***2017***



**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： | ACM1401 |
| 学 号： | U201414620 |
| 姓 名： | 胡汉鹏 |
| 电 话： | 15927093748 |
| 邮 件： | [540358178@qq.com](mailto:540358178@qq.com) |
| 完成日期： | 2017-03-10 周五上午 |

目 录

[1 课程设计概述 1](#_Toc478115227)

[1.1 课设目的 1](#_Toc478115228)

[1.2 设计任务 1](#_Toc478115229)

[1.3 设计要求 1](#_Toc478115230)

[1.4 技术指标 2](#_Toc478115231)

[2 总体方案设计 4](#_Toc478115232)

[2.1 单周期CPU设计 4](#_Toc478115233)

[2.2 中断机制设计 9](#_Toc478115234)

[2.3 流水CPU设计 11](#_Toc478115235)

[2.4 气泡式流水线设计 12](#_Toc478115236)

[2.5 数据转发流水线设计 13](#_Toc478115237)

[2.6 FPGA流水线设计 13](#_Toc478115238)

[2.7 中断流水线设计 13](#_Toc478115239)

[2.8 动态分支预测机制 14](#_Toc478115240)

[3 详细设计与实现 15](#_Toc478115241)

[3.1 单周期CPU 实现 15](#_Toc478115242)

[3.2 中断机制实现 23](#_Toc478115243)

[3.3 流水CPU实现 28](#_Toc478115244)

[3.4 气泡式流水线实现 30](#_Toc478115245)

[3.5 数据转发流水线实现 36](#_Toc478115246)

[3.6 FPGA流水线实现 41](#_Toc478115247)

[3.7 中断流水线实现 42](#_Toc478115248)

[3.8 动态分支预测机制实现 45](#_Toc478115249)

[4 实验过程与调试 49](#_Toc478115250)

[4.1 测试用例和功能测试 49](#_Toc478115251)

[4.2 性能分析 53](#_Toc478115252)

[4.3 主要故障与调试 54](#_Toc478115253)

[4.4 实验进度 56](#_Toc478115254)

[5 设计总结与心得 57](#_Toc478115255)

[5.1 课设总结 57](#_Toc478115256)

[5.2 课设心得 57](#_Toc478115257)

[6 参考文献 60](#_Toc478115258)

# 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，简化 |
| 26 | MTC0 | 访问CP0 | 中断相关，简化 |
| 27 | ERET | 中断返回 | 异常返回 |
| 28 | SRAV | 算术可变右移 |  |
| 29 | XORI | 异或立即数 |  |
| 30 | LH | 加载半字 |  |
| 31 | BGEZ | 大于等于 0 转移 |  |

# 总体方案设计

## 单周期CPU设计

本次单周期CPU的设计采用的是硬布线的方式，通过控制器对指令译码产生控制电路的各种信号，直接送至相关部件运行。将指令存储器和数据存储器分开，可以一定程度上避免结构相关。同时在实施过程中采用自己在前面设计的ALU和寄存器组完成实验。

总体结构图如图 2‑1所示。



图 2‑1总体结构原理图

本实验要做一个基于MIPS指令的单周期的CPU，主要需要以下几个部件组成：

1. 时钟信号产生，对时钟信号进行处理，当有停机信号时屏蔽时钟信号，否则按照一定频率产生脉冲信号。
2. 指令存储器，用只读存储器存储需要执行的指令，控制指令按一定次序被读出并执行。
3. 控制器模块，针对从指令存储器中读出的MIPS指令，解析后产生CPU操作的各种信号，并送到各个部件控制各部件的工作，实现指令要求实现的功能。
4. 寄存器组，32个寄存器，0号寄存器始终为0.，其他寄存器用于暂时报错运算过程中的数据。
5. ALU，实现算术逻辑运算。
6. 数据存储器，针对lw和sw操作，对数据存储器要进行读写操作，程序操作过程中需要或者会产生数据，这些数据可以通过数据存储器保存下来。
7. 指令地址控制部件，正常情况下指令每次执行后指令地址自增4，跳转指令会改变指令计数器的值，针对不同的跳转 指令要设置PC的值。
8. 数码管显示模块，当$v0不为0时，数码显示$a0的值，并且每次syscall刷新一次。
9. 书上还有一个ALU\_OP控制器，这里将ALU\_OP控制器与总的控制器合并，即ALU功能选择信号也由控制器一起生成。
10. 指令计数模块，对总的指令、R型、I型和J型指令进行计数。

由于上次组原实验已经大致完成单周期CPU的设计，这里的再次更改后增加了四条扩展指令。

### 主要功能部件

单周期CPU各个功能部件的具体设计思路如下。

#### 程序计数器PC

程序指令计数器PC的设计思想大致分为以下几个部分。首先用一个寄存器锁存当前PC的值，而下一个PC作为输入端，每当时钟上升沿来临，将输入端的下一PC输入到寄存器。第二部分是决定下一PC的电路。指令执行下一PC有以下几种情况：

1. 直接跳转指令j和jal，根据指令给出的地址决定下一个PC值；
2. 直接跳转指令jr，根据指令指定的寄存器的值决定下一个PC的值；
3. 分支指令bne、beq、bgez，根据计算的结构决定是否跳转，如果决定跳转，下一PC值由指令给出的跳转地址得到；
4. 除以上几种情况，NPC应等于PC值加4。

#### 指令存储器IM

指令存储器用logisim自带的只读存储器ROM实现，数据宽度设置为32位，地址宽度设置为10位。

#### 运算器ALU

如表 2‑1和表 2‑2，运算器ALU引用自行设计的32位先行加法器模块以及其他Logisim自带相应模块实现ALU基本功能。其基本思路对于Result和Result2，其中两数相加和相减均由32位先行加法器实现，所有功能的结果都会生成，最后通过选择器选择具体输出相对应功能需要的输出。有符号加减溢出符号OF、无符号加减溢出符号UOF和相等符号Equal符号单独生成。

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 运算符功能

|  |  |  |
| --- | --- | --- |
| 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] 有符号 |
| 0100 | 4 | Result = X/Y; Result2 = X%Y 无符号 |
| 0101 | 5 | Result = X + Y Result2=0 (Set OF/CF) |

### 寄存器堆RF

用32个Logisim自带寄存器构成32\*32位寄存器组，每个寄存器与时钟输入相连，0号寄存器始终为0，使能端置0使其不能写入。读出数据时将所有用reg1#和reg2#分别选择32 寄存器的输出信号，写入数据时用write\_reg#选择具体写入哪一个寄存器，即用write\_reg#选择write\_date和WE导通的寄存器。选择相应的数据通路，将相应使能端置为1，写入数据。

### 数据通路的设计

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。

下面给出指令系统数据通路的整体框架，如表 2‑3所示。

表 2‑3指令系统数据通路框架

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | PC | RF | | | | ALU | | | DM | |
| R1# | R2# | W# | Din | A | B | OP | Addr | Din |
| lh | PC+4 | base/rs | rt | 1 | rt | R1 | offset | 5 | AluReslut | R2 |

### 控制器的设计

该实验最重要的就是通过控制器产生CPU各部件运算需要的信号。实现控制器的思路是对mips指令的op字段和func字段进行译码，得到是何种指令后根据指令类型分别产生各个信号，如表 2‑4所示。

表 2‑4 控制信号信息表

|  |  |  |  |
| --- | --- | --- | --- |
| 控制信号 | 位宽 | 功能 | 产生方式 |
| R型指令 | 1 | 判断是否为R型指令，为1时代表是R型指令 | 对应R型指令功能信号相或产生 |
| Shamt | 1 | 判断是否为移位指令sll、sra或srl，为1时选择ALU的两个输入端输入分别为R2和（zero-extend）shamt | sll、sra和srl对应功能信号相或产生 |
| ALUop | 4 | 产生ALU操作的功能号 | 根据各功能号代表的运算分别生成对应的ALU功能号，默认为加法5，再用数据选择器选择生成 |
| Jmp | 2 | 选择输入指令计数器的数据来源。  00代表是j或jal功能产生的地址  01代表顺序地址或bne或beq跳转地址  10和11代表jr对应寄存器地址 | 低位来自jal+j  高位来自jr |
| Syscall | 1 | 代表系统功能调用信号，当该控制信号为1，$v0为0时，产生停止PAUSE信号；若$v0不为0，数码管更新$a0的最新数据 | 来自syscall功能信号 |
| ALUSrc | 1 | 与信号Shamt一起选择输入ALU第二个操作数的来源，当Shamt为0且ALUSrc为1时，选择来自immediate扩展后的数据；当Shamt为0且ALUSrc为0时，选择来自rt的数据； | 由addi、addiu、andi、ori、slti、lw、sw相或产生 |
| beq | 1 | 为1且ALU的Equal输出为1时跳转到immediate扩展后的地址 | 来自beq功能信号 |
| bne | 1 | 为1且ALU的Equal输出为0时跳转到immediate扩展后的地址 | 来自bne功能信号 |
| bgez | 1 | 为1且ALU的比较结果最低位为1时跳转到immediate扩展后的指令 | 来自bgez功能信号 |
| RegDst | 2 | 用于选择写入寄存器组的寄存器号。  00代表写入rt  01代表写入rd  10和11代表写入$s31 | 低位来自ALUSrc取反，高位来自jal功能信号 |
| RegWrite | 1 | 写寄存器使能信号，为1时代表允许写寄存器 | 由addi、addiu、andi、ori、slti、lw、jal和所有除jr和syscall的R型功能信号相或产生 |
| MemWrite | 1 | 为1代表允许写数据存储器 | 来自sw功能信号 |
| MemRead | 1 | 为1代表允许读出数据存储器内容 | 来自lw功能信号 |
| toReg | 2 | 选择送至寄存器数据的来源  00代表来自ALU相加  01代表来自数据存储器的输出  10和11代表当前PC | 低位来自lw功能信号，高位来自jal功能信号 |
| Unsigned | 1 | 代表对数据处理用无符号方式，为1时对immediate扩展时采用无符号扩展 | 来自addiu、addu、andi、ori、sltu相或 |
| srav | 1 | 算术右移，为1时算术右移，右移位数等于rs中数据的低5位指定的偏移 | 来自srav命令 |
| lh | 1 | 加载半字信号，从RAM中取出offset处开始的16位数据 | 来自lh命令 |

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 2‑5所示。

表 2‑5主控制器控制信号框架

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | R型指令 | Shamt | ALUop(十进制) | Jmp | Syscall | ALUSrc | beq | bne | bgez |
| lh | 0 | 0 | 5 | 01 | 0 | 1 | 0 | 0 | 0 |

## 中断机制设计

### 总体设计

本实验实现多级嵌套中断。总共有3个按键中断源，分别对应3个中断服务程序，且优先级为3>2>1>0(CPU)。不同按键可以触发不同的中断服务程序，如正在执行中断，高优先级中断课打断当前中断服务，且中断过程中能缓存新的中断请求。需要构造一个简化的CP0协处理器，用于存储中断时的返回地址epc、屏蔽字以及相应的中断号。另需要有一个中断处理电路以及中断仲裁电路以锁存中断信号并选择中断信号。中断处理模块思路如图 2‑2所示。

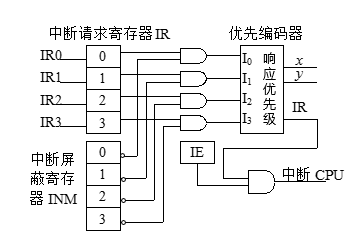


图 2‑2中断结构原理

### 硬件设计

硬件上首先需要构造一个简化的CP0协处理器，具有读写功能。0号寄存器存储发生中断的返回地址npc也即epc，1号寄存器存储屏蔽字，2号寄存器存储发生中断时的中断号。

其次需要构造中断产生电路，对每一个中断信号，对应一个寄存器锁存信号，并且锁存之前需要判断是否reset，若reset则屏蔽。信号引出后还需要判断对应屏蔽位是否为1，若为1则屏蔽。而后进入优先选择模块，选出优先级最高的中断信号，其余信号若有则等待。优先选择器能产生中断号以及中断有无信号，中断号扩展之后送入2号CP0寄存器。

另外为了方便实现，这里将指令存储器ROM分为搞地址和低地址两部分，当中断来领，CPU从髙地址首地址开始运行，继而能够执行事先存放在里面的中断程序。

中断来临的时候，硬件能够自动将npc压入0号CP0寄存器，中断程序运行结束时对eret信号处理，能够将CP0中的epc送入PC寄存器，并且接受排队等待的下一中断信号。

### 软件设计

软件层次，中断进入后首先需要关中断，这里通过设置屏蔽字实现。关中断时，屏蔽字需要设置为111。而后能够将当前epc压栈，并将中断程序中用到的寄存器进行压栈。接着开中断，将屏蔽字设置为000即可。而后读取2号CP0寄存器读取中断号，根据中断号判断进入哪一个中断处理程序。进入中断处理程序后，设置屏蔽字。对3号中断，屏蔽字应设置为111；对2号中断，屏蔽字应设置为011；对1号中断，屏蔽字应设置为001；否则屏蔽字应设置为00。由于单周期CPU设计模块已经实现了mfc0、mtc0指令，可以直接用这两条指令对1号寄存器进行读写而设置屏蔽字。中断程序结束后恢复寄存器，并将epc从栈中压回CP0的0号寄存器，此阶段开始前需要关中断，结束后需要开中断。最后执行eret指令，返回用户程序。

## 流水CPU设计

### 总体设计

单周期CPU在一个周期内就执行完一条指令，下面将指令的执行分为五个阶段，依次为取指令IF、指令译码ID、执行运算EX、访存阶段MEM、写回阶段WB。每个阶段为一个时钟周期，多条指令可以同时在CPU上运行。每两个阶段之间用一个流水寄存器缓冲锁存各指令执行至当前段时的状态。

### 流水接口部件设计

各个段需要给下一个段传递信息，以使得指令的状态能够传递。各个段需要传递的信息如下。

1. IF->IF/ID：  
   下一指令PC即NPC、指令IR
2. ID->ID/EX：  
   下一指令PC即NPC、指令IR、R1、R2、扩展后的偏移值shamt、R2、扩展后的立即数immediate、偏移信号isShamt、ALU的op信号AluOp、ALU操作数选择信号AluSrc、beq信号、bne信号、写寄存器信号RegWrite、写内存信号MemWrite、读内存信号MemRead、写寄存器数据选择信号toReg、lh信号、srav信号、BGEZ信号、停止信号pause、mfc0信号、mtc0信号、写寄存器的编号Write\_reg。
3. EX->EX/Mem：  
   ALU运算结果AluResult、下一指令PC即NPC、指令IR、写寄存器信号RegWrite、写内存信号MemWrite、读内存信号MemRead、写寄存器数据选择信号toReg、lh信号、srav信号、BGEZ信号、停止信号pause、mfc0信号、mtc0信号、写寄存器的编号Write\_reg。
4. Mem->Mem/WB：  
   ALU运算结果AluResult、内存读取数据MemData、下一指令PC即NPC、指令IR、写寄存器信号RegWrite、写寄存器数据选择信号toReg、停止信号pause、mfc0信号、mtc0信号、写寄存器的编号Write\_reg。

### 理想流水线设计

理想流水线中，假设所有对象均通过同样的部件，不同的阶段之间没有共享的资源，且各阶段传输延迟一致，进入流水线的对象不收其他阶段的影响，且假设指令中没有分支指令。因此实现理想流水线只需要将原来的单周期CPU分为5个相应的阶段，每两个阶段之间用一个流水寄存器存储当前指令执行到目前段时的状态。其结构示意图如图 2‑3所示。



图 2‑3理想流水原理图

## 气泡式流水线设计

由于指令之间存在数据相关和分支指令，所以需要插入气泡来解决这两个问题。

针对数据相关，在ID段判断当前指令所用的寄存器是否满足以下条件：

1. ID段当前指令确定要读某个寄存器；
2. 该寄存器号与在EX段或Mem段或WB段的状态中要写的寄存器号WriteReg相等；
3. 满足条件2的同时相应段的写寄存器信号RegWrite为1；

满足以上条件，则发生了数据相关，解决方法为向后段插入一个气泡（接口信号清零，最关键的是写回信号）向前给出阻塞信号（流水线停顿）避免当前指令被新指令。具体需要给ID/EX流水寄存器中nop信号，给PC寄存器和IF/ID寄存器一个halt信号，使得下一始终周期EX段为一个空指令，IF和ID段等待。

另外还有一些特殊指令会用到特定的寄存器，需要另外判断，如syscall指令。

这里需要更改ID/EX流水线寄存器和IF/ID流水线寄存器，使其分别具有传递空指令功能和暂停传递指令功能。

此外对于分支指令bne、beq以及bgez需要在EX段才知道下一指令的PC，因此若成功跳转，需要在IF/ID流水线寄存器和ID/EX流水线寄存器输入插入气泡信号。另外对于直接跳转指令j、jal、jr则在运行到ID段的时候得知下一指令PC，需要在IF/ID流水线寄存器输入插入气泡信号。

## 数据转发流水线设计

部分数据相关可以通过数据重定向解决。实现方法分为以下几种情况：

1. ID段和EX段的寄存器发生冲突，而EX段的指令不是load指令或者mfc0指令，则直接将EX段ALU的运算结果作为ID段的对应寄存器的值；
2. ID段和EX段的寄存器发生冲突，而EX段的指令是load指令或者mfc0指令，则需要用插入气泡的方法解决Load\_Use冲突，其解决方法同气泡流水线。
3. ID段和Mem段的寄存器发生冲突，而EX段的指令不是load指令或者mfc0指令，则直接将Mem段ALU的运算结果作为ID段的对应寄存器的值；
4. ID段和EX段的寄存器发生冲突，而EX段的指令不是load指令，则直接将Mem段内存读出数据MemDate作为ID段的对应寄存器的值；
5. ID段和EX段的寄存器发生冲突，而EX段的指令不是mfc0指令，则直接将对应协处理器的寄存器的值作为ID段的对应寄存器的值；
6. 针对特殊指令如syscall等，直接判断EX段和Mem段是对a0或v0寄存器进行写操作，若使用了则直接将相应段的ALU计算结果作为相应寄存器的值进行处理。

## FPGA流水线设计

在logisim上完成重定向流水线之后，将其用verilog语言在vivado上实现。此处针对指令存储器ROM和数据存储器RAM由vivado自带的IP核实现，其他模块都由自己编写。

## 中断流水线设计

前面已经实现单周期CPU上的中断处理，中断流水大体思路同单周期中断，主要改动如下。首先选择中断的时机，合适的中断时机是在写回段WB，在写回段打断执行后前面四段都插入气泡后执行中断程序即可。但本人采用的是在译码段打断程序，故需要等待当前译码段指令执行结束后再进入中断程序。主要实现方法是中断来临后设置计数器，连续在IF段插入气泡，等到中断来临时最后一条指令执行结束，计数器产生进位，下一指令地址NPC由硬件送入协处理器的0号寄存器，后指令跳转到髙地址ROM运行，开始执行中断程序。中断程序结束后再将epc的值作为npc返回程序断点。

## 动态分支预测机制

实现动态分支预测的基础是建立一张BHT表(Branch History Table)，这里总共采用8个表项，表项内容分别为指令的PC值，分支指令的跳转地址BranchAddr，双预测位以及LRU调度标记。

这里采用的最久未使用方法使用以下策略。初始化时LRU依次标记为0到7，每次查找若与查找PC不相等，则LRU加1，若相等则清零，最后在做替换的时候只要查找LRU为7的表项即可。

在IF取值阶段，首先用当前PC值为关键字在BHT表中做全相联比较查找，是否有相等的PC在表中，若有，则根据双预测位决定是否跳转；否则NPC仍旧处理为PC+4。这里采用的双预测位的方法，00和01判断为不跳转，10和11则判断为跳转。

在EX执行阶段，再次用当前PC值为关键字在BHT表中做全相联比较，若命中，且预测位判断结果和实际跳转结果相等，则更新预测位，预测成功预测位加1，预测失败则预测位减1，同时LRU标记为0；若预测位判断结果和实际跳转结果不同，则预测失败，按实际跳转结果更新预测位的同时需要在IF和ID段插入气泡，同时LRU调度标记也要标记为0；若BHT开始就没有命中，选取LRU值为7的表项替换，讲对应信息填入该表项。

以上加1或者减1在到达取值边界的时候都不执行。

# 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，触发方式为上升沿触发，输入为下一条将要执行的指令的地址，输出为当前执行指令的地址。PAUSE为停机信号，当PAUSE为0的时候，CPU停止工作。将PAUSE信号取非之后与trap\_halt信号和nop信号以及eret\_halt信号相或，结果再取非作为程序计数器PC的使能端。Nop信号在之后的气泡流水线和重定向中用到，trap\_halt信号和eret\_halt信号都是用来实现中断。如图 3‑1所示。

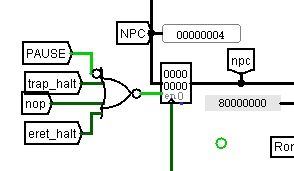


图 3‑1程序计数器（PC）

1. FPGA实现：

程序计数器PC的Verilog代码如下：

register\_32 reg1(.date(pc\_temp),

.clk(clk),

.en( (~(~PAUSE | nop | trap\_halt | eret\_halt)) & start),

.clr(clear),

.out(npc));

指令存储器（IM）

1. Logism实现：

使用两个只读存储器ROM实现指令存储器（IM）。设置两个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，使用分线器只取32位指令地址的2-11位作为两个指令存储器的输入地址，取指令最高位作为片选信号，最高位为0，表示低地址，最高位为1表示髙地址。低地址存放用户程序，髙地址存放中断程序。如图 3‑2所示。

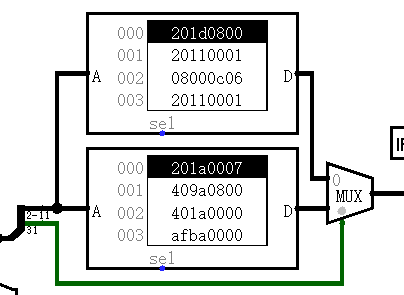


图 3‑2指令存储器（IM）

1. FPGA实现：

直接使用Vivado中自带IP核中的ROM作为指令存储器，其设置如图 3‑3所示。选择ROM的数据位宽为32位，因为该ROM的地址位宽为10位，所以选择ROM的大小选择为1024。

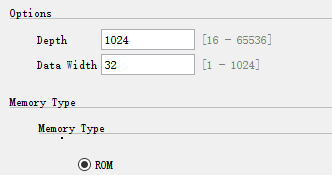


图 3‑3 IP核实现ROM

指令存储器IM的Verilog代码如下：

Rom\_User benchmark (

.a(cpc[11:2]), // input wire [9 : 0] a

.spo(ir\_low) // output wire [31 : 0] spo

);

Rom\_Trap your\_instance\_name (

.a(cpc[11:2]), // input wire [9 : 0] a

.spo(ir\_high) // output wire [31 : 0] spo

);

mux2\_1width\_32 mux19(cpc[31], ir\_low, ir\_high, IR\_IF);

直接调用之前设置的ROM作为指令存储器，输入为指令地址的2-11位，输出为该指令。

### 数据通路的实现

本次课程设计采用的工程化的设计模式，一次性构建所有的数据通路。主要实现方法为，对于每一条指令，将其改写成RTL（Register Transfer Level），忽略控制类信号，仅保留数据类信号，根据RTL功能填写对应指令的数据通路表，描述五大部件之间的连接关系，记录各部件输入端数据来源。

根据总体方案设计中数据通路设计那一小节的详细内容，具体分析每一条指令在执行过程中各个主要部件的输入和输出端口的连接，完成指令系统数据通路表的填写，如表 3‑1所示。

表 ‑指令系统数据通路表

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | PC | RF | | | | ALU | | | DM | |
| R1# | R2# | W# | Din | A | B | OP | Addr | Din |
| lh | PC+4 | base/rs | rt | 1 | rt | R1 | offset | 5 | AluReslut | R2 |
| lw | PC+4 | base/rs | rt | 1 | rt | R1 | offset | 5 | AluReslut | R2 |
| sw | PC+4 | base/rs | rt | 0 |  | R1 | offset | 5 | AluReslut | R2 |
| add | PC+4 | rs | rt | 1 | rd | R1 | R2 | 5 | AluReslut | R2 |
| addu | PC+4 | rs | rt | 1 | rd | R1 | R2 | 5 | AluReslut | R2 |
| and | PC+4 | rs | rt | 1 | rd | R1 | R2 | 7 | AluReslut | R2 |
| nor | PC+4 | rs | rt | 1 | rd | R1 | R2 | 10 | AluReslut | R2 |
| or | PC+4 | rs | rt | 1 | rd | R1 | R2 | 8 | AluReslut | R2 |
| sll | PC+4 | rs | rt | 1 | rd | R2 | shamt | 0 | AluReslut | R2 |
| slt | PC+4 | rs | rt | 1 | rd | R1 | R2 | 11 | AluReslut | R2 |
| sltu | PC+4 | rs | rt | 1 | rd | R1 | R2 | 12 | AluReslut | R2 |
| sra | PC+4 | rs | rt | 1 | rd | R1 | shamt | 1 | AluReslut | R2 |
| srav | PC+4 | rs | rt | 1 | rd | R2 | R1 | 1 | AluReslut | R2 |
| srl | PC+4 | rs | rt | 1 | rd | R2 | shamt | 2 | AluReslut | R2 |
| sub | PC+4 | rs | rt | 1 | rd | R2 | R2 | 6 | AluReslut | R2 |
| subu | PC+4 | rs | rt | 1 | rd | R1 | R2 | 6 | AluReslut | R2 |
| xor | PC+4 | rs | rt | 1 | rd | R1 | R2 | 9 | AluReslut | R2 |
| addi | PC+4 | rs | rt | 1 | rt | R1 | immediate | 5 | AluReslut | R2 |
| addiu | PC+4 | rs | rt | 1 | rt | R1 | immediate | 5 | AluReslut | R2 |
| andi | PC+4 | rs | rt | 1 | rt | R1 | immediate | 7 | AluReslut | R2 |
| ori | PC+4 | rs | rt | 1 | rt | R1 | immediate | 8 | AluReslut | R2 |
| slti | PC+4 | rs | rt | 1 | rt | R1 | immediate | 11 | AluReslut | R2 |
| sltiu | PC+4 | rs | rt | 1 | rt | R1 | immediate | 12 | AluReslut | R2 |
| xori | PC+4 | rs | rt | 1 | rt | R1 | immediate | 9 | AluReslut | R2 |
| beq | if (GPR[rs] == GPR[rt]) PC = PC + sign\_extend(offset<<02) else PC = PC + 4 | rs | rt | 0 |  | R1 | R2 | 5 | AluReslut | R2 |
| bgez | if (GPR[rs] >= 0) PC = PC + sign\_extend(offset<<02) else PC = PC + 4 | rs | rt | 0 |  | R1 | R2 | 11 | AluReslut | R2 |
| bne | if (GPR[rs] ≠ 0) PC = PC + sign\_extend(offset<<02) else PC =PC + 4 | rs | rt | 0 |  | R1 | R2 | 5 | AluReslut | R2 |
| j | target | rs | rt | 0 |  | R1 |  | 5 | AluReslut | R2 |
| jal | target | rs | rt | 0 | $31 | R1 |  | 5 | AluReslut | R2 |
| jr | （rs） | rs | $31 | 0 |  | R1 |  | 5 | AluReslut | R2 |
| eret | 协处理器中的npc | rs | rt | 0 |  | R1 |  | 5 | AluReslut | R2 |
| mfc0 | PC+4 | rs | rt | 1 | rt | R1 |  | 5 | AluReslut | R2 |
| mtc0 | PC+4 | rs | rt | 0 |  | R1 |  | 5 | AluReslut | R2 |
| syscall | PC+4 | rs | rt | 0 |  | R1 |  | 5 | AluReslut | R2 |

在完成指令系统数据通路表的填写之后，根据列出的数据通路表，进行多指令数据通路的合并输入数、表，将各个主要功能部件进行连接，根据数据通路合并表的最终结果，对于所有的多输入部件使用多路选择器进行输入选择。最终便可以完成数据通路的搭建。如图 3‑4，是最初的单周期版本，故PC寄存器处于上述介绍有所不同，在后面会另加说明有所变动。

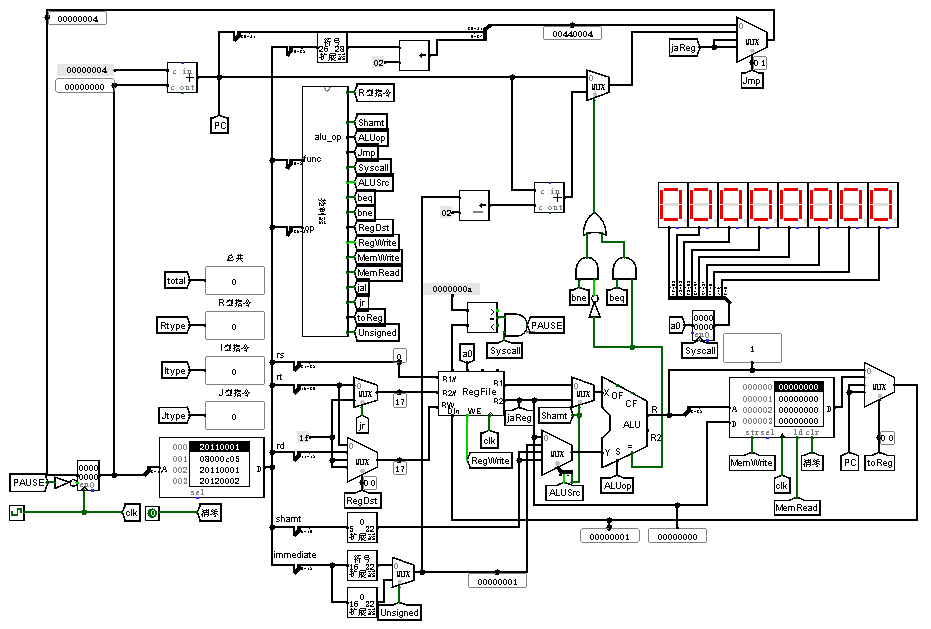


图 3‑4 单周期CPU数据通路（Logism）

### 控制器的实现

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

首先利用表达式生成法，对12位输入op和func转化为24+4位输出，每一位代表一个功能号，也即一种mips指令。因此当相应输出为1时，表示为相应指令，每次仅会有一位输出为1。再利用28位功能号输出产生控制信号，各控制信号以及信息如表 3‑2所示。

表 3‑2主控制器控制信号

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | R型指令 | Shamt | ALUop(十进制) | Jmp | Syscall | ALUSrc | beq | bne | bgez |
| lh | 0 | 0 | 5 | 01 | 0 | 1 | 0 | 0 | 0 |
| lw | 0 | 0 | 5 | 01 | 0 | 1 | 0 | 0 | 0 |
| sw | 0 | 0 | 5 | 01 | 0 | 1 | 0 | 0 | 0 |
| add | 1 | 0 | 5 | 01 | 0 | 0 | 0 | 0 | 0 |
| addu | 1 | 0 | 5 | 01 | 0 | 0 | 0 | 0 | 0 |
| and | 1 | 0 | 7 | 01 | 0 | 0 | 0 | 0 | 0 |
| nor | 1 | 0 | 10 | 01 | 0 | 0 | 0 | 0 | 0 |
| or | 1 | 0 | 8 | 01 | 0 | 0 | 0 | 0 | 0 |
| sll | 1 | 1 | 0 | 01 | 0 | 0 | 0 | 0 | 0 |
| slt | 1 | 0 | 11 | 01 | 0 | 0 | 0 | 0 | 0 |
| sltu | 1 | 0 | 12 | 01 | 0 | 0 | 0 | 0 | 0 |
| sra | 1 | 1 | 1 | 01 | 0 | 0 | 0 | 0 | 0 |
| srav | 1 | 1 | 1 | 01 | 0 | 0 | 0 | 0 | 0 |
| srl | 1 | 1 | 2 | 01 | 0 | 0 | 0 | 0 | 0 |
| sub | 1 | 0 | 6 | 01 | 0 | 0 | 0 | 0 | 0 |
| subu | 1 | 0 | 6 | 01 | 0 | 0 | 0 | 0 | 0 |
| xor | 1 | 0 | 9 | 01 | 0 | 0 | 0 | 0 | 0 |
| addi | 0 | 0 | 5 | 01 | 0 | 1 | 0 | 0 | 0 |
| addiu | 0 | 0 | 5 | 01 | 0 | 1 | 0 | 0 | 0 |
| andi | 0 | 0 | 7 | 01 | 0 | 1 | 0 | 0 | 0 |
| ori | 0 | 0 | 8 | 01 | 0 | 1 | 0 | 0 | 0 |
| slti | 0 | 0 | 11 | 01 | 0 | 1 | 0 | 0 | 0 |
| sltiu | 0 | 0 | 12 | 01 | 0 | 1 | 0 | 0 | 0 |
| xori | 0 | 0 | 9 | 01 | 0 | 1 | 0 | 0 | 0 |
| beq | 1 | 0 | 5 | 01 | 0 | 0 | 1 | 0 | 0 |
| bgez | 1 | 0 | 11 | 01 | 0 | 0 | 0 | 0 | 1 |
| bne | 1 | 0 | 5 | 01 | 0 | 0 | 0 | 1 | 0 |
| j | 0 | 0 | 5 | 00 | 0 | 0 | 0 | 0 | 0 |
| jal | 0 | 0 | 5 | 00 | 0 | 0 | 0 | 0 | 0 |
| jr | 1 | 0 | 5 | 11 | 0 | 0 | 0 | 0 | 0 |
| eret | 0 | 0 | 5 | 01 | 0 | 0 | 0 | 0 | 0 |
| mfc0 | 0 | 0 | 5 | 01 | 0 | 0 | 0 | 0 | 0 |
| mtc0 | 0 | 0 | 5 | 01 | 0 | 0 | 0 | 0 | 0 |
| syscall | 1 | 0 | 5 | 01 | 1 | 0 | 0 | 0 | 0 |
| 指令 | RegDst | RegWrite | MemWrite | MemRead | toReg | Unsigned | srav | lh |  |
| lh | 00 | 1 | 0 | 1 | 01 | 0 | 0 | 1 |  |
| lw | 00 | 1 | 0 | 1 | 01 | 0 | 0 | 0 |  |
| sw | 00 | 0 | 1 | 0 | 00 | 0 | 0 | 0 |  |
| add | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| addu | 01 | 1 | 0 | 0 | 00 | 1 | 0 | 0 |  |
| and | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| nor | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| or | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| sll | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| slt | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| sltu | 01 | 1 | 0 | 0 | 00 | 1 | 0 | 0 |  |
| sra | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| srav | 01 | 1 | 0 | 0 | 00 | 0 | 1 | 0 |  |
| srl | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| sub | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| subu | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| xor | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| addi | 00 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| addiu | 00 | 1 | 0 | 0 | 00 | 1 | 0 | 0 |  |
| andi | 00 | 1 | 0 | 0 | 00 | 1 | 0 | 0 |  |
| ori | 00 | 1 | 0 | 0 | 00 | 1 | 0 | 0 |  |
| slti | 00 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| sltiu | 00 | 1 | 0 | 0 | 00 | 1 | 0 | 0 |  |
| xori | 00 | 1 | 0 | 0 | 00 | 1 | 0 | 0 |  |
| beq | 01 | 0 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| bgez | 01 | 0 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| bne | 01 | 0 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| j | 01 | 0 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| jal | 11 | 1 | 0 | 0 | 10 | 0 | 0 | 0 |  |
| jr | 01 | 0 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| eret | 01 | 0 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| mfc0 | 01 | 1 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| mtc0 | 01 | 0 | 0 | 0 | 00 | 0 | 0 | 0 |  |
| syscall | 01 | 0 | 0 | 0 | 00 | 0 | 0 | 0 |  |

1. FPGA实现

根据在Logism实现中得到的各个一位指令信号的表达式，直接使用数据流建模，使用assign分的Verilog代码过于冗长，故只取对于指令信号i1即add指令信号的生成代码举例如下：

assign i1 = ~a11& ~a10& ~a9& ~a8& ~a7& ~a6& a5& ~a4& ~a3& ~a2 &~a1& ~a0;

然后根据生成的指令信号产生对应的控制信号，同样这部分的代码过于冗长，只截取一部分示意。

assign Unsigned = xori | addiu | addu | andi | ori | sltu;

## 中断机制实现

### 总体实现

总体上中断的流程为，中断源产生中断信号，各个中断源处理子模块以及中断仲裁之后产生中断发生信号trap，CPU依据trap信号指令跳转到髙地址(0x8000000)处运行，此处存储有中断处理程序，同时由硬件将中断处下一指令地址传送至协处理器的0号寄存器处。中断实现硬件方面主要依靠两部分，一部分是协处理器，用来存储中断发生地址epc、中断号以及中断屏蔽字；另一部分是中断产生电路，包括对应每个中断源的中断处理子模块以及后续的中断仲裁子模块。如图 3‑5所示。

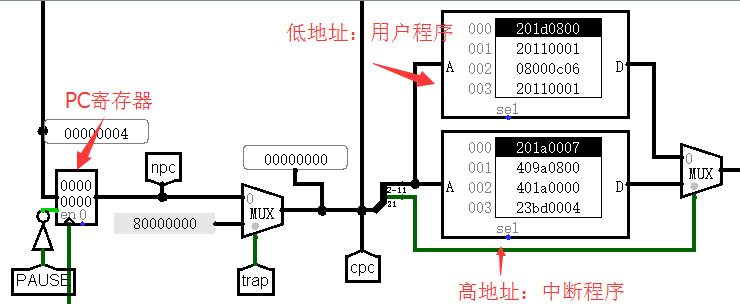


图 3‑5 中断实现——PC的实现

中断程序运行最后执行eret指令，通过一个二路选择器读出协处理器0号寄存器中断的EPC作为下一指令的地址。如图 3‑6所示。

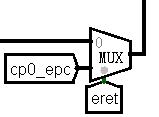


图 3‑6 中断实现——PC选择

下面是每个分模块的具体实现。

### 中断处理子模块

对于每一个按键产生的中断，将中断信号作为一D触发器的时中断，使得中断信号一来临，触发器的值变为1表示有中断产生，和之前中断锁存的信号相或产生中断灯信号，该灯信号和中断关闭信号的非相与作为一锁存器的输入，锁存的输出与中断屏蔽位的非相与产生最终的中断排队信号。logisim示意如图 3‑7所示。

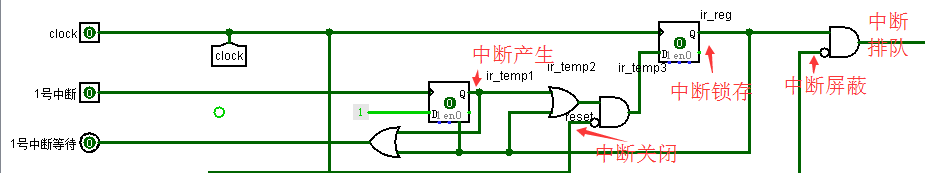


图 3‑7 中断处理子模块图

其实现部分代码如下：

always @(posedge ir, posedge ir\_reg\_wire) begin

if(ir\_reg\_wire) begin

ir\_out <= 0;

end

else begin

ir\_out <= 1;

end;

end

assign ir\_temp1 = ir\_out;

assign ir\_temp2 = ir\_temp1 | ir\_reg;

assign ir\_temp3 = ir\_temp2 & ~reset;

always @(posedge clk) begin

ir\_reg <= ir\_temp3;

end

assign ir\_reg\_wire = ir\_reg;

assign led = ir\_temp1 | ir\_reg\_wire;

assign send = ir\_reg\_wire & ~imw;

### 中断仲裁子模块

通过中断处理子模块产生的各个中断源对应的中断排队信号经过优先选择器后输出优先级最高的中断信号，未选中的信号会被锁存在上图中断第二个寄存器，等待当前中断程序处理完毕之后再处理。由于没有中断信号产生时优先选择器的输出为错误值，故此对输出做一个二路选择，当没有中断信号时，中断号选择0代表当前没有中断产生。中断号经过零扩展成32位之后作为协处理器2号寄存器的输入，等待时钟下降沿来临可以写进该寄存器。图中红色框中的处理的目的是控制总的中断产生信号trap只有一个时钟周期，作为CPU进入中断的标志。如图 3‑8所示。该部分代码略。

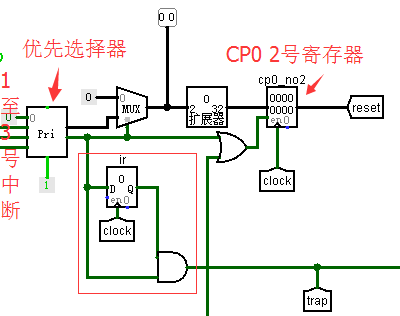


图 3‑8 中断仲裁电路图

能够写该寄存器的情况有两种，一是当中断来临，优先选择器的中断是否存在的信号会变1时；二是中断程序执行到最后一句eret指令，允许该寄存器的值发生改变，达成返回用户程序或者执行在排队电路中等待的下一中断。将上述两个信号相或接入该寄存器的使能端即可。

另外还需要通过协处理器2号寄存器的值，也就是当前正在处理的中断号，来产生中断重置信号，如图 3‑9所示，通过一个解复用器来产生各个中断处理子模块中的中断关闭信号。

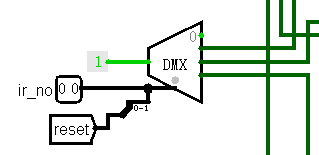


图 3‑9中断关闭电路

其对应代码如下：

always @ (ir\_no) begin

if(ir\_no == 3) begin

set = 3'b100;

end

else if( ir\_no == 2) begin

set = 3'b010;

end

else if( ir\_no == 1) begin

set = 3'b001;

end

else begin

set = 3'b0;

end

end

### 协处理器模块

实现中断的另一个关键就是构造一个协处理器，其本质是一组寄存器并包括其读写的机制。本实验只用了简易的协处理器，用到三个寄存器，功能分别如下：

1. 0号寄存器用于存放中断返回的地址EPC；
2. 1号寄存器用于屏蔽字，因此本实验控制屏蔽字主要通过软件来实现；
3. 2号寄存器用于存放当前正在运行的中断处理程序的中断号。

该模块提供了两个读寄存器的接口以及一个写寄存器的接口，另外对每个寄存器都有单独的直接读接口，方便硬件上的功能实现。如图 3‑10所示。该部分代码略。

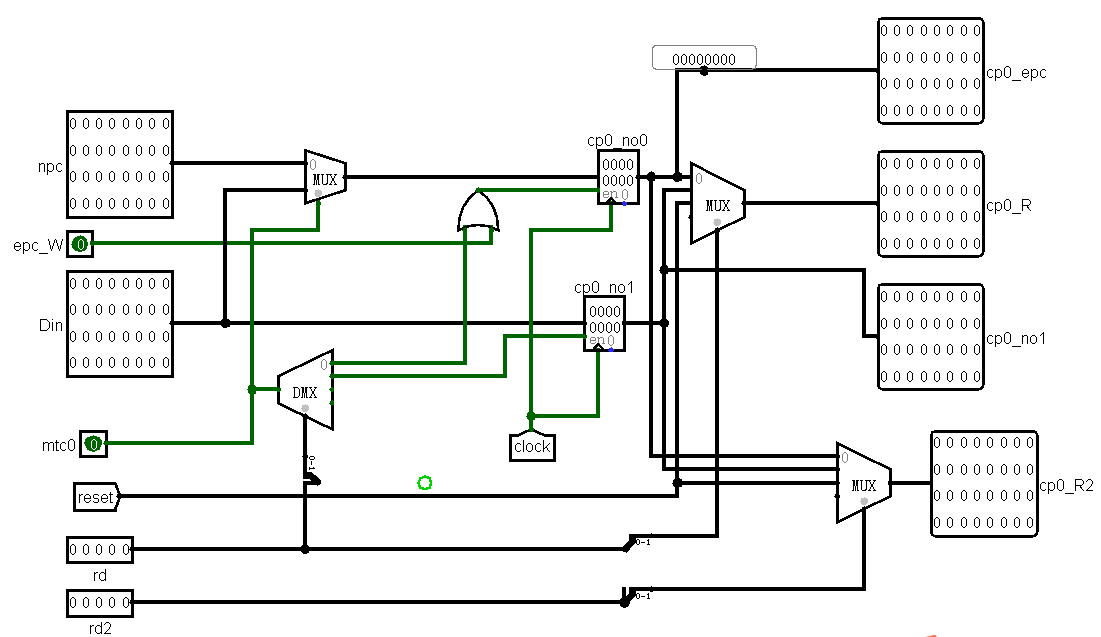


图 3‑10 协处理器模块电路图

### 软件部分实现

软件层次，中断进入后首先需要关中断，这里通过设置协处理器中的屏蔽字寄存器实现。关中断时，屏蔽字需要设置为111。用mtc0指令对1号协处理器寄存器写入7即可。代码如下。

addi $k0, $0, 7 #cli

mtc0 $k0, $1

下面将当前epc压栈，并将中断程序中用到的寄存器进行压栈。由于mips指令没有直接的压栈指令，这里通过存储指令sw存储和用$sp-4，分两步实现压栈。首先通过指令mfc0读出epc的值，后压栈。代码如下：

#store epc

mfc0 $k0, $0

addi $sp, $sp, 4

sw $k0, 0($sp)

接着开中断，将屏蔽字设置为000，原理和关中断相同。而后读取2号CP0寄存器读取中断号，根据中断号判断进入哪一个中断处理程序。对应代码如下。

mfc0 $s0, $2

addi $s1, $0, 1

beq $s0, $s1, handler1

addi $s1, $0, 2

beq $s0, $s1, handler2

addi $s1, $0, 3

beq $s0, $s1, handler3

进入中断处理程序后，设置屏蔽字实现中断的响应优先级。对3号中断，屏蔽字应设置为111；对2号中断，屏蔽字应设置为011；对1号中断，屏蔽字应设置为001；否则屏蔽字应设置为00。中断程序结束后恢复寄存器，并将epc从栈中压回CP0的0号寄存器，此阶段开始前需要关中断，结束后需要开中断。最后执行eret指令，返回用户程序。

## 流水CPU实现

单周期CPU在一个周期内就执行完一条指令，下面将指令的执行分为五个阶段，依次为取指令IF、指令译码ID、执行运算EX、访存阶段MEM、写回阶段WB。每个阶段为一个时钟周期，多条指令可以同时在CPU上运行。每两个阶段之间用一个流水寄存器缓冲锁存各指令执行至当前段时的状态。

### 流水接口部件实现

针对理想流水设计给出的各个流水寄存器需要锁存的信号，设计每个流水寄存器。流水寄存器需要有以下几个功能。

1. 锁存指令执行的状态，即伴随指令执行到特定阶段的各个信号。具体通过寄存器即可实现。针对某些位数比较少的信号这里可以通过只用一个寄存器实现，如ID/EX流水寄存器中需要锁存较多的控制信号，可以综合用一个寄存器传递。如图 3‑11所示。

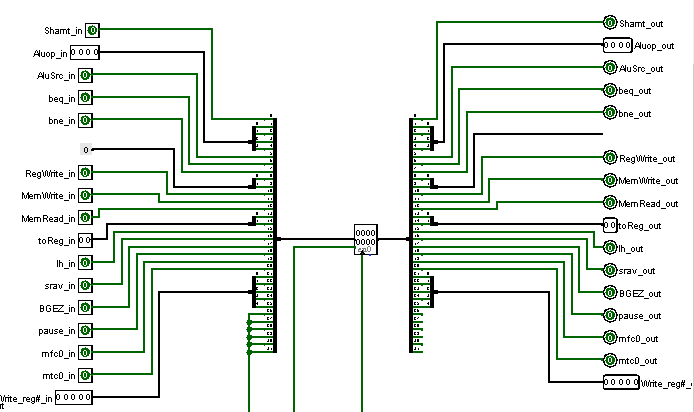


图 3‑11 流水寄存器部分电路图

1. 需要提供暂停功能，讲模块统一输入的暂停信号作为各个寄存器的使能端即可。
2. 传递气泡功能，此功能用于后续的气泡流水线中，具体实现为为每个寄存器的输入增加一个二路选择器，当nop信号来临所有寄存器的输入变为0即可。

### 理想流水线实现

理想流水线的实现方式即将指令的执行分为取指令IF、指令译码ID、执行运算EX、访存阶段MEM、写回阶段WB五个阶段。将原先的单周期CPU电路按照五个阶段将各个部分分别调整到合适位置，并在每两个段之间插入相应的流水线寄存器。每个阶段先从上一流水线寄存器中读出前几个阶段产生的控制信号以及相关信息，并将指令之后还需要使用的信号作为输入送至下一流水寄存器中。由于理想流水线不需要考虑数据相关以及分支相关，有关跳转指令部分这里未做处理，只做了PC+4的正常指令执行过程。如图 3‑12所示。

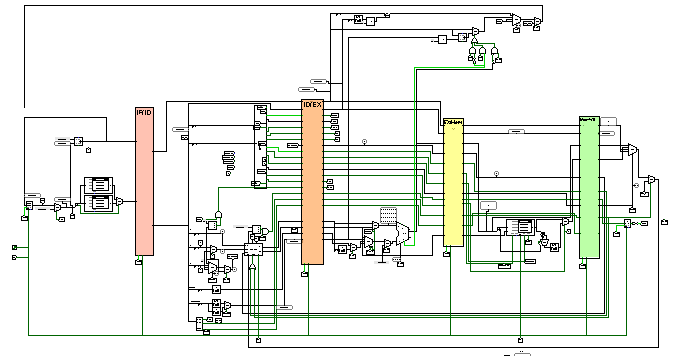


图 3‑12 理想流水线实现电路图

各个段需要给下一个段传递信息，以使得指令的状态能够传递。各个段需要传递的信息如下。

1. IF->IF/ID：  
   下一指令PC即NPC、指令IR
2. ID->ID/EX：  
   下一指令PC即NPC、指令IR、R1、R2、扩展后的偏移值shamt、R2、扩展后的立即数immediate、偏移信号isShamt、ALU的op信号AluOp、ALU操作数选择信号AluSrc、beq信号、bne信号、写寄存器信号RegWrite、写内存信号MemWrite、读内存信号MemRead、写寄存器数据选择信号toReg、lh信号、srav信号、BGEZ信号、停止信号pause、mfc0信号、mtc0信号、写寄存器的编号Write\_reg。
3. EX->EX/Mem：  
   ALU运算结果AluResult、下一指令PC即NPC、指令IR、写寄存器信号RegWrite、写内存信号MemWrite、读内存信号MemRead、写寄存器数据选择信号toReg、lh信号、srav信号、BGEZ信号、停止信号pause、mfc0信号、mtc0信号、写寄存器的编号Write\_reg。
4. Mem->Mem/WB：  
   ALU运算结果AluResult、内存读取数据MemData、下一指令PC即NPC、指令IR、写寄存器信号RegWrite、写寄存器数据选择信号toReg、停止信号pause、mfc0信号、mtc0信号、写寄存器的编号Write\_reg。

## 气泡式流水线实现

### 气泡流水线整体实现

由于理想流水线中没有考虑指令中可能出现的数据相关以及分支相关，下面需要对理想流水线做出修改以使其能解决这些冲突。解决数据先关的原理在设计中已经阐明，下面描述具体实现方法。

首先规定几个信号的来源及其作用：

1. nop信号，存在数据相关时为1，给予指令计数器PC和IF/ID流水线寄存器使其暂停流水线传送，并控制ID/EX流水线寄存器传递一个空指令，即给EX段传递气泡。
2. nop\_IF\_ID信号，由分支跳转指令共同产生，专门控制IF/ID流水线寄存器产生气泡。当为1时说明需要跳转，控制IF/ID流水线寄存器传递一个空指令，即给ID段传递气泡。
3. nop\_ID\_EX信号，由分支指令产生，专门控制ID/EX流水线寄存器传递一个气泡，当为1时说明需要分支跳转，控制ID/EX流水线寄存器传递一个空指令，即给EX段传递气泡。

根据上述功能描述，对CPU整体做出修改，针对指令计数器PC，使能端增加一个nop，使得nop信号为1时指令计数器停止计数。如图 3‑13所示。

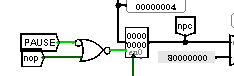


图 3‑13 气泡流水——PC计数器实现

针对IF/ID流水线寄存器，其暂停输入端需要或上一个nop信号，当nop为1时该流水寄存器能够停止传送。另外在流水线寄存器nop端需要输入nop\_IF\_ID信号，当nop\_IF\_ID为1时流水线寄存器下一时钟上升沿输出为全零，实现传递气泡。如图 3‑14所示

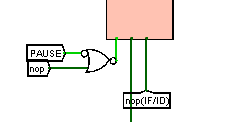


图 3‑14 气泡流水——IF/ID接口示意图

针对ID/EX流水线寄存器，在流水线寄存器nop端需要输入nop\_IF\_ID与nop信号相或的结果，当nop\_IF\_ID或nop为1时都要给EX段传递气泡，即流水线寄存器下一时钟上升沿输出为全零，实现传递气泡。如图 3‑15所示。

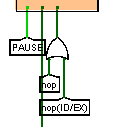


图 3‑15 气泡流水——ID/EX接口示意图

下面是对数据相关和分支相关分别作出处理，以产生上述信号。

### 数据相关处理

数据相关需要在ID段判断当前指令所用的寄存器是否满足以下条件：

1. ID段当前指令确定要读某个寄存器；
2. 该寄存器号与在EX段或Mem段或WB段的状态中要写的寄存器号WriteReg相等；
3. 满足条件2的同时相应段的写寄存器信号RegWrite为1；

根据上述3个条件首先对ID段的rt进行判断，分别判断rt是否与EX段、Mem段和WB段的写寄存器的编号Write\_reg是否相等，若相等且rt不是0（排除空指令的干扰），并且相应段的写使能信号RegWrite为1，则rt与相应段有数据相关，三个段的判断结果相或的结果说明rt与三个段是否存在数据相关。下面需要判断ID段是否需要使用了rt，因为对于不使用rt的指令，rt也会被译码产生从而可能表面上看起来可能和后续三个段出现数据相关，但这种情况我们认为没有发生相关。实现方法即将三个段的判断结果与上使用rt的信号rt\_use，该信号由所有ID段需要用到rt的指令信号相或产生。

根据表 3‑3可以得到，用到rt的指令可以用ID段的下面几个信号得到：beq、bne以及除syscal以为的R型指令，将这几个信号相或就可以得到rt\_use信号。

表 3‑3 指令与寄存器关系表

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 类型 | 指令 | op | rs（base） | rt | rd | shamt | func | offset/immediate | index |
| 访存 | lh | 100001 | √ | √ |  |  |  | √ |  |
| lw | 100011 | √ | √ |  |  |  | √ |  |
| sw | 101011 | √ | √ |  |  |  | √ |  |
| R-R | add | 0 | √ | √ | √ | 0 | 100000 |  |  |
| addu | 0 | √ | √ | √ | 0 | 100001 |  |  |
| and | 0 | √ | √ | √ | 0 | 100100 |  |  |
| nor | 0 | √ | √ | √ | 0 | 100111 |  |  |
| or | 0 | √ | √ | √ | 0 | 100101 |  |  |
| sll | 0 | 0 | √ | √ | √ | 0 |  |  |
| slt | 0 | 0 | √ | √ | √ | 101010 |  |  |
| sltu | 0 | 0 | √ | √ | √ | 101011 |  |  |
| sra | 0 | 0 | √ | √ | √ | 11 |  |  |
| srav | 0 | √ | √ | √ | 0 | 111 |  |  |
| srl | 0 | 0 | √ | √ | √ | 10 |  |  |
| sub | 0 | √ | √ | √ | 0 | 100010 |  |  |
| subu | 0 | √ | √ | √ | 0 | 100011 |  |  |
| xor | 0 | √ | √ | √ | 0 | 100110 |  |  |
| R-I | addi | 1000 | √ | √ |  |  |  | √ |  |
| addiu | 1001 | √ | √ |  |  |  | √ |  |
| andi | 1100 | √ | √ |  |  |  | √ |  |
| ori | 1101 | √ | √ |  |  |  | √ |  |
| slti | 1010 | √ | √ |  |  |  | √ |  |
| sltiu | 1011 | √ | √ |  |  |  | √ |  |
| xori | 1110 | √ | √ |  |  |  | √ |  |
| 分支 | beq | 100 | √ | √ |  |  |  | √ |  |
| bgez | 1 | √ | 1 |  |  |  | √ |  |
| bne | 101 | √ | √ |  |  |  | √ |  |
| 跳转 | j |  |  |  |  |  |  |  | √ |
| jal | 11 |  |  |  |  |  |  | √ |
| jr | 0 | √ | 0 | 0 | 0 | 1000 |  |  |
| CP0 | eret | 10000 | 10000 | 0 | 0 | 0 | 11000 |  |  |
| mfc0 | 10000 | 0 | √ | √ | 0 | 0 |  |  |
| mtc0 | 10000 | 100 | √ | √ | 0 | 0 |  |  |
| 系统 | syscall | 0 |  |  |  |  | 1100 |  |  |

同理，对rs数据相关的处理和对rt数据相关的处理方式类似，只不过对使用rs的指令的判断有所不同，查表得到，用到rs的指令可以由ID段下面几个信号得到：lh、lw、I型指令、beq、bne、jr指令以及除syscall以外的R型指令，以上信号相或即可得到rs\_use信号。

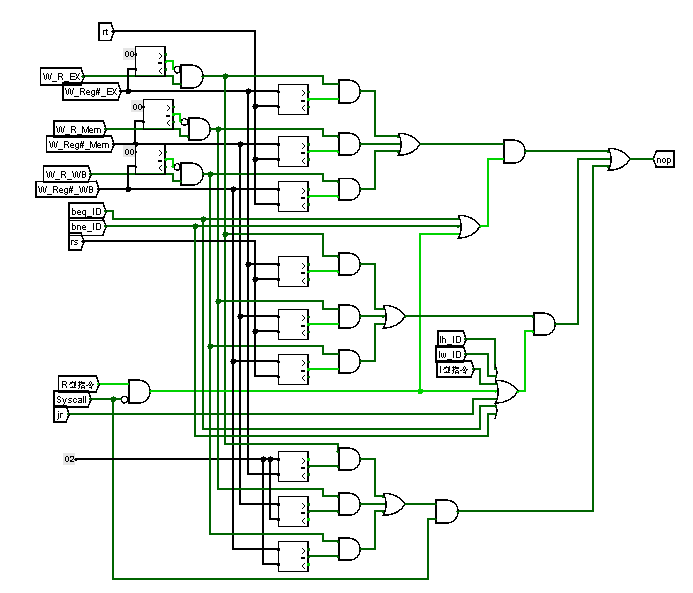


图 3‑16 数据相关产生气泡模块电路图

除此之外还需要判断特殊指令是否产生数据相关。这里判断了syscall指令。由于syscall指令需要判断$v0寄存器的值，并显示$a0的值，故需要用上述判断rt、rs数据相关的方法来判断$v0和$a0寄存器是否数据相关，区别在于上述的rt\_use此处用syscall信号替代。此处根据汇编代码中每次调用syscall之前都会先对$a0赋值，再对$v0进行判断，所以此处只需要判断$v0是否发生冲突就能同时判断出$a0的数据相关。

最后将上述三种判断的结果相或，得到最后的nop信号，即最终的是否存在数据相关的信号。其总体电路如图 3‑16所示。

### 分支相关处理

分支相关发生的条件是分支指令成功跳转和直接跳转指令来临时，转移指令使得流水线发生中断，提前取出的指令作废，流水线需要清空，PC值设置为跳转地址再继续运行。

针对分支指令，在EX段才能判断是否跳转，故在EX段对分支指令进行判断，若成功跳转，则需要分别在ID段和EX段插入气泡，即nop\_IF\_ID信号和nop\_ID\_EX信号置为1。

针对直接跳转指令，因为在译码段就可以知道需要跳转，故在译码段处理直接跳转指令可以少插入一个气泡。当译码段判断为是直接跳转指令时，nop\_IF\_ID信号置为1，给ID段插入一个气泡。

如图 3‑17所示，branch是bne、beq以及扩展指令bgez在EX段最后判断出来的是否跳转信号，1代表成功跳转。Jmp信号值为00时代表j跳转或jal跳转，为11时代表jr跳转，默认不会出现10情况，故只需要判断Jmp不等于01即为直接跳转指令。

此处将eret也看作是一条直接跳转指令，因为eret也会导致指令跳转。

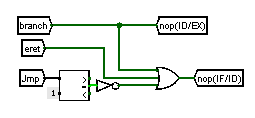


图 3‑17 分至相关气泡信号产生

由于理想流水线没有对分支跳转做处理，这里还需要处理非顺序的NPC判断过程。如图 3‑18所示，在IF取值阶段，通过选择器选出NPC地址，其中除了PC+4，其余信号都是后续段的反馈信号。这里需要注意的是分支跳转branch和直接跳转的优先级。由于本人采用的是分支和直接跳转分开处理，故可能存在EX段分支成功branch为1且ID段时直接跳转指令，此时需要保证下一指令地址是分支指令的跳转地址，因为EX段的指令先执行。

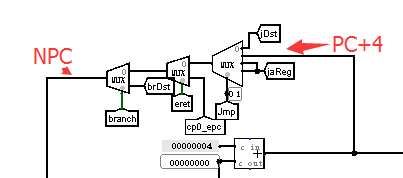


图 3‑18 目标地址选择模块

## 数据转发流水线实现

### 重定向整体实现

数据相关可以通过插入气泡的方式解决，但是该方法会造成程序周期数大大增加，造成资源浪费。下面用重定向的方法解决部分数据相关。

重定向主要由两部分实现，一是重定向的判断子模块，产生CPU重定向需要的信号；二是重定向处理子模块，依据判断子模块产生的重定向信号修正ID段从寄存器读出的值以及数码管显示模块的$a0和$v0的值。下面列举了各种情况重定向的处理方法。

1. ID段和EX段的寄存器发生冲突，而EX段的指令不是load指令或者mfc0指令，则直接将EX段ALU的运算结果作为ID段的对应寄存器的值；
2. ID段和EX段的寄存器发生冲突，而EX段的指令是load指令或者mfc0指令，则需要用插入气泡的方法解决Load\_Use冲突，其解决方法同气泡流水线。
3. ID段和Mem段的寄存器发生冲突，而EX段的指令不是load指令或者mfc0指令，则直接将Mem段ALU的运算结果作为ID段的对应寄存器的值；
4. ID段和EX段的寄存器发生冲突，而EX段的指令不是load指令，则直接将Mem段内存读出数据MemDate作为ID段的对应寄存器的值；
5. ID段和EX段的寄存器发生冲突，而EX段的指令不是mfc0指令，则直接将对应协处理器的寄存器的值作为ID段的对应寄存器的值；
6. 针对特殊指令如syscall等，直接判断EX段和Mem段是对a0或v0寄存器进行写操作，若使用了则直接将相应段的ALU计算结果作为相应寄存器的值进行处理。

下面详细介绍重定向判断子模块和重定向处理子模块的实现。

### 重定向判断子模块

重定向判断子模块电路在气泡流水线中数据相关判断子模块的基础上改进得到。针对上述各种情况产生信号方法如下：

1. 若ID段的rt与EX段的写寄存器编号数据相关，判断方法同气泡流水线中数据相关的判断方法，这里对rt\_use作出改进，将ID段的bne、beq和MemWrite信号、以及mtc0信号相或得到rt\_use。此时若EX段不是Load类型指令或者mfc0指令，则用EX段的AluResult修正ID段的rt对应寄存器的读取值，将exRedT信号置为1；否则，若EX段是Load类型指令或者mfc0指令，则无法用重定向的方法解决，故仍用插入气泡的方法解决，能够使nop信号变为1。
2. 若ID段的rt与Mem段的写寄存器编号数据相关。此时若Mem段不是Load类型指令或者mfc0指令，则用Mem段的AluResult修正ID段的rt对应寄存器的读取值，将redAluT信号置为1；否则，若Mem段是Load类型指令，则用Mem段的RAM读取值MemData修正ID段的rt对应寄存器的读取值，将redLoadT置为1；否则，若Mem段是mfc0指令，同样不能用重定向解决，故仍用插入气泡的方法解决，能够使nop信号变为1。
3. 针对rs段的判断方法同针对rt段的判断方法类似，得到exRedS信号、redLoadS信号、redAluS信号，同时也对nop信号产生作出修改。

针对syscall指令，需要对$a0和$v0进行重定向判断。此处不能像气泡流水线一样省略对$a0的判断，因为在气泡流水线中，对$v0的数据相关解决中插入气泡时其实已经解决了$a0的数据相关，但在重定向中，对$v0的重定向不能解决$a0的数据相关，故此处需要对$a0单独进行重定向判断。对于$v0,若在EX段对$v0进行了写，则用EX段AluResult对$v0进行修正，将信号exRedv0置为1；若Mem段对$v0进行了写，则用Mem段AluResult对$v0进行修正，将信号redAluv0置为1。同理，判断$a0的数据相关情况产生信号exReda0和redAlua0信号。

重定向判断的电路如图 3‑19所示。

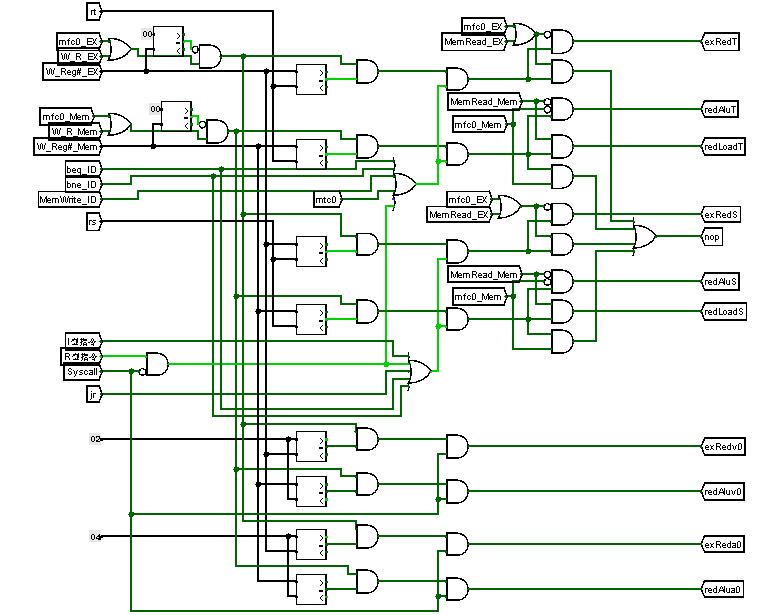


图 3‑19 重定向信号产生模块

其对应部分代码如下：

Equal\_5 equal3(W\_Reg\_EX, 5'b0, W\_Reg\_temp\_EX); //'

assign W\_EX = ~W\_Reg\_temp\_EX & (W\_R\_EX | mfc0\_EX);

Equal\_5 equal4(W\_Reg\_EX, rt, rt\_equal\_EX);

assign rt\_1 = W\_EX & rt\_equal\_EX & rt\_use;

Equal\_5 equal5(W\_Reg\_Mem, 5'b0, W\_Reg\_temp\_Mem); //'

assign W\_Mem = ~W\_Reg\_temp\_Mem & (W\_R\_Mem | mfc0\_Mem);

Equal\_5 equal6(W\_Reg\_Mem, rt, rt\_equal\_Mem);

assign rt\_2 = W\_Mem & rt\_equal\_Mem & rt\_use;

assign exRedT = ~(MemRead\_EX | mfc0\_EX) & rt\_1;

assign redAluT = ~MemRead\_Mem & ~mfc0\_Mem & rt\_2;

assign redLoadT = MemRead\_Mem & rt\_2;

### 重定向处理子模块

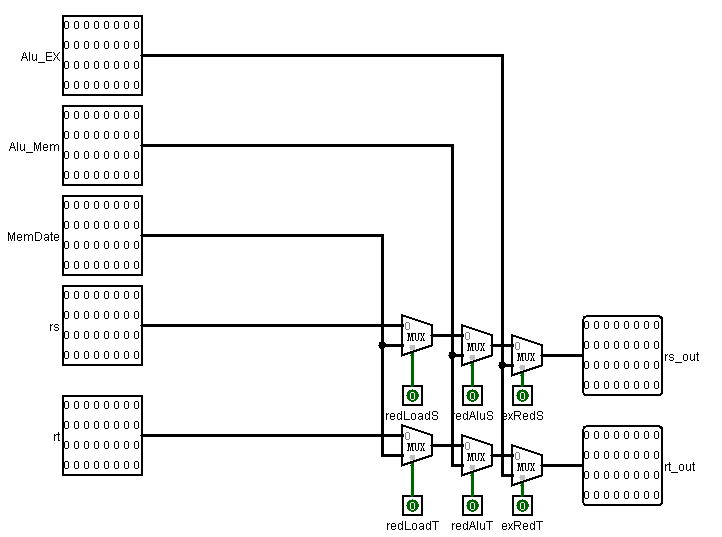


图 3‑20 重定向处理模块

依据重定向判断子模块产生的信号，修正寄存器读出值R1和R2，其中R1对应rs，R2对应rt。重定向处理子模块内部电路如图 3‑20所示。将EX段的ALU计算结果Alu\_EX、Mem段的ALU计算结果Alu\_Mem以及Mem段RAM读出值MemData作为输入，exRedT、redLoadT、redAluT、exRedS、redLoadS、redAluS作为选择信号选择最后修正的寄存器读出值。这里需要注意各个信号的优先级。比如对于rt，由于EX段对ID段时最近的指令，rt的值应该等于EX段时rt的值，所以exRedT信号的优先级要高于另外两个信号。

其部分代码如下：

mux2\_1width\_32 mux1(redLoadS, rs, MemDate, x1);

mux2\_1width\_32 mux2(redAluS, x1, Alu\_Mem, x2);

mux2\_1width\_32 mux3(exRedS, x2, Alu\_EX, rs\_out);

mux2\_1width\_32 mux4(redLoadT, rt, MemDate, y1);

mux2\_1width\_32 mux5(redAluT, y1, Alu\_Mem, y2);

mux2\_1width\_32 mux6(exRedT, y2, Alu\_EX, rt\_out);

在主模块中调用重定向处理子模块，结构如图所示。修正后的R1和R2送入ID/EX流水线寄存器，从而下一节拍传到EX段的R1和R2值就已经是修正后的值了。重定向处理模块接口如图 3‑21所示。代码略。

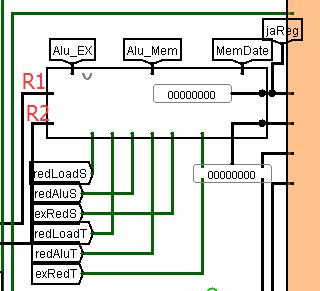


图 3‑21 重定向处理模块接口

如图 3‑22所示，实现对$v0的重定向处理，用重定向信号exRedv0和redAluv0修正$v0的值，同样这里也有优先级，EX段产生的exRedv0优先级要高于redAluv0。代码略。

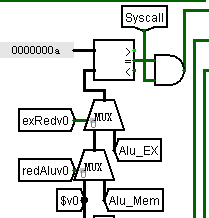


图 3‑22 对$v0的重定向处理

如图 3‑23所示，是显示模块中对$a0的重定向处理，用重定向信号exReda0和redAlua0修正$a0的值，同样这里也有优先级，EX段产生的exReda0优先级要高于redAlua0。

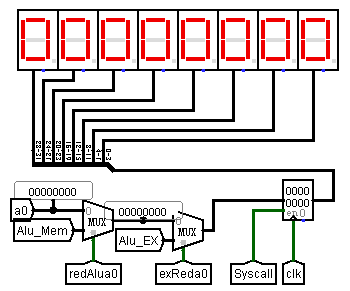


图 3‑23 数码管显示/对$a0重定向处理

对$a0重定向部分代码如下：

mux2\_1width\_32 mux17(redAlua0, a0, Alu\_Mem, led\_temp1);

mux2\_1width\_32 mux18(exReda0, led\_temp1, Alu\_EX, led\_temp2);

register\_32 reg3(.date(led\_temp2), .clk(clk), .en(Syscall), .clr(clear), .out(led\_temp3));

## FPGA流水线实现

在logisim上完成重定向流水线之后，将其用verilog语言在vivado上实现。此处针对指令存储器ROM和数据存储器RAM由vivado自带的IP核实现，其他模块都

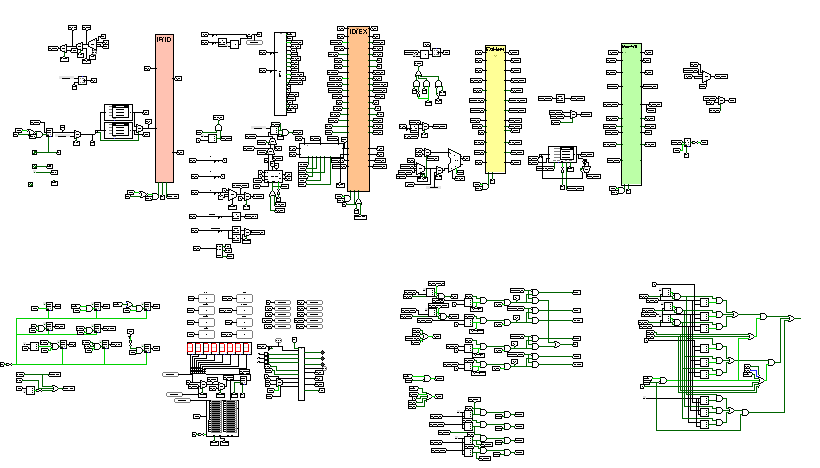


图 3‑24 隧道版电路图

由自己编写。部分verilog代码见各模块介绍。此处用到一个技巧是将电路图里面的连线全部用隧道表示，提前为每一个信号命名，如图 3‑24所示。所谓的磨刀不误砍柴工，虽然花了较长时间将原电路图转化成全用隧道实现的电路图，但同坐这种方式就不必在写代码的时候还去思考如何为某个输出或某个输入命名，后续写代码的时候轻松了很多。

## 中断流水线实现

前面已经实现单周期CPU上的中断处理，中断流水大体思路与单周期中断相同，主要改动在设计中已经给出，下面介绍具体实现方式。

### 中断处理模块接口

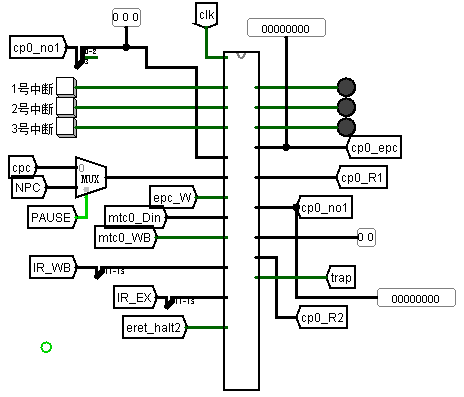


图 3‑25 中断模块接口

中断模块输入如图所示，epc\_W信号接口专门用于中断产生时硬件对0号寄存器的写操作。信号eret\_halt2用于中断返回时控制协处理器2号寄存器的值改变，使其能够返回用户程序或接受正在等待的中断程序。如图 3‑25所示。

### 中断进入

由于采用了译码段打断程序并处理中断的方式，故需要等待当前译码段指令执行结束后再进入中断程序。实现方法如图 3‑26所示，中断信号trap为1时，产生trap\_halt信号控制PC计数器暂停和IF/ID流水寄存器传递气泡。同时计数器开始计数，计数器的输出非零时也能控制trap\_halt信号为1，保证能等到流水线中的指令执行完成。直到计数器产生进位，进位信号即写epc信号epc\_W为1，此时将被打断程序的NPC压入协处理器的0号寄存器中。信号epc\_W作为一寄存器的输入，下一时钟上升沿Rom\_choose信号为1，表示指令跳转到髙地址ROM运行，即开始执行中断程序。Rom\_choose输回计数器的异步清零端，使计数器清零。

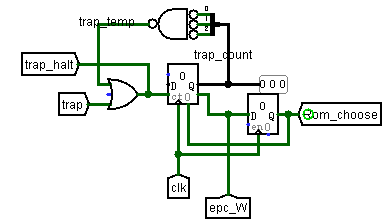


图 3‑26 中断进入模块

其对应部分代码如下：

assign trap\_halt = trap | trap\_temp;

CountUp #(.bound(7)) c\_up1(.clk(clk),

.clr(clear | Rom\_choose),

.start(trap\_halt),

.count(trap\_count\_temp),

.up(epc\_W)

);

assign trap\_count = trap\_count\_temp[2:0];

assign trap\_temp = ~(~trap\_count[0] & ~trap\_count[1] & ~trap\_count[2]);

Register #(.width(1)) reg4(.date(epc\_W), .clk(clk), .en(1), .clr(0),

.out(Rom\_choose)

);

### 中断结束

中断程序结束，执行最后一条指令eret，信号eret为1，为保证中断程序不对之后的程序产生影响，同样需要等待几个时钟周期让流水线的中断程序执行完毕。实现方法如图所示，eret信号为1时，产生eret\_halt信号控制PC计数器暂停和IF流水寄存器传递气泡。同时计数器开始计数，计数器的输出非零时也能控制eret\_halt信号为1，保证能等到流水线中的指令执行完成。直到计数器产生进位，进位信号作为一寄存器的输入，下一时钟上升沿eret\_halt2信号为1，该信号控制中断处理模块中协处理器2号寄存器的值改变，使其能够返回用户程序或接受正在等待的中断程序。信号eret\_halt2输回计数器的异步清零端，使计数器清零。如图 3‑27所示。

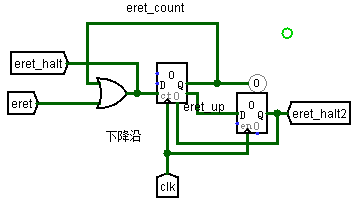


图 3‑27 中断退出模块

其对应部分代码如下：

assign eret\_halt = eret | eret\_count;

CountUp #(.bound(1)) c\_up2(.clk(clk),

.clr(clear | eret\_halt2),

.start(eret\_halt),

.count(eret\_count\_temp),

.up(eret\_up)

);

assign eret\_count = eret\_count\_temp[0];

Register #(.width(1)) reg21(.date(eret\_up), .clk(clk), .en(1), .clr(0),

.out(eret\_halt2)

);

### PC值的产生以及流水寄存器更改

针对上述模块中产生的控制信号，对PC计数器、NPC的判断以及IF/ID流水寄存器做出更改。PC计数器增加两个使能信号trap\_halt和eret\_halt信号，两者其一为1时PC计数器停止指令地址传输。如图 3‑28所示。

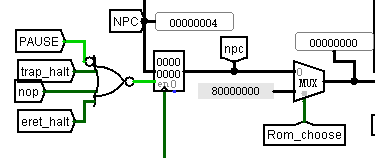


图 3‑28 PC计数器处理

对于NPC的选择，用eret\_halt2信号选择cp0\_epc，即读出协处理器0号寄存器中的epc值。如图 3‑29所示。

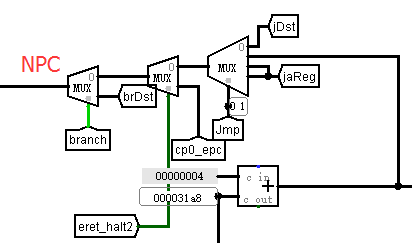


图 ‑ PC地址选择

对于IF/ID流水寄存器，将eret\_halt和trap\_halt信号作为流水寄存器的nop信号输入，代表两者有其一为1时，流水寄存器传递一个气泡。如图 3‑30所示。

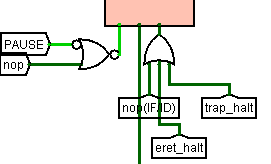


图 3‑30 IF/ID流水寄存器的修改

## 动态分支预测机制实现

实现动态分支预测的原理在设计里已经阐述，该实验没有在logisim上连线，直接编写verilog代码完成。本实验采用双预测位，10和11代预测成功要跳转，00和01代表预测失败不跳转。下面结合具体代码分析实现方法。

### 最久未使用机制实现

这里采用的最久未使用方法使用以下策略。初始化时LRU依次标记为0到7，初始化各个表项的LRU代码如下。

if(clear) begin

for(k = 1; k < 9; k = k+1) begin

BHT\_PC[k] = 0;

BHT\_pre[k] = 0;

BHT\_LRU[k] = k - 1; //'

BranchAddr[k] = 0;

end

end

在EX段若每次查找若与查找PC不相等，则LRU加1，若相等则清零，最后在做替换的时候只要查找LRU为7的表项即可。相应代码如下。

BHT\_LRU[1] = (index == 1) ? 0 : ((BHT\_LRU[1] == 7)? 7 : (BHT\_LRU[1] + 1));

BHT\_LRU[2] = (index == 2) ? 0 : ((BHT\_LRU[2] == 7)? 7 : (BHT\_LRU[2] + 1));

BHT\_LRU[3] = (index == 3) ? 0 : ((BHT\_LRU[3] == 7)? 7 : (BHT\_LRU[3] + 1));

BHT\_LRU[4] = (index == 4) ? 0 : ((BHT\_LRU[4] == 7)? 7 : (BHT\_LRU[4] + 1));

BHT\_LRU[5] = (index == 5) ? 0 : ((BHT\_LRU[5] == 7)? 7 : (BHT\_LRU[5] + 1));

BHT\_LRU[6] = (index == 6) ? 0 : ((BHT\_LRU[6] == 7)? 7 : (BHT\_LRU[6] + 1));

BHT\_LRU[7] = (index == 7) ? 0 : ((BHT\_LRU[7] == 7)? 7 : (BHT\_LRU[7] + 1));

BHT\_LRU[8] = (index == 8) ? 0 : ((BHT\_LRU[8] == 7)? 7 : (BHT\_LRU[8] + 1));

根据上述策略，当在EX段要替换BHT表中某一表项时，只需要查找其中LRU为7且序号最小的表项替换即可。

### IF段预测

在IF取值阶段，首先用当前PC值为关键字在BHT表中做全相联比较查找，index2返回查找结果。Index2初值为0，若查找后非0，说明有相等的PC在表中，则判断双预测位是否为3或2，如是则预测成功，NPC选择信号hit赋为1，候选npc置为相应跳转地址。否则NPC仍旧处理为PC+4。

if(index2) begin

if(BHT\_pre[index2] == 2'd3 || BHT\_pre[index2] == 2'd2) begin

hit = 1;

bht\_pc = BranchAddr[index2];

end

else begin

hit = 0;

end

end

else begin

hit = 0;

end

### EX段修正并维护

在EX执行阶段，同样用当前PC值为关键字在BHT表中做全相联比较，index返回查找结果，初值为0。若返回结果非0则命中，其相应的处理方式用表 3‑4表示。

表 3‑4 预测位与跳转判断关系表

|  |  |  |
| --- | --- | --- |
| 预测位\ branch | 0 | 1 |
| 00 | bad2good = 0  good2bad = 0  预测位不变 | bad2good = 1  good2bad = 0  预测位加1 |
| 01 | bad2good = 0  good2bad = 0  预测位减一 | bad2good = 1  good2bad = 0  预测位加1 |
| 10 | bad2good = 0  good2bad = 1  预测位减一 | bad2good = 0  good2bad = 0  预测位加一 |
| 11 | bad2good = 0  good2bad = 1  预测位减一 | bad2good = 0  good2bad = 0  预测位不变 |

以上操作之后同时LRU标记为0，说明最近使用过该PC。

若index返回0说明没有命中，用上述替换方法找出最久未使用项，用LRU\_no返回，而将当前指令信息插入BUT表，LRU标记为0，预测位初始为2，表示默认下次会跳转。

BHT\_PC[LRU\_no] = pc\_ex\_temp;

BranchAddr[LRU\_no] = brDst;

BHT\_pre[LRU\_no] = 2'd2; //' default success

BHT\_LRU[LRU\_no] = 0;

### NPC产生部分以及流水寄存器修改

上述操作后还要对NPC产生部分作出修改。分别用IF阶段的判断信号hit和EX段的修正信号bad2good、good2bad和hit\_bht选择下一PC地址，此处需要注意各个信号的优先级，EX段产生的信号的优先级要高于IF段的选择信号。

Mux #(.width(32)) mux4(hit, pc\_eret, bht\_pc, pc\_temp1);

Mux #(.width(32)) mux21(bad2good, pc\_temp1, brDst, pc\_temp2);

Mux #(.width(32)) mux22(good2bad, pc\_temp2, PC\_EX, pc\_temp3);

Mux #(.width(32)) mux23(((~hit\_bht)&branch), pc\_temp3, brDst, pc\_temp);

此外还需要对流水寄存器作出修改。由于需要在分支判断错误的时候插入气泡，即式子(~hit\_bht)&branch) | bad2good | good2bad为1时在ID段和EX段插入气泡。用上式可以方便作出修改这里不再赘述。

# 实验过程与调试

## 测试用例和功能测试

### 扩展指令测试

用自己编写的程序测试扩展的四条指令，以下是自行编写的测试程序。首先初始化$a0为0x1234。

.text

addi $a0, $0, 0x1234

addi $v0,$0,34 # system call for LED display

syscall # display

接着测试srav指令，将$s2赋为4，将$a0右移$s2，显示得到0x00000123。如图 4‑1所示。

addi $s2, $0, 4

srav $a0, $a0, $s2

addi $v0,$0,34 # system call for LED display

syscall # display

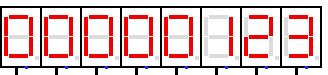


图 4‑1 测试显示00000123

下面测试xori指令，$a0亦或上立即数3，显示得到0x00000120，如图 4‑2所示。

xori $a0, $a0, 3

addi $v0,$0,34 # system call for LED display

syscall # display

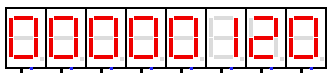


图 4‑2 测试显示00000120

下面测试lh指令，先将0x12341234存储进0($s0)处，其中$s0赋为0，后用lh读出同一地址的半字值，显示得到0x00001234，如图 4‑3所示。

addi $s1, $0, 0x1234

addi $s0, $0, 0

sll $s3, $s1, 16

or $s1, $s1, $s3

add $a0, $0, $s1

addi $v0,$0,34 # system call for LED display

syscall # display

sw $s1, 0($s0)

lh $s2, 0($s0)

add $a0, $0, $s2

addi $v0,$0,34 # system call for LED display

syscall # display

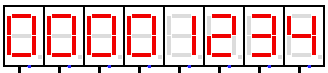


图 4‑3测试显示00001234

下面测试bgez指令，当$s2大于等于0时跳转，会显示0x00000001，否则显示0x00000000，如图所示，显示0x00000001，成功跳转。如图 4‑4所示。

bgez $s2 yes

add $a0, $0, $0

addi $v0,$0,34 # system call for LED display

syscall # display

j out

yes:

addi $a0, $0, 1

addi $v0,$0,34 # system call for LED display

syscall # display

out:

addi $v0,$0,10

syscall # halt

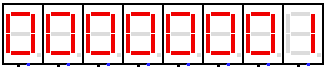


图 4‑4 测试显示00000001

### 嵌套中断测试

载入自行编写的测试程序之后在logisim平台上进行测试。第一步按下1号中断按钮，立即执行1号中断程序，走马灯显示1；此时按下3号中断按钮，1号中断被打断，执行3号中断程序，走马灯显示3；此时按下2号按钮，2号中断等待灯亮，此时如图 4‑5所示。

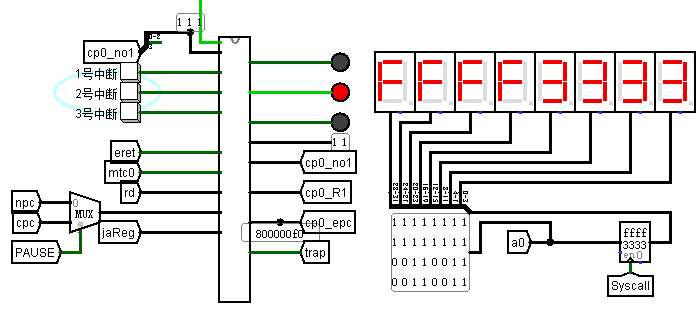


图 4‑5 3号中断执行并按下2号中断键

直到3号中断程序执行结束，执行2号中断程序，走马灯显示2；2号中断结束之后返回1号中断继续执行；1号中断程序执行结束后返回用户程序继续执行。

### 重定向流水线测试

在logisim平台上用benchmar测试程序测试重定向流水线，在benchma首行加入指令“addi $sp, $0, 0x800”，用于初始化$sp，故估统计结果如下：

总周期数 =1546+4+分支误取深度\*分支数+load-Use次数

=1546+4+2\*276+1\*38+120+1=2261

运行结果如图 4‑6所示。由于我采用的策略是分支指令误取深度为2而直接跳转指令的误取深度为1，故最后总指令条数为2298-38+1 = 2261.

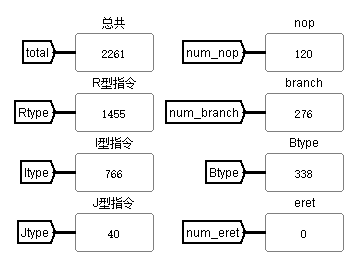


图 4‑6 重定向测试计数结果

### 分支预测测试

在vivado平台上写好verilog代码之后用benchmark测试程序测试，并将程序烧录FPGA板，运行之后的结果显示正确,最终显示00000038，如图 4‑7所示。

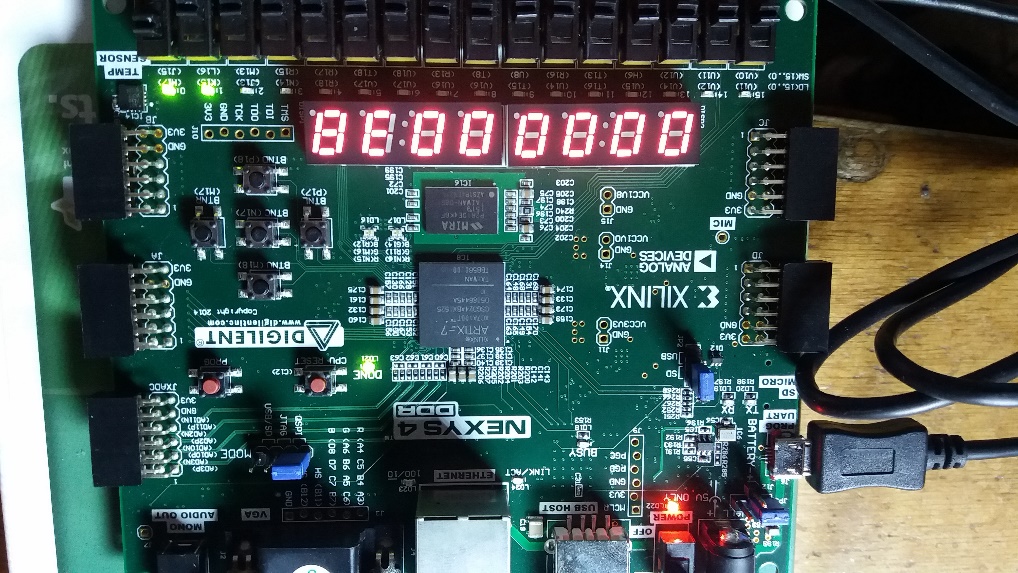


图 4‑7 分支预测运行结果

切换显示，显示总的时钟周期数为000006fd=1789条. 如图 4‑8所示。预测失败次数为40次，如图 4‑9所示。故其预测正确率为1-(40/338)=88.1%.

验证其周期数等式：

总周期数 =1546+4+分支误取深度\*错误分支次数+load-Use次数

=1546+4+2\*40+1\*38+120+1=1789

可以验证试验结果是正确的。

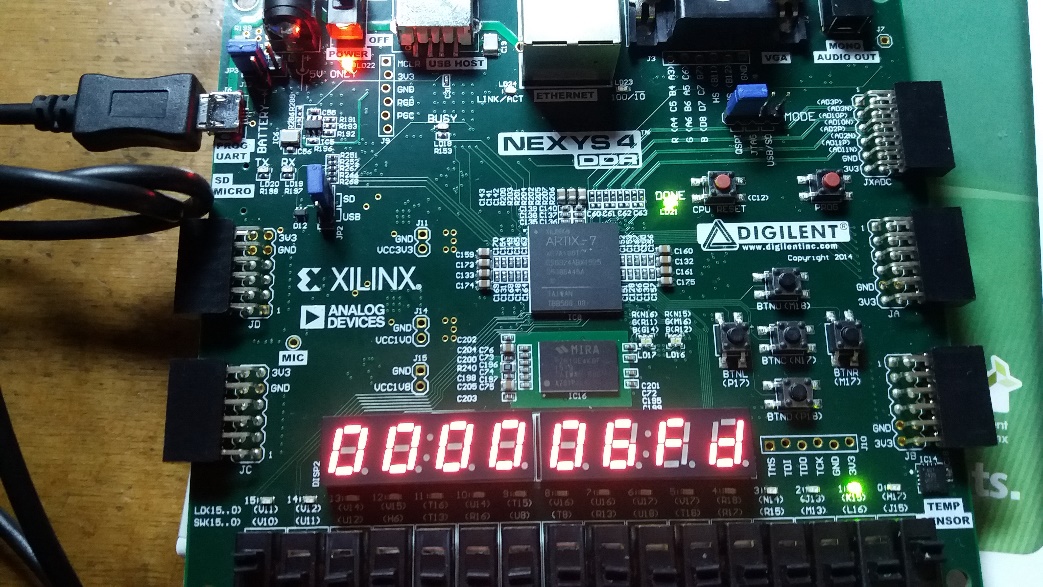


图 4‑8 分支预测指令条数

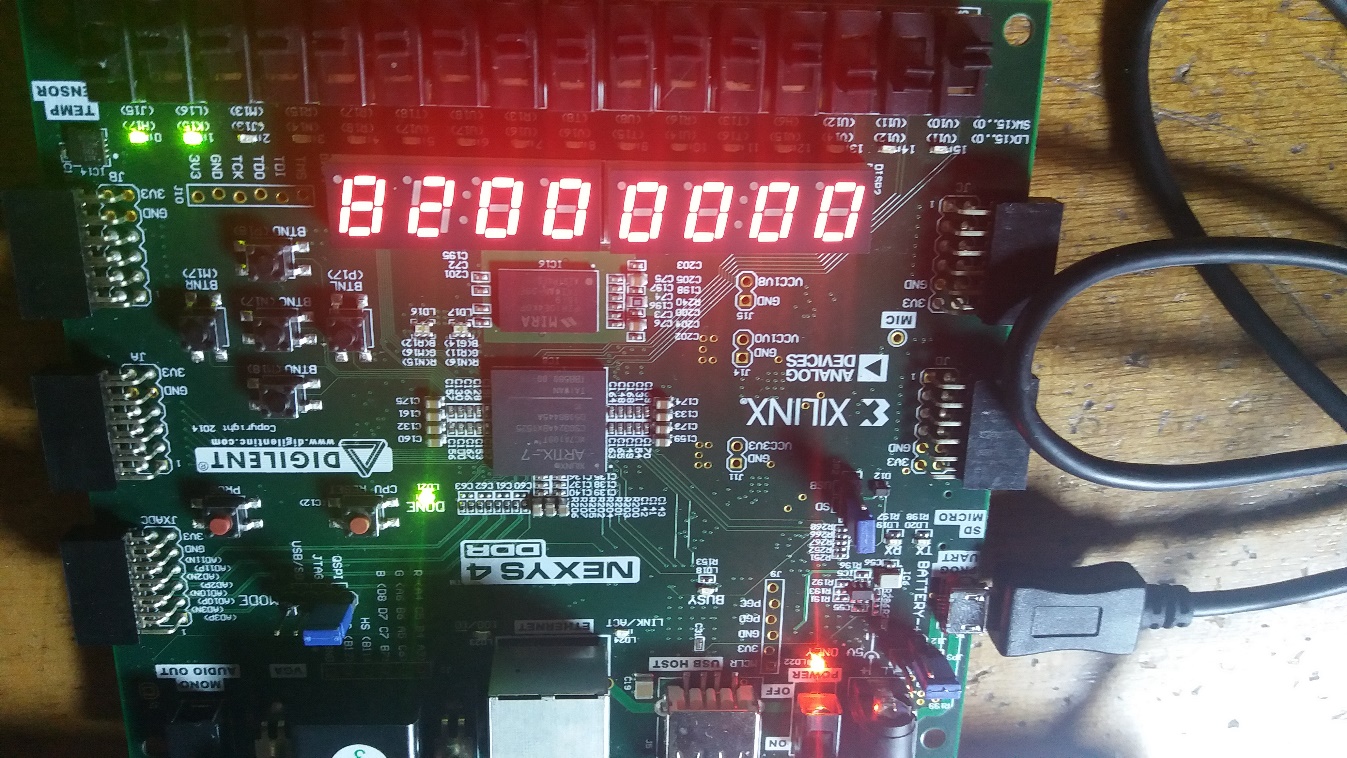


图 4‑9 分支预测错误次数

## 性能分析

### 插入气泡的方案分析

我采用的方案是针对bne、beq、bgez指令在EX段进行处理，故判断错误的时候需要插入两个气泡；而对j、jal、jr指令在ID段进行处理，故判断错误的时候需要插入1个气泡，最后的周期数=1546+4+2\*276+1\*38+120+1=2261。

若分支跳转指令全部都在EX段进行判断，则判断错误都需要插入两个气泡，理论最后的周期数应该=1546+4+2\*(276+38)+120+1=2299.

比较上述两种方案，采用第一种方案可以减少时钟周期数。

### 中断流水线方案分析

中断流水线需要选择在合适的时机处理中断，其时机可以有多种选择，下面介绍两种方案。我采用的方法是在IF段处理中断，为了让 流水线中的指令执行结束，需要让PC计数器暂停计数4个时钟周期，并在EX段连续插入4个气泡，并且在每次结束中断结束程序的时候也需要让流水线中的程序执行完，处理方法相同。故每处理一个中断需要额外的8个时钟周期。

方案二是在写回WB段处理中断，只要WB段的指令执行结束，可以将流水线中前4段的指令清空，不影响程序整体的执行，关键在于压入的epc是在WB段判断出的下一指令地址。采用这种方法可以在执行完当前用户指令后立即执行中断程序，中断程序执行结束之后可以立即返回用户程序执行，因此没有额外的始终周期浪费。

结合以上两种方案，显然第二种方案更优。由于开始我采用的思路便是第一种，后续没有对其进行修改。

## 主要故障与调试

### 流水线数据传输故障

理想流水线： 接口处数据传输问题。

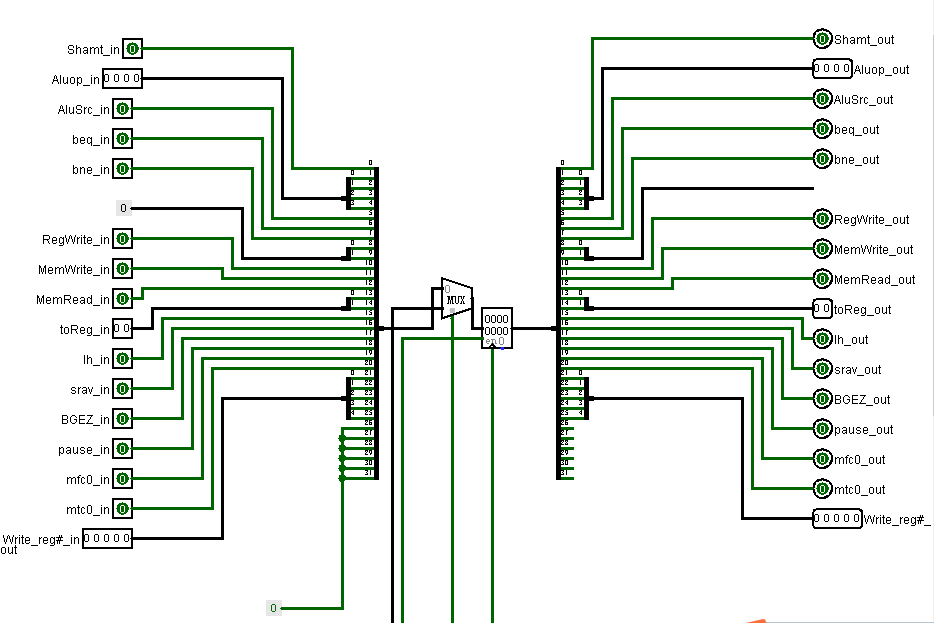


图 4‑10 流水寄存器部分电路图

**故障现象：**数据输出显示Error（即输出xxxxxxxx）。

**原因分析：**如图 4‑10，刚开始做的时候输入端分线器没用到的位就直接挂起，没有输入，这样会导致寄存器的输入相应的位为Error，导致寄存器无法传递数据，寄存器的输出为Error。

**解决方案：**寄存器输入端分线器未使用位全部与常量0相连接。

### 重定向流水线故障

重定向流水线测试时，运行错误。

**故障现象：**数码管一直显示全0，之后测试流水线中断的时候，中断返回地址错误，导致不能正常返回中断，排序结果为dcba9876654321。

**原因分析：**在logisim上单步跟踪程序的运行，对比mars上的运行结果，当syscall指令执行到ID段的时候，发现此时的$v0和$a0不是想要的值，因为此时还没对$v0和$a0进行写操作，结论是忘记对syacall两个特殊寄存器进行重定向。

测试中断程序，单步运行到mtc0指令，发现mtc0写入协处理器的值不是想要的值，结论是未对mtc0和mfc0等指令进行重定向处理。

单步测试排序程序，发现运行到sw指令时，存储的地址不是想要的地址，结论是未对sw进行重定向处理。

**解决方案**：增加对上述指令的重定向处理，对于syscall的重定向在实现方法里面已经详细地介绍。

对于mtc0指令和mfc0重定向时，一方面mtc0会对普通寄存器进行读操作，所以要判断是否会和EX段、Mem段发生数据相关；另一方面mfc0会写普通寄存器，故也要判断该指令之后的指令是否会与其发生数据相关。

对于sw指令，需要同时对rs和rt判断是否和前面的指令发生了数据相关。

### 嵌套中断流水线故障

嵌套中断流水线测试时，运行错误。

**故障现象：**压入epc的错误，按132的顺序按下中断按钮时2号中断不能正常返回1号中断。

**原因分析：** 按132的顺序按下中断按钮，并在logisim上单步调试，调试到执行mtc0指令时，写回发生在ID段，结论是错将ID段译出的mtc0信号当做协处理器的写信号，而写回应该发生在WB段。

更改之后发现出现了上述第二个错误，继续单步调试，发现eret指令执行后，eret对应计数器开始计数，但计数还未结束就产生了trap信号，trap计数器开始计数，当eret计数器计数结束准备返回中断者程序时由于trap\_halt信号程序继续暂停，不能返回，直到trap计数器计数结束，但此时PC的选择段输出的已经不是epc的值，故此时压入epc的值错误，导致2号中断不能正常返回。

**解决方案：**针对第一个问题，修改ID/EX、EX/Mem、Mem/WB流水线寄存器，以使其传递mtc0信号，并将WB段的mtc0信号作为epc寄存器的直接写信号。

针对第二个问题，关键在于保证eret计数器计数结束之后才产生trap信号，并且保证先返回被中断的程序再响应下一个中断。这里的解决方法是将eret计数器输出信号锁存一次，并将锁存信号作为中断处理模块中的eret输入，这样能够保证eret计数结束之后再产生中断信号，且之间先返回被中断的程序。

## 实验进度

实验进度如表 4‑1所示。

表 ‑ 课程设计进度表

| 时间 | 进度 |
| --- | --- |
| 第一天 | 完成了四条扩展指令并测试成功 |
| 第二天 | 完成单级中断，嵌套探索中 |
| 第三天 | 完成嵌套中断 |
| 第四天 | 完成理想流水线和气泡流水线，周期数4312 |
| 第五天 | 完成重定向流水线，周期数2261,直接跳转指令误取深度为1 |
| 第六天 | 写好verilog代码，调试中 |
| 第七天 | 流水线上板成功，指令条数2260，直接跳转类型指令误区深度为1，共38条 |
| 第八天 | 在logisim实现流水线嵌套中断，完成verilog代码开始调试 |
| 第九天 | 检查流水嵌套中断，开始写分支预测 |
| 第十天 | 完成分支预测，周期数1789 |

# 设计总结与心得

## 课设总结

1. 完成了srav、xori、bgez、lh四条指令扩展，完成了嵌套中断的实现，完成气泡流水线，完成重定向流水线、完成将重定向流水线verilog代码并烧录进FPGA板成功运行，完成logisim和FPGA板上的嵌套中断流水线，完成FPGA上的分支预测。
2. 最终版CPU除了具有基本的24条指令功能，还具有扩展的四条指令srav、xori、bgez、lh功能，另外具有自行添加的用于实现中断的mtc0、mfc0、eret指令功能。此外该CPU具有解决数据相关的功能，能同时用重定向和插入气泡解决数据相关；具有解决分支相关的功能；具有中断程序功能，并能实现多级中断和嵌套中断，实现中断的优先级以及嵌套中断能够正常返回；具有分支预测功能，一定程度上减少分支跳转失败概率。
3. 分支相关将分支指令bne、beq、bgez放在EX段处理，将直接跳转指令j、jr、jal放在ID段处理，故误区深度前者为2，后者为1.

## 课设心得

本次课设终于结束，过程是异常艰辛的，但成果也是丰硕的。在课设开始之前，就已经看到有其他班已经开始做组原课设的同学被其“折磨”得怨天怨地，因此心里对组原课设的难度也有了一定程度上的心理准备。课设第一天谭老师也坦言课设难度很大。果不其然，为了完成课设，连续两周每天早起爬到南一楼八楼从早到晚甚至在周末也加了班，终于是在最后一天通关。相比班上一些牛人来说我的进度算是慢的，但最后靠自己打通关就算时间较晚心里的成就感还是难以言喻的。现在回顾整个课设，每个阶段的经历都留下了值得深思的经验。

第一关是扩展四条指令，这一关其实是没有什么难度的，因为在上一学期已经使用logisim完成了单周期的CPU的设计，本人认为这一关除了保证每个人课设不会抄袭之外更重要的是有一贯让我们缓冲，能够熟悉自己隔了一个寒假之前完成的电路。这关主要的更改一方面是译码器的更改，由于单周期CPU实验中采用的是先根据指令生成各个指令的指令信号，后根据指令信号在产生各个控制信号，所以扩展指令相对比较简单。

第二关是单周期的嵌套中断的实现，这关应该是难度较大的一关。较为幸运的是老师给出了单个中断产生的电路，根据该电路扩展方便了很多。但一开始我就想实现嵌套中断，结果出现了错误开始不知道到底哪里出现了错误，比较好的流程是先实现单级中断，测试无误后再开始做多级嵌套中断。一步一步来可以减少纠错的时间。另外动手在logisim上布线之前一定要先想清楚逻辑，不能着急在布线过程中一边布线一百边思考怎么做，这样逻辑容易混乱，也因此在中断仲裁以及中断屏蔽部分卡了比较久的时间。这里卡的比较久还有一个原因是因为开始不懂协处理器是什么，也不清楚要用协处理器，开始还计划在普通寄存器中寻找寄存器来实现相应的功能。我觉得这一部分老师可以做一定程度上的提示，提示大家需要用协处理器实现，并且自行寻找协处理器有关知识。有这点启示可以减少我们之前盲目探索的时间。

第三关是理想流水线的实现，这关难度也较低。由于我们班提前上过流水线相关知识，所以基本上思路也比较清晰。但困难在于实现单周期CPU时为了电路看上去美观布线都比较紧凑，将其分为5个部分的时候大范围的移动是非常痛苦的，而且图中logisim经常会崩溃，需要重启，因此及时备份也是非常重要的。另外针对流水线寄存器的设计一定要提前设计好接口，避免后续再添加新的接口造成不必要麻烦。

第四关是气泡流水线实现，其实现原理在ppt上也讲得很清楚，只需要判断ID段和EX段、Mem段、WB段只要有冲突就会在EX段插入气泡。同时还需要让pc寄存器和IF/ID暂停指令传输。插入气泡的方法只要传空指令即可。这里有一点需要注意的是需要排除nop指令的干扰。

第五关是重定向流水线，其原理在系统结构课上也有提及，过程也比较顺利。

第六关是另一个难关，FPGA流水线。由于本身verilog基础不高，在编写程序的时候经常要返回去查语法，另外有些实现方法也过于冗余，可以用更加简单的实现方法。所以编写之前还是需要花相当的时间复习verilog语法。另外vivado有自带的IP核，提供了存储器等一些常用的原件代码供调用，但我用的是自己编写的存储器，结果产生了各种各样的问题，比如会出现行为仿真正确但时序仿真错误的现象，所以之后换做IP核实现之后顺利了很多。此外为了方便编程，我也花了较多的时间将logisim电路图全部用隧道实现，为各个端口命名，这也算是移植的一个技巧。另外在调试的时候行为仿真执行后可以将模块里的各个输入输出拖拽至仿真窗口而后重新仿真，这样可以很方便地查看想要查看的变量，而不需要把每一个需要查看的变量都定义成输出来查看。

第六关是嵌套中断流水线，关键在选择好中断的时机，最好的时机还是在写回阶段，因为此时清空流水线里的指令也不会对程序运行造成影响。这个阶段卡了比较久是因为之前在重定向时没有对中断模块进行处理，修改过程中解决一个错误有出现新的错误，所以必须在做重定向的时候就对所有模块所有指令都完成重定向，不能想着之后再做处理。

最后第七关是分支预测，网上的实现方法很多且很复杂，看了之后还是不知道怎么实现。但看ppt上的实现方法之后很快就有了思路，ppt上的方案较为简单。

整个过程基本都是自己一个人完成，但更重要的是，一定要相互讨论。过程中很多问题是和同学探讨才有了思路，不然会卡得更久。因此晨间讨论也方便了我们互相讨论，给讨论营造了一个氛围。当然讨论也不局限与组内，总之多讨论肯定有益无害。

虽然可以讨论可以问同学并且鼓励多讨论，但这是基于自己有一定独立思考的基础上，自己也需要有自己检索资料的能力，老师已经提供较为详实的ppt，也改了资料包，另外有些问题也可以通过检索网上资料解决，也不要习惯于问同学，这于人于己都是没有好处。

当然对于本次课设还是有点建议，一个是指导方面还是偏少，除了第一天早上对课设的框架描述了之后就没有后续的辅导，建议定期的稍微讲解各个阶段的原理，点到为止即可。有老师讲解过的话相信我们的进度会快很多，一个很明显的对比是我们班提前上过部分流水线相关知识，因此我们班在流水线上都比较轻松。另外对于组内贡献奖励，这个奖励一般也是进度快的人能够拿到，进度慢的人本身分数就相对低，这一加分自然有拉大了分差。自然这个制度是为了鼓励大家帮别人解决问题，但这事情主要还是看一个人会不会提问，只要会提问我相信班里的人都会帮他解决。其实想要实现帮助进度慢的人有个天梯赛制度就可以，考察团队整体进度来加分的同样能让进度快的人去帮助进度慢的人。最后选贡献奖是谁的时候也很尴尬因为大家都有互相帮助，选谁也是对不选的人不公平。

两周时间既充满天梭的艰辛也充满阶段性成功的喜悦，虽然过程艰辛，但通关之后觉得一切都值得。最后也非常感谢课程组的老师们，组原课程组是大学以来所知的最认真的课程组，是有真的认真思考如何培养我们的能力，如何虐我们。也感谢几位老师同样两周每天陪我们一起爬八楼，为我们耐心解答。

总之，这次课设让我受益匪浅。

# 参考文献

1. DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第4版).北京：机械工业出版社.
2. David Money Harris(美).数字设计和计算机体系结构（第二版）. 机械工业出版社
3. 秦磊华，吴非，莫正坤.计算机组成原理. 北京：清华大学出版社，2011年.
4. 袁春风编著. 计算机组成与系统结构. 北京：清华大学出版社，2011年.
5. 张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008年.

|  |
| --- |
| 一、原创性声明 |
| 本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。  特此声明！  **作者签字:** |