# 目 录

# 1 课程设计概述

## 1.1 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。

## 1.2 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。

## 1.3 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。

## 1.4 技术指标

（8） 支持表 1.1 前 27 条基本 32 位 MIPS 指令；

（9） 支持教师指定的 4 条扩展指令；

（10）支持多级嵌套中断，利用中断触发扩展指令集测试程序；

（11）支持 5 段流水机制，可处理数据冒险，结构冒险，分支冒险；

（12）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。

（13）能运行教师提供的标准测试程序，并自动统计执行周期数

（14）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 | |

| # | 指令助记符 | 简单功能描述 | 备注 |
|---|---|---|---|
| 14 | LW | 加载字 | |
| 15 | SW | 存字 | |
| 16 | BEQ | 相等跳转 | |
| 17 | BNE | 不相等跳转 | |
| 18 | SLT | 小于置数 | |
| 19 | SLTI | 小于立即数置数 | |
| 20 | SLTU | 小于无符号数置数 | |
| 21 | J | 无条件转移 | |
| 22 | JAL | 转移并链接 | |
| 23 | JR | 转移到指定寄存器 | If $v0==10 halt(停机指令) |
| 24 | SYSCALL | 系统调用 | else 数码管显示$a0 值 |
| 25 | MFC0 | 访问 CP0 | 中断相关，可简化，选做 |
| 26 | MTC0 | 访问 CP0 | 中断相关，可简化，选做 |
| 27 | ERET | 中断返回 | 异常返回，选做 |
| 28 | SLLV | 可变左移 | R[$rd]←R[$rt]<<R[$rs] |
| 29 | SUBU | 无符号减 | |
| 30 | LB | 加载字节 | |
| 31 | BLTZ | 有符号小于 0 跳转 | RS<0 PC+=Ext({imm, 00}) |

# 2 总体方案设计

## 2.1 单周期 CPU 设计

本次我们采用的方案是硬布线控制，且采用哈佛结构（指令存储器与数据存储器分离的方式），硬布线控制器由运算器控制器和控制信号生成两个部件组成，通过输入（指令码的 op 及 funct 字段）产生需要的控制信号。同时在实施的过程中，首先在 logisim 上完成整体电路的设计，然后利用 verilog 设计各种部件，完成数据通路，最后上板。

总体结构图如图 2.1 所示。

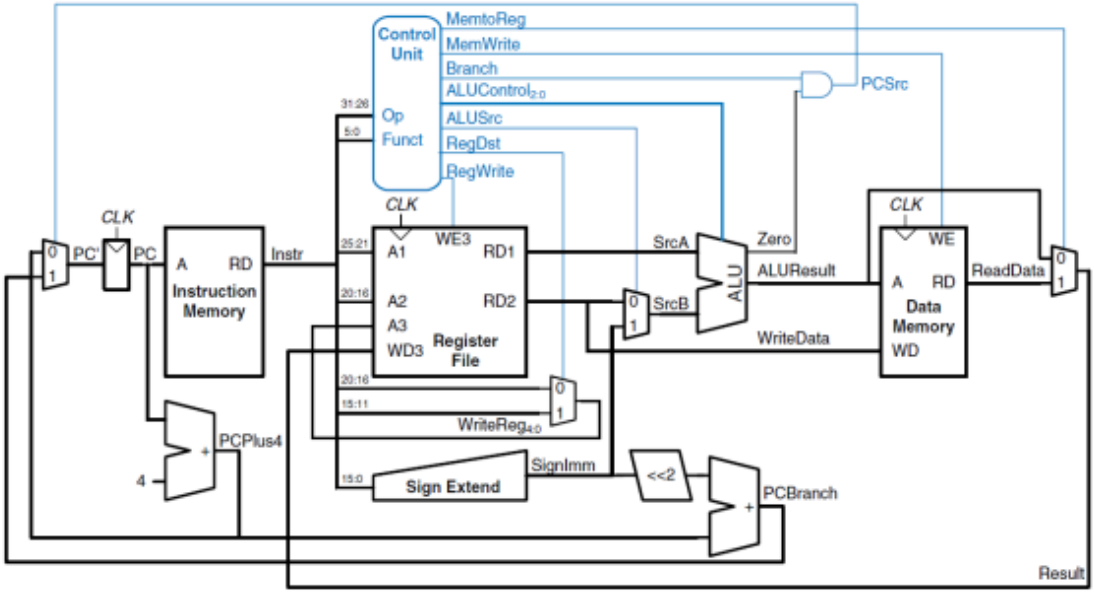


图 2.1 总体结构图

### 2.1.1 主要功能部件

1. 程序计数器 PC

程序计数器 PC 的本质上就是一个寄存器，在时钟上升沿到来时，如果使能端为高电平的话，将输入的数据锁存并且输出，如果清零信号为高电平的话，将寄存器清零。

2. 指令存储器 IM

本次实验的指令存储器实质上就是一个 ROM。为了易用性方面的考虑，采用直接读文件的方式存储指令，文件路径名作为参数。采用数组的方式存储，每个数组元素为 32 位，同时数组的大小也可以根据参数来直接设定。输出就根据输入地址选择输出。

3. 运算器

在 logisim 中运算器是实验包中已经给出的，所以值需要在 verilog 中实现就行。运算器需要实现 13 个功能，根据不同的操作码，可以实现的功能有左移，算数右移，逻辑右移，乘、除、加、减、与、或、异或、或非、有符号比较、无符号比较。其中算数右移需要考虑符号位的问题，在 verilog 中>>操作符默认逻辑右移。同时有符号比较也要注意一下，只要将输入数据的正负四种情况考虑进去就可以实现比较功能了，如正数和负数的比较等。

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 |

4. 寄存器堆 RF

寄存器在 logisim 中已经给出实现了，只需在 verilog 中实现即可。为了后面流水线的便利，寄存器堆设置为时钟下降沿有效。寄存器读只需要将输入的 R1 和 R2 寄

存器编号作为下标读取寄存器堆的数据即可。当下降沿到来的时候，如果写使能信号为高电平，那么就将输入的数据写入写寄存器，写寄存器的编号也是从输入中获得的。

5. 数据存储器 DM

本次实验的数据存储器实质上就是一个RAM。RAM和ROM的设计是很类似的。当时钟上升沿来时，如果存储使能 str 为高电平，那么就将输入的数据存入 DM 相应的位置，地址信息从输入中获得；如果清零信号为高电平，那么将所有的数据置 0。最后 DM 的输出就是 DM 中相应地址所存放的数据。

### 2.1.2 数据通路的设计

数据通路的设计是在上学期 CPU 单周期的数据通路基础上进行修改而来的。由于 R 型指令的数据通路基本上都是一样的，所以只需要针对特殊指令设计特殊的数据通路即可。一般 R 型数据通路如图 2.2，需要添加额外数据通路的两条特殊指令就是 JR 和 SLLV 指令。对于 JR 来说，它需要将 PC 的值更新为 R1 寄存器的值，因此在 PC 更新时，要额外加一个多路选择器，当 JR 信号为高电平时，选择 R1 寄存器的值作为输出。对于 SLLV 指令来说，主要增加的额外数据通路就是 ALU 的 shamt 入口处，与一般的 R 型移位指令（需要移位的数据在指令码 shamt 字段）不同，SLLV 要将 R1 寄存器的低 5 位作为移位数送入 ALU 的 shamt 输入。因此就要在 ALU 的 shamt 入口处加一个多路选择器，当 SLLV 信号为高电平时，选择对应的 shamt 的值。

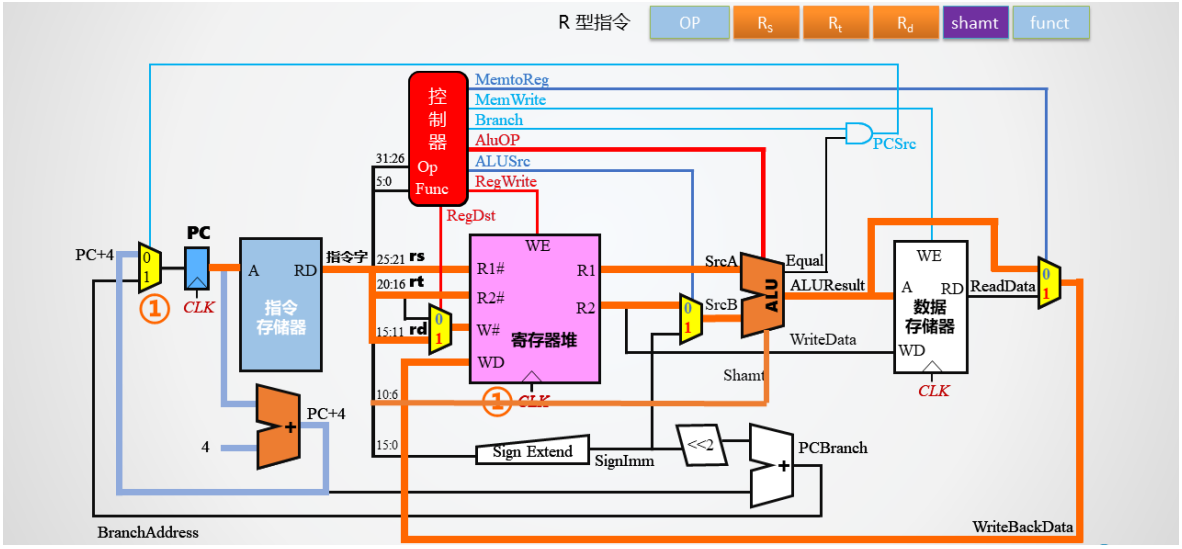


图 2.2 R 型指令通路

对于 syscall 指令而言，跟上学期实现的有所不同，因为在课程设计中，加入了打印中断不阻止程序继续运行，go 的实现。因此 PC 使能端就有一定程度的修改。具体的逻辑为(~syscall)|(syscall&打印中断)|go。其中打印中断的逻辑只需将 2 号寄存器的值与 34 比较即可。因此也需要在 R1#前添加多路选择器。

对于 J 型指令，这里主要指 J 和 JAL。首先它们实现的共同功能就是使 PC 的值跳转。这里在 PC 寄存器之前加一个多路选择器就可以了，当 J 或 JAL 信号为高电平时，选择 PCGPRLEN-1..28 || instr_index || 00 作为输入。除此之外，JAL 还需要做的工作是 GPR[31]← PC＋4。因此需要在寄存器堆的 W#输入前加一个多路选择器，当 JAL 为高电平时，选择 31 作为 W#的输入；同时在 Din 输入前也要加多路选择器，当 JAL 为高电平时，选择 PC＋4 作为 Din 的输入。

对于 I 型指令（ADDI、ANDI、ADDIU、SLTI、ORI），与 R 型指令大致相似（具体参考图 2.2）。不同之处在于寄存器堆 W#的来源以及 ALU B 的来源。W#的来源为 rt，因此 I 型指令的 RegDst 信号为低电平。同时，ALU B 的来源为指令码的立即数字段，AluSrc 信号为高电平。这里一个特例就是 ORI 指令，因为一般立即数是要用符号扩展的，而 ORI 的立即数需要无符号扩展。因此 ORI 的符号扩展信号 SignExt 为 0。

对于 LW，LB，SW 指令而言，只要理解了上面的基本数据通路，它们的数据通路也就很好理解了，如图 2.3，图 2.4。对于 LB 指令，只需在 LW 通路基础上，在 WD 源加入多路选择器，当 LB 信号为高电平时，选择相应的字节数据作为输入。字节的选择由 ALUResult 的低两位完成。

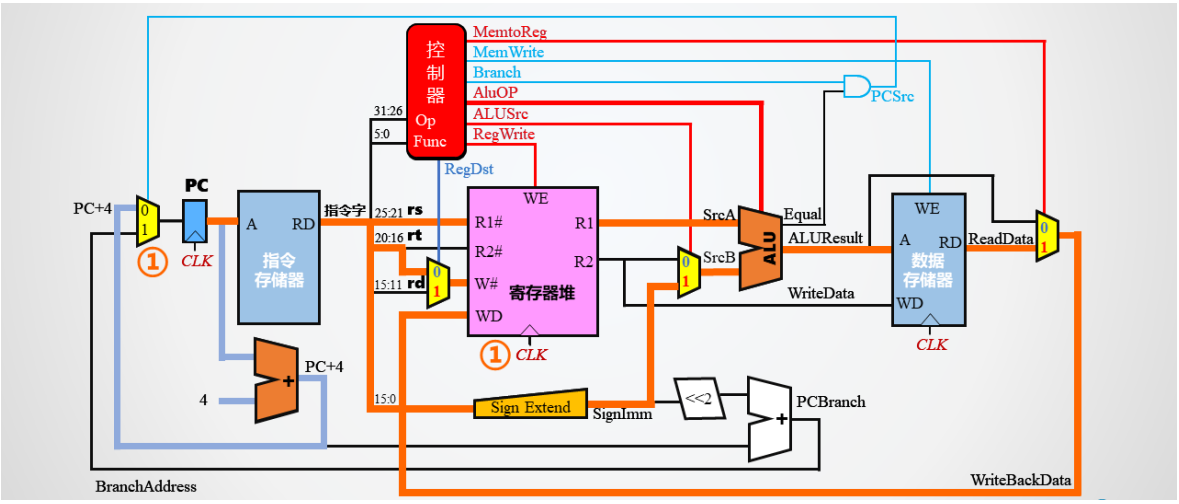


图 2.3 LW 指令数据通路

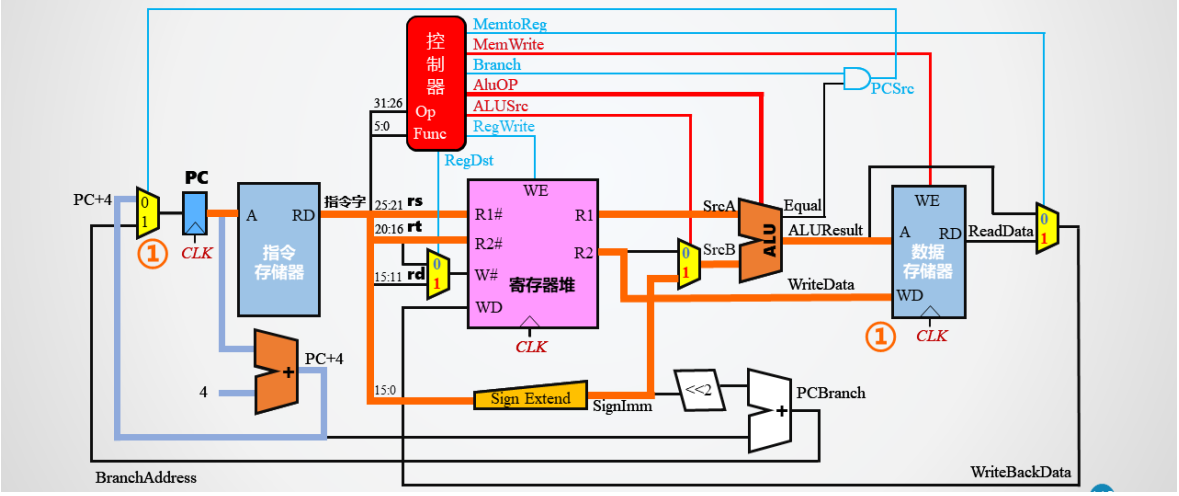


图 2.4 SW 指令数据通路

对于 BEQ、BNE、BLTZ 指令，数据通路的设计也是水到渠成的。如图 2.5 BEQ 指令数据通路，BNE 只是在 ALU 的 Equal 信号为低电平时使得 PC 跳转，其余没有什么变化。BLTZ 也是这样，如果 R1 的值小于 0，那么要令 PC 跳转。这里的比较是利用 ALU 完成的，因此 ALU B 应为 0，所以就应该在其入口处添加多路选择器，当 BLTZ 信号为高电平时，选择 0 作为输入。

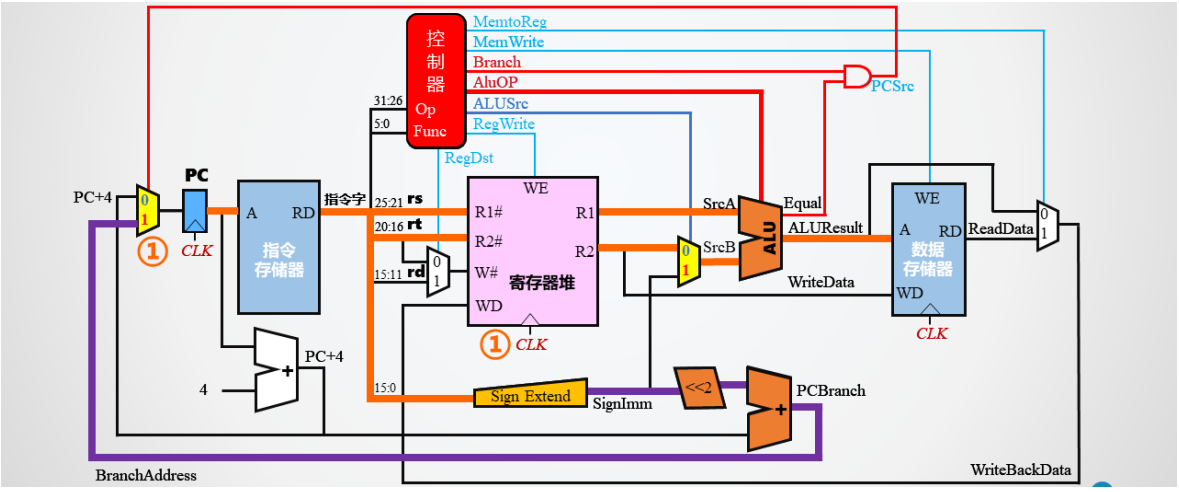


图 2.5 BEQ 指令数据通路

显示数据就只需要当打印逻辑来临，将 4 号寄存器的值送入一个寄存器，并且输出到数码管。因此很容易就知道需要在 R2#前添加多路选择器。

具体的指令系统数据通路框架在详细设计与实现中给出，因此在这里就不再给出图表的展示了。

### 2.1.3 控制器的设计

首先对于控制信号进行统计，包括各个主要部件所需要输入的控制信号，以及数据通路合并表中所示的具有多输入的主要部件需要进行输入选择的控制信号，并且对各个统计信号的各种取值情况进行定义，统计得到的控制信号以及说明如表 2.2。

表 2.2 主控制器控制信号的作用说明

| 控制信号 | 取值 | 说明 |
|---|---|---|
| ALU_OP | 0-15 | ALU 进行不同的运算，如移位，比较，加减乘除等 |
| MemToReg | 0 | 寄存器堆写入数据来自 ALU |
| | 1 | 寄存器堆写入数据来自 DM |
| MemWrite | 0 | DM 写使能无效 |
| | 1 | DM 写使能有效 |
| ALU_SRC | 0 | ALU B 的输入来自 R2 |
| | 1 | ALU B 的输入来自 imm 字段 |
| RegWrite | 0 | 寄存器堆写使能无效 |
| | 1 | 寄存器堆写使能有效 |
| SysCALL | 0 | 非 syscall 指令 |
| | 1 | Syscall 指令，用于联合判断 PC 写使能是否有效 |
| SignedExt | 0 | 无符号扩展 |
| | 1 | 有符号扩展 |
| RegDst | 0 | 寄存器堆 W#输入来自 rt |
| | 1 | 寄存器堆 W#输入来自 rd |
| Beq | 0 | 非 beq 指令 |
| | 1 | Beq 指令，联合判断是否跳转 |
| Bne | 0 | 非 bne 指令 |
| | 1 | Bne 指令，联合判断是否跳转 |
| JR | 0 | 非 JR 指令 |
| | 1 | JR 指令，跳转 |
| JMP | 0 | 非 J 指令 |

| 控制信号 | 取值 | 说明 |
|---|---|---|
| JMP | 1 | J 指令，跳转 |
| JAL | 0 | 非 JAL 指令 |
| | 1 | JAL 指令，跳转，寄存器堆 W#的输入为 31，Din 的输入为 PC+4 |
| SLLV | 0 | 非 SLLV 指令 |
| | 1 | SLLV 指令，ALU shamt 输入来源于 R1 的低 5 位 |
| LB | 0 | 非 LB 指令 |
| | 1 | LB 指令，寄存器堆 Din 的输入为需要的字节数据 |
| BLTZ | 0 | 非 BLTZ 指令 |
| | 1 | BLTZ 指令，ALU B 的输入为 0，联合判断是否跳转 |
| MFC0 | 0 | 非 MFC0 指令 |
| | 1 | MFC0 指令，中断相关 |
| MTC0 | 0 | 非 MTC0 指令 |
| | 1 | MTC0 指令，中断相关 |
| ERET | 0 | 非 ERET 指令 |
| | 1 | ERET 指令，中断相关，返回 |

对照所有控制信号，依次分析各条指令，分析该指令执行过程中需要哪些控制信号，对于与本条指令无关的控制信号，控制信号的取值一律为 0，以简化控制器电路的设计。该控制信号表的框架如表 2.3 所示。

表 2.3 主控制器控制信号框架

| 指令 | ALU_OP | MemtoReg | MemWrite | ALU_SRC | RegWrite | SYSCALL | SignedExt | RegDst | BEQ | BNE | JR | JMP | JAL | SLLV | LB | BLTZ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SLL | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SRA | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SRL | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADD | 5 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADDU | 5 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

| 指令 | ALU_OP | MemtoReg | MemWrite | ALU_SRC | RegWrite | SYSCALL | SignedExt | RegDst | BEQ | BNE | JR | JMP | JAL | SLLV | LB | BLTZ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SUB | 6 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| AND | 7 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| OR | 8 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| NOR | 10 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SLT | 11 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SLTU | 12 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| JR | X | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| SYSCALL | X | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| J | X | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| JAL | X | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| BEQ | X | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| BNE | X | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADDI | 5 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ANDI | 7 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ADDIU | 5 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SLTI | 11 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ORI | 8 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| LW | 5 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SW | 5 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| SLLV | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| SUBU | 6 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| LB | 5 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| BLTZ | 11 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

## 2.2 中断机制设计

### 2.2.1 总体设计

如图 2.6 多重中断流程图，可以看到中断需要完成保存断点、中断识别、保护现场与设置新的屏蔽字，设备中断服务子程序、恢复现场、中断返回几个功能。完成这几个功能，本次课程设计利用 MFC0,MTC0,ERET 三条指令完成。

本次课程实验由于不需要很复杂的功能，因此没有系统实现 CP0 协处理器，只是用了一个封装好的电路实现中断功能。保存断点就是将旧 PC 入栈，当中断要返回时，读取栈中的 PC 值，将其送入 PC 寄存器中。中断识别就是根据案件产生的信号，识别中断源和中断号、切换到中断处理程序地址执行中断子程序。保护现场的工作可以用 MFC0,MTC0,ERET,LW,SW 等指令完成，这样可以实现多重中断。恢复现场同理，当中断返回时，取出旧 PC，送入 PC 寄存器即可。

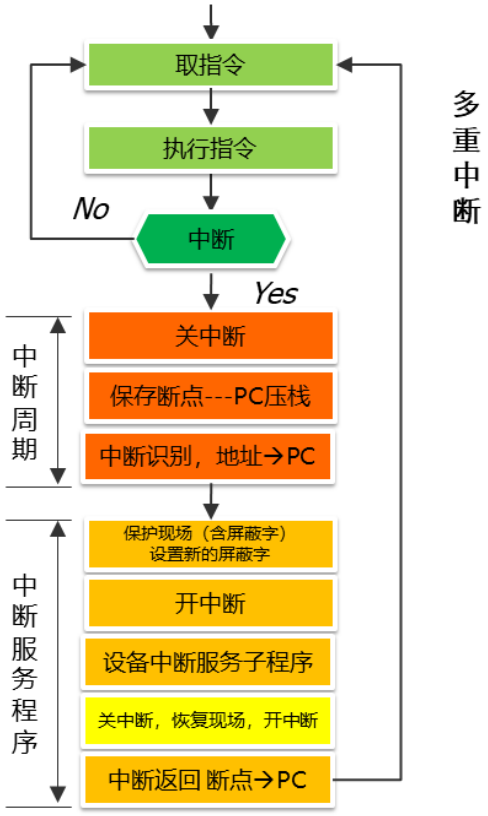


图 2.6 多重中断流程图

### 2.2.2 硬件设计

中断识别原理图如图 2.7。

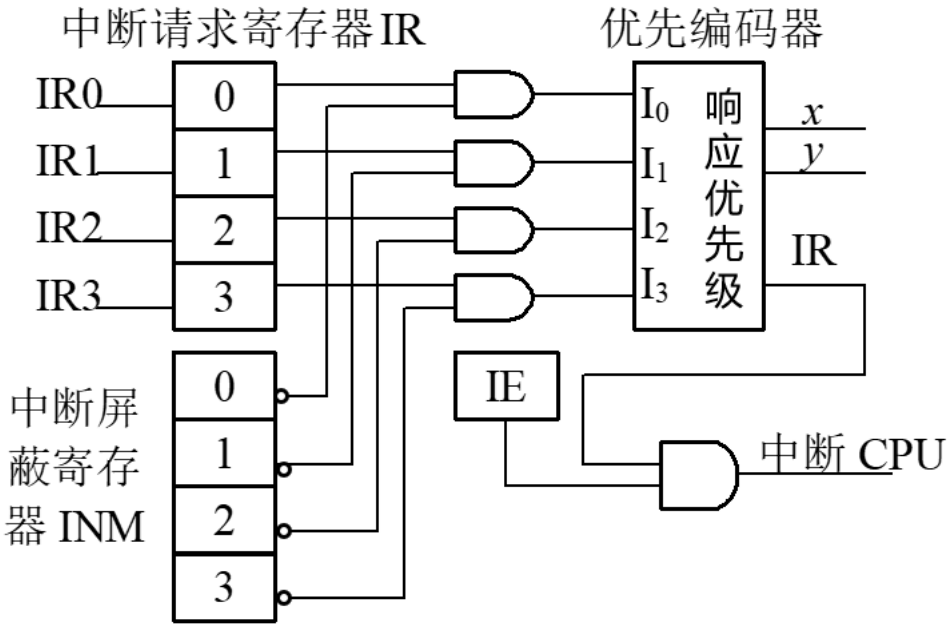


图 2.7 中断识别原理图

有中断来临时，结合中断屏蔽字，来判断是否进行中断。优先编码器可以天然地识别优先级最高的中断，其输出与中断使能信号进行进行与操作，可以得到中断 CPU 信号。当此中断得到相应后，将中断信号清零，此部分可以用两个 D 触发器以及逻辑门实现。中断使能信号 IE 由一个寄存器保存，当 mtc0 指令执行时，寄存器可写入，输出最低位取反作为 IE。在现在程序中发生了那些中断可以用寄存器保存这些信号，当有中断等待或者当前中断正在执行时，这些信号保持高电平，然后通过优先编码器可以得到优先级最高的中断号，如果有优先级更高的中断来临，则中断号寄存器使能端有效，将中断号保存。

优先级高的中断（3 号）清零信号逻辑为：当前中断为 3 号中断，并且当前正在中断状态，并且没有 eret 指令。这样可以使得当中断被执行后，将中断清号清零。其余优先级的中断清零信号逻辑也是如此，但是当系统中有其他优先级更高的中断时，它不应该被清零。

当有中断来时，我的设计是将 PC 跳转到相应中断程序的入口地址。利用优先编码器可以很快获得当前中断号，当能发生中断时，利用多路选择器，选择信号为中断号，将入口地址送入 PC。

### 2.2.3 软件设计

要完成多级中断，需要在程序中运用 mfc0，mtc0，eret 指令。首先利用 mfc0 指令读出 epc 的值，然后利用 sw 指令保存。之后关中断，利用 sw 指令将中断程序要用到的寄存器入栈，然后进入中断服务子程序，当执行完后，关中断，利用 lw 指令将栈内的数据出栈到相应寄存器，然后开中断。最后恢复 PC 的值，返回。

## 2.3 流水 CPU 设计

### 2.3.1 总体设计

通过 mooc 的学习，掌握了流水线的设计。在原来单周期的基础上，加入流水接口部件，将信号依次传输，在同一时钟周期内，有多条指令在不同的段执行，这就是流水线的基本思想。五段流水线分为 IF、ID、EX、MEM、WB 这五段。

理想流水线不同考虑一些复杂的问题，每个指令要经过五段，并且没有数据相关。

分支相关流水线要解决分支相关问题，主要就是当分支指令执行到 EX 段时，要将前两段信号清零，因为这是两个误取指令。

气泡流水线就要进一步解决数据相关问题了，主要就是要用到检测机制，在下面会给出详细设计。

重定向流水线相较于气泡流水线的区别就是，当出现数据相关问题时，利用旁路将需要的数据提前送达，可有效减少气泡数。但这样也不能解决 LoadUse 相关，因此要加入 LoadUse 相关检测。

具体的流水线数据通路图在详细实现里贴出。

### 2.3.2 流水接口部件设计

4 个流水接口部件就是寄存器的集合，用来保存接来下要用到的信号。首先设计复杂的 ID/EX，然后其他部件可以在它的基础上修改，加快了设计的速度。

Verilog 实现也是十分简单，只需要利用寄存器模块即可。

### 2.3.3 理想流水线设计

按照总体设计里所说，理想流水线不需要考虑很多，只需加入 4 个流水接口部件，修改一下单周期数据通路即可。

## 2.4 气泡式流水线设计

气泡流水线就要在分支相关流水线上进一步解决数据相关问题了，气泡流水线采用插入气泡的方式来解决这一问题，在 ID 段检测到数据相关后，将 ID/EX 清零（插入气泡），并且暂停 IF/ID 与 PC 的写入。数据相关的检测也需要用硬件电路来实现，首先要解决的就是当前指令源寄存器使用情况，这一点在 mooc 上有详细讲解，因此在这里就不赘述，要比较的就是 ID 段的 R1，R2 寄存器编号是否与 EX 或 MEM 段的 W#编号相同，要注意的一点是零号寄存器恒 0，不当做数据相关。

具体的流水线数据通路图在详细实现里贴出。

## 2.5 数据转发流水线设计

重定向流水线就是当指令在 ID 段时，检测指令要用到 EX、MEM 哪一段的数据，这样就直接利用旁路输送数据，不用插入气泡。但这样仍然不能解决 LoadUse 相关，当检测到 LoadUse 后，就要清空 ID/EX 段，并且暂停 IF/ID 段，等待写回。

具体的流水线数据通路图在详细实现里贴出。

在完成 logisim 的数据通路图后，就开始实现 FPGA 重定向流水线了。只需要增加流水接口的部件，并且按照新的数据通路编写即可。只要单周期上板很好地理解了设计方法，重定向流水线的 FPGA 实现就是水到渠成的。

## 2.6 动态分支预测机制

动态分支预测是指在 IF 段就根据跳转指令的历史跳转信息作出预测，决定下一条 PC 的值。动态分支预测在重定向流水线的基础上实现，可以有效减少周期数（因为程序里循环、无条件跳转指令等存在）。动态分支预测策略比较简单的就是 BHT（Branch History Tabel），BHT 将记录下分支跳转指令的跳转地址，以及决定是否跳转。仿照上学期 cache 实验，设计 BHT，包括 valid 位，PC 值，跳转地址，双预测位以及淘汰标志位。

值得注意的就是预测策略，在这里我根据文献，采用了简单的 2-bit 状态机。它是这样工作的：规定一个初始值，每当一个分支跳转预测正确了，就加 1,；每当一个分支跳转预测不正确，就减 1。等到状态 00 的时候，分支预测错误维持现状 00，状态为 11 的时候，分支预测正确维持现状 11。然后根据状态的高位判断是否跳转，如 11、10 表示跳转，00、01 表示不跳转。原理就是，当预测出现错误时，先不急着

修改预测方向，而是再给一次机会，如果连续出错两次，则改变预测。相当于给了一次试错的机会。根据文献内容，仅仅是这样简单的动态预测策略，跑过了一些标准的 benchmark 后，也能有至少 80%的正确率，甚至最高能达到 99%。

对于 LRU 淘汰算法，主要思想就是当某一行命中时，它的计数清零，其余行的计数加 1。当所有行都不命中时，所有计数加 1。最后淘汰计数最高的一项，完成 LRU 淘汰。

其余的就和上学期 cache 实验相同，在本次报告中就不以这个为重点。

# 3 详细设计与实现

## 3.1 单周期 CPU 实现

### 3.1.1 主要功能部件实现

1) 程序计数器（PC）

① 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。输入的数据时经过三个二路选择器选择的数据，使能端的逻辑在总体设计中已经实现，在此不再赘述。如图 3.1 程序计数器（PC）。

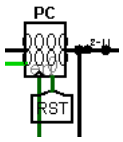


图 3.1 程序计数器（PC）

② FPGA 实现：

程序计数器 PC 的 Verilog 代码如下：

```
module PCreg(
    input clk,
    input rst,
    input ena,
    input [31:0]data_in,
    output [31:0]data_out
    );

    reg [31:0]data = 32'b0;

    always @(posedge clk or posedge rst) begin
        if(rst) data<=32'h00000000;          //reset key
```

```
        else begin
            if(ena) data<=data_in;          //enable ,input
        end
    end

    assign data_out = data;

endmodule
```

2) 指令存储器（IM）

① Logism 实现：

使用一个只读存储器 ROM 实现指令存储器（IM）。设置该只读存储器的地址位宽为 10 位，数据位宽为 32 位。因为 PC 中存储的指令地址有 32 位，而 ROM 地址线宽度有限，仅为 10 位，故将 32 位指令地址高位部分和字节偏移部分直接屏蔽，使用分线器只取 32 位指令地址的 2-11 位作为指令存储器的输入地址。如图 3.2 指令存储器。


图 3.2 指令存储器

② FPGA 实现：

按照总体设计中的方法进行 verilog 设计，其设置如图 3.2 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 的大小应该为 1024，在这里为了提高综合的速度，ROM 的大小参数默认为 512。

指令存储器 IM 的 Verilog 代码如下：

```
module ROM#(
    parameter Nmem = 512,// Number of memory entries,
    parameter ROM_DATA = "ROM_DATA.txt"// file to read data from
    )(
    input wire    [31:0]    addr,
    output wire [31:0]      data
```

```
    );

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

    initial begin
        $readmemh(ROM_DATA, memory, 0, Nmem-1);
    end

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

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。

3) 运算器（ALU）

① Logism 实现：

使用 logisim 自带的器件完成各种运算操作。然后根据操作码利用多路选择器选择对应的结果即可。如图 3.3。

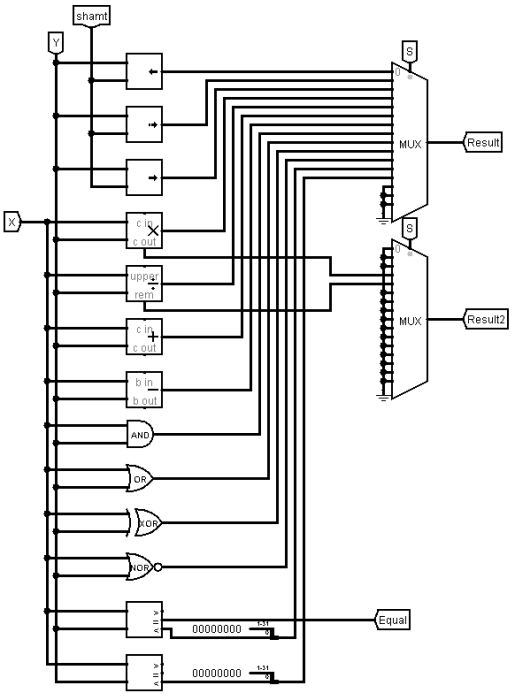


图 3.3 ALU

② FPGA 实现：

按照总体设计中的方法进行 verilog 设计。X、Y 作为两个操作数输入，S 作为操作码，shamt 为移位字段，输出为 Result、Result2、Equal 信号。

运算器（ALU）的 Verilog 代码如下：

```
module ALU(input [31:0] X,input [31:0] Y,input [3:0] S,input [4:0] shamt,
            output reg[31:0] Result,output reg[31:0] Result2,output reg Equal
    );
            reg compare_flag;
             initial begin
                  Result = 0;
                  Result2 = 0;
                  Equal = 0;
             end
             always @(*) begin
                 if(X==Y) Equal = 1;
                 else Equal = 0;
                 case(S)
                    4'b0000:begin
                          Result = Y<<shamt;
                          Result2 = 0;
…
```

4) 寄存器堆 RF

① Logism 实现：

直接使用 CS3410 组件中 Register File。如图 3.4。

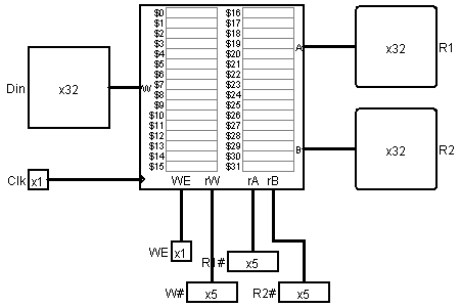


图 3.4 Regfile

② FPGA 实现：

按照总体设计中的方法进行 verilog 设计，其设置如图 3.4 所示。R1#、R2#、W#均为 5 位输入，输入数据 Din 为 32 位。R1 和 R2 是相应编号寄存器的值。

寄存器堆 RF 的 Verilog 代码（部分）如下：

```
always @(negedge CLK or negedge clr)
begin
    if(clr)
        for(i=0;i<NUM;i=i+1) REG_FILE[i]<=0;
    else
        if(WE) REG_FILE[W_ADDR]<=Din;
end

assign R1=REG_FILE[R1_ADDR];
assign R2=REG_FILE[R2_ADDR];
```

5) 数据存储器（DM）

① Logism 实现：

使用一个随机存储器 RAM 实现数据存储器（DM）。该数据存储器地址的设置与 IM 相同，在此不赘述。如图 3.5。

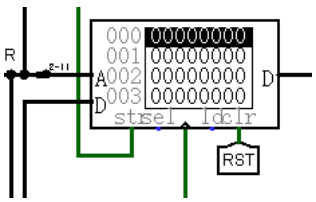


图 3.5 数据存储器 RAM

② FPGA 实现：

按照总体设计中的方法进行 verilog 设计，其设置如图 3.5 所示。同样在这里为了提高综合的速度，ROM 的大小默认为 50。

数据存储器 DM 的 Verilog 代码（部分）如下：

```
module RAM(… );
…
    reg [31:0] state [0:50];
```

```
    always@(posedge clk ) begin
        // reset
        if(rst) begin
            for (j=0; j < 50; j=j+1) begin
                state[j] <= 32'b0; //reset array
            end
        end
        // Store data
        if(str) begin
            if(sel[3]!=0) state[addr][31:24]<=data_in[31:24];
            if(sel[2]!=0) state[addr][23:16]<=data_in[23:16];
            if(sel[1]!=0) state[addr][15:8]<=data_in[15:8];
            if(sel[0]!=0) state[addr][7:0]<=data_in[7:0];
        end
    end
    // Load data
    assign data_out=state[addr];
endmodule
```

### 3.1.2 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。首先在 logisim 中实现理想电路图，然后用 verilog 实现。

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 3.1 指令系统数据通路表

| 指令 | PC | IM | RF | | | | ALU | | | DM | | 备注 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | R1# | R2# | W# | Din | A | B | OP | Addr | Din | |
| ADD | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 5 | | | |

| 指令 | PC | IM | RF | | | | ALU | | | DM | | 备注 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | R1# | R2# | W# | Din | A | B | OP | Addr | Din | |
| ADDI | PC+4 | PC | rs | | rt | alu | r1 | 立即数 | 5 | | | |
| ADDIU | PC+4 | PC | rs | | rt | alu | r1 | 立即数 | 5 | | | |
| ADDU | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 5 | | | |
| AND | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 7 | | | |
| ANDI | PC+4 | PC | rs | | rt | alu | r1 | 立即数 | 7 | | | |
| SLL | PC+4 | PC | | rt | rd | alu | r2 | 立即数 | 0 | | | |
| SRA | PC+4 | PC | | rt | rd | alu | r2 | 立即数 | 1 | | | |
| SRL | PC+4 | PC | | rt | rd | alu | r2 | 立即数 | 2 | | | |
| SUB | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 6 | | | |
| OR | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 8 | | | |
| ORI | PC+4 | PC | rs | | rt | alu | r1 | 立即数 | 8 | | | |
| NOR | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 10 | | | |
| SLT | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 11 | | | |
| SLTU | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 12 | | | |
| JR | r1 | PC | rs | rt | rd | alu | r1 | r2 | | | | |
| SYSCALL | | | | | | | | | | | | |
| J | PC31..28 \|\| instr_index\|\|00 | PC | | | | | | | | | | |
| JAL | PC31..28 \|\| instr_index\|\|00 | PC | | | 31 | PC+4 | | | | | | |
| BEQ | PC+sign_extend (offset \|\| 02) | PC | rs | rt | | | r1 | r2 | | | | |
| BNE | PC+sign_extend (offset \|\| 02) | PC | rs | rt | | | r1 | r2 | | | | |
| SLTI | PC+4 | PC | rs | | rt | alu | r1 | 立即数 | 11 | | | |
| LW | PC+4 | PC | rs | | rt | DM | r1 | 立即数 | 5 | | | |
| SW | PC+4 | PC | rs | rt | | | r1 | 立即数 | 5 | alu | r2 | |

| 指令 | PC | IM | RF | | | | ALU | | | DM | | 备注 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | R1# | R2# | W# | Din | A | B | OP | Addr | Din | |
| SLLV | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 0 | | | Shamt r1 |
| SUBU | PC+4 | PC | rs | rt | rd | alu | r1 | r2 | 6 | | | |
| LB | PC+4 | PC | rs | | rt | DM | r1 | 立即数 | 5 | | | 选择字节 |
| BLTZ | PC+sign_extend (offset \|\| 02) | PC | rs | rt | | | r1 | 0 | 11 | | | |

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。

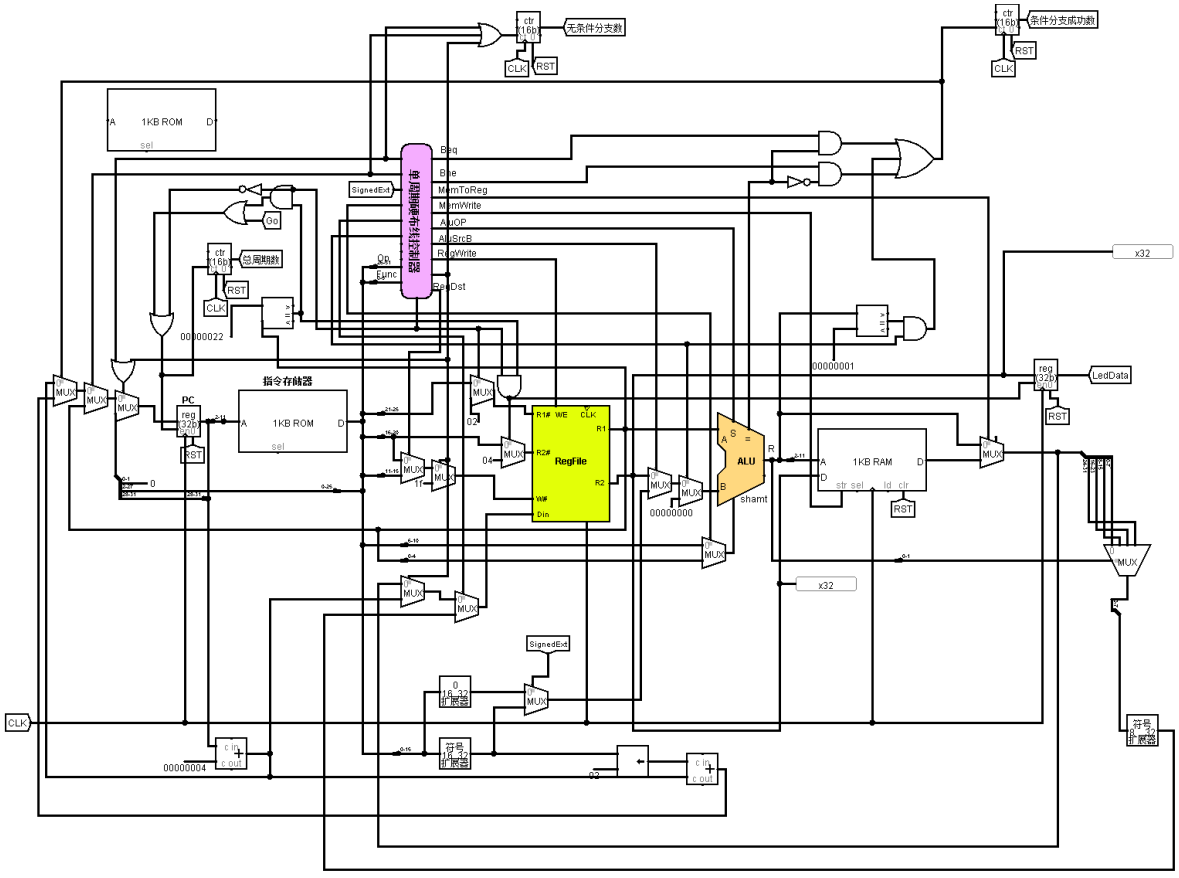


图 3.6 单周期 CPU 数据通路（Logism）

在 Vivado 中使用 Verilog 语言搭建的数据通路的原理图如图 3.7 所示。

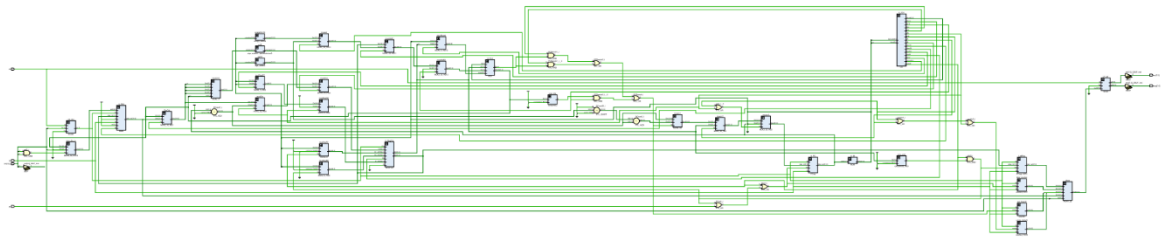

图 3.7 单周期 CPU 数据通路（FPGA）

### 3.1.3 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 上进行控制器的具体实现。表格已经在设计中呈现，受篇幅限制，在此不再赘述，并且图片也只能显示一部分（图片过长）。

① Logisim 实现

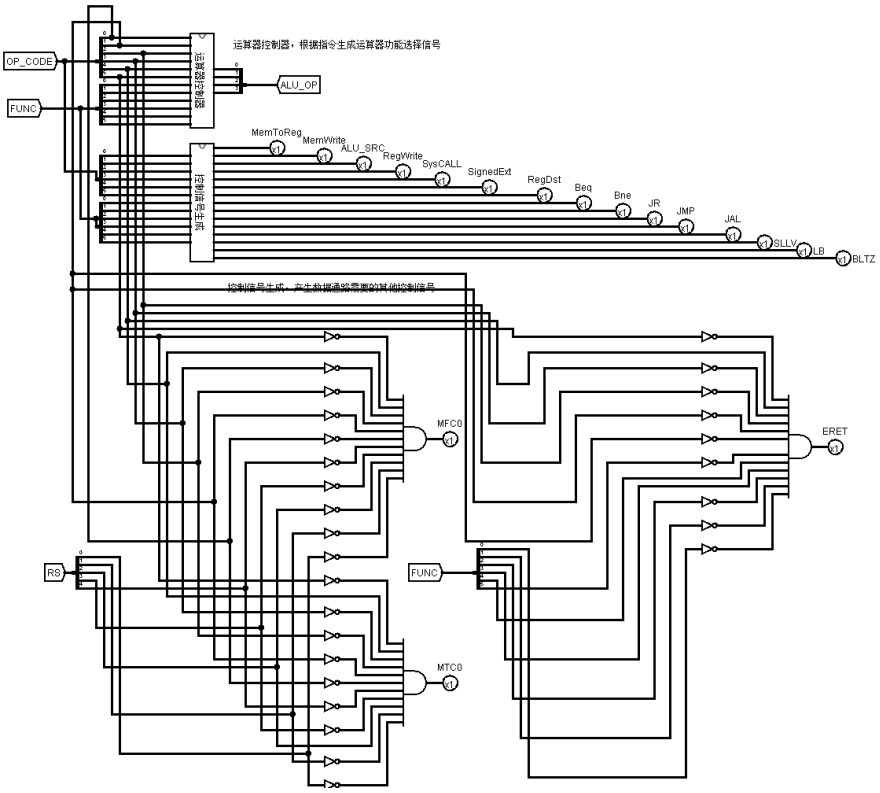


图 3.8 控制器

② FPGA 实现

将各个信号作为输出，对于不同指令，输出不同的信号，由于 Verilog 代码过于冗长，在此只显示 BEQ 指令的信号：

```
`INSTR_BEQ_OP:
```

```
        begin
            Mem2R = 0;
            MemW = 0;
            Alusrc = 0;
            RegW = 0;
            syscall=0;
            SignedExt = 1;
            RegDst = 0;
            Beq=1;
            Bne=0;
            JR=0;
            JMP = 0;
            JAL=0;
            SLLV=0;
            LB=0;
            BLTZ=0;
        end
```

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 3.9 所示。

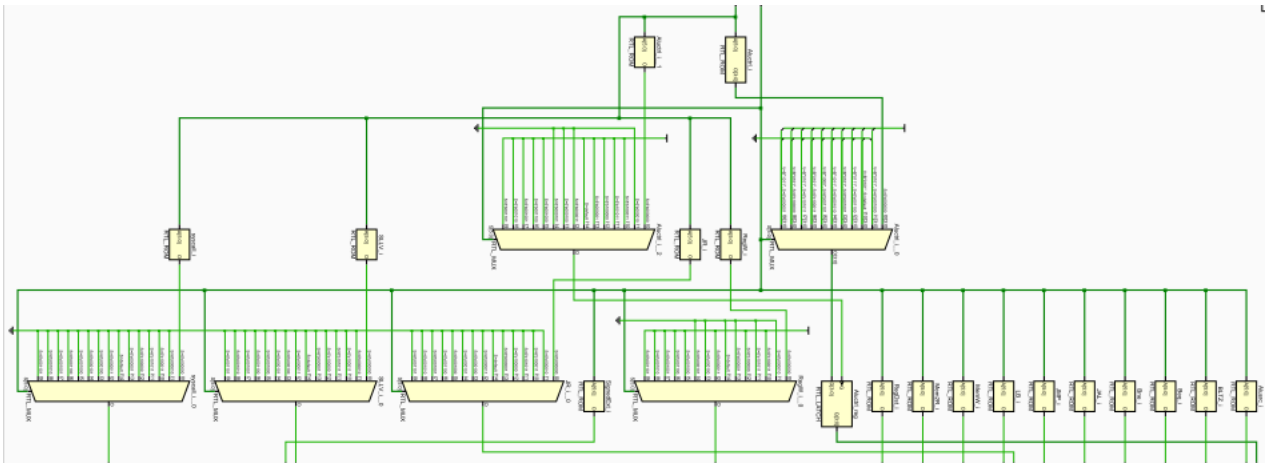

图 3.9 主控制器原理图（横置）

## 3.2 中断机制实现

### 3.2.1 中断识别响应电路

依据总体设计里提到的中断实现，完成中断识别响应电路，如图 3.10。

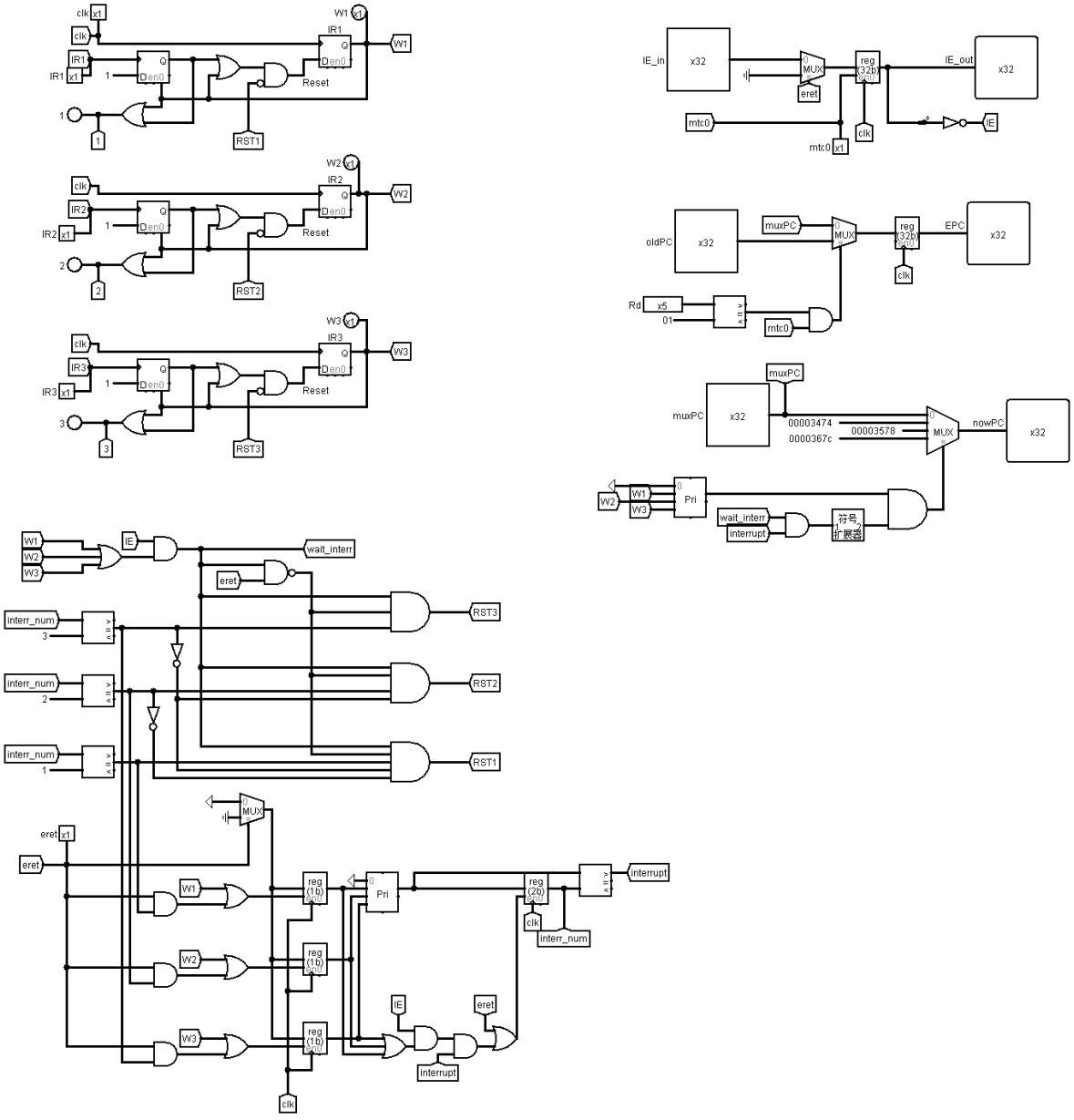


图 3.10 中断识别响应电路

左上角的电路就是中断信号产生电路，利用两个 D 触发器实现。左下角的电路作用是产生清零信号，中断等待信号，被中断信号，以及中断号。wait_interr 为中断等待信号，其逻辑十分简单。清零信号的逻辑也在设计中说明了，在这里主要讲一下 interrupt 信号，这个信号的作用是判断是否被中断，由将要执行的中断等级与现在中

断等级比较，如果前者高，那么就要被中断。右边的电路就是存储中断需要的一些数据。最上方是 IE，最低位取反作为中断时能信号，中间的是 EPC，当中断要返回时，将栈内的 PC 值送入 EPC，最下面就是中断跳转功能，4 个数字依次为正常情况下 PC 的值、1 号中断入口地址、2 号中断入口地址、3 号入口中断地址。多路选择器的信号就是中断号（在中断能被执行的情况下，否则为 0）。

### 3.2.2 添加数据通路

加入了中断机制后，必须要添加数据通路来实现中断。如图 3.11。

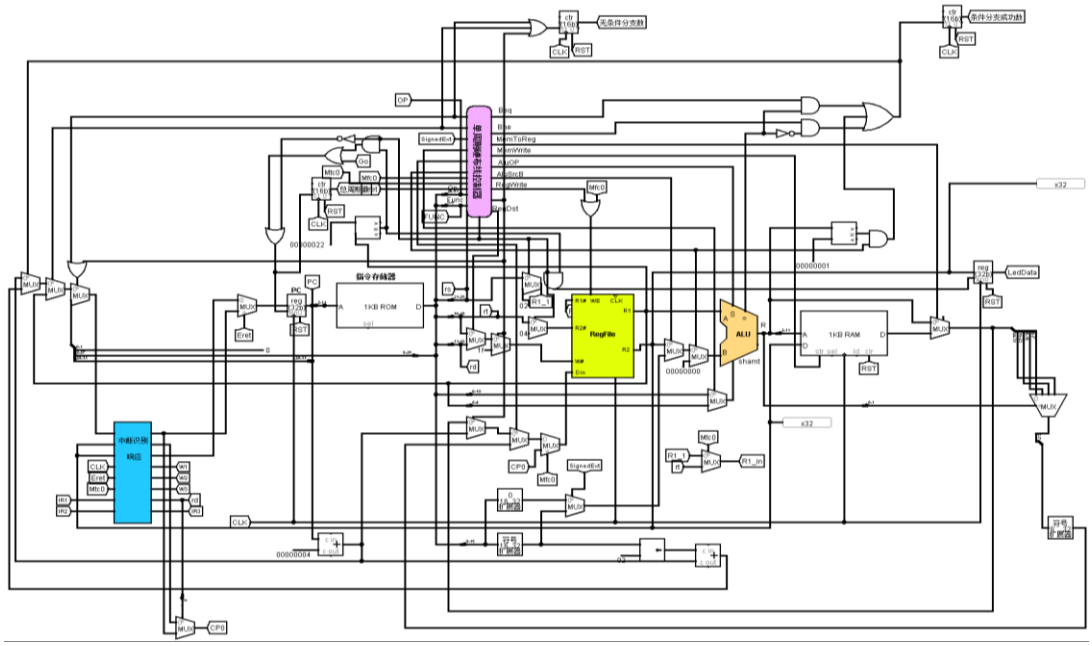

图 3.11 添加中断功能后的数据通路

左下角的部件就是图 3.10，输入输出已经很清楚了，在此不赘述。PC 寄存器，当 eret 信号为高电平时，将中断前保存的 PC 送入寄存器。同时 R1#，Din 前也要添加多路选择器，当 mfc0 信号为高电平时，选择 rt 作为 R1#输入，选择 CP0 作为输入。CP0 保存的是 IE 或者 PC 值，由 rd 编号低位来决定。完成后，只需要编写测试程序即可测试中断是否正常执行。由于篇幅限制，代码详见附件。

## 3.3 流水 CPU 实现

### 3.3.1 流水接口部件实现

① Logisim 实现

像设计里说的那样，流水接口部件就是寄存器的集合，在这里先设计 ID/EX 部件，其余的就在此基础上删除或增加某些寄存器。如图 3.12 ID/EX 流水接口部件。

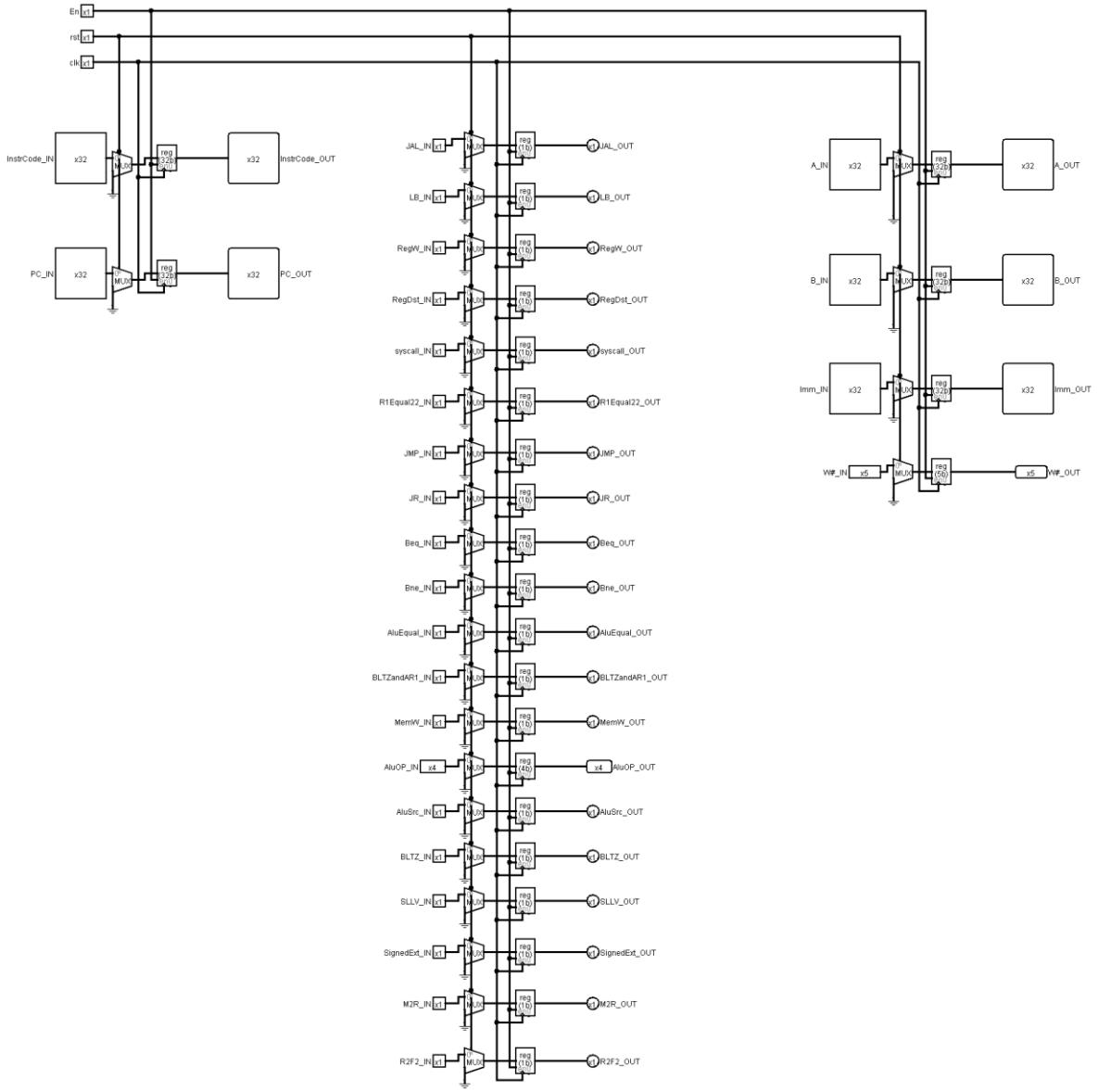


图 3.12 ID/EX 流水接口部件

② FPGA 实现

调用寄存器模块，完成接口部件的实现，这一部分十分简单，但是要细心，不然一不注意就很容易将输入输出弄错，部分 verilog 代码如下：

```
……
    reg_pipeline #(32) Imm_IN (     clk,
                                    0,
                                    ena,
```

```
                                    rst ? 0 : Imm_in ,
                                    Imm_out);

    reg_pipeline #(5) W_IN (    clk,
                                    0,
                                    ena,
                                    rst ? 0 :   WReg_in,
                                    WReg_out);

    reg_pipeline #(1) JAL_IN (    clk,
                                    0,
                                    ena,
                                    rst ? 0 :   JAL_in,
                                    JAL_out);
……
```

### 3.3.2 理想流水线实现

理想流水线主要就是数据通路的修改，如图 3.13 理想流水线数据通路。

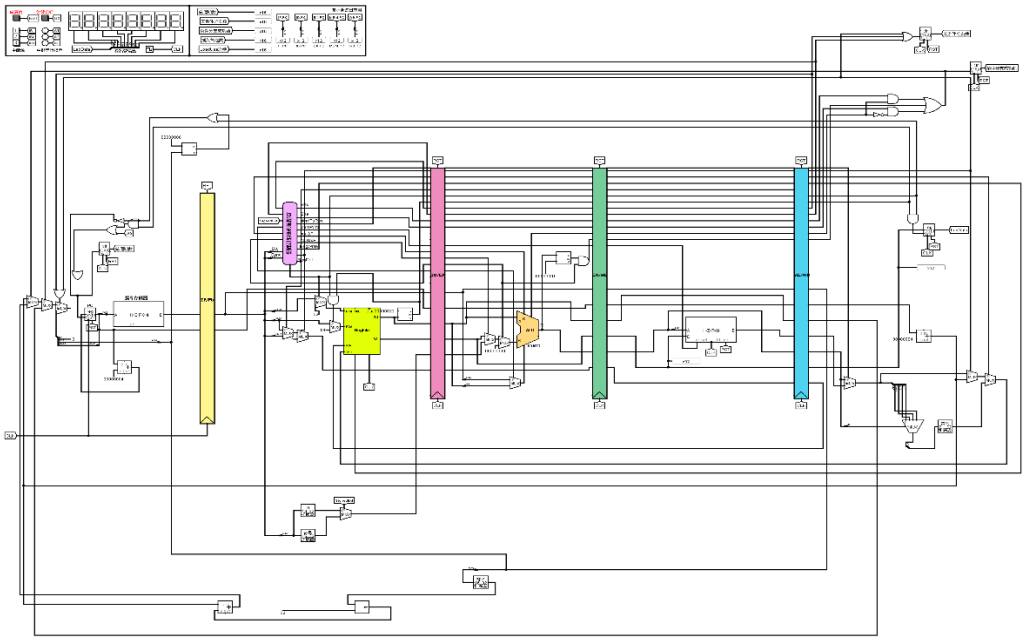

图 3.13 理想流水线数据通路

主要就是将各个阶段的需要锁存的信号锁存，传到下一阶段。

## 3.4 气泡式流水线实现

按照设计，在分支相关流水线的基础上解决数据相关问题，需要设计数据检测电路，如图 3.14 数据相关检测，其中源寄存器使用情况如何设计在上文已经进行了很明确的说明了，这里由于生成电路过于复杂，受篇幅限制无法贴图。。

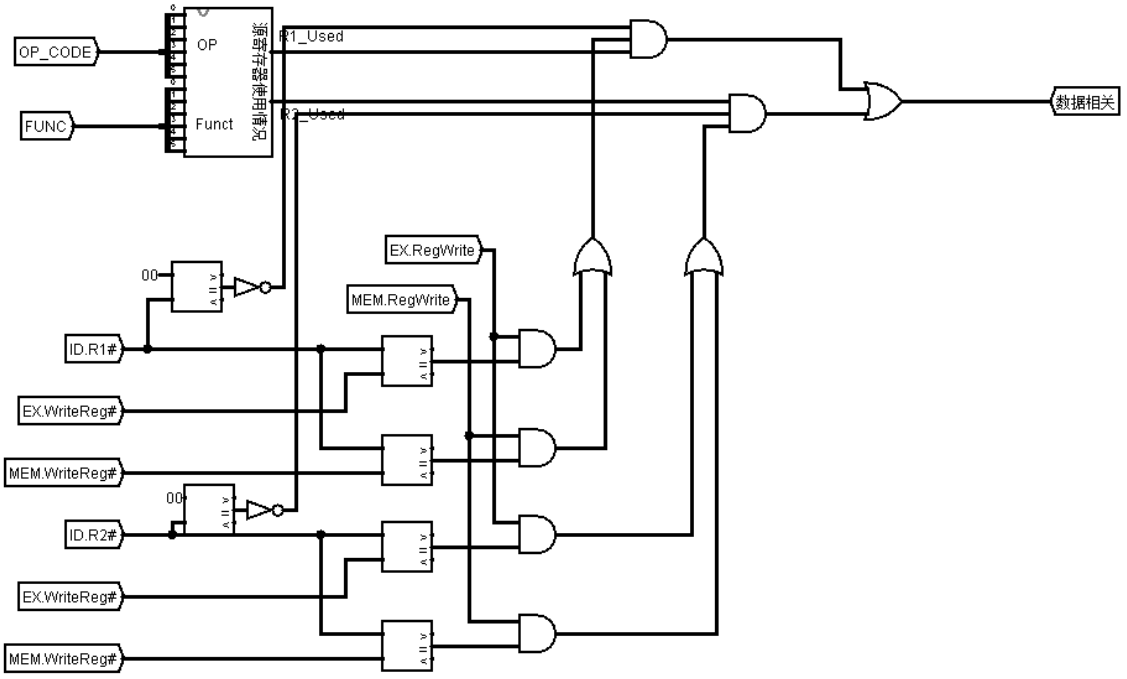


图 3.14 数据相关检测

以 R1 冲突举例，其逻辑为：（（R1#与 EX 段 W#相同）&EX 段 RegWrite 信号有效|（R1#与 MEM 段 W#相同）&MEM 段 RegWrite 信号有效）&R1 被使用&R1#不为 0.其余的冲突检测都是类似的，在此不赘述。

完成这个模块后，只需在分支流水线的 ID 段加入冲突检测逻辑，产生信号 DATA_DETC，利用这个信号，当产生冲突时，将 ID/EX 清零（插入气泡），将 IF/IF 锁存。如图 3.15 气泡流水线数据通路。

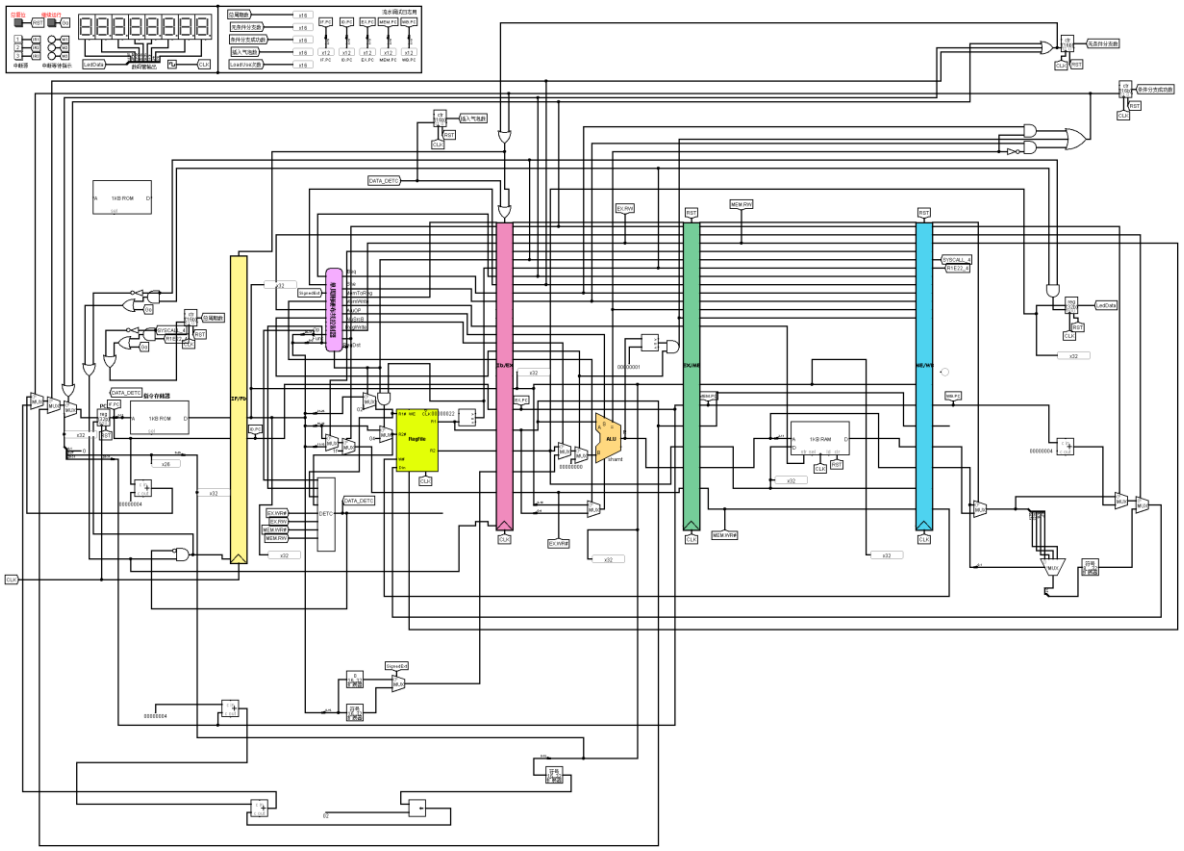

图 3.15 气泡流水线数据通路

## 3.5 数据转发流水线实现

① Logisim 实现

重定向流水线的冲突检测需要在气泡流水线冲突检测部件的基础上进行一定程度的修改，由于 RegFile 是下降沿有效，因此很好地解决了 ID 段与 MEM 段的 LoadUse 问题。如图 3.16 重定向数据相关检测。产生了四个信号 R1EX,R1MEM,R2EX,R2MEM，根据这四个信号，可以有效决定在在 ID 段的指令在 EX 段时 ALU 需要的数据的来源。其逻辑十分简单，如图 3.17。完成了这些工作之后，就要开始修改数据通路了。如图 3.18 重定向流水线数据通路，首先要注意的就是 ID 段会产生 LoadUse 信号，上文提到了四个信号，在这里也用上了，当 EX 段的指令时 LW 或 LB，并且 ID 段的 R1#或 R2#与 EX 段的 W#相同时，LoadUse 信号为高电平。利用这一信号，将 IF/ID 锁存，将 ID/EX 清零。解决玩这些后，还有一个重要的问题就是在 EX 段 ALU 的输入来源，因为是重定向，所以要明确数据来自 ID 段，还是 MEM 段还是 WB 段。这一选择的依据就是上文的四个信号，以 ALU A 为例，当 R1EX、R1MEM 都为 0 时，表示数据不冲突，因此选择 ID 段的数据；当 R1EX 为

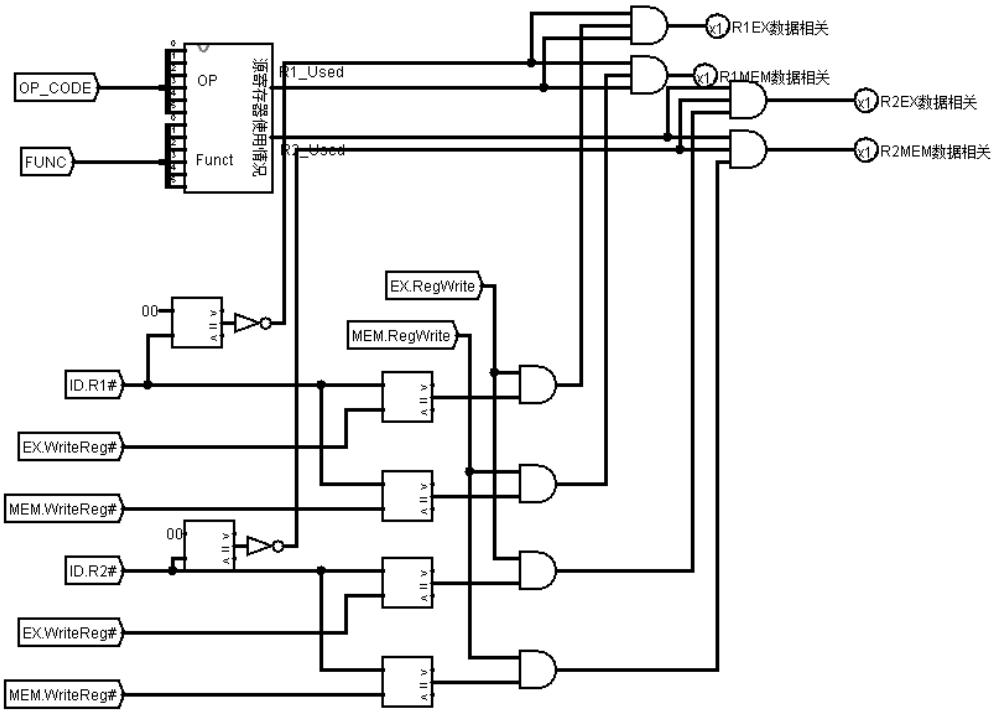


图 3.16 重定向数据相关检测

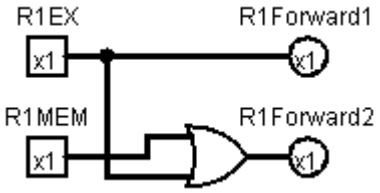


图 3.17 Forward 信号

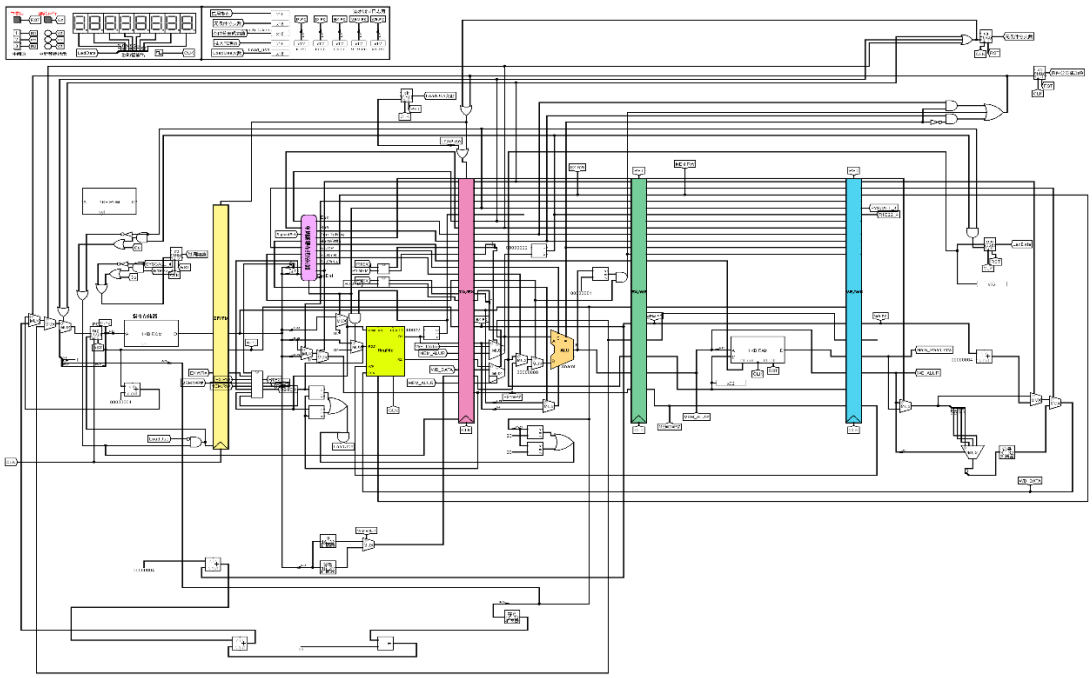

图 3.18 重定向流水线数据通路

0 时，R1MEM 为 1 时，数据来自 WB 段（MEM 段过一个阶段）。当 R1EX 为 1 时，数据来自 MEM 段（EX 段过一个阶段）。同理，ALU B 的来源与 ALU A 的设计相同，在此就不赘述了。

② FPGA 实现

参照控制器的实现方法，对每个指令，写出 R1 和 R2 时使用情况，部分 verilog 代码如下：

```
……
always@(OpCode or funct)
    begin
        case (OpCode)
            `INSTR_RTYPE_OP:
            begin
                case (funct)
                    `INSTR_ADD_FUNCT:
                    begin
                        R1_Used=1;
                        R2_Used=1;
                     end
                    `INSTR_ADDU_FUNCT:
                    begin
                        R1_Used=1;
                        R2_Used=1;
                     end
……
```

冲突检测部分的实现也比较好些，直接使用 assign 语句，依据设计里提到的逻辑实现即可，如下为部分代码：

```
……
    RegUseCondition redirect_reg_use (    OpCode,                    //指令操作码字?
                                          funct,                     //指令功能字段
                                          R1_Used,
```

```
                                        R2_Used
    );
    assign R1EX_Data_relate = R1_Used & ( ~(ID_R1 == 0)) &
  ( (ID_R1==EX_WReg)&EX_RegW );
    assign R1MEM_Data_relate = R1_Used & ( ~(ID_R1 == 0)) &
  ( (ID_R1==MEM_WReg)&MEM_RegW );
    assign R2EX_Data_relate = R2_Used & ( ~(ID_R2 == 0)) &
  ( (ID_R2==EX_WReg)&EX_RegW );
    assign R2MEM_Data_relate = R2_Used & ( ~(ID_R2 == 0)) &
  ( (ID_R2==MEM_WReg)&MEM_RegW );
……
```

最后就是 Forward 信号产生，这个也是 assign 数据流语句直接实现：

```
    assign R1Forward1 = R1EX;
    assign R1Forward2 = R1EX | R1MEM;
```

实现完需要的部件后，就可以按照图 3.18 重定向流水线数据通路，使用 verilog 编写了。编写时需要细心，仔细看清楚线的连接，以下是部分代码（篇幅限制）：

```
……
 //IF stage
    //mux pc1 instantiate
    assign PCnext1=IF_PC+4;
    assign extDataIn16=EX_instrCode[15:0];
    sign_extend ext16(extDataIn16, signextDataOut);
    assign PCnext2= (signextDataOut<<2) + EX_PC + 4 ;
    compare32 cmp_bltz(EX_aluResult, 1, EX_bltz_equal);
    assign PCBranch = (EX_Beq&EX_aluEqual) | (EX_Bne&(~EX_aluEqual)) |
  (EX_bltz_equal&EX_BLTZ);
……
```

## 3.6 动态分支预测机制实现

动态分支预测策略采用的是设计里面描述的 2 位状态机进行预测。如图 3.19。

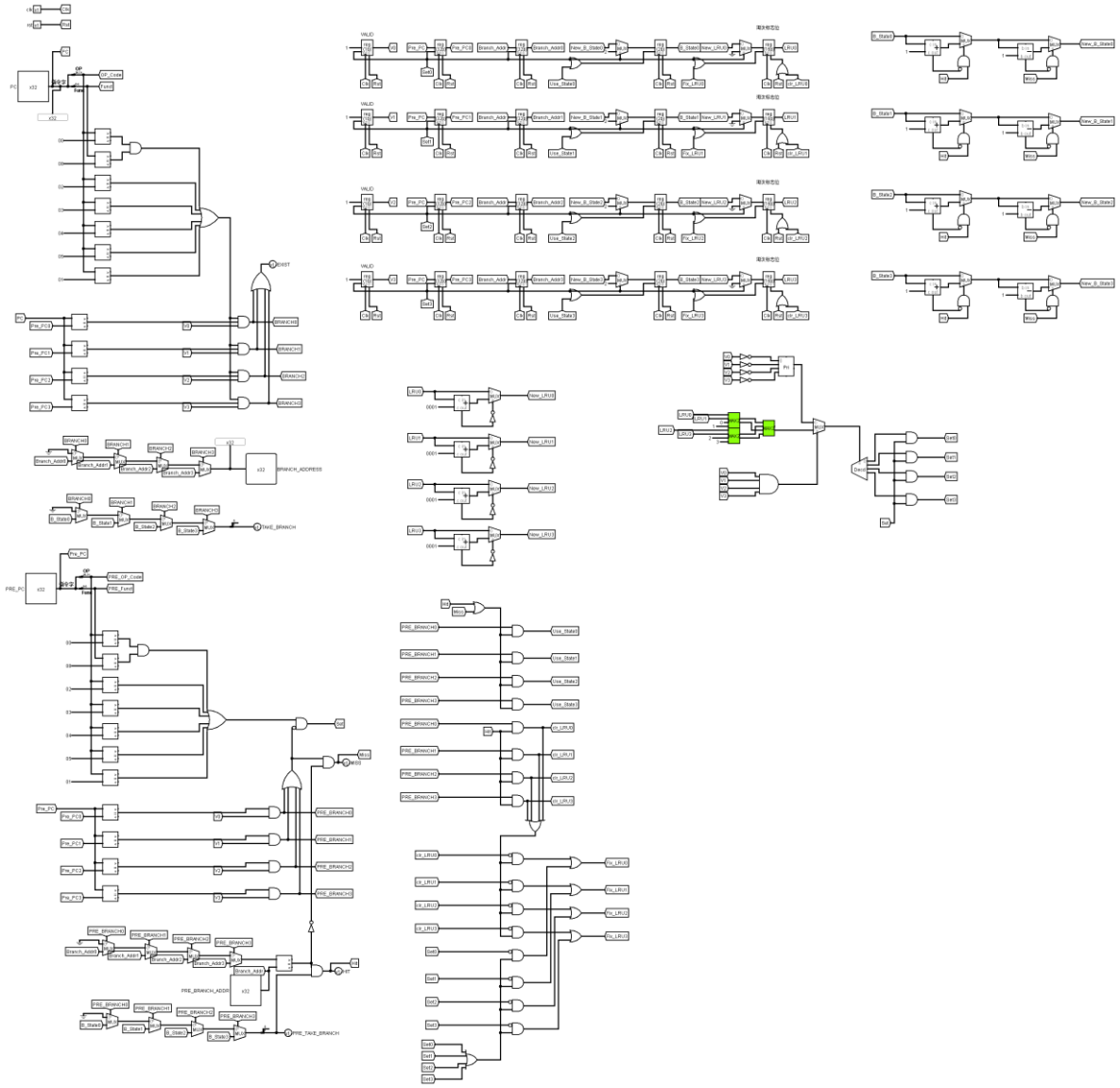

图 3.19 动态分支预测电路

电路左边是选择跳转地址逻辑，以及判断是否预测成功的逻辑。中间的为 LRU 淘汰算法电路，最右边的是状态机的更新逻辑。其中 MAX2 的功能为输出较大数的编号，在上学期 cache 实验包中，在此就不给出实现了。

与上学期比较，新增的就是状态机的状态更新，以及 Hit Miss 的逻辑，左下角的逻辑已经很清楚了，在此不赘述，这里重点讲一下状态机的状态变迁。如果命中（预测成功）并且加法没有溢出的情况下，选择原始状态+1 作为输出，否则输出即为原始状态。如果未命中（预测失败）并且减法没有溢出的情况下，选择原始状态-1 作为输出，否则输出即为原始状态。值得一提的是我将状态机的初始值设为 10，即一开始的状态就是预测跳转，这一点也提升了预测的效率。

这里还有一个重点就是数据通路的修改，如图 3.20。这里重点就是 PC 的更新是

怎么完成的。如果上一次采取了预测的方向并且预测成功了，那么直接送入 PC+4 即可（没有误取指令），否则按照原来的应该有的 PC 进行。在这之后，如果本条指令预测要跳转时，将保存的跳转地址作为输出，否则输出为上一个多路选择器的输出。在这之后，如果上一次采取了预测的方向并且预测失败了，并且不为条件跳转时，选择 PC+4 作为输出，否则输出为上一个多路选择器的输出。

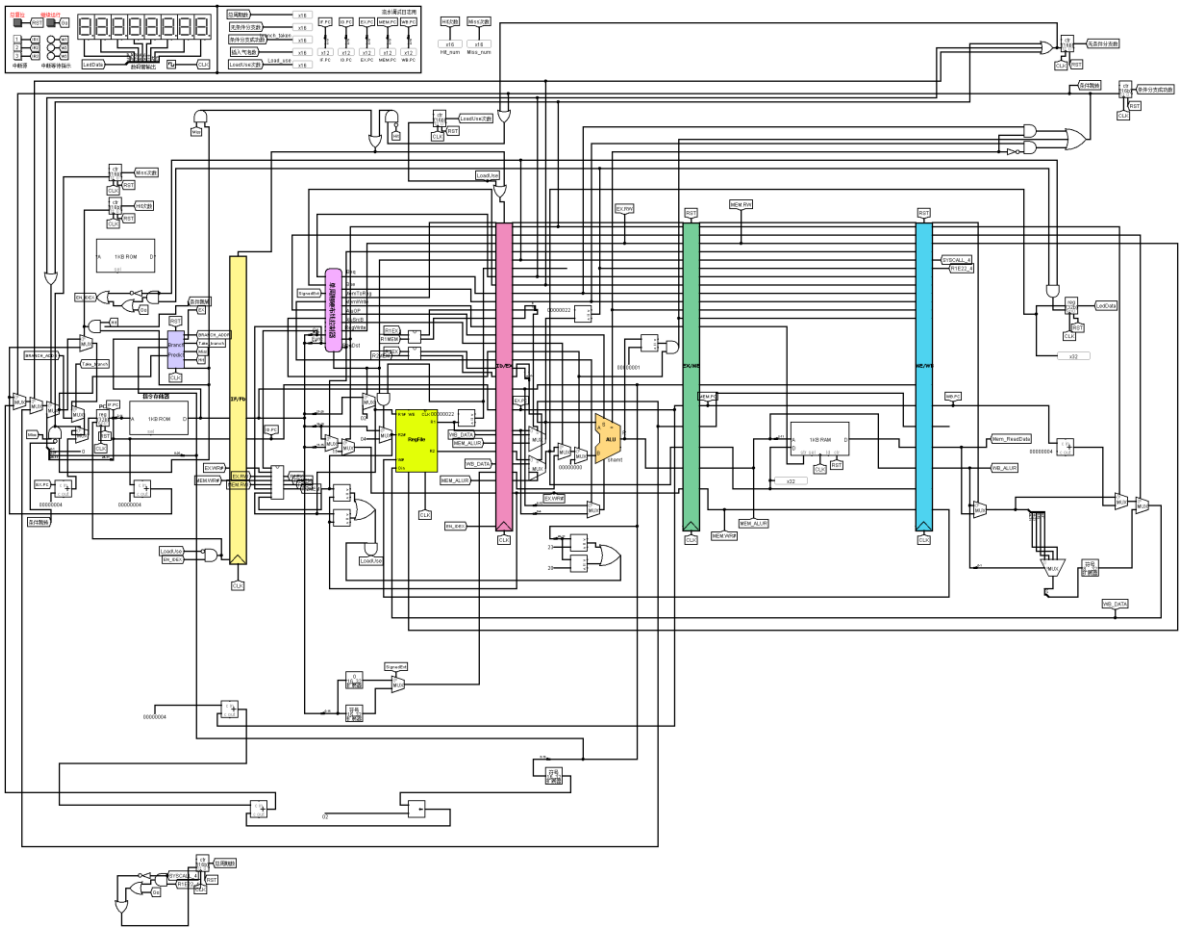

图 3.20 动态分支预测数据通路

# 4 实验过程与调试

## 4.1 测试用例和功能测试

### 4.1.1 测试用例 1 单周期上板

单周期上板已经通过了验收，在此贴出一张有纪念意义的图。

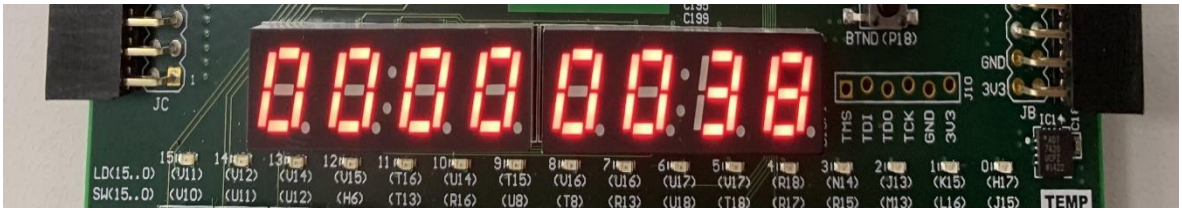


图 4.1 单周期上板

### 4.1.2 测试用例 2 重定向流水线

如图 4.2 重定向流水线 logisim，可以看到总周期数 2297，LoadUse 次数 120，符合标准。如图 4.3 图 4.4 图 4.5，可以看到 FPGA 开发板上也成功实现了，数据采用十六进制显示。

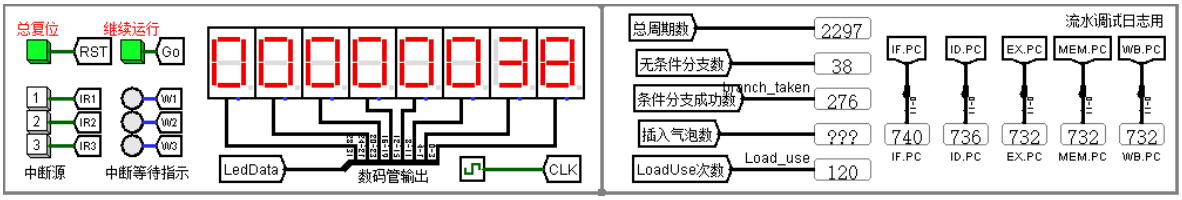


图 4.2 重定向流水线 logisim


图 4.3 LoadUse 以及总周期数


图 4.4 无条件以及条件分支数

图 4.5 数据存储器显示

### 4.1.3 测试用例 3 中断机制

分别点击 2,3,1 号中段按键，应先后进入 2→3→2→1→CPU

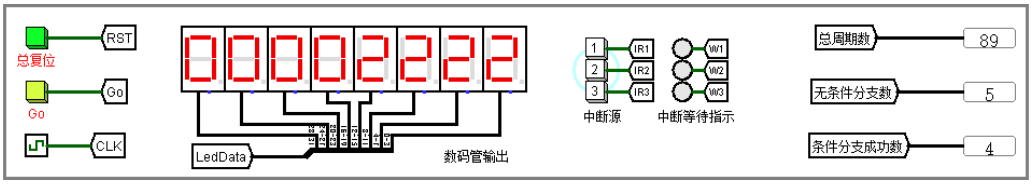


图 4.6 中断 2 号子程序

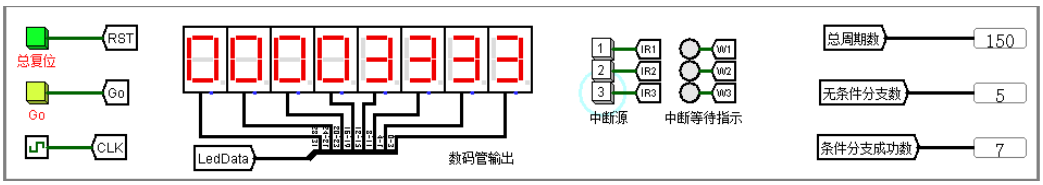


图 4.7 中断 3 号子程序

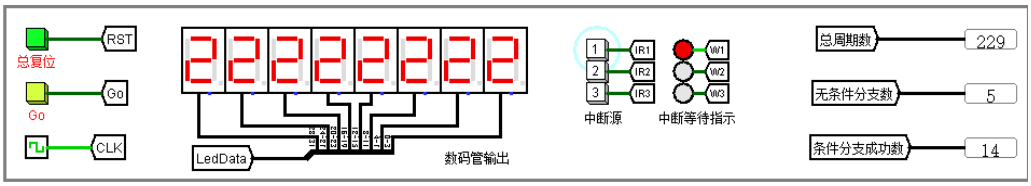


图 4.8 中断 2 号子程序

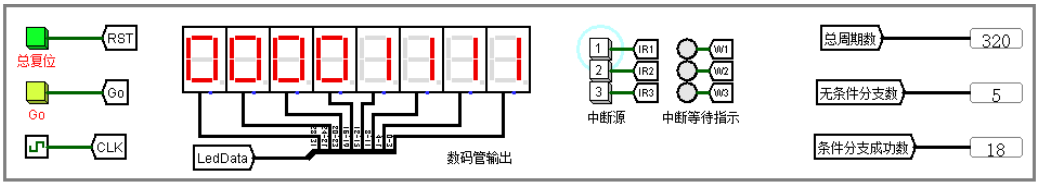


图 4.9 中断一号子程序

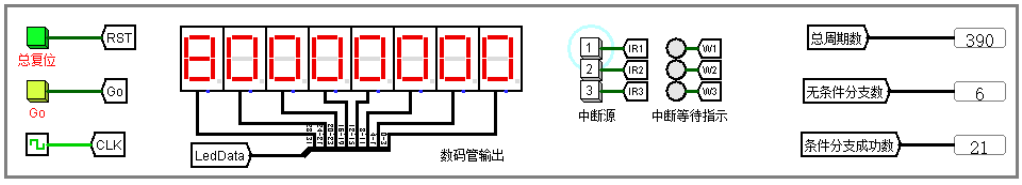


图 4.10 中断全部执行完

如上面 5 幅图，中断功能完成。

### 4.1.4 测试用例 4 动态分支预测

如图 4.11，采用动态分支预测后，总周期数减少为 1821，Hit 次数为 264，Miss 次数为 26，并且重定向周期数为 2297，可以得出 1821=2297-（264-26）*2，可知动态分支预测功能正常。其中 264-26 指的是采用动态分支预测后，减少的误取指令数，2 代表误取深度。预测正确率达到 94.3%。

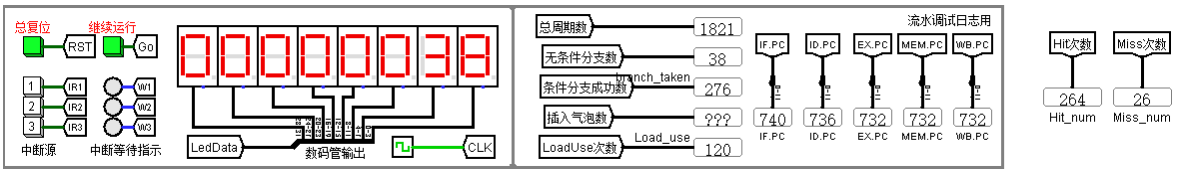


图 4.11 动态分支预测

## 4.2 性能分析

单周期 CPU 运行 benchmark 的总周期数为 1545，重定向流水线的总周期数为 2297，表面上看重定向流水线的总周期数增加了，但是流水线的 clocktime 减少了，只是采取了最长时间段（访存阶段）作为 clocktime，因此在最后的总时间上，流水线肯定快。

再将气泡流水线与重定向流水线做比较，气泡流水线的周期数为 3623，比重定向多了 1000 多个，从原理上来看，重定向流水线多了旁路结构，气泡流水线要比重定向流水线插入更多的气泡，总周期数肯定要多很多。

在重定向流水线的基础上采用动态分支策略也会减少总周期数，本次课程设计中我将周期数从 2298 减少到了 1821，减少了一定的周期数，这与程序的结构有关，比如如果在大量循环存在的程序中，动态分支预测也许会表现得更好。

从单周期到气泡流水线到重定向流水线再到动态分支预测重定向流水线，CPU 的效率是逐渐提高的，也就是 CPU 性能逐渐在提高。单周期到流水线是结构的改变，气泡流水线到重定向流水线是修改，再加上动态分支预测是有力的工具，这些改变共同提高了 CPU 的性能。

## 4.3 主要故障与调试

### 4.3.1 logisim 电路出现内部震荡故障

动态分支预测： 电路出现内部震荡。

**故障现象：**启用时钟模拟时，电路出现红线，下方提示电路出现内部震荡。

**原因分析：**如图 4.12 动态分支预测 PC 更新（当时出现问题时没有截图，这是功能正常的图），在这里的逻辑是有点复杂的，多路选择器的顺序不对的话会造成整个 CPU 功能不正常，出现这个问题的时候，我上网去查了相关问题的解决方法，但是没有答案。想到震荡出现的原因是频繁替换，于是我首先检查了动态分支预测电路的问题，发现问题不出在那里。之后我检查了下图的逻辑，发现当时在考虑的时候，思维有点混乱，于是连线上就出现了很大的问题。出现震荡的原因是在某一部分电路中，我把电路的输出重新作为这部分电路的输入，于是就出现了震荡的现象。

**解决方案：**在重新思考 PC 是怎样更新的，解决逻辑上的问题，没有出现把一个电路的输出作为这个电路的输入的情况。

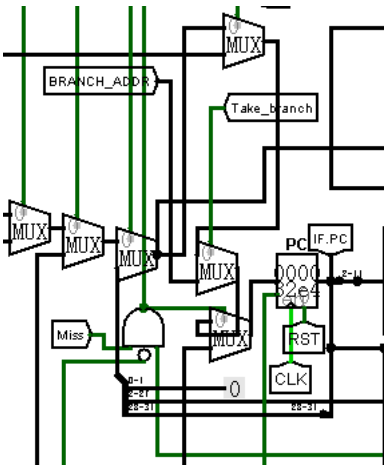


图 4.12 动态分支预测 PC 更新

### 4.3.2 FPGA 重定向流水线仿真故障

Verilog 实现重定向流水线：最后周期数统计错误，内存排序错误。

**故障现象：**由于当时没有截图，所以在这里没有用图片。进行行为仿真的时候，发现总周期数不对，条件跳转分支数不对，内存排序不对。

**原因分析：**条件跳转分指数出现错误的原因大概率是排序不对，所以我又检查了内存排序，发现内存排序果然是混乱的。我首先是重新检查了一下数据通路，把软件生成的 RTL 级电路图重新看了一遍，发现没有问题。于是我想到重定向流水线相比单周期增加的模块就那么几个，于是我又去看了源寄存器使用情况模块，仔细对照表格，没有问题。当时找了好久，最后才找到是 Forward 信号产生出现了问题，当时真的没有想到会出现这么低级的错误，所以一直没有看这部分。 就是下面几条语句：

```
assign R1EX_Data_relate = R1_Used & ( ~(ID_R1 == 0)) &
( (ID_R1==EX_WReg)&EX_RegW );
assign R1MEM_Data_relate = R1_Used & ( ~(ID_R1 == 0)) &
( (ID_R1==MEM_WReg)&MEM_RegW );
assign R2EX_Data_relate = R2_Used & ( ~(ID_R2 == 0)) &
( (ID_R2==EX_WReg)&EX_RegW );
assign R2MEM_Data_relate = R2_Used & ( ~(ID_R2 == 0)) &
( (ID_R2==MEM_WReg)&MEM_RegW );
```

当时写这个的时候是直接把上面类似的语句复制下来的，结果后面(ID_R2==MEM_WReg)&MEM_RegW 中的 ID_R2（直接复制下来是 ID_R1）没有改，导致错误。

**解决方案：**修改语句即可。

由于篇幅限制，在这里只能列出有限故障。

## 4.4 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，完成了运算控制器、控制信号生成模块。 |
| 第二天 | 完成了数据通路的编写，测试了 benchmark 程序，有待改进 |
| 第三天 | 修正了 bug,完善了显示模块,通过了 benchmark_ccmb 开发板测试 |
| 第四天 | 完成理想流水线，分支相关流水线，气泡流水线待完成 |
| 第五天 | 完成气泡流水线、重定向流水线以及动态分支预测 |
| 第六天 | 完成单级中断、多级中断 |
| 第七天 | 流水线上板，周期数调试中 |
| 第八天 | 完成流水线上板 |
| 第九天 | 开始完成课程设计报告 |

# 5 设计总结与心得

## 5.1 课设总结

本次实验的主要目标是完成五段流水线 CPU 的上板。基础是单周期 CPU 的上板，额外具有挑战性的任务是中断机制和动态分支预测的设计。作了如下几点工作：

1） 设计了单周期 CPU 每条指令的控制信号产生逻辑，实现了单周期硬布线控制器。

2） 设计了单周期 CPU 数据通路，在 logisim 平台上完成了单周期 CPU。

3） 完成了单周期 CPU FPGA 上板。

4） 设计了流水接口部件 IF/ID、ID/EX、EX/MEM、MEM/WB。

5） 完成了理想流水线。

6） 设计了分支相关处理机制，完成了分支相关流水线。

7） 设计了数据相关检测机制，完成了相应电路以及气泡流水线。

8） 设计了旁路机制以及重定向 LoadUse 检测机制，重新设计了重定向数据相关检测机制，完成了重定向流水线。

9） 设计了中断机制，完成了中断识别与响应电路。

10） 重新设计了单周期 CPU 数据通路，使它能完成中断功能，完成了多级中断的实现。

11） 设计了动态分支预测策略，实现了预测电路，完成了具有预测功能的重定向流水线，提高了性能。

12） 依据 logisim 上的重定向流水线数据通路图，完成了重定向流水线 FPGA 上板。

## 5.2 课设心得

本次课程设计和其他的课程有很大不同，体现在制度以及难度上。本次课设采用天梯制与组队方式来完成。在刚开始做课设的时候，没有遇到太大困难，可是到后来的扩展任务时，就有挑战的意思了。两周的课设时间以及周末的加班才让我在第二周顺利达到了天梯最高点。这其中的过程是很艰辛的，但是现在回过头来看，也会怀念

那种感觉。通过这次课程设计，我也收获了很多。

课程设计的前置任务是寒假布置的 logisim 28 条指令 CPU 的设计，这个任务比较简单，只花了几个小时就设计出来了。然后课程设计一开始的要求就是完成单周期 CPU 的上板，由于之前组原实验用的都是 logisim，很久没有用 verilog 了，并且 logisim 很方便，线的连接是看的非常清楚的。所以一开始上手是有一定难度的，好在分组能够减轻每个组员的工作量，能够简化设计步骤，在有了所有的模块之后，我们需要做的就只有把 logisim 的数据通路用 verilog 实现就好。尽管现在看来这是一个比较简单的任务，但是当时 verilog 语言不熟练，并且如果 vivado 仿真出错了的话，要找错误比较麻烦，不像 logisim 那样 debug 很方便。而且数据通路的编写是一个比较枯燥繁琐的过程，线的连接不能出一点错，需要注意力集中。由于线路很多，每条线的命名也是一个大问题，最好是有一个规范，这样在后期 debug 的时候能够更快速定位错误。而且在这里最重要的是每个组员编写的模块不能出错，一旦某个模块出错，那么就会给后续的工作带来很多麻烦，而且 debug 的时候只会去找数据通路的问题，反而忽略了模块。

完成单周期上板后，就来到了本次课程设计的重点，流水线的实现。上学期的课程中并没有介绍流水线的相关知识，好在老师给了十分实用的资源——mooc，在 mooc 中首先是介绍了流水线的基本概念，思想。并且讲述了流水线的问题该怎么解决，这里主要是数据相关问题。mooc 的讲解简单易懂，于是流水线这一部分就没有什么难度了。一开始完成理想流水线，就要设计流水线的基本部件——流水接口。之后只要稍微修改一下数据通路，就可以完成，然而这只是最基本的一步。在后面的实验中，一步一步解决了分支相关，数据相关的问题。只要将数据为什么会有相关性，以及出现了这种相关性后该怎么解决弄明白了之后，气泡流水线没有什么难度。之后的重定向流水线更加直接，你这一阶段要用数据，我不必等待，直接将其他段产生的数据提前给你就可以了，这也就是旁路机制。这一设计大大增加了流水线的效率，但是这一方式仍然无法解决 LoadUse 问题，当 LW 或 LB 指令还没有将数据写入寄存器堆，而另外的指令又要用到相应寄存器的值时，就产生了 LoadUse 相关。解决这一问题就只能靠插入气泡了。最后的实验结果，验证了重定向流水线性能更佳。

在课设时，我完成了 logisim 的重定向流水线后就直接开始了动态分支预测的设计，在现在看来，可以先把流水线的上板先完成了。因为流水线的上板与单周期没有什么区别，只是增加了流水接口部件（寄存器的集合，设计简单），冲突检测电路等

实现，数据通路的编写方法与单周期一样，只要掌握了单周期上板，多周期上板很快就能实现。又说回来，在完成动态分支预测的时候，首先是上网查阅了资料，发现有很详细的资料，它是一篇英文文献，里面描述了很多动态分支预测的方法，但是由于时间关系，只能采取最简单的一种策略。这种策略在设计中已经详细阐述了，结合上学期 cache 实验，就能比较快地完成这一部分。在这一实验出现了花费我很长时间的 bug，具体在故障分析里有给出，这一 bug 出现又给我增长了很多经验。

最后就是最难的中断设计了，这一设计没有太多的资料，给的资料太多，看了 see mips run 这本书，发现在有限的时间不可能设计出完善的带 CP0 协处理器的中断处理电路。但是了解了中断的硬件电路是怎么实现的，这里最主要的就是中断号的识别，以及屏蔽字的设置，屏蔽字决定了中断是否能背中断，但是在这里只是利用了优先选择器的天然特性以及题目的要求，可以用这个来判断中断是否能被中断，具体的设计见上文。

在完成了所有的工作后，我是很有成就感的，毕竟两周时间可以完成这样的实验，是一件非常高兴的事，并且两周时间是很合理的，既不紧又不松，很好。而且天梯制度是非常好的，这让我们能够积极去追求通关。

对这次课设的内容以及制度，我还有一些小的建议。首先是分组的问题，我觉得这种选秀方式比较合理，避免了能力强的人扎堆的问题，能够均匀化，帮助所有同学完成课程设计，并有所收获。但是合作的话，单周期是每人编写一些模块，数据通路就是自己设计了。这种合作其实还是不太够的，特别是到了后面中断这种比较难的任务，合作就基本没有了，每个人都会花时间去看文献，然后课外自己做一些工作，导致课上的进度不同，很难实现讨论。但是这种问题又不好解决，每个人的心理不同，自然行动也不同。内容的话可以适当简化一些简单的任务，比如流水线部分可以直接做重定向流水线，我觉得根据 mooc 的讲解这也是没问题的。可以把时间多留给后面的挑战任务，比如优化动态分支预测，优化中断机制。

最后感谢课程组老师带来这样一次课程设计，也感谢老师的耐心解答以及组员之间的相互讨论。在大学期间能够完成这样一次课程设计，是十分快乐的，让人真正感到学有所得。

# 参考文献

[1] DAVID A.PATTERSON(美).计算机组成与设计硬件/软件接口(原书第 4 版).北京：机械工业出版社.

[2] David Money Harris(美).数字设计和计算机体系结构（第二版）. 机械工业出版社

[3] 秦磊华，吴非，莫正坤.计算机组成原理. 北京：清华大学出版社，2011 年.

[4] 谭志虎,秦磊华,胡迪青.计算机组成原理实践教程.北京:清华大学出版社,2018.

[5] 袁春风编著. 计算机组成与系统结构. 北京：清华大学出版社，2011 年.

[6] 张晨曦，王志英. 计算机系统结构. 高等教育出版社，2008 年.

[7] James Smith. A study of branch prediction strategies. IEEE Computer Society Press Los Alamitos, CA, USA ©1981.

[8] Dominic Sweetman. See MIPS Run(Second Edition) MORGAN KAUFMANN, 2007

## 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:**