***2023***



**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： | 计算机21本硕博 |
| 学 号： | U202115674 |
| 姓 名： | 姚晨炫 |
| 电 话： | 15105919611 |
| 邮 件： | deadfffool@gmail.com |

目 录

[1 课程设计概述 3](#_Toc150518719)

[1.1 课设目的 3](#_Toc150518720)

[1.2 设计任务 3](#_Toc150518721)

[1.3 设计要求 3](#_Toc150518722)

[1.4 技术指标 4](#_Toc150518723)

[2 总体方案设计 6](#_Toc150518724)

[2.1 单周期CPU设计 6](#_Toc150518725)

[2.2 中断机制设计 10](#_Toc150518726)

[2.3 流水CPU设计 11](#_Toc150518727)

[3 详细设计与实现 13](#_Toc150518728)

[3.1 单周期CPU 实现 13](#_Toc150518729)

[3.2 中断机制实现 18](#_Toc150518730)

[3.3 流水CPU实现 21](#_Toc150518731)

[4 实验过程与调试 26](#_Toc150518732)

[4.1 测试用例和功能测试 26](#_Toc150518733)

[4.2 性能分析 27](#_Toc150518734)

[4.3 主要故障与调试 28](#_Toc150518735)

[4.4 实验进度 30](#_Toc150518736)

[5 团队任务部分 31](#_Toc150518737)

[5.1 总体设计 31](#_Toc150518738)

[5.2 开发平台 31](#_Toc150518739)

[5.3 软件设计 31](#_Toc150518740)

[5.4 Logisim硬件设计 32](#_Toc150518741)

[5.5 Verilog硬件设计 32](#_Toc150518742)

[5.6 团队分工 33](#_Toc150518743)

[5.7 开发中遇到的问题 33](#_Toc150518744)

[6 设计总结与心得 34](#_Toc150518745)

[6.1 课设总结 34](#_Toc150518746)

[6.2 课设心得 34](#_Toc150518747)

[参考文献 36](#_Toc150518748)

# 课程设计概述

## 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。

## 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。

## 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。

## 技术指标

1. 支持规定的32位RISC-V指令，具体见表格1.1；
2. 在CCAB扩展指令集中支持2条C类运算指令，1条M类存储指令，1条B类分支指令。
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。
8. 表 1.1 指令集

| **#** | **指令助记符** | **简单功能描述** | **备注** |
| --- | --- | --- | --- |
| 1 | ADD | 加法 | 指令格式与功能请参考 RISC-V32 指令集英文手册，或参考 RARS 模拟器 |
| 2 | ADDI | 立即数加 |
| 3 | AND | 与 |
| 4 | SLLI | 逻辑左移 |
| 5 | SRAI | 算数右移 |
| 6 | SRLI | 逻辑右移 |
| 7 | SUB | 减 |
| 8 | OR | 或 |
| 9 | ORI | 立即数或 |
| 10 | XORI | 或非/立即数异或 |
| 11 | LW | 加载字 |
| 12 | SW | 存字 |
| 13 | BEQ | 相等跳转 |
| 14 | ANDI | 立即数与 |
| 15 | BNE | 不相等跳转 |
| 16 | SLT | 小于置数 |
| 17 | SLTI | 小于立即置数 |
| 18 | SLTU | 小于无符号置数 |
| 19 | JAL | 转移并链接 |
| 20 | JALR | 转移到指定寄存器 |
| 21 | CSRRSI | 访问CSR寄存器 |
| 22 | CSRRCI | 访问CSR寄存器 |
| 23 | URET | 中断返回 | if ($a7==34) LED 输出$a0 的值 else 停机等待 Go 按键按下注意显示逻辑锁存住数据 |
| 24 | ECALL | 系统调用 |
| 25 | SRL | 逻辑右移 |  |
| 26 | LUI | 高位立即数加载 |  |
| 27 | SB | 低位字节写入内存 |  |
| 28 | BLT | 小于时分支跳转 |  |

# 总体方案设计

## 单周期CPU设计

本次单周期CPU设计我们采用程序存储器和数据存储器分开的设计，来保证一个周期内可以正确完成一条指令，通过硬布线控制器，我们可以设置好不同指令对应的控制信号与ALU的运算控制。

总体结构图如图 2.1所示。



图 2.1 总体结构图

### 主要功能部件

#### 程序计数器PC

程序计数器PC的思想是借助于寄存器，寄存器存入当前周期的PC的值输出端连接指令存储器，而输入端具有多种可能分别是分支于按序的下一条指令的地址，PC寄存器的使能至于ECALL指令有关，ECALL指令根据寄存器数据的不同作出print和halt的操作。

#### 指令存储器IM

指令存储采用rom存储器，提前将对应的程序写入存储器中，根据PC取出指令，利用分线器将不同字段送入不同部件，控制相关的字段送入硬布线程序控制器，寄存器相关字段送入寄存器堆，立即数字段通过相关扩展后送入ALU或者直接写入内存。

#### 运算器

运算器引脚及功能如下表2.1所示：

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 |
| Result | 输出 | 32 | ALU运算结果 |
| Result2 | 输出 | 32 | ALU结果第二部分，用于乘法指令结果高位或除法指令的余数位，其他操作为零 |
| OF | 输出 | 1 | 有符号加减溢出标记，其他操作为零 |
| UOF | 输出 | 1 | 无符号加减溢出标记，其他操作为零 |
| Equal | 输出 | 1 | Equal=(x==y)?1:0, 对所有操作有效 |

#### 寄存器堆RF

由多个寄存器组成的寄存器堆，通过rs，rt控制寄存器堆的读，通过Rin和W来控制寄存器堆的写。

### 数据通路的设计

数据通路设计如下图所示

A diagram of a computer

Description automatically generated

图 2.2 数据通路图

### 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 2.2。

表 2.2主控制器控制信号的作用说明

| 控制信号 | 取值 | 说明 |
| --- | --- | --- |
| MemtoReg | 0 | 不写回 |
| 1 | 数据存储器写回寄存器堆 |
| MemWrite | 0 | 数据存储器写禁止 |
| 1 | 数据存储器写允许 |
| ALU\_Src | 0 | 取寄存器堆中r2地址对应的内容 |
| 1 | 把立即数送入到ALU的输入 |
| RegWrite | 0 | 寄存器堆写不允许 |
| 1 | 寄存器堆写允许 |
| Ecall | 0 |  |
| 1 | 进行终止判断 |
| S\_TYPE | 0 | 不是S型指令 |
| 1 | 是S型指令 |
| BEQ | 0 | 非BEQ指令 |
| 1 | 是BEQ指令 |
| BNE | 0 | 非BNE指令 |
| 1 | 是BNE指令 |
| JAL | 0 | 非JAL指令 |
| 1 | 是JAL指令 |
| Result2 | 0 | 运算器结果为result1 |
| 1 | 结果为result2 |
| Memex | 0 | 数据寄存器内容输出不需要扩展 |
| 1 | 数据寄存器内容输出需要扩展 |
| pcimm | 0 | 分支关 |
| 1 | 分支开 |
| CSRRSI | 1 | 开中断启动 |
| 0 | 无 |
| CSRRCI | 1 | 关中断启动 |
| 0 | 无 |

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 2.3所示。

表 2.3主控制器控制信号框架

| 指令 | ALU\_OP | MemtoReg | MemWrite | ALU\_Src | RegWrite | ecall | S\_Type | BEQ | BNE | ALU\_OP | MemtoReg | … |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|
| add | 5 |  |  |  | 1 |  |  |  |  |  |  | … |

## 中断机制设计

### 总体设计

（1）单级中断：在单周期的数据通路与控制器不变的情况下，加入不同中断的响应装置，中断相关的寄存器，根据是否中断开始以及中断信号处理是否完成，加入相关的地址跳转信号。

（2）多级中断：增加多级中断的相关调度器，以及更改寄存器结构来适应多级中断中可能出现的更多断点地址。

### 硬件设计

（1）中断请求信号：硬件中断设计的核心是中断请求信号。使用程序包中的中断按键参考信号来实现中断请求信号。

（2）中断控制器：为了有效地管理多个中断请求，CPU通常会使用中断控制器。中断控制器负责接收中断请求，确定哪个中断应该被响应（优先级管理）以及通知CPU开始执行相应的中断服务例程。

（3）中断向量：每个中断都有一个唯一的标识号，称为中断向量。中断向量用于告诉CPU响应哪个中断。在多级中断的调度器中，使用优先编码器来调度。

（4）中断处理：一旦CPU确定要响应中断，它会保存当前的上下文（如寄存器状态）并跳转到中断服务例程的地址。在单周期CPU中，这通常涉及将中断服务例程的地址加载到程序计数器（PC）中，以便开始执行该例程。执行中断服务例程后，CPU通常会根据中断的类型进行一些额外的操作，如清除中断标志、保存中断发生时的寄存器状态等。

（5）中断返回：一旦中断服务例程完成，CPU需要恢复之前保存的上下文，以便从中断点继续执行。这包括恢复寄存器状态和重新设置PC。在我们的设计中，会根据控制信号加载EPC中寄存好的地址。

## 流水CPU设计

### 总体设计

理想流水线在设计时根据指令执行的特征分为了五个阶段，分别是取指阶段，译码阶段，运算阶段，访存阶段，写回阶段，在每个部分之间设置一个流水寄存器将整个过程都统一划分为5个时钟周期，从而保证了流水线的完整操作。

### 流水接口部件设计

流水线的接口均采用寄存器的方式便于传输和存储数据，寄存器负责传输相关的控制信号，IR，PC寄存器中的数据，以及读取和运算的数据内容。寄存器需要有统一的使能和复位信号，便于控制流水线的暂停和气泡。

### 理想流水线设计

理想流水线不考虑分支指令和读写数据的相关冲突，因此较为简单，只需要按序输入PC取出的指令即可。流水线中的每一个阶段对应的都是一条不同指令，理想流水线中没有气泡和重定向设计。

### 气泡流水线设计

与理想流水线不同的是，气泡流水线考虑相关的指令相关，指令相关指在指令流水线中，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后才能开始，也就是两条指令间存在某种依赖关系，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)，流水线冲突是指由于指令相关的存在，指令流水线出现“阻塞”或“暂停”，下一条指令不能在预期的时钟周期内加载到流水线中。流水线冲突包括结构冲突、控制冲突、数据冲突3种。

结构冲突是由于读写地址和数据均通过不同的结构冲突，但由于 RISC-V 寄存器堆的读写端口进入，读写逻辑可以并发操作，因此这种结构冲突并不存在。

分支指令会引起控制冲突，要解决控制冲突，在执行程序分支跳转时必须清除流水线中分支指令后续的若干条误取指令。

数据冲突时由于ID段取寄存器数据时会与EX段和MEM段存在相关的依赖关系，因此流水线中必须增加硬件逻辑来实现 ID 段与 EX、MEM 段指令的数据相关性检测。有了数据相关检测逻辑，当发生数据相关时，给ID/EX 流水寄存器一个同步清空信号 Flush即可；而要暂停IF、ID 段指令的执行，只需保证程序计数器 PC 和IF/ID流水寄存器的值不变即可。要做到这一点，只需要控制寄存器使能端即可，当使能端为1时，寄存器正常工作，为 0时则忽略时钟输入，寄存器值保持不变。

### 重定向流水线设计

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指令流水线的性能。重定向流水线的思路是先不考虑ID 段所取的寄存器操作数是否正确，等到指令实际使用这些寄存器操作数时再考虑其正确性问题。

重定向技术根据不同的冲突类型产生不同的控制信号，将需要更新的数据部分从流水线后段直接传入，不等待指令完全执行完毕，使得流水线的效率大大提高。

### 分支预测流水线设计

采用重定向机制后，指令流水线中的数据相关基本不需要插入气泡就可解决，只有少数Load-Use 相关还需要插入一个气泡，流水线性能得到极大的提升。此时流水线中的控制冲突对流水线性能影响最大，基于加快经常性事件的原理，应优先考虑减少分支指令引起的分支延迟损失。为减少分支延迟损失，应尽可能提前执行分支指令，例如将分支指令放在ID 段完成。动态分支技术根据跳转历史的准确率对跳转策略不断调整，提高预测准确率，因为分支具有局部性，比如程序中for循环和while循环，动态分支预测就是利用这种局部性来进行预测

# 详细设计与实现

## 单周期CPU 实现

1. 程序计数器（PC）

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，经过一个非门后与寄存器使能端相连，如图 3.1所示。

A computer screen shot of a computer

Description automatically generated

图 3.1程序计数器（PC）

FPGA实现：

程序计数器使用register寄存器的实例化，register的Verilog代码如下：

module register(CLK, RST, EN, Din, Dout);

parameter WIDTH = 32; // Register width, default is 32 bits

input CLK, RST, EN; // Clock, reset, and enable signals

input [WIDTH-1:0] Din; // Data input

output [WIDTH-1:0] Dout; // Data output

reg [31:0] ram; // Register

initial ram = 0; // Initialize the register to 0

always @(posedge CLK) begin

if (RST)

ram <= 0; // If reset signal is active, set register value to 0

else if (EN)

ram <= Din;

else

ram = ram; // Otherwise, keep the register value unchanged

end

assign Dout = ram; // Data output is the register value

endmodule

1. 程序计数器（PC）

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）

FPGA实现：

直接使用Vivado中自带的RAM，数据位宽为32位，地址位宽为4位。

指令存储器IM的Verilog代码如下：

module RegFile(Din, R1Adr, R2Adr, WAdr, WE, CLK, R1, R2);

input [31:0] Din; // Data input

input [4:0] WAdr, R1Adr, R2Adr; // Write address, R1 address, and R2 address

input WE, CLK; // Write enable and clock signal

output [31:0] R1, R2; // R1 and R2 outputs

reg [31:0] RAM[31:0]; // Register file, size is 32 registers

integer i;

initial begin

for (i = 0; i <= 31; i = i + 1) begin

RAM[i] = 0; // Initialize all registers to 0

end

end

assign R1 = RAM[R1Adr]; // R1 output is the value at the specified address

assign R2 = RAM[R2Adr]; // R2 output is the value at the specified address

always @(posedge CLK) begin

if (WE && WAdr != 0)

RAM[WAdr] <= Din;

end

endmodule

1. 硬布线控制器

Logisim实现

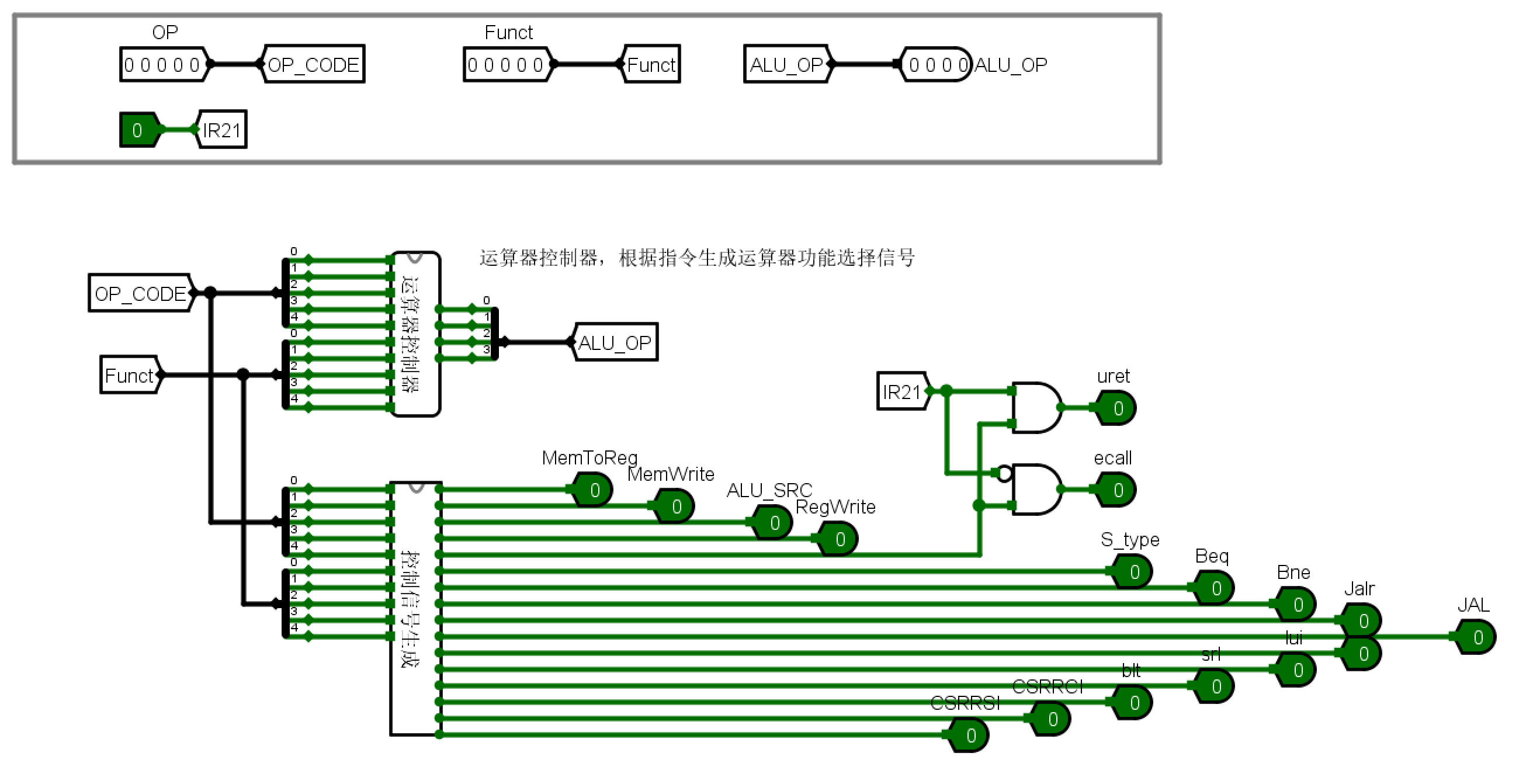


图 3.3硬布线控制器

硬布线控制器通过OP\_CODE和Funct来综合判断对应的控制型号和运算方式，运算器控制器和控制信号通过execl自动生成，如下图所示



图 3.4 控制信号生成

根据生成的逻辑表达式，利用Logisim电路仿真实现硬布线控制器的设计

FPGA实现

是用assign语句，如图3.3根据不同的指令情况分配不同的信号，一下是ALUOP的assign语句，控制信号同理。

assign ALUOP =(Funct7 == 7'b0000000 && Funct3 == 3'b000 && OpCode == 5'b01100) ? 4'b0101 : // add

(Funct7 == 7'b0100000 && Funct3 == 3'b000 && OpCode == 5'b01100) ? 4'b0110 : // sub

(Funct7 == 7'b0000000 && Funct3 == 3'b111 && OpCode == 5'b01100) ? 4'b0111 : // and

(Funct7 == 7'b0000000 && Funct3 == 3'b110 && OpCode == 5'b01100) ? 4'b1000 : // or

(Funct7 == 7'b0000000 && Funct3 == 3'b010 && OpCode == 5'b01100) ? 4'b1011 : // slt

(Funct7 == 7'b0000000 && Funct3 == 3'b011 && OpCode == 5'b01100) ? 4'b1100 : // sltu

(Funct3 == 3'b000 && OpCode == 5'b00100) ? 4'b0101 : // addi

(Funct3 == 3'b111 && OpCode == 5'b00100) ? 4'b0111 : // andi

(Funct3 == 3'b110 && OpCode == 5'b00100) ? 4'b1000 : // ori

(Funct3 == 3'b100 && OpCode == 5'b00100) ? 4'b1001 : // xori

(Funct3 == 3'b010 && OpCode == 5'b00100) ? 4'b1011 : // slti

(Funct7 == 7'b0000000 && Funct3 == 3'b001 && OpCode == 5'b00100) ? 4'b0000 : // slli

(Funct7 == 7'b0000000 && Funct3 == 3'b101 && OpCode == 5'b00100) ? 4'b0010 : // srli

(Funct7 == 7'b0100000 && Funct3 == 3'b101 && OpCode == 5'b00100) ? 4'b0001 : // srai

(Funct3 == 3'b010 && OpCode == 5'b00000) ? 4'b0101 : // lw

(Funct3 == 3'b010 && OpCode == 5'b01000) ? 4'b0101 : // sw

(Funct3 == 3'b000 && OpCode == 5'b11000) ? 4'b1011 : // beq

(Funct3 == 3'b001 && OpCode == 5'b11000) ? 4'b1011 : // bne

(OpCode == 5'b11011) ? 4'b0101 : // jal

(Funct3 == 3'b000 && OpCode == 5'b11001) ? 4'b0101 : // jalr

4'b0000; //default

Endmodule

1. ALU

Logisim设计

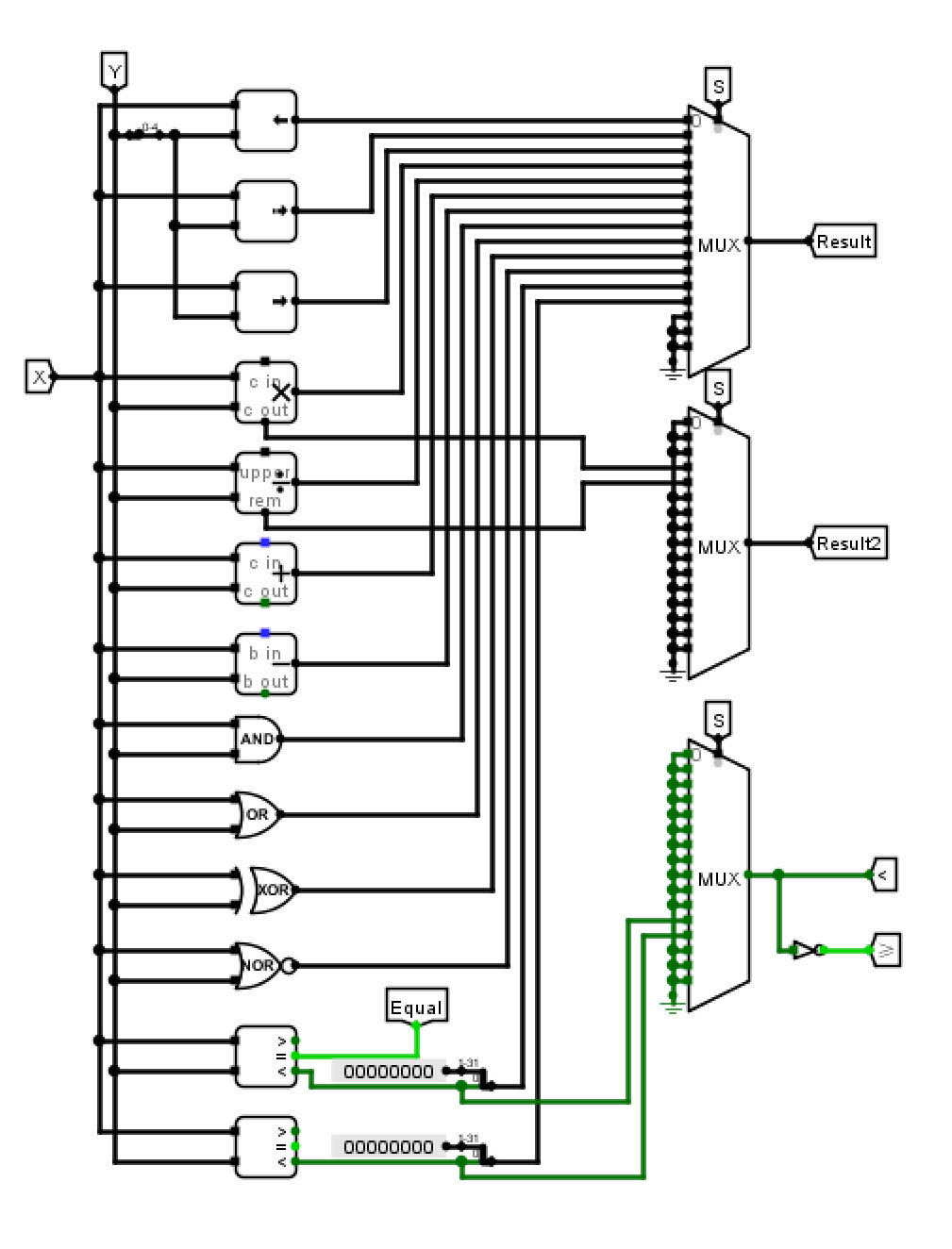


图 3.5 ALU设计电路图

ALU设计中，X，Y作为输入，根据ALUOP控制信号的选择不同设置不同的运算器，然后输出运算结果

FPGA实现

与硬布线相似，根据不同的输入，利用assign语句来选择不同的运算器，使用组合逻辑和线网形。

assign ALU\_result = (ALU\_op == 4'b0000)? x<<y[4:0]:

(ALU\_op == 4'b0001)? $signed($signed(x)>>>y[4:0]):

(ALU\_op == 4'b0010)? x>>y[4:0]:

(ALU\_op == 4'b0011)? x\*y[31:0]:

(ALU\_op == 4'b0100)? x/y:

(ALU\_op == 4'b0101)? x+y:

(ALU\_op == 4'b0110)? x-y:

(ALU\_op == 4'b0111)? x&y:

(ALU\_op == 4'b1000)? x|y:

(ALU\_op == 4'b1001)? x^y:

(ALU\_op == 4'b1010)? ~(x|y):

(ALU\_op == 4'b1011)? ($signed(x)<$signed(y))?1:0:

(ALU\_op == 4'b1100)? ((x<y)?1:0):

32'b0;

## 中断机制实现

1. 中断按键

Logisim设计

通过D触发器，根据当前中断请求输出下一个周期的中断信号，同时可以根据复位型号来清空中断信号，采用LED灯来指示中断。



图 3.6 中断按键设计电路图

FPGA实现

module Interrupter(

input CLK,IR,uret,

output interrupt,LED

);

wire D1out,D2in;

assign LED = interrupt | D1out;

assign D2in = (~uret) & ( D1out | interrupt);

DFlipFlop D1(.CLK(IR),.RST(interrupt),.d(1'b1),.q(D1out));

DFlipFlop D2(.CLK(CLK),.RST(1'b0),.d(D2in),.q(interrupt));

Endmodule

1. 中断调度器

采用优先编码器（Pri）根据中断信号到来的顺序和优先级我们可以判断出要执行那个中断程序以及译码出其中断地址，同时根据是否有uret信号从而使对应的中断信号消失。

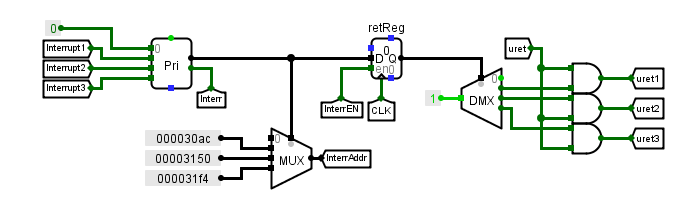


图 3.7 中断调度器电路图

FPGA实现

使用优先编码器，使用assign语句根据输入的不同来切换到不同的中断地址。

module Scheduler(IRR1,IRR2,IRR3,IRR4,IRR5,interr,uret1,uret2,uret3,uret4,uret5,uret,interrEN,CLK,interrAddr);

input IRR1,IRR2,IRR3,IRR4,IRR5;

input interrEN,CLK,uret;

output interr;

output uret1,uret2,uret3,uret4,uret5;

output [31:0]interrAddr;

wire [2:0]choice,uret\_choice;

wire [4:0]urettmp;

PriorityEncoder P(.inputs({IRR5,IRR4,IRR3,IRR2,IRR1}),.interr(interr),.encoded(choice));

//Interrupt handler Address

mux8 mux(.out(interrAddr), .in0(32'h30ac),.in1(32'h3150),.in2(32'h31f4),.in3(32'h30ac),.in4(32'h30ac),.in5(32'h30ac),.in6(32'h0),.in7(32'h0),.sel(choice));

register #(3) retReg(.CLK(CLK),.EN(interrEN),.Din(choice),.Dout(uret\_choice),.RST(0));

assign urettmp = (uret\_choice == 3'b000) ? 5'b00001:

(uret\_choice == 3'b001) ? 5'b00010:

(uret\_choice == 3'b010) ? 5'b00100:

(uret\_choice == 3'b011) ? 5'b01000:

(uret\_choice == 3'b100) ? 5'b10000:

5'b00000; //defualt

assign uret1 = urettmp[0]&uret;

assign uret2 = urettmp[1]&uret;

assign uret3 = urettmp[2]&uret;

assign uret4 = urettmp[3]&uret;

assign uret5 = urettmp[4]&uret;

endmodule

1. 多级中断中的断点存储

通过中断信号的不同，将断点地址存入不同的寄存器中，便于返回时候获取。

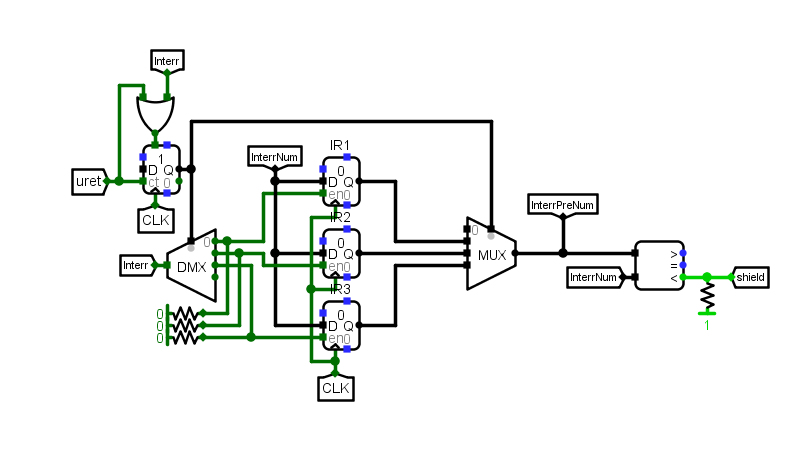


图 3.8 多级中断EPC电路图

## 流水CPU实现

### 流水接口部件实现

普通流水接口均采用多个寄存器的并发互联，包括PC,IR以及众多的控制信号和要传输的数据，采用寄存器的形式进行实现和保存。在不同指令阶段中插入流水线接口部件来实现不同阶段控制信号的传递，IF/ID的接口部件如下图所示

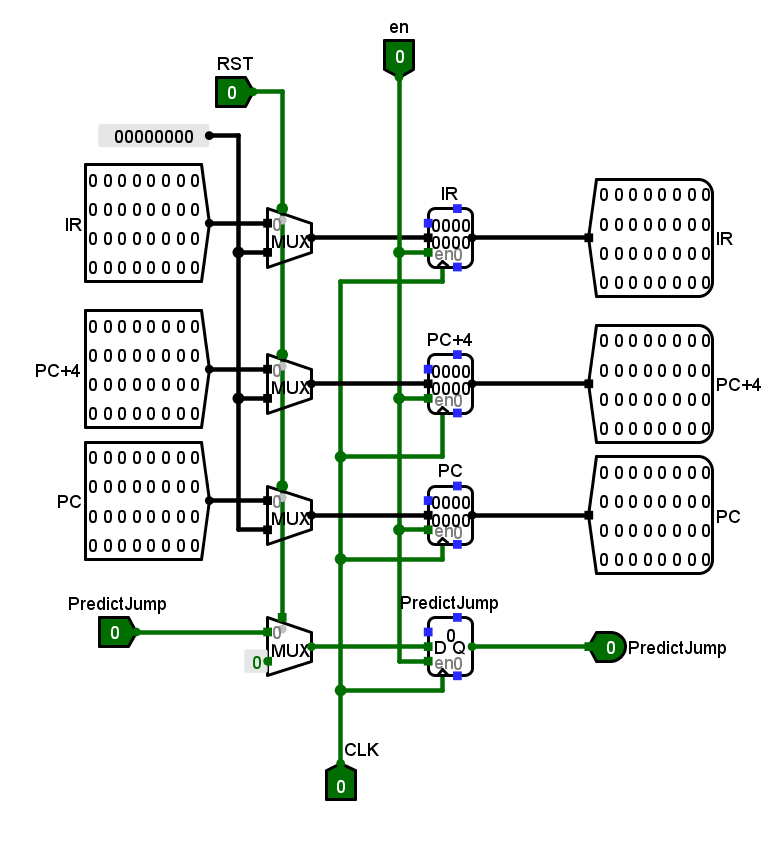


图 3.9 IF/ID接口电路图

### 理想流水线实现

理想流水线就是采用流水接口封装，将指令分为IF，IR，EX，MEM，WE五个阶段。电路图如下。

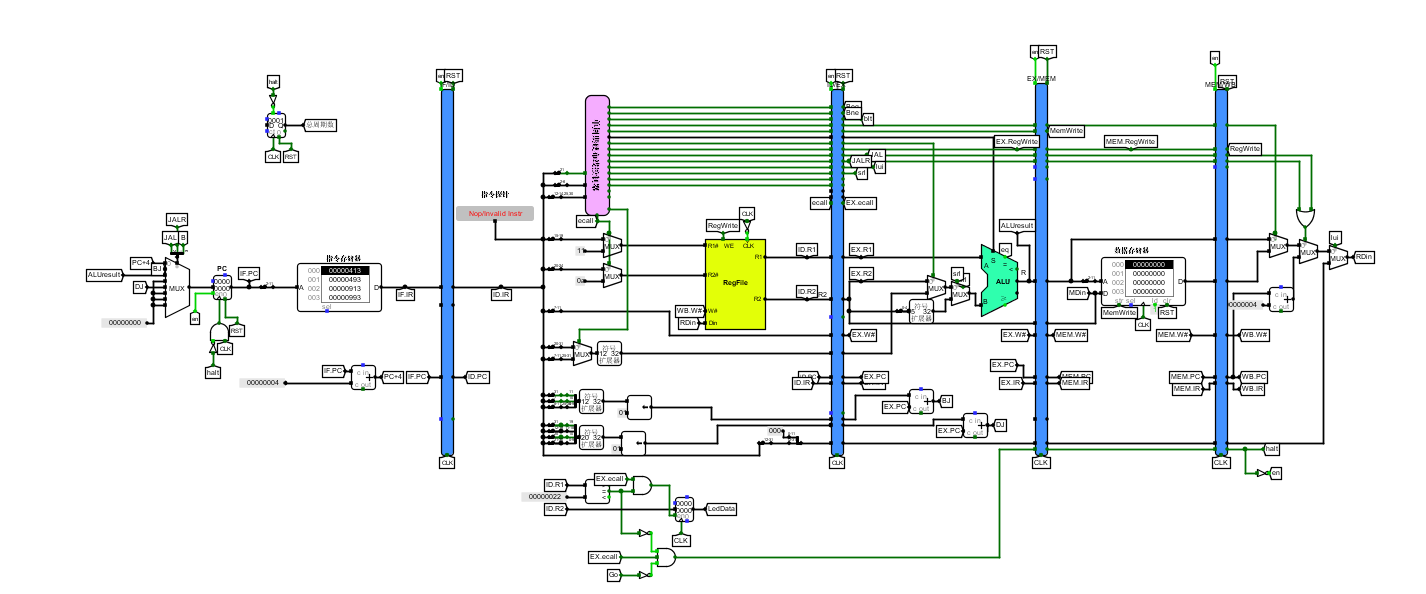


图 3.10 理想流水线电路图

### 气泡流水线实现

前文已经描述过气泡流水线中的load-use，可以通过插入气泡来解决这个问题，我们使用数据相关检测逻辑来判断是否需要插入气泡，如下图所示。

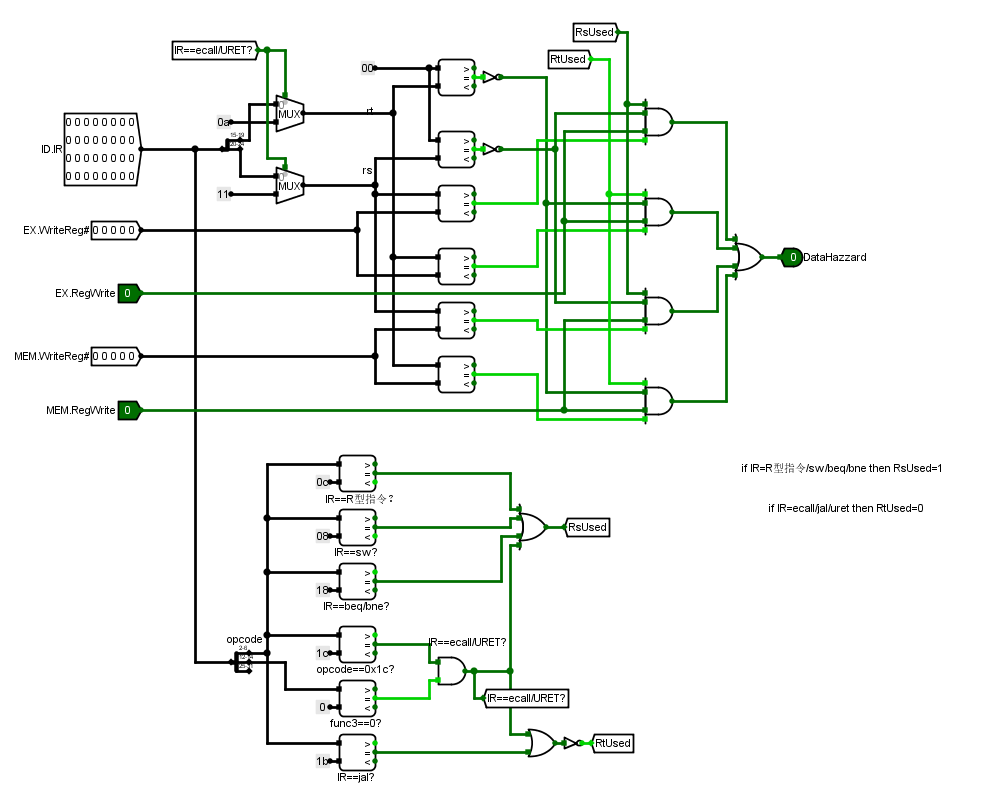


图 3.11 数据相关检测逻辑电路图

其对应的相关逻辑为

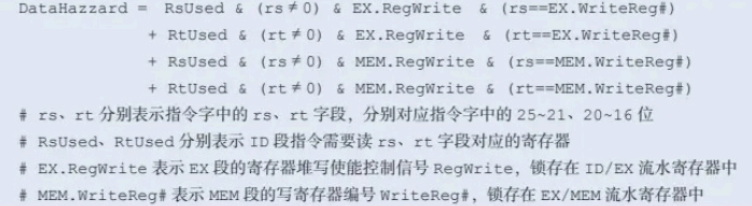


图 3.12 数据冲突表达式

在理想流水线中插入数据相关检测逻辑即可完成。

### 重定向流水线实现

前文已经介绍过重定向流水线所面临的问题，因此我们在EX阶段插入多路选择器，根据不同的冲突情况来复用之前代码中的数据或地址，如下图所示

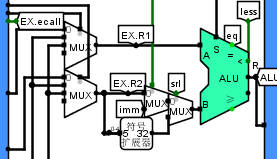


图 3.13 重定向中的多路选择

因此我们需要重定向检测逻辑来产生控制信号，使得多路选择器可以正确的传输数据，电路图如下

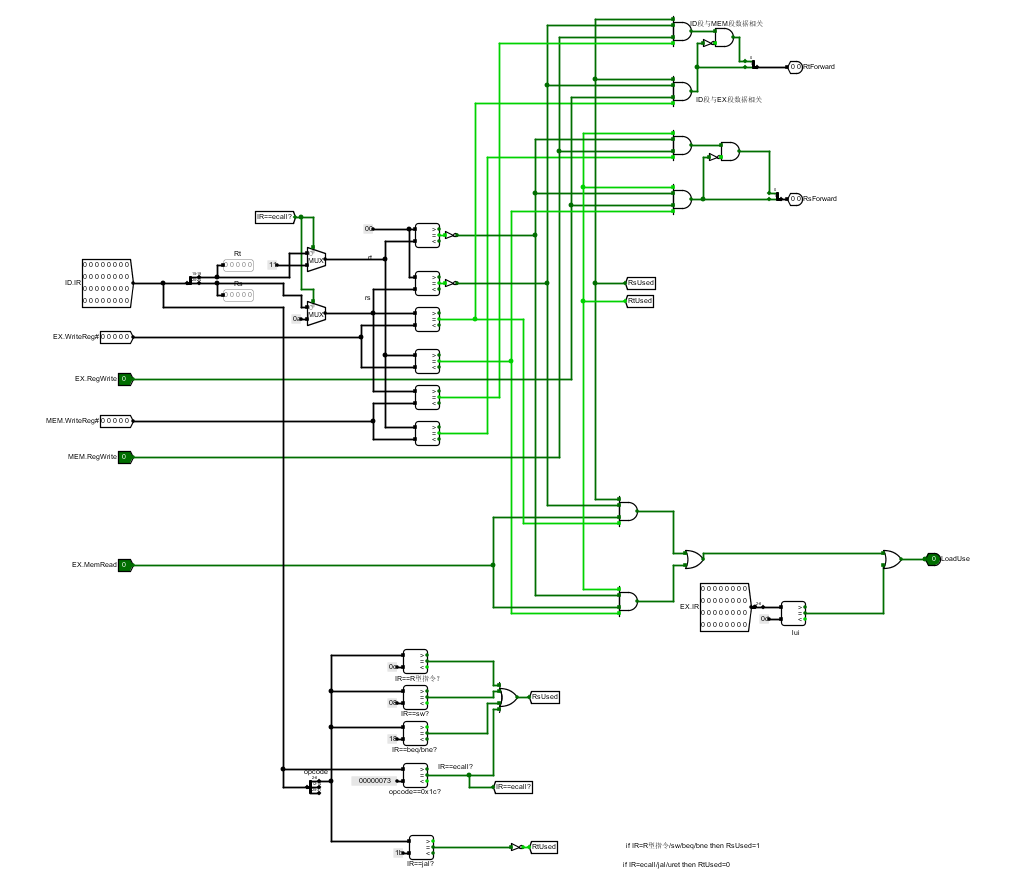


图 3.14 重定向中的多路选择

举例Rsforward依据的检测逻辑如下

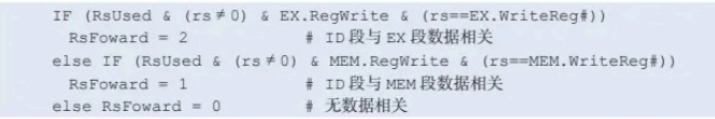


图 3.15 相关处理逻辑

需要注意的是，如果相邻两条指令存在数据相关，且前一条指令是访存指令时，这种数据相关不能采用重定向方式进行处理，虽然在功能上可以实现，但这样做的后果是 EX 段的关键路径延迟变成了 MEM 段访存延迟加EX段运算器运算延迟，而流水线频率取决于流水线中最慢的功能段，使流水线频率大大降低。因此在电路图里面添加了单独的检测逻辑

### 分支预测流水线实现

分支预测需要一个BTB表来存储历史跳转记录，如下图所示，我使用了一个八路cache

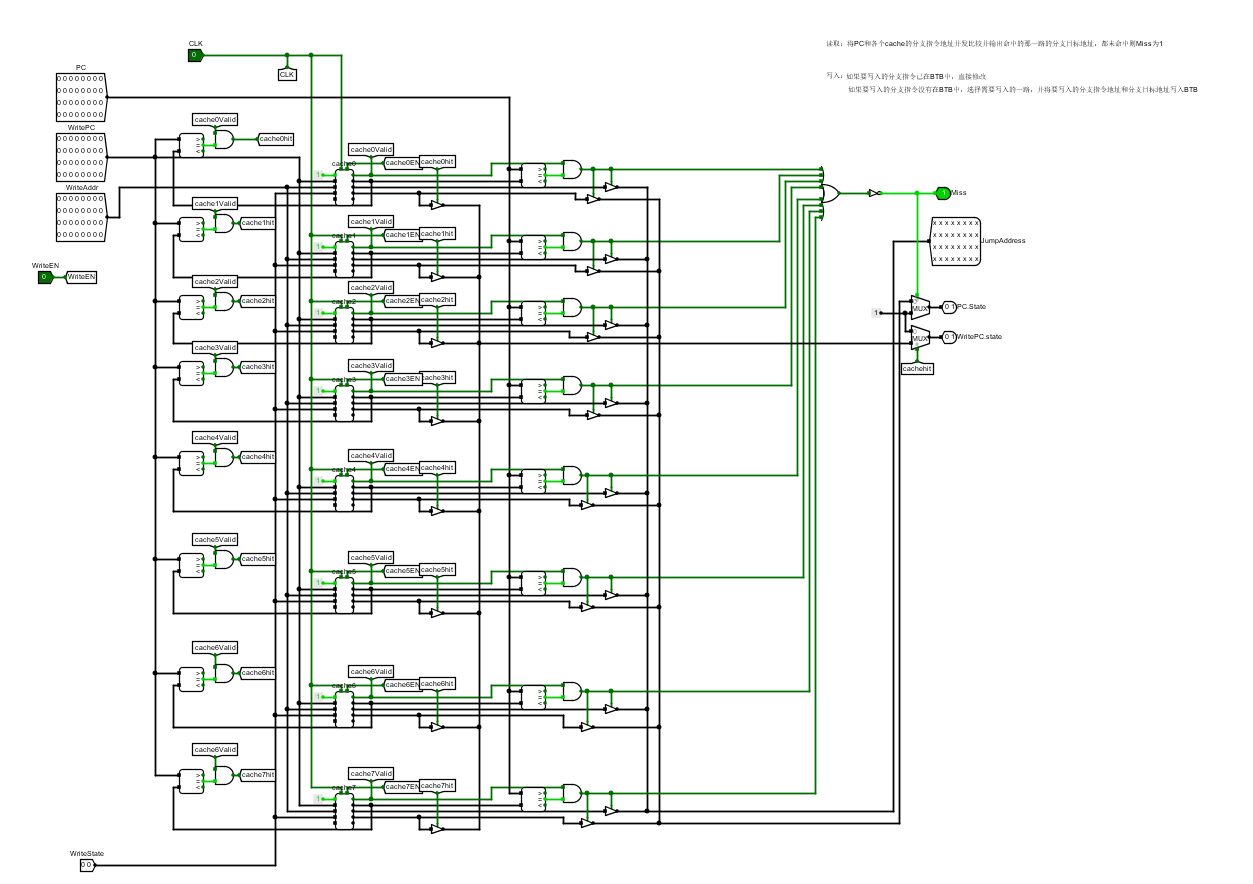


图 3.16 八路cache

有了分支预测的历史状态表，根据下图所示的分支预测状态机切换绘制电路

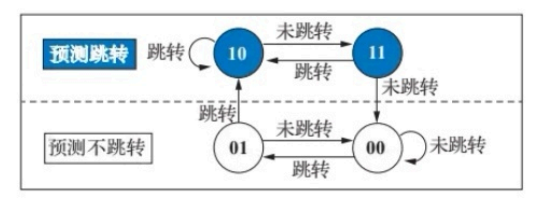


图 3.17 分支预测状态转换图

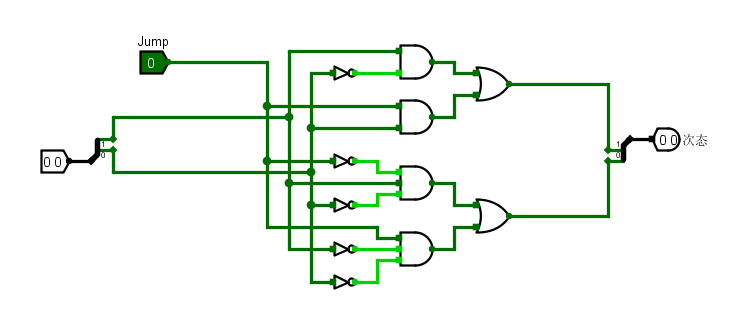


图 3.18 分支预测状态转换电路

在取指阶段放入BTB，根据不同历史记录与信号输入来决定是否需要分支跳转

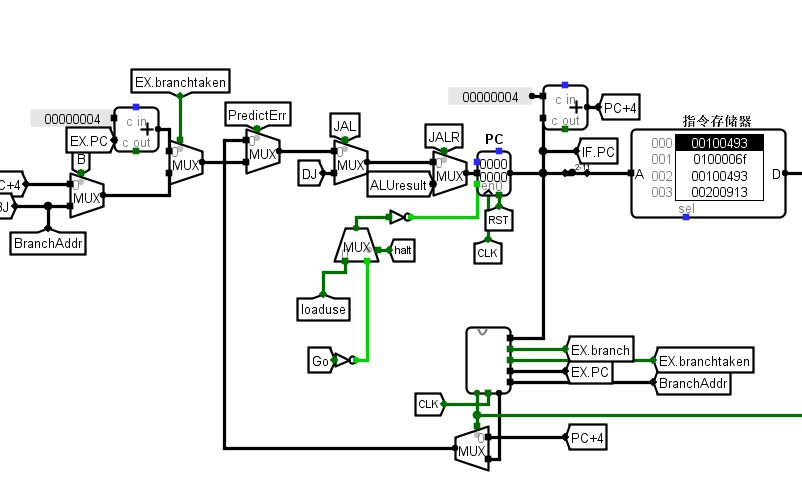


图 3.19 分支预测实现

# 实验过程与调试

## 测试用例和功能测试

所有程序的头歌测试均已通过



图4.1 测试通过截图

### 测试用例1

使用rars程序，在benchmark中插入自己CCAB支持的相关RISC-V代码，然后编译成hex文件（16进制），插入到程序内存中即可执行和测试。如下图所示

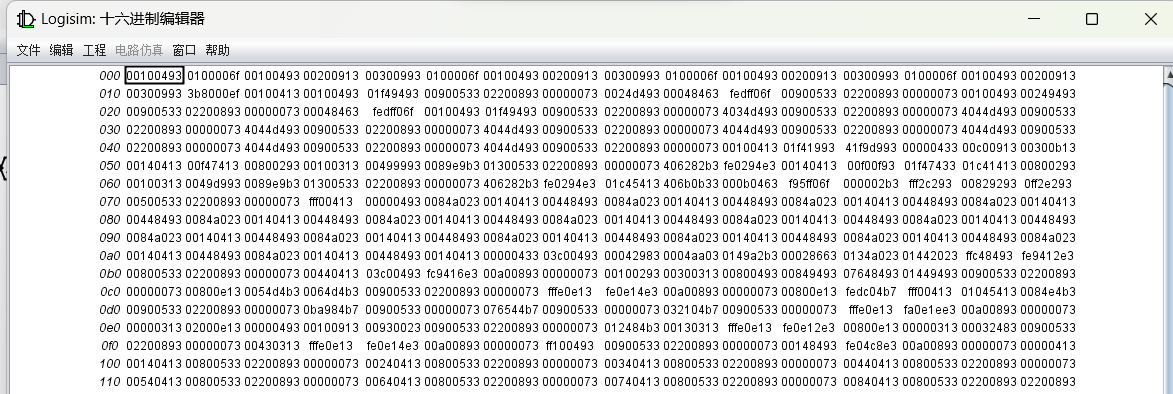


图4.2 程序文件载入内存

### 测试用例2

使用rars程序，在中断测试程序中插入自己CCAB支持的相关RISC-V代码，然后编译成hex文件（16进制），插入到程序内存中即可执行和测试。

## 性能分析

因为流水线中气泡流水线和重定向流水线都加载的是同一个文件，因此一这两个实验作为对比。两者对于分支指令的处理是相同的，均是清空已经执行的未经跳转的错误指令的流水寄存器，而在对于有数据冲突的情况处理方式大有不同。首先气泡流水线对于数据冲突的处理是遇到了数据冲突会插入气泡，从而使寄存器陷入等待时间这样，会浪费流水线的很多性能。因此气泡流水线的时间周期会长很多。

对于重定向和分支预测的性能差异，采用重定向机制后，指令流水线中的数据相关基本不需要插入气泡就可解决，只有少数Load-Use 相关还需要插入一个气泡，流水线性能得到极大的提升。此时流水线中的控制冲突对流水线性能影响最大，基于加快经常性事件的原理，应优先考虑减少分支指令引起的分支延迟损失。为减少分支延迟损失，应尽可能提前执行分支指令，例如将分支指令放在 ID 段完成，在分支预测中，提前跳转大大降低了总周期数。

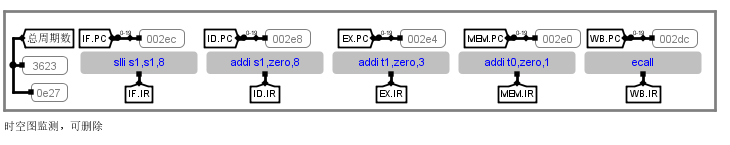


图4.3 气泡流水线总周期数

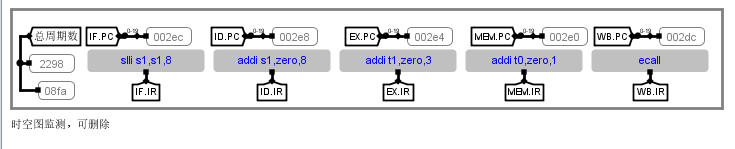


图4.4 重定向流水线总周期数

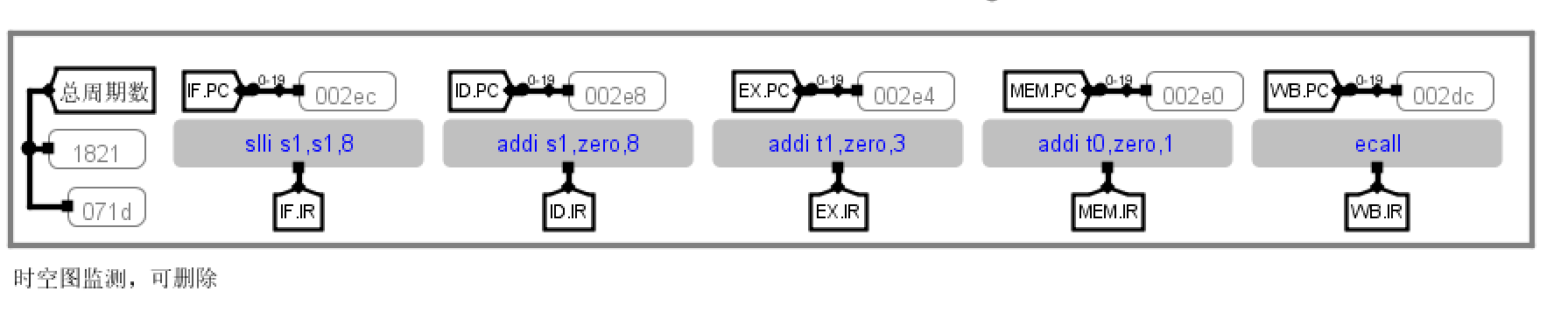


图4.5 分支预测流水线总周期数

## 主要故障与调试

### 无法正常分支

气泡流水线：无法正常分支

**故障现象：**气泡流水线中无法正常分支。

**原因分析：**原来的流水寄存器采用的是异步清零信号，会导致ID/EX流水寄存器中beq 指令所需数据和控制信号被立即清除掉，程序无法实现正常分支

**解决方案：**改用时钟敏感型的同步清零信号。

### 机器halt问题

流水线CPU：无法正确停机。

**故障现象：**CPU在halt时按下go，会导致此时位于PC寄存器的指令消失，使得按下go时候无法继续执行接下来的指令

**原因分析：**stall指令与halt指令应该有优先级的区别，如果处于halt状态按下go应该优先响应halt

**解决方案：**更改PC的使能结构，使其优先响应halt

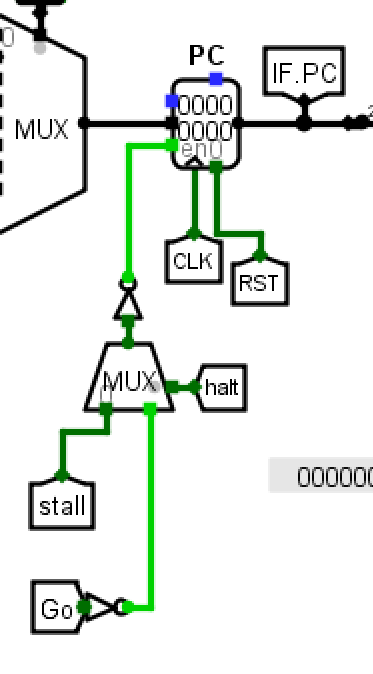
****

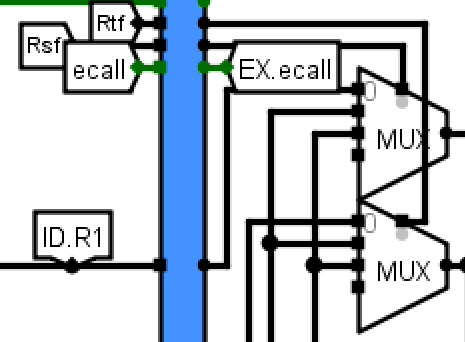
图4.6 使能更改

### 重定向Rs，Rt搞混

重定向流水线：数据选择问题。

**故障现象：**ALU计算结果出错。

**原因分析：**ALU有两个计算数据，通过重定向数据检测来判断需要使用哪个，如下图所示，一开始的时候Rt和Rs弄反了，导致ALU输入数据出错。



**解决方案：**反转Rt和Rs。

### Verilog延时

Verilog实现单周期CPU：延时问题。

**故障现象：**Verilog无法正确执行指令。

**原因分析：**最开始的代码里硬布线控制器采用了太多了寄存器而不是线网形，时序逻辑对于硬布线控制器来说效率过低，在中断时产生错误。

**解决方案：**修改地址硬布线控制器**，**利用assign语句和线网形来提高效率。

## 实验进度

表 4.1 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，并设计出单周期CPU各部件的数据通路，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。。 |
| 第三天 | 完成了单周期cpu控制器的基本调试，并且提交了单周期cpu通过测试。完成了CCAB指令的相关内容 |
| 第四天 | 设计了理想流水线，完成简单测试 |
| 第五天 | 设计了气泡流水线，通过头歌实验 |
| 第六天 | 设计了重定向流水线，通过头歌测试 |
| 第七天 | 完成了气泡流水线和重定向的CCAB指令支持 |
| 第八天 | 完成了CPU单级中断，通过测试 |
| 第九天 | 完成了CPU多级中断，通过测试 |
| 第十天 | 学习分支预测相关知识，了解清楚原理 |
| 第十一天 | 完成了分支预测流水线，通过了测试 |
| 第十二天 | 继续使用Verilog进行实现单周期CPU的工作，完成了所有部件的编写、控制器的编写，以及所有部件以及控制器的仿真测试，正在进行数据通路的拼接。 |
| 第十三天 | 使用Verilog完成单周期CPU数据通路的连接。相关功能测试通过 |
| 第十四天 | 设计Verilog中断相关部件，完成代码编写 |
| 第十五天 | 重写单周期CPU数据通路的连接，使其支持中断功能。相关功能测试通过 |

# 团队任务部分

## 总体设计

在完成基础个人实验任务的基础上，基于Logisim上的单级中断CPU和FPGA开发板上的单级中断CPU来开发一个走迷宫小游戏作为演示程序。该迷宫采用32\*32规模的LED点阵元件作为游戏的显示屏幕，游戏目的为操纵可移动的小光点从右上角走到左下角终点处。CPU运行后首先显示初始化界面，然后载入地图，在Logisim上采用鼠标点击键盘逻辑处，通过输入键盘WASD来操纵小光点的移动；在FPGA开发板上采用开发板自带的五个按钮，通过中断来实现光点的移动，采用VGA显示器接口，在外界的屏幕上显示32\*32规格的地图和移动的过程。

## 开发平台

利用Visual Studio Code来完成汇编程序的编写，利用RARS来进行hex文件的转换，利用Logisim平台来完成电路绘制。Verilog代码使用Vivado 2023进行开发，开发板型号为Artix-7 xc7a100tcsg324-1

## 软件设计

软件方面主要是迷宫游戏的汇编程序的编写然后将其利用RARS软件转换为Hex文件输入CPU。对于FPGA和Logisim都可以采用相同的软件程序。程序的逻辑如下：首先初始化的时候设定某些固定的点阵亮起作为初始化界面，然后进入空操作的循环等待用户进行关卡选择，载入相应难度的迷宫地图，然后同样进入空指令nop的循环，依靠上下左右移动四个中断程序来实现光点位置的移动，在每个移动程序的末尾都有一个当前位置是否为终点的判断，保证一旦抵达终点就能跳转到游戏通关部分程序段，修改点阵信息显示通关字样。光点右移部分中断代码如下

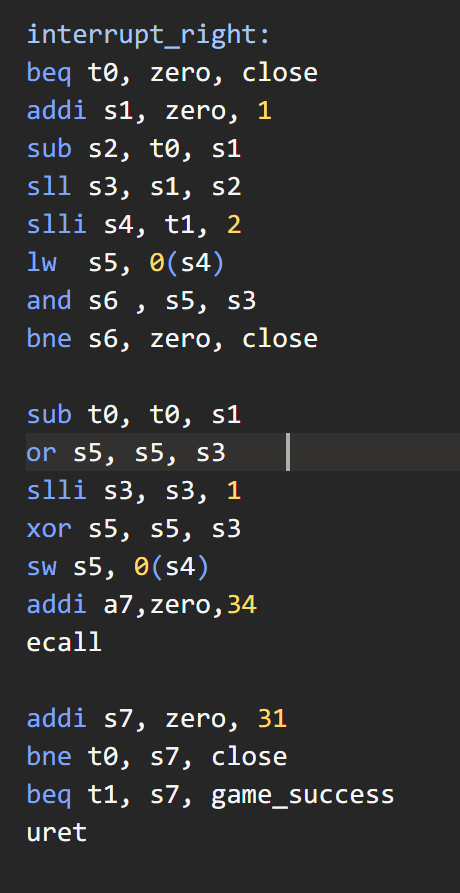


图 6.1 中断程序光点右移代码

## Logisim硬件设计

硬件方面主要依靠单人任务中实现的单级中断CPU作为游戏的主控CPU。在该CPU上增添对游戏的相关硬件支持。如支持XOR，LUI等CCAB指令来符合程序的需求，添加额外的中断电路。

然后是关卡选择和上下左右移动六个中断逻辑的实现，需要更改原本设定好的三个中断逻辑为程序经过rars编译后的六个中断服务程序的入口作为断点。

还有显示逻辑的硬件支持，需要设计32个32位的寄存器来控制对应的LED点阵的亮灭，实现的时候要设计两个32位寄存器电路一个用于备份，保证将地图和操作光点数据写入内存的同时也要备份一份以便于能在同一时钟周期内全部读取，利用Ecall指令的调用来更新地图信息，将备份寄存器的数据更新到点阵链接的寄存器组即可。

## Verilog硬件设计

通过之前实现的单级中断CPU的Verilog版本来作为游戏的主控，也是同样先添加游戏的相关指令支持，添加程序中包含的XOR，LUI等CCAB指令。

然后由于开发板没有显示模块，需要自行完成相关的VGA显示模块来外接显示器显示地图，上网查阅相关资料后，根据开发板和VGA接口的特性，编写VGA显示模块，点亮屏幕。

然后将存放在内存里的地图根据相关处理逻辑实时投射到显示器上，实现相关游戏的可视化。显示效果和部分代码如下所示。



图 6.2 迷宫显示效果图 图6.3 VGA接口部分代码

## 团队分工

主要工作分配如下：

刘兴元主要负责软件部分汇编代码的编写。

姚晨炫主要负责硬件电路Verilog中断逻辑和VGA显示接口的实现。

蔡宇杰负责Logisim上的软件装载和硬件协调。

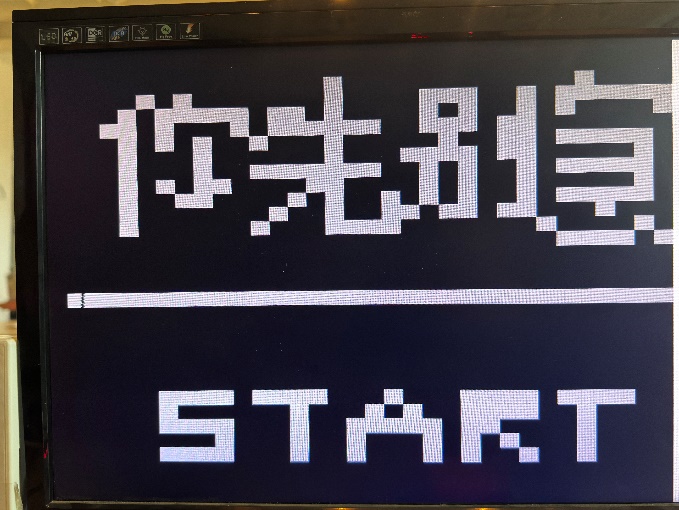
刘文博负责协调软硬件的协同问题，根据实际情况调整软件或者硬件方面的设计，进行相关测试进行debug。

## 开发中遇到的问题

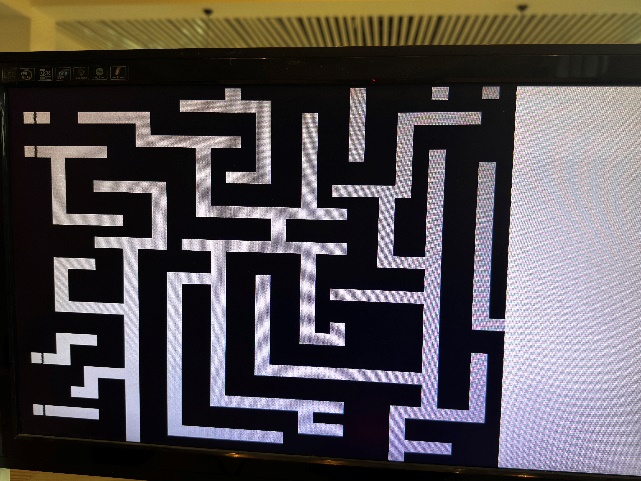
编写走迷宫的汇编程序时，其中上下左右键移动的程序逻辑比较复杂，要判断是否抵达终点，是否撞墙超出边界等。在程序的实践中发现当向左碰到墙壁后无法再进行移动，后续通过程序中逻辑的检查发现了向左撞墙程序段相比其他方向撞墙程序少了处理程序。在进行设计的时候发现还需要额外扩展指令的存在，例如xor指令不存在移动光点会产生拖影，sll指令不存在无法移动光点，还有LUI指令等，根据这些在软件中需要的指令我们又在CPU中增添了对该指令的支持。

在FPGA上实现时，首先遇到了中断时序的问题，因为FPGA上的时序要求比Logisim更高，并且没有可视化界面比较难排查，debug后发现是EPC所接的clk有误，因为不同工作模型所需要的clk时不同的（VGA为65mhz而CPU为100mhz）。在VGA开发的过程中遇到了无法实时将光点移动更新在显示器上的问题，为原有架构上的缺陷，我们重构了内存上的部分接口，开发了逻辑模块将VGA显示器直接接在内存上，直接显示位于内存中的地图和相关数据，完成了显示模块的开发。

最后实现的两种演示程序如下



6.3 游戏开始界面



6.4 迷宫界面



6.5 Logisim界面

# 设计总结与心得

## 课设总结

基于对象的存储是为了克服当前基于块的存储存在的诸多难题，在存储接口和结构层次的重要发展。可以根据应用负载选择优化的存储策略。作了如下几点工作：

1. 我们成功实现了所有任务的数据通路和控制器，并对主要组件的功能进行了完整的实现。在logisim环境中，我们成功完成了所有实验任务。。
2. 实现了FPGA上的单周期CPU设计和单级中断设计。
3. 进行了详尽的测试，同时对于个人任务的四个CCAB指令都进行了设计和测试。

## 课设心得

这次课程设计任务非常具有挑战性，是我参与过的所有课程设计中最大的一次，需要我付出无数次耐心的分析和解决问题，以及多个晚上的加班才能完成基本任务。回顾这个过程，我既感到遗憾没有完成所有实验，也感到满足因为我全力以赴并获得了一些重要的经验教训。

在第一个实验中，尽管我之前有设计MIPS CPU的经验，但合理地连接指令的数据通路仍然花费了相当多的时间。此外，每条指令的细节与MIPS指令不同，因此需要重新设计。刚开始时，由于对指令不熟悉，容易导致连线错误。只有在反复查阅RISC-V指令手册并熟悉所有指令后，我才对数据通路的实现有了充分的理解。虽然经过几次调试后通过了测试，但在后面设计流水线CPU和中断程序时，我遇到了前所未有的问题，因为没有经验，所以在调试时花费了大量时间。在设计重定向流水线时，由于分成了多个步骤，我花费了大量时间进行逐一检查和验证其正确性。在这个过程中，我深刻认识到规范化和工程化设计的重要性。没有规范的设计只会在后续调试中引发更多问题。

编写单周期CPU的Verilog代码是一项富有挑战性的任务。在这个过程中，我学到了精心规划和仔细设计的价值。从确定指令集架构到建立数据通路和控制信号，每一步都需要深思熟虑。耐心和细致是编写代码的关键，以确保正确性和性能。与团队成员的合作也加速了解决问题和提高效率。最重要的是，这个经历深化了我对计算机体系结构的理解，使我更加熟悉指令执行、数据通路和控制单元等关键概念。这个过程为我提供了宝贵的实践经验，将对我的学术和职业发展产生积极影响。总的来说，编写单周期CPU的Verilog代码是一次富有成就感的学习之旅。

然而，我对这次课程设计还有一些小小的建议。在涉及到流水线和中断设计之前，可以将实验任务适当拆分，以降低初次上手的难度。这样做可以更轻松地验证各个部分的正确性，并且可以考虑增加实验提交的时间或减少实验任务量，以提高实验成果的完善度。最后，我要感谢三位老师在本次课程设计中的耐心解答，本次课设让我受益良多。

# 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，周军龙，肖亮. 计算机组成原理实验指导与习题解析.北京：人民邮电出版社，2022.
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: 姚晨炫** |