****

**《系统硬件综合设计》**

**报告**

**学生姓名 刘嘉伟**

**学 号 2018213106**

**专业班级 计算机创新实验18-1班**

**指导教师 阙夏**

**院系名称 计算机与信息学院**

**2021 年 1 月 15 日**

[一、实验目的 3](#_Toc61724325)

[二、实验平台 3](#_Toc61724326)

[三、实验设计与过程 3](#_Toc61724327)

[3.1 RISC-V 32I指令格式 3](#_Toc61724328)

[3.2 计划实现的指令列表如下 5](#_Toc61724329)

[3.3 整体设计图 6](#_Toc61724330)

[3.4 常量定义 7](#_Toc61724331)

[3.5 各模块具体设计与编写 9](#_Toc61724332)

[四、实验结果 23](#_Toc61724333)

[仿真 23](#_Toc61724334)

[FPGA验证： 26](#_Toc61724335)

[五、实验总结 31](#_Toc61724336)

# 实验目的

设计并实现一个多周期流水CPU

* 三种类型的指令各若干条
* MIPS、ARM、RISC-V等类型CPU都可以
* 下载到FPGA上进行验证

# 实验平台

Vivado 2020.2

# 实验设计与过程

我设计的CPU指令集是RISC-V32I，该指令集总共包含47条指令，我实现了其中的37条。

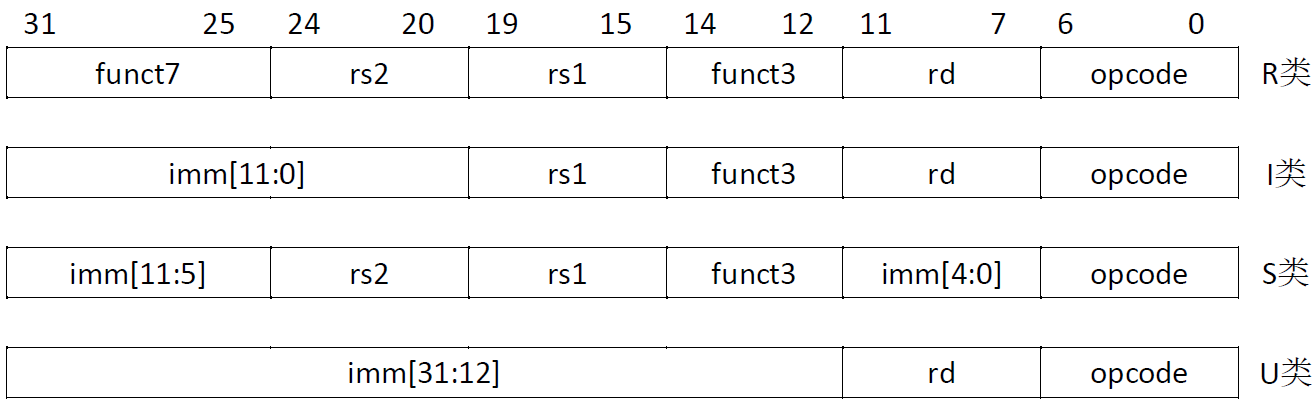
## 3.1 RISC-V 32I指令格式

有四种核心指令格式（R/I/S/U），所有的指令长度都是固定的32位，并且在存储器中必须4字节边界对齐。

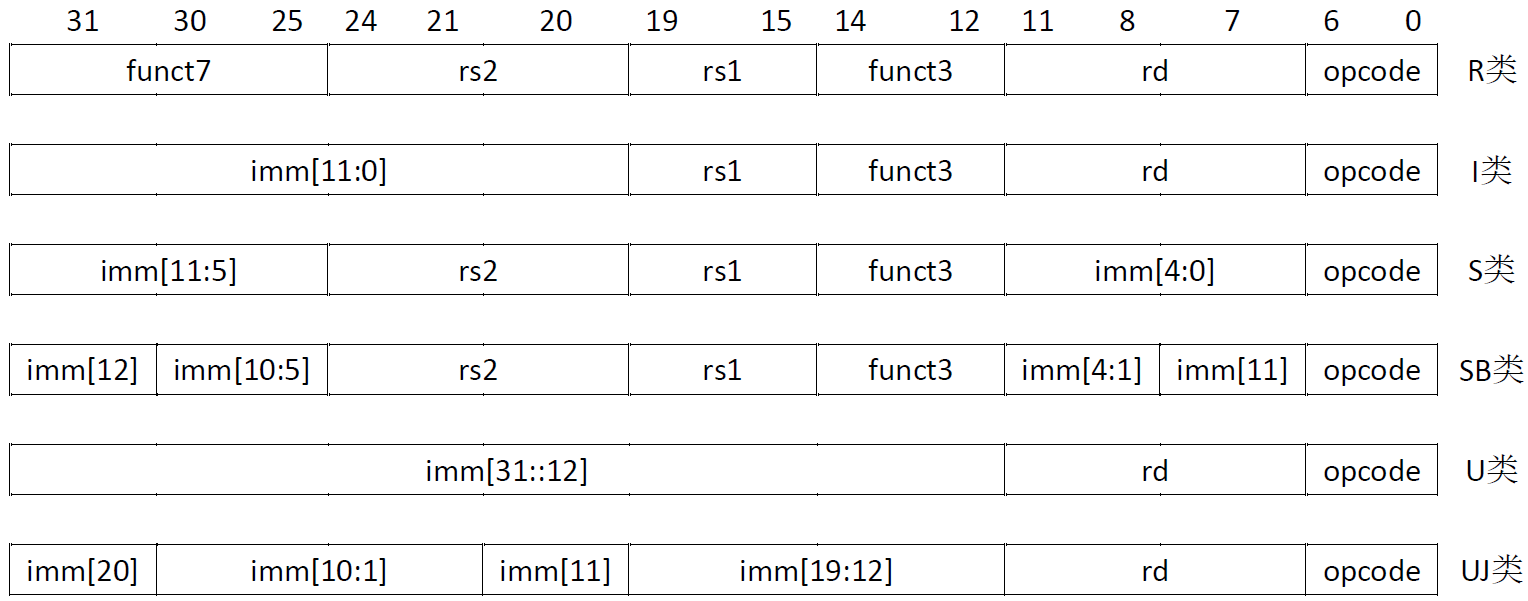
其中opcode字段指明了该条指令的操作类型，一般对于一大类指令，再根据funct3或/和funct7字段来确定具体指令（某些指令直接由opcode确定）

rs1，rs2，rd字段，分别指源操作数1、源操作数2、目的操作数的寄存器编号，在所有的格式中，这三者位置是固定的，方便指令译码。

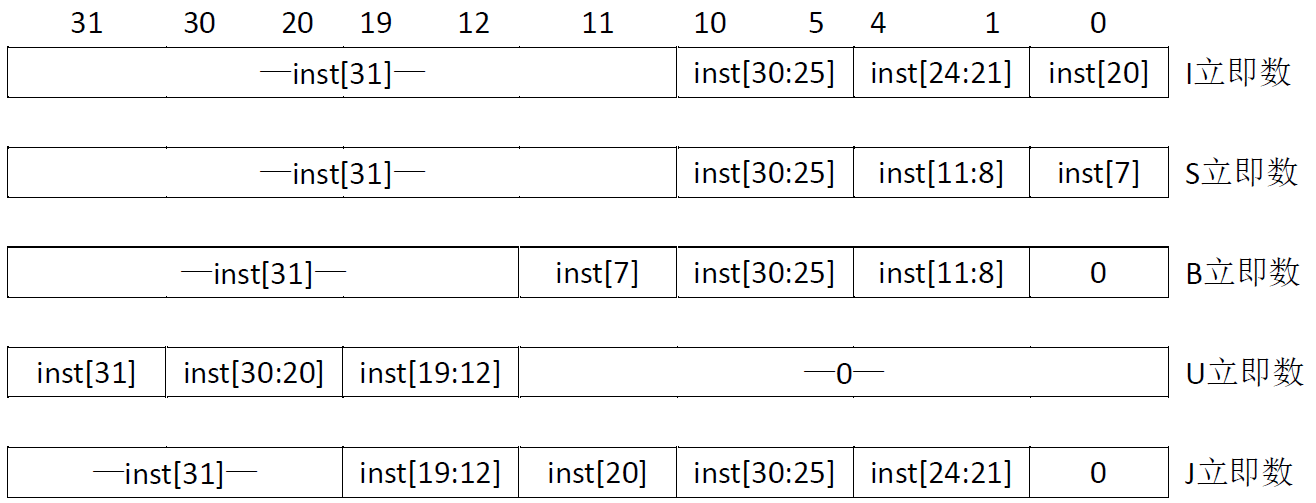
另外，所有立即数的符号位总是在指令的第31位，以加速符号扩展电路。



为了方便对立即数的处理，增加了两种指令格式变种，由 S 类派生出的 SB 类，由 U 类派生出的 UJ 类



除了 R 类指令，剩下五类指令中都包含立即数，使用立即数前应该对立即数进行符号拓展（32 位），得到以下 5 类立即数



其中 inst[x]表示，立即数的这一位是指令中的第 x 位(0-31)，符号拓展总是在指令的第 31 位。

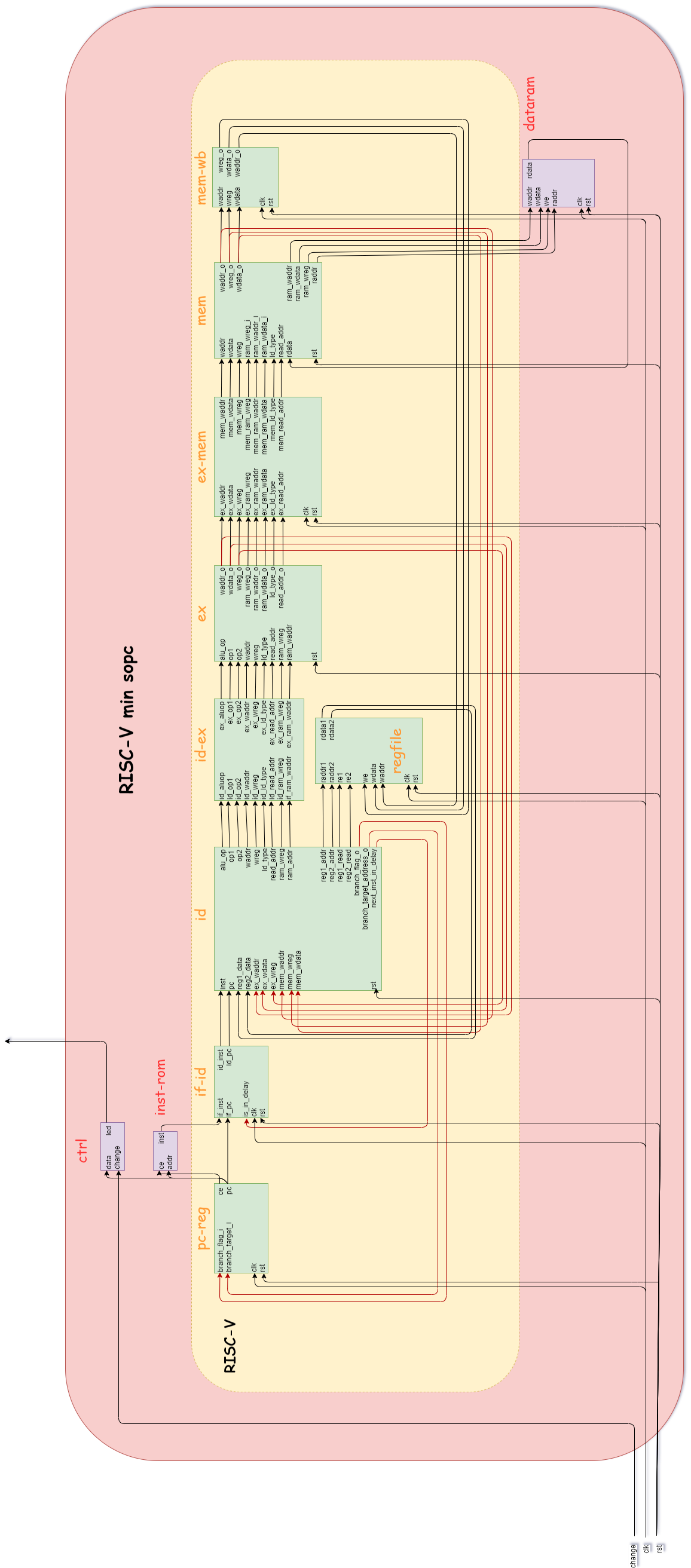
S 和 SB 格式唯一的区别在于，在 SB 格式中，12 位立即数字段用于编码 2 的倍数的分支偏移量。（SB 立即数的第 0 位固定为 0）

U 和 UJ 格式唯一的区别在于，20 位立即数被左移 12 位以生成 U 立即数，而被左移 1 位以生成 J 立即数。

## 3.2 计划实现的指令列表如下

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 指令 | 类型 | Inst[31:25] | Inst[24:20] | Inst[19:15] | Inst[14:12] | Inst[11:7] | Inst[6:0] |
| LUI | U | imm[31:12] | | | | rd | 0110111 |
| AUIPC | U | imm[31:12] | | | | rd | 0010111 |
| JAL | J | imm[20|10:1|11|19:12] | | | | rd | 1101111 |
| JALR | I | imm[11:0] | | rs1 | 000 | rd | 1101111 |
| BEQ | B | imm[12|10:5] | rs2 | rs1 | 000 | imm[4:1|11] | 1100011 |
| BNE | imm[12|10:5] | rs2 | rs1 | 001 | imm[4:1|11] | 1100011 |
| BLT | imm[12|10:5] | rs2 | rs1 | 100 | imm[4:1|11] | 1100011 |
| BGE | imm[12|10:5] | rs2 | rs1 | 101 | imm[4:1|11] | 1100011 |
| BLTU | imm[12|10:5] | rs2 | rs1 | 110 | imm[4:1|11] | 1100011 |
| BGEU | imm[12|10:5] | rs2 | rs1 | 111 | imm[4:1|11] | 1100011 |
| LB | I | imm[11:0] | | rs1 | 000 | rd | 0000011 |
| LH | imm[11:0] | | rs1 | 001 | rd | 0000011 |
| LW | imm[11:0] | | rs1 | 010 | rd | 0000011 |
| LBU | imm[11:0] | | rs1 | 100 | rd | 0000011 |
| LHU | imm[11:0] | | rs1 | 101 | rd | 0000011 |
| SB | S | imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 |
| SH | imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 |
| SW | imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 |
| ADDI | I | imm[11:0] | | rs1 | 000 | rd | 0010011 |
| SLTI | imm[11:0] | | rs1 | 010 | rd | 0010011 |
| SLTIU | imm[11:0] | | rs1 | 011 | rd | 0010011 |
| XORI | imm[11:0] | | rs1 | 100 | rd | 0010011 |
| ORI | imm[11:0] | | rs1 | 110 | rd | 0010011 |
| ANDI | imm[11:0] | | rs1 | 111 | rd | 0010011 |
| SLLI | 0000000 | rs2 | rs1 | 001 | rd | 0010011 |
| SRLI | 0000000 | rs2 | rs1 | 101 | rd | 0010011 |
| SRAI | 0100000 | rs2 | rs1 | 101 | rd | 0010011 |
| ADD | R | 0000000 | rs2 | rs1 | 000 | rd | 0110011 |
| SUB | 0100000 | rs2 | rs1 | 000 | rd | 0110011 |
| SLL | 0000000 | rs2 | rs1 | 001 | rd | 0110011 |
| SLT | 0000000 | rs2 | rs1 | 010 | rd | 0110011 |
| SLTU | 0000000 | rs2 | rs1 | 011 | rd | 0110011 |
| XOR | 0000000 | rs2 | rs1 | 100 | rd | 0110011 |
| SRL | 0000000 | rs2 | rs1 | 101 | rd | 0110011 |
| SRA | 0100000 | rs2 | rs1 | 101 | rd | 0110011 |
| OR | 0000000 | rs2 | rs1 | 110 | rd | 0110011 |
| AND | 0000000 | rs2 | rs1 | 111 | rd | 0110011 |

## 3.3 整体设计图



## 3.4 常量定义

预先在模块defines.v中定义实验中使用的常量，方便阅读。

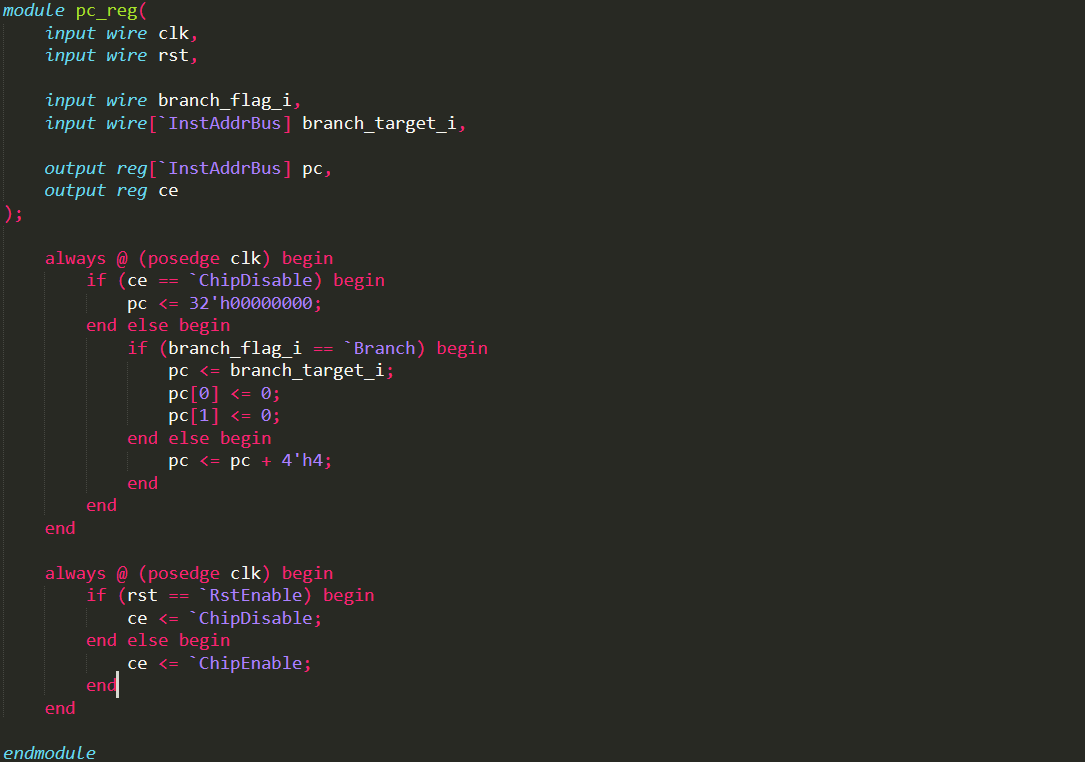
|  |
| --- |
| `define RstEnable 1'b1  `define RstDisable 1'b0  `define ZeroWord 32'h00000000  `define WriteEnable 1'b1  `define WriteDisable 1'b0  `define ReadEnable 1'b1  `define ReadDisable 1'b0  `define ChipEnable 1'b1  `define ChipDisable 1'b0  `define I\_OP 7'b0010011  `define R\_OP 7'b0110011  `define B\_OP 7'b1100011  `define LUI\_OP 7'b0110111  `define AUIPC\_OP 7'b0110111  `define S\_OP 7'b0100011  `define JAL\_OP 7'b1101111  `define JALR\_OP 7'b1100111  `define LOAD\_OP 7'b0000011  `define ALU\_NO\_OP 4'b1111  `define ADD 4'b0000  `define SLL 4'b0001  `define SLT 4'b0010  `define SLTU 4'b0011  `define XOR 4'b0100  `define SRL 4'b0101  `define OR 4'b0110  `define AND 4'b0111  `define SUB 4'b1000  `define LUI 4'b1001  `define SB 4'b1010  `define SH 4'b1011  `define SW 4'b1100  `define SRA 4'b1101  `define JAL 4'b1110  `define ADDI 4'b0000  `define SLLI 4'b0001  `define SLTI 4'b0010  `define SLTIU 4'b0011  `define XORI 4'b0100  `define SRLI 4'b0101  `define ORI 4'b0110  `define ANDI 4'b0111  `define SRAI 4'b1101  `define Branch 1'b1  `define NotBranch 1'b0  `define BEQ 3'b000  `define BNE 3'b001  `define BLT 3'b100  `define BGE 3'b101  `define BLTU 3'b110  `define BGEU 3'b111  `define LB 3'b000  `define LW 3'b010  `define LH 3'b001  `define LBU 3'b100  `define LHU 3'b101  //指令存储器inst\_rom  `define InstAddrBus 31:0  `define InstBus 31:0  `define InstMemNum 131071  `define InstMemNumLog2 17  //通用寄存器regfile  `define RegAddrBus 4:0  `define RegBus 31:0  `define SelBus 2:0  `define OpBus 3:0  `define NOPRegAddr 5'b00000 |

在该模块中定义了数据总线宽度，数据地址线宽度等一系列宽度，还定义了各种使能信号，如芯片使能、读使能，写使能等。里面还定义了各类指令的指令码opcode，以及指令码的子类型编码funct3和funct7。

## 3.5 各模块具体设计与编写

|  |  |  |  |
| --- | --- | --- | --- |
| pc\_reg.v | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | clk | 时钟信号 |
| input | [0:0] | rst | 复位信号 |
| input | [0:0] | branch\_flag\_i | 是否跳转标志 |
| input | [31:0] | branch\_target\_i | 跳转目标地址 |
| output | [31:0] | pc | 下一条指令地址 |
| output | [0:0] | ce | 是否取指信号 |

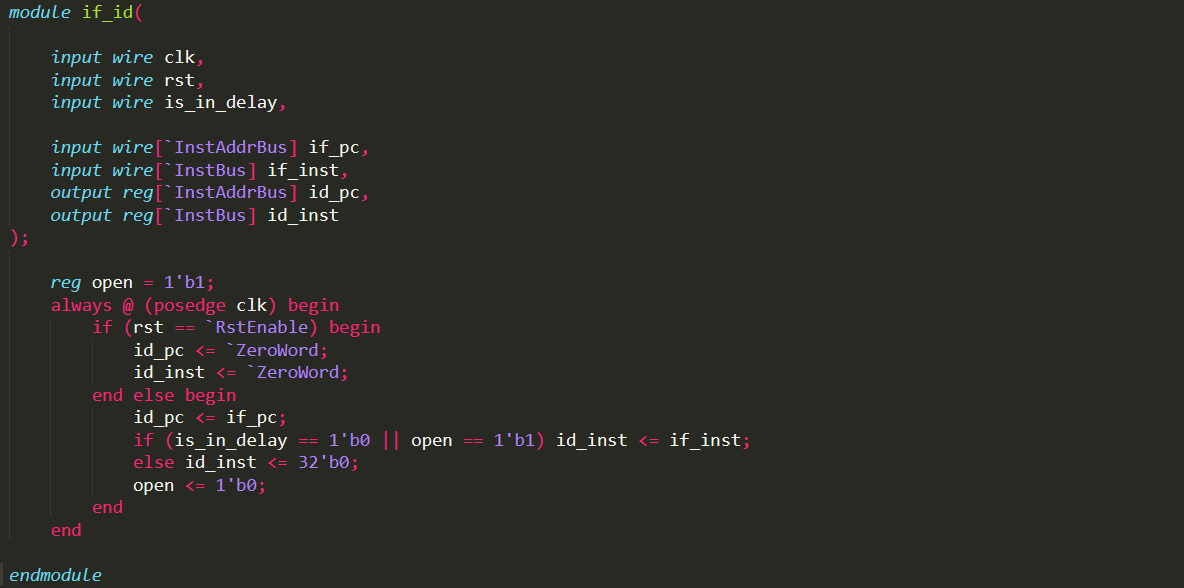
该模块是pc模块，输入跳转标志和跳转的目标地址来进行指令的跳转，并且内部的PC每个时钟周期自增4，因为CPU是按字节存储，每条指令一个字，占4个字节，所以每次都要递增4。当跳转标志为真时，PC将跳转至目标地址，否则自增4。具体代码如下：



|  |  |  |  |
| --- | --- | --- | --- |
| if\_id.v | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | clk | 时钟信号 |
| input | [0:0] | rst | 复位信号 |
| input | [0:0] | is\_in\_delay | 该条指令是否处于延迟槽 |
| input | [31:0] | if\_pc | 该条指令地址 |
| input | [31:0] | if\_inst | 该条指令值 |
| output | [31:0] | id\_pc | 指令地址输出到译码模块 |
| output | [0:0] | id\_inst | 指令值输出到译码模块 |

该模块是取指和译码的中间模块，将输入的指令值和指令地址送到译码模块，其中的时钟信号是用于控制流水，使每个时钟周期向后处理一步，也就是流水，后面的中间模块的时钟的作用都是为了控制流水，在后面就不一一解释。

如果译码阶段发现了转移指令，那么当前处于取指阶段的指令就处于延迟槽中，也就是一条无效指令，所以为了方便，当在译码阶段检测出当前指令是一条转移或跳转指令的时候，将会发送一个延迟信号到本模块，当本模块处于延迟槽时，指令无效，所以在这个模块将下一步输送的指令设为无效指令，到后续模块如果遇到无效指令自然不会去处理执行，所以就实现了转移指令的延迟槽处理。但这有个细节需要注意，在这个模块中，进行的判断是如果指令不处于延迟槽的话才会将原始指令输送到译码阶段，所以这就造成了一个问题，CPU刚开始工作的时候延迟槽信号没有数据输送过来，信号是高阻状态，将会无法传递指令。为了解决这个问题，我设置了一个开关open，初始阶段open为true，为传递指令增加一个判断条件，如果open==true，就可以直接传递指令到下一模块，然后将open关掉，open=false，经过一个周期后，延迟槽信号都会出现值，所以就可以直接通过is\_in\_delay进行判断是否处于延迟槽了。open信号也就没有作用了。代码如下：

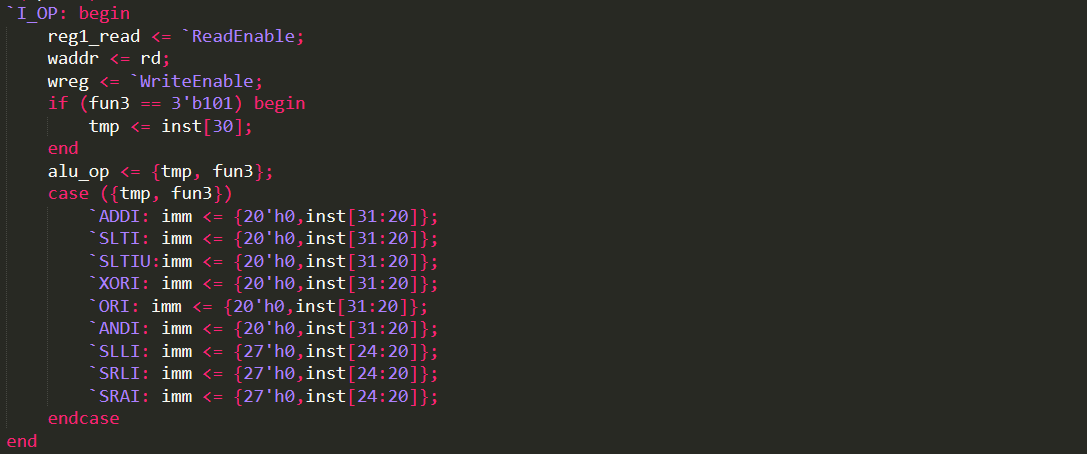


|  |  |  |  |
| --- | --- | --- | --- |
| id | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | rst | 复位信号 |
| input | [31:0] | inst | 指令值 |
| input | [31:0] | pc | 当前指令的地址 |
| input | [31:0] | reg1\_data | 寄存器端口1的值 |
| input | [31:0] | reg2\_data | 寄存器端口2的值 |
| input | [0:0] | ex\_wreg | ex模块数据前推读信号 |
| input | [31:0] | ex\_wdata | ex模块数据前推数据 |
| input | [4:0] | ex\_waddr | ex模块数据前推地址 |
| input | [0:0] | mem\_wreg | mem模块数据前推读信号 |
| input | [31:0] | mem\_wdata | mem模块数据前推数据 |
| input | [4:0] | mem\_waddr | mem模块数据前推地址 |
| output | [0:0] | wreg | 寄存器写信号 |
| output | [4:0] | waddr | 写寄存器地址 |
| output | [31:0] | op1 | 操作数1 |
| output | [31:0] | op2 | 操作数2 |
| output | [3:0] | alu\_op | 运算类型 |
| output | [4:0] | reg1\_addr | 寄存器读地址1 |
| output | [4:0] | reg2\_addr | 寄存器读地址2 |
| output | [0:0] | reg1\_read | 寄存器端口1读使能 |
| output | [0:0] | reg2\_read | 寄存器端口2读使能 |
| output | [0:0] | branch\_flag\_o | 跳转信号 |
| output | [31:0] | branch\_target\_address\_o | 跳转地址 |
| output | [0:0] | next\_inst\_in\_delay | 下条指令是否处于延迟槽 |
| output | [0:0] | ram\_wreg | ram写使能 |
| output | [31:0] | ram\_addr | ram写地址 |
| output | [31:0] | read\_addr | ram读地址 |
| output | [2:0] | ld\_type | load指令的类型 |

该模块是译码模块，非常重要的一个模块，处理指令的译码，为后续运算提供基础。该模块的设计比较复杂，我将分指令来介绍工作原理。

I类型指令：

该类型指令是移位或加减运算指令，操作数1是位于rs1的寄存器值，操作数2是立即数imm，立即数的译码如下：



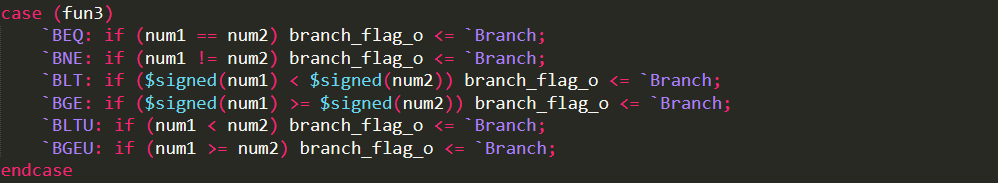
R类型指令：

R类型指令与I类型指令的作用相似，唯一不同的是I类型指令的第二操作数是imm立即数，而R类型指令的操作数是位于rs2的寄存器的值。由于两者的运算是一致的，因此alu\_op的赋值是一致的，能简化代码。

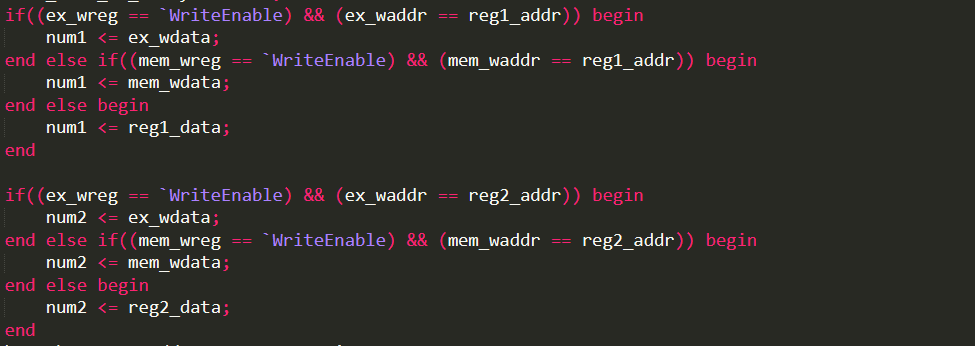


B类型指令：

B类型指令是条件转移指令，需要读取rs1和rs2寄存器中的值 ，然后依据具体的funct3的值来判断运算的类型：

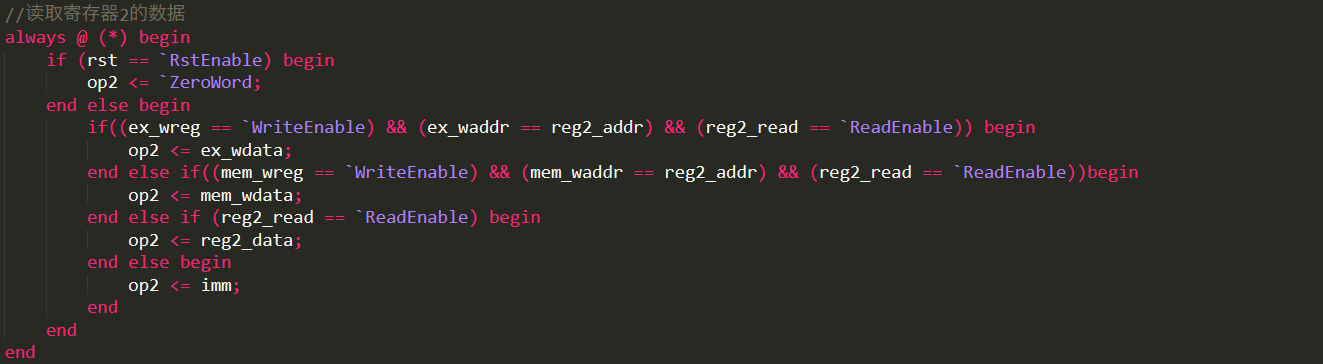


num1和num2是读取的rs1和rs2寄存器的值，原本只需要用op1和op2就能进行判断，但是不知道为什么在用op1或op2的时候，仿真的时候就会直接停止，所以我又定义了两个变量与op1和op2类型相同，赋值语句也相同。



LUI指令：

该指令是将立即数左移12位，然后保存到rd的寄存器中。实现方案很简单。由于不需要读取rs1和rs2，所以两者的读使能都是0。在这我使用了个小技巧，先看op2的赋值语句：

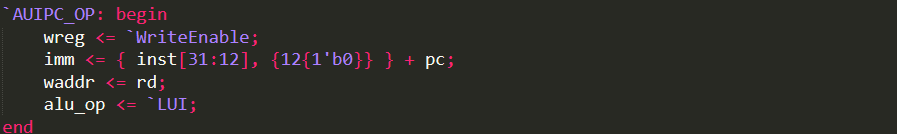


op2首先判断是否有数据前推的要处理，在这肯定没有，因为数据前推是在需要读取寄存器的值的时候才进行，这里不读取寄存器。然后如果寄存器的第二个读使能位1，就将对于的地址的寄存器内的值赋给op2，否则直接将立即数赋给op2，在这就利用这个imm，将待保存的数据放在imm中，下个时钟周期会直接通过op2将保存的数据传到执行阶段，然后再执行阶段执行相应的逻辑就可以了。在后面的一些指令译码也用到了该特性，在这同意解释，后面就不进行解释了。代码如下：



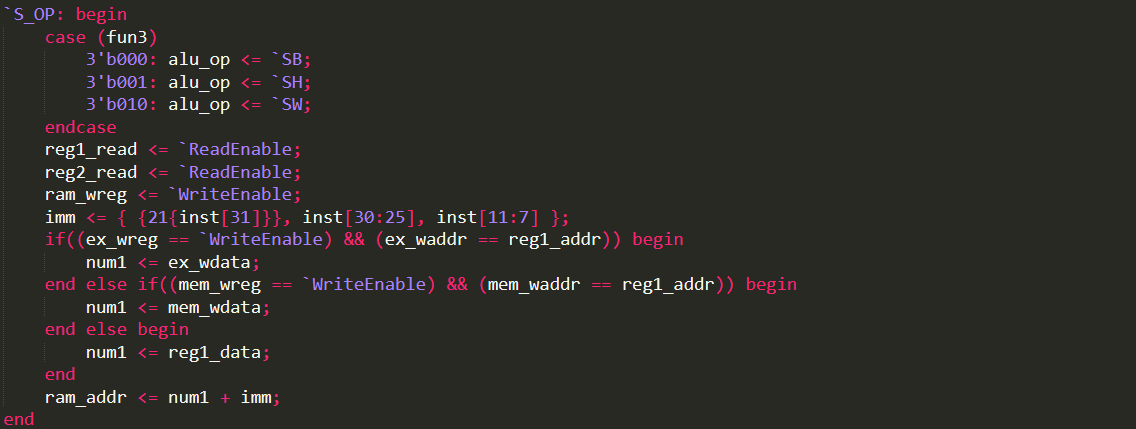
AUIPC指令：

该指令与LUI指令类似，不过保存的不是立即数，而是以立即数作为偏移量的pc的值。逻辑与LUI指令相同。



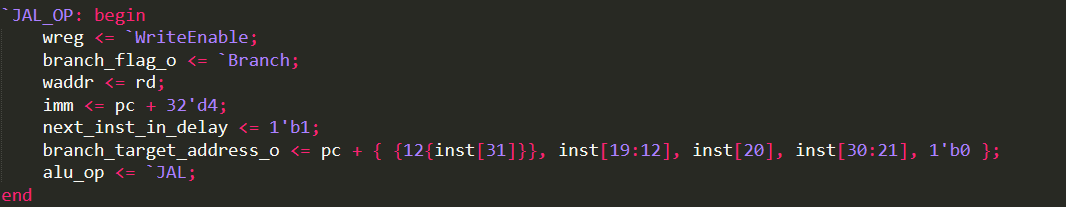
S类型指令：

S类型指令将rs2寄存器的值存放到以rs1寄存器为基址的，imm为偏移量的ram中。需要读取两个寄存器的值，然后计算出保存的地址，放到ram\_addr中传递下去，然后在执行阶段根据fun3子类型进一步判断是保存的位数是8位还是16位，进行截取。代码如下：



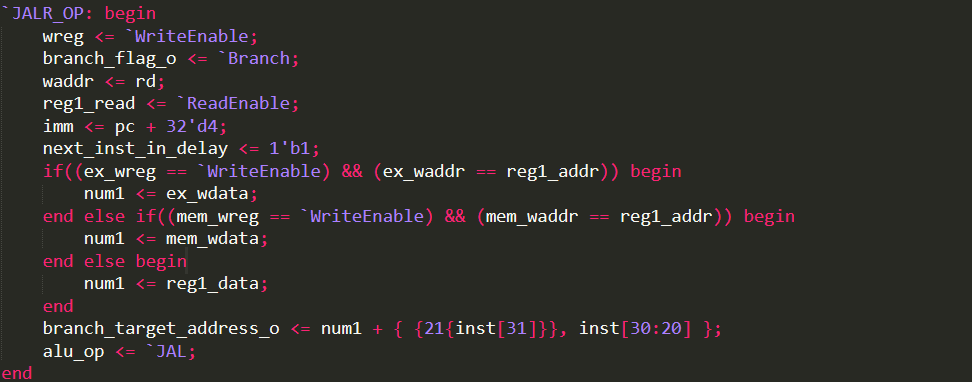
JAL指令：

这是转移指令，目标地址是以pc为基址，imm为偏移量。



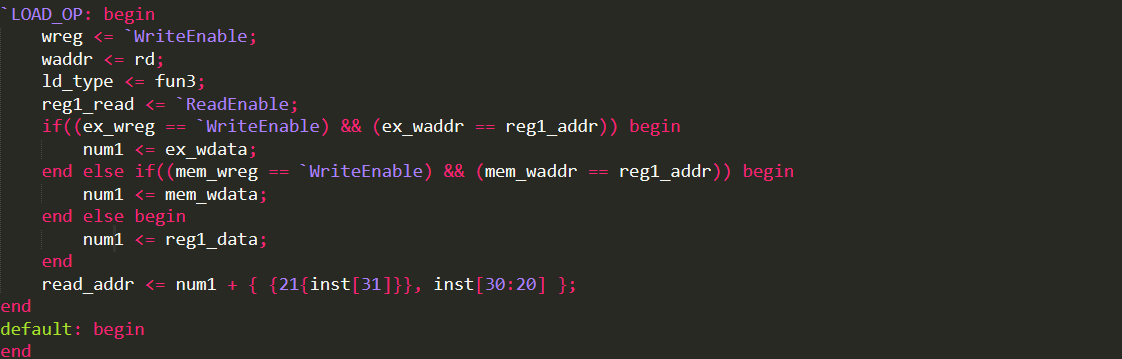
JALR指令：

与JAL指令类似，不同点在于基址为rs1寄存器的值，由于作用相同，所以操作符与JAL设置一样的，节省一个类型。

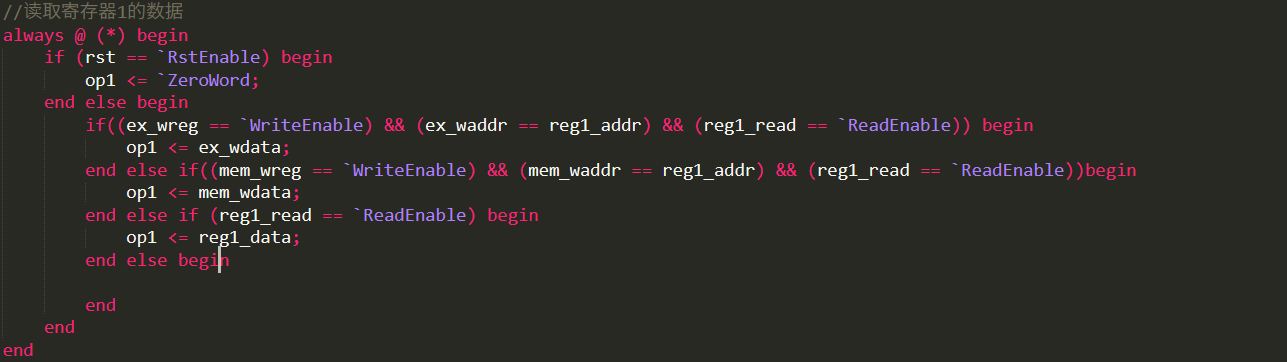


LOAD指令：

LOAD指令加载ram中的值到寄存器中，ram地址为num1寄存器中的值加imm立即数。

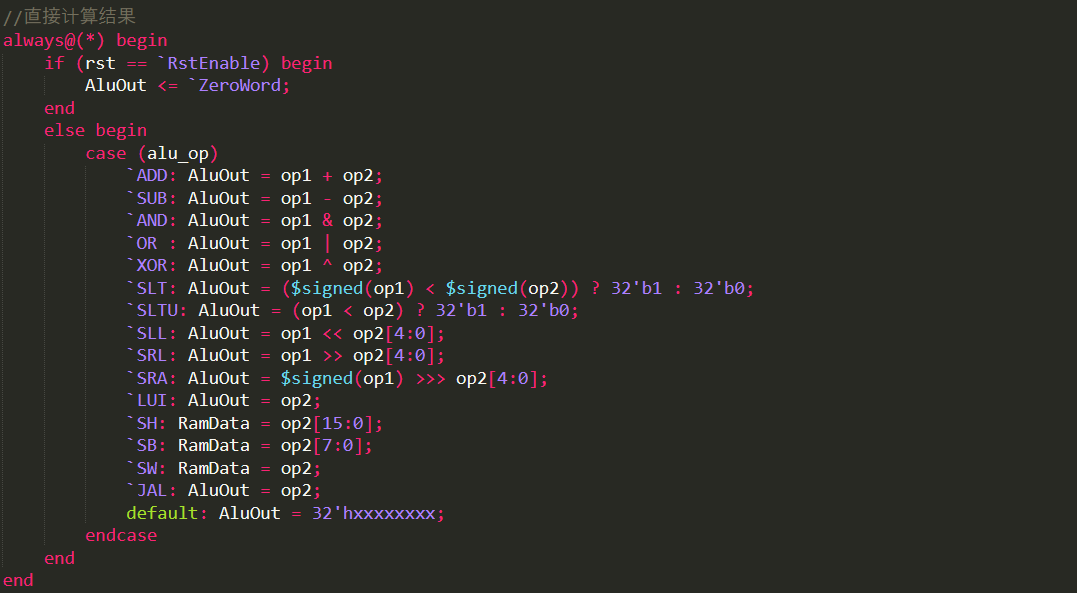


寄存器1的读取：



|  |  |  |  |
| --- | --- | --- | --- |
| ex.v | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | rst | 复位信号 |
| input | [3:0] | alu\_op | 操作符 |
| input | [0:0] | wreg | 寄存器写使能 |
| input | [4:0] | waddr | 寄存器写地址 |
| input | [31:0] | op1 | 操作数1 |
| input | [31:0] | op2 | 操作数2 |
| input | [0:0] | ram\_wreg | ram写使能 |
| input | [31:0] | ram\_waddr | ram写地址 |
| input | [2:0] | ld\_type | load指令类型 |
| input | [31:0] | read\_addr | ram读地址 |
| output | [4:0] | waddr\_o | 寄存器写地址 |
| output | [31:0] | wdata\_o | 寄存器写数据 |
| output | [0:0] | wreg\_o | 寄存器写使能 |
| output | [0:0] | ram\_wreg\_o | ram写使能 |
| output | [31:0] | ram\_waddr\_o | ram写地址 |
| output | [31:0] | ram\_wdata\_o | ram写数据 |
| output | [2:0] | ld\_type\_o | load指令类型 |
| output | [31:0] | read\_addr\_o | ram读取地址 |

该模块是运算模块，进行各种指令的运算，需要注意的是进行有符号运算的时候使用了$signed，可以直接进行有符号拓展。核心代码：



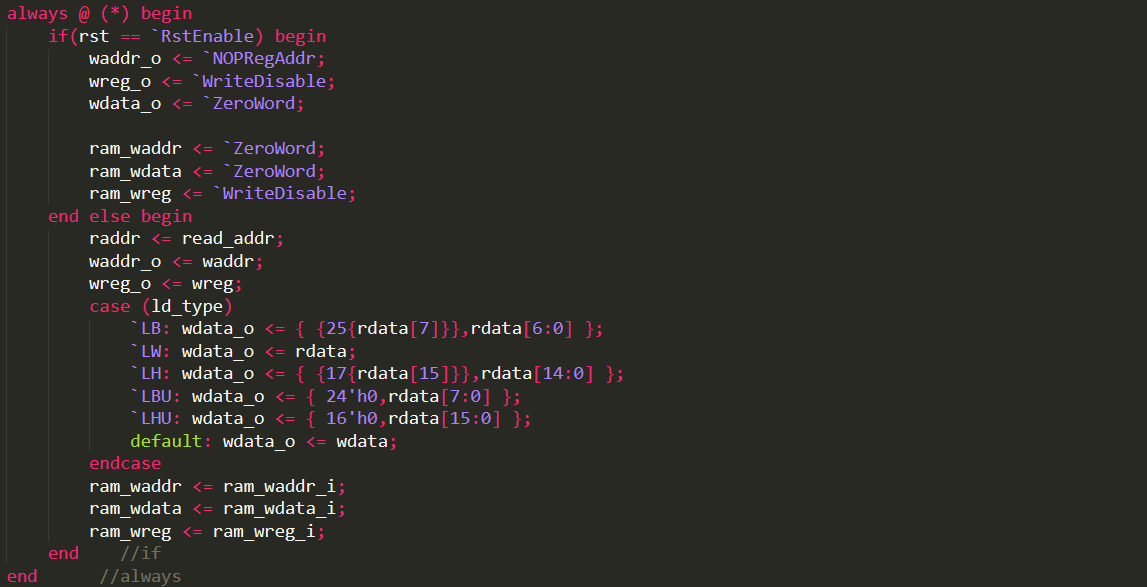
|  |  |  |  |
| --- | --- | --- | --- |
| regfile.v | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | clk | 时钟信号 |
| input | [0:0] | rst | 复位信号 |
| input | [4:0] | raddr1 | 读端口1地址 |
| input | [4:0] | raddr2 | 读端口2地址 |
| input | [0:0] | re1 | 读端口1使能 |
| input | [0:0] | re2 | 读端口2使能 |
| input | [31:0] | wdata | 写数据 |
| input | [4:0] | waddr | 写地址 |
| input | [0:0] | we | 写使能 |
| output | [31:0] | rdata1 | 读端口1数据 |
| output | [31:0] | rdata2 | 读端口2数据 |

该模块是寄存器模块，负责读取和写回寄存器的数据。在该模块处理了数据相关的冲突，如果需要读取的代码是刚要写入的代码，那么就直接将待写的数据送到输出。代码如下：



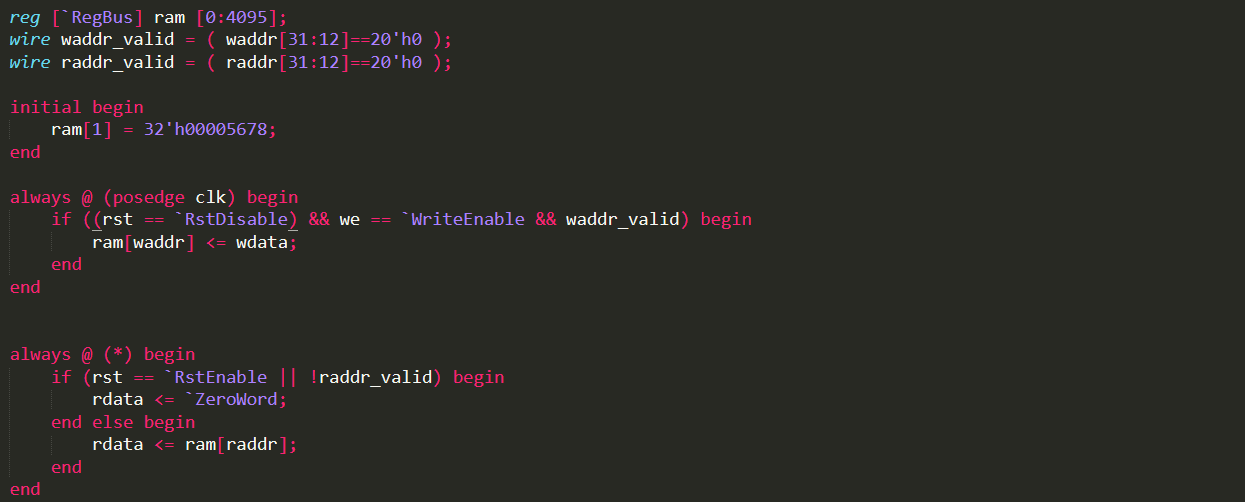
|  |  |  |  |
| --- | --- | --- | --- |
| mem.v | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | rst | 复位信号 |
| input | [4:0] | waddr | 寄存器写地址 |
| input | [0:0] | wreg | 寄存器写使能 |
| input | [31:0] | wdata | 寄存器写数据 |
| input | [2:0] | ld\_type | load指令子类型 |
| input | [31:0] | read\_addr | ram读取地址 |
| input | [31:0] | rdata | ram读取到的数据 |
| input | [31:0] | ram\_wdata\_i | 写回ram的数据 |
| input | [31:0] | ram\_waddr\_i | 写回ram地址 |
| input | [0:0] | ram\_wreg\_i | ram写使能 |
| output | [31:0] | waddr\_o | 寄存器写地址 |
| output | [0:0] | wreg\_o | 寄存器写使能 |
| output | [31:0] | wdata\_o | 寄存器写数据 |
| output | [31:0] | ram\_waddr | 写回ram的地址 |
| output | [31:0] | ram\_wdata | ram写数据 |
| output | [0:0] | ram\_wreg | ram写使能 |
| output | [31:0] | raddr | ram读取地址 |

该模块是存储模块，负责将数据写到ram中，或者从ram中读取数据保存到寄存器中。如果是读取数据load指令的话，需要判断子类型ld\_type，分有符号和无符号，字节、半字和字读取。核心代码如下：



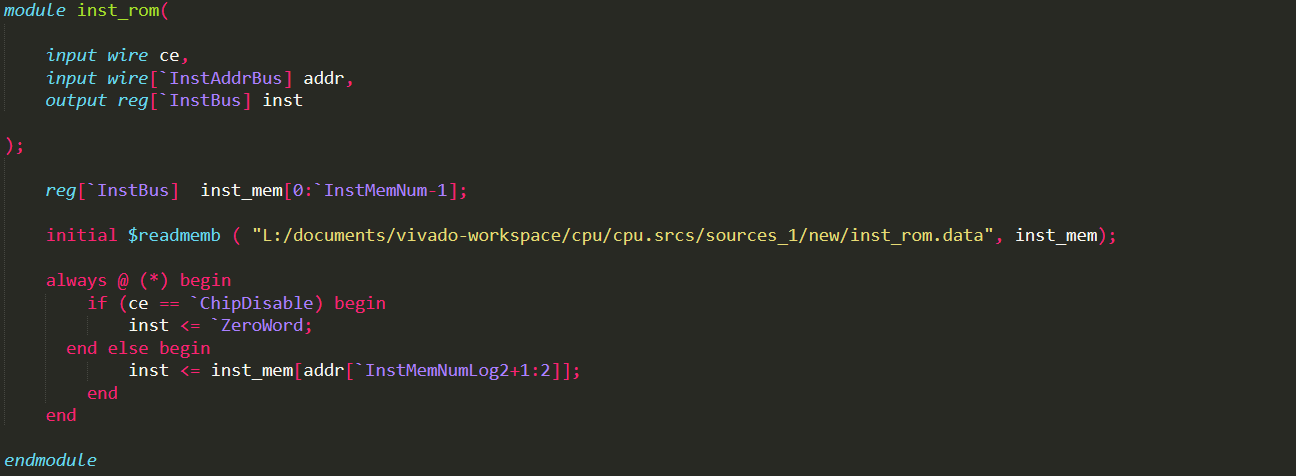
|  |  |  |  |
| --- | --- | --- | --- |
| dataram.v | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | clk | 时钟信号 |
| input | [0:0] | rst | 复位信号 |
| input | [0:0] | we | 写使能 |
| input | [31:0] | wdata | 写数据 |
| input | [31:0] | waddr | 写地址 |
| input | [31:0] | raddr | 读地址 |
| output | [31:0] | rdata | 读数据 |

该模块是内存模块，内存数据的读取和写入。这个模块需要注意的是，由于内存只分配了4096个字（12位），因此如果读取的地址或写入的地址高20位不全为0的话，就是无效地址不进行任何处理。代码如下：



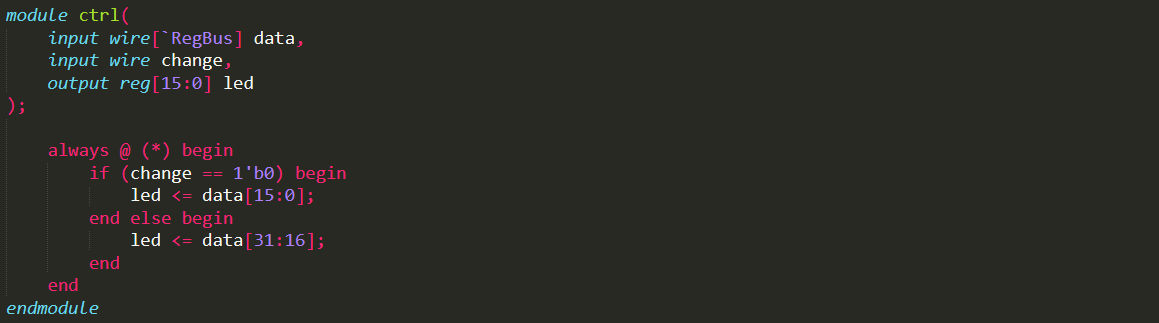
|  |  |  |  |
| --- | --- | --- | --- |
| inst\_rom.v | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | ce | 使能信号 |
| input | [31:0] | addr | 读取地址 |
| output | [31:0] | inst | 输出的指令值 |

该模块是指令存储器，通过输入的指令地址输出对应指令。代码如下：



|  |  |  |  |
| --- | --- | --- | --- |
| ctrl.v | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [31:0] | data | 指令的值 |
| input | [0:0] | change | 开关信号 |
| led | [15:0] | led | 输出的led信号 |

该模块是FPGA的连接控制模块，我连接到FPGA的是指令的值，连接到的器件是16个led灯，但是指令是32位的，所以还有个开关连接到change信号上，控制高低电平，高电平就显示高16位，低电平就显示低16位，输出的led就是FPGA上展示的高低16位指令值。代码如下：

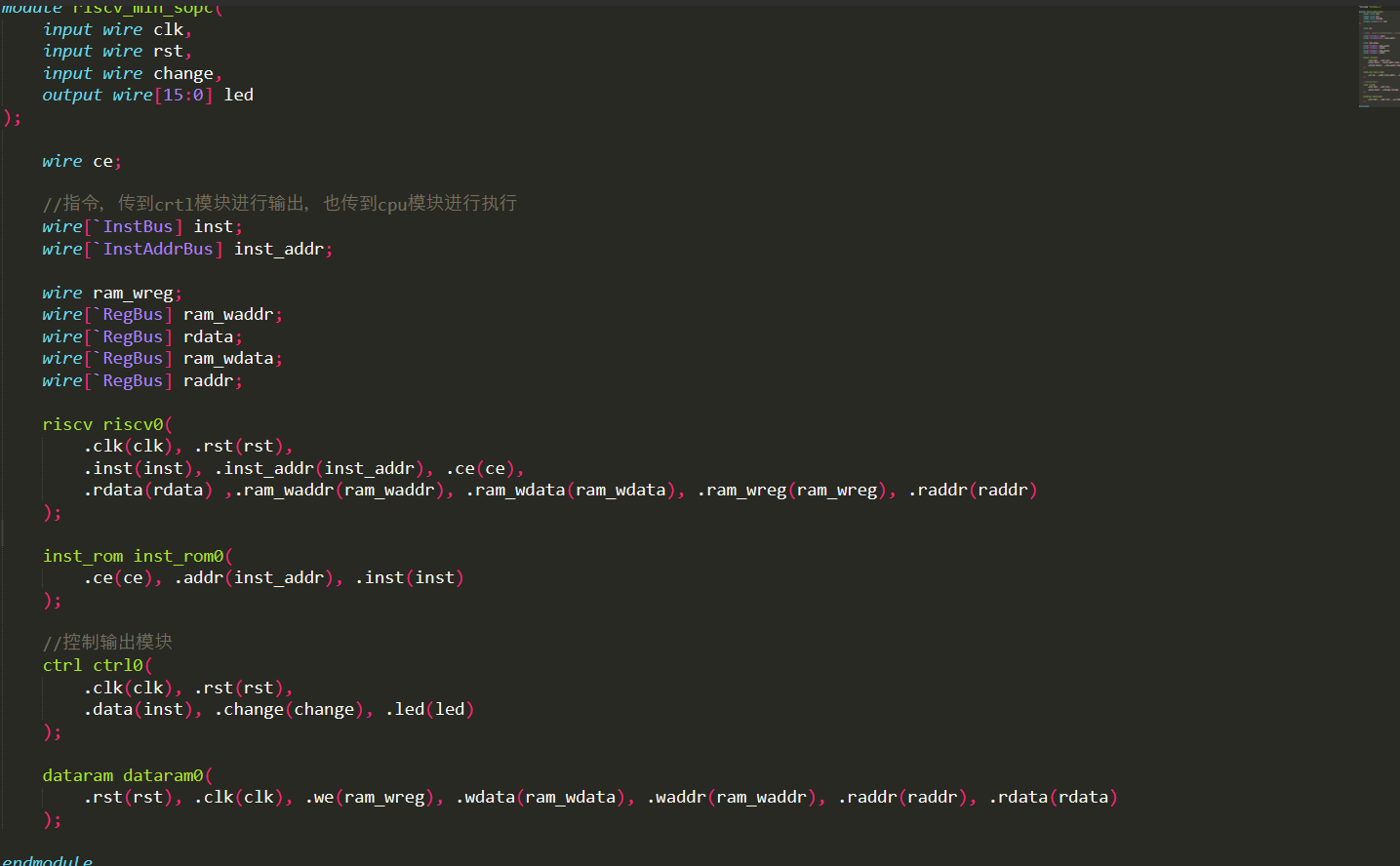


|  |  |  |  |
| --- | --- | --- | --- |
| riscv.v | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | clk | 时钟信号 |
| input | [0:0] | rst | 复位信号 |
| input | [31:0] | inst | 输入的指令值 |
| input | [31:0] | rdata | 输入的ram数据 |
| output | [0:0] | ce | inst\_rom使能信号 |
| output | [31:0] | inst\_addr | 指令地址 |
| output | [31:0] | ram\_wdata | ram写数据 |
| output | [31:0] | ram\_waddr | ram写地址 |
| output | [0:0] | ram\_wreg | ram写使能 |
| output | [31:0] | raddr | ram读取地址 |

该模块是risc-v cpu的综合模块，实例化各个模块并连接各个模块的线。代码略。

|  |  |  |  |
| --- | --- | --- | --- |
| riscv\_min\_sopc | | | |
| 输入/输出 | 宽度 | 信号名 | 说明 |
| input | [0:0] | clk | 时钟信号 |
| input | [0:0] | rst | 复位信号 |
| input | [0:0] | change | 高低电平 |
| output | [15:0] | led | 输出的led信号 |

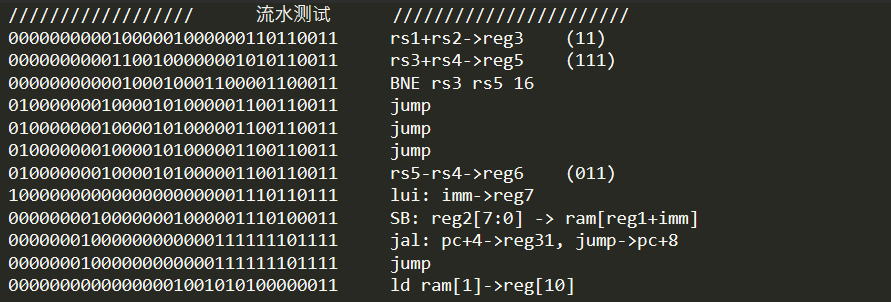
该模块是整合模块，连接CPU和指令寄存器和内存ram。输入和输出都连接到FPGA上来测试。clk信号连接一个按钮，rst信号连接一个开关，change信号连接一个开关，led信号连接16个led灯。代码如下：



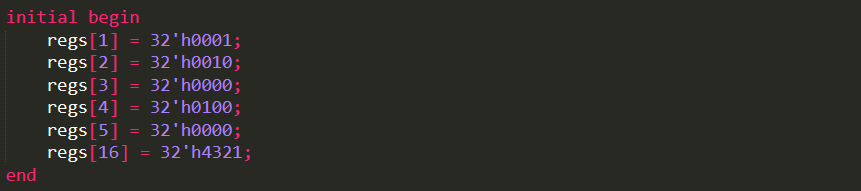
# 四、实验结果

## 仿真

我设计了12条指令对该CPU进行测试，具体测试指令如下：

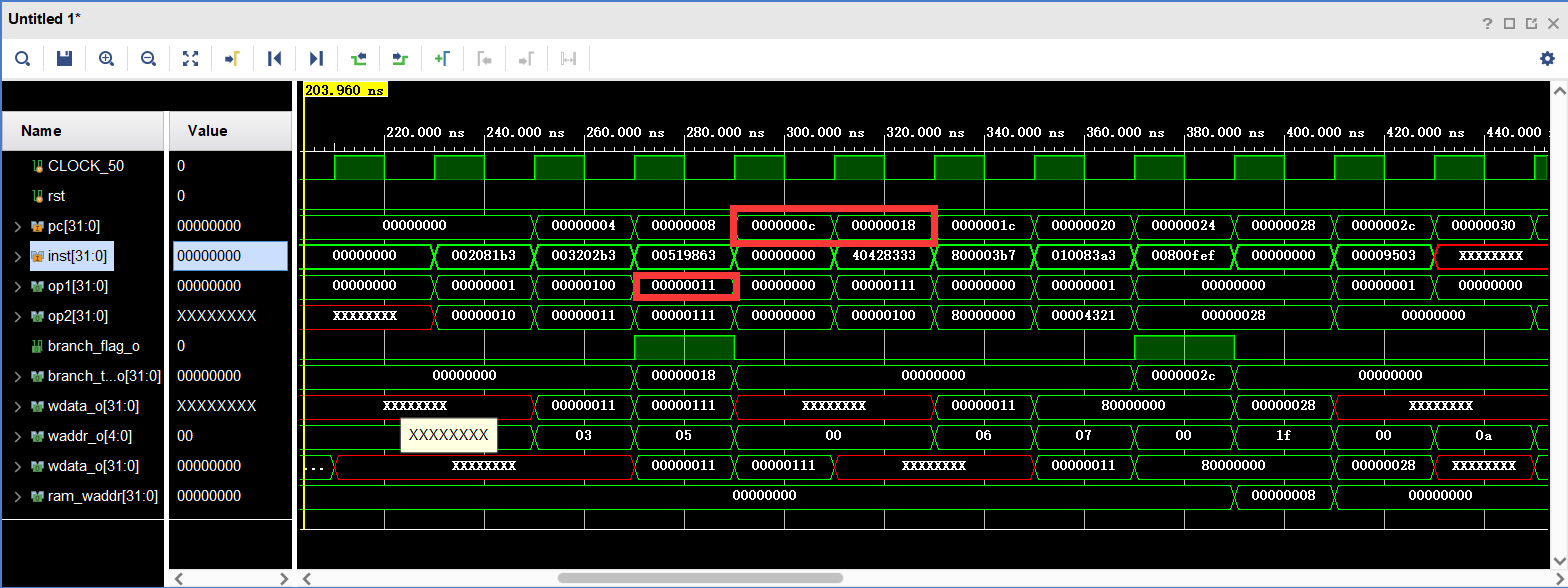


寄存器的初始化如下：



该测试代码涵盖了所有类型的指令，并且测试了流水的数据冲突，以及转移指令。

仿真结果：

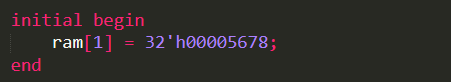


第一条指令将reg[1]+reg[2]送到reg[3]存储，第二条指令将reg[3]+reg[4]送到reg[5]中存储，由于原本reg[3]放的是0，在图中可以看到取得的reg[3]是00000011，而不是0，说明数据相关的冲突得到了解决。

第三条指令是转移指令，比较reg[3]和reg[5]中的值，如果不相等则转移到以pc位基址，imm为偏移量的地址去，这里imm是16，此时reg[3]为00000011，而reg5为0，所以肯定会转移，在图中圈起来的地方可以看到，地址有0000000c变成了00000018，而0000000c下方的指令值inst变成了0，说明转移成功。

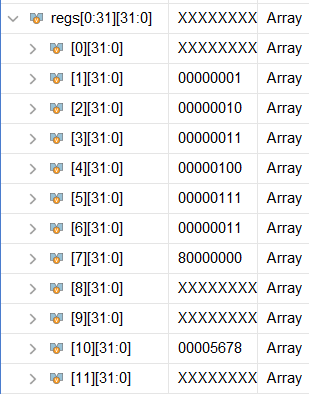
中间几条指令是保存指令就不解释，等下看regfile的值就能知道是否保存。下面看倒数第三条指令，这条指令是jal指令，跳转到pc+imm处，并将pc+4保存到寄存器中去，一般返回地址都是保存在第31个，所以我就保存到第31寄存器中。在图中可以看到，在pc位00000028的位置下方，inst的值为0，说明跳转成功，而跳转是加8，所以pc的值看起来并没有变换。

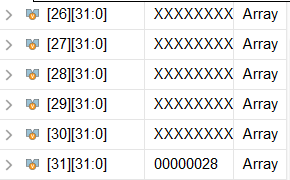
接下来执行最后一条指令load指令，将ram[1]中的值存储到reg[10]中，存储的是半字。



我们将ram[1]初始化为上述的值，半字也就是5678，reg[10]中的内容将会是5678。

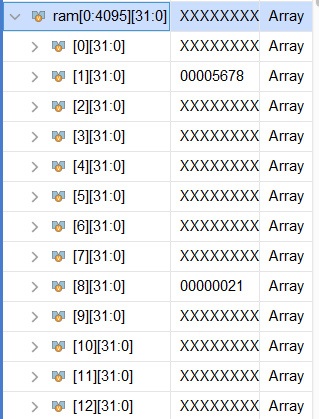
下面来看regfile中的值：





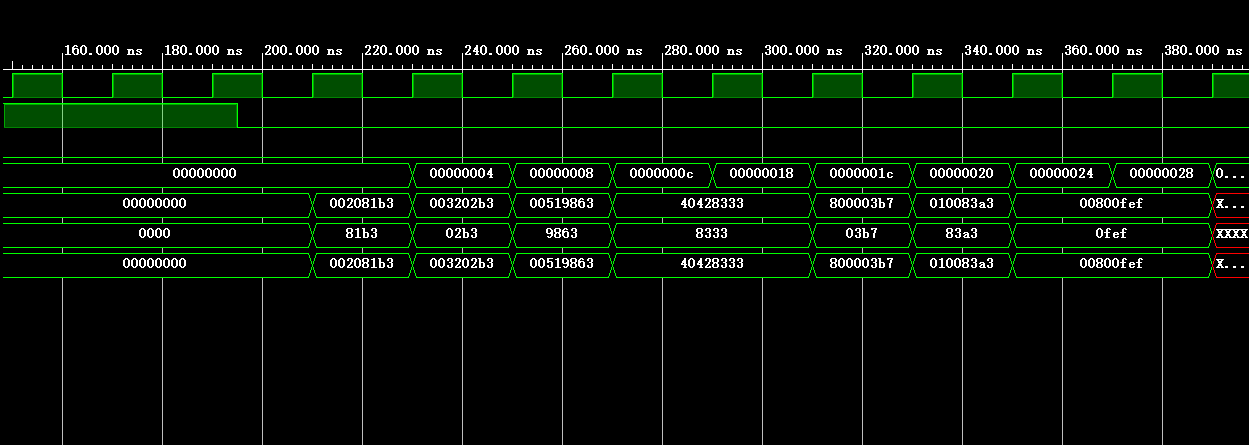
可以看到reg[10]的值是5678，reg[31]的值是28，也就是24+4，也是正确的。

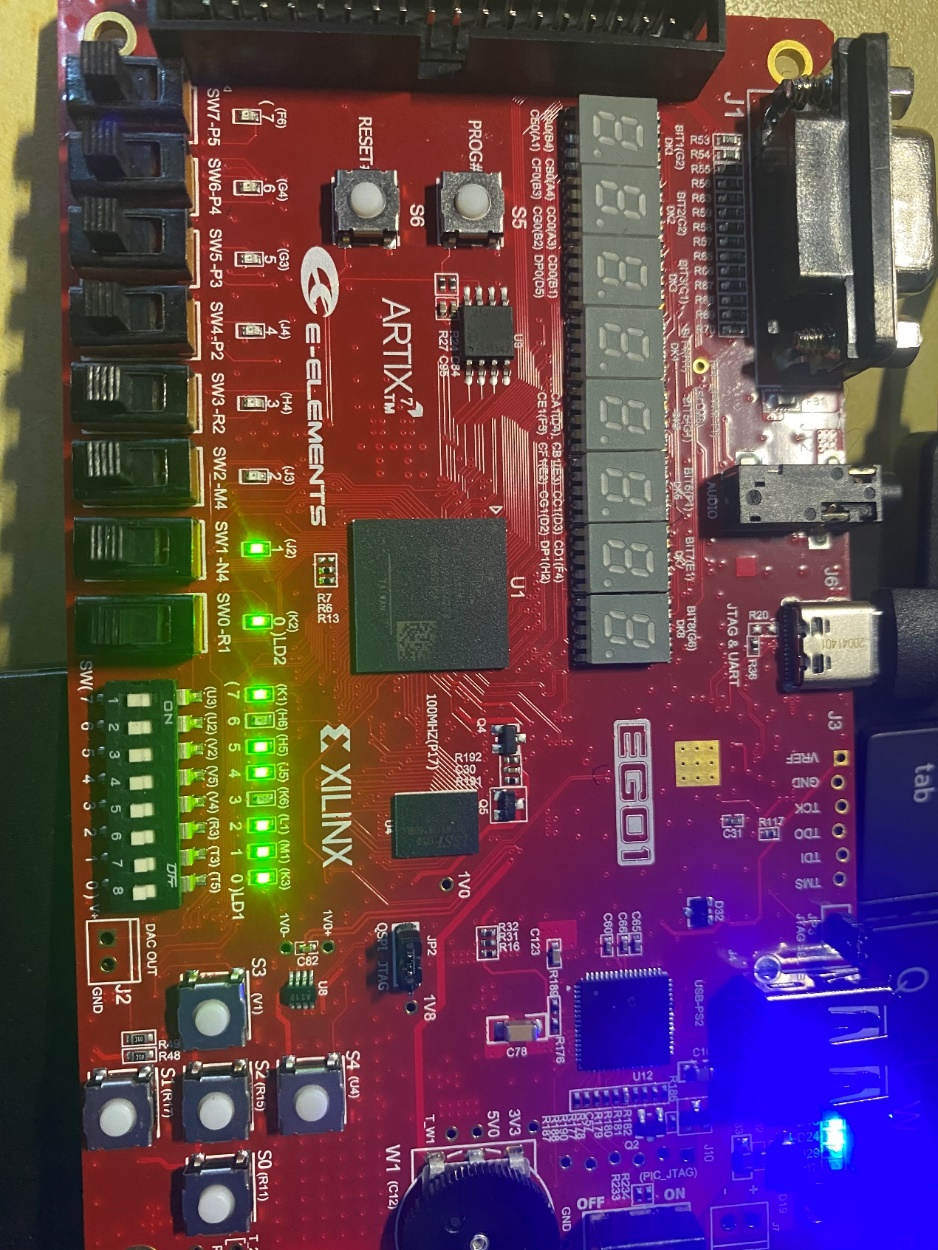
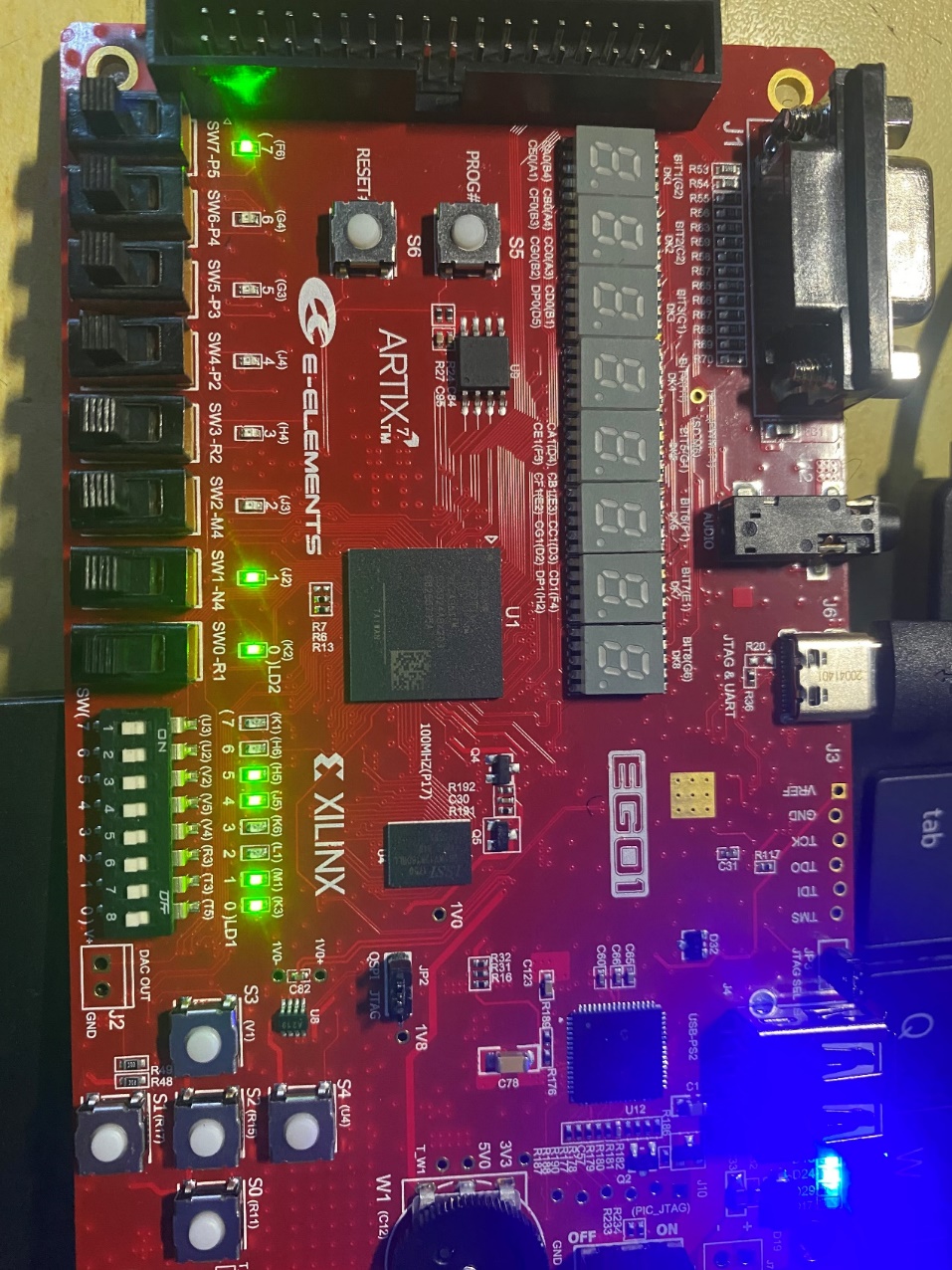
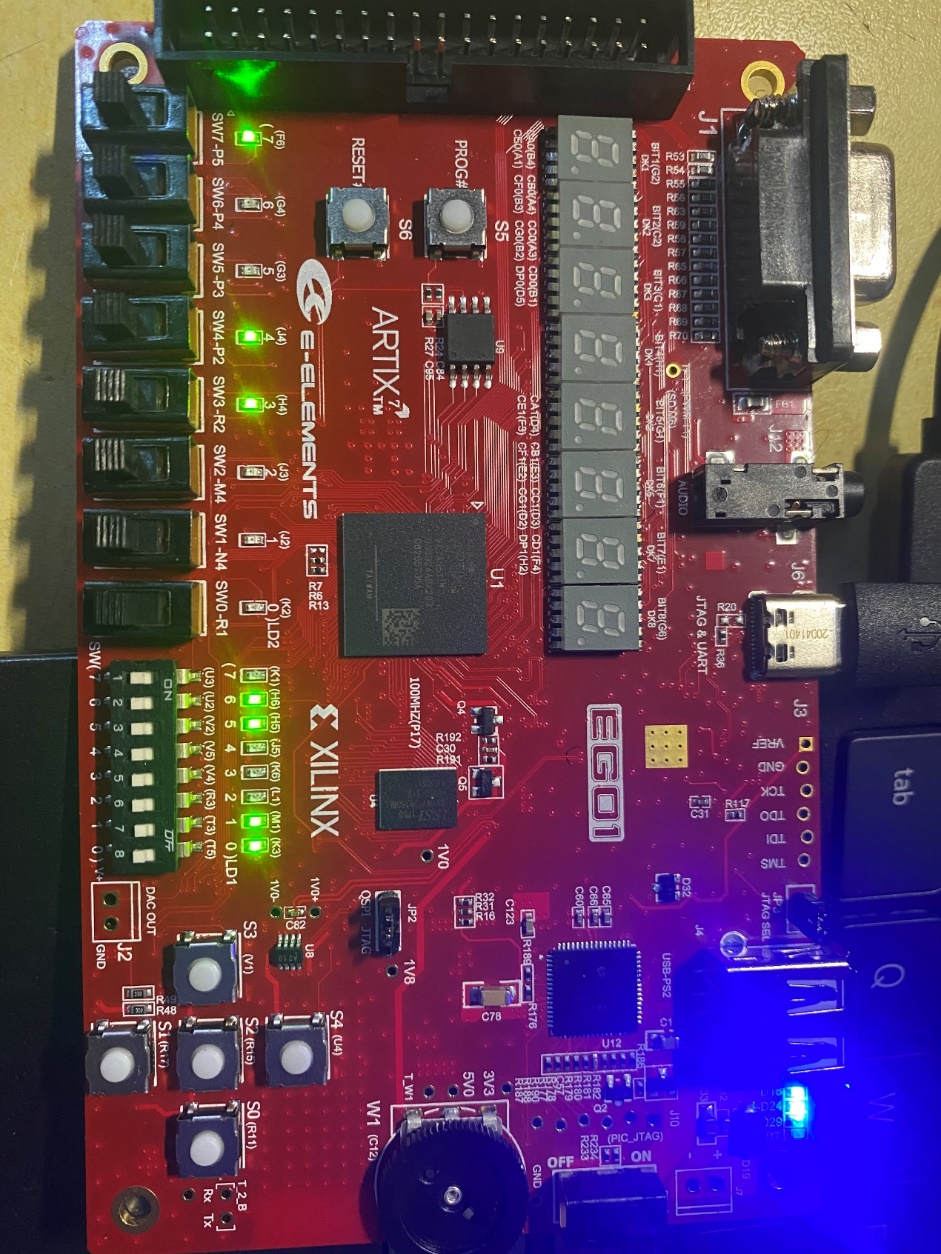
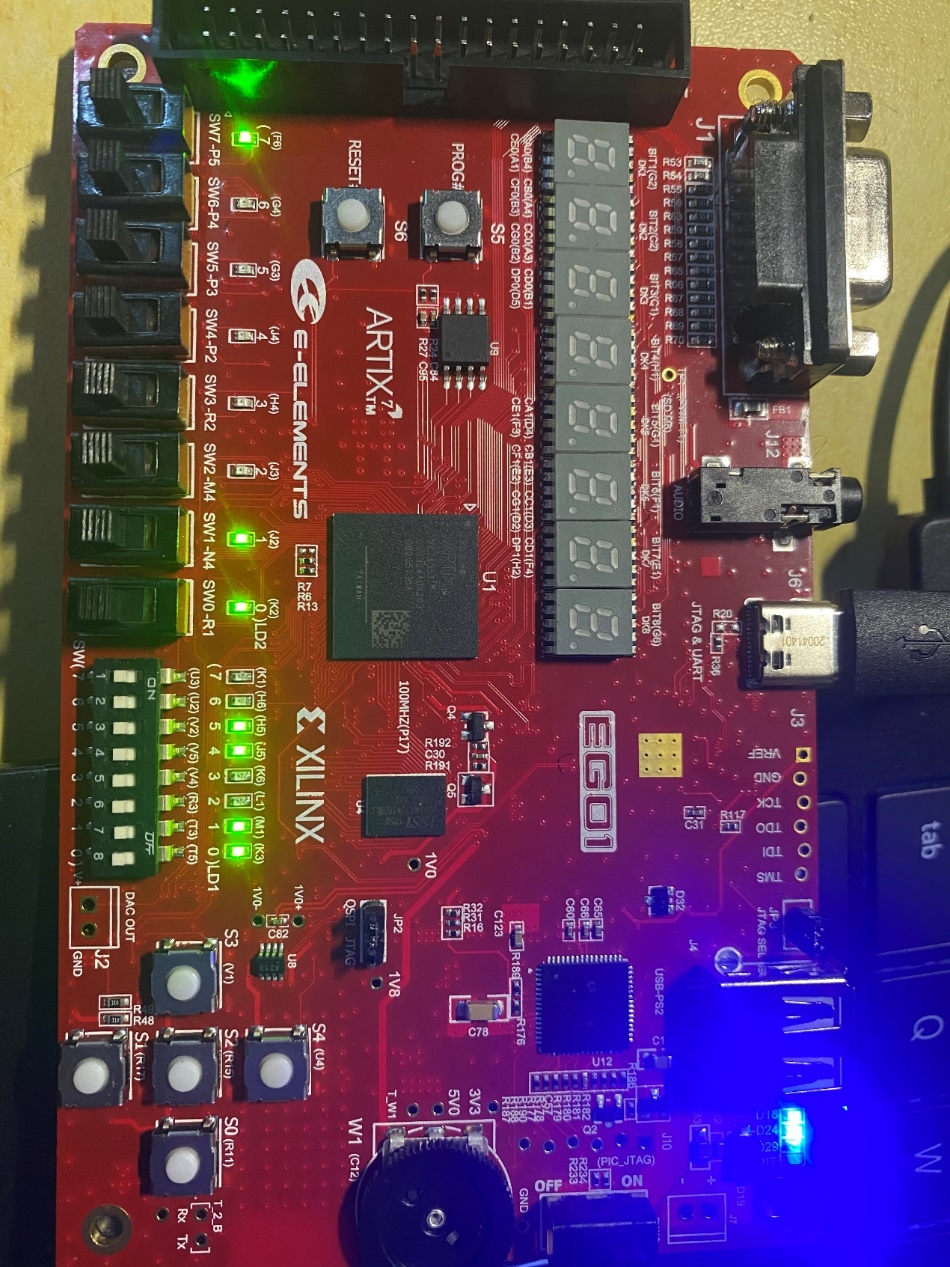
最后看一下ram中的数据，SB将reg[16]保存到ram中，类型是字节，

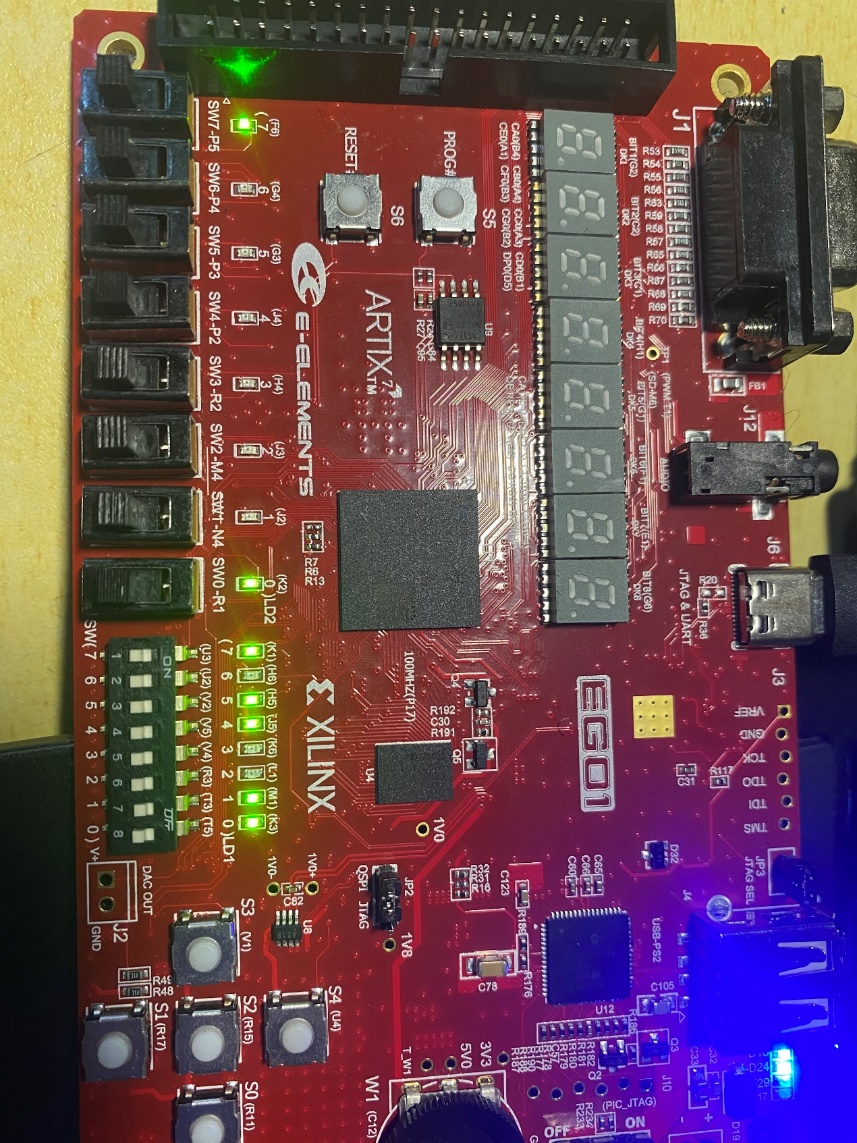


看到ram[8]是21，结果正确。

## FPGA验证：







# 五、实验总结

这个CPU设计对我的收获非常巨大，不仅让我重温并更加深刻理解了计算机组成原理的知识，还再一次熟悉掌握了Verilog硬件描述语言，并且还学习了FPGA的使用。

刚开始的时候我完全不知道该如何下手写这个CPU，觉得那个自己动手写CPU也太多了，肯定看不完看不懂，但是被逼无奈，不看更不会写。于是我就花了整整三天时间把这本书看到了乘除模块的编写，边看边敲代码理解。到了乘除模块不太看得懂也不太愿意看了。于是就停下打算就去验收了算了。但是看到很多人都是这样验收的，觉得这样就算我不看这个书，直接把代码复制过去验收，也能验收啊，那我不是白看了书吗？所以我又觉得不甘心，这样去验收肯定不会有很好的成绩。

那天是星期三，看到他们都去验收了，我就很心急，当天晚上就开始自己写一个CPU，然后网上就去找RISC-V的指令集，开始自己写一个CPU。然后就写出了这个CPU。花了我一天一夜的时间，从星期三晚上九点开始写，写到星期四晚上十二点，中间只睡了五个小时，上了一节课。之后验收前还花了一晚上时间增加了几条指令。当自己写完CPU的时候，心里是很高兴的，也是很自豪的，这是我自己写出来的CPU！虽然流水线与那本教程上的是一样的，但是内部的逻辑其实都是我自己的，并且很多教程上麻烦的代码都被我自己给改进改的简单了。

总的来说，这个CPU花费了我大量的时间，也让我学到了大量的知识。当初学习计算机组成原理的时候完全不懂数据前推是什么概念，只知道处理冲突有这个方法，考试的时候写这个就对了，写完这个CPU后就理解了数据前推的作用和原理。在我设计的这个CPU中其实还有一点不完整，就是LOAD指令的数据冲突作用，没有考虑进去，因为当时写的时候没想到会有冲突。不过我也想到了一种解决方案，下面描述一下该解决方案。由于load指令会将数据加载到寄存器中，而回写到寄存器还需要一个时钟周期，所以如果下一个周期有指令读取写的数据的话，是不能正确读取的。这时候可以做如下修改：在译码阶段可以直接将load指令需要的参数直接送到mem模块（记得一个词叫旁路技术，不知道这是不是），然后从mem模块接一根数据线到译码模块，作用就是数据前推，设置一个标志，如果需要数据前推就直接使用mem传来的数据，而不使用读取到的数据。我认为这个解决方案是可行的。

最后，这个报告也花了我一整天的时间，画那个电路图找了很多工具，最后是用draw.io绘制的。