**32位MIPS处理器设计**

**实验报告**

|  |  |  |
| --- | --- | --- |
| **无42** | **林子恒** | **2014011054** |
| **无45** | **何子昊** | **2014011150** |
| **无47** | **范承泽** | **2014011199** |

目录

[一、 实验目的 5](#_Toc461371815)

[1. 熟悉现代处理器的基本工作原理； 5](#_Toc461371816)

[2. 掌握单周期和流水线处理器的设计方法。 5](#_Toc461371817)

[二、 实验分工 5](#_Toc461371818)

[1. 林子恒：负责 ALU、UART、汇编代码 的编写 5](#_Toc461371819)

[2. 何子昊：负责 单周期处理器 的编写 5](#_Toc461371820)

[3. 范承泽：负责 流水线处理器 以及 编译器 的编写 5](#_Toc461371821)

[三、 设计方案 6](#_Toc461371822)

[1. 32位ALU设计（林子恒） 6](#_Toc461371823)

[1.1 ALU功能表如下所示： 6](#_Toc461371824)

[1.2 ALU实现框图： 6](#_Toc461371825)

[1.3 说明： 7](#_Toc461371826)

[2. 单周期mips处理器设计（何子昊） 8](#_Toc461371827)

[2.1 总体框图： 8](#_Toc461371828)

[2.2 说明： 8](#_Toc461371829)

[3. UART串口设计（林子恒） 9](#_Toc461371830)

[3.1 UART结构框图： 9](#_Toc461371831)

[3.2 设计思路： 9](#_Toc461371832)

[4. 汇编程序编写（林子恒） 10](#_Toc461371833)

[4.1 程序总体框架： 10](#_Toc461371834)

[5. 编译器设计（范承泽） 11](#_Toc461371835)

[6. 流水线设计（范承泽） 12](#_Toc461371844)

[6.1 基本框架 12](#_Toc461371845)

[6.2 冒险 15](#_Toc461371846)

[6.3 最终设计框架 18](#_Toc461371847)

[四、 关键代码分析及文件清单 19](#_Toc461371848)

[1. ALU部分（林子恒） 19](#_Toc461371849)

[1.1 顶层模块： 19](#_Toc461371850)

[1.2 Add模块： 20](#_Toc461371851)

[1.3 Cmp模块： 20](#_Toc461371852)

[1.4 Logic模块： 21](#_Toc461371853)

[1.5 Shift模块： 21](#_Toc461371854)

[1.6 16，8，4，2，1 Shift模块 21](#_Toc461371855)

[2. 单周期部分（何子昊） 22](#_Toc461371856)

[2.1 Control模块，代码文件为Control.v 22](#_Toc461371857)

[2.2 RegFile模块，代码文件为regfile.v 25](#_Toc461371858)

[2.3 DataMem模块，代码文件为DataMem.v 25](#_Toc461371859)

[2.4 Peripheral模块，代码文件为Peripheral.v 26](#_Toc461371860)

[2.5 Rom模块，代码文件为Rom.v 27](#_Toc461371861)

[2.6 单周期模块，代码文件为CPU\_Single.v 27](#_Toc461371862)

[3. UART部分（林子恒） 29](#_Toc461371863)

[4. 汇编程序部分（林子恒） 30](#_Toc461371864)

[4.1 开始部分： 30](#_Toc461371865)

[4.2 将内核态转换为用户态 30](#_Toc461371866)

[4.3 UART轮询： 30](#_Toc461371867)

[4.4 计算最大公约数程序： 30](#_Toc461371868)

[4.5 中断处理代码（显示数码管）： 31](#_Toc461371869)

[5. 编译器部分（范承泽） 32](#_Toc461371870)

[5.1 主程序： 32](#_Toc461371871)

[6. 流水线部分（范承泽） 33](#_Toc461371872)

[6.1 总体设计（Pipeline.v） 33](#_Toc461371873)

[6.2 IF\_ID 36](#_Toc461371874)

[6.3 Control 37](#_Toc461371875)

[6.4 ID\_EX 38](#_Toc461371876)

[6.5 Forward 40](#_Toc461371877)

[6.6 EX\_MEM 41](#_Toc461371878)

[6.7 Memory 42](#_Toc461371879)

[6.8 MEM\_WB 42](#_Toc461371880)

[6.9 Harzard 43](#_Toc461371881)

[五、 仿真结果及分析 44](#_Toc461371882)

[1. ALU 44](#_Toc461371883)

[2. 单周期： 46](#_Toc461371884)

[3. 流水线 47](#_Toc461371885)

[3.1 总体仿真 47](#_Toc461371886)

[3.2 其他部分 47](#_Toc461371887)

[六、 综合情况 48](#_Toc461371888)

[1. 时间综合情况 48](#_Toc461371889)

[2. 空间综合情况 48](#_Toc461371890)

[七、 硬件调试情况 49](#_Toc461371891)

[八、 思想体会 50](#_Toc461371892)

[1. 林子恒 50](#_Toc461371893)

[2. 何子昊 50](#_Toc461371894)

[3. 范承泽 51](#_Toc461371895)

# 实验目的

## 熟悉现代处理器的基本工作原理；

## 掌握单周期和流水线处理器的设计方法。

# 实验分工

## 林子恒：负责 ALU、UART、汇编代码 的编写

## 何子昊：负责 单周期处理器 的编写

## 范承泽：负责 流水线处理器 以及 编译器 的编写

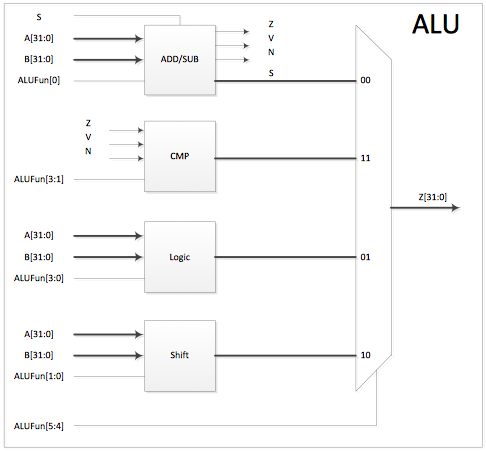
# 设计方案

## 32位ALU设计（林子恒）

### ALU功能表如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| **类型** | **功能** | **ALUFun** | **描述** |
| 算术 | ADD | 000000 | S=A+B |
| SUB | 000001 | S=A-B |
| 位运算 | AND | 011000 | S=A&B |
| OR | 011110 | S=A|B |
| XOR | 010110 | S=A^B |
| NOR | 010001 | S=~(A|B) |
| ‘A’ | 011010 | S=A |
| 移位运算 | SLL | 100000 | S=B<<A[4:0] |
| SRL | 100001 | S=B>>A[4:0] |
| SRA | 100011 | S=B>>a[4:0] 算术移位 |
| 关系运算 | EQ | 110011 | If(A==B) S=1 else S=0 |
| NEQ | 110001 | If(A!=B) S=1 else S=0 |
| LT | 110101 | If(A<B) S=1 else S=0 |
| LEZ | 111101 | If(A<=0) S=1 else S=0 |
| LTZ | 111011 | If(A<0) S=1 else S=0 |
| GTZ | 111111 | If(A>0) S=1 else S=0 |

### ALU实现框图：



### 说明：

#### 加法直接采用＋运算符实现，减法运算通过补码用加法实现

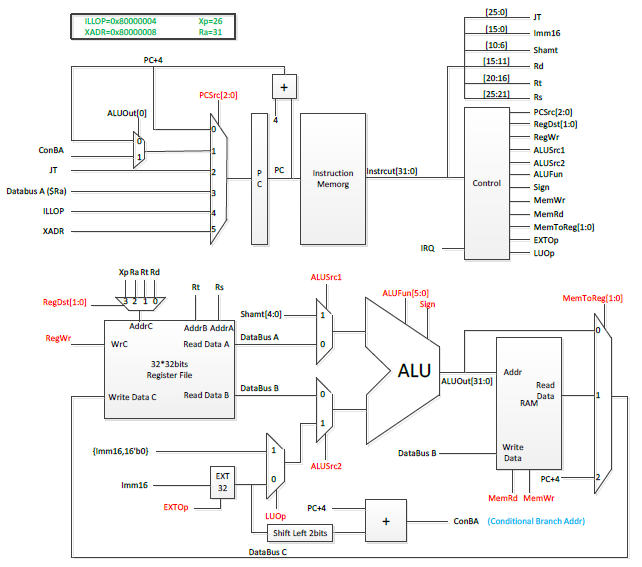
#### 比较运算根据减法运算的结果产生

#### 移位运算通过分解为16 位移位、8 位移位、4 位移位、2 位移位、1 位移位得到运算结果

#### 逻辑运算直接用逻辑运算符得到结果

## 单周期mips处理器设计（何子昊）

### 总体框图：

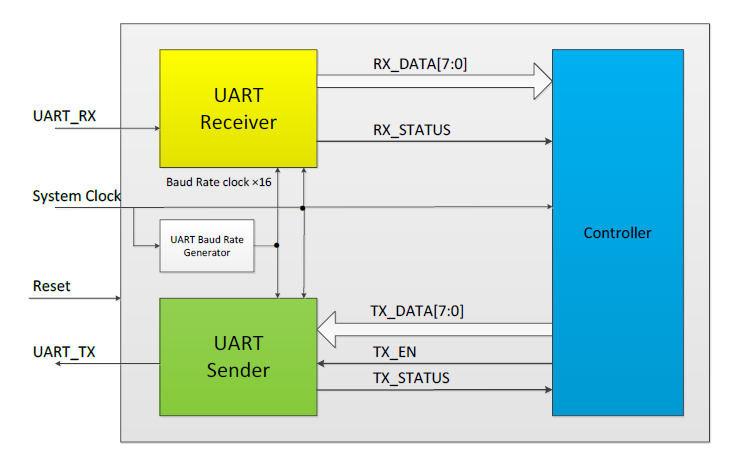


### 说明：

单周期CPU整体上由ALU、Control、RegFile、DataMem、ROM、Peripheral组成，各模块在CPU\_Single中被连接起来，并根据控制信号增加相应的选择器选通某路信号，每条指令需要一个时钟周期完成，因此时钟最高频率被最长路径所限制。在下个部分分析各个模块的代码，其中ALU模块在前面已经分析，不再赘述。

## UART串口设计（林子恒）

### UART结构框图：



### 设计思路：

此部分的UART接收器与UART发射器都与之前所做第四次实验的部分相同，不同之处为控制器模块。

在这里我们采用轮询的方式，当UART的控制器接收到两次接收器传过来的RX\_STATUS信号时，且控制器接收到CPU计算好的最大公约数时，控制器就将这个数传递给发射器，并利用TX\_EN通知发射器使能。

## 汇编程序编写（林子恒）

### 程序总体框架：

由于本次CPU要求以PC[31]为监督位定义用户态、内核态，在内核态时不能发生中断，因此在程序一开始即将内核态转换为用户态，并且在中断处理程序结束后也要将内核态转换为用户态。

程序总体框架如下：

用户态：

转换为用户态

初始化，UART轮询

计算最大公约数

内核态（中断处理程序）：

保存现场

数码管扫描显示

恢复现场

转换为用户态

跳转回 $k0 （产生中断位置）

## 编译器设计（范承泽）

为了让mips处理器运转起来，我们需要为其写汇编程序，但处理器并不能够直接读取汇编程序，所以需要编译器将汇编语言转换为机器码。

为了达成这个目的，我选择用python来担任这个工作。在编写程序的过程中，我主要考虑到了可拓展性，即增加新的指令，所以我采用了数据和程序分离的结构，这样也避免了在代码中写过多的分类讨论和逻辑判断，增强了代码的可读性和效率。

Data.py存储了机器码和汇编语言的结构，以及寄存器的别名与其二进制的对应。

Main.py则是汇编语言的主程序，整体思路如下：

## 打开汇编语言文件，读入到asms

## 将每行汇编构建一个instruction类，并根据出现的先后顺序赋予address，将所有instruction类型的实例存到insts列表中

## 循环遍历insts，处理label，将label存入labels的字典中，便于下一步替换label

## 再次循环遍历insts，完成以下

## 去除空格，将指令各个组成部分按照空格split成list，存入asm中

## 分别处理list，根据不同的特征将其转换为机器码（imm、shamt、regs、labels），存入inst.binary中

## 根据type\_inst的内容来决定为asm中的不同操作数的赋值顺序，写入inst.binary

## 合并inst.binary为一个字符串，打印

## 流水线设计（范承泽）

### 基本框架

流水线是数字系统中一种提高系统稳定性和工作速度的方法，广泛应用在高档CPU的构建中。根据MIPS处理器指令的特点，将整体的处理过程分为取指令（IF）、指令译码、执行、存储器访问和寄存器写回五级，对应多周期CPU的五个处理阶段。如图1所示，一个指令的执行需要五个时钟周期，每个时钟周期上升沿来临时，此指令所代表的一系列数据和控制信息转移到下一级处理，从起步到满负荷工作再到停止。

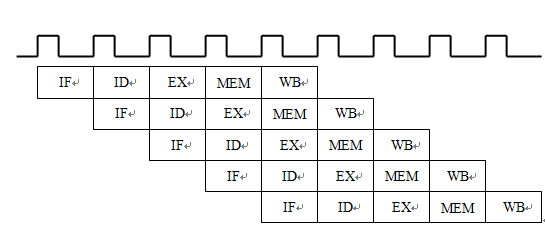


图1 流水线流水作业示意图

一条MIPS指令分为五个处理步骤，即五级流水线，分别为IF（解码）、ID（译码）、EX（执行）、MEM（访存）和WB(写回)。流水线的基本执行过程在理论课我么已经学过，其结构如图2所示。

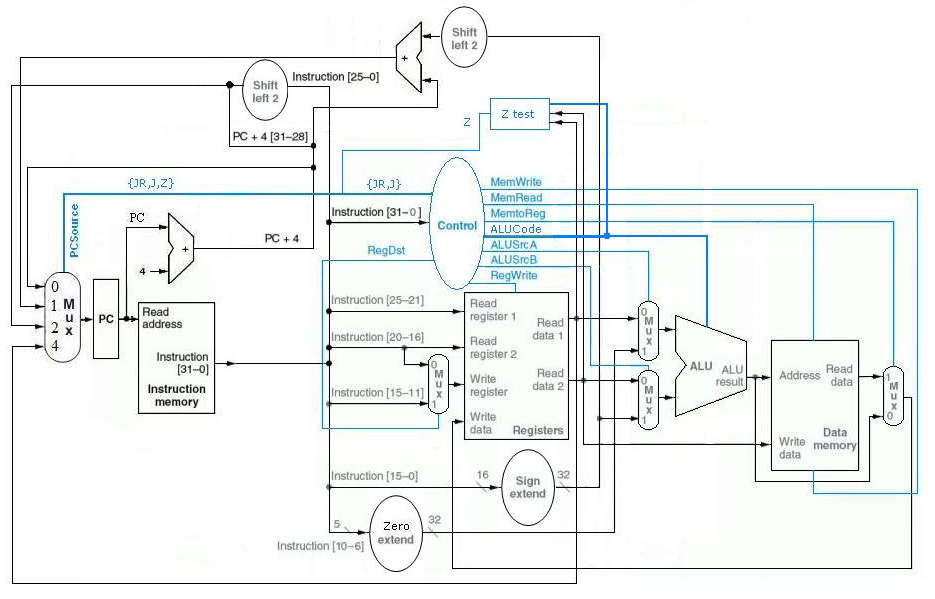


图2 五级流水线MIPS CPU初步原理框图

为了将各级的数据保存和传递，需要在上图中各级分割线处都加入寄存器，加入寄存器后的结构如图3所示

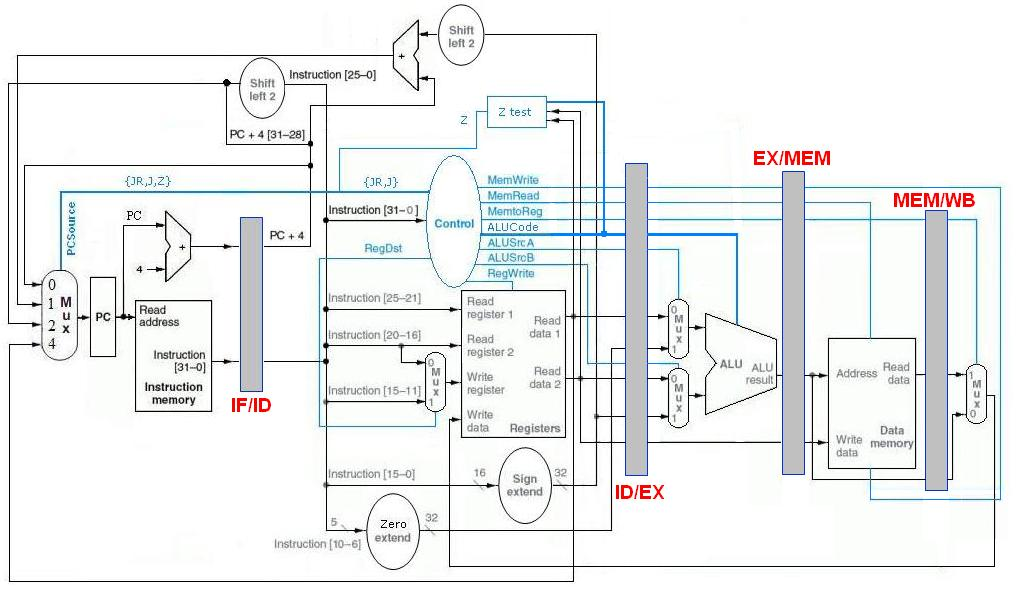
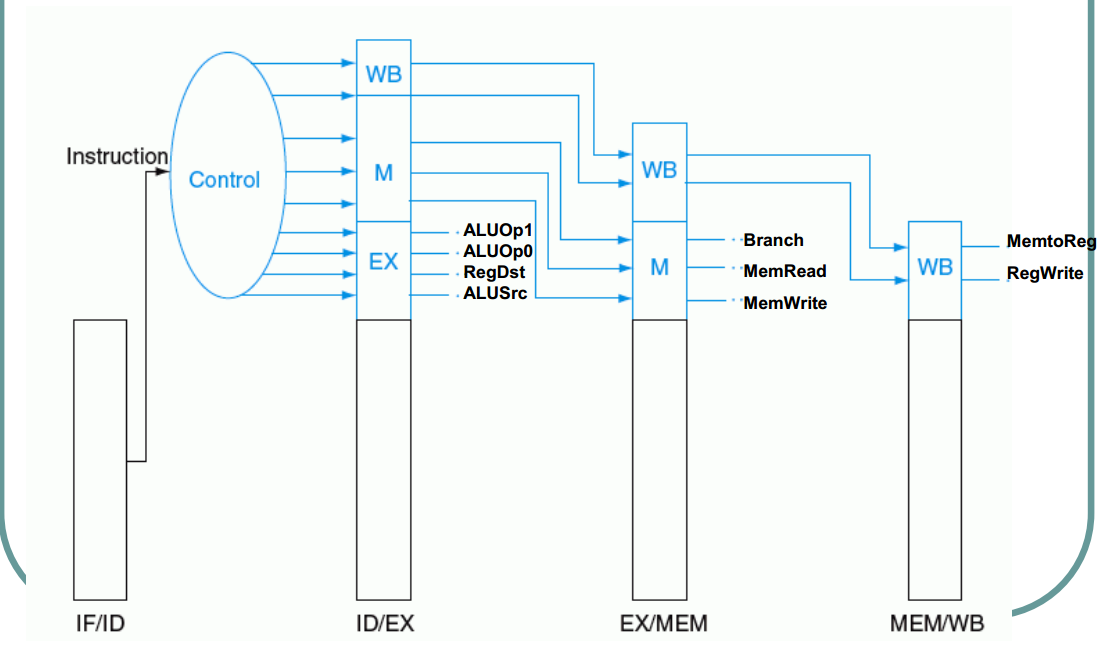


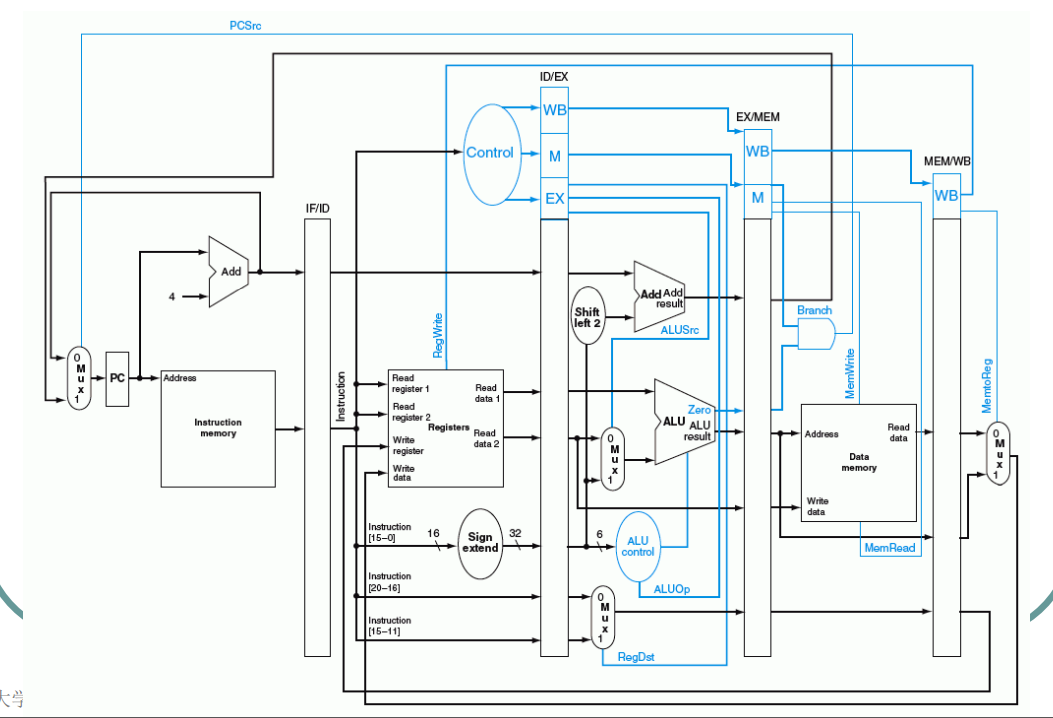
图3.在各级分割线添加寄存器后的框图

然而因为流水线在同时执行很多条指令的不同部分，如果这些指令互相独立，则毫无问题，否则就会出现各种各样的冒险，导致指令无法正确。所以为了成功构建一条有效的流水线，我们必须解决这些冒险问题。

同时，为了保证各部分能够执行正确的操作，需要将指令信息依次向后传递。



所以最终流水线的数据通路是这样的：



### 冒险

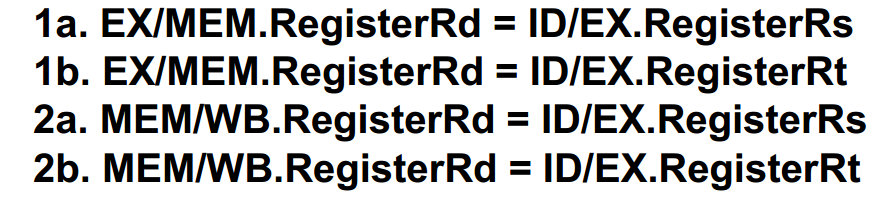
* + 1. 数据冒险与解决措施
       1. 非Load Use冒险
          1. 原因

根据理论课所学，我们知道如果前一条R型指令的目标地址是下一条或者下下一条的操作数就会发生冒险。具体来说，因为指令在MEM阶段结束的下一个周期才会被写入Memory，而此时下一条和下下条的指令操作数已读取完毕（读取操作数的周期为ID）。

* + - * 1. 判定

根据数据冒险发生的原因，我们很容易知道该冒险发生的条件：

冒险关系



指令类型检测——只转发写寄存器的指令

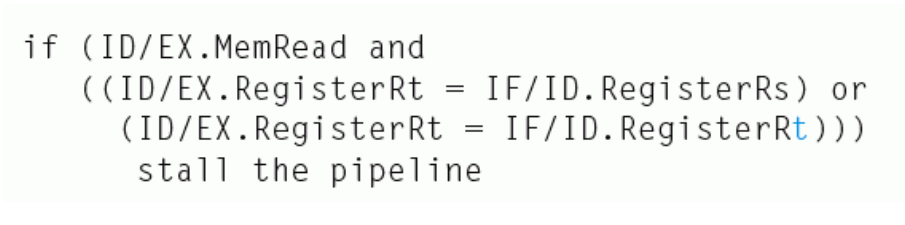
这种检测只需查看RegWrite信号是否有效，即检测流水线寄存器**EX**和**MEM**阶段的**WB**控制字段以确定**RegWrite**信号是否置位。

目标寄存器不是零·寄存器

* + - * 1. 解决方法——转发
      1. Load Use冒险
         1. 原因

当一条**load**指令紧跟一条需要读取其结果的指令时，数据转发无法解决问题，此时，必须采取措施阻塞流水线

* + - * 1. 判定

` 

* + - * 1. 解决方法——阻塞+转发

转发我们刚才已经进行说明了，这里对如何阻塞进行简要说明。

阻止指令继续执行的方式是保持PC和IF/ID寄存器不变——引入PCWrite和IF/IDWrite信号，同时EX段后半部分的流水线执行nop指令——将所有控制信号变为0

* + 1. 分支冒险与解决措施
       1. 判定

遇到了branch型指令

* + - 1. 解决方法——提前分支

将分支指令提早执行，即将分支加法器从EX段移到ID段。如果分支成立，则清除IF段中的指令，为此引入IFFlush信号。

* + 1. 中断/异常与解决措施
       1. 原因

在处理器正常执行程序过程中，如果出现异常的指令，或者必须处理的外部中断，处理器就会停止当前的任务来执行相应的处理程序，此时必须清除当前指令和已经处于ID、IF阶段的指令。

* + - 1. 解决措施
         1. 普通指令

ID阶段的地址PC（不是PC+4）写入寄存器，再用IDFlush和IFFlush清除指令。

* + - * 1. J指令

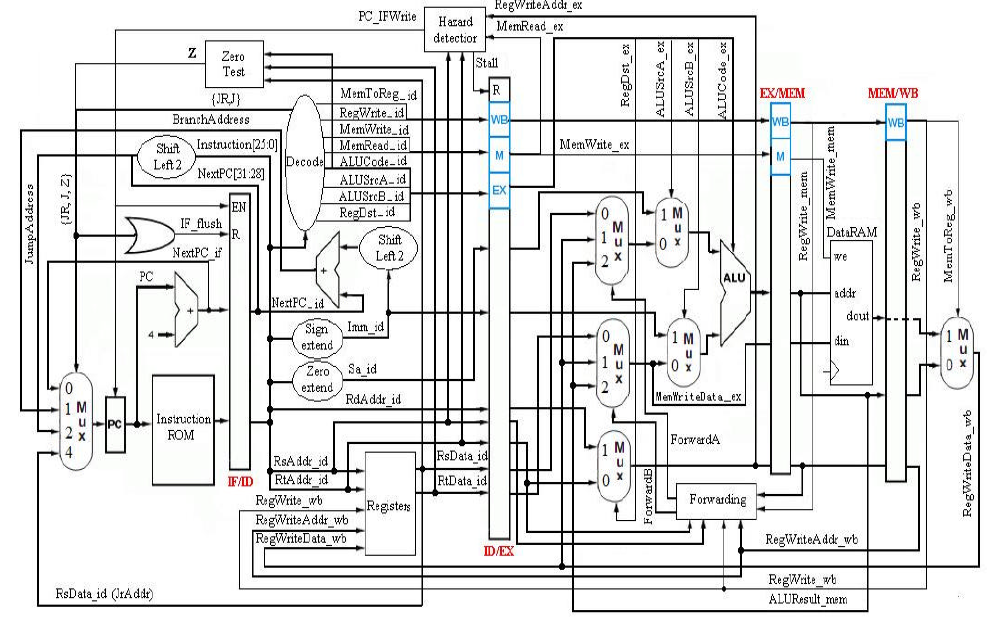
中断发生在J型指令后一条指令，需要将J型指令的地址（而不是下一条地址）写入寄存器，即IF\_flush = 1时，IF/ID寄存器的PC输出保持原输出一周期。

* + - * 1. Branch指令

中断发生在branch型指令后一条指令，需要将branch指令的地址写入寄存器，即ID\_flush = 1时，ID/EX寄存器PC输出赋值为EX/MEM的寄存器输出

### 最终设计框架

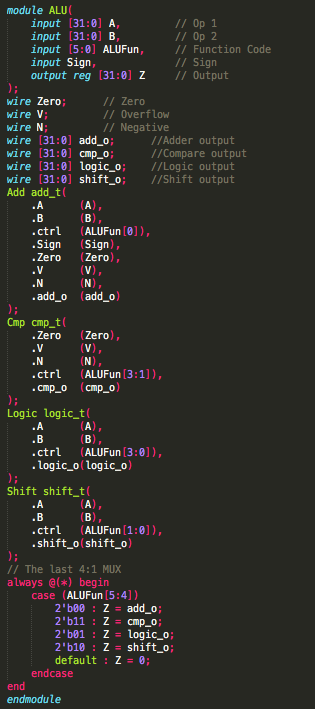
在解决了以上的各种冒险之后，一个无冒险的CPU框架如下图



# 关键代码分析及文件清单

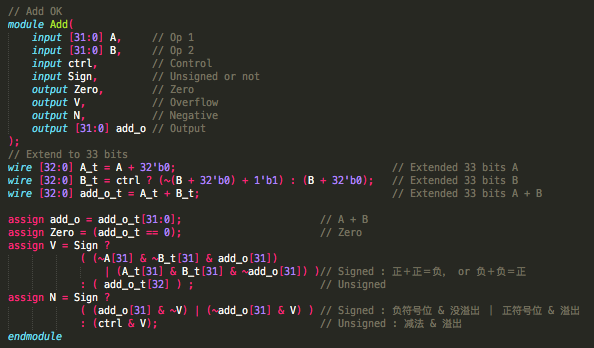
## ALU部分（林子恒）

### 顶层模块：

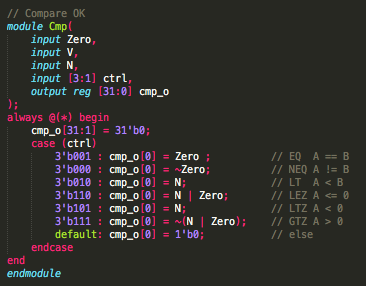


按照前述设计图，分别实现Add，Cmp，Logic，Shift四个模块，然后最后使用一个多路选择器奖结果输出。

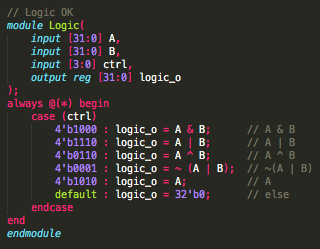
### Add模块：



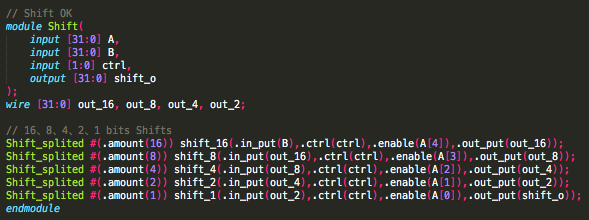
### Cmp模块：



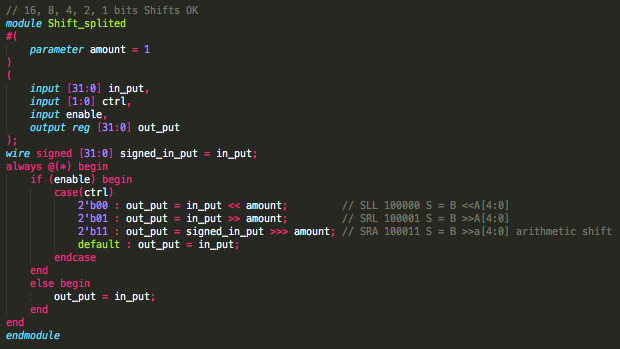
### Logic模块：



### Shift模块：



### 16，8，4，2，1 Shift模块



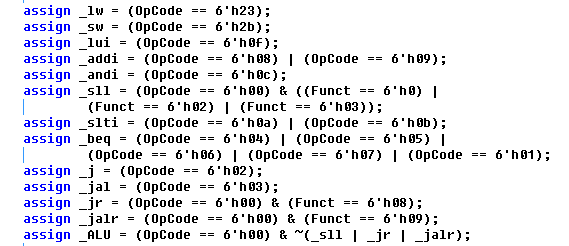
## 单周期部分（何子昊）

### Control模块，代码文件为Control.v

首先根据理论课上所学，列出对于每个命令与相应信号的关系，需要注意的是增加了int(interrupt)和exp(exception)信号，虽然二者不是汇编命令，但是会覆盖当前执行命令产生中断。前者为定时器中断，将返回地址PC+4置于$k0中，PC变为0x80000004执行中断服务程序；后者遇到了未知命令，PC进入0x80000008进入死循环。

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **PCSrc[2:0]** | **RegDst[1:0]** | **RegWr** | **ALUSrc1** | **ALUSrc2** | **ALUFun[5:0]** | **Sign** | **MemWr** | **MemRd** | **MemtoReg[1:0]** | **ExtOp** | **LuOp** |
| **lw** | 0 | 1 | 1 | 0 | 1 | 00 | X | 0 | 1 | 1 | 1 | 0 |
| **sw** | 0 | X | 0 | 0 | 1 | 00 | X | 1 | 0 | X | 1 | 0 |
| **lui** | 0 | 1 | 1 | 0 | 1 | 00 | X | 0 | X | 0 | X | 1 |
| **add** | 0 | 0 | 1 | 0 | 0 | 00 | X | 0 | X | 0 | X | X |
| **addu** | 0 | 0 | 1 | 0 | 0 | 00 | X | 0 | X | 0 | X | X |
| **sub** | 0 | 0 | 1 | 0 | 0 | 01 | X | 0 | X | 0 | X | X |
| **subu** | 0 | 0 | 1 | 0 | 0 | 01 | X | 0 | X | 0 | X | X |
| **addi** | 0 | 1 | 1 | 0 | 1 | 00 | X | 0 | X | 0 | 1 | 0 |
| **addiu** | 0 | 1 | 1 | 0 | 1 | 00 | X | 0 | X | 0 | 1 | 0 |
| **and** | 0 | 0 | 1 | 0 | 0 | 18 | X | 0 | X | 0 | X | X |
| **or** | 0 | 0 | 1 | 0 | 0 | 1E | X | 0 | X | 0 | X | X |
| **xor** | 0 | 0 | 1 | 0 | 0 | 16 | X | 0 | X | 0 | X | X |
| **nor** | 0 | 0 | 1 | 0 | 0 | 11 | X | 0 | X | 0 | X | X |
| **andi** | 0 | 1 | 1 | 0 | 1 | 18 | X | 0 | X | 0 | 0 | 0 |
| **sll** | 0 | 0 | 1 | 1 | 0 | 20 | X | 0 | X | 0 | X | X |
| **srl** | 0 | 0 | 1 | 1 | 0 | 21 | X | 0 | X | 0 | X | X |
| **sra** | 0 | 0 | 1 | 1 | 0 | 23 | X | 0 | X | 0 | X | X |
| **slt** | 0 | 0 | 1 | 0 | 0 | 35 | 1 | 0 | X | 0 | X | X |
| **slti** | 0 | 1 | 1 | 0 | 1 | 35 | 1 | 0 | X | 0 | 1 | 0 |
| **sltiu** | 0 | 1 | 1 | 0 | 1 | 35 | 0 | 0 | X | 0 | 1 | 0 |
| **beq** | 1 | X | 0 | 0 | 0 | 33 | X | 0 | X | X | X | X |
| **bne** | 1 | X | 0 | 0 | 0 | 31 | X | 0 | X | X | X | X |
| **blez** | 1 | X | 0 | 0 | X | 3D | 1 | 0 | X | X | X | X |
| **bgtz** | 1 | X | 0 | 0 | X | 3F | 1 | 0 | X | X | X | X |
| **bltz** | 1 | X | 0 | 0 | X | 3B | 1 | 0 | X | X | X | X |
| **j** | 2 | X | 0 | X | X | X | X | 0 | X | X | X | X |
| **jal** | 2 | 2 | 1 | X | X | X | X | 0 | X | 2 | X | X |
| **jr** | 3 | X | 0 | 0 | X | X | X | 0 | X | X | X | X |
| **jalr** | 3 | 2 | 1 | 0 | X | X | X | 0 | X | 2 | X | X |
| **int** | 4 | 3 | 1 | X | X | X | X | 0 | X | 2 | X | X |
| **exp** | 5 | 3 | 1 | X | X | X | X | 0 | X | 2 | X | X |

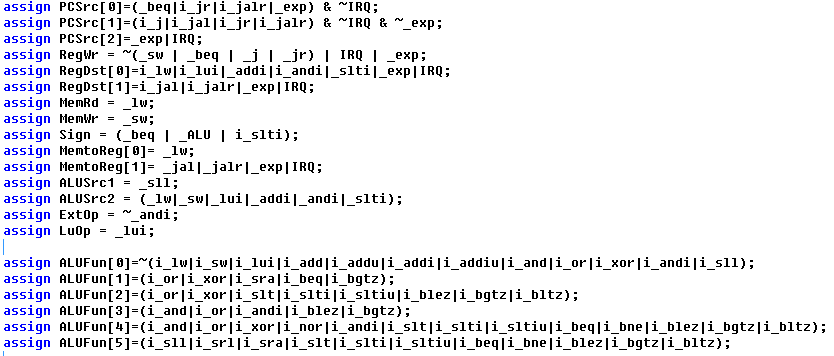
根据上表，可以发现有几条功能相近的指令的控制信号相近，因此用相同的颜色标出，在判断控制信号时可以合并处理。如下图所示。



同时对于ALUFunc基本每条指令都不相同，因此需要表示出每一条指令，如下图，这样避免了对ALUFun信号使用复杂的逻辑表达式，而是精简为二级与或形式以提高时钟频率。

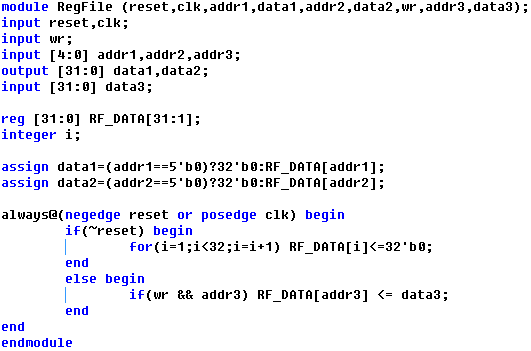


同时考虑到中断和异常的特殊情况，在生成控制信号时要加入int和exp信号。综合上表，可以得出控制信号如下。



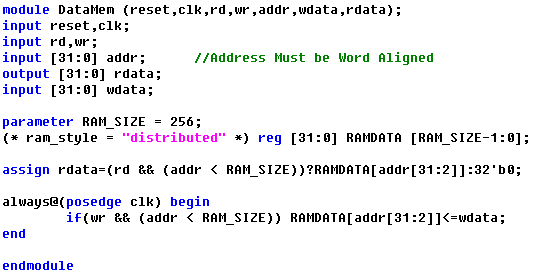
为了减少时间开销，尽量避免了使用多路选择器的?:形式，而是多采用二级或的形式，对于PCSrc、RegDst、MemtoReg、ALUFun的每一位进行设置。

### RegFile模块，代码文件为regfile.v



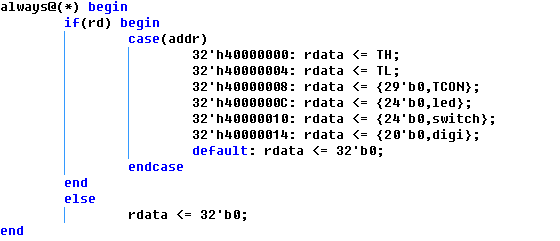
此代码段由助教提供，data1/2直接输出RF\_DATA数据，存数据时只能存入非$zero的寄存器，复位时全部寄存器清零。

### DataMem模块，代码文件为DataMem.v

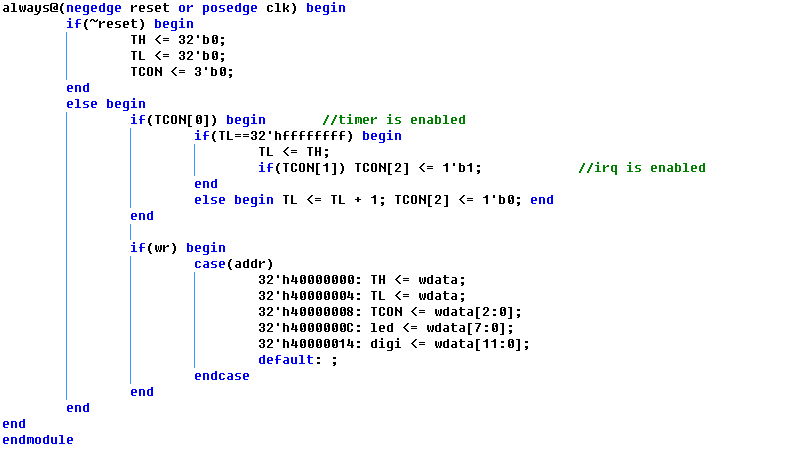


此代码段由助教提供，提供256个字存储，即最高数据地址为0x3FF，超过范围则输出零，wr置位且地址合法时将数据写入相应位置。

### Peripheral模块，代码文件为Peripheral.v

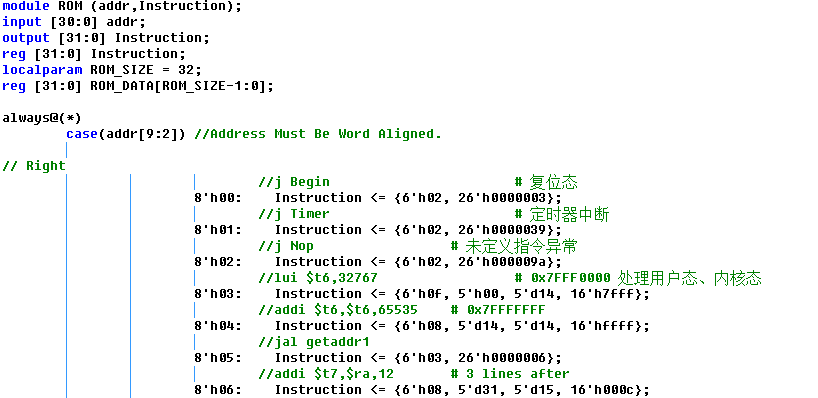


根据指导书的寄存器映射，0x40000000~0x40000017为定时器和led、开关、数码管的地址，读数据即为读出当前设置。长度不足需要补高位。



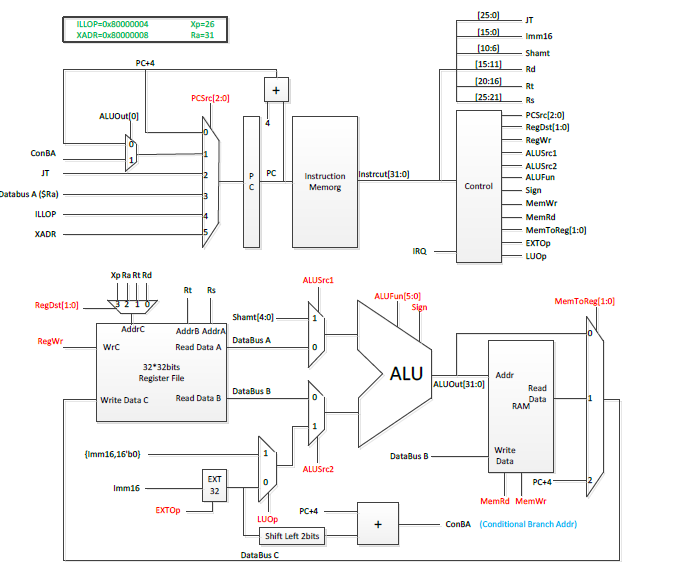
reset复位时寄存器置零，当计数器开始时TL减一，并检查是否溢出以判断是否已达到一个周期，然后重新置数倒计时，在中断允许的情况下将中断位置位。

### Rom模块，代码文件为Rom.v



Rom文件中为汇编程序所对应的二进制码，根据相应地址位读出对应的指令，此文件由编译器从原始汇编源文件gcd.asm自动生成。

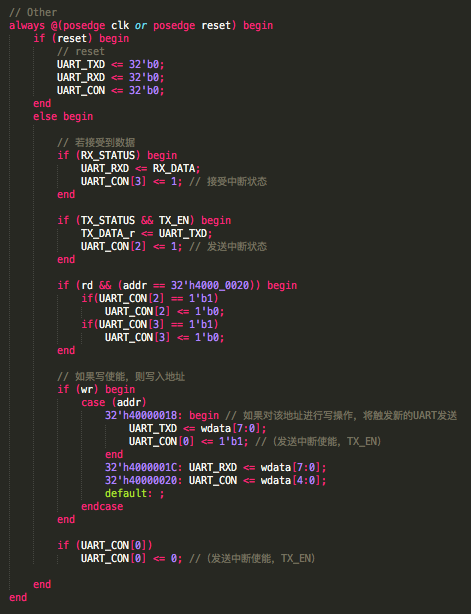
### 单周期模块，代码文件为CPU\_Single.v



单周期模块会按照上述原理图的模块连接方式实例化各子模块并连接。需要注意PC+4的最高位不能改变，Rom地址仅仅决定于低31位，地址最高位仅仅表明了处于用户态还是内核态。EXT32根据ExtOp决定是否进行符号位拓展，发生异常或中断时跳到指定的PC位置。根据读取Memory位置不同从不同模块读入数据。

## UART部分（林子恒）

此部分的接收器与发射器部分代码与第四次试验相同，故不再赘述。与之前作业不一样的部分如下：



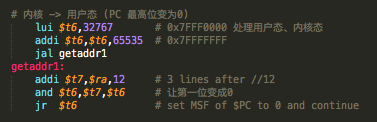
基本按照实验指导书上的思路，采用轮询的方式。在接收到数据之后，将UART\_CON[3]置为1；读取状态之后将UART\_CON[2]，UART\_CON[3]置为0；若对UART\_TXD地址进行写操作，则触发新的UART发送。

## 汇编程序部分（林子恒）

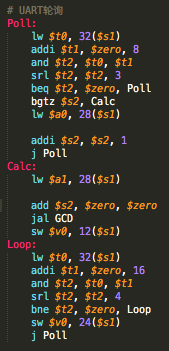
### 开始部分：

Screen%20Shot%202016-09-10%20at%2011.51.44%20AM.png

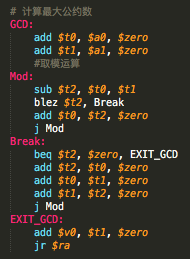
### 将内核态转换为用户态



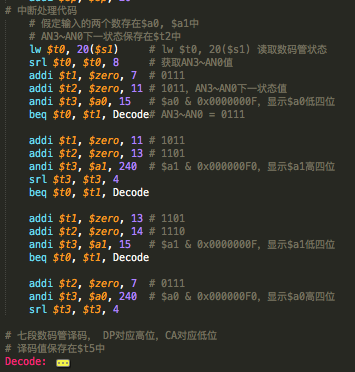
### UART轮询：



### 计算最大公约数程序：

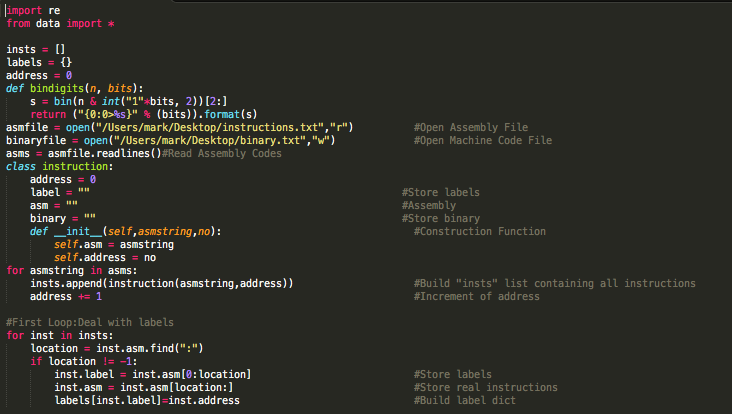


### 中断处理代码（显示数码管）：



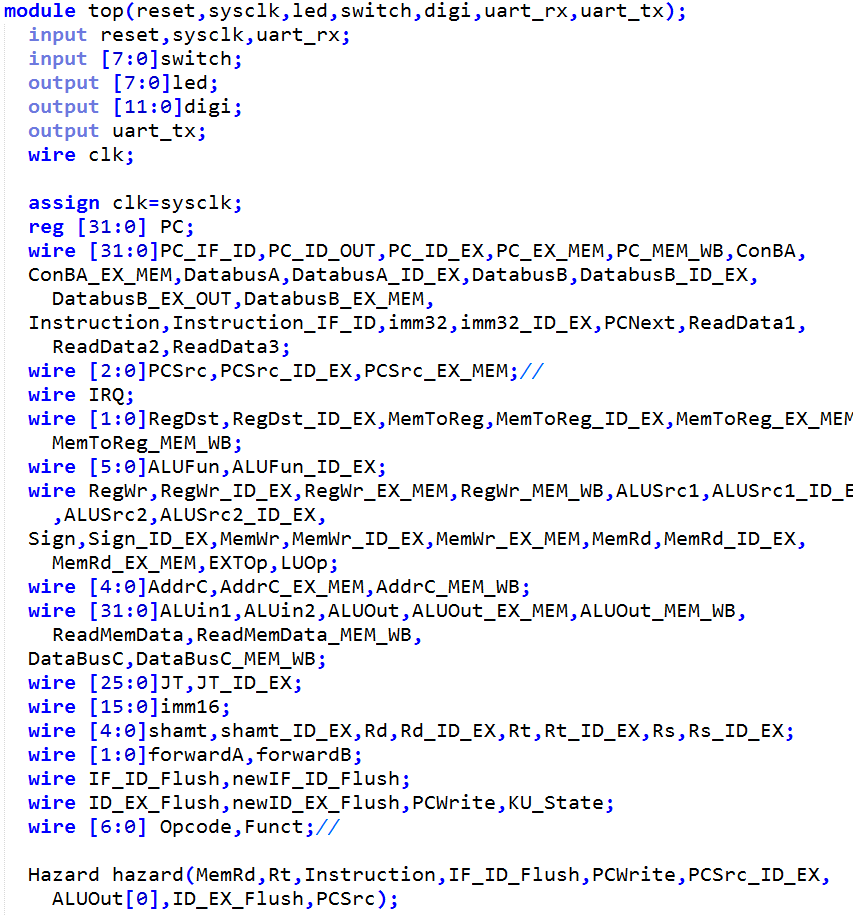
## 编译器部分（范承泽）

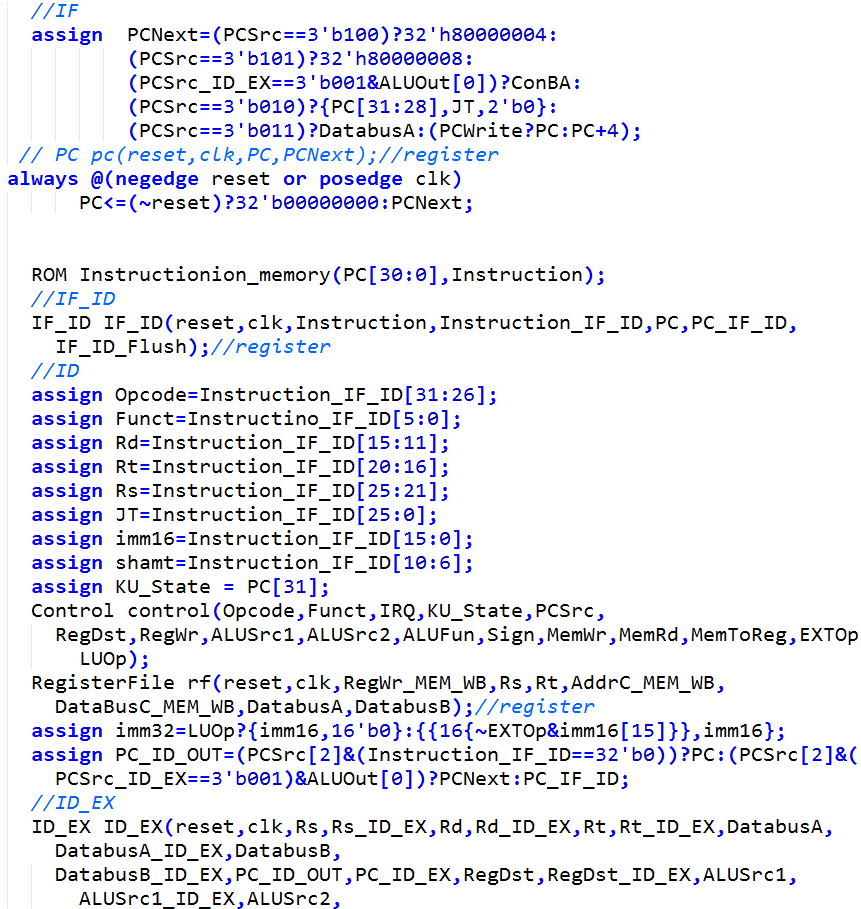
### 主程序：

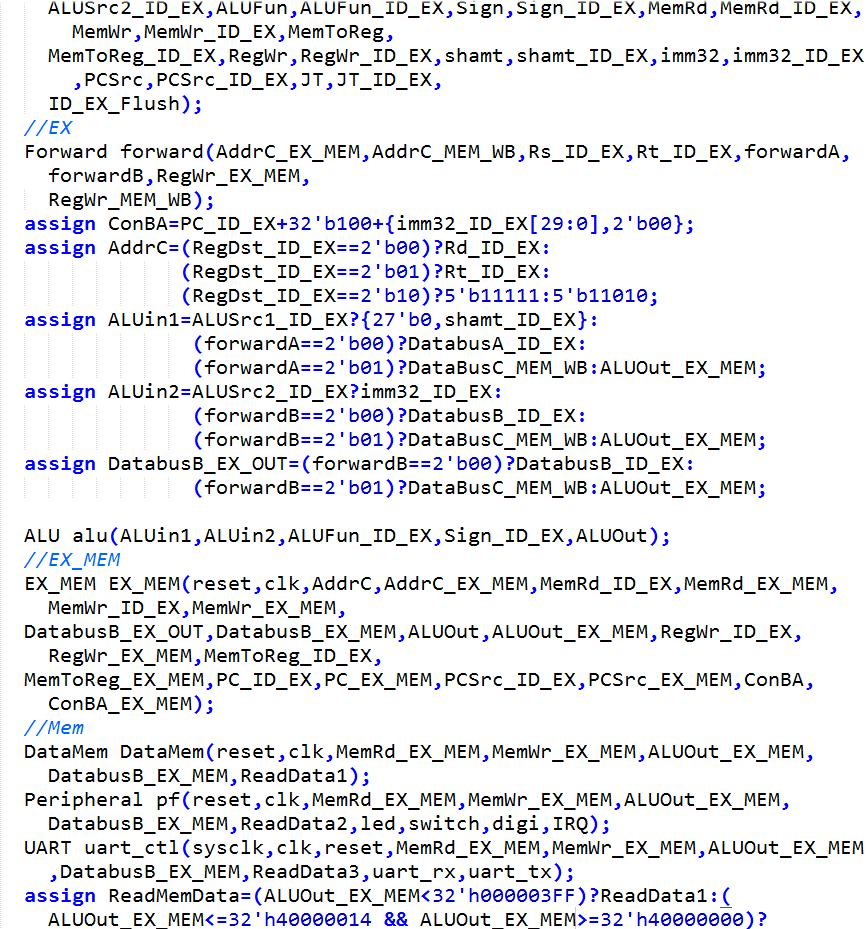


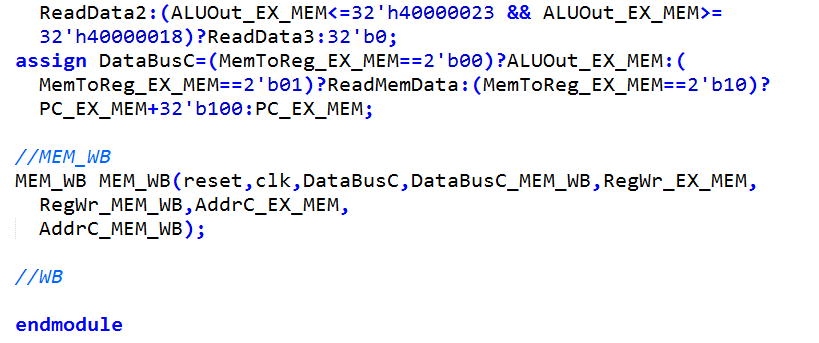
## 流水线部分（范承泽）

### 总体设计（Pipeline.v）





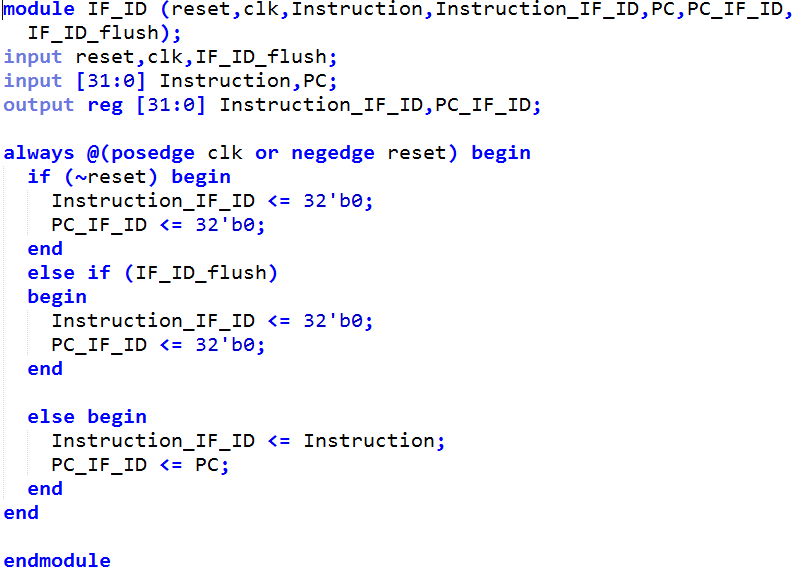




基本是按照CPU的结构从左到右依次将各个部分实例化来设计的。

在这里特别要PC的生成部分，需要根据原理部分一点点分析来决定。

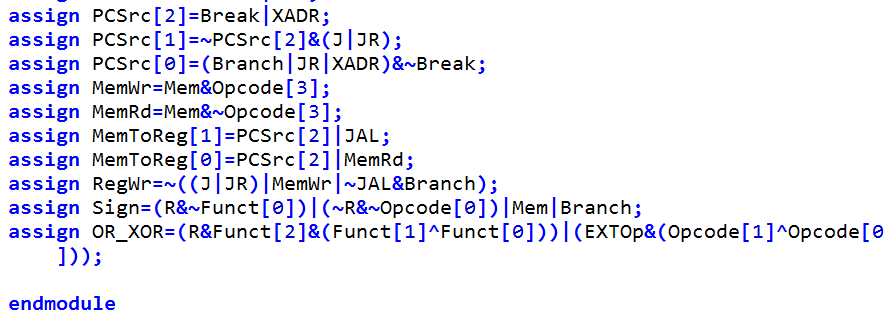
### IF\_ID



按照理论课上的分析结果将需要传递下去的参数传下去即可。注意IFIDFlush的应用。

### Control

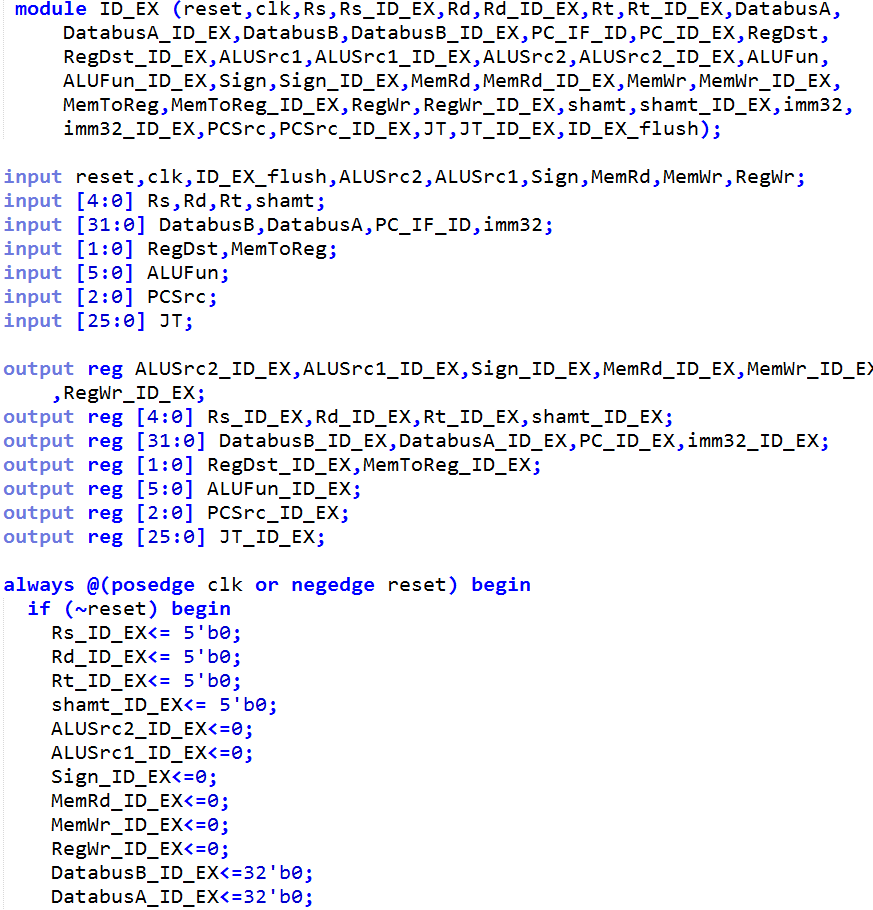


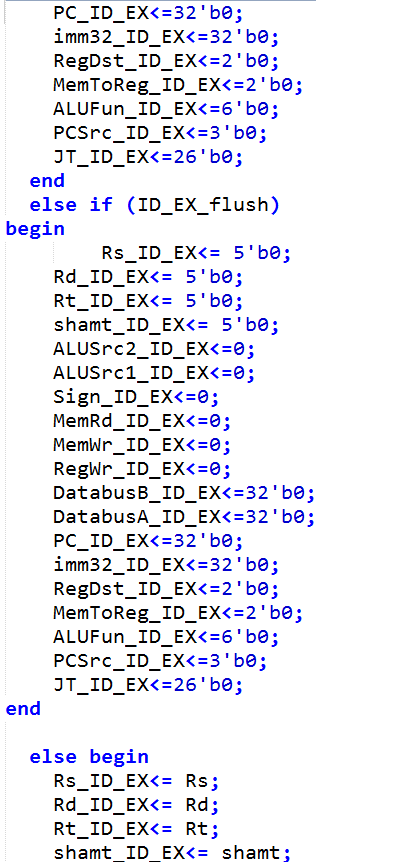


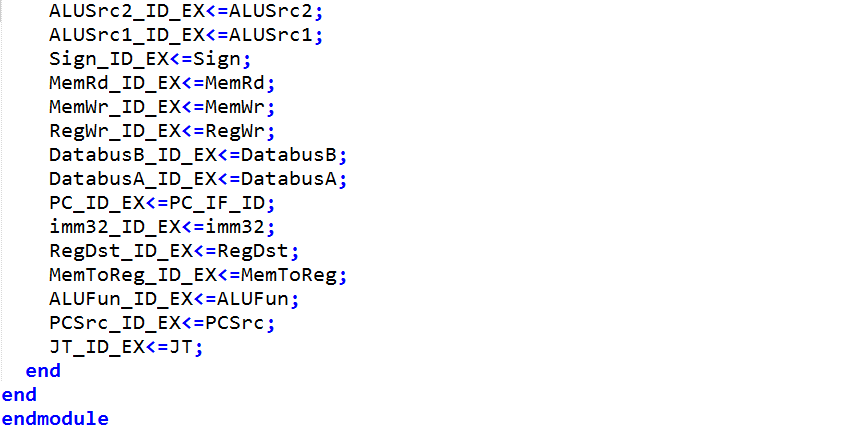
因为Control的信号非常多，所以在这里为了理清思路，我们先还原出各个指令的类型，再根据指令的类型来决定各个信号。具体内容参考了单周期CPU的Control模块实现、原理部分分析、理论课课件以及网上资料。

### ID\_EX

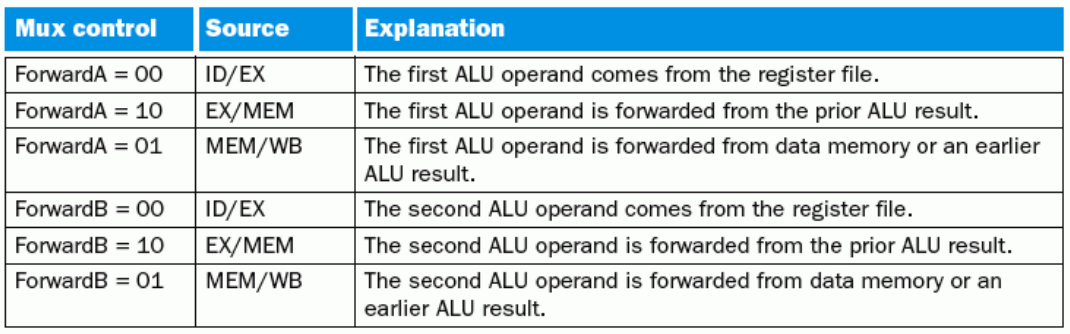
原理同样是传值到EX阶段，和同样IF/ID寄存器也有flush信号，这里不再赘述。







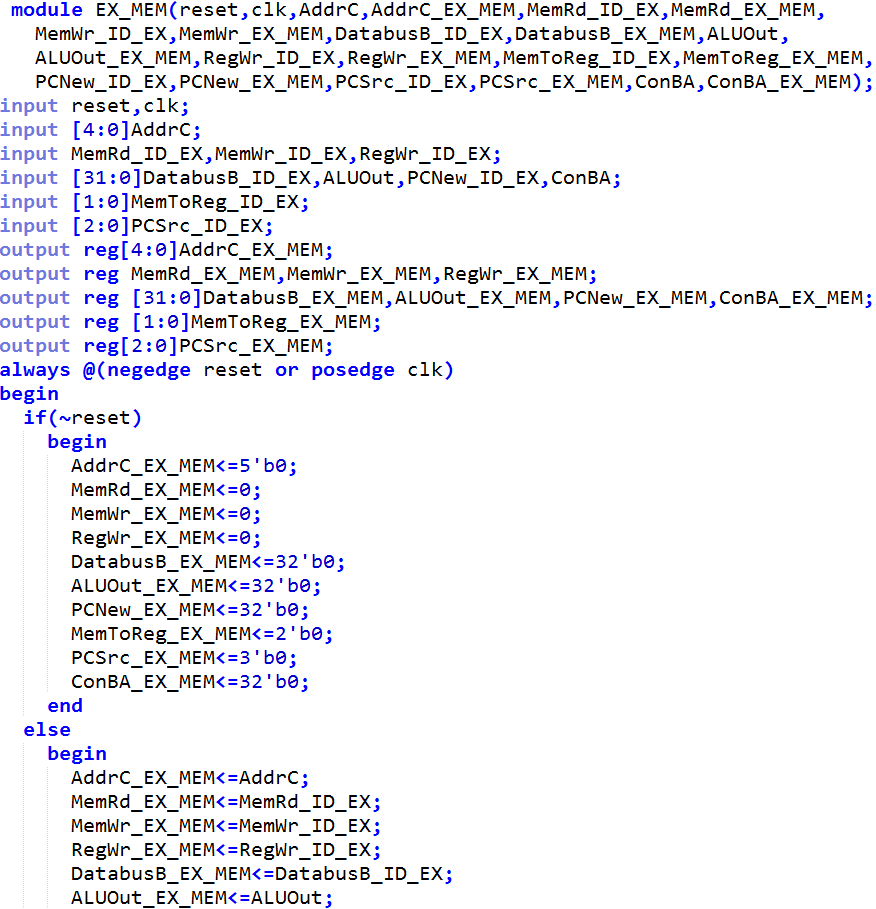
### Forward

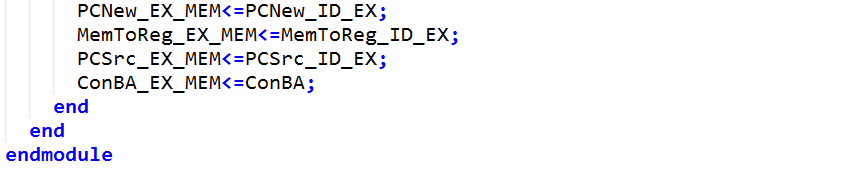


根据上表，我们很容易得到Forward模块输出的信号值。



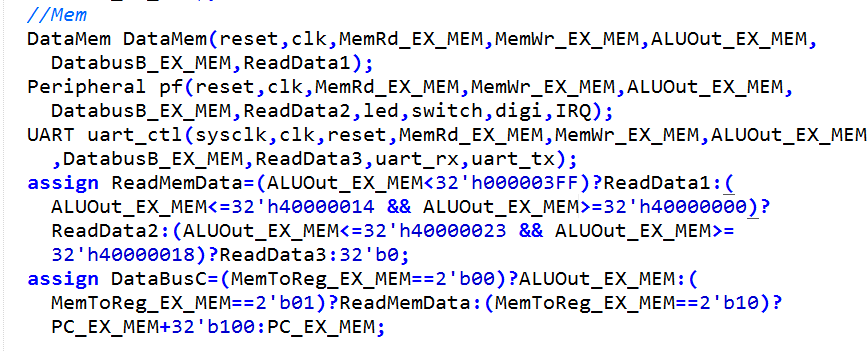
### EX\_MEM



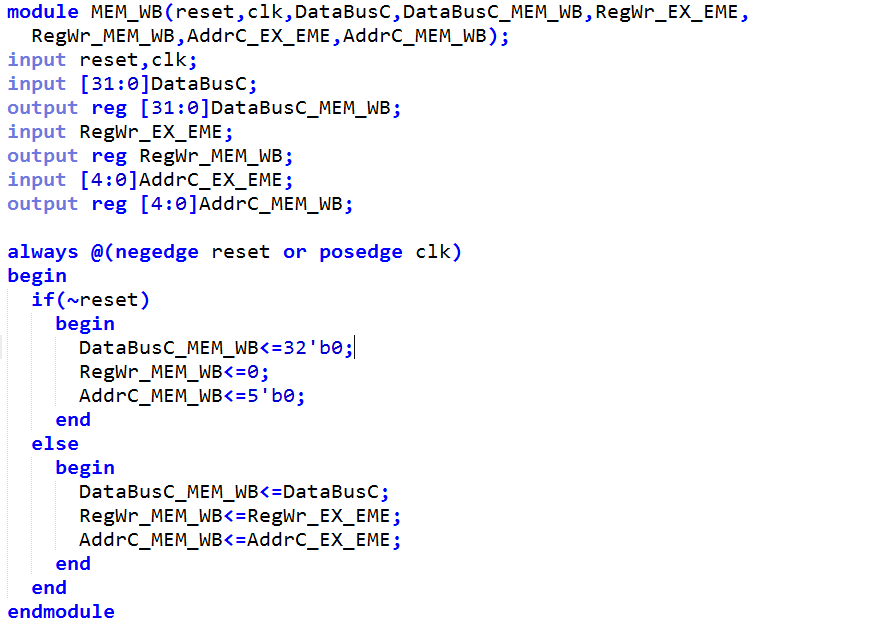


### Memory

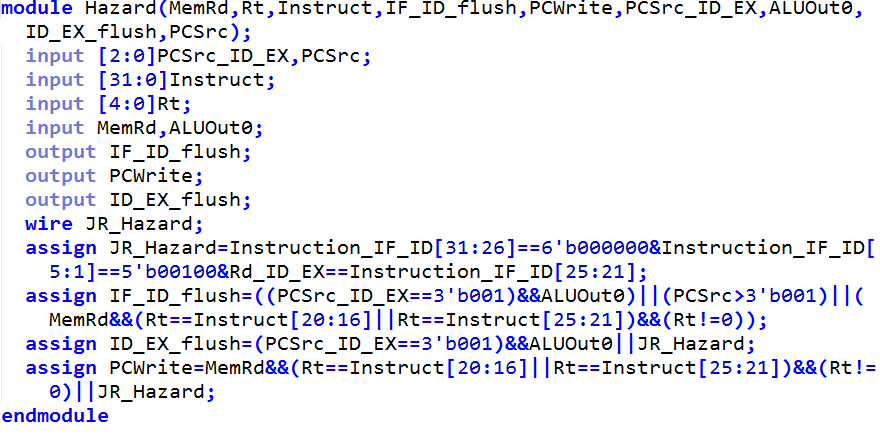
这一部分参考了单周期CPU的实现。



### MEM\_WB



### Harzard

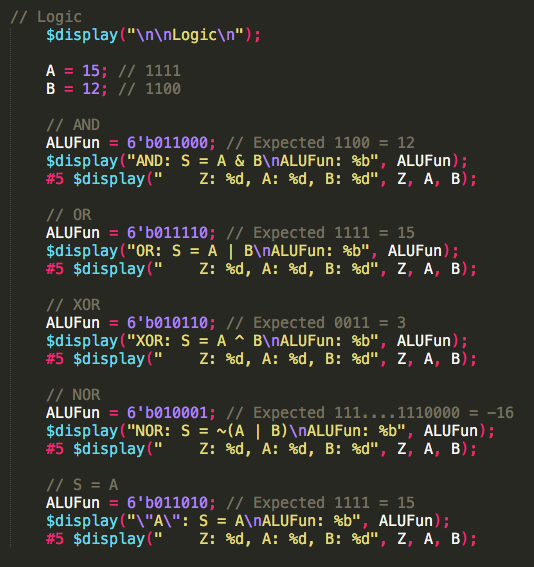
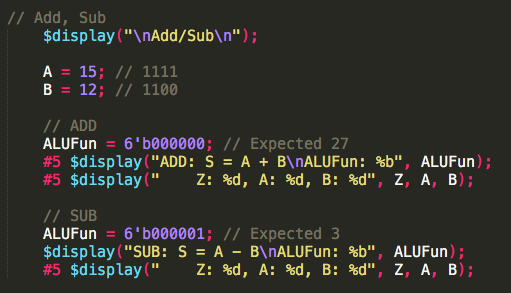


我们根据上面的分析，结合Control信号的实现，来简化Hazard的控制—IF/IDFlush、ID/EXFlush和PCWrite。具体含义见原理部分分析以及Control模块实现。

# 仿真结果及分析

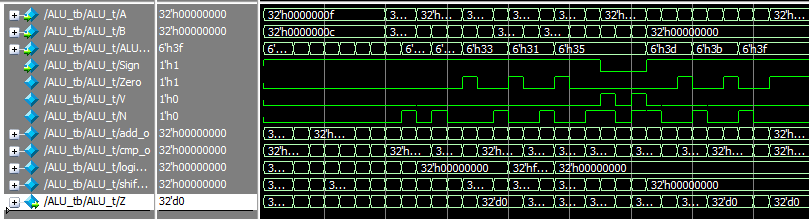
## ALU

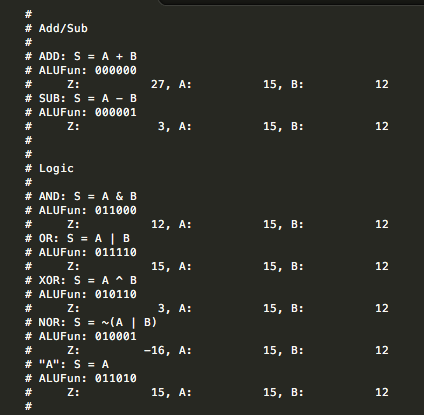
为了测试ALU的功能是否正确实现，编写了如下的测试文件，由于代码冗长，仅列出Add、Logic模块的仿真代码及结果：



仿真结果如下：

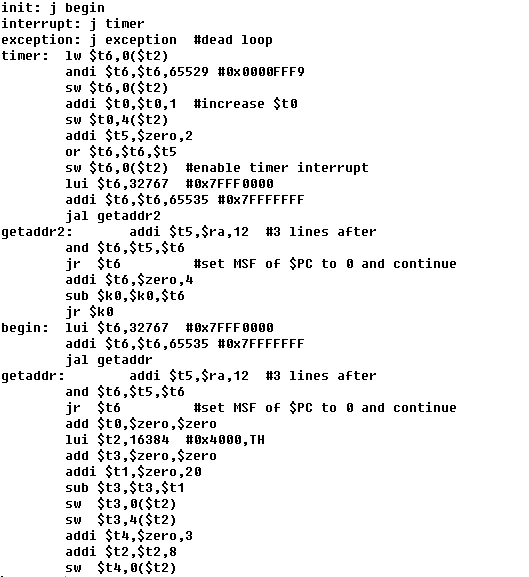
在仿真代码里，我们对ALU输入测试信号，依序测试Add，Logic，Shift，Compare模块。经检验，可以看到，ALU模块完成了正常的工作。



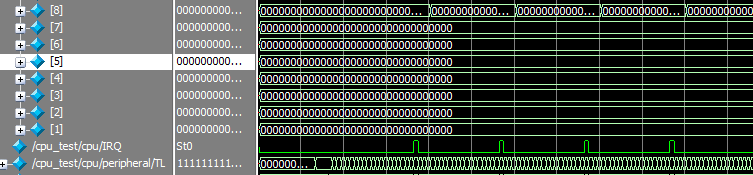


Compare、Shifter模块仿真代码以及输出文字结果见ALU\_test.v、ALU\_test\_result.md

## 单周期：



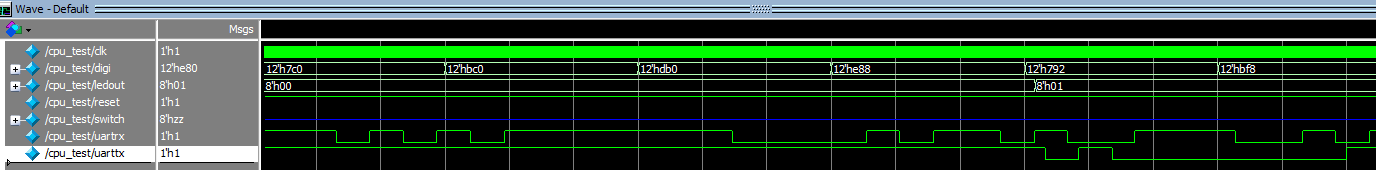
为了初步测试单周期CPU的功能完整性，故写了一个简单的定时增加$t0的汇编代码，其中timer首先去除中断使能位和中断位，然后使$t0加1，然后允许中断使能位，然后要将PC最高位恢复为0处于用户态运行，再跳回到原地址继续运行。在程序复位后会跳到begin处运行，begin首先需要将PC最高位恢复为0处于用户态运行，然后设置TH、TL以设定时钟周期，最后使能定时器。



上图为仿真情况，IRQ为中断指示信号，对应着每次$t0（8号寄存器）自增1，可见计时器功能正常工作。

## 流水线

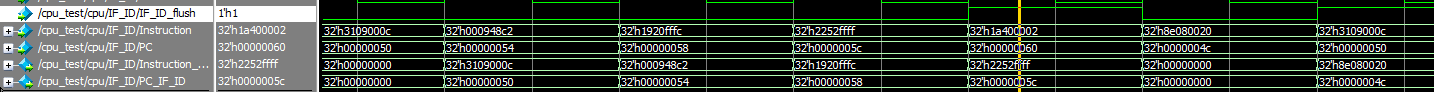
### 总体仿真



我们看到了根据testbench的仿真，我们的CPU能够进行正常的数据收发，同时根据烧到板子上的显示我们也确认了CPU的功能和ROM中的代码的正确性。

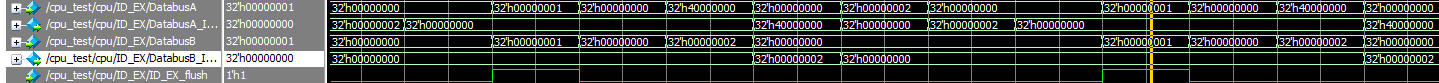
### 其他部分

#### IF\_ID



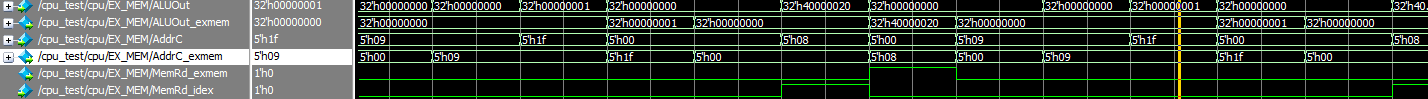
我们可以看到IFIDFlush置位时，输出都为0，其他时候输入输出的对应关系也都正常。

#### ID\_EX



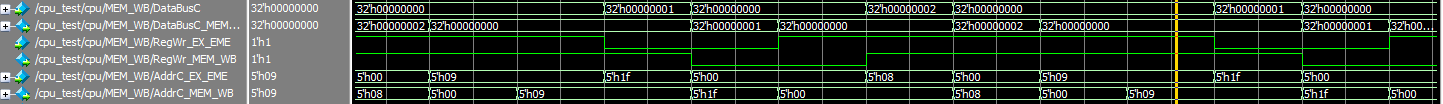
我们可以看到IDEXFlush置位时，输出都为0，其他时候输入输出的对应关系也都正常。

#### EX\_MEM



EX\_MEM寄存器可以正常传递对应的信号。

#### MEM\_WB



MEM\_WB寄存器可以正常传递对应的信号。

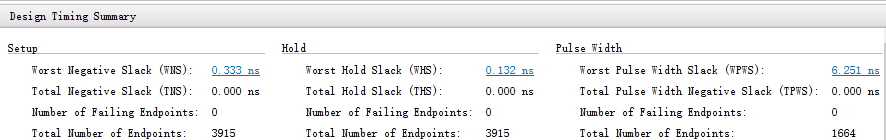
#### Harzard(在总体里面找)

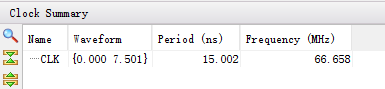


这是一个Load Use类型的Hazard，应该阻塞一个周期，我们看到了Hazard检测模块成功设置了ID\_EXFlush、IF\_IDFlush、PCWrite信号。

# 综合情况

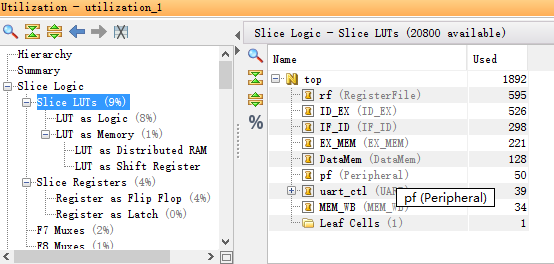
## 时间综合情况





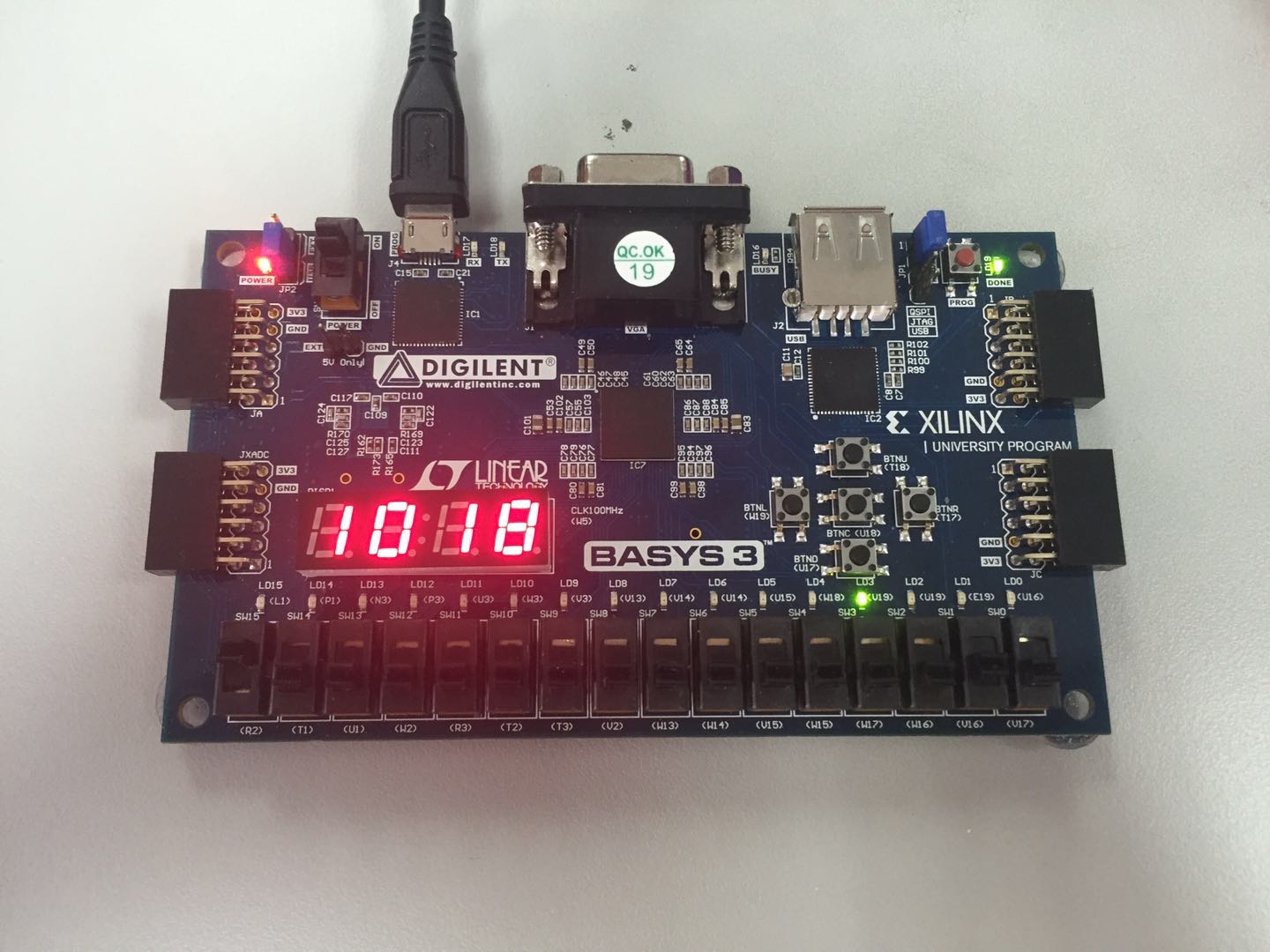
最后流水线的时钟频率为66.658MHz

## 空间综合情况



# 硬件调试情况

使用串口助手传入10,18两16进制数，经CPU计算得到最大公约数为08，串口助手正确回传该数字，在FPGA上数码管及LED灯也正常显示。



# 思想体会

## 林子恒

跟之前所做的四次小实验比起来，这次小学期所写的CPU对我来讲是一个完全不一样的考验。与之前的实验不同的是这次小学期所完成的是一个由非常非常多模块结合而成的大工程，非常考验模块实现的鲁棒性以及团队协作的能力。

一开始在做UART的部分的时候我其实是崩溃的，由于在做第四次试验时，我实现UART接受及发射器的方法没有将系统时钟传入，导致在这次小学期一开始实现UART时产生了非常多的bug，比如明明只接收到了一个信号，CPU确认为已经接收到了两次，这是由于我的RX\_STATUS信号没有与系统时钟对齐的缘故。

总体而言，这次试验使我仿真找bug的技巧有了大幅度的提升，在满屏的绿线中找自己写的代码哪里出了问题是我从来没有过的体验；这次试验也让我充分锻炼了团队协作的技巧，也让我深刻地认识到了唯有充分了解队友所写代码的结构以及实现方法，才能避免走入很多误区。

## 何子昊

相比较于数字逻辑理论课大作业的单周期CPU设计，这次小学期的CPU增加了异常和中断功能，还有用户态和内核态的切换，使我更加深入了解了CPU设计原理以及相应标志位的意义。同时在某些指令中还要考虑到中断和异常的影响，使得中断和异常的信号能够优先被考虑，同时只有用户态才能产生中断，这些设计都有一些小型操作系统的意味。对于外设的支持也比较新颖，不是直接对硬件管脚进行操作，在汇编看来仅仅是向某个内存地址读取或写入数据即可达到操控外设的功能，而这一切都是由CPU作为媒介进行转换，同时CPU对于UART进行了充分的包装，使得汇编语言能够方便快捷的调用已实现的功能。

综上，此次单周期CPU设计让我学到了很多更加真实的设计需求，同时也需要我进行充分的测试以供同组的同学在我的基础上修改为流水线设计，增强了我的团队协作意识。

## 范承泽

这个流水线CPU是我上大学以来完成的最复杂的一个项目，虽然有小伙伴的分工协助，但是从单周期CPU改造成一个无冒险的流水线CPU的难度仍然是超乎想象的。

首先，这个复杂的工程让我学会了要分模块按照顺序分析、完成项目。同时，要在具体动手设计之前构建好整个系统的结构，尤其是复杂系统更是如此。在第一个版本时，在进行CPU的总体设计时，我按照之前的习惯直接写模块和连接和相应的没有按照空间顺序从左至右设计，因为信号控制逻辑太过复杂，信号传输太过复杂，信号过多而且同一个信号有不同的版本，导致花费了很长时间调bug，但在分层设计之后，信号连接的问题就迎刃而解了。

第二，我磨练了硬件调试的技巧和verilog代码书写技巧。

第三，通过自己手工编写一个流水线CPU，增强了对单周期CPU和流水线CPU的理论，对于数逻理论课上的知识点理解更加深刻了。

虽然开发调试过程很折磨人，但是在历经千辛万苦终于看到一个CPU从自己手中诞生的心情的确是溢于言表的。最后还要感谢老师和助教耐心的解答所有疑问，这对于我能顺利完成流水线的开发起到了至关重要的作用。