Andes riscv文档代码学习记录分享

> 参考代码：demo-plic-V5、demo-smp-V5、demo-cache-V5、demo-cache-lock-V5、demo-pma-V5、demo-pmp-V5、demo-mmu-V5。 参考文档：AndeSight_v5.3_BSP_User_Manual_UM275_V1.1.pdf、AndesCore_AX45MPV_DS220_V1.4.pdf、其他riscv培训文档或者教材。

一、命令行debug

安装了andesight后可以通过ide和命令行两种方式debug。**命令行debug模拟器**参考AndeSight_v5.3_BSP_User_Manual_UM275_V1.1.pdf的3.1小节，**命令行debug ice target板**参考AndeSight_v5.3_BSP_User_Manual_UM275_V1.1.pdf的3.2小节。

二、启动方式

andes riscv startup demos有xip、burn、load三种启动方式，可以在demo/config.h中选择。xip方式是从rom/flash启动并一直在rom/flash运行，burn方式是从rom/flash启动然后跳到memory运行，load方式是通过gdb将程序load到memory然后从memory运行。andes a350平台物理地址空间的0x0-0x7fffffff是ram，紧接着ram的0x80000000-0x87ffffff是flash。

Table 259: AE350 Memory Map

Address		Description
Begin	End	
0x00000000	0x7FFFFFFF	RAM Bridge
0x80000000	0x87FFFFFF	SPI1 AHB Memory/Reset Vector

xip启动

> 从reset_vector开始运行

xip启动的链接脚本是ae350-xip.ld，生成连接脚本的andes sag文件是ae350-xip.sag。xip模式先从flash开始启动运行，启动后再将数据段从flash拷贝到memory，代码段始终在flash中。

ae350-xip.sag

```
USER_SECTIONS   .vector_table

EXEC 0x80000000          //加载区EXEC的lma为0x80000000
{
    FLASH 0x80000000     //执行区FLASH的vma为0x80000000
    {
        * (+ISR,+RO)     //所有目标文件的ISR段(.nds_vector),只读代码和只读数据
    }
}

DATA +0 ALIGN 32         //加载区DATA的lma为align_of_32(lma(EXEC) + sizeof(EXEC) + 0)
```

```
{
    RAM 0x00000000      //执行区RAM的vma为0x00000000
    {
        LOADADDR NEXT __data_lmastart   // __data_lmastart = lma(RAM) = lma(RAM)
        ADDR NEXT __data_start          // __data_start = vma(RAM) = 0x0
        * (.vector_table)
        * (+RW,+ZI)                     //所有目标文件的.data和.bss段
        STACK = 0x08000000
    }
}
```

EXEC加载区的lma和FLASH执行区的vma都是0x80000000，表示将程序烧写到0x80000000，不用搬运可以直接运行。

ae350-xip.ld

```
...
    /* FLASH (EXEC) */
    . = ALIGN(0x80000000, 16);
    FLASH . : AT(EXEC_LMA_) {}
    . = ALIGN(ALIGNOF(.nds_vector));
    .nds_vector     : { KEEP(*(.nds_vector )) KEEP(*(SORT(.nds_vector.* ))) }
...
```

从链接脚本看vma 0x80000000开始的第一个段是.nds_vector，因此系统上电后cpu从start.S的reset_vector开始运行。

start.S

```
#include "core_v5.h"
    .section .nds_vector, "ax"
    .global reset_vector

reset_vector:
    /* Decide whether this is an NMI or cold reset */
    csrr t0, mcause
    bnez t0, nmi_handler

    .global _start
    .type _start,@function

_start:
    ...

    /* Initialize stack pointer */
    la t0, _stack       //链接脚本中指定0x08000000
    mv sp, t0           //设置栈
    ...

    /* Initial machine trap-vector Base */
    la t0, __vectors
    csrw mtvec, t0      //设置trap基地址
```

```
    /* Enable vectored external PLIC interrupt */
    csrsi mmisc_ctl, 2  //打开可以使能plic Vector mode的开关

    /* System reset handler */
    call reset_handler  //重定位,设置复位地址,使能plic Preempt and Vector mode,跳到main
    ...
```

start.S主要负责设置栈，然后跳到c语言函数reset_handler。

reset.c

```
__attribute__((weak)) void reset_handler(void)
{
    extern int main(void);

    c_startup();    //将.data从flash拷贝到memory，初始化memory的.bss
    system_init();   //设置复位地址，使能plic preempt和vector模式
    ...
    main();     //跳到应用主函数
}
```

burn启动

> 从boot_loader()开始运行

burn启动的链接脚本是ae350.ld，生成连接脚本的andes sag文件是ae350.sag。burn模式先从flash开始启动运行，然后将代码段和数据段从flash拷贝到memory，最后再从flash的代码段跳转到memory的代码段运行。burn模式将代码分为bootloader、loader、app三个部分，执行顺序是bootloader->loader->app。

ae350.sag

```
USER_SECTIONS   .bootloader
USER_SECTIONS   .loader
USER_SECTIONS   .vector_table

HEAD 0x00000000             //加载区HEAD的lma为0x0
{
    BOOTLOADER 0x80000000   //执行区BOOTLOADER的vma为0x80000000
    {
        ADDR __flash_start              // __flash_start = vma(BOOTLOADER)
        * KEEP (.bootloader)
    }
}

LDSECTION +0 ALIGN 8        //加载区LDSECTION的lma为align_of_8(lma(HEAD) + sizeof(HEAD) + 0)
{
    LOADER 0x0007FF00       //执行区LOADER的vma为0x0007FF00
    {
        LOADADDR NEXT __loader_lmastart // __loader_lmastart = lma(LOADER) = lma(LDSECTION)
        ADDR NEXT __loader_start        // __loader_start = vma(LOADER) = 0x0007FF00
```

```
        * KEEP (.loader)
        LOADADDR __loader_lmaend        // __loader_lmaend = lma(LOADER) + sizeof(LOADER)
    }
}

MEM +0 ALIGN 256            //加载区MEM的lma为align_of_256(lma(LDSECTION) +
sizeof(LDSECTION) + 0)
{
    RAM 0x00000000          //执行区RAM的vma为0x0
    {
        LOADADDR NEXT __text_lmastart   // __text_lmastart = lma(RAM) = lma(MEM)
        ADDR NEXT __text_start          // __text_start = vma(RAM) = 0x0
        * (.vector_table)
        * (+ISR,+RO,+RW,+ZI)
        STACK = 0x08000000
    }
}
```

sag文件中的bootloader的加载区HEAD的lma为0x0，它是为了计算loader和app在flash上的偏移__loader_lmastart和__text_lmastart，bootloader作为最先启动的软件，它需要烧到flash的0x80000000，因此loader和app在flash上的地址分别为0x80000000+__loader_lmastart和0x80000000+__text_lmastart。

ae350.ld

```
...
    /* BOOTLOADER (EXEC) */
    . = ALIGN(0x80000000, 16);
    BOOTLOADER .    : AT(HEAD_LMA_) {}
    __flash_start = .;
    .bootloader     : AT(ALIGN(LOADADDR(BOOTLOADER), ALIGNOF(.bootloader)))
        { KEEP(*(.bootloader )) }
...
```

loader.c/boot_loader()

```
/* The entry function when boot, executing on ROM/FLASH. */
void __attribute__((naked, no_execit, no_profile_instrument_function,
section(".bootloader"))) boot_loader(void) {

    register unsigned long *src_ptr, *dst_ptr;
    register unsigned long i, size;

    ...

    /* Copy loader() code to VMA area (On memory) */
    size = ((unsigned long)&__loader_lmaend - (unsigned long)&__loader_lmastart +
(sizeof(long) - 1)) / sizeof(long);   //loader的大小
    src_ptr = (unsigned long *)((unsigned long)&__flash_start + (unsigned
long)&__loader_lmastart); //loader的实际flash起始地址(不是lma)
    dst_ptr = (unsigned long *)&__loader_start; //loader的vma起始地址
```

```
    for (i = 0; i < size; i++)
        *dst_ptr++ = *src_ptr++;

    register void (*ldr_entry)(void) = (void *)&__loader_start;
    ldr_entry();    //跳到loader
}
```

cpu上电从boot_loader()开始执行，boot_loader()中的变量都放寄存器里不用栈，所以在没有初始化栈的时候可以运行。boot_loader()将loader从flash拷贝到链接脚本指定的vma，然后跳到loader入口loader()。

loader.c/loader()

```
void __attribute__((naked, no_execit, no_profile_instrument_function, section(".loader")))
loader(void) {

    register unsigned long *src_ptr, *dst_ptr;
    register unsigned long i, size;

    /* Copy code bank to VMA area */
    size = ((unsigned long)&_edata - (unsigned long)&__text_start + (sizeof(long) - 1)) /
sizeof(long); //app的大小
    //app的实际flash起始地址(不是lma)
    src_ptr = (unsigned long *)((unsigned long)&__flash_start + (unsigned
long)&__text_lmastart);
    dst_ptr = (unsigned long *)&__text_start;   //app的vma起始地址

    for (i = 0; i < size; i++)
        *dst_ptr++ = *src_ptr++;

    /* Go to entry function */
    _start();   //跳到app的入口地址_start
}
```

loader()和boot_loader()方法一样，将app从flash拷贝到链接脚本指定的vma，然后跳到app入口_start。

load启动

> 从_start开始运行

load启动的链接脚本、生成连接脚本和burn启动共用，但是load模式没有bootloader和loader。load模式由调试器gdb负责将app从文件系统加载到到memory，然后直接跳转到app的入口地址_start。

三、复位异常中断

20.1 Reset

When the coreN_reset_n input signal of the processor is deasserted, the following operations are applied:

- CSRs are set to their reset values.
- All integer registers (listed in Table 77) are set to zero.
- BTB is initialized.
- Program execution starts with the address specified by the coreN_reset_vector input signal.

24.9.12 Hart 0 Reset Vector Register (HART0_RESET_VECTOR_LO) (0x50)

This register controls the value driven to the hart0_reset_vector[31:0] input signal of the AX45MPV processor.

Field Name	Bits	Description	Type	Reset
RESET_VECTOR	[31:0]	Entry address upon processor reset	RW	0x80000000

系统上电后riscv cpu会从0x80000000开始运行，之后可以修改复位向量寄存器的值使得cpu复位后从不同的地址开始运行。异常和中断在riscv中都属于trap，异常是指令执行同步产生的，中断异步产生。riscv的中断又分为timer interrupt、software interrupt、externel interrupt三大类。

复位异常中断初始化

初始化代码调用路径

```
start.S
    ->reset_handler
        ->system_init
        ->main
```

start.S

```
#include "core_v5.h"
    .section .nds_vector, "ax"
    .global reset_vector

reset_vector:
    /* Decide whether this is an NMI or cold reset */
    csrr t0, mcause
    bnez t0, nmi_handler

    .global _start
    .type _start,@function

_start:
    ...

    /* Initialize stack pointer */
    la t0, _stack
```

```
    mv sp, t0
    ...

    /* Initial machine trap-vector Base */
    la t0, __vectors
    csrw mtvec, t0      //设置trap基地址,此时可以发生异常,但不会发生中断

    /* Enable vectored external PLIC interrupt */
    csrsi mmisc_ctl, 2  //打开可以使能plic Vector mode的开关

    /* System reset handler */
    call reset_handler  //重定位,设置复位地址,使能plic Preempt and Vector mode,跳到main

    /* Infinite loop, if returned accidently */
1:  j 1b

    .weak __platform_init
__platform_init:
    ret

    .weak nmi_handler
nmi_handler:
1:  j 1b

    .text

    .global default_irq_entry
    .align 2

default_irq_entry:
1:  j 1b

    .weak trap_handler
trap_handler:
1:  j 1b

    .macro INTERRUPT num
    .weak entry_irq\num
    .set entry_irq\num, default_irq_entry
    .long entry_irq\num
    .endm

    /* Vector table
     * NOTE:
     * The Vector Table base alignment requirement has to be :
     * " 2^ceiling(log2(N)) x 4 " bytes,
     * if the PLIC device supports N interrupt sources.
     */
#define VECTOR_NUMINTRS         32
    .section .vector_table, "a"

    .global __vectors
    .balign 4096
```

```
__vectors:
    /* Trap vector */
    .long trap_entry

    /* PLIC interrupt vector */
    .altmacro
    .set irqno, 1
    .rept VECTOR_NUMINTRS
    INTERRUPT %irqno
    .set irqno, irqno+1
    .endr
```

start.S中对异常中断的设置只有trap基地址设置为__vectors，此时中断还没打开，不会发生中断，但是异常可以。然后剩下对中断的设置在reset_handler调用的system_init和main函数中。

ae350.c

```
void system_init(void)
{
    /* Reset the CPU reset vector for this program. */
    AE350_SMU->RESET_VECTOR = (unsigned int)(long)reset_vector; //设置复位地址

    /* Enable PLIC features */
    if (read_csr(NDS_MMISC_CTL) & (1 << 1)) {
        //使能plic preempt and vector mode
        __nds__plic_set_feature(NDS_PLIC_FEATURE_PREEMPT | NDS_PLIC_FEATURE_VECTORED);
    } else {
        /* External PLIC interrupt is NOT vectored */
        __nds__plic_set_feature(NDS_PLIC_FEATURE_PREEMPT);
    }

    ...
}
```

system_init设置复位寄存器为reset_vector，如果之后不是整个系统复位，只复位cpu，那么cpu会从reset_vector开始运行。然后使能plic的preempt和vector模式。剩下的中断设置在main()中。

main.c

```
int main(void)
{
    volatile int local_cnt = 0;

    //关闭第二级timer、software、externel中断开关。
    clear_csr(NDS_MIE, MIP_MEIP | MIP_MTIP | MIP_MSIP);

    setup_mtimer(); //设置timer超时时间和打开第二级timer中断开关
    setup_mswi();   //设置software中断优先级并打开第二级software中断开关和第三级开关
    setup_pit_timer(PIT_TIMER_PERIOD);  //设置外部定时器周期和对应的外部中断权限和打开第三级开关

    //打开第二级timer、software、externel中断开关。
```

```
    set_csr(NDS_MIE, MIP_MEIP | MIP_MTIP | MIP_MSIP);

    //打开第一级全局中断开关
    set_csr(NDS_MSTATUS, MSTATUS_MIE);

    while(1) {
        if (trigger_mswi_flag) {
            trigger_mswi_flag = 0;
            trigger_mswi();     //设置interrupt pending寄存器触发software中断
        }
        local_cnt++;
    }

    return 0;
}
```

main()负责最后需要使用的中断的各级开关的打开，然后跑应用，中断发生后处理中断。

异常中断处理

发生异常或中断时硬件会自动做： 1）将trap前的pc保存到mepc寄存器。 2）将trap前的处理器模式保存到mstatus.MPP，trap前的mstatus.MIE保存到mstatus.MPIE，mstatus.MIE设置为0。 3）mcause和mtval根据trap原因更新相应的值。

从异常或中断返回(通过软件mret返回)时硬件自动做： 1）设置pc为mepc的值。 2）当前处理器模式恢复为mstatus.MPP，mstatus.MIE恢复为mstatus.MPIE。

当同时发生异常，externel中断，timer中断，sw中断时，cpu先处理externel中断，然后sw中断，然后timer中断最后异常。他们四个之间不能抢占，抢占只能发生在使能了抢占特性的externel中断之间。

```
//异常，timer中断，software中断入口
trap_entry
    ->mtime_handler
    ->mswi_handler
    ->syscall_handler   //系统调用也是异常
    ->except_handler

//externel中断入口
entry_irq1
    ->NESTED_IRQ_ENTER
    ->rtc_period_irq_handler
    ->NESTED_VPLIC_COMPLETE_INTERRUPT
    ->NESTED_IRQ_EXIT

entry_irq2
    ...
entry_irq3
    ...
...
```

在start.S中将__vectors设置为trap基地址，因此发生异常，timer中断，software中断会跳到__vectors[0]即trap_entry，发生externel中断会跳到__vectors[i]即entry_irqi。

异常，timer中断，software中断处理

```
void trap_entry(void) __attribute__ ((interrupt ("machine") , aligned(4)));
void trap_entry(void)
{
    long mcause = read_csr(NDS_MCAUSE);
    long mepc = read_csr(NDS_MEPC);
    long mstatus = read_csr(NDS_MSTATUS);
    ...

    if ((mcause & MCAUSE_INT) && ((mcause & MCAUSE_CAUSE) == IRQ_M_TIMER)) {
        mtime_handler();
    } else if ((mcause & MCAUSE_INT) && ((mcause & MCAUSE_CAUSE) == IRQ_M_SOFT)) {
        mswi_handler();
        /* Machine SWI is connected to PLIC_SW source 1 */
        __nds__plic_sw_complete_interrupt(1);   //clear中断源
    } else if (!(mcause & MCAUSE_INT) && ((mcause & MCAUSE_CAUSE) == TRAP_M_ECALL)) {
        /* Machine Syscal call */
        __asm volatile(
                "mv a4, a3\n"
                "mv a3, a2\n"
                "mv a2, a1\n"
                "mv a1, a0\n"
                "mv a0, a7\n"
                "call syscall_handler\n"
                : : : "a4"
        );
        mepc += 4;  //如果是系统调用则返回到下一条指令
    } else {
        /* Unhandled Trap */
        mepc = except_handler(mcause, mepc);
    }

    /* Restore CSR */
    write_csr(NDS_MSTATUS, mstatus);
    write_csr(NDS_MEPC, mepc);
}
```

需要注意只有software中断需要做clear中断源的处理，而timer中断不用。因为ae350的software中断连接的是另一个plic的中断源。还有一个要注意的是trap_entry()结束后要返回trap前的状态，所以trap_entry()的最后应该是mret指令，但是从函数源代码中没有看到，原因在于trap_entry()指定了interrupt属性会在编译的时候最后插入mret指令，通过反汇编文件可以验证。

externel中断处理

```
void __attribute__ ((interrupt)) entry_irqX(void)
{
    NESTED_IRQ_ENTER();           // save mepc and mstatus
    XXX_irq_handler();               // Call irqx ISR
    NESTED_VPLIC_COMPLETE_INTERRUPT(IRQ_X_SOURCE);  //clear source
    NESTED_IRQ_EXIT();           // restore mepc and mepc
}
```

externel中断处理主要一个不同是要清中断，这是plic规定的，和arm的gic类似。其他的和上面的trap_entry()一样，先保存mepc和mstatus，然后处理中断，处理完后再恢复mepc和mstatus，这样是为了防止中断抢占破坏mepc和mstatus导致无法正确返回。

四、SMP

demo给的例子是双核启动，开机或复位后hart0和hart1都会运行，但是hart1很早就通过wfi进入等待状态，hart0设置好自己的环境以及双核公用环境后通过software interrupt唤醒hart1，最后hart1设置自己的环境然后正常运行。

start.S

```
#include "core_v5.h"
#define STACKSIZE        0x100000

    .section .init
    .global reset_vector

reset_vector:
    /* Decide whether this is an NMI or cold reset */
    csrr t0, mcause
    bnez t0, nmi_handler

    .global _start
    .type _start,@function

_start:
    ...

    /* Initialize stack pointer */
    la t0, _stack   //链接脚本中指定0x08000000
    mv sp, t0

    /* get cpu id */
    csrr t0, mhartid

    /* sp = _stack - (cpuid * STACKSIZE) */
    li t1, STACKSIZE
    mul t0, t0, t1
    sub sp, sp, t0  //hart0的栈基地址为0x08000000, hart1的sp栈基地址为0x08000000-0x100000

    ...
```

```
    /* Initial machine trap-vector Base */
    la t0, trap_entry
    csrw mtvec, t0

    /* If not first cpu core, jump to other cpu entry directly */
    csrr a0, mhartid
    bnez a0, other_hart_wfi     //如果不是第一个hart则跳到other_hart_wfi

    /* Do system low level setup. It must be a leaf function */
    call __platform_init

    /* System reset handler */
    call reset_handler

    /* Infinite loop, if returned accidently */
1:  j 1b

    .weak __platform_init
__platform_init:
    ret

    .weak nmi_handler
nmi_handler:
1:  j 1b

other_hart_wfi:
    li t0, MIP_MSIP
    csrrs t0, mie, t0   //打开mie的software开关
    li t0, MSTATUS_MIE
    csrrc t0, mstatus, t0   //关闭mstatus的全局中断开关
    wfi         //mstatus.MIE为0情况下也能被唤醒,只要mie寄存器没关闭,参考21.2
other_hart_entry:
    la a0, other_hart_boot_flag
    lw a0, 0(a0)
    li a1, 0xa55a
    bne a0, a1, other_hart_entry
    call reset_handler
1:  j 1b

    .section .data
    .global other_hart_boot_flag
    .align 3
other_hart_boot_flag:
    .dword 0
```

双核处理器上电后同时启动，运行一样的代码，但是在代码中走不同的分支。在start.S中hart0和前文的单核demo一样跑到reset_handler进一步初始化，而hart1在start.S中会跳到other_hart_wfi等待hart0准备好后发送核间中断唤醒，hart1被唤醒后再跳到reset_handler进一步初始化。hart0除了初始化自己的还要初始化和hart1公用的东西，而hart1只需要初始化自己的东西。

reset.c

```
__attribute__((weak)) void reset_handler(void)
{
    extern int main(void);

    switch (read_csr(NDS_MHARTID)) {
        case 0:
            c_startup();    //共用的数据段重定位
            system_init();  //hart0的复位地址设置和共用的plic特性设置
            break;
        case 1:
            system_init();  //hart1的复位地址设置
            break;
    }
    main();     //跳到main()
}
```

hart0会先后运行c_startup()、system_init()、main()来做私有和公共的初始化，而hart1被唤醒后会先后运行system_init()、main()做私有的初始化。

ae350.c

```
void c_startup(void)
{
#define MEMCPY(des, src, n)     __builtin_memcpy ((des), (src), (n))
#define MEMSET(s, c, n)         __builtin_memset ((s), (c), (n))
    /* Data section initialization */
    extern char  _edata, _end;
    unsigned int size;
#ifdef CFG_XIP
    extern char __data_lmastart, __data_start;

    //将数据段从flash拷贝到memory
    size = &_edata - &__data_start;
    MEMCPY(&__data_start, &__data_lmastart, size);
#endif

    //memory中的bss清0
    size = &_end - &_edata;
    MEMSET(&_edata, 0, size);
}

void system_init(void)
{
    switch (read_csr(NDS_MHARTID)) {
        case 0:
            /* Reset the mhart0 reset vector of this program. */
            DEV_SMU->HART_RESET_VECTOR[0] = (unsigned int)(long)reset_vector;

            /* Enable PLIC features */
            if (read_csr(NDS_MMISC_CTL) & (1 << 1)) {
                /* External PLIC interrupt is vectored */
                __nds__plic_set_feature(NDS_PLIC_FEATURE_PREEMPT |
NDS_PLIC_FEATURE_VECTORED);
```

```
            } else {
                /* External PLIC interrupt is NOT vectored */
                __nds__plic_set_feature(NDS_PLIC_FEATURE_PREEMPT);
            }

            /* Enable misaligned access and non-blocking load. */
            set_csr(NDS_MMISC_CTL, (1 << 8) | (1 << 6));

            /* Initial UART port */
            uart_init(38400);
            break;
        case 1:
            /* Reset the mhart1 reset vector of this program. */
            DEV_SMU->HART_RESET_VECTOR[1] = (unsigned int)(long)reset_vector;

            /* Enable misaligned access and non-blocking load. */
            set_csr(NDS_MMISC_CTL, (1 << 8) | (1 << 6));
            break;
    }
}
```

hart0在reset_handler中调用c_startup做数据段的重定位，以及system_init中对plic特性的设置，串口的设置，这些都是hart0和hart1共用的部分，同时hart0也在system_init做复位向量和cpu内存访问特性的私有设置，而hart1只在system_init中做复位向量和cpu内存访问特性的私有设置。

main.c

```
void setup_mswi_hart0(void)
{
    /* Core mhart0 init core mhart0 PLIC_SW source 1 */
    __nds__plic_sw_set_priority(1, 1);      //设置plic_sw source 1的优先级为1
    ___nds__plic_sw_enable_interrupt(0, 1); //使能plic_sw source 1通向hart0的路由
    ___nds__plic_sw_disable_interrupt(0, 2);//关闭plic_sw source 2通向hart0的路由
}

void setup_mswi_hart1(void)
{
    /* Core mhart0 init core mhart1 PLIC_SW source 2 */
    __nds__plic_sw_set_priority(2, 1);      //设置plic_sw source 2的优先级为1
    ___nds__plic_sw_enable_interrupt(1, 2); //使能plic_sw source 2通向hart1的路由
    ___nds__plic_sw_disable_interrupt(1, 1);//关闭plic_sw source 1通向hart1的路由
}

void other_hart_up(void)
{
    other_hart_boot_flag = 0xa55a;
    mb();   //asm volatile ("fence" ::: "memory")
    /* Core mhart0 trigger IPI to awake core mhart1 from wfi */
    HAL_IPI_SEND(1);    //25.5.4
}

//定时处理内部定时器中断
int start_hart1(void)
```

```
{
    //清software interrupt,因为hart1被唤醒后到现在还没清中断
    __nds__plic_sw_claim_interrupt();
    __nds__plic_sw_complete_interrupt(2);

    /* Disable the Machine external, timer and software interrupts until setup is done */
    clear_csr(NDS_MIE, MIP_MEIP | MIP_MTIP | MIP_MSIP);

    /* Setup machine mtime */
    setup_mtimer();

    /* Enable the Machine Timer/Software interrupt bit in MIE. */
    set_csr(NDS_MIE, MIP_MTIP | MIP_MSIP);

    /* Enable interrupts in general. */
    set_csr(NDS_MSTATUS, MSTATUS_MIE);

    while(1);

    return 0;
}

//定时处理外部定时器中断
int start_hart0(void)
{
    /* Disable the Machine external, timer and software interrupts until setup is done */
    clear_csr(NDS_MIE, MIP_MEIP | MIP_MTIP | MIP_MSIP);

    /* Setup PLIC PIT interrupt */
    setup_pit_timer(PIT_TIMER_PERIOD);

    /* Enable the Machine External/Software interrupt bit in MIE. */
    set_csr(NDS_MIE, MIP_MEIP | MIP_MSIP);

    /* Enable interrupts in general. */
    set_csr(NDS_MSTATUS, MSTATUS_MIE);

    while(1);

    return 0;
}

int main(void)
{
    int mhartid = read_csr(NDS_MHARTID);

    if (mhartid == 0) {
        /* Setup core mhart0 PLIC_SW source 1 */
        setup_mswi_hart0(); //设置PLIC_SW source 1优先级，以及可以路由到hart0

        /* Setup core mhart1 PLIC_SW source 2 */
        setup_mswi_hart1(); //设置PLIC_SW source 2优先级，以及可以路由到hart1
```

```
        /* Core mhart0 finish the initialization, start other core mhart1 */
        other_hart_up();    //触发PLIC_SW source 2
    }

    switch (mhartid) {
        case 0:
            start_hart0();  //跑hart0自己的业务
            break;
        case 1:
            start_hart1();  //跑hart1自己的业务
            break;
    }

    return 0;
}
```

最后在main()中hart0先设置好两个software interrupt的路由然后通过其中一个software interrupt唤醒hart1，hart0最后进入start_hart0处理自己的业务。而hart1被唤醒后一路跑到main后直接进入start_hart1处理自己的业务。

五、Cache

demo显示的ax45mpv处理器使用的l1-icache是2路组相联，cacheline 32b，size 32kb，vipt。l1-dcache是4路组相联，cacheline 32b，size 32kb，pipt。没有使用l2 cache，如果启用l2 cahce，那是pipt的unify cache。下文用icache表示l1-icache，dcache表示l1-dcache。

cache operation

按功能类型划分cache操作有invalidate、clean、flush、lock四种，icache不支持clean和flush。按操作类型ax45mpv的cache操作有cctl和cmo两种，cctl是操作csr寄存器来操作cache，cmo是通过cmo扩展指令来操作cache，除了这两类操作外还有fence/fence.i指令也会影响cache。l1和l2 cache有各自的cctl操作，l1的cctl支持invalidate、clean、flush、lock四种操作，l2的cctl支持invalidate、clean、flush三种操作。cmo会根据需要对l1或l2做对应的cache操作。fence/fence.i是内存屏障指令，对cache有隐式影响。cctl、cmo、fence/fence.i对cache具体操作和说明如下：

l1 cctl

l1的cctl主要有IX和VA两种类型，IX直接指定way，index，tag来操作cache line，VA通过虚拟地址来操作对应的cache line。参考10.6。

```
// 1. Invalidating Cache Blocks (L1D_VA_INVAL, L1I_VA_INVAL, L1D_IX_INVAL, L1I_IX_INVAL)
invalidate对应的l1 cache lines，locked的cache lines会被unlock和invalidated。

// 2. Writing Back Cache Blocks (L1D_VA_WB, L1D_IX_WB, L1D_WB_ALL)
clean对应的l1 cache lines，locked的cache lines也会clean但保持locked状态。

// 3. Writing Back & Invalidating Cache Blocks (L1D_VA_WBINVAL, L1D_IX_WBINVAL,
L1D_WBINVAL_ALL)
flush对应的l1 cache lines，即先clean然后invalidate。locked的cache lines会unlock。
```

```
// 4. Filling and Locking Cache Blocks (L1D_VA_LOCK, L1I_VA_LOCK)
lock对应的l1 cache lines。首先fill cache lines然后lock。locked的cache lines不会被其他va miss换出。lock的icache也不会被自己va fetch ins换出，至于dcache会不会被自己va访问换出，还需做实验验证。lock个人理解是给关键的性能代码用的，dcache应该不用这东西。

// 5. Unlocking Cache Blocks (L1D_VA_UNLOCK, L1I_VA_UNLOCK)
unlock对应的l1 cache lines。
```

l2 cctl

l2的cctl主要有IX和PA两种类型，IX直接指定way，index，tag来操作cache line，PA通过物理地址来操作对应的cache line。l2 cache不支持lock。参考11.4。

```
// 1. Invalidating Cache Blocks (L2_PA_INVAL, L2_IX_INVAL)
invalidate对应的l2 cache lines。

// 2. Writing Back Cache Blocks (L2_PA_WB, L2_IX_WB)
clean对应的l2 cache lines。

// 3. Writing Back & Invalidating Cache Blocks (L2_PA_WBINVAL, L2_IX_WBINVAL, L2_WBINVAL_ALL)
flush对应的l2 cache lines。
```

cmo

cmo扩展指令即可能操作l1也可能操作l2 cache，不支持icache的操作和lock操作，但是额外支持预取和清零操作。如果在不使用cctl的前提下要操作icache需要fence.i指令。需要lock操作只能使用cctl并且只能lock l1 cache。

```
// 1. Invalidating Cache Blocks (cbo.inval rs1)
invalidate rs1对应的cache line。

// 2. Cleaning Cache Blocks (cbo.clean rs1)
clean rs1对应的cache line。

// 3. Flushing Cache Blocks (cbo.flush rs1)
flush rs1对应的cache line。

// 4. Zeroing Cache Blocks (cbo.flush rs1)
用0填充rs1对应的cache line。

// 5. Prefetching Cache Blocks (prefetch.i offset(rs1), prefetch.r offset(rs1), prefetch.w offset(rs1))
分别表示预取指令到cache，读操作预取数据到cache，写操作预取数据到cache。
```

fence/fence.i

fence/fence.i是内存屏障指令，在使能了cache后会对cache有影响。fence指令对l1-icahce和l1-dcache没有影响，fence.i会invalidate l1-icache，write back l1-dcache。

Table 97: Effects of FENCE/FENCE.I Instructions

Cache	FENCE	FENCE.I
I-Cache	None	Invalidates all cache lines
D-Cache	None	Writes back all cache lines

cache coherence

AX45MPV支持1/2/4/8核的dcache和l2 cache的cache coherency。因此必须要使能l2才能支持cache coherency。如果使能了l2 cache，cache manager会追踪l2 cache和dcache来维持cache coherency。参考17章。

使能了cache coherency一致性后dcache line state有Modified 、Exclusive、Shared、Invalid四种状态，而l2 cache还是普通的Invalid、Clean、Dirty三种状态。在支持cache coherency的前提下，l2 cache operations对dcache和l2 cache的line state的影响参考11.4.2的Table 105如下截图，dcache operations对dcache和l2 cache的line state的影响参考mesi规范或教程。

Operation	Cache Line State Before Operation		Cache Line State After Operation		Transaction to L3 Memory
	D-Cache	L2-Cache	D-Cache	L2-Cache	
L2_IX_WB, L2_PA_WB	Invalid	Invalid	Invalid	Invalid	None
	Invalid	Clean	Invalid	Clean	None
	Shared	Clean	Shared	Clean	None
	Exclusive	Clean	Shared	Clean	None
	Modified	Clean	Shared	Clean	Write back the dirty data
	Invalid	Dirty	Invalid	Clean	Write back the dirty data
	Shared	Dirty	Shared	Clean	Write back the dirty data
	Exclusive	Dirty	Shared	Clean	Write back the dirty data
	Modified	Dirty	Shared	Clean	Write back the dirty data
L2_IX_INVAL, L2_PA_INVAL	Invalid, Shared, Exclusive, or Modified	Invalid, Clean, or Dirty	Invalid	Invalid	None
L2_IX_WBINVAL, L2_PA_WBINVAL, L2_WBINVAL_ALL, L2_FILTERED_FLUSH	Invalid	Invalid	Invalid	Invalid	None
	Invalid	Clean	Invalid	Invalid	None
	Shared	Clean	Invalid	Invalid	None
	Exclusive	Clean	Invalid	Invalid	None
	Modified	Clean	Invalid	Invalid	Write back the dirty data
	Invalid	Dirty	Invalid	Invalid	Write back the dirty data
	Shared	Dirty	Invalid	Invalid	Write back the dirty data
	Exclusive	Dirty	Invalid	Invalid	Write back the dirty data

Continued on next page...

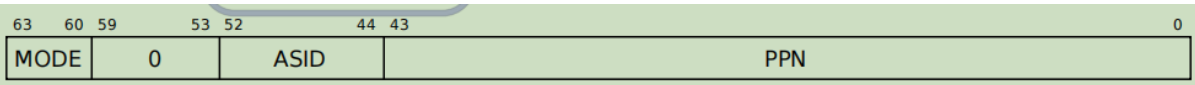
Table 105: (continued)

Operation	Cache Line State Before Operation		Cache Line State After Operation		Transaction to L3 Memory
	D-Cache	L2-Cache	D-Cache	L2-Cache	
	Modified	Dirty	Invalid	Invalid	Write back the dirty data

六、Memory

mmu

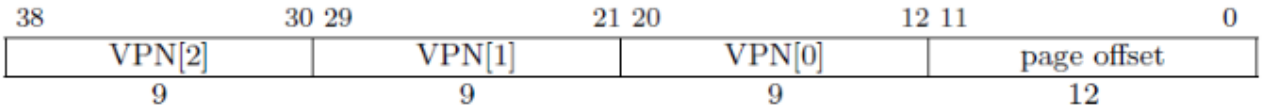
MMU负责将虚拟地址转换成物理地址，它和Instruction Fetch Control Unit (IFU), Load/Store Unit (LSU)以及Vector Load/Store Unit (VLSU)一起工作。MMU依靠内存中的页表来翻译虚拟地址，而TLB是页表的缓存，和cache架构也类似。AX45MPV的TLB有iTLB、dTLB、vTLB以及STLB，IFU使用iTLB，LSU使用dTLB，VLSU使用vTLB，这些就类似l1的icache和dcache，而STLB类似l2 cache，是由这些单元公用，页表类似l3 memory。 MMU模式默认是关闭的，需要通过satp CSR使能。参考22.9.1。satp的格式如下，MODE写8或者9都是使能MMU，一般使用Sv39的39位虚拟地址的三级页表模式。PPN表示第一级页表的物理基地址。



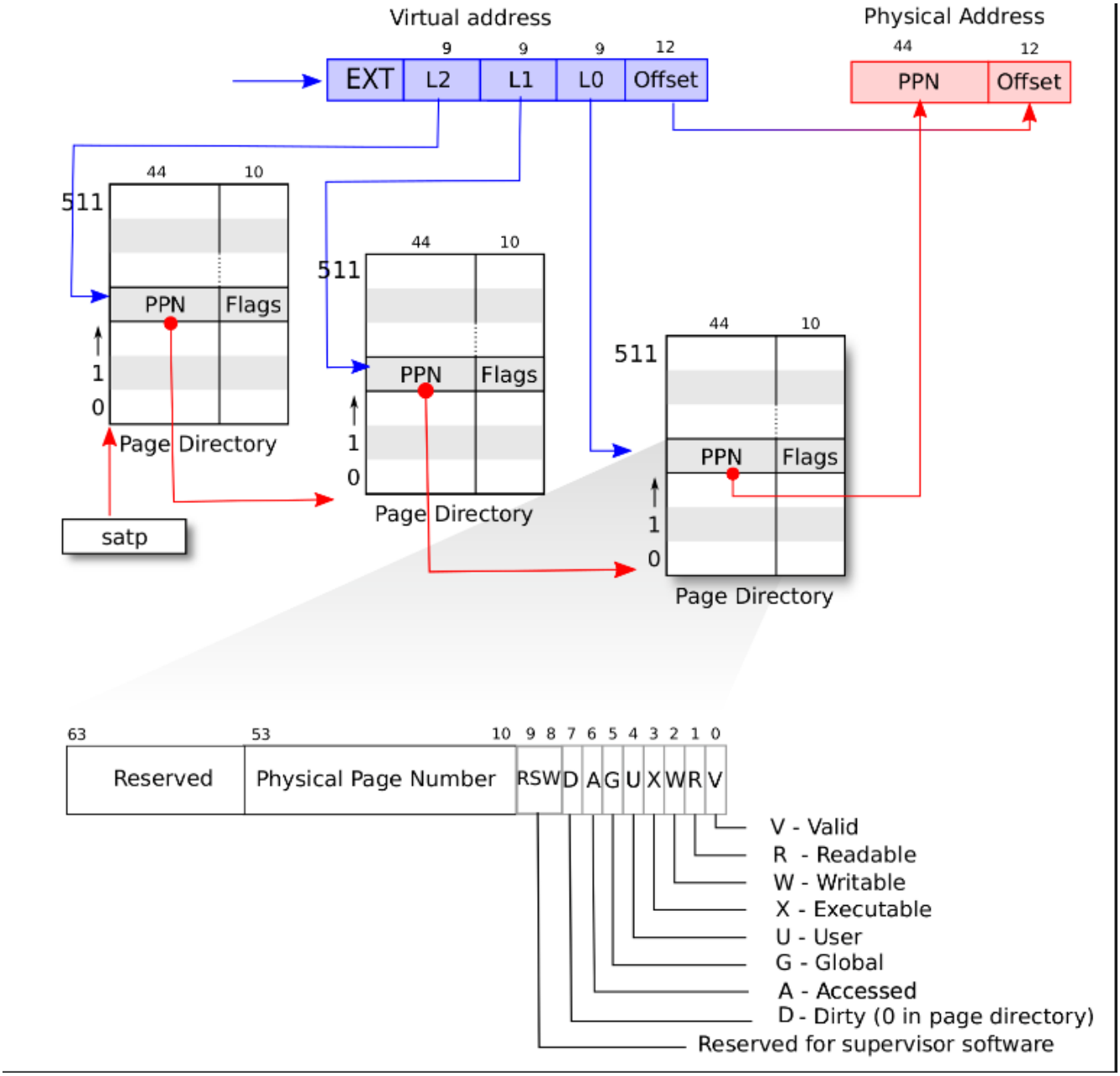
Field Name	Bits	Description	Type	Reset
PPN	[43:0]	PPN holds the physical page number of the root page table.	RW	0
ASID	[52:44]	ASID holds the address space identifier.	RW	0
MODE	[63:60]	MODE holds the page translation mode. When MODE is Bare, virtual addresses are equal to physical addresses in S-mode. When MMU is not supported in the product, this CSR will be hardwired to 0.	RW	0

Value	Name	Meaning
0	Bare	No page translation
8	Sv39	Page-based 39-bit virtual addressing
9	Sv48	Page-based 48-bit virtual addressing

如下所示，Sv39映射的虚拟地址只使用了前39bit，并且最后25位必须和第39位相同。其中30-38是索引第一级页表entry的索引，总共可以索引2的9次方即512个entry，每个entry占8个字节，所以第一级页表刚好4kb即一页大小。同理21-29和12-20分别是第二级和第三级页表entry的索引，也是都可以索引512个entry，每个entry 8字节，页表也是4kb大小。



页表条目格式和地址翻译过程如下。



首先页表entry各字段含义如下：

- V：有效位，1表示valid即是一个页表项，0表示invalid即不是页表项。
- R，W，X：分别表示读、写、执行权限，当三者都为0时表示指向下一级页表的页表项，否则是一个leaf页表项。
- U：1表示用户模式可以访问该页。
- G：1表示该页面是全局的，TLB属于全局，否则TLB属于进程独有，使用ASID。对于 non-leaf PTE，全局设置意味着页表后续级别中的所有映射都是全局的。
- A：1表示页面已经被cpu访问过，cpu访问A为0的页面会触发缺页异常，然后软件可以将A设置成1。也可以使用硬件方式更新，cpu访问A为0的页面不会触发缺页异常，cpu自动设置A为1。
- D：1表示页面已经被修改过，cpu尝试修改D为0的页面会触发缺页异常，然后软件可以将D设置成1。也可以使用硬件方式更新，cpu尝试修改D为0的页面不会触发缺页异常，cpu自动设置D为1。
- RSW：为软件保留，硬件不会使用。

地址翻译过程如下，另外在遍历页表过程中会做**PMP**检查。

- 从satp中取出44位的PPN字段然后左移12位得到56位的物理地址pa1指向第一级页表起始处。

- 取出虚拟地址的30-38作为第一级页表的页表项索引 index1，然后从物理地址pa1+8*index1处得到第一级页表的页表项，从页表项中取出44位PPN然后左移12位得到指向第二级页表起始处的56位物理地址pa2。
- 取出虚拟地址的21-29作为第二级页表的页表项索引 index2，然后从物理地址pa2+8*index2处得到第二级页表的页表项，从页表项中取出44位PPN然后左移12位得到指向第三级页表起始处的56位物理地址pa3。
- 取出虚拟地址的21-29作为第三级页表的页表项索引 index3，然后从物理地址pa3+8*index3处得到第三级页表的页表项，从页表项中取出44位PPN然后左移12位加上虚拟地址的0-11即得到最终的物理访问地址，最后还要做**PMA**和页表项的权限检查，如果不通过则触发访问异常，通过则正常访问内存。

pma与pmp

物理内存属性(pma)和物理内存保护(pmp)和内存管理单元(mmu)三者都有对物理内存访问有权限检查和保护的作用。但他们在粒度上有不同而且还有各自独特的功能。mmu的leaf页表项检查对应的页的权限，pmp使用pmp配置寄存器和pmp地址寄存器(22.16)来确定相应地址区间的权限，pma使用静态(14.2)或者动态编程方式(14.3)确定相应地址区间的权限。pma静态方式是硬件设计配置时固定的，软件改动不了，AX45MPV最大支持8个静态pma配置device region，访问地址在这些device region中是设备内存类型，即non-cacheable，non-bufferable，读写还可能有副作用，访问地址在这些device region之外就是memory region，默认是cacheable，write-back，read and write-allocate(MTYP=11)；pma的动态编程方式类似pmp，通过使用pma配置寄存器和pma地址寄存器(22.17)来确定对应地址区间的权限，如果一个地址同时出现在pma静态和动态中，那么动态会覆盖静态。当一个地址访问能同时通过mmu，pma，pmp三个(假如3个都打开)的权限时才能访问。除了权限检查外，mmu还有虚拟地址映射物理地址的功能；pma有指定内存属性的功能，即内存是device或memory类型，是否shareable，bufferable，cacheable，是否支持amo等，值得注意的是在arm中mmu的页表项可以指定内存属性，而riscv的mmu可以通过Svpbmt扩展来支持内存属性，但目前AX45MPV好像不支持；pmp相对mmu和pma就没有啥权限检查之外的功能了。　　pma和pmp配置寄存器和地址寄存器的格式含义和用法可以参考网上资料和22.16，22.17。这里只说一些注意点，对于AX45MPV来说，pmp支持TOR和NAPOT两种地址匹配模式，而pma只支持NAPOT一种地址匹配模式，当地址出现在多个区间里，pmp编号小的优先级高，pma应该也是(没找到文档说明，以后可以实验验证)，shareablility属性只影响MTYP为8，9，10，11的memory region(14.4)。

barrier

riscv中主要的内存屏障指令有三条，分别是fence、fence.i、sfence.vma。其中fence是多核屏障，fence.i和sfence.vma是单核屏障。也就是说fence对执行fence指令的cpu核也对其他cpu核生效，而fence.i和sfence.vma只对执行fence.i和sfence.vma指令的cpu核生效。

fence

> fence pred, succ

fence指令格式如上，其中pred和succ都是iorw的组合，该指令对I/O设备和普通内存的访问排序。在后续指令中的内存和 I/O 访问对自己和外部（例如其他核）可见之前，使这条指令之前的内存及 I/O 访问对自己和外部可见。

fence.i

> fence.i

fence.i用于在当前cpu核中使对内存指令区域的读写，对后续取指令可见。fence.i的实现取决于cpu的设计，简单的实现就是失效本地cpu的指令cache和清刷指令流水线。如果需要其他cpu也生效则需要通过ipi通知其他cpu也执行fence.i。

sfence.vma

> sfence.vma rs1, rs2

sfence.vma用来做虚拟地址映射相关的同步。比如修改了satp寄存器，就要使用sfence.vma指令使能后面的指令能感受到。sfence.vma根据参数不同可以刷新整个TLB也可以刷新指定的TLB。rs1表示vaddr，rs2表示asid。和fence.i一样sfence.vma也只能刷新本地cpu的TLB，如果需要刷新其他cpu则需要通过ipi通知其他cpu也执行sfence.vma。

七、Atomic

cpu需要提"供读-修改-回写"的原子操作指令才能实现在多处理器上的多线程并发访问统一资源。riscv atomic扩展支持两种原子操作：

- 加载保留/条件存储（load reserved / store conditional）
- 内存原子操作（AMO）

加载保留与条件存储

加载保留和条件存储类似arm的独占加载和独占存储，这种实现方式在一些教材中称为连接加载/条件存储（Load-Link/Store-Conditional, LL/SC）指令。这种方式就是先通过LL读取值，进行一些计算修改，然后通过SC写回去，如果SC失败那重新开始整个操作。 riscv的加载保留指令格式如下。

> lr.w rd, (rs1) lr.d rd, (rs1)

其中w表示加载4字节数据，d表示加载8字节数据，lr指令表示从rs1地址处加载4字节或8字节的数据到rd中，并且会注册一个包含rs1地址的保留集。 riscv的条件存储指令格式如下。

> sc.w rd, rs2, (rs1) sc.d rd, rs2, (rs1)

sc指令表示有条件地把rs2寄存器的值存储到rs1地址中，执行的结果ok或者fail反映到rd寄存器中。如果rd为0说明结果ok，即sc成功把数据写入rs1地址处，如果rd不为0说明结果fail，需要重新执行lr-修改-sc。无论sc指令执行是否成功，保留级中对应rs1地址的数据都会失效，这也是失败后要重新lr的原因。

```
#include <stdio.h>

static inline void atomic_add(int i, unsigned long *p)
{
    unsigned long tmp;
    int result;

    asm volatile(
        "1: lr.d %[tmp], (%[p])\n"
        "   add %[tmp], %[i], %[tmp]\n"
        "   sc.d %[result], %[tmp], (%[p])\n"
        "   bnez %[result], 1b\n"
        : [result]"=&r" (result), [tmp]"=&r" (tmp), [p]"+r" (p)
        : [i]"r" (i)
        : "memory"
    );
}
```

```
int main(void)
{
    unsigned long p = 0;
    atomic_add(5, &p);
    printf("atomic add: %ld\n", p);
}
```

上面是一个使用lr/sc的代码例子。这个程序通过atomic_add对p进行原子加5的操作。在atomic_add内部，先通过lr.d加载p地址处的数据到tmp变量中，然后通过add使得tmp加上i，然后通过sc.d将修改后的tmp写回指针p指向的内存，并将结果保存到result中，如果result为0表示成功则函数结束退出，如果不为0则跳回lr重新执行。

内存原子操作

AMO指令对内存中的操作数执行一个“读-修改-写回”原子操作，并将目标寄存器设置为操作前的内存值。AMO指令格式如下。

> amo.w rd, rs2, (rs1) amo.d rd, rs2, (rs1)

其中op表示操作后缀，一共有swap,add,and,or,xor,max,maxu,min,minu九种。w表示操作数的位宽为32位，d表示操作数的位宽为64位。rd表示目标寄存器，用来存指令执行后原来内存中的值。rs2表示源操作数，rs1表示需要原子操作的内存地址，因此(rs1)表示该内存中的值。

```
static inline void atomic_add(int i, unsigned long *p)
{
    unsigned long result;

    asm volatile(
        "amoadd.d %[result], %[i], (%[p])\n"
        : [result]"=&r"(result), [p]"+r"(p)
        : [i]"r" (i)
        : "memory"
    );
}
```

上面是用AMO指令实现的atomic_add，可以看到相比ls/sc代码简单很多，只要一条指令。然后再看一个使用AMO指令实现自旋锁的例子。

```
// get_lock(unsigned long *lock)
get_lock:
    li a2, 1
retry:
    amomax.d a1, a2, (a0)
    bnez a1, retry
    ret

// free_lock(unsigned long *lock)
free_lock:
    sd x0, (a0)
```

这个例子用一个指定的内存中的值表示锁，值为1表示锁已经被占用，值为0表示锁还没被占用。在get_lock中通过amomax.d a1, a2, (a0)将(a0)和1中的最大值原子写如(a0)，也就是说无论(a0)之前是什么执行这个指令后都为1。如果(a0)之前为0，那么amomax指令结束后a1为0，(a0)为1，表示锁是空闲的，然后ret，调用get_lock的函数就获得了锁。如果(a0)之前为1，那么amomax指令结束后a1为1，(a0)还是1，表示锁已经被占用，然后跳回retry继续执行amomax指令，这个过程一直循环到锁被其他地方释放。而释放锁就比较简单，一条普通的store指令将(a0)设置为0即可，因为释放锁只能是锁拥有者调，所以这样是可以的。

八、FreeRTOS

> start.S

九、Zephyr

> reset.S

十、Linux启动流程

- ROM：ZSBL (Zero Stage Boot Loader)，BOOTROM，M-mode。
- LOADER：FSBL (First Stage Boot Loader)，U-Boot SPL，M-mode。
- RUNTIME：OpenSBI，M-mode。
- BOOTLOADER：U-Boot Proper，S-mode。
- OS：Linux kernel，S-mode。

riscv的linux启动流程按启动顺序为以上5个阶段，下文单独分析每个阶段的启动流程。

BOOTROM

U-Boot SPL

OpenSBI

U-Boot Proper

Linux kernel