



Nuclei™ N 级别处理 器内核指令架构手册

版权声明

版权所有 © 2018–2020 芯来科技（Nuclei System Technology）有限公司。保留所有权利。

Nuclei™是芯来科技公司拥有的商标。本文件使用的所有其他商标为各持有公司所有。

本文件包含芯来科技公司的机密信息。使用此版权声明为预防作用，并不意味着公布或披露。未经芯来科技公司书面许可，不得以任何形式将本文的全部或部分信息进行复制、传播、转录、存储在检索系统中或翻译成任何语言。

本文文件描述的产品将不断发展和完善；此处的信息由芯来科技提供，但不做任何担保。

本文件仅用于帮助读者使用该产品。对于因采用本文件的任何信息或错误使用产品造成的任何损失或损害，芯来科技概不负责。

联系我们

若您有任何疑问，请通过电子邮件 support@nucleisys.com 联系芯来科技，或访问芯来科技“用户中心”网站 <http://user.nucleisys.com> 查看或提交问题。

修订历史

版本号	修订日期	修订的章节	修订的内容
1.0.0	2019/5/12	N/A	1. 初始版本
1.1.0	2019/7/29	5.3.2.3, 6.2.4, 7.2, 7.4.39, 9.2.2, 9.2.3, 9.4	1. 添加了存储器访问错误中断 (5.3.2.3, 6.2.4) 2. 修改了 mmisc_ctl 寄存器 (7.2, 7.5.5) 3. 添加了 DSP 相关的寄存器 (7.2, 7.4.39) 4. 修改了 WFE 的相关描述 (9.2.2, 9.2.3, 9.4)
1.2.0	2019/10/8	1.2	1. 删除了 N 扩展的描述, 增加了 TEE 相关的描述。
1.3.0	2019/12/17	3.4.2, 4.6, 6.1.7, 6.2.4	1. 添加了 mdcause 寄存器 2. 删除了 bus error 转内部中断的描述 3. 添加了 MSFTRST 寄存器的描述
1.4.0	2020/1/9	6.1.2, 6.1.4	1. 将 mstop 改名为 mtimectl, 并添加了自清使能位以及 TIMER 计数时钟模式位
1.5.0	2020/1/20	7.5.5	1. 增加了 Scratchpad 的相关描述
1.5.1	2020/2/17	1.3, 8.1 6.2.3, 6.2.4, 6.2.5	1. 删除了型号 N201 的描述 2. 将 ECLIC 的编程模型中的理论最大中断数目修正为 4096
1.5.2	2020/3/11	6.2.5	1. 修改错误描述, 将 mth 寄存器的地址由 ox8 改成 oxb
2.0.0	2020/4/29	7	1. 添加配置信息寄存器 2. 对应 Nuclei_DSP_QuickStart 2.0.0 版本 3. 对应 Nuclei_CCM_Mechanism 1.0 版本

版本号	修订日期	修订的章节	修订的内容
2.0.1	2020/5/27	7.5.7	1. 修改 mcache_ctl[1]/scratchpad mode 复位后的默认值为 1
2.0.2	2020/6/29	7, 10	1. 增加 ECC 相关 CSR 寄存器描述 2. 增加 ECC 处理机制介绍

目录

版权声明	0
联系我们	0
修订历史	1
表格清单	9
图片清单	11
1. N 级别处理器内核指令集与 CSR 介绍	12
1.1. RISC-V 指令集介绍	12
1.2. N 级别处理器内核支持指令集	12
1.3. CSR 寄存器	13
2. N 级别处理器内核特权架构介绍	14
2.1. 总体介绍	14
2.2. 特权模式 (PRIVILEGE MODES)	14
2.2.1. 机器模式 (Machine Mode)	14
2.2.2. 用户模式 (User Mode)	15
2.2.3. 机器子模式 (Machine Sub-Mode)	15
2.2.4. 模式 (Mode) 的查看	16
2.2.5. Machine Mode 到 User Mode 的切换	16
2.2.6. User Mode 到 Machine Mode 的切换	17
2.2.7. 中断、异常、NMI 的嵌套	18
2.3. 物理存储器保护 (PMP)	18
3. N 级别处理器内核异常机制介绍	19
3.1. 异常概述	19
3.2. 异常屏蔽	19
3.3. 异常的优先级	19
3.4. 进入异常处理模式	19
3.4.1. 从 <i>mtvec</i> 定义的 PC 地址开始执行	20
3.4.2. 更新 CSR 寄存器 <i>mcause</i> 和 <i>mdcause</i>	21
3.4.3. 更新 CSR 寄存器 <i>mepc</i>	22
3.4.4. 更新 CSR 寄存器 <i>mtval</i>	22
3.4.5. 更新 CSR 寄存器 <i>mstatus</i>	23
3.4.6. 更新 <i>Privilege Mode</i>	23
3.4.7. 更新 <i>Machine Sub-Mode</i>	23

3.5.	退出异常处理模式	24
3.5.1.	从 <i>mepc</i> 定义的 PC 地址开始执行.....	25
3.5.2.	更新 CSR 寄存器 <i>mstatus</i>	25
3.5.3.	更新 <i>Privilege Mode</i>	26
3.5.4.	更新 <i>Machine Sub-Mode</i>	26
3.6.	异常服务程序.....	27
3.7.	异常嵌套	27
4.	N 级别处理器内核 NMI 机制介绍.....	28
4.1.	NMI 概述	28
4.2.	NMI 屏蔽	28
4.3.	进入 NMI 处理模式.....	28
4.3.1.	从 <i>mnvec</i> 定义的 PC 地址开始执行.....	29
4.3.2.	更新 CSR 寄存器 <i>mepc</i>	29
4.3.3.	更新 CSR 寄存器 <i>mcause</i>	30
4.3.4.	更新 CSR 寄存器 <i>mstatus</i>	30
4.3.5.	更新 <i>Privilege Mode</i>	30
4.3.6.	更新 <i>Machine Sub-Mode</i>	31
4.4.	退出 NMI 处理模式.....	31
4.4.1.	从 <i>mepc</i> 定义的 PC 地址开始执行.....	32
4.4.2.	更新 CSR 寄存器 <i>mstatus</i>	32
4.4.3.	更新 <i>Privilege Mode</i>	33
4.4.4.	更新 <i>Machine Sub-Mode</i>	33
4.5.	NMI 服务程序	33
4.6.	NMI/异常嵌套.....	34
4.6.1.	进入 NMI/异常嵌套.....	34
4.6.2.	退出 NMI/异常嵌套.....	36
5.	N 级别处理器内核中断机制介绍	39
5.1.	中断概述	39
5.2.	中断控制器 ECLIC	39
5.3.	中断类型	40
5.3.1.	外部中断.....	40
5.3.2.	内部中断.....	40
5.4.	中断屏蔽	41
5.4.1.	中断全局屏蔽.....	41
5.4.2.	中断源单独屏蔽.....	41
5.5.	中断级别、优先级与仲裁.....	42
5.6.	进入中断处理模式	42

5.6.1.	从新的 PC 地址开始执行.....	43
5.6.2.	更新 Privilege Mode.....	44
5.6.3.	更新 Machine Sub-Mode.....	44
5.6.4.	更新 CSR 寄存器 mepc.....	44
5.6.5.	更新 CSR 寄存器 mcause 和 mstatus.....	45
5.7.	退出中断处理模式.....	46
5.7.1.	从 mepc 定义的 PC 地址开始执行.....	47
5.7.2.	更新 CSR 寄存器 mcause 和 mstatus.....	47
5.7.3.	更新 Privilege Mode.....	48
5.7.4.	更新 Machine Sub-Mode.....	48
5.8.	中断向量表.....	49
5.9.	进出中断的上下文保存和恢复.....	50
5.10.	中断响应延迟.....	50
5.11.	中断嵌套.....	50
5.12.	中断咬尾.....	51
5.13.	中断的向量处理模式和非向量处理模式.....	52
5.13.1.	非向量处理模式.....	52
5.13.2.	向量处理模式.....	56
6.	N 级别处理器内核 TIMER 和 ECLIC 介绍.....	61
6.1.	TIMER 介绍.....	61
6.1.1.	TIMER 简介.....	61
6.1.2.	TIMER 寄存器.....	61
6.1.3.	通过 mtime 寄存器进行计时.....	62
6.1.4.	通过 mtimectl 寄存器控制计时器计数.....	62
6.1.5.	通过 mtime 和 mtimecmp 寄存器生成计时器中断.....	63
6.1.6.	通过 msip 寄存器生成软件中断.....	63
6.1.7.	通过 msfrst 寄存器生成软件复位请求.....	63
6.2.	ECLIC 介绍.....	64
6.2.1.	ECLIC 简介.....	65
6.2.2.	ECLIC 中断目标.....	66
6.2.3.	ECLIC 中断源.....	67
6.2.4.	ECLIC 中断源的编号 (ID).....	67
6.2.5.	ECLIC 的寄存器.....	68
6.2.6.	ECLIC 中断源的使能位 (IE).....	71
6.2.7.	ECLIC 中断源的等待标志位 (IP).....	71
6.2.8.	ECLIC 中断源的电平或边沿属性 (Level or Edge-Triggered).....	72
6.2.9.	ECLIC 中断源的级别和优先级 (Level and Priority).....	73

6.2.10. ECLIC 中断源的向量或非向量处理 (Vector or Non-Vector Mode)	75
6.2.11. ECLIC 中断目标的阈值级别	75
6.2.12. ECLIC 中断的仲裁机制	76
6.2.13. ECLIC 中断的响应、嵌套、咬尾机制	76
7. N 级别处理器内核 CSR 寄存器介绍	76
7.1. N 级别处理器内核 CSR 寄存器概述	76
7.2. N 级别处理器内核的 CSR 寄存器列表	76
7.3. N 级别处理器内核的 CSR 寄存器的访问权限	80
7.4. N 级别处理器内核支持的 RISC-V 标准 CSR	81
7.4.1. misa	81
7.4.2. mie	82
7.4.3. mvendorid	82
7.4.4. marchid	83
7.4.5. mimpid	83
7.4.6. mhartid	83
7.4.7. mstatus	83
7.4.8. mstatus 的 SIE 域	84
7.4.9. mstatus 的 MIE 域	84
7.4.10. mstatus 的 SPIE 与 SPP 域	85
7.4.11. mstatus 的 MPIE 和 MPP 域	85
7.4.12. mstatus 的 FS 域	85
7.4.13. mstatus 的 XS 域	86
7.4.14. mstatus 的 MPRV 域	87
7.4.15. mstatus 的 SUM 域	87
7.4.16. mstatus 的 SD 域	87
7.4.17. mtvec	87
7.4.18. mtvt	88
7.4.19. mscratch	89
7.4.20. mepc	89
7.4.21. mcause	90
7.4.22. mtval (mbadaddr)	90
7.4.23. mip	91
7.4.24. mnxti	91
7.4.25. mintstatus	92
7.4.26. mscratchcsw	92
7.4.27. mscratchcswl	93
7.4.28. mcycle 和 mcycleh	94
7.4.29. minstret 和 minstret	94
7.4.30. mtime、mtimecmp、msip 和 mtimectl	95

7.4.31.	<i>fcsr</i>	96
7.4.32.	<i>fflags</i>	96
7.4.33.	<i>frm</i>	97
7.4.34.	<i>cycle</i> 和 <i>cycleh</i>	98
7.4.35.	<i>instret</i> 和 <i>instreth</i>	98
7.4.36.	<i>time</i> 和 <i>timeh</i>	98
7.4.37.	<i>mcounteren</i>	98
7.4.38.	<i>pmpcfg</i> < <i>x</i> > 寄存器.....	99
7.4.39.	<i>pmpaddr</i> < <i>x</i> > 寄存器.....	99
7.4.40.	<i>ucode</i>	99
7.5.	N 级别处理器内核自定义的 CSR.....	100
7.5.1.	<i>mcountinhibit</i>	100
7.5.2.	<i>milm_ctl</i>	100
7.5.3.	<i>mdlm_ctl</i>	101
7.5.4.	<i>mnvec</i>	101
7.5.5.	<i>msubm</i>	102
7.5.6.	<i>mdcause</i>	102
7.5.7.	<i>mcache_ctl</i>	103
7.5.8.	<i>mmisc_ctl</i>	104
7.5.9.	<i>msavestatus</i>	105
7.5.10.	<i>msaveepc1</i> 和 <i>msaveepc2</i>	105
7.5.11.	<i>msavecause1</i> 和 <i>msavecause2</i>	106
7.5.12.	<i>msavedcause1</i> 和 <i>msavedcause2</i>	106
7.5.13.	<i>pushmsubm</i>	106
7.5.14.	<i>mtvt2</i>	107
7.5.15.	<i>jalmnxti</i>	107
7.5.16.	<i>pushmcause</i>	107
7.5.17.	<i>pushmepc</i>	108
7.5.18.	<i>sleepvalue</i>	108
7.5.19.	<i>txevt</i>	108
7.5.20.	<i>wfe</i>	109
7.5.21.	<i>mcfg_info</i>	109
7.5.22.	<i>micfg_info</i>	110
7.5.23.	<i>mdefg_info</i>	112
7.5.24.	<i>mppicfg_info</i>	113
7.5.25.	<i>mfiocfg_info</i>	114
7.5.26.	<i>mecc_lock</i>	115
7.5.27.	<i>mecc_code</i>	115
8.	N 级别处理器内核物理存储器保护机制介绍.....	117
8.1.	PMP 简介.....	117

8.2.	PMP 的 CSR 寄存器	117
8.2.1.	PMP 的 CSR 寄存器列表.....	117
8.2.2.	<i>pmpcfg</i> < <i>x</i> > 寄存器.....	118
8.2.3.	<i>pmpaddr</i> < <i>x</i> > 寄存器.....	120
8.3.	PMP 表项的锁定	120
8.4.	PMP 保护触发异常.....	121
8.5.	PMP 表项匹配的优先级	121
8.6.	PMP 表项匹配的跨边界情况.....	122
8.7.	PMP 权限检测的原则	122
9.	N 级别处理器内核低功耗机制介绍	124
9.1.	进入休眠状态.....	124
9.2.	退出休眠状态.....	125
9.2.1.	NMI 唤醒	125
9.2.2.	中断唤醒.....	125
9.2.3.	Event 唤醒.....	126
9.2.4.	Debug 唤醒.....	126
9.3.	WAIT FOR INTERRUPT 机制	126
9.4.	WAIT FOR EVENT 机制	126
10.	N 级别处理器 ECC 处理机制介绍.....	128
10.1.	NUCLEI ECC 总体介绍	128
10.2.	NUCLEI ECC 相关的 CSR 寄存器	131

表格清单

表 3-1 MCAUSE 寄存器中的 EXCEPTION CODE 以及对应的 MDCAUSE.....	21
表 6-1 TIMER 寄存器的存储器映射地址	61
表 6-2 寄存器 MTIMECTL 的比特域.....	62
表 6-3 寄存器 MSIP 的比特域	63
表 6-4 寄存器 MSFTRST 的比特域.....	64
表 6-5 ECLIC 中断源编号和分配.....	67
表 6-6 ECLIC 寄存器的单元内地址偏移量	68
表 6-7 寄存器 CLICCFG 的比特域	69
表 6-8 寄存器 CLICINFO 的比特域	69
表 6-9 寄存器 MTH 的比特域.....	70
表 6-10 寄存器 CLICINTIP[I]的比特域.....	70
表 6-11 寄存器 CLICINTIP[I]的比特域	70
表 6-12 寄存器 CLICINTATTR[I]的比特域.....	70
表 7-1 N 级别处理器内核支持的 CSR 寄存器列表.....	77
表 7-2 MSTATUS 寄存器各控制位.....	84
表 7-3 MTVEC 寄存器各控制位	88
表 7-4 MTVT 对齐方式.....	88
表 7-5 MEPC 寄存器各控制位	89
表 7-6 MCAUSE 寄存器各控制位	90
表 7-7 MINSTATUS 寄存器的控制位	92
表 7-8 FCSR 寄存器各控制位	96
表 7-9 舍入模式位	97
表 7-10 MCOUNTEREN 寄存器各控制位	98
表 7-11 UCODE 寄存器各控制位	99
表 7-12 MCOUNTINHIBIT 寄存器各控制位.....	100
表 7-13 MILM_CTL 寄存器	100
表 7-14 MDLM_CTL 寄存器.....	101
表 7-15 MSUBM 寄存器各控制位.....	102
表 7-16 MDCAUSE 寄存器各控制位	102
表 7-17 MCACHE_CTL 寄存器各控制位	103
表 7-18 MMISC_CTL 寄存器各控制位.....	104
表 7-19 MSAVESTATUS 寄存器各控制位	105
表 7-20 MTVT2 寄存器各控制位.....	107
表 7-21 SLEEPVALUE 寄存器各控制位.....	108
表 7-22 TXEVT 寄存器各控制位.....	109
表 7-23 WFE 寄存器各控制位	109
表 7-24 MCFG_INFO 寄存器描述.....	110
表 7-25 MICFG_INFO 寄存器描述.....	110

表 7-26 MDCFG_INFO 寄存器描述.....	112
表 7-27 MPPICFG_INFO 寄存器描述	113
表 7-28 MFIOCFG_INFO 寄存器描述	114
表 7-29 MECC_LOCK 寄存器的控制位.....	115
表 7-30 MECC_CODE 寄存器的控制位	115
表 8-1 PMP 的 CSR 寄存器列表	118
表 8-2 PMP<X>CFG 中各控制域.....	119
表 8-3 寄存器 PMP<X>CFG 中的“A”比特域.....	119
表 8-4 PMPADDR<X>中各控制域.....	120
表 8-5 PMP 表项设定的地址区间大小	120
表 10-1 NUCLEI ECC 相关 CSR 寄存器列表.....	131

图片清单

图 3-1 异常响应总体过程	20
图 3-2 进入/退出异常时 CSR 寄存器的变化.....	24
图 3-3 退出异常总体过程.....	25
图 4-1 NMI 响应总体过程.....	29
图 4-2 进入/退出 NMI 时 CSR 寄存器的变化	31
图 4-3 退出 NMI 总体过程.....	32
图 4-4 N 级别处理器内核两级 NMI/异常状态堆栈机制示意图.....	34
图 5-1 中断类型示意图	40
图 5-2 中断仲裁示意图.....	42
图 5-3 响应中断总体过程.....	43
图 5-4 进入/退出中断时 CSR 寄存器的变化	46
图 5-5 退出中断总体过程	47
图 5-6 中断向量表示意图	49
图 5-7 中断嵌套示意图	51
图 5-8 中断咬尾示意图.....	52
图 5-9 中断的非向量处理模式示例（总是支持嵌套）	54
图 5-10 三个先后到来的（非向量处理模式）中断形成嵌套	55
图 5-11 中断咬尾示意图.....	56
图 5-12 中断的向量处理模式示例	57
图 5-13 中断的向量处理模式示例（支持中断嵌套）	59
图 5-14 三个先后到来的（向量处理模式）中断形成嵌套	60
图 6-1 ECLIC 逻辑结构示意图.....	65
图 6-2 ECLIC 关系结构图.....	66
图 6-3 寄存器 CLICINTCTL[I]的格式示例.....	73
图 6-4 LEVEL 的数字值解读方式.....	74
图 6-5 CLICCFG 设置的若干示例	74
图 7-1 MISA 寄存器低 26 位各域表示的模块化指令子集.....	82
图 7-2 FS 域表示的状态编码.....	86
图 8-1 CSR 寄存器 PMPCFG0 的格式.....	118

1. N 级别处理器内核指令集与 CSR 介绍

1.1. RISC-V 指令集介绍

N 级别处理器内核遵循的标准 RISC-V 指令集文档版本为：“指令集文档版本 2.2”（[riscv-spec-v2.2.pdf](https://riscv.org/specifications/)）。用户可以在 RISC-V 基金会的网站上需注册便可关注并免费下载其完整原文（<https://riscv.org/specifications/>）。

除了 RISC-V “指令集文档版本 2.2” 英文原文之外，用户还可以参阅中文书籍《手把手教你设计 CPU——RISC-V 处理器篇》的附录 A、附录 C~G 部分，其使用通俗易懂的中文对 RISC-V 指令集标准进行了系统讲解。

1.2. N 级别处理器内核支持指令集

RISC-V 指令集基于模块化设计，可以根据配置进行灵活组合。Nuclei N 级别处理器内核支持的是如下模块化指令集：

- RV32 架构：32 位地址空间，通用寄存器宽度 32 位。
- I：支持 32 个通用整数寄存器。
- M：支持整数乘法与除法指令
- C：支持编码长度为 16 位的压缩指令，提高代码密度。
- A：支持原子操作指令。
- F：支持单精度浮点指令。
- D：支持双精度浮点指令。
- N：支持用户异常和中断在用户模式下处理。
- P：支持数字信号处理指令。

按照 RISC-V 架构命名规则，以上指令子集的组合可表示为 RV32IMAC，RV32IMFC，RV32IMAFDC，RV32IMAFDCNP 等。RISC-V 架构定义 IMAFD 为通用组合（General Purpose），以字母 G 表示，因此 RV32IMAFDC 也可表示为 RV32GC。

注意：

- N 级别处理器内核是 Nuclei 的一个处理器内核（**Processor Core**）级别，这个级别中包含了 N200, N300, N600, N900 四个系列，每个系列中具体每一款处理器内核（譬如 N203, N205 等）支持的指令子集组合可能略有差异（譬如 N203 可以配置为仅支持 RV32IMC），请参见各个系列的简明数据手册了解其详情。
- N100 系列是一个极小面积的处理器内核系列，其进行了大量的架构删减以求达到极小的硬件面积，因此其不遵循本文档中定义的架构细节。有关 N100 系列支持的指令集和架构详情请参见《Nuclei N100 系列指令架构手册》。

另外，本文中所描述的架构并不支持 TEE（**Turst Execution Environment**）安全特性，如需了解 TEE 相关架构的更多信息，请参见《Nuclei TEE Architecture》文档。

1.3. CSR 寄存器

RISC-V 的架构中定义了一些控制和状态寄存器（**Control and Status Register, CSR**），用于配置或记录一些处理器核的运行状态。CSR 寄存器是处理器核内部的寄存器，使用其专有的 12 位地址编码空间。详情请参见第 7 章。

2. N 级别处理器内核特权架构介绍

2.1. 总体介绍

N 级别处理器内核遵循的标准 RISC-V 特权架构文档版本为：“特权架构文档版本 1.11”（[riscv-privileged-v1.11.pdf](https://riscv.org/specifications/riscv-privileged-v1.11.pdf)）。用户可以在 RISC-V 基金会的网站上需注册便可关注并免费下载其完整原文（<https://riscv.org/specifications/>）。

除了 RISC-V “特权架构文档版本 1.11” 英文原文之外，用户还可以参阅中文书籍《手把手教你设计 CPU——RISC-V 处理器篇》的附录 A、附录 C~G 部分，其使用通俗易懂的中文对 RISC-V 特权架构标准进行了系统讲解。

2.2. 特权模式（Privilege Modes）

N 级别处理器内核支持两个特权模式（Privilege Modes）：

- 机器模式（Machine Mode）是必须的模式，该 Privilege Mode 的编码是 0x3。
- 用户模式（User Mode）是可配置的模式，该 Privilege Mode 的编码是 0x0。
 - 如果配置了 User Mode，则也支持 PMP（Physical Memory Protection）单元对物理地址区间进行隔离和保护。

注意：如果配置了 TEE 特性，则还支持监督模式（Supervisor Mode），如需了解 TEE 相关架构的更多信息，请参见《Nuclei TEE Architecture》文档。

2.2.1. 机器模式（Machine Mode）

N 级别处理器内核有关 Machine Mode 的关键要点如下：

- 处理器内核被复位后，默认处于 Machine Mode。
- 在 Machine Mode 下，程序能够访问所有的 CSR 寄存器。
- 在 Machine Mode 下，程序能够访问所有的物理地址区域（除了 PMP 禁止访问的区域之

外，请参见第 8 章了解 PMP 的详情)。

2.2.2. 用户模式 (User Mode)

N 级别处理器内核有关 User Mode 的关键要点如下：

- 在 User Mode 下只能够访问 User Mode 限定的 CSR 寄存器，请参见第 7.3 节了解详情。
- 在 User Mode 下只能够访问 PMP 设定权限的物理地址区域，请参见第 8 章了解 PMP 的详情。

2.2.3. 机器子模式 (Machine Sub-Mode)

N 级别处理器内核的 Machine Mode 可能处于四种不同的状态下，将之称之为机器子模式 (Machine Sub-Mode)：

- 正常机器模式 (该 Machine Sub-Mode 的编码是 0x0)：
 - 处理器内核被复位之后，处于此子模式之下。处理器复位后如果不产生异常、NMI、中断，则一直正常运行于此模式之下。
- 异常处理模式 (该 Machine Sub-Mode 的编码是 0x2)：
 - 响应异常后处理器内核处于此状态。
 - 有关异常机制的详情，请参见第 3 章。
- NMI 处理模式 (该 Machine Sub-Mode 的编码是 0x3)：
 - 响应 NMI 后处理器内核处于此状态。
 - 有关 NMI 机制的详情，请参见第 4 章。
- 中断处理模式 (该 Machine Sub-Mode 的编码是 0x1)：
 - 响应中断后处理器内核处于此状态。
 - 有关中断机制的详情，请参见第 5 章。

处理器内核当前处于的 Machine Sub-Mode 反映在 CSR 寄存器 msubm 的 TYP 域中，因此软件可以通过读取此 CSR 寄存器查看当前处于的 Machine Sub-Mode。有关 msubm 寄存器的详情，

请参见第 7.5.5 节。

注意：在 RISC-V 架构中，进入异常、NMI 或者中断也被统称为 Trap。

2.2.4. 模式 (Mode) 的查看

处理器模式 (Mode) 查看的关键要点如下：

- 根据 RISC-V 的架构定义，处理器当前的 Machine Mode 或者 User Mode 并没有反映在任何软件可见的寄存器中（处理器内核会维护一个对软件不可见的硬件寄存器），因此软件程序无法通过读取任何寄存器而查看当前自己所处的 Machine Mode 或者 User Mode。
- N 级别处理器内核的四种机器子模式 (Machine Sub-Mode) 反映在 CSR 寄存器 msubm 的 TYP 域中，因此软件可以通过读取此 CSR 寄存器查看当前处于的 Machine Sub-Mode。

2.2.5. Machine Mode 到 User Mode 的切换

在 Machine Mode 下可以直接执行 mret 指令。从 Machine Mode 切换到 User Mode 只能通过执行 mret 指令发生。由于如第 2.2.3 节中所述，Machine Mode 可能处于四种不同的状态下，分别介绍如下：

- 如果是在正常机器模式下，执行 mret 指令的硬件行为与异常处理模式下执行 mret 指令的行为相同，请参见第 3.5 节了解其详情。
- 因此，如果在正常机器模式下，希望从 Machine Mode 切换到 User Mode，那么需要软件先修改 mstatus 的 MPP 域的值，然后执行 mret 指令达到模式切换的效果。典型的程序代码片段如下所示：

```
/* Switch Machine sub-mode to User mode */
li t0, MSTATUS_MPP // MSTATUS_MPP 的值为 0x00001800，即对应 mstatus 的 MPP 位域，请参
// 见第 7.4.7 节了解 mstatus 的位域详情。
csrr mstatus, t0 // 将 mstatus 寄存器的 MPP 位域清为 0
la t0, 1f // 将前面的标签 1 所在的 PC 地址赋值给 t0
csrr mepc, t0 // 将 t0 的值赋值给 CSR 寄存器 mepc
mret // 执行 mret 指令，则会将模式切换到 User Mode，并且从前的标签 1 处开始执行
// 程序 (标签 1 即为 mret 的下一条指令的位置)
1: // 标签 1 的位置
```

- 如果是在异常处理模式下，执行 **mret** 指令的硬件行为，请参见第 3.5 节了解其详情。
 - 通常来说，**mret** 指令用于从异常处理模式下退出至进入异常之前的模式。
 - 如果明确希望从 **Machine Mode** 退出至 **User Mode**（或者正常机器模式），那么需要软件先修改 **mstatus** 的 **MPP** 域的值，然后执行 **mret** 指令达到模式切换的效果。
- 如果是在中断处理模式下，执行 **mret** 指令的硬件行为，请参见第 5.7 节了解其详情。
 - 通常来说，**mret** 指令用于从中断处理模式下退出至进入中断之前的模式。
 - 如果明确希望从 **Machine Mode** 退出至 **User Mode**（或者正常机器模式），那么需要软件先修改 **mstatus** 的 **MPP** 域的值，然后执行 **mret** 指令达到模式切换的效果。
- 如果是在 **NMI** 处理模式下，执行 **mret** 指令的硬件行为，请参见第 4.4 节了解其详情。
 - 通常来说，**mret** 指令用于从 **NMI** 处理模式下退出至正常机器模式。
 - 如果明确希望从 **Machine Mode** 退出至 **User Mode**（或者正常机器模式），那么需要软件先修改 **mstatus** 的 **MPP** 域的值，然后执行 **mret** 指令达到模式切换的效果。

注意：

- 如果在 **User Mode** 下直接执行 **mret** 指令会产生非法指令（**Illegal Instruction**）异常。
- 在 **User Mode** 下处理器只能够访问 **PMP** 设定权限的物理地址区域，因此在切换到 **User Mode** 之前，需要配置 **PMP** 相关寄存器设定 **User Mode** 可以访问的物理地址区域，请参见第 8 章了解其详情。

2.2.6. User Mode 到 Machine Mode 的切换

N 级别处理器内核从 **User Mode** 切换到 **Machine Mode** 只能通过异常、响应中断或者 **NMI** 的方式发生：

- 响应异常进入异常处理模式。请参见第 3.4 节了解其详情。
 - 注意：软件可以通过调用 **ecall** 指令强行进入 **ecall** 异常处理函数。

- 响应中断进入中断处理模式。请参见第 5.6 节了解其详情。
- 响应 NMI 进入 NMI 处理模式。请参见第 4.3 节了解其详情。

2.2.7. 中断、异常、NMI 的嵌套

中断和异常能自我发生嵌套，NMI 无法自我嵌套：

- 在 NMI 处理模式下，如果再次发生 NMI，新来的 NMI 会被屏蔽掉，因此，NMI 无法自我嵌套，请参见第 4.6 节了解详情。
- 在异常处理模式下，如果再次发生异常，这属于异常嵌套情形，请参见第 3.7 节了解详情。
- 在中断处理模式下，如果再次发生中断，这属于中断嵌套情形，请参见第 5.11 节了解详情。

中断、异常和 NMI 彼此之间也可能会发生嵌套，存在如下情形：

- 在中断处理模式下发生了异常，则进入异常处理模式。
- 在 NMI 处理模式下发生了异常，则进入异常处理模式。
- 在中断处理模式下发生了 NMI，则进入 NMI 处理模式。
- 在异常处理模式下发生了 NMI，则进入 NMI 处理模式。
- 注意：在 NMI 和异常模式下默认由于全局中断位被硬件自动关闭，因此不会再响应中断。

为了保证异常和 NMI 彼此之间发生嵌套后还能够恢复到之前的状态（Recoverable），N 级别处理器内核实现了一种“两级 NMI/异常状态堆栈（Two Levels of NMI/Exception State Save Stacks）”技术，请参见第 4.6 节了解更多详情。

2.3. 物理存储器保护（PMP）

由于 N 级别处理器内核是面向微控制器领域的低功耗内核，其不支持虚拟地址管理单元（Memory Management Unit），因此所有的地址访问操作都是使用的物理地址。为了根据不同的存储器物理地址区间和不同的 Privilege Mode 进行权限隔离和保护，N 级别处理器内核支持物理存储保护，即（Physical Memory Protection, PMP）单元。有关 PMP 单元的详情，请参见第 8 章。

3. N 级别处理器内核异常机制介绍

3.1. 异常概述

异常（Exception）机制，即处理器核在顺序执行程序指令流的过程中突然遇到了异常的事情而中止执行当前的程序，转而去处理该异常，其要点如下：

- 处理器遇到的“异常的事情”称为异常（Exception）。异常是由处理器内部事件或程序执行中的事件引起的，譬如本身硬件故障、程序故障，或者执行特殊的系统服务指令而引起的，简而言之是一种内因。
- 异常发生后，处理器会进入异常服务处理程序。

3.2. 异常屏蔽

RISC-V 架构中规定异常是不可以被屏蔽的，也就是说一旦发生了异常，处理器一定会停止当前操作转而进入异常处理模式。

3.3. 异常的优先级

处理器内核可能存在多个异常同时发生的情形，因此异常也有优先级。异常的优先级如表 3-1 中所示，异常编号数字越小的异常优先级越高。

3.4. 进入异常处理模式

进入异常时，N 级别处理器内核的硬件行为可以简述如下。注意，下列硬件行为在一个时钟周期内同时完成：

- 停止执行当前程序流，转而从 CSR 寄存器 `mtvec` 定义的 PC 地址开始执行。
- 更新相关 CSR 寄存器，分别是以下几个寄存器：
 - `mcause`（Machine Cause Register）
 - `mdcause`（Machine Detailed Trap Cause Register）

- mepc (Machine Exception Program Counter)
- mtval (Machine Trap Value Register)
- mstatus (Machine Status Register)

■ 更新处理器内核的 Privilege Mode 以及 Machine Sub-Mode。

异常响应总体过程如图 3-1 所示。

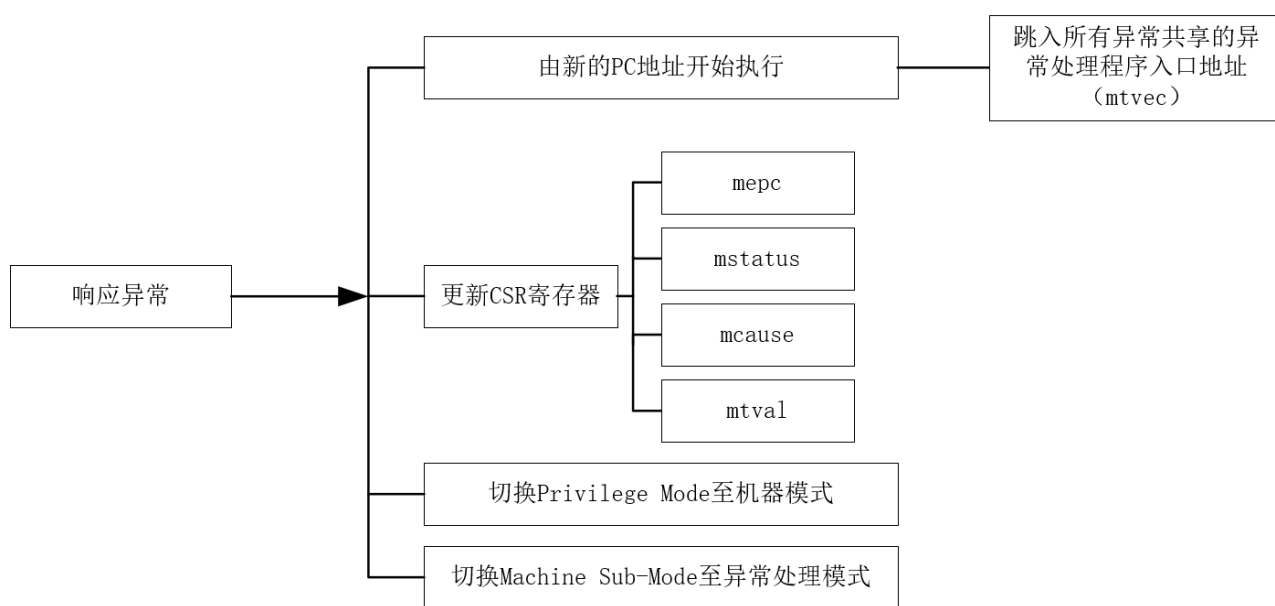


图 3-1 异常响应总体过程

下文将分别予以详述。

3.4.1. 从 mtvec 定义的 PC 地址开始执行

N 级别处理器内核遇到异常后跳入的 PC 地址由 CSR 寄存器 mtvec 指定。

mtvec 寄存器是一个可读可写的 CSR 寄存器，因此软件可以编程更改其中的值。mtvec 寄存器的详细格式如表 7-3 所示。

3.4.2. 更新 CSR 寄存器 mcause 和 mdcause

N 级别处理器内核在进入异常时，CSR 寄存器 mcause 和 mdcause 被同时（硬件自动）更新，以反映当前的异常种类，软件可以通过读此寄存器查询造成异常的具体原因。

mcause 寄存器的详细格式如表 7-6 所示，其中低 5 位为异常编号域，用于指示各种不同的异常类型，如表 3-1 所示。

表 3-1 mcause 寄存器中的 Exception Code 以及对应的 mdcause

异常编号 (Exception Code)	异常和中断类型	同步/异步	描述
0	指令地址非对齐 (Instruction address misaligned)	同步	指令 PC 地址非对齐。 注意：该异常类型在配置了“C”扩展指令子集的处理器中不可能发生。
1	指令访问错误 (Instruction access fault)	同步	取指令访问错误。 mdcause 提供详细的指令访问错误类型： ■ 1: PMP 检测指令访问出错 ■ 2: 指令访问返回总线错误
2	非法指令 (Illegal instruction)	同步	非法指令。
3	断点 (Breakpoint)	同步	RISC-V 架构定义了 EBREAK 指令，当处理器执行到该指令时，会发生异常进入异常服务程序。该指令往往用于调试器 (Debugger) 使用，譬如设置断点
4	读存储器地址非对齐 (Load address misaligned)	同步	Load 指令访问地址非对齐。 注意：N 级别处理器内核支持可配置的地址非对齐的数据存储器读写操作，如果没有配置此选项或者未打开此开关，访问地址非对齐时会产生此异常。
5	读存储器访问错误 (Load access fault)	非精确异步	Load 指令访问错误。 mdcause 提供详细的读存储器访问错误类型： ■ 1: PMP 检测读操作访问存储器出错 ■ 2: 读操作访问存储器返回总线错误 ■ 3: NICE 长指令错误
6	写存储器和 AMO 地址非对齐 (Store/AMO address misaligned)	同步	Store 或者 AMO 指令访问地址非对齐。注意：N 级别处理器内核支持可配置的地址非对齐的数据存储器读写操作，如果没有配置此选项或者未打

			开此开关，访问地址非对齐时会产生此异常。AMO 指令不支持非对其访问。
7	写存储器 and AMO 访问错误 (Store/AMO access fault)	非精确异步	Store 或者 AMO 指令访问错误。 mdcause 提供详细的写存储器访问错误类型： <ul style="list-style-type: none"> 1: PMP 检测写操作访问存储器出错 2: 写操作访问存储器返回总线错误
8	用户模式环境调用 (Environment call from U-mode)	同步	User Mode 下执行 ecall 指令。 RISC-V 架构定义了 ecall 指令，当处理器执行到该指令时，会发生异常进入异常服务程序。该指令往往供软件使用，强行进入异常模式。
11	机器模式环境调用 (Environment call from M-mode)	同步	Machine Mode 下执行 ecall 指令。 RISC-V 架构定义了 ecall 指令，当处理器执行到该指令时，会发生异常进入异常服务程序。该指令往往供软件使用，强行进入异常模式。

mdcause 寄存器的详细格式如表 7-16 所示，其中低 2 位为异常子编号域，用于指示详细的异常类型，如表 3-1 所示。

3.4.3. 更新 CSR 寄存器 mepc

N 级别处理器内核退出异常时的返回地址由 CSR 寄存器 mepc (Machine Exception Program Counter) 保存。在进入异常时，硬件将自动更新 mepc 寄存器的值，该寄存器将作为退出异常的返回地址，在异常结束之后，能够使用它保存的 PC 值回到之前被异常停止执行的程序点。

注意：

- 出现异常时，异常返回地址 mepc 的值被更新为当前发生异常的指令 PC。
- 虽然 mepc 寄存器会在异常发生时自动被硬件更新，但是 mepc 寄存器本身也是一个可读可写的寄存器，因此软件也可以直接写该寄存器以修改其值。

3.4.4. 更新 CSR 寄存器 mtval

N 级别处理器内核在进入异常时，硬件将自动更新 CSR 寄存器 mtval (Machine Trap Value Register)，以反映引起当前异常的存储器访问地址或者指令编码：

- 如果是由存储器访问造成的异常，譬如遭遇硬件断点、取指令、存储器读写造成的异常，则将存储器访问的地址更新到 **mtval** 寄存器中。
- 如果是由非法指令造成的异常，则将该指令的指令编码更新到 **mtval** 寄存器中。

3.4.5. 更新 CSR 寄存器 **mstatus**

mstatus 寄存器的详细格式如表 7-2 所示，N 级别处理器内核在进入异常时，硬件将自动更新 CSR 寄存器 **mstatus**（Machine Status Register）的某些域：

- **mstatus.MPIE** 域的值被更新为异常发生前 **mstatus.MIE** 域的值，**mstatus.MPIE** 域的作用是在异常结束之后，能够使用 **mstatus.MPIE** 的值恢复出异常发生之前的 **mstatus.MIE** 值。
- **mstatus.MIE** 域的值则被更新成为 0（意味着进入异常服务程序后中断被全局关闭，所有的中断都将被屏蔽不响应）。
- **mstatus.MPP** 域的值被更新为异常发生前的 Privilege Mode，**mstatus.MPP** 域的作用是在异常结束之后，能够使用 **mstatus.MPP** 的值恢复出异常发生之前的 Privilege Mode。

3.4.6. 更新 Privilege Mode

异常需要在机器模式（Machine Mode）下处理，在进入异常时，处理器内核的 Privilege Mode 被更新为机器模式。

3.4.7. 更新 Machine Sub-Mode

N 级别处理器内核的 Machine Sub-Mode 实时反映在 CSR 寄存器 **msubm.TYP** 域中。在进入异常时，处理器内核的 Machine Sub-Mode 被更新为异常处理模式，因此：

- CSR 寄存器 **msubm.PTYP** 域的值被更新为异常发生前的 Machine Sub-Mode（**msubm.TYP** 域的值），**msubm.PTYP** 域的作用是在异常结束之后，能够使用 **msubm.PTYP** 的值恢复出异常发生之前的 Machine Sub-Mode 值。

- CSR 寄存器 `mstatus.TYP` 域的值则被更新为“异常处理模式”以实时反映当前的模式已经是“异常处理模式”。

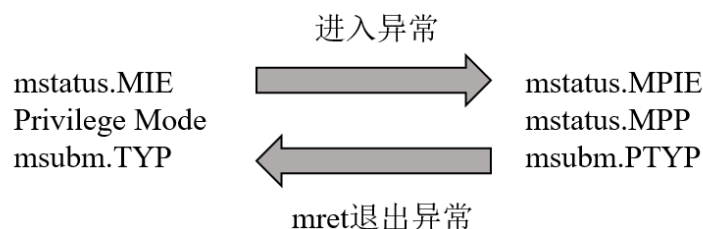


图 3-2 进入/退出异常时 CSR 寄存器的变化

3.5. 退出异常处理模式

当程序完成异常处理之后，最终需要从异常服务程序中退出。

由于异常处理处于 **Machine Mode** 下，所以退出异常时，软件必须使用 `mret` 指令。处理器执行 `mret` 指令后的硬件行为如下。注意，下列硬件行为在一个时钟周期内同时完成：

- 停止执行当前程序流，转而从 CSR 寄存器 `mepc` 定义的 PC 地址开始执行。
- 更新 CSR 寄存器 `mstatus` (Machine Status Register)，并更新处理器内核的 **Privilege Mode** 以及 **Machine Sub-Mode**。

退出异常的总体过程如所图 3-3 示。

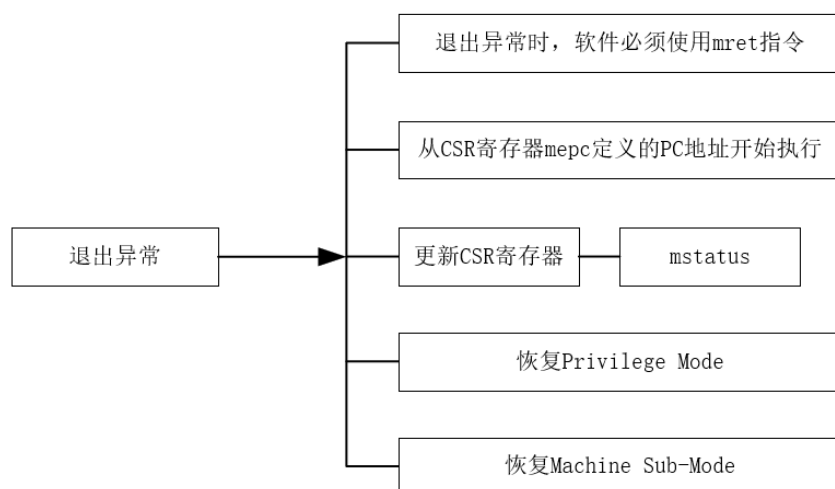


图 3-3 退出异常总体过程

下文将分别予以详述。

3.5.1. 从 mepc 定义的 PC 地址开始执行

在进入异常时，mepc 寄存器被同时更新，以反映当时遇到异常的指令 PC 值。通过这个机制，意味着 mret 指令执行后处理器回到了当时遇到异常的指令的 PC 地址，从而可以继续执行之前被中止的程序流。

注意：退出异常之前可能需要使用软件更新 mepc 的值。例如，如果异常由 ecall 或 ebreak 产生，由于 mepc 的值被更新为 ecall 或 ebreak 指令自己的 PC。因此在异常返回时，如果直接使用 mepc 保存的 PC 值作为返回地址，则会再次跳回 ecall 或者 ebreak 指令，从而造成死循环（执行 ecall 或者 ebreak 指令导致重新进入异常）。正确的做法是在异常处理程序中软件改变 mepc 指向下一条指令，由于现在 ecall/ebreak 都是 4 字节指令，因此改写设定 $mepc = mepc + 4$ 即可。

3.5.2. 更新 CSR 寄存器 mstatus

mstatus 寄存器的详细格式如表 7-2 所示。在执行 mret 指令后，硬件将自动更新 CSR 寄存器

mstatus 的某些域:

- mstatus.MIE 域的值被恢复为当前 mstatus.MPIE 的值。
- 当前 mstatus.MPIE 域的值则被更新为 1。
- mstatus.MPP 域的更新值分为以下两种情形:
 - 配置了用户模式 U-mode 时, mstatus.MPP 被更新为 0x0。
 - 没有配置用户模式 U-mode 时, mstatus.MPP 被更新为 0x3。

在进入异常时, mstatus.MPIE 的值曾经被更新为异常发生前的 mstatus.MIE 值, 而 mret 指令执行后, 将 mstatus.MIE 域的值恢复为 mstatus.MPIE 的值。通过这个机制, 则意味着 mret 指令执行后, 处理器的 mstatus.MIE 值被恢复成异常发生之前的值 (假设之前的 mstatus.MIE 值为 1, 则意味着中断被重新全局打开)。

3.5.3. 更新 Privilege Mode

在进入异常时, mstatus.MPP 的值曾经被更新为异常发生前的 Privilege Mode, 而在执行 mret 指令后, 处理器的 Privilege Mode 被恢复为 mstatus.MPP 的值, 通过这个机制, 保证了处理器回到了异常发生前的处理器的 Privilege Mode。

3.5.4. 更新 Machine Sub-Mode

N 级别处理器内核的 Machine Sub-Mode 实时反映在 CSR 寄存器 msubm.TYP 域中。在执行 mret 指令后, 硬件将自动恢复处理器的 Machine Sub-Mode 为 msubm.PTYP 域的值:

- 在进入异常时, msubm.PTYP 域的值曾经被更新为异常发生前的 Machine Sub-Mode 值。而使用 mret 指令退出异常后, 硬件将处理器 Machine Sub-Mode 的值恢复为 msubm.PTYP 域的值, 如图 4-2 所示。通过这个机制, 则意味着退出异常后, 处理器的 Machine Sub-Mode 被恢复成异常发生之前的 Machine Sub-Mode。

3.6. 异常服务程序

当处理器进入异常后，即开始从 `mtvec` 寄存器定义的 PC 地址执行新的程序，该程序通常为异常服务程序，并且程序还可以通过查询 `mcause` 中的异常编号（Exception Code）以及 `mdcause` 中的详细异常信息决定进一步跳转到更具体的异常服务程序。譬如当程序查询 `mcause` 中的值为 `0x2`，则得知该异常是非法指令错误（Illegal Instruction）引起的，因此可以进一步跳转到非法指令错误异常服务子程序中去。

注意：由于进入异常和退出异常机制中没有硬件自动保存和恢复上下文的操作，因此需要软件明确地使用（汇编语言编写的）指令进行上下文的保存和恢复。请参见《Nuclei_N 级别处理器内核 SDK 使用说明》结合一个完整的异常服务程序代码示例对其进行理解。

3.7. 异常嵌套

N 级别处理器内核支持两级 NMI/异常状态堆栈（Two Levels of NMI/Exception State Save Stacks），更多细节请参见 4.6 节。

4. N 级别处理器内核 NMI 机制介绍

4.1. NMI 概述

NMI (Non-Maskable Interrupt) 是处理器内核的一根特殊的输入信号，往往用于指示系统层面的紧急错误（譬如外部的硬件故障等）。在遇到 NMI 之后，处理器内核应该立即中止执行当前的程序，转而去处理该 NMI 错误。

4.2. NMI 屏蔽

N 级别处理器内核中 NMI 是不可以被屏蔽的，也就是说一旦发生了 NMI，处理器一定会停止当前操作转而处理 NMI。

4.3. 进入 NMI 处理模式

进入 NMI 处理模式时，N 级别处理器内核的硬件行为可以简述如下。注意，下列硬件行为在一个时钟周期内同时完成：

- 停止执行当前程序流，转而从 CSR 寄存器 `mnvec` 定义的 PC 地址开始执行。
- 更新相关 CSR 寄存器，分别是以下几个寄存器：
 - `mepc` (Machine Exception Program Counter)
 - `mstatus` (Machine Status Register)
 - `mcause` (Machine Cause Register)
- 更新处理器内核的 Privilege Mode 以及 Machine Sub-Mode。

NMI 响应总体过程如图 4-1 所示。

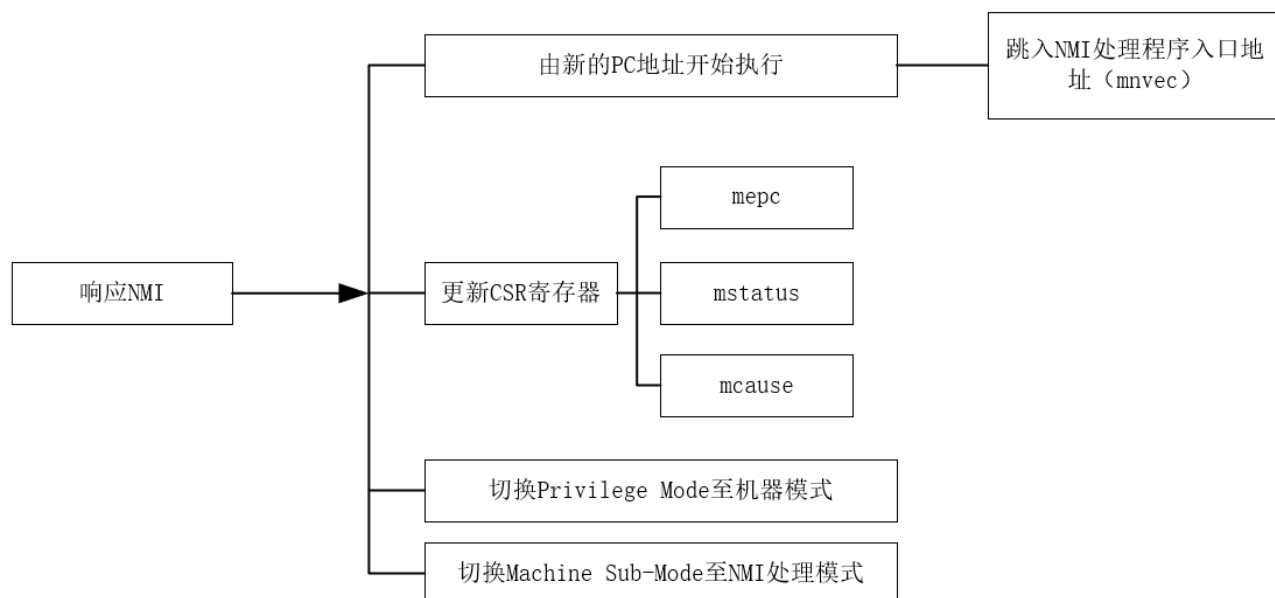


图 4-1 NMI 响应总体过程

下文将分别予以详述。

4.3.1. 从 mnvec 定义的 PC 地址开始执行

N 级别处理器内核遇到 NMI 后跳入的 PC 地址由 CSR 寄存器 mnvec 指定。mnvec 寄存器的值有以下两种情况：

- 当 mmisc_ctl[9]=1 时，mnvec 寄存器的值等于 mtvec，即 NMI 与异常拥有相同的 Trap 入口地址。
- 当 mmisc_ctl[9]=0 时，mnvec 寄存器的值等于 reset_vector，reset_vector 为处理器被 reset 后的 PC 的值。

4.3.2. 更新 CSR 寄存器 mepc

N 级别处理器内核退出 NMI 时的返回地址由 CSR 寄存器 mepc（Machine Exception Program Counter）保存。在进入 NMI 时，硬件将自动更新 mepc 寄存器的值，该寄存器将作为退出 NMI 的返回地址，在 NMI 结束之后，能够使用它保存的 PC 值回到之前被停止执行的程序点。

注意：

- 出现 NMI 时，NMI 返回地址 `mepc` 被指向下一条尚未执行的指令（因为 NMI 时的指令已经被正确执行）。那么在退出 NMI 后，程序便会回到之前的程序点，从下一条指令开始重新执行。
- 虽然 `mepc` 寄存器会在 NMI 发生时自动被硬件更新，但是 `mepc` 寄存器本身也是一个可读可写的寄存器，因此软件也可以直接写该寄存器以修改其值。

4.3.3. 更新 CSR 寄存器 `mcause`

`mcause` 寄存器的详细格式如表 7-6 所示。N 级别处理器内核在进入 NMI 时，硬件自动保存当前 Trap 的 ID 到 `mcause`，以表明 Trap 的原因。中断、异常以及 NMI 都有各自特殊的 Trap ID。NMI 的 Trap ID 有以下两种值：

- 当 `mmisc_ctl[9]=1` 时，NMI Trap ID 为 `0xfff`。
- 当 `mmisc_ctl[9]=0` 时，NMI Trap ID 为 `0x1`。

通过给每个 Trap 分配特定的 Trap ID，可以识别 Trap 的原因，软件可以根据 Trap 的原因来设计特定的处理程序处理 Trap。

4.3.4. 更新 CSR 寄存器 `mstatus`

`mstatus` 寄存器的详细格式如表 7-2 所示，N 级别处理器内核在进入 NMI 时，硬件将自动更新 CSR 寄存器 `mstatus` 的某些域：

- `mstatus.MPIE` 域的值被更新为 NMI 发生前 `mstatus.MIE` 域的值，如图 4-2 所示。
`mstatus.MPIE` 域的作用是在 NMI 结束之后，能够使用 `mstatus.MPIE` 的值恢复出 NMI 发生之前的 `mstatus.MIE` 值。
- `mstatus.MIE` 域的值则被更新成为 0（意味着进入 NMI 服务程序后中断被全局关闭，所有的中断都将被屏蔽不响应）。
- `mstatus.MPP` 域的值被更新为 NMI 发生前的 Privilege Mode。保存 `mstatus.MPP` 域的作用是在 NMI 结束之后，能够使用 `mstatus.MPP` 的值恢复出 NMI 发生前的 Privilege Mode。

4.3.5. 更新 Privilege Mode

NMI 处理是在机器模式（Machine Mode）下完成的，所以在进入 NMI 时，处理器内核的特权模式（Privilege Mode）切换到机器模式。

4.3.6. 更新 Machine Sub-Mode

N 级别处理器内核的 Machine Sub-Mode 实时反映在 CSR 寄存器 msubm.TYP 域中。在进入 NMI 时，处理器内核的 Machine Sub-Mode 被更新为 NMI 处理模式，因此：

- CSR 寄存器 msubm.PTYP 域的值被更新为 NMI 发生前的 Machine Sub-Mode（msubm.TYP 域的值），如图 4-2 所示。msubm.PTYP 域的作用是在 NMI 结束之后，能够使用 msubm.PTYP 的值恢复出 NMI 发生之前的 Machine Sub-Mode 值。
- CSR 寄存器 msubm.TYP 域的值则被更新为“NMI 处理模式”如图 4-2 所示，以实时反映当前的模式已经是“NMI 处理模式”。

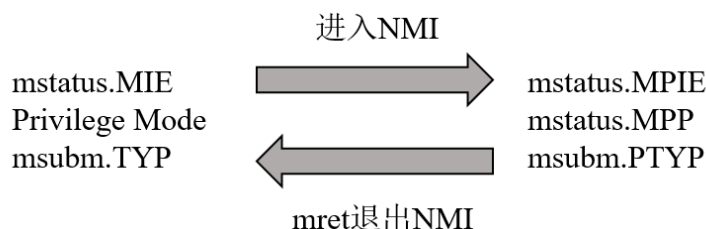


图 4-2 进入/退出 NMI 时 CSR 寄存器的变化

4.4. 退出 NMI 处理模式

当程序完成 NMI 处理之后，最终需要从 NMI 服务程序中退出，并返回主程序。

由于 NMI 处理处于 Machine Mode 下，所以在退出 NMI 时，软件必须使用 mret 指令。处理器执行 mret 指令后的硬件行为如下。注意，下列硬件行为在一个时钟周期内同时完成：

- 停止执行当前程序流，转而从 CSR 寄存器 mepc 定义的 PC 地址开始执行。
- 更新 CSR 寄存器 mstatus。
- 更新 Privilege Mode 以及 Machine Sub-Mode。

退出 NMI 的总体过程如图 4-3 所示。

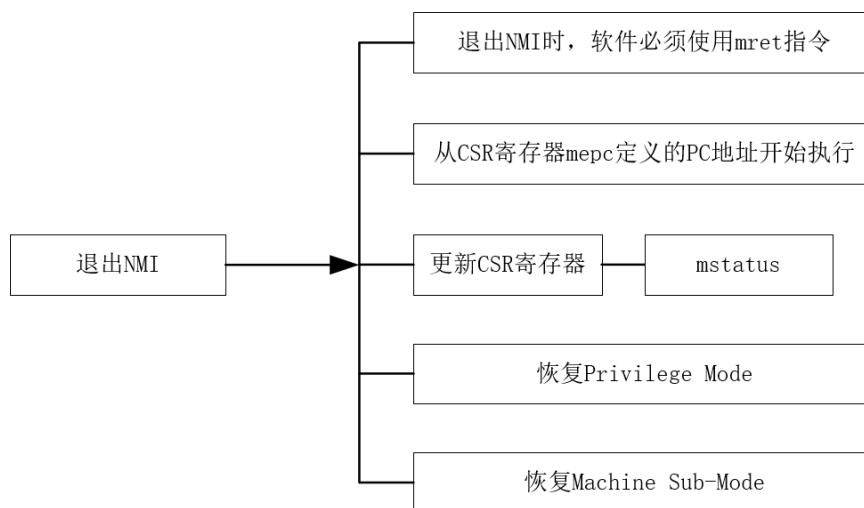


图 4-3 退出 NMI 总体过程

下文将分别予以详述。

4.4.1. 从 mepc 定义的 PC 地址开始执行

在进入 NMI 时，mepc 寄存器被同时更新，以反映当时遇到 NMI 的下一条指令的 PC 值。通过这个机制，意味着 mret 指令执行后处理器回到了当时遇到 NMI 的下一条指令的 PC 地址，从而可以继续执行之前被中止的程序流。

4.4.2. 更新 CSR 寄存器 mstatus

mstatus 寄存器的详细格式如表 7-2 所示，在执行 mret 指令后，硬件将自动更新 CSR 寄存器 mstatus 某些域：

- mstatus.MIE 域的值被恢复为当前 mstatus.MPIE 的值。
- mstatus.MPIE 域的值则被更新为 1。

■ `mstatus.MPP` 域的更新值分为以下两种情形：

- 配置了用户模式 U-mode 时，`mstatus.MPP` 被更新为 `0x0`。
- 没有配置用户模式 U-mode 时，`mstatus.MPP` 被更新为 `0x3`。

在进入 NMI 时，`mstatus.MPIE` 的值曾经被更新为 NMI 发生前的 `mstatus.MIE` 值。而 `mret` 指令执行后，将 `mstatus.MIE` 的值恢复为 `mstatus.MPIE` 的值，如图 4-2 所示。通过这个机制，则意味着 `mret` 指令执行后，处理器的 `mstatus.MIE` 值被恢复成 NMI 发生之前的值（假设之前的 `mstatus.MIE` 值为 1，则意味着中断被重新全局打开）。

4.4.3. 更新 Privilege Mode

在进入 NMI 时，`mstatus.MPP` 的值曾经被更新为 NMI 发生前的 Privilege Mode，而在执行 `mret` 指令后，处理器的 Privilege Mode 被恢复为 `mstatus.MPP` 的值，如图 4-2 所示。通过这个机制，保证了处理器回到了 NMI 发生前的处理器的 Privilege Mode。

4.4.4. 更新 Machine Sub-Mode

N 级别处理器内核的 Machine Sub-Mode 实时反映在 CSR 寄存器 `msubm.TYP` 域中。在执行 `mret` 指令后，硬件将自动恢复处理器的 Machine Sub-Mode 为 `msubm.PTYP` 域的值：

- 在进入 NMI 时，`msubm.PTYP` 域的值曾经被更新为 NMI 发生前的 Machine Sub-Mode 值。而使用 `mret` 指令退出 NMI 后，硬件将处理器 Machine Sub-Mode 的值恢复为 `msubm.PTYP` 域的值，如图 4-2 所示。通过这个机制，则意味着退出 NMI 后，处理器的 Machine Sub-Mode 被恢复成 NMI 发生之前的 Machine Sub-Mode。

4.5. NMI 服务程序

当处理器进入 NMI 后，即开始从 `mnvec` 寄存器定义的 PC 地址执行新的程序，该程序通常为 NMI 服务程序。

注意：由于进入 NMI 和退出 NMI 机制中没有硬件自动保存和恢复上下文的操作，因此需要软件明确地使用（汇编语言编写的）指令进行上下文的保存和恢复。请参见《Nuclei_N 级别处理器内核 SDK 使用说明》结合一个完整的 NMI 服务程序代码示例对其进行理解。

4.6. NMI/异常嵌套

N 级别处理器内核自定义了如图 4-4 所示的两级 NMI/异常状态堆栈（Two Levels of NMI/Exception State Save Stacks），至多保存三级 NMI/异常的处理器状态，可以实现二级可恢复的 NMI/异常嵌套。

注意：由于处理器处于 NMI 状态时，NMI 的响应在硬件上被屏蔽掉了，因此 NMI 无法实现自我嵌套。N 级别处理器内核中的 NMI/异常嵌套只支持以下 3 种嵌套：

- NMI 嵌套异常
- 异常嵌套异常
- 异常嵌套 NMI

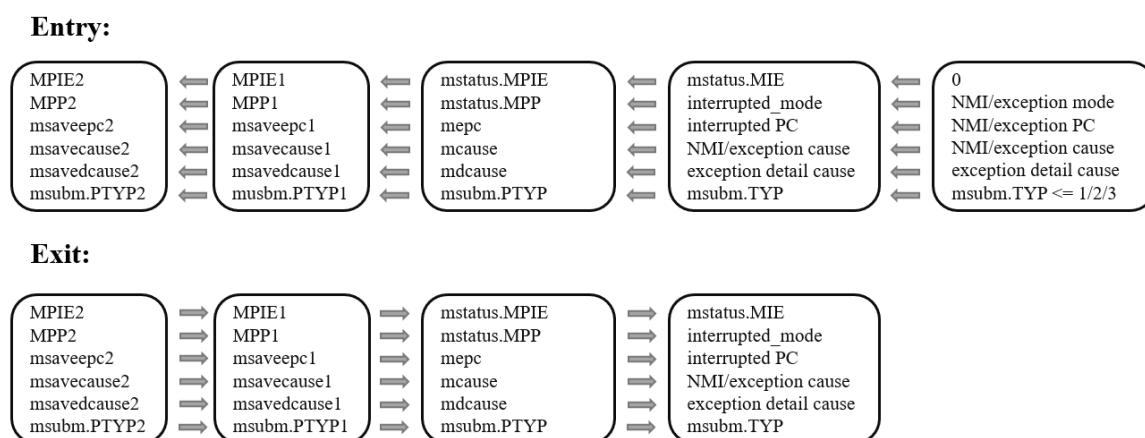


图 4-4 N 级别处理器内核两级 NMI/异常状态堆栈机制示意图

4.6.1. 进入 NMI/异常嵌套

响应 NMI 与异常时，N 级别处理器内核的硬件行为如图 4-4 所示，可以简述如下。

- 停止执行当前程序流，转而从新的 PC 地址开始执行。
 - 如果响应的是异常则从 **mtvec** 所存储的 PC 地址开始执行。
 - 如果响应的是 NMI 则从 **mnvec** 所存储的 PC 地址开始执行。
- 更新相关 CSR 寄存器，分别是以下几个寄存器及其相关的域：
 - **mepc**: 记录当前 NMI/异常发生前的 PC，退出 NMI/异常时可从 **mepc** 恢复 NMI/异常发生前的 PC。
 - **msaveepc1**: 第一级 NMI/异常状态堆栈记录第一级嵌套 NMI/异常（被当前 NMI/异常嵌套的 NMI/异常）发生前的 PC，亦即当前 NMI/异常发生前的 **mepc** 值，退出 NMI/异常时可从 **msaveepc1** 恢复 **mepc** 的值。
 - **msaveepc2**: 第二级 NMI/异常状态堆栈记录第二级嵌套 NMI/异常（被第一级嵌套 NMI/异常嵌套的 NMI/异常）发生前的 PC，亦即当前 NMI/异常发生前的 **msaveepc1** 的值，退出 NMI/异常时可从 **msaveepc2** 恢复 **msaveepc1** 的值。
 - **mstatus**:
 - ◆ **MPIE**: 记录当前 NMI/异常发生之前的 MIE。
 - ◆ **MPP**: 记录当前 NMI/异常发生之前的 Privilege Mode。
 - **msavestatus**:
 - ◆ **MPIE1**: 第一级 NMI/异常状态堆栈记录第一级嵌套 NMI/异常发生前的 MIE，亦即当前 NMI/异常发生前的 **MPIE**，退出 NMI/异常时可从 **MPIE1** 恢复 **MPIE** 的值。
 - ◆ **MPIE2**: 第二级 NMI/异常状态堆栈记录第二级嵌套 NMI/异常发生前的 MIE，亦即当前 NMI/异常发生前的 **MPIE1**，退出 NMI/异常时可从 **MPIE2** 恢复 **MPIE1** 的值。
 - ◆ **MPP1**: 第一级 NMI/异常状态堆栈记录第一级嵌套 NMI/异常发生前的 Privilege Mode，亦即当前 NMI/异常发生前的 **MPP**，退出 NMI/异常时可从 **MPP1** 恢复 **MPP** 的值。

- ◆ **MPP2:** 第二级 NMI/异常状态堆栈记录第二级嵌套 NMI/异常发生前的 Privilege Mode, 亦即当前 NMI/异常发生前的 MPP1, 退出 NMI/异常时可从 MPP2 恢复 MPP1 的值。
 - **mcause:** 记录发生当前 NMI/异常的原因。
 - **msavecause1:** 第一级 NMI/异常状态堆栈记录第一级嵌套 NMI/异常原因。
 - **msavecause2:** 第二级 NMI/异常状态堆栈记录第二级嵌套 NMI/异常原因。
 - **mdcause:** 记录发生当前异常的详细原因。
 - **msavedcause1:** 第一级 NMI/异常状态堆栈记录第一级嵌套异常的详细原因。
 - **msavedcause2:** 第二级 NMI/异常状态堆栈记录第二级嵌套异常的详细原因。
 - **msubm:**
 - ◆ **TYP:** 记录当前 NMI/异常的 Trap 类型。
 - ◆ **PTYP:** 记录当前 NMI/异常发生前处理器所处 Trap 的类型
 - ◆ **PTYP1:** 第一级 NMI/异常状态堆栈记录第一级嵌套 NMI/异常发生前的 Machine Sub Mode, 亦即当前 NMI/异常发生前的 PTYP, 退出 NMI/异常时可从 PTYP1 恢复 PTYP 的值。
 - ◆ **PTYP2:** 第二级 NMI/异常状态堆栈记录第二级嵌套 NMI/异常发生前的 Machine Sub Mode, 亦即当前 NMI/异常发生前的 PTYP1, 退出 NMI/异常时可从 PTYP2 恢复 PTYP1 的值。
- NMI/异常处理是在机器模式 (Machine Mode) 下完成的, 所以在进入 NMI/异常时, 处理器内核的特权模式 (Privilege Mode) 切换成机器模式。

4.6.2. 退出 NMI/异常嵌套

当程序完成 NMI/异常处理之后, 最终需要从 NMI/异常服务程序退出, 返回上级 NMI/异常或者主程序, 退出之前需要从相关寄存器恢复处理器状态, 这是通过 **mret** 指令完成的, 处理器执

行 **mret** 指令后的硬件行为如图 4-4，可以简述如下。

- 停止执行当前程序流，转而从 CSR 寄存器 **mepc** 定义的 PC 地址开始执行。
- 更新相关 CSR 寄存器，分别是以下几个寄存器及其相关的域：
 - **mepc**: 恢复为存储在 **msaveepc1** 中第一级嵌套 NMI/异常发生前的 PC。
 - **msaveepc1**: 第一级 NMI/异常状态堆栈，**mret** 发生时从第二级 NMI/异常状态堆栈 **msaveepc2** 恢复寄存器 **msaveepc1** 值，即恢复为存储在 **msaveepc2** 中的第二级嵌套 NMI/异常发生前的 PC。
 - **mstatus**
 - ◆ **MPIE**: 恢复为存储在 **MPIE1** 中的第一级嵌套 NMI/异常发生前的 MIE。
 - ◆ **MPP**: 恢复为存储在 **MPP1** 中的第一级嵌套 NMI/异常发生前的 Privilege Mode。
 - **msavestatus**:
 - ◆ **MPIE1**: 第一级 NMI/异常状态堆栈，**mret** 发生时从第二级 NMI/异常状态堆栈 **MPIE2** 恢复寄存器域 **msavestatus.MPIE1** 的值，即恢复为存储在 **MPIE2** 中的第二级嵌套 NMI/异常发生前的 MIE。
 - ◆ **MPP1**: 第一级 NMI/异常状态堆栈，**mret** 发生时从第二级 NMI/异常状态堆栈 **MPP2** 恢复寄存器域 **msavestatus.MPP1** 的值，即恢复为存储在 **MPP2** 中的第二级嵌套 NMI/异常发生前的 Privilege Mode。
 - **mcause**: 恢复为存储在 **msavecause1** 中的第一级嵌套 NMI/异常的原因。
 - **msavecause1**: 第一级 NMI/异常状态堆栈，**mret** 发生时从第二级 NMI/异常状态堆栈 **msavecause2** 恢复寄存器 **msavecause1** 的值，即恢复为存储在 **msavecause2** 中的第二级嵌套 NMI/异常的原因。
 - **mdcause**: 恢复为存储在 **msavedcause1** 中的第一级嵌套异常的详细原因。
 - **msavedcause1**: 第一级 NMI/异常状态堆栈，**mret** 发生时从第二级 NMI/异常状态堆栈 **msavedcause2** 恢复寄存器 **msavedcause1** 的值，即恢复为存储在 **msavedcause2** 中的第二级嵌套异常的详细原因。

- **msubm (Machine Sub-Mode Register)**
 - ◆ **TYP:** 恢复为存储在 **msubm.PTYP** 中的当前 NMI/异常发生前处理器的 **Trap** 类型。
 - ◆ **PTYP:** 恢复为存储在 **msubm.PTYP1** 中第一级嵌套 NMI/异常发生前处理器的 **Trap** 类型。
 - ◆ **PTYP1:** 第一级 NMI/异常状态堆栈，**mret** 发生时从第二级 NMI/异常状态堆栈 **PTYP2** 恢复寄存器域 **msubm.PTYP1** 的值，即恢复为存储在 **msubm.PTYP2** 中的第二级嵌套 NMI/异常发生前处理器的 **Trap** 类型。
- 根据 **mstatus.MPP** 域的值更新处理器的 **Privilege Mode**。

5. N 级别处理器内核中断机制介绍

5.1. 中断概述

中断（Interrupt）机制，即处理器内核在顺序执行程序指令流的过程中突然被别的请求打断而中止执行当前的程序，转而去处理别的事情，待其处理完了别的事情，然后重新回到之前程序中断的点继续执行之前的程序指令流。

中断的若干基本知识要点如下：

- 打断处理器执行的“别的请求”便称之为中断请求（Interrupt Request），“别的请求”的来源便称之为中断源（Interrupt Source），中断源通常来自于内核外部（称之为外部中断源），也可以来自于内核内部（成为内部中断源）。
- 处理器转而去处理的“别的事情”便称之为中断服务程序（Interrupt Service Routine, ISR）。
- 中断处理是一种正常的机制，而非一种错误情形。处理器收到中断请求之后，需要保存当前程序的现场，简称为“保存现场”。等到处理完中断服务程序后，处理器需要恢复之前的现场，从而继续执行之前被打断的程序，简称为“恢复现场”。
- 可能存在多个中断源同时向处理器发起请求的情形，需要对这些中断源进行仲裁，从而选择哪个中断源被优先处理。此种情况称为“中断仲裁”，同时可以给不同的中断分配级别和优先级以便于仲裁，因此中断存在着“中断级别”和“中断优先级”的概念。

5.2. 中断控制器 ECLIC

如第 7.4.17 节中所述，通过软件的不同配置，N 级别处理器内核支持“默认中断模式”和“ECLIC 中断模式”，推荐使用“ECLIC 中断模式”，本文仅对“ECLIC 中断模式”进行介绍。

N 级别处理器内核实现了一个“改进型内核中断控制器（Enhanced Core Local Interrupt Controller, ECLIC）”，可用于多个中断源的管理。N 级别处理器内核中的所有类型（除了调试中断之外）的中断都由 ECLIC 统一进行管理，有关 ECLIC 的详情请参见第 6.2 节。有关 N 级别处理

器内核支持的所有中断类型的介绍请参见第 5.3 节。

5.3. 中断类型

N 级别处理器内核支持的中断类型如图 5-1 中所示。

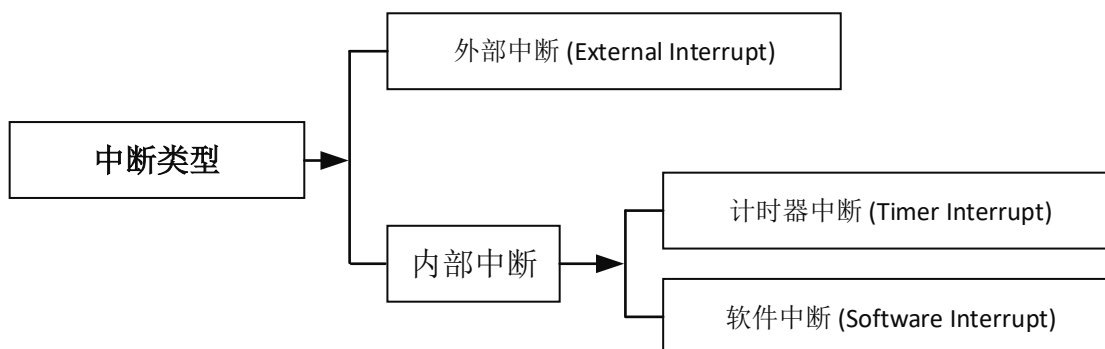


图 5-1 中断类型示意图

下文将分别予以详述。

5.3.1. 外部中断

外部中断是指来自于处理器核外部的中断。外部中断可供用户连接外部中断源，譬如外部设备 UART、GPIO 等产生的中断。

注意：N 级别处理器内核支持多个外部中断源，所有外部中断都由 ECLIC 进行统一管理。

5.3.2. 内部中断

N 级别处理器内核有几种内核私有的内部中断，分别为：

- 软件中断 (Software Interrupt)
- 计时器中断 (Timer Interrupt)

注意：N 级别处理器内核的内部中断也都由 ECLIC 进行统一管理。

5.3.2.1 软件中断

软件中断要点如下：

- N 级别处理器内核实现了一个 **TIMER** 单元，**TIMER** 单元里定义了一个 **msip** 寄存器，通过其可以产生软件中断，请参见第 6.1.6 节了解详情。
- 注意：软件中断也由 **ECLIC** 进行统一管理。

5.3.2.2 计时器中断

计时器中断要点如下：

- N 级别处理器内核实现了一个 **TIMER** 单元，**TIMER** 单元里定义了一个计时器，通过其可以产生计时器中断，请参见第 6.1.5 节了解详情。
- 注意：计时器中断也由 **ECLIC** 进行统一管理。

5.4. 中断屏蔽

5.4.1. 中断全局屏蔽

N 级别处理器内核的中断可以被屏蔽掉，**CSR** 寄存器 **mstatus** 的 **MIE** 域控制中断的全局使能。请参见第 7.4.9 节了解详情。

5.4.2. 中断源单独屏蔽

对于不同的中断源而言，**ECLIC** 为每个中断源分配了各自的中断使能寄存器，用户可以通过配置 **ECLIC** 寄存器来管理各个中断源的屏蔽，请参见第 6.2.6 节了解详情。

5.5. 中断级别、优先级与仲裁

当多个中断同时出现时，需要进行仲裁。对于 N 级别处理器内核处理器而言，ECLIC 统一管理所有的中断。ECLIC 为每个中断源分配了各自的中断级别和优先级寄存器，用户可以通过配置 ECLIC 寄存器来管理各个中断源的级别和优先级，当多个中断同时发生时，ECLIC 会仲裁出级别和优先级最高的中断，如图 5-2 中所示。请参见第 6.2.9 节了解其详情。

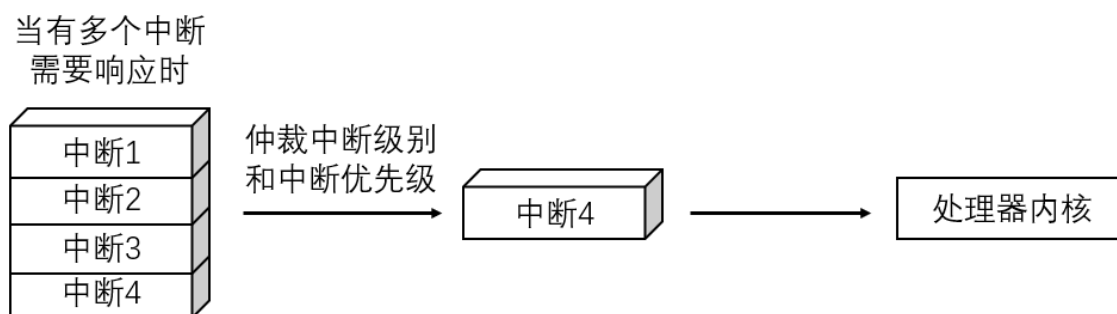


图 5-2 中断仲裁示意图

5.6. 进入中断处理模式

响应中断时，N 级别处理器内核的硬件行为可以简述如下。注意，下列硬件行为在一个时钟周期内同时完成：

- 停止执行当前程序流，转而从新的 PC 地址开始执行。
- 进入中断不仅会让处理器跳转到上述的 PC 地址开始执行，还会让硬件同时更新其他几个 CSR 寄存器，如图 5-4 所示，分别是以下几个寄存器：
 - mepc (Machine Exception Program Counter)
 - mstatus (Machine Status Register)
 - mcause (Machine Cause Register)
 - mintstatus (Machine Interrupt Status Register)

- 除此之外，进入中断还会更新处理器内核的 Privilege Mode 以及 Machine Sub-Mode。
- 总体过程如图 5-3 中所示。

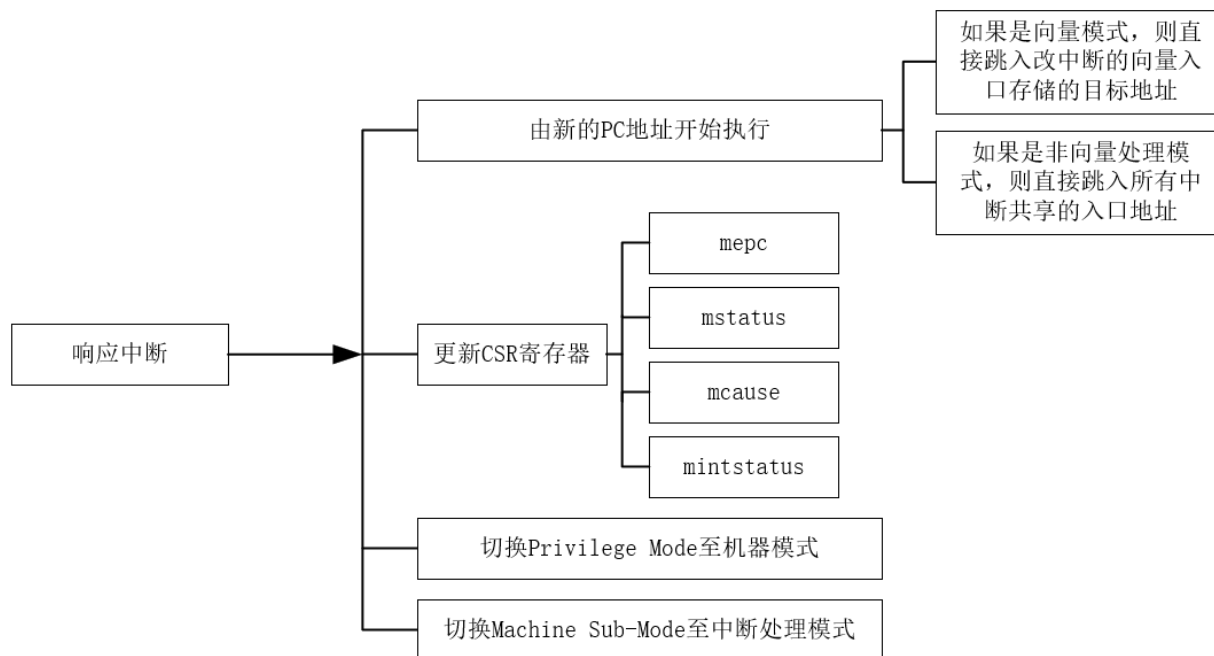


图 5-3 响应中断总体过程

下文将分别予以详述。

5.6.1. 从新的 PC 地址开始执行

ECLIC 的每个中断源均可以设置成向量或者非向量处理(通过寄存器 clicintattr[i]的 shv 域)，其要点如下：

- 如果被配置成为向量处理模式，则该中断被处理器内核响应后，处理器直接跳入该中断的向量入口（Vector Table Entry）存储的目标地址。有关中断向量表的详细介绍，请参见第 5.8 节，有关向量处理模式的详细介绍，请参见第 5.13.2 节。

- 如果被配置成为非向量处理模式，则该中断被处理器内核响应后，处理器直接跳入所有中断共享的入口地址。有关中断非向量处理模式的详细介绍，请参见第 5.13.1 节。

5.6.2. 更新 Privilege Mode

在进入中断时，处理器内核的 **Privilege Mode** 被更新为 **Machine Mode**。

5.6.3. 更新 Machine Sub-Mode

N 级别处理器内核的 **Machine Sub-Mode** 实时反映在 CSR 寄存器 **msubm.TYP** 域中。在进入中断时，处理器内核的 **Machine Sub-Mode** 被更新为中断处理模式，因此：

- CSR 寄存器 **msubm.PTYP** 域的值被更新为中断发生前的 **Machine Sub-Mode** (**msubm.TYP** 域的值)，如图 5-4 所示。**msubm.PTYP** 域的作用是在中断结束之后，能够使用 **msubm.PTYP** 的值恢复出中断发生之前的 **Machine Sub-Mode** 值。
- CSR 寄存器 **msubm.TYP** 域的值则被更新为“中断处理模式”，如图 5-4 所示，以实时反映当前的模式已经是“中断处理模式”。

5.6.4. 更新 CSR 寄存器 mepc

N 级别处理器内核退出中断时的返回地址由 CSR 寄存器 **mepc** 指定。在进入中断时，硬件将自动更新 **mepc** 寄存器的值，该寄存器将作为退出中断的返回地址，在中断结束之后，能够使用它保存的 PC 值回到之前被停止执行的程序点。

注意：

- 出现中断时，中断返回地址 **mepc** 被指向一条指令，此指令因为中断的出现而未能完成执行。那么在退出中断后，程序便会回到之前的程序点，从 **mepc** 所存储的未执行完的指令开始重新执行。
- 虽然 **mepc** 寄存器会在中断发生时自动被硬件更新，但是 **mepc** 寄存器本身也是一个可读可写的寄存器，因此软件也可以直接写该寄存器以修改其值。

5.6.5. 更新 CSR 寄存器 mcause 和 mstatus

mcause 寄存器的详细格式如表 7-6 所示。N 级别处理器内核在进入中断时，CSR 寄存器 mcause 被同时（硬件自动）更新，如图 5-4 所示，详情如下：

- 当前的中断被响应后，需要有一种机制能够记录当前这个中断源的 ID 编号。
 - N 级别处理器内核在进入中断时，CSR 寄存器 mcause.EXCCODE 域被更新以反映当前响应的 ECLIC 中断源的 ID 编号，因此软件可以通过读此寄存器查询中断源的具体 ID。
- 当前的中断被响应，有可能是打断了之前正在处理的中断（中断级别相对低，因此可以被打断），需要有一种机制能够记录被打断中断的中断级别（Interrupt Levels）。
 - N 级别处理器内核在进入中断时，CSR 寄存器 mcause.MPIL 域被更新以反映被打断的中断级别（mintstatus.MIL 域的值）。mcause.MPIL 域的作用是在中断结束之后，能够使用 mcause.MPIL 的值恢复出中断发生之前的 mintstatus.MIL 值。
- 当前的中断被响应后，需要有一种机制能够记录响应中断之前的中断全局使能状态和特权模式。
 - N 级别处理器内核在进入中断时，CSR 寄存器 mstatus.MPIE 域的值被更新为中断发生前中断的全局使能状态（mstatus.MIE 域的值）。mstatus.MIE 域的值则被更新成为 0（意味着进入中断服务程序后中断被全局关闭，所有的中断都将被屏蔽不响应）。
 - N 级别处理器内核在进入中断时，处理器的当前特权模式（Privilege Mode）切换到机器模式（Machine Mode），而 CSR 寄存器 mstatus.MPP 域的值被更新为中断发生前特权模式（Privilege Mode）。
- 当前响应的中断如果是向量处理模式，则处理器响应中断后会直接跳入该中断的向量入口（Vector Table Entry）存储的目标地址。有关中断向量处理模式的详细介绍，请参见第 5.13.2 节。从硬件实现上来说，处理器需要分“两步走”，第一步从中断向量表中取出存储的目标地址，然后第二步再跳转到目标地址中去。那么，在第一步“从中断向量表中取出存储的目标地址”这个存储器访问操作的过程中有可能会发生存储器访问错误，需要有一种机制能够记录这种特殊的存储器访问错误。
 - N 级别处理器内核在进入中断时，如果该中断是向量处理模式，CSR 寄存器

`mcause.minhiv` 域的值被更新为 1，直到上述“两步走”操作彻底成功完成后 `mcause.minhiv` 域的值清除为 0。假设中途发生了存储器访问错误，则最终处理器会发生指令访问错误（Instruction access fault），且 `mcause.minhiv` 域的值为 1（没有被清除）。

- 注意：`mstatus.MPIE` 域和 `mstatus.MPP` 域的值与 `mcause.MPIE` 域和 `mcause.MPP` 域的值是镜像关系，即，在正常情况下，`mstatus.MPIE` 域的值与 `mcause.MPIE` 域的值总是完全一样，`mstatus.MPP` 域的值与 `mcause.MPP` 域的值总是完全一样。

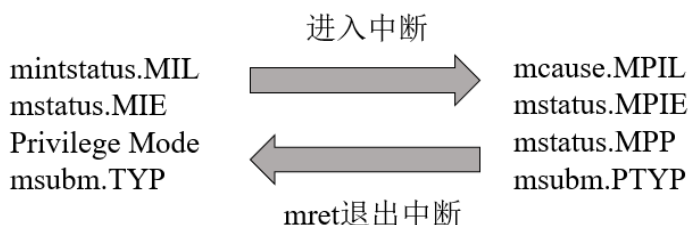


图 5-4 进入/退出中断时 CSR 寄存器的变化

5.7. 退出中断处理模式

当程序完成中断处理之后，最终需要从中断服务程序中退出，并返回主程序。由于中断处理处于 Machine Mode 下，所以退出中断时，软件必须使用 `mret` 指令。处理器执行 `mret` 指令后的硬件行为如下。注意，下列硬件行为在一个时钟周期内同时完成：

- 停止执行当前程序流，转而从 CSR 寄存器 `mepc` 定义的 PC 地址开始执行。
- 执行 `mret` 指令不仅会让处理器跳转到上述的 PC 地址开始执行，还会让硬件同时更新其他几个 CSR 寄存器，如图 5-4 所示，分别是以下几个寄存器：
 - `mstatus`（Machine Status Register）
 - `mcause`（Machine Cause Register）
 - `mintstatus`（Machine Interrupt Status Register）
- 除此之外，进入中断还会更新处理器内核的 Privilege Mode 以及 Machine Sub-Mode。

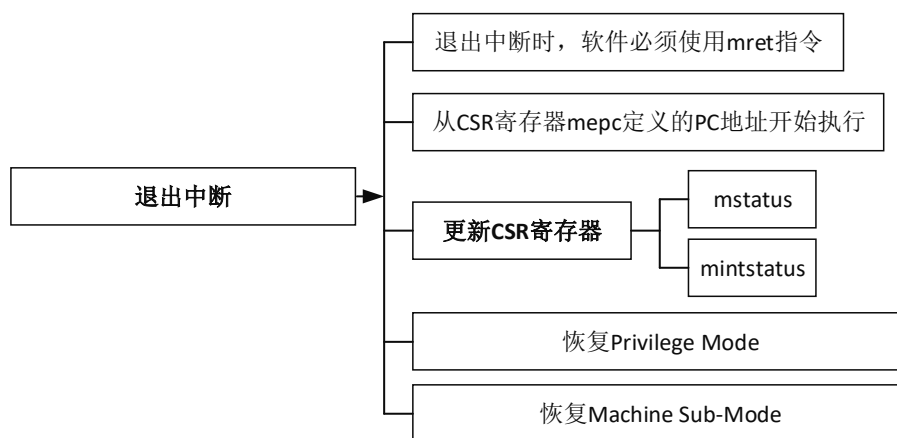


图 5-5 退出中断总体过程

下文将分别予以详述。

5.7.1. 从 mepc 定义的 PC 地址开始执行

在进入中断时，mepc 寄存器被同时更新，以反映当时遇到中断时的 PC 值。软件必须使用 mret 指令退出中断，执行 mret 指令后处理器将从 mepc 定义的 pc 地址重新开始执行。通过这个机制，意味着 mret 指令执行后处理器回到了当时遇到中断时的 PC 地址，从而可以继续执行之前被中止的程序流。

5.7.2. 更新 CSR 寄存器 mcause 和 mstatus

mcause 寄存器的详细格式如表 7-6 所示，执行 mret 指令后，硬件将自动更新 CSR 寄存器 mcause 的某些域：

- 在进入中断时，mcause.MPIL 的值曾经被更新为中断发生前的 mintstatus.MIL 值。而使用 mret 指令退出中断后，硬件将 mintstatus.MIL 的值恢复为 mcause.MPIL 的值。通过这个机制，则意味着退出中断后，处理器的 mintstatus.MIL 值被恢复成中断发生之前的值。
- 在进入中断时，mcause.MPIE 的值曾经被更新为中断发生前的 mstatus.MIE 值。而使用 mret 指令退出中断后，硬件将 mret 指令执行后，将 mstatus.MIE 的值恢复为 mcause.MPIE

的值，如图 5-4 所示。通过这个机制，则意味着退出中断后，处理器的 `mstatus.MIE` 值被恢复成中断发生之前的值。

- 在进入中断时，`mcause.MPP` 的值曾经被更新为中断发生前的特权模式（**Privilege Mode**）。而使用 `mret` 指令退出中断后，硬件将处理器特权模式（**Privilege Mode**）恢复为 `mcause.MPP` 的值，如图 5-4 所示。通过这个机制，则意味着退出中断后，处理器的特权模式（**Privilege Mode**）被恢复成中断发生之前的模式。
- 注意：`mstatus.MPIE` 域和 `mstatus.MPP` 域的值与 `mcause.MPIE` 域和 `mcause.MPP` 域的值是镜像关系，即，在正常情况下，`mstatus.MPIE` 域的值与 `mcause.MPIE` 域的值总是完全一样，`mstatus.MPP` 域的值与 `mcause.MPP` 域的值总是完全一样。

5.7.3. 更新 Privilege Mode

在执行 `mret` 指令后，硬件将自动更新处理器的 **Privilege Mode** 为 `mcause.MPP` 域的值：

- 在进入中断时，`mcause.MPP` 的值曾经被更新为中断发生前的特权模式（**Privilege Mode**）。而使用 `mret` 指令退出中断后，硬件将处理器特权模式（**Privilege Mode**）恢复为 `mcause.MPP` 的值。通过这个机制，则意味着退出中断后，处理器的特权模式（**Privilege Mode**）被恢复成中断发生之前的模式。

5.7.4. 更新 Machine Sub-Mode

N 级别处理器内核的 **Machine Sub-Mode** 实时反映在 CSR 寄存器 `msubm.TYP` 域中。在执行 `mret` 指令后，硬件将自动恢复处理器的 **Machine Sub-Mode** 为 `msubm.PTYP` 域的值：

- 在进入中断时，`msubm.PTYP` 域的值曾经被更新为中断发生前的 **Machine Sub-Mode** 值。而使用 `mret` 指令退出中断后，硬件将处理器 **Machine Sub-Mode** 的值恢复为 `msubm.PTYP` 域的值，如图 5-4 所示。通过这个机制，则意味着退出中断后，处理器的 **Machine Sub-Mode** 被恢复成中断发生之前的 **Machine Sub-Mode**。

5.8. 中断向量表

如图 5-6 中所示，中断向量表是指在存储器里面开辟的一段连续的地址空间，该地址空间的每个字(Word)用于存储 ECLIC 每个中断源对应的中断服务程序(Interrupt Service Routine, ISR)函数的 PC 地址。

中断向量表的起始地址由 CSR 寄存器 `mtvt` 指定，通常可以将 `mtvt` 寄存器设置为整个代码段的起始位置。

中断向量表的作用非常重要，当处理器响应某个中断源后，无论中断是向量处理模式还是非向量处理模式，硬件最终都将通过查询中断向量表中存储的 PC 地址跳转到其对应的中断服务程序函数中去，请参见第 5.13 节了解更多详细介绍。

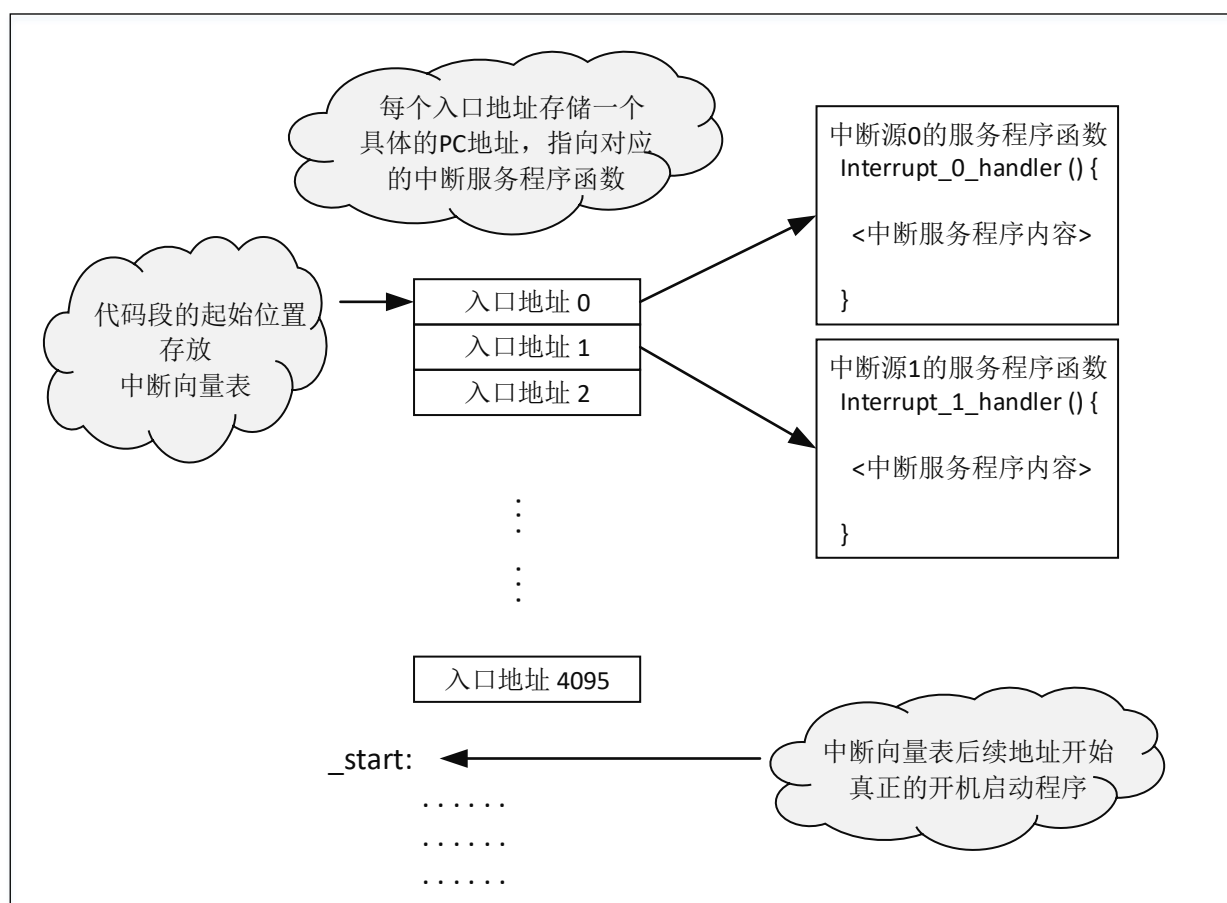


图 5-6 中断向量表示意图

5.9. 进出中断的上下文保存和恢复

RISC-V 架构的处理器在进入和退出中断处理模式时没有硬件自动保存和恢复上下文（通用寄存器）的操作，因此需要软件明确地使用（汇编语言编写的）指令进行上下文的保存和恢复。根据中断是向量处理模式还是非向量处理模式，上下文的保存和恢复涉及到的内容会有所差异，请参见第 5.13 节了解更多详细介绍。

5.10. 中断响应延迟

中断响应延迟的概念通常是指，从“外部中断源拉高”到“处理器真正开始执行该中断源对应的中断服务程序（Interrupt Service Routine, ISR）中的第一条指令”所消耗的指令周期数。因此，中断响应延迟通常会包含如下几个方面的周期开销：

- 处理器内核响应中断后进行跳转的开销
- 处理器内核保存上下文所花费的周期开销
- 处理器内核跳转到中断服务程序（Interrupt Service Routine, ISR）中去的开销。

取决于中断是向量处理模式还是非向量处理模式，中断响应延迟会有所差异，请参见第 5.13 节了解更多详细介绍。

5.11. 中断嵌套

处理器内核正在处理某个中断的过程中，可能有一个级别更高的新中断请求到来，处理器可以中止当前的中断服务程序，转而开始响应新的中断，并执行其“中断服务程序”，如此便形成了中断嵌套（即前一个中断还没响应完，又开始响应新的中断），并且嵌套的层次可以有好多层。

以图 5-7 中的示例为例：

- 假设处理器正在处理计时器中断，突然有另外一个按键 1 中断到来（级别比计时器中断高），那么处理器会暂停处理计时器中断，开始处理按键 1 中断。
- 但是突然又有另外一个按键 2 中断到来（级别比按键 1 中断更高），那么处理器会暂停处理按键 1 的中断，开始处理按键 2 中断。

- 之后再没有其他更高级别的中断到来，则按键 2 中断不会再被打断，处理器能够顺利处理完毕按键 2 的中断，然后重新回到按键 1 中断的处理程序中去，完成按键 1 中断的处理。
- 完成按键 1 中断的处理之后，处理器会重新回到计时器中断的处理程序中去，完成计时器中断的处理。

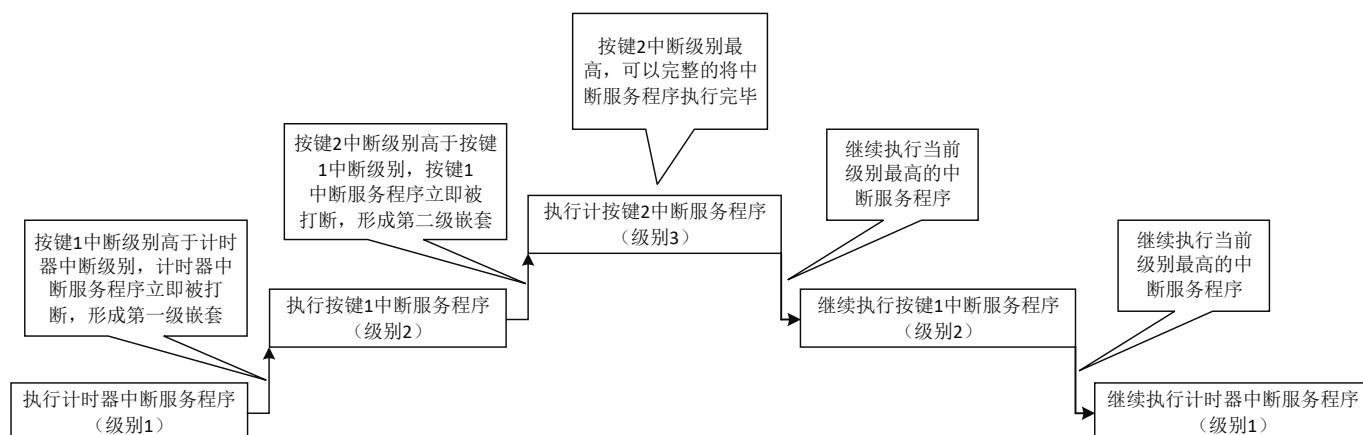


图 5-7 中断嵌套示意图

注意：假设新来的中断请求的优先级比正在处理的中断级别低（或者相同），则处理器不应该响应这个新的中断请求，处理器必须完成当前的中断服务程序之后才考虑响应新的中断请求（因为新中断请求的级别并不比当前正在处理的中断级别高）。有关中断级别的设定请参见第 6.2.9 节了解更多信息。

在 N 级别处理器内核中，取决于中断是向量处理模式还是非向量处理模式，中断嵌套的支持方法会有所差异，请参见第 5.13 节了解更多详细介绍。

5.12. 中断咬尾

处理器内核正在处理某个中断的过程中，可能有新中断请求到来，但是“新中断的级别”低于或者等于“当前正在处理的中断级别”，因此，新中断不能够打断当前正在处理的中断（因此不会形成嵌套）。

当处理器完成当前中断之后，理论上需要恢复上下文，然后退出中断回到主应用程序，然后重新响应新的中断，响应新的中断又需要再次保存上下文。因此，存在着一次背靠背的“恢复上下文”

和“保存上下文”操作，如果将此背靠背的“恢复上下文”和“保存上下文”省略掉，则称之为“中断咬尾”，如图 5-8 中所示，显而易见，中断咬尾可以加快多个中断的背靠背处理速度。

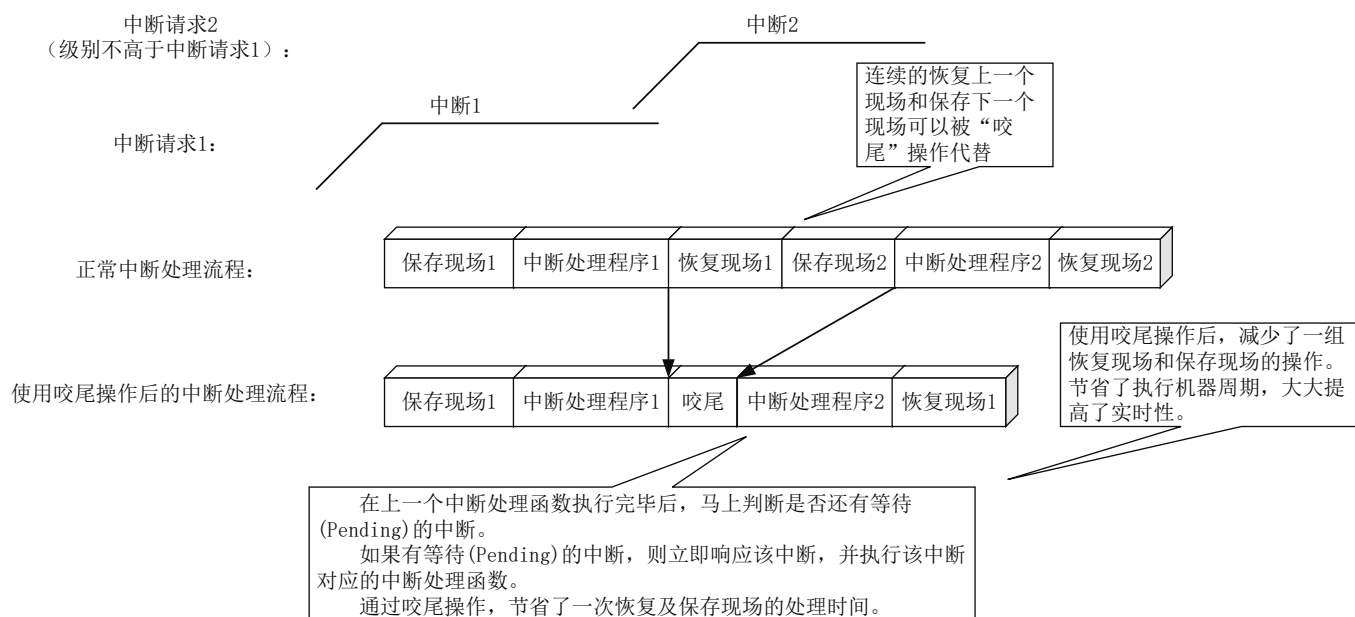


图 5-8 中断咬尾示意图

在 N 级别处理器内核中，只有非向量处理模式才支持中断咬尾，请参见第 5.13.1.3 节了解更多详细介绍。

5.13. 中断的向量处理模式和非向量处理模式

如第 6.2.10 节中所述，ECLIC 的每个中断源均可以设置成向量或者非向量处理（通过寄存器 clicintattr[i] 的 shv 域），向量处理模式和非向量处理模式二者有较大的差别，分别介绍如下。

5.13.1. 非向量处理模式

5.13.1.1 非向量处理模式的特点和延迟

如果被配置成为非向量处理模式，则该中断被处理器内核响应后，处理器会直接跳入到所有非向量中断共享的入口地址，该入口地址可以通过软件进行设置：

- 如果配置 CSR 寄存器 `mtvt2` 的最低位为 0（上电复位默认值），则所有非向量中断共享的入口地址由 CSR 寄存器 `mtvec` 的值（忽略最低 2 位的值）指定。由于 `mtvec` 寄存器的值也指定异常的入口地址，因此，意味着在这种情况下，异常和所有非向量中断共享入口地址。
- 如果配置 CSR 寄存器 `mtvt2` 的最低位为 1，则所有非向量中断共享的入口地址由 CSR 寄存器 `mtvt2` 的值（忽略最低 2 位的值）指定。为了让中断以尽可能快的速度被响应和处理，推荐将 CSR 寄存器 `mtvt2` 的最低位设置为 1，即，由 `mtvt2` 指定一个独立的入口地址供所有非向量中断专用，和异常的入口地址（由 `mtvec` 的值指定）彻底分开。

在进入所有非向量中断共享的入口地址之后，处理器会开始执行一段共有的软件代码，如图 5-9 中所示的例子，这段软件代码内容通常如下：

- 首先保存 CSR 寄存器 `mepc`、`mcause`、`msubm` 入堆栈。保存这几个 CSR 寄存器是为了保证后续的中断嵌套能够功能正确，因为新的中断响应会重新覆盖 `mepc`、`mcause`、`msubm` 的值，因此需要将它们先保存入堆栈。
- 保存若干通用寄存器（处理器的上下文）入堆栈。
- 然后执行一条特殊的指令 “`csrrw ra, CSR_JALMNXTI, ra`”。如果没有中断在等待（Pending），则该指令相当于是个 Nop 指令不做任何操作；如果有中断在等待（Pending），执行该指令后处理器会：
 - 直接跳入该中断的向量入口（Vector Table Entry）存储的目标地址，即该中断源的中断服务程序（Interrupt Service Routine, ISR）中去。
 - 在跳入中断服务程序的同时，硬件也会同时打开中断的全局使能，即，设置 `mstatus` 寄存器的 MIE 域为 1。打开中断全局使能后，新的中断便可以响应，从而达到中断嵌套的效果。
 - 在跳入中断服务程序的同时，“`csrrw ra, CSR_JALMNXTI, ra`” 指令还会达到 JAL（Jump and Link）的效果，硬件同时更新 Link 寄存器的值为该指令的 PC 自身作为函数调用的返回地址。因此，从中断服务程序函数返回后会回到该 “`csrrw ra, CSR_JALMNXTI, ra`” 指令重新执行，重新判断是否还有中断在等待（Pending），从而达到中断咬尾的效果。

- 在中断服务程序的结尾处同样需要添加对应的恢复上下文出栈操作。并且在 CSR 寄存器 mepc、mcause、msubm 出堆栈之前，需要将中断全局使能再次关闭，以保证 mepc、mcause、msubm 恢复操作的原子性（不被新的中断所打断）。

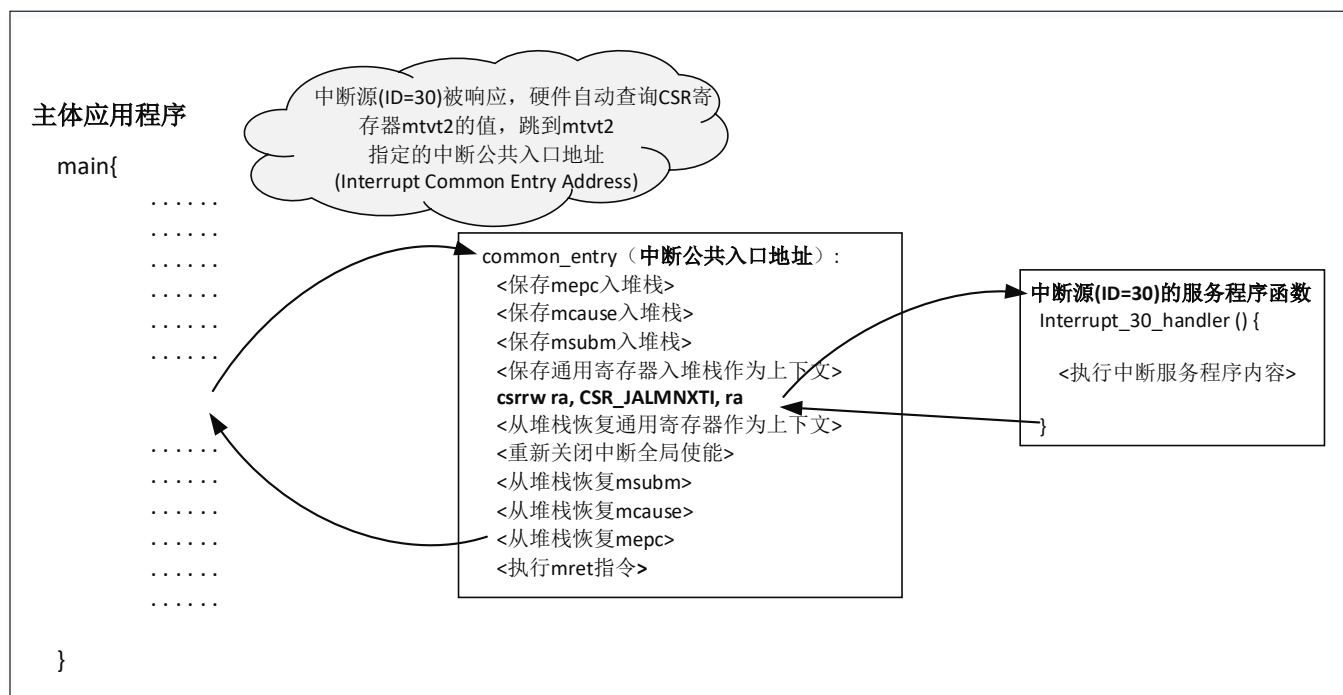


图 5-9 中断的非向量处理模式示例（总是支持嵌套）

由于非向量处理模式时处理器在跳到中断服务程序之前需要先执行一段共有的软件代码进行上下文的保存，因此，从中断源拉高到处理器开始执行中断服务程序中的第一条指令，需要经历以下几个方面的时钟周期开销：

- 处理器内核响应中断后进行跳转的开销。理想情况下约 4 个时钟周期。
- 处理器内核保存 CSR 寄存器 mepc、mcause、msubm 入堆栈的开销。
- 处理器内核保存上下文所花费的周期开销。如果是 RV32E 的架构，则需要保存 8 个通用寄存器，如果是 RV32I 的架构，则需要保存 16 个通用寄存器。
- 处理器内核跳转到中断服务程序（Interrupt Service Routine, ISR）中去的开销。理想情况下约需要 5 个时钟周期。

5.13.1.2 非向量处理模式的中断嵌套

如上文所述，非向量处理模式总是能够支持中断嵌套，如图 5-10 中所示的示例：假设中断源 30、31、32 这三个中断源先后到来，且“中断源 32 的级别” > “中断源 31 的级别” > “中断源 30 的级别”，那么后来的中断便会打断之前正在处理的中断形成中断嵌套。

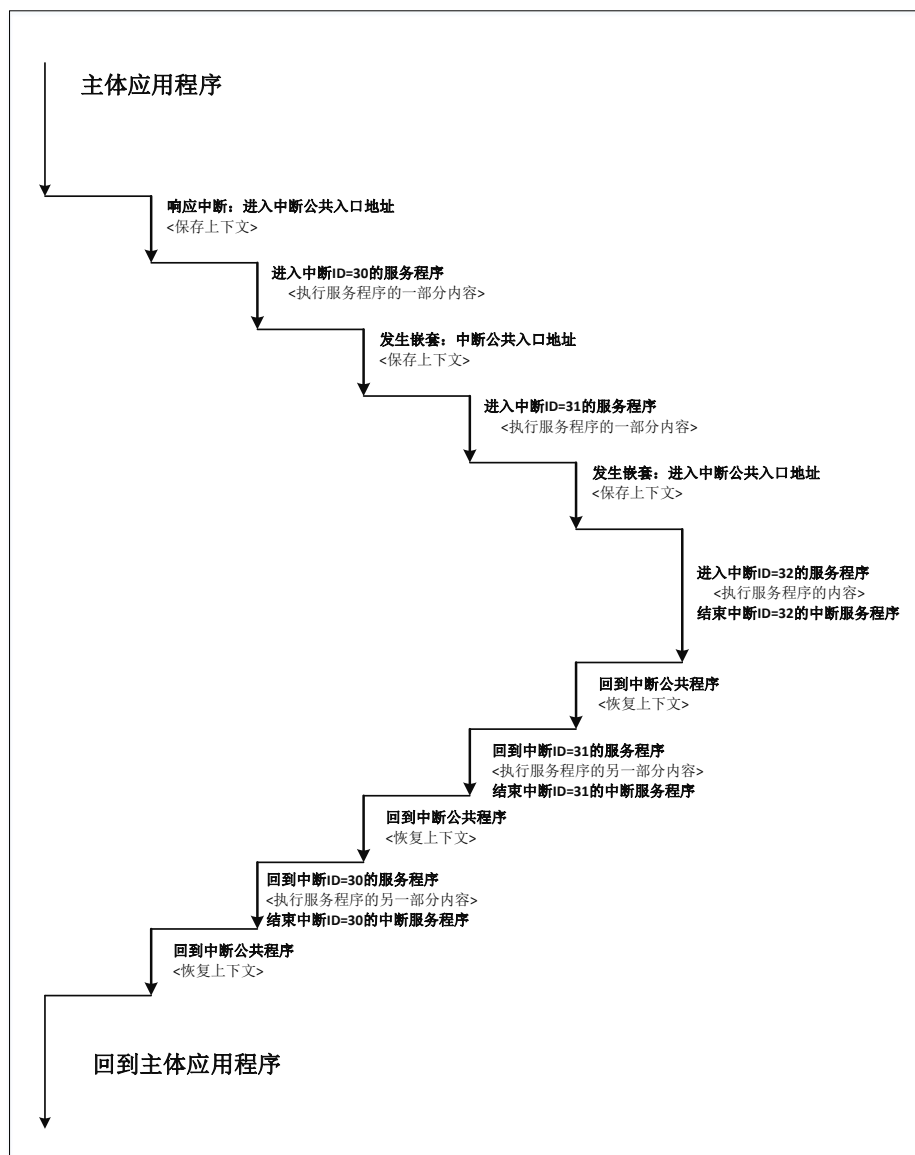


图 5-10 三个先后到来的（非向量处理模式）中断形成嵌套

5.13.1.3 非向量处理模式的中断咬尾

对于非向量处理模式的中断而言，由于在跳入和退出中断服务程序之前，处理器要进行上下文的保存和恢复，因此进行“中断咬尾”能够节省显著的时间（节省一次背靠背的保存上下文和恢复上下文）。

如上文所述，在所有非向量中断共享的共有代码段中，在跳入中断服务程序的同时，“csrrw ra, CSR_JALMNXTI, ra”指令还会达到 JAL（Jump and Link）的效果，硬件同时更新 Link 寄存器的值为该指令的 PC 自身作为函数调用的返回地址。因此，从中断服务程序函数返回后会回到该“csrrw ra, CSR_JALMNXTI, ra”指令重新执行，重新判断是否还有中断在等待（Pending），从而达到中断咬尾的效果。

如图 5-11 中所示的示例：假设中断源 30、29、28 这三个中断源先后到来，且“中断源 30 的级别” \geq “中断源 29 的级别” \geq “中断源 28 的级别”，那么后来的中断不会打断之前正在处理的中断（不会形成中断嵌套），但是会被置于等待（Pending）状态。当中断源 30 完成处理后，将会直接开始中断源 29 的中断处理，省掉中间的“恢复上下文”和“保存上下文”过程。

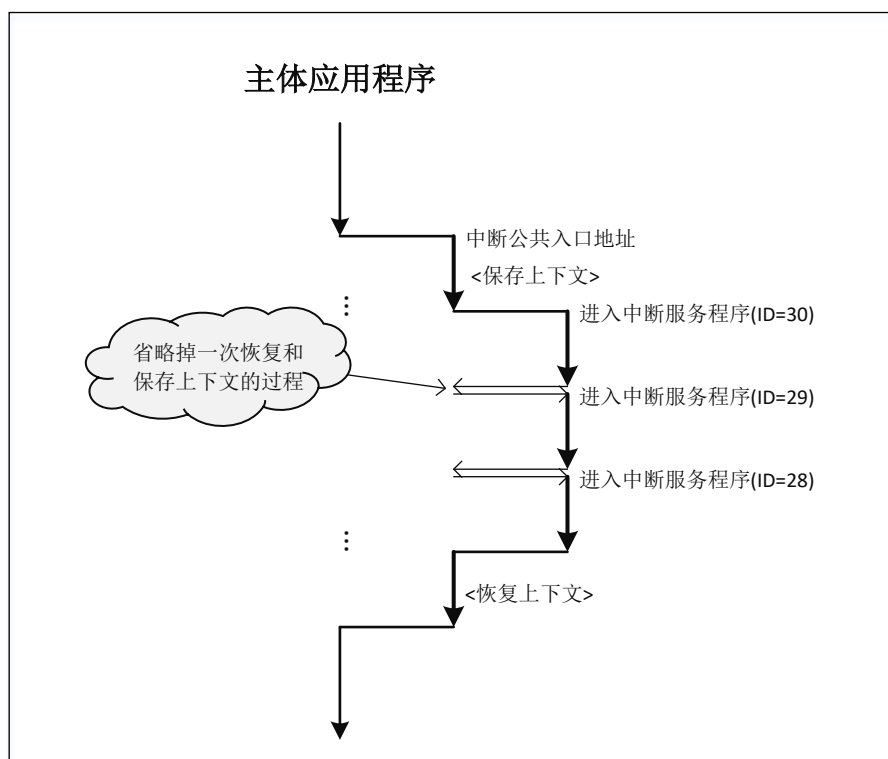


图 5-11 中断咬尾示意图

5.13.2. 向量处理模式

5.13.2.1 向量处理模式的特点和延迟

如果被配置成为向量处理模式，则该中断被处理器内核响应后，处理器会直接跳入该中断的向量入口（Vector Table Entry）存储的目标地址，即该中断源的中断服务程序（Interrupt Service Routine, ISR），如图 5-12 中所示的例子。

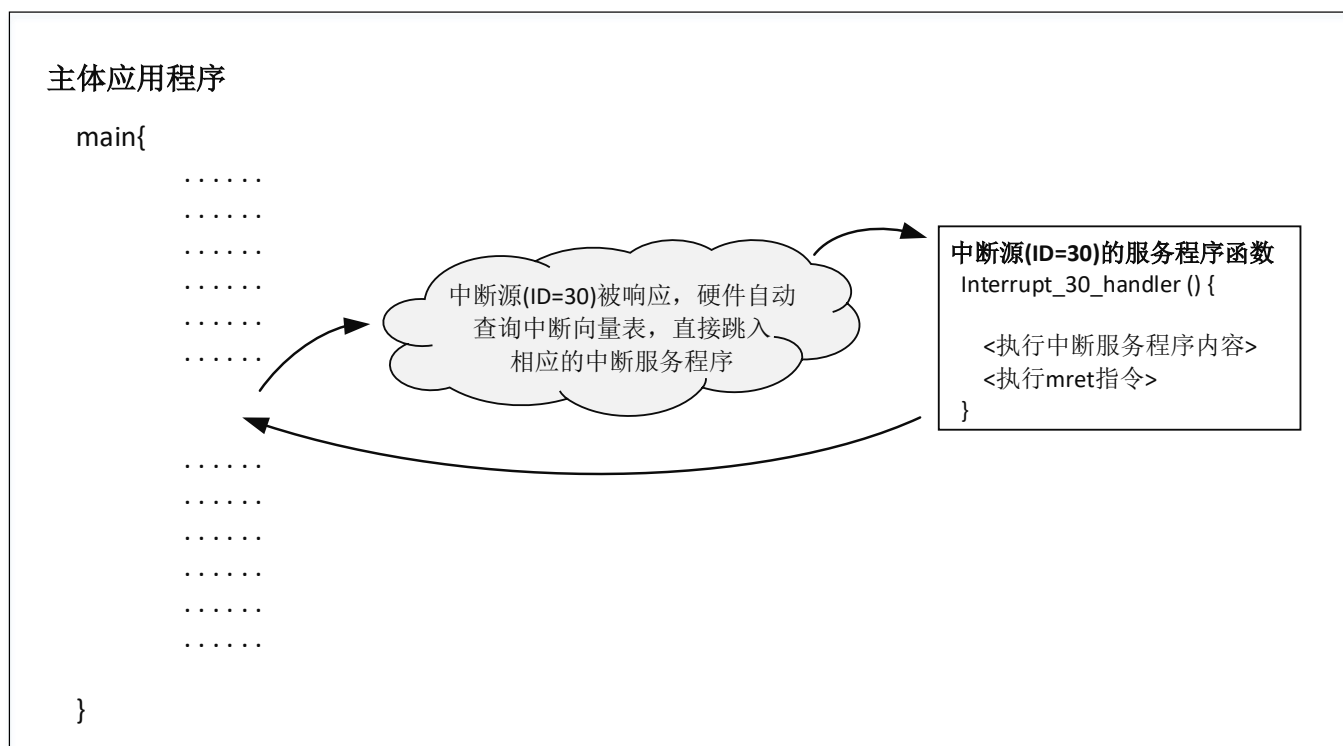


图 5-12 中断的向量处理模式示例

向量处理模式具有如下特点：

- 向量处理模式时处理器会直接跳到中断服务程序，并没有进行上下文的保存，因此，中断响应延迟非常之短，从中断源拉高到处理器开始执行中断服务程序中的第一条指令，基本上只需要硬件进行查表和跳转的时间开销，理想情况下约 6 个时钟周期。
- 对于向量处理模式的中断服务程序函数，一定要使用特殊的 `__attribute__((interrupt))` 来修饰中断服务程序函数。

- 向量处理模式时，由于在跳入中断服务程序之前，处理器并没有进行上下文的保存，因此，理论上中断服务程序函数本身不能够进行子函数的调用（即，必须是 **Leaf Function**）。
- 如果中断服务程序函数不小心调用了其他的子函数（不是 **Leaf Function**），如果不加处理则会造成功能的错误。为了规避这种不小心造成的错误情形，只要使用了特殊的 `__attribute__((interrupt))` 来修饰该中断服务程序函数，那么编译器会自动的进行判断，当编译器发现该函数调用了其他子函数时，便会自动的插入一段代码进行上下文的保存。注意：这种情况下虽然保证了功能的正确性，但是由于保存上下文造成的开销，又会事实上还是增大中断的响应延迟（与非向量模式相当）并且造成代码尺寸（**Code Size**）的膨胀。因此，在实践中，如果使用向量处理模式，那么不推荐在向量处理模式的中断服务程序函数中调用其他的子函数。
- 向量处理模式时，由于在跳入中断服务程序之前，处理器并没有进行任何特殊的处理，且由于处理器内核在响应中断后，`mstatus` 寄存器中的 **MIE** 域将会被硬件自动更新成为 **0**（意味着中断被全局关闭，从而无法响应新的中断）。因此向量处理模式默认是不支持中断嵌套的，为了达到向量处理模式且又能够中断嵌套的效果，如图 5-13 中所示，需要在中断服务程序的开头处添加特殊的入栈操作：
 - 首先保存 CSR 寄存器 `mepc`、`mcause`、`msubm` 入堆栈。保存这几个 CSR 寄存器是为了保证后续的中断嵌套能够功能正确，因为新的中断响应会重新覆盖 `mepc`、`mcause`、`msubm` 的值，因此需要将它们先保存入堆栈。
 - 重新打开中断的全局使能，即，设置 `mstatus` 寄存器的 **MIE** 域为 **1**。打开中断全局使能后，新的中断便可以响应，从而达到中断嵌套的效果。
 - 在中断服务程序的结尾处同样需要添加对应的恢复上下文出栈操作。并且在 CSR 寄存器 `mepc`、`mcause`、`msubm` 出堆栈之前，需要将中断全局使能再次关闭，以保证 `mepc`、`mcause`、`msubm` 恢复操作的原子性（不被新的中断所打断）。

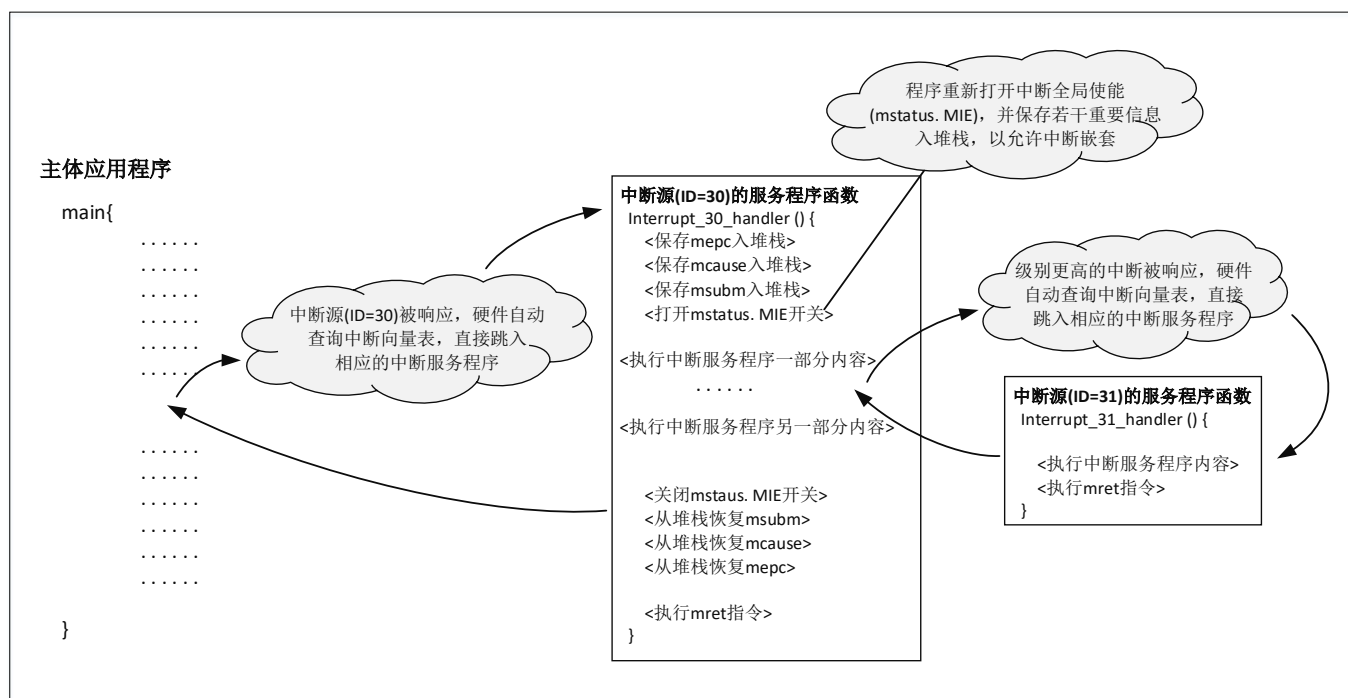


图 5-13 中断的向量处理模式示例（支持中断嵌套）

5.13.2.2 向量处理模式的中断嵌套

如上文所述，向量处理模式的中断经过特殊处理也可以支持中断嵌套，如图 5-14 中所示的示例：假设中断源 30、31、32 这三个中断源先后到来，且“中断源 32 的级别” > “中断源 31 的级别” > “中断源 30 的级别”，那么后来的中断便会打断之前正在处理的中断形成中断嵌套。

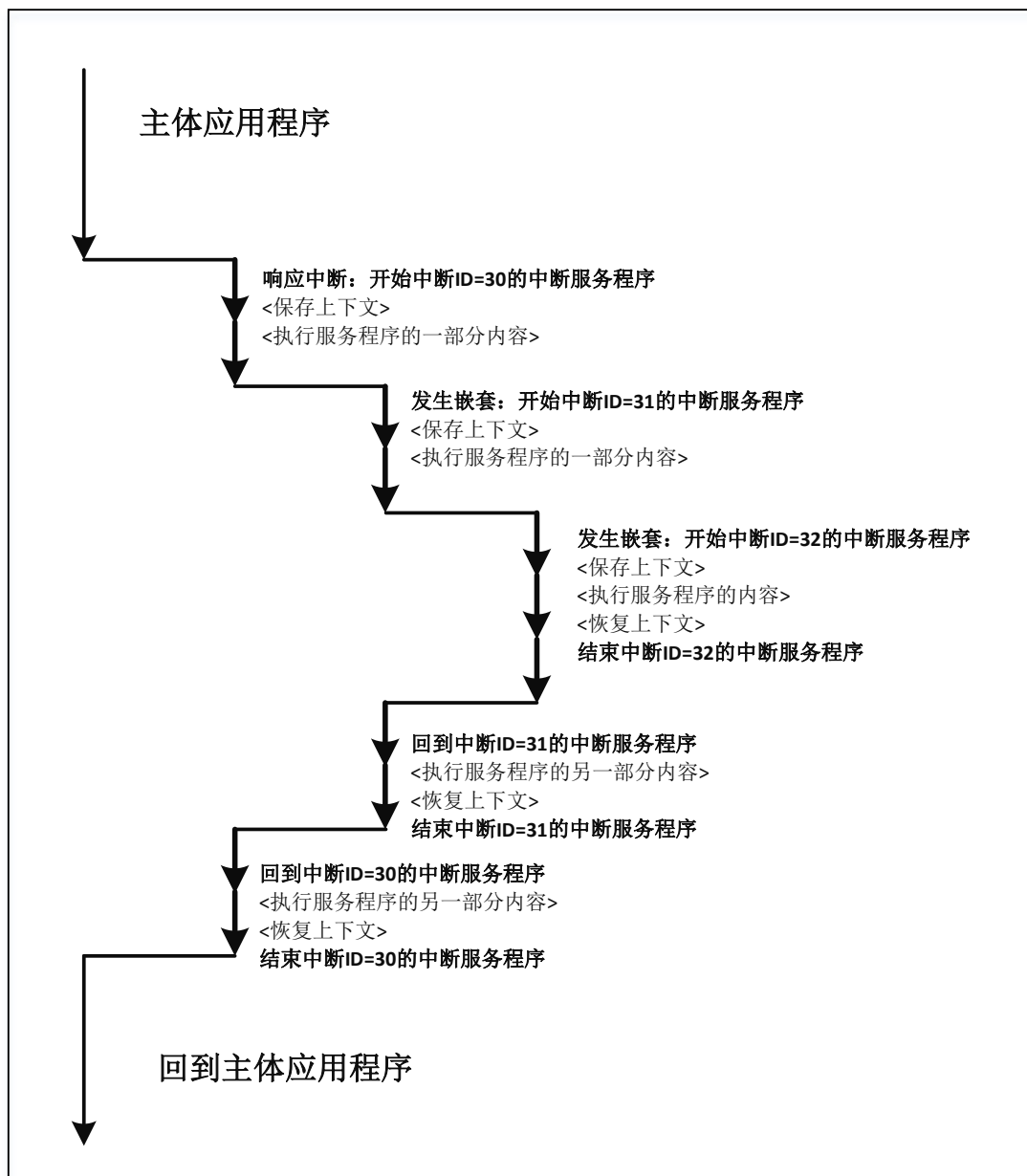


图 5-14 三个先后到来的（向量处理模式）中断形成嵌套

5.13.2.3 向量处理模式的中断咬尾

对于向量处理模式的中断而言，由于在跳入中断服务程序之前，处理器并没有进行上下文的保存，因此进行“中断咬尾”的意义不大，因此，向量处理模式的中断，没有“中断咬尾”处理能力。

6. N 级别处理器内核 TIMER 和 ECLIC 介绍

6.1. TIMER 介绍

6.1.1. TIMER 简介

计时器单元 (Timer Unit, TIMER)，在 N 级别处理器内核中主要用于产生计时器中断 (Timer Interrupt) 和软件中断 (Software Interrupt)。请参见第 5.3.2.1 节和第 5.3.2.2 节了解计时器中断与软件中断的详细信息。

6.1.2. TIMER 寄存器

TIMER 是一个存储器地址映射的单元：

- TIMER 单元在 N 级别处理器内核中的基地址请参见《Nuclei_N 级别处理器内核简明数据手册》中的介绍。
- TIMER 单元内寄存器和地址偏移量如表 6-1 中所示。

表 6-1 TIMER 寄存器的存储器映射地址

模块内偏移地址	读写属性	寄存器名称	复位默认值	功能描述
0x0	可读可写	mtime_lo	0x00000000	反映计时器 mtime 的低 32 位值, 参见第 6.1.3 节了解其详细介绍。
0x4	可读可写	mtime_hi	0x00000000	反映计时器 mtime 的高 32 位值, 参见第 6.1.3 节了解其详细介绍。
0x8	可读可写	mtimecmp_lo	0xFFFFFFFF	配置计时器的比较值 mtimecmp 低 32 位, 参见第 6.1.5 节了解其详细介绍。
0xC	可读可写	mtimecmp_hi	0xFFFFFFFF	配置计时器的比较值 mtimecmp 高 32 位, 参见第 6.1.5 节了解其详细介绍。
0xFF0	可读可写	msfrst	0x00000000	生成软件复位请求, 参见第 6.1.7 节了解其详细介绍。
0xFF8	可读可写	mtimectl	0x00000000	控制计时器计数, 参见第 6.1.4 节了解其详细介绍。
0xFFC	可读可写	msip	0x00000000	生成软件中断, 参见第 6.1.6 节了解其详细介绍。
注意： <ul style="list-style-type: none"> ■ TIMER 的寄存器只支持操作尺寸 (Size) 为 word 的对齐读写访问。 ■ TIMER 的寄存器区间为 0x0 ~ 0xFFF, 除了上表中列出的寄存器之外的其他地址内的值为常数 0。 				

下文对各寄存器的功能和使用进行详细描述。

6.1.3. 通过 mtime 寄存器进行计时

TIMER 可以用于实时计时，要点如下：

- TIMER 中实现了一个 64 位的 mtime 寄存器，由{mtime_hi, mtime_lo}拼接而成，该寄存器反映了 64 位计时器的值。计时器根据低速的输入节拍信号进行自增计数，计时器默认是打开的，因此会一直进行计数。
- 在 N 级别处理器内核中，此计数器的自增频率由处理器的输入信号 mtime_toggle_a 或着内核常开时钟 core_aon_clk 控制，请参见文档《Nuclei_N 级别处理器内核简明数据手册》了解 mtime_toggle_a 的详情。

6.1.4. 通过 mtimectl 寄存器控制计时器计数

由于 TIMER 的计时器上电后默认会一直进行自增计数, TIMER 中实现了一个 mtimectl 寄存器来控制计数器的行为。mtimectl 寄存器中实现了三个域分别用于在某些特殊情况下关闭计时器计数，选择计时器的计数触发时钟，控制计时器中断触发后计数器是否清零。具体功能如表 6-2 中所示。

表 6-2 寄存器 mtimectl 的比特域

域名	比特位	属性	复位值	描述
Reserved	31:3	只读，写忽略	N/A	未使用的域，值为常数 0
CLKSRC	2	可读可写	0	控制计数器的时钟。如果该域为 1，使用常开时钟 core_aon_clk 触发 TIMER 计数，否则使用 mtime_toggle_a 信号触发 TIMER 的计数。
CMPCLEN	1	可读可写	0	控制计时器自动清零的开启或者关闭。如果该域的值 1，则在触发了计时器中断后，TIMER 会自动清零 mtime 寄存器，否则继续正常自增计数。
TIMESTOP	0	可读可写	0	控制计时器运行或者暂停。如果该

				域的值 1 ，则计时器暂停计数，否则正常自增计数。
--	--	--	--	----------------------------------

注意：ECLIC 模式下如果设置 CMPCLREN 位开启计数器自动清零，TIMER 中断请求只能维持一个脉冲，需要将 TIMER 中断设置为沿触发模式。

6.1.5. 通过 mtime 和 mtimecmp 寄存器生成计时器中断

TIMER 可以用于生成计时器中断，要点如下：

- TIMER 中实现了一个 64 位的 mtimecmp 寄存器，由{mtimecmp_hi, mtimecmp_lo} 拼接而成，该寄存器作为计时器的比较值，假设计时器的值 mtime 大于或者等于 mtimecmp 的值，则产生计时器中断。软件可以通过改写 mtimecmp 或者 mtime 的值（使得 mtimecmp 大于 mtime 的值）来清除计时器中断。

注意：计时器中断是连接到 ECLIC 单元进行统一管理，有关 ECLIC 的详情请参见第 6.2 节。

6.1.6. 通过 msip 寄存器生成软件中断

TIMER 可以用于生成软件中断。TIMER 中实现了一个 msip 寄存器，如表 6-3 中所示，msip 寄存器只有最低位为有效位，该有效位直接作为软件中断，因此：

- 软件写通过写 1 至 msip 寄存器产生软件中断；
- 软件可通过写 0 至 msip 寄存器来清除该软件中断。

注意：软件中断是连接到 ECLIC 单元进行统一管理，有关 ECLIC 的详情请参见第 6.2 节。

表 6-3 寄存器 msip 的比特域

域名	比特位	属性	复位值	描述
Reserved	7:1	只读，写忽略	N/A	未使用的域，值为常数 0
MSIP	0	可读可写	0	该域用于产生软件中断

6.1.7. 通过 msftrst 寄存器生成软件复位请求

TIMER 可以用于生成软件复位请求。TIMER 中实现了一个 msftrst 寄存器，如所示，msftrst 寄存器只有最高位为有效位，该有效位直接作为软件复位请求。详细的功能描述如下：

- 软件通过写 `0x80000a5f` 至 `msftrst`, 才能使 `msftrst` 最高位更新为 `1`, 产生软件复位请求。此写法是为了防止软件误操作导致的不预期的软件复位请求。
- `msftrst` 的最高位只能通过复位操作生效来清零, 所以当软件复位请求生效后, `msftrst` 的最高位会被清除。

注意: 软件复位请求是发送给 Core 外 SOC, 再由 Core 外的 SOC 生成复位信号发回给 Core。

表 6-4 寄存器 `msftrst` 的比特域

域名	比特位	属性	复位值	描述
MSFTRST	31	可读可写	0	该域用于产生软件复位请求
Reserved	30: 0	只读, 写忽略	N/A	未使用的域, 值为常数 0

6.2. ECLIC 介绍

N 级别处理器内核支持在 RISC-V 标准 CLIC 基础上优化而来的“改进型内核中断控制器 (Enhanced Core Local Interrupt Controller, ECLIC)”, 用于管理所有的中断源。

注意:

- ECLIC 只服务于一个处理器内核, 为该处理器内核私有。
- ECLIC 的软件编程模型也向后兼容标准的 CLIC。

6.2.1. ECLIC 简介

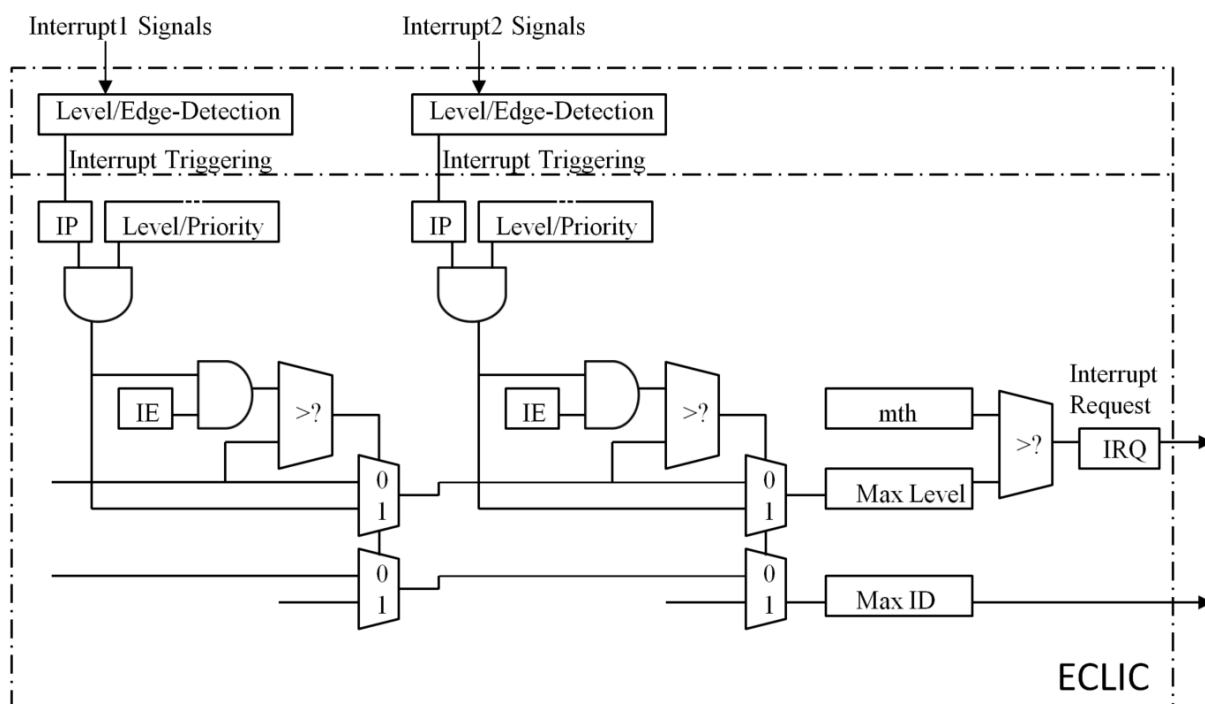


图 6-1 ECLIC 逻辑结构示意图

ECLIC 用于对多个内部和外部中断源进行仲裁、发送请求并支持中断嵌套。ECLIC 的寄存器如表 6-6 所述，逻辑结构如图 6-1 所示，相关概念如下：

- ECLIC 中断目标
- ECLIC 中断源
- ECLIC 中断源的编号
- ECLIC 的寄存器
- ECLIC 中断源的使能位
- ECLIC 中断源的等待标志位
- ECLIC 中断源的电平或边沿属性
- ECLIC 中断源的级别和优先级

- ECLIC 中断源的向量或非向量处理
- ECLIC 中断目标的阈值级别
- ECLIC 中断的仲裁机制
- ECLIC 中断的响应、嵌套、咬尾机制

下文将分别予以详述。

6.2.2. ECLIC 中断目标

ECLIC 单元生成一根中断线，发送给处理器内核（作为中断目标），其关系结构如图 6-2 所示。

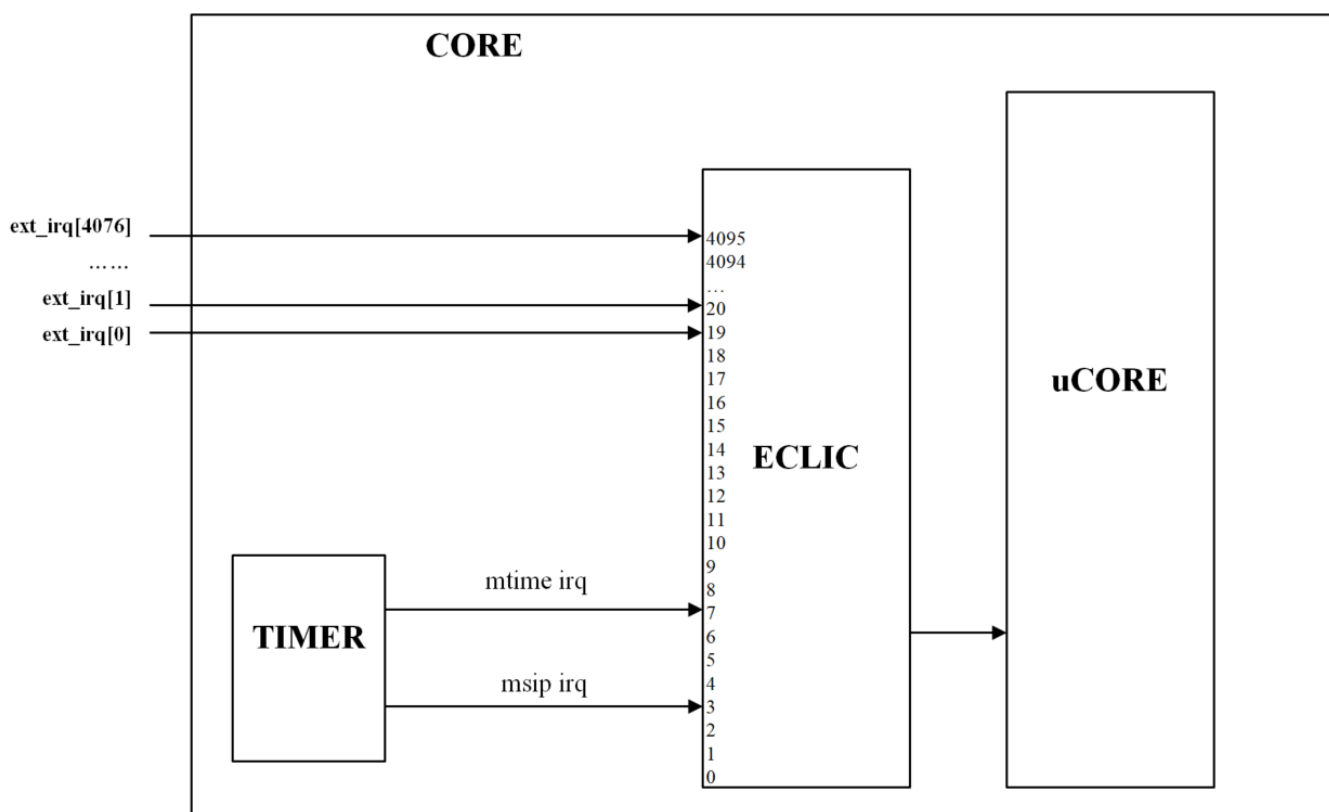


图 6-2 ECLIC 关系结构图

6.2.3. ECLIC 中断源

如图 6-2 所示,ECLIC 理论上从编程模型上可以支持多达 4096 个中断源(Interrupt Source)。ECLIC 为每个中断源定义了如下特性和参数:

- 编号 (ID)
- 使能位 (IE)
- 等待标志位 (IP)
- 电平或边沿属性 (Level or Edge-Triggered)
- 级别和优先级 (Level and Priority)
- 向量或非向量处理 (Vector or Non-Vector Mode)

下文分别予以介绍。

6.2.4. ECLIC 中断源的编号 (ID)

ECLIC 为每个中断源分配了一个独一无二的编号 (ID)。譬如, 假设某 ECLIC 的硬件实现真正支持 4096 个 ID, 则 ID 应为 0 至 4095。注意:

- 在 N 级别处理器内核中, 中断 ID 编号 0 至 18 的中断被预留作为了内核特殊的内部中断。
- 普通外部中断分配的中断源 ID 从 19 开始, 用户可以用于连接外部中断源。

详细介绍如表 6-5 中所示。

表 6-5 ECLIC 中断源编号和分配

ECLIC 中断编号	功能	中断源介绍
0	预留	N 级别处理器内核没有使用该中断。
1	预留	N 级别处理器内核没有使用该中断。
2	预留	N 级别处理器内核没有使用该中断。
3	软件中断	N 级别处理器内核的 TIMER 单元生成的软件中断。
4	预留	N 级别处理器内核没有使用该中断。
5	预留	N 级别处理器内核没有使用该中断。
6	预留	N 级别处理器内核没有使用该中断。
7	计时器中断	N 级别处理器内核的 TIMER 单元生成的计时器中

		断。
8	预留	N 级别处理器内核没有使用该中断。
9	预留	N 级别处理器内核没有使用该中断。
10	预留	N 级别处理器内核没有使用该中断。
11	预留	N 级别处理器内核没有使用该中断。
12	预留	N 级别处理器内核没有使用该中断。
13	预留	N 级别处理器内核没有使用该中断。
14	预留	N 级别处理器内核没有使用该中断。
15	预留	N 级别处理器内核没有使用该中断。
16	预留	N 级别处理器内核没有使用该中断。
17	预留	N 级别处理器内核没有使用该中断。
18	预留	N 级别处理器内核没有使用该中断。
19 ~ 4096	外部中断	普通外部中断供用户连接使用。注意： ■ 虽然 ECLIC 从编程模型上支持最多 4096 个中断源，但是实际硬件支持的中断源数目反映在信息寄存器 clicinfo.NUM_INTERRUPT 中。

6.2.5. ECLIC 的寄存器

ECLIC 是一个存储器地址映射的单元：

- ECLIC 单元在 N 级别处理器内核中的基地址请参见《Nuclei_N 级别处理器内核简明数据手册》中的介绍。
- ECLIC 单元内的寄存器和地址偏移量如表 6-6 中所示。

表 6-6 ECLIC 寄存器的单元内地址偏移量

	属性	名称	宽度
0x0000	可读可写	cliccfg	8 位
0x0004	只读，写忽略	clicinfo	32 位
0x000b	可读可写	mth	8 位
0x1000+4*i	可读可写	clicintip[i]	8 位
0x1001+4*i	可读可写	clicintie[i]	8 位
0x1002+4*i	可读可写	clicintattr[i]	8 位
0x1003+4*i	可读可写	clicintctl[i]	8 位
注意：			
■ 上述的 i 表示中断的 ID 编号，带有[i]后缀的寄存器表示这是针对每个中断源会有一份独立的寄存器。			
■ ECLIC 的寄存器支持操作尺寸（Size）为 byte、half-word、或 word 的对齐读写访问。			
■ 向上述“只读”寄存器进行写操作会被忽略，但是不会产			

- 生总线出错异常。
- 实际的 ECLIC 可能没有配置 4096 个中断源，那么不存在的中断源对应寄存器的值为常数 0。
 - ECLIC 单元内寄存器区间为 0x0000 ~ 0xFFFF，除了上表中列出的寄存器之外的其他地址内的值为常数 0。

下文对各个寄存器进行详细介绍。

6.2.5.1 寄存器 cliccfg

cliccfg 寄存器是全局性的配置寄存器，软件可以通过改写该寄存器配置若干全局性的参数，其具体比特域的信息请参见表 6-7 中所示。

表 6-7 寄存器 cliccfg 的比特域

域名	比特位	属性	复位值	描述
Reserved	7:5	只读，写忽略	N/A	未使用的域，值为常数 0
nlbits	4:1	可读可写	0	用于指定 clicintctl[i] 寄存器中 Level 域的比特数，参见第 6.2.9 节了解其详细介绍。
Reserved	0	只读，写忽略	N/A	未使用的域，值为常数 1

6.2.5.2 寄存器 clicinfo

clicinfo 寄存器是全局性的信息寄存器，软件可以通过读该寄存器查看若干全局性的参数，其具体比特域的信息请参见表 6-8 中所示。

表 6-8 寄存器 clicinfo 的比特域

域名	比特位	属性	复位值	描述
Reserved	31:25	只读，写忽略	N/A	未使用的域，值为常数 0
CLICINTCTLBITS	24:21	只读，写忽略	N/A	用于指定 clicintctl[i] 寄存器中有效位的比特数，参见第 6.2.9 节了解其详细介绍。
VERSION	20:13	只读，写忽略	N/A	硬件实现的版本号
NUM_INTERRUPT	12:0	只读，写忽略	N/A	硬件支持的中断源数目

6.2.5.1 寄存器 mth

mth 寄存器是中断目标的阈值级别寄存器，软件可以通过改写该寄存器配置中断目标的阈值级别，其具体比特域的信息请参见表 6-9 中所示。

表 6-9 寄存器 mth 的比特域

域名	比特位	属性	复位值	描述
mth	7:0	可读可写	N/A	中断目标的阈值级别寄存器，参见第 6.2.11 节了解其详细介绍。

6.2.5.2 寄存器 clicintip[i]

clicintip[i]寄存器是中断源的等待标志寄存器，其具体比特域的信息请参见表 6-10 中所示。

表 6-10 寄存器 clicintip[i]的比特域

域名	比特位	属性	复位值	描述
Reserved	7:1	只读，写忽略	N/A	未使用的域，值为常数 0
IP	0	可读可写	0	中断源的等待标志位，参见第 6.2.7 节了解其详细介绍。

6.2.5.3 寄存器 clicintie[i]

clicintie[i]寄存器是中断源的使能寄存器，其具体比特域的信息请参见表 6-11 中所示。

表 6-11 寄存器 clicintie[i]的比特域

域名	比特位	属性	复位值	描述
Reserved	7:1	只读，写忽略	N/A	未使用的域，值为常数 0
IE	0	可读可写	0	中断源的使能位，参见第 6.2.6 节了解其详细介绍。

6.2.5.4 寄存器 clicintattr[i]

clicintattr[i]寄存器是中断源的属性寄存器，软件可以通过改写该寄存器配置中断源的若干属性，其具体比特域的信息请参见表 6-12 中所示。

表 6-12 寄存器 clicintattr[i]的比特域

域名	比特位	属性	复位值	描述
----	-----	----	-----	----

Reserved	7:6	只读，写忽略	N/A	未使用的域，值为常数 3
Reserved	5:3	只读，写忽略	N/A	未使用的域，值为常数 0
trig	2:1	可读可写	0	指定该中断源的电平或边沿属性，参见第 6.2.8 节了解其详细介绍。
shv	0	可读可写	0	指定该中断源使用向量处理模式还是非向量处理模式，参见第 6.2.10 节了解其详细介绍。

6.2.5.5 寄存器 clicintctl[i]

clicintctl[i]寄存器是中断源的控制寄存器，软件可以通过改写该寄存器配置中断源的级别（Level）和优先级（Priority），Level 和 Priority 域根据 cliccfg.nlbits 的值动态进行分配，参见第 6.2.9 节了解其详细介绍。

6.2.6. ECLIC 中断源的使能位（IE）

如图 6-2 所示，ECLIC 为每个中断源分配了一个中断使能位（IE），反映在寄存器 clicintie[i].IE 中，其功能如下：

- 每个中断源的 clicintie[i]寄存器是存储器地址映射的可读可写寄存器，从而使得软件可以对其编程。
- 如果 clicintie[i]寄存器被编程配置成为 0，则意味着此中断源被屏蔽。
- 如果 clicintie[i]寄存器被编程配置成为 1，则意味着此中断源被打开。

6.2.7. ECLIC 中断源的等待标志位（IP）

如图 6-2 所示，ECLIC 为每个中断源分配了一个中断等待标志位（IP），反映在寄存器 clicintip[i].IP 中，其功能如下：

- 如果某个中断源的 IP 位为高，则表示该中断源被触发。中断源的触发条件取决于它是电平触发还是边沿触发的属性，请参见第 6.2.8 节的详细介绍。
- 中断源的 IP 位软件可读可写，软件写 IP 位的行为取决于它是电平触发还是边沿触发的属性，请参见第 6.2.8 节的详细介绍。
- 对于边沿触发的中断源，其 IP 还可能存在硬件自清的行为，请参见第 6.2.8 节的详细介绍。

6.2.8. ECLIC 中断源的电平或边沿属性（Level or Edge-Triggered）

如图 6-2 所示，ECLIC 的每个中断源均可以设置电平触发或者边沿触发的属性（通过寄存器 clicintattr[i] 的 trig 域），其要点如下：

- 当 clicintattr[i].trig[0] == 0 时，设置该中断属性为电平触发的中断：
 - 如果该中断源被配置为电平触发，中断源的 IP 位会实时反映该中断源的电平值。
 - 如果该中断源被配置为电平触发，由于中断源的 IP 位实时反映该中断源的电平值，所以软件对该中断 IP 位的写操作会被忽略，即，软件无法通过写操作设置或者清除 IP 位的值。如果软件需要清除中断，只能通过清除中断的最终源头的方式进行。
- 当 clicintattr[i].trig[0] == 1 和 clicintattr[i].trig[1] == 0 时，设置该中断属性为上升沿触发的中断：
 - 如果该中断源被配置为上升沿触发，则 ECLIC 检测到该中断源的上升沿时，该中断源在 ECLIC 中被触发，该中断源的 IP 位被置高。
 - 如果该中断源被配置为上升沿触发，软件对该中断 IP 位的写操作会生效，即，软件可以通过写操作设置或者清除 IP 位的值。
 - 注意：对于上升沿触发的中断而言，为了能够提高中断处理的效率，当该中断被响应，处理器内核跳入中断服务程序（Interrupt Service Routines, ISR）之时，ECLIC 的硬件会自动清除该中断的 IP 位，从而无需软件在 ISR 内部对该中断的 IP 位进行清除。
- 当 clicintattr[i].trig[0] == 1 和 clicintattr[i].trig[1] == 1 时，设置该中断属性为下降沿触发的中断：
 - 如果该中断源被配置为下降沿触发，则 ECLIC 检测到该中断源的下降沿时，该中断源在 ECLIC 中被触发，该中断源的 IP 位被置高。
 - 如果该中断源被配置为下降沿触发，软件对该中断 IP 位的写操作会生效，即，软件可以通过写操作设置或者清除 IP 位的值。
 - 注意：对于下降沿触发的中断而言，为了能够提高中断处理的效率，当该中断被响应，处理器内核跳入中断服务程序（Interrupt Service Routines, ISR）之时，ECLIC 的硬件会自动清除该中断的 IP 位，从而无需软件在 ISR 内部对该中断的 IP 位进行清除。

6.2.9. ECLIC 中断源的级别和优先级（Level and Priority）

如图 6-2 所示, ECLIC 的每个中断源均可以设置特定的级别和优先级(通过寄存器 `clcintctl[i]`), 其要点如下:

- 每个中断源的 `clcintctl[i]` 寄存器理论上有 8 位宽, 其中硬件真正实现的位有效位数由 `clcinfo` 寄存器的 `CLICINTCTLBITS` 域来指定。譬如, 假设 `clcinfo.CLICINTCTLBITS` 域的值 6, 则表示 `clcintctl[i]` 寄存器只有高 6 位是真正有效位, 最低 2 位的值为常数 1, 如图 6-3 中示例所示。
- 注意: `CLICINTCTLBITS` 域的值是只读的固定常数, 软件无法对其进行编程改写。其理论上的合理范围是 $2 \leq \text{CLICINTCTLBITS} \leq 8$, 具体的实际值由处理器内核的硬件实现决定。
- 在 `clcintctl[i]` 寄存器的有效位中, 包含两个动态的域, 分别用于指定该中断源的级别 (Level) 和优先级 (Priority)。Level 域的宽度由 `clccfg` 寄存器的 `nlbits` 域来指定。譬如, 假设 `clccfg.nlbits` 域的值 4, 则表示 `clcintctl[i]` 寄存器有效位的高 4 位是 Level 域, 其他的低位有效位为 Priority 域, 如图 6-3 中示例所示。
- 注意: `clccfg.nlbits` 域的值是可读可写域, 软件可以对其进行编程改写。

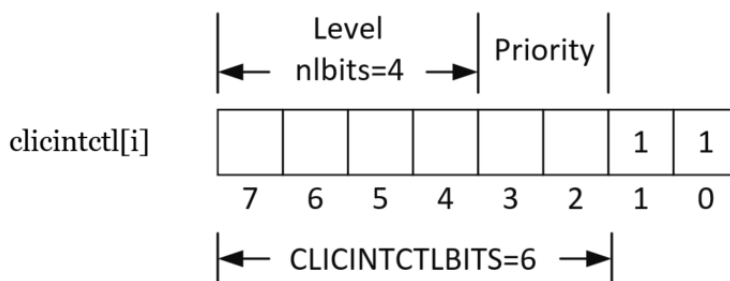


图 6-3 寄存器 `clcintctl[i]` 的格式示例

- 中断源的级别 (Level) 相关的要点如下:
 - Level 的数字值采取左对齐的方式进行解读, 有效位宽 (由 `clccfg.nlbits` 指定) 之外的低位全部采用补常数 1 的方式填充, 如图 6-4 中示例所示。
 - ◆ 注意: 如果 `clccfg.nlbits > clcinfo.CLICINTCTLBITS`, 则意味着 `nlbits` 指示的

位数超出了 clicintctl[i]寄存器的有效位,则超出的位全部采用补常数 1 的方式填充。

- ◆ 注意: 如果 cliccfg.nlbits = 0, Level 的数字值会被认为是固定的 255。如图 6-5 中示例所示。
- Level 的数字值越大,则表示其级别越高,注意:
 - ◆ 高级别的中断可以打断低级别的中断处理,从而形成中断嵌套,请参见第 5.11 节的详细介绍。
 - ◆ 多个中断同时等待(IP 位为高),ECLIC 需要仲裁决定哪个中断被发送给内核进行处理,仲裁时需要参考每个中断源的 Level 数字值。请参见第 5.5 节的详细介绍。

#nlbits	编码	Level的数字值							
1	L..... (= L1111111)				127,				255
2	LL..... (= LL111111)		63,		127,		191,		255
3	LLL..... (= LLL11111)	31,	63,	95,	127,	159,	191,	223,	255
4	LLLL.... (= LLLL1111)	15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255							
“L” 比特表示Level的域									
“.” 表示Level域之外的其他比特，采用全部补1的方式									

图 6-4 Level 的数字值解读方式

cliccfg设置的若干示例:			
CLICINTCTLBITS	nlbits	clicintctl[i]	Level的数字值
0	2	255
1	2	L.....	127, 255
2	2	LL.....	63, 127, 191, 255
3	3	LLL.....	31, 63, 95, 127, 159, 191, 223, 255
4	1	LPPP...	127, 255

图 6-5 cliccfg 设置的若干示例

- 中断源的优先级（Priority）相关的要点如下：
 - Priority 的数字值也采取左对齐的方式进行解读，有效位宽（`clinfo.CLICINTCTLBITS - cliccfg.nlbits`）之外的低位全部采用补常数 1 的方式填充。
 - Priority 的数字值越大，则表示其优先级越高，注意：
 - ◆ 中断优先级（Priority）不参与中断嵌套的判断，即中断能否嵌套与中断优先级（Priority）的数值大小没有关系，而是与中断级别（Level）的数值大小有关。
 - ◆ 多个中断同时 Pending 时，ECLIC 需要仲裁决定哪个中断被发送给内核进行处理，仲裁时需要参考每个中断源的 Priority 数字值。请参见第 6.2.12 节的详细介绍。

6.2.10. ECLIC 中断源的向量或非向量处理（Vector or Non-Vector Mode）

ECLIC 的每个中断源均可以设置成向量或者非向量处理（通过寄存器 `clicintattr[i]` 的 `shv` 域），其要点如下：

- 如果被配置成为向量处理模式，则该中断被处理器内核响应后，处理器直接跳入该中断的向量入口（Vector Table Entry）存储的目标地址。有关中断向量处理模式的详细介绍，请参见第 5.13 节。
- 如果被配置成为非向量处理模式，则该中断被处理器内核响应后，处理器直接跳入所有中断共享的入口地址。有关中断非向量处理模式的详细介绍，请参见第 5.13 节。

6.2.11. ECLIC 中断目标的阈值级别

如图 6-1 中所示，ECLIC 可以设置特定的中断目标的阈值级别（`nth`），其要点如下：

- `nth` 寄存器是完整的 8 位寄存器，所有位均可读可写，软件可以通过写此寄存器配置目标阈值。注意：该阈值表征的是一种级别（Level）的数字值。
- ECLIC 最终仲裁出的中断的“级别（Level）数字值”只有高于“`nth` 寄存器中的值”，该中断才能够被发送给处理器内核。

6.2.12. ECLIC 中断的仲裁机制

如图 6-2 所示，ECLIC 对其所有中断源进行仲裁选择的原则如下：

- 只有满足下列所有条件的中断源才能参与仲裁：
 - 中断源的使能位（`clicintie[i]`寄存器）必须为 1。
 - 中断源的等待标志位（`clicintip[i]`寄存器）必须为 1。
- 从所有参与仲裁的中断源中进行仲裁的规则为：
 - 首先判断级别（Level），Level 数字值越大的中断源，其仲裁优先级越高。
 - 如果 Level 相等，则其次判断优先级（Priority），Priority 数字值越大的中断源，其仲裁优先级越高。
 - 如果 Level 和 Priority 都相等，则再次判断判断中断 ID，中断 ID 越大的中断源，其仲裁优先级越高。
- 如果最后仲裁出的中断源的 Level 数字值高于中断目标的阈值级别（`mtb`），则产生最终的中断请求，将通往处理器内核的中断请求信号拉高。

6.2.13. ECLIC 中断的响应、嵌套、咬尾机制

ECLIC 中断请求发送给处理器内核之后，处理器内核将对其进行响应。通过 ECLIC 和内核协同，可以支持中断嵌套、快速咬尾等等机制。请参见第 5.6 节、第 5.11 节、第 5.12 节的详细介绍。

7. N 级别处理器内核 CSR 寄存器介绍

7.1. N 级别处理器内核 CSR 寄存器概述

RISC-V 的架构中定义了一些控制和状态寄存器（Control and Status Register, CSR），用于配置或者记录一些运行的状态。CSR 寄存器是处理器核内部的寄存器，使用其专有的 12 位地址编码空间。

7.2. N 级别处理器内核的 CSR 寄存器列表

Nuclei N 级别处理器内核支持的 CSR 寄存器列表如表 7-1 所示，其中包括 RISC-V 标准的 CSR 寄存器（RV32IMAFDC 架构支持 Machine Mode 和 User Mode 相关）和 N 级别处理器内核自定义扩展的 CSR 寄存器。

表 7-1 N 级别处理器内核支持的 CSR 寄存器列表

类型	CSR 地址	读写属性	名称	全称
RISC-V 标准 CSR (Machine Mode)	0xF11	MRO	mvendorid	商业供应商编号寄存器 (Machine Vendor ID Register)
	0xF12	MRO	marchid	架构编号寄存器 (Machine Architecture ID Register)
	0xF13	MRO	mimpid	硬件实现编号寄存器 (Machine Implementation ID Register)
	0xF14	MRO	mhartid	Hart 编号寄存器 (Hart ID Register)
	0x300	MRW	mstatus	异常处理状态寄存器
	0x301	MRW	misa	指令集架构寄存器 (Machine ISA Register)
	0x304	MRW	mie	局部中断屏蔽控制寄存器 (Machine Interrupt Enable Register)
	0x305	MRW	mtvec	异常入口基地址寄存器
	0x307	MRW	mtvt	ECLIC 中断向量的基地址
	0x340	MRW	mscratch	暂存寄存器 (Machine Scratch Register)
	0x341	MRW	mepc	异常 PC 寄存器 (Machine Exception Program Counter)
	0x342	MRW	mcause	异常原因寄存器 (Machine Cause Register)
	0x343	MRW	mtval	异常值寄存器 (Machine Trap Value Register)
	0x344	MRW	mip	中断等待寄存器 (Machine Interrupt Pending Register)
	0x345	MRW	mnxti	标准寄存器用于使能中断，处理下一个中断并返回下一个中断的 Handler 入口地址
	0x346	MRW	mintstatus	标准寄存器用于保存当前中断 Level
	0x348	MRW	mscratchsw	标准寄存器用于在特权模式变化时交换 mscratch 与目的寄存器的值
	0x349	MRW	mscratchswl	标准寄存器用于在中断 Level 变化时交换 mscratch 与目的寄存器的值
	0xB00	MRW	mcycle	周期计数器的低 32 位 (Lower 32 bits of Cycle counter)
	0xB80	MRW	mcycleh	周期计数器的高 32 位 (Upper 32 bits of Cycle counter)
	0xB02	MRW	minstret	完成指令计数器的低 32 位 (Lower 32 bits of Instructions-retired counter)
	0xB82	MRW	minstreth	完成指令计数器的高 32 位 (Upper 32 bits of Instructions-retired counter)

	N/A	MRW	mtime	计时器寄存器 (Machine-mode timer register)
	N/A	MRW	mtimecmp	计时器比较寄存器 (Machine-mode timer compare register)
	N/A	MRW	msip	机器模式软件中断等待寄存器 (Machine-mode Software Interrupt Pending Register)
	N/A	MRW	mtimectl	用于停止计时器, 切换计时时钟, 控制 TIMER 中断发生时, 计数器是否自动清零
	0x3A0+x	MRW	pmpcfg<x>	PMP 表项权限配置寄存器
	0x3B0+x	MRW	pmpaddr<x>	PMP 表项 x 的地址配置寄存器
RISC-V 标准 CSR (User Mode)	0x001	URW	fflags	浮点累积异常 (Floating-Point Accrued Exceptions)。 注意: 此寄存器只有在配置了浮点指令 (“F” 或者 “D” 指令子集) 时才会存在。
	0x002	URW	frm	浮点动态舍入模式 (Floating-Point Dynamic Rounding Mode)。 注意: 此寄存器只有在配置了浮点指令 (“F” 或者 “D” 指令子集) 时才会存在。
	0x003	URW	fcsr	浮点控制和状态寄存器 (Floating-Point Control and Status Register)。 注意: 此寄存器只有在配置了浮点指令 (“F” 或者 “D” 指令子集) 时才会存在。
	0xC00	URO	cycle	mcycle 寄存器的只读副本 注意: 该寄存器在 User Mode 下是否可读由 CSR 寄存器 mcounteren 的 CY 比特域来控制, 请参见第 7.4.37 节了解其详情。
	0xC01	URO	time	mtime 寄存器的只读副本 注意: 该寄存器在 User Mode 下是否可读由 CSR 寄存器 mcounteren 的 TM 比特域来控制, 请参见第 7.4.37 节了解其详情。
	0xC02	URO	instret	minstret 寄存器的只读副本 注意: 该寄存器在 User Mode 下是否可读由 CSR 寄存器 mcounteren 的 IR 比特域来控制, 请参见第 7.4.37 节了解其详情。
	0xC80	URO	cycleh	mcycleh 寄存器的只读副本 注意: 该寄存器在 User Mode 下是否可读由 CSR 寄存器 mcounteren 的 CY 比特域来控制, 请参见第 7.4.37 节了解其详情。
	0xC81	URO	timeh	mtimeh 寄存器的只读副本 注意: 该寄存器在 User Mode 下是否可读由 CSR 寄存器 mcounteren 的 TM 比特域来控制, 请参见第 7.4.37 节了解其详情。
	0xC82	URO	instreth	minstreth 寄存器的只读副本 注意: 该寄存器在 User Mode 下是否可

				读由 CSR 寄存器 mcounteren 的 IR 比特域来控制，请参见第 7.4.37 节了解其详情。
	0x801	URW	ucode	DSP 运算溢出标志寄存器 注意：此寄存器只有在配置了“P”扩展指令集时才会存在。
N 级别处理器内核 自定义 CSR	0x320	MRW	mcountinhibit	自定义寄存器用于控制计数器的开启和关闭
	0x7c0	MRW	milm_ctl	ILM 地址空间的打开或关闭
	0x7c1	MRW	mdlm_ctl	DLM 地址空间的打开或关闭
	0x7c2	MRW	mecc_code	ECC 校验码注入寄存器，可用于模拟 ECC 错误
	0x7c3	MRW	mnvec	NMI 处理入口基地址寄存器
	0x7c4	MRW	msubm	自定义寄存器用于保存 Core 当前的 Trap 类型，以及进入 Trap 前的 Trap 类型。
	0x7c9	MRW	mdcause	自定义寄存器，作为 mcause 的补充，用于保存详细的异常原因。
	0x7ca	MRW	mcache_ctl	自定义寄存器用来控制 I-CACHE 相关功能 注意：此寄存器只有配置了 I-CACHE 才会存在
	0x7d0	MRW	mmisc_ctl	自定义寄存器用于控制 NMI, Misaligned Access 和 BPU 的相关功能
	0x7d6	MRW	msavestatus	自定义寄存器用于保存 mstatus 值 注意：此寄存器只有配置了两级异常嵌套恢复才会存在
	0x7d7	MRW	msaveepc1	自定义寄存器用于保存第一级嵌套 NMI 或异常的 mepc 注意：此寄存器只有配置了两级异常嵌套恢复才会存在
	0x7d8	MRW	msavecause1	自定义寄存器用于保存第一级嵌套 NMI 或异常的 mcause 注意：此寄存器只有配置了两级异常嵌套恢复才会存在
	0x7d9	MRW	msaveepc2	自定义寄存器用于保存第二级嵌套 NMI 或异常的 mepc 注意：此寄存器只有配置了两级异常嵌套恢复才会存在
	0x7da	MRW	msavecause2	自定义寄存器用于保存第二级嵌套 NMI 或异常的 mcause 注意：此寄存器只有配置了两级异常嵌套恢复才会存在
	0x7db	MRW	msavedcause1	自定义寄存器用于保存第一级嵌套异常的 mdcause 注意：此寄存器只有配置了两级异常嵌套恢复才会存在
	0x7dc	MRW	msavedcause2	自定义寄存器用于保存第二级嵌套异常的 mdcause 注意：此寄存器只有配置了两级异常嵌套恢复才会存在
	0x7de	MRW	mecc_lock	用于锁定 ECC 的配置，所有相关的 ECC 寄存器均不能再修改，除非复位

	0x7eb	MRW	pushmsubm	自定义寄存器用于将 msubm 的值存入堆栈地址空间
	0x7ec	MRW	mtvt2	自定义寄存器用于设定非向量中断处理模式的中断入口地址
	0x7ed	MRW	jalmnxti	自定义寄存器用于使能 ECLIC 中断，该寄存器的读操作能处理下一个中断同时返回下一个中断 Handler 的入口地址，并跳转至此地址。
	0x7ee	MRW	pushmcause	自定义寄存器用于将 mcause 的值存入堆栈地址空间
	0x7ef	MRW	pushmepc	自定义寄存器用于将 mepc 的值存入堆栈地址空间
	0x7fo	MRO	mppicfg_info	PPI 配置信息
	0x7f1	MRO	mfiocfg_info	FIO 配置信息
	0x811	URW	sleepvalue	WFI 的休眠模式寄存器
	0x812	URW	txevt	发送 Event 寄存器
	0x810	URW	wfe	Wait for Event 控制寄存器
	0xfc0	MRO	micfg_info	ILM and I-Cache 配置信息
	0xfc1	MRO	mdcfg_info	DLM and D-Cache 配置信息
	0xfc2	MRO	mcfg_info	处理器配置信息
注意： <ul style="list-style-type: none"> ■ MRW 表示 Machine Mode Readable/Writeable ■ MRO 表示 Machine Mode Read-Only ■ URW 表示 User Mode Readable/Writeable ■ URO 表示 User Mode Read-Only 				

7.3. N 级别处理器内核的 CSR 寄存器的访问权限

Nuclei N 级别处理器内核对于 CSR 寄存器的访问权限规定如下：

- 无论是在 Machine Mode 还是 User Mode 下：
 - 如果向不存在的 CSR 寄存器地址区间进行读写操作，则会产生 Illegal Instruction Exception。
 - 注意：fflags、frm、fcsr 寄存器只有在配置了浮点指令（“F” 或者 “D” 指令子集）时才会存在。
- 在 Machine Mode 下：

- 对 MRW 或者 URW 属性的 CSR 寄存器进行读写操作则一切正常。
- 对 MRO 或者 URO 属性的 CSR 寄存器进行读操作则一切正常。
- 如果向 MRO 或者 URO 属性的 CSR 寄存器进行写操作，则会产生 Illegal Instruction Exception。

■ 在 User Mode 下：

- 对 URW 属性的 CSR 寄存器进行读写操作均一切正常。
- 对 URO 属性的 CSR 寄存器进行读操作则一切正常。
 - ◆ 注意：对于 URO 属性的 cycle、cycleh、time、timeh、instret、instreth 寄存器，其能否可读还受 mcounteren 的相关比特域来控制，请参见 7.4.37 了解其详情。
- 如果向 URO 属性的 CSR 寄存器进行写操作，则会产生 Illegal Instruction Exception。
- 如果向 MRW 或者 MRO 属性的 CSR 寄存器进行读写操作，则会产生 Illegal Instruction Exception。

7.4. N 级别处理器内核支持的 RISC-V 标准 CSR

本节介绍 N 级别处理器核支持的 RISC-V 标准 CSR 寄存器

7.4.1. misa

misa 寄存器用于指示当前处理器所支持的架构特性。

misa 寄存器的最高两位用于指示当前处理器所支持的架构位数：

- 如果最高两位值为 1，则表示当前为 32 位架构（RV32）。
- 如果最高两位值为 2，则表示当前为 64 位架构（RV64）。
- 如果最高两位值为 3，则表示当前为 128 位架构（RV128）。

misa 寄存器的低 26 位用于指示当前处理器所支持的 RISC-V ISA 中不同模块化指令子集，每一位表示的模块化指令子集如图 7-1 中所示。该寄存器其他未使用到的比特域为常数 0。

Bit	Character	Description
0	A	Atomic extension
1	B	<i>Tentatively reserved for Bit operations extension</i>
2	C	Compressed extension
3	D	Double-precision floating-point extension
4	E	RV32E base ISA
5	F	Single-precision floating-point extension
6	G	Additional standard extensions present
7	H	<i>Reserved</i>
8	I	RV32I/64I/128I base ISA
9	J	<i>Tentatively reserved for Dynamically Translated Languages extension</i>
10	K	<i>Reserved</i>
11	L	<i>Tentatively reserved for Decimal Floating-Point extension</i>
12	M	Integer Multiply/Divide extension
13	N	User-level interrupts supported
14	O	<i>Reserved</i>
15	P	<i>Tentatively reserved for Packed-SIMD extension</i>
16	Q	Quad-precision floating-point extension
17	R	<i>Reserved</i>
18	S	Supervisor mode implemented
19	T	<i>Tentatively reserved for Transactional Memory extension</i>
20	U	User mode implemented
21	V	<i>Tentatively reserved for Vector extension</i>
22	W	<i>Reserved</i>
23	X	Non-standard extensions present
24	Y	<i>Reserved</i>
25	Z	<i>Reserved</i>

图 7-1 **misa** 寄存器低 26 位各域表示的模块化指令子集

注意：**misa** 寄存器在 RISC-V 架构文档中被定义为可读可写的寄存器，从而允许某些处理器的设计能够动态地配置某些特性。但是在 N 级别处理器内核的实现中，**misa** 寄存器为只读寄存器，恒定地反映不同型号处理器核所支持的 ISA 模块化子集，写此寄存器会被忽略。譬如 N205 核支持 RV32IMC，则反映于此寄存器中，最高两位值为 1，低 26 位中 I/M/C 对应域的值即为高。

7.4.2. mie

ECLIC 中断模式下 **mie** 寄存器的控制位不起作用，读 **mie** 返回全 0。

7.4.3. mvendorid

此寄存器是只读寄存器，用于反映该处理器核的商业供应商编号（Vendor ID）。

如果此寄存器的值为 0，则表示此寄存器未实现。

7.4.4. marchid

此寄存器是只读寄存器，用于反映该处理器核的硬件实现微架构编号（Microarchitecture ID）。

如果此寄存器的值为 0，则表示此寄存器未实现。

7.4.5. mimpid

此寄存器是只读寄存器，用于反映该处理器核的硬件实现编号（Implementation ID）。

如果此寄存器的值为 0，则表示此寄存器未实现。

7.4.6. mhartid

此寄存器是只读寄存器，用于反映当前 Hart 的编号（Hart ID）。

Hart（取“Hardware Thread”之意）表示一个硬件线程，单个处理器核中可能实现多份硬件线程，譬如硬件超线程（Hyper-threading）技术，每套线程有自己独立的寄存器组等上下文资源，但大多数的运算资源均被所有硬件线程复用，因此面积效率很高。在这样的硬件超线程处理器中，一个核内便存在着多个硬件线程（Hart）。

N 级别处理器内核中 Hart 编号值受输入信号 core_mhartid 控制。注意：RISC-V 架构规定，如果在单 Hart 或者多 Hart 的系统中，起码要有一个 Hart 的编号必须是 0。

7.4.7. mstatus

mstatus 寄存器是机器模式（Machine Mode）下的状态寄存器。mstatus 寄存器中各控制位域如表 7-2 所示。

表 7-2 mstatus 寄存器各控制位

域	位	复位值	描述
Reserved	0	N/A	未使用的域为常数 0
SIE	1	0	参见第 7.4.8 节了解其详情
Reserved	2	N/A	未使用的域为常数 0
MIE	3	0	参见第 7.4.9 节了解其详情
Reserved	4	N/A	未使用的域为常数 0
SPIE	5	0	参见第 7.4.10 节了解其详情
Reserved	6	N/A	未使用的域为常数 0
MPIE	7	0	参见第 7.4.11 节了解其详情
Reserved	10:8	N/A	未使用的域为常数 0
MPP	12:11	0	参见第 7.4.11 节了解其详情
FS	14:13	0	参见第 7.4.12 节了解其详情
XS	16:15	0	参见第 7.4.13 节了解其详情
MPRV	17	0	参见第 7.4.14 节了解其详情
SUM	18	0	参见第 7.4.15 节了解其详情
Reserved	30:19	N/A	未使用的域为常数 0
SD	31	0	参见第 7.4.16 节了解其详情

7.4.8. mstatus 的 SIE 域

mstatus 寄存器中的 SIE 域只有在配置了 TEE 才有效，否则为常数 0，具体信息请参见《Nuclei TEE Architecture》文档。

7.4.9. mstatus 的 MIE 域

mstatus 寄存器中的 MIE 域表示机器模式下的全局中断使能：

当 MIE 域的值为 1 时，表示中断的全局开关打开，中断能够被正常响应；

当 MIE 域的值为 0 时，表示全局关闭中断，中断被屏蔽，无法被响应。

当处理器内核处于用户模式下时，中断无法被屏蔽，一律正常响应。

注意：N 级别处理器内核在进入异常、中断或者 NMI 处理模式时，MIE 的值会被更新成为 0（意味着进入异常、中断或者 NMI 处理模式后中断被屏蔽）。

7.4.10. mstatus 的 SPIE 与 SPP 域

mstatus 寄存器中的 SPIE 和 SPP 域只有在配置了 TEE 才有效，否则为常数 0，具体信息请参见《Nuclei TEE Architecture》文档。

7.4.11. mstatus 的 MPIE 和 MPP 域

mstatus 寄存器中的 MPIE 和 MPP 域分别用于自动保存进入异常，NMI 和中断之前 mstatus.MIE、特权模式（Privilege Mode）时进行自动恢复。

N 级别处理器内核进入异常时更新 mstatus 寄存器 MPIE 和 MPP 域的硬件行为，请参见 3.4.5 节了解其详情。

N 级别处理器内核退出异常时（在异常处理模式下执行 mret 指令）更新 mstatus 寄存器 MPIE 和 MPP 域的硬件行为，请参见 3.5.2 节了解其详情。

N 级别处理器内核进入 NMI 时更新 mstatus 寄存器 MPIE 和 MPP 域的硬件行为，请参见 4.3.4 节了解其详情。

N 级别处理器内核退出 NMI 时（在异常处理模式下执行 mret 指令）更新 mstatus 寄存器 MPIE 和 MPP 域的硬件行为，请参见 4.4.2 节了解其详情。

N 级别处理器内核进入中断时更新 mstatus 寄存器 MPIE 和 MPP 域的硬件行为，请参见 5.6.5 节了解其详情。

N 级别处理器内核退出中断时（在异常处理模式下执行 mret 指令）更新 mstatus 寄存器 MPIE 和 MPP 域的硬件行为，请参见 5.7.2 节了解其详情。

注意： mstatus.MPIE 域和 mstatus.MPP 域的值与 mcause.MPIE 域和 mcause.MPP 域的值是镜像关系，即，在正常情况下， mstatus.MPIE 域的值与 mcause.MPIE 域的值总是完全一样， mstatus.MPP 域的值与 mcause.MPP 域的值总是完全一样。

7.4.12. mstatus 的 FS 域

`mstatus` 寄存器中的 **FS** 域用于维护或反映浮点单元的状态。

注意：此域只有在配置了浮点指令（“F”或者“D”指令子集）时才会存在。

FS 域由两位组成，其编码如下图所示。

Status	FS Meaning	XS Meaning
0	Off	All off
1	Initial	None dirty or clean, some on
2	Clean	None dirty, some clean
3	Dirty	Some dirty

图 7-2 **FS** 域表示的状态编码

FS 域的更新准则如下：

- **FS** 上电后的默认值为 **0**，意味着浮点单元的状态为 **Off**。因此为了能够正常使用浮点单元，软件需要使用 **CSR** 写指令将 **FS** 的值改写为非 **0** 值以打开浮点单元（**FPU**）的功能。
- 如果 **FS** 的值为 **1** 或者 **2**，当执行了任何的浮点指令之后，**FS** 的值会自动切换为 **3**，表示浮点单元的状态为 **Dirty**（状态发生了改变）。
- 如果处理器不想使用浮点运算单元（譬如将浮点单元关电以节省功耗），可以使用 **CSR** 写指令将 `mstatus` 寄存器的 **FS** 域设置成 **0**，将浮点单元的功能予以关闭。当浮点单元的功能关闭之后，任何访问浮点 **CSR** 寄存器的操作或者任何执行浮点指令的行为都将会产生非法指令（**Illegal Instruction**）异常。

除了用于上述功能，**FS** 域的值还用于操作系统在进行上下文切换时的指引信息，感兴趣的用户请参见 **RISC-V “特权架构文档版本 1.11”** 原文。

7.4.13. `mstatus` 的 **XS** 域

`mstatus` 寄存器中的 **XS** 域与 **FS** 域的作用类似，但是其用于维护或反映用户自定义的扩展指令单元状态。

在标准的 **RISC-V “特权架构文档版本 1.11”** 中定义 **XS** 域为只读域，其用于反映所有自定义扩

展指令单元的状态总和。但请注意：在 N 级别处理器内核的硬件实现中，将 XS 域设计成可写可读域，其作用完全与 FS 域类似，软件可以通过改写 XS 域的值达到打开或者关闭协处理器扩展指令单元的目的。

与 FS 域类似，XS 除了用于上述功能之外还用于操作系统在进行上下文切换时的指引信息，感兴趣的用户请参见 RISC-V “特权架构文档版本 1.11” 原文。

7.4.14. mstatus 的 MPRV 域

mstatus 寄存器中的 MPRV 域用于控制在 Machine Mode 下存储器的数据读写 (Load 和 Store) 操作是否被当作 User Mode 下的操作来进行 PMP 的保护。

如果 MPRV 域的值为 1，并且 MPP 域的值表示 User Mode，则 PMP 将存储器的数据读写 (Load 和 Store) 操作被当作发生在 User Mode 下来进行保护，并根据表项配置的 R/W 属性进行权限确认，如果违反了权限，则触发异常。请参见 8.4 节了解其详情。

7.4.15. mstatus 的 SUM 域

mstatus 寄存器中的 SUM 域用于控制在 Supervisor Mode 下是否被允许读写 (Load 和 Store) 用户 (User) 存储区域的数据。

7.4.16. mstatus 的 SD 域

mstatus 寄存器中的 SD 域是一个只读域，其反映了 XS 域或者 FS 域处于脏 (Dirty) 状态。其逻辑关系表达式为： $SD = ((FS == 11) \text{ OR } (XS == 11))$ 。

之所以设置此只读的 SD 域是为了方便软件快速的查询 XS 域或者 FS 域是否处于脏 (Dirty) 状态，从而在上下文切换时可以快速判断是否需要浮点单元或者扩展指令单元进行上下文的保存。感兴趣的用户请参见 RISC-V “特权架构文档版本 1.11” 原文。

7.4.17. mtvec

mtvec 寄存器用于配置中断和异常处理程序的入口地址。

■ 当 mtvec 配置中断的异常处理程序入口地址时要点如下：

- 异常处理程序采用 4byte 对齐的 mtvec 地址（将 mtvec 的低 2bit 用 0 代替）作为入口地址。

■ 当 mtvec 配置中断程序的入口地址时要点如下：

- 当 mtvec.MODE != 6'b0000011 时，处理器使用“默认中断模式”。
- 当 mtvec.MODE = 6'b0000011 时，处理器使用“ECLIC 中断模式”，推荐使用此模式。
 - ◆ 中断为非向量处理模式时的入口地址和要点如第 5.13.2 节中所述。
 - ◆ 中断为向量处理模式时的入口地址和要点如第 5.13.1 节中所述。

mtvec 寄存器各地址位域如表 7-3 所示。

表 7-3 mtvec 寄存器各控制位

域	位	描述
ADDR	31:6	mtvec 地址
MODE	5: 0	■ MODE 域为中断处理模式控制域： <ul style="list-style-type: none"> ● 0000011: ECLIC 中断模式（推荐模式） ● Others: 默认中断模式

7.4.18. mtvt

mtvt 寄存器用于保存 ECLIC 中断向量表的基地址，此基地址至少为 64byte 对齐。

为了提升性能减少硬件门数，硬件根据实际实现的中断的个数来决定 mtvt 的对齐方式，具体如表 7-4 所示。

表 7-4 mtvt 对齐方式

最大中断个数	mtvt 对齐方式
--------	-----------

0 to 16	64-byte
17 to 32	128-byte
33 to 64	256-byte
65 to 128	512-byte
129 to 256	1KB
257 to 512	2KB
513 to 1024	4KB

7.4.19. mscratch

mscratch 寄存器用于 Machine Mode 下的程序临时保存某些数据。mscratch 寄存器可以提供一种保存和恢复机制，譬如，在进入中断或者异常处理模式后，将应用程序的用户堆栈指针（SP）寄存器临时存入 mscratch 寄存器中，然后在退出异常处理程序之前，将 mscratch 寄存器中的值读出恢复至用户堆栈指针（SP）寄存器。

7.4.20. mepc

mepc 寄存器用于保存进入异常之前处理器正在执行指令的 PC 值，作为异常的返回地址。

为了理解此寄存器，请先参见第 3 章系统地了解异常的相关信息。

注意：

- 处理器进入异常时，mepc 寄存器被同时更新以反映当前遇到异常的指令的 PC 值。
- 值得注意的是，虽然 mepc 寄存器会在异常发生时自动被硬件更新，但是 mepc 寄存器本身也是一个（在 Machine Mode 下）可读可写的寄存器，因此软件也可以直接写该寄存器以修改它的值。

mepc 寄存器各地址位域如表 7-5 所示。

表 7-5 mepc 寄存器各控制位

域	位	描述
EPC	31: 1	保存异常发生前处理器正在执行的指令的 PC 值
Reserved	0	未使用的域为常数 0

7.4.21. mcause

mcause 寄存器，用于保存进入 NMI、异常和中断之前的出错原因，以便于对 Trap 原因进行诊断和调试。

mcause 寄存器各地址位域如表 7-6 所示。

表 7-6 mcause 寄存器各控制位

域	位	描述
INTERRUPT	31	表示当前 Trap 种类： ■ 0: 异常或者 NMI ■ 1: 中断
MINHV	30	表示处理器正在读取中断向量表
MPP	29:28	进入中断之前的特权模式，与 mstatus.mpp 相同
MPIE	27	进入中断之前的中断使能，与 mstatus.mpie 相同
Reserved	26:24	未使用的域为常数 0
MPIL	23:16	前一个中断级别
Reserved	15:12	未使用的域为常数 0
EXCCODE	11:0	异常/中断编码

注意：

- mstatus 寄存器的 MPIE 和 MPP 域与 mcause 的 MPIE 和 MPP 域为镜像关系。
- NMI 的 mcause.EXCCODE 可能为 0x1 或者 0xffff，实际值由 mmisc_ctl 控制，详情请参考 7.5.8 节。

7.4.22. mtval (mbadaddr)

mtval 寄存器（又名 mbadaddr，有些版本的工具链只识别此名称），用于保存进入异常之前的出错指令的编码值或者存储器访问的地址值，以便于对异常原因进行诊断和调试。

为了理解此寄存器，请先参见第 3 章系统地了解异常的相关信息。

N 级别处理器内核进入异常时，**mtval** 寄存器被同时更新以反映当前遇到异常的信息。

7.4.23. mip

ECLIC 中断模式下 **mip** 寄存器的控制位不起作用，读 **mip** 返回全 0。

7.4.24. mnxti

mnxti（Next Interrupt Handler Address and Interrupt-Enable CSR）可以被软件访问用来处理处于相同 **Privilege Mode** 下的下一个中断，同时不会造成冲刷流水线以及上下文保存恢复。

mnxti 寄存器可通过 **CSRRSI/CSRRCI** 指令来访问，读操作返回值是下一个中断的 **handler** 地址，而 **mnxti** 的写回操作会更新中断使能的状态。

注意：

1. 对于不同 **Privilege Mode** 的中断，硬件会以中断嵌套的方式处理，因此 **mnxti** 只会处理相同 **Privilege Mode** 下的下一个中断。
2. **mnxti** 寄存器与常规的 **CSR** 指令不一样，其返回值与常规寄存器的 **RMW**（read-modify-write）操作的值不同：
 - **mnxti** 的 **CSR** 读操作的返回值有以下两种情况：
 - ◆ 当出现以下情况时，返回值为 0。
 - 没有可以响应的中断
 - 当下最高优先级的中断是向量中断
 - ◆ 当中断为非向量中断时，返回此中断的中断处理程序入口地址。
 - **mnxti** 的 **CSR** 写操作会更新以下寄存器及寄存器域：
 - ◆ **mstatus** 是当前 **RMW**（read-modify-write）操作的目的寄存器
 - ◆ **mcause.EXCCODE** 域和会被分别更新为当前响应中断的中断 **id**

- ◆ `mintstatus.MIL` 域被更新为当前响应中断的中断级别（Level）。

7.4.25. `mintstatus`

`mintstatus` 寄存器保存每个 **Privilege Mode** 下的有效中断的中断 **level**。注意，写此寄存器会被忽略。

表 7-7 `mintstatus` 寄存器的控制位

域	位	描述
MIL	31:24	Machine Mode 的有效中断 level
Reserved	23: 8	未使用的域为常数 0
UIL	7:0	User Mode 的有效中断 level

7.4.26. `mscratchcsw`

`mscratchcsw` 寄存器用于在不同特权模式间切换时，交换目的寄存器与 `mscratch` 的值来加速中断处理。

使用带读操作的 CSR 指令访问 `mscratchcsw`，在出现中断前后特权模式不一致时，有以下伪指令所示的寄存器操作：

```
csrrw rd, mscratchcsw, rs1

// Pseudocode operation.
if (mcause.mpp!=M-mode) then {
    t = rs1; rd = mscratch; mscratch = t;
} else {
    rd = rs1; // mscratch unchanged.
}

// Usual use: csrrw sp, mscratchcsw, sp
```

处理器在低特权模式（**Privilege Mode**）时发生中断，处理器进入高特权模式处理中断，在处理中断时，需要使用堆栈来保存进入中断前的处理器状态。此时如果继续使用低特权模式下的堆栈指针（**SP**），则高特权模式下堆栈的数据会存储在低特权模式可以访问的区间，导致出现高特权模

式的数据泄露给低特权模式这一安全漏洞。为避免此安全漏洞，RISC-V 架构规定当处理器处于低特权模式时，需要将高特权模式的堆栈指针（SP）保存至 `mscratch` 寄存器，这样在进入高特权模式后，处理器可以用 `mscratch` 寄存器的值来恢复高特权模式的堆栈指针（SP）。

使用常规指令来执行以上的程序需要耗费较多的 `cycle`，为此 RISC-V 架构定义 `mscratchcs` 寄存器，在进入中断后立刻执行 `mscratchcs` 寄存器指令，交换 `mscratch` 与 SP 的值，用来恢复高特权模式的堆栈指针（SP），同时备份低特权模式的堆栈指针（SP）至 `mscratch` 寄存器。在执行 `mret` 指令退出中断前，也加上一条 `mscratchcs` 指令，交换 `mscratch` 寄存器和堆栈指针（SP）的值，将高特权模式的堆栈指针（SP）再次备份到 `mscratch`，同时恢复低特权模式的堆栈指针（SP）。这样，只需要两条指令便可以解决不同特权模式的堆栈指针（SP）切换问题，加速了中断处理。

注意：为了避免虚拟化的漏洞，软件不能直接读取处理器当前的特权模式（Privilege Mode）。如果软件试图在更低的特权模式下访问给定特权模式下的 `mscratchcs` 做寄存器 `swap` 操作会导致处理器进入 `Trap`，因此 `mscratchcs` 不会导致虚拟化漏洞。

7.4.27. mscratchcs

`mscratchcs` 寄存器用于支持同一特权模式下出现中断程序与应用程序切换时，快速交换堆栈指针。

使用带读操作的 CSR 指令访问 `mscratchcs`，当特权模式不变，在出现中断程序和应用程序的切换时，有以下伪指令所示的寄存器操作：

```
csrrw rd, mscratchcs, rs1

// Pseudocode operation.
if ( (mcause.mpi==0) != (mintstatus.mil == 0) ) then {
    t = rs1; rd = mscratch; mscratch = t;
} else {
    rd = rs1; // mscratch unchanged.
}

// Usual use: csrrw sp, mscratchcs, sp
```

在单一特权模式下，将中断处理程序任务与应用程序任务的堆栈空间分离可以增强健壮性、减少空间使用并有助于系统调试。中断处理程序任务具有非零中断级别，而应用程序任务具有零中断级别，根据这一特性 RISC-V 架构定义了 `mscratchcs` 寄存器。与 `mscratchcs` 类似，在中断程

序入口和出口分别添加一条 `mscratchcswl` 可以实现中断处理程序与应用程序之间的快速的堆栈指针切换，保证中断处理程序和应用程序的堆栈空间分离。

7.4.28. `mcycle` 和 `mcycleh`

RISC-V 架构定义了一个 64 位宽的时钟周期计数器，用于反映处理器执行了多少个时钟周期。只要处理器处于执行状态时，此计数器便会不断自增计数。

`mcycle` 寄存器反映了该计数器低 32 位的值，`mcycleh` 寄存器反映了该计数器高 32 位的值。

`mcycle` 和 `mcycleh` 寄存器可以用于衡量处理器的性能，且具备可读可写属性，因此软件可以通过 CSR 指令改写 `mcycle` 和 `mcycleh` 寄存器中的值。

由于考虑到此计数器计数会消耗某些动态功耗，因此在 N 级别处理器的实现中，在自定义 CSR 寄存器 `mcountinhibit` 中额外增加了一位控制域，软件可以配置此控制域将 `mcycle` 和 `mcycleh` 对应的计数器停止计数，从而在不需衡量性能之时停止计数器以达到省电的作用。请参见 7.5.1 节了解更多 `mcountinhibit` 寄存器信息。

注意：如果在调试模式下时，此计数器并不会计数，只有在正常功能模式下，计数器才会进行计数。

7.4.29. `minstret` 和 `minstreth`

RISC-V 架构定义了一个 64 位宽的指令完成计数器，用于反映处理器成功执行了多少条指令。只要处理器每成功执行完成一条指令，此计数器便会自增计数。

`minstret` 寄存器反映了该计数器低 32 位的值，`minstreth` 寄存器反映了该计数器高 32 位的值。

`minstret` 和 `minstreth` 寄存器可以用于衡量处理器的性能，且具备可读可写属性，因此软件可以通过 CSR 指令改写 `minstret` 和 `minstreth` 寄存器中的值。

由于考虑到此计数器计数会消耗某些动态功耗，因此在 N 级别处理器内核的实现中，在自定义的 CSR 寄存器 `mcountinhibit` 中额外增加了一位控制域，软件可以配置此控制域将 `minstret` 和 `minstreth` 对应的计数器停止计数，从而在不需衡量性能之时停止计数器以达到省电的作用。请

参见 7.5.1 节了解更多 `mcountinhibit` 寄存器信息。

注意：如果在调试模式下时，此计数器并不会计数，只有在正常功能模式下，计数器才会进行计数。

7.4.30. `mtime`、`mtimecmp`、`msip` 和 `mtimectl`

RISC-V 架构定义了一个 64 位的计时器，按照系统的低速实时时钟（Real Time Clock）频率进行计时。该计时器的值实时反映在 `mtime` 寄存器中。RISC-V 架构还定义了一个 64 位的 `mtimecmp` 寄存器，该寄存器作为计时器的比较值，假设计时器的值 `mtime` 大于或者等于 `mtimecmp` 的值，则产生计时器中断。注意：RISC-V 架构没有将 `mtime` 和 `mtimecmp` 寄存器定义为 CSR 寄存器，而是定义为存储器地址映射（Memory Address Mapped）的系统寄存器，具体的存储器映射地址 RISC-V 架构并没有规定，而是交由内核设计者自行实现。

RISC-V 架构定义了一种软件中断，可以通过软件写 1 至 `msip` 寄存器来触发。有关软件中断的信息请参见 6.1.6 节。注意：RISC-V 架构并没有定义 `msip` 寄存器为 CSR 寄存器，而定义为存储器地址映射的系统寄存器，具体的存储器映射地址 RISC-V 架构并没有规定，而是交由内核设计者自行实现。

在 N 级别处理器内核的实现中，`mtime`/`mtimecmp`/`msip`/`mtimectl` 均由 TMR 单元实现，有关 TMR 实现要点以及 `mtime`/`mtimecmp`/`msip`/`mtimectl` 分配的存储器地址区间，请参见各个系列的简明数据手册。

注意：由于 `mtime`/`mtimecmp`/`msip`/`mtimectl` 的地址是存储器地址空间，因此其访问权限由 PMP 进行设定和保护。

由于考虑到计时器计数会消耗某些动态功耗，因此在 N 级别处理器的实现中，在自定义 `mtimectl` 寄存器中额外增加了一位控制域，软件可以配置此控制域将 `mtime` 对应的计时器停止计数，从而在不需要之时停止计时器达到省电的作用。

注意：如果在调试模式下时，此计数器并不会计数，只有在正常功能模式下，计数器才会进行计数。

7.4.31. fcsr

此寄存器只有在配置了浮点指令（“F”或者“D”指令子集）时才会存在。

RISC-V 架构规定，如果支持单精度浮点指令或者双精度浮点指令，则需要增加一个浮点控制状态寄存器。该寄存器包含了浮点异常标志位域（Accrued Exceptions）和浮点舍入模式（Rounding Mode）域。

fcsr 寄存器中各控制位域如表 7-8 所示。

表 7-8 fcsr 寄存器各控制位

域		位	描述
Reserved		31:8	未使用的域为常数 0
Rounding Mode (frm)		7:5	浮点舍入模式
Accrued Exceptions (fflags)	NV	4	非法操作（Invalid Operation）异常标志位
	DZ	3	除零（Divide by Zero）异常标志位
	OF	2	上溢出（Overflow）异常标志位
	UF	1	下溢出（Underflow）异常标志位
	NX	0	非精确（Inexact）异常标志位

7.4.32. fflags

此寄存器只有在配置了浮点指令（“F”或者“D”指令子集）时才会存在。

fflags 寄存器为浮点控制状态寄存器（fcsr）中浮点异常标志位（Accrued Exceptions）域的别名。之所以单独定义一个 fflags 寄存器，是为了方便使用 CSR 指令直接地单独读写浮点异常标志位域。

如表 7-8 所示，fcsr 寄存器包含浮点异常标志域（fflags），不同的异常标志位所表示的异常类型。如果浮点运算单元在运算中出现了相应的异常，则会将 fcsr 寄存器中对应的异常标志位设置为高，且会一直保持累积。软件可以通过写 0 的方式单独清除某个异常标志位。

注意：在很多处理器架构中，浮点运算产生结果异常都会触发异常跳转（Trap）从而进入异常模式。但是 RISC-V 架构的浮点指令在产生结果异常时并不会跳转进入异常模式，而是如上所述仅设置 fcsr 寄存器中的异常标志位。

7.4.33. frm

此寄存器只有在配置了浮点指令（“F” 或者 “D” 指令子集）时才会存在。

frm 寄存器为浮点控制状态寄存器中浮点舍入模式（Rounding Mode）域的别名。之所以单独定义一个 frm 寄存器，是为了方便使用 CSR 指令直接地单独读写浮点舍入模式。

根据 IEEE-754 标准，浮点数运算需要指定舍入模式（Rounding Mode）， RISC-V 架构浮点运算的舍入模式可以通过两种方式指定。

- 静态舍入模式：浮点指令的编码中有 3 位作为舍入模式域，有关浮点指令列表以及指令编码请参见“指令集文档版本 2.2”（riscv-spec-v2.2.pdf）。RISC-V 架构支持如
 -
- 表 7-9 所示的五种合法的舍入模式。
 - 如果舍入模式编码为 101 或者 110，则为非法模式。
 - 如果舍入模式编码为 111，则意味着使用动态舍入模式。
- 动态舍入模式：如果使用动态舍入模式，则使用 fcsr 寄存器中的舍入模式域。如表 7-8 所示，fcsr 寄存器包含舍入模式域。不同的舍入模式编码同样如
 -
- 表 7-9 所示，仅支持物种合法的舍入模式。如果 fcsr 寄存器中的舍入模式域指定为非法的舍入模式，则后续浮点指令会产生非法指令异常。

表 7-9 舍入模式位

舍入模式编码	舍入模式（Rounding Mode）	描述
000	RNE	最近舍入，朝向偶数方向（round to nearest, ties to even）
001	RTZ	朝零舍入（round towards zero）
010	RDN	向下舍入（round down）
011	RUP	向上舍入（round up）
100	RMM	最近舍入，朝向最大幅度方向（round to nearest, ties to max magnitude）
101		非法值

110		非法值
111		动态舍入模式

7.4.34. cycle 和 cycleh

cycle 和 cycleh 分别是 mcycle 和 mcycleh 的只读副本。该寄存器在 User Mode 下是否可读由 CSR 寄存器 mcounteren 的 CY 比特域来控制，请参见第 7.4.37 节了解其详情。

7.4.35. instret 和 instreth

instret 和 instreth 分别是 minstret 和 minstreth 的只读副本。该寄存器在 User Mode 下是否可读由 CSR 寄存器 mcounteren 的 IR 比特域来控制，请参见第 7.4.37 节了解其详情。

7.4.36. time 和 timeh

time 和 timeh 分别是 mtime 和 mtimeh 的只读副本。该寄存器在 User Mode 下是否可读由 CSR 寄存器 mcounteren 的 TM 比特域来控制，请参见第 7.4.37 节了解其详情。

7.4.37. mcounteren

该寄存器只有在支持 User Mode 的配置下才会存在。mcounteren 寄存器中各控制位域如表 7-10 所示。

表 7-10 mcounteren 寄存器各控制位

域	位	描述
CY	0	此位控制在 User Mode 下是否能够访问 cycle 和 cycleh 寄存器： <ul style="list-style-type: none"> 如果此位为 1，则在 User Mode 下能够正常访问 cycle 和 cycleh。 如果此位为 0，则在 User Mode 下访问 cycle 和 cycleh 会触发 illegal instruction exception。 此位复位默认值为 0
TM	1	此位控制在 User Mode 下是否能够访问 time 和 timeh 寄存器： <ul style="list-style-type: none"> 如果此位为 1，则在 User Mode 下能够正常访问 time 和 timeh。

		<ul style="list-style-type: none"> 如果此位为 0，则在 User Mode 下访问 time 和 timeh 会触发 illegal instruction exception。 此位复位默认值为 0
IR	2	此位控制在 User Mode 下是否能够访问 instret 和 instreth 寄存器： <ul style="list-style-type: none"> 如果此位为 1，则在 User Mode 下能够正常访问 instret 和 instreth。 如果此位为 0，则在 User Mode 下访问 instret 和 instreth 会触发 illegal instruction exception。 此位复位默认值为 0
Reserved	3~31	其他未使用的域为常数 0

7.4.38. pmpcfg<x>寄存器

pmpcfg<x>寄存器用于指定 PMP 表项的权限配置寄存器，每个 pmpcfg<x>寄存器可以管理四个 PMP 表项目，譬如 pmpcfg0 包含了四个不同的域（pmp0cfg~pmp4cfg），分别管理表项 0 到表项 4。有关 pmpcfg<x>寄存器的详细信息请参见 8.2.2 节。

7.4.39. pmpaddr<x>寄存器

pmpaddr<x>寄存器用于指定 PMP 每个表项的地址配置寄存器。有关 pmpaddr<x>寄存器的详细信息请参见 8.2.3 节。

7.4.40. ucode

此寄存器只有配置了数字信号处理单元（“P”扩展）才存在。

ucode 寄存器主要用来记录 DSP 运算的结果是否有溢出，如果有溢出则将溢出标志位置 1，可以通过软件写入的方式将溢出标志位清 0。ucode 寄存器各控制位域如表 7-11 所示。

表 7-11 ucode 寄存器各控制位

域	位	描述
OV	0	DSP 溢出标志位，如果有溢出则将溢出标志位置 1，写 0 可清除。
Reserved	31:1	未使用的域为常数 0

7.5. N 级别处理器内核自定义的 CSR

本节介绍 N 级别处理器核自定义的 CSR 寄存器。

7.5.1. mcountinhibit

mcountinhibit 寄存器用于控制 mcycle 和 minstret 的计数，各控制位域如表 7-12 所示。

表 7-12 mcountinhibit 寄存器各控制位

域	位	描述
Reserved	31:3	未使用的域为常数 0
IR	2	IR 为 1 时 minstret 的计数被关闭
Reserved	1	未使用的域为常数 0
CY	0	CY 为 1 时 mcycle 的计数被关闭

7.5.2. milm_ctl

mil_mctl 寄存器用于打开或关闭 ILM 地址空间，用户可根据具体的应用场景来进行相应的配置。

表 7-13 mil_mctl 寄存器

域	位	描述
ILM_BPA	XLEN-1:10 (只读)	ILM 的基址，需要对齐于 ILM 的大小（整数倍）； 例如，基址为 12KB (0x3000)，大小为 4KB，则 ILM_BPA 为 0xC (高 22 位)。
Reserved	9:4	预留位，0
ILM_RWECC	3	用于控制将 mecc_code 写入 ILM，可用于模拟错误注入： 0：关闭由 mecc_code 注入 ILM； 1：打开由 mecc_code 注入 ILM；
ILM_ECC_EXCP_EN	2	0：关闭 2 比特 ILM ECC 例外； 1：打开 2 比特 ILM ECC 例外；
ILM_ECC_EN	1	0：关闭 ILM ECC；（默认） 1：打开 ILM ECC；
ILM_EN	0 (UX 可写)	1：打开 ILM (默认)； 0：关闭 ILM；

注意：ILM 只有在 UX 级别且配置了 MMU 和 ILM 的情况下才能被关闭，该文档主要描述 N 级别指令集架构，所以 ILM_EN 位在 N 级别内核里默认打开，不可关闭，UX 系列指令集文档请参考 “Nuclei RISC-V ISA SPEC”。

7.5.3. mdlm_ctl

mdlm_ctl 寄存器用于打开或关闭 DLM 地址空间，用户可根据具体的应用场景来进行相应的配置。

表 7-14 mdlm_ctl 寄存器

域	位	描述
DLM_BPA	XLEN-1:10 (只读)	DLM 的基址，需要对齐于 DLM 的大小（整数倍）； 例如，基址为 12KB (0x3000)，大小为 4KB，则 DLM_BPA 为 0xC (高 22 位)。
Reserved	31:1	预留位，0
DLM_RWECC	3	用于控制将 mecc_code 写入 DLM，可用于模拟错误注入： 0：关闭由 mecc_code 注入 DLM； 1：打开由 mecc_code 注入 DLM；
DLM_ECC_EXCP_EN	2	0：关闭 2 比特 DLM ECC 例外； 1：打开 2 比特 DLM ECC 例外；
DLM_ECC_EN	1	0：关闭 DLM ECC；（默认） 1：打开 DLM ECC；
DLM_EN	0 (UX 可写)	1：打开 DLM (默认)； 0：关闭 DLM；

注意 DILM 只有在 UX 级别且配置了 MMU 和 DLM 的情况下才能被关闭，该文档主要描述 N 级别指令集架构，所以 DLM_EN 位在 N 级别内核里默认打开，不可关闭，UX 系列指令集文档请参考 “Nuclei RISC-V ISA SPEC”。

7.5.4. mnvec

mnvec 寄存器用于配置 NMI 的入口地址，写此寄存器会被忽略。

为了理解此寄存器，请先参见第 4 章系统地了解 NMI 的相关信息。

在处理器的程序执行过程中，一旦遇到 NMI 发生，则终止当前的程序流，处理器被强行跳转到一个新的 PC 地址，N 级别处理器内核进入 NMI 后跳入的 PC 地址即由 mnvec 寄存器指定。

注意：mnvec 的值由 mmisc_ctl 控制，更多细节请参考 7.5.8 节。

7.5.5. msubm

N 级别处理器内核自定义 msubm 寄存器用于保存进入 Trap 前后的 Trap 类型。

msubm 寄存器中各控制位域如表 7-15 所示。

表 7-15 msubm 寄存器各控制位

域	位	描述
Reserved	31:10	未使用的域为常数 0
PTYP	9:8	保存进入 Trap 之前的 Trap 类型： <div> <div>0: 非 Trap 状态</div> <div>1: 中断</div> <div>2: 异常</div> <div>3: NMI</div> </div>
TYP	7:6	指示 Core 当前的 Trap 类型： <div> <div>0: 非 Trap 状态</div> <div>1: 中断</div> <div>2: 异常</div> <div>3: NMI</div> </div>
Reserved	5:0	未使用的域为常数 0

7.5.6. mdcause

mdcause 主要用来提供更加详细的异常信息，以区别具有相同 mcause 值的不同异常。

mdcause 寄存器中各控制位域如表 7-16 所示。

表 7-16 mdcause 寄存器各控制位

域	位	描述
Reserved	31: 2	未使用的域为常数 0
mdcause	1:0	更详细的异常信息，以区别具有相同 mcause 的不同异常。 当 mcause=1 (指令访问错误): <div> <div>0: Reserved</div> </div>

		<ul style="list-style-type: none"> ■ 1: PMP 检测指令访问出错 ■ 2: 指令访问返回总线错误 ■ 3: Reserved <p>当 mcause=5(读取存储器访问错误)</p> <ul style="list-style-type: none"> ■ 0: Reserved ■ 1: PMP 检测读操作访问存储器出错 ■ 2: 读操作访问存储器返回总线错误 ■ 3: NICE 长指令错误 <p>当 mcause=7(写入存储器访问错误)</p> <ul style="list-style-type: none"> ■ 0: Reserved ■ 1: PMP 检测写操作访问存储器出错 ■ 2: 写操作访问存储器返回总线错误 ■ 3: Reserved
--	--	---

7.5.7. mcache_ctl

此寄存器只有配置了 I-CACHE 才会存在。

mcache_ctl 主要用来控制 I-CACHE 的开启关闭以及 Scratchpad 的模式。具体的控制位域如表 7-17 所示。

表 7-17 mcache_ctl 寄存器各控制位

域	位	描述
Reserved	31: 21	未使用的域为常数 0
DC_RWDECC	20	控制 D-Cache Data Ram 的 ECC code 注入, ECC code 注入寄存器是 mecc_code, 该特性可以用来进行 ECC 错误注入: 0: 关闭 D-Cache Data Ram 的 ECC code 注入 1: 打开 D-Cache Data Ram 的 ECC code 注入
DC_RWTECC	19	控制 D-Cache Tag Ram 的 ECC code 注入, ECC code 注入寄存器是 mecc_code, 该特性可以用来进行 ECC 错误注入: 0: 关闭 D-Cache Tag Ram 的 ECC code 注入 1: 打开 D-Cache Tag Ram 的 ECC code 注入
DC_ECC_EXCP_EN	18	D-Cache 2 比特 ECC 错误异常的开关: 0: 关闭 D-Cache 2 比特 ECC 错误异常 1: 打开 D-Cache 2 比特 ECC 错误异常
DC_ECC_EN	17	D-Cache ECC 开关: 0: 关闭 ECC (默认) 1: 打开 ECC
DC_EN	16	0: D-Cache 关闭, 默认值; 1: D-Cache 打开;
Reserved	15:6	未使用的域为常数 0
IC_RWDECC	5	控制 I-Cache Data Ram 的 ECC code 注入, ECC code 注入寄存器是 mecc_code, 该特性可以用来进行 ECC 错误注入:

		0: 关闭 I-Cache Data Ram 的 ECC code 注入 1: 打开 I-Cache Data Ram 的 ECC code 注入
IC_RWTECC	4	控制 I-Cache Tag Ram 的 ECC code 注入, ECC code 注入寄存器是 mecc_code, 该特性可以用来进行 ECC 错误注入: 0: 关闭 I-Cache Tag Ram 的 ECC code 注入 1: 打开 I-Cache Tag Ram 的 ECC code 注入
IC_ECC_EXCP_EN	3	I-Cache 2 比特 ECC 错误异常的开关: 0: 关闭 I-Cache 2 比特 ECC 错误异常 1: 打开 I-Cache 2 比特 ECC 错误异常
IC_ECC_EN	2	I-Cache ECC 开关: 0: 关闭 ECC (默认) 1: 打开 ECC
IC_SCPD_MOD	1	Scratchpad 的模式: 0: Scratchpad 作为 ICache 的 Data Ram 1: Scratchpad 作为 ILM 的 SRAM, 默认值
IC_EN	0	I-CACHE 使能: 0: I-CACHE 关闭, 默认值。 1: I-CACHE 开启。

注意: 只有在 mcache_ctl[1:0]为 2'b10 时(复位后的默认值), Scratchpad 作为 ILM 的 SRAM, 否则 Scratchpad 是被当作 ICache 的 Data Ram 使用。

7.5.8. mmisc_ctl

N 级别处理器内核自定义 mmisc_ctl 寄存器用于控制 NMI, Misaligned Access 和 BPU 的相关功能。

mmisc_ctl 寄存器中各控制位域如表 7-18 所示。

表 7-18 mmisc_ctl 寄存器各控制位

域	位	描述
Reserved	31:10	未使用的域为常数 0
NMI_CAUSE_FFF	9	控制 mnvec 及 NMI 的 mcause.EXCCODE: ■ 0: mnvec 的值等于处理器 reset 后的 PC, NMI 的 mcause.EXCCODE 为 0x1, 此为默认值。 ■ 1: mnvec 的值与 mtvec 一致, NMI 的 mcause.EXCCODE 为 0xffff
Reserved	8:7	未使用的域为常数 0
MISALIGN	6	控制内核是否支持 Misaligned Access 功能: ■ 0: Misaligned Access 功能关闭, Misaligned Access 操作会产生异常 ■ 1: Misaligned Access 功能开启, 此为默认值 注意: 此域只有配置了非对齐数据访问功能才有效, 否则为常数 0

Reserved	5:4	未使用的域为常数 0
BPU	3	控制分支预测器是否开启： ■ 0: 分支预测器关闭 ■ 1: 分支预测器开启，此为默认值 注意：此域只有配置了分支预测器才有效，否则为常数 0
Reserved	2:0	未使用的域为常数 0

7.5.9. msavestatus

msavestatus 用于存储 mstatus 和 msubm 的值，以保证 mstatus 和 msubm 的各个域的状态不会被 NMI 或者异常冲刷掉。msavestatus 有两级堆栈，最多可支持 3 级异常/NMI 状态保存。更多两级 NMI/异常状态堆栈，请参见 4.6 节。

msavestatus 寄存器各控制位如表 7-19 所示。

表 7-19 msavestatus 寄存器各控制位

域	位	描述
Reserved	31:16	未使用的域为常数 0
PTYP2	15:14	第二级嵌套 NMI/异常发生前的 Trap 类型
Reserved	13:11	未使用的域为常数 0
MPP2	10:9	第二级嵌套 NMI/异常发生前的 Privilege mode
MPIE2	8	第二级嵌套 NMI/异常发生前的中断使能状态
PTYP1	7:6	第一级嵌套 NMI/异常发生前的 Trap 类型
Reserved	5:3	未使用的域为常数 0
MPP1	2:1	第一级嵌套 NMI/异常发生前的 Privilege mode
MPIE1	0	第一级嵌套 NMI/异常发生前的中断使能状态

7.5.10. msaveepc1 和 msaveepc2

msaveepc1 和 msaveepc2 分别作为一级 NMI/异常状态堆栈和二级 NMI/异常状态堆栈，用来存储第一级嵌套 NMI/异常发生前的 PC，以及第二级嵌套 NMI/异常发生前的 PC。

■ $\text{msaveepc2} \leq \text{msaveepc1} \leq \text{mepc} \leq \text{interrupted PC} \leq \text{NMI/exception PC}$

当执行 mret 指令，同时 mcause.INTERRUPT 为 0（例如 NMI，或者异常），msaveepc1 和 msaveepc2 分别通过一级和两级 NMI/异常状态堆栈来恢复处理器的 PC。

- `msaveepc2 => msaveepc1 => mepc => PC`

7.5.11. msavecause1 和 msavecause2

`msavecause1` 和 `msavecause2` 分别作为一级 NMI/异常状态堆栈和二级 NMI/异常状态堆栈，用来存储第一级嵌套 NMI/异常发生前的 `mcause`，以及第二级嵌套 NMI/异常发生前的 `mcause`。

- `msavecause2 <= msavecause1 <= mcause <= NMI/exception cause`

当执行 `mret` 指令，同时 `mcause.INTERRUPT` 为 0（例如 NMI，或者异常），`msavecause1` 和 `msavecause2` 分别通过一级和两级 NMI/异常状态堆栈来恢复 `mcause` 状态。

- `msavecause2 => msavecause1 => mcause`

7.5.12. msavedcause1 和 msavedcause2

`msavedcause1` 和 `msavedcause2` 分别作为一级 NMI/异常状态堆栈和二级 NMI/异常状态堆栈，用来存储第一级嵌套 NMI/异常发生前的 `mdcause`，以及第二级 NMI/异常发生前的 `mdcause`。

- `msavedcause2 <= msavedcause1 <= mdcause <= exception dcause`

当执行 `mret` 指令，同时 `mcause.INTERRUPT` 为 0（例如 NMI，或者异常），`msavedcause1` 和 `msavedcause2` 分别通过一级和两级 NMI/异常状态堆栈来恢复 `mcause` 状态。

- `msavedcause2 => msavedcause1 => mdcause`

7.5.13. pushmsubm

N 级别处理器定义了通过 `pushmsubm` 寄存器 `csrrwi` 操作实现的 CSR 指令，存储 `msubm` 的值到堆栈指针作为基地址的 memory 空间。

以如下指令为例介绍此 CSR 指令：

```
csrrwi x0, PUSHMSUBM, 1
```

该指令的操作是将 `msubm` 寄存器的值存到 `SP`（堆栈指针）+1*4 的地址。

7.5.14. mvt2

mtvt2 用于指定 ECLIC 非向量模式的中断 common-code 入口地址。

mtvt2 寄存器中各控制位域如表 7-20 所示。

表 7-20 mvt2 寄存器各控制位

域	位	描述
COMMON-CODE-ENTRY	31:2	在 mvt2.MTVT2EN=1 时，此域决定 ECLIC 非向量模式中断 common-code 入口地址。
Reserved	1	未使用的域为常数 0
MTVT2EN	0	mtvt2 使能位： <div> <div>■ 0: ECLIC 非向量模式中断 common-code 入口地址由 mtvec 决定</div> <div>■ 1: ECLIC 非向量模式中断 common-code 入口地址由 mvt2.COMMON-CODE-ENTRY 决定</div> </div>

7.5.15. jalmnxti

N 级别处理器定义了 jalmnxti 寄存器用于减少中断延迟，加速中断咬尾。

jalmnxti 除了包含 mnxti 的开启中断使能，处理下一个中断，返回下一个中断的入口地址等功能之外，还有跳转至中断 handler 的功能，因此可以缩短中断处理的指令个数，达到减少中断延迟，加速中断咬尾的目的。有关 jalmnxti 的更多细节请参见 5.13.1.3 节。

7.5.16. pushmcause

N 级别处理器定义了通过 pushmcause 寄存器 csrrwi 操作实现的 CSR 指令，存储 mcause 的值到堆栈指针作为基地址的 memory 空间。

以如下指令为例介绍此 CSR 指令：

```
csrrwi x0, PUSHMCAUSE, 1
```

该指令的操作是将 mcause 寄存器的值存到 SP（堆栈指针）+1*4 的地址。

7.5.17. pushmepc

N 级别处理器定义了通过 pushmepc 寄存器 csrrwi 操作实现的 CSR 指令，存储 mepc 的值到堆栈指针作为基地址的 memory 空间。

以如下指令为例介绍此 CSR 指令：

```
csrrwi x0, PUSHMPEC, 1
```

该指令的操作是将 mepc 寄存器的值存到 SP（堆栈指针）+1*4 的地址。

7.5.18. sleepvalue

N 级别处理器内核自定义了一个 CSR 寄存器 sleepvalue 用于控制不同的休眠模式，请参见第 9.1 节了解更多详情。sleepvalue 寄存器中各控制位域如表 7-21 所示。

表 7-21 sleepvalue 寄存器各控制位

域	位	描述
SLEEPVALUE	0	控制 WFI 的休眠模式 <ul style="list-style-type: none"> 0: 浅度休眠模式（执行 WFI 后，处理器内核主工作时钟 core_clk 被关闭） 1: 深度休眠模式（执行 WFI 后，处理器内核主工作时钟 core_clk 和处理器内核的常开时钟 core_aon_clk 都被关闭） 此位复位默认值为 0
Reserved	31:1	未使用的域为常数 0

7.5.19. txevt

N 级别处理器内核自定义了一个 CSR 寄存器 txevt，用于对外发送 Event。

txevt 寄存器中各控制位域如表 7-22 所示。

表 7-22 txevt 寄存器各控制位

域	位	描述
TXEVT	0	控制发送 Event: <ul style="list-style-type: none"> 如果向此位写 1, 则会触发 N 级别处理器内核的输出信号 tx_evt 产生一个单周期脉冲信号, 作为对外的 Event 信号。 该比特位为自清比特位, 即, 向此位写入 1 之后, 下一个周期其被自清为 0。 向此位写入 0 则无任何反应和操作。 此位复位默认值为 0
Reserved	31:1	未使用的域为常数 0

7.5.20. wfe

N 级别处理器内核自定义了一个 CSR 寄存器 wfe, 用于控制 WFI 指令的唤醒条件是使用中断还是使用 Event。请参见第 9.2.3 节了解更多详情。

wfe 寄存器中各控制位域如表 7-23 所示。

表 7-23 wfe 寄存器各控制位

域	位	描述
WFE	0	控制 WFI 指令的唤醒条件是使用中断还是使用 Event。 <ul style="list-style-type: none"> 0: 处理器内核进入休眠模式时, 可以被中断和 NMI 唤醒。 1: 处理器内核进入休眠模式时, 可以被 Event 和 NMI 唤醒。 此位复位默认值为 0。
Reserved	31:1	未使用的域为常数 0

7.5.21. mcfg_info

该寄存器用于查询处理器的配置信息, 只读属性。

表 7-24 mcfg_info 寄存器描述

域	位	描述
Reserved	31:11	未使用域 0
DCACHE	10	D-Cache 配置: <ul style="list-style-type: none"> 0: 没有配置 D-Cache

		<div> <div></div> 1: 配置了 D-Cache </div>
ICACHE	9	I-Cache 配置: <div> <div></div> 0: 没有配置 I-Cache <div></div> 1: 配置了 I-Cache </div>
DLM	8	DLM 配置: <div> <div></div> 0: 没有配置 DLM <div></div> 1: 配置了 DLM </div>
ILM	7	ILM 配置: <div> <div></div> 0: 没有配置 ILM <div></div> 1: 配置了 ILM </div>
NICE	6	NICE 配置: <div> <div></div> 0: 没有配置 NICE <div></div> 1: 配置了 NICE </div>
PPI	5	PPI 配置: <div> <div></div> 0: 没有配置 PPI <div></div> 1: 配置了 PPI </div>
FIO	4	FIO 配置: <div> <div></div> 0: 没有配置 FIO <div></div> 1: 配置了 FIO </div>
PLIC	3	PLIC 配置: <div> <div></div> 0: 没有配置 PLIC <div></div> 1: 配置了 PLIC </div>
CLIC	2	CLIC 配置: <div> <div></div> 0: 没有配置 CLIC <div></div> 1: 配置了 CLIC </div>
ECC	1	ECC 配置: <div> <div></div> 0: 没有配置 ECC <div></div> 1: 配置了 ECC </div>
TEE	0	TEE 配置: <div> <div></div> 0: 没有配置 TEE <div></div> 1: 配置了 TEE </div>

7.5.22. micfg_info

此寄存器用于查询 ILM 和 I-Cache 的配置信息，只读属性。

表 7-25 micfg_info 寄存器描述

域	位	描述
Reserved	XLEN-1:23	未使用域 0
ILM_ECC	22	ILM 配置 ECC: <div> <div></div> 0: ILM 没有配置 ECC <div></div> 1: ILM 配置 ECC </div>
ILM_XONLY	21	ILM 是否是只执行属性，如果是，Load/Store 无法访问 ILM: <div> <div></div> 0: ILM 不是只执行 <div></div> 1: ILM 是只执行 </div>

ILM_SIZE	20:16	ILM 的大小，需为 2 的 n 次方： <ul style="list-style-type: none"> ■ 0: 0 Byte ■ 1: 256 Bytes ■ 2: 512 Bytes ■ 3: 1 KiB ■ 4: 2 KiB ■ 5: 4 KiB ■ 6: 8 KiB ■ 7: 16 KiB ■ 8: 32 KiB ■ 9: 64 KiB ■ 10: 128 KiB ■ 11: 256 KiB ■ 12: 512 KiB ■ 13: 1 MiB ■ 14: 2 MiB ■ 15: 4 MiB ■ 16: 8 MiB ■ 17: 16 MiB ■ 18: 32 MiB ■ 19: 64 MiB ■ 20: 128 MiB ■ 20: 256 MiB ■ 21: 512 MiB ■ Others: Reserved
Reserved	15:11	<ul style="list-style-type: none"> ■ Reserved 0
IC_ECC	10	I-Cache 是否配置 ECC： <ul style="list-style-type: none"> ■ 0: I-Cache 没有配置 ECC ■ 1: I-Cache 配置 ECC
IC_LSIZE	9:7	I-Cache line 大小： <ul style="list-style-type: none"> ■ 0: No I-Cache ■ 1: 8 bytes ■ 2: 16 bytes ■ 3: 32 bytes ■ 4: 64 bytes ■ 5: 128 bytes ■ Others: Reserved
IC_WAY	6:4	I-Cache way（路数）： <ul style="list-style-type: none"> ■ 0: Direct-mapped（直接映射） ■ 1: 2 way ■ 2: 3 way ■ 3: 4 way ■ 4: 5 way ■ 5: 6 way ■ 6: 7 way ■ 7: 8 way
IC_SET	3:0	I-Cache sets per way:（每路的行数） <ul style="list-style-type: none"> ■ 0: 8 ■ 1: 16 ■ 2: 32

		<ul style="list-style-type: none"> ■ 3: 64 ■ 4: 128 ■ 5: 256 ■ 6: 512 ■ 7: 1024 ■ 8: 2048 ■ 9: 4096 ■ 10: 8192 ■ Others: Reserved
--	--	--

7.5.23. mdcfg_info

此寄存器用于查询 DLM 和 D-Cache 的配置信息，只读属性：

表 7-26 mdcfg_info 寄存器描述

域	位	描述
Reserved	XLEN-1:22	Reserved 0
DLM_ECC	21	DLM 是否配置 ECC: <ul style="list-style-type: none"> ■ 0: DLM 没有配置 ECC ■ 1: DLM 配置 ECC
DLM_SIZE	20:16	DLM 的大小，需为 2 的 n 次方: <ul style="list-style-type: none"> ■ 0: 0 Byte ■ 1: 256 Bytes ■ 2: 512 Bytes ■ 3: 1 KiB ■ 4: 2 KiB ■ 5: 4 KiB ■ 6: 8 KiB ■ 7: 16 KiB ■ 8: 32 KiB ■ 9: 64 KiB ■ 10: 128 KiB ■ 11: 256 KiB ■ 12: 512 KiB ■ 13: 1 MiB ■ 14: 2 MiB ■ 15: 4 MiB ■ 16: 8 MiB ■ 17: 16 MiB ■ 18: 32 MiB ■ 19: 64 MiB ■ 20: 128 MiB ■ 20: 256 MiB ■ 21: 512 MiB ■ Others: Reserved
Reserved	15:11	<ul style="list-style-type: none"> ■ Reserved 0

DC_ECC	10	D-Cache 是否配置 ECC: <ul style="list-style-type: none"> 0: D-Cache 没有配置 ECC 1: D-Cache 配置 ECC
DC_LSIZE	9:7	D-Cache line 大小: <ul style="list-style-type: none"> 0: No D-Cache 1: 8 bytes 2: 16 bytes 3: 32 bytes 4: 64 bytes 5: 128 bytes Others: Reserved
DC_WAY	6:4	D-Cache ways (路数): <ul style="list-style-type: none"> 0: Direct-mapped (直接映射) 1: 2 way 2: 3 way 3: 4 way 4: 5 way 5: 6 way 6: 7 way 7: 8 way
DC_SET	3:0	D-Cache sets per way (每路的行数): <ul style="list-style-type: none"> 0: 8 1: 16 2: 32 3: 64 4: 128 5: 256 6: 512 7: 1024 8: 2048 9: 4096 10: 8192 Others: Reserved

7.5.24. mppicfg_info

该寄存器用于查询 PPI 的配置信息。

表 7-27 mppicfg_info 寄存器描述

域	位	描述
PPI_BPA	XLEN-1:10	PPI 的基址, 需为 PPI 大小的整数倍 (对齐); 例如, PPI 的大小为 4KB, PPI 的基址为 12KB (0x3000), 则 PPI_BPA 为 0xC (高 22 位)
Reserved	9:6	Reserved 0
PPI_SIZE	5:1	PPI 的大小, 需为 2 的 n 次方: <ul style="list-style-type: none"> 0: Reserved

		<ul style="list-style-type: none"> ■ 1: 1 KiB ■ 2: 2 KiB ■ 3: 4 KiB ■ 4: 8 KiB ■ 5: 16 KiB ■ 6: 32 KiB ■ 7: 64 KiB ■ 8: 128 KiB ■ 9: 256 KiB ■ 10: 512 KiB ■ 11: 1 MiB ■ 12: 2 MiB ■ 13: 4 MiB ■ 14: 8 MiB ■ 15: 16 MiB ■ 16: 32 MiB ■ 17: 64 MiB ■ 18: 128 MiB ■ 19: 256 MiB ■ 20: 512 MiB ■ 21: 1 GiB ■ 22: 2 GiB Others: Reserved
Reserved	0	Reserved 1

7.5.25. mfiocfg_info

此寄存器用于查询 FIO 的配置信息。

表 7-28 mfiocfg_info 寄存器描述

域	位	描述
FIO_BPA	XLEN-1:10	FIO 的基址，需为 FIO 大小的整数倍（对齐）； 例如，FIO 的大小为 4KB，FIO 的基址为 12KB (0x3000)，则 FIO_BPA 为 0xC（高 22 位）
Reserved	9:6	Reserved 0
FIO_SIZE	5:1	FIO 的大小，需为 2 的 n 次方： <ul style="list-style-type: none"> ■ 0: Reserved ■ 1: 1 KiB ■ 2: 2 KiB ■ 3: 4 KiB ■ 4: 8 KiB ■ 5: 16 KiB ■ 6: 32 KiB ■ 7: 64 KiB ■ 8: 128 KiB ■ 9: 256 KiB ■ 10: 512 KiB ■ 11: 1 MiB

		<ul style="list-style-type: none"> ■ 12: 2 MiB ■ 13: 4 MiB ■ 14: 8 MiB ■ 15: 16 MiB ■ 16: 32 MiB ■ 17: 64 MiB ■ 18: 128 MiB ■ 19: 256 MiB ■ 20: 512 MiB ■ 21: 1 GiB ■ 22: 2 GiB Others: Reserved
Reserved	0	Reserved 1

7.5.26. mecc_lock

mecc_lock 寄存器用于控制是否锁定 ECC 相关的配置寄存器。

表 7-29 mecc_lock 寄存器的控制位

域	位	读写权限	复位值	描述
Reserved	XLEN-1:1	RO	0	-
ECC_LOCK	0	RW	0	用于锁定ECC相关对的配置寄存器： 0: 表是不锁定； 1: 表示锁定； ecc_lock用于锁定ECC的配置，锁定之后，所有相关的ecc寄存器均不能再修改，除非复位。

7.5.27. mecc_code

mecc_code 寄存器用于 ECC 校验码注入，可用于软件模拟 ECC 错误。

表 7-30 mecc_code 寄存器的控制位

Field Name	Bits	RW	Reset Value	Description
Reserved	XLE N-1:2 9	RO	0	-
SRAMID[4:0]	28:24	RW	0	用于标识出现1比特ECC错误，软件可清除此位：

				SRAMID[0]: I-Cache有1比特ECC错误; SRAMID[1]: D-Cache有1比特ECC错误; SRAMID[2]: Main TLB有1比特ECC错误; (UX级别) SRAMID[3]: ILM有1比特ECC错误; SRAMID[4]: DLM有1比特ECC错误;
Reserved	23:21	RW	0	-
RAMID[4:0]	20:16	RW	0	用于标识出现2比特ECC错误, 软件可清除此位: SRAMID[0]: I-Cache有2比特ECC错误; SRAMID[1]: D-Cache有2比特ECC错误; SRAMID[2]: Main TLB有2比特ECC错误; (UX级别) SRAMID[3]: ILM有2比特ECC错误; SRAMID[4]: DLM有2比特ECC错误;
Reserved	[15:7] or [15:8] or [15:9]	RO	0	-
Code	[6:0] or [7:0] or [8:0]	RW	0	用于注入的ECC校验码: Max(TLB Width, Data Bus Width) > 64: The width of Code is 9; Max(TLB Width, Data Bus Width) > 32: The width of Code is 8; Max(TLB Width, Data Bus Width) <= 32: The width of Code is 7;

8. N 级别处理器内核物理存储器保护机制介绍

8.1. PMP 简介

由于 N 级别处理器内核是面向微控制器领域的低功耗内核，其不支持虚拟地址管理单元（MMU），因此所有的地址访问操作都是使用的物理地址。为了根据不同的物理地址区间和不同的 Privilege Mode 进行权限隔离和保护，N 级别处理器内核（可配置地）支持 PMP（Physical Memory Protection）单元。

注意：具体每一款处理器内核（譬如 N203，N205 等）是否支持 PMP 可能略有差异，请参见各个系列的简明数据手册了解其详情。

PMP 的相关要点如下：

- PMP 支持若干个表项（Entries），每个表项设定某个地址区间的权限（是否可读、写、执行）。PMP 每个表项的权限设定需要通过 CSR 寄存器配置完成，相关 CSR 寄存器的详解请参见第 8.2 节。
 - 注意：N 级别处理器内核的 PMP 支持的表项数目可以配置，请参见各个系列的简明数据手册了解其配置详情。
- PMP 可以支持最小 4 个字节（32bits）大小的地址区间进行权限设定。相关 CSR 寄存器的详解请参见第 8.2 节。
- PMP 允许配置更加粗粒度的保护区域，粗粒度的定义可以参考“RISC-V Privileged Spec”，该配置选项信息可参考相应处理器系列的数据手册。

8.2. PMP 的 CSR 寄存器

8.2.1. PMP 的 CSR 寄存器列表

Nuclei N 级别处理器内核的 PMP 支持的 CSR 寄存器列表如表 8-1 所示。

表 8-1 PMP 的 CSR 寄存器列表

类型	CSR 地址	读写属性	名称	全称
PMP 相关 CSR 寄存器	0x3A0	MRW	pmpcfg0	PMP 表项权限配置寄存器
	0x3B0	MRW	pmpaddro	PMP 表项 0 的地址配置寄存器
	0x3B1	MRW	pmpaddr1	PMP 表项 1 的地址配置寄存器
	0x3B2	MRW	pmpaddr2	PMP 表项 2 的地址配置寄存器
	0x3B3	MRW	pmpaddr3	PMP 表项 3 的地址配置寄存器
	MRW
	(0x3B0+n)	MRW	pmpaddrn	PMP 表项 n 的地址配置寄存器
注意： <ul style="list-style-type: none"> ■ 在 N 级别处理器内核中，如果没有配置 User Mode，则相应的不会有 PMP 特性和配套的 CSR 寄存器。 ■ MRW 表示只有在 Machine Mode 下才可以对其进行读写。 				

8.2.2. pmpcfg<x>寄存器

pmpcfg<x>寄存器用于指定 PMP 表项的权限配置寄存器，每个 pmpcfg<x>寄存器可以管理四个 PMP 表项目，譬如 pmpcfg0 包含了四个不同的域（pmp0cfg~pmp4cfg），分别管理表项 0 到表项 4。其他一次类推，如图 8-1 中所示。

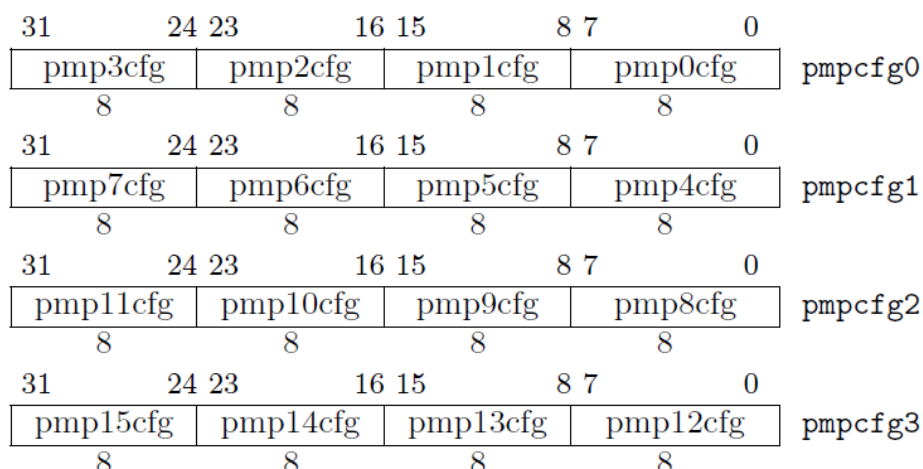


图 8-1 CSR 寄存器 pmpcfg0 的格式

每个表项的权限配置寄存器 pmp<x>cfg 中各控制位域如表 8-2 所示。

表 8-2 pmp<x>cfg 中各控制域

域	位	描述
R	0	此位控制该表项的数据可读权限： <ul style="list-style-type: none"> 如果此位为 1，则表示此表项设定的地址区间具备数据可读的权限。 如果此位为 0，则表示此表项设定的地址区间不具备数据可读的权限。 此位复位默认值为 0。
W	1	此位控制该表项的数据可写权限： <ul style="list-style-type: none"> 如果此位为 1，则表示此表项设定的地址区间具备数据可写的权限。 如果此位为 0，则表示此表项设定的地址区间不具备数据可写的权限。 此位复位默认值为 0。
X	2	此位控制该表项的代码可执行权限： <ul style="list-style-type: none"> 如果此位为 1，则表示此表项设定的地址区间具备代码可执行的权限。 如果此位为 0，则表示此表项设定的地址区间不具备代码可执行的权限。 此位复位默认值为 0。
A	4:3	此位控制该表项的地址区间模式，参见表 8-3 了解其详情。 此位复位默认值为 0。
Reserved	6:5	未使用的域为常数 0
L	7	此位控制是否锁定该 PMP 表项： <ul style="list-style-type: none"> 如果此位为 1，则该表项被锁定 如果此位为 0，则该表项没有被锁定 请参见第 8.3 节了解表项锁定的详情。 此位复位默认值为 0。

每个表项的权限配置寄存器 pmp<x>cfg 中的“A”比特域（第 4~3 位）用于控制该表项的地址区间模式，其编码和模式的含义如表 8-3 所示。

表 8-3 寄存器 pmp<x>cfg 中的“A”比特域

编码	模式名称	英文全称	中文解释
0	OFF	Null region (disabled)	空模式，即：没有配置此表项
1	注意：N 级别处理器内核不支持此编码的模式，如果向此配置域写入 1，硬件上得到的结果仍然是 0		
2	NA4	Naturally aligned four-byte region	4 个字节大小的区域（地址对齐），参见表 8-5 了解其详情。
3	NAPOT	Naturally aligned power-of-two region (>= 8 bytes)	2 的幂次方大小的区域（区域大小起码大于等于 8 个字节，并且地址对齐），参见表 8-5 了解其详情。

8.2.3. pmpaddr<x>寄存器

pmpaddr<x>寄存器用于指定 PMP 每个表项的地址配置寄存器。每个 pmpaddr<x>寄存器的格式如表 8-4 中所示。

表 8-4 pmpaddr<x>中各控制域

域	位	描述
ADDR	29:0	配置地址和地址模式，参见表 8-5 了解其详情。
Reserved	31:30	未使用的域为常数 0

PMP 每个表项设定的地址区间大小需要由其对应的地址配置寄存器 pmpaddr<x>的“ADDR”比特域和权限配置寄存器 pmp<x>cfg 的“A”比特域联合设置，如表 8-5 中所示。

表 8-5 PMP 表项设定的地址区间大小

表项地址寄存器 pmpaddr<x>的“ADDR” 比特域设置的值	表项权限寄存器 pmp<x>cfg 的“A”比 特域设置的模式	表项设定的地址区间大小
aa.....aaaa	NA4	4 个字节的区域（地址对齐），该区域的基地址为“aa...aaaa00”
aa.....aaa0	NAPOT	“2 的 3 次方（8）”个字节的区域（地址对齐），该区域的基地址为“aa...aaa000”
aa.....aa01	NAPOT	“2 的 4 次方（16）”个字节的区域（地址对齐），该区域的基地址为“aa...aa0000”
aa.....a011	NAPOT	“2 的 5 次方（32）”个字节的区域（地址对齐），该区域的基地址为“aa...a00000”
.....
01.....1111	NAPOT	“2 的 32 次方（4G）”个字节的区域（地址对齐）。由于 N 级别处理器内核的地址空间为 32 位，则意味着整个 32 位地址空间都被覆盖。
注意：上表中的字母 a 表示任意值。		

8.3. PMP 表项的锁定

pmp<x>cfg 寄存器的“L”比特域（第 7 位）用于控制是否将 PMP 表项<x>进行锁定，其要

点如下：

- 如果 `pmp<x>cfg` 寄存器的“L”比特域（第 7 位）被写为 1，则表项<x>被锁定。
- 当表项<x>被锁定后，则表项<x>对应的权限配置寄存器 `pmp<x>cfg` 和地址配置寄存器 `pmpaddr<x>` 将不能够再被改写。
- 当表项<x>被锁定后，由于对应的权限配置寄存器 `pmp<x>cfg` 不能够再次被改写，因此软件无法通过清零“L”比特域进行解锁。唯一解锁 PMP 表项的方法便是整个处理器内核的复位。
- 当表项<x>没有锁定时，该表项设定的权限和地址区间保护仅对 User Mode 下的操作进行保护；当表项<x>被锁定后，该表项设定的权限和地址区间保护同样适用于 Machine Mode 下的操作。

8.4. PMP 保护触发异常

任何存储器访问操作都需要进行 PMP 权限检测，PMP 权限检测原则请参见第 8.7 节。

如果访问存储器操作违反了 PMP 设定的权限，则处理器会触发异常，PMP 触发的异常种类根据情形分为：

- 如果是取指令操作造成的异常为“Instruction access fault”。
- 如果是 Load 指令（数据存储器读操作）造成的异常为“Load access fault”。
- 如果是 Store 指令（数据存储器写操作）或者 AMO（原子操作指令）造成的异常为“Store/AMO access fault”。

有关“异常”的详情，请参见第 3 章。

8.5. PMP 表项匹配的优先级

由于 N 级别处理器内核支持多个 PMP 表项，因此当一个操作同时匹配多个表项时，则编号数目小的表项具有更高的优先级。

8.6. PMP 表项匹配的跨边界情况

由于 N 级别处理器 PMP 的 `pmp<x>cfg` 中 “A” 比特域能够设置的地址模式都是 2 的幂次方（或者 4 个字节）对齐的地址区间，因此，如非对齐的存储器读写访问可能跨越了一个 PMP 设定的地址区间边界，其处理方式如下：

- 如果是普通数据存储器读写操作（Load、Store 指令）：
 - 如果 N 级别处理器内核被配置为硬件不支持地址非对齐的数据存储器读写操作，那么当访问地址非对齐时会产生此“地址非对齐”异常，从而没有机会产生 PMP 触发的异常。
 - 如果 N 级别处理器内核被配置为硬件支持地址非对齐的数据存储器读写操作，那么当访问地址非对齐时，硬件会将原本的操作拆分为逐个字节的访问操作，譬如，一个“半字（half-word）”的非对齐操作会被拆分为 2 个“字节（Byte）”的操作。那么，对于每一个“字节”的操作，会单独进行 PMP 检查，如果违反了 PMP 设定的权限，就会触发异常。注意：此类异常属于非精确异常。
- 如果是 AMO 数据存储器读写操作（AMO 指令）：
 - RISC-V 架构规定 AMO 指令不支持地址非对齐的数据存储器读写操作，因此当访问地址非对齐时会产生此“地址非对齐”异常，从而没有机会产生 PMP 触发的异常。
- 如果是取指令操作，可能一条 32 位宽的指令正好位于一个非对齐的边界。由于 N 级别处理器内核总是以 32 位对齐的方式进行指令的预取，即，一个非对齐的 32 位指令预取会被拆分成两个 32 位对齐的读操作，因此，这两个独立的读操作（32 位对齐）会分别进行 PMP 的权限检测，只要任何一部分违反了 PMP 设定的权限，都会触发异常。

8.7. PMP 权限检测的原则

PMP 权限检测的原则和要点如下：

- 如果当前的操作处于 Machine Mode 下，则：

- 如果当前操作没有匹配到任何 PMP 表项，则该操作不会触发异常。
- 如果当前的操作匹配到了 PMP 表项：
 - ◆ 首先判断 `mstatus.MPRV` 域的值是否为 1：
 - 如果 `mstatus.MPRV` 域的值 1，并且 `mstatus.MPP` 域的值表示 User Mode，则 PMP 用 User Mode 下的权限检测规则对存储器的数据读写 (Load 和 Store) 操作做权限检测，并根据表项配置的 R/W 属性进行权限确认，如果违反了权限，则触发异常。
 - ◆ 否则继续判断表项是否被锁定：
 - 如果表项没有被锁定，则意味着权限检测不适用于 Machine Mode，无需对当前操作进行权限检测。
 - 如果表项被锁定，则意味着权限检测同样适用于 Machine Mode，需根据表项配置的 X/R/W 属性进行权限确认，如果违反了权限，则触发异常。
- 如果当前的操作处于 User Mode 下，则：
 - 如果当前操作没有匹配到任何 PMP 表项，则该操作一定会触发异常。这意味着在 User Mode 下，如果 PMP 没有配置合适的表项，任何访问操作都不能够成功。
 - 如果当前的操作匹配到了 PMP 表项，则需根据表项配置的 X/R/W 属性进行权限确认，如果违反了权限，则触发异常。

9. N 级别处理器内核低功耗机制介绍

N 级别处理器内核可以支持休眠模式实现较低的静态功耗。

9.1. 进入休眠状态

N 级别处理器内核可以通过 WFI 指令进入休眠状态。当处理器执行到 WFI 指令之后，将会：

- 立即停止执行当前的指令流；
- 等待处理器内核完成任何尚未完成的滞外操作（Outstanding Transactions），譬如取指令和数据读写操作，以保证发到总线上的操作都完成；
 - 注意：如果在等待总线上的操作完成的过程中发生了存储器访问错误异常，则会进入到异常处理模式，而不会休眠。
- 当所有的滞外操作（Outstanding Transactions）都完成后，处理器会安全地进入一种空闲状态，这种空闲状态可以被称之为“休眠”状态。
- 当进入休眠模式后：
 - N 级别处理器内核内部的各个主要单元的时钟将会被门控关闭以节省静态功耗；
 - N 级别处理器内核的输出信号 `core_wfi_mode` 会拉高，指示此处理器核处于执行 WFI 指令之后的休眠状态；
 - N 级别处理器内核的输出信号 `core_sleep_value` 会输出 CSR 寄存器 `sleepvalue` 的值（注意：该信号只有在 `core_wfi_mode` 信号为高电平时生效；`core_wfi_mode` 信号为低电平时 `core_sleep_value` 的值一定是 0）。软件可以通过事先配置 CSR 寄存器 `sleepvalue` 来指示不同的休眠模式（0 或者 1）。注意：
 - ◆ 对于不同的休眠模式而言，N 级别处理器内核的行为完全一样。此休眠模式只是仅供 SoC 系统层面的 PMU（Power Management Unit）进行相应不同的控制。

9.2. 退出休眠状态

N 级别处理器内核处理器退出休眠模式的要点如下：

- N 级别处理器内核的输出信号 `core_wfi_mode` 会相应拉低。
- N 级别处理器内核处理器可以通过以下四种方式被唤醒：
 - NMI
 - 中断
 - Event
 - Debug 请求

下文将予以详细介绍。

9.2.1. NMI 唤醒

NMI 总能够唤醒处理器内核。当处理器内核检测到输入信号 `nmi` 的上升沿，处理器内核被唤醒，进入到 NMI 服务程序开始执行。

9.2.2. 中断唤醒

中断也可以唤醒处理器内核：

- 如果 CSR 寄存器 `wfe.WFE` 域被配置为 `0`，则：
 - 如果 `mstatus.MIE` 域被配置为 `1`（表示全局中断被打开），则：
 - ◆ 当 ECLIC（通过将外部请求的中断进行仲裁）向处理器内核发送了中断，处理器内核被唤醒，进入到中断服务程序开始执行。
 - 如果 `mstatus.MIE` 域被配置为 `0`（表示全局中断被关闭），则：
 - ◆ 当 ECLIC（通过将外部请求的中断进行仲裁）向处理器内核发送了中断，处理器

内核被唤醒，继续顺序执行之前停止的指令流（而不是进入到中断服务程序）。

- 如果 CSR 寄存器 `wfe.WFE` 域被配置为 1，则等待 Event 唤醒，请参见下节描述。

9.2.3. Event 唤醒

当满足如下条件时，Event 可以唤醒处理器内核：

- 如果 CSR 寄存器 `wfe.WFE` 域被配置为 1，则：
 - 当处理器内核检测到输入信号 `rx_evt`（称之为 Event 信号）为高电平时，处理器内核被唤醒，继续执行之前停止的指令流（而不是进入到中断服务程序）。

9.2.4. Debug 唤醒

Debug 请求总能够唤醒处理器内核，如果调试器（Debugger）接入，也会将处理器内核唤醒而进入调试模式。

9.3. Wait for Interrupt 机制

Wait for Interrupt 机制，是指将处理器内核进入休眠模式，然后等待中断唤醒处理器内核，醒来后进入相应中断的处理函数中去。

如第 9.1 节和第 9.2 节所述，Wait for Interrupt 机制可以直接通过 WFI 指令（配合 `mstatus.MIE` 域被配置为 1）完成。

9.4. Wait for Event 机制

Wait for Event 机制，是指将处理器内核进入休眠模式，然后等待 Event 唤醒处理器内核，醒来后继续先前停止的程序（而不是进入中断的处理函数中去）。

如第 9.1 节和第 9.2 节所述，Wait for Event 机制可以直接通过 WFI 指令，配合如下指令序列完成：

第 1 步：配置 `wfe.WFE` 域为 1

第 2 步：调用 WFI 指令。调用此指令后处理器会进入休眠模式，当 Event 或者 NMI 将其唤醒后将会继续向下执行。

第 3 步：恢复 `wfe.WFE` 域为 0

10. N 级别处理器 ECC 处理机制介绍

N 级别处理器内核可以在 ILM, DLM, I-Cache, D-Cache 存储模块配置 ECC 保护机制。

10.1. Nuclei ECC 总体介绍

Nuclei ECC 的保护策略如下：

- 支持 1 比特纠错，2 比特检错；
 - 支持检测并纠正 1 比特错误；
 - 支持检测 2 比特错误，但不能纠正 2 比特错误；

Nuclei ECC 支持保护如下几类存储单元：

- Cache
 - I-Cache, 包括 Tag Ram 和 Data Ram
 - D-Cache, 包括 Tag Ram 和 Data Ram
- Local Memory
 - ILM
 - DLM
- 注意：在 RV32 和 RV64 架构下，对于 DLM 和 D-Cache 的 Data Ram，统一以 32 比特为单位进行保护；对于 ILM 和 I-Cache 的 Data Ram 则统一以 64 比特为单位进行保护，对于 I-Cache 和 D-Cache 的 Tag Ram 则以实际的数据位宽为单位进行保护。

Nuclei ECC SRAM 的更新策略：

- Full Write
 - 以 Word 方式写 DLM 为例，Nuclei ECC 以 32 比特为单位进行保护，ECC 检验码会同 32 比特的数据一同更新到 DLM 的 SRAM 中；
- Partial Write
 - 以写 Byte/Half Word 写 DLM 为例，Nuclei ECC 以 32 比特为单位进行保护，Nuclei

ECC 机制规定须先从 DLM 的 SRAM 中读出对应访问地址的数据，再和写数据进行 Merge，并通过 Merge 后的数据生成 ECC 校验码，最后再将 Merge 后的数据和对应的 ECC 校验码一同更新到 DLM 的 SRAM 中；Partial Write 的策略也简称 Read-Modify-Write。

Nuclei ECC 模式控制：

- Nuclei ECC 支持对于每一类存储单元进行单独的模式控制，这些存储单元包括：I-Cache、D-Cache、ILM、DLM，模式控制包括：ECC 使能和 ECC 错误异常使能；
- ECC 使能，用于控制是否进行 ECC 检测：

ECC 使能打开，ECC 检测逻辑的行为如下：

- 更新 SRAM 数据的时候，会同时更新 ECC 校验码；
- 对于 Partial Write，采用 Read-Modify-Write 更新 SRAM；
- 对于 1 比特错误，会进行纠错，并将纠错后的结果（数据+ECC 校验码）更新至 SRAM，同时将纠错后的结果送给处理器内核（请求源）；
- 对于 2 比特错误，直接将错误的结果送给处理器内核并上报 ECC 错误（ECC 错误异常使能须打开才能上报异常）；

ECC 使能关闭，ECC 检测逻辑的行为如下：

- 更新 SRAM 数据的时候，不再更新 ECC 校验码；
- 对于 Partial Write，不再采用 Read-Modify-Write 更新 SRAM，而是直接更新 SRAM；
- 对于 1 比特错误，不会进行纠错和更新至 SRAM，并将直接将错误的结果送给处理器内核（请求源）；
- 对于 2 比特错误，直接将错误的结果送给处理器内核，但不报两比特 ECC 错误（即使 ECC 错误异常使能已经打开）；

ECC 错误异常使能，用于控制发生 2 比特错误时，是否上报异常，ECC 错误异常使能，只有在 ECC 使能打开的情况下才有效；ECC 错误异常使能打开，ECC 逻辑的行为如下：

- 对于 2 比特错误，上报 ECC 错误（ECC 错误异常使能须打开才能上报异常）；

ECC 错误异常使能关闭，ECC 逻辑的行为如下：

- 对于 2 比特错误，不上报 ECC 错误；

不同的访问源造成的 2 比特 ECC 错误的处理：

- **Pipeline:** IFU 取指访问上报精确异常，LSU 的 load/store 上报非精确异常；
- **I-Cache CCM (缓存控制和维护操作):** 对于单个地址的操作，会将 2 比特错误返回给 CCM 状态机，再由 CCM 状态机上报给 longp_wbck(store access fault), mecc_code CSR 寄存器中 RAMID 的 i-cache 位置 1, mtval/stval 更新成 0；对于 INV_ALL 的操作，硬件不会关心 Cache-Line 是否有 ECC 错误，直接 invalidate 整个 I-Cache，也不会上报异常；
- **D-Cache CCM (缓存控制和维护操作):** 对于单个地址的操作，如果在 D-Cache 访问阶段出现 2 比特错误 (cpbk 的除外)，返回给 CCM 状态机，再由 CCM 状态机上报给 longp_wbck (store access fault), mecc_code CSR 寄存器中 RAMID 的 d-cache 位置 1, mtval/stval 更新成 0；对于 ALL (FLUSH_ALL/INV_ALL/FLUSH&INV_ALL) 的操作，如果某个 Cache Line 出现 2 比特 ECC 错误，则会终止 ALL 操作，并将 2 比特 ECC 错误结果返回给 CCM 状态机，再由 CCM 状态机上报给 longp_wbck(store access fault), mecc_code CSR 寄存器中 RAMID 的 d-cache 位置 1, mtval/stval 更新成 0；
- **D-Cache evict (cpbk):** 上报非精确异常，mecc_code CSR 寄存器中 RAMID 的 d-cache 位置 1；
- **Slave Port:** 会将 2 比特 ECC 的错误通过 slave port 的 bus error 信号上报 (不区分 ECC 还是 bus error)；
- **Debug SBA:** 上报 2 比特 ECC 错误给 SBA，但不区分 bus error 和 ECC 错误；
- 对 I-Cache/D-Cache/ILM/DLM 发起的任何类型的访问，如果出现 2 比特 ECC 错误，mecc_code CSR 寄存器中 RAMID 对应的 bit 会置 1，表明该类型的 memory 发生过 2 比特 ECC 错误；需要注意的是，RAMID 对应的 bit 位置 1，并不意味着一定会产生异常，例如，Slave Port 或者 SBA 的访问中出现了 2 比特 ECC 错误，RAMID 的对应 bit 会置 1，但不会产生异常，或者 IFU 取指过程中出现了 2 比特 ECC 错误，RAMID 对应的 bit 会置 1，但是这条错误的指令在 pipeline 中可能被跳转指令 flush 掉了；
- 2 比特 ECC 错误不再更新 mdcause，而是更新 mecc_code CSR 寄存器中 RAMID，并且 2 比特 ECC 错误导致的异常，都会上报 access fault，而不是 page fault，因此在 access fault 的 handler 中也需要查询 RAMID 来判断是否出现了两比特 ECC 的错误；
- Core 的顶层会输 2 比特 ECC 错误信息 (每类存储单元分配一个比特)，供 SoC 使用；
- 对 I-Cache 和 D-Cache 发起的任何类型的访问，Tag Ram 比对期间，只要其中有任何一个或者多个 way 出现 ECC 错误，都会上报异常，即使其中有 hit 且无 2 比特 ECC 错误的 way；

不同的访问源造成的 1 比特 ECC 错误的处理：

- 硬件自动纠正 1 比特错误，且不上报异常；
- 对 I-Cache/D-Cache/ILM/DLM 发起的任何类型的访问，如果出现 1 比特错误，mecc_code CSR 寄存器中 SRAMID 对应的 bit 会置 1，表明该类型的 memory 发生过 1 比特错误；
- Core 的顶层会输 1 比特 ECC 错误信息（每类存储单元分配一个比特），供 SoC 使用；

ECC 注错：

- 对于 ILM（ILM 需要能够被 LSU 访问）/DLM，可以通过 store 的方式，将 mecc_code 寄存器中指定的 ECC 校验码进行注入，由 milmm_ctl 和 mdlm_ctl CSR 寄存器控制；
- 对于 I-Cache 和 D-Cache，通过 refill 的方式，将 mecc_code 寄存器中指定的 ECC 校验码进行注入，可由 mcache_ctl CSR 寄存器控制从 Tag Ram 注入还是从 Data Ram 注入；

ECC Lock：

- 防止 ECC 的安全漏洞，由 mecc_lock CSR 寄存器控制；

10.2. Nuclei ECC 相关的 CSR 寄存器

ECC 相关的寄存器属于 Nuclei 自定义寄存器，全部的寄存器汇总见下表所示。

表 10-1 Nuclei ECC 相关 CSR 寄存器列表

类型	CSR 地址	读写属性	名称	全称
ECC 自定义 CSR	0xFC0	MRO	micfg_info	用于反应 ILM/I-Cache 的配置信息
	0xFC1	MRO	mdcfg_info	用于反应 DLM/D-Cache 的配置信息
	0xFC2	MRO	mcfg_info	反应一些微架构相关的配置信息
	0x7C0	MRW	milmm_ctl	用于反应 ILM 的配置和控制信息
	0x7C1	MRW	mdlmm_ctl	用于反应 DLM 的配置和控制信息
	0x7C2	MRW	mecc_code	ECC 校验码获取和注入寄存器，用于软件模拟 ECC 错误
	0x7DE	MRW	mecc_lock	ECC 的 lock，用于锁定 ECC 的配置，锁定之后，所有相关的 ecc 寄存器均不能再修改，除非复位
	0x7CA	MRW	mcache_ctl	Cache 控制寄存器