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：	CS2009
学　　号：	
姓　　名：	raindaydream
电　　话：	187xxxx5392
邮　　件：	daydream@hust.edu.cn

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。硬件综合训练课程是完成该计算机组成原理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。

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。

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

（1）支持规定的 32 位 RISC-V 指令集（指令集任选），具体见表 1.1；

（2）在 CCAB 扩展指令集中支持 2 条 C 类运算指令，1 条 A 类存储指令，1 条 B 类分支指令，具体任务每位同学不一样，指令编号详见见公文包中的任务分配；

（3）支持多级嵌套中断，利用中断触发扩展指令集测试程序；

（4）支持支持 5 段流水机制，可处理数据冒险、结构冒险、分支冒险；

（5）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。

（6）能运行教师提供的标准测试程序，并自动统计执行周期数。

（7）能自动统计各类无条件分支指令数目，条件分支成功次数、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

#	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	SLL	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	
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	

#	RISC-V	指令类型	简单功能描述	备注
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	If $a7==34 LED 输出 $a0 的值 else 等待 GO 按键暂停
22	CSRRSI	I	访问 CP0	中断相关，可简化为开中断
23	CSRRCI	I	访问 CP0	中断相关，可简化为关中断
24	URET	I	中断返回	清中断，MEPC 送 PC，开中断
25	SLL	R	逻辑左移	
26	XOR	R	按位异或	
27	LHU	I	无符号半字加载	
28	BLT	B	小于跳转	

2 总体方案设计

在本次课程设计，需要使用 LOGISIM 来创建一个 32-位 5 段流水 CPU，该 CPU 运行的是标准 RISC-V 指令集的子集，关于指令功能请仔细查阅 RISC-V 指令手册，常见指令格式如图 2.1 所示：

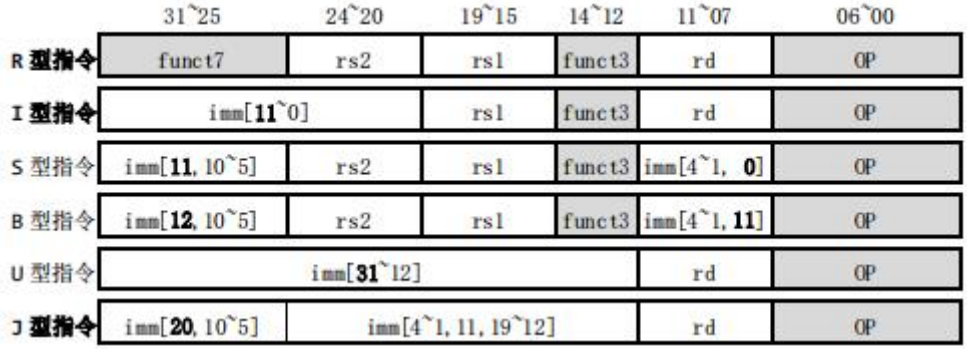

图 2.1 RISC-V 指令格式

2.1 单周期 CPU 设计

该 CPU 为 RISC-V 类型，本次我们采取的方案是硬布线控制器的方法，采用了三级时序硬布线控制器。在这里我们采用了指令存储器和数据存储器相分离的哈佛结构来实现方案的设计。在设计过程中，我们将硬布线控制器中的控制信号和运算信号分开生成封装再组成硬布线控制器，组成后再加上指令的数据通路就可以得到 CPU 的整个电路图。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，我们只需先完成支持一条固定 R 型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。

总体结构图如 2.2 所示。

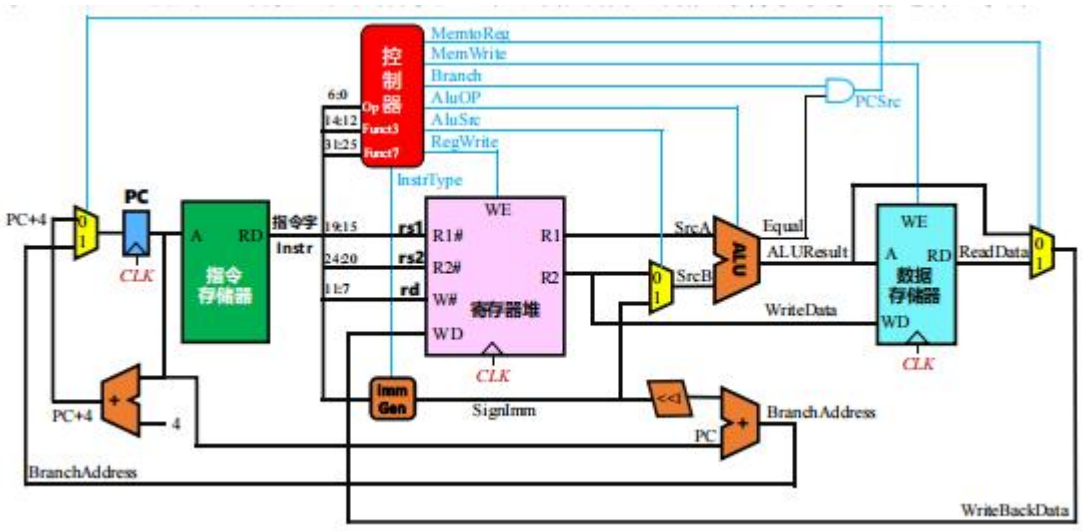

图 2.2 单周期方案数据通路顶层试图

2.1.1 主要功能部件

1. 运算器

运算器的各个引脚说明如表 2.1 所示，其基础功能在课程设计的要求中已经提供，这里不再赘述。

表 2.1 算术逻辑运算单元引脚与功能描述

引脚	输入/输出	位宽	功能描述
X	输入	32	操作数 X
Y	输入	32	操作数 Y
ALU_OP	输入	4	运算器功能码，具体功能见下表 2.2
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
NOTLESS	输出	1	比较器不小于的标志
LESS	输出	1	比较器小于的标志
Equal	输出	1	Equal=(x==y)?1:0, 对所有操作有效

2. 寄存器堆 RegFile

寄存器堆有读操作和写操作。读操作需要输入两个寄存器的地址，输出两个寄

存器所保存的数据；写操作为输入写入的数据和写入的地址和写使能信号，无输出；这两个操作都由时钟端控制。

3. 数据存储器

数据存储器有读操作和写操作。读操作输入地址，输出一个数据；写操作需要输入地址、数据、写使能信号，输出无意义。

2.1.2 数据通路的设计

表 2.2 指令系统数据通路框架

<table>
<tr><th rowspan="2">指令</th><th rowspan="2">PC</th><th colspan="4">RF</th><th colspan="2">ALU</th><th colspan="2">DM</th></tr>
<tr><th>R1#</th><th>R2#</th><th>W#</th><th>Din</th><th>A</th><th>B</th><th>Addr</th><th>Din</th></tr>
<tr><td>ADD</td><td>PC+4</td><td rowspan="18">X[RS1]</td><td rowspan="18">X[RS2]</td><td rowspan="19">X[RD]</td><td rowspan="17">ALUR</td><td>R1</td><td rowspan="6">R2</td><td rowspan="19">ALUR</td><td rowspan="19">R2</td></tr>
<tr><td>SUB</td><td>PC+4</td><td>R1</td></tr>
<tr><td>AND</td><td>PC+4</td><td>R1</td></tr>
<tr><td>OR</td><td>PC+4</td><td>R1</td></tr>
<tr><td>SLT</td><td>PC+4</td><td>R1</td></tr>
<tr><td>SLTU</td><td>PC+4</td><td>R1</td></tr>
<tr><td>ADDI</td><td>PC+4</td><td>R1</td><td rowspan="12">X[31:20]</td></tr>
<tr><td>ANDI</td><td>PC+4</td><td>R1</td></tr>
<tr><td>ORI</td><td>PC+4</td><td>R1</td></tr>
<tr><td>XORI</td><td>PC+4</td><td>R1</td></tr>
<tr><td>SLTI</td><td>PC+4</td><td>R1</td></tr>
<tr><td>SLLI</td><td>PC+4</td><td>R1</td></tr>
<tr><td>SRLI</td><td>PC+4</td><td>R1</td></tr>
<tr><td>SRAI</td><td>PC+4</td><td>R1</td></tr>
<tr><td>LW</td><td>PC+4</td><td>MEMR</td><td>R1</td></tr>
<tr><td>SW</td><td>PC+4</td><td>/</td><td>R1</td><td>X[7:11,25:31]</td></tr>
<tr><td>ECALL</td><td>PC+4</td><td>17</td><td>10</td><td>/</td><td>R1</td><td>R2</td></tr>
<tr><td>BEQ</td><td>PC+4/PC+OFFSET</td><td>X[RS1]</td><td>X[RS2]</td><td>/</td><td>R1</td><td>R2</td></tr>
</table>

指令	PC	RF				ALU		DM	
		R1#	R2#	W#	Din	A	B	Addr	Din
BNE	PC+4/PC+OFFSET	X[RS1]	X[RS2]	X[RD]	/	R1		ALUR	R2
JAL	PC+OFFSET				PC+4	R1			
JALR	(PC+OFFSET)&~1				PC+4	R1	X[31:20]		
SLL	PC+4				ALUR	R1	R2		
XOR	PC+4				ALUR	R1	R2		
LHU	PC+4				MEMR	R1	X[31:20]		
BLT	PC+4/PC+OFFSET				/	R1	R2		

2.1.3 控制器的设计

首先对于控制信号进行统计，包括各个主要部件所需要输入的控制信号，以及数据通路合并表中所示的具有多输入的主要部件需要进行输入选择的控制信号，并且对各个统计信号的各种取值情况进行定义，统计得到的控制信号以及说明如表2.4。

表 2.3 主控制器控制信号的作用说明

控制信号	取值	说明
Ecall	0	寄存器堆 R1、R2 口读取 rs 字段指示寄存器的值
	1	寄存器堆 R1、R2 口读取 2 号寄存器的值，控制 halt 信号
S_type	0	立即数取 I 型指令对应的字段
	1	立即数取 S 型指令对应的字段
JAL/JALR	0/1	取值为 1 时跳转，PC 取跳转地址的值
BEQ/BNE	0/1	取值为 1 且需要满足的条件成立时跳转，PC 取跳转地址的值
ALUOP	0-12	选择 ALU 的功能
ALUSRCB	0/1	确定 ALU 的 B 输入为 R2 还是立即数
MEMWRITE	0/1	取值为 1 时允许写入数据存储器
MEMTOREG	0/1	取值为 1 时写入寄存器的数据为读取到的数据存储器中的值
REGWRITE	0/1	取值为 1 时允许写入寄存器
HALF	0/1	取值为 1 时说明结果取半字

控制信号	取值	说明
CSR	0/1	取值为 1 时说明当前指令为 CSRRCI/CSRRSI

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图所示。

#	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	half	CSR
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								1					
19	bne		1	18									1				
20	jal			1b					1					1			
21	jalr		0	19	5			1	1						1		
22	CSRRSI		6	1c	8			1									1
23	CSRRCI		7	1c	7			1									1
24	URET		0	1c						1							
25	SLL	0	1	c	0				1								
26	XOR	0	4	c	9				1								
27	LHU		5	0	5	1		1	1							1	
28	BLT		4	18	11												

图 2.3　主控制器控制信号框架

2.2 单周期 CPU 单级中断机制设计

2.2.1 总体设计

在已经实现的单周期 CPU 基础上增加中断处理部分的逻辑电路，单级中断需要完成①中断响应、②保护现场、③中断服务、④恢复现场、⑤开中断、⑥中断返回的流程来处理。

中断响应：关中断，再由中断信号在中断使能段有效时产生中断响应信号，对断点地址进行保存。

保护现场：需要对寄存器进行保存，寄存器用指令保存在 RAM 中。

中断服务：将中断处理程序的入口地址送入地址寄存器。

恢复现场：寄存器恢复为原始值，寄存器写回采用指令从 RAM 中取出。

开中断：中断使能位恢复为 1。

中断返回：保存的断点地址写回指令地址寄存器。

2.2.2 硬件设计

本实验中，中断源有两个：①定时产生；②人为按键。在中断源存在的基础上，采用中断按键电路产生稳定的中断请求。中断响应信号由中断信号存在并且中断使能位有效的情况下产生。中断使能位由当前的执行程序的情况和中断请求一起作用决定。断点地址由寄存器保存，中断处理程序的入口地址由多路选择器选择。

2.3 单周期 CPU 多级中断机制设计

2.3.1 总体设计

在已经实现的单周期 CPU 基础上增加多级中断处理部分的逻辑电路，多级中断需要完成①中断响应、②保护现场、③开中断、④中断服务、⑤关中断、⑥恢复现场、⑦开中断。

多级中断响应和处理和单级中断的响应和处理大部分是相同的流程，不同为：

保护现场：多级中断增加对中断屏蔽字的保护。

开中断：更高级的中断可以中断低级中断的中断处理程序。

关中断：恢复现场的时候需要关中断保证寄存器的值正确。

恢复现场：还需要恢复中断屏蔽字。

2.3.2 硬件设计

多级中断处理的硬件设计和单级中断有许多共同之处，共同的部分不再赘述，可以查看上面的单级中断的硬件设计部分。

多级中断增加的点为：①断点地址采用寄存器数组进行保存；②中断号也采用了寄存器数组进行保存，因为需要记录中断的级数，在该方面我采用了硬件实现的方法解决；③中断使能位的判断逻辑也不同了，因为多级中断增加了一个开中断和关中断，因此需要添加信号来区分两个开关中断。

2.4 理想流水 CPU 设计

2.4.1 总体设计

流水线 CPU 在指令执行过程细分为五个阶段，每个阶段由对应的功能部件完成，五个阶段为 IF→ID→EX→MEM→WB：

取指令(IF)：负责从指令存储器取出指令；

指令译码(ID)：操作控制器对指令字进行译码，同时从寄存器堆取操作数；

指令执行(EX)：执行运算操作或计算地址；

访存(MEM)：对存储器进行读写操作；

写回(WB)：将指令执行结果写回寄存器堆。

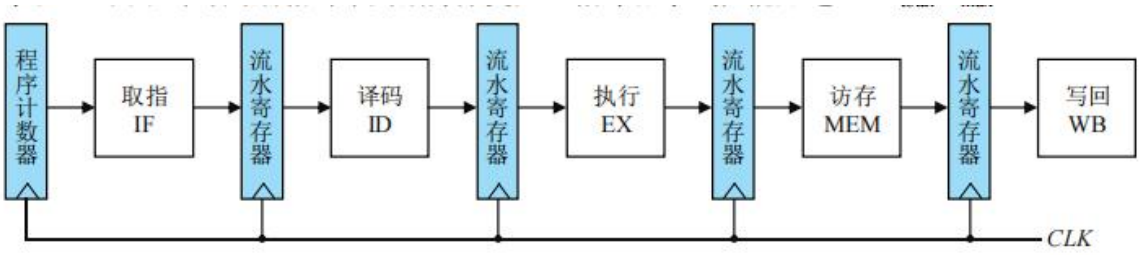

图 2.4　流水线逻辑架构

2.4.2 流水接口部件设计

流水线需要将每个阶段的信息保存一个周期，然后将必要的信息传递给下一个阶段，因此流水接口部件以时钟源、复位键、要保存的数据为输入接口，要传递给下一阶段的数据为输出接口，采用寄存器实现。

2.4.3 理想流水线设计

对单周期 CPU 进行改造，将其取值、译码、执行、访存、写回的部分分开，并在这几个阶段之间添加流水接口部件，使得几个阶段在执行上独立并且串行起来。对于该数据通路图中需要跨越不同阶段的数据都需要通过流水接口部件进行保存。整体的逻辑可以参照 2.4.1 中的图 。

2.5 气泡式流水线设计

2.5.1 冲突检测逻辑

冲突主要有结构冲突、控制冲突、数据冲突三种类型。其中结构冲突可以采用哈佛结构将数据和指令分开存储来解决；控制冲突可以采用分支预测的方法解决；数据冲突可以采用插入气泡的方法解决。

结构冲突采用改变硬件存储的方式解决，控制冲突和数据冲突采用增加冲突检测逻辑模块的方式解决。在控制冲突发生时，需要增加清空信号将分支指令的指令清空，在数据冲突时，需要增加气泡生成信号来增加气泡使得数据暂停，因此中断检测逻辑检测 EX 和后续阶段的指令信号来生成解决冲突需要的控制信号。

冲突检测逻辑如下图所示，通过 EX、MEM 段是否有数据写入内存与 ID 段的指令所取的寄存器地址相匹配来得出冲突逻辑值。

```
DataHazzard =  RsUsed & (rs≠0) & EX.RegWrite  & (rs==EX.WriteReg#)
             + RtUsed & (rt≠0) & EX.RegWrite  & (rt==EX.WriteReg#)
             + RsUsed & (rs≠0) & MEM.RegWrite & (rs==MEM.WriteReg#)
             + RtUsed & (rt≠0) & MEM.RegWrite & (rt==MEM.WriteReg#)
# rs、rt 分别表示指令字中的 rs、rt 字段，分别对应指令字中的 25~21、20~16 位
# RsUsed、RtUsed 分别表示 ID 段指令需要读 rs、rt 字段对应的寄存器
# EX.RegWrite 表示 EX 段的寄存器堆写使能控制信号 RegWrite，锁存在 ID/EX 流水寄存器中
# MEM.WriteReg#表示 MEM 段的写寄存器编号 WriteReg#，锁存在 EX/MEM 流水寄存器中
```

图 2.5　冲突检测逻辑表达式

```
Stall = DataHazzard                          # 数据相关时要阻塞暂停 IF、ID 段指令的执行
PC.EN = ~Stall                               # 程序计数器 PC 使能端输入
IF/ID.EN = ~Stall                            # IF/ID 寄存器使能端输入
IF/ID.CLR = BranchTaken                        # 出现分支跳转时要清空 IF/ID
ID/EX.CLR = Flush = BranchTaken + DataHazzard    # 出现分支或数据相关时要清空 ID/EX
```

图 2.6　冲突检测逻辑输出信号

2.5.2　气泡流水线

气泡流水线由理想流水线引入冲突检测逻辑得到，并且需要将流水接口部件的接口更新加入相关的控制信号。理想流水线中流水接口部件只接受简单的时钟信号和异步复位信号以及使能信号，在引入冲突检测逻辑之后，需要给接口部件增加同步清空信号，同时在输入时增加清空逻辑、使能逻辑。

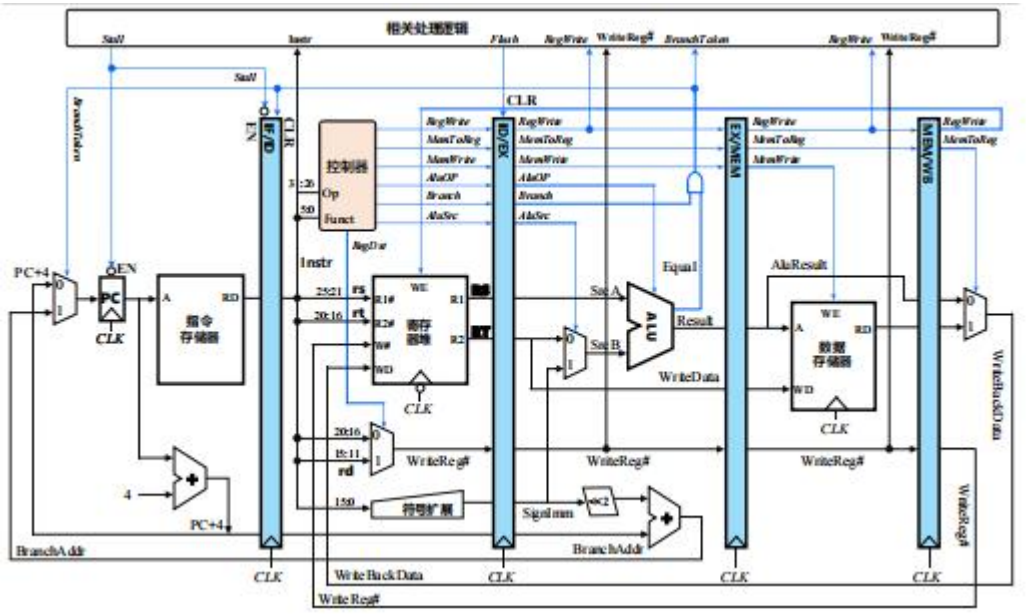

图 2.7　气泡流水线顶层试图

2.6 重定向流水线设计

重定向原理不会直接在冲突时直接将前面的 ID 阶段的数据清空，而是正常的运行流水线，但在取值的时候，采用重定向将 EX 段取出的数据进行纠正。但是该方案对于 load-use 相邻的指令并不适用，因为传递给 EX 阶段的值可能并非写好的值，因此会导致错误，那么该指令下依旧需要通过增加气泡的方式来解决数据冲突。

因此重定向需要增加一个和气泡流水线相似的冲突检测逻辑模块，该模块通过 EX 阶段的信号决定重定向信号和气泡插入信号。逻辑表达式如图所示，而顶层试图和气泡流水线基本相同。

```
LoadUse =  RsUsed & (rs≠0) & EX.MemRead & (rs==EX.WriteReg#)
+ RtUsed & (rt≠0) & EX.MemRead & (rt==EX.WriteReg#)
# 注意单周期 CPU 实现中为了简化电路，只实现了 MemWrite 写信号，没有实现 MemRead 信号，但由于该信号和 MemToReg 信号是同步的，所以可以用 MemToReg 信号代替 MemRead 信号
```

图 2.8 load-use 检测逻辑

```
IF/ID.CLR = BranchTaken                    # 出现分支跳转时要清空 IF/ID
ID/EX.CLR = Flush = BranchTaken + LoadUse  # 分支跳转或 Load-Use 相关时要清空 ID/EX
PC.EN = ~Stall                             # 程序计数器 PC 使能端输入
```

图 2.9 信号逻辑

2.7 流水段中断机制设计

2.7.1 单级 EX 中断

在实现单周期 CPU 单级中断时，已经在前面的 2.2 中阐述了单级中断的原理，而流水线单级中断与单周期 CPU 基本相同，所以这里不再赘述，具体原理可以查看 2.2 中的总体设计部分。

但在实现部分，流水线中断和单周期 CPU 中断还是有所不同。在单周期 CPU 中，由于一个周期下保存的断点和恢复的断点只有一个 PC 值，因此不需要做出选择，而流水线中在一个时钟周期下会有五个 PC 值，因此需要选择一个固定的阶段来中断。

这里将 EX 作为中断点，对 EX 段要执行的指令进行中断，在选择下一个 PC 地址和进行冲突逻辑检测时可以将中断与跳转看作同等地位来执行相关指令，但是相比跳转，中断增加了断点保存部分、中断信号处理逻辑、PC 地址选择逻辑、中断使能判断逻辑等与中断有关的逻辑。

2.7.2 多级 EX 中断

流水线多级中断的原理与单周期 CPU 多级中断的原理基本相同，因此具体原理可以查看 2.3 中的总体设计部分。但在实现方面与单周期 CPU 不同，和流水线单级中断有许多相似之处。

在多级中断中，我们仍采用 EX 作为中断点，对 EX 段要执行的指令进行中断。但是与单级中断不同的是，中断使能信号的判断逻辑发生了改变，这里引入了两个信号 CSRRCI、CSRRSI 来辅助判断中断使能信号，由于高级中断可以在低级中断的过程中继续中断，因此这两个 CSR 信号可以用来判断是否开关中断。而断点保存以及中断信号保存与单周期 CPU 多级中断相同，PC 选择逻辑与流水线单级中断相同。

2.8 动态分支预测机制

2.8.1 动态分支预测技术

在执行程序的过程中，有一些分支跳转为循环跳转指令，因此该类的分支跳转的下一条指令在循环过程中有较为固定的值，因此可以将该条指令放在分支跳转指

令后面直接执行从而不必再引入气泡从而提高执行效率。

动态分支预测技术便是在分支跳转的过程中记录其第一次跳转时的跳转值并且根据跳转的次数来改变该分支跳转的状态从来决定是否选择要跳转的下一条指令。

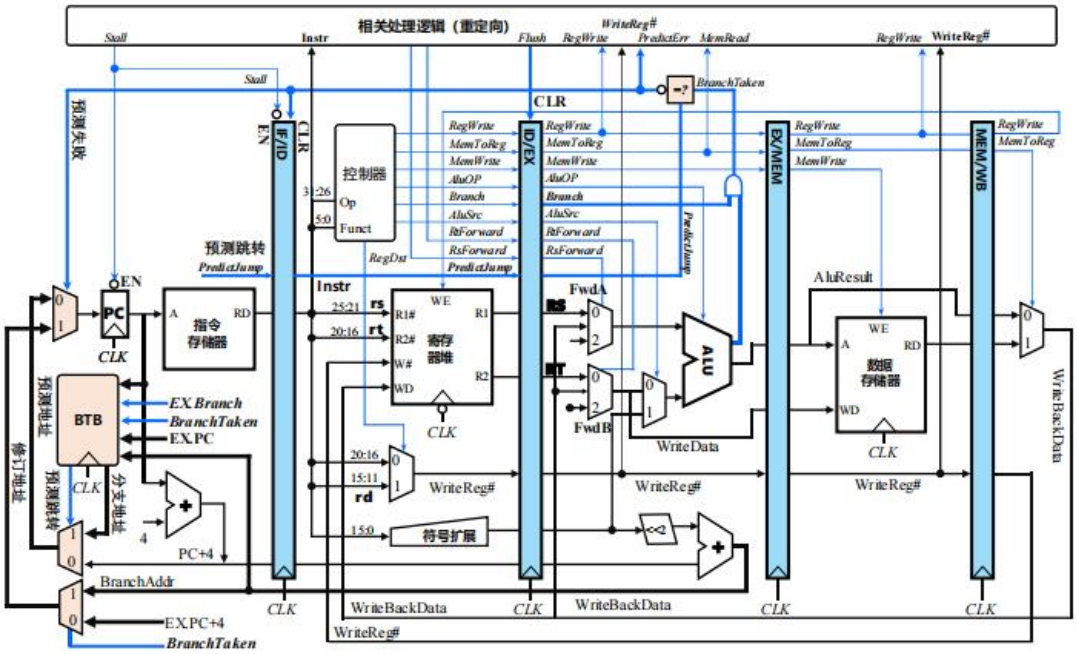

图 2.10　动态分支预测技术的顶层文件

2.8.2　BTB

1.　BTB 原理

BTB 表即为动态分支预测中的跳转指令查询表，BTB 表中包含 valid 位、分支指令地址，分支目标地址，预测状态位、置换标记五项，其中保存的每一项的跳转指令都包含这五项，用于查询。

在某一分支跳转指令第一次出现时，BTB 表会为它建立一项用来初始化其值；如果当前 BTB 中仍有空白项，那么初始化的该项便可以直接写入该空白项，否则会根据置换标记将最久没有被使用过的条目替换掉给新的分支跳转的指令。

在载入该分支跳转指令之后，如果该指令重新出现，那么就会更新其预测状态位和置换标记位，置换标记会直接清零表示该指令目前刚被访问，预测状态位会根据之前的状态和当前的访问操作一起产生新的次态来为后续的查询服务。

其中预测状态位需要用有限状态机 FSM 来实现，而表项的更新和写入需要采用分支选择逻辑、写入逻辑、预测结果输出逻辑等模块实现。

2. 数据通路中 BTB 的实现

在采用了 BTB 之后，分支跳转指令不必在该指令之后一定加气泡来使得下一条指令为正确的跳转指令，而是在 EX 阶段判断该分支跳转的跳转点和 ID 阶段的指令是否相同来决定之后是否要加入气泡来确保之后的指令执行顺序正确。因此冲突检测逻辑的输入需要按照该原理进行更新。

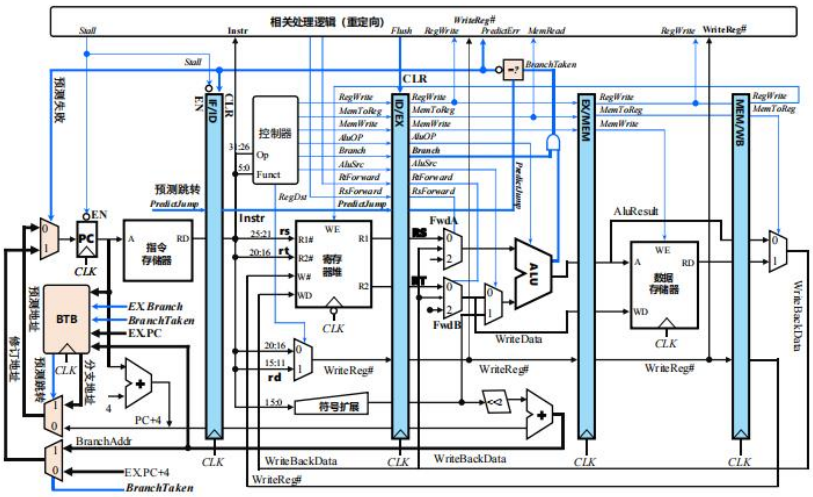

图 2.11 动态分支预测技术的数据通路

2.9 单周期 CPU 上板

CPU 上板的方式主要有两种：①logisim 转 verilog；②手写 verilog 实现 CPU。本次我采用了第一种方式＋verilog 下 debug 的方式来实现了 CPU 上板。

采用第一种方式相较简单，首先需要根据手册将电路种的空输入、三态、标签等细节修改完整，然后通过工具转成 verilog 的源文件，在之后便可以在 vivado 上对文件进行综合实现来修正语法错误，并编写 testbench 来检测逻辑正确性，在综合实现正确并且 tb 正确之后便可以生成比特流上板成功。

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）

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，GO 为继续运行信号，halt 通过非门取反之后和 GO 信号相或，当需要进行停机时，Halt 控制信号为 1，经过非门之后为 0，传入使能端，屏蔽时钟信号，使整个电路停机。如图 3.1 所示。

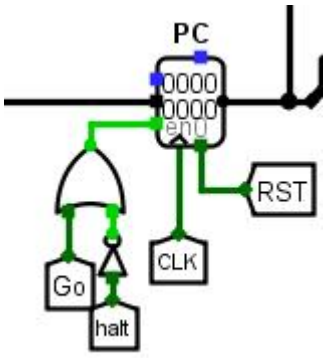

图 3.1 程序计数器（PC）

2. 指令存储器（IM）

使用一个只读存储器 ROM 实现指令存储器（IM）。设置该只读存储器的地址位宽为 10 位，数据位宽为 32 位。因为 PC 中存储的指令地址有 32 位，而 ROM 地址线宽度有限，仅为 10 位，故将 32 位指令地址高位部分和字节偏移部分直接屏蔽，使用分线器只取 32 位指令地址的 2-11 位作为指令存储器的输入地址。

3.1.2 数据通路的实现

本次课程设计采用的工程化的设计模式，一次性构建所有的数据通路。主要实现方法为，对于每一条指令，将其改写成 RTL（Register Transfer Level），忽略控

制类信号，仅保留数据类信号，根据 RTL 功能填写对应指令的数据通路表，描述五大部件之间的连接关系，记录各部件输入端数据来源。

根据总体方案设计中数据通路设计那一小节的详细内容，具体分析每一条指令在执行过程中各个主要部件的输入和输出端口的连接，完成指令系统数据通路表的填写，如表 2.3 所示。

在完成指令系统数据通路表的填写之后，根据列出的数据通路表，进行多指令数据通路的合并输入数，表，将各个主要功能部件进行连接，根据数据通路合并表的最终结果，对于所有的多输入部件使用多路选择器进行输入选择。最终便可以完成数据通路的搭建。

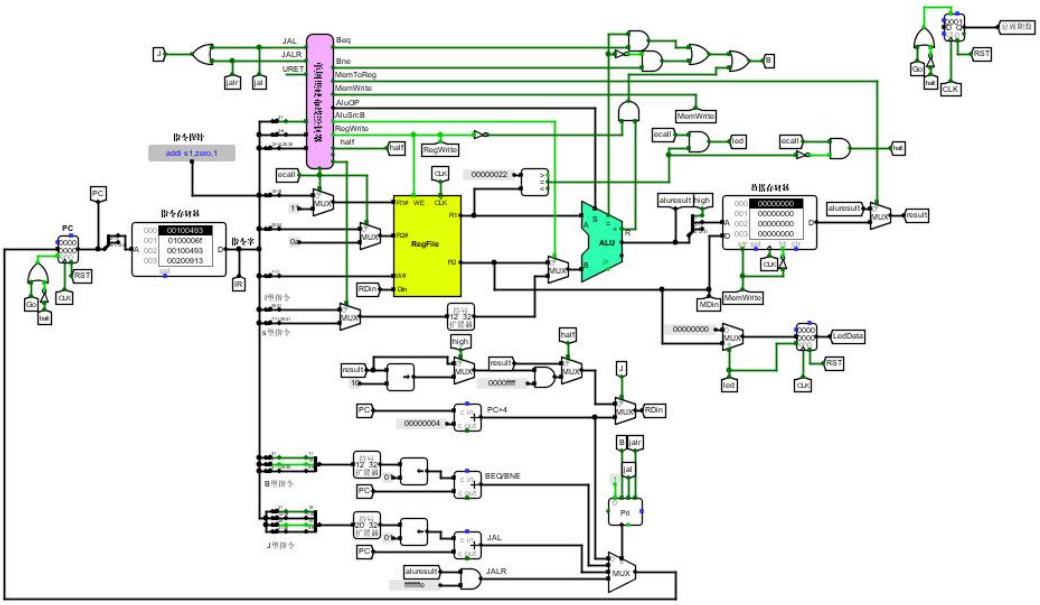

图 3.2 单周期 CPU 数据通路（Logism）

3.1.3 控制器的实现

单周期硬布线控制器的输入为指令中的 OP_CODE、FUNCT、IR21 三个操作字段，输出为各个控制信号。

在控制器的实现中，我们可以采用 logisim 中的组合逻辑电路自动生成的功能来自动生成电路，因此需要我们先输入各个控制信号和输入的变量的逻辑关系。由于 logisim 中的自动生成电路的功能对引脚的数目有限制，因此我们将该部分的控制信号分成两部分来自动生成，一部分为运算控制信号，一部分为其他控制信号。这两部分的信号可以根据前面的方案设计部分得到逻辑表达式，输入到 logisim 中自动生

成两个组件后即可组装成完整的单周期硬布线控制器。

3.2 中断机制实现

3.2.1 单周期 CPU 单级中断

1. 中断信号产生

中断源产生中断请求信号，每一个中断源根据中断产生逻辑产生一个中断响应信号，其中中断产生逻辑引用了提供的中断按键参考电路。

2. 中断使能控制

在单级中断中，在进入一个中断处理程序之后，中断使能端会被关闭从而保证其不会被打断，因此中断使能端可以由中断信号和中断返回信号控制，在没有中断信号或者一个中断处理程序结束时，中断使能端会置 1，因此可以采用或门和寄存器来实现 IE。

3. 中断响应阶段

中断响应需要当前有中断信号并且中断使能端为 1 的时候才会产生，因此采用与门实现。

4. 断点地址保存

在产生中断响应时需要将当前的 PC 值存入寄存器，因此可以用中断响应信号做使能端控制寄存器将地址写入。

5. 中断返回地址选择

根据 RARS 转换出的代码找到中断处理程序对应的入口地址，采用优先编码器和寄存器得到当前的中断逻辑号得到入口地址并将其传递给 PC 端的选择接口。根据该中断逻辑号，还需要将中断返回信号对中断信号进行清空，该清空操作需要生成清空信号，并将该清空信号给予中断产生逻辑。

6. PC 值的选择

在要执行的下一条指令处，需要选择①正常处理的下一条指令，②中断处理程序的入口地址，③保存的断点地址。该选择由中断响应信号、中断返回信号来决定，当产生中断响应信号时选②，产生中断返回信号时选③，否则就选①。

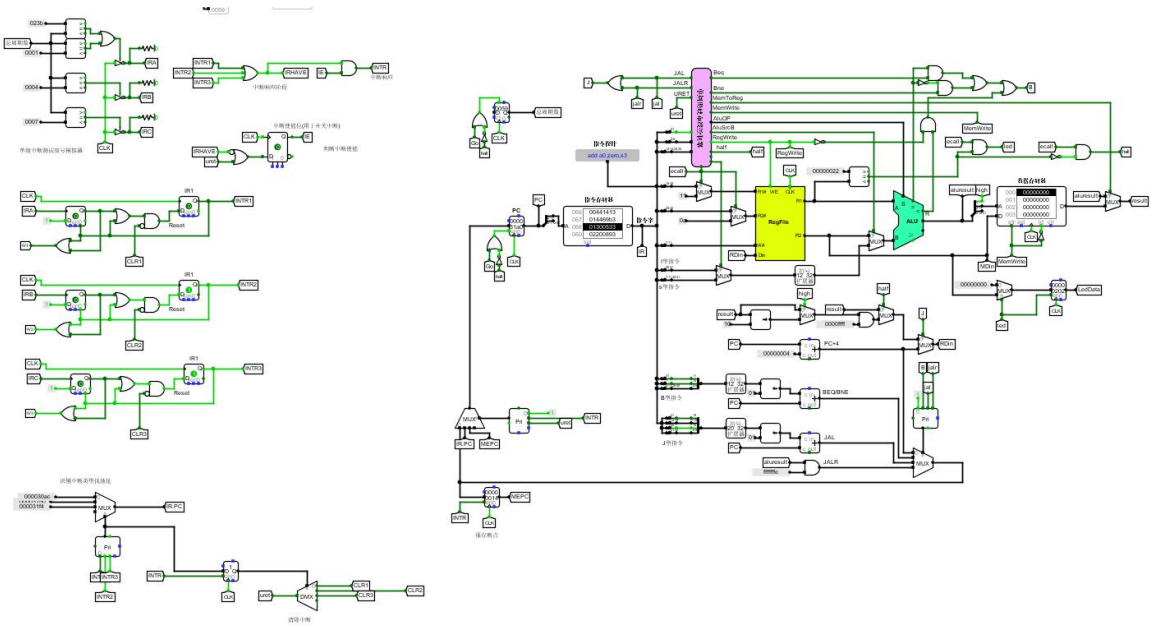

图 3.3　单周期 CPU 单级中断数据通路

3.2.2　单周期 CPU 多级中断

多级中断的实现相较于单级中断基本相同，但是在中断使能逻辑、中断响应产生逻辑、断点保存和中断信号保存上有一些差别。

1.　中断使能逻辑

在多级中断中，高级中断可以在低级中断处理程序中继续中断，因此在中断处理程序的准备工作结束后需要将中断使能为置为 1，直到最后程序结束的恢复阶段，因此中断使能逻辑需要加入 CSRRCI 和 CSRRSI 两个信号，来表示中断处理程序的两个阶段。在 CSRRCI 为 1 时关中断，在 CSRRSI 为 1 时开中断，这两个信号可以与中断返回信号和中断响应信号一起作用与中断使能位。

2.　中断响应产生逻辑

在原本的中断响应逻辑中，只需要有各自的中断响应信号即可，而多级中断需要增加当前中断与已有中断的比较，将该比较结果用于判断中断响应信号。

3.　断点保存和中断信号保存

多级中断需要保存多个断点地址，因此需要串行的地址寄存器和中断号寄存器来保存产生过的中断信号。在增加了一级中断时，中断地址和中断号都要向后传递一位，在中断返回时，需要将所有的中断地址和中断号出栈一个，后面的都向前推

一位。因此在寄存器的输入位需要加一个选择器并用中断返回信号决定出栈还是入栈。

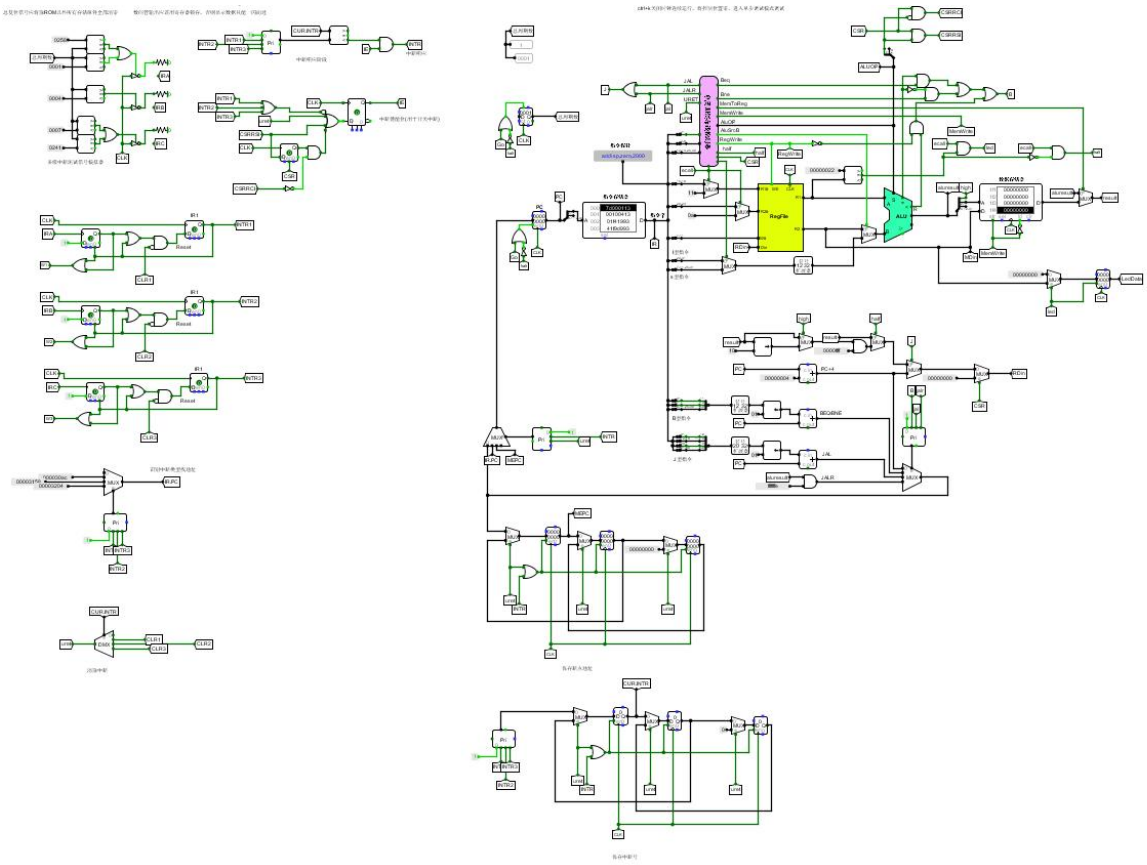

图 3.4 单周期 CPU 多级中断数据通路

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

流水接口部件的主要功能是在一个周期中保存该阶段的值来保证下一个时钟周期的各个阶段可以使用上一个阶段的运算结果。每个阶段要保存的值不完全相同，因此需要设计四个流水接口部件：IF/ID、ID/EX、EX/MEM、MEM/WB。这些接口部件由寄存器实现，采用同一个时钟周期进行保存，并且使用相同的复位信号。

3.3.2 理想流水线实现

理想流水线不考虑数据冲突等问题，只按照前面的 2.4.1 所阐述的原理将单周期 CPU 划分为五个阶段然后将这五个阶段需要的数据在流水接口部件中保存并向下

传递即可，如图所示。

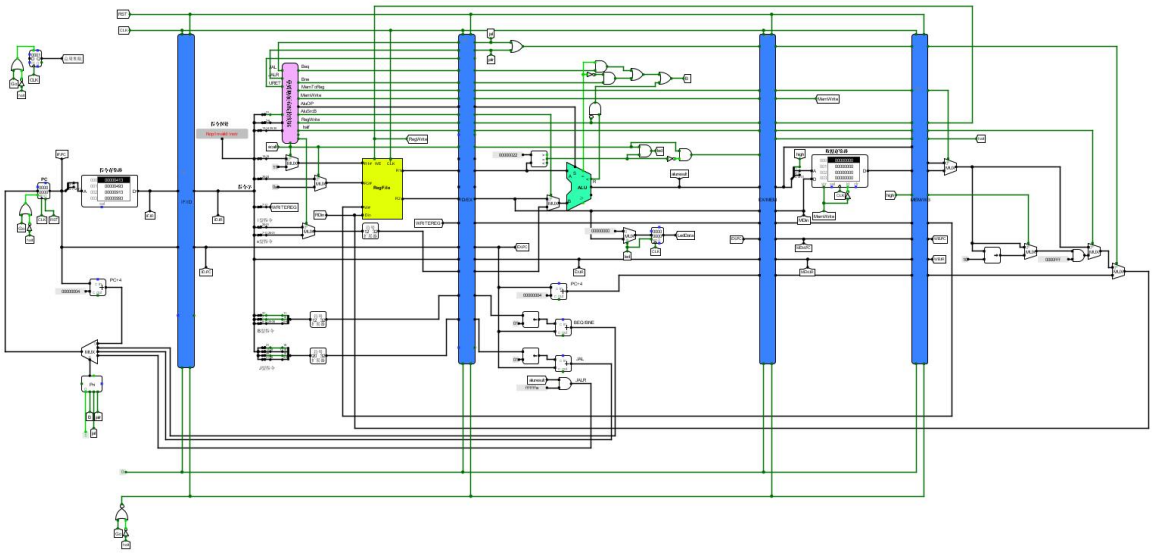

图 3.5 理想流水线的具体实现

3.4 气泡式流水线实现

3.4.1 冲突检测逻辑的实现

冲突检测逻辑需要根据 ID 的指令和后续 EX 和 MEM 段的相关信号判断是否存在冲突来增加气泡。具体的原理在 2.5.1 已经详细说明，根据 2.5.1 的原理公式，我们可以利用基本的与或非实现该逻辑。

3.4.2 气泡流水线数据通路

数据冲突会利用产生气泡来解决，分支需要通过清空前方指令来解决，因此需要对数据通路增加分支检测和冲突检测并增加相关信号来将信息传递给流水接口部件使其保存正确的值。

分支检测逻辑利用 J 型指令和 B 型指令的信号获得，在有分支指令并且分支指令会跳转成功时，会产生分支冲突信号并将该信号传递给 IF/ID、ID/EX 段从而清空 IF、ID 阶段的数据。

冲突信号由冲突检测逻辑模块获得，该信号需要作用于流水接口部件的选择端信号，在 ID 阶段产生一个 nop 的气泡，同时需要将该信号与分支信号一起或一下作用于 IF/ID 的清零端，从而保证指令与气泡之间没有值传递。

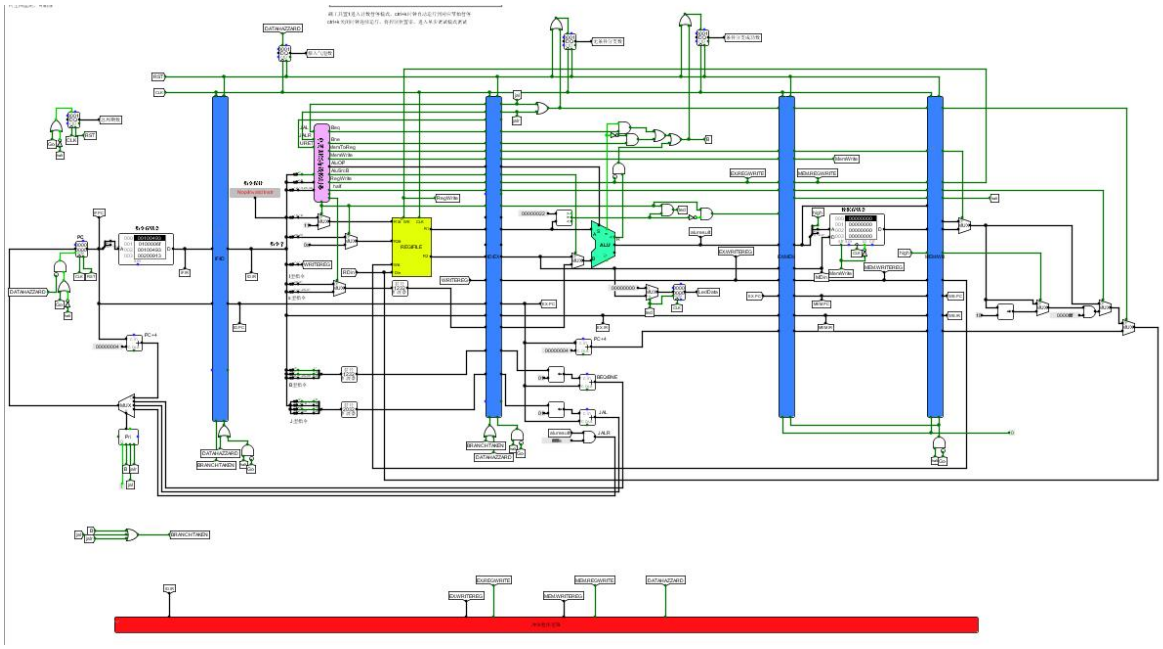

图 3.6 气泡流水线数据通路

3.5 重定向流水线实现

3.5.1 冲突检测逻辑

重定向需要检测当前所取出的寄存器值是否正确以及当前是否位 load-use 指令，因此需要产生两个寄存器冲突信号和一个 load-use 信号。

根据 2.6 中的重定向原理，我们可以知道重定向的信号产生公式，根据该公式，我们可以获得上述的三个信号。具体实现如图所示，由 ID 阶段的指令可以得到两个寄存器的使用信号，在与 IF、MEM 阶段的写信号综合之后可以得到三个信号，即根据公式通过与或非门即可。

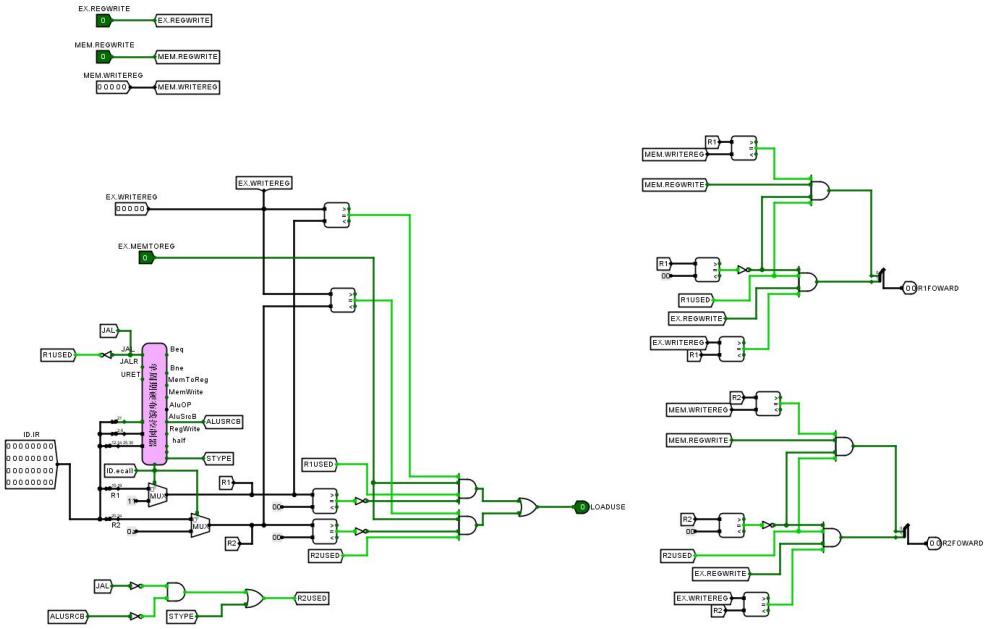

图 3.7 冲突检测逻辑实现

3.5.2 重定向流水线数据通路

Load-use 和分支跳转需要增加气泡来解决，数据访问冲突需要通过重定向解决。

分支跳转信号和气泡流水线中的产生逻辑相同，load-use 信号由冲突检测逻辑模块生成，在分支中，需要将 IF、ID 两阶段的数据清空，在 load-use 中，需要给 ID 阶段增加一个气泡，因此需要将分支信号加在 IF\ID、ID\MEM 的清零端，将 load-use 信号加载在 IF\ID 的选择端和 ID\MEM 的清零端。

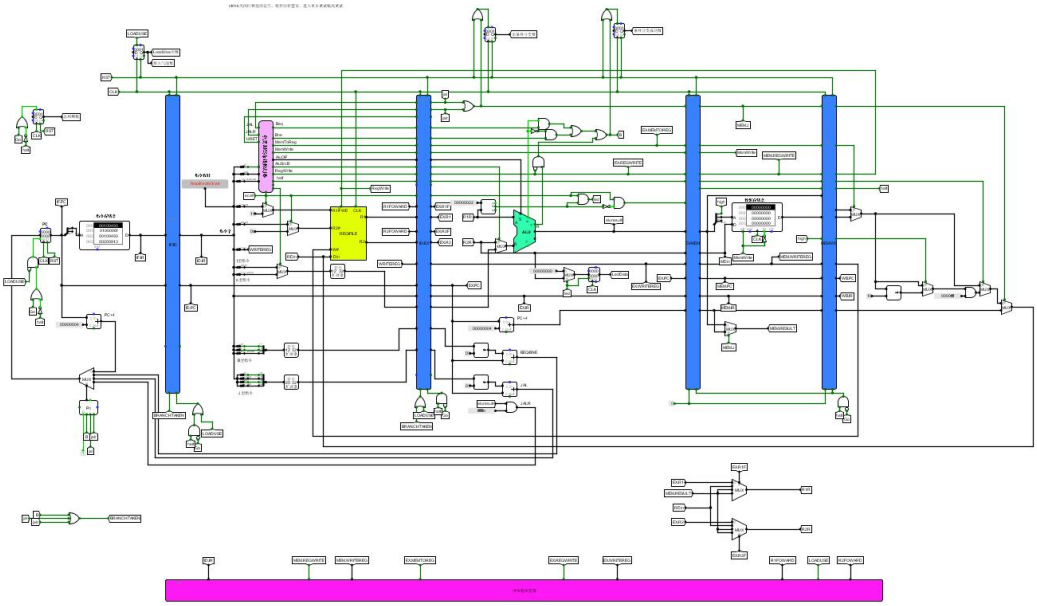

图 3.8 重定向流水线数据通路

3.6 动态分支预测机制实现

3.6.1 BTB 实现

BTB 表以 IF 的指令、EX 阶段的指令和分支冲突信号和地址、分支信号为输入，以分支跳转地址和选择信号为输出。其内部为一个全相联的 cache 结构。

地址写入逻辑选择一个空条目写入当前的分支，该写入逻辑需要找到一个空闲的条目或者最有可能被淘汰的条目写入，存在空条目的时候利用解复用器选择一个位置将该信号传递给对应的写入使能信号，如果不存在就需要比较置换标记来选择位置，采用比较器找到最大的置换标记信息，并将该位置的信息输入到解复用器的选择端中使其输出信号给要替换的位置。将有效信号传递给相应的位置之后需要将其他的写入使能信号置为 0，并将这些信号传递给 cache 的写入使能端，

更新历史预测位需要利用有限状态机来生成次态，需要将当前的要写入的条目的现态和分支跳转是否成功的信号传递给有限状态机，产生次态后根据前面所阐述的写入使能信号将状态更新在寄存器中。这里我采用 00/11 为初始状态，在第一次遇见该分支并且成功的时候初始化为 11，但如果第一次遇见该分支跳转指令却没有成功跳转的话就初始化为 00。

预测逻辑需要判断当前的 PC 指令和 cache 所保存的指令地址是否相同，如果相同就将该地址对应的预测位和跳转地址输出用于流水线中判断下一条指令的地址。

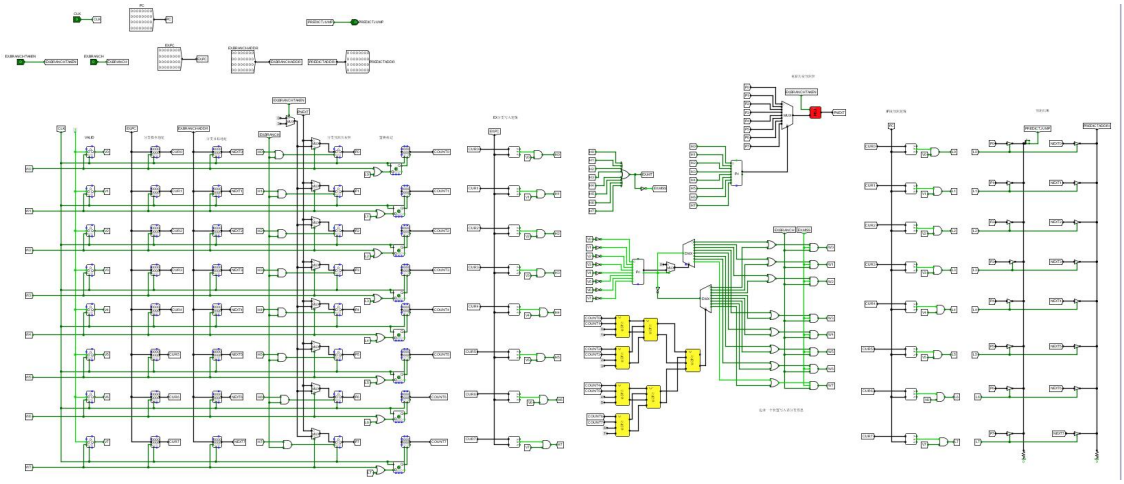

图 3.9 BTB 实现

3.6.2 动态分支预测数据通路

该数据通路在重定向流水线的基础上进行，数据冲突用重定向来解决，load-use

通过添加气泡解决。但是分支跳转通过 BTB 预测和检测是否正确来解决。

在 IF 阶段，需要增加 BTB 来查询下一条指令的地址，将当前的 IF 阶段的 PC 值传递进去并输出预测位和下一条指令地址供选择，同时还需要将 EX 阶段的指令和信号传递进去来更新 cache 的信息。

在 EX 阶段遇到分支跳转指令时需要将传递过来的预测信号和是否要跳转的信号进行比较得到 ID 阶段的指令是否正确，如果预测信号和跳转信号相同就说明 ID 阶段指令正确不需要清空 IF、ID，如果不一样就清空，即将比较信号传递给流水接口部件的清空端。

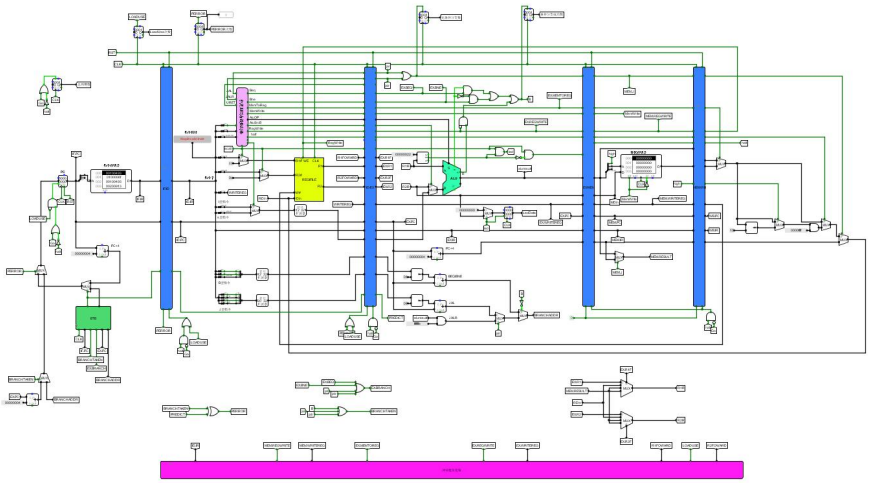

图 3.10 动态分支预测的数据通路

3.7 流水线中断

流水线中断在重定向流水线基础上实现，其实现和单周期 CPU 中断基本相同，中断响应逻辑、中断使能逻辑、中断返回逻辑都相同，但在断点地址处不相同。

在单级中断中，选择将 ID 段指令作为中断处，即该地址保存在断点地址寄存器中，但是如果 EX 阶段时跳转指令，那么 ID 阶段的指令地址将是不争取的，因此需要增加 EX 段跳转成功的信号然后在 IF 阶段通过多路选择器去选择是将 ID 阶段指令还是 EX 阶段计算出来的指令地址保存进去。

同时在进入中断处理程序时，相当于跳转，因此需要按照分支跳转指令的处理方法将前面的流水接口部件给清空保证指令顺序正确。因此可以将中断进入信号和中断返回信号与之前的分支跳转信号或一下得到新的跳转信号进入流水接口部件的清空段。

而对于多级中断，中断响应逻辑、中断使能逻辑、中断返回逻辑和中断地址的

保存方式都与单周期 CPU 相同，而中断地址选择上与流水线单级中断相同。

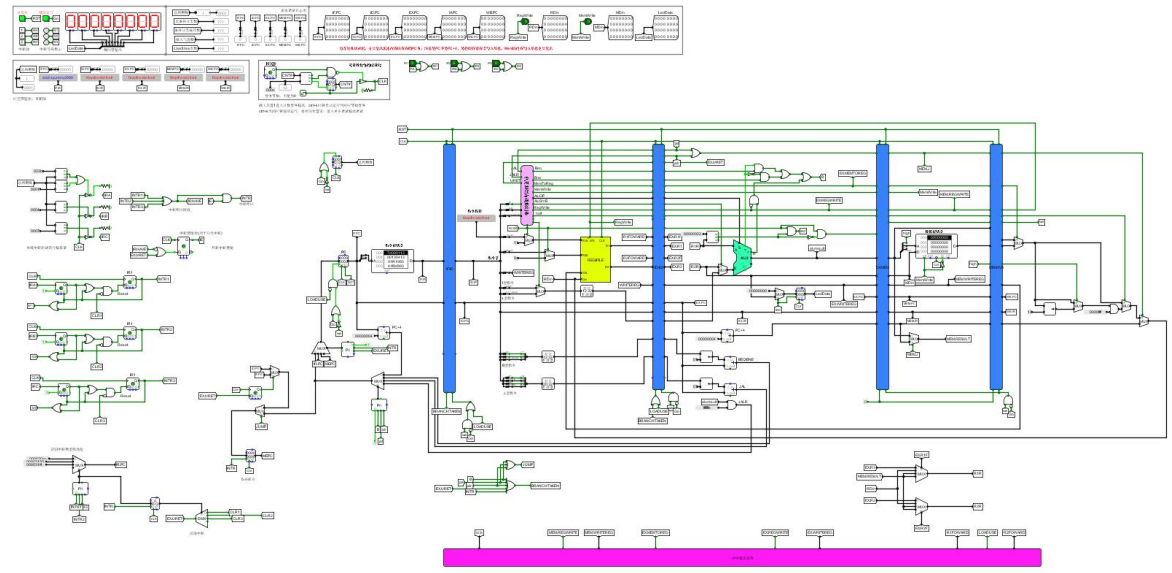

图 3.11 流水线单级中断

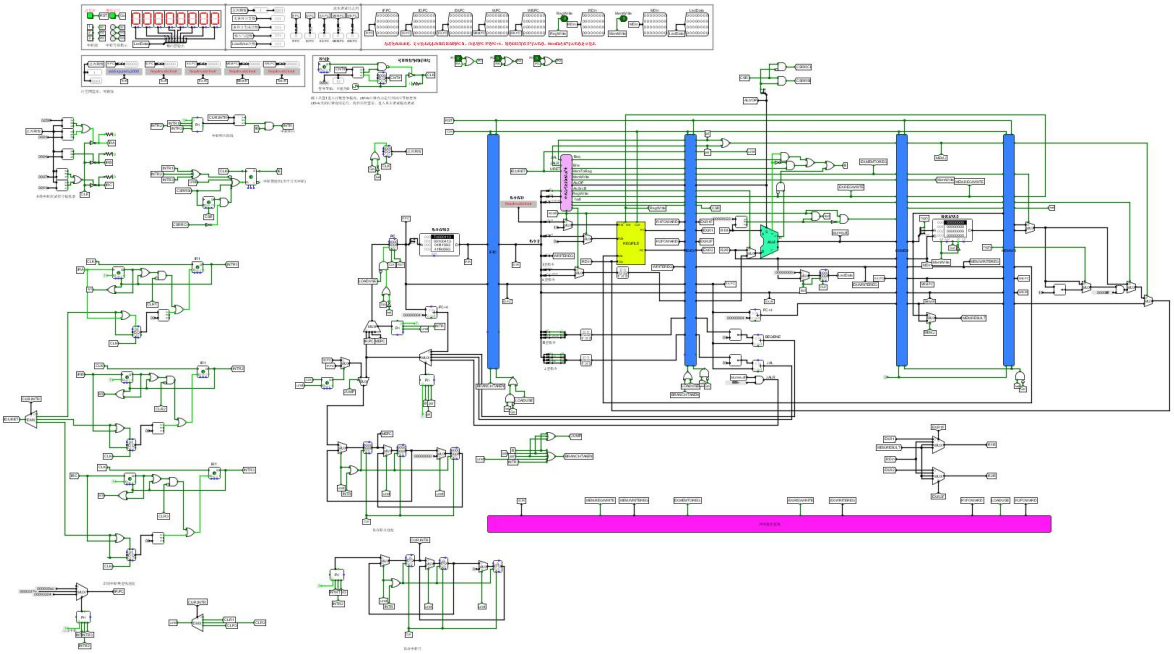

图 3.12 流水线多级中断

3.8 单周期 CPU 上板

首先需要根据 2.9 中的原理将需要修改的地方修改正确然后转 verilog 并在 vivado 中命令行打开后对文件 tb 修正即可。

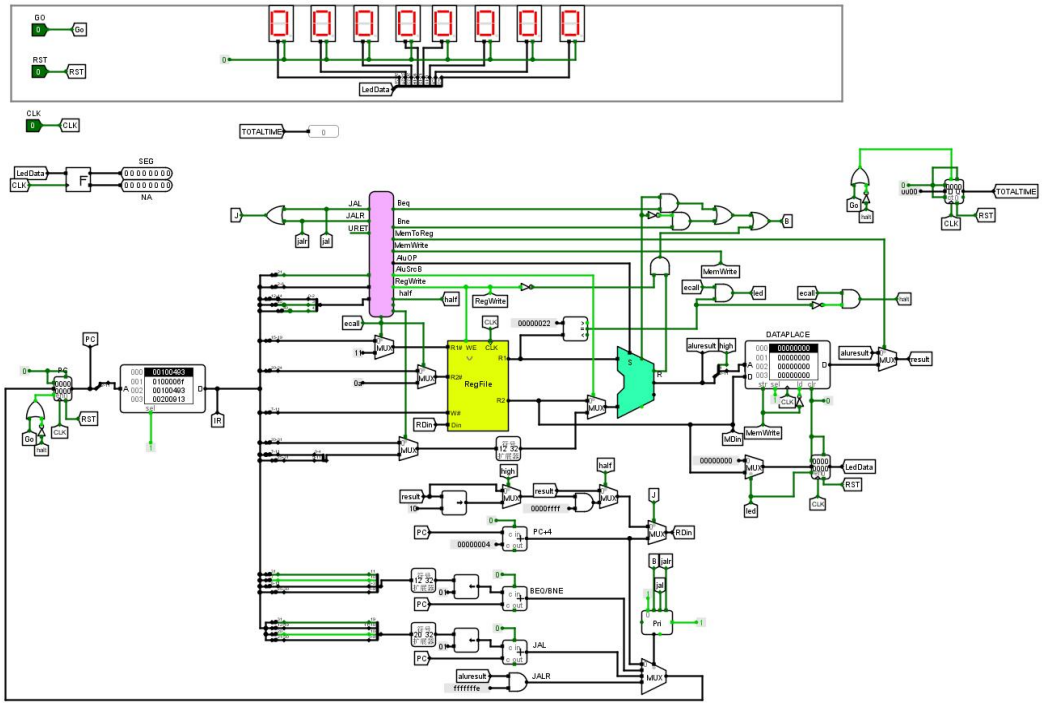

图 3.13 转换成功的 logisim 图

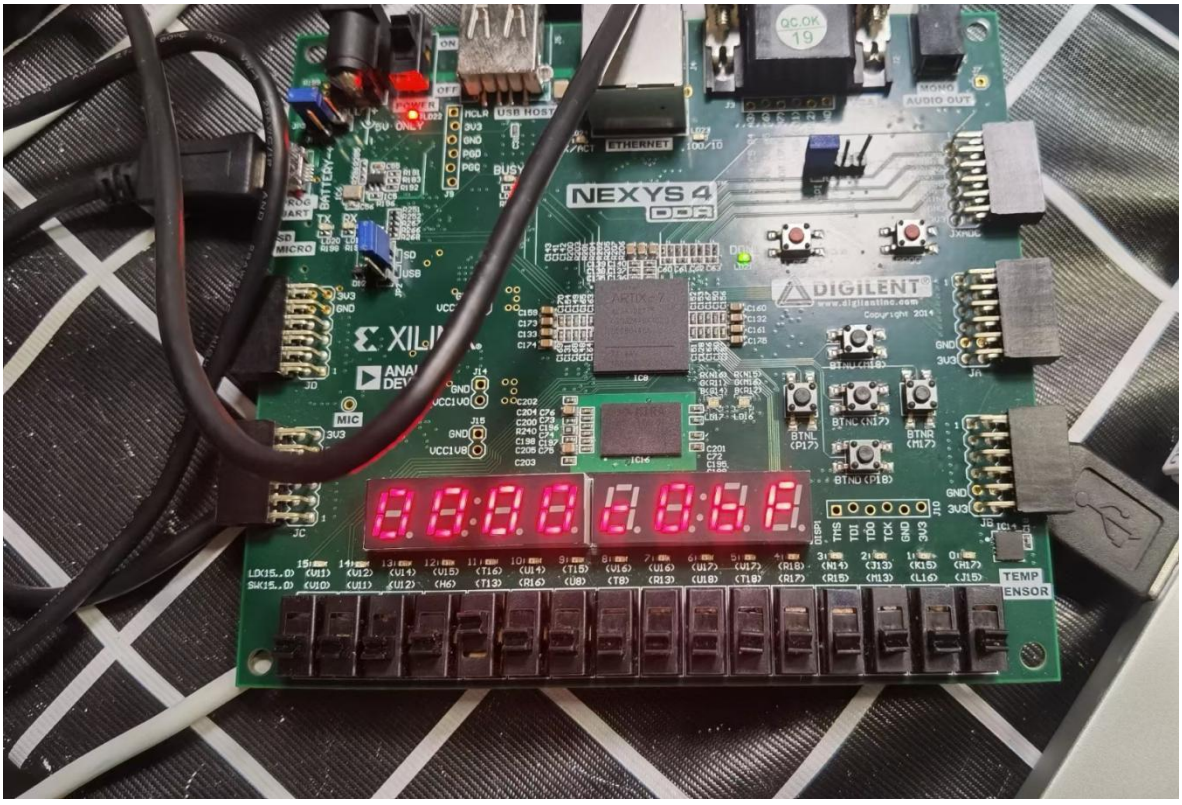

图 3.14 上板后的 CCAB 结束处

4 实验过程与调试

4.1 测试用例和功能测试

对所有电路需要通过汇编测试用例进行测试。

4.1.1 测试用例 1-benchmark

对单周期 CPU 进行基础+CCAB 的检测，需要将 CCAB 的指令写入 benchmark 的程序中，然后再将该程序通过 RARS 进行转换得到 hex 文件写入指令寄存器，运行可得到理应的结果。

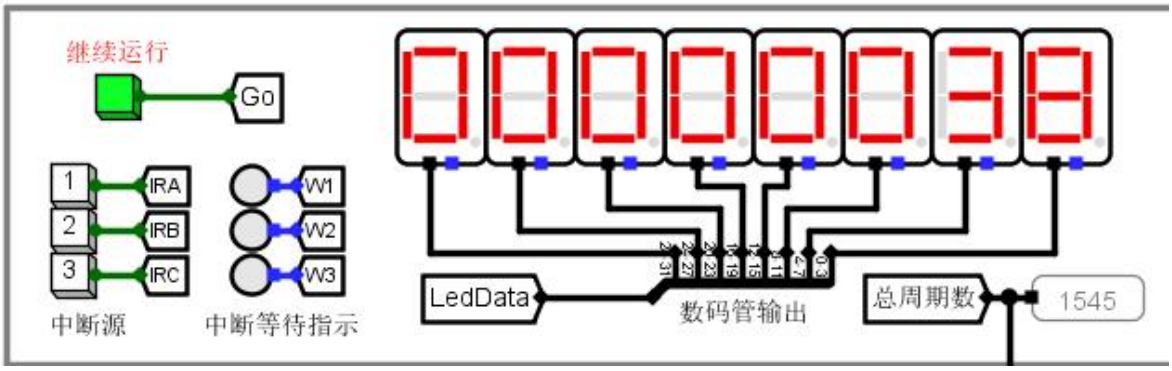

图 4.1 基础程序结束

4.1.2 测试用例 2-流水线与动态分支预测

气泡流水线、重定向流水线、动态分支预测采用 benchmark 作为测试程序。输出的 leddata 结果与单周期 CPU 相同，但周期数有所不同，在 4.2 性能分析中将具体进行分析。

4.1.3 测试用例 3-中断

中断需要根据不同的中断方式采用对应的中断程序来测试。

在中断测试中，需要对其测试 1-3-2-1 等中断源产生顺序来测试中断处理程序执行顺序的正确性，而该测试可以通过 leddata 的显示来对比。

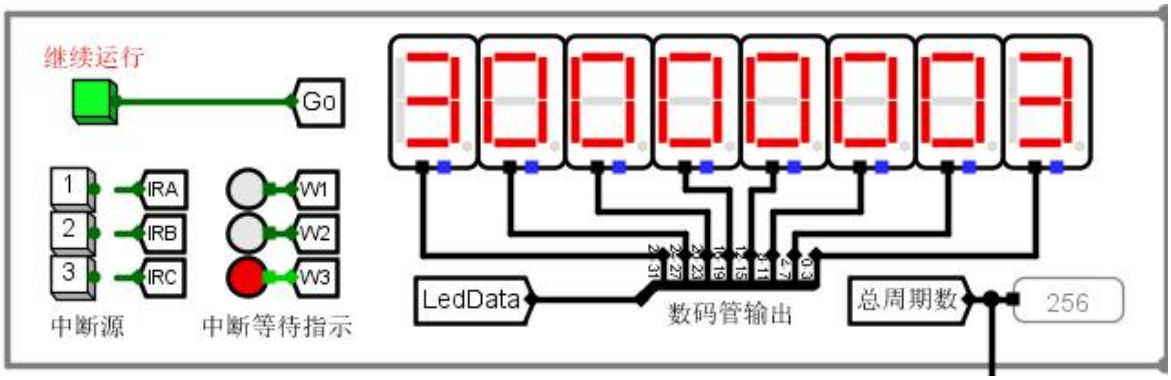

图 4.2 单级中断

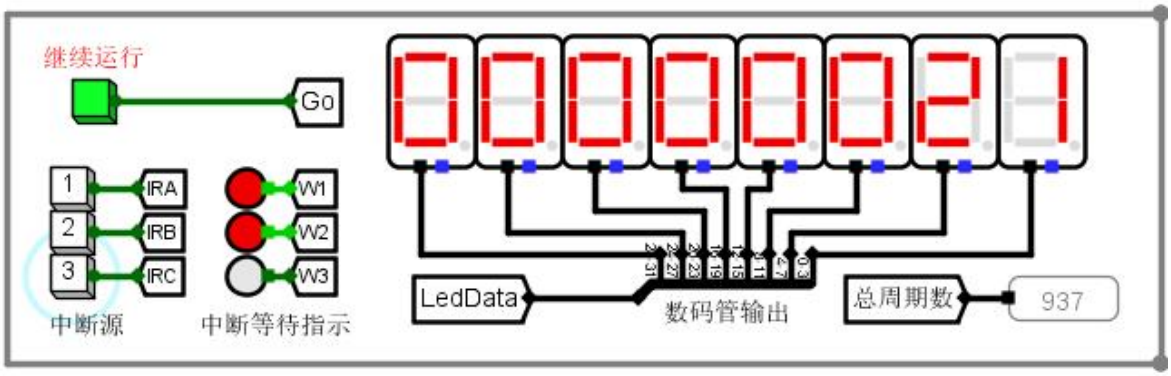

图 4.3 多级中断

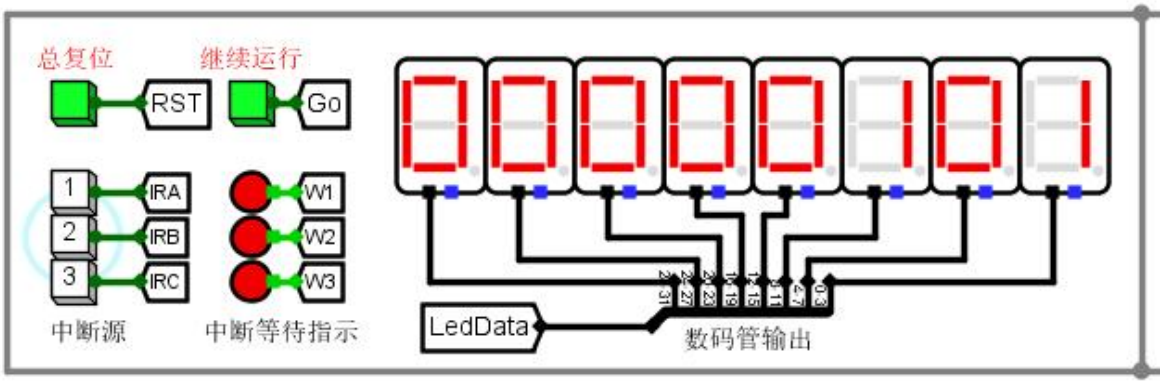

图 4.4 流水线单级中断

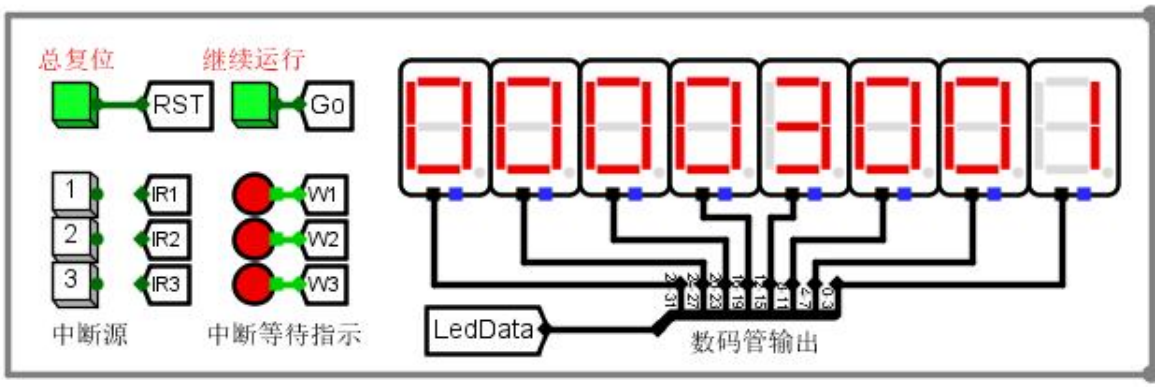

图 4.5 流水线多级中断

4.2 性能分析

流水线将一条指令分成五个阶段来实现，因此实际中流水线的单个周期的时间长度要比单周期 CPU 短得多，因此虽然所需要的周期数流水线要大于单周期 CPU，但所需的执行时间仍更短。

气泡流水线由于与冲突和分支都采用增加气泡的方式进行，因此需要增加气泡的周期数，所需要的周期数最多，重定向解决了数据冲突问题，在寄存器取数冲突时可以不增加气泡，因此周期数相较于气泡流水线更少，分支预测在重定向的基础上减少了分支跳转需要清空而造成和气泡一样效果的周期数，因此所需周期数更少。

综上所述，重定向+动态分支预测技术的性能最好。

表 4. 1benchmark 执行周期对比

	单周期 CPU	气泡流水线	重定向流水线	动态分支预测
Benchmark	1545	3623	2297	1786

4.3 主要故障与调试

4.3.1 单周期 CPU 中的故障

CPU 运行 benchmark 时的周期数不正确。

故障现象：单周期 cpu 在运行基础代码时总周期数不是 1546，而是 1659。

原因分析：开始时采用 halt 的非和 clk 的与信号得到计数器的时钟端信号，ecall 信号的产生和 a7 与 34 的比较之间时有延迟的，因此 halt 会出现信号为 1 的毛刺，在 clk 为 1 的半周期里面会进行一次额外的计数，导致计数增加。本地出错而

educoder 上通过的原因是 educoder 采用的自己的周期计数方法，因此 clk 并未发生错误，其他的电路皆为正确的。

解决方案：将 halt 的非接入使能端而非原本的与门接入时钟端。

4.3.2 流水线中的故障

4. 理想流水线 PC 值不对的问题。

故障现象：IF 阶段的 PC 值为下一条要执行的指令的地址，该指令地址在 EX 阶段选择下一条指令之后得到的指令地址和理应执行的指令地址不同。

原因分析：pc+4 需要在 IF 阶段完成，其他的跳转指令改变的 pc 需要在 EX 阶段完成。保证每一个阶段都有一条指令在执行，否则的话在每五个周期中有三个周期的 IF/ID 两个阶段的指令是不变的，因此需要先把 pc+4 传进去，保证每个周期都有一个 pc 值传入 pc 寄存器中。

解决方案：将 IF 阶段 PC+4 的值传递给下一阶段改为直接在 IF 选择下一条指令地址。

5. 理想流水线中 halt 位置不正确

故障现象：在执行完基础程序之后程序不会暂停而是直接执行后面的 CCAB 指令。

原因分析：应该在 WB 阶段将 halt 的值传递给 IF 阶段的 halt，因为需要用 halt 暂停整个流水线，所以 halt 应该在最后一个阶段的位置，否则 halt 后面的阶段在 halt 信号传递给 IF 阶段的时候不会暂停，只有 halt 前面的阶段才会暂停。

解决方案：将 WB 阶段的 halt 的值传递给 IF 阶段的 halt。

6. 气泡流水线中的增加气泡所需信号问题

故障现象：在向流水接口部件传递了清零信号之后所执行的指令出错。

原因分析：不能使用置零位直接将其置零，因为这个是电平敏感的异步清零信号，需要使用多路选择器来选择输入的信号和 0 来进行同步清零。

解决方案：增加一个信号用来选择 0 和上一条指令来增加气泡，这样只有在 CLK 信号传递进去才会同步赋值。

7. 气泡流水线的停止问题

故障现象：气泡流水线的总周期在 3623 的时候并未停止

原因分析：在 ecall 指令后面还有指令，ecall 在 EX 阶段同时满足 a7==34 的时

候会使得 halt 为 1，此时 EX 阶段的 halt 为 1 ，但是前面的 IF、ID 阶段还存在指令，此时设置的流水寄存器的使能端都是常量，因此 ecall 在传入下一阶段的时候 ecall 后面的指令就会向后传，这样的话 ID、IF 阶段向后传了指令以后 halt 会再次变为 0，EX-MEM-WB 阶段的 halt 由 100 变为 010 变为 001 变为 000，使得 halt 在 3623 时候为 1 之后又变成了 0，从了 pc 只暂定了一个周期就继续再运行了。

解决方案：将 halt 信号与流水接口部件的使能端相联系起来，在 halt 为 1 以后流水接口部件不再向后传，这样就会停止。

4.3.3 单周期 CPU 中断中的故障

1. 单周期 CPU 单级中断的中断请求问题

故障现象：单周期 CPU 单级中断时的中断请求不对

原因分析：清除中断请求信号时按照优先级进行清除的。单级中断指的是无法嵌套，但是会响应所有的中断请求，因此在执行过程中，需要记录所有的中断请求，此时先考虑时间先后上的请求信号，在同一时间的中断请求再考虑优先级，那么在中断请求信号清除的时候也是同样的要求，因此需要记录当下正在执行的中断请求的信号，在后续清楚的过程中应该按照该信号进行清除，而非优先级。

解决方案：利用中断响应信号保存当前正在进行中断服务的编号。

2. 单周期 CPU 多级中断的中断使能问题

故障现象：多级中断中高级中断对低级中断进行中断的时候使能位为 0 不能产生响应。

原因分析：多级中断在中断发生后的所要保存的数据保存了以后需要开中断，并且在后面恢复数据的时候要关中断，因此需要在中断执行的过程中保持一段时间的开中断，在单级中断的基础上需要增加一些操作使得中断使能位在这段过程中保持，如果只采用 CSRRCI 和 CSRRSI 这两个信号的号只能维持一个周期，但是需要保持多个周期让中断处理程序执行完。

解决方案：采用寄存器并用 CSR 信号作为使能端，便可以在 CSRRCI 和 CSRRSI 之间保持信号，但是这样的话开中断会有一个周期的延迟，因此需要寄存器保存和该信号直接传递两种方式结合采用或门完成。同样的对于关中断也有一个延迟，此时清除中断使能位可以采用与门清空。

4.3.4 流水线中断中的故障

1. 流水线走马灯显示问题

故障现象：.流水线中断中走马灯显示的不是 0XFFFFFFF1 而是 0x00000001

原因分析：跳转的指令地址有问题，导致有些指令没有执行使得最后的数据产生错误。在原本的选择保存的断点地址上，是利用 loaduse 信号进行判断的，但是有一个中断信号在第一个周期就产生了，此时 loaduse 信号为 0 并且 IF、MEM、WB 阶段的指令都尚未传递至该位置，因此就会造成错误判断。

解决方案：采用 EX 阶段产生的信号作为多路选择器的选择位，这样既可以保存原有的功能，因为 IR 被清除了以后信号与要跳转的地址所需要的信号是相同的，还可以解决后面阶段尚未传达指令的问题。

2. 流水线中断跳转地址问题

故障现象：.流水线在中断返回时返回的地址发生错误。

原因分析：在 uret 指令执行的同时给予了中断信号，在原有的电路逻辑中存在的问题为，断点地址的选择只从 ID 的 pc 和 EX 执行后的选择结果来选择了，但是 uret 指令并不会产生一般的跳转指令所产生的信号，并且此时 ID 阶段的值会被清空，因此要保存的断电地址就会发生错误变为 0。

解决方案：保存 IF 的 pc 值作为断点地址，并额外增加 IF 和 ID 的 pc 的选择的电路。

4.3.5 动态分支预测中的故障

1. benchmark 执行周期问题

故障现象：.动态分支预测中周期数并未明显减少

原因分析：第一个问题是解复用器和译码器的问题，在选择位为不确定值时对应的所有输出都是不确定的，这个时候就会造成判断错误，一个重要的错误是写使能受到了影响，导致分支指令并未正常的写入，使得后面的历史预测位始终都是 0，造成了和不使用该技术一样的效果。

第二个问题是异步清零的问题，在清空置换位的时候，采用了异步清零的做法，使得该值在不正确的时候清零，造成了后面写入分支指令时的判断错误，使得频繁被使用的条目被置换，造成周期数极大增加的问题。

解决方案：第一个问题需要增加一个硬件层面的判断条件，即如果输入为不确定值时输出固定 0，第二个问题的解决办法是加一个 D 触发器使其同步。

2. 动态分支预测的指令跳转问题

故障现象：.动态分支预测中 pc 不对。

原因分析：一开始是忘记考虑动态分支预测下的 IF 阶段的 pc+4 并非跳转指令后面的指令，因此在找跳转指令的顺序地址时出错了；另一个是只考虑了在不应该跳转的时候跳转了的错误行为，忘记考虑在应该跳转时没有跳转的错误行为，导致 pc 的下一个值出错了。

解决方案：改变原有的重定向的 pc 选择方法，重新加入 BRANCHADDR 找到下一条指令。

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 Logism 单周期 CPU 故障报告。
第三天	设计流水段间接口部件，实现理想流水线
第四天	学习组成原理的冲突检测机制的原理并初步设计冲突检测逻辑
第五天	完整的实现了冲突检测逻辑并实现气泡流水线
第六天	实现了重定向机制和冲突检测逻辑并实现重定向流水线
第七天	复习组成原理中断的相关知识并初步设计中断相关信号
第八天	完整的设计出中断相关信号和数据通路并实现单级中断
第九天	完整的实现多级中断的相关信号和数据通路并实现多级嵌套中断
第十天	设计流水线中需要的中断机制并实现流水线中断
第十一天	学习动态分支预测技术并实现动态分支预测
第十二天	复习 verilog 的相关知识并初步将 logisim 转化为 verilog
第十三天	实现 verilog 剩余部分的代码并实现单周期上开发板

5 团队任务

5.1 团队任务概述

我们小组的迷宫游戏一共有难易两关。用32*32 的 LED 点阵作为游戏屏幕，游戏开始时显示欢迎界面，之后选择将要闯关的关卡后显示对应的地图。图标从右上角开始移动，走到左下角即游戏成功，屏幕会显示 SUCCESS 字样。屏幕中绿色的为墙壁，无法走通。 该项目需要完成电路 logism 实现和汇编代码实现。我们决定选择这个项目，是因为大家都对这个游戏比较熟悉，游戏本身有趣味性且相对易于实现。我们小组在已经实现的单周期单级中断 CPU 电路图的基础上进行改造，编写汇编程序，最终完成了本次团队任务。

5.2 迷宫原理

5.2.1 电路原理

单周期 CPU 负责控制游戏的总体逻辑。 通过单级中断控制游戏进行，一共六级中断，两个关卡中断，四个移动中断，分别是上下左右。用 32 个 32 位寄存器实现 32*32 的 LED 点阵显示游戏界面。

通过控制指令将地图以及人物的数据写入内存，同时将数据写入 32 个 32 位寄存器中保存，需要更新游戏界面时调用 ecall 指令，将保存到寄存器中的数据更新到控制点阵输出的寄存器中即可。

5.2.2 汇编原理

按照正确的游戏逻辑，编写汇编程序。

首先显示欢迎界面，同时提示现在可以选择 2 个游戏关卡，此时程序进入空操作死循环等待关卡选择；游戏用户选择关卡后，此时进入相应的中断服务程序，然后程序显示相应的关卡的游戏界面，再次进入空操作死循环，等待游戏用户通过键盘进行上下左右的移动操作，进入相应动作的中断服务程序，程序最后检查游戏用户是否到达左下角，如果到达了，程序进入到显示 SUCCESS 游戏界面的程序位置显示 SUCCESS 字样。

5.3 迷宫实现

5.3.1 Logisim 中的实现（xym/smy）

在迷宫的电路实现中，由实现了单级中断的单周期 CPU 来执行指令，在本游戏中，由于有六个交互按键，中断产生逻辑扩展为六个并且将中断返回地址等更新；由地图保存逻辑保存指令执行过程中产生的地图；由展示模块输出地图的值；上述三个模块放在一起封装起来可以生成一个由用户输入、地图二进制输出的 RISCV 模块。

在交互电路中，采用键盘逻辑得到四个输入来控制方块的移动；在获取了输入后，将其转换为中断信号输入到 RISCV 中，得到地图的实时响应值从而显示出来界面，因此流程图如下图所示。

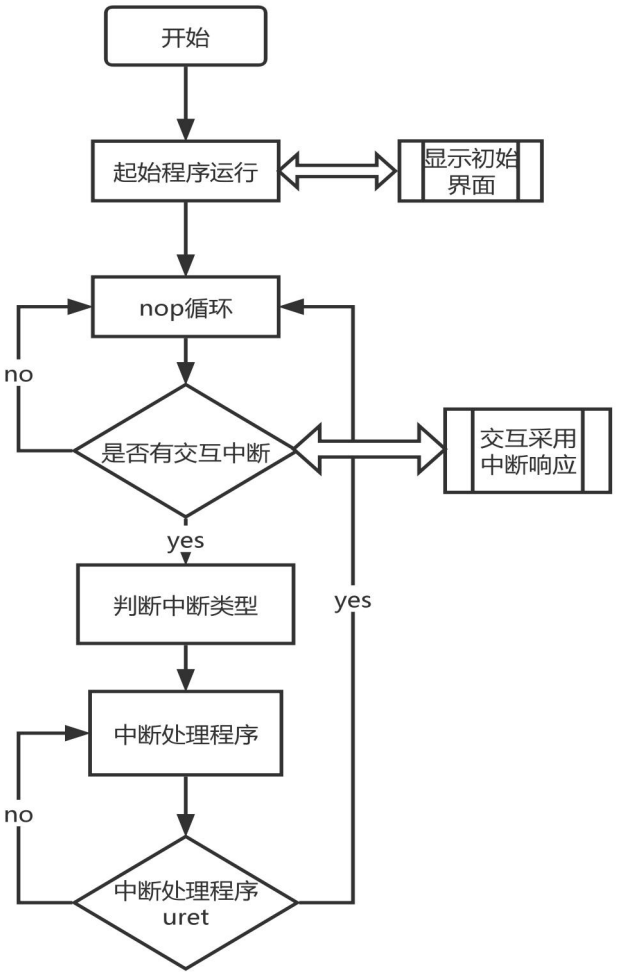

图 5.1 游戏流程图

1. 迷宫的 RISCV 核心

该电路部分采用了个人任务中的单周期 CPU 单级中断的部分。其中整体的数据通路增加了一条新的 CCAB 指令 lui 方便后续的汇编代码实现，其他部分没有改变；中断逻辑部分采用了新的中断源、增加了几个中断信号判断和返回地址判断；输出部分由写入寄存器的地图值作为地图保存模块的输入，因为在汇编部分，地图通过写入数据存储器中得以保存，同时地图保存模块的输入地址也与写入存储器的地址相同；在输出模块，在展示一个地图的整个界面之后，会产生一个 led 信号（由软件实现），通过 led 信号控制地图的 32 个 32 位的值同时输出从而保证地图输出正确，在每移动一次都会产生一副新的地图，因此需要在中断处理程序结束时给出一个信号。

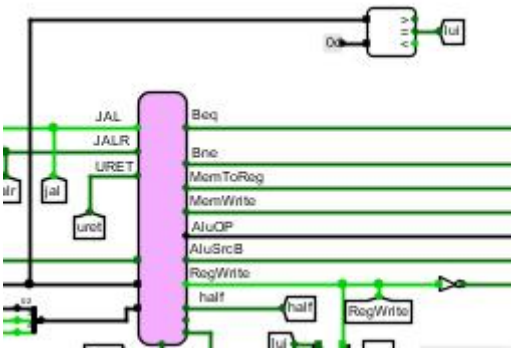

图 5.2 lui 信号的产生

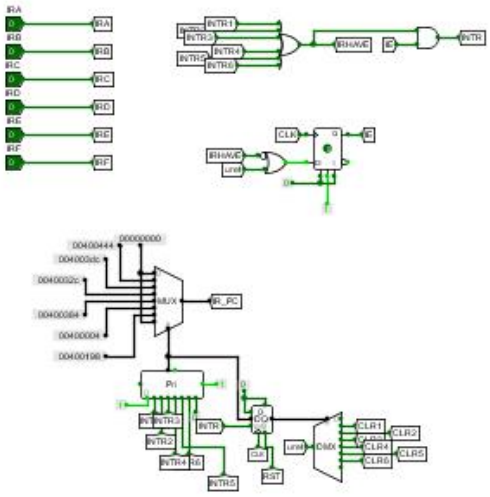

图 5.3 中断逻辑

2. 地图保存和展示

地图保存模块和展示模块就是将地图的 32 个 32 位数据保存并输出，因此这两部分主要由 32 个 32 位寄存器组成。

地图保存模块每次只能保存一个值，因此需要将地址输入给 MAZEMAP（地图保存）模块，在该模块内部，会根据该地址利用优先编码器得到 32 位寄存器的写入使能，在写入使能的控制下，时钟信号将 data 写入对应地址的寄存器中。

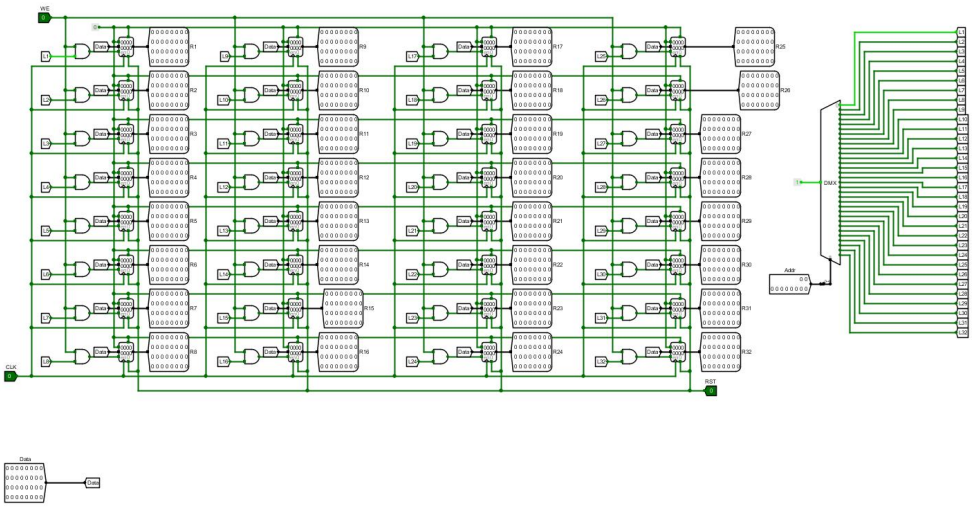

图 5.4 MAZEMAP 模块

展示模块就是将 32 个 32 位值一次性输出，因为 MAZEMAP 模块一次只能写入一个值，因此展示模块需要等 MAZEMAP 将所有值写入以后才可以输出，因此采用了 led 信号输入作为 32 个寄存器的使能端来写入 DISPLAY 模块并输出。

3. 输入逻辑

在本游戏中，游戏关卡直接通过鼠标点击选择，而移动是通过 W、A、S、D 来实现，而该输入并不能直接输入给 RISCV 模块直接进入处理程序，因此需要将该输入转换成中断信号，因此采用比较器来生成上下左右移动的中断信号。

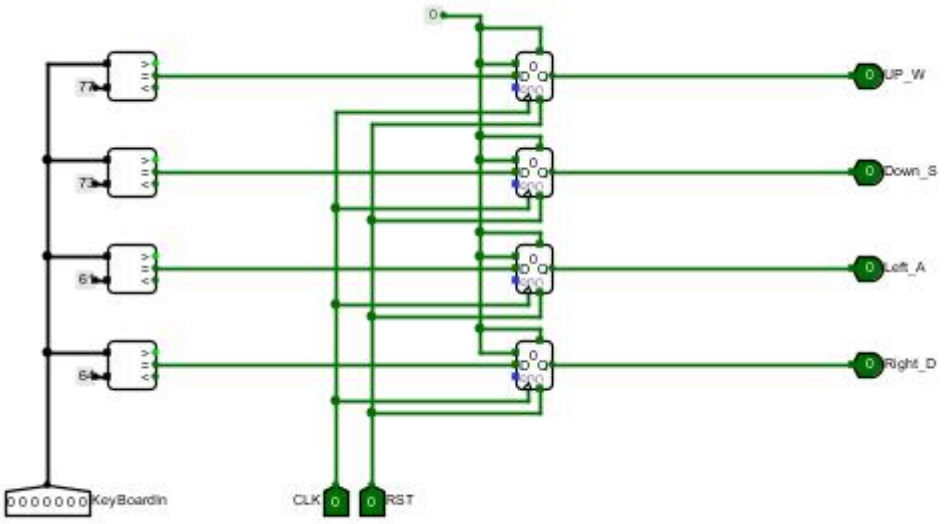

图 5.5 输入判断逻辑

4. 交互界面

在最终的交互界面 PLAYER 中，下面的键盘组件用来接收输入，并将输入的信号传递给键盘判断逻辑并产生 up、down、left、right 四个中断信号对应上下左右四个移动方向，level1 和 level2 信号通过按键产生，这六个中断信号传递给 RISCV 模块并产生每一个操作下的 32*32 地图数据，这些数据传递给 led 点阵产生了地图。

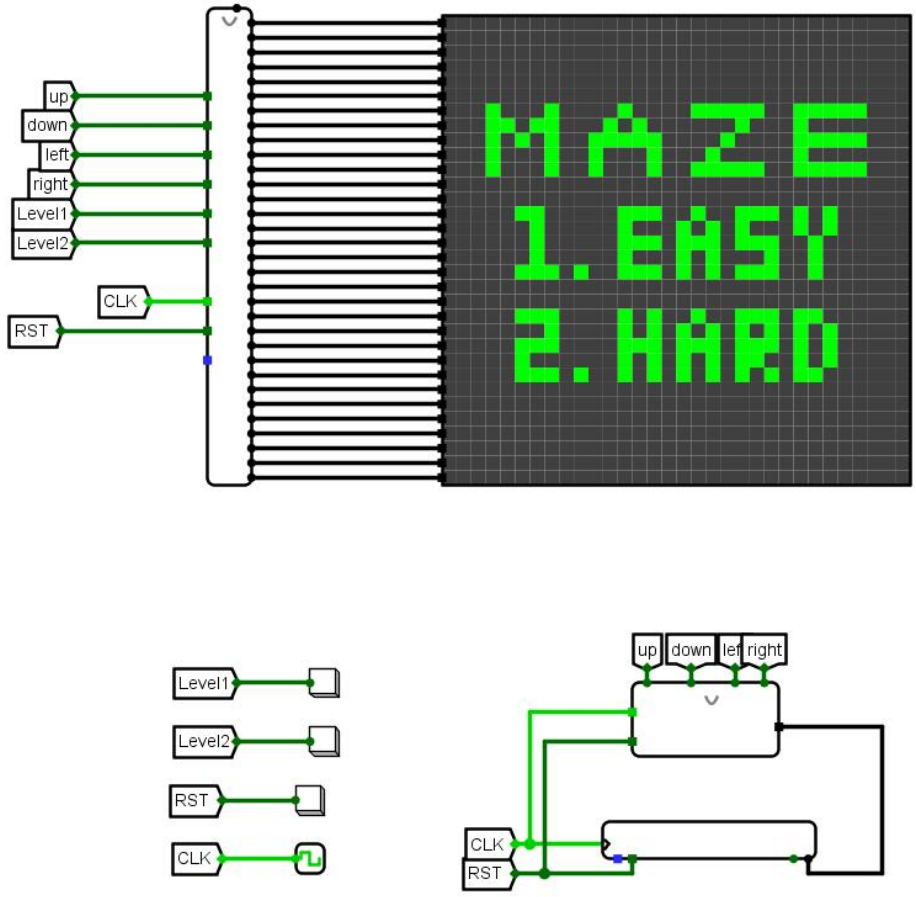

图 5.6　交互界面

5.3.2　汇编实现（zyt/smy）

1. 32*32 的 LED 点阵显示游戏界面

```
li t1, 0x11087cf8
sw t1, 24(zero)
```

图 5.7 li 和 sw 指令

如图 5.6 所示，点阵中每一行亮的位置为 1，暗的位置为 0，每一行对应一个二进制数，通过 li 指令实现给 t1 寄存器赋 32 位二进制的值，sw 指令保存到对应的存储器位置，从而传递给 led 点阵显示游戏界面。

2. 上下左右按键中断服务程序

```
right:
beq s2, zero, step2_closed (1)
addi t2, zero, 0           (2)
beq s0, t2, border         (3)
addi t1, zero, 1           (4)
sub t2, s0, t1             (5)
sll t3, t1, t2             (6)
slli t4, s1, 2             (7)
lw  t5, 0(t4)              (8)
and t6 , t5, t3            (9)
bne t6, zero, wall         (10)
sub s0, s0, t1             (11)
or t5, t5, t3              (12)
slli t3, t3, 1             (13)
xor t5, t5, t3             (14)
sw t5, 0(t4)               (15)
addi a7, zero, 34
ecall
```

图 5.8 中断处理程序

以向右一格为例，s0 代表 x 坐标，s1 代表 y 坐标，首先（1）判断是否选择了关卡，因为只有选择了关卡之后移动才有效。（2）-（3）判断是否到达边界。（4）-（10）判断右侧是否为墙，因为只有不是墙的情况下才能右移。具体地，将下一步要走的位置置 1，其余位置 0，与当前这行按位与，如果为 0 则不是墙，否则就是墙。如果不是墙，则进行（11）-（15），首先更新 x 坐标，将对应的位置置 1，上一步的位置置 0，之后保存新的一行，更新点阵。

向左一格、向上、向下一格同理。

3. 判断是否到达终点

```
addi t2, zero, 31
beq s0, t2, next_judge2
uret
next_judge2:
beq s1, t2, game_success
step2_closed:
uret
```

图 5.9 判断成功的程序

终点在点阵左下角，则判断 x、y 是否等于 31，如果等于则表示到达终点。

5.3.3 Verilog 中的实现（wyx/xym）

开发板上的实现，使用 vivado 编写 Verilog 代码，最终生成比特流烧入板子。Verilog 代码中分为两个主要模块：RISCV CPU 模块和 vga 显示模块。

图 5.10 模块封装

CPU 模块采用“logisim 转 verilog”软件实现，vga 模块代码参考网上的 vga 模块介绍资料并且实现自己的算法实现。

1. CPU 模块的实现

最初，我尝试直接将原电路输出为 Verilog 代码，但是一直无法达到预期的效果。用 Verilog 生成原理图观察，发现中断信号生成与 logisim 中原本的连线相去甚远，连线十分混乱。究其原因，发现转换软件转出的 D 触发器的代码接口很混乱，在连线时接口没有接到对应的位置。对比图如下：

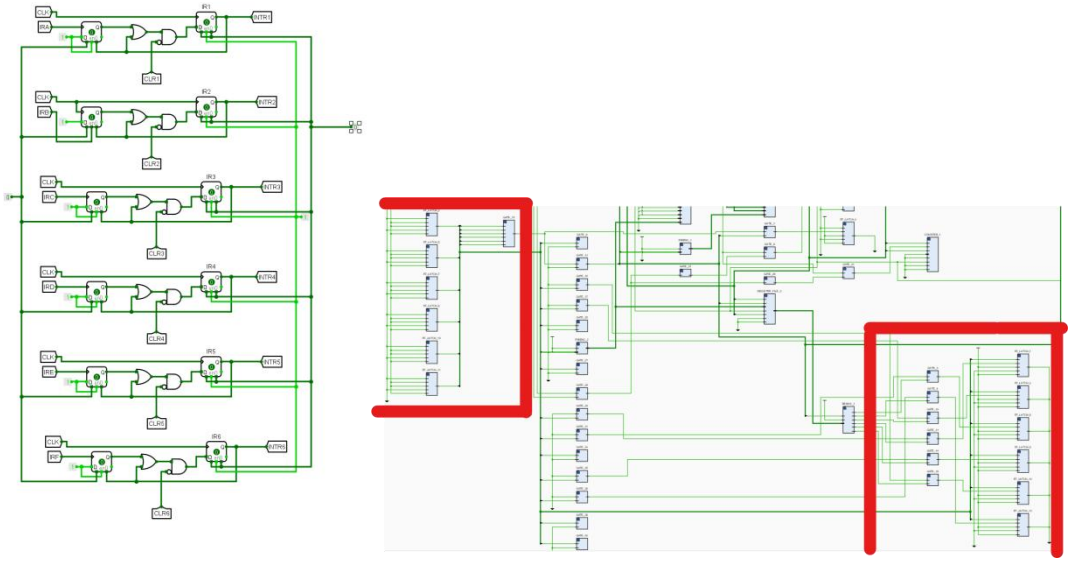

图 5.11 5.12 D 触发器 logisim 与 verilog 对比

然后我尝试将含有 D 触发器的模块（中断信号的生成、中断使能控制）重新封装，防止含有 D 触发器的模块连线混乱导致污染整个电路，然后再更改所封装好的模块中混乱的连线。

经检查原理图，与 logisim 中完全一致。对比图如下：

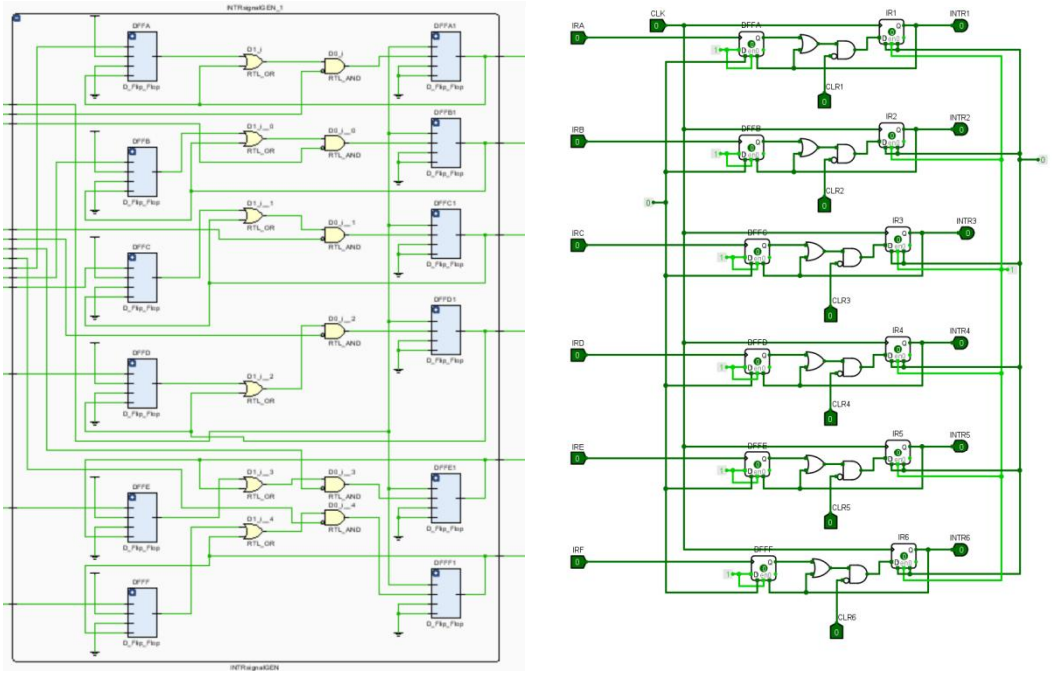

图 5.13 5.14 中断 logisim 与 verilog 对比

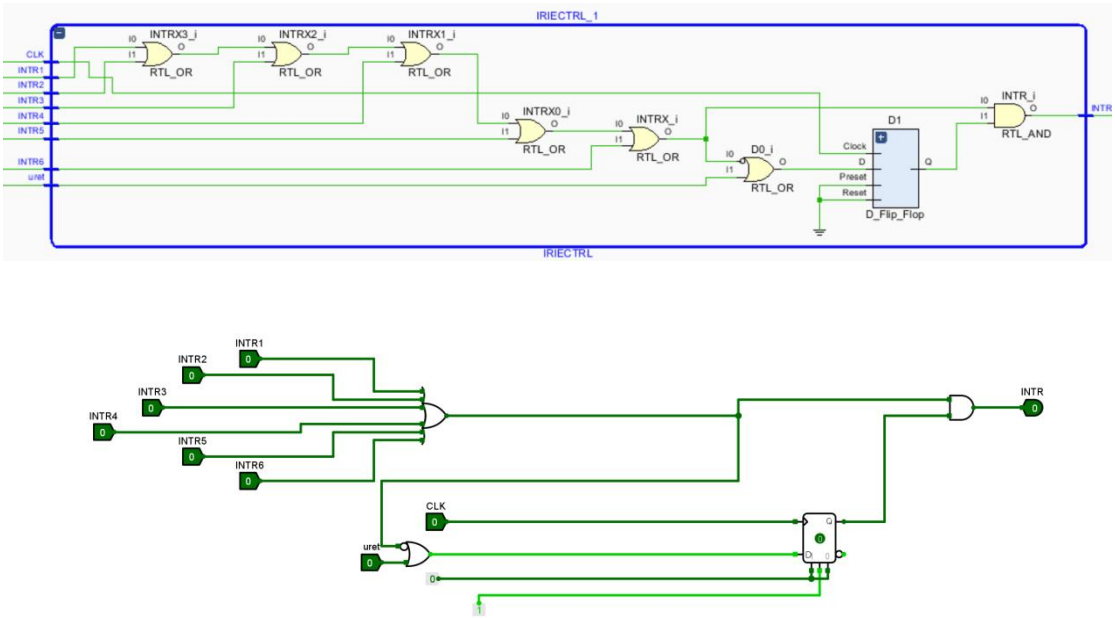

图 5.15 5.16 信号 logisim 与 verilog 对比

至此，CPU 模块的设计基本结束。

2. VGA 模块的实现

查阅相应 VGA 接口资料后，组内课设的算法均由我自己设计实现，因此可能难免会有疏漏之处。以下描述我的实现思路：

①显示区域的划分：

我们组实现一个 32*32 像素块的迷宫，对应的我的思路是用一个[1023：0]的 reg 数组来记录每个区域的位置，代码如下：

```
integer i,j;
always@(posedge clk) begin
    for(i = 0; i < 32; i = i+1) begin
        for(j = 0; j < 32; j = j+1) begin
            dis[i*32+j]=
((xpos>=704-15-j*15)&&(xpos<=704-j*15))&&((ypos>=515-i*15-15)&&(ypos<=515-i
*15));
        end
    end
end
```

dis 数组中每一个元素记录一个像素块的区域信息。测试时，让所有像素块全亮，

测试结果如下：

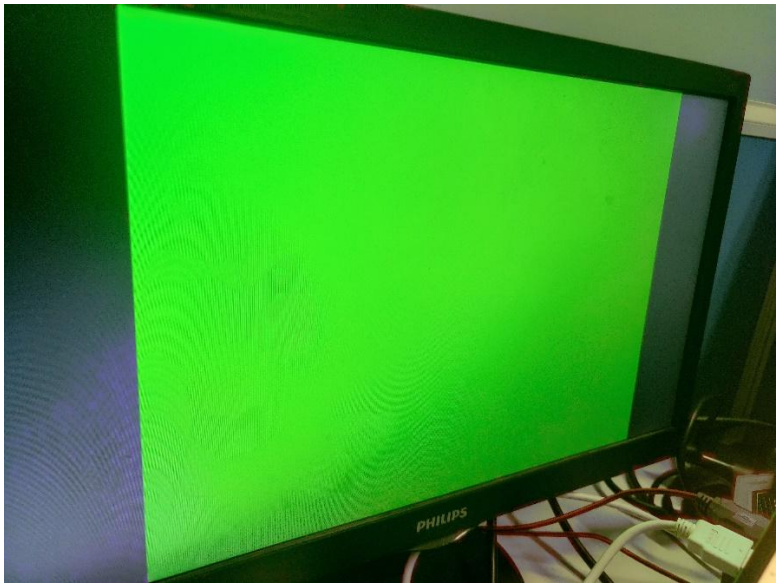

图 5.17 屏幕色块调试

在实际显示的过程中，需要传递一个区块的亮暗信息，对应于 logisim 中使用 32 根[31:0]线路，在此我们从 CPU 传递一个 reg [31:0]R[31:0]数组，即一个 32 位数组，每个元素对应一个 32 位二进制数字，其含义为显示器上该行的亮暗，对应位为 1 代表亮，对应位为 0 代表暗。测试时，先自己定义一个 32*32 的数组，全部赋值 1，任选 2 个元素赋值 0，理论上应该看到屏幕上出现两个暗块。实际效果：

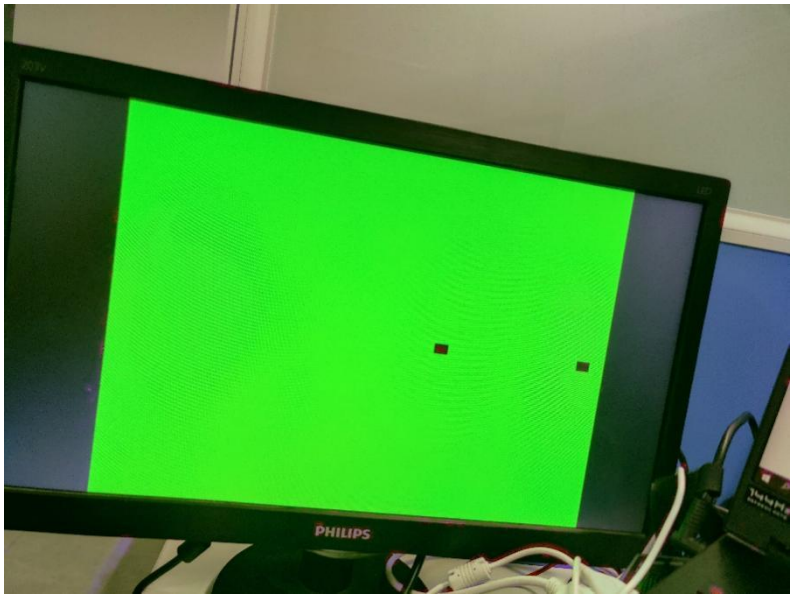

图 5.18 屏幕色块显示

说明已经实现了屏幕显示区域 32*32 的划分！

实际的实现方法更加复杂一些，起初我使用一个标记位 green 来记录此时 vga

扫描到的像素点对应的亮暗，使用循环赋值语句：

```
integer ii;
        always@(posedge clk) begin
            for(ii = 0; ii < 32*32; ii = ii+1) begin
                green = green | (dis[ii] && display[ii]);
            end
        end
```

其中 green 从第一个显示像素块遍历到最后一个显示像素块，持续进行或运算，实现的功能是若有区域需要显示，那么 green 标记为 1，此刻显色。

但是实际操作的过程中，显示器完全黑屏，并没有达到预期的目标。经独立思考，green 作为一个寄存器类型变量，无法对自己进行连续赋值。于是我定义了一个 1025 位的寄存器类型数组 green，每次循环给更高位赋值并将当前位置零（方便下一 clk 的显色，防止当前 clk 的寄存器缓存污染下一时刻）。代码如下：

```
integer ii;
        always@(posedge clk) begin
            for(ii = 0; ii < 32*32; ii = ii+1) begin
                green[ii+1] = green[ii] | (dis[ii] && display[ii]);
//display[ii]为 1 说明此处有像素点需要显示
                green[ii] = 0;
            end
        end
```

再次生成比特流，出现了预期的效果：

图 5.19 结果显示

以下代码实现 rgb 显色，经查阅资料实现，并没有过多的个人设计：

```
assign green[0] = green[32*32];
assign green[1] = green[32*32];
assign green[2] = green[32*32];
assign green[3] = green[32*32];
```

约束文件进行如下绑定：

```
set_property -dict { PACKAGE_PIN C6 IOSTANDARD LVCMOS33 } [get_ports { green[0] }]; #IO_L1P_T0_AD4P_35 Sch=vga_g[0]
set_property -dict { PACKAGE_PIN A5 IOSTANDARD LVCMOS33 } [get_ports { green[1] }]; #IO_L3N_T0_DQS_AD5N_35 Sch=vga_g[1]
set_property -dict { PACKAGE_PIN B6 IOSTANDARD LVCMOS33 } [get_ports { green[2] }]; #IO_L2N_T0_AD12N_35 Sch=vga_g[2]
set_property -dict { PACKAGE_PIN A6 IOSTANDARD LVCMOS33 } [get_ports { green[3] }]; #IO_L3P_T0_DQS_AD5P_35 Sch=vga_g[3]
```

至此，我们已经实现了 VGA 模块实现。

②顶层模块的编写：

顶层模块调用 CPU 模块和 VGA 模块，将 CPU 中的 1024 位显示数据传递到 VGA 模块中，并通过 VGA 模块进行显示，目前实现的效果如下：

图 5.20　上屏成功显示

该界面与 5.3.1 中的 logisim 中的图交互界面 5.6 完全相符。

6 设计总结与心得

6.1 课设总结

本次课设在 RISCV 基础上实现了一个 CPU，做了如下几点工作：

1） 完成方案总结，主要工作如下：

在单周期 CPU 中，首先根据组原课程中学习到的数据通路的知识对 CPU 设计总结了设计流程和思路；在流水线设计过程中，首先对各阶段进行划分，再对冲突信号和逻辑表达式进行原理设计；在中断机制实现过程中，根据课程所学知识对中断流程进行划分并确定每一步的表达式；在动态分支预测中，根据课程所学过的 cache 结构对 BTB 进行设计。在团队任务中，完成了汇编原理设计和实际场景下的 CPU 设计以及 VGA 接口的设计。

2） 完成功能总结，实现成果如下：

首先实现了基本的 24+4 指令的数据通路；完成了基本的流水线数据通路；完成了基于增加气泡的流水线通路，保证程序正确；实现了流水线优化，完成了重定向技术和动态分支预测技术；完成了单周期 CPU 和流水线中的单级中断和多级中断；完成 VERILOG 实现 CPU 并上板成功；完成了团队任务并上屏。

6.2 课设心得

这次的课程设计实验的难度是不小的，在开学的前两周紧赶慢赶的把个人任务全部完成了，在之后的时间又和队友一起做团队任务做了很久，最后终于成功实现了一个小游戏并显示在屏幕上。在这个过程中付出了不少的时间和努力，当然最后所收获的也不少。

在一开始设计 CPU、流水线、中断等数据通路时，因为有 educoder 的帮助，可以获得错误信息，这在一定程度上可以帮助我 debug，所以调试的还算顺利，在获取到错误的指令周期后在 logisim 中在该指令周期附近一步步进行调试找到错误的数值或者信号再向前推，便还算顺利的找到错误并且纠正。

在后续设计流水线中断和动态分支预测等电路时，由于没有 educoder 的提示，所以只能根据测试用例中给出的 leddata 理论输出值和电路中的 leddata 输出值进行比对，

找到不一样的输出的周期附近一步步进行调试。而在动态分支预测这一电路实现过程中，需要对其进行不断的优化来减少 benchmark 的执行周期数，因此要不断地尝试不同的优化思路来对电路进行优化找到最佳的优化思路，我在这次实验中也不断的尝试优化思路，最后将指令的执行周期数减少到 1700 左右。

在本次的实验中，一开始设计和实现这些电路的时候，自己也是犯了不少小错误从而导致结果不正确调试了很久，因此我也长了教训，在之后的实验中一定要先认真阅读官方手册将特殊的容易犯错的地方标注出来，在每次设计的时候先对这些错误进行排查，再去纠正逻辑错误。在整个调试的过程中，我对组原的理论知识得到了巩固，对 CPU 有了更深刻的了解，并且在团队任务中，更切实的感受到了 CPU 在实际应用的调整和设计的不同，加深了我对计算机硬件设计的了解，为之后的课程中打下了基础，尤其是中断机制等，为之后的操作系统等课程中打下了比较好的基础。

然而对于本次课程设计，我还有一些小小的建议和改进。本次课程设计在后续团队任务中需要用到一些 verilog 的知识和屏幕显示的原理等，这些对于我们是比较陌生的，网上所给的资料也并不多，而且网上的资料往往没有系统性和针对性，这在我们前期推进团队任务时是非常困难的，因此课程组或许可以为同学们提供一些相关方面的指导或者比较使用的资料等，带领同学们入门，也会使得同学们锻炼能力的同时完成的更高效。

最后我非常感谢老师和本组所有成员在课程设计中对于我的答疑解惑和帮助。这次的课程设计让我收获颇多，也是我在大学生活中比较精彩的一笔。

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：xym