UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

Progetto e sviluppo del supporto per timer e tastiera in un nucleo multiprogrammato per architettura RISC-V64

Relatore:
Prof: Giuseppe Lettieri

Candidato:
Niccolò Pratesi

ANNO ACCADEMICO 2023/2024

Indice

1 Introduzione **3**
1.1 RISC-V . . . 3
1.2 QEMU . . . 4
1.3 Obiettivi . . . 4

2 Strumenti per lo sviluppo **6**
2.1 RISC-V GNU Compiler Toolchain . . . 6
2.2 QEMU RISC-V System emulator . . . 6
2.3 Progetto . . . 7

3 Organizzazione del sistema **8**

4 Timer **9**
4.1 Interruzioni . . . 9
4.2 Nei sistemi RISC-V . . . 10
4.3 Timer in QEMU . . . 11
4.4 Estensione "Sstc" . . . 12
4.5 Implementazione . . . 12
4.5.1 Driver . . . 12
4.5.2 Primitiva `delay()` . . . 14
4.5.3 Programma utente . . . 15

5 Tastiera **17**
5.1 PCI . . . 17
5.1.1 Capabilities List . . . 18
5.1.2 MSI-X . . . 19
5.2 VirtIO . . . 24
5.2.1 Device Status Field . . . 24
5.2.2 Feature Bits . . . 25
5.2.3 Notifications . . . 25
5.2.4 Device Configuration Space . . . 25
5.2.5 Virtqueue . . . 25
5.2.6 Input Device . . . 28
5.3 Implementazione . . . 30
5.3.1 Configurazione IMSIC . . . 30
5.3.2 PCI Device Discovery . . . 30
5.3.3 Funzioni della tastiera . . . 31
5.3.4 Modulo I/O . . . 39
5.3.5 Test . . . 40

6 Conclusioni e sviluppi futuri **42**

7 Ringraziamenti **43**

1 Introduzione

1.1 RISC-V

RISC-V (pronunciato "risk-five") è in un'ISA (Istruction Set Architecture) nata per supportare lo sviluppo delle architetture dei calcolatori e l'educazione. Attualmente si sta espandendo anche in campo commerciale grazie al suo essere interamente open source che ne riduce a zero i costi di licenza. Come suggerito dal nome, RISC-V implementa un'architettura RISC (Reduced Istruction Set Architecture). Le architetture di questo tipo si basano sull'essere più semplici e lineari rispetto alle loro controparti CISC (Complex Istruction Set Computer). Come conseguenza si ha che (in generale) le prime sono più veloci nell'esecuzione delle singole istruzioni, ma ne richiedono un numero maggior per eseguire la stessa operazione. Nello specifico, RISC-V fa uso di istruzioni di dimensione fissa (32-bit) che possono essere eseguite in un solo ciclo di clock e che hanno un formato comune che ne semplifica la fase di fetch e decodifica. In RISC-V sono presenti solamente due istruzioni che permettono di interagire con la memoria: l (load) per il caricamento, e s (store) per il salvataggio. L'architettura non presenta uno spazio di I/O. Le periferiche connesse al calcolatore sono quindi mappate in memoria tramite il meccanismo del Memory Mapped I/O. RISC-V ha la peculiarità di fare uso del meccanismo delle estensioni. La specifica permette di scegliere tra alcune ISA di base chiamate, Base Integer ISA, che contengono solamente un numero estremamente ridotto di istruzioni aritmetiche, e che differiscono tra loro principalmente nella dimensione dei registri (32, 64 o 128) indicata con la sigla XLEN. La dimensione di tali registri specifica anche la dimensione dello spazio di memoria. Viene poi offerta la possibilità di ampliare questo insieme di base tramite delle estensioni che aggiungono nuove funzionalità o ne semplificano l'implementazione (come vedremo anche in seguito). Di contorno alle specifiche relative all'ISA, sono presenti anche molti altri documenti che descrivono il funzionamento dei componenti presenti all'interno di un calcolatore basato su architettura RISC-V.

Nel primo volume dell'ISA [8] sono descritte le ISA di base e le possibili estensioni. Di seguito le sezioni di tale documento utilizzate durante questa tesina:

- L'ISA di base utilizzata dal nostro sistema è la RV64I (Capitolo 4) che estende l'insieme di istruzioni presente nell'ISA RV32I (Capitolo 2).
- In alcune situazioni è necessario garantire che il processore non esegua le istruzioni fuori ordine (come accade nei processori moderni). A questo scopo viene introdotta l'istruzione `fence` all'interno dell'estensione Zifencei (Capitolo 6).
- Zicsr è il nome dell'estensione che si occupa di implementare i Control and Status Register o CSR (Capitolo 7). Questi registri sono accessibili tramite un nuovo spazio di indirizzamento a loro dedicato.

Il secondo volume dell'ISA [9] descrive l'architettura privilegiata di un sistema RISC-V. Al suo interno è possibile trovare istruzioni privilegiate e nuove funzionalità necessarie per l'esecuzione dei sistemi operativi e per connettere dispositivi esterni. È in questo documento che sono descritti i vari livelli di privilegio in cui può trovarsi

il processore (Tabella 1.1). Affinché sia possibile che il processore esegua codice a livelli diversi è necessario che questi siano codificati in alcuni registri, per questo l'estensione Zicsr è fondamentale per poter utilizzare le funzionalità di tale volume. Del documento sono particolarmente rilevanti i capitoli: 2, contenente informazioni generiche sui CSR implementati; 3, sull'ISA di livello macchina; e 10, riguardante l'ISA di livello supervisore.

Livello	**Codifica**	**Nome**	**Abbreviazione**
0	`00`	Utente/Applicazione	U
1	`01`	Supervisore	S
2	`10`	*Riservato*	
3	`11`	Macchina	M

Tabella 1.1: RISC-V livelli di privilegio

1.2 QEMU

QEMU (Quick EMUlator) è un emulatore basato sulla tecnica di traduzione dinamica, attraverso la quale, permette di ottenere delle prestazioni equiparabili a quelle di una macchina reale. QEMU era l'emulatore utilizzato nel precedente sistema e, permettendo di emulare anche macchine RISC-V, si è deciso di mantenerlo anche nel nuovo. Dal punto di vista delle estensioni QEMU offre la possibilità di fare uso di tutto ciò che è in uno stato *frozen* o *ratified*[1].

Essendo RISC-V un'architettura che può differire molto nell'implementazione, QEMU offre la possibilità di scegliere sia l'ISA di base (nel nostro caso RV64I), sia una "board" che rappresenta una piattaforma diversa. Come piattaforma è stata scelta la "virt board" [4], il cui scopo è principalmente quello di essere utilizzata all'interno di sistemi virtuali per lo sviluppo di software. Nella Tabella 1.2 è riportato il modo in cui lo spazio di memoria viene assegnato ai vari dispositivi dalla virt board[2].

1.3 Obiettivi

L'obiettivo di questa tesi è la migrazione di parte del nucleo di Calcolatori Elettronici da architettura x86 a RISC-V. Il lavoro si articola in due parti: una prima riguardante il dispositivo timer e la primitiva `delay()`, e una seconda parte inerente alle funzioni che permettono di interagire con la tastiera, comprensive di driver. In entrambi i casi è presente del codice di test che fa uso delle funzionalità implementate e ne verifica il corretto funzionamento.

La tesi è stata scritta con l'obiettivo di produrre un documento che possa essere di aiuto a coloro che proseguiranno il progetto in futuro. Per questo, si è data particolare attenzione al modo con cui sono presentati i nuovi argomenti teorici,

[1] Il termine *frozen* indica le estensioni che non dovrebbero essere soggette a cambi significativi prima di essere ratificate.

[2] Tale tabella è ricavata direttamente dal codice sorgente di QEMU `https://github.com/qemu/qemu/blob/master/hw/riscv/virt.c`.

cercando di riportare le specifiche in maniera semplice e concisa. Nelle sezioni di codice, l'obiettivo è stato quello di evidenziare le scelte implementative effettuate e il motivo dietro di esse.

Nome	**Base**	**Dimensione**
VIRT_DEBUG	0x0	0x100
VIRT_MROM	0x1000	0xF000
VIRT_TEST	0x100000	0x1000
VIRT_RTC	0x101000	0x1000
VIRT_CLINT	0x2000000	0x10000
VIRT_ACLINT_SSWI	0x2F00000	0x4000
VIRT_PCIE_PIO	0x3000000	0x10000
VIRT_PLATFORM_BUS	0x4000000	0x2000000
VIRT_PLIC	0xc000000	variabile
VIRT_APLIC_M	0xC000000	variabile
VIRT_APLIC_S	0xD000000	variabile
VIRT_UART0	0x10000000	0x100
VIRT_VIRTIO	0x10001000	0x1000
VIRT_FW_CFG	0x10100000	0x18
VIRT_FLASH	0x20000000	0x4000000
VIRT_IMSIC_M	0x24000000	variabile
VIRT_IMSIC_S	0x28000000	variabile
VIRT_PCIE_ECAM	0x30000000	0x10000000
VIRT_PCIE_MMIO	0x40000000	0x40000000
VIRT_DRAM	0x80000000	0x0

Tabella 1.2: Assegnazione dello spazio di memoria all'interno della virt board

2 Strumenti per lo sviluppo

Lo sviluppo del nucleo avviene solitamente su un sistema Linux. Se non si dispone di una macchina con tale sistema installato è possibile usare il software WSL (Windows Subsystem for Linux), che permette di eseguire in maniera nativa su Windows dei file binari Linux, oppure tramite la creazione di una macchina virtuale su un'applicazione come VirtualBox. In ogni caso, è consigliato fare uso di una distribuzione user-friendly come Debian o Ubuntu.

Affinché sia possibile sviluppare e testare del codice per architettura RISC-V è necessario disporre di due strumenti: un cross-compilatore, ovvero un compilatore che produce codice eseguibile su una macchina differente rispetto a quella sul quale viene eseguito, e un emulatore in grado di eseguire codice RISC-V. Come cross compilatore è stato scelto quello presente all'interno della RISC-V GNU Compiler Toolchain [2], mentre come emulatore QEMU, come già anticipato nel precedente capitolo. Di seguito sono riportati i comandi utilizzati per installare questi strumenti sulla distro Linux Debian.

2.1 RISC-V GNU Compiler Toolchain

All'interno del repository ufficiale è presente una serie di istruzioni per fare la build della toolchain. Sebbene questo possa essere utile per ottenere l'ultima versione disponibile del compilatore, non è necessario per il nostro scopo. È sufficiente eseguire il seguente comando che ci permette di scaricare la toolchain tramite il package manager:

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

Codice 2.1: Comando di installazione della toolchain

Il debugger `gcc-riscv64-unknown-elf`, nelle ultime versioni, non è incluso all'interno del pacchetto. Tuttavia, è possibile scaricarlo seguendo le istruzioni presenti all'interno del repository.

2.2 QEMU RISC-V System emulator

Per ottenere QEMU è necessario installare un pacchetto contenente un insieme di emulatori differenti. Tra di essi sono presenti anche quelli per RISC-V sia a 32 che a 64-bit. Per installare il pacchetto possiamo eseguire il comando seguente:

```
sudo apt install qemu-system-misc
```

Codice 2.2: Comando di installazione di QEMU

2.3 Progetto

I file sorgenti del progetto possono essere scaricati, ad esempio tramite il comando `git clone`, direttamente dal repository `https://github.com/niccolopratesi/RISC-V-Calcolatori-Elettronici` (branch master). Una volta fatto ciò, saranno disponibili i seguenti comandi per compilare ed eseguire il codice del nucleo in versione RISC-V (la loro definizione è presente nel Makefile del progetto):

- `make [compile]`
  compila il codice del progetto;
- `make run`
  compila il codice e lo esegue all'interno di QEMU;
- `make debug`
  compila il codice e lo esegue con il flag `-S`, in questo modo è possibile connettersi per il debug tramite gdb alla porta TCP 1234;
- `make libce`
  compila i file presenti all'interno della cartella `libCE`, contenenti funzioni di utilità;
- `make clean`
  rimuove i file oggetto e gli eseguibili presenti all'interno della cartella `build` e `objs`.

3 Organizzazione del sistema

Cartella	**Descrizione**
`/build`	Contiene i file eseguibili.
`/doc`	Contiene le relazioni e le slide delle tesine svolte sul progetto.
`/include`	Contiene i file header che possono essere inclusi dai vari moduli.
`/io`	Contiene il codice C++ e Assembly del modulo I/O.
`/kernel`	Contiene il codice C++ e Assembly del modulo sistema, oltre al bootloader.
`/libCE`	Contiene la libreria utilizzata dal nucleo migrata a RISC-V.
`/objs`	Contiene i file oggetto.
`/user`	Contiene il codice C++ e Assembly del modulo utente.

Tabella 3.1: Organizzazione delle cartelle del sistema

4 Timer

4.1 Interruzioni

Prima di analizzare il funzionamento del timer nei sistemi RISC-V, introduciamo brevemente il modo con cui è possibile interagire con le interruzioni tramite i vari registri di controllo CSR. Per farlo riportiamo i registri coinvolti e il loro funzionamento[3]. Alcuni di essi presentano più campi, in tal caso saranno trattati solamente quelli rilevanti per il timer.

- `sstatus`
  Il registro `sstatus` è un registro di SXLEN-bit accessibile in lettura e scrittura, che tiene traccia dello stato corrente a cui opera il processore. Questo registro è un "sottoinsieme" del registro CSR `mstatus`. Il bit SIE (Supervisor Interrupt Enable) abilita o disabilita tutte le interruzioni quando il processore è in modalità supervisore. Quando il processore si trova a livello utente, il valore di SIE è ignorato, e le interruzioni di livello supervisore sono abilitate. Queste possono comunque essere disabilitate tramite il registro `sie`.

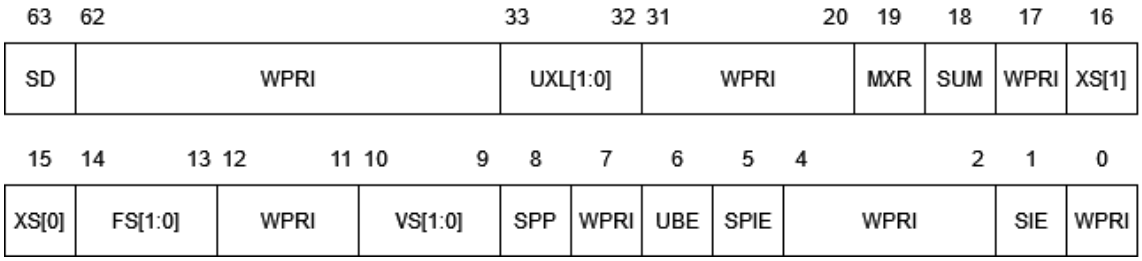

Figura 4.1: Registro di stato di livello supervisore (`sstatus`)

- `sie` e `sip`
  Il registro `sip` è un registro di SXLEN-bit accessibile in lettura e scrittura contenente informazioni sui pending interrupt, mentre `sie` è il corrispondente registro di SXLEN-bit accessibile in lettura e scrittura contenente i bit di enable per gli interrupt.

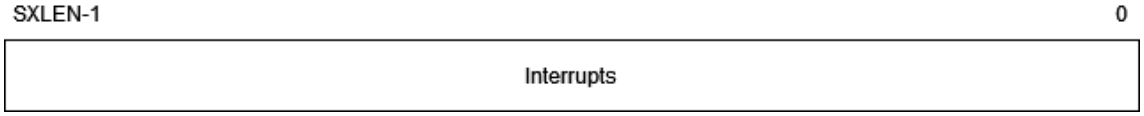

Figura 4.2: Registri di interrupt-pending e interrupt-enable di livello supervisore (`sip` e `sie`)

  La parte standard di entrambi i registri è realizzata come mostrato in Figura 4.3. I bit STI* (STIP per pending e STIE per enable) sono i bit per le interruzioni del timer di livello supervisore. Il bit STIP è di tipo read-only e viene resettato tramite un metodo specifico della piattaforma che vedremo in seguito.

[3]Nella specifica i CSR fanno uso della notazione SXLEN che indica la dimensione dei registri vista dal livello supervisore. Nel nostro caso corrisponde a quella macchina, cioè 64-bit.

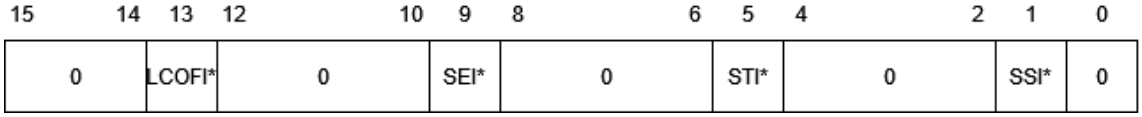

Figura 4.3: Porzione standard (bit 15:0) dei registri `sip` e `sie`

- `scause`
  Il registro `sip` è un registro di SXLEN-bit accessibile in lettura e scrittura contenente informazioni riguardanti la causa dell'interruzione. Il registro è formattato come mostrato in Figura 4.4. Il bit Interrupt è settato se la trap è stata causata da un'interruzione. Nel caso di un interrupt di tipo timer, il valore assunto dal campo Interrupt è 1 e il campo Exception Code è 5.

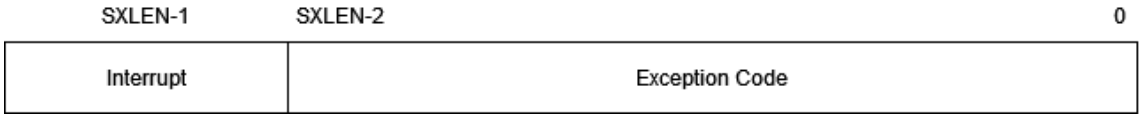

Figura 4.4: Registro di causa di livello supervisore (`scause`)

4.2 Nei sistemi RISC-V

In RISC-V le funzionalità del timer seguono la specifica ACLINT (Advanced Core Local Interruptor) [5]. All'interno del documento sono descritti i registri usati per lo scopo di inter-processor interrupt (IPI) e per le funzionalità del timer (noi vedremo solo questi) che sono locali al singolo core. I registri del timer si trovano all'interno di un Machine-level Timer Device (MTIMER). Più MTIMER possono essere presenti nella stessa piattaforma. Ognuno di essi può essere connesso a più HART (o anche a nessuno) fino ad un massimo di 4095, e mostra in memoria un registro MTIME e un registro MTIMECMP per ogni HART. Ad ogni HART a cui è connesso assegna un indice a partire da 0 che non necessariamente ha una relazione con l'identificatore univoco dell'HART all'interno del sistema (HART ID). Ogni dispositivo a due indirizzi di base differenti: uno per il registro MTIME e uno per i registri MTIMECMP. Questo è fatto per permettere a più dispositivi di condividere lo stesso registro MTIME. La Tabella 4.1 e la Tabella 4.2 mostrano come sono mappati questi registri in riferimento all'indirizzo di base.

Offset	**Dim.**	**Attr.**	**Nome**	**Descrizione**
`0x00000000`	8B	RW	MTIME	Contatore di tempo a livello macchina

Tabella 4.1: Mappa del registro Time dell'ACLINT MTIMER

Il registro MTIME è un registro a 64-bit accessibile in lettura e scrittura il cui valore viene incrementato con una frequenza costante. In caso di overflow il conteggio riparte da 0. Al reset il suo valore è impostato a 0. Il registro MTIMECMP è un

Offset	Dim.	Attr.	Nome	Descrizione
0x00000000	8B	RW	MTIMECMP0	Comparatore dell'HART di indice 0 a livello macchina
0x00000008	8B	RW	MTIMECMP1	Comparatore dell'HART di indice 1 a livello macchina
...	...	...	...	...
0x00007FF0	8B	RW	MTIMECMP4094	Comparatore dell'HART di indice 4094 a livello macchina

Tabella 4.2: Mappa dei registri Compare dell'ACLINT MTIMER

registro per-HART a 64-bit accessibile in lettura e scrittura. Il suo valore al reset non è specificato. L'interrupt del timer di livello macchina di un HART è attivo ogniqualvolta MTIME è maggiore o uguale del valore contenuto nel registro MTIMECMP corrispondente. Viceversa, l'interrupt è disattivato ogniqualvolta MTIME ha un valore minore di MTIMECMP.

La specifica RISC-V Privileged Architecture propone un modo alternativo per accedere al registro MTIME. È infatti possibile fare uso del registro CSR `time` che, secondo specifica, è "un'ombra" di tipo read-only del registro in memoria MTIME. Nell'implementazione, come vedremo, faremo uso di tale opzione essendo l'accesso a un registro più semplice rispetto a un accesso in memoria.

4.3 Timer in QEMU

Come possiamo vedere dalla Tabella 1.2 la virt board di QEMU segue la specifica ACLINT e, in particolare, implementa il dispositivo SiFive CLINT (Core-Local Interruptor) [3]. Il CLINT veniva usato in ambienti RISC-V prima che venisse formalizzata la specifica ACLINT, la quale è stata quindi pensata per essere retrocompatibile con tale dispositivo. Dal punto di vista della specifica ACLINT il dispositivo SiFive CLINT può essere semplicemente visto come un dispositivo MSWI (per la gestione degli IPI) e un dispositivo MTTIMER mappati in memoria ad indirizzi contigui, come mostrato nella Tabella 4.3.

SiFive Clint Intervallo di Offset	Dispositivo ACLINT	Funzionalità
0x00000000 - 0x0003FFF	MSWI	Inter-processor (o software) interrupt di livello macchina
0x00004000 - 0x0000BFFF	MTIMER	Contatore con frequenza costante e eventi del timer di livello macchina

Tabella 4.3: Un dispositivo SiFive CLINT corrisponde a due dispositivi ACLINT

RISC-V all'interno delle proprie specifiche indica che la piattaforma deve fornire un meccanismo per determinare il periodo del tick del timer. In QEMU possiamo trovare questo valore in maniera statica, cioè non a livello di esecuzione, che per il nostro scopo è più che sufficiente. Per farlo avviamo la macchina virtuale e entriamo in modalità "monitor" tramite la combinazione di tasti `Ctrl+Alt+2`. Eseguiamo poi il comando `info qtree`, il quale stampa a video tutti i dispositivi inizializzati da QEMU. Tra di essi troviamo il dispositivo `riscv.aclint.mtimer` accompagnato dal seguente output:

```
dev: risc.aclint.mtimer, id ""
    ...
    timebase-freq = 10000000 (0x989680)
    ...
```

Codice 4.1: Output del comando `info qtree`

Tra le varie informazioni fornite, il campo `timebase-freq` indica la frequenza del clock che governa l'incremento del contatore MTIME. Nel caso di QEMU tale frequenza è di 10MHz. All'interno del nostro sistema siamo interessati a ricevere un interrupt ogni 50ms, per questo incrementeremo il valore di MTIMECMP di 500000 per ognuno di essi.

4.4 Estensione "Sstc"

Tutto ciò che è stato appena descritto è valido per il livello macchina, a noi però interessa lavorare quanto più possibile a livello supervisore. A tale scopo facciamo uso dell'estensione "Sstc" presente nella specifica RISC-V Privileged Architecture. Una delle funzionalità offerte dall'estensione consiste nell'aggiunta di un ulteriore CSR chiamato `stimecmp`, il cui compito è quello di produrre interrupt del timer direttamente a livello supervisore. Senza di esso sarebbe stato necessario intercettare quelli di livello macchina e, tramite una scrittura nel registro `sip`, creare via software un interrupt di tale livello, cosa nettamente più complessa e dispendiosa computazionalmente. Bisogna notare che il registro `stimecmp` rispetta le specifiche ACLINT che indicano la presenza di un registro di comparazione del timer per HART, essendo i registri CSR locali al singolo HART. Infine, il registro CSR `stimecmp`, a differenza dei registri MTIMCMP, non richiede al software di individuare il registro corrispondente a ciascun HART (che comunque nel nostro caso non sarebbe stato necessario, usando un sistema mono-HART), rendendo quindi l'accesso più immediato (oltre al vantaggio già visto anche per `time` e MTIME).

4.5 Implementazione

4.5.1 Driver

Il driver del timer si occupa di programmare le sue interruzioni e di richiamare la funzione `c_driver_td()` per gestire la coda dei processi sospesi. Il primo passo

per la programmazione delle interruzioni consiste nell'inizializzare il timer stesso. Questa fase avviene all'interno del file `kernel/start.s`.

```
1     ...
2     li t0, 0x8000000000000000
3     csrs menvcfg, t0
4
5     csrr t0, time
6     li t1, TIMER_DELAY
7     add t1, t1, t0
8     csrw stimecmp, t1
9
10    csrsi mcounteren, 0b10
11    ...
```

Codice 4.2: Inizializzazione timer

Settiamo il bit 63 (costante `0x8000000000000000`), chiamato STCE (STimeCmp Enable), all'interno del CSR `menvcfg` (Machine Environment Configuration Register), così da abilitare l'estensione Sstc (righe 2-3). Questo CSR permette di controllare alcune caratteristiche dell'ambiente di esecuzione per i livelli meno privilegiati del livello macchina. In seguito calcoliamo l'istante a cui deve essere inviata la prima interruzione del timer. Per fare ciò, sommiamo il valore presente all'interno del CSR `time` con la costante `TIMER_DELAY` definita all'inizio del file e avente valore 500000 per i motivi visti nella sezione precedente (righe 5-7). Salviamo il valore appena calcolato all'interno del CSR `stimecmp` (riga 8). Infine, impostiamo il bit 2 (costante `0b10`) (riga 10), chiamato TM, all'interno del registro `mcounteren` in modo da abilitare il livello supervisore a fare accessi ai registri `time` e `stimecmp`. Il registro di controllo `mcounteren` (Machine Counter-Enable Register) è utilizzato per controllare la disponibilità dei vari registri di conteggio ai livelli inferiori.

Ogniqualvolta un'interruzione viene prodotta dal timer, è necessario eseguire lo schedule dell'interrupt successivo. Il codice per la gestione delle interruzioni si trova all'interno del file `kernel/traps_c.cpp`. È in questo file che possiamo trovare la funzione `supervisor_handler()`, la quale viene mandata in esecuzione all'arrivo di un interrupt. Se l'interruzione non è di tipo software viene richiamata a sua volta la funzione `dev_int()`. In caso di interrupt proveniente dal timer il registro `scause` contiene il valore `0x8000000000000005` (per i motivi visti in precedenza), viene quindi eseguito il Codice 4.3. Il suo compito è quello di richiamare le funzioni `schedule_next_timer_interrupt()` e `c_driver_td()` (righe 7-8).

```
1  int dev_int()
2  {
3      ...
4      if (scause == 0x8000000000000005L) {
5          // timer_debug();
6
7          schedule_next_timer_interrupt();
8          c_driver_td();
9
```

```
10          return 3;
11      }
12      ...
13  }
```

Codice 4.3: Driver del timer

La prima funzione può essere trovata all'interno del file `kernel/traps_asm.s`. Il suo scopo è proprio quello di programmare l'istante di arrivo del prossimo interrupt del timer. Il codice eseguito è di fatto identico a quello che abbiamo visto anche all'interno del file `kernel/start.s`, con l'unica differenza riguardante il valore letto dal timer che in questo caso proviene dal CSR `stimecmp` invece che dal registro `time`. Non utilizziamo il registro `time` perché il suo valore potrebbe variare tra l'arrivo dell'interruzione e la lettura del suo valore durante lo schedule dell'interrupt successivo. La conseguenza sarebbe che gli interrupt provenienti dal timer potrebbero essere soggetti a inconsistenze nella frequenza di arrivo.

```
1  .global schedule_next_timer_interrupt
2  schedule_next_timer_interrupt:
3      csrr t0, stimecmp
4      li t1, 500000
5      add t0, t0, t1
6      csrw stimecmp, t0
7      ret
```

Codice 4.4: Programmazione prossimo interrupt del timer

4.5.2 Primitiva `delay()`

La funzione `c_driver_td()` è definita all'interno del file `kernel/timer.cpp`. Al suo interno sono presenti anche le funzioni `inserimento_lista_attesa()` e `c_delay()`, la struttura dati `richiesta` e la variabile di tale tipo `p_sospesi`. Le loro definizioni sono rimaste invariate dal vecchio sistema.

```
1  .global delay
2  delay:
3      .cfi_startproc
4      li a7, TIPO_D
5      ecall
6      ret
7      .cfi_endproc
```

Codice 4.5: Primitiva `delay()`

Infine, si è ripristinata la possibilità di chiamare la system call `delay()` all'interno dei moduli utente e I/O. La dichiarazione di tale funzione è presente all'interno del file `include/sys.h`, mentre la sua definizione (Codice 4.5) è stata introdotta nei

file kernel/syscall.s e user/user_asm.s. La funzione esegue l'istruzione ecall dopo aver salvato il valore della costante TIPO_D nel registro a7 (righe 4-5).

Dal punto di vista del nucleo è necessario associare la chiamata di tale primitiva con l'esecuzione della funzione c_delay(). Per fare ciò è sufficiente aggiungere un "case" all'interno dello switch presente nella funzione syscall_user(). Quest'ultima viene richiamata dalla funzione supervisor_handler() ogniqualvolta l'interrupt è di tipo software.

```
1  ...
2  case TIPO_D:
3      c_delay(p->contesto[I_A0]);
4      break;
5  ...
```

Codice 4.6: Case della system call delay()

4.5.3 Programma utente

Per testare le funzionalità del timer è stato scritto il programma utente riportato nel Codice 4.7. La funzione main() crea due processi figli che eseguono la primitiva delay() per 5 e 10 cicli di clock, dopo di che stampano un messaggio sul terminale e terminano la propria esecuzione.

```
1  void main_body(natq wait)
2  {
3      flog(LOG_DEBUG, "Mi metto in attesa di %d cicli di timer", wait);
4      delay(wait);
5      flog(LOG_DEBUG, "Attesa terminata");
6      terminate_p();
7  }
8
9  extern "C" void main()
10 {
11     flog(LOG_DEBUG, ">>>INIZIO<<<");
12
13     flog(LOG_DEBUG, "Creo il primo processo utente");
14     activate_p(main_body, 5, MIN_PRIORITY+1, LIV_UTENTE);
15     flog(LOG_DEBUG, "Creo il secondo processo utente");
16     activate_p(main_body, 10, MIN_PRIORITY+1, LIV_UTENTE);
17
18     flog(LOG_DEBUG, "Cedo il controllo al primo processo utente");
19     terminate_p();
20 }
```

Codice 4.7: Programma utente (timer)

Per verificare il corretto funzionamento è possibile rimuovere il commento della funzione timer_debug() all'interno di dev_int() nel file kernel/traps_c.cpp.

La funzione si occupa semplicemente di stampare sul terminale "Timer fired", permettendoci di individuare l'arrivo di ogni interruzione del timer. Nel Codice 4.8 è riportato l'output sul terminale prodotto dal processo utente con la funzione `timer_debug()` abilitata.

```
...
DBG    4    >>>INIZIO<<<
DBG    4    Creo il primo processo utente
INF    4    proc=5 entry=ffff8000000012c4(5) prio=2 liv=0
DBG    4    Creo il secondo processo utente
INF    4    proc=6 entry=ffff8000000012c4(10) prio=2 liv=0
DBG    4    Cedo il controllo al primo processo utente
INF    4    Processo 4 terminato
DBG    5    Mi metto in attesa di 5 cicli di timer
DBG    6    Mi metto in attesa di 10 cicli di timer
INF    0    Timer fired
INF    0    Timer fired
INF    0    Timer fired
INF    0    Timer fired
INF    0    Timer fired
DBG    5    Attesa terminata
INF    5    Processo 5 terminato
INF    0    Timer fired
INF    0    Timer fired
INF    0    Timer fired
INF    0    Timer fired
INF    0    Timer fired
DBG    6    Attesa terminata
INF    6    Processo 6 terminato
INF    0    Shutdown
```

Codice 4.8: Output programma utente (timer)

5 Tastiera

Il vecchio sistema implementava la tastiera PS/2 come dispositivo di ingresso. Purtroppo la virt board del nuovo nucleo non la implementa più. Per vedere i dispositivi supportati da essa usiamo il comando `qemu-system-riscv64 -device help`. Nella sezione "input devices" troviamo, tra le altre cose, i seguenti dispositivi:

```
...
name "usb-kbd", bus usb-bus
...
name "virtio-keyboard-device", bus virtio-bus
name "virtio-keyboard-pci", bus PCI, alias "virtio-keyboard"
...
```

Codice 5.1: Output del comando `qemu-system-riscv64 -device help`

Le tre tastiere appena riportate sono le uniche messe a disposizione dalla virt board, si è quindi dovuto scegliere tra una di esse. La tastiera `usb-kbd` fa uso del protocollo USB sia per lo scambio di dati che per la configurazione. Inoltre, essendo la virt board sprovvista di un metodo diretto per accedere al bus USB, è necessario fare uso di un ponte PCI-USB come xHCI, che non è trasparente e che richiede anch'esso una configurazione. Questo, unito al fatto che il protocollo è di una certa complessità, ha fatto sì che non si optasse per tale tastiera.

La scelta è quindi ricaduta su una delle due tastiere che fa uso del protocollo VirtIO per lo scambio dei dati. La tastiera `virtio-keyboard-device` mappa i propri registri in memoria (MMIO), mentre la tastiera `virtio-keyboard-pci` è un dispositivo che segue lo standard PCI. Lo standard VirtIO specifica che il comportamento di un dispositivo "memory mapped" è basato su quello di un dispositivo PCI, ma che il primo è preferibile solamente nel caso in cui l'ambiente virtuale non supporti un bus PCI (come nel caso di sistemi embedded). Inoltre, il dispositivo `virtio-keyboard-device` è stato implementato all'interno di QEMU solamente per retrocompatibilità[4]; si può notare ciò anche dal fatto che tale dispositivo è di tipo "Legacy"[5]. Per questo la scelta finale è ricaduta sulla tastiera `virtio-keyboard-pci`. Infine, bisogna notare che nel caso di RISC-V anche un dispositivo di tipo PCI rende disponibile il proprio spazio di configurazione tramite accessi in memoria, essendo l'architettura sprovvista di uno spazio di I/O.

Prima di passare all'implementazione vera e propria della tastiera, alcune note sulle funzionalità utilizzate al suo interno che non erano presenti nel vecchio nucleo.

5.1 PCI

La tastiera `virtio-keyboard-pci` fa uso di alcune nuove specifiche offerte dallo standard PCI, nello specifico Capabilities List e MSI-X [1]. Di seguito una breve

[4]Come suggerito nel seguente post: `https://stackoverflow.com/questions/76219021/who-will-configure-virtio-mmio-device-register-in-qemu`

[5]I dispositivi indicati come legacy, ovvero dispositivi implementati prima del rilascio della versione 1.0 della specifica, e che sono quindi "obsoleti".

spiegazione sul loro funzionamento.

5.1.1 Capabilities List

Alcune funzionalità offerte dal dispositivo PCI sono gestite tramite l'aggiunta di registri addizionali a una "linked list" chiamata Capabilities List. Il bit 4 (Capabilities List) all'interno del registro Status, se settato, indica la presenza di tale struttura dati, e in tal caso il registro Capabilities Pointer punta al primo elemento della lista di capability.

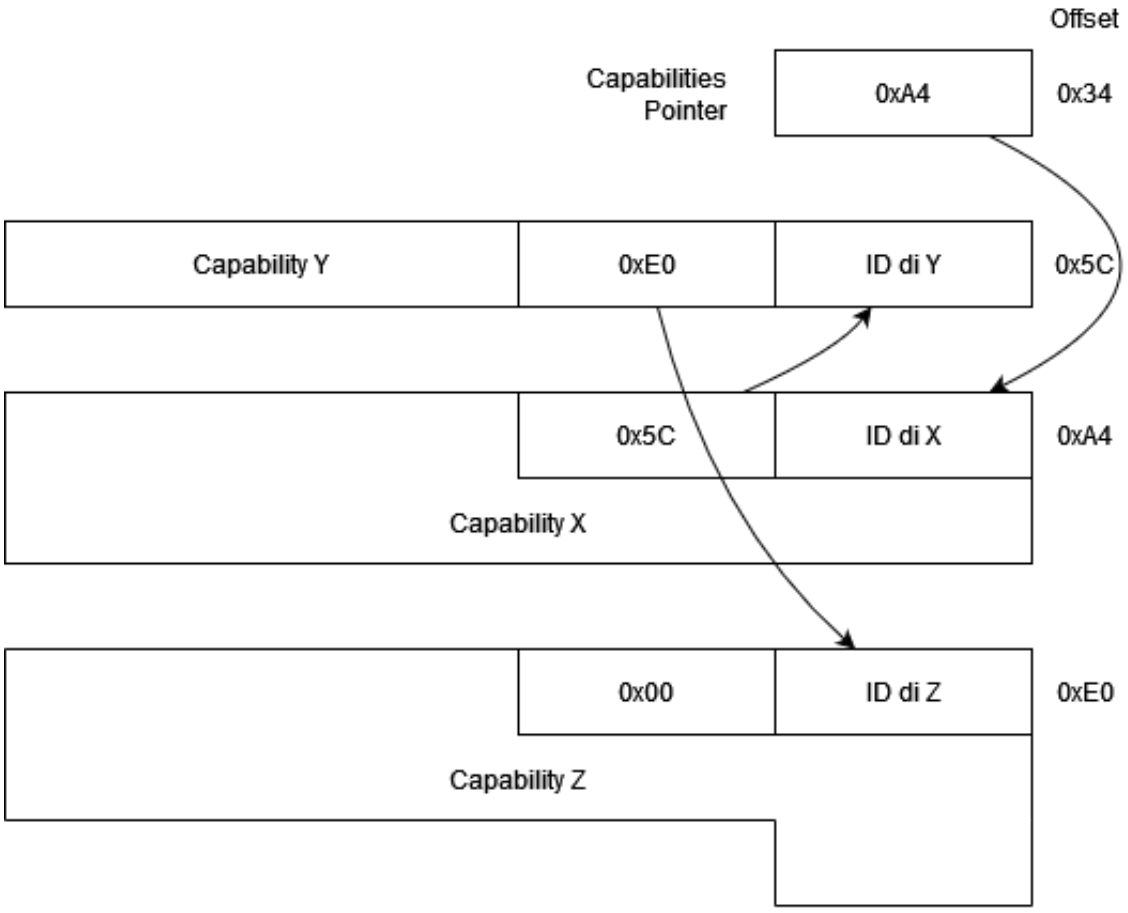

Figura 5.1: Esempio di capability list

Come mostrato in Figura 5.1, ogni capability contiene: un campo ID di 8-bit assegnato dal PCI-SIG (PCI Special Interest Group), un puntatore di 8-bit alla prossima capability nello spazio di configurazione, e un certo numero di registri dopo il puntatore che implementano la capability vera e propria. Il valore `0x00` nel campo puntatore indica l'ultimo elemento della lista.

Ogni capability che è stata definita deve avere un ID assegnato dal SIG, esattamente come accade con i Vendor ID. Ogni capability deve descrivere la struttura e l'uso dei registri che seguono il puntatore. Nel seguito vedremo due tipologie di capability: quella MSI-X avente ID `0x11` e quella VirtIO avente ID `0x09`.

5.1.2 MSI-X

Message Signaled Interrupt (MSI) è una feature opzionale che permette a un dispositivo PCI di inviare un interrupt tramite la scrittura di un preciso valore a un preciso indirizzo di memoria, dipendentemente dal sistema. Il dato da scrivere e l'indirizzo, chiamati "vector", sono definiti dal software di sistema durante la fase di configurazione del dispositivo.

MSI-X definisce un'estensione delle funzionalità di MSI separata da questa. Rispetto a MSI, i principali vantaggi di MSI-X sono: supportare un maggior numero di vector per funzione PCI, e la possibilità per ogni vector di usare un indirizzo e un valore indipendente, specificato tramite una tabella nello spazio di memoria. La tastiera utilizzata nel sistema fa uso della funzionalità di MSI-X per inviare gli interrupt. La Figura 5.2 mostra la struttura della capability relativa a MSI-X.

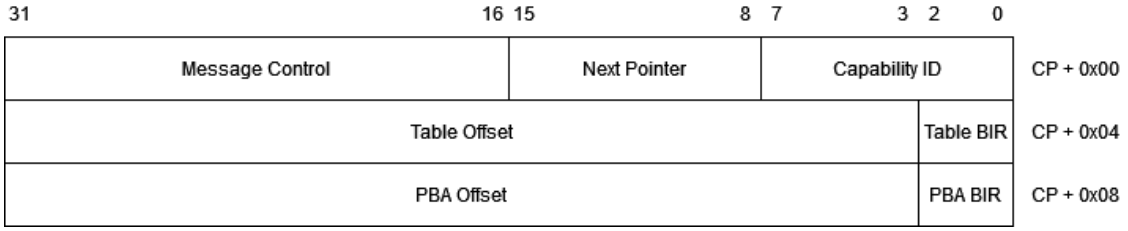

Figura 5.2: Struttura della MSI-X capability

I campi all'interno di tale capability assumono i seguenti significati:

- Capability ID e Next pointer
  I campi Capability ID e Next Pointer sono quelli standard della Capabilities List. All'interno del campo Capability ID deve essere presente il valore `0x11` per indicare che la funzione è in grado di usare MSI-X.

- Message Control
  Il campo Message Control è ulteriormente suddiviso al proprio interno come mostrato in Figura 5.3.

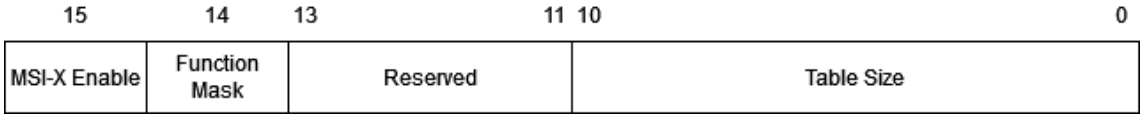

Figura 5.3: Struttura del campo Message Control della MSI-X Capability

Il campo MSI-X Enable permette al software di abilitare o disabilitare il dispositivo a usare gli MSI-X. Nel caso in cui il suo valore sia a 0, il dispositivo invierà un interrupt tramite il proprio INTx# pin. Al reset il suo valore è 0, rendendo la funzionalità MSI-X disabilitata di default.

Il campo Function Mask permette di mascherare tutti i vector della funzione, indipendentemente dal fatto che siano mascherati singolarmente o meno. Se settato a 1, tale bit impedisce al dispositivo di fare scritture MSI-X, che farà

quindi uso della struttura PBA (che vedremo dopo) per notificare il driver che un interrupt è pending. Al reset il suo valore è 0.

Il campo Table Size è usato per determinare la dimensione della MSI-X Table. Le dimensione N è codificata in questo campo come N-1, quindi, ad esempio, il valore `00000000011` indica una tabella di dimensione 4.

- Table Offset e Table BIR
  I campo Table BIR (BAR Indicator Register) indica quale dei BAR della funzione è usato per mappare nello spazio di memoria la MSI-X Table. Il valore 0 indica il BAR0, il valore 1 indica il BAR1, e così via. Il campo Table Offset contiene l'offset dall'indirizzo contenuto nel BAR specificato nel campo Table BIR, a partire dal quale è mappata la tabella. I 3 bit inferiori sono settati a 0 dal software in modo da ottenere un offset allineato a 8-byte.

DWORD3	DWORD2	DWORD1	DWORD0		
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 0	Base
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry 1	Base + 1*16
...	...	...	...	...	...
Vector Control	Msg Data	Msg Upper Addr	Msg Addr	entry (N-1)	Base + (N-1)*16

Figura 5.4: Struttura della tabella MSI-X

Come mostrato nella Figura 5.4, una tabella MSI-X contiene normalmente più entry, ognuna delle quali è in grado di specificare un vector differente. I campi Message Address e Message Upper Address contengono rispettivamente i 32 bit inferiori e superiori dell'indirizzo di memoria a cui deve essere eseguita la scrittura. L'indirizzo deve essere allineato a 4 byte, avrà quindi i primi due bit sempre a 0. Il campo Message Data contiene i 4 byte, dipendenti dall'architettura, che devono essere scritti nell'invio dell'interrupt. Infine, il campo Vector Control è utilizzato per mascherare singolarmente i singoli vector. In particolare, se il bit 0 è settato, allora la funzione non ha il permesso di usare tale entry per inviare un messaggio MSI-X.

- PBA Offset e PBA BIR
  I campi PBA BIR e PBA Offset hanno la stessa funzione dei campi Table BIR e Table Offset, con l'unica differenza che fanno riferimento alla MSI-X PBA invece della MSI-X Table.

  Per ogni pending bit settato, la funzione è in attesa di inviare il messaggio associato all'entry della tabella MSI-X. Al pending bit 0 sarà quindi associata l'entry 0, al pending bit 1 sarà associato l'entry 1, e così via.

Come descritto precedentemente, i valori che devono essere inseriti all'interno dei vector MSI-X sono specifici dell'architettura. Nei sistemi RISC-V questi dettagli

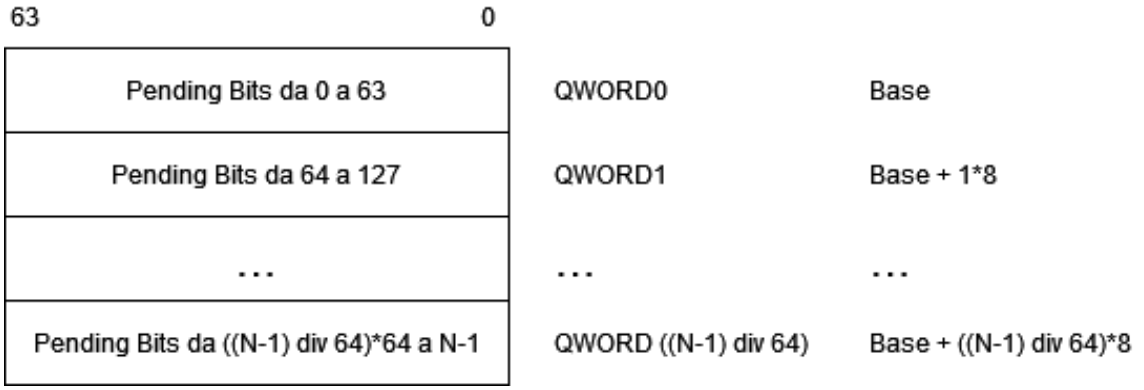

Figura 5.5: Struttura della MSI-X PBA

sono specificati nel documento AIA (Advanced Interrupt Architecture) [6]. Nei sistemi che supportano tale specifica e gli MSI, ogni HART possiede un Incoming MSI Controller (IMSIC) che assume il compito di controllore degli interrupt esterni per quello specifico HART, come mostrato in Figura 5.6.

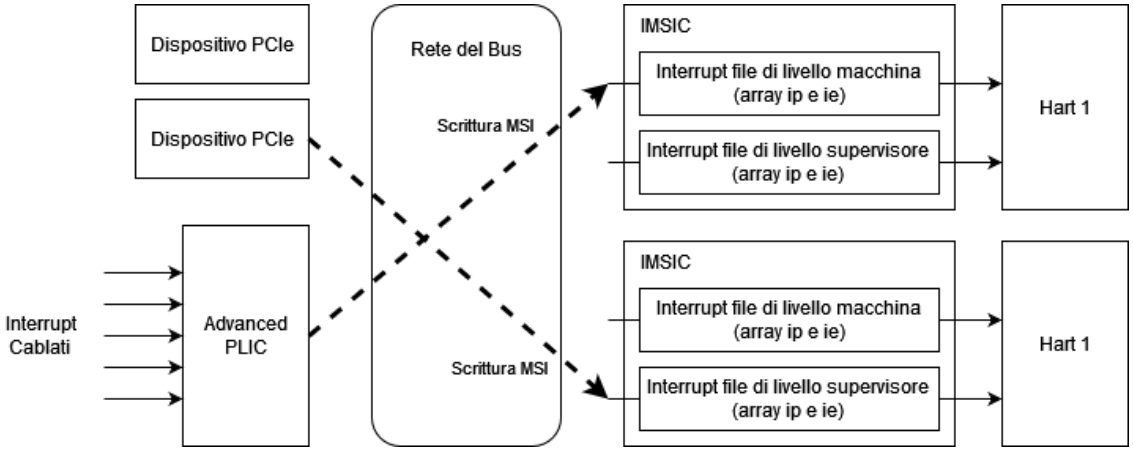

Figura 5.6: Recapito degli MSI quando un HART possiede un dispositivo IMSIC

Fondamentalmente, gli MSI sono delle scritture in memoria a degli indirizzi specifici che l'hardware interpreta come interrupt. A tale scopo, ad ogni IMSIC è assegnato uno o più indirizzi all'interno dello spazio di indirizzi in memoria della macchina, e quando una scrittura avviene in uno di questi indirizzi con il formato corretto, l'IMSIC ricevente traduce la scrittura in un interrupt esterno per il rispettivo HART. Ogni IMSIC possiede indirizzi separati per gli MSI diretti al livello macchina e supervisore.

Prima di proseguire riprendiamo il funzionamento degli interrupt visto nel capitolo precedente e espandiamolo aggiungendo gli interrupt esterni. I bit SEI* (SEIP per pending e SEIE per enable), all'interno dei registri `sip` e `sie`, sono i bit per le interruzioni esterne di livello supervisore. Il bit SEIP è di tipo read-only e viene resettato tramite un metodo specifico della piattaforma che vedremo in seguito. Nel caso di un'interruzione esterna, all'interno del registro `scause` il valore assunto dal campo Interrupt è 1 e il campo Exception Code è 9. La specifica AIA amplia il

significato del campo Exception Code. Tale valore assume infatti il nome di "major identity number" e identifica univocamente la causa dell'interrupt. Nel caso di interruzioni esterne però questo non è sufficiente, si fa quindi uso di un ulteriore valore chiamato "minor identity number" che distingue le varie sorgenti (solitamente dispositivi di I/O). Tali valori sono gestiti direttamente dal controllore delle interruzioni esterne, che nel nostro caso è l'IMSIC.

Gli MSI indirizzati a uno specifico livello di privilegio di un HART sono registrati all'interno di una struttura dati chiamata "interrupt file". Un interrupt file consiste principalmente in due array della stessa dimensione: uno di interrupt pending e uno di interrupt enable. Ogni bit all'interno di questi array corrisponde a un differente "identity number" che permette di distinguere gli MSI da sorgenti differenti. Essendo l'IMSIC il controllore delle interruzioni esterne, tali identity number diventano poi i minor identities per gli interrupt esterni all'HART. L'indirizzo MSI di scrittura è l'indirizzo fisico di un particolare registro a 16-bit fisicamente collegato a uno specifico interrupt file. Il valore di un MSI è semplicemente l'identity number da rendere pending all'interno di tale interrupt file. Tramite la configurazione degli MSI vector ai vari dispositivi il software è in grado di controllare: quale HART riceve un particolare interrupt, a che livello tale interrupt è ricevuto e l'identity number che rappresenta tale MSI all'interno dell'interrupt file. All'interno di un singolo interrupt file, le priorità degli interrupt sono determinate direttamente dall'identity number. In particolare, un identity number più piccolo corrisponde a una priorità più alta. Ogni interrupt file all'interno di un IMSIC possiede uno o due registri di 32-bit mappati in memoria per ricevere la scrittura di un MSI. Questi registri si trovano all'interno di una regione di 4-KiB allineata naturalmente dedicata al singolo interrupt file (quindi una pagina per interrupt file). I registri sono disposti come mostrato nella Tabella 5.1.

Offset	**Dimensione**	**Nome del registro**
`0x000`	4 byte	`seteipnum_le`
`0x004`	4 byte	`seteipnum_be`

Tabella 5.1: Registri dei singolo interrupt file

Il registro `seteipnum_le` è utilizzato per MSI di tipo Little Endian, mentre il registro `seteipnum_be` è utilizzato per MSI di tipo Big Endian. La specifica VirtIO stabilisce che gli MSI effettuati da dispositivi ad essa congruenti siano di tipo Little Endian. Noi siamo quindi interessati solamente al primo dei due registri. Se l'identity number i è implementato, scrivere il valore i con formato Little Endian setta il pending bit relativo all'interrupt i. Se l'identity number non è implementato, la scrittura è semplicemente ignorata. All'interno di ogni IMSIC, per ogni livello che può ricevere un interrupt è presente un insieme di CSR separato. Nel nostro caso, i CSR di livello supervisore di livello supervisore interagiscono con l'interrupt file di livello supervisore dell'IMSIC. Per tale livello, i CSR rilevanti sono i seguenti:

- `siselect` e `sireg`: questi registri forniscono una finestra per accedere a più CSR in maniera indiretta. Il valore contenuto da `siselect` determina quale

registro è attualmente accessibile in lettura e scrittura attraverso il registro `sireg`;

- `stopei`: il suo valore indica l'interrupt avente maggiore priorità che è sia pending che abilitato, e che rientra nei limiti di `eithreshold`. Se nessun interrupt rispetta tali criteri, una lettura da `stopei` ritorna 0. Altrimenti, il formato del dato letto è il seguente:

bit 26:16	Interrupt identity
bit 10:0	Priorità dell'interrupt (come interrupt identity)

Tabella 5.2: Formato del registro `stopei`

L'interrupt identity in `stopei` rappresenta la minor identity per un interrupt esterno nell'HART. Una scrittura eseguita sul registro fa sì che venga resettato l'interrupt pending bit relativo all'interrupt identity che veniva mostrata al suo interno. Il valore scritto non è rilevante. Solitamente, quindi, viene eseguita contemporaneamente una scrittura e una lettura (tramite CSRRW, CSRRS o CSRRC) in modo da ottenere il valore del pending bit che è stato resettato.

I registri a 64-bit accessibili indirettamente tramite `siselect` e `sireg` sono mostrati in Tabella 5.3.

Valore	**Registro**
0x70	eidelivery
0x72	eithreshold
0x80	eip0
0x81	eip1
...	...
0xBF	eip63
0xC0	eie0
0xC1	eie1
...	...
0xFF	eie63

Tabella 5.3: Registri accessibili indirettamente

Il registro `eidelivery` controlla se gli interrupt provenienti da questo interrupt file sono recapitati dall'IMSIC all'HART corrispondente in modo che questi appaiano come interrupt esterni pending all'interno del CSR `mip`. I valori possibili sono 0 per interrupt disabilitati e 1 per interrupt abilitati. Il registro `eithreshold` è un registro che determina il minimo livello di priorità (massimo interrupt identity) che permette a un interrupt di essere riportato dal file all'HART connesso. Quando il registro ha valore P, le interrupt identity P e superiori non contribuiscono alla segnalazione di interrupt, come se fossero mascherati nel vettore `eie`. Quando assume valore nullo tutte le interrupt identities possono contribuire al creare interrupt. Nel caso di architettura a 64-bit (quindi nel nostro caso) i registri `eip`k aventi k

dispari non sono abilitati. Per i k pari, i registri `eip`k contengono i bit pending per gli interrupt aventi interrupt identity da $k \times 32$ a $k \times 32 + 63$. I registri `eie`k seguono lo stesso funzionamento, ma contengono i corrispondenti bit di enable.

Nella virt board offerta da QEMU di default non è presente il dispositivo IMSIC. Per attivarlo è necessario aggiungere un parametro all'avvio della macchina virtuale: `aia=aplic-imsic`. L'aggiunta del dispositivo IMSIC comporta anche la sostituzione del dispositivo PLIC con la sua versione avanzata: l'APLIC (Advanced PLIC).

5.2 VirtIO

VirtIO (Virtual I/O Device) [7] è un protocollo nato per standardizzare la comunicazione tra sistema operativo host e guest all'interno degli ambienti virtuali. I dispositivi definiti da questa specifica sono quindi pensati per essere implementati virtualmente ma sono trattati come dispositivi fisici. VirtIO è caratterizzato dall'essere: lineare, efficiente, standard e estensibile. Ogni dispositivo VirtIO è caratterizzato dalle seguenti parti:

- Device Status field;
- Feature Bits;
- Notifications;
- Device Configuration space;
- una o più Virtqueue.

5.2.1 Device Status Field

Il campo Device Status provvede a fornire un'indicazione di basso livello dei passaggi completati durante l'inizializzazione. I bit definiti sono i seguenti:

- `ACKNOWLEDGE` 0: indica che il sistema operativo guest ha individuato il dispositivo e lo ha riconosciuto come un dispositivo VirtIO valido.
- `DRIVER` 1: indica che il sistema operativo guest sa come controllare il dispositivo.
- `DRIVER_OK` 2: indica che il driver è impostato e pronto a controllare il dispositivo.
- `FEATURE_OK` 3: indica che il driver ha riconosciuto tutte le feature che è in grado di comprendere, e che la negoziazione delle feature è completa.
- `DEVICE_NEEDS_RESET` 6: indica che il dispositivo ha incontrato un errore dal quale non può recuperare senza un reset.
- `FAILED` 7: indica che qualcosa non è andato a buon fine all'interno del guest durante l'inizializzazione, e che abbandona il dispositivo.

5.2.2 Feature Bits

Ogni dispositivo VirtIO offre tutte le feature che comprende. Durante la fase di inizializzazione, il driver legge tali feature e indica al dispositivo quelle che accetta (non necessariamente tutte). L'unico modo per eseguire nuovamente la negoziazione è resettare il dispositivo. Le feature sono, in generale, delle caratteristiche peculiari di quel dispositivo che possono essere sia comuni tra i vari dispositivi, sia specifiche per quella tipologia di dispositivo. Ad ogni feature è assegnato un bit (attualmente da 0 a 127, ma è possibile che in futuro aumentino) che la identifica univocamente. Nel nostro caso la tastiera fa uso del feature bit 32 chiamato `VIRTIO_F_VERSION_1`. Il suo scopo è quello di indicare che il dispositivo è conforme alle specifiche 1.0, così da individuare facilmente i dispositivi di tipo legacy. Se un dispositivo presenta tale bit (e dovrebbe), il driver è obbligato a accettarlo durante la negoziazione.

5.2.3 Notifications

La nozione di inviare una notifica (da driver a dispositivo e viceversa) gioca un ruolo importante all'interno delle periferiche. Il modo con cui queste sono inviate è specifica del metodo di trasporto. Ci sono tre tipologie di notifiche:

- notifiche di cambio configurazione;
- notifiche di buffer disponibile;
- notifiche di buffer utilizzato.

Le notifiche relative ai cambi di configurazione e ai buffer utilizzati sono inviate dal dispositivo al driver. La prima indica che lo spazio di configurazione del dispositivo ha subito un cambio non comandato dal driver; la seconda indica che un buffer è stato utilizzato all'interno della virtqueue obiettivo. Le notifiche relative ai buffer disponibili sono inviate dal driver al dispositivo. Queste sono usate per indicare che un nuovo buffer è disponibile all'interno della virtqueue obiettivo. Il modo con cui le notifiche sono recapitate ai vari estremi è specifico del metodo di trasporto. Nel nostro caso quest'ultimo è PCI, e le notifiche da driver a dispositivo sono inviate tramite MSI-X.

5.2.4 Device Configuration Space

Lo spazio di configurazione del dispositivo è solitamente utilizzato per i parametri che cambiano raramente durante l'utilizzo del dispositivo o che sono rilevanti a livello di inizializzazione. Se dei campi sono opzionali la loro presenza è indicata tramite i feature bits.

5.2.5 Virtqueue

Il meccanismo con cui i dati sono scambiati è chiamato virtqueue. Ogni dispositivo può avere zero o più virtqueue, ognuna delle quali è identificata tramite un indice. Il driver rende disponibile una richiesta al dispositivo aggiungendo un buffer che la descrive alla coda e, possibilmente, inviando una notifica di buffer disponibile.

Il dispositivo esegue la richiesta e, al termine, marca il buffer come utilizzato. Il dispositivo può, a questo punto, inviare una notifica di buffer utilizzato al driver. Per ogni buffer utilizzato il dispositivo indica il numero di byte che ha scritto in memoria al suo interno.

Attualmente sono presenti due formati per le virtqueue chiamati split virtqueue e packed virtqueue. Noi vedremo solamente il primo dei due, essendo quello che viene implementato all'interno della tastiera che utilizziamo. Le virtuque di tipo split suddividono la queue in alcune parti, ognuna delle quali può essere scritta dal driver o dal dispositivo, ma mai da entrambi. Ogni coda ha un parametro di 16-bit, chiamato queue size, che ne identifica il numero di entry e la dimensione totale della coda. Ogni coda è composta da tre parti:

- Descriptor Table;
- Available Ring;
- Used Ring.

Ogni parte è allocata in maniera contigua all'interno della memoria e ha differenti requisiti di allineamento. L'allineamento in memoria e la dimensione, in byte, di ogni parte della virtqueue è riassunta in Tabella 5.4.

Parte della virtqueue	**Allineamento**	**Dimensione**
Descriptor Table	16	$16 \times$ (queue size)
Available Ring	2	$6 + 2 \times$ (queue size)
Used Ring	4	$6 + 8 \times$ (queue size)

Tabella 5.4: Dimensione e allineamento delle parti di una virtqueue

Il parametro queue size corrisponde al massimo numero di buffer all'interno della virtqueue. Il suo valore deve sempre essere una potenza di 2. Quando il driver vuole mandare un buffer al dispositivo, riempie una entry all'interno della tabella dei descrittori e scrive l'indice di tale descrittore all'interno dell'available ring. Invia poi una notifica al dispositivo. Quando il dispositivo ha utilizzato un buffer, scrive l'indice del descrittore all'interno del used ring e invia una notifica di buffer utilizzato al driver.

La Descriptor Table, realizzata come un array di `virtq_desc` (Codice 5.2), fa riferimento ai buffer che il driver sta utilizzando per il dispositivo. Il campo `addr` contiene l'indirizzo fisico, e `len` la sua dimensione. Ogni descrittore specifica, attraverso il campo `flags`, alcune proprietà del buffer. Se il bit 1 di tale campo è settato allora il buffer è di tipo write-only (per il dispositivo), altrimenti è read-only. I descrittori possono essere concatenati con `next`. Questo campo può essere utilizzato sia per creare una lista di descrittori vuoti, sia per creare una serie di buffer che il dispositivo utilizzerà in una singola operazione (come fosse un singolo buffer la cui dimensione è ottenuta sommando delle dimensioni dei buffer appartenenti alla coda).

```
struct virtq_desc {
    natq addr;
    natl len;
    natw flags;
    natw next;
};
```

Codice 5.2: Definizione struttura `virtq_desc`

Il driver utilizza il campo `ring` della struttura `virtq_avail` (Codice 5.3) per offrire i buffer al dispositivo. Viene, infatti, scritto solamente dal driver e letto dal dispositivo. Ogni entry dell'array `ring` fa riferimento alla testa di una catena di descrittori. Il campo `idx` indica l'indice dell'array `ring` dove il driver andrà a inserire il prossimo indice del descrittore (modulo la dimensione della coda). Il suo valore parte da 0 e aumenta. Il bit 0 del campo `flags`, se settato, indica al dispositivo che il driver non vuole ricevere delle notifiche di tipo buffer utilizzato.

```
struct virtq_avail {
    natw flags;
    natw idx;
    natw ring[];
};
```

Codice 5.3: Definizione struttura `virtq_avail`

Il campo `ring` della struttura `virtq_used` (Codice 5.4) è dove il dispositivo restituisce i buffer una volta che ne ha fatto uso. Viene, infatti, scritto solamente dal dispositivo e letto dal buffer. Ogni entry dell'array è una coppia di valori (Codice 5.5): il campo `id` indica l'indice della testa di descrittori che rappresentano il buffer, e il campo `len` contiene il numero di byte scritti all'interno del buffer. Il campo `idx` indica l'indice dell'array `ring` dove il dispositivo andrà a inserire il prossimo dato (modulo la dimensione della coda). Il suo valore parte da 0 e aumenta. Il bit 0 del campo `flags`, se settato, indica al driver che il dispositivo non vuole ricevere delle notifiche di tipo buffer disponibile.

```
struct virtq_used {
    natw flags;
    natw idx;
    struct virtq_used_elem ring[];
};
```

Codice 5.4: Definizione struttura `virtq_used`

```
struct virtq_used_elem {
    natl id;
    natl len;
```

```
};
```

Codice 5.5: Definizione struttura `virtq_used_elem`

Il Codice 5.6 contiene la definizione della struttura `virtq`. Questa struttura permette di raccogliere in un singolo tipo tutte le parti che compongono una virtqueue.

```
struct virtq {
    unsigned int num;

    struct virtq_desc *desc;
    struct virtq_avail *avail;
    struct virtq_used *used;
};
```

Codice 5.6: Definizione struttura `virtq`

Tutte le strutture dati presentate sono allocate e inizializzate dal driver e poi condivise, tramite un metodo che è "transport specific", con il dispositivo. Il modo in cui il driver presenta un nuovo buffer a quest'ultimo verrà presentato nella sezione relativa all'implementazione. È importante specificare che nel nostro nucleo non si fa utilizzo di tutte funzionalità messe a disposizione da VirtIO, ma solamente di quelle essenziali[6].

5.2.6 Input Device

Nella specifica VirtIO la tastiera è vista come un dispositivo appartenente alla categoria Input Device. Un Input Device è un dispositivo che emula una interfaccia uomo-macchina come mouse, tablet o, per l'appunto, tastiere. Le virtqueue utilizzate da tale dispositivo sono due:

- eventq (indice 0): utilizzata dal dispositivo per inviare nuovi dati al sistema operativo guest, come, ad esempio, pressione e rilascio di un tasto o movimento del mouse;
- statusq (indice 1): utilizzata dal driver del dispositivo per inviare status feedback, come, ad esempio, l'aggiornamento dei LED in una tastiera.

Entrambe le code utilizzano lo stesso formato dei buffer. La struttura è chiamata `virtio_input_event` ed è composta da tre campi: `type`, `code` e `value` (Codice 5.7). Questi campi sono riempiti secondo la "Linux input layer interface", comunemente chiamata "evdev interface".

```
struct virtio_input_event {
    natw type;
```

[6] A tal proposito all'interno delle strutture dati `virtq_used` e `virtq_avail` si è omesso un campo opzionale (da negoziare tramite feature bits) che nel nostro caso non sarà utilizzato.

```
3       natw code;
4       natl value;
5   };
```

Codice 5.7: Definizione struttura `virtio_input_event`

Nel caso specifico della tastiera i campi assumono i seguenti significati:

- Il campo `type` indica il tipo di evento che riguarda il dispositivo. Nel nostro caso siamo interessati al tipo `EV_KEY` che indica la pressione o il rilascio di un tasto. I punti seguenti sono descritti supponendo proprio l'invio di un tale evento.
- Il campo `code` rappresenta il codice del tasto con cui si è interagito, come, ad esempio, `KEY_Q`.
- Infine, `value` può assumere valore `KEY_PRESSED` (1) o `KEY_RELEASED` (0) a seconda che l'evento identifichi la pressione o il rilascio di un tasto.

I dispositivi di tipo Input Device presentano una struttura dati usata per la configurazione specifica del dispositivo, chiamata `virtio_input_config` dalla specifica. Noi non faremo uso di tale struttura, per questo non è riportata all'interno di questa tesi. Ciò però di cui faremo uso, è la struttura dati di configurazione comune a tutti i dispositivi che utilizzano come mezzo di trasporto PCI. La sua definizione è riportata all'interno del Codice 5.8.

```
1   struct virtio_pci_common_cfg {
2       natl device_feature_select;
3       natl device_feature;
4       natl driver_feature_select;
5       natl driver_feature;
6       natw config_msix_vector;
7       natw num_queues;
8       natb device_status;
9       natb config_generation;
10
11      natw queue_select;
12      natw queue_size;
13      natw queue_msix_vector;
14      natw queue_enable;
15      natw queue_notify_off;
16      natq queue_desc;
17      natq queue_driver;
18      natq queue_device;
19      natw queue_notify_config_data;
20      natw queue_reset;
21
22      natw admin_queue_index;
23      natw admin_queue_num;
24  };
```

Codice 5.8: Definizione struttura `virtio_pci_common_cfg`

5.3 Implementazione

5.3.1 Configurazione IMSIC

Come descritto all'interno del capitolo relativo agli MSI-X, nelle architetture RISC-V il dispositivo destinatario delle scritture in memoria tramite MSI è l'IMSIC. Lo scopo del Codice 5.9, contenuto nel file `kernel/start.s`, è la sua configurazione.

```
1     ...
2     li t0, 0x70
3     csrw siselect, t0
4     li t1, 1
5     csrw sireg, t1
6     li t0, 0x72
7     csrw siselect, t0
8     li t1, 0
9     csrw sireg, t1
10
11    li t0, 0xC0
12    csrw siselect, t0
13    li t1, 0b10
14    csrw sireg, t1
15    ...
```

Codice 5.9: Configurazione IMSIC

Abilitiamo il dispositivo a inviare interrupt esterni all'HART scrivendo il valore 1 all'interno del registro `eidelivery` (righe 2-5). Facciamo "l'unmask" di tutti gli identity number azzerando il registro `eithreshold` (righe 6-9). Infine, specificatamente per la tastiera, abilitiamo l'identity number 1 (lo 0 non è mai supportato) settando tale bit all'interno del registro `eie0` (righe 11-14). Sarà poi compito dell'inizializzazione della tabella MSI-X far sì che tale valore venga scritto nel registro `seteipnum_le` in caso di interrupt della tastiera.

5.3.2 PCI Device Discovery

L'implementazione della tastiera parte dalla fase di device discovery, che avviene all'interno del file `kernel/pci.cpp`. La tastiera proposta all'interno delle specifiche VirtIO presenta i seguenti valori: Device ID `0x1052` e Vendor ID `0x1AF4`. Il valore del campo Device ID può essere ottenuto tramite la somma della costante `0x1040` con il valore Virtio Device ID, che nel caso della tastiera è 18, essendo uno specifico tipo di Input Device. Il valore del Vendor ID è invece costante per i dispositivi di tipo VirtIO. Una volta individuato il dispositivo all'interno della funzione `pci_init()`, viene richiamata la funzione `kbd_setup()` (Codice 5.10), il cui compito consiste nell'inizializzare i BAR del dispositivo in maniera statica.

```
1  void kbd_setup(PCI_config *conf)
2  {
3    conf->bar1 = kbd::MSIX;
4    conf->bar4 = kbd::MMIO;
```

```
5   conf->bar5 = kbd::MMIO >> 32;
6   conf->command |= 0b110;
7 }
```

Codice 5.10: Funzione `kbd_setup()`

La tastiera implementata richiede che le vengano assegnate due zone nello spazio di memoria in cui possa presentare i propri registri. La prima viene utilizzata per mappare le strutture dati di MSI-X. Il suo indirizzo, che deve essere di 32-bit, viene comunicato al dispositivo tramite il BAR 1 (riga 3). La seconda zona viene utilizzata dalla tastiera per mappare le informazioni inerenti allo scambio di dati tramite VirtIO. I BAR 4 e 5 sono adibiti a contenere l'indirizzo su 64-bit di tale zona (righe 4-5).

Oltre all'assegnazione degli indirizzi, è necessario abilitare il dispositivo a rispondere ad accessi nello spazio di memoria, così che il driver possa interagire con le strutture dati mostrate dalla tastiera (mappate nei BAR), e a fare da bus master, in modo che possa eseguire scritture in memoria per inviare i dati prodotti, e le interruzioni tramite MSI. Per fare ciò settiamo i bit 1 e 2 (Memory Space e Bus Master) nel registro Command dello spazio di configurazione PCI (riga 6).

5.3.3 Funzioni della tastiera

Le funzioni generiche della tastiera, utilizzabili anche in programmi di tipo "bare" (senza, cioè, la presenza del nucleo), sono state inserite all'interno della cartella `libCE`. Ognuna di esse si trova in un file separato avente come prefisso `kbd`. Per farne uso è necessario fare l'include del file `include/kbd.h` e `include/libce.h`, contenenti le dichiarazioni delle variabili e delle funzioni. Specifichiamo che tutto il codice che riguarda la tastiera si trova all'interno del namespace `kbd`, come accadeva anche nel vecchio sistema, ma, per semplicità, non è stato riportato nelle sezioni di codice seguenti. Analizziamo quindi i file contenenti il codice per il funzionamento della tastiera.

File `kbd_vars.h` Le variabili `MAX_CODE`, `shift`, `tab`, `tabmin` e `tabmax` sono rimaste invariate dal vecchio sistema, sia come dichiarazione che come utilizzo. La nuova tipologia di tastiera ha però richiesto l'aggiunta di ulteriori variabili.

- `QUEUE_SIZE`: dimensione standard delle code eventq e statusq;
- `PCI`: contiene l'indirizzo nello spazio di configurazione PCI in cui è mappata la tastiera;
- `MMIO` e `MSIX`: sono gli indirizzi assegnati ai BAR all'interno della fase di device discovery;
- `eventq_notify_addr`: il suo valore indica l'indirizzo in memoria a cui deve essere inviata la notifica di buffer disponibile per la coda eventq;
- `msix_cap`: contiene un riferimento alla capability relativa a MSI-X, utilizzata per abilitare e disabilitare gli interrupt;

- `buf`: array di buffer di tipo `virtio_input_event` utilizzati per lo scambio di dati con la tastiera;
- `next_idx_read`: contiene l'indice (modulo `QUEUE_SIZE`) del prossimo dato da leggere all'interno dell'array `ring` della struttura `eventq.used`;
- `eventq` e `statusq`: virtqueue specifiche della tastiera. Attualmente solamente la coda `eventq` è utilizzata dalle funzioni della tastiera;
- `get_real_addr`: funzione utilizzata per convertire l'indirizzo virtuale ottenuto dalla funzione `alloca()` nel suo indirizzo fisico. All'interno del nucleo tale funzione sarà `trasforma()`, mentre in un programma bare sarà semplicemente la funzione identità.

```
1  const natl MAX_CODE = 42;
2  const natw QUEUE_SIZE = 64;
3
4  const paddr PCI  = 0x30010000UL;
5  const paddr MMIO = 0x60000000UL;
6  const paddr MSIX = 0x60010000UL;
7
8  extern bool shift;
9  extern natw tab[MAX_CODE];
10 extern char tabmin[MAX_CODE];
11 extern char tabmai[MAX_CODE];
12
13 extern paddr eventq_notify_addr;
14 extern MSIX_capability *msix_cap;
15
16 extern virtio_input_event buf[QUEUE_SIZE];
17 extern natw next_idx_read;
18 extern virtq eventq;
19 extern virtq statusq;
20
21 extern paddr (*get_real_addr)(void *ff);
22 bool init(paddr (*func)(void *ff));
```

Codice 5.11: File `kbd_vars.h`

Il file contiene anche la dichiarazione di alcune macro (non riportate nel Codice 5.11) relative all'evdev layer in modo da semplificarne la scrittura e aumentarne la leggibilità delle funzioni per la tastiera. Le macro sono dichiarate in accordo con la specifica che può essere trovata al seguente link: `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/input-event-codes.h`

File `kbd_vars.cpp` Il file `kbd_vars.cpp` contiene la definizione delle variabili dichiarate all'interno di `kbd.h`. Oltre alla dichiarazione delle nuove variabili che non erano presenti nel vecchio sistema, è degna di nota la dichiarazione della variabile `tab`. La variabile nel nuovo sistema è stata adattata al funzionamento del layer

evedev presente nei sistemi Linux. Il suo tipo è stato cambiato da `natb` a `natw` in accordo con la dimensione del campo `code` che, nel caso di pressione di un tasto, indica quale esso sia.

```
...
natw tab[MAX_CODE] = {
    KEY_1, KEY_2, KEY_3, KEY_4, KEY_5, KEY_6, KEY_7, KEY_8, KEY_9,
    KEY_0, KEY_MINUS, KEY_EQUAL,
    KEY_Q, KEY_W, KEY_E, KEY_R, KEY_T, KEY_Y, KEY_U, KEY_I, KEY_O,
    KEY_P,
    KEY_A, KEY_S, KEY_D, KEY_F, KEY_G, KEY_H, KEY_J, KEY_K, KEY_L,
    KEY_Z, KEY_X, KEY_C, KEY_V, KEY_B, KEY_N, KEY_M, KEY_SPACE,
    KEY_ENTER, KEY_BACKSPACE, KEY_ESC
};
...
```

Codice 5.12: Parte del file `kbd_vars.cpp`

File `kbd_init.cpp` La funzione `init()` è stata suddivisa in più sezioni in modo da migliorarne la leggibilità.

Assegniamo il parametro passato alla funzione alla variabile `get_real_addr`, che verrà utilizzata all'interno delle funzioni di supporto VirtIO per ricavare l'indirizzo fisico delle strutture dati a partire da quello virtuale. Inizializziamo anche le variabili utilizzate durante la funzione `init()`. In particolare, la variabile `conf` contiene un puntatore alla base dello spazio di configurazione PCI, mentre la variabile `comm_cfg` un puntatore alla struttura dati di configurazione condivisa da tutte le periferiche.

```
get_real_addr = func;

PCI_config *conf = (PCI_config *) PCI;
virtio_pci_common_cfg *comm_cfg = (virtio_pci_common_cfg *) MMIO;
natl notify_off_multiplier;
```

Codice 5.13: Inizializzazione variabili della tastiera (funzione `init()`)

```
if (conf->revision_id < 1) {
    return false;
}
if (conf->subsystem_id < 0x40) {
    return false;
}
```

Codice 5.14: Controllo dei campi di configurazione PCI della tastiera (funzione `init()`)

Dopo aver inizializzato le variabili, eseguiamo un controllo sui campi Revision ID e Subsystem ID, come specificato all'interno delle specifiche VirtIO.

In seguito eseguiamo la scansione della capability list. Per verificarne la presenza controlliamo che il bit 4 del campo PCI Status sia settato (riga 1). Scorriamo la lista in cerca delle capability relative a MSI-X e alle notifiche VirtIO. La prima ci permette di inizializzare la variabile `msix_cap` e di abilitare il dispositivo a inviare interrupt tramite MSI (riga 9), seppur per il momento mascherandoli (riga 10). Attraverso la capability `VIRTIO_PCI_CAP_NOTIFY_CFG` otteniamo il valore del parametro `notify_offset_multiplier` (righe 15-16), utilizzato per inviare notifiche alla tastiera.

```
1  if ((conf->status & 0b10000) == 0) {
2      return false;
3  }
4  paddr conf_base = (paddr) conf;
5  capability_elem *cap = (capability_elem *) (conf_base + conf->
       capabilities_pointer);
6  while (true) {
7      if (cap->id == 0x11) {
8          msix_cap = (MSIX_capability *) cap;
9          msix_cap->message_control |= 0x8000;
10         msix_cap->message_control |= 0x4000;
11     } else if (cap->id == 0x09) {
12         virtio_pci_cap *vpc = (virtio_pci_cap *) cap;
13         switch (vpc->cfg_type) {
14         case VIRTIO_PCI_CAP_NOTIFY_CFG:
15             virtio_pci_notify_cap *vpnc = (virtio_pci_notify_cap *) vpc
       ;
16             notify_off_multiplier = vpnc->notify_off_multiplier;
17             break;
18         }
19     }
20     if (!cap->next_pointer)
21         break;
22     cap = (capability_elem *) (conf_base + cap->next_pointer);
23 }
```

Codice 5.15: Scansione Capabilities List della tastiera (funzione `init()`)

Segue l'inizializzazione del dispositivo dal punto di vista del protocollo VirtIO. Per prima cosa azzeriamo il dispositivo impostando a 0 il registro `device_status` (riga 1). Attendiamo che questo venga effettivamente azzerato dal dispositivo che ha riconosciuto la volontà del driver di eseguire il reset (riga 2). Settiamo in successione i bit 0 `ACKNOWLEDGE_STATUS_BIT` e 1 `DRIVER_STATUS_BIT` (righe 4 e 6). Normalmente, in seguito avviene la negoziazione dei feature bits. Nel nostro caso ci comportiamo come se già sapessimo le feature offerte dal dispositivo e saltiamo la parte di individuazione di queste. Settiamo quindi solamente il bit 32 che corrisponde alla feature standard `VIRTIO_F_VERSION_1` (righe 8-9), essendo obbligatoria. Infine, settiamo il bit 3 `FEATURES_OK_STATUS_BIT` (riga 11). È poi necessario controllare che tale bit sia ancora settato. Se così non fosse vorrebbe indicare che la

tastiera non ha accettato quel sottoinsieme di feature che il driver ha scelto.

```
comm_cfg->device_status = 0;
while (comm_cfg->device_status);

comm_cfg->device_status |= ACKNOWLEDGE_STATUS_BIT;

comm_cfg->device_status |= DRIVER_STATUS_BIT;

comm_cfg->driver_feature_select = 0x1;
comm_cfg->driver_feature = (1 << (VIRTIO_F_VERSION_1 - 32));

comm_cfg->device_status |= FEATURES_OK_STATUS_BIT;

if ((comm_cfg->device_status & FEATURES_OK_STATUS_BIT) == 0) {
    goto error_set_failed;
}
```

Codice 5.16: Inizio configurazione VirtIO della tastiera (`init()`)

Assegniamo l'indirizzo fisico alla variabile `eventq_notify_addr` secondo i calcoli richiesti dalla specifica. La variabile sarà utilizzata ogniqualvolta un nuovo buffer verrà aggiunto alla coda eventq e la tastiera richiederà l'invio di una notifica.

```
comm_cfg->queue_select = 0;
eventq_notify_addr = MMIO + 0x3000 + comm_cfg->queue_notify_off *
   notify_off_multiplier;
```

Codice 5.17: Calcolo dell'indirizzo delle notifiche della tastiera (funzione `init()`)

Eseguiamo il setup della tabella che descrive gli MSI-X. Per farlo, facciamo uso della funzione di supporto `msix_add_entry` (righe 1 e 4). Impostiamo poi il vettore 0 di tale tabella come quello che deve essere usato in caso di cambio di configurazione della tastiera (riga 7). Individuiamo un errore se, dopo aver impostato tale valore, si legge all'interno di tale registro il valore `VIRTIO_MSI_NO_VECTOR`.

```
if (!msix_add_entry((MSIX_entry *) MSIX, 0, VIRT_IMSIC_S, 1)) {
    goto error_set_failed;
}
if (!msix_add_entry((MSIX_entry *) MSIX, 1, VIRT_IMSIC_S, 1)) {
    goto error_set_failed;
}
comm_cfg->config_msix_vector = 0;
if (comm_cfg->config_msix_vector == VIRTIO_MSI_NO_VECTOR) {
    goto error_set_failed;
}
```

Codice 5.18: Configurazione MSI-X della tastiera (funzione `init()`)

In seguito, eseguiamo l'inizializzazione delle virtqueue del dispositivo. Per entrambe richiamiamo la funzione `create_virtq()` che alloca lo spazio necessario (righe 1 e 10). Abilitiamo poi le virtqueue all'interno della struttura dati di configurazione della tastiera, tramite la funzione di supporto `enable_virtq()` (righe 4 e 13). Nel caso della virtqueue eventq, inizializziamo la struttura dati desc (righe 7-9) chiamando la funzione `add_buf_desc()`, la quale salva l'indirizzo e la dimensione di un buffer all'interno del descrittore.

```
1  if (!create_virtq(eventq, QUEUE_SIZE, 0)) {
2      goto error_set_failed;
3  }
4  if (!enable_virtq(*comm_cfg, 0, QUEUE_SIZE, 1, eventq, get_real_addr))
       {
5      goto error_set_failed;
6  }
7  for (int i = 0; i < QUEUE_SIZE; i++) {
8      add_buf_desc(eventq, i, (natq) &buf[i], sizeof(buf[i]),
       VIRTQ_DESC_F_WRITE, 0, get_real_addr);
9  }
10 if (!create_virtq(statusq, QUEUE_SIZE, 0)) {
11     goto error_set_failed;
12 }
13 if (!enable_virtq(*comm_cfg, 1, QUEUE_SIZE, 1, statusq, get_real_addr))
       {
14     goto error_set_failed;
15 }
```

Codice 5.19: Configurazione delle virtqueue della tastiera (funzione `init()`)

Impostiamo il bit 2 `DRIVER_OK_STATUS_BIT` all'interno del registro di stato (riga 1). Eseguiamo un controllo sul bit 6 `DEVICE_NEEDS_RESET_BIT` per verificare che il dispositivo non abbia incontrato un errore dal quale non può riprendersi senza un reset. In tal caso, abbandoniamo la sua configurazione. Infine, richiamiamo la funzione `add_max_buf()` (riga 6) così da rendere disponibili al dispositivo tutti i buffer allocati.

```
1  comm_cfg->device_status |= DRIVER_OK_STATUS_BIT;
2  if (comm_cfg->device_status & DEVICE_NEEDS_RESET_BIT) {
3      goto error_set_failed;
4  }
5
6  add_max_buf();
```

Codice 5.20: Fine inizializzazione tastiera (funzione `init()`)

In caso di errore in seguito al reset del dispositivo è necessario impostare il bit 7 all'interno del registro di stato (riga 2). Questo indica alla tastiera che il driver ha incontrato un errore durante la sua inizializzazione e che la abbandona.

```
error_set_failed:
    comm_cfg->device_status |= FAILED_STATUS_BIT;
    return false;
```

Codice 5.21: Errore durante l'inizializzazione della tastiera (funzione `init()`)

File `kbd_add_max_buf.cpp` Lo scopo della funzione `add_max_buf()` è quello di aggiungere tutti i buffer disponibili (cioè non già forniti alla tastiera o da leggere) nel descrittore avail della coda eventq (righe 5-8). Al termine del ciclo, ci assicuriamo che le operazioni appena eseguite siano visibili alla tastiera tramite la funzione `memory_barrier()`. Incrementiamo la variabile `eventq.avail->idx` del numero di buffer aggiunti `added` (riga 10), e eseguiamo nuovamente la funzione `memory_barrier()`. Infine, se richiesto dalla tastiera, inviamo una notifica scrivendo l'indice della coda eventq (0) all'indirizzo salvato in `eventq_notify_addr` (riga 13).

```
1  void add_max_buf()
2  {
3      int added = 0;
4      int i = eventq.avail->idx;
5      while (i % QUEUE_SIZE != next_idx_read % QUEUE_SIZE || (i == eventq
       .used->idx && next_idx_read == eventq.used->idx)) {
6          eventq.avail->ring[(eventq.avail->idx + added++) % QUEUE_SIZE]
       = i % QUEUE_SIZE;
7          i++;
8      }
9      memory_barrier();
10     eventq.avail->idx += added;
11     memory_barrier();
12     if (eventq.used->flags == 0)
13         *((natw *) eventq_notify_addr) = 0;
14 }
```

Codice 5.22: File `kbd_add_max_buf.cpp`

File `kbd_disable_intr.cpp` Per disabilitare gli interrupt di tipo MSI-X è sufficiente resettare il bit 14 all'interno del campo `message_control` della capability corrispondente. Notiamo il fatto che, nonostante gli interrupt siano mascherati, è ancora possibile individuare quando uno di essi è pending tramite la struttura PBA.

```
1  void disable_intr()
2  {
3      msix_cap->message_control |= 0x4000;
4  }
```

Codice 5.23: File `kbd_disable_intr.cpp`

File kbd_enable_intr.cpp Per abilitare gli interrupt di tipo MSI-X è sufficiente eseguire l'operazione inversa a quella eseguita dalla funzione `disable_intr()`, settando il bit 14.

```
1 void enable_intr()
2 {
3     msix_cap->message_control &= ~0x4000;
4 }
```

Codice 5.24: File `kbd_enable_intr.cpp`

File kbd_char_read_intr.cpp La funzione `char_read_intr()` è stata leggermente modificata rispetto alla sua versione presente nel sistema precedente. Questo è stato fatto per rispecchiare il modo in cui sono inviati i dati dalla nuova tastiera. Come prima cosa individuiamo il prossimo buffer da leggere e lo salviamo nella variabile `vie` (righe 3-4). Essendo che la tastiera implementata da QEMU scrive due buffer per ogni evento relativo al tasto, il primo contenente l'informazione relativa all'evento vero e proprio, e il secondo relativo a un evento di sincronizzazione, è necessario eseguire un primo filtro sul tipo dell'evento. Nel caso in cui questo sia diverso da `EV_KEY` si ritorna il valore 0. Viene poi controllato se il tasto con cui si è interagito era lo shift sinistro, e, in tal caso, viene aggiornata la variabile `shift` in accordo. Infine, se l'evento del tasto era relativo al rilascio di esso si ritorna 0, altrimenti si ritorna il valore prodotto dalla funzione `conv()`. La funzione `conv()` è presente all'interno del file `kbd_conv.cpp` ed è rimasta invariata durante la migrazione.

```
1  char char_read_intr()
2  {
3      natw idx = eventq.used->ring[next_idx_read++ % QUEUE_SIZE].id;
4      virtio_input_event vie = *((virtio_input_event *) eventq.desc[idx].
       addr);
5      if (vie.type != EV_KEY)
6          return 0;
7      if (vie.code == KEY_LEFTSHIFT)
8          shift = (vie.value == KEY_PRESSED);
9      if (vie.value == KEY_RELEASED)
10         return 0;
11     return conv(vie.code);
12 }
```

Codice 5.25: File `kbd_char_read_intr.cpp`

File kbd_more_to_read.cpp La funzione `more_to_read()` ritorna `true` nel caso in cui sia presente un buffer che è stato utilizzato dalla tastiera ma non ancora letto dal driver. Essendo che la variabile `next_idx_read` tiene conto dell'indice del prossimo elemento di `eventq.used->ring[]` che deve essere letto, ci è sufficiente

confrontarlo con `eventq.used->idx` che contiene il prossimo elemento in cui la tastiera eseguirà una scrittura.

```
bool more_to_read()
{
    return next_idx_read != eventq.used->idx;
}
```

Codice 5.26: File `kbd_more_to_read.cpp`

5.3.4 Modulo I/O

All'interno del modulo I/O è stato necessario riesaminare tutte le funzioni che interagivano con la tastiera, ovvero quelle relative alla console. In particolare, due funzioni sono state modificate in accordo al nuovo dispositivo: `kbd_init()` e `estern_kbd()`. La prima viene utilizzata durante l'inizializzazione del modulo I/O per abilitare la tastiera e fa uso delle funzioni presenti in `libCE` descritte precedentemente.

```
bool kbd_init()
{
    if (!kbd::init(trasforma)) {
        flog(LOG_ERR, "kbd: impossibile configurare la tastiera");
        return false;
    }
    ...
}
```

Codice 5.27: Parte della funzione `kbd_init()` nel file `io/io.cpp`

Rispetto al vecchio sistema la funzione `init()` può anche incontrare un errore. Per questo è stata inserita all'interno di un if che in caso di errore stampa un messaggio sul terminale e ritorna `false`. Inoltre, passiamo come parametro la funzione `trasforma()` per i motivi visti in precedenza.

```
void estern_kbd(int)
{
    des_console *d = &console;
    char a;
    bool fine;

    for (;;) {
        kbd::disable_intr();

        while (kbd::more_to_read()) {
            a = kbd::char_read_intr();

            if (a == 0)
```

```
14                      continue;
15                  ...
16              }
17
18              kbd::add_max_buf();
19              ...
20          }
21  }
```

Codice 5.28: Parte della funzione `estern_kbd()` nel file `io/io.cpp`

I principali cambiamenti riguardano il modo in cui sono letti i caratteri. Per ogni interrupt inviato dalla tastiera potrebbero essere stati utilizzati più buffer, è quindi necessario fare uso di un ciclo (righe 10-16). Ad esempio, per ogni tasto premuto sono presenti almeno due nuovi buffer da leggere: il primo contenente il dato relativo al tasto premuto o rilasciato e il secondo viene usato come sincronizzazione. Il secondo cambiamento riguarda poi la necessità di introdurre la funzione `add_max_buf()` (riga 18) in modo da aggiungere nuovamente alla coda avail i buffer che erano stati scritti dalla tastiera e che sono stati letti dal driver.

5.3.5 Test

Al momento della scrittura di questa tesi le primitive fornite dal modulo I/O non sono ancora state ripristinate. Per questo non è possibile creare un programma utente che invochi la primitiva `readconsole()` (l'unica che fa uso del driver della tastiera). Possiamo comunque eseguire un test, richiamando tale primitiva, al termine della funzione `main()` del modulo I/O. Il Codice 5.29 è quello che è stato utilizzato per svolgere tale test. L'unica nota da fare riguarda la dimensione del buffer utilizzato per la lettura dalla console. Questo è stato, infatti, scelto volutamente di dimensione 66 in modo che sia possibile leggere 65 caratteri dalla tastiera. Questo ci permette di verificare che la eventq utilizzata dalla tastiera, che ha una lunghezza di 64, non presenti problemi in seguito al wrap-around. Il Codice 5.30 contiene l'output su terminale e la Figura 5.7 la console dopo aver terminato l'inserimento dei caratteri.

```
1  ...
2  flog(LOG_INFO, "=== INIZIO TEST READCONSOLE ===");
3
4  char buf[66];
5  natq buf_len = 65;
6  natq read = c_readconsole((char *) trasforma(buf), buf_len);
7  buf[read] = '\0';
8  flog(LOG_INFO, "ho letto %d byte", read);
9  flog(LOG_INFO, "ho letto il messaggio %s", buf);
10
11 flog(LOG_INFO, "==== FINE TEST READCONSOLE ====");
12 ...
```

Codice 5.29: Test della primitiva `readconsole()`

```
INF    2    === INIZIO TEST READCONSOLE ===
INF    2    ho letto 65 byte
INF    2    ho letto il messaggio
   01234567890123456789012345678901234567890123456789012345678901234
INF    2    ==== FINE TEST READCONSOLE ====
```

Codice 5.30: Output sul terminale della primitiva `readconsole()`

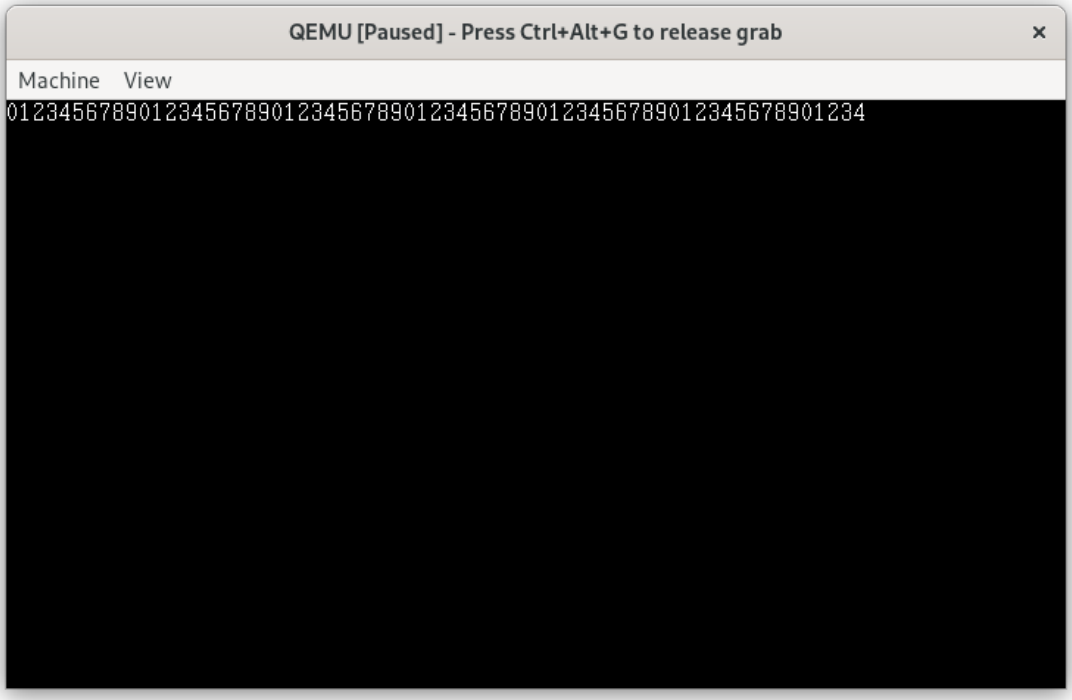

Figura 5.7: Output sulla console della primitiva `readconsole()`

6 Conclusioni e sviluppi futuri

Il risultato di questa tesina è stato il completamento delle funzionalità essenziali del modulo sistema attraverso la reintroduzione del timer e della primitiva `delay()`, e del proseguimento del modulo I/O tramite il completamento della console. La correttezza di entrambe le sezioni è stata poi testata estensivamente attraverso sia un programma utente che faccia uso delle funzionalità offerte dal timer, sia un frammento di codice temporaneo nel modulo I/O che richiamasse le funzioni relative alla console.

Sebbene il lavoro di migrazione ad architettura RISC-V stia pian piano giungendo al termine, ci sono ancora molti aspetti che possono essere migliorati e nuove funzionalità che possono essere aggiunte al nucleo. Se qualche studente avesse la volontà di proseguire questo progetto, alcuni aspetti critici che sono stati individuati durante la realizzazione di questa tesi sono:

- ripristino delle primitive fornite dal modulo I/O. Sebbene il modulo I/O sia presente e funzionante, tuttora non è possibile invocare dal modulo utente le primitive da questo fornite;
- riorganizzazione del sistema nel suo completo per renderlo più coerente con quello usato nel corso di Calcolatori Elettronici, compresa la revisione del Makefile;
- scrittura di un semplice BIOS che si occupi di assegnare gli indirizzi alle varie periferiche, e che fornisca la possibilità ai vari driver di conoscere dove la propria periferica è stata allocata;
- aggiornamento delle funzioni relative all'heap, era infatti presente un bug che è stato sistemato in maniera rapida, senza una fase di testing adeguata;
- aggiornamento del dispositivo PLIC alla sua versione avanzata APLIC;
- sviluppo di ulteriori periferiche, come l'hard disk, che facciano uso del protocollo VirtIO.

7 Ringraziamenti

Un ringraziamento speciale va al Prof. Giuseppe Lettieri che durante tutto lo svolgimento della tesi ha sempre mostrato disponibilità e celerità nel rispondere a qualsiasi tipo di richiesta di aiuto e chiarimento.

Ringrazio la mia famiglia e, in particolare, i miei genitori che, specialmente durante questa avventura, hanno saputo alternare coscienziosamente severità e comprensione.

Un ultimo grazie (ma non per importanza) va a tutti i miei amici che ho frequentato durante questi anni. Coloro che conoscevo prima di iniziare questo percorso e coloro che si sono aggiunti durante di esso.

Elenco dei codici

2.1 Comando di installazione della toolchain 6
2.2 Comando di installazione di QEMU 6
4.1 Output del comando `info qtree` 12
4.2 Inizializzazione timer 13
4.3 Driver del timer 13
4.4 Programmazione prossimo interrupt del timer 14
4.5 Primitiva `delay()` 14
4.6 Case della system call `delay()` 15
4.7 Programma utente (timer) 15
4.8 Output programma utente (timer) 16
5.1 Output del comando `qemu-system-riscv64 -device help` . . 17
5.2 Definizione struttura `virtq_desc` 27
5.3 Definizione struttura `virtq_avail` 27
5.4 Definizione struttura `virtq_used` 27
5.5 Definizione struttura `virtq_used_elem` 27
5.6 Definizione struttura `virtq` 28
5.7 Definizione struttura `virtio_input_event` 28
5.8 Definizione struttura `virtio_pci_common_cfg` 29
5.9 Configurazione IMSIC 30
5.10 Funzione `kbd_setup()` 30
5.11 File `kbd_vars.h` 32
5.12 Parte del file `kbd_vars.cpp` 33
5.13 Inizializzazione variabili della tastiera (funzione `init()`) 33
5.14 Controllo dei campi di configurazione PCI della tastiera (funzione `init()`) 33
5.15 Scansione Capabilities List della tastiera (funzione `init()`) 34
5.16 Inizio configurazione VirtIO della tastiera (`init()`) 35
5.17 Calcolo dell'indirizzo delle notifiche della tastiera (funzione `init()`) 35
5.18 Configurazione MSI-X della tastiera (funzione `init()`) 35
5.19 Configurazione delle virtqueue della tastiera (funzione `init()`) . . . 36
5.20 Fine inizializzazione tastiera (funzione `init()`) 36
5.21 Errore durante l'inizializzazione della tastiera (funzione `init()`) . . 36
5.22 File `kbd_add_max_buf.cpp` 37
5.23 File `kbd_disable_intr.cpp` 37
5.24 File `kbd_enable_intr.cpp` 38
5.25 File `kbd_char_read_intr.cpp` 38
5.26 File `kbd_more_to_read.cpp` 39
5.27 Parte della funzione `kbd_init()` nel file `io/io.cpp` 39
5.28 Parte della funzione `estern_kbd()` nel file `io/io.cpp` 39
5.29 Test della primitiva `readconsole()` 40
5.30 Output sul terminale della primitiva `readconsole()` 41

Elenco delle figure

4.1 Registro di stato di livello supervisore (`sstatus`) 9
4.2 Registri di interrupt-pending e interrupt-enable di livello supervisore (`sip` e `sie`) 9
4.3 Porzione standard (bit 15:0) dei registri `sip` e `sie` 10
4.4 Registro di causa di livello supervisore (`scause`) 10
5.1 Esempio di capability list 18
5.2 Struttura della MSI-X capability 19
5.3 Struttura del campo Message Control della MSI-X Capability 19
5.4 Struttura della tabella MSI-X 20
5.5 Struttura della MSI-X PBA 21
5.6 Recapito degli MSI quando un HART possiede un dispositivo IMSIC 21
5.7 Output sulla console della primitiva `readconsole()` 41

Elenco delle tabelle

1.1 RISC-V livelli di privilegio 4
1.2 Assegnazione dello spazio di memoria all'interno della virt board . . . 5
3.1 Organizzazione delle cartelle del sistema 8
4.1 Mappa del registro Time dell'ACLINT MTIMER 10
4.2 Mappa dei registri Compare dell'ACLINT MTIMER 11
4.3 Un dispositivo SiFive CLINT corrisponde a due dispositivi ACLINT . 11
5.1 Registri dei singolo interrupt file 22
5.2 Formato del registro `stopei` 23
5.3 Registri accessibili indirettamente 23
5.4 Dimensione e allineamento delle parti di una virtqueue 26

Bibliografia

[1] *PCI Local Bus Specification*, ver. 3.0, Intel, 3 feb. 2004.

[2] «RISC-V GNU Compiler Toolchain,» RISC-V Collaboration. (12 mar. 2015), indirizzo: `https://github.com/riscv-collab/riscv-gnu-toolchain`.

[3] *SiFive E31 Core Complex Manual*, ver. v1p2, SiFive, 11 ott. 2017.

[4] «QEMU virt board,» Software Freedom Conservancy. (12 ott. 2021), indirizzo: `https://www.qemu.org/docs/master/system/riscv/virt.html`.

[5] *RISC-V Advanced Core Local Interruptor Specification*, ver. 1.0-rc4, RISC-V Platform Specification Task Group, 10 gen. 2022, Draft.

[6] J. Hauser, *The RISC-V Advanced Interrupt Architecture*, ver. 1.0, RISC-V International, 30 giu. 2023.

[7] M. S. Tsirkin e C. Huck, *Virtual I/O Device (VIRTIO)*, ver. 1.3, Virtual I/O Device (VIRTIO) TC, 6 ott. 2023.

[8] *The RISC-V Instruction Set Manual: Volume I*, ver. 20240411, RISC-V International, 11 apr. 2024.

[9] *The RISC-V Instruction Set Manual: Volume II*, ver. 20240411, RISC-V International, 11 apr. 2024.