# Li RISC-V

[Li RISC-V 1](#_Toc100927554)

[1内核部分（8周期版本） 2](#_Toc100927555)

[1.1读指令逻辑 2](#_Toc100927556)

[1.2PC管理逻辑 3](#_Toc100927557)

[1.3算数逻辑单元 5](#_Toc100927558)

[1.4解码逻辑 5](#_Toc100927559)

[1.5通用寄存器 6](#_Toc100927560)

[1.6控制状态寄存器（中断） 6](#_Toc100927561)

[1.6.1中断流程 12](#_Toc100927562)

[1.6.2中断处理流程 12](#_Toc100927563)

[1.6.3发生中断时core和csr需要做的事 14](#_Toc100927564)

[1.6.4中断返回mret 15](#_Toc100927565)

[2总线部分 15](#_Toc100927566)

[2.1总线优先权裁决 17](#_Toc100927567)

[2.2总线路由地址 17](#_Toc100927568)

[2.3实际总线连接 20](#_Toc100927569)

[3定时器部分 21](#_Toc100927570)

[3.1定时器寄存器 21](#_Toc100927571)

[3.2定时器计数和中断触发部分 23](#_Toc100927572)

[3.3串口部分 23](#_Toc100927573)

[3.4串口发送 24](#_Toc100927574)

[3.5串口接收 24](#_Toc100927575)

[3.6串口接收中断 26](#_Toc100927576)

[3.7串口下载程序 26](#_Toc100927577)

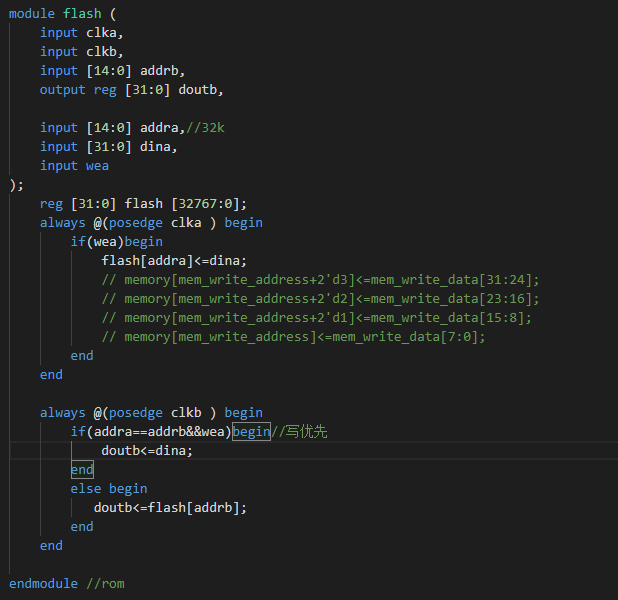
[4移植到FPGA平台 30](#_Toc100927578)

## 1内核部分（8周期版本）

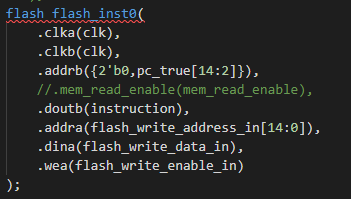
内核部分主要的目的是读指令、解码指令（通用寄存器读地址：reg\_read\_a\_address，reg\_read\_b\_address, 通用寄存器写地址:reg\_write\_address,通用寄存器写数据：reg\_write\_data,通用寄存器写使能：reg\_write\_enable，alu算数逻辑单元的操作数alu\_opa，alu\_opb，操作alu\_op,读内存地址：core\_read\_address\_out写内存地址：core\_write\_address\_out写内存数据:core\_write\_data\_out写内存使能:core\_write\_enable\_out读控制状态寄存器地址csr\_read\_address：写控制状态寄存器地址csr\_write\_address：写控制状态寄存器数据：csr\_write\_data写控制状态寄存器使能:csr\_write\_enable）、控制pc。

### 1.1读指令逻辑

和vivado的block mem gen 端口兼容

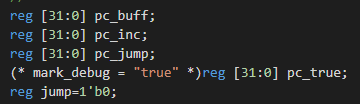


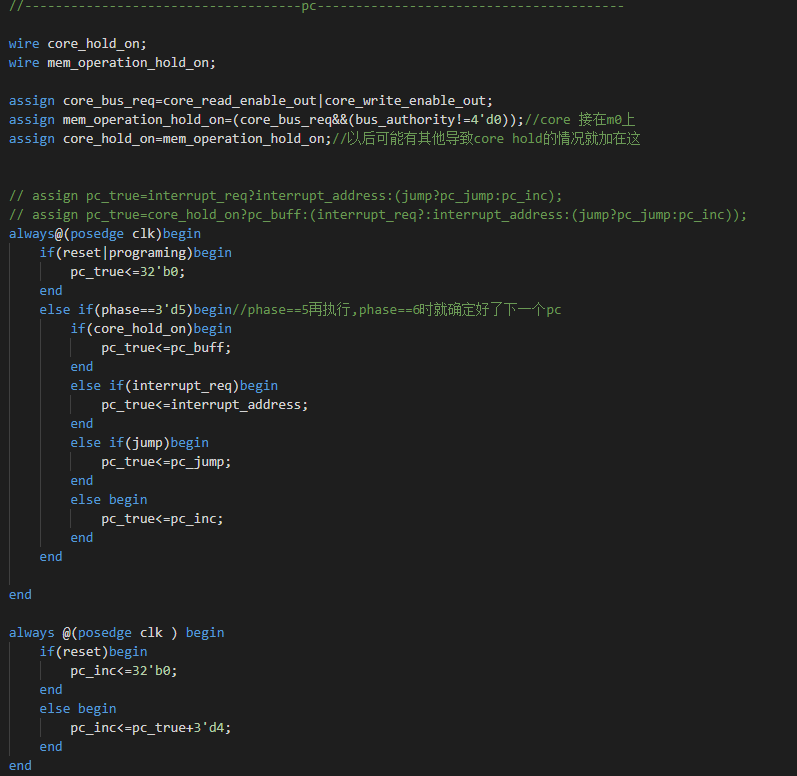
Flash模块里装的指令，clka的上升沿将数据写入flash[addra]，clkb的上升沿将flash读地址addrb的数据寄存到doutb端口，如果对flash的同一个地址同时读写的话，将要写入的数据放到读端口，并写入。则pc应连到addrb。

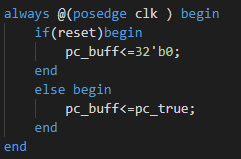


addrb连的{2‘b0，pc\_true[14:2]}因为一个指令四个字节，flash中也是一个地址装的四字节，右移两位刚好除以4，pctrue加4对应flash读地址加1。

### 1.2PC管理逻辑







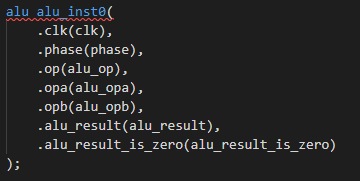
Pc true是真实要读取下个指令的地址（比如中断地址，跳转地址），pc buff是当前指令对应地址，pc inc是当前pc对应的下一个地址（比如当前pc为32‘d4，则pc inc为32‘d8）。Pc jump为程序跳转地址（比如jarl，beq）。

Pc true在phase=5的时钟上升沿更新，先看内核是不是被暂停了（比如读写外设但总线控制权不在这个内核，内核就会被暂停，TODO有问题是按单周期的写的），被暂停了pc true就为当前指令pc，然后看是否有中断请求，有就让pctrue为中断入口地址，再看是否需要跳转，要跳转就让pctrue为pc jump，否则pctrue为pc inc即下一条指令地址。

Todo：pc inc和pc buff加上phase约束。

### 1.3算数逻辑单元

Phase是8周期的phase，alu在phase=2的上升沿更新计算结果（alu result is zero这个信号没用到，可以删掉）



### 1.4解码逻辑

Phase=7的时钟上升沿，确定

reg\_read\_a\_address,reg\_read\_b\_address,csr\_reg\_read\_address

Phase=0得到

reg\_read\_a\_address,reg\_read\_b\_address, csr\_reg\_read\_address

Phase=1 得到

reg\_read\_a\_data.reg\_read\_b\_data, csr\_reg\_read\_data

Phase=1的上升沿，确定

alu\_opa,alu\_opb,alu\_op

Phase=2得到

alu\_opa,alu\_opb,alu\_op

Phase=3得到

alu\_result

Phase=3的上升沿，确定

core\_read\_address

Phase=4 得到

core\_read\_address

Phase=4 的上升沿 确定

Pc\_jump，jump,

csr\_reg\_write\_address, csr\_reg\_write\_data,csr\_reg\_write\_enable,

Phase=5 得到

core\_read\_data

pc\_jump jump

csr\_reg\_write\_address, csr\_reg\_write\_data,csr\_reg\_write\_enable,

执行写控制状态寄存器操作。

Phase=5的上升沿 确定

reg\_write\_address,reg\_write\_data,reg\_write\_enable

core\_write\_address,core\_write\_data,core\_write\_enable,

Phase=6 得到

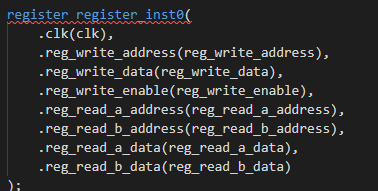
reg\_write\_address,reg\_write\_data,reg\_write\_enable, core\_write\_address,core\_write\_data,core\_write\_enable

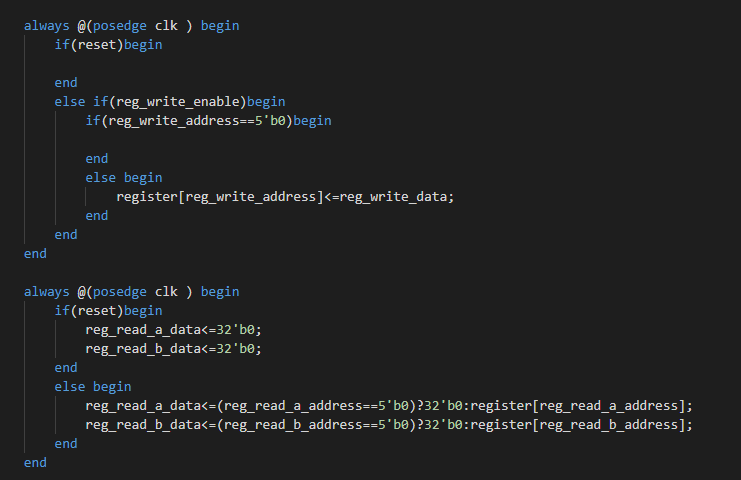
Phase=7 执行 写通用寄存器操作,写外设操作

TODO：将csr的写操作也放到pahse=7。

### 1.5通用寄存器

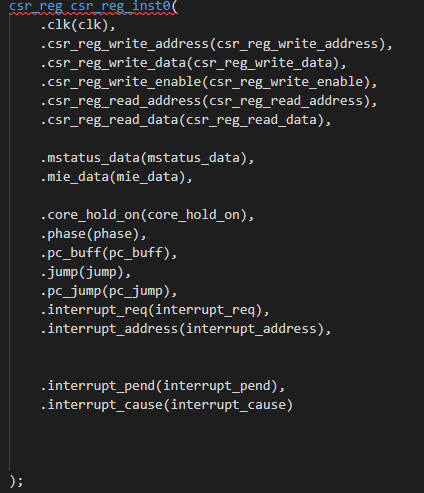
通用寄存器在core内部的连线，有两个读端口因为alu需要同时得到两个寄存器的数据



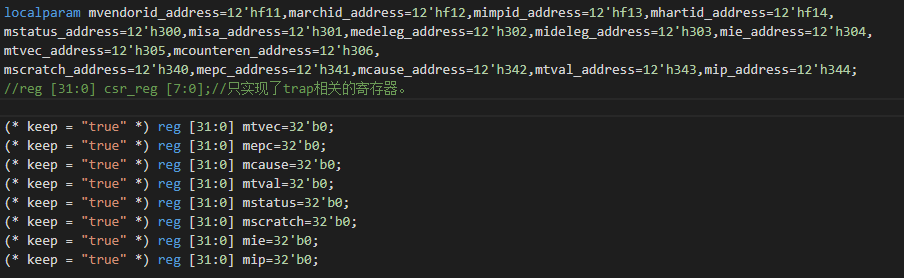
逻辑很简单，写使能写入对应寄存器，读出对应地址的寄存器内容，这里注意riscv规定通用寄存器0的值一直是0；

### 1.6控制状态寄存器（中断）

控制状态寄存器在core内部连线

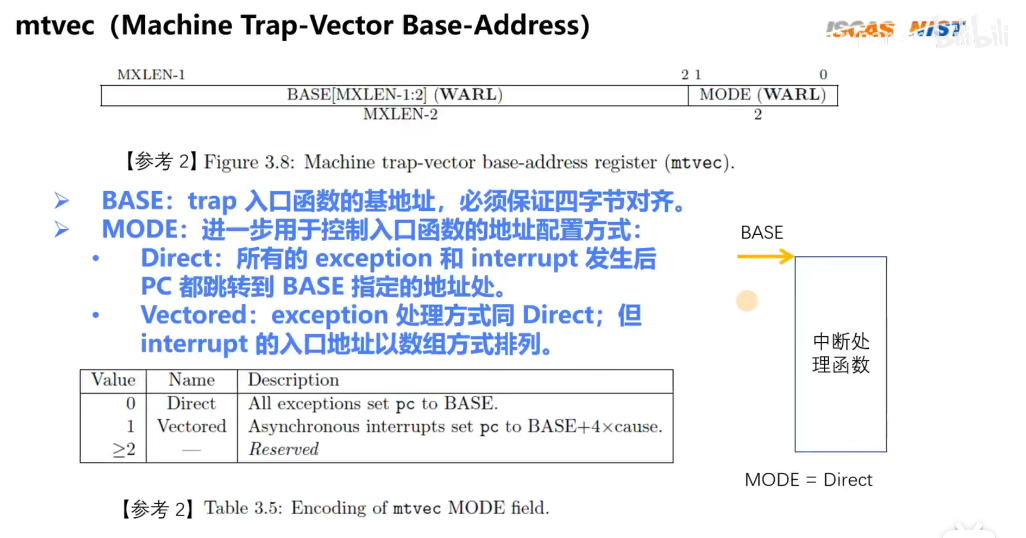


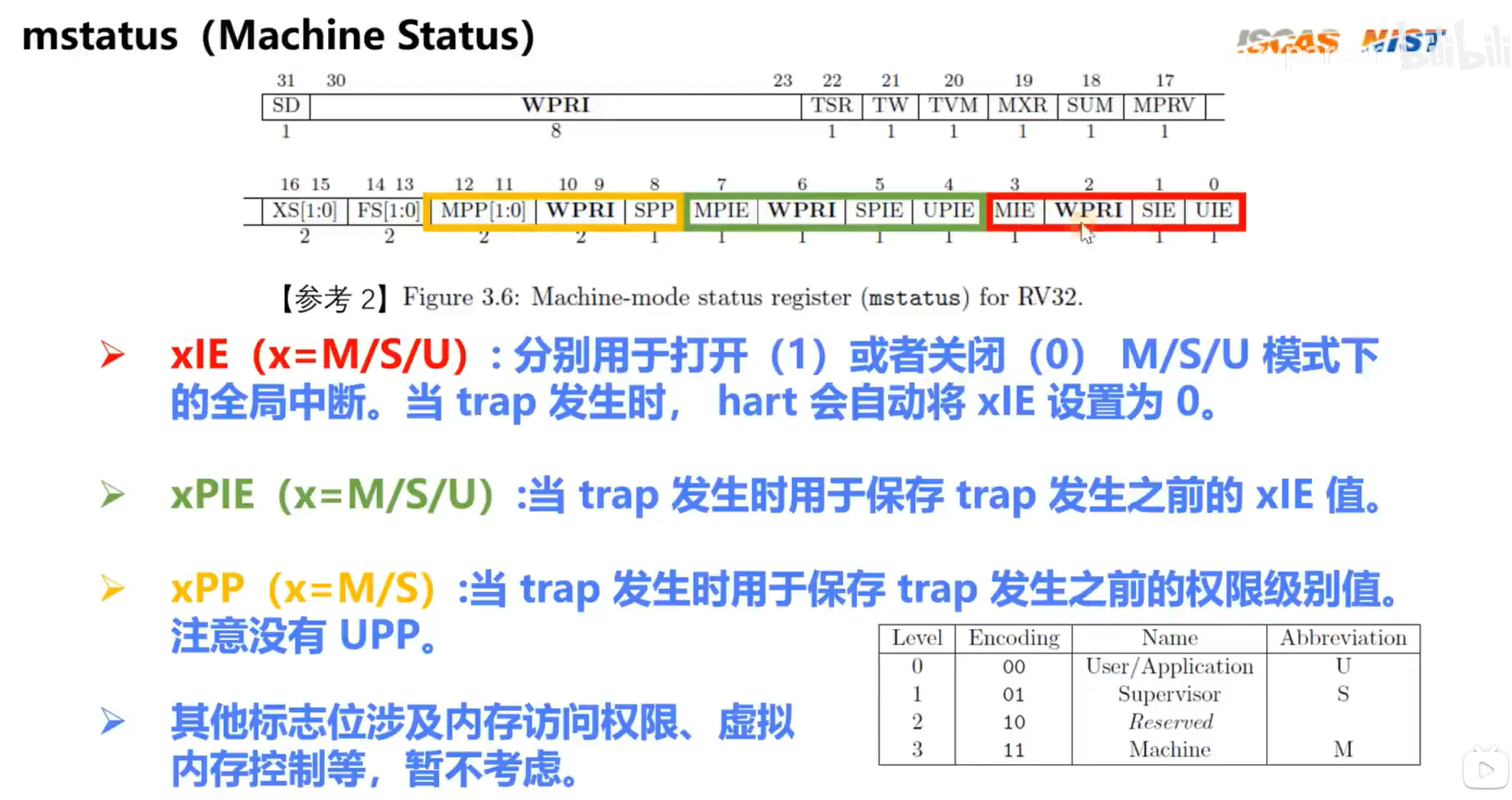
控制状态寄存器的读写和通用寄存器的逻辑是一样的，重点是中断相关的内容

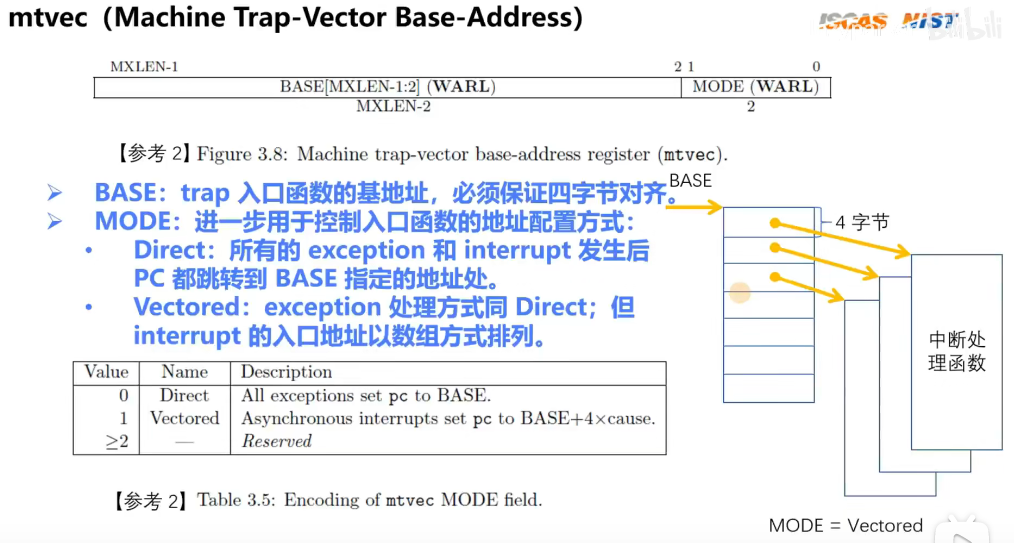


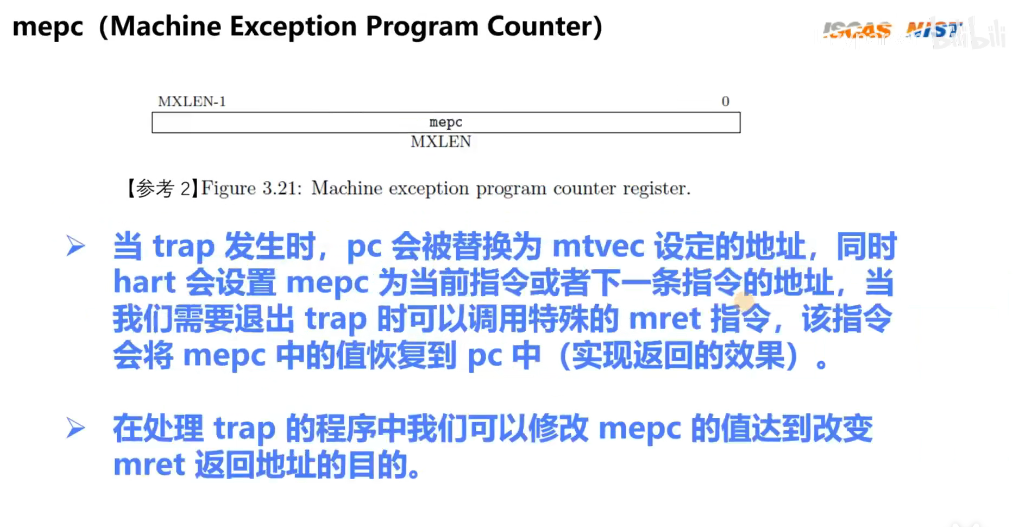
只实现了trap相关的寄存器，



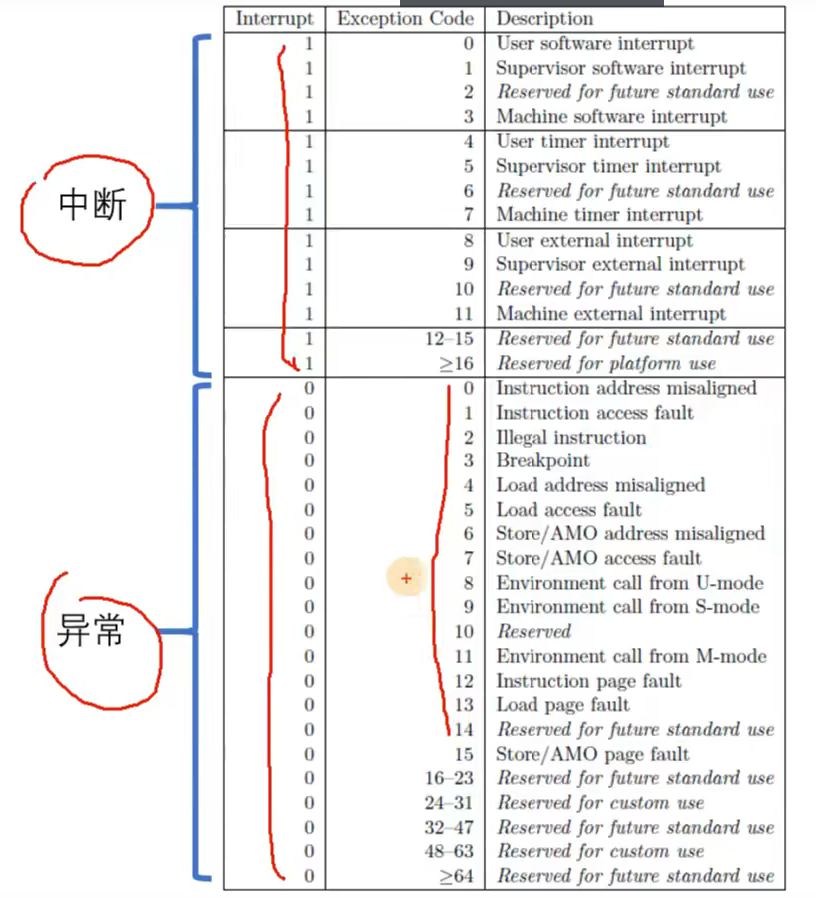


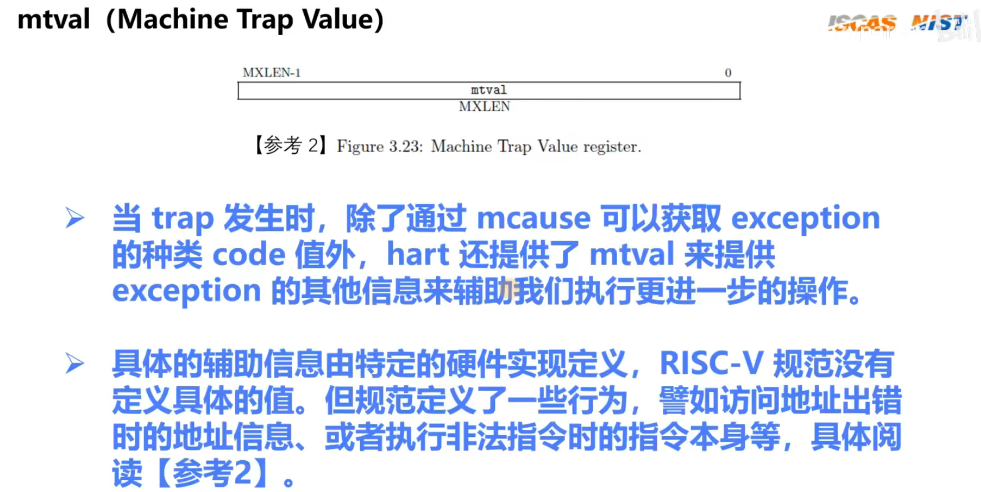


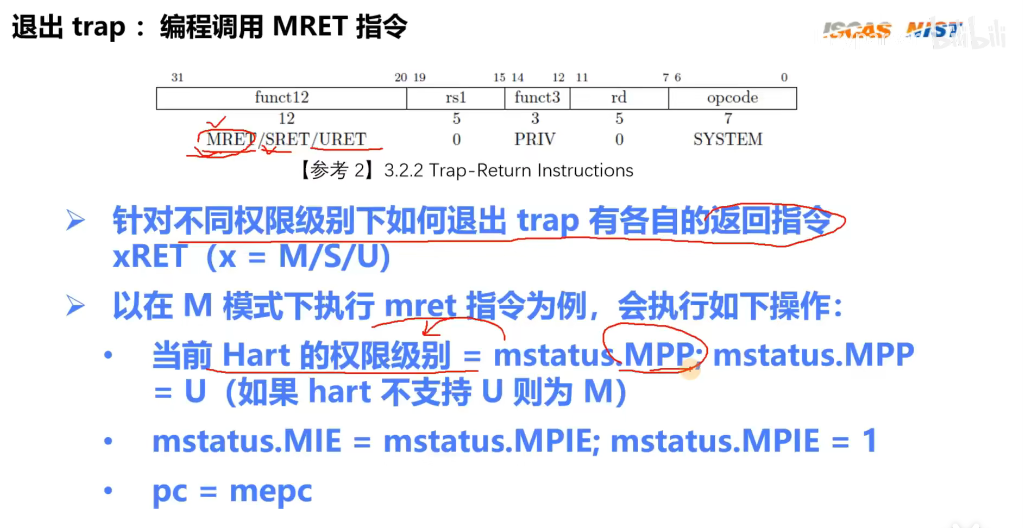


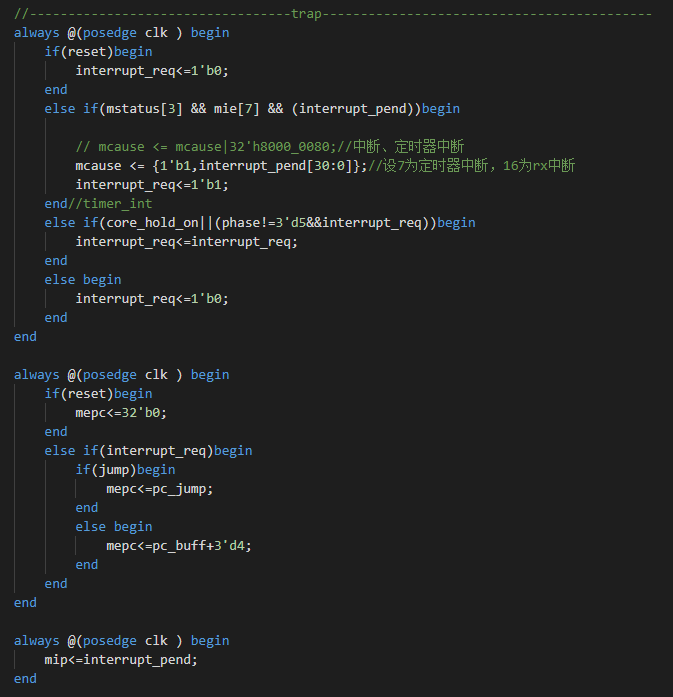










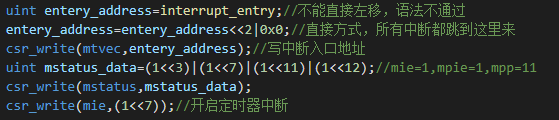


这里实现了定时器中断和串口接收中断，（mcause的最高位为1表示中断，为0表示异常）

其中定时器中断号是riscv规定的为7，大于等于16为平台使用的中断号，这里设计的串口接收中断号为16；当中断允许mstatus[3]为1和中断使能mie[7]为1和有未处理的中断interrupt\_pend不为0时，csr向core发出中断请求（目前是开环的todo 加上中断处理反馈）core在phase=5时响应中断请求，跳转到中断入口地址interrupt\_address={2'b0,mtvec[31:2]};todo：重大问题，interrupt pend是中断类型，具体中断原因要写在mcause里，不能直接把interrupt pend写给mcause，比如设置interrupt的第16位为硬件中断，那么串口，spi等的中断类型都共用这个中断类型，然后根据中断请求写mcause为串口中断还是spi中断，比如规定串口中断，mcause写0，spi中断写1。

1.6.1中断流程

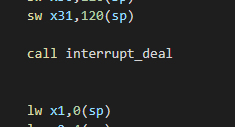
编写中断处理函数（interrupt entry），取中断处理函数地址entery\_address=entery\_address<<2|0x0;；地址四字节对齐，低两位为0表示直接方式，为1表示向量模式，（参考b站）写入mtvec。（这里的m前缀都是指机器模式）配置mstatus的中断使能位mie=1，发生中断前的中断使能位mpie=1，发生中断前的运行时特权级别mpp=11（机器模式），写入mstatus，最后开启相应中断，这里开了定时器中断（todo串口接收中断没约束在mie）

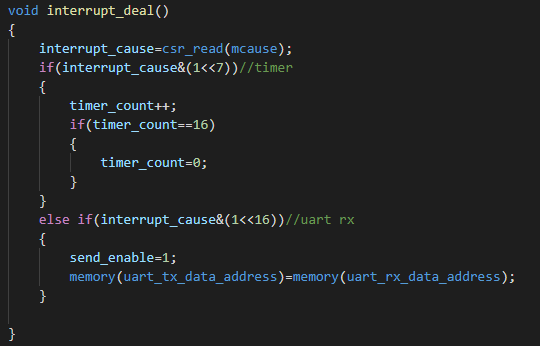


#### 1.6.2中断处理流程

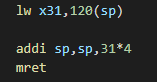
进入interrupt entry后将堆栈指针下移31\*4个字节，用来保存发生中断时通用寄存器的值，保护现场，（不是所有的寄存器都需要保存，这里只是图方便）然后执行中断interrupt deal，执行中断完后恢复现场，最后将堆栈指针加31\*4，退出中断。



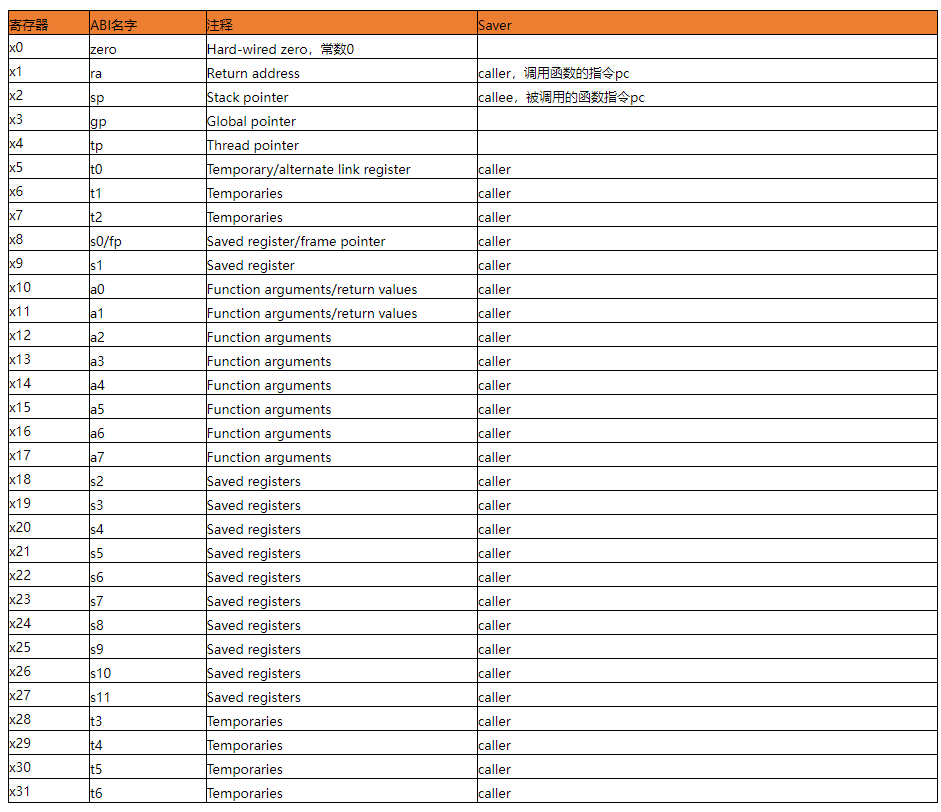




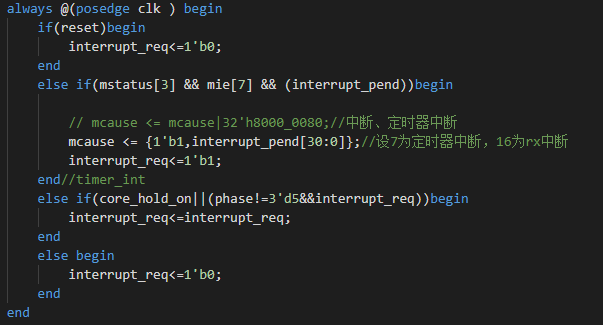
中断处理函数里先读取mcause，判断是什么中断，再做相应的处理。



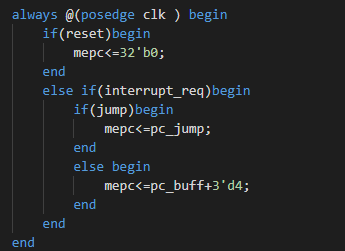
sw x1,0(sp)：将x1寄存器的值保存到sp偏移0的地址处



#### 1.6.3发生中断时core和csr需要做的事



除了上述的内容外，如果发生中断时core正被挂起，或者core的phase不在5，此时无法响应中断，则保持中断请求状态，因为外设的中断请求都只保持一个时钟周期。



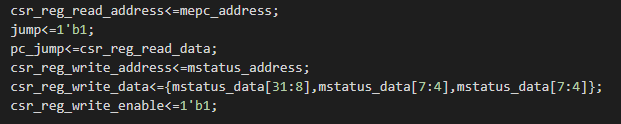
发生中断后，将中断返回地址写如mepc，如果发生中断时core正要跳转，则将跳转地址给mepc否则把发生中断时的下一条指令地址给mepc（异常是将当前指令地址给mepc，参考b站）。



中断发生后，mstatus中断允许位清零禁止中断。

#### 1.6.4中断返回mret

支持中断返回指令mepc：

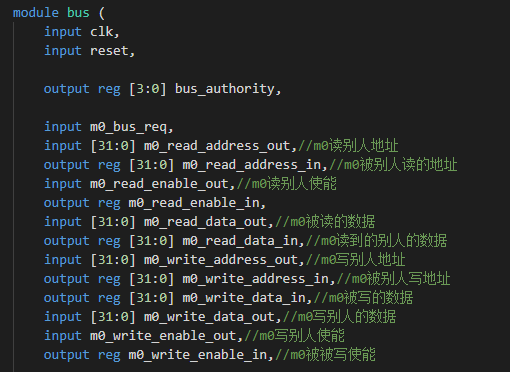


将pc jump设置为mepc指定的地址，jump置1，即跳转到mepc指定的地址，同时恢复mstatus；

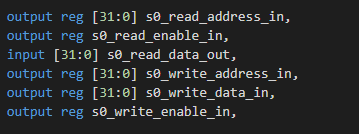
## 2总线部分

Core 读写外部设备靠总线路由，首先要确定各个外设的地址范围，举个例子，

定时器模块有两个32位寄存器，一个是控制状态寄存器，一个是计数个数寄存器，那么定时器占用了2个地址空间，32位处理器一共能访问2^32个地址空间，我们可以把这两个地址空间放在这2^32个地址空间的任意位置，这里是放在了32‘h9000\_0000和32‘h9000\_0001这两个地址空间。总线只分主机和从机，总线响应主机的读写请求并路由到（按主机的读写地址）相应的从机，比如core（主机）要读定时器（从机）的csr寄存器，core给总线的地址为32‘h9000\_0000，总线根据这个地址判断是要访问从机1（假设定时器接在总线的从机1上，从机1的地址为32‘h9000\_0000-32’h9fff\_ffff）就将core和定时器的读写端口接在了一起，就像core直连定时器一样。

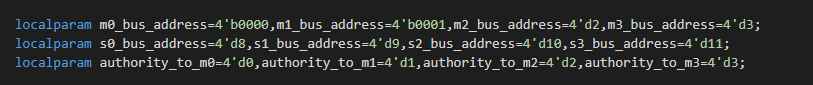


M前缀是主机



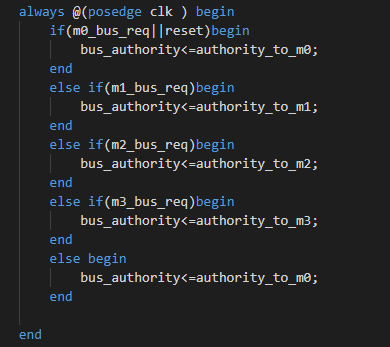
S前缀是从机

注意数据流向是设备对外->总线对内，s0\_read\_data\_out是对s0是输出，对总线是输入。



这里为了方便把2^32分成了16分，当然也可以用多少地址分配多少地址空间，每个主机或者从机占2^28个地址空间。

### 2.1总线优先权裁决



M0最高，m3最低。

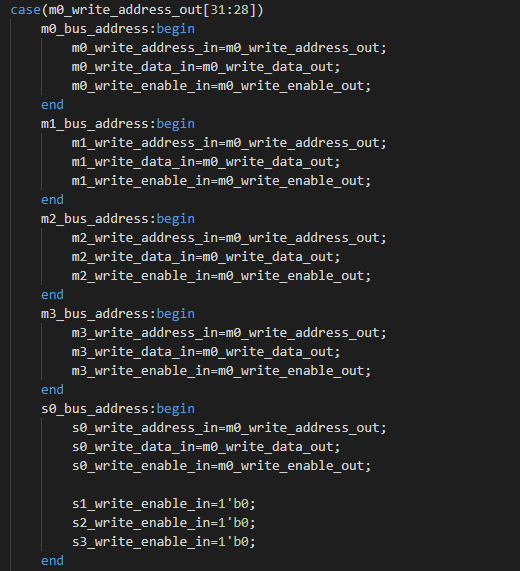
### 2.2总线路由地址

主机读：



注意优先级裁决是时序逻辑，而路由部分是组合逻辑（core读一个地址，需要下个时钟周期能得到数据）。

主机写：



注意把其他从机的写使能拉低，读写地址一变，接口切换可能快于信号切换，可能写使能就一直被拉高了。

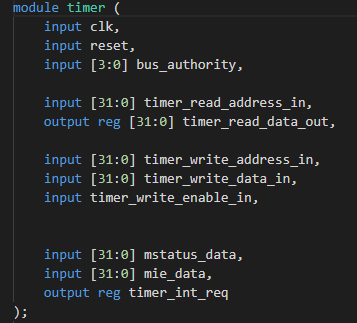
### 2.3实际总线连接



主机0接的core，从机0接的内存，从机1接的定时器timer，从机2接的gpio，从机3接的串口。

## 3定时器部分

定时器端口（兼容总线）：

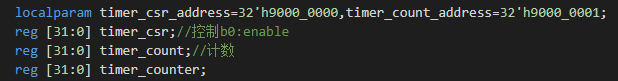


Mstatus data和mie data是从core的csr直接接过来的，因为timer的运行需要看中断是否允许。

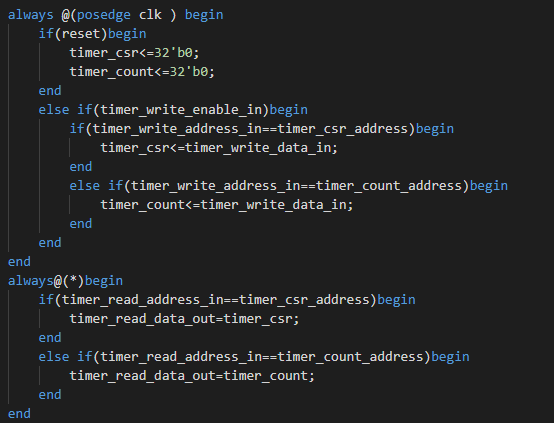


定时器的中断请求直接接在csr的interrupt pend上。

### 3.1定时器寄存器

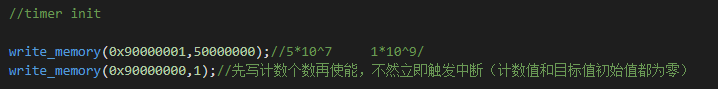


Timer Csr是控制状态寄存器，第0位为使能位，timer count是定时器需要计数的个数，timer counter是实际计数过程中的个数。



Todo：读改成时序逻辑。

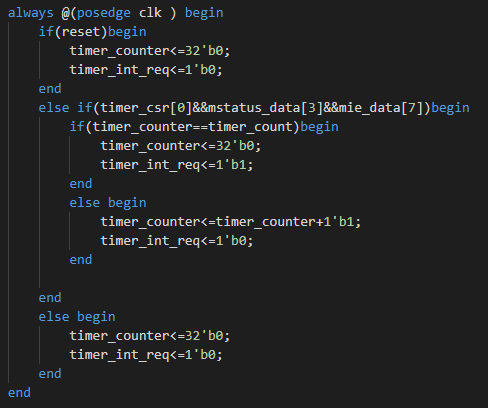
写使能时，根据写的地址将数据写到对应寄存器中。程序里配置如下：计5\*10^7个数，使能定时器。



开启总中断和定时器中断。



### 3.2定时器计数和中断触发部分

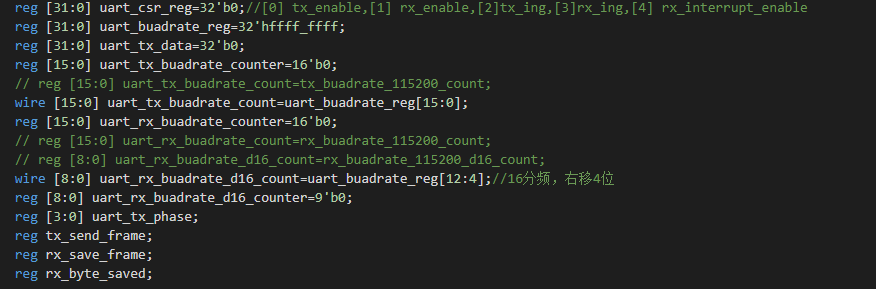


当允许定时器中断和定时器使能时，定时器counter开始计数，一个时钟周期计一个，当达到count时，触发中断请求并清零counter。

## 3.3串口部分



串口有四个寄存器，csr控制状态寄存器（[0] tx\_enable,[1] rx\_enable,[2]tx\_ing,[3]rx\_ing,[4] rx\_interrupt\_enable），波特率设置寄存器，发送数据寄存器，结束数据寄存器。



Tx send frame是当串口正在发送时，该位置1，rx save frame是当串口正在接收时，该位为1，rx byte saved是接收到一个byte置1，保持一个时钟周期。

实现部分有些繁琐，参考小梅哥的教程187页。

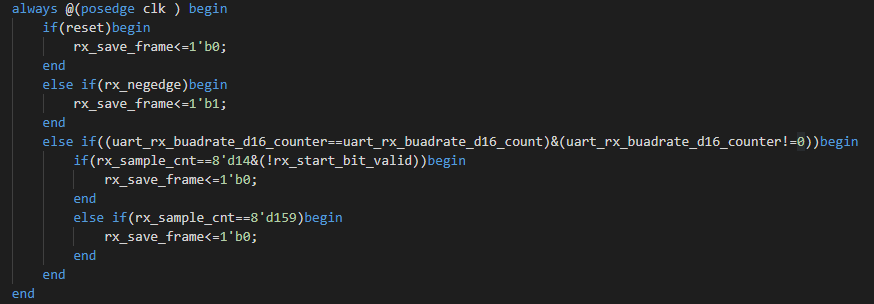
### 3.4串口发送



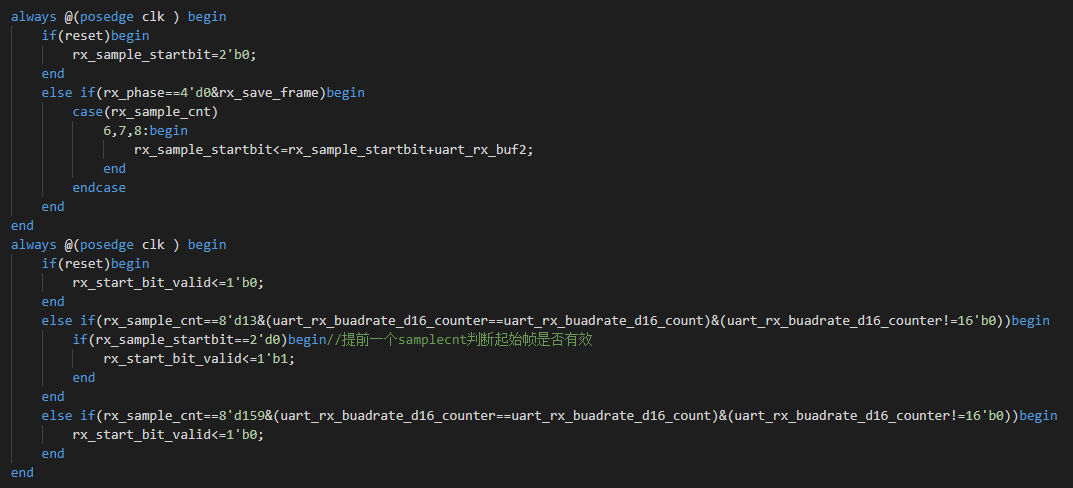


当向串口的uart tx data 寄存器写数据时，自动触发发送一字节。

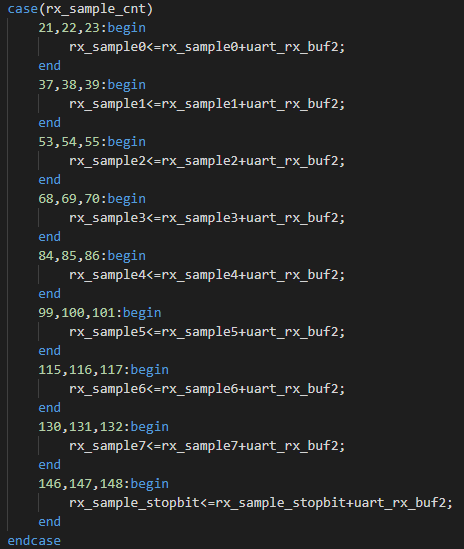
### 3.5串口接收



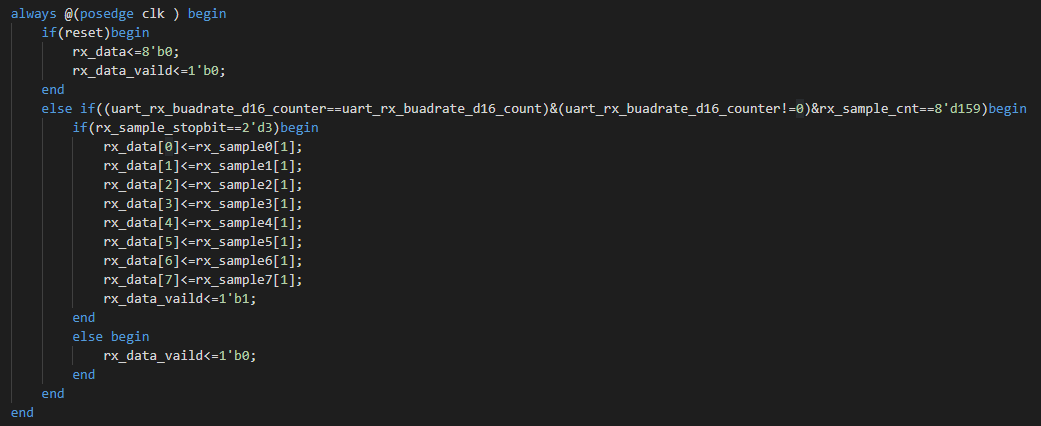
当rx下降沿来的时候触发接收。



先判断起始帧是否有效，如无效则终止接收，这里sample cnt为8时采样起始帧完成，下个时钟周期就可以判断是否有效了，不过这里在sample cnt为13时判断，只要比采样下一bit早就行。

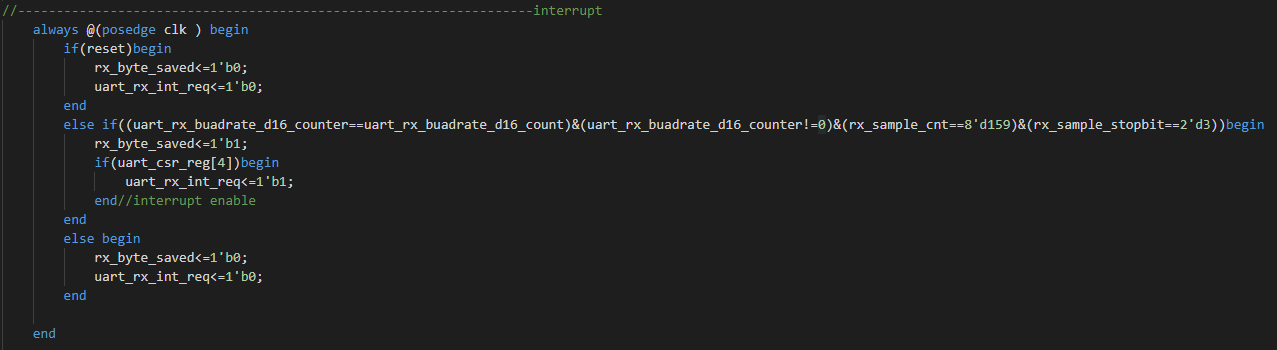


将每bit的采样时间分成16份，在每bit中间采样。包括起始帧和结束帧一共是10位，所以采样计数器rx sample cnt为0-159。



如果结束帧为高电平，则当前byte有效。

### 3.6串口接收中断





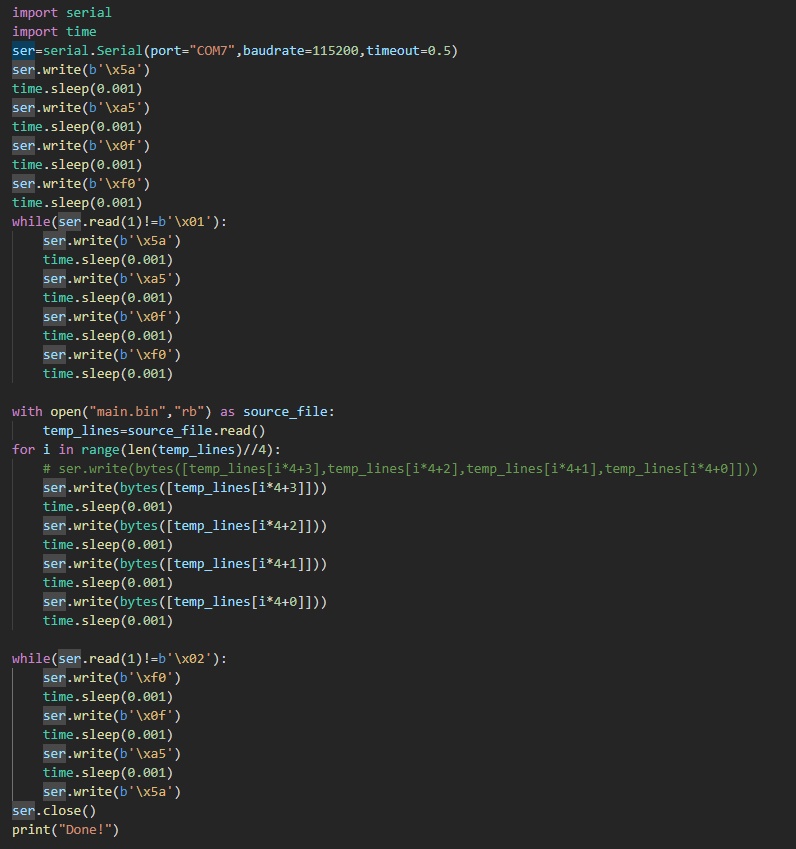
接收到一个有效byte时，判断串口中断是否使能，是就发出中断请求，和定时器中断请求一样，都是接在csr的interrupt pend上。

### 3.7串口下载程序

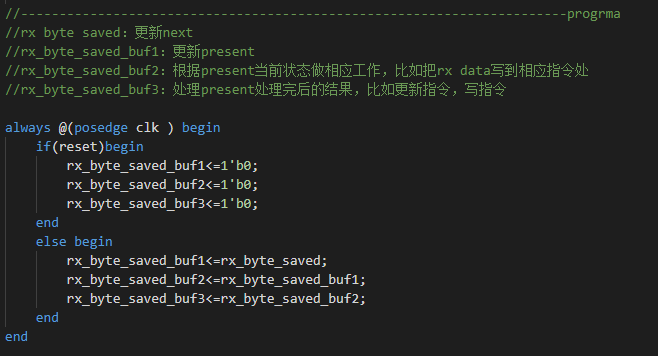
（这里将串口和下载程序混杂在了一起，todo将下载程序部分独立出来，单独给时钟）

协议是串口从数据流查找起始帧（0x5a，0xa5，0x0f，0xf0），找到起始帧后，串口发送一个应答字节0x01，然后将core挂起，开始接收编程流，每4个字节写入flash里，并在编程流里查找终止帧（0xf0，0x0f，0xa5，0x5a），找到后串口发送接收完成反馈0x02。

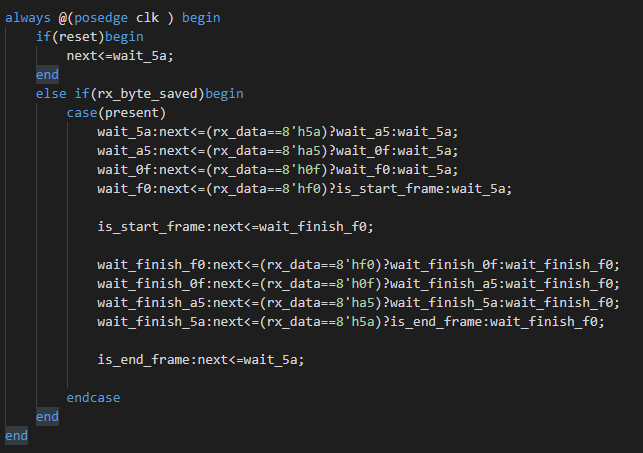
编程的上位机代码：



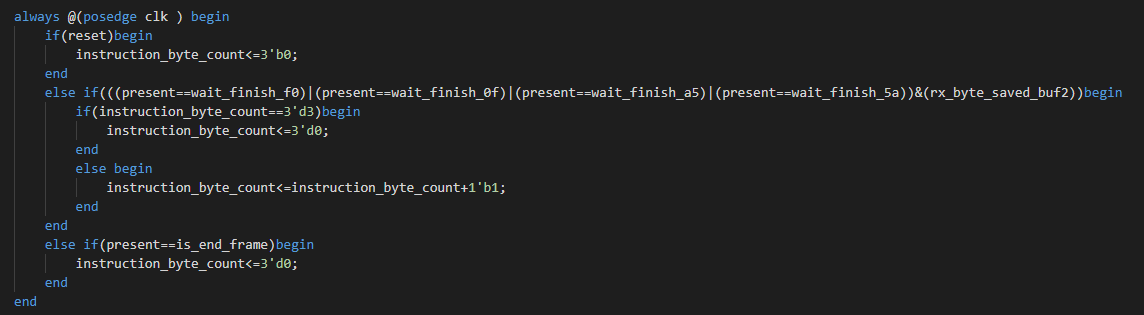
Main.bin就是编译出来的代码。

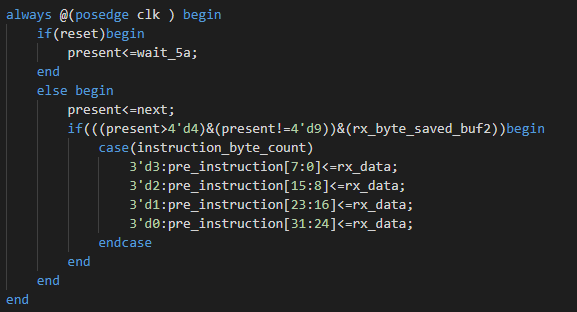


当串口介绍到一字节时，接收信号会寄存三个时钟周期。

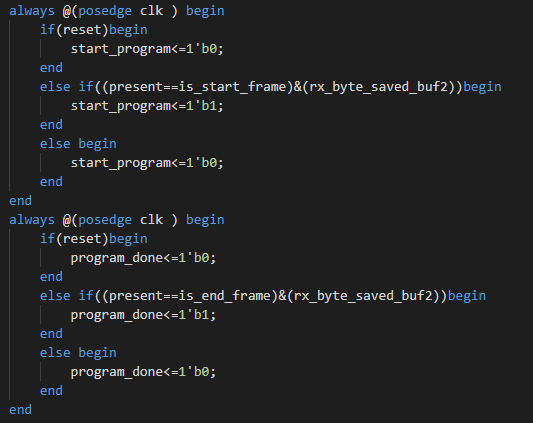


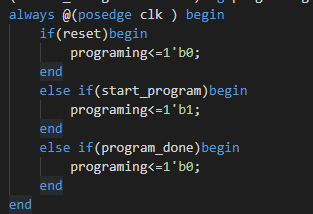
编程逻辑的状态机，在接收到一字节时更新next的状态。

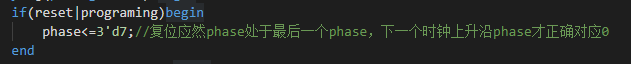




如果当前接收到一个字节的第2个时钟周期，present处于查找结束帧的状态，就将接收到的数据4个一组写入flash。

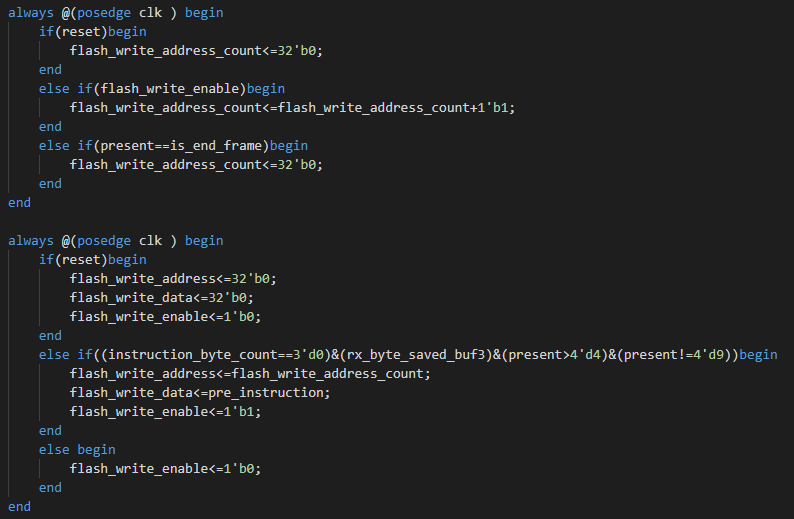








声明开始和结束编程信号。一旦开始编程，core的pc指针归0，编程往后就能执行新的程序。



写core的flash。

# 4移植到FPGA平台

这里以xilinx平台为例，其他平台类似，只要资源和时序满足要求即可（时钟、gpio换成自己fpga上对应的）。

根据以上分析，实际对硬件（fpga）的资源需求是一个时钟模块，一个flash块（装程序），一个memory块（装运行时数据ram）。和一些lut，目前差不多要2k个lut。最好别用lut去生成flash和memory块，资源要合理运用。现在fpga平台都带有片上ram，有更好的时序和功耗。

生成时钟：

（这里最高试过跑125MHz，可以正常运行，再高就没试过了）

以50MHz为例（示例代码是按50MHz配置的串口波特率计数值）