***2018***



**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： | CS1501 |
| 学 号： | U20151477 |
| 姓 名： | 华龙 |
| 电 话： | 15927598966 |
| 邮 件： | [709603507@qq.com](mailto:709603507@qq.com) |
| 完成日期： | 2018-03-09 |

目 录

[1 课程设计概述 3](#_Toc510920703)

[1.1 课设目的 3](#_Toc510920704)

[1.2 设计任务 3](#_Toc510920705)

[1.3 设计要求 3](#_Toc510920706)

[1.4 技术指标 4](#_Toc510920707)

[2 总体方案设计 6](#_Toc510920708)

[2.1 单周期CPU设计 6](#_Toc510920709)

[2.2 流水CPU设计 10](#_Toc510920710)

[2.3 理想流水线设计 12](#_Toc510920711)

[2.4 气泡式流水线设计 12](#_Toc510920712)

[2.5 重定向流水线设计 14](#_Toc510920713)

[3 详细设计与实现 16](#_Toc510920714)

[3.1 单周期CPU实现 16](#_Toc510920715)

[3.2 理想流水CPU实现 29](#_Toc510920716)

[3.3 气泡式流水线实现 30](#_Toc510920717)

[3.4 重定向流水线实现 32](#_Toc510920718)

[4 实验过程与调试 35](#_Toc510920719)

[4.1 单周期CPU上板测试 35](#_Toc510920720)

[4.2 理想流水线CPU功能测试 35](#_Toc510920721)

[4.3 气泡流水线CPU功能测试 36](#_Toc510920722)

[4.4 重定向流水线CPU功能测试 37](#_Toc510920723)

[4.5 性能分析 37](#_Toc510920724)

[4.6 实验进度 38](#_Toc510920725)

[5 设计总结与心得 39](#_Toc510920726)

[5.1 课设总结 39](#_Toc510920727)

[5.2 课设心得 39](#_Toc510920728)

[参考文献 41](#_Toc510920729)

# 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所中的前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

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 |
| 24 | SYSCALL | 系统调用 | If $v0==10 halt(停机指令)  else 数码管显示$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 | SRLV | 逻辑可变右移 |  |
| 29 | SLTIU | 小于立即数置 1(无符号) |  |
| 30 | LH | 加载半字 |  |
| 31 | BLTZ | 小于 0 转移 |  |

# 总体方案设计

## 单周期CPU设计

结构设计采用指令存储器与数据存储器分开的哈佛结构，控制器采用硬布线电路实现，总体结构图如图 2.1所示。



图 . 单周期CPU总体结构图

### 主要功能部件

#### 程序计数器PC

PC计数器只需一个32位输入引脚和一个32位输出引脚，然后用一个32位加法器使输出的一直比输入的值多4即可。

#### 指令存储器IM

指令存储器用ROM实现，指令存放好后不能再修改，所以只需一个12位的地址输入和一个32位的指令输出。

#### 数据存储器DM

数据存储器用RAM实现，给定地址即输出数据，而数据只在时钟下降沿且写信号有效的情况下才存入对应的地址中，DM的引脚与功能描述见表 2.1。

表 2.1 DM引脚与功能描述

| 引脚 | 输入/输出 | 位宽 | 功能描述 |
| --- | --- | --- | --- |
| Address | 输入 | 32 | 要访问的地址 |
| DataIn | 输入 | 32 | 要存入的数据 |
| WE | 输入 | 1 | 写使能端，为1有效 |
| clk | 输入 | 1 | 时钟输入，数据在时钟下降沿写入 |
| clr | 输入 | 1 | 清零端，为1时清空RAM数据 |
| DataOut | 输出 | 32 | 数据输出 |

#### 运算器ALU

运算器包含基本的运算功能，给定输入即开始计算，最终的输出依据ALUOP用多路选择器选择，其引脚与功能描述见表 2.2。

表 2.2 ALU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 |

#### 寄存器堆RF

寄存器堆共包含32个寄存器，一旦指定要读取的寄存器的编号，便从对应输出寄存器的值。其中的0号寄存器一直为0，不可被修改，其他寄存器均可在写使能端有效时，在时钟下降沿被修改为输入的数据。RF的引脚与功能描述见表 2.3。

表 2.3 RF引脚与功能描述

| 引脚 | 输入/输出 | 位宽 | 功能描述 |
| --- | --- | --- | --- |
| clk | 输入 | 1 | 时钟输入，在下降沿将数据写入寄存器 |
| rA | 输入 | 5 | 要在A输出的寄存器编号 |
| rB | 输入 | 5 | 要在B输出的寄存器编号 |
| rW | 输入 | 5 | 要写入的寄存编号 |
| W | 输入 | 32 | 要写入寄存器的 数据 |
| WE | 输入 | 1 | 写使能端，为1有效 |
| A | 输出 | 32 | 输出rA指定的寄存器中的值 |
| B | 输出 | 32 | 输出rB指定的寄存器中的值 |

### 数据通路的设计

使用工程化方法，绘制主要功能部件输入来源表，用于描述各部件之间的连接关系，记录各部件输入端数据来源，并将表中部件输入按列进行合并，对于多输入来源的输入，引入多路选择器，同时新增多路选择器选择控制信号，对于空输入，或其他输入不影响指令执行，可直接忽略，否则需要增加额外的控制电路。主要功能部件的输入来源表框架如表 2.4所示。

表 2.4 指令系统数据通路框架

| 指令 | PC | IM | RF | | | | ALU | | | DM | | Tube |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| R1# | R2# | W# | Din | A | B | OP | Addr | Din |
|  |  |  |  |  |  |  |  |  |  |  |  |  |

### 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所示。

表 2.5 控制器控制信号的作用说明

| 控制信号 | 取值 | 说明 |
| --- | --- | --- |
| ALUOP | 0000 | Result = X << Y 逻辑左移 |
| 0001 | Result = X >>>Y 算术右移 |
| 0010 | Result = X >> Y 逻辑右移 |
| 0101 | Result = X + Y |
| 0110 | Result = X - Y |
| 0111 | Result = X & Y |
| 1000 | Result = X | Y |
| 1010 | Result = ~(X |Y) |
| 1011 | Result = (X < Y) ? 1 : 0 符号比较 |
| 1100 | Result = (X < Y) ? 1 : 0 无符号比较 |
| rA<=$v0 | 0 | A引脚输出$v0寄存器的值 |
| 1 | A引脚输出rs指定的寄存器的值 |
| rB<=$a0 | 0 | B引脚输出$a0寄存器的值 |
| 1 | B引脚输出rt指定的寄存器的值 |
| rW<=rd | 0 | 将数据写入rt指定的寄存器 |
| 1 | 将数据写入rd指定的寄存器 |
| rW<=GPR[31] | 0 | 将数据写入rt或rd指定的寄存器 |
| 1 | 将数据写入31号寄存器 |
| RF\_WE | 0 | 不修改寄存器的值 |
| 1 | 在时钟下降沿修改寄存器的值 |
| Din<=PC+1 | 0 | 将ALU输出的值存入寄存器 |
| 1 | 将PC+1的值存入寄存器 |
| Din<=DM | 0 | 将ALU输出或PC+1的值存入寄存器 |
| 1 | 将DM输出内容存入寄存器 |
| Din<=DM.half | 0 | 将DM输出的字内容存入寄存器 |
| 1 | 将DM输出的半字内容存入寄存器 |
| X<=RF.B | 0 | ALU的操作数X为RF的A的输出 |
| 1 | ALU的操作数X为RF的B的输出 |
| Y<=RF.A | 0 | ALU的操作数Y为RF的B的输出 |
| 1 | ALU的操作数Y为RF的A的输出 |
| Y<=S\_EXT | 0 | ALU的操作数Y为RF的A或B的输出 |
| 1 | ALU的操作数Y为位扩展器的输出 |
| extend(shamt) | 0 | 扩展立即数0xa |
| 1 | 扩展shamt字段 |
| extend(IMM) | 0 | 扩展shamt字段或立即数0xa |
| 1 | 扩展IMM字段 |
| zero\_ext(IMM) | 0 | 符号扩展IMM字段 |
| 1 | 补0扩展IMM字段 |
| jump | 0 | PC的输入为PC+1或有条件跳转的目标地址 |
| 1 | PC的输入为无条件跳转的目标地址 |
| jump register | 0 | 不是jr指令 |
| 1 | 是jr指令 |
| beq | 0 | 不是beq指令 |
| 1 | 是beq指令 |
| bne | 0 | 不是bne指令 |
| 1 | 是bne指令 |
| bltz | 0 | 不是bltz指令 |
| 1 | 是bltz指令 |
| syscall | 0 | 不是syscall指令 |
| 1 | 是syscall指令 |

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，控制信号的取值默认为0，以简化控制器电路的设计。该控制信号表的框架如表 2.6所示。

表 2.6 主控制器控制信号框架

| 指令 | R | RW | WE | X | EXT | Y | ALUop | MemWrite | MemRead | Din | Branch | SYSCALL |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|
|  |  |  |  |  |  |  |  |  |  |  |  |  |

## 流水CPU设计

### 总体设计

如图 2.2所示，将指令执行过程细分为取指令IF、指令译码ID、指令执行EX、访存MEM、写回WB共五个阶段，其中IF段包括程序计数器PC，NPC下址逻辑，指令存储器等功能模块；ID段包括控制器逻辑、取操作数逻辑；EX段主要包括运算器模块；MEM段主要包括内存读写模块，写回WB段主要包括寄存器写入控制模块。在每个阶段的后面增加一个锁存器，即流水接口部件，用于锁存本段的处理完成的数据或结果，以保证该阶段的执行结果给下一个阶段使用。依据如图 2.1所示的单周期CPU结构图，增加流水部件后的五段流水线CPU总体结构图如图 2.3所示。

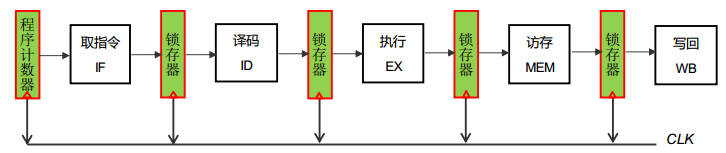


图 . 指令流水线逻辑架构

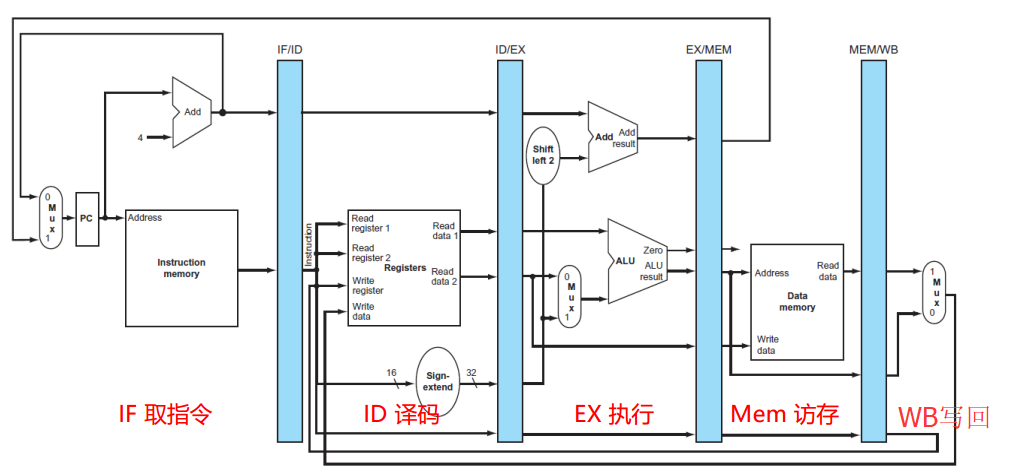


图 . 五段流水线CPU总体结构

### 流水接口部件设计

流水接口部件需要包含三个控制输入引脚、若干个数据输入引脚和数据输出引脚，内部用若干个寄存器实现锁存功能，以IF/ID接口部件为例，其电路原理图如图 2.4所示，其中CLK引脚为同步时钟输入端，lock为锁定引脚，当lock为1时，寄存器在时钟上升沿更新为输入的内容，当lock为0时，寄存器的值变，同步清零引脚则实现的是寄存器的同步清零功能，当lock和同步清零端都为1时，寄存器的值清零，这个设计对后面的非理想流水线中插入气泡的实现，起着至关重要的作用。其他流水接口部件与IF/ID接口部件类似，不同之处仅仅在于保存的数据不同。

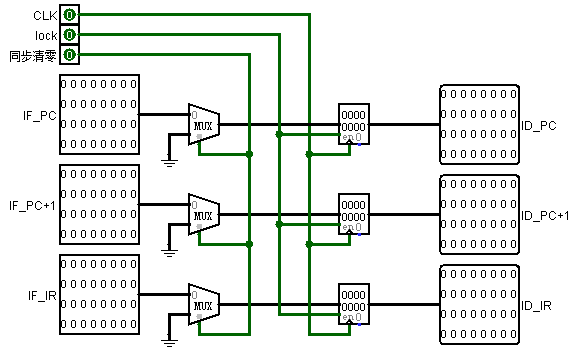


图 . IF/ID流水接口部件原理图

## 理想流水线设计

由于理想流水线CPU无需考虑冲突冒险的处理，因此只需根据上一结的大概设计，将单周期CPU的数据通路改造成分为五段的流水线数据通路即可。

需要补充的是，流水线通过流水接口部件为后段提供数据信息，控制信息，向前段传递反馈信息，流水线后段对数据的加工处理依赖于前段通过流水接口部件传递过来的信息。最主要的是ID段译码生成该指令的所有控制信号，该控制信号需要通过锁存器逐段向后传递，从而后段功能部件所需的控制信号不需要单独生成，直接从锁存器获取即可。

再者，为了方便调试，各阶段的PC值也应当逐段向后传递，以便获得程序运行过程的时空图。

## 气泡式流水线设计

在理想流水线CPU的基础上，增加数据相关检测逻辑、插入气泡逻辑、流水暂停逻辑和分支冲突处理逻辑，使得该流水线能处理数据冲突，控制冲突，结构冲突等所有流水冲突。

### 数据相关检测逻辑

由于ID段需要取操作数，所以数据相关检测逻辑应设置在ID段。图 2.3所示的五段流水结构中，数据只能在WB段写入，所以ID段正在处理的指令与EX、MEM、WB段的指令都存在数据相关。但是ID段与WB段的数据相关，可通过设置寄存器的值在时钟下降沿跟新，各阶段PC在时钟上升沿更新来解决，所以只需在ID段检测当前指令与EX段和MEM段的指令是否存在数据相关即可。判断数据相关的依据则是比较当前指令是否存在源操作数和后续段的目的操作数是否相同。

数据相关检测模块的引脚定义及说明如表 2.7所示。

表 2.7 数据相关检测模块引脚定义

| 引脚 | 输入/输出 | 位宽 | 功能描述 |
| --- | --- | --- | --- |
| rA | 输入 | 5 | ID段指令的源操作数 |
| rB | 输入 | 5 | ID段指令的源操作数 |
| ID\_OP | 输入 | 6 | ID段指令的操作码，用于解析指令类型 |
| ID\_Funct | 输入 | 6 | ID段指令的功能码，用于解析指令类型 |
| EX\_OP | 输入 | 6 | EX段指令的操作码，判断Load-Use情况 |
| EX\_rW | 输入 | 5 | EX段指令的目的操作数 |
| EX\_WE | 输入 | 1 | EX段指令的寄存器写使能端控制信号 |
| MEM\_rW | 输入 | 5 | MEM段指令的目的操作数 |
| MEM\_WE | 输入 | 1 | MEM段指令的寄存器写使能端控制信号 |
| Load-Use | 输出 | 1 | Load-Use情况 |
| Bubble | 输出 | 1 | 存在数据冲突 |

### 插入气泡和流水暂停逻辑

当出现数据冲突的时候，需要暂停流水先，插入气泡，以等待冲突解除，具体实现可通过在EX段插入气泡，IF、ID 段暂停，即，使PC计数器的使能端和IF/ID接口部件的传输端无效，以保存当前的值,即暂停流水，而令ID/EX接口部件的输出清零，以实现插入气泡。

### 分支冲突处理逻辑

由于在EX段处理分支跳转，所以指令成功跳转时，需要将误取到IF和ID段的指令清除，相当于插入两个气泡，具体实现为IF/ID接口部件和ID/EX接口部件的清零端都置为有效，如此一来，在下一个时钟上升沿到来的时候，IF段的指令就为正确的指令，ID与EX段的指令为nop指令，程序就得以正确运行。

## 重定向流水线设计

进一步改造气泡流水线CPU，增加重定向机制和Load-Use数据相关检测机制，使得流水线能在不插入气泡的情况下处理除Load-Use情况之外的其他数据相关问题。

### 重定向数据通路

将EX/MEM中锁存的运算器运算结果、MEM/WB中锁存的运算器运算结果和访存数据，传回ID段，寄存器堆读取的数据与传回到ID段的数据，依据是否要进行重定向来进行多路选择，然后再传入ID/EX流水部件，以实现重定向传输到EX段的数据。

### 重定向控制逻辑

重定向可控制模块包括两个部分，即重定向判断和重定向数据选择，重定向判断的实现与上一节中的数据冲突检测逻辑大致相同，只是在其基础上细分出普通数据相关和Load-Use数据相关。对与普通的数据相关，可通过数据重定向解决，而对于Load-Use类型的数据相关，必须得插入一个气泡消除相关。

至于重定向的数据选择，有四种情况：ID段的rA源操作数与EX段的目的操作数相同时，将EX段的数据重定向到RF\_A的输入；ID段的rB源操作数与EX段的目的操作数相同时，将EX段的数据重定向到RF\_B的输入；ID段的rA源操作数与MEM段的目的操作数相同时，将MEM段的数据重定向到RF\_A的输入；ID段的rB源操作数与MEM段的目的操作数相同时，将MEM段的数据重定向到RF\_B的输入。

重定向控制模块的引脚定义及说明如表 2.8所示。

表 2.8 重定向控制模块引脚定义

| 引脚 | 输入/输出 | 位宽 | 功能描述 |
| --- | --- | --- | --- |
| rA | 输入 | 5 | ID段指令的源操作数 |
| rB | 输入 | 5 | ID段指令的源操作数 |
| Ctrl | 输入 | 32 | ID段的控制信号，用于解析指令类型 |
| EX\_OP | 输入 | 6 | EX段指令的操作码，用于判断Load-Use情况 |
| EX\_rW | 输入 | 5 | EX段指令的目的操作数 |
| EX\_WE | 输入 | 1 | EX段指令的寄存器写使能端控制信号 |
| MEM\_rW | 输入 | 5 | MEM段指令的目的操作数 |
| MEM\_WE | 输入 | 1 | MEM段指令的寄存器写使能端控制信号 |
| EX\_Data | 输入 | 32 | EX段传回的数据 |
| MEM\_Data | 输入 | 32 | MEM段传回的数据 |
| rd\_RF\_A | 输出 | 32 | 要重定向到RF\_A的数据 |
| r\_RF\_A | 输出 | 1 | 重定向RF\_A的控制端，为1有效 |
| rd\_RF\_B | 输出 | 32 | 要重定向到RF\_B的数据 |
| r\_RF\_B | 输出 | 1 | 重定向RF\_B的控制端，为1有效 |
| Load-Use | 输出 | 1 | Load-Use情况，控制插入气泡 |

# 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。Halt为停机信号，将此控制信号通过非门取反之后和时钟相与，当需要进行停机时，Halt控制信号为1，经过非门之后为0，与时钟信号相与，屏蔽时钟信号，使整个电路停机。如图 3.1所示。

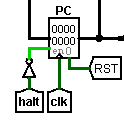


图 . 程序计数器（PC）

1. FPGA实现：

程序计数器PC的Verilog代码如下：

reg [31:0]PC;

initial PC = 0;

always @(posedge clk)

if(reset) PC <= 0;

else if(!halt) PC <= PC\_in;

1. 运算器（ALU）
2. Logism实现：

ALU的电路设计图如图 3.2所示，图中的隧道各表示一个引脚，各引脚的定义和功能描述见表 2.2 ALU引脚与功能描述，操作数传入后，就传到各运算部件计算所有运算的结果，但最终结果通过操作码配合多路选择器进行输出。

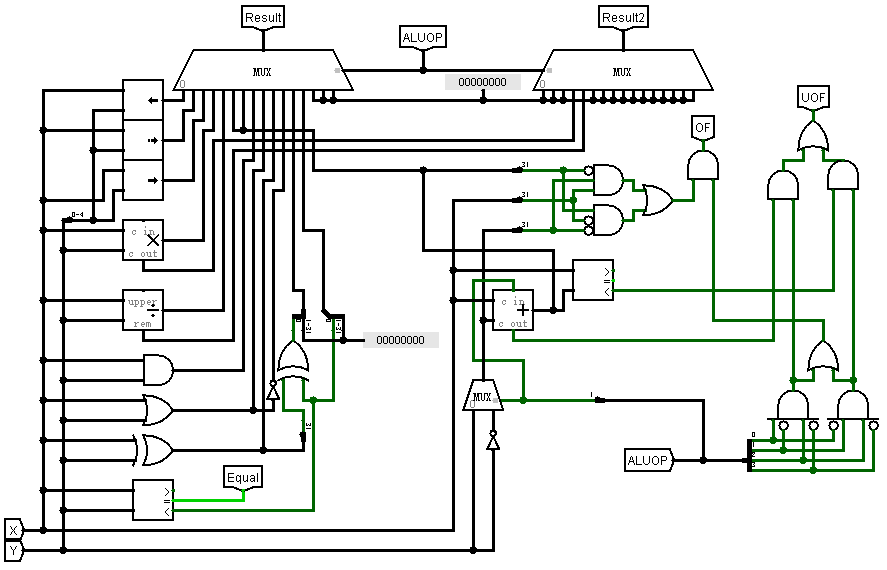


图 . 运算器原理图（ALU）

1. FPGA实现：

各种运算均可直接通过Verilog语言的运算符直接实现，最后的输出则是实例化两个16路选择器，根据ALUOP的值进行选择，其中16路选择器可用case语句简单实现，具体实现见16路选择器的Verilog代码。

运算器ALU的Verilog代码如下：

module ALU( S,

X,

Y,

Result,

Result2,

Equal,

Overflow,

UOF);

input [3:0]S;

input signed [31:0]X;

input signed [31:0]Y;

output [31:0]Result;

output [31:0]Result2;

output Equal;

output Overflow;

output UOF;

wire[31:0] const\_32bit\_0;

wire[30:0] const\_31bit\_0;

assign const\_32bit\_0 = 32'd0;

assign const\_31bit\_0 = 31'd0;

wire[31:0] BUF\_logical\_left;

wire[31:0] BUF\_arithmetic\_right;

wire[31:0] BUF\_logical\_right;

wire[31:0] BUF\_product\_lower;

wire[31:0] BUF\_product\_upper;

wire[31:0] BUF\_quotient;

wire[31:0] BUF\_remainder;

wire[31:0] BUF\_add\_result;

wire[31:0] BUF\_sub\_result;

wire[31:0] BUF\_and\_32\_bit;

wire[31:0] BUF\_or\_32\_bit;

wire[31:0] BUF\_xor\_32\_bit;

wire[31:0] BUF\_nor\_32\_bit;

wire[31:0] BUF\_signed\_comparator;

wire[31:0] BUF\_unsigned\_comparator;

Multiplexer\_16 #(.NrOfBits(32))

MUX\_Result (.Enable(1'b1),

.MuxIn\_0(BUF\_logical\_left),

.MuxIn\_1(BUF\_arithmetic\_right),

.MuxIn\_2(BUF\_logical\_right),

.MuxIn\_3(BUF\_product\_lower),

.MuxIn\_4(BUF\_quotient),

.MuxIn\_5(BUF\_add\_result),

.MuxIn\_6(BUF\_sub\_result),

.MuxIn\_7(BUF\_and\_32\_bit),

.MuxIn\_8(BUF\_or\_32\_bit),

.MuxIn\_9(BUF\_xor\_32\_bit),

.MuxIn\_10(BUF\_nor\_32\_bit),

.MuxIn\_11(BUF\_signed\_comparator),

.MuxIn\_12(BUF\_unsigned\_comparator),

.MuxIn\_13(const\_32bit\_0),

.MuxIn\_14(const\_32bit\_0),

.MuxIn\_15(const\_32bit\_0),

.MuxOut(Result),

.Sel(S));

Multiplexer\_16 #(.NrOfBits(32))

MUX\_Result2 (.Enable(1'b1),

.MuxIn\_0(const\_32bit\_0),

.MuxIn\_1(const\_32bit\_0),

.MuxIn\_2(const\_32bit\_0),

.MuxIn\_3(BUF\_product\_upper),

.MuxIn\_4(BUF\_remainder),

.MuxIn\_5(const\_32bit\_0),

.MuxIn\_6(const\_32bit\_0),

.MuxIn\_7(const\_32bit\_0),

.MuxIn\_8(const\_32bit\_0),

.MuxIn\_9(const\_32bit\_0),

.MuxIn\_10(const\_32bit\_0),

.MuxIn\_11(const\_32bit\_0),

.MuxIn\_12(const\_32bit\_0),

.MuxIn\_13(const\_32bit\_0),

.MuxIn\_14(const\_32bit\_0),

.MuxIn\_15(const\_32bit\_0),

.MuxOut(Result2),

.Sel(S));

assign BUF\_logical\_left = X << Y[4:0];

assign BUF\_arithmetic\_right = X >>> Y[4:0];

assign BUF\_logical\_right = X >> Y[4:0];

assign {BUF\_product\_upper,BUF\_product\_lower} = X \* Y;

assign BUF\_quotient = X / Y;

assign BUF\_remainder = X % Y;

assign BUF\_add\_result = X + Y;

assign BUF\_sub\_result = X - Y;

assign BUF\_and\_32\_bit = X & Y;

assign BUF\_or\_32\_bit = X | Y;

assign BUF\_xor\_32\_bit = X ^ Y;

assign BUF\_nor\_32\_bit = ~(X | Y);

assign BUF\_signed\_comparator = ($signed(X) < $signed(Y));

assign BUF\_unsigned\_comparator = ($unsigned(X) < $unsigned(Y));

assign Equal = (X == Y);

assign Overflow = 0;

assign UOF = 0;

endmodule

16路多路选择器的Verilog代码如下：

module Multiplexer\_16( Enable,

MuxIn\_0,

MuxIn\_1,

MuxIn\_10,

MuxIn\_11,

MuxIn\_12,

MuxIn\_13,

MuxIn\_14,

MuxIn\_15,

MuxIn\_2,

MuxIn\_3,

MuxIn\_4,

MuxIn\_5,

MuxIn\_6,

MuxIn\_7,

MuxIn\_8,

MuxIn\_9,

Sel,

MuxOut);

parameter NrOfBits = 1;

input Enable;

input[NrOfBits-1:0] MuxIn\_0;

input[NrOfBits-1:0] MuxIn\_1;

input[NrOfBits-1:0] MuxIn\_10;

input[NrOfBits-1:0] MuxIn\_11;

input[NrOfBits-1:0] MuxIn\_12;

input[NrOfBits-1:0] MuxIn\_13;

input[NrOfBits-1:0] MuxIn\_14;

input[NrOfBits-1:0] MuxIn\_15;

input[NrOfBits-1:0] MuxIn\_2;

input[NrOfBits-1:0] MuxIn\_3;

input[NrOfBits-1:0] MuxIn\_4;

input[NrOfBits-1:0] MuxIn\_5;

input[NrOfBits-1:0] MuxIn\_6;

input[NrOfBits-1:0] MuxIn\_7;

input[NrOfBits-1:0] MuxIn\_8;

input[NrOfBits-1:0] MuxIn\_9;

input[3:0] Sel;

output[NrOfBits-1:0] MuxOut;

reg[NrOfBits-1:0] s\_selected\_vector;

assign MuxOut = s\_selected\_vector;

always @(\*)

begin

if (~Enable) s\_selected\_vector <= 0;

else case (Sel)

4'b0000: s\_selected\_vector <= MuxIn\_0;

4'b0001: s\_selected\_vector <= MuxIn\_1;

4'b0010: s\_selected\_vector <= MuxIn\_2;

4'b0011: s\_selected\_vector <= MuxIn\_3;

4'b0100: s\_selected\_vector <= MuxIn\_4;

4'b0101: s\_selected\_vector <= MuxIn\_5;

4'b0110: s\_selected\_vector <= MuxIn\_6;

4'b0111: s\_selected\_vector <= MuxIn\_7;

4'b1000: s\_selected\_vector <= MuxIn\_8;

4'b1001: s\_selected\_vector <= MuxIn\_9;

4'b1010: s\_selected\_vector <= MuxIn\_10;

4'b1011: s\_selected\_vector <= MuxIn\_11;

4'b1100: s\_selected\_vector <= MuxIn\_12;

4'b1101: s\_selected\_vector <= MuxIn\_13;

4'b1110: s\_selected\_vector <= MuxIn\_14;

default: s\_selected\_vector <= MuxIn\_15;

endcase

end

endmodule

### 数据通路的实现

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如表 3.1所示，再在Logisim和Vivado上进行数据通路的实现。

1. Logism实现：

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**错误!未找到引用源。**所示。

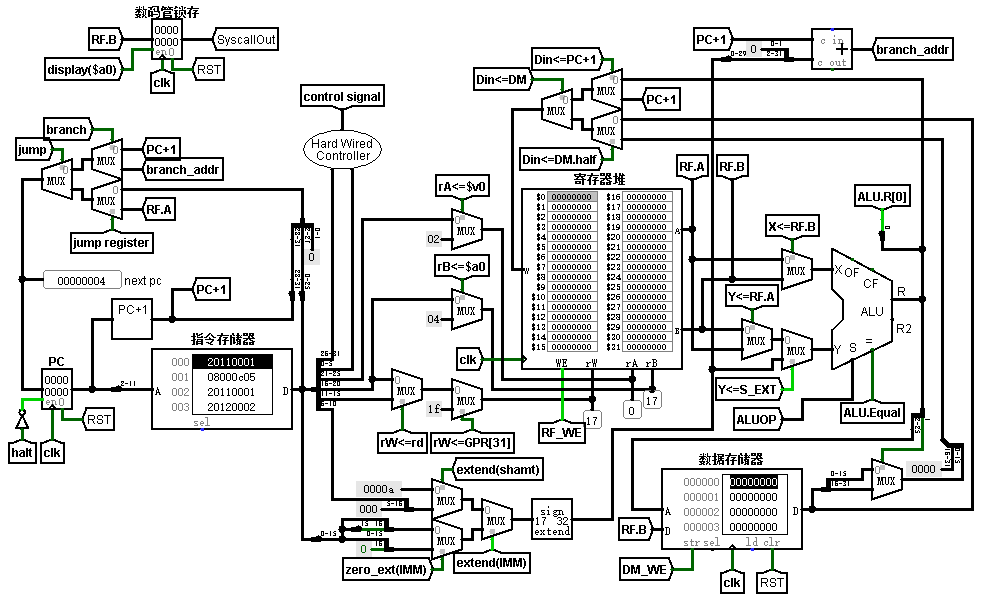


图 . 单周期CPU数据通路（Logism）

1. FPGA实现：

根据**错误!未找到引用源。**所示的原理图，实例化已实现的主要模块，并进行数据通路的连接。

在Vivado中使用Verilog语言搭建的数据通路的原理图如图 3.4所示。

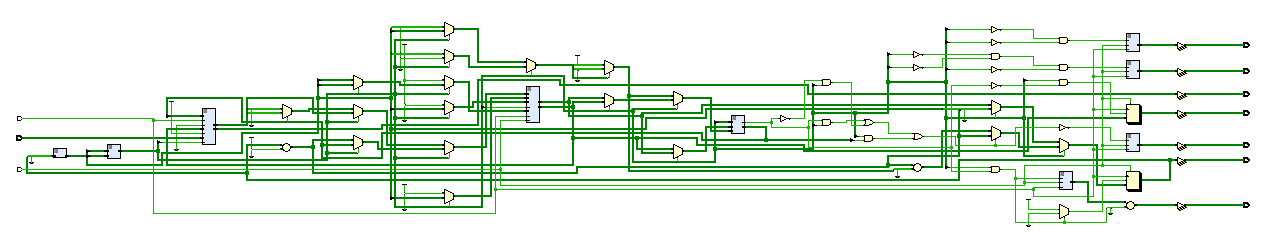


图 3.4 单周期CPU数据通路（FPGA）

|  |
| --- |
| 表 . 数据通路表 |

### 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，在Logism和Vivado上进行控制器的具体实现。其中，完整的控制信号表见表 3.2。

1. Logisim实现

用比较器比较OP和Funct字段的值以解析出当前的指令，然后对控制信号输出的每一位，使其为需要该信号为真的指令相或的结果，如 控制器原理图（Logisim）图 3.5所示。

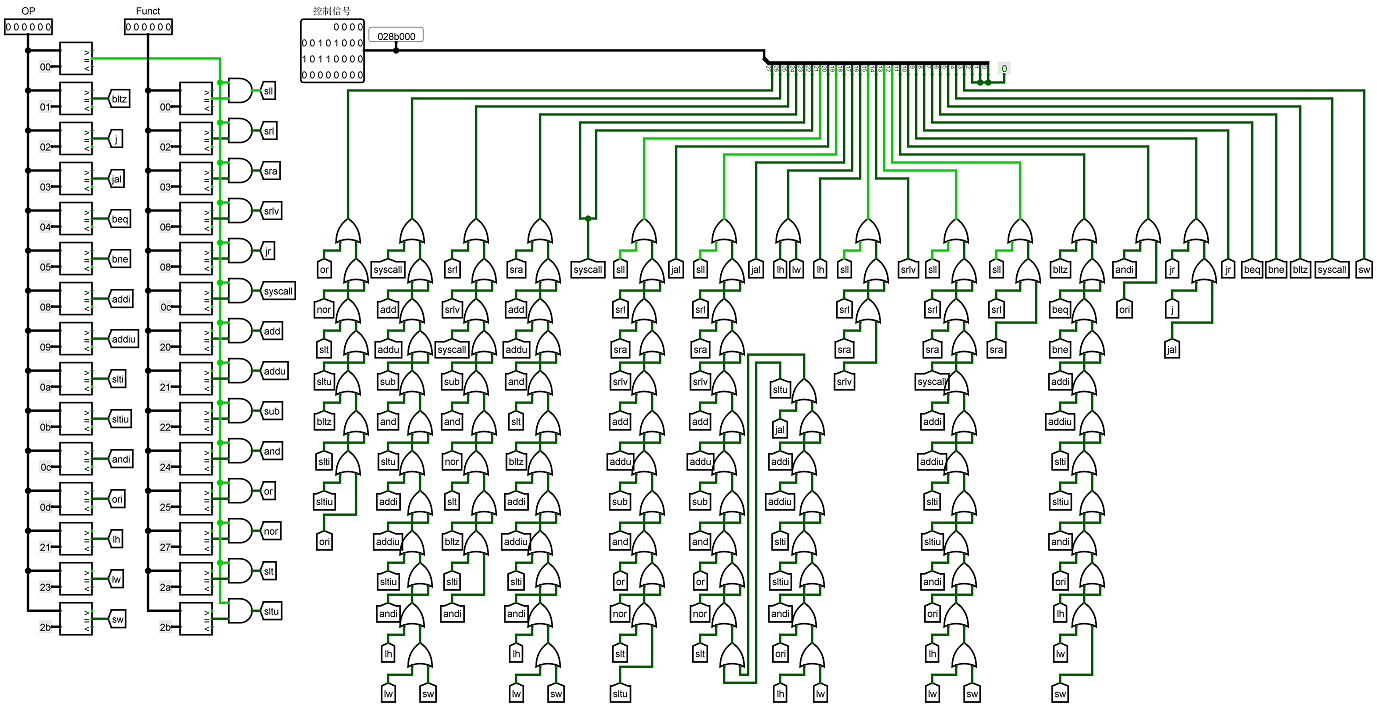


图 . 控制器原理图（Logisim）

1. FPGA实现

根据在Logism中的原理图，可用简单的逻辑运算得到每一位控制信号的输出，在Vivado中使用Verilog语言构成的主控制器原理图如图 3.6所示。

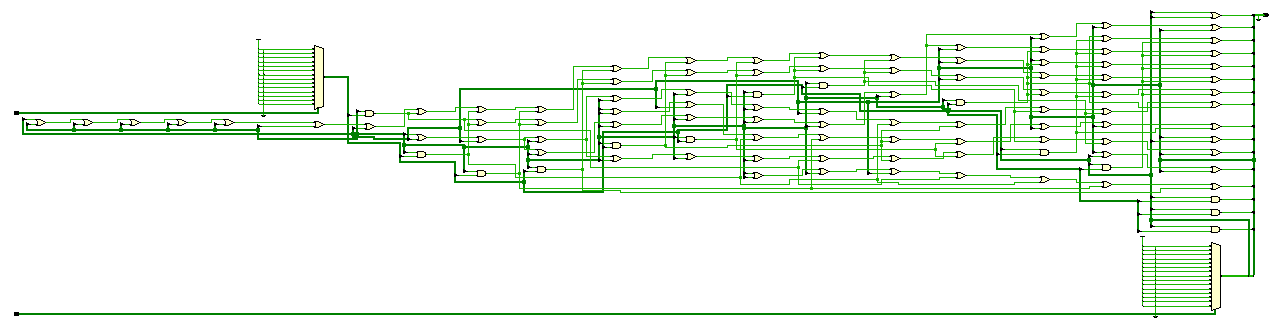


图 . 控制器原理图（FPGA）

|  |
| --- |
| 表 . 空制信号表 |

### 控制器的Verilog代码如下：

module controller(

input [5:0] OP,

input [5:0] Funct,

output [27:0] control\_signal

);

wire op0,bltz,j,jal,beq,bne,addi,addiu,slti,sltiu,andi,ori,lh,lw,sw;

wire f1,f2,f3,f4,f5,f6,f7,f8,f9,f10,f11,f12,f13,f14;

wire sll,srl,sra,srlv,jr,syscall,add,addu,sub,c\_and,c\_or,c\_nor,slt,sltu;

assign op0 = (OP == 6'h00);

assign bltz = (OP == 6'h01);

assign j = (OP == 6'h02);

assign jal = (OP == 6'h03);

assign beq = (OP == 6'h04);

assign bne = (OP == 6'h05);

assign addi = (OP == 6'h08);

assign addiu = (OP == 6'h09);

assign slti = (OP == 6'h0a);

assign sltiu = (OP == 6'h0b);

assign andi = (OP == 6'h0c);

assign ori = (OP == 6'h0d);

assign lh = (OP == 6'h21);

assign lw = (OP == 6'h23);

assign sw = (OP == 6'h2b);

assign f1 = (Funct == 6'h00);

assign f2 = (Funct == 6'h02);

assign f3 = (Funct == 6'h03);

assign f4 = (Funct == 6'h06);

assign f5 = (Funct == 6'h08);

assign f6 = (Funct == 6'h0c);

assign f7 = (Funct == 6'h20);

assign f8 = (Funct == 6'h21);

assign f9 = (Funct == 6'h22);

assign f10 = (Funct == 6'h24);

assign f11 = (Funct == 6'h25);

assign f12 = (Funct == 6'h27);

assign f13 = (Funct == 6'h2a);

assign f14 = (Funct == 6'h2b);

assign sll = op0&f1;

assign srl = op0&f2;

assign sra = op0&f3;

assign srlv = op0&f4;

assign jr = op0&f5;

assign syscall = op0&f6;

assign add = op0&f7;

assign addu = op0&f8;

assign sub = op0&f9;

assign c\_and = op0&f10;

assign c\_or = op0&f11;

assign c\_nor = op0&f12;

assign slt = op0&f13;

assign sltu = op0&f14;

assign control\_signal[27] = ori|sltiu|slti|bltz|sltu|slt|c\_nor|c\_or;

assign control\_signal[26] = lw|sw|lh|andi|sltiu|addiu|addi|sltu|c\_and|sub|addu|add|syscall;

assign control\_signal[25] = andi|slti|bltz|slt|c\_nor|c\_and|sub|syscall|srlv|srl;

assign control\_signal[24] = lw|sw|lh|andi|slti|addiu|addi|bltz|slt|c\_and|addu|add|sra;

assign control\_signal[23] = syscall;

assign control\_signal[22] = syscall;

assign control\_signal[21] = sltu|slt|c\_nor|c\_or|c\_and|sub|addu|add|srlv|sra|srl|sll;

assign control\_signal[20] = jal;

assign control\_signal[19] = lh|lw|ori|andi|sltiu|slti|addiu|addi|jal|sltu|slt|c\_nor|c\_or|c\_and|sub|addu|add|srlv|sra|srl|sll;

assign control\_signal[18] = jal;

assign control\_signal[17] = lh|lw;

assign control\_signal[16] = lh;

assign control\_signal[15] = srlv|sra|srl|sll;

assign control\_signal[14] = srlv;

assign control\_signal[13] = lw|sw|lh|ori|andi|sltiu|slti|addiu|addi|syscall|sra|srl|sll;

assign control\_signal[12] = sra|srl|sll;

assign control\_signal[11] = sw|lw|lh|ori|andi|sltiu|slti|addiu|addi|bne|beq|bltz;

assign control\_signal[10] = ori|andi;

assign control\_signal[9] = jal|j|jr;

assign control\_signal[8] = jr;

assign control\_signal[7] = beq;

assign control\_signal[6] = bne;

assign control\_signal[5] = bltz;

assign control\_signal[4] = syscall;

assign control\_signal[3] = sw;

assign control\_signal[2] = 1'b0;

assign control\_signal[1] = 1'b0;

assign control\_signal[0] = 1'b0;

endmodule

## 理想流水CPU实现

依据总体设计方案中理想流水线CPU设计的那一小节，在 单周期CPU数据通路（Logism）的基础上，将数据通路改造成流水线形式，在Logisim平台上实现的理想流水线电路图如图 3.7所示。

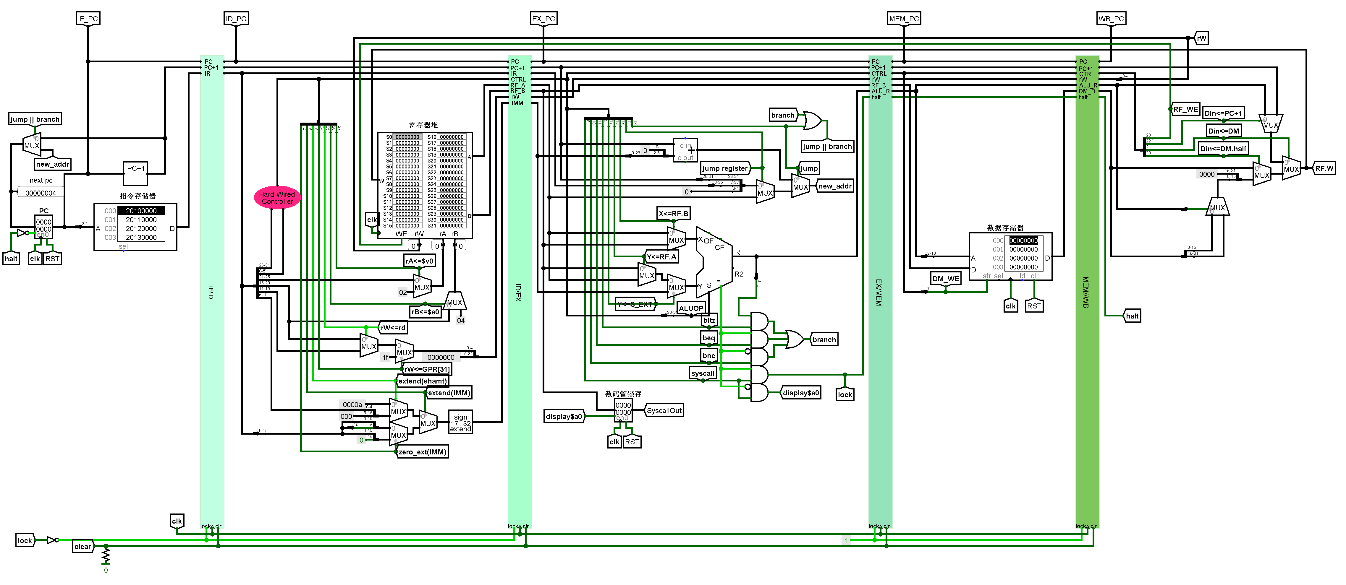


图 . 理想流水线CPU电路图

## 气泡式流水线实现

### 数据相关检测逻辑的实现

依据表 2.7中的引脚定义和2.4.1小节的设计，在Logisim平台上实现的数据相关检测部件如图 3.8所示。其中隧道read\_rA表示ID段指令需要从rA指定的寄存器读取数据，隧道read\_rB表示ID段指令需要从rB指定的寄存器读取数据，隧道syscall表示ID段指令为系统调用指令，这几个隧道通过解析输入引脚ID\_OP和ID\_Funct来得到。

### 插入气泡和流水暂停逻辑的实现

按照2.4.2和2.4.1小节的设计，在Logisim平台上实现插入气泡逻辑和流水暂停逻辑，其电路图如图 3.9所示，当需要在EX段插入气泡时，IF/ID接口部件的lock端被置位0，而ID/EX接口部件的clr清零端被置位1；当需要处理分支冲突时，IF/ID接口部件和ID/EX接口部件的清零端都被置位1。

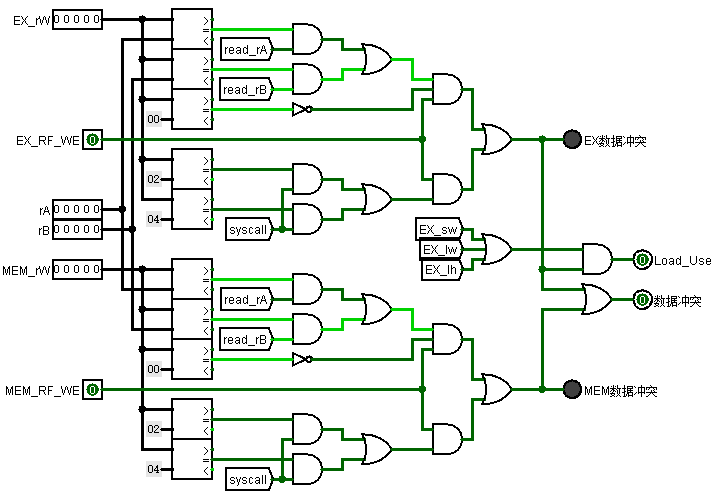


图 . 数据相关检测部件电路图

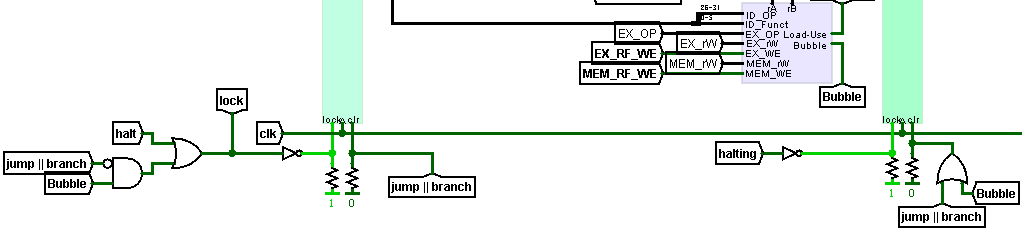


图 . 气泡插入和流水暂停实现

### 气泡流水功能的实现

将数据冲突检查部件和冲突处理逻辑加入流水线CPU电路中，即得到了气泡流水先的电路，如图 3.10所示。

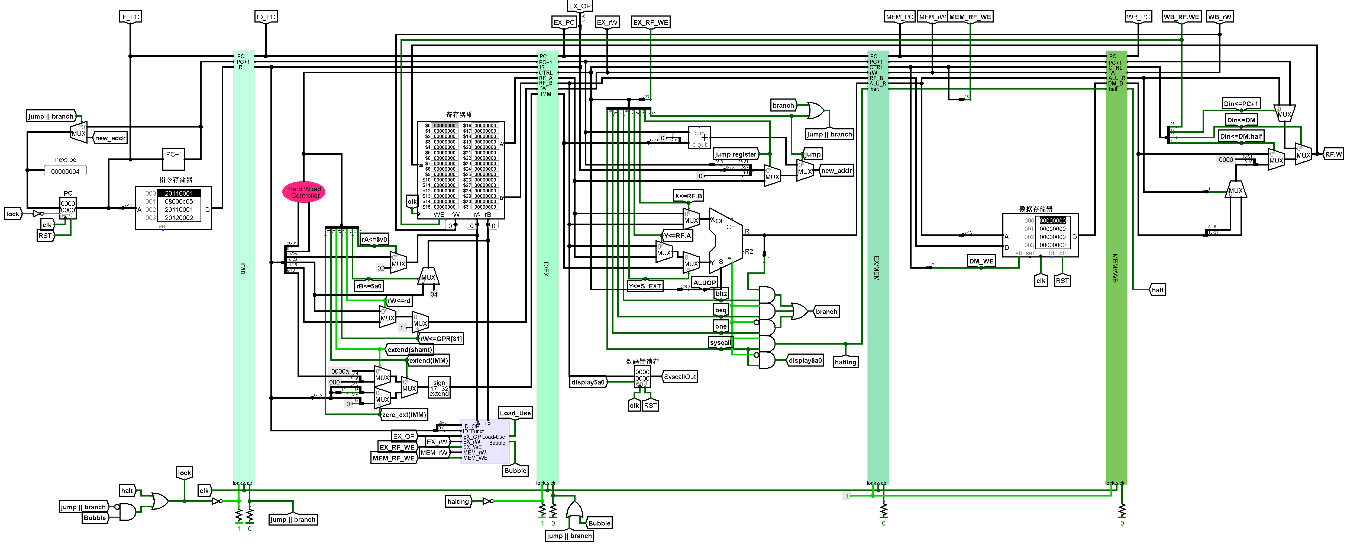


图 . 气泡流水线CPU电路图

## 重定向流水线实现

### 重定向数据通路的实现

依据2.5.1小节的设计，在Logisim平台上实现重定向数据通路如图 3.11所示。其中EX\_Data和MEM\_Data分别是EX段和MEM段传回的数据，这两个数据传入重定向控制模块，然后依据重定向的判断结果，决定是否要将RF\_A和RF\_B的输入结果重定向，并在重定向模块内决定要重定向的数据。

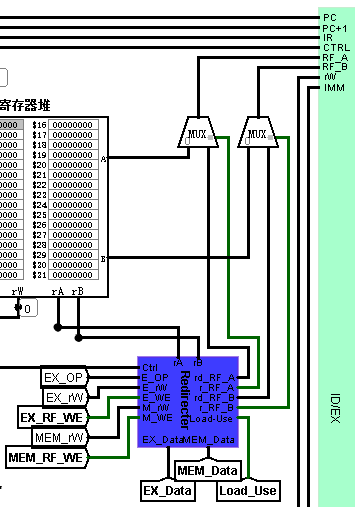


图 . 重定向数据通路电路图

### 重定向控制逻辑的实现

依据2.5.2小节的设计方案，在Logisim平台上实现重定向控制模块的电路图如图 3.12所示。

其中，根据传入的ID段控制信号判断当前指令是否要读取rA或rB指定的寄存器的数据，再判断源操作数与EX段和MEM段的目的操作数是否相同，来判断数据相关，特别的，根据EX\_OP判断当前处于EX段的指令是否为访存指令，当EX段的指令是访存指令，而且ID段指令与EX段指令存在数据相关是，即判断该情况为Load-Use相关。

重定向数据的选择则是根据ID段的指令与EX段的指令存在数据相关，还是与MEM段的指令存在数据相关来进行选择，见图 3.12的最右部分。

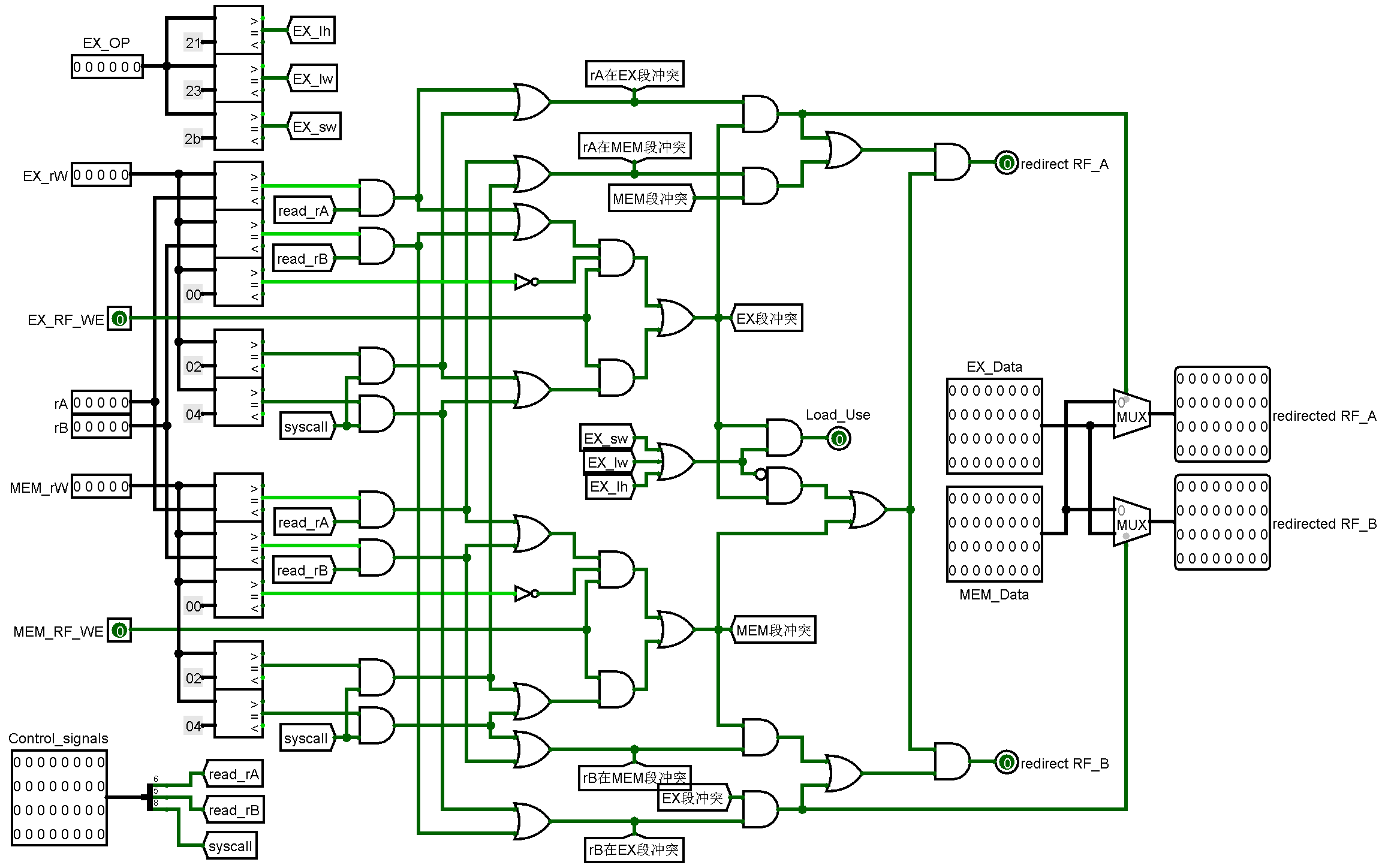


图 . 重定向控制模块电路图

### 重定向功能的实现

将气泡流水线CPU中的气泡插入控制模块改造为重定向数据通路和重定向控制模块，即得到了重定向流水线CPU的电路，如图 3.13所示。

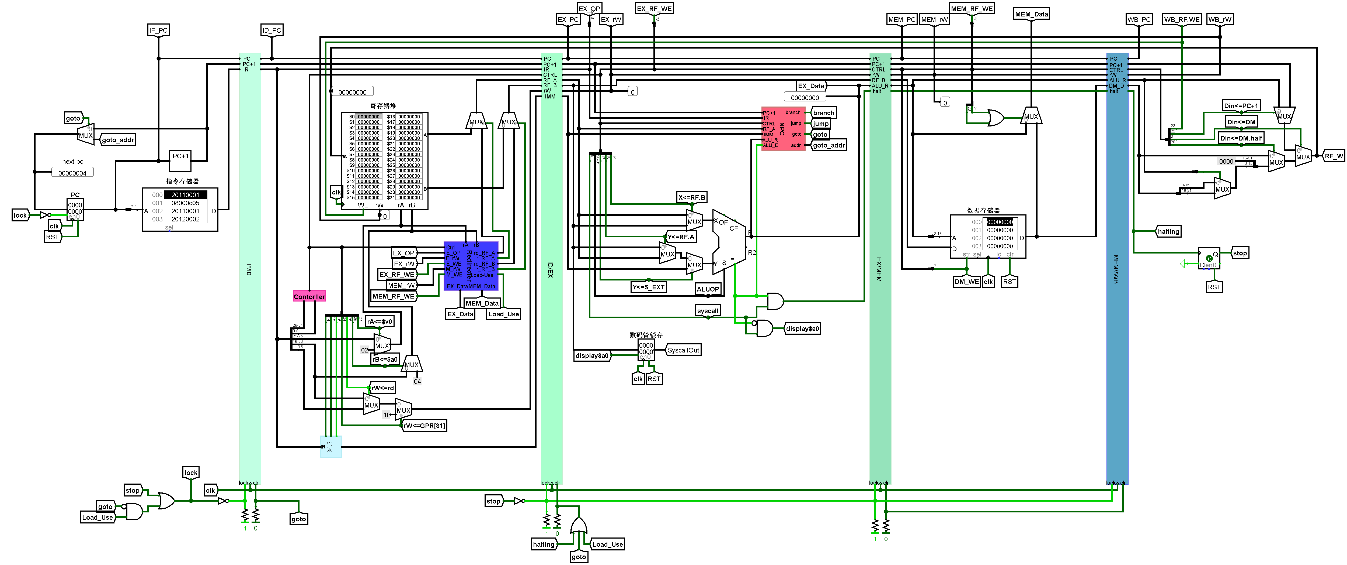


图 . 重定向流水线电路图

# 实验过程与调试

## 单周期CPU上板测试

### 测试程序和理论结果

测试程序为benchmark程序，将程序机器码加载入指令存储器IM后，进行CPU模块的功能仿真，理论结果是指令总周期数为1546，无条件分支跳转指令数为38，有条件分支指令数为338，有条件分支指令成功跳转次数为276，且最后系统调用输出的十六进制数为0x00000038，最后PC停止在0x000032dc地址处。

### 测试结果与分析

在Vivado平台上对CPU模块进行功能测试，行为仿真的波形图如图 4.1所示。从图中可以看到，CPU停机后，指令总周期数v\_total\_cycles为1546，无条件分支跳转指令数为38，有条件分支指令数为338，有条件v\_branch\_cycles分支指令成功跳转次数v\_branch\_success\_cycles为276，且最后系统调用输出v\_Syscall\_out为0x00000038，最后v\_PC停止在0x000032dc地址处。测试结果与理论结果完全相符，因此得出单周期CPU的功能正确。

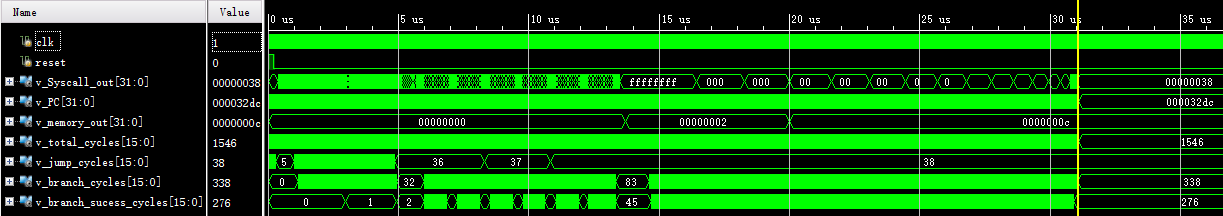


图 . 单周期CPU功能测试结果

## 理想流水线CPU功能测试

### 测试程序和理论结果

测试程序为理想流水线CPU测试程序，程序中所有指令均无相关性，一共17条指令，执行完后理论周期数为21，且内存中的前四个字数据应分别为0x00000000、0x00000001、0x00000002和0x00000003。

### 测试结果与分析

在Logisim平台上进行仿真测试，测试程序运行结果如图 4.2和图 4.3所示，从图中可以看到仿真结果与理论结果相符，即理想流水线CPU功能正确。

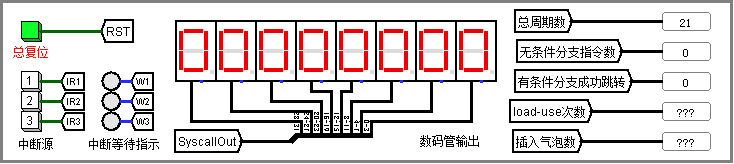


图 . 理想流水线CPU测试结果

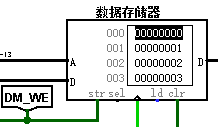


图 . 理想流水程序运行结束后DM的状态

## 气泡流水线CPU功能测试

### 测试程序和理论结果

测试程序为benchmark程序，将程序机器码加载入电路中的指令存储器IM后，启动时钟进行CPU模块的功能仿真，理论结果是指令总周期数为3624或3623，无条件分支跳转指令数为38，有条件分支指令成功跳转次数为276，插入的气泡数为1446，且最后系统调用输出的十六进制数为0x00000038，且最后数据存储器DM中前16个字的内容依次为十进制的14到-1的降序。

### 测试结果与分析

在Logisim平台上进行气泡流水线CPU的功能测试，仿真结果如图 4.4和图 4.5所示，从图中各可以看出仿真结果与理论结果相符。其中，总周期是为3623是因为停机指令在EX段就已执行完毕，从WB段流出时没有对该指令计数。

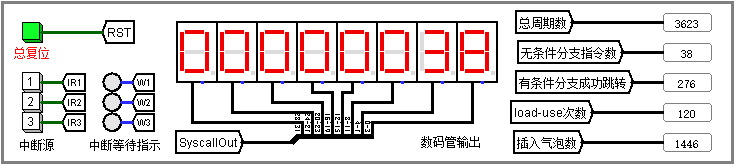


图 . 气泡流水线CPU仿真结果



图 . 气泡流水线仿真后DM中的内容

## 重定向流水线CPU功能测试

### 测试程序和理论结果

测试程序为benchmark程序，将程序机器码加载入电路中的指令存储器IM后，启动时钟进行CPU模块的功能仿真，理论结果是指令总周期数为2298，无条件分支跳转指令数为38，有条件分支指令成功跳转次数为276，插入的气泡数和Load-Use次数都为120，且最后系统调用输出的十六进制数为0x00000038，且最后数据存储器DM中前16个字的内容依次为十进制的14到-1的降序。

### 测试结果与分析

在Logisim平台上进行气泡流水线CPU的功能测试，仿真结果如图 4.6和图 4.7所示，从图中各可以看出仿真结果与理论结果相符。

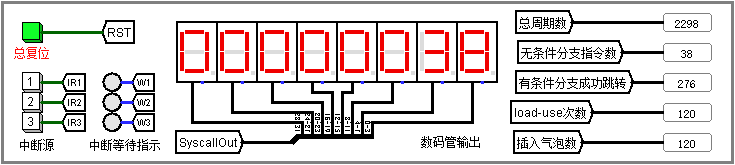


图 . 重定向流水线CPU仿真结果



图 . 重定向流水线仿真后DM中的内容

## 性能分析

对于同样的benchmark测试程序，单周期CPU的总周期数虽然最少，但是其单时钟周期长度是以速度最慢的指令需要的时钟周期长度来设计的，所以它是性能最差的一种方案设计；

气泡流水线CPU的总周期数为3623，虽然对于单周期CPU来说，性能有了不少的提升，但是其有1446个周期是插入的气泡，有效周期只占总周期的60%，效率比较低；

重定向流水线CPU的总周期数为2298，比气泡流水线CPU的总周期数减少了大概三分之一，插入的气泡数为120个，有效周期占总周期的95%，性能和效率都比气泡流水想高很多。

## 实验进度

表 . 课程设计进度表

| 时间 | 进度 |
| --- | --- |
| 第一天 | 完成了ALU的设计 |
| 第二天 | 完成了数据通路和CPU模块 |
| 第三天 | 完成了单周期CPU上板子测试，着手写理想流水线cpu |
| 第四天 | 完成了理想流水线，了解了气泡流水的原理 |
| 第五天 | 完成了气泡流水线，开始做重定向流水线设计 |
| 第六天 | 完成了重定向流水线，开始做动态分支预测设计 |
| 第七天 | 完成了动态分支预测逻辑设计和LRU算法的设计 |
| 第八天 | 完成了带动态分支预测功能的流水线设计 |
| 第九天 | 完成了数据重定向的Verilog代码并成功烧入FPGA板内，测试成功 |
| 第十天 | 完成了单周期CPU多级中断机制的实现 |

# 设计总结与心得

## 课设总结

### 完成的设计方案

本次课程设计主要完成了一下方案的设计：

1. 单周期CPU的设计
2. 理想流水线CPU的设计
3. 气泡流水线的设计
4. 重定向流水线CPU的设计

### 实现的工作

1. 单周期CPU上板实验
2. 在Logisim平台上实现了理想流水线CPU的设计
3. 在Logisim平台上实现了气泡流水线CPU的设计
4. 在Logisim平台上实现了重定向流水线CPU的设计

## 课设心得

从大二开始就听说了组成原理的课程设计非常难，这两个星期一做，觉得它真的是迄今为止所有实验以及课程设计中难度最大的一门，也是设计内容最多的一门课设，感觉如果只用两个星期中课堂内的时间，根本完成不了，毕竟我可是通宵了两天才将除流水中断之外的内容都做完。

不过，不得不承认，这门课设所用到了知识和从中另外学到的知识比做过的其他课程设计都要多的多，上学期老师在组成原理课上没有讲过流水线，想不到通过做课程设计的过程中，我就掌握了经典5段MIPS流水线CPU的设计方法，并且掌握了解决数据相关、结构相关和分支相关的简单方法，以及数据重定向流水线CPU的的原理与设计。在扩展内容的完成过程中，我在FPGA开发板上实现了重定向流水线CPU，还了解并实现了流水线CPU的动态分支预测机制，这些扩展内容都非常有趣，而且都有一定的难度，做完之后很是有成就感。

第一个任务的单周期上板实验比较简单，但是由于将Logisim电路用Verilog代码描述很是繁琐，在加上对Vivado的使用还不是很熟练，所以还是花了不少时间，期间遇到过不少坑，觉得写代码时，端口的命名规范很重要，不然在连接数据通路是，很容易连错，导致找bug要花很多时间。

第二个任务理想流水线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，因为这些方法老师都没讲过，书上也没有，还好课程设计指南上讲的比较清楚易懂，学习了一段时间后就可以按照上面的步骤和思想设计电路了。

至于对本次课程设计的建议，我觉得第一个任务通过小组合作完成这个机制比较好，不仅能减轻每个人的工作负担，而且锻炼了我们的团队合作能力，建议一直保留这个机制。另外，我建议实验内容减少一点，或者每一项扩展内容的分数加一点，虽然说扩展的内容可以选择不做，但大家都想要高一点的分数嘛。而且，我觉得完成的内容太多，报告好难写呀，模版给的也不是很详细，不知道每个任务具体什么要写在报告里，所有内容都写的话太费时间了，而且格式控制也汉繁琐，所以建议以后的课程设计报告能给一个更加详细模版，或者说清楚对每一个任务，要在报告上说清楚的内容。

总之，组成原理课程设计绝对是大学里最难忘的一次经历。

# 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. 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.
3. 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.
4. 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:** |