

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

**TITOLO**  
**TITOLO**  
**TITOLO**

**Relatore:** Trentini Andrea

**Correlatore:** Carraturo Alexjan

**Tesi di Laurea di:** Bianchessi Mattia  
**Matr. 931455**

**Anno Accademico xxxxxxxx**

# Todo list

■ Introduzione . . . . .	2
■ Rileggi/Correggi capitolo 1: RISC-V . . . . .	10
■ Rileggi/Correggi capitolo 2: ISA. . . . .	12
■ Better write . . . . .	18
■ Sezione/ due parole sulle pseudo istruzioni . . . . .	24
■ Rileggi/Correggi capitolo 3: Compilatore. . . . .	29
■ Reference section cross-compiler . . . . .	38
■ Componenti della toolchain ? . . . . .	38
■ Completa il capitolo 4 . . . . .	41
■ repo . . . . .	41
■ Analizza costo algoritmo . . . . .	43
■ Info Sieve of Atkin . . . . .	43
■ Valutazione generatore casuale . . . . .	43
■ Info argoritmi . . . . .	44
■ Ulteriori Info . . . . .	46
■ CALCOLA MIPS . . . . .	48
■ Inserisci sorgenti Addizione generale . . . . .	52
■ add ref para . . . . .	55
■ Controllo eff pseudo istruzioni . . . . .	57
■ Inserisci valutazione generatore numeri casuali . . . . .	57
■ info algo . . . . .	64
■ Inserisci immagine . . . . .	65
■ More . . . . .	67
■ More . . . . .	68
■ rileggi . . . . .	70

# Introduzione

Introduzione

Introduzione  
...

# Indice

<b>1 RISC-V</b>	<b>10</b>
1.1 Dal codice alle istruzioni . . . . .	10
1.2 progetto RISC-V . . . . .	11
<b>2 ISA RISC-V</b>	<b>12</b>
2.1 ISA Overview . . . . .	12
2.1.1 Codifica Istruzione . . . . .	12
2.1.2 Memorizzazione . . . . .	12
2.1.3 Cambio di flusso . . . . .	13
2.1.4 Nomi delle estensioni . . . . .	13
2.2 ISA Base . . . . .	15
2.3 RV32I . . . . .	15
2.3.1 Formato istruzioni . . . . .	17
2.3.2 Istruzioni per computazione intera . . . . .	17
2.3.3 Istruzioni registro-immediate . . . . .	18
2.3.4 Operazioni registro-registro . . . . .	19
2.3.5 Controllo del flusso . . . . .	20
2.3.6 Load e Store . . . . .	21
2.3.7 Controllo e i registri di stato . . . . .	22
2.4 Alcune estensioni . . . . .	24
2.4.1 Estensione M . . . . .	24
2.4.2 Estensione F . . . . .	25
2.4.3 E per Embedded . . . . .	27
2.5 Altri Set . . . . .	28
<b>3 Compilatori</b>	<b>29</b>
3.1 Descrizione . . . . .	29
3.2 Storia . . . . .	30
3.3 Cross-compilazione . . . . .	30
3.4 Compilatori moderni . . . . .	31
3.5 Un buon compilatore . . . . .	31

3.6	CLANG e GCC . . . . .	32
3.6.1	Vantaggi dei compilatori . . . . .	35
3.6.2	Community . . . . .	36
3.7	Toolchain di RISC-V . . . . .	38
<b>4</b>	<b>BenchMarking</b>	<b>41</b>
4.1	Descrizione board . . . . .	41
4.2	Ambiente di sviluppo . . . . .	41
4.3	presentazione programmi . . . . .	41
4.3.1	Operazioni Aritmetiche . . . . .	42
4.3.2	Prime Number . . . . .	43
4.3.3	Moltiplicazione o shift . . . . .	43
4.3.4	Montecarlo Pi . . . . .	43
4.3.5	Sorting . . . . .	43
4.3.6	BackTracking . . . . .	44
<b>5</b>	<b>Comparativa</b>	<b>46</b>
5.1	Operazioni . . . . .	46
5.2	PrimeNumber . . . . .	49
5.3	MultOrShift . . . . .	49
5.4	analisi codice assembly . . . . .	51
5.5	Operazioni . . . . .	51
5.5.1	Addizione con costante . . . . .	51
5.5.2	Addizione . . . . .	52
5.5.3	Moltiplicazione . . . . .	53
5.5.4	Divisione . . . . .	57
5.6	MonteCarloPi . . . . .	57
5.7	Sorting . . . . .	58
5.7.1	Bubble sort . . . . .	60
5.7.2	Insertion sort . . . . .	62
5.7.3	QuickSort . . . . .	63
5.7.4	Heap sort . . . . .	64
5.7.5	Confronto Sorting . . . . .	65
5.8	BackTracking . . . . .	68
<b>6</b>	<b>Progetti</b>	<b>69</b>
<b>7</b>	<b>Dibattito ARM vs RISC-V</b>	<b>70</b>
<b>8</b>	<b>Conclusione</b>	<b>72</b>

# Elenco delle figure

2.1	Lunghezza Istruzioni RISC-V . . . . .	13
2.2	Memorizza istruzioni . . . . .	13
2.3	Registri architettura . . . . .	16
2.4	Formati istruzione RISC-V . . . . .	17
2.5	Istruzioni registro-Immediato nel formato R . . . . .	18
2.6	Istruzioni registro-Immediato nel formato I . . . . .	19
2.7	Istruzioni registro-Immediato nel formato U . . . . .	19
2.8	Istruzioni registro-registro . . . . .	20
2.9	Istruzioni per salto incondizionato . . . . .	20
2.10	Istruzioni per salto condizionato . . . . .	21
2.11	Istruzioni load e store . . . . .	22
2.12	Istruzioni CSR . . . . .	23
2.13	Time e counter . . . . .	24
2.14	Istruzioni moltiplicazione e divisione dell'estensione M . . . . .	25
2.15	Istruzioni Single-Precision registro-register nel formato R . . . . .	26
2.16	Istruzioni Single-Precision registro-register in un formato speciale . . . . .	26
3.1	Schema meccanismo di compilazione . . . . .	29
3.2	Loghi dei compilatori, GCC (a sinistra) CLANG (a destra) . . . . .	33
3.3	Tempo di compilazione . . . . .	34
3.4	Performance di esecuzione . . . . .	35
3.5	Confronto dimensione dei file generati dai compilatori . . . . .	39
3.6	Confronto dimensione . . . . .	40
4.1	Board vista dall' alto . . . . .	45
4.2	Schema a blocchi della scheda di sviluppo . . . . .	45
5.1	Funzione di somma . . . . .	52
5.2	Funzione di somma . . . . .	53
5.3	Moltiplicazione per 2 . . . . .	54
5.4	Moltiplicazione per 8 . . . . .	54

5.6	Moltiplicazione per 31 . . . . .	56
5.7	Moltiplicazione per 30 . . . . .	56
5.8	Algoritmi di ordinamento a confronto . . . . .	66
5.9	Algoritmi di ordinamento a confronto eseguiti su PC e su RISC-V . . . . .	66
5.10	Algoritmi di ordinamento a confronto eseguiti su Raspberry e su RISC-V . . . . .	68

# Elenco delle tabelle

2.1	Tabella nomenclatura ISA RISC-V . . . . .	14
2.2	Campo fnt . . . . .	27
2.3	Formato del risultato di FCLASS . . . . .	27
3.1	Dimensione media dei compilatori . . . . .	39
3.2	Dimensione media . . . . .	39
4.1	Caratteristiche della board . . . . .	42
5.1	Tempi di esecuzione MacBook-Air . . . . .	47
5.2	Tempi di esecuzione RISC-V . . . . .	48
5.3	Tempi di esecuzione operazioni calcolati in ms . . . . .	50
5.4	Tempi di esecuzione dell'algoritmo . . . . .	59
5.5	Valori calcolati . . . . .	59
5.6	complessita bubble sort . . . . .	60
5.7	Tempi di esecuzione bubble sort . . . . .	61
5.8	complessita insertion sort . . . . .	62
5.9	Tempi di esecuzione insertion sort . . . . .	62
5.10	Tempi di esecuzione quick sort . . . . .	64
5.11	Tempi di esecuzione heap sort . . . . .	65
5.12	Tempi di esecuzione degli algoritmi di sorting su PC . . . . .	67
5.13	Tempi di esecuzione Raspberry Pi B . . . . .	67
5.14	Tempi di esecuzione del solutore di sudoku . . . . .	68
5.15	Tempi di esecuzione del solutore di labirinti . . . . .	68

# Listings

5.1	Addizione . . . . .	46
5.2	Sottrazione . . . . .	46
5.3	Moltiplicazione . . . . .	47
5.4	Divisione . . . . .	47
5.5	Modulo . . . . .	47
5.6	Impostazioni dei dati . . . . .	49
5.7	Impostazioni dei dati . . . . .	49
5.8	Impostazioni dei dati . . . . .	50
5.9	Addizione . . . . .	51
	PorzioniCodice/ConfrontoCompilatori/Assembler/RISC-V/get_num.txt	52
	PorzioniCodice/ConfrontoCompilatori/Assembler/ARM/get_num.txt	52
	PorzioniCodice/ConfrontoCompilatori/Assembler/x86/get_num.txt	52
5.10	Addizione generale . . . . .	52
	PorzioniCodice/ConfrontoCompilatori/Assembler/RISC-V/sumGen.txt	53
	PorzioniCodice/ConfrontoCompilatori/Assembler/ARM/sumGen.txt	53
	PorzioniCodice/ConfrontoCompilatori/Assembler/x86/sumGen.txt	53
5.11	Moltiplicazione per potenza di 2 . . . . .	53
	PorzioniCodice/ConfrontoCompilatori/Assembler/RISC-V/mult2.txt	54
	PorzioniCodice/ConfrontoCompilatori/Assembler/ARM/mult2.txt .	54
	PorzioniCodice/ConfrontoCompilatori/Assembler/x86/mult2.txt .	54
	PorzioniCodice/ConfrontoCompilatori/Assembler/RISC-V/mult8.txt	54
	PorzioniCodice/ConfrontoCompilatori/Assembler/ARM/mult8.txt .	54
	PorzioniCodice/ConfrontoCompilatori/Assembler/x86/mult8.txt .	54
	PorzioniCodice/ConfrontoCompilatori/Assembler/RISC-V/mult31.txt	56
	PorzioniCodice/ConfrontoCompilatori/Assembler/ARM/mult31.txt	56
	PorzioniCodice/ConfrontoCompilatori/Assembler/x86/mult31.txt .	56
	PorzioniCodice/ConfrontoCompilatori/Assembler/RISC-V/mult30.txt	56
	PorzioniCodice/ConfrontoCompilatori/Assembler/ARM/mult30.txt	56
	PorzioniCodice/ConfrontoCompilatori/Assembler/x86/mult30.txt .	56
5.14	Funzione per il calcolo del pigreco con il metodo di Monte Carlo	58
5.15	Algoritmo bubble sort scritto in c . . . . .	60

5.16 Algoritmo bubble sort scritto in c . . . . .	62
5.17 Algoritmo quick sort scritto in c . . . . .	63
5.18 Algoritmo heap sort scritto in c . . . . .	64

# Capitolo 1

## RISC-V

Rileggi/Correggi  
capitolo 1:  
RISC-V ...

### 1.1 Dal codice alle istruzioni

**Computer** : Apparecchio elettronico in grado di svolgere operazioni matematiche e logiche e di memorizzare informazioni a una velocità e in una quantità superiori a quelle di cui è comune-mente capace il cervello umano; nelle sue componenti materiali ( hardware ) [...] e da un insieme di circuiti e di dispositivi sui quali si svolgono le funzioni di memoria, di elaborazione e di controllo, che avvengono grazie a programmi contenenti istruzioni ( software ); tali programmi sono basati su un sistema di computazione binario e sono scritti in vari linguaggi di programmazione; [DefinizioneComputer]

Essere in grado di programmare significa essere in grado di scrivere un elenco di istruzioni interpretabili dal computer ed eseguibili. Il linguaggio scelto, che sia di alto livello o di basso livello, deve essere tradotto in linguaggio macchina in questo modo puo essere eseguito dal nostro computer. Questo processo di chiama processo di compilazione mediante il quale il codice , scritto in linguaggio leggibile (human readable) viene tradotto in codice sorgente che verrà poi sottoposto a determinate verifiche per poter essere approvato e trasformato in codice oggetto e poi file eseguibile.

All' interno del computer la componente che si occupa dell' esecuzione del codice è la CPU che, leggendo le istruzioni tradotte dal codice (scritto e poi compilato) esegue delle operazioni specifiche. Come è normale pensare esistono differenti processori , differenti per frequenza di clock, memoria di cache, architettura interna o tensione di alimentazione. Quindi il è necessario

che il codice si chiaro e capibile per un determinato processore.

Un'Instruction Set Architecture (ISA) definisce il modello astratto di un computer ovvero come la CPU controlla hardware e software., specificando sia ciò che il processore è in grado di fare sia come viene fatto.

L'ISA fornisce l'unico modo attraverso il quale un utente è in grado di interagire con l'hardware. Può essere visto come un manuale del programmatore perché è la parte della macchina visibile al programmatore in linguaggio macchina, allo scrittore del compilatore e al programmatore dell'applicazione. L'ISA, inoltre, definisce i tipi di dati supportati, i registri, il modo in cui l'hardware gestisce la memoria principale.

In informatica esistono due tipi popolari di architetture basate sul set di istruzioni. Si tratta di CISC (Complex Instruction Set Computing) e RISC (Reduced Instruction Set Computing).

Il primo approccio consiste in un ISA formata da un set di istruzioni in grado di eseguire operazioni complesse tramite una singola istruzione. Contrariamente il secondo approccio snellisce il set di istruzioni che porta ad avere un architettura più semplice e lineare.

## 1.2 progetto RISC-V

Il RISC-V è un progetto open source basato su un architettura di tipo RISC. Il progetto iniziato nel 2010 all'Università della California, Berkeley. In origine Prof. Krste Asanović e alcuni studenti (Yunsup Lee e Andrew Waterman) svilupparono un ISA per scopi didattici, inizialmente solo progettata e successivamente, dopo dei finanziamenti, prodotta. Il primo workshop risale al 2015 e al lancio il progetto contava 36 membri fondatori.

L'interesse per RISC-V non è dovuto all' architettura RISC o alla tecnologia rivoluzionaria ma il punto fondamentale è che RISC-V è uno standard libero che consente a chiunque di sviluppare in modo libero il proprio hardware per seguire il software.

Dopo il lancio il progetto fu studiato e utilizzato da molte realtà , come ad esempio in DARPA e Linux Foundation , e fu per questo che si fece conoscere molto presto al mondo. Ora Il progetto RISC-V presenta numerose partnership con aziende e una community sparsa per tutto il mondo.

# Capitolo 2

## ISA RISC-V

Rileggi/Correggi  
capitolo 2:  
ISA...

### 2.1 ISA Overview

L'istruzione set architecture di RISC-V, originariamente pensato per supportare la ricerca e l'educazione, si è evoluto fino ad arrivare nell'industria. Alcuni punti focali dello sviluppo dell'isa:

- Un ISA open completamente disponibile per accademie e industrie
- Un ISA adatto per l'implementazione diretta per hardware, non una semplice emulazione
- Un ISA modulare estendibile e modificabile.
- Un ISA con supporto per implementazioni multicore e facilmente parallelizzabile per processi eterogenei

#### 2.1.1 Codifica Istruzione

La base dell'ISA di RISC-V è fissa di base a lunghezza 32-bit ma è modificabile a 16 o a 64 bit o altre lunghezze. Ogni lunghezza è codificata in una maniera chiara come mostrato in figura (Figura 2.1). Di base l'ISA è codificata in modo little-endian, (anche in questo caso modificabile con altre tipologie di memorizzazione).

#### 2.1.2 Memorizzazione

Le istruzioni sono memorizzati in blocchi da 16.bit memorizzati in maniera little-endianess , non modificabile. Se ad esempio si ha un'istruzione su 32

	<code>xxxxxxxxxxxxxxaa</code>	16-bit ( <code>aa</code> ≠ 11)
	<code>xxxxxxxxxxxxxxx</code>	<code>xxxxxxxxxxxxbbb11</code> 32-bit ( <code>bbb</code> ≠ 111)
<code>...xxxx</code>	<code>xxxxxxxxxxxxxxx</code>	<code>xxxxxxxxxx011111</code> 48-bit
<code>...xxxx</code>	<code>xxxxxxxxxxxxxxx</code>	<code>xxxxxxxxx0111111</code> 64-bit
<code>...xxxx</code>	<code>xxxxxxxxxxxxxxx</code>	<code>xnnnxxxxx1111111</code> (80+16*nnn)-bit, <code>nnn</code> ≠ 111
<code>...xxxx</code>	<code>xxxxxxxxxxxxxxx</code>	<code>x111xxxxx1111111</code> Reserved for ≥192-bits

Byte Address:    base+4                          base+2                          base

Figura 2.1: Lunghezza Istruzioni RISC-V

```
// Store 32-bit instruction in x2 register to location pointed to by x3.
sh x2, 0(x3)      // Store low bits of instruction in first parcel.
srli x2, x2, 16    // Move high bits down to low bits, overwriting x2.
sh x2, 2(x3)      // Store high bits in second parcel.
```

Figura 2.2: Memorizza istruzioni

bit l'istruzione avviene divisa in due blocchi da 16-bit la parte più bassa dell'istruzione viene memorizzata nel primo pacchetto e la parte più alta nel secondo pacchetto (Figura 2.2).

### 2.1.3 Cambio di flusso

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.

### 2.1.4 Nomi delle estensioni

Nel manuale sono trattate diverse estensioni ognuna delle quali è identificata da una lettera M, A, F, D Q ,L, C, B, J ,T, P,V, N ogni estensione aggiunge delle istruzioni per un proprio fine. Oltre alle estensioni sono presentati 4 ISA di base RV32I, RV32E, RV64I, RV128I. Per ogni estensione e versione base sono indicate le versioni e se una di queste sono versioni definitive. (Tabella 2.1)

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

La tabella 2.1 presenta i set base RV32I, RV32E, RV64I, RV128I con le rispettive versioni. Ogni set presenta la casella 'Definitiva' che specifica se il set o l'estensione è definitiva (S) o non lo è (N).

## 2.2 ISA Base

L'istruzione set base ha la sigla di RV32I.

RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation. RV32I contains 47 unique instructions, though a simple implementation might cover the eight SCALL/SBREAK/CSRR\* instructions with a single SYSTEM hardware instruction that always traps and might be able to implement the FENCE and FENCE.I instructions as NOPs, reducing hardware instruction count to 38 total. RV32I can emulate almost any other ISA extension (except the A extension, which requires additional hardware support for atomicity).[\[RISCV·ISA-DOC\]](#)

## 2.3 RV32I

Il set è caratterizzato da registri a 32 bit. 31 dei quali registri general-purpose che memorizzano valori interi, identificati dalle sigle x1 - x31, il registro x0 è presente la costante 0. Il termine XLEN rappresenta la lunghezza dei registri che in questo caso è 32.(Figura 2.3)

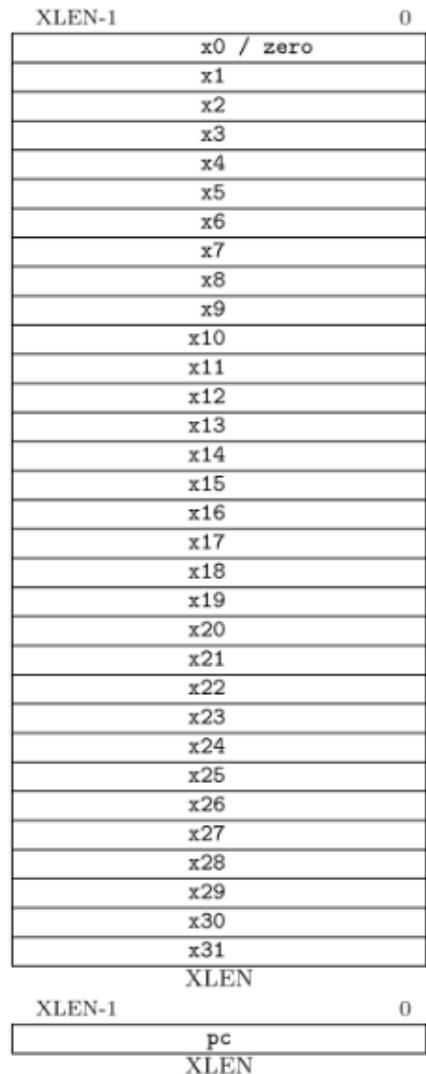


Figura 2.3: Registri architettura

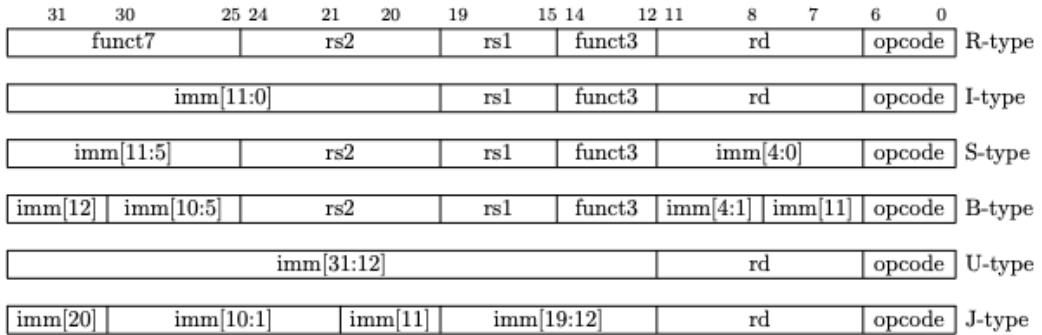


Figura 2.4: Formati istruzione RISC-V

### 2.3.1 Formato istruzioni

Di base questo set presenta sei formati di istruzione (R/I/S/U/B/J) come mostrato in figura (Figura 2.4). Tutti i formati sono su 32 bit. In questi formati i registri source (rs1 e rs2) e il registro destination (rd) vengono mantenuti nelle stesse posizioni per semplificare la decodifica.

- R: Il formato presenta i due registri source e il registro destinazione
- I : Il formato presenta un registro source, una costante(immediato) e il registro destinazione.
- S : Il formato presenta una costante e due registri source.
- B : come il formato S. La costante su 12 bit rappresenta offsets di salto (branch) codificato sui propri bit in multipli di 2.
- U:come il formato I. I 20 bit del registro costante sono shiftati di 12 bit a sinistra.
- J:come il formato U. I 10 bit del registro costante sono shiftati solamente di 1 bit.

### 2.3.2 Istruzioni per computazione intera

La maggior parte delle istruzioni che computano su valori interi opera su bit XLEN dei valori contenuti nei registri. Le istruzioni sono codificate e utilizzato il formato I (register-immediate) o il formato R (register-register). Il registro di destinazione. La destinazione (rd) è la stessa per le istruzioni I

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12 I-immediate[11:0]	5 src	3 ADDI/SLTI[U]	5 dest	7 OP-IMM	
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM	

Figura 2.5: Istruzioni registro-Immediato nel formato R

che per le R. Le istruzioni a computazione itera non causano eccezioni aritmetiche.

### 2.3.3 Istruzioni registro-immediate

Istruzioni di questo tipo utilizzano il formato I:

- ADDI. Viene aggiunta la costante nel registro immediato (12 bit) con estensione del segno per registrare rs1. L'overflow aritmetico viene ignorato e il risultato è semplicemente i bit bassi del risultato.
- NOP. Questa istruzione non modifica alcuno stato visibile all'utente, fatta eccezione per l'avanzamento del Program Counter (PC). L'istruzione è codificata come ADDI x0, x0, 0.
- SLTI. Pone il valore 1 nel registro rd se il registro rs1 è minore del registro immediato con estensione del segno, altrimenti 0. rs1 viene anche trattato come un numero con segno.
- SLTIU. Come SLTI ma il valore in rs1 viene interpretato come privo di segno.
- ANDI, ORI, XORI. Sono operazioni logiche che eseguono AND, OR e XOR bit per bit sul registro rs1 e il registro immediato a 12 bit con estensione del segno e posizionano il risultato in rd.

Le istruzioni in figura 2.6 sono nel formato I. Nel registro costante è presente un valore, codificato nei 5-bit inferiori, che è il valore di shift per il registro source. Lo shift a destra o a sinistra a seconda del sesto bit del campo costante.

Better write

- SLLI. shift logico a sinistra.
- SRRI. shift logica a destra.
- SRAI. shift aritmetico a destra.

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM	
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM	
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM	

Figura 2.6: Istruzioni registro-Immediato nel formato I

31	12 11	7 6	0
imm[31:12]	rd	opcode	
20	5	7	
U-immediate[31:12]	dest	LUI	
U-immediate[31:12]	dest	AUIPC	

Figura 2.7: Istruzioni registro-Immediato nel formato U

Le seguenti istruzioni utilizzano il formato U:

- LUI. viene utilizzato per creare costanti a 32 bit utilizzando il formato U. Posiziona il valore U-immediato nei primi 20 bit del registro di destinazione rd, riempiendo i 12 bit più bassi con zeri.
- AUIPC. viene utilizzato per creare indirizzi relativi al PC utilizzando il formato U. Il risultato, come il caso precedente, forma un offset di 32 bit dai 20 bit U-immediati, riempiendo i 12 bit più bassi con zeri, aggiunge questo offset al PC, quindi inserisce il risultato nel registro rd.

### 2.3.4 Operazioni registro-registro

Queste istruzioni utilizzano il formato R, gli operandi si trovano nei registri source e il risultato dell'operazione è scritto nel registro destinazione. I campi funct7 e funct3 identificano il tipo di operazione.

- ADD, SUB. Addizione e sottrazione. L'overflow è ignorato e il risultato è scritto nel registro destinazione.
- SLT, SLTU. Operazione di comparazione tra i due registri rs1 e rs2, il risultato 1 (rs1 < rs2) o 0 è scritto nel registro destinazione.
- SLL, SRL, SRA shift logici e aritmetici del valore codificato nei 5 bit più bassi del registro rs2. Il valore da shiftare è nel registro rs1 e la

31	30	21	20	19	12 11	7 6	0
imm[20]	imm[10:1]	imm[11]	imm[19:12]		rd	opcode	
1	10	1	8		5	7	JAL

offset[20:1]    dest

Figura 2.8: Istruzioni registro-registro

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12 offset[11:0]	5 base	3 0	5 dest	7	JALR

Figura 2.9: Istruzioni per salto incondizionato

direzione è identificata dal 6 bit di rs2. Il risultato è memorizzato nel registro rd.

### 2.3.5 Controllo del flusso

RV32I presenta due tipi di trasferimento di controllo: salto incondizionato e condizionato.

#### Incondizionato

- JAL. Usa il formato J. Il campo immediato codifica numeri con segno in multipli di 2 byte.. The offset is sign-extended and added to the PC to form the jump target address. L'offset viene esteso con il segno e aggiunto al PC per formare l'indirizzo di destinazione del salto, l'indirizzo di destinazione è memorizzato in rd.
- JALR. Usa il formato I.L'indirizzo di destinazione si ottiene sommando i 12 bit con segno I-immediate al registro rs1, quindi impostando a zero il bit meno significativo del risultato. L'indirizzo dell'istruzione successiva al salto (PC+4) viene scritto nel registro rd. Il registro x0 può essere utilizzato come destinazione se il risultato non è richiesto.

Entrambe le istruzioni possono generare dei disallineamenti se questo si verifica si verifica un eccezione.

31	30	25 24	20 19	15 14	12 11	8	7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]		opcode	
1	6	5	5	3	4	1		7	
offset[12,10:5]		src2	src1	BEQ/BNE		offset[11,4:1]		BRANCH	
offset[12,10:5]		src2	src1	BLT[U]		offset[11,4:1]		BRANCH	
offset[12,10:5]		src2	src1	BGE[U]		offset[11,4:1]		BRANCH	

Figura 2.10: Istruzioni per salto condizionato

### Condizionato

Tutte le istruzioni per il salto condizionato sono del formato B. Per calcolare l'indirizzo target viene utilizzata la cosante su i 12 bit che viene aggiunta al valore corrente di PC.

- BQE, BNE. Salto eseguito se rs1 è uguale a rs2 nel primo caso, diversi nel secondo.
- BLT, BLTU. Salto eseguito se rs1 è minore di rs2. Nel primo caso i due operandi vengono considerati con segno, nel secondo senza segno.
- BGE, BGEU. Salto eseguito se rs1 è maggiore o uguale di rs2. Nel primo caso i due operandi vengono considerati con segno, nel secondo senza segno.
- BGT, BGTU, BGT, BGTU. Possiamo sintetizzarli come l'inverso dei precedenti.

### 2.3.6 Load e Store

In RV32I sono presenti diverse istruzione di load e store queste accedono alla memoria mentre le istruzioni aritmetiche operano solo sui registri della CPU. Lo spazio fornito da RV32I è uno spazio di indirizzi a 32 bit con in modalità little-endian. Le istruzioni di load e store con destinazione il registro x0 sollevano un eccezione, anche se il valore poi verrà scartato.

Le istruzioni di load e store trasferiscono il valore tra i registri e la memoria. Le istruzioni di load utilizzano il formato I mentre il formato per le istruzioni store è quello S. L'indirizzo del byte effettivo si ottiene aggiungendo il registro rs1 all'offset a 12 bit con estensione del segno. Store (vedi Figura ). Load (vedi Figura ).

- LW, LH, LHU, LB, LBU. LW carica il valore codificato su 32 bit dalla memoria in rd. LHU carica un valore a 16 bit dalla memoria, quindi il

31	imm[11:0]	20 19	15 14	12 11	7 6	0
		rs1	funct3	rd	opcode	
12		5	3	5	7	
offset[11:0]		base	width	dest	LOAD	
31	imm[11:5]	25 24	20 19	15 14	12 11	7 6
7		rs2	rs1	funct3	imm[4:0]	0
offset[11:5]		src	base	width	offset[4:0]	opcode
					7	
					STORE	

Figura 2.11: Istruzioni load e store

segno si estende a 32 bit prima di archiviarlo in rd, mentre LHU fa lo stesso ma esegue un'estensione zero a 32 bit. LB e LBU sono definiti in modo analogo a LH e LHU ma per valori a 8 bit.

- SW, SH, SB. Memorizzano rispettivamente valori a 32 bit, 16 bit e 8 bit dai bit bassi del registro rs2 alla memoria.

L'ISA di base supporta accessi disallineati per i dati, ma questo potrebbe essere molto inefficienti e lento, a seconda dell'implementazione. Per questo motivo, vengono garantiti accessi a sezioni allineate solo per l'esecuzione atomica.

### 2.3.7 Controllo e i registri di stato

Le informazioni di sistema sono memorizzate in registri speciali chiamati Control Status Registers (CSR). Questi registri di solito memorizzano informazioni sull'istruzione precedente eseguita e sulla modalità operativa. RV32I consente di accedere a quei registri con formato di istruzione di tipo I. In base all'implementazione, queste istruzioni possono richiedere l'esecuzione di un accesso privilegiato.

#### Istruzioni CSR

Tutte le istruzioni:

- CSRRW. ("Atomic Read/Write CSR"), questa istruzione scambia in modo atomico i valori nei CSR e nei registri interi. CSRRW legge il vecchio valore del CSR, estende a zero il valore a 32 bit, quindi lo scrive nel registro intero rd. Il valore iniziale in rs1 viene scritto nel CSR. Se rd=x0, l'istruzione non dovrebbe leggere il CSR.

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
source/dest	source	CSRRW	dest	SYSTEM	
source/dest	source	CSRRS	dest	SYSTEM	
source/dest	source	CSRRC	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRWI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRSI	dest	SYSTEM	
source/dest	uimm[4:0]	CSRRCI	dest	SYSTEM	

Figura 2.12: Istruzioni CSR

- CSRRS. ("Atomic Read and Set Bits in CSR"), questa istruzione legge il valore del CSR, estende a zero il valore a 32 bit e lo scrive nel registro intero rd. Il valore iniziale nel registro intero rs1 viene trattato come una maschera di bit che specifica le posizioni di bit da impostare nel CSR. Qualsiasi bit alto in rs1 farà sì che il bit corrispondente venga impostato nel CSR, se quel bit CSR è scrivibile. Gli altri bit nella CSR non sono interessati. Come per l'istruzione precedente, se rs1=x0, l'istruzione non scriverà affatto nel CSR.
- CSRRC.("Atomic Read and Clear Bits in CSR"), questa istruzione legge il valore del CSR, estende a zero il valore a 32 bit e lo scrive nel registro intero rd. Il valore iniziale nel registro intero rs1 viene trattato come una maschera di bit che specifica le posizioni di bit da cancellare nel CSR. Qualsiasi bit alto in rs1 provocherà la cancellazione del bit corrispondente nel CSR, se quel bit CSR è scrivibile. Gli altri bit nella CSR non vengono modificati. Se rs1=x0, l'istruzione non scriverà affatto nel CSR.
- CSRRWI, CSRRSI, CSRRCI. Sono varianti delle precedenti e sono simili tranne per il fatto che aggiornano il CSR utilizzando un valore a 32 bit ottenuto estendendo a zero un campo immediato senza segno (uimm[4:0]) a 5 bit codificato nel campo rs1 invece di un campo valore da un registro intero.

## Timer e counter

RV32I fornisce una serie di contatori a livello utente di sola lettura a 64 bit, che sono mappati nello spazio degli indirizzi CSR a 12 bit e accessibili in parti a 32 bit utilizzando le istruzioni CSRRS.

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	
RDCYCLE[H]	0	CSRRS	dest	SYSTEM	
RDTIME[H]	0	CSRRS	dest	SYSTEM	
RDINSTRET[H]	0	CSRRS	dest	SYSTEM	

Figura 2.13: Time e counter

- RDCYCLE[H]. La pseudo-istruzione RDCYCLE legge i 32 bit bassi del ciclo CSR che contiene un conteggio del numero di cicli di clock eseguiti dal core del processore che è in esecuzione da un'ora di inizio arbitraria nel passato. RDCYCLEH è un'istruzione solo RV32I che legge i bit 63–32 dello stesso contatore di cicli. (La frequenza del ciclo (cicli/secondo) del contatore dipendono solamente dall'implementazione.)
- RDTIME[H]. La pseudo-istruzione RDTIME legge i 32 bit bassi del CSR temporale, che conta il tempo reale trascorso da un'ora di inizio arbitraria nel passato. RDTIMEH è un'istruzione solo RV32I che legge i bit 63–32 dello stesso contatore in tempo reale. (L'ambiente di esecuzione fornisce i dati per determinare il periodo del contatore in tempo reale (secondi/tick).)
- RDINSTRET[H]. Questa pseudo-istruzione legge i 32 bit bassi della CSR INSTRET, che conta il numero di istruzioni eseguite da un punto di inizio arbitrario in passato. RDINSTRETH è un'istruzione esclusiva per RV32I che legge i bit 63–32 dello stesso contatore di istruzioni.

Sezione/  
due parole  
sulle pseudo  
istruzioni

## 2.4 Alcune estensioni

### 2.4.1 Estensione M

I lettori più attenti si saranno di certo accorti che tra le istruzioni presentate nel set base non è presente nessuna operazione di moltiplicazione. Certamente è possibile implementare questa operazione utilizzando addizioni in loop ma è più macchinoso di scrivere un'unica istruzione. L'estensione denominata "M" introduce le istruzioni standard di moltiplicazione e divisione di interi, che moltiplicano o dividono i valori contenuti in due registri di interi. Le istruzioni introdotte da questa estensione sono:

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
MULDIV	multiplier	multiplicand	MUL/MULH[[S]U]	dest	OP	
31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
MULDIV	divisor	dividend	DIV[U]/REM[U]	dest	OP	
MULDIV	divisor	dividend	DIV[U]W/REM[U]W	dest	OP-32	

Figura 2.14: Istruzioni moltiplicazione e divisione dell'estensione M

- MUL, MULH, MULHU, MULHSU. Operazione di moltiplicazione. L'operazione MUL moltiplica gli operandi su 32 bit a posizionano i 32 bit meno significativi new registro destinazione. Le altre 3 invece operano su operandi considerati entrambi con segno, entrambi senza segno e rs1 con segno e rs2 senza segno , rispettivamente, e il risultato sono i 32 bit più significativi che vengono memorizzati nel registro destinazione.
- DIV, DIVU. Calcolano il risultato della divisione con segno e senza segno degli interi e i registri corrispondenti.
- REM, REMU. Calcolano il resto della divisione con segno e senza segno degli interi e i registri corrispondenti.

### 2.4.2 Estensione F

Questa estensione provvede a fornire delle operazioni Single-Precision e Floating-Point. L'estensione F prevede 32 registri a virgola mobile, f0–f31 e un registro di controllo e di stato a virgola mobile (FCSR), che contiene la modalità operativa e lo stato di eccezione dell'unità a virgola mobile. mostra il file di registro a virgola mobile e l'FCSR. Il termine FLEN descrive la larghezza dei registri a virgola mobile nel RISC-V ISA. FLEN=32 corrisponde alla lunghezza per l'estensione in virgola mobile a precisione singola. Con questa estensione ci sono le operazioni:

- FADD. Calcola l'addizione
- FSUB. Calcola la sottrazione
- FMUL. Calcola la sottrazione
- FDIV. Calcola la divisione

31	27 26	25 24	20 19	15 14	12 11	7 6	0
funct5	fmt	rs2	rs1	rm	rd	opcode	
5	2	5	5	3	5	7	
FADD/FSUB	S	src2	src1	RM	dest	OP-FP	
FMUL/FDIV	S	src2	src1	RM	dest	OP-FP	
FMIN-MAX	S	src2	src1	MIN/MAX	dest	OP-FP	
FSQRT	S	0	src	RM	dest	OP-FP	

Figura 2.15: Istruzioni Single-Precision registro-register nel formato R

31	27 26	25 24	20 19	15 14	12 11	7 6	0
rs3	fmt	rs2	rs1	rm	rd	opcode	
5	2	5	5	3	5	7	
src3	S	src2	src1	RM	dest	F[N]MADD/F[N]MSUB	

Figura 2.16: Istruzioni Single-Precision registro-register in un formato speciale

- FMIN. Scrive il minimo tra rs1 e rs2 in rd
- FMAX. Scrive il massimo tra rs1 e rs2 in rd
- FSQRT. Calcola la radice.

Oltre a queste operazioni semplici l'estensione presenta anche altre operazioni.

- FMADD  $rs1 \times rs2 + rs3$ .
- FMSUB  $rs1 \times rs2 - rs3$ .
- FNMSUB  $-rs1 \times rs2 + rs3$ .
- FNMMADDr  $rs1 \times rs2 - rs3$ .

Per distinguere le operazioni in single-Precision e in double-Precisione l'estensione prevede un ulteriore lettera identificativa all nome dell'istruzione. A livello di codifica il campo fmt 2 bit identifica la codifica del numero floating-point. (Tabella 2.2) Guardando del codice ci permette di capire se parliamo di un n-esima Precisione e ci fa capire la lunghezza dei registri utilizzati.

Oltre alle istruzioni aritmetiche sono introdotte anche le apposite istruzioni di load e store, move e di comparazione che sono molto simili come logica all ISA base. Ogni istruzione può sollevare un'eccezione di Invalid Operation se almeno un operatore non rispetta il tipo specificato. Come ultima cosa con questa estensione è introdotta un'istruzione:

fmt	Lettera	Significato
00	S	32-bit single precision
01	D	64-bit double-precision
10	-	riservato
11	Q	128-bit quad-precision

Tabella 2.2: Campo fmt

Valore	Significato
0	rs1 is $-\infty$ .
1	rs1 is a negative normal number.
2	rs1 is a negative subnormal number.
3	rs1 is $-0$ .
4	rs1 is $+0$ .
5	rs1 is a positive subnormal number.
6	rs1 is a positive normal number.
7	rs1 is $\infty$ .
8	rs1 is a signaling NaN.
9	rs1 is a quiet NaN.

Tabella 2.3: Formato del risultato di FCLASS

- FCLASS.S. Questa istruzione esamina il valore nel registro a virgola mobile rs1 e scrive nel registro intero rd una maschera a 10 bit che indica la classe del numero a virgola mobile. Il formato della maschera è descritto nella Tabella 2.3. Il bit corrispondente in rd sarà impostato se la proprietà è true e clear in caso contrario. Tutti gli altri bit in rd vengono cancellati. Si noti che verrà impostato esattamente un bit in rd.

### 2.4.3 E per Embedded

RV32E Isa sviluppato per dispositivi Embedded. La differenza principale sta nel avere la metà dei registri x0-x15 (x0 mantiene sempre la costante 0) al posto dei 32 di RV32I. Le istruzioni utilizzabili sono le istruzioni che utilizzano i registri ammissibili mentre le istruzioni che richiedono i registri x16-x31 risultano illegali e, se utilizzate, sollevano eccezioni. Anche con RV32E può essere estensibile con alcune estensioni standard.

## 2.5 Altri Set

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 un piccolo accorgimento che l'estensione E ed F sollevano eccezioni di operazioni non valide.

Esistono altre estensioni come l'estensione per le istruzioni atomiche "A". Essa contiene istruzioni che leggono-modificano-scrivono in modo atomico dalla e alla memoria per supportare la sincronizzazione tra più thread RISC-V in esecuzione nello stesso spazio di memoria. Le due forme di istruzione atomica vengono fornite istruzioni apposite per supportare la sincronizzazione. Entrambi i tipi di istruzione atomica supportano vari ordini di coerenza della memoria, inclusi semantica disordinata, di acquisizione, di rilascio e sequenzialmente coerente. Queste istruzioni consentono a RISC-V di supportare la coerenza della memoria.

E infine è presente un'estensione che comprime il codice. Questa è chiamata "C". Con questa estensione l'istruzione viene ridotta a 16 bit e compressa. L'estensione C può essere aggiunta a qualsiasi base ISA (RV32, RV64, RV128) e usiamo il termine generico "RVC" per coprire ognuno di questi. Tipicamente, Il 50%–60% delle istruzioni RISC-V in un programma può essere sostituito con istruzioni RVC, risultanti con una riduzione della dimensione del codice del 25%–30%.

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

# Capitolo 3

## Compilatori

Rileggi/Correggi  
capitolo 3:  
Compilatore...  
...

### 3.1 Descrizione

Un compilatore è un programma che trasforma il codice sorgente scritto in un determinato linguaggio di programmazione, in un altro linguaggio informatico (codice target). Il motivo più comune per trasformare il codice sorgente è creare un programma eseguibile. Il linguaggio attraversa diverse fasi prima di diventare il codice target. La prima attività è l' analisi lessicale dove il programma viene analizzato parola per parola e genera dei token. La seconda attività è l analisi sintattica dove viene controllata la correttezza formale del programma. In questa fase i token precedentemente generati vengono analizzati e utilizzati per generare una struttura ad albero che nella fase di analisi semantica viene analizzato per verificare la correttezza delle espressioni scritte. Infine viene generato il codice target. Alcuni compilatori prevedono un ulteriore fase la fase di ottimizzazione dove il codice viene ottimizzato

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 programmatori 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.

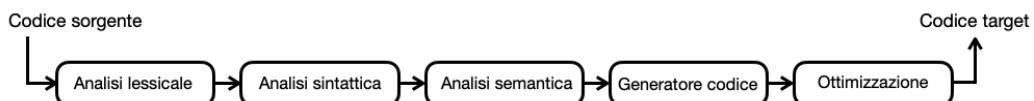


Figura 3.1: Schema meccanismo di compilazione

## 3.2 Storia

Il primo compilatore teorico fu pensato da Corrado Böhm che nel 1951 lo sviluppò per la sua tesi di dottorato . Il primo compilatore implementato è stato scritto da Grace Hopper , coniando il termine "compilatore". Il primo compilatore , chiamato sistema A-0, funzionava come caricatore o linker , non come i moderni compilatori. E' importante menzionare che una versione successiva (la versione A-2 datata 1953)fu il primo software libero e open source della storia dell'informatica.

Il primo compilatore ALGOL 58 fu completato alla fine del 1958 da Friedrich L. Bauer , Hermann Bottenbruch, Heinz Rutishauser e Klaus Samelson per il computer Z22 . Nel 1960, un compilatore Fortran esteso, ALTAC, era disponibile sulla Philco 2000, quindi è probabile che un programma Fortran sia stato compilato per le architetture di computer IBM e Philco a metà del 1960. Il primo linguaggio di alto livello multipiattaforma noto è stato COBOL.

Come qualsiasi altro software, ci sono vantaggi nell'implementare un compilatore in un linguaggio di alto livello. In particolare, un compilatore può essere self-hosted , ovvero scritto nel linguaggio di programmazione che compila. Il sopracitato compilatore di Böhm è un compilatore self-hosted. Il suo compilatore, oltre ad essere completo, fu il primo definito con il proprio linguaggio. Oltre a quel compilatore possiamo citare il Navy Electronics Laboratory International ALGOL Compiler (NELIAC) o il Lisp.

Come durante una comunicazione verbale anche i programmi hanno una propria grammatica con regole queste vengono analizzate da un parser, generato da un generatore, ha lo scopo di analizzare come è scritto il programma e generare i token associati. Un primo progetto risale al 1960 come progetto dell'università di Università di Manchester, non è da considerare un compilatore moderno ma un buon punto per la costruzione e per l'ideazione del sistema Unix nel 1969 da parte di Ken Thompson.

## 3.3 Cross-compilazione

Un compilatore incrociato è un compilatore in grado di creare codice eseguibile per una piattaforma diversa da quella su cui è in esecuzione il compilatore. Ad esempio, un compilatore che viene eseguito su un PC ma genera codice che viene eseguito su uno smartphone Android è un compilatore incrociato.

Un compilatore incrociato è utile per compilare codice per più piattaforme da un host di sviluppo. La compilazione diretta sulla piattaforma di destinazione potrebbe non essere fattibile, ad esempio su sistemi embedded con risorse informatiche limitate.

### 3.4 Compilatori moderni

Nel ecosistema dei compilatori moderni i piu famosi sono: GCC e CLANG. Il primo GCC (GNU Compiler Collection) è un compilatore multipiattaforma creato nel 1987 da Richard Stallman. Oggi GCC viene sviluppato da programmatore da ogni parte del mondo ed è stato portato su più tipi di processori e sistemi operativi. GCC è il compilatore ufficiale di GNU e usato per lo sviluppo di altri sistemi operativi(macOS, DOS). Nato per il linguaggio C, oggi di vari front end per altri linguaggi, tra cui Java, C++, Objective C, e vari back-end in grado di generare linguaggi macchina per varie architetture, come ad esempio x86,, ARM.

Il secondo, piu giovane, nato nel 2005 da Apple Inc. l'inizio dello sviluppo è dovuto a esigenze di avere un compilatore di casa Apple ottimizzato per dispositivi Apple. Il progetto LLVM originariamente intendeva utilizzare il front-end di GCC. Il codice sorgente di GCC, tuttavia, è grande e alquanto ingombrante; come ha affermato uno sviluppatore GCC di lunga data riferendosi a LLVM,

Trying to make the hippo dance is not really a lot of fun

Cercare di far ballare l'ippopotamo non è davvero molto divertente.

Inoltre, il software Apple utilizza Objective-C, che è una priorità bassa per gli sviluppatori GCC. Pertanto, GCC non si integra perfettamente nell'ambiente di sviluppo integrato (IDE) di Apple. Infine, il contratto di licenza di GCC, la GNU General Public License (GPL) V3, richiede agli sviluppatori che distribuiscono estensioni o versioni modificate di GCC di rendere disponibile il loro codice sorgente, ma la licenza software permissiva di LLVM è priva di tale impedimento. Alla fine, Apple ha scelto di sviluppare Clang, un nuovo front-end del compilatore che supporta C, Objective-C e C++. Nel luglio 2007 il progetto ha ricevuto l'approvazione per diventare open-source.

### 3.5 Un buon compilatore

I processori moderni hanno tutti pipeline superscalari e lunghe e strutture interne complesse e supportano unità di estensione vettoriale. Inoltre, gli

standard dei moderni linguaggi avanzati astraggono costantemente i dettagli dell'hardware e delle strutture dati sottostanti per generare codice generale più logico e matematico, invece di istruzioni operative specifiche e percorsi di accesso alla memoria. Gli standard dei linguaggi, come C++, si fanno sempre più espressivi e astratti. Una maggiore espressività aumenta l'onere del compilatore di generare un buon codice assembly dalle complesse strutture compilate dai programmati. Il compilatore deve essere più intelligente e lavorare di più per massimizzare le prestazioni utilizzando il codice. Non tutti i compilatori possono farlo. Quando si seleziona un compilatore, è necessario considerare se lo stesso segmento di codice può generare comandi assembly più efficienti con uno e l'altro compilatore. Oltre a generare programmi eseguibili ad alte prestazioni, i compilatori moderni 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. In termini di estensione del linguaggio, i moderni sistemi informatici con più kernel, capacità di elaborazione vettoriale e acceleratori forniscono capacità superiori alle capacità naturali dei comuni linguaggi di programmazione. Pertanto, specifici framework HPC (High Performance Computing), come OpenMP e OpenACC, per poter colmare questa lacuna. Questi framework forniscono API (Application Program Interface) che i programmati possono utilizzare per esprimere il parallelismo nel codice.. Pertanto, i compilatori devono stare al passo con lo sviluppo degli standard di estensione del linguaggio.

In conclusione, un buon compilatore ci permette di concentrarci sul processo di programmazione, piuttosto che combatterne le carenze. Può supportare gli standard linguistici più recenti, generare comandi ottimizzati dal codice più astratto e compilare il codice sorgente in meno tempo.

## 3.6 CLANG e GCC

Clang è compatibile con GCC. Clang è un sostituto di GCC. Ma detta così ci si potrebbe chiedere perché usare uno o l'altro. Analizziamo un po più in dettaglio i due compilatori. Gli sviluppatori di Clang mirano a ridurre l'ingombro di memoria e aumentare la velocità di compilazione rispetto ai compilatori concorrenti, come GCC. Nell'ottobre 2007 furono eseguiti dei test comparativi che portarono dei primi risultati notevoli, Clang ha compilato le librerie Carbon a una velocità doppia rispetto a GCC, utilizzando circa un sesto di memoria e spazio su disco di GCC. Con lo sviluppo di Clang e GCC il primo compila ancora costantemente più velocemente di GCC Tuttavia, entro il 2019, Clang è significativamente più lento nella compilazione del

kernel Linux rispetto a GCC, pur rimanendo leggermente più veloce nella compilazione di LLVM.

Sebbene Clang sia stato storicamente più veloce di GCC nella compilazione, la qualità dell'output è rimasta indietro. A partire dal 2014, le prestazioni dei programmi compilati da Clang sono rimaste indietro rispetto alle prestazioni del programma compilato da GCC, a volte per fattori importanti (fino a 5x), replicando i precedenti rapporti di prestazioni più lente. Entrambi i compilatori si sono evoluti per aumentare le loro prestazioni da allora, con la riduzione del divario:

I confronti nel novembre 2016 tra GCC 4.8.2 e clang 3.4, su un numerosi file di test, mostrano che GCC supera il clang di circa il 17% su un codice sorgente ben ottimizzato. I risultati dei test sono specifici del codice e il codice sorgente C non ottimizzato può invertire tali differenze. I due compilatori sembrano quindi comparabili. Da un lato la velocità di compilazione di Clang supera quella di GCC ma le performance del codice generato inverte la classifica. I confronti nel 2019 su Intel Ice Lake hanno dimostrato che i programmi generati da Clang 10 hanno raggiunto il 96% delle prestazioni di GCC 10 su 41 benchmark diversi. [GCCvsCLANG]

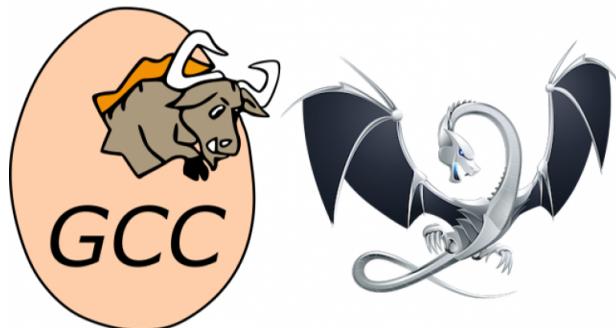


Figura 3.2: Loghi dei compilatori, GCC (a sinistra) CLANG (a destra)

### Benchmark utilizzato

### Architettura per il test

Architecture: x86\_64

Processore: Intel (R) Xeon (R) Platinum 8163 CPU @ 2.50 GHz

L1 cache: 32 KB

L2 cache: 1,024 KB

L3 cache: 33,792 KB

Memoria: 800 GB

OS: Alibaba Group Enterprise Linux Server release 7.2 (Paladin)

Kernel: 4.9.151-015.ali3000.alios7.x86\_64

Compilatori: Clang/LLVM 8.0 GCC8.3.1

## Programma

SPEC CPU 2017 è un set di strumenti di test del sottosistema CPU per testare CPU, cache, memoria e compilatore. Contiene 43 test di quattro categorie, tra cui SPECspeed 2017 INT e FP per la velocità di calcolo intera e in virgola mobile e SPECrace 2017 INT e FP che testano il tasso di concorrenza intero e a virgola mobile. Clang non supportando il Fortran. sono stati scelti C/C++ per testare la differenza di prestazioni single-core tra i programmi binari generati da Clang e GCC.

Di seguito alcuni grafici che rappresentano il tempo di compilazione di alcuni test e le performance del codice generato.

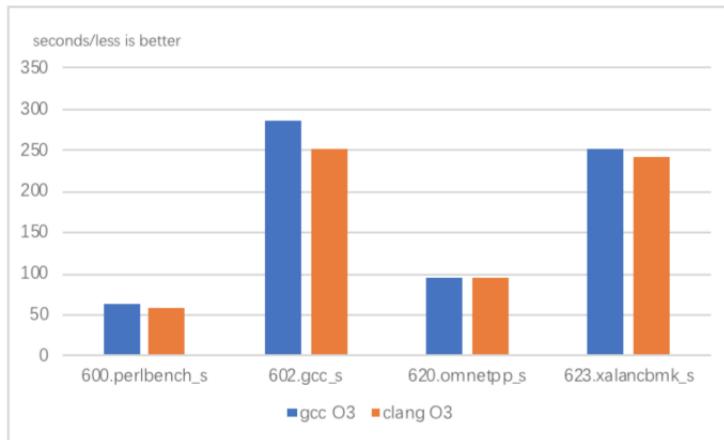


Figura 3.3: Tempo di compilazione

Il processo di compilazione di GCC è il seguente: leggere il file sorgente, preelaborare il file sorgente, viene convertito in un IR, ottimizzato e generato un file assembly. Quindi l'assemblatore genera un file oggetto. Clang e LLVM non si basano su compilatori indipendenti, ma integrano compilatori auto-implementati nel back-end. Rispetto a GCC, la struttura dei dati di LLVM, è più concisa e occupa meno memoria durante la compilazione ed è più veloce. Pertanto, Clang è vantaggioso in termini di tempo di compilazione, come dimostrano i dati ottenuti dalla compilazione 3.3.

Per valutare la performance i programmi sono stati compilati con l'opzione di ottimizzazione 2 e 3.

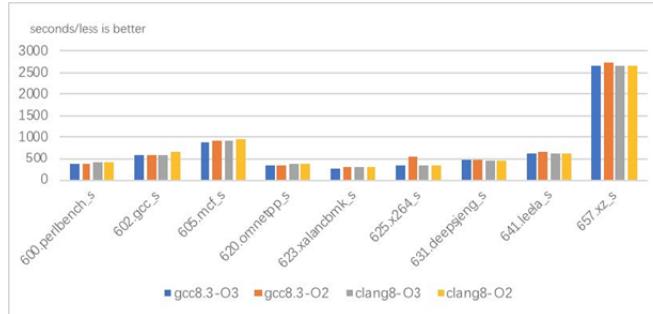


Figura 3.4: Performance di esecuzione

### 3.6.1 Vantaggi dei compilatori

#### Compilatore GCC

- GCC supporta linguaggi più tradizionali rispetto a Clang e LLVM.
- GCC supporta architetture più meno popolari e supporta RISC-V prima di Clang e LLVM.
- GCC supporta più estensioni del linguaggio e più funzionalità del linguaggio assembly rispetto a Clang e LLVM.

#### Compilatore CLANG

- I linguaggi emergenti utilizzano i framework LLVM, come Swift, Rust, Julia e Ruby.
- Clang e LLVM rispettano gli standard C e C++ in modo più rigoroso rispetto a GCC.
- Clang supporta anche alcune estensioni, come gli attributi per il controllo di sicurezza dei thread.
- Clang fornisce ulteriori utili strumenti, come scan-build e clang static Analyzer e altri per l'analisi statica, clang-format e clang-tidy per l'analisi della sintassi, nonché il plug-in dell'editor Clangd.
- Clang fornisce informazioni diagnostiche più accurate e intuitive ed evidenzia i messaggi di errore, le righe di errore, i prompt delle righe di errore e i suggerimenti per la riparazione.
- Clang considera le informazioni diagnostiche come una caratteristica.

### 3.6.2 Community

E bene tener presente che entrambi i compilatori hanno delle community di supporto e di sviluppo per entrambi i progetti.

#### GCC Community

Come altre comunità di software open source, la comunità GCC è dominata da appassionati di software libero. Nel processo di sviluppo, oggi si sono formanti dei meccanismi di gestione e partecipazione della comunità. Attualmente, la comunità GCC è una società relativamente stabile e ben definita in cui ogni persona ha ruoli e doveri chiari:

- Richard Stallman e Free Software Foundation (FSF): anche se raramente coinvolti nella gestione della comunità GCC, Richard Stallman e FSF sono ancora distaccati nelle licenze e negli affari legali.
- Comitato industriale GCC: gestisce gli affari della comunità GCC, gli argomenti di sviluppo GCC indipendenti dalla tecnologia e la nomina e l'annuncio di revisori e manutentori. Attualmente conta 13 membri.
- Manutentori globali: dominano le attività di sviluppo di GCC. In una certa misura, determinano il trend di sviluppo di GCC. Attualmente, ci sono 13 manutentori globali, che non ricoprono tutte le cariche nel Comitato Industriale GCC.
- Manutentori di frontend, middle-end e back-end: Sono responsabili del codice dei moduli di GCC corrispondente e molti di loro sono i principali contributori al codice del modulo. Vale la pena notare che i revisori sono generalmente classificati in questo gruppo. La differenza è che i revisori non possono approvare la propria patch, mentre i manutentori possono inviare le proprie modifiche nell'ambito della propria responsabilità senza l'approvazione dei revisori.
- Collaboratori: sono i gruppi di sviluppatori più estesi nella comunità di GCC. Dopo aver firmato l'accordo sul copyright, tutti gli sviluppatori possono richiedere l'autorizzazione Scrivere dopo l'approvazione dalla community e quindi inviare il codice da soli.

Come altre comunità open source fanno gola alle società commerciali che hanno iniziato a svolgere ruoli importanti nella comunità, come il reclutamento di sviluppatori e la sponsorizzazione di riunioni di sviluppo. Attualmente, la comunità GCC è dominata dai seguenti tipi di società commerciali:

- Venditori di sistemi inclusi (RedHat, SUSE).
- Venditori di chip (Intel, ARM, AMD, IBM).
- Venditori specifici per determinati linguaggi o determinati servizi.

Nell'attuale comunità GCC, i fornitori di chip dominano lo sviluppo del back-end, mentre i fornitori di sistemi guidano altre aree di sviluppo. In termini di sviluppo della comunità, il codice GCC è attualmente ospitato sul proprio server SVN. Viene fornita un'API Git per facilitare lo sviluppo e l'invio. La revisione delle patch è simile a quella della comunità del kernel Linux e utilizza il modulo Mailing List. Come accennato in precedenza, la comunità GCC è una società di conoscenza relativamente stabile (o chiusa). La comunità ha fondamentalmente da 150 a 200 contributori attivi ogni anno e tiene una conferenza degli sviluppatori a settembre ogni anno.

## **LLVM Community**

La comunità LLVM è una comunità di compilatori giovane. Risponde rapidamente alle domande dei nuovi utenti e alle recensioni delle patch. Tutti i progetti e i problemi LLVM vengono discussi tramite l'elenco e-mail DevExpress e l'invio del codice viene notificato tramite l'elenco e-mail dei commit. Tutti i bug e le modifiche alle funzionalità vengono tracciati tramite l'elenco dei bug. Le patch inviate sono consigliate per i branch master. Lo stile è conforme agli standard di codifica LLVM e la revisione del codice viene eseguita tramite Phabricator. Attualmente, il repository di codice LLVM è stato migrato su GitHub. A differenza della comunità GCC, la comunità LLVM ha solo la LLVM Foundation. La Fondazione LLVM ha otto membri. Oltre a gestire gli affari della comunità LLVM, ogni membro della LLVM Foundation deve guidare i problemi di sviluppo di LLVM relativi alla tecnologia. Attualmente il presidente è Tanya Lattner, moglie di Chris Lattner. Lo stesso Chris Lattner è anche un membro della fondazione e ha un forte controllo sulla comunità LLVM e sulla direzione dello sviluppo di LLVM. La politica di revisione del codice nella comunità LLVM è sostanzialmente la stessa della comunità GCC. La differenza è che, a causa del rapido sviluppo di LLVM, molti contributori non hanno il permesso di accesso al commit e devono inviare il loro codice tramite i manutentori. Attualmente, le comunità Clang e LLVM hanno più di 1.000 contributori ogni anno.

Reference  
section  
cross-  
compiler

Componenti  
della tool-  
chain ?

## 3.7 Toolchain di RISC-V

Per in alcuni sistemi non è presente nessun compilatore, per questi è possibile utilizzare la tecnica della cross-compilation ([\(\)](#)). Per una cross-compilazione di base è necessario avere una toolchain. una toolchain è l'insieme dei programmi usati nello sviluppo di un prodotto, tipicamente un altro programma o sistema di programmi. Tali strumenti sono utilizzati in catena in modo tale che producano del codice per un determinato dispositivo. Per dispositivi RISC-V viene utilizzata la toolchain disponibile per LLVM/CLANG e GCC. Entrambe le toolchain sono mantenute dalle community, quella più aggiornata e utilizzata è quella del progetto GNU.La documentazione e i vari tool sono disponibili sul loro sito.

### GCC toolchain

La toolchain GNU RISC-V è composta da Binutils, newlib e glibc. La toolchain GCC si basa su GCC 6.1.0 e riceve commit su molto di frequente. E' possibile usare uno script per creare l'intera toolchain per RISC-V. La toolchain di test utilizzata è stata RISC-V a 64bit.

### CLANG toolchain

La toolchain di LLVM per RISC-V è iniziata nella versione LLVM 3.3 ed è attualmente stabile. Tuttavia, la versione 3.3 è piuttosto vecchia c'è anche un'altra versione LLVM 3.8 meno stabile. La toolchain di CLANG è più macchinosa utilizzarla ma più configurabile.

### Comparazione

#### Confronto tra compilatori

Per la comparazione sono stati utilizzati RISC-V 32, 64 con i compilatori GCC e CLANG v 3.8. Il test eseguito è sulla dimensione del codice generato dai due compilatori. I programmi utilizzati fanno parte del benchmark MiBench.<sup>1</sup>.

In media, GCC produce dimensioni del codice abbastanza simili sia per RISC-V 32 che per 64. Il codice generato da Clang è leggermente più grande, in particolare per RISC-V 32, la cui dimensione del codice è in realtà maggiore rispetto a RISC-V 64. Questo può dimostrare che CLANG può essere

<sup>1</sup>a free, commercially representative embedded benchmark suite

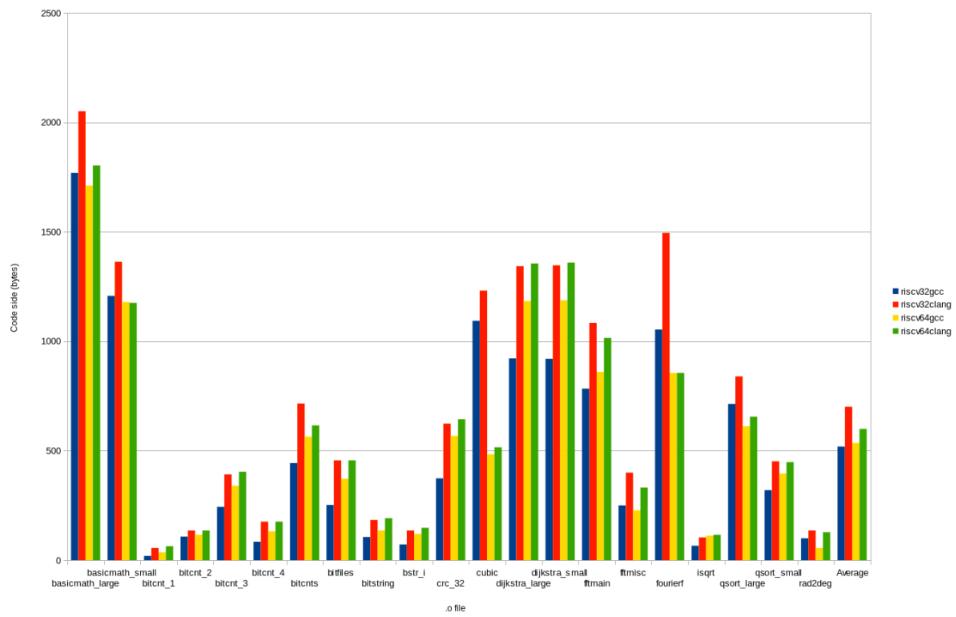


Figura 3.5: Confronto dimensione dei file generati dai compilatori

Compilatore	RISC-V 32	RISC-V 64
GCC	519 bytes	536 bytes
CLANG	701 bytes	600 bytes

Tabella 3.1: Dimensione media dei compilatori

ulteriormente migliorato.

### Comparazione del codice RISC-V e ARM

Architettura	Dimensione media
RISC-V 32	519 bytes
RISC-V 64	536 bytes
ARMv7m	535 bytes
ARMv8m	535 bytes
ARMv8a	693 bytes

Tabella 3.2: Dimensione media

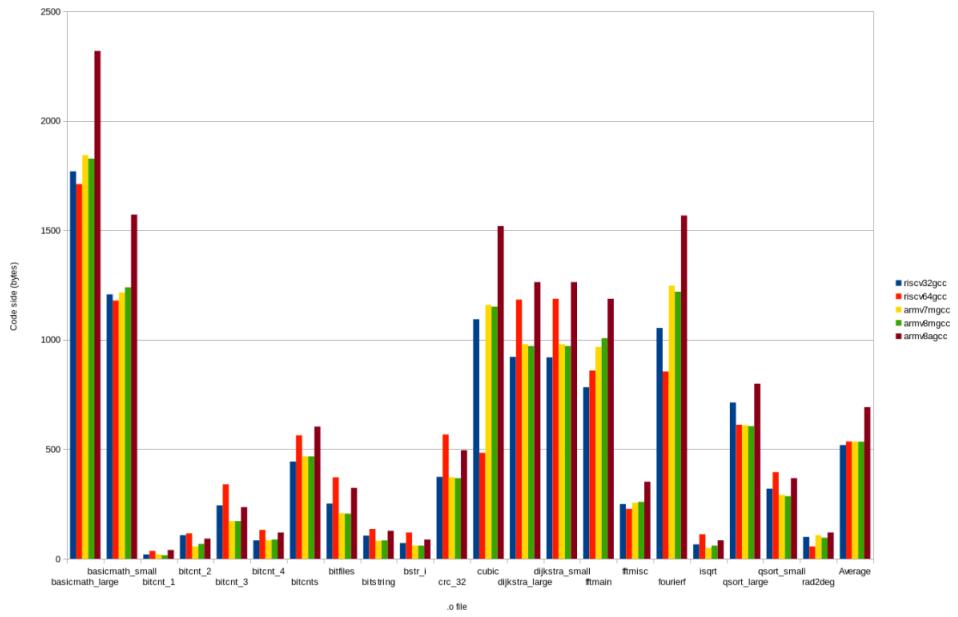


Figura 3.6: Confronto dimensione

### Confronto tra le architetture

In Conclusione le toolchain si comporta come aspettato. Il compilatore GCC produce codice più contenuto della controparte CLANG, le rispettive tool-chain si comportano allo stesso modo. Al momento, la toolchain GCC per RISC-V produce codice che è in media il 18% più piccolo del codice prodotto dalla toolchain LLVM per un sottoinsieme dei benchmark MiBench. Per lo stesso insieme di benchmark, il codice RISC-V 64 ha le stesse dimensioni del codice ARMv7m e ARMv8m. Il codice RISC-V 32 è circa il 3% più piccolo del codice RISC-V 64, ARMv7m e ARMv8m.

# Capitolo 4

## BenchMarking

Completa il  
capitolo 4

### 4.1 Descrizione board

La scheda di sviluppo utilizzata è D1-H Nezha basata sul design del chip Allwinner 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 sulla [Ogni](#) [repo](#)

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

directory contiene i dati di esecuzione di ciascun programma eseguito sulla board in un file con estensione .csv e un notebook jupyter per visualizzare i dati e, in alcuni casi confrontarli.

### 4.3.1 Operazioni Aritmetiche

Operazioni: la directory contiene un programma che esegue le operazioni aritmetiche di base

- addizione
- sottrazione
- moltiplicazione
- divisione
- 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 1 e 200, la seconda ha valori tra 1 e 100.

Per ogni matrice vengono eseguite 5 prove da cui viene calcolato il tempo medio di esecuzione

### 4.3.2 Prime Number

Il programma generai primi 5000 numeri primi. Viene calcolato il tempo di esecuzione. Nella directory sono presenti 2 programmi. Il primo 'prime.c' che controlla da 2 in poi se n-esimo numero è divisibile. Il secondo utilizza il crivello di Atkin per calcolare i numeri primi.

Analizza  
costo algoritmo

Info Sieve of  
Atkin

### 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 0 e maxInt e un array di potenze di 2 comprese tra 2 e 1024. Entrambi gli array hanno 1000 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 1000 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 5 volte e per ogni iterazione viene memorizzato il valore calcolato con il proprio tempo di esecuzione.

Valutazione  
generatore  
casuale

### 4.3.5 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 set 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.

Info algoritmi

#### 4.3.6 BackTracking

In questa directory sono contenuti due programmi:

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 Nel file sudoku.h sono presenti 8 sudoku 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 Questa directory è presente il file mazeGen.py che è stato utilizzato come generatore di labirinti. 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.

Labirinto personale Il file con estensione .txt, se si vuole utilizzare un altro labirinto, dovrà avere:

sulla prima linea il numero di linee del labirinto. sulla seconda linea il numero di colonne. dalla terza in poi il labirinto che dovrà avere un simbolo , come ad esempio '#', per indicare i muri, lo spazio ' ' per indicare i corridoi, uno e un solo simbolo 'I' per indicare l'inizio e uno e un solo simbolo 'O' per indicare l'uscita del labirinto. Sono presenti alcuni esempi nella sottodirectory maze di labirinti.

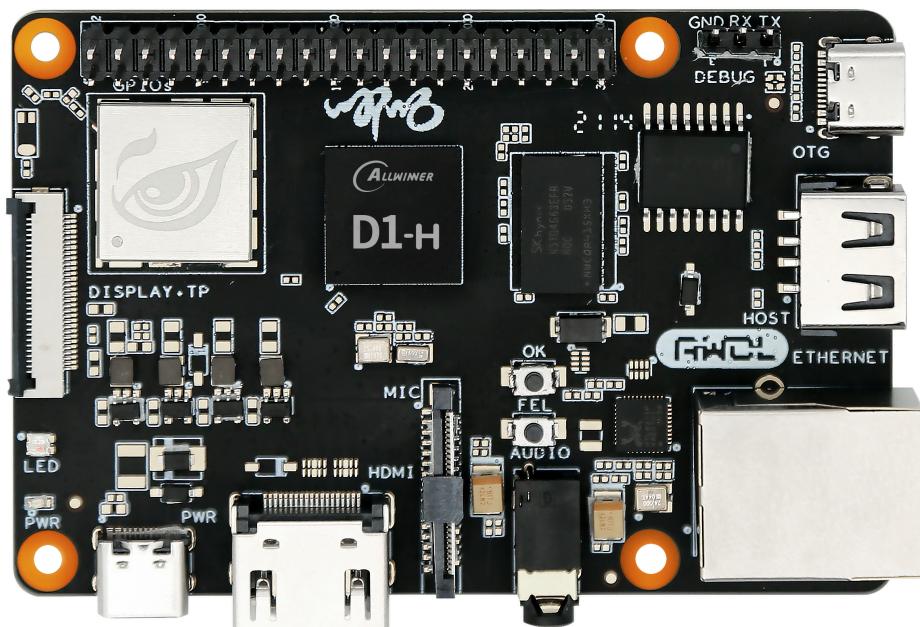


Figura 4.1: Board vista dall' alto

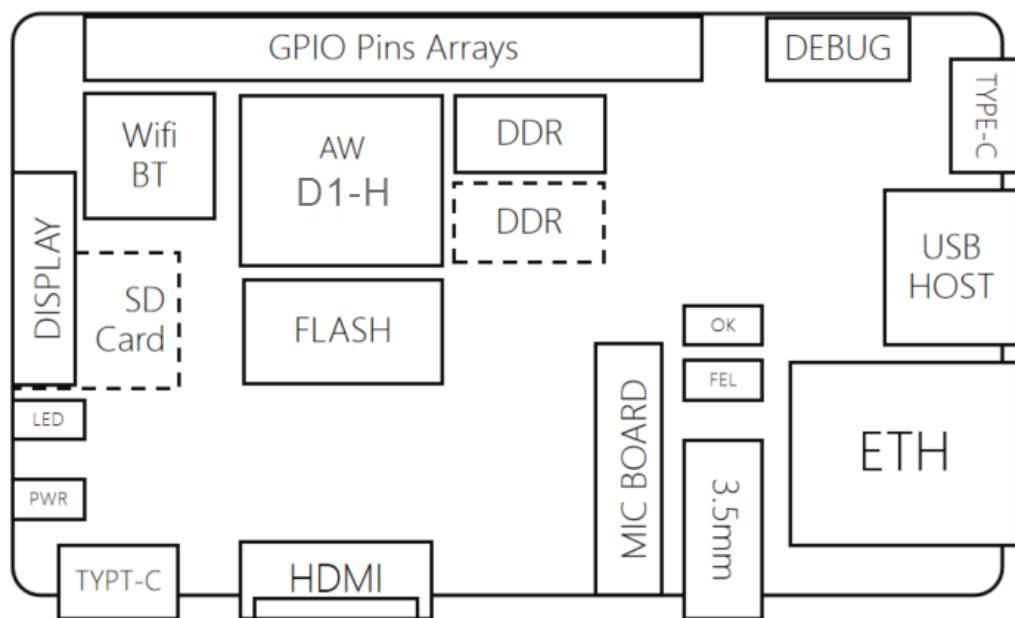


Figura 4.2: Schema a blocchi della scheda di sviluppo

# Capitolo 5

## Comparativa

### 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, cicla ogni elemento dell' array a e dell' array b, ne esegue l'operazione e memorizza il risultato in un array c. L'array a ha valori interi compresi tra 1 e 200, l'array b ha valori interi compresi tra 1 e 100, le matrici utilizzate sono matrici quadrate e le dimensioni utilizzate sono 1000, 2500, 5000, 8000, 10000, 15000. Per ogni operazione vengono eseguite 5 prove da cui viene calcolato il tempo medio di esecuzione.

Per riferimento il programma è stato utilizzato anche su un MacBook Air .

Ulteriori Info

```
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

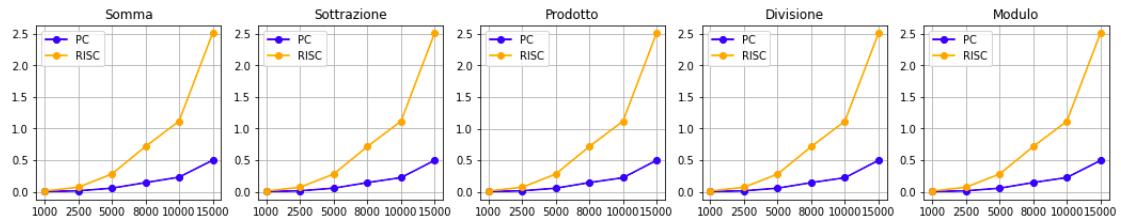
Di seguito vengono riportati i tempi di esecuzione medi per operazione e per dimensione

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  $1000 \times 1000$  è di 0.011256 e per la matrice  $2500 \times 2500$  è di 0.069917 il che è 6.2115 (circa) il tempo di esecuzione e 6.25 la dimensione dei dati. Per il MacBook il tempo di esecuzione per la matrice  $1000 \times 1000$  è di 0.002317 e per la matrice  $2500 \times 2500$  è di 0.013848 il che è 5.97 (circa) il tempo di esecuzione e 6.25 la dimensione dei dati. i tempi di esecuzioni della stessa operazione sono più o meno lo stesso valore e il rapporto rimane simile per tutte le operazioni. Il tempo di esecuzione totale del programma è di 23.653595 per Macbook mentre 117.665045 per RISC. Quindi il MacBook è 4.97 % più veloce del processore RISC, risultato abbastanza prevedibile. Confrontando le architetture il processore del MacBook è di 2.2 GHz con memoria da 8Gb da 1600MHz mentre il processore RISC è di 1 GHz con memoria da 2Gb a 1333 MHz. Avere una CPU da 1 Hertz significa avere un processore che può fare una cosa al secondo, quindi il processore del Mac può eseguire 2.2 miliardi di operazioni in un solo secondo di contro il processore RISC solamente 1 miliardo, meno della metà invece la memoria RAM il processore RISC è di 2Gb con 1333 MHz di frequenza e il MacBook 8Gb con 1600 MHz.

CALCOLA  
MIPS

## 5.2 PrimeNumber

### 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 0 e maxInt e un array di potenze di 2 comprese tra 2 e 1024. Entrambi gli array hanno 1000 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 1000 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 5.6: 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 5.7: 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
20 }
```

Code 5.8: Impostazioni dei dati

Il codice 5.6 imposta i due array. I codici 5.7 e 5.8 mostrano l'operazione eseguita. La tabella ?? 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 5.3: 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 5.9: Addizione

La funzione è una semplice funzione scritta in C che dato un numero di tipo intero restituisce il numero sommato a 23.

#### RISC-V

Il sorgente compilato con il compilatore RISC-V ?? da riga 2 fino a riga 6 predispone la chiamata della procedura posizionando sullo stack il necessario, da riga 7 inizia la funzione. Su quella riga viene recuperato il valore di num che alla riga 8, 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 23, il risultato è esteso su 64 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 5.1c , compilato con gcc ARM, mostra che la preparazione della procedura si esegue da riga 2 a riga 5, le due righe successive si esegue la funzione. La riga 6 recupera il valore di num la riga successiva calcola il valore del risultato e, infine, alla riga 8 si sposta il risultato nel registro di restituzione

#### x86

Il sorgente 5.1b è compilato con gcc di x86. Da riga 2 fino a riga 4 viene preparato lo stack, a riga 5 viene posizionato num nel registro eax che a riga 6 viene sommato a 23 che viene memorizzato nel registro eax. Infine viene ridato il controllo al chiamante.

<pre> 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 </pre>	<pre> 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 </pre>	<pre> 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 </pre>
(a) RISC-V	(b) ARM	(c) x86

Figura 5.1: Funzione di somma

### 5.5.2 Addizione

Nel caso generale viene calcolata la somma di 3 numeri.

```

1 int sumGen(int num, int num2, int num3) {
2     return num + num2 + num3 ;
3 }

```

Code 5.10: Addizione generale

Inserisci sorgenti Addizione generale

#### RISC-V

Nel caso del compilatore RISC-V la somma avviene tra 13 e 19 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 17.

#### ARM

Nel caso ARM avviene lo stesso meccanismo. Tra le righe 8 e 12 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 7 e 11 nello stesso modo con cui viene eseguito in ARM.

<pre> 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 </pre>	<pre> 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 </pre>	<pre> 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 </pre>
---	---	---

(a) RISC-V

(b) ARM

(c) x86

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 5.11: Moltiplicazione per potenza di 2

La funzione dato un numero di tipo intero restituisce il numero moltiplicato per 2. Per i sorgenti le parti di preparazione sono simili per le rispettive preparazioni precedenti.

Nel sorgente RISC-V ?? l'operazione di moltiplicazione per 2 avviene tramite uno shift logical left di 1 bit (SLLIW). Stesso concetto avviene nel sorgente ARM 5.2c dove l'operazione di moltiplicazione per 2 avviene tramite lo shift left di 1 bit. Invece nel sorgente x86 ?? la moltiplicazione avviene tramite una somma. Questa somma è un caso particolare, infatti se volesse moltiplicare per una qualsiasi potenza di 2 (ad esempio in figura 3) le operazioni avvengono tutte tramite shift left di un opportuno valore. Con la figura 4 viene mostrato il calcolo di una moltiplicazione per 8. Nei casi RISC-V e ARM il calcolo avviene attraverso Shift e nel caso x86 avviene anche qui uno shift.

#### Moltiplicazione per una costante

<pre> 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 </pre>	<pre> 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     lr </pre>	<pre> mult2:     push   rbp     mov    rbp, rsp     mov    DWORD PTR [rbp     ↪ -4], edi     mov    eax, DWORD PTR     ↪ [rbp-4]     add    eax, eax     pop    rbp     ret </pre>
--	---	--

(a) RISC-V

(b) ARM

(c) x86

Figura 5.3: Moltiplicazione per 2

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

(a) RISC-V

(b) ARM

(c) x86

Figura 5.4: Moltiplicazione per 8

```

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

Code (5.12) Moltiplicazione per potenza di 2

```

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

Code (5.13) Moltiplicazione per potenza di 2

Vengono presentati due codici molto simili, il primo moltiplica il numero per 31 che rappresenta più in generale un numero che dista da una potenza di 2 di 1. 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 2, nel nostro caso il numero è 30.

Nel caso della moltiplicazione per 31 l'approccio dei 3 sorgenti è identico. Viene calcolata la moltiplicazione per 32 attraverso shift logici e poi viene sottratto una volta il valore per ottenere la moltiplicazione per 31. Se il valore costante fosse 33 , il numero successivo alla potenza di 2, l'operazione di sottrazione viene sostituita con una di addizione. Nel caso più generale invece abbiamo un approccio differente.

Partendo dall x86 la moltiplicazione avviene semplicemente con l'istruzione imul a riga 6. Nel caso ARM l'operazione di moltiplicazione viene comunque eseguita da una singola istruzione(riga 8) ma vengono utilizzati i registri r2 e r3 che precedentemente (riga 6 e 7) vengono riempiti con gli operandi. Infine l' implementazione di RISC-V utilizza ancora shift. Nel caso della moltiplicazione per 30 avviene prima uno shift di 4 ( moltiplicazione per 16) successivamente sottratto una volta il numero e infine al risultato avviene applicato uno shift di 1 (moltiplicazione per 2).

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

[add ref para](#)

combinate con addizioni e sottrazioni per ottenere il valore della costante.

<pre> 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 </pre>	<pre> 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 </pre>	<pre> 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 </pre>
--	--	--

(a) RISC-V

(b) ARM

(c) x86

Figura 5.6: Moltiplicazione per 31

<pre> 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 </pre>	<pre> 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    lr </pre>	<pre> 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 </pre>
--	---	--

(a) RISC-V

(b) ARM

(c) x86

Figura 5.7: Moltiplicazione per 30

Controllo  
eff pseudo  
istruzioni

### 5.5.4 Divisione

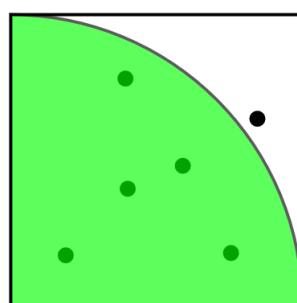
Per quanto riguarda la divisione viene utilizzato la stessa metodologia della moltiplicazione.

## 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 0 e 100 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 10 valori e questi sono 1,2,3,4,5,6,7,8,9,10 notiamo subito una sequenza riconoscibile e la casualità del prossimo numero è poco randomica. Quindi le proprietà che ci interessano per un generatore di numeri sono una distribuzione uniforme e la non predicitività dell' n+1 esimo numero dopo n numeri.

Inserisci va-  
lutazione  
generatore  
numeri ca-  
suali

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. In pratica il metodo funziona così: si generano numeri casuali compresi tra 0 e 1 che rappresentano la coordinata x e y del nostro punto. Applicando il teorema di Pitagora se l'ipotenusa supera 1 significa che il nostro tiro è finito nell'area del quadrato, se invece è minore di 1 sta a significare che il nostro tiro è finito all'interno dell'area del cerchio (e ovviamente del quadrato).



Una volta generati i punti e valutati se ricadono all'interno del primo quadrante del cerchio il calcolo del valore del pigreco è facile:

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

```

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 5.14: 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.

Le tabelle 5.4 e 5.5 mostrano i risultati in termini di valori calcolati e il tempo di esecuzione. Le colonne 1,2,3,4,5 indicano il valore dell'i-esima prova mentre ogni riga identifica il numero di punti generati.

## 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.

	1	2	3	4	5	T.EXE Totale	MI
$10^1$	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 5.4: Tempi di esecuzione dell'algoritmo

	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 5.5: Valori calcolati

- 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 riferi-

mento 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 5.6: complessità bubble sort

Di seguito viene riportato il codice (Code 5.15) dell'algoritmo e i tempi di esecuzione (Tab 5.7) 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 5.15: 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 5.7: Tempi di esecuzione bubble sort

### 5.7.2 Insertion sort

L'algoritmo 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 5.8: 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 5.16: 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 5.9: 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         while (v[j] > v[in])
20             j--;
21         if (i <= j)
22         {
23
24             int t = v[i];
25             v[i] = v[j];
26             v[j] = t;
27         }
28     }
29
30     int tt = v[in];
31     v[in] = v[i - 1];
32     v[i - 1] = tt;
```

```

34     return i - 1;
35 }
36

```

Code 5.17: 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 5.10: Tempi di esecuzione quick sort

### 5.7.4 Heap sort

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

info algo

```

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     {
31         heapify(arr, n, i);
32     }
33     for (int i = n - 1; i >= 0; i--)
34     {
35         swap(&arr[0], &arr[i]);
36         heapify(arr, i, 0);
37     }
38 }
```

Code 5.18: 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 5.11: Tempi di esecuzione heap sort

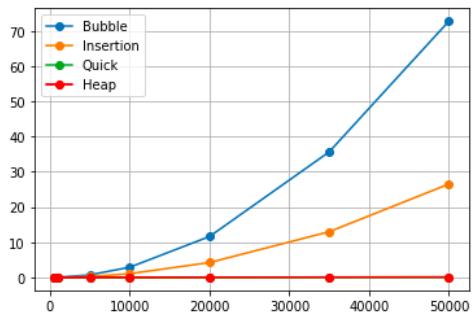
Come mostrato dai grafici 5.8 i risultati mostrano che il peggior algoritmo è il bubblesort mentre il migliore è quicksort. Risultato poco notevole, nella sezione successiva un confronto più significativo.

### 5.7.5 Confronto Sorting

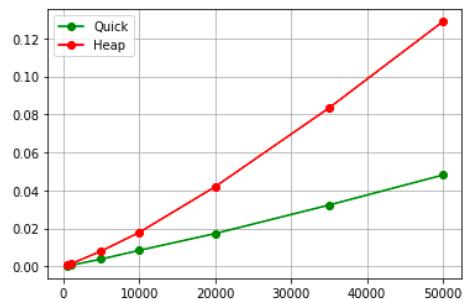
In questa sezione vengono visualizzati i risultati a confronto. Per prima cosa iniziamo con confrontare i risultati con l'esecuzione su MacBook.

La tabella 5.12 mostra i tempi di esecuzione degli algoritmi di ordinamento. Confrontando i tempi del PC e si nota che in generale si ha, circa, un ordine di grandezza di differenza ricordando i 2.2 GHz del processore e 8Gb da 1600MHz del MacBook confrontati con il processore da 1 GHz con memoria da 2Gb a 1333 MHz vengono tradotti in un miglior tempo di esecuzione di un fattore 10. Un altro confronto lo possiamo fare con raspberry pi.

Inserisci immagine



(a) Bubblesort insertionsort, heapsort e quicksort



(b) Quicksort e Heapsort

Figura 5.8: Algoritmi di ordinamento a confronto

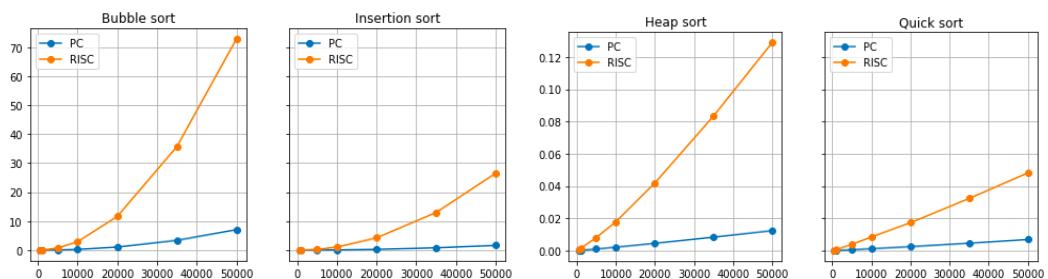


Figura 5.9: Algoritmi di ordinamento a confronto eseguiti su PC e su RISC-V

	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 5.12: Tempi di esecuzione degli algoritmi di sorting su PC

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 5.13 mostra i tempi di esecuzione degli algoritmi di ordinamento eseguiti su Raspberry.

	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 5.13: Tempi di esecuzione Raspberry Pi B

I grafici 5.10 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. Possiamo concludere che il miglioramento è dato dalla CPU e dalla memoria.

More

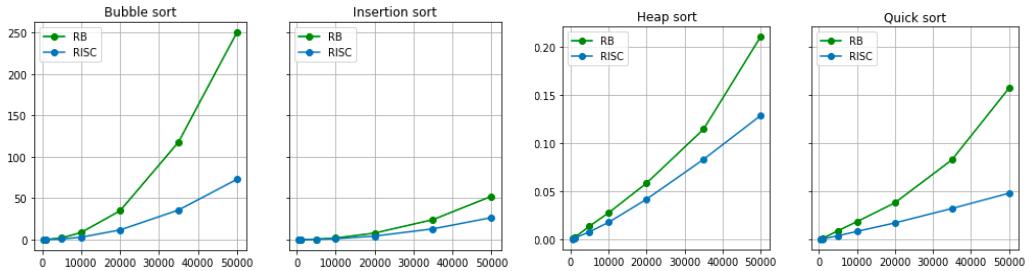


Figura 5.10: Algoritmi di ordinamento a confronto eseguiti su Raspberry e su RISC-V

	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 5.14: Tempi di esecuzione del solutore di sudoku

## 5.8 BackTracking

Un altro problema che si puo 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 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. [More](#)

	RISC	PC
maze0	0.000102	0.000007
maze1	0.005261	0.000235
maze2	0.009123	0.000371

Tabella 5.15: Tempi di esecuzione del solutore di labirinti

# **Capitolo 6**

## **Progetti**

**DESKTOP CON PROCESSORE RISC-V SEE BIB**

# Capitolo 7

## Dibattito ARM vs RISC-V

rileggi ...

Quando si parla di processori una cosa importante da prendere in considerazione è ISA, quando si vuole scrivere un programma è molto utile conoscere il target per il proprio codice. Nell'ecosistema eterogeneo degli ISA i più utilizzati sono x86 e ARM che hanno un approccio CISC e RISC rispettivamente. Le due ISA vengono applicate a sistemi ben differenti, ad esempio il sistema sviluppato da Intel si applica a sistemi di grandi dimensione come i sistemi HPC e i cluster mentre l'ambiente ARM viene utilizzato a sistemi ben più piccoli. In questo contesto RISC-V, con il suo approccio RISC, sembrerebbe che voglia sostituirsi ARM.

Facciamo confronto con ARM e RISC il primo è basato su un IP proprietario che viene concesso alle società con un numero relativamente contenuto di fornitori, RISC invece ha un'architettura delle specifiche royalty-free e i processori sono disponibili e esistono anche core con licenza commerciale. Le community hanno dimensioni differenti, più strutturata quella di ARM che aiuta i progettisti per specifiche applicazioni, contrapposta è quella di RISC, che con poco più di 10 anni di vita la community è più contenuta ma continua crescita. ARM dispone di team di ingegneri che sviluppano sistemi hardware che rendono facile per i progettisti incorporare CPU ARM, mentre i progettisti possono sperimentare e sviluppare sistemi RISC-V gratuitamente, c'è poco o nessun supporto per la progettazione dell'hardware. Per concludere L'ISA RISC-V è organizzato in gruppi di istruzioni e offre la possibilità di utilizzare estensioni fornendo allo standard ISA un supporto per applicazioni specifiche. Ad esempio, l'estensione RISC-V Vector (RVV) recentemente rilasciata consente ai core del processore basati su RISC-V ISA di elaborare array di dati insieme alle tradizionali operazioni scalari, per accelerare il calcolo di singoli flussi di istruzioni su grandi insiemi di dati. In contraddizione con l'approccio iniziale di ARM che non forniva estendibilità ma solo in seguito.

Passiamo ora a sistemi un po più grandi. L'azienda E4 Computer Engineering ha progettato il suo primo Arm-based cluster nel 2012 (INSERISCI NOTE BIB). Oggi l' E4 sta integrando un cluster basato sul processore HiFive Unmatched (di SiFive). Questa azione inaugura una nuova era di sviluppo RISC-V basato su Linux. Il processore utilizzato è basatosu RISC-V che utilizza multi-core ad alte prestazioni, 64 bit dual-issue e superscalare. In questo modo si fornisce l'accesso ad entrambe le architetture ARM e RISC-V, cosicchè gli utenti e sviluppatori possono testare ampiamente le loro applicazioni e codici e fornire indicazioni per trovare la migliore soluzione complessiva ai loro requisiti.

L'uso di processori RISC-V è crescente, le grandi aziende stanno già cercando alternative ad ARM. Anche se ARM è stata acquisita da NVIDIA, RISC-V si fa sentire sul mercato. Alcuni dei suoi partner sono Google, Intel e Huawei o altri come Arduino, AntMicro e Oculus.

Tuttavia, solo perché qualcosa è gratuito non significa che prenderà il comando. Prendiamo come esempio il sistema Linux la maggior parte delle distribuzioni sono gratuite, ma Linux costituisce una piccola percentuale dei sistemi operativi in tutto il mondo. Con il migliorare della tecnologia il supporto a RISC-V aumenterà, grazie anche alla community, che porterà i progettisti ad avere un'opzione di un'architettura gratuita senza limitazioni e altamente configurabile ed estendibile ma non si ha certezza se prenderà una grossa fetta di mercato.

# **Capitolo 8**

## **Conclusione**