***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： | 物联网工程 |
| 班 级： | IOT201401 |
| 学 号： | U201414952 |
| 姓 名： | 刘剑一祥 |
| 电 话： | 13297913956 |
| 邮 件： | [295671947@qq.com](mailto:295671947@qq.com) |
| 完成日期： | 2012-03-22 周三下午 |

目 录

[1 课程设计概述 1](#_Toc478377522)

[1.1 课设目的 1](#_Toc478377523)

[1.2 设计任务 1](#_Toc478377524)

[1.3 设计要求 1](#_Toc478377525)

[1.4 技术指标 2](#_Toc478377526)

[2 总体方案设计 4](#_Toc478377527)

[2.1 单周期CPU设计 4](#_Toc478377528)

[2.2 中断机制设计 9](#_Toc478377529)

[2.3 流水CPU设计 11](#_Toc478377530)

[2.4 数据转发流水线设计 11](#_Toc478377531)

[3 详细设计与实现 13](#_Toc478377532)

[3.1 单周期CPU 实现 13](#_Toc478377533)

[3.2 中断机制实现 27](#_Toc478377534)

[3.3 流水CPU实现 30](#_Toc478377535)

[3.4 数据转发流水线实现 34](#_Toc478377536)

[4 实验过程与调试 41](#_Toc478377537)

[4.1 测试用例和功能测试 41](#_Toc478377538)

[4.2 性能分析 47](#_Toc478377539)

[4.3 主要故障与调试 47](#_Toc478377540)

[4.4 实验进度 52](#_Toc478377541)

[5 设计总结与心得 53](#_Toc478377542)

[5.1 课设总结 53](#_Toc478377543)

[5.2 课设心得 53](#_Toc478377544)

[参考文献 55](#_Toc478377545)

# 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**错误!未找到引用源。**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。

表格 1‑1 指令集

| **#** | **指令助记符** | **简单功能描述** | **备注** |
| --- | --- | --- | --- |
| 1 | ADD | 加法 | 指令格式参考MIPS32指令集，最终功能以MARS模拟器为准。 |
| 2 | ADDI | 立即数加 |
| 3 | ADDIU | 无符号立即数加 |
| 4 | ADDU | 无符号数加 |
| 5 | AND | 与 |
| 6 | ANDI | 立即数与 |
| 7 | SLL | 逻辑左移 |
| 8 | SRA | 算数右移 |
| 9 | SRL | 逻辑右移 |
| 10 | SUb | 减 |
| 11 | OR | 或 |
| 12 | ORI | 立即数或 |
| 13 | NOR | 或非 |
| 14 | LW | 加载字 |
| 15 | SW | 存字 |
| 16 | BEQ | 相等跳转 |
| 17 | BNE | 不相等跳转 |
| 18 | SLT | 小于置数 |
| 19 | STI | 小于立即数置数 |
| 20 | SLTU | 小于无符号数置数 |
| 21 | J | 无条件转移 |
| 22 | JAL | 转移并链接 |
| 23 | JR | 转移到指定寄存器 | If $v0==10 halt(停机指令)  else 数码管显示$a0值 |
| 24 | SYSCALL | 系统调用 |
| 25 | MFC0 | 访问CP0 | 中断相关，可简化，选做 |
| 26 | MTC0 | 访问CP0 | 中断相关，可简化，选做 |
| 27 | ERET | 中断返回 | 异常返回，选做 |
| 28 | LH | 加载半字 |  |
| 29 | SLLV | 可变逻辑左移 |  |
| 30 | LUI | 立即数加载至高位 |  |
| 31 | BLEZ | 小于等于 0 转移 |  |

# 总体方案设计

## 单周期CPU设计

单周期CPU总体上分为地址跳转、指令读取、控制单元、运算、存储等几个部分；有三个总线，即地址总线，控制总线、数据总线，分别用于地址的传输、控制信号灯的传输以及数据的传输。

整个过程先使用logism设计出对应的电路，再使用verilog语言在vivado上写出相关电路的代码部分，生成FPGA格式

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

运算器部分，具体设计思路如下

#### 程序计数器PC

程序计数器PC使用寄存器保存，每个时钟周期更新数值，由指令决定是跳转到对应地址还是直接+4执行下一条指令。

#### 指令存储器IM

指令存储器采用ROM来存储所需执行的指令，根据PC的2-11位作为地址来选择对应时钟周期所执行的指令，每个时钟周期执行一条指令。

#### 运算器

表格 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 |

#### 寄存器堆RF

寄存器堆中含有32个32位寄存器，其中零号寄存器的值恒为零，整体上通过外部输入的R1#和R2选择对应的寄存器进行读操作，RW选择对应的寄存器进行写操作。

R型：

表格 2‑2 R型指令

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | OP | Funct |
| Add | add $rd, $rs, $rt | 000000 | 100000 |
| Add Unsigned | addu $rd, $rs, $rt | 000000 | 100001 |
| And | and $rd, $rs, $rt | 000000 | 100100 |
| Shift Left Logical | sll $rd, $rt, shamt | 000000 | 000000 |
| Shift Right Arithmetic | sra $rd, $rt, shamt | 000000 | 000011 |
| Shift Right Logical | srl $rd, $rt, shamt | 000000 | 000010 |
| Sub | sub $rd, $rs, $rt | 000000 | 100010 |
| Or | or $rd, $rs, $rt | 000000 | 100101 |
| Nor | nor $rd, $rs, $rt | 000000 | 100111 |
| Set Less Than | slt $rd, $rs, $rt | 000000 | 101010 |
| Set Less Than Unsigned | sltu $rd, $rs, $rt | 000000 | 101011 |
| Jump Register | jr $rs | 000000 | 001000 |
| syscall（display or exit） | Syscall | 000000 | 001100 |
| Shift left logical Variable | Sllv $rd, $rs, $rt | 000000 | 000100 |

I型：

表格 2‑3 I型指令

|  |  |  |
| --- | --- | --- |
|  |  | OP |
| Add Immediate | addi $rt, $rs, immediate | 001000 |
| Add Immediate Unsigned | addiu $rt, $rs, immediate | 001001 |
| And Immediate | andi $rt, $rs, immediate | 001100 |
| Or Immediate | ori $rt, $rs, immediate | 001101 |
| Load Word | lw $rt, offset($rs) | 100011 |
| Load Half Word | lh $rt, offset($rs) | 100001 |
| Store Word | sw $rt, offset($rs) | 101011 |
| Branch on Equal | beq $rs, $rt, label | 000100 |
| Branch on Not Equal | bne $rs, $rt, label | 000101 |
| Blez | Blez $rs, label | 000110 |
| Set Less Than Immediate | slti $rt, $rs, immediate | 001010 |
| Add Immediate | addi $rt, $rs, immediate | 001000 |
| Add Immediate Unsigned | addiu $rt, $rs, immediate | 001001 |
| And Immediate | andi $rt, $rs, immediate | 001100 |
| Or Immediate | ori $rt, $rs, immediate | 001101 |
| Lui | Lui $rt, immediate | 001111 |

J型：

表格 2‑4 J型指令

|  |  |  |
| --- | --- | --- |
|  |  | OP |
| Jump | j label | 000010 |
| Jump and Link | jal label | 000011 |

### 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格 2‑5 主控制器控制信号的作用说明。

表格 2‑5 主控制器控制信号的作用说明

| 控制信号 | 取值 | 说明 |
| --- | --- | --- |
| ALUsrc | 0 | ALU Y为R2 |
| 1 | ALU Y为zero Extend |
| 2 | ALU Y为sign Extend |
| 3 | ALU Y为R1 |
| Shift | 0 | ALU X为R1 |
| 1 | ALU X为R2 |
| MemtoReg | 0 | 选择ALU输出作为结果 |
| 1 | 选择RAM输出作为结果 |

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所示表格 2‑6。

表格 2‑6 主控制器控制信号框架

|  | RegDst | Branch | Jump | MemtoReg | MemRead | MemWrite | ALUSrc | ALUop | RegWrite |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|
| sllv | 1 | 0 | 0 | 0 | 0 | 0 | 11 | 0000 | 1 |
| xor | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 1001 | 1 |
| addi | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 00 | 1 |
| addu | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 00 | 1 |
| and | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 10 | 1 |
| sll | 1 | 0 | 0 | 0 | 0 | 0 | 01 | 10 | 1 |
| sra | 1 | 0 | 0 | 0 | 0 | 0 | 01 | 10 | 1 |
| srl | 1 | 0 | 0 | 0 | 0 | 0 | 01 | 10 | 1 |
| sub | 1 | 0 | 0 | 0 | 0 | 0 | 00 |  | 1 |
| or | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 10 | 1 |
| nor | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 10 | 1 |
| slt | 1 | 0 | 0 | 0 | 0 | 0 | 00 |  | 1 |
| sltu | 1 | 0 | 0 | 0 | 0 | 0 | 00 |  | 1 |
| jr | 0 | 0 | 0 |  | 0 | 0 |  |  | 0 |
| Syscall | 0 |  |  |  |  |  |  |  |  |
| addi | 0 | 0 | 0 | 0 | 0 | 0 | 10 |  | 1 |
| addiu | 0 | 0 | 0 | 0 | 0 | 0 | 10 |  | 1 |
| andi | 0 | 0 | 0 | 0 | 0 | 0 | 10 |  | 1 |
| ori | 0 | 0 | 0 | 0 | 0 | 0 | 10 |  | 1 |
| lw | 0 | 0 | 0 | 1 | 1 | 0 | 10 |  | 1 |
| lhu | 0 | 0 | 0 | 1 | 1 | 0 | 10 |  | 1 |
| sw | 0 | 0 | 0 | 1 | 0 | 1 | 10 |  | 0 |
| beq | 0 | 1 | 0 | 0 | 0 | 0 |  |  | 0 |
| bne | 0 | 1 | 0 | 0 | 0 | 0 |  |  | 0 |
| bltz | 0 | 1 | 0 | 0 | 0 | 0 |  |  | 0 |
| slti | 1 | 0 | 0 | 0 | 0 | 0 | 10 | 00 | 1 |
| j |  |  | 1 |  |  |  |  |  |  |
| jal |  |  | 1 |  |  |  |  |  |  |

## 中断机制设计

### 总体设计

**单级中断：**

从中断在main电路中需要的几个信号开始分析，首先需要由op、funct、rs产生的判断MFC0、MTC0、ERET三个控制信号的电路（ir\_control）。

其次需要对pc转换以及临时存储的电路（pc\_change），此电路需要向main提供各个中断的入口地址，并在中断开始时根据中断号将此地址送给pc，此后pc又要变成正常运作，意味着此电路需要中断开始时得到一个小段高脉冲来控制这一过程。当MTC0指令来时，需要在这用寄存器缓冲从regfile获得的原地址，以便eret命令时再给pc。当eret来时又提供一次选择，选择MTC0指令获得的地址给pc。

方才所述需要中断开始时的小段高脉冲，则需要一个电路（interrupt\_class\_judge）来产生，除此之外此电路还需要由中断按钮判断此时按下的中断号，并用eret信号清零。

**多级中断：**

在main电路中并无大改。

### 硬件设计

**单级中断：**

ir\_control：根据MTC0、MFC0、ERET三条指令的特性，发现它们op都相同（010000）且不为0，方便了拓展，回到原来的控制单元将op为010000时各个原控制信号的值配合电路进行调整。然后这三条指令的区别为eret的funct为011000，而MTC0的rs为00100，由此可以用op、funct、rs产生这三个控制信号。

pc\_change：无中断时pc直接传下去，当中断到达，产生一个高脉冲，此时选择对应中断号的入口地址，并将原来的pc存入EPC中（此pc是一条还未执行的指令），接着进入中断服务程序，先执行MFC0，使EPC中的值存入regfile中，当中断结束，先执行MTC0，从regfile中获得地址存入寄存器中，ERET时送给main中pc寄存器的入口。

interrupt\_class\_judge：单级中断由于没有中断屏蔽的电路，导致产生中断开始的高脉冲相对复杂，这个真没什么思路，要靠自己东搞西搞把它凑出来。电路中利用了老师提供的中断信号产生电路，将三个这样的电路的输出导入到优先编码器以得到中断号，这三个输出或的结果表示中断程序正在执行，取非后将此结果反馈到1、2、3号中断的输入端进行与运算，以屏蔽在中断执行中的其他中断按下。再有ERET控制使三个中断信号产生电路清零。

**多级中断：**

pc\_change：为配合在一个中断执行过程中被其他中断打断的情况，需要转存 进中断前的pc值，从一个regfile的寄存器存到另一个，增加了一个二路选择器。

interrupt\_class\_judge：增加了中断屏蔽相关电路，即是一个可以存三个中断的栈，可以通过当前中断号来对非高级中断屏蔽，ERET来时出栈，有更高级中断来时入栈，另外对中断信号产生电路的清零也从中断结束变成了中断开始的第一个周期，使其可以存同一中断的按键两次。因为现在有了屏蔽，所以中断开始的高脉冲只要简单将三个或起来即可。

### 软件设计

**单级中断：**

中断服务程序第一句写MFC0，然后写将中断服务程序中用到的regfile寄存器存入RAM（本中断服务的主程序没用RAM），中间写可以在数码管显示的主程序，之后写从RAM载入regfile最后写MTC0和ERET。

中断服务程序贴在benchmark.hex的后面，算好在哪个位置，在pc\_change中进行选择。

**多级中断：**

每个中断的MTC0、MFC0所使用的寄存器都必须不同。

## 流水CPU设计

### 总体设计

原来的部件、模块之间间距太小，首先对原始CPU进行拉伸。将指令过程分成5个阶段 IF 、ID 、EX、MEM、WB需，要根据各个控制信号在微指令中发挥的作用确定在每一个段内应该部署的器件，并判断哪些信号需要传递给下一个段。在段之间插入各段的接口寄存器，并把其拉长使各段分割更明显，接口寄存器的输入输出的位置最好对应，方便查看理解。在段内可以适当增加隧道的使用避免线路布局太乱，段间的传输或反馈则不适用隧道，不然很难找另一端。

### 流水接口部件设计

最大的问题是判断哪些信号该输入到下一段哪些在本段就使用，理想流水线在这点上考虑的较少。由于不要插气泡和阻拦上一段的数据，因此对接口的控制只需clk即可。

### 理想流水线设计

停机信号halt的处理与之前不同，由于最后一条syscall结束后还有3条00000000仍在跑，halt需要寄存器存下以持续关停pc寄存器。由于没有J、B指令所以pc直接在IF段自增4即可。控制信号在ID段产生，后续部件控制信号不再单独生成，一级一级传递，pc、IR每一段都要传递以便于调试。

## 数据转发流水线设计

重定向流水线的设计在理想流水线之上，因此在这仅仅阐述设计改动的部分。

### 总体设计

分支、loaduse、重定向三者的优先级为：分支>loaduse>重定向，又可以同时触发。

regfile写变为时钟下降沿触发。

分支跳转地址的反馈交给MEM段。

产生halt信号交给了EX段而非ID段。

MemtoReg、lh交给了WB段。

syscall在每一段都传并且都使用。

增加了分支次数与loaduse次数的计数。

增加了判断是否重定向的部分。

增加了判断是否为loaduse的部分。

### 流水接口部件设计

每个接口寄存器增加了clear、en两种控制，clear作用为产生气泡，而en作用为向前给出阻塞信号（流水线停顿）避免当前指令被新指令取代。当然这两个控制信号只是在需要的时候才接。

### 分支判断

是否分支跳转，由jump、jr以及B指令判断跳转后的信号，三者或的结果决定。

### 重定向判断

主要思想为判断在EX段要进行运算的寄存器是否与在MEM段、WB段还未写回的寄存器号是否相等，即使相等，也要判断在这三个段上运行的是何种指令，若符合条件，则产生四种结果，即MEM写回EX段的RS寄存器、MEM写回EX段的RT寄存器、WB写回EX段的RS寄存器、WB写回EX段的RT寄存器。

如果MEM、WB都要写回同一个寄存器，则优先MEM段写回。

### loaduse判断

只要EX段的rt与ID段的rs或rt相等，且此时指令又是存储指令时，可判定为loaduse情况，并由判断的顺序可知，loaduse判断比重定向更早。

# 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控制PC寄存器使能端，当需要进行停机时，Halt控制信号为1，经过非门之后为0，使时钟端失效，整个电路停机。如图 3‑1所示。

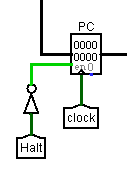


图 3‑1程序计数器（PC）

1. FPGA实现：

程序计数器PC的Verilog代码如下：

module pc(clock,Halt,reset,i\_pc,o\_pc);

input wire clock,Halt,reset;

input wire [31:0] i\_pc;

output reg [31:0] o\_pc;

initial begin

o\_pc=0;

end

always @(posedge clock or posedge reset or posedge Halt) begin

if(reset) begin

o\_pc = 0;

end

else if(Halt) begin

o\_pc = o\_pc;

end

else begin

o\_pc = i\_pc;

end

end

endmodule

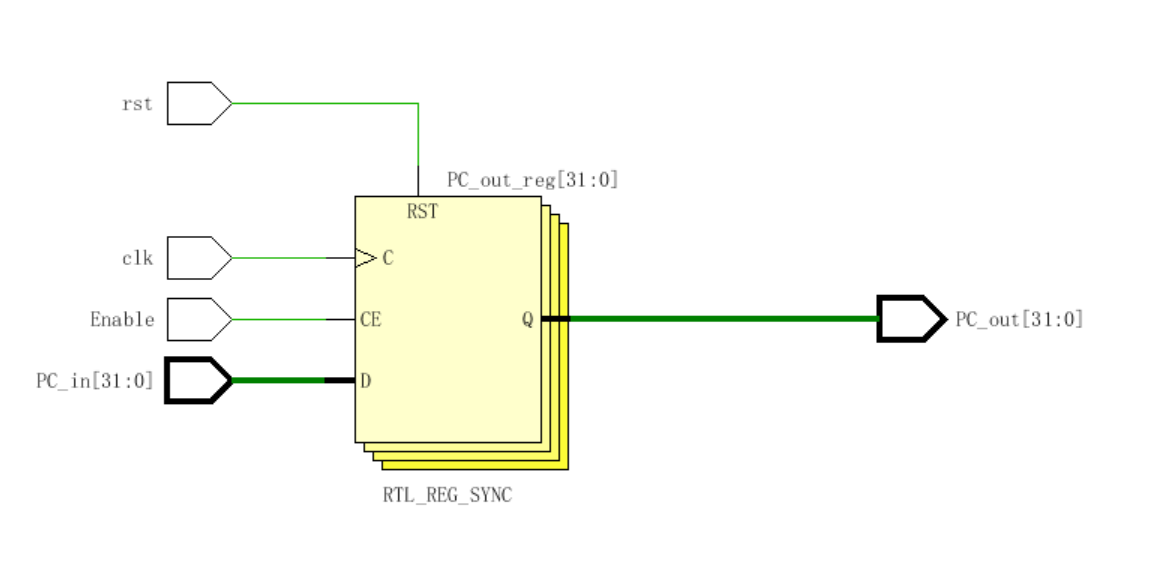


图 3‑2 PC的线路图

1. 指令存储器（IM）
2. 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 3‑3所示。



图 3‑3指令存储器（IM）

1. FPGA实现：

指令存储器IM的Verilog代码如下：

module ROM(rom\_o, addr);

input [9:0] addr;

output reg [31:0] rom\_o;

reg [31:0] mem [0:1023];

initial begin

$readmemh("F:\\text.hex", mem);

rom\_o = 0;

end

always @( addr )

begin

rom\_o = mem[addr];

end

endmodule

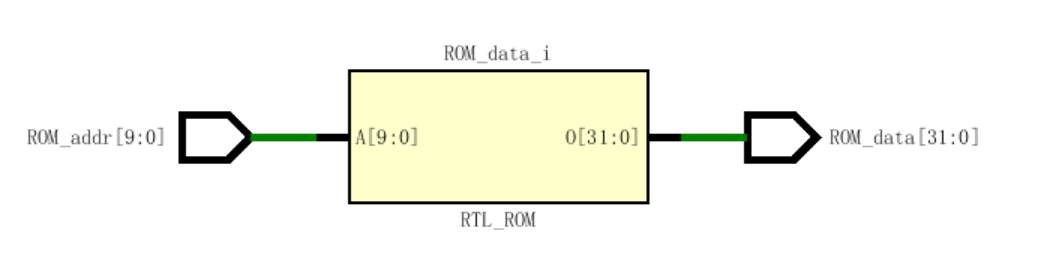


图 3‑4 IM的线路图

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。

1. 运算器（ALU）
2. Logism实现：

使用整体封装的封装的形式，通过ALUop来选择对应的运算操作，X、Y为两个32位输入，R为32位输出结果，R2为对应操作时余数、是否溢出等结果显示，Zero端为独立即时判断的X、Y是否相等输出。

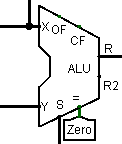


图 3‑5 运算器（ALU）

1. FPGA实现：

运算器ALU的Verilog代码如下：

module general\_ALU(

input [31:0] X\_input0,

input [31:0] X\_input1,

input Shift,

input [31:0] Y\_input0,

input [31:0] Y\_input1,

input [31:0] Y\_input2,

input [31:0] Y\_input3,

input [1:0] ALUSrc,

input [5:0] Funct,

input [5:0] op,

output Zero,

output reg[9:0] R2\_11,

output [31:0] R

);

initial

begin

R2\_11 = 0;

end

wire [31:0]X;

wire [31:0]Y;

wire [3:0]S\_temp;

assign X = Shift? X\_input1 : X\_input0;

assign Y = ALUSrc[1]? (ALUSrc[0]? Y\_input3 : Y\_input2): (ALUSrc[0]? Y\_input1 : Y\_input0);

ALU uALU1(.X(X), .Y(Y), .Result(R), .AlU\_OP(S\_temp), .Zero(Zero));

ALUCtrl uALUCtrl1(.Funct(Funct), .op(op), .ALUOP(S\_temp));

always@(R)

begin

R2\_11 = R[11:2];

end

endmodule

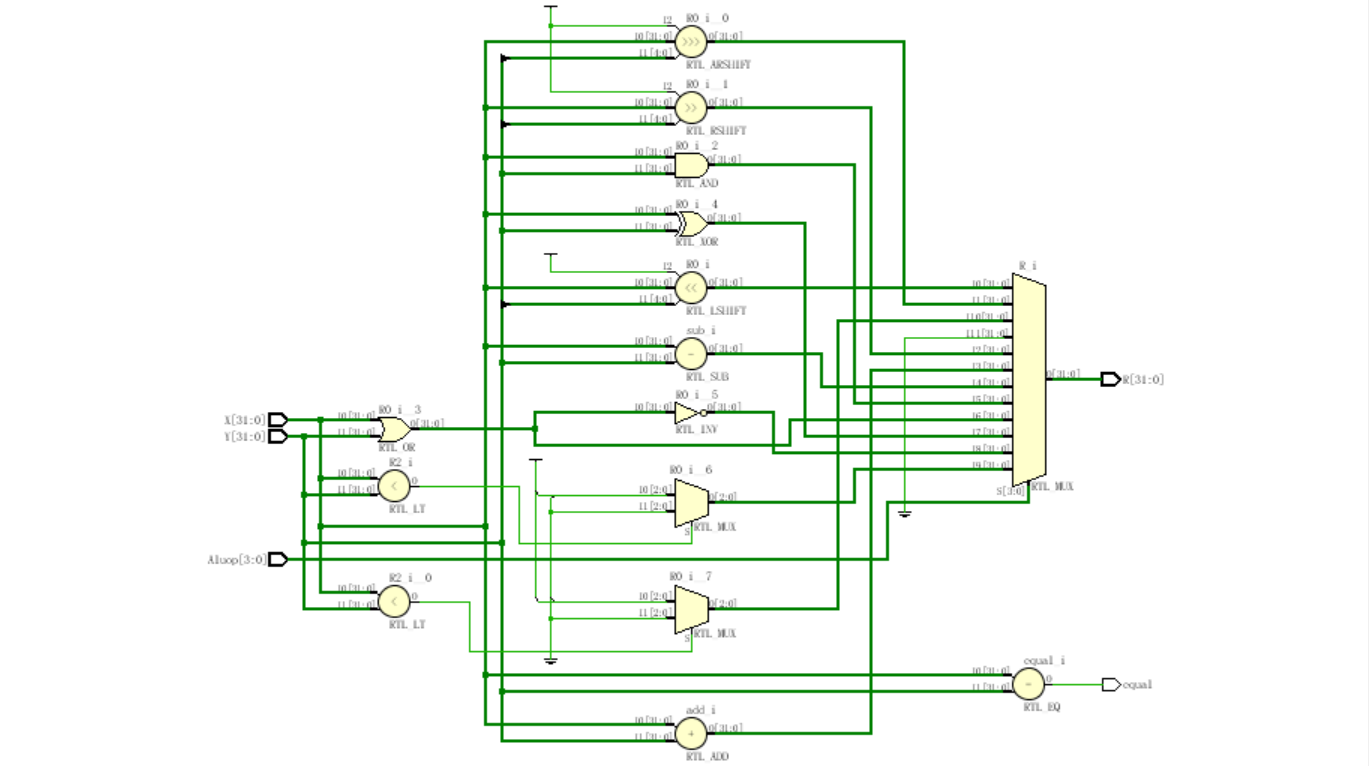


图 3‑6 ALU的线路图

1. 数据存储器（RAM）
2. 实现代码：

module RAM(

input clk,

input MemRead,

input MemWrite, // Memwrite= ram\_write

input [ADDR\_WID-1:0] ALU, // ALU == ram\_addr

input [DATA\_WID-1:0] RegFile\_R2, // RegFile\_ R2=ram\_din

output [DATA\_WID-1:0] Data // Data=ram\_dout

);

parameter DATA\_WID = 32;

parameter ADDR\_WID = 10;

reg [DATA\_WID-1:0] ram\_content[2\*\*ADDR\_WID-1:0];

always@(posedge clk)

begin

if(MemWrite)

begin

ram\_content[ALU]<=RegFile\_R2;

end

//if(MemRead)

// begin

// Data = ram\_content[ALU];

// end

end

assign Data = ram\_content[ALU];

endmodule

1. verilog线路图：

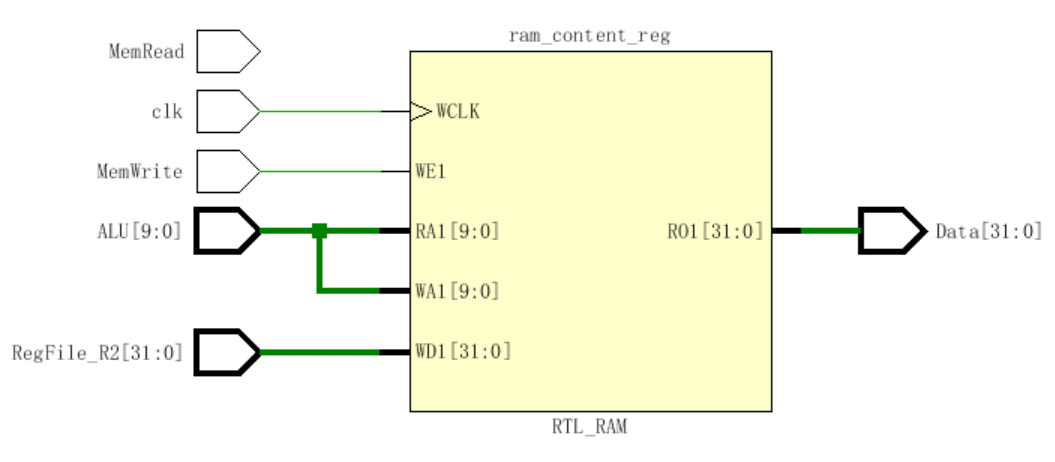


图 3‑7 RAM的线路图

1. 寄存器组（RF）
2. 实现代码：

module RegFile(r1,r2,RW,Din,WE,clk,R1,R2);

input clk;

input WE;

input [4:0]RW;

input [31:0]Din;

input [4:0]r1;

input [4:0]r2;

output [31:0]R1;

output [31:0]R2;

reg [31:0] regfile[31:0];

always@(posedge clk)

begin

if(WE && (RW!=5'd0))

begin

regfile[RW]<=Din;

end

end

assign R1=(r1!=5'd0)?regfile[r1]:32'd0;

assign R2=(r2!=5'd0)?regfile[r2]:32'd0;

endmodule

1. Verilog线路图：

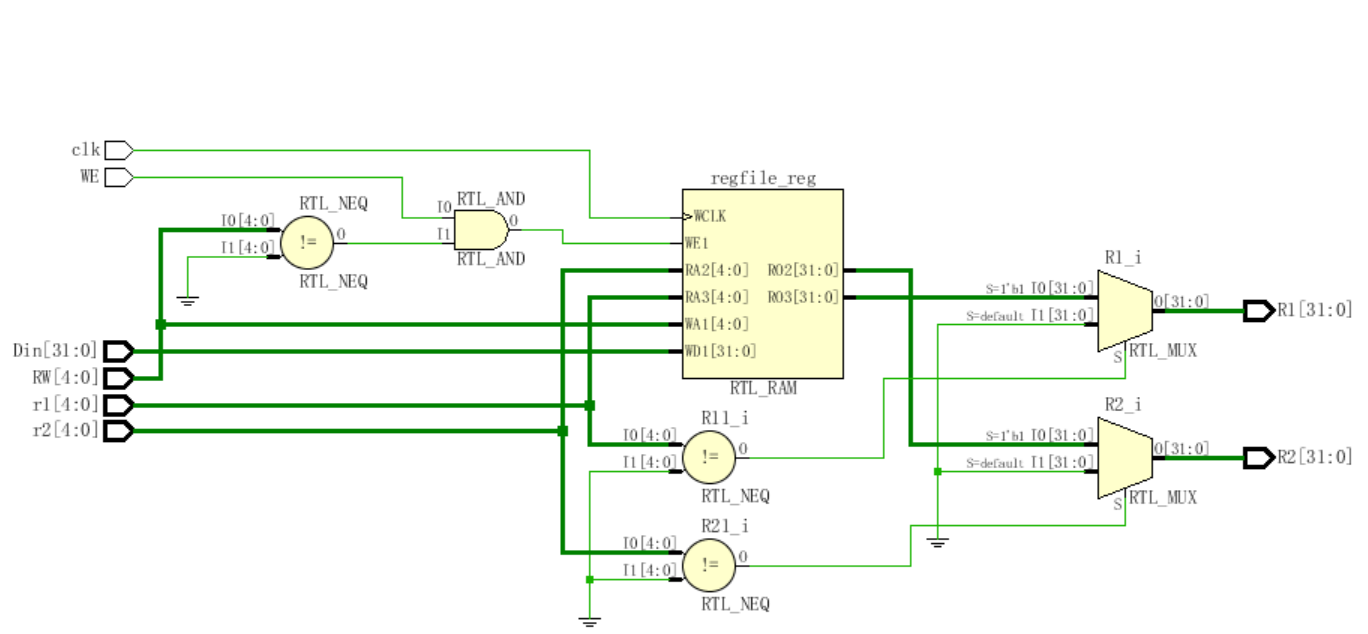


图 3‑8 RF的线路图

### 数据通路的实现

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

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

表格 3‑1 指令系统数据通路表

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

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。

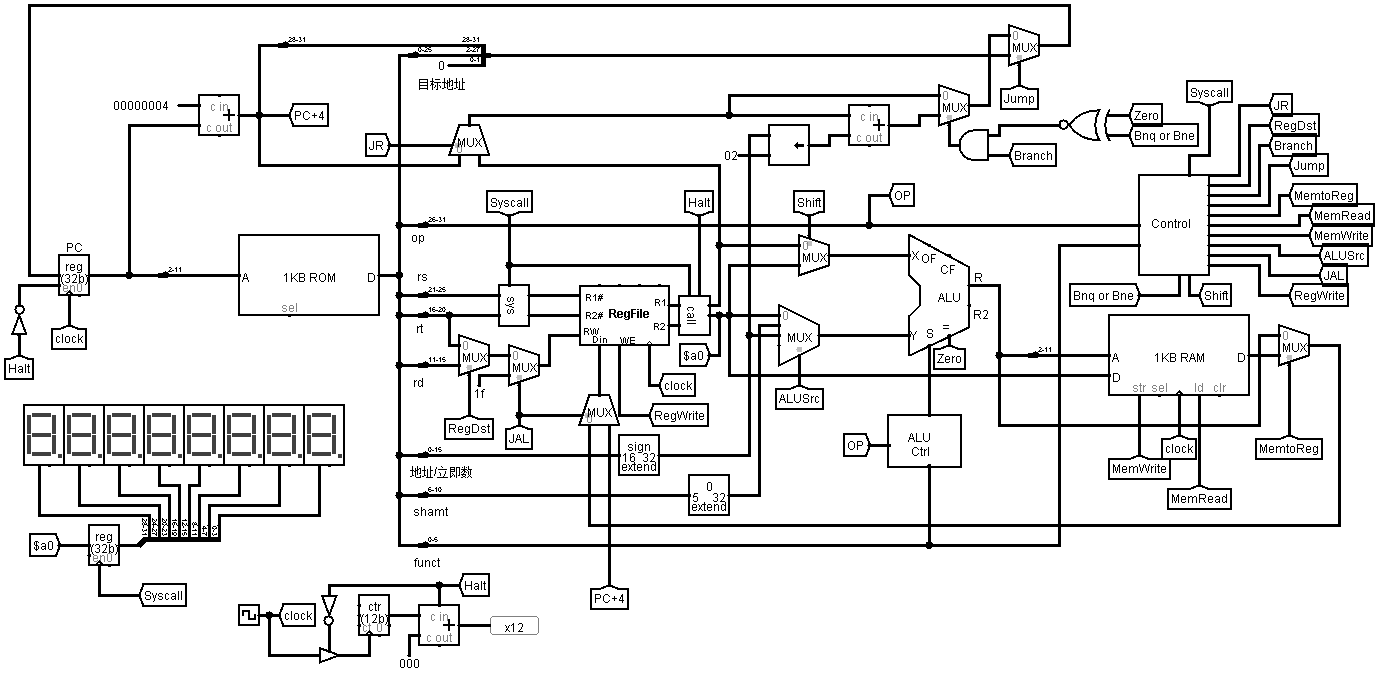


图 3‑9 单周期CPU数据通路（Logism）

在Vivado中使用Verilog语言搭建的数据通路的原理图如图 3‑10所示。

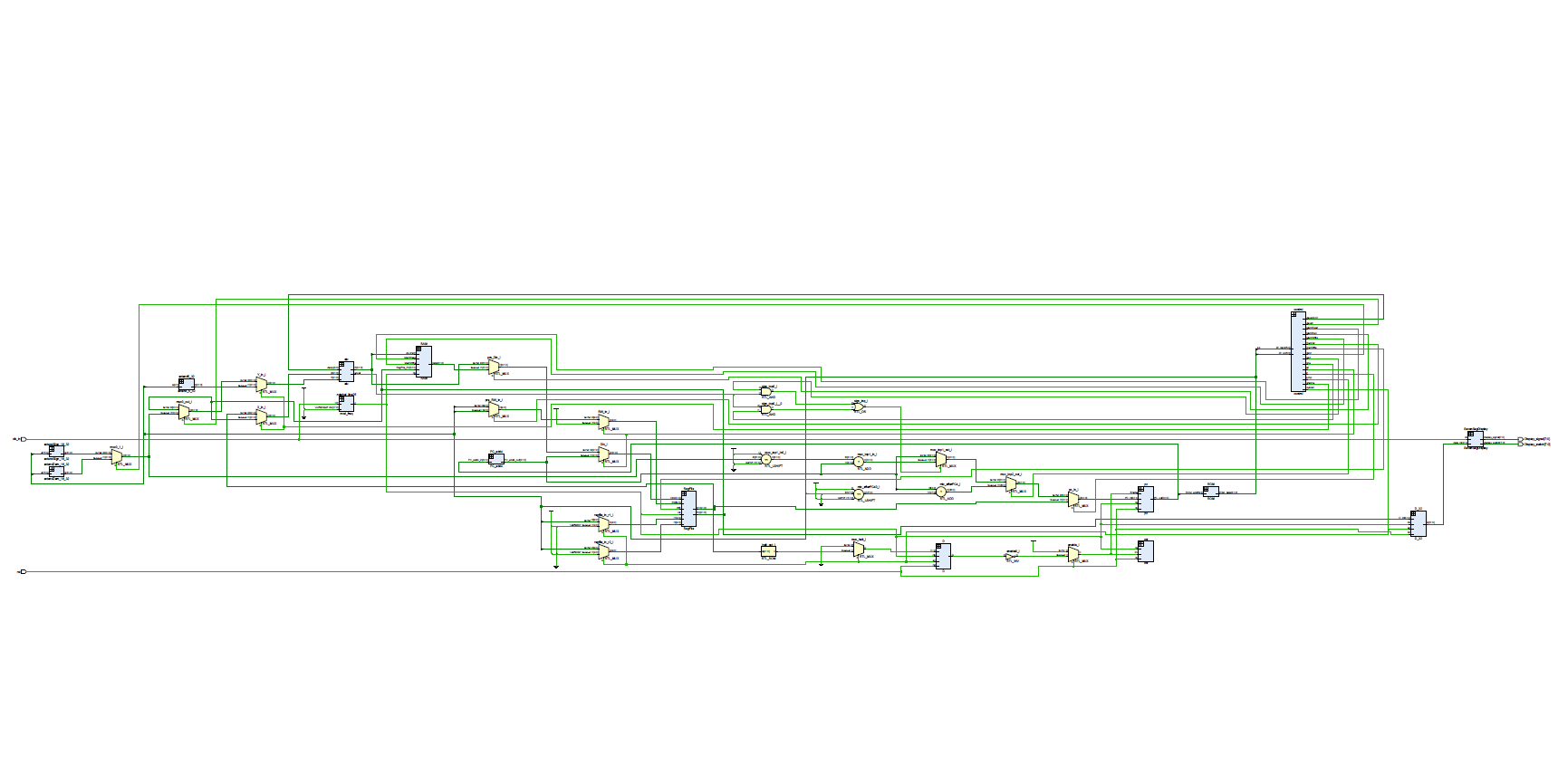


图 3‑10单周期CPU数据通路（FPGA）

### 控制器的实现

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

主控制器

对照表格 3‑2所示。

表格 3‑2 主控制器控制信号

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | RegDst | Branch | Jump | MemtoReg | MemRead | MemWrite | ALUSrc | ALUop | RegWrite |
| sllv | 1 | 0 | 0 | 0 | 0 | 0 | 11 | 00 | 1 |
| xor | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 01 | 1 |
| addi | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 00 | 1 |
| addu | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 00 | 1 |
| and | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 10 | 1 |
| sll | 1 | 0 | 0 | 0 | 0 | 0 | 01 | 10 | 1 |
| sra | 1 | 0 | 0 | 0 | 0 | 0 | 01 | 10 | 1 |
| srl | 1 | 0 | 0 | 0 | 0 | 0 | 01 | 10 | 1 |
| sub | 1 | 0 | 0 | 0 | 0 | 0 | 00 |  | 1 |
| or | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 10 | 1 |
| nor | 1 | 0 | 0 | 0 | 0 | 0 | 00 | 10 | 1 |
| slt | 1 | 0 | 0 | 0 | 0 | 0 | 00 |  | 1 |
| sltu | 1 | 0 | 0 | 0 | 0 | 0 | 00 |  | 1 |
| jr | 0 | 0 | 0 |  | 0 | 0 |  |  | 0 |
| Syscall | 0 |  |  |  |  |  |  |  |  |
| addi | 0 | 0 | 0 | 0 | 0 | 0 | 10 |  | 1 |
| addiu | 0 | 0 | 0 | 0 | 0 | 0 | 10 |  | 1 |
| andi | 0 | 0 | 0 | 0 | 0 | 0 | 10 |  | 1 |
| ori | 0 | 0 | 0 | 0 | 0 | 0 | 10 |  | 1 |
| lw | 0 | 0 | 0 | 1 | 1 | 0 | 10 |  | 1 |
| lhu | 0 | 0 | 0 | 1 | 1 | 0 | 10 |  | 1 |
| sw | 0 | 0 | 0 | 1 | 0 | 1 | 10 |  | 0 |
| beq | 0 | 1 | 0 | 0 | 0 | 0 |  |  | 0 |
| bne | 0 | 1 | 0 | 0 | 0 | 0 |  |  | 0 |
| bltz | 0 | 1 | 0 | 0 | 0 | 0 |  |  | 0 |
| slti | 1 | 0 | 0 | 0 | 0 | 0 | 10 | 00 | 1 |
| j |  |  | 1 |  |  |  |  |  |  |
| jal |  |  | 1 |  |  |  |  |  |  |

1. FPGA实现

代码部分才用case方式判断指令OP以及Funct部分，由于涉及到的指令较多，以下只列出部分代码，其他指令控制信息可在表3.2中找到，编码结构大致相同。

控制单元的Verilog部分代码如下：

always@(OP or Funct) begin

if(!OP) begin

case( Funct )

6'b000100:

begin //Shift Word Left Logical Variable

RegDst=1;

Branch=0;

Jump=0;

JAL=0;

MemtoReg=0;

MemRead=0;

MemWrite=0;

ALUSrc=2'b11;

RegWrite=1;

BnqBne=0;

Shift=1;

JR=0;

Syscall=0;

Half=0;

BLTZ=0;

end

6'b100110:

begin //Exclusive OR

RegDst=1;

Branch=0;

Jump=0;

JAL=0;

MemtoReg=0;

MemRead=0;

MemWrite=0;

ALUSrc=2'b00;

RegWrite=1;

BnqBne=0;

Shift=0;

JR=0;

Syscall=0;

Half=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 3‑11所示。

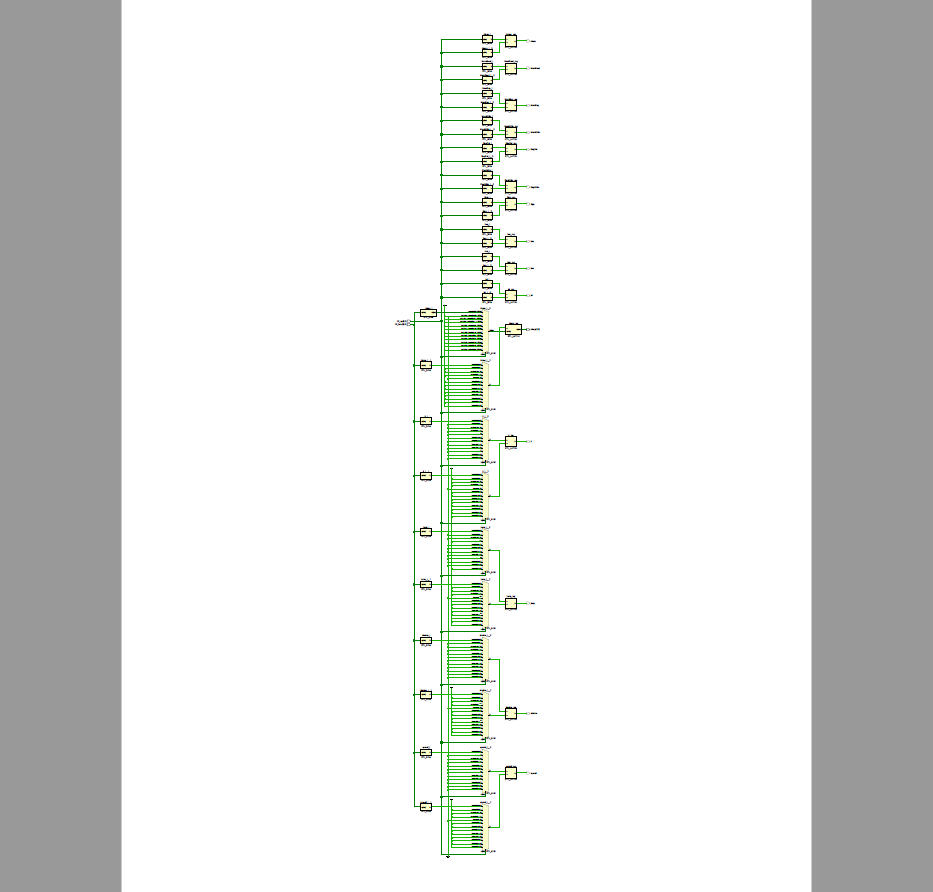


图 3‑11 vivado控制器

### 引脚绑定文件

下面代码是vivado的引脚绑定文件，参考了别人的写法，根据我们的需要修改了绑定的引脚的编号。

set\_property IOSTANDARD LVCMOS33 [get\_ports rst]

set\_property IOSTANDARD LVCMOS33 [get\_ports clk\_in]

set\_property PACKAGE\_PIN E3 [get\_ports clk\_in]

set\_property PACKAGE\_PIN N17 [get\_ports rst]

# set\_property IOSTANDARD LVCMOS33 [get\_ports {}]

# set\_property PACKAGE\_PIN H17 [get\_ports {}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_signal[7]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_signal[6]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_signal[5]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_signal[4]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_signal[3]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_signal[2]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_signal[1]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_signal[0]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_switch[7]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_switch[6]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_switch[5]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_switch[4]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_switch[3]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_switch[2]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_switch[1]}]

set\_property IOSTANDARD LVCMOS33 [get\_ports {Display\_switch[0]}]

set\_property PACKAGE\_PIN U13 [get\_ports {Display\_switch[7]}]

set\_property PACKAGE\_PIN K2 [get\_ports {Display\_switch[6]}]

set\_property PACKAGE\_PIN T14 [get\_ports {Display\_switch[5]}]

set\_property PACKAGE\_PIN P14 [get\_ports {Display\_switch[4]}]

set\_property PACKAGE\_PIN J14 [get\_ports {Display\_switch[3]}]

set\_property PACKAGE\_PIN T9 [get\_ports {Display\_switch[2]}]

set\_property PACKAGE\_PIN J18 [get\_ports {Display\_switch[1]}]

set\_property PACKAGE\_PIN J17 [get\_ports {Display\_switch[0]}]

set\_property PACKAGE\_PIN T10 [get\_ports {Display\_signal[7]}]

set\_property PACKAGE\_PIN R10 [get\_ports {Display\_signal[6]}]

set\_property PACKAGE\_PIN K16 [get\_ports {Display\_signal[5]}]

set\_property PACKAGE\_PIN K13 [get\_ports {Display\_signal[4]}]

set\_property PACKAGE\_PIN P15 [get\_ports {Display\_signal[3]}]

set\_property PACKAGE\_PIN T11 [get\_ports {Display\_signal[2]}]

set\_property PACKAGE\_PIN L18 [get\_ports {Display\_signal[1]}]

set\_property PACKAGE\_PIN H15 [get\_ports {Display\_signal[0]}]

## 中断机制实现

### ir\_control

根据MTC0、MFC0、ERET三条指令的特性，发现它们op都相同（010000）且不为0，方便了拓展，回到原来的控制单元将op为010000时各个原控制信号的值配合电路进行调整。然后这三条指令的区别为eret的funct为011000，而MTC0的rs为00100，由此可以用op、funct、rs产生这三个控制信号。

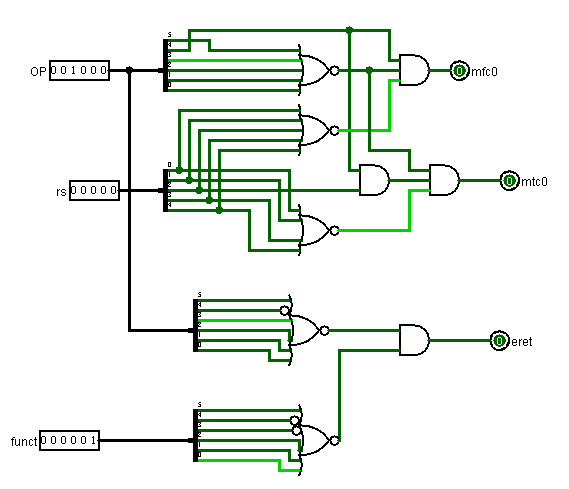


图 3‑12 中断控制信号产生 ir\_control

### pc\_change

**单级中断：**

无中断时pc直接传下去，当中断到达，产生一个高脉冲，此时选择对应中断号的入口地址，并将原来的pc存入EPC中（此pc是一条还未执行的指令），接着进入中断服务程序，先执行MFC0，使EPC中的值存入regfile中，当中断结束，先执行MTC0，从regfile中获得地址存入寄存器中，ERET时送给main中pc寄存器的入口。

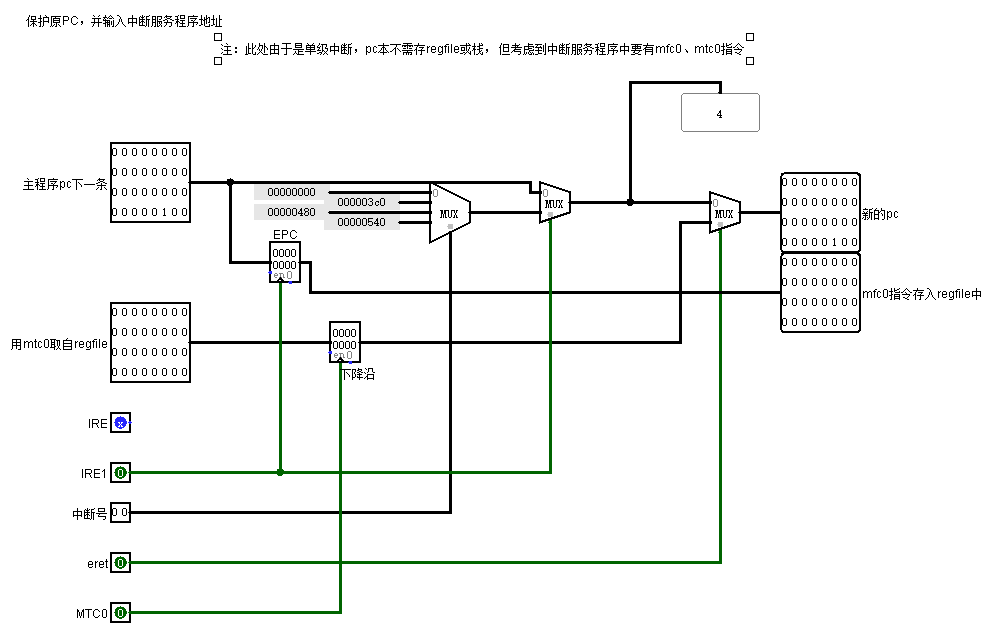


图 3‑13 PC替换单级中断 pc\_change

**多级中断：**

为配合在一个中断执行过程中被其他中断打断的情况，需要转存 进中断前的pc值，从一个regfile的寄存器存到另一个，增加了一个二路选择器。

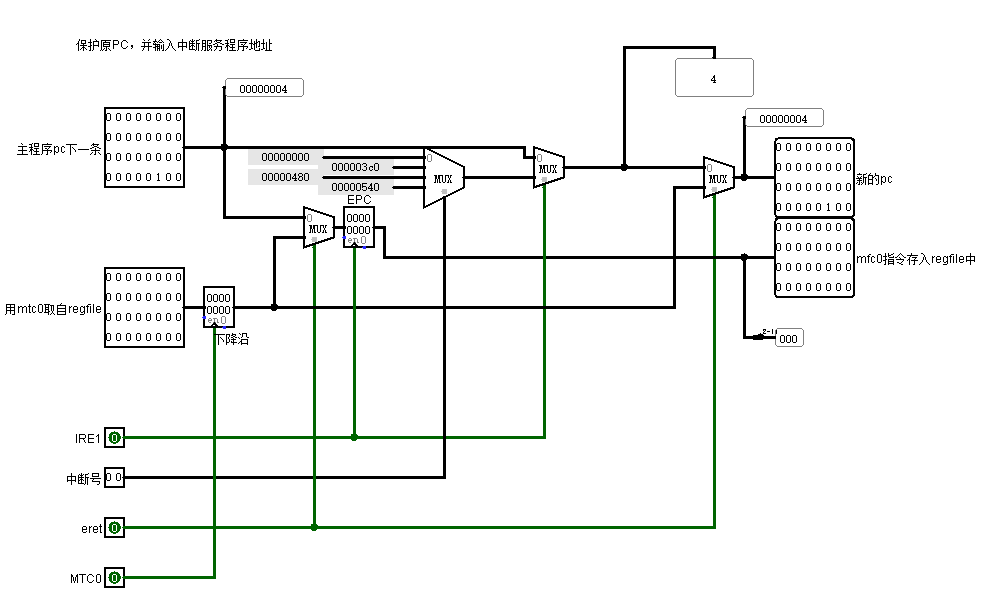


图 3‑14 PC替换多级中断 pc\_change

### interrupt\_class\_judge

**单级中断：**

单级中断由于没有中断屏蔽的电路，导致产生中断开始的高脉冲相对复杂，这个真没什么思路，要靠自己东搞西搞把它凑出来。电路中利用了老师提供的中断信号产生电路，将三个这样的电路的输出导入到优先编码器以得到中断号，这三个输出或的结果表示中断程序正在执行，取非后将此结果反馈到1、2、3号中断的输入端进行与运算，以屏蔽在中断执行中的其他中断按下。再有ERET控制使三个中断信号产生电路清零。

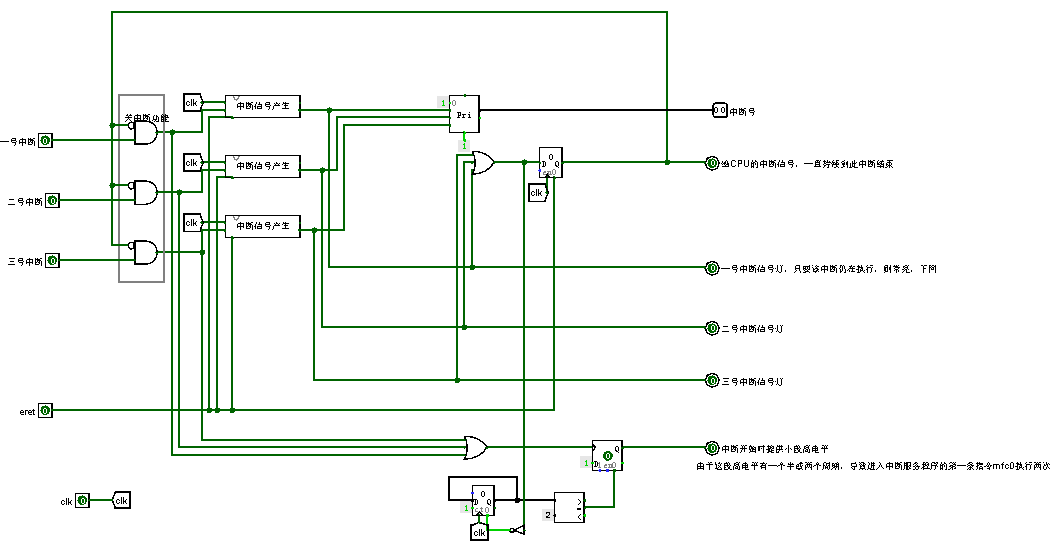


图 3‑15 中断处理 interrupt\_class\_judge单级中断

**多级中断：**

增加了中断屏蔽相关电路，即是一个可以存三个中断的栈，可以通过当前中断号来对非高级中断屏蔽，ERET来时出栈，有更高级中断来时入栈，另外对中断信号产生电路的清零也从中断结束变成了中断开始的第一个周期，使其可以存同一中断的按键两次。因为现在有了屏蔽，所以中断开始的高脉冲只要简单将三个或起来即可。

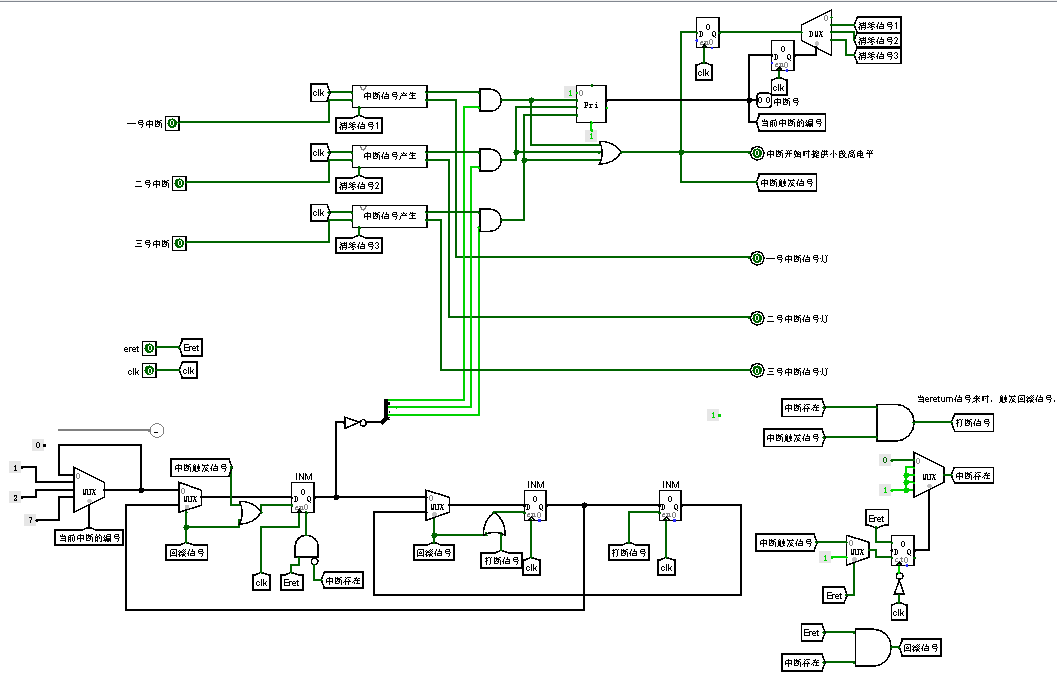


图 3‑16 中断处理 interrupt\_class\_judge多级中断

## 流水CPU实现

### 流水接口部件实现

IF/ID：简单的将PC、IR传递给下一段

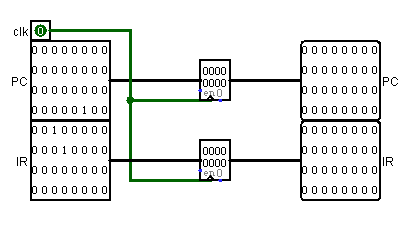


图 3‑17 IF/ID

ID/EX：halt和17位控制信号传给下一段，R1、R2为regfile的两个输出。

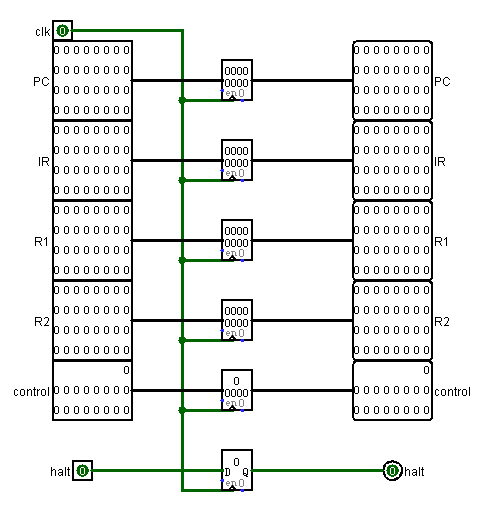


图 3‑18 ID/EX

EX/MEM：7位控制信号，Result为ALU运算的结果，R2为上一段regfile的R2

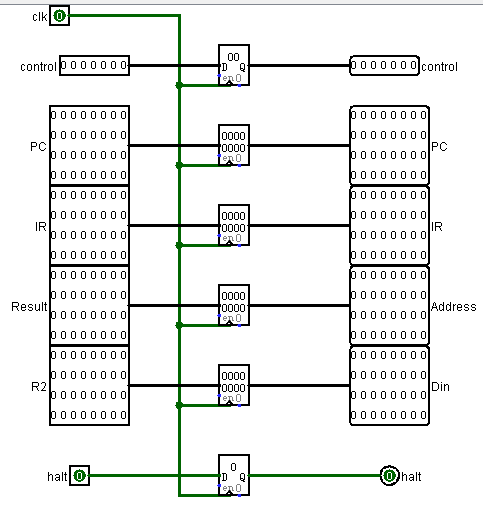


图 3‑19 EX/MEM

MEM/WB：三位控制信号，wb为将要写回的结果。

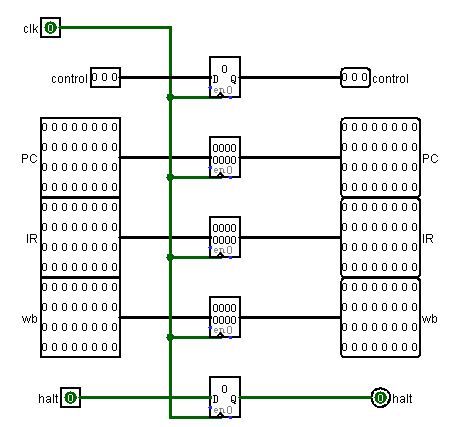


图 3‑20 MEM/WB

### 理想流水线实现

由于没有J、B指令，直接选择PC+4 。

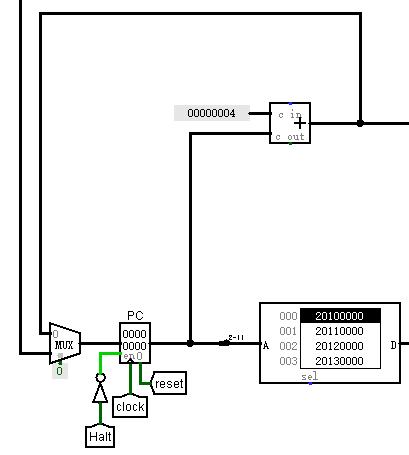


图 3‑21 PC自增

如下图，为停机与周期计数电路，未标明的输入为单周期CPU时的halt信号

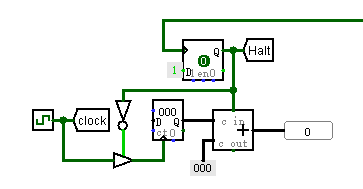


图 3‑22 停机

总体视图，控制信号在ID段产生，要跳转的地址在EX段反馈到IF。

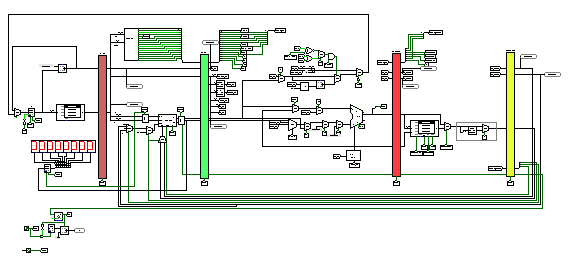


图 3‑23 总体视图

## 数据转发流水线实现

### 流水接口部件实现

ID/IF：clear不接寄存器的clear端，而是用二路选择器用0给寄存器清空，这样比原先要晚半个周期，主要是为了在EX/MEM上避免一确定分支跳转就把寄存器清零导致地址都还没来得及反馈回去。

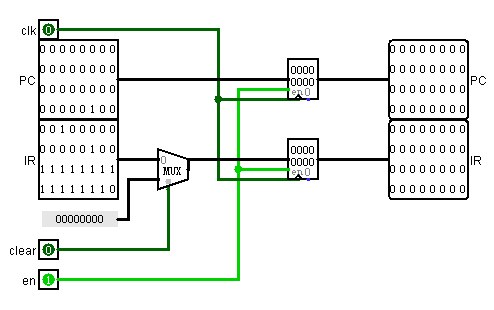


图 3‑24 ID/IF

IF/EX：输入了重定向的判断，共有四种情况。

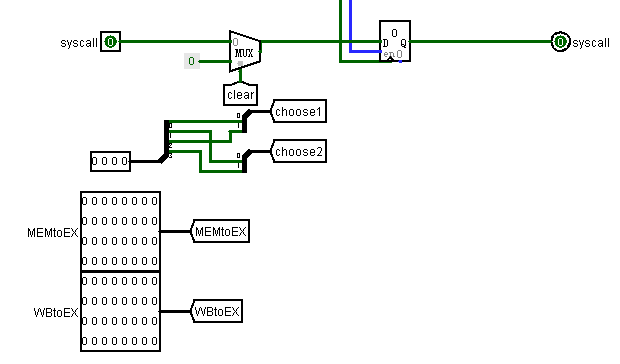


图 3‑25 IF/EX（1）

IF段的数据在进入ALU中运算前就被重定向的数据替换。

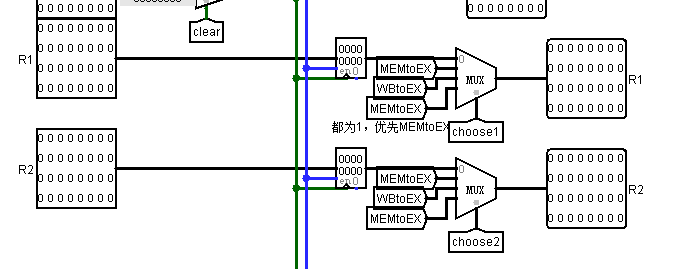


图 3‑26 IF/EX（2）

EX/MEM：将分支跳转交给了MEM段，并且也把syscall传给下一段。

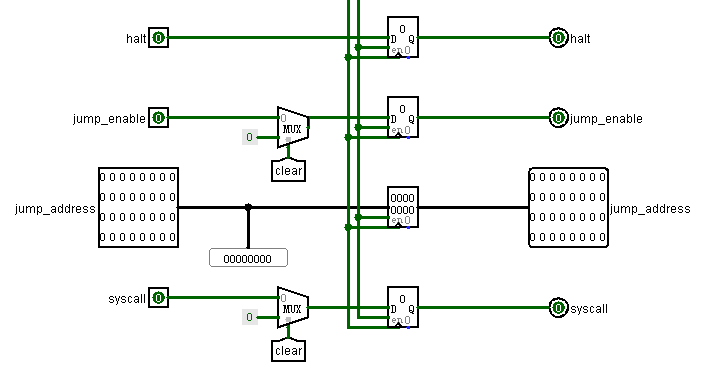


图 3‑27 EX/MEM

MEM/WB：控制信号除了syscall外还多传了lh、MemtoReg两个信号。

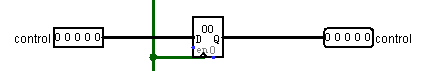


图 3‑28 MEM/WB（1）

a0为MEM段的R2，当syscall为1时，a0用来提供显示在数码管上的值，亦即数码管的显示值是在WB段产生。

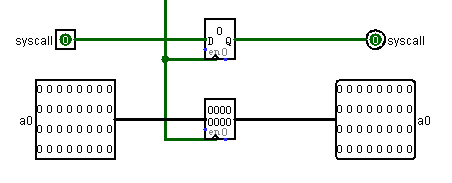


图 3‑29 MEM/WB（2）

### 数据相关检测

主要思想为判断在EX段要进行运算的寄存器是否与在MEM段、WB段还未写回的寄存器号是否相等，即使相等，也要判断在这三个段上运行的是何种指令，若符合条件，则产生四种结果，即MEM写回EX段的RS寄存器、MEM写回EX段的RT寄存器、WB写回EX段的RS寄存器、WB写回EX段的RT寄存器。

如果MEM、WB都要写回同一个寄存器，则优先MEM段写回。

在此原理之上，还有几点需要补充：

1.若译码结果为syscall指令，需要将rs、rt替换为2、4号寄存器，这与main电路的处理是一致的。

2.要写回的寄存器是rs、还是rd，在这个器件中用RegDst、Jal再做一次判断，可以避免还要分rt写回还是rd写回的情况。

3.需要根据EX段的控制信号判断在EX段的指令是否为I型，若是，则不用考虑写回到rt的情况。

4.要排除sw这条奇葩的I型指令的影响，若为sw则肯定不需写回。

5.若EX段的rs、rd为0时，本来就写回无效，此时不能将重定向的结果给下一段，因此要把rs\_ex、rt\_ex与0比较以排除这种情况。

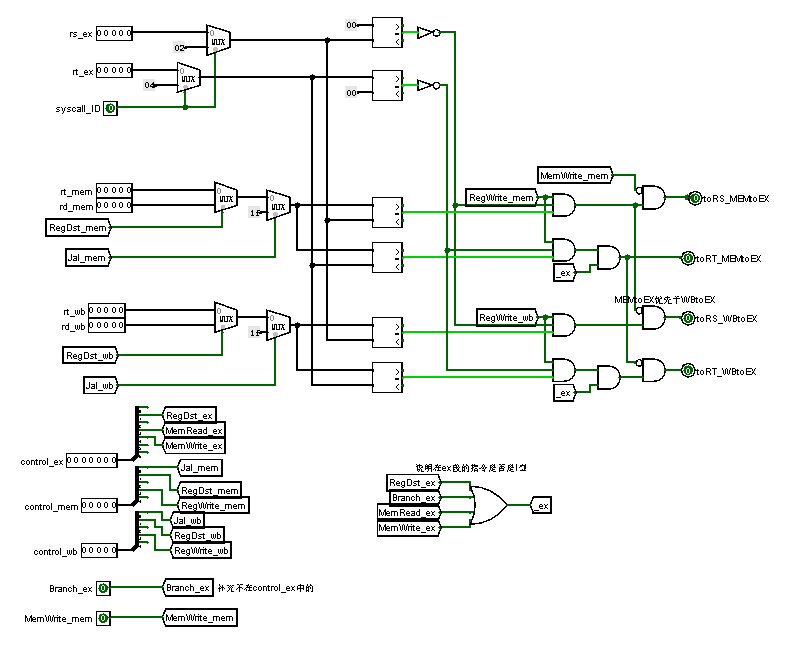


图 3‑30 数据相关检测（related\_detection2）

### 分支判断

是否分支跳转，由jump、jr以及B指令判断跳转后的信号，三者或的结果决定。

图中最右边产生的信号（clear\_jump）经过EX/MEX后反馈给pc寄存器。

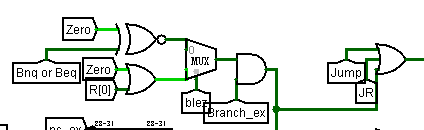


图 3‑31 分支判断

如下三图分别对应IF/ID、ID/EX、EX/MEX三个接口寄存器的控制

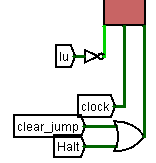


图 3‑32

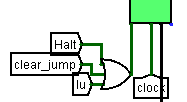


图 3‑33

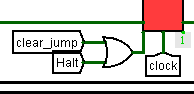


图 3‑34

当clear\_jump为1，选择需要跳转的地址给pc。

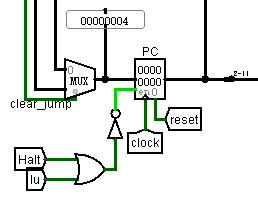


图 3‑35 pc跳转

### loaduse判断

只要EX段的rt与ID段的rs或rt相等，且此时指令又是存储指令时，可判定为loaduse情况，并由判断的顺序可知，loaduse判断比重定向更早。

如图 3‑32以及图 3‑33，loaduse电路产生的lu信号IF/ID作为en信号，阻塞IF段的信号，ID/EX中作为clear，作用为插入一个气泡。

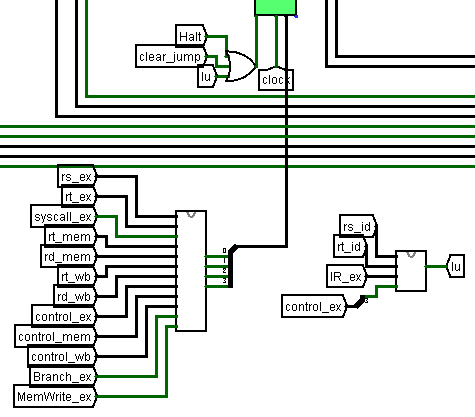


图 3‑36 数据相关检测和loaduse判断

如图，比较EX段的rs与ID段的rs或rt，并且EX段的lw得是1。

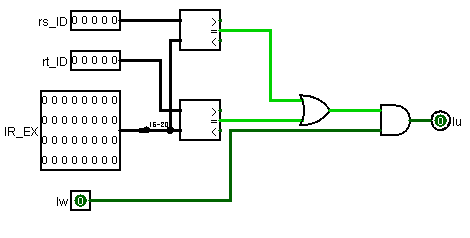


图 3‑37 loaduse判断（load\_use）

### 整体视图

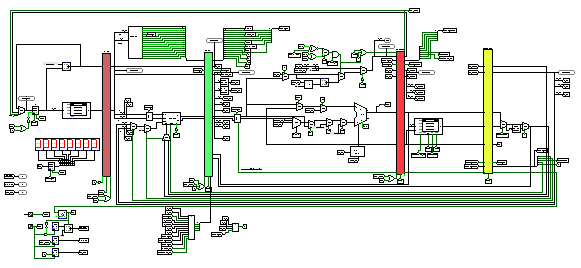


图 3‑38 整体视图

# 实验过程与调试

## 测试用例和功能测试

### 扩展指令测试（blez1）

addi $t1, $zero, 2

blez $t1, BLEZ

addi $t2, $zero, 3

BLEZ:

addi $t3, $zero, 4

### 扩展指令测试（blez2）

addi $t1, $zero, 0

blez $t1, BLEZ

addi $t2, $zero, 3

BLEZ:

addi $t3, $zero, 4

### 扩展指令测试（blez3）

addi $t1, $zero, -1

blez $t1, BLEZ

addi $t2, $zero, 3

BLEZ:

addi $t3, $zero, 4

### 扩展指令测试（lh1）

addi $t1, $zero, -2

sw $t1, 0($zero)

lh $t2, 0($zero)

### 扩展指令测试（lh2）

addi $t1, $zero, 4

sw $t1, 0($zero)

lh $t2, 0($zero)

### 扩展指令测试（lh3）

addi $t1, $zero, 0x7fff

addi $t2, $t1, 2 #$t2 = 0x00008001

sw $t2, 0($zero)

lh $t3, 0($zero) #$t3 = 0xffff8001

### 扩展指令测试（lui）

lui $t1, 0x55AA

### 扩展指令测试（sllv）

addi $t1, $zero, 5 #$t1 = 0101

addi $t2, $zero, 31

sllv $t3, $t1, $t2

### 扩展指令测试（四合一有显示）

addi $a0, $zero, -2

syscall

blez $a0, BLEZ1

addi $a0, $zero, 3

BLEZ1:

addi $a0, $a0, 4

syscall

lui $a0, 0x55AA

syscall

addi $a0, $a0, 0x7fff

addi $a0, $a0, 2

syscall

sw $a0, 0($zero)

lh $a0, 0($zero)

syscall

addi $t2, $zero, 5

sllv $a0, $a0, $t2

syscall

addi $v0, $zero, 10

syscall

### 中断测试1

mfc0 $k1, $1

mtc0 $k1, $0

eret

### 中断测试2

# push epc

mfc0 $k0, $1

add $sp, $sp, -4

sw $k0, ($sp)

# protect environment

sw $a0, -4($sp)

sw $s0, -8($sp)

add $sp, $sp, -8

# main

addi $s0, $zero, 9

addi $a0, $zero, 1

label1:

syscall

sll $a0, $a0, 4

addi $s0, $s0, -1

bne $s0, $zero, label1

# restore environment

lw $s0, ($sp)

lw $a0, 4($sp)

lw $k0, 8($sp)

add $sp, $sp, 12

mtc0 $k0, $1

# return

eret

### benchmark

太长不予贴出。

### 测试中断CPU

1.开始令CPU先运行一段时间,中间暂停时数码管显示为

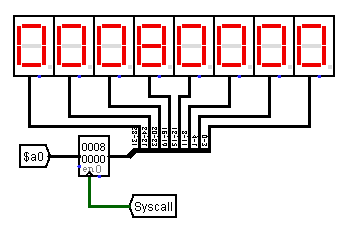
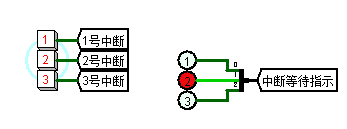


图 4.1

2.此时按下2号中断图4.2

3.进入2号中断

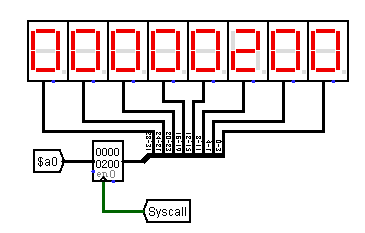
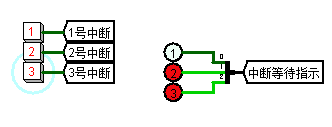


图 4.3

4.按下3号中断图4.4

5.进入3号中断

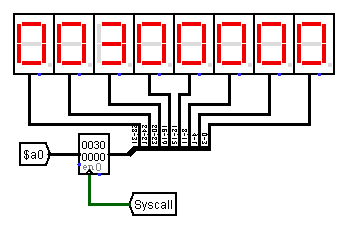


图 4.5

6.返回2号中断

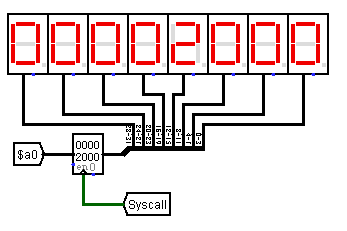


图 4.6

7.返回原程序

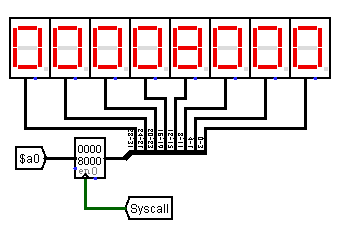
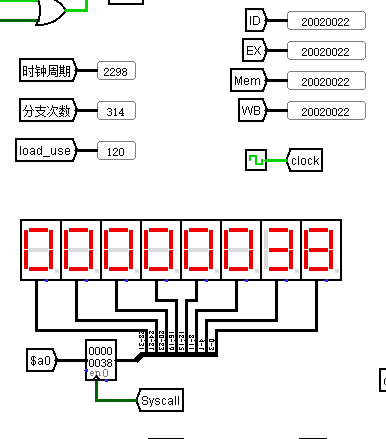


图 4.7

### 重定向测试

运行benchmark，结果如图4.8



## 性能分析

单周期CPU跑benchmark周期数为1546，而重定向周期数为2612，分支深度为3，每一次loaduse加一个气泡，所以2612 = 1546+314\*3+120。

## 主要故障与调试

### Verilog部分运算器设计故障

**故障现象：**将verilog代码烧入开发板后，运行Benchmark的过程中应该显示为8的数码管显示为3且后续数码管显示均是该位出现问题。

**原因分析：** 经过多次调试后，发现是在verilog代码ALU部分的移位处理出现了问题，此前试验中正确的移位应该是Y输入只取低5位，而我们在编写代码过程中忽略了这个细节，导致移位的处理一直与预期不相符合

**解决方案：**修改verilog代码ALU部分关于移位的表达式，后测试成功。

### 理想流水线停机故障

**故障现象：**理想流水线最后一条syscall跑完后依然在跑，执行00000000指令而无法停机。

**原因分析：**syscall在wb段本应执行完毕，但是执行完后00000000进入WB段使halt又变成了0，又把pc寄存器给激活了

**解决方案：**如下图，增加一个D触发器，使得halt为1一次即可保存结果，使停机指令一直执行。

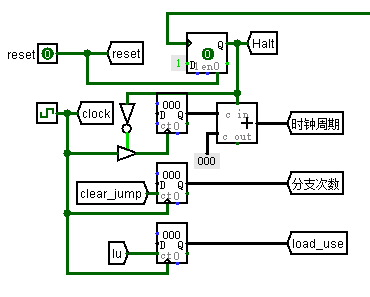


图 4‑1

### 周期数故障

中断开始的高脉冲有一个半或两个时钟周期。

**故障现象：**当中断按键在clk = 0触发时为一个半周期，当中断按键在clk = 1时为两个周期，这样会导致MTC0被触发两次。

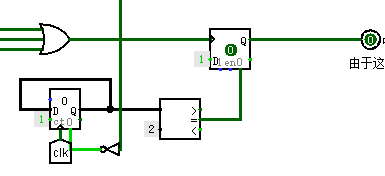


图 4‑2

**原因分析：**对脉冲计数的寄存器受clk控制所致。

**解决方案：**增加屏蔽后直接由于中断屏蔽自己而使此高脉冲只持续一个时钟周期。

### 数据相关判断故障

**故障现象：**无法正确停机，有些指令不能正确重定向，loaduse失败。

**原因分析：**此数据相关判断电路最大的问题在于我将三个段的rs、rt、td都比了一遍，共比较了8次，这把结果搞的太复杂，这样还要判断了MEM段和WB段的指令类型，情况实在太多，分析得头大，再加上几个特殊的指令sw、sll、sra、srl整个电路要考虑的太多，于是总是出问题。另外loaduse必须在ID段就判断而非与重定向一样在EX段判断。

**解决方案：**判断在EX段要进行运算的寄存器是否与在MEM段、WB段还未写回的寄存器号是否相等，即使相等，也要判断在这三个段上运行的是何种指令，若符合条件，则产生四种结果，即MEM写回EX段的RS寄存器、MEM写回EX段的RT寄存器、WB写回EX段的RS寄存器、WB写回EX段的RT寄存器。

loaduse用一个专门的电路实现。

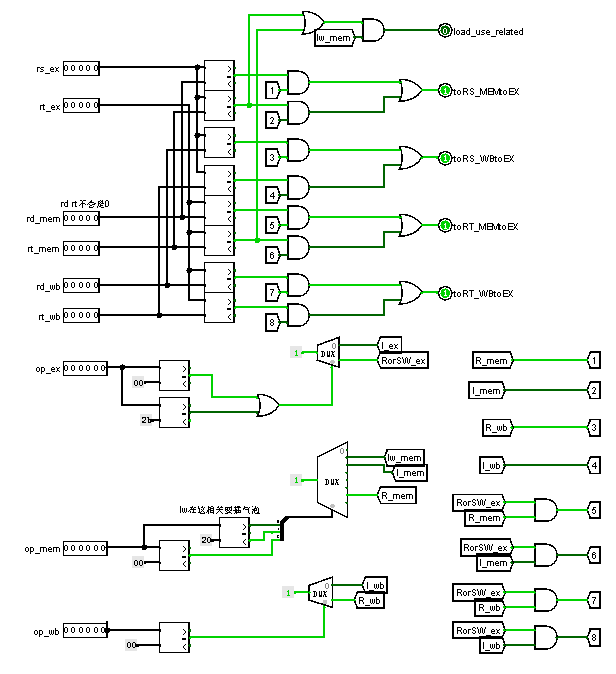


图 4‑3

### 重定向故障

**故障现象：**无法正常重定向，在不需要重定向的分支跳转指令时也会重定向。

**原因分析：**我们组的公版单周期CPU的控制信号有问题，当指令为J或B指令时，数据通路无关的控制信号未设置，而使其缺省。

**解决方案：**更改控制单元，使其能够在跳转指令时也能输出正确的信号。

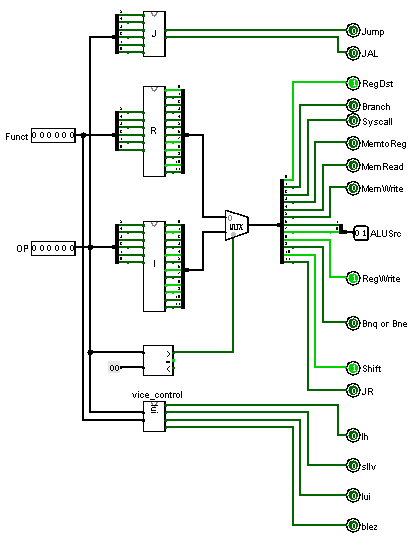


图 4‑4

### 数码管显示故障

**故障现象：**如下图，数码管的显示有闪烁，并且最后每显示一段数据就会自动清零，最后结束时显示00000001，syscall\_wb在没有syscall指令时会有毛刺导致数据显示错误。

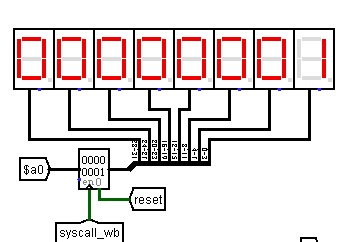


图 4‑5

**原因分析：**main电路的syscall产生的halt信号用成了在ID段产生的，这样使halt信号在非syscall信号时也会有为1的情况。

**解决方案：**将产生halt信号的电路移入EX段。

## 实验进度

表 4.1 课程设计进度表

| 时间 | 进度 |
| --- | --- |
| 第一天 | 熟悉了tower、一起写的使用，与组员分配模块，完成了ALU、RAM的相关电路，未进行仿真。 |
| 第二天 | 完成ALU、RAM的调试工作，将所有部件凑成CPU，未调试 |
| 第三天 | 对verilog语言编译成电路并进行纠错，主要负责数据通路的修改 |
| 第四天 | 对各器件进行单独测试 |
| 第五天 | 纠正了ALU的错误，完成了单周期CPU的调试，进入下一步写新增的三条指令 |
| 第六天 | 完成扩展指令与中断三条指令 |
| 第七天 | 构建中断处理电路与pc置换电路 |
| 第八天 | 完成单级中断并检查验收，开始做理想流水线 |
| 第九天 | 完成理想流水线，以及对相关性的检测电路 |
| 第十天 | 完成重定向，还有一些仍需调试，待检查 |

# 设计总结与心得

## 课设总结

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。作了如下几点工作：

1. 完成支持单级中断的单周期CPU设计与实现（FPGA平台）。
2. 实现单级与多级嵌套中断（LOGISIM平台）。
3. 完成理想流水线的多周期CPU（LOGISIM平台）。
4. 完成数据转发方式的数据冒险处理（LOGISIM平台）

## 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周末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。

体会：绝望，无助，在做单周期上板时，最大的失误是那些固定的器件regfile、ALU、RAM、ROM全都是自己写，而没有上网找现成的。我负责的ALU、RAM模块都出现的大问题，比如逻辑右移，课本上还是网上只会告诉你是>>>，而没有它真正的用法和效果演示，实际上要加signed才有用。vivado也太难用了，这也算IDE？如果是05年的还差不多，Verilog语法松散，真不知道用“[]”时到底是大数在前还是小数在前，做二路选择器用assign+？：还是用always + if。对Verilog的习惯用法太不熟，如果我们班做了数字逻辑的课设的话说不定还能有点体会，然而并没做。做扩展指令的时候倒还好说，毕竟上学期才在logisim做完一个CPU，还有一点比较麻烦就是用的别人的公版CPU，就不好从头开始改控制单元，只能在main里加个副控制单元，加几个二路选择器，如果是自己做自己的就不会有几个二路选择器挤在一起而是变成一个多路选择器了。到了中断，感觉自己不是在做课设，而是在发明中断这个机制，什么ppt上讲的IE、IR、INM那是什么能吃吗，到底在电路中处于哪个位置什么作用完全不知道。课程任务书上写MFC0、MTC0、ERET三条指令选做，这要是选做难度才大大提高，这不是选做而是必须做，不然更不知道中断是个什么机制。手上有几份别人做的中断作为参考，看了人家做的是真服——这要靠什么才能想出这么光怪陆离的做法。反正对我是一点参考作用没起到，还浪费了大把时间去研究他是怎么写的。理想流水线一个上午轻松搞定，重定向流水由遇到了大问题，还是那个这份公版CPU的问题，很多细节想不到他是怎么搞的走了很多弯路，特别是关于syscall处理被坑大了。虽说重定向比中断花的时间差不多，但是很明显重定向有路可循，让人能看到希望。听说有英文的参考资料，然而我看英文头晕。

收获：第一次知道了中断的处理步骤，课本上说的实在是太缩略了完全没法用，意识到理论与现实的巨大差距。又一步理解到了自己能力的不足，平时缺少锻炼关键时候掉链子。看着自己画的图，强迫症表示很高兴。

熟悉了logisim、mars、vivado的使用，更进一步理解了CPU的运行机制，对自己的硬件能力有了进一步提升，还是挺有成就感，虽然没能完成是一大遗憾。

建议：

1.两周时间太短，希望不要回收板子，让我们还能回去继续把没做完的给做完。

2.分组的好处是在不懂的时候可以问别人怎么操作，与别人关系更近，坏处是公版CPU没了别人就自己做不下去，看不懂他怎么想的。遇到问题一般来说自己不会的同组的人也不会，都不会还要讨论什么，因此基本上没有讨论。每天开会本来就不知道自己能做到哪里，只能走一步看一步哪里轮得到去规划别人一天的安排。分组的作用只持续到了单周期上板，之后就是各自为战了。

3.建议不要一开始就单周期上板，这样明显增大了工作量一开始就接触恶心的vivado对人是一种折磨，消耗人的精神。给三周时间还差不多，像我们这种只做到logisim重定向的就吃了大亏了。

# 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: C:\Users\Barry\Pictures\IMG_20170325_211452 (2).jpg** |