密级
机密

技术报告

名称： WH 处理器内核 API 接口说明

编号：

版本号：V0.1

作者 廖力灵

项目

日期 2019-05-29

深圳优矽科技有限公司

版本更新历史：

日期	版本	更新人	更新内容
2019-05-29	v0.1	廖力灵	
2019-07-17	v0.1.1	叶锡聪	增加中断机制
2019-07-18	v0.1.2	叶锡聪	增加模式切换
2020-07-13	v0.1.3	叶锡聪	更新 PMP、SV32 相关 API 描述

阅读说明

在阅读本文档之前，请先阅读该部分。

（1）本文档主要介绍 WH 处理器内核机制，包括中断机制，PMP 机制，模式切换，虚拟内存转换机制，并提供简单的 API 库。

（2）本文档提及的寄存器可分为 CSR 寄存器和内存映射寄存器，对于 CSR 寄存器的描述会使用前缀“CSR”。

（3）本文档所有提及的内存映射寄存器的操作都是基于指针寻址的操作，各个设备的物理地址在相应设备说明中已经列出，同时每个设备的功能寄存器地址都是基于该设备物理首地址的偏移地址。（注意：采用的地址为字节地址）。

（4）本参考文档基于 RISC-V 指令集手册进行描述。

目录

1. 中断机制

WH 具有三种中断，分别是 PLIC 中断、定时器中断、软件中断，其中 PLIC 中断数量最大可以达到 1024，而 PLIC 的中断由 PLIC 控制器负责管理。

由于 WH 处理器支持机器模式（M-Mode）、管理员模式（S-Mode）、用户模式（U-Mode），每种特权模式都具有自己的定时器中断和软件中断（PLIC 中断由 PLIC 控制器负责管理），所以处理器在不同的特权模式下，对中断的响应会有不同的操作，不过中断的处理流程大致一样，M-Mode 的中断处理如图 1 所示。

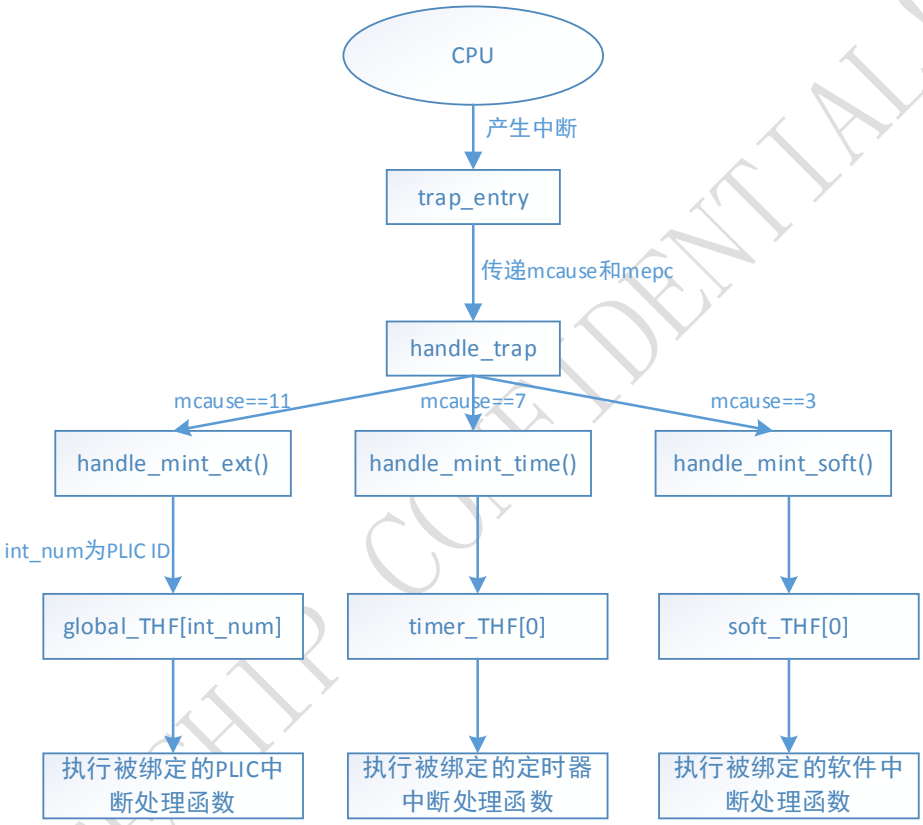


图 1 M-Mode 中断处理流程

以 M-Mode 中断为例，在 M-Mode 特权下产生了中断后，会陷入到 trap_entry 函数中，它会把 mcause 和 mepc 的值作为参数传递给 handle_trap，handle_trap 再对 mcause 进行分类处理，不同的 mcause 会调用不同的中断处理函数。

1.1 驱动文件列表

文件名称	文件说明
trap_entry.S	该文件为汇编文件，它定义了 M-Mode 中断入口函数 trap_entry
int.{c,h}	定义了中断入口函数，以及总中断的使能、失能函数
plic.{c,h}	定义了 PLIC 中断函数和相关宏定义
clint.{c,h}	定义了 CLINT 中断函数和相关宏定义

1.2 宏定义

1.2.1 plic.h 文件中的宏定义：

宏定义	具体值	描述
PLIC_NUM	256	PLIC 中断数量
PLIC_PRIORITY_OFFSET	0x0000	PLIC 优先级寄存器的偏移地址
PLIC_PRIORITY_SIZE	PLIC_NUM<<2	PLIC 优先级寄存器的大小
PLIC_PENDING_OFFSET	0x1000	PLIC 标志位寄存器的偏移地址
PLIC_ENABLE_OFFSET	0x2000	PLIC 使能寄存器的偏移地址
PLIC_ENABLE_SIZE	(PLIC_NUM>>3)+1	PLIC 使能寄存器的大小
PLIC_THRESHOLD_OFFSET	0x200000	PLIC 阈值寄存器的偏移地址
PLIC_CLAIM_OFFSET	0x200004	PLIC CLAIM 寄存器的偏移地址

1.2.2 clint.h 文件中的宏定义：

宏定义	具体值	描述
LOCAL_INT_MSIP	0x0000	MSIP 寄存器的偏移地址
LOCAL_INT_MTIMECMP	0x4000	MTIMECMP 寄存器的偏移地址
LOCAL_INT_MTIMECMP_SIZE	0x8	MTIMECMP 寄存器的字节大小
LOCAL_INT_MTIME	0xBFF8	MTIME 寄存器的偏移地址
LOCAL_INT_MTIME_SIZE	0x8	MTIME 寄存器的字节大小

1.3 函数介绍

1.3.1 void trap_entry(void)

1） 描述：

M-Mode 中断入口函数。int_init 函数会将 trap_entry 的地址写入到 mtvec 中，这样当 M-Mode 的中断产生时，CPU 就会自动跳转到 trap_entry 函数，然后该函数会把所有寄存器压栈，并且把 mcause、mepc 寄存器的值作为参数传给 handle_trap 函数。

1.3.2 void int_init(void)

1） 描述：

将所有的 CLINT、PLIC 中断复位，初始化所有中断，同时将 trap_entry 函数所在的地

址传到 mtvec 寄存器中。

1.3.3 void int_enable(void)

1） 描述：

将 mstatus 寄存器中的 MIE 位置 1，开启 M-Mode 中断。

1.3.4 void int_disable(void)

1） 描述：

与 void int_enable(void)函数类似，不过该函数将 mstatus 寄存器中的 MIE 位置 0，关闭所有 M-Mode 中断。

1.3.5 int handle_trap(int mcause, int epc)

1） 描述：

当中断产生后，CPU 会跳转到 trap_entry 函数。trap_entry 函数会调用 handle_trap 函数，并且把 mcause 和 mepc 寄存器的值作为参数传过来，对于不同的 mcause，handle_trap 函数会调用不同的中断处理函数。

2） 参数：

mcause：mcause 寄存器的值，不同的值代表不同的异常和中断

epc：mepc 寄存器的值

3） 返回值：

该函数会返回 PC 应该跳转的下一条指令的所在地址

1.3.6 void memzero(void * base_addr, unsigned int size)

1） 描述：

该函数会将[base_addr, base_addr+size-1]地址映射的存储全部置 0.

2） 参数：

base_addr：需要复位的基地址

size：需要复位的存储大小

1.3.7 void global_int_init(void)

1） 描述：

初始化所有 PLIC 寄存器。

1.3.8 void global_int_enable(int id)

1） 描述：

使能编号为 id 的 PLIC 中断。

2） 参数：

id：需要使能的 PLIC 中断编号。

1.3.9 void global_int_disable(int id)

1） 描述：

关闭编号为 id 的 PLIC 中断。

2） 参数：

id：需要使能的 PLIC 中断编号。

1.3.10 void global_int_priority(int id, int priority)

1） 描述：

为编号为 id 的 PLIC 中断设置优先级。

2） 参数：

id：PLIC 中断编号

priority：中断优先级

1.3.11 void global_int_threshold(int priority)

1） 描述：

为 PLIC 设置优先级阈值，所有优先级小于或等于 priority 的 PLIC 中断，都会被屏蔽。

2） 参数：

priority：中断优先级阈值

1.3.12 void global_int_bind_handler(int id, void(*handle_function)())

1） 描述：

将编号为 id 的 PLIC 中断与 handle_function 函数绑定，当该编号的 PLIC 中断产生时，会跳转到 handle_function 所在的函数进行处理。

2） 参数：

id：PLIC 中断编号

handle_function：需要绑定的中断处理函数

1.3.13 void timer_int_init(void)

1） 描述：

将 mie 寄存器中 MTIE 位置 0，关闭 M-Mode 的定时器中断。

1.3.14 void timer_int_enable(void)

1） 描述：

将 mie 寄存器中 MTIE 位置 1，使能 M-Mode 的定时器中断。

1.3.15 void timer_int_disable(void)

1） 描述：

将 mie 寄存器中 MTIE 位置 0，关闭 M-Mode 的定时器中断。

1.3.16 unsigned long get_timer_val(void)

1） 描述：

获取 mtime 寄存器的值。

2） 返回值：

返回 mtime 寄存器中的值

1.3.17 void set_timer_cmp(unsigned long value)

1） 描述：

将 mtimecmp 寄存器的值设置为 value

2） 参数：

value：需要设置到 mtimecmp 寄存器中的数值

1.3.18 void timer_int_bind_handler(void(*handle_function)(void))

1） 描述：

将 M-Mode 定时器中断与 handle_function 函数进行绑定，当 M-Mode 的定时器中断产生时，就会跳转到 handle_function 函数进行处理。

2） 参数：

handle_function：需绑定的中断处理函数

1.3.19 void soft_int_init(void)

1） 描述：

将 mie 寄存器中 MSIE 位置 0，关闭 M-Mode 的软件中断。

1.3.20 void soft_int_enable(void)

1） 描述：

将 mie 寄存器中 MSIE 位置 1，使能 M-Mode 的软件中断。

1.3.21 void soft_int_disable(void)

1） 描述：

将 mie 寄存器中 MSIE 位置 0，关闭 M-Mode 的软件中断。

1.3.22 void soft_int_bind_handler(void(*handle_function)(void))

1） 描述：

将 M-Mode 软件中断与 handle_function 函数进行绑定，当 M-Mode 的软件中断产生时，就会跳转到 handle_function 函数进行处理。

2） 参数：

handle_function：需绑定的中断处理函数

1.3.23 void soft_int_start(void)

1） 描述：

往 msip 寄存器写 1，触发软件中断。

1.3.24 void soft_int_stop(void)

1） 描述：

往 msip 寄存器写 0，清除软件中断。

1.4 例程

1.4.1 M-Mode 定时器中断

M-Mode 定时器中断的触发、处理流程如下：

（1） 初始化所有中断，并且将 trap_entry 函数所在地址写入到 mtvec 寄存器中；

（2） 使能 M-Mode 的定时器中断；

（3） 绑定定时器中断函数；

（4） 设置 mtimecmp 寄存器的值为 50，那么当 mtime 大于或等于 50 时，M-Mode 的定时器中断就会被触发；

（5） 打开 M-Mode 中断（否则只开定时器中断是无法触发中断的）；

（6） 将中断委派寄存器 mideleg 置 0（避免对 M-Mode 中断的干扰）

（7） 等待中断触发；

（8） 中断触发后会进入 RTC_IRQ_Handler，在函数内部关闭定时器中断；

```
#include "platform.h"

void RTC_IRQ_Handler(void);
int main(void)
{
   int_init();
```

```
    timer_int_init();
    timer_int_enable();
    timer_int_bind_handler(RTC_IRQ_Handler);
    set_timer_cmp(50);
    int_enable();
    write_csr(mideleg, 0x00);
    while(1)
        continue;
}

void RTC_IRQ_Handler()
{
    set_timer_cmp(get_timer_val() + 0x10000000);
    timer_int_disable();
}
```

1.4.2 M-Mode 软件中断

M-Mode 软件中断的触发、处理流程如下：

（1） 初始化所有中断，并且将 trap_entry 函数所在地址写入到 mtvec 寄存器中；

（2） 使能 M-Mode 的软件中断；

（3） 绑定软件中断函数；

（4） 将中断委派寄存器 mideleg 置 0（避免对 M-Mode 中断的干扰）；

（5） 打开 M-Mode 中断（否则只开软件中断是无法触发中断的）；

（6） 触发软件中断；

（7） 中断触发后会进入 SOFT_IRQ_Handler，在函数内部关闭软件中断；

```
#include "platform.h"

void SOFT_IRQ_Handler()
{
    soft_int_stop();
}

int main(void)
{
    int_init();
    soft_int_enable();
    soft_int_bind_handler(SOFT_IRQ_Handler);
    write_csr(mideleg, 0x00);
```

```
    int_enable();
    soft_int_start();
    while(1)
        continue;
}
```

2. 模式切换

WH 处理器支持机器模式（M-Mode）、管理员模式（S-Mode）、用户模式（U-Mode），具体如表 1 所示，M-Mode 权限最高，S-Mode 次之，U-Mode 权限最低。

表 1 RISC-V privilege levels

Level	*Encoding*	*Name*	*Abbreviation*
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	-
3	11	Machine	M

WH API 提供从 M-Mode 切换到 S-Mode/U-Mode 函数接口，不过需要注意的是，PMP 会限制 S-Mode/U-Mode 的存储器访问范围，所以在切换至 S-Mode/U-Mode 之前，需要在 M-Mode 状态下设置 PMP，具体的 PMP API 见本文档的 *3.PMP 机制*。

模式切换具有两种状态，一种状态为将 CPU 的指令、数据访问全部切换至特定的特权模式，另外一种是只将 CPU 的数据访问切换至特定的特权模式。

例如，如果只是将 CPU 的数据访问切换至 S-Mode，那么 CPU 的数据访问会受到 PMP 的限制，但是指令还是能够访问 M-Mode 的 CSR 寄存器。如果将指令、数据访问都切换至 S-Mode，那么除了会受到 PMP 的限制外，对于一些 M-Mode 的 CSR，CPU 是无法访问的。

2.1 M-Mode 切换至 S-Mode/U-Mode(指令&数据)

2.1.1 void privilege_drop_to_mode(int privilege_mode, privilege_entry_point_t entry_point, int relocate_sp_flag)

1） 描述：

从 M-Mode 切换至 S-Mode 或者 U-Mode

2） 参数：

privilege_mode：新的特权模式

entry_point：新特权模式的函数入口

relocate_sp_flag: 1 表示需要重定向栈指针，0 表示不需要

2.2 M-Mode 切换至 S-Mode/U-Mode（仅数据）

2.2.1 void mem_privilege_drop_to_mode(int privilege_mode, int relocate_sp_flag)

1） 描述：

从 M-Mode 切换至 S-Mode 或者 U-Mode，当只限于数据访问

2） 参数：

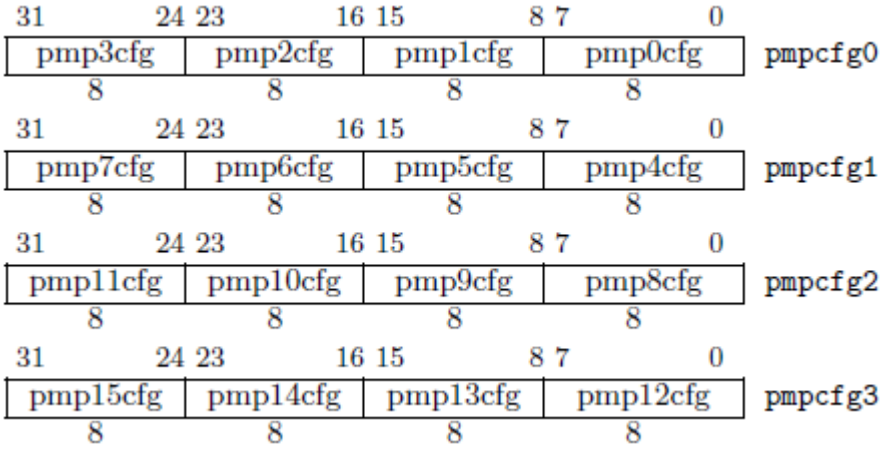
privilege_mode：新的特权模式

relocate_sp_flag: 1 表示需要重定向栈指针，0 表示不需要

3. PMP 机制

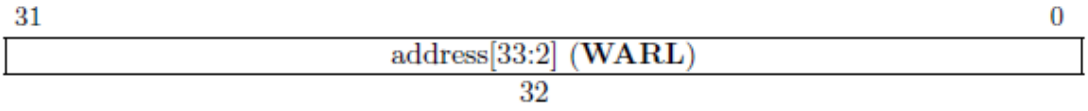
在 WH 处理器中，提供了可选的物理内存保护机制（PMP）来支持安全处理。

PMP 机制会检测运行在 S Mode 或 U Mode 硬件线程的访问权限，S-Mode 和 U-Mode 的相关信息见第二章：模式切换。PMP 机制也会对虚拟内存转换机制的页表访问权限进行检查。PMP 机制由 8 bit 的 CSR 寄存器 CSR.pmpcfg 和 32bit 的 CSR 寄存器 CSR.pmpaddr 对物理内存段进行控制。PMP 机制最多提供了 16 个物理内存段的管理。WH 32bit 处理器的 CSR.pmpcfg 寄存器布局如下：



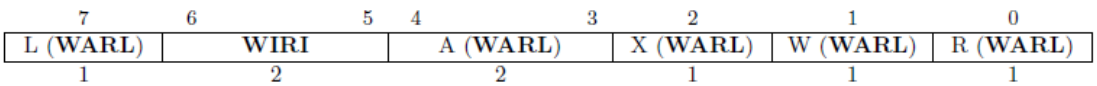
因为 32bit WH 处理器的 CSR 寄存器都是 32bit ,因此每 4 个 8 bit 的 CSR.pmpcfg 寄存器会拼接成一个 CSR 寄存器。

CSR.pmpaddr 的寄存器格式如下：



需要注意的是 CSR.pmpaddr 存放的是字（word）地址，而不是字节地址。因此它可以保证对最少 4 Byte 的内存进行控制。

CSR.pmpcfg 的寄存器格式如下：



1. R,W,X 位如果设置，表明 PMP 管理的物理内存区域允许读，写，执行
2. L 位如果设置，表明 PMP 管理的物理内存区域对 M-Mode 也有效，而且一旦

设置，除非复位，否则无法对当前的 PMP CSR 寄存器进行修改。

3. A 位可以配合 CSR.pmpaddr 对 PMP 管理的物理内存区域大小进行匹配，A 位支持的模式有：

A	Name	Description
0	OFF	Null region (disabled)
1	TOR	Top of range
2	NA4	Naturally aligned four-byte region
3	NAPOT	Naturally aligned power-of-two region, ≥8 bytes

TOR 模式匹配的物理内存区域范围为 $pmpaddr_{i-1} \leqslant addr < pmpaddr_i$，如果 i 为 0，则匹配范围为 $0 <= addr < pmpaddr_0$ 。

NA4 和 NAPOT 匹配的物理内存区域都是 CSR.pmpaddr 自然对齐的。对齐的大小与 CSR.pmpaddr 进行配合（例如 32-byte NAPOT range 要求 pmpaddr 地址是 32-byte 对齐的），如下表：

pmpaddr	pmpcfg.A	Match type and size
aaaa...aaaa	NA4	4-byte NAPOT range
aaaa...aaa0	NAPOT	8-byte NAPOT range
aaaa...aa01	NAPOT	16-byte NAPOT range
aaaa...a011	NAPOT	32-byte NAPOT range
...	...	...
aa01...1111	NAPOT	2^{XLEN}-byte NAPOT range
a011...1111	NAPOT	2^{XLEN+1}-byte NAPOT range
0111...1111	NAPOT	2^{XLEN+2}-byte NAPOT range

根据上表，NA4 和 NAPOT 对 pmpaddr 寄存器配置的方式如下：

首先定义对齐大小的掩码为 AlignMask，2^n 字节对齐大小的 AlignMask 为 2^n-1，例如 8byte 的 AlignMask 为 3’b111。

假设对齐地址为 BaseAddr，则设置 CSR.pmpaddr 的计算方式为：

CSR.pmpaddr = (BaseAddr | (AlignMask >> 1)) >> 2

3.1 函数接口

（1）PMP.h 文件的宏定义：

宏定义	具体值	描述
OFF	0	用于 CSR.pmpcfg.A，为 PMP 地址匹配模式
TOR	1	用于 CSR.pmpcfg.A，为 PMP 地址匹配模式
NA4	2	用于 CSR.pmpcfg.A，为 PMP 地址匹配模式
NAPOT	3	用于 CSR.pmpcfg.A，为 PMP 地址匹配模式
PMP_CFG_LOCK_BIT	7	CSR.pmpcfg L 位最低位所在的位置
PMP_CFG_LOCK_MASK	0x1	CSR.pmpcfg L 位的掩码
PMP_CFG_AREA_BIT	3	CSR.pmpcfg A 位最低位所在的位置
PMP_CFG_AREA_MASK	0x3	CSR.pmpcfg A 位的掩码

PMP_CFG_EXE_BIT	2	CSR.pmpcfg X 位最低位所在的位置
PMP_CFG_EXE_MASK	0x1	CSR.pmpcfg X 位的掩码
PMP_CFG_WRITE_BIT	1	CSR.pmpcfg W 位最低位所在的位置
PMP_CFG_WRITE_MASK	0x1	CSR.pmpcfg W 位的掩码
PMP_CFG_READ_BIT	0	CSR.pmpcfg R 位最低位所在的位置
PMP_CFG_READ_MASK	0x1	CSR.pmpcfg R 位的掩码

（2）pmp.c 文件的函数定义：

```
void PMPConfig(
    int index,                  // PMP 寄存器索引
    unsigned int addr,          // 需要设置的物理内存地址（TOR 时，为内存上边界）
    unsigned int align_mask,    // 内存对齐掩码（若为 TOR，可为任意值）
    unsigned int lock,          // pmpcfg.L
    unsigned int mode,          // pmpcfg.A
    unsigned int r,             // pmpcfg.R
    unsigned int x,             // pmpcfg.X
    unsigned int w);            // pmpcfg.W
```

（3）例程：

将某段物理地址设置为不可写，然后触发写入异常：

（1） 将***[0x4100_0000~0x4200_0000)***区间的地址设置为不可写属性，；

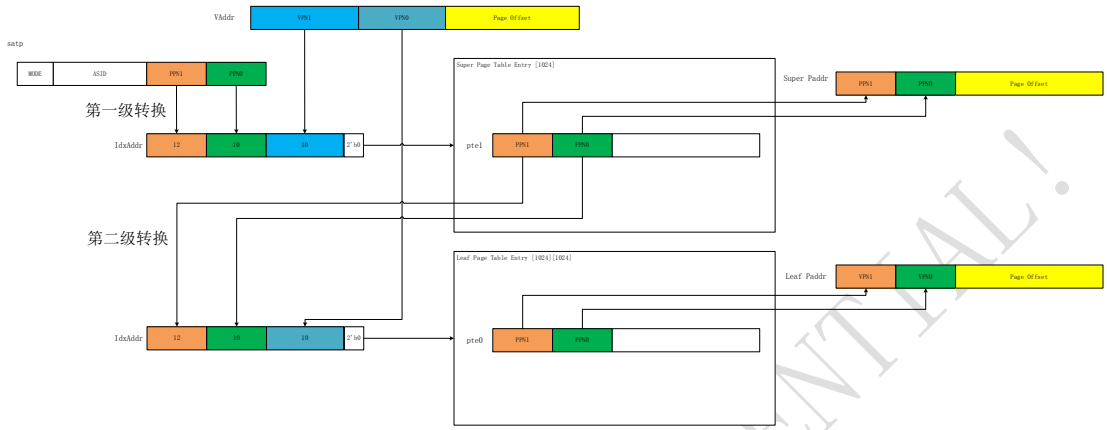
（2） 往 0x4100_0000 地址写值，触发 Store access fault 异常；

```
#include "platform.h"
#include <stdio.h>
unsigned int * mem_ptr = (unsigned int *)(0x41000000);
int main(void)
{
    printf("PMP TEST\r\n");
    pmp_config(0,   0x41000000, 0,   0,       0,    0,  0,  0);
    //       index    |        |    |        |     |   |   |
    //                addr     |    |        |     |   |   |
    //                         mask |        |     |   |   |
    //                              lock     mode   r   x   w
    pmp_config(1,   0x42000000, 0,   1, PMP_TOR,   1,  1,  0);
    //       index    |        |    |        |     |   |   |
    //                addr     |    |        |     |   |   |
    //                         mask |        |     |   |   |
    //                              lock     mode   r   x   w
    printf("Attempting to write to protected address\r\n");
    *mem_ptr = 6;
}
```

4. 虚拟内存转换和保护机制

WH 处理器实现了 sv32 虚拟内存转换系统，sv32 是一个 2 级页表转换系统，关于 sv32 的详细内容，参考文档《 The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10》

Sv32 虚拟地址转换物理地址的过程如下图：



Sv32 是一个典型的二级分级页表，通过两级索引获取到对应的物理地址：

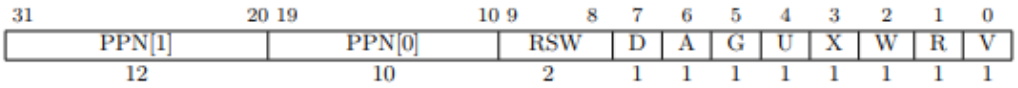
第一级

{satp.PPN , vaddr.VPN1 , 2'b0} 获取到超页表的 PTE

第二级

{SPTE.PPN,vaddr.VPN0,2'b0} 获取到叶页表的 PTE

在内存中定义了两个数组：Super Page Table Entry(PTE)和 Leaf Page Table Entry (PTE) 来维护物理地址空间的权限，每个叶 PTE 管理 4K 内存。PTE 的格式如下如所示：



其中：

V 表示该 PTE 有效

R 表示可读

W 表示可写

X 表示可执行

U 表示该 PTE 管理的内存为 S-Mode page，
S-Mode 如果想要访问，必须 sstatus.SUM 置 1

G 表示全局映射

A 表示虚拟页表已经被访问过

D 表示虚拟页表已经被写过

如果 R,W,X 同时为 0，表明该 PTE 为超页表，指向下一级页表。

（1）PTE.h 文件的宏定义：

宏定义	具体值	描述
PADDR_PPN1_BIT	22	PADDR PPN1 在 PADDR 的位置
PADDR_PPN1_MASK	0xFFF	PADDR PPN1 的掩码
PADDR_PPN0_BIT	12	PADDR PPN0 在 PADDR 的位置

PADDR_PPN0_MASK	0x3FF	PADDR PPN0 的掩码
VADDR_VPN1_BIT	22	VADDR VPN1 在 VADDR 的位置
VADDR_VPN1_MASK	0x3FF	VADDR VPN1 的掩码
VADDR_VPN0_BIT	12	VADDR VPN0 在 VADDR 的位置
VADDR_VPN0_MASK	0x3FF	VADDR VPN0 的掩码
PTE_PPN1_BIT	20	PTE PPN1 在 PTE 的位置
PTE_PPN1_MASK	0xFFF	PTE PPN1 的掩码
PTE_PPN0_BIT	10	PTE PPN0 在 PTE 的位置
PTE_PPN0_MASK	0x3FF	PTE PPN0 的掩码
PTE_DIRTY_BIT	7	PTE Dirty 在 PTE 的位置
PTE_DIRTY_MASK	0x1	PTE Dirty 的掩码
PTE_ACCESS_BIT	6	PTE Access 在 PTE 的位置
PTE_ACCESS_MASK	0x1	PTE Access 的掩码
PTE_GLOBAL_BIT	5	PTE Global 在 PTE 的位置
PTE_GLOBAL_MASK	0x1	PTE Global 的掩码
PTE_USER_BIT	4	PTE User 在 PTE 的位置
PTE_USER_MASK	0x1	PTE User 的掩码
PTE_EXECUTEABLE_BIT	3	PTE Executable 在 PTE 的位置
PTE_EXECUTEABLE_MASK	0x1	PTE Executable 的掩码
PTE_WRITEABLE_BIT	2	PTE Writeable 在 PTE 的位置
PTE_WRITEABLE_MASK	0x1	PTE Writeable 的掩码
PTE_READABLE_BIT	1	PTE Readable 在 PTE 的位置
PTE_READABLE_MASK	0x1	PTE Readable 的掩码
PTE_VALID_BIT	0	PTE Valid 在 PTE 的位置
PTE_VALID_MASK	0x1	PTE Valid 的掩码
PAGE_OFFSET_MASK	0xFFF	页表偏移的掩码
SATP_MODE_BIT	31	SATP MODE 在 SATP 的位置
SATP_MODE_MASK	0x1	SATP MODE 的掩码
SATP_ASID_BIT	22	SATP ASID 在 SATP 的位置
SATP_ASID_MASK	0x1FF	SATP ASID 的掩码
SATP_PPN_BIT	0	SATP PPN 在 SATP 的位置
SATP_PPN_MASK	0x3FFFFF	SATP PPN 的掩码

（2）PTE.c 文件的函数定义：

1. 页表初始化函数：void sv32_init(void)

在 API 中定义一个全局超页表 super_page_sv32[1024]和全局叶页表 leaf_page_sv32[1024][1024]，本函数的功能是对这两个页表进行初始化。

2. 虚拟内存系统使能函数 void sv32_enable(void)

该函数的功能是使能 WH 处理器的虚拟内存系统

3. 虚拟内存系统禁用函数 void sv32_disable(void)

该函数的功能是禁用 WH 处理器的虚拟内存系统

4. 地址映射函数：该函数的功能是根据物理地址获取对应的虚拟地址，并将其相关权限记录在 PTE 中。函数返回虚拟地址。

```
void * sv32_addr_map(  int Level ,            // 1：使用一级页表，2：使用 2 级页表
                       void *Paddr,           // 物理地址
                       unsigned int Exe,      // 执行权限
                       unsigned int Write,    // 可写权限
                       unsigned int Read,     // 可读权限
                       unsigned int User,     // User 权限
                       unsigned int Global) // 全局映射权限
```