華中科技大學

2018

计算机组成原理　　课程设计报告

题　　目：	5 段流水 CPU 设计
专　　业：	计算机科学与技术
班　　级：	CS1504
学　　号：	U201514588
姓　　名：	王旭东
电　　话：	1771309472
邮　　件：	370474473@qq.com
完成日期：	2018-04-04 周三下午

计算机科学与技术学院

目 录

1 课程设计概述

1.1 课设目的

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

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

1.2 设计任务

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

1.3 设计要求

（1） 根据课程设计指导书的要求，制定出设计方案；

（2） 分析指令系统格式，指令系统功能。

（3） 根据指令系统构建基本功能部件，主要数据通路。

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

（5） 设计出实现指令功能的硬布线控制器；

（6） 调试、数据分析、验收检查；

（7） 课程设计报告和总结。

1.4 技术指标

（8） 支持表 1.1 前 27 条基本 32 位 MIPS 指令；

（9） 支持教师指定的 4 条扩展指令；

（10）支持多级嵌套中断，利用中断触发扩展指令集测试程序；

（11）支持 5 段流水机制，可处理数据冒险，结构冒险，分支冒险；

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

（13）能运行教师提供的标准测试程序，并自动统计执行周期数

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

表 1.1 指令集

#	指令助记符	简单功能描述	备注
1	ADD	加法	指令格式参考 MIPS32 指令集，最终功能以 MARS 模拟器为准。
2	ADDI	立即数加	
3	ADDIU	无符号立即数加	
4	ADDU	无符号数加	
5	AND	与	
6	ANDI	立即数与	
7	SLL	逻辑左移	
8	SRA	算数右移	
9	SRL	逻辑右移	
10	SUb	减	
11	OR	或	
12	ORI	立即数或	
13	NOR	或非	

#	指令助记符	简单功能描述	备注
14	LW	加载字	
15	SW	存字	
16	BEQ	相等跳转	
17	BNE	不相等跳转	
18	SLT	小于置数	
19	STI	小于立即数置数	
20	SLTU	小于无符号数置数	
21	J	无条件转移	
22	JAL	转移并链接	
23	JR	转移到指定寄存器	If $v0==10 halt(停机指令)
24	SYSCALL	系统调用	else 数码管显示$a0 值
25	MFC0	访问 CP0	中断相关，可简化，选做
26	MTC0	访问 CP0	中断相关，可简化，选做
27	ERET	中断返回	异常返回，选做
28	扩展指令 Xor	异或	
29	扩展指令 Xori	异或立即数	
30	扩展指令 Lhu	加载半字（无符号）	
31	扩展指令 Blez	小于等于 0 时转移	

2 总体方案设计

2.1 单周期 CPU 上板设计

在本次实验中首先需要我们分小组合作，基于上学期第四次实验单周期 CPU，完成其 FPGA 上板开发。本次我们“皮一下就很开心”小组采用的是何同学的单周期 CPU，对于单周期 CPU 的设计（包括扩展指令 CCMB）在组成原理实验中已经做了详细介绍，这里不再赘述。

本次实验主要是将单周期 CPU 通过 Verilog 语言移植到 FPGA 开发板上，最终在开发板上跑通 benchmark 指令。

总体结构图如图 2.1 所示。

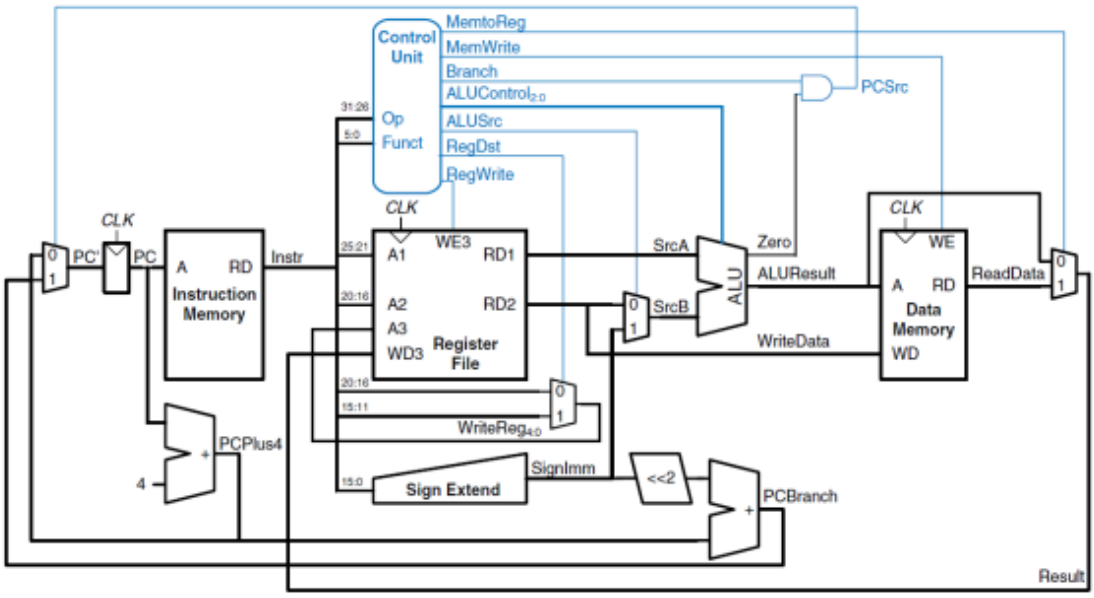

图 2.1 单周期 CPU 总体结构图

2.1.1 主要功能部件

主要功能部件在单周期 CPU 中已有详细描述，在这里需要做的是分别将单周期 CPU 各个部分通过 Verilog 语言实现，logisim 中大部分为组合逻辑、少部分写操作为时序逻辑，存储器在 Verilog 中用数组实现即可。开关和数码管显示部分在之前的数字逻辑课程设计中有所涉及，实现起来并不难。

2.1.2 数据通路的设计

单周期数据通路设计在组成原理实验中已有详细描述。在单周期 FPGA 上板实验中，只需要通过 Verilog 语言将单周期各个部分连通即可。

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，统计得到控制信号。

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。

2.2 中断机制设计

2.2.1 总体设计

计算机系统运行时，若系统外部、内部或当前程序本身出现某种非预期的事件，CPU 将暂时停下当前程序，转向为该事件服务，待事件处理完毕，再恢复执行原来被暂停的程序，这个过程称为中断。产生中断的事件对 CPU 来说是随机发生的，中断技术把有序的程序运行和无序的中断事件统一起来，大大增强了计算机系统的处理能力和灵活性。 中断是现代计算机普遍采用的一项重要技术， 可以实现主机和外设并行工作，以提高了整个系统的工作效率， 可以方便计算机进行程序调试、人机交互、故障处理、实时处理。

根据引起中断的事件来自于 CPU 内部还是 CPU 外部，可分为内中断(也称为软中断)和外中断(也称为硬件中断)；根据进入中断的方式可分为自愿中断和强迫中断；根据其重要性可分为可屏蔽中断和不可屏蔽中断。

本次实验通过软硬件结合的方式实现中断机制，具体流程见图 2-3。

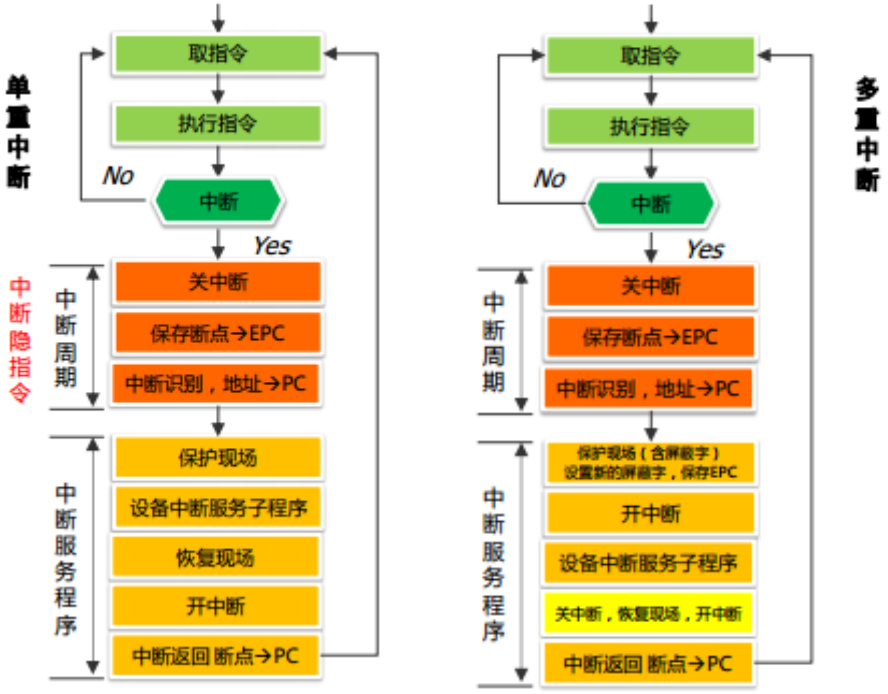

图 2.2 单级/多级嵌套中断流程图

2.2.2 硬件设计

（1）生成中断信号。

本次实验需要实现的有 3 个中断信号，分别编号为 1 号中断，2 号中断，3 号中断。多级中断优先级为：3>2>1。3 号中断具有最高优先级。当执行主程序时具有最低优先级，级数为 0 级。3 个中断信号分别对应不同的中断子程序。中断信号的产生使用的是中断信号产生电路，中断信号产生电路使用的是老师给的参考资料的电路。中断信号发生电路具有统一的时钟断和不同的清零端，中断信号使用上升沿触发。

（2）中断信号筛选。

中断信号筛选电路需要实现中断信号的筛选，单级中断中按照“先来后到”原则，依次实现多个中断；多级中断中优先级高的中断来临的时候转向去执行优先级更高的中断，如果有优先级低的中断来临着将中断信号压栈，如果是同级中断，则忽略该中断。中断筛选电路的实现需要使用当前的中断优先级以及新中断的优先级还要使用到优先编码器，通过有线编码器决定中断信号的选择中断信号。

（3）PC 值的转变。

在中断实验部分，转向到中断程序的 PC 值的过程是由硬件实现的。通过执行中断返回指令 ERET 指令，在物理电路中设计一个由 ERET 信号控制的控制信号，当执

行 ERET 指令时，跳转到对应的 PC 值，这个 PC 值可能是优先级低的中断或者是主程序。

2.2.3 软件设计

（1）开关中断。

本次实验的开关中断是使用软件来实现，使用 mfc0 和 mtc0 实现中断的开关，但同时在硬件部分需要使用一个使能控制信号来控制中断信号这个使能信号需要结合软件部分进行使用。

（2）保护现场。

当进入中断程序之前，需要进行现场的保存，如果不保护现场那么返回后以前的值没有了，也就不能恢复当断点的状态。现场保存主要是将各个寄存器的值进行压栈处理，数据堆栈通常放在数据存储器中。MIPS 处理器的堆栈是向下生长的，因此 SP 寄存器初始指向存储器的最高地址，每保存一个值，SP 要减 4；出栈时则 SP 加 4。

2.3 流水 CPU 设计

2.3.1 总体设计

在本次实验中，需要完成流水 CPU 的设计。流水 CPU 的设计是建立在单周期 CPU 基础之上的。计算机中的流水线技术是把一个复杂的任务分解为若干个阶段，每个阶段与其它阶段并行运行， 其运行方式和工业流水线十分相似，因此被称为流水线技术。把流水线技术应用于运算的执行过程，就形成了运算操作流水线，如浮点数加法运算可分解为求阶差、对阶、尾数加和规格化 4 个阶段。 把流水线技术应用于指令的解释执行过程，就形成了指令流水线， 如图 2.3 所示， MIPS 指令流水线通常将指令执行过程细分为取指令 IF、指令译码 ID、指令执行 EX、访存 MEM、写回 WB 共五个阶段， 在每个阶段的后面都需要增加一个锁存器（又称为流水接口部件，用于锁存本段的处理完成的数据或结果），以保证该阶段的执行结果给下一个阶段使用。

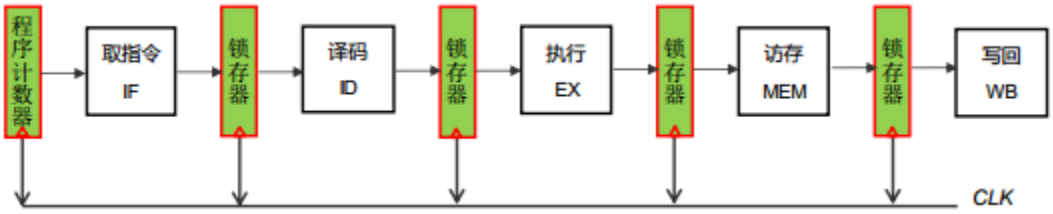

图 2.3 指令流水线逻辑架构

2.3.2 流水接口部件设计

流水接口部件，内容主要是锁存器， 如图 2.4 所示，在单周期 CPU 实现基础上需要增加 IF/ID、 ID/EX、 EX/MEM、 MEM/WB 共 4 个流水接口部件， 4 个流水接口均采用公共时钟进行同步， 流水接口定义尽可能简化， 其内部主要是若干寄存器， 用于锁存段间数据。

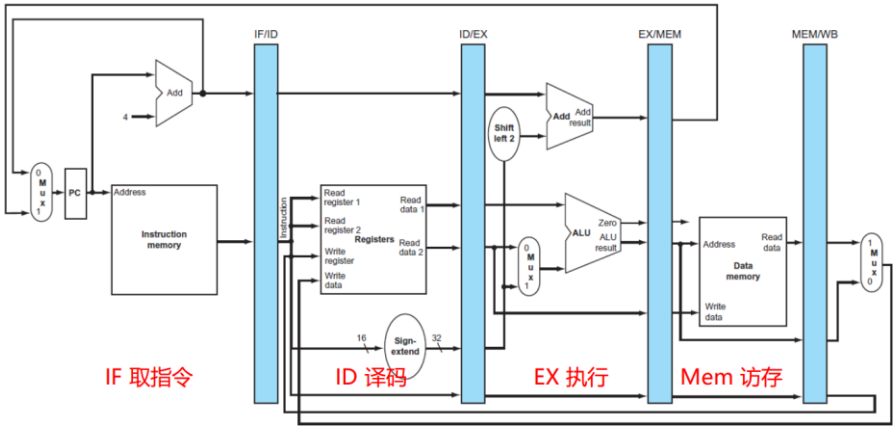

图 2.4 五段流水 MIPS CPU

流水线通过流水接口部件为后段提供数据信息，控制信息， 如图 2.5 所示， 向前段传递反馈信息， 流水线后段对数据的加工处理依赖于前段通过流水接口部件传递过来的信息。 ID 段译码生成该指令的所有控制信号（图中蓝色线为控制信号），控制信号通过锁存器逐段向后传递，后段功能部件所需的控制信号不需要单独生成，直接从锁存器获取。注意单周期 CPU 中的控制器可以在 ID 段直接复用。 不同的流水接口部件锁存的数据和控制信号不同，具体可根据前后段之间的交互信息进行考虑， 以最为复杂的 ID/EX 接口部件为例，该锁存器锁存 ID 段由控制器产生的所有控制信号，同时还需要锁存由取操作数部件取出的寄存器值或立即数，ID/EX 部件设计完成后，其他各段流水接口部件可以直接复制后进行适当精简。

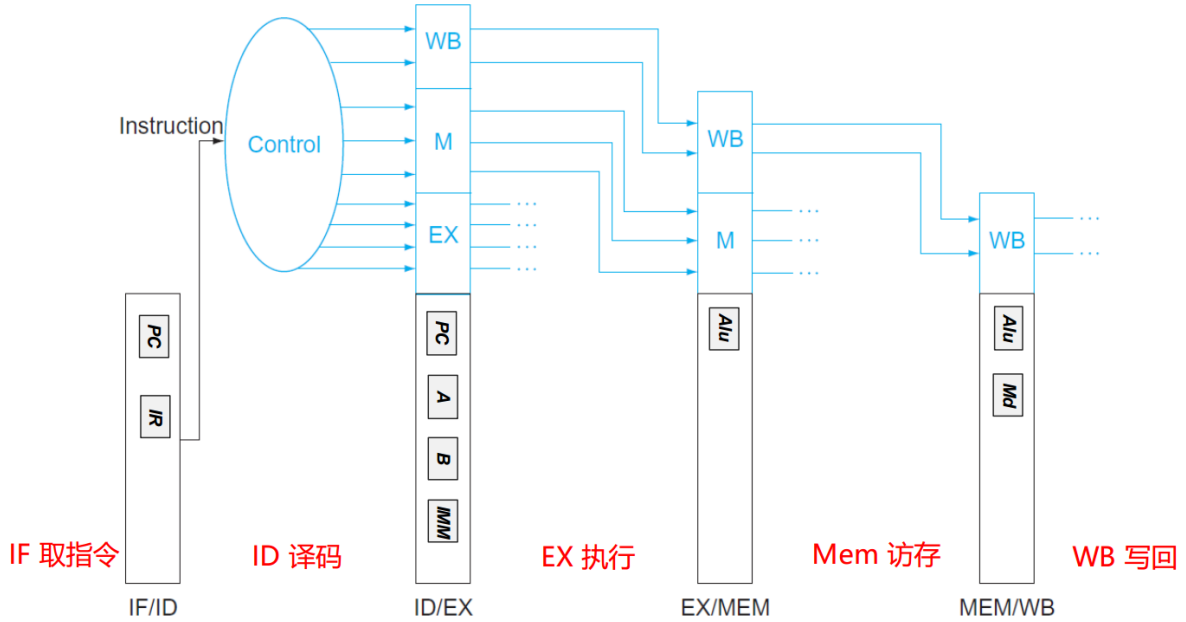

图 2.5 五段流水 MIPS CPU 数据与控制信号传递

2.3.3 理想流水线设计

为了便于表达理想流水线的设计思想，在本次实验中，理想流水线的设计主要涵盖部分简单指令，没有任何冲突。为了完成理想流水线的设计，不得不对每一条指令重新走一遍数据通路，将每条指令的数据在什么阶段使用，需要使用什么样的控制信号必须搞清楚。还未使用的数据和控制信号需要使用该阶段的寄存器组进行缓存，以便在流水线的下一个阶段使用。需要注意的是，理想流水线的测试程序较为精简，因此电路可以大大简化，可去掉很多不必要的信号。

2.4 气泡式流水线设计

理想流水线所有待加工对象均需要通过相同的部件（阶段），不同阶段之间无共享资源，且各段传输延迟一致，进入流水线的对象也不应受其他功能段的影响，但这仅仅适合工业生产流水线，计算机指令流水线存在较多的指令相关，会引起流水线的冲突和停顿。所谓指令相关，是指指令流水线中，如果某指令的某个阶段必须等到它前面的某条指令的某个阶段完成才能开始，也即是两条指令间存在着某种依赖关系，则两条指令存在指令相关。指令相关包括数据相关、结构相关、控制相关，指令相关会导致流水线冲突/冒险（Hazzard）。流水线冲突是指由于指令相关的存在，导致指令流水线出现“断流”或“阻塞”，下一条指令不能在预期的时钟周期加载到流水线中。流水线冲突包括数据冲突、结构冲突、控制冲突。

气泡式流水线，通过“插气泡”的方式解决上述的各种冲突，当指令存在某种相关时，暂停后续指令的执行，同时在正在执行的流水部件接口插入空指令（全 0 指令），等正在执行的指令执行到一定的阶段后再允许新指令进入流水线中，空指令即为“气泡”，这种思想就是“插气泡”。

具体实现时首先添加一个数据相关检测来检测数据冲突，然后对检测到的数据冲突进行处理，添加插入气泡逻辑模块（例如插气泡时需要停 PC、同步清空无效指令或者锁存住有效但不能执行的指令），从而解决数据冲突；紧接着分析结构与控制冲突，并且对每种情况做出相应的处理即可。

2.5 数据转发流水线设计

气泡流水线通过延缓 ID 段取操作数动作的方式解决数据冲突问题，但大量气泡的插入会严重影响流水性的性能，还有一种思路是先不考虑 ID 段所取的操作数是否正确，而是等到 EX 段实际需要使用这些操作数时再考虑正确性问题，如图 1.19 所示，EX 段的 or 指令与 MEM 段，WB 段的两条指令均存在数据相关，此时 EX 段取得的操作数应该是错误的数据，正确的数据分别存放在 EX/MEM 以及 MEM/WB 流水接口部件中，还未来得及写回到寄存器中，此时可以直接将正确数据从对应位置重定向（Forwarding）到运算器 ALU 的操作数端（也称为旁路 Bypass），这样就可以避免插入气泡引起的流水线性能下降，重定向方式可以解决大部分的数据相关问题，可大大优化流水线性能。因此在重定向流水线中需要添加一个重定向机

制，增加 Load_Use 检测模块，当这个信号有效时直接将相应数据送回旁路即可。

3 详细设计与实现

3.1 单周期 CPU 实现

3.1.1 主要功能部件实现

1) 程序计数器（PC）

① 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，将此控制信号通过非门取反之后和时钟相与，当需要进行停机时，Halt 控制信号为 1，经过非门之后为 0，与时钟信号相与，屏蔽时钟信号，使整个电路停机。如图 3.1 所示。

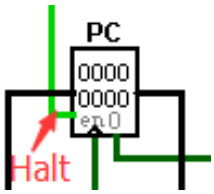

图 3.1 程序计数器（PC）

② FPGA 实现：

程序计数器 PC 的 Verilog 代码如下：

```
module PC(clk,Reset,halt,PCNew,PCout );
    input clk,Reset,halt;
    input [31:0] PCNew;
    output reg [31:0] PCout;

    initial begin
        PCout = 0;
        end
    always @(posedge clk )begin
        if (Reset == 1) begin
            PCout = 0;
```

```
            end
            else if (halt == 1)begin
            PCout =PCout;
        end
        else
        PCout=PCNew;
        end
endmodule
```

2) 指令存储器（IM）

① Logism 实现：

使用一个只读存储器 ROM 实现指令存储器（IM）。设置该只读存储器的地址位宽为 10 位，数据位宽为 32 位。因为 PC 中存储的指令地址有 32 位，而 ROM 地址线宽度有限，仅为 10 位，故将 32 位指令地址高位部分和字节偏移部分直接屏蔽，使用分线器只取 32 位指令地址的 2-11 位作为指令存储器的输入地址。如图 3.2 所示。

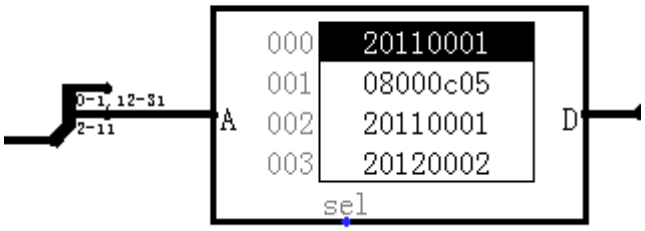

图 3.2 指令存储器（IM）

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：

```
module IM(PCNow,PC);


    input   [9:0]PCNow;
    output reg [31:0] PC;
```

```
        reg [31:0] mem[0:1023];
        initial begin
            $readmemh("F:/benchmark.hex", mem); //  从文件中读取指令二进制代码赋值给 mem
            //$readmemh("F:/yw.hex", mem);
            PC = 0; //  指令初始化
        end

        always @(*) begin
            PC = mem[PCNow];
        end

    endmodule
```

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）寄存器文件（RF）

① Logism 实现：

使用 CS3410 库中的寄存器文件即可，如图 3.3 所示。

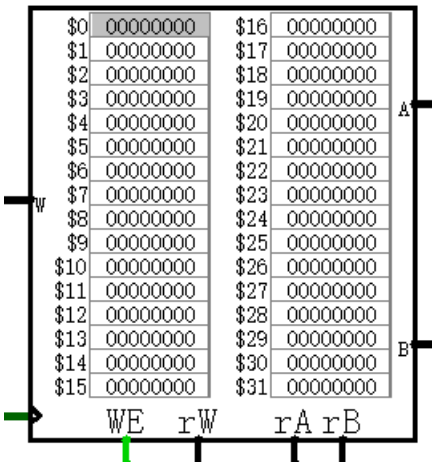

图 3.3 寄存器文件（RF）

② FPGA 实现：

定义一个 32 个数据、每个数据为 32 位的数组，并且提供一个写端口，两个读端口即可。

寄存器文件的 Verilog 代码如下：

```
module regFile(clk, WE, rw, ra, rb, writeData, A, B);
        input clk, WE;
        input [4:0] rw, ra, rb;
        input [31:0] writeData;
        output [31:0] A, B;
        reg [31:0] regMem[0:31];
        integer i;
        initial begin
            for(i = 0; i < 32; i = i + 1)
                regMem[i] = 0;
        end
        assign A = regMem[ra];
        assign B = regMem[rb];
        always @(posedge clk) begin
            if((rw != 0) && (WE == 1))    regMem[rw] = writeData;
        end
endmodule
```

4）数据存储器（DM）

① Logism 实现：

使用一个可读可写存储器 RAM 实现指令存储器（DM）。设置该存储器的地址位宽为 24 位，数据位宽为 32 位。因为 PC 中存储的指令地址有 32 位，而 RAM 地址线宽度有限，为 24 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 2-25 位作为指令存储器的输入地址。如图 3.4 图 3.2 所示。

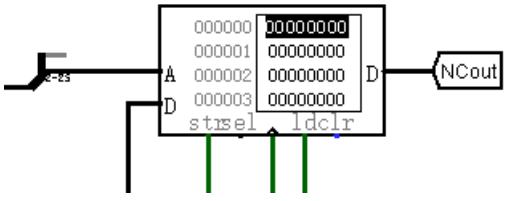

图 3.4 数据存储器（DM）

② FPGA 实现：

使用一个 reg 型的 32 位长的数组来表示指令的存储空间，由于存储单元数越多，下板子速度就越慢，并且本次实验也用不了很多数据存储器单元，因此只分配了 1024 个单元，但这已经足够跑 benchmark 了。数据存储器 DM 的 Verilog 代码如下：

```
module dm(addr, datain, str, clk, ld, clr, dataout, dmData, dmout);
        input [31:0] addr;
        input [31:0] datain;
        input str,clk,ld, clr;
        input [3:0] dmData;
        output    [31:0] dataout, dmout;
        reg [31:0] memory[0:1023];
        integer i;
        wire [9:0] address;
        assign address = addr[11:2];
        assign dmout = memory[dmData];
        initial begin
            for(i = 0; i < 1024; i = i + 1) begin
                memory[i] = 0;
            end
        end
        assign      dataout = memory[address];      //read
        always @(posedge clk) begin       //write
            if(str == 1 && clr == 0)
                  memory[address] = datain;
        end
endmodule
```

3.1.2 数据通路的实现

本次课程设计采用的工程化的设计模式，一次性构建所有的数据通路。主要实现

方法为，对于每一条指令，将其改写成 RTL（Register Transfer Level），忽略控制类信号，仅保留数据类信号，根据 RTL 功能填写对应指令的数据通路表，描述五大部件之间的连接关系，记录各部件输入端数据来源。

根据总体方案设计中数据通路设计那一小节的详细内容，具体分析每一条指令在执行过程中各个主要部件的输入和输出端口的连接，完成指令系统数据通路表的填写，如表 3.1 所示。

表 3.1 指令系统数据通路表

指令类型	指令	PC	IM	RF				z_ext	s_ext	ALU		DM	
				RA#	RB#	RW#	W			X	Y	Addr	Din
R	Add	PC+4	PC	rs	rt	rd	ALURes			[rs]	[rt]		
I	Add Immediate	PC+4	PC	rs		rt	ALURes		imm[15:0]	[rs]	s_ext		
I	Add Immediate Unsigned	PC+4	PC	rs		rt	ALURes		imm[15:0]	[rs]	s_ext		
R	Add Unsigned	PC+4	PC	rs	rt	rd	ALURes			[rs]	[rt]		
R	And	PC+4	PC	rs	rt	rd	ALURes			[rs]	[rt]		
I	And Immediate	PC+4	PC	rs		rt	ALURes	imm[15:0]		[rs]	z_ext		
R	Shift Left Logical	PC+4	PC		rt	rd	ALURes	sa[10:6]		[rt]	z_ext		
R	Shift Right Arithmetic	PC+4	PC		rt	rd	ALURes	sa[10:6]		[rt]	z_ext		
R	Shift Right Logical	PC+4	PC		rt	rd	ALURes	sa[10:6]		[rt]	z_ext		
R	Sub	PC+4	PC	rs	rt	rd	ALURes			[rs]	[rt]		
R	Or	PC+4	PC	rs	rt	rd	ALURes			[rs]	[rt]		
I	Or Immediate	PC+4	PC	rs		rt	ALURes	imm[15:0]		[rs]	z_ext		
R	Nor	PC+4	PC	rs	rt	rd	ALURes			[rs]	[rt]		
I	Load Word	PC+4	PC	base		rt	DM		offset[15:0]	[base]	s_ext	ALURes	
I	Store Word	PC+4	PC	base	rt				offset[15:0]	[base]	s_ext	ALURes	[rt]
I	Branch on Equal	PC+4/ PC+offset	PC	rs	rt					[rs]	[rt]		
I	Branch on Not Equal	PC+4/ PC+offset	PC	rs	rt					[rs]	[rt]		
R	Set Less Than	PC+4	PC	rs	rt	rd	ALURes			[rs]	[rt]		
I	Set Less Than Immediate	PC+4	PC	rs		rt	ALURes		imm[15:0]	[rs]	s_ext		
R	Set Less Than Unsigned	PC+4	PC	rs	rt	rd	ALURes			[rs]	[rt]		
J	Jump	PC31..28||instr_index||00	PC										
J	Jump and Link	PC31..28||instr_index||00	PC			11111	ALURes			PC	1		
	Jump Register	[rs]	PC	rs									
	//syscall(display or exit)	code32	PC	2	4								
I	Exclusive OR Immediate	PC+4	PC	rs		rt	ALURes		imm[15:0]	[rs]	z_ext		
I	Set on Less Than Immediat	PC+4	PC	rs		rt	ALURes	imm[15:0]		[rs]	s_ext		
I	Load Halfword	PC+4	PC	base		rt	DM		offset[15:0]	[base]	s_ext	ALURes	
I	Branch on Less Than or Equ	PC+4/ PC+offset	PC	rs			0/1			[rs]	0		

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。

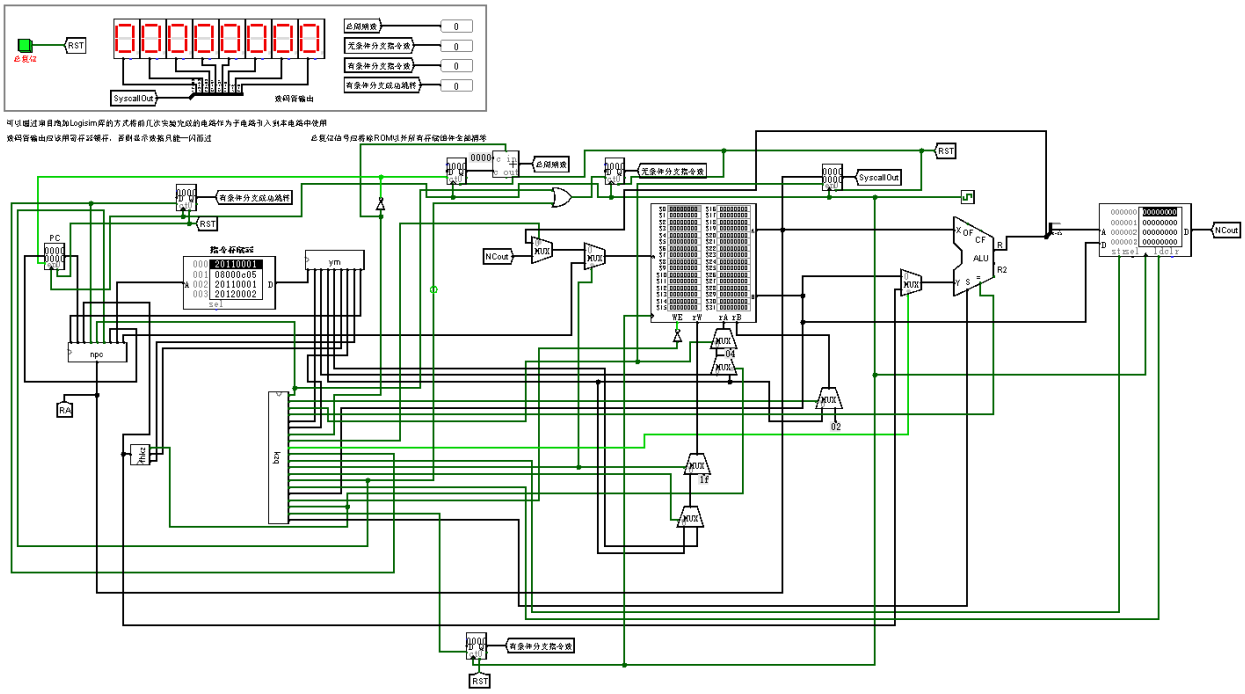

图 3.3 单周期 CPU 数据通路（Logism）

在 Vivado 中使用 Verilog 语言搭建的数据通路的原理图如图 3.46 所示。

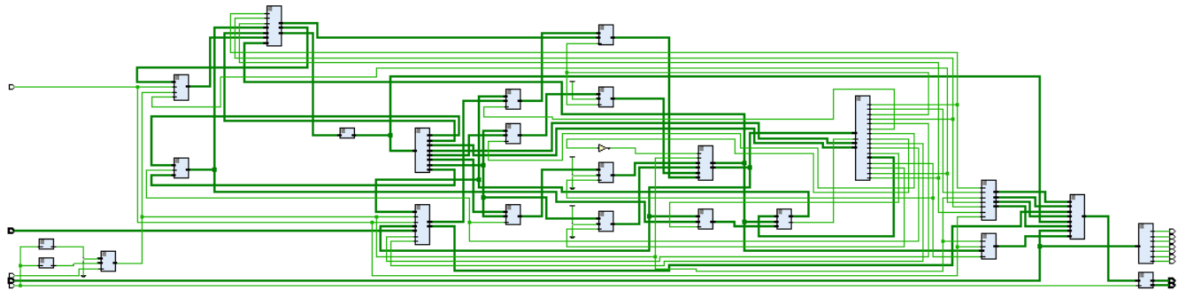

图 3.4 单周期 CPU 数据通路（FPGA）

3.1.3 控制器的实现

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

① FPGA 实现

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

```
module control(op,funct,equal,RB,jump,NCtoREG,
                    YWorNOBranch,Branch,Store,JAL,OPR,JR,Load,
```

```
            RW,YW,Branches,alu_op,dispsrc,disp,halt);
input [5:0] op;
input [5:0] funct;
input equal;
input [31:0] RB;
output   jump,NCtoREG,YWorNOBranch,Branch,Store;
output   JAL,OPR,JR,Load,RW,YW,Branches, dispsrc,disp,halt;
output [3:0] alu_op;
   assign OPR=(op==0)?1:0;
   assign jump=(op==6'b000010||op==6'b000011)?1:0;
   assign JAL=(op==6'b000011)?1:0;
   assign NCtoREG=(op[4:0]==5'b00011)?1:0;
   assign Load=(op==6'b100011)?1:0;
   assign Branches=(op[4:0]==5'b00100||op[4:0]==5'b00101)?1:0;
   assign Store=(op==6'b101011)?1:0;
   assign Branch=((equal&&(op[4:0]==5'b00100))||
                  (~equal&&op[4:0]==5'b00101))?1:0;
   assign YW=(OPR&&(~funct[5:5])&&((funct[4:0]==5'b00000)||
             (funct[4:0]==5'b00010)||(funct[4:0]==5'b00011)))?1:0;
   assign YWorNOBranch=(~(Branches||OPR)||YW);
   assign RW=(Store||(op[4:0]==5'b00010)||Branches||
             (OPR&&(funct[4:0]==5'b01100)))?1:0;
   assign JR=(OPR&&(~funct[5:5])&&(funct[4:0]==5'b01000))?1:0;
   assign dispsrc=(OPR&&(funct[4:0]==5'b01100))?1:0;
   assign halt=(dispsrc&&(RB==10))?1:0;
   assign disp=(dispsrc&&(RB!=10))?1:0;
   assign alu_op=(OPR&&(funct==6'b100000))?4'b0101:
                 (op==6'b010000)?4'b0101:
                 (op==6'b010001)?4'b0101:
                 (OPR&&(funct==6'b100001))?4'b0101:
```

```
                (OPR&&(funct==6'b100100))?4'b0111:
                (op==6'b001100)?4'b0111:
                (OPR&&(funct==6'b000000))?4'b0000:
                (OPR&&(funct==6'b000011))?4'b0001:
                (OPR&&(funct==6'b000010))?4'b0010:
                (OPR&&(funct==6'b100010))?4'b0110:
                (OPR&&(funct==6'b100101))?4'b1000:
                (op==6'b001101)?4'b1000:
                (OPR&&(funct==6'b100111))?4'b1010:
                (op==6'b100011)?4'b0101:
                (op==6'b101011)?4'b0101:
                (OPR&&(funct==6'b101010))?4'b1011:
                (op==6'b001010)?4'b1011:
                (OPR&&(funct==6'b101011))?4'b1100:
                (op==6'b000011)?4'b0101:4'b0101;
              //当不在上述情况时用 0101 作为 default 情况
endmodule
```

在 Vivado 中使用 Verilog 语言构成的主控制器原理图如图 3.57 所示。

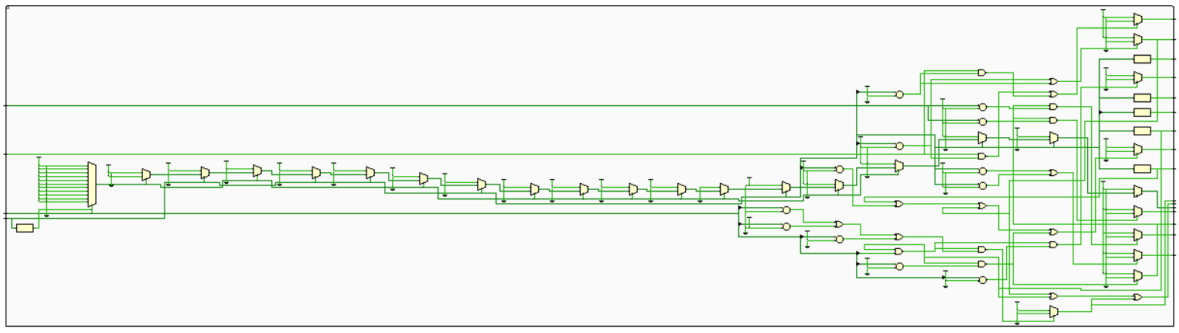

图 3. 5 主控制器原理图

3.2 中断机制实现

3.2.1 单级中断

（1） 硬件实现。需要添加中断识别电路（如图 3.8）、中断采样电路（如图 3.9）、

EPC 寄存器（如图 3.10）。

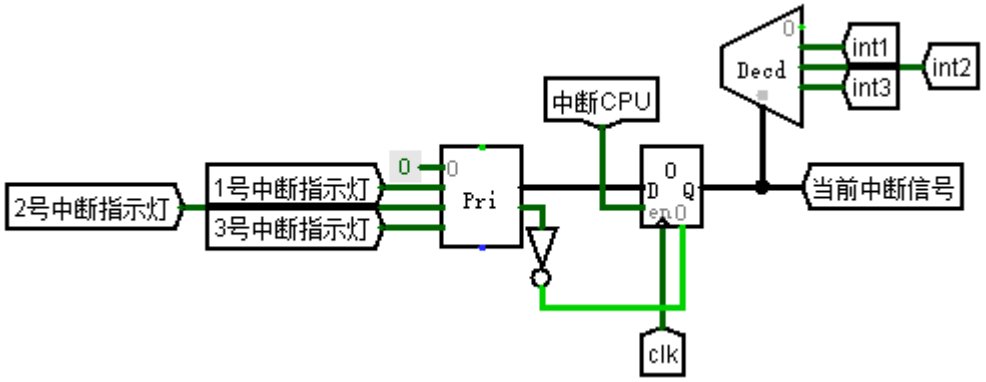

图 3.6 中断识别电路

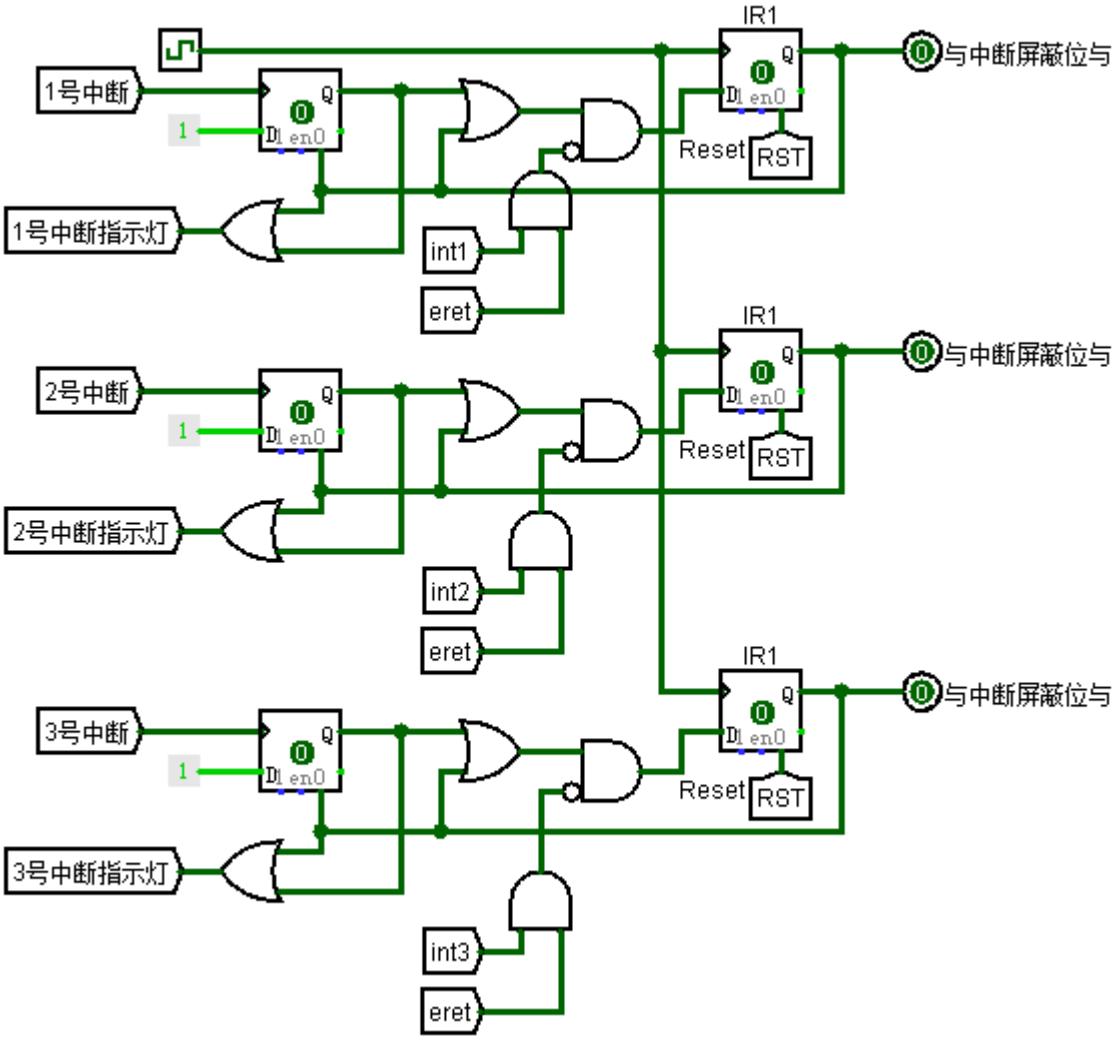

图 3.7 中断采样电路

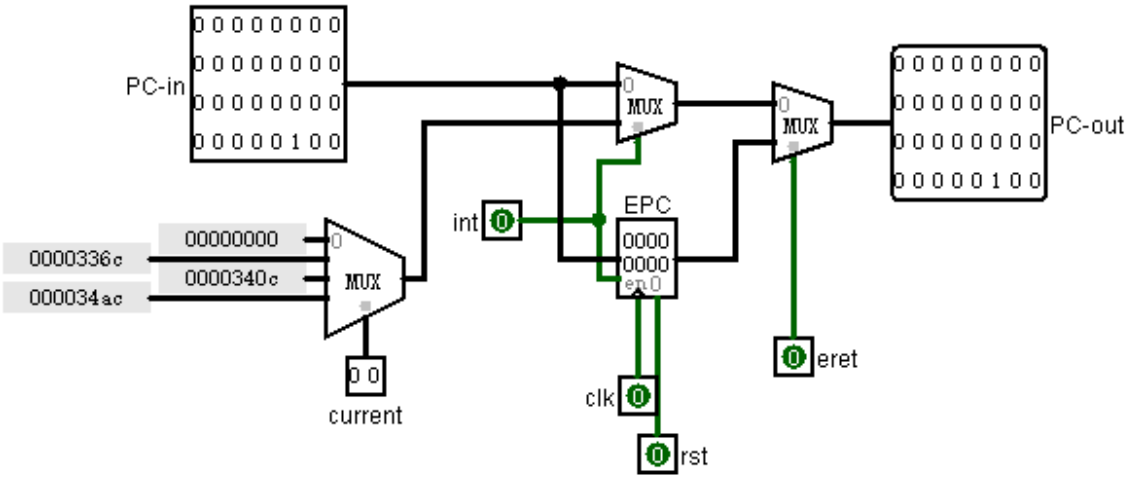

图 3.8 EPC 寄存器

中断采样电路、中断识别电路均有老师提供参考的电路修改而来，其中中断采样电路的作用是获取中断信号的输入，中断识别电路的作用是确定当前程序应该进入那个中断例行程序。

图 3.8 中给出了增加中断后的 PC 及 NPC 逻辑图，由于需要硬件来保存进入中断前时的下一条指令的地址，所以在图中添加了 EPC 寄存器，另外，在图的左下方有一个多路选择器，其中 0x336c、0x340c、0x34ac 是三个中断程序的入口地址，current 信号由中断识别电路给出，用于让程序进入第几号中断。最后添加 eret 指令，稍微修改数据通路即可。

（2） 软件实现

需要编写三个中断程序，代码放在 benchmark 后面，每次进入中断时需要保护现场，退出中断时需要恢复现场，保护现场与恢复现场的代码如下：

```
addiu $sp,$sp,-4          #保护现场
sw $s0,0($sp)
addiu $sp,$sp,-4
sw $s1,0($sp)
addiu $sp,$sp,-4
sw $s2,0($sp)
addiu $sp,$sp,-4
sw $s3,0($sp)
addiu $sp,$sp,-4
```

```
sw $t0,0($sp)
addiu $sp,$sp,-4
sw $t1,0($sp)
addiu $sp,$sp,-4
sw $t2,0($sp)
addiu $sp,$sp,-4
sw $t3,0($sp)
addiu $sp,$sp,-4
sw $a0,0($sp)
addiu $sp,$sp,-4
sw $v0,0($sp)
addiu $sp,$sp,-4
sw $t8,0($sp)

lw $t8,0($sp)          #恢复现场
addiu $sp,$sp,4
lw $v0,0($sp)
addiu $sp,$sp,4
lw $a0,0($sp)
addiu $sp,$sp,4
lw $t3,0($sp)
addiu $sp,$sp,4
lw $t2,0($sp)
addiu $sp,$sp,4
lw $t1,0($sp)
addiu $sp,$sp,4
lw $t0,0($sp)
addiu $sp,$sp,4
lw $s3,0($sp)
addiu $sp,$sp,4
```

```
lw $s2,0($sp)
addiu $sp,$sp,4
lw $s1,0($sp)
addiu $sp,$sp,4
lw $s0,0($sp)
addiu $sp,$sp,4

eret                    # we are back
```

3.2.2 多级中断

（1） 硬件实现

多级中断中允许优先级高的中断打断优先级低的中断，其中断识别电路如图 3.11 所示

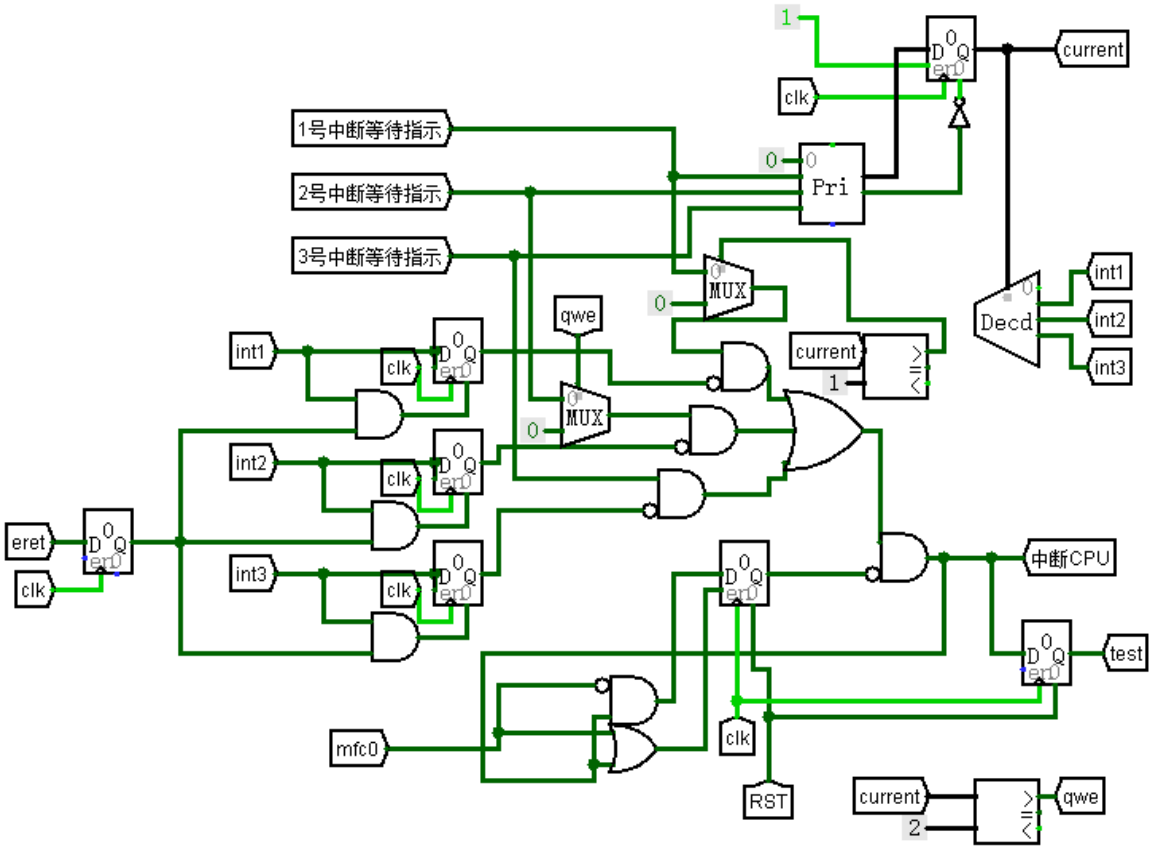

图 3.9 多级中断识别电路

（2） 软件实现

需要编写三个中断程序，代码放在 benchmark 后面，每次进入中断时需要保护现场，退出中断时需要恢复现场，这时候保存(恢复)的现场跟单级中断保存的现场稍稍有点不同，保护现场多了一个 mfc0 操作，恢复现场多了一个 mtc0 操作，保护现场与恢复现场的代码如下：

```
addiu $sp,$sp,-4            #保护现场
sw $s0,0($sp)
addiu $sp,$sp,-4
######################
mfc0 $s0, $0
sw $s0,0($sp)
addiu $sp,$sp,-4
######################
sw $s1,0($sp)
addiu $sp,$sp,-4
sw $s2,0($sp)
addiu $sp,$sp,-4
sw $s3,0($sp)
addiu $sp,$sp,-4
sw $t0,0($sp)
addiu $sp,$sp,-4
sw $t1,0($sp)
addiu $sp,$sp,-4
sw $t2,0($sp)
addiu $sp,$sp,-4
sw $t3,0($sp)
addiu $sp,$sp,-4
sw $a0,0($sp)
addiu $sp,$sp,-4
sw $v0,0($sp)
```

```
addiu $sp,$sp,-4
sw $t8,0($sp)
lw $t8,0($sp)          #恢复现场
addiu $sp,$sp,4
lw $v0,0($sp)
addiu $sp,$sp,4
lw $a0,0($sp)
addiu $sp,$sp,4
lw $t3,0($sp)
addiu $sp,$sp,4
lw $t2,0($sp)
addiu $sp,$sp,4
lw $t1,0($sp)
addiu $sp,$sp,4
lw $t0,0($sp)
addiu $sp,$sp,4
lw $s3,0($sp)
addiu $sp,$sp,4
lw $s2,0($sp)
addiu $sp,$sp,4
lw $s1,0($sp)
addiu $sp,$sp,4
################
lw $s0, 0($sp)
mtc0 $s0, $0
addiu $sp,$sp,4
################
lw $s0,0($sp)
addiu $sp,$sp,4
eret                          # we are back
```

3.2.3 流水中断

流水中断在单级中断与重定向流水线的基础上实现，因此这部分的硬件、汇编代码与单级中断是一样的。

理论上，5 个流水段的任意一个段上都可以处理中断，为了实现的方便通常会选择一个段 来处理中断，可供选择的段常常是 EX 或者是 WB。但是，不管选择那个流水段来处理，都要 做下面的事情（以在 EX 段处理中断为例，其实在 WB 段处理更加方便）：（1）EX 段前面的 2 条指令要继续执行完；（2）EX 段后面进入流水线的 2 条指令要取消；（3）处于 EX 段本身的 这条指令有 2 种选择，如果这条指令允许继续执行完，返回的断点地址应该是该指令的 PC+4； 如果不允许这条指令继续执行，返回的断点就是这条指令的 PC；（通常做法是不允许这条指令 继续执行）；另外如有指令在 EX 段之前完成，比如无条件分支指令在 ID 段完成，此时逻辑又 有区别。（4）暂停流水线进行中断响应；（根据具体的实现方法也可以不暂停）（5）重新启动 流水线从新的 PC 值处取指令（新的 PC 即是中断入口地址）。

3.3 流水 CPU 实现

3.3.1 流水接口部件实现

流水线的接口部件主要是使用一些寄存器组来存放需要使用到的数据和控制信号，主要有 5 个阶段分别是：IF（取址阶段），ID（译码阶段），EX（执行阶段），MEM（访存阶段），WB（回写阶段）。这些寄存器组的数据宽度不一，因为有的是立即数，有的是地址，有的是控制信号，主要有两类，一类是 32 位的数据；一类是 1 位的控制信号。由于要解决指令相关冲突，所以流水接口应实现同步清零。

五段流水线中四个流水接口实现方式基本相似，只需要确定好信号是在哪个阶段产生，在哪个阶段使用，然后通过流水接口传递该信号即可。例如停机信号 EN 是在 ID 段产生的，而在 WB 段使用，所以需要在 ID/EX、EX/MEM、MEM/WB 三个流水接口中分别添加 EN 端口。

在此仅将理想流水接口 ID/EX(最为复杂的一个接口)为例，其他的流水接口都可以参照这个接口，并且理想流水线、气泡流水线、重定向流水线中需要传输的值有个别差异，具体可以仿照这里的 ID/EX 接口，具体 logisim 实现如图 3.12 所示。

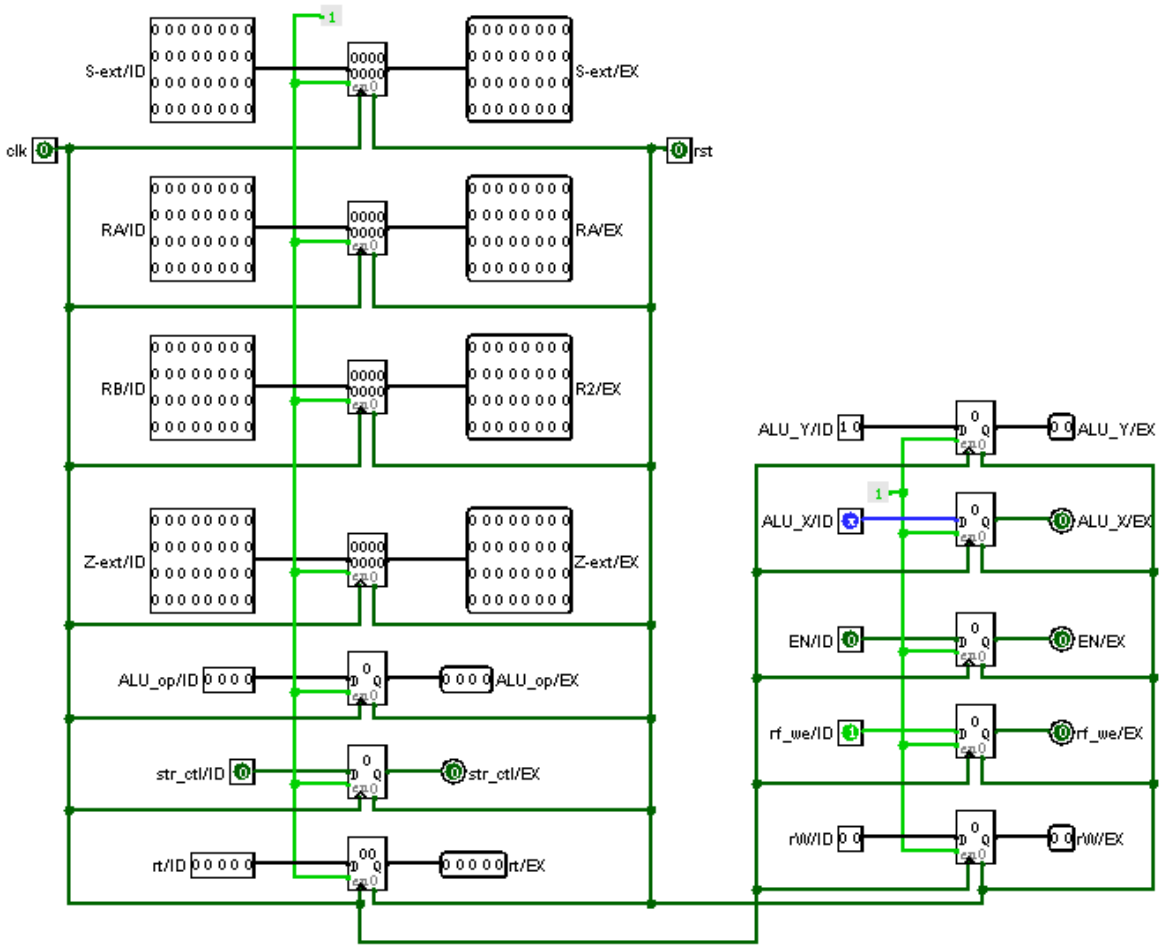

图 3.12 ID/EX 流水接口 logisim 实现

3.3.2 理想流水线实现

理想流水线是最简单的流水线，不用考虑任何的冲突，只需要在原来的单周期 CPU 的基础上将 IF/ID、ID/EX、EX/MEM、MEM/WB 连接起来即可，同时稍微修改 NPC 逻辑即可。

在 logisim 上最终实现的电路图如图 3.103 所示，图中四个流水线接口将指令的执行过程分为五段，因此为 MIPS 五段流水线 CPU。

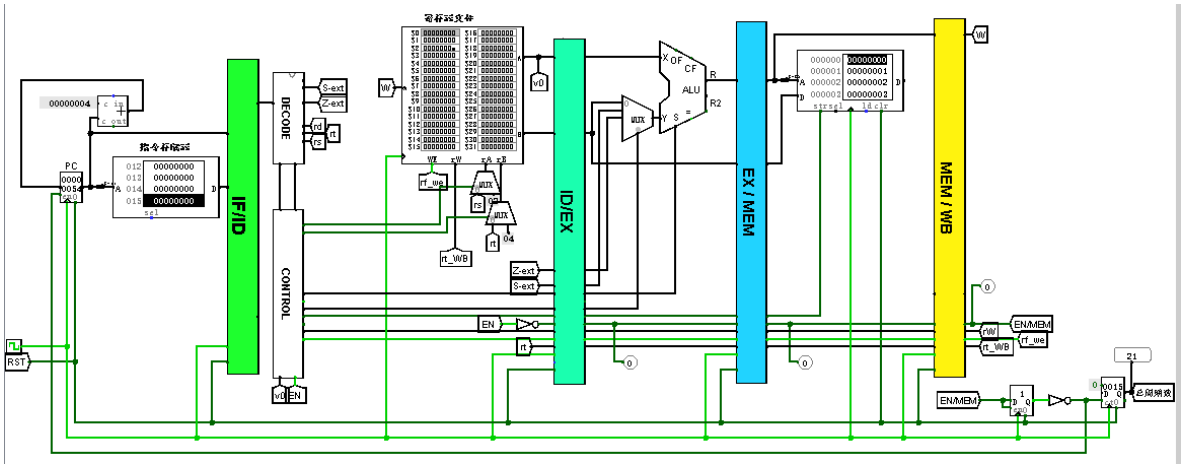

图 3.103 理想流水线实现图

3.4 气泡式流水线实现

进一步改造理想流水线 MIPS CPU，增加数据相关检测逻辑，增加插入气泡逻辑，增加流 水暂停逻辑，增加分支冲突处理逻辑，使得该流水线能处理数据冲突，控制冲突，资源冲突等所有流水冲突。

（1） 数据相关检测。由于译码 ID 段中需要取操作数，所以数据相关检测逻辑应该设置在 ID 段，数据相关的依据是比较当前指令的源操作数是否和后续段的目的操作数是否相同。具体数据相关检测如图 3.114 所示。

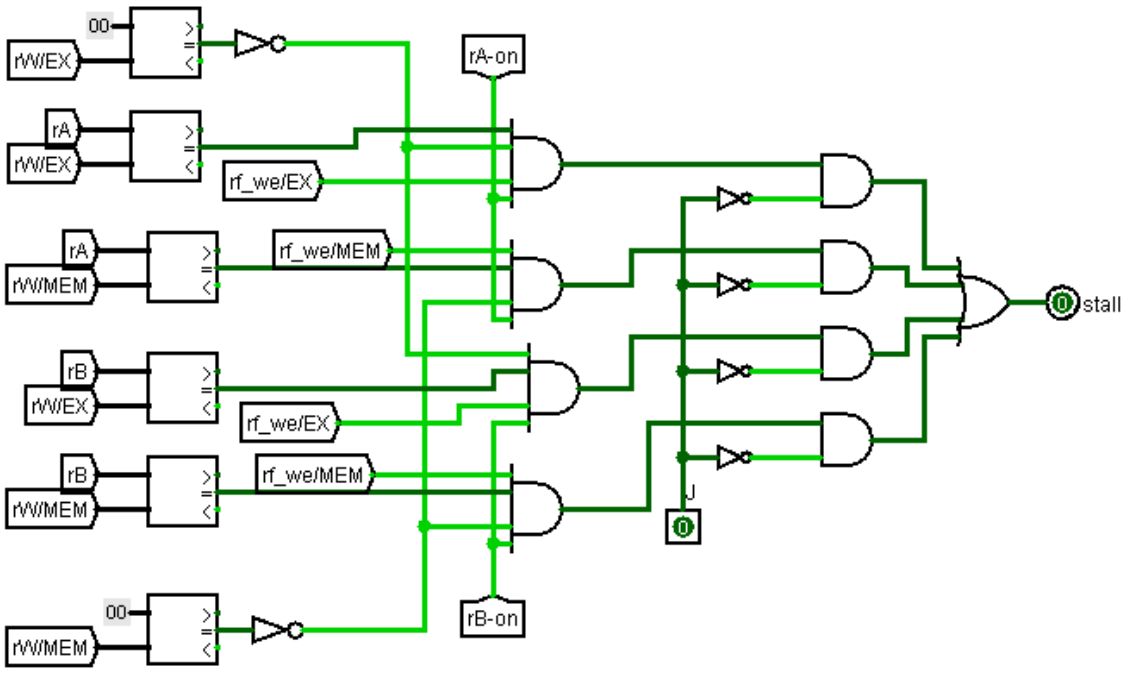

图 3. 114 数据相关检测模块

（2） 数据相关处理。ID 段与写回 WB 段的数据相关可以通过先写后读的方 式解决，具体方式是数据写入寄存器文件时采用下跳沿写入。而 ID 段与 EX 段和 MEM 段的数据相关 则只能通过插入气泡的方式解决。

（3） 插入气泡逻辑。例如当译码 ID 段与执行 EX 段存在数据相关，假设相关检测逻辑已经检测到存在数据相关，应产生 stall（暂停）控制信号，暂停取指 IF 段以及译码 ID 段的工作以延缓取操作数的执行（可以通过控制程序计数器和 IF/ID 流水接口部件的使能端实现,这样时钟到来时锁存器的值不会改变），另外下一个时钟上跳沿到来时，需要向执行 EX 段插入一个气泡 （空指令，MIPS 中空指令是全零的机器码，功能是 0 号寄存器左移零位送入 0 号寄存器），以避免译码 ID 段的指令向后传送。插入气泡可以通过 ID/EX 流水接口部件清零完成，为此需要 为流水接口部件增加清零引脚。

（4）气泡流水线 logisim 电路图如图 3.15 所示

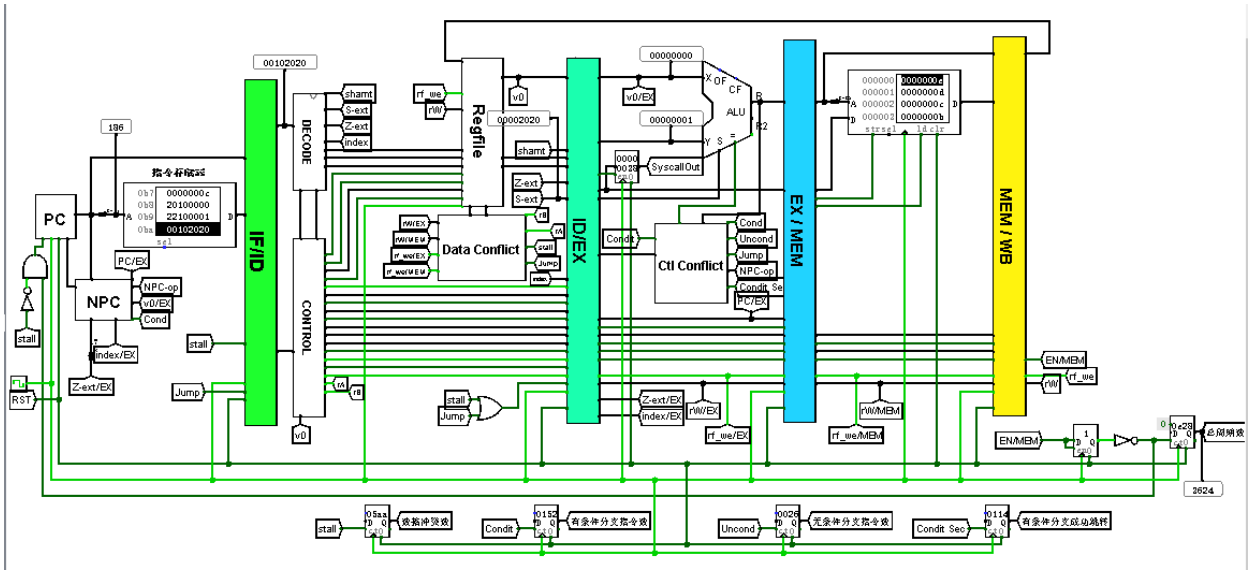

图 3. 125 气泡流水线电路图

3.5 数据转发流水线实现

（1） 数据转发流水线又称重定向流水线，亦称旁路流水线，主要思想就是在数据冲突时需要用到的数据直接从 EX 或 MEM 段直接取回来，而不用等待数据写入寄存器中再取出来，这就更加提高了气泡流水线的效率，这部分电路是基于气泡流水线实现的。

（2） 与气泡流水线不同，当发现 ID 与 EX 之间有数据冲突并且在 EX 段的指令为 LW 时，loadUse 信号有效，此时根据这个信号在 ID/EX 段插入一个气泡，loadUse 信号由 Check 产生的 ex_conf 信号和 EX 段的 lw(加载数据指令)信号相与产生，其他产生冲突的情况只需要在相应数据端口添加多路选择器，并且把数据传回来即可，而不用向气泡流水线那样插入气泡，通过这种方式大大减少了气泡数。

（3） 重定向流水线通路图如图 3.16 所示。

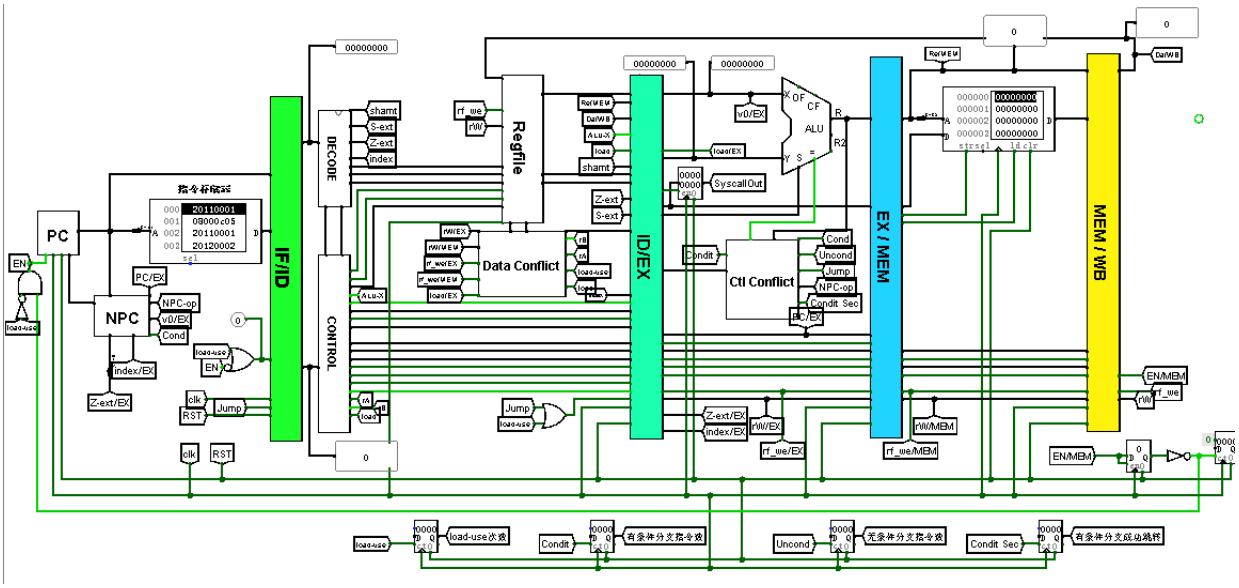

图 3.16 重定向流水线通路图

4 实验过程与调试

4.1 测试用例和功能测试

本次课程设计的测试主要包括以下内容：

- 测试单周期 CPU 上板，观察 vivado 中仿真的波形图；
- 测试理想流水线、气泡流水线、重定向流水线，在 logisim 下观察执行结果；
- 测试单级中断、多级嵌套中断、重定向流水中断，并在 logisim 下观察执行结果；
- 测试重定向流水线上板，观察 vivado 中仿真的波形图。

4.1.1 单周期 CPU 上板测试

通过仿真查看总周期数、无条件分支指令数、有条件分支指令数、有条件分支成功跳转数，以及数据存储器中前十六个单元的值，结果如图 4.1 所示。

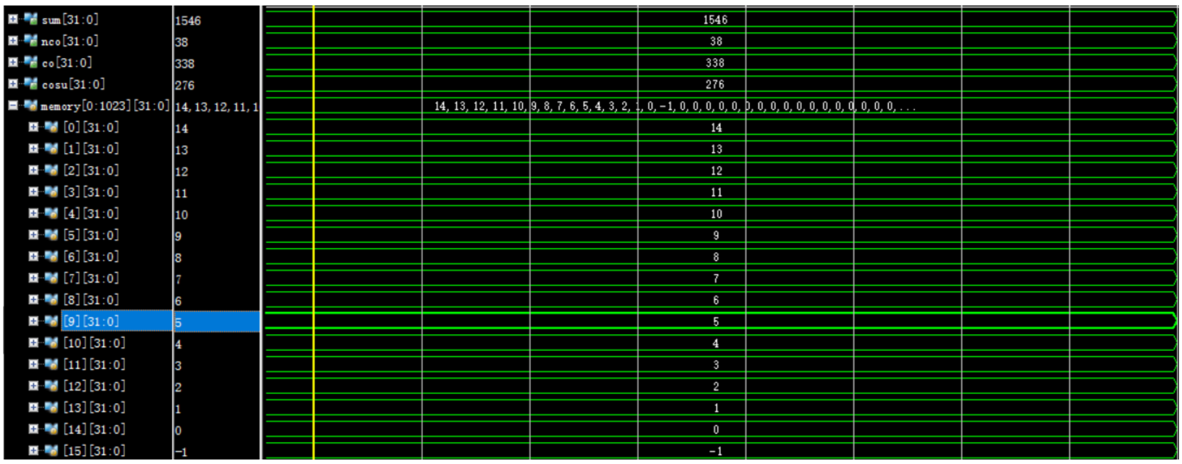

图 4.1 单周期 CPU 仿真

根据图中所示，总周期数、无条件分支指令数、有条件分支指令数、有条件分支成功跳转数结果分别为：1546、38、338、276，并且数据存储器中前 16 号单元的值是 14～-1 的降序排列。

由图可知，仿真结果与预期结果一致。

4.1.2 流水线测试

（1） 理想流水线测试

理想流水线运行结果如图 4.2 所示

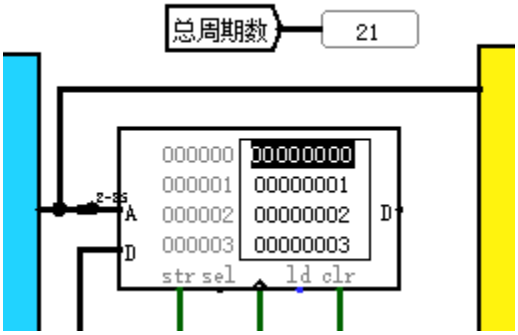

图 4.2 理想流水线测试

由图知，总周期数为 21，数据存储器中写入为 0，1，2，3，结果正确。

（2） 气泡流水线测试

根据气泡流水线公式：总周期数 = 1546 + 1 + 气泡数 + 无条件*2 +有条件成功*2 -1

=1546 + 1 + 1450 + 38*2 + 276*2 – 1

=3624

运行结果如图 4.3 所示

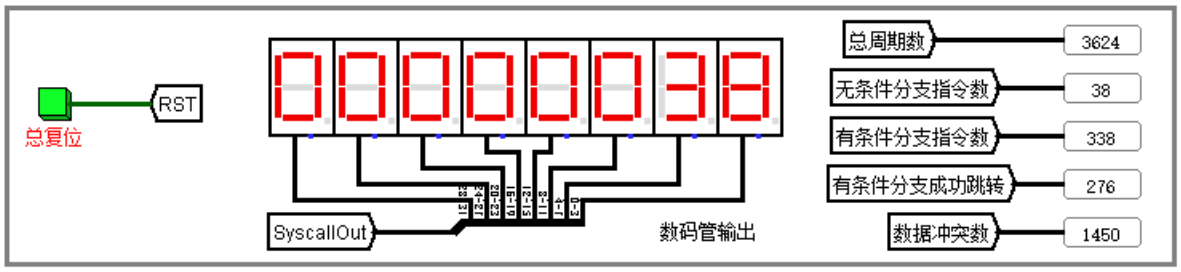

图 4.3 气泡流水线测试

由图可知，运行结果正确无误。

（3） 重定向流水线测试

根据重定向流水周期运算公式：总周期数 = 1546 + 4 + Load_use 次数 + 无条件*2 + 有条件成功*2

=1546 + 4 + 120 + 38*2 + 276*2

=2298

运行结果如图 4.4 所示

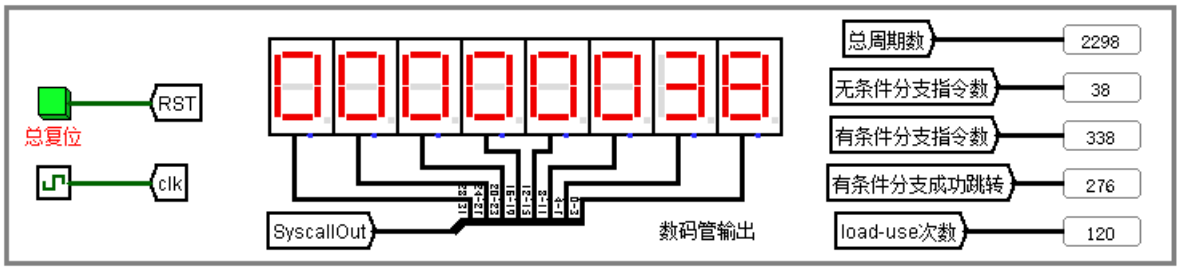

图 4.4 重定向流水线测试

由图可知测试结果正确

4.1.3 中断测试

中断测试包括 1、2、3 三个中断测试，进入哪个中断则显示该数字的跑马灯。例如进入“1”号中断，则停止当前程序并显示数字 1 的跑马灯，显示完后回到之前的程序。

（1） 单级中断测试

单级中断测试结果如图 4.5 所示

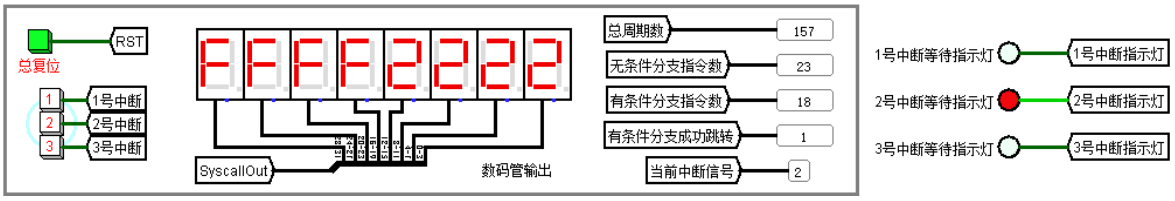

图 4.5 单级中断测试

由图可知测试结果无误。

（2） 多级中断测试

依次点击 1，2，3 号中断，2 号中断将打断 1 号中断，3 号中断将打断 2 号中断，测试结果如图 4.6 所示

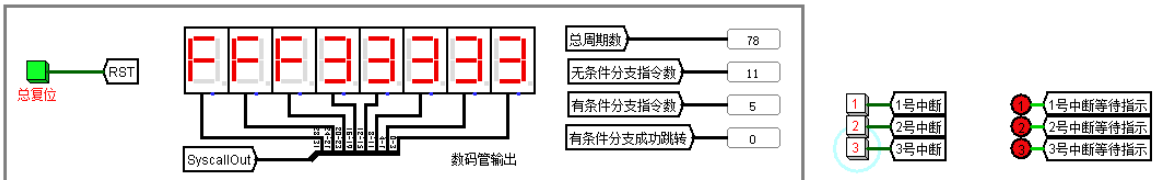

图 4.6 多级中断测试

（3） 流水单级中断测试

流水单级中断测试结果如图 4.7 所示

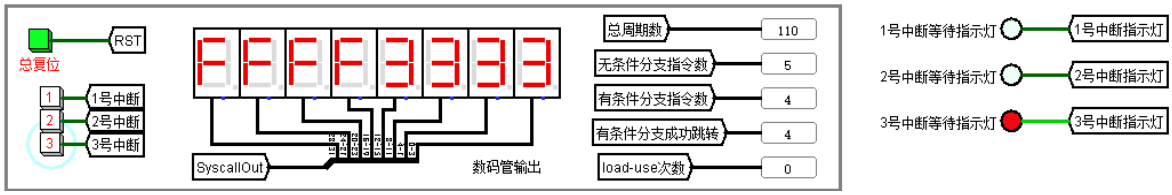

图 4.7 流水单级中断测试

由测试结果知，可正确完成 1、2、3 号中断并返回，结果正确。

4.2 性能分析

主要通过各个阶段的执行 benchmark 的总周期数来作性能的比较。

表 4.1 不同 CPU 周期数比较

CPU 类型	总周期	无条件分支	条件分支	条件成功跳转	气泡
单周期	1546	38	338	276	0
气泡流水	3624	38	338	276	1450
重定向流水	2298	38	338	276	120

根据表 4.1，可以看出气泡流水线虽然在单周期 CPU 基础上将其改成了流水线，但是其效率也降低了不少，总周期数由原来的 1546 增加到了 3624，虽然 cpu 的时钟频率理论上可以上升五倍，但是总周期也增加了一倍多，显然这有很大的缺陷；而重定向流水线在气泡流水线的基础上，将周期数降到了 2298，这是一个非常好的改进，理论上不仅可以将频率提升五倍，同时保持到总周期数只增加了 48.6%。

综上所述，重定向流水线应该是效率最高的，性能最好的，但是由于流水接口的添加，一些重定向模块的添加，相应的重定向的成本也是最高的，典型的用成本换取性能。

4.3 主要故障与调试

4.3.1 单周期 CPU 上板故障

单周期 CPU 上板： always 块编写错误。

故障现象：七段数码管无显示，led 灯无显示

原因分析：写 always 块时@( )括号中为空。

解决方案：改为 always @(SW)或者 always @(*)

4.3.2 理想流水线故障

理想流水线：周期错误。

故障现象： 理想流水周期应为 21，而我的周期数为 18

原因分析：通过错误排查，发现是停机指令传错了，在 ID 段便停机了，导致有三个周期中的三条指令没有执行完。

解决方案：将停机指令通过流水传到 WB 段，在 WB 段判断停机即可。

4.3.3 气泡流水线故障

气泡流水线：地址转移逻辑错误。

故障现象：气泡流水线数码管显示错误，并且总周期无法停下来。

原因分析：通过排查发现是因为在 EX 段有 Bne、Beq 等指令时，下一个 PC 地址应该使用 EX 段的 PC 地址计算得到新的 PC，而不是 IF 段的 PC 值。

解决方案：将 IF 段的 PC 值和指令一起通过流水线传到 EX 段，当判断成功跳转时再将该 PC 值送回到 NPC 进行计算。

4.4 实验进度

表 4.2 课程设计进度表

时间	进度
第一天	完成了单周期 CPU 上板的分频器、数码管、led 灯显示模块的编写
第二天	对显示模块进行了仿真测试，并与其他组员一起进行模块综合
第三天	对单周期 CPU 上板中有问题的模块进行调试、修正，与其他组员一起合作最终上板演示成功

时间	进度
第四天	理解并设计理想流水线，对单周期 CPU 进行修改，并对复杂部分进行封装
第五天	完成了理想流水线的检查，开始着手理解并设计气泡流水线
第六天	经过周末的加班，成功完成并检查了气泡流水线，着手在气泡流水线的基础上设计重定向流水线并进行调试。
第七天	完成并检查了重定向流水线，开始看中断，并在单周期 CPU 的基础上设计中断机制。
第八天	完成并检查了单级中断，并开始做流水单级中断
第九天	完成并检查了了流水单级中断，开始做多级嵌套中断
第十天	调试多级嵌套中断，与此同时准备重定向流水的上板模块编写。

5 设计总结与心得

5.1 课设总结

本次组成原理课程设计使用到了 Logisim 以及 FPGA 平台，从单周期 CPU 最终晚完善到了重定向 CPU，并添加了中断机制，除了动态分支预测功能、重定向流水上板没有实现以外，其他内容全部实现。主要做了如下几点工作：

1） 小组合作完成了单周期 CPU 上板；

这部分由小组合作完成，我负责编写显示部分，主要包括七段数码管的显示、led 灯的显示、总周期数以及各种跳转指令数的显示。最后将所有模块整合在一起，编写顶层模块并进行仿真调试。

2） 独立完成了理想流水线、气泡流水线、重定向流水线的设计；

这部分独立完成，过程总体来说比较复杂，通过三种流水设计完成了对单周期 CPU 性能的提升。

3） 独立完成了单级中断、流水单级中断、多级嵌套中断等三个中断。

这部分也是独立完成的，其中中断的实现包括硬件和软件的实现，不仅修改了电路，增加了中断判断机制等，还编写了汇编代码，实现中断的演示和中断返回等。

5.2 课设心得

本次课程设计总体来说内容还是比较多的，课程要求的任务量也比较大，两个星期从早到晚的不懈努力以及假期的辛苦加班才终于完成了整个课程设计的部分设计任务。现在再来回顾整个课程设计的整个过程，虽然没有能完成所有的任务，但其中也有不少的细节值得我去深思与体会，也学到了很多知识，锻炼了自己的能力。

本次课程设计是建立在上学期组成原理实验的基础之上，很多内容都是在单周期 CPU 基础上加以修改扩展实现。课程设计刚刚开始的时候，第一个任务是小组合作完成单周期 CPU 上板，在将该电路使用 Verilog 语言进行描述时，麻烦接踵而至。因为 Logism 对于电路是完全可视化的，连接过程清晰明了，但是使用 Verilog 语言进行数据通路时，因为各个部件的接口在定义时起名不是很规范，又因为只能使用各个 wire

型变量对于关键部件进行连接，而这些变量的名字在定义时同样没有做到一目了然，从而导致了我在完成了数据通路的连接后，使用了大量的时间对于整个电路进行排错和检查，最终还是将所有的接口和连线规范化的起了名字，才终于解决了问题。我们小组分工明确，在第二天就几乎完成了模块综合，然而，第三天遇到一个 bug 几乎就调了大半天。这也就说明了工程化、规范化对于电路设计或是程序设计的重要性。

紧接着，理想流水线 CPU 的设计并没有什么难度，主要是需要对流水线足够理解，但是使用插入气泡、数据重定向技术对于流水线 CPU 进行冒险处理时，因为这些方法书本上并没有，老师提供的 PPT 上也只有简单的一些描述，这就要求我不断地在网上搜索相关的知识内容，和小组内的成员进行相关探讨。

然后是中断部分，这部分的难度还是相当大的，我自己也在这部分耗费了不少时间。主要是单级中断、多级中断内容较多，不仅要考虑硬件部分，还要考虑软件部分，好在老师在实验过程中不断更新了实验指导文档，提供了很大帮助。后面自己实现的时候依旧遇到了很多问题，通过和已经做完的同学进行交流，看看别人是怎么做的，然后再去看自己的问题在哪儿，这样做起来就比较快了，这也说明了合作交流是十分重要的。

对于本次课程设计，我还有一些小小的建议和改进。本次课程采用分组机制，总体来说，分组是有利于完成任务的。但是仍存在小组分组的效果不明显。分组内的同学进度不一，有些做的快的同学也不愿意帮助落后的同学，有些同学掌握的不是很好，逐渐边缘化。另外，我建议精简报告，写到目前为止花了我整整两天一夜，报告内容多且杂，写报告的时间远比调 bug 的时间多。

总而言之，本次课程设计还是有一定难度的，但是还好有老师的无微不至的教导与解释，同时有丰富的资料进行查阅。同时课程实验的设计文档也提供了很好的帮助，任务分为很多个小的阶段，循序渐进，让同学越做越有信心。小组讨论以及工资日志，完成进度的方式督促同学每天需要完成任务，不要拖沓，我认为这是一个非常好的模式。

最后要感谢大萝卜老师的付出，不管是深夜在群里解答问题，还是多次修改丰富实验文档，都能看出大萝卜老师在教学方面花的心血，同时也感谢其他老师答疑解惑，也感谢同学们我的帮助和建议。这次课设让我受益匪浅，也将对我以后的学习工作有很大帮助。

参考文献

[1] DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第 4 版).北京：机械工业出版社.

[2] David Money Harris(美).数字设计和计算机体系结构（第二版）. 机械工业出版社

[3] 秦磊华，吴非，莫正坤.计算机组成原理. 北京：清华大学出版社，2011 年.

[4] 袁春风编著. 计算机组成与系统结构. 北京：清华大学出版社，2011 年.

[5] 张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008 年.

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: