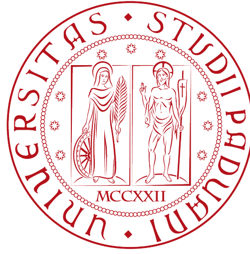


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale
in Ingegneria Informatica

Una Panoramica di RISC-V

Dettagli Tecnici e Impatto sull'Industria

Relatore:
Prof. Franceso Silvestri

Laureando:
Enrico Bolzonello
1216351

Anno Accademico 2021/2022

Abstract

Il mercato delle architetture fino ad ora è stato dominato da x86 nel settore computing e da Arm nel mercato embedded, tuttavia le esigenze stanno cambiando soprattutto grazie alla crescita di applicazioni avanzate come la computazione ad altre prestazioni, le comunicazioni 5g e le auto a guida autonoma. Per questo una nuova architettura aperta come RISC-V non può essere ignorata, testimoniata anche dall'interesse di Intel e Google.

La tesi si occupa di presentare la ISA RISC-V partendo dalla sua storia e dai suoi predecessori per poi descriverla tecnicamente, includendo esempi di codice. Successivamente analizzeremo brevemente lo stato attuale delle microarchitetture, tra cui gli acceleratori hardware, e il responso dell'industria, in particolare il caso Western Digital.

Indice

Elenco delle figure **v**

Elenco delle tabelle **vii**

Elenco dei codici **ix**

1 Introduzione **1**

1.1 Cos'è una ISA 1
1.2 Perché una ISA gratuita e open source 1
1.3 Perché RISC 2
1.4 Storia di RISC-V 2
1.5 Obiettivi di RISC-V 3
1.6 Panoramica 4
1.6.1 Uso e Gestione della Memoria 5
1.6.2 Modalità di Indirizzamento 7
1.6.3 Gestione delle Trap, Eccezioni e Interrupt 7
1.6.4 CSR 8
1.7 Genealogia 9

2 Modello di Consistenza della Memoria RVWMO **11**

2.1 Definizione 11
2.1.1 Preserved Program Order 13
2.1.2 Assiomi del Modello di Memoria 14
2.2 FENCE 16

3 ISA Non Privilegiata **17**

3.1 Base Intera RV32I 18
3.1.1 Registri 18
3.1.2 Istruzioni Base 19

3.2 Base RV32E 25
3.3 Base Intera RV64I 25
3.4 Base Intera RV128I 26

4 ISA Privilegiata 27
4.1 Livelli di Privilegio 27
4.2 ISA Livello Macchina 29
4.2.1 Istruzioni 29
4.2.2 Cosa succede al Reset 29
4.2.3 Attributi della Memoria Fisica (PMA) 30
4.2.4 Protezione della Memoria Fisica (PMP) 32
4.3 ISA Livello Supervisore 34
4.3.1 Memoria Virtuale 34
4.3.2 Istruzioni 37

5 Esempi di Codice 39
5.1 Linguaggio Assembly 39
5.1.1 Direttive dell'Assembler 40
5.1.2 Pseudo Istruzioni 41
5.2 Azzeramento di un Vettore 42
5.3 Quick Sort 46

6 Responso dell'Industria 51
6.1 Western Digital 51
6.1.1 Famiglia di processori SweRV 52
6.1.2 OmniXtend 53
6.2 Intel 54
6.2.1 Open Chiplet 54
6.3 Gli Approcci Europei e Cinesi 56

Conclusioni 59

Bibliografia 61

Elenco delle figure

2.1 Istruzione FENCE 16

3.1 Registri con il nome e lo scopo definiti dalla ABI (Application Binary Interface) 18
3.2 Formati base delle istruzioni più varianti 19
3.3 Opcode di RV32I 20
3.4 Composizione dei bit delle istruzioni load e store 24

4.1 Opcode di RV32I 32
4.2 Conversione dell'indirizzo virtuale in indirizzo fisico 35
4.3 Istruzione SFENCE.VMA 37

6.1 Schema del core SweRV EH1 53

Elenco delle tabelle

1.1 Instruction Set usate per il confronto 9

2.1 Sequenza di eventi per chiarire il concetto di modello di consistenza della memoria[9]. A e B hanno valori di default rispettivamente 1 e 2 11

3.1 Istruzioni computazionali di RV32I 21
3.2 Istruzioni di trasferimento di controllo di RV32I 22
3.3 Istruzioni aggiuntive di RV64I per gestire valori a 32 bit con registri a 64 25
3.4 Istruzioni aggiuntive di memoria di RV64I 26

4.1 Alcuni profili base 28
4.2 PMA AMO supportate 31
4.3 PMA di riservabilità supportati 31
4.4 Codifica di A nel registro d'indirizzo 33

5.1 Direttive dell'Assembler 40
5.2 Pseudoistruzioni 41

Elenco dei codici

2.1 esempio per spiegare CoRR. 13
3.1 Codice per caricare una variabile a distanza 0x234 dal pc 21
5.1 Codice C++ per la funzione di azzeramento di un array con indice 42
5.2 Codice C++ per la funzione di azzeramento di un array con puntatori 42
5.3 Codice assembly per azzerare un array con indice 42
5.4 Codice assembly per azzerare un array con puntatori 43
5.5 Risultato dell'object dump di azzera1 43
5.6 Risultato dell'objdump di azzera2 44
5.7 Implementazione in C++ del pseudocodice dell'algoritmo di Quick-sort ricorsivo[22] 46
5.8 Risultato dell'objdump di Quick Sort 47
5.9 Risultato dell'objdump della funzione Partition di Quick Sort . . . 49

Capitolo 1

Introduzione

RISC-V è un insieme di istruzioni macchina (ISA, Instruction Set Architecture) *open* sviluppato originariamente dall'Università di Berkley e basato su principi RISC (Reduced Instruction Set Computer).
Il nome è stato scelto per rappresentare il quinto design dell'Università di Berkley. È rappresentato come numero romano per un gioco di parole con "variations" e "vectors" dato che uno degli obiettivi espliciti del design è quello di supportare diverse aree ricerche nell'architettura. [1]

1.1 Cos'è una ISA

Una Instruction Set Architecture (ISA) definisce l'insieme di operazioni base che un computer deve supportare. Viene anche definita la funzione delle operazioni e come invocarle, oltre alla definizione dei tipi dei dati e dei registri.
Dato che definisce il comportamento del codice macchina, è indipendente dall'implementazione che è detta **micro architettura** e dunque garantisce la compatibilità tra diversi dispositivi.

1.2 Perché una ISA gratuita e open source

Si noti che una ISA è una specificazione di interfaccia, non un'implementazione. Ci sono tre tipi di implementazione:

- closed source
- open source su licenza
- libera

Le ISA proprietarie come ARM e x86 permettono solo i primi due tipi di implementazione, per permettere tutte e tre è necessaria una ISA open. Inoltre creerebbe un vero mercato libero della progettazione di processori che potrebbe portare a:

- **maggior innovazione** grazie alla competizione del mercato libero
- **design condivisi**, che garantirebbero una trasparenza maggiore e possibili tempi di commercializzazione ridotti
- **processori meno costosi**

1.3 Perché RISC

RISC è un tipo di architettura di microprocessore che utilizza un piccolo set di istruzioni e altamente ottimizzato[2]. Nonostante le molteplici implementazioni e approcci di questo paradigma, vi sono delle caratteristiche comuni [3]:

- **una istruzione per ciclo**, un ciclo è definito come il tempo per prendere due operandi dai registri, eseguire una operazione ALU e immagazzinare il risultato in un registro
- **operazioni registro-registro**, solo istruzioni LOAD e STORE accedono alla memoria
- **modalità di indirizzamento semplici**
- **formato delle istruzioni semplice**, la lunghezza delle istruzioni e le posizioni dei campi sono fissi

Essendo RISC-V una nuova architettura e dunque non avendo codice legacy, il costo dell'hardware aggiuntivo e il costo in energia di traduzione che comporterebbe una architettura CISC come x86 sono difficili da giustificare[4].

1.4 Storia di RISC-V

Il progetto RISC-V è iniziato nel Maggio 2010 come parte del progetto Parallel Computing Laboratory (Par Lab) all'Università di Barkleye diretto al prof. David Patterson. Il laboratorio era finanziato da *Intel* e *Microsoft*, oltre ad altre aziende e allo stato della California. Non era finanziato federalmente, ma ha ricevuto dei

fondi dal progetto POEM del *Darpa*, l'agenzia governativa statunitense per lo sviluppo di tecnologie per uso militare.

La prima pubblicazione fu la prima versione del manuale della ISA livello utente[1] il 13 Maggio 2011 e nello stesso anno venne ultimato il primo tape-out[1].
Nel 2015 venne creata la **RISC-V Foundation**, un'organizzazione no-profit per direzionare l'adozione iniziale della ISA. Nel Novembre 2018 venne annuciata una collaborazione con la Linux Foundation, che provvede supporto operazionale, tecnico e strategico.
Per sottolineare la propria neutralità, nel Marzo 2020 venne costituita la **RISC-V International Association** in Svizzera. [5]

1.5 Obiettivi di RISC-V

Gli obiettivi dichiarati nelle specifiche tecniche della ISA user-mode[1] sono i seguenti:

1. realizzare una ISA completamente aperta gratuitamente disponibile alle facoltà accademiche e all'industria
2. adatta ad un'implementazione hardware nativa diretta
3. indipendente da un particolare stile di micro architettura o tecnologia di implementazione
4. modulare, con una base più delle estensioni stardard opzionali
5. supporto per lo standard floating-point 2008 IEEE-754
6. supporto per estensioni e varianti personalizzate
7. spazio di indirizzamento 32 e 64 bit
8. supporto per implementazioni multi core o manycore
9. istruzioni a lunghezza variabile opzionali
10. completamente virtualizzabile
11. semplificazione di esperimenti con l'architettura privilegiata

[1] Il tapeout è la fase finale del ciclo di sviluppo dei micro processori

1.6 Panoramica

L'ISA RISC-V è una architettura load-store[2] con un ristretto numero di istruzioni, solo 49 di base, e un alto numero di registri. È definita come una base ISA obbligatoria per gestire gli interi più estensioni opzionali. La base è ristretta ad un minimo insieme di istruzioni e dunque è un conveniente scheletro su cui costruire ISA personalizzate.
In realtà RISC-V è una famiglia di ISA connesse, di cui 4 base a livello utente, trattate nel Capitolo 3, che sono:

1. **RV32I**, con XLEN, cioè lo spazio di indirizzamento, di 32 bit. Sufficiente per scopi educativi e adeguato per molti dispositivi embedded e client che richiedono basso consumo di energia e memoria
2. **RV64I**, con XLEN 64 bit. Richiesto per sistemi più complessi
3. **RV32E**, sottoinsieme di RV32I per dispositivi embedded
4. **RV128I**, con XLEN 128 bit. Potrebbe essere richiesto in futuro, anche se pochi dispositivi lo implementano attualmente

Tutte le basi usano complemento a due per la rappresentazione di valori interi con segno. La base per gli interi è denominata "I" e contiene istruzioni computazionali per interi, load e store di interi e istruzioni per il flusso di controllo
Il vantaggio principale di separare le ISA base è che ogni base può essere ottimizzato senza dover supportare tutte le operazioni richieste dalle altre basi. Lo svantaggio principale è che complica l'hardware che serve per emulare una base su un'altra (ad esempio emulare RV32I su RV64I), ma le basi sono abbastanza simili da poter supportare diverse versioni ad un costo relativamente basso.

Per quanto riguarda le estensioni, possono essere di tre tipi in base alla standardizzazione:

- **standard**, definite dalla RISC-V Foundation e non devono essere in conflitto con altre estensioni standard
- **reserved**, non definite ma salvate per usi futuri
- **non standard**, non definita dalla Foundation. Si dice **non-conforming** se usa delle codifiche di una estensione o base standard o reserved.

[2]Significa che solo le istruzioni load e store possono accedere alla memoria

Le estensioni standard sono le seguenti:

1. "M", aggiunge istruzioni per moltiplicazioni e divisioni di interi
2. istruzioni atomiche "A", istruzioni di lettura-scrittura-modifica atomiche[3]
3. virgola mobile a singola precisione "F", aggiunge registri, istruzioni, load e store a virgola mobile a singola precisione
4. virgola mobile a doppia precisione "D", espande i registri e aggiunge istruzioni, load e store a virgola mobile a doppia precisione
5. istruzioni compresse "C", istruzioni a 16 bit

Tutte queste estensioni e basi sono a livello utente, dunque non gestiscono l'hardware. Per questo scopo, c'è bisogno di una ISA privilegiata, tratta nel Capitolo 4. Separando la parte privilegiata dalla parte utente si sottolinea ancora la modularità di RISC-V, in quanto si può rimpiazzare la ISA privilegiata senza dover modificare la base utente.

1.6.1 Uso e Gestione della Memoria

La ISA base può supportare sia sistemi di memoria **little-endian** che **big-endian** per supportare tutte le aree di applicazione. Le istruzioni invece sono immagazzinate come una sequenza di pacchetti di 16 bit little-endian, indipendentemente dall'endianness della memoria.
Ogni pacchetto ha nei bit meno significativi i bit per la codifica della lunghezza in modo che la lunghezza di un'istruzione a lunghezza variabile sia stabilita velocemente e per evitare di dividere i campi opcode.

Per quanto riguarda le celle della memoria principale accessibili, ogni hart[4] RISC-V ha uno spazio indirizzabile di 2^{XLEN} byte circolare, con $XLEN$ lunghezza degli indirizzi che dipende dalla base scelta. Non c'è overflow, in quanto i calcoli dell'address vengono scalati in modo adeguato dividendo per il modulo 2^{XLEN}, sfruttando dunque la caratteristica circolare.

[3] Significa che non sono divisibili in risultati parziali ed eseguono nello stesso processo

[4] Un hart è la contrazione di hardware thread ed è l'astrazione di un ambiente con un insieme completo di registri ed esegue programmi indipendentemente dagli altri hart

Diversi intervalli di indirizzi di uno spazio di indirizzamento possono:

1. essere vacanti, non accessibili
2. contenere memoria principale
3. contenere uno o più dispositivi di I/O

L'ambiente di esecuzione decide quale parti degli spazi di indirizzamento non vacanti sono accessibili. Di solito ci si aspetta che qualche porzione sia specificato come memoria principale.

Ogni istruzione macchina può prevedere uno o più accessi in memoria, divisi in accessi **impliciti** ed **espliciti**. Per ogni istruzione eseguita, viene fatta una lettura implicita a memoria per ottenere l'istruzione, cioè il fetch. Alcune istruzioni hanno accessi espliciti alla memoria con load e store, che sono le uniche istruzioni che possono eseguire questa operazione.

Gli accessi alla memoria possono avvenire in ordini diversi ma sempre limitati dai modelli di coerenza della memoria. Il modello di coerenza di default è il RISC-V Weak Memory Ordering (RVWMO) trattato nel Capitolo 2, opzionalmente si può adottare il modello più restrittivo del Total Store Ordering, usato anche in x86. L'ambiente di esecuzione può aggiungere altri vincoli.

1.6.2 Modalità di Indirizzamento

Essendo un'architettura RISC e dunque avendo tutte le istruzioni a lunghezza fissa, non è possibile avere direttamente costanti o indirizzi superiori alla lunghezza assegnata al campo dell'istruzione. Per ovviare a questo problema si usano quelle che sono dette **modalità di indirizzamento**:

- **indirizzamento immediato**, l'operando è una costante nell'istruzione. Limitato ai bit assegnati all'operando
- **indirizzamento a registro**, l'operando è un registro
- **indirizzamento di base e spostamento**, l'operando è la somma tra il contenuto di un registro e una costante
- **indirizzamento relativo al PC**, somma del contenuto del Program Counter e una costante. Caso particolare dell'indirizzamento a registro, viene usato solo nelle istruzioni di salto dato che l'indirizzo di destinazione è molto probabile che sia vicino all'indirizzo del PC.

1.6.3 Gestione delle Trap, Eccezioni e Interrupt

Sia le eccezioni che gli interrupt sono eventi inaspettati che interrompono la normale esecuzione delle istruzioni. La differenza tra i due è che un'eccezione è un evento sincrono generato dalle istruzioni stesse, mentre un interrupt è un evento asincrono generato da agenti esterni.
Una Trap è l'operazione di trasferimento di controllo ad un trap handler e può essere causata sia da un'eccezione sia da un interrupt. Possono avere quattro effetti:

- **Trap contenuta**, visibile e gestita dal software
- **Trap richiesta**, è una richiesta di azione del software, non è imprevista come le altre. Un esempio sono le system call
- **Trap invisibile**, non visibile all'ambiente di esecuzione, dunque il software non sa della trap
- **Trap Fatale**, causa la terminazione dell'esecuzione

Nella ISA privilegiata 4 viene definito un interrupt aggiuntivo, il NMI (Non-Maskable Interrupt), usato solo per errori di hardware.

1.6.4 CSR

Oltre ai registri interi di numero e lunghezza variabile a seconda della base usata che verranno trattati nel Capitolo 3, RISC-V definisce nell'estensione Zicsr i registri di controllo e stato CSR che sono dei registri usati per memorizzare informazioni sulle istruzioni, oltre ad istruzioni specifiche per la gestione di essi.
Vengono definiti 4096 CSR con un indirizzo da 12 bit, e sono obbligatori per l'architettura privilegiata. I primi due bit dell'indirizzo dettano se il registro è di lettura,scrittura o entrambi; i successivi due indicano il livello più basso che può accedere al registro stesso.
La lista completa dei CSR disponibili nella seconda sezione del secondo capitolo del manuale dell'architettura privilegiata[6].

1.7 Genealogia

RISC-V si ispira ad architetture RISC del passato, infatti confrontando le istruzioni di RV32G (cioè la base I più le estensioni M, A e F) con i precedenti storici della tabella 1.1, scelti tra le principali architetture RISC proprietarie, oltre alle architetture RISC realizzate dalla UC Berkley in passato:

Year Published	Instruction Set Architecture	Year Published	Instruction Set Architecture
1964	CDC 6600	1992	DEC Alpha
1981	RISC I / RISC II	1992	MIPS III
1984	SOAR (RISC IIII)	1992	IBM PowerPC
1984	Intel i960	1992	Torrent T0
1985	IBM RP3	1994	MIPS IV
1987	ARMv2	1995	PA-RISC 2.0
1988	SPUR (RISC IV)	1997	Hitachi SH-4
1990	DLX	2002	ARMv6
1990	SPARCv8	2003	Cray X1

[7]

Tabella 1.1: Instruction Set usate per il confronto

Il risultato[7] è che tra le 122 istruzioni in RV32G:

- 6 sono uniche di RISC-V
- 98 delle 116 appaiono in almeno tre diversi instruction set

Vengono considerate precedenti se l'istruzione implementa lo stesso comportamento del corrispondente in RISC-V.

Capitolo 2

Modello di Consistenza della Memoria RVWMO

RISC-V usa un modello di consistenza della memoria[1] chiamato **RVWMO** (RISC-V Weak Memory Ordering), definito nel manuale della ISA livello utente[1]. Tramite questo modello, codice che esegue in un hart appare in ordine dalla prospettiva di altre istruzioni di memoria nello stesso hart, ma potrebbe non esserci lo stesso ordine dal punto di vista di un'altro hart, in quanto potrebbero esserci delle operazioni per ottimizzare le prestazioni, tra cui il riordinamento o la fusione delle istruzioni di memoria. Per la sincronizzazione esplicita si usa l'istruzione FENCE, presente nella ISA base.

2.1 Definizione

Per chiarire il concetto di modello di memoria, consideriamo il seguente esempio:

CPU 1	**CPU 2**
(1) A = 3	(3) x = B
(2) B = 4	(4) y = A

Tabella 2.1: Sequenza di eventi per chiarire il concetto di modello di consistenza della memoria[9]. A e B hanno valori di default rispettivamente 1 e 2

L'ordine di esecuzione dal punto di vista di un hart esterno può essere una delle

[1] Un modello di consistenza della memoria è una specificazione del comportamento consentito dai programmi multithread eseguiti con memoria condivisa[8], in questo caso un insieme di regole che specificano i valori che possono essere ritornati dai load

24 combinazioni delle istruzioni, ma i risultati[9] possono essere quattro in sistemi multiprocessore o con compilatori che ottimizzano il codice:

- $x = 2$ e $y = 1$
- $x = 2$ e $y = 3$
- $x = 4$ e $y = 1$
- $x = 4$ e $y = 3$

Come si può vedere, i risultati sono molto diversi tra loro e dunque è necessario specificare delle regole in modo da ridurre i risultati possibili. I comportamenti possibili di un programma concorrente sono specificati dal cosiddetto **modello di memoria debole**[10], di cui RVWMO ne è un esempio.
Rispetto ad un modello forte, come il TSO[2] di Intel, ha meno garanzie, ma anche una minore complessità a livello hardware ed è possibile ottenere maggiori performance grazie al maggior numero di riordinamenti possibili dai software di ottimizzazione.

Il modello di memoria di RISC-V preserva tre ordinamenti[1]:

- **Program Order**, cioè l'ordine delle operazioni di memoria[3] di un singolo hart
- **Global Memory Order**, cioè l'ordine delle operazioni di memoria di tutti gli hart
- **Preserved Program Order**, cioè il sottoinsieme del Program Order che deve essere sempre rispettato

[2]Total Store Ordering

[3]Le operazioni di memoria sono **load** e **store**: un load si dice completato quando il valore di ritorno è determinato, uno store si dice completato non quando esegue nella pipeline, ma quando il valore è propagato nella memoria globale

2.1.1 Preserved Program Order

L'ordine globale non rispetta tutte le regole di programma di tutti gli hart: il sottoinsieme del Program Order che viene rispettato dal Global Memory Order viene detto **Preserved Program Order**. Concettualmente, gli eventi ordinati in un hart dal Preserved Program Order devono apparire in ordine anche dalla prospettiva di altri hart o osservatori esterni.

L'operazione di memoria a precede l'operazione di memoria b nel Preserved Program Order se accedono entrambe alla memoria principale, a precede b nel Program Order e almeno una delle seguenti regole vale:

- **Ordini di Indirizzi Sovrapposti**, per esempio se b è uno store e a e b accedono ad indirizzi sovrapposti. Due load con lo stesso indirizzo richiedono il cosiddetto **CoRR** (Coherence for Read-Read pairs), cioè che un load non può ritornare un valore che è più vecchio di un valore ritornato da un load precedente nello stesso hart e che accede allo stesso indirizzo. Per esempio, nel seguente codice:

```
1  li a0, 4
2  xor t2, a0, a0
3  add s2, a0, t2
4  addi s3, a0, 1
5  lw a1, 0(s2)
6  lw a2, 0(s3)
```

Codice 2.1: esempio per spiegare CoRR.

  Mettere l'istruzione alla riga 6 prima di quella alla riga 5 nell'ordine di memoria globale violerebbe CoRR, perché la 5 ritornerebbe un valore più vecchio della 6.

- **Sincronizzazione Esplicita** tramite istruzioni di FENCE, annotazioni di acquire su a o annotazioni di release su b.

- **Dipendenze Sintattiche**, un'istruzione j con registro di origine r ha una dipendenza sintattica sull'istruzione i con registro di destinazione s se almeno una delle seguenti regole vale:

  – s è lo stesso di r e non ci sono istruzioni nel Program Order tra i e j con r come registro di destinazione

  - tra i e j è presente un'istruzione m tale che m ha una dipendenza sintattica su i, j ha una dipendenza sintattica su m e m ha una dipendenza tra il suoi registri di origine e destinazione.

  Altre definizioni di dipendenze sintattiche comprendono operazioni di memoria a e b generate rispettivamente dalle istruzioni i e j e sono definite come seguono:

  - b ha una **dipendenza sintattica di indirizzo** su a se j ha una dipendenza sintattica su i tramite il registro di origine r di j
  - b ha una **dipendenza sintattica di dati** su a se b è un'operazione di store e j ha una dipendenza sintattica su i tramite il registro di origine r di j
  - b ha una **dipendenza sintattica di controllo** su a se c'è un'istruzione m tra i e j take che m sia un branch o un salto indiretto e m ha una dipendenza sintattica su i

- **Dipendenze delle Pipeline**:

  1. supponendo un load b, tra a, operazione di memoria qualunque, e b è presente un'istruzione di store m tale che ha una dipendenza di indirizzo o di dati su a e b ritorna un valore scritto da m
  2. supponendo uno store b, tra a, operazione di memoria qualunque e b è presente un'istruzione qualsiasi m che ha una dipendenza di indirizzo su a

2.1.2 Assiomi del Modello di Memoria

Il modello RVWMO oltre ad avere un ordine di memoria globale che rispetta il Preserved Program Order deve anche rispettare i seguenti assiomi:

Assioma del Valore di Load

> "Ogni load ritorna il valore scritto dall'ultimo store nell'ordine di memoria globale tra gli store sia nell'ordine globale che nel program order che precede i load."

Da notare che l'assioma deve essere rispettato solo dal Global Order e dal Program Order, non dal Preserved Program Order. Il motivo è la presenza in molte

macchine dei cosiddetti **store buffer**: dato che uno store carica il valore nello store buffer prima che nella memoria princiale, il load può reperire il valore nello store buffer, dunque prima che lo store abbia concluso la propria esecuzione.

Assioma di Atomicità

> "Se r e w sono load e store accoppiati generati da LR e SC, s è uno store sul byte x e r ritorna un valore scritto da s, allora s deve precedere w e non ci possono essere altri store da altri hart sul byte x che seguono s e precedono w."

L'assioma praticamente proibisce agli store di altri hart di essere messo in mezzo ad una coppia LR SC nell'ordine globale, però permette store dallo stesso hart in mezzo ad una coppia LR SC.

Assioma di Avanzamento

> "Nessuna operazione di memoria può essere preceduta da infinite operazioni di memoria."

Proibisce cicli infiniti, dunque garantisce che i valori di uno store siano visibili ad altri hart in un tempo finito e i load siano in grado di leggere quei valori anch'esso in un tempo finito.

2.2 FENCE

Dato che RVWMO è un modello rilassato, con conseguenza che dal punto di vista di un thread esterno all'esecuzione l'ordine delle istruzioni di memoria può essere diverso, è necessaria una cosiddetta **barriera di memoria**[9] per forzare un ordine, senza il quale potrebbero presentarsi problemi con il concurrent computing. Una barriera di memoria assicura che le operazioni prima della barriera vengano eseguite prima delle operazioni dopo la barriera. Questo però non assicura che venga rispettato l'ordine originale, in quanto i due sottoinsiemi di istruzioni prima e dopo la barriera possono avere il loro ordine. In RISC-V la barriera di memoria viene definita sotto forma dell'istruzione FENCE.

31 28	27	26	25	24	23	22	21	20	19 15	14 12	11 7	6 0
fm	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
FM	predecessor				successor				0	FENCE	0	MISC-MEM

Figura 2.1: Istruzione FENCE

Il campo fm, che sono i primi quattro bit nella Figura 2.1, descrive la semantica di FENCE. Se $fm = 0000$ è una fence che ordina solo i load, se $fm = 1000$ è una FENCE.TSO che oltre ad ordinare i load ordina anche gli store. Altre codifiche sono riservate per uso futuro.
I campi dal bit 27 al bit 20 indicano che istruzioni ordinare: i campi da 27 a 24 indicano di ordinare le istruzioni prima della fence, i campi da 23 a 20 indicano di ordinare le istruzioni dopo la fence. I campi nominati con le lettere finali I,O,R e W indicano di ordinare rispettivamente le istruzioni di input e output dei dispositivi esterni e di lettura e scrittura della memoria.
Una FENCE senza parametri è mappato come `FENCE iorw, iorw` ,cioè su tutta la memoria e tutti i dispositivi I/O.

Capitolo 3

ISA Non Privilegiata

L'ISA non privilegiata definisce una serie di registri, oltre ad istruzioni varie. Attualmente sono definite quattro basi e una serie di estensioni. Le basi sono:

- RV32I
- RV32E
- RV64I
- RV128I

In questo capitolo vengono trattate solo le basi, definite nel manuale della ISA Non Privilegiata[1].

3.1 Base Intera RV32I

La base RV32I è stata progettata per essere sufficiente a supportare ambienti operativi moderni e per ridurre l'hardware necessario al minimo.
Comprende 40 istruzioni uniche.

3.1.1 Registri

I 32 registri interi di RV32I sono lunghi 32 bit (XLEN=32). I registri sono nominati con x seguiti dal numero. Il primo registro chiamato $x0$ è cablato con tutti i bit a 0. I registri $x1 - x31$ sono registri a scopo generale e tengono valori. L'ABI (Application Binary Interface) di RISC-V definisce la convenzione di chiamata come nella Figura 3.1.
L'ultimo registro è il program counter *pc* che immagazzina l'indirizzo dell'istruzione corrente.

Name	ABI Mnemonic	Calling Convention	Preserved across calls?
x0	zero	Zero	n/a
x1	ra	Return address	No
x2	sp	Stack pointer	Yes
x3	gp	Global pointer	n/a
x4	tp	Thread pointer	n/a
x5-x7	t0-t2	Temporary registers	No
x8-x9	s0-s1	Saved registers	Yes
x10-x17	a0-a7	Argument registers	No
x18-x27	s2-s11	Saved registers	Yes
x28-x31	t3-t6	Temporary registers	No

https://danielmangum.com/posts/risc-v-bytes-caller-callee-registers/

Figura 3.1: Registri con il nome e lo scopo definiti dalla ABI (Application Binary Interface)

Da notare che non ci sono registri dedicati per lo stack pointer o per subroutine return, questo perché è permesso usare ogni registro. Nonostante ciò vi sono delle convenzioni:

- per immagazzinare l'indirizzo di ritorno viene usato il registro $x1$ o in alternativa $x5$
- come stack pointer viene usato il registro $x2$

Sono stati scelti 32 registri per *evitare dimensioni intermedie* e perché *un numero maggiore di registri interi aiuta le prestazioni in codice ad altre prestazioni* [11].

3.1.2 Istruzioni Base

Le 47 istruzioni di RV32I sono lunghe 32 bit e sono allineate in memoria in little-endian. Per codificarle, ci sono 6 formati totali di cui 4 base (R/I/S/U) e 2 varianti (B/J) rispettivamente di S e U.

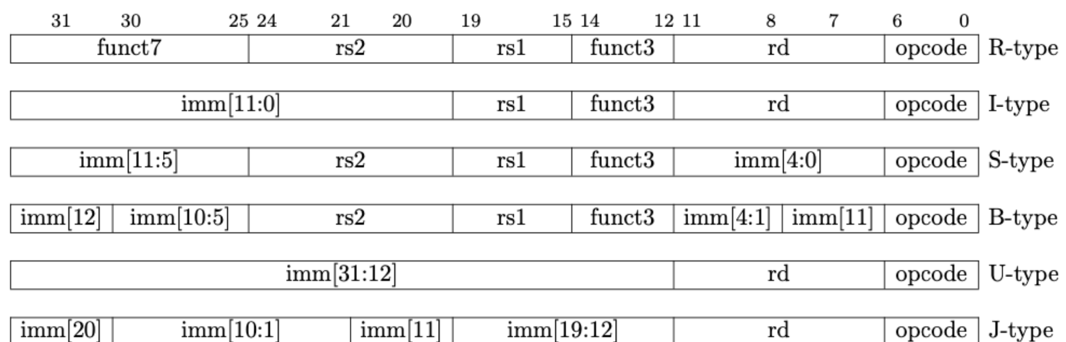


Figura 3.2: Formati base delle istruzioni più varianti

In tutti i formati base i registri di sorgente ($rs1$ e $rs2$) e di destinazione (rd) sono mantenuti nella stessa posizione per semplificare la decodifica.
I valori immediati (nella Figura 3.2 contrassegnati con imm) sono tutti con segno, anche se viene rappresentato un valore senza segno. Il bit di segno è posizionato nel 31esimo bit per permettere la sign extension[1] in parallelo con la decodifica dell'istruzione.

La variante B di S modifica il campo immediato lungo 12 bit, che viene usato per codificare l'offset del branch in multipli di 2. Il bit imm[12] è il bit di segno

[1]È l'operazione di aumentare il numero di bit di un numero binario mantenenedo il valore e il segno

mentre il bit imm[11] è posizionato alla fine del valore immediato ed è il bit più significativo togliendo il segno. La variante J di U modifica il campo immediato lungo 20 bit spostandoli a sinistra di 13 bit.

Ruotando i bit invece che moltiplicando il valore immediato per 2 tramite dei multiplexer si riduce il costo dei multiplexer per i valori immediati.

Gli opcode nella ISA base sono lunghi 7 bit, ma i due bit meno significativi sono impostati ad 1. L'immagine seguente riassume l'uso degli opcode per RV32I.

inst[4:2] inst[6:5]	000	001	010	011	100	101	110	111
00	Loads	*F Ext.*		Fences	Arithmetic	AUIPC	*RV64I*	
01	Stores	*F Ext.*		*A Ext.*	Arithmetic	LUI	*RV64I*	
10	*F Ext.*	*F Ext.*	*F Ext.*	*F Ext.*	*F Ext.*		*RV128I*	
11	Branches	JALR		JAL	System		*RV128I*	

[12]

Figura 3.3: Opcode di RV32I

Istruzioni Computazionali

Le istruzioni computazionali sono 21 in totale, riassunte nella tabella 3.1. Molte di loro operano su XLEN bit di valori mantenuti nei registri di interi. Vengono codificate con un formato I se è un'operazione tra registro e immediato o con un formato R se è un'operazione tra registri.

Le istruzioni ADD, SUB e ADDI ignorano gli overflow aritmetici. ADDI rd, rs1,0 viene usata per implementare la pseudoistruzione MV rd, rs1.
SLTU, SLTIU e SLTI (significano Set Less Than) mette il valore 1 nel registro rd se il registro $rs1$ è minore del valore del registro o del valore immediato, altrimenti mette il valore 0. $SLTIU\ rd, rs1, 1$ implementa la pseudoistruzione $SEQZ\ rd, rs$: imposta rd a 1 se $rs1$ è uguale a zero, altrimenti imposta rd a 0. $SLTU\ rd, x0, rs2$ implementa la pseudoistruzione $SNEZ\ rd, rs$ con lo stesso comportamento di SEQZ.
XORI $rd, rs1, -1$ implementa la pseudoistruzione NOT rd, rs.

L'istruzione LUI mette il valore immediato nei primi 20 bit del registro di destinazione e riempie i restanti 12 bit con tutti zeri. Viene usata per dichiarare costanti a 32 bit.

Istruzione	Risultato
add rd, rs1, rs2	somma di valori nei registri
addi rd, rs1, imm[11:0]	somma di un valore immediato e valore in un registro
sub rd, rs1, rs2	sottrazione di valori nei registri
sll rd, rs1, rs2	shift logico a sinistra di un valore in un registro
srl rd, rs1, rs2	shift logico a destra di un valore in un registro
slli rd, rs1, shamt[4:0]	shift logico a sinistra di un valore immediato
srli rd, rs1, shamt[4:0]	shift logico a destra di un valore immediato
sra rd, rs1, rs2	shift aritmetico a destra di un valore in un registro
srai rd, rs1, shamt[4:0]	shift aritmetico a destra di un valore immediato
and rd, rs1, rs2	and bit a bit con valori in un registro
or rd, rs1, rs2	or bit a bit con valori in un registro
xor rd, rs1, rs2	xor bit a bit con valori in un registro
andi rd, rs1, imm[11:0]	and bit a bit con valore immediato
ori rd, rs1, imm[11:0]	or bit a bit con valore immediato
xori rd, rs1, imm[11:0]	xor bit a bit con valore immediato
slt rd, rs1, rs2	imposta un valore a complemento a 2 se è minore di un valore nel registro
sltu rd, rs1, rs2	imposta un valore senza segno se è minore di un valore nel registro
slti rd, rs1, imm[11:0]	imposta un valore a complemento a 2 se è minore di un valore immediato
sltiu rd, rs1, imm[11:0]	imposta un valore senza segno se è minore di un valore immediato
lui rd, imm[31:12]	carica l'immediato superiore
auipc rd, imm[31:12]	aggiungi l'immediato superiore a pc

Tabella 3.1: Istruzioni computazionali di RV32I

L'istruzione AUIPC prima forma un valore a 32 bit dall'immediato riempiendo i restanti 12 con zeri, aggiunge questo valore all'indirizzo dell'istruzione, infine mette il risultato nel registro *rd*. Viene usata per costruire indirizzi relativi al program counter.

Impostando il valore immediato a 0 si ottiene il PC. Inoltre supporta combinazioni di due istruzioni: la combinazione di AUIPC e il valore immediato di uno JALR può trasferire il controllo ad ogni indirizzo relativo al PC, la combinazione di AUIPC e l'offset immediato di un load o uno store può accedere ad ogni indirizzo dati relativo al PC.

Questa istruzione permette un considerevole risparmio di codice e un'aumento delle prestazioni come si vede dal codice seguente [12].

```
1 @ con AUIPC
2 auipc x4, 0x1
3 lw x4, 0x234(x4)
4
5 @ senza AUIPC
6 jal x4, 0x4
7 lui x5, 0x1
8 add x4, x4, x5
9 lw x4, 0x230(x4)
```

Codice 3.1: Codice per caricare una variabile a distanza 0x234 dal pc

Istruzioni di Trasferimento del Controllo

Istruzione	Risultato
jal rd, imm[20:1]	jump and link
jalr rd, rs1, imm[11:0]	jump and link a registro
beq rs1, rs2, imm[12:1]	salta se i valori di $rs1$ e $rs2$ sono uguali
bne rs1, rs2, imm[12:1]	salta se i valori $rs1$ e $rs2$ non sono uguali
blt rs1, rs2, imm[12:1]	salta se rs1 è minore di rs2, compara valori a complemento a due
bge rs1, rs2, imm[12:1]	salta se rs1 è maggiore di rs2, compara valori a complemento a due
bltu rs1, rs2, imm[12:1]	salta se rs1 è minore di rs2, compara valori senza segno
bgeu rs1, rs2, imm[12:1]	salta se rs1 è maggiore di rs2, compara valori senza segno

Tabella 3.2: Istruzioni di trasferimento di controllo di RV32I

Come si puó vedere nella tabella 3.2 le istruzioni di trasferimento di controllo sono essenzialmente di due tipi: **salti non condizionati**, con le istruzioni di jump e link, e **salti condizionati**.
Le due istruzioni per i *salti non condizionati* sono:

- **JAL** (jump and link) usa il formato J. Il valore immediato a 20 bit è shiftato a destra di 1 e poi esteso alla lunghezza dell'istruzione in modo da poter essere sommato all'indirizzo dell'istruzione stessa. In questo modo si può fare un salto in un range di $\pm 1MB$.
  L'istruzione successiva al JAL viene memorizzata nel registro rd, la ABI indica che venga usato il registro $x1$ o in alternativa $x5$ perché la codifica differisce di un solo bit.

- **JALR**, salto indiretto e usa il formato I. L'indirizzo di arrivo è ottenuto aggiungendo il valore immediato a 12 bit esteso alla lunghezza dell'istruzione al registro il cui indirizzo è salvato in $rs1$, per poi porre il bit meno significativo del risultato a zero. L'ultimo passaggio viene effettuato per semplificare l'hardware e per permettere di memorizzare più informazioni.

Come accennato nella sezione 1.6.2, le istruzioni di salto non condizionato usano l'indirizzamento relativo al Program Counter perché è molto probabile che l'indirizzo di destinazione sia vicino all'indirizzo dell'istruzione corrente, quindi al valore memorizzato nel Program Counter.
Nel caso in cui si voglia saltare più di 2^{18} parole si usa una sequenza di due istruzioni: lui per scrivere i bit da 12 a 31 in un registro temporaneo e $jalr$ effettua il salto sommando i 12 bit meno significativi all'indirizzo nel registro temporaneo.

Le istruzioni di *salto condizionato* usano tutte il formato B, in cui il campo immediato da 12 bit rappresenta l'offset, che viene esteso e aggiunto all'indirizzo dell'istruzione per ottenere l'indirizzo di arrivo.
Tutte le istruzioni per il salto condizionato comparano due registri, ma con condizioni diverse:

- BEQ e BNE controllano che $rs1$ e $rs2$ siano rispettivamente uguali o non uguali
- BLT e BLTU controllano che $rs1$ sia minore di $rs2$, rispettivamente con un confronto con segno o senza segno
- BGE e BGEU controllano che $rs1$ sia maggiore di $rs2$, rispettivamente con un confronto con segno o senza segno

Load e Store

La funzione di load e store è quello di trasferire un valore tra registri e memoria. Sono le uniche istruzioni che possono accedere ai registri, dato che RISC-V è una architettura load-store [2] come molte altre architetture RISC.

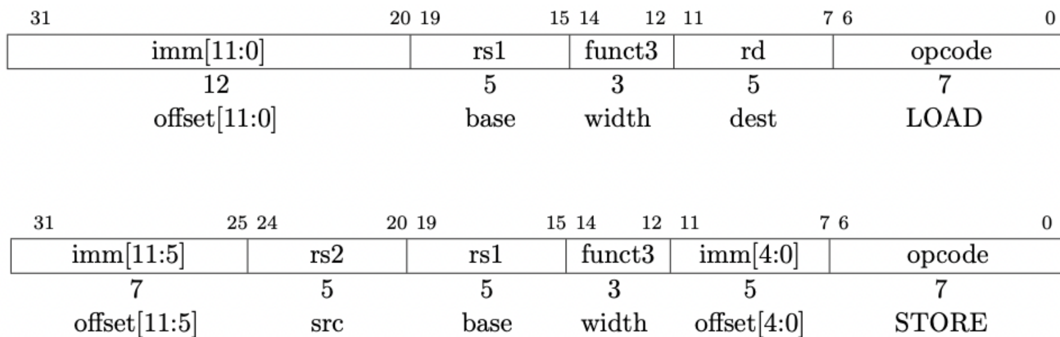


Figura 3.4: Composizione dei bit delle istruzioni load e store

L'istruzione LOAD è codificata nel formato I. Copia un valore dalla memoria al registro rd, l'indirizzo si ottiene sommando il registro $rs1$ e l'offset da 12 bit esteso. Ha diverse varianti:

- LW, carica una word da 32 bit
- LH, carica una half-word da 16 bit
- LHU, carica una half-word ma prima di memorizzarla viene zero-estesa[3] a 32 bit
- LB, carica un byte
- LBU, come LHU ma con un byte

Lo STORE è codificato nel formato S, copia il valore nel registro $rs2$ nella memoria ed . L'indirizzo si ottiene nello stesso modo della LOAD e ha varianti per salvare word da 32 bit (SW), half-word da 16 bit (SH) e byte (SB).

[2]In un'architettura load-store le istruzioni di memoria (load e store) sono separate dalle istruzioni ALU

[3]Tutti i bit più significativi oltre al valore da 16 bit vengono posti a zero

3.2 Base RV32E

RV32E è stato pensato per dispositivi embedded piccoli che richiedono meno performance e dunque devono essere anche meno costi, come ad esempio i microcontroller. L'unico cambiamento dalla base a 32 bit intera è nel numero di registri, ridotti a 16 ($x0 - x15$) di cui 15 utilizzabili.
Tutte le istruzioni della base RV32I sono supportate, con il limite di poter accedere solo ai 16 registri presenti.
Non è compatibile con la ABI di RV32I in quanto non è pensato per sistemi operativi con tutte le funzioni. [12]

3.3 Base Intera RV64I

RV32I non è sufficiente a soddisfare le esigienze di memoria indirizzabile al giorno d'oggi, soprattutto nel mondo personal computer in cui la quasi totalità dei dispositivi è a 64 bit [4][13] ma anche sempre di più nei dispositivi mobili, dove il Google Play Store richiede la compatibilità per i dispositivi a 64 bit nelle app[14] e si prevede per il 2023 dispositivi completamente a 64 bit[15].
La Base Intera RV64I si basa su RV32I, aumentando la lunghezza dei registri e del Program Counter a 64 bit oltre ad aggiungere 12 nuove istruzioni elencate nelle tabelle 3.3 e 3.4.

Istruzione	**Risultato**
addw rd, rs1, rs2	somma di registri
addiw rd, rs1, imm[11:0]	aggiungi valore immediato
subw rd, rs1, rs2	sottrazione di registri
sllw rd, rs1, rs2	shift logico a sinistra di un registro
slliw rd, rs1, shamt[4:0]	shift logico a sinistra di un immediato
srlw rd, rs1, rs2	shift logico a destra di un registro
srliw rd, rs1, shamt[4:0]	shift logico a destra di un immediato
sraw rd, rs1, rs2	shift aritmetico a destra di un registro
sraiw rd, rs1, shamt[4:0]	shift aritmetico a destra di un immediato

Tabella 3.3: Istruzioni aggiuntive di RV64I per gestire valori a 32 bit con registri a 64

[4]Le statistiche sono prese da Steam, la più grande piattaforma di distrubuzione digitale di videogiochi, dunque considera una nicchia di utenza che richiede medie-alte prestazioni.

Queste istruzioni sono usate per gestire valori a 32 bit, creano valori con segno a 32 bit ignorando i 32 bit superiori.
Per SLL, SRL e SRA il valore di shift viene preso considerando i 6 bit di ordine significativo più basso del valore nel registro $rs2$. Analogamente per SLLW, SRLW e SRAW vengono presi i 4 bit meno significativi.

Istruzione	**Risultato**
lwu rd, imm[11:0] o rs1	carica una word senza segno
ld rd, imm[11:0] o rs1	carica una doppia word
sd rs2, imm[11:0] o rs2	memorizza una doppia word

Tabella 3.4: Istruzioni aggiuntive di memoria di RV64I

Dato che i registri sono a 64 bit, oltre alle istruzioni di RV32I per caricare word, half-word e byte, servono anche istruzioni di LOAD e STORE per caricare double-word, cioè 64 bit. Per fare ciò sono definite LD per il caricamento e SD per la memorizzazione.
Le istruzioni di memoria di RV32I sono adattate alla lunghezza dei registri, il load zero-estende i valori dalla memoria mentre lo store prende i valori meno significativi.

3.4 Base Intera RV128I

Anche se al momento della scrittura non sono ancora necessari più di 64 bit di memoria indirizzabile, la commissione di RISC-V ha deciso di definire una ISA con 128 bit di spazio indirizzabile perché è la soluzione migliore e più facile[1].
RV128I viene costruita sopra RV64I, esattamente come succedeva con RV64I che costruisce sopra a RV32I: i registri vengono allungati a 128 bit, vengono definite nuove istruzioni con il suffisso D per gestire valori a 64 bit e ulteriori load e store per gestire quadword.
Al momento non è ancora congelata per evolverla quando verrà effettivamente utilizzata.

Capitolo 4

ISA Privilegiata

La ISA che si occupa di gestire il sistema è separata dalla ISA a livello utente, trattata nel capitolo precedente. In questo modo tutta la parte privilegiata potrebbe essere sostituita senza modificare la parte non privilegiata.
Quindi l'architettura descritta nel report tecnico[6] e di conseguenza in questo capitolo è di riferimento ed è pensata per supportare un sistema Unix con memoria virtuale.

Una ISA privilegiata è necessaria in quanto la ISA base non gestisce risorse condivise che hanno bisogno di protezione come la memoria, i dispositivi di I/O o i core. Inoltre permette di gestire eventi asincroni esterni come i software interrupt.

4.1 Livelli di Privilegio

I livelli di privilegio sono usati per garantire protezione all'hardware e al software, infatti se del software prova ad eseguire operazioni non permesse dal suo livello verrà lanciata un'eccezione. Ogni livello aggiunge qualche istruzione e dei CSR (Registri di Controllo e Stato). Di base i livelli di RISC-V sono:

1. User con abbreviazione U, le cui istruzioni permesse sono state trattate nel Capitolo 3

2. Supervisor con abbrieviazione S, trattato nella Sezione 4.3

3. Machine con abbreviazione M, trattato nella Sezione 4.2

Non è necessario che siano presenti tutti i livelli, l'unico obbligatorio è il livello Macchina, ma molte implementazioni includono la modalità Utente in cui eseguire

applicazioni. Dato che è un modello piramidale, i livelli con maggiori privilegi possono eseguire le istruzioni dei livelli con meno privilegi, ma non il contrario.

Le diverse combinazioni di livelli di privilegio e altre opzioni cosituiscono i cosiddetti **Profili della Piattaforma**, definiti nella RISC-V Platform Specification [16] dalla RISC-V Platform Horizontal Subcommittee[17].
Attualmente sono definite due piattaforme principali, con due sottocategorie ciascuna:

1. **Piattaforma OS-A**, sono piattaforme che supportano sistemi operativi complessi come Linux o Windows. Le sottocategorie sono indipendenti una dall'altra e sono:
   - OS-A Embedded
   - OS-A Server
2. **Piattaforma M**, sono piattaforme per applicazioni senza OS o con un piccolo OS su un microcontroller. Ha una base e un'estensione chiamata PMP (Physical Memory Protection).

In futuro ci si aspetta che vengano aggiunte altre piattaforme come "machine-learning" o "edge computing".

Profilo	**Livelli**	**Chi ha la Fiducia**
Embedded senza Protezione	M	Completa
Embedded con Protezione	M e U	no Applicazioni
Sistemi con OS tipo Unix	M,S e U	OS

Tabella 4.1: Alcuni profili base

4.2 ISA Livello Macchina

La modalità macchina è la più alta nella gerarchia dei livelli e ha l'accesso completo all'hardware. È la prima modalità in cui si entra dopo un reset.

4.2.1 Istruzioni

Le istruzioni aggiunte a livello macchina hanno tutte opcode SYSTEM e sono:

- **ECALL**, usata per implementare le system call[1]. Sono delle richieste di codice a più basso privilegio di eseguire codice a livello di privilegio più alto, genera un'eccezione diversa in base al livello di privilegio originante. Ad esempio, la modalità M genera un'eccezione environment-call-from-M-mode. Imposta il registro `epc` con l'indirizzo dell'ECALL.

- **EBREAK**, genera un'eccezione di breakpoint. Viene usato dai debugger per trasferire il controllo all'ambiente di debugging. Imposta il registro `epc` con l'indirizzo dell'EBREAK.

- **MRET**, **SRET**, usate per ritornare da una trap. Sono rispettivamente per la modalità M e per la modalità S. Imposta il `pc` al valore nel registro `xepc`.

- **WFI**, serve per dire all'implementazione che l'hart può essere bloccato finché un interrupt non è servito. Un'implementazione possibile è semplicemente come un NOP.

4.2.2 Cosa succede al Reset

Come accennato prima, dopo un reset un hart viene impostato al livello di privilegio M. Inoltre vengono modificati i seguenti registri:

- campi MIE e MPRV del registro `mstatus/mstatush` a 0

- registro `misa` reimpostato con il massimo numero di estensioni e il MXLEN più lungo

- program counter ad un valore definito dall'implementazione

[1] Richiesta dal livello utente di un servizio a livello più basso, di solito al kernel

- registro `mcause` impostato ad un valore che indica la causa del reset. Il valore 0 è usato per indicare il reset più completo nelle implementazioni che distinguono le cause del reset, mentre è l'unico valore di ritorno in implementazioni che non le distinguono.
- campi `A` e `L` di `PMP` a 0

4.2.3 Attributi della Memoria Fisica (PMA)

Il sistema operativo di un dispositivo ha bisogno di sapere che zone della memoria può interrogare nei momenti giusti. Per questo scopo, si usa una struttura dati che definisce gli spazi di memoria detta **Mappa della Memoria** o **Mappa degli Indirizzi** e include vari intervalli di indirizzi tra cui:

- effettive regioni di memoria, memoria principale.
- registri di controllo mappati a memoria. Classificate come regioni I/O come qualsiasi porzione di memoria che non è principale.
- zone vacanti, che servono, per esempio, ad I/O mappati a memoria o per periferiche PCI. Classificate come regioni I/O a cui non si può accedere.

Ogni regione ha delle proprietà, come ad esempio se supporta la lettura o la scrittura.
In ARM, le proprietà di accesso sono regolate da dei bit di controllo di accesso detti AP (Access Permission) posti in ogni voce della tabella di memoria virtuale. Questi bit, insieme all'indirizzo fisico, vengono mandati alla memoria principale dalla TLB[2], se la voce della tabella è presente in essa[18].

In RISC-V vengono chiamati **PMA** (Physical Memory Attributes) e non si limitano solo all'accesso. Come gli AP di ARM, i PMA vanno controllati ad ogni accesso alla memoria fisica. La differenza principale con Arm è che la definizione e il controllo avvengono in due strutture hardware differenti: la definizione dipende dall'implementazione e può essere, ad esempio, definito nel chip, mentre il controllo avviene nel *PMA checker*.
La memoria principale richiede diversi PMA:

- Tipo di Accesso Supportato, specifica le lunghezze supportate e se sono supportati accessi disallineati.

[2]Translation Lookaside Buffer, è una cache delle voci della tabella virtuale

- AMO[3],definisce che operazioni possono operare sulla zona di memoria, ha quattro livelli:

AMP PMA	Operazioni supportate
AMOnone	nessuna
AMOSwap	amoswap
AMOLogical	sopra + amoand, amoor,amoxor
AMOArithmetic	sopra + amoadd,amomin,amomax,amominu,amomaxu

[6]

Tabella 4.2: PMA AMO supportate

- Riservabilità, supporto per le operazioni LR e SC[4], ci sono tre livelli:

PMA	Operazioni supportate
RsrvNone	nessuna
RsvrNonEventual	supportate, ma senza la garanzia evenutale di successo
RsrvEventual	supportate con la garanzia

Tabella 4.3: PMA di riservabilità supportati

- Allineamento, se le istruzioni LR/SC e AMO non allineate[5] generano un'eccezione "indirizzo non allineato", allora anche tutti i load,store,LR/SC e AMO devono generare la stessa eccezione

- Coerenza[6] e Cachebilità, come provvedere alla coerenza in diversi casi:

  1. per una regione di memoria non memorizzata nella cache, il PMA indica che non deve essere messo in una cache privata[7] o condivisa.
  2. per regioni di memoria di sola lettura, il PMA indica che può essere messa in cache e le scritture non sono supportate
  3. per regioni di lettura e scrittura accessibili da un solo agente, il PMA indica che può essere messa in cache

[3]Sono le operazioni di memoria atomiche, definite nell'estensione standard "A" per le istruzioni atomiche. Fanno delle operazioni di lettura-modifica-scrittura per la sincronizzazione multiprocessore [1]

[4]Sempre dall'estensione standard "A", sono operazioni di memoria atomiche che operano su una singola o doppia word di memoria [1]

[5]Non nell'estensione standard "A"

[6]Proprietà per la quale una scrittura sarà visibile ad altri agenti nel sistema

[7]Con cache privata si intende una cache collegata ad un agente master

4. per regioni di lettura e scrittura accessibili ad altri agenti, è richiesto una schema di coerenza della cache (hardware o software). In questo caso il PMA, per ogni regione con coerenza hardware, indica che la regione è coerente e il controller di coerenza hardware da utilizzare.

- Idempotenza, specifica se le scritture o letture ad una regione sono idempotenti, cioè se le operazioni vengono applicate più volte il risultato rimane uguale alla prima applicazione.

I primi tre PMA sono raggruppabili nei PMA di Atomicità, che descrivono che istruzioni atomiche (LR/SC e AMO) sono supportate.

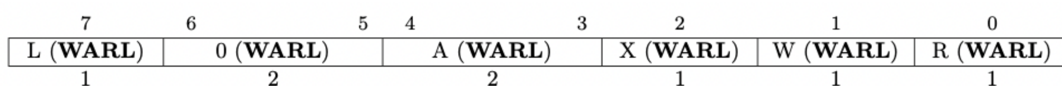
4.2.4 Protezione della Memoria Fisica (PMP)

L'unità opzionale PMP (Physical Memory Protection) permette di limitare gli indirizzi fisici accessibili al software tramite registri di controllo CSR in ogni hart. I controlli avvengono su tutte le operazioni che operano con modalità di privilegio S o U, anche se, opzionalmente, è possibile controllare anche gli accessi con modalità M.

Possono esserci al massimo 64 valori del PMP, e ognuno viene rappresentato da un registro di configurazione da 8 bit e da un registro degli indirizzi da MXLEN bit.

Registri di Configurazione

I registri di configurazione vengono compressi nei CSR: in RV32 un CSR può tenere 4 configurazioni, dunque 15 CSR (`pmpcf0-pmpcf15`) tengono tutti i 64 valori; in RV64 un CSR può tenere 8 configurazioni, dunque 8 configurazioni tengono tutti i 64 valori.
Un registro di configurazione è formato come nella figura 4.1.



[6]

Figura 4.1: Opcode di RV32I

Il campo R abilita la lettura, il campo W la scrittura e il campo X l'esecuzione. Il campo A codifica la modalità di indirizzamento del registro d'indirizzo PMP corrispondente. Le codifiche sono:

Nome	Valore di A	Descrizione
OFF	0	disabilitato
TOR	1	limite superiore
NA4	2	regione da 4 byte allineata naturalmente[8]
NAPOT	3	regione maggiore di 8 byte e una potenza di 2 allineata naturalmente

Tabella 4.4: Codifica di A nel registro d'indirizzo

Se è selezionato TOR, l'intervallo di indirizzi è delimitato superiormente dal registro d'indirizzo associato e inferiormente dal registro d'indirizzo PMP.

Il campo L indica che il valore del PMP è bloccato, dunque tutte le scritture ai registri sia di configurazione che di indirizzo sono ignorate. Inoltre se il campo L è impostato, i permessi R/W/X vengono impostati per tutti i livelli di privilegio, altrimenti solo per il livello M.

Registri d'Indirizzo

Per RV32 ogni registro d'indirizzo codifica i bit da 33 a 2 di un'indirizzo fisico a 34 bit, mentre in RV64 codifica i bit da 55 a 2 di un'indirizzo fisico da 56 bit.
I registri sono chiamati `pmpaddrX` con `X` tra 0 e 63.

[8] L'allineamento naturale è la più piccola potenza di 2 che può contenere il valore

4.3 ISA Livello Supervisore

Il livello Supervisore è posto tra il livello Macchina e il livello Utente. Ha un'interazione ristretta con l'hardware fisico, il necessario per supportare la virtualizzazione[9] [6]

4.3.1 Memoria Virtuale

Come accennato nell'introduzione, il livello Supervisore è pensato per supportare la virtualizzazione e di conseguenza la memoria virtuale. Infatti questo livello, quando il csr `satp` viene impostato correttamente, esegue in un sistema di memoria virtuale a pagine in cui gli indirizzi virtuali dei livello supervisore ed utente vengono trasformati in indirizzi fisici del livello supervisore.

La memoria virtuale è un sistema per aumentare la memoria principale usando la memoria di massa. Viene realizzata per due motivi:

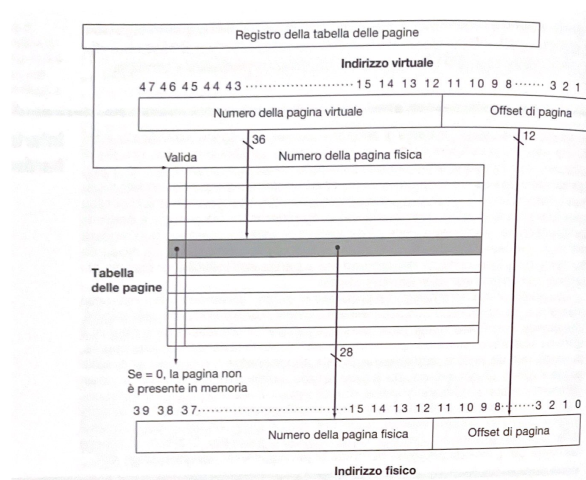
- aumentare la limitata memoria principale, e dunque non costringere il programmatore a gestire lo spazio
- condividere la stessa memoria tra più macchine virtuali in modo sicuro

Per ogni pagina[10] della memoria virtuale, viene generato un'indirizzo virtuale. In RISC-V l'indirizzo è diviso nel numero della pagina virtuale e nell'offset e può essere da 32,39,48 o 57 bit.
L'indirizzo virtuale viene tradotto in indirizzo fisico trasformando il numero di pagina virtuale grazie alle tabelle di pagine. Nell'indirizzo virtuale a 32 bit, la tabella di pagina è a due livelli; salendo di spazio di indirizzamento aumentano di uno i livelli fino ad arrivare alla tabella a 5 livelli con indirizzo virtuale a 57 bit. L'offset, invece, rimane invariato. La Figura 4.2 rappresenta la traduzione di un'indirizzo virtuale a indirizzo fisico con tabella di pagine a un livello.
Le tabelle su più livelli sono usate per ridurre lo spazio in memoria della tabella stessa, che richiederebbe 0,5 TiB[20]. Il numero di pagina virtuale è diviso in varie porzioni di lunghezza uguale che rappresentano una voce nelle tabelle di vario livello. La porzione con bit più significativi viene usata nella tabella di

[9] Usa il software per creare un livello di astrazione che permette di dividere l'hardware di un computer in diversi computer virtuali, chiamate Virtual Machine (VM), ognuna con il suo sistema operativo [19]

[10] Una pagina è un blocco nella memoria virtuale

livello 0, la seconda porzione nella tabella di livello 1 e così via. Si può procedere al livello successivo solo se il livello precedente è stato completato, dunque è stata trovata la pagina nella tabella. Si risparmia spazio in quanto vengono salvati segmenti non contigui e non serve allocare la tabella delle pagine completa.



[20]

Figura 4.2: Conversione dell'indirizzo virtuale in indirizzo fisico

La tabella delle pagine è costituita dalle voci, chiamate PTE (page-table entries), ognuna da 4 byte. Ogni voce ha i primi 10 bit codificati nel modo seguente:

- bit 0 codificato come V, indica se la voce è valida. Se è 0, la pagina non è presente in memoria
- bit 1 codificato come R, indica se la pagina è leggibile
- bit 2 codificato come W, indica se la pagina è scrivibile
- bit 3 codificato come X, indica se la pagina è eseguibile
- bit 4 codificato come U, indica se la pagina è accessibile in modalità utente

- bit 5 codificato come G, indica una mappatura globale, cioè che esiste in tutti gli spazi di indirizzamento
- bit 6 codificato come A, indica che dall'ultima volta in cui il bit è stato cancellato, sono state effettuate operazioni sulla pagina, per esempio delle letture
- bit 7 codificato come D, indica che la pagina è stata scritta

Il problema principale della memoria virtuale è che usando la memoria di massa, molto più lenta della memoria principale, i page fault[11] richiedono molti cicli di clock per essere risolti. Per questo, la dimensione delle pagine è abbastanza grande, in RISC-V 4 KiB, in modo da smorzare il tempo perso.

[11] La pagina non è presente nella memoria principale

4.3.2 Istruzioni

Il Livello Supervisore aggiunge una sola nuova istruzione: SFENCE.VMA.
Questa istruzione ha la stessa funzione della FENCE, cioè quella di ordinare la visualizzazione dell'esecuzione di istruzioni dal punto di vista di hart esterni all'esecuzione. La differenza è che SFENCE.VMA viene usato per sincronizzare aggiornamenti nelle strutture dati per la gestione della memoria con l'esecuzione corrente[6]. In pratica, viene usata per svuotare la cache costringendo la MMU[12] a cercare nella memoria principale e quindi permettere al sistema di accorgersi dei cambiamenti.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
SFENCE.VMA	asid	vaddr	PRIV	0	SYSTEM

Figura 4.3: Istruzione SFENCE.VMA

Le operazioni ordinate da SFENCE.VMA dipendono da $rs1$ e $rs2$:

- se $rs1$ e $rs2$ sono uguali a $x0$, ordina tutte le letture e le scritture di tutte le tabelle delle pagine
- se solo $rs1 = x0$, ordina le letture e le scritture delle tabelle nello spazio di indirizzi $rs2$
- se solo $rs2 = x0$, ordina le letture e le scritture delle tabelle di livello superiore allo 0 corrispondenti all'indirizzo virtuale $rs1$
- se nessuno dei due è uguale a $x0$, ordina le letture e le scritture delle tabelle di livello superiore allo 0 corrispondenti all'indirizzo virtuale $rs1$ e nello spazio di indirizzi $rs2$

[12]Memory Management Unit, traduce gli indirizzi virtuali in fisici [3]

Capitolo 5

Esempi di Codice

Questa sezione presenta alcuni algoritmi scritti in assembler Risc-V.
Prima del codice, vengono introdotte le direttive dell'assembler utilizzabili.

5.1 Linguaggio Assembly

I programmi RISC-V vengono scritti in linguaggio assembly, che viene poi convertito dall'assembler in codice macchina eseguibile.
Un file assembly contiene direttive, istruzioni e macro e ha l'estensione *.s* o *.S* che indica che il file va manipolato dal preprocessore C. Il file assembly viene convertito dall'assembler in un file oggetto con estensione *.o* che non è eseguibile, ma è input per il linker.

Un file oggetto è composto da diverse sezioni, identificate nel file assembly con le seguenti direttive:

- *.text*, sezione di sola lettura che contiene le istruzioni eseguibili
- *.data*, sezione di lettura e scrittura che contiene i dati per le variabili statiche e gloabali
- *.rodata*, sezione che contiene dati di sola lettura
- *.bss*, il nome completo è Basic Service Set, sezione di lettura e scrittura che contiene dati non inizializzati per variabili locali

5.1.1 Direttive dell'Assembler

Controllano le operazioni dell'assembler. Le direttive dell'assembler di RISC-V sono elencate nella tabella 5.1.

Direttiva	**Descrizione**
.byte valore	inizializza 8 bit non allineati
.2byte valore	inizializza 2 byte non allineati
.4byte valore	inizializza 4 byte non allineati
.8byte valore	inizializza 8 byte non allineati
.half valore	inizializza 16 bit allineati
.word valore	inizializza 32 bit allineati
.dword valore	inizializza 64 bit allineati
.asciz string	emette una stringa
.string string	emette una stringa
.zero int	riserva un blocco di memoria
.align int	allinea la prossima istruzione ad un confine di n byte
.balign byte	allinea con byte
.p2align int	allinea con una potenza di 2
.global simbolo	globalizza un simbolo
.local simbolo	limita la visibilità del simbolo
.equ simbolo, espressione	imposta il valore del simbolo ad espressione
.option [rvc,norvc,pic,nopic,push,pop]	opzioni
.macro nome, arg1, [argN]	inizio definizione macro
.endm	fine definizione macro

Tabella 5.1: Direttive dell'Assembler

5.1.2 Pseudo Istruzioni

Alcune istruzioni con specifiche combinazioni di valori o registri sono molto comuni nella programmazione assembly RISC-V, quindi, per facilitare la comprensione al programmatore, sono stati definiti per queste istruzioni degli alias, chiamate pseudo-istruzioni. Le pseudo-istruzioni di RISC-V sono elencate nella tabella 5.2.

pseudo-istruzione	**istruzione reale**	**descrizione**
nop	addi zero,zero,0	nulla
mv rd,rs1	addi rd,rs,0	copiare un registro
not rd,rs1	xori rd,rs,-1	complemento a 1
neg rd,rs1	sub rd,x0,rs	complemento a 2
negw rd,rs1	subw rd,x0,rs	complemento a 2 per una word
sext.w rd,rs1	addiw rd,rs,0	estensione di segno di una word
seqz rd,rs1	sltiu rd,rs,1	imposta se $rs1$ uguale a 0
snez rd,rs1	sltu rd,x0,rs	imposta se $rs1$ diverso da 0
sltz rd,rs1	slt rd,rs,x0	imposta se $rs1$ minore di 0
sgtz rd,rs1	slt rd,x0,rs	imposta se $rs1$ maggiore di 0
beqz rs1,offset	beq rs,x0,offset	salto se zero
bnez rs1,offset	bne rs,x0,offset	salto se diverso da zero
blez rs1,offset	bge x0,rs,offset	salto se minore uguale di zero
bgez rs1,offset	bge rs,x0,offset	salto se maggiore uguale di zero
bltz rs1,offset	blt rs,x0,offset	salto se minore di zero
bgtz rs1,offset	blt x0,rs,offset	salto se maggiore di zero
bgt rs,rt,offset	blt rt,rs,offset	salto se maggiore
ble rs,rt,offset	bge rt,rs,offset	salto se minore uguale
bgtu rs,rt,offset	bltu rt,rs,offset	salto se maggiore, senza segno
bleu rs,rt,offset	bltu rt,rs,offset	salto se minore uguale, senza segno
j offset	jal x0,offset	salto
jr offset	jal x1,offset	salto a registro
ret	jalr x0,x1,0	ritorna dalla subroutine

Tabella 5.2: Pseudoistruzioni

5.2 Azzeramento di un Vettore

Nel testo "Struttura e Progetto dei Calcolatori"[20] come esempio vengono presentati due metodi alternativi per azzerare una sequenza di interi:

```
void azzera1(long long int vettore[], int dim){
    for(int i=0; i<dim; i++){
        vettore[i] = 0;
    }
}
```

Codice 5.1: Codice C++ per la funzione di azzeramento di un array con indice

```
void azzera2(long long int* vettore, int dim){
    long long int *p;
    for(*p = &vettore[0]; p < &vettore[dim]; p++){
        *p=0;
    }
}
```

Codice 5.2: Codice C++ per la funzione di azzeramento di un array con puntatori

Il primo utilizza un indice sull'array per accedere ai valori, mentre il secondo utilizza i puntatori. Il testo propone come codice assembly per il primo metodo il seguente:

```
    li x5,0 # i=0
ciclo1:
    slli x6,x5,3        # i*8
    add x7,x10,x6       # indirizzo di vettore[i]
    sd x0,0(x7)         # vettore[i]=0
    addi x5,x5,1        # i++
    blt x5,x11,ciclo1   # salto a ciclo1 se i<dim
```

Codice 5.3: Codice assembly per azzerare un array con indice

in cui il registro `x5` viene usato per memorizzare l'iteratore, `x6` per la posizione dell'elemento i in byte rispetto all'inizio dell'array, `x10` per la posizione del primo elemento dell'array, `x11` per il parametro `dim` e `x7` per l'indirizzo assoluto dell'elemento i dell'array. Il codice sarà nella sezione `.text` e molto probabilmente sarà presente la linea `.global azzera1` che permette al linker di vedere il simbolo in modo da poter essere usata da altri file oggetto. Se non ci fosse, il simbolo sarebbe considerato locale al file. La riga 5 del Codice 5.3 effettua una dereference

in quanto x7 è un indirizzo e memorizza zero in quella cella di memoria. La riga 7 permette di effettuare il ciclo finché l'iteratore non è maggiore o uguale della dimensione dell'array.
Il secondo metodo in assembly è scritto nel modo seguente:

```
1      mv x5,x10            # indirizzo di vettore[0]
2      slli x6,x11,3        # dim * 8
3      add x7,x10,x6        # indirizzo di vettore[dim]
4  ciclo2:
5      sd x0,0(x5)          # memoria[p]=0
6      addi x5,x5,8         # p+8
7      bltu x5,x7,ciclo2    # salta a ciclo 2 se p<&vettore[dim]
```

Codice 5.4: Codice assembly per azzerare un array con puntatori

che usa tre istruzioni in meno nel ciclo. In entrambi i codici, l'istruzione slli viene usata per moltiplicare l'iteratore per 8 per ottenere l'offset in byte, ricordando che fare uno shift a sinistra di N posizioni è equivalente ad effettuare una moltiplicazione per 2^N.

Il compilatore non genererà esattamente i codici 5.3 e 5.4. Infatti usando la Risc-V GNU toolchain [21], che permette di usare il compilatore gcc con RISC-V, usando i comandi:
`riscv64-unknown-elf-gcc -nostartfiles file.cpp -o file`
per compilare usando la toolchain e
`riscv64-unknown-elf-objdump -D file`
per visualizzare l'assembly a partire dal file oggetto otteniamo:

```
1      0000000000010078 <azzera1>:
2     10078: 7179                addi sp,sp,-48
3     1007a: f422                sd s0,40(sp)
4     1007c: 1800                addi s0,sp,48
5     1007e: fca43c23          sd a0,-40(s0)
6     10082: 87ae                mv a5,a1
7     10084: fcf42a23          sw a5,-44(s0)
8     10088: fe042623          sw zero,-20(s0)
9     1008c: a831                j 100a8 <azzera1+0x30>
10    1008e: fec42783          lw a5,-20(s0)
11    10092: 078e                slli a5,a5,0x3
12    10094: fd843703          ld a4,-40(s0)
13    10098: 97ba                add a5,a5,a4
14    1009a: 0007b023          sd zero,0(a5)
15    1009e: fec42783          lw a5,-20(s0)
```

```
16    100a2: 2785                addiw a5,a5,1
17    100a4: fef42623          sw a5,-20(s0)
18    100a8: fec42783          lw a5,-20(s0)
19    100ac: 873e                mv a4,a5
20    100ae: fd442783          lw a5,-44(s0)
21    100b2: 2701                sext.w a4,a4
22    100b4: 2781                sext.w a5,a5
23    100b6: fcf74ce3          blt a4,a5,1008e <azzera1+0x16>
24    100ba: 0001                nop
25    100bc: 0001                nop
26    100be: 7422                ld s0,40(sp)
27    100c0: 6145                addi sp,sp,48
28    100c2: 8082                ret
```

Codice 5.5: Risultato dell'object dump di azzera1

per il metodo con indici. Le righe 21 e 22 permettono di estendere i valori presenti in `a4` e `a5` in modo da poter effettuare un confronto corretto alla riga successiva. A differenza del Codice 5.1, il `mov` viene posizionato appena prima delle estensioni di segno, ma al primo ciclo sarà comunque la prima istruzione ad eseguire grazie al jump presente alla riga 9. Il ciclo poi comincerà dall'istruzione appena dopo il jump.

Per il metodo dei puntatori invece:

```
1      0000000000010078 <azzera2>:
2     10078: 7179                addi sp,sp,-48
3     1007a: f422                sd s0,40(sp)
4     1007c: 1800                addi s0,sp,48
5     1007e: fca43c23          sd a0,-40(s0)
6     10082: 87ae                mv a5,a1
7     10084: fcf42a23          sw a5,-44(s0)
8     10088: fd843783          ld a5,-40(s0)
9     1008c: fef43423          sd a5,-24(s0)
10    10090: a811                j 100a4 <azzera2+0x2c>
11    10092: fe843783          ld a5,-24(s0)
12    10096: 0007b023          sd zero,0(a5)
13    1009a: fe843783          ld a5,-24(s0)
14    1009e: 07a1                addi a5,a5,8
15    100a0: fef43423          sd a5,-24(s0)
16    100a4: fd442783          lw a5,-44(s0)
17    100a8: 078a                slli a5,a5,0x2
18    100aa: fd843703          ld a4,-40(s0)
19    100ae: 97ba                add a5,a5,a4
20    100b0: fe843703          ld a4,-24(s0)
```

```
21      100b4: fcf76fe3           bltu a4,a5,10092 <azzera2+0x1a>
22      100b8: 0001                 nop
23      100ba: 0001                 nop
24      100bc: 7422                 ld s0,40(sp)
25      100be: 6145                 addi sp,sp,48
26      100c0: 8082                 ret
```

Codice 5.6: Risultato dell'objdump di azzera2

In entrambi i codici, la prima riga di codice, all'indirizzo 10078, alloca spazio decrementando lo stack pointer per fare in modo di avere spazio per le variabili locali. Le due righe successive servono per impostare $s0$ come frame pointer.
Il compilatore, a differenza dei codice 5.3 e 5.4, non usa i registri come memoria, ma piuttosto come supporto temporaneo per effettuare le operazioni, in quanto i dati vengono salvati nello stack e le istruzioni operano su registri. Tutti i load e gli store aggiuntivi, tranne `sd zero,0(a5)` che viene usato per azzerare il valore, sono operazioni per reperire o memorizzare valori come l'indice o l'indirizzo dell'array nello stack. Dato che cresce verso l'alto e lo stack pointer è posto alla base, sono necessari degli offset negativi per raggiungere l'indirizzo corretto.
Le ultime tre istruzioni in entambi i codici sono per reimpostare lo stack pointer e ritornare dalla subroutine.
Ignorando le istruzioni di load e store generate dal compilatore per la gestione dello stack, l'object dump di `azzera1` risulta molto simile al Codice 5.3, mentre l'object dump di `azzera2` ha qualche differenza dal Codice 5.4: in particolare, al primo ciclo viene salvato anche l'indirizzo base nella posizione dello stack `-40` e ad ogni ciclo viene calcolato l'indirizzo della fine dell'array alla riga 19, non fuori dal ciclo come nel codice proposto dal testo.

5.3 Quick Sort

Il Quicksort è algoritmo di ordinamento divide-and-conquer ricorsivo. L'approccio è quello di dividere l'array in due e riordinare ricorsivamente le due metà dividendole ancora.In C++ l'implementazione del pseudocodice[22] è la seguente:

```
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}

int partition (int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high- 1; j++)
    {
        if (arr[j] <= pivot)
        {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quickSort(int arr[], int low, int high) {
    if (low < high)
    {
        int pivot = partition(arr, low, high);

        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}
```

Codice 5.7: Implementazione in C++ del pseudocodice dell'algoritmo di Quicksort ricorsivo[22]

Sono implementate due funzioni principali e la funzione di utility swap per scambiare due elementi. La funzione `quicksort` é la principale, chiama `partition` e se stessa ricorsivamente due volte: la prima chiamata é sull'insieme tra il lower bound e il pivot, la seconda tra il pivot e l'upper bound.
La funzione `partition` modifica l'array sul posto dividendolo in due parti, una con gli elementi maggiori del pivot e l'altra con elementi minori. Ritorna la posizione del pivot.

La traduzione della funzione `quicksort` in Assembly, ottenuta tramite lo stesso comando usato nella sezione precedente, è:

```
1  0000000000010184 <_Z9quickSortPiii>:
2     10184: 7179                 addi sp,sp,-48
3     10186: f406                 sd ra,40(sp)
4     10188: f022                 sd s0,32(sp)
5     1018a: 1800                 addi s0,sp,48
6     1018c: fca43c23           sd a0,-40(s0)
7     10190: 87ae                 mv a5,a1
8     10192: 8732                 mv a4,a2
9     10194: fcf42a23           sw a5,-44(s0)
10    10198: 87ba                 mv a5,a4
11    1019a: fcf42823           sw a5,-48(s0)
12    1019e: fd442783           lw a5,-44(s0)
13    101a2: 873e                 mv a4,a5
14    101a4: fd042783           lw a5,-48(s0)
15    101a8: 2701                 sext.w a4,a4
16    101aa: 2781                 sext.w a5,a5
17    101ac: 04f75863           bge a4,a5,101fc <_Z9quickSortPiii+0x78>
18    101b0: fd042703           lw a4,-48(s0)
19    101b4: fd442783           lw a5,-44(s0)
20    101b8: 863a                 mv a2,a4
21    101ba: 85be                 mv a1,a5
22    101bc: fd843503           ld a0,-40(s0)
23    101c0: eefff0ef           jal ra,100ae <_Z9partitionPiii>
24    101c4: 87aa                 mv a5,a0
25    101c6: fef42623           sw a5,-20(s0)
26    101ca: fec42783           lw a5,-20(s0)
27    101ce: 37fd                 addiw a5,a5,-1
28    101d0: 0007871b           sext.w a4,a5
29    101d4: fd442783           lw a5,-44(s0)
30    101d8: 863a                 mv a2,a4
31    101da: 85be                 mv a1,a5
32    101dc: fd843503           ld a0,-40(s0)
```

```
33    101e0: fa5ff0ef           jal ra,10184 <_Z9quickSortPiii>
34    101e4: fec42783           lw a5,-20(s0)
35    101e8: 2785                 addiw a5,a5,1
36    101ea: 2781                 sext.w a5,a5
37    101ec: fd042703           lw a4,-48(s0)
38    101f0: 863a                 mv a2,a4
39    101f2: 85be                 mv a1,a5
40    101f4: fd843503           ld a0,-40(s0)
41    101f8: f8dff0ef           jal ra,10184 <_Z9quickSortPiii>
42    101fc: 0001                 nop
43    101fe: 70a2                 ld ra,40(sp)
44    10200: 7402                 ld s0,32(sp)
45    10202: 6145                 addi sp,sp,48
46    10204: 8082                 ret
```

Codice 5.8: Risultato dell'objdump di Quick Sort

La ricorsione avviene nelle righe con indirizzo 101e0 e 101f8, con dei jump all'inizio della funzione. Prima del salto vengono impostati i registri $a1$ e $a2$ per avere rispettivamente i valori di low e high: nella prima ricorsione il valore di low é il parametro della funzione chiamante, quindi é sufficiente fare un load all'address corretto mentre il valore di high é pivot decrementato di 1, nella seconda ricorsione la situazione é ribaltata e il pivot va aumentato di 1. I valori vengono prima modificati su $a4$ e $a5$ prima di essere spostati su $a1$ e $a2$ in modo da avere una copia dei parametri senza cambiamenti.

La funzione `partition` viene tradotta nel modo seguente:

```
00000000000100ae <_Z9partitionPiii>:
   100ae: 7179                 addi sp,sp,-48
   100b0: f406                 sd ra,40(sp)
   100b2: f022                 sd s0,32(sp)
   100b4: 1800                 addi s0,sp,48
   100b6: fca43c23           sd a0,-40(s0)
   100ba: 87ae                 mv a5,a1
   100bc: 8732                 mv a4,a2
   100be: fcf42a23           sw a5,-44(s0)
   100c2: 87ba                 mv a5,a4
   100c4: fcf42823           sw a5,-48(s0)
   100c8: fd042783           lw a5,-48(s0)
   100cc: 078a                 slli a5,a5,0x2
   100ce: fd843703           ld a4,-40(s0)
   100d2: 97ba                 add a5,a5,a4
   100d4: 439c                 lw a5,0(a5)
   100d6: fef42223           sw a5,-28(s0)
   100da: fd442783           lw a5,-44(s0)
   100de: 37fd                 addiw a5,a5,-1
   100e0: fef42623           sw a5,-20(s0)
   100e4: fd442783           lw a5,-44(s0)
   100e8: fef42423           sw a5,-24(s0)
   100ec: a881                 j 1013c <_Z9partitionPiii+0x8e>
   100ee: fe842783           lw a5,-24(s0)
   100f2: 078a                 slli a5,a5,0x2
   100f4: fd843703           ld a4,-40(s0)
   100f8: 97ba                 add a5,a5,a4
   100fa: 4398                 lw a4,0(a5)
   100fc: fe442783           lw a5,-28(s0)
   10100: 2781                 sext.w a5,a5
   10102: 02e7c863           blt a5,a4,10132 <_Z9partitionPiii+0x84>
   10106: fec42783           lw a5,-20(s0)
   1010a: 2785                 addiw a5,a5,1
   1010c: fef42623           sw a5,-20(s0)
   10110: fec42783           lw a5,-20(s0)
   10114: 078a                 slli a5,a5,0x2
   10116: fd843703           ld a4,-40(s0)
   1011a: 00f706b3           add a3,a4,a5
   1011e: fe842783           lw a5,-24(s0)
   10122: 078a                 slli a5,a5,0x2
   10124: fd843703           ld a4,-40(s0)
   10128: 97ba                 add a5,a5,a4
   1012a: 85be                 mv a1,a5
```

```
44     1012c: 8536                mv a0,a3
45     1012e: f4bff0ef           jal ra,10078 <_Z4swapPiS_>
46     10132: fe842783           lw a5,-24(s0)
47     10136: 2785                addiw a5,a5,1
48     10138: fef42423           sw a5,-24(s0)
49     1013c: fd042783           lw a5,-48(s0)
50     10140: 873e                mv a4,a5
51     10142: fe842783           lw a5,-24(s0)
52     10146: 2701                sext.w a4,a4
53     10148: 2781                sext.w a5,a5
54     1014a: fae7c2e3           blt a5,a4,100ee <_Z9partitionPiii+0x40>
55     1014e: fec42783           lw a5,-20(s0)
56     10152: 0785                addi a5,a5,1
57     10154: 078a                slli a5,a5,0x2
58     10156: fd843703           ld a4,-40(s0)
59     1015a: 00f706b3           add a3,a4,a5
60     1015e: fd042783           lw a5,-48(s0)
61     10162: 078a                slli a5,a5,0x2
62     10164: fd843703           ld a4,-40(s0)
63     10168: 97ba                add a5,a5,a4
64     1016a: 85be                mv a1,a5
65     1016c: 8536                mv a0,a3
66     1016e: f0bff0ef           jal ra,10078 <_Z4swapPiS_>
67     10172: fec42783           lw a5,-20(s0)
68     10176: 2785                addiw a5,a5,1
69     10178: 2781                sext.w a5,a5
70     1017a: 853e                mv a0,a5
71     1017c: 70a2                ld ra,40(sp)
72     1017e: 7402                ld s0,32(sp)
73     10180: 6145                addi sp,sp,48
74     10182: 8082                ret
```

Codice 5.9: Risultato dell'objdump della funzione Partition di Quick Sort

Capitolo 6

Responso dell'Industria

Da strumento creato per le università, RISC-V, nonostante sia piuttosto recente, ha guadagnato popolarità anche nelle aziende di alcuni settori, principalmente IoT ed elettronica industriale, attratte dalla natura modulare e senza licenze. In questo capitolo vengono trattati diversi casi di aziende ed istituzioni che hanno deciso di supportare RISC-V.

6.1 Western Digital

Western Digital, uno dei colossi della produzione di dispositivi di memoria, è uno dei principali sostenitori di rilievo di RISC-V: oltre ad essere uno dei membri fondatori della RISC-V Foundation, ha creato, insieme ad Esperanto[1], Google e SiFive[2], la Chips Alliance, pensata per "lavorare in modo collaborativo a core, periferiche e SoC basati su standard aperti, tra cui RISC-V"[23].
Le ragioni dichiarate dal vicepresidente della sezione ricerca Richard New[24] per l'investimento in RISC-V sono tre:

- a breve termine, migliorare i prodotti attuali grazie alle CPU della famiglia SweRV
- a medio termine, combinare memoria e calcolo per nuovi prodotti
- a lungo termine, favorire l'innovazione e l'openness

Oltre a questo, l'azienda è interessata a passare da una esclusiva produttrice di sistemi di archiviazione dei dati ad un'azienda di tecnologia dei dati[25] e quindi

[1] Azienda specializzata in chip RISC-V per l'IA
[2] Principale azienda produttrice di chip RISC-V, fondata da Krste Asanović, Andrew Waterman e Yunsup Lee

investire in settori come il Big Data computing, il Machine Learning e l'Edge Computing.
Ma i componenti delle architetture a scopo generale non scalano in modo indipendente a causa delle licenze e molto spesso hanno dei componenti non necessari che portano ad uno spreco di risorse e soldi. RISC-V, con la sua architettura modulare e aperta, risolve entrambi i problemi, ed è per questo che è stata scelta dall'azienda come architettura di riferimento per i prossimi prodotti.

6.1.1 Famiglia di processori SweRV

Per quanto riguarda gli obiettivi a breve termine, WD ha sviluppato una famiglia di processori chiamata SweRV da includere nei propri prodotti, in particolare per i controller negli SSD, la cui vendita contribuisce a quasi la metà del fatturato dell'azienda.
Il primo chip realizzato, chiamato SweRV EH1, è il più semplice ed implementa la base RV32I con estensioni M per le moltiplicazioni e le divisioni e C per le istruzioni compresse. È un core superscalare a 2 vie[3] con pipeline a 9 livelli, quasi tutti in ordine[26], e include una cache delle istruzioni, delle TCM[4], un interrupt controller, un blocco di debug e 4 bus da 64 bit. La figura 6.1 mostra lo schema del core.
A livello fisico è realizzato dalla TSMC con tecnologia a 28 nm.

Secondo l'azienda, il nuovo chip ha 40% in più di performance, consuma 30% di energia in meno e occupa il 25% in meno di area fisica rispetto al microcontroller di terze parti attualmente utilizzato.
Oltre alla superiorità tecnica dichiarata, l'azienda ha acquisito il pieno controllo del design del chip e quindi la possibilità di modificarlo e migliorarlo secondo le necessità.

Il chip EH2 è simile a EH1, ma con possibilità di operare in dual thread. Per completare la famiglia, sono stati introdotti un chip con pipeline a 4 stadi chiamato EL2 e un chip basato su RV64 chiamato EHX3.

[3]Un processore superscalare può eseguire più di una istruzione per ciclo di clock usando più unità di esecuzione. In questo caso sono due unità di esecuzione

[4]Tightly Coupled Memory, memoria a bassa latenza senza l'imprevidibilità nel tempo d'accesso che è presente nelle cache[27]

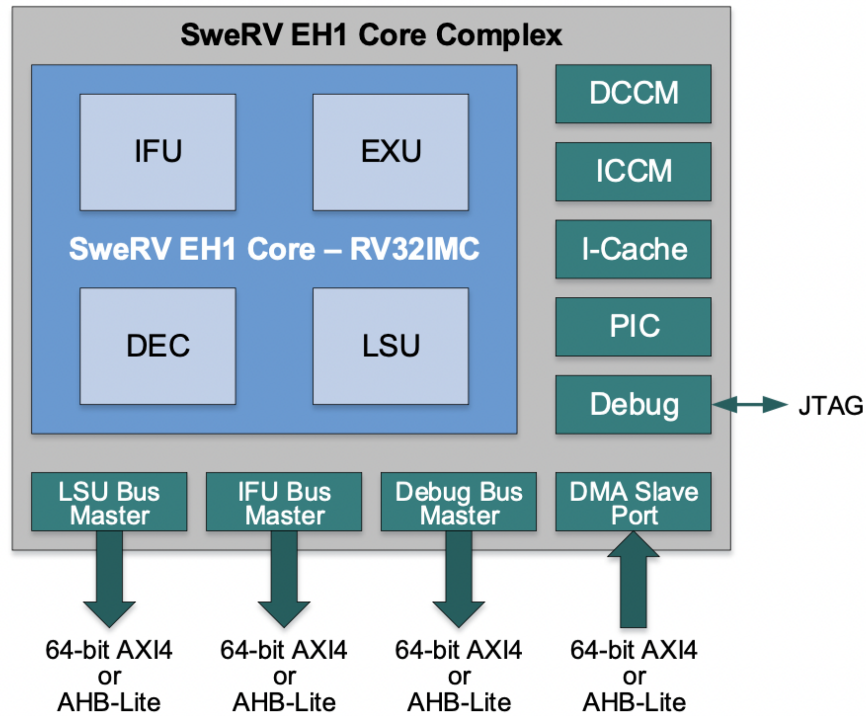


Figura 6.1: Schema del core SweRV EH1

6.1.2 OmniXtend

Il secondo prodotto sviluppato dalla Western Digital con RISC-V è OmniXtend[29], un protocollo di coerenza della cache. Tramite switch Ethernet permette alle cache, ai controller della memoria e agli acceleratori di scambiare messaggi di coerenza direttamente in una struttura compatibile con il protocollo Ethernet.

6.2 Intel

Tramite l'IFS[5] Innovation Fund, Intel ha annunciato il 7 Febbraio 2022 una serie di azioni per supportare RISC-V ed altri standard aperti[30]:

- fondo da 1 miliardo di dollari per supportare start-up ed aziende che lavorano nell'ambito microprocessori
- l'azienda entra a far parte dell'organizzazione RISC-V International
- tramite accordi di partnership, i principali venditori di chip RISC-V (tra cui Andes, Esperanto, SiFive e Ventana) produrranno i loro prodotti nelle fabbriche IFS
- Bob Brunner, vice presidente dell'Ingegneria per le Soluzioni dei Clienti, si unisce al consiglio amministrazione e al comitato tecnico direttivo di RISC-V
- creazione di Accelerator, una alleanza tra diverse aziende nell'ambito microprocessori

Intel ha anche già sviluppato un microprocessore soft[6] per le proprie schede FPGA, chiamato NIOS V[31], basato su RV32IA.
Ma l'annuncio più importante è quello della creazione della piattaforma Open Chiplet.

6.2.1 Open Chiplet

Nel suo famosissimo paper[32] Gordon Moore, oltre a formulare la Legge di Moore, predisse che "potrebbe essere più economico costruire grandi sistemi da funzioni più piccole, che sono separatamente confezionati e interconnessi".
I chiplet sono proprio questo: dei blocchi modulari di un semiconduttore, molto usati attualmente soprattutto nei data center per incorporare acceleratori dentro al chip. Con l'aumentare dei costi al diminuire dei benefici con il design monolitico, l'industria si sta spostando verso l'uso di chip modulari.

[5] Azienda sussidiaria di Intel che si occupa di produrre chip per clienti esterni. Simile a TSMC e Samsung Foundry, con le quali compete

[6] Un microprocessore soft è un chip che può essere implementato interamente usando la sintesi logica

I design modulari hanno diversi vantaggi:

- miglior rendimento e costi
- poter combinare e riusare diversi componenti per creare cpu personalizzate
- ogni blocco può essere creato con tecnologie diverse, quindi pezzi che non hanno la necessità di essere molto veloci possono essere progettati con tecnologie più vecchie e dunque meno costose

Nonostante i vantaggi, i chiplet introducono complessità, sia del chip stesso che della catena di fornitura, ed introducono potenziali costi aggiuntivi a livello energetico e di area occupata.
Tutto questo ha bisogno di una piattaforma di sviluppo dei chiplet aperta , sviluppata insieme ai Cloud Service Provider,in modo da avere poter combinare chiplet di diverse aziende in unico prodotto. Inoltre, insieme a Microsoft, Meta, Google Cloud, Qualcomm, TSMC, Samsung e ASE, è stato creato uno standard di interconnessione aperto tra chiplet chiamato UCIe (Universal Chiplet Interconnect Express)[33], la cui specificazione iniziale è stata donata da Intel.
Nell'Agosto 2022, Intel ha annunciato che le prossime famiglie di processori per PC e laptop, chiamate Meteor Lake e Arrow Lake, useranno il design a chiplet.
In questo mercato in transizione, RISC-V potrebbe ritagliarsi una fetta nei chiplet, soprattutto negli acceleratori[7] o DSA (Domain specific Accelerators).

[7]Gli acceleratori sono unità di calcolo hardware specializzato per specifici compiti come il deep learning, la bioinformatica e l'image processing

6.3 Gli Approcci Europei e Cinesi

Europa

Anche l'Unione Europea ha deciso di utilizzare RISC-V per i propri progetti, in particolare per sviluppare i propri acceleratori hardware nel programma EPI (European Processor Initiative)[34].
L'European Processor Initiative è un progetto finanziato dalla Commissione Europea, tramite il programma EU Horizon 2020, per sviluppare microprocessori a scopo generale per High Performance Computing, acceleratori e piattaforme per il settore automobilistico completamente europei.
Proprio l'indipendenza in ambito High Performance è uno degli obiettivi del programma, insieme a rafforzare la competitività dell'industria e della scienza europea, essendo l'Europa in terza posizione dietro a Stati Uniti e Cina per spese nella ricerca e sviluppo[35].

Come accennato prima, RISC-V viene usato per sviluppare acceleratori ad alta efficienza energetica e potenza chiamati EPAC (European Processor Accelerators) che include:

- RISC-V Vector Tile (RVV), un chiplet per le operazioni sui vettori. Composto da un core a scopo generale a 64 bit chiamato Avispado che implementa RV64GVC e da un'unità di processamento dei vettori chiamata Vitruvius. Molteplici di queste unità, fino a 512, possono essere collegate insieme

- Stencil[8]/Tensor Accelerator Tile (STX), blocco specializzato per Deep Learning e accelerazione Stencil. È composto da quattro unità di computazione, di cui ognuna ha dai 5 ai 17 core RISC-V a 32 bit Snitch, di cui uno è per il trasferimento di dati, e da 0 a 4 SPU (Stencil Processing Unit)

- Variable Precision Tile (VRP), chiplet per eseguire operazioni di algebra lineare iterative con precisione variabile in base alle necessità. Infatti la precisione viene aumentata gradualmente finché non si raggiunge il valore minimo desiderato. È composto da un core a 64 bit con un unità floating-point a precisione variabile, un paio di cache e una serie di registri

[8]operazioni iterative con matrici per risolvere o approsimare diversi algoritmi, per esempio per risolvere equazioni differenziali parziali[36]

Cina

Oltre ad ospitare il RISC-V Summit nel 2022[37], la Cina sta investendo considerevolmente in RISC-V, principalmente per ottenere indipendenza dalle aziende con base occidentale in modo da aggirare sanzioni, che già sono in atto per alcune aziende cinesi come Huawei[38].
Per questo motivo, nel quattordicesimo Piano Quinquennale[39], uno dei pilastri fondamentali è proprio l'industria dei microprocessori, che è cresciuta del 30,8% nel 2020 superando Taiwan[40], e punta a superare Giappone ed Europa nei prossimi anni in numero di chip distribuiti. L'ISA però non proteggerà la Cina completamente dalle sanzioni in quanto gli Stati Uniti controllano gli strumenti EDA[9] e le tecnologie di produzione.

Nonostante ciò, metà dei partner premium della organizzazione RISC-V International sono cinesi ed è stato inaugurata la Beijing Open Source IC Academy per innovare sia in termini di chip che di open standard, a dimostrazione del fatto che la Cina è determinata a puntare su RISC-V per il futuro.

[9]Electronic Design Automation, strumenti software per progettare sistemi elettronici

Conclusioni

Nonostante non sia la prima ISA open source, RISC-V sta guadagnando sempre più mercato grazie alla sua semplicità, alla sua modularità, ma anche ad un periodo storico favorevole, in cui la legge di Moore sta morendo e si sta passando ai chiplet e in cui le nazioni cercano alternative alle tecnologie statunitensi e dei suoi alleati.
Ritagliandosi una sua fetta di mercato sempre in crescita, RISC-V sta unendo l'industria per creare gli standard e gli strumenti necessari e potenzialmente permetterà di avere una varietà di processori mai vista.

Bibliografia

[1] Andrew Waterman e Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA*. Rapp. tecn. 2019 (cit. alle pp. 1, 3, 11, 12, 17, 26, 31).

[2] *What Is RISC?* URL: https://www.arm.com/glossary/risc (cit. a p. 2).

[3] William Stallings. «Computer Organization and Architecture: Designing for Performance». In: Pearson, 2016 (cit. alle pp. 2, 37).

[4] Krste Asanovic e David Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Rapp. tecn. 2014 (cit. a p. 2).

[5] *Tagesregister 10'027 vom 06.03.2020: RISC-V International Association*. URL: https://riscv.org/wp-content/uploads/2020/03/Extract-from-daily-register-RISC_V-International-Association.pdf (cit. a p. 3).

[6] Andrew Waterman, Krste Asanovic e John Hauser. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*. Rapp. tecn. 2021 (cit. alle pp. 8, 27, 31, 32, 34, 37).

[7] Tony Chen e David Patterson. *RISC-V Geneology*. Rapp. tecn. 2016 (cit. a p. 9).

[8] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill e David A. Wood. «A Primer on Memory Consistency and Cache Coherence, Second Edition». In: Morgan& Claypool, 2020. Cap. 3.2 (cit. a p. 11).

[9] David Howells, Paul E. McKenney, Will Deacon e Peter Zijlstra. *LINUX KERNEL MEMORY BARRIERS*. URL: https://www.kernel.org/doc/html/latest/staging/index.html#memory-barriers (cit. alle pp. 11, 12, 16).

[10] Ori Lahav e Viktor Vafeiadis. *Explaining Relaxed Memory Models with Program Transformations*. Rapp. tecn. Max Planck Institute for Software Systems (MPI-SWS) (cit. a p. 12).

[11] Andrew Waterman e Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA*. Rapp. tecn. SiFive Inc. e CS Division, EECS Department, University of California, Berkley, 2019, p. 14 (cit. a p. 19).

[12] Andrew Shell Waterman. «Design of the RISC-V Instruction Set Architecture». University of California, Berkeley, 2016 (cit. alle pp. 20, 21, 25).

[13] *Steam Hardware & Software Survey: July 2022*. 2022. URL: `https://store.steampowered.com/hwsurvey` (cit. a p. 25).

[14] Edward Cunningham. *Improving app security and performance on Google Play for years to come*. 2017. URL: `https://android-developers.googleblog.com/2017/12/improving-app-security-and-performance.html` (cit. a p. 25).

[15] David Whaley. *What 64-Bit Android Apps Mean for the Future of Mobile*. 2020. URL: `https://www.arm.com/blogs/blueprint/android-64bit-future-mobile` (cit. a p. 25).

[16] Andrew Waterman, Krste Asanovic, Palmer Dabbelt, Al Stone e Kumar Sankaran. *RISC-V Platform Specification*. Rapp. tecn. Dic. 2021 (cit. a p. 28).

[17] *UNIX-Class Platform Specification Task Group*. URL: `https://lists.riscv.org/g/tech-unixplatformspec` (cit. a p. 28).

[18] William Stallings. «Computer Organization and Architecture: Designing for Performance». In: Pearson, 2016. Cap. 8.5 (cit. a p. 30).

[19] *Virtualization*. URL: `https://www.ibm.com/cloud/learn/virtualization-a-complete-guide` (cit. a p. 34).

[20] David A. Patterson e John L. Hennessy. *Struttura e progetto dei calcolatori. Progettare con RISC-V*. Trad. da Alberto Borghese. Zanichelli, 2019 (cit. alle pp. 34, 35, 42).

[21] *RISC-V GNU Compiler Toolchain*. URL: `https://github.com/riscv-collab/riscv-gnu-toolchain` (cit. a p. 43).

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009 (cit. a p. 46).

[23] *Chips Alliance*. URL: https://chipsalliance.org (cit. a p. 51).

[24] *Strategic Innovation: RISC-V at Western Digital*. URL: https://www.youtube.com/watch?v=id8vC0q6XDA%5C&t=12s (cit. a p. 51).

[25] Vivek Tyagi. *RISC-V: Enabling a New Era of Open Data-Centric Computing Architectures*. 18 Giu. 2018. URL: https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/service-brief/service-brief-risc-v.pdf (cit. a p. 51).

[26] *RISC-V and Open Source Hardware Address New Compute Requirements*. Rapp. tecn. Western Digital, dic. 2019 (cit. a p. 52).

[27] *Tightly Coupled Memory*. URL: https://developer.arm.com/documentation/den0042/a/Tightly-Coupled-Memory (cit. a p. 52).

[28] *RISC-V SweRVTM EH1 Programmer's Reference Manual*. Rapp. tecn. Western Digital, 24 gen. 2019 (cit. a p. 53).

[29] Marjan Radi, Wesley W. Terpstra, Paul Loewenstein e Dejan Vucinic. *OmniXtend: Direct to Caches over Commodity Fabric* (cit. a p. 53).

[30] Jason Gorss. *Intel Launches $1 Billion Fund to Build a Foundry Innovation Ecosystem*. 7 Feb. 2022. URL: https://www.intc.com/news-events/press-releases/detail/1525/intel-launches-1-billion-fund-to-build-a-foundry (cit. a p. 54).

[31] *Nios® V Processors*. URL: https://www.intel.com/content/www/us/en/products/details/fpga/nios-processor/v.html (cit. a p. 54).

[32] Gordon E. Moore. «Cramming more components onto integrated circuits». In: *Electronics* 38.8 (19 apr. 1965) (cit. a p. 54).

[33] Dr. Debendra Das Sharma. *Universal Chiplet Interconnect Express (UCIe)®: Building an open chiplet ecosystem*. UCIe, 2022 (cit. a p. 55).

[34] *European Processor Initiative*. URL: https://www.european-processor-initiative.eu (cit. a p. 56).

[35] *Research and Development: U.S. Trends and International Comparisons*. National Science Foundation, 28 apr. 2022 (cit. a p. 56).

[36] *AN 870: Stencil Computation Reference Design*. Intel. 10 Ott. 2018 (cit. a p. 56).

[37] *RISC-V Summit China 2022*. URL: `https://riscv-summit-china.com` (cit. a p. 57).

[38] Code of Federal Regulations. *Control Policy: End-User and End-Use Based*. Ago. 2022. URL: `https://ecfr.gov/current/title-15/subtitle-B/chapter-VII/subchapter-C/part-744` (cit. a p. 57).

[39] Xinhua News Agency. «Outline of the People's Republic of China 14th Five-Year Plan for National Economic and Social Development and Long-Range Objectives for 2035». Trad. da Inc. Etcetera Language Group. In: (mar. 2021). A cura di Ben Murphy (cit. a p. 57).

[40] Semiconductor Industry Association. *China's Share of Global Chip Sales Now Surpasses Taiwan's, Closing in on Europe's and Japan's*. Gen. 2022. URL: `https://semiconductors.org/chinas-share-of-global-chip-sales-now-surpasses-taiwan-closing-in-on-europe-and-japan` (cit. a p. 57).