it

***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： | CS1409 |
| 学 号： | U201414800 |
| 姓 名： | 刘一龙 |
| 电 话： | 15927004132 |
| 邮 件： | sabertazimi@gmail.com |
| 完成日期： | 2016-02-24 周五下午 |

目 录

[1 课程设计概述 3](#_Toc474706961)

[1.1 课设目的 3](#_Toc474706962)

[1.2 设计任务 3](#_Toc474706963)

[1.3 设计要求 3](#_Toc474706964)

[1.4 技术指标 4](#_Toc474706965)

[2 总体方案设计 6](#_Toc474706966)

[2.1 单周期CPU设计 6](#_Toc474706967)

[2.2 中断机制设计 8](#_Toc474706968)

[2.3 流水CPU设计 9](#_Toc474706969)

[2.4 气泡式流水线设计 9](#_Toc474706970)

[2.5 数据转发流水线设计 10](#_Toc474706971)

[2.6 动态分支预测机制 10](#_Toc474706972)

[3 详细设计与实现 11](#_Toc474706973)

[3.1 单周期CPU 实现 11](#_Toc474706974)

[3.2 中断机制实现 17](#_Toc474706975)

[3.3 流水CPU实现 17](#_Toc474706976)

[3.4 气泡式流水线实现 17](#_Toc474706977)

[3.5 数据转发流水线实现 18](#_Toc474706978)

[3.6 动态分支预测机制实现 18](#_Toc474706979)

[4 实验过程与调试 19](#_Toc474706980)

[4.1 测试用例和功能测试 19](#_Toc474706981)

[4.2 可自行安排章节 19](#_Toc474706982)

[4.3 性能分析 19](#_Toc474706983)

[4.4 主要故障与调试 20](#_Toc474706984)

[4.5 实验进度 22](#_Toc474706985)

[5 设计总结与心得 24](#_Toc474706986)

[5.1 课设总结 24](#_Toc474706987)

[5.2 课设心得 24](#_Toc474706988)

[参考文献 26](#_Toc474706989)

# 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 | Move from CP0 |
| 26 | MTC0 | 访问CP0 | Move to CP0 |
| 27 | ERET | 中断返回 | EPC -> PC |
| 28 | DIVU | 无符号除 | $Rs/$Rt -> LO |
| 29 | MFLO | 读LO寄存器 | LO -> $Rd |
| 30 | LB | 读字 | MEM[$Rs+Offset][Addr1..0] -> Rt |
| 31 | BGTZ | 大于0跳转 | $Rs > 0 Jmp |

# 总体方案设计

## 单周期CPU设计

本次我们采用的方案硬布线控制，且采用哈佛结构进行内存管理（指令存储与数据存储分离）的方案，即利用控制单元生成控制信号，控制整个 CPU 随时钟变化进行工作。

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

#### 指令译码器

根据 MIPS 3种类型的指令，将指令的各个位段提取出来，即可完成指令译码器，输出的位段有 op（6位）、rs（5位）、rt（5位）、rd（5位）、sham（5位）、funct（6位）、imm16（16位）、imm26（26位）。

利用 op 位(6bit) 与 funct 位(6bit)，可以为所有指令生成一个多元与式，以确定当前指令类型。其关系表如表 2.1 指令类型与关系表所示：

表 2.1 指令类型与关系表

| 指令 | op5 | op4 | op3 | op2 | op1 | op0 | fn5 | fn4 | fn3 | fn2 | fn1 | fn0 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| add | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| addi | 0 | 0 | 1 | 0 | 0 | 0 | x | x | x | x | x | x |
| addiu | 0 | 0 | 1 | 0 | 0 | 1 | x | x | x | x | x | x |
| addu | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| and | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| andi | 0 | 0 | 1 | 1 | 0 | 0 | x | x | x | x | x | x |
| sll | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sra | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| srl | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| sub | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| or | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| ori | 0 | 0 | 1 | 1 | 0 | 1 | x | x | x | x | x | x |
| nor | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| lw | 1 | 0 | 0 | 0 | 1 | 1 | x | x | x | x | x | x |
| sw | 1 | 0 | 1 | 0 | 1 | 1 | x | x | x | x | x | x |
| beq | 0 | 0 | 0 | 1 | 0 | 0 | x | x | x | x | x | x |
| bne | 0 | 0 | 0 | 1 | 0 | 1 | x | x | x | x | x | x |
| slt | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| slti | 0 | 0 | 1 | 0 | 1 | 0 | x | x | x | x | x | x |
| sltu | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| j | 0 | 0 | 0 | 0 | 1 | 0 | x | x | x | x | x | x |
| jal | 0 | 0 | 0 | 0 | 1 | 1 | x | x | x | x | x | x |
| jr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| sys | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |

利用上述关系，可以构造任意指令的布尔信号表达式。以 add（R型） 为例，其布尔信号为

add = ~op5 & ~op4 & ~op3 & ~op2 & ~op1 & ~op0 & fn5 & ~fn4 & ~fn3 & ~fn2 & ~fn1 & ~fn0；

以 j（非R型）为例，其布尔信号为

j = ~op5 & ~op4 & ~op3 & ~op2 & op1 & ~op0。

同理可得到所有指令的布尔信号表达式，利用 logisim 的分析电路功能，利用这些表达式自动生成布尔信号电路。由于 logisim 分析电路将输入输出门个数限制在 12 个，故需生成 2 次电路（1 次 12 条指令），再将其封装成一个完整的指令布尔信号电路。

#### PC 更新单元

将整个 CPU 分成 2 个部分。首先观察 PC 更新部分，即取指单元（IFU）。分析所要实现的 28 条指令可以发现，除beq、bne、bgtz、j、jal、jr这 6 条指令外，大部分指令对于 PC 的操作都是简单的 PC += 4 bytes。利用第 2 个设计中的指令布尔信号，可以得到当前是否为跳转指令。可以把 24 条指令大致分为 4 组，第 1 组为顺序取指（大部分指令），第 2 组为相对 PC 跳转取指（beq/bne）,第 3 组为绝对跳转取指（j/jal），第 4 组为寄存器间接跳转取指（jr）。于是可以得到，此时需要 3 个 1 选 2 选择器，以根据不同指令选择不同的 PC 更新方式，由此产生了 3 个控制信号。假设 PC + 4 为默认更新方式，则第 2 组选择控制信号应为 Bne & ~Eq | Beq & Eq | Bgtz & ~Less & ~Eq，第 3 组选择控制信号为 J | Jal，第 4 组选择控制信号为 Jr，将上述逻辑综合成一张关系表，如表 2.2 PC 更新逻辑表所示。

表 2.2 PC 更新逻辑表

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Beq | Bne | Bgtz | Eq | Less | J | Jal | Jr | PC |
| 1 | 0 | 0 | 1 | x | 0 | 0 | 0 | PC相对 |
| 0 | 1 | 0 | 0 | x | 0 | 0 | 0 | PC相对 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | PC相对 |
| x | x | x | x | x | 1 | x | 0 | 伪绝对 |
| x | x | x | x | x | x | 1 | 0 | 伪绝对 |
| x | x | x | x | x | x | x | 1 | PC+OF |

#### 运算器

根据 ALU\_OP，利用多选器将不同部件（如移位器、加法器、乘法器等）的输出选择输出即可。

表 2.3 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 |

ALU 唯一需要注意的便是溢出检测单元。

无符号溢出检测：无符号加时，若产生溢出，则加法器最高位进位为1；无符号减时，若产生溢出，则加法器（Y进行处理后直接输入 32 位全并行加法器）最高位进位为0。利用 4 选 16 选择电路将无符号加/减的溢出输出至 UOF。即选择器输入 5 的逻辑式为：in5 = adder\_cout，选择器输入6的逻辑式为：in6 = ～adder\_cout。

有符号溢出检测：有符号加时，产生溢出的情况只可能为 ++ -> - 与 -- -> +，故可得溢出码逻辑式为：addOF = (X\_f & Y\_f & !S\_f)|(!X\_f & !Y\_f & S\_f)；同理，有符号减时，产生溢出的情况只可能为 +- -> - 与 -+ -> +，故可得溢出码逻辑式为：subOF = (X\_f & !Y\_f & !S\_f)|(!X\_f & Y\_f & S\_f)。同无符号溢出检测，利用 4 选 16 选择电路将有符号加/减的溢出输出至 OF。

#### 寄存器堆RF

利用logisim平台构建一个MIPS寄存器组，内部包含32个32位寄存器，其具体功能如下，具体封装文件为regfile.circ，如表 2.4 芯片引脚与功能描述所示。

表 2.4 芯片引脚与功能描述

| 引脚 | 输入/输出 | 位宽 | 功能描述 |
| --- | --- | --- | --- |
| R1# | 输入 | 5 | 读寄存器1编号 |
| R2# | 输入 | 5 | 读寄存器2编号 |
| W# | 输入 | 5 | 写入寄存器编号 |
| Din | 输入 | 32 | 写入数据 |
| WE | 输入 | 1 | 写入使能信号，为1时，CLK上跳沿将Din数据写入W#寄存器 |
| CLK | 输入 | 1 | 时钟信号，上跳沿有效 |
| R1 | 输出 | 32 | R1#寄存器的值 |
| R2 | 输出 | 32 | R2#寄存器的值 |
| $s0 | 输出 | 32 | 编号为16的寄存器的值 |
| $s1 | 输出 | 32 | 编号为17的寄存器的值 |
| $s2 | 输出 | 32 | 编号为18的寄存器的值 |
| $ra | 输出 | 32 | 编号为31的寄存器的值 |

在设计寄存器组的实现方式时，唯一需要注意的地方便是寄存器组的使能端。只要利用合理的组合逻辑，使得同一时刻只有 1 个寄存器（写目标寄存器）的使能端为 1（有效），便可实现寄存器组的输入功能。在此种情况下，时钟源与输入数据只需同时连接 31 个寄存器即可（除开 0 号寄存器），无需而外逻辑。下面利用一个单独的模块，实现上述使能端信号选择功能。

给定一个 W#（5 bits） 信号，表示当前写入目标寄存器的编号，则只需利用一个解复用器，将 W# 作为选择端，常量 1 作为输入端，即可实现同时只有目标寄存器的写使能信号为 1 的设计。输出为 8 组使能信号，每组 4 bits 使能信号，同时只有 1 bit 使能信号为0。（分组是为了使能信号接入寄存器组时的连线美观）

只需利用多路选择器，将 R1#/R2#（读出寄存器编号）作为选择信号，32 寄存器输出端最为选择输入端即可实现此功能。

将上述 2 个设计综合在一起，即可得到完整的寄存器组。唯一值得注意的地方，需要使 0 号寄存器（$zero）保持常 0。

### 数据通路的设计

数据通路的实现并非采用工程化的方式，而是利用逐步扩展的方式完成整个数据通路的实现。下面是各个阶段的设计思路：

常规 R 型指令数据通路：只需寄存器组与 ALU 即可实现常规 R 型指令的数据通路，此时只需 2 个控制信号 ，RegWe 控制寄存器组的写使能，ALUOp 控制 ALU 运算逻辑，此类指令包括 add、addu、and、sub、or、nor、slt、sltu。

常规 I 型指令数据通路：只需在 R 型指令通路的基础上，利用新的控制信号将 ALU\_Y 改为立即数输入，同时将写入寄存器编号 RW# 改为 rt 即可，此时需要 3 个新的控制信号，并加入 3 个多选器，RegDst 选择写入寄存器编号（rt 或 rd），ALUSrc 选择 Y 端输入（rt 或 imm16），ExtOp 选择立即数扩展方式（无符号或有符号扩展），此类指令包括 addi、addiu、andi、ori、slti。值得一提的是， addiu 指令对 imm16 进行的也是符号扩展。

移位指令数据通路：移位指令的源操作数不再为 （rs），而是 （rt），且移位字段不是由寄存器/imm16给出，故需要 1 个新的信号 ALUSham，并加入 2 个多选器，以改变 ALU 2 个输入端的值。

访存指令数据通路：从寄存器组到 ALU 中间的数据通路，lw/sw 指令与常规 I 型指令间没有差异，唯一不同的是，lw/sw 指令 ALU 的输出端不再是普通的值，而是计算好的地址值。由于访存操作的存在，需要引入新的主存组件至数据通路中，并增加 2 个新的控制信号，以控制主存的读写。对于lw 指令，需要从主存读取数据至寄存器，增加新信号 RAMtoReg，并增加一个多选器，以选择寄存器组的输入数据（ALU\_Result 或 RAMDataOut）；对于 sw 指令，需要增加 1 个新信号 RAMWe，以控制对主存的写访问，并将 （rt）接至主存加载数据端（RAMDataIn）。 条件跳转指令数据通路：beq/bne 指令无需修改数据通路，只需给出合适的控制信号，使得 ALU\_X = （rs），ALU\_Y = （rt），并从 ALU 得到 Eq 标志量，从而在 IFU 中完成指令跳转功能。

直接跳转指令数据通路：j 与jr指令无需修改数据通路，直接通过前述 IFU 设计即可实现指令跳转功能。jal 指令还需借助原数据通路将 rs 值取出加 4 后写入 $a0 寄存器，此时 ALU\_X = （rs），ALU\_Y = （4），ALU\_OP = ADD，RW# = 0x1f，故需修改数据通路。

系统调用指令数据通路：关于 syscall 的实现，无需修改数据通路，只需将 $v0 的值从寄存器组引出，并将其与 0xah 进行比较，再结合 Syscall 指令布尔信号，即可得到 Halt 停机信号与输出七段管信号。利用 Halt信号，对时钟源进行简单处理，即可实现 syscall 0xah 功能。

扩展指令数据通路：divu 需要增加一个 LO 寄存器与一个 WriteToLO 信号（使能控制信号，将 ALU 除的结果在合适的时机写入 LO 寄存器），需将 ALUOp 译码成除法操作；mflo 需要修改写回寄存器的数据通路，额外增加一个控制信号，将写回寄存器的数据修改为 LO 寄存器的值；bgtz 只需在原来 branch 分支跳转成功逻辑电路的基础上增加判断 bgtz 跳转成功的逻辑即可，即 branch = （Beq & Eq） | （Bne & ~Eq）|（Bgtz & ~Eq & ~Less）;lb 指令只需将 RAM 读出的数据进行字节选择并进行符号扩展，得到新的一个 32 bit 的数据，再利用控制信号 RAMByte在原来 RAM 数据输出和新数据间进行多选即可。

### 控制器的设计

首先对于控制信号进行统计，包括各个主要部件所需要输入的控制信号，以及数据通路合并表中所示的具有多输入的主要部件需要进行输入选择的控制信号，并且对各个统计信号的各种取值情况进行定义，统计得到的控制信号以及说明如表 2.5 主控制器控制信号的作用说明所示。

表 2.5 主控制器控制信号的作用说明

| 控制信号 | 取值 | 说明 |
| --- | --- | --- |
| ALUOp | 0-15 | ALU 不同计算 |
| RegWe | 0 | 寄存器组写使能无效 |
| 1 | 寄存器组写使能有效 |
| ALUSrc | 0 | ALU\_Y 的输入来自 $rt |
| 1 | ALU\_Y 的输入来自 imm16 |
| ALUSham | 0 | ALU\_Y 的输入来自 $rt/imm16 |
| 1 | ALU\_Y 的输入来自 sham |
| RegDst | 0 | 寄存器组写寄存器号为 rt |
| 1 | 寄存器组写寄存器号为 rd |
| ExtOp | 0 | imm16 进行无符号扩展 |
| 1 | imm16 进行有符号扩展 |
| RAMWe | 0 | RAM 写使能无效 |
| 1 | RAM 写使能有效 |
| RAMtoReg | 0 | 寄存器组写寄存器数据来自 ALU |
| 1 | 寄存器组写寄存器数据来自 RAM |
| Beq/Bne/Bgtz/J/Jal/Jr | 0 | 当前不是对应的指令（beq/bne/bgtz/j/jal/jr） |
| 1 | 当前是对应的指令（beq/bne/bgtz/j/jal/jr） |
| Syscall | 0 | 当前指令不是系统调用 |
| 1 | 当前指令是系统调用 |
| WritetoLO | 0 | LO 寄存器写使能无效 |
| 1 | LO 寄存器写使能有效 |
| LOtoReg | 0 | 寄存器组写寄存器数据来自 ALU/RAM |
| 1 | 寄存器组写寄存器数据来自 LO寄存器 |
| RAMByte | 0 | RAM 的输出为字数据 |
| 1 | RAM 的输出为字节扩展数据 |

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。设计好数据通路，即可按照指令特性与数据通路走向，确定对于每一个指令的控制信号集。控制信号结果如表 2.6 控制信号表的框架（1）、表 2.7 控制信号表的框架（2）、表 2.8 控制信号表的框架（3）、表 2.9 控制信号表的框架（4）。利用多路或门，即可实现所有控制信号的生成，从而实现完整的控制单元。

表 2.6 控制信号表的框架（1）

| 控制信号 | add | addi | addiu | addu | and | andi | sll | sra |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ALUOp0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| ALUOp1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| ALUOp2 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| ALUOp3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RegWe  1enable | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| ALUSrc  0rt  1imm16 | 0 | 1 | 1 | 0 | 0 | 1 | x | x |
| ALUSham  1=sham | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| RegDst  0=rt  1=rd | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| ExtOp  1=s | x | 1 | 1 | x | x | 0 | x | x |
| RAMWe | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RAMtoReg | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Beq | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Bne | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Jmp | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Jal | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Jr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Syscall | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

表 2.7 控制信号表的框架（2）

| 控制信号 | srl | sub | or | ori | nor | lw | sw | beq |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ALUOp0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | x |
| ALUOp1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | x |
| ALUOp2 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | x |
| ALUOp3 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | x |
| RegWe  1enable | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| ALUSrc  0rt  1imm16 | x | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| ALUSham  1=sham | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RegDst  0=rt  1=rd | 1 | 1 | 1 | 0 | 1 | 0 | x | x |
| ExtOp  1=s | x | x | x | 0 | x | 1 | 1 | x |
| RAMWe | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| RAMtoReg | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 |
| Beq | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| Bne | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Jmp | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Jal | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Jr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Syscall | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

表 2.8 控制信号表的框架（3）

| 控制信号 | bne | slt | slti | sltu | j | jal | jr | syscall |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ALUOp0 | x | 1 | 1 | 0 | x | 1 | x | x |
| ALUOp1 | x | 1 | 1 | 0 | x | 0 | x | x |
| ALUOp2 | x | 0 | 0 | 1 | x | 1 | x | x |
| ALUOp3 | x | 1 | 1 | 1 | x | 0 | x | x |
| RegWe  1enable | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| ALUSrc  0rt  1imm16 | 0 | 0 | 1 | 0 | x | x | x | 0 |
| ALUSham  1=sham | 0 | 0 | 0 | 0 | x | x | x | 0 |
| RegDst  0=rt  1=rd | x | 1 | 0 | 1 | x | x | x | x |
| ExtOp  1=s | x | x | 1 | x | x | x | x | x |
| RAMWe | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| RAMtoReg | 0 | 0 | 0 | 0 | x | 0 | x | x |
| Beq | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Bne | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Jmp | 0 | 0 | 0 | 0 | 1 | 0 | x | 0 |
| Jal | 0 | 0 | 0 | 0 | 0 | 1 | x | 0 |
| Jr | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Syscall | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

表 2.9 控制信号表的框架（4）

| 控制信号 | divu | mflo | lb | bgtz |
| --- | --- | --- | --- | --- |
| ALUOp0 | 0 | x | 1 | 1 |
| ALUOp1 | 0 | x | 0 | 1 |
| ALUOp2 | 1 | x | 1 | 0 |
| ALUOp3 | 0 | x | 0 | 1 |
| RegWe  1enable | 0 | 1 | 1 | 0 |
| ALUSrc  0rt  1imm16 | 0 | x | 1 | 0 |
| ALUSham  1=sham | 0 | x | 0 | 0 |
| RegDst  0=rt  1=rd | x | 1 | 0 | x |
| ExtOp  1=s | x | x | 1 | x |
| RAMWe | 0 | 0 | 0 | 0 |
| RAMtoReg | 0 | 0 | 1 | 0 |
| Beq | 0 | 0 | 0 | 0 |
| Bne | 0 | 0 | 0 | 0 |
| Jmp | 0 | 0 | 0 | 0 |
| Jal | 0 | 0 | 0 | 0 |
| Jr | 0 | 0 | 0 | 0 |
| Syscall | 0 | 0 | 0 | 0 |
| WriteToLO | 1 | 0 | 0 | 0 |
| LOToReg | 0 | 1 | 0 | x |
| RAMByte | 0 | 0 | 1 | x |
| Bgtz | 0 | 0 | 0 | 1 |

## 中断机制设计

### 总体设计

中断在本质上，可以等同于 J/Jal/Jr 等指令，但由于其具有一定的特殊性，在实现上比 J/Jal/Jr 等指令复杂得多。多级嵌套中断的整体过程如图 2.2 多级嵌套中断流程图所示。可以看到一次中断执行大致分为 5 个部分：中断识别与保护断点，保护现场与屏蔽字，中断处理服务例程，恢复现场与屏蔽字，中断返回。需要实现以下电路：中断识别电路、断点保存电路以及 3 条特殊的中断相关指令。

利用中断识别电路，利用硬件识别当前中断源与中断号，利用硬件将 PC 切换至中断处理程序地址，可以完成中断识别的任务，而中断号保存于 Cause 寄存器。

利用断点保存电路，简单地将旧的 PC\_Next 值存于一个新的寄存器，以起到断点保存的作用，将这个新寄存器称为 EPC 寄存器。

除此之外，开/关中断、保护现场与屏蔽字、恢复现场与屏蔽字以及中断返回，皆可利用 mfc0、mtc0、eret 以及一些通用指令（如 lw/sw实现栈帧）完成，利用栈帧可实现多级中断的现场维护，使得保护现场与恢复现场都在栈帧上进行操作，使得中断得以嵌套，不会影响上一级中断现场的保护与恢复。其中，屏蔽字与中断使能位保存于 Status 寄存器。将上述提到的 3 个特殊寄存器 Status（中断使能位与屏蔽字）、EPC（断点）、Cause（中断号）封装成一个组件，实现一个简易的 CP0 协处理器（See MIPS Run 第二版）。

综上所述，所需实现的任务有中断识别电路、简易CP0协处理器、PC 保护与恢复电路逻辑。

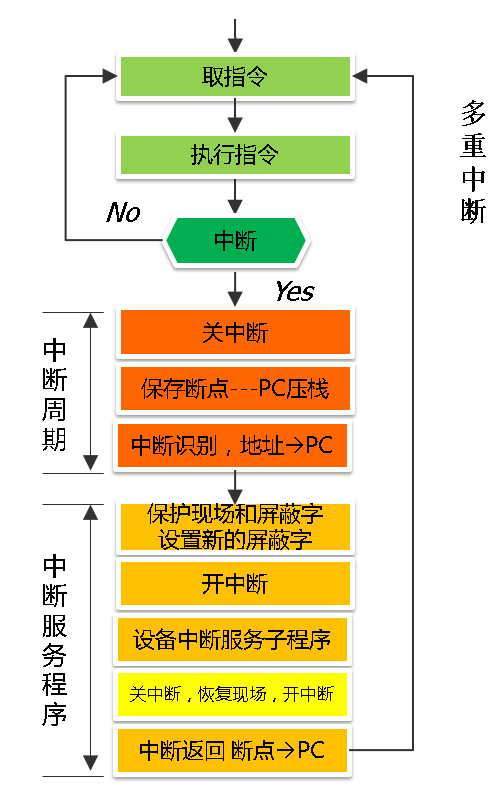


图 2.2 多级嵌套中断流程图

### 硬件设计

中断识别电路如所示。当有中断来时，结合中断屏蔽字，判定当前是否产生中断，将中断的发生信号接入优先编码器，即可利用优先编码器的特性，识别具有最高响应优先级的中断，并生成其中断号。同时优先编码器的有效位输出与中断使能信号相与，可得到最终的中断发生信号，其表示当前系统是否有中断发生。当中断得到响应与处理后，将相应中断寄存器中锁存的中断信号利用清零端清零即可。其中，中断屏蔽字由 Status 寄存器 8~14 位提供（此次实验只需 8-10 位），中断使能信号由 Status 寄存器 0 位提供（此次实验利用 1 个逻辑非门将其功能反置，0 表示开中断，1 表示关中断，使得整个 CPU 默认处于开中断状态）,中断号输出接至 Cause 寄存器。

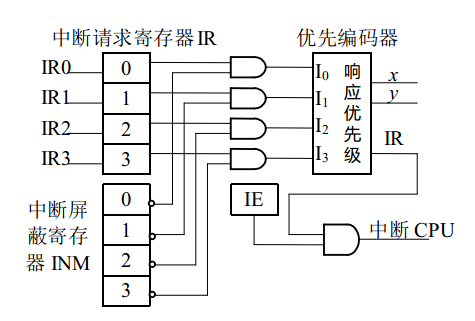


图 2.3 中断识别与响应电路

Int\_NO

PC

GR

Status

EPC

Cause

图 2.4 简易 CP0 示意图

CP0 协处理器：在此次实验中，其本质是一个简易的寄存器组，拥有 3 个寄存器，分别为 Status、EPC、Cause，用于保存屏蔽字、中断使能位、断点（旧 PC\_Next）、中断号。其功能与寄存器组一致，包括输入与输出 2 部分，其原理图如图 2.4 简易 CP0 示意图所示。输出部分逻辑简单，根据 MFC0 指令中 rd 部分对 3 个寄存器输出进行多选即可。输入部分逻辑更复杂：Status 只有 1 个输入，为 MTC0 中rt 对应的通用寄存器组中寄存器的值；Cause 只有 1 个输入，本次实验无需利用软件对其进行修改，故其输入仅为中断识别电路中的优先编码器的中断号输出；EPC 有 2 个输入，一个为 MTC0 指定的通用寄存器数据，另一个即为用于保护断点的旧 PC值，利用中断信号的有无对其进行多选即可。

PC更新：通过分析中断过程可以发现，只有在 2 个时机需要改变 PC 值，即中断进入与中断返回（eret）时。只需在原来 IFU 模块中的 PC 更新处增加 2 个多选器，分别输入 0x800 （中断程序地址）与 EPC 即可完成中断进入与中断返回的硬件支持。

### 软件设计

要完成多级嵌套中断的实现，需要 3 条指令的支持，分别为 mfc0、mtc0和eret，其作用分别为从 CP0 取值至通用寄存器组、将通用寄存器组的值送 CP0、以及中断返回。其硬件设计已在 2.2.3 节讲述，即增加 CP0RegWe控制信号以完成 对CP0中寄存器的写、CP0toReg 控制信号以完成对通用寄存器的写（将数据输入端由 ALU/RAM/LO 切换为 CP0）、Eret控制信号以完成对 PC 的恢复（EPC 送 PC）。这 3 条指令的控制信号如表 2.10 中断相关指令控制信号表所示。

表 2.10 中断相关指令控制信号表

| 控制信号 | mfc0 | mtc0 | eret |
| --- | --- | --- | --- |
| ALUOp0 | x | x | x |
| ALUOp1 | x | x | x |
| ALUOp2 | x | x | x |
| ALUOp3 | x | x | x |
| RegWe  1enable | 1 | 0 | 0 |
| ALUSrc  0rt  1imm16 | x | x | x |
| ALUSham  1=sham | x | x | x |
| RegDst  0=rt  1=rd | 0 | x | x |
| ExtOp  1=s | x | x | x |
| RAMWe | 0 | 0 | 0 |
| RAMtoReg | 0 | x | x |
| Beq | 0 | 0 | 0 |
| Bne | 0 | 0 | 0 |
| Jmp | 0 | 0 | 0 |
| Jal | 0 | 0 | 0 |
| Jr | 0 | 0 | 0 |
| Syscall | 0 | 0 | 0 |
| WriteToLO | 0 | 0 | 0 |
| LOToReg | 0 | x | 0 |
| RAMByte | 0 | 0 | 0 |
| Bgtz | 0 | 0 | 0 |
| CP0RegWe | 0 | 1 | 0 |
| CP0ToReg | 1 | x | 0 |
| Eret | 0 | 0 | 1 |

## 流水CPU设计

### 总体设计

流水线是一种实现多条指令重叠执行的技术。一个 MIPS 指令通过包含以下 5 个处理步骤：从指令存储器中读指令，称为取指阶段（IF）；指令译码的同时读取寄存器，称为译码/取数阶段（ID）；执行操作，进行逻辑运算，称为执行阶段（EX）；从数据存储器中读取操作数，或将数据写入数据存储器，称为访存阶段（MEM）；将结果写回寄存器，称为写回阶段（WB）。

可以看出，单周期 CPU 的实现与器件排布可以十分简单地扩展、拉伸成 5 段流水，唯二需要注意的 2 点：写寄存器相关的 3 个输入即写寄存器号、写寄存器使能、写寄存器数据，需要从寄存器组周围大迁徙至 WB 阶段；需要仔细推敲地址计算与分支跳转逻辑的放置阶段，本次实验决定按照《数字设计与计算机体系结构（Digital Design and Computer Architecture）》一书中所提供的思路，将其置于 ID阶段，与译码、取数同时进行，降低误取深度，提升流水线性能。

除此之外，还需解决流水线 3 大经典问题，结构冒险、数据冒险与控制冒险。由于本次实验采用哈佛结构，故取指与访问不会存在结构冒险，同时由于 MIPS 规定只有 L/S 型指令可以访存，基于以上 2 点可以得出当前流水线不存在结构冒险；通过重定向，将 MEM 或 WB 阶段的数据通过旁路接至 EX 端供计算使用，可以解决大部分数据冒险，再利用插气泡的方式解决 Load-Use 数据冒险即可；当误取指令后，需要清空前段流水线，将误取指令清除，由于将地址计算放置于 ID 段，故只需清除 ID/IF 段流水寄存器。需要注意的是，由于将地址计算置于 ID 段，需要在此段解决 Beq/Bne/Bgtz 存在的数据冒险，同样地可以利用重定向与插气泡的方式解决。

经过上述分析，可以得到流水线基本原理图如图 2.5 流水线原理图（Digital Design and Computer Architecture）所示。实际实现与其存在一些差异，如 ALU 输入端的多选、地址计算与跳转逻辑以及 PC 更新逻辑，具体差异参见实现一节。

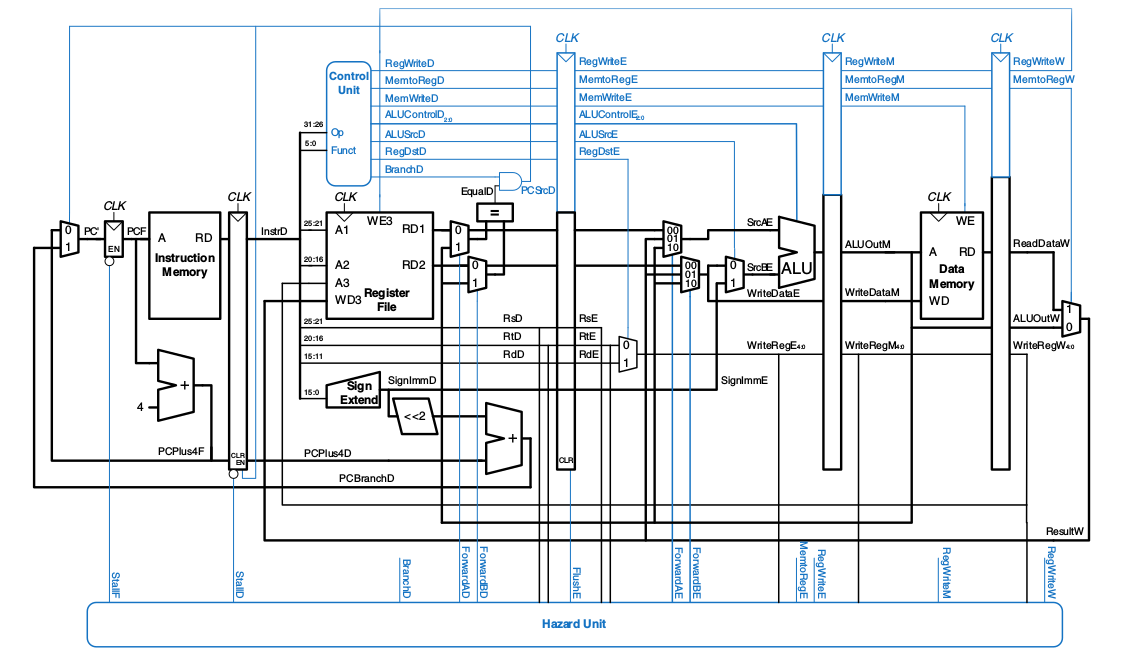


图 2.5 流水线原理图（Digital Design and Computer Architecture）

### 流水接口部件设计

通过观察流水线各个阶段，只需将后面阶段所需信号与数据通过流水线向后传递即可。各个流水寄存器输入输出设计如下所示：

IF/ID：PC（ID/EX/MEM/WB）、IR（ID/EX/MEM/WB）

ID/EX：PC（EX/MEM/WB）、IR（EX/MEM/WB）、WritetoLO/RegWe/RAMtoReg/LOtoReg/Syscall（WB）、RAMWe/RAMByte（MEM）、RegDst/ALUOp/ALUSrc/ExtOp/ALUSham/Jal（EX）、rs/rt/rd/sham/imm16/R1（EX）、R2（EX/MEM）

EX/MEM：PC（MEM/WB）、IR（MEM/WB）、WritetoLO/RegWe/RAMtoReg/LOtoReg/Syscall（WB）、RAMWe/RAMByte（MEM）、rt/R2（MEM）、Result/RW#（MEM/WB）

MEM/WB：PC、IR、WritetoLO/RegWe/RAMtoReg/LOtoReg/Syscall、RAMData/Result/RW#。

利用寄存器将上述信号与数据在流水段之间随时钟进行锁存即可。

其中，将 PC 与 IR 充满整个流水线，方便调试。

### 理想流水线设计

理想流水线无需考虑数据冒险与控制冒险，只需利用 4 个流水寄存器将 CPU 分为 5 段流水即可，其原理图如图 2.6 理想流水线原理图所示。

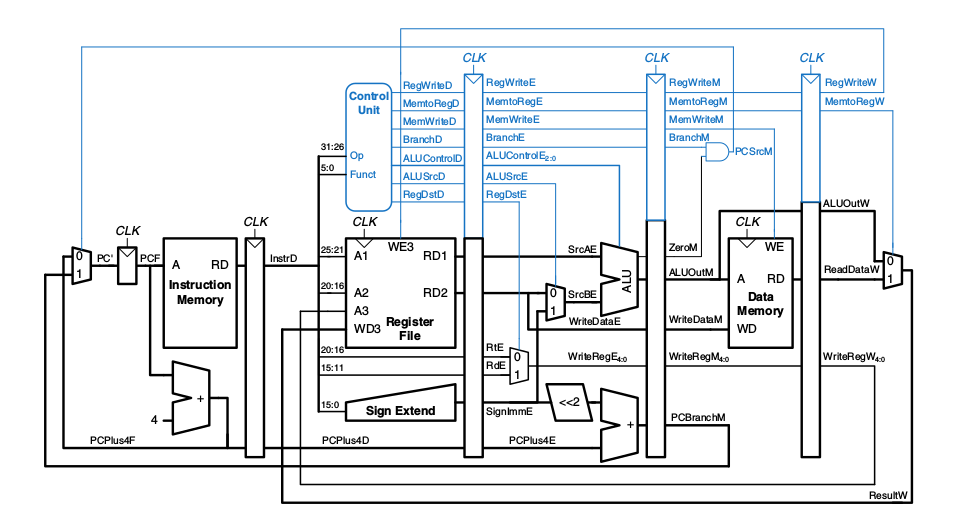


图 2.6 理想流水线原理图

## 数据转发流水线设计

在总体设计中提到过，大部分数据冒险都可以用转发解决，故在此次实验中先完成数据转发模块，再进行利用气泡方式清除剩余数据冒险。

在 ID 阶段，由于将地址计算与分支逻辑置于此阶段，故需要将此进行数据转发。其转发源头为 MEM ALU计算结果 Result 以及 WB 写回寄存器的值 RegData。以 R1 为例，其转发逻辑如下所示（同理，R2 的转发只需将 rs 换成 rt 即可）：

if ((IF/ID.rs != 0) && (IF/ID.rs == EX/MEM.RW#) && EX/MEM.RegWe) begin

IF/ID.ForwardA = 10 (MEM)

end else if ((IF/ID.rs != 0) && (IF/ID.rs == MEM/WB.RW#) && MEM/WB.RegWe) begin

IF/ID.ForwardA = 01 (WB)

end else begin

IF/ID.ForwardA = 00 (no forwarding)

end

在 EX 阶段，若此时此阶段为任何使用寄存器的指令，且 MEM与WB阶段对其用到的某个寄存器进行写，则会发生数据冒险，需要进行转发。以 ALU\_X 端输入为例，其转发逻辑如下所示（ALU\_Y 端转发同理，将 rs 替换成 rt 即可）：

if ((ID/EX.rs != 0) && (ID/EX.rs == EX/MEM.RW#) && EX/MEM.RegWe) begin

ID/EX.ForwardA = 10 (MEM)

end else if ((ID/EX.rs != 0) && (ID/EX.rs == MEM/WB.RW#) && MEM/WB.RegWe) begin

ID/EX.ForwardA = 01 (WB)

end else begin

ID/EX.ForwardA = 00 (no forwarding)

end

最后，在 MEM 阶段，若此时此阶段为 sw指令，WB 阶段为 lw 指令，则利用转发可以解决数据冒险，此种不插入气泡即可解决此种看上去为 Load-Use 的数据冒险，其转发逻辑如下所示：

MEMForward = (MEM/WB.RegWe && EX/MEM.RAMWe && EX/MEM.rt != 0 && EX/MEM.rt == MEM/WB.RW#)

综上，利用上述 3 处的转发，即可解决大部分数据冒险。

## 气泡式流水线设计

解决完大部分数据冒险后，仍有一种数据冒险无法靠转发解决，即 Load-Use 冒险。当 EX 端为寄存器读指令，而 MEM 阶段为 lw 指令时，由于写入寄存器的值不在 EX/MEM 与 MEM/WB 流水寄存器，由于数据从存储器读出具有一定的延迟，故用转发不适合解决此种数据冒险。最有效地办法为插入气泡，使 Use 指令停一个周期，等 lw 指令来到 WB 阶段后，再利用原先的转发逻辑解决数据冒险。而 Load-Use 阶段应置于 ID - EX 段，其检测逻辑如下所示：

lwstall = ID/EX.rt != 0 && ID/EX.RAMtoReg && (ID/EX.rt == IF/ID.rs || ID/EX.rt == IF/ID.rt)

当检测到 Load-Use 冒险后，在下一个上升沿，关闭 PC 寄存器与 IF/ID 流水寄存器的写使能，打开 ID/EX 流水寄存器的同步清零信号，即可成功地在 EX 段插入一个气泡，使得 ID 段的 Use 指令延后一个周期，达到清除冒险的任务。

除此之外，还存在一个数据冒险。由于将地址计算放置于 ID 段，当 EX 段指令为写寄存器指令或 MEM 段为 lw 指令时，无法利用 ID 段的转发逻辑解决此冒险（理由与 Load-Use 冒险一样，ALU 运算器与 RAM 读取皆存在一定的时延），故需向流水线 EX 插入一个气泡，其检测逻辑如下所示：

branchstall = (IF/ID.JmpNeedReg && ID/EX.RegWe && ID/EX.RW# != 0 && (ID/EX.RW# == IF/ID.rs || ID/EX.RW# == IF/ID.rt))

|| (IF/ID.JmpNeedReg && EX/MEM.RAMtoReg && EX/MEM.RW# != 0 && (EX/MEM.RW# == IF/ID.rs || EX/MEM.RW# == IF/ID.rt))

当检测到分支指令数据冒险时，插入气泡即可解决。

最后，利用气泡解决控制冒险。当指令为 J/Jal/Jr 或者成功跳转的 Beq/Bne/Bgtz 时，流数线会误取 1 条指令至 IF/ID 流水寄存器，所以需要及时将误取的指令清零。只需将清零信号接至 IF/ID 流水寄存器的同步清零端，在合适的时机清空流水寄存器即可插入气泡（此时无需暂停流水线），其检测逻辑即为单周期 CPU 中 PC 更新逻辑中的信号，即为

flush = (J || Jal || Jr) || (Beq && Eq) || (Bne && ~Eq) || (Bgtz && ~Eq && ~Less)

至此，所有冒险全部解决。

## 动态分支预测机制

通过《计算机组成与设计-硬件/软件接口》一书以及网络上 Princeton、MIT、CMU 有关于动态分支预测的课件的学习与研究，得到了一个较为简易的带 BTB 表的动态分支预测机制。

采用插入气泡以清除误取指令的方式来处理控制冒险会使得 CPU 的效率很低，所以采取分支预测的方式，提前预测分支指令可能的跳转地址。若分支预测的准确性较高（甚至达到 80% 以上），则可以大幅度降低因误取指令而带来的流水线损耗，提升 CPU 处理效率。

动态分支预测策略是通过观察上一次执行分支指令时分支是否发生，来决定此次预测是否执行分支。实现此策略的方法为分支预测缓存（Branch Target Buffer），将之前分支预测的历史保存在一块较小的缓存（全相联存储器）中，当在 IF 阶段遇到分支指令时，先尝试从缓存中读取历史，若读取成功，则根据历史记录进行分支预测；若读取失败，则重新计算地址，采取初始策略（跳转或不跳转），进行分支预测，同时更新 BTB 对应表项。

当预测失败时，只需像之前的设计那样，清除在流水寄存器的误取指令即可，并将 PC 修正为正确的跳转地址。

换言之，带 BTB 的动态分支预期器由 2 大部分组成，预测策略状态机与 BTB 跳转历史缓存。

此次实验采用 2 位的预测策略状态机，其状态变迁如图 2.7 2 位预测状态机变迁图所示。其状态分为 4 个，分别为强跳转（Strongly Taken）、弱跳转（Weakly Taken）、弱不跳转（Weakly Not Taken）、强不跳转（Strongly Not Taken）。根据实际跳转方向，不断修改当前状态。而预测的方向，只于当前状态有关。所以这是一个 Moore 型的状态机。

BTB 表设计为一个全相联映射的缓存，其地址划分如表 2.11 BTB 表项地址划分所示。有效位表示当前表项是否被占据，预测位表示对应分支的预测策略（1 位表示是否跳转，0 位表示策略强度），分支标记保存分支指令的 PC值，用于识别存储于此行的历史属于哪个分支指令，预测地址为对应的跳转地址缓存。

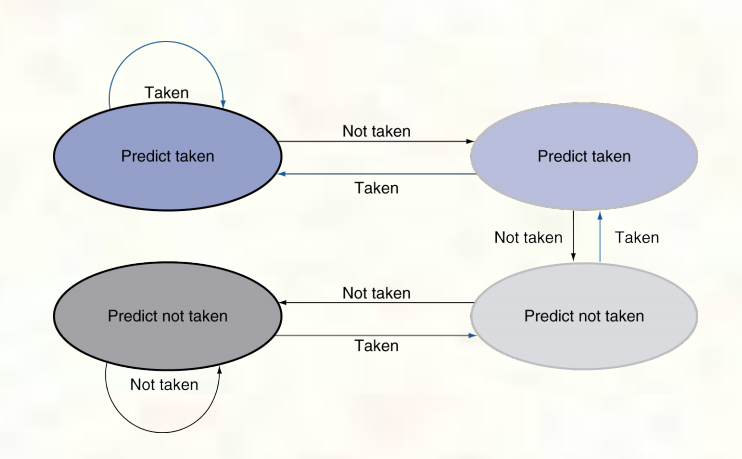


图 2.7 2 位预测状态机变迁图

表 2.11 BTB 表项地址划分

| 1 bit | 2 bit | 32 bit | 32 bit |
| --- | --- | --- | --- |
| 有效位 | 预测位 | 分支标记 | 预测地址 |

# 详细设计与实现

## 单周期CPU 实现

### 主要功能部件实现

1. ALU 实现

Logism实现：

根据前述设计，利用多选器对不同运算进行多选，然后将结果输出即可完成一个简易的 ALU，如图 3.1 ALU结构图（1）、图 3.2 ALU结构图（2）所示。

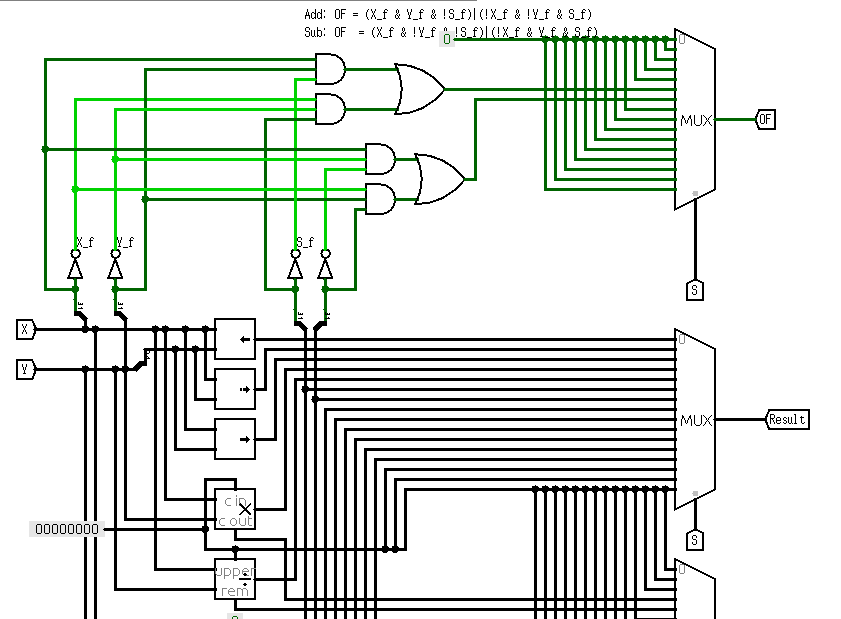


图 3.1 ALU结构图（1）

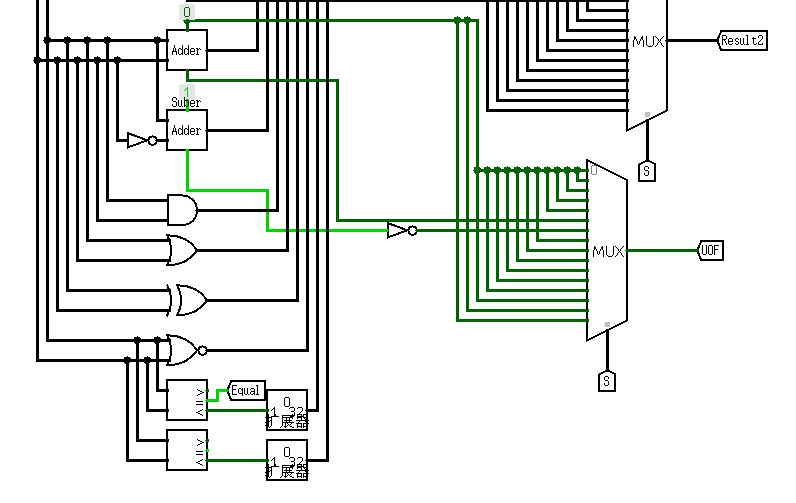


图 3.2 ALU结构图（2）

FPGA实现：

在 Verilog 中，所有变量默认都是 unsigned 型的，但 >>> （算数右移）、<（有符号比较）需要 signed 型的数，故需利用 $signed 任务将无符号变量转化为有符号变量再进行运算。

ALU 的Verilog代码如下：

assign signed\_srcA = $signed(srcA);

assign signed\_srcB = $signed(srcB);

always @ ( \* ) begin

case (aluop)

4'd0: aluout <= srcA << srcB;

4'd1: aluout <= signed\_srcA >>> srcB;

4'd2: aluout <= srcA >> srcB;

4'd3: aluout <= srcA \* srcB;

4'd4: aluout <= srcA / srcB;

4'd5: aluout <= srcA + srcB; // awesome tip

4'd6: aluout <= srcA - srcB;

4'd7: aluout <= srcA & srcB;

4'd8: aluout <= srcA | srcB;

4'd9: aluout <= srcA ^ srcB;

4'd10: aluout <= ~(srcA | srcB);

4'd11: aluout <= (signed\_srcA < signed\_srcB) ? 1 : 0;

4'd12: aluout <= (srcA < srcB) ? 1 : 0;

default: aluout <= 0;

endcase

end

1. Regfile 实现

Logism实现：

根据前述设计，将输入部分与输出部分利用复用器与解复用器进行分离，实现对不同寄存器的写与读，再结合寄存器阵列，即可实现一个包含 32 个寄存器的通用寄存器组。写入使能选择电路如图 3.3 使能信号选择电路所示，用于控制寄存器写；输出部分电路如图 3.4 寄存器组输出部分所示，用于控制寄存器读；寄存器组完整电路如图 3.5 寄存器组完整电路所示。

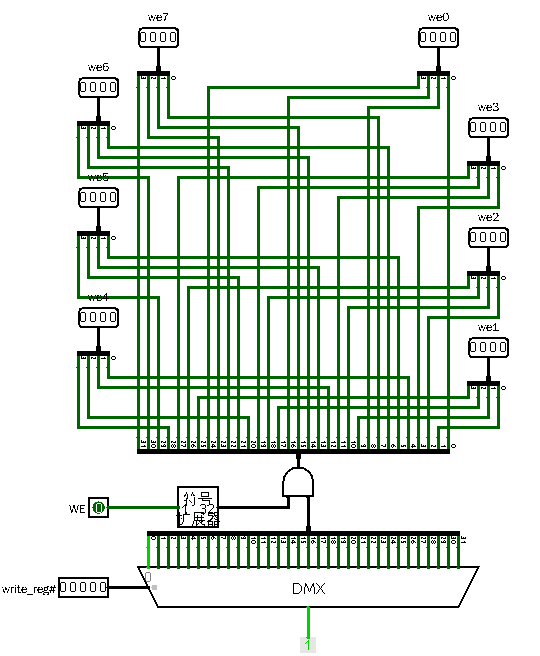


图 3.3 使能信号选择电路

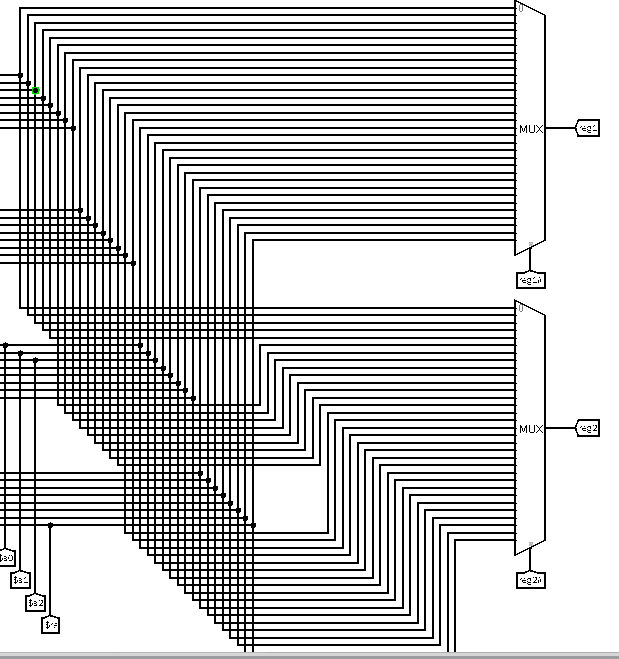


图 3.4 寄存器组输出部分

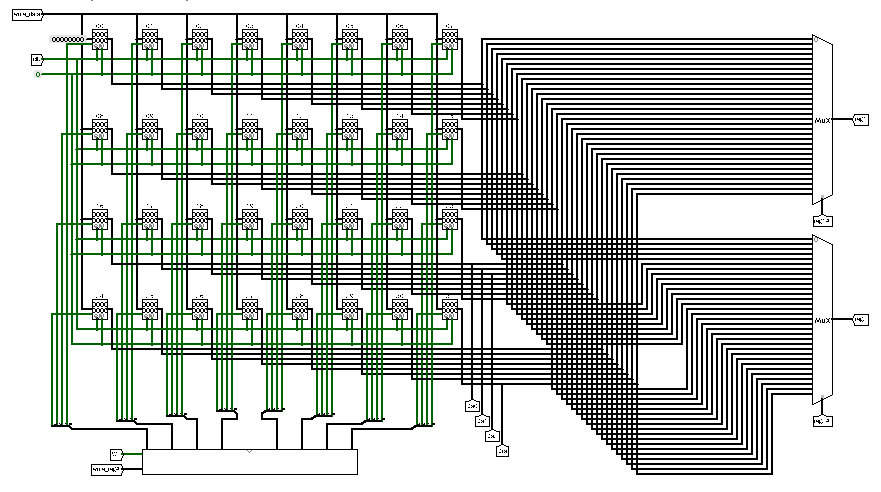


图 3.5 寄存器组完整电路

FPGA实现：

Verilog 实现较为简单，直接利用 Reg 型的数组即可实现寄存器组。

其 Verilog代码如下：

reg [DATA\_WIDTH-1:0] regfile [0:31]; ///< three ported regfile contains 32 registers

always @ (posedge clk) begin

if (rst) begin

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

begin

regfile[i] <= 0;

end

end else if (we && waddr != 0) begin

regfile[waddr] <= wdata;

end

end

assign regA = (raddrA != 0) ? regfile[raddrA] : 0;

assign regB = (raddrB != 0) ? regfile[raddrB] : 0;

1. 指令译码器

Logism实现：

根据 MIPS 3种类型的指令，将指令的各个位段提取出来，即可完成指令译码器，输出的位段有 op（6位）、rs（5位）、rt（5位）、rd（5位）、sham（5位）、funct（6位）、imm16（16位）、imm26（26位）。如图 3.6 指令译码电路图所示

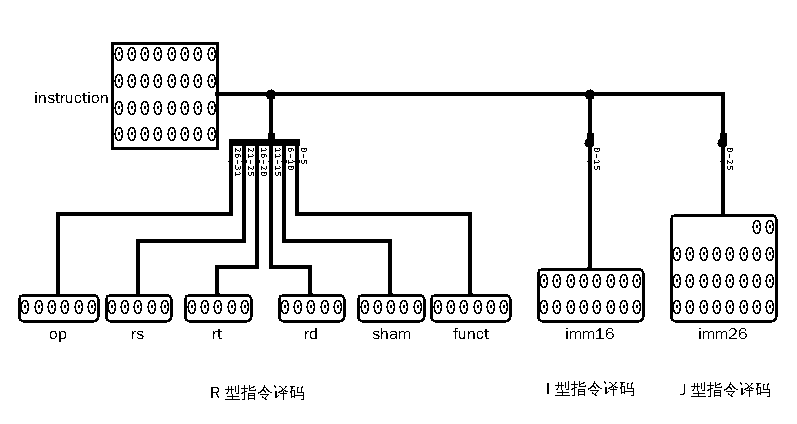


图 3.6 指令译码电路图

FPGA实现：

利用位操作符即可分离信号，其Verilog代码如下：

assign op = instruction[`OP];

assign rs = instruction[`RS];

assign rt = instruction[`RT];

assign rd = instruction[`RD];

assign sham = instruction[`SHAM];

assign funct = instruction[`FUNCT];

assign imm16 = instruction[`IMM16];

assign imm26 = instruction[`IMM26];

1. PC 更新逻辑

Logism实现：

按照前述设计，利用多个多选器，根据当前译码信号 J/Jal/Jr/Beq/Bne/Bgtz以及地址计算结果 Eq/Less，生成当前跳转方式的表达式，如表 2.2 PC 更新逻辑表所示。以相对跳转方式为例，需要将 PC 按此种方式更新的逻辑表达式为

Branch = (Beq && Eq) || (Bne && ~Eq) || (Bgtz && ~Eq && ~Less)。

其具体实现如图 3.7 PC 更新逻辑单元所示。

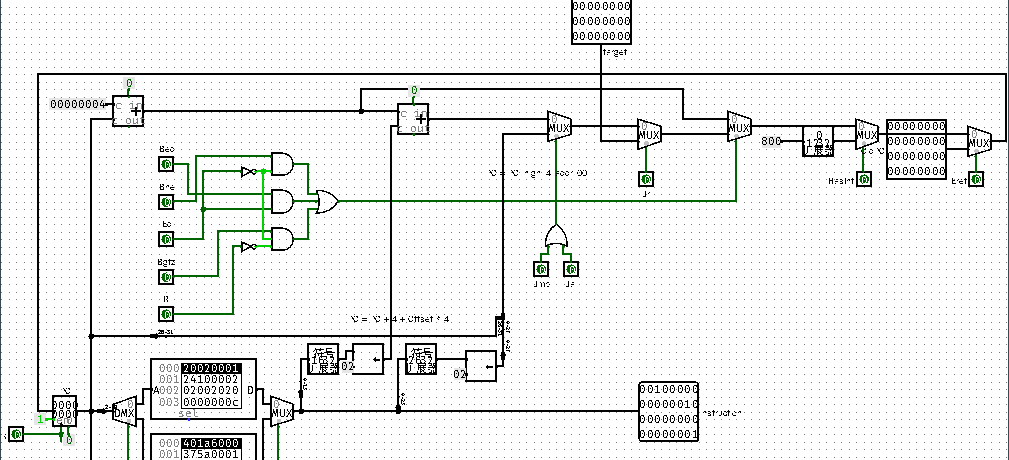


图 3.7 PC 更新逻辑单元

FPGA实现：

利用三元表达式（?:），可以快速地实现 PC 更新逻辑。首先利用控制单元生成的信号得到跳转信号，再计算跳转地址，根据跳转信号选择合适的跳转地址送至 PC 寄存器的输入端即可。

其Verilog代码如下：

assign jmp\_imm = j || jal;

assign jmp\_reg = jr;

assign jmp\_branch = (beq && eq) || (bne && ~eq) || (bgtz && ~eq && ~less);

assign addr\_reg = r1;

assign extshft\_imm16 = {{(DATA\_WIDTH-16){imm16[15]}}, imm16} << 2;

assign extshft\_imm26 = {{(DATA\_WIDTH-26){imm26[25]}}, imm26} << 2;

assign addr\_imm = {pc[31:28], extshft\_imm26[27:0]};

assign addr\_branch = extshft\_imm16 + pc + 4;

assign pc\_next = jmp\_reg ? addr\_reg

: jmp\_imm ? addr\_imm

: jmp\_branch ? addr\_branch

: (pc + 4);

1. ROM 和 RAM

Logism实现：

直接利用 Logisim 内建组件即可。

FPGA实现：

无论是 ROM 还是 RAM，实现思路都与 Regfile 类私，利用 Reg 型的数组来模拟连续内存空间即可。

ROM的Verilog代码如下：

reg [DATA\_WIDTH-1:0] ROM [0:(2\*\*BUS\_WIDTH)-1];

assign rdata = ROM[addr];

RAM的Verilog代码如下：

reg [DATA\_WIDTH-1:0] RAM [0:(2\*\*BUS\_WIDTH)-1];

always @ (posedge clk) begin

if (we) begin

RAM[addr] <= wdata;

end

end

assign rdata = re ? RAM[addr] : {(DATA\_WIDTH-1){1'bx}};

### 数据通路的实现

本次课程设计为采用工程化的方式实现数据通路，而是逐步在原有数据通路的基础上整合新的数据通路。其实现过程如下所示：

常规 R 型指令数据通路：按照前述设计，此时只需 2 个控制信号 ，RegWe 控制寄存器组的写使能，ALUOp 控制 ALU 运算逻辑，此类指令包括 add、addu、and、sub、or、nor、slt、sltu，其数据通路如图 3.8 常规 R 型指令数据通路图所示。

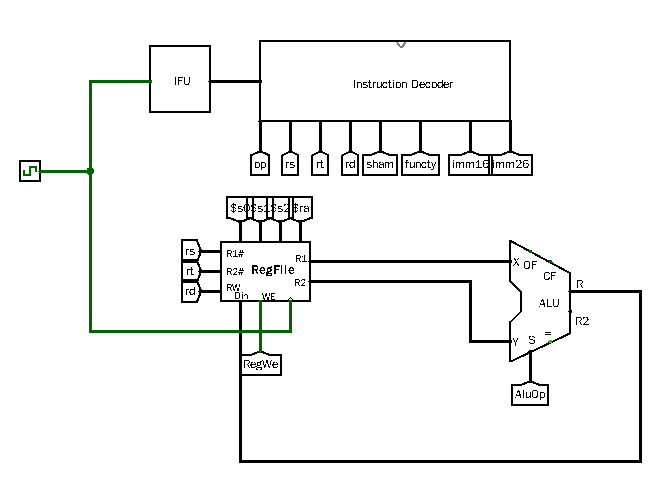


图 3.8 常规 R 型指令数据通路图

常规I型数据通路：按照前述设计，利用 3 个新的控制信号，RegDst 选择写入寄存器编号（rt 或 rd），ALUSrc 选择 Y 端输入（rt 或 imm16），ExtOp 选择立即数扩展方式（无符号或有符号扩展），此类指令包括 addi、addiu、andi、ori、slti。其具体通路如图 3.9 常规 I 型指令数据通路图所示。

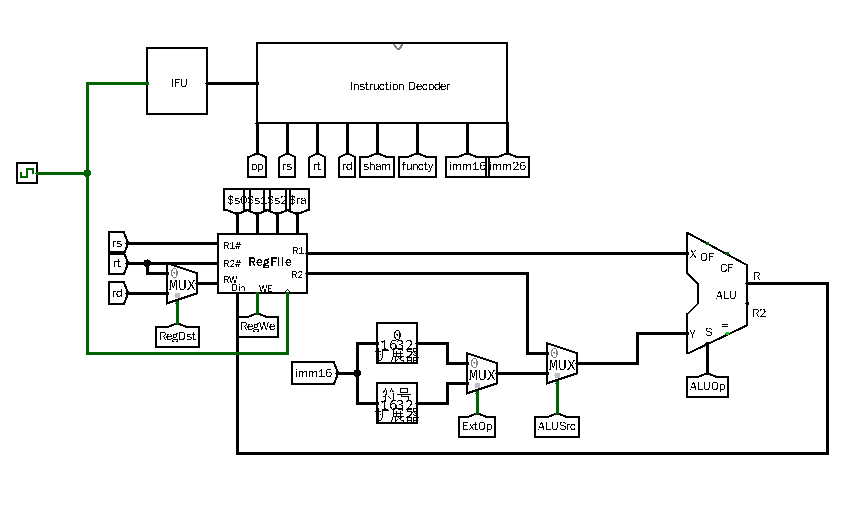


图 3.9 常规 I 型指令数据通路图

移位指令数据通路：按照前述设计， 1 个新的信号 ALUSham，以改变 ALU 2 个输入端的值，其通路如图 3.10 移位指令数据通路图所示。

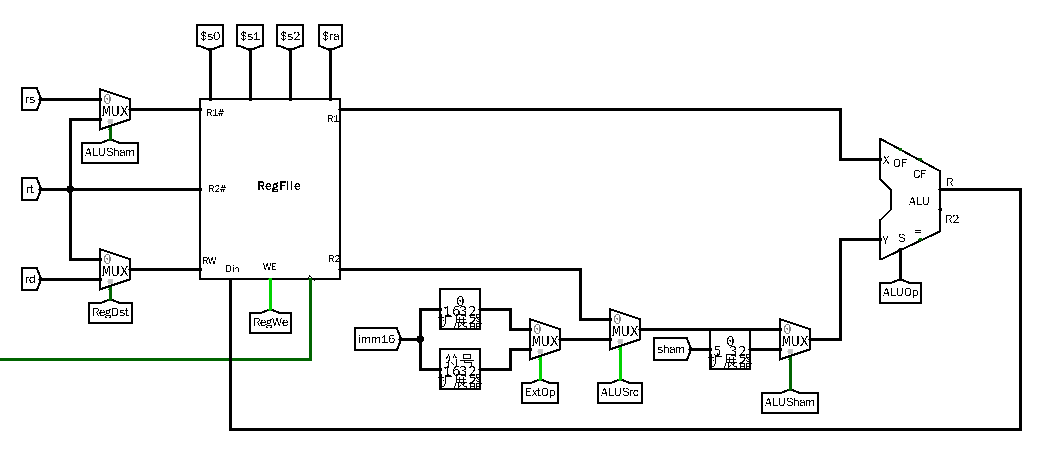


图 3.10 移位指令数据通路图

访存指令数据通路：按照前述设计，2 个新的控制信号，以控制主存的读写。对于lw 指令，需要从主存读取数据至寄存器，增加新信号 RAMtoReg，以选择寄存器组的输入数据（ALU\_Result 或 RAMDataOut）；对于 sw 指令，需要增加 1 个新信号 RAMWe，以控制对主存的写访问，并将 （rt）接至主存加载数据端（RAMDataIn）。其具体通路如图 3.11 访存指令数据通路图所示。

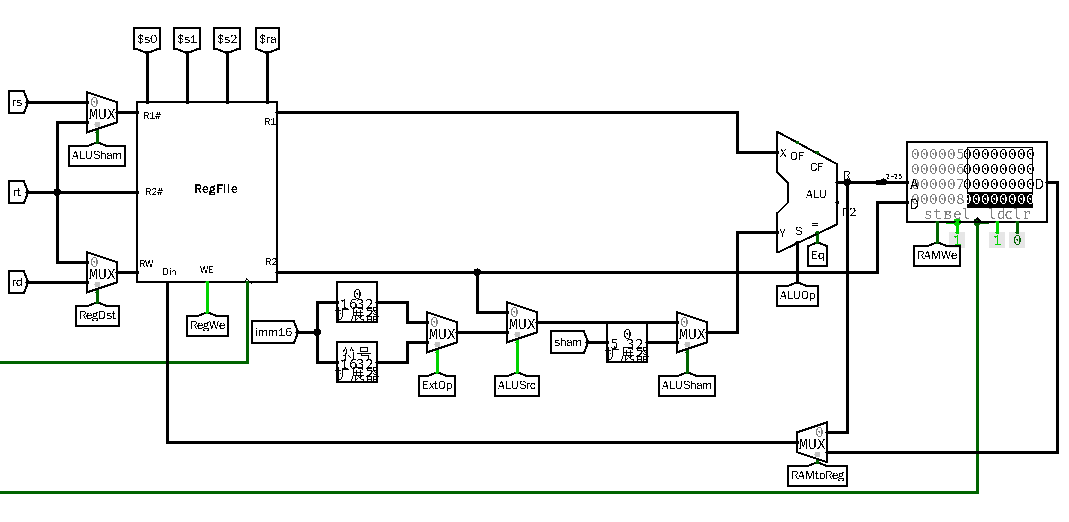


图 3.11 访存指令数据通路图

跳转指令：j/jal/jr/beq/bne/bgtz 指令需要按照图 3.7 PC 更新逻辑单元所示，修改 PC 更新逻辑，在合适的是否选择合适的新 PC 值更新 PC 寄存器。除此之外，jal 指令还需借助原数据通路将 rs 值取出加 4 后写入 $a0 寄存器，此时 ALU\_X = （rs），ALU\_Y = （4），ALU\_OP = ADD，RW# = 0x1f，故需修改数据通路，如图 3.12 jal 指令数据通路图所示。

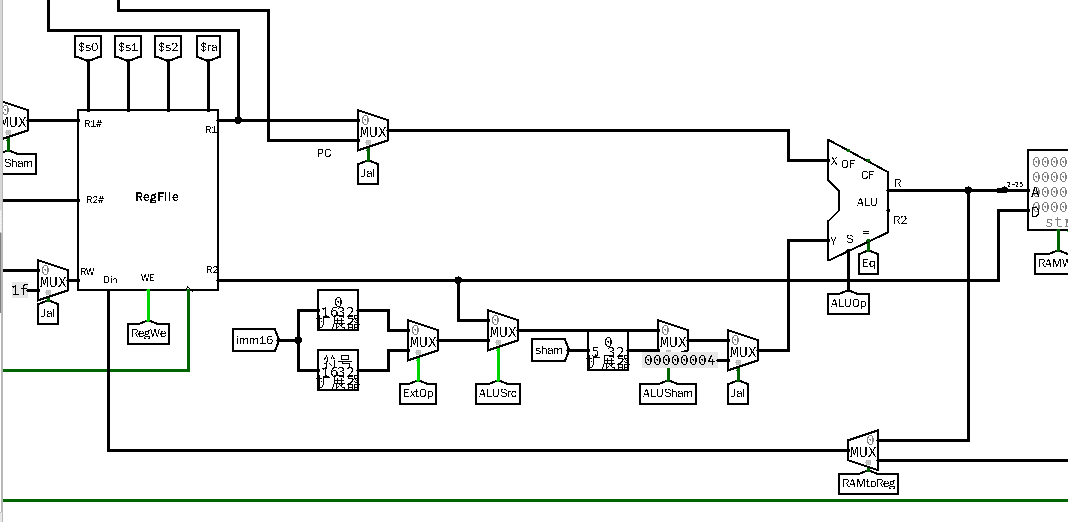


图 3.12 jal 指令数据通路图

关于 syscall 的实现，无需修改数据通路，只需将 $v0 的值从寄存器组引出，并将其与 0xah 进行比较，再结合 Syscall 指令布尔信号，即可得到 Halt 停机信号与输出七段管信号。其具体实现如图 3.13 syscall 实现电路图所示。

利用 Halt信号，对时钟源进行简单处理，即可实现 syscall 0xah 功能，停机功能具体实现如图 3.14 利用 syscall 模块生成的 halt 信号处理时钟源所示。

两张图中的计数器均为达到最大值则停止计数，实现锁存。

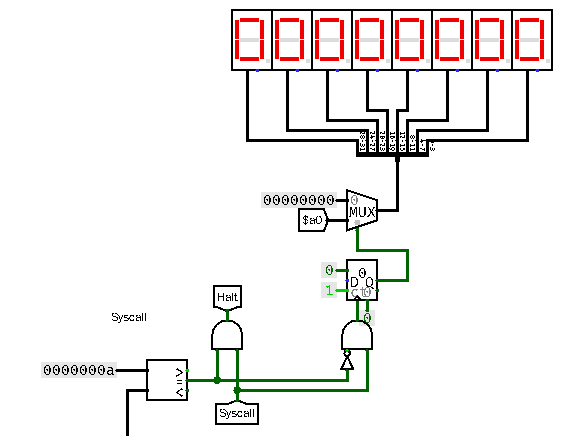


图 3.13 syscall 实现电路图

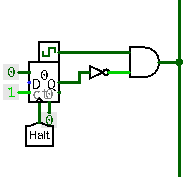


图 3.14 利用 syscall 模块生成的 halt 信号处理时钟源

扩展指令数据通路：divu 需要增加一个 LO 寄存器与一个 WriteToLO 信号（使能控制信号，将 ALU 除的结果在合适的时机写入 LO 寄存器），需将 ALUOp 译码成除法操作；mflo 需要修改写回寄存器的数据通路，额外增加一个控制信号，将写回寄存器的数据修改为 LO 寄存器的值；;lb 指令只需将 RAM 读出的数据进行字节选择并进行符号扩展，得到新的一个 32 bit 的数据，再利用控制信号 RAMByte在原来 RAM 数据输出和新数据间进行多选即可。divu 与 mflo 数据通路如图 3.15 divu 与 mflo 数据通路图所示，lb 数据通路如图 3.16 lb 数据通路图所示。

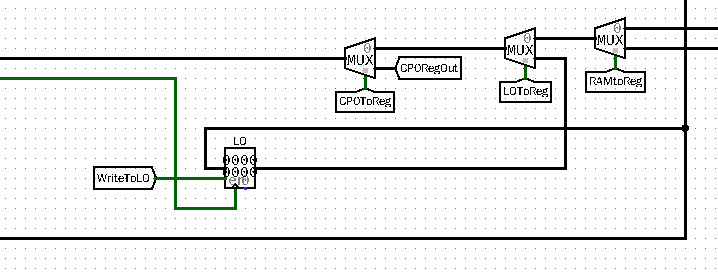


图 3.15 divu 与 mflo 数据通路图

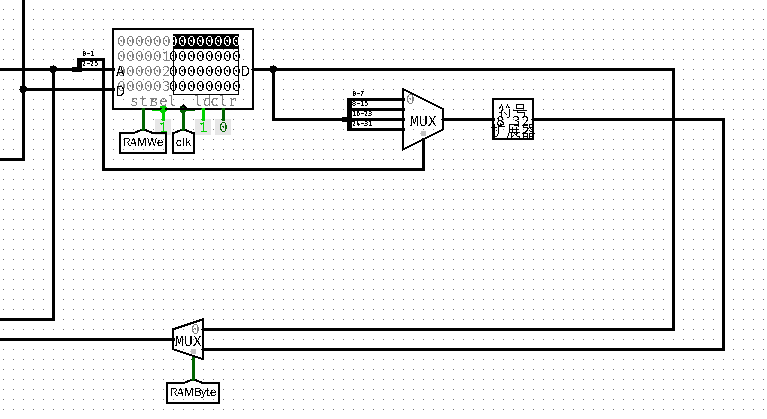


图 3.16 lb 数据通路图

### 控制器的实现

设计好数据通路，即可按照指令特性与数据通路走向，确定对于每一个指令的控制信号集。控制信号结果如表 2.6 控制信号表的框架（1）、表 2.7 控制信号表的框架（2）、表 2.8 控制信号表的框架（3）、表 2.9 控制信号表的框架（4）所示。利用多路或门，即可实现所有控制信号的生成，从而实现完整的控制单元，其电路如图 3.17 控制单元电路图（1）、图 3.18 控制单元电路图（2）所示。

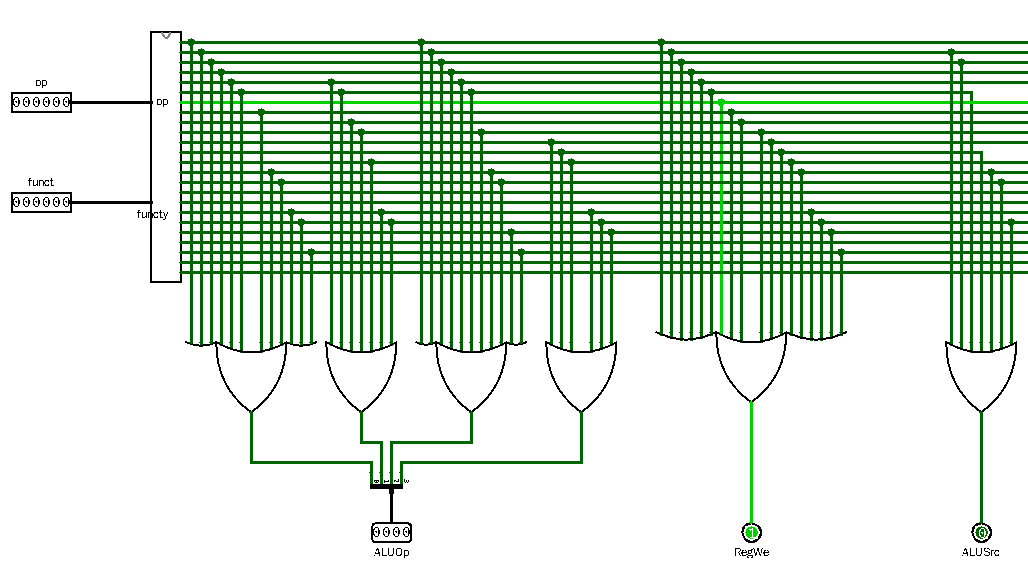


图 3.17 控制单元电路图（1）

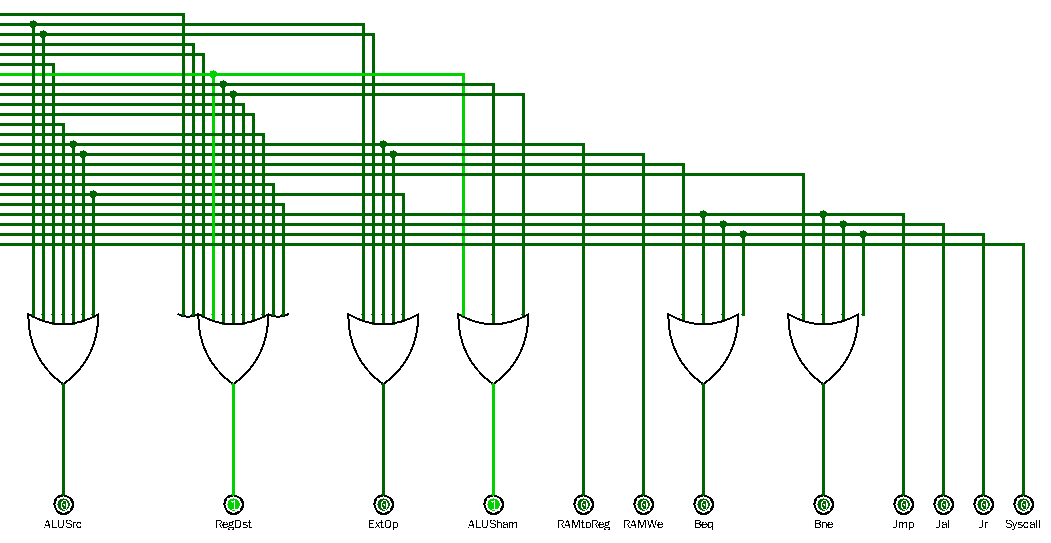


图 3.18 控制单元电路图（2）

根据在Logism实现中得到的各个一位控制信号的表达式，直接使用数据流建模，使用assign分的Verilog代码过于冗长，故只取对于控制信号RegWe的生成代码举例如下：

assign regwe = add || addi || addiu || addu || s\_and || andi

|| sll || sra || srl || sub || s\_or || ori || s\_nor || lw

|| slt || slti || sltu || jal || mflo || lb;

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语言构成的主控制器原理图如图 3.19 主控制器原理图所示。

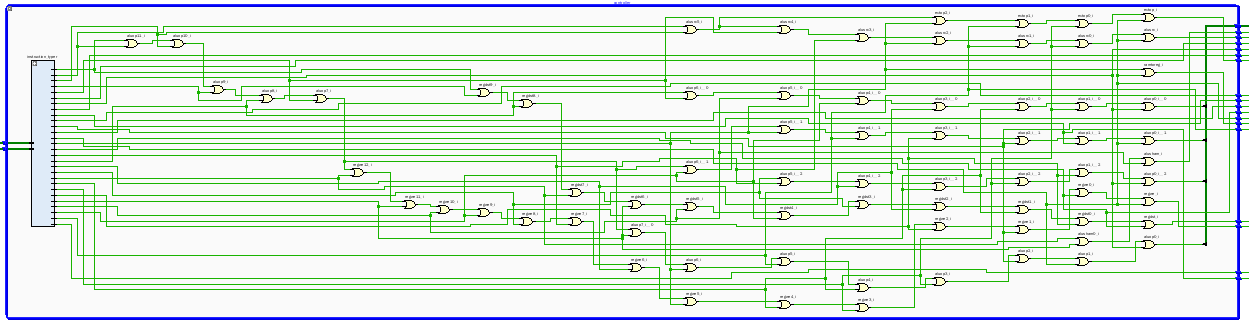


图 3.19 主控制器原理图

## 中断机制实现

### 中断识别与响应电路

按照如图 2.3 中断识别与响应电路所示原理图，在 Logisim 上实现等价电路，并此基础上增加中断清零逻辑，即可完成中断识别与响应电路，如图 3.20 中断识别与响应电路所示。当有中断请求后，通过优先编码器的作用可以得到中断请求原始信号与中断号，中断号会送 Cause 寄存器。而作为输入的中断屏蔽字与中断使能位由 Status 寄存器确定。图中的复用器与解复用器实现对指定中断进行清除操作，当有中段被响应后，复用器与解复用器的循环会根据当前中断号，同步清零中断请求寄存器，其输出反过来清零右下角的寄存器，使得清零信号也被清零，达到稳态。

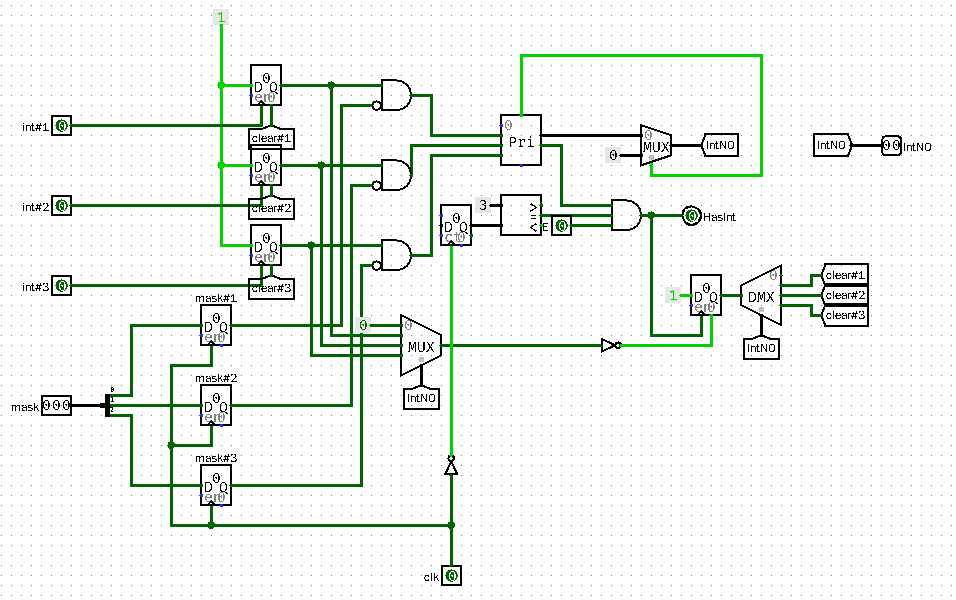


图 3.20 中断识别与响应电路

### CP0 协处理器

按照前述设计，如图 2.4 简易 CP0 示意图所示，利用实现通用寄存器组的原理，实现一个简易的只含 3 个寄存器（Status、EPC、Cause）的 CP0 协处理器，如图 3.21 简易CP0协处理器实现所示。三个数据输入，分别为通用寄存器的值 Datain、旧的 PC以及中断号，三个控制输入，HasInt 表示当前有中断发生，此时 EPC 的输入改为 PC 寄存器，用于保存断点，rd 用于控制输入与输出的目标寄存器，以实现 mfc0与 mtc0 指令，clk 为时钟源．四个输出，其中 Status 的0 位与 8-10 位分别作为中断使能位与中断屏蔽字接入中断识别与响应电路，而OldPC用于接入 PC 更新单元用于实现 Eret，CP0RegOut接入经多选后通用寄存器组的数据输入端用于实现 mfc0。利用以上逻辑，便可实现 mfc0、mtc0与eret三条指令，为中断实现软件支撑。

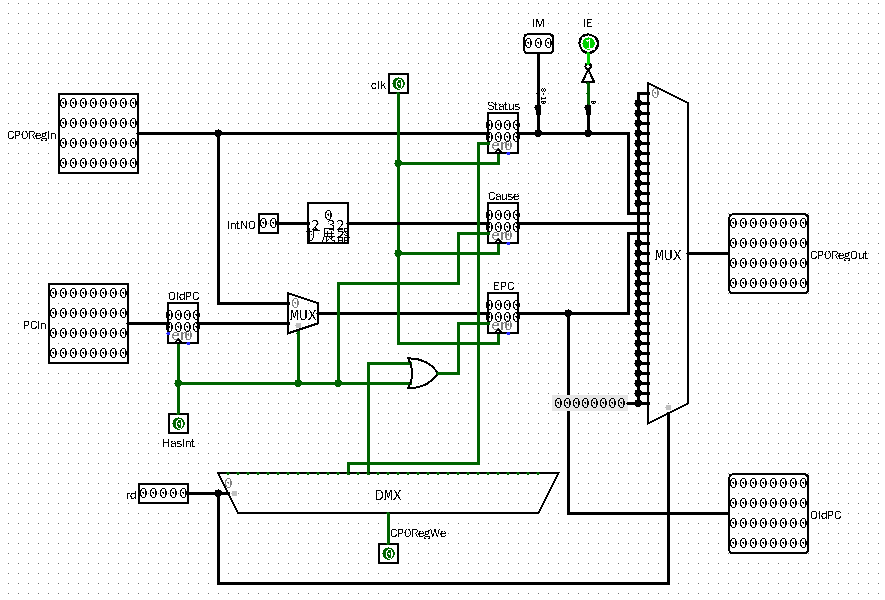


图 3.21 简易CP0协处理器实现

### PC 更新

按照前述设计，修改 PC 更新单元，额外利用多选器实现中断跳转与中断返回，如图 3.22 修改 PC 更新逻辑所示。HasInt 表示当前有中断发生，PC跳至 0x800，开始处理中断；Eret 信号表示当前有中断返回，PC被修改为 EPC 中保存的断点，返回原用户程序继续执行。

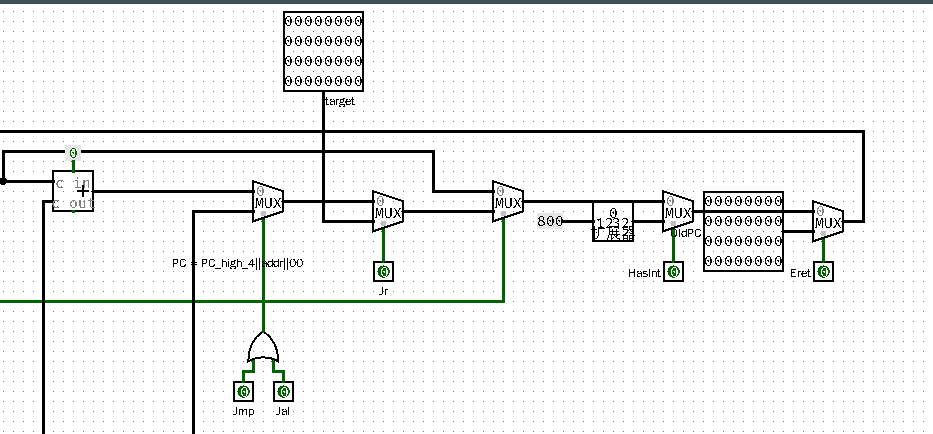


图 3.22 修改 PC 更新逻辑

完成以上 3 大单元，即可实现多级嵌套中断的硬件布局，其中中断识别与响应电路和 CP0电路之间的耦合如图 3.23 完整中断电路所示。

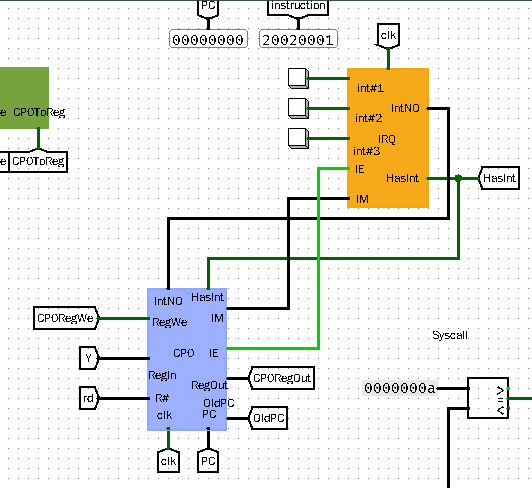


图 3.23 完整中断电路

完成硬件实现后，只需编写中断处理程序，根据当前断点，利用软件分支来实现不同的中断处理例程。利用 mtc0 xx, $12来开启/关闭中断，利用 lw/sw xx, $fp，实现现场的保护（压栈）与恢复（出栈），利用 mfc0 xx, $14得到中断号，利用 beq/bne 实现分支，根据中断号跳转不同分支，对 3 类中断进行不同处理，完整的中断处理程序见附件代码 single\_cycle\_interrupt.asm。

## 流水CPU实现

### 流水接口部件实现

根据前述设计，简单地将数据与信号通过寄存器锁存即可实现流水接口部件，所有 4 个流水寄存器所需的数据与信号皆在设计部分已给出。下面以最复杂的 ID/EX 流水寄存器为例，讲述 Logisim 与 FPGA实现。

其 Logisim 实现十分简单，只需利用内建的寄存器，将不同的数据与信号作为寄存器的输入与输出即可，如图 3.24 ID/EX流水寄存器所示。

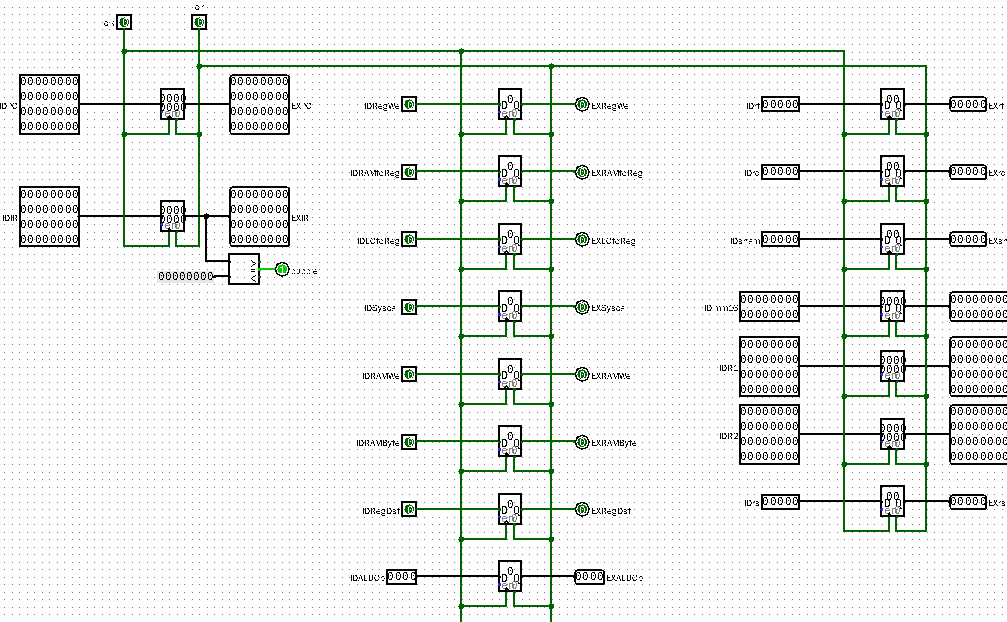


图 3.24 ID/EX流水寄存器

其 FPGA实现也十分简单，首先实现一个带 parameter DATA\_WIDTH（数据位宽）的同步清零寄存器，然后在 ID\_EX 流水寄存器模块不断地修改 DATA\_WIDTH实例化不同长度（数据 5-32位不等，信号1位）的寄存器对信号与数据进行锁存即可，其关键 Verilog 代码如下所示：

module register #(parameter DATA\_WIDTH = 32)

(

input clk,input rst,input en,input [DATA\_WIDTH-1:0] din,

output reg [DATA\_WIDTH-1:0] dout

);

always @ (posedge clk) begin

if (rst) begin

dout <= 0; // reset

end else if (en) begin

dout <= din; // update

end else begin

dout <= dout; // hold

end

end

endmodule // register

### 理想流水线实现

实现完所有 4 个流水寄存器后，按照如图 2.6 理想流水线原理图所示，将整个数据通路串联起来，如图 3.25 理想流水线实现所示。

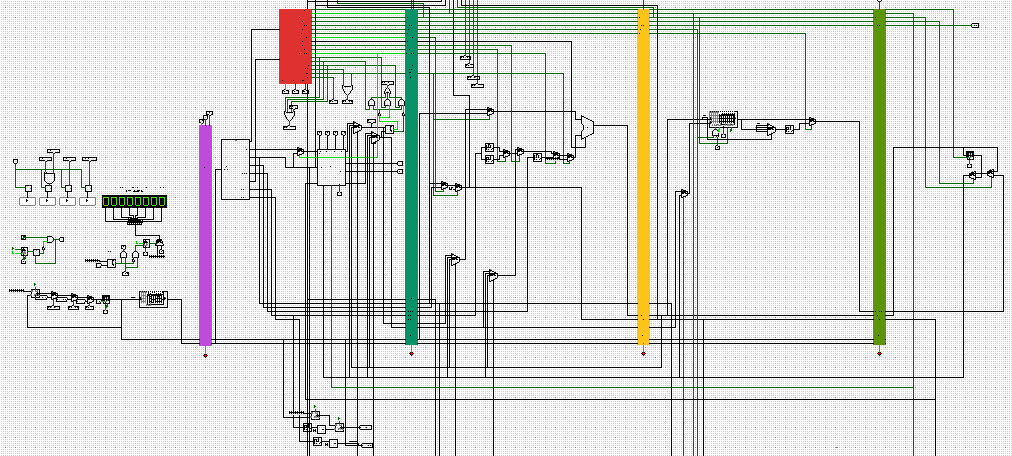


图 3.25 理想流水线实现

## 数据转发流水线实现

根据前述2.4设计，将 3 个阶段的转发信号逻辑表达式（if else 分支判断）转化成电路图，其中 ID 段转发信号如图 3.26 ID转发信号图所示，EX 段转发信号如图 3.27 EX 段转发信号图所示，MEM段转发信号如图 3.28 MEM 段转发信号图所示。修改原有数据通路，如图 3.29 ID 段转发通路、图 3.30 EX 段转发通路、图 3.31 MEM 段转发通路所示。

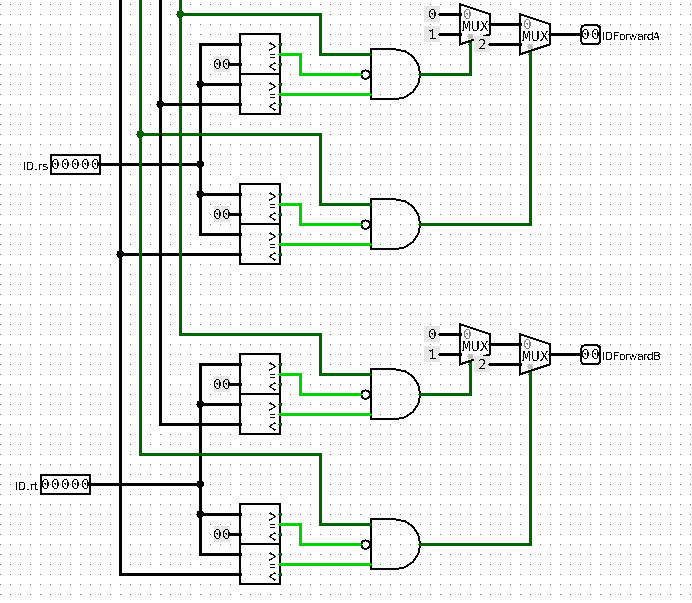


图 3.26 ID转发信号图

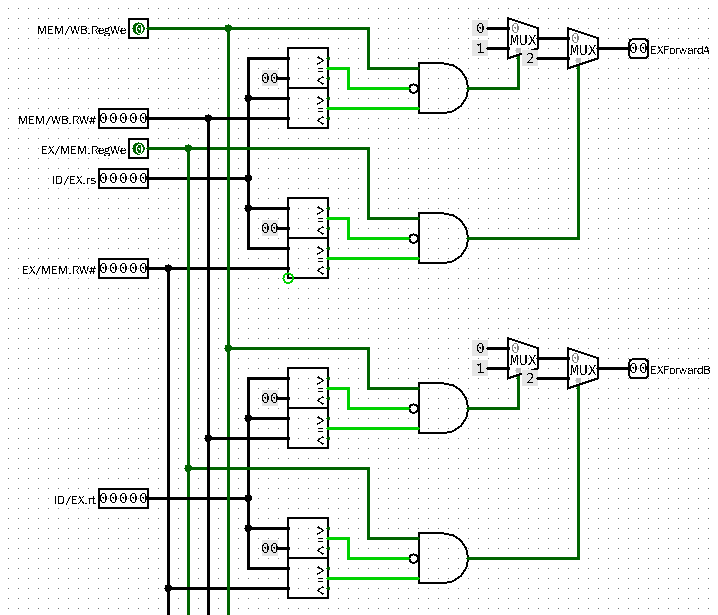


图 3.27 EX 段转发信号图

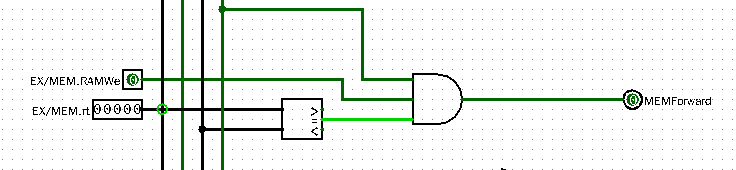


图 3.28 MEM 段转发信号图

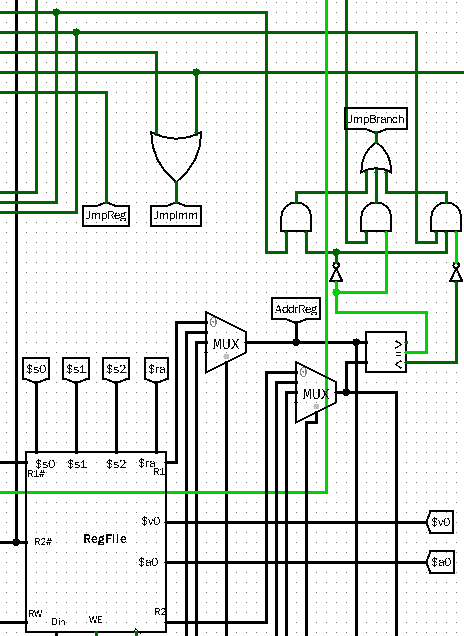


图 3.29 ID 段转发通路

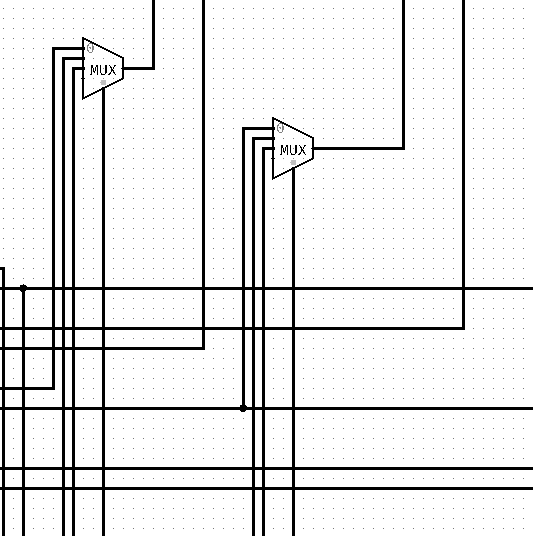


图 3.30 EX 段转发通路

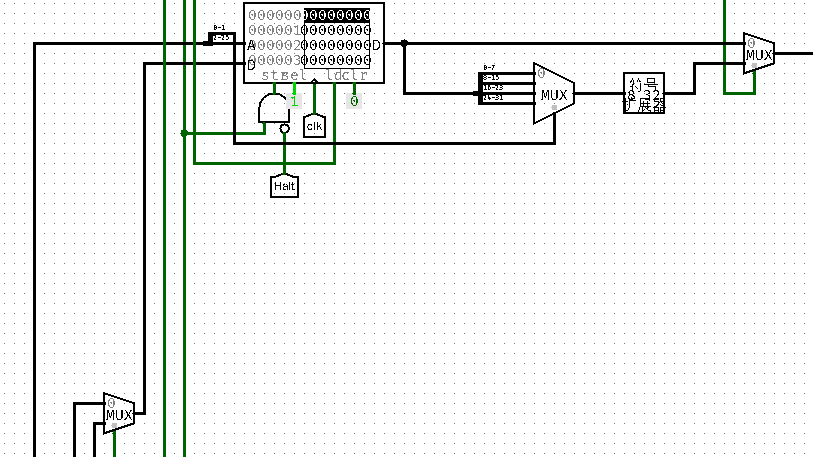


图 3.31 MEM 段转发通路

以 ID R1的转化信号为例，其转化为 Verilog 代码如下所示：

always @ ( \* ) begin

if ((ID\_rs != 0) && (ID\_rs == MEM\_RW) && MEM\_regwe) begin

ID\_forwardA <= 2'b10; // from MEM stage

end else if ((ID\_rs != 0) && (ID\_rs == WB\_RW) && WB\_regwe) begin

ID\_forwardA <= 2'b01; // from WB stage

end else begin

ID\_forwardA <= 2'b00; // no forwarding

end

end

## 气泡式流水线实现

按照前述2.5设计，实现 load-use 冒险检测模块，产生 lwstall 信号，向流水线插气泡，如图 3.32 load-use冒险检测模块所示；实现 branch 冒险检测模块，产生 branchstall 与 flushD信号，向流水线插气泡与清空误取指令，如图 3.33 分支数据冒险检测模块、图 3.34 分支控制冒险检测模块所示。

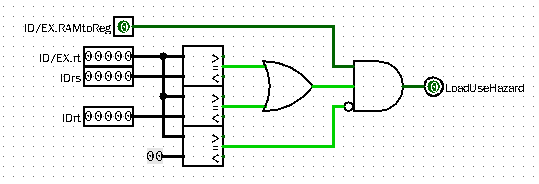


图 3.32 load-use冒险检测模块

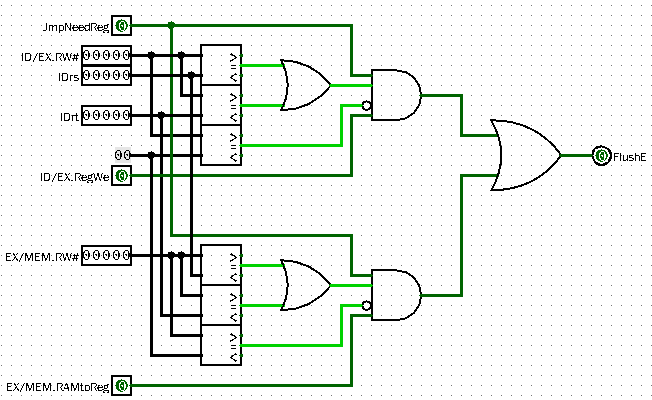


图 3.33 分支数据冒险检测模块

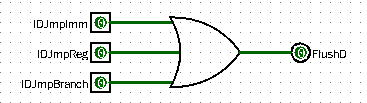


图 3.34 分支控制冒险检测模块

其转化为 Verilog代码如下：

assign load\_use\_hazard = EX\_rt != 0 && EX\_ramtoreg && (EX\_rt == ID\_rs || EX\_rt == ID\_rt);

assign branch\_flushD = ID\_jmp\_reg || ID\_jmp\_imm || ID\_jmp\_branch;

assign branch\_flushE = (ID\_jmp\_need\_reg && EX\_regwe && EX\_RW != 0 && (EX\_RW == ID\_rs || EX\_RW == ID\_rt))

|| (ID\_jmp\_need\_reg && MEM\_ramtoreg && MEM\_RW != 0 && (MEM\_RW == ID\_rs || MEM\_RW == ID\_rt));

## 动态分支预测机制实现

按照前述设计，此部分实现分为 2 大部分，一个为二位预测器的 Moore 型状态机实现，一个为全相联 BTB表的实现（实际上为 Cache的实现）。

按照如图 2.7 2 位预测状态机变迁图所示，实现一个 Moore型状态机，其 Verilog代码如下所示：

case (predict\_bits[ID\_access\_line])

`STRONGLY\_TAKEN:

case (misprediction)

0:predict\_bits[ID\_access\_line] <= `STRONGLY\_TAKEN;

1: predict\_bits[ID\_access\_line] <= `WEAKLY\_TAKEN;

endcase

`WEAKLY\_TAKEN:

case (misprediction)

0: predict\_bits[ID\_access\_line] <= `STRONGLY\_TAKEN;

1: predict\_bits[ID\_access\_line] <= `WEAKLY\_NOT\_TAKEN;

endcase

`WEAKLY\_NOT\_TAKEN:

case (misprediction)

0: predict\_bits[ID\_access\_line] <= `STRONGLY\_NOT\_TAKEN;

1: predict\_bits[ID\_access\_line] <= `WEAKLY\_TAKEN;

endcase

`STRONGLY\_NOT\_TAKEN:

case (misprediction)

0: predict\_bits[ID\_access\_line] <= `STRONGLY\_NOT\_TAKEN;

1: predict\_bits[ID\_access\_line] <= `WEAKLY\_NOT\_TAKEN;

endcase

endcase

第二步，完成相联比较器，利用多个比较电路，再利用一个 always 块进行编码，即可完成相联比较并得到目标行号，此模块的输出为hit与hit\_line，分别表示是否命中，以及命中时的行号其关键Verilog代码如下：

assign hit0 = valid0 && (src\_tag == tag0);

assign hit1 = valid1 && (src\_tag == tag1);

assign hit2 = valid2 && (src\_tag == tag2);

assign hit3 = valid3 && (src\_tag == tag3);

assign hit4 = valid4 && (src\_tag == tag4);

assign hit5 = valid5 && (src\_tag == tag5);

assign hit6 = valid6 && (src\_tag == tag6);

assign hit7 = valid7 && (src\_tag == tag7)

利用相联比较器的比较结果，可以得到此次访问的实际 BTB表行号，access\_line = hit ? hit\_line ：lru\_line。若此次访问命中，则从 BTB 中取出跳转地址；若不命中，则更新表项。除了在 IF 段访问时有可能修改表项，当 ID 段检测到预测错误时，也会更新表项，其关键 Verilog代码如下：

if (~IF\_hit && IF\_branch) begin

valid[IF\_access\_line] <= 1;

predict\_bits[IF\_access\_line] <= `WEAKLY\_TAKEN;

branch\_tags[IF\_access\_line] <= IF\_branch\_pc;

target\_PCs[IF\_access\_line] <= IF\_predict\_addr;

end

if (ID\_branch) begin

if (~ID\_hit) begin

valid[ID\_access\_line] <= 1;

predict\_bits[ID\_access\_line] <= `WEAKLY\_TAKEN;

branch\_tags[ID\_access\_line] <= ID\_branch\_pc;

target\_PCs[ID\_access\_line] <= ID\_branch\_addr;

end

得到最终的预测输出如下所示，即实现了 BTB表的基本逻辑：

assign taken = IF\_hit ? predict\_bits[IF\_access\_line] : `WEAKLY\_TAKEN;

assign btb\_branch\_addr = IF\_hit ? target\_PCs[IF\_access\_line] : IF\_predict\_addr;

除此之外，还需实现预测错误修正逻辑，当预测出错时，需要进行 2 项工作，其一为清除误取指令，修改 flushD的逻辑表达式即可，其二为修正跳转地址，修改 PC 更新逻辑即可。其关键 Verilog代码如下所示：

assign branch\_flushD = ID\_jmp\_reg || ID\_misprediction;

assign IF\_mispredict\_fix\_addr = ID\_jmp\_branch ? ID\_addr\_branch : (ID\_pc + 4);

assign IF\_pc\_next = ID\_jmp\_reg ? ID\_addr\_reg

: ID\_misprediction ? IF\_mispredict\_fix\_addr

: IF\_predict\_addr;

最后是关于 LRU（Least Recently Used）算法的实现，出现了一定问题，基本思路为给每一行额外添加 3 bit位，用于表示当前行的引用计数，每当BTB表被访问后，被访问行计数清零，其余行计数+1，计数到 8 时便停止计数。下一次再次访问BTB表时，若需要淘汰某一行时，选择引用计数最大的一行进行替换（表示其最久未被访问）即可。遗憾地是，进行硬件实现的时候，出现了一些未知故障，在验收期前并未将其调通。于是导致当前的BTB表只能有效地利用1行进行跳转历史的保存，真是尴尬。LRU算法的实现详见 lru\_counter.v模块。

# 实验过程与调试

## 测试用例和功能测试

### 测试用例1 ：扩展指令测试

XXXXX……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是

### 测试用例2：中断测试

XXXXX……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是

### 测试用例2：5段流水线测试

XXXXX……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是……(请自行扩展修改)指令的格式设计是……指令的寻址方式设计是

## 性能分析

**分析不同方案时钟周期数差异**,

## 主要故障与调试

### bgtz故障

Logisim 单周期 CPU：bgtz 实现错误。

**故障现象：**执行bgtz 测试时（如图 4.1 bgtz 测试用例所示)，程序会陷入死循环。

**原因分析：**经过比对 Mars 与 Logisim 之间执行过程，可以发现当执行第 28 条语句 bgtz $s1, not\_jmp 时，出现了错误。CPU 错误地进行了此次跳转，可以推出应该是跳转信号出了问题，如图 4.2 错误的跳转信号所示

**解决方案：**修改跳转信号逻辑为 branch\_jmp = Beq & Eq + Bne & ~Eq + Bgtz & ~R & ~Eq，如图 4.3 正确的跳转信号所示，即可使得 bgtz 正确地只在大于 0 时跳转，而不会错误地在等于 0 时也进行跳转。

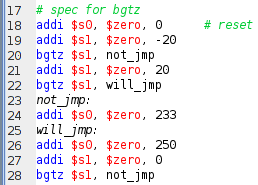


图 4.1 bgtz 测试用例

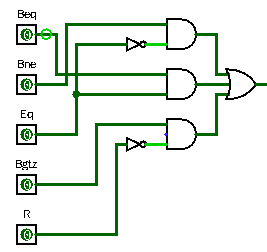


图 4.2 错误的跳转信号

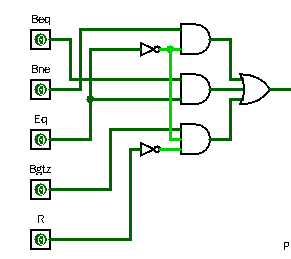


图 4.3 正确的跳转信号

### lb故障

Logisim 单周期 CPU：lb 实现错误。

**故障现象：**z执行如图 4.4 lb 测试用例所示程序时，发现 CPU 无法读取每个字的第1、2、3个字节，只可读取地址为 4 的倍数的字节。

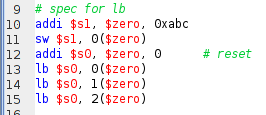


图 4.4 lb 测试用例

**原因分析：**仔细阅读指令集说明后，发现遗漏了关键信息。指令集说明上提示，需要将地址的最低 2 位作为字节选择信号。由于 MIPS 内存是按字编址的，故需要利用此方法，将读出的某个字分成 4 个字节，利用字节选择信号选择不同的字节，送入指令指定寄存器。

**解决方案：**修改内存读取逻辑，将地址线的最低 2 位作为 4 选 1 多选器的选择端，以选择内存读出数据的指定字节，如图 4.5 正确的内存读取逻辑所示。

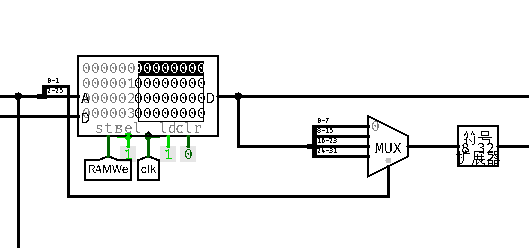
****

图 4.5 正确的内存读取逻辑

### 保存断点故障

Logisim 单周期 CPU：EPC 保存断点错误。

**故障现象：**如图 4.6 错误的断点保存电路所示，EPC总是会将新的 PC（即中断处理例程的起始地址 0x800） 保存进来，而不是保存断点地址。

**原因分析：**由于 PC 寄存器与 EPC 寄存器的时钟源都为 clk，故当 EPC进行锁存时，PC往往为新的锁存值，从而导致 EPC 保存了中断例程首址，而不是断点地址。

**解决方案：**如图 4.7 正确的断点保存电路所示，加入一个 PC 缓冲寄存器，以 HasInt（中断请求信号）为时钟源，便可立即将旧的 PC 锁存，并在下一个时钟来临时，将其锁存至 EPC 寄存器**。**

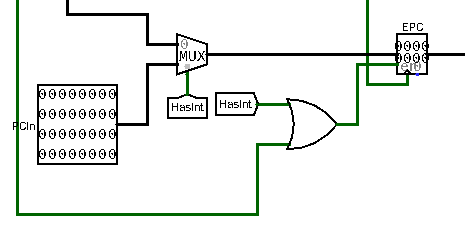
****

图 4.6 错误的断点保存电路

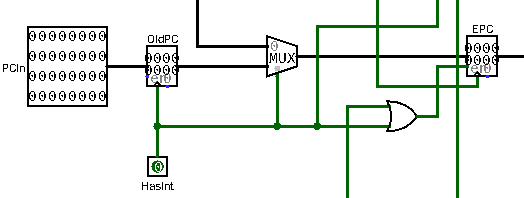
****

图 4.7 正确的断点保存电路

### 中断服务例程缺陷

Logisim 单周期 CPU：中断返回存在风险。

**故障现象：**尽管未观察到故障，但由于原有的中断程序是利用 jr 指令进行中断返回的，需要 2 条指令才可完成（取返回地址+跳转），而取地址为开中断前，跳转为开中断后，存在被打断的风险。

**解决方案：**在 CPU 中加入对 eret 指令的支持（EPC 送 PC），并修改中断程序，利用 eret 进行中断返回，使其变为原子操作，去除被打断的风险。

### 中断请求响应电路设计缺陷

Logisim 单周期 CPU：中断请求响应电路存在设计缺陷。

**故障现象：**通过学习后发现，原有的响应电路（如图 4.8 简易中断请求响应电路所示）并未将中断请求进行锁存这一设计，是不符合实际要求的。若时钟足够慢，则输入的中断源必须保持足够长的时间，才可以成功地引起中断，从而使 CPU 转入中断处理程序，这一设计显然是不合理的。

**原因分析：**由于 3 个中断请求寄存器的时钟源为clk 使得其不能及时锁存中断源，故应修改中断请求寄存器的时钟源。

**解决方案：**如图 4.9 同步清0锁存中断源的中断请求响应电路所示，将中断请求寄存器的时钟源改为中断源，即可实现中断请求的立即锁存，再利用 clk 进行组合逻辑，得到 HasInt信号。这样一来，既可以不遗漏任何中断源，又可以实现中断请求信号的产生同步于时钟。

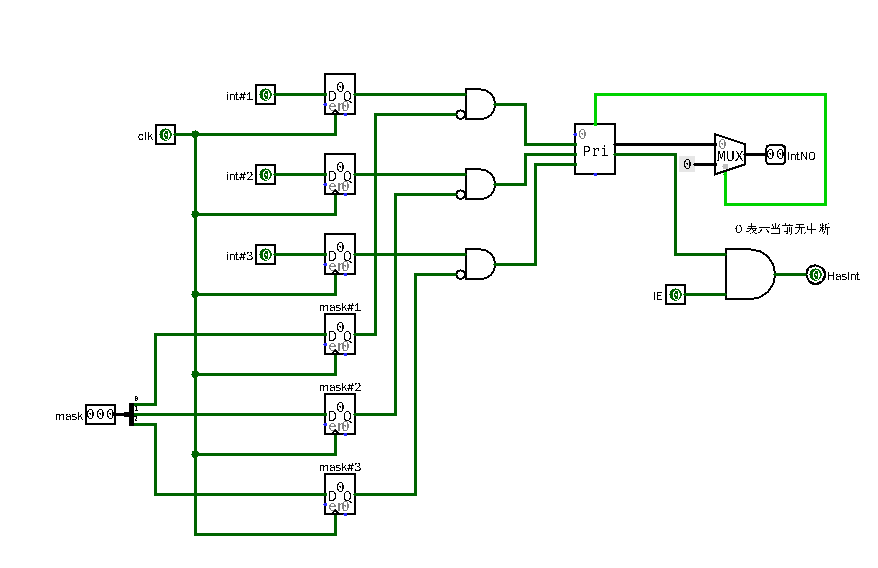
****

图 4.8 简易中断请求响应电路

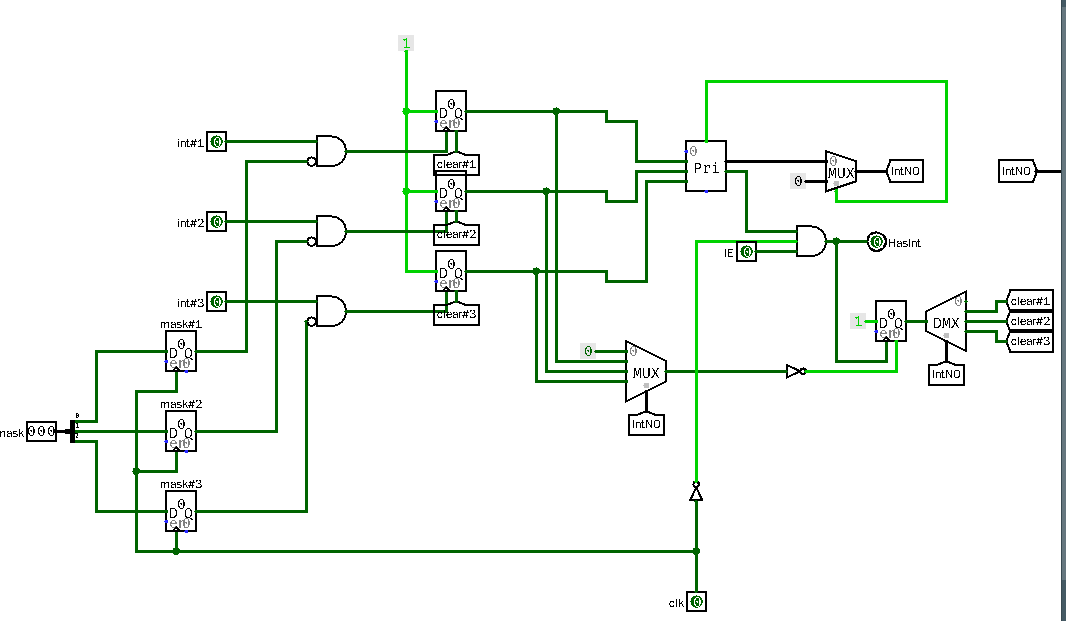


图 4.9 同步清0锁存中断源的中断请求响应电路

### XXX 故障

Logisim 单周期 CPU：EPC 保存断点错误。

**故障现象：**如图 4.6 错误的断点保存电路所示，EPC总是会将新的 PC（即中断处理例程的起始地址 0x800） 保存进来，而不是保存断点地址。

**原因分析：**由于 PC 寄存器与 EPC 寄存器的时钟源都为 clk，故当 EPC进行锁存时，PC往往为新的锁存值，从而导致 EPC 保存了中断例程首址，而不是断点地址。

**解决方案：**如图 4.7 正确的断点保存电路所示，加入一个 PC 缓冲寄存器，以 HasInt（中断请求信号）为时钟源，便可立即将旧的 PC 锁存，并在下一个时钟来临时，将其锁存至 EPC 寄存器**。**

### XXX 故障

Logisim 单周期 CPU：EPC 保存断点错误。

**故障现象：**如图 4.6 错误的断点保存电路所示，EPC总是会将新的 PC（即中断处理例程的起始地址 0x800） 保存进来，而不是保存断点地址。

**原因分析：**由于 PC 寄存器与 EPC 寄存器的时钟源都为 clk，故当 EPC进行锁存时，PC往往为新的锁存值，从而导致 EPC 保存了中断例程首址，而不是断点地址。

**解决方案：**如图 4.7 正确的断点保存电路所示，加入一个 PC 缓冲寄存器，以 HasInt（中断请求信号）为时钟源，便可立即将旧的 PC 锁存，并在下一个时钟来临时，将其锁存至 EPC 寄存器**。**

## 实验进度

表 4.1 课程设计进度表

| 时间 | 进度 |
| --- | --- |
| 第一天 | 复习组成原理CPU相关理论知识，阅读课设任务书，阅读MIPS指令手册，并列出CPU各部件的数据通路表，并完成数据通路的基本构建。 |
| 第二天 | 完成单周期CPU的控制信号表，使用Logisim搭建控制器，实现了单周期CPU并且通过了测试。完成部分Logism单周期CPU故障报告。 |
| 第三天 | 完成Logism单周期CPU的故障报告，并且通过了Logism单周期CPU的检查。使用Verilog实现了部分单周期CPU的重要部件，并通过仿真检查。 |
| 第四天 | 继续使用Verilog进行实现单周期CPU的工作，完成了所有部件的编写、控制器的编写，以及所有部件以及控制器的仿真测试，正在进行数据通路的拼接。 |
| 第五天 | 使用Verilog完成单周期CPU数据通路的连接，并且通过仿真测试。使用Verilog完成时钟分频以及七段数码管的代码编写，正在调试。 |
| 第六天 | 完成CPU电路的功能仿真和时序仿真，并成功将生成bit流烧入FPGA板内实现预计功能。 |
| 第七天 | 复习关于指令流水线的知识点，完成理想流水线的verilog代码，正在调试。 |
| 第八天 | 调试成功理想流水线verilog代码，并成功将bit流烧至FPGA板中。完成冒险处理中的数据冲突处理和分支处理代码编写，正在调试。 |
| 第九天 | 完成冒险处理中的数据冲突和分支处理，并成功烧入FPGA板内。完成数据重定向的Verilog代码的编写，正在调试。 |
| 第十天 | 完成数据重定向的Verilog代码并成功烧入FPGA板内。成功实现动态分支预测，预测成功率显著提高，并成功将代码烧入FPGA板内。 |

# 设计总结与心得

## 课设总结

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

1. 完成方案总结（自行修订扩充）（自行修订扩充）（自行修订扩充）（自行修订扩充）（自行修订扩充）。
2. 功能总结（自行修订扩充）（自行修订扩充）（自行修订扩充）（自行修订扩充）（自行修订扩充）。
3. 其他需要总结的内容，（自行修订扩充）。

## 课设心得

本次课程设计可以说是迄今为止所有实验以及课程设计中难度最大的一门。两个星期从早到晚的不懈努力以及国庆节假期的辛苦加班才终于完成了整个课程设计的设计任务。现在再来回顾整个课程设计的整个过程，满满的成就感自是不用说，但是其中也有不少的细节值得我去深思与体会。

课程设计刚刚开始的时候，第一个任务是使用Logism设计单周期CPU，该任务和上学期实验课的任务类似，所以整个过程还算比较迅速。然而，在将该电路使用Verilog语言进行描述时，麻烦接踵而至。因为Logism对于电路是完全可视化的，连接过程清晰明了，但是使用Verilog语言进行数据通路时，因为各个部件的接口在定义时起名不是很规范，又因为只能使用各个wire型变量对于关键部件进行连接，而这些变量的名字在定义时同样没有做到一目了然，从而导致了我在完成了数据通路的连接后，使用了大量的时间对于整个电路进行排错和检查，最终还是将所有的接口和连线规范化的起了名字，才终于解决了问题。这也就说明了工程化、规范化对于电路设计或是程序设计的重要性。

紧接着，理想流水线CPU的设计并没有什么难度，但是使用插入气泡、数据重定向技术对于流水线CPU进行冒险处理时，因为这些方法书本上并没有，老师提供的PPT上也只有简单的一些描述，这就要求我不断地在网上搜索相关的知识内容，和小组内的成员进行相关探讨。随后的动态分支预测的设计难度更高，老师并没有提供任何相关的内容，这又使得我不断地去网上搜寻资料文献，阅读全英文的学术论文。从这一切也可以看出团队在一个工作中的重要性，以及自我学习能力的必要性。

然而对于本次课程设计，我还有一些小小的建议和改进。本次课程设计按照分组的机制，但是因为所有人所需要完成的工作相同，分组也只是更便于组内的同学在设计过程中进行互相交流，分组的作用并不是非常明显。分组内的同学们进度完全不同，且有些同学因为对于课程内容的掌握不是很好，逐渐边缘化，分组并不能调动所有同学加入到课程设计任务中。本课程设计的难度很大，但是时间又有一点不够充分，可以适当简化一下任务（并不是降低最终目标），使得在有限的时间内，在完成全部课程设计目标的基础上，适当减少任务中其中重复的繁杂的工作，以提高整个课程设计的效率。

最后在这里也感谢三位老师对于我在本次课程设计中无数问题的耐心解答，也感谢本组所有成员在课程设计中对于我的帮助和建议。我相信组成原理课程设计必将成为我整个大学生涯中一段无比难忘的回忆。

# 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:** |