華中科技大學

2019

# 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： | ACM1601 |
| 学　　号： | U201614831 |
| 姓　　名： | 苏墨馨 |
| 电　　话： | 13129967683 |
| 邮　　件： | 392558193@qq.com |

# 目 录

# 1 课程设计概述

## 1.1 课设目的

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

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

## 1.2 设计任务

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

## 1.3 设计要求

（1） 根据课程设计指导书的要求，制定出设计方案；

（2） 分析指令系统格式，指令系统功能。

（3） 根据指令系统构建基本功能部件，主要数据通路。

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

（5） 设计出实现指令功能的硬布线控制器；

（6） 调试、数据分析、验收检查；

（7） 课程设计报告和总结。

## 1.4 技术指标

（1） 支持表 1.1 前 27 条基本 32 位 MIPS 指令；

（2） 支持教师指定的 4 条扩展指令；

（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 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 | SLTI | 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 | XOR | 异或 | |
| 29 | XORI | 立即数异或 | |
| 30 | SH | 存储半字 | |
| 31 | BGTZ | 大于 0 转移 | |

# 2 总体方案设计

## 2.1 单周期 CPU 设计

单周期 CPU 的设计分为了两部分完成，第一部分是由 logisim 设计完成的 28 条指令的 CPU，第二部分是用 verilog 语言开发编写的 CPU。本次我们采用的方案是硬布线控制，且主、控存分开的方案，即采用硬布线控制方式，实现主存储器（MM）和微程序控制存储器（CM）不共用一个存储器的方式完成方案的设计。根据 logisim 的设计图完成 CPU 的模块划分。包括 ALU 运算器模块，controller 硬布线控制器模块，IM 指令存储器模块，显示模块，数据扩展模块，RAM 数据存储器模块，PC 程序计数器，npc 下一个 PC 计算模块，系统调用处理模块。

总体结构图如图 2.1 所示。

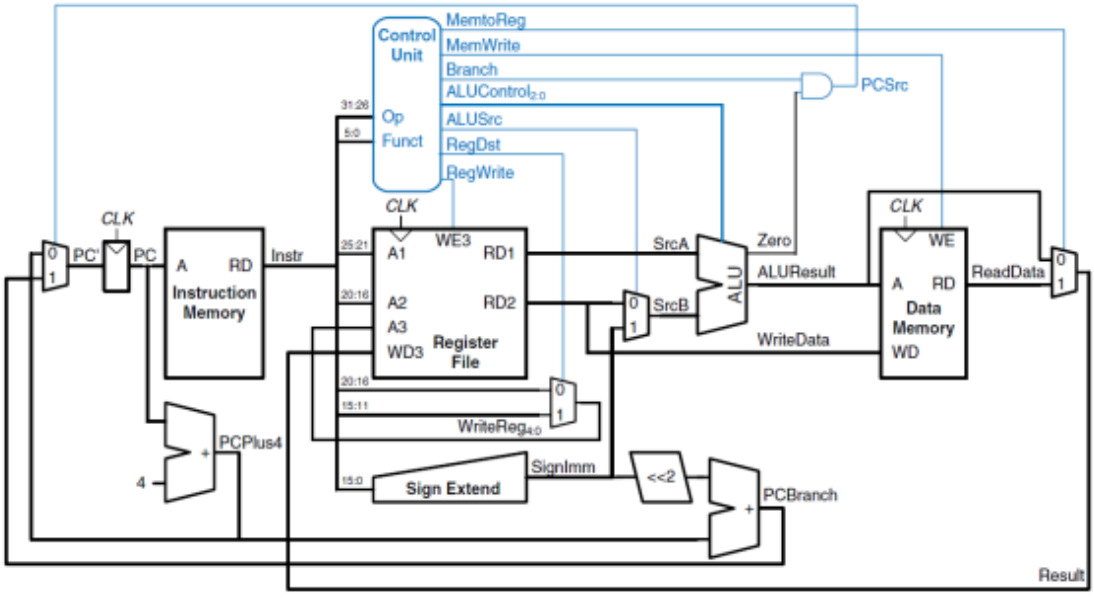


图 2.1 总体结构图

### 2.1.1 主要功能部件

在分模块编写的时候，我们划分了很多个模块进行编写，其中主要功能部件包括程序计数器 PC，指令存储器 IM，运算器 ALU，控制器 control，寄存器堆 RF 以及数据存储器 RAM。

1. 程序计数器 PC

程序计数器 PC 模块实现比较简单，当有时钟 clk_in 到来的时候，判断是否有清零信号 rst_in，如果有输出 PC＝0，再判断使能端是否为 1，如果为 1，输出 nextpc_in。nextpc_in 是在 NPC 模块里根据输入的信号进行判断结合多路选择器完成。有四种情况，PC+4，Rs 寄存器的值，PC+4+(exend)imm << 2，以及 PC 高 4 位和指令的 index 位加末位两个 0 的值。

2. 指令存储器 IM

用$readmemh 命令可以直接读取路径上的数据到数组 mem 中。因为在开始的时候路径设置出现问题导致读取内存失败没有读出指令（在流水上板时发现），所以代码中用 initial 初始化。

3. 运算器

根据输入的ALU_OP选择对应的操作并进行输出，因为我们没有用到OF和UOF，所以没有设置这两个接口。Result2 有接口但没有用到。

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 ALU 功能表 |
| Result | 输出 | 32 | ALU 运算结果 |
| Result2 | 输出 | 32 | ALU 结果第二部分，用于乘法指令结果高位或除法指令的余数位，其他操作为零 |
| Equal | 输出 | 1 | Equal=(x==y)?1:0, 对所有操作有效 |

表 2.2 ALU 功能表

| 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

在时钟控制下读取寄存器 reg[R1#]、reg[R2#]的值分别输出到 R1,R2；当使能端 WE 为 1 时，将第 W#个寄存器 reg[W#]写入为 Din。

### 2.1.2 数据通路的设计

根据 MIPS 指令集分析数据通路。根据不同的器件的输入的选择信号用多路选择器进行选择。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 2.2 所示。

**表 2. 2 指令系统数据通路框架**

| 指令 | PC | IM | RF | | | | ALU | | | DM | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | R1# | R2# | W# | Din | A | B | OP | Addr | Din |
| ADD | PC+4 | PC | rs | rt | rd | ALU | R1 | R2 | 5 | | |
| ADDI | PC+4 | PC | rs | | rt | ALU | R1 | Imm_ext | 5 | | |
| ADDIU | PC+4 | PC | rs | | rt | ALU | R1 | Imm_ext | 5 | | |
| ADDU | PC+4 | PC | rs | rt | rd | ALU | R1 | R2 | 5 | | |

| 指令 | PC | IM | RF | | | | ALU | | | DM | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | R1# | R2# | W# | Din | A | B | OP | Addr | Din |
| AND | PC+4 | PC | rs | rt | rd | ALU | R1 | R2 | 7 | | |
| ANDI | PC+4 | PC | rs | | rt | ALU | R1 | Imm_ext | 7 | | |
| SLL | PC+4 | PC | rs | rt | rd | ALU | R1 | Imm_ext | 0 | | |
| SRA | PC+4 | PC | rs | rt | rd | ALU | R1 | Imm_ext | 1 | | |
| SRL | PC+4 | PC | rs | rt | rd | ALU | R1 | Imm_ext | 2 | | |
| SUB | PC+4 | PC | rs | rt | rd | ALU | R1 | R2 | 6 | | |
| OR | PC+4 | PC | rs | rt | rd | ALU | R1 | R2 | 8 | | |
| ORI | PC+4 | PC | rs | | rt | ALU | R1 | Imm_ext | 8 | | |
| NOR | PC+4 | PC | rs | rt | rd | ALU | R1 | R2 | 10 | | |
| LW | PC+4 | PC | rs | rt | rd | DM.out | R1 | Imm_ext | 5 | ALU | |
| SW | PC+4 | PC | rs | rt | | | R1 | Imm_ext | 5 | ALU | R2 |
| BEQ | PC+4+offset 或 PC+4 | PC | rs | rt | | | R1 | R2 | | | |
| BNE | PC+4+offset 或 PC+4 | PC | rs | rt | | | R1 | R2 | | | |
| SLT | PC+4 | PC | rs | rt | rd | ALU | R1 | R2 | 11 | | |
| SLTI | PC+4 | PC | rs | | rt | ALU | R1 | Imm_ext | 11 | | |
| SLTU | PC+4 | PC | rs | rt | rd | ALU | R1 | R2 | 12 | | |
| J | PC[31:28] \|Index\|2'b00 | PC | | | | | | | | | |
| JAL | PC[31:28] \|Index\|2'b00 | PC | | | | PC+4 | | | | | |
| JR | GPR[rs] | PC | rs | | | | | | | | |
| SYSCALL | | | 0x02 | 0x04 | 0x1f | | | | | | |
| XOR | PC+4 | PC | rs | rt | rd | ALU | R1 | R2 | 9 | | |
| XORI | PC+4 | PC | rs | | rt | ALU | R1 | Imm_ext | 9 | | |
| SH | PC+4 | PC | rs | rt | | | R1 | Imm_ext | 5 | ALU | R2 |
| BGTZ | PC+4+offset 或 PC+4 | PC | rs | rt | | | R1 | R2 | 11 | | |

### 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 2.3 所示。

表 2.3 主控制器控制信号的作用说明

| 控制信号 | 说明 | 产生条件 |
|---|---|---|
| RegWrite | 寄存器写使能 | 寄存器写回信号 |
| MemWrite | 写内存控制信号 | sw 指令未单独设置 MemRead 信号 |
| AluOP | 运算器操作控制符（4 位） | R 型指令根据 Func 选择 |
| MemToReg | 寄存器写入数据来自存储器 | lw 指令 |
| RegDst | 写入寄存器编号 rt/rd 选择 | R 型指令 |
| AluSrcB | 运算器 B 输入选择 | lw 指令，sw 指令，立即数运算类指令 |
| SignedExt | 立即数符号扩展 | ADDI、ADDIU、SLTI 指令 |
| JR | 寄存器跳转指令译码信号 | JR 指令 |
| JAL | JAL 指令译码信号 | JAL 指令 ，选择寄存器写回编号，写回值 |
| JMP | 无条件分支控制信号 | J、JAL、JR 指令，选择无条件分支地址 |
| Beq | Beq 指令译码信号 | Beq 指令，用于有条件分支控制 |
| Bne | Bne 指令译码信号 | Bne 指令，用于有条件分支控制 |
| Syscall | Syscall 指令译码信号 | 根据$V0 寄存器的值，决定是停机还是输出 |

对照所有控制信号，依次分析各条指令，分析该指令执行过程中需要哪些控制信号，对于与本条指令无关的控制信号，控制信号的取值一律为 0，以简化控制器电路的设计。该控制信号表的框架如表 2.4 所示。

表 2.4 主控制器控制信号框架

| 指令 | Memt oReg | Mem Write | ALU_ SRC | RegW rite | SYSC ALL | Signe dExt | Reg Dst | BEQ | BNE | JR | JMP | JAL | SH | BGTZ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SLL | | | | 1 | | | 1 | | | | | | | |
| SRA | | | | 1 | | | 1 | | | | | | | |
| SRL | | | | 1 | | | 1 | | | | | | | |
| ADD | | | | 1 | | | 1 | | | | | | | |
| ADDU | | | | 1 | | | 1 | | | | | | | |

| 指令 | MemtoReg | MemWrite | ALU_SRC | RegWrite | SYSCALL | SignedExt | RegDst | BEQ | BNE | JR | JMP | JAL | SH | BGTZ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SUB | | | | 1 | | | 1 | | | | | | | |
| AND | | | | 1 | | | 1 | | | | | | | |
| OR | | | | 1 | | | 1 | | | | | | | |
| NOR | | | | 1 | | | 1 | | | | | | | |
| SLT | | | | 1 | | | 1 | | | | | | | |
| SLTU | | | | 1 | | | 1 | | | | | | | |
| JR | | | | | | | | | | | 1 | 1 | | |
| SYSCALL | | | | | 1 | | | | | | | | | |
| J | | | | | | | | | | | | 1 | | |
| JAL | | | | 1 | | | | | | | | 1 | 1 | |
| BEQ | | | | | | | | 1 | | | | | | |
| BNE | | | | | | | | | 1 | | | | | |
| ADDI | | | 1 | 1 | | 1 | | | | | | | | |
| ANDI | | | 1 | 1 | | | | | | | | | | |
| ADDIU | | | 1 | 1 | | 1 | | | | | | | | |
| SLTI | | | 1 | 1 | | 1 | | | | | | | | |
| ORI | | | 1 | 1 | | | | | | | | | | |
| LW | 1 | | 1 | 1 | | | | | | | | | | |
| SW | | 1 | 1 | | | | | | | | | | | |
| XOR | | | | 1 | | | 1 | | | | | | | |
| XORI | | | 1 | 1 | | | | | | | | | | |
| SH | | 1 | 1 | | | | | | | | | | 1 | |
| BGTZ | | | | | | 1 | | | | | | | | 1 |

## 2.2 中断机制设计

### 2.2.1 总体设计

设计 CP0 寄存器用于中断处理，增加一个单独的控制部件处理 MTC0、MFC0 和

ERET 三条指令。增加控制信号 eret,mfc0 和 mtc0。根据中断产生信号的优先级判断现在需要执行的中断指令，当接收到 eret 信号时中断退出。中断的总体设计如图 2.2 所示。

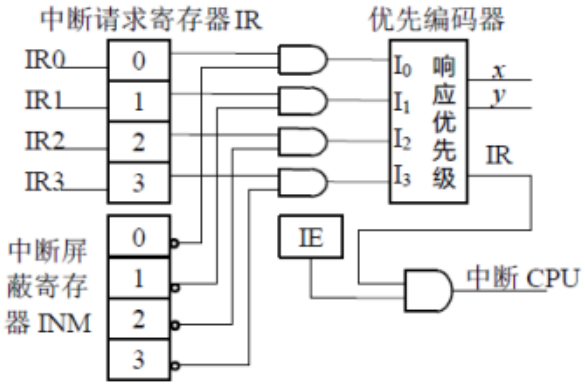


图 2.2 中断总设计

### 2.2.2 硬件设计

硬件需要实现中断识别和中断响应过程。需要设计有中断按键产生电路。中断识别的时候需要根据当前收到的中断请求，通过优先编码器从而实现按照优先级进行响应。进入中断响应是指令执行周期结束后，根据中断请求信号暂停当前执行的程序进入中断隐指令周期。CPU 需要实现关中断，保存 PC，进行中断源的识别获取中断服务程序地址并将中断服务程序入口地址送入 PC。为了实现中断响应，除了优先级之外还需要知道中断程序的入口地址。用硬件方式实现向量中断机制，根据编写的中断软件程序在指令存储器中的存储地址，直接用多路选择器选择对应的 PC。需要将之前的 PC 送至寄存器进行保存，在中断结束后恢复。因为优先级固定，所以不需要考虑中断屏蔽寄存器部分。

### 2.2.3 软件设计

软件设计即编写中断服务程序。中断服务程序需要保护现场，所有中断服务程序需要用到的寄存器都需要被保护，PC 因为会发生改变，也需要进行保护。保护现场开始前需要关中断，避免被其他程序打断，结束后开中断允许 CPU 响应其他的中断（多级嵌套时需要）。中断服务程序结束后，需要恢复现场，和保护现场一样需要先关中断再开中断。最后用 eret 指令实现中断返回。开中断和中断返回需要同时完成，否则开中断之前如果有其他中断请求在等待，中断返回 eret 指令将会被中断，保存断

点的 EPC 寄存器将会被破坏，这里关中断由硬件完成。另外，中断返回时，当前中断被处理完毕，应该将当前中断服务程序对应 IR 清零。

## 2.3 流水 CPU 设计

### 2.3.1 总体设计

将单周期 CPU 改造为流水线 CPU，需要将 MIPS 指令的执行过程划分为 5 个阶段：取指令 IF 段，译码 ID 段，执行指令 EX 段，访存 MEM 段，写回 WB 段。IF 阶段包括程序计数器 PC，指令存储器 IM；ID 段包括寄存器堆 RegFile 和控制器；EX 段包括运算器 ALU；MEM 段是内存读写阶段，包括数据存储器 DM；WB 段是写回阶段，获得寄存器堆写入的数据。

每个阶段之间由流水接口部件连接，流水接口部件是一个锁存器，用于暂存当前处理完成的数据和结果，并将结果传递给下一个阶段。流水接口部件除了时钟之外还需要同步清零信号和使能端控制。MIPS 五段流水数据通路如图 2.3 所示

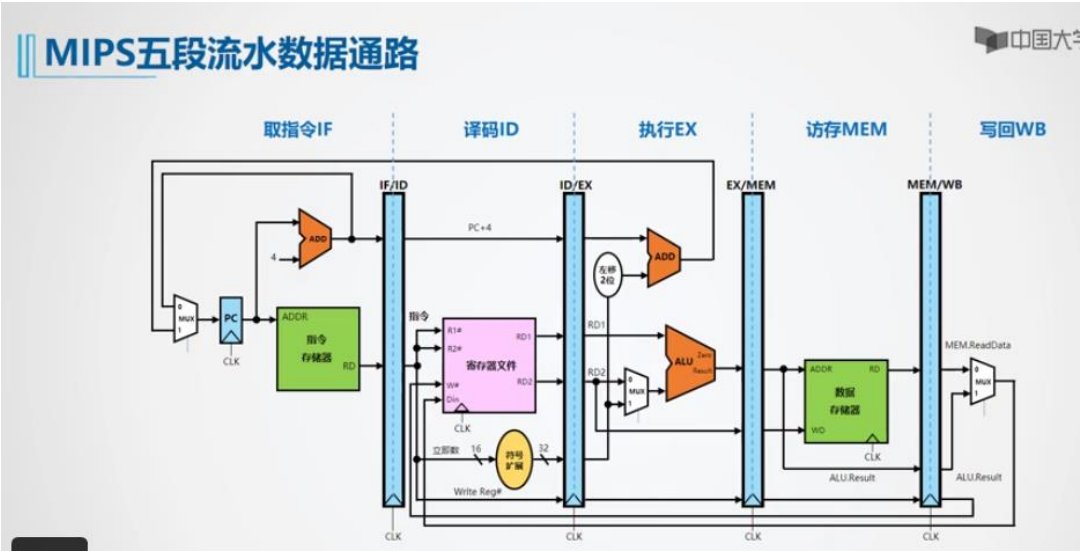


图 2.3 MIPS 五段流水数据通路

### 2.3.2 流水接口部件设计

流水接口部件主要是用寄存器来锁存数据。根据后面需要的信号设计各个阶段的流水接口部件。此外还需要时钟，同步清零信号以及使能端信号。可以在上学期 CRC 码流水传输实验中老师提供的流水接口部件的基础上进行修改，增加需要的信号从而实现 CPU 中需要的流水接口部件。

### 2.3.3 理想流水线设计

理想流水线实现了一个简单的流水线，这个流水线不涉及分支或者冲突问题。理想流水线中所有指令执行的阶段数相同，段时延相同，没有资源冲突并且没有段间互锁（无数据相关也无分支相关）。IF/ID、ID/EX、EX/MEM 和 MEM/WB 四个流水接口部件需要的信号不一样。根据图 2.4 所示的各个阶段所需要的信号设计流水接口部件。

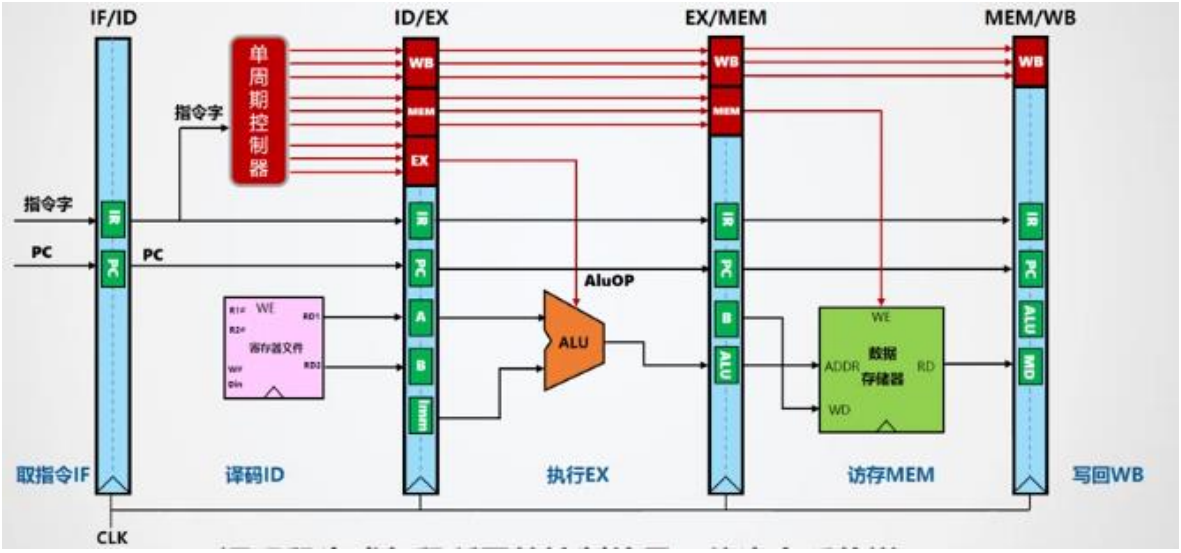


图 2.4 流水接口部件传递的信号

## 2.4 气泡式流水线设计

在非理想的指令执行的过程中会出现指令流水线的相关、冲突和冒险。三种相关分别是资源相关、分支相关以及数据相关。

资源相关通常有主存争用和 ALU 争用。IF 段取指令和 ID 段取操作数导致主存争用，可以通过哈佛结构解决。多周期中计算 PC、分支地址和运算指令导致 ALU 争用，可以通过增加加法器消除。

分支相关是指在指令分支跳转成功之后未跳转地址的后续指令仍在流水线中，如果继续执行会导致错误。所以分支跳转发生之后需要清空之前的误取指令，通过控制流水接口部件的同步清零端引入气泡实现。

数据相关是指指令操作数依赖于前一条指令的执行结果，通过暂停流水线直到数据写回之后再读取可以解决。ID 和 WB 段的数据相关即先写后读冲突，可以通过在时钟下降沿写入寄存器文件，在时钟上升沿流水接口工作解决。ID 段和 MEM 段或

EX 段的数据相关可以在 ID 端增加数据相关检测逻辑，如果发生数据相关暂停 IF、ID 段等待数据写回，在 EX 段插入气泡，如果相关解除，流水线继续正常工作。

## 2.5 数据转发流水线设计

气泡流水线通过插入气泡和暂停流水线实现指令的正常执行，但这样插入大量气泡会导致流水线的性能大幅降低。为了避免插入大量气泡，修改气泡流水线为重定向流水线。

重定向流水线顾名思义是构造数据旁路，将后段处理后还没来得及写会的数据进行重定向，在真正需要数据的时候直接将后段的正确数据重定向。另外，因为 Load-use 相关没有办法通过数据重定向解决，所以需要在 ID 段增加 Load-use 检测，插入气泡。

重定向流水线的数据通路图如图 2.5 所示

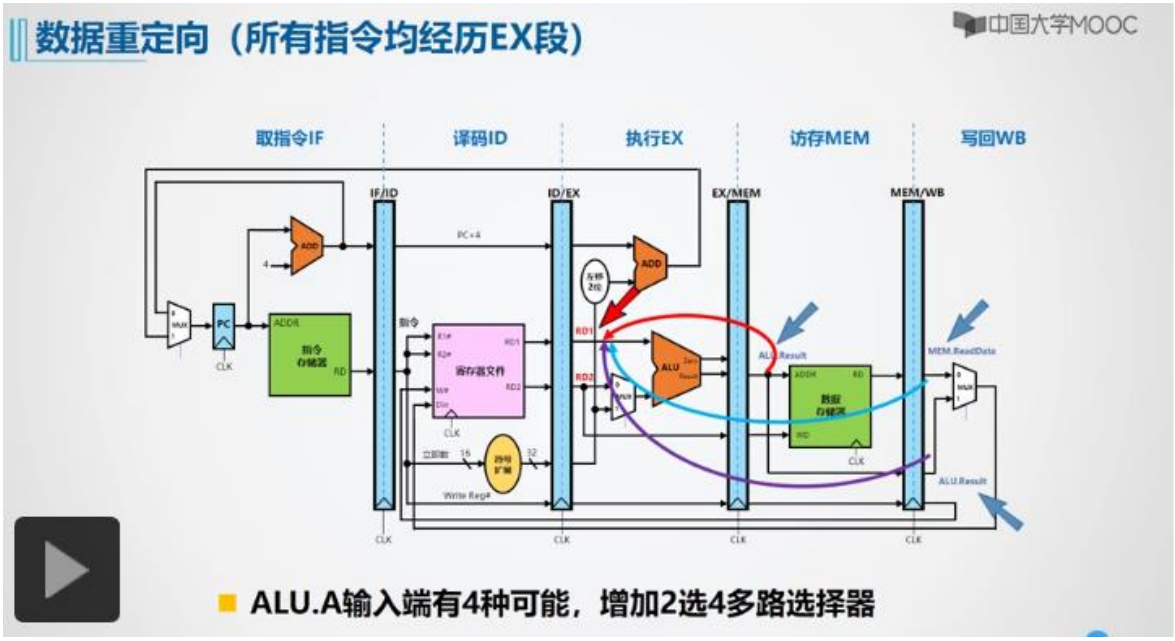


图 2.5 重定向流水线的数据通路

# 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。en 为使能信号，将此控制信号连接到 PC 的使能端。当需要停机的时候，en 信号为 0，如图 3.1 所示。

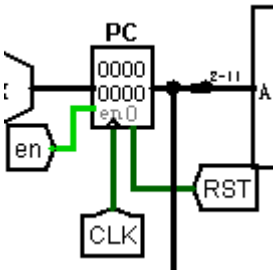


图 3.1 程序计数器（PC）

② FPGA 实现：

程序计数器 PC 的 Verilog 代码如下：

```
module PC (nextpc_in, enable_in, clk_in, rst_in, pc_out);
    input wire [31:0] nextpc_in;
    input wire enable_in,clk_in,rst_in;
    output reg [31:0] pc_out;
     initial begin
        pc_out = 0;
    end
    always @(posedge clk_in) begin
        if ( rst_in )
            pc_out = 32'h00000000;
        else if ( enable_in )
```

```
            pc_out = nextpc_in;
        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。

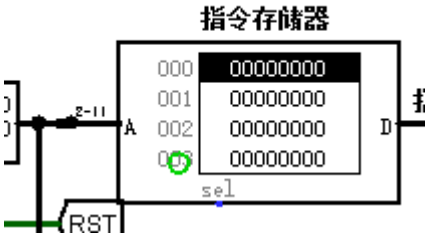


图 3.2 指令存储器（IM）

② FPGA 实现：

$readmemh 系统调用实现指令的读取。

指令存储器 IM 的 Verilog 代码如下：

```
module IM( input[31:0] addr,output[31:0]dout);
  reg [31:0]im_mem[1023:0];
      initial begin
          $readmemh("G:\junior\CPU \ben.hex",mem);
      end
endmodule
```

3) 寄存器文件（RF）

① Logism 实现：

使用 cs3410 库封装的 RegFile 寄存器文件。输入为指令译码之后的得到的寄存器编号 R1、R2、W，后面得到的 Din 数据，RegDst（WE）信号。如图 3.3 所示。

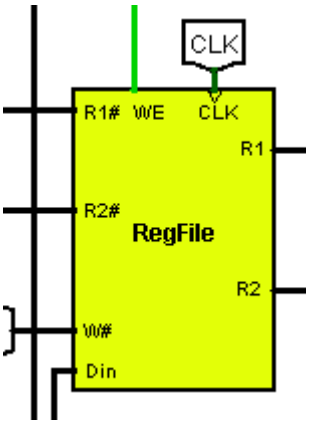


图 3.3 寄存器文件（RF）

② FPGA 实现：

$readmemh 系统调用实现指令的读取。

指令存储器 IM 的 Verilog 代码如下：

```
module RegFile(r1, r2, w, din, we, clk, a, b);
    input [4:0] r1,r2,w;//寄存器编号
    input clk, we;          //时钟，写入控制
    input [31:0] din; //写入数据
    output [31:0] a,b;//两个输出
    assign a = regs[r1];
    assign b = regs[r2];
    reg [31:0] regs[31:0];//  寄存器中存放的数据
always @(posedge clk)
        begin
            if(we == 1'b1)
                begin
                    regs[w] = din;
                end
        end
endmodule
```

4) 数据存储器（DM）

① Logism 实现：

使用一个 RAM 实现数据存储器（DM）。设置该存储器的地址位宽为 20 位，数

据位宽为 32 位。取 ALU 结果的 2-11 位做为数据存储器的低位，高位补 0。为了实现 SH 指令，需要根据 SH 信号决定写入的位置，控制 sel 位实现这个功能。实现如图 3.4 所示。

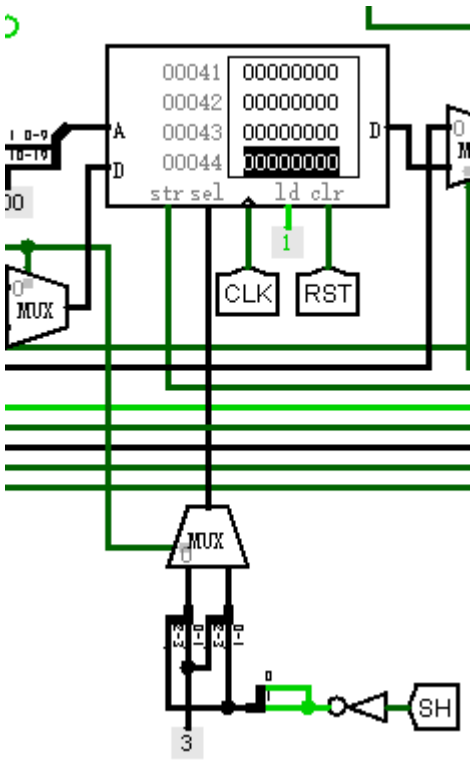


图 3.4 数据存储器（DM）

② FPGA 实现：

$readmemh 系统调用实现指令的读取。

指令存储器 IM 的 Verilog 代码如下：

```
module IM( input[31:0] addr,output[31:0]dout);
  reg [31:0]im_mem[1023:0];
      initial begin
          $readmemh("G:\junior\CPU \ben.hex",mem);
      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中的数据通路表 2.2 所示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。

单周期 CPU 数据通路如图 3.5 所示。

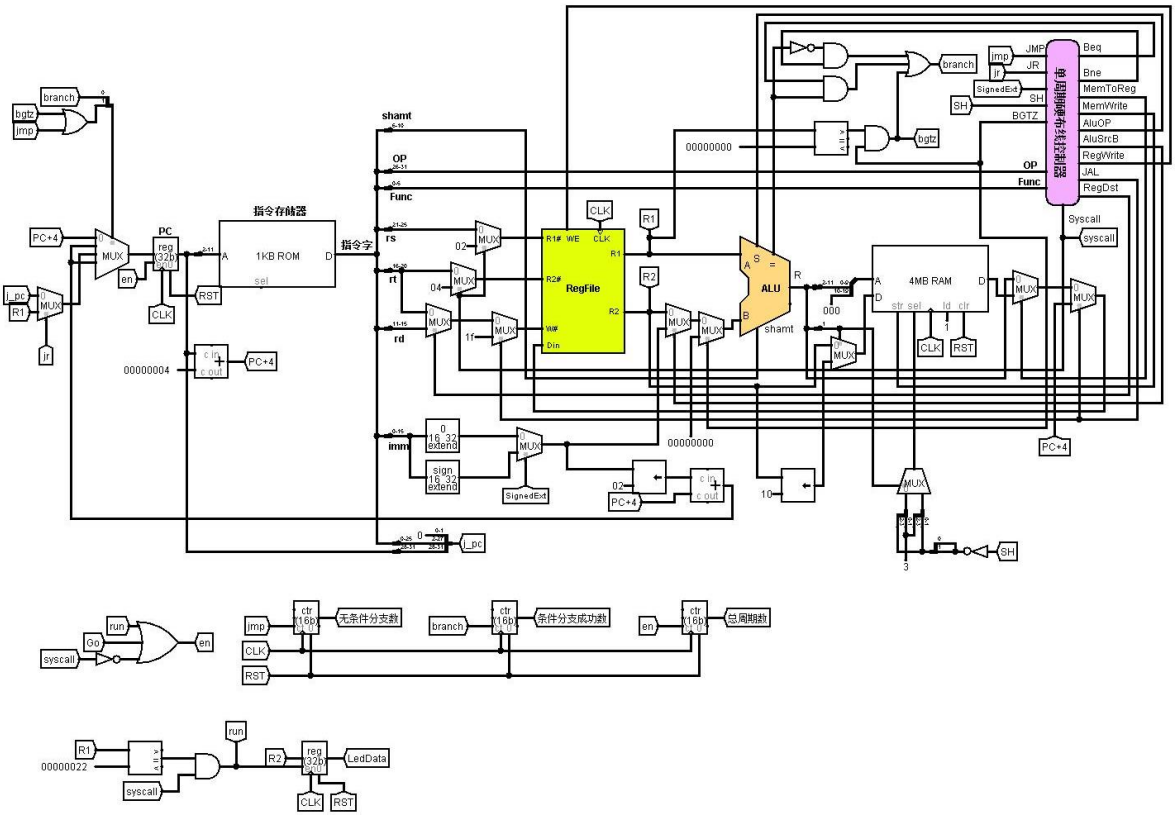


图 3.5 单周期 CPU 数据通路（Logism）

在 Vivado 中使用 Verilog 语言搭建的数据通路的原理图如图 3.6 所示。

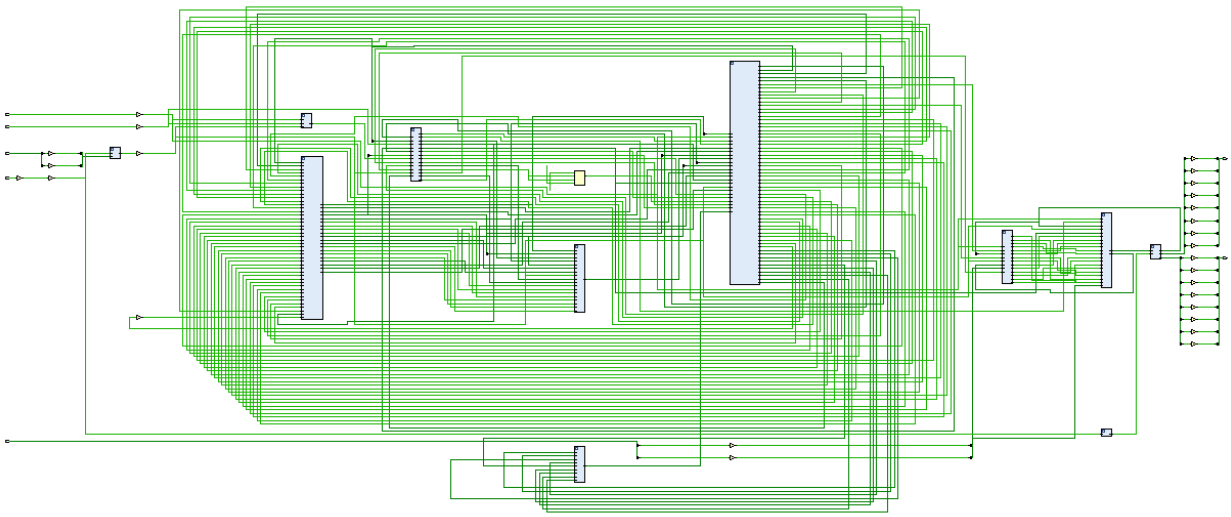

图 3.6 单周期 CPU 数据通路（FPGA）

### 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。

主控制器对照控制器信号**错误!未找到引用源。**，根据 excel 表得到真值表，然后利用 logisim 的电路分析功能直接生成硬布线控制电路。

Branch 控制器是根据 ALU 的 equal 信号以及 R1 和 0 比较的大小进行跳转的，我在实现 BGTZ 指令的时候没有用 ALU 而是用了一个单独的比较器进行比较。如果条件跳转指令成功，则产生信号 branch 连接到 PC 输入端的 MUX 进行地址的跳转，Branch 控制器的实现如图 3.7 所示

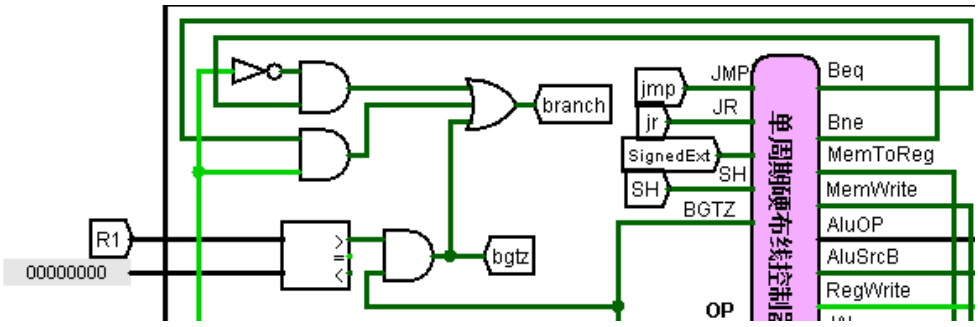


图 3.7 Branch 控制器

SYSCALL 控制器根据寄存器文件读出的 R1 的值判断需要停机还是显示数据。同样根据 syscall 指令获得 PC 的使能端信号 en，实现 Go 继续运行而 syscall 停机或者显示数据。SYSCALL 控制器的实现如图 3.8 所示。

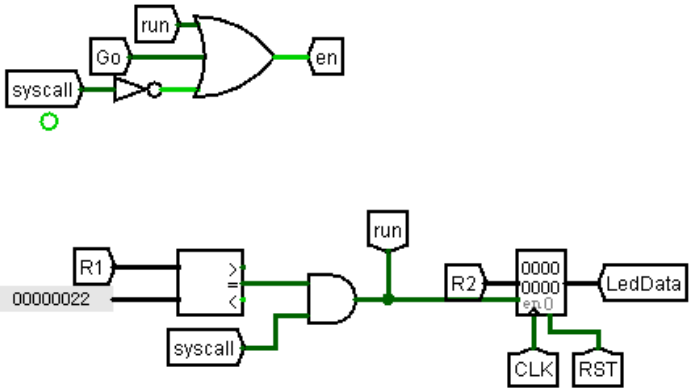


图 3.8 SYSCALL 控制器

① FPGA 实现

根据输入的 OP_CODE 和 FUNC 字段，解析出指令并输出所需要的信号实现主控制器。解析指令部分的代码如下。

```
always@(OP_CODE or FUNC) begin
    if(OP_CODE == 0)begin
        SLL = (FUNC == 0) ? 1 : 0;
        SRA = (FUNC == 3) ? 1 : 0;
        SRL = (FUNC == 2) ? 1 : 0;
        ADD = (FUNC == 32) ? 1 : 0;
        ADDU = (FUNC == 33) ? 1 : 0;
        SUB = (FUNC == 34) ? 1 : 0;
        AND = (FUNC == 36) ? 1 : 0;
        OR = (FUNC == 37) ? 1 : 0;
        NOR = (FUNC == 39) ? 1 : 0;
        SLT = (FUNC == 42) ? 1 : 0;
        SLTU = (FUNC == 43) ? 1 : 0;
        JR = (FUNC == 8) ? 1 : 0;
        SYSCALL = (FUNC == 12) ? 1 : 0;
        XOR = (FUNC == 38) ? 1 : 0;
        J = 0;
        JAL = 0;
        BEQ = 0;
```

```
        BNE = 0;
        ADDI = 0;
        ANDI = 0;
        ADDIU = 0;
        SLTI = 0;
        ORI = 0;
        LW = 0;
        SW = 0;
        SH = 0;
        BGTZ = 0;
        XORI = 0;
    end
    else begin
        SLL = 0;
        SRA = 0;
        SRL = 0;
        ADD = 0;
        ADDU = 0;
        SUB = 0;
        AND = 0;
        OR = 0;
        NOR = 0;
        SLT = 0;
        SLTU = 0;
        JR = 0;
        SYSCALL = 0;
        XOR = 0;
        J = (OP_CODE == 2) ? 1 : 0;
        JAL = (OP_CODE == 3) ? 1 : 0;
        BEQ = (OP_CODE == 4) ? 1 : 0;
```

```
        BNE = (OP_CODE == 5) ? 1 : 0;
        ADDI = (OP_CODE == 8) ? 1 : 0;
        ANDI = (OP_CODE == 12) ? 1 : 0;
        ADDIU = (OP_CODE == 9) ? 1 : 0;
        SLTI = (OP_CODE == 10) ? 1 : 0;
        ORI = (OP_CODE == 13) ? 1 : 0;
        LW = (OP_CODE == 35) ? 1 : 0;
        SW = (OP_CODE == 43) ? 1 : 0;
        SH = (OP_CODE == 41) ? 1 : 0;
        BGTZ = (OP_CODE == 7) ? 1 : 0;
        XORI = (OP_CODE == 14) ? 1 : 0;
    end
    ALU_OP =
        SLL ? 0 :
        SRA ? 1 :
        SRL ? 2 :
        (ADD | ADDU | ADDI | ADDIU || LW || SW || SH) ? 5 :
        SUB ? 6 :
        (AND | ANDI) ? 7 :
        (OR | ORI) ? 8 :
        (XOR | XORI) ? 9 :
        NOR ? 10 :
        (SLT | SLTI | BGTZ) ? 11 :
        SLTU ? 12 :
        0;
    MemToReg = LW ? 1 : 0;
    MemWrite = (SW | SH) ? 1 : 0;
    ALU_SRC = (ADDI | ANDI | ADDIU | SLTI | ORI | LW | SW | SH | XORI) ? 1 : 0;
    RegWrite = (SLL | SRA | SRL | ADD | ADDU | SUB | AND | OR | NOR | SLT | SLTU |
JAL | ADDI | ANDI | ADDIU | SLTI | ORI | LW | XOR | XORI) ? 1 : 0;
```

```
    SysCALL = SYSCALL ? 1 : 0;
    SignedExt = (BEQ | BNE | ADDI | ADDIU | SLTI | LW | SW | SH | BGTZ) ? 1 : 0;
    RegDst = (SLL | SRA | SRL | ADD | ADDU | SUB | AND | OR | NOR | SLT | SLTU |
  XOR)? 1 : 0;
    Beq = BEQ ? 1 : 0;
    Bne = BNE ? 1 : 0;
    JMP = (J | JAL | JR) ? 1 : 0;
end
```

在 Vivado 中使用 Verilog 语言构成的主控制器部分原理图如图 3.所示。

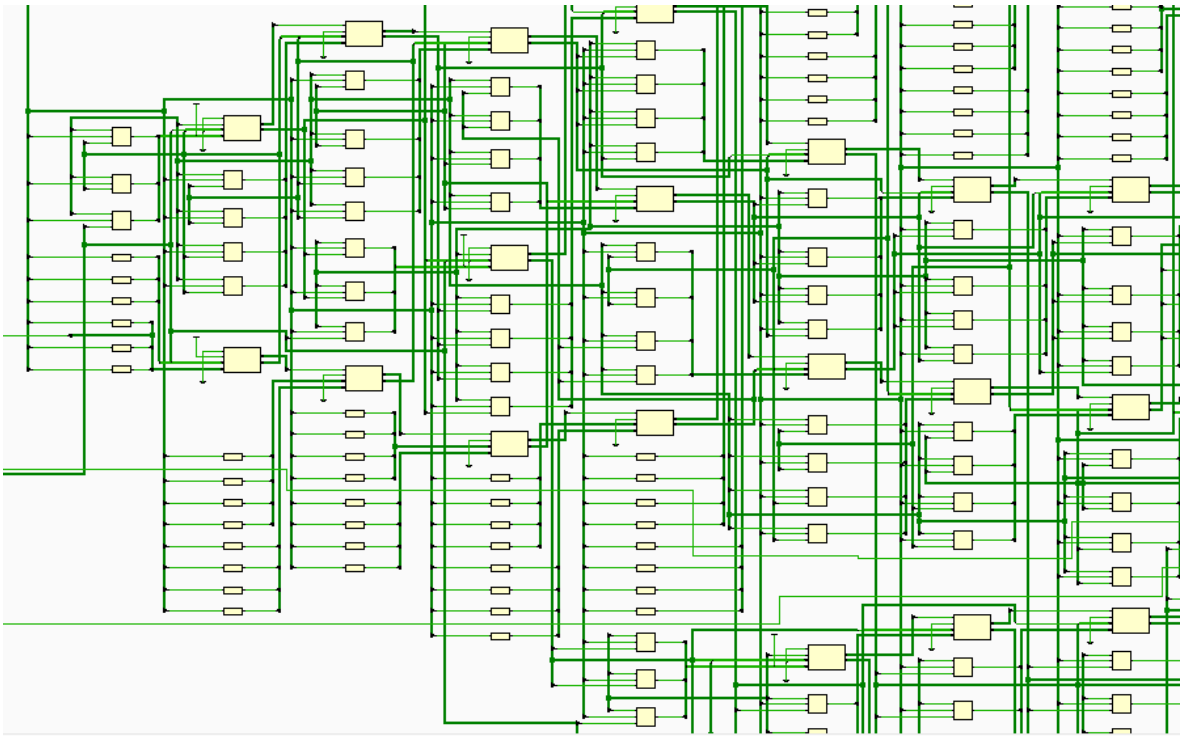

图 3.9 主控制器部分原理图

根据 syscall 控制器的原理实现 syscall 控制器的 verilog 代码如下

```
module syscall( input [31:0]r1,mwdata,input clk,rst,sys,
output [31:0]LedData,output led,halt,pause );
reg [31:0]data;
initial begin
    data=0;
end
assign LedData = data;
```

```
assign led = (r1==34 && sys == 1);
assign halt = 0;
assign pause = (sys == 1 && led==0 && halt==0);


always@(posedge clk or posedge rst)
   begin
      if(rst) data=0;
      else begin
         if(led) begin
            data = mwdata;
         end
      end
   end
endmodule
```

根据分支跳转的 logisim 实现编写下址字段的 verilog 代码如下

```
module npc( input rst,clk,signedext,jr,branch,jmp,input [31:0] pcnow,R1,input [15:0]
    Imm,input [25:0] Index,
   output wire [31:0] nextpc,output wire [31:0] imm_ext //ç«‹å³æ•?
);
      wire [31:0] pcj0;
      wire [31:0] pcj1;
      wire [31:0] pcj;
      wire [31:0] be;
      wire [31:0] pc_4;
      wire [31:0] stmp,utmp;
      wire [31:0] tmp;


      assign pc_4 = pcnow+4;
      assign pcj0 = {pc_4[31:28] ,Index,2'b00};
      assign pcj1 = R1;
```

```
    mux mux_1(.choose(jr),.data1(pcj0),.data2(pcj1),.out(pcj));
    sign_ex_16to32 sign_ex_hh(.data_16bit(Imm),.data_32bit(stmp));
    zero_ex_16to32 zero_ex_hh(.data_16bit(Imm),.data_32bit(utmp));
    mux mux_2(.choose(signedext),.data1(utmp),.data2(stmp),.out(imm_ext));
    assign be = pc_4+(imm_ext<<2);
    mux mux_3(.choose(branch),.data1(pc_4),.data2(be),.out(tmp));
    mux mux_4(.choose(jmp),.data1(tmp),.data2(pcj),.out(nextpc));
endmodule
```

## 3.2 中断机制实现

### 3.2.1 总体设计

中断机制总体设计的流程图如图 3.10 所示。

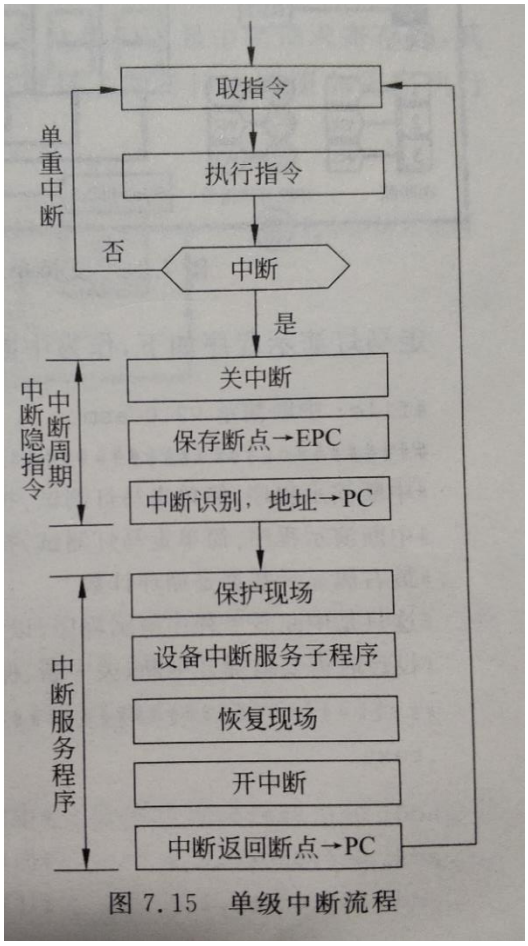


图 3.10 单级中断流程图

### 3.2.2 硬件实现

设计中断按键信号产生电路和中断识别逻辑，这里 IRi 隧道代表第 i 号中断源被按下，IRi 寄存器是第 i 号中断请求寄存器，Wi 代表第 i 号中断的中断指示灯信号。根据当前的中断号进行片选，在有 eret 指令的时候对 IRi 寄存器进行清零。中断信号需要响应中断优先级，通过优先编码器实现优先级的响应。用寄存器记录当前的中断号，因为在关中断的时候不允许响应其他中断，用开关中断产生的指令信号 mfc0 控制寄存器的使能端进行写入。如图 3.11 和 3.12 所示。

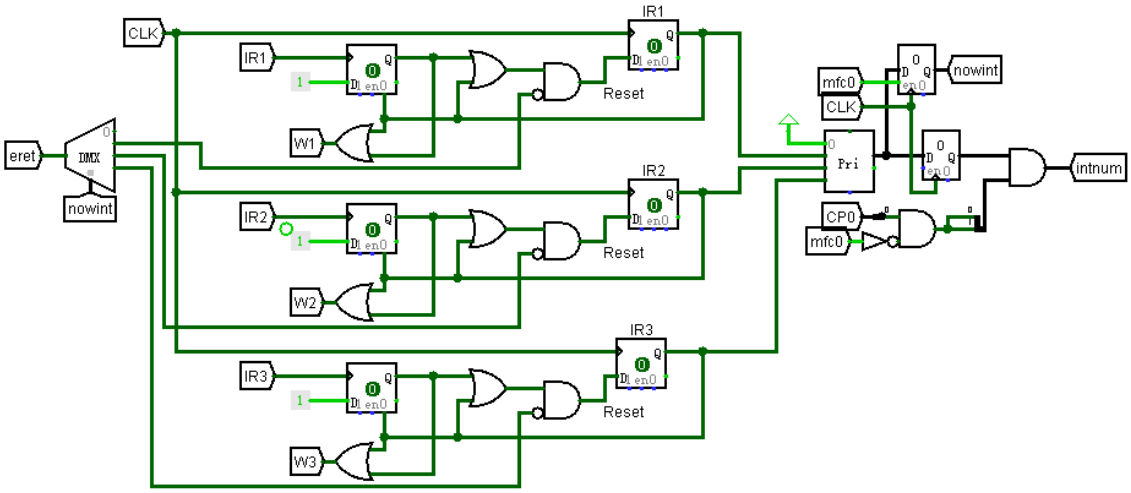


图 3.11 单级中断识别

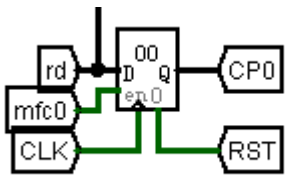


图 3.12 CP0 寄存器

除了中断识别之外，还需要获取中断服务程序的入口地址并将中断服务程序入口地址送 PC。如果当前中断号不为 0，EPC 寄存器保存断点处的 PC 的值，在接收到 eret 指令之后返回原来 PC 的位置，通过寄存器和多路选择器实现。根据当前中断号通过 mux 选择程序正常执行的 PC 或者中断向量表中的中断处理程序的地址。如图 3.13 所示。

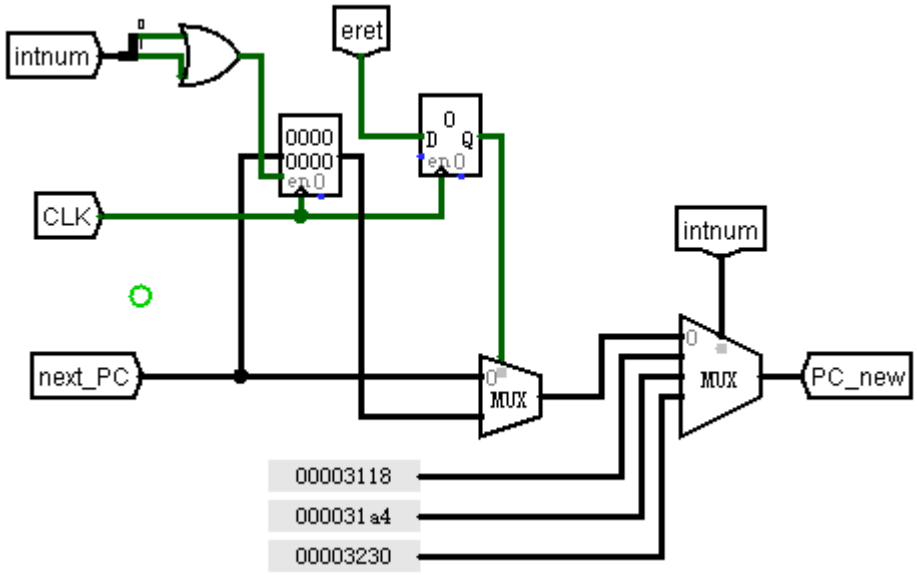


图 3.13 中断程序 PC 计算

添加了单级中断处理的 CPU 数据通路和原来的 CPU 基本上一样，除了增加了上述部分，改变了 PC 连接的隧道，还增加了 eret，mfc0，mtc0 三条指令信号的控制器。为了不修改原来的控制器封装，添加了一个中断指令控制器子模块如图 3.14 所示。其余部分和原来相同，不重复截图了。

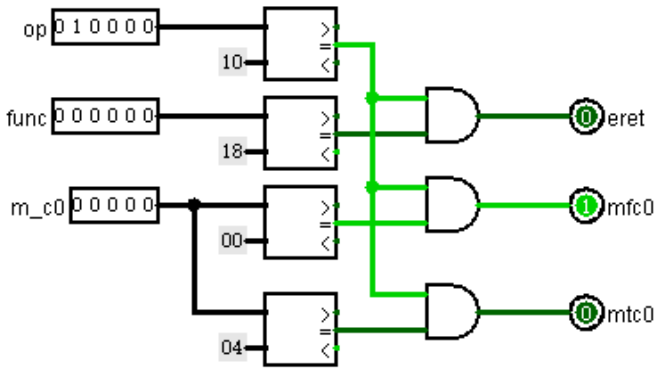


图 3.14 中断指令控制器

### 3.2.3 软件实现

在老师提供的中断演示程序的基础上增加 CCMB 程序以及中断处理程序。以 1 号中断的中断处理程序为例，汇编代码如下。

```
interrupt1_func:
    mfc0 $s1,$zero
    addi $sp, $zero, 64
    addi $sp, $sp, -4
    sw     $s0, 0($sp)
    addi $sp, $sp, -4
    sw     $s3, 0($sp)
    addi $sp, $sp, -4
    sw     $t0, 0($sp)
    addi $sp, $sp, -4
    sw     $t1, 0($sp)
    addi $sp, $sp, -4
    sw     $v0, 0($sp)

    addi $s0,$zero,1
    addi $s3,$zero,0

    addi $t0,$0,8
    addi $t1,$0,1
    int1_left:
    sll $s3, $s3, 4
    or $s3, $s3, $s0

    add       $a0,$0,$s3          # display $s3
    addi     $v0,$0,34             # system call for LED display
    syscall                        # display
    sub $t0,$t0,$t1
    bne $t0,$0,int1_left

    lw   $v0, 0($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    $s0, 0($sp)
addiu      $sp, $sp, 4
eret
mfc0 $s1, $1
```

首先实现了保护现场，通过 sw 指令将程序入口地址压栈，再将中断演示中用到的寄存器保存起来。通过 mfc0 指令实现关中断。Int1_left 处的代码实现了中断程序执行时的跑马灯效果。后面通过 lw 指令恢复现场，继续当前指令的执行。最后用 eret 指令返回并开中断。

## 3.3 流水 CPU 实现

### 3.3.1 流水接口部件实现

以 IF/ID 流水接口部件为例，流水接口部件主要需要时钟信号，同步清零信号以及使能端信号。使能端信号为高电平触发。清零信号用于选择输入的需要传递到下一个信号的值或者 0 给到寄存器的输入端。当使能端信号为 0 时，这个值写入到寄存器。输出现在寄存器内的值。IF/ID 流水接口部件的实现如图 3.15 所示。

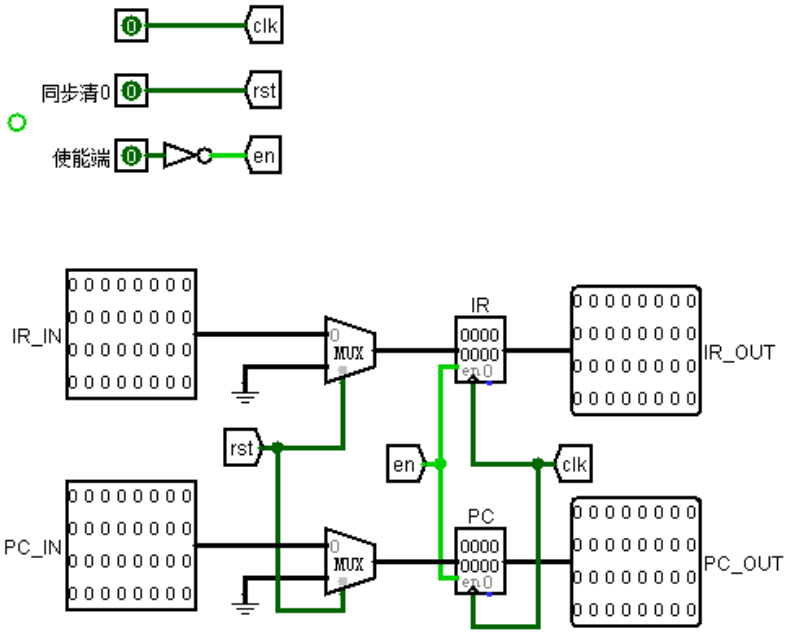


图 3.15 IF/ID 流水接口部件

IF/ID 阶段需要传递的数据很少，只有 PC 和 IR 需要传递，这两个数据在所有的流水接口部件都需要暂存。ID/EX 阶段需要传递除了 RegDst 信号之外所有控制器产生的信号，还需要传递寄存器文件读写时的寄存器编号 R1#、R2#、W#编号用于判断数据相关。EX/MEM 阶段需要传递写入的寄存器编号 W#，与访存和写回相关的信号 SH, RegWrite, JAL, MemWrite, MemToReg, 用于显示数据的寄存器 R2 的值以及和停机相关的信号。MEM/WB 阶段需要传递数据存储器输出的值，RegWrite,JAL 信号，写入的寄存器编号 W#，寄存器 R2 的值以及和停机相关的信号。

### 3.3.2 理想流水线实现

修改原来的单周期 CPU，将原来指令的执行过程划分为 IF、ID、EX、MEM、WB 五个阶段，根据流水线的设计以及流水接口部件的具体实现连接信号得到理想流水线 CPU 如图 3.16 所示。

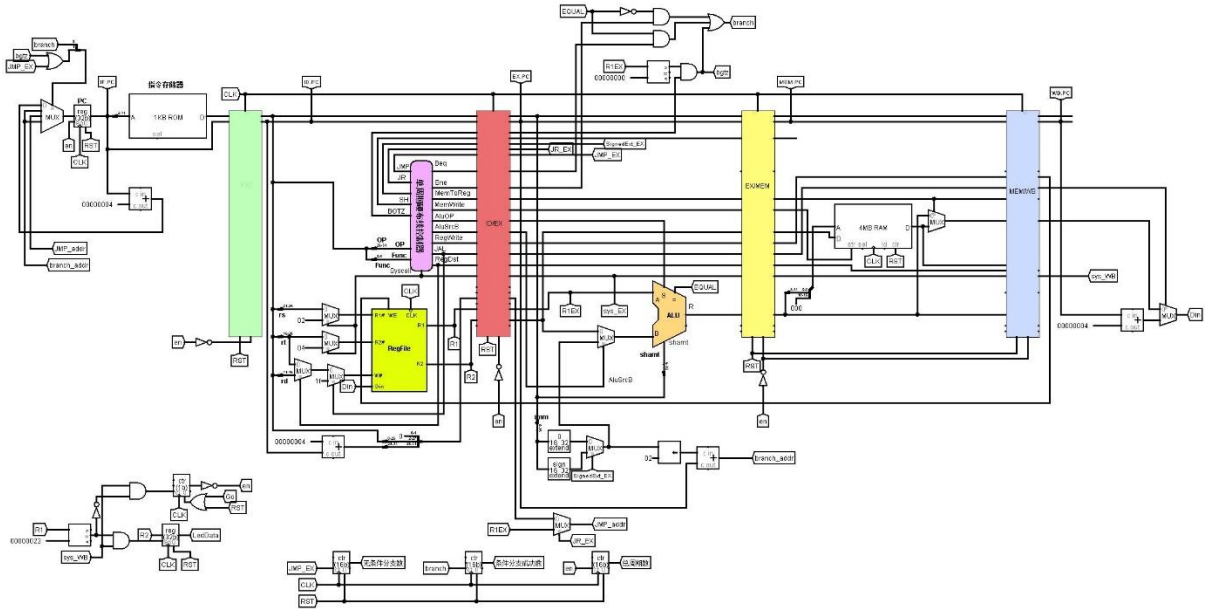

图 3.16 理想流水线 CPU

## 3.4 气泡流水线实现

气泡流水线需要判断是否出现数据相关的情况。首先根据指令判断寄存器 R1 和 R2 是否被使用，根据 excel 生成的表达式直接用分析电路生成源寄存器使用情况的电路图。

数据相关检测逻辑是当 ID 段当前指令读寄存器与后续两条指令写的寄存器相同的时候出现数据相关。根据源寄存器使用情况和指令的读写情况判断数据相关，电路实现如图 3.17 所示。

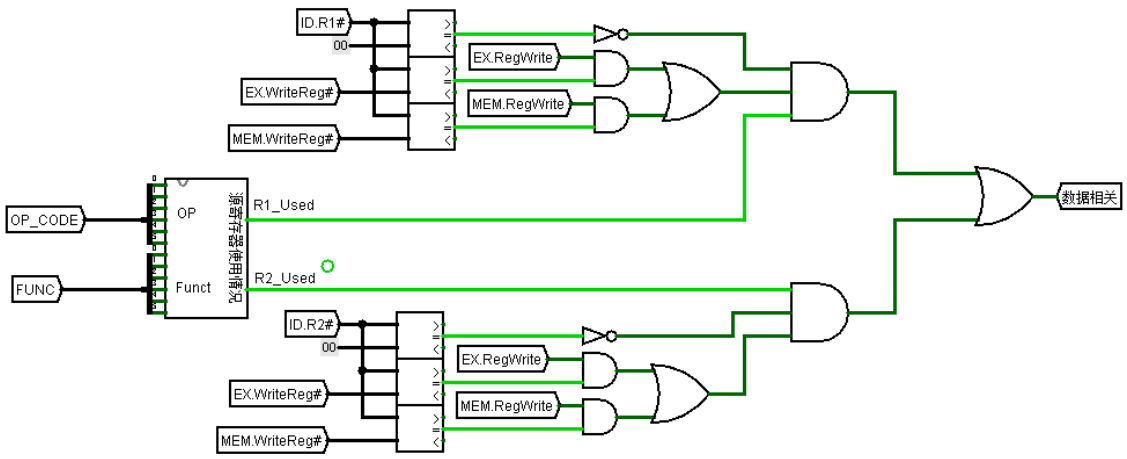


图 3.17 数据相关检测

根据气泡流水线的设计，修改 syscall 模块以及流水线接口部件的信号，使得当

出现数据相关的时候流水线插入气泡。另外注意分支指令的计算在 EX 段。连接电路得到气泡流水线如图 3.18 所示。

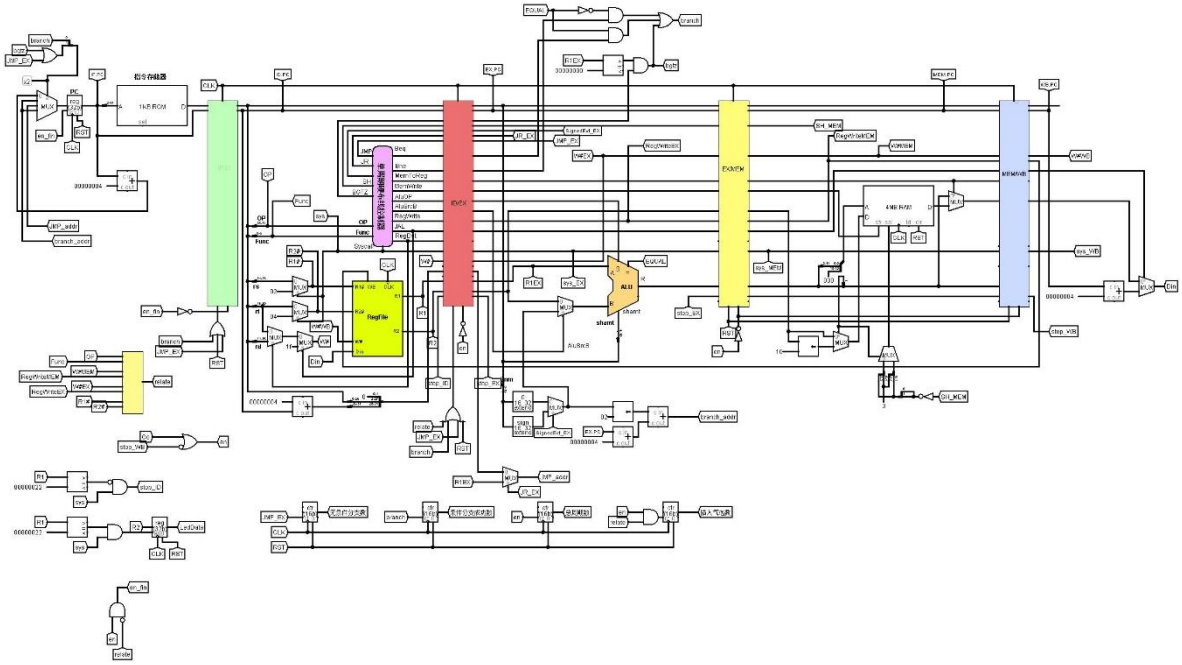

图 3.18 气泡流水线 CPU

## 3.5 数据转发流水线实现

重定向流水线中，只有在出现 load-use 相关时才需要插入气泡。添加 load-use 相关检测电路如图 3.19 所示。

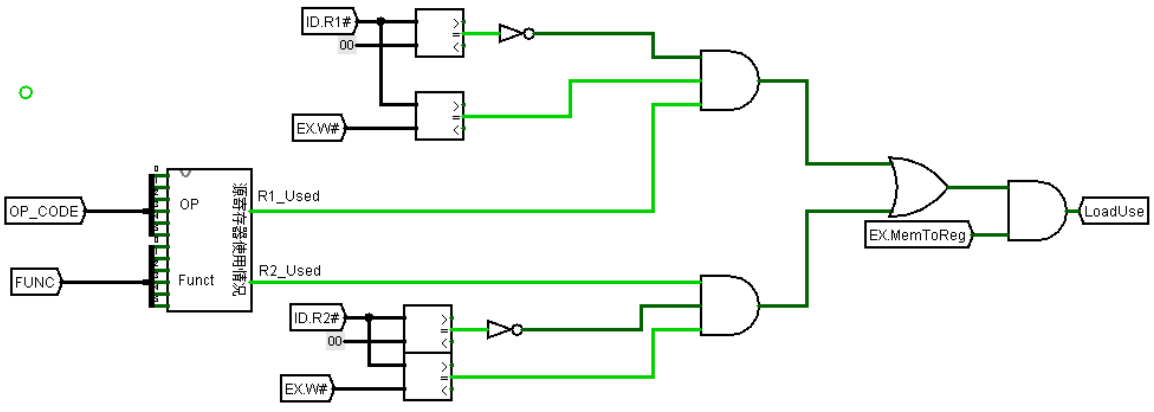


图 3.19 load-use 相关检测

需要根据数据冲突的情况选择需要重定向的数据。判断 ID 段的指令与 EX 段、MEM 段的指令的目的操作数是否相同，还需要根据冲突的寄存器是 R1 还是 R2 输出 ALU 的输入 A 和 ALU 的输入 B。数据可能来自于 EX 段的 R1 或者 MEM 段的 ALU

计算结果或者 WB 段写回寄存器堆的值。根据上述判断逻辑完成 ALU 重定向电路如图 3.20 所示。

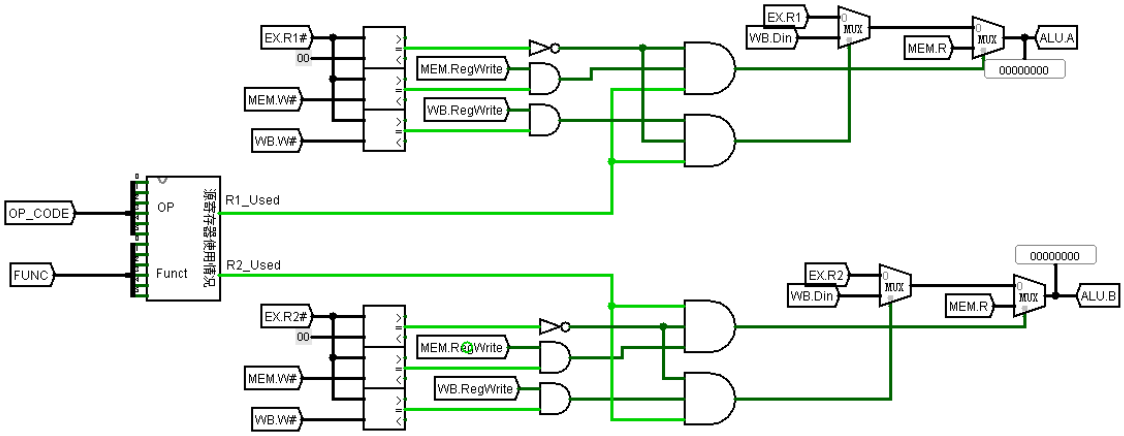


图 3.20 ALU 重定向

同样根据重定向流水线的设计，修改气泡流水线的流水部件接口信号以及 ALU 输入信号，连接得到重定向流水线电路图如图 3.21 所示。

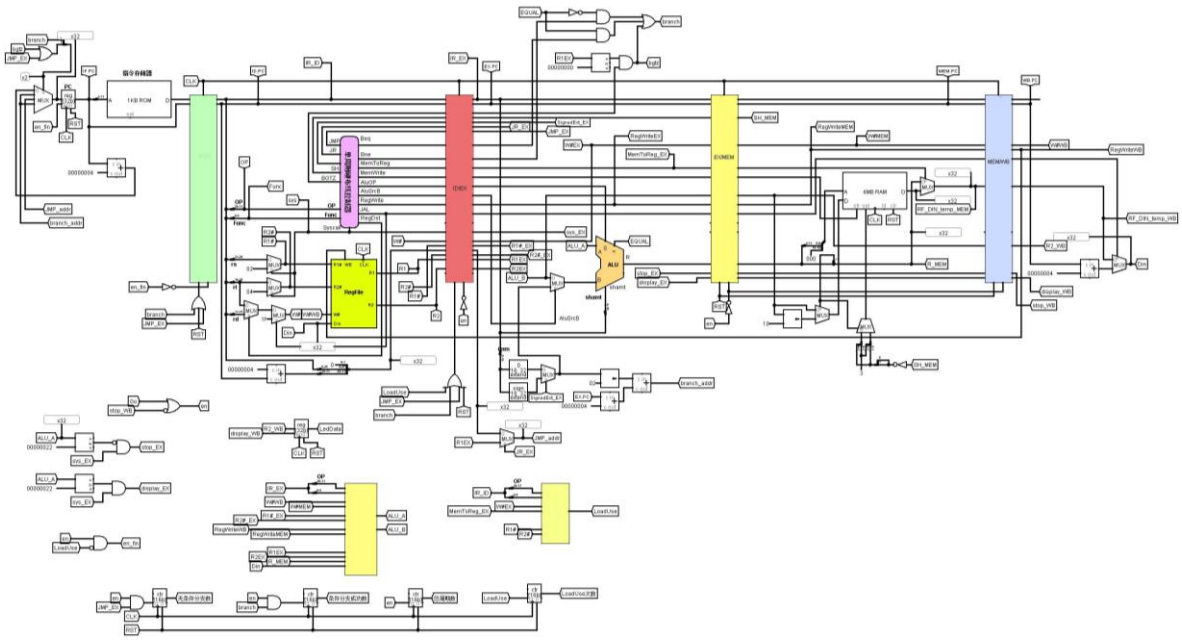

图 3.21 ALU 重定向流水线

# 4 实验过程与调试

## 4.1 测试用例和功能测试

### 4.1.1 单周期 CPU

单周期 CPU 在 FPGA 开发板上的显示和计数和 logisim 一致，因为现在没有开发板的拍照记录。用 logisim 单周期 CPU 加载 branchmark_ccmb 测试程序并运行的最终结果如图 4.1 所示。

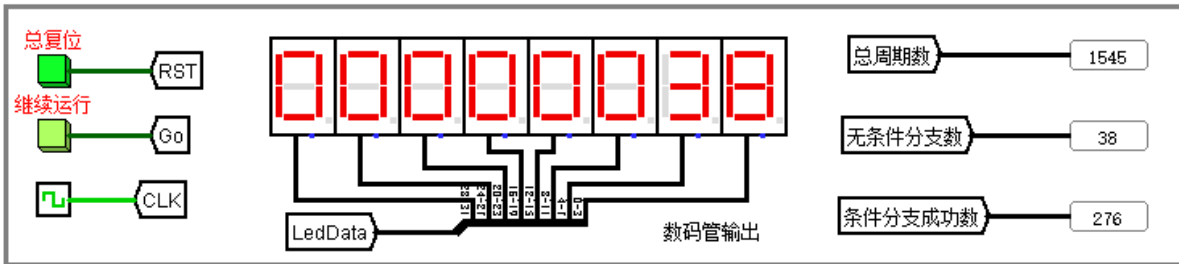


图 4.1 branchmark_ccmb 运行结果

ccmb 单条指令测试如下

XOR 测试结果如图 4.2 和 4.3 所示。

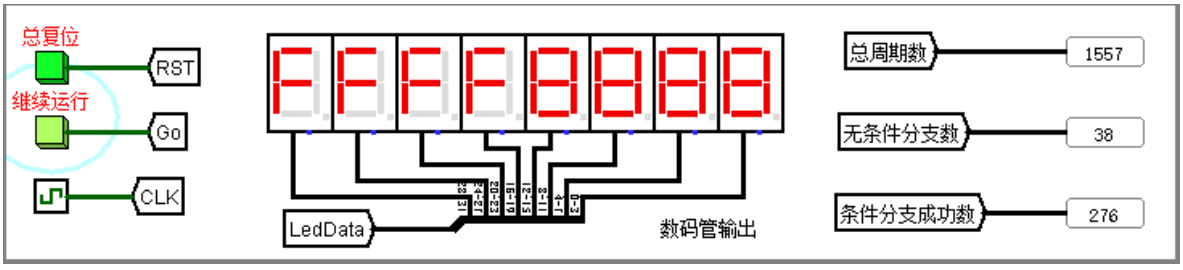


图 4.2 XOR 测试结果 1

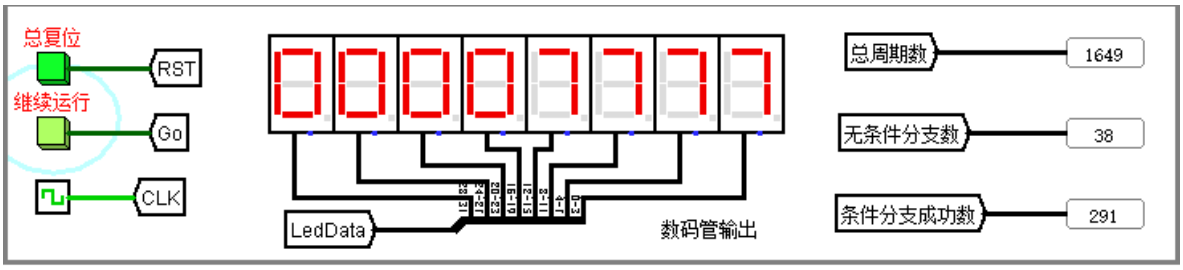


图 4.3 XOR 测试结果 2

XORI 测试结果如图 4.4 和 4.5 所示。

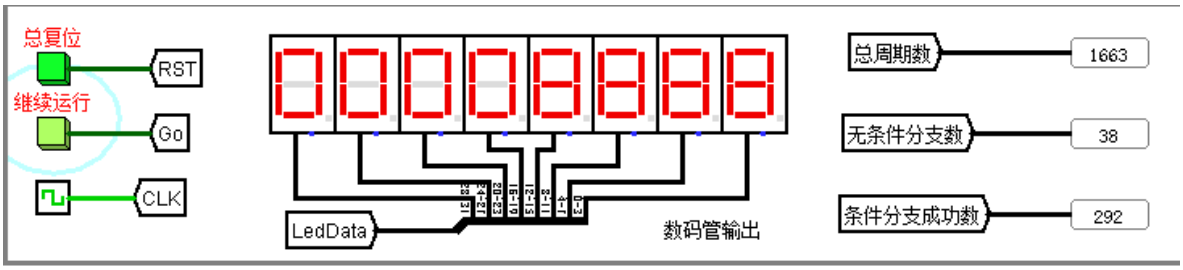


图 4.4 XORI 测试结果 1

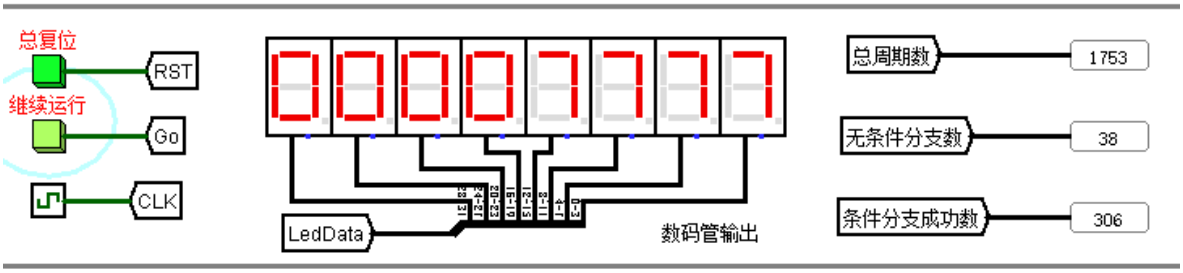


图 4.5 XORI 测试结果 2

SH 测试结果如图 4.6 所示。

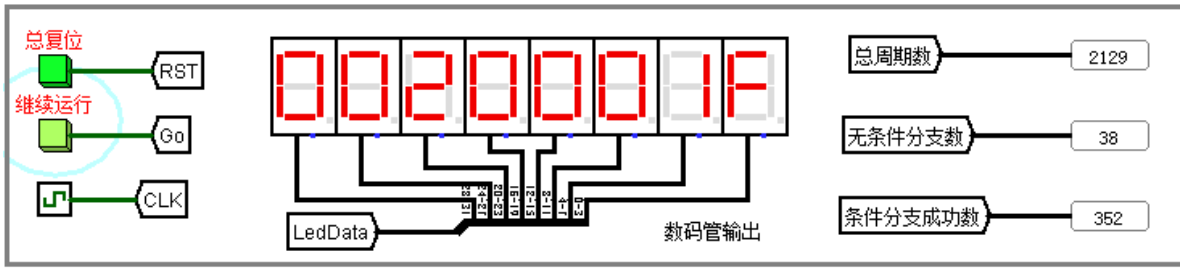


图 4.6 SH 测试结果

BGTZ 测试结果如图 4.7 所示。

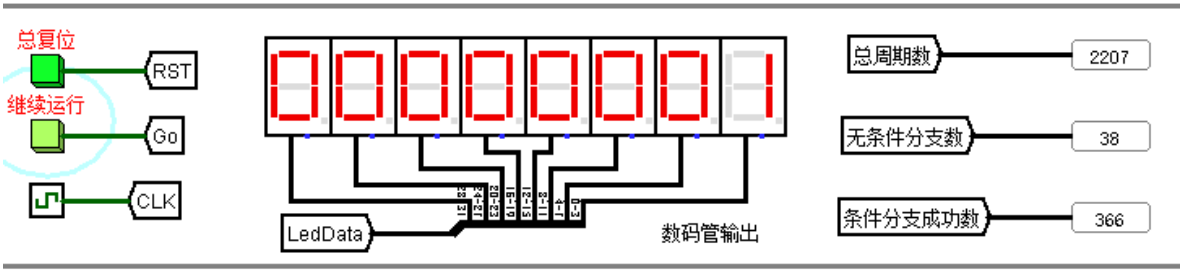


图 4.7 BGTZ 测试结果

### 4.1.2 理想流水线测试

加载理想流水线测试程序的 hex 文件到指令存储器，运行结果周期数为 21 个周期，数据存储器中写入 0x0,0x1,0x2,0x3。测试结果如图 4.8 和 4.9 所示。

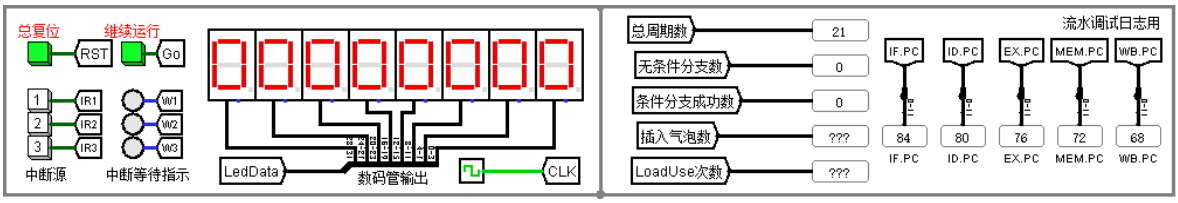


图 4.8 理想流水线测试总周期数


图 4.9 理想流水线测试数据存储器内容

### 4.1.3 气泡流水线测试

加载 benchmark 演示程序并运行的最终结果如图 4.10 所示，得到的总时钟周期数和预期一致。

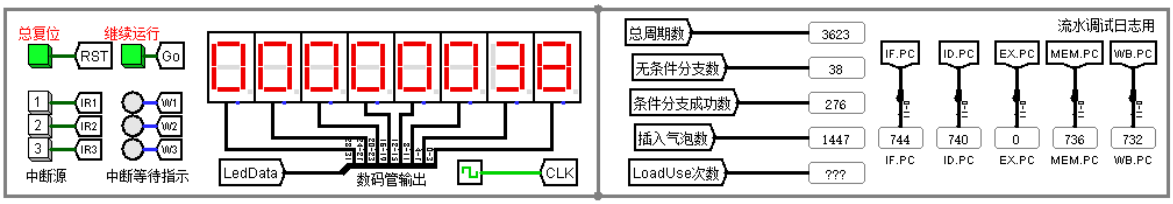


图 4.10 气泡流水线 branchmark_ccmb 运行结果

### 4.1.4 重定向流水线测试

加载 benchmark 演示程序并运行的最终结果如图 4.11 所示，得到的总时钟周期数和预期一致。

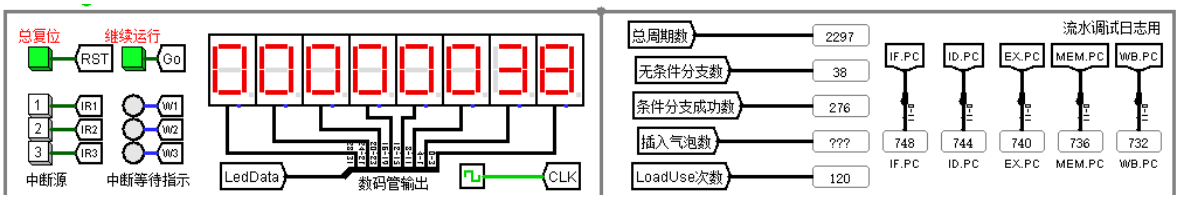


图 4.11 重定向流水线 branchmark_ccmb 运行结果

### 4.1.5 中断测试

按照 2 号、1 号、3 号中断的顺序按下中断源，单级中断因为不允许中断嵌套因此会首先执行 2 号中断程序，由于 3 号中断的优先级高于 1 号中断，将会继续执行 3 号中断，最后执行 1 号中断。测试结果如图 4.12、4.13 和 4.14 所示。

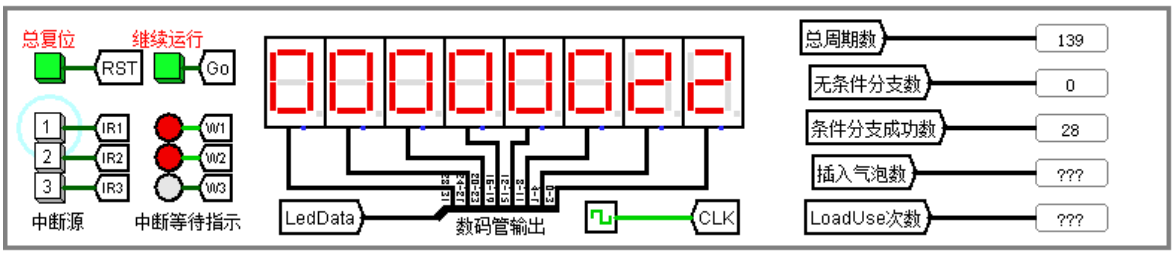


图 4.12 执行 2 号中断

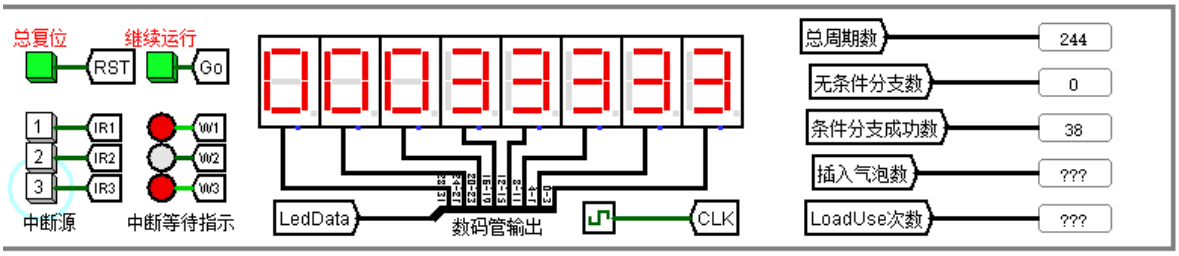


图 4.13 执行 3 号中断

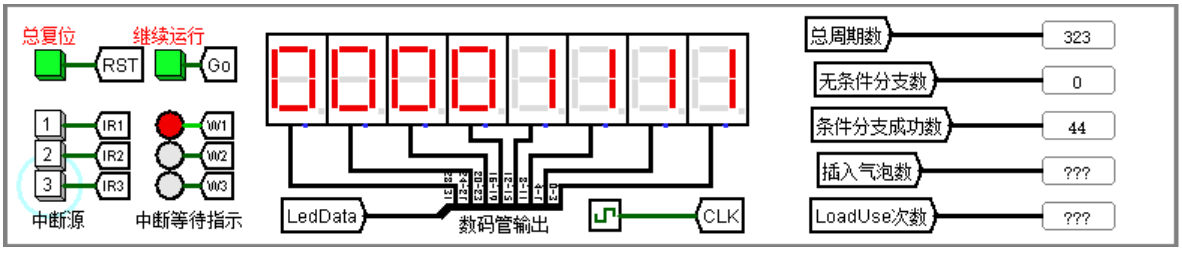


图 4.14 执行 1 号中断

### 4.1.6 重定向流水线上板测试

因为没有开发板的拍照记录，已经给老师检测通过了，所以这里没有图。实验中

测试的结果和重定向流水测试的周期数一致。

## 4.2 性能分析

单周期 MIPS CPU 的时钟周期数为 1545。

气泡流水线的时钟周期数为 3623，插入气泡数为 1447。

重定向流水线的时钟周期数为 2297，Load-use 次数为 120。

从总周期的比较可以看出来，重定向流水线相对于气泡流水线性能提升了很多，减少了气泡插入次数。

## 4.3 主要故障与调试

### 4.3.1 停机故障

理想流水线： syscall 阶段错误。

**故障现象：**指令执行停机之后的时钟周期数为 19。

**原因分析：**因为停机指令 syscall 解析出来是在 EX 阶段，但是为了不影响后续指令的执行，应该用 WB 阶段的 syscall 指令控制停机。否则 MEM 和 WB 阶段的指令没有执行完毕。虽然不影响本次实验的内存写入，但有可能在 IM 为其他代码的时候有影响。

**解决方案：**将 syscall 指令传递至 WB 段，再进行 syscall 的解析。修改后的 syscall 控制器如图 4.1 所示

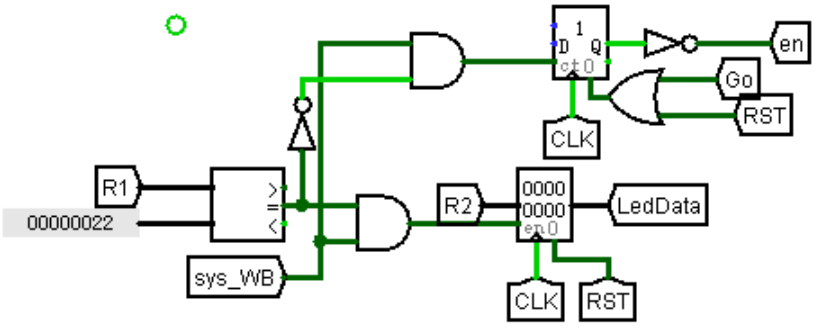


图 4.1 syscall 控制器

### 4.3.2 分支指令执行故障

BGTZ 指令执行不对。

**故障现象：**执行完成后，BGTZ 指令进入死循环。

**原因分析：**BGTZ 指令的判断放在了 ID 段，其他指令跳转放在了 EX 段，并且分支指令地址的计算在 EX 段。 ID 段判断的 BGTZ 指令没有经过流水接口部件直接用来判断跳转导致错误。

**解决方案：**将 BGTZ 指令的判断放在 EX 段。

### 4.3.3 流水上板分支指令故障

无法正确跳转。

**故障现象：**跳转的地址和预期的地址不一致。

**原因分析：**分析地址计算模块发现代码编写错误。直接使用了单周期 CPU 编写的 npc 模块，但是实际上没有发生分支用到的 PC 和分支时用到的 PC 不是同一个阶段的 PC。

**解决方案：**修改下址计算 npc 模块。添加一个 PC 接口并修改数据通路。

## 4.4 实验进度

表 4.1 课程设计进度表

| 时间 | 进度 |
|---|---|
| 第一天 | 根据上学期的单周期 CPU，课设任务书以及 MIPS 指令集，完善寒假完成的 24 条指令 CPU，扩展至 28 条指令。 |
| 第二天 | 开始写单周期 CPU 的 FPGA 实现部分模块代码。 |
| 第三天 | 完成了代码，分模块进行测试，开始进行数据通路的连接。 |
| 第四天 | 完成了单周期 CPU 的 FPGA 实现。 |
| 第五天 | 学习流水线的相关知识，开始设计流水线 CPU 的数据通路。 |
| 第六天 | 理想流水线设计调试完成。开始做气泡流水线。 |
| 第七天 | 气泡流水线和重定向流水线完成。 |

| 时间 | 进度 |
| --- | --- |
| 第八天 | 复习中断相关知识，开始做单级中断。 |
| 第九天 | 单级中断调试成功，开始修改用来的 verilog 代码做流水上板。 |
| 第十天 | 流水上板成功。 |

# 5 设计总结与心得

## 5.1 课设总结

本次课设完成了单周期 MIPS CPU 的 logisim 和 FPGA 开发板实现、理想流水线、气泡流水线、重定向流水线、单级中断以及流水线的 FPGA 开发。作了如下几点工作：

1） 完成单周期 MIPS CPU、单级中断以及流水线的设计。

2） 在 logisim 平台上完成单周期 MIPS CPU、单级中断以及三种流水线的实现。

3） 和小组同学一起完成了单周期 CPU 上板，独立完成了重定向流水线上板。

## 5.2 课设心得

本次课程设计的难度比较大，在实验后期我没有拿到通关的成绩也比较遗憾。因为后面流水上板出现了很多因为写的代码不够规范导致的 bug，修改了很长时间。课程设计是分组完成的，单周期 CPU 上板部分是一起完成的，后面的部分也有经过讨论的。虽然没有坚持做完，但还是在这个过程中学到了很多东西。

第一部分 Logisim 完成在寒假和刚开学的时候，这一部分的任务是对上学期单周期硬布线控制 CPU 的一个扩展，增加了一些新的控制信号以支持新的命令。这一部分的实现还比较简单。然后是小组合作上板，因为我们分工编写模块代码，每个人的编码习惯不同，实现的单周期 MIPS CPU 的数据通路也有少许的不同，在连接数据通路的时候花了很长时间 debug。出现了很多低级错误，甚至一些错误是在写代码的时候可以避免的，比如接口不匹配，A 信号连到了 B 这种错误。

第二部分是流水线，这一部分的内容之前没有讲过，需要看视频和资料学习。最开始做理想流水线困难在于不知道该如何修改用来的单周期 CPU。我最初是想在之前的 CPU 上直接添加流水接口部件，开始做了之后发现这样会导致线非常混乱，而且会出现连线的错误，流水部件也会很混乱。最后我分段设计流水线，根据它们各个阶段需要的信号以及单周期 CPU 的设计图修改完成。气泡流水线在出现冲突的时候需要暂停 PC 插入气泡，因此需要修改理想流水线里流水接口部件的使能端以及 PC 的使能端。另外为了处理 ID 和 WB 段的冲突，需要修改 RegFile 的触发方式。添加

了冲突检查之后这一部分基本上完成。重定向流水线也比较简单，删除前面的数据冲突检测然后根据在根据信号在需要重定向的地方加上 MUX 选择即可。理解了流水线的执行过程之后这一部分还完成的比较快。

选作部分我选择了流水上板和单级中断，最开始做的单级中断就是简单的按顺序进行中断响应，后来可以根据它们的优先级进行选择，多级中断需要保存前一个中断的地址，比较复杂，我没有做完。流水上板比较顺利，但还是因为写代码不规范导致了很多错误。基本上改完位数和接口就可以直接上板运行了。

这次课程设计比较困难，但是我还是完成了基础的工作，虽然很遗憾没有通关，我还是很感谢这样的课程设计，让我们学会了自己查找资料学习，学会了和别人合作。不仅巩固了之前的知识，而且流水线等扩展对后面的课程学习也非常有帮助。最后希望老师能录 mooc 讲解一下中断，分支预测等扩展部分。

# 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] 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:**