



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 2022-2023

Todo list

- Ho trovato i primi 2 capitoli molto più leggibili. Hai fatto bene a smaltire 15

Introduzione

Il processore è il componente hardware con il quale vengono svolte le istruzioni. Oggigiorno sono presenti numerosi processori ognuno caratterizzato da un insieme di istruzioni che definiscono le istruzioni che, un processore, può eseguire. RISC-V , con un *instruction set* basato su un approccio *reduced instruction set computer*, sta facendo parlare di sè.

L'obiettivo di questa tesi è quello di fare alcuni test sul processore dotato di RISC-V e valutarne le prestazioni.

Per la valutazione sono stati scrittura dei codici in linguaggio C compilati con la *toolchain* di RISC-V.I risultati dei programmi vengono riportati e , in alcuni casi, comparati con risultati di programmi simili compilati per altri processori. In alcuni casi viene anche analizzato il sorgente compilato.

La tesi inizia con una panoramica sul progetto RISC-V presentando gli obiettivi e una panoramica sulla storia del progetto. Il capitolo successivo presenta in sintesi l' ISA di RISC-V presentando alcuni punti importanti del progetto. Successivamente viene presentati alcuni tool di compilazione utilizzati per i programmi. I capitolo 4 e 5 vengono presentati alcuni programmi utilizzati per la valutazione e analizzati i risultati ottenuti.

Indice

1 RISC-V	8
1.1 Gli obiettivi di RISC-V	8
1.2 Panoramica	9
2 ISA RISC-V	11
2.1 Base	11
2.2 Estensioni	11
2.3 Istruzioni base	13
2.4 RV32I	14
2.5 Le altre basi	15
3 Compilatori	16
3.1 Descrizione	16
3.2 Storia	16
3.3 Cross-compilazione	17
3.4 Compilatori moderni	17
3.5 Un buon compilatore	17
3.6 Toolchain di RISC-V	18
4 BenchMarking	19
4.1 Descrizione board	19
4.2 Ambiente di sviluppo	19
4.3 Presentazione programmi	19
4.3.1 Operazioni Aritmetiche	20
4.3.2 Prime Number	20
4.3.3 Moltiplicazione o shift	21
4.3.4 Montecarlo Pi	21
4.3.5 Sorting	21
4.3.6 BackTracking	22

5 Comparativa	24
5.1 Operazioni	24
5.2 PrimeNumber	26
5.3 MultOrShift	28
5.4 Analisi codice assembly	30
5.5 Operazioni	30
5.5.1 Addizione con costante	30
5.5.2 Addizione	31
5.5.3 Moltiplicazione	32
5.5.4 Divisione	35
5.6 MonteCarloPi	35
5.6.1 Generatore casuale	35
5.7 Sorting	38
5.7.1 Bubble sort	38
5.7.2 Insertion sort	40
5.7.3 QuickSort	41
5.7.4 Heap sort	42
5.7.5 Confronto Sorting	44
5.8 BackTracking	45
6 Conclusione	48

Elenco delle figure

2.1	Formati istruzione RISC-V	13
2.2	Convenzione dei registri	14
3.1	Schema meccanismo di compilazione	16
4.1	Board vista dall' alto	23
4.2	Schema a blocchi della scheda di sviluppo	23
5.1	Funzione di somma	31
5.2	Funzione di somma	32
5.3	Moltiplicazione per 2	33
5.4	Moltiplicazione per 8	33
5.6	Moltiplicazione per 31	34
5.7	Moltiplicazione per 30	35
5.8	Esempio di coordinate generate casualmente	36
5.9	Algoritmi di ordinamento a confronto	44
5.10	Algoritmi di ordinamento a confronto eseguiti su PC e su RISC-V	44
5.11	Vista superiore Raspberry model B	45
5.12	Algoritmi di ordinamento a confronto eseguiti su Raspberry e su RISC-V	46

Elenco delle tabelle

2.1	Tabella nomenclatura ISA RISC-V	12
4.1	Caratteristiche della board	20
5.1	Tempi di esecuzione MacBook-Air	26
5.2	Tempi di esecuzione RISC-V	26
5.3	Tempi di esecuzione numeri primi	27
5.4	Tempi di esecuzione operazioni calcolati in ms	30
5.5	Tempi di esecuzione dell'algoritmo	37
5.6	Valori calcolati	38
5.7	complessita bubble sort	39
5.8	Tempi di esecuzione bubble sort	40
5.9	complessita insertion sort	40
5.10	Tempi di esecuzione insertion sort	41
5.11	Tempi di esecuzione quick sort	42
5.12	Tempi di esecuzione heap sort	43
5.13	Tempi di esecuzione degli algoritmi di sorting su PC	45
5.14	Tempi di esecuzione Raspberry Pi B	46
5.15	Tempi di esecuzione del solutore di sudoku	47
5.16	Tempi di esecuzione del solutore di labirinti	47

Listings

5.1	Addizione	24
5.2	Sottrazione	25
5.3	Moltiplicazione	25
5.4	Divisione	25
5.5	Modulo	25
5.6	Impostazioni dei dati	27
5.7	Sorgente prime ottimizzato con -O0	27
5.8	Impostazioni dei dati	28
5.9	Impostazioni dei dati	29
5.10	Impostazioni dei dati	29
5.11	Addizione	30
5.15	Addizione generale	31
5.19	Moltiplicazione per potenza di 2	32
5.34	Funzione per il calcolo del pigreco con il metodo di Monte Carlo	37
5.35	Algoritmo bubble sort scritto in c	39
5.36	Algoritmo bubble sort scritto in c	40
5.37	Algoritmo quick sort scritto in c	41
5.38	Algoritmo heap sort scritto in c	42

Capitolo 1

RISC-V

RISC-V è una ISA basato sul principio RISC nato come progetto accademico. Nel 2010, a Berkeley, il progetto inizio 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 la direzione di sviluppo di RISC-V. Nel 2018 viene annunciata una collaborazione tra l'azienda e *Linux Foundation* con il quale si supporta lo sviluppo del progetto RISC-V.

1.1 Gli obiettivi di RISC-V

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

- un ISA con una licenza *open source* disponibile per accademie 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 per ogni implementazione.
- modulare , organizzata in ISA più piccole con la possibilità di usare estensioni.
- supporto per lo standard floating-point 2008 IEEE-754.
- supporto delle estensioni .

- spazio di indirizzamento 32 e 64 bit.
- supporto a delle implementazioni multicore e manycore sia ererogenee sia omogenee.
- Istruzioni a lunghezza variabile.
- completamente virtualizzabile
- 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 *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 della lunghezza in questo modo il sistema lo decodifica, in questo modo 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 dividendo per un modulo adeguato , sfruttando dunque la caratteristica circolare.

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.
- **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.

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:

Si parla di eccezione quando una condizione non comune avviene a *run time* associata a un'istruzione. Si parla di *trap* quando 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.

Capitolo 2

ISA RISC-V

Prima di discutere dei risultati ottenuti presentiamo 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. Usare un'estensione significa estendere le funzionalità e aggiungere il supporto per determinate azioni. Di seguito vengono riportate alcune estensioni:

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

- ”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, istruzioni, load e store a virgola mobile a singola precisione.
- ”M” 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.
- **reserved** non ancora definite ma riservate per usi futuri.
- **non standard** non definita dalla RISC-V Foundation.

La tabella 2.1, presa da SPEC-2.2 [3], 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).

31	30	25 24	21	20	19	15 14	12 11	8	7	6	0	
		funct7		rs2		rs1	funct3		rd		opcode	R-type
		imm[11:0]			rs1	funct3		rd		opcode		I-type
		imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode		S-type
		imm[12]	imm[10:5]		rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
				imm[31:12]					rd		opcode	U-type
		imm[20]	imm[10:1]		imm[11]		imm[19:12]		rd		opcode	J-type

Figura 2.1: Formati istruzione RISC-V
[3]

In tutti i formati i registri sorgente (*rs1* e *rs2*) e il registro destinazione (*rd*) vengono mantenuti nelle stesse posizioni per velocizzare la codifica. I formati B e J possono essere visti come delle variazioni dei formati S e U nel senso che il campo imm dei formati B e J sono un ulteriore divisione dello stesso campo dei formati S e U, ad esempio il formato S, ha un solo campo imm compreso tra 25:31 bit per il formato B il campo è diviso in due ma mantenuto nella stessa posizione. L'ISA presenta 4 categorie di istruzioni:

- **Istruzioni computazionali:** Sono presenti 21 istruzioni computazionali e vengono codificate nel formato R se è un'operazione tra registri o nel formato I se è un'operazione tra registro e immediato. Le istruzioni di questo tipo includono istruzioni aritmetiche, logiche e di comparazione sia per valore senza segno che 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* 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 sono utilizzate 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) vengono identificati da x seguito da un numero. Il primo registro x0 contiene la costante zero e ogni istruzione che cerca di modificarlo solleva un'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* né un *return address* 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. Tutto ciò che è stato detto è valido anche per queste con alcuni accorgimenti che sollevano eccezioni di operazioni non valide. Rispettivamente il valore di XLEN per RV64I e per RV128I è 128.

Sul documento che propone RISC-V sul proprio ISA vengono anche impostate le linee guida per fare delle proprie estensioni.

Ho trovato i primi 2 capitoli molto più leggibili. Hai fatto bene a smaltire

Capitolo 3

Compilatori

3.1 Descrizione

Un compilatore è un programma che trasforma il codice sorgente in linguaggio macchina. Il motivo più comune per trasformare il codice sorgente è creare un programma eseguibile su una determinata macchina. Il processo di compilazione prevede diverse fasi. La prima 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 un codice intermedio. Questo codice intermedio attraversa una fase di ottimizzazione e infine viene generato il codice target.

Qualsiasi programma scritto in un linguaggio di programmazione di alto livello deve essere tradotto in codice oggetto prima di poter essere eseguito, quindi 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.

3.2 Storia

Il primo compilatore teorico fu pensato da Corrado Böhm che nel 1951 che lo sviluppò per la sua tesi di dottorato. Una prima implementazione di un

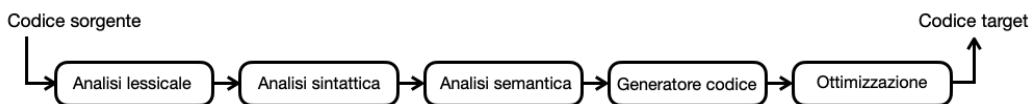


Figura 3.1: Schema meccanismo di compilazione

compilatore è dovuta a Grace Hopper che ha anche coniato il termine “compilatore”. Il primo compilatore , chiamato *A-0 System* , funzionava come caricatore o linker , non come i moderni compilatori. E’ importante menzionare che una versione successiva del *A-0 System* (la versione A-2 datata 1953) fu rilasciato ai clienti con lo scopo di sviluppare dei miglioramenti all’UNIVAC, quindi possiamo considerarlo come il primo software libero e open source della storia dell’informatica [7].

3.3 Cross-compilazione

Con il termine **Cross.compilazione** si intende la tecnica mediante la quale si utilizza un compilatore per compilare il codice sorgente e il codice generato ha come target un architettura diversa da quella su cui viene utilizzato il cross-compilatore. Alcuni esempi di una cross-compilazione può essere la compilazione di un applicazione per un sistema embedded.

3.4 Compilatori moderni

Nel ecosistema dei compilatori moderni i due dei più famosi sono: GCC e CLANG. Il primo GCC (GNU Compiler Collection) fu creato nel 1987 oggi viene sviluppato da programmatori di tutto il mondo. Inizialmente nato per il linguaggio C oggi supporta altri linguaggi come Java, C++, Objective C [1].

Il secondo nato nel 2005 sviluppato da Apple Inc. Inizialmente fu sviluppato con lo scopo di avere un compilatore Apple ottimizzato per i loro dispositivi[8].

Domanda: Ma tu hai usato CLANG?

3.5 Un buon compilatore

La scelta di un compilatore per un progetto è una scelta importante.I processori, oggi, sono strutturati in pipeline supescalari 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. 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 ci permette di concentrarci sul processo di programmazione, piuttosto che farci preoccupare della struttura del sistema e deve essere in grado di produrre del 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. Per RISC-V sono disponibili le *toolchain gcc*[6] e *toolchain clang*[4]. Entrambe le *toolchain* sono mantenute dalle community, quella più aggiornata e utilizzata è quella del progetto GNU. Sul sito sono elencate tutte le *toolchain* disponibili ma anche 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* attualmente è alla versione 15.0.2. A differenza della *toolchain gcc* è mantenuta solamente da Alex Bradbury.

Capitolo 4

BenchMarking

4.1 Descrizione board

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

4.2 Ambiente di sviluppo

La scheda di sviluppo D1-H viene fornita con il sistema Tina Linux. Il kernel fornito 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.

4.3 Presentazione programmi

I programmi utilizzati sono scritti in C e sono contenuti su github in una repository. Ogni directory contiene i dati di esecuzione di ciascun programma

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	lunghezza 85 mm, larghezza 56 mm, spessore 1,7 mm

Tabella 4.1: Caratteristiche della board

eseguito in un file con estensione .csv e un notebook jupyter per visualizzare i dati e, in alcuni casi, confrontarli.

4.3.1 Operazioni Aritmetiche

Il primo programma utilizzato esegue le operazioni aritmetiche di base addizione, sottrazione, moltiplicazione, divisione e modulo. Il programma genera due matrici quadrate di numeri interi su cui vengono eseguite le operazioni elemento per elemento. Le dimensioni delle matrici sono 1000 x 1000, 2500 x 2500, 5000 x 5000, 8000 x 8000, 10000 x 10000, 15000 x 15000.

La prima matrice generata ha valori tra uno e duecento, la seconda ha valori tra uno e cento.

Per ogni matrice vengono eseguite cinque prove da cui viene calcolato il tempo medio di esecuzione.

4.3.2 Prime Number

Il programma genera primi cinquemila numeri primi. Il programma "prime.c" controlla da due in poi se n-esimo numero è divisibile se non è divisibile allora lo memorizzo altrimenti passo al prossimo numero.

4.3.3 Moltiplicazione o shift

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).

4.3.4 Montecarlo Pi

Algoritmo di approssimazione del pigreco tramite il metodo di Montecarlo. Nell'esecuzione vengono utilizzati da 10 a (10^{10}) punti per l'approssimazione. Ogni iterazione viene eseguita cinque volte e per ogni iterazione viene memorizzato il valore calcolato con il proprio tempo di esecuzione. Il programma si basa sulla funzione rand.

4.3.5 Sorting

In questa directory sono presenti alcuni algoritmi di sorting. Gli algoritmi di sorting utilizzati sono:

- BubbleSort
- InsertionSort
- QuickSort
- HeapSort

Ogni algoritmo viene eseguito su array di dimensione diversa 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.

4.3.6 BackTracking

In questa directory sono contenuti due programmi che utilizzano metodi di backtracking:

- Un solutore di sudoku
- Un solutore di labirinti

In entrambi i casi vengono utilizzate tecniche di backtracking. Lo scopo è quello di valutare il tempo di esecuzione di programmi che fanno uso del backtracking su processore RISC.

Sudoku

Il solutore di sudoku risolve il puzzle matematico. Sono presenti otto sudoku nel file sudoku.h di varia difficoltà, 2 di essi non sono fattibili, uno per costruzione(sbagliato in partenza) e l'altro è impossibile risolverlo per la configurazione.

MazeSolver

Il solutore mazeSolver.c, una volta in esecuzione, chiede il path del labirinto da risolvere e successivamente è possibile salvare la soluzione, se trovata, in un nuovo file. Sono presenti alcuni esempi nella sottodirectory maze di labirinti. In directory è presente un ulteriore file mazeGen.py che è stato utilizzato come generatore di labirinti. Inoltre viene fornita la possibilità di utilizzare un proprio labirinto, il labirinto personale dovrà avere l'estensione .txt specificare il numero di linee e il numero di colonne del labirinto sulle prime due righe. Successivamente il labirinto. Le componenti del labirinto sono i muri , indicati con '#', un inizio , indicato con 'I' e una fine con 'O'. Le zone dello attraversabili sono idenficate dallo spazio ' '.

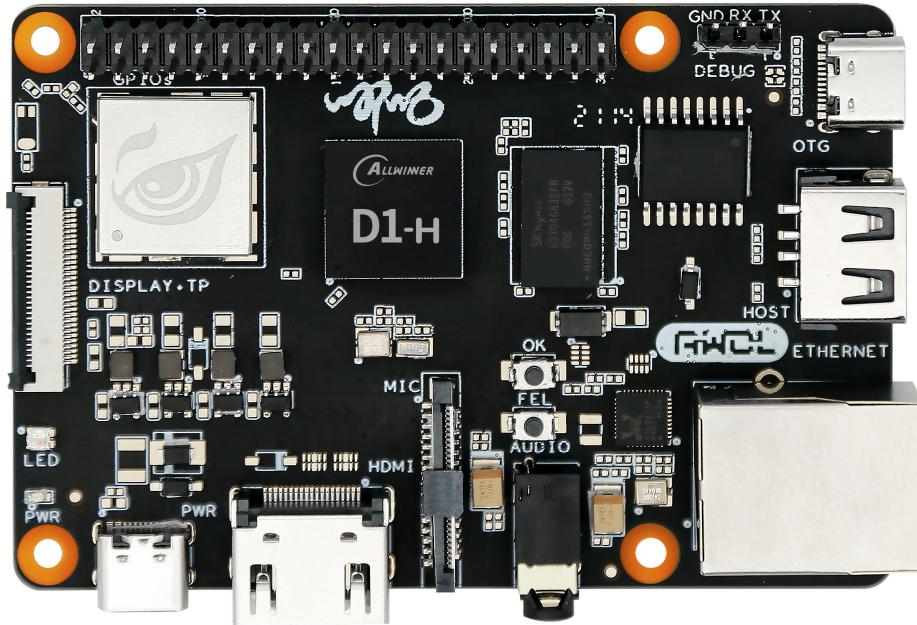


Figura 4.1: Board vista dall' alto

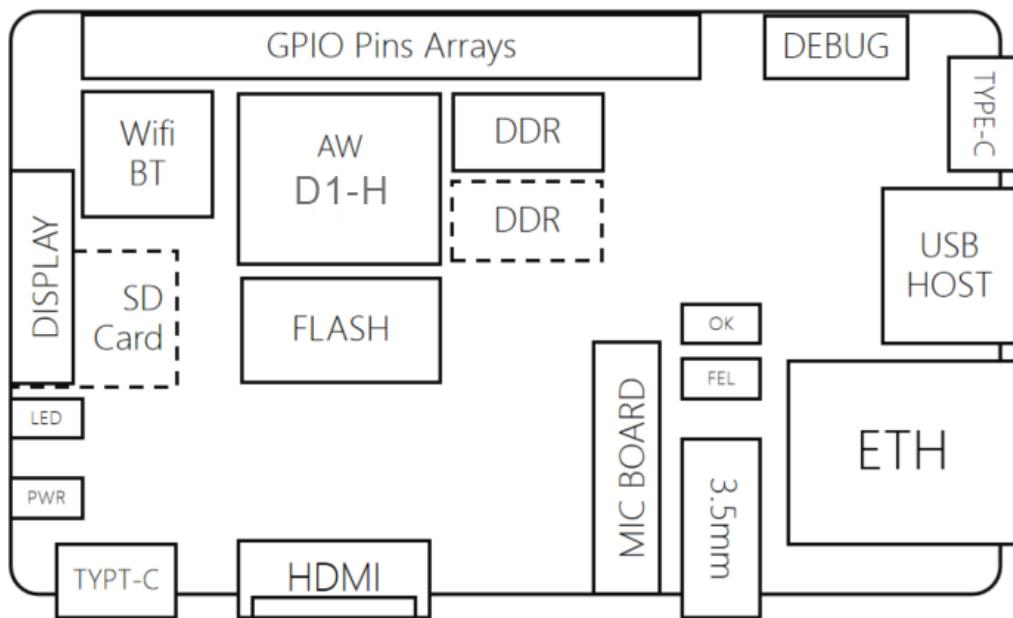


Figura 4.2: Schema a blocchi della scheda di sviluppo

[5]

Capitolo 5

Comparativa

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.

5.1 Operazioni

Con il programma operazioni viene visualizzato il tempo di esecuzione di alcune operazioni matematiche di base. Come mostrato nelle porzioni di codice Code 5.1, 5.2, 5.3, 5.4, 5.5 mostrano le funzioni che si occupano delle operazioni. Ogni funzione, semplicemente, esegue un ciclo su ogni elemento di a e di b , ne esegue l'operazione e memorizza il risultato in c . La matrice a ha valori interi compresi tra uno e duecento, la matrice b ha valori interi compresi tra uno e cento. Le matrici utilizzate sono matrici quadrate e le dimensioni utilizzate sono 1000, 2500, 5000, 8000, 10000, 15000. Per ogni operazione vengono eseguite cinque prove da cui viene calcolato il tempo medio di esecuzione. Per riferimento il programma è stato utilizzato anche sul Pc 5.

```
1 void sumMatrice(int dim, int a[dim][dim], int b[dim][dim])
2 {
3     for(int i = 0; i < dim; i++){
4         for(int j = 0 ;j < dim ; j++){
5             c[i][j] = a[i][j] + b[i][j];
6         }
7     }
```

```
8| }
```

Code 5.1: Addizione

```
1 void subMatrice(int dim, int a[dim][dim], int b[dim][dim])
2 {
3     for (int i = 0; i < dim; i++){
4         for (int j = 0; j < dim; j++){
5             c[i][j] = a[i][j] - b[i][j];
6         }
7     }
8 }
```

Code 5.2: Sottrazione

```
1 void multMatrice(int dim, int a[dim][dim], int b[dim][dim])
2 {
3     for (int i = 0; i < dim; i++){
4         for (int j = 0; j < dim; j++){
5             c[i][j] = a[i][j] * b[i][j];
6         }
7     }
8 }
```

Code 5.3: Moltiplicazione

```
1 void divMatrice(int dim, int a[dim][dim], int b[dim][dim])
2 {
3     for (int i = 0; i < dim; i++){
4         for (int j = 0; j < dim; j++){
5             c[i][j] = a[i][j] / b[i][j];
6         }
7     }
8 }
```

Code 5.4: Divisione

```
1 void modMatrice(int dim, int a[dim][dim], int b[dim][dim])
2 {
3     for (int i = 0; i < dim; i++){
4         for (int j = 0; j < dim; j++) {
5             c[i][j] = a[i][j] % b[i][j];
6         }
7     }
8 }
```

Code 5.5: Modulo

Di seguito vengono riportati i tempi di esecuzione medi per operazione e per dimensione

Operazione	1000	2500	5000	8000	10000	15000
Somma	0.002317	0.013848	0.055129	0.142010	0.227784	0.504650
Sottrazione	0.002324	0.013921	0.054900	0.142941	0.227505	0.504767
Prodotto	0.002344	0.014010	0.055036	0.143293	0.226141	0.504829
Divisione	0.002352	0.013953	0.055298	0.142226	0.227948	0.505007
Modulo	0.002298	0.013880	0.055009	0.142129	0.227835	0.505035

Tabella 5.1: Tempi di esecuzione MacBook-Air

Operazione	1000	2500	5000	8000	10000	15000
Somma	0.011256	0.069917	0.279804	0.714786	1.117411	2.512942
Sottrazione	0.011237	0.069850	0.279895	0.714680	1.117352	2.512978
Prodotto	0.011234	0.069899	0.279219	0.715381	1.117296	2.514183
Divisione	0.011170	0.069781	0.279884	0.714673	1.116763	2.513539
Modulo	0.011212	0.069845	0.279913	0.715371	1.117348	2.514198

Tabella 5.2: Tempi di esecuzione RISC-V

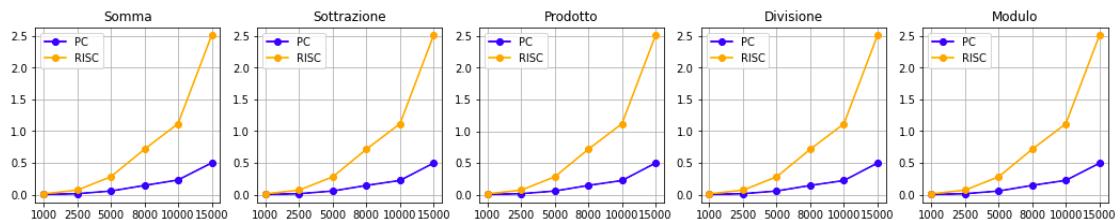
Di seguito i grafici rappresentativi delle tabelle precedenti.

Prendiamo in considerazione l'operazione di somma del processore RISC.

Il tempo di esecuzione per la matrice 1000×1000 è di 0.011256 e per la matrice 2500×2500 è di 0.069917 il che è 6.2115 (circa) il tempo di esecuzione e 6.25 la dimensione dei dati. Per il Pc il tempo di esecuzione per la matrice 1000×1000 è di 0.002317 e per la matrice 2500×2500 è di 0.013848 il che è 5.97 (circa) il tempo di esecuzione e 6.25 la dimensione dei dati. Il tempo di esecuzione totale del programma è di 23.653595 per il Pc mentre 117.665045 per RISC. Quindi il MacBook è 4.97 % più veloce del processore RISC.

5.2 PrimeNumber

Il programma utilizza una scansione lineare per trovare i primi cinquemila numeri primi. In seguito viene mostrato il codice della funzione principale. La funzione controlla dal numero due in avanti se il numero è primo, se lo è



viene inserito nell' array altrimenti no e il ciclo viene eseguito fino a trovare cinquemila numeri primi.

```

1 int DIM = 5000;
2
3 void fillPrime(int *a){
4     int num = 2;
5     int pos = 0;
6     while(pos < DIM){
7
8         if( divisibile(a, num) == 0){
9             a[pos] = num;
10            pos++;
11        }
12
13        num++;
14    }
15 }
```

Code 5.6: Impostazioni dei dati

I file sono stati compilato con diverse livelli di ottimizzazione. Come pre-

	PC	RISC-V
-O2	0.041286	0.434372
-O1	0.033024	0.346976
-O0	0.032363	0.3122784

Tabella 5.3: Tempi di esecuzione numeri primi

Cedentemente notato è l'ordine di grandezza di differenza dei due processori. Andando a guardare nei sorgenti assembly notiamo delle differenze. La prima abbastanza ovvia è la dimensione del file che nel caso del' ottimizzazione 0 è superiore (172 linee 4KB) rispetto all livello 2 (130 linee 4KB). Il secondo è l'utilizzato dell'istruzione *nop* nel livello 0. Il codice 5.7 fa riferimento alla porzione di codice dove vengono utilizzate le istruzioni *nop*.

Code 5.7: Sorgente prime ottimizzato con -O0

```

add    a5,a4,a5
lw     a5,0(a5)
bne   a5,zero,.L4
nop
nop
lw     ra,44(sp)
lw     so,40(sp)
addi  sp,sp,48
```

La pseudo-istruzione *nop*(No operation) ha come implementazione addi x0, x0, 0 che somma zero (costante) al registro contenente zero (x0) e lo memorizza nel registro x0. Questo è un caso eccezionale in cui si ha la possibilità di scrivere il registro x0, seppur con poco significato algebricamente,

ogni altro caso in cui si vuole modificare il registro x0 solleverebbe errori. La motivazione della presenza di queste operazioni è dovuto alla struttura hardware dotata di interlock. La maggior parte delle implementazioni hardware ha degli interlock. Se un'istruzione necessita di dati che non sono ancora disponibili, la pipeline si interrompe finché i dati non sono disponibili. Oppure, se si tratta di una pipeline di esecuzione fuori ordine, un'istruzione si blocca finché i dati non sono disponibili e altre istruzioni possono continuare l'esecuzione. I nop emessi dal compilatore con -O0 non vengono emessi a -O1 o superiore.

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 {
5     int *a = malloc(N * sizeof(int));
6     for (int i = 0; i < N; i++)
7     {
8         a[i] = rand();
9     }
10    return a;
11}
12
13 int *esponenti(int *a)
14 {
15     int *e = malloc(N * sizeof(int));
16     for (int i = 0; i < N; i++)
17     {
18         e[i] = (int)log2(rand());
19     }
20    return e;
21}
```

Code 5.8: 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
12     for (int i = 0; i < N; i++)
13     {
14         res[i] = a[i] / b[i];
15     }
16
17 }
```

Code 5.9: Impostazioni dei dati

```

1 void eseguiMultShift(int *a, int *b, int *res)
2 {
3
4     for (int i = 0; i < N; i++)
5     {
6         res[i] = a[i] << b[i];
7     }
8
9 }
10
11
12 void eseguiDivShift(int *a, int *b, int *res)
13 {
14
15     for (int i = 0; i < N; i++)
16     {
17         res[i] = a[i] >> b[i];
18     }
19 }
```

Code 5.10: Impostazioni dei dati

Il codice 5.8 imposta i due array. I codici 5.9 e 5.10 mostrano l'operazione eseguita. La tabella 5.4 mostra i tempi di esecuzione calcolati in millisecondi delle varie operazioni.

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

Tabella 5.4: Tempi di esecuzione operazioni calcolati in ms

5.4 Analisi codice assembly

Osserviamo i compilatori a confronto. 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 sul 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).

5.5 Operazioni

5.5.1 Addizione con costante

```

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

Code 5.11: Addizione

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

RISC-V

Il sorgente compilato con il compilatore RISC-V 5.12 da riga due fino a riga sei predispone la chiamata della procedura posizionando sullo stack il 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, vengono ignorati gli errori di overflow. Successivamente tramite la pseudo istruzione sext.w che 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 restituisce il controllo al chiamante.

ARM

Il sorgente 5.14 , compilato con gcc ARM, mostra che la preparazione della procedura si esegue da riga due a riga cinque, le 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 5.13 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 che a riga sei viene sommato a ventitré che viene memorizzato nel registro eax. Infine viene ridato il controllo al chiamante.

Code (5.12) 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 (5.13) 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
```

Code (5.14) 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 5.1: Funzione di somma

5.5.2 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 5.15: Addizione generale

RISC-V

Nel caso del compilatore RISC-V la somma avviene tra tredici e diciannove dove gli operandi vengono caricati nei registri a4 e a5, 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 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 (5.16) 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
    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 (5.17) 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
```

Code (5.18) 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
```

Figura 5.2: Funzione di somma

5.5.3 Moltiplicazione

Moltiplicazioni per potenze di 2

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

Code 5.19: 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 per le rispettive preparazioni precedenti.

Nei sorgenti in figura 5.2 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

Code (5.20) 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 (5.21) 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 (5.22) 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 5.3: Moltiplicazione per 2

Code (5.23) 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 (5.24) 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 (5.25) 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 5.4: Moltiplicazione per 8

potenza di due le operazioni avvengono tutte tramite shift left di un opportuno valore. Con la figura 5.4 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 molto simili, il primo moltiplica il numero per trentuno che rappresenta più in generale un numero che dista da una potenza di due di uno. Il secondo è un caso più generale dove avviene la moltiplicazione di un numero non potenza di due e che dista da una potenza almeno di due, nel nostro caso il numero è trenta.

```

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

```

Code (5.26) moltiplicazione per 31

```

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

```

Code (5.27) 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 , il numero successivo alla potenza di due, l' operazione di sottrazione viene sostituita con una di addizione. Nel caso più generale invece 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 avviene prima uno shift di quattro (moltiplicazione per sedici) successivamente sottratto una volta il numero e infine al risultato avviene 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 opportuni shift combinate con addizioni e sottrazioni per ottenere il valore della costante.

Code (5.28) 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 (5.29) 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    lr
```

Code (5.30) 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 5.6: Moltiplicazione per 31

Code (5.31) 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($0)
    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 (5.32) 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 (5.33) 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 5.7: Moltiplicazione per 30

5.5.4 Divisione

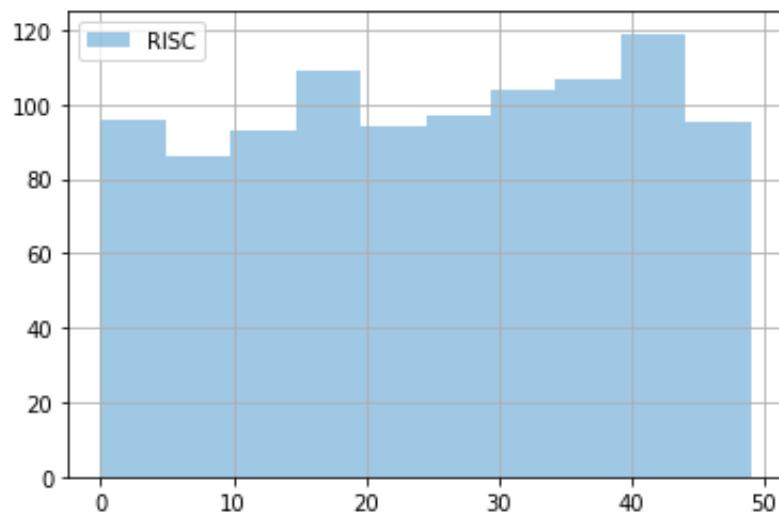
Per quanto riguarda la divisione viene utilizzato la stessa metodologia della moltiplicazione.

5.6 MonteCarloPi

Al giorno d'oggi è comune utilizzare la generazione di numeri casuali in tecniche o simulazioni, ad esempio numeri di conto bancario e crittografia. È una tecnica necessaria e di base nella programmazione di computer. Non è possibile avere un generatore scelto tra tanti senza valutarne la casualità. Ad esempio un generatore che genera dei valori tra zero e cento ma con probabilità maggiore su uno specifico valore, è normale pensare che se si utilizza più probabilmente verrà generato quel numero di tutti gli altri. Un altro esempio se utilizzassimo un generatore che ha una probabilità uniforme nel generare numeri generiamo i primi dieci valori e questi sono i valori da uno a dieci notiamo subito una sequenza riconoscibile e la casualità del prossimo numero è poco casuale. Quindi le proprietà che ci interessano per un generatore di numeri sono una distribuzione uniforme e la non predicitività.

5.6.1 Generatore casuale

Per il programma è stato utilizzata la funzione rand() del linguaggio c. Per la valutazione dei valori sono state generati i valori da zero a novantanove, estremi inclusi, per cinque volte. I campioni non presentano predicitività e seguono una distribuzione normale. Possiamo concludere che il generatore è un generatore accettabile.



Un utilizzo del generatore di numeri casuali è un algoritmo di Monte Carlo. Il metodo di Monte Carlo è un'ampia classe di metodi computazionali basati sul campionamento casuale per ottenere risultati numerici. In particolare si è utilizzato il metodo applicato al calcolo del pigreco. Il metodo utilizzato funziona nel seguente modo:

- 1 Genera due numeri casuali tra 0 e 1 che rappresentano le coordinate di un punto.
- 2 Applica il teorema di Pitagora, se supera uno allora il punto non appartiene alla circonferenza, altrimenti appartiene.
- 3 Ripeti i punti 1,2.
- 4 Calcola il valore del pigreco.

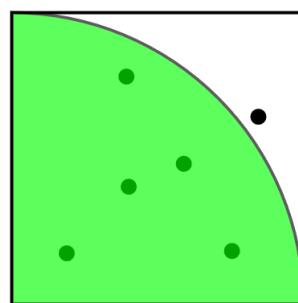


Figura 5.8: Esempio di coordinate generate casualmente

$$\pi = 4 * \frac{\text{Punti interni alla circonferenza}}{\text{Numerodi punti generati}}$$

```

1 double valuePi(int times)
2 {
3     int i, count;
4     double x, y, eq, pi;
5
6     for (i = 0; i < times; ++i)
7     {
8         x = ((double)rand() / RAND_MAX);
9         y = ((double)rand() / RAND_MAX);
10
11         eq = x * x + y * y;
12         if (eq <= 1)
13         {
14             count++;
15         }
16     }
17
18     pi = (long double)4 * (long double)count / (long double)
19     ↪ times;
20     return pi;
}

```

Code 5.34: Funzione per il calcolo del pigreco con il metodo di Monte Carlo

In seguito vengono riportate le tabelle che rappresentano il tempo di esecuzione e il valore calcolato.

	1	2	3	4	5	T.EXE Totale	MF
10^1	0,000008	0,000008	0,000006	0,000005	0,000006	0,000033	0,0
10^2	0,000019	0,000019	0,000018	0,000018	0,000018	0,000092	0,0
10^3	0,000141	0,000140	0,000141	0,000140	0,000140	0,000702	0,0
10^4	0,001467	0,001501	0,001524	0,001362	0,001382	0,007236	0,0
10^5	0,014029	0,014082	0,013851	0,013994	0,013971	0,069927	0,0
10^6	0,138878	0,138625	0,138920	0,138916	0,138660	0,693999	0,1
10^7	1,393739	1,387346	1,387775	1,387397	1,390657	6,946914	1,3
10^8	13,881088	13,880401	13,882388	13,880326	13,880459	69,404662	13,8
10^9	138,820115	138,830064	138,831942	138,863054	138,809745	694,154920	138,
10^{10}	195,756279	195,752334	195,733842	195,746235	195,755829	978,744519	195,

Tabella 5.5: Tempi di esecuzione dell'algoritmo

Le tabelle 5.5 e 5.6 mostrano i risultati in termini di valori calcolati e il tempo di esecuzione. Le colonne indicano il valore dell*i*-esima prova mentre ogni riga identifica il numero di punti generati.

	1	2	3	4	5	MEDIA	DEV.STD
10^1	3,600000	3,200000	2,400000	3,200000	3,200000	3,120000	0,438178
10^2	2,840000	3,160000	3,040000	3,160000	3,000000	3,040000	0,132665
10^3	3,100000	3,136000	3,236000	3,168000	3,192000	3,166400	0,052046
10^4	3,160000	3,128000	3,157200	3,110400	3,156400	3,142400	0,022114
10^5	3,134120	3,138920	3,145400	3,143120	3,143200	3,140952	0,004482
10^6	3,140340	3,140840	3,145420	3,139996	3,143304	3,141980	0,002319
10^7	3,142072	3,141478	3,142091	3,141413	3,141392	3,141689	0,000360
10^8	3,141343	3,141380	3,141467	3,141383	3,141244	3,141363	0,000081
10^9	3,141499	3,141617	3,141583	3,141508	3,141531	3,141548	0,000051
10^{10}	3,141637	3,141616	3,141607	3,141607	3,141506	3,141595	0,000051

Tabella 5.6: Valori calcolati

5.7 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 come segue.

- La complessità temporale. Si basano su quanti valori si ha da distribuire. Questi vengono indicati con n . Possiamo avere una prestazione buona come $\mathcal{O}(n \log n)$ oppure peggiori come $\mathcal{O}(n!)$.
- Memoria utilizzata.
- Stabilità ovvero se viene preservato l’ordine relativo dei dati con chiavi uguali all’interno dei valori da ordinare.

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.

5.7.1 Bubble sort

Bubble sort è un semplice algoritmo di ordinamento. L’algoritmo funziona nel modo seguente:

- Confronta gli elementi adiacenti, se il primo è maggiore del secondo, scambialo.
- Fai lo stesso confronto di prima dalla prima coppia all'ultima coppia.
- Ripeti i 2 passaggi precedenti per tutti gli elementi tranne l'ultimo.
- Ripeti tutti i 3 passaggi precedenti finché non sono necessari elementi da scambiare.

L'efficienza dell'algoritmo è basata sull' ordinamento dell'array da ordinare. Nel caso migliore l'array è già ordinato e, passandolo all'algoritmo, esegue solo il punto 1 dall'inizio alla fine. In contrario il caso peggiore è quando l'array si trova nell'ordine non crescente.

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

Tabella 5.7: complessità bubble sort

Di seguito viene riportato il codice (Code 5.35) dell'algoritmo e i tempi di esecuzione (Tab 5.8) secondo la dimensione dell'array e la situazione iniziale dell'array:

```

1 void bubbleSort(int a[], int dim)
2 {
3     int temp;
4     for (int i = 0; i < dim; i++)
5     {
6         for (int j = 0; j < dim - i - 1; j++)
7         {
8             if (a[j] > a[j + 1])
9             {
10                 temp = a[j];
11                 a[j] = a[j + 1];
12                 a[j + 1] = temp;
13             }
14         }
15     }
16 }
```

Code 5.35: 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 5.8: Tempi di esecuzione bubble sort

5.7.2 Insertion sort

L'arlogitmo consiste nel considerare un elemento alla volta, inserendo ciascuno nella posizione corretta tra gli elementi che sono stati ordinati.

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

Tabella 5.9: complessita insertion sort

```

1 void insertionSort(int a[], int dim)
2 {
3
4     int temp, j;
5     for (int i = 1; i < dim; i++)
6     {
7         temp = a[i];
8         j = i - 1;
9         while (j >= 0 && a[j] > temp)
10        {
11            a[j + 1] = a[j];
12            j--;
13        }
14        a[j + 1] = temp;
15    }
16 }
```

Code 5.36: 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 5.10: Tempi di esecuzione insertion sort

5.7.3 QuickSort

L'algoritmo quicksort è un algoritmo ricorsivo del tipo divide et impera. L'algoritmo si basa:

- Scegli un pivot dagli elementi dell'array.
- Ordina l'array. Se l'elemento è più grande del pivot, allora mettilo dopo il pivot, altrimenti prima.
- Ripetere questi due passaggi per i sottoarray finché ogni elemento non è nell'ordine corretto.

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

```

1 void QuickSort(int v[], int in, int fin)
2 {
3     if (fin <= in)
4         return;
5     int pos = partiziona(v, in, fin);
6
7     QuickSort(v, in, pos - 1);
8     QuickSort(v, pos + 1, fin);
9 }
10
11 int partiziona(int v[], int in, int fin)
12 {
13
14     int i = in + 1, j = fin;
15     while (i <= j)
16     {
17         while ((i <= fin) && (v[i] <= v[in]))
18             i++;
19

```

```

20     while (v[j] > v[in])
21         j--;
22     if (i <= j)
23     {
24
25         int t = v[i];
26         v[i] = v[j];
27         v[j] = t;
28     }
29 }
30
31 int tt = v[in];
32 v[in] = v[i - 1];
33 v[i - 1] = tt;
34
35 return i - 1;
36 }
```

Code 5.37: 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 5.11: Tempi di esecuzione quick sort

5.7.4 Heap sort

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

```

1 void swap(int *a, int *b)
2 {
3     int temp = *a;
4     *a = *b;
5     *b = temp;
6 }
7
8 void heapify(int arr[], int n, int i)
9 {
```

```

10 int largest = i;
11 int left = 2 * i + 1;
12 int right = 2 * i + 2;
13
14 if (left < n && arr[left] > arr[largest])
15     largest = left;
16
17 if (right < n && arr[right] > arr[largest])
18     largest = right;
19
20 if (largest != i)
21 {
22     swap(&arr[i], &arr[largest]);
23     heapify(arr, n, largest);
24 }
25
26
27 void heapSort(int arr[], int n)
28 {
29     for (int i = n / 2 - 1; i >= 0; i--)
30         heapify(arr, n, i);
31
32     for (int i = n - 1; i >= 0; i--)
33     {
34         swap(&arr[0], &arr[i]);
35         heapify(arr, i, 0);
36     }
37 }
38

```

Code 5.38: 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 5.12: Tempi di esecuzione heap sort

Come mostrato dai grafici 5.9 i risultati mostrano che il peggior algoritmo è il bubblesort mentre il migliore è quicksort. Risultato poco notevole, nella sezione successiva un confronto più significativo.

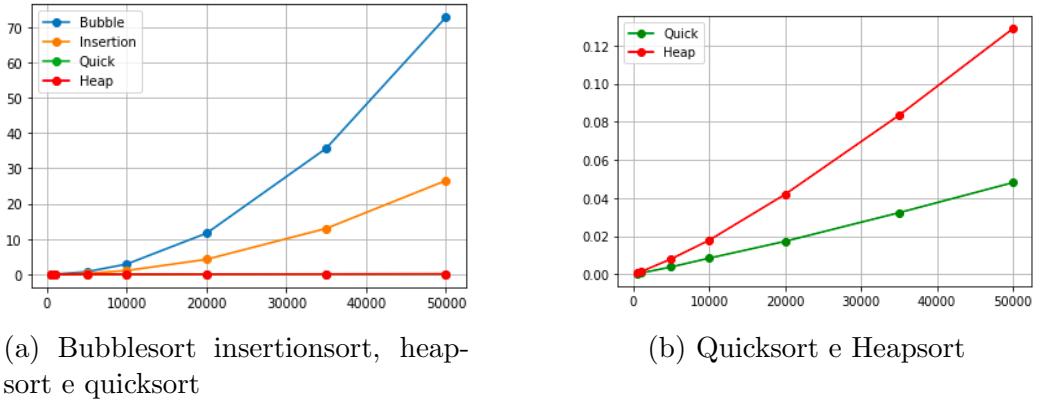


Figura 5.9: Algoritmi di ordinamento a confronto

5.7.5 Confronto Sorting

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

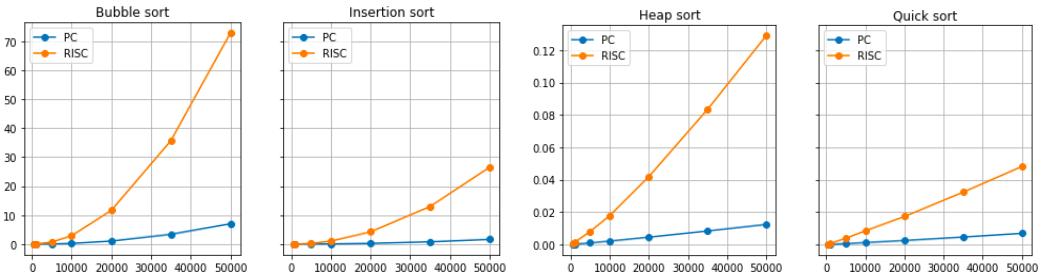


Figura 5.10: Algoritmi di ordinamento a confronto eseguiti su PC e su RISC-V

La tabella 5.13 mostra i tempi di esecuzione degli algoritmi di ordinamento eseguiti sul PC. Un altro confronto lo possiamo fare con Raspberry pi¹.

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 da 100 MB porta. In aggiunta a ciò, ci sono pin di input/output (GPIO) per uso generico per collegare alcuni hardware. La tabella 5.14 mostra i tempi di esecuzione degli algoritmi di ordinamento eseguiti su Raspberry [2].

¹Il modello utilizzato è il modello Raspberry model B mostrato in figura 5.11

	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 5.13: Tempi di esecuzione degli algoritmi di sorting su PC



Figura 5.11: Vista superiore Raspberry model B

I grafici 5.12 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. Prendendo in considerazione il Bubblesort nel caso con 50000 elementi la board con processore RISC-V è 4 volte più veloce del raspberry in altri casi, come ad esempio per heap sort, il miglioramente di RISC-V è 1.6 rispetto a Raspberry.

5.8 BackTracking

Un altro problema che si può affrontare è il problema del backtracking, ovvero quella tecnica per cui è necessario tornare su dei passi precedenti per trovare la soluzione al problema. Il primo programma utilizzato è un solutore di

	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 5.14: Tempi di esecuzione Raspberry Pi B

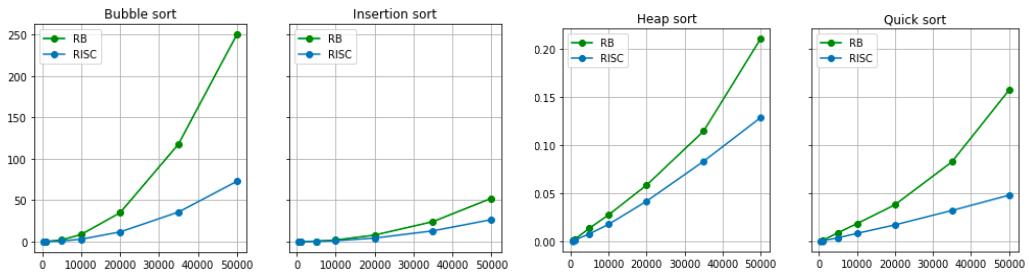


Figura 5.12: Algoritmi di ordinamento a confronto eseguiti su Raspberry e su RISC-V

sudoku. I primi sudoku sono di difficoltà crescente mentre gli ultimi sono impossibili, uno impossibile per costruzione (sbagliato dalla partenza) , e l’altro impossibile completarlo.

Il secondo problema che utilizza il backtracking è un solutore di labirinti.

	RISC-V	PC
1	0.004475	0.000384
2	0.055382	0.004791
3	0.000282	0.000026
4	0.048991	0.005037
5	0.297553	0.025939
6	1.488701	0.122498
7	6.548320	0.534505
8	0.001508	0.000241
Vuoto	0.001680	0.000267

Tabella 5.15: Tempi di esecuzione del solutore di sudoku

	RISC	PC
maze0	0.000102	0.000007
maze1	0.005261	0.000235
maze2	0.009123	0.000371

Tabella 5.16: Tempi di esecuzione del solutore di labirinti

Capitolo 6

Conclusione

In questa tesi si sono fatti alcuni test su RISC-V. Si è controllato aspetti comuni come le operazioni aritmetiche e alcuni algoritmi di ordinamento. Si è guardato il codice assembly generato dal compilatore e osservato le implementazioni delle varie operazioni comparandolo con altri codici assembly. Oltre alla comparazione con altri codici assembly si è comparato con diversi livelli di ottimizzazione. Per ogni operazione è stato calcolato il tempo di esecuzione. Infine sono stati utilizzati degli algoritmi di ordinamento i cui risultati sono stati confrontati con un Raspberry.

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.

Bibliografia

- [1] Brian J. Gough. *An Introduction to GCC*. [Online]. 2005. URL: <https://web.archive.org/web/20100531171042/http://www.network-theory.co.uk/docs/gccintro/>.
- [2] Yuan Yue. *Performance Evaluation of Sorting Algorithms in Raspberry Pi and Personal Computer*. Rapp. tecn. Universita di Vaasa, 2015.
- [3] *riscv-spec-v2.2*. 2017. URL: <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>.
- [4] SiFive. *riscv-llvm*. 2019. URL: <https://github.com/sifive/riscv-llvm>.
- [5] Awol. *Introduzione del chip D1-H*. 2021. URL: <https://d1.docs.awol.com>.
- [6] RISC-V Software Collaboration. *riscv-gcc*. 2021. URL: <https://github.com/riscv-collab/riscv-gcc>.
- [7] 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.
- [8] Wikipedia. *Clang*. [Online; sito ufficiale: <https://clang.llvm.org/>]. 2022. URL: <https://it.frwiki.wiki/wik/Clang>.