



UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE
Corso di Laurea Informatica

RISC-V stress testing

Relatore: Trentini Andrea

Correlatore: Carraturo Alexjan

Tesi di Laurea di: Bianchessi Mattia
Matr. 931455

Anno Accademico 2021-2022

Todo list

■ Rileggere	13
■ atrent: fonte? - mattia: Questa l'ho fatta io - atrent: ok allora in intro aggiungi una nota in cui avvisi che tutte le immagini sono tue salvo citazione di altra fonte	18
■ aveva una licenza considerabile libera?!? - mattia: non ho trovato nessun riferimento al tipo di licenza - atrent: allora non puoi dire "licenza libera", a meno che nei bibref che citi non ci siano affermazioni che ti supportano, nel caso citale direttamente	19
■ Rileggere	21
■ Rileggere	26
■ mattia: aggiunto alla fine qualche parola di commento(CPI, lun- ghezza, dimensione) - atrent: ok	36
■ atrent: conclusione minimale, riesci a dire se ci sono aspetti in cui uno è meglio dell'altro e viceversa? comparazioni di costo? corrente consumata? ecc. - mattia: ok - atrent: meglio	51
■ atrent: nelle conclusioni dovresti a questo punto (a valle del tuo lavoro di analisi) poter dire non solo come si confronta risc-v con gli altri, ma anche quali sono le aree di debolezza e forza, tanto da ipotizzare quale potrebbero essere i contesti d'uso favorevoli - mattia: ok? - atrent: forse alcune affermazioni (es. machine learning) andrebbero supportate con qualche bibref	51

Indice

1 RISC-V	8
1.1 Gli obiettivi di RISC-V	8
1.2 Panoramica	9
1.3 Descrizione board	10
1.4 Ambiente di sviluppo	10
2 ISA RISC-V	13
2.1 Base	13
2.2 Estensioni	13
2.3 Istruzioni base	14
2.4 RV32I	16
2.5 Le altre basi	17
3 Compilatori	18
3.1 Descrizione	18
3.2 Storia	18
3.3 Cross-compilazione	19
3.4 Compilatori moderni	19
3.5 Un buon compilatore	19
3.6 Toolchain di RISC-V	20
4 BenchMarking	21
4.1 Storia	22
4.2 Benchmark standard	23
4.2.1 CoreMark	23
4.2.2 LINPACK	24
4.3 Altri programmi	24
4.3.1 MultOrShift	24
4.3.2 Sorting	25

5 Comparativa	26
5.1 CoreMark	26
5.2 LINPACK	27
5.3 MultOrShift	28
5.4 Analisi codice assembly	30
5.4.1 Addizione con costante	30
5.4.2 Addizione	31
5.4.3 Moltiplicazione	32
5.4.4 Divisione	35
5.4.5 Confronto ISA	35
5.5 Sorting	37
5.5.1 Bubble sort	37
5.5.2 Insertion sort	39
5.5.3 QuickSort	40
5.5.4 Heap sort	41
5.5.5 Confronto Sorting	42
6 Conclusione	50

Elenco delle figure

1.1	Board vista dall'alto	12
1.2	Schema a blocchi della scheda di sviluppo	12
2.1	Formati istruzione RISC-V[9]	16
2.2	Convenzione dei registri	17
3.1	Schema meccanismo di compilazione	18
5.1	Vista superiore Raspberry model B	27
5.2	Funzione di somma	31
5.3	Funzione di somma	32
5.4	Moltiplicazione per 2	33
5.5	Moltiplicazione per 8	33
5.7	Moltiplicazione per 31	35
5.8	Moltiplicazione per 30	35
5.9	Algoritmi di ordinamento a confronto	42
5.10	Tempi di esecuzione Bubblesort PC e RISC-V	43
5.11	Tempi di esecuzione Insertionsort PC e RISC-V	44
5.12	Tempi di esecuzione Quicksort PC e RISC-V	45
5.13	Tempi di esecuzione Heapsort PC e RISC-V	45
5.14	Tempi di esecuzione Bubblesort Raspberry e RISC-V	46
5.15	Tempi di esecuzione Insertionsort Raspberry e RISC-V	46
5.16	Tempi di esecuzione Quicksort Raspberry e RISC-V	47
5.17	Tempi di esecuzione Heapsort Raspberry e RISC-V	47
5.18	Tempi di esecuzione Bubblesort	48
5.19	Tempi di esecuzione Insertionsort	48
5.20	Tempi di esecuzione Quicksort	49
5.21	Tempi di esecuzione Heapsort	49

Elenco delle tabelle

1.1	Caratteristiche della board	11
2.1	Tabella nomenclatura ISA RISC-V	15
5.1	Score di CoreMark ad ogni esecuzione	27
5.2	Score medio e deviazione standard di CoreMark	27
5.3	MFLOPS LINPACK	28
5.4	Tempo di risoluzione LINPACK	29
5.5	Tempi di esecuzione operazioni calcolati in ms	30
5.6	CPI per RISC-V	36
5.7	CPI per ARM	36
5.8	CPI a confronto	37
5.9	Code size in byte [B]	37
5.10	complessità bubble sort	38
5.11	Tempi di esecuzione bubble sort in ms	38
5.12	complessità insertion sort	39
5.13	Tempi di esecuzione insertion sort in ms	39
5.14	Tempi di esecuzione quick sort in ms	41
5.15	Tempi di esecuzione heap sort in ms	42
5.16	Tempi di esecuzione degli algoritmi di sorting su PC in ms	43
5.17	Tempi di esecuzione Raspberry Pi B in ms	44
5.18	Rapporto prestazioni ARM e RISC-V	44

Listings

5.1	compilazione CoreMark	26
5.2	compilazione LINPACK	28
5.3	Impostazioni dei dati	28
5.4	Moltiplicazione	29
5.5	Shift	29
5.6	Addizione	30
5.10	Addizione generale	31
5.14	Moltiplicazione per potenza di 2	33
5.29	Algoritmo bubble sort scritto in c	38
5.30	Algoritmo bubble sort scritto in c	39
5.31	Algoritmo quick sort scritto in c	40
5.32	Algoritmo heap sort scritto in c	41

Introduzione

Nel panorama delle architetture moderne x86 e ARM la fanno da padrone, il primo nel settore computing e il secondo per embedded. In questo contesto sta emergendo una nuova architettura RISC-V.

La tesi si occupa di presentare il progetto RISC-V e la sua ISA per poi eseguire dei benchmark e altri programmi con cui valutarne le caratteristiche.

Per la valutazione sono stati utilizzati due benchmark ufficiali (Coremark, LINPACK) e altri codici in linguaggio C compilati con la *toolchain* di RISC-V. I risultati dei test vengono riportati e, in alcuni casi, comparati con i risultati di programmi simili compilati per altri processori. Viene poi analizzato il codice sorgente di RISC-V e confrontato con il codice sorgente di ARM.

La tesi è strutturata nel modo seguente¹:

- **Capitolo 1:** Presenta il progetto RISC-V presentando gli obiettivi e la storia del progetto.
- **Capitolo 2:** Contiene una sintesi dell'ISA di RISC-V evidenziando alcuni punti importanti.
- **Capitolo 3:** Descrive la *toolchain* utilizzata per la compilazione dei programmi. Il capitolo inizia con la presentazione dei compilatori e della loro storia fino ad arrivare a parlare della *toolchain*.
- **Capitolo 4:** Contiene una descrizione dei programmi utilizzati. Il capitolo tratta la storia dei benchmark e, successivamente, descrive i benchmark utilizzati e gli altri programmi.
- **Capitolo 5:** Contiene i dati raccolti durante l'esecuzione dei programmi scelti confrontandoli e analizzandoli.
- **Capitolo 6:** Conclusione della tesi.

¹Le immagini utilizzate sono state autoprodotte salvo citazioni di altre fonti

Capitolo 1

RISC-V

RISC-V è una ISA basato sul principio RISC nato come progetto accademico. Nel 2010, a Berkeley(California), il progetto iniziò diretto dal prof. David Patterson finanziato da *Intel* e *Microsoft* e da alcune aziende Californiane. La prima pubblicazione è dell'anno successivo. Nel 2015 viene fondata **RISC-V Foundation**, un'azienda no-profit che controlla lo sviluppo di RISC-V [10]. Nel 2018 viene annunciata una collaborazione tra l'azienda e *Linux Foundation* con la quale si supporta lo sviluppo del progetto RISC-V [21].

1.1 Gli obiettivi di RISC-V

Durante la fase di progettazione, i progettisti hanno voluto mettere nero su bianco lo scopo di RISC-V. Gli obiettivi dichiarati nell'introduzione della specifica dell'ISA user-mode sono:

- Un ISA con una licenza *open source* disponibile per accademia e industria.
- Un ISA adatta ad un'implementazione hardware diretta, non una simulazione.
- Un ISA non specifica per una micro-architettura o una tecnologia ma che permetta un'implementazione efficiente.
- Un ISA modulare, organizzata in ISA più piccole con la possibilità di usare estensioni¹.

¹Insiemi di istruzioni opzionali che possono essere aggiunte alla base dell'ISA di RISC-V per fornire funzionalità specifiche. Sezione 2.2.

- Supporto per lo standard floating-point 2008 IEEE-754 ².
- Supporto delle estensioni.
- Spazio di indirizzamento 32 e 64 bit.
- Supporto a delle implementazioni multi core e manycore ³ sia eterogenei che omogenei.
- istruzioni a lunghezza variabile.
- un ISA completamente virtualizzabile.
- un ISA che permetta la semplificazione degli esperimenti con nuovi progetti ISA con *supervisor-level* e *hypervisor-level*.

1.2 Panoramica

L'ISA RISC-V è un architettura load-store con solo 49 istruzioni base. L'ISA supporta sistemi di memoria sia *little-endian* che *big-endian*. Le istruzioni sono organizzate in pacchetti da 16-bit memorizzati in maniera *little-endian* indipendentemente dall'*endianness* del sistema. Ogni pacchetto ha nei bit meno significativi i bit per la codifica delle istruzioni. In questo le istruzioni di lunghezza variabile sono decodificate velocemente.

Le celle della memoria principale sono di lunghezza variabile a seconda dell'ISA base scelto, il numero di celle di memoria è XLEN e la dimensione della singola cella è di 2^{XLEN-1} . Ad esempio l'ISA RV32I ha un XLEN di 32. Lo spazio di indirizzamento è circolare quindi l'errore di overflow non c'è in quanto i calcoli degli indirizzi vengono scalati in modo adeguato dividendosi per un modulo adatto sfruttando così la caratteristica circolare [9].

Avendo tutte le istruzioni di lunghezza fissa, non è possibile avere direttamente costanti o indirizzi superiori alla lunghezza assegnata al campo dell'istruzione. La soluzione è l'utilizzo della modalità di indirizzamento:

- **indirizzamento immediato:** l'operando è una costante nell'istruzione, è comunque limitato dai bit assegnati del campo.
- **indirizzamento a registro:** l'operando è un registro.

²<https://ieeexplore.ieee.org/document/4610935> [7]

³I processori Manycore sono un tipo speciale di processori multi-core progettati per un alto grado di elaborazione parallela. Contengono numerose core più semplici e indipendenti.

- **indirizzamento di base con spostamento:** l'operando è la somma tra il contenuto di un registro e una costante.
- **indirizzamento relativo al PC:** l'operando dipende dal *Program Counter* che viene sommato a una costante.

Si parla di eccezione quando una condizione non comune avviene a *run time* associata a un'istruzione. Si parla di *trap* quando si verifica un trasferimento di controllo da parte del *trap handler*, da un *thread* ad un altro. Si parla di *interrupt* quando la situazione imprevista è qualcosa di esterno.

1.3 Descrizione board

La scheda di sviluppo utilizzata è D1-H Nezha basata sul design del chip All-winner D1-H [26]. La board integra una CPU Ali Pingtou Ge RISC-V C906, con clock a 1 GHz, supporta il kernel Linux standard, ha la memoria 2G DDR3, 258 MB di spin-nand, connessione WiFi/Bluetooth, con interfacce audio e video, può essere collegata a varie periferiche, MIPI-DSI+TP, supporta il collegamento con scheda SD, HDMI, auricolari da 3,5 mm, Gigabit Ethernet, USB,USB-C, UART, 40 pin.

1.4 Ambiente di sviluppo

La scheda di sviluppo D1-H viene fornita con il sistema Tina Linux. Il kernel fornito è adattato al kernel Linux 5.4. La board fornisce il supporto di base e gestione delle risorse hardware del dispositivo. Ulteriori informazioni sono disponibili sul sito della board ⁴.

⁴Link al sito della board in esame https://d1.docs.aw-ol.com/study/study_1tina/

CPU	Allwinner D1-H
Clock	1GHz
DRAM	DDR3 2GB
Memoria	256MB spin-nand integrato
Supporto memoria	USB e SD
Rete	Gigabit Ethernet, 2.4G WiFi e Bluetooth, antenna integrata
Display	MIPI-DSI + TP, HDMI,SPI
Audio	jack per cuffie da 3,5 mm
Tasti	FEL, LRADC OK
Luci	alimentazione, LED tricolore
DEBUG	UART, USB ADB
USB	USB, USB OTG, USB2.0
PIN	array di pin 40
Alimentazione	USB-C 5V-2A
Dimensioni	85 x 56 x 1,7 mm

Tabella 1.1: Caratteristiche della board

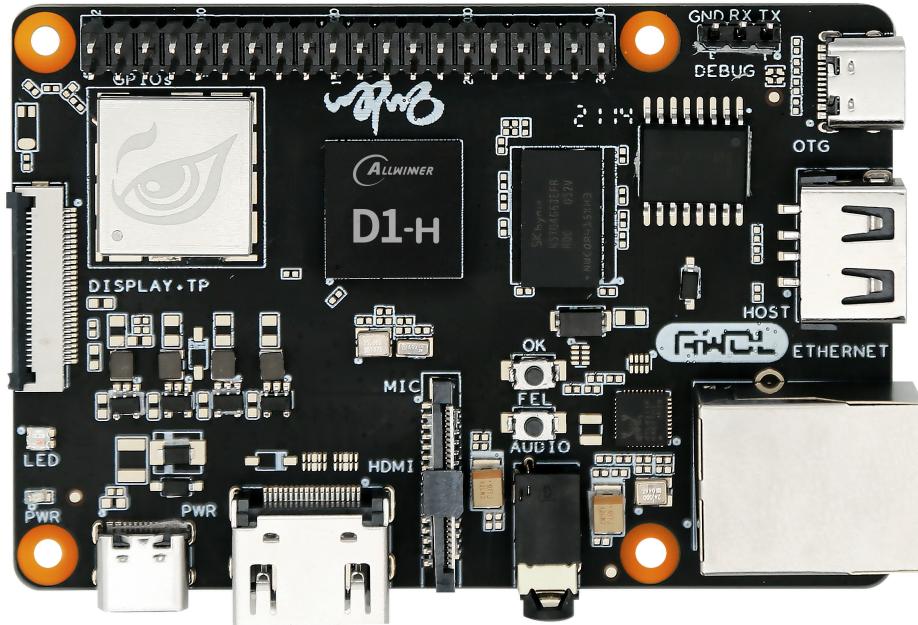


Figura 1.1: Board vista dall'alto

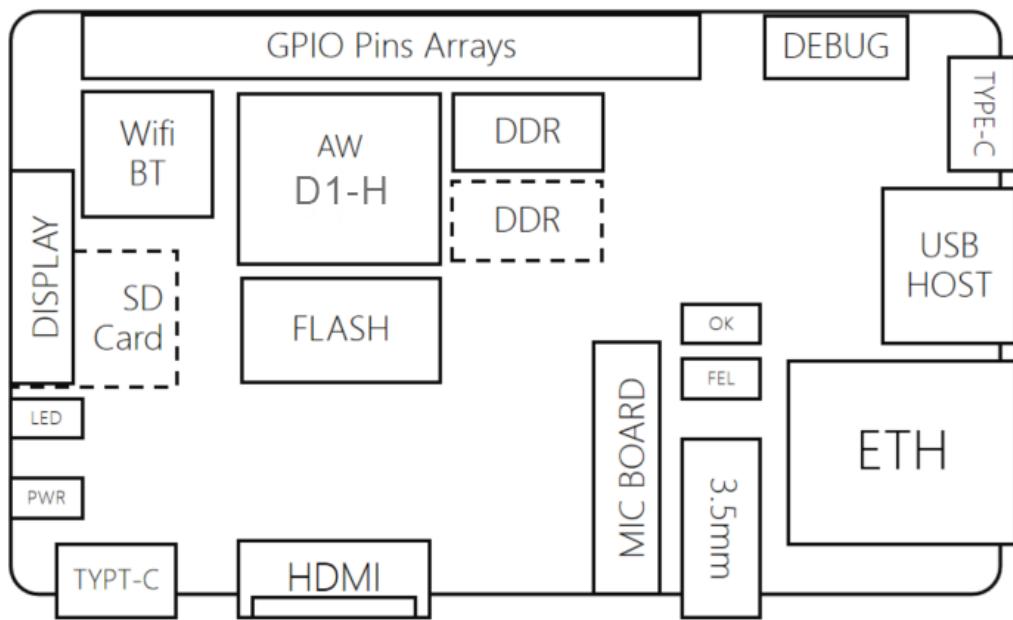


Figura 1.2: Schema a blocchi della scheda di sviluppo
[14]

Capitolo 2

ISA RISC-V

Rileggere

Prima di discutere dei risultati ottenuti presentiamo L'*Instruction Set Architecture* o **ISA**.

2.1 Base

RISC-V prevede un nucleo di istruzioni di base mediante le quali si può supportare dei sistemi funzionanti. Esistono diversi nuclei base denominati a seconda di quanti bit utilizza il sistema. Esistono 4 basi:

- RV32I, che ha lo spazio di indirizzamento di 32 bit.
- RV64I, che ha lo spazio di indirizzamento di 64 bit.
- RV128I, che ha lo spazio di indirizzamento di 128 bit.
- RV32E, sotto-insieme di RV32I che ne offre un supporto simile ed è pensata per dispositivi *embedded*.

Tutte le ISA base usano il complemento a due per la rappresentazione di valori interi con segno. Le basi per la computazione a valori interi è identificata dalla lettera "I".

2.2 Estensioni

Se si utilizza solo una base si hanno solo delle funzionalità basilari, per questo motivo vengono introdotte le estensioni. Le estensioni dell'ISA di RISC-V sono insiemi di istruzioni opzionali che possono essere aggiunte alla base dell'ISA di RISC-V per fornire funzionalità specifiche e personalizzate. L'ISA base di RISC-V è progettata per essere estensibile in modo modulare, il

che significa che le estensioni possono essere aggiunte senza influire sulla compatibilità con il software esistente.

Usare un'estensione significa estendere le funzionalità e aggiungere il supporto per determinate azioni. Di seguito vengono riportate alcune estensioni:

- ”M”, aggiunge istruzioni per le operazioni di moltiplicazioni e divisioni di interi.
- ”A” istruzioni atomiche, istruzioni di lettura-scrittura-modifica atomiche.
- ”F” istruzioni a virgola mobile a singola precisione, aggiungendo anche registri.
- ”D” istruzioni a virgola mobile a doppia precisione.
- ”C” istruzioni a 16 bit.

Le estensioni, possono essere di tre tipi in base alla standardizzazione:

- **standard** definite dalla RISC-V Foundation, come le sopra citate.
- **reserved** non ancora definite ma riservate per usi futuri.¹
- **non standard** non ancora standardizzate o ampiamente adottate.

La tabella 2.1, presa da SPEC-2.2 [9], presenta i set base e le estensioni standard con le rispettive versioni. Ogni base o estensione presenta la casella ”Definitiva” che specifica se il set o l'estensione è definitiva (S) o non lo è (N).

2.3 Istruzioni base

Di base l'ISA presenta un piccolo insieme di istruzioni. Le istruzioni base sono solamente 47 e vengono codificate in 6 formati (R/I/S/U/B/J).

In tutti i formati i registri sorgente ($rs1$ e $rs2$) e il registro destinazione (rd) vengono mantenuti nelle stesse posizioni per velocizzare la codifica. L'ISA presenta 4 categorie di istruzioni:

¹utilizzato dalle aziende per la progettazione di processori personalizzati e spesso aggiungono estensioni proprietarie per soddisfare esigenze specifiche del loro settore o dei loro clienti.

Base	Versione	Definitiva?
RV32I	2.0	S
RV32E	1.9	N
RV64I	2.0	S
RV128I	1.7	N
Estensione	Versione	Definitiva?
M	2.0	S
A	2.0	S
F	2.0	S
D	2.0	S
Q	2.0	S
L	0.0	N
C	2.0	S
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.2	N
N	1.1	N

Tabella 2.1: Tabella nomenclatura ISA RISC-V

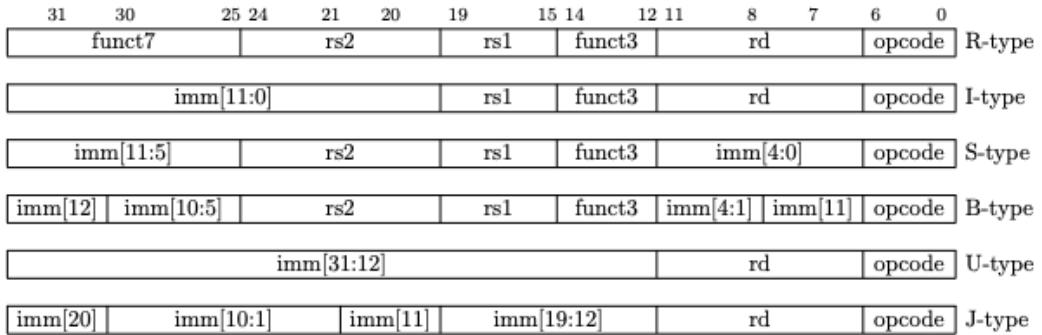


Figura 2.1: Formati istruzione RISC-V[9]

- **Istruzioni computazionali:** Sono presenti 21 istruzioni computazionali e vengono codificate nel formato R se è un'operazione tra i registri o nel formato I se è un'operazione tra registro e costante. Le istruzioni di questo tipo includono istruzioni aritmetiche, logiche e di comparazione sia per valore senza segno che per i valori con segno.
- **Accesso alla memoria:** Le istruzioni di accesso alla memoria permettono il trasferimento di dati dalla memoria e alla memoria. Sono presenti 8 istruzioni in totale, 5 di load codificate nel formato I e 3 di store codificate nel formato S.
- **Controllo del flusso:** Le istruzioni di controllo permettono di alterare il normale flusso sequenziale del programma. Sono presenti 6 istruzioni di questo tipo che permettono il trasferimento codificate nel formato B. Le istruzioni prevedono il confronto degli operandi in *rs1* e *rs2* e se la condizione è verificata viene aggiunto il valore del campo imm al *program counter* per raggiungere l'indirizzo di arrivo.
- **Istruzioni di sistema:** Con RV32I sono presenti 8 istruzioni di controllo del sistema. Possiamo dividerle in due gruppi. Il primo gruppo (ECALL, EBREAK) gestisce le *system call*. Il secondo gruppo è utilizzato per leggere e scrivere i registri di stato.

2.4 RV32I

L'ISA base è stata progettata per supportare i moderni sistemi operativi. Questa base contiene 47 istruzioni uniche. I 32 registri sono di 32 bit (XLEN = 32) e vengono identificati da x seguito da un numero. Il primo registro x0

contiene la costante zero e ogni istruzione che cerca di modificarlo solleva una eccezione. Gli altri registri x1 - x31 sono *general purpose*. L'ABI definisce le convenzioni di utilizzo dei registri(Figura 2.2).

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

Figura 2.2: Convenzione dei registri

Come mostrato in Figura 2.2 notiamo che non è presente un registro dedicato allo *stack pointer* nè un *return adress* ma vengono utilizzati rispettivamente il registro x2 e il registro x1.

2.5 Le altre basi

Esistono altri set che aumentano i bit utilizzati dai registri come RV64I e RV128I.

Capitolo 3

Compilatori

3.1 Descrizione

Un compilatore è un programma che trasforma il codice sorgente in linguaggio macchina. Il processo di compilazione prevede diverse fasi. La prima fase prevede un'analisi lessicale da cui vengono generati dei token. La seconda, utilizzando i token, prevede un'analisi sintattica e infine un'analisi semantica dopo la quale viene generato il codice intermedio. Questo codice intermedio attraversa una fase di ottimizzazione e infine viene generato il codice target [18] [20].

Qualsiasi programma scritto in un linguaggio di programmazione di alto livello deve essere tradotto in codice oggetto prima di poter essere eseguito. Tutti i programmatore che utilizzano tale linguaggio utilizzano un compilatore o un interprete. I miglioramenti a un compilatore possono portare a un gran numero di funzionalità migliorate nei programmi eseguibili.

atrent: fonte? - mattia: Questa l'ho fatta io - atrent: ok allora in intro aggiungi una nota in cui avvisi che tutte le immagini sono tue salvo citazione di altra fonte

3.2 Storia

Il primo compilatore teorico fu pensato da Corrado Böhm, nel 1951, lo sviluppò per la sua tesi di dottorato. Una prima implementazione di un compilatore è dovuta a Grace Hopper che ha anche coniato il termine “compilatore”.

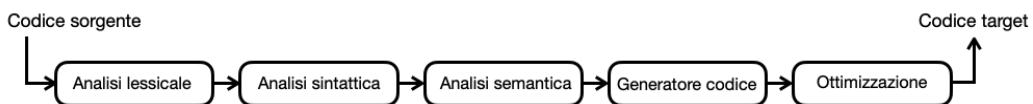


Figura 3.1: Schema meccanismo di compilazione

Il programma, chiamato *A-O System*[17], funzionava come caricatore o linker, non come i moderni compilatori. È importante menzionare che il codice sorgente della versione A-2, datata 1953, fu rilasciato ai clienti con lo scopo di sviluppare dei miglioramenti.

3.3 Cross-compilazione

La cross-compilazione è il processo di compilazione del codice sorgente su un sistema diverso da quello in cui il codice verrà eseguito. In altre parole, il compilatore viene eseguito su una piattaforma diversa rispetto a quella di destinazione del software compilato. La cross-compilazione è spesso utilizzata nel contesto dello sviluppo di software per dispositivi embedded dove lo sviluppatore scrive il codice sorgente su un computer host più potente e poi lo compila per l'hardware a cui è destinato.

aveva una licenza considerabile libera?!? - mattia: non ho trovato nessun riferimento al tipo di licenza - atrent: allora non puoi dire "licenza libera", a meno che nei libref che citi non ci siano affermazioni che ti supportano, nel caso citale direttamente

3.4 Compilatori moderni

Nell'ecosistema dei compilatori moderni i due più famosi sono: GCC e CLANG. Il primo GCC (GNU Compiler Collection) fu creato nel 1987 ed oggi viene sviluppato da programmatore di tutto il mondo. Inizialmente nato per il linguaggio C oggi supporta altri linguaggi come Java, C++, Objective C [5].

Il secondo è nato nel 2005 e sviluppato da Apple Inc. con lo scopo di avere un compilatore Apple ottimizzato per i loro dispositivi[22].

3.5 Un buon compilatore

La scelta di un compilatore per un progetto è una scelta importante. I processori, oggi, sono strutturati in pipeline superscalari e in altre complesse strutture interne. Inoltre i linguaggi moderni astraggono dalla struttura hardware per ottenere un linguaggio logico più generale. Quindi si predilige un approccio meno specifico e non incentrato sulla struttura della macchina. Gli standard dei linguaggi, si fanno sempre più espressivi e astratti. Questa espressività dei linguaggi aumenta l'onere dei compilatori che devono essere in grado di generare un buon codice assembly. La selezione di un compilatore è una scelta cruciale per il proprio progetto. Si deve tener conto che la stessa porzione di codice, utilizzando compilatori differenti, può generare comandi assembly più o meno efficienti. Oltre a generare programmi eseguibili ad alte prestazioni, i compilatori devono anche avere prestazioni elevate. Un progetto software di grandi dimensioni può contenere da centinaia a migliaia

di singole unità di traduzione. Le unità di traduzione sono parti del codice sorgente che vengono tradotte dal compilatore in un file oggetto o in un file eseguibile. Ogni unità di traduzione può contenere migliaia di righe di codice. Oltre all'efficienza delle prestazioni del codice generato si deve tener conto anche del tempo in cui si genera il codice.

Quindi, un buon compilatore permette di concentrarci sul processo di programmazione piuttosto che farci preoccupare della struttura del sistema e deve essere in grado di produrre un codice che abbia delle buone performance in tempo relativamente breve.

3.6 Toolchain di RISC-V

In sistemi privi di un compilatore viene utilizzata la tecnica della **cross-compilazione**(Sezione 3.3). RISC-V mette a disposizione alcune *toolchain* di compilazione. Una *toolchain* è un insieme di programmi utilizzati sequenzialmente per la creazione di un software. In generale i programmi presenti sono un compilatore, un linker, un assembler, un debugger, un profiler, degli strumenti di analisi del codice e strumenti di gestione del progetto. Per RISC-V sono disponibili le *toolchain gcc*[15] e *toolchain clang*[13]. Entrambe le *toolchain* sono mantenute dalle community. Quella più aggiornata e utilizzata è quella del progetto GNU. Sul sito ¹ sono elencate tutte le *toolchain* disponibili e il sito offre una panoramica su tutto l'ecosistema RISC-V.

GCC toolchain

La toolchain GCC di RISC-V supporta linguaggi come C, C++, Objective-C, e Go. La versione corrente è la 12.2 (datata Agosto 2022). La toolchain ha aggiornamenti molto di frequente. Ci sono quattro manutentori Andrew Waterman, Jim Wilson, Kito Cheng, Palmer Dabbelt .

CLANG toolchain

La toolchain LLVM/CLANG è sviluppata da *lowRISC project* e attualmente è alla versione 15.0.2. A differenza della *toolchain gcc* è mantenuta solamente da Alex Bradbury.

¹<https://wiki.riscv.org/display/HOME/RISC-V+Software+Ecosystem>

Capitolo 4

BenchMarking

Rileggere

Quando si parla di benchmark si intendono dei test di prova con lo scopo di fornire una valutazione delle prestazioni di un computer. In generale esistono due principali categorie di benchmark: sintetici o applicativi. I primi hanno lo scopo di misurare le prestazioni del sistema riguardo specifiche operazioni mentre gli ultimi si riferiscono a software applicativi. Come affermato da Wei Dai e Daniel Berleant in *Benchmarking Contemporary Deep Learning Hardware and Frameworks: a Survey of Qualitative Metrics* , [11] ci sono sette caratteristiche fondamentali che un benchmark deve avere:

1. Rilevanza: i benchmark dovrebbero misurare caratteristiche importanti.
2. Rappresentatività: le metriche delle prestazioni dovrebbero essere accettate dall'industria e dal mondo accademico.
3. Equità: tutti i sistemi dovrebbero essere paragonati in modo equo.
4. Ripetibilità: è possibile verificare i risultati dei benchmark.
5. Rapporto costo-efficacia: i test di benchmark sono economici.
6. Scalabilità: i test di benchmark dovrebbero funzionare sia su sistemi che possiedono poche risorse che numerose risorse.
7. Trasparenza: le metriche utilizzate dovrebbero essere facili da capire.

In questa sezione viene prima presentata la storia dei programmi di benchmark dalle origini fino a oggi presentando alcuni standard attuali dei benchmark. Successivamente vengono approfonditi alcuni benchmark utilizzati.

4.1 Storia

Il primo benchmark citato in letteratura prende il nome di Whetstone. I suoi autori, H.J. Curnow e B.A. Wichmann, svilupparono un programma con lo scopo di valutare le prestazioni in virgola mobile. Pubblicato nel 1976 questo benchmark fu il primo esempio di benchmark sintetico. La sua semplicità è data dalla sua organizzazione in moduli, ognuno specializzato per un aspetto. Un altro benchmark è LINPACK che risale al 1979 (non pensato per diventare un benchmark). Questo benchmark utilizza una libreria *BLAS* (*Basic Linear Algebra Subprograms*) per eseguire operazioni su vettori e su matrici per risolvere sistemi lineari. Lo scopo originale era quello di valutare le prestazioni in virgola mobile dei supercomputer dell'epoca ma poi fu utilizzato anche su architetture meno potenti [33]. Un altro benchmark simile è il famoso Dhrystone. Questo benchmark sintetico pubblicato nel 1984 ha lo scopo di valutare le prestazioni dei microprocessori e dei sistemi di elaborazione dati [3]. Tutti i programmi sviluppati prima dell'introduzione di Dhrystone erano pensati e scritti per valutare un unico aspetto. Il primo tentativo di avere un programma che valutasse più aspetti ci fu nel 1986. Il benchmark si chiama Livermore loops. Questo benchmark è composto da ventiquattro loop interni che valutano aspetti differenti. Dopo lo sviluppo di un programma *general purpose*, si iniziò a pensare a delle suite di test che permettessero l'esecuzione solo di alcuni test per valutare determinati aspetti. Una delle prime raccolte di test degna di nota è la raccolta prodotta dalla *Stanford University* e *the University of California* quando in corrispondenza con la progettazione del primo sistema RISC furono raccolti piccoli programmi. Tra questi erano presenti programmi di: permutazione, risoluzione di problemi come la torre di Hanoi, le otto regine, puzzle e altri tipo moltiplicazione di matrici, quicksort, bubblesort, treesort, moltiplicazione di matrici con valori in virgola mobile e FFT (*Fast Fourier Transform*). L'importanza di questo *Stanford Small Programs Benchmark Set* è dovuta al primo confronto tra architetture RISC e CISC [1]. Nel 1988 fu fondato lo SPEC una cooperativa con lo scopo di produrre benchmark validi [41]. I programmi utilizzati fino ad ora erano di piccole dimensioni e, con il miglioramento delle tecnologie, questi programmi non potevano valutare le nuove architetture. Così fu fondato lo SPEC (*The Standard Performance Evaluation Corporation*) il cui scopo era ed è quello di mantenere delle suite di test standardizzate per le nuove generazioni di computer. Come nei sistemi comuni sono state sviluppate delle suite di test, così anche nel mondo embedded. Nel 1997 fu fondata EEMBC (*l'Embedded Microprocessor Benchmark Consortium*) che, negli anni 2000, pubblicò una prima suite di test dedicati al mondo embedded [29]. Uno dei suoi prodotti più famosi è il benchmark chiamato CoreMark pubblicato

nel 2009. Sempre nel mondo embedded fu creata la suite di test Embench, nel 2019 [30]. Questa suite di test è stata sviluppata da altri progetti come *The Bristol Embecosm Embedded Benchmark Suite (BEEBS)*, *MiBench*, *the WCET benchmark collection*, *DSPstone*, *the Simple Generic Library* e *the Nettle low-level cryptographic library*[35][6][27] [42] [34].

Oggi tra gli standard di settore c'è il precedentemente citato SPEC, in particolare SPECint e SPECfp, il primo incentrato su prestazioni di calcolo con numeri interi e il secondo in virgola mobile. Per quanto riguarda il mondo embedded, EEMBC è uno standard riconosciuto che, oltre a CoreMark, ha sviluppato altri benchmark mirati per dispositivi specifici (AutoBench per dispositivi automotive e industriali, Netwoking2.0 per router e switch e altri). Altri standard di settore sono TPC (*Transaction Processing Performance Council*) che si occupa di creare dei benchmark per i DBMS[40].

Oltre agli standard di settore, con gli anni, si sono sviluppati dei benchmark e delle suite di test open source. Un esempio *phoronix test suite* [37] che supporta Linux, Windows Osx, e altri sistemi. Questa suite di test dà la possibilità di eseguire dei benchmark in maniera semplice (con l'unica richiesta di avere php da linea di comando) e i risultati possono essere caricati su OpenBenchmark [36].

4.2 Benchmark standard

4.2.1 CoreMark

Per citare il sito ufficiale:

CoreMark is a simple, yet sophisticated benchmark that is designed specifically to test the functionality of a processor core. Running CoreMark produces a single-number score allowing users to make quick comparisons between processors.

CoreMark è un benchmark che misura le prestazioni dei microcontrollori e delle CPU utilizzate nei sistemi embedded. In CoreMark sono implementati algoritmi di list processing, manipolazione di matrici, macchina a stati e CRC (controllo di ridondanza ciclico) . È progettato per funzionare su dispositivi da 8 bit a 64 bit [28].

CoreMark è un benchmark sintetico come Dhrystone e, come Dhrystone, CoreMark è piccolo, portatile, gratuito¹ ma, a differenza di Dhrystone, CoreMark ha regole di esecuzione e reporting specifiche ed è stato progettato

¹La licenza di CoreMark è la "CoreMark License Agreement". La CoreMark License Agreement stabilisce che l'utente può utilizzare il benchmark CoreMark solo per scopi di valutazione delle prestazioni dei microprocessori e non per fini commerciali.

per evitare aspetti problematici di Dhrystone. Mentre Dhrystone risulta un benchmark del compilatore, CoreMark si focalizza sulle capacità di lavoro della MCU o di una CPU [2]. Il codice sorgente è disponibile nel repository Github ².

4.2.2 LINPACK

Il LINPACK Benchmark è una misura della velocità di esecuzione in virgola mobile di un computer. Viene determinato eseguendo un programma che risolve un denso sistema di equazioni lineari. Durante gli anni si sono sviluppate tre versioni di questo programma.

La prima versione viene chiamata LINPACK 100 ed è molto simile alla versione del 1979. La soluzione si ottiene attraverso $\frac{2}{3n^3} + 2n^2$ operazioni in virgola mobile con $n = 100$. La seconda versione si chiama LINPACK 1000 e differentemente risolve problemi con $n = 1000$. L'ultima versione chiamata HPL (*High-Performance Linpack*) è dedicata ad una implementazione parallela di LINPACK[4] [32].

4.3 Altri programmi

In questa sezione vengono presentati due programmi scritti da me che non sono standard di benchmarking. Il primo programma viene utilizzato per analizzare il codice assembly generato dal compilatore e per confrontare le architetture RISC-V e ARM, mentre con il secondo vengono utilizzati degli algoritmi di sorting e comparato il tempo di esecuzione con gli stessi algoritmi eseguiti sul Raspberry.

4.3.1 MultOrShift

Il programma confronta la velocità di esecuzione di moltiplicazione e divisione aritmetiche con lo shift logico. Il programma genera un array di interi di valori tra zero e maxInt e un array di potenze di due comprese tra due e millesettantaquattro. Entrambi gli array hanno mille elementi. Dopo la generazione viene calcolato il risultato elemento per elemento e viene memorizzato il tempo di esecuzione. Il risultato del programma è il tempo medio di esecuzione di mille operazioni per tipo (moltiplicazione, divisione, moltiplicazione tramite shift, divisione tramite shift). Lo scopo di questo programma è osservare la differenza delle implementazioni delle operazioni a livello assembly

²<https://github.com/eembc/coremark> [38]

compilato con la *toolchain* di RISC-V e confrontarle con le stesse operazioni compilate per ARM.

4.3.2 Sorting

Vengono utilizzati degli algoritmi di *sorting* per confrontare il tempo di esecuzione sulla board con gli stessi algoritmi su Raspberry. Gli algoritmi di sorting utilizzati sono:

- BubbleSort
- InsertionSort
- QuickSort
- HeapSort

Ogni algoritmo viene eseguito su array di dimensioni diverse: 500, 1000, 5000, 10000, 20000, 35000, 50000. Per ogni dimensione vengono eseguite 150 prove sulle quali viene calcolato il tempo di esecuzione. Il programma informa per ogni gruppo di array il tempo massimo, il minimo e il tempo medio. Per alcuni algoritmi vengono utilizzate anche delle configurazioni particolari dell'array da ordinare, come ad esempio utilizzando il BubbleSort vengono ordinati array strettamente crescenti e strettamente decrescenti oltre che ai campioni casuali. Nella situazione generale gli array vengono generati con numeri interi casuali.

Capitolo 5

Comparativa

Rileggere

MacBook Air

Per alcuni benchmark è stato utilizzato un MacBook Air per confronto. Il PC ha un processore 2, 2 GHz Intel Core i7 dual-core con memoria 8 GB 1600 MHz. Nelle sezioni successive i termini PC fanno riferimento a questa architettura.

Raspberry Pi

Il Raspberry Pi utilizzato è Raspberry Pi model B. La board datata 2013 è stata sviluppata come board scolastica e successivamente applicata a molti contesti come nel mondo embedded. Con la dimensione di una carta di credito il Raspberry ha una RAM di 512 MB e una CPU da 700 MHz, due porte USB (Universal Serial Bus) e un'Ethernet. In aggiunta a ciò, ci sono pin di input/output (GPIO) per uso generico per collegare alcuni hardware. La tabella 5.17 mostra i tempi di esecuzione degli algoritmi di ordinamento eseguiti su Raspberry [8].

5.1 CoreMark

CoreMark è stato compilato con:

```
./riscv64-unknown-elf-gcc -O2 -o coremark.exe core_list_join.c core_main.c core_matrix.c core_state.c  
                           → core_util.c simple/core_portme.c -DPERFORMANCE_RUN=1 -DITERATIONS=1000
```

Code 5.1: compilazione CoreMark



Figura 5.1: Vista superiore Raspberry model B

I codici sorgente sono stati compilati utilizzando alcuni file di configurazione per RISC-V presenti nella repository[25]. I file utilizzati non alterano la validità dell'esecuzione di CoreMark [24].

Il programma è stato eseguito quindici volte con i seguenti risultati:

	Score		Score		Score		Score
1	2312,479985	5	2311,881624	9	2311,874448	13	2311,444275
2	2312,596169	6	2311,149103	10	2311,432578	14	2310,397101
3	2311,137567	7	2310,215865	11	2311,265017	15	2311,553826
4	2311,144929	8	2312,048460	12	2311,013694		

Tabella 5.1: Score di CoreMark ad ogni esecuzione

Media	2311, 307997
Dev.std	0, 754112

Tabella 5.2: Score medio e deviazione standard di CoreMark

5.2 LINPACK

Il benchmark sia nella versione LINPACK100 che LINPACK1000 è stato compilato con:

	Cortex A72	BCM2837	BCM2835	D1	A20	X1000E
CoreMark	48.626	15.364	1.303, 8	2.240, 8	2.086, 2	2.231, 1
CoreMark/MHz	22, 670	12, 803	1, 8625	2, 2408	2, 0862	2, 2311

```
| gcc -O4 -DDP -DROLL -o linpackc linpack.c -lm
```

Code 5.2: compilazione LINPACK

Il programma è stato eseguito quindici volte con i seguenti risultati:

	MFLOPS		MFLOPS		MFLOPS		MFLOPS
1	169,004840	5	163,725958	9	170,431042	13	175,125393
2	166,062072	6	161,682756	10	175,125393	14	175,528289
3	168,424495	7	163,999682	11	173,313142	15	173,094698
4	168,672726	8	169,840877	12	175,573170		

N	100	1000
Media	170, 098131	120, 066523
Dev.std	4, 565740	0, 282790

Tabella 5.3: MFLOPS LINPACK

5.3 MultOrShift

Il programma confronta la velocità di esecuzione di moltiplicazione e divisione aritmetiche con lo shift. Il programma genera un array di interi di valori tra zero e maxInt e un array di potenze di due comprese tra due e milleventiquattro. Entrambi gli array hanno mille elementi. Dopo la generazione viene calcolato elemento per elemento il risultato e viene memorizzato il tempo di esecuzione. Il risultato del programma è il tempo medio di esecuzione di mille operazioni per tipo (moltiplicazione normale, divisione normale, moltiplicazione tramite shift, divisione tramite shift). Di seguito alcune porzioni di codice:

```
1 #define N 1000
2
3 int *generaNumeri(){
4     int *a = malloc(N * sizeof(int));
5     for (int i = 0; i < N; i++){
6         a[i] = rand();
7     }
```

	N	100	1000
Media	0, 004040	5, 569164	
Dev.std	0, 000109	0, 013134	

Tabella 5.4: Tempo di risoluzione LINPACK

```

8     return a;
9 }
10
11 int *esponenti(int *a){
12     int *e = malloc(N * sizeof(int));
13     for (int i = 0; i < N; i++){
14         e[i] = (int)log2(rand());
15     }
16     return e;
17 }
```

Code 5.3: Impostazioni dei dati

```

1 void eseguiMult(int *a, int *b, int* res){
2
3     for(int i = 0; i < N; i++){
4         res[i] = a[i] * b[i];
5     }
6
7 }
8
9 void eseguiDiv(int *a, int *b, int* res){
10
11    for (int i = 0; i < N; i++){
12        res[i] = a[i] / b[i];
13    }
14
15 }
```

Code 5.4: Moltiplicazione

```

1 void eseguiMultShift(int *a, int *b, int *res){
2
3     for (int i = 0; i < N; i++){
4         res[i] = a[i] << b[i];
5     }
6
7 }
8
9 void eseguiDivShift(int *a, int *b, int *res){
10
11    for (int i = 0; i < N; i++) {
```

```

12     res[i] = a[i] >> b[i];
13 }
14 }
15 }
```

Code 5.5: Shift

Il codice 5.3 imposta i due array. I codici 5.4 e 5.5 mostrano l'operazione eseguita. La tabella 5.5 mostra i tempi di esecuzione calcolati in millisecondi delle varie operazioni.

	PC		RISC-V	
Normale	0.0036	0.0042	0.0430	0.0572
Shift	0.0034	0.0034	0.0404	0.0826

Tabella 5.5: Tempi di esecuzione operazioni calcolati in ms

5.4 Analisi codice assembly

Per capire meglio i risultati osserviamo e compariamo i codici generati dai compilatori. I Compilatori confrontati sono *gcc RISC-V*, *gcc ARM* e *gcc x86-64*. I compilatori RISC-V e ARM vengono confrontati per il loro focus sull'ambiente embedded ed entrambi sono architetture RISC. Tutti i codici presentati in questa sezione vengono compilati e presentati con il livello di ottimizzazione di default (-O0) [16].

5.4.1 Addizione con costante

```

1 int get_num(int num) {
2     return 23 + num;
3 }
```

Code 5.6: Addizione

La funzione è una semplice funzione scritta in C che dato un numero di tipo intero restituisce il numero sommato a ventitré¹.

RISC-V

Il sorgente compilato con il compilatore RISC-V 5.7 da riga due fino a riga sei, predispone la chiamata della procedura posizionando sullo stack il

¹La scelta del numero è puramente casuale

necessario. Da riga sette inizia la funzione. Su quella riga viene recuperato il valore di *num* che alla riga otto, tramite l'operazione di add immediate, viene sommato a num. Il risultato dell'operazione addiw è la somma del valore di num sommato alla costante ventitré. Il risultato è esteso su sessantaquattro bit e vengono ignorati gli errori di overflow. Successivamente tramite la pseudo-istruzione sext.w prende i 32 bit inferiori e li memorizza nel registro rd. Questa istruzione corrisponde a addiw rd, rs, 1 0. Il risultato viene spostato nel registro a0 che, nei processori RISC-V, viene utilizzato come restituzione di risultato. Le righe successive ripristinano lo stack e restituiscono il controllo al chiamante.

ARM

Il sorgente 5.9, compilato con gcc ARM, mostra che la preparazione della procedura si esegue da riga due a riga cinque. Con le due righe successive si esegue la funzione. La riga sei recupera il valore di num. La riga successiva calcola il valore del risultato e, infine, alla riga otto si sposta il risultato nel registro di restituzione

x86

Il sorgente 5.8 è compilato con gcc di x86. Da riga due fino a riga quattro viene preparato lo stack, a riga cinque viene posizionato num nel registro eax e a riga sei viene sommato a ventitré e memorizzato nel registro eax. Infine viene ridato il controllo al chiamante.

Code (5.7) RISC-V

```
get_num:
    addi    sp,sp,-32
    sd     s0,24(sp)
    addi    s0,sp,32
    mv     a5,a0
    sw     a5,-20(s0)
    lw     a5,-20(s0)
    addiw   a5,a5,23
    sext.w  a5,a5
    mv     a0,a5
    ld     s0,24(sp)
    addi    sp,sp,32
    jr     ra
```

Code (5.8) Arm

```
get_num:
    push   {r7}
    sub    sp, sp, #12
    add    r7, sp, #0
    str   r0, [r7, #4]
    ldr   r3, [r7, #4]
    adds  r3, r3, #23
    mov   r0, r3
    adds  r7, r7, #12
    mov   sp, r7
    ldr   r7, [sp], #4
    bx    lr
```

Code (5.9) x86

```
get_num:
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp
    ↪ -4], edi
    mov    eax, DWORD PTR
    ↪ [rbp-4]
    add    eax, 23
    pop   rbp
    ret
```

Figura 5.2: Funzione di somma

5.4.2 Addizione

Nel caso generale viene calcolata la somma di tre numeri.

```
1| int sumGen(int num, int num2, int num3) {
```

```

2|     return num + num2 + num3 ;
3|

```

Code 5.10: Addizione generale

RISC-V

Nel caso del compilatore RISC-V la somma avviene tra riga tredici e riga diciannove dove gli operandi vengono caricati nei registri a4 e a5 e successivamente calcolato il risultato e memorizzato in a4 che poi verrà sommato con l'ultimo operando, caricato a riga diciassette.

ARM

Nel caso ARM avviene lo stesso meccanismo. Tra le righe otto e dodici avviene il caricamento dei primi due operandi e la somma parziale e infine la somma totale.

x86

Infine per x86 il calcolo avviene tra le righe sette e undici nello stesso modo con cui viene eseguito in ARM.

Code (5.11) RISC-V

```

sumGen:
    addi    sp,sp,-32
    sd      s0,24(sp)
    addi    s0,sp,32
    mv      a5,a0
    mv      a3,a1
    mv      a4,a2
    sw      a5,-20(s0)
    mv      a5,a3
    sw      a5,-24(s0)
    mv      a5,a4
    sw      a5,-28(s0)
    lw      a4,-20(s0)
    lw      a5,-24(s0)
    addw   a5,a4,a5
    sext.w a5,a5
    lw      a4,-28(s0)
    addw   a5,a4,a5
    sext.w a5,a5
    mv      a0,a5
    ld      s0,24(sp)
    addi   sp,sp,32
    jr      ra

```

(a) RISC-V

Code (5.13) x86

```

sumGen:
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp
    ↗ -4], edi
    mov    DWORD PTR [rbp
    ↗ -8], esi
    mov    DWORD PTR [rbp
    ↗ -12], edx
    mov    edx, DWORD PTR
    ↗ [rbp-4]
    mov    eax, DWORD PTR
    ↗ [rbp-8]
    add    edx, eax
    mov    eax, DWORD PTR
    ↗ [rbp-12]
    add    eax, edx
    pop    rbp
    ret

```

Code (5.12) ARM

```

sumGen:
    push   {r7}
    sub    sp, sp, #20
    add    r7, sp, #0
    str   r0, [r7, #12]
    str   r1, [r7, #8]
    str   r2, [r7, #4]
    ldr   r2, [r7, #12]
    ldr   r3, [r7, #8]
    add   r2, r2, r3
    ldr   r3, [r7, #4]
    add   r3, r3, r2
    mov   r0, r3
    adds r7, r7, #20
    mov   sp, r7
    ldr   r7, [sp], #4
    bx    lr

```

Figura 5.3: Funzione di somma

5.4.3 Moltiplicazione

Moltiplicazioni per potenze di 2

Code (5.15) RISC-V

```

mult2:
    addi    sp,sp,-32
    sd      s0,24(sp)
    addi    s0,sp,32
    mv      a5,a0
    sw      a5,-20(s0)
    lw      a5,-20(s0)
    slliw   a5,a5,1
    sext.w  a5,a5
    mv      a0,a5
    ld      s0,24(sp)
    addi    sp,sp,32
    jr      ra

```

Code (5.16) ARM

```

mult2:
    push   {r7}
    sub    sp, sp, #12
    add    r7, sp, #0
    str    r0, [r7, #4]
    ldr    r3, [r7, #4]
    lsls   r3, r3, #1
    mov    r0, r3
    adds   r7, r7, #12
    mov    sp, r7
    ldr    r7, [sp], #4
    bx

```

Code (5.17) x86

```

mult2:
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp
    ↪ -4], edi
    mov    eax, DWORD PTR
    ↪ [rbp-4]
    add    eax, eax
    pop    rbp
    ret

```

Figura 5.4: Moltiplicazione per 2

Code (5.18) RISC-V

```

mult8:
    addi   sp,sp,-32
    sd     s0,24(sp)
    addi   s0,sp,32
    mv     a5,a0
    sw     a5,-20(s0)
    lw     a5,-20(s0)
    slliw  a5,a5,3
    sext.w a5,a5
    mv     a0,a5
    ld     s0,24(sp)
    addi   sp,sp,32
    jr     ra

```

Code (5.19) ARM

```

mult8:
    push   {r7}
    sub    sp, sp, #12
    add    r7, sp, #0
    str    r0, [r7, #4]
    ldr    r3, [r7, #4]
    lsls   r3, r3, #3
    mov    r0, r3
    adds   r7, r7, #12
    mov    sp, r7
    ldr    r7, [sp], #4
    bx

```

Code (5.20) x86

```

mult8:
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp
    ↪ -4], edi
    mov    eax, DWORD PTR
    ↪ [rbp-4]
    sal    eax, 3
    pop    rbp
    ret

```

Figura 5.5: Moltiplicazione per 8

```

1 int mult2(int num){
2     return 2 * num;
3 }

```

Code 5.14: Moltiplicazione per potenza di 2

La funzione "dato un numero di tipo intero" restituisce il numero moltiplicato per due. Per i sorgenti le parti di preparazione sono simili alle rispettive preparazioni precedenti.

Nei sorgenti in figura 5.3 viene mostrata l'operazione di moltiplicazione per due. Questa avviene per RISC-V e per ARM tramite uno shift logical left di un bit (SLLIW) mentre per x86 avviene tramite una somma. Questa somma è un caso particolare: infatti se volessimo moltiplicare per una qualsiasi potenza di due, le operazioni avvengono tutte tramite shift left di un opportuno valore. Con la figura 5.5 viene mostrato il calcolo di una moltiplicazione per otto. In tutti i casi il calcolo avviene attraverso Shift.

Moltiplicazione per una costante

Vengono presentati due codici: il primo moltiplica per un numero che dista da una potenza di due solamente uno, il secondo che dista di due. I numeri presi per questo esempio sono trentuno, nel primo caso, e trenta nel secondo.

```

1 int mul31(int a) {
2     return a * 31;
3 }
```

Code (5.21) moltiplicazione per
31

```

1 int mul31(int a) {
2     return a * 30;
3 }
```

Code (5.22) moltiplicazione per
30

Nel caso della moltiplicazione per trentuno l'approccio dei tre sorgenti è identico. Viene calcolata la moltiplicazione per trentadue attraverso shift logici e poi viene sottratto una volta il valore per ottenere la moltiplicazione per trentuno. Se il valore costante fosse trentatré, numero successivo alla potenza di due, l'operazione di sottrazione viene sostituita con una di addizione. Nel secondo caso abbiamo un approccio differente.

Partendo dal x86 la moltiplicazione avviene semplicemente con l'istruzione imul a riga sei. Nel caso ARM l'operazione di moltiplicazione viene comunque eseguita da una singola istruzione(riga otto) ma vengono utilizzati i registri r2 e r3 che precedentemente (riga sei e sette) vengono riempiti con gli operandi. Infine l'implementazione di RISC-V utilizza ancora shift. Nel caso della moltiplicazione per trenta viene prima uno shift di quattro (moltiplicazione per sedici) successivamente sottratto una volta il numero e infine al risultato viene applicato uno shift di uno (moltiplicazione per due). Quindi:

$$\begin{aligned}
 & ((num * 2^4) - num) * 2^1 = \\
 & = ((num * 16) - num) * 2 = \\
 & = 15 * num * 2 = num * 30
 \end{aligned}$$

In generale RISC-V utilizza opportune operazioni di shift combinate con addizioni e sottrazioni per ottenere il valore della costante.

Code (5.23) RISC-V

```

mul31:
    addi    sp,sp,-32
    sd      s0,24(sp)
    addi    s0,sp,32
    mv      a5,a0
    sw      a5,-20(s0)
    lw      a4,-20(s0)
    mv      a5,a4
    slliw   a5,a5,5
    subw   a5,a5,a4
    sext.w  a5,a5
    mv      a0,a5
    ld      s0,24(sp)
    addi    sp,sp,32
    jr      ra

```

Code (5.24) ARM

```

mul31:
    push   {r7}
    sub    sp, sp, #12
    add    r7, sp, #0
    str    r0, [r7, #4]
    ldr    r2, [r7, #4]
    mov    r3, r2
    lsls   r3, r3, #5
    subs   r3, r3, r2
    mov    r0, r3
    adds   r7, r7, #12
    mov    sp, r7
    ldr    r7, [sp], #4
    bx

```

Code (5.25) x86

```

mul31:
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp
    ↪ -4], edi
    mov    edx, DWORD PTR
    ↪ [rbp-4]
    mov    eax, edx
    sal    eax, 5
    sub    eax, edx
    pop    rbp
    ret

```

Figura 5.7: Moltiplicazione per 31

Code (5.26) RISC-V

```

mul30:
    addi    sp,sp,-32
    sd      s0,24(sp)
    addi    s0,sp,32
    mv      a5,a0
    sw      a5,-20(s0)
    lw      a4,-20(s0)
    mv      a5,a4
    slliw   a5,a5,4
    subw   a5,a5,a4
    slliw   a5,a5,1
    sext.w  a5,a5
    mv      a0,a5
    ld      s0,24(sp)
    addi    sp,sp,32
    jr      ra

```

Code (5.27) ARM

```

mul30:
    push   {r7}
    sub    sp, sp, #12
    add    r7, sp, #0
    str    r0, [r7, #4]
    ldr    r3, [r7, #4]
    movs   r2, #30
    mul    r3, r2, r3
    mov    r0, r3
    adds   r7, r7, #12
    mov    sp, r7
    ldr    r7, [sp], #4
    bx

```

Code (5.28) x86

```

mul30:
    push   rbp
    mov    rbp, rsp
    mov    DWORD PTR [rbp
    ↪ -4], edi
    mov    eax, DWORD PTR
    ↪ [rbp-4]
    imul   eax, eax, 30
    pop    rbp
    ret

```

Figura 5.8: Moltiplicazione per 30

5.4.4 Divisione

Per quanto riguarda la divisione viene utilizzata la stessa metodologia della moltiplicazione.

5.4.5 Confronto ISA

Le ISA, RISC-V e ARM, prese in esame hanno un approccio RISC. Molte ISA di tipo RISC utilizzano la stessa struttura di pipeline per le proprie implementazioni architettoniche e non è da meno RISC-V. La struttura di questa ISA è caratterizzata da una pipeline a 5 *stages*. In ogni fase della pipeline l'istruzione compie una determinata azione che permette l'esecuzione quasi parallela. Le cinque fasi della pipeline RISC-V sono *Instruction fetch*, *Instruction decode*, *Execute*, *Memory access*, *Writeback*. Naturalmente la parziale parallelizzazione del codice non evita situazioni critiche(*Hazard*) che vengono risolte tramite le tecniche di *Bypassing* e l'*Interlock*. Queste situazioni criti-

che portano a modifiche necessarie a livello di struttura ma comportano un miglioramento di esecuzione del codice ². Per quanto riguarda ARM, inizialmente le pipeline ARM erano organizzate in pipeline a 3 *stages*: *Instruction fetch*, *Instruction decode*, *Execute*. In questa situazione tre istruzioni diverse possono occupare i tre stadi. Con questa struttura quando il processore esegue istruzioni di *data processing* la latenza massima è pari ai tre cicli. Inoltre ogni istruzione di trasferimento dati causa uno stallo della pipeline [23]. Avendo la memoria unica per dati e per le istruzioni, non è possibile leggere l'istruzione successiva mentre si leggono dati. Quindi aumenta la latenza. Solamente dal ARM9TDMI, nel 1998, la pipeline è diventata a 5 stadi. Tuttavia le architetture ARM con focus nell'embedded sono caratterizzate da una pipeline a 3 stadi.

RISC-V	Clock	Linee	CPI
get_num	17	12	1.41
sumgen	24	22	1.09
mult2	17	12	1.41
mult8	17	12	1.41
mult30	20	15	1.33
mult31	20	14	1.42

Tabella 5.6: CPI per RISC-V

ARM	Clock	Linee	CPI
get_num	27	11	2.45
sumgen	31	16	1.93
mult2	21	11	1.9
mult8	21	11	1.9
mult30	25	12	2.08
mult31	25	13	1.92

Tabella 5.7: CPI per ARM

Le tabelle 5.6 e 5.7 mostrano i clock richiesti per eseguire ogni programma. La tabella 5.8 confronta direttamente i CPI delle architetture. Dalle tabelle possiamo capire che pur avendo un numero di linee di codice maggiore e una dimensione maggiore, RISC-V riesce, grazie alla struttura in 5 stages, ad avere un numero di clock per esecuzione minore rispetto a ARM.

²Core della CPU D1-H è XuanTie C906 [26], ha una pipeline a 5 *stages*

mattia: aggiunto alla fine qualche parola di commento(CPI, lunghezza, dimensione) - atrent: ok

CPI	RISC-V	ARM
get_num	1.41	2.25
sumgen	1.09	1.93
mult2	1.41	1.9
mult8	1.41	1.9
mult30	1.33	2.08
mult31	1.42	1.92

Tabella 5.8: CPI a confronto

Dimensione codice	get_num	sumGen	mult2	mult8	mult30	mult31
ARM	2546	2896	2535	2537	2584	2569
RISC-V	3295	3588	3284	3286	3306	3322

Tabella 5.9: Code size in byte [B]

5.5 Sorting

Gli algoritmi di ordinamento sono una parte importante dell’elaborazione dei dati e sono ampiamente utilizzati in molti aspetti ad esempio in crittografia e nella ricerca di informazioni. Esistono molti tipi di algoritmi di ordinamento e ognuno ha i suoi vantaggi e limiti. In informatica, l’algoritmo di ordinamento è solitamente classificato per:

- Complessità temporale.
- Memoria utilizzata.
- Stabilità.

In base alle proprietà dei diversi tipi di dati, l’efficienza può essere migliorata scegliendo algoritmi di ordinamento appropriati. In questo capitolo verranno descritti quattro algoritmi di ordinamento e verranno analizzati comparandoli all’esecuzione su raspberry. L’ordinamento a cui si fa riferimento nelle sezioni successive è un ordinamento di array da mettere in ordine non decrescente.

5.5.1 Bubble sort

Bubble sort è un semplice algoritmo di ordinamento. L’algoritmo consiste nel scansionare tutte le coppie di valori adiacenti nell’array e scambiarli se

non rispettano l'ordinamento. Questo processo viene reiterato per ogni coppia dell'array un numero di volte pari alla dimensione meno uno.

L'efficienza dell'algoritmo è basata sull'ordinamento dell'array da ordinare. Nel caso migliore l'array è già ordinato e vengono eseguite solo un numero di scansioni pari al numero di elementi meno uno. Al contrario il caso peggiore è quando l'array si trova nell'ordine opposto a quello desiderato.

	Complessità
Caso peggiore	$\mathcal{O}(n^2)$
Caso migliore	$\mathcal{O}(n)$

Tabella 5.10: complessità bubble sort

Di seguito viene riportato il codice (Code 5.29) dell'algoritmo e i tempi di esecuzione(Tab 5.11) secondo la dimensione dell'array e la situazione iniziale dell'array:

```

1 void bubbleSort(int a[], int dim){
2     int temp;
3     for (int i = 0; i < dim; i++){
4         for (int j = 0; j < dim - i - 1; j++){
5             if (a[j] > a[j + 1]){
6                 temp = a[j];
7                 a[j] = a[j + 1];
8                 a[j + 1] = temp;
9             }
10        }
11    }
12 }
```

Code 5.29: Algoritmo bubble sort scritto in c

	Caso peggiore	Caso migliore	Caso Generale
500	0.009426	0.004667	0.007156
1000	0.037558	0.018877	0.028745
5000	0.941429	0.470478	0.718178
10000	3.784664	1.893387	2.889989
20000	15.249109	7.639312	11.617705
35000	46.744558	23.460390	35.640101
50000	95.389578	47.888541	72.735791

Tabella 5.11: Tempi di esecuzione bubble sort in ms

5.5.2 Insertion sort

L'algoritmo consiste nell'inserire un elemento dopo l'altro nella parte di array ordinata fino a terminare gli elementi da ordinare.

	Complessità
Caso peggiore	$\mathcal{O}(n!)$
Caso migliore	$\mathcal{O}(n)$
Caso medio	$\mathcal{O}(n!)$

Tabella 5.12: complessità insertion sort

```

1 void insertionSort(int a[], int dim){
2
3     int temp, j;
4     for (int i = 1; i < dim; i++){
5         temp = a[i];
6         j = i - 1;
7         while (j >= 0 && a[j] > temp){
8             a[j + 1] = a[j];
9             j--;
10        }
11        a[j + 1] = temp;
12    }
13 }
```

Code 5.30: Algoritmo bubble sort scritto in c

Come nel caso del bubble sort è riportato anche il tempo di esecuzione nel caso peggiore e migliore.

	Caso peggiore	Caso migliore	Caso Generale
500	0.005423	0.000028	0.002634
1000	0.020836	0.000055	0.010575
5000	0.521686	0.000287	0.260862
10000	2.103697	0.000645	1.048553
20000	8.497346	0.001114	4.220421
35000	26.066370	0.001928	12.969309
50000	53.187672	0.002720	26.499820

Tabella 5.13: Tempi di esecuzione insertion sort in ms

5.5.3 QuickSort

L'algoritmo quicksort è un algoritmo ricorsivo del tipo divide et impera. L'algoritmo consiste nello scegliere un elemento come perno e spostare a destra e sinistra gli altri elementi rispetto all'ordine scelto. Questo procedimento viene ripetuto successivamente per la parte di sinistra e di destra fino ad ottenere l'ordine desiderato.

Il tempo di esecuzione nel caso peggiore è $\mathcal{O}(n!)$, il tempo di esecuzione medio di quicksort è $\mathcal{O}(n * \log n)$. In seguito viene mostrata l'implementazione.

```
1 void QuickSort(int v[], int in, int fin){
2     if (fin <= in)
3         return;
4     int pos = partiziona(v, in, fin);
5
6     QuickSort(v, in, pos - 1);
7     QuickSort(v, pos + 1, fin);
8 }
9
10 int partiziona(int v[], int in, int fin){
11
12     int i = in + 1, j = fin;
13     while (i <= j){
14         while ((i <= fin) && (v[i] <= v[in]))
15             i++;
16         while (v[j] > v[in])
17             j--;
18         if (i <= j){
19             int t = v[i];
20             v[i] = v[j];
21             v[j] = t;
22         }
23     }
24
25     int tt = v[in];
26     v[in] = v[i - 1];
27     v[i - 1] = tt;
28
29     return i - 1;
30 }
```

Code 5.31: Algoritmo quick sort scritto in c

	Tempo
500	0.000273
1000	0.000610
5000	0.003813
10000	0.008423
20000	0.017266
35000	0.032285
50000	0.048096

Tabella 5.14: Tempi di esecuzione quick sort in ms

5.5.4 Heap sort

L'heapsort è un algoritmo di ordinamento iterativo, l'implementazione utilizzata è in-place.

```

1 void swap(int *a, int *b){
2     int temp = *a;
3     *a = *b;
4     *b = temp;
5 }
6
7 void heapify(int arr[], int n, int i){
8     int largest = i;
9     int left = 2 * i + 1;
10    int right = 2 * i + 2;
11
12    if (left < n && arr[left] > arr[largest])
13        largest = left;
14
15    if (right < n && arr[right] > arr[largest])
16        largest = right;
17
18    if (largest != i){
19        swap(&arr[i], &arr[largest]);
20        heapify(arr, n, largest);
21    }
22 }
23
24 void heapSort(int arr[], int n){
25     for (int i = n / 2 - 1; i >= 0; i--)
26         heapify(arr, n, i);
27
28     for (int i = n - 1; i >= 0; i--){
29         swap(&arr[0], &arr[i]);
30         heapify(arr, i, 0);
31     }

```

Code 5.32: Algoritmo heap sort scritto in c

	Tempo
500	0.000569
1000	0.001272
5000	0.008016
10000	0.017815
20000	0.041886
35000	0.083366
50000	0.128863

Tabella 5.15: Tempi di esecuzione heap sort in ms

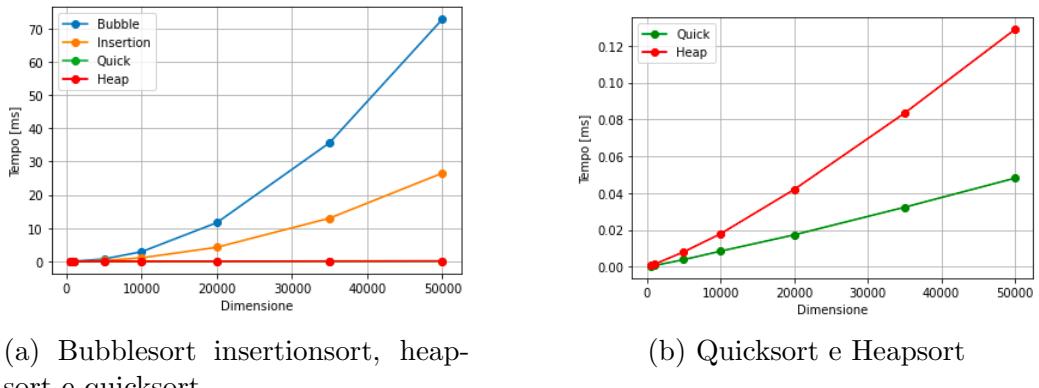


Figura 5.9: Algoritmi di ordinamento a confronto

5.5.5 Confronto Sorting

In questa sezione vengono visualizzati i risultati a confronto. Per prima cosa iniziamo con confrontare i risultati con l'esecuzione su MacBook.

La tabella 5.16 mostra i tempi di esecuzione degli algoritmi di ordinamento eseguiti sul PC. Un altro confronto lo possiamo fare con Raspberry pi³.

I grafici 5.14 5.15 5.16 5.17 mostrano i tempi di esecuzione degli algoritmi di sorting comparati tra Rasberry e RISC-V. In ogni grafico si vede che il tempo di esecuzione è migliore su RISC-V. La tabella 5.18 mostra il rapporto

³Il modello utilizzato è il modello Raspberry model B mostrato in figura 5.1

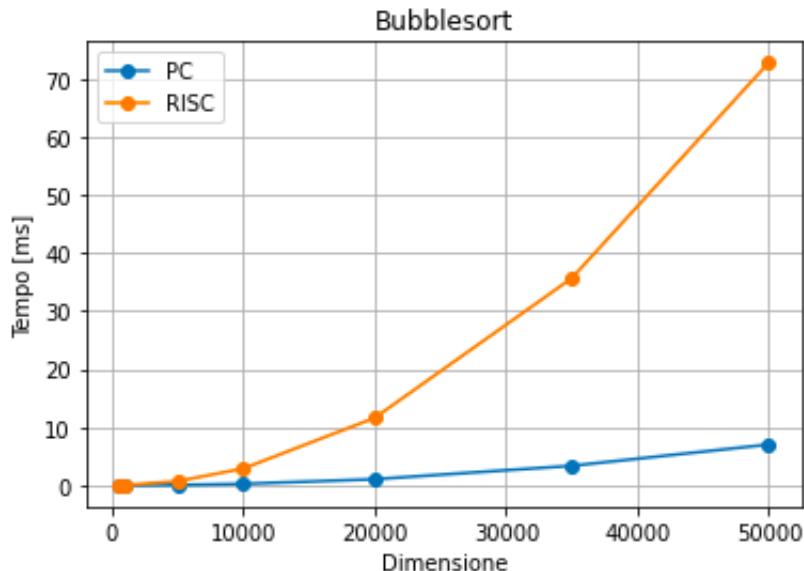


Figura 5.10: Tempi di esecuzione Bubblesort PC e RISC-V

	Bubblesort	Insertionsort	Heapsort	Quicksort
500	0.000491	0.000196	0.000057	0.000052
1000	0.001871	0.000723	0.000145	0.000080
5000	0.055505	0.016859	0.000975	0.000544
10000	0.247916	0.066411	0.002083	0.001164
20000	1.056383	0.261627	0.004485	0.002441
35000	3.359662	0.800075	0.008340	0.004557
50000	7.019640	1.633171	0.012359	0.006857

Tabella 5.16: Tempi di esecuzione degli algoritmi di sorting su PC in ms

tra gli algoritmi eseguiti sul Raspberry e quelli eseguiti sulla board RISC-V. Prendendo in considerazione il Bubblesort nel caso con 50000 elementi, la board con processore RISC-V è 3.44 volte più veloce del raspberry. In altri casi, come ad esempio per heap sort, il miglioramento di RISC-V è 1.64 rispetto a Raspberry. In media abbiamo un miglioramento di 2.12. Il che significa che la board in esame esegue il doppio più velocemente gli algoritmi di ordinamento del Raspberry.

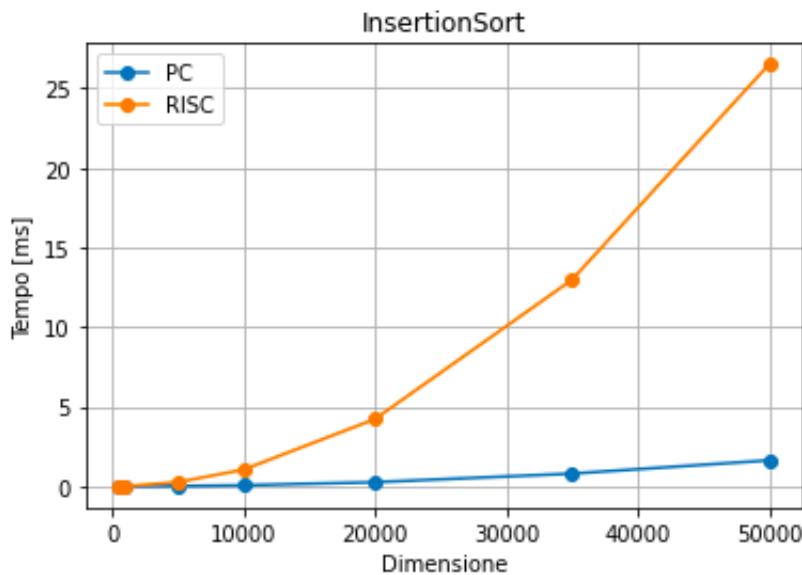


Figura 5.11: Tempi di esecuzione Insertionsort PC e RISC-V

	Bubblesort	Insertionsort	Heapsort	Quicksort
500	0.020337	0.004877	0.000825	0.000511
1000	0.078527	0.020353	0.001894	0.001141
5000	2.030446	0.045018	0.013863	0.009072
10000	8.535436	1.874295	0.027671	0.018542
20000	34.733894	7.842624	0.058596	0.038194
35000	117.622606	24.076289	0.114716	0.082891
50000	250.008917	51.757841	0.210807	0.157344

Tabella 5.17: Tempi di esecuzione Raspberry Pi B in ms

	Bubblesort	Insertionsort	Heapsort	Quicksort	
500	2.84	1.85	1.45	1.87	
1000	2.73	1.92	1.49	1.87	
5000	2.83	0.17	1.73	2.38	
10000	2.95	1.79	1.55	2.20	
20000	2.99	1.86	1.40	2.21	
35000	3.30	1.86	1.38	2.57	
50000	3.44	1.95	1.64	3.27	
Media	3.01	1.63	1.52	2.34	2.12

Tabella 5.18: Rapporto prestazioni ARM e RISC-V

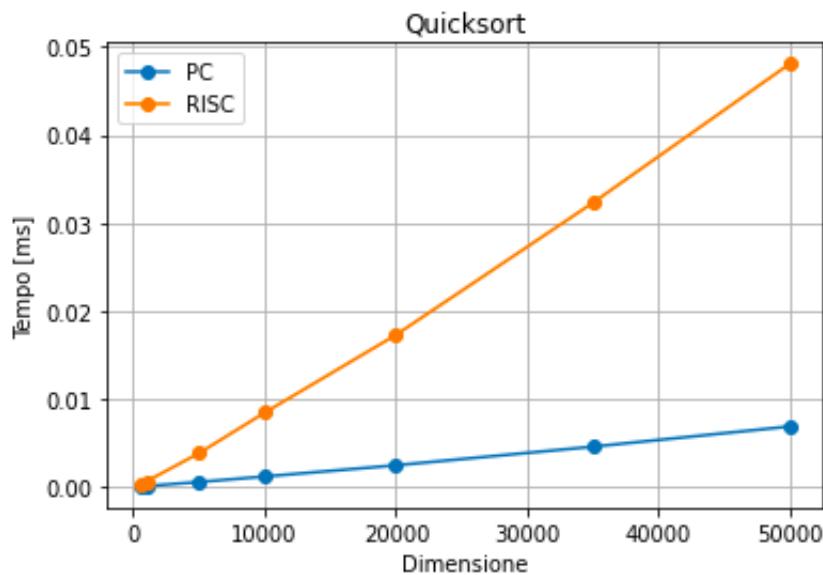


Figura 5.12: Tempi di esecuzione Quicksort PC e RISC-V

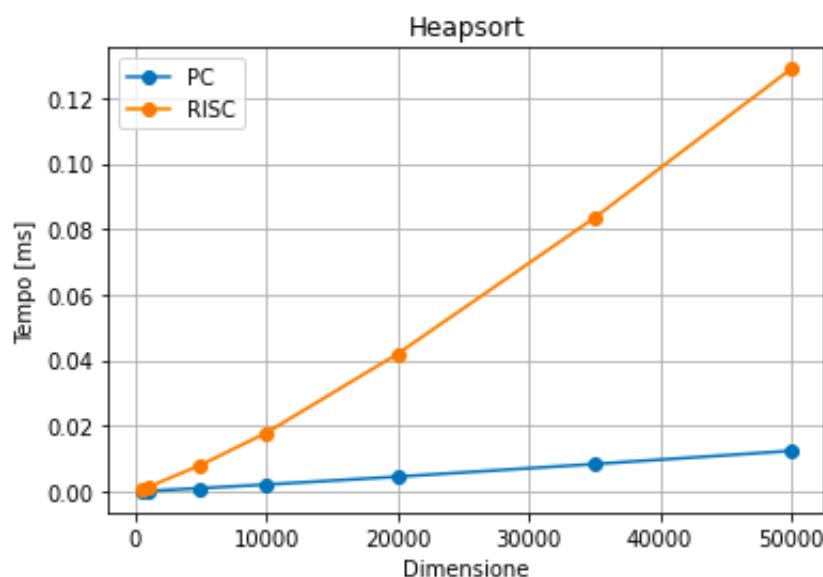


Figura 5.13: Tempi di esecuzione Heapsort PC e RISC-V

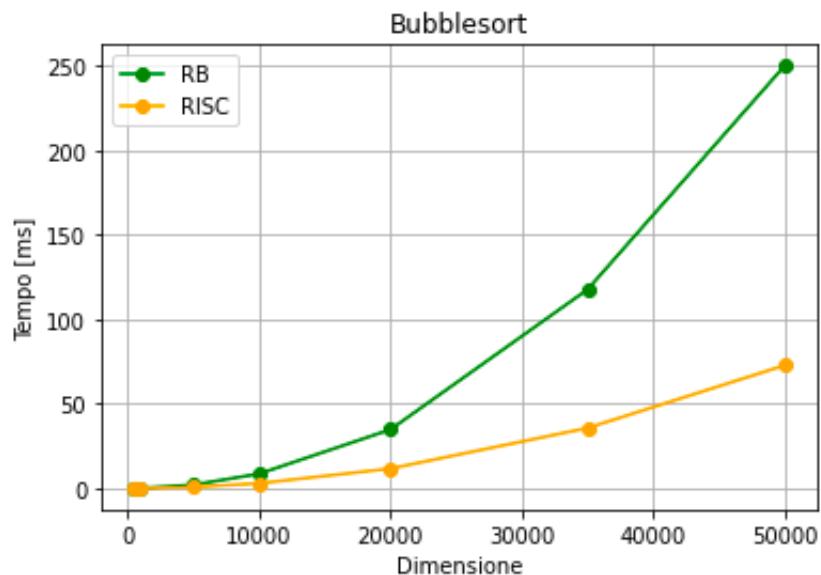


Figura 5.14: Tempi di esecuzione Bubblesort Raspberry e RISC-V

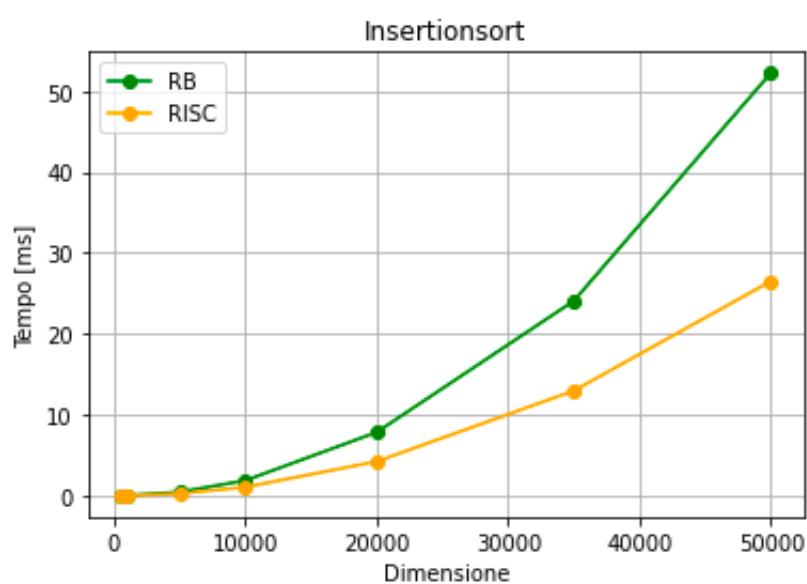


Figura 5.15: Tempi di esecuzione Insertionsort Raspberry e RISC-V

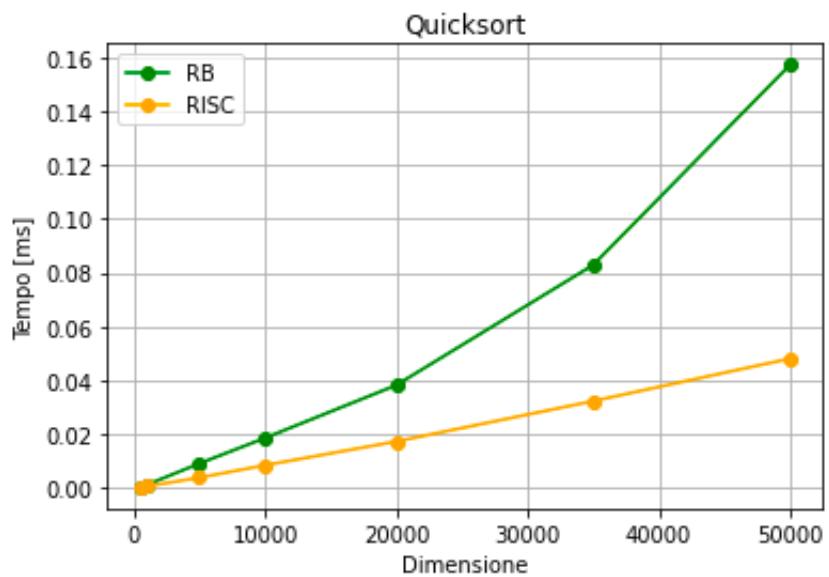


Figura 5.16: Tempi di esecuzione Quicksort Raspberry e RISC-V

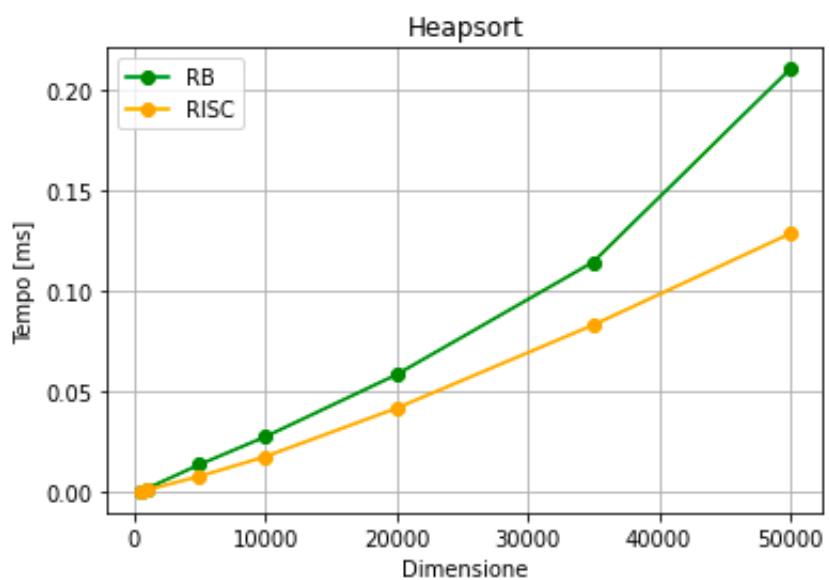


Figura 5.17: Tempi di esecuzione Heapsort Raspberry e RISC-V

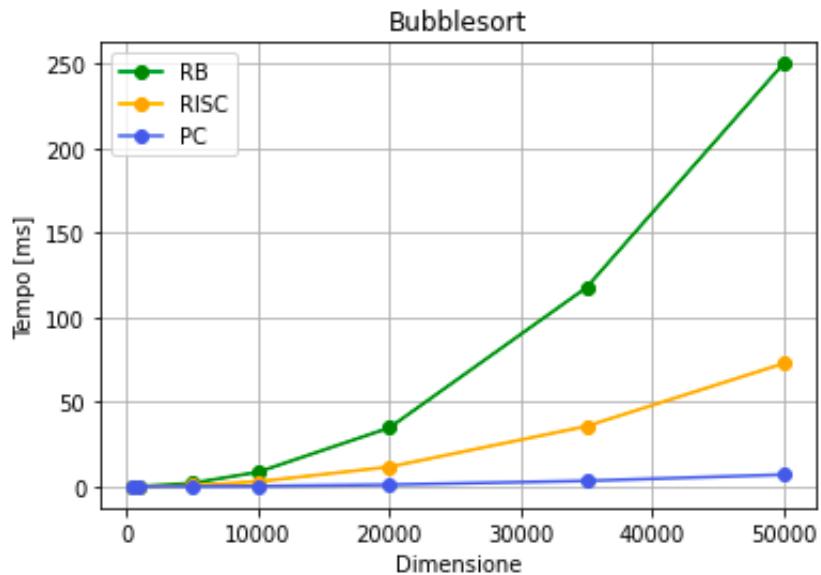


Figura 5.18: Tempi di esecuzione Bubblesort

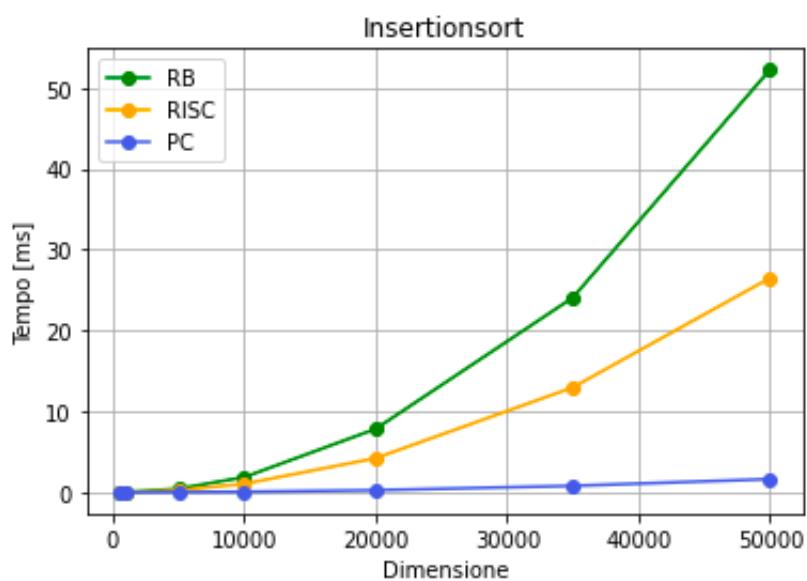


Figura 5.19: Tempi di esecuzione Insertionsort

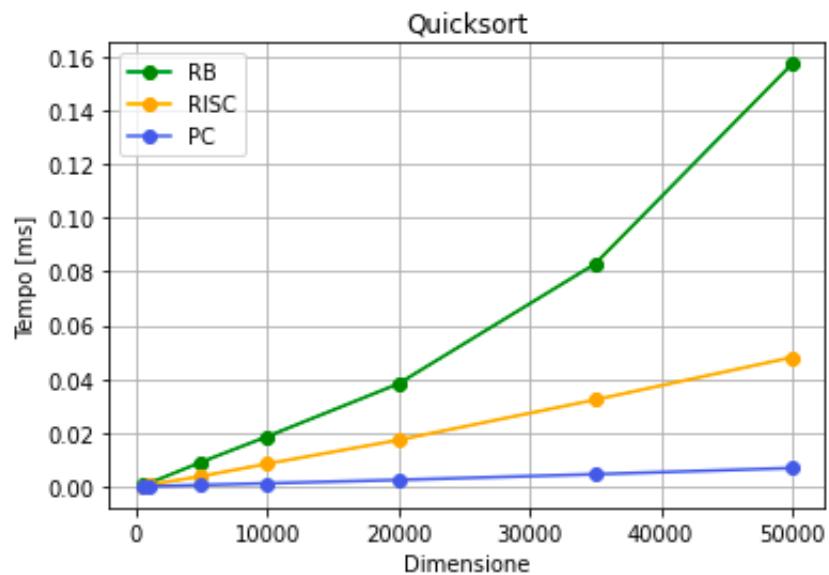


Figura 5.20: Tempi di esecuzione Quicksort

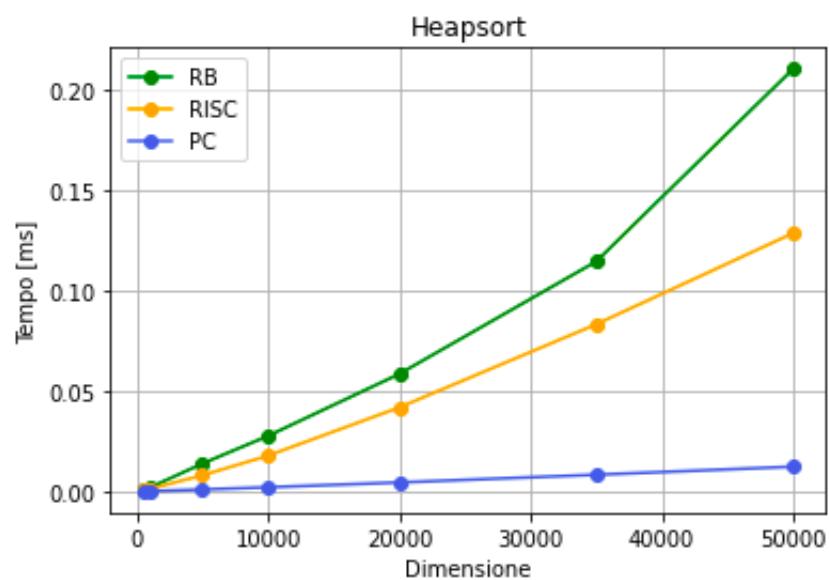


Figura 5.21: Tempi di esecuzione Heapsort

Capitolo 6

Conclusione

In questa tesi sono stati fatti alcuni test su RISC-V.

Per quanto riguarda il codice assembly, generato il compilatore RISC-V genera codici di dimensioni maggiori rispetto ad ARM. RISC-V richiede più spazio quindi è più costoso in termini di memoria di programma richiesta, ma richiede meno cicli di clock per esecuzione. I codici riportati mostrano che la media richiesta da ARM è di 2.03 CPI mentre per RISC-V è 1.34. Il che significa che in media una singola istruzione per ARM richiede circa due cicli per essere completata mentre RISC-V per una singola istruzione, in media, richiede meno di due cicli. Inoltre, grazie alla sua struttura RISC-V ha un numero maggiore di Istruzioni per clock (IPC).

Per quanto riguarda il sorting, il confronto con gli algoritmi tra RISC-V, il Raspberry e il pc, mostra che la board dotata di RISC-V è sempre più veloce del Raspberry in ogni algoritmo ma è sempre più lento del pc. Rispetto ai tempi di esecuzione del Raspberry la board migliora il tempo di esecuzione in media di 2.12. Possiamo concludere osservando che la board RISC-V è una valida sostituta del Raspberry in termini di velocità di calcolo.

Utilizzando anche altri studi condotti su RISC-V fatti su consumi e potenza, sempre comparando RISC-V con ARM, i processori RISC-V consumano meno potenza rispetto ad ARM e offrono prestazioni migliori per watt. Per quanto riguarda i costi dei chip, i processori RISC-V tendono ad avere dimensioni inferiori rispetto ai processori ARM quindi meno costosi in termini di silicio, rendendoli più adatti per applicazioni sensibili ai costi [39].

In conclusione possiamo affermare che il progetto RISC-V, nato come progetto didattico, è arrivato sul mercato supportato da una community sempre più crescente che lo ha portato a notevoli sviluppi. Oggi si affaccia sul mercato affiancandosi ad ARM dando la possibilità ai progettisti di scegliere un'alternativa free e open-source per le loro necessità. L'offerta di RISC-V è di una tecnologia a basso consumo, a costo ridotto di materiale, con

la possibilità di personalizzare liberamente l'ISA per specifiche funzionalità. Con queste premesse le aree di applicazione della tecnologia RISC-V sono molteplici.

Le aree di applicazione di RISC-V oggi sono i *microcontrollers* e il mondo embedded dove vengono progettati piccoli dispositivi a basso consumo per svolgere determinate funzioni che possono essere per *smart home* o dispositivi IoT [19]. Un'altra area di applicazione è il *machine learning* [31], dove RISC-V grazie alla sua configurabilità ottiene la possibilità di soddisfare requisiti specifici [12].

Questa tesi analizza solo alcuni aspetti di RISC-V. Terminato questo lavoro di tesi vorrei sottolineare che non si tratta di un lavoro concluso, ma un punto di partenza per possibili approfondimenti futuri.

atrent: conclusione minimale, riesci a dire se ci sono aspetti in cui uno è meglio dell'altro e viceversa? comparazioni di costo? corrente consumata? ecc. - mattia: ok - atrent: meglio

atrent: nelle conclusioni dovresti a questo punto (a valle del tuo lavoro di analisi) poter dire non solo come si confronta risc-v con gli altri, ma anche quali sono le aree di debolezza e forza, tanto da ipotizzare quale potrebbero essere i contesti d'uso favorevoli - mattia: ok? - atrent: forse alcune affermazioni (es. machine learning) andrebbero supportate con qualche bibref

Bibliografia

- [1] Reinhold P. Weicker. «An Overview of Common Benchmarks». In: (dic. 1990).
- [2] Alan R. Weiss. *Dhrystone Benchmark*. 2002. URL: https://www.eembc.org/techlit/datasheets/dhrystone_wp.pdf.
- [3] Alan R. Weiss. *Dhrystone Benchmark History, Analysis, "Scores" and Recommendations*. 2002. URL: <https://web.archive.org/web/20110726210001/http://www.johnloomis.org/NiosII/dhrystone/ECLDhrystoneWhitePaper.pdf>.
- [4] Jack J. Dongarra, Piotr Luszczek e Antoine Petitet. «The LINPACK Benchmark: past, present and future». In: *Concurrency and Computation: Practice and Experience* 15.9 (2003), pp. 803–820. DOI: <https://doi.org/10.1002/cpe.728>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728>.
- [5] Brian J. Gough. *An Introduction to GCC*. [Online]. 2005. URL: <https://web.archive.org/web/20100531171042/http://www.network-theory.co.uk/docs/gccintro/>.
- [6] Marian Vittek, Peter Borovansky e Pierre-Etienne Moreau. «A Simple Generic Library for C». In: *Reuse of Off-the-Shelf Components: , Proceedings of 9th International Conference on Software Reuse, Turin, Italy*. Springer, 2006, pp. 423–426.
- [7] «IEEE Standard for Floating-Point Arithmetic». In: *IEEE Std 754-2008* (2008), pp. 1–70. DOI: [10.1109/IEEESTD.2008.4610935](https://doi.org/10.1109/IEEESTD.2008.4610935).
- [8] Yuan Yue. *Performance Evaluation of Sorting Algorithms in Raspberry Pi and Personal Computer*. Rapp. tecn. Universita di Vaasa, 2015.
- [9] *riscv-spec-v2.2*. 2017. URL: <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>.
- [10] Patrick H Stakem. *RISC Microprocessors, History and Overview*. Independently published, 2018.

- [11] Wei Dai e Daniel Berleant. «Benchmarking contemporary deep learning hardware and frameworks: A survey of qualitative metrics». In: *2019 IEEE First International Conference on Cognitive Machine Intelligence (CogMI)*. IEEE. 2019, pp. 148–155.
- [12] Marcia Sahaya Louis et al. «Towards deep learning using tensorflow lite on risc-v». In: *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*. Vol. 1. 2019, p. 6.
- [13] SiFive. *riscv-llvm*. 2019. URL: <https://github.com/sifive/riscv-llvm>.
- [14] Awol. *Introduzione del chip D1-H*. 2021. URL: <https://d1.docs.awol.com>.
- [15] RISC-V Software Collaboration. *riscv-gcc*. 2021. URL: <https://github.com/riscv-collab/riscv-gcc>.
- [16] Yu Liu, Kejiang Ye e Cheng-Zhong Xu. «Performance Evaluation of Various RISC Processor Systems: A Case Study on ARM, MIPS and RISC-V». In: *International Conference on Cloud Computing*. Springer. 2021, pp. 61–74.
- [17] Wikipedia. *Sistema A-0*. [Online; pagina editata in: 7 Dicembre 2021; Controllata 15 Novembre 2022]. 2021. URL: https://it.upwiki.one/wiki/a-0_system.
- [18] GCC. *GCC, the GNU Compiler Collection*. 2022. URL: <https://gcc.gnu.org/>.
- [19] STACEY HIGGINBOTHAM. *RISC-V is coming to the internet of things*. 2022. URL: <https://staceyoniot.com/risc-v-is-coming-to-the-internet-of-things/> (visitato il 28/03/2022).
- [20] LLVM. *The LLVM Compiler Infrastructure*. 2022. URL: <https://llvm.org/>.
- [21] RISCV. *RISCV History*. 2022. URL: <https://riscv.org/about/history/>.
- [22] Wikipedia. *Clang*. [Online; sito ufficiale: <https://clang.llvm.org/>]. 2022. URL: <https://it.frwiki.wiki/wiki/Clang>.
- [23] arm developer. URL: <https://developer.arm.com/documentation/ddi0337/e/Introduction/Execution-pipeline-stages>.
- [24] CoreMark github. Sezione: 'Systems Without make'. URL: <https://github.com/eembc/coremark>.

- [25] *CoreMark RISCV github*. URL: <https://github.com/riscv-boom/riscv-coremark>.
- [26] *Documentazione nezha d1-h*. URL: <https://www.linuxadictos.com/it/allwinner-xuantie-c906-risc-v.html>.
- [27] *DSPstone*. URL: <https://www.ice.rwth-aachen.de/research/tools-projects/closed-projects/dspstone/>.
- [28] Coremark by EEMBC. URL: <https://www.eembc.org/coremark/>.
- [29] *EEMBC products*. URL: <https://www.eembc.org/products/>.
- [30] *Embench*. URL: <https://www.embench.org/news.html>.
- [31] Inc. Esperanto Technologies. *Esperanto is leading the RISC-V revolution for AI and enabling a new level of AI performance*. URL: <https://www.esperanto.ai/technology/> (visitato il 2023).
- [32] *hpl*. URL: <https://netlib.org/benchmark/hpl/>.
- [33] Cleve Moler e Gilbert Stewart Jack Dongarra Jim Bunch. *LINPACK*. URL: <https://netlib.org/linpack/>.
- [34] *MiBench*. URL: <https://vhosts.eecs.umich.edu/mibench/>.
- [35] *Nettle - a low-level cryptographic library*. URL: <https://www.lysator.liu.se/~nisse/nettle/>.
- [36] *OpenBenchmark*. URL: <https://openbenchmarking.org/>.
- [37] *Phoronix test suite*. URL: <https://www.phoronix-test-suite.com/>.
- [38] *repo CoreMark*. URL: <https://github.com/eembc/coremark>.
- [39] *RISC-V vs ARM: Which Instruction Set Architecture Will Dominate the Future?* URL: <https://www.electropages.com/blog/2021/03/arm-vs-risc-v>.
- [40] Kim Shanley. *Transaction Processing Performance Council*. URL: <https://www.tpc.org/information/about/history5.asp>.
- [41] *SPEC*. URL: <https://www.spec.org/spec/>.
- [42] *WCET*. URL: <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>.