***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： | U201610504 |
| 姓 名： | 刘逸帆 |
| 电 话： | 13007159315 |
| 邮 件： | [lyf460315457@qq.com](mailto:lyf460315457@qq.com) |

目 录

[1 课程设计概述 3](#_Toc4977955)

[1.1 课设目的 3](#_Toc4977956)

[1.2 设计任务 3](#_Toc4977957)

[1.3 设计要求 3](#_Toc4977958)

[1.4 技术指标 4](#_Toc4977959)

[2 总体方案设计 6](#_Toc4977960)

[2.1 单周期CPU设计 6](#_Toc4977961)

[2.2 流水CPU设计 12](#_Toc4977962)

[2.3 气泡式流水线设计 14](#_Toc4977963)

[2.4 重定向流水线设计 15](#_Toc4977964)

[2.5 动态分支预测机制设计 16](#_Toc4977965)

[2.6 中断机制设计 18](#_Toc4977966)

[3 详细设计与实现 20](#_Toc4977967)

[3.1 单周期CPU 实现 20](#_Toc4977968)

[3.2 流水CPU实现 27](#_Toc4977969)

[3.3 气泡式流水线实现 29](#_Toc4977970)

[3.4 重定向流水线实现 30](#_Toc4977971)

[3.5 动态分支预测机制实现 33](#_Toc4977972)

[3.6 中断机制实现 36](#_Toc4977973)

[4 实验过程与调试 40](#_Toc4977974)

[4.1 测试用例和功能测试 40](#_Toc4977975)

[4.2 性能分析 42](#_Toc4977976)

[4.3 主要故障与调试 43](#_Toc4977977)

[4.4 实验进度 46](#_Toc4977978)

[5 设计总结与心得 47](#_Toc4977979)

[5.1 课设总结 47](#_Toc4977980)

[5.2 课设心得 47](#_Toc4977981)

[参考文献 50](#_Toc4977982)

# 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 | STI | 小于立即数置数 |
| 20 | SLTU | 小于无符号数置数 |
| 21 | J | 无条件转移 |
| 22 | JAL | 转移并链接 |
| 23 | JR | 转移到指定寄存器 | If $v0==10 halt(停机指令)  else 数码管显示$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 | SLLV | 可变左移 | 指令格式参考MIPS32指令集，最终功能以MARS模拟器为准。 |
| 29 | XOR | 异或 |
| 30 | SB | 存字节 |
| 31 | BLTZ | 小于等于跳转 |

# 总体方案设计

## 单周期CPU设计

本次采用的设计方案是硬布线控制、且主控存分开的方案，即采用硬布线方式实现用于主控CPU的控制器（Control Unit）设计。同时在实施的过程中，先利用logisim平台对主要功能部件进行设计，再设计数据通路将各个部件联系起来，并在logisim平台上使用benchmark程序对电路进行测试。Logisim测试完成后，最后使用verilog语言进行FPGA编程，从而在开发板上模拟CPU进行验证。

支持24条指令的单周期CPU总体结构图如图 2.1所示。



图 2.1 单周期CPU总体结构图

### 主要功能部件设计

单周期CPU主要由以下几个功能部件组成：程序计数器PC、指令存储器IM、寄存器堆RegFile、运算器ALU、数据存储器DM、控制器CU。

#### 程序计数器PC

程序计数器PC用于存储下一条指令在指令存储器中的地址。PC的输入端口有：下一条指令在IM中的地址、控制PC寄存器是否跳转到输入的使能信号、时钟信号和复位信号；输出端口有：取指阶段的指令地址。

对于logisim平台，使用平台内置的普通寄存器即可。

对于verilog语言，使用同步清零的方式仿照logisim平台中的寄存器器件进行设计即可。

#### 指令存储器IM

指令存储器IM用于存储二进制形式的MIPS指令。IM的输入端口有：指令的存储地址；输出端口有：二进制MIPS指令。

对于logisim平台，使用平台内置的ROM器件即可，对ROM器件右键即可进行ROM的初始化从而载入程序。

对于verilog语言，使用同步清零的方式仿照logisim平台中的ROM器件设计一个32位寄存器组即可，用readmemh指令实现初始化。

#### 寄存器堆RegFile

寄存器堆RegFile是多个寄存器组成的阵列，具有专门的读端口与写端口，可以多路并发访问不同的寄存器。RegFile的输入端口有：R1、R2和写入寄存器的地址、写入寄存器的数据、写使能信号、时钟信号；输出端口有：R1、R2寄存器的值。

对于logisim平台，使用资源包CS3410中封装好的Register File文件即可。

对于verilog语言，使用同步清零的方式仿照logisim平台中的器件设计一个32位寄存器组即可，需支持并发访问与多路输出。

#### 运算器ALU

运算器用于完成操作数的四则运算以及左移、右移、比较等运算，其输入与输出端的引脚功能如表2.1所示。其中X与Y通过控制器控制的多路选择器从指令还是寄存器文件输入，shmat用于控制移位指令的移位位数，ALU\_OP控制信号则直接通过controller的输出得到。其输出端口Result用于后续的PC跳转、寄存器文件写回等操作。此外，ALU还提供了Result2，OF，UOF，Equal等输出，其中除了用Equal进行跳转判断之外其它的输出在本CPU的实现中均未使用到。

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 |
| shmat | 输入 | 5 | 移位指令的移位位数shmat |
| ALU\_OP | 输入 | 4 | 运算器功能码，具体功能见表2.2 |
| Result | 输出 | 32 | ALU运算结果 |
| Result2 | 输出 | 32 | ALU结果第二部分，用于乘法指令结果高位或除法指令的余数位，其他操作为零 |
| 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 |

ALU\_OP的输入值与对应的功能如表 2.2 所示。

表 2.2 ALU\_OP 输入及其对应的功能

|  |  |  |
| --- | --- | --- |
| ALU OP | 十进制 | 运算功能 |
| 0000 | 0 | Result = X << shmat 逻辑左移 （Y 取低五位） |
| 0001 | 1 | Result = X >>> shmat 算术右移 （Y 取低五位） |
| 0010 | 2 | Result = X >> shmat 逻辑右移 （Y 取低五位） |
| 0011 | 3 | Result = (X \* Y)[31:0]; Result2 = (X \* Y)[63:32] 有符号 |
| 0100 | 4 | Result = X/Y; Result2 = X%Y 无符号 |
| 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 |

对于logisim平台，使用实验资源中已经封装好的ALU器件即可。

对于verilog语言，通过行为描述中的case语句对不同的输入进行运算处理即可。

#### 数据存储器DM

数据存储器用于支持CPU中数据的读写，需要使用RAM。其输入端口有：数据地址addr、写入数据使能信号str、写入数据内容data、数据输出使能信号load、片选信号sel、时钟信号clk和复位信号rst；输出端口有：输出数据data。

对于logisim平台，为支持SB指令的字节存储，不应使用平台自带的RAM器件，转而使用资源包CS3410中的MIPS RAM器件即可。

对于verilog平台，则需要创建一个32位寄存器组即可，同时需要能够通过sel片选信号选择存储时的数据长度，复位采用同步清零。

#### 控制器CU

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信号的作用说明如表 2.3。

表 2.3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，控制数据存储器的写入 |
| ALUOP | 运算器操作控制符(4位) | R型指令根据Funct选择 |
| MemToReg | 寄存器写入数据来自存储器 | lw指令，选择写回数据是存储器还是ALU的输出 |
| RegDst | 写入寄存器编号rt/rd选择 | R型指令，选择写回寄存器是rt还是rd |
| AluSrcB | 运算器B输入选择 | lw指令、sw指令、立即数运算类指令，选择ALU的输入B是寄存器还是指令的值 |
| SignedExt | 立即数符号拓展 | ADDI、ADDIU、SLTI指令，控制有无符号拓展 |
| JR | 寄存器跳转指令译码信号 | JR指令，选择无条件分支地址 |
| JAL | JAL指令译码信号 | JAL指令，选择无条件分支地址 |
| 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 |
| SLLV | SLLV指令译码信号 | SLLV指令，控制ALU的shmat输入 |
| SB | SB指令译码信号 | SB指令，控制DM的片选信号 |
| BLTZ | BLTZ指令译码信号 | BLTZ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 2.4所示。

表 2.4主控制器控制信号框架

| 指令 | OpCode | FUNCT | ALUOP | MemToReg | MemWrite | ALU\_SRC | RegWrite | SYSCALL | SignedExt | RegDst | BEQ | BNE | JR | JMP | JAL | SLLV | SB | BLTZ |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SLL | 0 | 0 | 0 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| SRA | 0 | 3 | 1 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| SRL | 0 | 2 | 2 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| ADD | 0 | 32 | 5 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| ADDU | 0 | 33 | 5 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| SUB | 0 | 34 | 6 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| AND | 0 | 36 | 7 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| OR | 0 | 37 | 8 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| NOR | 0 | 39 | 10 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| SLT | 0 | 42 | 11 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| SLTU | 0 | 43 | 12 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| JR | 0 | 8 | X |  |  |  |  |  |  | 1 |  |  | 1 | 1 |  |  |  |  |
| SYSCALL | 0 | 12 | X |  |  |  |  | 1 |  | 1 |  |  |  |  |  |  |  |  |
| J | 2 | X | X |  |  |  |  |  |  |  |  |  |  | 1 |  |  |  |  |
| JAL | 3 | X | X |  |  |  | 1 |  |  |  |  |  |  | 1 | 1 |  |  |  |
| BEQ | 4 | X | X |  |  |  |  |  |  |  | 1 |  |  |  |  |  |  |  |
| BNE | 5 | X | X |  |  |  |  |  |  |  |  | 1 |  |  |  |  |  |  |
| ADDI | 8 | X | 5 |  |  | 1 | 1 |  | 1 |  |  |  |  |  |  |  |  |  |
| ANDI | 12 | X | 7 |  |  | 1 | 1 |  |  |  |  |  |  |  |  |  |  |  |
| ADDIU | 9 | X | 5 |  |  | 1 | 1 |  | 1 |  |  |  |  |  |  |  |  |  |
| SLTI | 10 | X | 11 |  |  | 1 | 1 |  | 1 |  |  |  |  |  |  |  |  |  |
| ORI | 13 | X | 8 |  |  | 1 | 1 |  |  |  |  |  |  |  |  |  |  |  |
| LW | 35 | X | 5 | 1 |  | 1 | 1 |  | 1 |  |  |  |  |  |  |  |  |  |
| SW | 43 | X | 5 |  | 1 | 1 |  |  | 1 |  |  |  |  |  |  |  |  |  |
| SLLV | 0 | 38 | 0 |  |  |  | 1 |  |  | 1 |  |  |  |  |  | 1 |  |  |
| XOR | 0 | X | 9 |  |  |  | 1 |  |  | 1 |  |  |  |  |  |  |  |  |
| SB | 40 | X | 5 |  | 1 | 1 |  |  | 1 |  |  |  |  |  |  |  | 1 |  |
| BLTZ | 1 | 0 | 11 |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 1 |

表中首列未指令的名称，随后两列分别为指令对应的Opcode字段与Funct字段，对于非R型指令，其Funct字段对应设置为X。第四列为ALU的操作码，用于控制ALU的行为，对于不需要使用ALU的值指令，对应字段设置为X。此外，方便起见，表中未填写部分均为0。

### 数据通路的设计

单周期CPU中所有指令的数据通路如表2.5所示，Logisim中数据通路的搭建将围绕此表进行。

表 2.5指令系统数据通路框架

| 指令 | PC | IM | RF | | | | ALU | | | DM | | Tube |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R1# | R2# | W# | Din | A | B | OP | Addr | Din |
| addu | -- | PC | IM | IM | IM | ALU | R1 | R2 | 0101 | -- | -- | -- |
| addi | -- | PC | IM | -- | IM | ALU | R1 | IM | 0101 | -- | -- | -- |
| addiu | -- | PC | IM | -- | IM | ALU | R1 | IM | 0101 | -- | -- | -- |
| addu | -- | PC | IM | IM | IM | ALU | R1 | R2 | 0101 | -- | -- | -- |
| and | -- | PC | IM | IM | IM | ALU | R1 | R2 | 0111 | -- | -- | -- |
| andi | -- | PC | IM | -- | IM | ALU | R1 | IM | 0111 | -- | -- | -- |
| sll | -- | PC | IM | IM | IM | ALU | R2 | IM | 0000 | -- | -- | -- |
| sra | -- | PC | IM | IM | IM | ALU | R2 | IM | 0001 | -- | -- | -- |
| srl | -- | PC | IM | IM | IM | ALU | R2 | IM | 0010 | -- | -- | -- |
| sub | -- | PC | IM | IM | IM | ALU | R1 | R2 | 0110 | -- | -- | -- |
| or | -- | PC | IM | IM | IM | ALU | R1 | R2 | 1000 | -- | -- | -- |
| ori | -- | PC | IM | -- | IM | ALU | R1 | IM | 1000 | -- | -- | -- |
| nor | -- | PC | IM | IM | IM | ALU | R1 | R2 | 1010 | -- | -- | -- |
| lw | -- | PC | IM | -- | IM | DM | R1 | IM | 0101 | ALU | -- | -- |
| sw | -- | PC | IM | IM | IM | -- | R1 | IM | 0101 | ALU | R2 | -- |
| beq | RF | PC | IM | IM | -- | ALU | R1 | R2 | -- | -- | -- | -- |
| bne | RF | PC | IM | IM | -- | ALU | R1 | R2 | -- | -- | -- | -- |
| slt | -- | PC | IM | IM | IM | ALU | R1 | R2 | 1011 | -- | -- | -- |
| slti | -- | PC | IM | -- | IM | ALU | R1 | IM | 1011 | -- | -- | -- |
| sltu | -- | PC | IM | IM | IM | ALU | R1 | R2 | 1100 | -- | -- | -- |
| j | IM | PC | -- | -- | -- | -- | -- | -- | -- | -- | -- | -- |
| jal | IM | PC | -- | -- | 常数 | ALU | PC | 常数 | 0101 | -- | -- | -- |
| jr | RF | PC | IM | -- | IM | -- | -- | -- | -- | -- | -- | -- |
| sys | -- | PC | 常数 | 常数 | IM | -- | -- | -- | -- | -- | -- | R1,R2 |

## 流水CPU设计

### 总体设计

由于不需要考虑数据冲突和有无条件跳转指令的问题，理想流水线 CPU 的设计较为简单。虽然理想流水线支持的指令较少，但这一设计能够作为气泡流水线以及重定向流水线的基石，十分重要。流水线CPU相比单周期CPU，主要新增了流水接口这一部件，需要注意的是在设计实现流水接口时，要考虑到后续气泡流水线和重定向流水线对流水接口的要求，预留好对应的位置。另外在修改数据通路时，由于不同段可能包含同一控制信号，需要注意控制信号应从流水线的哪一段取出，以免出错。

在设计了理想流水线后，为了扩充其支持的指令集，在EX段添加对无条件跳转指令和有条件跳转指令的处理电路，从而使流水线支持分支指令。

### 流水接口部件设计

每一个流水接口应该包含以下几个端口：时钟输入、使能端以、清零端、数据输入端以及与数据输入端一一对应的数据输出端。对于的流水接口，根据需求的不同其数据输入端以及数据输出端的数量和位数也不同，从而需要使用不同的寄存器，但对于所有的寄存器都应该使用同步清零的方式设置清零端，否则将导致清零功能的异常。

流水接口应该完成如下功能：使能为高电平时，在每一个时钟周期从所有的数据输入端读入数据并将其缓存在寄存器中，数据输出端的值与寄存器保持一致；使能为低电平则不进行操作。若清零端为高电平则进行同步清零，同步清零在logisim上需要借助多路选择器与接地来实现。

不同的流水接口需要缓存的数据不同。在本CPU的设计中，instruction、PC这两个数据需要在每一段流水线中都传递有对应的值以供调试；对于其余的控制信号、R1、R2寄存器数据等数据，将其向下一段流水线传递直至数据被使用且不再被后续流水线需要，便可以停止传递，即不再需要对应的流水接口。

### 理想流水线设计

在完成了对于流水接口的设计后，只需要考虑每一条指令的数据通路并将对应的信号连接到对应的模块端口即可。需要注意在取得控制信号时，要由对应的流水段取出，如不能将ID段的RegWrite信号与WB段的数据Data同时向RegFile输入，否则会形成错误的逻辑。此外，虽然在流水线CPU中一些信号可以在不同的流水段进行处理，但对于syscall而言，为保证程序的完整执行，其需要在WB 段处理。

### 分支相关流水线设计

相比于理想流水线，分支相关流水线需要在EX段判断以下信息：当前指令是否是跳转指令、跳转指令的条件是否满足、根据跳转指令的类型计算出目标地址，随后通过清零端清空前两段流水接口中的无效指令，并将计算出的目标地址送回IF段中的PC寄存器即可在流水线中实现分支指令的支持。

## 气泡式流水线设计

### 总体设计

流水线CPU中存在包括数据冲突在内的冒险现象，基于理想流水线扩充的分支相关流水线并不能实现对数据冲突的处理，故而需要实现气泡流水线从而处理对应冒险现象。具体来说，为实现在数据冲突发生时令需要使用数据的ID段等待EX或MEM段的数据写回，需要添加数据相关检测模块用于产生DataCrash信号，在DataCrash信号产生时，将IF/ID流水接口使能置为0并向EX段中插入气泡。

与此同时，数据相关检测模块产生的DataCrash信号同样应该参与对PC寄存器使能的控制，在数据冲突发生时，需要保证PC寄存器中的地址不变，从而实现保存IF、ID段的状态直至冲突消失。

### 数据相关检测模块设计

数据相关检测模块接收ID段的指令Op、Func字段和R1、R2寄存器的地址，同时模块接受EX、MEM段的指令Op、Func字段和写回寄存器地址，从而数据相关检测模块能够判断三段流水中的指令类型，进而能够判断是否ID段需要使用的寄存器尚未被写回，从而判断是否需要产生气泡。

若ID段与EX或MEM段发生了数据冲突，则数据相关检测模块在 EX 段产生一个气泡并控制PC使能和IF/ID使能为0。直至数据在WB段被写回寄存器，数据相关消失，流水线继续工作。

### 气泡流水线设计

根据2.3.2中对数据相关检测模块的设计，将其添加进分支相关流水线中并用其产生的DataCrash信号对流水线电路进行相应的控制，完成上述工作后流水接口就应该能够处理所有的指令，并能够正确运行benchmark程序了。气泡的插入将会导致流水线的性能下降，为了检验气泡流水线是否正确，可通过以下公式计算流水线的周期数，从而与预期的周期数进行比较：

气泡流水线周期数 = 指令条数 + 流水充满时间 − 1 + 分支跳转指令 × 预取深度(2) + 数据相关气泡数

## 重定向流水线设计

### 总体设计

为了解决气泡流水线插入气泡过多导致流水线性能下降的问题，修改气泡流水线的逻辑，在发生数据冲突时不再进行插入气泡的操作，而是在下一个时钟周期将尚未写回至寄存器堆的数据直接重定向至需要数据的EX段中。

就具体分析而言，需要重定向的数据有四种可能性，即EX/MEM中的AluResult，MEM/WB 中的 AluResult 以及 MemResult（写回 RegFile 中的数据）均有可能需要被重定向到 EX 段，因而需要对气泡流水线中的数据相关检测模块进行功能扩充，将其改造为Forward模块，从而能够指示需要将哪一个数据重定向到ALU的输入A或输入B。

需要注意的是重定向流水线会引入LoadUse问题，即在一条load指令之后若紧随有一条需要使用该寄存器的指令，不能直接将数据存储器DM的输出重定向到EX段中，因为这样会使整个CPU的关键路径包含访存，从而关键路径变长影响CPU整体的工作效率。为解决此问题，需要在Forward模块中添加对LoadUse情况进行判断的逻辑，在检测到该情况发生时，向EX段添加一个气泡，便能消除LoadUse现象。

### Forward模块设计

由于在重定向流水线中对数据相关检测模块的功能需求进行了扩展，需要将其改造为Forward模块。具体而言，在Forward器件中对R1冲突和R2冲突分别设计判断电路，同时对每个寄存器的冲突而言，依据与EX发生冲突的流水段和信号不同，对冲突结果进行编码，从而产生两位的RD1st和RD2st信号用于控制ALU两个输入端的信号分别源自何处。以RD1st为例，若在ID段发现与EX段存在数据冲突则RD1st==1，ALU输入端A的输入数据由EX/MEM流水接口中的Aluout重定向而来；若在ID段发现与MEM段存在数据冲突且MemToReg==0则RD1st==2，ALU输入端A的输入数据由MEM/WB流水接口中的Aluout重定向而来，否则若MemToReg==1则ED1st==3，ALU输入端A的输入数据由MEM/WB流水接口中的Memout重定向而来；其余情况没有产生数据冲突，直接将ID/EX流水接口中的寄存器取值作为输入即可。

此外，若在Forward模块中检测到ID与EX段出现数据冲突且EX段中指令种类为Load，则生成LoadUse信号。

### 重定向流水线设计

使用2.4.2中设计的Forward模块替换气泡流水线中的数据相关检测模块，Forward模块输出的RD1st和RD2st信号用于在EX端中对ALU的两个数据输入端进行片选，选择对应的原始数据或重定向数据进行输入；输出的LoadUse信号参与控制IF/ID端流水接口的使能与PC寄存器的使能，在LoadUse信号产生时，暂停IF、ID端的行为并向EX端中插入一个气泡，解除LoadUse信号后重定向流水线即可继续工作。

为验证此时的重定向流水线是否正确，有周期公式：

流水线运行周期数 = 指令条数 + (流水充满时间 - 1) + (分支跳转次数) \* 预取深度 + Load-Use气泡数

## 动态分支预测机制设计

### 总体设计

气泡流水线与重定向流水线均无法解决在跳转指令生效前由于预取深度产生的周期浪费，为尝试利用起因为预取浪费掉的时钟周期，通过分支预测的方式尝试对跳转指令之后的指令进行预测，从而达到尝试预取正确指令的目的。

为实现对跳转指令下一条指令的方式，使用类似构造cache的方式实现BHT这一硬件，并用其对分支指令进行预测。

### BHT设计

动态分支预测的核心便是BHT这一硬件，BHT中存储了分支跳转的历史，并通过跳转历史来判断未来可能的跳转结果，常用的构建BHT的方式有两位状态机，即一个双预测位的cache，其工作流程如图2.2所示，已经验证采用这种结构的BHT能够对分支进行较为高命中率的预测。本次实验中方便起见，没有使用双预测位，仅使用一般的八行cache与最基本的LRU淘汰策略进行BHT的构建。

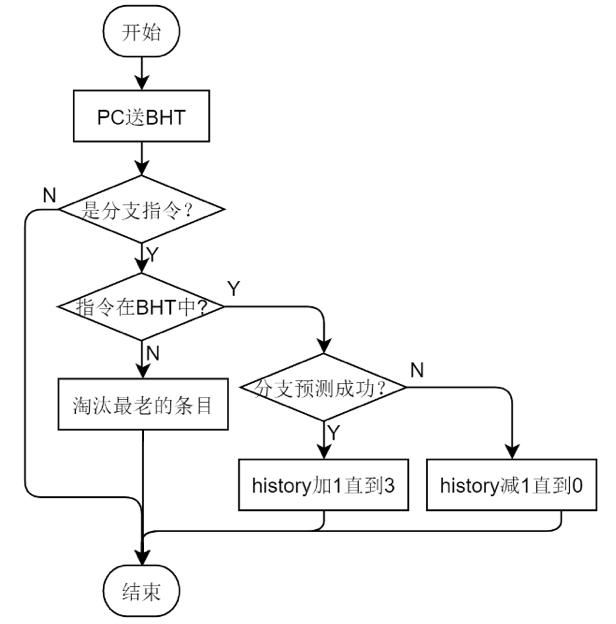


图2.2 BHT器件工作流程

在实现BHT的过程中，尤其需要注意对标志位的清零端需要使用同步清零，否则会因为清零不同步造成LRU算法无法正常工作。除此之外在对淘汰标志位进行自增操作时，需要考虑计数器位数有限，在计数到最大值时应停止继续计数。对于BHT的读操作和写操作两个功能建议分立，从而能在设计时思路更加清晰简单。

### 动态分支预测流水线设计

向重定向流水线中添加BHT器件从而新增动态分支预测功能，添加动态分支预测后在IF端识别到分支指令时，流水线将会出现三种情况需要处理：没有使用BHT进行分支预测，即未命中，此时流水线的表现与未添加分支预测时的流水线一致；使用了BHT中的BTB表但预测失败，即命中了但预测错误，此时需要在EX段中对预测结果进行判断，因为预测失败所以向IF、ID段插入气泡并向PC寄存器汇总传入正确的目标地址，与未添加分支预测的流水线表现一致；使用了BHT中的BTB表且预测成功，此时流水线能够顺利向下推进而无需插入气泡，避免了由于预取深度造成的周期浪费。

为检验此时的流水线是否正常工作，有下述周期公式：

分支预测总周期数 + (预测成功次数 – 预测失败次数) \* 2 = 重定向总周期数

## 中断机制设计

### 总体设计

中断分为外部中断和内部中断，本次实验中只考虑可屏蔽的外部中断。对于外部中断机制的设计，需要考虑的又有硬件实现的中断支持和软件实现的中断处理。

首先考虑如何进入单级中断，由于执行中断程序完毕后需要能够返回进入中断的位置并恢复整个CPU的状态，这就需要在进入中断时首先保存CPU的状态，即需将PC的值爆存在EPC中，并在退出中断时将EPC的值返回给PC寄存器。而在实现多级中断时，由于EPC寄存器不能够存储多次进入中断所产生的多个PC的值，便需要将PC的值进行压栈，在退出中断时依次出栈即可正常返回原来的代码处继续执行程序。在本实验中，可直接使用logisim中内置RAM器件模拟用于存储PC的堆栈。

在进入多级中断时，除了实现存储PC的堆栈外，还需要对不同等级的中断进行处理。由于多级中断存在优先级的问题，需要由硬件进行判断新的中断能否打断旧的中断。

在退出多级中断时，则需要判断继续执行一条被打断的低级中断还是开始执行一个已经在排队的低级中断。

除硬件之外，还需要设计对应的软件，即中断处理程序，并对每个中断处理程序计算好对应的程序地址，从而在触发中断信号时能够正确跳转。

### 硬件设计

如总体设计中所述，在实现多级中断时，需要在进入中断时将PC的值进行压栈，退出中断时依次出栈即可正常返回原来的代码处继续执行程序。就本实验中的具体设计而言，直接使用logisim中内置RAM器件模拟用于存储PC的堆栈：在每一次进入新中断时将地址自增并向RAM中存入进入中断前PC的值；在每一个中断执行结束时将RAM中指向的栈顶PC值存入寄存器并将RAM地址自减，从而指向前一个存入的PC值。

对于进入多级中断时需要考虑优先级的需求，在硬件中首先使用先行进位器筛选已发出中断请求的等级最高的中断，随后将此中断的等级与前一个中断的等级(若无，则为0)进行比较，若新中断等级较高则产生中断开始信号IRStart，此时将旧PC压栈并切换至中断处理程序处继续执行。若新中断等级较低，则记录中断请求将其排队。

对于退出多级中断时是恢复中断还是开始中断的判断，设计硬件缓存三种不同等级中断的执行情况：若在退出中断时，低级中断存在且尚未执行，则触发中断开始信号IRStart；否则若低级中断之前被打断，则恢复PC后直接继续执行即可。

### 软件设计

中断的软件设计需要考虑中断向量表的设计以及中断程序本身的设计。为了简便起见，将指令寄存器IM的地址由10位拓展为12为，高二位用作片选信号，选择四块不同的ROM作为指令来源，指令存储器包含的四块ROM分别用于存储benchmark程序、一级中断处理程序、二级中断处理程序和三级中断处理程序。经过这样的设计，每个中断处理程序的入口地址便显而易见。

对于中断程序的具体设计而言，主要需要考虑保存寄存器的值。通过编写程序保护现场，即进入中断时将寄存器压入堆栈，并在中断返回前从堆栈中弹出寄存器的值来保护所有在中断处理程序中需要用到的寄存器。由于PC的值已经通过硬件实现的堆栈进行保存，在程序中无需考虑PC的压栈。

# 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. Logisim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，输入为经过多路选择器片选的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。DIS为显示信号、Go为继续执行信号、Syscall为停机信号，仅当未按下Go按钮、程序触发停机信号Syscall且不是用于输出数据时，组合电路将控制PC的使能为低电平从而使整个电路停机。如图 3.1所示。

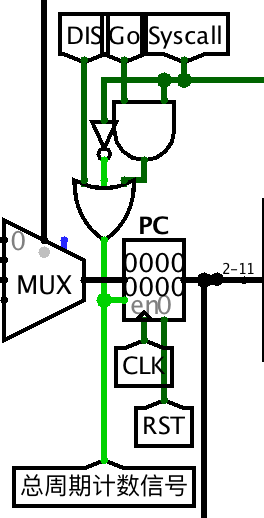


图 3.1程序计数器（PC）

1. FPGA实现：

程序计数器PC的核心Verilog代码如下：

always @(posedge clk or posedge rst) begin

if(rst) data<=32'h00000000; //reset key

else begin

if(ena) data<=data\_in; //enable ,input

end

end

1. 指令存储器（IM）
2. Logisim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。

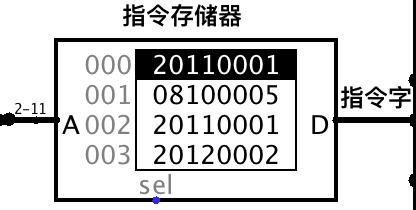


图 3.2指令存储器（IM）

1. FPGA实现：

使用verilog语言设计一个大小为Nmem的32位寄存器组，通过readmemh指令对设计的ROM进行初始化。指令存储器IM的核心Verilog代码如下：

reg [31:0] memory [0:Nmem-1]; // 32-bit memory with Nmem entries

initial begin

$readmemh(ROM\_DATA, memory, 0, Nmem-1);

end

assign data = memory[addr[11:2]][31:0];

1. 寄存器堆（RegFile）
2. Logisim实现：

直接使用资源包CS3410 Components中的寄存器堆并对外浠进行简单封装即可。如图3.3所示。

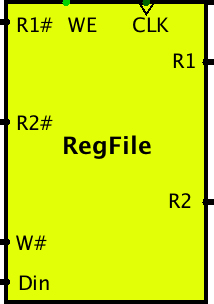


图 3.3寄存器堆（RegFile）

1. FPGA实现：

设计一个大小为NUM的SIZE位寄存器组REG\_FILE，清零方式使用同步清零，输出由输入的寄存器地址决定。寄存器堆RegFile的核心Verilog代码如下：

reg [SIZE-1:0] REG\_FILE[0:NUM-1];

integer i;

always @(posedge CLK or posedge clr)

begin

if(clr)

for(i=0;i<NUM;i=i+1) REG\_FILE[i]<=0;

else

if(WE) REG\_FILE[W\_ADDR]<=Din;

end

assign R1=REG\_FILE[R1\_ADDR];

assign R2=REG\_FILE[R2\_ADDR];

1. 运算器（ALU）
2. Logisim实现：

直接使用实验材料中的MIPS ALU即可。

1. FPGA实现：

由同组成员负责设计，主要使用了verilog中的行为描述与case语句，在输入的操作码不同时直接为结果寄存器赋不同的计算结果即可。

1. 数据存储器（DM）
2. Logisim实现：

直接使用资源包CS3410 Components中的MIPS RAM即可，其提供的片选端口可用于实现SB指令需要的字节存储。如图3.4所示。



图 3.4数据存储器（DM）

1. FPGA实现：

设计一个大小为50的32位寄存器组state，清零方式使用同步清零，输出由输入的寄存器地址和片选信号共同决定。数据存储器DM的核心Verilog代码如下：

// ram here

reg [31:0] state [0:49];

always@(posedge clk) begin

// reset

if(rst) begin

for (j=0; j<50; j=j+1) begin

state[j] <= 32'b0; //reset array

end

end

// Store data

if(str & (~rst)) begin

if(sel[3]!=0) state[addr][31:24]<=data\_in[31:24];

if(sel[2]!=0) state[addr][23:16]<=data\_in[23:16];

if(sel[1]!=0) state[addr][15:8]<=data\_in[15:8];

if(sel[0]!=0) state[addr][7:0]<=data\_in[7:0];

end

end

// Load data

assign data\_out=state[addr];

1. 控制器（ControlUnit）
2. Logisim实现：

单周期CPU的硬布线控制器由两部分子电路组成，分别是运算器控制电路与控制信号生成电路，其中运算器控制电路根据2.1.1节中的表2.4所示真值表进行实现，由输入指令的OpCode和Func字段生成ALU运算器的控制信号ALUOP；控制信号生成电路根据2.1.1节中的表2.4所示真值表进行实现，由输入指令的OpCode和Func字段生成所有控制信号。在Logisim平台上封装后的单周期控制器如图3.5所示。

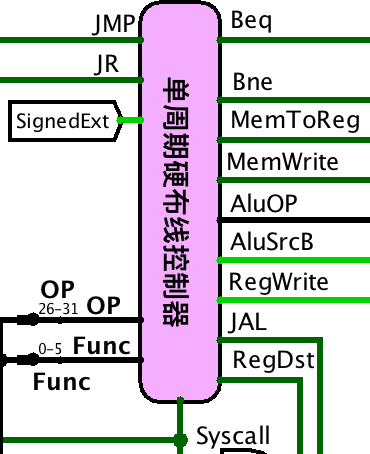


图 3.5数据存储器（DM）

1. FPGA实现：

对网上能够找到的MIPS单周期控制器源码进行精简，控制器的verilog程序中包含两个头文件instruction\_def.v和ctrl\_encode\_def.v，两个文件中分别对指令种类和控制信号进行了编码，在控制器代码Ctrl\_1.v中通过行为描述进行case语句的嵌套即可清晰的实现单周期CPU的硬布线控制器。Ctrl\_1.v的部分代码如下所示：

`INSTR\_J\_OP: begin

Mem2R = 0;

MemW = 0;

Alusrc = 0;

RegW = 0;

syscall=0;

SignedExt = 1;

RegDst = 0;

Beq=0;

Bne=0;

JR=0;

JMP = 1;

JAL=0;

SLLV=0;

LB=0;

BLTZ=0;

End

使用vivado软件对项目进行综合，所产生的控制器RTL图如图3.6所示。

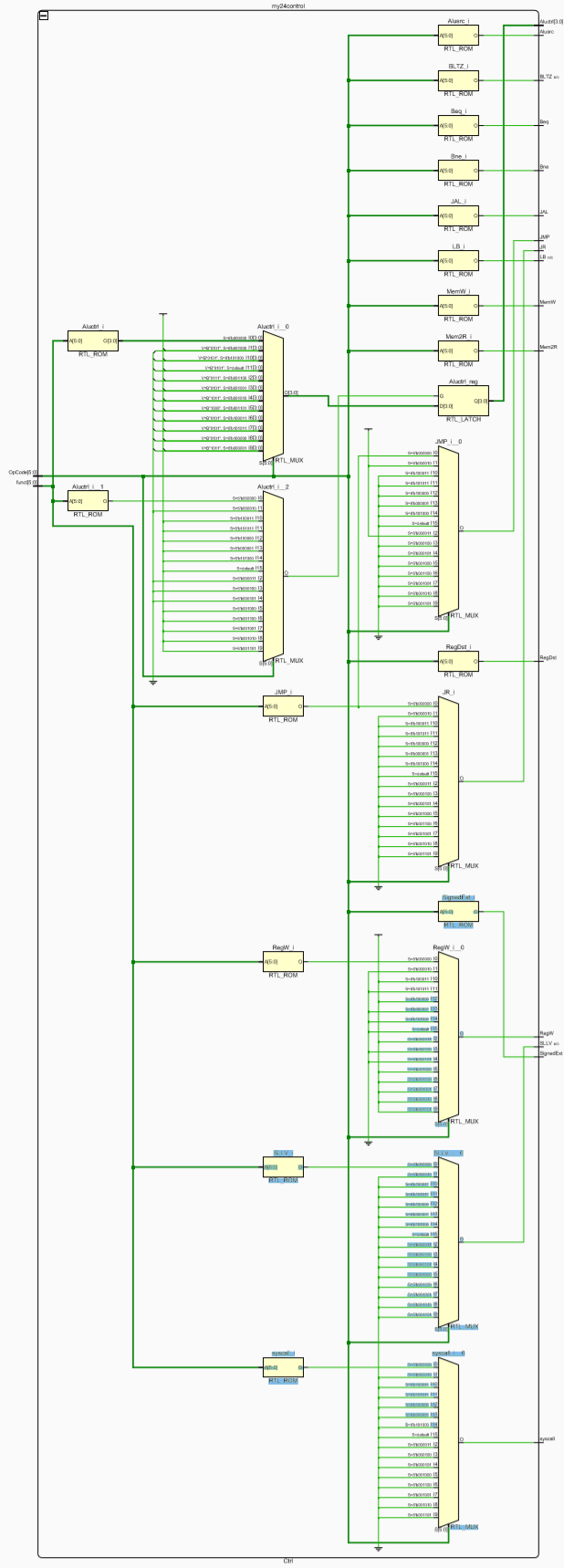


图 3.6 单周期CPU控制器（FPGA）

### 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。

1. Logism实现：

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，并如总体方案设计中2.1.2节表2.5所示列出数据通路表。根据数据通路表2.5对数据通路进行合并，再将各个主要功能部件按表进行连接后即可得到最终结果。在设计数据通路时，对于所有的多输入部件均使用多路选择器进行输入选择，数据通路便搭建完成。

在logisim中搭建数据通路图如图3.7所示

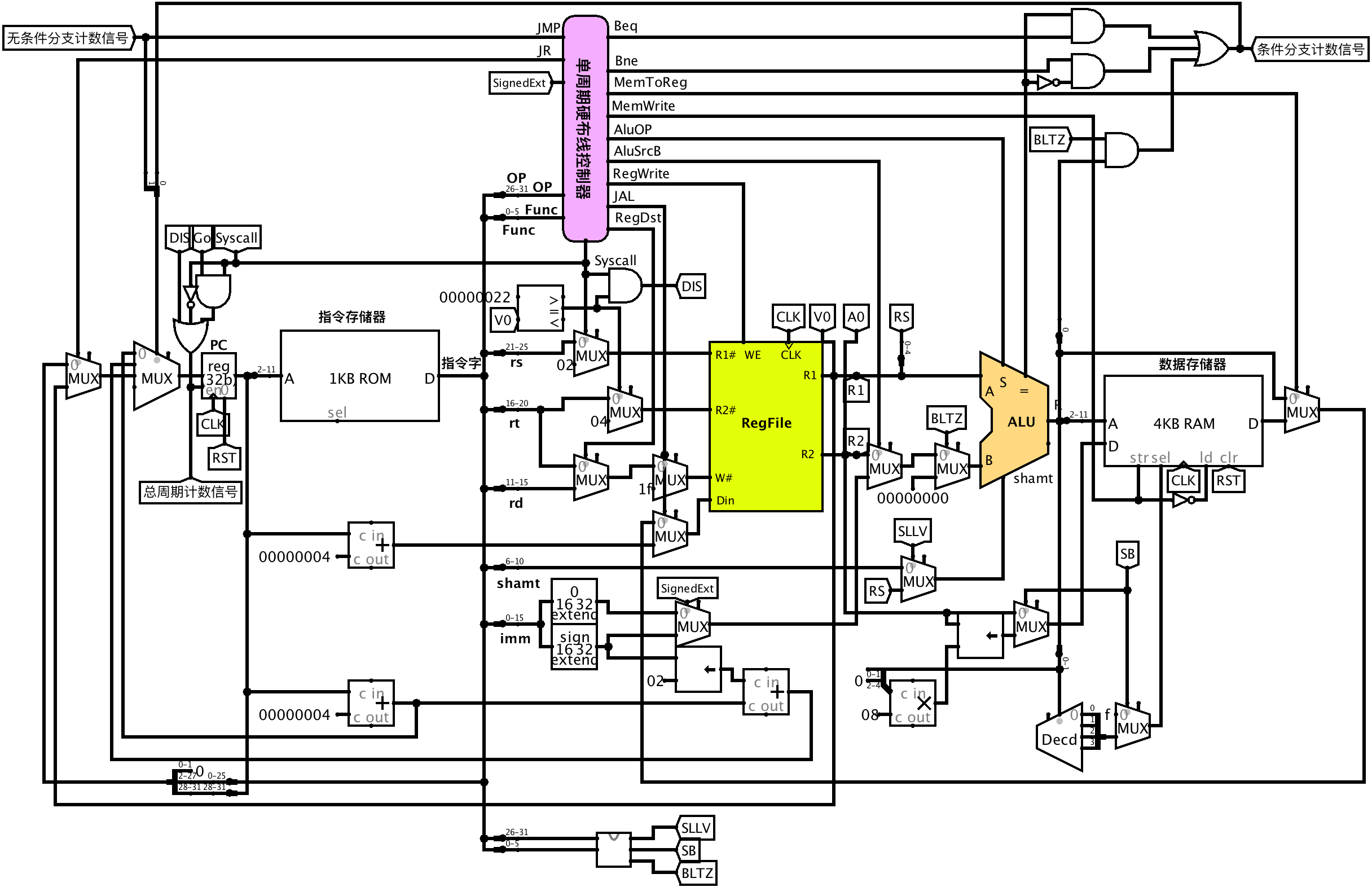


图 3.7 单周期CPU数据通路（Logism）

1. FPGA实现：

在Vivado中，使用Verilog语言根据logisim上的数据通路原理图进行电路的搭建。搭建完成后通过Vivado进行综合，数据通路的RTL图如图 3.8所示。

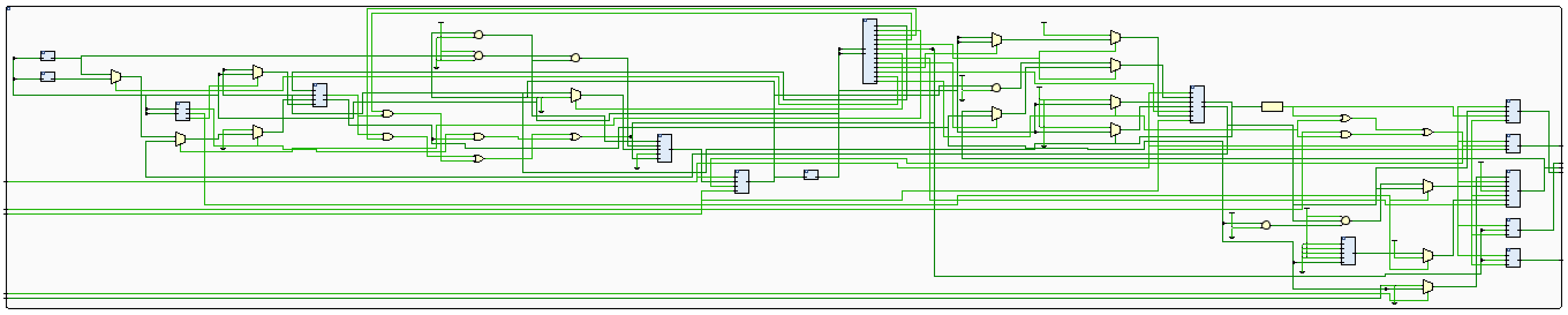


图 3.8单周期CPU数据通路（FPGA）

## 流水CPU实现

### 流水接口部件实现

1. Logism实现：

依据2.2.2中对流水接口部件的设计，在logisim上设计相应的由寄存器组构成的流水接口部件，同时应具备时钟、清零和使能端三个输入端，清零端应作为同步清零使用。以IF/ID流水接口为例，其内部结构如图3.9所示。

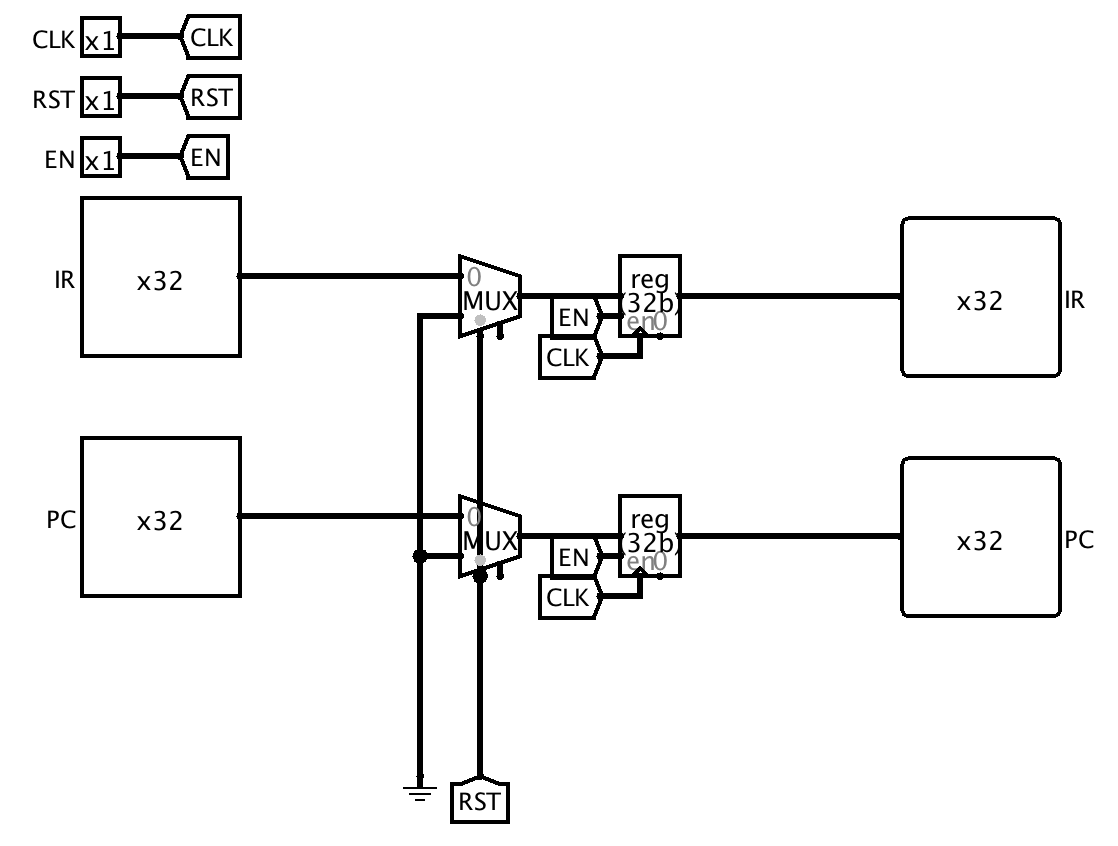


图 3.9流水接口部件IF/ID（Logisim）

1. FPGA实现：

仿照logisim电路在Vivado中使用verilog语言进行实现即可，以IF/ID流水接口为例，verilog核心代码下所示：

always @(posedge clk)begin

if(rst) begin

IR\_id = 32'b0;

PC\_id = 32'b0;

end

else if(en) begin

IR\_id = IR\_if;

PC\_id = PC\_if;

end

end

### 理想流水线实现

1. Logism实现：

根据2.2.3中理想流水线的设计，向单周期CPU中新增流水接口，并对单周期数据通路修改连接使其成为流水线数据通路。重新连接数据通路后的理想流水线Logisim电路图如图3.10所示。

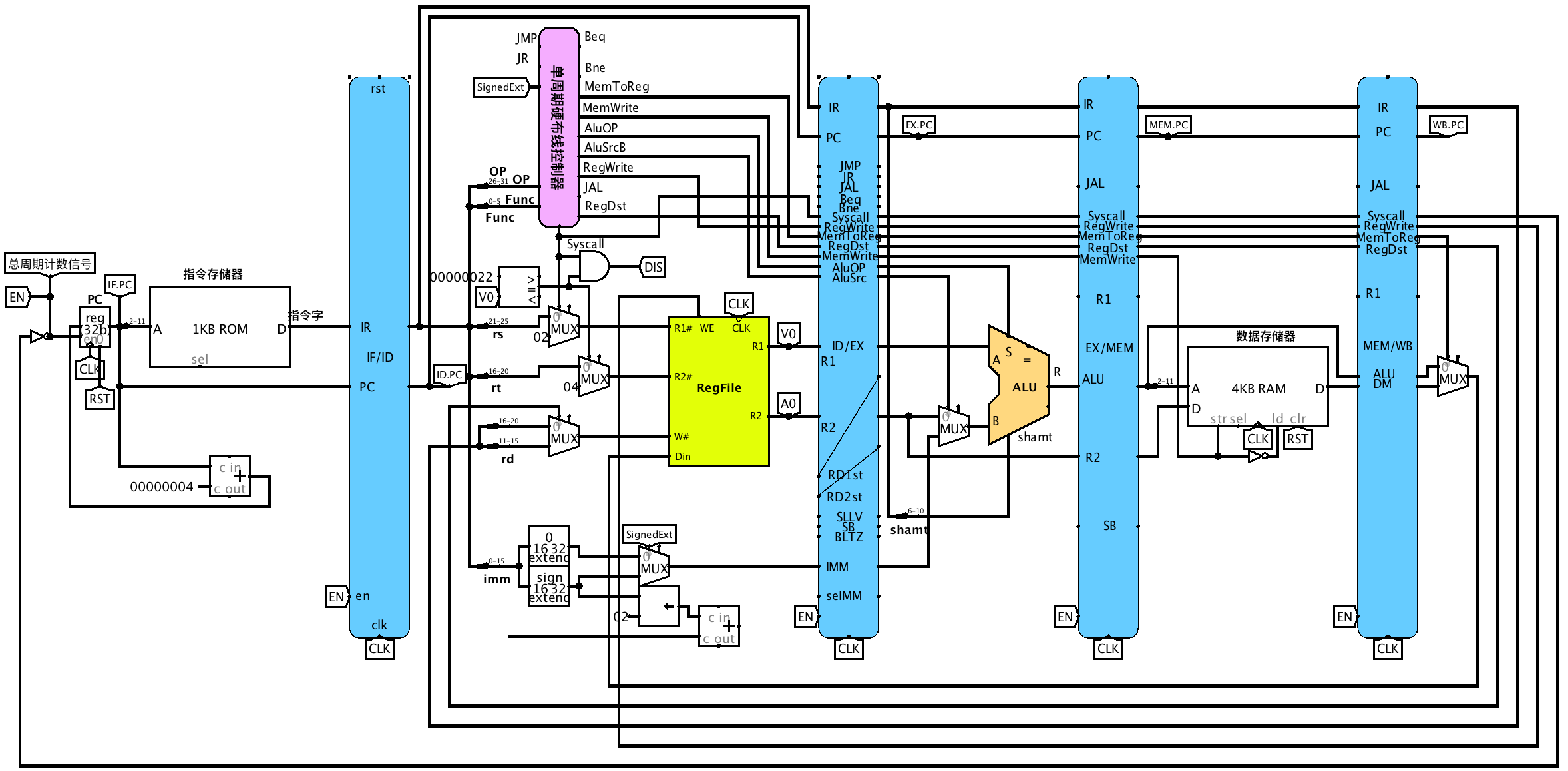


图 3.10理想流水线CPU数据通路（Logisim）

1. FPGA实现：

由于直接使用verilog编写了重定向流水线，不再叙述理想流水线的实现。

### 分支相关流水线实现

1. Logisim实现：

根据总体方案设计2.2.4节中对分支相关流水线的设计，在理想流水线的设计上稍微修改数据通路以新增对分支指令的支持即可构建分支相关流水线。修改后的Logisim电路图如图3.11所示。

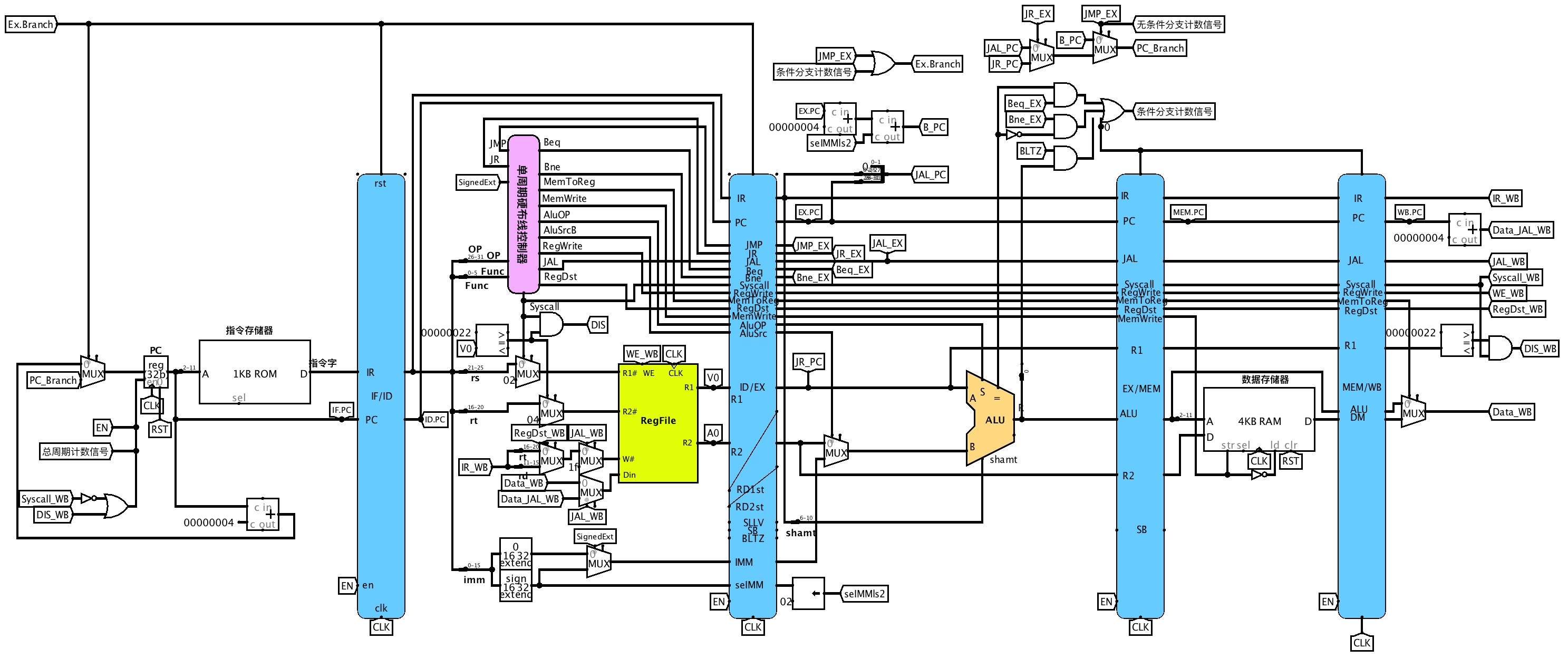


图 3.11分支相关流水线CPU数据通路（Logisim）

1. FPGA实现：

由于直接使用verilog编写了重定向流水线，不再叙述分支相关流水线的实现。

## 气泡式流水线实现

### 数据相关检测模块实现

根据2.3.2节中对数据相关检测模块的设计，模块的内部逻辑如图3.12所示。

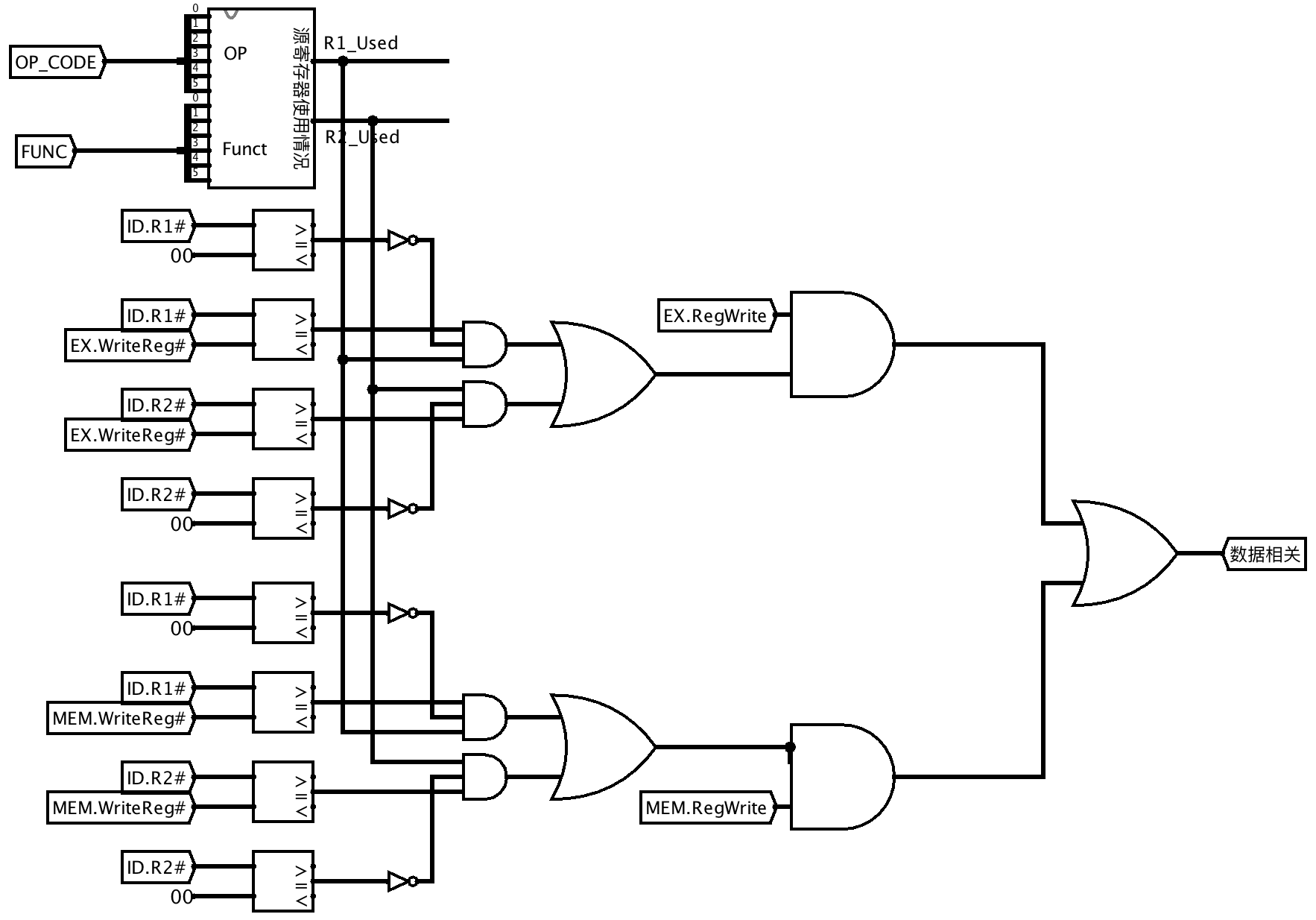


图 3.12数据相关检测模块（Logisim）

数据相关检测模块中源寄存器使用情况检测模块用于依据ID流水段的指令判断R1和R2寄存器是否将被使用，再分别判断EX段和MEM段中的写回寄存器是否与R1、R2发生冲突，从而产生数据相关信号。此外，需要忽略零号寄存器的冲突情况。

### 气泡流水线实现

根据总体方案设计2.3.3节中对气泡流水线的设计，将数据相关检测模块应用至分支相关流水线中并使用其输出的数据相关信号对数据通路进行修改即可。气泡流水线对应的Logisim原理图如图3.13所示。

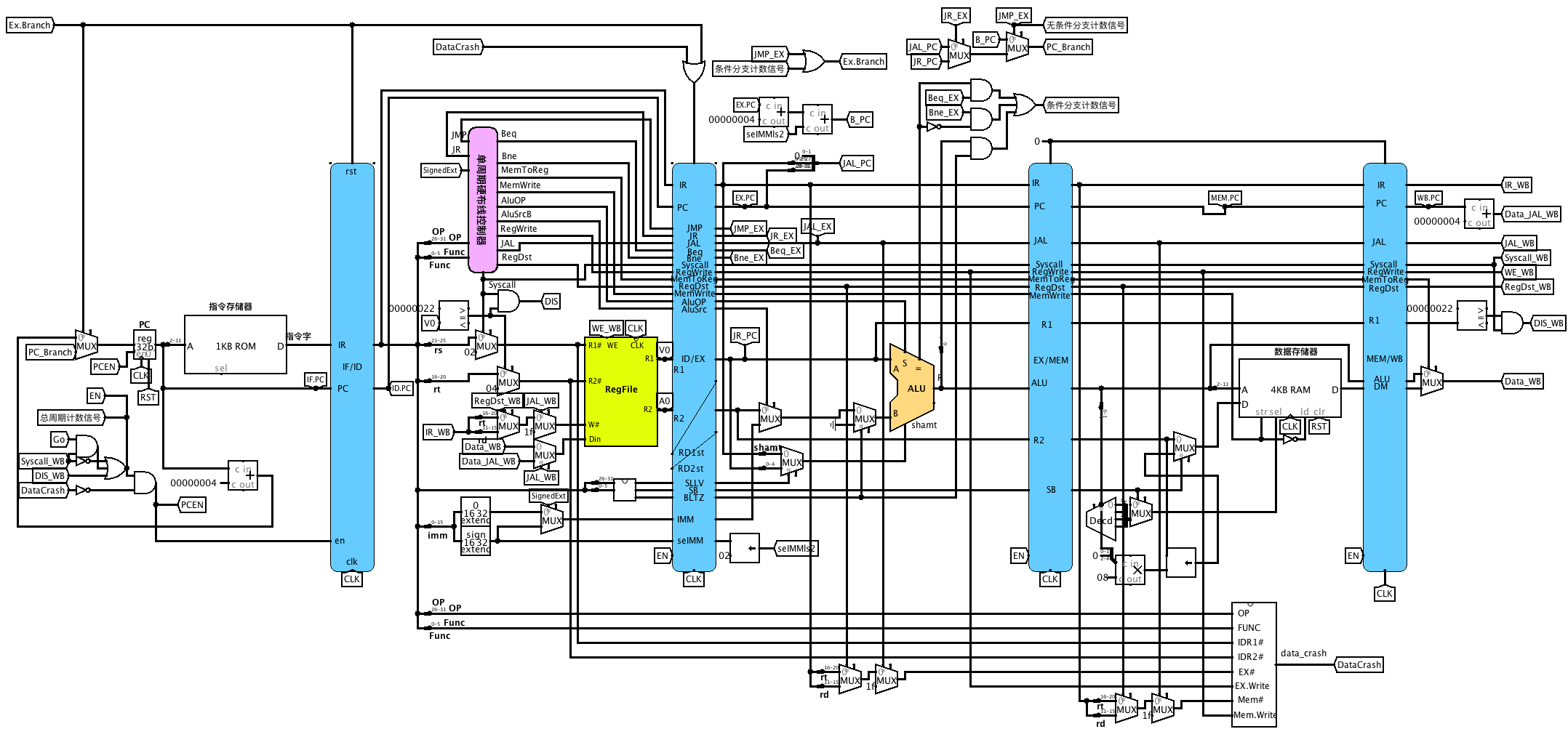


图 3.13气泡流水线CPU数据通路（Logisim）

## 重定向流水线实现

### Forward模块实现

1. Logisim实现：

根据总体方案设计2.4.2节中对Forward模块的设计，在数据相关检测模块设计的基础上考虑新增对冲突发生位置的检测并编码即可。Forward模块的Logisim电路图如图3.14所示。

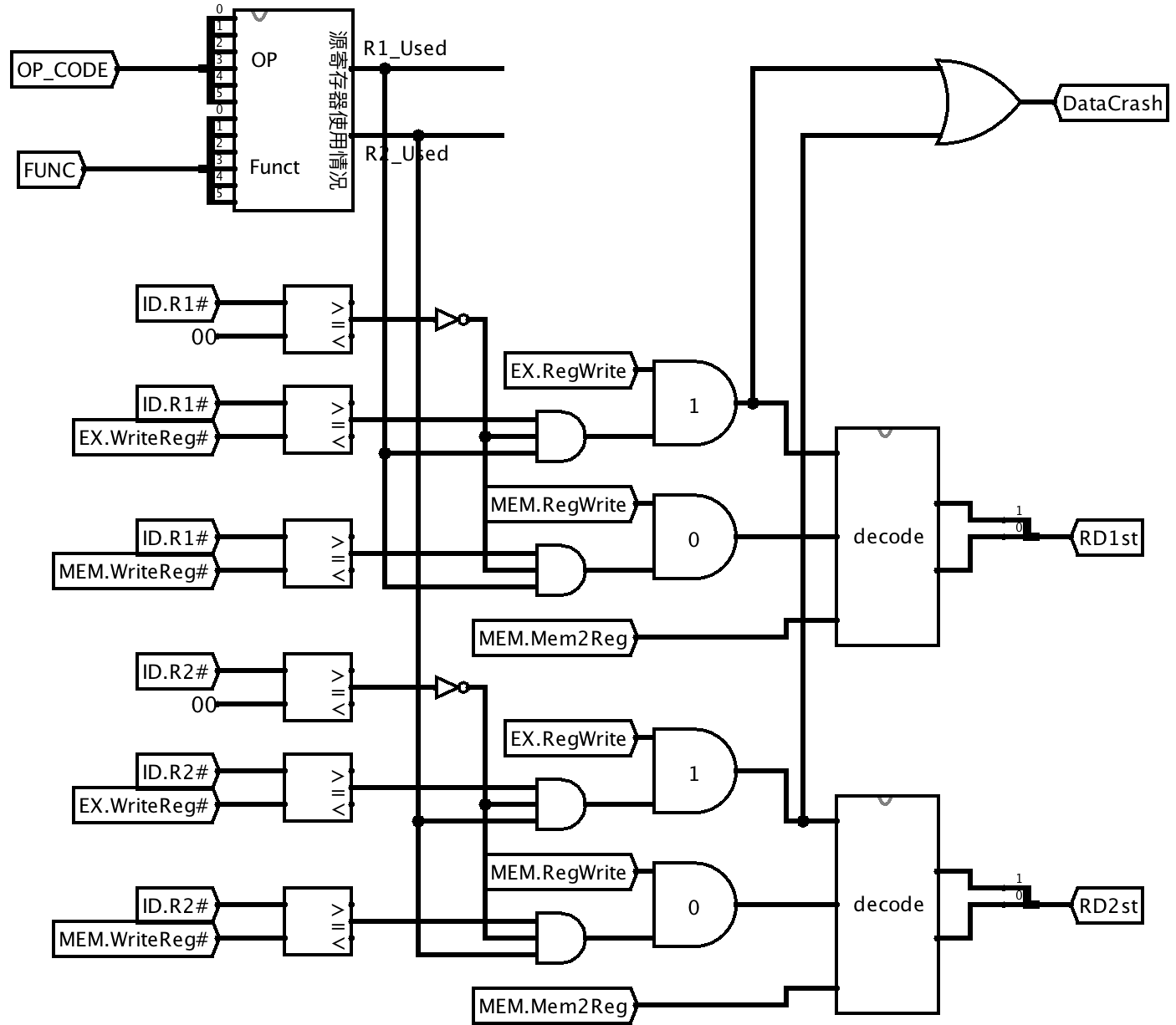


图 3.14 Forward模块（Logisim）

1. FPGA实现：

根据logisim中的电路图对Forward器件进行设计，verilog核心代码如下：

assign ex\_crash\_1 = exRegWrite & r1used & (r1addr==exaddr) & (r1addr!=0);

assign mem\_crash\_1 = memRegWrite & r1used & (r1addr==memaddr) & (r1addr!=0);

assign ex\_crash\_2 = exRegWrite & r2used & (r2addr==exaddr) & (r2addr!=0);

assign mem\_crash\_2 = memRegWrite & r2used & (r2addr==memaddr) & (r2addr!=0);

forward\_decode myfdecode1(

.ex\_crash(ex\_crash\_1),

.mem\_crash(mem\_crash\_1),

.memMemToReg(memMemToReg),

.RDst(RD1st)

);

forward\_decode myfdecode2(

.ex\_crash(ex\_crash\_2),

.mem\_crash(mem\_crash\_2),

.memMemToReg(memMemToReg),

.RDst(RD2st)

);

assign datacrash = ex\_crash\_1 | ex\_crash\_2;

Vivado下Forward器件的RTL图如图3.15所示。

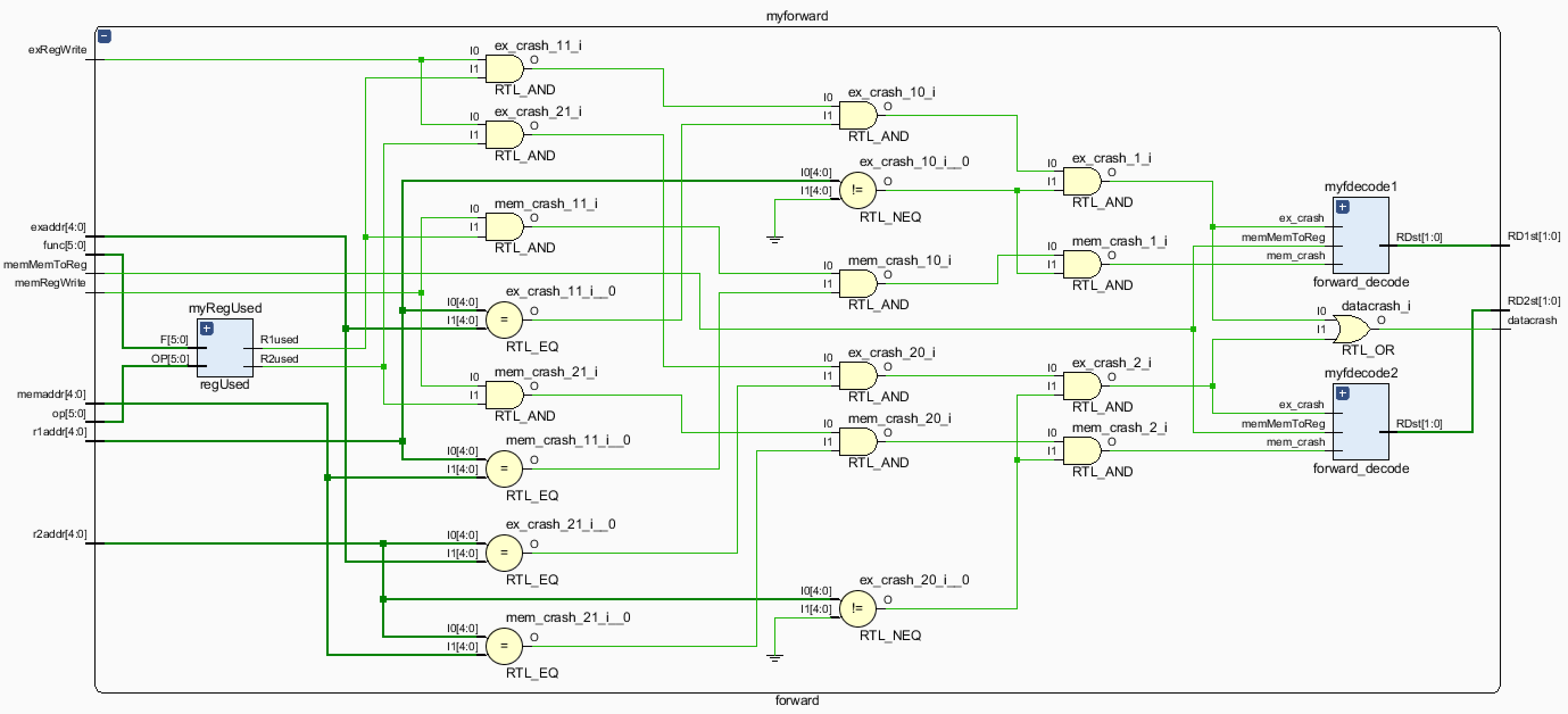


图 3.15 Forward器件（FPGA）

### 重定向流水线实现

1. Logisim实现：

根据总体方案设计2.4.3节中的具体设计，需要使用Forward器件替换气泡流水线中的数据相关检测模块，使用其输出RD1st、RD2st对数据通路进行修改以使ALU的输入端支持数据重定向，再使用其输出DataCrash生成LoadUse信号处理LoadUse情况即可构建重定向流水线CPU。重定向流水线的logisim电路图如图3.16所示。

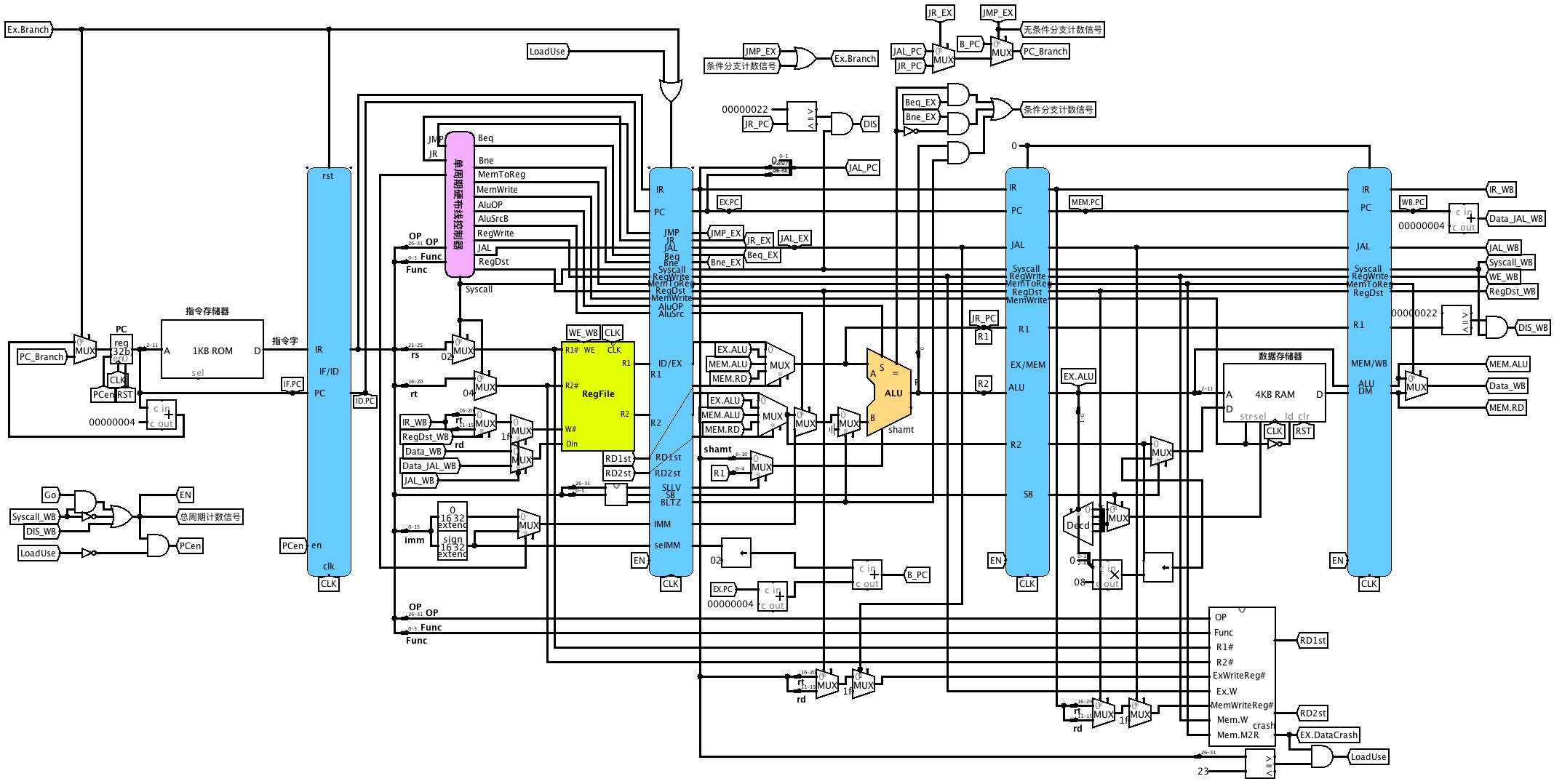


图 3.16重定向流水线CPU数据通路（Logisim）

1. FPGA实现：

在vivado内将四个流水线接口与Forward模块新增至单周期CPU的工程文件中，并依据logisim伤的重定向流水线数据通路图，通过verilog语言重新搭建重定向流水线CPU的数据通路。设计完成后基于FPGA实现的重定向流水线数据通路RTL图如图3.17所示。

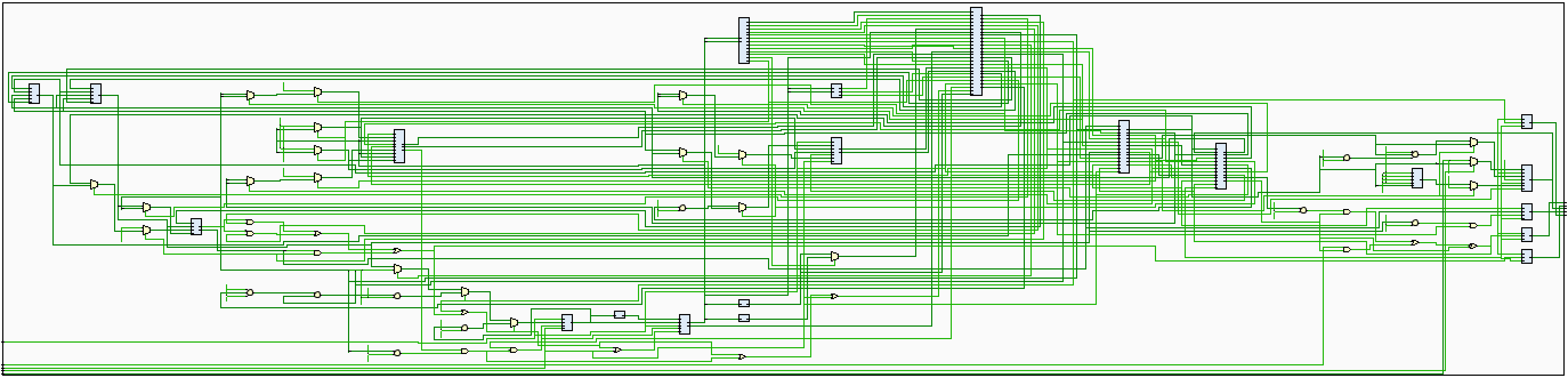


图 3.17重定向流水线CPU数据通路（FPGA）

## 动态分支预测机制实现

### BHT实现

根据总体方案设计2.5.2节中的具体设计，通过构建八行cache的方式构建BHT器件，BHT的Logisim电路图的核心部分如图3.18所示。



图 3.18 BHT的核心结构（Logisim）

其中淘汰标志位的输入NC0～NC7是由如图3.19所示的子电路计算得到，目的是防止有限位数的标志位计数溢出。

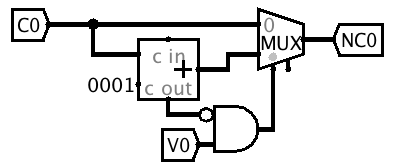


图 3.19 计算标志位子电路（Logisim）

用于控制BHT输出的Rd信号和控制标志位清零的Hit信号由如图3.20所示的子电路产生，其中使用了LRU淘汰算法以提高BHT器件的命中率。

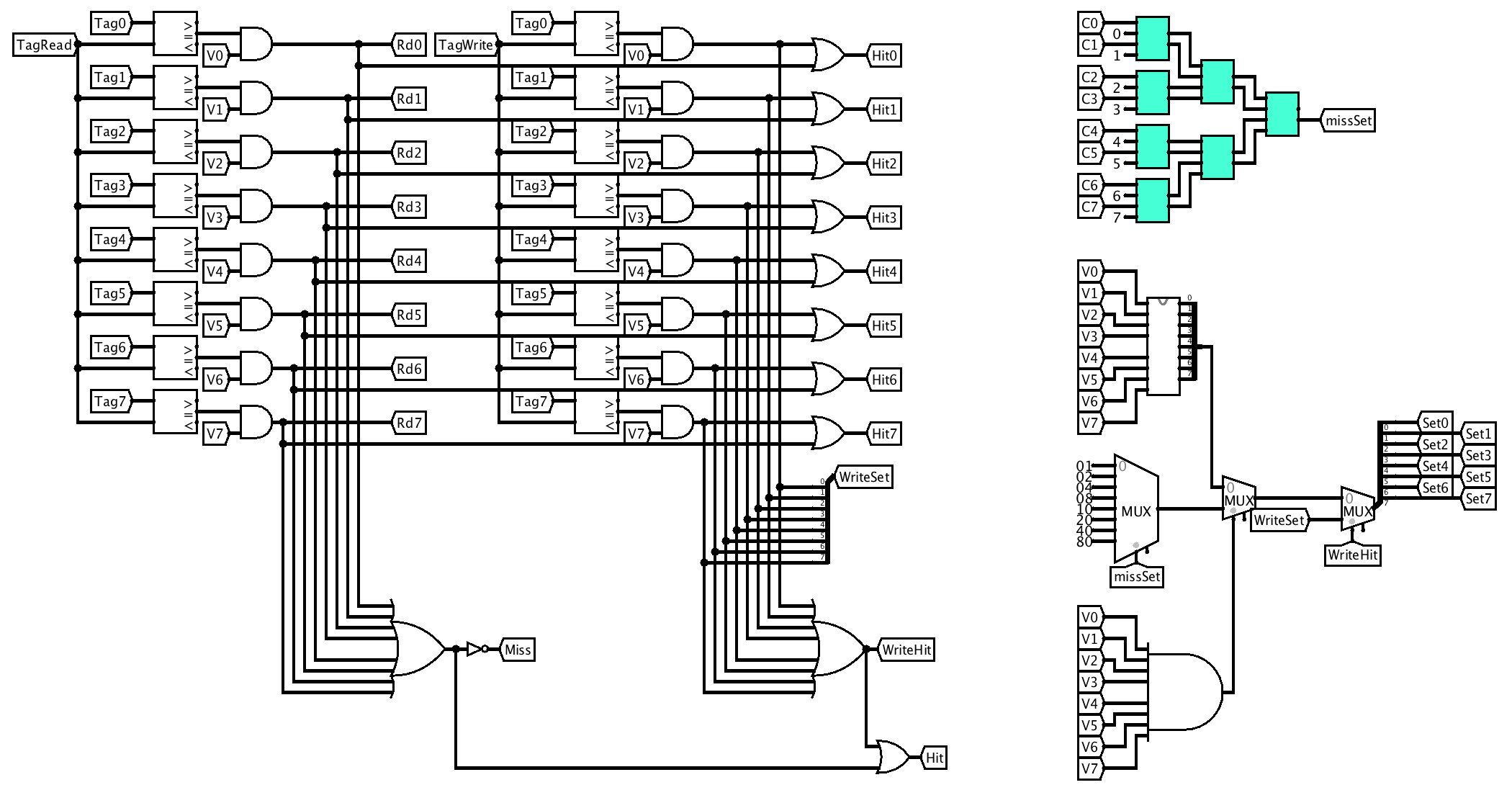


图 3.20 命中信号产生电路（Logisim）

### 动态分支预测流水线实现

根据总体设计方案2.5.3节中的具体设计，将BHT器件应用至重定向流水线中的IF段用于控制PC寄存器的输入即可，同时依据三种预测情况分别对IF/ID、ID/EX两个流水接口的清零端和使能端设置相应的信号以使CPU能够正常执行程序。动态分支预测流水线的Logisim电路如图3.21所示。

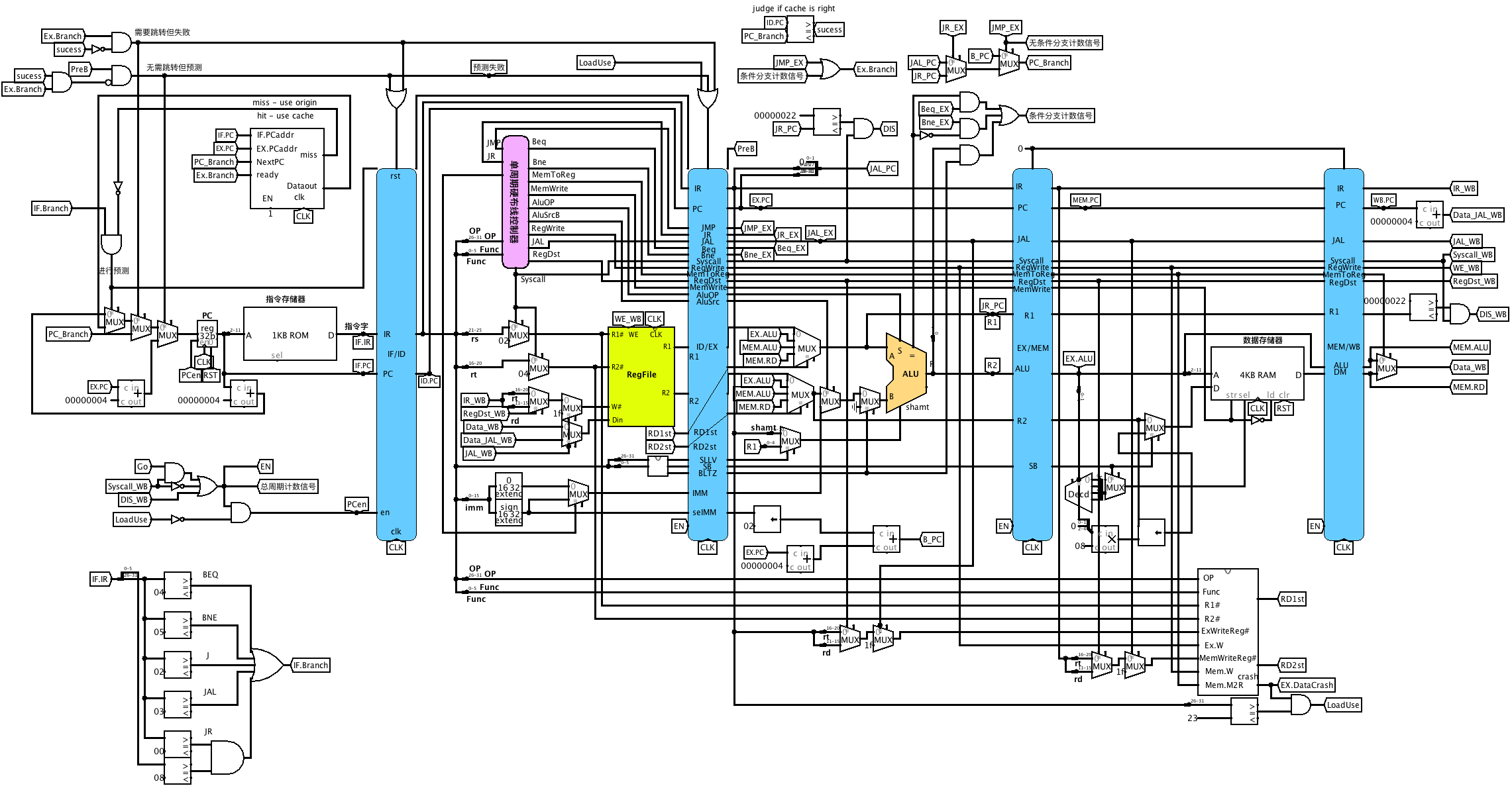


图 3.21 动态分支预测流水线电路（Logisim）

## 中断机制实现

### 硬件实现

根据总体设计方案2.6.2节中的具体设计，需要通过硬件实现以下功能：在按下每一级中断请求按钮后，硬件需要对中断请求进行记录，直至该级中断被处理完成后消除对该中断请求的记录；在每一次进入多级中断时将当前PC的值向PC栈中压栈，以便中断处理程序执行完成后能够使PC寄存器恢复到进入中断前的状态；在每一级中断开始执行后执行结束前，硬件需要对中断执行情况进行记录以确保中断处理程序的正确跳转；根据硬件维护的所有信号，在中断请求到来和中断处理程序执行结束触发ERET指令时，对PC寄存器进行正确的赋值或还原，从而保证多级中断的正确运行；对于数据通路中的指令存储器IM需要进行拓展，以存放中断处理程序。

硬件模拟记录中断请求的电路如图3.22所示。在电路中，通过两个D触发器将三级中断请求信号IR1、IR2、IR3进行存储，在每一级中断请求信号被对应的复位信号重置前，对应的指示灯W1、W2、W3输出会常亮。对于可能同时出现的三级中断信号，使用优先编码器Pri选择优先响应优先级较高的中断请求。

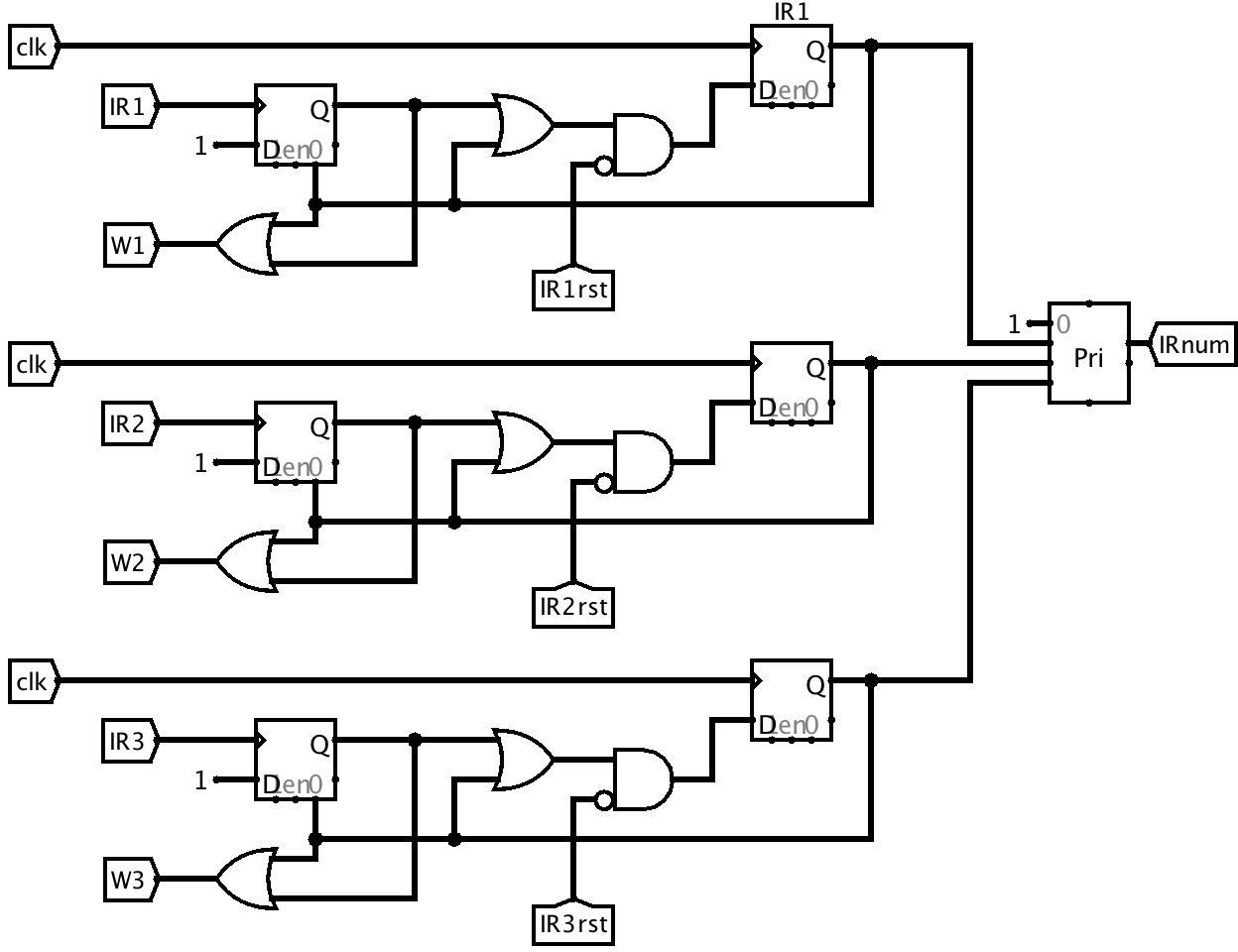


图 3.22 存储中断请求信号（Logisim）

硬件模拟储存PC的堆栈如图3.23所示。在电路中，当产生IRstart信号进入新的中断时，计数器对应的RAM地址自增并将旧的PC值存入RAM中；当产生ERET信号中断退出时，PC值被取出至寄存器中且计数器对应的RAM地址自减。

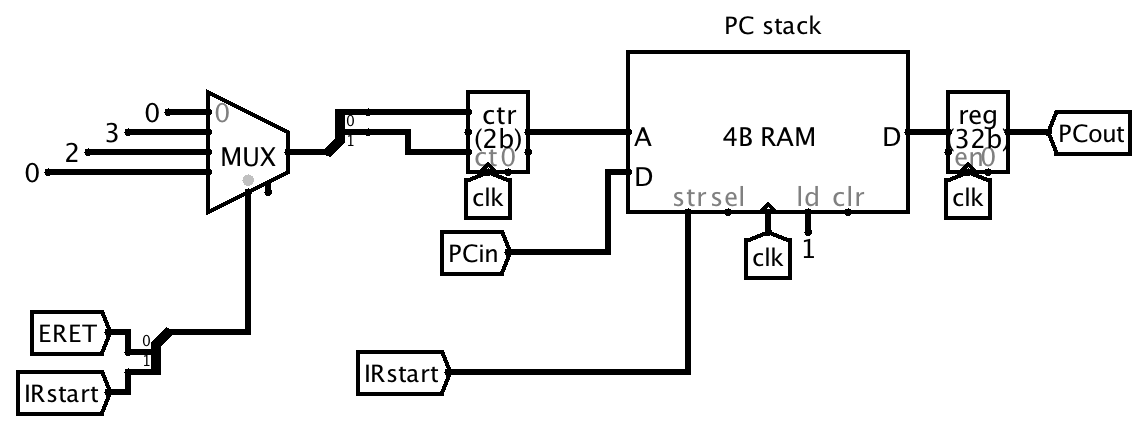


图 3.23 PC存储栈（Logisim）

当每一级中断请求因为IRstart信号开始进入中断时，记录该中断已被响应但尚未运行结束，从而方便在多级中断执行完毕退出时对更低级的中断进行处理。存储中断程序运行状态的电路如图3.24所示。

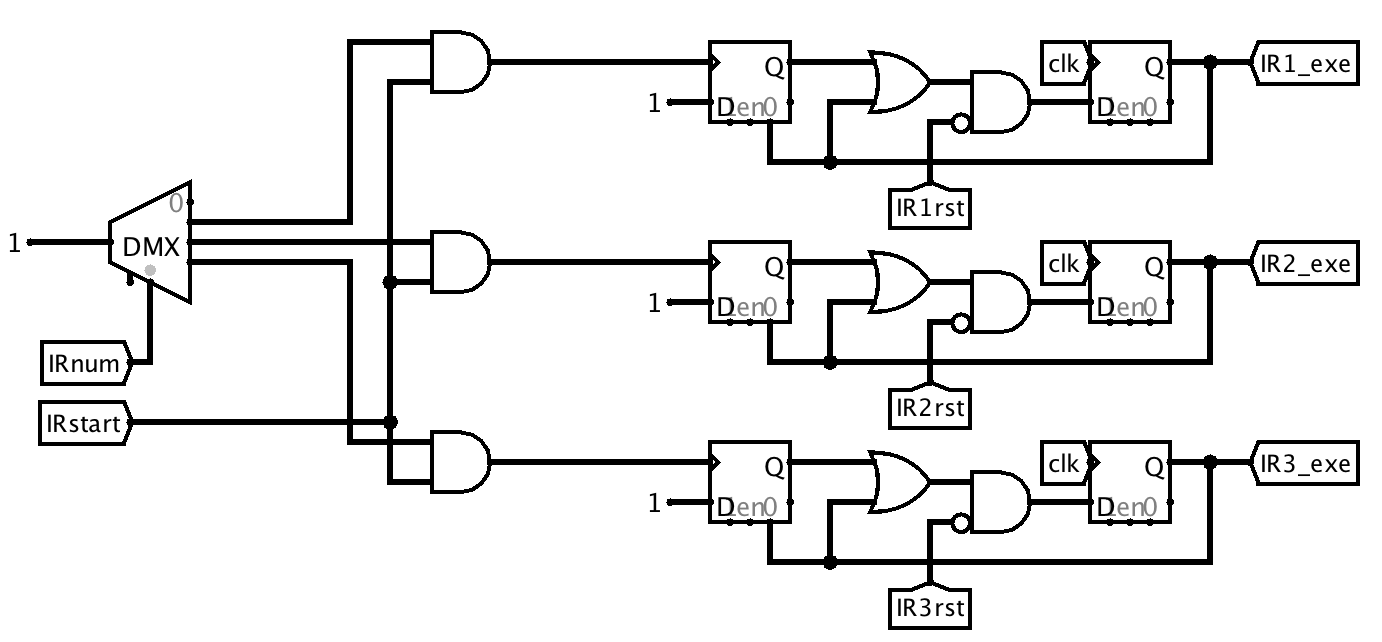


图 3.24 存储中断运行状态（Logisim）

根据硬件维护的各种信息对中断进行处理的电路如图3.25所示。在新的中断请求信号到来时，通过比较器判断新中断是否等级高于旧中断，是则产生IRstart信号从而打断低级中断，否则对新中断进行排队处理。在每个中断处理程序执行完毕触发ERET信号时，若上一级中断已经开始执行则不进行处理，否则产生IRstart信号进入排队中的低级中断。同时在接收到ERET信号时需要产生对应的rst信号重置该中断。

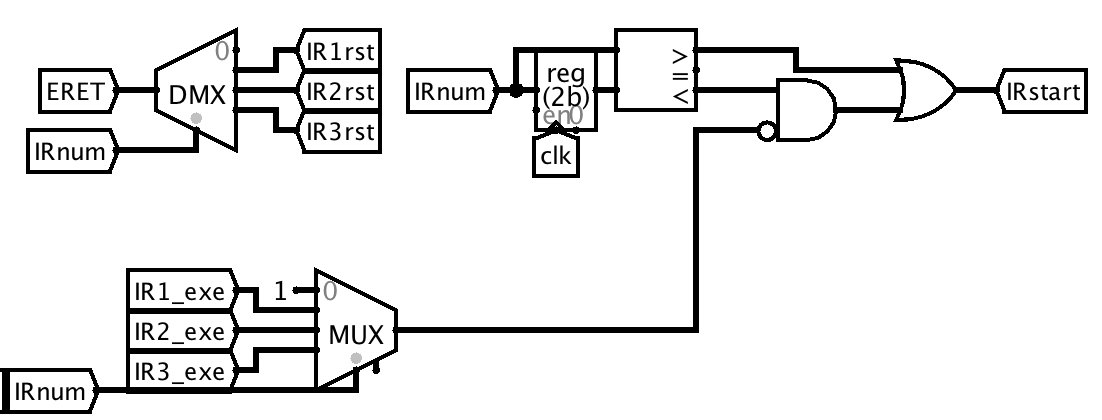


图 3.25 处理中断请求（Logisim）

封装后的中断处理硬件模块CP0如图3.26所示.

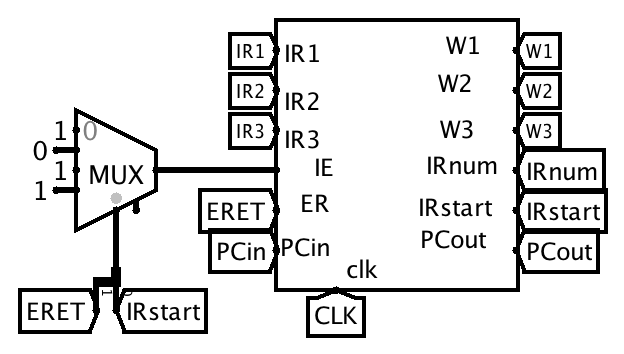


图 3.26 CP0的封装形式（Logisim）

指令存储器需要被扩展为四个ROM，从而方便获得三级中断处理程序的地址。修改后的PC寄存器与IM部分电路如图3.27所示。

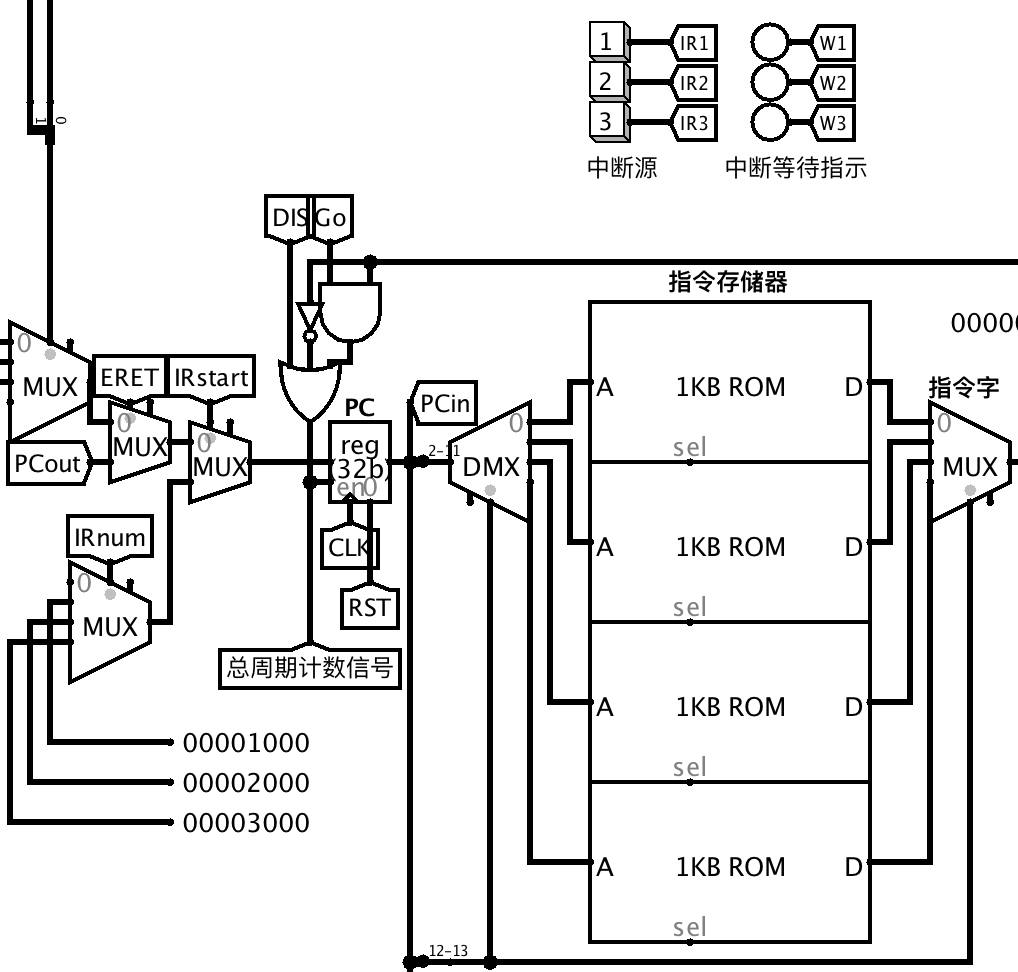


图 3.27 添加中断处理程序后的PC与IM（Logisim）

### 软件实现

根据总体方案设计2.6.3节中的详细设计，在设计中断处理程序时需要注意在进入中断处理程序后先进行现场的保护，随后对输出器件(七段译码管)进行控制以方便观测，退出中断前需要恢复现场，最后执行ERET指令退出中断处理程序。

以一级中断处理程序为例，中断处理程序的MIPS汇编指令如下所示：

.text

addiu $sp,$sp,-4 #入栈，保护现场

sw $s0,0($sp)

addiu $sp,$sp,-4

sw $s2,0($sp)

addiu $sp,$sp,-4

sw $s3,0($sp)

addiu $sp,$sp,-4

sw $t0,0($sp)

addiu $sp,$sp,-4

sw $t1,0($sp)

addiu $sp,$sp,-4

sw $a0,0($sp)

addiu $sp,$sp,-4

sw $v0,0($sp)

addiu $sp,$sp,-4

sw $t8,0($sp)

addi $s0,$zero,1 #中断程序

sll $s3, $s0, 31 # $s3=0x80000000

sra $s3, $s3, 31 # $s3=0xFFFFFFFF

addu $s0,$zero,$zero # $s0=0

addi $s2,$zero,12

…

lw $t8,0($sp) #出栈，恢复现场

addiu $sp,$sp,4

lw $v0,0($sp)

addiu $sp,$sp,4

lw $a0,0($sp)

addiu $sp,$sp,4

…

eret

# 实验过程与调试

## 测试用例和功能测试

在本次实验中，每种原理不同的CPU均需要用不同方法进行测试，但由于重定向流水线囊括了包括单周期CPU、理想流水线等电路在内的具体功能，在本部分仅记录使用benchmark测试程序对支持28条指令的重定向流水线进行的测试过程。

此外，对分支预测功能与多级中断支持需要单独进行测试。

### Benchmark测试重定向流水线

实验提供的benchmark程序中实现了对24条通用指令的测试功能。修改benchmark测试程序，使其额外支持对SB、SLLV、XOR、BLTZ四条指令进行测试。

通过MARS程序对.asm代码进行编译，将生成的十六进制文件导入重定向流水线中基于ROM实现的指令存储器IM，如图4.1所示。

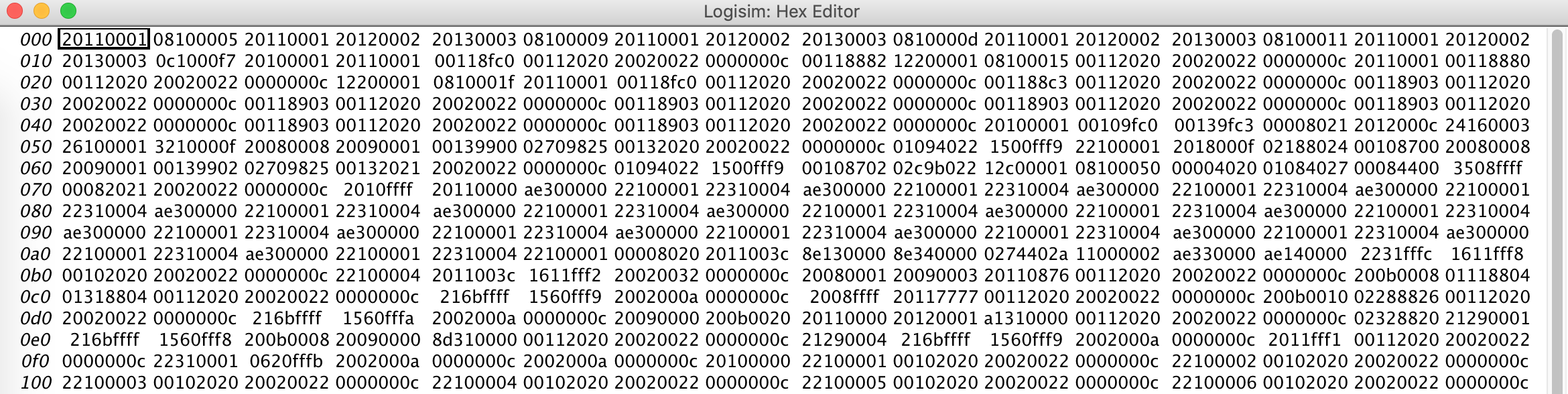


图 4.1 IM指令存储器中的十六进制指令内容

通过ctrl+k或command+k快捷键启动模拟时钟，程序将会开始运行。在运行过程中，程序将驱动八个七段译码管以及数据存储器进行工作，依次通过移位测试、走马灯测试以及排序测试。在移位测试和走马灯测试执行的过程中，需要对七段译码管的输出数据进行观测保证程序的正确运行；在排序测试执行后，需要观察数据存储器中的内容以保证排序功能所需指令的正确性，排序测试结果如图4.2所示。观察到DM中的数据降序排列，说明排序测试程序正确执行。

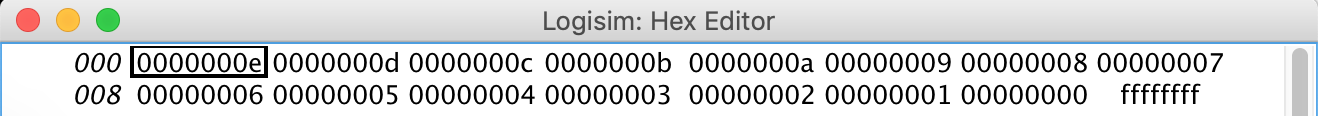


图 4.2 排序测试结束后DM数据存储器中的内容

在benchmark的24条指令测试部分完整运行后，对比程序的总运行周期数与实验要求中的预期周期数，发现总周期数、无条件分支数、有条件分支数、插入气泡数、LoadUse次数均与预期一致，如图4.3所示，这证明了重定向流水线堆24条指令的数据通路设计正确。



图 4.3 24条指令测试程序benchmark执行后的周期计数

对于后续的四条CCMB指令，通过Go按钮继续执行，对测试程序运行结果的观测与24条指令的观测方法类似，不再赘述。测试结果说明设计的重定向流水线CPU能够正确支持28条MIPS指令的正常工作，验证了CPU设计的正确性。

Logisim设计完成后，利用verilog语言与vivado实现重定向流水线的上板，前仿真测试结果如图4.4所示，发现测试程序执行结束后记录的各个周期数据与预期一致，从而进一步验证了CPU设计的正确性。

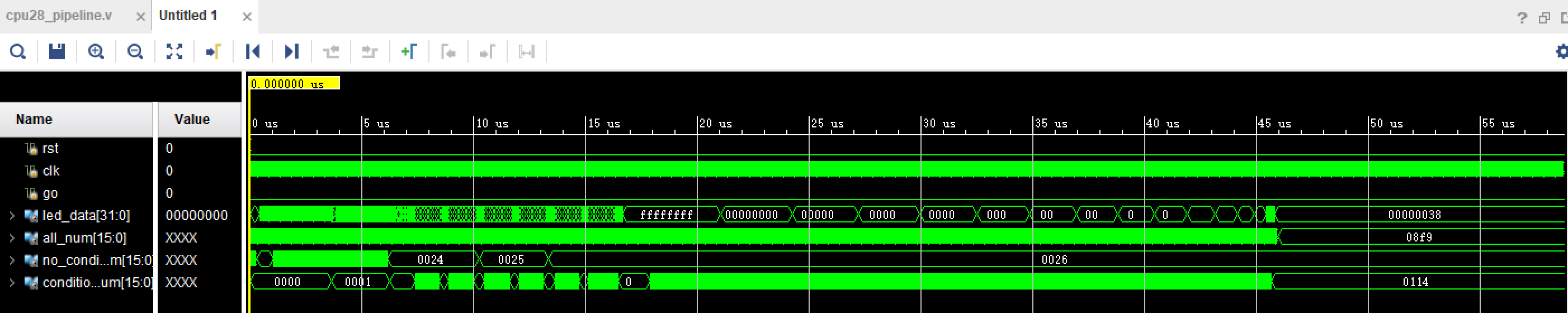


图 4.4 重定向流水线CPU的前仿真结果

最后经过综合并将比特流加载到开发板上后，运行测试程序，输出与预期一致。

### 分支预测

对于支持动态分支预测的重定向流水线，需要使用实验材料包中提供的汇编代码“分支预测测试.asm”进行测试。

如4.1.1中对重定向流水线进行测试一样，将测试程序的汇编代码通过MARS编译，将产生的十六进制文件导入CPU中的指令存储器IM中并启动模拟时钟。等待程序执行结束，周期计数与预期一致，验证了分支预测功能的正确性。

分支预测功能的正确性测试完毕后，对电路的优化效果进行观测：使用分支预测电路执行benchmark测试程序，执行结束后的周期数、分支预测次数以及预测失败数如图4.5所示。如图所示，分支预测成功次数为294-27=267次，分支预测失败数为27次。对应的新总周期数1817+(预测成功次数267-失败次数27)\*2与未使用分支预测的重定向流水线总周期数一致，说明分支预测的结果是可解释的，进一步验证了分支预测功能设计的正确性，也测得该功能为程序执行节省了480个时钟周期。

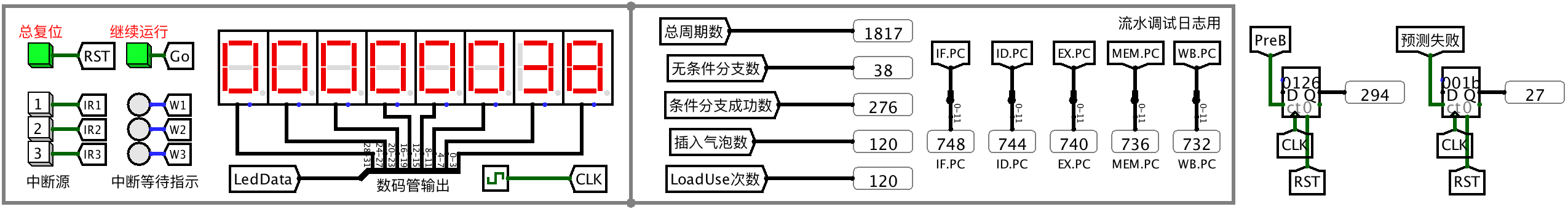


图 4.5 分支预测对benchmark测试程序的优化效果

### 中断测试

首先向支持中断的单周期CPU的指令寄存器中依次装载benchmark测试程序、一级中断处理程序、二级中断处理程序和三级中断处理程序。

首先对多级中断能否被打断和恢复进行测试：令支持中断的单周期CPU执行benchmark测试程序，在程序执行过程中首先请求一级中断，在一级中断处理程序的执行过程中请求二级中断。观察程序运行结果，发现CPU首先切换到二级中断处理程序进行执行，在二级中断执行完毕后触发ERET信号返回一级中断继续执行，最后再回到benchmark程序被中断处继续执行。

再对多级中断中低级中断不能够打断高级中断进行测试：令支持中断的单周期CPU执行benchmark测试程序，在程序执行过程中首先请求二级中断，在二级中断处理程序执行过程中再次请求一级中断。观察程序运行结果，发现虽然一级中断的请求信号灯亮起，但CPU并不会立即跳转到一级中断进行执行，而是等待二级中断处理程序执行完毕触发ERET信号后才进入一级中断并运行一级中断处理程序，最后回到benchmark程序被中断处继续执行。

根据这两次的测试结果，验证了单周期CPU的多级中断处理功能正常。

## 性能分析

在对单周期CPU进行FPGA开发板烧入时，为工作时钟提供了两种分频模式，一种是便于观测benchmark测试程序执行的实时结果的低频模式，另一种直接使用开发板提供的100MHz以测试CPU的最高工作频率，两种分频模式的切换通过开发板上一个开关的高低电平控制。

在比特流烧入完成后，发现单周期CPU在低频模式下运行benchmark程序正常。在切换至高频模式后，通过Go按钮能够控制CPU在执行某一段测试程序后因为syscall指令停止，且由于24条指令的测试程序部分所占运行时间较长，大多数情况下CPU会停止在24条指令测试程序结束时，此时观察程序执行总周期的记录，发现与预期一致，即依然工作正常。

即对于设计的单周期CPU而言，开发板已不能提供更高的时钟频率以测试其关键路径导致的频率极限。对应地，对于重定向流水线而言，也无法通过将电路烧入开发板来实际测试是否效率会高于单周期CPU。但从理论上来看，重定向流水线CPU通过流水线的方式大幅降低了关键路径的长短，同时通过重定向的方式减少了预取深度可能造成的时钟周期浪费，效率应会大大高于纯粹的单周期CPU。

## 主要故障与调试

### 单周期SB指令故障

单周期CPU：数据存储器写入时片选问题。

**故障现象：**执行SB指令时无法正确存储字节。

**原因分析：**分析发现logisim自带的RAM器件所具备的片选输入信号sel只能启用或禁用整片RAM，无法实现对字节或半字等的选取。

**解决方案：**将用作数据存储器DM的RAM器件替换为CS3410资源包中的MIMPS RAM，新的RAM器件中片选端口sel由一位变成了四位，由此可为SB指令设计数据存储器周围的数据通路如图4.6所示。

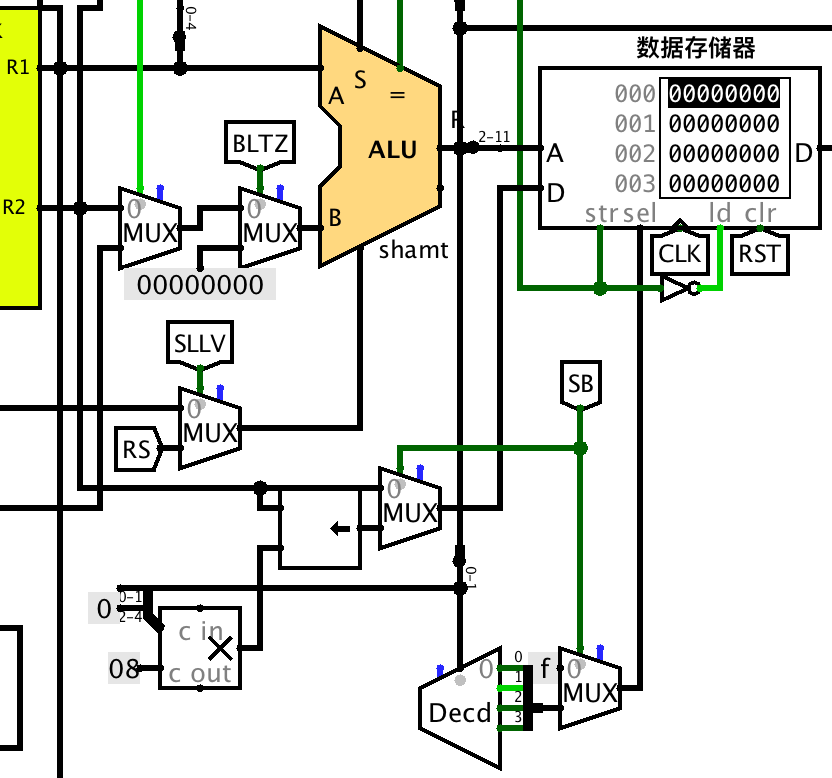


图 4.6数据存储器片选信号设计

### 周期总数比预期少1

单周期CPU：PC寄存器输入有误问题。

**故障现象：**在执行benchmark测试程序时，总周期数比预期周期数少1。

**原因分析：**对benchmark测试程序中的指令逐条执行后发现在执行JAL指令时，向寄存器中传入的数据输入为PC+8，翻阅MIPS指令手册后发现手册上使用PC+8是因为考虑了流水线的延迟槽技术，从而确保了地址为PC+4必然有效能够继续执行，所以跳转到再之后的一条指令即可。

**解决方案：**本次实验中的CPU没有使用延迟槽技术，故而将JAL指令所需的PC+8更换为PC+4即可。

### 单周期上板时控制器信号故障

单周期CPU上板：控制器信号生成函数有误。

**故障现象：**在单周期CPU上板的过程中，经过对vivado前仿真波形的观察，总有信号在被需求时没有输出。

**原因分析：**由于单周期CPU上板这项任务为小组合作，控制器由同组同学进行设计。在与同学交流了对方的真值表后，发现对于J型指令他并没有严格按照实验要求中的要求生成控制信号而是对数据通路进行了一些修改，从而导致不同的通路下信号输出有误。

**解决方案：**在发现问题后便对控制器中J型指令的控制信号进行了修改，从而使程序能够正常执行。

### 流水线中寄存器写回数据有误

理想流水线：寄存器堆写回时机有误。

**故障现象：**在使用理想流水线运行测试程序时，发现所得输出与预期不符，进一步调试发现是寄存器堆中写入了错误数据。

**原因分析：**在写入错误数据的周期中对数据通路进行检查，发现寄存器堆的写回数据输入来自于WB流水段，而写回使能信号来自于ID段，从而导致故障的出现。

**解决方案：**修改控制信号来源为WB段即可，修改后的数据通路如图4.7所示。

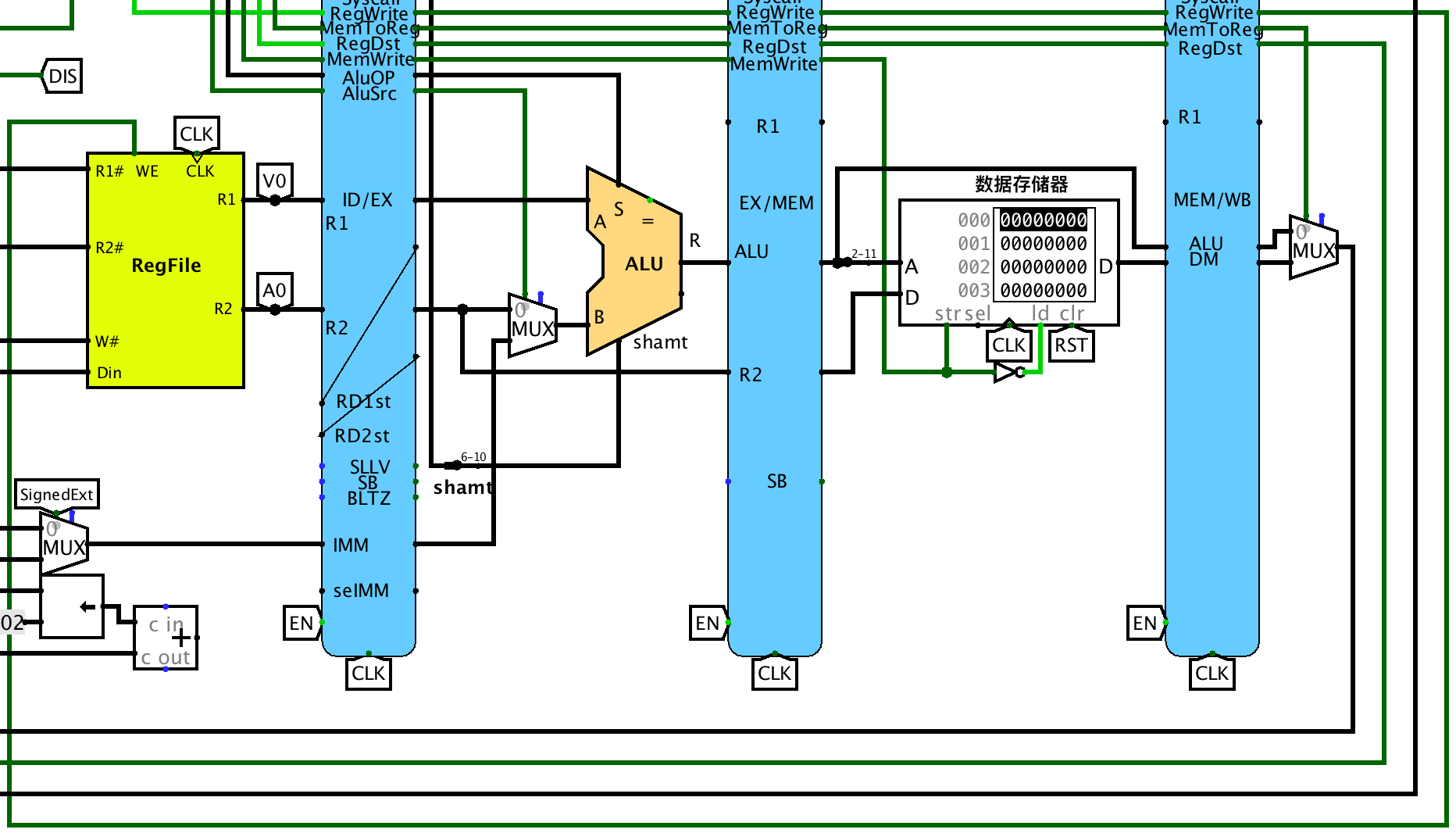


图 4.7 由WB段牵回RegWrite使能信号

### 流水线接口清零故障

分支相关流水线：在检测到分支指令并向流水接口插入气泡时，流水线故障。

**故障现象：**在对流水线接口置清零端使能为高电平后，发现对应的流水段数据并没有清零。

**原因分析：**回忆上学期的流水线编码实验，其流水线接口中的寄存器使用的是同步清零的方式，而本次实验设计中起初使用的是清零信号直接接入寄存器rst端的异步清零，考虑可能是由于异步导致清零时机有误从而清零失败。

**解决方案：**对流水接口器件进行修改，通过多路选择器和接地使所有的寄存器均使用同步清零的方式工作。如图4.8所示。修改流水接口后流水线正常工作。

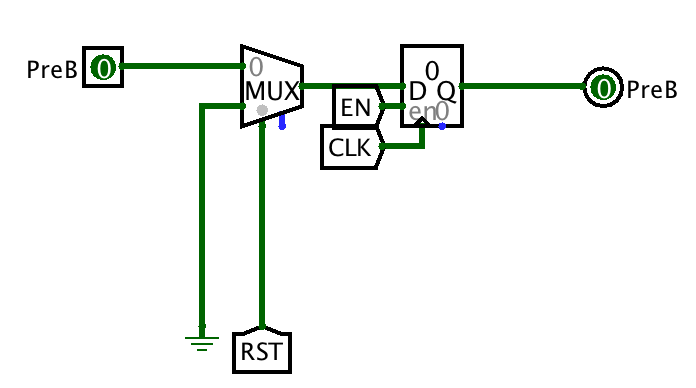
****

图 4.8 同步清零寄存器举例

## 实验进度

表 4.1 课程设计进度表

| 时间 | 进度 |
| --- | --- |
| 第一天 | 小组分工进行单周期CPU上板。负责完成了PC、数据存储器、（有无）符号扩展模块。 |
| 第二天 | 对ROM器件的初始化功能进行了完善、为RAM添加了片选信号以支持SB一类的指令、设计了部分数据通路。 |
| 第三天 | 完成了数据通路的设计，调试通过了单周期CPU的上板测试。 |
| 第四天 | 在logisim平台上完成了理想流水线、分支相关流水线的设计。 |
| 第五天 | 在logisim平台上完成气泡流水线、重定向流水线的设计。 |
| 第六天 | 在logisim平台上完成分支预测的设计。 |
| 第七天 | 在logisim平台上完成多级中断、完成了重定向流水线的上板测试。 |
| 第八天 | 无 |

# 设计总结与心得

## 课设总结

在本次课程设计中，通过循序渐进的方式完成了从单周期CPU到效率大幅提升的重定向流水线CPU最后再到为CPU提供分支预测和多级中断等高级功能的支持等实验内容。总的来说在整个实验过程中作了如下几点工作：

* + 1. 完成了单周期CPU、理想流水线CPU、分支相关流水线CPU、气泡流水线CPU、重定向流水线CPU、支持分支预测的重定向流水线CPU以及支持多级中断的单周期CPU的Logisim设计。同时对于其中的单周期CPU和重定向流水线CPU，还通过FPGA的方式进行了实现与验证。
    2. 设计的单周期CPU能够支持24条基本MIPS指令+4条独立的CCMB指令的运行，对应的测试可以通过benchmark测试程序进行。设计的流水线CPU提升了CPU执行指令的效率，其中气泡流水线CPU解决了数据相关的问题、重定向流水线解决了气泡流水线预取深度造成周期浪费的问题、分支预测流水线解决了重定向流水线中分支指令造成周期浪费的问题。而单周期CPU上的多级中断也能正确实现三级中断支持，且中断屏蔽功能工作正常。
    3. 在将单周期CPU与重定向流水线CPU通过FPGA编程的形式烧入开发板时，实现了通过LED输出周期计数结果、通过拨动开关切换显示域、添加内存观测窗口以通过LED输出RAM中的排序结果、以及通过拨动开关切换CPU工作频率从而测试CPU的最大工作频率。

## 课设心得

**我的体会与收获：**

本次组成原理课程设计实验需要通过两周时间循序渐进设计一个较为复杂的MIPS CPU。总的来说实验难度略难，但实验材料十分齐全，为实验过程提供了很大的帮助，同时在进行实验的过程中我也学到了很多东西。

课设要求在假期中在Logisim平台上完成支持24条指令的单周期CPU设计，由于在组成原理实验中已有类似要求，只需要对支持的指令进行扩展，难度并不高。

课设真正的第一步是以小组形式将单周期CPU以FPGA的方式实现在开发板上，由于是小组开发的模式，便出现了一系列的便利与麻烦。对于硬件描述语言而言，由于硬件本身是模块化的，分工合作十分容易，但由于我们组内在分工设计时并没有统一的原理图，共享模块后容易出现“货不对版”的情况，比如控制器的设计就在J型指令的控制信号生成上出现了分歧，导致需要每个人再对其稍作修改以适应自己的数据通路。事后经过反思，协同开发时一定需要约定好模块的接口，以及模块的内部逻辑也需要进行描述，从而方便他人使用也避免返工的需要。

随后的实验内容则是对单周期CPU性能的逐步改进和对功能的逐步添加。在这一部分的实验内容中，我通过自学掌握了流水线CPU的工作原理以及其性能提升的理由，同时对于其可能产生的数据冲突以及周期浪费等问题也掌握了相应的应对方法，最后为了再进一步提升流水线的工作效率，学习并实现了动态分支预测这一功能。经过这一系列学习我领会到了优化的重要性以及优化的具体方法，可谓学习到了满满干货。同时，我逐步体会到了Logisim与Verilog设计各自的优点，Logisim平台更加直观且便于操作，但其操作虽然简单却又繁琐，同时因为大多时候在上面设计的电路只起到原理图的作用，对于细节上的关键路径、电路险象等不一定能够很好的进行表达；而基于硬件描述语言的FPGA编程则功能十分强劲，但对应地其描述较为抽象，需要对硬件描述语言有一定的掌握才能运用自如，同时因为不能即时将代码转换为电路图，容易出现漏接一根线或接错一根线的情况且查错较难，对编程者的编程水平要求较高。

最后单周期多级中断的支持也是一个难点，其实中断逻辑并不复杂，但要实现硬件支持还是有些吃力。最后我采用了完全使用硬件支持的方式完成了多级中断的任务要求，虽然达到了目的但方式并不优雅。有机会还是要设计一个真正的CP0器件，并通过汇编指令开关中断的方式实现可变更屏蔽字的多级中断。

**我对课程的建议：**

本次课程设计的节奏十分到位，在投入学习的状态下能够不紧不慢的完成所有实验内容并“知其所以然”，同时mooc的实验讲解和实验包中材料的提供也十分详尽且十分有用，课设的学习体验很棒，绝对是掌握知识最多的一门课设。

但对于课设我依然有一些建议，首先是小组合作的方式虽然名分在，但并没有很好的被运用起来，比如合作单周期上板，在分工设计完基础器件后，由于CCMB导致各个同学的通路不同，直接设计一份28条指令通路和看懂他人代码并修改为28条指令通路的工作量并没有显著差异，自此小组合作也几乎变为了“坐在一块做自己的实验”，对应的其实可以设置一部分完全一致的实验内容作为小组合作的攻克对象，或者增加小组分工设计的范围，比如需要组员自己编写用于测试的汇编程序等，可能能更加发挥出小组合作的功效。

其次是对于课设最后的成果，在已经设计出能够正常工作的CPU的基础之上，其实可以考虑利用CPU运行一些比较有趣的程序而不只是benchmark测试程序，甚至也可以考虑用上Nexys开发板上的一些拓展接口，应当会使实验更加具有趣味性。

实验内容中其实还有很多部分值得我慢慢回味和摸索学习，总之十分感谢课程组老师对这门课程设计付出的心血，希望这门课能越办越好！

# 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:** |