一次插入气泡的操作，这是因为访存指令的速度较慢，则执行段必定要等待很长时间才能获取重定向的数据，降低了流水线的效率。

2.5.2 定向操作设计

重定向 ALU 的 XY 操作数在 EX 段有四种中可能的来源：MEM 段的 ALU 运算结果，写回段的 ALU 运算结果和数据存储器取出的内容，以及初始状态下从 ID 段传过来的 XY 操作数。因此 Fwd 选择信号需要两位。

Fwd 信号的具体取值与气泡流水线中的数据相关信号生成有相似之处。在 3.5 的重定向流水线实现中说明。

2.5.3 loaduse 问题解决设计

loaduse 信号的产生和气泡流水线中的数据相关类似，可以和 Fwd 信号一起在信号生成模块中生成。当 loaduse 信号有效时，需要将 PC 的使能端暂停，IF/ID 段接口部件同样暂停，在 ID/EX 接口部件处使得清零信号有效，这样当下一个时钟信号来临时，访存指令就会进入写回段，这时才会进行重定向操作。

2.6 动态分支预测设计

2.6.1 总体设计

动态分支预测可以使得在取指令阶段以 PC 的值为关键字查询 BHT，根据历史统计信息直接预测下一条指令的地址，提升预测正确率后，避免分支指令的清空或者暂停现象，从而提升流水线的性能。

流水线进行预测时会出现以下几种情况：以 PC 为关键字进行全相连比较，如果命中且预测成功，则直接跳转，继续运行。如果命中但是预测错误则需要插入气泡以清空错误导致的误取指令，如果没有命中，则流水线的取指令照常进行。

2.6.2 BHT 设计

BHT 为预测分支历史表，用于存放预测历史，并且根据预测历史做出相应的预测，并且能够自主更新，该功能部件是动态分支预测实现的基础。

BHT 包括以下几个数据项：有效位，用于判断当前分支指令地址能否被用于预测；分支指令地址，用于全相联的查找；分支目标地址，存放历史跳转地址，也就是下一次命中时的预测内容；LRU 置换标记，用于置换时选择被替换的那一位。

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。Run 为运行信号，Go 是继续运行信号，Sys 是系统调用信号，通过或门之后成为 Run 信号，Run 信号为 PC 的使能信号。当需要进行停机时，Run 控制信号为 0，使整个电路停机。

PC 计数器模块还包括了输入信号，由条件分支信号 branch，无条件分支信号 JMP 和无条件分支信号中的 JR 信号构成，如图 3.1 所示，对应的 PC 值可以按照表 2.4 对照 branch，JMP，JR，JAL 的功能推出。

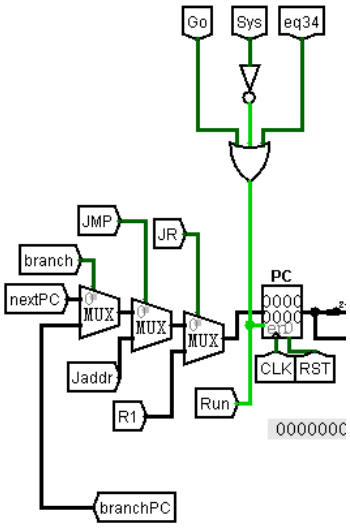

图 3.1 程序计数器

② FPGA 实现：

程序计数器 PC 的 Verilog 代码如下：

```
always @(posedge clock) begin      //clk,rst,run
    if(reset) begin
        PC <= 32'h00003000;
```

```
        end
        else if (run && ~reset) begin
            PC <= JR_MUXout;
        end
    end
```

选择 PC 寄存器值的多路选择器的 Verilog 代码如下：

```
    always @(sel or a or b)
    begin
        case (sel)
            1'b0 :   out1 = a;
            1'b1 :   out1 = b;
        endcase
    end
```

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。

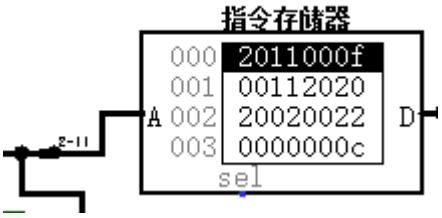

图 3.2 指令存储器

② FPGA 实现：

由于指令存储器的实现不需要运行时修改，因此可以使用组合逻辑实现 ROM。编写 python 程序读取.hex 文件并且自动生成 rom.v 文件。

生成的指令存储器的 Verilog 代码如下：

```
module Rom #( ADDRESS_WIDTH = 32, DATA_WIDTH = 32)(
        input wire[ADDRESS_WIDTH - 1: 0] address,
        output reg[DATA_WIDTH - 1: 0] result
    );
    always @(address) begin
        case (address & 32'h00003fff)
        32'h00003000: result = 32'h20110001;
        …
    end
```

3) 数据存储器（DM）

① Logism 实现：

使用一个 MIPS RAM 实现数据存储器（DM）。设置该只读存储器的地址位宽为 20 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 20 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 2-21 位作为数据存储器的输入地址。如图 3.3 所示。

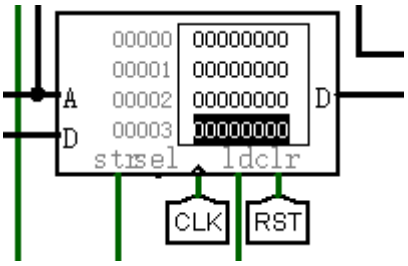

图 3.3 数据存储器

② FPGA 实现：

数据存储器的 Verilog 代码如下：

```
    always @(posedge clk) begin
        if (store) begin
            ram[address] <= dataIn;
        end
        if(reset) begin
            for(counter = 0; counter < RAM_SIZE; counter = counter + 1) begin
                ram[counter] <= 0;
```

```
            end
        end
    end
```

4) 寄存器堆（RF）

① Logism 实现：

使用一个 Register file 实现寄存器堆（RF）。设置寄存器编号值为 5 位，写入数据 Din 和输出的 R1，R2 为 32 位。输入部分的选择见 3.1.2 数据通路实现部分。logisim 实现如图 3.4 所示。

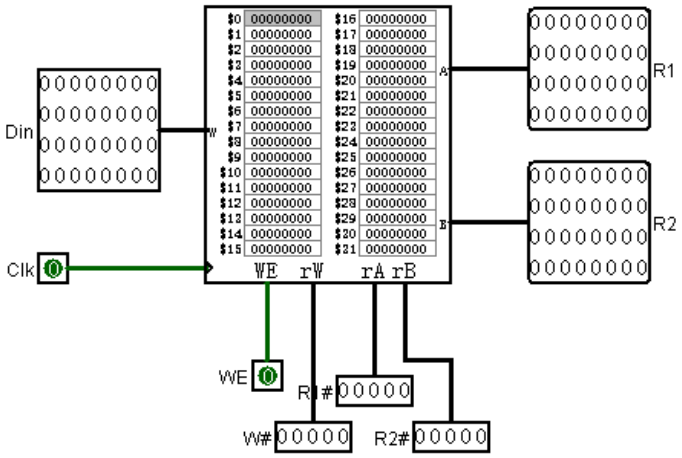

图 3.4 寄存器堆

② FPGA 实现：

寄存器堆的 Verilog 代码如下：

```
    reg[31: 0] regfile[31: 0];


    always @ (negedge clk) begin        //posedge
        if (writeEnable && regWrite != 0) begin
            regfile[regWrite] <= writeData;
        end
    end
    assign resultA = (regA == 0) ? 0 : regfile[regA];
    assign resultB = (regB == 0) ? 0 : regfile[regB];
```

5) 运算器（ALU）

① Logism 实现：

使用 MIPS ALU 模块实现运算器（ALU）。输入引脚为操作数 XY，4 位操作码，5 位移位量，输出引脚为 Result，equal 信号。logisim 实现如图 3.5 所示。

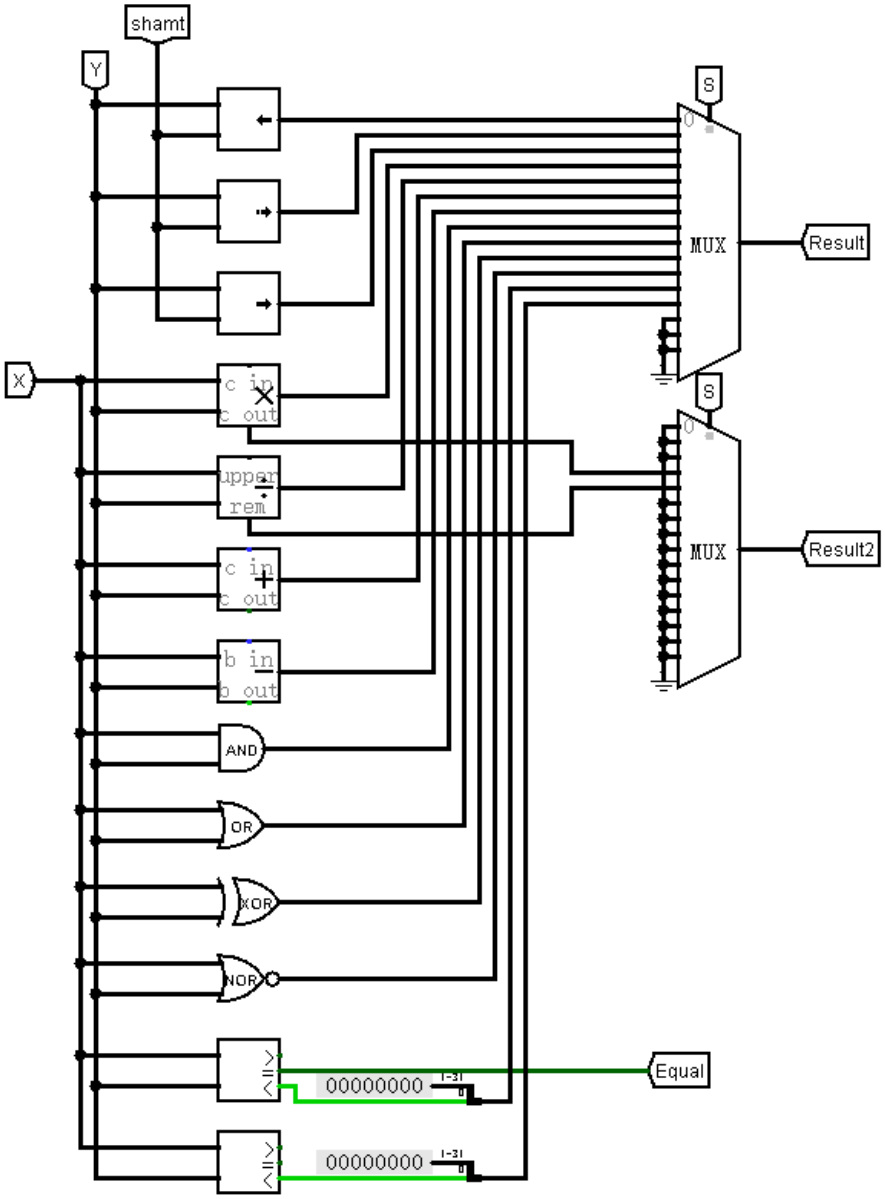

图 3.5 ALU 内部实现图

② FPGA 实现：

运算器的 Verilog 代码如下：

```
assign equal = (x == y);


always @(x, y, shamt, alu_op) begin
    case(alu_op)
```

```
                4'd0: begin
                        result1 = y << shamt;
                        end
                .......
            endcase
    end
```

equal 信号由输入操作数决定，result 由 case 语句赋值，通过 alu_op 选择，由于运算规则太多，代码中不一一列举，各个操作码对应的功能请见表 2.2。

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 数据通路信号表

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			
J	(PC+4)||instr_index||0^2	PC							2			
JR	r1	PC	rs						0			
JAL	(PC+4)||instr_index||0^2	PC			31	PC+4			3			
BEQ	(PC+4)+(IM<<2)	PC	rs	rt			r1	r2	4			
BNE	(PC+4)+(IM<<2)	PC	rs	rt			r1	r2	5			
SLTI	PC+4	PC	rs		rt	alu	r1	立即数	10			
ORI	PC+4	PC	rs		rt	alu	r1	立即数	13			
LW	PC+4	PC	rs		rt	DM	r1	立即数	35	alu		
SW	PC+4	PC	rs	rt			r1	立即数	43	alu	r2	
SYSCALL	PC+4	PC	2	4			r1	10	0			
SRLV	PC+4	PC		rt	rd	alu		r2	0			
XOR	PC+4	PC	rs	rt	rd	alu	r1	r2	0			
LBU	PC+4	PC	rs		rt	DM(Byte)	r1	立即数	36	alu		
BGTZ	(PC+4)+(IM<<2)	PC	rs	0			r1	r2	7			

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，单周期 CPU 的数据通路图如图 3.6 所示。

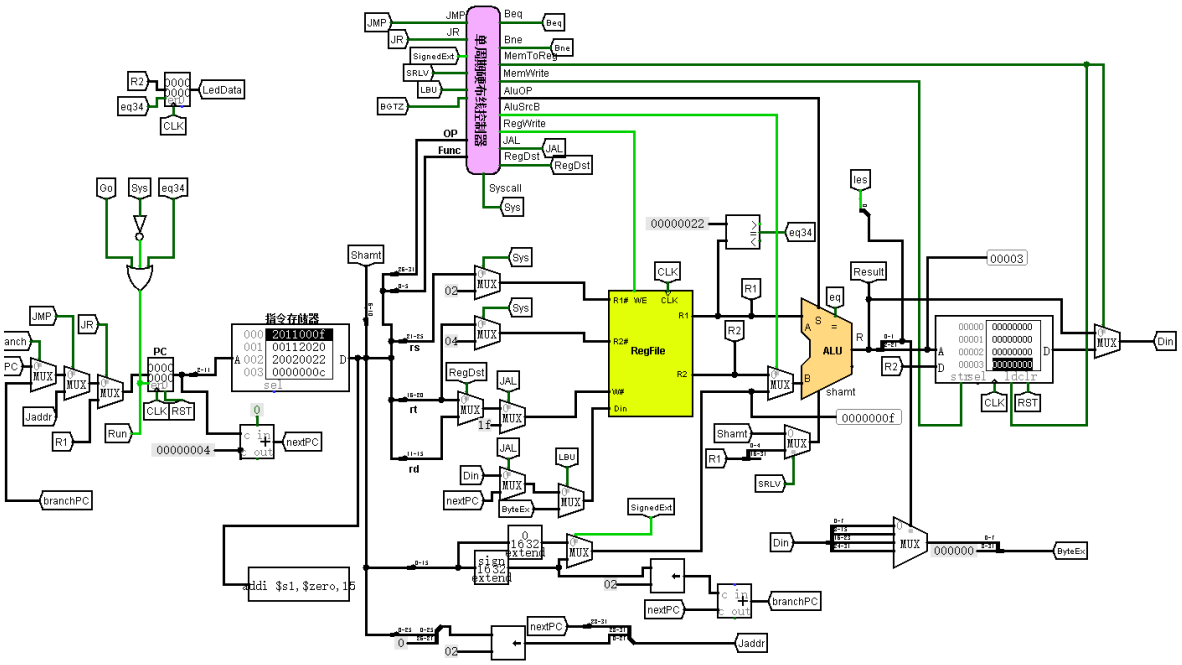

图 3.6 单周期 CPU 数据通路（Logism）

在 Vivado 中使用 Verilog 语言搭建的数据通路的原理图如所示。

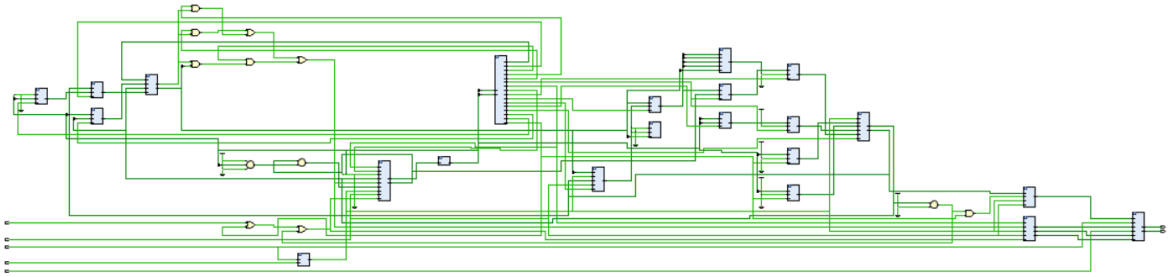

图 3.7 单周期 CPU 数据通路（FPGA）

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 控制器的具体实现。

1) 主控制器

对照表 3.1 数据通路信号表，可以用 logisim 中的自动生成功能生成如下信号：

表 3.2 主控制器控制信号

指令	OpCode	FUNCT	MemtoReg	MemWrite	ALU_SRC	RegWrite	SYSCALL	SignedExt	RegDst	BEQ	BNE	JR	JMP	JAL	SRLV	LBU	BGTZ
SLL	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1
SRA	0	3	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1
SRL	0	2	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1
ADD	0	32	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
ADDU	0	33	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
SUB	0	34	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
AND	0	36	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
OR	0	37	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
NOR	0	39	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
SLT	0	42	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
SLTU	0	43	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
JR	0	8	0	0	0	0	0	0	0	0	0	1	1	0	0	1	0
SYSCALL	0	12	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1
J	2	X	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
JAL	3	X	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0
BEQ	4	X	0	0	0	0	0	0	0	1	0	0	1	0	0	1	1
BNE	5	X	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1
ADDI	8	X	0	0	1	1	0	1	0	0	0	0	0	0	0	1	0
ANDI	12	X	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0
ADDIU	9	X	0	0	1	1	0	1	0	0	0	0	0	0	0	1	0
SLTI	10	X	0	0	1	1	0	1	0	0	0	0	0	0	0	1	0
ORI	13	X	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0
LW	35	X	1	0	1	1	0	0	0	0	0	0	0	0	0	1	0
SW	43	X	0	1	1	0	0	0	0	0	0	0	0	0	0	1	1
SRLV	0	6	0	0	0	1	0	0	1	0	0	0	0	0	1	1	1
XOR	0	38	0	0	0	1	0	0	1	0	0	0	0	0	0	1	1
LBU	36	X	1	0	1	1	0	1	0	0	0	0	0	0	0	1	1
BGTZ	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

① FPGA 实现

根据在 Logism 实现中得到的各个一位控制信号的表达式，直接使用数据流建模，使用 assign 分的 Verilog 代码过于冗长，取对于控制信号 MemToReg 的生成代码举例如下：

```
assign
MemToReg=OP5&~OP4&~OP3&~OP2&OP1&OP0|
    OP5&~OP4&~OP3&OP2&~OP1&OP0 | OP5&~OP4&~OP3&OP2&~OP1&~OP0;
```

ALU_OP 操作码的生成代码举例如下，S3，S2，S1，S0 合并为 4 位操作码，只取 S3 为例：

```
assign ALU_OP = {S3,S2,S1,S0};


    assign S3=~OP5&~OP4&~OP3&~OP2&~OP1&~OP0&F5&~F4&~F3&F2&~F1&F0
    | ~OP5&~OP4&~OP3&~OP2&~OP1&~OP0&F5&~F4&~F3&F2&F1&F0 |
    ~OP5&~OP4&~OP3&~OP2&~OP1&~OP0&F5&~F4&F3&~F2&F1&~F0 |
    ~OP5&~OP4&~OP3&~OP2&~OP1&~OP0&F5&~F4&F3&~F2&F1&F0 |
    ~OP5&~OP4&OP3&~OP2&OP1&~OP0 | ~OP5&~OP4&OP3&OP2&~OP1&OP0 |
    ~OP5&~OP4&OP3&~OP2&OP1&OP0 | ~OP5&~OP4&~OP3&OP2&OP1&OP0 |
    ~OP5&~OP4&~OP3&~OP2&~OP1&~OP0&F5&~F4&~F3&F2&F1&~F0 |
    ~OP5&~OP4&~OP3&OP2&OP1&OP0;
```

以此类推，最终便可以实现整个主控制器中所有控制信号的生成。在 Vivado 中

使用 Verilog 语言构成的主控制器原理图如图 3.8 主控制器原理图所示，其中 alu_contr 用于生成 ALU 操作码，sig_gen 用于生成其他信号。

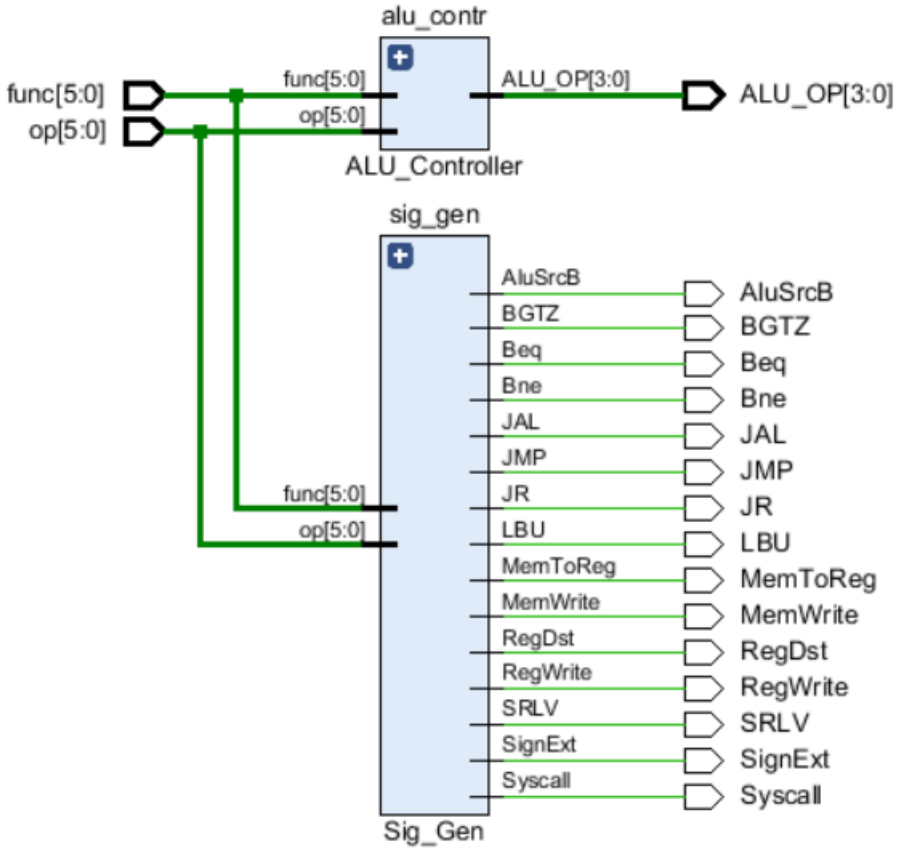

图 3.8 主控制器原理图

2) branch 信号控制器：

① logisim 实现

当 EQ 信号有效且 BEQ 指令有效或者 EQ 信号无效且 BNE 指令有效或者 BGTZ 指令有效且 EQ 指令无效且有符号和 0 比较的输出 EXles 为 0 时，branch 信号有效，logisim 实现如图 3.9。

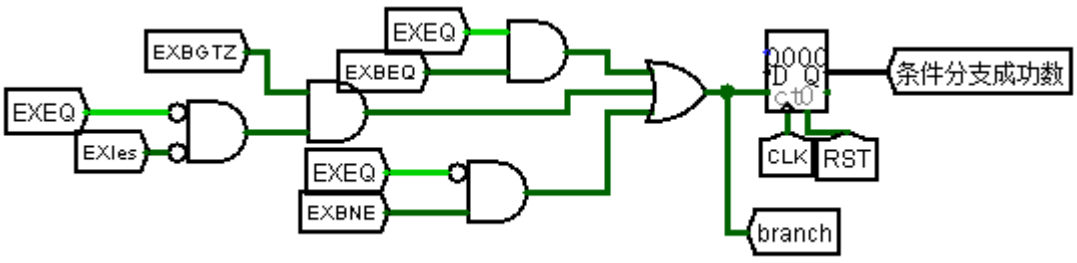

图 3.9 branch 信号控制器

② FPGA 实现

根据在 Logism 实现中得到的各个一位控制信号的表达式，直接使用数据流建模如下：

```
assign branch = (Bne & ~Equal) | (Beq & Equal) | (BGTZ & (~Equal & ~les));
```

3.2 中断机制实现

3.2.1 硬件实现

1) 中断控制器

中断控制器是原有电路和中断处理程序的接口。当输入当前 PC 值，eret，mtc0 指令控制信号以及寄存器堆输出值 R2 时，能够正确输出 PC 下一步跳转的地址，当中断产生时能够跳转到中断处理程序运行而中断返回时能够跳到原有地址运行。logisim 实现图如图所示：图中左上角的多路选择器的结果送往 PC，当 eret 信号有效时，选择已经恢复的 EPC 回到中断前地址，无效时，则选择中断控制器输出的新的 PC 值，logisim 实现如图 3.10。

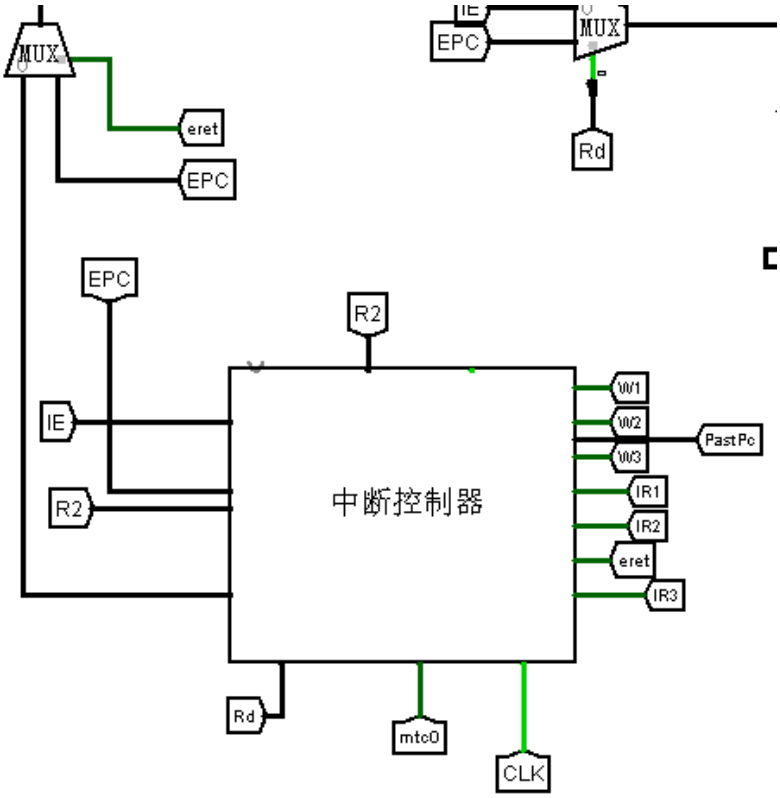

图 3.10 中断控制器

2) 中断使能信号生成器

该部件用于生成中断使能信号，使能端为 mtc0，由于 mtc0 在开中断和关中断时

都会用到，因此该信号可用于同步控制使能寄存器。IE_IN 信号为待更改的中断屏蔽信号的值，当 mtc0 有效且 eret 无效时，IE 会被更改，起到开关终端的作用。eret 有效时，IE 应该恢复开中断的状态，如图 3.11。

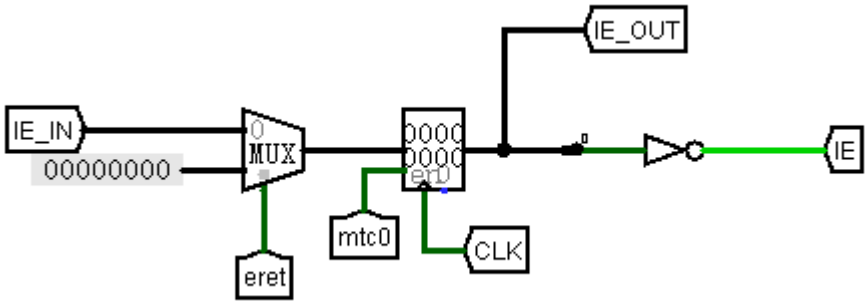

图 3.11 中断使能信号

3) 中断地址跳转生成器

中断地址跳转生成器是中断控制器的核心部件，用于生成中断有效时的跳转地址。由一个多路选择器实现对于原有 PC 和三个中断程序地址的选择。W1~W3 是中断点击的信号，当中断产生后，无论该中断是否被执行，W1~W3 都会保持直到该中断运行结束。当没有中断信号时或者中断被屏蔽或者中断的优先级不够高时，多路选择器的选择端不会出现变化。而当一个优先级较高的中断来临且中断使能寄存器有效时，intpri 也会有效（intpri 为通过比较器生成的表示当前中断优先级较高的信号），多路选择器的选择端会变化为该高优先级中断处理程序的地址，如图 3.12 所示。

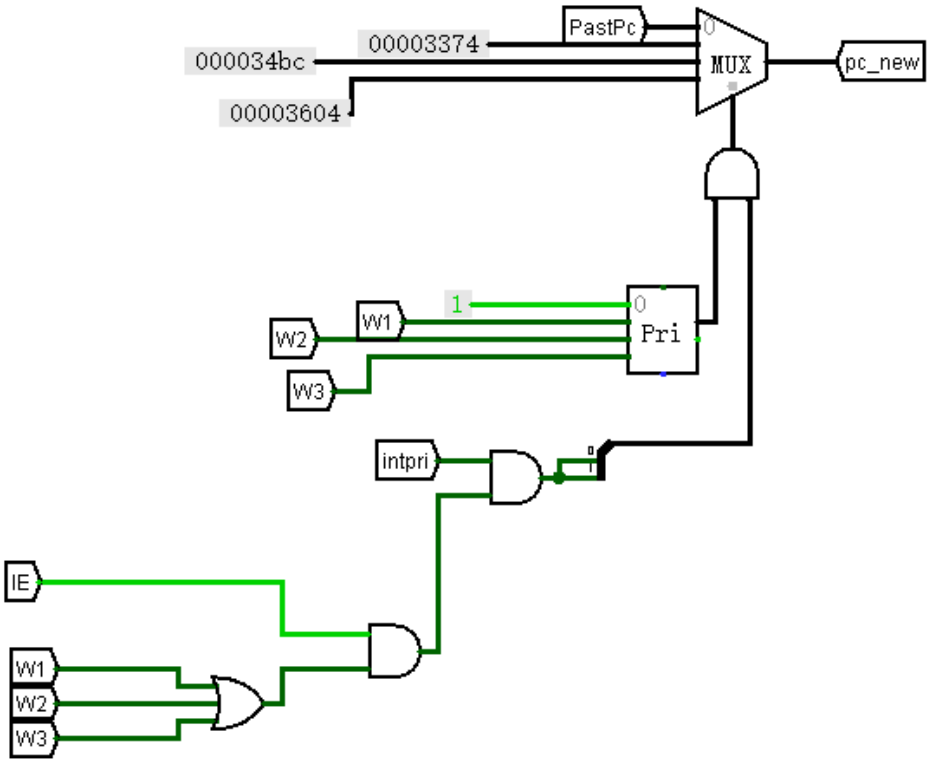

图 3.12 中断地址跳转

3.2.2 软件实现

如下指令可以配合硬件实现关中断和开中断的操作：

```
addiu $k1,$0,1    #关中断
mtc0 $k1,$0
```

将 k1 寄存器置为 1 后，中断屏蔽字有效，调用 mtc0 指令，IE 取相反值，这样就起到了关中断的作用，开中断同理。

如下指令可以保护现有当前 PC 的值以及保护现场：

```
mfc0 $k0,$1
addiu $gp,$gp,-4        #保护 EPC 的值
sw $k0,0($gp)


addiu $sp,$sp,-4     #入栈，保护现场
sw $s0,0($sp)
addiu $sp,$sp,-4
```

```
sw $s2,0($sp)
addiu $sp,$sp,-4
sw $s3,0($sp)
addiu $sp,$sp,-4
sw $t0,0($sp)
addiu $sp,$sp,-4
sw $t1,0($sp)
addiu $sp,$sp,-4
sw $a0,0($sp)
addiu $sp,$sp,-4
sw $v0,0($sp)
addiu $sp,$sp,-4
sw $t8,0($sp)
```

上述指令是将各个通用寄存器的值存入 sp 指向的数据存储器中，相当于入栈操作。恢复现场时则利用 lw 指令实现反向操作。

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

1) logisim 实现

流水接口部件的主要实现是基于寄存器的，如图 3.13 所示为 IF/ID 接口部件电路图。

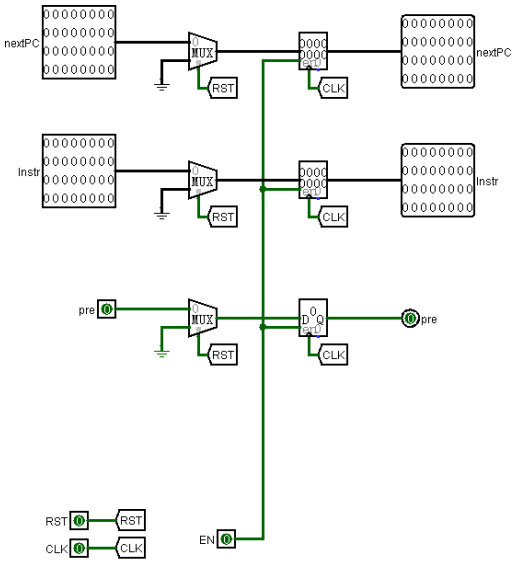

图 3.13 IF/ID 接口部件

部件输入端为 IF 段取出的 nextPC 的值和取出的具体指令，一个时钟周期后，同步时钟 CLK 将寄存器中的值送往输出端。另外，同一个接口部件的清零信号 RST 也是同步的。pre 信号是分支预测中需要的，这里可以忽略。EN 为使能信号，当使能信号位于低电平时，输入流水接口的信息即无效。

2) FPGA 实现

IF/ID 接口流水部件的 Verilog 代码如下所示：

```
always @(posedge clk) begin
        if(rst)  begin
           p_out<=0;    //送往 ID 段的 PC
           i_out<=0;    //送往 ID 段的指令
        end
        else if(IF_EN) begin     //使能端有效
           p_out<=p_in;
           i_out<=i_in;
        end
        else begin
             p_out<=p_out;
             i_out<=i_out;
        end
```

```
end
```

理想流水线中有四处接口部件构成五段流水线，其余流水线实现的功能类似，因此仅举一例。

3.3.2 理想流水线实现

根据理想流水线的设计，各个流水段所需要的控制器产生信号如表 3.3 所示。最终理想流水线实现的 logisim 图如图 3.14 所示

表 3.3 不同流水段所需信号表

流水段 / 信号	IF	ID	EX	MEM	WB
MemToReg		√	√	√	√
MemWrite		√	√	√	
AluOp		√	√		
AluScrB		√	√		
RegWrite		√	√	√	√
RegDst		√	√	√	√
SignedExt		√			
Syscall		√	√	√	√

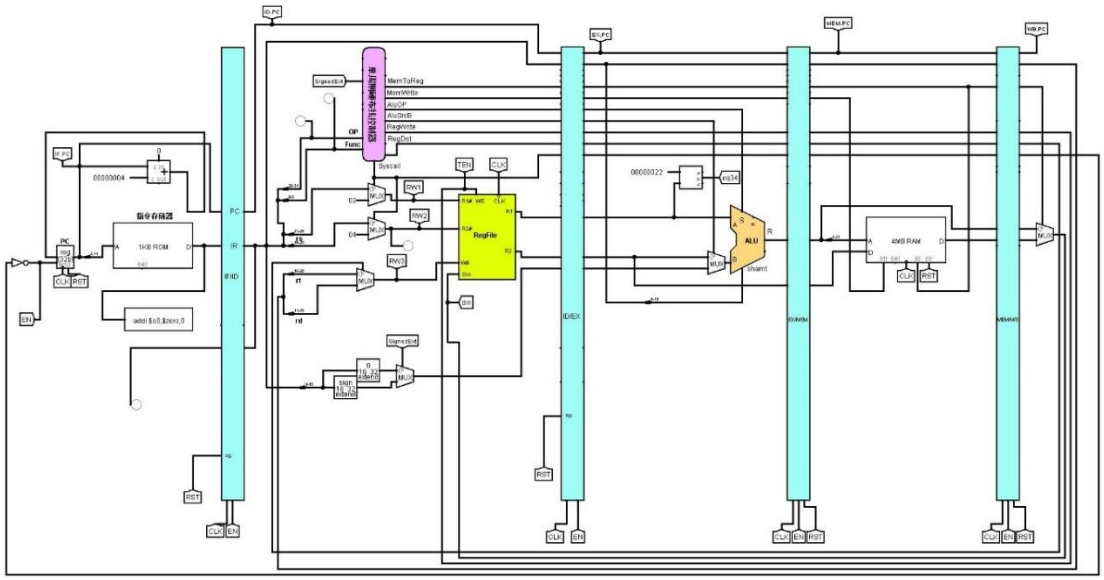

图 3.14 理想流水线 logisim 图

3.4 气泡式流水线实现

3.4.1 数据相关信号生成模块

该模块的目的是生成数据相关信号 Drelate。输入信号和每个输入信号的意义如表 3.4 所示：

表 3.4 数据相关信号生成模块输入信号表

输入信号	备注
OP	MIPS 指令 OP 字段
EX.WriteReg#	执行阶段的写寄存器编号
EX.RegWrite	执行阶段的写寄存器信号
MEM.RegWrite#	访存阶段的写寄存器编号
MEM.RegWrite	访存阶段的写寄存器信号
ID.R1#	R1 寄存器编号
ID.R2#	R2 寄存器编号
Func	MIPS 指令 Func 字段

其中 EX.WriteReg#和 MEM.RegWrite#需要分别根据对应流水段的 RegDst 信号和 JAL 信号确定。以 EX 段为例，当 JAL 信号有效时，该指令会将下一条指令地址送到低 31 号寄存器，因此 EX.WriteReg#为 31。当 JAL 信号无效并且 RegDst 信号有效时，写寄存器号由指令的 Rd 字段决定。当 JAL 和 RegDst 都无效时，写寄存器号由 Rt 字段决定。其余信号只要根据主控制器的输出喜好确定即可。

若需要数据相关信号有效，基础条件是 R1，R2 寄存器被读写，若 R1 被读写，则 R1_used 信号有效，若 R2 被读写，则 R2_used 信号有效。R1，R2 的读写情况由指令本身的任务决定，也就是由指令的 OP 字段和 Func 字段决定，R1_used 和 R2_used 信号如表 3.5 R1_used 信号生成表所示。通过操作码字段和 funct 字段利用 logisim 可以自动生成。

表 3.5 R1_used 信号生成表

指令	OpCode	FUNCT	R1_used	R2_used
SLL	0	0	0	1
SRA	0	3	0	1
SRL	0	2	0	1
ADD	0	32	1	1
ADDU	0	33	1	1
SUB	0	34	1	1
AND	0	36	1	1
OR	0	37	1	1
NOR	0	39	1	1
SLT	0	42	1	1
SLTU	0	43	1	1
JR	0	8	1	0
SYSCALL	0	12	1	1
J	2	X	0	0
JAL	3	X	0	0
BEQ	4	X	1	1
BNE	5	X	1	1
ADDI	8	X	1	1
ANDI	12	X	1	1
ADDIU	9	X	1	1
SLTI	10	X	1	1
ORI	13	X	0	0
LW	35	X	0	0
SW	43	X	0	0
SRLV	0	6	0	0
XOR	0	38	0	0
LBU	36	X	0	0
BGTZ	7	0	1	1

信号生成模块内部如图 3.15 数据相关信号生成所示：

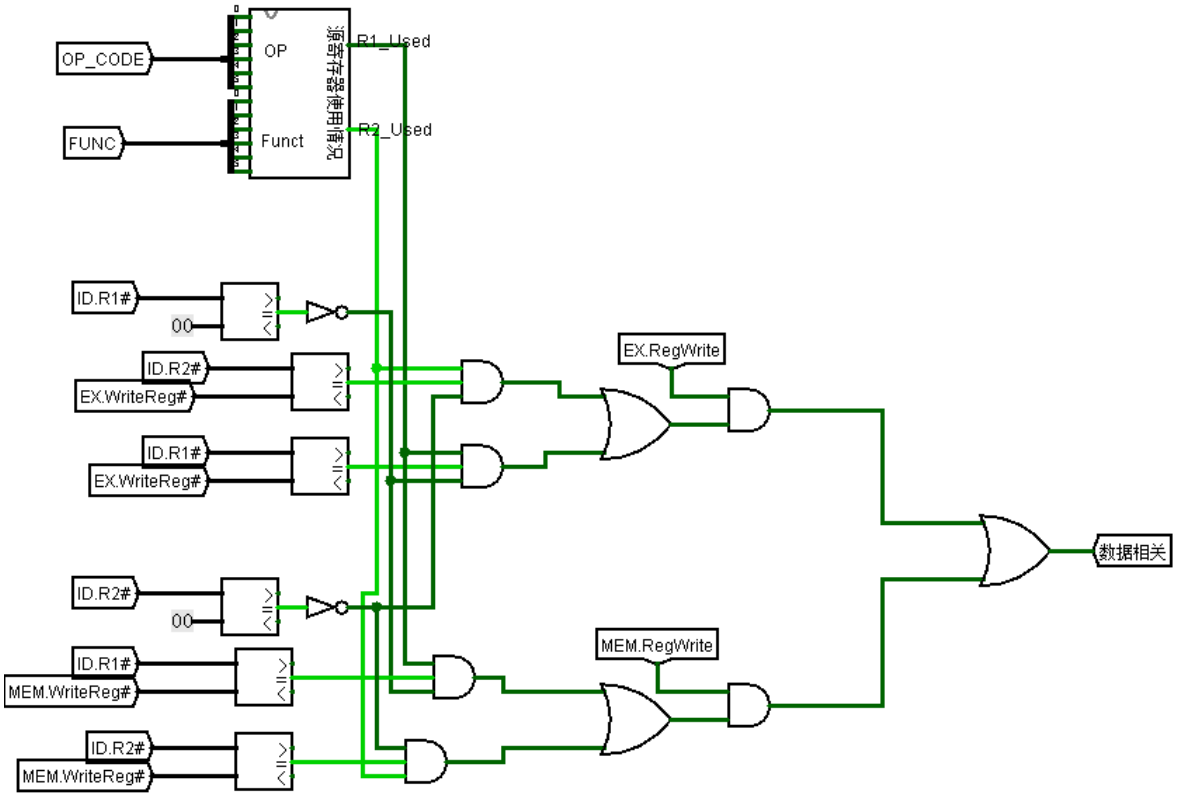

图 3.15 数据相关信号生成

数据相关生成信号分析：以 EX 段和 ID 段导致的数据相关为例，R1，R2 为非 0 寄存器时才考虑相关情况，因为如果是 0 号寄存器，是不可以改变内容的。在此基础

上，当 ID 段的读寄存器号和 EX 段的写寄存器号相等并且写寄存器信号有效时，EX 段和 ID 段数据相关。MEM 段和 ID 段导致的数据相关类似。

3.4.2 气泡式流水线实现

该流水线基于理想流水线实现，因此只涉及不同内容的阐述：

分支相关问题解决需要实现无条件分支信号有效和有条件分支信号有效的生成，分别用 EXJMP 和 branch 信号代替。其中 EXJMP 信号是 J 指令和 JAL 指令流水至 EX 段时的有效信号。而 branch 信号的生成如图 3.6 单周期 CPU 数据通路（Logism）所示。

数据相关基于分支式相关流水线实现，主要的添加点在于数据相关信号 Drelate 信号的使用。当 Drelate 信号有效时，PC 使能信号无效，使得 PC 不会取出下一条指令，IF/ID 段使能信号无效，使得 ID 段数据不会改变，从而起到暂停作用。另外，ID/EX 流水接口部件除了如分支相关电路中在无条件和有条件的分支指令时需要实现清零功能之外，需要加上 Drelate 信号，将其和剩下的信号相或作为清零信号。这样当数据相关时，EX 段会出现一个气泡。

另外，需要由于理想流水线不能完全运行 Benchmark，因此若干信号在流水接口部件的传递中被省略，完整版各个流水段所需信号如表 3.6 所示：

表 3.6 不同流水段所需信号表

流水段 / 信号	IF	ID	EX	MEM	WB
MemToReg		√	√	√	√
MemWrite		√	√	√	
AluOp		√	√		
AluScrB		√	√		
RegWrite		√	√	√	√
RegDst		√	√	√	√
SignedExt		√			
Syscall		√	√	√	√
JMP/JR/JAL		√	√	√	√

流水段 / 信号	IF	ID	EX	MEM	WB
BGTZ		√	√		

3.5 重定向流水线实现

3.5.1 信号生成模块

重定向需要产生 ALU 操作数的多路选择信号 Fwd1 和 Fwd2 以及 loaduse 信号。输入信号和每个输入信号的意义如表 3.7 所示：

表 3.7 信号生成模块输入信号表

输入信号	备注
OP	MIPS 指令 OP 字段
EX.WriteReg#	执行阶段的写寄存器编号
EX.RegWrite	执行阶段的写寄存器信号
MEM.RegWrite#	访存阶段的写寄存器编号
MEM.RegWrite	访存阶段的写寄存器信号
Mem.Mem2Reg	访存阶段的数据存储器回写信号
Ex.Mem2Reg	执行阶段的数据存储器回写信号
ID.R1#	R1 寄存器编号
ID.R2#	R2 寄存器编号
Func	MIPS 指令 Func 字段

其中 EX.WriteReg#和 MEM.RegWrite#的确定和模块内部需要的 R1_used 和 R2_used 信号与气泡式流水线的信号完全相同，不再赘述。

信号生成模块的 logisim 实现如图 3.16 所示：

1) logisim 实现：

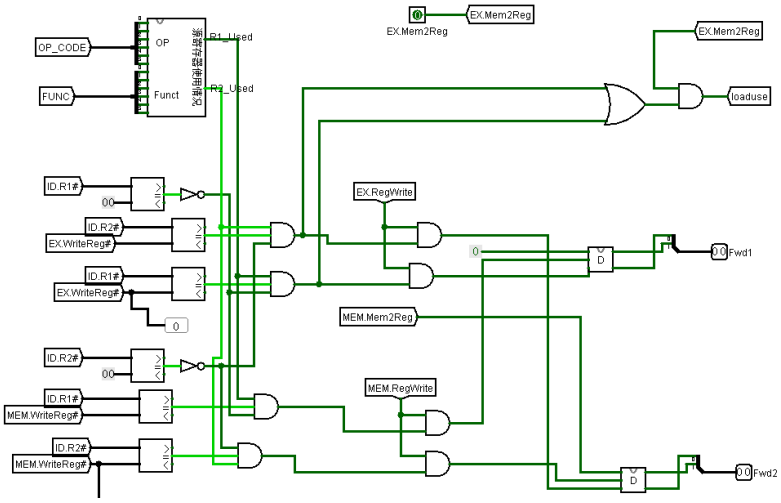

图 3.16 信号生成模块 logisim

如上图所示，loaduse 信号的生成有两个条件：其一是 EX 段使用了 LW 指令，该条件在 EX.MemToReg 信号有效时满足，其二是当 ID 段的读寄存器号和 EX 段的写寄存器号一致时，该条件通过比较 ID.R2#/ ID.R1#和 EX.WriteReg#得到。

下面以 fwd2 为例分析 fwd 信号的生成：当 fwd2 为 10 时，定向的数据从 MEM 段的数据存储器中取出。这种情况是前一条指令为 LW 指令，后一条指令需要读取 LW 写的寄存器，触发 loaduse 后，在 MEM 段完成重定向。当 fwd2 为 01 或者 11 时，重定向数据为 MEM 段的 ALU 输出结果。该情况出现在计算得到的数值需要存入后一条指令所需的寄存器时。

2) FPGA 实现：

重定向流水线的信号生成模块均由 assign 数据流描述方式，代码量太多，举生成 loaduse 信号为例：

```
assign loaduse = EX_MemToReg_out&&(( ~(ID_R2_N==5'h00) &&
(ID_R2_N==EX_RegWrite_N) && R2_used )|| ( R1_used &&
(ID_R1_N==EX_RegWrite_N) && ~(ID_R1_N==5'h00)));


        always @(posedge clock) begin      //clk,rst,run
            if(reset) begin
                PC <= 32'h00003000;
            end
            else if (run && ~reset) begin
```

```
                PC <= JMP_MUXout;
            end
    end

        always @(posedge clk) begin
            if(reset) begin
                totalCycle <= 0;
                unconditionalJump <= 0;
                conditionalSuccessfulJump <= 0;
            end
            else if (run && ~reset) begin
                totalCycle <= totalCycle + 1;
                unconditionalJump <= unconditionalJump + JMP;
                conditionalSuccessfulJump <= conditionalSuccessfulJump + branch;
            end
            else begin
                totalCycle <= totalCycle;
                unconditionalJump <= unconditionalJump;
                conditionalSuccessfulJump <= conditionalSuccessfulJump;
            end
    end
```

3.5.2 重定向实现

重定向流水线的实现是基于气泡式流水线的数据通路实现，主要的不同在于 ALU 输入操作数处的重定向功能。如图 3.17 所示，根据 3.5.1 信号生成模块中的 fwd 重定向控制信号可以实现多路选择器的重定向功能。

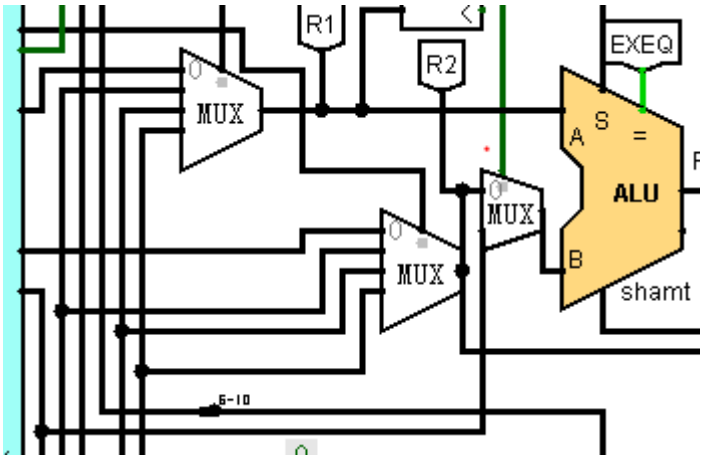

图 3.17 ALU 操作数实现重定向图

Verilog 所对应的 RTL 图如图 3.18 所示：

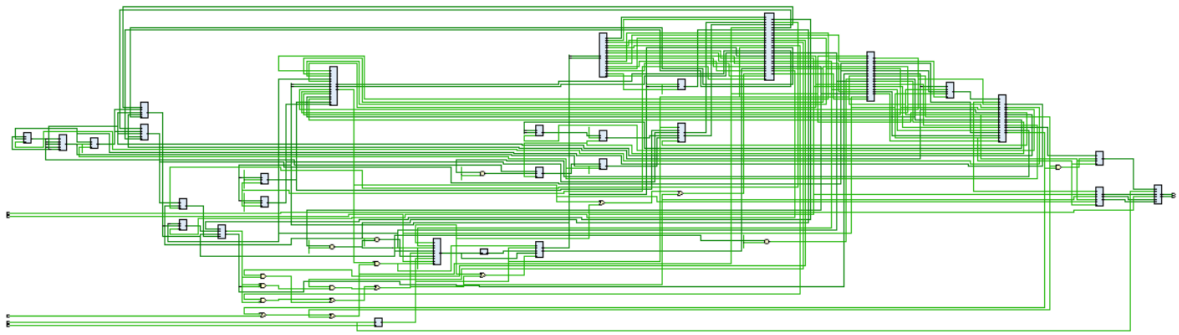

图 3.18 重定向流水线 RTL 图

3.6 动态分支预测机制实现

3.6.1 数据通路实现

如图 3.19 所示为分支预测需要实现的重要数据通路，当 BHT 中 miss 信号无效并且是分支指令时，进行预测，输入 PC 的 Dout 即为预测内容。如果是分支指令，进行预测而预测失败，则按照 EX 段的 PC+4 送入 PC，这时会损失两个流水段。如果未进行预测，则依然按照正常的 PC 执行。

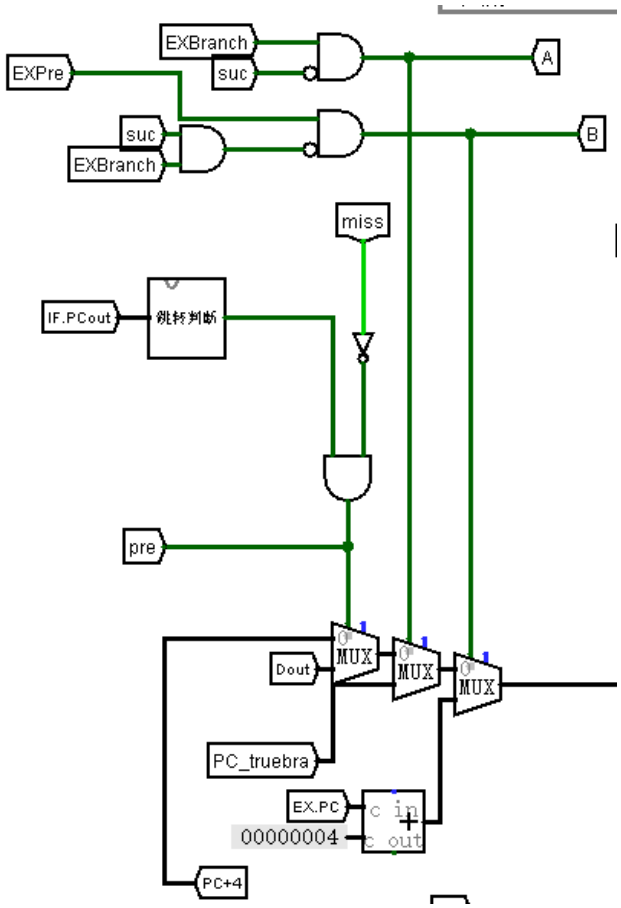

图 3.19 分支预测数据通路

3.6.2 BHT 实现

如图 3.20，为 BHT 的 logisim 实现，未实现双位预测：

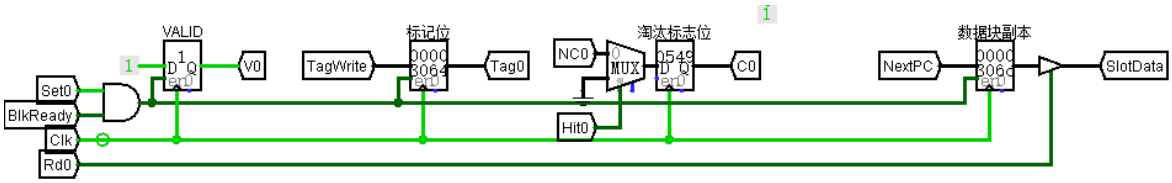

图 3.20 BHT logisim 实现

BHT 的实现基于上学期的 Cache 实验，当第一次分支指令到来时，BlkReady 就是分支信号。写入后 Set 信号有效，那么此时历史跳转值有效，该行可用于分支预测。而且 TagWrite 标记位得到了更新，写入了的分支指令 PC 值，NextPC 得到了更新，写入了本次跳转目的地址。

Set 信号的获取如图 3.21 所示：当需要写历史跳转位时，WriteHit 为 1，WriteSet 为 8 行 BHT 是否被写的信号构成的 8 位信号串。对应的 Set 信号实际上是 WriteSet 的映射，多路选择器起到映射作用。

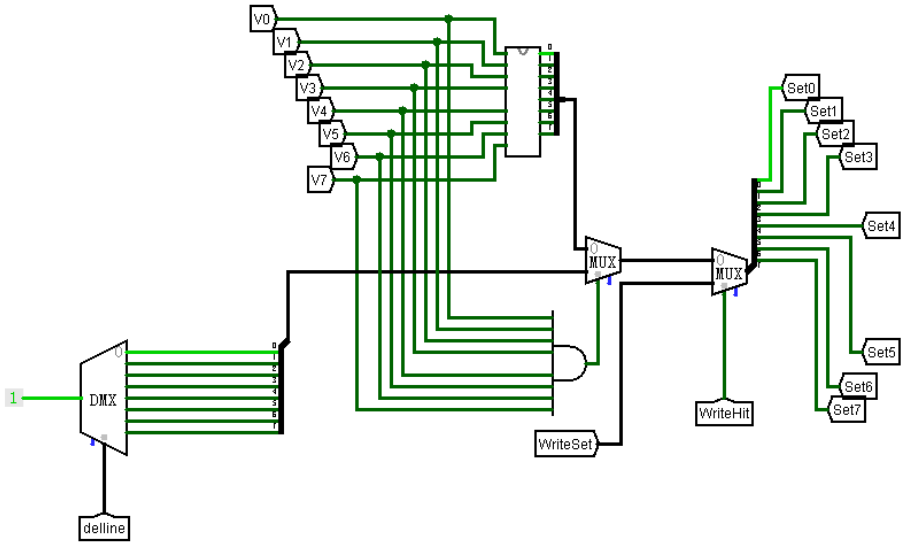

图 3.21 Set 信号生成模块

淘汰标记为根据 Hit 信号设置，当一行被读或者写时，该行的淘汰标记位要被清零。当所有行都已经满了的时候且新的写信号到来且分支指令也到来时，Set 信号更新。BlkReady 为 1，和 Set 信号功能作用于使能端，实现 LRU 淘汰机制。

4 实验过程与调试

4.1 测试用例和功能测试

实验的测试用例为添加了 SRLV，BGTZ，XOR，LBU 的 benchmark 测试程序，利用 Mars 软件编译后生成.hex 文件导入指令存储器进行测试。测试单周期和多周期时主要以总周期数评判性能。将总周期数和分支预测时的总周期数进行比较，测试是否能够降低总周期数。在多级中断测试中，主要测试是否能够正确实现多级中断内容的跳转。

4.1.1 测试单周期流水线

如图 4.1 为 logisim 平台下的周期数以及终止效果图。如图可知排序功能实现。

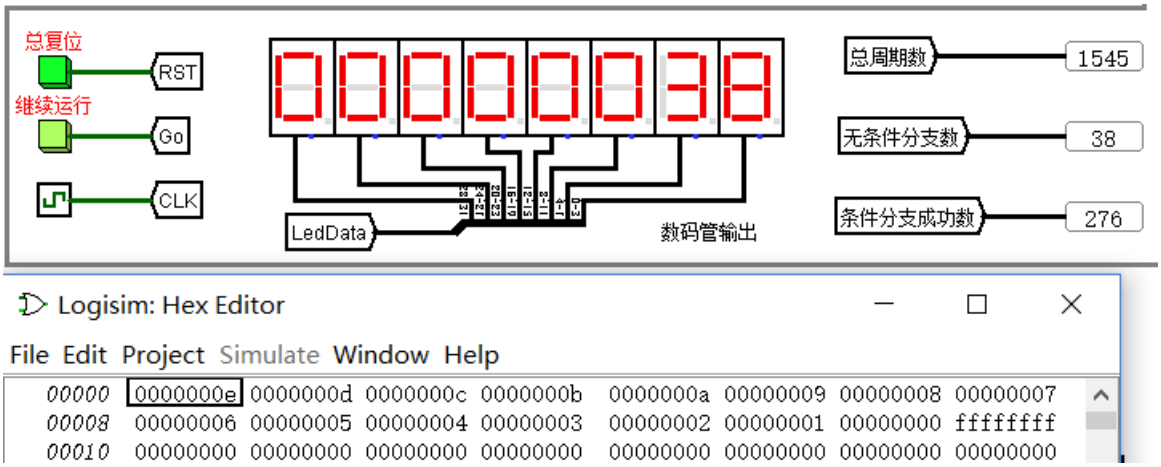

图 4.1 单周期流水线测试 benchmark 图

继续点击 Go 按钮可以测试 ccmb，直到最后 BGTZ 指令运行结束，显示如图 4.2。

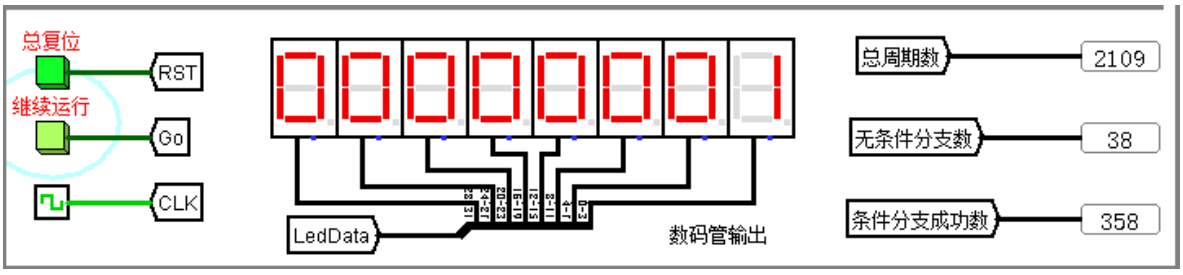

图 4.2 单周期 ccmb 测试

4.1.2 测试气泡流水线

如图 4.3 为 logisim 平台下的周期数以及终止效果图。如图可知排序功能实现。

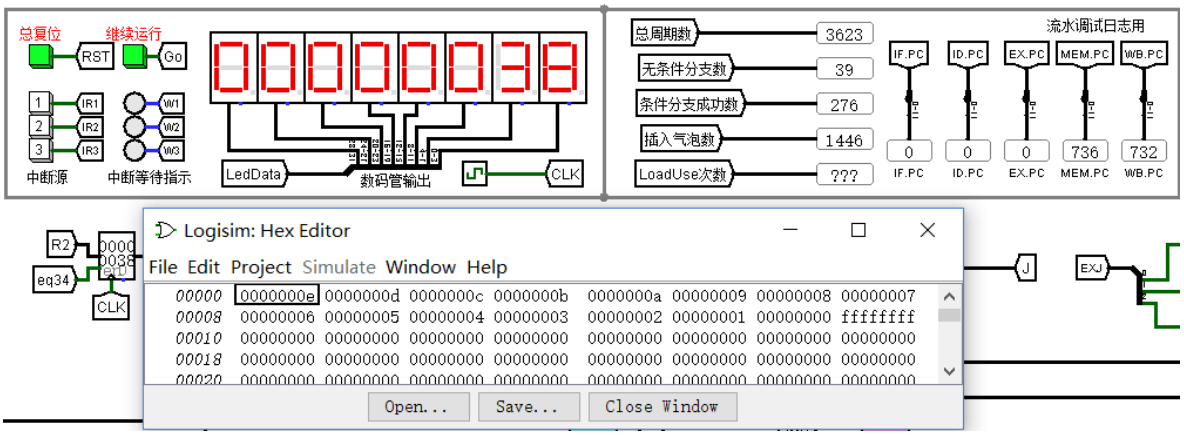

图 4.3 气泡流水线测试

4.1.3 测试重定向流水线

如图图 4.4 为 logisim 平台下的周期数以及终止效果图。如图可知排序功能实现。

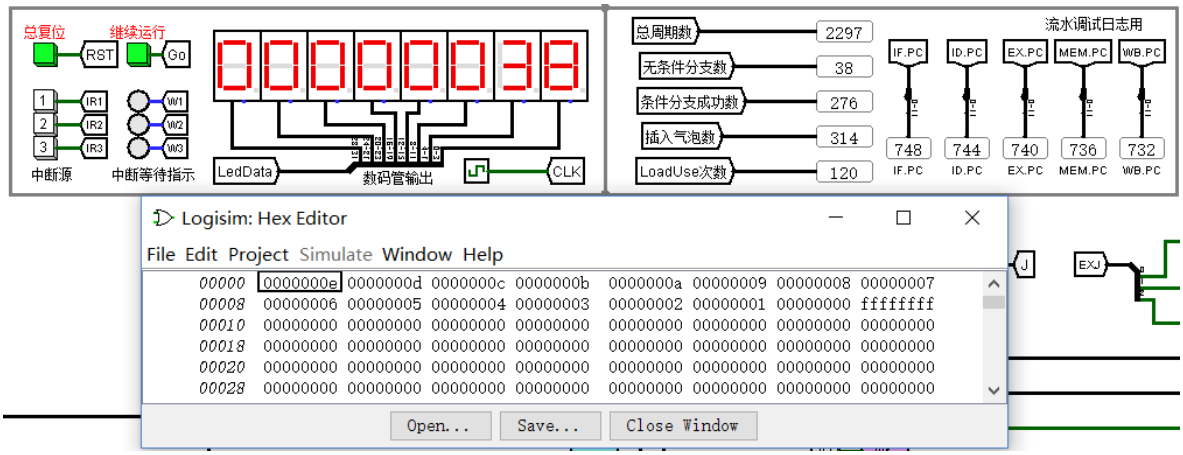

图 4.4 重定向流水线测试

如图 4.5 为运行完 XOR 指令后的 LED 显示情况

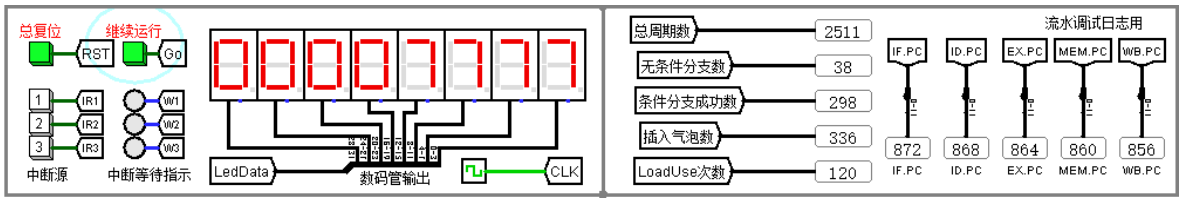

图 4.5 重定向 ccmb 测试

流水线上板的仿真结果如图 4.6 所示，最后 LED 显示 38。

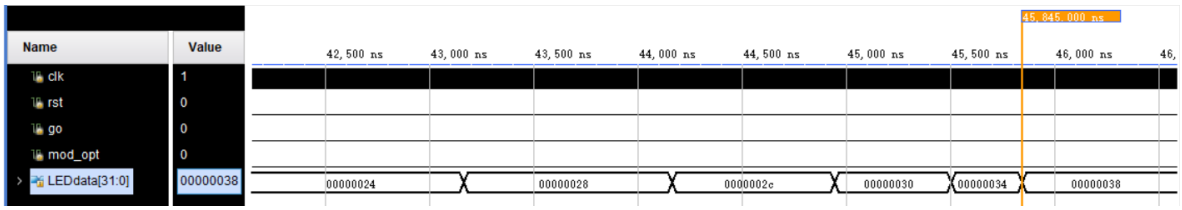

图 4.6 重定向上板仿真

4.1.4 测试多级中断

为了体现多级中断的嵌套特性，此处先按 2 级中断，再按 3 级中断，最后观察程序是否能回到原有程序执行处。

步骤 1：首先令程序正常运行至图 4.7 中所示：

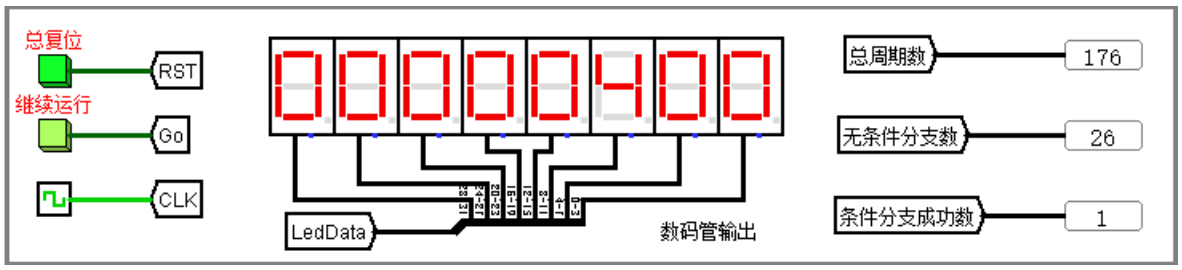

图 4.7 多级中断测试步骤 1

步骤 2：接着点击 2 号中断，进入 2 号中断程序

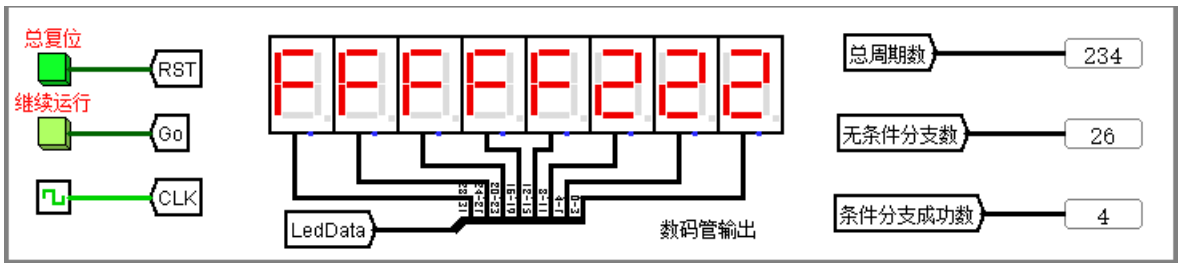

图 4.8 多级中断测试步骤 2

步骤 3：立刻点击 3 号中断，进入 3 号中断程序：

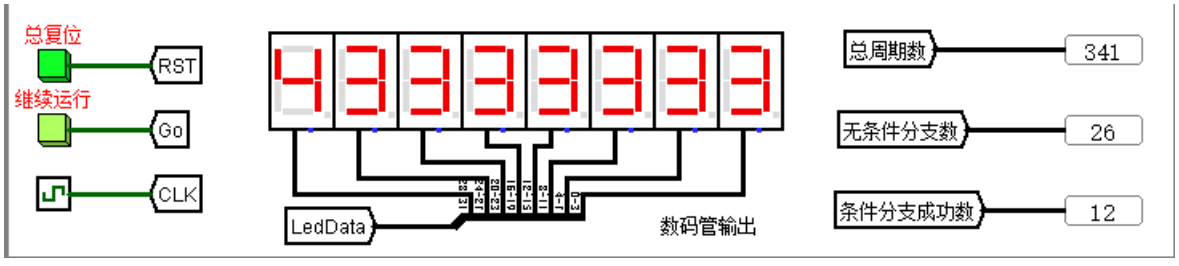

图 4.9 多级中断测试步骤 3

步骤 4：接着等待程序返回至 2 号中断处继续运行：

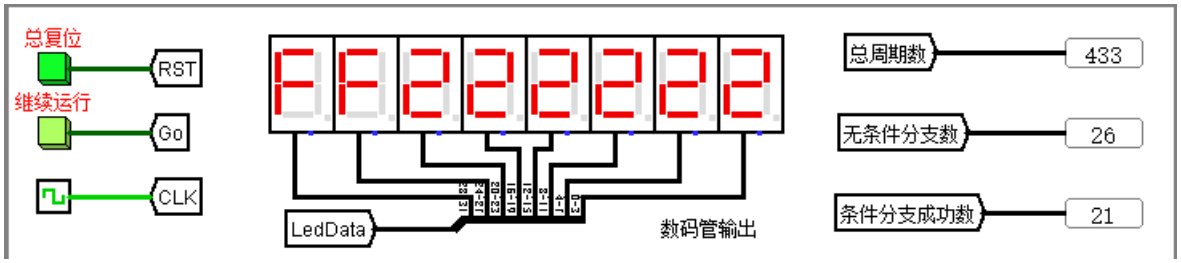

图 4.10 多级中断测试步骤 4

步骤 5：最后程序顺利返回原有运行处继续运行：

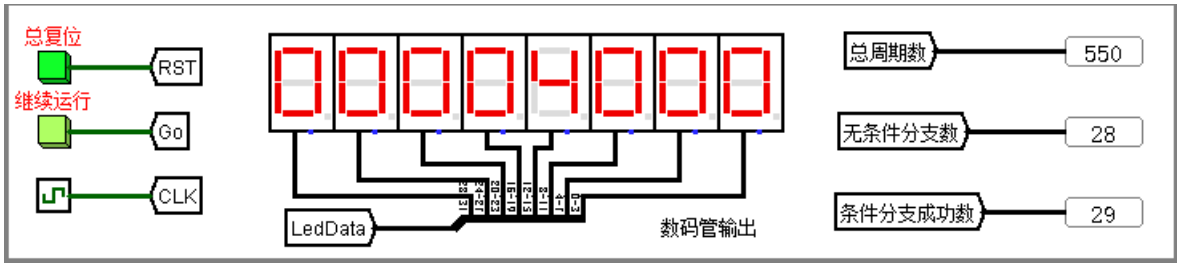

图 4.11 多级中断测试步骤 5

4.1.5 测试分支预测功能

如图 4.12 所示，分支预测完成后总周期数为 1819。

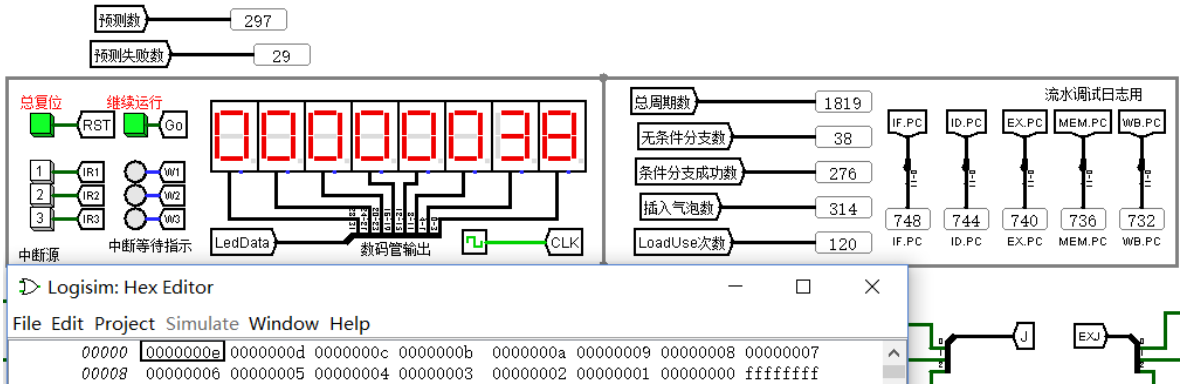

图 4.12 分支预测测试图

该预测结果与预期相符合：分支预测的总周期数=原有周期数(2297)-(预测数*2-预测失败数*4)。

4.2 性能分析

不同实验方案的总周期数不同：最基础的单周期 CPU 需要 1545 个时钟周期数。

气泡流水线需要 3623 个时钟周期，但是气泡流水线时钟周期可以设计得更小，因此和单周期的性能相比，孰优孰劣还有待讨论。但是有一点是确定的，气泡流水线插入了大量气泡，因此还有更大的提升空间。

重定向流水线的性能相比较气泡流水线有了更大的提升，在发生数据相关时除了从内存中读取数据和后一条指令相关的情况需要使用 loaduse 信号插入气泡之外，其余情况可以将数据直接定向到 ALU 端口，因此显著提高了性能，周期数降到了 2297。重定向流水线在分支相关的问题上仍然有较大的提升空间，因此引进了分支预测功能，可以在预测正确率提高的情况下小幅提高性能，该功能帮助流水线的时钟周期数降低至 1819 个周期。

4.3 主要故障与调试

4.3.1 多级中断故障

多级中断：优先级控制错误。

故障现象：在高优先级中断情况下开启低优先级中断直接卡住不运行，如图 4.13 所示，1 级中断信号灯常量，而 2 级中断停在如下状态。

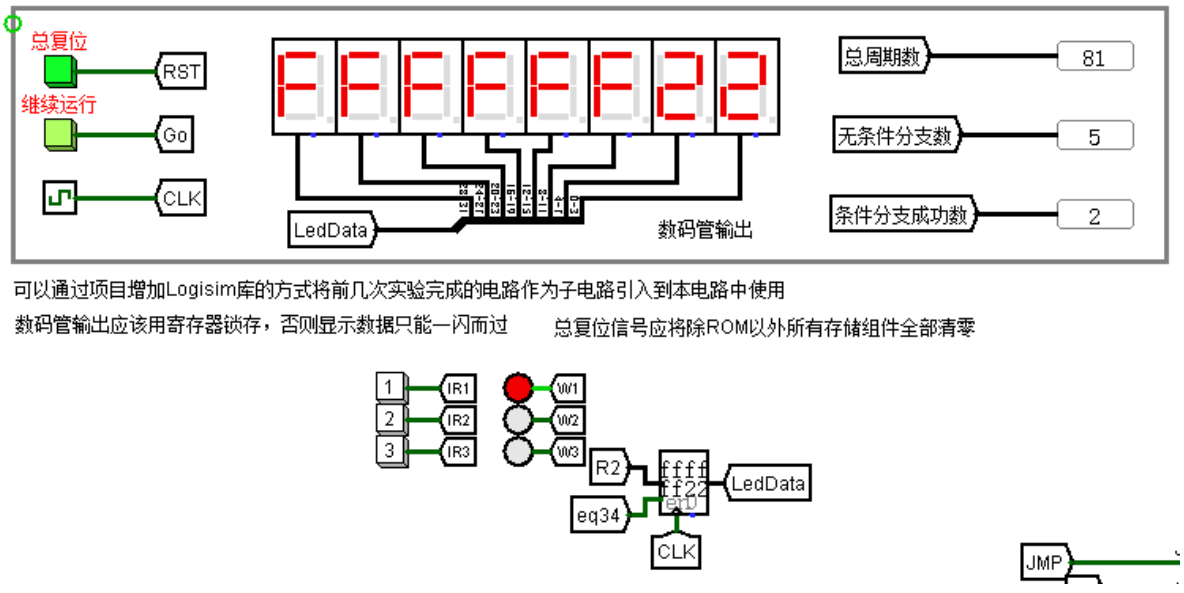

图 4.13 多级中断故障图

原因分析：如图 4.14，虽然使用了优先编码器，但是该编码器只能控制进入中断的程序位置，而没有能够控制是否能够进入中断。即实际上该编码器未能实现中断的优先级控制功能，而只是实现了编码器的功能。这是 pc_new 实际上保持着 1 号中断

的入口地址不变，导致 PC 跳转后一直位于 1 号中断的入口处。

要解决这个错误，需要解决两个问题：1.高优先级对低优先级的屏蔽；2.跳转后能够继续运行中断处理程序。

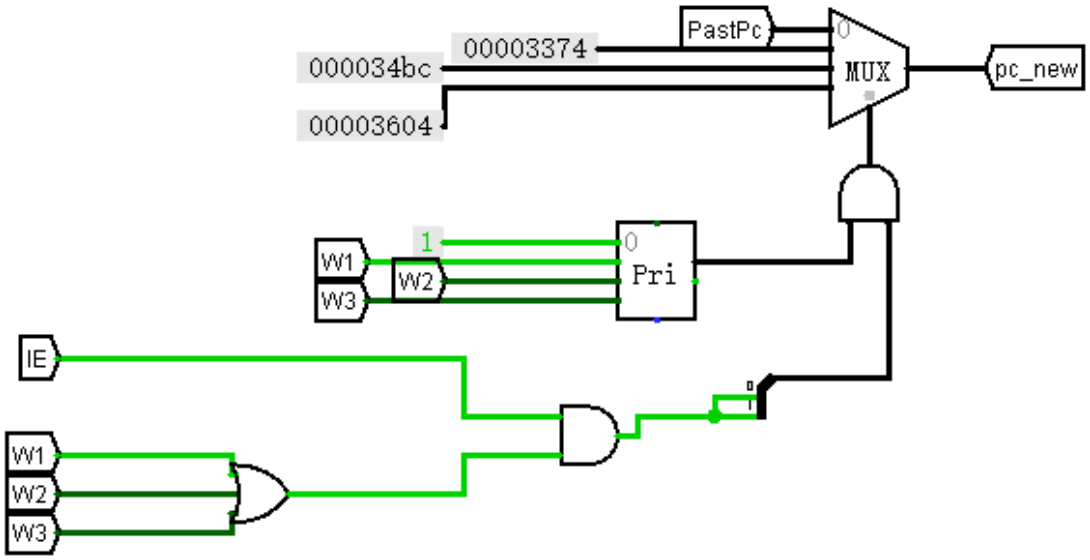

图 4.14 原有方案示意图

解决方案：提供 intpri 信号，通过比较器，将当前中断优先级和原有中断优先级比较，如果优先级高则有效。该信号可以同时解决两个问题，因为如果当前中断优先级相同，那么 intpri 也是无效的，那么多路选择器的选择端将选择正常情况下的 PastPc，如图 4.16 所示。

intpri 信号的产生如图 4.15 所示，其中 IR1_和 IR2_和 IR3_是原有的中断信号，编码得到中断优先级。

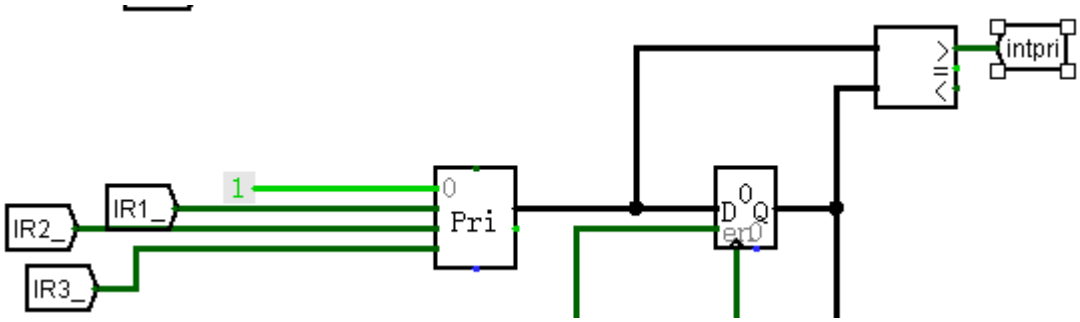

图 4.15 intpri 信号产生图

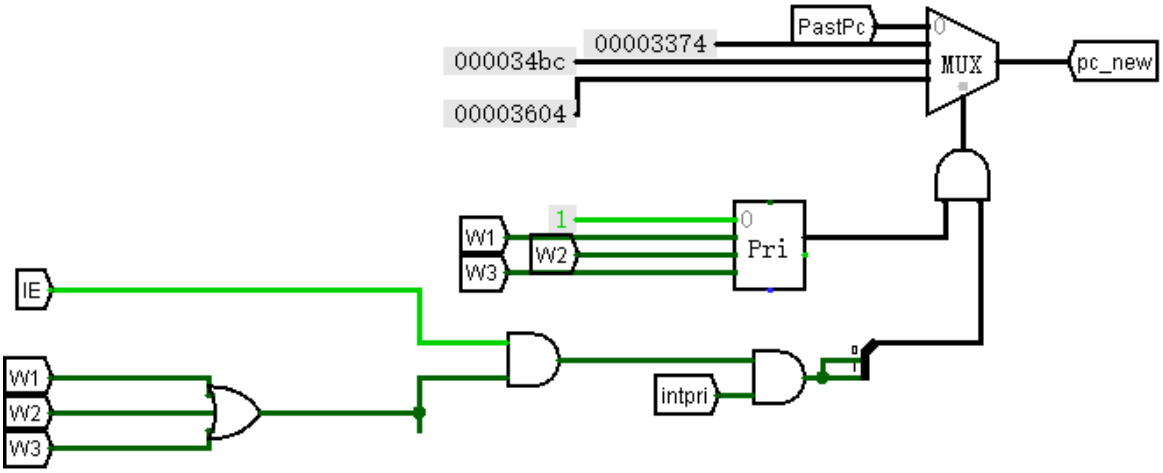

图 4.16 解决方案图

4.3.2 LED 显示故障

重定向流水线：LED 显示控制错误。

故障现象：在重定向流水线中的 LED 显示与实际明显不相符合。但是总周期是正确的。如图，走马灯长时间显示 0。

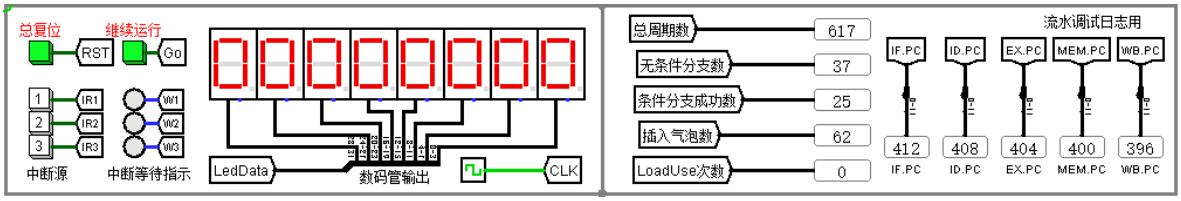

图 4.17 多级中断故障图

原因分析： 在重定向流水线中的 LED 显示与实际明显不相符合。但是总周期是正确的。逻辑没有问题，必然是显示内容出现了问题。问题的原因就是 R1 等于 34 的 LED 显示使能信号未能与 LEData 同步。在气泡流水线中没有出现这个问题的原因是当时的 eq34 信号不需要传递至 WB 段，而重定向流水线中需要，即重定向中的 eq34 信号实际上是 WB.eq34，而非 EX.eq34，因此会出现该显示的时候没有显示的情况。

解决方案：在流水部件上另外增加接口，将 R2 的值引至 WB 段，使得可以同步显示。

4.3.3 Vivado multi-driven 故障

单周期上板：多个驱动的错误。

故障现象：单周期 CPU 上板时仿真没有问题，综合时 vivado 报如下的错误。定位至我负责的 PC 模块。

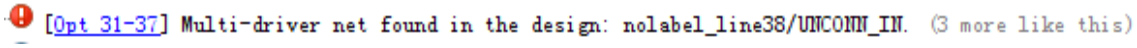

图 4.18 multi-driven 故障图

原因分析：通过上网查询资料，了解到这个错误是因为对于同一个信号，在不同的地方给它赋值。比如在两个 always 块中给同一个信号赋值。为了平均工作量，我的 PC 模块中包含了计数模块和 PC 模块，在单独测试 PC 时，并不会出现这样的问题，并且整个 PC 模块只有一个 always 块。

解决方案：将在同组同学杨晞珩的建议下，我将 PC 中的计数模块拆分出来，新建一个模块 Count，则可以顺利解决这个问题。Count 和 PC 的 verilog 代码如下：

```
always @(posedge clock) begin      //clk,rst,run
        if(reset) begin
            PC <= 32'h00003000;
        end
        else if (run && ~reset) begin
            PC <= JMP_MUXout;
        end
end


    always @(posedge clk) begin
        if(reset) begin
            totalCycle <= 0;
            unconditionalJump <= 0;
            conditionalSuccessfulJump <= 0;
        end
        else if (run && ~reset) begin
            totalCycle <= totalCycle + 1;
            unconditionalJump <= unconditionalJump + JMP;
            conditionalSuccessfulJump <= conditionalSuccessfulJump + branch;
        end
```

```
            else begin
                totalCycle <= totalCycle;
                unconditionalJump <= unconditionalJump;
                conditionalSuccessfulJump <= conditionalSuccessfulJump;
            end
    end
```

4.4 实验进度

表 4.1 课程设计进度表

时间	进度
第一天	完成了 PC 计数器模块的设计
第二天	完成了 PC 模块的代码编写和行为仿真，帮助调试 24 条指令 CPU
第三天	完成了 28 条指令 CPU 的 logisim 和 verilog 修改
第四天	完成 28 条指令单周期的上板，最高跑到 100MHZ
第五天	完成理想流水线通路、气泡流水线设计和调试
第六天	完成重定向 logisim 设计和实现
第七天	完成多级中断功能和分支预测功能
第八天	重定向流水线上板
第九天	

5 设计总结与心得

5.1 课设总结

本次课设通过通过逐步升级的方式，从单周期 CPU 升级到流水线 CPU，接着在理想流水线的基础上实现多种功能，在这个过程中主要做了如下的工作：

1） 完成了单周期 CPU、气泡流水线 CPU（基于理想流水线 CPU）、重定向流水线 CPU、单周期多级嵌套中断、分支预测的 logisim 实现。完成了单周期 CPU 和重定向流水线 CPU 的 Verilog 开发并在开发板上进行了验证。

2） 单周期 CPU 和重定向流水线 CPU 实现了 28 条支持指令的支持，这些指令为：SLL，SRA，SRL，ADD，ADDU，SUB，AND，OR，NOR，SLT，SLTU，JR，SYSCALL，J，JAL，BEQ，BNE，ADDI，ANDI，ADDIU，SLTI，ORI，LW，SW，SRLV，XOR，LBU，BGTZ。气泡流水线 CPU 实现了通过插入气泡方式避免分支相关和数据相关问题的功能。重定向流水线 CPU 实现了将正确数据重定向至 ALU 操作数入口的功能。实现了通过 BHT 功能部件基于历史跳转，对于分支结果进行预测的功能，进一步提升了性能。另外，通过软硬件的合作，实现了单周期 3 级多级嵌套中断。

5.2 课设心得

收获有三点：技能上更加精进，知识上大幅提升，提升了团队合作的经验。

本次课设我使用了 Mooc 里面所讲的高级调试技巧，将 logisim 中的日志进行输出，输入 excel 进行比对，这样可以较为快速地实现大规模指令运行情况下的错误定位。其余的 logisim 调试技巧和 verilog 调试技巧分别在组成原理的课程实验和数字电路的课程设计中进行了锻炼，这次课程设计也再次加深了印象。

知识上来讲，本次课程设计第一次实践了流水线的概念，并且对于流水线需要处理的数据相关和分支相关问题有了初步的了解，掌握了基础的解决这些问题的方法。这对我在近两周的系统结构课程中的学习帮助非常大。此外，分支预测和多级中断的

概念虽然在上学期的组原课程上已有介绍，但是自己真正完成了之后才体现到了实现的困难和复杂。

团队合作中我也吸取了一些教训。28 条 CPU 的单周期指令是由团队合作完成，我负责的模块是程序计数器模块，虽然代码量的编写不是特别多，也自己完成了测试，但是由于接口命名的问题让负责数据通路的成员较为迷惑，再加上没有及时进行交流，导致我的 PC 模块出现了接口错误，也花费了团队一些时间才找到该错误。

下面提出一些建议：

首先，希望老师可以将中断的任务扩展一下设计为多人合作任务，该任务比较适合作为多人合作型，因为该任务有软件设计和硬件设计两部分，两部分可以由不同的成员完成。另外，现实开发中的硬件开发和软件开发是分层的，小组合作的时候也可以这样做，这样可以避免硬件设计者由于只需要满足特定的任务而简化硬件设计的情况，比如将 CP0 给简化。

另外，方案扩展还可以多样化一些，例如 MIPS 指令的分支相关可以通过延迟槽技术去解决。延迟槽可以由编译器实现，但是也应该可以由 logisim 实现，不同的方案也可以最后来比较一下，让学生知道哪一种方案是更好的，业界更常用的。

最后，感谢课程组的精品课设，祝萝卜老师的计算机硬件系统设计 MOOC 人数爆满。

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: 蒋苡杭