***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： | CSIE1601 |
| 学 号： | U201616277 |
| 姓 名： | 李星儒 |
| 电 话： | 15629109360 |
| 邮 件： | [1021551729@qq.com](mailto:13456@qq.com) |

目 录

[1 课程设计概述 3](#_Toc4883616)

[1.1 课设目的 3](#_Toc4883617)

[1.2 设计任务 3](#_Toc4883618)

[1.3 设计要求 3](#_Toc4883619)

[1.4 技术指标 4](#_Toc4883620)

[2 总体方案设计 6](#_Toc4883621)

[2.1 单周期CPU设计 6](#_Toc4883622)

[2.2 中断机制设计 14](#_Toc4883623)

[2.3 流水CPU设计 14](#_Toc4883624)

[2.4 气泡式流水线设计 17](#_Toc4883625)

[2.5 重定向流水线设计 20](#_Toc4883626)

[2.6 动态分支预测机制 22](#_Toc4883627)

[3 详细设计与实现 26](#_Toc4883628)

[3.1 单周期CPU 实现 26](#_Toc4883629)

[3.2 中断机制实现 42](#_Toc4883630)

[3.3 流水CPU实现 50](#_Toc4883631)

[3.4 气泡式流水线实现 52](#_Toc4883632)

[3.5 重定向流水线实现 54](#_Toc4883633)

[3.6 动态分支预测机制实现 56](#_Toc4883634)

[4 实验过程与调试 61](#_Toc4883635)

[4.1 测试用例和功能测试 61](#_Toc4883636)

[4.2 性能分析 63](#_Toc4883637)

[4.3 主要故障与调试 64](#_Toc4883638)

[4.4 实验进度 67](#_Toc4883639)

[5 设计总结与心得 69](#_Toc4883640)

[5.1 课设总结 69](#_Toc4883641)

[5.2 课设心得 69](#_Toc4883642)

[参考文献 71](#_Toc4883643)

# 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 |
| 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 $v0==10 halt(停机指令)  else 数码管显示$a0值 |
| 24 | SYSCALL | 系统调用 |
| 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 | SRLV | 逻辑可变右移 | C2 |
| 29 | SRAV | 算术可变右移 | C3 |
| 30 | LBU | 取一个字节 | M3 |
| 31 | BGEZ | 有符号比较，大于等于0时跳转 | B4 |

# 总体方案设计

## 单周期CPU设计

本次课程设计中采用的方案是硬布线控制，在设计中要注意加上四条扩展指令。首先构造单周期CPU的大体框架，之后通过Excel表格完善相关信号，通过表达式自动生成运算控制器和控制信号生成两个模块，最后完善benchmark.asm为benchmark\_ccmb.asm，保证测试程序的正确运行。

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

PC程序计数器用来保存下一条指令的地址，应使用由时钟输入端的寄存器保存其值，保证在一个周期内PC不会发生变化。其中由于halt信号为停机信号，而go信号为继续运行信号，所以程序计数器的使能端信号为(!halt || go)

#### 指令存储器IM

指令寄存器使用的是只读寄存器ROM，通过加载镜像将指令导入到指令寄存器中之后，在运行过程中指令寄存器的值都不会发生变化。通过程序控制器

#### 运算器

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 |

表 2.2 算术逻辑运算单元功能描述

| 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 |

#### 寄存器堆RF

表 2.3 寄存器堆RF引脚与功能描述

| 引脚 | 输入/输出 | 位宽 | 功能描述 |
| --- | --- | --- | --- |
| CLK | 输入 | 1 | 时钟信号，上升沿触发 |
| WE | 输入 | 1 | 写使能信号 |
| Din | 输入 | 32 | 写入的数据 |
| RW# | 输入 | 5 | 写入的寄存器编号 |
| R1# | 输入 | 5 | 读寄存器编号 |
| R2# | 输入 | 5 | 读寄存器编号 |
| R1 | 输出 | 32 | R1#寄存器中的值 |
| R2 | 输出 | 32 | R2#寄存器中的值 |

#### 数据存储器

为了避免根据PC读取存储器和最终运算结果写入存储器的操作发生冲突，采用分离的指令存储器和数据存储器。

#### 控制器

控制器是一种OP\_CODE字段和FUNC字段与ALU\_OP和各种信号的逻辑对应关系，需要首先深入理解每个指令的执行过程，才能通过分析得到指令与ALU\_OP和各控制信号的对应关系，然后通过查找MIPS指令集得到指令与OP\_CODE和FUNC字段的对应关系，最终获得一系列逻辑表达式，可使用Logisim自动生成电路。

### 数据通路的设计

首先通过查找MIPS32指令集，首先统计指令类型分类，即R型、I型、J型，同时记录各指令的具体功能如下表2.4所示。

表 2.4 指令功能具体描述

| 指令 | op | rs | rt | rd | shamt | funct | 功能描述 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| SLL | 000000 | 000000 | rt | rd | sa | 000000 | rd<-rt<<sa |
| SRA | 000000 | 000000 | rt | rd | sa | 000011 | rd<-rt>>sa |
| SRL | 000000 | 000000 | rt | rd | sa | 000010 | rd<-rt>>sa |
| ADDU | 000000 | rs | rt | rd | 00000 | 100000 | rd<-rs+rt |
| ADD | 000000 | rs | rt | rd | 00000 | 100001 | rd<-rs+rt |
| SUB | 000000 | rs | rt | rd | 00000 | 100010 | rd<-rs-rt |
| AND | 000000 | rs | rt | rd | 00000 | 100100 | rd<-rs and rt |
| OR | 000000 | rs | rt | rd | 00000 | 100101 | rd<-rs or rt |
| NOR | 000000 | rs | rt | rd | 00000 | 100111 | rd<-rs nor rt |
| SLT | 000000 | rs | rt | rd | 00000 | 101010 | rd<-(rs<rt) |
| SLTU | 000000 | rs | rt | rd | 00000 | 101011 | rd<-(rs<rt) |
| JR | 000000 | rs | 00000 | 00000 |  | 001000 | PC<-rs |
| SYSCALL | 000000 | rs |  |  |  | 001100 |  |
| SRLV | 000000 | rs | rt | rd | 00000 | 000110 | rd<-rt >> rs |
| SRAV | 000000 | rs | rt | rd | 00000 | 000111 | rd<-rt >> rs |
| BEQ | 000100 | rs | rt |  |  |  | if rs = rt then branch |
| BNE | 000101 | rs | rt |  |  |  | if rs ≠ rt then branch |
| ADDI | 001000 | rs | rt |  |  |  | rt<-rs+immediate |
| ADDIU | 001001 | rs | rt |  |  |  | rt<-rs+immediate |
| SLTI | 001010 | rs | rt |  |  |  | rt-<(rs<immediate) |
| ANDI | 001100 | rs | rt |  |  |  | rt<-rs and immediate |
| ORI | 001101 | rs | rt |  |  |  | rt<-rs or immediate |
| LW | 100011 | rs | rt |  |  |  | rt<-memory[rs+offset] |
| SW | 101011 | rs | rt |  |  |  | memory[rs+offset]<-rt.. |
| LBU | 100100 | rs | rt |  |  |  | rt<-memory[rs+offset] |
| BGEZ | 0000001 | rs | 00001 |  |  |  | if rs ≥ 0 then branch |
| J | 000010 |  |  |  |  |  | pc<-(pc+4)[31:28]+index+00 |
| JAL | 000011 |  |  |  |  |  | pc<-(pc+4)[31:28]+index+00 |

根据指令的功能和具体实现分析其数据通路，完成下表2.5。

表 2.5指令系统数据通路框架

| 指令 | PC | RF | | | | ALU | | | | DM | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R1# | R2# | W# | Din | A | B | OP | shamt | Addr | Din |
| ADD | PC+4 | rs | rt | rd | ALU | R1 | R2 | 5 |  |  |  |
| ADDI | PC+4 | rs |  | rt | ALU | R1 | IMM16 | 5 |  |  |  |
| ADDIU | PC+4 | rs |  | rt | ALU | R1 | IMM16 | 5 |  |  |  |
| ADDU | PC+4 | rs | rt | rd | ALU | R1 | R2 | 5 |  |  |  |
| AND | PC+4 | rs | rt | rd | ALU | R1 | R2 | 7 |  |  |  |
| ANDI | PC+4 | rs |  | rt | ALU | R1 | IMM16 | 7 |  |  |  |
| SLL | PC+4 |  | rt | rd | ALU |  | R2 | 0 | shamt |  |  |
| SRA | PC+4 |  | rt | rd | ALU |  | R2 | 1 | shamt |  |  |
| SRL | PC+4 |  | rt | rd | ALU |  | R2 | 2 | shamt |  |  |
| SUb | PC+4 | rs | rt | rd | ALU | R1 | R2 | 6 |  |  |  |
| OR | PC+4 | rs | rt | rd | ALU | R1 | R2 | 8 |  |  |  |
| ORI | PC+4 | rs |  | rt | ALU | R1 | IMM16 | 8 |  |  |  |
| NOR | PC+4 | rs | rt | rd | ALU | R1 | R2 | 10 |  |  |  |
| LW | PC+4 | rs |  | rt | DM.Din | R1 | IMM16 | 5 |  | ALU |  |
| SW | PC+4 | rs | rt |  |  | R1 | IMM16 | 5 |  | ALU |  |
| BEQ | PC+4 or PC+4+(IMM16<<2) | rs | rt |  |  | R1 | R2 |  |  |  |  |
| BNE | PC+4 or PC+4+(IMM16<<2) | rs | Rs |  |  | R1 | R2 |  |  |  |  |
| SLT | PC+4 | rs | rt | rd | ALU | R1 | R2 | 11 |  |  |  |
| SLTI | PC+4 | rs |  | rt | ALU | R1 | IMM16 | 11 |  |  |  |
| SLTU | PC+4 | rs | rt | rd | ALU | R1 | R2 | 12 |  |  |  |
| J | {(PC+4)[31:28],IMM26,00} |  |  |  |  |  |  |  |  |  |  |
| JAL | {(PC+4)[31:28],IMM26,00} |  |  | 0x1f | PC+4 |  |  |  |  |  |  |
| JR | R1 | rs |  |  |  |  |  |  |  |  |  |
| SYSCALL |  | 0x2 | 0x4 |  |  | R1 | R2 | 11 |  |  |  |
| MFC0 | PC+4 | 中断相关指令，须在CP0中另作处理 | | | | | | | | | |
| MTC0 | PC+4 |
| ERET | PC+4 |
| 扩展：SRLV | PC+4 |  | rt | rd | ALU |  | R2 | 2 | R1[4:0] |  |  |
| 扩展：SRAV | PC+4 |  | rt | rd | ALU |  | R2 | 1 | R1[4:0] |  |  |
| 扩展：LBU | PC+4 | rs |  | rt | DM.Din | R1 | IMM16 | 5 |  | ALU |  |
| 扩展：BGEZ | PC+4 or PC+4+(IMM16<<2) | rs |  |  |  | R1 |  | 11 |  |  |  |

根据上表，可对指令进行整合分类，同时根据以上两个表格可准确判定控制器各控制信号的产生条件，以此完成控制器的设计。

### 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，统计得到的控制信号以及说明如表 2.6所示。

表 2.6 主控制器控制信号的作用说明

| 控制信号 | 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 | 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 |
| AluSrc | 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 |  |
| 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 |
| 扩展：SV | 可变逻辑移位译码控制信号 | SRAV、SRLV指令，选择移位量 |
| 扩展：LBU | LBU指令译码信号 | LBU指令，用于选择从数据存储器读出的数据 |
| 扩展：BGEZ | BGEZ指令译码信号 | BGEZ指令，用于有条件分支控制 |

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 2.7所示。

表 2.7 主控制器控制信号

| 指令 | MemToReg | MemWrite | ALU\_SRC | RegWrite | SYSCALL | SignedExt | RegDst | BEQ | BNE | JR | JMP | JAL | BGEZ | SV | LBU |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SLL | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SRA | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SRL | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADD | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADDU | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SUB | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AND | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| OR | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| NOR | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SLT | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SLTU | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JR | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| SYSCALL | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| J | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| JAL | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| BEQ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| BNE | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADDI | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ANDI | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADDIU | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SLTI | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ORI | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| LW | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SW | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SRLV | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| SRAV | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| LBU | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| BGEZ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

## 中断机制设计

### 总体设计

中断处理的整体思路就是使用存储器模拟栈，存放现场，跳转到中断处理程序后执行程序，在最后返回时恢复现场，最终将以上描述的相关实现电路封装成一个CP0，作为中断寄存器。

### 硬件设计

中断相关指令有ERET、MFC0、MTC0，三个指令的功能和分析在指令分析部分已完成，所以首先需要对IR进行译码，分析当前指令是否是中断指令，如果不是则直接将输入的PC输出，如果不是则要将当前输入的PC输出到作为EPC中，用来在之后存放到数据存储器中。考虑多级中断的情况还需要设计一个多级中断处理程序，实现对中断程序的切换操作。

### 软件设计

CP0寄存器的实现需要在代码中实现现场保护，因其中只是用到了t0、t1、s0、s1、a0、v0寄存器，所以在保护现场时仅保存这六个寄存器的值。最后运行完程序时从栈中读出对应的寄存器数值，以实现ret功能。

多级中断时还要考虑在高优先级中断打断低优先级中断时需要等到低优先级中断完成保存现场操作，因此需要设置一个信号IE来表示当前有中断在进行保护现场过程，不能执行其他中断，只有当完成保存现场后才能通过某种操作提醒保存当前EPC和修改当前IE。这里采用的具体方式是采用R2的值作为要写入IE信号的值，其中R2只取最低位，若存在写入信号则将R2写入IE寄存器。

## 流水CPU设计

### 总体设计

在计算机中可将重复的过程分解为若干个子过程，每个子过程由专门的功能部件来实现，多个处理过程在时间上错开，依次通过各功能段，实现子进程之间的并行，这就是流水线技术。实验中采用的是经典的5段RISC流水线，分为取指令（IF）、指令译码/读寄存器（ID）、执行/有效地址计算（EX）、存储器访问（MEM）、写回周期（WB）五个阶段，其工作示意图如下所示：

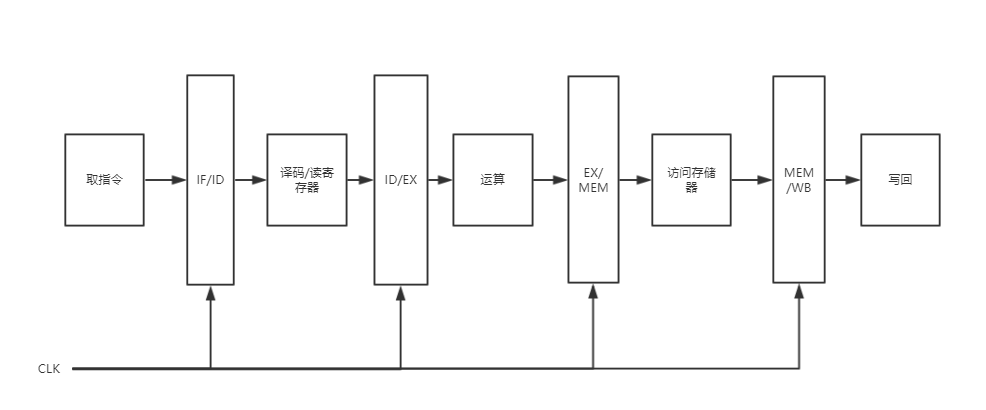


图 2.2 工作示意图

分析具体功能，对各阶段进行分析如下：

取值令（IF）阶段主要功能是根据PC的值选出指令，由PC寄存器、IM指令寄存器以及生成PC的多种选择通路组成。

译码（ID）阶段主要功能是对取出的指令IR进行译码，由控制器、寄存器堆以及使能信号控制通路组成。

执行（EX）阶段主要功能是对准备好的操作数进行运算和处理，由ALU运算器和计算新PC数值的相关通路组成，其中有效地址的相关运算使用单独的加法器进行计算。因考虑虽然在ID阶段已可以判断出无条件跳转指令，但有条件跳转指令的下一条指令结果在EX段才计算出来，若将其分开讨论即无条件跳转在ID阶段实现同时有条件跳转在EX阶段实现，则可能会造成PC选择的冲突（上一条指令为有条件跳转指令，下一条指令为无条件跳转指令）。

访存（MEM）阶段主要功能是将数据存储进存储器，主要由数据存储器和控制信号组成。

写回（WB）阶段主要功能是将结果写入统用寄存器堆，主要由控制信号和选择通路组成，该控制信号用于控制ID阶段寄存器堆的写入功能。其中由于ID段需要对通用寄存器堆进行读操作，而WB阶段需要对通用寄存器堆进行写操作，为了结果对同一统用寄存器的读写冲突，将写操作安排在时钟周期的前半拍完成，把读操作安排在时钟周期的后半拍完成，具体实现方式是将寄存器堆的触发方式设置为下降沿触发，表示如果由写控制信号，则在下降沿进行写操作；读操作没有触发条件，当存在读信号则直接读出数据，而读信号的产生和其他信号一样是在时钟上升沿产生。

### 流水接口部件设计

在设计接口时为了保证方便仿真调试，首先实现每个阶段都存在PC、IR信号。其次每个接口部件需保存前一阶段运算或处理后的结果和之后阶段所需要的相关信号。最后每个流水线寄存器应存在CLK时钟端、RST清零端、EN使能端、IN传输状态。同时为了便于理解将各种信号命名为段名\_信号名字，更详细的指令信号，例如IF\_PC表示IF段的PC信号。

IF/ID寄存器中，需要保存由取值阶段的PC以及根据PC取出的IR。

ID/EX寄存器中，首先将IR译码，分为OP\_CODE、FUNC、rs、rt、rd、shamt、imm16、imm26字段。OP\_CODE和FUNC字段通过控制器译码为一组控制信号，通过选择得到的ID.R1#和ID.R2#通过寄存器堆得到寄存器的值ID\_R1和ID\_R2。ID段直接使用的信号只有RegDst信号和SV信号。其中RegDst信号用来选择写入的寄存器是rd或rt或0x1f寄存器，SV寄存器用来选择shamt的值是IR本身的shamt还是ID\_R1的低五位。所以需要传递的信号包括PC和IR、除RegDst和SV以外的所有信号、从寄存器堆读出的R1和R2、译码的到的字段imm16和imm26、选择获得的shamt和寄存器编号。同时为了方便后期实现重定向等操作，可同时将ID.R1#和ID.R2#两个寄存器编号也同时传递到EX段。

EX/MEM寄存器中，由于在EX阶段控制运算的相关信号已进行计算，只需要向后传递MEM和WB所需要的信号。所以需要传递的信号包括PC和IR、LBU信号、MemToReg、MemWrite、RegWrite、JAL、SYSCALL、R1、R2、WriteReg#以及ALU得到的结果result1和result2。

MEM/WB寄存器中，需要将WB相关信号保存下来，同时需要保存SYSCALL信号、R1和R2的值用来判断是停止还是显示数据。其中包括PC和IR、LBU、MemToReg、RegWrite、JAL、SYSCALL、ALU的结果result1、数据存储器读出的Dout、需要写入的寄存器编号、R1和R2。

### 理想流水线设计

理想流水线不考虑冲突问题，在程序中多条指令间可能存在相关和依赖，但理想流水线仅需运行无数据相关、无分支相关的程序。其设计图如下所示：

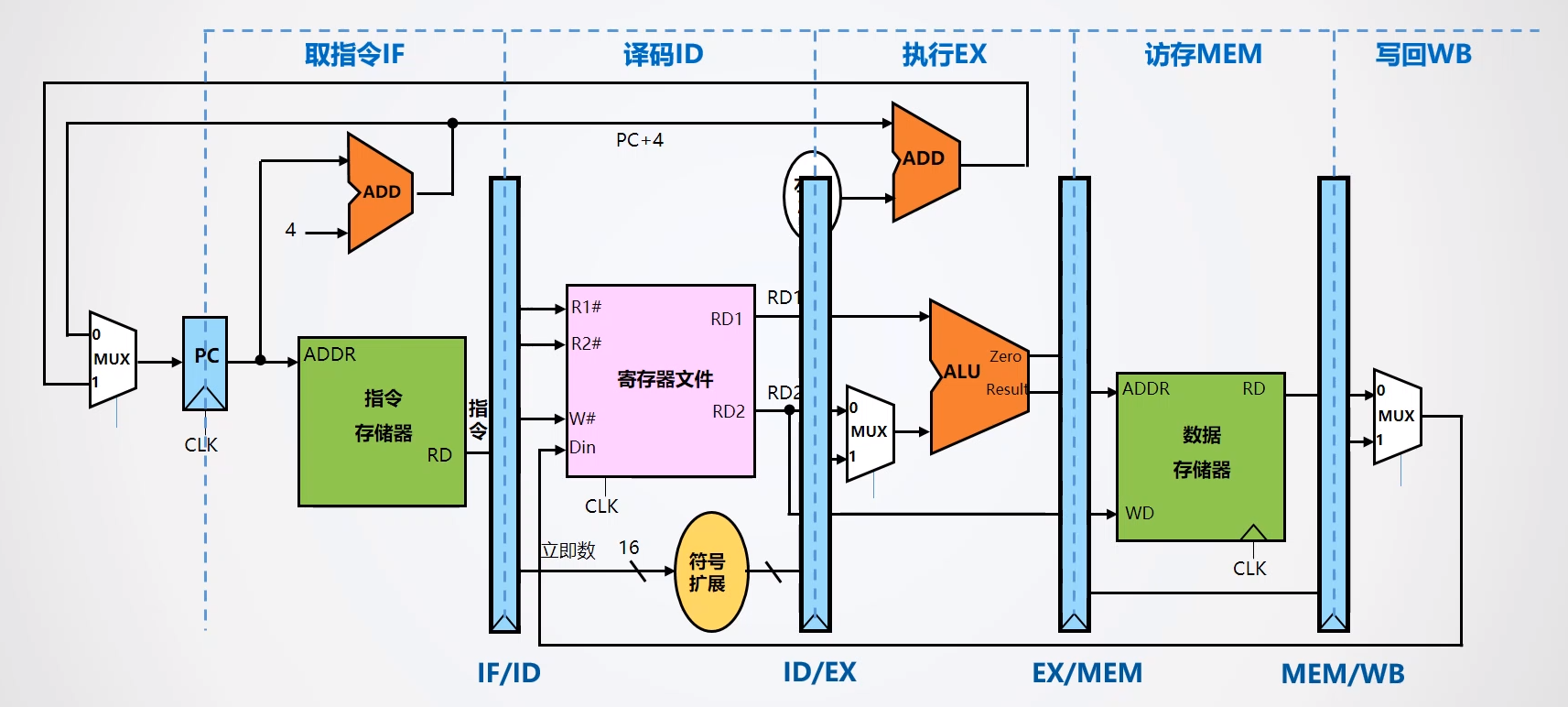


图 2.3 理想流水线设计图

## 气泡式流水线设计

指令流水线中存在相关和冲突现象，相关是指两条指令之间存在某种依赖关系，冲突是指对于具体的流水线来说，由于相关的存在，使得指令流中的下一条指令不能在指定时钟周期开始执行。具体描述入下表2.8所示。

表 2.8 流水线的相关与冲突

| 类型 | 描述 | 解决方法 |
| --- | --- | --- |
| 资源相关 | 争用主存：IF取指令、MEM读写操作数 | 采用哈佛结构使指令存储器和数据存储器分离 |
| 资源相关 | 争用ALU：运算指令、有效地址计算 | 运算指令使用ALU继续进行计算，其他直接使用加法器 |
| 分支相关 | EX段实现跳转，包括有条件跳转和无条件跳转 | 流水线中提前取出的指令全部作废，将相关流水线寄存器清空 |
| 数据相关 | 指令操作数依赖于前一条指令的运算结果 | 流水线停顿，直到数据写回 |
| 数据相关 | ID段读取寄存器、WB段写入寄存器 | 设置寄存器堆先写后读，寄存器堆下降沿写入，流水接口上升沿有效 |

### 分支相关

对于有条件跳转指令和无条件跳转指令，在EX段首先对ALU的运算结果与BNE、BEQ、BGEZ三个信号进行逻辑运算得到有条件跳转的信号EX\_B\_EN，将该信号与EX\_JR、EX\_JMP逻辑或得到EX段的跳转信号EX\_Branch。同时在EX段中需要计算出下一条有效地址，其中各指令结果如下表2.9所示，其中BranchAddr||00意为BranchAddr后加两个0。

表 2.9 有效地址计算

| 指令 | 有效地址 |
| --- | --- |
| BEQ/BNE/BGEZ | PC + 4 + sign\_extend(BranchAddr||00) |
| JR | GPR[rs]=EX\_R1 |
| J | (pc+4)[31:28]||BranchAddr ||00 |
| JAL | PC+4，将PC+4存入寄存器，运行结束后通过寄存器的值返回，在WB阶段实现 |

当检测到EX\_Branch信号即分支成功信号时将IF/ID和ID/EX两个流水线寄存器清零，同时以EX\_Branch为控制信号选择下一条PC为EX段计算出的有效地址或是IF.PC+4。

### 数据相关

讨论指令间的数据相关首先要分析各指令对寄存器的使用情况，具体情况如下表2.10所示：

表 2.10 源寄存器使用情况

| 指令 | R1\_Used | R2\_Used | 指令 | R1\_Used | R2\_Used |
| --- | --- | --- | --- | --- | --- |
| SLL | 0 | 1 | JAL | 0 | 0 |
| SRA | 0 | 1 | BEQ | 1 | 1 |
| SRL | 0 | 1 | BNE | 1 | 1 |
| ADD | 1 | 1 | ADDI | 1 | 0 |
| ADDU | 1 | 1 | ANDI | 1 | 0 |
| SUB | 1 | 1 | ADDIU | 1 | 0 |
| AND | 1 | 1 | SLTI | 1 | 0 |
| OR | 1 | 1 | ORI | 1 | 0 |
| NOR | 1 | 1 | LW | 1 | 0 |
| SLT | 1 | 1 | SW | 1 | 1 |
| SLTU | 1 | 1 | SRLV | 1 | 1 |
| JR | 1 | 0 | SRAV | 1 | 1 |
| SYSCALL | 1 | 1 | LBU | 1 | 0 |
| J | 0 | 0 | BGEZ | 1 | 0 |

依照此表格可建立OP\_CODE、FUN与R1\_Used、R2\_Uesd的关系。在IF段使用该数据相关检测得到源寄存器使用情况，之后将ID段的寄存器使用情况与EX段和MEM段进行比对，其中0号寄存器不考虑相关性。具体逻辑如下所示：

数据相关信号 =

( R1\_Used && (ID.R1#==EX.WriteReg#) && (ID.R1#!=0) && EX.RegWrite) ||

( R2\_Used && (ID.R2#==EX.WriteReg#) && (ID.R2#!=0) && EX.RegWrite) ||

( R1\_Used && (ID.R1#==MEM.WriteReg#) && (ID.R1#!=0) && MEM.RegWrite) ||

( R2\_Used && (ID.R2#==MEM.WriteReg#) && (ID.R2#!=0) && MEM.RegWrite) ；

表 2.11 数据相关子电路引脚与功能描述

| 引脚 | 输入/输出 | 位宽 | 功能描述 |
| --- | --- | --- | --- |
| OP\_CODE | 输入 | 6 | ID段译码得到的OP字段 |
| FUNC | 输入 | 6 | ID段译码得到的FUNC字段 |
| ID.R1# | 输入 | 5 | ID段使用的寄存器编号1 |
| ID.R2# | 输入 | 5 | ID段使用的寄存器编号2 |
| EX.WriteReg | 输入 | 5 | EX段写入的寄存器编号 |
| EX.RegWrite | 输入 | 1 | EX段写入的寄存器的控制信号 |
| MEM.WriteReg | 输入 | 5 | MEM段写入的寄存器编号 |
| MEM.RegWrite | 输入 | 1 | MEM段写入的寄存器的控制信号 |
| 数据相关 | 输出 | 1 | 数据相关信号 |

如果存在数据相关现象，即数据相关信号为1，则ID段需要读取的寄存器与EX段或MEM段要写入新数据的寄存器相同，所以需要一直等到数据写入寄存器后再进行ID段的读取操作，最终指令在WB段的下降沿写入数据，并在下一个上升沿开始继续运行ID段指令。因此首先需要控制PC寄存器的使能端和流水线寄存器的使能端，使得IF、ID、IF/ID的流水线停止运行，同时需要对ID/EX段的流水线寄存器清零以实现插入气泡的功能，避免重复执行当前EX段指令。

## 重定向流水线设计

### 一般数据相关

在气泡流水线中通过插入气泡的方式来解决数据冲突问题，但因气泡和流水线暂停的现象导致流水线性能受损，所以在重定向流水线中通过添加数据旁路来避免流水线的暂停情况。主要思想就是在发生写后读的情况下，如果在计算结果尚未写入寄存器前已需要进行读操作，则可以通过旁路直接将新的数据传送到下一条指令所在的运行阶段中，这个过程就是重定向过程。实验中采用在EX段进行重定向操作。

分析重定向旁路所需要的使用到的引脚得到以下表格。

表 2.12 重定向数据相关自电路引脚与功能描述

| 引脚 | 输入/输出 | 位宽 | 功能描述 |
| --- | --- | --- | --- |
| Ori\_EX\_R1 | 输入 | 32 | ID/EX寄存器中传递过来的R1 |
| Ori\_EX\_R2 | 输入 | 32 | ID/EX寄存器中传递过来的R2 |
| EX.IR | 输入 | 32 | EX段的指令，用来分离字段OP和FUNC |
| EX.R1# | 输入 | 5 | EX段使用的寄存器编号1 |
| EX.R2# | 输入 | 5 | EX段使用的寄存器编号2 |
| WB.WriteReg | 输入 | 5 | WB段写入的寄存器编号 |
| WB.RegWrite | 输入 | 1 | WB段写入的寄存器的控制信号 |
| MEM.WriteReg | 输入 | 5 | MEM段写入的寄存器编号 |
| MEM.RegWrite | 输入 | 1 | MEM段写入的寄存器的控制信号 |
| Din | 输入 | 32 | WB段最终要写入寄存器的数据 |
| MEM\_Addr | 输入 | 32 | EX/MEM寄存器中传递过来的ALU运算结果 |
| EX\_R1 | 输出 | 32 | 选择后的EX段ALU操作数R1 |
| EX\_R2 | 输出 | 32 | 选择后的EX段ALU操作数R2 |

首先在旁路中对EX.IR进行译码，得到字段OP和FUNC，使用气泡流水线中相同的源寄存器使用情况子电路进行判断得到R1\_Used和R2\_Used。之后在旁路中设置四个信号，R1\_WB：EX段的R1与WB阶段写入的寄存器冲突；R1\_MEM：EX段的R1与MEM阶段写入的寄存器冲突；R2\_WB：EX段的R2与WB阶段写入的寄存器冲突；R2\_MEM：EX段的R2与MEM阶段写入的寄存器冲突；

R1\_WB=( R1\_Used && (EX.R1#==WB.WriteReg#) && (EX.R1#!=0) && WB.RegWrite);

R2\_WB=( R2\_Used && (EX.R2#==WB.WriteReg#) && (EX.R2#!=0) && WB.RegWrite);

R1\_MEM=( R1\_Used && (EX.R1#==MEM.WriteReg#) && (EX.R1#!=0) && MEM.RegWrite) ;

R2\_MEM=( R2\_Used && (EX.R2#==MEM.WriteReg#) && (EX.R2#!=0) && MEM.RegWrite) ;

最后通过四个相关信号，选择得到新的EX\_R1，需要注意MEM段的MEM\_Addr的优先级应比WB段的Din高，因为在WB段和MEM段同时与EX段出现数据相关时，则意味着在上一个周期时，MEM段（现于WB段）和EX段（现于MEM段）的指令在之前也出现锅数据相关，此时已通过重定向旁路使得EX段（现于MEM段）使用正确的数据，所以在当前时钟周期若WB段和MEM段同时出现数据冲突，则已不需要考虑WB段对EX段的影响，仅需考虑MEM段对EX段的影响，整合到电路中则优先选择MEM段的数据相关选择结果即可。

### Load-Use相关

并不是所有的数据冲突都可以使用重定向解决，比如在运行到LW指令（现于MEM段）时，之后的下一条指令需要使用到LW指令的寄存器（现于EX段），理论上可以考虑将MEM访存的结果重定向到EX段，但由于访存的时延时最长的，并不能保证重定向到EX的数据是否是已经更新过的数据，若要保证该数据正确则需要增加EX段的时延，即待读出数据后再将数据送入ALU，此时EX段时延=MEM段时延+ALU时延，这会成为整个流水线的关键路径，导致流水线性能下降。因此在MIPS流水线中要单独处理Load相关指令的操作。

处理Load-Use相关的方式与气泡流水线处理数据相关的方式相同，即使流水线停顿并插入气泡，直至冲突消失。因此处理Load-Use相关时可直接使用数据相关检测子电路，控制输入的MEM.WriteReg#和MEM.RegWrite均为0，即只需要判断ID段和EX段的冲突关系，最终输出一个数据相关信号，由于MemToReg信号仅在Load相关指令时为1，所以最终的逻辑表达式为LoadUse相关=数据相关 && EX.MemToReg，意为ID段和EX段出现数据相关且EX段为load指令时产生LoadUes相关信号，该信号用来控制PC的使能端，IF/ID的使能端和ID/EX的清零端，以实现IF、ID段的停顿和EX段的气泡。

## 动态分支预测机制

分支指令在分支成功即需要跳转时，采用的方式时清空前面作废的指令，再继续执行后续指令，这种方式使得流水线会运行多个空指令，产生分支延迟，当分支指令较多时产生的分支延迟无法忽略，会给流水线带来相当大的性能损失，因此为了更充分发挥流水线的性能，使用动态分支预测机制。

动态分支预测机制的主要思想就是，提前预测流水线的分支是否会成功，同时要计算出分支目标地址。具体设计的方法是将分支指令及其分支目标地址存放在Cache中，在ID段对指令继续判断，若在Cache中命中则直接将IF的PC输入修改为从Cache中读出的目标地址，依此避免了流水线的暂停操作，一定程度上增大了流水线的效率。分析分支相关信号得到如下表所示三个信号的产生条件

表 2.13主控制器控制信号的作用说明

| 产生信号 | 信号说明 | 产生条件 |
| --- | --- | --- |
| predict\_suc | 分支预测成功 | 对分支指令进行预测且分支成功 |
| predict\_fail | 分支预测失败 | 对分支指令进行预测但分支失败 |
| predict\_clk | 因分支预测产生的清零信号，用来插入气泡 | 未对分支指令进行预测且分支成功或对分支指令进行预测但分支失败 |

用EX.NewPC来表示EX段计算处的跳转结果，若非跳转指令则EX.NewPC=32‘b0；用EX跳转指令来表示EX段指令是一个跳转指令；用ID.PC表示ID段当前的指令地址。由此可将三个控制信号表示为如下逻辑表达式：用EX.Branch表示EX段当前分支成功。

predict\_suc = (ID.PC==EX.NewPC) && EX.跳转指令 && EX.Branch；

predict\_fail = (ID.PC==EX.NewPC) && EX.跳转指令 && !EX.Branch；

predict\_clk = predict\_fail || ((ID.PC!=EX.NewPC) && EX.跳转指令 && EX.Branch)；

### BHT

BHT全名branch prediction table，意为分支预测表，本次实验中使用的是全相连Cache，即主存中的一个地址可以被映射到cache的任意一行。在具体设计时当cache没有满时选择从0-7八个cache行，每次都选择标号最小的且有效位为0的cache行来存放当前数据，当cache满时则通过LRU淘汰算法选出淘汰标志位数值最大的cache行，使新的数据覆盖原数据。流程图如下所示：

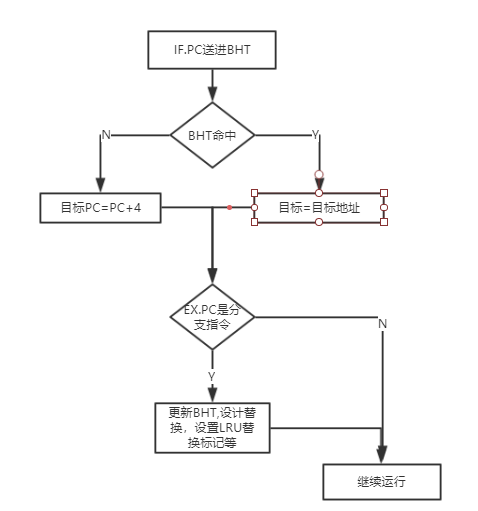


图 2.4 分支预测流程图

### 完善

初期分支预测采用一位valid有效位来判断，但在实际程序中并不一定都是循环操作，所以分支预测失败的次数过多，对流水线的效率没有较大的提高，所以采用双位检测，其状态图如下所示，当状态为00、01时预测不跳转，为10、11时预测跳转，当前指令分支跳转成功与否将会决定状态的变迁。

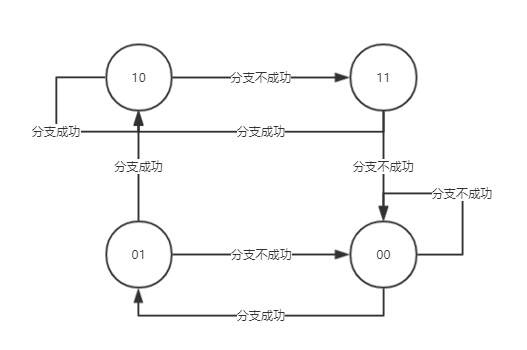


图 2.5 预测状态转移图

修改后的流程图如下所示：

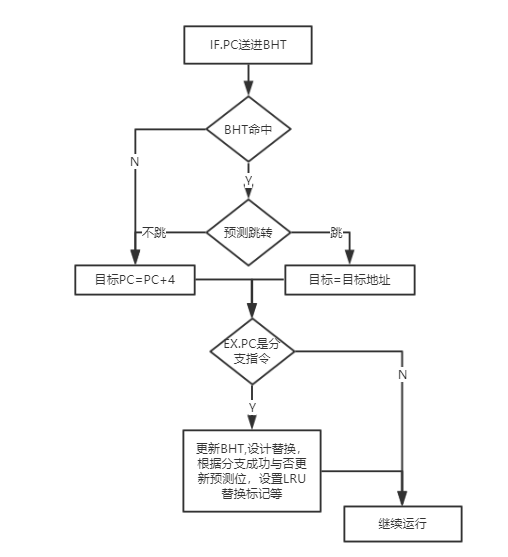


图 2.6 分支预测流程图

所以在cache行的设计中，首先用set0-set7表示替换时的行选择信号，当行选择信号为1时，需要控制有效位、标记位、数据寄存器的使能端以及预测位的清零端；使用L0-L7表示行命中信号，即IF.PC与该行的标记位相同，需要清空淘汰标记为位并将数据传送出去；使用S0-S7表示行命中信号，但该命中信号是指EX.PC与该行的标记位相同，需要控制预测位的使能端，同时根据分支成功与否更新预测位。

# 详细设计与实现

## 单周期CPU 实现

### 主要功能部件实现

1. 程序计数器（PC）
2. 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。PC的输入有四种可能，采用一个数据选择器对其进行选择，作为寄存器期间的输入端，如图 3.1所示。

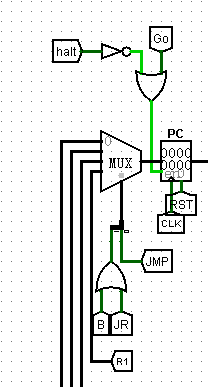


图 3.1程序计数器（PC）

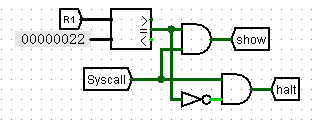


图3.2 show和halt信号产生

1. FPGA实现：

程序计数器PC的Verilog代码如下：

always@(posedge clk\_dvid) begin

/\* ----- PC ----- \*/

if (JMP) begin

if (JR) begin /\* PC = $RS \*/

PC = R1;

end

else begin

PC=PC+4;

PC = {PC[31:28],RAM\_data[25:0],2'b00};

end

end

else if (BJMP) begin /\* beq or bne \*/

PC = PC + 4 + ( $signed(IMM) << 2);

end

else begin

if (!HALT | go) begin

PC = PC + 4;

end

end

if (reset) begin

PC = 0;

end

end

show和halt信号的Verilog代码如下：

assign show=((data==32'h00000022)&syscall)?1:0;

assign halt=((data!=32'h00000022)&syscall)?1:0;

1. 指令存储器（IM）
2. Logism实现：

使用Logisim自带的只读存储器（ROM），并设置其地址位宽为10，数据位宽为32。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。

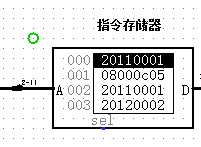


图 3.3指令存储器（IM）

1. FPGA实现：

设置二位数组存放指令，每行有DATA\_WIDTH个数字表示位宽为DATA\_WIDTH，一共有2^ ADDR\_WIDTH行，表示取值范围皆可取到指令，以此作为指令存储器，其中A为输入指令存储器的PC地址，D为从指令存储器读出的一个指令IR。

指令存储器IM的Verilog代码如下：

module ins\_mem

#(parameter DATA\_WIDTH=32,parameter ADDR\_WIDTH=10)

(

input [(ADDR\_WIDTH-1):0] A,

output [(DATA\_WIDTH-1):0] D

);

reg[(DATA\_WIDTH-1):0] mem[2\*\*ADDR\_WIDTH-1:0];

initial begin

mem[0] = 0;

$readmemh(路径,mem);

end

assign D = mem[A];

endmodule

1. 运算器（ALU）
2. Logism实现：

使用一个封装的ALU，其内部使用纯逻辑电路实现运算器所需要的功能。

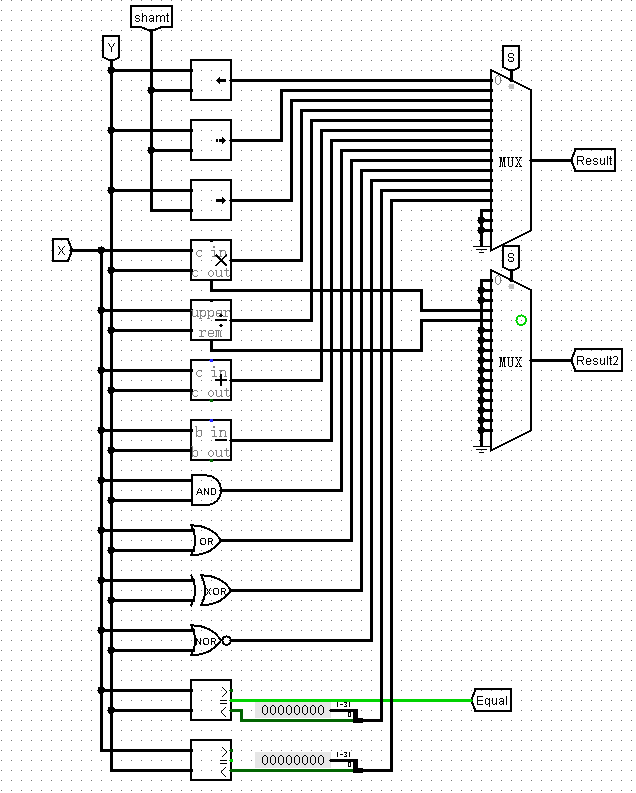


图 3.4运算器（ALU）

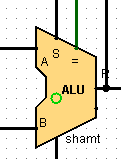


图3.5 运算器封装

1. FPGA实现：

运算器ALU的Verilog代码如下：

always@(\*)begin

case(alu\_op)

4'b0000:{result2,result}<=y<<shamt;

4'b0001:{result2,result}<=($signed(y))>>>shamt;

4'b0010:{result2,result}<=y>>shamt;

4'b0011:{result2,result}<=x\*y;

4'b0100:begin result<=x/y;result2<=x%y; end

4'b0101:{result2,result}<=x+y;

4'b0110:{result2,result}<=x-y;

4'b0111:begin result<=x&y;result2<=0;end

4'b1000:begin result<=x|y;result2<=0;end

4'b1001:begin result<=x^y;result2<=0;end

4'b1010:begin result<=~(x|y);result2<=0;end

4'b1011:begin result<=($signed(x)<$signed(y))?1:0;result2<=0;end

4'b1100:begin result<=($unsigned(x)<$unsigned(y))?1:0;result2<=0;end

endcase

end

1. 寄存器堆RF
2. Logism实现：

使用封装后的寄存器组，触发方式为下降沿触发。

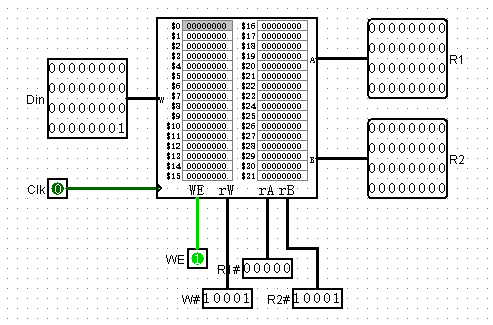


图 3.6寄存器堆（RF）

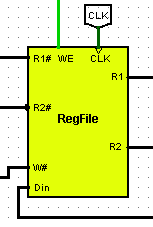


图3.7 寄存器堆封装

1. FPGA实现：

设置二位数组存放数据，每行有32个数字表示位宽为32，一共有32行表示有32个寄存器，以此作为寄存器堆RF。

寄存器堆RF的Verilog代码如下：

reg [31:0] regs[0:31];

initial begin

regs[0] <= 0;

end

assign A = regs[ra];

assign B = regs[rb];

always @(posedge clk) begin

if((WE == 1)&&rw!=0 ) regs[rw] = Din;

end

1. 数据存储器
2. Logism实现：

使用Logisim自带的随机存储器（RAM），并设置其地址位宽为10，数据位宽为32。

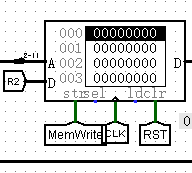


图 3.8数据存储器（ROM）

1. FPGA实现：

和指令存储器一样设置二位数组存放指令，但数据存储器采用四个二位数组实现，每行有8个数字表示位宽为8，一共有2^10行，表示取值范围皆可取到指令，将四个二维数组统一起来作为数据存储器，每个二维数组的具体实现在ram1k\_8中实现。其中A为输入数据存储器的PC地址，outD为从数据存储器读出的一个数据，sel为读出的位置（00：低8位--11：高8位），用来实现按字节读出。因数据存储器是一个RAM，要实现对数据存储器的写入操作，所以添加写入的数据inD。因考虑到最后在FPGA开发板上展示结果，要实现通过开发板上的开关选择，显示对应位置上的数据，所以添加一个输入data\_addr为开发板上输入的4位地址，添加一个输出data\_out为通过输入的四位地址读取到的32位数字。

数据存储器RAM的Verilog代码如下：

wire [9:0]sel\_data;

reg [31:0] inBytes;

always @(sel,inD) begin

case(sel)

4'b0000:inBytes= 32'b0;

4'b0001:inBytes= {24'b0,inD[7:0]};

4'b0010:inBytes= {24'b0,inD[15:8]};

4'b0100:inBytes= {24'b0,inD[23:16]};

4'b1000:inBytes= {24'b0,inD[31:24]};

4'b1100:inBytes= {16'b0,inD[31:16]};

4'b0011:inBytes= {16'b0,inD[15:0]};

4'b1111:inBytes= inD;

default:inBytes= 0;

endcase

end

assign sel\_data={6'd0,data\_addr};

wire [7:0] Byte0,Byte1,Byte2,Byte3;

ram1k\_8 ram0(.A(A),.inD(inBytes[7:0]) , .str(sel[0]&str),.clk(clk),.outD(Byte0),.sel\_data(sel\_data),.data\_out(data\_out[7:0]));

ram1k\_8 ram1(.A(A),.inD(inBytes[15:8]), .str(sel[1]&str),.clk(clk),.outD(Byte1),.sel\_data(sel\_data),.data\_out(data\_out[15:8]));

ram1k\_8 ram2(.A(A),.inD(inBytes[23:16]),.str(sel[2]&str),.clk(clk),.outD(Byte2),.sel\_data(sel\_data),.data\_out(data\_out[23:16]));

ram1k\_8 ram3(.A(A),.inD(inBytes[31:24]),.str(sel[3]&str),.clk(clk),.outD(Byte3),.sel\_data(sel\_data),.data\_out(data\_out[31:24]));

always @(sel,Byte0,Byte1,Byte2,Byte3) begin

case (sel)

4'b0000:outD = 0;

4'b0001:outD = {24'b0,Byte0};

4'b0010:outD = {24'b0,Byte1};

4'b0100:outD = {24'b0,Byte2};

4'b1000:outD = {24'b0,Byte3};

4'b1100:outD = {16'b0,Byte3,Byte2};

4'b0011:outD = {16'b0,Byte2,Byte1};

4'b1111:outD = {Byte3,Byte2,Byte1,Byte0};

default:outD = 32'd0;

endcase

end

ram1k\_8是一个位宽为8的数据存储器，其同样需要输入参数输入的地址A，读出的数据outD，输入的数据inD，写入信号str，以及选择显示的地址sel\_data和选择显示的数据data\_out。

ram1k\_8的Verilog代码如下：

module ram1k\_8

#(parameter BYTE\_WIDTH=8,parameter ADDR\_WIDTH=10)

(

input [(ADDR\_WIDTH-1):0]A,

input [(BYTE\_WIDTH-1):0]inD,

input str,

input clk,

output [(BYTE\_WIDTH-1):0]outD,

input [3:0] sel\_data,

output[7:0] data\_out

);

reg[(BYTE\_WIDTH-1):0] ram[(2\*\*ADDR\_WIDTH-1):0];

assign outD = ram[A];

always @ (posedge clk) begin

if(str) ram[A] <= inD;

end

assign data\_out = ram[sel\_data];

endmodule

### 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，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。

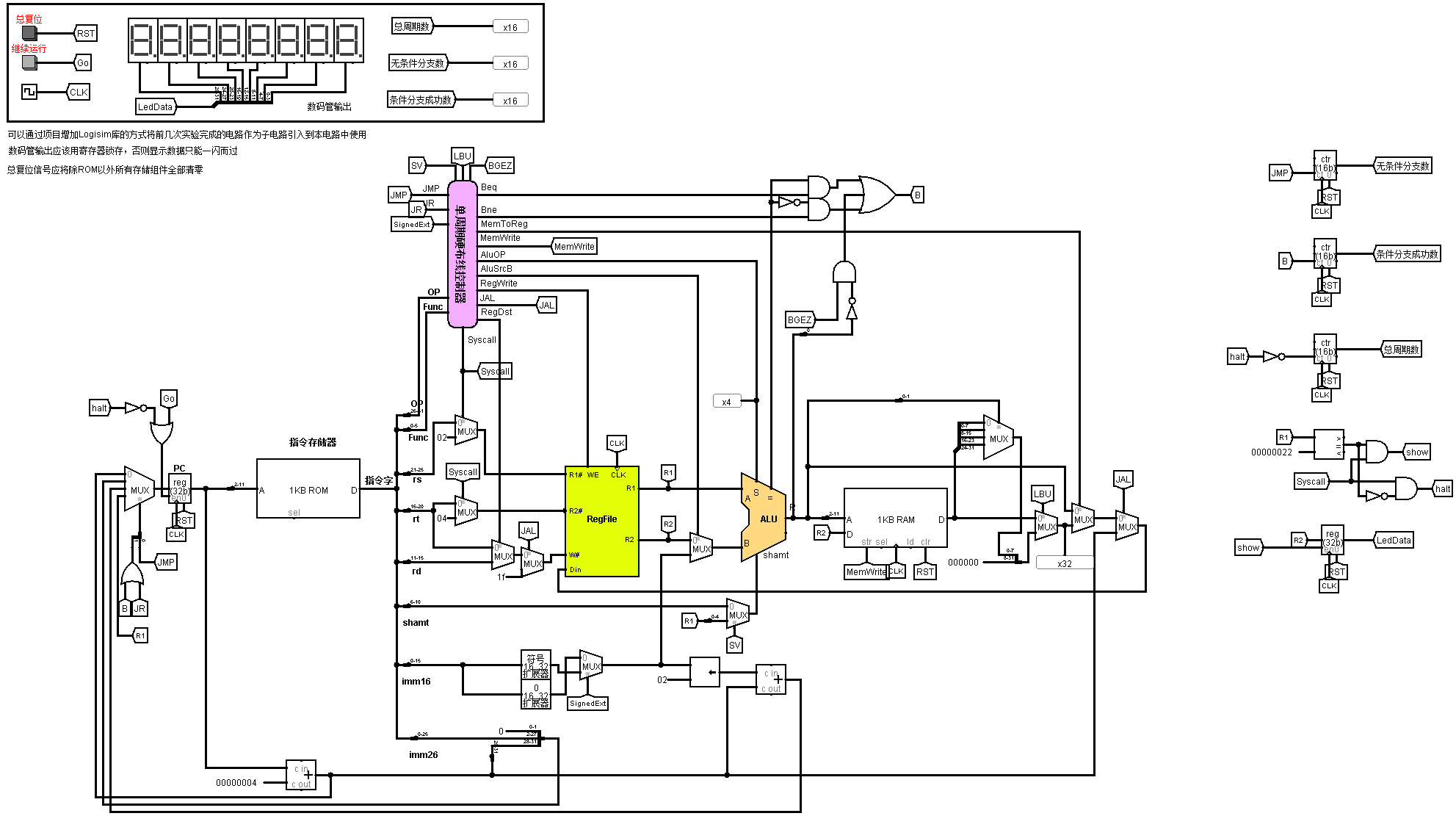


图 3.9 单周期CPU数据通路（Logism）

计数模块和显示模块则如下所示。

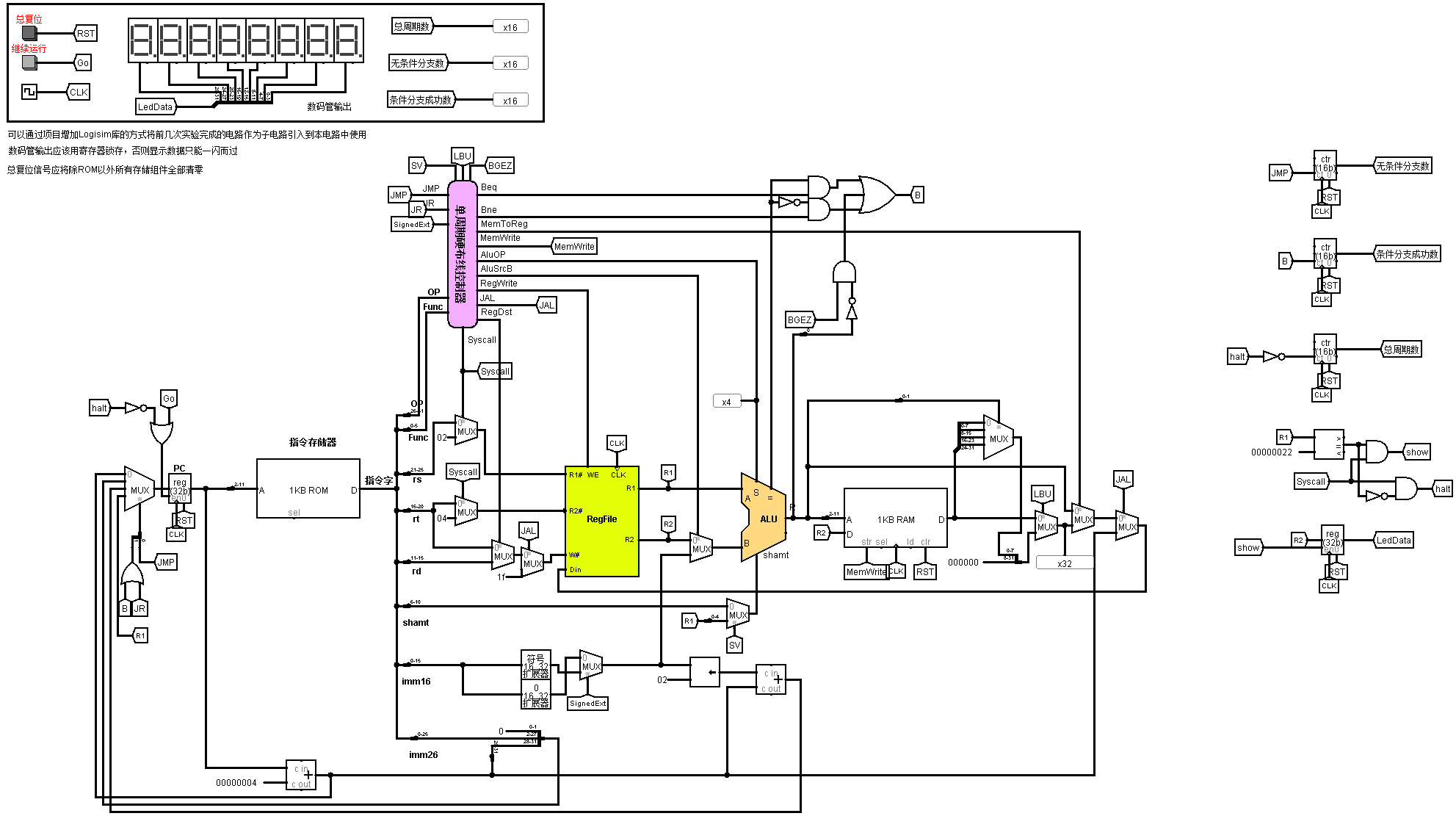


图 3.10 计数相关电路（Logism）

在Vivado中使用Verilog语言搭建的数据通路的原理图如图3.11所示。



图 3.11单周期CPU数据通路（FPGA）

### 控制器的实现

结合各指令的具体功能描述表格中指令与OP\_CODE和FUNC的对应关系和主控制器的信号框架表格，通过筛选可得到其对应表达式，形式如下所示。

RegWrite =

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 ~F5 ~F4 ~F3 ~F2 ~F1 ~F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 ~F5 ~F4 ~F3 ~F2 F1 F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 ~F5 ~F4 ~F3 ~F2 F1 ~F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 F5 ~F4 ~F3 ~F2 ~F1 ~F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 F5 ~F4 ~F3 ~F2 ~F1 F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 F5 ~F4 ~F3 ~F2 F1 ~F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 F5 ~F4 ~F3 F2 ~F1 ~F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 F5 ~F4 ~F3 F2 ~F1 F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 F5 ~F4 ~F3 F2 F1 F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 F5 ~F4 F3 ~F2 F1 ~F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 F5 ~F4 F3 ~F2 F1 F0 +

~OP5 ~OP4 ~OP3 ~OP2 OP1 OP0 +

~OP5 ~OP4 OP3 ~OP2 ~OP1 ~OP0 +

~OP5 ~OP4 OP3 OP2 ~OP1 ~OP0 +

~OP5 ~OP4 OP3 ~OP2 ~OP1 OP0 +

~OP5 ~OP4 OP3 ~OP2 OP1 ~OP0 +

~OP5 ~OP4 OP3 OP2 ~OP1 OP0 +

OP5 ~OP4 ~OP3 ~OP2 OP1 OP0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 ~F5 ~F4 ~F3 F2 F1 ~F0 +

~OP5 ~OP4 ~OP3 ~OP2 ~OP1 ~OP0 ~F5 ~F4 ~F3 F2 F1 F0 +

OP5 ~OP4 ~OP3 OP2 ~OP1 ~OP0

获得全部控制信号的对应表达式之后使用logisim生成电路，其中logisim中只能自动生成12个输出的电路，所以先生成除扩展信号以外的信号，之后再讲扩展信号补充进去。

1. Logism实现：

由于生成电路图片较大，所以展示封装后的控制器，如下图所示：。

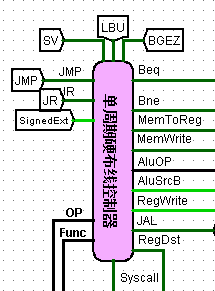


图 3.12 控制器封装（Logism）

1. FPGA实现

控制器具体可分为两部分，一部分产生ALU\_OP信号，一部分生成其他控制信号。设计如下所示：

ALU\_control calc\_signal(Func,OP,ALU\_OP);

GEN\_control cpu\_signal(Func,OP,MemToReg,MemWrite,ALU\_SRC,

RegWrite,SysCALL,SignedExt,RegDst,

Beq, Bne,JR,JMP,JAL,BGEZ,SV,LBU);

ALU\_control用来生成ALU\_OP，其具体如下所示：

assign ALU\_OP= (OP==0&&Func==0)?4'd0:

(OP==0&&(Func==3||Func==7))?4'd1:

(OP==0&&(Func==2||Func==6))?4'd2:

((OP==0&&(Func==32||Func==33))||OP==8||OP==9||OP==35||OP==43||OP==36)?4'd5:

(OP==0&&(Func==34||Func==34))?4'd6:

((OP==0&&Func==36)||OP==12)?4'd7:

((OP==0&&Func==37)||OP==13)?4'd8:

(OP==0&&Func==39)?4'd10:

((OP==0&&Func==42)||OP==10||OP==1)?4'd11:

(OP==0&&Func==43)?4'd12:4'd0;

GEN\_control用来生成其他控制信号，其具体设计如下所示：

assign MemToReg= (OP==6'd35||OP==36)?1:0;

assign MemWrite=(OP==6'd43)?1:0;

assign ALU\_SRC = (OP==6'd8)?1'b1:

(OP==6'd12)?1'b1:

(OP==6'd9)?1'b1:

(OP==6'd10)?1'b1:

(OP==6'd13)?1'b1:

(OP==6'd35||OP==36)?1'b1:

(OP==6'd43)?1'b1:1'b0;

assign RegWrite=((OP==0)&&(Func==6'd0||Func==6'd3||Func==6'd2||Func==6'd32

||Func==6'd33||Func==6'd34||Func==6'd36||Func==6'd37

||Func==6'd39||Func==6'd42||Func==6'd43||Func==6||Func==7))||

(OP==6'd3||OP==6'd8||OP==6'd12||

OP==6'd9||OP==6'd10||OP==6'd13||OP==6'd35||OP==36)?1'b1:1'b0;

assign SysCALL=((OP==6'd0)&&(Func==6'd12))?1'b1:1'b0;

assign SignedExt= (OP==6'd8||OP==1||OP==36)?1'b1:

(OP==6'd9)?1'b1:

(OP==6'd10)?1'b1:

(OP==6'd35)?1'b1:

(OP==6'd43)?1'b1:

(OP==6'd40)?1'b1:1'b0;

assign RegDst=(OP!=0)?1'b0:

((Func==0)||(Func==3)||(Func==2)||

(Func==32)||(Func==33)||(Func==34)||

(Func==36)||(Func==37)||(Func==39)||

(Func==42)||(Func==43)||Func==6||Func==7)?1'b1:1'b0;

assign Beq=(OP==4)?1'b1:1'b0;

assign Bne=(OP==5)?1:0;

assign JR=((OP==0)&&(Func==8))?1'b1:1'b0;

assign JMP=((OP==0)&&(Func==8)||(OP==2)||(OP==3))?1'b1:1'b0;

assign JAL=(OP==3)?1'b1:1'b0;

assign BGEZ=(OP==1)?1'b1:1'b0;

assign SV=((OP==0)&&(Func==6||Func==7))?1'b1:1'b0;

assign LBU=(OP==36)?1'b1:1'b0;

以此类推，最终便可以实现整个主控制器中所有控制信号的生成。在Vivado中使用Verilog语言构成的RTL图如下所示。

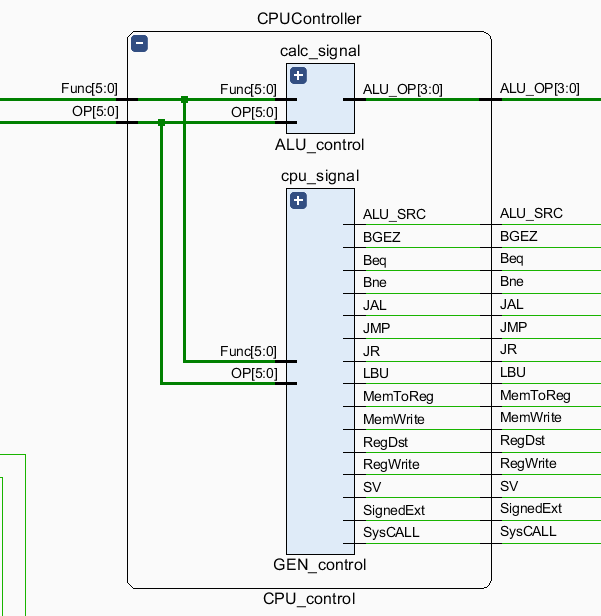


图 3.13主控制器封装（Vivado）

### 单周期CPU实现

统合以上三个部分的设计可完成单周期CPU的设计，其最终展示结果如下所示：

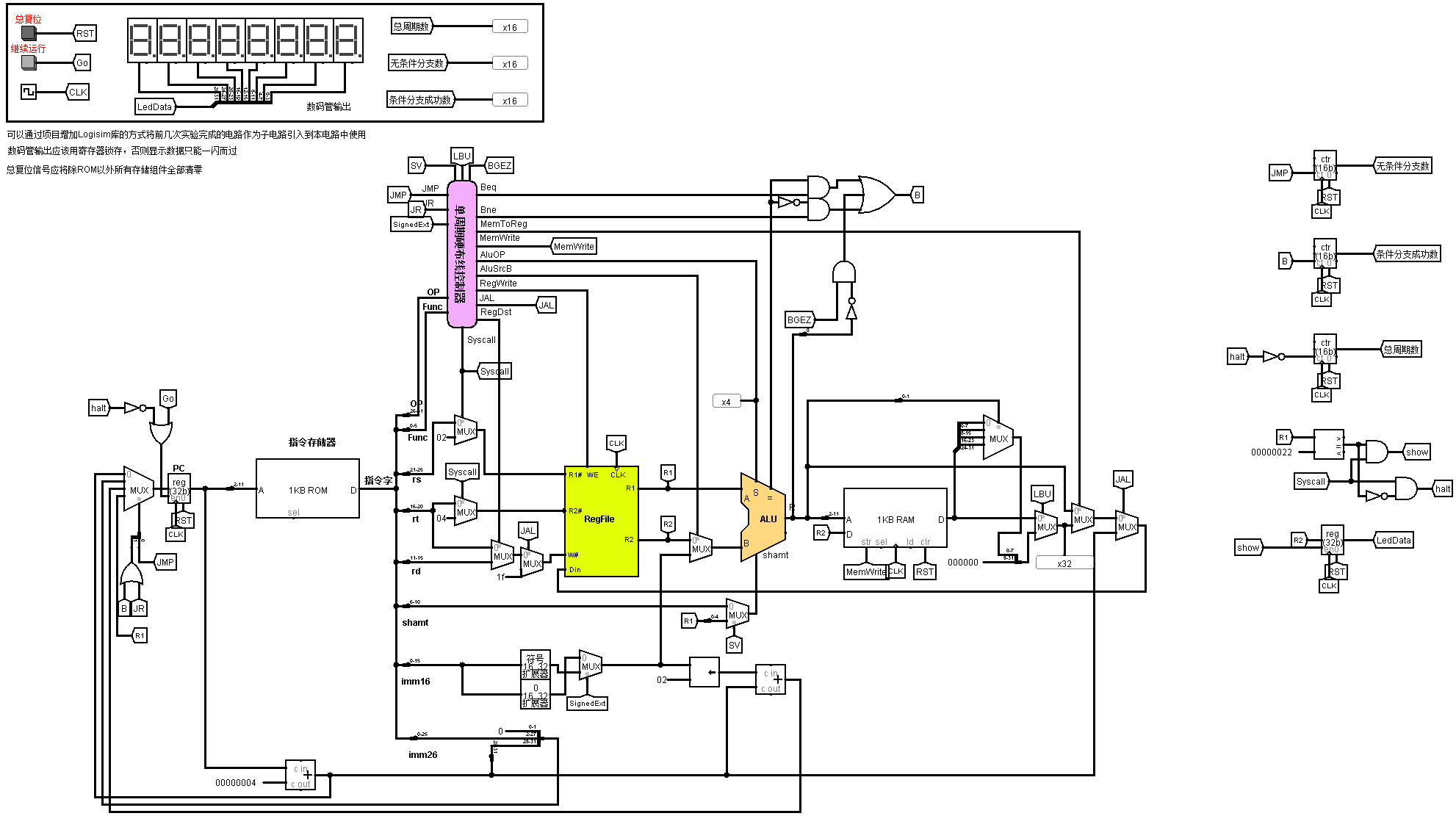


图 3.14 单周期CPU

## 中断机制实现

### 硬件实现

终端信号产生电路如下所示，当按下中断按钮时，会产生对应的中断信号，且该信号一直维持到出现RST信号才清零，完成多级中断需要将以下电路复制三份并修改对应标签。

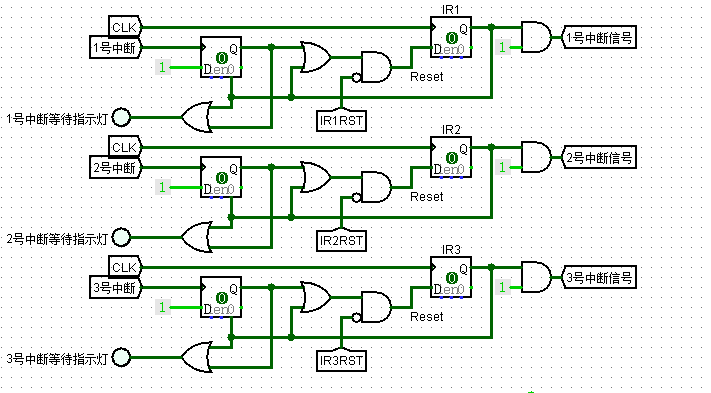


图 3.9 中断信号产生电路

RST信号和中断程序地址电路如下所示，首先从三个中断信号中根据优先编码原则选出当前应执行的中断程序号存入int\_num，通过该编号选择中断程序地址和在ERET信号下选择RST清零信号。其中三个中断程序对应的程序地址是修改benchmark代码后使用Mars软件单步运行时观察到地址，本次实验采用直接定下中断程序入口地址的方式。

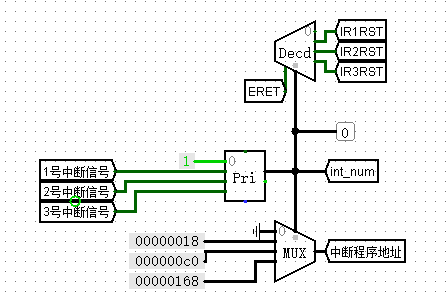


图 3.11 中断程序地址选择电路

当运行到中断相关指令是首先要对IR进行译码，产生对应的MTC、MFC、IEWrite、EPCWrite、EPCRead等信号。

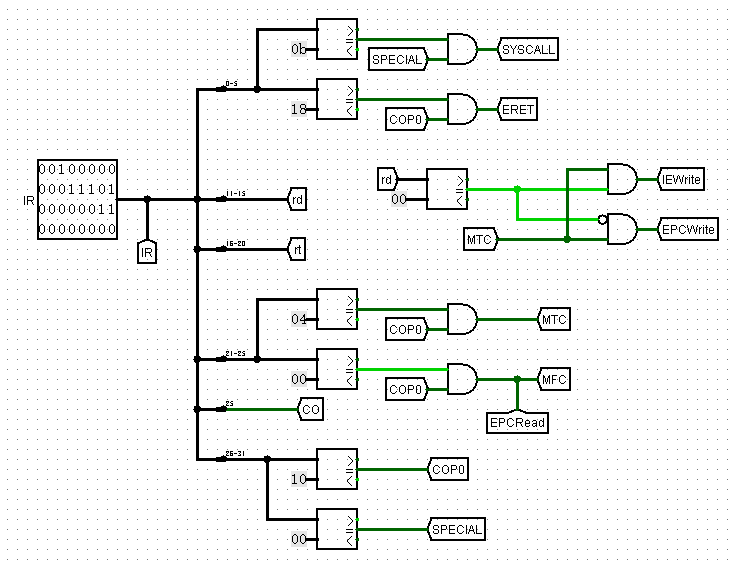


图 3.12 中断指令译码电路

处理多级中断的电路如下所示，其中hasIR信号表明存在中断被打断的现象，若存在hasIR信号，则通过多级中断处理电路，在开终端（IE为1）时，对三个中断的顺序进行变动，转移到新的中断中去，同时在ERET信号产生时将返回到上一个中断。

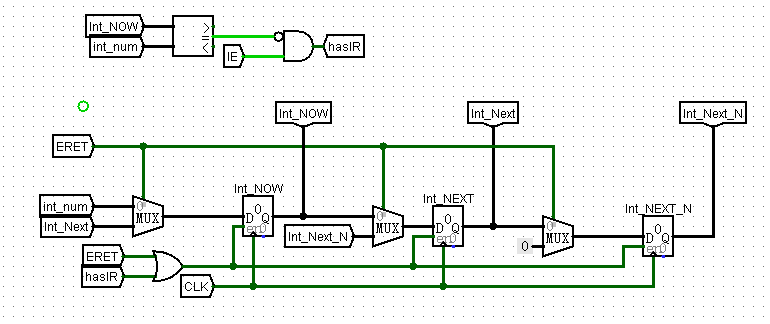


图 3.13 多级中断处理电路

EPC相关电路如下所示，首先在IEWrite信号下要将R2的末位写入IE寄存器，其余时候如果存在hasIR信号或IE信号时IE要一直维持高电平，仅在ERET信号产生是需要修正为0，表示关中断此时不能进行上图所示多级中断处理电路。若无中断相关操作，PCout的值就是PCin；若存在中断信号则根据软件部分产生EPCWrite信号会选择将PC写入EPCout；若存在多级中断则是将当前EPC存入EPCout；若存在ERET信号，则返回EPC作为PCout，用来返回程序。

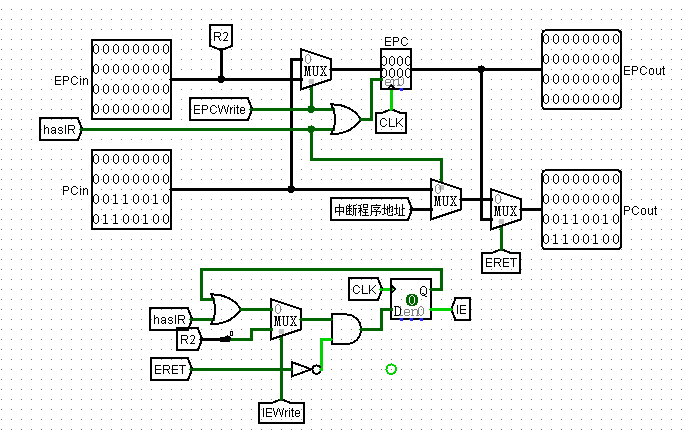


图 3.14 EPC相关

将以上电路封装为CP0，封装效果如下所示：

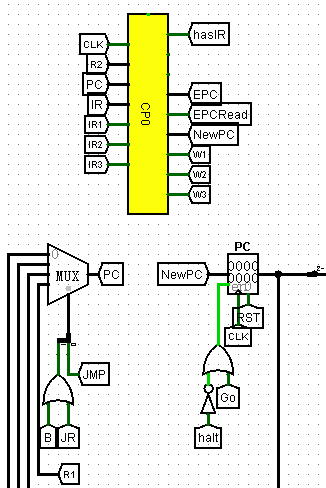


图 3.15 CP0封装

### 软件实现

在代码中添加保护现场和恢复现场的部分，具体如下IR1所示，需要在三个中断处理程序中做同样的操作：

IR1:

# protect env

addi $sp, $sp, -4

sw $t0, 0($sp)

addi $sp, $sp, -4

sw $t1, 0($sp)

addi $sp, $sp, -4

sw $s0, 0($sp)

addi $sp, $sp, -4

sw $s1, 0($sp)

addi $sp, $sp, -4

sw $a0, 0($sp)

addi $sp, $sp, -4

sw $v0, 0($sp)

mfc0 $t0, $1

addi $sp, $sp, -4

sw $t0, 0($sp)

#Open Interrupt

addi $t0, $zero, 0

mtc0 $t0, $0

# IR1 program

addi $s1,$zero,1 #中断号1,2,3 不同中断号显示值不一样

add $s0,$zero,$s1

IntLeftShift1:

sll $s0, $s0, 4

add $a0,$0,$s0 #display $s0

addi $v0,$0,34 # display hex

syscall # we are out of here.

bne $s0, $zero, IntLeftShift1

#Close Interrupt

addi $t0, $zero, 1

mtc0 $t0, $0

# restore env

lw $t0, 0($sp)

addi $sp, $sp, 4

mtc0 $t0, $1 #restore epc

lw $v0, 0($sp)

addi $sp, $sp, 4

lw $a0, 0($sp)

addi $sp, $sp, 4

lw $s1, 0($sp)

addi $sp, $sp, 4

lw $s0, 0($sp)

addi $sp, $sp, 4

lw $t1, 0($sp)

addi $sp, $sp, 4

lw $t0, 0($sp)

addi $sp, $sp, 4

eret

设计程序的开头，直接设置跳转到benchmark指令和三个中断处理程序，用来得到对应的中断程序入口地址，具体设计如下所示：

addi $sp,$zero,0x300

addi $t0, $zero, 10

j benchmark\_start

j IR1

j IR2

j IR3

### 单周期CPU+多级中断

在单周期CPU中首先添加CP0处理PC的值，电路图如下图所示：

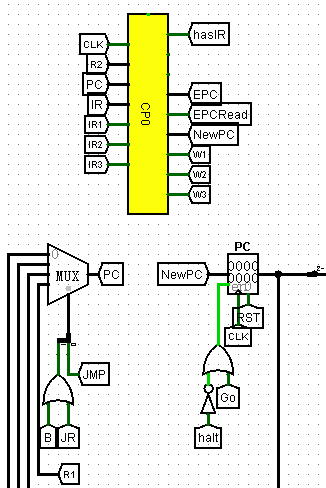


图 3.16 单周期CPU+多级中断

之后由于要将EPC写入数据存储器，所以EPCRead信号也需要控制寄存器堆的WE段，同时EPC将是Din的一种输入，电路如下所示：

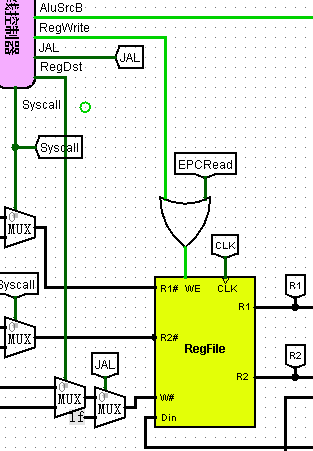


图 3.17 添加EPCRead信号

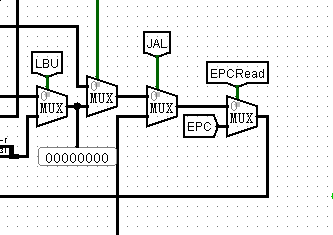


图 3.18 添加EPCRead信号

### 重定向流水线+多级中断

设计时选择在EX段实现中断，首先在EX段添加CP0，将其PC输入设为IF.PC/ID.PC，即下一条有效的PC。

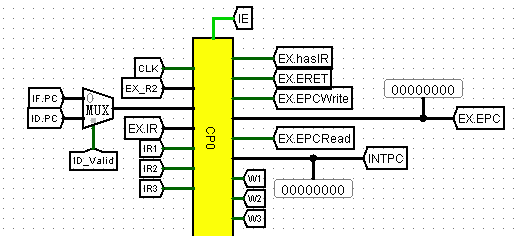


图 3.19 重定向流水线+多级中断

在WB阶段，由于没有将中断相关信号传递过来所以需要堆IR进行译码获得当前信号，用此信号控制寄存器堆的WE信号和Din的选择。

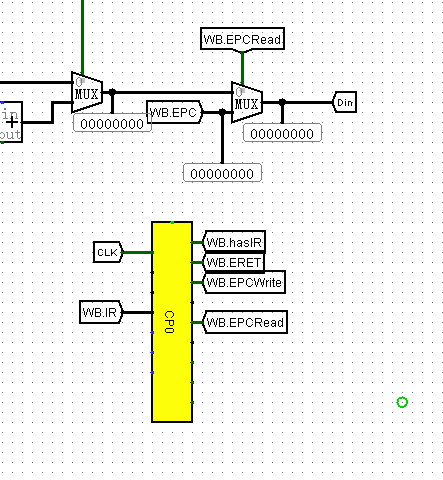


图 3.20 重定向流水线+多级中断

## 流水CPU实现

### 流水接口部件实现

流水寄存器主要用于暂存数据，并将上一阶段的数据在下一个时钟周期传入下一个阶段中，由于整个流水线为上升沿触发，所以流水寄存器也要实现上升沿触发且同步清零功能。

1. Logism实现：

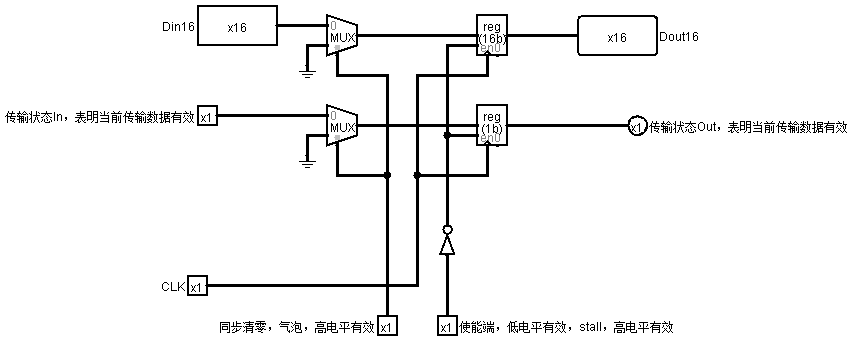


图 3.21 流水寄存器

以上电路为海明编码流水传输中的流水寄存器电路，是一个典型的同步电路。在具体实现中均依靠本电路进行扩展，将设计部分所分析出的需要向后传递的信号全部传递过去，依此实现IF/ID、ID/EX、EX/MEM、MEM/WB四个流水寄存器。

1. FPGA实现：

与logisim设计相同需要注意同步清零问题，由于完整程序代码较唱，以下仅举例IF/ID模块的代码，其中后缀为in的为IF段传入的数据，后缀为out的为传递到ID段的数据。其余三个流水寄存器均与之类似。

IF/ID的Verilog代码如下：

initial begin

IRout <= 0;

PCout <= 0;

end

always@(posedge clk)begin

if(en==1);

else if(rst==1)begin

IRout <= 0;

PCout <= 0;

end

else begin

PCout <= PCin;

IRout <= IRin;

end

end

### 理想流水线实现

将单周期CPU的最终设计图进行分割，同时根据接口部件设计完成流水寄存器，最终线路图如下所示：

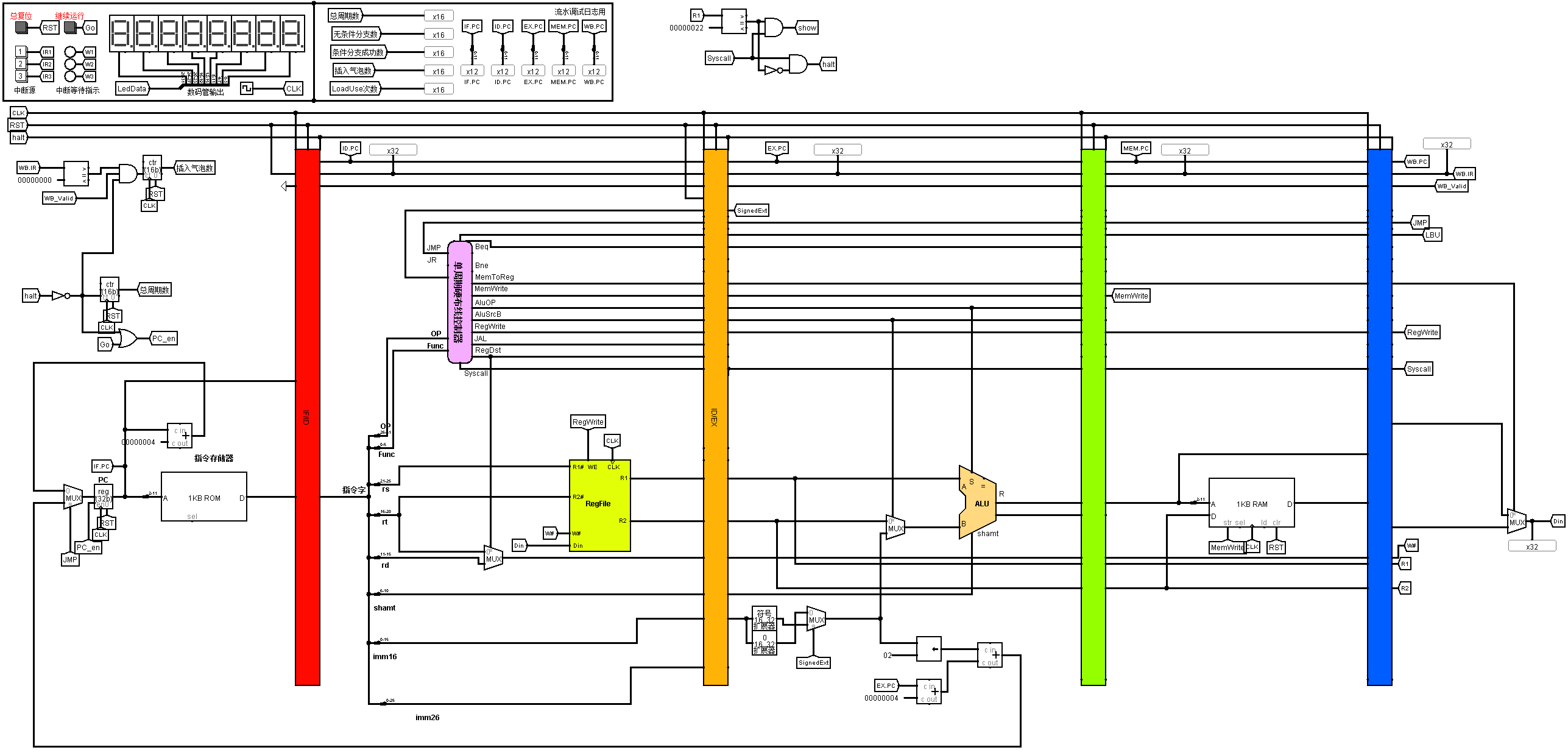


图 3.22 理想流水线

## 气泡式流水线实现

首先完善分支指令的数据通路，具体如下图所示，

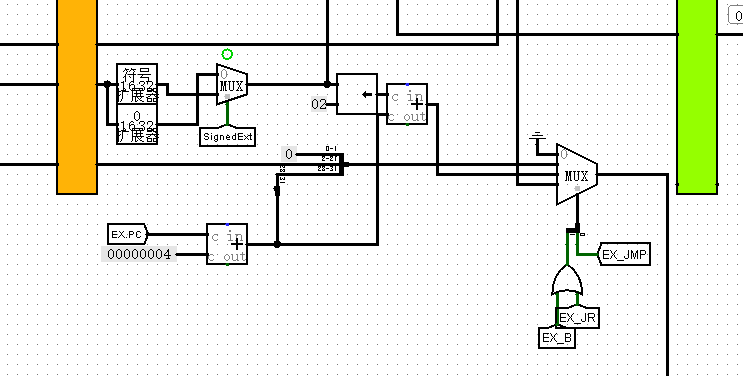


图 3.23 有效PC生成电路

其次完成数据相关子电路，其中要根据气泡流水线设计中的源寄存器使用情况表格自动生成其对应的子电路，对齐进行封装后实现数据相关检测电路，如下所示：

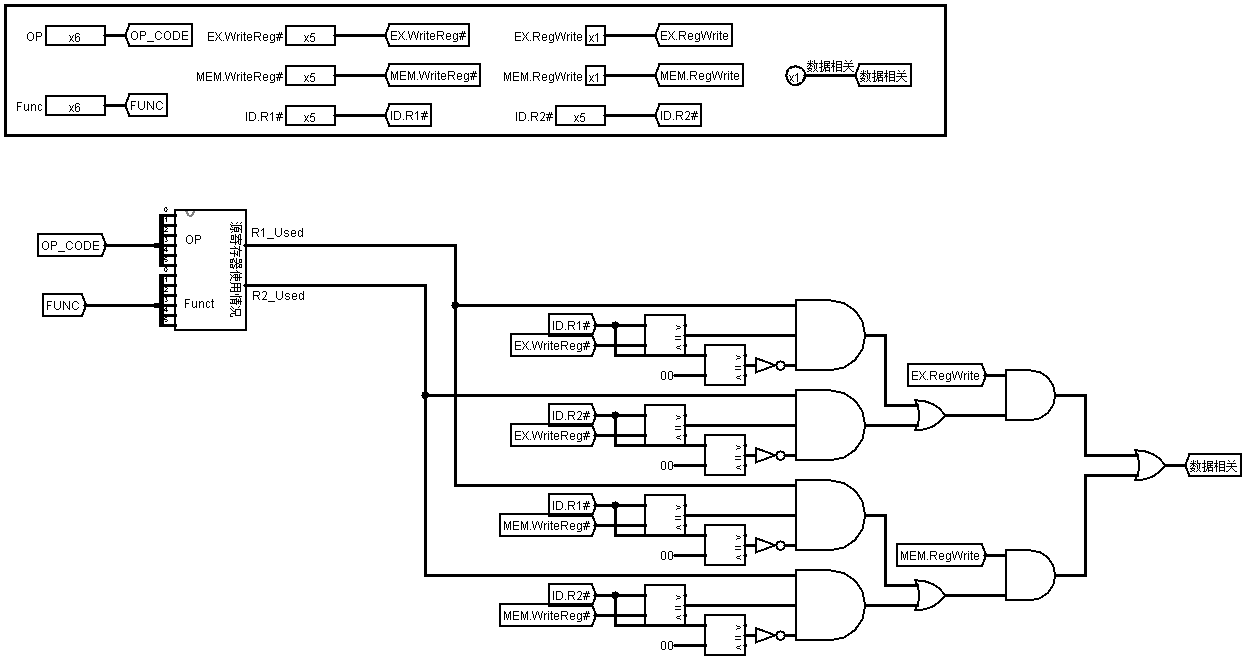


图 3.24 数据相关电路

通过数据相关检测电路产生一个数据相关的信号，该信号需要控制PC的使能端、IF/ID的使能端、ID/EX段的清零端。

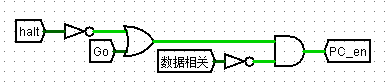


图 3.25 PC使能端

完整电路如下图所示：

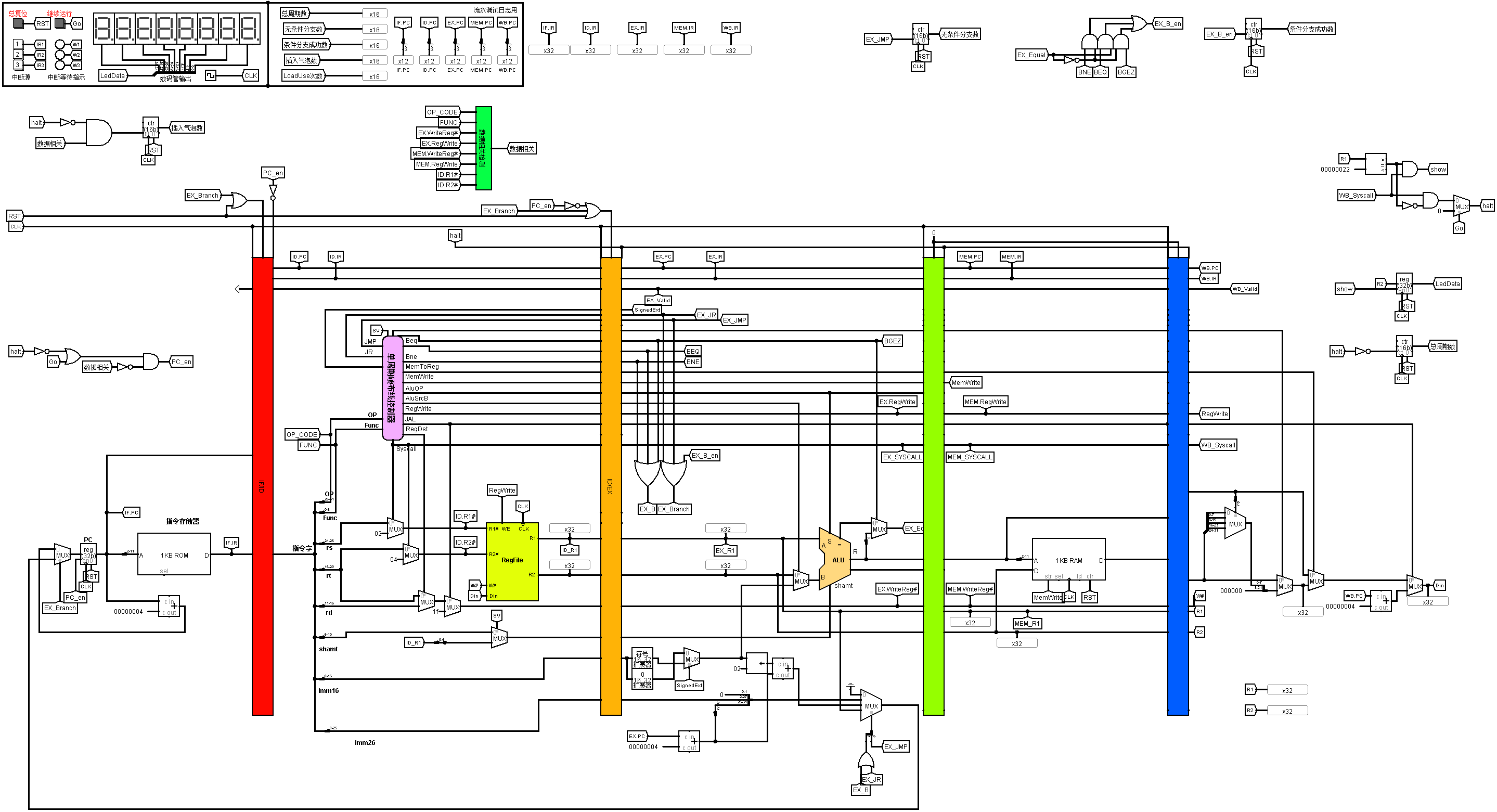


图 3.26 气泡流水线

## 重定向流水线实现

在流水线上板步骤中选择的是重定向流水线，具体实现如下所述。

1. Logism实现：

根据重定向流水线的设计部分，首先完成冲突信号的产生电路，具体如下图所示

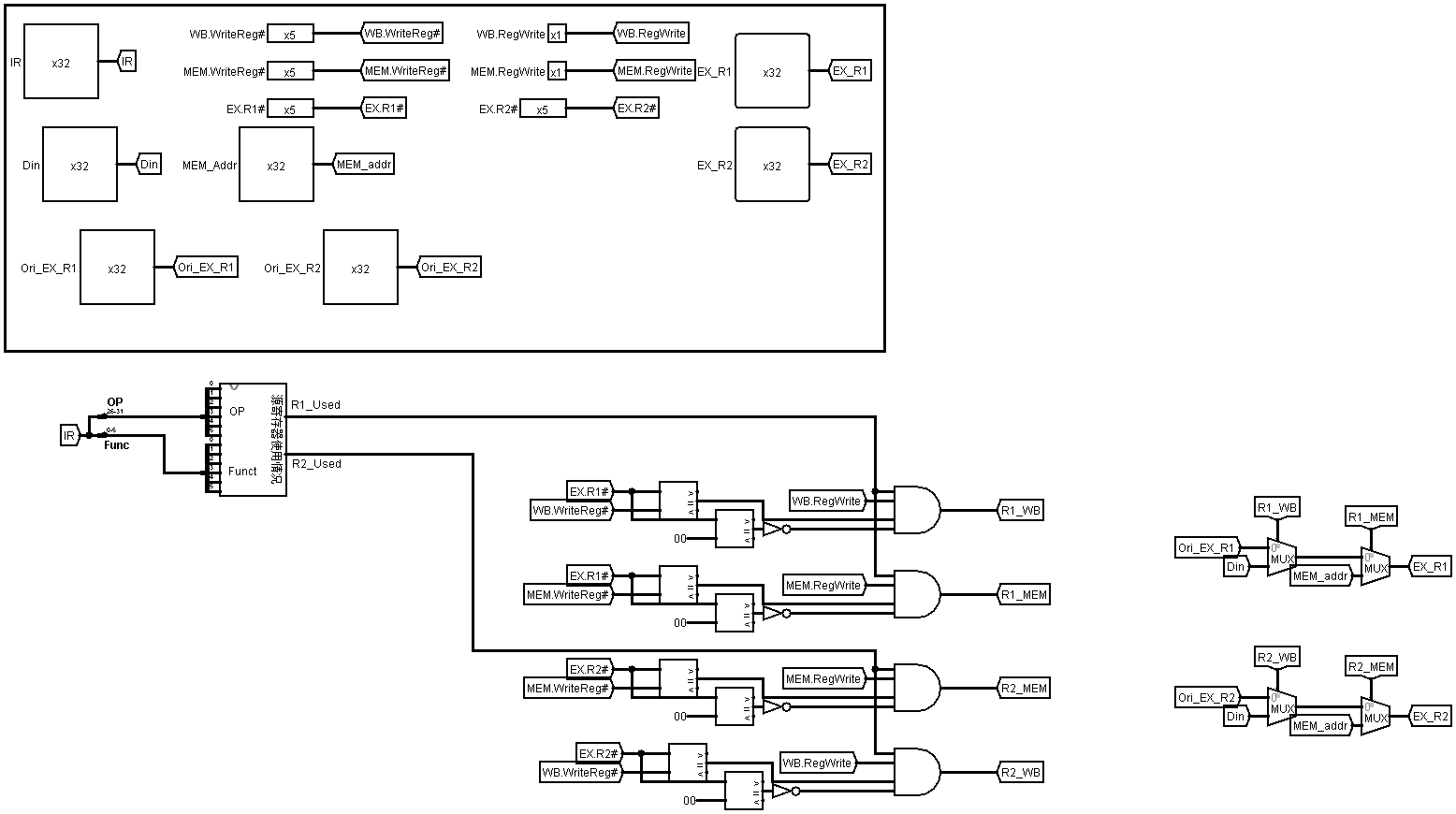


图 3.27 重定向数据相关

之后完成选择结果的数据通路，具体如下图所示

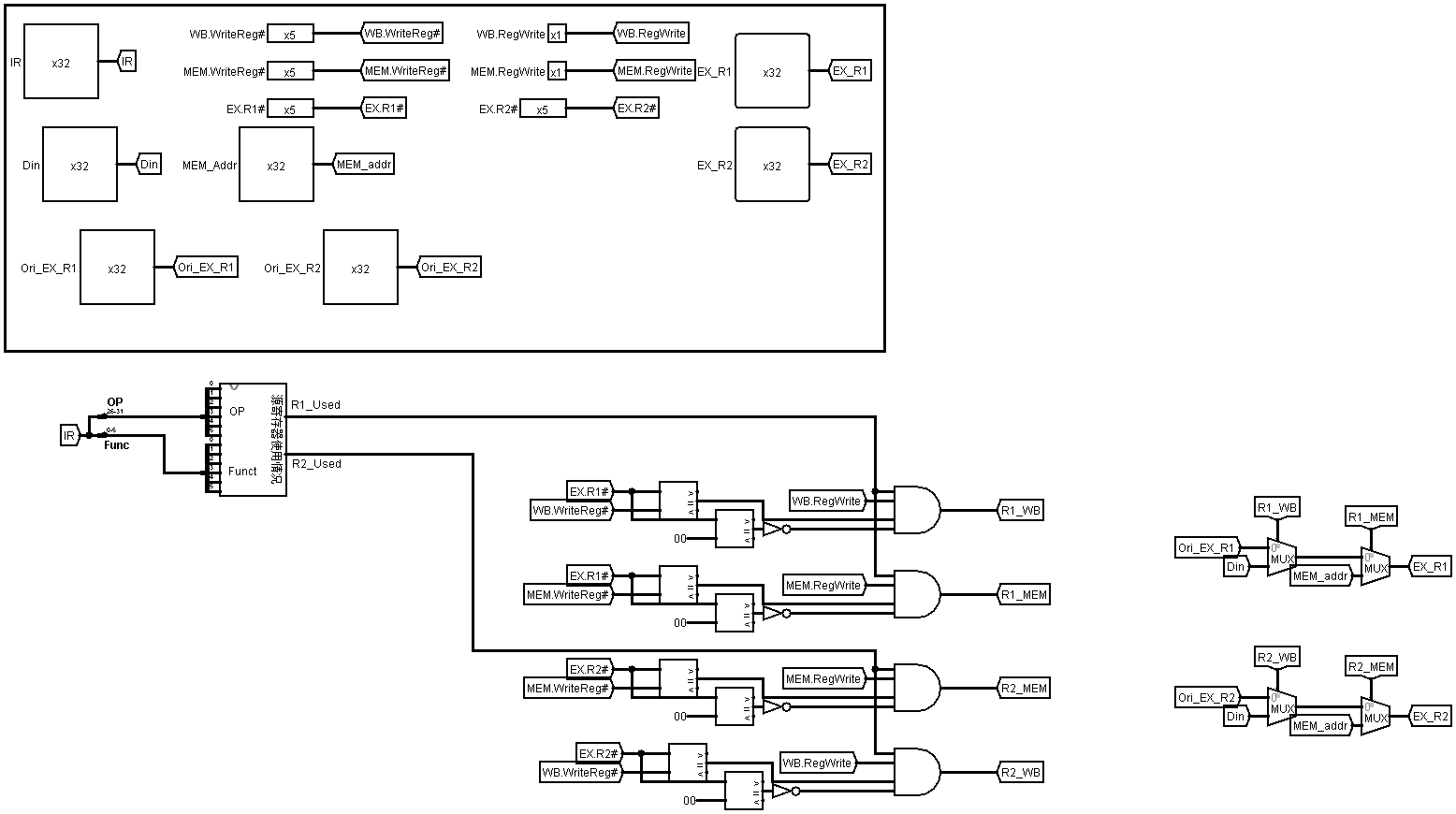


图 3.28 重定向数据相关

1. FPGA实现：

对应处理redirect过程的Verilog代码如下：

wire [5:0]op = IR[31:26];

wire [5:0]func = IR[5:0];

wire R1\_Used, R2\_Used;

assign R1\_Used = (((op==6'd0)&&(func==6'd32||func==6'd33||func==6'd34||func==6'd36||func==6'd37||func==6'd39

||func==6'd42||func==6'd43||func==6'd8||func==6'd12||func==6'd6||func==6'd7))

||(op==6'd4||op==6'd5||op==6'd8||op==6'd12||op==6'd9||op==6'd10||op==6'd13||op==6'd35||op==6'd43||op==6'd36||op==6'd1))?1'b1:1'b0;

assign R2\_Used = (((op==6'd0)&&(func==6'd0||func==6'd3||func==6'd2||func==6'd32||func==6'd33||func==6'd34||func==6'd36||func==6'd37||func==6'd39||func==6'd42||func==6'd43||func==6'd12||func==6'd6||func==6'd7))

||(op==6'd4||op==6'd5||op==6'd43))?1'b1:1'b0;

wire R1\_WB,R1\_MEM,R2\_WB,R2\_MEM;

assign R1\_WB = (EX\_RA==WB\_RW) && (EX\_RA!=6'd0) && R1\_Used && WB\_RegWrite;

assign R1\_MEM = (EX\_RA==MEM\_RW) && (EX\_RA!=6'd0) && R1\_Used && MEM\_RegWrite;

assign R2\_WB = (EX\_RB==WB\_RW) && (EX\_RB!=6'd0) && R2\_Used && WB\_RegWrite;

assign R2\_MEM = (EX\_RB==MEM\_RW) && (EX\_RB!=6'd0) && R2\_Used && MEM\_RegWrite;

assign EX\_R1 = R1\_MEM ? MEM\_addr : (R1\_WB ? Din : Ori\_EX\_R1);

assign EX\_R2 = R2\_MEM ? MEM\_addr : (R2\_WB ? Din : Ori\_EX\_R2);

同时处理LoadUse现象的时候引入dependency信号，使用该信号表示此时是LoadUse现象，要注意对各部件使能端和清零端的控制。

对应的产生dependency信号的Verilog代码如下：

initial begin

IRout <= 0;

PCout <= 0;

end

always@(posedge clk)begin

if(en==1);

else if(rst==1)begin

IRout <= 0;

PCout <= 0;

end

else begin

PCout <= PCin;

IRout <= IRin;

end

end

同时微改数据通路相关代码，最终实现的重定向流水线RTL图如下所示：

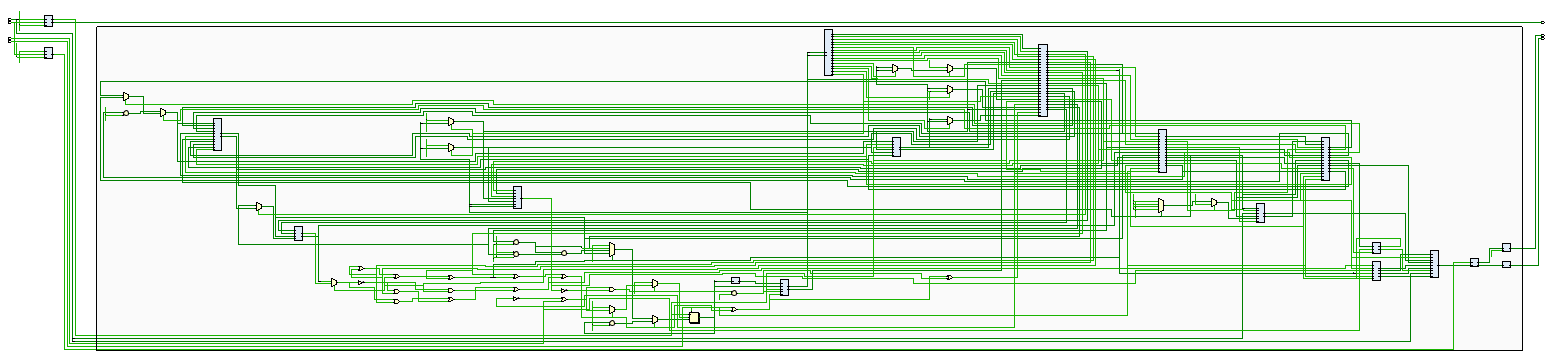


图 3.27 重定向流水线数据通路（FPGA）

## 动态分支预测机制实现

### 预测位状态转移电路

根据设计部分的状态转移图完成如下状态转移电路：

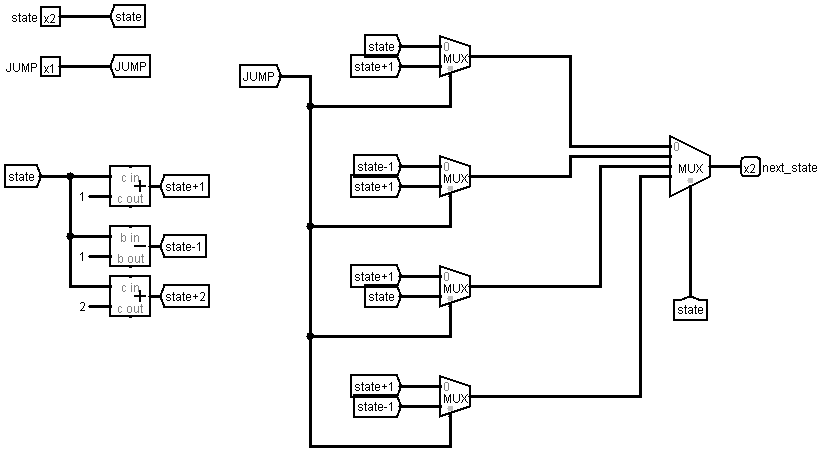


图 3.28 预测状态状态转移图

### Cache

设计Cache槽，如下图所示。VAILD数据位输出V0，标记位输出到Tag0，淘汰标志位输出到C0，预测位输出到state0，数据块副本输出到SlotData。Set0表示第0行被选中作为将要写入的行数即第0行要被替换的信号，L0表示第0行在IF段命中，S0段表示第0行在EX段命中。两者的区别在于在IF段命中且判断预测位state是否大于1，若大于1则是预测跳转的情况，而在EX段命中仅仅用于更新预测位state的数据。将下图所示cache槽赋值八份修改其中信号的隧道标签使其构成有效的电路。

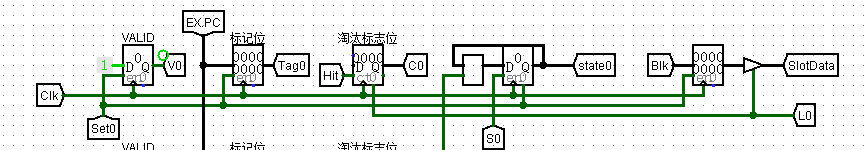


图 3.29 Cache槽

之后使用IF.PC和EX.PC分别产生L0、S0、hit、EX\_hit等信号，具体电路如下所示：

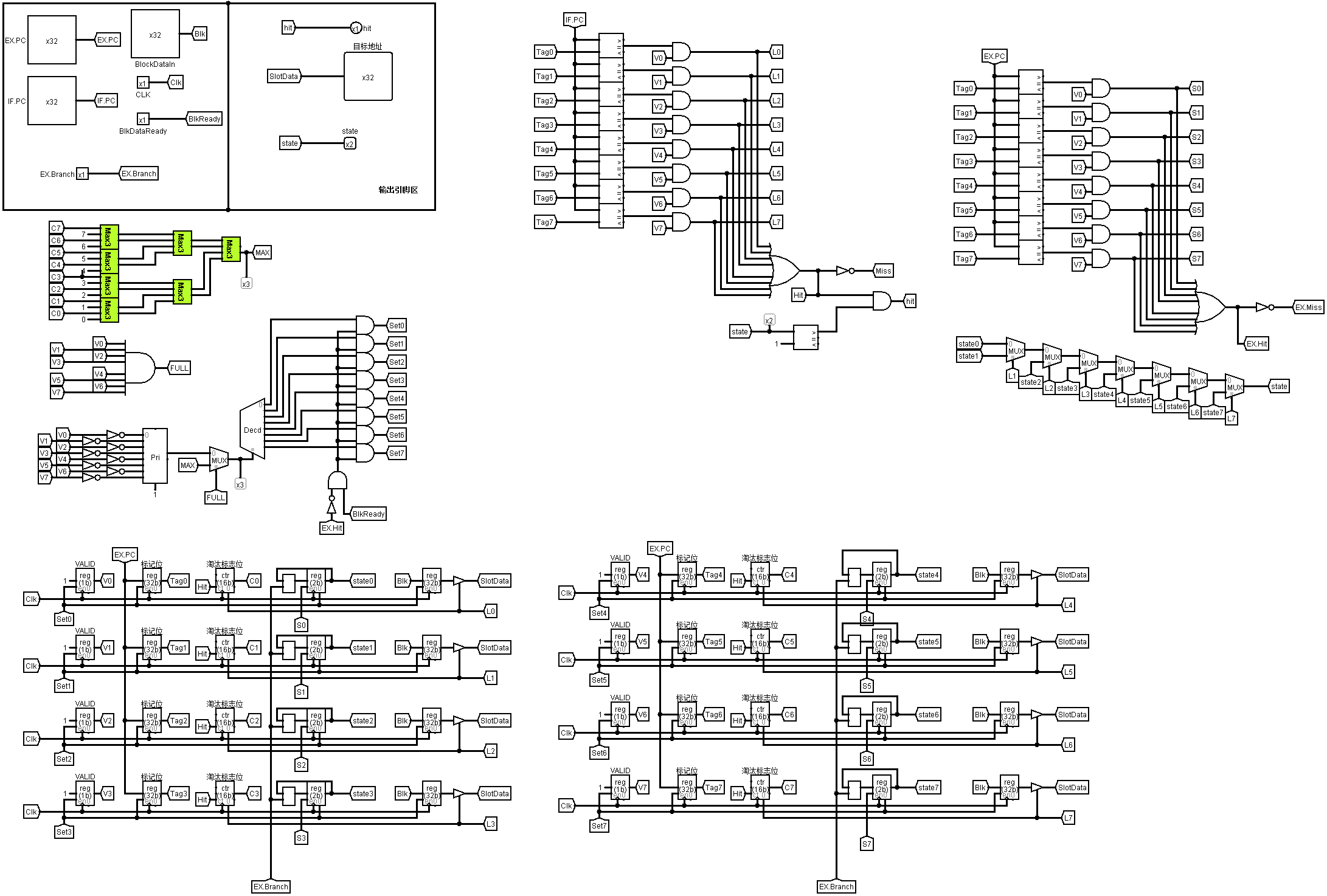


图 3.30 控制信号生成电路

最后通过比较器、优先编码器等完成对Cache槽的选择电路，具体如下所示：

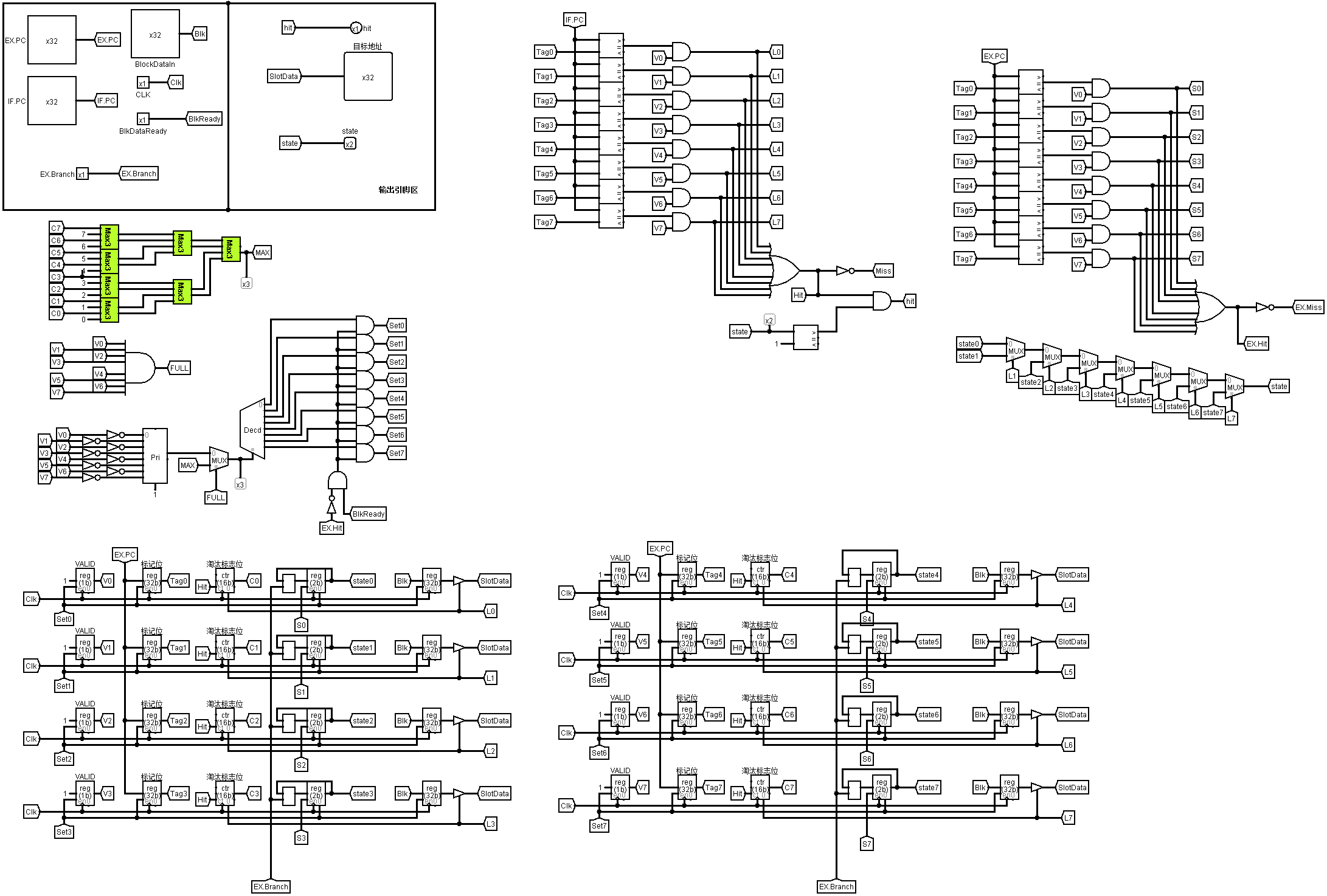


图 3.31 控制信号生成电路

整个cache的电路图如下所示：

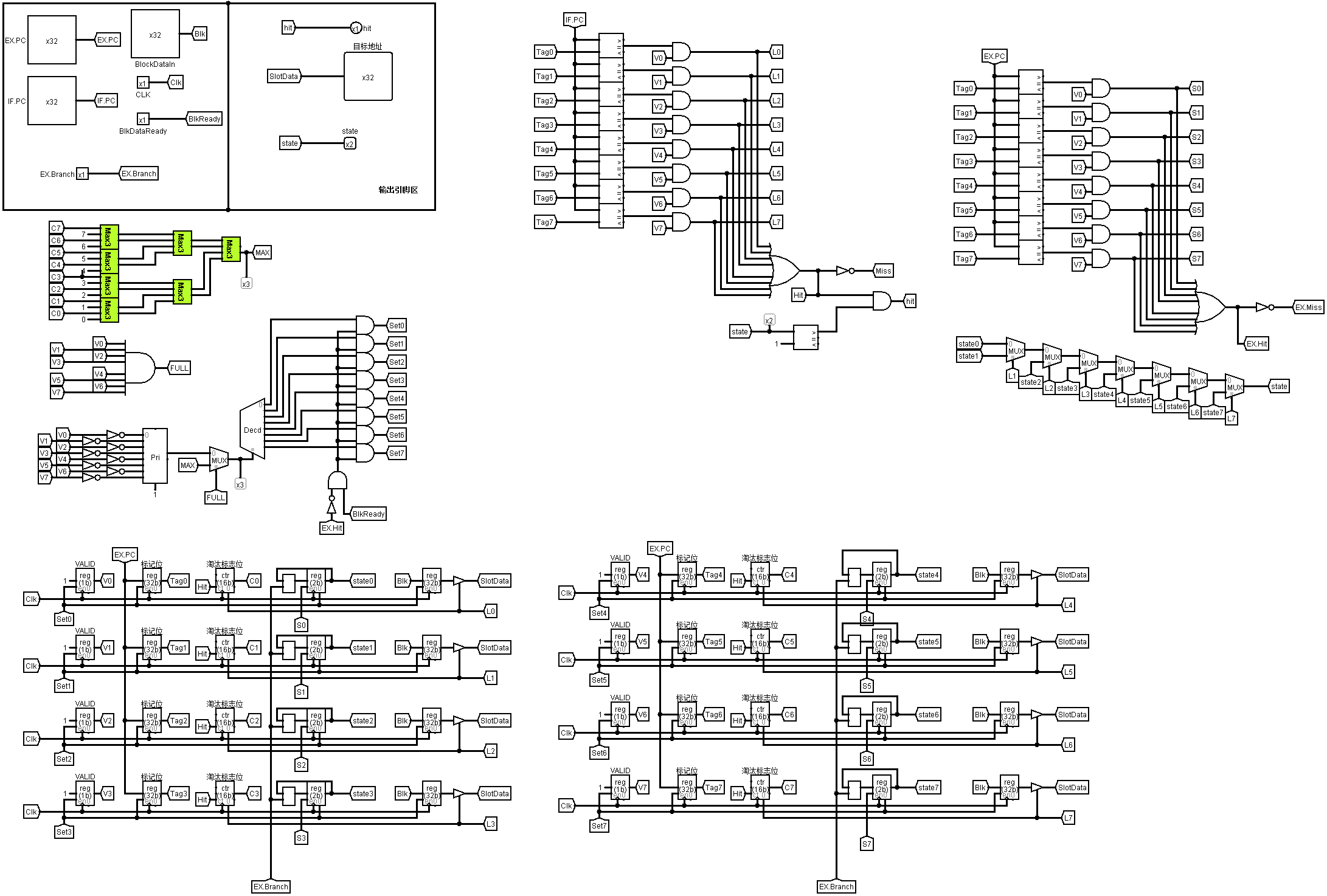


图 3.32 Cache完成电路

### 分支预测子电路

分支预测子电路中包含BHT和相关信号的产生，封装之后的效果如下所示。

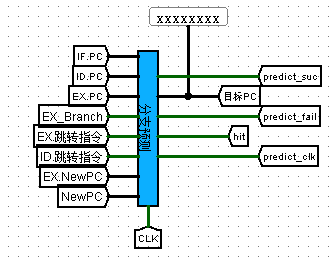


图 3.33 分支预测封装

其中EX.NewPC指EX段根据跳转信号选出的地址，NewPC指的是正确的下一条指令地址，通过设计部分对三个predict相关信号的描述，在分支预测子电路中完成三个信号的生成，具体设计如下所示：

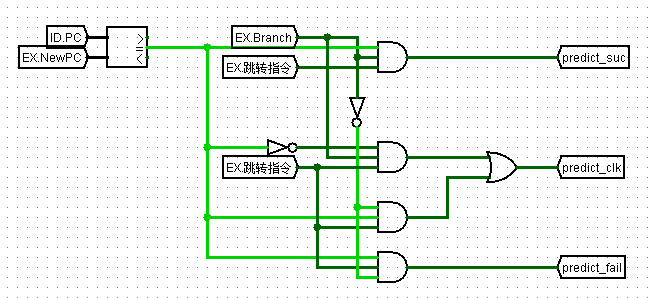


图 3.34 分支预测相关信号生成电路

### 主数据通路

在主数据通路中，分支预测主要影响到的是IF段的PC值，分析可得，在hit的情况下应选择分支预测电路得到的目标PC，但如果存在predict\_clk信号，即分支预测失败或需要清零时，应输入的EX段得到的NewPC，其余时候PC的值位IF段的PC+4，因此具体电路如下所示：

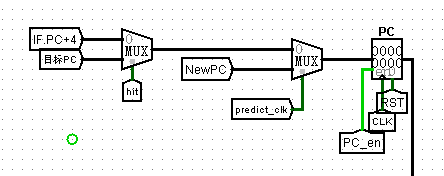


图 3.35 添加PC控制

# 实验过程与调试

## 测试用例和功能测试

测试过程除中断外采用的是老师给的标准程序，观察其运行现象和计数结果与标准结果进行对比进行测试，由于进行的测试过程较多且均重复操作，所以在测试用例中仅写明一列logisim下测试结果和vivado下测试结果，其余测试仅将结果在性能分析中列表记录，而测试中遇到的问题将写在主要故障和调试部分。

### logisim-重定向流水线

将扩展四条指令对应程序填入benchmark.asm的对应位置中，之后转化为.hex后缀文件，在Logisim中导入镜像，开启时钟模拟，其计数结果如下所示：

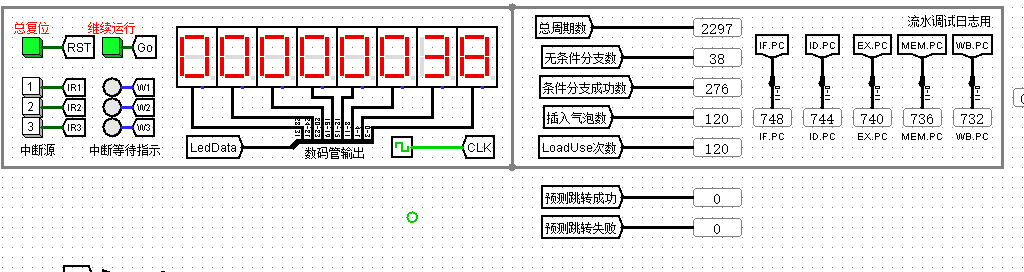


图 4.1 benchmark运行结果

按下Go按钮运行完成SRLV指令后记数结果如下所示：

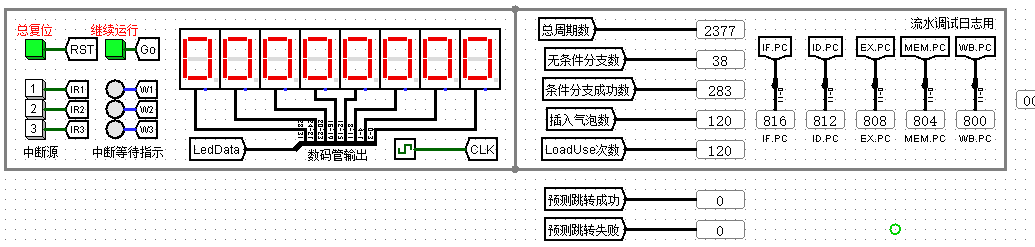


图 4.2 benchmark+srlv运行结果

按下Go按钮运行完成SRAV指令后记数结果如下所示：

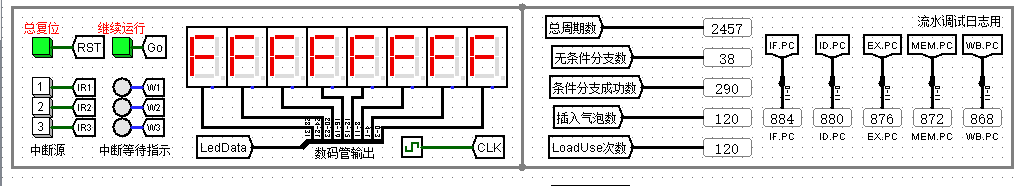


图 4.3 benchmark+srlv+srav运行结果

按下Go按钮运行完成LBU指令后记数结果如下所示：

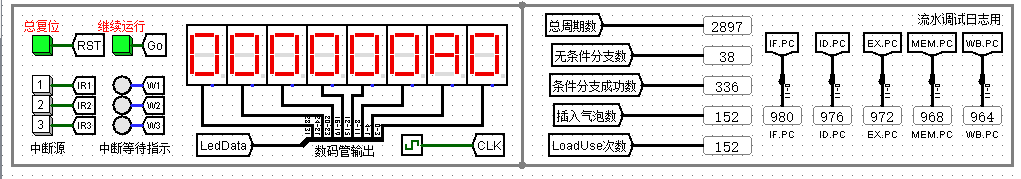


图 4.4 benchmark+srlv+srav+lbu运行结果

按下Go按钮运行完成BGEZ指令后记数结果如下所示：

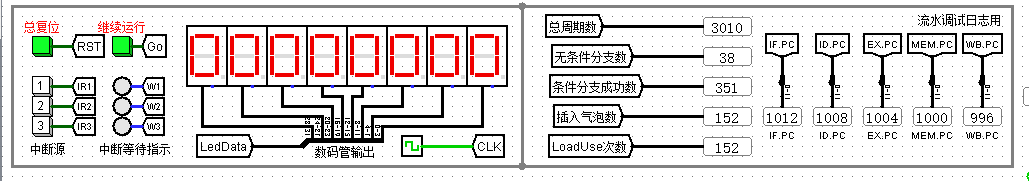


图 4.5 benchmark+srlv+srav+lbu+bgez运行结果

再次按下Go按钮将重新运行benchmark。

### vivado-重定向流水线

测试程序sim\_CPU.v如下所示：

reg clk, reset;

reg[2:0] display\_switch;

reg [31:0]count;

wire [7:0]seg\_an, seg;

single\_cycle\_MIPS CPU(

.clk\_dvid(clk),

.seg\_dvid(clk),

.reset(reset),

.display\_switch(display\_switch),

.seg\_an(seg\_an),

.seg(seg)

);

initial begin

clk = 0;

reset = 0;

display\_switch = 0;

count = 0;

end

always@(\*) begin

#1 clk = ~clk;

count <= count + 1;

end

在通过在ins\_mem.v可选择需要读入的测试文件，测试中直接读取benchmark\_ccmb.hex，其仿真结果如下所示，可明显观察出五段流水线的PC和IR变化。

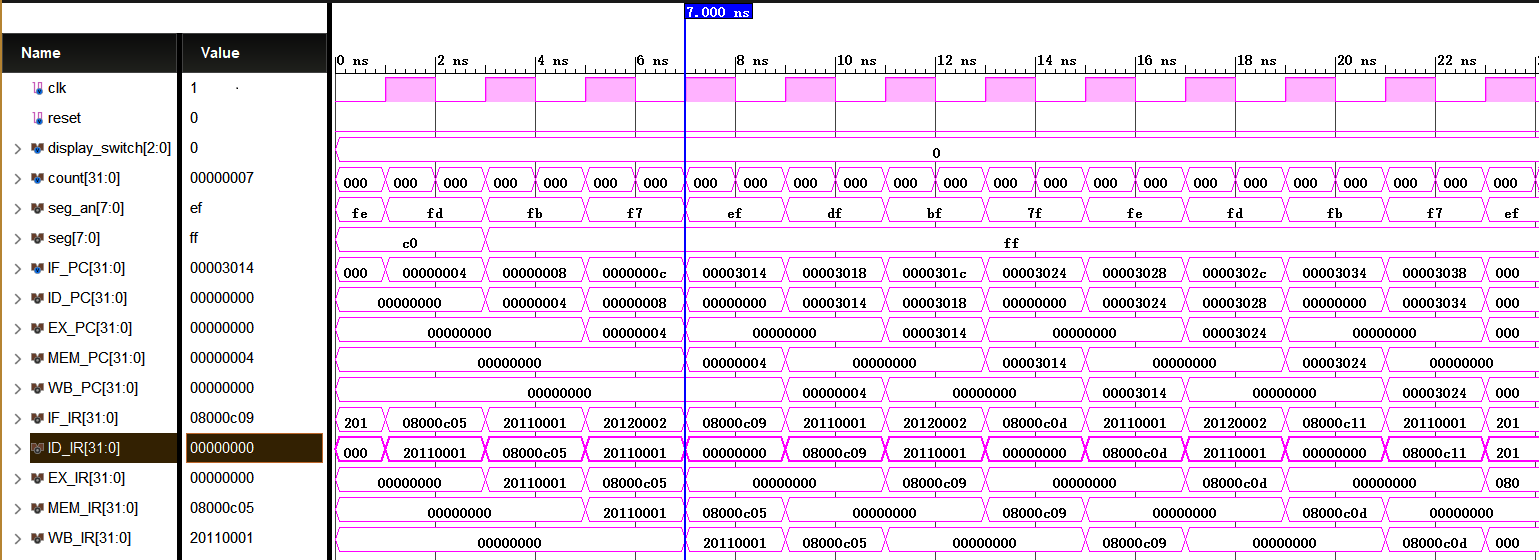


图 4.6 重定向流水线仿真图

在测试流水线上板时首先会生成比特流直接在FPGA开发板上观察现象，当观察到现象错误时可直接将ins\_mem.v中修改测试文件，之后通过仿真结果与流水线高级调试输出log相结合的方式准确定位错误位置，可更有效的修改完善程序。

## 性能分析

通过分析不同方案时钟周期数差异进行性能分析，其中仅测试benchmark程序，测试结果如下表所示：

表 4.1 性能对比表

|  | 单周期CPU | 气泡流水线 | 重定向流水线 | 重定向流水线+分支预测 |
| --- | --- | --- | --- | --- |
| 总周期数 | 1545 | 3623 | 2297 | 1847 |
| 无条件分支数 | 38 | 38 | 38 | 38 |
| 条件分支成功数 | 276 | 276 | 276 | 276 |
| 插入气泡数 |  | 1446 | 120 | 120 |
| LoadUse次数 |  |  | 120 | 120 |
| 预测跳转成功 |  |  |  | 252 |
| 预测跳转失败 |  |  |  | 27 |

分析上表可发现，气泡流水线->重定向流水线->重定向流水线+分支预测这三种设计中性能是在逐渐改进的。在气泡流水线中，虽然提高了指令的并行性，但由于存在大量的跳转指令和数据冲突，通过对比和单周期CPU的数据就可以看出该流水线的性能较低，之后在流水线中加入旁路，即实现重构定向流水线，可以明显看出插入的气泡数减少了十几倍，这表明流水线停顿的次数减少了十几倍，因此对流水线的性能有较大的提高。在重定向流水线的基础上加入分支预测后，可观察到插入的气泡数是相同的（LoadUse情况），但加入分支预测后，总周期数减少了450。

添加分支预测后统计了预测该指令跳转成功的次数为252次，预测该指令跳转但失败的次数为27次。在重定向流水线中，由于跳转指令均在EX段执行所以对于分支成功的情况都需要停顿两个周期，而在分支预测中，如果预测成功则可以省去这两个周期的停顿，如果预测失败，即预测跳转但分支失败，则需要停顿两个周期，所以有以下式子成立：

（重定向流水线+分支预测）总周期数+预测成功\*2-预测失败\*2

=1847+252\*2-27\*2=2297=重定向流水线总周期数

## 主要故障与调试

### LoadUse统计故障

重定向流水线：LoadUse数据相关错误。

**故障现象：**重定向流水线中LoadUse次数统计结果为121。

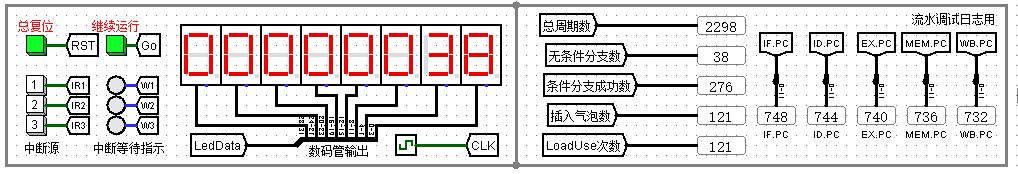


图 4.7 错误运行效果图

**原因分析：**首先排除统计方式的问题，检查统计电路，如下所示：

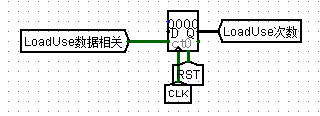


图 4.8 LoadUse计数电路

由此可确定错误是因为在某个位置多判断了一次LoadUse相关，通过流水线高级调试方式找到错误的地方，观察错误位置附近代码，指令类型为addi、lw、lw，检查自己的LoadUse产生电路，如下所示：

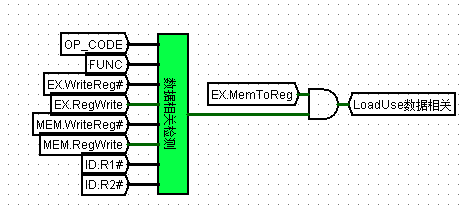


图 4.9 错误电路

其运行时的现象为LW（ID）、LW（EX）、ADDI（MEM），此时并不属于LoadUse情况，但由于ID段和MEM段存在数据相关且EX段为lw指令则判断为LoadUse数据相关，错误在于LoadUse数据相关仅需判断ID段和EX段的相关性。

**解决方案：**为使用气泡流水线中的数据相关检测子电路，只需要将MEM.WriteReg#和 MEM.RegWrite两个输入修改为常量0即可，在数据相关检测子电路中0号寄存器不进行讨论，也就意味着对ID段和MEM段的相关性不进行讨论，修改后的电路如下所示：

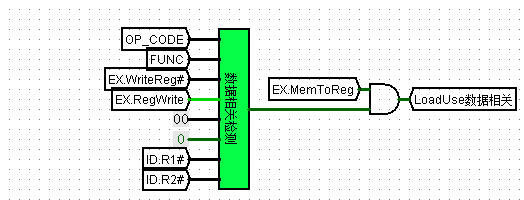


图 4.10 正确电路

### 流水线中断故障

重定向流水线+中断：中断错误，返回到程序的开头。

**故障现象：**仅测试单级中断，即不在一个中断未完成的情况下点击其他中断的产生按钮，可正确产生中断信号，但在返回时却回到了程序的开头。

**原因分析：**首先由于在单周期CPU上实现的多级中断已可以正常运行，所以CP0可以确定是正确的。单步调试并点击中断信号，可观察到正确进入中断程序，但在返回时跳转到了全0的PC地址。经过多次测试并查看数据存储器的内容发现数据存储器中写入的返回地址就是全0，考虑写入数据的错误。

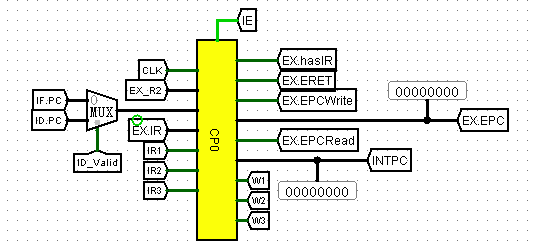


图 4.11 EX段CP0

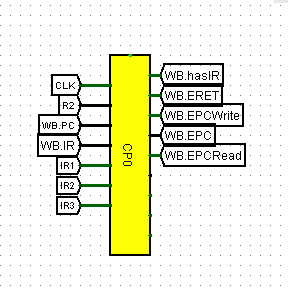


图 4.12 WB段CP0

分析这部分电路，由于对IR的译码部分在CP0中，所以想通过这个方式对WB.IR进行译码得到相关控制信号，但是由于复制电路时没有考虑太多，此时的WB.EPC是根据WB.PC、WB.R2生成的，当点击按钮产生中断信号时，此时在EX段的CP0中是正确的中断处理程序，但在WB的CP0也同样做了处理，甚至生成了错误的WB.EPC。

**解决方案：**将EX段生成的EX.EPC沿着流水线传递到WB段成为WB.EPC数据，具体如下图所示，其中左边绿色为EX/MEM，右边蓝紫色为MEM/WB。

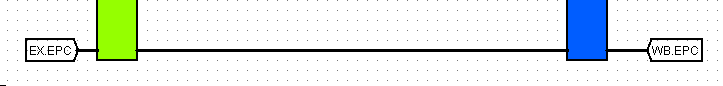
****

图 4.13 EPC的传递过程

## 实验进度

表 4.2 课程设计进度表

| 时间 | 进度 |
| --- | --- |
| 第一天 | 完成ALU、数据选择器、syscall等模块的编写和测试 |
| 第二天 | 考虑进位等原因改进了ALU，协助完成单周期上板调试 |
| 第三天 | 通过benchmark\_ccmb开发板测试,完成理想流水线,正在学习并调试气泡流水线 |
| 第四天 | 完成气泡流水线和重定向流水线，但关于周期和气泡的计数存在一定问题，待解决。 |
| 第五天 | 解决气泡流水线和重定向流水线遗留的问题，之后学习和完成动态分支预测。 |
| 第六天 | 完成分支预测，正在进行重定向流水上板 |
| 第七天 | 完成重定向流水线上板，学习中断 |
| 第八天 | 完成单周期CPU多级中断，正在调试流水中断 |

# 设计总结与心得

## 课设总结

1. 为设计支持24+4指令的单周期CPU分析指令独立完成控制信号表。
2. 独立在logisim上设计完成了单周期CPU。
3. 以小组形式分配任务承担单周期CPU部分模块的设计，并协助完成单周期CPU的上板过程，并正确添加四条扩展指令。
4. 设计并实现理想流水线。
5. 设计并实现气泡流水线。
6. 设计并实现重定向流水线。
7. 学习其他同学的中断设计部分，最终经过参考和完善完成单周期CPU的中断。
8. 设计并实现流水线中断。

## 课设心得

本次课程设计为时两周且难度较大，也是第一次在课程设计中采取分组模式，起到了一定的督促作用，同时在小组中也从同组同学那里学到了很多知识，以下是我在完成课程设计的过程中所积累的收获和心得。

1. 课程设计中巩固了组原的相关知识，也从零开始学习了流水线的很多知识。
2. 本次实验对logisim和vivado有了更好的使用经验，尤其是在logisim中学会了输出日志的调试方法，在流水线的调试过程中非常有用。
3. 在设计电路尤其流水线的时候非常需要注意标签的规范性，在流水线中通过在在信号前加上流水线段名来区分不同段的控制信号，这使得电路更加清晰明了，在调试的时候也更容易找到错误。因为可以分成每一段来看，而不需要总是以全局的方式来检查电路。
4. 学习了很多中断方面知识，最初着手写中断没有什么思路，想法过于固定觉得自然是要在硬件设计上完成所有东西，所以一直没有想明白保存现场等相关操作要如何实现。之后询问了其他同学，在同学的讲解下接受了中断大部分由软件设计完成的正确思路，并自行完成了单级中断。但在多级中断时又需要考虑很多问题，比如中断被打断的时候，如何保存两个中断的信息，并实现正确的中断顺序。
5. 设计电路时应注意通用性，在本次实验中有部分封装好的子电路是可以公用的，但由于部分细微差距我选择了添加新的子电路，增加了不少工作量。
6. 在部分可进行赋值的电路中，即只需要修改标签的时候，需要非常注意复制过来的电路是否每个输入输出都是正确的。
7. 在着手画电路或写代码之前要首先做好充分的设计，避免边设计边实现的过程中，突然发现不可避免的问题，需要通过修改之前的设计部分来实现，这种情况下很容易因无法顾全全局而导致设计错误或设计较乱。
8. 本次实验采用分组模式，也提示了我在学习时不能闭门造车，要学会主动的去问问题，才能节省很多不必要的时间，但在问问题之前也一定要自己做好充分的思考，才能在问问题的时候更精准的表达自己的问题所在，有利于更好地找到解决方法。

建议在分组的时候对成绩进行分段处理，分出比较实力平均的小组。在本次实验中我们小组的分组是非常合理的，但每天会注意天梯赛的排行榜，也注意到了一些小组在前期明显落后于其他组，猜测可能是有些班级的分组方式不太合理，不太利于整个组的合作和进度。课程虽难，但回想起来会是很有意思的一段经历，最后感谢老师们和帮助过我的同学们。

# 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:** |