Calcolatori

Mattia Marini

1 ottobre 2025

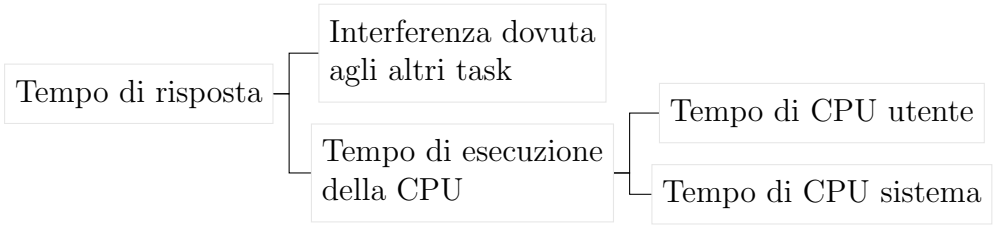
Indice

Comandi

1 Misurazione delle prestazioni

Quendo di parla di prestazioni di una macchina spesso si è interessati a 2 cose

- *Tempo medio di risposta* - per il singolo utente
- *Throughput* - per un centro di calcolo (es. server)



Nel calcolo del tempo necessario per eseguire un programma devo tenere in consiredazione 3 elementi:

- Numero di istruzioni
- Cicli di clock per istruzione (CPI)
- Frequenza di clock della CPU

allora avrò che :

$$\begin{aligned}\text{Tempo CPU} &= \text{Numero Istruzioni} \times \text{CPU} \times \text{Periodo Clock} \\ &= \frac{\text{Numero Istruzioni} \times \text{CPU}}{\text{Frequenza Clock}}\end{aligned}$$

2 Numeri e calcolatori

Possiamo rappresentare un numero n in base b nel seguente modo

$$n = \sum_{i=0}^{k} c_i b^i$$

dove c_i è l'i-esima cifra di n. scriveremno $n = (c_0 \dots c_k)_b$

2.0.0 Base binaria ed esadecimale

I computer "leggono" codice binario in quanto sono in grado di rappresentare gli stati di 1 e 0 tramire stati di alta e bassa tensione in un transistor.

Un'importante caratteristica della base 16 è che 4 cifre di un numero binario corrispondono alla corrispondente cifra di un numero esadecimale. Vediamo un esempio:

$$(0011\ 1110\ 1000)_2 = 3E8_{16}$$

in quanto $0011 = 3$, $1110 = 14 = E$ e $1000 = 8$

2.1 Conversioni di basi

2.1.0 Base 16 $\Leftrightarrow$ base 2

Basta convertire 4 bit alla volta come visto nella sezione precedente

2.1.0 Base $b \Rightarrow$ base 10

Basta considerare che

$$n = \sum_{i=0}^{k} c_i b^i$$

2.1.0 Base 10 $\Rightarrow$ base b

Bisogna dividere in modo iterativo per b, segnandosi i resti per poi leggerli al contrario. Ad esempio convertendo 1421 in base 9

x	$x/9$	$x\%9$
1421	157	8
157	17	4
17	1	8
1	0	1

- Divido iterativamente
- Segno i resti
- Leggo al contrario i resti, ottenendo 1848

2.2 Operazioni

2.2.0 Somma

	1	1	1	0	1	0	1	+
	1	1	0	0	0	0	1	=
1	1	0	1	0	1	1	0	

nota che se ho $1 + 1$ con riporto di 1 allora farà 1

2.2.0 Sottrazione

1	1	1	0	1	0	1	-
1	1	0	0	0	0	1	=
0	0	1	0	1	0	0	

2.3 Rappresentazione numeri interi

Abbiamo principalmente 3 modi di rappresentare anche i numeri interi negativi $\mathbb{Z}$.

2.3.0 Modulo e segno

Il bit più significativo indica il segno del numero.

- $1 \rightarrow$ segno negativo
- $0 \rightarrow$ segno positivo

$$011 = 3 \qquad 001 = 1$$
$$111 = -3 \qquad 101 = -1$$

NB: lo zero ammette rappresentazione doppia: $100_2 = 000_2 = 0$

2.3.0 Complemento a 1

Il corrisondente negativo di un numero naturale è ottenuto effettuando il complementare bit a bit

- Numero inizia per 1 $\rightarrow$ negativo
- Numero inizia per 0 $\rightarrow$ positivo

$$011 = 3 \qquad 001 = 1$$
$$100 = -3 \qquad 110 = -1$$

NB: lo zero ammette rappresentazione doppia: $100_2 = 000_2 = 0$

Per sommare due numeri in complemento a 1 devo:

- Somma la rappresentazione dei due numeri
- Se ho riporto sulla cifra più significativa, allora aggiungo 1 al risultato
- Occhio agli overflow (*se hai due positivi non puoi ottenere un negativo, e viceversa*)

2.3.0 Complemento a 2

In questo caso il corrispondente alternativo di un numero naturale è ottenuto effettuando il complemento a 1 di questo solo sulla porzione sinistra fino all'ultimo 1 verso destra (non comperso)

- Scorro numero da destra a sinistra finchè incontro un 1
- Effettuo complemento a 1 solo sulla porzione alla sinistra di questo 1 (non compreso)
- Numero inizia per 1 $\rightarrow$ negativo
- Numero inizia per 0 $\rightarrow$ positivo

$$011 = 3 \qquad 001 = 1$$
$$101 = -3 \qquad 111 = -1$$

NB: lo zero ammette rappresentazione univoca

Per sommare due numeri in complemento a 2 devo:

- Somma la rappresentazione dei due numeri
- Occhio agli overflow (*se hai due positivi non puoi ottenere un negativo, e viceversa*)

Decimale	Modulo e segno	Complemento a 1	Complemento a 2
-3	/	/	101
-2	110	100	110
-1	101	101	111
0	100/000	111/000	000
1	001	001	001
2	010	010	010
3	011	011	011

Tabella 1: Tabella riassuntiva delle diverse codifiche

2.4 Rappresentazione numeri con la virgola

2.4.0 Rappresentazione a virgola fissa

Se ho n bit ne riservo k per la parte intera e $n-k$ per la parte decimale.

$$101100_2 = \left(1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0\right) + \left(1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3}\right) = 5.5$$

Per convertire nel senso opposo converto la parte intera normalmente e moltiplico iterativamente la parte decimale per 2 ricordandomi la parte intera:

$$6.125_{10} =?_2$$

parte intera $6_{10} = 110_2$

Conto	parte intera
$0.125 \cdot 2 = 0.25$	0
$0.25 \cdot 2 = 0.5$	0
$0.5 \cdot 2 = 1$	1

dunque ho che

$$\begin{aligned} \text{parte intera} &= 110 \\ \text{parte decimale} &= 001 \end{aligned}$$

Il numero è 110.001

2.4.0 Rappresentazione a virgola mobile

Il numero viene rappresentato con la notazione scientifica $(-1)^s \cdot M^{E'}$ dove s è segno, M mantissa e E' esponente

- Esponente: viene shiftato in modo tale da poterne rappresentare di negativi
  - Ad esempio, ho 8 bit per l'esponente. Il numero massimo rappresentabile è $2^8 - 1 = 255$
  - se shifto il numero di 255/2 posso ottenere numeri nel range $(-127, 127]$

- Più in generale, dato e il numero di bit riservati all'esponente, allora il numero rappresentato dall'esponente è E:

$$-2^{e-1}+1 < E \leq 2^{e-1}-1$$

inoltre avrò che

- Se $E' > 0$ numero normalizzato
  - Mantissa sottointende un 1. (es. se $M = 101$, allora la base è 1.101)
  - $E = E' - (2^{e-1}-1)$
- Se $E' = 0$ numero denormalizzato
  - Mantissa sottointende un 0. (es. se $M = 101$, allora la base è 0.101)
  - $E = -(2^{e-1}-2)$

In quest'ultimo caso se ci pensi ha senso che ci sia $E = -(2^{e-1}-2)$ in quanto non viene sottointeso un 1, bensì uno 0

2.4.0 Dati in c

- `float`: precisione singola
  - $k = 32$
  - $e = 8, \quad m = 23$
- `double`: precisione doppia
  - $k = 64$
  - $e = 11, \quad m = 52$
- `long double`: precisione estesa o quadrupla
  - $k = 80/128$
  - $e = 15, \quad m = 64/112$

Inoltre sono presenti dei valori particolari dei dati che vengono riservati e tornano molto utili

Categoria	E'	M	s
Numeri normalizzati	1 - 254	qualunque	0/1
Numeri denormalizzati	0	non zero	0/1
$\pm$ zero	0	0	0/1
$\pm\infty$	255	0	0/1
NaN (Not a Number)	255	non zero	0/1

3 Reti logiche

Le reti logiche sono circuiti che trasformano valori in ingresso in altri in uscita, secondo determinati criteri. I valori sono sempre stati di alta tensione (1) o bassa tensione (0). Si dividono in: 3.0.0 Combinatorie 3.0.0 Sequenziali

- Comportamento funzionale *input* → *output*
- Non hanno memoria (l'out dipende solo dall'in)
- L'output dipende dalla storia delle operazioni eseguite sulla medesima rete
- Hanno memoria (l'output è influenzato dallo *stato* della rete)

3.1 Rappresentazione reti logiche

Una rete logica può essere rappresentata da una tabella che elenca i valori di output in corrispondenza di ogni possibile combinazione di valori di ingresso, ad esempio:

INPUT			OUTPUT		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Alternativamente si possono specificare reti logiche tramite *l'algebra di Boole* tramite gli operatori or$(+)$, and$(\cdot)$ e not $(\overline{A})$. La tabella a sinistra corrisponde alle seguenti espressioni:

$$D = A + B + C$$
$$E = \left(A \cdot B \cdot \overline{C}\right) + \left(A \cdot C \cdot \overline{B}\right) + \left(B \cdot C \cdot \overline{A}\right)$$
$$F = A \cdot B \cdot C$$

Nota che mentre ricavare D e F in funzione dei tre input è scontato, per E è più laborioso. Potremmo generalizzare un algoritmo per ricavare un'espressione booleana da una tabella:

- Considero un valore di output (nel nostro caso E)
- Considero ogni combinazione di input che fa si che il valore considerato sia 1 (nel nostro caso le combinazioni a righe 4,6,7)
- Esprimo il valore considerato come *or* di *and*

Ossio posso dire che E sarà 1 quando B e C sono 1 e A non lo è $(B \cdot C \cdot \overline{A})$, oppure quando $\left(A \cdot C \cdot \overline{B}\right)$ oppure $\left(A \cdot B \cdot \overline{C}\right)$. Questo modo di procedere verrà approfondito in seguito e si chiamerà forma canonica SP

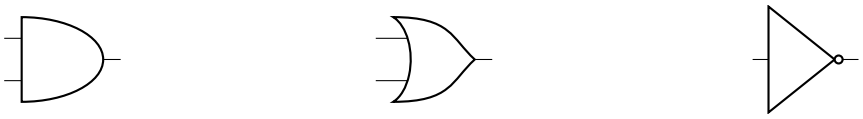
3.2 Proprietà espressioni logiche

- *Identità:* $A + 0 = A$ e $A \cdot 0 = 0$
- *Regola zero e uno:* $A + 1 = 1$ e $A \cdot 0 = 0$
- *Regola dell'inversa:* $A + \overline{A} = 1$ e $A \cdot \overline{A} = 0$
- *Proprietà commutativa:* $A + B = B + A$ e $A \cdot B = B \cdot A$

- *Proprietà associativa:* $(A + B) + C = A + (B + C)$ e $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
- *Proprietà distibutiva:* $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ e $A + (B \cdot C) = (A + B) \cdot (A \cdot C)$
- *Regole di De Morgan:* $\overline{A \cdot B} = \overline{A} + \overline{B}$ e $\overline{(A + B)} = \overline{A} \cdot \overline{B}$

Queste ultime due proprietà sono importantissime visto che per via di queta proprietà ogni espressione logica può essere rappresentata tramite operatori **NAND**(not and) e **NOR**(not or). Questo è usato nelle cpu, per semplificarne la costruzione

3.3 Porte logiche



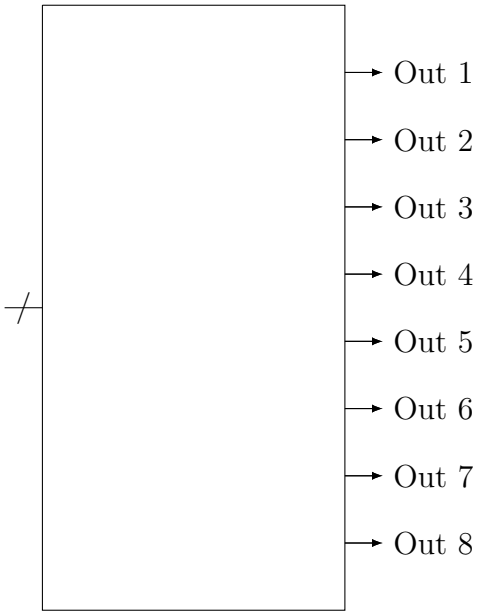
(a) *AND* (b) *OR* (c) *NOT*

Figura 1: Porte logiche and, or e not

3.4 Circuiti combinatori

3.4.0 Decoder

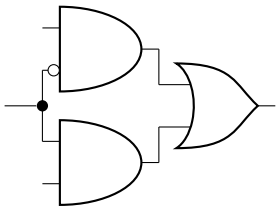
Ad ogni combinazione di input si associa accende un solo output. Vediamo un esempio di un decode con ingresso a 3 bit



Il dispositivo qui a sinistra prende 3 input e, per ogni input, vi sarà solo uno degli 8 output che si attiveranno

Inputs			Outputs							
In2	In1	Ino	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

3.4.0 Multiplexer



Il multiplexer è un dispositivo che funziona come una leva selettrice. Dati due input ed un terzo di selezione permette di dare in output uno dei due in base al valore del inout selettore

3.4.0 Multiplexer a più vie

Un multiplexer può essere un selettore a molteplici ingressi. Questo fa uso di un decoder

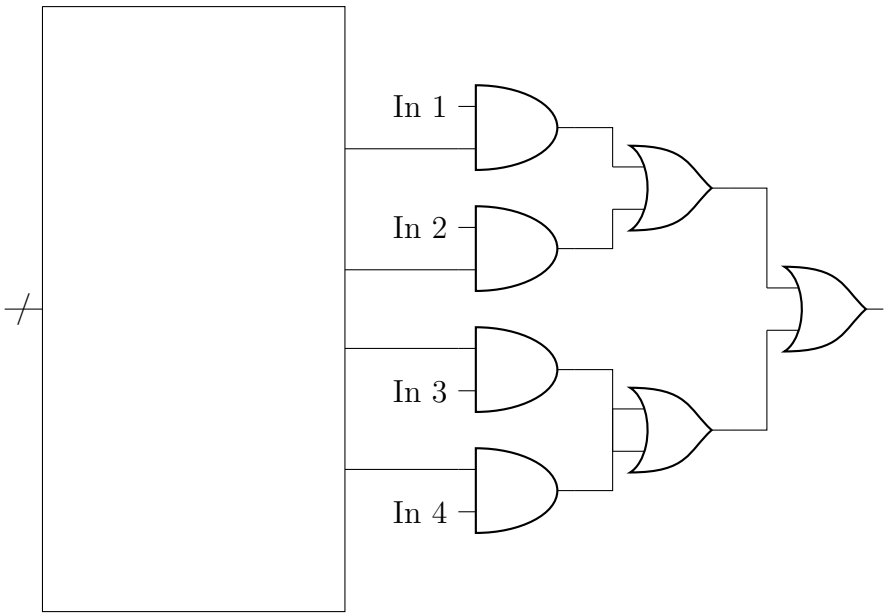
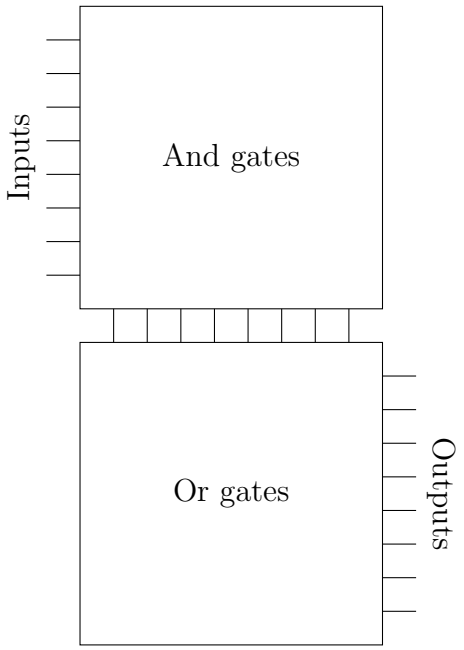


Figura 2: Multiplexer a più vie

3.4.0 Programmable Logic Array (PLA)

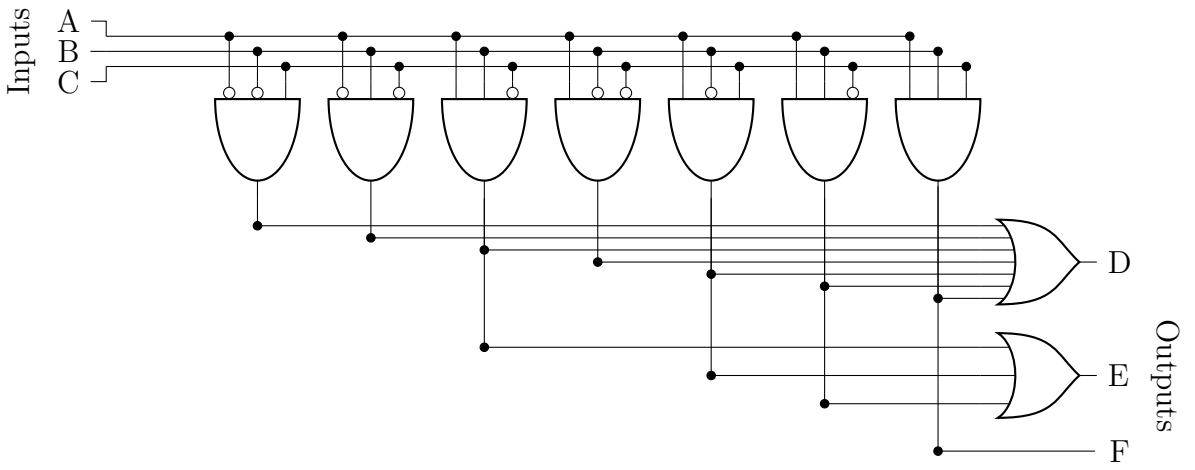


Come visto in *sezione* **??**, possiamo convertire qualsiasi espressione da da una tabella logica ad un'espressione booleana seguendo il principio di *forma canonica SP*, ossia utilizzando una serie di *or* fra gruppi di *and*. Lo stesso procedimento può essere applicato alle porte logiche come riportato a sinistra.

Vediamo un esempio. Cerchiamo di rappresentare le seguenti espressioni tramite un *PLA*:

$$D = A + B + C$$
$$F = A \cdot B \cdot C$$
$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

In questo caso potrei rappresentare queste espressioni tramite il seguente circuito:



3.5 Costo e ottimizzazione operazioni

Chiaramente per massimizzare l'efficienza è opportuno ridurre al minimo il numero di operazioni da effettuare. Per fare ciò possiamo applicare l'algebra di boole, ad esempio

$$\begin{aligned} f(x_1, x_2, x_3) &= \bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2x_3 + x_1\bar{x}_2\bar{x}_3 + x_1\bar{x}_2x_3 \\ &= \bar{x}_1\bar{x}_2(\bar{x}_3 + x_3) + x_1\bar{x}_2(\bar{x}_3 + x3) \\ &= \bar{x}_1\bar{x}_2 + x_1\bar{x}_2 \\ &= (\bar{x}_1 + x_1)\bar{x}_2 \\ &= \bar{x}_2 \end{aligned}$$

tuttavia la semplificazione delle reti logiche non è sempre una mansione semplice. Talvolta ci si basa sull'applicazione iterativa delle regole dell'*algebra di boole*, per casi semplici ci si appooggia a metodi grafici (*mappe di Karnaugh*). Questo topic è approfondito nel corso Reti Logiche

3.6 Circuiti sequenziali

I circuiti sequenziali sono necessari l'addove viene introdotto qualche meccanismo in grado di memorizzare dei dati. Tale memorizzazione avviene tramite un meccanismo di retroretazione, ossia reindirizzando l'uscita del circuito all'entrata del circuito stesso. Vediamo un esempio:

3.6.0 R-S Latch

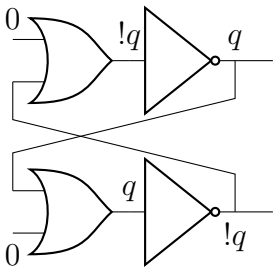
(a) Set

(b) Reset

Nelle operazioni di reset e set nota che:

- La porta or che vede come input un 1 chiaramente avrà come output 1, che negato diventerà 0
- Lo 0 verra retropropagato alla porta or con ingresso 0 che darà come out 0, che negato diventerà 1
- Abbiamo ottenuto così una situazione stabile

Chiaramente il discorso vale anche per l'operazione di reset. Diverso è il discorso per l'operazione di store



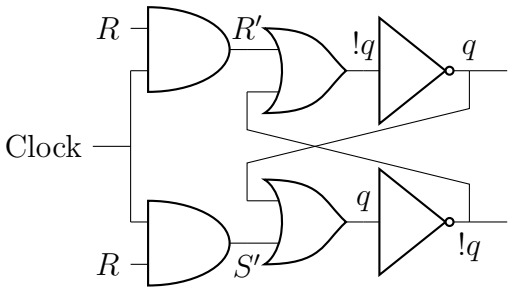
In questo caso la situazioe è la seguente:

- Suppiniamo di fare un fermaimmagine e cambiare improvvismente l'input alle porte or
- Avendo una porta or con un'estremità uguale a 0 l'output coinciderà con l'input proveniente dalla porta non nulla
- Otteniamo così anche qui una situazione stabile: supponendo di avere un segnale di tipo q e $!q$ in uscita dalle porte *or* finchè verra dato in ingresso 0 ad entrambe si avrà in uscita sempre q e $!q$

NB: se si fornisce 1 ad entrambe le porte *or* il circuito entrerebbe in uno stato indeterministico. Ciò vuol dire che non è possibile stabilire l'output del circuito

3.6.0 Clock

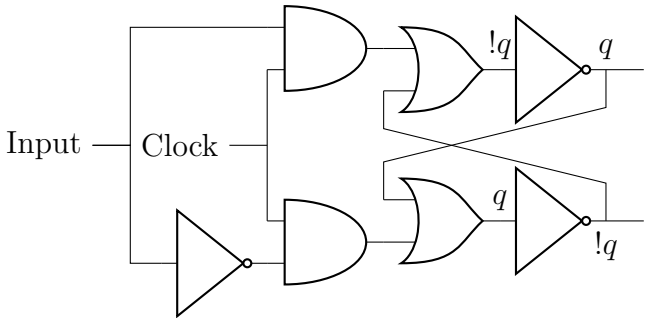
Per negare la possibilità di effettuare operazioni di *set* e *reset* in determinati momenti è possibile aggiungere un terzo input al *R-S Latch*:



In questo caso se il clock è 0 non possono avvenire transizioni di stato (stato *no change*). Se invece il clock vale 1 è come se le due porte *and* non esistano, ed il circuito funziona come spiegato sopra

3.6.0 D-Latch

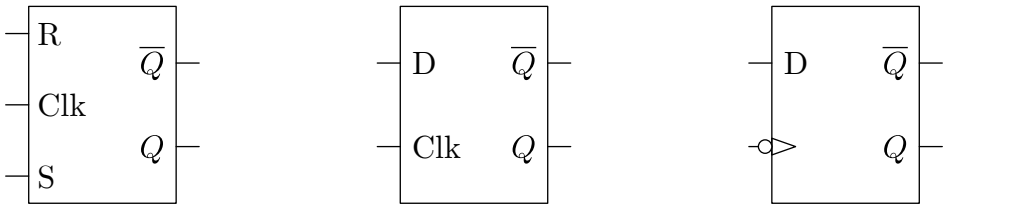
Per risolvere il problema del possibile input di due 1 in un *R-S Latch* un input di clock. Il circuito è il seguente:



Negando uno dei due ingressi non è possibile che si presenti il caso in cui entrambi siano 1. L'operazione di *store*(che richiede entrambi gli input nulli) verrà effettuata solo tramite il clock

3.6.0 Rappresentazione compatta latch

Visto che questi componenti sono estremamente comuni, si possono rappresentare in versone compatta come segue, all'interno di un circuito:

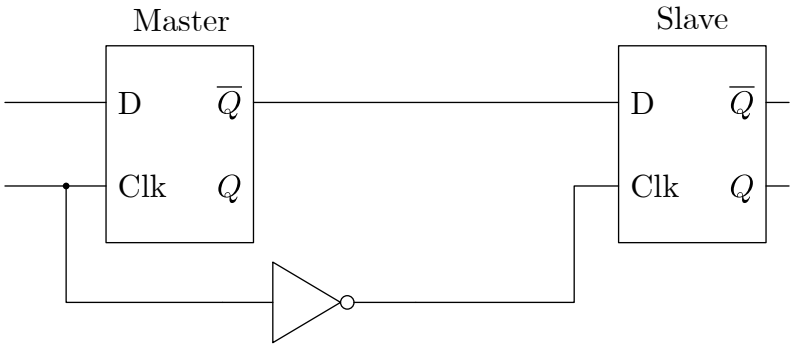


(a) RS_latch (b) D_latch (c) Flip Flop Master Slave

Figura 4: Rappresentazione simbolica latch

3.6.0 Flip Flop master-slave

Questo tipo di *latch* permette di effettuare la memorizzazione esattamente alla fine del ciclo di clock. Il circuito è il seguente:



In questo caso abbiamo che:

- Se il *clock* è 1 allora il latch *slave* è "inattivo" (stato di *store*), mentre *master* sarà sensibile a *set* e *reset*
- Se il *clock* è 0 allora *master* memorizzerà lo stato e setterà il latch *slave* si conseguenza
- In quessto caso quindi, la memorizzazione avviene quando il clock diventa 0

4 Terminologia e definizioni

4.0.0 Funzionamento cpu

Ogni processore è in grado di comunicare solo fornendo in ingresso delle stringhe di dati, che possiedono una data codifica la quale cambia da cpu a cpu. In particolare una cpu funziona nel modo seguente:

- *Fetch* l'istruzione da eseguire viene prelevata dalla memoria. Esiste un particolare registro (Program Counter PC/ Instructions Pointer (IP))
- *Decode* l'istruzione (stringa di byte) viene decodificata
- *Execute* l'istruzione decodificata viene eseguita

Il linguaggio assembly permette di chiamare queste procedure rese possibili dalla cpu, seza scrivere codice binario. Ogni qualvolta si chiami una procedura assembly questa viene mappata dunque in una stringa di bit (solitamente 32)

4.0.0 Registri

Un registro non è altro che un'area di memoria vicino al processore ad accesso rapido. Quest'area è composta da batterie di registri flip flop di tipo D (k registri per k byte). In genere sono presenti dai 4 ai 64 registri

4.0.0 Macchina a 32/64 bit

Quando si dice che una macchina è a x bit si intende che i registri di questa hanno tutti dimensione x bit

4.0.0 Instructions Set Architecture

L'Instruction Set Architecture è l'insieme di regole per interfacciarsi con una determinata cpu. In poche parole sono le stringhe di bit che sono interpretaili da una cpu. Le seguenti cose sono definite all'interno dell'isa di una cpu:

- Sintassi e semantica
- Istruzioni disponibili
- Numero di registri, tipo e dimensione
- Modalità di accesso alla memoria (*indirizzamento*)
- Istruzioni assembly disponibili

Alcuni registri sono messi a disposizione per eseguire e memorizzarci temporaneamente dati (i cosiddetti registri *general-purpose*) da parte dell'utente, mentre altri sono riservati e contengono dati specifici

Esistono 3 macrocategorie di ISA:

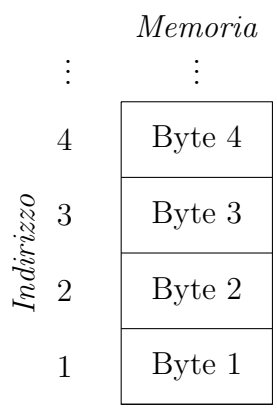
- *RISC* Reduced Instruction Set Computer
  - Semplice e regolare
  - Istruzioni lavorano su registri e non su memoria direttamente
  - Accesso alla memoria solo tramite 2 istruzioni: *load* e *store*
  - Molte meno istruzioni e meno potenti
  - Cpu molto più facile da costruire
- *CISC* Complex Instruction Set Computer
  - Molte più istruzioni più complesse e più potenti
  - Ogni istruzione può accedere ai registri o direttamente alla memoria
- *ARM* Advanced RISC Machine
  - Via di mezzo fra RISC e CISC.

4.0.0 Application Binary Interface (ABI)

Le modalità di uso dei registri non è spesso imposta dall'hardware. Per questa ragione esistono le *ABI*, ossia convenzioni software circa l'uso dei registri. Le *convenzioni di chiamata* ne sono un esempio.Ad esempio le convenzioni di chiamata specificatno

- Come / dove passare i parametri ad una subroutine/funzione
- Quali registri vanno preservati nel momento in cui si chiama una funzione
- Quali registri conterranno il valore di ritorno di una data funzione

4.0.0 Accesso alla memoria



La memoria di una macchina è indicizzata: ci si può riferire ad ogni zona della memoria ram specificando un numero. Questo numero viene detto *indirizzo*. La memoria è suddivisa in gruppi da 8 bit, ossia in gruppi di *byte*. L'indizirro incrementa di 1 per ongi byte

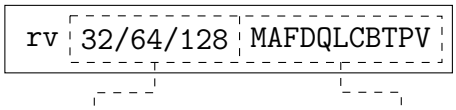
L'accesso alla memoria può avvenire secondo diverse modalità

- *Indirizzo assoluto*: l'indirizzo di memoria viene indicato all'interno dell'istruzione stessa. Questo può comportare problemi se nell'architettura le istruzioni hanno larghezza costante (ad esempio 32 bit), in quanto il numero dell'indirizzo può essere troppo grande
- *Indiretto*: l'indirizzo è contenuto all'interno di un registro indicato nell'istruzione stessa
- *Base + spiazzamento*: indirizzo è ottenuto sommando un valore costante (immediato) ad un indirizzo contenuto in un registro
- *Base + indice*: Simile al metodo precedente, tuttavia anziché usare un valore costante si utilizza un altro registro contenente il valore di spiazzamento
- Simile al metodo precedente, con la possibilità di aggiungere anche un valore costante

In alcune ISA è presente il cosidetto vincolo di allineamento, ossia è possibile accedere solo a indirizzi multipli di 8 o 4 byte. Ciò è sensato visto che i dati manipolati sono quasi sempre stringhe di 64 p 32 bit

5 Assembly Risc-V

L'ISA Risc-V ha la caratteristica di essere modulare. Ciò vuol dire che il produttore di ciascun chip può decidere di implementare o meno date funzionalità. Tali funzionalità vengono dette estensioni. Le estenzioni che un chip implementa sono specificate da una sigla, nel seguente formato:



Architettura a x bit Estensioni incluse

Le estensioni più comuni sono le seguenti:

- RV32I *Base Integer Instruction Set* implementa tutte le funzionalità di base. Ogni ISA Risc-V deve necessariamente implementare almeno questa estensione
- RV32M*Standard Extension for Integer Multiplication and Division* implementa funzioni quali `div` o `mul`
- RV32F *Standard Extension for Single-Precision Floating-Point* permette di lavorare su numeri con la virgola
- RV32G *General Purpose scalar Instruction Set* implementa tutte le seguenti estensioni: *IMAFD* i

5.1 Istruzioni base RV64I

5.1.0 Istruzioni aritmetiche

1. `add dest reg1 reg2`
 somma le variabili *reg1* e *reg2* salvandole in *dest*

2. `sub dest reg1 reg2` prova
 sottrae *reg2* a *reg1*, salvando in *dest*

5.1.0 Accesso alla memoria

3. `lw dest x(base)`
 carica la word situata in $base + x$ (x espresso in byte) in *dest*

4. `sw dest x(base)`
 salva la word situata in $base + x$ (x espresso in byte) in *dest*

5. `ld dest x(base)`
 carica la double word situata in $base + x$ (x espresso in byte) in *dest*

6. `sd dest x(base)`
 salva la double word situata in $base + x$ (x espresso in byte) in *dest*

5.1.0 Operatori logici

7. `and dest reg1 reg2`
 effettua l'*and* bit a bit fra *reg1* e *reg2*, salvando il risultato in *dest*

8. `or dest reg1 reg2`
 effettua l'*or* bit a bit fra *reg1* e *reg2*, salvando il risultato in *dest*

9. `xor dest reg1 reg2`
 effettua lo *xor* bit a bit fra *reg1* e *reg2*, salvando il risultato in *dest*

Lo *xor* restituisce 1 se entrambi i numeri sono diversi, altrimenti 0. Ha 2 use cases:

- Ottenere 0x000 effettuando lo *xor* con se stesso
- Applicare il negato, effettuando lo *xor* con tutti uni

5.1.0 Operazioni di shifting

Lo shift è importantissimo in quanto shiftare un numero a sinistra/destra corrisponde alla moltiplicazione/divisione per una potenza di 2

10. `slli dest reg1, ammount`
shifta di *ammount* posizioni sinistra il numero contenuno in *reg1* e lo salva in *dest*

Nota che se si applica lo shift a sinistra a determinati numeri si ottiene on overflow, ad esempio:

10000000	00000000	00000000	00000000	00000000	00000000	00000000	00000001

11. `srli dest reg1, ammount`
shifta di *ammount* posizioni a destra il numero contenuno in *reg1* e lo salva in *dest*

Allo stesso modo, anche uqesto shift può dare peroblemi di overflow, ad esempio:

11111111	11111111	00000000	00000000	00000000	00000000	00000000	00000001

eseguento uno shift a destra di 4 diventerebbe:

00001111	11111111	11110000	00000000	00000000	00000000	00000000	00000001

12. `srai dest reg1, ammount`
shifta di *ammount* posizioni a destra il numero contenuno in *reg1* e lo salva in *dest*. A differenza di *s*

11111111	11111111	00000000	00000000	00000000	00000000	00000000	00000001

eseguento uno shift a destra aritmetico di 4 diventerebbe:

11111111	11111111	11110000	00000000	00000000	00000000	00000000	00000001

5.1.0 Salti condizionati

13. `beq reg1, reg2, L1`
se il contenuto del registo *reg1* è uguale al registro *reg2* salta all'etichetta *l1*

14. `bne reg1, reg2, L1`
se il contenuto del registo *reg1* è diverso al registro *reg2* salta all'etichetta *l1*

15. `blt reg1, reg2, L1`
se il contenuto del registo *reg1* è minore al registro *reg2* salta all'etichetta *l1*

16. `bge reg1, reg2, L1`
se il contenuto del registo *reg1* è maggiore o uguale al registro *reg2* salta all'etichetta *l1*

17. `bltu reg1, reg2, L1`
se il contenuto del registo *reg1* è minore al registro *reg2* salta all'etichetta *l1*. I dati nei registri sono

18. `bgeu reg1, reg2, L1`
se il contenuto del registo *reg1* è maggiore o uguale al registro *reg2* salta all'etichetta *l1*. I dati nei re

5.2 Procedure e convenzioni di chiamata

5.2.0 Uso registri

Nel momento in cui si chiama una funzione, sia il *chiamante* che il *chiamato* sono soggetti alle cosiddette convenzioni di chiamata. Fanno parte di queste convenzioni anche regole per l'uso dei registri:

- `x10 - x17` vengono usati per il passaggio di parametri
- `x1` viene usato per contenere l'*indirizzo* di memoria della istruzione per ritornare al punto dal quale la funzione è stata chiamata

In particolare, i registri hanno le seguenti funzionalità:

Register	ABI Name	Description	Saver
`x0`	`zero`	Hard-wired zero	-
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x3`	`gp`	Global pointer	-
`x4`	`tp`	Thread pointer	-
`x5-7`	`t0-2`	Temporaries	Caller
`x8`	`s0/fp`	Saved register/frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-11`	`a0-1`	Function arguments/return values	Caller
`x12-17`	`a2-7`	Function arguments	Caller
`x18-27`	`s2-11`	Saved registers	Callee
`x28-31`	`t3-6`	Temporaries	Caller
`f0-7`	`ft0-7`	FP temporaries	Caller
`f8-9`	`fs0-1`	FP saved registers	Callee
`f 10-11`	`fa0-1`	FP arguments/return values	Caller
`f12-17`	`fa2-7`	FP arguments	Caller
`f 18-27`	`fs2-11`	FP saved registers	Callee
`f28-31`	`ft8-11`	FP temporaries	Caller

Tabella 2: Uso dei registri in Risc-V

- `x1 ra` viene usato per contenere l'*indirizzo* di memoria della istruzione per ritornare al punto dal quale la funzione è stata chiamata
- `x2 sp` viene usato per contenere *l'indirizzo* di memoria dell'ultimo elemento allocato sulla stack. Il prossimo indirizzo libero sarà `sp-4`
- `x3 gp` viene usato per contenere l'indirizzo della memoria statica di un programma (*static segment*)
- `x8 fp` viene usato per contenere il primo indirizzo di memoria dello *stack*

Le convenzioni di chiamata specificano anche un fatto importante:

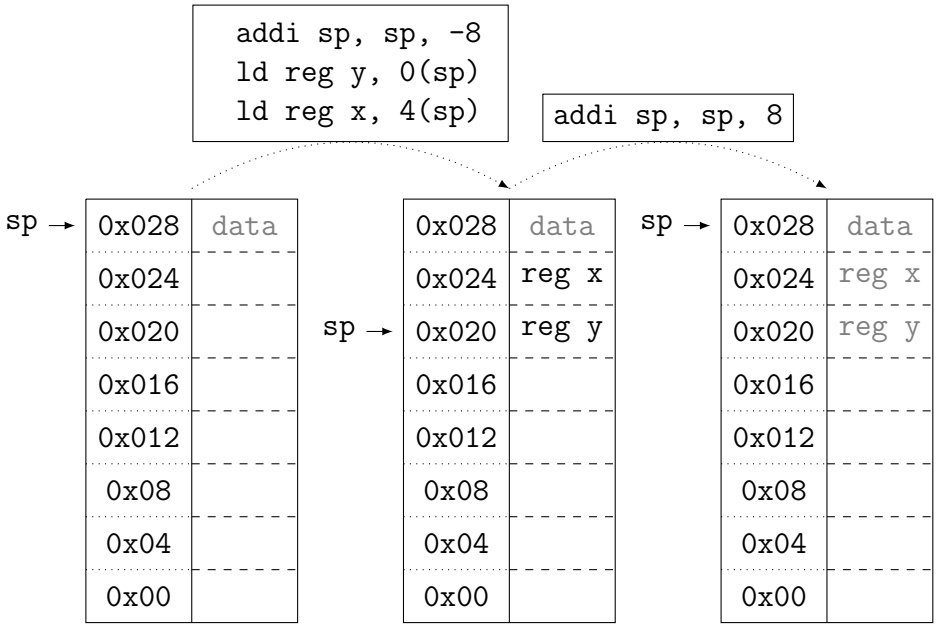
Per convenzione, alcuni registri possono essere sovrascritti durante l'esecuzione di una funzione, altri no

Ecco perchè nella tabella è indicato che i registri `s0-s11` vanno salvati dal *callee* (dalla funzione chiamata), facendo sì che al termine della procedura questi abbiano lo stesso valore che avevano prima della chiamata, mentre i registri `t0-6` possono essere usati senza essere salvati

5.2.0 Uso stack

Se i registri per i parametri non dovessero bastare, è possiblie utilizzare lo stasck, tenendo in mente che:

- Lo stack cresce "verso il basso". Lo stack pointer (registro `sp, x2`) si sposta a valori via via più piccoli man mano che allochiamo nuovi dati sullo stack
- Lo stack va svuotato alla fine di cisascuna procedura



5.3 Esempi codice

5.4 Esempio if

Il seguente if

```
if (i == j) f = g + h; else f = g - h;
```

verrebbe tradotto con il seguente codice assembly, assumendo che `x22=i`, `23=j`, `x19=g`, `x20=h`

```
      bne x22, x23, ELSE  # Salta a ELSE se x22 div. da x23
      add x19, x20, x21   # f = g + h
      beq x0, x0, ESCI    # Salto incondizionato a ESCI
ELSE: sub x19, x20, x21   # f = g - h
ESCI: ...
```

5.5 Esempio ciclo while

Il seguente ciclo

```
while (salva[i] == k)
  i += 1;
```

verrebbe tradotto con il seguente codice assembly, assumendo che `x22=i`, `x25=salva`, `x24=k`, `x20=h`

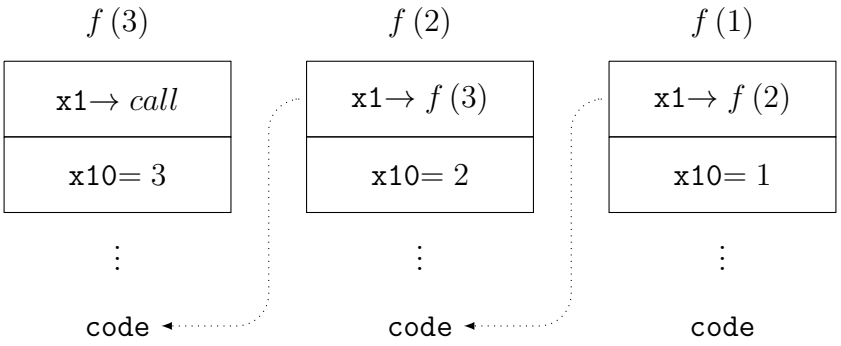
```
Ciclo: slli x10, x22, 3  # Registro temp. x10 = 8*i
       add x10, x10, x25 # Ind. di salva[i] in x10
       ld x9, 0(x10)     # Carica salva[i] in x9
       bne x9, x24, Esci # Esci se raggiunto limite
       addi x22, x22, 1  # i = i+1
       beq x0, x0, Ciclo
Esci: ...
```

5.6 Esempio funzione ricorsiva

Nelle funzioni ricorsive c'è il problema del fatto che i registri vengano sovrascritti. In particolare, va ricordato che i cosiddetti *saved registers* vanno salvati, così come i registri di `ra` (x1). Ciò vuol dire che tali registri vanno salvati nello stack dal chiamante

```
fact:
    addi sp, sp, -16  # Allochiamo spazio per due elementi
    sd x1, 8(sp)      # Salviamo x1
    sd x10, 0(sp)     # Salviamo x10

    addi x5, x10, -1  # Calcoliamo x5 = n-1
    bge x5, x0, L1    # Se n-1 >= 0, saltiamo a L1

    addi x10, x0, 1   # Carico 1 in x10
    addi sp, sp, 16   # Ripuliamo lo stack
    jalr x0, 0(x1)    # Ritorniamo al programma chiamante

L1:
    addi x10, x10, -1 # Decrementiamo n (arg. diventa: n-1)
    jal x1, fact      # Chiamiamo fact(n-1)

    addi x6, x10, 0   # x6 = fact(n-1)
    ld x10, 0(sp)     # Ripristino dell'argomento n, x10
    ld x1, 8(sp)      # Ripristino del reg. di ritorno, x1
    addi sp, sp, 16   # Ripuliamo lo stack

    mul x10, x10, x6  # Restituiamo n*fact(n-1)
    jalr x0, 0(x1)    # Ritorniamo al programma chiamante
```

Appena entrati nella funzione salviamo il valore di `x1 (ra)` e di `x10` (ossia il parametro di ingresso `n`)



5.7 Funzione foglia

Traduciamo la seguente funzione in assembly:

```
int64 esempio_foglia(int64 g, int64 h, int64 i, int64 j) {
    int64 f;
    f = (g+h) - (i+j);
    return f;
}
```

Tale funzione è detta funzione foglia, in quanto non chiama al suo interno nessun'altra funzione. Per questa ragione tutte le sue variabili possono evitare di essere allocate sullo stack, rendendola molto più veloce! Vediamo la traduzione, supponendo che i parametri si trovino in a0, a1, a2, a3

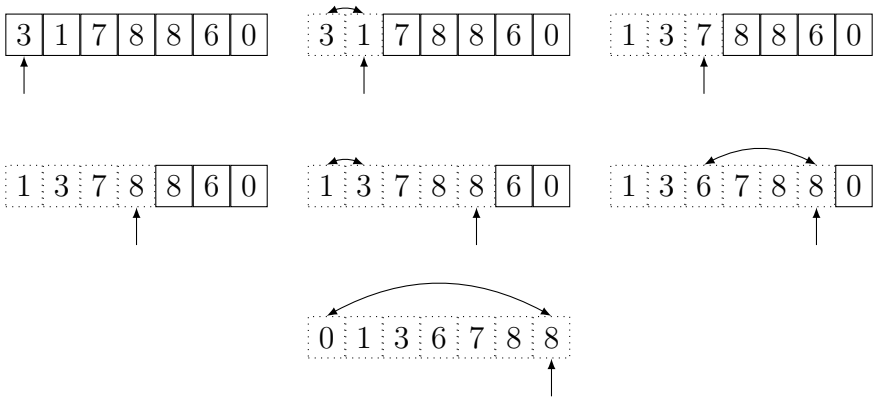
```
esempio_foglia:
    add a0, a0, a1   # g+h
    sub a0, a0, a2   # (g+h)-i
    sub a0, a0, a3   # ((g+h)-i)-j
    jalr x0, 0(ra)   # ritorno al chiamante
```

5.8 Insertion sort

```
//Funzione aux
void sposta(int v[], size_t i) {

  size_t j;
  int appoggio = v[i];
  j = i - 1;

  while ((j >= 0) && (v[j] > appoggio)) {
    v[j+1] = v[j];
    j = j-1;
  }

  v[j+1] = appoggio;
}

//Funzione sort
void ordina(int v[], size_t n) {
```

```
    size_t i;
    i = 1;
    while (i < n) {
      sposta(v, i);
      i = i+1;
    }
  }
```

L'algoritmo funziona scorrendo il vettore e costruendone quello ordinato inserendo di volta in volta l'*i-esimo* elemento mantenendo l'ordine della porzione precedente: Considera il seguente vettore



Le due funzioni, *sposta* e *ordina* fanno:

- *Sposta* prende l'elemento i-esimo e lo inserisce nel sotto vettore $[0, i-1]$ mantenendone l'ordine
- *Ordina* chiama la funzione *sposta* per ogni suo elemento

I registri sono utilizzati nel seguente modo:

- `a0` = `v`
- `a1` = `i` e in seguito `j`
- `a3` = `appoggio`
- `a0` = `v`
- `a0` = `v`
- `a0` = `v`

```
sposta:
    slli a4,a1,2        #a4 = i*4
    add a5,a0,a4        #a5 = &v[i]
    lw a3,0(a5)         #a3 = v[i]
    addiw a1,a1,-1      #a1 = a1-1 (i-1)

    blt a1, x0, .L2     # se j < 0 esci dal ciclo
```

```
    lw a4,-4(a5)        # a4 = v[i-1]=v[j]
    bge a3,a4,.L2       # se appoggio è >= v[j] esci
    li a2,-1            # carica -1 in a2
.L3:
    sw a4,0(a5)         # memorizza v[j] (a4) in v[j+1)
    addiw a1,a1,-1      # a1 = a1-1
    beq a1,a2,.L4       # salta se a1 = -1
    addi a5,a5,-4       # j=j-1
    lw a4,-4(a5)
    bgt a4,a3,.L3       # salta se v[j] > appoggio
    j .L2
.L4:
    li a1,-1
.L2:
    addi a1,a1,1
    slli a1,a1,2
    add a1,a0,a1
    sw a3,0(a1)
    ret
```

6 Assembly Intel x64

Le architetture Intel sono nate con intenti drasticamente diversi rispetto alle architetture ARM o Risc-V. Queste architetture sono basate sui seguenti principi:

- Garantire la retrocompatibilità
  - Eseguire codice vecchio
  - Lunghezza istruzioni variaible
  - Maggire complessità

Per via della sua complessità è impossibile analizzare l'intera architettura intel, ci concentreremo sulla modalità a 64 bit (la quale garantisce retrocompatibilità con 18, 32 bit). In particolare vedremo la sintassi utilizzata dall'assemblatore GNU

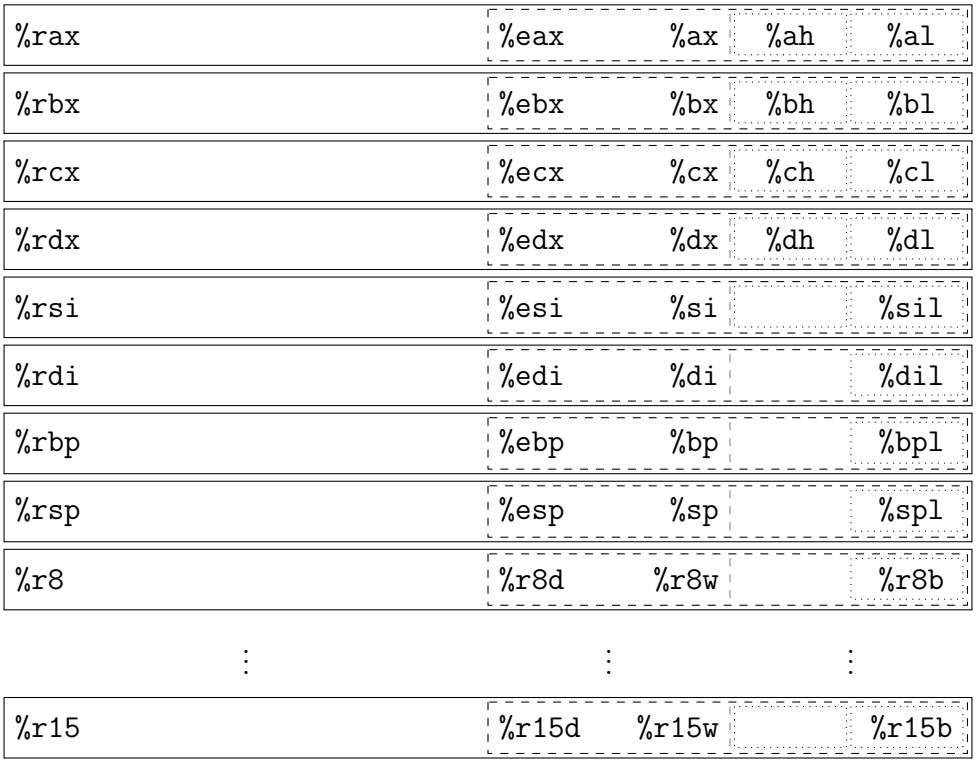
6.1 Registri

- 16 registri
- Nome preceduto da `%`
- `%rax %rbx %rcx %rdx %rsx %rdx %rbp %rsp`
- `%r8 ... %r15`
- Registri specializzati:
  - `%rsp` → *stack pointer*
  - `%rbp` → *base pointer*

- `%rip` → *instruction pointer*

- Registri flag `%rflags`: estende `%eflags` che estende `%flags`, ogni bit corrisponde ad una flag
  - `cf` *Carry flag*
  - `zf` *Zero flag*
  - `sign` *Sign flag*
  - `of` *2's overflow flag*

In più i registri delle architetture intel sono divisi in sotto-registri a 32 e 16 bit come segue:



6.1.0 Convenzioni di chiamata

Ricordiamo che queste non fanno parte dell'ISA ma dell'ABI

- Primi 6 argomenti in `%rdi %rsi %rdx %rcx %r8 %r9`
- Valori di ritorno in `%rax %rdx`

Register	Description	Saver
`%rax`	Return value	Caller
`%rbx`	Saved register	Callee

`%rcx`
`%rdx`
`%rsi`
`%rdi`
	Function arguments	Caller

`%rbp` `%rsp`	Base pointer Stack pointer	Caller
`%r8` `%r9`	Function arguments	Caller
`%r10` `%r11`	Temporaries	Caller

`%r12`
`%r13`
`%r14`
`%r15`
	Saved registers	Callee

Tabella 3: Uso dei registri in Intel x64

6.1.0 Struttura istruzioni

Tutti i registri sono preceduti da `%`, mentre gli immediati da `$`

- Tendenzialmente hanno due operandi
  - Primo operando *sorgente*
  - Secondo operando *destinazione implicita*

Destinazione e sorgente possono essere registri, memoria o immediati. Tuttavia vige la limitazione che non possono essere entrambi indirizzi di memoria

6.1.0 Modalità di indirizzamento

Sono molto più complesse rispetto al Risc-V. Hanno la seguente forma:

```
<displacement> (<base reg>, <index reg>, <scale>)
```

- *Valore immediato* in byte, di quanto va shiftato l'indirizzo (come Risc-V)
- Registro contenente l'indirizzo di memoria a cui accedere
- Indice si usa solo se si indica anche scala e viceversa. Shiftano l'indirizzo di `<index reg> * <scale>`

7 Assembly ARM

- 16 registri quasi tutti registri generl purpuse numerati da `r0 ... r15` con nome simbolito (ad esempio `r13 = sp` o `r14 = lr`, ossia *stack pointer e link register*)
- Utilizza flags per istruzioni condizionali
- Ogni istruzione può essere resa condizionale, postponendo determinate sillabe
- Gli immediati vengono indicati con `#`

Register	Description	Saver

`r0`
`r1`
`r2`
`r3`
	Return values
	Function parameters
		Caller

`r4-r11`	Saved registers	Callee
`r12`	Temporary register	Caller
`r13`	Stack pointer	Callee
`r14`	Return address	Callee
`r15`	Program counter/flags	/

Tabella 4: Uso dei registri in ARM

I flag sono contenuti nei bit più significativi del registro `r15` in una sotto porzione chiamata `apsr`. Questi sono:

- `z` se è 0
- `n` se è negativo
- `c` se è avvenuto un overflow fra dati unsigned
- `v` se è avvenuto un overflow fra dati signed

In ARM tutte le istruzioni possono essere rese condizionali postponendo le seguenti sillabe dopo l'istruzione

Tramite questi bit possiamo veribicare le seguenti condizioni:

- `eq` *equal*
- `ne` *not equal*
- `hs` *higher or same*
- `lo` *lower*
- `mi` *minus*
- `pl` *plus*

- `vs` *overflow*
- `vc` *overflow clear*
- `hi` *higher*
- `ls` *lower or same*
- `ge` *greater or equal*
- `it` *less than*
- `gt` *greater than*
- `le` *less or equal*

7.1 Modalità di indirizzamento

La sintatti per indicare un indice è della forma seguente:

```
i = <base> + {<offset>  <indice (shiftato)>}
```

7.1.0 Pre e post indexing

In ARM, il valore del registro di base può essere aggiornato direttamente nell'istruzione di indirizzamento.

- *Pre-indexing*:
  - Senza writeback: Accedo all'indirizzo e lascio `<base>` invariato
  - Con writeback: salvo indirizzo calcolato in `<base>`
- *Post-indexing*:
  - Solo con writeback: accedo all'idirizzo indicato in `<base>` e ci accedo. Poi aggiorno base con il valore indicato

Ad esempio:

- Pre index - no writeback : `ld r0, [r1, #4]` → `r0 = [r1 + 4]`, `r1` invariato
- Pre index - writeback: `ld r0, [r1, #4]!` → `r0 = mem[r1+4]`, `r1 = r1 + 4`
- Post index - writeback: `ld r0, [r1], #4` → `r0 = mem[r1]`, `r1 = r1 + 4`

Inoltre posso usare indice shiftato come segue

`ldr r0, [r1, r2, lsl #4]` ossia `r0 = mem[r1 + r2 << 4]`

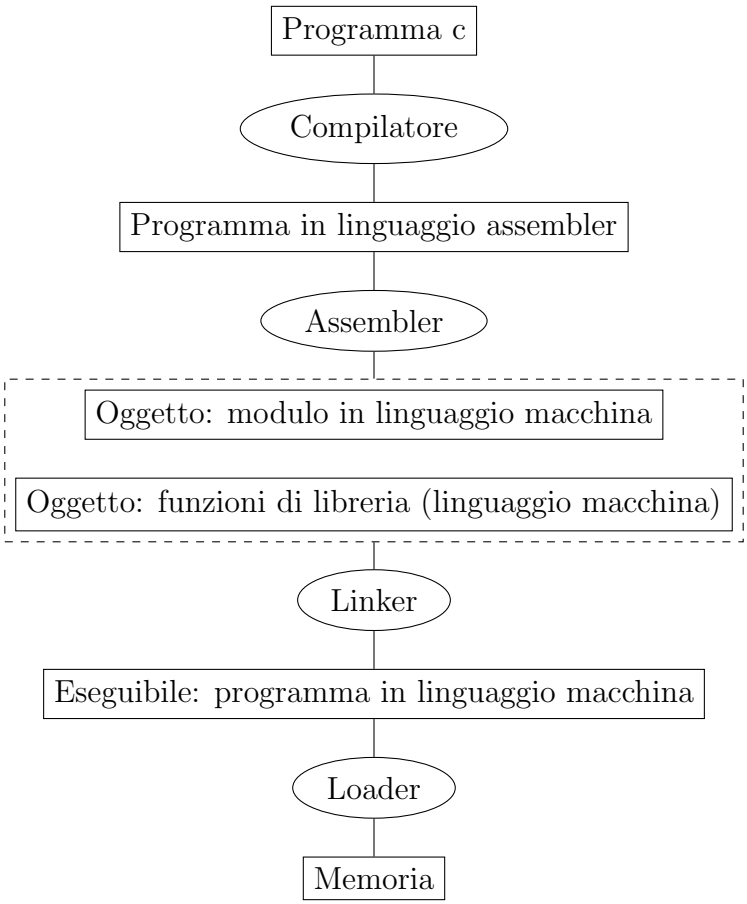
7.2 Letture e scritture multiple

8 Toolchain

Abbiamo visto che un processore è in grado di "leggere" sequenze di bit, chiamate istruzioni. Tuttavia per noi è troppo complicato andare a scrivere un programma avanzato in Assembly (il quale costituisce un insieme di codici mnemonici tramite i quali vengono convertiti direttamente in stringe binarie). Per questa ragione il codice assembly viene generato a partire da un linguaggio ad alto livello, ad esempio `c` o `c++`

Una toolchain fornisce gli strumenti adeguati al fine di poter passare da un linguaggio di alto livello a linguaggio macchina

:



Possiamo, ad esempio attraverso il *gnu compiler collection* o `gcc`, far si che il processo di compilazione si fermi prima, tramite i seguenti comandi:

19. `gcc -S`
ferma la compilazione dopo aver generato il file assembly

20. `gcc -c`
ferma la compilazione dopo aver generato il file oggetto

Nota che il gcc è formato da 4 sottoprogrammi:

- `cpp` è il preprocessor, risolve le macro e gli import

- `cc` traduce il programma in codice assembly
- `as` assemblatore, traduce l'assembly in linguaggio macchina
- `ld` esegue il linking, ossia aggiunge le librerie

8.0.0 Assemblatore

L'assemblatore spesso fa più che tradurre con corrispondenza diretta le istruzioni assembly in istruzioni macchina, ad esempio:

- Converte pseudo istruizioni
  - `j LABEL` diventa `jal x0, LABEL`
  - `mv x10, x11` diventa `addi x10, x11, 0`
- Converte numeri da esadecimale/decimale a binario
- Converte le label in indirizzi di memoria contenenti l'istruzione
- Gestrisce salti troppo ampi (nell'istruzione jump abbiamo un numero limitato di byte per indicare lo spiazzamento)
- Genera metadati

8.0.0 File oggetto

I file oggetto sono i file che verranno poi linkati. Contengono le seguenti informazioni, divise in segmenti:

- *Header*: specifica la dimensione e la posizione degli altri segmenti
- *Segmenti*
  - *Segmento di testo*: contiene il codice in linguaggio macchina
  - *Segmento di dati*: contiene tutti i dati allocati durante la durata del programma
- *Tabella dei simboli*:
  - Associa i simboli (varibili, funzioni) ad indirizzi relativi
  - Enumera simboli non definiti (sono definiti altrove)
- *Tabella di rilocazione*
  - Se, ad esempio, una jump punta ad un'indirizzo assoluto e non relativo all'indirizzo dell'istruzione corrente, questo va a finire nella suddetta tabella, in quando deve essere patchato

Il linker si occupa quindi principalmente di mettere in ordine e concatenare opportunamente questi file oggetto, eliminando dunque le symbol e relocation tables

8.0.0 Librerie

Una libreria non è altro che una collezione di file oggetto. Ne esistono di due tipi:

- *Statiche* con estensione `.a`. Linkate da `ld`
  - Migliore velocità di esecuzione
  - Maggiore peso del programma
  - Programma autocontenuto in un solo eseguibile
  - Più semplici
- *Dinamiche* con estensione `.so`. Linkate a runtime
  - Meno efficienti
  - Programma più leggero in termini di byte
  - Programma le necessita a runtime (eseguibile non autoconenuto)
  - Più complicate da implementare

Inoltre con le librerie dinamiche, è possibile anche seguire il cosiddetto lazy loading:

- Solitamente le librerie vengono caricate dal *sistema operativo* all'avvio del programma
- Per ridurre questi tempi è possibile rimandare il caricamento al momento in cui una libreria viene utilizzata
- Quando si chiama una funzione di libreria per la prima volta, viene chiamato uno stub, che esegue il suo caricamento durante runtime
- La seconda volta che la funzione viene chiamata, questa sarà gia linkata e quindi verrà usata direttamente

9 Il processore

Affrontiamo ora come funziona un processore, facendo riferimento all'ISA Risc-V.

9.0.0 Primi passi esecuzioe

Nell'esecuzione delle istruzioni, sono sempre presenti due passi:

- *Fetch* ossia il prelievo dell'istruzione:
  - Vado a indirizzo puntato da *program counter*
  - Leggo contenuto (32 bit nel caso Risc-V)
- *Lettura registri.* Vengono letti i registri coinvolti nell'istruzione

ciò che avviene dopo dipende da architettura a architettura

9.0.0 ALU

Tutti i tipi di istruzioni utilizzano la *alu* (Unità Logico Aritmetica). Tutte le istruzioni la utilizzano:

- *Accessi a memoria* (calcolo indirizzo)
- *Istruzioni aritmetico/ logiche*
- *Salti condizionati* (per verificare se sono uguali due registri)

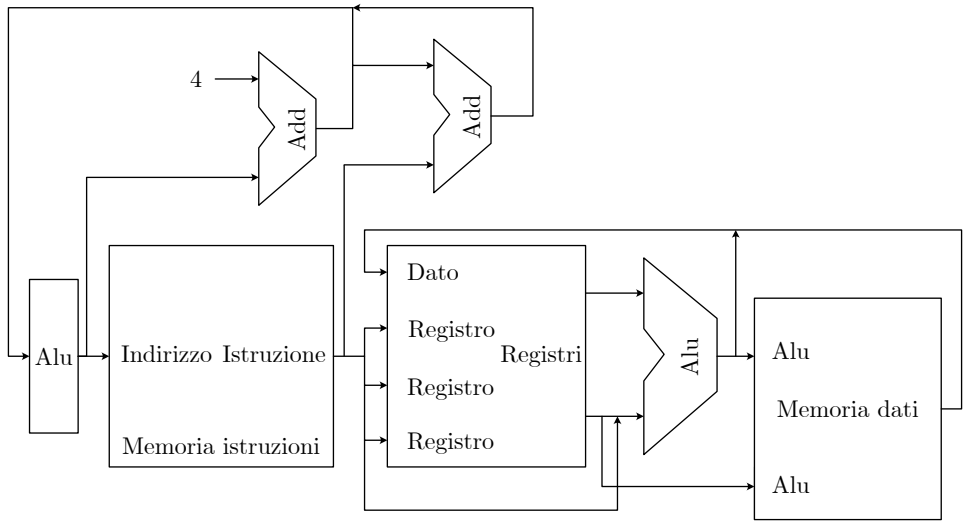


Figura 5: Datapath 1

Tuttavia, questa prima immagine risulta molto approssimativa, in quanto è necessario coordinare tramite dei mux quale dato sceglere laddove si presentino diramazioni. Ciò è fatto grazie ai nuovi elementi introdotti in tratteggiato:

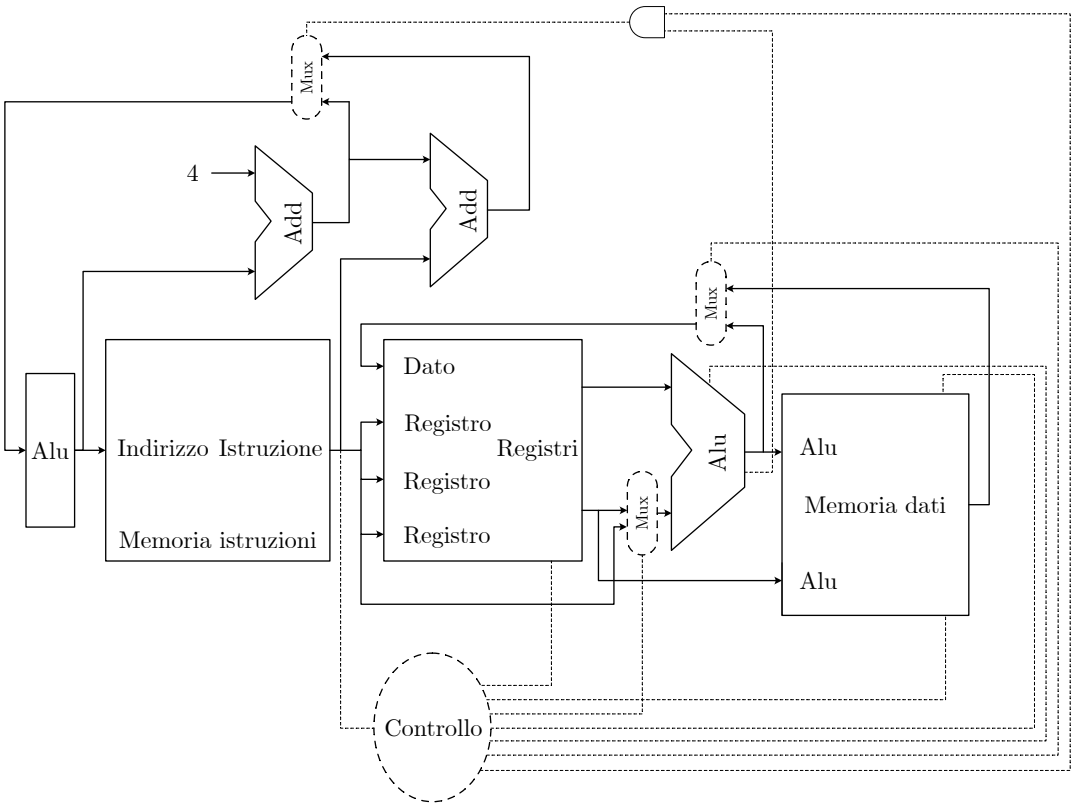


Figura 6: Datapath 2, con aggiunta di elementi di controllo

Nota che quasi tutti i segnali in uscita sono a 1 bit, ad eccezione del segnale che entra all'interno dell'ALU. Quel messagio codifica l'operazione che deve eseguire la alu

10 Memoria

10.1 La cache

Nella creazione di dispositivi capaci di immagazzinare dati, c'è sempre un trade off fra efficienza nell'accesso e dimensione dell'archivio. Per questo si usano diverse gerarchie di memoria, per ottenere il compromesso ideale

Immagina di avere immagazzinato dei dati un un archivio. Ogni volta che devo estrarre un dato devo alzarmi dalla scrivania e prendere il fascicolo corrispondente. Il 90% di tempo verrà utilizzato proprio per andare dalla scrivania all'archivio. In alternativa posso portare tutti tutti i fascicoli sulla scrivania risparmiando tempo. Le gerarchie di memoria sono le seguenti:

- *Registri* - velocissimi ma capacità ridottissima
- *Cache* - intermedio fra memoria principale e registri
- *Memoria principale* - lenta ma molto capiente

Tecnologia	Tempo di accesso tipico	$ per GB (2010)	$ per GB (2012)
SRAM	$0.5 - 2.5$ ns	$\$2000 - \$5000$	$\$500 - \$1000$
DRAM	$50 - 70$ ns	$\$20 - \$75$	$\$10 - \$20$
Memoria flash	$70 - 150$ ns	$\$4 - \$12$	$\$0.75 - \$1$
Dischi magnetici	$5000000 - 20000000$ ns	$\$0.2 - \$2$	$\$0.05 - \$0.1$

10.1.0 Struttura gerarchia della memoria

La memoria, per le ragioni elencate prima, è strutturata ponendo:

- Memoria a bassa capacità e alta velocità *vicino* al processore
- Memoria ad alta capacità e bassa velocità *lontano* al processore

Per questa ragione è necessario implementare meccanismi che controllino a quale livello della gerarchia è disponibile un determinato dato

10.1.0 Terminologia

- Blocco unità minima di informazione che può essere presente o assente in ciascun livello
- Hit rate frequenza con cui trovo il dato nel livello superiore
- Miss rate complementare dell'hit rate: $\rightarrow$ 1 - hit rate
- Tempo di hit quanto tempo serve per accedere ad un dato se è presente al livello corrente
- Penalità di miss quanto tempo devo sprecare per accedere al dato se devo andare al livello più distante dal processore

Il tempo di hit è di molto maggiore rispetto alla penalità di miss, per questa ragione è vantaggioso implementare gerarchie di memoria

10.2 Tipo di cache

10.2.0 Cache a mappatura diretta

Come posso verificare se un dato è presente o meno in cache? Questo meccanismo di mappatura è implementato diversamente in diversti tipi di cache

- *Cache a mappatura diretta*. Ogni indirizzo della memoria principale è mappato ad un indirizzo della memoria cache tramite il modulo:

$$\text{Indirizzo cache} = \text{indirizzo principale } \% \text{ dimensione cache}$$

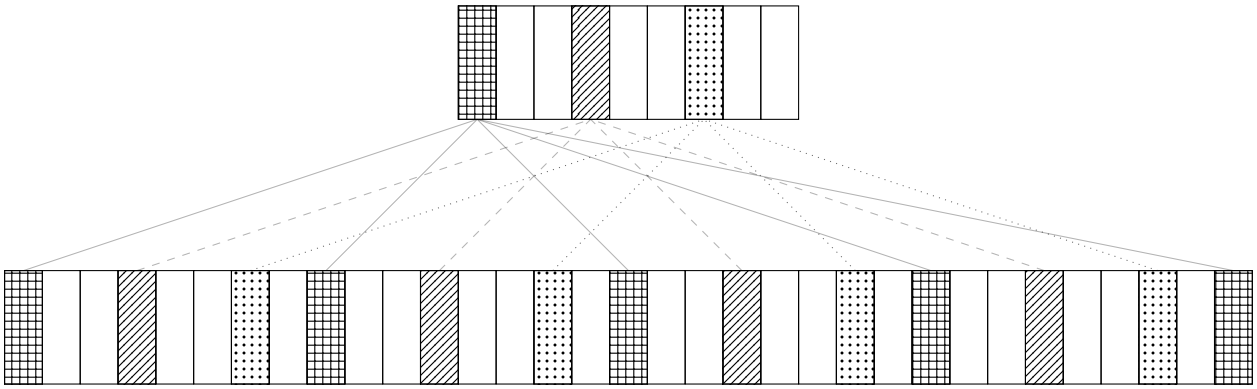
10.2.0 Gestione delle miss

Nel momento in cui non abbiamo miss, la presenza della cache non altera di molto il funzionamento di un processore. Se c'è una miss, tuttavia, dovra succedere una cosa del tipo:

- Inviare il valore *program counter* - 4 alla memoria (incremento viene eseguito all'inizio)
- Comandare la memoria di eseguire una lettura e attenderne il completamento
- Scrivere il blocco che proviene dalla memoria della cache aggiornando il tag
- Far ripartire l'istruzione dal fetch, che stavolta troverà l'istruzione in cache

10.2.0 Tag

Il tag è una parte di memoria cache che serve ad indicare quale indirizzo di memoria è salvato all'interno di un determinato blocco di cache



Ad esempio prendiamo il primo blocco di cache a sinistra: in questo caso tutti il tag ci dirà quale dei blocchi corrispondendi nella memoria principale corrispondono a quest'ultimo. In particolare supponiamo di avere:

- Indirizzo su 64 bit
- Dimensione della cache $= 2^n$
- Dimensione del blocco di cache $= 2^m$ words $= 2^{m+2}$ bytes

allora:

$$\text{dimensione tag } = 64 - n - m - 2$$

10.2.0 Cache completamente associativa

Dato un indirizzo di memoria, anzichè avere una mappatura 1:1 con la sua posizione nella cache, devo cercare all'interno di tutta la cache e verificare se è presente il dato che ricerco.

- Vantaggi:

- Ogni dato può essere salvato ovunque nella cache. Se in una cache a mappatura diretta ho bisogno di un dato che si mappa nella stessa zona della cache spesso, allora avrà molti miss. Questo problema non si presenta con la cache associativa

- Svantaggi:
  - Costosissima da realizzare
  - Il tag è sempre l'intero indirizzo di memoria
  - Ho bisongo di n comparatori per ricercare efficacemente l'indirizzo all'interno della cache

10.2.0 Cache set-associativa

Posso mischiare le tecnologie di cache a mappatura diretta ed associativa per ottenere la cache set-associativa. In particolare in questa cache avrò che

- Un indirizzo in memoria è mappato ad un insieme di blocchi detto linea *cache a mappatura diretta*
- All'interno della linea, la ricerca del blocco avviene tramire n comparatori *cache puramente associativa*

Linea	Tag	Dato	Tag	Dato
0				
1				
2				
3				

(a) Set associative cache a 2 vie

Linea	Tag	Dato
0		
1		
2		
3		
4		
5		
6		
7		

(b) Set associative cache a 1 via (*a mappatura diretta*)

Linea	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato
0								
1								

(c) Set associative cache a 4 vie

Linea	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato
0												

(d) Set associative cache a 8 vie (*completamente associativa*)

Figura 7: Configurazioni diverse di cache

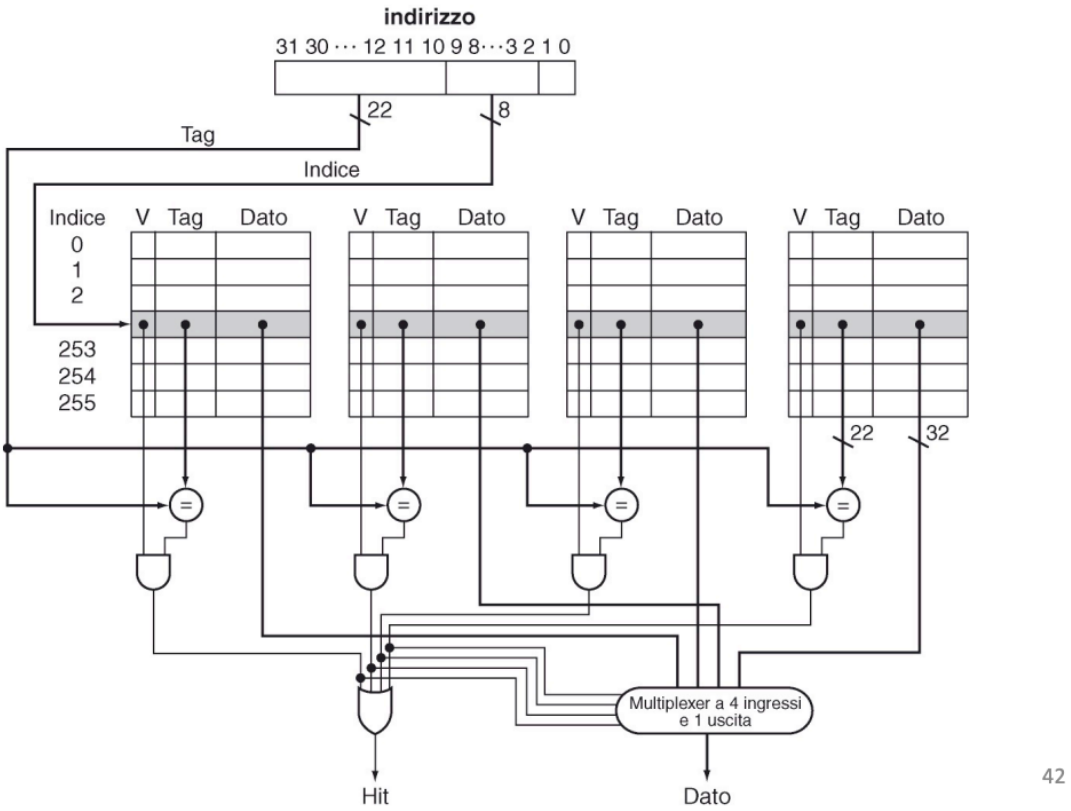


Figura 8: Esempio implementazione cache a 4 vie

Nota che nella cache a mappatura diretta se avviene un miss poi nella cache so in quale indirizzo caricare il nuovo dato. Ciò non è vero per una cache associativa

Esistono più policy per risolvere questo problema:

- Policy FIFO
- Policy Least Recently Used (usa una sequenza di bit per memorizzare l'ultimo accesso)

11 Input-output

Un calcolatore sarebbe inutile se non ci fosse un modo per interfacciavisi. Ciò viene reso possibile da dispositivi *input-output*. I dispositivi sono collegati al processore tramite un *dispositivo di comunicazione* chiamato bus:

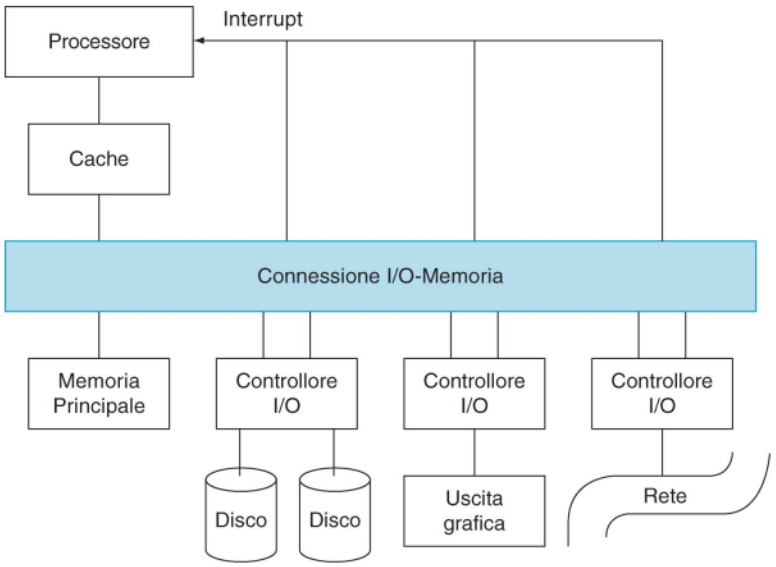


Figura 9: Esempio semplicifato del bus

11.0.0 Caratterizzazione

I dispositivi io sono caratterizzati da:

- *Comportamento*: possono effettuare operazioni di lettura o scrittura (R/W)
- *Partner*: può essere uomo o macchina
- *Velocità di trasferimento*

Dispositivo	Comportamento	Partner	Frequenza dati (Mbit/s
Tastiera	Input (ingresso)	Uomo	0,0001
Mouse	Input (ingresso)	Uomo	0,0038
Input vocale	Input (ingresso)	Uomo	0,2640
Input audio	Input (ingresso)	Macchina	3,0000
Scanner	Input (ingresso)	Uomo	3,2000
Output vocale	Output (uscita)	Uomo	0,2640
Output audio	Output (uscita)	Uomo	8,0000
Stampante laser	Output (uscita)	Uomo	3,2000
Display grafico	Output (uscita)	Uomo	$800,0000 - 8000,0000$
Modem via cavo	Input o output	Macchina	$0,1280 - 6,0000$
Rete/LAN	Input o output	Macchina	$100,000 - 10000,0000$
Rete/LAN wireless	Input o output	Macchina	$11,0000 - 54,0000$
Disco ottico	Memoria	Macchina	$80,0000 - 220,0000$
Nastro magnetico	Memoria	Macchina	$5,0000 - 120,0000$
Memoria flash	Memoria	Macchina	$32,0000 - 200,0000$
Disco magnetico	Memoria	Macchina	$800,0000 - 3000,0000$

11.1 Bus

Un bus, come anticipato, può essere inteso come un "cavo" che collega due elementi di un calcolatore (processore - memoria, periferiche - processore). I bus possono essere

- *Bus processore/memorie*
  - specializzati, corti e veloci
- *Bus I/O*
  - Possono essere lunghi e collegano periferiche *eterogenee*
  - Tipicamente non collegati alla memoria in maniera diretta, passano per il processore e ci arrivano tramite un altro bus

11.1.0 Bus sincroni

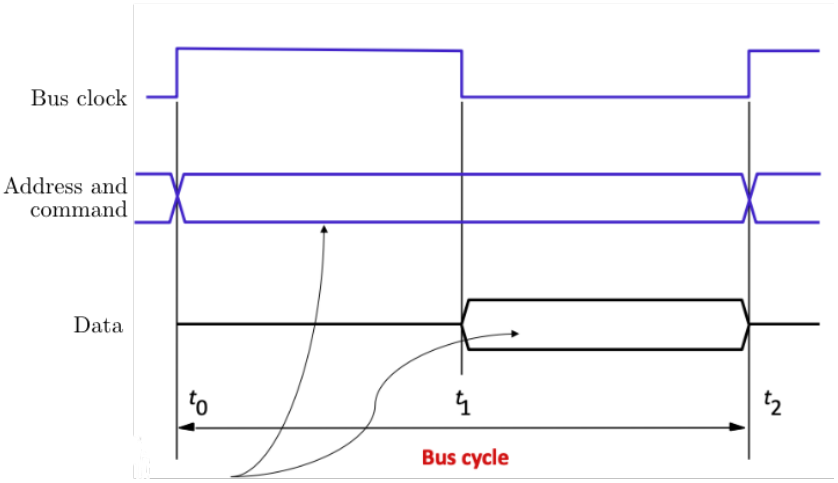


Figura 10: Bus sincrono

Nota come su ogni fronte basso il segnale viene spedito e rimane sul bus fino al prossimo fronte alto

- Pro
  - Molto semplice da implementare
  - Molto veloce
- Contro
  - Poca robustezza alla variazione del clock (*drift*)
  - Tutte le periferiche sono vincolate alla velocità di clock

11.1.0 Bus asincroni

Per ovviare ai problemi dei bus sincroni, spesso si ricorre all'uso dei bus asincroni

- Non esiste un clock
- Le transazioni sono regolate da segnali handshake

- Vengono introdotte linee di controllo per inizio e fine di transazioni

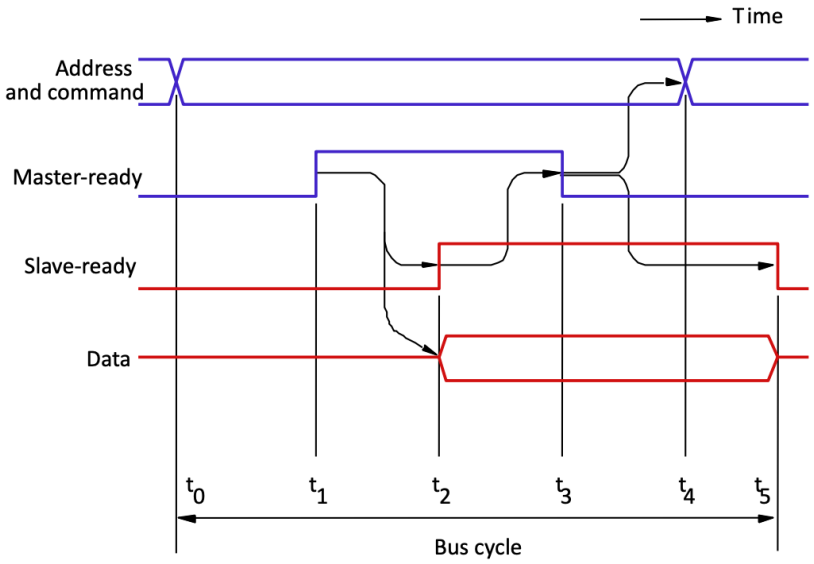


Figura 11: Bus sincrono

- Il master (spesso il processore) manda un segnale allo slave (spesso memoria)
- Quando lo slave si setta a 1, il segnale viene retropropagato al master, che viene settato a 0
- Il dato rimane nel bus fino a quanto lo slave è settato a 1

Naturalmente, anche questa tecnologia prevede dei pro e dei contro

- Pro
  - Robusto rispetto al *drift*
  - Comunica con periferiche di tipo diverso
- Contro
  - Lento nelle interaioni
  - Cirtuiteria molto più complessa e costrosa

Per tale ragione spesso si usano tecnologie ibride, anche se prevalentemente asincrone

Caratteristica	Firewire (1394)	USB 2.0	PCI Express
Utilizzo previsto	Esterno	Esterno	Interno
Numero dispositiviti per canale 63	127	1	1
Larghezza base dei dati	4	2	2 per linea
Larghezza di banda picco teorica	50- 100 mb/s	0.2-1.5-60 mb/s	250 per linea
Collegamento a caldo	Si	Si	Dipende dalle dimensioni
Lunghezza massima del bus (piste in rame)	4.5 metri	5 metri	0.5 metri
Nome dello standard	IEEE 1394	Forum degli implementatori usb	SIG PCI

Caratteristica	Serial ATA	Seria Attached SCSI
Utilizzo previsto	Interno	Esterno
Numero disposiviti per canale 63	1	4
Larghezza base dei dati	4	4
Larghezza di banda picco teorica	300 mb/s	300 mb/s
Collegamento a caldo	Si	Si
Lunghezza massima del bus (piste in rame)	1 metro	8 metri
Nome dello standard	SATA-IO	Comitato T10

11.1.0 Esempio calcolatore a 32 bit

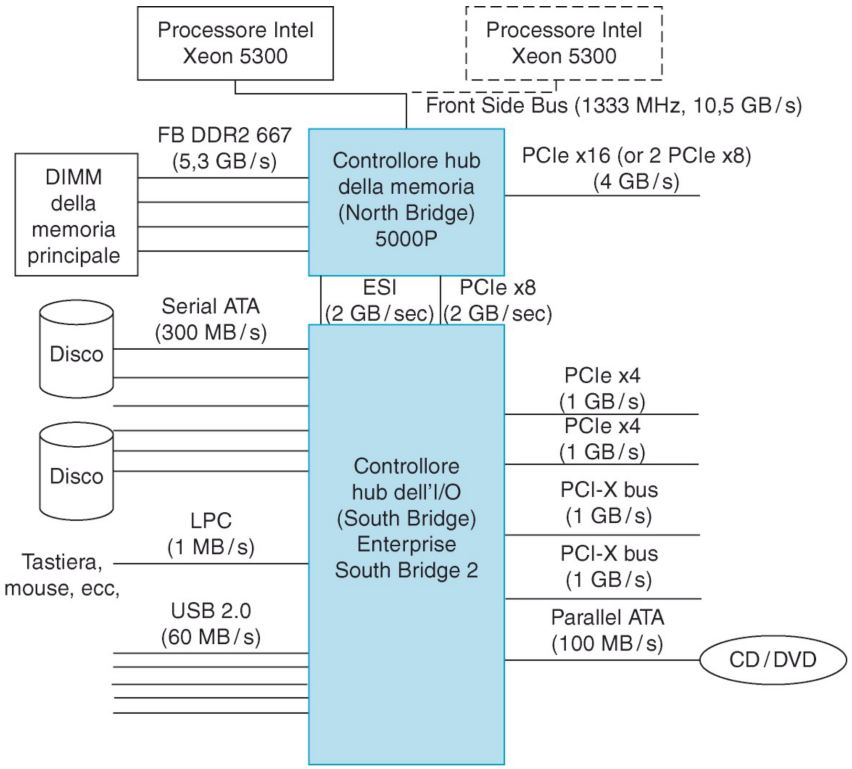


Figura 12: Esempio x86

11.2 Ruolo sistema operativo

Il sistema operativo gestisce tutte le rischieste di input e output, in modo da garantire la collaborazione efficace da parte di tutte le prefiferiche. In particolare solo una parte del S.O. ossia il kernel, interagiste in modalità protetta (modalità supervisor) con il processore per gestire l'interazione fra le periferiche. Questa modalità si chiama modalità ad interrupt (*la periferica* manda un segnale che viene gestito dal kernel). Questa modalità serve per gestire i problemi di concorrenza (ad esempio quando due programmi vogliono interagine con la stessa periferica)

il sistema operativo deve dunque implementare le seguenti funzionalità:

1. Possibilità di inviare comandi alle *periferiche*
2. *Le periferiche* devono poter notificare la corretta esecuzione di un'operazione, lanciando "eccezioni" nel caso vi siano errori
3. Consentire trasferimenti diretti da dispositivo e memoria

11.2.0 Memory mapped I/O

La comunicazione alle periferiche può avvenire tramite la scrittura in determinate zone di memoria come segue:

- Ogni dispositivo possiede area di memoria specifica (non accessibile al programmatore ma solo al S/O)
- Scrivendo su questa zona il *controllore I/O* intercetta il segnale ed esegue il comando

Ad esempio, posso stampare su terminale tramite una system call e a stampa finita un preciso bit verrà settato ad 1 per indicare che la stampa è avvenuta correttamente

Per accertarmi che il comando sia avvenuta correttamente posso fare polling, ossia continuo a "domandare" alla periferica se l'operazione è stata conclusa correttamente

Il problema del polling è che il polling viene fatto sempre, anche se l'operazione di I/O non viene eseguita

11.3 Interrupt

Nel momento in cui l'attesa attiva(polling) risulta inaccettabile, (ad esempio nel caso di un disco rigido) si può utilizzare la modalità a interrupt

11.3.0 Le eccezioni

In realtà un interrupt è un caso particolare di un evento del s.o.: le eccezioni

Vi sono due tipi di eccezioni:

- *Esterne*: ad esempio gli interrupt
- *Interne*: create dal programma stesso, ad esempio *segmentation fault*

Le eccezioni avvengono durante l'esecuzione di un programma, e vengono gestite dal system exception handler

- Interrupts
  - Causate da eventi esterni (I/O)
  - Asincrone
  - Sospendono il programma e ripendono da dove era stato interrotto
- Traps (eccezioni)
  - Causate da eventi *interni* al programma
    - Condizioni eccezionali (*arithmetic overflow, undefined instr*)
    - Errori (*hardware malfunction, segmentation fault*)
    - Fault (*non-resident page*)
  - Sincrone
  - Gestite da trap halndler
  - Possono causare l'aborto del programma
- Environment call/break

– All'interno del programma chiediamo esplicitamente di eseguire chiamate a sitema
– Invocate ocn istruzione "ecall", ad esempio la stampa di un carattere
– Invocate tramite, ad esempio, l'aggiunta di un breakpoint tramite debugger (istruzione *ebreak*)

11.3.0 Gestione istruzioni