

**2021~2022学年 第二学期**

**《系 统 硬 件 综 合 设 计》**

设 计 报 告

班 级 计算机科学与技术20-3班 学 号 2020214296

姓 名 马俊豪 成 绩 \_\_\_\_\_\_\_\_\_\_\_\_\_

2022年11月

**目 录**

目录

[1 设计要求 3](#_Toc121324131)

[2 CPU指令及数据通路 3](#_Toc121324132)

[2.1算术运算指令 3](#_Toc121324133)

[2.1.1 算术运算指令格式 3](#_Toc121324134)

[2.1.2 算术指令运算数据通路 5](#_Toc121324135)

[2.2 访存指令(大端模式) 6](#_Toc121324137)

[2.2.1 访存指令格式 6](#_Toc121324138)

[2.2.2 访存指令数据通路 6](#_Toc121324139)

[2.3 跳转指令 7](#_Toc121324141)

[2.3.1 跳转指令格式 7](#_Toc121324142)

[2.3.2 跳转指令数据通路 8](#_Toc121324143)

[2.4 异常指令 8](#_Toc121324144)

[2.4.1 异常指令格式 8](#_Toc121324145)

[2.4.2 异常指令数据通路 9](#_Toc121324146)

[3 功能部件说明 10](#_Toc121324148)

[3.1 InsRom 10](#_Toc121324149)

[3.2 DatRam 11](#_Toc121324150)

[3.2 Core 12](#_Toc121324151)

[3.2.1 PC 13](#_Toc121324152)

[3.2.2 Registers (相同意义的接口说明不再赘述) 14](#_Toc121324153)

[3.2.3 ID (相同意义的接口说明不再赘述) 15](#_Toc121324154)

[3.2.4 EX (相同意义的接口说明不再赘述) 16](#_Toc121324155)

[3.2.5 MEM (相同意义的接口说明不再赘述) 19](#_Toc121324156)

[3.2.6 DIV除法器 (相同意义的接口说明不再赘述) 20](#_Toc121324157)

[3.2.7 CP0 协处理器 21](#_Toc121324159)

[3.2.8 Ctrl 控制流水线暂停以及刷新 22](#_Toc121324160)

[4 测试指令 23](#_Toc121324161)

[5 仿真波形 24](#_Toc121324162)

[6 生成烧写文件 30](#_Toc121324163)

[6.1 准备约束文件 30](#_Toc121324164)

[6.2 Synthesis 31](#_Toc121324165)

[6.2 Implementation 32](#_Toc121324166)

[6.3 Generate Bitstream 33](#_Toc121324167)

[参考文献 35](#_Toc121324168)

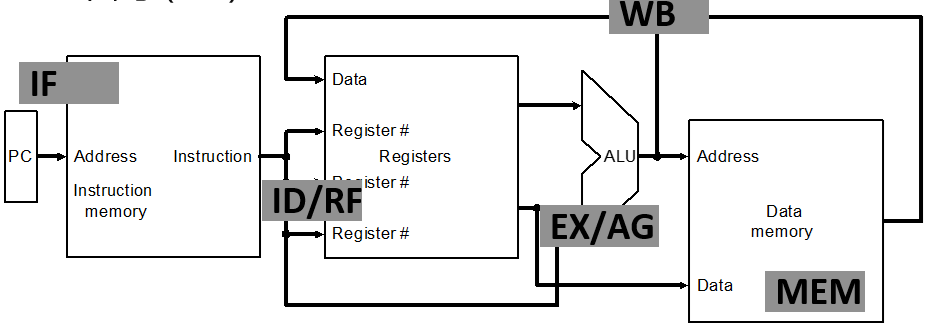
[总结 35](#_Toc121324169)

# 1 设计要求

基于先修课程，根据系统设计思想，使用硬件描述语言设计实现一款基于MIPS32指令集的微处理器（CPU）。具有5级流水线并实现了外部中断异常功能，并将设计的CPU下载至FPGA开发板（ego-1）上运行。以此贯穿数字逻辑、计算机组成原理、计算机体系结构课程，实现从逻辑门至完整CPU处理器的设计。进一步加深了对于计算机科学底层知识的了解。

# 2 CPU指令及数据通路

对于一条指令的处理，五级流水线要经过以下阶段：1.取值(IF)、2.译码(ID)、3.执行(EX)、4.访存(MEM)、5.回写(WB)，基本结构如下。



**图1 指令的典型处理过程**

## 2.1算术运算指令

注：rt、rs代表操作数寄存器，rd代表目标寄存器。

### 2.1.1 算术运算指令格式

31 26 25 21 20 16 15 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ori 001101 | rs | rt | immediate | 立即数或运算 |
| andi 001100 | rs | rt | immediate | 立即数与运算 |
| xori 001110 | rs | rt | immediate | 立即数异或运算 |
| addi 001000 | rs | rt | immediate | 立即数加 |
| addiu 001001 | rs | rt | immediate | 无符号立即数加 |
| slti 001010 | rs | rt | immediate | 有符号立即数比较 |
| sltiu 001011 | rs | rt | immediate | 无符号立即数比较 |

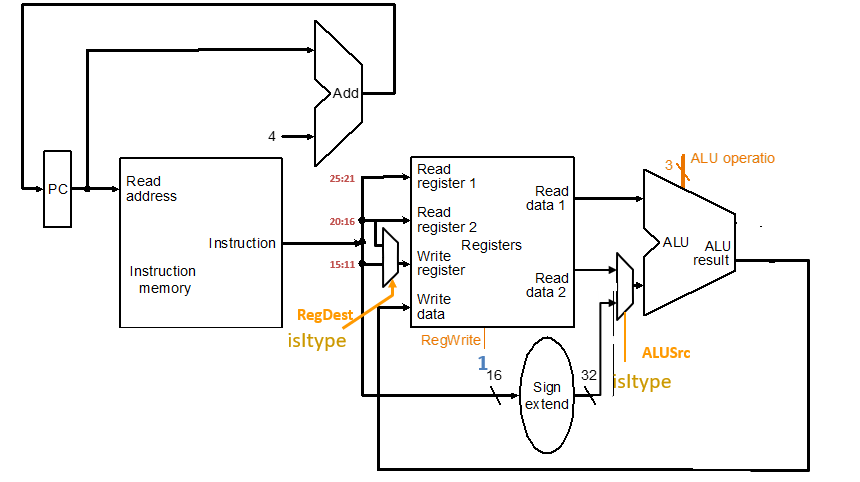
**表1 立即数运算指令格式**

31 26 25 21 20 16 15 11 10 6 5 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 000000 | rs | rt | rd | 00000 | and 100100 | 寄存器与运算 |
| 000000 | rs | rt | rd | 00000 | or 100101 | 寄存器或运算 |
| 000000 | rs | rt | rd | 00000 | xor 100110 | 寄存器异或运算 |
| 000000 | rs | rt | rd | 00000 | nor 100111 | 寄存器同或运算 |
| 000000 | 00000 | rt | rd | sa | sll 000000 | 逻辑左移 |
| 000000 | 00000 | rt | rd | sa | srl 000010 | 逻辑右移 |
| 000000 | 00000 | rt | rd | sa | sra 000011 | 算术右移 |
| 000000 | rs | rt | rd | 00000 | sllv 000100 | 逻辑左移(寄存器) |
| 000000 | rs | rt | rd | 00000 | srlv 000110 | 逻辑右移(寄存器) |
| 000000 | rs | rt | rd | 00000 | srav 000111 | 算术右移(寄存器) |
| 000000 | 00000 | 00000 | 00000 | 00001 | sync 001111 | 空指令 |
| 000000 | rs | rt | rd | 00000 | add 100000 | 寄存器加运算 |
| 000000 | rs | rt | rd | 00000 | addu 100001 | 无符号加运算 |
| 000000 | rs | rt | rd | 00000 | sub 100010 | 寄存器减运算 |
| 000000 | rs | rt | rd | 00000 | subu 100011 | 无符号减运算 |
| 000000 | rs | rt | rd | 00000 | slt 101010 | 有符号比较 |
| 000000 | rs | rt | rd | 00000 | sltu 101011 | 无符号数比较 |
| 011100 | rs | rt | rd | 00000 | mult 011000 | 有符号乘运算 |
| 000000 | rs | rt | 00000 | 00000 | multu 011001 | 无符号乘运算 |
| 011100 | rs | rt | 00000 | 00000 | madd 000000 | 有符号乘累加 |
| 011100 | rs | rt | 00000 | 00000 | maddu 000001 | 无符号乘累加 |
| 011100 | rs | rt | 00000 | 00000 | msub 000100 | 有符号乘累减 |
| 011100 | rs | rt | 00000 | 00000 | msubu 000101 | 无符号乘累减 |
| 000000 | rs | rt | 00000 | 00000 | div 011010 | 有符号除 |
| 000000 | rs | rt | 00000 | 00000 | divu 011011 | 无符号除 |
| 000000 | rs | rt | rd | 00000 | movn 001011 | If rt!=0 rd<-rs |
| 000000 | rs | rt | rd | 00000 | movz 001010 | If rt==0 rd<-rs |
| 000000 | 00000 | 00000 | rd | 00000 | mfhi 010000 | rd<-hi |
| 000000 | 00000 | 00000 | rd | 00000 | mflo 010010 | rd<-lo |
| 000000 | rs | 00000 | 00000 | 00000 | mthi 010001 | hi<-rs |
| 000000 | rs | 00000 | 00000 | 00000 | mtlo 010011 | lo<-rs |

**表2 寄存器运算指令格式**

### 2.1.2 算术指令运算数据通路



**图2 算术指令运算数据通路**

# 在译码阶段获取指令后，将取得运算的操作数1、操作数2以及最终运算的目的寄存器，并给出运算类型。这些信息将通过寄存器传递到下一阶段执行，运算结束后再通过寄存器传入访存阶段，访存阶段不再对操作数进行操作而是转入回写阶段进行回写寄存器。数据转移指令实际上也是没有进行实际运算的算术运算，因此可用同一条数据通路。

## 2.2 访存指令(大端模式)

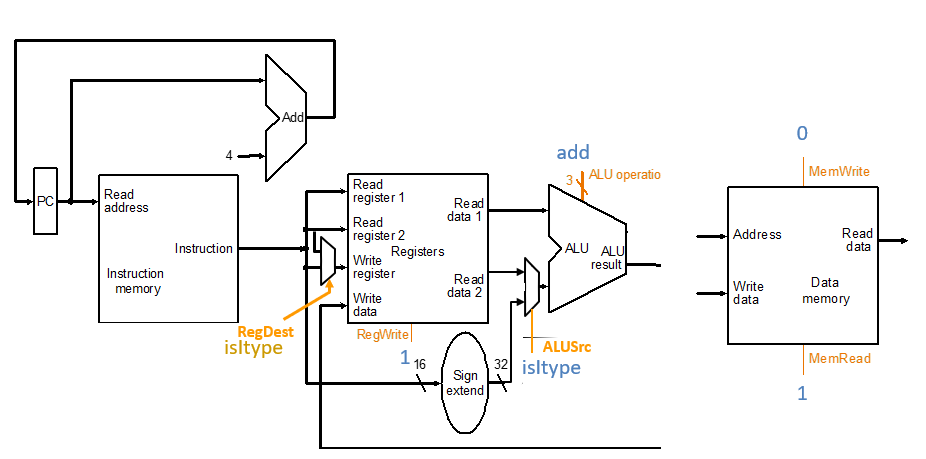
### 2.2.1 访存指令格式

31 26 25 21 20 16 15 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| lb 100000 | base | rt | offset | 加载一个字节符号扩展 |
| lbu 100100 | base | rt | offset | 加载一个字节无符号扩展 |
| lw 100011 | base | rt | offset | 加载一个存储字 |
| sb 101000 | base | rt | offset | 存储一个字节 |
| sw 101011 | base | rt | offset | 存储一个存储字 |

**表3 访存指令格式**

### 2.2.2 访存指令数据通路



**图3 访存指令数据通路**

# 指令将在译码阶段给出操作类型。对于加载指令而言，不再有源操作数而只有目的寄存器，每次读取都是获取一个存储单元的存储字，再根据地址的末两位决定加载哪一个字节，如果是加载字则直接加载整个存储单元的数据。对于存储指令而言，将会获取源操作数并存入内存的指定位置。访存地址将在执行阶段计算得出并传入访存阶段。

## 2.3 跳转指令

### 2.3.1 跳转指令格式

31 26 25 21 20 16 15 11 10 6 5 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 000000 | rs | 00000 | 00000 | 00000 | jr 001000 | PC<-rs |
| 000010 | instruction\_index | | | | | PC<-{(PC+4),des[25:0],2’b00} |

**表4 无条件跳转指令格式**

31 26 25 21 20 16 15 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| beq 000100 | rs | rt | offset | If rs==rt branch |
| bgtz 000111 | rs | 00000 | offset | If rs>0 branch |
| blez 000110 | rs | 00000 | offset | If rs<=0 branch |
| bne 000101 | rs | rt | offset | If rs!=rt branch |

**表5 有条件跳转指令格式**

31 26 25 21 20 16 15 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 000001 | rs | bltz 00000 | offset | If rs<0 branch |
| 000001 | rs | bgez 00001 | offset | If rs>=0 branch |

**表6 有条件跳转指令格式**

### 2.3.2 跳转指令数据通路

**图4 跳转指令数据通路**

在本设计中，取指阶段取到的指令传递给译码阶段。译码阶段得到指令后将直接计算目标地址(因此没有设计过于复杂的跳转指令)，并且在下一阶段将目标地址和是否是跳转指令的信号回传给取指阶段。此时它还会标记下一条指令为延迟槽指令，以便于判断(如果确认跳转则延迟槽指令的写入不会执行)以及中断的实现。

## 2.4 异常指令

### 2.4.1 异常指令格式

31 26 25 21 20 16 15 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ll 110000 | base | rt | offset | rt<-offset(base) Llbit=1 |
| sc 111000 | base | rt | offset | If Llbit=1 rt->offset(base) |

**表7 原子操作指令格式**

31 26 25 6 5 0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 000000 | code | | syscall 001100 | 系统调用，从用户态切换至内核态 |
| 010000 | 1 | 24’h000000 | eret 011000 | 从异常处理函数返回 |

**表8 异常指令格式**

31 26 25 21 20 16 15 11 10 3 2 0

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 010000 | mf 00100 | rt | rd | 8’h00 | sel | CP0[rd]<-rt |
| 010000 | mt 00000 | rt | rd | 8’h00 | sel | CP0[rd]->rt |

**表9 协处理器访问指令格式**

### 2.4.2 异常指令数据通路

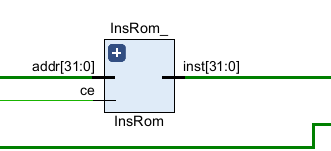
# 

**图5 异常指令数据通路**

在本设计中，主要支持以下异常：syscall系统调用、eret异常返回、8个外部中断（其中包含6个硬件中断以及2个软件中断，在后面烧录fpga实现特定功能时会进行使用）、无效指令异常。协处理器CP0包含对各个异常进行处理与记录的寄存器。为了确保精确异常，所以要保证在发生异常的指令之前的所有指令都要被执行，因此总是在访存阶段处理该条指令异常（此时上一条指令正在回写，而以前的指令回写阶段也都已经结束），CP0处理器中会记录所有的异常，在访存阶段冲刷流水线，记录PC+4（用于处理异常后返回），并将PC的值改为异常处理程序的地址，异常处理结束后调用eret返回。对于原子指令而言，其作用就相当于x86中的sti和cli，对于原子操作，本设计并不保证它一定是原子性的，但只在它确实是原子的运行了时才会生效。执行sc指令时，会对从ll开始的指令序列开始进行检查，判断是否受到干扰，也就是判断Llbit是否为1，如果是说明没有受到干扰，操作是原子的。反之就设置一个通用寄存器的值为0表示操作失败。

# 3 功能部件说明

## 3.1 InsRom



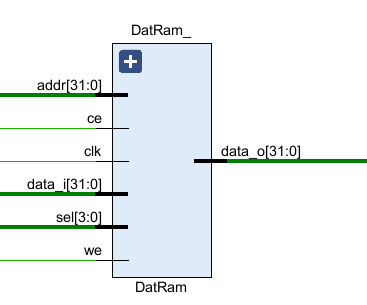
**图6 指令存储器**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| addr | 32 | 输入 | 要取的指令地址 |
| ce | 1 | 输入 | 指令存储器使能信号 |
| inst | 32 | 输出 | 输出的指令 |

**表10 指令存储器接口说明**

对于指令存储器而言，它的每个存储单元32bit，也就是4个字节。对于CPU部分而言PC每次确定下一条指令的地址时都是PC+4。因此，在真正取出指令时对应的存储字的地址是addr[31:2]（左移两位）。

## 3.2 DatRam



**图7 数据存储器**

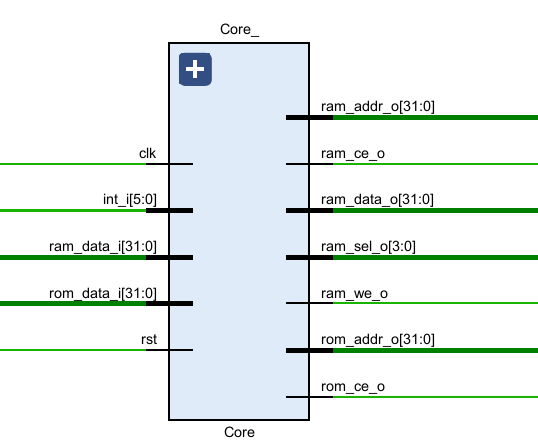
|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| addr | 32 | 输入 | 访问的数据内存单元的地址 |
| ce | 1 | 输入 | 数据存储器使能信号 |
| clk | 1 | 输入 | 时钟信号 |
| data\_i | 32 | 输入 | 要写入的数据 |
| sel | 4 | 输入 | 选择信号，选择1个32bit存储字的哪些部分 |
| we | 1 | 输入 | 数据存储器写使能信号 |
| data\_o | 32 | 输出 | 对于访存指令数据存储器输出的值 |

**表11 数据存储器接口说明**

对于数据存储器它的每个存储单元的存储字长度也为32bit，因此对于输入的地址而言，也是首先将它左移两位确定数据存储器存储单元的地址，再根据sel信号进行判断选择哪一个字节，例如sel=4’b1111代表将整个存储字读出，而当

sel=4’b0001时代表仅将最低字节读出，其它位用0来进行填充。

## 3.2 Core



**图8 CPU核心**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| int\_i | 6 | 输入 | 6个外部中断 |
| ram\_data\_i | 32 | 输入 | 从数据存储器取得的数据 |
| rom\_data\_i | 32 | 输入 | 从指令存储器取得的指令 |
| clk | 1 | 输入 | 时钟信号 |
| rst | 1 | 输入 | 复位信号 |
| ram\_addr\_o | 32 | 输出 | 要访问的数据地址 |
| ram\_ce\_o | 1 | 输出 | 数据存储器使能信号 |
| ram\_data\_o | 32 | 输出 | 要向数据存储器写入的数据 |
| ram\_sel\_o | 4 | 输出 | 数据存储器选择信号 |
| ram\_we\_o | 1 | 输出 | 数据存储器写使能信号 |
| rom\_addr\_o | 32 | 输出 | 要取的指令地址 |
| rom\_ce\_o | 1 | 输出 | 指令存储器使能信号 |

**表12 CPU核心接口说明**

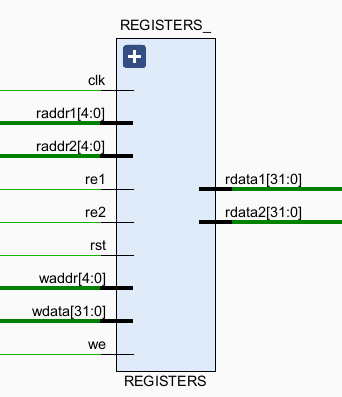
### 3.2.1 PC

**图9 取指令部件**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| clk | 1 | 输入 | 时钟信号 |
| rst | 1 | 输入 | 复位信号 |
| flush | 1 | 输入 | 刷新部件信号 |
| branch\_flag\_i | 1 | 输入 | 分支跳转信号 |
| branch\_target\_address\_i | 32 | 输入 | 分支跳转目标地址 |
| new\_pc | 32 | 输入 | 异常返回后要执行的指令 |
| stall | 6 | 输入 | 流水线停顿信号 |
| ce | 1 | 输出 | 指令存储器使能信号 |
| pc | 32 | 输出 | 输出的指令地址 |

**表13 取指令部件接口说明**

### 3.2.2 Registers (相同意义的接口说明不再赘述)



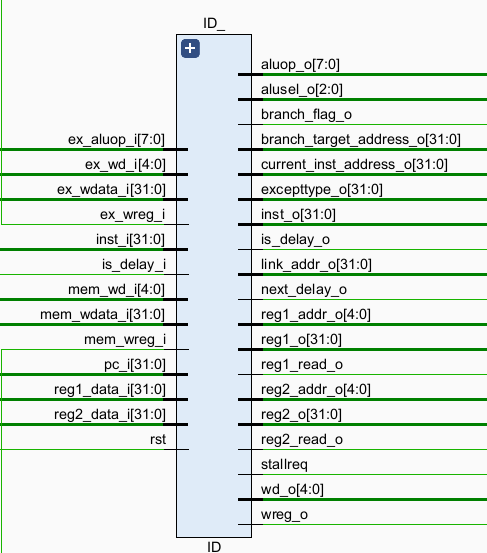
**图10 寄存器堆**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| re1 | 1 | 输入 | 读端口1使能信号 |
| raddr1 | 5 | 输入 | 读端口1要读的寄存器地址 |
| raddr2 | 5 | 输入 | 读端口2要读的寄存器地址 |
| re2 | 1 | 输入 | 读端口2使能信号 |
| waddr | 5 | 输入 | 写端口要写入的寄存器地址 |
| wdata | 32 | 输入 | 写端口要写入的数据 |
| we | 1 | 输入 | 写使能信号 |
| rdata1 | 32 | 输出 | 读端口1读出的数据 |
| rdata2 | 32 | 输出 | 读端口2读出的数据 |

**表14 寄存器堆部件接口说明**

在该部件中实现了数据前推以解决RAW冲突问题。判断waddr==raddr1或者waddr==raddr2，也就是说如果回写阶段写入的寄存器恰好是正要读出的寄存器，则直接将回写的数据读出。

### 3.2.3 ID (相同意义的接口说明不再赘述)



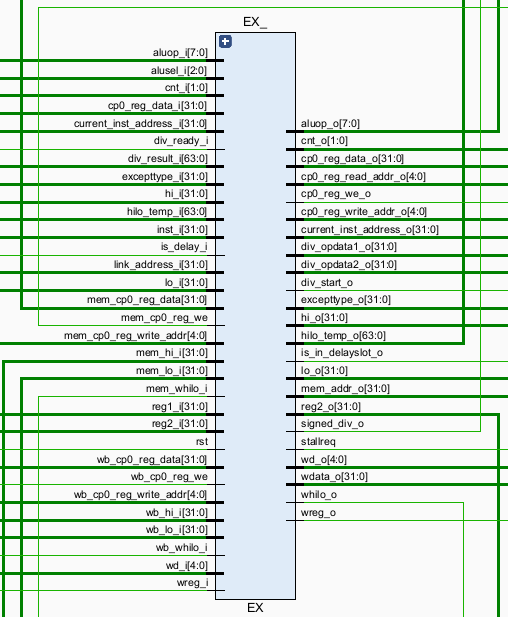
**图11 译码部件**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| ex\_aluop\_i | 8 | 输入 | 执行操作类型 |
| ex\_wd\_i | 5 | 输入 | 流水线执行阶段要写入的目的寄存器地址 |
| ex\_wdata\_i | 32 | 输入 | 流水线执行要写入目的寄存器的数据 |
| is\_delay\_i | 1 | 输入 | 当前指令是否是分支指令的下一条指令 |
| mem\_wd\_i | 5 | 输入 | 流水线访存阶段要写入目的寄存器的地址 |
| mem\_wdata\_i | 32 | 输入 | 流水线访存阶段要写入目的寄存器的数据 |
| aluop\_o | 8 | 输出 | 译码指令的操作子类型(具体的加减乘除) |
| alusel\_o | 3 | 输出 | 译码指令的操作类型(比如是算术运算) |
| current\_inst\_address\_o | 32 | 输出 | 当前指令的地址 |
| link\_addr\_o | 32 | 输出 | 该接口保留使用 |
| next\_delay\_o | 1 | 输出 | 下一条指令是否是分支跳转指令的下一条指令 |
| wd\_o | 1 | 输出 | 本条指令要写入的目的寄存器地址 |
| excepttype\_o | 32 | 输出 | 本阶段出现的异常，传递给下一个阶段 |

**表14 译码部件接口说明**

注意到在译码阶段有三个关于要写入目的寄存器地址和数据的接口，其中两个其实是为了解决RAW冲突。首先判断ex\_wd\_i是否与要读出的寄存器地址一致，即判断流水线执行阶段的冲突，再判断mem\_wd\_i是否与要读出的寄存器地址一致，即判断流水线访存阶段的冲突。而流水线回写阶段的冲突已经在寄存器堆部件处解决。至此，数据相关问题利用数据前推解决完毕。

### 3.2.4 EX (相同意义的接口说明不再赘述)



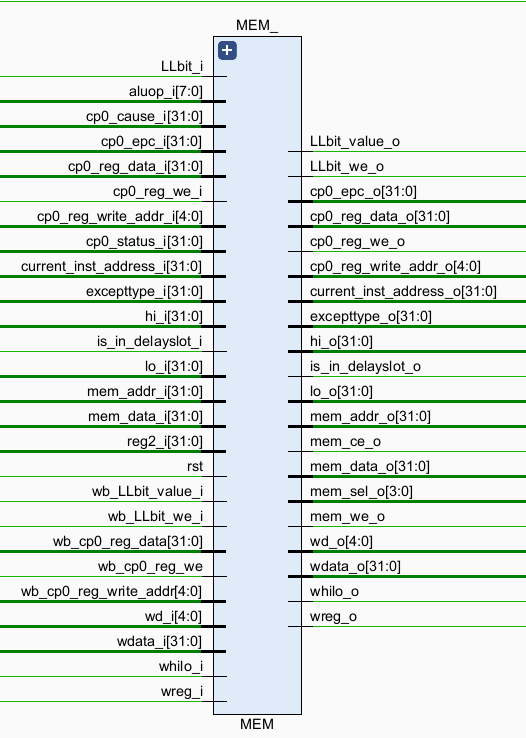
**图12 执行部件**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| cnt\_i | 2 | 输入 | 用于特殊多阶段指令，判断当前是第几个阶段 |
| cp0\_reg\_data\_i | 32 | 输入 | 协处理器中某个寄存器的值 |
| div\_ready\_i | 1 | 输入 | 除法指令执行是否完毕(一共32个时钟周期) |
| div\_result\_i | 32 | 输入 | 除法指令运算结果 |
| excepttype\_i | 32 | 输入 | 上一个阶段发生的异常 |
| hi\_i | 32 | 输入 | Hi寄存器的值，用于乘累加/减或保存余数 |
| lo\_i | 32 | 输入 | Lo寄存器的值，用于乘累加/减或保存商 |
| hilo\_temp\_i | 64 | 输入 | 乘累加/减第一个乘法阶段的值 |
| mem\_hi\_i | 32 | 输入 | 访存阶段要写入hi寄存器的值 |
| mem\_lo\_i | 32 | 输入 | 访存阶段要写入lo寄存器的值 |
| wb\_hi\_i | 32 | 输入 | 回写阶段要写入hi寄存器的值 |
| wb\_lo\_i | 32 | 输入 | 回写阶段要写入lo寄存器的值 |
| cp0\_reg\_data\_o | 32 | 输出 | 要写入协处理器的值 |
| div\_opdata1\_o | 32 | 输出 | 被除数 |
| div\_opdata2\_o | 32 | 输出 | 除数 |
| div\_start\_o | 32 | 输出 | 除法运算开始信号 |
| signed\_div\_o | 1 | 输出 | 是否是有符号除法运算 |
| whilo\_o | 1 | 输出 | 执行阶段的指令是否要写hi lo寄存器 |
| cp0\_reg\_read\_addr | 5 | 输出 | 要读取的协处理寄存器的地址 |

**表15 执行部件接口说明**

不同于其他指令在译码阶段获取操作数，关于协处理的指令是在执行阶段获取操作数，在流水线中获取操作数的操作总会遇到RAW的问题。此处还是采用数据前推的方法，注意到接口mem\_cp0\_write\_addr和wb\_cp0\_reg\_write\_addr，一旦流水线中访存阶段或者回写阶段要写入的协处理器地址与执行阶段要读取的值相同，则直接进行读取访存阶段或者回写阶段要写入的值。

### 3.2.5 MEM (相同意义的接口说明不再赘述)



**图13 访存部件**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| LLbit\_i | 1 | 输入 | 当前LLbit\_i寄存器的值，用于原子指令 |
| cp0\_cause\_i | 32 | 输入 | 协处理器cause寄存器的值 |
| cp0\_epc\_i | 32 | 输入 | 协处理器epc寄存器的值 |
| cp0\_status\_i | 32 | 输入 | 协处理器status寄存器的值 |
| wb\_LLbit\_value\_i | 1 | 输入 | 回写阶段写入LLbit寄存器的值 |
| reg2\_i | 32 | 输入 | 要写入存储器的寄存器中的数据 |
| LLbit\_value\_o | 1 | 输出 | 写入LLbit寄存器的值 |
| current\_inst\_address\_o | 32 | 输出 | 当前指令的地址，被epc寄存器所使用 |
| mem\_ce\_o | 1 | 输出 | 数据存储器使能信号 |

**表16 访存部件接口说明**

在访存阶段，除了执行正常的访存操作之外，还负责给出最终的异常类型，因此就要获取协处理中相关寄存器的值。正如前面所述，流水线获取值避免不了RAW相关，因此还是采用数据前推的方式，添加新的接口wb\_cp0\_reg\_write\_addr以及其他，用来最终决定本阶段读取到的值，对于LLbit寄存器也是如此。在访存阶段给出最终异常类型，此前的相关写入操作均已执行完毕，因此做到了精确异常。给出最终异常的同时还要向Ctrl部件传递消息，令其在下一个时钟周期刷新流水线，并准备跳转到异常处理程序之处。本设计对于外部中断处理的程序放在了地址32’h0000000c处。

### 3.2.6 DIV除法器 (相同意义的接口说明不再赘述)

# 

**图14 除法器部件**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| annul\_i | 1 | 输入 | 是否取消除法运算 |
| opdata1\_i | 32 | 输入 | 被除数 |
| opdata2\_i | 32 | 输入 | 除数 |
| signed\_div\_i | 1 | 输入 | 有符号除法信号 |
| start\_i | 1 | 输入 | 开始运算信号 |
| ready\_o | 1 | 输出 | 除法运算是否结束 |
| result\_o | 64 | 输出 | 除法结果：商32位、余数32位 |

**表17 除法器部件接口说明**

### 3.2.7 CP0 协处理器

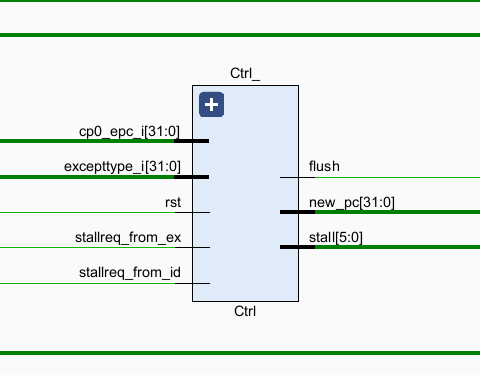
**图15 CP0协处理器部件**

|  |  |
| --- | --- |
| 寄存器 | 说明 |
| count[31:0] | 计数寄存器，从0开始每经过一个时钟周期+1 |
| compare[31:0] | 当compare==count时，通过timer\_int\_o产生一个时钟中断 |
| status[0] | 为1表示中断使能，为0表示中断禁止 |
| status[1] | 异常发生时设置本字段为1表示处于异常级 |
| status[15:8] | 对应8个外部中断源，为1代表不屏蔽，为0代表屏蔽 |
| cause[15:10] | 中断挂起字段，相应位表示中断是否发生，对应6个硬件中断 |
| cause[9:8] | 中断挂起字段，相应位表示中断是否发生，对应2个软件中断 |
| cause[6:2] | 记录发生了哪种异常 |
| epc[31:0] | 用来存储异常处理后返回的地址，可读可写 |

**表18 协处理器各寄存器说明**

CP0负责接收访存阶段传递而来的异常信息以及外部中断信息，并将接收到的指令地址保存到epc当中作为异常返回的地址。

### 3.2.8 Ctrl 控制流水线暂停以及刷新



**图16 Ctrl部件**

|  |  |  |  |
| --- | --- | --- | --- |
| 接口名 | 宽度 | 类型 | 说明 |
| cp0\_epc\_i | 32 | 输入 | 获取epc寄存器的值作为pc |
| excepttype\_i | 32 | 输入 | 异常类型 |
| stallreq\_from\_ex | 1 | 输入 | 执行阶段请求流水线暂停 |
| stallreq\_from\_id | 1 | 输入 | 译码阶段请求流水线暂停 |
| flush | 1 | 输出 | 流水线刷新信号 |
| new\_pc | 32 | 输出 | 新的pc值 |
| stall | 6 | 输出 | 暂停信号，哪些阶段需要暂停 |

**表19 Ctrl部件接口说明**

控制部件Ctrl负责流水线的暂停，译码阶段需要获取访存阶段前推值时会将流水线暂停1个时钟周期，执行阶段进行除法运算时会将流水线暂停32个时钟周期，比如当执行阶段请求暂停时，停止取值与译码操作，而其后的访存与回写执行完毕后插入nop操作。Ctrl还负责流水线的刷新，与返回地址的写入，当检测到异常时，Ctrl会向各个锁存器部件发送流水线刷新信号，然后跳转至异常处理程序，当eret异常调用时，Ctrl会将epc寄存器的值送入PC之中。

由于篇幅原因，负责传递数据流的中间锁存器部件不再说明。比如IftoID、IdtoEX等，它们的作用以及接口可以很容易理解其作用。

# 4 测试指令

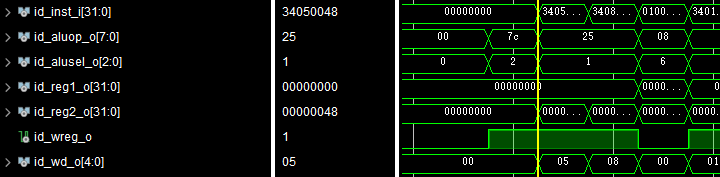
1. ori 5 0 72
2. ori 8 0 40
3. jr 8
4. ori 1 0 1
5. sw 1 0(0)
6. ori 2 0 1
7. sw 2 0(0)
8. ori 3 0 2
9. sw 3 0(0)
10. eret
11. ori 9 0 65281
12. mtc0 9 12
13. ori 1 0 1
14. sw 1 0(0)
15. ori 2 0 1
16. sw 2 0(0)
17. ori 3 0 2
18. sw 3 0(0)
19. **or** 1 0 2
20. **or** 2 0 3
21. add 3 1 2
22. sw 3 0(0)
23. jr 5
24. sync
25. sync
26. sync
27. sync

该段指令的数据流覆盖了所有类型，包括算术运算、访存、跳转、异常。1-2行用来初始化寄存器，5号寄存器和8号寄存器存放了跳转的目标地址。第3行跳转到地址40处执行，因为4-10行为中断处理程序。接下来11-12行负责初始化协处理，使能中断并且不屏蔽。13-17行负责初始化1、2、3号寄存器分别为1、1、2，即斐波那契数列的前三项。

利用Core实现的功能是在fpga芯片数码管上显示斐波那契数列，当数值大于10时，触发外部中断重新从最初项显示，也就是4-10行的中断处理程序。在Core内的数据怎样被外部所得知呢？本设计利用了访存部件的mem\_data\_o管脚，通过sw指令将寄存器中的数据加载到某个内存单元(此处加载到了0号单元)，在访存阶段外部通过获取mem\_data\_o管脚的值就可以得知当前的值了。

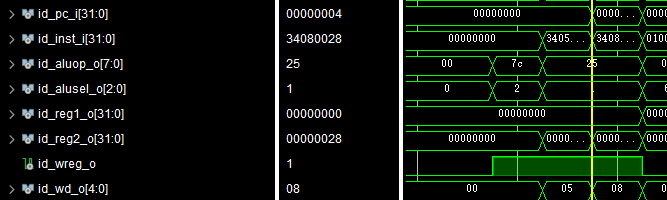
接下来就是不断计算下一项的值，将2号寄存器的值送入1号寄存器，将3号寄存器的值送入2号寄存器，利用add指令相加将结果存放在三号寄存器，循环。（第23行：jr 5 跳转到5号寄存器的值，即之前初始化的72）。工程文件内提供了python脚本可将对应指令翻译成二进制指令(01格式)以供读入。

# 5 仿真波形



**图17 第一条指令**

如波形图所示，72(16进制为48)将被写入到5号寄存器。



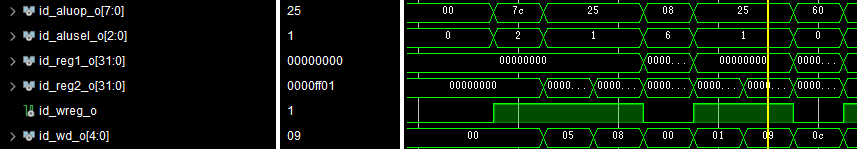
**图18 第二条指令**

如波形图所示，40(16进制表示为28)将被写入到8号寄存器。

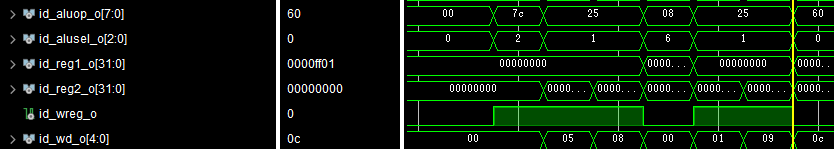


**图19 第三条指令**

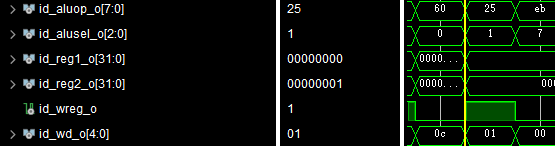
如波形图所示，pc值为0x28，但是正在译码的指令地址却是0x0c，说明第三条跳转指令生效。



**图20 第四条指令**

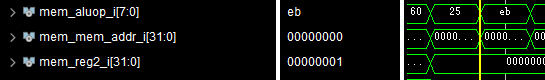
如波形图所示，0xff01将被写到9号寄存器之中，用来使能中断，关掉中断屏蔽。

**图21 第五条指令**

再通过访问协处理指令将9号寄存器的值写入到status寄存器当中。

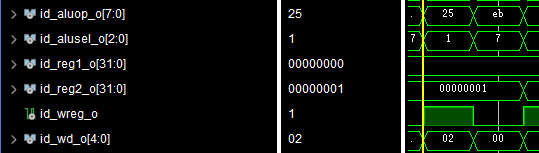
**图22 第六条指令**

数值1被写入1号寄存器之中。



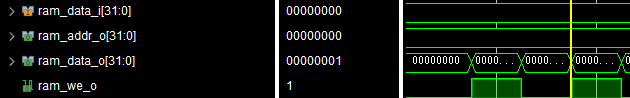
**图23 第七条指令**

1号寄存器中的值在访存阶段被送出。



**图24 第八条指令**

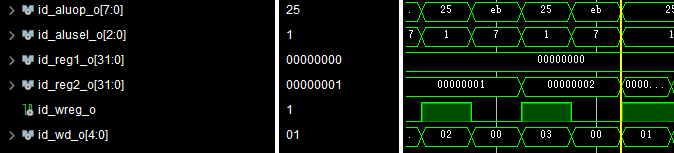
数值1被写入二号寄存器之中。



**图25 第九条指令**

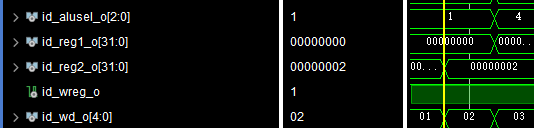
再次将数值1送入外部。

接下来三号寄存器值被初始化为2然后送出，波形图与上述相仿，此处不再进行赘述。



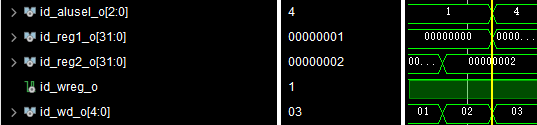
**图26 第十条指令**

将0号寄存器中的值与2号寄存器中的值或运算送入1号寄存器之中。



**图27 第十一条指令**

将0号寄存器中的值与3号寄存器中的值或运算送入2号寄存器之中。



**图28 第十二条指令**

将1号寄存器中的值与2号寄存器中的值相加送入3号寄存器之中。



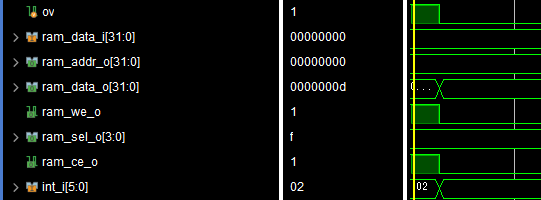
**图29 第十三条指令**

将相加后的值送入0号单元之中。



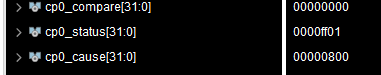
**图30 第十四条指令**

跳转回之前继续执行指令。



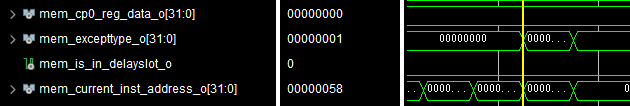
**图31 第十五条指令**

当访存指令将0x0d送出时，检测到大于10的值，ov置为1，外部中断输入int\_i将被送入到CP0协处理之中。

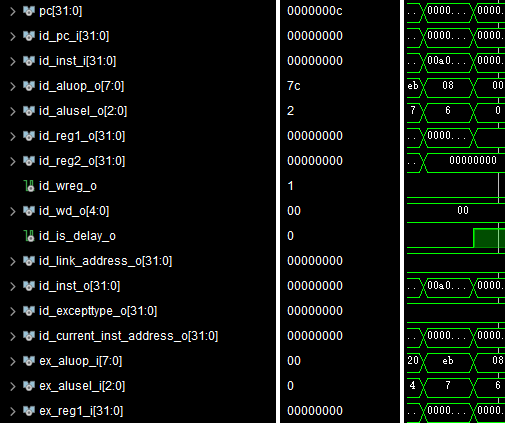


**图32 第十六条指令**

下一个时钟周期，Cause寄存器相应的位被置1。

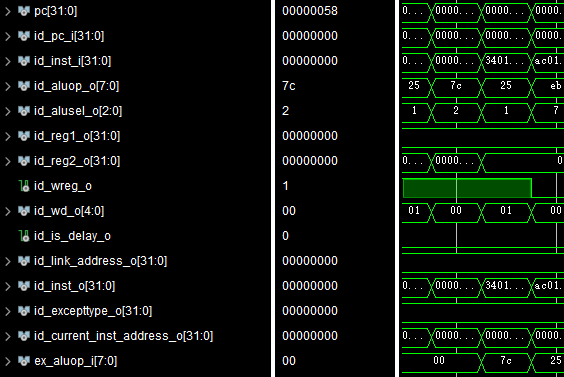


MEM部件也根据cause寄存器的值给出了最终异常类型。



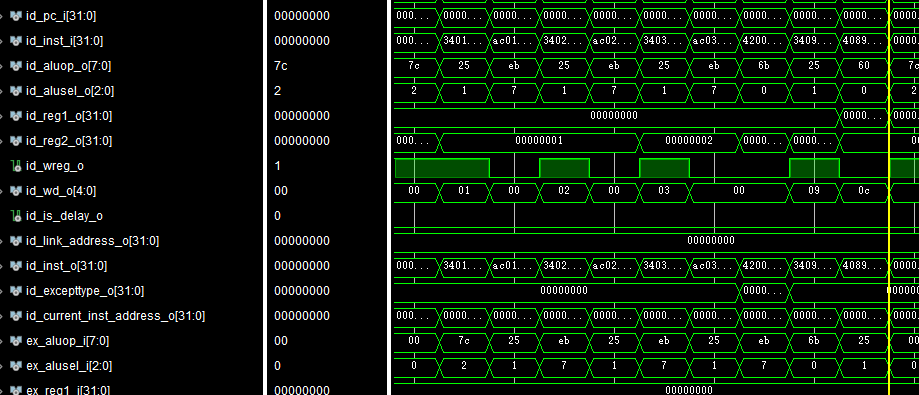
**图33 第十七条指令**

下一个时钟周期流水线被冲刷，pc被指为异常处理程序地址。



**图34 第十八条指令**

调用eret返回异常后，流水线再次被冲刷，此时pc被置为上次发生异常时指令的地址即跳转处jr 5，跳转后继续进行循环执行。



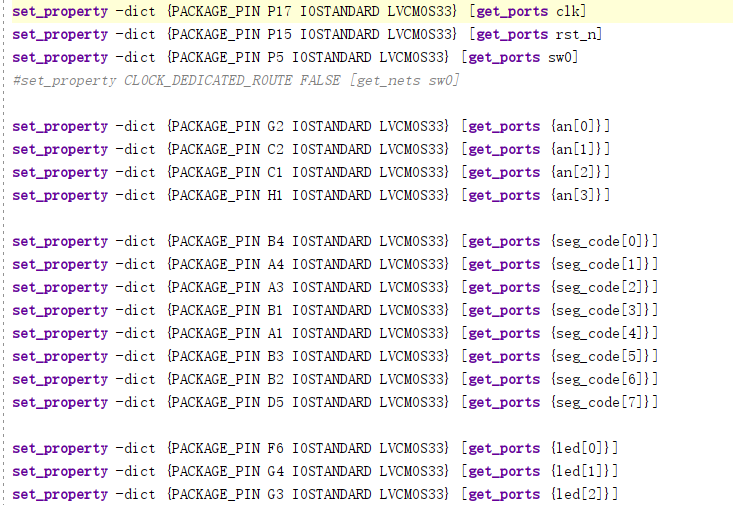
**图35 数据流水线**

可以看出流水线是符合程序逻辑的。

# 6 生成烧写文件

## 6.1 准备约束文件

此步骤指明了目标开发板与引脚之间的关系，将对应的引脚要声明在其中。



**图36 约束文件示例**

开发板提供了一个100MHz频率的时钟，直接拿来做时钟信号肯定是不行的。有两种解决方案，一种是利用寄存器分频。

1. always@(posedge clk)**begin**
2. **if**(rst\_n==0)**begin**
3. rst\_w<=1;
4. cnt\_w<=0;
5. start<=0;
6. **end**
7. **else** **begin**
8. cnt\_w<=cnt\_w+1;
9. start<=start+1;
10. **if**(start==27'b111111111111111111111111111) rst\_w<=0;
11. **end**
12. **end**
13. assign clk\_w=(cnt\_w==1);

第二种方法就是利用开发板上的某个按键作为时钟，比如按下为高电平、松开为低电平。但是直接将其接到时钟接口会出现如下错误：

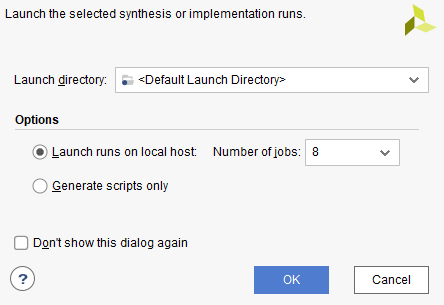
[Place 30-574] Poor placement for routing between an IO pin and BUFG. If this sub optimal condition is acceptable for this design, you may use the CLOCK\_DEDICATED\_ROUTE constraint in the……………………..

因此需要添加如下语句：

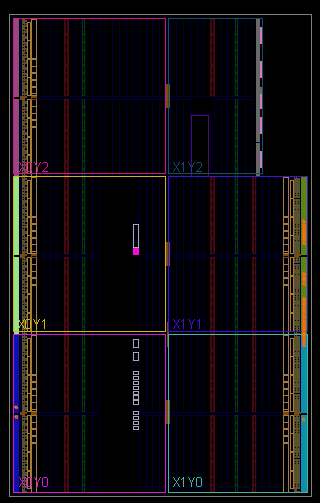
1. set\_property CLOCK\_DEDICATED\_ROUTE **FALSE** [get\_nets sw0]

之后就可以利用开关模拟时钟信号。

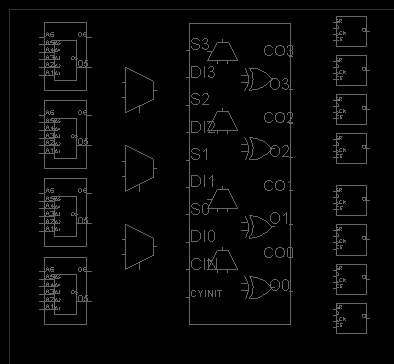
## 6.2 Synthesis



**图37 进行综合**



**图38 综合完毕**

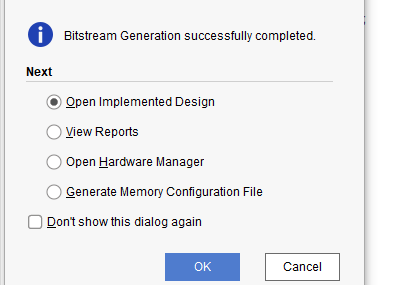


**图39 微观布局**

## 6.2 Implementation

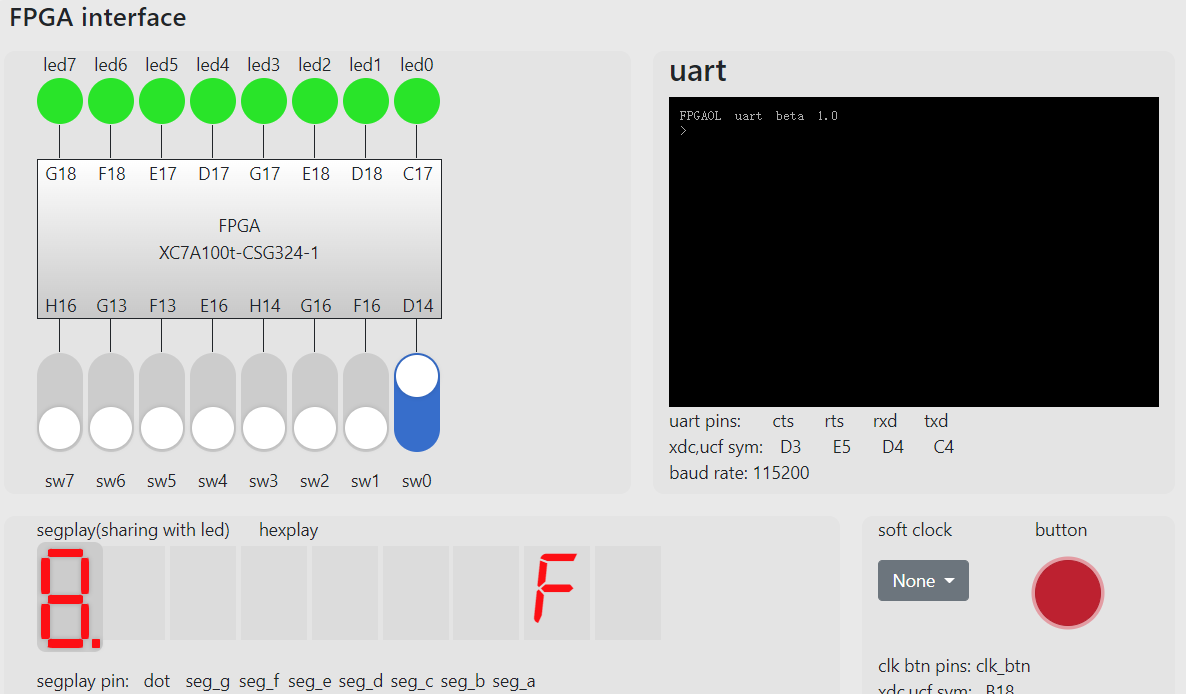
**图40 实现完毕**

## 6.3 Generate Bitstream

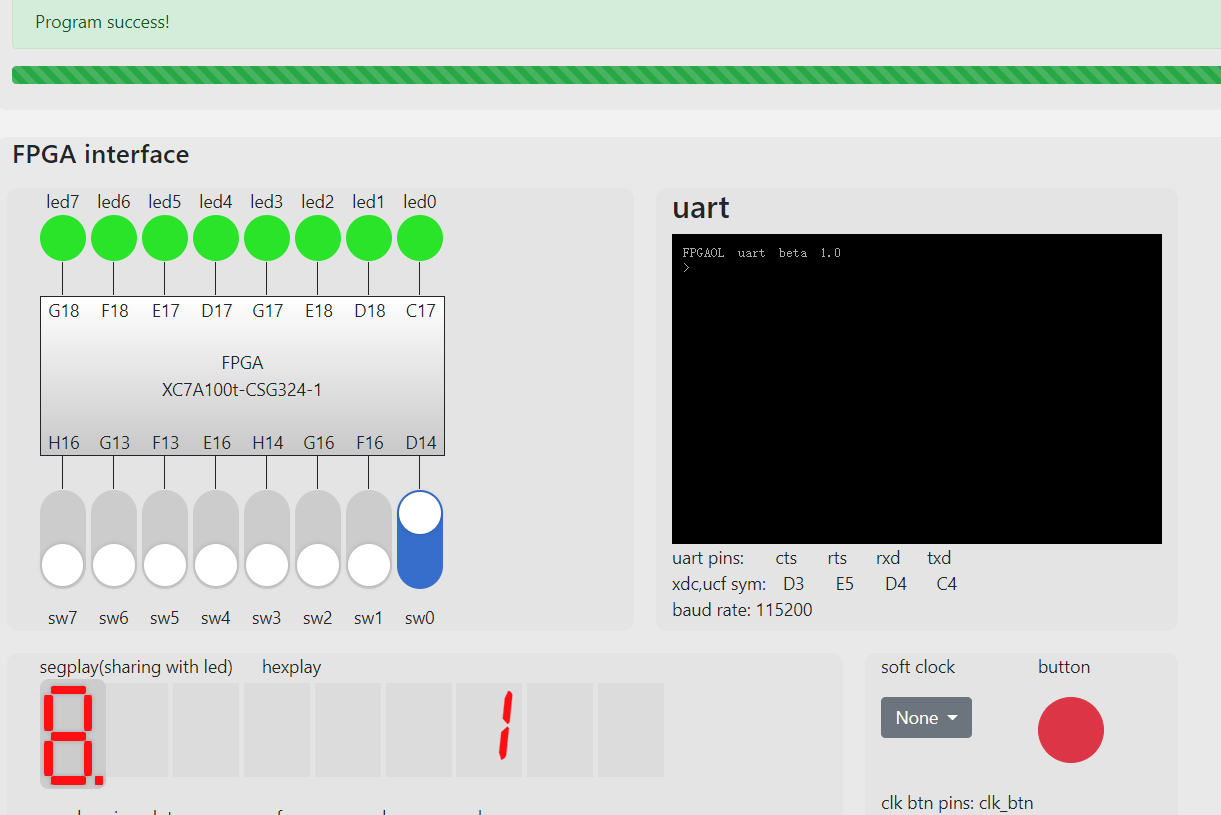


**图41 生成比特流完毕**

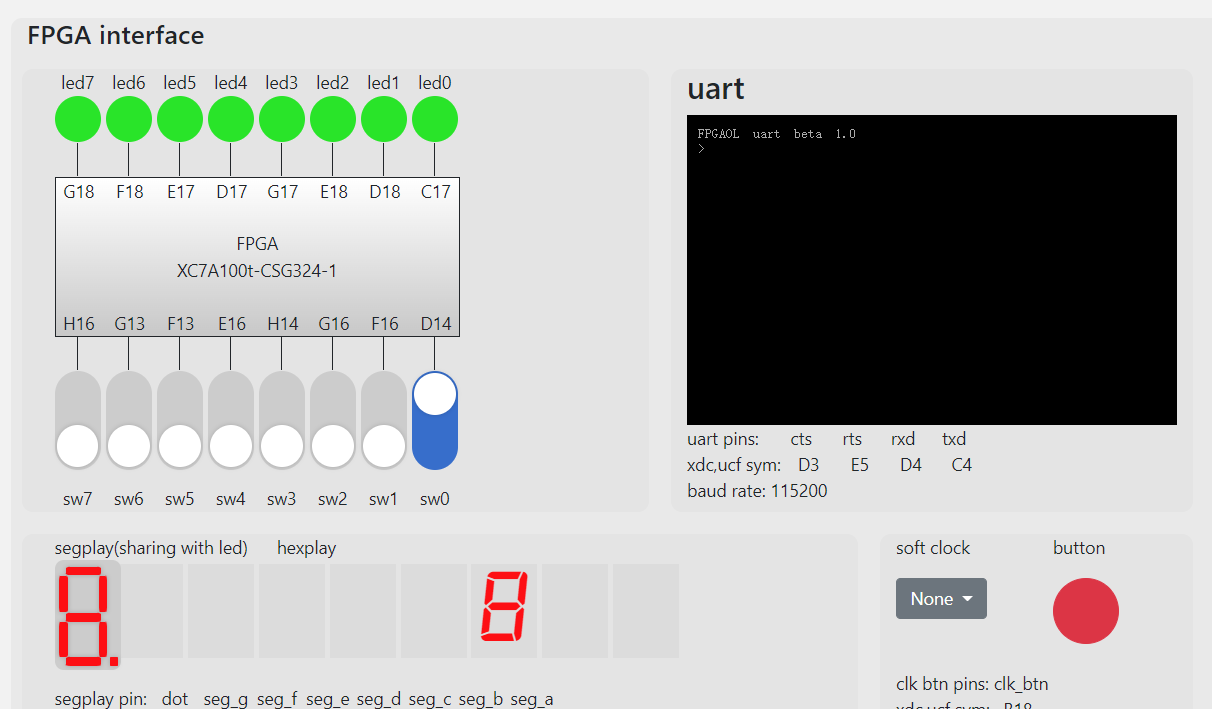
当初烧写完毕后没有及时拍照直接去验收了，等到撰写实验报告时疫情突发，无法再借到开发板，因此只能用中科大的FPGAOL虚拟平台进行。将生成的.bit文件进行烧写(在此之前要更换一下型号)：



**图42 按下复位键**



**图43 从1开始显示**



**图43 到达最后一项，之后会触发中断重新从1开始**

# 参考文献

1. 雷思磊. 自己动手写CPU. 电子工业出版社，2014年.
2. David A.Patterson, John L.Hennessy. 计算机组成设计：硬件/软件接口.机械工业出版社，2010年

# 总结

经过本次课程设计，我更加深刻理解了计算机的底层硬件结构，也理清了老师上课所讲的一些问题。比如在计算机体系结构中曾经提到各流水线部件之间要有锁存器来传递数据流，一开始我还会感到疑惑。等到后来真正实践时我才搞明白它们的真正作用，流水线部件获取数据流可能不会在上升沿到来直接得到，有时需要先输出才能得到数据，比如mem部件不会在时钟上升沿到来就获取内存单元的存储字，而是要在它输出地址值之后才会将存储字送入其中。这时候锁存器连接它后面的流水线部件，锁存器起到时序单元的作用，而流水线部件更像是组合逻辑单元，以此相互配合来完成工作。

还有关于分支的处理问题，对于跳转后面一条指令的处理，在我所参考的书籍中的数据流是直接让他进入了流水线。我认为书中的做法略有不妥，应该在回写阶段再加1个接口，如果分支指令成功跳转，则分支指令后面的一条指令在流水的最后一个阶段也就是回写阶段就直接插入一个nop操作，如果没有成功跳转，那么就继续进行回写操作。

总的来说，经过系统硬件综合设计这门课之后，我感觉带给我个人的提升非常大。

|  |
| --- |
| 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:** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | 评价内容 | | 权重 | 得分 | | **验收** | 实验原理是否理解；程序能否运行；实验结果是否正确；任务是否全部完成。 | | 0.6 |  | | **设计报告** | 1 | 报告格式是否规范，语言使用是否规范，行文是否流畅，是否图文并茂； | 0.1 |  | | 2 | 实验原理、实验步骤描述是否正确、详实；  程序流程图是否规范，代码实现是否正确；  实验数据记录是否完整，实验结果是否正确；  实验结果的分析、对比是否充分； | 0.2 |  | | 3 | 实验体会是否正确，是否提出了自己独到见解。 | 0.1 |  | | 合计 |  | | | | | 指导教师（签章）： 年 月 日 | | | | | |