华中科技大学

2022

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：	CS2006
学　　号：	
姓　　名：	魏子腾
电　　话：	
邮　　件：	

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 24 条基本 32 位 RISC-V 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 基本指令集

#	RISC-V	指令类型	简单功能描述	备注
1	ADD	R	加法	指令格式与功能请参考 RISC-V32 指令集英文手册，或参考 RARS 模拟器
2	ADDI	I	立即数加	
3	AND	R	与	
4	ANDI	I	立即数与	
5	SLLI	I	逻辑左移	
6	SRAI	I	算数右移	
7	SRLI	I	逻辑右移	
8	SUB	R	减	
9	OR	R	或	
10	ORI	I	立即数或	
11	XORI	I	或非/立即数异或	
12	LW	I	加载字	
13	SW	S	存字	

#	RISC-V	指令类型	简单功能描述	备注
14	BEQ	B	相等跳转	
15	BNE	B	不相等跳转	
16	SLT	R	小于置数	
17	SLTI	I	小于立即数置数	
18	SLTU	R	小于无符号数置数	
19	JAL	J	转移并链接	
20	JALR	I	转移到指定寄存器	
21	ECALL	I	系统调用	中断相关，可简化，选做
22	CSRRSI	I	访问 CSR 寄存器	中断相关，可简化，选做
23	CSRRCI	I	访问 CSR 寄存器	异常返回，选做
24	URET	I	中断返回	指令格式与功能请参考 RISC-V32 指令集英文手册，或参考 RARS 模拟器
25	SRL	R	逻辑右移	
26	SLTIU	I	小于无符号立即数置数	
27	SB	S	存字节	
28	BLT	B	小于跳转	

2 总体方案设计

2.1 单周期 CPU 设计

本次设计的是支持 24 条基本 RISC-V 指令和 4 条差异化指令的单周期 CPU，采用硬布线控制方式，指令存储器和数据存储器分离的哈佛架构，依托原理图连接各个模块，首先实现 R 型指令的数据通路，接着在此基础上根据 S、B、J 型指令的功能完善数据通路，根据各个指令功能在 Excel 表格中进行控制信号填写，自动生成硬布线控制器，最后添加新的控制信号和数据通路来实现差异化指令。

单周期 CPU 的总体结构图如图 2.1 所示。

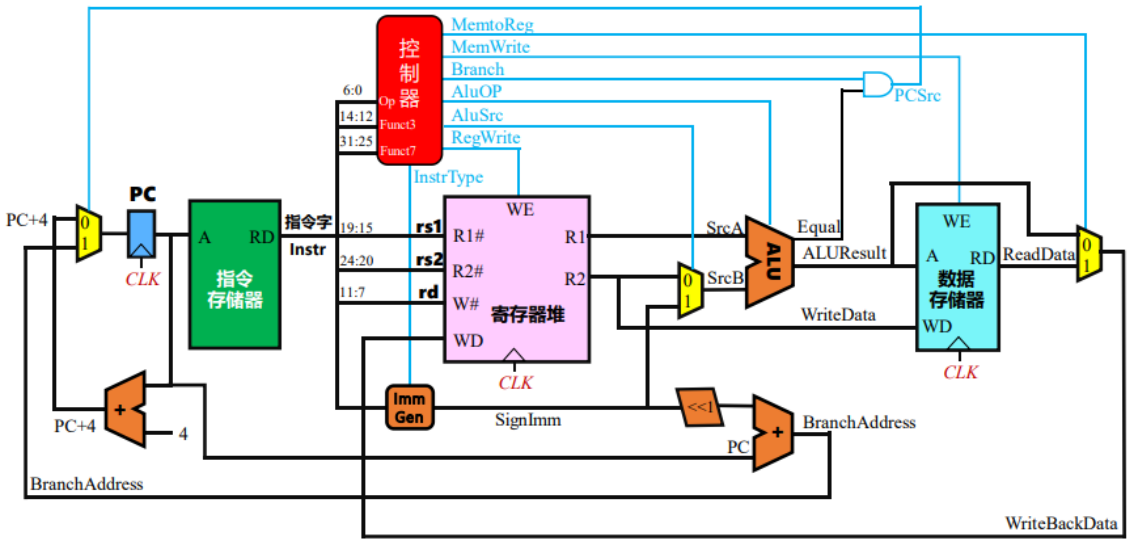

图 2.1 单周期 CPU 总体结构图

2.1.1 主要功能部件

1. 程序计数器 PC

程序计数器 PC 的功能是存储指令的地址，在时钟为上升沿时将下一条指令的地址载入寄存器并输出给指令寄存器 IM，根据指令的不同，下一条指令的地址也有所不同，本次设计中主要包括四种地址：顺序地址 PC+4、B 型指令分支跳转地址 Branch_Adr_B、J 型指令分支跳转地址 Branch_Adr_J、JALR 跳转地址 JALR_Adr。

同时 PC 寄存器的另一个功能是控制 CPU 停机，将停机信号连接在寄存器使能

端，在产生停机信号后 PC 不会再载入新的指令地址，即整个 CPU 将会循环执行最后一条指令 ecall，而 ecall 不涉及对任何模块的写入操作，由此我们借助 PC 寄存器实现了 CPU 的停机操作。

2. 指令存储器 IM

指令存储器 IM 是一个地址位宽为 10，输出位宽为 32 的 ROM，将我们所要运行程序的数据镜像载入到 ROM 中即可在我们设计的单周期 CPU 中运行该程序。

3. 运算器

ALU 模块有两个输入引脚 X、Y 即参与运算的两个 32 位数据，输出引脚 Result 为运算结果，我们不需要用到 Result2，equal 是 X、Y 相等的信号并且在任何运算中有效，而 < 和≥信号则是 X<Y 和 X≥Y 的信号并且它们只在第 11 和 12 号运算中有效。

运算器的规格如表 2.1 所示。

表 2.1 运算器规格

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	$Result = (X * Y)_{[31:0]}$; $Result2 = (X * Y)_{[63:32]}$ 无符号乘法
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

在 RISC-V 中有 32 个寄存器，而这 32 个寄存器的值则存储在具有读写功能的寄存器堆 RF 中，寄存器堆有两个输出寄存器值的引脚，这是为了处理 R 型指令，相应的它也有两个读地址输入引脚；WE 信号控制寄存器堆将数据写入写地址对应的寄存器中，寄存器堆在单周期 CPU 中是上升沿触发，而在气泡和重定向流水线中是下降沿触发。

5. 数据存储器 MEM

数据存储器 MEM 用于支持 lw 和 sw 以及我的差异化指令 sb，它是一个地址位宽为 10，输出位宽为 32 的 RAM，因为 RISC-V 中的 32 个寄存器往往是不够用的，在编程中经常要用到其他的变量，因此 CPU 需要有另外一个空间存储数据，为此我们引入了数据存储器 MEM，写信号为 MemWrite，时钟上升沿时写入数据。

2.1.2 数据通路的设计

依照图 2.1 所示的整体架构，我们将各个模块连接起来即可。需要另外设计数据通路的模块为 Imm Gen，这是用于从 IR 中解析出立即数的部件，我们根据硬布线控制器输出的控制信号可以得到不同类型指令的信号：B_signal=beq + bne + blt，S_signal=S_Type，J_signal=JAL，显然如果这三个信号都为 0 则对应的是 I 型指令，根据 RISC-V 的指令架构，我们可以利用分线器得到这四类指令对应的立即数，然后通过多路选择器进行选择即可，多路选择器的输入端从 0-3 分别对应 I、S、B、J 型指令的立即数，选择信号生成的真值表如表 2.2 所示。

表 2.2 Imm_Sel 真值表

B_signal	S_signal	J_signal	Imm_Sel	
			Imm_Sel[1]	Imm_Sel[0]
0	0	0	0	0
0	0	1	1	1
0	1	0	0	1
1	0	0	1	0

由于指令的差异，数据通路的构造还需在图 2.1 的基础上完善 PC 寄存器输入端的多路选择，Branch=B_signal|J_signal，同时再添加一个多路选择器，选择端信号为 jalr，而 jalr 的跳转地址为 ALU 的运算结果。ecall 指令是一条非常重要的指令，它控制着单周期 CPU 的数据显示与停机，需要用到 a0 和 a7 的值，因此我们在 Regifile 的输入端添加两个多路选择器，当 ecall 信号产生时，我们输入的地址不再是从 IR 中分线出的地址，而是 a0 和 a7 的地址。将 a7 与 34 作比较，如果不相等则产生停机信号并利用 D 触发器锁存，反之则将 a0 的值输出至数码管。

2.1.3 控制器的设计

单周期 CPU 用到的是硬布线控制器，我们利用 Excel 表格对各个指令需要的信号进行填写，生成逻辑表达式所对应的电路即可。对于 R 型指令，涉及到寄存器堆的写回，只会用到 RegWrite 信号；对于 I 型指令，涉及到寄存器堆的写回和立即数的使用，要用到 RegWrite 和 ALU_Src 信号，特别的，lw 指令因为要将数据存储器 MEM 的数据存储至寄存器堆 RF，因此还需要 MemtoReg 信号而 jalr 还需要一个特征信号控制跳转；对于 S 型指令，因为要将数据写入 MEM 并且要用到立即数，所以需要 MemWrite、ALU_Src 和 S_Type 信号；对于 B 型指令和 J 型指令，它的跳转地址有另外的计算通路，我们只需要为相应指令添加相应的一个特征信号即可，J 型指令涉及的链接过程还需要一个 RegWrite 信号；同时我们再为必要的差异化指令添加信号，即可得到如图 2.2 所示的控制信号填表。

#	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	sltiu	sb	blt	R1Used	R2Used
1	add	0	0	c	5				1										1	1
2	sub	32	0	c	6				1										1	1
3	and	0	7	c	7				1										1	1
4	or	0	6	c	8				1										1	1
5	slt	0	2	c	11				1										1	1
6	sltu	0	3	c	12				1										1	1
7	addi		0	4	5			1	1										1	
8	andi		7	4	7			1	1										1	
9	ori		6	4	8			1	1										1	
10	xori		4	4	9			1	1										1	
11	slti		2	4	11			1	1										1	
12	slli	0	1	4	0			1	1										1	
13	srli	0	5	4	2			1	1										1	
14	srai	32	5	4	1			1	1										1	
15	lw		2	0	5	1		1	1										1	
16	sw		2	8	5		1	1			1								1	1
17	ecall		0	1c						1									1	1
18	beq		0	18								1							1	1
19	bne		1	18									1						1	1
20	jal			1b					1					1						
21	jalr			19	5			1	1						1				1	
22	CSRRSI		6	1c																
23	CSRRCI		7	1c																
24	URET		0	1c						1										
25	srl	0	5	c	2				1										1	1
26	sltiu		3	4	12			1	1							1			1	
27	sb		0	8	5		1	1			1						1		1	1
28	blt		4	18	11													1	1	1

图 2.2 硬布线控制器控制信号设计

2.2 中断机制设计

2.2.1 总体设计

本次课程设计中我们要在单周期 CPU 的基础上添加单级中断与多级中断，共有 3 个中断源，其优先级为 1<2<3，单级中断的开关中断由硬件实现，多级中断的开关中断由硬件结合软件实现。两种中断的构成模块相同：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，中断的响应机制如图 2.3 所示。

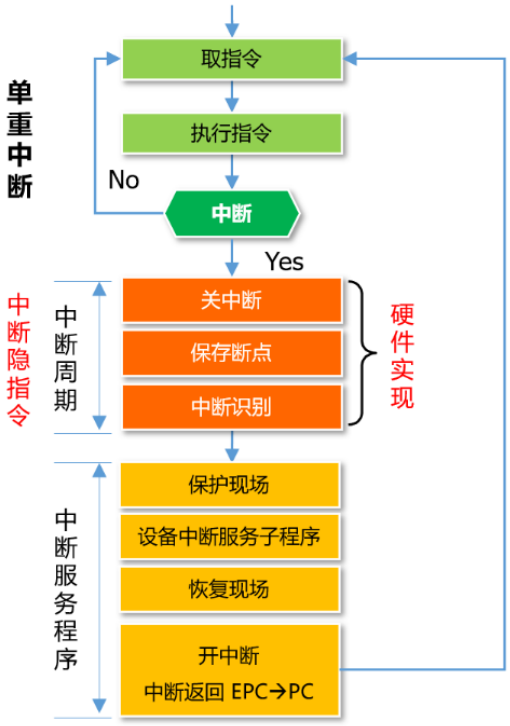

图 2.3 中断处理机制

2.2.2 单级中断设计

中断控制器：在中断源发出中断信号后会在下一个上升沿将中断信号存入中断寄存器中，三个中断的这部分实现相同，三个中断寄存器的输出端的或运算即为中断信号，表示当前有中断发生，而中断号则通过将三个中断寄存器的输出端连到优先编码器上，优先编码器输出中断优先级最高的那个中断的中断号。

中断使能信号产生：利用中断使能寄存器存储使能信号 IE，即利用 D 触发器的当前触发状态的非值作为中断使能信号，D 触发器使用上升沿触发，其输入端为 URET

信号的非值和中断信号的与运算，在产生中断信号的下一个上升沿时 IE 会被置 0，而 URET 信号产生的那个上升沿 IE 会被重新置 1，由此实现了开关中断。

中断请求信号产生：中断请求信号即为中断信号与中断使能信号的与运算，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，显然中断使能信号要用在选择正常地址和中断入口地址的多路选择器的选择端处。

中断清零信号产生：三个中断的清零信号是相互独立的，我们利用中断号用译码器输出当前正在运行的是哪个中断并且将该信号与 URET 信号进行与运算来产生相应中断的清零信号，注意这里的中断号需要用寄存器锁存，因为在某个中断未返回之前可能产生优先级更高的中断导致中断控制器输出的中断号发生改变。

中断入口逻辑：用 rars 打开单级中断测试程序，查看三个中断的地址分别为：0x000030ac、0x00003150、0x000031f4，利用中断号进行多路选择即可。

保护现场模块：这是中断实现中最为重要的模块，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，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。

2.2.3 多级中断设计

接着我们要实现多级中断，多级中断的中断控制器、中断清零信号、中断请求信号的产生和中断入口逻辑的实现与单级中断大同小异，不在此进行阐述，具体见实现部分，但是多级中断在产生中断使能信号以及保护现场上有着较大的差别。

中断使能信号产生：因为涉及到软件的开关中断，所以我们需要产生 csrrsi 和 csrrci 信号，此处我们可以不修改硬布线控制器，可以直接利用比较器得到这两个信号，URET 控制异步开中断，所以要连到 D 触发器的置 1 端，csrrsi 是同步开中断，所以连到 D 触发器的使能端，产生中断请求后会进入保护现场的阶段，此处的关中断并没有 csrrci 指令，因此我们需要利用另一个 D 触发器实现关中断。

当前中断号产生：产生高优先级信号只需要比较中断控制器输出的中断号和当前运行中断号的大小即可产生。当产生更高优先级信号或者 URET 信号并且处于中断使能阶段时，则会对当前各个中断寄存器的状态进行更新，按照优先级决定当前中断号。

保护现场：实现思路与单级中断相同，但是需要 3 个存储地址的寄存器，根据中

断控制器输出的中断号决定现场存储在哪个寄存器中，根据当前中断号决定返回哪个寄存器中的现场。

多级中断中开关中断有两条指令进行控制，分别是 csrrsi 控制开中断，csrrci 控制关中断，在进入中断服务程序后硬件可以实现关中断，所以我们不需要 csrrci 指令，但是在保护完现场后需要 csrrsi 指令进行开中断，同时中断服务程序执行结束后要还原现场，此时需要 csrrci 指令进行关中断，最后的中断返回与开中断的工作由 URET 指令实现。

2.3 流水 CPU 设计

2.3.1 总体设计

我们本次设计的是五段流水 CPU，将 RISC-V 单周期 CPU 的每条指令的执行过程划分为取指 IF、译码 ID、执行 EX、访存 MEM、写回 WB 五个阶段。其中 IF 段包括程序计数器 PC、指令存储器以及计算下条指令地址逻辑；ID 段包括操作控制器、取操作数逻辑、立即数符号扩展模块；EX 段主要包括算术逻辑运算单元 ALU、分支地址计算模块；MEM 段主要包括数据存储器读写模块；WB 段主要包括寄存器写入控制模块。需要设计 4 个流水寄存器用于实现数据和信号在这五个阶段之间的传递。五段流水线总体设计框架如图 2.4 所示。

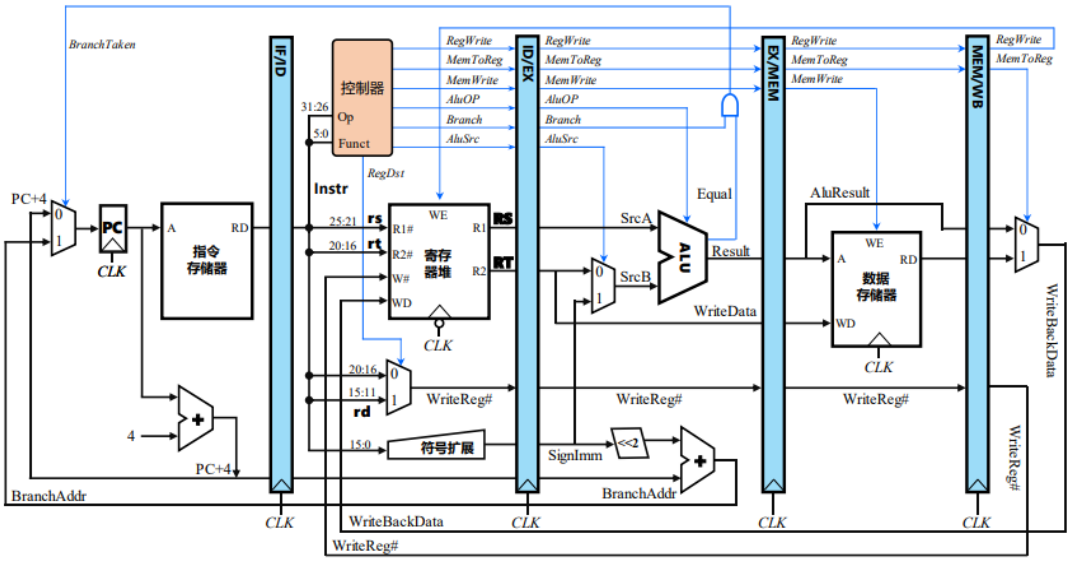

图 2.4 五段流水线总体结构图

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 分别存储相邻两个阶段之间需要传送的数据与控制信号，流水寄存器上升沿触发存储，并且需要有复位端（高电平有效）和使能端（低电平有效），便于支持后续气泡流水线和重定向流水线的设计，各个流水寄存器的存储数据与信号如表 2.3 所示。

表 2.3 流水寄存器设计

流水寄存器	存储的数据
IF/ID	IR、PC、PC+4
ID/EX	IR、PC、PC+4、R1、R2、Imm_I、Imm_S、Imm_B、Imm_J、RegWrite、AluSrcB、AluOP、MemWrite、MemToReg、BNE、BEQ、JAL、JALR、URET、ecall、Sltiu、Sb、BLT、S_Type
EX/MEM	IR、PC、PC+4、R1、R2、AluData、RegWrite、MemWrite、MemToReg、JAL、JALR、ecall、Sb
MEM/WB	IR、PC、PC+4、R1、R2、AluData、MemData、RegWrite、MemToReg、JAL、JALR、ecall

2.3.3 理想流水线设计

根据表 2.3 的流水寄存器设计，我们可以根据各阶段所需的信号很容易得将单周期 CPU 划分为五个阶段，在取指 IF 阶段 PC 寄存器输出指令地址并由指令寄存器输出 IR；在译码 ID 阶段，得到硬布线控制器得输出、寄存器堆的输出以及 I、S、B、J 型指令的立即数；在执行 EX 阶段，为了后续气泡流水线和重定向流水线在 EX 段执行分支的设计，我们选择在 EX 阶段将分支信号传回 IF 段，同时得到 ALU 的运算结果；在访存 MEM 阶段，实现存储并取出存储器中的数据；在写回 WE 阶段，我们需要保证 RegWrite 是从前面传过来的并和 Din 同步传回 ID 阶段的寄存器堆 RF，由此我们通过拆分单周期 CPU 设计好了理想流水线。

显然理想流水线并不能支持分支指令，因为会存在误取指令，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，这些问题在理想流水线得测试程序中已经被规避掉了，并且我们在后面设计的气泡流水线和重定向流水线就是为了解决这些问题的，因此在设计理想流水线的过程中我们需要选择分支指令的执行阶段。

2.4 气泡式流水线设计

2.4.1 设计原理

我们设计的理想流水线存在冲突，冲突分为三种：结构冲突、控制冲突、数据冲突。RISC-V 单周期 CPU 的哈佛结构和运算器的分离已经解决了结构冲突；而分支指令在 EX 段执行时如果需要发生跳转，那么进入 IF 段和 ID 段的两条指令都属于误取的指令，只需要将分支跳转信号连接到 IF/ID 和 ID/EX 两个流水寄存器的复位端上，在产生跳转时则会清空不该执行的指令，由此解决控制冲突；数据冲突有三种 RAW、WAR、WAW，其中 WAR 和 WAW 在一般流水线中不会造成影响，我们只需要重点关注 RAW，这就涉及到 ID 段分别与 EX、MEM、WB 段之间的数据冲突，我们可以将寄存器堆 RF 的写入改为下降沿触发，这样就解决了 ID 段与 WB 段的冲突，先写入后读出，我们只需再解决 ID 段与 EX、MEM 段之间的冲突即可。

对于不同的指令我们需要得到它们对寄存器堆中的 R1、R2 两个输出端的使用情况，通过 Excel 表格生成 R1Used 和 R2Used 信号，填表情况如图 2.2 所示。接着我们要生成数据冲突信号 DataHazzard，只要 ID 段的 R1Adr 和 R2Adr 与 EX 或 MEM 段的 rd 是相等的并且 EX 或 MEM 段有 RegWrite 信号，就说明在写回数据前会读出错误的数据，发生 RAW 冲突，我们需要在 EX 段插入气泡并暂停 IF、ID 段的数据传输。得到 DataHazzard 信号以及插入气泡的逻辑表达式如表 2.4 所示。

表 2.4 气泡流水线信号设计

信号	逻辑表达式
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)
BranchTaken	Branch + JAL + JALR
IF/ID.RST	BranchTaken + RST
IF/ID.en	DataHazzard + (halt&~Go)
ID/EX.RST	DataHazzard + BranchTaken + RST
PC.en	~DataHazzard & ~halt

2.5 重定向流水线设计

2.5.1 设计原理

前面设计的气泡流水线虽然解决了分支跳转与 RAW 冲突，但是插入过多的气泡显然是会导致流水线的效率下降的，此处实现的重定向流水线可以优化掉解决 RAW 冲突时插入的气泡，进一步得提高流水线性能。

为了在发生 RAW 冲突时不插入气泡，我们在冲突时不暂停指令的传输，而是在用到 R1 和 R2 的地方通过多路选择器将正确的 R1，R2 值传到后续的模块中使得程序正确的执行，但是 Load-Use 相关并不支持重定向操作，因为这样做的后果是 EX 段的关键路径延迟变成了 MEM 段访存延迟加 EX 段运算器运算延迟，而流水线频率取决于流水线中最慢的功能段，这会使得流水线频率大大降低。

因此我们在处理 ID 段与 EX 段的数据相关时，如果 EX 段的指令为 lw 并且与 ID 段的指令之间存在数据冲突，则需要产生一个 LoadUse 信号，该信号会暂停 IF 和 ID 段并在 EX 段插入一个气泡。

回到重定向的操作，我们需要对寄存器堆输出的 R1 和 R2 进行重定向，即多路选择，如果产生数据冲突，则将 EX 或 MEM 段的正确数据传入 ALU，反之则正常得传输 R1 或 R2，我们还需要生成 R1_F 和 R2_F 两个信号用于多路选择，这两个信号是在 ID 段产生的，我们需要利用流水寄存器 ID/EX 将其传到 EX 段进行多路选择。重定向所需的信号的逻辑表达式如表 2.5 所示。

表 2.5 重定向流水线信号设计

信号	逻辑表达式
LoadUse	R1Used & (R1Adr≠0) & EX.MemToReg & (R1Adr==EX.rd) + R2Used & (R2Adr≠0) & EX.MemToReg & (R2Adr==EX.rd)
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
IF/ID.RST	BranchTaken + RST
IF/ID.en	LoadUse + (halt&~Go)

信号	逻辑表达式
ID/EX.RST	LoadUse + BranchTaken + RST
PC.en	~LoadUse & ~halt

2.6 动态分支预测机制设计

2.6.1 设计原理

动态分支预测机制是对重定向流水线的进一步优化，因为重定向流水线仅仅是优化了气泡流水线在解决数据冲突时插入过多气泡的问题，而处理分支指令时仍要插入两个气泡，会大大降低流水线的性能，分支预测即在 IF 段对分支指令的下一条指令地址进行预测，尽可能得减少误取指令以减少气泡数。

我们采用 8 路全相联 cache 存储分支指令的跳转地址，cache 行由有效位 Valid、分支指令 PC、跳转地址 PC、分支预测历史位、淘汰计数位组成，其中 Valid 位说明当前 cache 行是否存储有数据；分支指令 PC 即为关键字 Tag 用于进行查找；跳转地址 PC 即为分支指令在跳转时转到的地址；分支预测历史位是一个由有限状态机控制的 2 位数据，用于决定是否进行跳转；淘汰计数位根据 LRU 的规则在 cache 满后对 cache 行进行淘汰替换，我们将这个 8 路全相联 cache 称为 BTB 表。

分支预测历史位的状态转移如图 2.5 所示，科学研究表明，双预测位可在较低的成本下实现很高的预测准确率，每当 cache 中存储的分支指令执行到 EX 阶段就会对分支预测历史位进行一次更新。分支预测历史位的初始值不同也会导致动态分支预测的效果不同，实际设计时应适当动态调整预测位初始值。

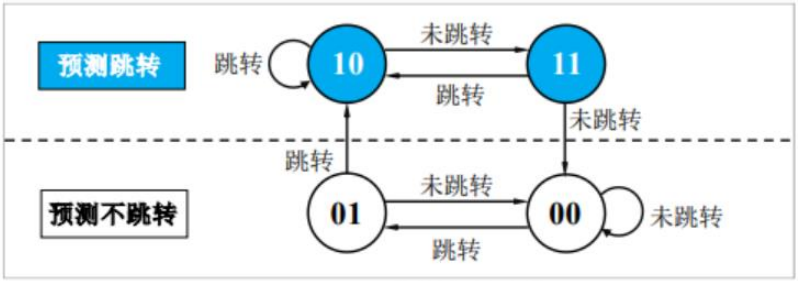

图 2.5 分支预测历史位状态转移图

既然涉及到 cache 的淘汰替换以及分支预测历史位的更新，所以我们需要在 BTB 表中生成 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，注意清空信号采用 D 触发器存储以避免毛刺，在 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。BTB 表的输入引脚为：EX 段分支指令信号 EX.Branch、EX 段分支指令的跳转地址 EX.BranchAddr、EX.PC、IF.PC，输出引脚为：根据分支预测历史位的最高位决定的跳转信号 PredictJump、预测的跳转地址 JumpAddr。

进行动态分支预测的数据通路设计时，需要修改 IF/ID 和 ID/EX 两个流水寄存器，为其添加一个预测跳转信号，将其传到 EX 段与 BranchTaken 信号进行比较，如果相同则说明我们动态分支预测正确，反之则产生 PredictError 信号用于清空 IF、ID 段的误取指令，通过原来插入气泡的方式重新执行分支指令。同时在 PC 寄存器的输入端，我们将原来数据通路处的 IF.PC+4 改为 EX.PC+4 这是为了应对动态分支预测错误时的情况，然后在原有的基础上添加 2 个新的多路选择器，一个负责在 BTB 输出的跳转地址和 IF.PC+4 之间进行多路选择且选择信号为 BTB 输出的 PredictJump，另一个负责在预测的下一条指令地址和原来设计的下一条指令地址之间进行多路选择且选择信号为 EX 段传过来的 PredictError，动态分支预测的与重定向流水线不同的几个信号的逻辑表达式如表 2.6 动态分支预测信号所示。

表 2. 6 动态分支预测信号设计

信号	逻辑表达式
PredictError	EX.PredictJump==BranchTaken
EX.Branch	EX.B_signal + EX.J_signal + EX.JALR
EX.BranchAddr	(EX.B_signal + EX.J_signal) ? EX.B/JAddr : (EX.JALR) ? EX.AluData : IF.PC
IF/ID.RST	PredictError + RST
ID/EX.RST	LoadUse + PredictError + RST

2.7 流水中断设计

2.7.1 设计原理

流水中断的设计与单周期 CPU 的中断有着较大的不同，因为它涉及到流水线的 5 个阶段的分别处理，我们选择在 EX 段响应中断，那么要做的工作有：MEM 和 WB 的两条指令要执行完，IF 和 ID 的两条指令要清空，EX 段的指令不执行，保存断点并重启流水线，将中断服务程序的地址传入 PC 执行中断服务程序，URET 指令执行过

后返回断点处继续执行指令。在产生中断信号、中断使能信号、中断请求信号、中断入口逻辑和中断清零信号处可以完全采用单周期 CPU 在单级中断中的设计，但是在保护现场阶段我们需要仔细思考要存储的断点到底是什么。

首先解决清空 IF 和 ID 段指令和执行 MEM 和 WB 段指令的问题，我们记产生中断瞬间的 IF 和 ID 段的两条指令为 IR1 和 IR2, MEM 和 WB 段的两条指令分别为 IR4 和 IR5，在产生中断后的下一个上升沿才会产生中断请求信号，此时 IR1 和 IR2 分别传到了 ID 和 EX 段，而在 IF 段新进入了一个指令 IR0，而在再下一个上升沿时才会对流水寄存器进行清空，此时 IR1 和 IR2 分别传到了 EX 和 MEM 段并且 IR4 和 IR5 已经执行完成，同时 IR0 进入了 ID 段，IR3 进入 WB 段，所以我们要利用中断请求信号清空全部流水寄存器的值。

接着我们要解决保护现场的问题，我们仍然从中断响应一步步分析，我们将在产生中断瞬间 EX 段的指令即为 IR3，此时并不会产生中断请求信号，而是在下一个上升沿时产生中断请求信号, 此时 IR3 传到了 MEM 段, 显然断点应该设置为 MEM.PC，但是由于重定向流水线中分支指令和 LoadUse 仍会插入气泡，所以此时 MEM.PC 可能是无效的，因此我们进行分类讨论：①产生中断时 EX 段指令不是气泡：显然，断点为 MEM.PC；②产生中断时 EX 段指令为分支指令的第一个气泡：产生中断时 IF 段 PC 是分支指令的跳转地址，下一个上升沿产生中断请求信号，因此存储的断点为 ID.PC；③产生中断时 EX 段指令为分支指令的第二个气泡：产生中断时 ID 段的 PC 是分支指令的跳转地址, 下一个上升沿产生中断请求信号, 因此存储的断点为 EX.PC；④产生中断时 EX 段指令为 LoadUse 插入的气泡：此时 ID 段的指令是正常执行的，下一个上升沿产生中断请求信号，因此存储的断点为 EX.PC。综上我们得到了保护的现场的具体情况，但是为了判断发生的是上述哪种情况，我们需要用寄存器暂存 EX 段产生的 BranchTaken 和 LoadUse 信号, BranchTaken 信号需要暂存 3 个周期, LoadUse 信号需要暂存 2 个周期，由此产生多路选择信号，我们选择在下降沿保护现场。

我们仍需注意的是单级中断在当前中断进程未结束之前可能产生新的中断进行等待，所以我们还要考虑另一个中断的保护现场，我们选择用 EX 段的 URET 信号开中断，IF 段的 URET 信号返回现场，那么当上升沿 URET 指令执行到 MEM 段时，如果仍存在中断信号则会产生中断请求信号，在下降沿保护现场，那么我们需要将 EX.PC 作为断点，EX.URET 信号需要暂存 1 个周期，由此我们完善保护现场的多路选择，这样可以保证不会遗漏任何一条指令的执行。

2.8 基于单级中断的 2048 游戏设计

2.8.1 设计原理

我们在支持单级中断的 RISC-V 单周期 CPU 的基础上进行修改，并添加相关模块以实现支持键盘操作的 2048 游戏。首先将整个游戏设计划分为 4 个模块：CPU 模块、同步存储模块、随机数生成模块、显示模块，各个模块的设计原理如下：

1. CPU 模块

在之前我们所设计的支持单级中断的单周期 CPU 中包含 3 个中断源，而 2048 的游戏需要响应上下左右 4 个按键的按下，我们希望将这 4 个按键的触发看成 4 个不同的中断进行处理，所以我们改为 4 个中断源，修改中断控制器并根据编写的代码修改中断入口逻辑即可。

2. 同步存储模块

2048 游戏包含 16 个单元格，我们利用 16 个 32×32 的 LED 实时显示 16 个单元格中存储的数据，我们希望在软件中修改这 16 个单元格的值，并且用 16 个变量记录这些值，因此就需要用到 lw 和 sw 指令在 MEM 中进行存储与修改，但是 MEM 仅有一个数据输出端口，不能同时得到 16 个单元格的数据，所以我们设计一个和 MEM 同步进行存储与修改的 RAM，它具有 16 个输出端口并且存储的 16 个变量与 MEM 中存储的 16 个变量值恒等。

3. 随机数生成模块

2048 游戏涉及到随机数的生成，首先每当玩家进行一次操作之后会选中一个随机的空单元格，然后会以不同的概率生成 2 或 4，后者的实现比较简单，通过随机数生成器和比较器即可实现，但是随机选中一个空位置并不容易实现，我们选择利用硬件来实现以简化程序的编写。首先将 16 个值两两一组分为 8 组，如果该组中的两个值都为空值或都非空则随机选择一个单元格编号，反之则选择那个空单元格编号，由此我们得到了 8 个编号；同理将 8 个编号两两一组进行同样的选择操作，直到最后得到一个随机的空位置编号。

4. 显示模块

2048 涉及到的数字为 2、4、8、16、…、2048，我们将这 11 个数字转成 32×32 的像素数字，并将像素数字划分为 32 列，每一列根据像素点的显示（有像素点的格为 1 反之为 0）转成 16 进制数，由此我们将每个数字转成了 32 个 16 进制数，将第 j 个数字的第 i 列对应的 16 进制数存储到第 i 个 ROM 中的第 j 行，即可完成了 LED 译码器的设计，输入端口为该单元格对应的数字，输出端口为 32 列 LED 译码，利用 16 个 LED 译码器和 16 个 32×32 的 LED 显示器组成显示模块。

完成 4 个模块的设计后，需要进行 RISC-V 代码的编写，根据我们的实现思路，首先对棋盘进行初始化，随机选中一个单元格生成随机数，然后进入一个死循环等待中断响应，完成中断处理之后继续进入死循环等待，需要注意的我们不希望在中断处理的过程中棋盘中显示的数字就发生改变，而是等中断处理结束之后棋盘各单元格再统一进行一次更新，因此要关注的问题是：软件如何告知硬件生成一个随机数？软件如何告知硬件对棋盘进行更新？

对于上述两个问题就需要用到在前面设计中控制 LED 显示与停机的 ecall 指令，它是软件与硬件进行通信的桥梁，执行 ecall 时，a7 的取值不同，执行的操作也不同，在 2048 游戏中我们不再需要停机与 LED 的显示，那么不妨修改 ecall 指令对应的操作，此处我们将 a7=34 时执行的操作改为生成随机数，并设置 a7=44 时执行的操作为更新棋盘，当软件执行 ecall 时，会生成相应的 random 和 update 信号负责硬件操作的执行。

2.9 团队分工

组长：魏子腾

组员：

表 2.7 团队分工

成员	学号	分工任务
魏子腾		2048 的 RISC-V 代码编写，随机数生成模块设计
		同步存储模块设计
		LED 显式模块设计
		单周期 CPU 的完善与修改

3 详细设计与实现

3.1 单周期 CPU 实现

单周期 CPU 将通过 Logisim 和 Verilog 两种方式实现，将分别从主要功能模块、数据通路以及硬布线控制器三部分展示实现过程。

3.1.1 主要功能部件实现

1. 程序计数器 PC

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，输出为当前执行指令的地址。halt 为停机信号，将此控制信号通过非门取反之后连接到时钟的使能端，当需要进行停机时，halt 控制信号为 1，经过非门之后为 0，使得 PC 无法载入下一条指令地址，使整个电路停机，具体实现如图 3.1 所示。

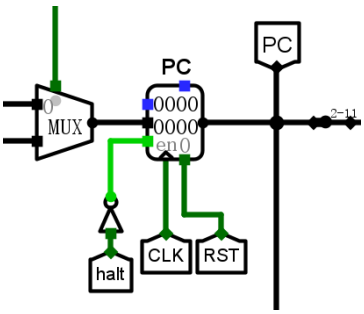

图 3.1 程序计时器 PC 实现

FPGA 实现：程序计数器 PC 的 Verilog 代码如下：设计思路与 Logisim 完全相同，并且我们可以省去 Logisim 中不必要的引脚，只需要时钟端 CLK，复位端 RST、使能端 EN、输入输出端 Din、Dout，其中各个引脚的所连接的逻辑电路将会在顶层模块中实现，同时该寄存器模块不仅要在 PC 处用到，还需要在存储 LedData 处用到，我们只需要修改引脚输入即可使得该寄存器模块负责 RISC-V 单周期 CPU 中不同的功能。

```
module register(CLK, RST, EN, Din, Dout);
    parameter WIDTH = 32;
    input CLK,RST,EN;
    input [WIDTH-1:0] Din;
```

```
    output [WIDTH-1:0] Dout;
    reg [31:0] ram;
    initial ram=0;
    always @(posedge CLK) begin
        if (RST) ram <=0;
        else if (EN) ram <= Din;
        else ram=ram;
    end
    assign Dout=ram;
endmodule
```

2. 指令存储器 IM

Logisim 实现：指令存储器 IM 是一个地址位宽为 10，数据位宽为 32 的只读存储器，我们可以直接使用 Logisim 提供的 ROM 模块，将所需运行的程序的 hex 文件载入即可，需要注意的是因为 RISC-V 的所有指令都是 32 位的即 1 个字，所以我们传入 IM 的地址应该为字地址，即取 PC 输出的地址的 2-11 位，其实现如图 3.2 所示。

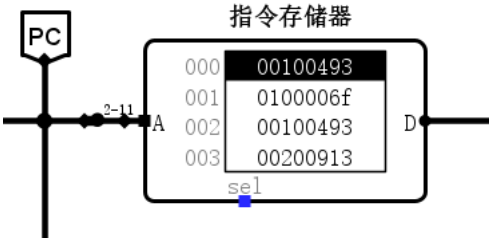

图 3.2 指令存储器 IM 实现

FPGA 实现：ROM 的 Verilog 实现如下所示，非常得简单暴力，通过 case 语句将不同的地址与不同的指令对应即可，可以通过编程生成 case 语句以减少工作量。

```
module ROM(Addr,Dout);
    input [9:0] Addr;
    output reg [31:0] Dout;
    always@(Addr)begin
        case(Addr)
            10'b0000000000 :Dout=32'h00100493;
```

```
                10'b0000000001 :Dout=32'h0100006f;
                10'b0000000010 :Dout=32'h00100493;
                …//其余地址与指令对应关系
                10'b0100011011 :Dout=32'h00008067;
                default: Dout=0;
            endcase
        end
    endmodule
```

3. 运算器

Logisim 实现：运算器在原项目中已经实现，我们可以直接使用，因此此处的实现不再过多赘述，具体功能见表 2.1 所示，需要注意的是 < 和≥信号仅在 11、12 号运算时有效，在填表与差异化指令设计中需要注意。

FPGA 实现：运算器的 Verilog 实现如下代码所示，根据运算器规格利用 case 语句实现即可，对于乘除法和移位运算可以完全采用 Verilog 中所提供的*、/、%、>>、>>>、<<、<<<运算符，不需要再单独编写模块，这也是 Verilog 的优势所在。

```
    module ALU(X,Y,ALUOP,Result1,Result2,greater_equal,lesser,equal);
        …//输入输出引脚声明
        assign equal=(X==Y);
        always@(X,Y,ALUOP) begin
            case(ALUOP)
                4'b0000:begin
                    Result1=X<<Y[4:0];
                    Result2=0;
                end
                …//其他运算
            endcase
        end
    endmodule
```

4. 寄存器堆 RF

Logisim 实现：寄存器堆在原项目中已经实现，可以直接使用，该模块用于存储 RISC-V 中的 32 个寄存器的值，在单周期 CPU 中上升沿触发，在流水线中下降沿触发，其实现如图 3.3 所示。

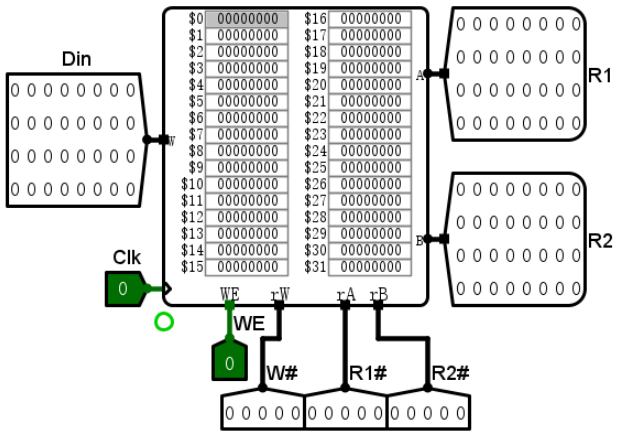

图 3.3 寄存器堆 RF 实现

FPGA 实现：Verilog 在设计时需要清楚寄存器堆的特性，首先 R1 和 R2 是随着 R1Adr 和 R2Adr 同步改变的，与时钟信号无关，所以要用到 assign 语句进行赋值，但是将 Din 写入 WAdr 对应的地址是在时钟沿完成的，所以需要 always 语句进行赋值，同时 0 号寄存器的值是不允许修改的，因此当 WAdr 为 0 时就算有写信号也不能对其进行赋值。实现代码如下所示，输入输出引脚的声明在此处省略。

```
module RegFile(Din,R1Adr,R2Adr,WAdr,WE,CLK,R1,R2);
    …//输入输出引脚声明
    reg [31:0] RAM[31:0];//存储寄存器
    integer i;
    …//寄存器值初始化为 0
    assign R1 = RAM[R1Adr];
    assign R2 = RAM[R2Adr];
    always@(posedge CLK) begin
        if (WE && WAdr!=0) RAM[WAdr] <= Din;//进行数据写入
    end
endmodule
```

5. 数据存储器 MEM

Logisim 实现：直接使用封装好的 MIPS RAM 即可，注意此处用到的也是字地址，所以地址端应该使用 ALU 运算结果的 2-11 位，其中 sel 端用于控制字、半字、字节读写，str 为写信号连接 MemWrite，实现如图 3.4 所示。

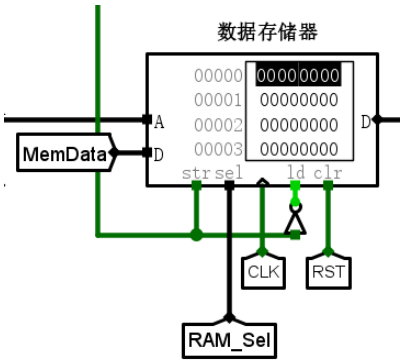

图 3.4 数据存储器 MEM 实现

FPGA 实现：数据存储器的实现与寄存器堆类似，但是我们需要设计字、半字、字节访问，根据模块的访存规则，sel=4’b1111 时进行字访问，sel=4‘b0001 时访问低 8 位，sel=4’b0010 时访问次低 8 位，依次类推，由于基础指令和差异化指令中只有 sw 和 sb，所以此处仅设计了字与字节的访问逻辑，Verilog 代码如下：

```
module MEM(Addr,Din,CLK,MemWrite,sel,Dout);
    …//输入输出引脚声明
    reg [31:0] RAM[2**20-1:0];//存储寄存器
    always@(posedge CLK) begin
        if (MemWrite) begin
            if (sel==4'b1111) RAM[Addr] <= Din;//进行数据写入
            else if (sel==4'b0001) RAM[Addr][7:0]<=Din[7:0];
            else if (sel==4'b0010) RAM[Addr][15:8]<=Din[15:8];
            else if (sel==4'b0100) RAM[Addr][23:16]<=Din[23:16];
            else if (sel==4'b1000) RAM[Addr][31:24]<=Din[31:24];
        end
    end
    assign Dout=RAM[Addr];
endmodule
```

3.1.2 数据通路的实现

本次课程设计中首先实现了 R 型指令的数据通路，接着在此基础上完善了 I、S、B、J 型指令的数据通路，在 Logisim 实现的基础上设计 Verilog 顶层模块。

首先我们实现 PC 输入端的地址选择如图 3.5 所示，从左到右有第一个多路选择器输入端分别为 PC+4 和 B 型与 J 型指令的跳转地址，选择端为 Branch + J_signal，其中 Branch=(BEQ & equal) + (BNE & ~equal) + (BLT & <)，由此得到 Data1，第二个多路选择器进行 Data1 和 AluData 之间的选择，选择端为 JALR，由此我们实现了顺序地址和 I、B、J 型指令之间的选择。

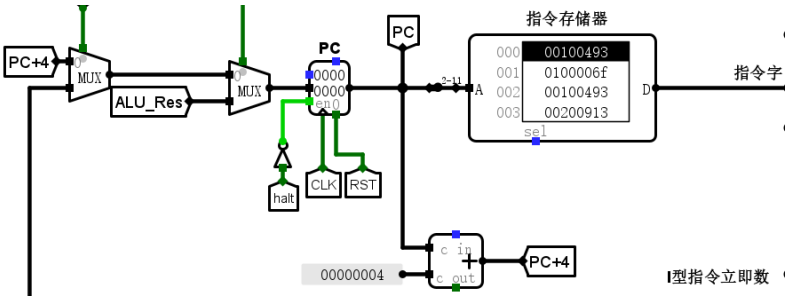

图 3.5 PC 输入端数据通路

接着我们实现 Imm_Gen 模块，根据 I、S、B、J 四种指令在 RISC-V 中立即数的位分布，通过分线器得到这四种类型的立即数，通过多路选择器输出相应的立即数，多路选择器选择端的真值表见表 2.2，由此我们得到如图 3.6 所示的立即数生成单元。

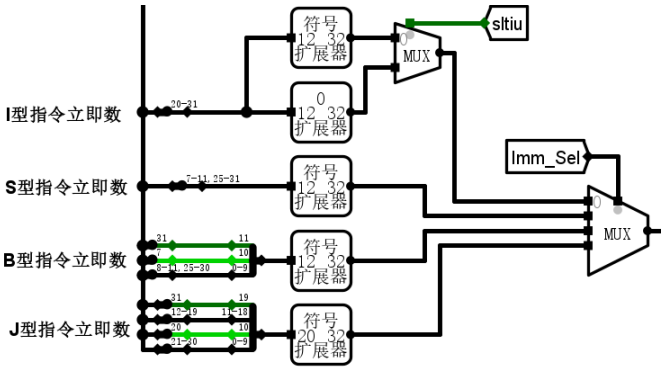

图 3.6 立即数生成器

寄存器堆处的数据通路需要特殊化处理 ecall 指令，因为 ecall 指令要用到 a0 和 a7 寄存器，它们的地址分别为 10 和 17，因此利用 ecall 信号对 R1Adr、10 和 R2Adr、17 进行多路选择，同时寄存器堆的写入端也要进行多路选择如图 3.7 所示，从左到右第一个多路选择器输入端分别为 ALU 运算结果和 MEM 读取出的数据，选择端的信号为 MemToReg，第二个多路选择器则在 JAL 或 JALR 指令时选择 PC+4 建立连接。

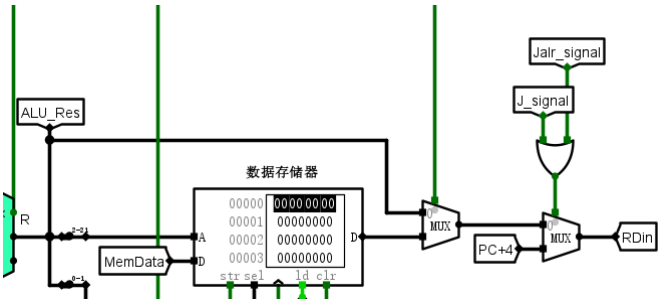

图 3.7 寄存器堆写入数据选择

处理完上述数据通路中的关键部分，我们还需要实现停机逻辑，ecall 指令在 a7 不等于 34 时进行停机，反之则将 a0 的值输出到 LED 数码管上，为了后续流水线的正常停机，我们利用 D 触发器对停机信号进行锁存如图 3.8 所示。

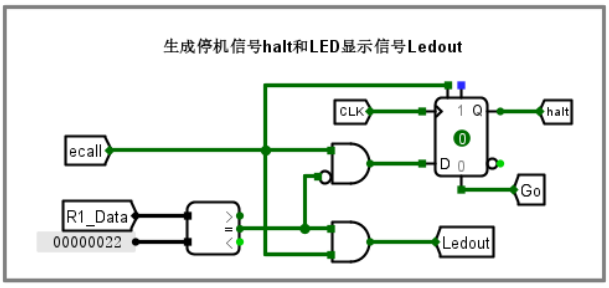

图 3.8 停机信号生成模块

综上我们将数据通路的关键部分实现后，最终得到的单周期 CPU 如图 3.9 所示。

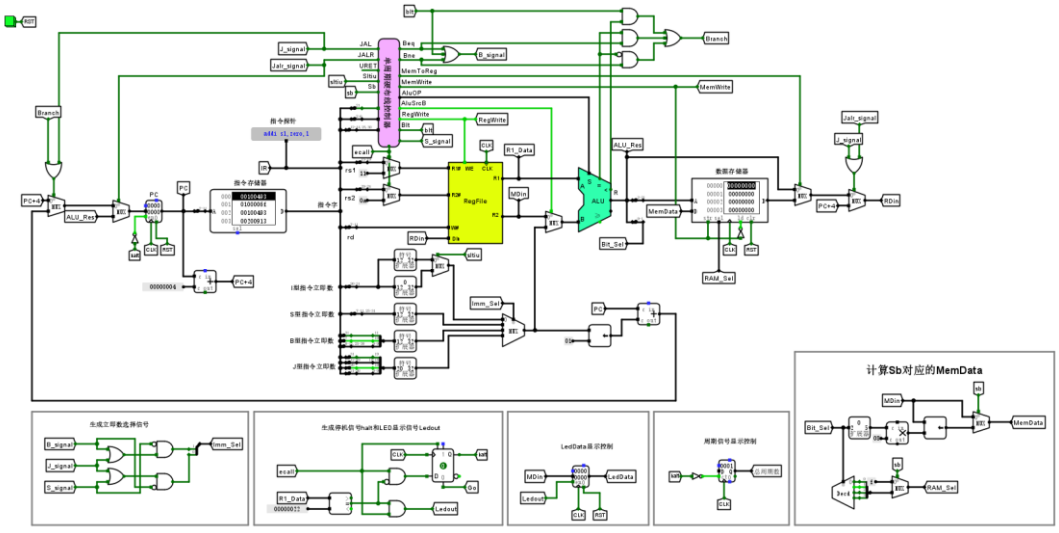

图 3.9 单周期 CPU 数据通路

我们可以参照图 3.9 利用已经实现好的模块用 Verilog 实现单周期 CPU 的顶层模块，对于模块之间的线路可以定义多个 wire 类型的变量进行连接，通过在实例化 ROM、RAM、regisiter 等模块时将各个端口连接正确 wire 变量即可实现单周期上开发板的设计，但是数据显示还需要依托 FPGADigit 模块，该模块内部通过分频器产生了一个频

率很高的时钟，该高频时钟结合计数器生成数码管的片选信号，选中不同数码管时显示不同的数字，由于人眼无法察觉高频的变化，所以走马灯式的数码管看起来像同时亮起来一样，由此我们实现了 LedData 的显示，FPGADigit 代码如下：

```
module FPGADigit(LedData,CLK,SEG,AN);
    input CLK;//开发板的时钟源
    input [31:0] LedData;//LedData 显示的数据
    output [7:0] SEG;//7 段译码信号
    output [7:0] AN;//  数码管片选信号

    wire CLK_N;//高频时钟信号
    wire [3:0] Led;//数码管显示数据
    wire [2:0] count;//计数器计数

    //实例化模块
    divider #(5000) div(.clk(CLK),.clk_N(CLK_N));//获得一个高频信号 CLK_N
    counter cot(.clk(CLK_N),.out(count));//高频计数器
    decoder3_8 decoder(.num(count),.sel(AN));//获得数码管片选信号
    display_sel sel(.num(count), .dig(LedData), .code(Led));//选择显示数字
    pattern SEG7_0(.code(Led),.patt(SEG));//七段译码器
endmodule
```

注意 FPGA 的设计需要添加额外的功能：①切换频率：通过分频器生成多个频率的时钟信号，通过多路选择器选择时钟；②切换显示数据：仍旧通过多路选择器将想要输出的数据输出到FPGADigit的数据端即可，最终实现的顶层模块如图 3.10所示。

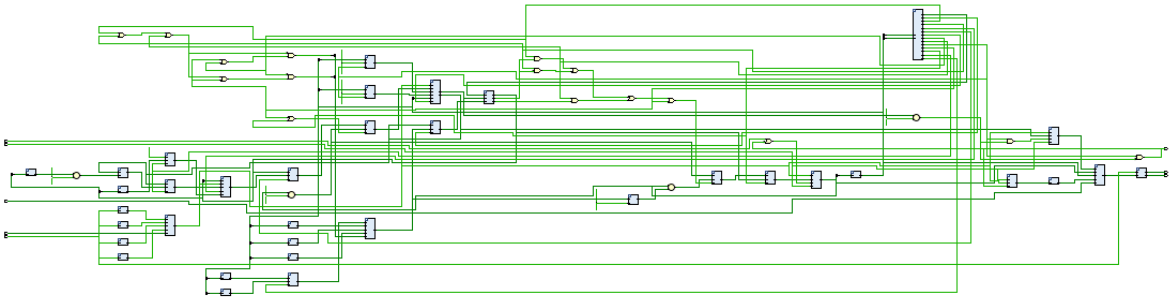

图 3.10 FPGA 原理图

3.1.3 控制器的实现

控制信号的设计见图 2.2，在 Logisim 中通过逻辑表达式直接生成电路，封装运算控制器与控制信号生成器即可得到最终的硬布线控制器模块如图 3.11 所示。

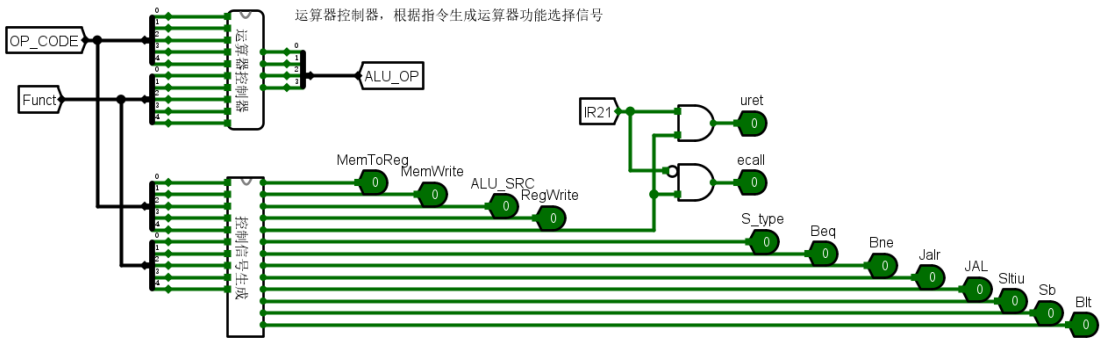

图 3. 11 硬布线控制器设计

在 FPGA 实现时我们也可以直接通过 assign 语句利用生成好的逻辑表达式进行赋值，但这样过于冗杂，我的实现方法是通过 if-else 分支语句对控制信号进行生成，这样更加有条例并且容易 debug，部分实现代码如下：

```
//运算信号生成
always@(Funct7,Funct3,OpCode)begin
    if(Funct7==7'b0000000 && Funct3==3'b000 && OpCode==5'b01100)
        ALUOP=4'b0101;//add
    else if(Funct7==7'b0100000 && Funct3==3'b000 && OpCode==5'b01100)
        ALUOP=4'b0110;//sub
        …//其他指令的运算信号生成
 end
//控制信号生成
always@(Funct7,Funct3,OpCode)begin
    if(Funct7==7'b0000000 && Funct3==3'b000 && OpCode==5'b01100)
        {MemtoReg,…,sb,blt}=13'b0001000000000;//add
    else if(Funct7==7'b0100000 && Funct3==3'b000 && OpCode==5'b01100)
        {MemtoReg,…,sb,blt}=13'b0001000000000;//sub
     …//其他指令的控制信号生成
end
```

在 Verilog 语言中 if-else 分支语句的硬件实现会转换成多路选择器，因此最后得到的控制器原理图如图 3.12 所示，通过多个 2 路选择器实现控制信号的生成。

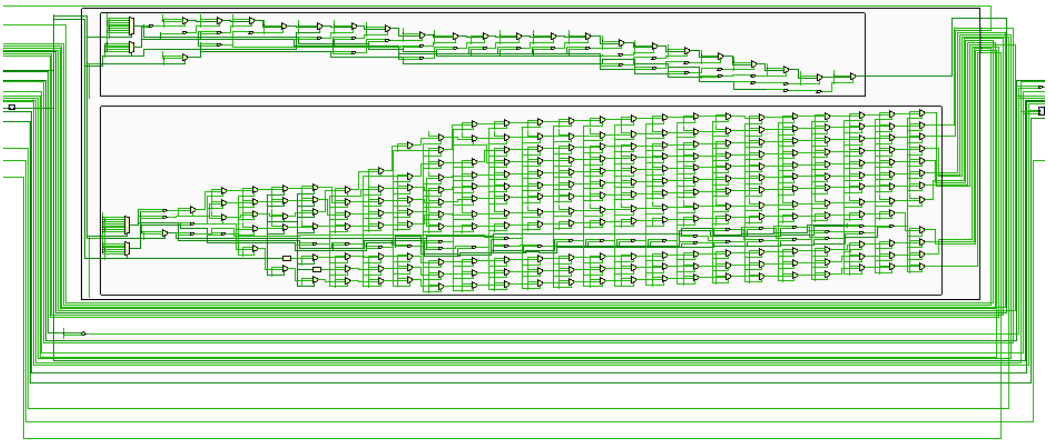

图 3.12 硬布线控制器设计（FPGA）

3.2 中断机制实现

3.2.1 单级中断实现

首先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，中断控制器实现如图 3.13 所示。

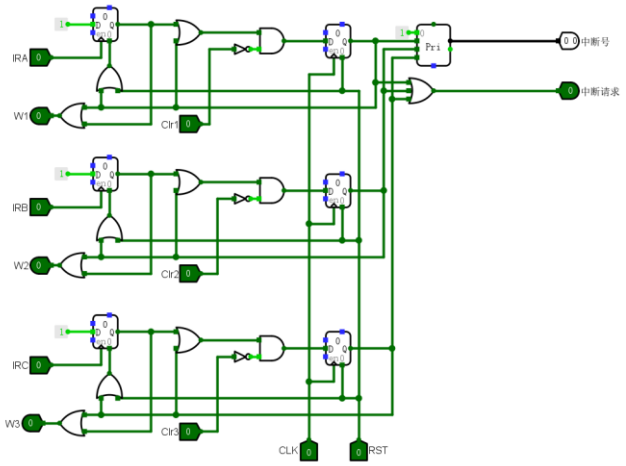

图 3.13 中断控制器实现

单级中断在响应第一个中断源后会立即关中断，直到 URET 指令执行后才会开中断，因此中断使能寄存器的实现很简单，我们只需要用 D 触发器的非值端作为使能信

号，在数据端输入~URET & 中断信号，中断请求信号即为中断信号与使能信号的与运算，最终实现如图 3.14 所示。

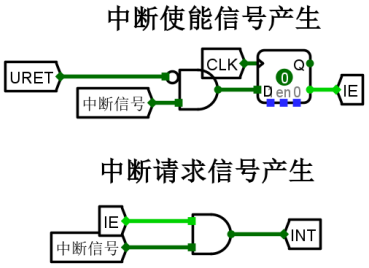

图 3.14 中断请求信号产生

中断清零信号处通过中断号选择产生哪一个中断源的清零信号，单级中断需要用寄存器暂存中断号，当有中断请求信号后才会更新中断号，在执行 URET 指令时产生清零信号。通过 RARS 打开单级中断测试程序即可查看 3 个中断服务程序的中断号，由此实现中断入口逻辑，实现如图 3.15 所示。

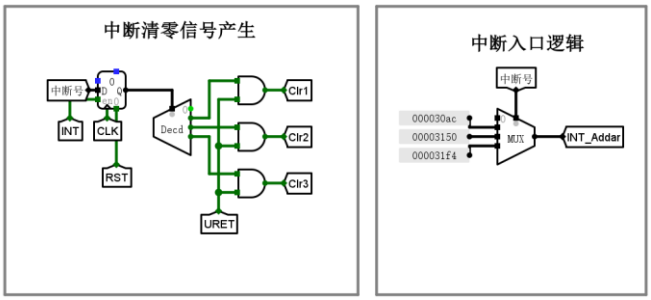

图 3.15 清零与入口逻辑

最后需要保护现场，需要保护的现场即存储的断点为 PC 寄存器输入端的值，在产生中断请求信号的下一个下降沿存储断点，在 URET 指令执行后返回断点，因此 PC 寄存器输入端的数据通路还需要进行修改，添加两个新的多路选择器，第一个多路选择器的选择端为 URET 用于在断点与正常地址间选择，第二个多路选择器的选择端为中断请求信号 INT 用于在中断入口与正常地址间选择，最终实现如图 3.16 所示。

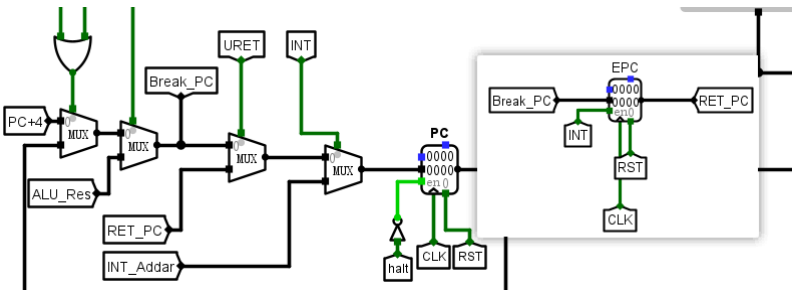

图 3.16 保护现场与数据通路设计

3.2.2 多级中断

多级中断的每个中断服务程序都会有一个保护现场和恢复现场的阶段，这两个阶段都会处于关中断状态，其中保护现场阶段需要硬件实现关中断、软件实现开中断，恢复现场的开关中断都由软件实现，首先需要生成用于开关中断的 csrrsi 和 csrrci，如图 3.17 所示。

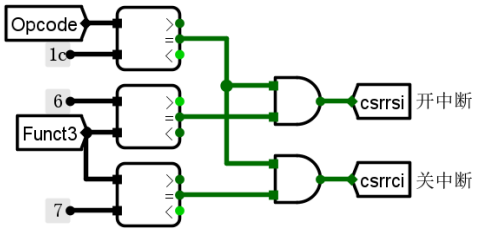

图 3.17 开关中断信号生成

我们需要一个高优先级信号进行中断嵌套，通过比较中断控制器输出的中断号和当前运行的中断号即可，在产生中断请求信号的瞬间异步关中断，其他的开关中断则通过 URET、csrrsi 和 csrrci 实现，最终得到的中断使能寄存器如图 3.18 所示。

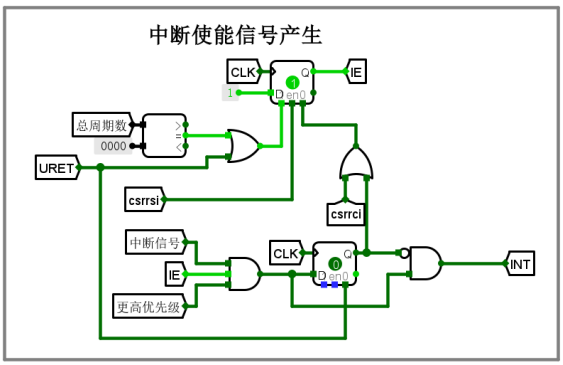

图 3.18 多级中断使能寄存器

在产生高优先级信号并且是开中断状态时会更新当前运行中断号，在执行 URET 指令后也会更新当前中断号，其存储方式如图 3.19 所示。

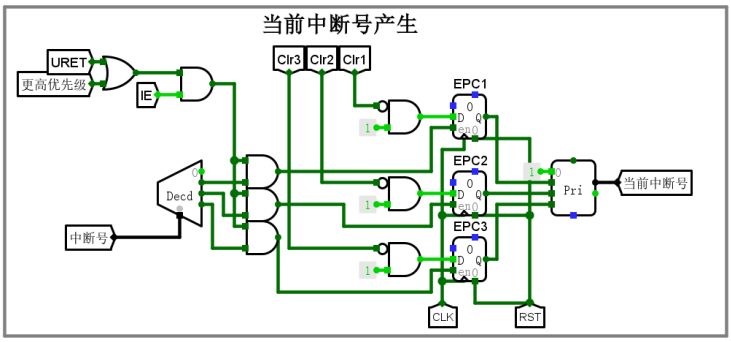

图 3.19 生成当前中断号

多级中断的中断控制器与入口逻辑和单级中断的实现方式完全相同，中断清零信号处不再需要寄存器暂存中断号，可以完全参考单级中断。保护现场部分多添加两个寄存器并根据中断号进行存储选择即可，如图 3.20 所示。

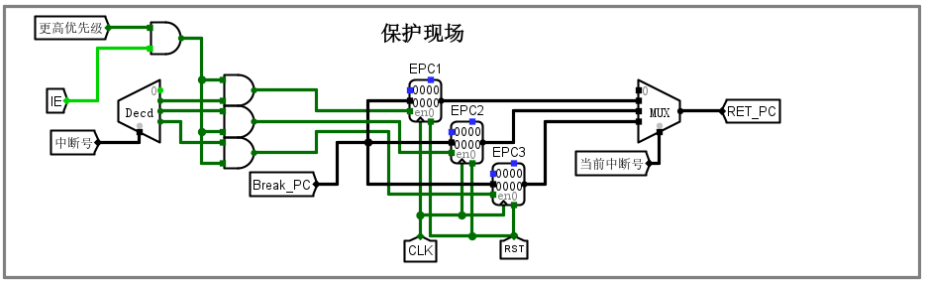

图 3.20 多级中断保护现场

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

各个流水接口部件所要传递的信号如表 2.3 流水寄存器设计所示，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 3.21 所示。

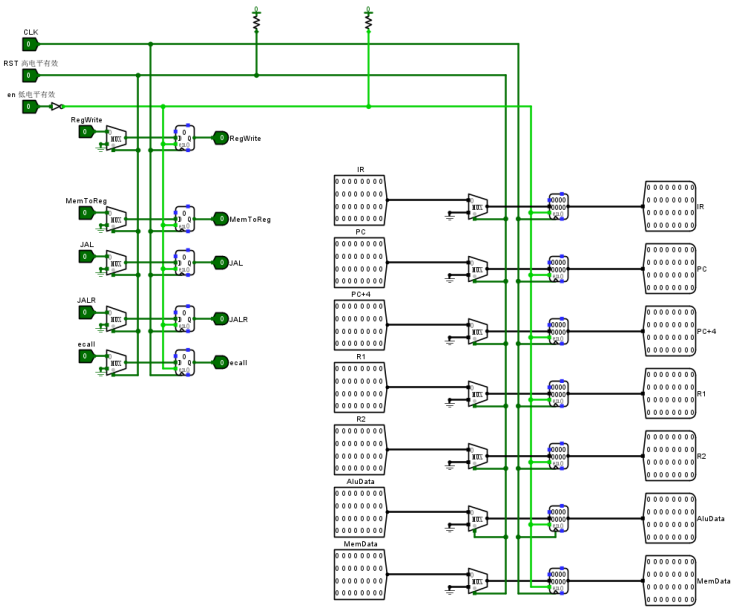

图 3.21 MEM/WB 流水寄存器设计

3.3.2 理想流水线实现

根据四个流水寄存器的设计，我们可以按照所需传输的信号将单周期 CPU 分为

5 个阶段，主要模块的分布为：①IF 段：程序计数器 PC、指令存储器 IM；②ID 段：寄存器堆 RF；③EX 段：运算器 ALU；④访存阶段：数据存储器 MEM，选择在 EX 段处理分支指令，最终得到的理想流水线如图 3.22 所示。

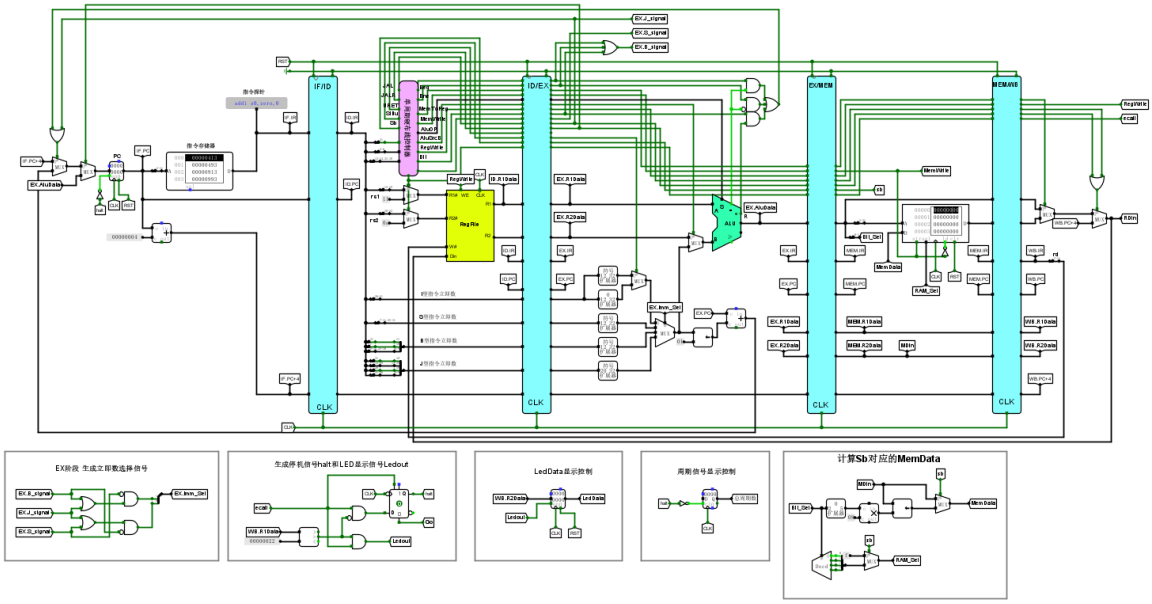

图 3. 22 理想流水线设计图

3.4 气泡式流水线实现

在理想流水线的基础上，我们需要利用分支跳转信号清空误取指令并且通过数据冲突信号在 EX 段插入气泡。R1Used 和 R2Used 信号通过 Excel 填表生成，接着需要利用这两个信号生成数据相关检测逻辑，DataHazzard 信号逻辑如表 2.4 所示，最终得到的数据相关检测电路如图 3.23 所示。

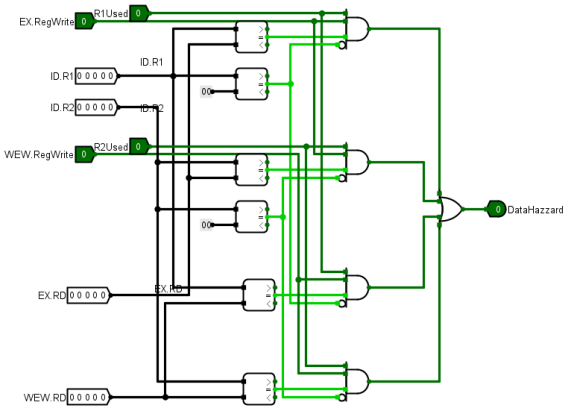

图 3. 23 数据相关检测电路

产生 DataHazzard 信号后我们只需要修改 IF/ID、ID/EX 流水寄存器的复位端 RST

与使能端 EN 即可完成气泡流水线的设计，具体设计见表 2.4，我们最终得到的气泡流水线如图 3.24 所示。

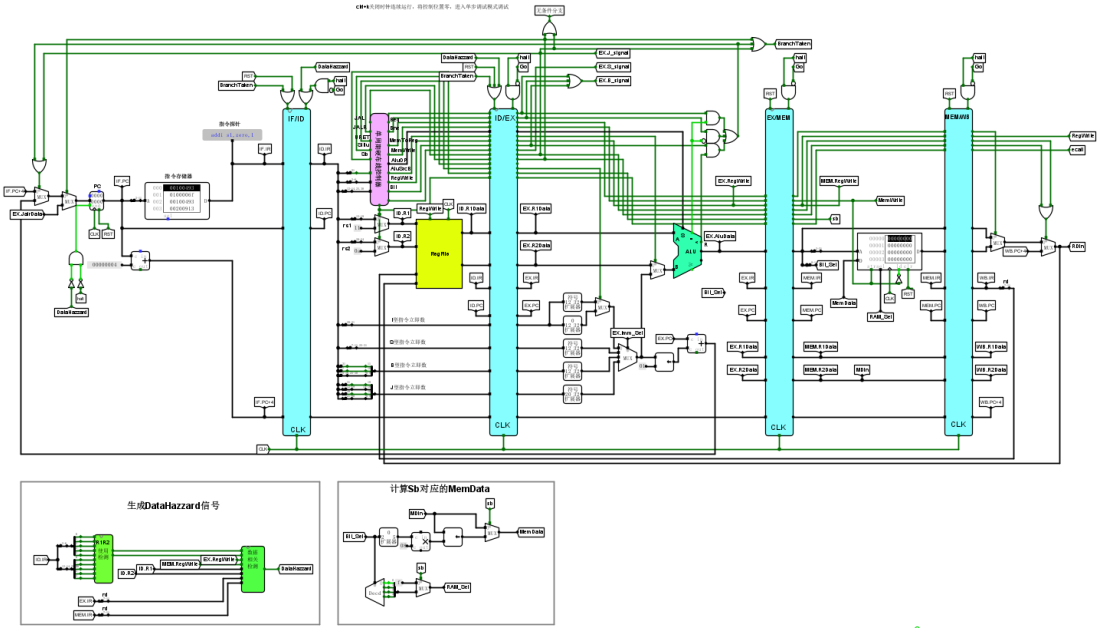

图 3. 24 气泡流水线设计图

3.5 重定向流水线实现

在气泡流水线的基础上，我们不再需要 DataHazzard 信号，而是用两个多路选择信号 R1_F 和 R2_F 的重定向取代插入气泡，以提高流水线的效率，这两个信号的逻辑表达式见表 2.5，实现电路如图 3.25 所示。

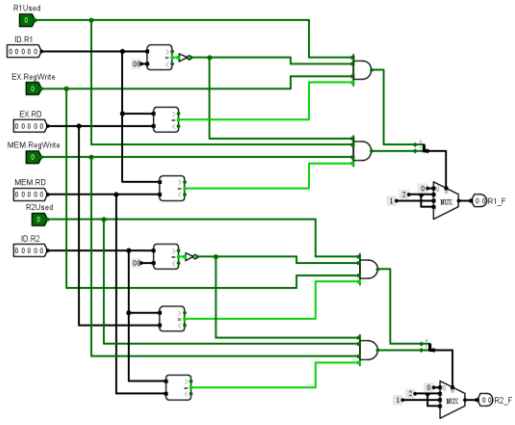

图 3. 25 R1_F、R2_F 信号生成

重定向并不能支持 LoadUse 冲突，因为这回改变 EX 段关键路径的延迟，所以我们还需要生成一个 LoadUse 信号用于特殊处理，逻辑表达式见表 2.5，最终的实现电路如图 3.26 所示。

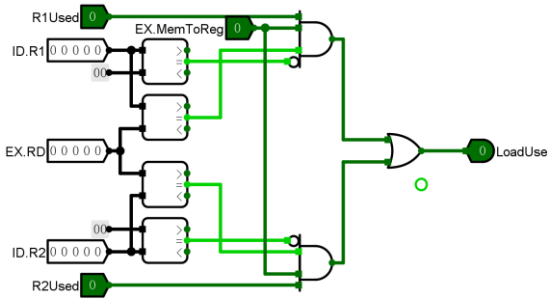

图 3.26 LoadUse 信号生成电路

同理，重定向流水线仍然要修改 IF/ID、ID/EX 的复位端与使能端，具体实现逻辑见表 2.5，最终得到的重定向流水线如图 3.27 所示。

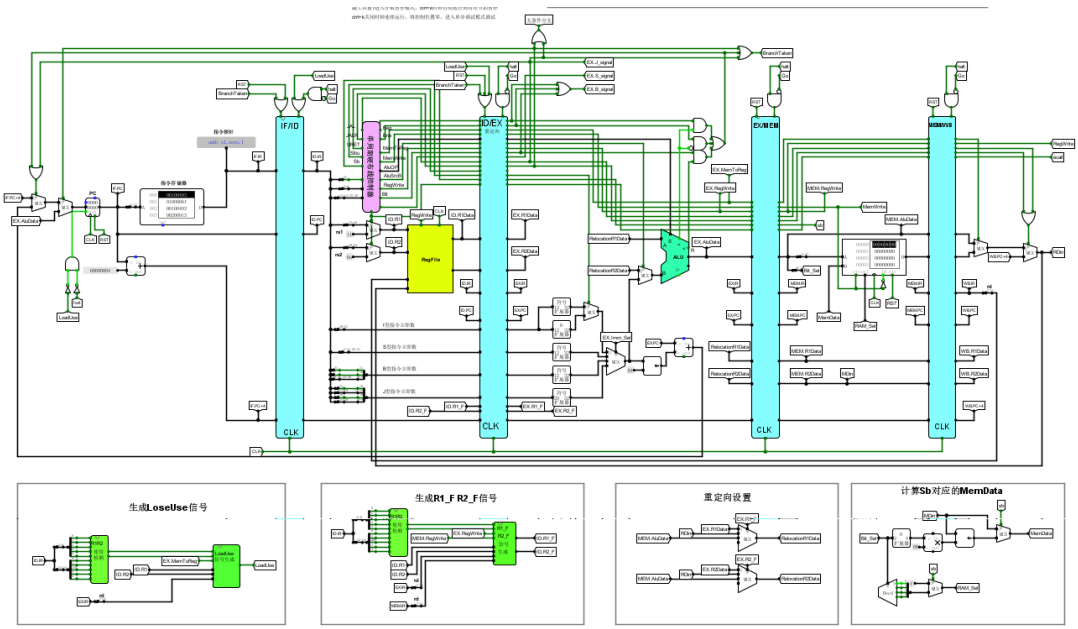

图 3.27 重定向流水线设计图

3.6 动态分支预测机制实现

首先我们需要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，状态机的转移过程如图 2.5 所示，实现电路如图 3.28 所示。

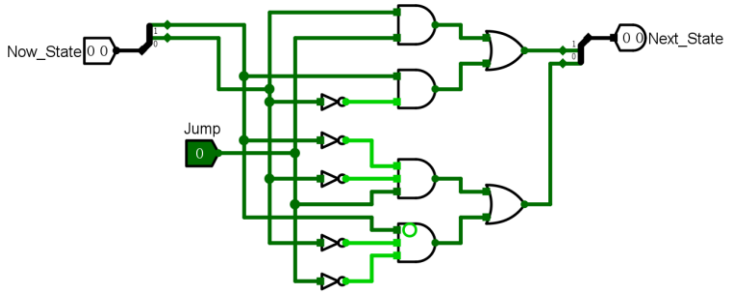

图 3.28 有限状态机

接着我们需要实现一个 8 路全相联 cache，并且支持 LRU 的淘汰替换策略，同时通过下降沿触发的 D 触发器结合多路选择器为分支预测历史位设置初值，整个 BTB 表设计如图 3.29 所示，在重定向流水线的基础上为 BTB 表添加相应的数据通路即可。

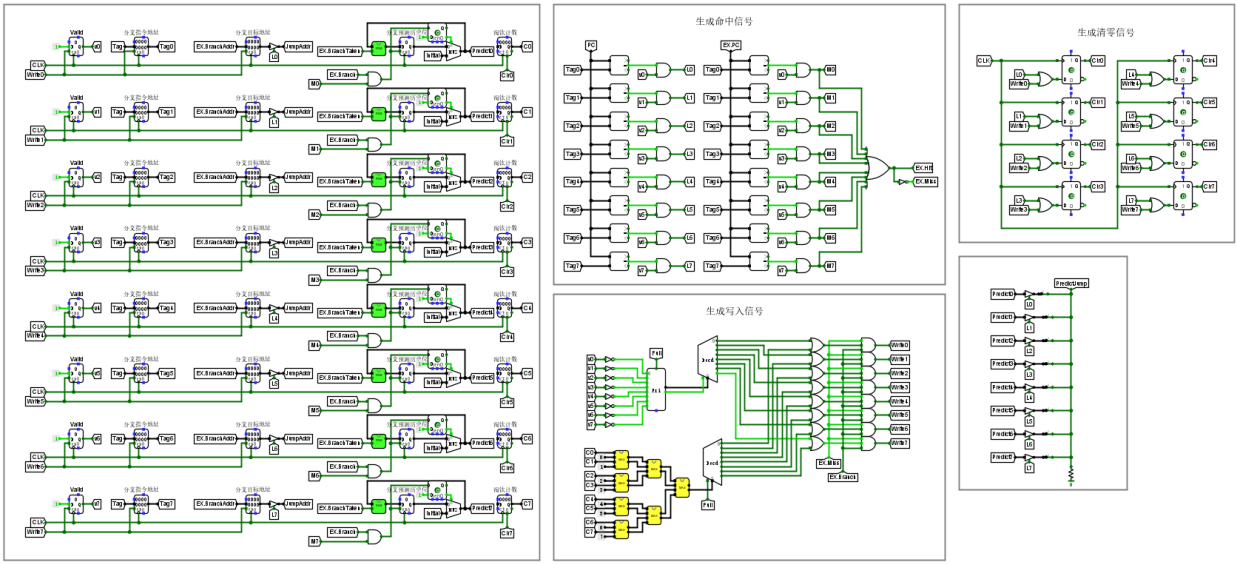

图 3.29 BTB 表

3.7 流水中断机制实现

我们选择在重定向流水线的基础上添加单级中断机制，相关逻辑与单级中断完全相同，我们在设计部分重点讨论的是保护现场的实现，仅有此部分需要进行特殊设计，设计思想参照 2.7.1 的设计原理，最终保护现场实现如图 3.30 所示。

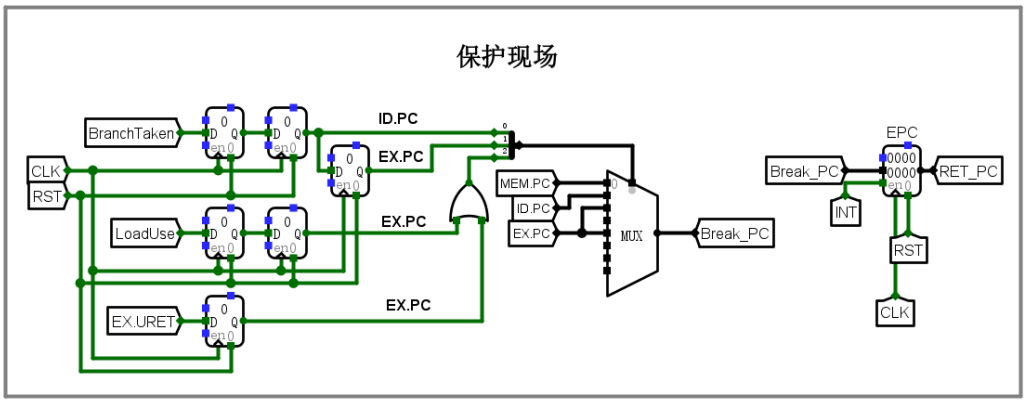

图 3.30 流水中断保护现场

同时我们还需要利用中断请求信号 INT 清空全部流水寄存器的值，保证产生中断的瞬间 IF、ID 和 EX 段的指令不执行，而 MEM 和 WB 段的指令执行结束，并且保存这一瞬间的 EX 段指令地址作为断点。

3.8 基于单级中断的 2048 游戏实现

依照我们在 2.8 中设计的模块，逐个进行实现。

1. RISC-V 代码编写

首先设计 ecall 指令对应的生成随机数与更新棋盘的信号生成逻辑，如图 3.31 random/update 信号生成所示，利用比较器和逻辑门即可生成。

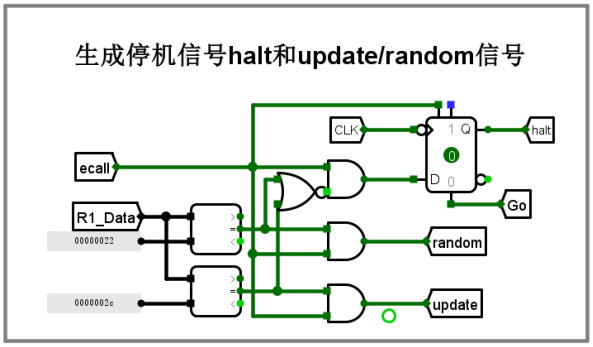

图 3.31 random/update 信号生成

接着进行初始化操作，生成一个随机位置与随机数 2 或 4，并更新棋盘，进入一个等待玩家按键中断的死循环之中，如下所示：

```
    # 预处理：生成一个随机数并在棋盘中显示
      addi a7,zero,34
      ecall #生成随机数
      addi a7,zero,44
      ecall #显示棋盘布局
main_loop:
      j main_loop #循环等待中断响应/
```

接着要实现玩家按下 4 个方向键对应的中断操作，因为这 4 个方向的实现原理相同，所以我们以上方为例：首先将所有单元格数字向上移动到顶部，保证各个数字之间在上方向上相邻，如图 3.32 Step1 所示，然后进行相邻单元格的自上而下的合并，每进行一次合并都要消除掉新产生的空格如图 3.32 Step2 所示，因为 logisim 上的渲染速度有限，为了提高代码得执行速度，对代码进行了尽可能得优化，对于全 0 列进行判断，避免对全 0 列进行无效处理，自下而上得监测空格的出现，如果出现空格才会进行数据的上移操作，实现原理如图 3.32 所示。

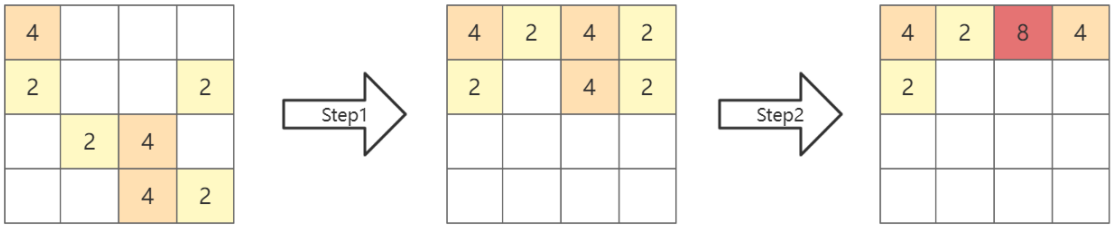

图 3.32 2048 代码实现原理

在执行完上述处理之后还要进行返回前操作，需要注意的是有可能玩家按下按键之后棋盘的布局没有任何变化，即此时按下的方向键是无效的，此时不应该生成随机数，因此我们还需要监测玩家是否进行了有效的按键操作，以决定是否生成随机数，在中断返回前的处理阶段决定是否生成随机数并更新棋盘布局。上按键对应的中断服务部分代码如下所示：

```
InteruptProgram_UP:
# 依次处理 1(0,4,8,12),2(1,5,9,13),3(2,6,10,14),4(3,7,11,15)列
 … #初始化首末寄存器和有效操作标志寄存器
move_up:
…  #利用 lw 载入从 MEM 数据到 s0,s1,s2,s3
# 处理全 0 列，加快程序速度
 bne s0,zero,move_up_start
 bne s1,zero,move_up_start
 bne s2,zero,move_up_start
 bne s3,zero,move_up_start
 j move_up_over
move_up_start:
# 最上层格子不动，只需要扫描下面三行
 bne s2,zero,move_up_1  #第 2 格非空则跳转
 beq s3,zero,move_up_1  #第 3 格为 0 不需要移位
 …      #移位操作
move_up_1:
 bne s1,zero,move_up_2  #第 1 格非空则跳转
 beq s2,zero,move_up_2  #第 2 格为 0 不需要移位
```

```
  …        #移位操作
move_up_2:
  bne s0,zero,move_up_3   #第 0 格非空则跳转
  beq s1,zero,move_up_3   #第 1 格为 0 不需要移位
  …        #移位操作
# 此时该列所有数都相邻，进入合并阶段
move_up_3:
  bne s0,s1,move_up_4
  … #合并与移位操作
move_up_4:
  bne s1,s2,move_up_5
  beq s1,zero,move_up_6   #此处 s1 为 0 说明已经不再需要考虑后续合并操作
  … #合并与移位操作
move_up_5:
  bne s2,s3,move_up_6
  beq s2,zero,move_up_6   #此处 s2 为 0 说明已经不再需要考虑后续合并操作
  … #合并与移位操作
move_up_6:  #4 个数据写回
  … #利用 sw 进行数据写回
move_up_over:
  … #生成随机数并更新棋盘布局
move_up_finish:
  uret #中断返回
```

2. 同步存储模块

因为 CPU 模块与支持单级中断的单周期 CPU 差别不大，所以不再过多赘述，此处实现同步存储模块，它的内部由 16 个 RAM 构成（用 32 位寄存器也可实现），与 MEM 类似，本模块也有使能端、时钟端、复位端，不同的是有 16 个输出端，内部实现如图 3.33 所示。

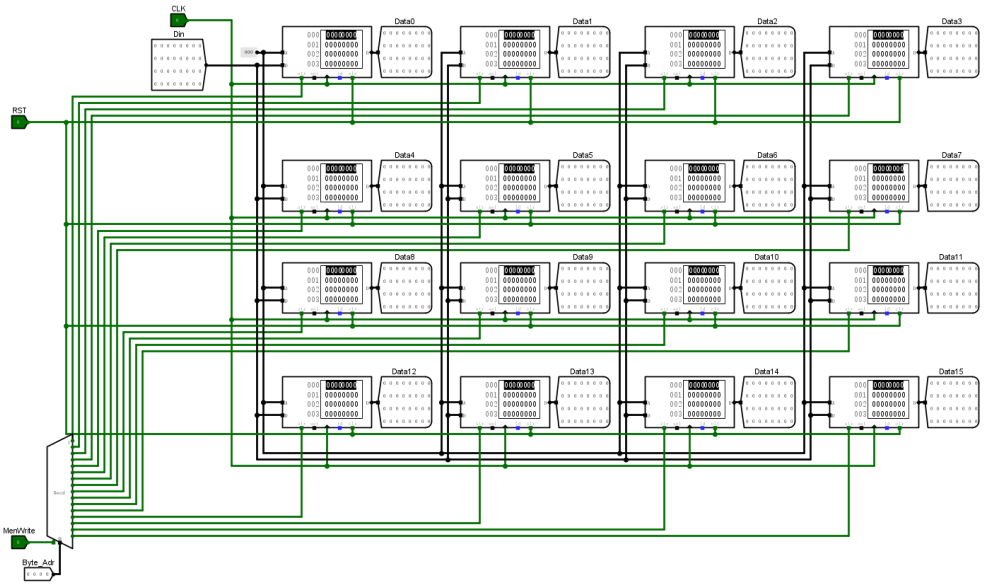

图 3.33 同步存储模块

随机数在生成后会靠硬件自动进行存储，而不需要在软件中利用 sw 指令存储，其实现原理如图 3.34 所示，产生随机数产生模块输出的随机数在实时变化，令 a7=34 并执行 ecall 指令时会产生 random 信号即写使能，在时钟上升沿将数据写入 RAM 中。

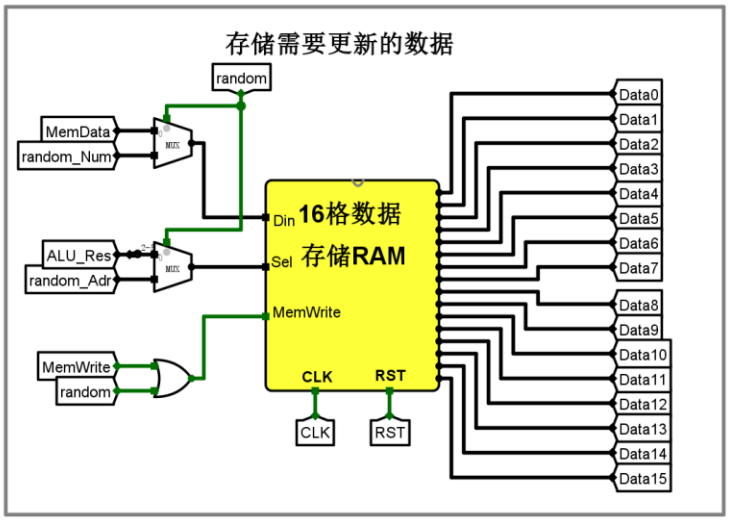

图 3.34 同步存储模块端口逻辑

3. 随机数生成模块

随机数生成的原理如 2.8 所述，将 16 个数据两两一组分为 8 组，每组中若两个数字都为 0 或都非 0 则随机选择一个数字，否则选择那个为 0 的数字，由此得到 8 个数字，继续两两分组分为 4 组，同理得到 4 个数字，再分为 2 组得到 2 个数字，最终得到 1 个数字，而这个数字的编号就是随机数生成的位置，该模块如图 3.35 所示。

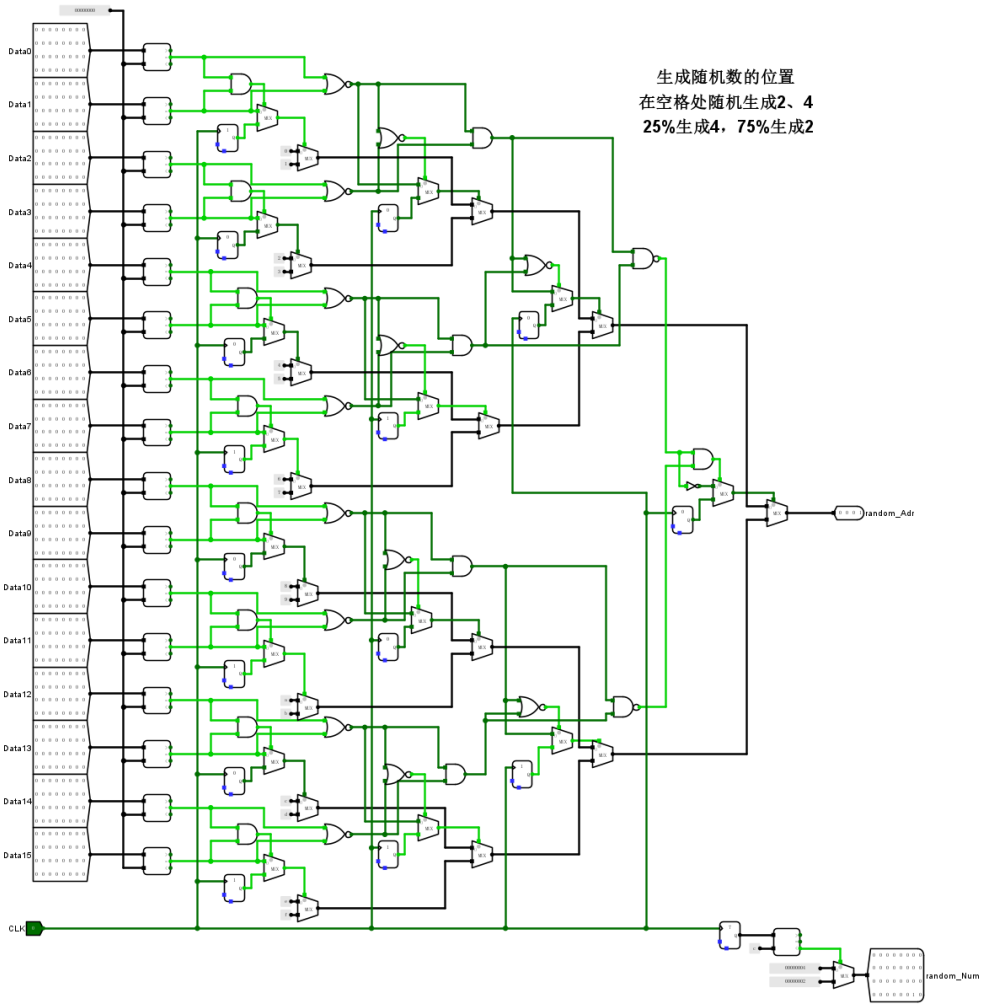

图 3.35 随机数生成模块

4. 显示模块

类似于汉字字库，将自定义的“数字字库”存储到 ROM 中，根据输入的数字实时输出 LED 编码，将 16 个单元格的数据输入到 LED 译码器中即可，如图 3.36 所示。

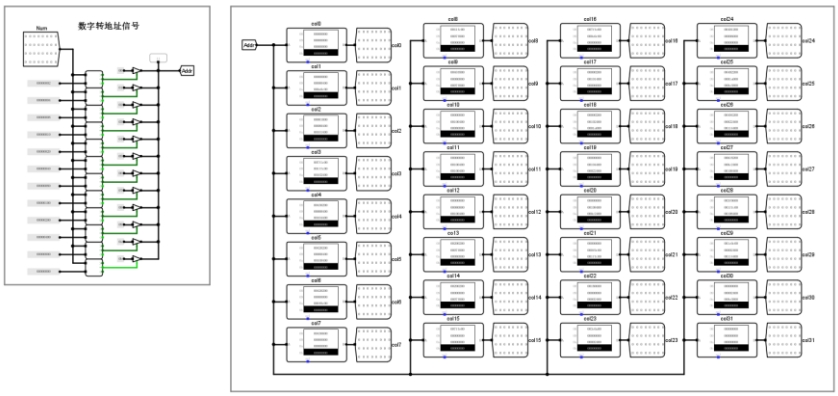

图 3.36 LED 译码器

最终实现的 2048 游戏如图 3.37 所示，玩家利用戳工具选中键盘组件后通过输入 WSAD 来完成上下左右的按键操作，按下一个按键就是触发一个中断源，CPU 会执行相应的中断服务程序实现对棋盘的更新。

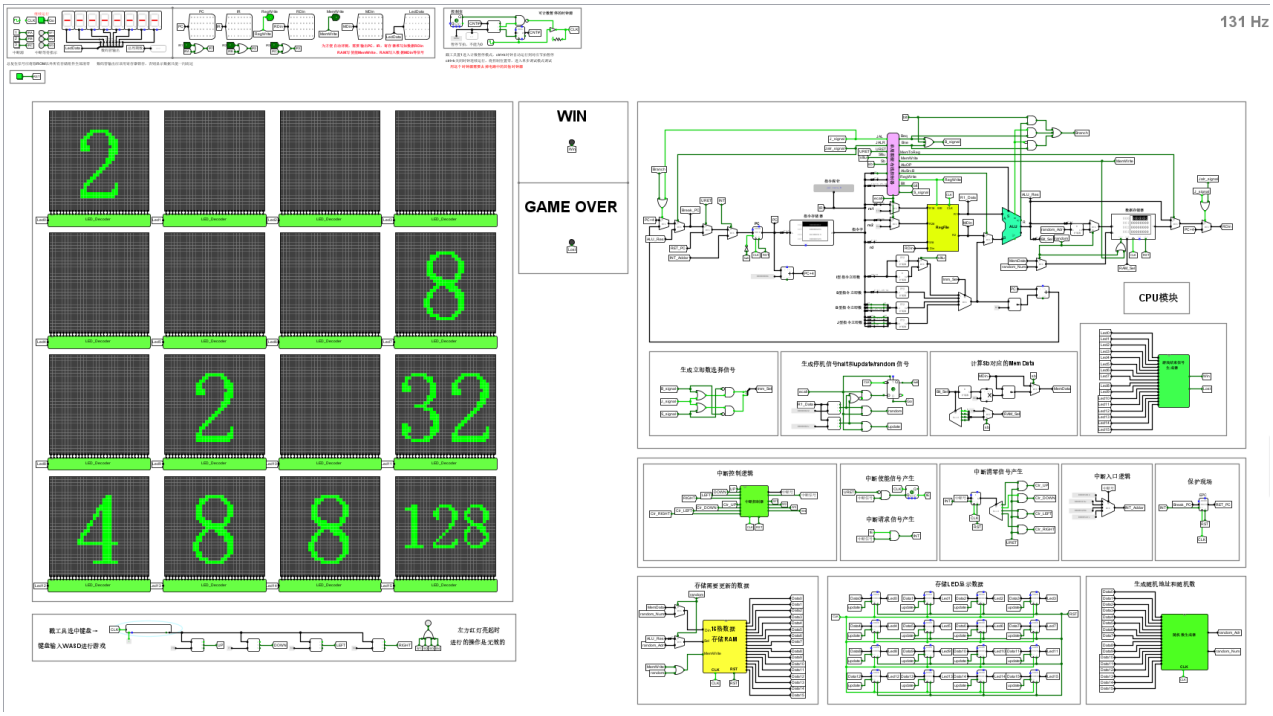

图 3.37 2048 游戏整体实现

经过测试，游戏可以正常得运行并且会正确得反映游戏的结束，如图 3.38 所示，当棋盘已经无法移动时 Game Over 的指示灯亮起。

图 3.38 游戏结束示例

4 实验过程与调试

4.1 测试用例和功能测试

4.1.1 测试用例 1

单周期 CPU 与流水线测试主要通过 risc-v-benchmark_ccab.hex 程序进行测试，可以在该程序中添加 4 条差异化指令。该程序的功能是走马灯式得显示数字，最后进行+4 计数，计数到 0x38 时进行停机。

4.1.2 测试用例 2

中断机制的测试主要用到 risc-v 单级中断测试程序.hex 和 risc-v 多级中断测试(EPC 硬件堆栈保护).hex，中断服务程序的入口从这两个程序中得出，1-3 号中断源对应的中断服务程序是执行 3 次相应中断号的走马灯式显示，主程序是进行走马灯式计数显示，不会主动停机。

4.2 测试结果

各个模块的测试结果记录如表 4.1 所示。气泡流水线的总周期数的计算公式为：总周期数=1546+4+气泡数-1；重定向流水线的总周期数计算公式为：总周期数=1546+4+分支误取深度*分支数+load-Use 次数-1；动态分支预测 1 对应的分支预测历史位初始值为 00，动态分支预测 2 对应的分支预测历史位初值为 01。气泡与重定向正确说明理想流水线正确，中断机制没有具体的数据记录，但是现象正确。

表 4.1 测试结果记录

模块	总周期数	无条件分支数	条件分支成功数	插入气泡数	LoadUse 次数	LED 显示
单周期 CPU	1546					38
气泡流水线	3624	38	276	2074		38
重定向流水线	2297	38	276	748	120	38
动态分支预测 1	1781	38	276	232	120	38
动态分支预测 2	1773	38	276	224	120	38

4.3 性能分析

气泡流水线因为要支持分支指令并且要解决数据冲突，所以会插入很多气泡，导致它的周期数比单周期 CPU 多得多，根据表 4.1 的记录结果，我们可以发现 2074 个气泡中有(38+276)×2=628 个气泡是分支指令导致，而剩下的 1446 个气泡是数据冲突导致的，这显然是需要优化的重点。

重定向流水线解决了大部分的数据冲突插入的气泡，虽然在 LoadUse 时仍然需要插入 1 个气泡，但是我们将数据冲突的 1446 个气泡减少到了 120 个，极大优化了气泡流水线，那么接下来优化的重点是分支指令插入的气泡。

动态分支预测在重定向流水线的基础上结合 8 路全相联 cache 对分支指令得跳转地址进行预测，预测成功则会减少两个气泡，预测失败不会增加多余得气泡，这使得分支预测历史位初值为 00 时分支指令的 628 个气泡减少到了 104 个，分支预测历史位初值设置为 01 时会有更进一步的优化，使气泡数再减少 8 个。

4.4 主要故障与调试

4.4.1 流水中断故障

中断请求信号的清零段没有选择正确。

故障现象：流水中断有时并不能正确得响应中断。

原因分析：一开始中断请求信号负责清空 ID/EX、EX/MEM 和 MEM/WB 寄存器的值，我们将发生中断瞬间在 IF、ID、EX、MEM、WB 段的指令记为 IR1~IR5，我们设计的是如果 IR3 不是 Nop 则将 IR3 指令的地址设置为断点，在下一个上升沿产生中断请求信号，此时 IR5 执行结束，IF 段进入了一个新指令 IR0，在下一个上升沿中断请求信号会清空 ID/EX 和 EX/MEM 的值，此时 IR4 执行结束，IR3、IR2、IR1 被清空，而我们并没有处理此时处于 ID 段的 IR0，如果 IR0 是一条将要跳转的分支指令，那么它在清空误取指令时会把将中断服务程序的指令清空。

解决方案：用中断请求信号 INT 清空所有流水寄存器的值，保证产生中断瞬间 IF、ID、EX 段的指令不执行，MEM、WB 段指令执行结束，同时下一个进入 IF 段的指令也不允许执行。

4.4.2 单周期上开发板故障

大小写错误导致输入悬空引脚。

故障现象：开发板显示全 0，程序并不能运行。

原因分析：经过查看原理图，发现在硬布线控制器的输入段出现了如图 4.1 所示的接地情况，显然 OpCode 端输入的不是正确数据。

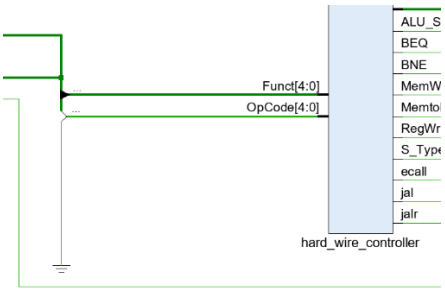

图 4.1 引脚悬空现象

解决方案：查看顶层模块，发现声明的 wire 型变量名为“OpCode“，但是实例化硬布线控制器时将输入端写成了“Opcode”，修改之后即可正确得生成控制信号。

4.5 实验进度

表 4.2 课程设计进度表

时间	进度
第 1-2 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，根据指令功能填写 Excel 表格生成控制信号并设计好单周期 CPU
第 3 天	为单周期 CPU 添加差异化指令，学习理想流水线相关知识，设计流水接口部件并将单周期 CPU 拆分成五个阶段构成理想流水线
第 4 天	学习气泡流水线相关知识，理解插入气泡的原理，完成气泡流水线的设计
第 5 天	学习重定向流水线相关知识，理解重定向的原理，完成重定向流水线的设计
第 6 天	复习组成原理中断的相关理论知识，参考单总线 CPU 的中断逻辑，完善 Excel 表格中 URET、CSRRSI、CSRRCI 对应控制信号的填写，设计好单级中断
第 7 天	修改单级中断电路将其设计成可以支持多级中断的单周期 CPU
第 8-9 天	完成流水中断机制和动态分支预测，完成部分故障报告
第 10 天	实现单周期 CPU 的 Verilog 代码编写，并成功烧入 FPGA 开发板，经过调试与仿真可以正常运行 benchmark，添加切换频率与显示的数据的功能

5 设计总结与心得

5.1 课设总结

基于单周期 CPU 的五段流水 CPU 设计，从理想流水线开始一步步完善与优化流水线功能，使我深入了解了流水线的工作原理与性能优化方式，作如下总结:

1) 设计了支持中断与动态分支预测的五段流水 CPU、支持 FPGA 运行的 Verilog 项目、可在 Logisim 上运行的游戏。

2) 实现了支持 24+4 条指令的单周期 CPU、只能支持简单指令的理想流水线、可以处理分支指令的气泡流水线、性能更佳的重定向流水线、支持中断机制的流水线、性能最佳的动态分支预测流水线。

3) 完成了从设计到优化的一个高性能流水线 CPU 并在 FPGA 上实现单周期 CPU 的过程, 并尝试了利用自己设计的 CPU 在 Logisim 上运行简单游戏。

5.2 课设心得

本次课程设计是我迄今为止做过的最有难度、多样性最高同时也是收获最大的项目，我不仅在 Logisim 平台实现了五段流水 CPU 并设计了一款游戏，还用 Verilog 语言在开发板上运行自己设计的单周期 CPU，这充分地锻炼了我的自主学习能力。

课程设计在上手时还是有些迷茫的，不知道先要做什么，一直在纠结 uret、csrrci 和 csrrsi 指令在单周期 CPU 中的实现，后来知道这三条指令在处理中断前不需要考虑才开始了正式的设计，单周期 CPU 参照课本与任务书还是比较顺利就设计好了，此时才算找到感觉，后面流水线的设计也是很快就完成了，在书写报告时也是进一步得理解了其工作原理与优化方式。遇到的最大得困难是单周期上开发板，一开始选择自动转化，但是顶层模块无法正确转换出来，最终还是选择手写 Verilog 实现，这就导致了出现了许多粗心得错误，各个模块的原理实现得完全没有问题，就是因为运算符或者大小写的错误导致仿真一直出错，不过最后也是顺利解决了这些问题。

目前在团队任务设计阶段, 希望以后的课程设计可以适当提供 FPGA 驱动显示器的资料，有助于完成团队演示系统的设计。最后还要感谢各位老师在课程设计过程中不断得进行答疑解惑，让本对硬件不感兴趣地我拥有了对硬件设计的热情。

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] 谭志虎，秦磊华，吴非，肖亮.计算机组成原理. 北京：人民邮电出版社，2021 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：魏子腾