

UNIVERSITA' DI PISA

DIPARTIMENTO DI INGEGNERIA  
DELL'INFORMAZIONE

Tesi di Laurea Triennale in Ingegneria Informatica

# Migrazione del nucleo di un calcolatore da architettura x86 ad architettura RISC-V

Candidato  
**Andrea Bedini**

Relatore  
**Prof. Giuseppe Lettieri**

---

Anno Accademico 2022/2023



# Indice

1. Introduzione all'architettura
2. Toolchain di sviluppo
3. Organizzazione cartelle del progetto
4. Componenti implementate del sistema
  - 4.1. **Bootloader** ed inizializzazione **VGA**
  - 4.2. **Makefile** e **Linker**
    - 4.2.1. Supporto C++
    - 4.2.2. Cartelle dedicate header e oggetti
    - 4.2.3. Suite di Test
  - 4.3. **libCE**
    - 4.3.1. Traduzione, modifica e/o rimozione file C++ e Assembler
    - 4.3.2. Input tastiera tramite UART
    - 4.3.3. Implementazione VGA di Sistema
  - 4.4. **Sistema**
    - 4.4.1. Implementazione Salva/Carica stato
    - 4.4.2. Implementazione Paginazione e Memoria Virtuale
    - 4.4.3. Implementazione iniziale di Timer e Interruzioni
    - 4.4.4. Pulizia e traduzione iniziale file Sistema.cpp e Sistema.S
5. Problematiche riscontrate e loro risoluzione
6. Conclusioni e suggerimenti per la continuazione del progetto
7. Ringraziamenti

# 1. Introduzione all'architettura

**RISC-V** è un'architettura, come indica il nome, **RISC**(Reduced InStruction Set), ovvero la cui ISA possiede un numero limitato di istruzioni rispetto a standard più "classici" come x86 o ARM. Originariamente ideata nel 2010 nei laboratori dell'Università di Berkeley, California, questa architettura ha rapidamente preso piede nel mondo embedded e IoT grazie al fatto che è **Open Source**, perciò priva di onerosi costi di licenza.

Alcune delle caratteristiche principali e peculiarità di **RISC-V** sono:

- Avere un ridotto numero di istruzioni. Molte di queste istruzioni possiedono poi alias, detti "pseudoistruzioni", per comodità d'uso del programmatore (eg. "ret" -> "jr ra");
- Evitare il più possibile l'accesso in memoria, in quanto lento. A questo fine, l'architettura RISC-V prevede un grande numero di registri di controllo e stato, coi quali si può interagire solo grazie alle dedicate istruzioni **csrr** e **csrw**;
- Un sistema di paginazione e memoria virtuale simile a x86, il quale però supporta diversi tipi di indirizzamenti ed un controllo granulare dei permessi RWX (Read Write Execute);
- Assenza di Spazio I/O in favore di Memory Mapped I/O. Questa feature comporta anche il fatto che non tutta la memoria fisica sia accessibile/scrivibile liberamente dal kernel, in quanto diverse aree di memoria sono assegnate ad un differente compito;

Questa tesi punta a migrare il Nucleo didattico, utilizzato nel corso di Calcolatori Elettronici, dalla sua originaria architettura x86 all'architettura RISC-V. Una migrazione completa richiederebbe una quantità di lavoro ingente, ben oltre quanto una singola persona possa svolgere. L'obiettivo è perciò implementare gradualmente le meccaniche utilizzate dal Nucleo, in modo tale che in futuro altri studenti possano continuare il lavoro.

La migrazione del Nucleo si appoggia sul lavoro preliminare effettuato dal mio collega Edoardo Geraci, il quale ha realizzato un piccolo bootloader per sistemi RISC-V, assieme all'implementazione della periferica UART in sola lettura e della modalità testuale per la periferica VGA.

## 2. Toolchain di sviluppo

Per poter compilare il codice del Nucleo è necessario installare una toolchain per RISC-V. A seconda della distro linux attualmente in uso, questo pacchetto può essere trovato in un package manager (apt, aur, etc...) sotto uno dei seguenti nomi:

- riscv64-unknown-elf-gcc
- riscv-gnu-toolchain-bin
- **gcc-riscv64-unknown-elf**

Su Ubuntu, ad esempio, la toolchain è installabile tramite il comando

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

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

Il codice prodotto dalla toolchain può essere debuggato tramite l'apposita versione di gdb, chiamata **riscv64-unknown-elf-gdb**. Questa versione di gdb si comporta esattamente come la versione per x86, ed è perciò compatibile con estensioni come **gef**. Il debugger potrà perciò essere lanciato, dopo l'avvio di qemu, tramite il comando

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

Per utilizzare la toolchain è inoltre necessario impostare correttamente le variabili di sistema, per rendere visibili i binari alla shell. Il percorso da impostare, in default, risulta in genere essere

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

Una volta terminata l'installazione degli strumenti necessari, è possibile compilare il codice tramite il comando **make**. La definizione delle regole per questo comando è contenuta nel **makefile** presente nella cartella root del progetto. I comandi disponibili per make 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 realizzati in questa tesi
- **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 /obj/ generati durante la compilazione. Questo comando si rende necessario in quanto non sempre make riesce a determinare correttamente se un file necessita di essere ricompilato o meno.

### 3. Organizzazione cartelle del progetto

<b>Cartella</b>	<b>Descrizione</b>
/doc	Contiene file pdf di relazioni e slide inerenti al progetto
/include	Cartella dedicata contenente file header
/kernel	Contiene il bootloader del sistema
/libCE	Contiene la libreria utilizzata dal Nucleo, completamente migrata a RISC-V
/objs	Cartella dedicata a contenere i file oggetto generati durante la compilazione
/sistema	Contiene i file del Nucleo
/test	Contiene i file utilizzati nella testsuite. Ognuno rappresenta una feature implementata e testata in isolamento.

## 4. Componenti implementate del sistema

### 4.1 Bootloader e inizializzazione VGA

Il sistema comincia la sua esecuzione da un bootloader scritto in C e Risc-V Assembler. La specifica di Risc-V prevede che il sistema debba essere caricato a partire dall'indirizzo fisico **0x80000000**. Per effettuare questo, è stato modificato il file **linker.ld** per prevedere una sezione di memoria ram a quell'indirizzo:

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

L'esecuzione comincia dal file **entry.S**, dove viene impostato uno stack che verrà poi utilizzato dal codice scritto in C. Il flusso del sistema continua nel **start.S**, dove avvengono i seguenti passaggi:

- viene preparato il registro **mstatus** per passare alla Modalità Supervisor dopo aver eseguito l'istruzione **mret**;
- viene impostato il registro **mepc** per far saltare il codice alla funzione **boot\_main** dopo aver eseguito l'istruzione **mret**;
- viene azzerato il registro **satp** responsabile della paginazione;
- viene configurata la protezione della memoria per permettere alla Modalità Supervisor di accedere a tutta la memoria fisica tramite la scrittura nei registri **pmpaddr0** e **pmpcfg0**;
- viene eseguita l'istruzione **mret** che porta il processore in Modalità Supervisore ed inizia il codice della funzione **boot\_main**;



La funzione **boot\_main** effettua poi l'inizializzazione della periferica **VGA** in **textmode 80x25**, il cui buffer è presente all'indirizzo **0x50000000**.

Infine, il bootloader implementa una funzionalità di output debug tramite l'interfaccia UART, il cui buffer è presente all'indirizzo **0x10000000**.

Nell'implementazione offerta dal collega Geraci, la UART era disponibile solo in modalità output per mandare log di debug alla console.

La funzionalità dell'interfaccia in questa versione del kernel è stata espansa per permettere anche l'input dei caratteri, con lo scopo di sostituire l'interfaccia **PS/2** per tastiera, non presente su RISC-V.

## 4.2 Makefile e Linker

### 4.2.1 Supporto C++

Il bootloader, originariamente, supportava solo file sorgente in linguaggio C ed Assembler. Questa funzionalità è stata espansa per includere file scritti in linguaggio **C++**, assieme ai relativi switch necessari per la compilazione. I parametri switch passati al compilatore g++ sono stati derivati dopo aver osservato l'utilizzo che ne fa il **Makefile** della **libCE**, ora non più necessario.

### 4.2.2 Cartelle dedicate header e oggetti

Con l'aumento della complessità dell'organizzazione del codice, è sorta la necessità di effettuare un'opera di pulizia. A questo fine, il **Makefile** è stato modificato per organizzare file header e oggetto nelle relative cartelle. Per quanto riguarda i file **header**, questo processo è piuttosto semplice. Basta infatti compilare il codice con lo switch **-I<nome\_cartella>**, che in questo caso diventa **-linclude**.

Per ottenere una cartella unica per i file **oggetto**, vista la provenienza eterogenea del codice sorgente (differenti cartelle e differenti linguaggi), è stato invece necessario sfruttare effettivamente una tecnica detta **Implicit Build Rules**: ognuna di queste regole può definire il comportamento del compilatore per ogni differente estensione di file, ma deve essere specificata nuovamente per ogni cartella in cui si trova il codice.

```
#####
# Implicit build rules to gather all .o files in the objs directory
$(ODIR)/%.o: $K/%.c
    $(CC) -c $(CFLAGS) $< -o $@

$(ODIR)/%.o: $K/%.cpp
    $(CXX) -c $(CXXFLAGS) $< -o $@

$(ODIR)/%.o: $K/%.s
    $(AS) $(ASFLAGS) $< -o $@
. . .
```

### 4.2.3 Suite di Test

Al fine di poter testare il codice del Nucleo, è stato necessario implementare una suite di test. Il codice può essere compilato in modalità di test tramite il comando **make test**. I file sorgente per la suite di test sono contenuti nella cartella **/test/**, e sono chiamati secondo lo schema

**test\_<nome\_feature>\_<nome\_estensione>**. Ogni componente viene implementata e testata singolarmente, ed è composta da uno o due file sorgente, in base al fatto se sia necessario effettuare operazioni a basso livello in Assembler. I file per la suite di test vengono definiti e raccolti nel Makefile nella variabile **TESTOBS**. Oltre a questa variabile, sono presenti anche le variabili **OBS** e **PRODOBS**, le quali raccolgono rispettivamente i file oggetto in comune a tutte le versioni, e i file dedicati espressamente alle versioni non test del sistema. Con questa organizzazione diventa quindi possibile e semplice compilare codice destinato a scopi differenti.

```
_OBS = \  
    entry.o \  
    start.o \  
    machine_interrupts.o \  
    vga.o \  
    pci.o \  
    ctors.o \  
    uart.o \  
  
OBS = $(patsubst %, $(ODIR)/%, $(_OBS))  
  
_PRODOBS = \  
    boot_main.o \  
  
PRODOBS = $(patsubst %, $(ODIR)/%, $(_PRODOBS))  
  
_TESTOBS = \  
    test_stato_c.o \  
    test_stato_asm.o \  
    test_main.o \  
    test_ctors_asm.o \  
    test_keyboard_c.o \  
    test_traps_asm.o \  
    test_traps_c.o \  
    test_paginazione_c.o \  
  
TESTOBS = $(patsubst %, $(ODIR)/%, $(_TESTOBS))
```

Infine, il codice delle singole componenti viene lanciato dal file **test\_main.c**, il quale si occupa di provvedere un'implementazione alternativa della funzione **boot\_main**, a cui salta il bootloader al termine delle sue operazioni.

```
int boot_main(){
    pci_init();
    boot_printf("\n\nStarting tests...\n\r");
    boot_printf("Starting salva/carica_stato test.\n\r");
    test_stato_c();
    boot_printf("Salva/carica_stato test done.\n\r");
    boot_printf("Starting keyboard test\n\r");
    test_keyboard_c();
    boot_printf("Keyboard test done\n\r");
    boot_printf("Starting paging test\n\r");
    test_paginazione_c();
    boot_printf("Paging test done\n\r");
    boot_printf("All tests done.\n\r");
    return 0;
}
```

## 4.3 libCE

### 4.3.1 Traduzione, modifica e/o rimozione file C++ e Assembler

Viste le vaste differenze architetturali, per poter ripristinare il funzionamento della libreria libCE, sulla quale si appoggiano molti dei meccanismi del sistema, è stato necessario svolgere un'estensiva operazione di valutazione e pulizia file.

Innanzitutto, è stato necessario categorizzare i file in base ad uno dei seguenti criteri:

- Non più utile, e perciò eliminabile
- Logicamente necessario, ma non più funzionante
- Funzionante in maniera parziale, e perciò da adattare
- Completamente funzionante

Nella prima categoria dei **File non più utili** rientrano alcuni dei meccanismi non supportati in quella forma dall'architettura. Tra questi sono presenti i file per l'**APIC** (In RISC-V le interruzioni sono principalmente gestite via software, altro su questo al **Cap 4.4.3**), la **Memoria secondaria**, o Hard Disk (L'architettura RISC-V utilizza un complesso protocollo per dispositivi a blocco detto **VIRTIO**, la cui implementazione esula da quanto possibile in una tesi triennale), e molte delle funzioni inerenti alla **Tastiera** (Anche questo dispositivo utilizza **VIRTIO**, tuttavia il suo funzionamento è stato ripristinato parzialmente in maniera alternativa, come si può leggere subito dopo al **Cap 4.3.2**).

Nella seconda categoria, contenente i **File logicamente necessari, ma non funzionanti**, cadono tutti i file Assembler presenti nella **libCE**. Contenuti nella sottocartella **/libCE/as/**, questi file sono stati completamente riscritti ove necessario, oppure eliminati. Questo trattamento si è reso necessario viste le differenze di architettura e la diversa sintassi del linguaggio Assembler. I file attualmente presenti nella cartella sono:

- invalida\_TLB.s
- satp\_read\_asm.s
- satp\_write\_asm.s
- stval\_read.s

La terza categoria, in cui sono presenti i **File parzialmente funzionanti**, raccoglie gran parte dei file misti scritti in C++ all'interno della libCE. Questi file mostravano un funzionamento in gran parte compatibile col sistema, ma hanno subito alcune modifiche a causa delle differenze architetturali tra processori. Alcuni esempi notevoli sono la **VGA** (spiegata più nel dettaglio al **Cap 4.3.3**), il file responsabile per la paginazione vm.h (Il comportamento della paginazione e memoria virtuale è spiegato nel **Cap 4.4.2**), e **flog**. **flog** è il sistema di logging su console utilizzato dal Nucleo. Il funzionamento di questo modulo è stato ripristinato adattando il codice al meccanismo **UART**.

Infine, nella quarta ed ultima categoria dei file **Completamente Funzionanti**, rientrano alcuni file che, sorprendentemente, hanno mantenuto intatto il loro funzionamento originale. Tra questi troviamo, ad esempio, gli iteratori per la tabella delle pagine.

### 4.3.2 Input tastiera tramite UART

Vista l'assenza dell'interfaccia per tastiera **PS/2**, è stato necessario eliminare quasi del tutto il codice originariamente presente inerente alla **Tastiera**.

L'architettura **RISC-V** prevede l'utilizzo del protocollo **VIRTIO** per comunicare con una tastiera. Tuttavia, come già menzionato, questo protocollo è piuttosto complesso e poco documentato, e richiederebbe perciò il lavoro di un'intera tesi triennale per essere correttamente implementato. Per non lasciare il Sistema sprovvisto di una componente fondamentale, è stato fatto il compromesso di espandere significativamente le funzionalità **UART** per permettere l'input di caratteri.

Questo va a sopperire quasi completamente alle necessità di una tastiera, anche se presenta qualche **svantaggio**:

- Input possibile solo di caratteri (eg. tasti speciali come ESC e CTRL non possono essere riconosciuti);
- Input possibile solo quando la finestra di log del sistema è in focus;

```
char char_read()
{
    natb c;

    do {
        c = READ_UART_REG(UART_LSR);
    } while (!(c & 0x01));
    c = READ_UART();

    return c;
}
```

### 4.3.3 Implementazione VGA di Sistema

Il Nucleo di Calcolatori Elettronici aveva già un'implementazione per utilizzare l'interfaccia VGA. Tuttavia, questo codice non veniva aggiornato da tempo, e si è reso necessario correggere alcuni comportamenti scorretti. Dopo aver individuato e risolto i due problemi riscontrati, è stato possibile ripristinare il funzionamento dell'interfaccia, andando a far operare il codice sulla stessa area di memoria utilizzata dall'implementazione VGA del bootloader.

```
namespace vid {  
  
volatile natw* video = (natw*)(0x50000000 | (0xb8000 - 0xa0000));  
  
}
```



## 4.4 Sistema

### 4.4.1 Implementazione Salva/Carica stato

Uno dei componenti fondamentali del sistema è l'abilità di poter salvare lo stato attuale di un processo in esecuzione, per poi poterla proseguire in un secondo momento. Questa funzionalità è fondamentale in un sistema multiprocesso e dotato di interruzioni.

Al momento di un cambio processo, o dell'arrivo di un'interruzione, lo stato attuale di esecuzione viene salvato in una struttura detta **des\_proc**. Il sistema espone una variabile detta **esecuzione**, che punta al **des\_proc** del processo attualmente attivo. Tra le informazioni da salvare, si può trovare:

- l'**id** del processo;
- il **livello** del processo;
- la livello di **precedenza** del processo;
- il **punt\_nucleo** che punta alla Pila di Sistema;
- il **contesto** del processo, ovvero l'insieme di tutti i registri generali del processore, che nell'architettura RISC-V risultano essere 31;
- il contenuto del registro **epc** (Exception Program Counter), ovvero l'indirizzo dell'istruzione successiva a quella che ha causato l'interruzione;
- il contenuto del registro **satp** (Supervisor Address Translation and Protection), il quale mantiene l'indirizzo della Tabella delle Pagine attualmente in uso (simile al registro **cr3** di x86, ma configurato in maniera differente, vedi **Cap 4.4.2**);

L'implementazione attuale permette il salvataggio e il caricamento dello stato di un processo, tuttavia non esegue alcune operazioni, tra cui:

- il cambio di **Tabella delle Pagine**, in quanto la parte Utente del sistema non è ancora implementata;
- il salvataggio e ripristino di **epc**, in quanto le interruzioni non sono ancora propriamente implementate;

Per i due meccanismi sopra elencati è stato lasciato uno "scheletro", incluso di commenti, nel caso in cui qualcuno prosegua il lavoro sul Nucleo.

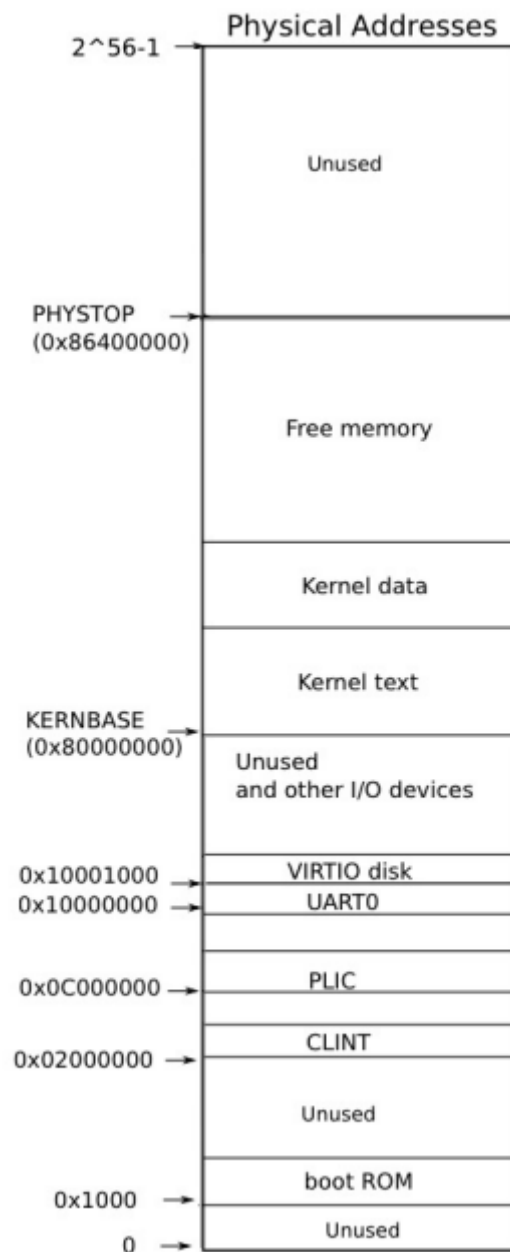
La funzionalità di salvataggio e ripristino dello stato necessita che, prima di effettuare il salvataggio, il registro RA venga salvato sullo stack. Questo è necessario per eseguire l'istruzione **ret** della **carica\_stato**, in quanto l'architettura RISC-V necessita di avere l'indirizzo di salto in un registro, che non potrebbe quindi essere ripristinato durante l'esecuzione della **carica\_stato**.

Questa limitazione potrà essere rimossa una volta implementato il meccanismo delle interruzioni, dato che il ritorno all'esecuzione ordinaria del codice potrà essere effettuato caricando il registro **sepc** con l'**epc** salvato, ed eseguendo l'istruzione **sret**.

```
test_stato_asm:
    addi sp,sp,-8
    sd ra,0(sp)
    call salva_stato
    call salva_success
    call carica_stato
    call carica_success
    ld ra,0(sp)
    addi sp,sp,8
    ret
```

## 4.4.2 Implementazione Paginazione e Memoria Virtuale

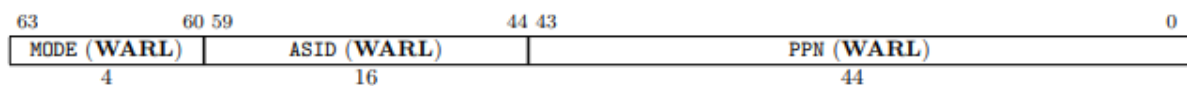
L'architettura RISC-V predispone regole abbastanza rigide riguardo il layout della memoria fisica. Dato che il sistema utilizza **mmio** (Memory Mapped I/O), certi intervalli di indirizzi fisici sono assegnati a scopi specifici. Di seguito un'immagine che sintetizza la distribuzione "standard" della memoria:



Come si può notare dall'immagine, RISC-V opera su un'architettura a 56 bit di indirizzo fisico. Di questi, il nostro sistema ne sfrutta come memoria RAM una sezione che parte dall'indirizzo **0x80000000** (vedi **Cap 4.1**).

Per quanto riguarda gli **Indirizzi Virtuali**, **RISC-V** supporta diverse modalità, in base al numero di bit che compongono l'indirizzo. La modalità utilizzata in questo è chiamata **Sv48**, ovvero imposta il sistema a gestire indirizzi virtuali su 48 bit. Questa scelta è stata fatta per essere compatibile con l'architettura x86-64 per cui era stato originariamente scritto il Nucleo, dato che anch'essa utilizza indirizzi a 48 bit.

La modalità di indirizzamento deve essere impostata tramite il registro **satp**, il quale contiene anche l'indice della pagina in cui si trova la radice della **Tabella delle Pagine**, necessaria per poter effettuare la traduzione da indirizzi fisici a virtuali:



RV64		
Value	Name	Description
0	Bare	No translation or protection.
1-7	—	<i>Reserved</i>
8	Sv39	Page-based 39-bit virtual addressing.
9	Sv48	Page-based 48-bit virtual addressing.
10	Sv57	<i>Reserved for page-based 57-bit virtual addressing.</i>
11	Sv64	<i>Reserved for page-based 64-bit virtual addressing.</i>
12-15	—	<i>Reserved</i>

Il campo **MODE** contiene i 4 bit che indicano il tipo di indirizzamento, mentre il campo **PPN** indica il numero di pagina contenente la radice della **Tabella delle Pagine**. La dimensione di ogni pagina su **RISC-V** è di **4KiB**, perciò basta usare 44 dei 56 bit di indirizzo fisico.

La **Tabella delle Pagine** di **RISC-V**, come in x86, è costituita da più livelli di traduzione, con un massimo di 3 per **Sv39** e 4 per **Sv48**. Ogni entrata della tabella può essere una foglia, ovvero contenere la traduzione dell'indirizzo fisico, oppure puntare alla pagina contenente il livello successivo della tabella.

Il formato di ogni entrata della tabella, per **Sv48**, è il seguente:

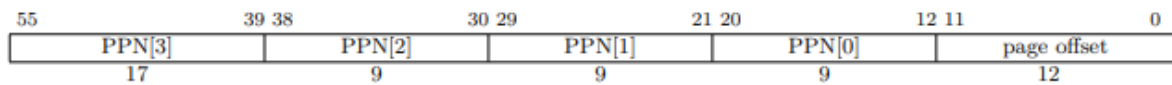


Figure 4.20: Sv48 physical address.

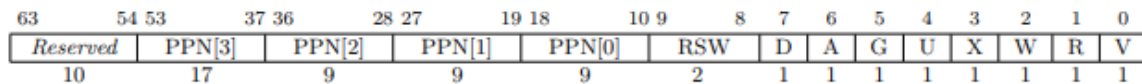


Figure 4.21: Sv48 page table entry.

I primi 8 bit sono flag che indicano le proprietà di ogni entry della tabella:

- **V**: bit di Validità. Se azzerato, l'entrata è da considerarsi invalida;
- **R**: bit di Lettura. Se azzerato, la pagina non può essere letta;
- **W**: bit di Scrittura. Se azzerato, la pagina non può essere scritta;
- **X**: bit di Esecuzione. Se azzerato, il codice presente nella pagina non può essere eseguito;
- **U**: bit Utente. Se azzerato, il processore non può avere accesso alla pagina se è correntemente in modalità Utente;
- **G**: bit Globale. Se settato, la pagina non viene invalidata a cambio di processo/invalidazione TLB;
- **A**: bit Accesso. Settato quando la entry viene percorsa dalla MMU;
- **D**: bit Dirty. Settato quando avviene una scrittura nella pagina puntata;

Una entry viene considerata **non foglia** quando possiede i 3 bit **RWX** a 0.

Altrimenti, la traduzione si interrompe in quel punto. Questo può essere utile per mappare intervalli contigui di grandi dimensioni, funzionamento analogo al **bit PS** presente nelle entry di x86.

Dato che il numero di pagina fisica parte dal bit 10, per impostare correttamente una entry è necessario azzerare tramite una maschera i 12 bit più bassi dell'indirizzo, e poi shiftare il tutto di due posti a destra.

Detto ciò, l'implementazione attuale del sistema va a sfruttare quella già presente nella libreria **libCE** (file **vm.h**), modificando opportunamente dove richiesto in base alle specifiche presentate sopra. Il risultato, testato in **/test/test\_paginazione\_c.cpp**, permette quindi di generare la **traduzione identità** necessaria al funzionamento del sistema, per poi caricarla nel registro **satp** in maniera opportuna.

Visto il peculiare layout di memoria di **RISC-V**, è necessario prestare attenzione al range mappato dalla traduzione, che dovrà andare a ricalcare solo quelli permessi dalla **ISA**:

```
// Mappa UART
if(map(root_tab, UART0, UART0+DIM_PAGINA, BIT_X | BIT_W | BIT_R,
identity_map) != (UART0+DIM_PAGINA)){
    return false;
}
// Mappa VIRTIO (al momento inutilizzato)
#define VIRTIO0 0x10001000L
if(map(root_tab, VIRTIO0, VIRTIO0+DIM_PAGINA, BIT_X | BIT_W | BIT_R,
identity_map) != (VIRTIO0+DIM_PAGINA)){
    return false;
}
// Mappa PLIC (al momento inutilizzato)
#define PLIC 0xc000000L
if(map(root_tab, PLIC, PLIC+0x400000, BIT_X | BIT_W | BIT_R, identity_map)
!= (PLIC+0x400000)){
    return false;
}
// Mappa VGA
#define VGA_BASE 0x3000000L
#define FRAMEBUFFER_VGA (0x50000000 | (0xb8000 - 0xa0000))
if(map(root_tab, VGA_BASE, VGA_BASE+DIM_PAGINA, BIT_X | BIT_W | BIT_R,
identity_map) != (VGA_BASE+DIM_PAGINA)){
    return false;
}
if(map(root_tab, FRAMEBUFFER_VGA, FRAMEBUFFER_VGA+DIM_PAGINA, BIT_X | BIT_W
| BIT_R, identity_map) != (FRAMEBUFFER_VGA+DIM_PAGINA)){
    return false;
}

// Mappiamo il kernel con tabelle di livello 2
if (map(root_tab, KERNBASE, KERNBASE+MEM_TOT, BIT_X | BIT_W | BIT_R,
identity_map, 2) != KERNBASE+MEM_TOT)
    return false;

boot_printf("Creata finestra sulla memoria centrale: [%p, %p)\n",
DIM_PAGINA, KERNBASE+MEM_TOT);

return true;
```

```

extern "C" void test_paginazione_c(){
    // iizializziamo la parte M2
    init_frame();

    // creiamo le parti condivise della memoria virtuale di tutti i processi
    // le parti sis/priv e usr/priv verranno create da crea_processo()
    // ogni volta che si attiva un nuovo processo
    paddr root_tab = alloca_tab();
    if (!root_tab)
        boot_printf("Errore allocazione tabella root");
    // finestra di memoria, che corrisponde alla parte sis/cond
    if(!crea_finestra_FM(root_tab))
        boot_printf("Errore creazione finestra FM\n");

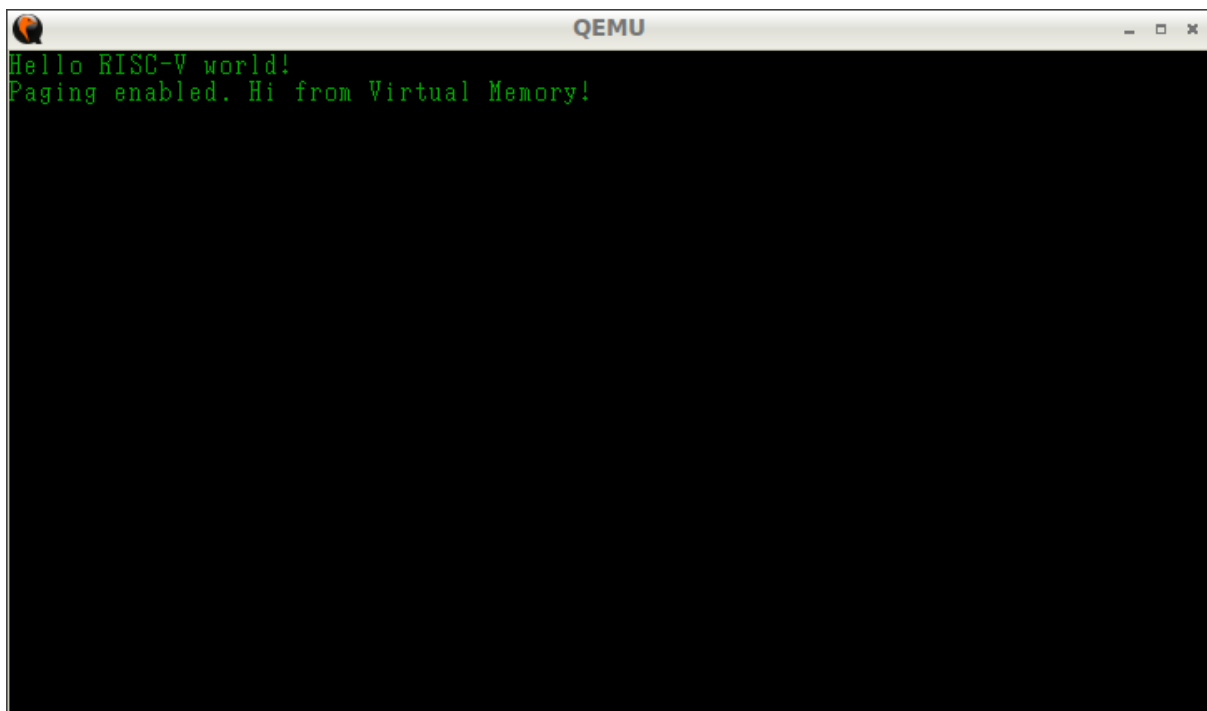
    // Attivazione paginazione
    writeSATP(root_tab);

    // Successo!
    printf("\nPaging enabled. Hi from Virtual Memory!\n\r");

    flog(LOG_INFO, "Hello from flog");
}

```

Il risultato del codice sopra riportato è che l'esecuzione prosegue dopo aver settato il registro **satp**, segno che la traduzione identità è stata correttamente generata, e rispetta gli indirizzi necessari sia alle istruzioni che al buffer VGA:



```

QEMU
Hello RISC-V world!
Paging enabled. Hi from Virtual Memory!

```

### 4.4.3 Implementazione iniziale di Timer e Interruzioni

Diversamente da quanto accade nell'architettura x86, che risolve le interruzioni via hardware tramite il dispositivo APIC, RISC-V utilizza una soluzione principalmente software.

Ogniqualvolta viene generata un'interruzione o un'eccezione, il processore interrompe l'esecuzione e salta all'indirizzo presente nel registro **stvec** o **mtvec** (questo comportamento è configurabile tramite i registri **medeleg** e **mideleg**).

Visto che il registro è unico per tutte le interruzioni, sarà poi compito del software gestire in maniera appropriata la situazione. La decisione avviene solitamente tramite la lettura del registro **scause**, che contiene il numero associato alla causa dell'interruzione. In maniera opzionale, alcuni dati potrebbero essere stati lasciati anche nel registro **stval**.

Nella versione attuale è presente uno “scheletro” di gestore delle interruzioni, nel caso qualche studente in futuro voglia proseguire la migrazione del Nucleo didattico.

E' tuttavia presente e pienamente funzionante l'implementazione del **timer** di Sistema, utilizzato per la pre-emption dei processi. Questa interruzione è l'unica configurata per sfruttare il registro **mtvec**, vista l'importanza dell'accuratezza degli intervalli temporali. Al termine dell'esecuzione, il gestore del timer alza il bit appropriato nel registro **sip**, per generare un'interruzione a livello Supervisore, che verrà poi gestita in maniera regolare.

```
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)

    # 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

```

Il timer viene inizialmente configurato ed abilitato nel bootloader in start.S:

```

# 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 machine mode interrupts
li t0, 8
csrs mstatus, t0

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

mret

```

#### 4.4.4 Pulizia e traduzione iniziale file Sistema.cpp e Sistema.S

Il codice attuale presenta una prima pulizia e traduzione dei file del Nucleo, chiamati sistema.cpp e sistema.s. Questo lavoro è parziale, visto l'immenso sforzo che è necessario per portare il sistema in uno stato completamente funzionante. Le varie componenti di sistema sono state implementate in isolamento nella Suite di Test, ma molte di queste necessitano di lavoro per poter essere integrate nel resto del sistema.

Il lavoro preliminare qua eseguito è lasciato, nel caso qualche altro studente intenda proseguire la migrazione del Nucleo, nella branch **wip** della repository in cui è ospitato questo progetto.

## 5. Problematiche riscontrate e loro risoluzione

Durante lo sviluppo di questo progetto sono sorte alcune problematiche. La principale di questa è comparsa dopo aver integrato la compilazione di file C++ nel **Makefile**. Da questo momento in poi, gli eseguibili hanno iniziato a produrre codice con indirizzi errati. Dopo attente ed esaustive ricerche, la causa sembra essere duplice: il registro **gp**, utilizzato come base per i salti globali, ed una particolare riga nel file del **linker**.

Il primo problema è stato risolto inizializzando il registro **gp** appena possibile, piazzando il seguente codice nel primo file eseguito all'avvio, **entry.S**:

```
# This is needed in order to produce a working executable
# alongside the changes made in the linker file kernel.ld
.option push
.option norelax
la gp, __global_pointer$
```

Il secondo problema è invece più elusivo. Dopo svariate ricerche e tentativi falliti, la soluzione sembra quella di commentare una specifica riga all'interno del file **linker.ld**. Vista la natura inusuale della soluzione, potrebbe non essere il procedimento corretto, tuttavia permette la creazione di eseguibili funzionanti, senza apparenti ripercussioni di altro tipo:

```
/* . = DATA_SEGMENT_END (.); */
```

## 6. Conclusioni e suggerimenti per la continuazione del progetto

Vista la quantità di lavoro limitata da svolgere per una tesi triennale, non è stato possibile effettuare una migrazione completa del Nucleo didattico. Tuttavia, molte delle componenti sono presenti, funzionanti, e facilmente integrabili.

Nel caso in cui qualche studente voglia proseguire questo lavoro, questi sono i punti su cui dovrebbe andare a lavorare:

- Terminare implementazione del Gestore delle Interruzioni;
- Completare salva/carica stato con le informazioni di epc e satp;
- Studiare funzionamento di multiboot per implementare correttamente il caricamento del segmento Utente in memoria;
- Integrare il meccanismo delle interruzioni con quello della paginazione per ottenere la Paginazione su Domanda, che dovrebbe essere già presente nel Nucleo;
- Implementare il protocollo VIRTIO per ripristinare la funzionalità della Memoria Secondaria (ed eventualmente della Tastiera);

Auguro in ogni caso un buon lavoro e buona fortuna a chiunque si voglia cimentare in questa impresa, che io ritengo estremamente complessa ma allo stesso tempo altamente formativa.

## 7. Ringraziamenti

Questa tesi è la conclusione di un percorso che è stato, a volte, arduo e accuminato. Ho voluto dare del mio meglio per essere fiero di me stesso, della strada che ho fatto, e della strada che percorrerò.

Ringrazio il Professor Giuseppe Lettieri, relatore di questa tesi, per avermi offerto la possibilità di cimentarmi in una tesi in cui ho provato genuino interesse personale, oltre che ad essere una persona molto “umana”, in un mondo accademico spesso formale e distaccato.

Ringrazio i miei amici, che mi hanno sempre supportato nel viaggio che ho percorso, e che mi hanno reso la persona che sono oggi.

Ringrazio tantissimo i miei genitori. Nonostante possiamo a volte battibeccare, sono i miei più grandi fan, e mi hanno spinto e supportato fin dal primo momento, sia nei giorni più belli che in quelli più brutti.

Ringrazio infine la mia ragazza, Tiziana “Lana” Domenichini, per essere stata la mia luce, anche nei momenti più bui.