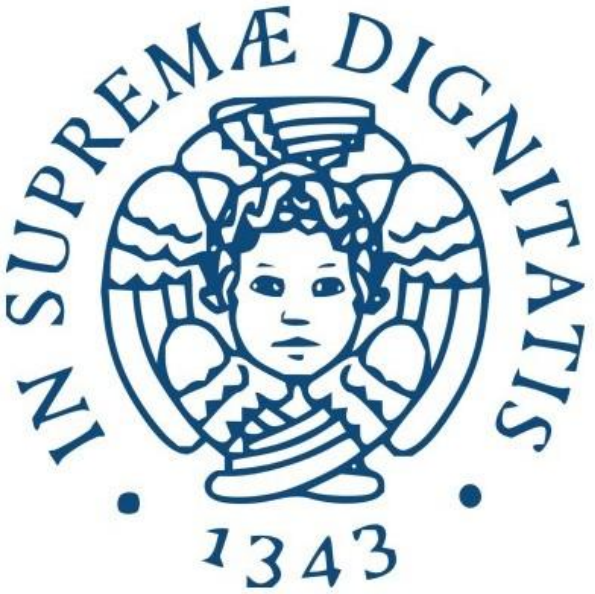


UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Tesi di Laurea Triennale in Ingegneria Informatica

Gestione delle interruzioni in architettura RISC-V

Candidato:
Chiara Panattoni

Relatore:
Prof. Giuseppe Lettieri

Anno Accademico 2022/2023

INDICE

1. Introduzione

L'architettura **RISC-V** è una nuova architettura del set di istruzioni (ISA), inizialmente ideata all'Università di Berkeley, California, nel 2010. Si basa sul principio RISC (*Reduced Instruction Set Computer*), che si caratterizza per il fatto di avere un set semplificato di istruzioni, ciascuna avente una lunghezza fissa.
La caratteristica più innovativa di RISC-V è il fatto che l'ISA è completamente open source e permette di avere un numero di istruzioni limitato rispetto alle architetture x86 o ARM, rendendo il costo di microprocessori per sistemi *embedded* molto più basso.
Le caratteristiche principali di RISC-V sono:

- Istruzioni che vengono eseguite in un unico ciclo di CPU.
- Una versione a 32 bit e una versione a 64 bit, con estensioni per supportare le istruzioni con floating point.
- Una forte modularità.
- Assenza di spazio I/O in favore di *Memory Mapped* I/O.

2. Prerequisiti per lo sviluppo del sistema

Per poter compilare ed eseguire il codice del sistema è necessario installare una toolchain per RISC-V.
Ciò si può fare tramite il seguente comando:

```
sudo apt install gcc-riscv64-unknown-elf
```

Una volta installata la toolchain, è necessario installare anche l'emulatore qemu e l'architettura RISC-V a 64 bit presente all'interno del pacchetto di architetture *qemu-system-misc*. La versione usata per l'emulazione del Nucleo è chiamata **qemu-system-riscv64**.

Il codice prodotto dalla toolchain può essere debuggato tramite l'apposita versione di gdb, **riscv64-unknown-elf-gdb**, che si comporta esattamente come la versione per x86. Dopo l'avvio di qemu, il debugger può dunque essere lanciato tramite il seguente comando:

```
riscv64-unknown-elf-gdb [file]
```

Infine, per utilizzare la toolchain è necessario impostare correttamente le variabili di sistema, per rendere visibili i binari alla shell. Si deve dunque inserire all'interno del config file il seguente percorso:

```
export PATH="/opt/riscv/bin:$PATH"
```

Terminata l'installazione degli strumenti necessari, il progetto potrà essere compilato tramite il comando **make**. La definizione delle regole per tale comando è contenuta nel *makefile* presente nella cartella root del progetto. In particolare, i comandi disponibili sono:

- **make run**: compila ed esegue il codice in modalità produzione.
- **make debug**: compila ed esegue il codice in modalità produzione, ed espone l'emulatore per il debug tramite gdb sulla porta TCP 1234.

- **make test**: compila ed esegue la testsuite contenente l'implementazione dei singoli moduli del Nucleo.
- **make test_debug**: compila ed esegue la testsuite, ed espone l'emulatore per il debug tramite gdb sulla porta TCP 1234.
- **make clean**: pulisce i file oggetto presenti nella cartella /objs generati durante la compilazione. Questo comando serve perché non sempre **make** riesce a determinare correttamente se un file necessiti di essere ricompilato.

3. Organizzazione delle cartelle

Cartella	**Descrizione**
/doc	Contiene i file pdf di relazione e slide inerenti al progetto.
/include	Contiene i file header.
/kernel	Contiene il bootloader del sistema.
/libCE	Contiene la libreria usata dal Nucleo, completamente migrata a RISC-V.
/objs	Contiene i file oggetto generati durante la compilazione
/sistema	Contiene i file del Nucleo.
/test	Contiene i file usati nella testsuite, ognuno dei quali rappresenta una funzionalità implementata e testata singolarmente.

4. Bootloader

Il sistema inizia la sua esecuzione da un bootloader scritto in linguaggio C e RISC-V Assembler.
Secondo le specifiche RISC-V, il sistema dev'essere caricato a partire dall'indirizzo fisico **0x8000_0000**. È stato dunque modificato il file kernel.ld, per prevedere una sezione di memoria RAM a tale indirizzo, grande 128 Mbyte, con i permessi di lettura, scrittura ed esecuzione.

```
/* START OF ADDITION */
MEMORY
{
  ram (rwxai) : ORIGIN = 0x80000000, LENGTH = 0x8000000
}
/* END OF ADDITION */
```

L'esecuzione effettiva ha inizio dal file entry.s, dove viene inizializzata la stack su cui verrà eseguito il codice C. A questo punto, il flusso del sistema continua in start.s, dove avvengono i seguenti passaggi:

- Vengono impostati i bit MPP[1:0] e SIE del registro `mstatus`, affinché, a seguito dell'istruzione *mret*, la modalità di privilegio passi a Supervisore con le interruzioni abilitate.
- Viene impostato il registro `mepc`, affinché, dopo l'istruzione *mret*, il codice salti alla funzione *boot_main*().
- Viene disabilitata la paginazione, azzerando il registro `satp`.
- Vengono delegate tutte le interruzioni e le eccezioni alla modalità Supervisore, scrivendo 1 nei 16 bits più bassi dei registri `mideleg` e `medeleg`. Vengono anche abilitate le interruzioni esterne, timer e software nel registro `sie`.

- Viene configurata la protezione della memoria tramite la scrittura nei registri `pmpaddr0` e `pmpcfg0`, per permettere alla modalità Supervisore di accedere a tutta la memoria fisica.
- Viene inizializzato il timer di sistema e il relativo gestore di interruzione.
- Viene inizializzato il registro `stvec` con l'indirizzo del gestore di interruzione generico.

Gli ultimi due punti verranno approfonditi nel Capitolo 6.

Viene infine eseguita l'istruzione *mret*, che permette di passare alla modalità Supervisore e iniziare a eseguire il codice contenuto nel file main.c.
La funzione *boot_main*() effettua l'inizializzazione del PCI e della periferica VGA in textmode 80x25, stampando inoltre dei messaggi di debug tramite l'interfaccia UART.

5. Privilegi e registri di RISC-V

5.1 Modalità di privilegio

In ogni momento, un *hardware thread* (hart) RISC-V gira in un qualche livello di privilegio, codificato in uno o più CSRs (*Control and Status Registers*). I livelli di privilegio vengono usati per fornire protezione tra diverse componenti software. Tentativi di effettuare operazioni non permesse dalla modalità di privilegio corrente portano a sollevare un'eccezione.

Attualmente, sono definiti tre livelli:

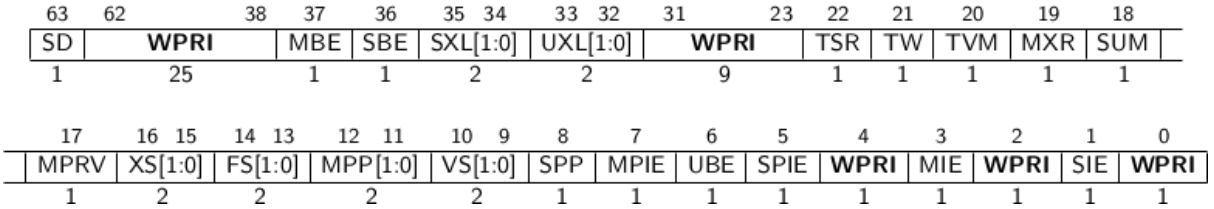
- Macchina. Codifica: 11. È il livello di privilegio più alto e l'unico obbligatorio da implementare.
- Supervisore. Codifica: 01. È prevista per l'utilizzo del sistema operativo ed è la modalità in cui gira il nostro progetto. Può eseguire le istruzioni privilegiate (es. abilitazione/disabilitazione delle interruzioni, gestione della paginazione, ...). Il software che gira in modalità Supervisore è detto *kernel*.
- Utente. Codifica: 00. È il livello di privilegio più basso, su cui girano le applicazioni normali. Non può usare le istruzioni privilegiate: se necessita di farlo, può chiamare l'istruzione *ecall*, che genera un'eccezione che viene delegata alla modalità Supervisore.

5.2 Registri

Di seguito, sono brevemente elencate le caratteristiche dei principali CSRs utilizzati per lo sviluppo di questa tesi.

`mstatus` e `sstatus`

Contengono le informazioni sullo stato dell'hart attualmente in esecuzione.



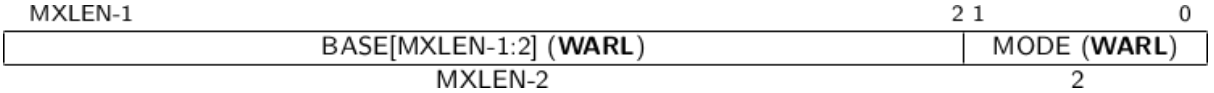
(In figura è riportato il CSR `mstatus`. Il CSR `sstatus` è formato da un sottoinsieme di questi campi.)

Sono di particolare rilevanza i campi:

- MPP e SPP, che mantengono la modalità di privilegio precedente all'arrivo di una trap, permettendo alla relativa istruzione x*ret* (dove x indica una tra M e S) di ripristinarla al termine della sua gestione.
- MPIE e SPIE, che mantengono il valore del bit di abilitazione alle interruzioni (MIE e SIE) attivo prima dell'arrivo della trap.
- MPRV, che determina il livello di privilegio con cui il sistema può accedere alla memoria.

`mtvec` e `stvec`

Mantengono la configurazione del trap vector.



Il campo BASE rappresenta l'indirizzo a cui viene impostato *pc* quando viene presa una trap. In base al valore del campo MODE, a BASE va aggiunto 4 volte il valore di `mcause`.

`mip/mie` e `sip/sie`

Contengono, rispettivamente, informazioni sulle interruzioni pendenti e bits per l'abilitazione delle interruzioni.

15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIP	0	SEIP	0	MTIP	0	STIP	0	MSIP	0	SSIP	0
4	1	1	1	1	1	1	1	1	1	1	1	1

15 … 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0
4	1	1	1	1	1	1	1	1	1	1	1	1

(CSRs `mip` e `mie`. I CSRs `sip` e `sie` sono formati da un sottoinsieme di questi campi.)

Il numero *i* riportato in *x*`cause` corrisponde al bit *i* sia in *x*`ip` che in *x*`ie`; così, nel caso della modalità Supervisore, il bit 9 equivale alle interruzioni esterne, il bit 5 alle interruzioni timer e il bit 1 alle interruzioni software.

Affinché una trap venga presa dalla modalità Supervisore, devono verificarsi entrambe le seguenti condizioni:

- (privilegio_corrente == S && `sstatus`.SIE = 1) || (privilegio_corrente < S);
- `sip`[*i*] = `sie`[*i*] = 1.

Scrivendo *x*`ip`[*i*]=0, l'interruzione *i* smette di essere pendente.

`mepc/sepc`

Quando viene presa una trap, questi registri vengono scritti con l'indirizzo virtuale dell'istruzione che è stata interrotta o che ha incontrato un'eccezione.

`mcause/scause`

Quando viene presa una trap, questi registri vengono scritti con un codice che indica l'evento che l'ha causata.

MXLEN-1	MXLEN-2 … 0
Interrupt	Exception Code (**WLRL**)
1	MXLEN-1

Se *Interrupt*=1, la trap è stata causata da un'interruzione.

Di seguito è riportata una lista di *Exception Code* del livello Macchina, rilevanti per il progetto (la lista per i valori di `scause` è un suo sottoinsieme).

Interrupt	*Exception Code*	Descrizione
1	1	Software (Supervisore)
1	3	Software (Macchina)
1	5	Timer (Supervisore)
1	7	Timer (Macchina)
1	9	Esterna (Supervisore)
1	11	Esterna (Macchina)
0	3	Breakpoint
0	8	Syscall (U-mode)
0	9	Syscall (S-mode)
0	11	Syscall (M-mode)
0	12	Page fault

`mtval/stval`

Quando viene presa una trap, questi registri possono venire scritti con delle informazioni specifiche sull'eccezione per aiutare il software a gestirla (solitamente, l'indirizzo virtuale che ha causato l'errore).

6. Gestione delle interruzioni

6.1 Eccezioni e interruzioni

Si parla di eccezione quando, a tempo di esecuzione, si verifica una condizione inusuale, associata a un'istruzione nell'hart corrente.
Si parla di interruzione quando si verifica un evento asincrono esterno che può causare un trasferimento di controllo inaspettato per l'hart.

Il manuale di RISC-V specifica tre sorgenti di interruzione:

- *Interruzioni software*. Vengono usate per accedere alle risorse del sistema operativo, passando il controllo dal programma chiamante a quello chiamato.
- *Interruzioni esterne*. Sono generate da dispositivi esterni (es. UART) e vengono gestite tramite il PLIC (vedi Sezione 6.4)
- *Interruzioni timer*. Sono causate dalla scadenza di un timer. Vengono usate quando si vuole che un certo evento accada ciclicamente e possono essere gestite solo dalla modalità Macchina. Il loro funzionamento è approfondito nella Sezione 6.5.

6.2 Inizializzazione

Quando si verifica un'interruzione, l'hardware procede ad aggiornare il valore di alcuni registri. Nello specifico:

- Salva il *program counter* del contesto interrotto nel registro `sepc`.
- Scrive il campo *Exception Code* del registro `scause` con un codice che indica la causa dell'interruzione. Nel caso in cui ciò sia dovuto a un *page fault*, scrive anche nel registro `stval` l'indirizzo che lo ha causato.
- Pone il campo `sstatus`.SPIE uguale al campo `sstatus`.SIE, mentre quest'ultimo viene azzerato, in modo da disabilitare ulteriori interruzioni.
- Salva la modalità di privilegio corrente (U o S) nel campo `sstatus`.SPP e passa al livello Supervisore.
- Copia l'indirizzo salvato in `stvec` in *pc* e riprende l'esecuzione da lì.

Il registro `stvec` era stato inizializzato nel file start.s.

```
# Set k_trap as interrupt handler
la t0, k_trap
csrw stvec, t0
```

6.3 Traps dallo spazio kernel

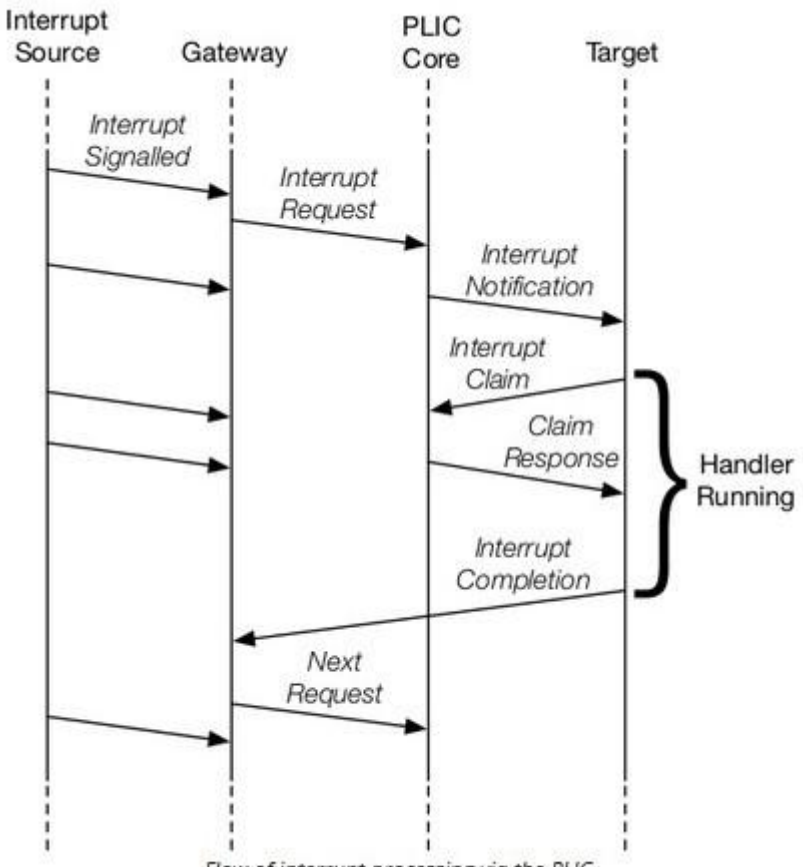
Quando il kernel è in esecuzione, il gestore delle interruzioni da utilizzare è `k_trap`. Dal momento che, normalmente, il nostro codice gira in modalità Supervisore, `k_trap` sa già che `satp` punta alla tabella delle pagine del kernel e dunque deve solo limitarsi a salvare i registri per permettere al codice interrotto, una volta gestita la trap, di riprendere ciò che stava facendo.

```
.global kInterruptHandler
.global k_trap
k_trap:
    addi sp,sp,-8
    sd ra,0(sp)
    call salva_stato

    call kInterruptHandler

    call carica_stato

    ld ra,0(sp)
    addi sp,sp,8

    sret
```

6.4 Platform-Level Interrupt Controller

Le interruzioni esterne nell'architettura RISC-V vengono gestite dal **PLIC** (*Platform-Level Interrupt Controller*), l'equivalente dell'APIC (*Advanced Programmable Interrupt Controller*) nell'architettura x86.

Il PLIC connette sorgenti di interruzione globali (dispositivi I/O) a targets di interruzione (contesti hart), e contiene un gateway di interruzione per ogni sorgente, insieme a un nucleo che si occupa del routing delle richieste di interruzione.
Il suo funzionamento è riportato nel seguente schema:



Flow of interrupt processing via the PLIC.

Una sorgente di interruzione invia un segnale al gateway, che lo converte in una richiesta da inoltrare al nucleo PLIC, e setta poi il relativo bit *x*IP (dove *x* è uno tra M e S).
Se non ci sono altre richieste pendenti, il gateway invia una notifica di interruzione al relativo target settando il suo bit *x*EIP.

I registri PLIC sono mappati sulla memoria fisica a partire dall'indirizzo **0x0c00_0000** e vengono inizializzati nella funzione *plic_init*().

```
#define PLIC           0x0c000000L
#define PLIC_PRIORITY  (PLIC + 0x0)
#define PLIC_PENDING   (PLIC + 0x1000)
#define PLIC_ENABLE    (PLIC + 0x2080)
#define PLIC_THRESHOLD (PLIC + 0x201000)
#define PLIC_CLAIM     (PLIC + 0x201004)
```

```
void plic_init() {
    // Set desired IRQ priorities non-zero (otherwise disabled)
    *(uint32*)(PLIC + UART0_IRQ*4) = 1;

    *(uint32*)PLIC_ENABLE = (1 << UART0_IRQ);

    *(uint32*)PLIC_THRESHOLD = 0;

    boot_printf("PLIC Initialized\n");

}
```

L'unica sorgente di interruzione presente attualmente è lo UART (*Universal Asynchronous Receiver-Transmitter*), dunque per prima cosa viene impostata la sua priorità al livello più basso possibile (il valore 0 disabiliterebbe le interruzioni). Si setta poi il bit corrispondente alle interruzioni UART all'interno del registro a 32 bits `PLIC_ENABLE` e, infine, si imposta il registro di soglia a 0, in modo da consentire l'interruzione di tutte le sorgenti aventi priorità maggiore.

Per testare il suo funzionamento, ho modificato il codice per l'input da tastiera, che era stato implementato dal collega Bedini senza l'utilizzo delle interruzioni.

Per prima cosa, ho inizializzato i registri UART, in particolare il registro `UART_IER`, che serve per abilitare le interruzioni sia in ingresso che in uscita (per il momento, sono attive solo quelle in ingresso).

```
void uart_init() {

    // disable interrupts
    WRITE_UART_REG(UART_IER, 0x00);

    // special mode to set baud rate
    WRITE_UART_REG(UART_LCR, UART_LCR_BAUD_LATCH);

    // LSB for baud rate of 38.4K
    WRITE_UART_REG(0, 0x03);

    // MSB for baud rate of 38.4K
    WRITE_UART_REG(1, 0x00);

    // leave set-baud modde,
    // and set word lenght to 8 bits, no parity
    WRITE_UART_REG(UART_LCR, UART_LCR_EIGHT_BITS);

    // Enable receive interrupts
    WRITE_UART_REG(UART_IER, UART_IER_RX_ENABLE);

}
```

La funzione viene richiamata all'interno di *test_keyboard_c*(), che stampa un messaggio e chiama *read_char*(), la quale cicla in attesa che venga premuto un tasto.

```
void read_char() {
    natb c;

    do {

        c = READ_UART_REG(UART_LSR);

    } while ((!(c & 0x01)) && !read);

}
```

Il bit più basso del registro `UART_LSR`, quando settato a 1, indica la presenza di un dato in `UART_RHR`. Si è resa necessaria l'aggiunta della variabile booleana *read* per uscire dal ciclo in quanto, al termine della gestione dell'interruzione, il flusso di esecuzione tornava qui, ma dal momento che la lettura del dato azzera il relativo bit in `UART_LSR`, il programma tornava a ciclare in attesa di ricevere un nuovo dato.

A questo punto, viene sollevata un'interruzione esterna (`scause`=9) che fa saltare il flusso di esecuzione al gestore di interruzione `kInterruptHandler`, chiamato dalla funzione `k_trap`.
Qui viene usata la funzione *dev_int*() per identificare la causa dell'interruzione, in particolare controlla se sia stata generata dal PLIC oppure dal timer.

```
int dev_int() {

    if (readSCAUSE() == 0x8000000000000009L) {
        // PLIC
        int irq = plic_claim();

        if (irq == UART0_IRQ)
            uart_intr();
        else
            boot_printf("Unexpected interrupt: %d\n", irq);

        if (irq)
            plic_complete(irq);

        return 1;
    }

    if (readSCAUSE() == 0x8000000000000001L) {
        // software int
        clearSSIP();

        return 2;
    }

    else
        return 0;
}
```

Nel caso del PLIC, utilizza la funzione *plic_claim*() per leggere dal registro `PLIC_CLAIM` l'ID dell'interruzione pendente a priorità più alta. Se non si tratta dello UART, lo segnala con un messaggio di errore, altrimenti salta al gestore di interruzione *uart_intr*(), che legge il valore contenuto in `UART_RHR`, salvandolo in una variabile globale, e pone *read* a true per segnalare che il dato è stato ricevuto.

Una volta gestita l'interruzione, è necessario informare il PLIC scrivendo l'ID dell'interruzione gestita all'interno del registro `PLIC_CLAIM`.

Il risultato è il seguente:

```
Starting tests...
PLIC Initialized
Starting salva/carica_stato test.
Salva_stato done.
Carica_stato done.
Salva/carica_stato test done.
Starting keyboard test
Press a character key:
Interruzione UART
PLIC Completed
The read character was: a
Keyboard test done
```

Con delle piccole modifiche al gestore UART, è stato possibile anche implementare la scrittura di una stringa di testo.

```
Write something: hello world!
You have written: hello world!
Keyboard test done
```

6.5 Interruzioni timer

Il timer di sistema era stato implementato nel bootloader, dove venivano impostati i registri `MTIME` e `MTIMECMP` affinché venisse generata un'interruzione dopo un certo tempo.

```
# Init timer #####

# Schedule the first interrupt
li a0, 0x200BFF8    # MTIME mem location
ld a1, 0(a0)        # Load MTIME, which are cycles since boot
li a2, TIMER_DELAY  # The delay at which to fire the int
add a2, a2, a1      # Get next cycle at which to fire the int
li a0, 0x2004000    # MTIMECMP mem location
sd a2, 0(a0)

# Set timer_machine_handler as the machine interrupt handler
la t0, timer_machine_handler
csrw mtvec, t0

# Enable timer interrupts
li t0, 128
csrw mie, t0

mret
```

Dal momento che un'interruzione timer può verificarsi in qualsiasi momento durante l'esecuzione del codice (utente o kernel), e non può essere disabilitata nemmeno durante le operazioni critiche, è importante che venga gestita esclusivamente in modalità macchina, così da evitare influenze indesiderate sul codice interrotto.
In particolare, il suo gestore (il cui indirizzo è mantenuto nel registro `mtvec`) si limita a impostare la prossima interruzione timer e a sollevare un'interruzione software, che verrà poi gestita normalmente dalla modalità Supervisore.

```
timer_machine_handler:

    # We need to use some registers, so we push them to the stack
    addi sp, sp, -24
    sd a0, 0(sp)
    sd a1, 8(sp)
    sd a2, 16(sp)

    # We schedule the next timer interrupt by reading the old MTIMECMP
    li a0, 0x2004000        # MTIMECMP mem location
    ld a1, 0(a0)            # Load old MTIMECMP
    li a2, TIMER_DELAY      # The delay at which to fire the int
    add a2, a2, a1          # Get next cycle at which to fire the int
    sd a2, 0(a0)

    # TEST:
    # Notify Supervisor Mode that a timer int fired
    li a1, 2
    csrw sip, a1

    # We restore the registers
    ld a0, 0(sp)
    ld a1, 8(sp)
    ld a2, 16(sp)
    addi sp, sp, 24

    mret
```

La funzione *dev_int*() controlla se si è verificata un'interruzione software (`scause`=1) e si limita ad azzerare il relativo bit nel registro `sip`.

```
clearSSIP:
    addi sp,sp,-16
    sd a0,0(sp)
    sd a1,8(sp)

    csrr a0, sip
    li a1, 2
    not a1, a1
    and a0,a0,a1
    csrw sip, a0

    ld a0,0(sp)
    ld a1,8(sp)
    addi sp, sp, 16
    ret
```

7. Conclusioni

La gestione delle traps nello spazio kernel è completamente implementata, mentre per quanto riguarda le traps nello spazio utente al momento è presente solo uno scheletro nel codice, nel caso in cui qualcun altro decida di proseguire il lavoro.

Una cosa importante da tenere a mente in questo caso è che l'hardware RISC-V non cambia la tabella delle pagine quando arriva un'interruzione. L'indirizzo del gestore puntato da `stvec` deve avere una valida mappatura sia nella tabella delle pagine utente sia nella tabella delle pagine del kernel. Ciò è possibile usando una *pagina trampolino*, che contiene il codice del gestore a cui punta `stvec` ed è mappata all'indirizzo `TRAMPOLINE`, il quale si trova sempre in cima allo spazio di indirizzi virtuali.

8. Ringraziamenti

Questa è la conclusione di un percorso pieno di alti e bassi, che nonostante le sue difficoltà sono felice di aver portato a termine.
Ringrazio inanzitutto il Professor Giuseppe Lettieri, per la disponibilità dimostrata e per avermi permesso di lavorare su un argomento che ho trovato molto interessante.
Ringrazio ovviamente i miei genitori e tutta la mia famiglia: senza il loro supporto, non credo che sarei arrivata fino a questo punto.
Infine voglio ringraziare i miei amici, quelli che ci sono da sempre e quelli che ho conosciuto lungo il percorso, per il loro sostegno e per la loro presenza ogni volta che ne ho avuto bisogno.