

计算机组成原理

课程设计报告

题 目: 5 段流水 CPU 设计

专 业:

班 级:

学 号:

姓 名: Dracula

电 话:

邮 件:

完成日期: 2017-03-010 周五下午



目 录

1 课程设计概述.....	3
1.1 课设目的.....	3
1.2 设计任务.....	3
1.3 设计要求.....	3
1.4 技术指标.....	4
2 总体方案设计.....	6
2.1 单周期 CPU 设计.....	6
2.2 中断机制设计.....	11
2.3 流水 CPU 设计.....	12
2.4 气泡式流水线设计.....	13
2.5 数据转发流水线设计.....	13
2.6 流水线中断与动态分支预测机制设计.....	14
3 详细设计与实现.....	16
3.1 单周期 CPU 实现.....	16
3.2 中断机制实现.....	27
3.3 流水 CPU 实现.....	32
3.4 气泡式流水线实现.....	35
3.5 数据转发流水线实现.....	39
3.6 流水线中断与动态分支预测机制实现.....	43
4 实验过程与调试.....	49
4.1 测试用例.....	49
4.2 功能测试.....	50
4.3 性能分析.....	55
4.4 主要故障与调试.....	57

4.5	实验进度.....	60
5	设计总结与心得.....	61
5.1	课设总结.....	61
5.2	课设心得.....	61
	参考文献.....	64

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, 根据任务书及老师指定, 所需完成的 4 条扩展指令为“1234”, 分别为 sllv、sriv、lh 和 bgez;

(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	SLLV	逻辑可变左移	扩展指令, 参考 MIPS 指令集, 以 Mars 模拟器为准
29	SRLV	逻辑可变右移	
30	LH	加载半字	
31	BGEZ	大于等于 0 转移	

2 总体方案设计

2.1 单周期 CPU 设计

本次课程设计的 CPU 采用的设计方案是硬布线，以及双存储器的设计方案。由于单周期 CPU 的特点，所以各主要器件无法在一个周期内共用，因此，将存储器分为两部分，指令存储器 ROM 和数据存储器 RAM。CPU 中的所有控制信号由控制器给出。同时在实施的过程中，采用 Logisim 和 FPGA 双平台实现，首先在 Logisim 平台上用基本逻辑器件搭建电路，然后利用 Verilog 并结合 Logisim 的电路图编写可在 FPGA 开发板上运行的“CPU”。单周期 CPU 的总体结构图如图 2.1 所示。

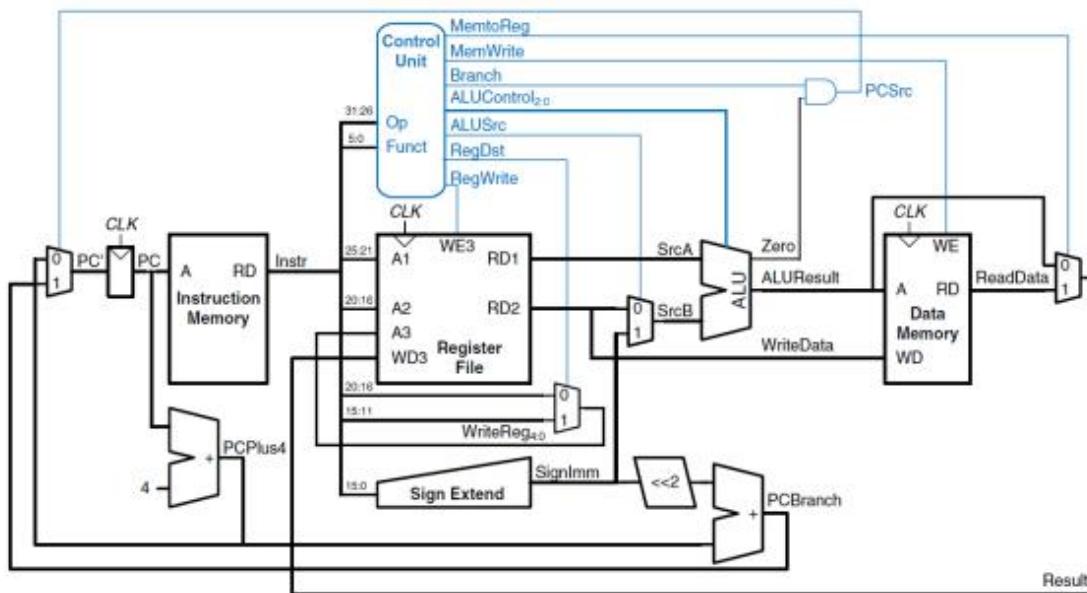


图 2.1 总体结构图

2.1.1 主要功能部件

在 CPU 中，主要的功能部件有：程序计数器 PC，是一个 32 位的寄存器，指向下一条指令的地址；指令存储器 IM，是一个只读存储器 ROM，存储待执行的指令集；运算器 ALU，负责执行指令中的计算部分，包括数值计算和跳转指令的地址计算；寄存器堆 RF，在 MIPS 中，RF 包含 32 个通用寄存器，0 号寄存器输出恒为 0。

1. 程序计数器 PC

PC 本质是一个寄存器，时钟信号应该是 CPU 统一的时钟信号，输出的数据应为当前指令的地址，输入的数据应为当前指令下一条指令的地址，并在时钟信号的上升沿将输入数据写入。

2. 指令存储器 IM

IM 本质是一个只读存储器 ROM，因为指令在执行过程中不能也不可能被修改，因此指令存储器选择 ROM 即可。在程序运行前，须将程序的指令集加载到 IM 中。其输入的数据应为 PC 的输出，即当前指令的地址，输出的数据应为当前指令。

3. 运算器

运算器 ALU，可支持算数加、减、乘、除，逻辑与、或、非、异或运算、逻辑左移、逻辑右移，算术右移运算，支持常用程序状态标志（有符号溢出 OF、无符号溢出 CF，结果相等 Equal）。运算器输入输出引脚见表 2.1 运算器引脚与功能描述，各操作符对应的运算操作如表 2.2 运算符对应运算器功能表所示。

表 2.1 运算器引脚与功能描述

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

表 2.2 运算符对应运算器功能表

ALU OP	十进制	运算功能		
0000	0	Result = $X \ll Y$	逻辑左移	(Y 取低五位) Result2=0
0001	1	Result = $X \ggg Y$	算术右移	(Y 取低五位) Result2=0
0010	2	Result = $X \gg Y$	逻辑右移	(Y 取低五位) Result2=0
0011	3	Result = $(X * Y)[31:0]$; Result2 = $(X * Y)[63:32]$	有符号	
0100	4	Result = X/Y ; Result2 = $X \% Y$	无符号	
0101	5	Result = $X + Y$	Result2=0	(Set OF/CF)
0110	6	Result = $X - Y$	Result2=0	(Set OF/CF)
0111	7	Result = $X \& Y$	Result2=0	
1000	8	Result = $X Y$	Result2=0	
1001	9	Result = $X \oplus Y$	Result2=0	
1010	10	Result = $\sim(X Y)$	Result2=0	
1011	11	Result = $(X < Y) ? 1 : 0$	Signed	Result2=0
1100	12	Result = $(X < Y) ? 1 : 0$	Unsigned	Result2=0
1101	13	Result = Result2=0		
1110	14	Result = Result2=0		
1111	15	Result = Result2=0		

4. 寄存器堆 RF

MIPS 寄存器堆，内部包含 32 个 32 位寄存器寄存器堆输入输出引脚及功能描述见表 2.3 寄存器堆引脚与功能描述。注意零号寄存器值应该恒零。

表 2.3 寄存器堆引脚与功能描述

引脚	输入/输出	位宽	功能描述
R1#	输入	5	读寄存器 1 编号
R2#	输入	5	读寄存器 2 编号
W#	输入	5	写入寄存器编号
Din	输入	32	写入数据

WE	输入	1	写入使能信号
CLK	输入	1	时钟信号, 上跳沿有效
R1	输出	32	R1#寄存器的值
R2	输出	32	R2#寄存器的值
\$v0	输出	32	编号为 2 的寄存器的值
\$a0	输出	32	编号为 4 的寄存器的值
\$ra	输出	32	编号为 31 的寄存器的值

2.1.2 数据通路的设计

首先, 构建一个数据通路的第一步是分析在数据通路上要用到哪些部件。往细了思考, 取指令通路需要用到 PC 寄存器、程序存储器 ROM、加法器, 指令解析模块作为一个芯片直接使用, 控制器在 2.1.3 控制器的设计中会详细介绍, 故这里也是作为一个封装好的芯片直接使用, 存储模块需要用到寄存器堆和数据存储器 RAM, 执行模块需要用到运算器 ALU、有符号扩展、无符号扩展等器件。除此之外, 考虑到很多器件的输入源可能有多个, 因此还需要用多路选择器来选择输入源。当然, 一些基本的门电路像与门、或门、与非门等肯定也是会用到的。

然后, 需要考虑具体是哪些器件有多个数据来源。根据上述考虑, 作出指令系统的数据通路框架见表 2.4 指令系统数据通路框架

表 2.4 指令系统数据通路框架

指令	PC	EPC	PCAdd		InsMem	RF				(Sign/Zero)	ALU			DM		
			A	B		R1#	R2#	W#	Din		Exts	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 2.5 主控制器控制信号的作用说明

表 2.5 主控制器控制信号的作用说明

控制信号	取值	说明
PCSrc	0	PC 寄存器数据来源为寄存器堆的 R1 口输出
	1	PC 寄存器数据来源为 PC+1
	2	PC 寄存器数据来源为 PC 加法器的输出
	3	PC 寄存器数据来源为 26->32 无符号扩展器的输出
PCAddSrc	0	PC 加法器加数 B 的数据来源为常数 0
	1	PC 加法器加数 B 的数据来源为有符号扩展器的输出
PCEn	1	PC 寄存器写使能
RFR1Src	0	寄存器堆 R1 口的数据来源为 Rs
	1	寄存器堆 R1 口的数据来源为 Rt
RFR2Src	0	寄存器堆 R1 口的数据来源为 Rs
	1	寄存器堆 R1 口的数据来源为 Rt
RFRWSrc	0	寄存器堆 RF 口的数据来源为 Rt
	1	寄存器堆 RF 口的数据来源为 Rd
	2	寄存器堆 RF 口的数据来源为常数 31
RFDinSrc	0	寄存器堆 Din 口的数据来源为 PC 加法器的输出
	1	寄存器堆 Din 口的数据来源为数据存储器 DM 的输出
	2	寄存器堆 Din 口的数据来源为运算器 ALU 的输出
	3	寄存器堆 Din 口的数据来源为有符号扩展器 2 的输出
RFWE	0	寄存器堆写使能端为 0
	1	寄存器堆写使能端为 1
ALUYSrc	0	运算器的操作数 Y 的临时数据来源为寄存器堆的 R2 口输出
	1	运算器的操作数 Y 的临时数据来源为 5->32 无符号扩展器的输出
	2	运算器的操作数 Y 的临时数据来源为有符号扩展器的输出
	3	运算器的操作数 Y 的临时数据来源为 16->32 无符号扩展器的输出
ALUYSrc2	0	运算器的操作数 Y 的数据来源为临时数据来源（四路选择器）
	1	运算器的操作数 Y 的数据来源为常数 0

对照所有控制信号，依次分析各条指令，分析该指令执行过程中需要哪些控制

信号，对于与本条指令无关的控制信号，控制信号的取值一律为 0，以简化控制器电路的设计。该控制信号表的框架如表 2.6 主控制器控制信号框架所示。

表 2.6 主控制器控制信号框架

指令	PC Src	PCAdd Src	PC EN	RFR1 Src	RFR2 Src	RFRW Src	RF WE	ALUY Src	ALUY Src2	ALU OP	DMWE

2.2 中断机制设计

2.2.1 总体设计

在标准 MIPS 的 CPU 中，为了能够响应和处理中断，添加了协处理器 CP0，并且 CP0 中也包含一组专用寄存器。在本设计中，借鉴了标准 MIPS-CPU 的设计思想，但对其实现方式进行了简化。对于 CP0 的一组（32 个）专用寄存器，也简化为 2 个寄存器——EPC（储存断点处的 PC 值）和包含中断使能与中断屏蔽字的寄存器。

除此之外，本次设计实现的为多级嵌套中断，共有 4 个中断源，可通过设置屏蔽字的方法实现中断的优先响应。

2.2.2 硬件设计

增加专门处理和响应中断的模块 INTR，该模块以中断源及一些控制信号作为输入，内部含有中断使能寄存器 IE 和屏蔽字寄存器 INM，可根据 IE 和 INM 的值对中断优先响应。除此之外，再增加一个专用寄存器 EPC 用于储存断点处的 PC 值。

IE 和 INM 被组合成“CP0”的偶数号寄存器，而储存中断触发时 PC 的 EPC 寄存器被封装为“CP0”的奇数号寄存器。通过 mtc0 和 mfc0 可访问这两个寄存器（前者 CP0 的寄存器号用偶数，后者用奇数）。简而言之，本 CPU 的“CP0”中只包含两个 32 位专用寄存器。

2.2.3 软件设计

扩展 3 条指令：mfc0、mtc0 和 eret。mfc0 用于将 CP0 中某个寄存器的值读入到通用寄存器中；mtc0 用于将 CP0 中通用寄存器的值写入到 CP0 的某个寄存器中；eret 用于从中断处理程序返回到断点并同时打开中断（这样设计的原因在于执行 eret 前现场已恢复，不能通过 mtc0 和 mfc0 开中断否则会破坏现场，而只能硬件开中断）。

3 条指令的格式等与标准 MIPS-CPU 一样，只是 CP0 中专用寄存器的编号不同，

本设计将所有的偶数号专用寄存器均指向 IE-INM 寄存器，将所有的奇数号专用寄存器均指向 EPC 寄存器。

2.3 流水 CPU 设计

2.3.1 总体设计

流水线的原理实际上是时间并行，即把任务分成若干子任务，使子任务在流水线的各阶段并发地执行。

因此将指令的执行周期划分为 5 个阶段：取指 IF 阶段、译码 ID 阶段、执行 EX 阶段、访存 MEM 阶段和写回 WB 阶段。各阶段之间设置接口部件，负责传递代加工的数据、控制数据加工的控制信号和反馈信号等。

在本设计中，所有的分支指令或者跳转指令均在执行 EX 阶段跳转，所以所有指令至少经过 5 个阶段中的前 3 个阶段。实际上，只有几条分支指令只经过 IF、ID 和 EX 阶段，其余指令均需要经过所有的阶段。

2.3.2 流水接口部件设计

指令的执行周期被划分为 5 个阶段，两个阶段之间需要设置一个接口部件，因此需要设置 4 个接口部件：IF/ID 接口部件、ID/EX 接口部件、EX/MEM 接口部件以及 MEM/WB 接口部件。

接口部件的本质是寄存器，用于传递与指令相关的数据信息、控制信息和反馈信息，后续部件对数据的加工处理依赖于前阶段传递过来的信息。

2.3.3 理想流水线设计

首先，要消除数据通路中的资源相关。由于在设计单周期 CPU 时已将指令存储器和数据存储器分离，分支地址的计算和运算指令指定的计算采用不同的加法器和运算器，所以这里不用考虑“取操作数与取指令都需要访问主存”和“运算器 ALU 的复用”的资源相关问题。

然后，按照总体设计中的描述，将指令的执行周期划分为 IF、ID、EX、MEM 和 WB 共五个阶段。

最后，在不同阶段之间设置换中接口部件，并确定每个接口部件需要传递的具体内容。需要注意的是：应该在 ID 段译码生成该指令的所有控制信号，并且将所有的控制信号向后传递，所以后续部件控制信号不再单独生成。除此之外，单周期 CPU 实现中实现的控制器可以在 ID 段复用。

2.4 气泡式流水线设计

气泡流水线主要是为了解决理想流水线中存在的相关冲突问题，主要包括数据相关和分支相关。

对于数据相关，其产生的根本原因是：后一条指令的操作数依赖于前一条指令的执行结果，通常为后一条指令要读取前一条指令需要写入的寄存器。由于寄存器的读取发生在 ID 译码阶段，而写回发生在 WB 写回阶段，所以后一条指令在 ID 阶段读取的数据并不一定是最新的数据。

为了消除数据相关，需要在 ID 段进行数据相关的检测，即检测 ID 阶段指令与 EX 阶段、MEM 阶段和 WB 阶段的指令是否存在数据相关。由于不同类型的指令需要读入的寄存器的个数和要求不完全相同，所以针对不同的指令要区分检测。若检测到数据相关，则需要向流水线后段插入气泡，同时向前给出流水线阻塞信号以避免当前指令被新指令取代。气泡要一直插入到整条流水线中不存在数据相关为止。

对于分支相关，其产生的根本原因是：指令跳转造成了流水线误取了分支指令或者跳转指令后面的两条指令。因为理想流水线中 PC 寄存器的数据来源总是 $PC+1$ ，即 CPU 默认顺序执行，所以当指令发生跳转时会造成指令的误取。

为了消除分支相关，首先需要对分支相关进行检测。由于本 CPU 的设计中分支指令和跳转指令的执行在 EX 执行阶段，所以分支相关的检测也放到了 EX 执行阶段。当检测到需要进行跳转（分支指令判断条件为真）时，向前段发出清零信号，清除误取的指令，同时保证后续指令继续执行。

气泡实际上等同于空操作，而在硬件中的实现方法则是将接口部件中的全部寄存器清零（最关键的是写回信号）。

2.5 数据转发流水线设计

对于气泡流水线：若流水线中产生了数据相关，则需要插入 1~3 个气泡（对于 ID 阶段和 EX 阶段的数据相关需要插入 3 个气泡）；若流水线中产生了分支相关，则等同于插入 2 个气泡，因为在 EX 阶段执行跳转的误取深度为 2。显然，这样的气泡机制会向流水线中插入大量的气泡，即执行大量的无意义的空操作，导致流水线的效率极低。

为了进一步优化，首先，将寄存器堆写入和读出过程进行分离。即寄存器堆并

不是同时读写，而是先写后读，这样在 ID 阶段和 WB 阶段的指令产生数据相关时，便没必要再插入气泡。具体的实现方法是将寄存器堆的写操作调整为时钟的下降沿触发，这样可以保证 ID 阶段的指令在向后传递数据传递的是 WB 阶段指令刚刚写回的数据。

然后，对大部分数据相关的情形，进行数据的重定向。数据相关的检测仍放在 ID 阶段，当检测到 ID 阶段指令与 EX 阶段、MEM 阶段的指令发生数据相关时，首先判断是否可以重定向，如若可以，则根据数据相关发生的阶段确定数据重定向的来源并传给 EX 阶段，当指令到达 EX 阶段执行时，根据 ID 阶段传来的数据重定向的来源直接读取数据。

需要注意的是，对于 Load-Use 类型的数据相关，无法进行数据重定向（否则会导致 EX 执行阶段的路径时间过长），只能向后插入一个气泡。

2.6 流水线中断与动态分支预测机制设计

2.6.1 流水线中断设计

流水线中断的设计和单周期 CPU 的中断机制设计的设计思想和设计方法基本相同，但在流水线中断机制的设计过程中，有两个问题需要额外考虑和确定。

第一个问题是当中断到来时，在哪个阶段响应中断，也就是记录哪个阶段的 PC 作为断点。考虑到本 CPU 设计中指令的跳转是在 EX 执行阶段，而且断点应该是返回来继续执行的位置，所以本设计决定在 EX 阶段响应中断，即中断到来时记录 EX 阶段的 PC 值作为断点。

第二个问题是保证断点处不是气泡。由于指令的跳转有可能会清空前面的指令，所以，执行阶段的指令有可能是无任何意义的气泡。气泡对应的 PC 和 IR 都是常数 0，显然这不是一个合理的断点。考虑到这种情况，为中断处理模块 INTR 添加一个 Res 中断响应信号，当 EX 阶段的指令为气泡时，Res 信号为 0 表示该中断在当前时钟周期不能被响应，需要维持到下一个时钟周期再响应。如此设计，便可以保证断点处一定不是气泡。

2.6.2 动态分支预测机制

动态分支预测机制最重要的便是设计和实现 BHT (Branch History Table) 表，

该表共含有 8 个表项，每一个表项包含 Valid 位、PC 值、BranchAddr 值、双预测位和 LRU 调度标记。其中，Valid 位表示当前表项是否有效，PC 值和 BranchAddr 值分别表示当前表项预测的 PC 和对应的跳转地址，双预测位和 LRU 调度标记下面会详细介绍。

预测的思路是：取指令阶段，以 PC 为关键字到 BHT 表做全相联比较查找，如果命中，直接根据预测位给出下一条指令正确地址；执行阶段遇到跳转指令，若 BHT 命中，根据跳转情况更新预测位，LRU 调度标记清零，若 BHT 不命中，将对应指令信息放入 BHT 表。

BHT 表项中的双预测位是一种状态机的思想，用 00、01、10、11（二进制）分别代表强不选择（strongly not taken）、弱不选择（weakly not taken）、弱选择（weakly taken）和强选择（strongly taken），状态机的状态跳转如图 2.2 所示。之所以这样设计，是为了避免特殊情况下来回跳转造成的读写颠簸现象。

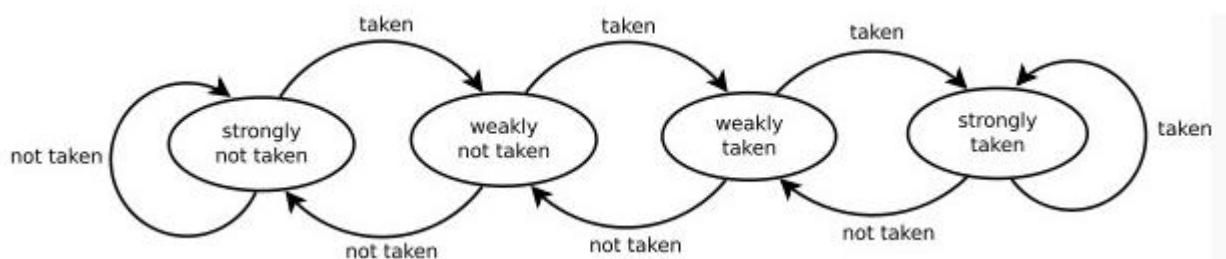


图 2.2 双预测位状态机的状态跳转

BHT 表项中的 LRU 调度标记位则体现了 LRU 调度算法。每个时钟周期，所有 BHT 表项的 LRU 调度标记位都会自动加 1，只有到 BHT 命中且预测成功时才会再 EX 阶段将对应 BHT 表项的 LRU 调度标记位清 0。这样，每次需要写入 BHT 表时，寻找 LRU 调度标记位最大的表项（最久未使用的）写入即可。

3 详细设计与实现

3.1 单周期 CPU 实现

3.1.1 主要功能部件实现

1. 程序计数器 (PC)

① Logisim 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（四路选择器的输出），输出为当前执行指令的地址，使能端位 PCEn 信号，清零端为 0。Clk 为停机信号 Halt 通过非门取反之后与 Logisim 自带时钟 ComClk 相与的结果。当需要进行停机时，Halt 控制信号为 1，经过非门之后为 0，与时钟信号相与，屏蔽时钟信号，使整个电路停机。PC 电路图如图 3.1 所示。

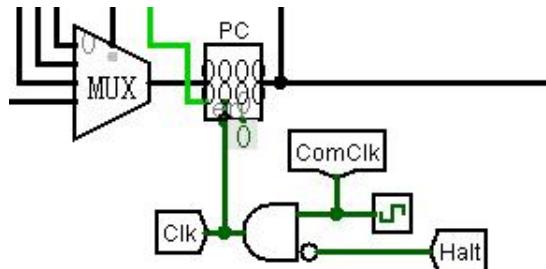


图 3.1 程序计数器 (PC)

② FPGA 实现:

程序计数器 PC 本质上是一个寄存器，对应的 Verilog 代码如下：

```
module register
  #(parameter WIDTH = 32)
  (
    input [WIDTH-1:0] din,
    input we,
    input rst,
    input clk,
    output reg [WIDTH-1:0] dout
  );
  always@(posedge clk or posedge rst)
  begin
```

```

if(rst)
begin
    dout <= 0;
end
else if(we)
begin
    dout <= din;
end
end
endmodule

```

2. 指令存储器 (IM)

程序存储器采用的是只读 ROM，其数据位宽为 32 位，但是地址为宽仅为 10 位，另一方面 PC 寄存器却是一个 32 位寄存器，所以本设计方案中，访问 ROM 时只取 PC 的低 10 位作为 10 位地址（访问数据 ROM 也采用同样的策略）。

程序存储 ROM 是按字（4 字节）访问，所以不像一般的 MIPS 架构的 CPU 那样 PC 每次加 4，PC 在这里需要每次加 1，而且除此之外，许多跟地址跳转有关的指令的具体操作也跟传统 MIPS 体系 CPU 的操作有所不同。

① Logisim 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。按照前述讨论，使用分线器取 32 位指令地址的 0-9 位作为指令存储器的输入地址，如图 3.2 所示。

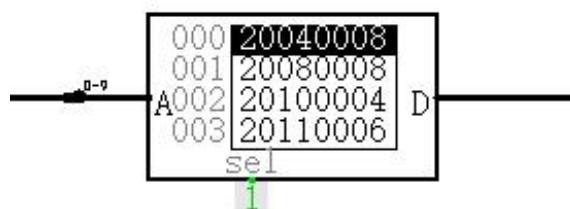


图 3.2 指令存储器 (IM)

② FPGA 实现：

指令存储器 IM 本质上是一个只读存储器，对应的 Verilog 代码如下：

```

module rom(
    input [9:0] addr,
    output [31:0] dout
);

```

```

reg [31:0] mem [0:1023];
assign dout = mem[addr];

initial
begin
    $readmemh("D:/Documents/verilog/cpu/testfile/benchmark.txt", mem);
end
endmodule

```

3. 运算器 (ALU)

① Logisim 实现:

在 Logisim 中，ALU 的设计主要可以分为两个部分：第一部分是设计并实现各个操作（乘除加减等）的电路，第二部分则是使用一个 多路选择器，实现给定操作码然后给出对应的操作结果的功能。其电路图如图 3.3 所示。

第一部分，对于除了加 (op 为 5) 和减 (op 为 6) 外，其余操作都可利用 logisim 自带运算器实现，因此实现起来相对简单。加法和减法均可通过一个加法器实现（先行进位加法器），其中减法采用补码运算。第二部分主要是两个多路选择器用来选择 result 和 result2 的来源，其中隧道“ROP i ”表示操作码为 i 的运算结果。

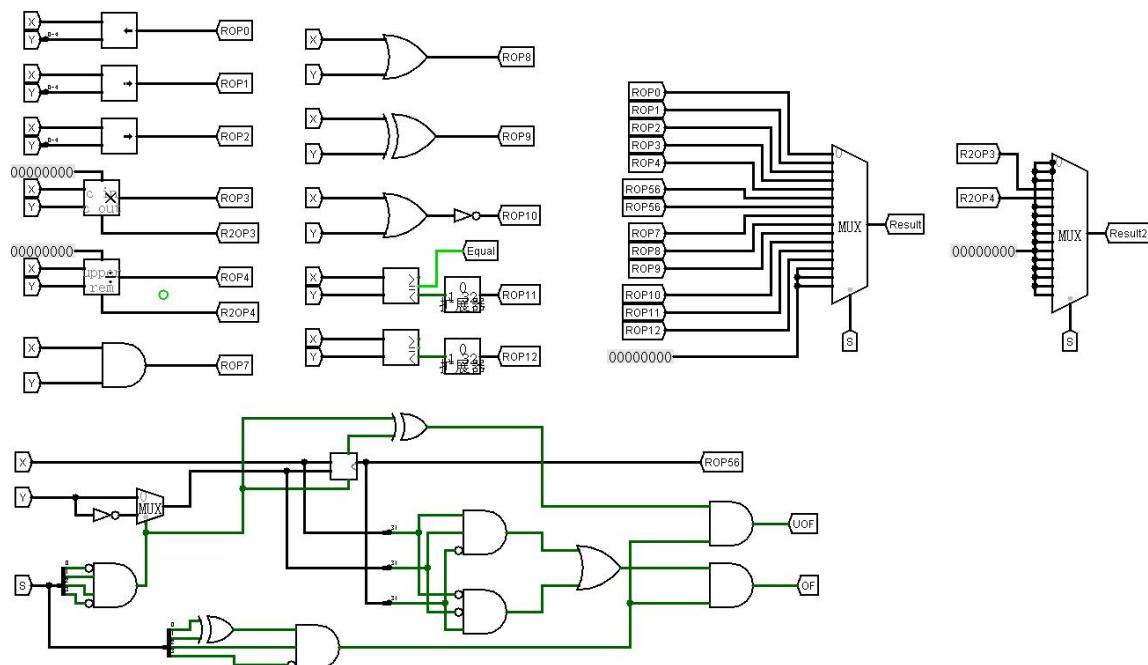


图 3.3 运算器 (ALU)

② FPGA 实现:

运算器（ALU）实际上是两个多路选择器，根据操作符输出对应的运算结果，对应的 Verilog 代码如下：

```
module alu(
    input [31:0] x,
    input [31:0] y,
    input [3:0] op,
    output [31:0] result,
    output [31:0] result2,
    output equal
);

    wire [63:0] product;
    assign product = x * y;

    assign result = (op == 4'h0) ? x << y[4:0] : 32'hz,
        result = (op == 4'h1) ? {{32{x[31]}}, x} >>> y[4:0] : 32'hz,
        result = (op == 4'h2) ? x >> y[4:0] : 32'hz,
        result = (op == 4'h3) ? product[31:0] : 32'hz,
        result = (op == 4'h4) ? x / y : 32'hz,
        result = (op == 4'h5) ? x + y : 32'hz,
        result = (op == 4'h6) ? x - y : 32'hz,
        result = (op == 4'h7) ? x & y : 32'hz,
        result = (op == 4'h8) ? x | y : 32'hz,
        result = (op == 4'h9) ? x ^ y : 32'hz,
        result = (op == 4'ha) ? ~(x | y) : 32'hz,
        result = (op == 4'hb) ? $signed(x) < $signed(y) : 32'hz,
        result = (op == 4'hc) ? x < y : 32'hz,
        result = (op >= 4'hd) ? 32'h0 : 32'hz;

    assign result2 = (op == 4'h3) ? product[63:32] : 32'hz,
        result2 = (op == 4'h4) ? x % y : 32'hz,
        result2 = (op <= 2 || op >= 5) ? 32'h0 : 32'hz;

    assign equal = (x == y);

endmodule
```

4. 寄存器堆（RF）

① Logisim 实现：

寄存器堆可以分为两大部分，其电路图如图 3.4 所示：

第一部分就是 32 个 32 位的寄存器，这 32 个寄存器按顺序依次编号，从 0 号到 31 号寄存器。需要注意的是，零号寄存器要求值恒为 0。寄存器堆应该为一个同步时序逻辑电路，所以所有的 32 个寄存器应该共用同一个时钟信号。

第二部分是控制部分，该部分控制选择写入哪个寄存器即写控制部分，读出哪两个寄存器的内容即读控制部分。此外根据要求，\$v0 和 \$a0 也应单独引出。

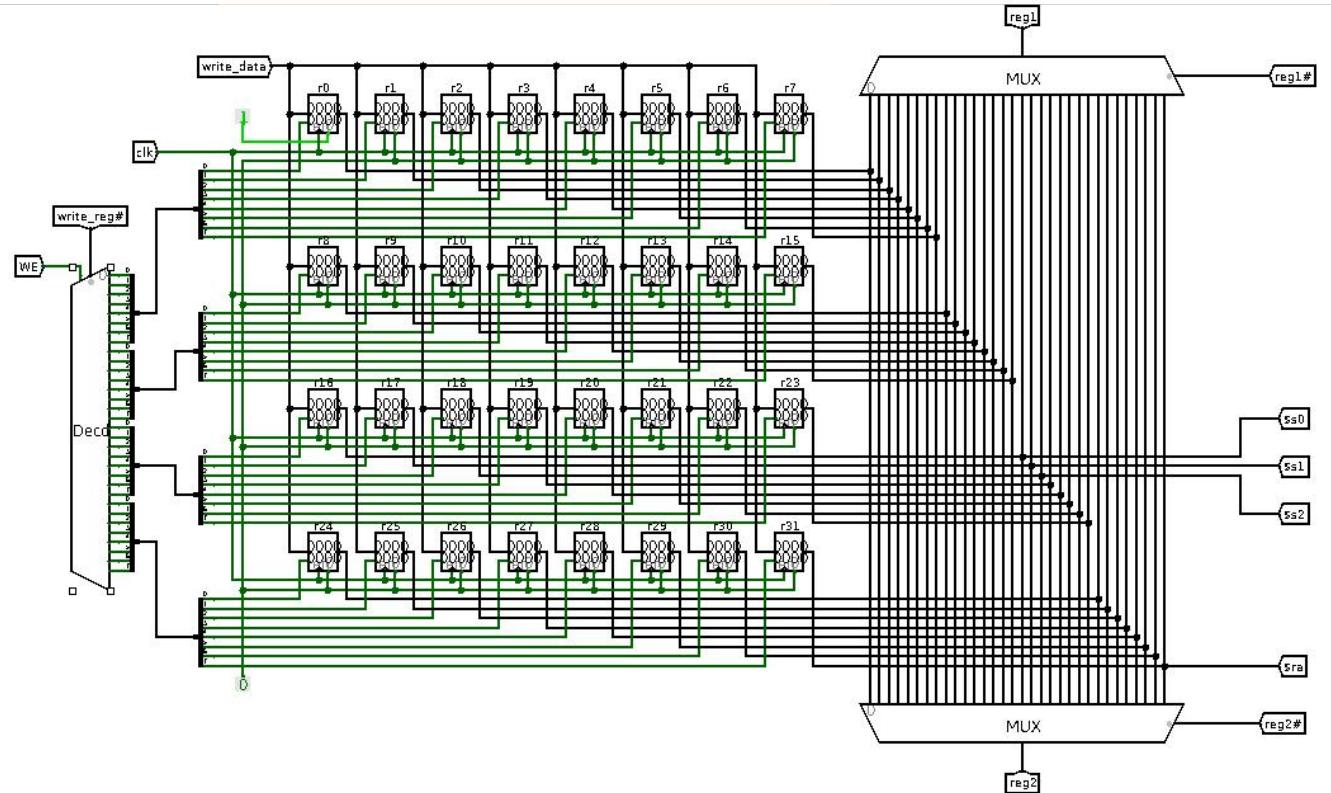


图 3.4 寄存器堆 (RF)

② FPGA 实现：

寄存器堆 RF 的核心是 32 个寄存器，可用 Verilog 的 reg 型变量实现。对于读控制部分，不需要时钟而直接读取，因此使用组合逻辑电路即可；对于写控制部分，则需要时钟沿触发。

```
module regfile(
    input [4:0] read_reg1,
    input [4:0] read_reg2,
    input [4:0] write_reg,
    input [31:0] write_data,
    input we,
    input rst,
    input clk,
```

```

        output [31:0] reg1,
        output [31:0] reg2,
        output [31:0] a0,
        output [31:0] v0
    );
    reg [31:0] regs [0:31];      // 32 32-bit register
    integer i;

    assign reg1 = regs[read_reg1];
    assign reg2 = regs[read_reg2];
    assign v0 = regs[2];
    assign a0 = regs[4];

    always @ (posedge clk or posedge rst)
    begin
        if (rst)
        begin
            for (i = 0; i < 32; i = i + 1)
            begin
                regs[i] = 0;
            end
        end
        else if (we && write_reg != 0)      // register 0 is always zero
        begin
            regs[write_reg] <= write_data;
        end
    end
endmodule

```

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的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 指令系统数据通路表所示。

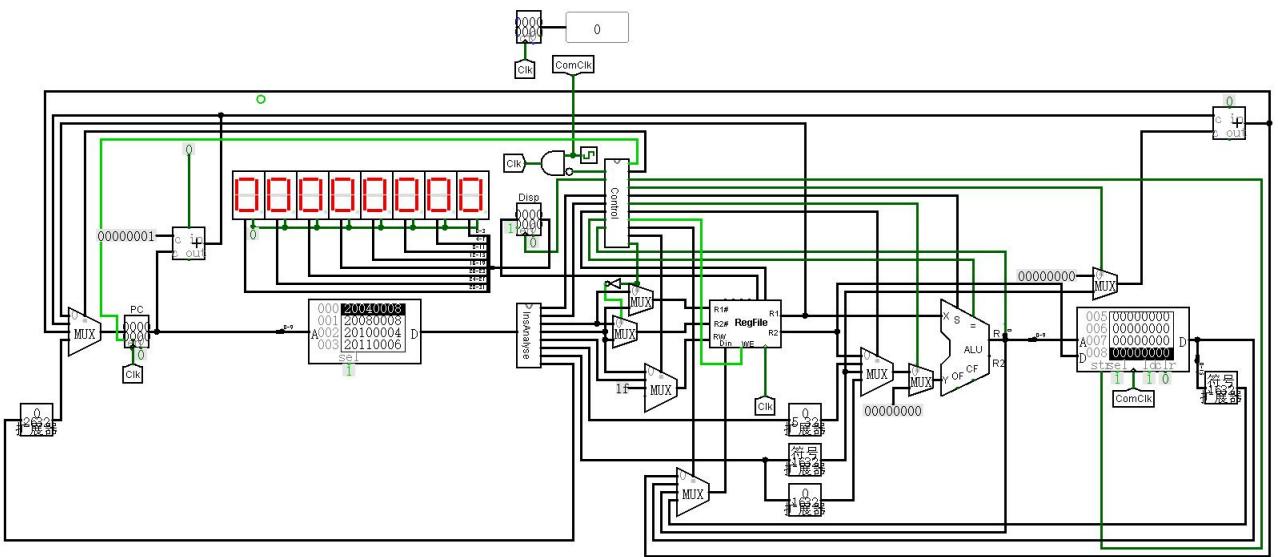


图 3.5 单周期 CPU 数据通路 (Logisim)

② FPGA 实现：

数据通路用 Verilog 实现相对复杂一些，对于图 3.5 中的电路图，主要部件可以用模块的实例化实现，部件之间的连线用 wire 类型变量实现，多路选择器用“assign”加三目运算符“?:”实现，扩展器用 Verilog 连接符“{}”实现。数据通路对应的部分 Verilog 代码（其余部分方法类似，篇幅原因不全部列出）如下。

```

module cpu(
    input ComClk, Rst,
    output [31:0] Digital_Tube,
    output [31:0] ClkCount
);
    // 部件之间的连线用 wire 变量
    wire [31:0] ALUY, RFD1, ALUResult;
    wire [3:0] ALUOP;
    wire Equal;
    // 多路选择器用 assign 加三目运算符实现
    assign RFDin = (RFDinSrc == 2'h0) ? PCAdd : 32'hz,
          RFDin = (RFDinSrc == 2'h1) ? DMDout : 32'hz,
          RFDin = (RFDinSrc == 2'h2) ? ALUResult : 32'hz,
          // 扩展器用连接符 “{}” 实现
          RFDin = (RFDinSrc == 2'h3) ? {{16{DMDout[15]}}, DMDout[15:0]} : 32'hz;
    // 主要部件用模块的实例化
    alu ALU(.x(RFD1), .y(ALUY), .op(ALUOP),
            .result(ALUResult), .equal(Equal));
endmodule

```

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 Logisim 和 Vivado 上进行控制器的具体实现。在设计控制器时，由于直接考虑的指令和各控点的信息，而不是具体的 OP 码和 Funct 值，所以在作表时，是直接作的指令和控点信息的关系表，如表 3.3 控制信号汇总表。

表 3.3 控制信号汇总表

指令	PCSrc		PCAddSrc	PCEn	RFR1Src	RFR2Src	RFRWSrc		RFDinSrc		RFWE	ALUYSrc		ALUYSrc2		ALUOP				DMWE
	M1	M0					M1	M0	M1	M0		M1	M0	M1	M0	OP3	OP2	OP1	OP0	
add	0	1	0	1	0	1	0	1	1	0	1	0	0	0	0	1	0	1	0	0
addi	0	1	0	1	0	1	0	0	1	0	1	1	0	0	0	0	1	0	1	0
addiu	0	1	0	1	0	1	0	0	0	1	0	1	1	0	0	0	1	0	1	0
addu	0	1	0	1	0	1	0	1	1	0	1	0	0	0	0	0	1	0	1	0
and	0	1	0	1	0	1	0	1	1	0	1	0	0	0	0	0	1	1	1	0
andi	0	1	0	1	0	1	0	0	0	1	0	1	1	0	0	0	1	1	1	0
sll	0	1	0	1	1	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0
sra	0	1	0	1	1	0	0	1	1	0	1	0	1	0	0	0	0	0	1	0
srl	0	1	0	1	1	0	0	1	1	0	1	0	1	0	0	0	0	1	0	0
sub	0	1	0	1	0	1	0	1	1	0	1	0	0	0	0	0	1	1	0	0
or	0	1	0	1	0	1	0	1	1	0	1	0	0	0	0	1	0	0	0	0
ori	0	1	0	1	0	1	0	0	0	1	0	1	1	0	1	0	0	0	0	0
nor	0	1	0	1	0	1	0	0	1	1	0	1	0	0	0	1	0	1	0	0
lw	0	1	0	1	0	1	0	0	0	0	1	1	1	0	0	0	1	0	1	0
sw	0	1	0	1	0	1	0	0	0	1	0	0	1	0	0	0	1	0	1	1
beq	equal	~equal	1	1	0	1	0	0	0	1	0	0	0	0	0	0	1	1	0	0
bne	~equal	equal	1	1	0	1	0	0	1	0	0	0	0	0	0	0	1	1	0	0
slt	0	1	0	1	0	1	0	1	1	0	1	0	0	0	1	0	1	1	0	0
slti	0	1	0	1	0	1	0	0	0	1	0	1	1	0	0	1	0	1	1	0
sltu	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	1	1	0	0	0
j	1	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
jal	1	1	0	1	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0
jr	0	0	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
syscall	0	1	0	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0
sllv	0	1	0	1	1	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0
sriv	0	1	0	1	1	0	0	0	1	1	0	1	0	0	0	0	0	0	1	0
lh	0	1	0	1	0	1	0	0	0	1	1	1	1	0	0	0	1	0	1	0
bgez	~judge	judge	1	1	0	1	0	0	0	1	0	0	0	0	1	1	0	1	1	0

(1) Logisim 实现：

在设计电路时，根据所作关系表，控制器需要知道指令信号（当前正在执行的指令的指令信号为 1，其余指令的指令信号均为 0）才能得到控点信息。但是，控制器的输入是指令解析结果的一部分字段，即 op 和 funct 字段然后再加上一些判断信息。要通过这些信息得到对应的指令信号，只能通过译码器，将 op 和 funct 作为译码器的选择信号，就可以得到一系列指令信号。但是这里需要注意的是 logisim 最大支持 5-32 译码器，但 op 和 funct 都是 6 位，所以需要用 1-2 译码器和 5-32 译码器结合使用。对应的转化电路如图 3.6 所示，从该电路中可知，当 op 为 0、funct 为 0 时，表明当前指令为 sll，这是正确的。显然，该电路可以将 op、funct 值转化为对应的指令信号。

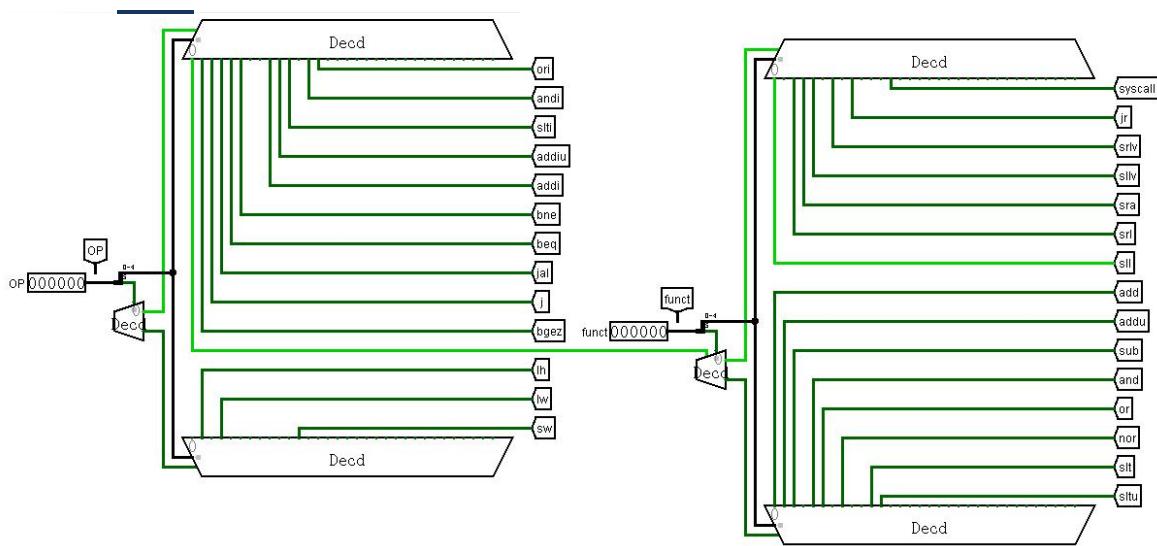


图 3.6 op、funct 转指令信号

前述对表 3.3 控制信号汇总表的分析，实质上都是在以行为单位（从指令的角度）分析表格。如果换个思路，以列为单位分析表，那么该表格还可以表明：对于某一指定控点，其在哪些指令时为 1，在哪些指令时为 0。根据这个信息，可以写出控点信息关于指令信号的逻辑表达式，从而得到控点信息的逻辑电路。由于每个控点都对应一个逻辑电路，故不一一给出每个控点的具体电路，而是只以 ALUYSrc 控点作为示例。

ALUYSrc 是一个四路选择器，故选择端包含两位 M1 和 M0。根据表 3.3 控制信号汇总表可知，M1 只有在当前指令为 addi、addiu、andi、ori、lw、sw 或 slti 时才为 1，故可以写出 M1 的逻辑表达式为：
 $ALUYSrc1 = addi + addiu + andi + ori + lw + sw + slti$ 。同理，也可以写出 M0 的逻辑表达式为：
 $ALUYSrc0 = andi + sll + sra + srl + ori$ 。最后，通过一个集线器汇总 ALUYSrc1 和 ALUYSrc0 便可以得到 ALUYSrc 控点的逻辑电路，如图 3.7 所示。

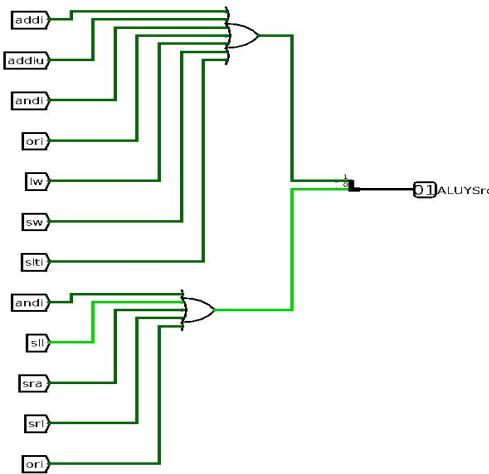


图 3.7 ALUYSrc 控点的逻辑电路

② FPGA 实现：

根据在 Logisim 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。但需要注意注意的地方与 Logisim 一样，首先需要将 OP、Funct 转化为对应的指令信号（wire 类型变量），具体的方法就是“assign”加三目运算符。然后，根据表 3.3 控制信号汇总表，写出控点信号对应指令信号的逻辑表达式，然后用 Logisim 的数据流建模实现即可。对应的部分 Verilog 代码（仍以 ALUYSrc 控点信号为例）如下。

```
// 声明指令信号
wire ins_ori, ins_andi, ins_slti, ins_syscall, ins_jr, ins_srlv;
// OP、Funct 转指令信号
// I 型指令
assign ins_ori = (op == 6'hd) ? 1 : 0;
assign ins_andi = (op == 6'hc) ? 1 : 0;
assign ins_slti = (op == 6'ha) ? 1 : 0;
// R 型指令
assign ins_syscall = (op == 6'h0 && funct == 6'hc) ? 1 : 0;
assign ins_jr = (op == 6'h0 && funct == 6'h8) ? 1 : 0;
assign ins_srlv = (op == 6'h0 && funct == 6'h6) ? 1 : 0;
// 生成控点信号
assign ALUYSrc =
{ins_addi | ins_addiu | ins_andi | ins_ori | ins_lw | ins_sw | ins_slti | ins_lh,
 ins_andi | ins_sll | ins_sra | ins_srl | ins_ori};
```

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。

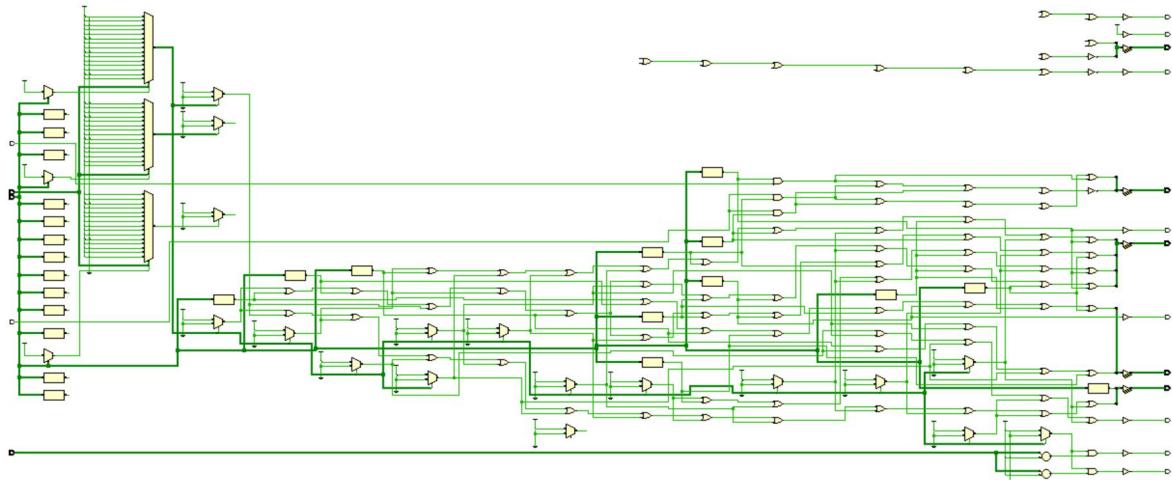


图 3.8 主控制器原理图

3.2 中断机制实现

3.2.1 总体设计

本设计实现的为多级嵌套中断。当低优先级中断的处理程序正在执行时，如果触发了一个高优先级中断，低优先级中断的处理程序应被打断转去执行高优先级中断的处理程序，执行完毕后接着继续执行低优先级中断的处理程序；当高优先级中断的处理程序正在执行时，如果触发了一个低优先级中断，低优先级中断应被“挂起”，当高优先级中断的处理程序执行完毕后才能去执行低优先级中断的处理程序。

根据多级嵌套中断的要求，设计添加中断功能 CPU 的执行流程如图 3.9 所示。

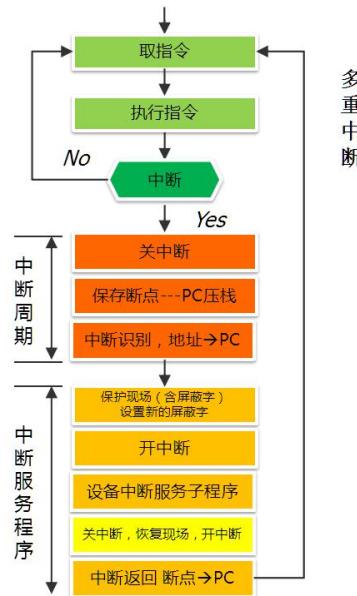


图 3.9 多重中断 CPU 执行流程

根据 2.2.1 总体设计中的阐述，为了达到设计目的，需要添加中断处理模块 INTR

(相当于简化版的“CP0”），还需添加两个（一个在 INTR 中）专用寄存器（简化版的 CP0 专用寄存器组）。

3.2.2 硬件设计

根据 2.2.2 硬件设计中的阐述，INTR 模块以中断源、使能-屏蔽字寄存器的输入及一些控制信号（例如时钟、清零等）作为输入，输出中断响应信号 INT（为 1 表示需要响应中断）、当前中断（如果有）的中断号能-屏蔽字寄存器的输出等。

根据总体设计的要求，INTR 模块要能借助中断屏蔽字实现中断的优先级，具体的硬件实现方法就是通过优先编码器 Pri。但考虑到中断屏蔽字，所以应将各中断的中断信号与对应中断屏蔽字相与后才能作为优先编码器的输入。除此之外，中断有可能因为优先级不够或者中断使能寄存器为 0 而“挂起”，所以应该还有一个中断等待信号 IRWTX，该信号在中断来临时置 1，响应中断后置零。

综上考虑，作出 INTR 模块的电路图如图 3.10 所示。

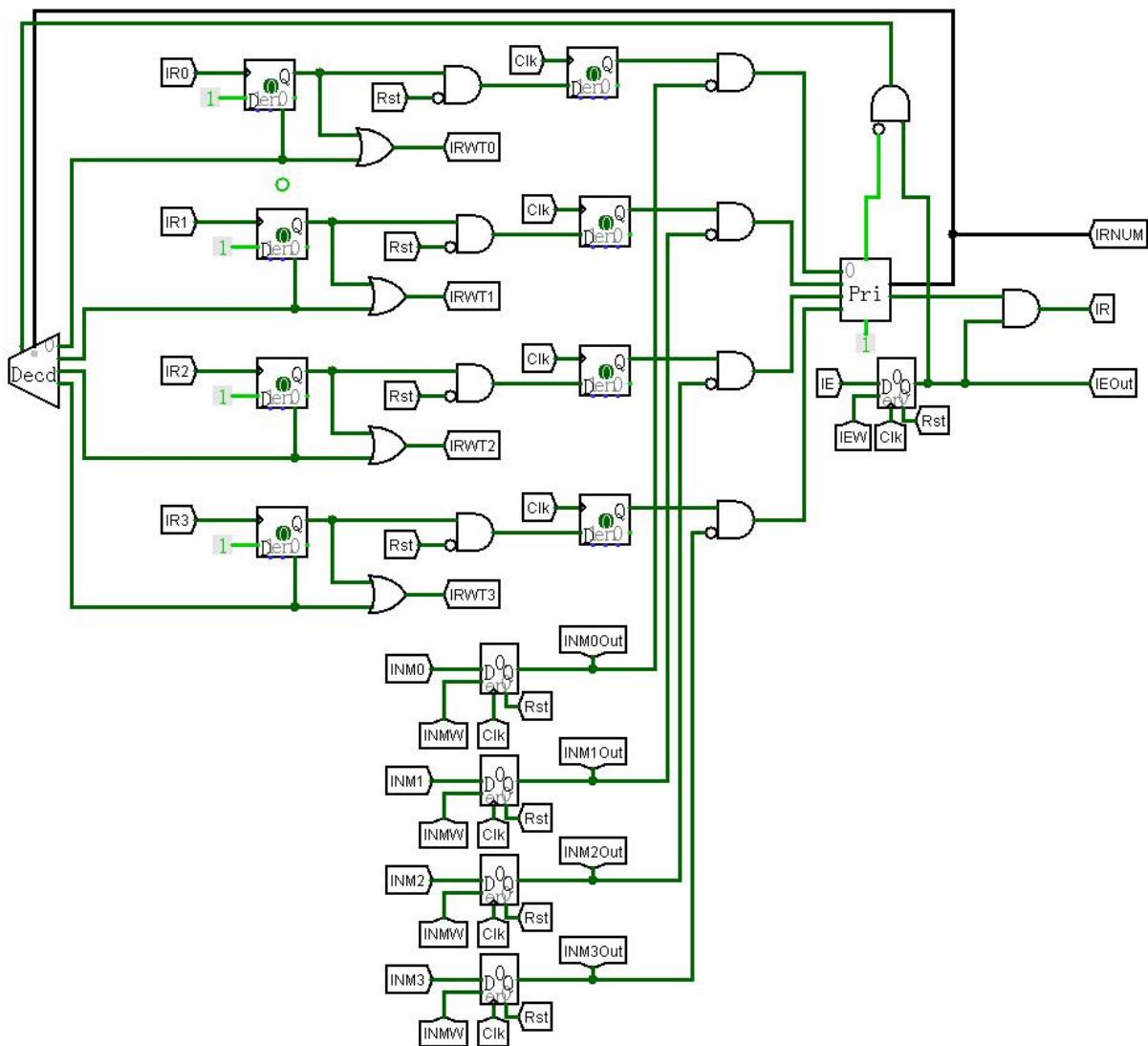


图 3.10 INTR 模块电路图

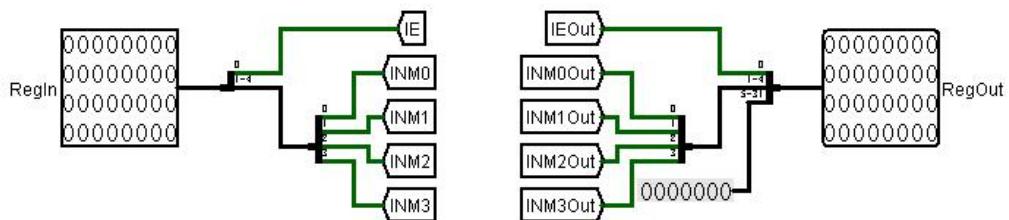


图 3.11 中断使能-屏蔽字寄存器

在图 3.10 中，是通过四路选择器根据中断号决定清零哪一个中断等候信号。除此之外，对于中断信号的处理则是采用了两级缓存，首先是由中断源自身触发第一级寄存器（D 触发器），然后由时钟触发第二级寄存器（D 触发器）才能产

生中断信号，再与中断屏蔽字相与作为优先编码器的输入。

在硬件上，为了使得 CPU 能够支撑中断，除了添加中断响应和处理模块 INTR 外，还需再添加两个专用寄存器——中断使能-屏蔽字寄存器与断点 PC 寄存器 EPC，需要注意的是中断使能-屏蔽字寄存器被封装在了 INTR 内部，如图 3.11 所示。

在图 3.9 中，“中断周期”中的 3 项操作——关中断、保存 PC、中断识别及服务程序入口地址放入 PC，均属于硬件完成，即不需要任何指令，在中断到来时硬件便会执行这 3 项操作，也就是所谓的“中断隐指令”。在本设计中，关中断是将中断使能-屏蔽字寄存器的使能位置 0，保存 PC 是将中断信号 INT 到来时 PC 的值保存至寄存器 EPC 中，中断识别即服务程序入口地址放入 PC 则是读取数据存储器地址为 $1020+ \text{中断号}$ 处的内容（在程序的初始化阶段，会将各中断服务程序的入口地址依次放入数据存储器 IM 的低端）放入 PC。

3.2.3 软件设计

在图 3.9 中，“中断服务程序”中的各项操作则均属于软件完成，包括：开、关中断，保存 EPC 和中断使能-屏蔽字寄存器的值，设置新的屏蔽字，中断返回等操作。

“中断隐指令”和“中断服务程序”中均有关中断的操作，但前者完全是硬件完成，而后者则是通过指令操作硬件实现。为了支持中断服务程序中涉及的各项操作，必须扩展单周期 CPU 的指令集，扩展：mfc0、mtc0 和 eret 三条指令。

mfc0 用于将 CP0 中某个寄存器的值读入到通用寄存器中，可用于保存上一中断的屏蔽字和当前中断的 EPC；mtc0 用于将 CP0 中通用寄存器的值写入到 CP0 的某个寄存器中，可用于设置新的屏蔽字；mfc0 和 mtc0 配合可以达到开关中断的功能；eret 用于从中断处理程序返回到断点，与图 3.9 稍有不同的是，eret 会同时打开中断（这样设计的原因在于执行 eret 前现场已恢复，不能通过 mtc0 和 mfc0 开中断否则会破坏现场，而只能硬件开中断）。

中断服务程序的示例代码如下。

```
int0:  
# 保护现场  
addiu $sp, $sp, -1  
sw $s0, ($sp)  
addiu $sp, $sp, -1  
mfc0 $s0, $8    # 保存上一中断的屏蔽字  
sw $s0, ($sp)
```

```
addiu $sp, $sp, -1
mfc0 $s0, $9      # 保存当前中断的 EPC
sw $s1, ($sp)
# 设置新的屏蔽字并开中断
addi $s0, $zero, 2
mtc0 $s0, $8
mfc0 $s0, $8
ori $s0, $s0, 1
mtc0 $s0, $8
# 设备服务程序
# 关中断
addi $s0, $zero, 2
mtc0 $s0, $8
# 恢复现场
lw $s0, ($sp)
mtc0 $s0, $9      # 恢复 EPC
addiu $sp, $sp, 1
lw $s0, ($sp)
srl $s0, $s1, 1
sll $s0, $s1, 1
mtc0 $s0, $8      # 恢复屏蔽字
addiu $sp, $sp, 1
lw $s0, ($sp)
# 中断返回并开中断
eret
```

3.2.4 最终设计

考虑了硬件设计扩展的 INTR 模块和两个寄存器，以及软件设计的指令扩展和中断服务程序后，还需考虑数据通路和控制器的修改。中断信号的引入和新的寄存器的加入，都需要控制器修改其所给出的控制信号。例如，寄存器堆 RF 的 Din 口数据来源就新添了两个——EPC 的输出和中断-使能屏蔽字寄存器的输出，而 EPC 和中断-使能屏蔽字寄存器的数据来源也会有多个。

修改后的支持中断的 CPU（数据通路）如图 3.12 所示。

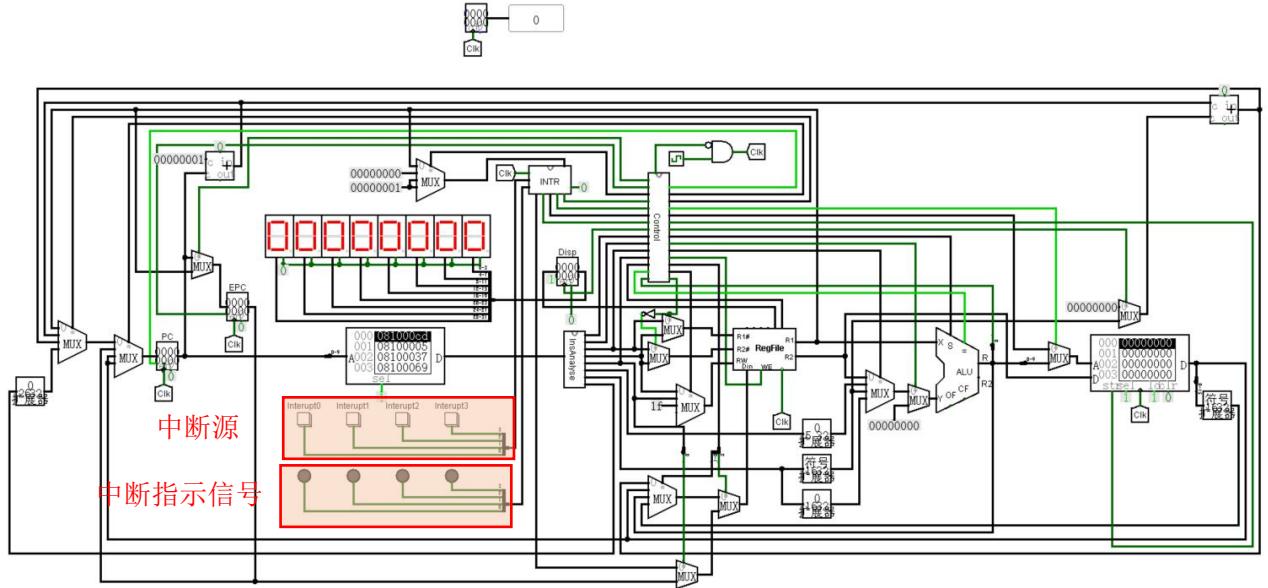


图 3.12 支持中断的 CPU (数据通路)

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

在 2.3 流水 CPU 设计中，已经说明，流水 CPU 中指令的执行周期被划分为 5 各阶段，对应的需要设计 4 各接口部件：IF/ID、ID/EX、EX/MEM 以及 MEM/WB 接口部件，用于传递与指令相关的数据信息、控制信息和反馈信息。

IF/ID 接口部件包含的数据信息、控制信息和反馈信息见表 3.4 IF/ID 接口部件定义

表 3.4 IF/ID 接口部件定义

接口传递的信息	位宽	类型	说明
PC_PLUS_1	32	数据	PC+1 的值
PC	32	数据	PC 的值
IR	32	数据	IR (指令) 的值

ID/EX 接口部件包含的数据信息、控制信息和反馈信息见表 3.5 ID/EX 接口部件定义

表 3.5 ID/EX 接口部件定义

接口传递的信息	位宽	类型	说明
Halt	1	控制	停机控制信号

接口传递的信息	位宽	类型	说明
Disp	1	控制	显示控制信号
PC_PLUS_1	32	数据	PC+1 的值
PC	32	数据	PC 的值
IR	32	数据	IR (指令) 的值
ALUOP	4	控制	ALU 的操作符
ALUYSrc	2	控制	ALU 第二个操作数选择端信号
ALUYSrc2	1	控制	ALU 第二个操作数选择端信号
RFD1	32	数据	寄存器堆 R1 口的输出
RFD2	32	数据	寄存器堆 R2 口的输出
Ext5to32	32	数据	5-32 无符号扩展器的输出
SExt16to32	32	数据	16-32 有符号扩展器的输出
Ext16to32	32	数据	16-32 无符号扩展器的输出
Ext26to32	32	数据	26-32 无符号扩展器的输出
DMWE	1	控制	数据存储器 IM 的写使能信号
PCAddSrc	1	控制	PC 加法器第二个加数的选择端信号
RFWE	1	控制	寄存器堆 RF 的写使能信号
RFDinSrc	2	控制	寄存器堆 RF 的 Din 口输入的选择端信号
BrIns	6	数据	跳转指令的指令信号总线
WReg	5	数据	寄存器堆 RF 的 Wreg 口输入

EX/MEM 接口部件包含的数据信息、控制信息和反馈信息见表 3.6 EX/MEM
接口部件定义表 3.5 ID/EX 接口部件定义

表 3.6 EX/MEM 接口部件定义

接口传递的信息	位宽	类型	说明
Halt	1	控制	停机控制信号
Disp	1	控制	显示控制信号
PC	32	数据	PC 的值
IR	32	数据	IR (指令) 的值

接口传递的信息	位宽	类型	说明
DMWE	1	控制	数据存储器 IM 的写使能信号
RFWE	1	控制	寄存器堆 RF 的写使能信号
RFDinSrc	2	控制	寄存器堆 RF 的 Din 口输入的选择端信号
ALU	32	数据	ALU 的运算结果
RFR2	32	数据	寄存器堆 RF 的 R2 口的输出
PCAdd	32	数据	PC 加法器的运算结果
WReg	5	数据	寄存器堆 RF 的 Wreg 口输入

MEM/WB 接口部件包含的数据信息、控制信息和反馈信息见表 3.7 MEM/WB
接口部件定义表 3.5 ID/EX 接口部件定义

表 3.7 MEM/WB 接口部件定义

接口传递的信息	位宽	类型	说明
Halt	1	控制	停机控制信号
Disp	1	控制	显示控制信号
PC	32	数据	PC 的值
IR	32	数据	IR (指令) 的值
RFWE	1	控制	寄存器堆 RF 的写使能信号
RFDinSrc	2	控制	寄存器堆 RF 的 Din 口输入的选择端信号
ALU	32	数据	ALU 的运算结果
DM	32	数据	数据存储器 IM 的输出
SExt16to32	32	数据	16-32 有符号扩展器 2 的输出
PCAdd	32	数据	PC 加法器的运算结果
WReg	5	数据	寄存器堆 RF 的 Wreg 口输入

根据上述 4 张接口部件的定义表，再使用 Logisim 自带的寄存器，便可实现 4 个接口部件。需要注意的是，接口部件中的所有寄存器在本理想流水 CPU 的设计中共用时钟、写使能端以及清零端信号。还有一个需要注意的地方为，Logisim 中的清零都为异步清零，为了到达同步清零的目的，须在原 Rst 端加一个 D 触发器。

3.3.2 理想流水线实现中

按照 2.3.3 理想流水线设计中的阐述，由于单周期 CPU 中已经考虑了资源相关，所以直接将指令的执行周期划分为 IF、ID、EX、MEM 和 WB 共 5 个阶段：将 PC 和指令存储器放到 IF 取指阶段，将控制器（单周期 CPU 的控制器可在此处复用）、指令解析器、寄存器堆 RF 以及一些符号扩展器放到 ID 译码阶段，将 ALU 放到 EX 执行阶段，将数据存储器 DM 放到 MEM 访存阶段。需要注意的是，由于寄存器的写发生在 WB 写回阶段，所以寄存器堆 RF 的写入寄存器编号 WriteReg、写入数据 Din 和写使能信号 WE 都应来自 WB 阶段的指令。由于控制器放到了 ID 阶段，所以在 ID 段译码生成该指令的所有控制信号，并且将所有的控制信号向后传递，后续部件控制信号不再单独生成。

综上所述，可在 Logisim 上实现理想流水线，如图 3.13 所示。

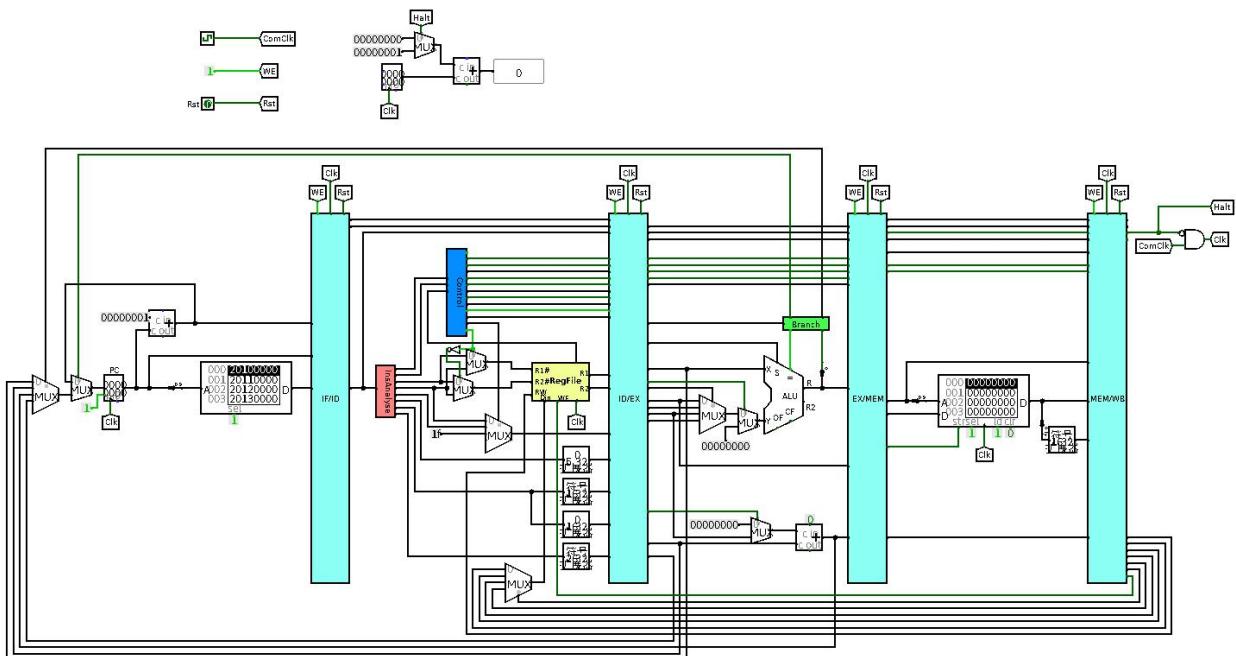


图 3.13 理想流水线电路图

3.4 气泡式流水线实现

在错误！未找到引用源。错误！未找到引用源。一节中提到过，气泡式流水线是为了解决理想流水线存在的相关冲突问题，主要是分支相关和数据相关。关于两个相关产生的原因和分析在错误！未找到引用源。一节已经阐述的十分详细了，本节只给出具体的设计方法。

为了消除数据相关，需要在 ID 段进行数据相关的检测，即检测 ID 阶段指令与

EX 阶段、MEM 阶段和 WB 阶段的指令是否存在数据相关。由于不同类型的指令需要读入的寄存器的个数和要求不完全相同，所以针对不同的指令要区分检测。

指令涉及到的寄存器个数和名称等情况见表 3.8 指令涉及的寄存器情况表所示。

表 3.8 指令涉及的寄存器情况表

指令	类型	操作数个数	涉及的寄存器
add	R	2	Rs, Rt
addu	R	2	Rs, Rt
and	R	2	Rs, Rt
sub	R	2	Rs, Rt
or	R	2	Rs, Rt
nor	R	2	Rs, Rt
slt	R	2	Rs, Rt
sltu	R	2	Rs, Rt
sllv	R	2	Rs, Rt
srlv	R	2	Rs, Rt
syscall	R	2	\$a0, \$v0
sll	R	1	Rt
sra	R	1	Rt
srl	R	1	Rt
jr	R	1	Rs
addi	I	1	Rs
addiu	I	1	Rs
andi	I	1	Rs
ori	I	1	Rs
lw	I	1	Rs

lh	I	1	Rs
slti	I	1	Rs
bgez	I	1	Rs
beq	I	2	Rs, Rt
bne	I	2	Rs, Rt
sw	I	2	Rs, Rt

在 ID 译码阶段进行数据相关检测时，根据指令类型，比较其涉及的寄存器的编号与 EX 阶段、MEM 阶段和 WB 阶段的 WriteReg，如果后三个阶段的 RFWE 为 1 且比较结果相等，则说明检测到数据相关，则需要向流水线后段插入气泡，同时向前给出流水线阻塞信号以避免当前指令被新指令取代。

硬件插入气泡的本质是清空对应阶段的接口部件，由于数据相关检测发生在 ID 译码阶段，所以若要向后插入气泡（1 个即可，一直有冲突就会一直插），需要清空的是 ID/EX 接口部件（Logisim 实现时给 ID/EX 一个 Rst 信号同步清零即可）。

阻塞信号本质上应该是控制的写使能端。为了避免当前指令被新指令取代，应该确保 PC 寄存器和 IF/ID 接口部件保持原内容不变。为此应该使阻塞信号接到 PC 寄存器和 IF/ID 接口部件的写使能端，当发生数据相关冲突时，阻塞信号置 0，从而保证 PC 寄存器和 IF/ID 接口部件不被新的指令写入。

在具体的 Logisim 实现时，将上述功能设计成了一个新的模块——DataHzd 模块，该模块位于 ID 译码阶段，负责数据相关冲突的检测和气泡、阻塞信号的产生。其部分电路图（产生气泡 Nop 和阻塞信号 Stop 部分）如图 3.14 所示。

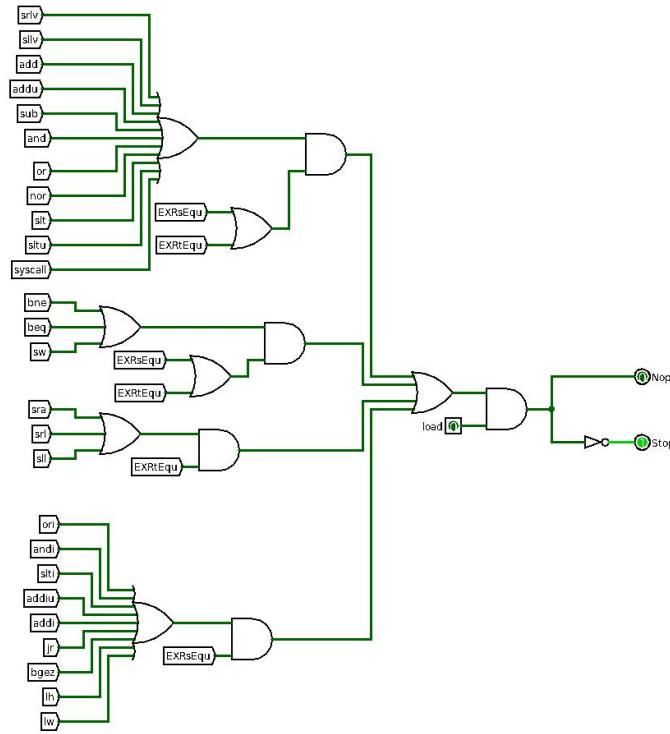


图 3.14 DataHzd 部分电路图

为了消除分支相关，首先需要在 EX 执行阶段堆分支相关进行检测。发生分支相关的情况有两类：一是，EX 阶段的指令为 `j`、`jal`、`jr` 这样的跳转指令；二是，EX 阶段的指令为 `beq`、`bne`、`bgez` 条件分支指令且满足分支条件。两类情况的区别在于后者需要额外的判断信息——ALU 给出的 Equal 信号和比较结果。当发生分支相关冲突时，需要向前段发出清零信号，清除误取的指令。具体来说，就是清空 IF 阶段和 ID 阶段误取的两条指令，具体的做法是将 IF/ID 和 ID/EX 接口部件同步清零。

在具体的 Logisim 实现时，将上述功能设计成了一个新的模块——Branch 模块，该模块位于 EX 执行阶段，负责分支相关冲突的检测和前半部分流水线清零信号的产生。其电路图如图 3.15 所示，该模块有三个输入两个输出 Equal 和 Judge 是由 ALU 产生的比较结果，BrPCSsrc 和 PCSrc 为 PC 寄存器数据来源多路选择器的控制信号，同时，PCSrc 也用作前半部分流水线的清零信号。

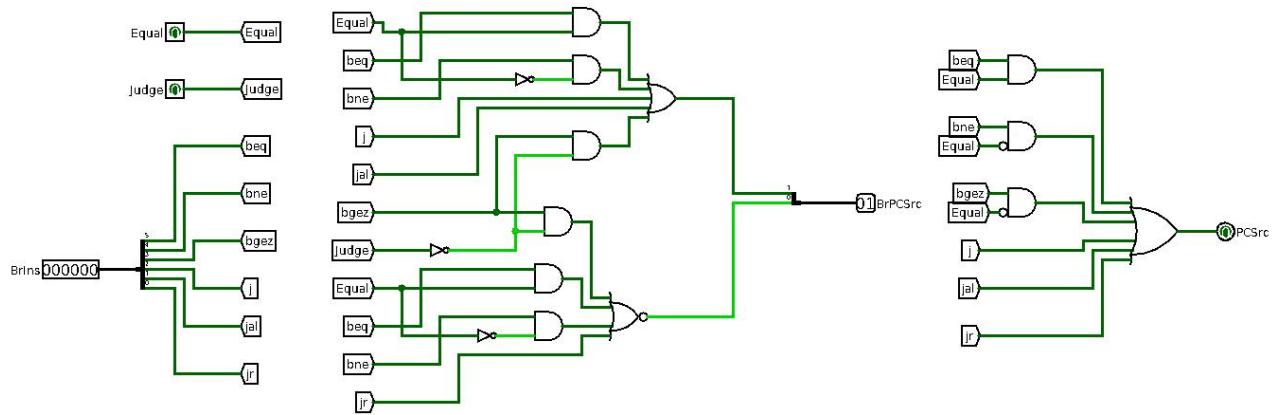


图 3.15 气泡流水线 Branch 电路图

在理想流水线中添加上述两个模块并适当修改，得到气泡流水线如图 3.16。

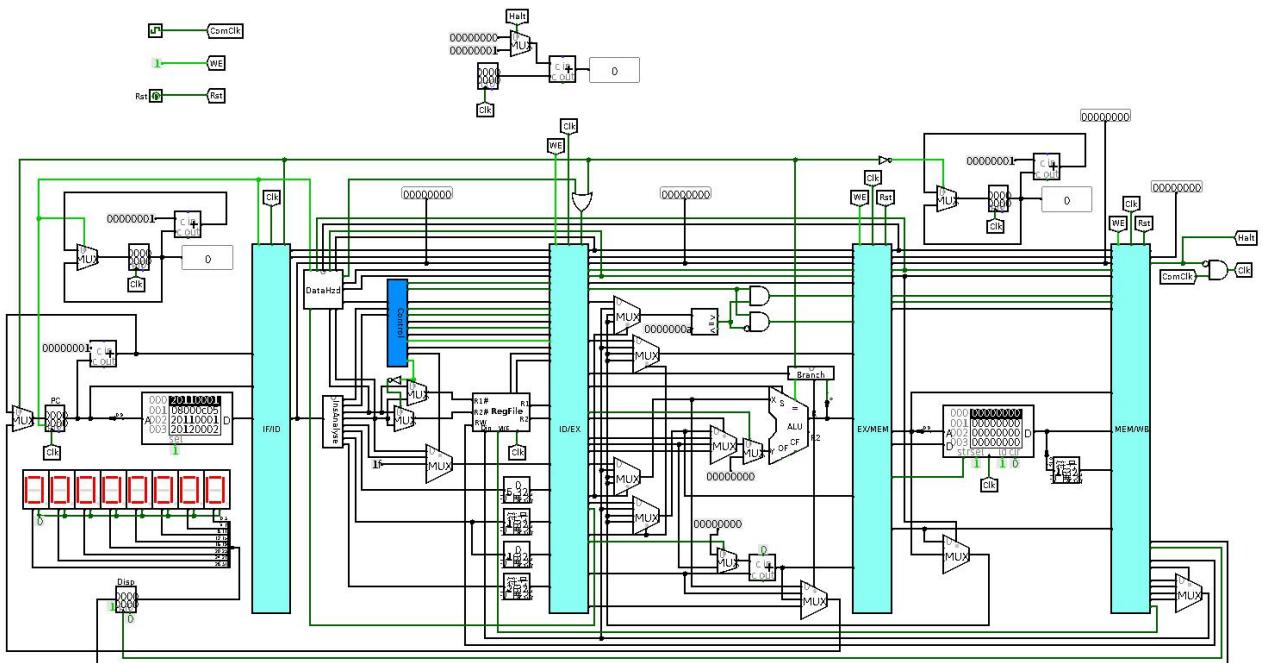


图 3.16 气泡式流水线

3.5 数据转发流水线实现

在 2.5 数据转发流水线设计中提到过数据转发流水线的意义和工作——优化。

首先，将寄存器堆写入和读出过程进行分离，在硬件电路中具体的实现方法是将寄存器堆 RF 的写操作调整为时钟的下降沿触发，这样可以保证 ID 阶段的指令在向后传递数据传递的是 WB 阶段指令刚刚写回的数据。

该步骤具体的实现方法很简单，Logisim 实现只需修改寄存器堆 RF 中所有寄存器的属性，将上跳沿写修改为下跳沿写；Verilog 则只需修改 regfile 模块中 always 循环的触发条件，将“posedge”改为“negedge”即可。

对于重定向流水线和气泡式流水线，差别在于它们解决数据相关冲突的方式上，在解决分支相关冲突上方法是相同的，所以气泡式流水线实现的 Branch 模块在重定向流水线中仍然适用。

然后，对大部分数据相关的情形进行数据的重定向（数据转发），这样可以避免气泡式流水线中插入过多的气泡。数据相关检测仍封装成 DataHzd 模块放在 ID 阶段，当检测到 ID 阶段指令与 EX 阶段、MEM 阶段的指令发生数据相关时，首先判断是否为 Load-Use 相关，如果不是 Load-Use 相关，则根据数据相关发生的阶段确定数据重定向的来源（本质上是多路选择器的选择信号）并传给 EX 阶段，当指令到达 EX 阶段执行时，根据 ID 阶段传来的数据重定向的来源直接读取数据；如果是 Load-Use 类型的数据相关，无法进行数据重定向（否则会导致 EX 执行阶段的路径时间过长），只能向后插入一个气泡，即将 ID/EX 接口部件清零。

① Logisim 实现：

在 Logisim 平台中考虑如何实现 DataHzd 模块，首先需要考虑一些预处理操作：EX_WReg、MEM_WReg 当且仅当对应使能信号 EX_WE、MEM_WE1 时有意义；

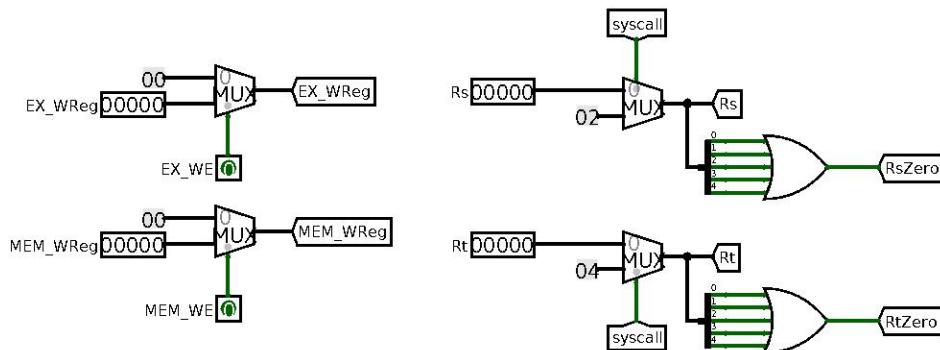


图 3.17 重定向流水线 DataHzd 预处理电路

指令涉及两个寄存器时，不一定是 Rs 和 Rt（syscall 指令涉及的两个寄存器分别是\$ra 和\$v0）。所以需要进行预处理得到对应的输入信号 EX_WReg、MEM_WReg、Rs、Rt 以及 RsZero、RtZero，预处理电路如图 3.17 所示。

判断 Rs、Rt 与 EX_WReg、MEM_WReg（预处理过的输入信号）是否相等，以及判断是否发生了 Load-Use 相关并产生气泡和停顿信号的电路如图 3.18 所示。

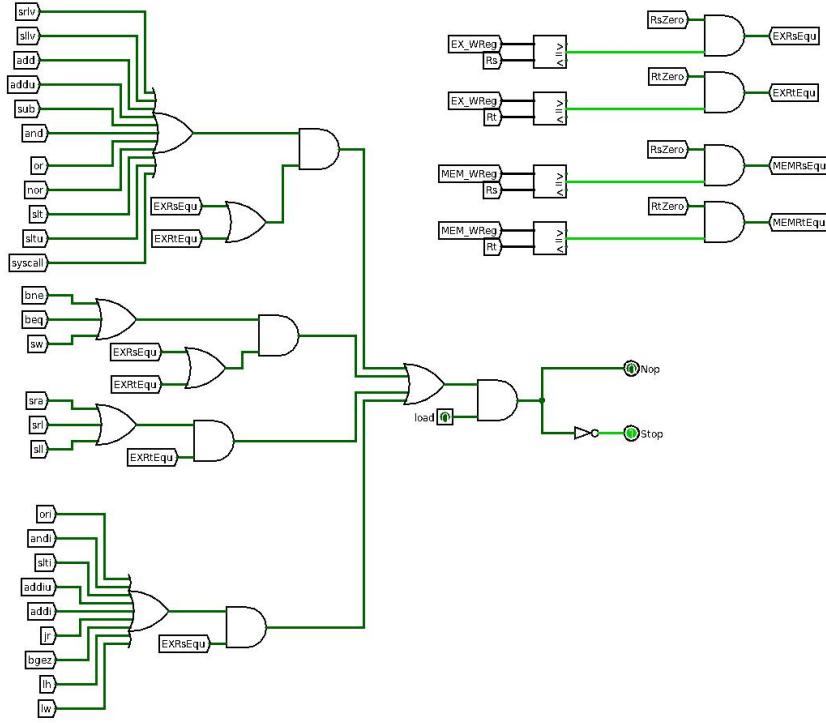


图 3.18 重定向流水线部分电路

重定向流水线 DataHzd 模块最关键的地方在于判断是否需要重定向以及重定向的数据来源。首先，考虑需要重定向的数据，即从寄存器中读出的 R1 和 R2，其即可能是 Rs 也可能是 Rt；然后，考虑重定向的数据来源，由于在 ID 阶段已经将 RF 的读写分离，说明指令最多只可能和其后两个阶段的指令发生数据相关，所以只需

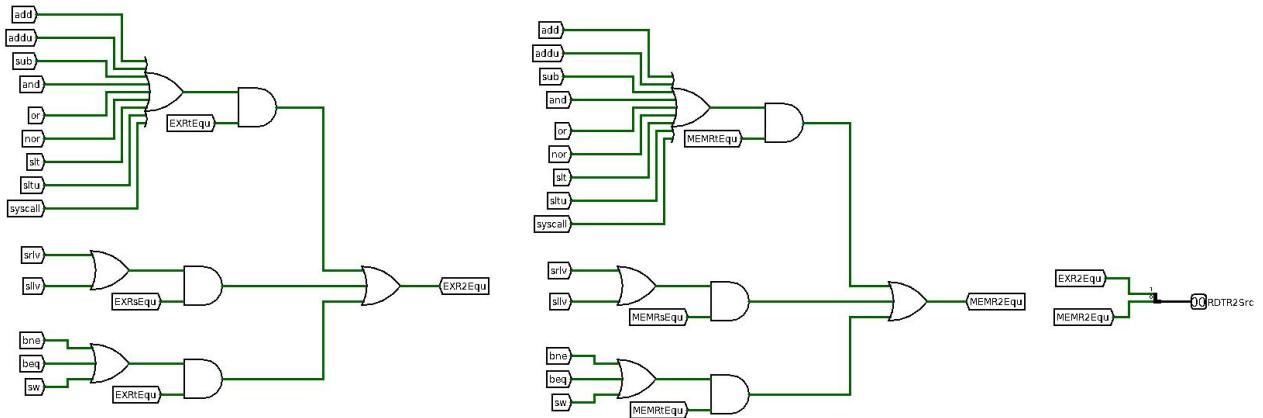


图 3.19 重定向数据来源确定电路

在 ID 阶段考虑 EX 执行阶段和 MEM 访存阶段即可，哪个阶段的指令与 ID 阶段的指令发生了数据相关，R1 和 R2 就重定向为哪个阶段的 RF_{DIN}。需要注意的是，如果 EX、MEM 两个阶段都有数据相关，则重定向 EX 阶段的 RF_{DIN}。这部分 Logisim 电路如图 3.19 所示，完成重定向流水线后的 CPU 电路如图 3.20 所示。

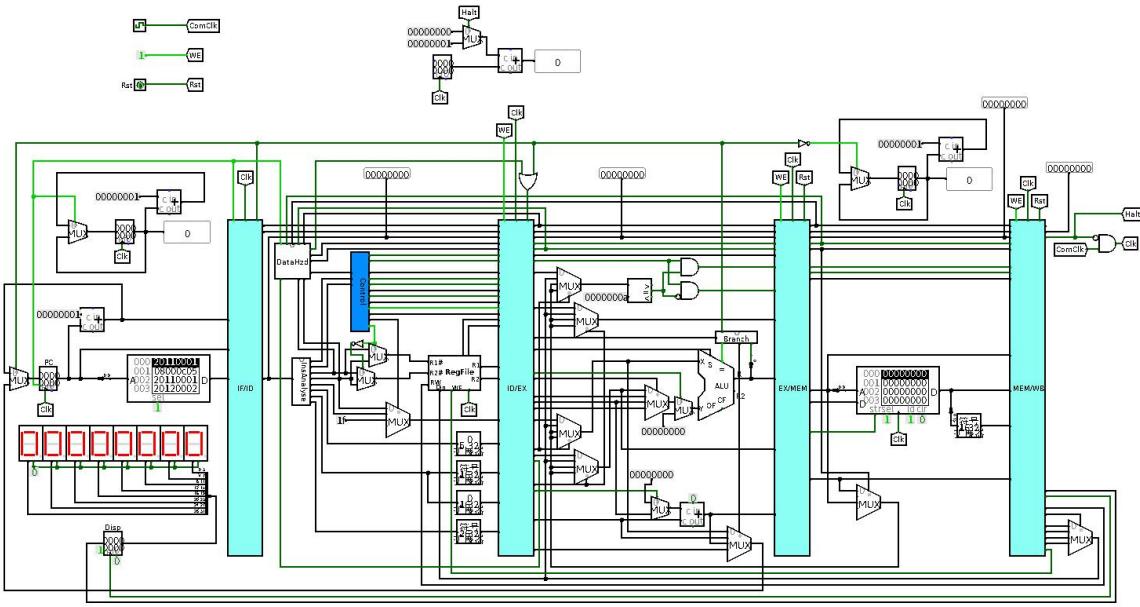


图 3.20 数据转发流水线

② FPGA 实现：

按照课设要求，重定向流水线是要求上板运行的。在 3.1 单周期 CPU 实现一节中已经详细介绍了 CPU 中主要功能部件的 FPGA 实现方法，这些主要功能部件都可以在重定向流水 CPU 中直接复用，除了寄存器堆 RF 需要将“posedge”修改为“negedge”。需要另外修改的是控制器 Controller 与数据通路，另外还需要添加对应的接口部件、Branch 模块和 DataHzd 模块。但控制器与数据通路的设计方法跟单周期 CPU 是一样的，所以本节不再累述，只对接口部件、Branch 模块和 DataHzd 模块的 FPGA 进行详细的阐述。

接口部件的实现极其简单，模块内部只是基本的寄存器部件（为了上板清零方便，除了寄存器原有的同步清零，又为其加了异步清零的端口）。模块的输入和输出分别对应所有寄存器的输入和输出，再加上同步时钟、清零、写使能和异步清零信号，以 IF/ID 接口部件为例，对应的 FPGA 代码如下：

```
module pipeline_if2id (
    input WE,      // 同步
    input Rst,
    input Clk,
    input Asyn_Rst, // 异步
    input [31:0] PC_PLUS_1_Din,
    input [31:0] PC_Din,
    input [31:0] IR_Din,
    output [31:0] PC_PLUS_1_Dout,
```

```

        output [31:0] PC_Dout,
        output [31:0] IR_Dout
    );
    pipeline_register #(32) PC_PLUS_1(.din(PC_PLUS_1_Din), .we(WE),
        .rst(Rst), .asyn_rst(Asyn_Rst), .clk(Clk), .dout(PC_PLUS_1_Dout));
    pipeline_register #(32) PC(.din(PC_Din), .we(WE), .rst(Rst),
        .asyn_rst(Asyn_Rst), .clk(Clk), .dout(PC_Dout));
    pipeline_register #(32) IR(.din(IR_Din), .we(WE), .rst(Rst),
        .asyn_rst(Asyn_Rst), .clk(Clk), .dout(IR_Dout));
endmodule

```

数据相关冲突处理部件 DataHzd 模块的实现参照图 3.17、图 3.18 和图 3.19，将对应的逻辑组和电路用 Verilog 的数据流级编程方式实现就很容易。例如，图 3.19 中部分电路图对应的 Verilog 代码如下：

```

assign EX_R2Equ = ((ins_srlv | ins_sllv) & EX_RsEqu) | ((ins_bne | ins_beq |
    ins_sw) & EX_RtEqu) | ((ins_add | ins_addu | ins_sub | ins_and | ins_or |
    ins_nor | ins_slt | ins_sltu | ins_syscall) & EX_RtEqu);
assign MEM_R2Equ = ((ins_srlv | ins_sllv) & MEM_RsEqu) | ((ins_bne | ins_beq |
    ins_sw) & MEM_RtEqu) | ((ins_add | ins_addu | ins_sub | ins_and | ins_or |
    ins_nor | ins_slt | ins_sltu | ins_syscall) & MEM_RtEqu);
assign RDTR2Src = {EX_R2Equ, MEM_R2Equ};

```

DataHzd 模块对应的其余部分按照类似方法实现即可，Branch 模块同理。

3.6 流水线中断与动态分支预测机制实现

3.6.1 流水线中断实现

在 2.6.1 流水线中断设计一节中讨论过流水线 CPU 中断，相比于单周期 CPU 中断需要额外考虑的问题有哪些：一是在哪个阶段相应中断，即记录哪个阶段的 PC 作为断点；二是如何保证断点处不是气泡。

除去上两个问题与数据通路中的部件安排问题，流水线 CPU 中断与单周期 CPU 中断基本上是一样的，所以在流水线 CPU 中复用了单周期 CPU 中的 INTR 模块，只是对其进行适当的修改。所以，流水线 CPU 中断的原理图完全可以按照单周期 CPU 中断的原理图实现。

针对第一个问题，2.6.1 节已经解释清楚，中断处理模块 INTR 在本设计中被放置到了 EX 执行阶段（INTR 中的中断使能-屏蔽字寄存器实际上相当于被放置到 ID 译码阶段），断点 PC 记录的是 EX 阶段正在执行的指令的 PC。

针对第二个问题，具体的解决方法是在 INTR 中添加 Res 响应信号，该信号与 EX_PC 和 EX_IR 有关。当且仅当 EX_PC 与 EX_IR 都为 0 时，Res 信号为 0 表示 EX 阶段正在执行的指令为气泡，其余情况 Res 信号都为 1 表示当前可以响应中断并进行中断处理。

① Logisim 实现：

首先修改 INTR 模块，添加（EX 阶段的）PC 和 IR 作为输入信号，然后（比较）生成 Res 信号，并参与相关逻辑控制，修改后的 INTR 模块的部分电路如图 3.21。

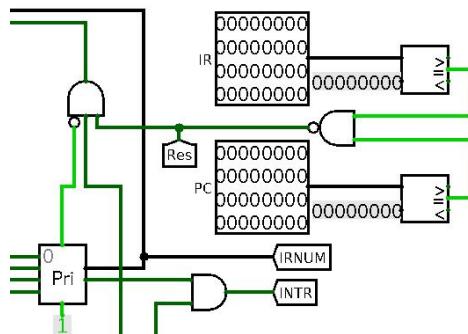


图 3.21 修改后的 INTR 模块的部分电路

然后，修改数据通路，在控制器 Controller 模块中添加相应的控制信号，得到最终的支持中断的重定向流水线 CPU 电路图如图 3.22 所示。

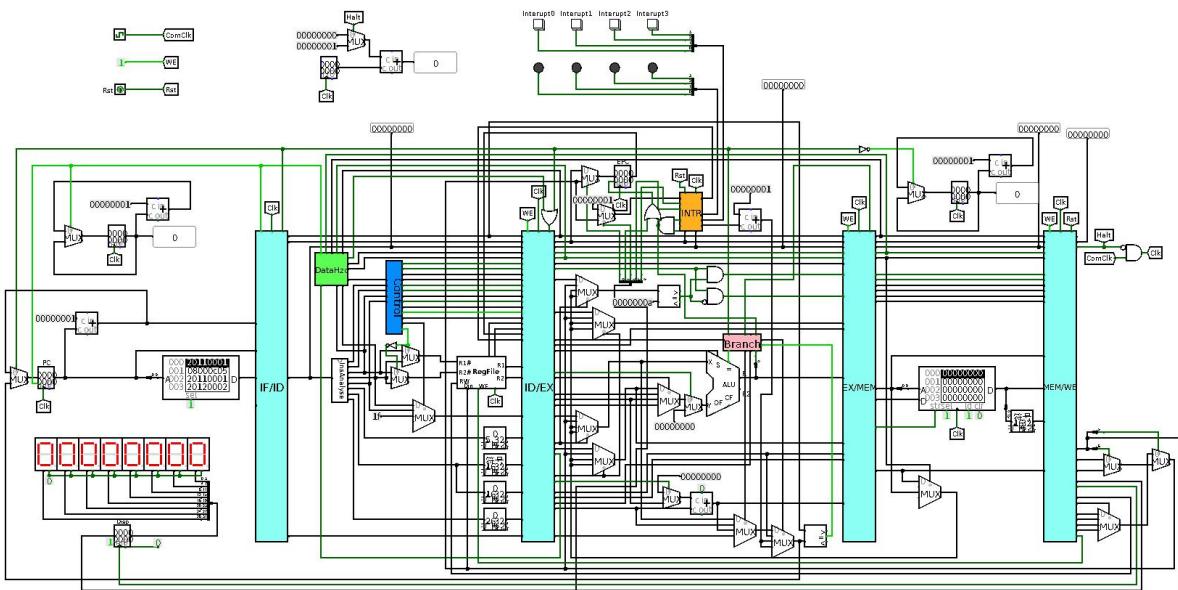


图 3.22 支持中断的重定向流水线 CPU

② FPGA 实现：

参照图 3.10 与图 3.22，利用 Verilog 的数据流级建模实现 INTR 模块并不困难，因为基本部件在前面的实现过程中都已经实现过，所以在 INTR 模块中直接使用就

可以。但有一点需要特别注意，在 Logisim 下的 INTR 模块中，为了缓存中断信号采用了两级只支持异步清零的 D 触发器，但在 Verilog 实现时，并没有实现 D 触发器这样的基本部件而实现的是寄存器部件，所以在用 Verilog 实现 INTR 模块，将 Logisim 中的 D 触发器分别用支持异步清零的和不支持的寄存器代替，INTR 模块中相关的 Verilog 代码如下（以 0 号中断为例）：

```
// 支持异步清零
pipeline_register #(1) IR0Cache1 (.din(1'h1), .we(~IR0Clr), .rst(IR0Clr),
                           .asyn_rst(Rst | IR0Clr), .clk(IR0), .dout(Cache1IR0));
assign IRWT0 = Cache1IR0 | IR0Clr;
// 只支持同步清零
register #(1) IR0Cache2 (.din(Cache1IR0 & ~Rst), .we(1'h1),
                           .rst(Rst), .clk(Clk), .dout(Cache2IR0));
register #(1) INM0Reg (.din(INM0), .we(INMW), .rst(Rst),
                      .clk(Clk), .dout(INM0Out));
```

与图 3.22 中 INTR 模块部分电路对应的 Verilog 代码如下：

```
assign Res = ~(IR == 32'h0 & PC == 32'h0);

assign ClrTemp = (IRNumber == 2'h0) ? 4'h8 : 4'hz,
       ClrTemp = (IRNumber == 2'h1) ? 4'h4 : 4'hz,
       ClrTemp = (IRNumber == 2'h2) ? 4'h2 : 4'hz,
       ClrTemp = (IRNumber == 2'h3) ? 4'h1 : 4'hz;
assign {IR0Clr, IR1Clr, IR2Clr, IR3Clr} =
       ((~enout & IEOut & Res) == 1'h1) ? ClrTemp : 4'hz,
{IR0Clr, IR1Clr, IR2Clr, IR3Clr} =
       ((~enout & IEOut & Res) == 1'h0) ? 4'h0 : 4'hz;
```

INTR 模块的其余部分实现方法与上述相同，不再累述。

3.6.2 动态分支预测机制实现

动态分支预测其实按照流水线的阶段来划分，可以分为两大部分：第一部分位于 IF 取指阶段，以 PC 为关键字到 BHT 表（BHT 表也放在了 IF 阶段）做全相联比较查找，如果命中，直接根据预测位给出下一条指令正确地址，同时把查询的结果通过接口部件传到 EX 阶段做进一步处理；第二部分位于 EX 执行阶段，如果正在执行的是跳转指令，若 BHT 命中，根据跳转情况更新预测位，LRU 调度标记清零，若 BHT 不命中，将对应指令信息放入 BHT 表，在具体实现时，这部分逻辑放到了 Branch 模块中实现。

① Logisim 实现:

首先考虑第一部分，最重要的就是要实现 BHT 表。在 Logisim 上实现 BHT 表与实现寄存器堆 RF 有些相似，不过 BHT 表是一个二维寄存器组，RF 只是一维的，所以 BHT 表的写控制电路相对复杂，需要将行控制信号（BHT 表序号）与列控制信号（各表项的写使能信号）相与。对于都控制电路，需要有一个 8 路比较器同时比较 8 个表项中的 PC 与查询 PC 是否相等，并用编码器将 8 路比较结果转化为命中表项号，并用多选择器输出其 BranchAddr 和相关信息，具体电路如图 3.23 所示。

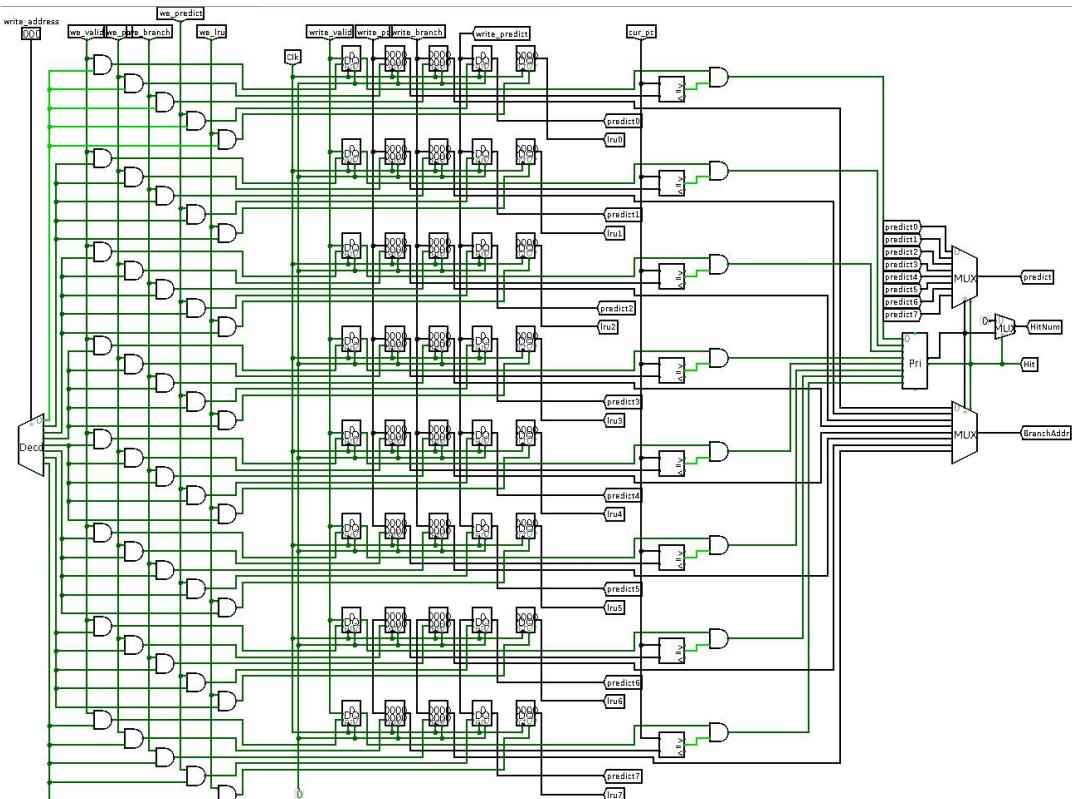


图 3.23 BHT 表主体电路

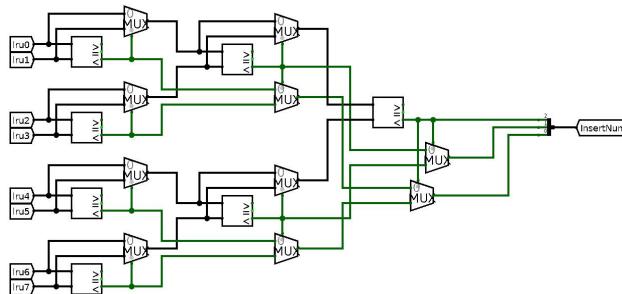


图 3.24 硬件实现 LRU 算法

BHT 表的另外一个难点在于硬件实现 LRU 算法，具体的实现方法是通过多路并行比较加二路选择器共同实现，电路如图 3.24 所示。lru0~lru7 表示的是 8 个表项对应的 lru 值，InsertNum 对应的是 lru 值最大的表项序号（不关心最大是多少），

而关心的是最大值出现在哪一个表项）。

考虑第二部分，即 Branch 模块中与 BHT 相关的逻辑电路。如果当前指令为跳转指令则需要向 BHT 表中写入相关信息，所以该模块需要给出 BHT 的各表项写使能信号和对应的数据来源，这一部分逻辑相对简单不具体介绍了。另外需要单独考虑的是 Branch 模块中如何实现 BHT 表中的双预测位状态机（如图 2.2 所描述），所谓状态机，本质上是根据当前的状态和输入确定下一状态，所以可以使用多路选择器实现“状态机”，具体电路如图 3.25 所示。PredictIn 表示的是命中表项当前双预测位的值，作为二路选择器的选择信号，PredictYes 表示的是分支指令的预测结果，Hit 表示 BHT 表是否命中，write_predict 表示的是要写入的双预测位的次态。

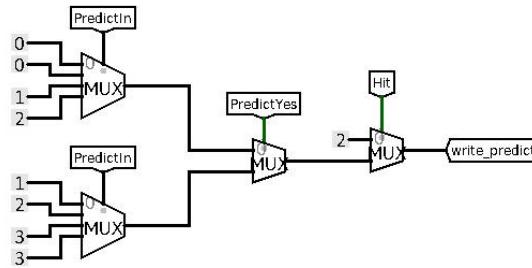


图 3.25 双预测位状态机电路实现

最后，修改数据通路，在控制器 Controller 模块中添加相应的控制信号，得到最终的支持动态分支预测和中断的重定向流水线 CPU 电路图如所示。

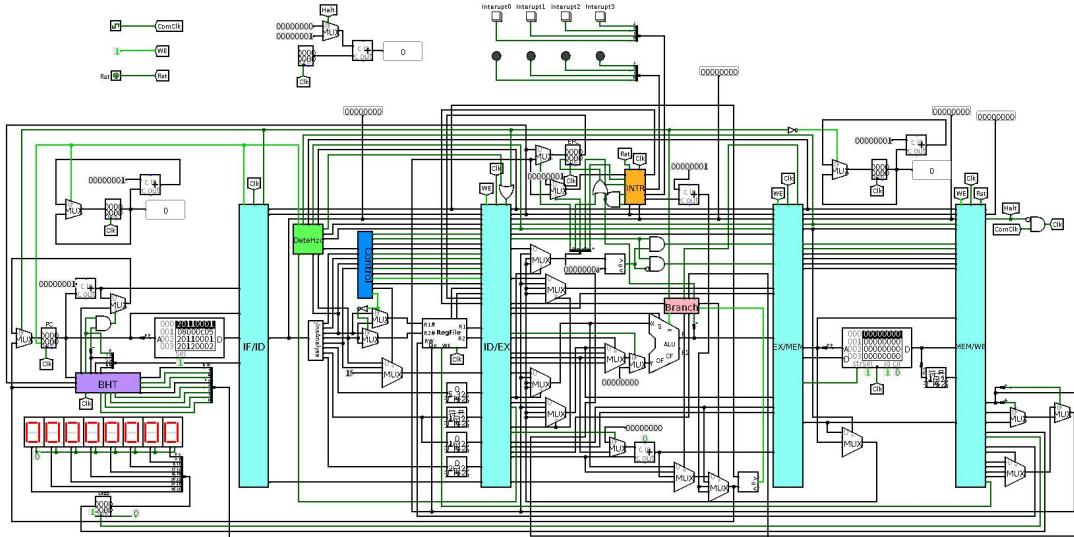


图 3.26 支持动态分支预测和中断的重定向流水线 CPU

② FPGA 实现：

关于实现的原理，在 Logisim 实现中已经阐述的较为清楚。

BHT 表主体电路，即写控制部分、寄存器组和读控制部分，在 Verilog 实现时，

只用一个 always 循环即可，唯一需要注意的是，每次时钟上沿触发循环时，各表项的 lru 值都需要先加 1（存储 lru 的是计数器不是寄存器），该循环参照寄存器堆 RF 的 FPGA 实现很容易写出来，这里就不给出相关代码了。

对于 lru 算法的实现，图 3.24 所示的硬件电路对应的 Verilog 代码如下：

```
assign lru01 = (lrus[0] < lrus[1]) ? lrus[1] : lrus[0];
assign lru23 = (lrus[2] < lrus[3]) ? lrus[3] : lrus[2];
assign lru45 = (lrus[4] < lrus[5]) ? lrus[5] : lrus[4];
assign lru67 = (lrus[6] < lrus[7]) ? lrus[7] : lrus[6];
assign lru0123 = (lru01 < lru23) ? lru23 : lru01;
assign lru4567 = (lru45 < lru67) ? lru67 : lru45;

assign InsertNum[2] = (lru0123 < lru4567);
assign InsertNum[1] = InsertNum[2] ? (lru45 < lru67) : (lru01 < lru23);
assign InsertNumTemp1 = (lru45 < lru67) ? (lrus[6] < lrus[7]) : (lrus[4] < lrus[5]);
assign InsertNumTemp2 = (lru01 < lru23) ? (lrus[2] < lrus[3]) : (lrus[0] < lrus[1]);
assign InsertNum[0] = InsertNum[2] ? InsertNumTemp1 : InsertNumTemp2;
```

在 Branch 的 FPGA 实现中，BHT 各表项写使能信号和写数据的逻辑和实现都比较简单，不过多讲述，但仍以双预测位状态机的实现为重点。用 Verilog 实现双预测位状态机时，采用了与 Logisim 中相同的原理，图 3.25 对应的 Verilog 代码如下：

```
assign HitWrongPredict = (PredictIn == 2'h0) ? 2'h0 : 2'hz,
      HitWrongPredict = (PredictIn == 2'h1) ? 2'h0 : 2'hz,
      HitWrongPredict = (PredictIn == 2'h2) ? 2'h1 : 2'hz,
      HitWrongPredict = (PredictIn == 2'h3) ? 2'h2 : 2'hz;
assign HitYesPredict = (PredictIn == 2'h0) ? 2'h1 : 2'hz,
      HitYesPredict = (PredictIn == 2'h1) ? 2'h2 : 2'hz,
      HitYesPredict = (PredictIn == 2'h2) ? 2'h3 : 2'hz,
      HitYesPredict = (PredictIn == 2'h3) ? 2'h3 : 2'hz;
assign ToBHT[5:4] = (Hit == 1'h0) ? 2'h2 : 2'hz,
      ToBHT[5:4] = (Hit == 1'h1) ?
                    (PredictYes ? HitYesPredict : HitWrongPredict) :
                    2'hz;
```

ToBHT 是由 Branch 模块发出的 BHT 的控制信号和数据信号总线（为了方便 Logisim 实现时布局的美观），其中第 5 和 4 位表示的是要写入双预测位的值，即双预测位状态机的次态。

4 实验过程与调试

4.1 测试用例

无论是 Logisim 平台还是 FPGA 平台，使用的测试用例都是基于老师提供的 Benchmark 测试程序稍作修改得到的，共有 3 个版本：第一个版本，就是老师提供的 Benchmark 测试程序，没做任何修改；第二个版本，是支持单周期 CPU 中断的 Benchmark 测试程序，主要的修改是在原 Benchmark 种添加了一些初始化代码（初始化栈、开中断等）以及中断处理程序；第三个版本，是支持重定向流水线 CPU 中断的 Benchmark 测试程序，主要的修改是在第二个版本基础上添加了一些空操作。

4.1.1 测试用例 1 – 标准 Benchmark

Benchmark 测试程序的第一个版本，由老师直接提供，包含几个部分，分别测试 CPU 是否可以：正确运行 j、jal、jr 等跳转指令，通过移位测试即正确运行移位指令、加法指令以及条件跳转指令，通过走马灯测试即正确运行逻辑运算和基本运算指令，通过排序测试即正确运行访存指令等。

若运行无误，数码显像管上应依次显示（十六进制）：0、1、3、6、A、...24, 80000000、20000000、08000000、...0, 4、10、40、100、...40000000, 0、80000000, f0000000、ff000000、...fffffff、fffffff1、fffffff11、...11111111, 21111111.....66666666, ffffffff, 0、4、8、c、10、14、18、1c、20、24、28、2c、30、34、38。且程序运行完，内存中应为 e 到-1 (0xffffffff) 的降序排列。

对于单周期 CPU，标准 Benchmark 的运行周期为 1546(包括 syscall 停机指令)；对于重定向流水线 CPU，标准 Benchmark 的运行周期为 $2298 = 1546 + 4$ (启动阶段填充流水线) + 314(分支误取数)*2(误取深度) + 120(Load-Use 数)。

4.1.2 测试用例 2 – 单周期中断 Benchmark

Benchmark 测试程序的第二个版本，在标准 Benchmark 基础上添加了中断处理程序以及初始化代码。本设计中 CPU 共支持 4 级中断，0 号中断处理程序会打印 F 的走马灯，1 号中断处理程序会打印 1 的走马灯，2 号、3 号中断处理程序功能同理。单周期中断 Benchmark 的前 5 条指令为 5 跳转指令，第 1 条指令跳转到主程序的初始化部分，剩余 4 条分别跳转到 0-3 号中断的处理程序。初始化阶段的代码包括：

为\$sp 赋初值 1023 (Logisim 中 ROM 的最高地址) , 然后将 3 到 0 号 (注意顺序) 中断处理程序的入口地址 4~1 依次压入栈中, 最后开中断并跳转到主程序执行。

如若没有触发中断, 则数码显像管的显示与标准 Benchmark 完全相同, 但周期数为 1562; 如若触发中断, CPU 会先转向对应的中断处理程序, 此时显像管会显示对应的走马灯, 然后继续执行 Benchmark。中断是有优先级的, 4 个中断的优先级顺序为: $3 > 2 > 1 > 0$, 支持中断嵌套, 即高优先级中断可打断低优先级中断, 但低优先级中断在 CPU 响应高优先级中断时只能被挂起。

4.1.3 测试用例 3 – 流水线中断 Benchmark

Benchmark 测试程序的第三个版本, 只是在单中期中断 Benchmark 基础上添加了一些空操作, 避免 mfc0、mtc0 两条指令与其他指令发生数据相关冲突 (为了简便设计, 未添加对应的重定向机制), 其余完全相同。

未触发中断时, 流水线中断 Benchmark, 单周期 CPU 的运行周期为 1564, 重定向流水线 CPU 的运行周期为 $2320 = 1564 + 4(\text{启动阶段填充流水线}) + 316(\text{分支误取数}) * 2(\text{误取深度}) + 120(\text{Load-Use 数})$ 。

数码显像管的显示与单周期中断 Benchmark 完全相同。

4.2 功能测试

4.2.1 单周期 CPU

① Logisim 测试:

单周期 CPU 的 Logisim 版本上学期已经实现, 故只给出最后的结果, 如图 4.1。

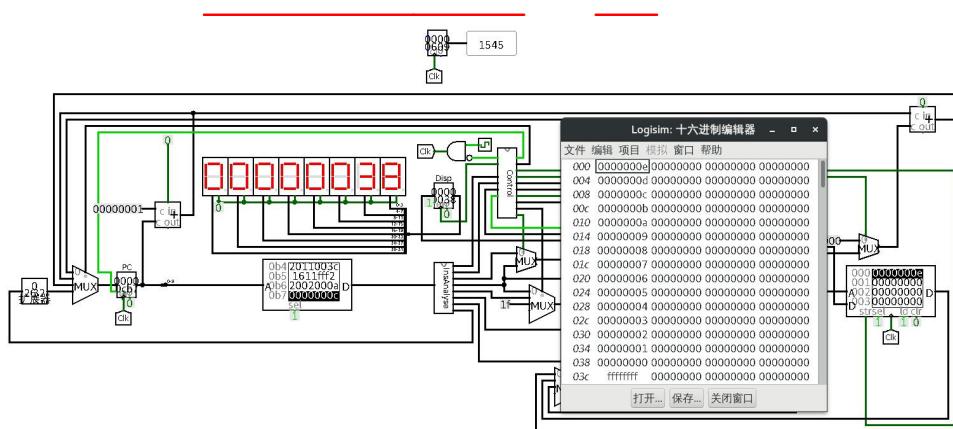


图 4.1 单周期 CPU 的 Logisim 测试

② FPGA 测试：

对单周期 CPU (ROM 中的测试程序为标准 Benchmark) 的 Verilog 工程综合、实现、生成 bit 流，最后将其烧入 FPGA 开发板中，对照显像管的输出与标准 Benchmark 的输出，经测试和比较，单周期 CPU 的 FPGA 实现成功。标准 Benchmark 运行截图如图 4.2、图 4.3 和图 4.4 所示。



图 4.2 FPGA-单周期 CPU 走马灯运行截图



图 4.3 FPGA-单周期 CPU 最终显示



图 4.4 FPGA-单周期 CPU 运行周期数

从图 4.4 可知，单周期 CPU 运行标准 Benchmark 的运行周期数为 $0x609 = 1545$ (没有加最后一条 syscall 停机指令)，周期数也正确。通过拨码开关，选择查看内存分布，此时内存 0-15 号依次为 e、d、c、...0、-1(0xffffffff)。

4.2.2 单周期 CPU 中断

对于支持中断的单周期 CPU，只要求 Logisim 实现。将测试程序——单周期中断 Benchmark 载入程序存储器 IM 中，并启动时钟模拟。运行一定指令后，先点 2 号按钮触发 2 号中断，当 2 号中断处理程序正在执行时，依次点击 3 号按钮和 1 号按钮触发 3 号、1 号中断，CPU 的运行情况（显像管）如图 4.5 所示。

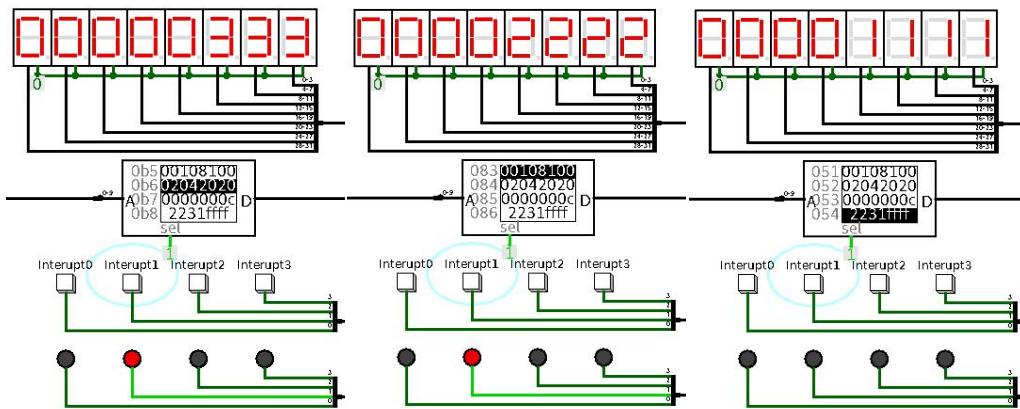


图 4.5 多级嵌套中断单周期 CPU 运行截图

从运行情况来看，本设计成功实现了支持多级嵌套中断的单周期 CPU。所有中断处理程序完执行完成后，CPU 返回原 Benchmark 继续执行，最后如图 4.6。

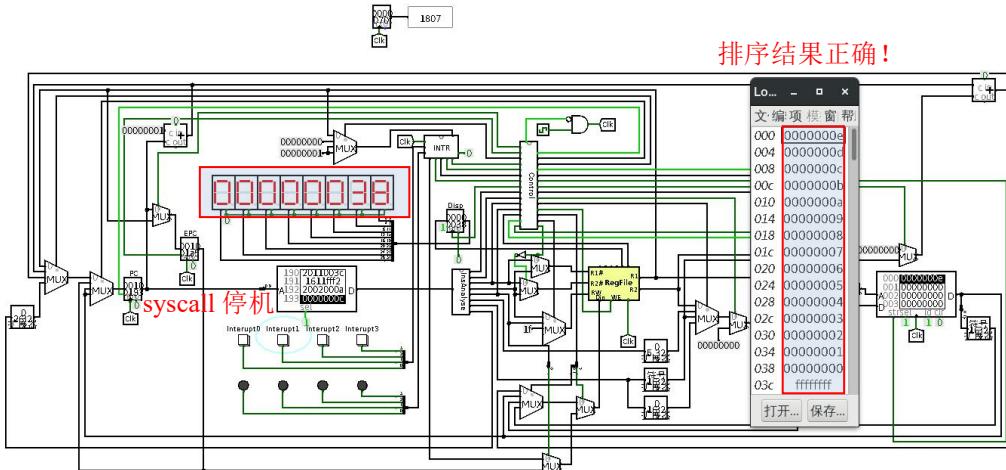


图 4.6 支持中断的 CPU 最终运行结果

4.2.3 重定向流水线 CPU

① Logisim 测试：

将测试程序——标准 Benchmark 载入程序存储器 IM 中，并启动时钟模拟。在运行过程中，对照显像管的输出与标准 Benchmark 的输出，完全一致。待 Benchmark 运行结束后，运行结果如。周期数 2298、分支指令误取数 314 和 Load-Use 数 120，以及内存的排序结果，均正确。

② FPGA 测试：

支持中断的重定向流水线 CPU 完全具有重定向流水线 CPU 的功能和电路，因此这一部分测试可以略。

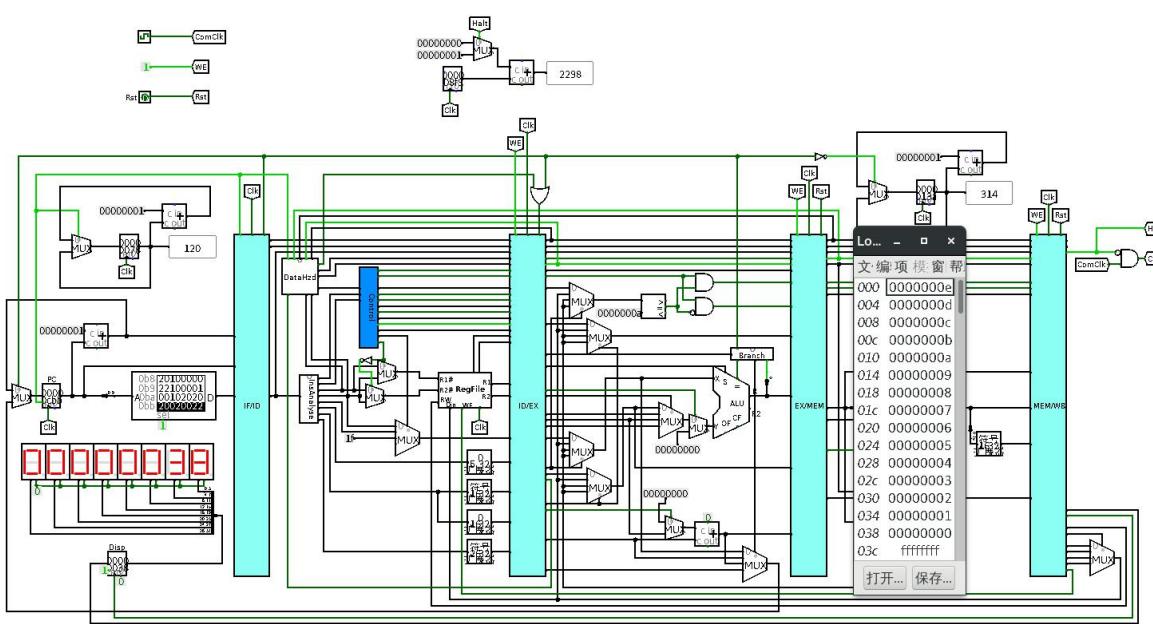


图 4.7 重定向流水线 CPU (Logisim) 最终运行结果

4.2.4 支持中断的重定向流水线 CPU

① Logisim 测试：

将流水线中断 Benchmark 载入程序存储器 IM 中，并启动时钟模拟。运行一定指令后，先点 2 号按钮触发 2 号中断，当 2 号中断处理程序正在执行时，依次点击 3 号按钮和 0 号按钮触发 3 号、0 号中断，CPU 的运行情况（显像管）如图 4.8。

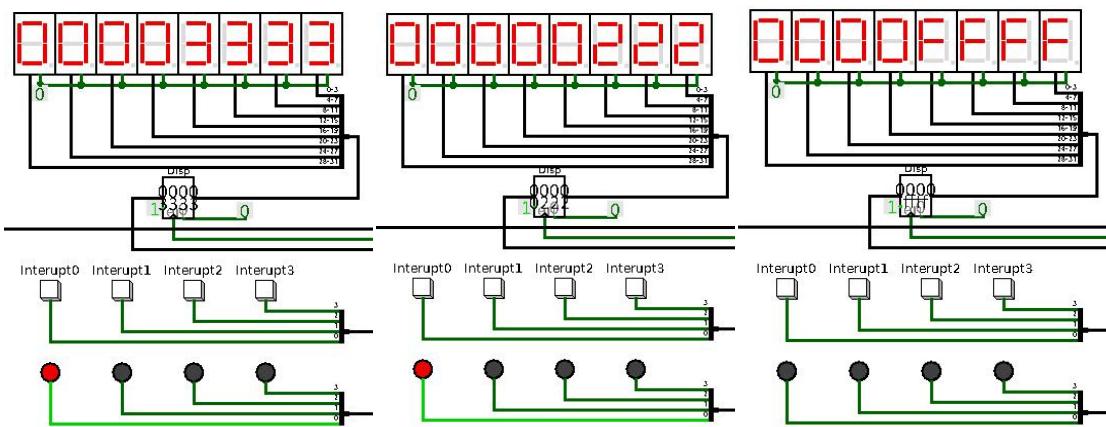


图 4.8 多级嵌套中断运行截图

运行完成后，内存的排序结果也是正确的。

② FPGA 测试：

对支持中断的重定向流水线 CPU（ROM 中的测试程序为流水线中断 Benchmark）的 Verilog 工程综合、实现、生成 bit 流，最后将其烧入 FPGA 开发板中，如 Logisim 测试一样依次触发 2、3、1 号中断，CPU 运行截图如图 4.9。对比图 4.9 和图 4.5，可以验证 CPU 的显像管显示结果与中断等候指示灯均输出正确，流水线中断上板成功。



图 4.9 FPGA 多级嵌套中断运行截图

4.2.5 支持动态分支预测和中断的重定向流水线 CPU

① Logisim 测试：

将流水线中断 Benchmark 载入程序存储器 IM 中，并启动时钟模拟，在 CPU 运行过程中不触发任何中断，最终的运行结果如图 4.10。最后的时钟周期数为 $1786 = 1564$ (流水线中断 Benchmark 单周期 CPU 运行周期数) + 4(填充流水线) + 49(误取分指数)*2(误取深度) + 120(Load-Use 数)，实际结果和计算结果相同，而且内存排序结果经检验也正确，可验证动态分支预测 Logisim 实现成功。

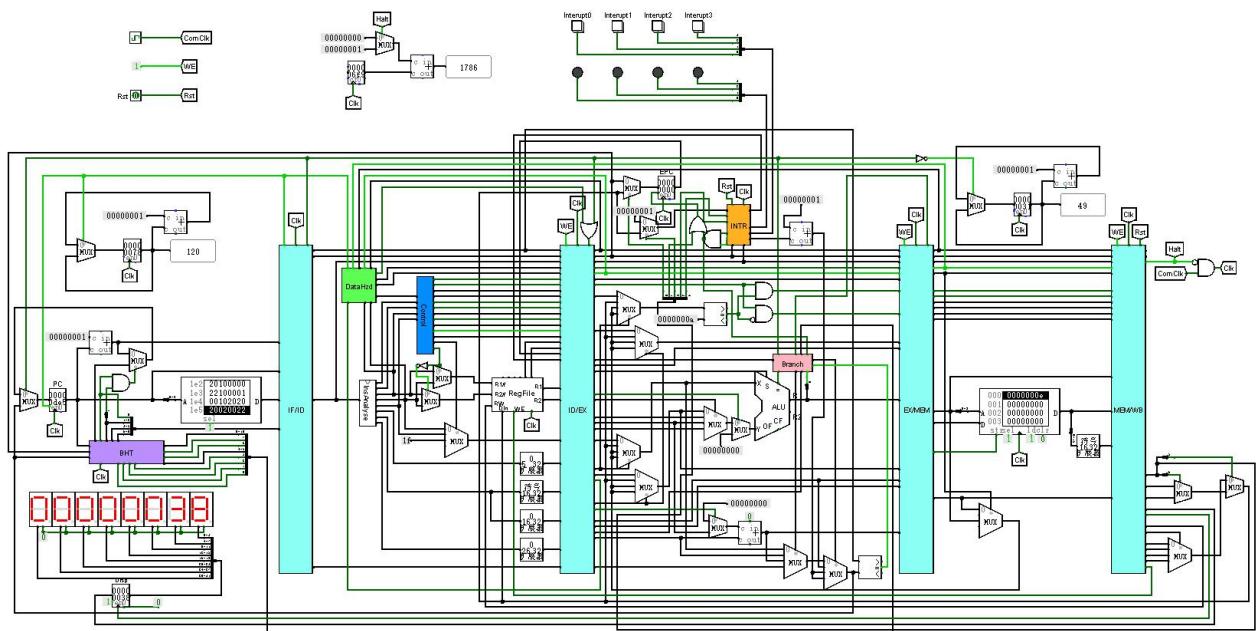


图 4.10 动态分支预测运行结果

② FPGA 测试：

对支持动态分支预测和中断的重定向流水线 CPU (ROM 中的测试程序为流水线中断 Benchmark) 的 Verilog 工程综合、实现、生成 bit 流，最后将其烧入 FPGA 开发板中，在 CPU 运行过程中不触发任何中断，最终的运行结果如图 4.11、图 4.12、图 4.13、图 4.14 所示。



图 4.11 FPGA 动态分支预测运行结果



图 4.12 FPGA 动态分支预测运行周期数



图 4.13 FPGA 动态分支预测运行的 Load-Use 数



图 4.14 FPGA 动态分支预测运行的误取分支数

最后的时钟周期数为 $0x6F8 = 1784 = 1564$ (流水线中断 Benchmark 单周期 CPU 运行周期数) + 4(填充流水线) + (0x30 = 48)(误取分指数)*2(误取深度) + (0x78 = 120)(Load-Use 数)，实际结果和计算结果相同，而且内存排序结果经检验也正确，可验证动态分支预测上板成功。

4.3 性能分析

关于此次课程设计中所设计的所有 CPU 的时钟周期数（包含最后一条 syscall 停机指令）共有几个版本：

一是单周期 CPU 的运行周期数。单周期 CPU 不支持中断，只能运行标准

Benchmark 测试程序，且运行周期数为 1546。

二是支持中断的单周期 CPU 的运行周期数。该 CPU 支持多级嵌套中断，可运行标准 Benchmark 和单周期中断 Benchmark 测试程序，其中，标准 Benchmark 的运行周期数仍为 1546，但单周期中断 Benchmark 的运行周期数为 1562。

三是重定向流水线 CPU 的周期数。重定向流水线 CPU 不支持中断，只能运行标准 Benchmark 测试程序，且运行周期数为 $2298 = 1546 + 4 + 314*2 + 120$ 。注意到，在本设计中误取深度为 2，即意味着跳转分支指令的执行是在 EX 阶段。如果跳转分支指令在 IF 阶段执行，则会使得 IF 阶段的延时过高，如此一来虽然最后的运行周期数减少，但 CPU 可支持的最大频率下降了，即 CPU 的运行速度会减慢；如果跳转分支指令在 MEM 阶段执行，则误取深度将为 3，最后的运行周期数将会变多。综合考虑，将跳转分支指令的执行放在 EX 阶段是较为合理的选择。流水线 CPU 的时钟周期数虽然相比单周期 CPU 而言大大提高，但每个阶段的延时大大降低，CPU 支持的最大频率也会大幅度提升，因此总体来说 CPU 执行速度是变快的。

四是支持中断的重定向流水线 CPU 的周期数。该 CPU 支持多级嵌套中断，可运行标准 Benchmark 和流水线中断 Benchmark 测试程序，其中，流水线中断 Benchmark 的运行周期数为 $2320 = 1564 + 4 + 316*2 + 120$ ，标准 Benchmark 的运行周期数与三相同。

五是支持动态分支预测和中断的重定向流水线 CPU 的周期数。该 CPU 不仅支持多级嵌套中断，还具有动态分支预测的机制，可运行标准 Benchmark 和流水线中断 Benchmark 测试程序。通过动态分支预测，可以大幅度地减少分支指令的误取数。以流水线中断 Benchmark 为例，在不添加动态分支预测前，其运行周期数为 2320，其中分支指令误取数为 316；添加动态分支预测后，其运行周期数为 1784(FPGA 版本)，其中分值指令误取数减少至为 48，预测的正确率为 $84.8\% = (316 - 48) / 316$ 。当支持动态分支预测后，流水线 CPU 的运行周期数与单周期 CPU 相当，可知流水线 CPU 的执行速度将远远大于单周期 CPU。

综上，对于最终版本，即支持动态分支预测和中断的重定向流水线 CPU，在 FPGA 开发板上运行流水线中断 Benchmark，运行周期数为 1784，仅比单周期 CPU 的运行周期数多了 $220 = 4 + 48*2 + 120$ 个周期数，表明本设计实现的 CPU 的执行性能还是比较可观的。

4.4 主要故障与调试

4.4.1 故障 1

Verilog 实现单周期 CPU： 移位指令执行错误。

故障现象： 使用 Vivado 仿真时，CPU 执行到 Benchmark 中测试移位指令的部分时，显像管输出与 Logisim 上单周期 CPU 的输出不同。当显像管输出“00000024”后，应该输出“80000000”，但仿真结果显示此时输出为“0000003e”。并且此后，CPU 陷入了死循环，显像管的输出循环变化（错误），如图 4.15 所示。

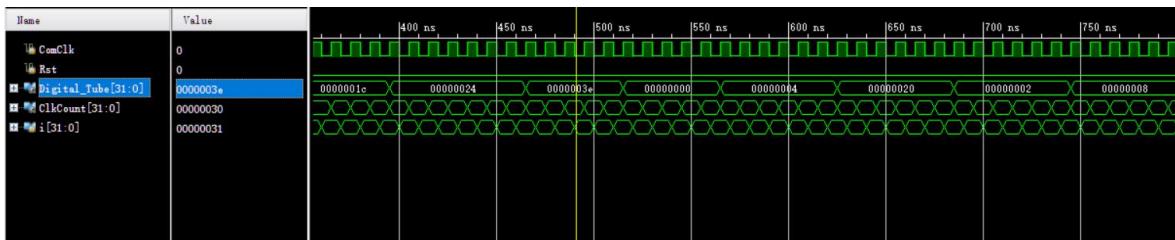


图 4.15 故障 1 示意图

原因分析： 将指令存储器 IM 即 ROM 中的输出、寄存器堆 RF 的 32 个寄存器的输出引入到波形图，对比仿真时指令的指令顺序、指令执行前后寄存器堆 RF 中对应寄存器的变化，发现 CPU 在执行移位指令“sll \$s1, \$s1, 31”，即对应机器码为“0018fc0”的指令时，寄存器\$s1 中的值并没有发生变化。为了找到对应的原因，再把 ALU 的操作数和运算结果引入到波形图中，发现时 ALU 的运算结果本身就不对。于是，返回 ALU 的 Verilog 代码，重点研究移位操作（操作符为 0~2）对应的代码，终于发现对于移位操作，Logisim 实现的 ALU 是操作数 X 左移或右移 Y 位，但 Verilog 实现的 ALU 确是操作数 Y 左移或右移 X 位。



图 4.16 故障 1 故障原因分析波形图

解决方案： 将 ALU 的 Verilog 代码中移位操作中的操作数 x、y 的顺序互换，从而使其与 Logisim 的 ALU 逻辑功能保持一致即可。

4.4.2 故障 2

FPGA 上运行单周期 CPU：单周期 CPU 上板运行后，无法查看内存地址大于 4 的内存单元的内容，只能查看 0~3 号内存单元的内容。

故障现象：将单周期 CPU 烧入 FPGA 时，由于实验要求里提到要可以展示内存 0~15 号单元中排序的结果，所以使用 4 个拨码开关来控制显示 16 个内存单元。但是，实际运行时只能显示 0、1、2、3 号内存单元的内容，就好像只有两个拨码开关（低地址）在起作用。当需要显示 4 号、8 号、12 号内存单元时，显示的均为 0 号单元的值“e”，如图 4.17 故障 2 示意图。注意到 $4=100B$, $8=1000B$, $12=1100B$ ，意味着内存地址的高两位地址始终为 0。



图 4.17 故障 2 示意图

原因分析：起初，也是从故障现象最容易想到的原因是，控制内存高两位地址的两个拨码开关硬件上是坏的，于是想到将高两位和低两位地址绑定的拨码开关对调，发现仍是控制高两位地址的拨码开关不起作用，故排除了开关坏掉的可能；找了好长时间的原因后，发现综合阶段 Vivado 给出了警告，如图 4.18。其中就有跟 RAM 相关的警告，根据 Viva 都给出的警告的提示，才找到了问题所在。访问 4 号内存单元的代码原本为类 “assign mem4 = mem[4'h10];”，注意到这个常数 “4'h10”， $0x10=16$ ，而 4 位可表示的最大数位 15。所以，出现了高位被舍弃的情况。

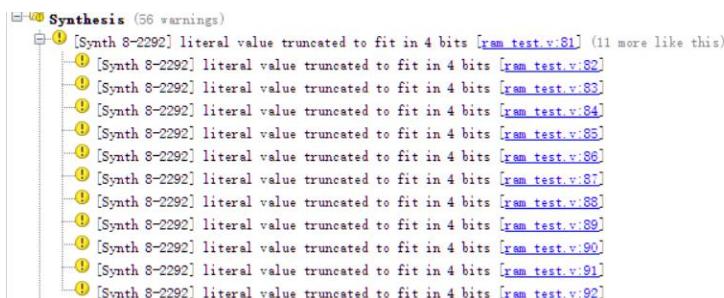


图 4.18 故障 2 原因分析

解决方案：修改动态分支预测器的敏感变量表，将此模块修改为时序控制，使用时钟的下降沿进行控制，在上一个时钟上升沿，分支指令进入 EX 段，并且判断出分支预测结果的正误，随后的时钟下降沿，使用判断结果作为依据进行状态机的状态转移，便可实现状态机状态的正确转移。

4.4.3 故障 3

Verilog 实现流水线中断：中断等候指示灯信号不能被清零。

故障现象：如图 4.19 所示，当有中断来临（IRSrc 由 0 变成 1，表明 0 号中断被触发）时，0 号中断等候指示灯变为 1，故 IRWT 由 0 变成 1。但在下一个周期，在 Logisim 实现的 INTR 模块中，0 号中断等候指示灯信号会被置零；但是仿真结果显示，0 号中断等候指示灯信号不仅没有被置零，反而其他三个中断指示灯信号均变成了 1，导致 IRWT 为 0xf。

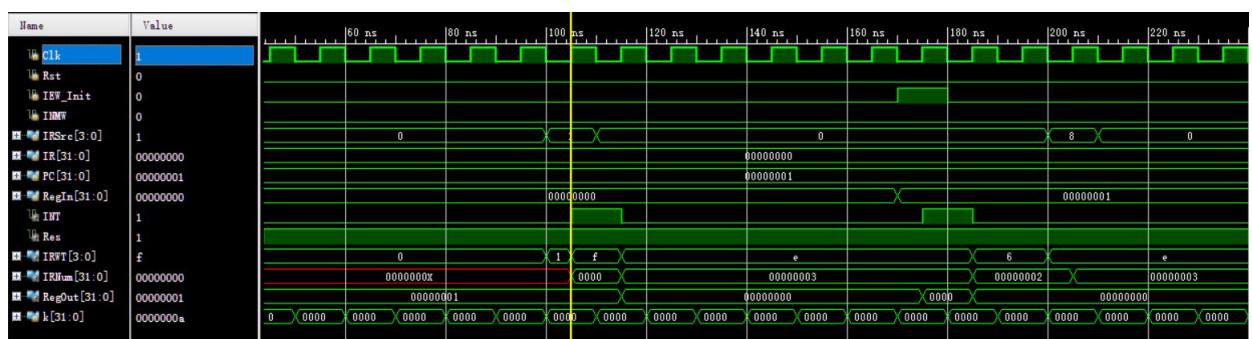


图 4.19 故障 3 示意图

原因分析：经过查看 INTR 模块各寄存器的输出发现，在图 4.20 中黄线所指周期内，其他三个中断指示信号灯之所以变成 1，是因为对应的中断第一级寄存器都被置为了 1。至于为什么这 3 个寄存器会变成 1，找了很久很久的问题，想了很久很久的原因，尝试很多很多可能，最终确定了问题的产生原因——异步清零。首先来看一下，异步清零的 Verilog 代码是如何写的：

```
always @ (posedge clk or posedge rst)
begin
    if (rst)
        reset_code;
    else if (we)
        write_code;
end
```

程序员并不清楚 Verilog 是如何将上面的代码转化为实际的电路的，但就上面代码而言，存在这么一种情况：rst 信号确实从 0 变成了 1，但只是一个很小的间隙

后紧接着变为 0，即 rst 信号是一个尖脉冲。这种情况下，很可能 rst 会触发 always 循环，但在进行 “if (rst)” 的判断时，rst 已经从 1 变成了 0，那么将会执行的不是 reset 操作而是 write 操作。所以，对于异步清零，如果清零信号 rst 持续时间不够长，就很可能造成上述问题。再来看中断一级寄存器的 Verilog 代码为：

```
pipeline_register #(1) IR0Cache1 (.din(1'h1), .we(1'h1), .rst(IR0Clr),
                           .asyn_rst(Rst | IR0Clr), .clk(IR0), .dout(Cache1IR0));
```

异步清零信号为 Rst | IR0Clr，而 IR0Clr 很有可能是一个尖脉冲，而写使能端又恒为 1，所以很可能出现异步清零但缺写入了数据的情形。

解决方案：为了避免异步清零带来的写如操作，且能够保证异步清零的功能正确性，所以不修改异步清零信号，而是修改写使能信号，将其修改为 IR0Clr。

4.5 实验进度

表 4.1 课程设计进度表

时间	进度
第一天	完成各个基本模块、控制器及 CPU 的 Verilog 代码
第二天	成功在 FPGA 上实现和运行单周期的 MIPS CPU，可测试 benchmark；在 Logisim 和 FPGA 上完成了扩展指令部分，并编写了扩展指令测试程序；修改了支持多中断的 Logisim 下 CPU；
第三天	logisim 实现了理想流水线，并解决了分支相关（插入气泡）
第四天	logisim 实现了气泡流水线，实现了重定向流水线（周期数位 2298）
第五天	完成 Verilog 的流水线代码，正在 Debug...
第六天	在 FPGA 的重定向流水线基础上实现了多级嵌套中断
第七天	完成动态分支分支预测的 Logisim 版本，完成动态分支预测的相关 FPGA 代码的书写，Debug 中...
第八天	通关！5 段流水+多级嵌套中断+动态分支预测上板成功！跑带中断处理程序的 Benchmark 周期数为 1784(比标准 benchmark 多 18 条指令，其中多两条跳转指令)，原始分支误取数为 316，加分支预测后误取数为 48，预测正确率为 84.8%
第九天	写报告，帮助组内和班里其他同学解决问题
第十天	写报告，帮助组内和班里其他同学解决问题

5 设计总结与心得

5.1 课设总结

本次课程设计主要完成了如下几点工作：

- 1) 完成了组原实验 CPU 的指令扩展方案的设计，思考并设计了单周期 MIPS-CPU 的多级嵌套中断机制，完成了 5 段理想流水线 MIPS-CPU 的设计方案，学习并设计了 5 段气泡式流水线 MIPS-CPU，完成了 5 段重定向流水线 MIPS-CPU 的方案设计，思考并设计了重定向流水线 MIPS-CPU 的多级嵌套中断机制，完成了重定向流水线 MIPS-CPU 动态分支预测机制方案的设计。
- 2) 在 Logisim 和 FPGA 两个平台实现了扩展指令后的单周期 MIPS-CPU，在 Logisim 平台上实现了支持多级嵌套中断的单周期 MIPS-CPU，在 Logisim 平台上实现了 5 段理想流水线 MIPS-CPU，在 Logisim 平台上实现了 5 段气泡式流水线 MIPS-CPU，在 Logisim 和 FPGA 两个平台实现了重定向流水线 MIPS-CPU，在 Logisim 和 FPGA 两个平台实现了支持多级嵌套中断的重定向流水线 MIPS-CPU，在 Logisim 和 FPGA 两个平台实现了支持动态分支预测和多级嵌套中断的重定向流水线 MIPS-CPU。
- 3) 本次课程设计的亮点在于，完成了任务路线中的所有节点（包括 3 个扩展），而且是在 Logisim 和 FPGA 双平台实现了流水线中断和动态分支预测，尤其是动态分支预测的 Logisim 版本（感觉做的人不多）。

5.2 课设心得

坦白的说，做完上学期的组原课程实验，其实一直有一种意犹未尽的感觉，所以对本学期要开始的组原课程设计很是期待，尽管知道又要写 Verilog 了。

本人是在第二周的星期三“打通了关”，完成了所有能完成的任务，还包括动态分支预测的 Logisim 版本。但是，考虑到课程设计正式开始前前提前 3 天就已经着手做扩展指令和单周期 CPU 的多级嵌套中断了，所以为了通关前前后后大约花了 12 天左右的时间，充满挑战和成就感的 12 天。

扩展指令之部分也许是因为我抽到的 4 条指令都相对简单，所以花了不到半天的时间就完成了；第一个挑战来自单周期 CPU 的多级嵌套中断机制的实现，由于当时做的时候老师还没开放相关资料，所以完全是凭借上学期讲中断的 PPT 独立思考出来的实现方法，也和队友（分组情况早就知道了）进行了一些探讨，印象最深刻就是中断隐指令这部分内容，当时本人经过思考后发现在 1 个周期内完成设计起来最简单，后来中断 19 问中正好有这个问题；然后是单周期 CPU 的上板，组内共享的基本部件由本人编写，而有了 Logisim 的参照，Verilog 写起来除了麻烦点没什么难点，而数电课设提升了本人使用 Vivado 波形图调试仿真错误的能力，所以经过 1 天的奋斗，成为班里第一个单周期 CPU 上板成功的人和小组；接下来是理想流水线，理解了理想流水线的原理，只用 Logisim 实现难度不大；气泡流水线的关键在于如何处理数据相关和分支相关，其中麻烦的是数据相关，因为要考虑不同指令操作数不同的特点；重定向流水线比本人想象的要复杂，一开始一直想不明白为什么要从 MEM 和 WB 阶段重定向，烧了一下午的脑并询问了老师一些问题，才算搞懂重定向的原理；接着是重定向流水 CPU 的上板，从早晨到凌晨 1 点半，花了整整一天的时间；流水线中断是所有任务里花的时间最长的一个，因为找 4.4.3 故障 3 就找了很长时间，结果还是因为老师一直强调的异步清零自己没听到心里去；最后一个动态分支预测，Logisim 实现起来真是复杂，连了 1 天才连完，但有了 Logisim 的铺垫，动态分支预测的 Verilog 实现就只花了 1 晚上就搞定了。

接下来，想谈一下自己在 Verilog 方面的心得。一是，Verilog 这种硬件描述语言和 C、C++、Java 等软件开发语言还是有着本质区别的，在使用 Verilog 进行程序开发时，必须时刻牢记其与硬件是直接相关联的。也就是说，利用 Verilog 开发的程序最终是要转变成具体的电路的。因此，也许从软件开发的角度来看，写出的程序并没有什么问题，但在使用 Vivado 进行 synthesis 和 implementation 时便会报错。如果不是语法错误和逻辑错误，便很有可能是因为程序并不能在既有的硬件上实现。举个例子，比如在两个“always”循环里对同一个寄存器变量进行赋值，这从软件开发的角度来看并没什么问题，符合语法和逻辑，但要从硬件的角度考虑，这样的设计是不妥甚至是错误的，因为如果用硬件实现会造成脉冲混乱。二是，像 Verilog 这种 HDL 即硬件描述语言，封装了底层实现的过程和方法，实际增加了调试的难度。也就是说，我们完全不清楚 Vivado 是如何把用 Verilog 语言开发的程序转化为具体的实现电路的，这样的话在程序遇到错误或者说是硬件表现不是预期

时，会很难发现错误的原因所在。纵使 Vivado 提供了仿真，但毕竟仿真不能真实模拟硬件环境，我相信很多同学跟我一样在开发过程中遇到过仿真完全正确但烧入板子后就不正确的体验（甚至 Vivado 自身也是由 Bug 的）。一旦这种情况发生，是很难或者说是很花时间去分析和寻找错误所在的。

然后，是对组原课程设计这门课的一些想法和建议。坦白的说，这种分组机制对我的帮助很有限，只有在一开始做单周期中断的时候，两个队友和我的进度差不多，我们对中断机制的理解和自己的设计思路进行了深入的讨论，让我感到收获颇丰。但在课程设计正式开始后，基本任务路线的每个阶段，我都是班里第一个完成的，所以大部分时间我总是在独自思考和设计。但这并不代表我不支持分组机制，相反，我认为这样的机制确实是有助于那些本来对实验内容并不理解或者理解并不到位的同学，小组讨论可以启发他们的思路，帮助他们更快地上手和开始设计。更为关键的是，组内的高手可以向他们传授一些设计、写代码、调试的经验和技巧，提高他们的效率。例如，在这次课程设计期间，我发现很多同学并没有完全掌握 Vivado 仿真调试的技巧，甚至不会设置指定的仿真时间，而这些技巧可以大幅度的提高调试效率。不客气的说，组原课程设计期间，我教会了不少同学“如何正确使用 Vivado 的仿真测试及波形图”，应该会为他们减少一些调试时间。

另外，建议老师可以继续延长任务路线，增加一些扩展任务，例如：可以增加在 RAM 和 CPU 之间加入 cache 的任务，在架构上更接近与真实 CPU；可以增加实现 MMU 部件的任务，以使课设 CPU 运行操作系统成为可能；可以增加实现 CPU 一些外设的任务，只有数码显像管确实有些单调。这些扩展任务可能大部分同学都不会完成，但有少数同学还是渴望难度更大的挑战的，渴望能做出一些更炫酷、不一样的东西来的。还有就是，可以将 ACM 和卓越班甚至其他普通班的工作日志合在一起，这样应该更能激励一些同学（本人就时常去 ACM 班的工作日志区看看，有竞争会提高一定的效率）。

最后，衷心感谢组成原理课程设计教师组的各位老师！无论是课设的任务书，还是课设的报告模板（超好用的模板，其他课设也都用的这个），都能感受到老师的用心！可以说，组成原理的实验和课程设计，是目前接触到的最正规、最接近国际水平的实验和课设！

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.

• 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：