华中科技大学

2025

硬件综合训练　　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：	CS 本硕博 2301
学　　号：	U202315763
姓　　名：	王家乐
电　　话：	15391560195
邮　　件：	chia.le@foxmail.com

目 录

1 课程设计概述

1.1 课设目的

计算机组成原理是计算机专业的核心基础课。该课程力图以“培养学生现代计算机系统设计能力”为目标，贯彻“强调软/硬件关联与协同、以 CPU 设计为核心/层次化系统设计的组织思路，有效地增强对学生的计算机系统设计与实现能力的培养”。课程设计是完成该课程并进行了多个单元实验后，综合利用所学的理论知识，并结合在单元实验中所积累的计算机部件设计和调试方法，设计出一台具有一定规模的指令系统的简单计算机系统。所设计的系统能在 LOGISIM 仿真平台和 FPGA 实验平台上正确运行，通过检查程序结果的正确性来判断所设计计算机系统正确性。

课程设计属于设计型实验，不仅锻炼学生简单计算机系统的设计能力，而且通过进行中央处理器底层电路的实现、故障分析与定位、系统调试等环节的综合锻炼，进一步提高学生分析和解决问题的能力。

1.2 设计任务

本课程设计的总体目标是利用 FPGA 以及相关外围器件，设计五段流水 CPU，要求所设计的流水 CPU 系统能支持自动和单步运行方式，能正确地执行存放在主存中的程序的功能，对主要的数据流和控制流通过 LED、数码管等适时的进行显示，方便监控和调试。尽可能利用 EDA 软件或仿真软件对模型机系统中各部件进行仿真分析和功能验证。在学有余力的前提下，可进一步扩展相关功能。

1.3 设计要求

（1） 根据课程设计指导书的要求，制定出设计方案；

（2） 分析指令系统格式，指令系统功能。

（3） 根据指令系统构建基本功能部件，主要数据通路。

（4） 根据功能部件及数据通路连接，分析所需要的控制信号以及这些控制信号的有效形式；

（5） 设计出实现指令功能的硬布线控制器；

（6） 调试、数据分析、验收检查；

（7） 课程设计报告和总结。

1.4 技术指标

（8） 支持表 1.1 前 27 条基本 32 位 MIPS 指令；

（9） 支持教师指定的 4 条扩展指令；

（10）支持多级嵌套中断，利用中断触发扩展指令集测试程序；

（11）支持 5 段流水机制，可处理数据冒险，结构冒险，分支冒险；

（12）能运行由自己所设计的指令系统构成的一段测试程序，测试程序应能涵盖所有指令，程序执行功能正确。

（13）能运行教师提供的标准测试程序，并自动统计执行周期数

（14）能自动统计各类分支指令数目，如不同种类指令的条数、冒险冲突次数、插入气泡数目、load-use 冲突次数、动态分支预测流水线能自动统计预测成功与失败次数。

表 1.1 指令集

#	指令助记符	简单功能描述	备注
1	ADD	加法	指令格式参考 MIPS32 指令集，最终功能以 MARS 模拟器为准。
2	ADDI	立即数加	
3	ADDIU	无符号立即数加	
4	ADDU	无符号数加	
5	AND	与	
6	ANDI	立即数与	
7	SLL	逻辑左移	
8	SRA	算数右移	
9	SRL	逻辑右移	
10	SUb	减	
11	OR	或	
12	ORI	立即数或	

#	指令助记符	简单功能描述	备注
13	NOR	或非	
14	LW	加载字	
15	SW	存字	
16	BEQ	相等跳转	
17	BNE	不相等跳转	
18	SLT	小于置数	
19	STI	小于立即数置数	
20	SLTU	小于无符号数置数	
21	J	无条件转移	
22	JAL	转移并链接	
23	JR	转移到指定寄存器	If $v0==10 halt(停机指令)
24	SYSCALL	系统调用	else 数码管显示$a0 值
25	MFC0	访问 CP0	中断相关，可简化，选做
26	MTC0	访问 CP0	中断相关，可简化，选做
27	ERET	中断返回	异常返回，选做
28	SRA	算术右移	C-3
29	SLTIU	无符号立即数比较	C-7
30	LHU	读入一个无符号半字	A-4
31	BLT	小于条件分支	B-1

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采用指令存储器和数据存储器相分离的结构（哈佛结构）。在这里我们采用简单迭代法实现单周期处理器。这种方法相对比较直观简单，先完成支持一条固定 R 型指令（如加法指令）的基本数据通路，测试运行通过后再在这个基础上不断增加新的数据通路，支持新的指令，直至所有指令都能正常运行。要支持一条指令的运行必须有取指令逻辑和执行指令的逻辑，所有设计首先应该完成取指令的数据通路。

需要特别说明的是，在单周期处理器中，一条指令执行过程中数据通路的任何资源都不能被重复使用，因此任何需要被多次使用的资源(如加法器)都需要设置多个，否则就会发生资源冲突，这是设计单周期处理器数据通路时必需考虑的重要环节。总体结构图如图 2.1 所示。

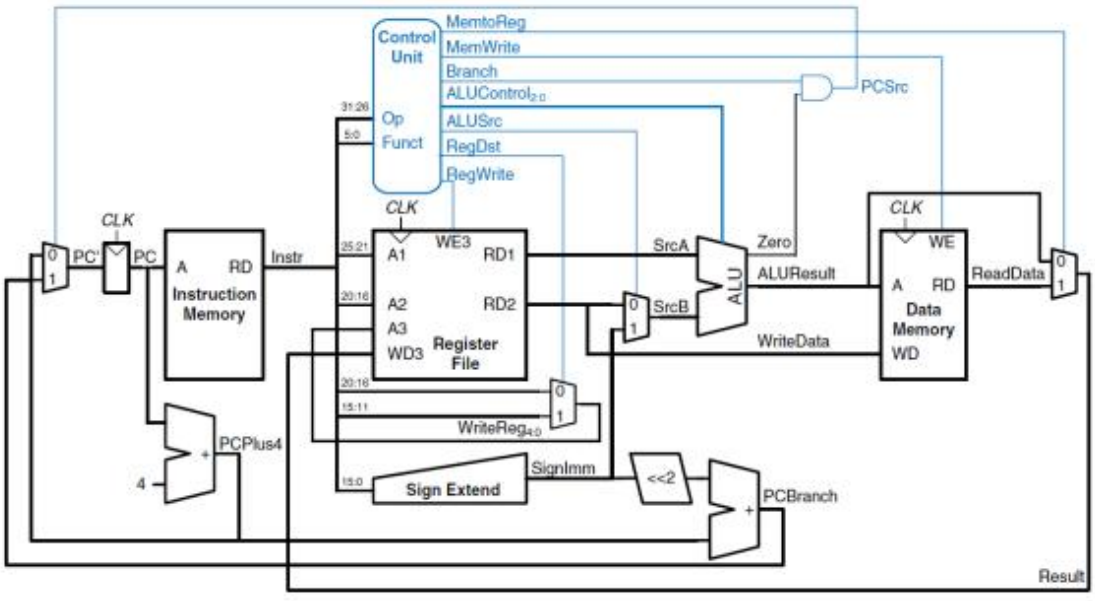


图 2.1 总体结构图

2.1.1 主要功能部件

主要功能部件具体设计思路如下：

1. 程序计数器 PC

取指令通路应能取出程序计数器 PC 锁存地址对应的的机器指令，并能在时钟上跳沿到来时实现 PC 自动加 4，从而进入下一条指令的指令周期。这里程序计数器 PC 可以采用寄存器实现，其输出作为指令存储器的地址输入，指令存储器经过一个存储周期后一次性取出 32 位的机器指令，故其数据宽度应为 32。

2. 指令存储器 IM

为简化电路设计，Logisim 中采用 ROM 实现。在顺序执行方式下，每一个时钟周期内 CPU 取指令后将 PC 寄存器的值加 4，形成下一条指令的地址。这里加“4”是因为 RISC-V32 中指令字长均为 4 字节，每条指令在存储器中占用 4 个字节的存储单元，而 PC 中存放的地址是字节地址。为避免资源冲突，这里加法器应该是独立的器件，和 CPU 中的算术逻辑运算单元分开。

表 2.1 指令存储器引脚与功能描述

引脚	输入/输出	位宽	功能描述
PC	输入	10	程序地址
IR	输出	32	待解析指令

3. 运算器 ALU

运算器接受两个 32 位的操作数输入，根据操作码决定对两个数进行何种运算，输出一个 32 位的结果，并输出一位数表示两个操作数是否相等。除此之外还有有符号溢出、无符号溢出等多种输出，由于在这里不需要使用，所以忽略这些输出。输入输出引脚描述如表 2.2。

表 2.2 算术逻辑运算单元引脚与功能描述

引脚	输入/输出	位宽	功能描述
X	输入	32	操作数 X

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

4. 寄存器堆 RF

寄存器文件 RF 是 RISCV - CPU 中 32 个 32 位寄存器组成的阵列，通常由快速的静态随机读写存储器（SRAM）实现。这种 RAM 具有专门的读端口与写端口， 可以多路并发访问不同的寄存器。RF 一次可以读取 R1 和 R2 两个寄存器；一次可写一个寄存器，用 Din 表示，用时钟控制，一位写使能信号用来标识当前周期是否需要写数据；写入和读出的数据都是 32 位的。输入输出引脚描述如表 2.3。

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
Clk	输入	1	时钟
W#	输入	5	写寄存器号
R1#	输入	5	读寄存器号 1
R2#	输入	5	读寄存器号 2
WE	输入	1	写使能
Din	输入	32	写入数据
R1	输出	32	读出数据 1
R2	输出	32	读出数据 2

2.1.2 数据通路的设计

对于不同的指令有不同的数据通路，具体设计如下：

1. 取指令数据通路

PC 寄存器存放当前取指令字地址，传给指令存储器的地址输入端。在顺序执行方式下，对 PC 进行+4 操作；在跳转方式下，传给 PC 跳转地址，通过多路选择器进行不同方式的选择。

2. R 型指令数据通路

R 型指令字通常包含源寄存器号 rs1 与 rs2，以及目的寄存器号 rd。所做操作通常为 R[rd] = R[rs1] op R[rs2]。那么需要通过寄存器堆读寄存器、运算器运算和写回寄存器堆的过程，数据通路中经过这三个主要部件。

3. I 型指令数据通路

I 型指令进行立即数和寄存器的联合运算，指令字包含 12 位立即数 I[11:0]和源寄存器号 rs1，以及目的寄存器号 rd。操作通常为 R[rd] = R[rs1] op I，特别的，对于 lw 指令操作为 R[rd] = M[R[rs1]+I]。因此，数据通路与 R 型指令类似，只是运算器的两个输入端分别为寄存器数据和扩展立即数，且 lw 指令下的数据通路还包含读数据存储器的通路。

4. S 型指令数据通路

此次设计中，S 型数指令只包含 sw 指令，指令字包含两个寄存器号 rs1 和 rs2，以及立即数 I。操作为 M[R[rs1]+I] = R[rs2]。数据通路中，依次经过取指，取寄存器数据，运算器相加运算和存储数据的过程，无需写回。

5. 分支指令数据通路

对于分支指令，可以分为条件分支指令与无条件分支指令。条件分支指令需要通过 ALU 获取状态信息，即 Beq 需要获取 equal 信号才能跳转。而无条件分支指令无需状态判断，即可实现跳转，如 jal 指令则是直接根据指令字中的立即数扩展来跳转的。相应的，数据通路根据不同指令，仅在 ALU 有所区别。

6. Ecall 指令数据通路

对于 ecall 指令，拥有分支功能。当 a7 寄存器为 10 时，实现停机功能；否则显示 a0 寄存器的数据。在 ecall 操作时，将 rs1 输入端置 a7 寄存器号 17，rs2 输入端置 a0 寄存器号 10，获取对应寄存器数据。再进行对应的分支判断，进行 LED 显示或各部件置无效使能。

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	控制信号	取值	说明
ALU_OP	0~12	运算器操作控制符（4 位）	BEQ	0/1	Beq 指令译码信号
MemToReg	0/1	寄存器写入数据来自数据存储器	BNE	0/1	Bne 指令译码信号
MemWrite	0/1	写数据存储器控制信号	JAL	0/1	Jal 指令译码信号
ALU_Src	0/1	运算器 B 输入端选择控制信号	JALR	0/1	Jalr 指令译码信号
RegWrite	0/1	寄存器写使能	LHU	0/1	Lhu 指令译码信号
Ecall	0/1	ecall 指令译码信号	BLT	0/1	Blt 指令译码信号
S_Type	0/1	S 型指令译码信号			

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。根据控制信号表达式生成表，如图 2.2，得到所有控制信号的逻辑表达式，利用逻辑表达式自动生成硬布线控制器逻辑线路。

#	指令	Funct7 (十进制)	Funct3 (十进制)	OpCode (十六进制)	ALU_OP	MemtoReg	MemWrite	ALU_Src	RegWrite	ecall	S_Type	BEQ	BNE	JAL	JALR	LHU	BLT
1	add	0	0	C	5				1								
2	sub	32	0	C	6				1								
3	and	0	7	C	7				1								
4	or	0	6	C	8				1								
5	slt	0	2	C	11				1								
6	sltu	0	3	C	12				1								
7	addi		0	4	5			1	1								
8	andi		7	4	7			1	1								
9	ori		6	4	8			1	1								
10	xori		4	4	9			1	1								
11	slti		2	4	11			1	1								
12	slli	0	1	4	0			1	1								
13	srli	0	5	4	2			1	1								
14	srai	32	5	4	1			1	1								
15	lw		2	0	5	1		1	1								
16	sw		2	8	5		1	1			1						
17	ecall	0	0	1C						1							
18	beq		0	18	6							1					
19	bne		1	18	6								1				
20	jal			1B					1					1			
21	jalr		0	19	6			1	1						1		
22	CSRRSI		6	1C	8												
23	CSRRCI		7	1C	7												
24	URET	2	0	1C													
25	sra	32	5	C	1				1								
26	sltiu		3	4	12			1	1								
27	lhu		5	0	5	1		1	1							1	
28	blt		4	18	11												1

图 2.2 控制信号表达式生成表

2.2 中断机制设计

2.2.1 总体设计

中断类型大体可以分为内部中断和外部中断，本次实验设计的中断为可屏蔽的外部中断。本次实验中完成了支持单级中断的单周期 CPU 以及支持多级中断的单周期 CPU。 对于所有的外部中断，跳转到中断指令的方法一般为：保存下一条指令地址，保护现场的寄存器，然后跳到中断源执行中断。从中断返回的方法一般为：根据 ecall 指令恢复中断现场的寄存器，指令跳转到存储的下一条指令的地址。所需要的中断的构成模块相同包括：中断控制器、中断使能信号产生模块、中断请求信号产生模块、中断清零信号产生逻辑、中断入口逻辑模块、保护现场模块等。中断的执行基本逻辑如图 2.3 所示。

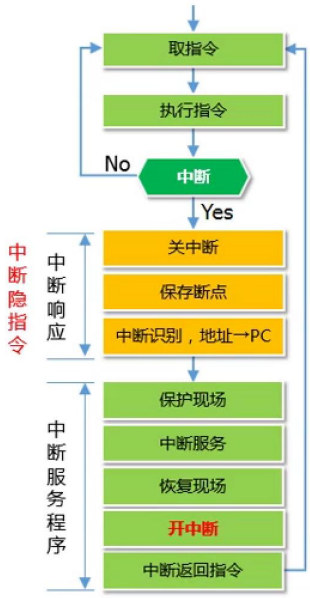


图 2. 3 中断处理机制

2.2.2 硬件设计

本次实验设计实现的中断方法为硬件保存，即对于每一种类、每一等级的中断，都单独使用一套完整的寄存器进行现场的保存。具体而言，一组中断寄存器由 EPC 和 IRID 分别保存中断返回地址和当前执行的中断号。此外还用 IE 寄存器保存当前是否允许执行更高等级的中断的 IE 信号，高电平有效。

对于两种中断，均需要支持一个中断请求队列，即在中断执行过程中接受到的中断在当前中断周期结束后转而去执行队列中的请求。这里只需要对每个中断请求

使用一个寄存器保存是否需要执行该中断的信号，若执行完整则激活对应清除请求的 reset 信号，即可将执行完整的中断从队列中弹出。

对于单级中断，由于在执行中断程序时不允许被中断去执行其他中断程序，因而实现较为简单，仅需使用一组中断寄存器保存当前中断的信息即可。利用中断使能寄存器存储使能信号 IE 和中断信号进行与运算，只有在产生中断信号并且处于使能阶段才能跳转到中断服务程序中。同时要保护现场，要保护的现场即为单周期 CPU 的 PC 寄存器输入端的数据，寄存器的使能端为中断请求信号，当产生中断请求时就会将 PC 寄存器的输入端的值存到寄存器中。处理完中断服务程序后，当产生 URET 信号时则要进行中断返回，以 URET 作为多路选择的信号将断点传回 PC，利用中断号来产生相应中断的清零信号。

对于多级中断，由于支持了中断程序继续被中断，因而需要三套中断寄存器保存中断信息。使用优先编码器来对当前中断信号进行选择，若正在执行中断程序且此时允许被打断则利用中断屏蔽字和中断号的比较器判断当前中断是否允许打断当前终端，这样可以动态的改变中断优先级。

2.2.3 软件设计

不需要软件部分支撑。

2.3 流水 CPU 设计

2.3.1 总体设计

流水 CPU 的设计采用五段流水架构，框架图如图 2.4 所示。程序计数器 PC 和各个流水寄存器采用统一时钟进行同步。

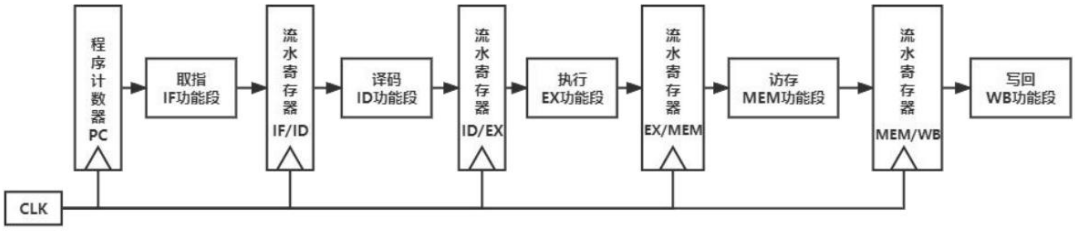


图 2.4 五段流水 CPU 框架图

五段是指将每条指令的执行过程分为五个阶段，依次为取指（IF）段、译码（ID）段、执行（EX）段、访存（MEM）段、写回（WB）段。流水是指通过各段之间的流水寄存器来锁存各段处理得到的结果并在下一时钟周期到来时传递数据与控制信号给下一个阶段。在理想状态下，这种结构能实现各阶段执行的相对独立性。

但实际的指令流水线中会存在指令相关问题，即某指令的某个阶段必须要等到它前面的某条指令的某个阶段完成后才能开始。如存在各种跳转指令时，取指令不再是按序依次，而要根据跳转指令的结果来进行。或者当前指令需要用先前指令的结果，而这个结果尚未送到指定的位置。上述控制冲突和数据冲突会导致流水线冲突，破坏了流水线各段之间的独立性。

我们将采用两种方案来解决流水线冲突：插入气泡和重定向。此外，我们将采用动态分支预测机制来减少分支延迟的损失，进一步优化流水线性能。

2.3.2 流水接口部件设计

四个流水寄存器 IF/ID、ID/EX、EX/MEM、MEM/WB 用于锁存各段处理得到的结果并在下一时钟周期到来时传递数据与控制信号给下一个阶段。流水接口部件的设计就是根据流水寄存器两端需求和结果给需要传输的数据设计对应的输入接口和输出接口。理想流水线的各流水寄存器锁存数据见表 2.5。

表 2.5 流水寄存器接口设计

流水寄存器	锁存数据说明
IF/ID	PC、IR
ID/EX	PC、IR、R1、R2、各类型指令的立即数、W#、控制信号、halt 信号
EX/MEM	PC、IR、Result、MDin、ByteSelect、W#、控制信号、halt 信号
MEM/WB	PC、IR、Result、MDout、W#、控制信号、Halt 信号

流水寄存器上升沿触发，有复位端（高电平有效）和使能端（低电平有效）以满足暂停、清空寄存器的需求。

2.3.3 理想流水线设计

理想流水线由于不涉及同时进入流水线的指令之间的相互作用，因而只需要将单周期的 CPU 按照上述五个步骤直接进行拆分，使用四组流水线接口即可。对于

停机指令，在 EX 段处理完停机逻辑后，需要经过两个周期传送到 WB 段再最终执行停机操作，以保证全部指令都执行完毕再停机。

同时理想流水线并不能支持分支指令，存在误取指令的现象，同时由于数据冲突的存在也会导致一些指令无法正确得执行。

2.4 气泡式流水线设计

为了支持同在流水线的指令的相互作用关系，在理想流水线的基础之上需要增加由当前的指令操作信号控制的 DataHazzard 组合逻辑以判定是否需要插入气泡，插入气泡等效于清空流水线接口内寄存器的值。DataHazzard 的主要判定逻辑是根据源寄存器的使用状态和数据相关检测来判定是否存在冲突。

其中 DataHazzard 信号对应的逻辑如下：

```
DataHazzard = R1Used & (R1Adr≠0) & EX.RegWrite & (R1Adr==EX.rd) + R2Used & (R2Adr≠0) & EX.RegWrite & (R2Adr==EX.rd) + R1Used & (R1Adr≠0) & MEM.RegWrite & (R1Adr==MEM.rd) + R2Used & (R2Adr≠0) & MEM.RegWrite & (R2Adr==MEM.rd)
```

具体而言，若判定需要插入气泡，则 ID/EX 接口和 IF/ID 接口均需要清空，以跳转到正确的指令处重新执行。由于指令的相互作用，气泡被插入了流水线，导致流水线的效率下降。

2.5 数据转发流水线设计

数据转发流水线（即重定向流水线）以气泡流水线的基础之上进行了一定的优化，结构类似，但是减少了气泡的插入。重定向将数据可能的全部路径进行汇总，然后根据指令的执行情况获取数据，选择一条分支。但是若相邻两条指令存在数据相关，且前一条是访存指令，此时就会出现冲突（称为 Load_Use），此时仍需要使用气泡来避免冲突。

其中 Load_Use 信号对应的逻辑如下：

```
Load_Use = R1Used & (R1Adr≠0) & EX.MemToReg & (R1Adr==EX.rd) + R2Used & (R2Adr≠0) & EX.MemToReg & (R2Adr==EX.rd)
```

同时，选择多路分支时需要的多路选择信号 R1_F、R2_F 也要根据指令的情况进行生成，以确保使用的 R1 和 R2 的正确性。

其中 R1_F、R2_F 信号对应的逻辑如下：

R1_F = (R1Used & (R1Adr≠0) & EX.RegWrite & (R1Adr==EX.rd)) ? 2 : (R1Used & (R1Adr≠0) & MEM.RegWrite & (R1Adr==MEM.rd))?1:0

R2_F = (R2Used & (R2Adr≠0) & EX.RegWrite & (R2Adr==EX.rd)) ? 2 : (R2Used & (R2Adr≠0) & MEM.RegWrite & (R2Adr==MEM.rd))?1:0

具体而言，在对应不同流水寄存器的使能端和复位端，以及 PC 寄存器的停机信号生成逻辑可以根据 Load_Use 生成。

2.6 动态分支预测机制

2.6.1 设计原理

动态分支预测是在重定向流水线上进一步的优化。重定向只是在气泡流水线上减少了气泡的产生，但是并未减少分支误取得代价，导致了周期的浪费。动态分支预测使用一个 BHT 表来记忆过去的跳转指令，使用 LRU 算法来动态更新 Cache 槽的存储内容，同时使用双预测位的四状态的状态机来判定是否需要跳转，如图 2.5 所示。不断根据当前实际跳转情况来进行状态机的更新。具体而言，电路在 IF 段通过状态判断出预测结果，在 EX 段根据实际结果通过硬件来更新状态机。通过这两个方法来实现预取得正确的指令，减少分支误取的代价。

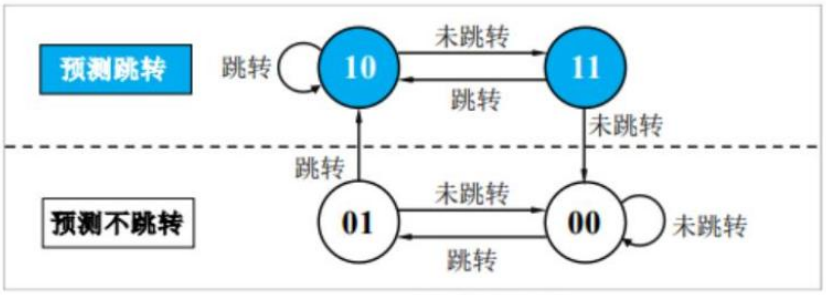


图 2.5 分支预测历史位状态转移图

2.6.2 BHT 表的设计

BHT 表全程分支预测分支历史表，存放了每次分支指令的地址、分治目标地址、历史跳转位、valid 位，硬件设计为 8 位的 Cache 槽。在具体插入地址时，优先插入空行，其次是最古老的跳转地址（即 LRU 算法）。生成 IF 段分支指令命中 L 和 EX

段分支指令命中的信号 M，在 IF 段命中时清空相应 cache 行的淘汰计数，采用 D 触发器存储以避免毛刺，EX 段命中时更新相应 cache 行的分支预测历史位，在 cache 行未满时按照 cache 行编号从大到小的顺序写入，否则则替换掉淘汰计数最大的那个 cache 行。

2.6.3 动态分支预测的流水线设计

动态分治预测在重定向流水线的基础上，增加了利用 BHT 进行预测功能。有以下三种情况：

1. BHT 未命中，则与正常重定向流水线行为一致。

2. BHT 命中但预测失败。需要插入气泡，将预取指令取出，并且更新 BHT 表。此时与正常重定向流水线行为大致一致。

3. BHT 命中且预测成功。减少了气泡的插入，IF 指令预取正确，电路继续指令。

整体而言，预测成功可以减少平均误取深度和气泡插入数目，因而效率相对于重定向流水线又有了进一步的提升。

3 详细设计与实现

3.1 单周期 CPU 实现

3.1.1 主要功能部件实现

1) 程序计数器（PC）

① Logism 实现：

使用一个 32 位寄存器实现程序计数器 PC，触发方式为下降沿触发，输入为下一条将要执行的指令的地址，输出为当前执行指令的地址。Halt 为停机信号，当需要进行停机时，Halt 控制信号为 1，经过非门之后为 0，接入计数器使能端，忽略时钟信号，使整个电路停机。如图 3.1 所示。

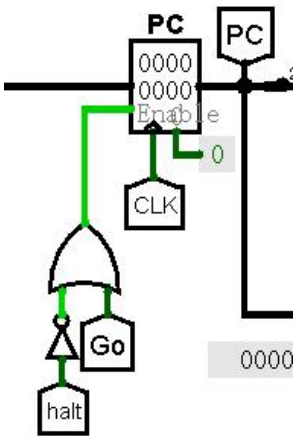


图 3.1 程序计数器（PC）

② FPGA 实现：

程序计数器 PC 的 Verilog 代码如下：

```
always @(posedge clk)
begin
    if (rst) begin
        temp <= 0;
        end
    else begin
        if (load) begin
```

```
                temp <= D;
                end
            else begin
                temp <= temp;
                end
        end
    end
    assign Q = temp;
```

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，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，使用分线器取 32 位指令地址的 2-11 位作为指令存储器的输入地址。如图 3.2 所示。

图 3.2 指令存储器（IM）

② FPGA 实现：

直接使用 Vivado 中自带的 ROM 作为指令存储器，其设置如**错误!未找到引用源。**所示。选择 ROM 的数据位宽为 32 位，因为该 ROM 的地址位宽为 10 位，所以选择 ROM 的大小选择为 1024。

指令存储器 IM 的 Verilog 代码如下：

```
pc pcmeml(im_in[11:2],im_out);
```

直接调用之前设置的 ROM 作为指令存储器，输入为指令地址的 2-11 位，输出为该指令。

3) 内存（MEM）

① Logism 实现：

电路中的内存和指令存储器一样，是由十位地址线（2-11 位）寻字，低两位寻字节的存储器，如图 3.3 所示。

图 3.3 内存（MEM）

② FPGA 实现：

内存（MEM）的 Verilog 代码如下：

```
always @(posedge clk) begin
    if (str) ramm[addr] <= din;
end

always @(rst or ld or addr) begin    // note: read memory don't need clk signal
    if (rst) begin
        for (i = 0; i < 2**11; i = i + 1) begin
            ramm[i] = 32'b0;
        end
    end
    if (ld) data = ramm[addr];
end
```

4) 寄存器（RegFile）

① Logism 实现：

RegFile 由 32 个 32 位的寄存器构成，其主要接口和功能都已经在总体设计方案中阐明。在设计包中 Logisim 的 RegFile 电路已经封装完成，因而只需要直接调用即可。Losigim 实现如图 3.4 所示：

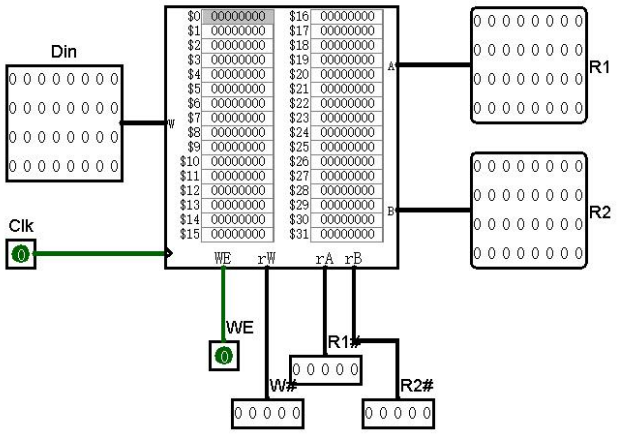


图 3.4 寄存器文件（RegFile）

② FPGA 实现：

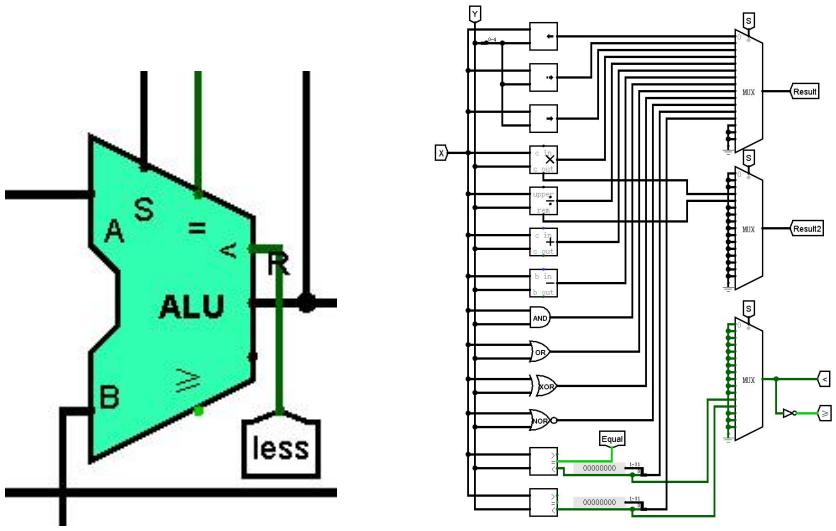
Verilog 中与 RAM 的设置相似，开辟了一块 RAM 存储空间，初始化为全 0。

```
initial begin
        for(i = 0; i <= DATA_WIDTH-1; i = i + 1) begin
            regfile[i] = 0;
        end
    end

    assign r1 = regfile[r1_addr];
    assign r2 = regfile[r2_addr];

    always @(posedge clk) begin
        if (write_en && w_addr != 0) regfile[w_addr] <= din;
    end
```

5) 运算器（ALU）

① Logism 实现：

运算器是多种计算功能的集合。通过输入端 X，Y 输入 32 位运算数据，输入端 ALUOP 输入 4 位功能号，输出端 Result 输出 32 位运算结果，其他状态输出端输出 1 位状态信号。采取并行运算的方式，对 X 和 Y 进行并行计算，通过多路选择器，输出 ALUOP 指示的输入端口作为运算结果。由于运算位数为 32 位，当存在立即数运算时，应将其扩展为 32 位的值传入 ALU 中。具体实现如图 3.5 所示。



a）ALU 封装图　　b）ALU 实现电路

图 3.5 运算器（ALU）

② FPGA 实现：

Verilog 中仅需要将每个 ALUOP 对应的计算指令进行转化即可，不再赘述。

3.1.2 数据通路的实现

根据单周期 CPU 的设计思路，将各主要功能部件构建好后，具体通过 5 个功能，取指、译码、执行、访存与写回，依次对部件进行连接，从而构造相应功能的基础数据通路。最后，再根据各类指令在不同功能上的区别，对数据通路进行细节调整。具体数据通路实现如图 3.6 所示。

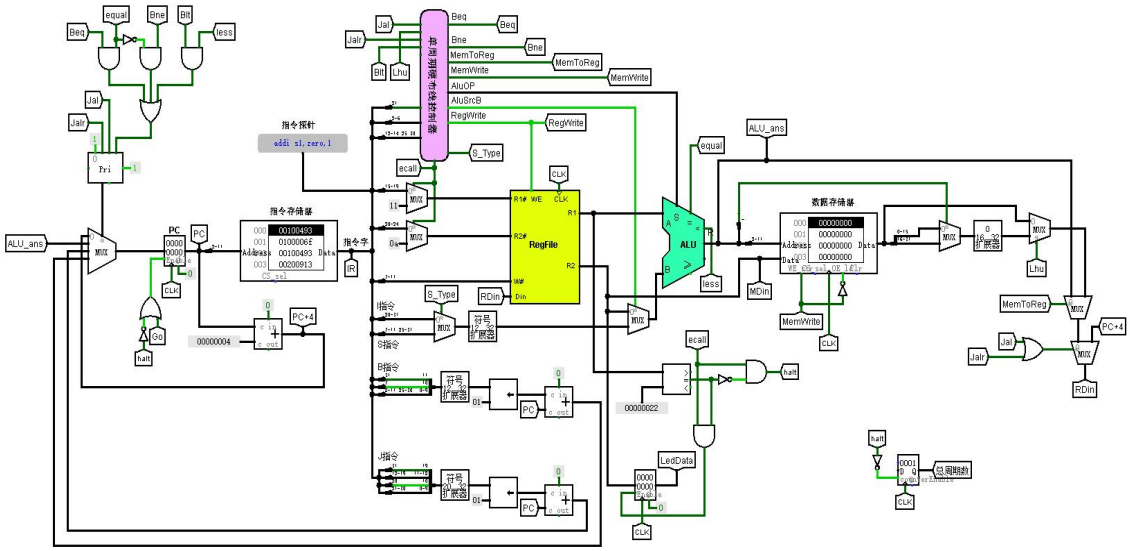


图 3.6 数据通路

3.1.3 控制器的实现

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

① Logisim 实现：

通过代码逻辑自动生成，如图 3.7。

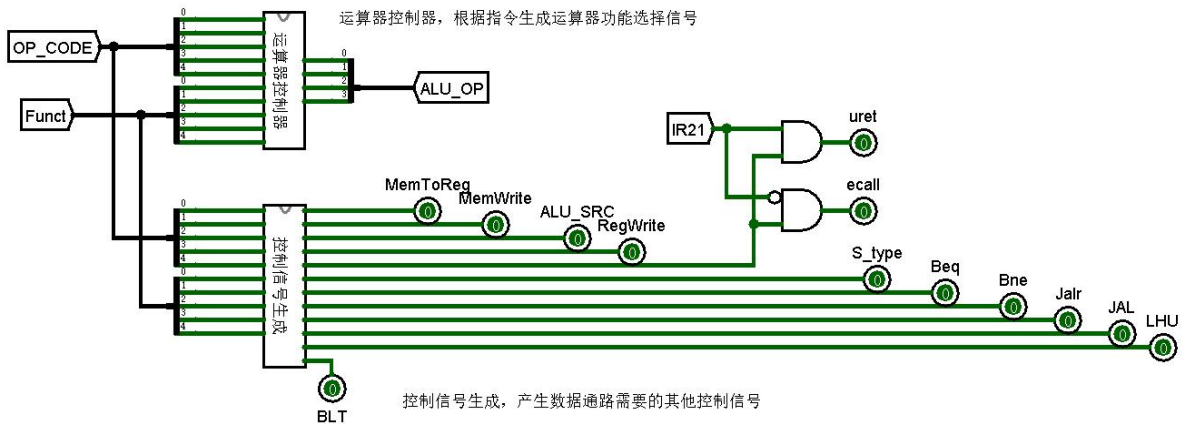


图 3.7 硬布线控制器

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，使用 assign 分的 Verilog 代码过于冗长，故只展示生成信号的示意图，如图 3.8。

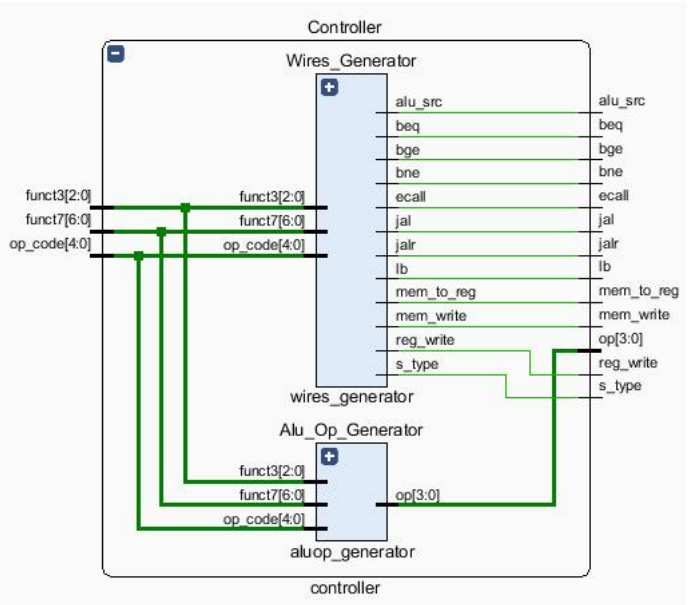


图 3.8 主控制器原理图

3.2 中断机制实现

3.2.1 单级中断

需要实现中断控制器，在产生中断的下一个上升沿输出中断请求与中断号， 这就需要两个寄存器，第一个寄存器在中断源产生脉冲时立刻载入 1，第二个中断再下一个上升沿时载入 1 并情况第一个寄存器，中断号借助优先编码器输出，中断信号通过三个寄存器的或运算得到，具体中断控制逻辑如图 3.9 所示。

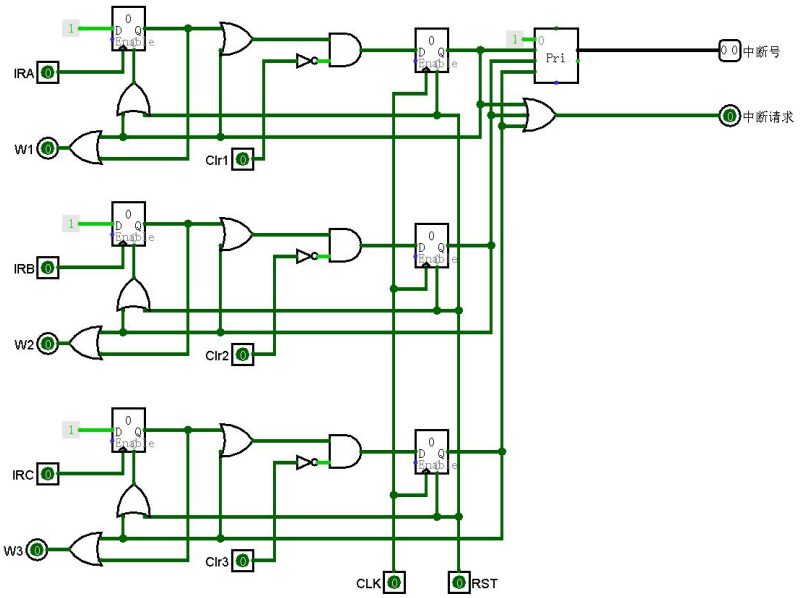


图 3.9 中断控制逻辑

硬件实现上需要通过多路选择器生成中断入口逻辑，中断源地址可以通过 RARS 仿真器得到具体地址，通过使用 EPC 和 IE 寄存器来保存当前中断的状态和返回地址，通过 URET 和中断信号决定生成中断清零信号，清空中断请求队列。具体的硬件实现如图 3.10 所示。

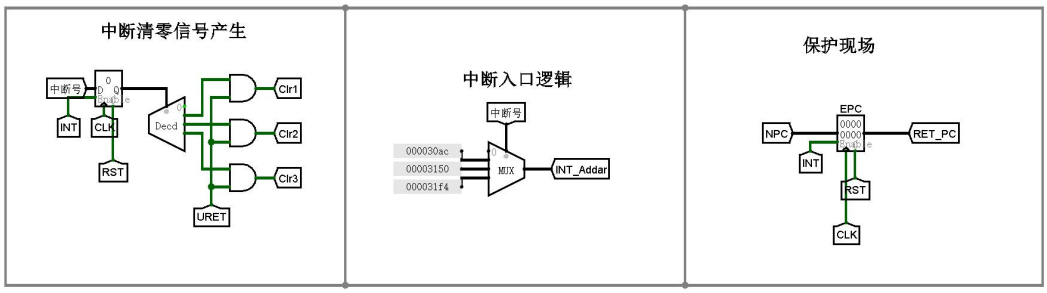


图 3.10 中断信号产生、中断入口逻辑和保护现场

中断处理同时也需要更改部分数据通路以实现中断服务，具体如图 3.11 所示，主要更改的便是 PC 寄存器相关的数据通路，具体如所示。其中 NPC 代表原数据通

路中的 PC 输入，N_PC 代表支持中断服务后的 PC 输入。

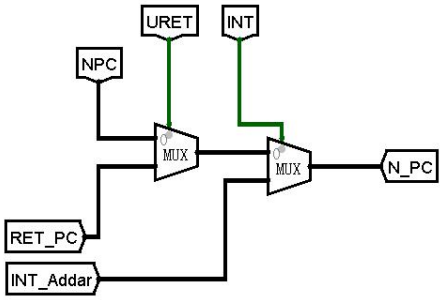


图 3.11 支持中断服务的数据通路更改

3.2.2 多级中断

多级中断中利用硬件保存的实现方法使用三套 EPC 和 IR 寄存器来保存中断号和返回地址，中断控制逻辑与单周期相同，并同时采用 CSRRCI 和 CSRRSI 来控制中断使能信号，具体而言是利用这两条指令来控制 IE 寄存器的信号，IE 寄存器初始为 1，高电平表示开中断。控制逻辑如图 3.12 所示。

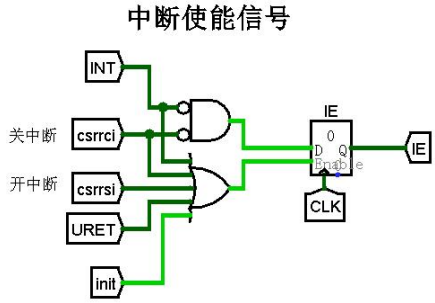


图 3.12 中断使能信号发生逻辑

硬件实现上生成中断入口逻辑方式和生成中断清零信号的方式与单级中断相同，也同样通过使用 EPC 和 IE 寄存器来保存当前中断的状态和返回地址，但是由于中断可以嵌套，所以需要多个 EPC 和 IE 寄存器进行中断保护。同时由于中断存在优先级，因此需要中断比较逻辑确定当前需要执行的中断服务，与单级中断差异化的硬件实现如图 3.13 所示。

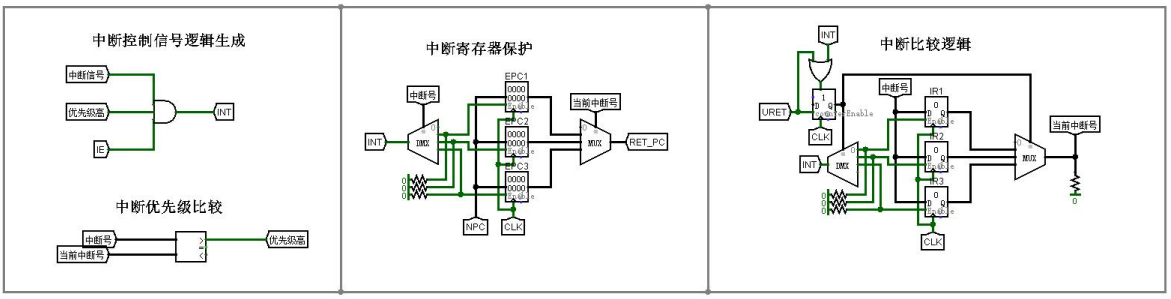


图 3.13 中断寄存器保护、中断优先比较、中断优先逻辑

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

流水接口部件所要传递的信号已经在上文列出， 由此我们可以设计 IF/ID、ID/EX、EX/MEM、MEM/WB 四个流水寄存器，以 MEM/WB 为例，各个信号通过寄存器存储，各个寄存器上升沿触发，RST 信号可以同步复位，如图 3.14 所示。

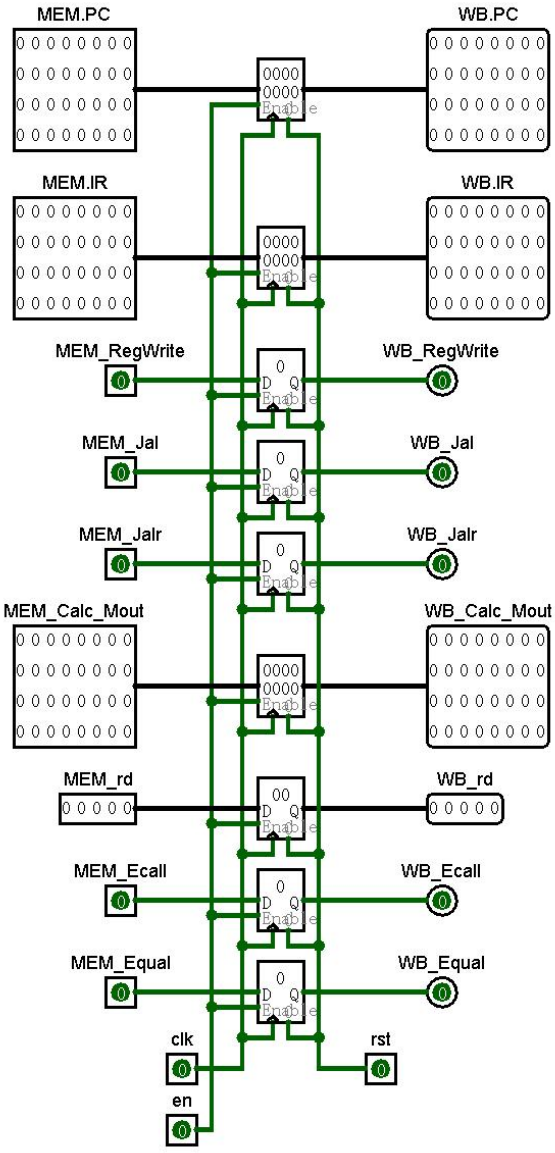


图 3. 14 流水接口部件

3.3.2 理想流水线实现

理想流水线是在单周期 CPU 的基础上，将各部件分成五个部分，每个部分使

用流水线接口进行拼接。主要模块的分布为：（1）IF 段：程序计数器 PC、指令存储器 IM；（2）ID 段：寄存器堆 RF；（3）EX 段：运算器 ALU；（4）MEM 段：数据存储器 MEM；（5）WB 段：写回数据。具体的理想流水线如图 3.15 所示。

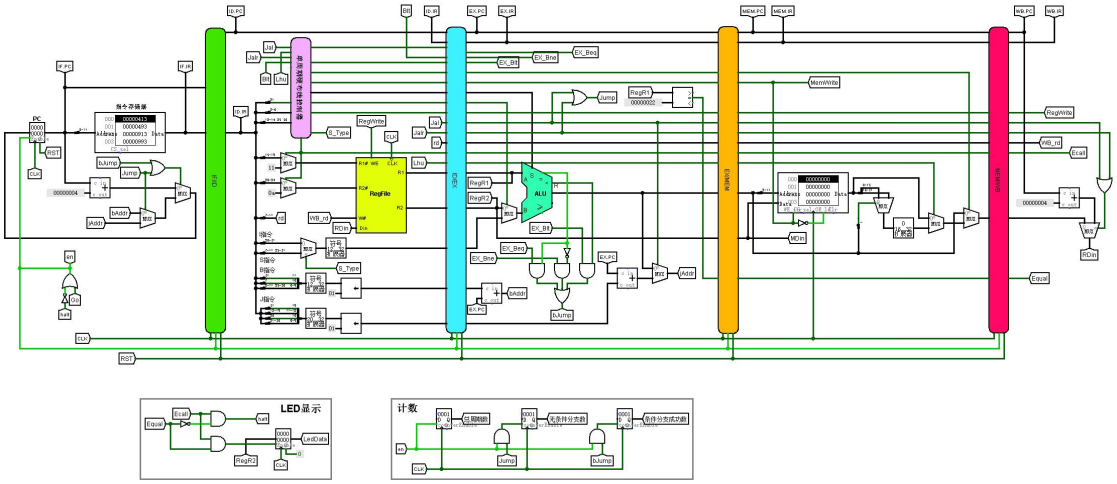


图 3.15 理想流水线结构图

3.4 气泡式流水线实现

气泡流水线就是在理想流水线的基础之上，根据当前电路的状态（RS1 和 RS2 是否被使用，WriteReg、RegWrite）增加 DataHazzard 逻辑判断，其具体组合电路如图 3.16 所示。

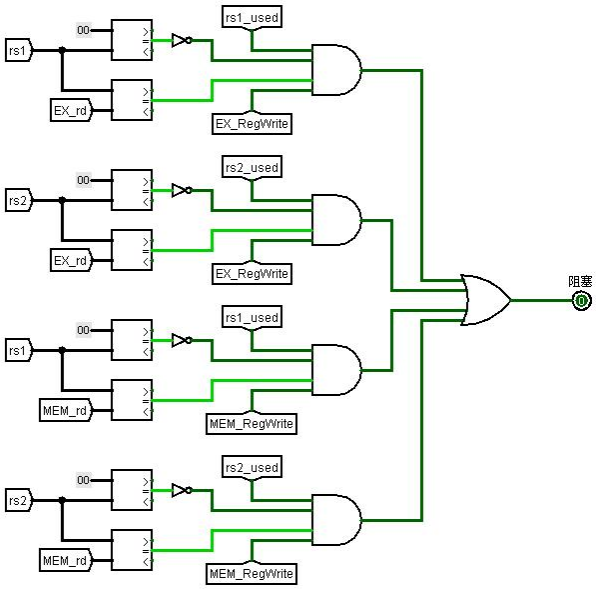


图 3.16 DataHazzard 组合逻辑

具体的气泡流水线如图 3.17 所示。

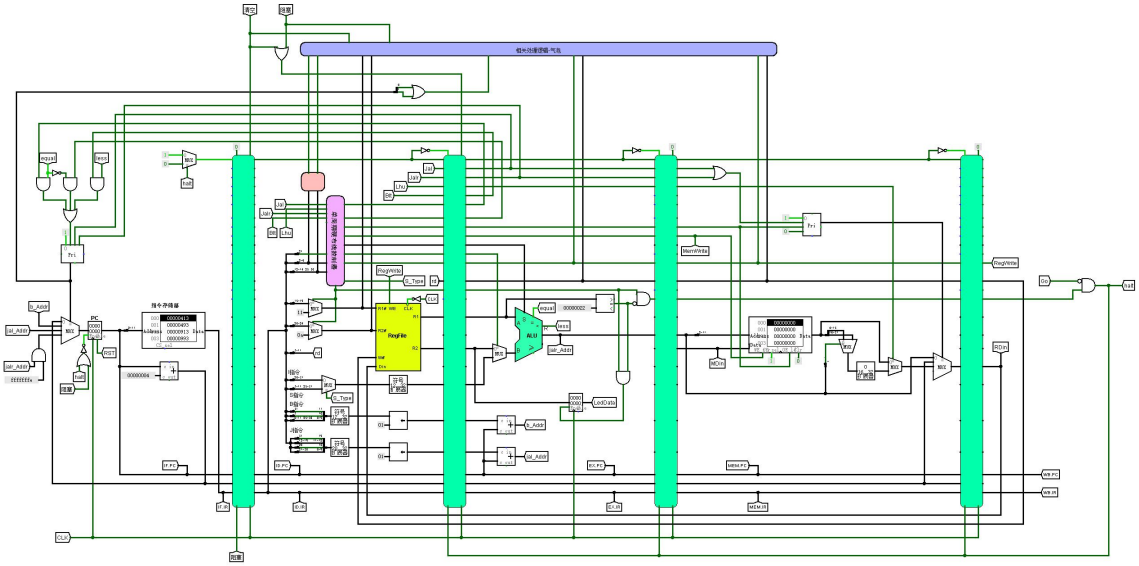


图 3.17 气泡流水线结构图

3.5 数据转发流水线实现

重定向流水线（数据转发流水线）就是在理想流水线的基础之上，根据当前电路的状态（RS1 和 RS2 是否被使用，WriteReg、RegWrite）增加 LoadUse、rs1_forward、rs2_forward 逻辑判断，其具体组合电路如图 3.18 所示。

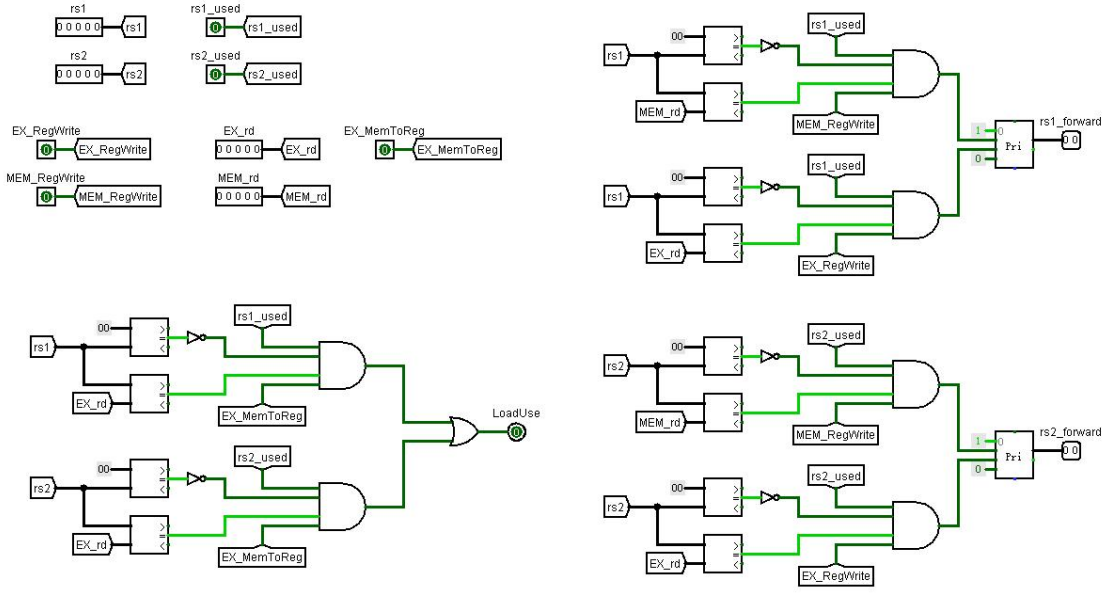


图 3.18 LoadUse、rs1_forward、rs2_forward 逻辑电路

3.6 动态分支预测机制实现

动态分支预测需要在重定向流水线的基础上增补 BHT 表，利用寄存器设计 8

位 Cache 槽，实现一个有限状态机用于支持分支预测历史位的状态转移，状态机的转移过程，BHT 表的设计实现如图 3.19 所示。

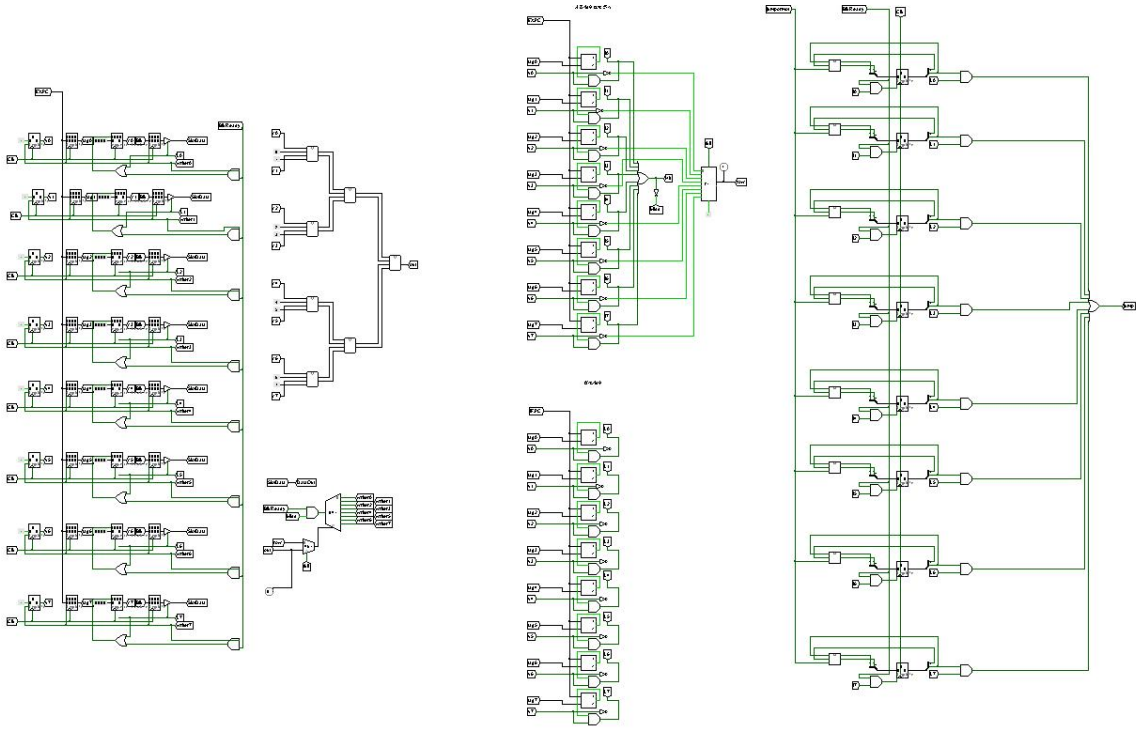


图 3.19 BHT 表结构图

与此同时，PC 端的入口逻辑也要进行大量的修改，通过分支预测的正确性和预测结果进行多路选择，以确保实现动态分支预测的入口地址修改和纠错等，需要同时修改流水器接口，增加分支预测的判断结果，和 EX 段的真实分支预测结果进行比较，确定转换逻辑是否正确，修改流水器接口的方法比较简单，在这里不列出其具体的修改结构图。PC 端入口具体逻辑如图 3.20 所示。

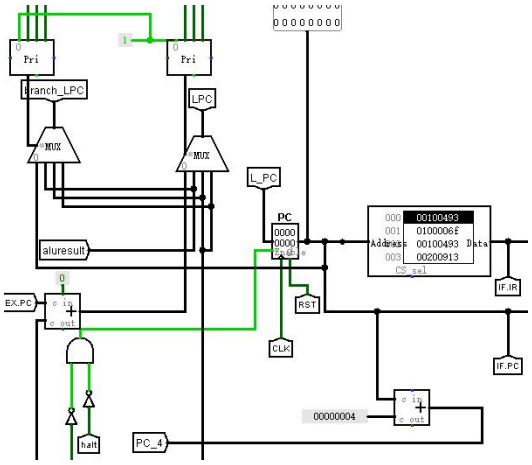


图 3.20 PC 端入口具体逻辑结构图

4 实验过程与调试

4.1 测试用例和功能测试

用测试用例 risc-v-benchmark_ccab 来测试各种 CPU（单周期、气泡、重定向、动态分支预测）的指令实现情况。用测试用例 risc-v 中断测试程序（走马灯）来测试单级和多级中断。理想流水线为一个基本架构，在测试气泡流水线与重定向流水线时其实就包含了对理想流水线的测试，故省略。

4.1.1 ccab 指令测试

四条 CCAB 指令（SRA、SLTU、LHU、BLT）单独测试结果均正确。

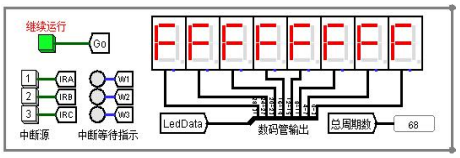


图 4.1 SRA 指令测试

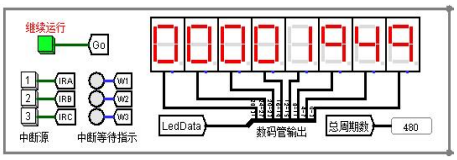


图 4.2 SLTU 指令测试

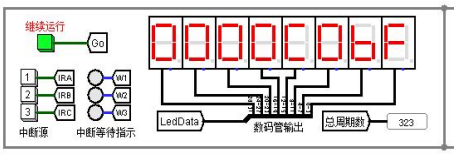


图 4.3 LHU 指令测试

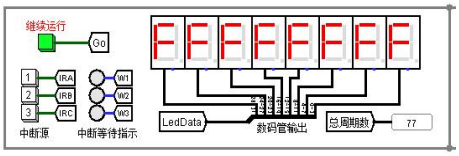


图 4.4 BLT 指令测试

4.1.2 测试用例 risc-v-benchmark_ccab

将带 risc-v-benchmark_ccab.hex 载入指令存储器。基础命令测试完毕后会运行到第一个停机的 ecall 指令，点击继续运行按钮，可以依次测试 4 条扩展指令，LED 显示符合预测值，测试成功。以下为 risc-v-benchmark_ccab.hex 在单周期（图 4.5）、气泡（图 4.6）、重定向（图 4.7）、动态分支预测（图 4.8）的运行结果。

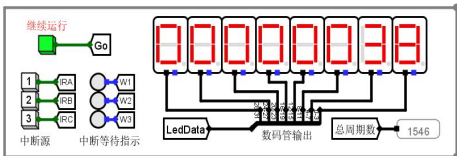


图 4.5 单周期测试

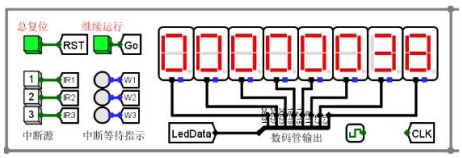


图 4.6 气泡流水线 CPU 测试

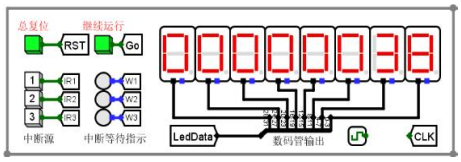


图 4.6 重定向流水线 CPU 测试

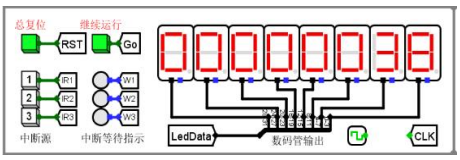


图 4.8 动态分支预测测试

4.1.3 单级中断测试

加载“risc-v 单级中断测试程序.hex”，主程序运行时快速依次按下中断“1，2，3”，执行顺序为“1，3，2”，最后返回到主程序执行，符合预期。下面是运行时截图。

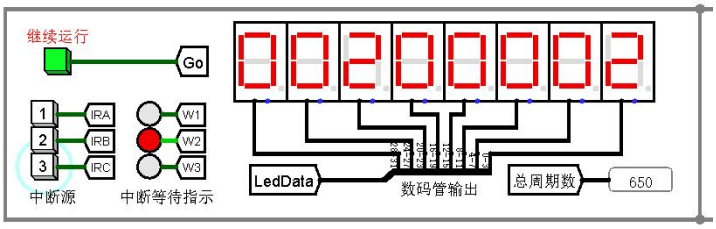


图 4.9 单级中断测试结果

4.1.4 多级中断测试

加载“risc-v 多级中断测试(EPC 硬件堆栈保护).hex”，主程序运行时依次按下中断“1，2，3”， 执行顺序为“1，2，3，2，1”。

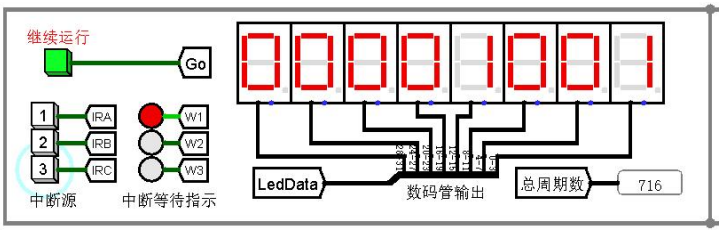


图 4.10 多级中断测试结果

4.2 性能分析

各种 CPU 运行不带 ccab 的 benchmark 程序的总周期数如表 4.1。

表 4.1 benchmark 运行总周期数统计

CPU	benchmark 程序运行总周期数	CPU	benchmark 程序运行总周期数
单周期	1546	重定向流水线	2298
气泡流水线	3624	动态分支预测	1784

单周期 CPU 总周期数为 1546，但是每周期长度较长，估算可认为其时钟周期约为 5 段流水 CPU 时钟周期的 5 倍，所以单周期 CPU 性能远远差于 5 段流水 CPU。对比各种流水线 CPU：气泡流水线性能较差，运行总周期数为 3624。重定向流水线优化了数据冲突时延，运行总周期数为 2298，比气泡流水线减少 1/3。动态分支预测则优化了分支冲突时延，运行总周期数为 1782，比重定向流水线又减少了 1/4。

4.3 主要故障与调试

4.3.1 ecall 指令的停机问题

故障现象：无 CCAB 的常规程序执行到最后那个本该暂停的 ecall 指令时停不下来。在头哥上进行测评得到结果：PCerr 11、Werr 01000。经查发现 PC 期望为 2dc，但实测是 2e0；RegW 期望是 0，但实测为 1。

原因分析：分析头哥报错中 PC 期望值与实测值的差别，发现 PC 继续向后面取了一个，那么应该是 halt 信号出了问题。halt 信号的产生如图 4.11，当 R2（$a7）的值等于 0x22 时，halt 信号寄存器的输入端便为 1，等到下一个时钟周期来临，halt 信号便被置一，问题就出在这个下一个时钟周期。在这里放一个寄存器是为了实现 Go 功能，寄存器默认是上升沿有效。上升沿有效的一个问题就是，当前时钟周期内执行 ecall 指令，halt 寄存器的输入端被置 1，但是 halt 信号没被置 1，当下一个时钟上升沿来临时，PC 的值会和 halt 信号的值同步被更新，所以导致本该暂停的 PC 又往下去了一个。

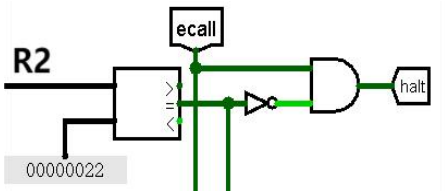


图 4.11 halt 信号的产生

解决方案：将存储 halt 信号的寄存器设为下降沿有效。

4.3.2 halt 信号生效问题

故障现象：部分指令无法完成，停留在流水线中。

原因分析：ID 由 ecall 信号产生 halt 信号，如果在 EX 段直接执行停机，则会导致后续 MEM 段和 WB 段的指令停留在流水线中，没有执行完毕。

解决方案：将 halt 信号通过流水线一直向后传递到 WB 段，再反馈到 PC 和流水线寄存器执行停机。

4.3.3 中断返回故障

故障现象：在执行多级中断服务时，对于新来的中断服务程序处理后找不到原来的中断服务程序继续执行。

原因分析：在多级中断服务的中断号确认时，没有能够正确保存中断号，导致处理过新的中断服务程序后，没有将中断号进行正确的找回，保存好的现场也就没有正确恢复。

解决方案：增添中断号保存寄存器，将中断号进行保存，在处理过新的中断服务程序后找回正确的中断号继续执行。

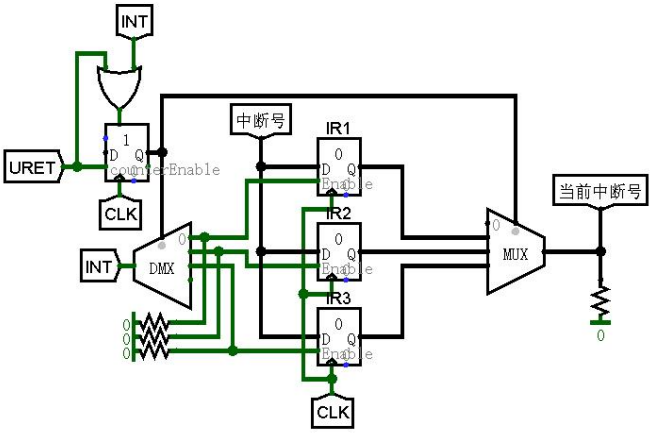


图 4.12 中断保存中断号结果图

4.4 实验进度

表 4.2 课程设计进度表

时间	进度
第一天	复习组成原理 CPU 相关理论知识，阅读课设任务书，阅读 RISC-V 指令手册，并列出 CPU 各部件的数据通路表，并完成数据通路的基本构建。
第二天	完成单周期 CPU 的控制信号表，使用 Logisim 搭建控制器，实现了单周期 CPU 并且通过了测试。完成部分 Logism 单周期 CPU 故障报告。

时间	进度
第三天	完成 Logism 单周期 CPU 的故障报告，并且通过了 Logism 单周期 CPU 的检查。使用 Verilog 实现了部分单周期 CPU 的重要部件，并通过仿真检查。
第四天	继续使用 Verilog 进行实现单周期 CPU 的工作，完成了所有部件的编写、控制器的编写，以及所有部件以及控制器的仿真测试，正在进行数据通路的拼接。
第五天	使用 Verilog 完成单周期 CPU 数据通路的连接，并且通过仿真测试。
第六天	完成 CPU 电路的功能仿真和时序仿真。
第七天	复习关于指令流水线的知识点，完成理想流水线的 Logism 编写，正在调试。
第八天	调试成功理想流水线 Logism。完成冒险处理中的数据冲突处理和分支处理。
第九天	完成冒险处理中的数据冲突和分支处理，完成数据重定向的 Logism 实现。
第十天	完成数据重定向，成功实现动态分支预测，预测成功率显著提高。

5 团队任务

5.1 项目选题

我们团队任务的选题是，利用 logisim 的单周期单级中断 CPU 电路，实现一款简单的俄罗斯方块游戏。

在此简单介绍一下俄罗斯方块的规则。在我们构思的俄罗斯方块游戏中，一共有四种类型的方块，其形状分别类似于大写字母 L、I、S 与 T，这也是我们团队取名为“LIST”的原因。系统将随机生成方块，玩家可以通过 W（翻转）、S（快速下降到最底部）、A（左移）、D（右移）四个键控制方块的移动。同时，玩家可以在任意时刻按下 R 以重玩。

在游戏未结束的情况下，一个方块抵达最终位置，可能会导致很多行被填满；这些行会被消除并由上层替代。一次消除的行数越多，每行贡献的分数越大，具体而言，每行贡献的分数是（1+ 行数）/2。

5.2 团队分工

该项目可分为软件、硬件两大模块。我主要负责软件设计，软硬件调试。

5.3 项目设计

5.3.1 软件设计

- 编写俄罗斯方块的 C 源代码，编译，再反汇编成汇编代码 test.s
- 硬件综合训练的 CPU 支持的指令有限，将未知指令用已知指令等效替代
- 把 test.s 复制到 RARS 汇编，导出十六进制字符表示的机器码
- 将机器码加载到 logisim CPU 的指令存储器，设置六个中断函数入口

5.3.2 硬件设计

- 主控 CPU：负责实现软件执行所需的所有指令程序
- 点阵显示逻辑：利用 32 个 32 位的寄存器实现界面显示
- 键盘输入逻辑：检测输入的按键对应的数字进行匹配

• 中断逻辑：转动方向，下移，左移，右移

5.4 项目实现

5.4.1 C 代码转 riscv 机器码的实现

• 在 test.c 文件中编写俄罗斯方块的 C 源代码，确认正确无误；

• 将 test.c 汇编为 test.s；

• 对 test.s 进行处理，主要包括：

1） 删除 objdump 导出的冗余提示信息

2） 将所有标签的尖括号删去

3） 在最前面加一个指令”jmain”

4） 将课设 CPU 不支持的指令用已实现的指令替代

5） 将中断处理函数最后的”ret”指令更换为 uret”

6） 对汇编代码的函数调用处进行手动重定向

• 把处理后的 test.s 中的内容复制到 RARS 进行汇编，即可得到机器码文件。得到机器码后只需将其加载到指令存储器中，并且设置六个中断函数入口即可。

5.4.2 C 代码的编写

C 代码的逻辑较为简单，不赘述。重点说明一下我们对特定问题的解决方案：

• 全局变量全部存入一个数组中：该 C 程序涉及到很多全局变量，比如用于定位方块位置的 x、y 坐标，随机数种子 seed，表示显示屏状态的二维表，以及表示游戏状态的 stop、start 标签等。但是我们并没有将其分开声明为多个变量，而是将其全部存放到一个数组 table[17][16]中。

• 5.4.1 中已经说明，我们是从可重定位文件 test.o 中获取 test.s 汇编文件的，这意味着需要手动完成所有的重定位工作。而如果直接按照类似“intstop,start;”这样的方式声明全局变量，这些变量在 test.o 中的地址都是 0，从而增加了我们进行重定位的难度。而如果使用一个表格 table，那么 table[0][0]的地址是 0，table 中其余元素的地址都会按照偏移量来计算，无需对全局变量符号进行重定位。

•与硬件沟通的 ecall 指令在 a7 寄存器为 34 时（否则调用 ecall 会停机），调用 ecall 指令可以触发特殊的硬件动作，调用号存放在寄存器 a0 中。

• 不同调用号具有不同的功能：调用号为-1：将硬件的随机数种子加载到 CPU 内存中。调用号为-2：清空 LEDMatrix 屏。调用号为-3：显示 Presswsad 的图案，表示游戏可以开始了。调用号为-4：显示 GameOver 的图案，表示游戏结束。

5.4.3 硬件实现

硬件设计的最外层是 I/O 模块，用于和用户的交互。其中 I 模块是一个键盘解析电路，使用 logisim 自带的 Keyboard 组件，将用户的按键转换为虚拟键码，然后翻译为对应的按键控制信号，如图 5.1 所示。同时，使用 logisim 自带的 LEDMatrix 进行屏幕输出（O 模块），如图 5.2 所示。

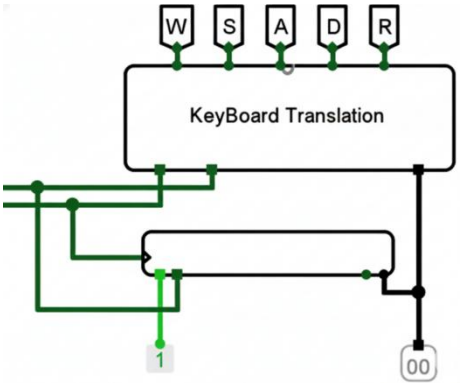


图 5.1 input 模块

图 5.2 output 模块

同时，周期性随机信号的生成如图 5.3 所示。该图的含义是，每 0x300 个时钟周期，产生一次 fall 中断信号。

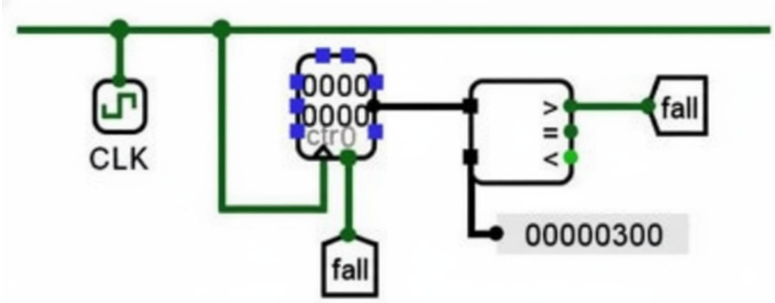


图 5.3 周期行 fall 信号产生

分数显示的逻辑则更为简单。在我们确定了 score 变量在 table 中的位置后，即确定了 score 的地址。因此只需将 CPU 的 Memwrite，Addr 和 MDin 引出，当 Memwrite 为 1 且 Addr 是 score 的地址时，即可将 Mdin 写入分数寄存器。

5.5 项目测试

俄罗斯方块是一个动态的游戏过程，过程中会出现很多不同的显示界面。限于篇幅，这里简单测试一下 WSA 三个按键的功能，gameover 的判定。

5.5.1 W 按键测试

首先启动时钟，开启游戏，可以看到 LEDMatrix 显示“PressWSAD”字样，如图 5.4 所示，当出现方格后我们按动 w 键，方块将顺时针旋转 90 度，见图 5.5。

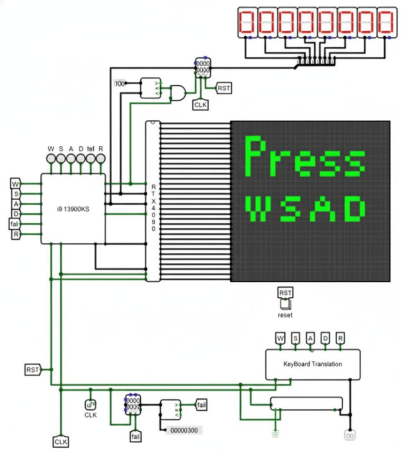


图 5.4 游戏开始界面

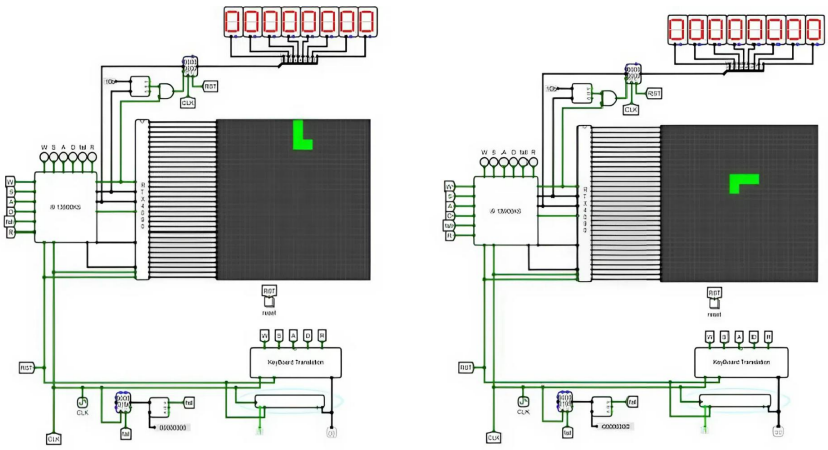


图 5.5 翻转前后示意图

5.5.2 A 按键测试

在上述操作之后，一直按下 A，方块被移动到最左端，见图 5.6。再次按下 A，观察到方块没有继续向左移动；同时指示灯 A 亮的时间变短（中断处理过程变短），

说明中断处理函数直接返回了。这都说明 A 按键的功能正常。

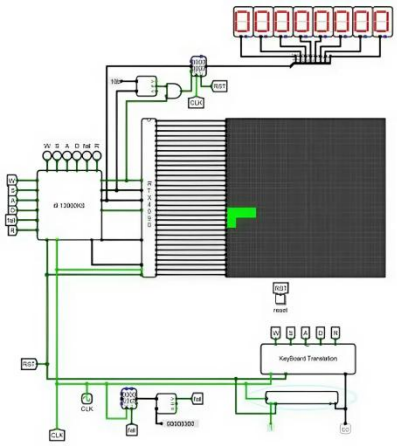
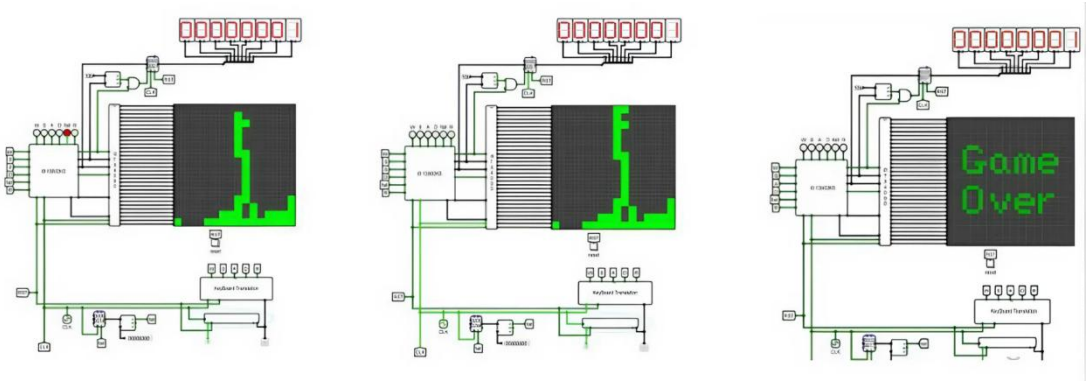


图 5.6 方块移动到左侧

5.5.3 GameOver 的判定测试

当我们游戏失败的时候将出现 GameOver 的显示，见图 5.7，表示游戏结束。



5.6 项目展望

我们的团队任务效果整体不错。游戏过程中，最主要的问题就是方块移动太慢，根本问题是 logisim 的时钟频率过低。以后如果有机会，将考虑项目上 FPGA 开发板，提高指令的执行速度，提高游玩体验。同时，由于一些不可控因素，随机数种子的获得与随机数生成模块没能按计划完成，实际上方块是轮流生成的。如果有机会，将投入更多的时间解决这一难题，实现真正的方块随机生成。

6 设计总结与心得

6.1 课设总结

在本次硬件综合训练中，我完成了如下几点工作：

1）设计实现了支持 24 条基础指令和 4 条扩展指令的单周期 CPU。

2）设计实现了理想流水线、气泡流水线和重定向流水线这三种 5 段流水 CPU。

3）设计实现了单周期 CPU 的单级中断、多级中断和 5 段流水 CPU 的单级中断。

4）设计实现了包含动态分支预测机制的重定向流水线 CPU。

5）完成了 educoder 平台的测试与线下的检查。

6）设计实现了团队任务。

6.2 课设心得

本次硬件综合训练可以说是迄今为止所有实验和课程设计中难度最大、工程味最浓的一次。从搭建支持 24 条基础指令和 4 条扩展指令的单周期 CPU，到逐步完善理想流水线、气泡流水线、数据重定向流水线，再到加入多级中断和动态分支预测机制，最后还要在 Educoder 平台上通过测试并完成团队任务，中间经历了无数次推翻、修改和调试。回过头再看这两个多星期几乎“连轴转”的日子，既有熬夜的疲惫，也有 CPU 终于一次性跑通 benchmark 和中断测试时那种发自内心的成就感，更重要的是，对“工程化设计”和“完整系统”的理解，比以前任何一门课都要立体、具体得多。

课程设计刚开始阶段，Logisim 下的单周期 CPU 实现相对顺利，但真正的考验其实出现在用 Verilog 重新描述整个数据通路和控制逻辑的时候。与可视化连线不同，代码实现时一切都要通过端口和 wire 来连接，一开始我对端口命名和信号规范并不重视，很多信号名字既不统一也不直观，结果在连线完成后，花了大量时间对照电路图逐根排错，常常因为一个方向搞反或一个位宽不一致导致整体功能异常。最后不得不“回炉重造”，把所有关键模块和信号重新规范命名、梳理层次结构，在这个过程中真切体会到：工程化、模块化和命名规范不是形式，而是保证复杂系统可读、可改、可调试的前提，这一点无论在硬件设计还是软件代码中都非常重要。

在流水线 CPU 部分，理想五段流水的结构并不复杂，但一旦加入气泡插入、数据重定向以及多级中断，问题就开始“成片”地暴露出来了。书本和 PPT 对气泡流水线、重定向流水线以及 Load-Use 冲突的处理都只是停留在概念层面，真正落实到具体的控制信号、流水线寄存器清空策略、PC 选择路径时，就只能一边查资料、一边和同组同学讨论，一边不断猜测和验证。像 halt 信号晚一个周期生效、ecall 导致流水线尾部指令不能正常“冲完”流水、多级中断返回后找不到原来的中断服务程序继续执行等问题，都迫使我去重新梳理每一级流水寄存器中到底应该保存哪些状态、哪些信号要一路向后传递、哪些需要在某一级被清空或屏蔽。虽然过程很痛苦，但也极大地锻炼了我自主查阅文献、独立分析问题和在团队中交流思想的能力。

团队任务部分则是本次课设中最富有趣味的一环。我们选择了在 Logisim 的单周期中断 CPU 上实现俄罗斯方块游戏，从 test.c 开始编写 C 源程序，到生成汇编、转成机器码，再加载到指令存储器中运行，同时还要设计随机数种子获取、中断控制按键、LED 点阵显示等外围逻辑。这个过程让我第一次比较完整地体验到“软件—汇编—机器码—硬件”的贯通：一条 C 语句最后如何变成若干条我们自己实现的指令、ECALL 是如何触发硬件动作、方块移动和旋转又是怎样依赖于中断和键盘输入的。当看到 LEDMatrix 上的方块按照预期规则移动、下落，并在游戏结束时正确显示 Game Over，那一刻会真切地感觉到：之前在单周期、流水线、中断和分支预测中投入的所有细致工作，最终都汇聚成了一个有趣、能和人交互的“完整系统”，这种软硬件协同的成就感是非常难忘的。

最后，非常感谢几位老师在整个硬件综合训练过程中对我的耐心解答和细致指导，也感谢同学们在调试、查错和资料分享过程中的互相帮衬。这次课程设计不仅让我对计算机组成原理有了更深刻、更具体的理解，也让我第一次真正体会到设计一套完整 CPU 及其应用系统所需要的耐心、细致和合作精神。我相信，这段紧张而充实的课设经历，一定会成为我大学阶段极其宝贵、也非常难忘的一段回忆。

参考文献

[1] DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第5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] 谭志虎，秦磊华，吴非，等.计算机组成原理（微课版 第2版）. 北京：人民邮电出版社，2025年.

[4] 谭志虎，周健，周游.计算机组成原理实验指导（基于RISC-V在线实训）. 北京：人民邮电出版社，2024年.

[5] 曹强，施展.计算机系统结构（微课版）. 北京：人民邮电出版社，2024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：王家乐