INTRODUCTION TO RISC-V

RISC-V入门教程

ISA | 汇编指令 | 系统编程 | 组成原理 | 嵌入式应用

异常和中断

主讲 邢建国

中国开放指令生态（RISC-V）联盟 | 浙江中心
China RICS-V Alliance | Zhejiang Center

浙江图灵算力研究院
ZHEJIANG TURING INSTITUTE

RISC-V®

异常和中断

- 异常和中断
- 与异常/中断相关的CSR
- 异常、ecall/uret
- 外部中断

异常

异常是CPU在执行指令时响应异常条件而生成的事件。例如，尝试执行非法指令是导致RISC-V CPU生成异常的条件。

异常通常会触发异常处理机制，以便在CPU继续执行程序之前处理异常条件。

这种机制通常会导致CPU将执行流程重定向到异常处理例程，该例程可能：

- 保存当前程序上下文；
- 处理异常条件；
- 恢复保存程序的上下文以继续执行

中断

硬件中断是一种机制，允许硬件通知CPU外设事件发生。

外部中断是由外部硬件（如外设）引起的中断，以通知CPU它们需要注意。当CPU接收到中断后，做如下处理：

- 保存当前程序的上下文
- 调用一个函数来处理硬件中断
- 恢复保存程序的上下文并继续执行

- 使用中断的场合：
  - 慢速设备
  - 输入是随机发生的

异常

异常处理流程与用于处理硬件中断的流程非常相似。

RISC-V CPU使用相同的机制来处理中断和异常，它保存部分当前上下文（例如PC内容），设置CSR，并将执行流程重定向到中断服务例程。

中断服务例程可以通过检查中断原因寄存器mcause和状态寄存器mstatus，来区分中断和异常。

- mcause INTERRUPT（最高位）表示中断还是异常
- mcause.EXCCODE（低31位）表示中断或异常的来源。

异常

RISC-V定义了多种异常和中断源。下表显示了中断和异常的来源及其相应编码。

mcause fields INTERRUPT	mcause fields EXCCODE	Cause
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	Reserved
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	Reserved for future standard use
0	15	Store/AMO page fault
0	16-23	Reserved for future standard use
0	24-31	Reserved for custom use
0	32-47	Reserved for future standard use
0	48-63	Reserved for custom use
0	≥64	Reserved for future standard use

mcause fields INTERRUPT	mcause fields EXCCODE	Cause
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	Reserved for future standard use
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	Reserved for future standard use
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	Reserved for future standard use
1	11	Machine external interrupt
1	12-15	Reserved for future standard use
1	≥ 16	Reserved for platform use

异常

异常处理机制通常用于保护系统免受非法用户代码操作。

在这种情况下，硬件由系统软件配置，以在特权模式设置为用户/应用程序并且CPU尝试执行某些操作时生成异常，例如访问映射到外围设备的地址或访问只能在机器模式下访问的控制和状态寄存器。

在异常时，属于系统软件的中断服务例程可以决定如何处理用户程序。

配置异常和软件中断机制

RISC-V CPU使用类似的机制来处理中断、异常和软件中断，它保存了部分当前上下文（PC内容），设置了例如mcause和其他CSR，并将执行流程重定向到中断服务例程。因此，配置异常和软件中断处理机制与配置外部中断处理机制非常相似。

系统软件通过注册将处理这些事件的例程来配置异常和软件中断机制。这与注册中断服务例程来处理外部中断的方式相同。

在直接模式下，注册一个异常处理例程，负责检查mcause，识别事件源并调用正确的处理函数。在向量模式下，外部中断、异常和软件中断处理例程必须在中断向量表中注册。

在RISC-V上，必须通过设置控制和状态寄存器的mstatus来启用外部中断。另一方面，异常和软件中断总是启用的，不需要额外的配置。

处理非法操作

当指令试图执行非法操作时，RISC-V CPU都会生成异常。例如执行CPU无法识别的指令。

使用机器和用户模式的RISC-V系统通常包括一个内存保护单元，可以配置为在CPU尝试读取或写入数据或从特定地址获取执行指令时生成异常。

操作系统可以通过配置此单元来保护系统，当用户模式执行的代码尝试访问受保护地址时生成异常。例如映射到包含操作系统或其他软件的外围设备和内存地址的地址。

在这些情况下，如果CPU尝试从/向受保护地址读取/写入数据，内存保护单元会生成“加载访问故障”（mstatus.EXECCODE=5）/“存储/AMO访问故障”异常（mstatus.EXECCODE=7）。

如果CPU从一个保护的地址取指令执行时，系统生成“指令访问错误”异常（mstatus.EXECCODE=1）

处理非法操作

每当产生异常时，RISC-V CPU执行以下动作：

- 将当前程序计数器保存到mepc中
- 使用标识异常源的代码设置mcause
- 将当前模式保存到mstatus MPP
- 将特权级别更改为机器模式
- 设置程序计数器以将执行重定向到异常处理例程

处理非法操作

一些异常还可能设置mtval，其中包含有关异常的外部信息。例如，当加载或存储访问异常发生时，mtval设置为引起异常的虚拟地址。

缺页异常是系统可以处理的异常，因此导致异常的软件可能会继续执行。

在这些情况下，处理异常通常类似于处理外部中断，即异常处理例程必须保存上下文，处理异常，并最终恢复上下文，以便在CPU上运行的软件可以继续执行。

在非法操作产生异常并且没有已知方法从问题中恢复的情况下，操作系统可能会杀死试图执行非法操作的进程。

处理系统调用

在RISC-V中，用户代码可以通过执行ecall来执行系统调用。

该指令生成一个软件中断，该中断调用中断和异常处理机制类似。

如果一条指令在用户模式下执行，硬件将mcause INTERRUPT设置为0，mcause EXCCODE设置为8。

如果中断/异常原因处理机制配置为直接模式，则主中断服务例程可以通过将mcause寄存器上的值与8进行比较来识别对操作系统的调用。

处理系统调用

每当发生异常或软件中断时，系统在mepc中记录当前程序计数器PC值。

在处理异常后，例如缺页异常，系统可能会将执行返回到导致异常的同一指令。

但软件中断处理例程必须在执行mret指令应返回epc之后的那条指令，因此在mret返回之前要之前调整值mepcin，使其指向下一条指令。

```
1   # Adjusting MEPC so mret returns to the instruction
2   # placed after the ecall instruction that
3   # generated the software interrupt
4   csrr a1, mepc  # load mepc into a1
5   addi a1, a1, 4 # adds 4 to a1
6   csrw mepc, a1  # writes the new address to mepc
```

■ 异常处理示例

异常是因为代码无法在当前路径上继续执行而不采取额外的行动。这可能是因为代码不正确（例如，加载访问未对齐）或者因为操作系统需要采取行动（ecall、ebreak、页面错误等）。异常处理程序允许程序处理这两种情况。

每个异常必须立即由程序。在RARS中，系统调用和断点通常由RARS处理，但其他故障通常通过在控制台打印错误消息来处理。程序的异常处理程序不一定要打印到控制台。下面是一个简单的处理程序，它只是跳过产生异常的指令。

（RARS只支持用户模式）

■ 异常处理示例

```
.text
main:
    la t0,handler
    csrrw zero, utvec, t0 # 将utvec（5）设置为处理程序地址（最低两位是特殊的）
    csrrsi zero, 0, 1 # 在ustatus（0）中设置中断启用位
here:
    lw zero, 0          # 触发加载访问异常
    j here

handler: # 通过移动epc（65）到下一条指令来忽略它
    csrrw t0, uepc, zero
    addi t0, t0, 4
    csrrw zero, uepc, t0
    uret
```

■ 与异常中断相关的CSR

为了简化，只支持用户模式（与RARS相同）

参见N扩展（软硬协同的用户态中断）
https://gallium70.github.io/rv-n-ext-impl/

用户状态寄存器 (ustatus，0)

用户陷入向量基址寄存器 (utvec，5)

用户中断寄存器 (uip，68 与 uie，4)

内核态陷入委托寄存器 (sedeleg 与 sideleg)

uscratch

用户异常程序计数器 (uepc，65)

用户陷入原因寄存器 (ucause，66)

用户陷入值寄存器 (utval，67)

Registers | Floating Point | Control and Status

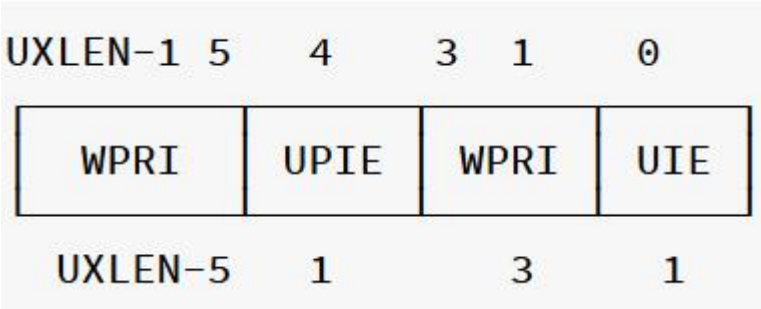
Name	Number	Value
ustatus	0	0x00000000
fflags	1	0x00000000
frm	2	0x00000000
fcsr	3	0x00000000
uie	4	0x00000000
utvec	5	0x00000000
uscratch	64	0x00000000
uepc	65	0x00000000
ucause	66	0x00000000
utval	67	0x00000000
uip	68	0x00000000
cycle	3072	0x00000000
time	3073	0x00000000
instret	3074	0x00000000
cycleh	3200	0x00000000
timeh	3201	0x00000000
instreth	3202	0x00000000

■ 与异常中断相关的CSR

用户状态寄存器 (ustatus)

ustatus 是一个 32 位长的可读写寄存器，记录和控制硬件线程当前的工作状态。

用户态中断使能位 UIE 为零时，用户态中断被禁用。为了向用户态陷入处理程序提供原子性，UIE 中的值在用户态中断发生时被复制到 UPIE ，且 UIE 被置为零。



指令 URET 用于从用户态陷入状态中返回。URET 将 UPIE 复制回 UIE，然后将 UPIE 置位，最后将 uepc 拷贝至 pc。

■ 与异常中断相关的CSR

用户陷入向量基址寄存器 (utvec)

utvec 是 32 位长的可读写寄存器，存储陷入向量的设置，包括向量基址 (BASE) 和向量模式。

```
| BASE[UXLEN-1 : 2] | MODE |
```

utvec 中的 BASE 为 WARL 字段，可以存储任何有效的虚拟地址或物理地址，地址需要对齐到 4 字节。其他的向量模式可能有额外的对齐约束条件。

我们使用基址模式，即utvec存储的是中断处理程序的地址（4字节对齐）

■ 与异常中断相关的CSR

用户中断寄存器 (uip 与 uie)

uip 和 uie 均为32位的可读写寄存器，其中 uip 存储等待处理的中断信息， uie 存储相应的中断使能位。

UXLEN-1	9	8	7	5	4	3	1	0
WPRI		UEIP	WPRI		UTIP	WPRI		USIP
WPRI		UEIE	WPRI		UTIE	WPRI		USIE
UXLEN-9		1	3		1	3		1

定义三种中断：软件中断、时钟中断和外部中断。用户态软件中断通过置位当前硬件线程的 uip 的软件中断等待位 (USIP) 来触发。清零该位可以清除待处理的软件中断。当 uie 中的 USIE 为零时，用户态软件中断被禁用。

■ 与异常中断相关的CSR

用户异常程序计数器 (uepc)

uepc 是 32 位的可读写寄存器。最低位（uepc[0]）恒为零。次低位 uepc[1] 视实现的对齐需求而定。

uepc 是 WARL 寄存器，必须能存储所有有效的虚拟地址，但不需要能够存储所有可能的无效地址。实现可以先将一些非法地址转为其他非法地址再写入 uepc。

当陷入在用户态处理时，被中断或触发异常的指令的虚拟地址被写入 uepc，除此之外 uepc 永远不会被硬件实现写入，但可能被软件显式写入。

异常：写入pc
中断：写入pcNext（下一条指令地址)

与异常中断相关的CSR

用户陷入原因寄存器 (ucause)
ucause 32位长读写寄存器。当陷入在用户态处理时，触发陷入的事件编号被写入 ucause，除此之外 ucause 永远不会被硬件实现写入，但可能被软件显式写入。

Interrupt	Exception Code	Description
1	0	用户态软件中断
1	1-3	*预留*
1	4	用户态时钟中断
1	5-7	*预留*
1	8	用户态外部中断
1	9-15	*预留*
1	≥16	*由平台使用*
0	0	指令地址未对齐
0	1	指令访问错误
0	2	非法指令
0	3	断点
0	4	加载地址未对齐
0	5	加载访问错误
0	6	存储/原子内存操作地址未对齐
0	7	存储/原子内存操作访问错误
0	8	用户态环境调用
0	9-11	*预留*
0	12	指令页错误
0	13	加载页错误
0	14	*预留*
0	15	存储/原子内存操作页错误
0	16-23	*预留*
0	24-31	*自定义用途*
0	32-47	*预留*
0	48-63	*自定义用途*
0	≥64	*预留*

■ 与异常中断相关的CSR

用户陷入值寄存器 (utval)

utval 是 UXLEN 位的可读写寄存器。当陷入在用户态处理时，和特定异常相关的信息将被写入 utval 以帮助软件处理陷入，除此之外 utval 永远不会被硬件实现写入，但可能被软件显式写入。硬件平台指定哪些异常必须将信息写入 utval ，以及哪些异常会无条件写入 0 。

当硬件断点被触发，或是一个指令/加载/存储地址未对齐/访问错误/页错误异常产生时，导致错误的虚拟地址被写入 utval 。当非法指令异常产生时，相应指令的前 XLEN 或 ILEN 位可能被写入 utval 。对于其他异常，utval 被置为 0 ，但未来的标准可能重新定义 utval 的设置。

与异常中断相关的CSR

从异常/中断返回指令uret（mret、sret）：

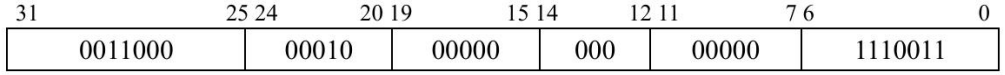
uret 将 pc 设置为 uepc， 将 ustatus.UIE 设置为 ustatus.UPIE， 从而恢复中断前的状态。

Uret的Opcode和csr、ecall、ebreak指令相同

mret ExceptionReturn(Machine)

机器模式异常返回*(Machine-mode Exception Return)*. R-type, RV32I and RV64I 特权架构

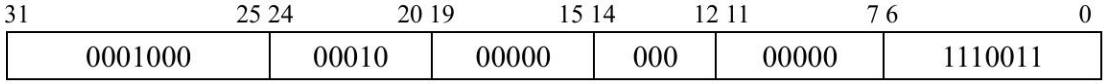
从机器模式异常处理程序返回。将 *pc* 设置为 CSRs[mepc]， 将特权级设置成 CSRs[mstatus].MPP, CSRs[mstatus].MIE 置成 CSRs[mstatus].MPIE，并且将 CSRs[mstatus].MPIE 为 1;并且，如果支持用户模式，则将 CSR [mstatus].MPP 设置为 0。



sret ExceptionReturn(Supervisor)

管理员模式例外返回*(Supervisor-mode Exception Return)*. R-type, RV32I and RV64I 特权指令。

从管理员模式的例外处理程序中返回，设置 *pc* 为 CSRs[spec]，权限模式为 CSRs[sstatus].SPP，CSRs[sstatus].SIE 为 CSRs[sstatus].SPIE，CSRs[sstatus].SPIE 为 1，CSRs[sstatus].spp 为 0。



- 相关指令：uret、ecall、ebreak、csrrw(i)、csrrs(i)、csrrc(i)

000000000000	00000	000	00000	1110011	I ecall
000000000001	00000	000	00000	1110011	I ebreak
csr	rs1	001	rd	1110011	I csrrw
csr	rs1	010	rd	1110011	I csrrs
csr	rs1	011	rd	1110011	I csrrc
csr	zimm	101	rd	1110011	I csrrwi
csr	zimm	110	rd	1110011	I cssrrsi
csr	zimm	111	rd	1110011	I csrrci

mret ExceptionReturn(Machine)

机器模式异常返回*(Machine-mode Exception Return)*. R-type, RV32I and RV64I 特权架构

从机器模式异常处理程序返回。将 *pc* 设置为 CSRs[mepc]， 将特权级设置成 CSRs[mstatus].MPP, CSRs[mstatus].MIE 置成 CSRs[mstatus].MPIE，并且将 CSRs[mstatus].MPIE 为 1；并且， 如果支持用户模式， 则将 CSR [mstatus].MPP 设置为 0。

31 25	24 20	19 15	14 12	11 7	6 0
0011000	00010	00000	000	00000	1110011

相关指令：uret、ecall、ebreak、csrrw(i)、csrrs(i)、csrrc(i)

```
wire system, csr, ecall, ebreak, uret;

assign system = opcode == 7'b11100_11 ? 1 : 0;

assign csr = system && (funct3 != 0);
assign ecall = system && (funct3 == 0) && (rs2 == 0);
assign ebreak = system && (funct3 == 0) && (rs2 == 1);
assign uret = system && (funct3 == 0) && (rs2 == 2);
```

000000000000	00000	000	00000	1110011	I ecall
000000000001	00000	000	00000	1110011	I ebreak
csr	rs1	001	rd	1110011	I csrrw
csr	rs1	010	rd	1110011	I csrrs
csr	rs1	011	rd	1110011	I csrrc
csr	zimm	101	rd	1110011	I csrrwi
csr	zimm	110	rd	1110011	I cssrrsi
csr	zimm	111	rd	1110011	I csrrci

mret ExceptionReturn(Machine)

机器模式异常返回*(Machine-mode Exception Return)*. R-type, RV32I and RV64I 特权架构

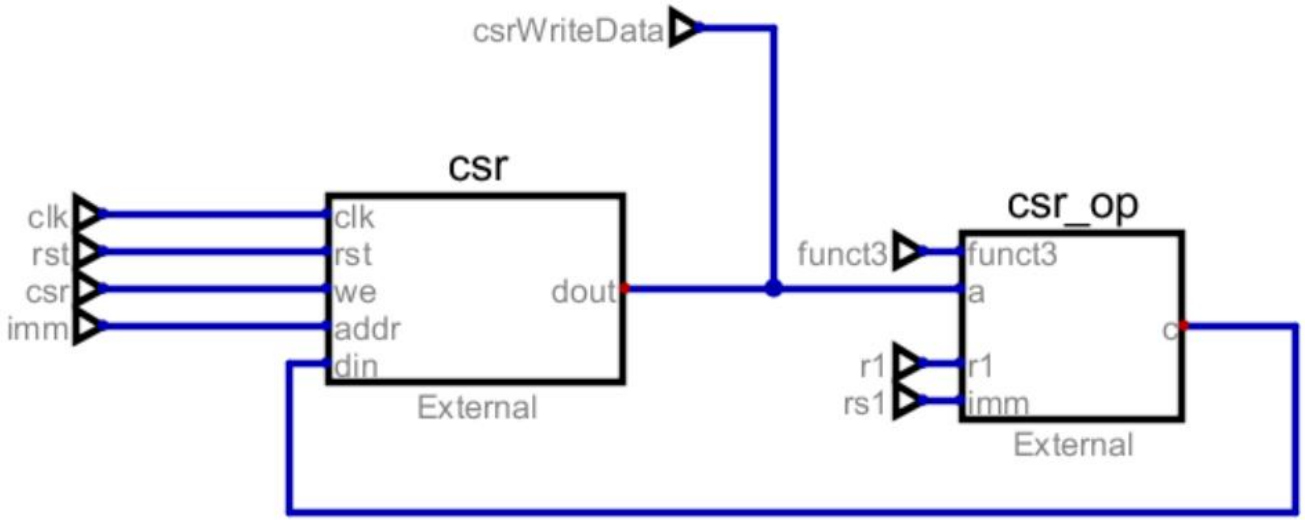
从机器模式异常处理程序返回。将 *pc* 设置为 CSRs[mepc]，将特权级设置成 CSRs[mstatus].MPP, CSRs[mstatus].MIE 置成 CSRs[mstatus].MPIE，并且将 CSRs[mstatus].MPIE 为 1；并且，如果支持用户模式，则将 CSR [mstatus].MPP 设置为 0。

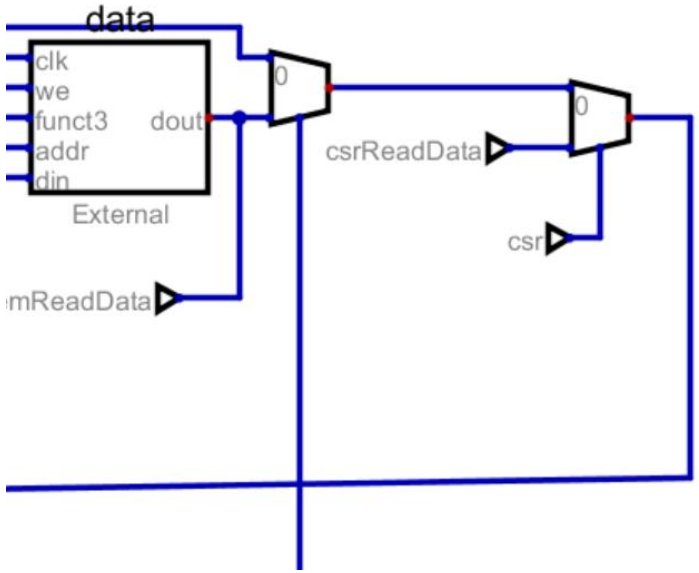
31 25	24 20	19 15	14 12	11 7	6 0
0011000	00010	00000	000	00000	1110011

wreg和wregSrc中增加csr判断（所有的csr指令都要写寄存器）

```
assign wreg = (lui || auipc || jal || jalr || itype || rtype || load || csr) ? 1 : 0;
assign wregSrc = lui ? 0:
         auipc ? 1 :
         (jalr || jal) ? 2 :
         (itype || rtype || csr) ? 3: 3;
```

csrrw(i)、csrrs(i)、csrrc(i)实现



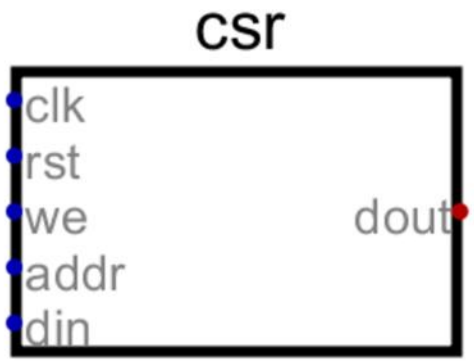


csrrw(i)、csrrs(i)、csrrc(i)实现

```
reg [31:0] ustatus, uie, utvec, uepc, ucause, utval, uip;
```

```
always @(posedge clk)
begin
    if(rst)
    begin
        ustatus <= 0;
        uie <= 0;
        utvec <= 0;
        uepc <= 0;
        ucause <= 0;
        utval <= 0;
        uip <= 0;
    end
    else if(we)
    begin
        case(addr[11:0])
            0: ustatus <= din;
            4: uie <= din;
            5: utvec <= din;
            65: uepc <= din;
            66: ucause <= din;
            67: utval <= din;
            68: uip <= din;
        endcase
    end
end
```

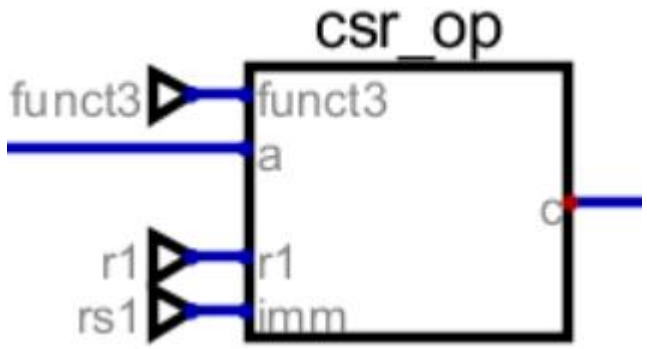
```
always @(*)
begin
    if(~we)
        dout <= 32'bz;
    else
    begin
        case(addr[11:0])
            0: dout <= ustatus;
            4: dout <= uie;
            5: dout <= utvec ;
            65: dout <= uepc ;
            66: dout <= ucause;
            67: dout <= utval ;
            68: dout <= uip;
            default: dout <= 0;
        endcase
    end
end
```



csrrw(i)、csrrs(i)、csrrc(i)实现

```
module csr_op(
    input [2:0] funct3,
    input [31:0] a,
    input [31:0] r1,
    input [4:0] imm,
    output reg [31:0] c
);
    wire [31:0] b;
    assign b = funct3[2] ? {27'd0,imm} : r1;

    always @(*)
    begin
        case(funct3[1:0])
            1:  c = b; //csrrw
            2:  c = a | b; //csrrs
            3: c = a & ~b; //csrrc
            default : c = a;
        endcase
    end

endmodule
```

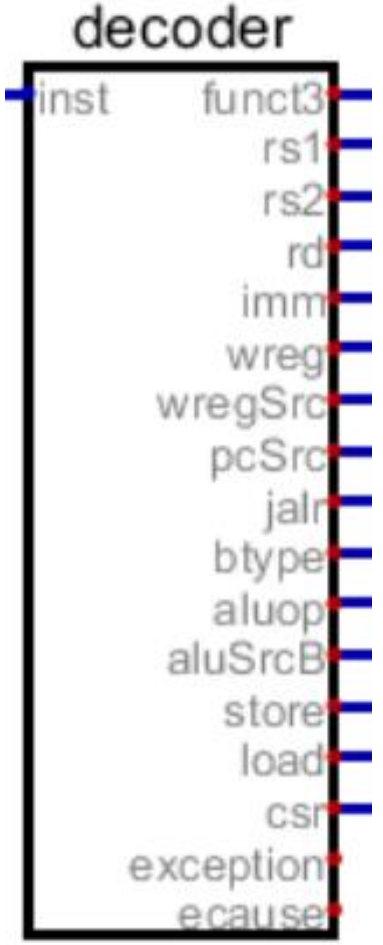
000000000000	00000	000	00000	1110011	I ecall
000000000001	00000	000	00000	1110011	I ebreak
csr	rs1	001	rd	1110011	I csrrw
csr	rs1	010	rd	1110011	I csrrs
csr	rs1	011	rd	1110011	I csrrc
csr	zimm	101	rd	1110011	I csrrwi
csr	zimm	110	rd	1110011	I cssrrsi
csr	zimm	111	rd	1110011	I csrrci



ecall、ebreak、illegal code异常的实现

忽略utval值

```
wire illegal_inst, exception;
assign illegal_inst = ~(lui || auipc || jal || jalr || btype
                  || load || store || rtype || itype || system);
assign exception = illegal_inst || ecall || ebreak;

wire [31:0] ecause;
assign ecause = illegal_inst ? 32'd2 : //非法指令
         ebreak ? 32'd3 :   //break
         ecall ? 32'd8 : 0 ; // ecall
```



■ 异常发生后硬件处理

硬件动作

- ucause <= ecause
- utval <= eval（忽略）
- uepc <= pc
- pc <= utvec
- ustatus：保存原有中断位，禁止中断

修改pc

- pc <= utvec

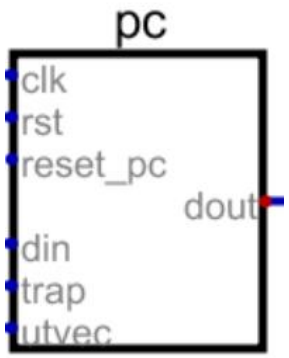
修改csr模块

- ustatus
- ucause <= ecause
- utval <= eval（忽略）
- uepc <= pc

■ 异常发生后硬件处理

修改pc

- pc <= utvec

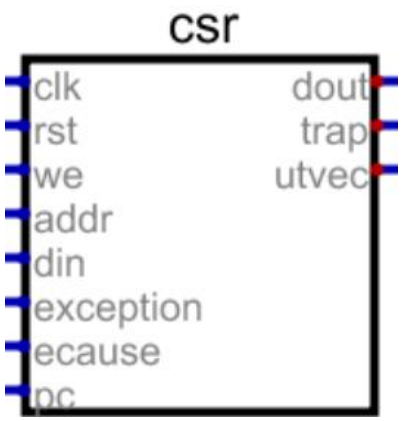



```
module pc(
    input clk,
    input rst,
    input [31:0] reset_pc,
    input [31:0] din,
    output reg [31:0] dout,

    input trap,
    input [31:0] utvec
);
    always @(posedge clk)
    begin
        if(rst)
            dout = reset_pc;
        else if(trap)
            dout = utvec;
        else
            dout = din;
    end
endmodule
```

■ 异常发生后硬件处理

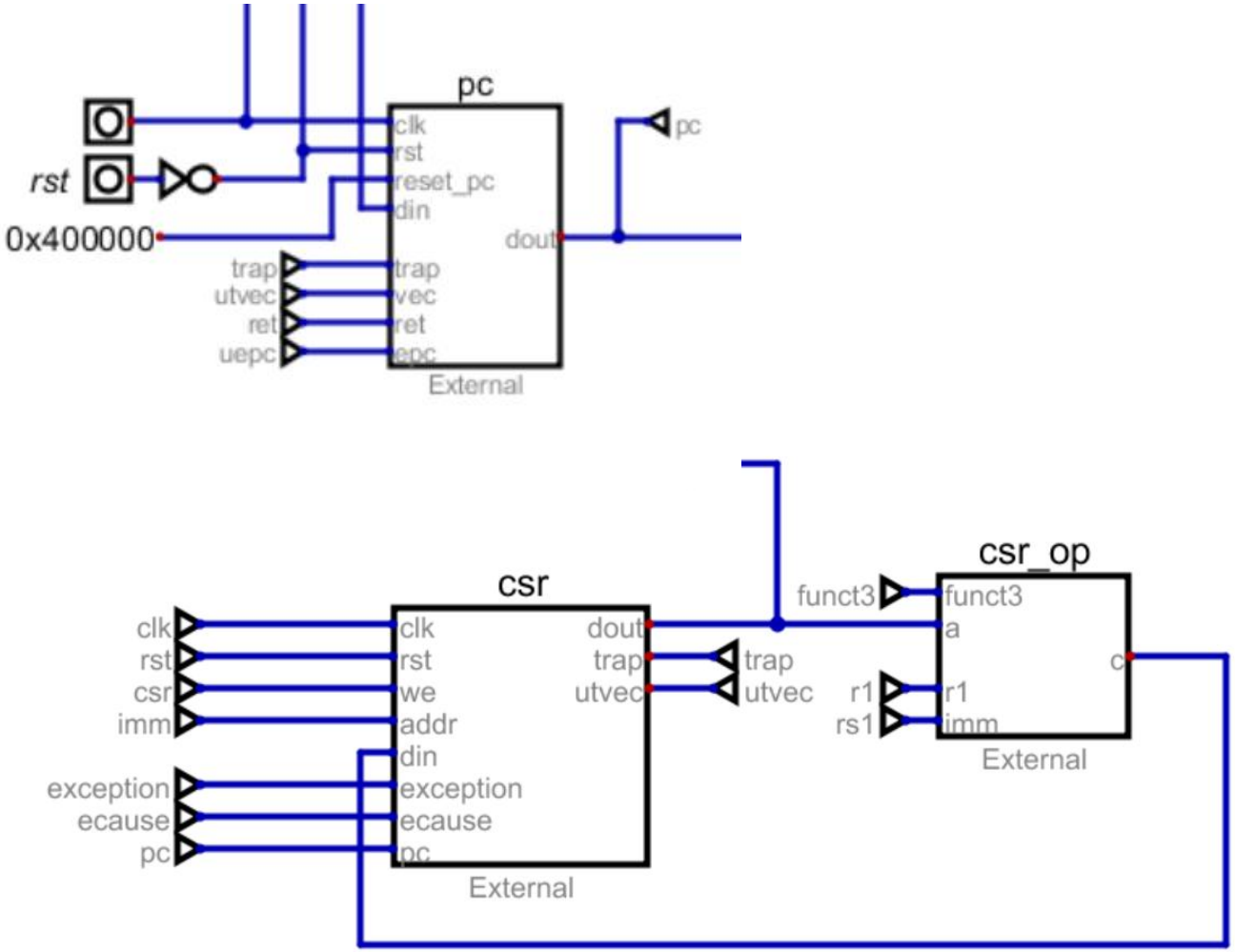
修改csr模块

- ustatus
- ucause <= ecause
- utval <= eval（忽略）
- uepc <= pc




```
always @(posedge clk)
begin
    if(rst)
    begin
        ustatus <= 0;
        uie <= 0;
        utvec <= 0;
        uepc <= 0;
        ucause <= 0;
        utval <= 0;
        uip <= 0;
    end
    else if(exception)
    begin
        ustatus <= {ustatus[31:4], ustatus[1],ustatus[2:1],1'b0};
        uepc <= pc;
        ucause <= ecause;
    end
    else if(we)
    begin
        case(addr[11:0])
            0: ustatus <= din;
            4: uie <= din;
            5: utvec <= din;
```

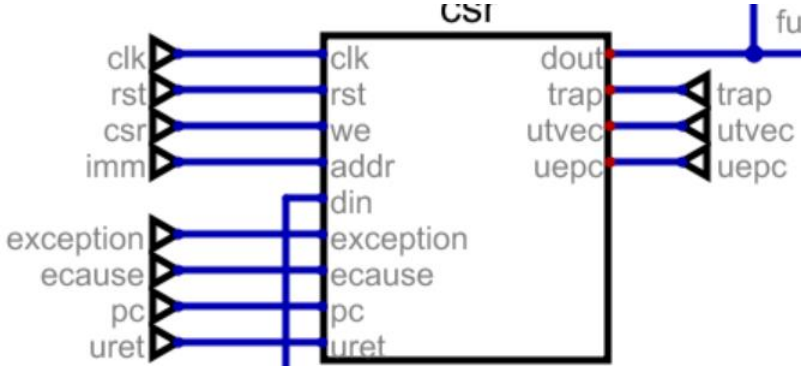
异常发生后硬件处理


pc
clk
rst
reset_pc
din
dout
pc
rst
0x400000
trap
utvec
ret
uepc
trap
vec
ret
epc
External
csr
clk
rst
we
addr
din
exception
ecause
pc
dout
trap
utvec
External
csr_op
funct3
a
c
r1
rs1
imm
External
code
addr
data
External
inst
decoder
inst
funct3
rs1
rs2
rd
imm
wreg
wregSrc
pcSrc
jalr
btype
aluop
aluSrcB
store
load
csr
exception
ecause
External
rs1
csr
exception
ecause

■ 从中断和异常返回：uret指令

uret硬件处理

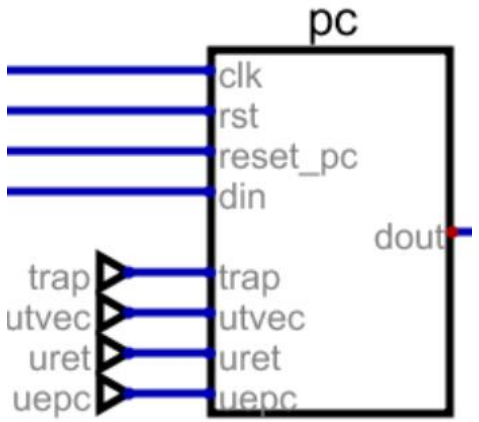
- pc <= uepc
- 恢复status中断位



- 在csr中增加修改ustatus寄存器

```
else if(uret)
    begin
        ustatus <= {ustatus[31:1], ustatus[3]};
    end
```

- 在pc中增加uret处理



异常软件处理

```
.text
        la t1, exception_handler #设置异常中断处理入口地址
        csrw t1, utvec
loop:
        fadd.d f1,f2,f3 #未实现指令，会触发异常
        j loop

.align 2
exception_handler: #异常处理入口地址
        csrr t1, uepc
        addi t1, t1, 4
        csrw t1, uepc  #返回到异常指令下一条指令
        uret
```

系统调用ecall

```
putc:
        li a7, 1
        ecall
        ret
```

```
.align 2
exception_handler: #异常处理入口地址
        csrr t1, ucause
        li t0, 8  #ecall
        beq t1, t0, do_ecall
        j exception_done

do_ecall:
        li t0, 1 #putc
        beq a7,t0, sys_putc
        j exception_done
```

```
sys_putc:
        li t0, term_base
        sw a0, 0(t0)
        j exception_done

exception_done:

        csrr t1, uepc
        addi t1, t1, 4
        csrw t1, uepc  #返回到异常指令下一条指令
        uret
```

系统调用ecall

主程序（在终端上重复输出字符‘x’）

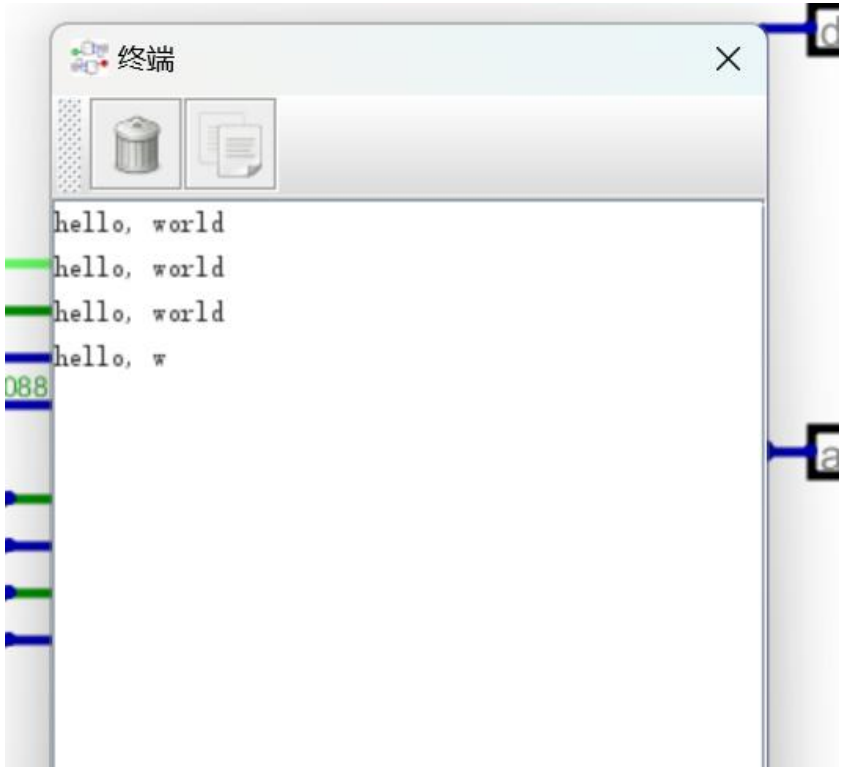
```
.data
hello: .asciz "hello, world\n"
.align 2
stack_end:
        .space 1024
stack_start:

.eqv term_base 0xffff0030
```

```
.text
        la sp, stack_start          #初始化堆栈

        la t1, exception_handler #设置异常中断处理入口地址
        csrw t1, utvec

loop:
        li a0, 'x'
        call putc

        j loop
```

系统调用ecall

主程序（打印字符串hello world：puts）

```
puts:
        addi sp, sp, -8 #保存ra、a1寄存器到堆栈
        sw ra, 0(sp)
        sw a1, 4(sp)

        mv a1, a0
puts_loop:
        lb a0, 0(a1)
        beq a0, zero, puts_done
        call putc
        addi a1,a1,1
        j puts_loop
puts_done:
       lw ra, 0(sp)#从堆栈恢复ra、a1寄存器
       lw a1, 4(sp)
       addi sp, sp, 4
       ret
```



中断

中断和异常是类似的。

中断是外部触发的，如定时器、键盘或其他处理器。

RV32中用户态有三种类型：
- 软件中断 (Software Interrupt)
- 时钟中断 (Timer Interrupt)
- 外部中断

我们以外部中断为例。

中断

外部事件发生时，硬件或软件将 uip.UXIP (X 表示中断种类) 置为 1 。在每条指令执行之前检测发现 uip 非零，进入中断的判断流程。

首先检查该中断是否被委托给用户态处理，即 sideleg 寄存器中对应的位是否为 1；

如果为真，检查用户态全局中断使能是否为真，即 ustatus.UIE 是否为 1 ； 若仍为真，

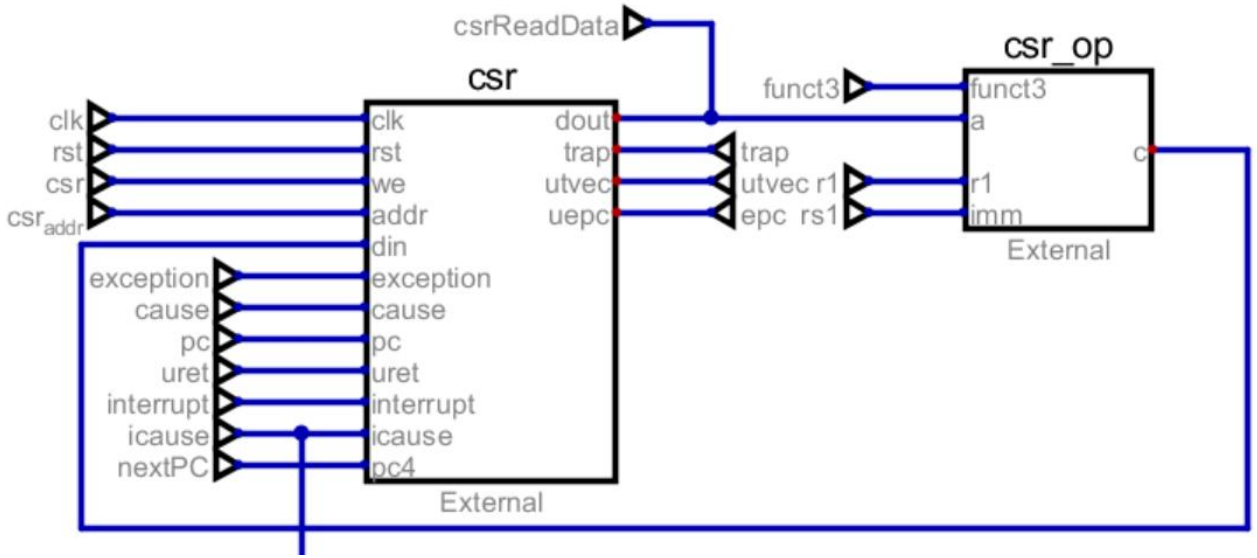
再检查该中断是否被使能，即 uie.UXIE 是否为 1 ； 如果还为真，则触发中断处理的流程。

中断

处理中断:
ucause <= 中断源
uepc <= pcNext
status <= ...(保存中断允许位，禁止中断)

pc <= utvec

修改csr:

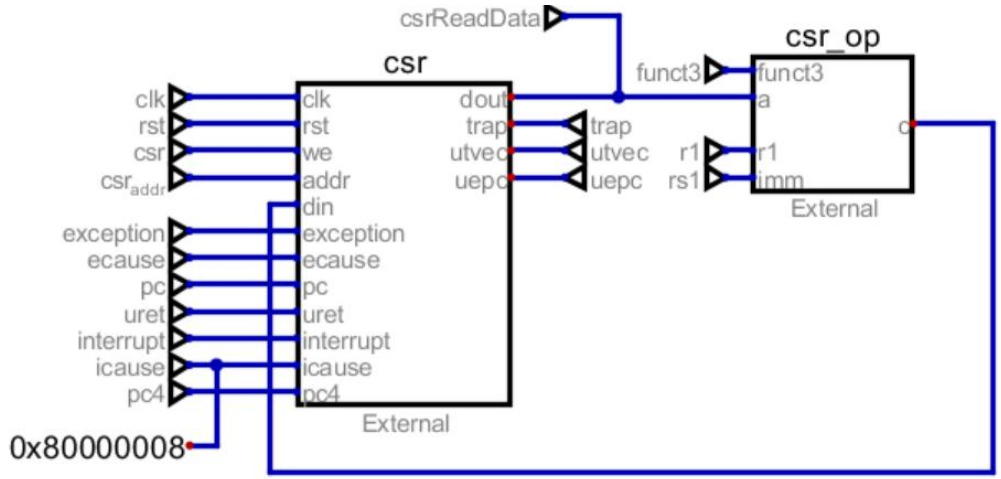


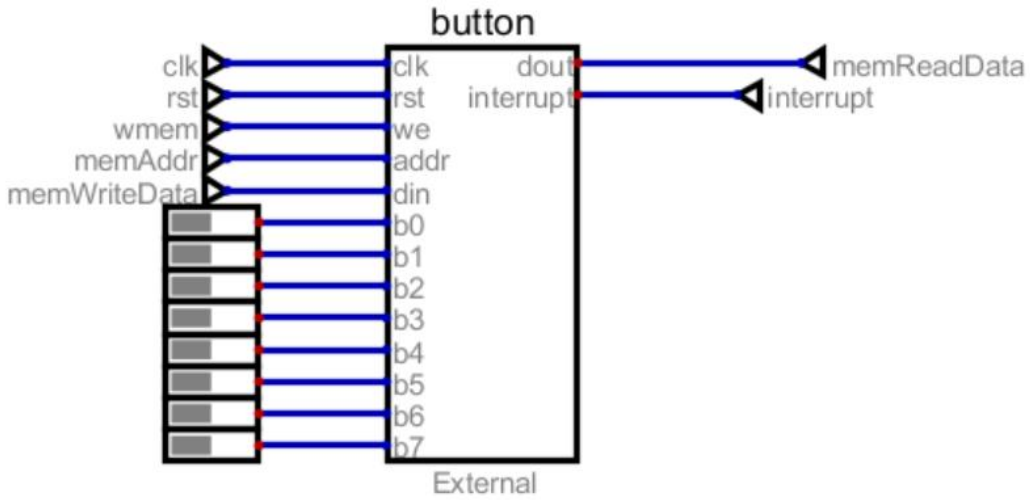

```
assign trap = exception || (interrupt && ustatus[0]);

    else if(interrupt && ustatus[0])
    begin
        ustatus <= {ustatus[31:4], ustatus[1],ustatus[2:1],1'b0};
        uepc <= pc4;
        ucause <= icause;
    end
```

按键中断

外部中断
ucause <= 0x8000_0008

```
reg [7:0] q0, q1;

always @(posedge clk)
begin
    if(rst)
    begin
        q0 <= 0;
        q1 <= 0;
    end
    else
    begin
        q0 <= {b7, b6, b5, b4, b3, b2, b1, b0};
        q1 <= q0;

    end
end

assign interrupt = q0 != q1;
```



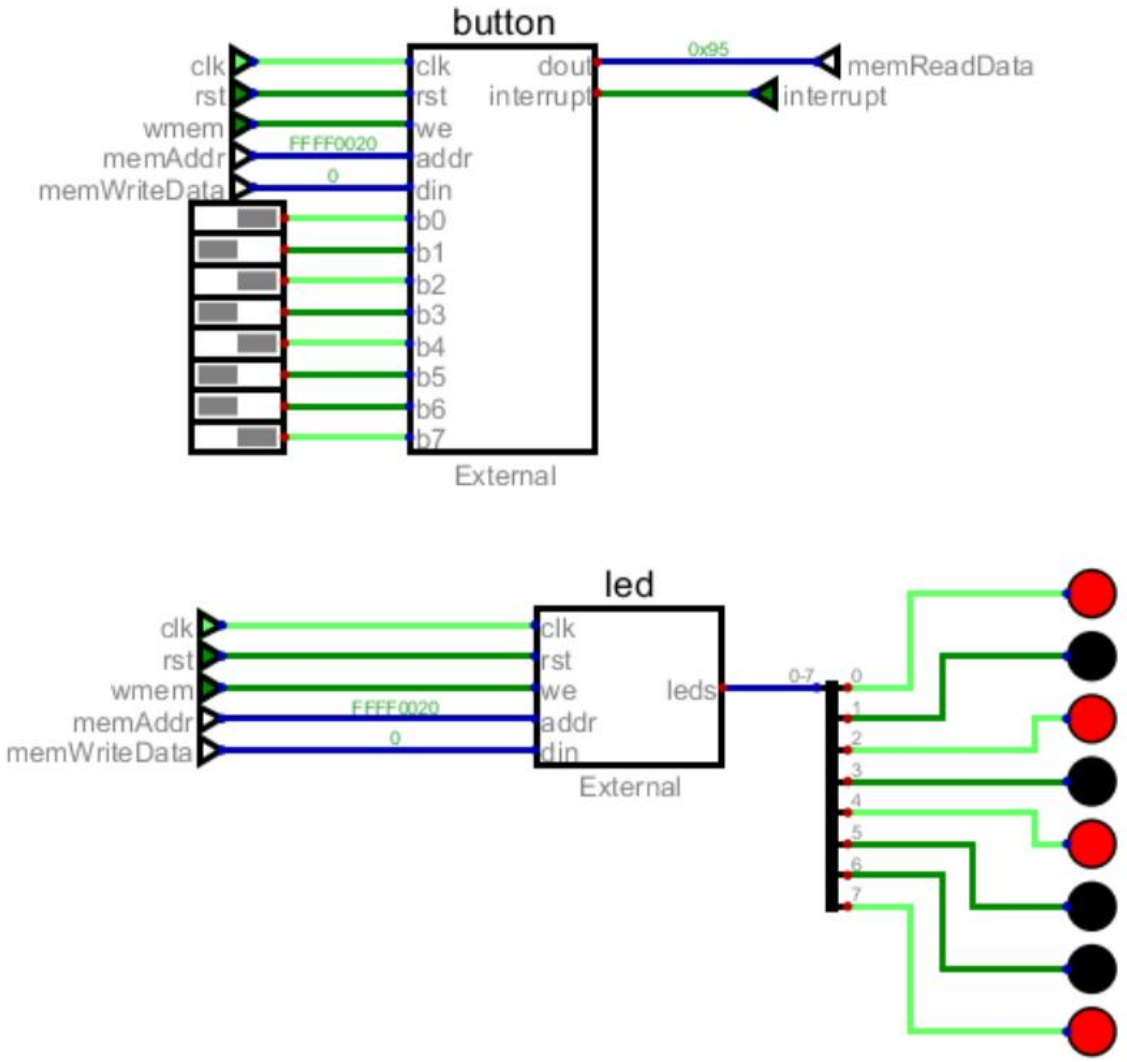


■ 异常/中断软件处理

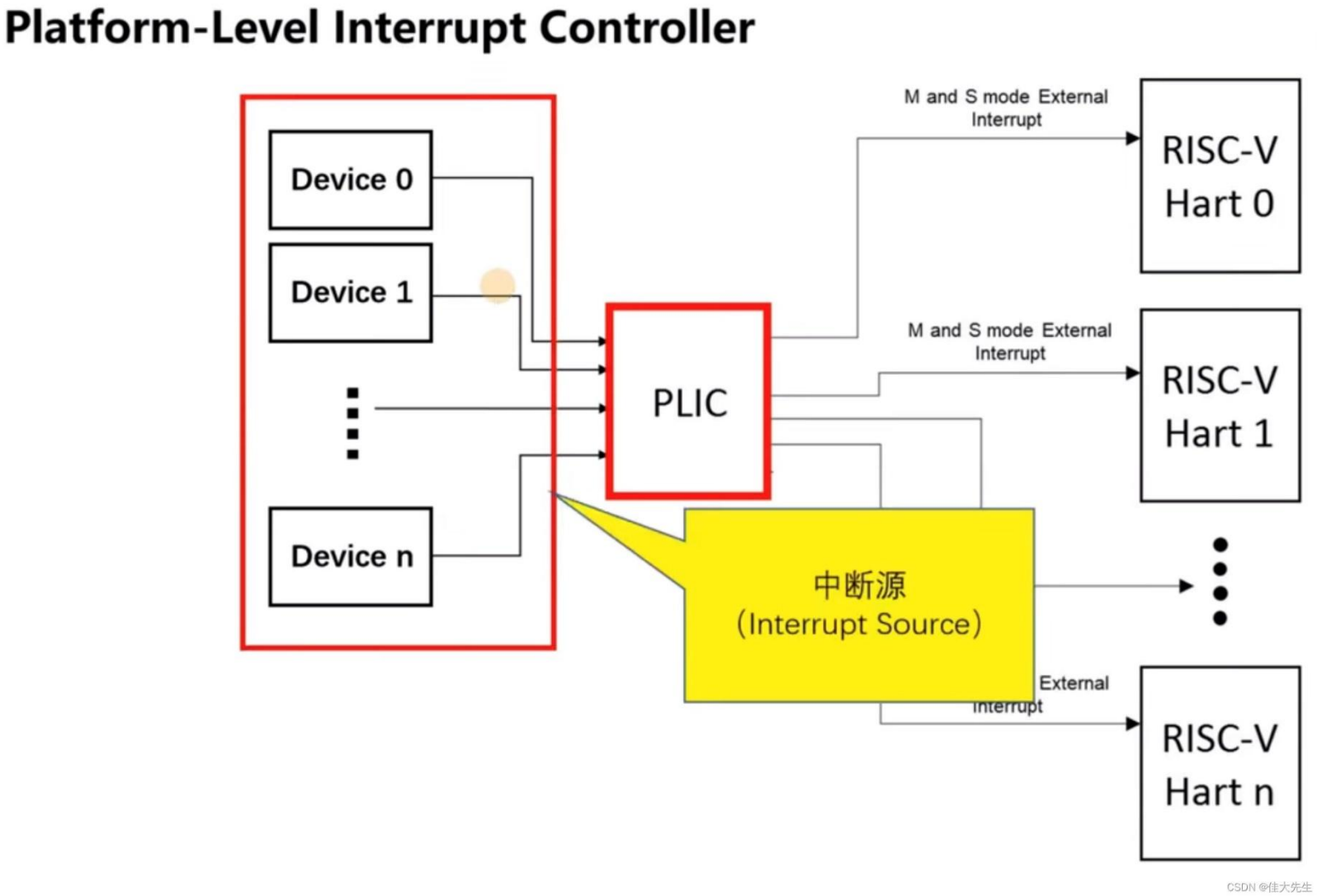
```
.text
        csrci ustatus,1 #关中断

        la t1, exception_handler #设置异常中断处理入口地址
        csrw t1, utvec

        csrsi ustatus, 1 #开中断

loop:   j loop
```

```
.align 2
exception_handler: #异常处理入口地址
        csrr t1, ucause
        li t0, 0x80000008  #外部中断
        beq t1, t0, do_ext_int
        j exception_done
do_ext_int: #根据按键点亮led
        li t0, button_base
        lw a0, 0(t0)
        li t0, led_base
        sw a0, 0(t0)
        uret
```

异常/中断软件处理


button
clk
rst
wmem
memAddr
memWriteData
FFFF0020
0
clk
rst
we
addr
din
b0
b1
b2
b3
b4
b5
b6
b7
dout
interrupt
0x95
memReadData
interrupt
External
led
clk
rst
wmem
memAddr
memWriteData
FFFF0020
0
clk
rst
we
addr
din
leds
0-7
0
1
2
3
4
5
6
7
External

PLIC



CSDN @佳大先生

RISC-V®

异常和中断

- 异常和中断
- 与异常/中断相关的CSR
- 异常、ecall/uret
- 外部中断