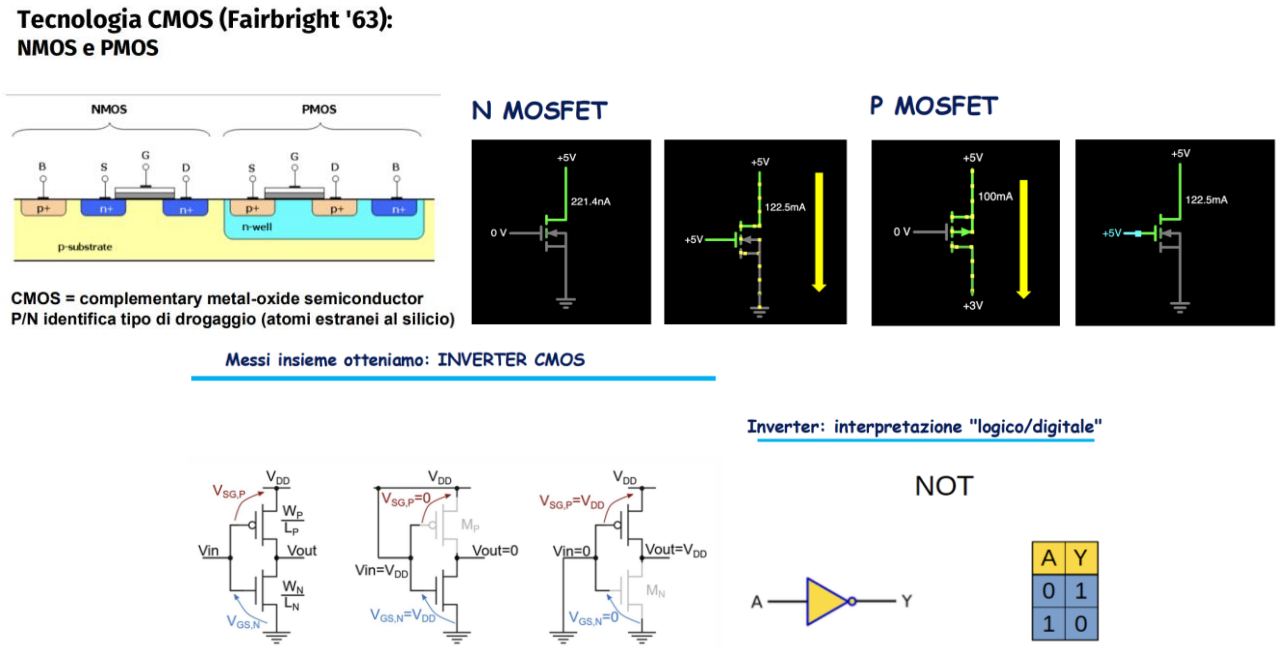
Architettura dei Calcolatori

Introduzione

Il termine architettura è usato per descrivere l'insieme degli attributi di un sistema, la sua struttura concettuale e il suo funzionamento. Con Architettura dei Calcolatori, si intende la scienza e l'arte di selezionare e collegare componenti hardware per creare calcolatori che soddisfano determinati requisiti (funzionali, di prestazioni e di costo).
Un'architettura non definisce l'implementazione dell'hardware, ma descrive le sue funzionalità.
Il calcolatore è una risorsa unica e in grado di eseguire una miriade di compiti eterogeni tra loro, in quanto esegue algoritmi scritti in un opportuno linguaggio di programmazione, i programmi.

Dai Chip alle Architetture dei Calcolatori

Un chip è composto da un wafer di materiale semiconduttore (es: silicio) con circuiti elettromagnetici integrati. Contiene milioni di componenti elettronici microscopici, i transistor, che trasmettono segnali di dati.
Il transistor è composto da tre strati, di materiale differente. Alla base troviamo un substrato in silicio (semiconduttore), mentre i terminali detti 'source' (sorgente) e 'drain' (collettore) sono di silicio drogato, ovvero con aggiunta di impurità. Il source e il drain sono sormontati da uno strato isolante di ossido, che si oppone al passaggio di corrente verso lo strato superiore, detto 'gate'. Il canale nel semiconduttore tra source e drain funziona come un condensatore, consente di regolare il passaggio di corrente all'interno di un circuito.
Con una tensione di gate di 0 volt, la resistenza del canale è elevata e il MOFSET si comporta come un interruttore aperto. Se la tensione è superiore ad un determinato livello, tensione di soglia, il canale si troverà in una condizione di bassa resistenza e lascerà passare la corrente, comportandosi come un circuito chiuso.



A	Y
0	1
1	0

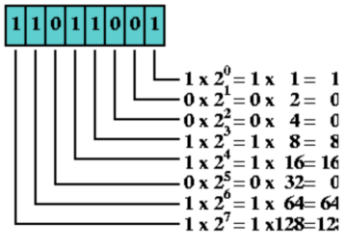
Legge di Moore: 'La complessità di un microciruito, misurata ad esempio tramite il numero di transistor per chip, raddoppia ogni 18 mesi (e quadruplica quindi ogni 3 anni).'

Oggi, i singoli transistor dentro ai circuiti integrati in silicio hanno dimensioni nei nanometri, dimensioni sotto la quale i fenomeni fisici passano da un comportamento classico lineare ad uno quantistico e probabilistico. Tuttavia, si può scalare anche su altre dimensioni, ad esempio usando diversi processori in parallelo (multicore). Questo comporta la necessità di usarli in modo opportuno.
In un processore multi-core tradizionale, ogni core è uguale agli altri. Quando la CPU è inattiva o esegue attività semplici, si manifesta sempre un consumo energetico minimo al di sotto del quale non è possibile scendere senza spegnere completamente la CPU. Ci sono quindi core prestazionali (core P) e core a basso consumo energetico (core E): su portatili e in generale su dispositivi che lavorano con una batteria, a seconda del carico di lavoro istantaneo, si può estendere l'autonomia del dispositivo.

Classificazione dei circuiti

Circuiti integrati analogici: I circuiti analogici trasmettono segnali continui e variabili lungo un periodo di tempo prestabilito. Il segnale di uscita è una funzione lineare dell'input con voltaggi direttamente proporzionali. Questo tipo di circuito integrato viene utilizzato per dispositivi come timer, comparatori, regolatori di voltaggio e amplificatori operazionali.
Circuiti integrati digitali: I circuiti digitali trasmettono segnali discontinui o binari. Il voltaggio di uscita può essere alto o basso. Un alto voltaggio rappresenta il valore booleano 1 ed uno basso il valore 0. I circuiti digitali sono progettati per eseguire diverse operazioni logiche come AND, OR e NAND. Ad esempio, l'operazione logica OR corrisponde all'addizione booleana ed è la base delle operazioni di addizione nell'informatica. In pratica, i circuiti integrati digitali formano la base di tutte le operazioni informatiche. Sono fondamentali per tutti i dispositivi programmabili, per le schede logiche, per i microcontrollori e per la memoria. L'unità standard di misura della quantità di informazione è il bit.

Codifica binaria dei numeri naturali



$1 + 8 + 16 + 64 + 128 = 217$

Circuiti integrali a segnale misto: Esistono chip misti che combinano gli elementi dei chip analogici e quelli digitali, e possono operare come convertitori digitale-analogico e viceversa. Sono circuiti integrali avanzati che rivestono un ruolo fondamentale nell'informatica moderna.
Circuiti quantistici: Sono il prossimo livello dell'evoluzione informatica. È una routine di calcolo che definisce una serie di operazioni quantistiche logiche sui qubit sottostanti, o bit quantistici. Mentre i segnali booleani sono 1 o 0, i qubit possono essere utilizzati in una sovrapposizione di stati. Possiamo vederli come vettori.

Quali sono i chip per computer?

Chip di memoria: conservano dati e programmi su computer e dispositivi di archiviazione. I chip RAM offrono una archiviazione temporanea, mentre i dischi (hard disk e SSD) possono conservare le informazioni in modo permanente.

Chip logici: i chip logici, o processori, elaborano i dati per completare le attività assegnate. Sono il cervello dei dispositivi. Le CPU sono i principali chip logici (altri es: GPU, TPU).
ASIC: i chip integrati per applicazione specifica sono progettati per eseguire routine ripetute di elaborazione per un'applicazione specifica. Possono essere usati ad esempio negli scanner di codici a barre o nel mining di bitcoin.
SoC: il sistema su un chip è un nuovo concetto di chip. Tutti i componenti elettronici dell'intero sistema sono integrati in un singolo chip.

Unità di misura per la memoria

EMBEDDED

PC

Termine decimale	Abbreviazione	Valore	Termine binario	Abbreviazione	Valore	% maggiore
Kilobyte	KB	10^3	Kibibyte	KiB	2^{10}	2%
Megabyte	MB	10^6	Mebibyte	MiB	2^{20}	5%
Gigabyte	GB	10^9	Gibibyte	GiB	2^{30}	7%
Terabyte	TB	10^{12}	Tebibyte	TiB	2^{40}	10%
Petabyte	PB	10^{15}	Pebibyte	PiB	2^{50}	13%
Exabyte	EB	10^{18}	Exbibyte	EiB	2^{60}	15%
Zettabyte	ZB	10^{21}	Zebibyte	ZiB	2^{70}	18%
Yottabyte	YB	10^{24}	Yobibyte	YiB	2^{80}	21%
Ronnabyte	RB	1000^9	Robibyte	RiB	2^{90}	24%
Queccabyte	QB	1000^{10}	Quebibyte	QiB	2^{100}	27%

RETE

DATA CENTER

Ordine di magnitudine della memoria

Misura	Acronimo	Descrizione	Relazione
Byte	B	1	8 bits
Kilobyte	KB	1,024	1.024 B
Megabyte	MB	1,048,576 $= 2^{20}$, Mebibyte	1.024 KB
Gigabyte	GB	1,073,741,824	1.024 MB
Terabyte	TB	1,099,511,627,776 $= 2^{40}$, Tebibyte	1.024 GB
Petabyte	PB	1,125,899,906,842,620	1.024 TB
Exabyte	EB	1,152,921,504,606,850,000	1.024 PB
Zetabyte	ZB	1,180,591,620,717,410,000,000	1.024 EB
Yottabyte	YB	1,208,925,819,614,630,000,000,000	1.024 ZB

Stratificazione dei sistemi di calcolo
Al fine di poter trattare sistemi molto complessi, gli informatici hanno sviluppato delle contromisure, come la decomposizione in livello di astrazione diversi.
Un algoritmo è un insieme di operazioni che, se eseguite, risolvono un determinato problema o una funzione ottenendo sempre lo stesso risultato, mentre l'esecutore è l'uomo o la macchina in grado di risolvere il problema eseguendo l'algoritmo. Le informazioni di ingresso (input) sono quelle che devono essere fornite affinché avvengano le trasformazioni desiderate, mentre le informazioni in uscita (output) sono i risultati prodotti.
Se l'esecutore è un calcolatore, dobbiamo dargli delle istruzioni su ogni singolo passo da eseguire, tramite **comandi**. Per programmare la macchina dobbiamo quindi usare un linguaggio che l'elaboratore possa capire, ossia il linguaggio macchina. È però a basso livello e quindi difficile da manipolare per il programmatore; quindi, possiamo usare dei traduttori per passare da un linguaggio più semplice per l'uomo a quello macchina.

La stratificazione in livelli avviene per colmare il divario tra il modo di pensare alla soluzione dei problemi da parte del programmatore ed il modo di realizzare delle macchine.
I livelli L0, L1 e L2 costituiscono il punto di vista hardware di un sistema di calcolo, mentre da L3 a L5 dal punto di vista del software.

Organizzazione di un calcolatore

Unità di elaborazione centrale (CPU): La CPU è il cervello del computer. È responsabile del recupero, della decodifica e dell'esecuzione delle istruzioni del programma. Esegue anche calcoli matematici e logici. Il chip del processore è identificato dal tipo di processore e dal produttore; questa informazione è solitamente incisa sul chip stesso.
Memoria ad accesso casuale (RAM): la RAM è un chip di computer che archivia temporaneamente dati dinamici per migliorare le prestazioni del computer mentre si lavora. È il luogo di lavoro del computer, dove vengono caricati programmi e dati attivi in modo che ogni volta che il processore li richiede, non si debbano recuperare dal disco rigido. È volatile, quindi perde il suo contenuto una volta spenta l'alimentazione.
Sistema di input/output di base (BIOS): Il BIOS è una memoria di 'sola lettura', costituita da un software di basso livello che controlla l'hardware del sistema e funge da interfaccia tra il sistema operativo e l'hardware. Contiene tutto il codice necessario per controllare la tastiera, lo schermo, le unità disco, le comunicazioni seriale e diverse funzioni varie. Il BIOS di sistema è un chip ROM sulla scheda madre utilizzato durante la routine di avvio per controllare il sistema e prepararsi ad eseguire l'hardware.
Batteria CMOS: Le schede madri hanno un piccolo blocco separato di memoria costituito da un chip RAM CMOS che viene mantenuto in vita da una batteria, anche quando il PC è spento, per impedire la riconfigurazione quando il PC viene acceso. Viene utilizzata per memorizzare informazioni di base sulla configurazione del PC, come: informazioni sulla CPU, dimensioni della RAM, data e ora, ecc.
Memoria cache: La memoria cache è un piccolo blocco di memoria ad altra velocità che migliora le prestazioni del PC precaricando le informazioni della memoria principale e trasmettendole al processore su richiesta.
Chipset del computer: Un chipset è un gruppo di piccoli circuiti che coordinano il flusso di dati da e verso i componenti chiave di un PC, quindi CPU, memoria, cache e tutti i dispositivi posizionati sui bus. Un computer ha due chipset principali:

- Il **northbridge** (chiamato anche controller di memoria), incaricato di controllare i trasferimenti tra il processore e la RAM, motivo per cui si trova fisicamente vicino al processore.
- Il **southbridge** (chiamato anche controller di input/output o controller di espansione) gestisce le comunicazioni tra dispositivi periferici più lenti.

Bus di espansione: È un percorso di ingresso/uscita dalla CPU ai dispositivi periferici. È composto da una serie di slot sulla scheda madre, e le schede di espansione si inseriscono nel bus. I bus di espansione migliorano le capacità del PC consentendo agli utenti di aggiungere funzionalità mancanti ai propri computer inserendo le schede adattatrici negli slot di espansione.
Clock della CPU: L'orologio della CPU sincronizza il funzionamento di tutte le parti del PC e fornisce il segnale temporale di base per la CPU. Utilizzando un cristallo di quarzo, l'orologio della CPU dà vita al microprocessore alimentandolo con un flusso costante di impulsi. Ad esempio, una CPU da 200 MHz riceve 200 milioni di impulsi al secondo dall'orologio. Allo stesso modo, in qualsiasi dispositivo di comunicazione, è possibile utilizzare un orologio per sincronizzare gli impulsi di dati tra mittente e destinatari.

Un "orologio in tempo reale", chiamato anche "orologio di sistema", tiene traccia dell'ora del giorno e rende questi dati disponibili al software. Un "orologio time-sharing" interrompe la CPU a intervalli regolari e consente al sistema operativo di dividere il proprio tempo tra utenti e/o applicazioni attivi.
Switch e Jumpers: Gli switch DIP (Dual In-line Package) sono piccoli interruttori elettronici presenti sul circuito che possono essere attivati o disattivati proprio come un normale interruttore. I pin dei jumper sono piccoli pin sporgenti sulla scheda madre. Un cappuccio o ponte del jumper viene utilizzato per collegare o cortocircuitare una coppia di pin. Quando il ponte è collegato a due pin qualsiasi tramite un collegamento di cortocircuito, completa il circuito e viene raggiunta una determinata configurazione. Posizionando un jumper su un diverso set di pin, è possibile modificare i parametri di una scheda.

Architettura di Von Neumann
Ciclo di Fetch, Decode ed Execute: Questa architettura è stata disegnata per eseguire il seguente ciclo

1) Carica istruzione corrente dalla memoria principale
2) Decodifica istruzione e leggi contenuto dei registri
3) Esegui operazione oppure calcola un indirizzo di memoria
4) Accedi agli operandi in memoria quando necessario
5) Scrivi risultato in un registro e torna a 1

La rappresentazione binaria dei dati
I dati sono rappresentazioni oggettive e non interpretate della realtà: sono informazioni conosciute, grezze, di base. Le informazioni sono una rappresentazione dei dati realizzata in modo da essere comprensibile per l'utente finale.

Rappresentazione binaria (o digitale) delle informazioni: ogni informazione deve essere codificata utilizzando due simboli, per rappresentare la presenza o assenza di carica elettrica, il passaggio o il non passaggio di corrente. L'unità decimale di rappresentazione è il BIT: 0 o 1. È necessaria l'informazione per capire cosa un dato binario rappresenta (e quindi decodificarlo).
Per rappresentare più valori è necessario usare sequenze di bit

1 bit = 2 valori: 0, 1
2 bit = 4 valori = 00, 01, 10, 11
3 bit = 8 valori = 000, 001, 010, 100, 011, 110, 101, 111

8 bit (256 valori) rappresentano 1 **byte**, utilizzato come unità di misura per indicare le dimensioni informatiche.

Esempi:

- Quanti valori posso rappresentare con 5 bit?
  2^5 = 32 valori diversi.
- Quanti valori posso rappresentare con 2 byte?
  2 byte = 16 bit, quindi 2^{16} = 65536 valori diversi.
- Quanti bit mi servono per rappresentare 1000 valori diversi?
  Devo trovare il numero n minimo di bit che soddisfi 2^n = 1000.
  2^{10} = 1024 valori diversi, quindi n = 10.
- Quanti bit mi servono per rappresentare 112 valori diversi?
  7 bit, perché 2^7 = 128 valori diversi.

Rappresentazione posizionale dei numeri naturali ed interi

POTENZA	VALORE		POTENZA	VALORE
2^0	1		2^{10}	1024 (1 KiB)
2^1	2		2^{11}	2048
2^2	4		2^{12}	4096
2^3	8		2^{13}	8192
2^4	16		2^{14}	16384
2^5	32		2^{15}	32768
2^6	64		2^{16}	65536 (65 KiB)
2^7	128		2^{18}	262144
2^8	256		2^{19}	524288
2^9	512		2^{20}	1048576 (1 MiB)

Sistema binario

da binario a decimale: prendiamo 10100101 in notazione binaria (o in base 2). A cosa corrisponde il sistema decimale di tale numero?

cifre	1	0	1	0	0	1	0	1
posizione	7	6	5	4	3	2	1	0

Facciamo quindi $1 * 2^7 + 0 + 1 * 2^5 + 0 + 0 + 1 * 2^2 + 0 + 1 * 2^0$

E quindi 128+0+32+0+0+4+0+1 = 165

Esempi:

- 1000010 = $1 * 2^6 + 1 * 2^1$ = 64 + 2 = 66
- 1011010 = $1 * 2^6 + 1 * 2^4 + 1 * 2^3 + 1 * 2^1$ = 64 + 16 + 8 + 2 = 90
- 1000011 = $1 * 2^6 + 1 * 2^1 + 1 * 2^0$ = 64 + 2 + 1 = 67
- 1010011 = $1 * 2^6 + 1 * 2^4 + 1 * 2^1 + 1 * 2^0$ = 64 + 16 + 2 + 1 = 83
- 0000010 = $1 * 2^1$ = 2
- 1010010 = $1 * 2^6 + 1 * 2^4 + 1 * 2^1$ = 64 + 16 + 2 = 82
- 11111111 = $1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$ = 128 + 64 + 32 +16 +8 + 4 + 2 + 1 = 255

da decimale a binario: dato un numero N rappresentato in base 10, per trovare la sua rappresentazione in base 2 si deve calcolare la divisione intera di N per 2: N/2 = N' con resto R', con R 0 o 1, creando una rappresentazione binaria partendo da destra. Si ripete il procedimento fino a quando il risultato della divisione è uguale a 0.

Esempi:

- Consideriamo il numero 13_{10} (13 in base 10)

  13/2 = 6 resto 1
  6/2 = 3 resto 0
  3/2 = 1 resto 1
  1/2 = 0 resto 1

  Leggendo i resti dal basso verso l'alto, si ha che 13 (base 10) è 1101 (base 2)

Altri tipi di numerazione

Codice ottale: cifre da 0 a 7.

Codice esadecimale: cifre da 0 a 9 + lettere da A ad F, che corrispondono ai numeri da 10 a 16.

È utilizzato per rappresentare numeri grandi come il MAC address o gli indirizzi IPv6.

Convertiamo l'esadecimale CA5A in decimale:

C	A	5	A
16^3	16^2	16^1	16^0

$12 \times 16^3 + 10 \times 16^2 + 5 \times 16^1 + 10 \times 16^0$

49152	2560	80	10

49152+2560+80+10 =51802.

Quindi $CA5A_{16}$ in esadecimale corrisponde a 51802_{10}.

Consideriamo ad esempio il numero 345_{10} e calcoliamo la sua rappresentazione in base sedici:

345/16 = 21 resto 9

21/16 = 1 resto 5

1/16 = 0 resto 1

Leggendo i resti dal basso verso l'alto si ha 159_{16}

Codifica binaria (4 bit)	↔	Cifra esadecimale	↔	Valore (decimale)
0000	↔	0	↔	0
0001	↔	1	↔	1
0010	↔	2	↔	2
0011	↔	3	↔	3
0100	↔	4	↔	4
0101	↔	5	↔	5
0110	↔	6	↔	6
0111	↔	7	↔	7
1000	↔	8	↔	8
1001	↔	9	↔	9
1010	↔	A	↔	10
1011	↔	B	↔	11
1100	↔	C	↔	12
1101	↔	D	↔	13
1110	↔	E	↔	14
1111	↔	F	↔	15

Rappresentazione di numeri più complessi

In realtà, una semplice codifica binaria come quella discusa fino ad ora non è sufficiente per i numeri naturali, razionali, i caratteri, ecc. Per loro vengono utilizzate rappresentazioni differenti.

Rappresentazione tramite modulo e segno

Quando si considerano valori interi, per rappresentare il segno si può pensare di usare uno dei bit, quello più significativo, quindi quello più a sinistra, e usare gli altri bit per rappresentare il numero. Si usa 1 per rappresentare i numeri negativi e lo 0 per rappresentare quelli positivi.

Esempio:

15 = 1111 che diventa in base al segno: +15 = 01111, -15 = 11111

Se volessi scrivere questi numeri all'interno di 8 bit, il bit del segno sarebbe quello all'estrema sinistra, indipendentemente dalla lunghezza del nostro numero.

Quindi: -15 = 10001111 e +15 = 00001111

Rappresentazione in complemento a 2

Prendendo un numero con base 2, faccio il complemento ad 1, ossia riscrivo il numero con i bit **invertiti**. Ad esempio, se il numero è 101011, facendo il complemento ad 1 esce 010100.

Per fare il complemento a 2, applico il complemento ad 1 e poi si aggiunge 1 al numero. Quindi, dall'esempio esce 010101.

Un altro modo è quello di ricopiare i numeri uguali al numero iniziale f**ino a quando non incontro il primo 1**, e da lì, gli altri numeri vengono ricopiati in complemento ad 1.

Esempio:

101000

$$\begin{array}{r} 010111 + \\ 1 = \\ \hline 11000 \end{array}$$

con il metodo somma ci sono tanti riporti quindi scomodo.

Quindi con secondo metodo scrivo le prime 4 cifre da destra normalmente, poi dopo il primo 1, faccio complemento ad 1. Quindi esce 011000.

Range numeri interi su 4 bit

Positivi da 0 a 2^3-1=7

0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7

Negativi: da -1 a -2^3

1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

0 in complemento a 2: not(0000)+1=1111+1=0000

-8 in complemento a 2: not(1000)+1=0111+1=1000

Operazioni con i numeri binari

Somma:

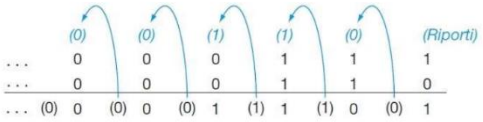


Figura 3.1 Somma binaria: sono visibili i riporti da destra verso sinistra. Nella posizione del bit più a destra si somma 1 a 0 e il risultato della somma di questo bit è 1, mentre il riporto generato vale 0. Quindi l'operazione da effettuare sul secondo bit a destra sarà 0 + 1 + 1. Questa genera uno 0 come somma e un riporto pari a 1. Sul terzo bit si esegue la somma 1 + 1 + 1, producendo una somma pari a 1 e un riporto pari a 1. Sul quarto bit si esegue 1 + 0 + 0, generando 1 come bit di somma e nessun riporto.

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 0$ con riporto di 1

Esempi:

$$\begin{array}{r} 1111 \\ 1011 + \\ 1101 = \\ \hline 11000 \end{array}$$

$$\begin{array}{r} 1 \\ 101 \\ 1110 + \\ 10011 + \\ 10111 = \\ \hline 110000 \end{array}$$

Sottrazione

Sottrazione

$$\begin{array}{r} 11000 - \\ 111 = \end{array} \Rightarrow \begin{array}{r} 11000 - \\ 111 = \end{array}$$

$$\begin{array}{r} 10000 - \\ 11\ 1 = \\ \hline 1 \end{array} \Rightarrow \begin{array}{r} 10110 - \\ 11\ 1 = \\ \hline \boxed{1000\ 1} \end{array}$$

- A - B = A + (-B)
- Oppure si esegue come una normale sottrazione.
- Bisogna però tenere conto che ogni volta che si deve SOTTRARRE dalla cifra 0 la cifra 1, occorre CHIEDERE IN PRESTITO una unità alla cifra di ordine immediatamente superiore e che essa vale 2 UNITA' dell'ordine immediatamente inferiore.

$0 - 0 = 0$

$0 - 1 = /$

$1 - 0 = 1$

$1 - 1 = 0$

NUMERO DECIMALE
29 -
14 =
15

prestito		2	2	2		
minuendo	1	1	1	0	1	-
sottraendo		1	1	1	0	=
differenza	0	1	1	1	1	

Esempi:

$$\begin{array}{r} 1101 - \\ 1011 = \\ \hline 0010 \end{array}$$

$$\begin{array}{r} 10111 - \\ 1110 = \\ \hline 1001 \end{array}$$

Quando il risultato di un'operazione non può essere rappresentato con l'hardware a disposizione si verifica un **overflow**.

Moltiplicazione:

- La lunghezza del prodotto tra un numero di a cifre ed uno di b cifre è $c \leq a+b$. Provare per credere...
- Algoritmo semplice:
- metto la cifra del moltiplicando se il moltiplicatore è 1
- Metto 0 altrimenti

Moltiplicando	×	1000_{dec}
Moltiplicatore		1001_{dec}
		1000
		0000
		0000
		1000
Prodotto		1001000_{dec}

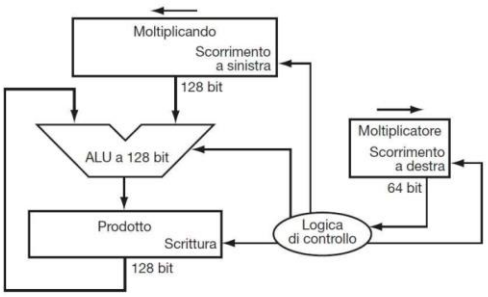


Figura 3.3 **Prima versione dell'hardware che realizza la moltiplicazione.** Il registro Moltiplicando, la ALU

Iterazione	Moltiplicando	Moltiplicatore	Prodotto
Inizio	00001000	1001	00000000
1	00001000	1001	00001000
2	00010000	100	00001000
3	00100000	10	00001000
4	01000000	1	01001000
Fine			01001000

$0 * 0 = 0$
$0 * 1 = 0$
$1 * 0 = 0$
$1 * 1 = 1$

Esempi:

```
 1011 ·
  101 =
 ----
 1011
 0000-   addizione
1011--
------
110111
```

```
 11001 ·
     11 =
 -----
 11001
11001-
------
1001011
```

Se moltiplico numeri con segno, lo posso tralasciare e considerare alla fine: avrò un numero negativo solo se i segni sono discordi.

I caratteri

Lo standard per rappresentare caratteri è ASCII, dove vengono rappresentati con 7 bit. C'è anche quella estesa dove sono rappresentati con 8 bit, e l'UNICODE per rappresentarli con 16 bit (per caratteri cinesi arabi, ecc.).

Rappresentazione di numeri razionali: lo standard IEEE-754 per le operazioni floating-point
I numeri IEEE-754 sono formati da 32 bit, di cui 1, il primo, che esprime il segno, i secondi 8 bit che esprimono l'esponente, e i restanti 23 bit che esprimono la mantissa.

Da IEEE-754 a decimale (https://youtu.be/LXF-wcoeT0o?si=YdQfr-CFJI60ueAN**)**
Prendiamo come esempio il numero: 0100 0011 0101 0100 0000 0000 0000 0000
Il primo bit esprime il numero: visto che è 0, il nostro equivalente decimale sarà positivo. I prossimi 8 bit esprimono l'esponente. Dobbiamo quindi convertire la rappresentazione binaria nella sua equivalente decimale.
Quindi: 10000110 = $1 * 2^1 + 1 * 2^2 + 1 * 2^7$ = 2 + 4 + 128 = 134
Da questo valore dobbiamo eliminare l'exponential bias (polarizzazione?), ossia 127. Tutti gli esponenti con valore minore di 127 esprimono un esponente negativo. L'esponenziale è quindi 134 – 127 = 7.
Gli ultimi bit sono la mantissa. Si moltiplica l'n bit per 2^{-n} e si fa la somma.
Quindi in questo caso, la mantissa è $1 * 2^{-1} + 1 * 2^{-3} + 1 * 2^{-5}$ = 0.65625
La formula finale è

$$(-1)^S * 2^{esponente-127} * (1 + mantissa)$$

Quindi il risultato dell'esempio sarebbe $(-1)^0 * 2^7 * (1 + 1 * 2^{-1} + 1 * 2^{-3} + 1 * 2^{-5})$.

Da decimale a IEEE-754 (https://youtu.be/8afbTaA-gOQ?si=KnG4T6XEkH-7sGl6**)**
Prendiamo come esempio il numero decimale 263,3.
Dobbiamo dividere il numero 263 per 2 fino a quando non otteniamo 0.

263/2 = 131 resto 1
131/2 = 65 resto 1
65/2 = 32 resto 1
32/2 = 16 resto 0
16/2 = 8 resto 0
8/2 = 4 resto 0
4/2 = 2 resto 0
2/2 = 1 resto 0
½ = 0 resto 1

263 in decimale è quindi 100000111.
Per convertire lo 0.3, dobbiamo moltiplicarlo per 2 e tenere conto dello 0 o dell'1 in questo modo:

0.3 * 2 = 0.6 0
0.6 * 2 = 1.2 1
0.2 * 2 = 0.4 0
0.4 * 2 = 0.8 0
0.8 * 2 = 1.6 1
0.6 * 2 = 1.2 1

A questo punto sappiamo che entrerebbe in un loop.
Quindi 0.3 sarebbe 01 0011 0011 0011 …
263.3 in binario sarebbe quindi 100000111.0100110011…
A questo punto dobbiamo rappresentare il numero in notazione scientifica, in questo caso facciamo uno shift di 8 volte. Quindi: 1.000001110100110011…$* 2^8$. I numeri dopo la virgola sono la mantissa.
Per l'esponente dobbiamo fare esponente 8 + 127 = 135, da trasformare in binario nello stesso modo iniziale. Ora rendiamo il numero in formato IEEE: 0 10000111 00000111010011001100110
segno esponente mantissa

RISC V (https://youtube.com/playlist?list=PLeFoe3FAv19dfNDjiUeJ6sD91vzPRqbch&si=QkOg7Ppct2Xaogpz**)**
La architettura RISC V è stata creata all'università di Berkeley nel 2010.

Istruzioni
Addizione: In C sarà a = b + c; mentre in RISC V sarà **add a, b, c**
Add è l'operazione da eseguire, mentre a, b e c sono gli operandi, di cui il primo è sempre quello di destinazione.
Esiste anche addi, somma immediata dove addi a, b, 20 quindi per sommare delle costanti.
Sottrazione: In C sarà a = b – c; mentre in RISC V sarà **sub a, b, c**
Sub è l'operazione da fare, b e c gli operandi sorgente e a l'operando destinazione.

Esempio:

a = b + c – d;

In RISC V spezziamo l'operazione e creiamo un temp:

```
add t, b, c      # t = b + c
sub a, t, d      # a = t – d
```

Il RISC V ha un registro da 32 bit. Ogni registro ha un suo fine e funzionalità.

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	X4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

Nel registro zero, c'è il valore 0. ra ha l'indirizzo dove la funzione deve essere ritornata. I registri che iniziano con s (0, 11) contengono variabili. I registri t (0, 6) contengono risultati temporanei (registri temporanei, come lo t dell'esempio di prima). I registri a (0, 7) contengono argomenti.
Possono essere quindi utilizzati sia i nomi che il registrer number, ma è più facile da leggere se si usa il nome.

Quindi, a = b + c; dovrebbe essere scritta così:

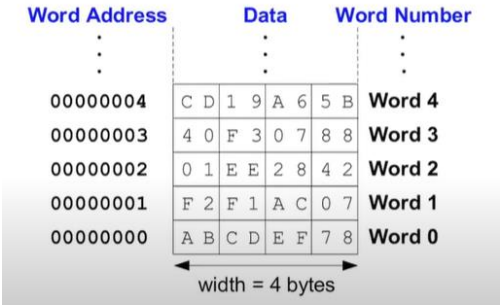
```
#s0 = a, s1 = b, s2 = c
add s0, s1, s2
```

viene usato per i commenti. addi viene usata invece in questo modo:

```
# s0 = a, s1 = b
Addi s0, s1, 6
```

Maggior parte delle volte, i dati sono troppi per essere tenuti solo in 32 registri. Per questo, si tengono i dati in memoria, che è più grande, ma più lenta, e teniamo le variabili più importanti in registri. La memoria può essere word-addressable o byte-addressable.

Ogni parola da 32 bit ha un suo indirizzo unico.



Abbiamo bisogno di istruzioni, dette load, che porta i dati dalla memoria la registro.
Quindi abbiamo **lw** (load word)

lw t1, 5(s0)

lw destinazione, offset(base)

offset(base) quindi è s0 + 5, e quindi t1 ha i dati dell'indirizzo s0 + 5.

In modo simile, store prende un valore dal registro e lo porta in memoria.
Quindi **sw** (store word) funziona così:

sw t4, 0x3 (zero) # scrive il valore in t4 sulla parola in memoria 3

Ogni byte ha un unico indirizzo. Le parole vanno prese o caricate (load o store) in singoli byte, e quindi si usa **lb** (load byte) e **sb** (store byte). Ogni parola è formata da 32 bit, quindi 4 byte, e gli indirizzi di parola incrementano di 4.
Quindi se faccio: lw s3, 8(zero) # legge la parola ad indirizzo 8 (quindi la terza parola, 0, 4, 8)

Generare costanti:
alcune istruzioni come addi possono prendere un immediate da 12 bit.

	#s0 = a, s1 = b
int a = -372;	addi s0, zero, -372
int b = a + 6;	addi s1, s0, 6

Se abbiamo bisogno di una costante più lunga, da più di 32 bit, abbiamo un'istruzione chiamata **lui** (load upper immediate). Mette un immediate nei primi 20 bit del registro di destinazione, e 0 negli altri 12 bit.

	# s0 = a
int a = 0xFEDC8765;	lui s0, 0xFEDC8
	addi s0, s0, 0x765

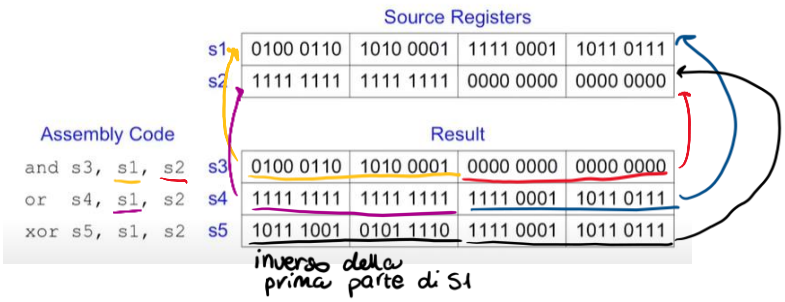
Se il 12 bit immediate ha un 1 nel bit più significativo, allora avrà un segno negativo.

int a = 0xFEDC8EAB;	nota: -341 = 0xEAB;

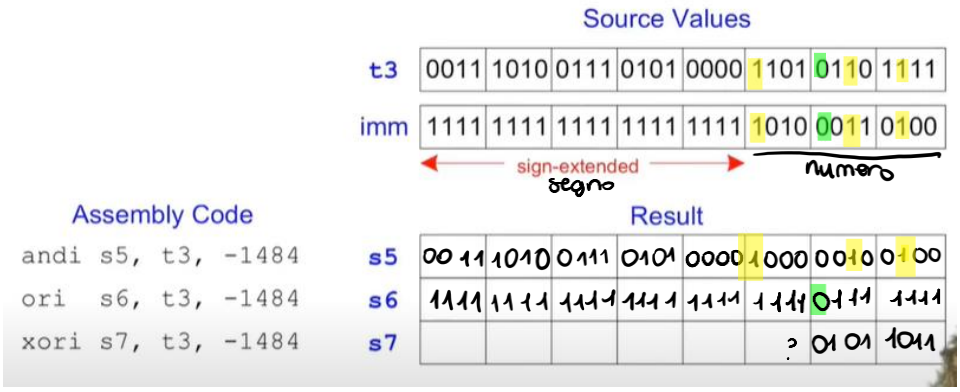
s0 = a
lui s0, 0xFEDC9 # s0 = 0xFEDC9000
addi s0, s0, -341 # s0 = 0xFEDC9000 + 0xFFFFFEAB = 0xFEDC8EAB

Istruzioni logiche e shift

- **and** per masking i bit
  0xF234012F and 0x000000FF = 0x0000002F
- **or** per combinare i bit
  0xF2340000 or 0x000012BC = 0xF23412BC
- **xor** per invertire i bit
  A xor -1 = not A (-1 = 0xFFFFFFFF)



Queste istruzioni possono essere anche con i, quindi andi, ori e xori. Gli immediates sono lunghi 12 bit, con il segno nei restanti bit.



Le istruzioni shift prendono due risorse e shiftano la prima sulla seconda.

- **sll**: shift left logical

  sll t0, t1, t2 # t0 = t1 << t2
- **srl**: shift right logical

  srl t0, t1, t2 # t0 = t1 >> t2
- **sra**: shift right arithmetic

  sra t0, t1, t2 # t0 = t1 >>> t2

Uno shift logico riempie i primi bit con 0, mentre quello aritmetico li riempie con una copia di bit assegnati del return number (?).
Lo shift a sinistra è equivalente a moltiplicare in potenze di due, mentre lo shift a destra è equivalente a dividere in potenze di due, e logical viene usato per i numeri senza segno mentre l'arithmetic viene usato per quelli con il segno, per preservare il bit del segno.
Vengono utilizzati solo gli ultimi 5 bit della seconda risorsa (nel registro).
Ci sono anche con l'immediate, quindi slli, srli e srai, per shiftare con una costante.

Moltiplicazione:
Quando moltiplichi due numeri da 32 bit, uscirà un risultato da 64 bit. Abbiamo quindi due istruzioni: mul e mulh.

mul s0, s1, s2 s0 = gli ultimi 32 bit (quelli a destra) nel risultato
mulh s0, s1, s2 s0 = i primi 32 bit (quelli a sinistra) del risultato

Quindi per avere un risultato completo

mulh s4, s1, s2
mul s3, s1, s2
# {s4, s3} = s1 x s2

Divisione:
Quando dividi 32 bit, hai 32 bit di quoziente e 32 bit di resto. Abbiamo quindi due istruzioni: div e rem.

div s3, s1, s2 # s3 = s1 / s2 (divisione)
rem s4, s1, s2 # s4 = s1 % s2 (resto)

Istruzioni di controllo di flusso: branch e jump
Fino ad adesso, le istruzioni erano eseguite in sequenza, ma branch e jump ci permettono di controllare questa sequenza e cambiare l'ordine in cui eseguire le istruzioni.

Ci sono due tipi di branch: condizionali e non-conditionali. I condizionali sono chiamati branch e i non condizionali jump.

- Condizionali:
  - beq: branch if equal
  - bne: branch if not equal
  - blt: branch if less than than
  - bge: branch if greater than or equal
- Non condizionali:
  - j: jump
  - jr: jump register
  - jal: jump and link
  - jalr: jump and link register

Queste istruzioni condizionali prendono due sorgenti e compara i loro valori.

Esempio 1:

```
        addi s0, zero, 4        # s0 = 0 + 4 = 4
        addi s1, zero, 1        # s1 = 0 + 1 = 1
        slli s1, s1, 2          # s1 = 1 << 2 = 4 perché shifting di due posizioni equivale a 2^2
        beq s0, s1, target      # il programma continua da target
        addi s1, s1, 1          # le funzioni dopo beq NON VENGONO ESEGUITE
        sub s1, s1, s0
target:                         # label
        add s1, s1, s0          # s1 = 4 + 4 = 8
```

Target è chiamata label ed è seguita da due punti (:).

Esempio 2:

```
        addi s0, zero, 4        # s0 = 0 + 4 = 4
        addi s1, zero, 1        # s1 = 0 + 1 = 1
        slli s1, s1, 2          # s1 = 1 << 2 = 4
        bne s0, s1, target      # il target non viene preso perché s1 e s2 sono uguali, e quindi il
                                  programma continua normalmente
        addi s1, s1, 1          # s1 = 4 + 1 = 5
        sub s1, s1, s0          # s1 = 5 – 4 = 1
target:
        add s1, s1, s0          # s1 = 1 + 4 = 5
```

Esempio 3:

```
        addi s0, zero, 4        # s0 = 0 + 4 = 4
        addi s1, zero, 1        # s1 = 0 + 1 = 1
        j target                # jump to target
        srai s1, s1, 2          # queste istruzioni NON vengono eseguite
        addi s1, s1, 1
        sub s1, s1, s0
target:
        add s1, s1, s0          # s1 = 1 + 4 = 5
```

Condizionali e loop (if, else, while, for)
if statements

```
                                #s0 = f, s1 = g, s2 = h, s3 = i, s4 = j
if (i == j)                     bne s3, s4, l1          # i == j ?
        f = g + h;              add s0, s1, s2          # f = g + h
f = f – i;                      l2:
                                        sub s0, s0, s3  # f = f – 1
```

if/else statements

```
                                #s0 = f, s1 = g, s2 = h, s3 = i, s4 = j
if (i == j)                     bne s3, s4, else        # i == j ?
        f = g + h;              add s0, s1, s2          # f = g + h
                                j done
else                            else:
        f = f – i;                      sub s0, s0, s3  # f = f – 1
                                done:
```

while loops

```
// determinate the power of x
// such that 2^x = 128          # s0 = pow, s1 = x
int pow = 1;                    addi s0, zero, 1        # pow = 1
int x = 0;                      addi s1, zero, 0        # x = 0
                                add t0, zero, 128
while (pow != 128) {            while:  beq s0, t0, done        # done ?
        pow = pow * 2;                  slli s0, s0, 1          # pow = pow * 2
        x = x + 1;                      add s1, s1, 1           # x = x + 1
}                                       j while
                                done:
```

for loops
I for in C hanno più parti: l'inizializzazione, la condizione e un'operazione in loop, oltre allo statement da eseguire.

```
// add the numbers form 0 to 9  # s0 = i, s1 = sum
int sum = 0;                    addi s1, zero, 0        # sum = 0
int i;                          add s0, zero, 0         # i = 0
                                add t0, zero, 10
for (i = 0; i != 10; i = i + 1) {       for:    beq s0, t0, done
        sum = sum + i;                          add s1, s1, s0          # sum = sum + i
}                                               add s0, s0, 1           # i = i + 1
                                                j for
                                done:
```

```
// add the powers of 2 form 1 to 100
int sum = 0;
int i;

for (i = 1; i < 101; i = i * 2) {
        sum = sum + i;
}
```

```
#s0 = i, s1 = sum
addi s1, zero, 0          # sum = 0
addi s0, zero, 1          # i = 1
addi t0, zero, 101
for:    bge s0, t0, done  # i < 101 ?
        add s1, s1, s0    # sum = sum + i
        slli s0, s0, 1    # i = i * 2
        j for
done:
```

Questo esempio si può fare anche con slt (set if less than), compara due numeri e mette un bit nel risultato, che è 0 se il primo è più grande, e 1 se è più piccolo.

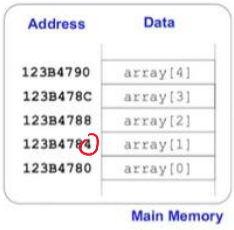
```
addi s1, zero, 0
addi s0, zero, 1
t0, zero, 101
for:
        slt t2, s0, t0
        beq t2, zero, done
        add s1, s1, s0
        slli s0, s0, 1
        j for
done:
```

Array

Gli array rappresentano un grande numero di dati simili, e ha un index (puntatore per accedere all'elemento) e una size (numero di elementi).

```
int array[5];
array[0] = array[0] * 2;
array[1] = array[1] * 2;
```

```
# s0 = indirizzo base array
lui s0, 0x123B4
addi s0, s0, 0x780      #ind array
lw t0, 0(s0)            #array[0]
slli t0, t0, 1          # array[0]*2
sw t0, 0(s0)            #array[0] = array[0]*2
lw t0, 4(s0)
slli t0, t0, 1
sw t0, 4(s0)            #array[1] = array[1]*2
```



Function calls

- Caller: la funzione che chiama la funzione (esempio: main)
- Callee: la funzione chiamata (esempio: sum)

```
caller
void main () {
    int y;
    y = sum (42, 7);
}
```

```
callee
int sum (int a, int b) {
    return (a + b);
}
```

Per 'saltare' alla funzione usiamo **jal**.
jr ra serve per ritornare, quindi la funzione callee quando ritorna, salva il return address dell'istruzione dopo il jal per tornare alla funzione principale.

Esempio:

codice C

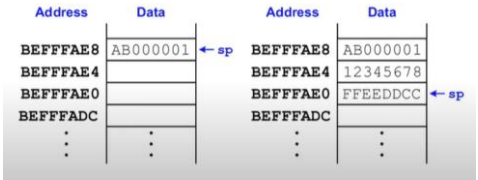
```
int main () {
        int y;
        y = diffofsums(2, 3, 4, 5);        // 4 argomenti
}
int diffofsums(int f, int g, int h, int i) {
        int result;
        result = (f + g) – (h + i);
        return result;                     // return value
}
```

Codice RISC-V

```
#s7 = y
main:
        # 4 argomenti
        addi a0, zero, 2
        addi a1, zero, 3
        addi a2, zero, 4
        addi a3, zero, 5
        jal diffofsums        # chiamo la funzione callee. jal diffofsums è pseudocodice
                                di jar ra, diffofsums
        add s7, a0, zero      # il risultato è ora in a0
diffofsums:
        add t0, a0, a1        # t0 = f + g
        add t1, a2, a3        # t1 = h + 1
        sub s3, t0, t1        # s3 (risultato) = (f + g) – (h + i)
        add a0, s3, zero      # mettiamo il risultato in a0 (si può fare anche direttamente
                                                    sub a0, t0, t1)
        jr ra                 # ritorna alla funzione caller
```

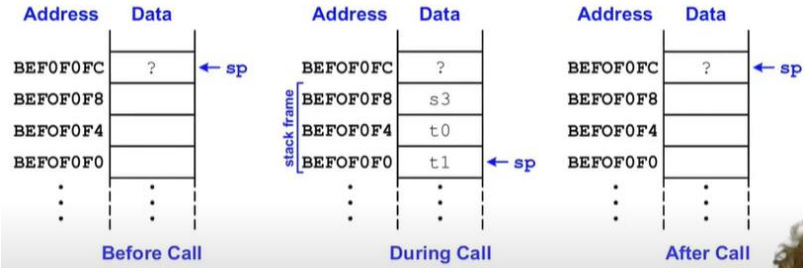
Stack

Possiamo usare gli stack per memorizzare temporaneamente dei registri. La memoria di uno stack si espande o contrae in base a quanta memoria è necessaria. Lo stack è come una pila di libri: quando vado a prendere quello sopra alla pila, sarà anche l'ultimo 'inserito'.
Nell'architettura del computer, lo stack è 'sottosopra', ossia parte da sopra e scende in basso.



Lo stack pointer punta all'ultimo elemento inserito nello stack: il top.
Per memorizzare valori nello stack:

```
diffofsums:
        addi sp, sp, -12        # fa spazio sullo stack per tre registri
        sw s3, 8(sp)            #salva s3 sullo stack
        sw t0, 4(sp)            #salva t0 sullo stack
        sw t1, 0(sp)            #salva t1 sullo stack
        add t0, a0, a1          # t0 = f + g
        add t1, a2, a3          # t1 = h + 1
        sub s3, t0, t1          # s3 (risultato) = (f + g) – (h + i)
        add a0, s3, zero        # mettiamo il risultato in a0 (si può fare anche direttamente
                                                              sub a0, t0, t1)
        lw s3, 8(sp)            # ripristina s3 nello stack
        lw t0, 4(sp)            # ripristina t0 nello stack
        lw t1, 0(sp)            # ripristina t1 nello stack
        addi sp, sp, 12         # dealloca lo spazio dello stack
        jr ra                   # ritorna alla funzione caller
```



Anche il ra (return address) viene salvato nello stack. Si fa ad esempio sw ra, 0(sp) e per fare il restore si fa lw ra, 0(sp).
Il caller mette gli argomenti tra a0 e a7, salva ipotetici registri in t0-t6, salva ra, chiama la funzione con jal callee, ripristina i registri e cerca il risultato in a0.
Il callee salva registri che possono essere disturbati in s0-s11, esegue le istruzioni, mette il risultato in a0, ripristina i registri e ritorna con jr ra.

Funzioni ricorsive

Le funzioni ricorsive sono funzioni che si richiamano da sole. In assembly, dobbiamo pensarle come se semplicemente chiamassero altre funzioni e non se stesse, e poi si penserà dopo come gestire le cose salvate nello stack.

```
int factorial (int n) {
        if (n <= 1)
                return 1;

        else
                return (n*factorial(n-1));

}
```

```
factorial:
        addi t0, zero, 1        # temp = 1
        bgt a0, t0, else        # se n>1, vai ad else
        addi a0, zero, 1        #se no, ritorna 1
        jr ra
else:
        addi a0, a0, -1         # n = n – 1
        jal factorial
        mul a0, a0, a0          # a0 = n * factorial(n – 1)
        jr ra
```

n quindi salviamo n in t0

return value: factorial (n-1)

Quindi la versione completa è:

```
factorial:
        addi sp, sp, -8         # alloca per i registri
        sw a0, 4(sp)
        sw ra, 0(sp)
        addi t0, zero, 1        # temp = 1
        bgt a0, t0, else        # se n>1, vai ad else
        addi a0, zero, 1        #se no, ritorna 1
        addi sp, sp, 8          # ripristina sp
        jr ra
else:
        addi a0, a0, -1         # n = n – 1
        jal factorial
        lw t1, 4(sp)            # ripristina n in t1
        lw ra, 0(sp)            # ripristina ra
        addi sp, sp, 8          # ripristina sp
        mul a0, t1, a0          # a0 = n * factorial(n – 1)
        jr ra
```

Circuiti combinatori

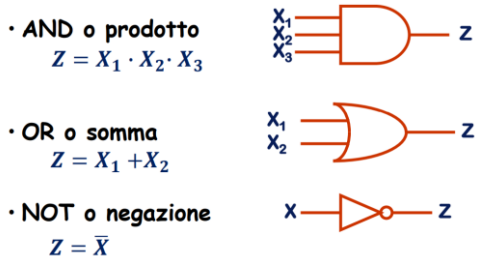
Una rete logica è tipicamente un sistema elettronico con queste caratteristiche:

- Unidirezionale, cioè in cui i segnali viaggiano dagli ingressi verso le uscite e non viceversa. Quindi, il sistema non è reversibile.
- I cui segnali di ingresso e di uscita sono digitali, cioè ogni filo assume valori 0, 1, indeterminato o alta-impedenza.
- Descrivibile con equazioni dell'algebra booleana, se si assume una situazione ideale (i cui valori dei segnali sono solo 0 o 1).

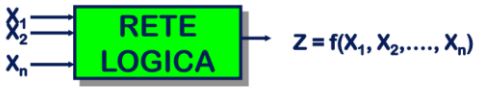
Le reti possono essere combinatorie o sequenziali:

- Reti combinatorie (RC o CN, Combinatorial Network)
    - In qualunque istante, le uscite sono funzione del valore che gli ingressi hanno in quell'istante.
    - Il comportamento (uscite in funzione degli ingressi) è descritto da una tabella.
- Reti sequenziali
    - In un determinato istante, le uscite sono funzione del valore che gli ingressi hanno in quell'istante e dei valori che gli ingressi hanno assunto precedentemente.
    - Presentano strati interni (sono quindi reti dotate di memoria), e quindi la descrizione è più complessa di una semplice tabella.
    - Presentano anelli di richiusura.

Una qualsiasi rete logica può essere realizzata con porte AND, OR e NOT

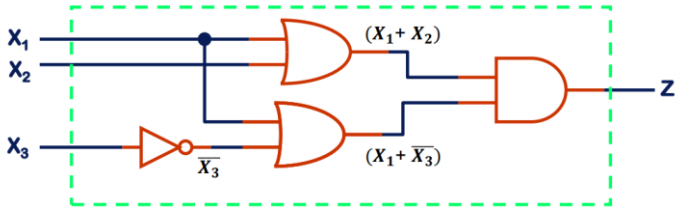


Data una rete logica, possiamo descrivere la relazione fra ingresso X e uscita Z tramite una funzione f.



Ad esempio, potremmo avere:

$$Z = f(X_1, X_2, X_3) = (X_1 + X_2) \cdot (X_1 + \overline{X_3})$$



Porte NAND e NOR

La porta logica NAND implementa la funzione logica di negazione del prodotto logico, ovvero restituisce il negato del prodotto logico degli ingressi. La porta logica NOR implementa la funzione logica di negazione della somma logica, ovvero restituisce in uscita il negato della somma logica degli ingressi.

- NAND

X_1	X_2	Z
0	0	1
0	1	1
1	0	1
1	1	0

$$Z = \overline{X_1 \cdot X_2}$$

- NOR

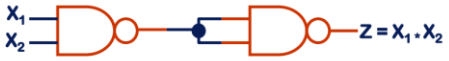
X_1	X_2	Z
0	0	1
0	1	0
1	0	0
1	1	0

$$Z = \overline{X_1 + X_2}$$

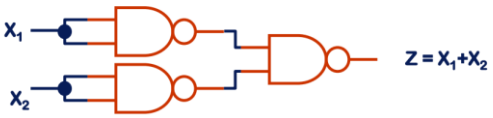
Utilizzando porte solo NAND (o solo NOR) è possibile realizzare qualunque rete logica. Infatti:

- NOT

- AND



- OR



Una funzione logica può sempre essere espressa da una tabella che prende il nome di Tabella di Verità. Una funzione logica di n variabili ammette 2^n possibili valori. Tale funzione viene descritta da una tabella che ha sulla sinistra le 2^n possibili configurazioni, e a destra i valori 0, 1 a seconda del valore della funzione

Esempio (funzione di tre variabili) $z = f(x_1, x_2, x_3)$

x_1	x_2	x_3	z
0	0	0	f(0,0,0)
0	0	1	f(0,0,1)
0	1	0	f(0,1,0)
0	1	1	f(0,1,1)
1	0	0	f(1,0,0)
1	0	1	f(1,0,1)
1	1	0	f(1,1,0)
1	1	1	f(1,1,1)

Una funzione booleana può essere definita tramite tabella di verità. È possibile derivare un'espressione algebrica a partire dalla tabella di verità in almeno due modi:

a	b	c	z
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

- Identificando tutte le righe che producono un 1 e sommando (OR) i corrispondenti termini (prodotti, ovvero AND, di variabili) che producono tale 1. Tali termini prodotto prendono il nome di **mintermini**. L'espressione risultante è detta **somma di prodotti**. Nella tabella esempio:

$$z = \bar{a}\cdot\bar{b}\cdot\bar{c} + \bar{a}\cdot b\cdot\bar{c} + a\cdot\bar{b}\cdot\bar{c} + a\cdot\bar{b}\cdot c + a\cdot b\cdot\bar{c}$$

- Identificando tutte le righe che producono uno 0 e facendo prodotto (AND) dei corrispondenti termini negati (somme, ovvero OR, di variabili) che producono tale 0. Tali termini somma prendono il nome di **maxtermini**. L'espressione risultante è detta **prodotto di somme**. Nella tabella esempio:

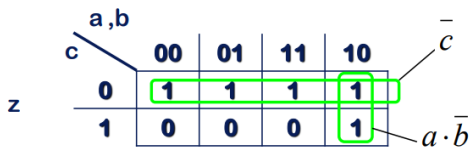
$$z = (a+b+\bar{c})\cdot(a+\bar{b}+\bar{c})\cdot(\bar{a}+\bar{b}+\bar{c})$$

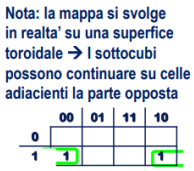
Mappa di Karnaugh (https://youtu.be/XiKTusssvPc?si=Zuyp7z2BeT_Q6BXZ**)**

Può essere comodo semplificare l'espressione booleana, per ottenere espressioni booleane più semplici e facili da verificare, e per sintetizzare una rete logica con un minor numero di porte (minor area).

La mappa di Karnaugh è una rappresentazione alternativa della tabella di verità che, sfruttando la proprietà di assorbimento ($xy + \bar{x}y = y$), permette rapidamente di identificare termini più semplici. Una mappa di Karnaugh si ottiene:

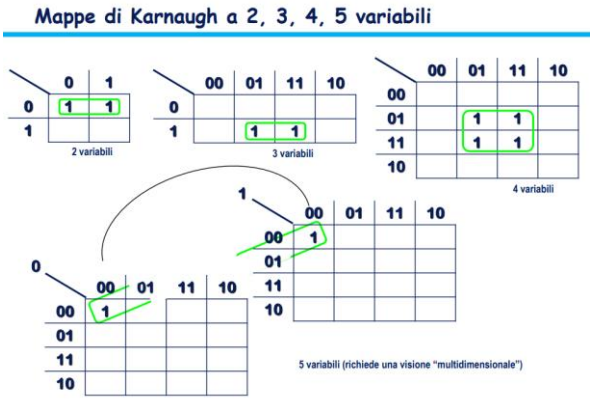
- disponendo gli indici generati dalle variabili di ingresso in modo che due indici adiacenti differiscano per una sola cifra.





- Localizzando i gruppi adiacenti di 1 con un numero di 1 pari a una potenza di 2 (detti sottocubi di ordine k se contengono 2^k uno)
- Un termine è identificato dagli indici che rimangono "fissi" nel sottocubo. Esempio:

$$z = \bar{c} + a * \bar{b}$$

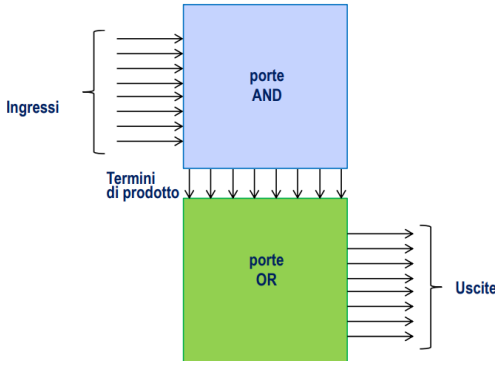


Note sulle mappe di Karnaugh:

Se la mappa contiene dei non-specificato (X o -), si può interpretare come è più conveniente per ottenere sottocubi più grandi. Se ci sono più possibilità con sottocubi piu' piccoli e più grandi, si preferiscono sempre coperture con sottocubi più grandi in quanto hanno meno ingressi. Si prendono solo i termini strettamente necessari per ottenere una lista di copertura "a costo minimo".

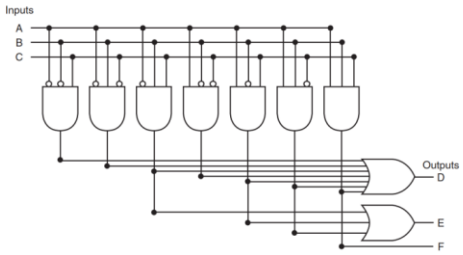
Programmable Logic Array (PLA)

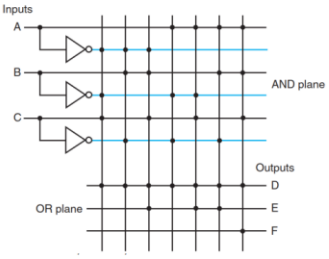
Dato che si può sempre rappresentare una funzione logica come tabella di verità, si può sempre rappresentare anche come somma di prodotto (o prodotto di somme). Questo conduce ad una tecnica di implementazione universale nota come PLA.

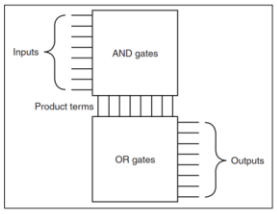


Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

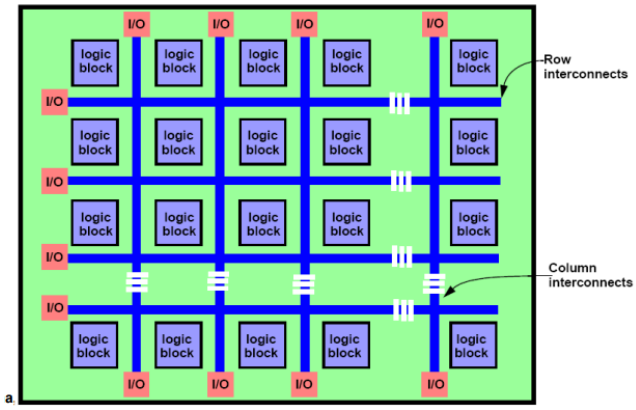
Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1







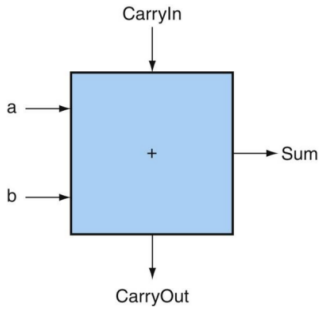
Un PLA complesso (CPLA) non è altro che un insieme di PLD (devices) e una struttura interconnessa. Comparato al CPLA, un FPGA (Field Programmable Gate Array) contiene un numero di piccoli blocchi individuali più grande e una struttura di interconnessione che domina l'intero chip.

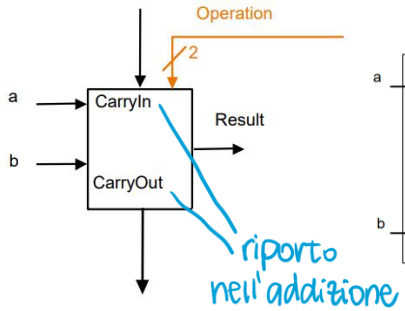


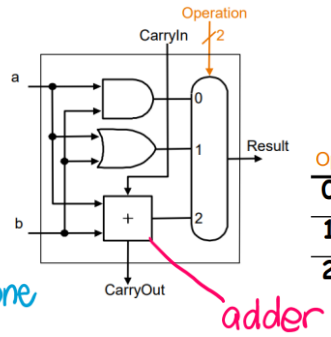
Costruzione di una Arithmetic Logic Unit (ALU)

L'ALU è un componente fondamentale della CPU, e serve ad eseguire operazioni aritmetiche (somma, sottrazione, ecc) e logiche (ad, or, not, ecc) sui dati binari.

Partiamo con una ALU da 1 bit. (https://youtu.be/y9EA-XBbRgg?si=XfJLAR8gr_A1W63J)

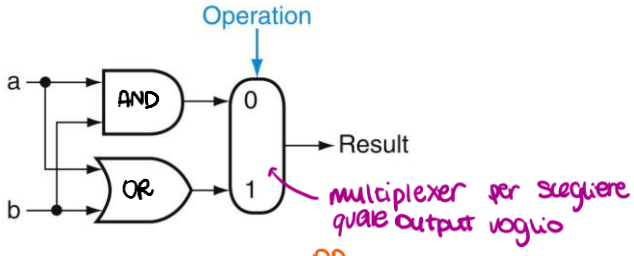




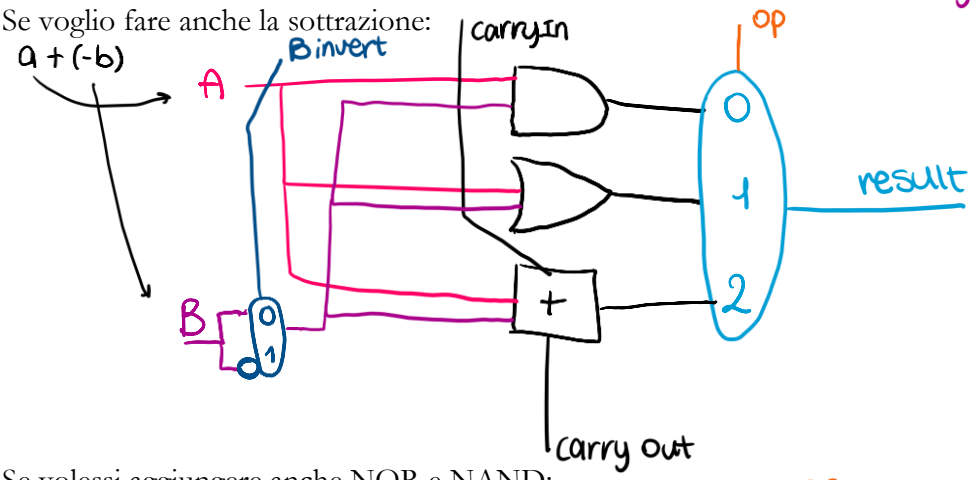


Operation	Function
0	AND
1	OR
2	ADD

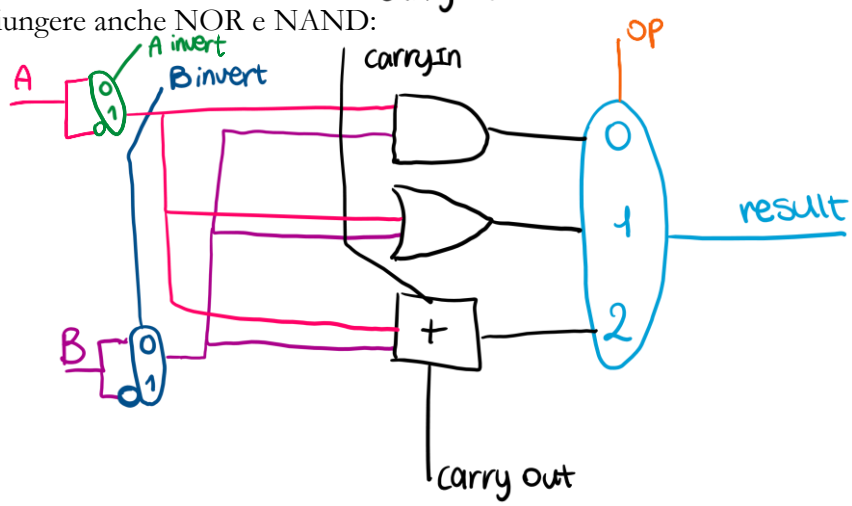
Questa è un'operazione AND e OR



Se voglio fare anche la sottrazione:



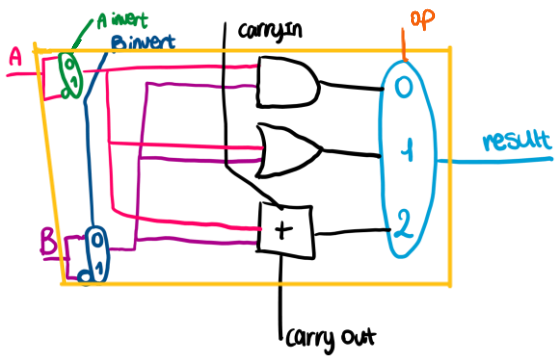
Se volessi aggiungere anche NOR e NAND:

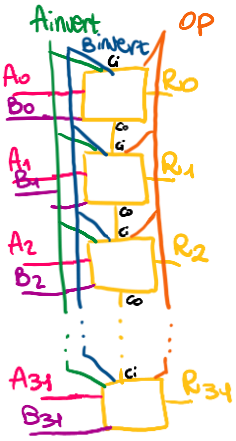


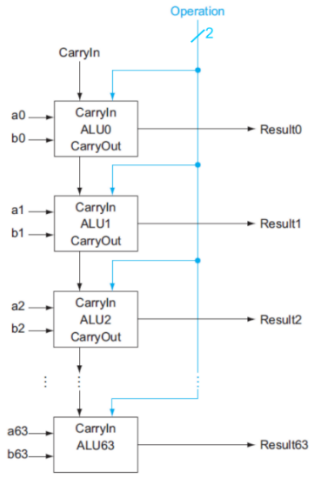
NOR $\rightarrow \neg(A \cup B) = \neg A \wedge \neg B$ NAND $\rightarrow \neg(A \wedge B) = \neg A \vee \neg B$

Un'ALU da **1 bit** può fare operazioni molto semplici: ad esempio, può sommare 1 + 1, risultato 0 con riporto 1: il bit meno significativo, quello a destra, è 0, e il riporto (carry) è 1, quindi va trasportato al bit successivo, creando 10, ossia 2 in binario.
Un'ALU da **32 bit** può sommare due numeri binari di 32 bit.
Un'ALU da **64 bit** è più potente e adatta per elaborare grandi quantità di dati (ad es. numeri a virgola mobile, indirizzi di memoria più grandi, ecc).

Ora vediamo come si costruisce un ALU da 32 bit. (https://youtu.be/IzCIxPCfEP0?si=pu6le1thzh0Fpk_T)
Intanto astraiamo l'ALU da un bit.



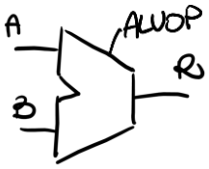




Connetto questa astrazione per 32 volte.

Il carry out di quello sopra diventa il carry in di quello sotto. Per il primo ALU, il carry in è sempre uguale alla linea B invert, almeno per il primo l'ALU (mentre gli altri sono appunto collegati al precedente).

In maniera astratta, un insieme di ALU verrà disegnata così:



Un problema nelle ALU da n bit (32, 64) è che il risutlato finale sarà disponibile dopo che l'ultimo ALU, quindi ALU n – 1, avrà ricevuto il carry.

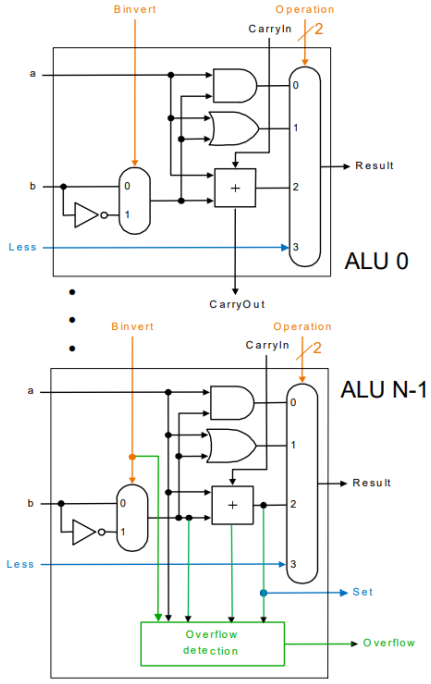
Supporto per la slt
Funzione set-on-less-than (slt) funziona che: se a < b produce 1, altrimenti 0.
Per farlo, basta usare la sottrazione: (a – b) < 0 implica a < b.
Per vedere se (a-b) è negativo basta che verificare se il bit più significativo vale 1.

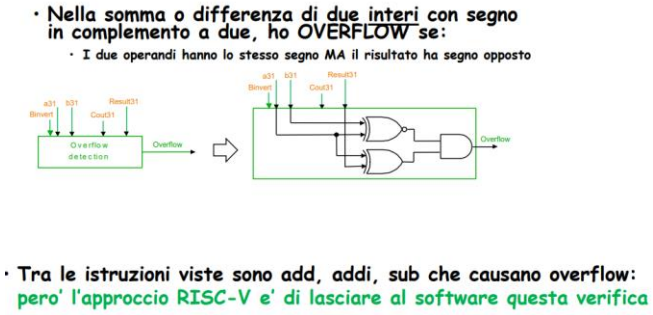
Supporto per il test di uguaglianza
Ci servirà per supportare le condizioni degli statement if/while/for, ossia per decidere se devo eseguire quanto segue o saltare a un altro pezzo di codice.
Se a == b allora produco 1, altrimenti 0.
Per farlo, uso nuovamente la sottrazione: (a – b) = 0 implica a=b.

Supporto alla slt nella ALU

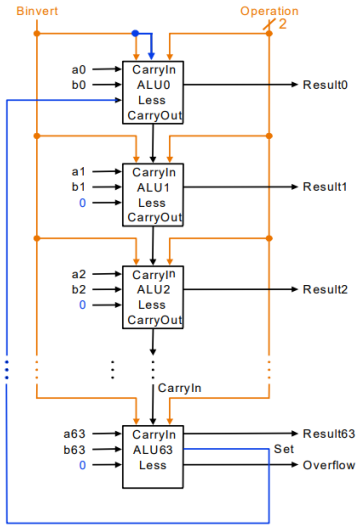
Inserisco il caso "3" al multiplexer, corrispondente a slt. Nelle ALU 0,1, ..., n-1 questo ingresso è collegato a un filo "less" che costituisce un "bypass" ingresso-uscita. Nella ALU n-1 inoltre (bit più significativo), controllo il valore di uscita 'set' che indica come è andato l'esito del confronto a < b.

Mentre per gli interi in complemento a due è semplice sapere quando il risultato è negativo (segnale 'set'), in generale per rivelare l'overflow ho bisogno di una rete logica più complessa:





Se il confronto ha esito positivo il filo less invia sul bit meno significativo il risultato 1, numero negativo.



• Aggiustiamo il $CarryIn_0$:

Operation	Function	CarryIn0	BInvert
0	AND	0	0
1	OR	0	0
2	ADD	0	0
2	SUB	1	1
3	SLT	1	1

→ $CarryIn_0$ = BInvert

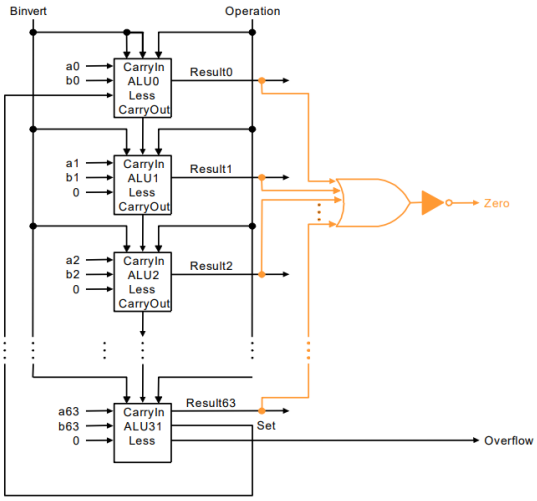
Test per l'uguaglianza (eq)

La rete fornisce il valore 1 quando tutti i bit del risultato sono a zero, in quanto:

(a – b) == 0 → a == b

Chiameremo tale segnale "Zero" (zero detector) e lo useremo per decidere se l'esito del confronto è positivo o meno. Controlla se tutti i bit dell'output della sottrazione sono 0: se A == B, la flag è 1, se no la flag è 0.

Quindi, "Zero" vale 1 quando il risultato è zero! Uso un Not OR.



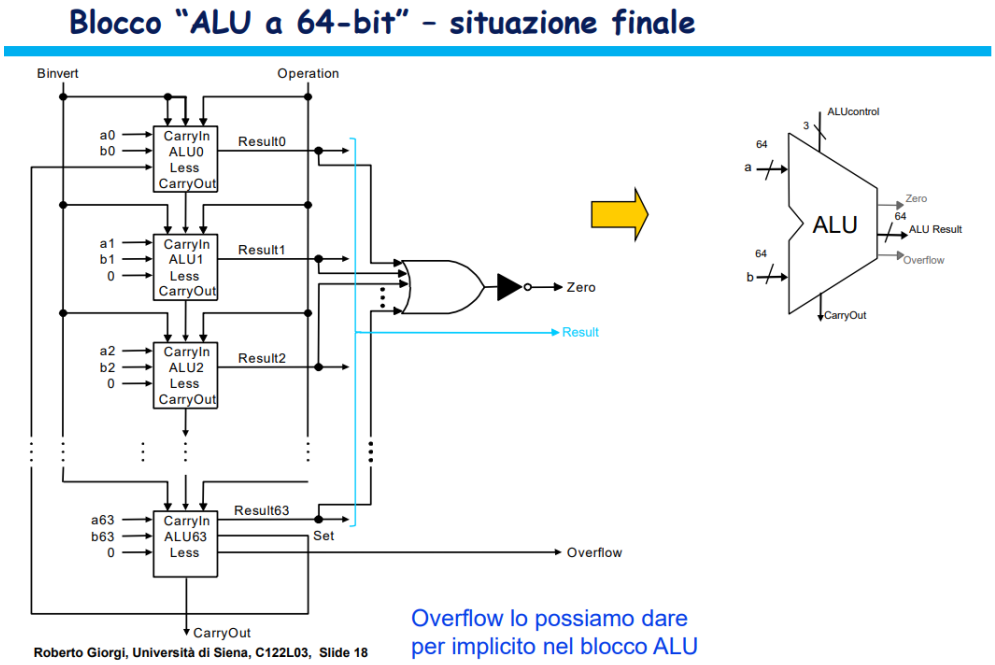
ALUcontrol

Aggiungiamo quindi la funzione EQ che sfrutta la sottrazione e produce un ulteriore filo di uscita dalla ALU (Zero). I fili di ingresso {Operation,Binvert} possono essere raggruppati nei fili chiamati «ALUcontrol» come mostrato in tabella:

Operation	Function	BInvert
0	AND	0
1	OR	0
2	ADD	0
2	SUB	1
3	SLT	1
2	EQ	1

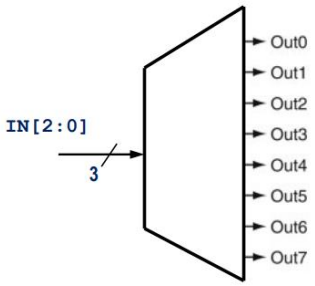
Binvert → $ALUcontrol_2$
$Operation_1$ → $ALUcontrol_1$
$Operation_0$ → $ALUcontrol_0$

ALUcontrol	Function
0 00	AND
0 01	OR
0 10	ADD
1 10	SUB
1 11	SLT
1 10	EQ



Elementi Architetturali Principali: Reti Combinatorie Notevoli

Decoder ed encoder (https://youtu.be/feBvhLFQEDk?si=kk6TMcuFj-lx6eN2**)**



a. A 3-bit decoder

Inputs			Outputs							
IN2	IN1	IN0	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

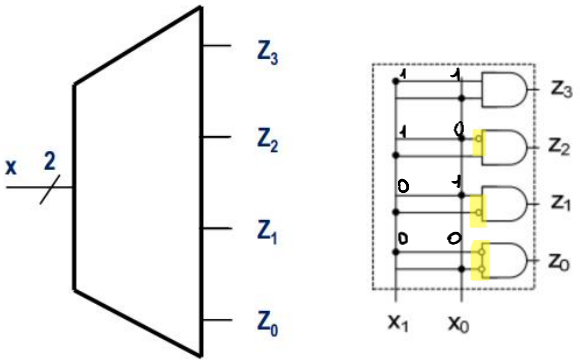
b. The truth table for a 3-bit decoder

Questo esempio è un **decoder** a 3 bit, ossia da 3 a 8.

Un solo output può valere 1, mentre gli altri valgono tutti 0: il decoder attiva solo l'uscita con indice uguale a questo valore (nell'esempio, l'output 101 'accende' l'output 5, perché 101 vuol dire 5 in binario).

In generale, un decoder 'da n a 2^n' ha n input e 2^n output.

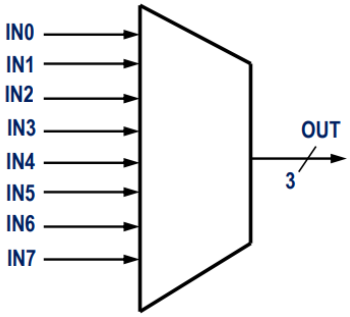
Si può sintetizzare in termini di porte logiche con 2^n porte AND, ognuna delle quali riconosce uno dei 2^n mintermini.



Es. decoder da-2-a-4

L'encoder effettua l'operazione inversa del decoder.

Posto un 1 sul k-esimo dei 2^n ingressi (gli altri sono posti a 0): le n uscite producono i bit corrispondenti alla rappresentazione in base due di k.



IN7	IN6	IN5	IN4	IN3	IN2	IN1	IN0	OUT2	OUT1	OUT0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

OUT0=IN1+IN3+IN5+IN7
OUT1=IN2+IN3+IN6+IN7
OUT2=IN4+IN5+IN6+IN7

Dunque, tale encoder può essere realizzato con 3 porte OR a 4 ingressi.

Nota sull'immagine: quella a destra NON è una tabella della verità; la si può interpretare come tabella di verità osservando che nelle mancanti 244 righe le uscite sono indeterminate (X o "don't care").

- L'encoder effettua l'operazione inversa del decoder
- Posto un 1 sul k-esimo dei 2^n ingressi (gli altri sono posti a 0): le n uscite producono i bit corrispondenti alla rappresentazione in base due di k

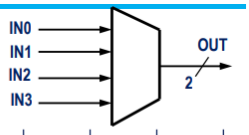


TABELLA DI VERITA' (COMPLETA):

IN3	IN2	IN1	IN0		OUT1	OUT0
0	0	0	1		0	0
0	0	1	0		0	1
0	1	0	0		1	0
1	0	0	0		1	1
0	0	0	0		X	X
0	0	1	1		X	X
0	1	0	1		X	X
0	1	1	0		X	X
0	1	1	1		X	X
1	0	0	1		X	X
1	0	1	0		X	X
1	0	1	1		X	X
1	1	0	0		X	X
1	1	0	1		X	X
1	1	1	0		X	X
1	1	1	1		X	X

IN3,IN2 \ IN1,IN0	00	01	11	10
00	X	0	X	1
01	0	X	X	X
11	X	X	X	X
10	1	X	X	X

OUT0

IN3,IN2 \ IN1,IN0	00	01	11	10
00	X	0	X	0
01	1	X	X	X
11	X	X	X	X
10	1	X	X	X

OUT1

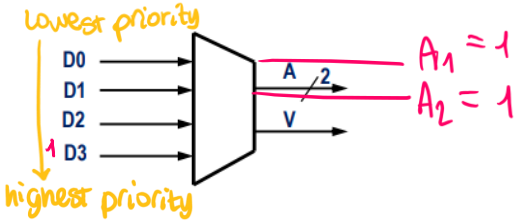
Roberto Giorgi, Università di Siena, C122L03, Slide 6

OUT0=IN1+IN3 OUT1=IN2+IN3

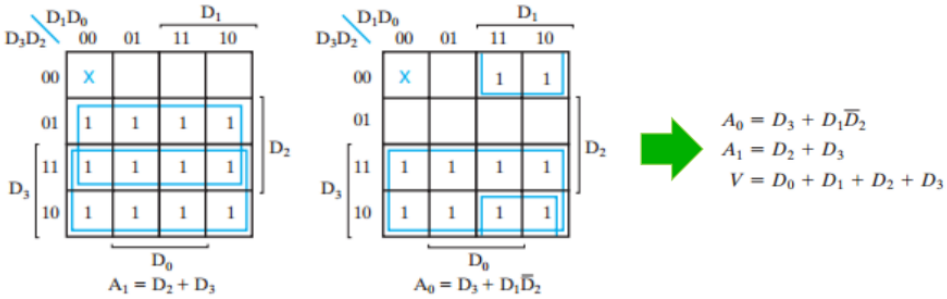
Priority encoder (https://youtu.be/kEj-m3YuGa4?si=jQFtRPOBjRGPV6SO)

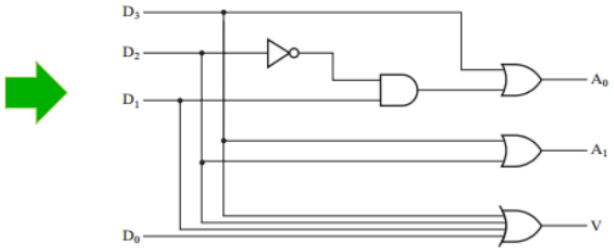
Inputs				Outputs		
D_3	D_2	D_1	D_0	A_1	A_0	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

Tabella di verita' completa

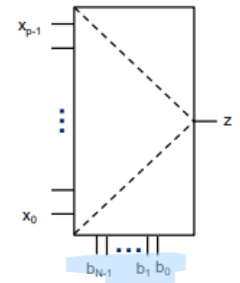


La priorità incrementa scendendo. Quindi, quando D3 vale 1 tutti gli altri bit non contano, non ci interessano e non li controlliamo nemmeno, l'uscita indica la massima priorità $(11)_2$, ovvero 3 (in binario), e quindi A_1 sarà 1 e A_2 sarà 1.
Quando D3 vale 0 ma D2 vale 1 gli altri bit a destra non contano e l'uscita indica priorità 2 e così via…
Il bit V è necessario per indicare se l'uscita è valida, ovvero se almeno un 1 è stato posto in ingresso.

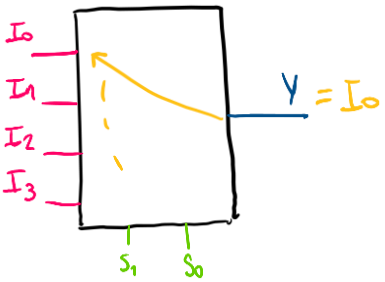




Multiplexer (https://youtu.be/FKvnmxte98A?si=3IRhQn9Z4dyFtr6D)
Il multiplexer seleziona dati presi da vari input e li trasmette in un'unica linea di uscita.



Multiplexer con 2^N ingressi



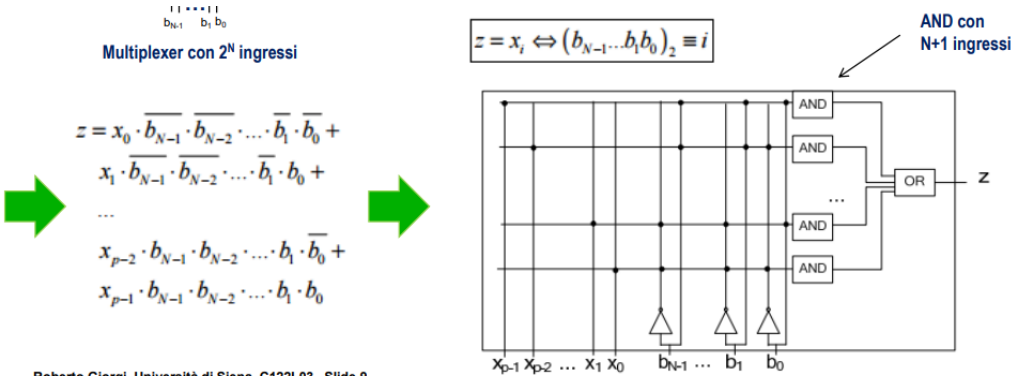
È un data selector, perché seleziona un dato tramite selector line/ selector value lo rende output.
L'output è sempre 1, nell'esempio è 4:1 MUX

Ha 2^N + N ingressi e 1 uscita. Gli N ingressi b sono detti variabili di comando. Specificando B = i viene selezionato uno (quello con indice i) dei 2^N = p ingressi x.

$n = 2^m$ → n° di select lines (ingressi)
↳ n° di input

$\Rightarrow m = \log_2 n$

nell'esempio disegnato $n=4$
$m = \log_2 4 = \log_2 2^2 = 2\log_2 2 = 2 \cdot 1 = 2$
↓ diventa 1 perche' base = argomento

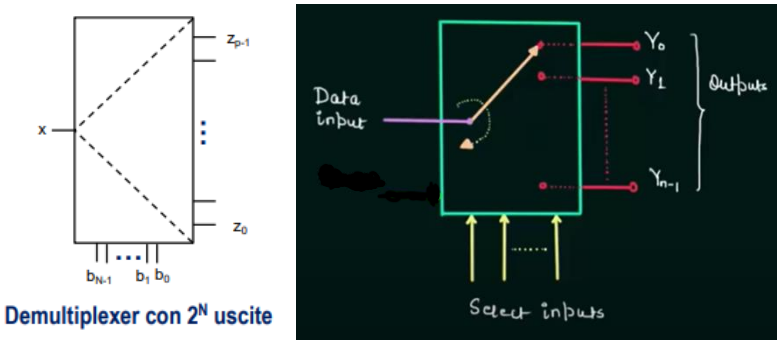


Quindi:

- Un multiplexer **seleziona uno tra diversi ingressi** e lo indirizza su **una sola uscita**.
- Ha **2^n ingressi**, **n bit di selezione** e **1 uscita**.
- I **bit di selezione** (chiamati *select lines*) determinano quale ingresso viene "passato" all'uscita.
- È come un **interruttore controllato digitalmente**.

Ad esempio, un MUX 4:1 ha 4 ingressi (I0, I1, I2, I3), 2 bit di selezione (S0, S1) e 1 uscita (Y).
Se S1S0 = 10, il MUX collega l'ingresso I2 all'uscita Y.

Demultiplexer (https://youtu.be/t3Ed13z9uz8?si=Wy4hJNtrUj4BaEVC**)**
Realizza la funzione inversa del multiplexer, ovvero data un'informazione x, la distribuisce su una delle 2^N uscite possibili.



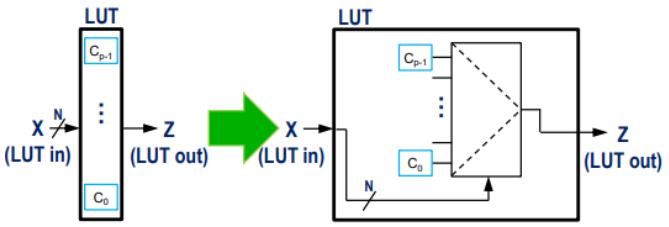
Ha 1+N ingressi e 2^N uscite. Gli N ingressi b sono detti variabili di comando. Specificando B = i viene inviato l'ingresso x su una delle 2^N = p uscite z. Le altre uscite valgono 0.

Quindi:

- Un demultiplexer fa l'opposto del multiplexer: prende **un solo ingresso** e lo indirizza a **uno tra molti output**, in base ai bit di selezione.
- Ha **1 ingresso**, **n bit di selezione** e **2^n uscite**.
- I **bit di selezione** determinano a quale uscita inviare il dato in ingresso.

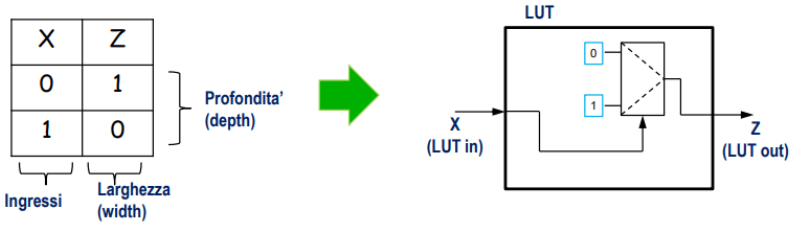
Ad esempio, un DEMUX 1:4 ha: 1 ingresso (D), 2 bit di selezione (S0, S1) e 4 uscite (Y0, Y1, Y2, Y3)
Se S1S0 = 01, il dato in ingresso D va in uscita Y1.

Look up table (LUT)
Una Look-Up Table (LUT) di multiplexer è una tecnica che utilizza multiplexer (MUX) per implementare funzioni logiche predefinite, imitando il comportamento di una tabella di verità.
È un Multiplexer in cui gli ingressi sono costanti. È utile per realizzare una funzione booleana generica.



Il multiplexer ha (con p=2^N) 2^N+ N ingressi e 1 uscita. Gli N ingressi X fungono da variabili di comando. Specificando X = i viene selezionato uno (quello con indice i) dei 2^N = p ingressi (FISSI ovvero COSTANTI). p il **numero di ingressi dati (costanti)** del multiplexer.

Esempio realizzazione della funzione "inverter" tramite LUT



LUT larga 4 e profonda 8

- **I segnali di controllo del MUX sono usati per selezionare le possibili uscite della funzione booleana**

A0	A1	A2	Out0	Out1	Out2	Out3
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

Stored Information

Depth

Address Inputs

Width

Una LUT è una struttura che memorizza i valori di uscita di una funzione logica per ogni possibile combinazione di ingressi. In pratica:

- Gli ingressi vengono interpretati come indirizzi.
- Il valore corrispondente viene "guardato" nella tabella e restituito in uscita.

I MUX possono essere usati per costruire una LUT, sfruttando i loro ingressi selettori per "navigare" tra i valori della tabella.
Quindi: una LUT di multiplexer è un modo per simulare una tabella di verità usando un MUX, dove gli ingressi del MUX contengono i risultati predefiniti della funzione logica e i bit di selezione del MUX rappresentano le variabili logiche.

Circuiti (o reti) sequenziali

Prima facciamo un recap sui circuiti combinatori: sono un tipo di circuito elettronico digitale in cui l'uscita dipende esclusivamente dagli ingressi in un determinato istante di tempo, senza alcuna memoria interna o dipendenza dallo stato precedente del sistema. L'uscita cambia immediatamente quando gli ingressi cambiano e per ogni combinazione di ingressi, l'uscita è sempre la stessa.
Questo non succede però con i circuiti sequenziali.

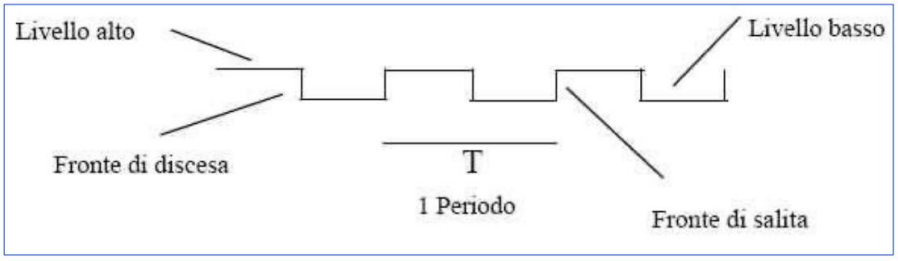
I circuiti sequenziali sono una classe di circuiti elettronici che non dipendono solo dagli ingressi attuali, ma anche dallo stato precedente al circuito. Questo significa che la loro uscita dipende sia dai segnali in ingresso che dalla memoria interna del circuito, che conserva informazioni sul passato. Questi circuiti sono utilizzati per elaborare segnali in base a una sequenza temporale, dove il risultato di una determinata operazione può influenzare le operazioni successive.

Alcuni esempi di circuito sequenziali sono: un semaforo, che deve memorizzare lo stato corrente e passare tra gli stati (verde, giallo, rosso) in sequenza. Un altro esempio è un contatore, che conta numeri partendo da un valore iniziale e incrementando di uno ad ogni ciclo di clock.

Gli elementi che memorizzano uno stato possono essere di tipo:

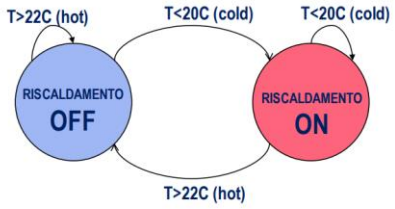
- Asincrono (non hanno bisogno di un segnale di clock)
- Sincrono (richiedono un segnale di clock)

Il clock del processore funge da metronomo per questi cicli di esecuzione. Genera un segnale ritmico, un "ticchettio" regolare, che scandisce il tempo di ogni fase. Più alto è il clock, maggiore è la frequenza con cui il processore può completare un ciclo di esecuzione e, di conseguenza, più istruzioni può elaborare al secondo. La frequenza del clock si misura in Hertz (Hz), che rappresenta il numero di cicli al secondo. Un processore con una frequenza di clock di 3 GHz, ad esempio, può completare 3 miliardi di cicli di esecuzione al secondo, traducendosi in un'enorme capacità di elaborazione.
Il segnale di clock serve per fornire un riferimento temporale ben preciso per effettuare le operazioni. Ad esempio, si può prendere come riferimento il fronte in salita o quello di discesa.



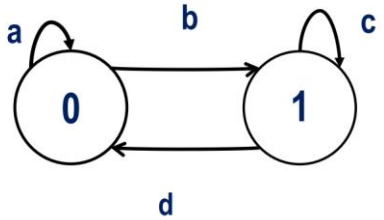
Progettazione di circuiti a strato interno
Il primo passo per la progettazione è comprendere il compito che il circuito deve svolgere. Per questo scopo si possono usare dei metodi di specifica dei requisiti del sistema basati ad esempio sulle macchine a strati finiti (FSM).
Esempio di macchina a Strati Finiti:

Un insieme di stati e di transizioni



Ci si sposta da uno stato all'altro attraverso una transizione (arco nel grafo) se la condizione è soddisfatta

- Un insieme di stati e di transizioni



Ci si sposta da uno stato all'altro attraverso una transizione (arco nel grafo) se la condizione è soddisfatta

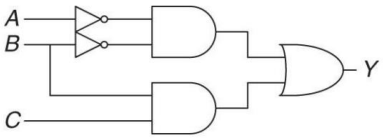
Nel campo dell'architettura dei calcolatori, le FSM sono molto importanti in quanto godono delle proprietà di essere:

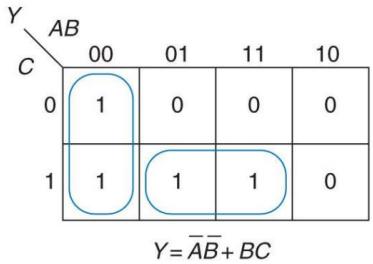
- Complete: tutti gli stati definiscono transizioni per tutti gli input
- Deterministiche: se si presenta un dato ingresso seguo sempre un dato arco

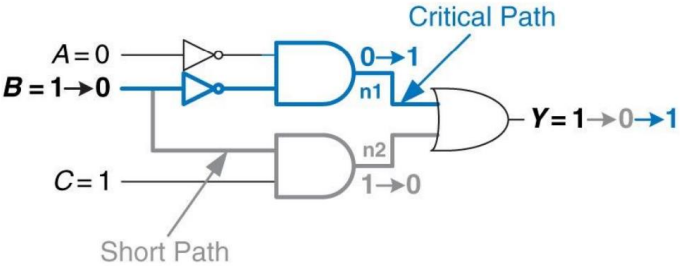
Inoltre, le FSM possono essere implementate fisicamente con circuiti sequenziali tramite alcuni schemi generici (es. modelli di Moore).
I circuiti (o reti) sequenziali possono avvicinarsi al comportamento ideale di una FSM facendo in modo che le transizioni fra uno stato ed il successivo avvengano in intervalli di tempo molto limitati, utilizzando particolari accorgimenti costruttivi o effettuando le transizioni su un fronte del clock.

Ritardi (Alee/Glitch/Hazard)

Può accadere che, variando un ingresso, si creino variazioni multiple dell'uscita.





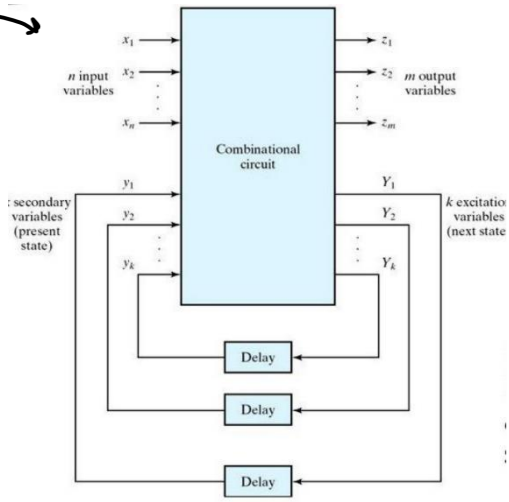


Teoria delle reti sequenziali

Negli anni 40-50 si è consolidata una teoria degli Automi (Chomsky) e in particolare delle cosiddette Macchine a Stati Finiti (FSM). Nel 1955 Mealy e nel 1956 Moore propongono dei modelli per implementare una FSM. L'idea era che una FSM potesse essere realizzata ricorrendo solo ad elementi di memoria e logica combinatoria opportunamente collegata; pertanto, è nata una teoria di come realizzare delle reti sequenziali che implementino FSM il cui comportamento è descritto o informalmente (tramite il linguaggio naturale) o formalmente (tramite un diagramma a stati).

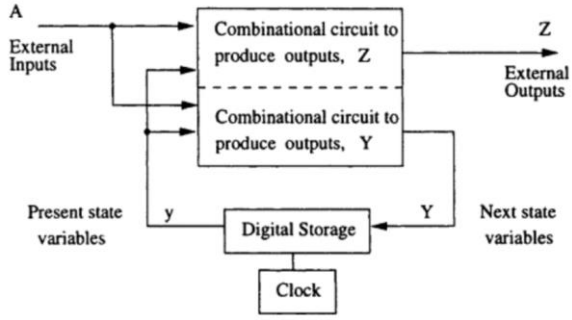
Reti sequenziali asincrone

Alcuni o tutti gli output non variano rispetto a transizioni del segnale di clock (cioè non variano tutti assieme).



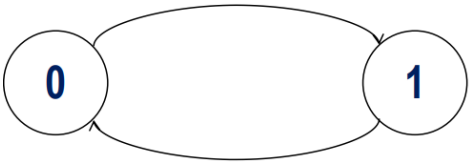
Reti sequenziali sincrone

Tutti gli output variano rispetto a transizioni del segnale di clock (cioè variano tutti assieme)



L'elemento di memoria: memorizziamo 1 bit

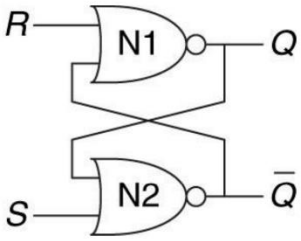
La FSM più semplice che si possa immaginare (e che varia il proprio stato) è una rete a due soli stati. Tale rete non rappresenta altro che un bit di informazione e quindi permette di implementare un singolo elemento di memoria.



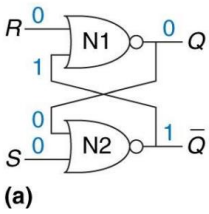
Latch SR (https://youtu.be/kt8d3CYWGH4?si=MBfLATCW0W4yQttM)

L'esempio di circuito di memoria ad 1 bit più semplice è il Latch SR: utilizza porte NOR e non è guidato da un segnale di «clock» (logica asincrona)

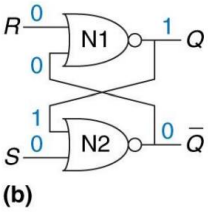
L'uscita dipende dai segnali presenti sugli ingressi e dai segnali precedentemente presentati in ingresso. (asincrono)



Come si vede da questo esempio: per SR=00, sia Q=0 (a) che Q=1 (b) sono situazioni possibili.



(a)



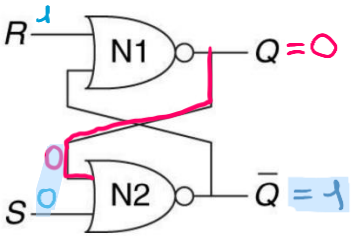
(b)

S	R	Q	/Q
0	0	Q	/Q
0	1	0	1
1	0	1	0
1	1	---	---

SET RESET

Si dice che "conserva"

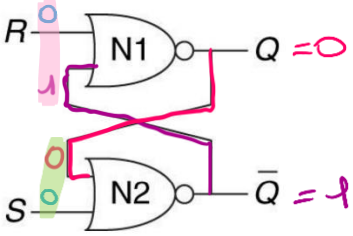
Tabella di Verità



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

case S=0, R=1

Passando al caso successivo, Q e $\bar{Q}$ non cambiano.



A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

case S=0, R=0

L'output, lo stored data, rimane sempre lo stesso, rimane in memory.

Non si può usare S = 1 e R = 1 perché Q e $\bar{Q}$ sarebbero entrambi 0.

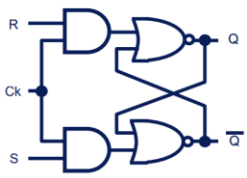
S	R	Q	$\bar{Q}$
0	0	Q	$\bar{Q}$ (memoria)
0	1	0	1
1	0	1	0
1	1	non utilizzabile	

S-R clocked (sincrono)

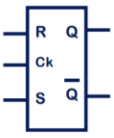
Il Latch SR è sempre sensibile ad ogni variazione di S e di R: potrebbero cambiare anche non intenzionalmente. Per fare in modo che S e R vengano ascoltati solo in determinati istanti si può usare un segnale di abilitazione Ck (clock). L'SR-clocked, sente S e R solo quando il segnale Ck (clock) vale 1.

Ck	S	R	Q	/Q
1	0	0	Q	/Q
1	0	1	0	1
1	1	0	1	0
1	1	1	---	---
0	X	X	Q	/Q

Tabella di Verità



Schema logico



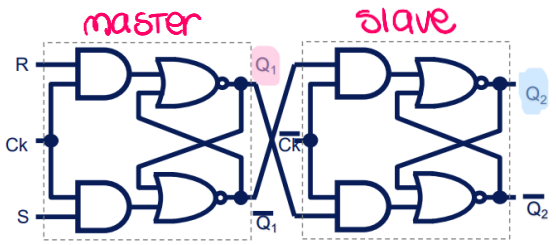
Simbolo

S-R edge triggered Flip-Flop

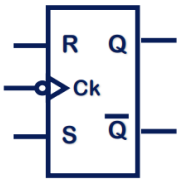
In questo caso vogliamo sensibilità solo sul fronte in discesa (falling edge) del clock.

Ck	S	R	Q	/Q
↓	0	0	Q	/Q
↓	0	1	0	1
↓	1	0	1	0
↓	1	1	---	---
0	X	X	Q	/Q
1	X	X	Q	/Q
↑	X	X	Q	/Q

Tabella di Verità



Schema logico



Simbolo

Sensibile sul fronte in discesa

Utilizza la tecnica Master-Slave:

- Master legge gli ingressi quando clock Ck=1
- Slave aggiorna l'uscita solo quando Ck passa da 1 a 0

Quando **Ck = 1**: Il Master è attivo (trasparente), e legge i valori S e R e li memorizza in Q_1. Slave è bloccato (opaco) e mantiene valore precedente in Q_2.

Quando **Ck = 0**: Master è opaco e mantiene valore precedente su Q_1. Slave si attiva, diventa trasparente, e copia in uscita Q_2 il valore memorizzato nel Q_1.

Fronte di discesa (Ck = 1 → 0): Master diventa opaco, smette di leggere S e R, lo Slave si attiva e prende in carico l'output, copiando il valore da Master e trasmette il dato che aveva memorizzato quando Ck = 1. Quindi in questo istante cambia l'uscita di Q (= Q_2), perché è edge-triggered sul fronte di discesa.

Fronte di salita (Ck = 0 → 1): Master diventa attivo, pronto a leggere nuovi input. Slave opaco e quindi il dato memorizzato rimane invariato. L'uscita, quindi, non cambia perché solo il Master legge, ma è lo Slave che controlla l'uscita, e in questo momento rimane fermo.

D-Latch (Delay/Data Latch) (serratura/chiavistello) o D-Trasparent

IL latch SR è problematico perché si comporta in modo impredicibile quando S R = 1 1

Il D-latch definisce separatamente cosa cambiare e quando cambiarlo. Quindi, D ti dice che valore vuoi che sia in uscita (0 o 1) e Ck ti dice quando lo stato deve essere cambiato.

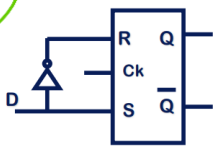
Ha due ingressi (D e Ck):

- D rappresenta il dato (indica COSA verrà memorizzato)
- Ck è il clock (indica QUANDO memorizzare)

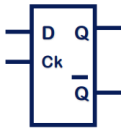
D	Ck	Q
X	0	Q
0	1	0
1	1	1

Tabella di Verità

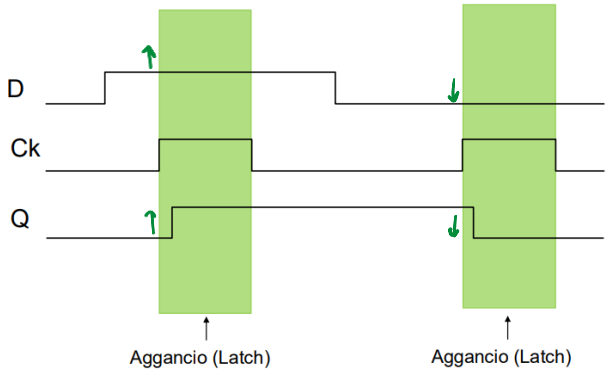
se Ck=0, Q rimane com'era



Schema logico



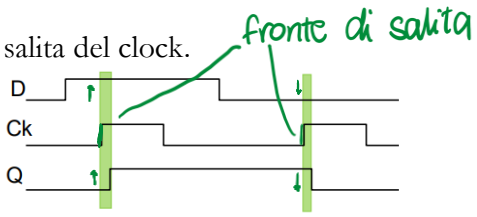
Simbolo del D-Latch



il grafico mostra perché viene anche chiamato transparent:
Le aree verdi indicano quando Ck = 1.
In quelle zone, Q **segue esattamente** D (dato da memorizzare)
Quando Ck = 0, anche se D cambia, Q rimane bloccato al valore che aveva.

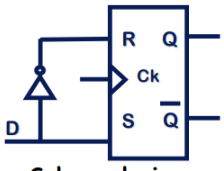
D-edge triggered

L'uscita cambia solo sul fronte in salita del clock.



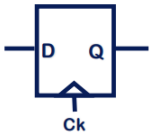
D	Ck	Q
X	1	Q
X	0	Q
X	↘	Q
0	↗	0
1	↗	1

Tabella di Verità



Schema logico

Layout simbolico: 44 transistors (42 per SR-ET + 2 per NOT)



Simbolo del D-Edge-Triggered

Un D-edge triggered è un flip-flop che ha due ingressi: D (dato da memorizzare) e Ck (clock). Ha una sola uscita, ossia Q. Quello che fa è memorizzare il valore di D solo nel preciso istante in cui Ck passa da 0 a 1 (cioè sul fronte di salita del clock).

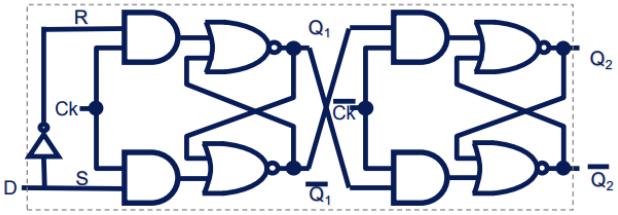
La tabella di verità esprime come l'uscita Q cambia solo all'istante in Ck sale da 0 a 1, e prende il valore di D in quel preciso istante.
Lo schema logico è formato da un NOT per trasformare D in R e S, un latch SR edge-triggered e il simbolo a triangolo sul clock, che significa che il trigger è sul fronte di salita.
Il simbolo del D-edge-triggered ha un triangolino sul Ck per dire che è sensibile al fronte di salita, un solo ingresso D e un'uscita Q.

Flip Flop D negative edge triggered

In questo caso vogliamo sensibilità solo sul fronte in discesa.

D	Ck	Q
X	1	Q
X	0	Q
X	↗	Q
0	↘	0
1	↘	1

Tabella di Verità



Simbolo

Un Flip-Flop D a fronte di discesa è un flip flop che memorizza il dato sul fronte di discesa del clock (quando passa da 1 a 0). È un circuito sequenziale sincrono che memorizza un bit (D) e aggiorna l'uscita Q solo nel momento in cui Ck scende da 1 a 0 (falling edge).

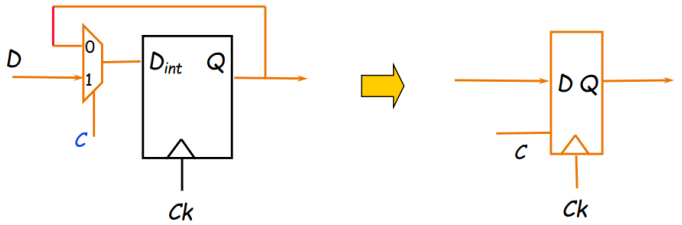
La tabella di verità ci spiega che sul fronte di discesa, e solo su quel fronte, Q prende il valore di D in quell'istante.
Lo schema logico è un flip-flop Master-Slave (che freaky). Il Master è attivo quando Ck = 1, e memorizza D in Q_1. Lo Slave è attivo quando Ck = 0, e prende Q_1 e lo trasmette a Q_2, che è l'uscita finale Q. Quindi, quando Ck passa da 1 a 0, il Master si blocca e si attiva lo Slave, copiando il valore memorizzato: è in quel momento che l'uscita Q si aggiorna.
Il simbolo per rappresentare il negative edge-triggered è il triangolo con pallino davanti a Ck, che indica appunto che è edge triggered e il pallino nega il segnale, per rappresentare il fronte di discesa.

Confronto Latch vs. Flip-flops
Nel latch, l'uscita cambia non appena cambia l'ingresso e il clock ha valore alto.
Nel Flip-flop: l'uscita cambia solo su un fronte del clock (metodologia edge-triggered)
Gli elementi in comune sono:

- l'uscita è pari al valore memorizzato nell'elemento (non è necessario fare azioni particolari per leggere il valore)
- il cambiamento dell'uscita avviene rispetto a un segnale di clock

Nota: nella logica sincrona si definisce l'istante in cui un valore è pronto per essere letto sugli ingressi del flip-flop: non si vuole leggere un dato D mentre la rete che genera D sta ancora "calcolandone" il valore.

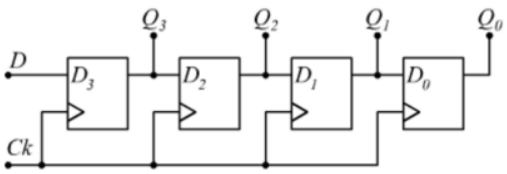
Cella di registro
Una cella di memoria da un bit usa un D-Latch e un multiplexer, il cui controllo si chiama LE (Latch Enable), WE (Write Enable) o C (control) (sono tutti la stessa cosa)

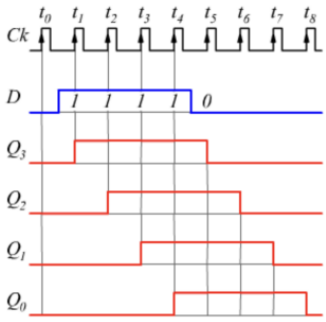


Simbolo della cella di registro da 1 bit

Il control decide quando effettivamente aggiornare quel bit, quindi per rendere questa cella scrivibile solo quando serve, si aggiunge un multiplexer davanti all'ingresso di D per far scegliere al MUX tra il dato in ingresso D e il vecchio valore già memorizzato (Q). Quindi, da Q rientra nel MUX come secondo ingresso e il controllo C/LE/WE comanda il MUX per scegliere se aggiornare o meno.
Questo è la base delle celle di memoria (register file, RAM, etc).

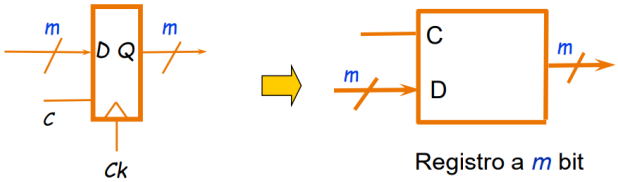
Registro con input in serie e output in parallelo





Ad ogni fronte di salita del clock il dato presente in D_i connesso all'uscita Q_i viene trasferito all'uscita Q_{i-1}. Se ad esempio il registro si trova nello stato $Q_3Q_2Q_1Q_0$ = 0000 con $D = D_3 = 1$, dopo la prima transizione di clock, sulle uscite si avrà la configurazione $Q_3Q_2Q_1Q_0$ = 1000; dopo la seconda transizione di clock si avrebbe $Q_3Q_2Q_1Q_0$ = 1100, alla terza 1110 e alla quarta 1111. Quindi, dopo quattro impulsi di clock, il dato D=1 risulta caricato in tutti gli elementi del registro.

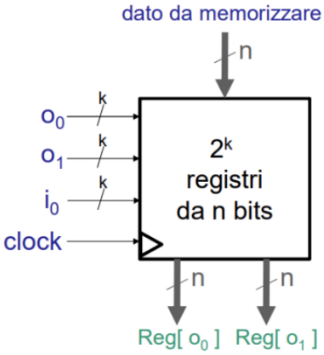
Registro a m bit

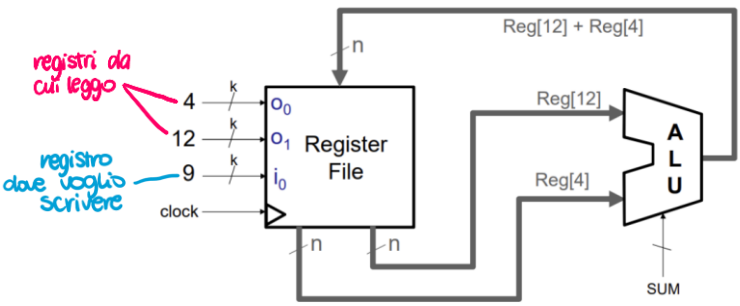


È un registro che memorizza m bit contemporaneamente. Il registro legge il valore su D e lo memorizza in Q solo se C è attivo e al fonte di salita del Ck. (nell'ultima rappresentazione si intende implicitamente che sia presente un segnale Ck per il clock).

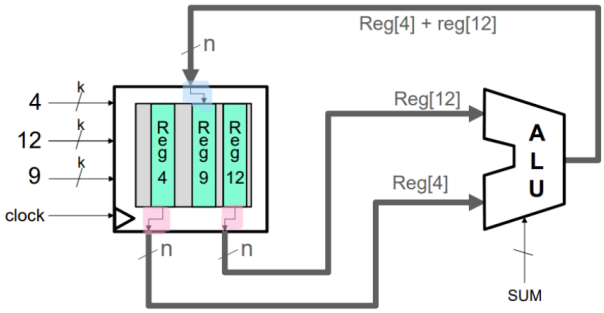
I **Register file** sono composti da n registri di questo tipo. Occorre della logica di scrittura e di lettura. Ad esempio, un register file a 2 uscite e un'entrata, per m = 2^k a n bit

- contiene m = 2^k registri da n bit ciascuno, numerati da 0 a 2^k – 1
- in output restituisce il valore attuale di due dei registri, specificati dagli indici o_0 e o_1
- in input, prende anche la parola di n bit che verrà memorizzata nel registro di indice i_0 (quando il clock è al fronte di salita)
- il clock per la sincronizzazione





Reg[9] ← Reg[12] + Reg[4]



Reg[9] ← Reg[12] + Reg[4]

Reg[12] e Reg[4] vengono letti e inviati all'ALU. L'ALU esegue l'op somma (SUM) e il risultato della somma viene poi memorizzato in Reg[9] SOLO AL PROSSIMO FRONTE DEL CLOCK.

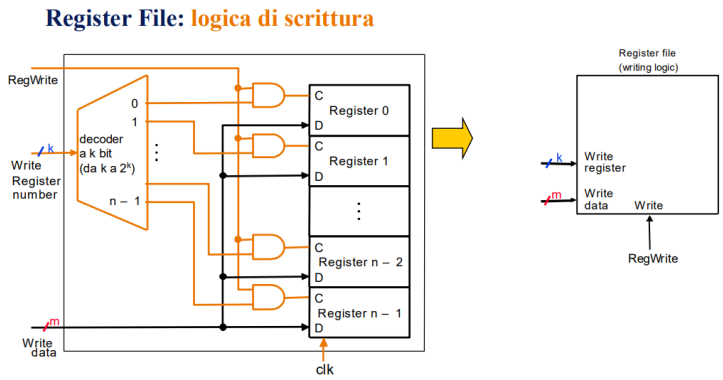
In un linguaggio ad alto livello (es. C o Java) scriveremo:

```
int tot_animali, n_mucche, n_pecore;
… /* altro codice */
tot_animali = n_mucche + n_pecore;
```

Il compilatore potrebbe (per es) decidere che le variabili con gli identificatori «tot_animali», «n_mucche», «n_pecore» vadano memorizzati nei registri n. 10, 11, e 12 rispettivamente, e tradurre il comando di assegnamento come:

```
ADD R10, R11, R12
```

Logica di scrittura



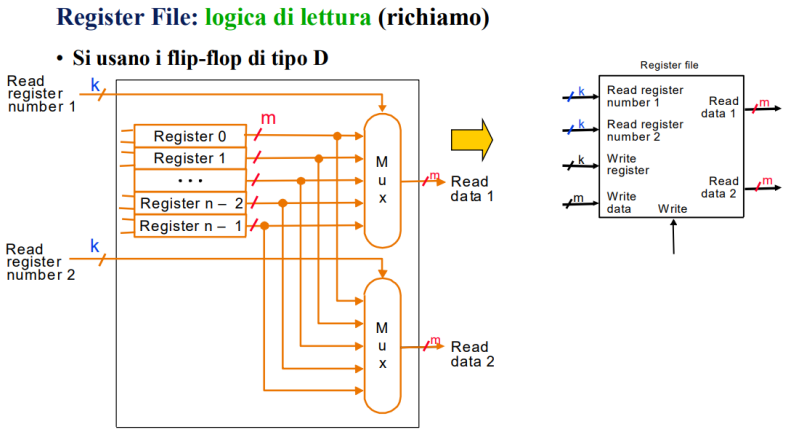
n = numero di registri disponibili (e.g. 32)

k = $\log_2(n)$ (e.g. 5)

m = larghezza dei registri (e.g. 64)

La scrittura in un register file implica inserire un dato m-bit all'interno di un registro specifico, al prossimo fronte del clock.
Abbiamo RegWrite che è un segnale che abilita la scrittura. Se 0, nessun registro verrà aggiornato.
Write Register Number (da k bit) codifica quale registro vogliamo aggiornare. Viene passato ad un decoder che da k bit di ingresso, crea n numero di linee in uscita. Il decoder trasforma il numero del registro in binario in una linea di selezione attiva per un solo registro tra tutti. Ad esempio, se Write Register = 5, solo la sesta uscita del decoder sarà 1, mentre le altre 0.
Anche se il decoder seleziona il registro giusto, la scrittura avverrà solo se RegWrite = 1. Questo viene rappresentato con un AND tra registro del decoder e RegWrite.
Ogni registro è composto da m flip-flop D in parallelo. La scrittura nel registro selezionato avviene sul fronte di salita del clock.
Quindi, se RegWrite = 1, allora il dato Write Data (m bit) viene scritto nel registro indicato da Write Register, al prossimo fronte di clock.

Logica di lettura

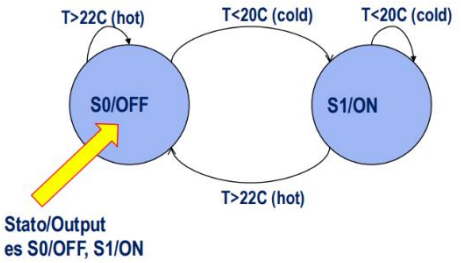


La lettura di un register file implica ottenere il contenuto di uno o più registri.
Lo schema ha due ingressi: Read Register number 1 e 2 (da k bit). Questi specificano quali due registri vogliamo leggere.
Ha due multiplexer, ognuno con n ingressi a m bit (cioè tutti i registri). Il MUX seleziona uno dei registri in base al numero fornito in input. Il primo MUX genera Read Data 1 e il secondo Read Data 2. I multiplexer leggono contemporaneamente 2 registri selezionati da Read Register 1 e Read Register 2, e ne restituiscono i valori su Read Data 1 e Read Data 2.

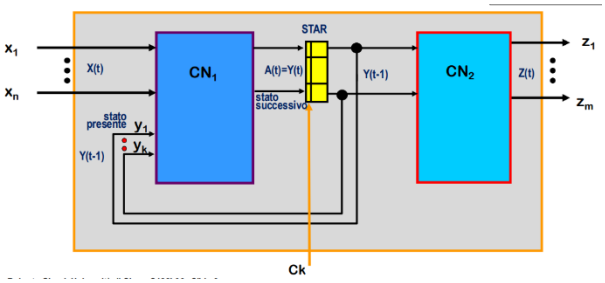
Specifica di una FSM per implementazione con Moore

Vogliamo costruire un circuito sequenziale, ovvero un circuito in cui l'uscita dipende anche dallo stato precedente, non solo dagli input attuali.

In questo esempio, ci sono due strati: s0 (OFF) che esprime che il riscaldamento è spento, e s1 (ON), che esprime che il riscaldamento è acceso. Ci sono due transizioni: si passa da OFF a ON se la temperatura T < 20C, e si passa da ON a OFF se T > 22C.



L'architettura di riferimento è la macchina di Moore, dove l'uscita dipende solo dallo stato corrente e non dagli input.



- $Z(t) = f(Y(t-1))$: in Moore, l'uscita (Z) dipende direttamente solo dallo stato (Y)
- $Y(t) = g(X(t), Y(t-1))$: lo stato futuro dipende dall'input e dallo stato presente. Y(t – 1) è lo stato presente e Y(t) lo stato successivo.
- L'output cambia solo quando cambia lo stato (quindi con il clock)

Per costruire un circuito sequenziale partendo dall'esempio dobbiamo:

1. Fare la tabella delle transizioni

STATO PRESENTE (Y) \ cold,hot (X)	00	01	11	10
S0	S0	S0	-	S1
S1	S1	S0	-	S1

STATO SUCCESSIVO (S)

2. Fare la codifica degli strati con numeri binari

Stato	S=Y
S0	0
S1	1

3. Sintesi della mappa (di karnaugh) per s

CN1

y \ cold/hot	00	01	11	10
0	0	0	-	1
1	1	0	-	1

s

$s = cold + \overline{hot} * y$

4. Sintesi delle uscite z

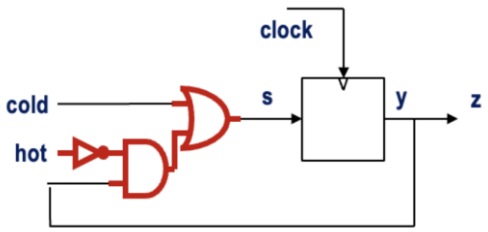
CN2

y	0	1
	0	1

z

$z = y$

Circuito sequenziale risultante:



Il processore

L'architettura di un computer è un insieme di istruzioni (linguaggio) e un'organizzazione interna (registri, memorie). Un'architettura non definisce l'implementazione hardware ma ne descrive le funzionalità.
Una **microarchitettura** definisce le connessioni tra registri e memorie, quante e quali ALU, e altri componenti che formano un microprocessore (cache, ecc).

I 5 passi di una implementazione base:

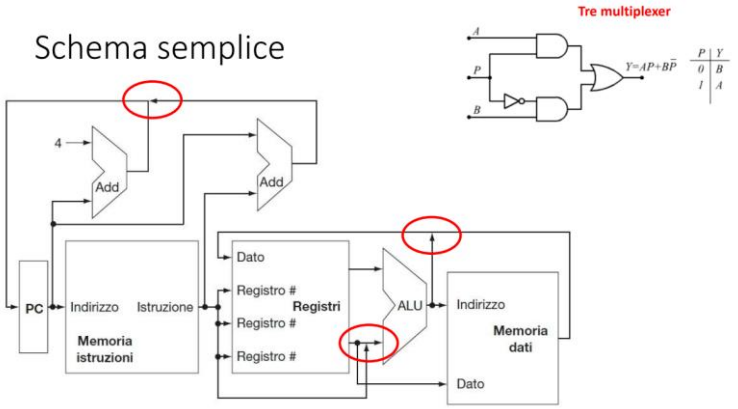
- **Instruction fetch:** prelevo la prossima istruzione da eseguire ed incremento il PC (Program Counter)
- **Instruction decode:** decodifico l'istruzione e leggo gli eventuali operandi dal register file
- **Execution:** uso la ALU per il calcolo. Un valore di un'operazione Aritmetico Logica o un indirizzo di memoria per load/store
- **Memory:** accedo alla memoria dati (RAM) se devo fare load/store
- **Write back:** scrivo il risultato (indietro) nel register file se proviene dall'ALU (cioè se l'istruzione era di tipo aritmetico/logico e non uno store)

Uno stored-program computer (programma memorizzato) è un computer che memorizza le istruzioni nella sua memoria per permettere di fare un insieme di task in sequenza intermittentemente. Introdotto da Van Neumann.

Implementazione base

Prendiamo in considerazione solo queste istruzioni:

- Accesso alla memoria: lw (load word, carica un valore dalla RAM in un registro) e sw (store word, salva un valore da un registro alla RAM)
- Istruzioni ALU (aritmetico-logiche): add, sub, and, or
- Branch condizionato: beq (branch if equal, salta ad un'altra istruzione se due registri sono uguali)



Questo è uno schema semplice per capire come si costruisce un processore a singolo ciclo.
I componenti principali sono:

- Il program counter (PC), che tiene traccia dell'indirizzo dell'istruzione corrente
- La memoria delle istruzioni, da dove si leggono le istruzioni
- Il registrer file, che contiene tutti i registri
- L'ALU, che esegue i calcoli (somma, sottrazione, prodotto, ecc)
- La memoria dati, per lw e sw (quindi la RAM)
- ADD, una somma costante per il program counter (PC + 4 per passare alla prossima istruzione, lunga 4 byte), e il secondo ADD che calcola l'indirizzo del salto per le istruzioni di tipo beq

- 3 multiplexer (i cerchi rossi), che sono:
    - Quello in altro a sinistra, che decide se il PC va incrementato di 4 oppure modificato (salto condizionato, beq)
    - Quello prima dell'ALU che decide se l'input per l'ALU viene da un registro o da un valore immediato (addi)
    - Quello prima della scrittura del dato nei registri (tra memoria dati e registri), che decide se scrivere nei registri il risultato dell'ALU oppure il valore caricato dalla memoria (lw)

Questo è invece lo schema completo, con l'aggiunta di:
- Un controllo
- I vari multiplexer
- Il branch: se due registri sono uguali, cambia il PC. In confronto viene fatto con un confronto nella ALU (zero)
- Segnali memRead e MemWrite, che controllano l'accesso in lettura/scrittura alla memoria dati

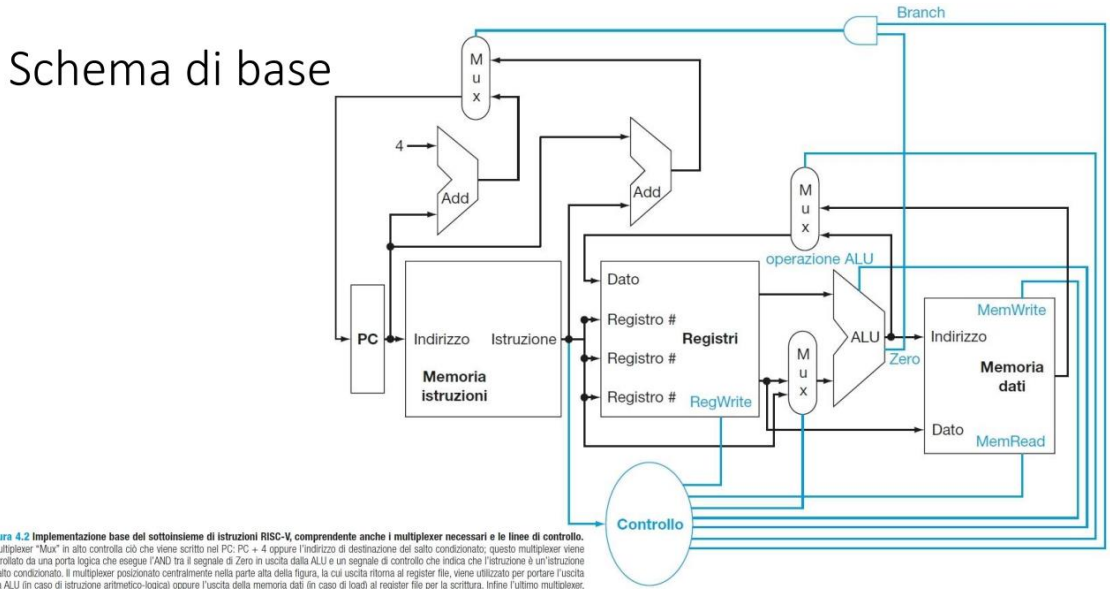
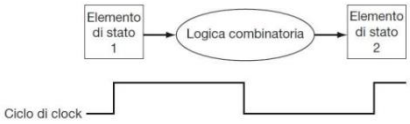


Figura 4.2 **Implementazione base del sottoinsieme di istruzioni RISC-V, comprendente anche i multiplexer necessari e le linee di controllo.** Il multiplexer "Mux" in alto controlla ciò che viene scritto nel PC: PC + 4 oppure l'indirizzo di destinazione del salto condizionato; questo multiplexer viene controllato da una porta logica che esegue l'AND tra il segnale di Zero in uscita dalla ALU e un segnale di controllo che indica che l'istruzione è un'istruzione di salto condizionato. Il multiplexer posizionato centralmente nella parte alta della figura, la cui uscita ritorna al register file, viene utilizzato per portare l'uscita della ALU (in caso di istruzione aritmetico-logicai oppure l'uscita della memoria dati (in caso di loadi al register file per la scrittura. Infine l'ultimo multiplexer.

Temporizzazione

I valori da inserire in un elemento di stato provengono tipicamente da una rete combinatoria come elaborazione di altri dati. Bisogna quindi aspettare che il dato prodotto dalla rete combinatorio sia valido e che il suo valore non stia variando. Si usa quindi una temporizzazione edge-triggered, dove il valore di un elemento di stato è aggiornato solo in corrispondenza di un fronte (salita). Il tempo necessario per la propagazione dei segnali dall'elemento 1 a 2 determina la durata minima del tempo in clock.



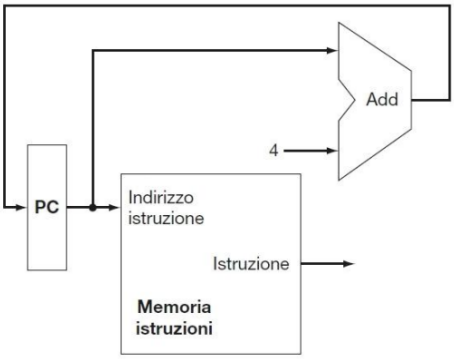
Elementi della CPU

Sono necessari due elementi di stato per memorizzare e accedere alle istruzioni, e serve un sommatore per calcolare l'indirizzo dell'istruzione successiva.

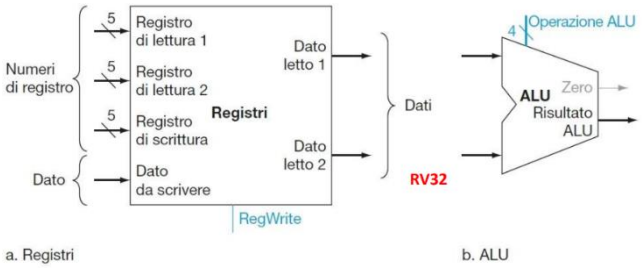
Il **program counter (PC)** è un registro da 32 bit (RV32) che conserva l'indirizzo di memoria della prossima istruzione da eseguire. Viene aggiornato al termine di ogni ciclo clock, quindi non richiede un segnale di controllo esplicito per la scrittura.

La **memoria delle istruzioni** fornisce solamente un accesso in lettura, perché essa non viene scritta dall'unità di elaborazione. Per questo, viene trattata come un circuito logico combinatorio

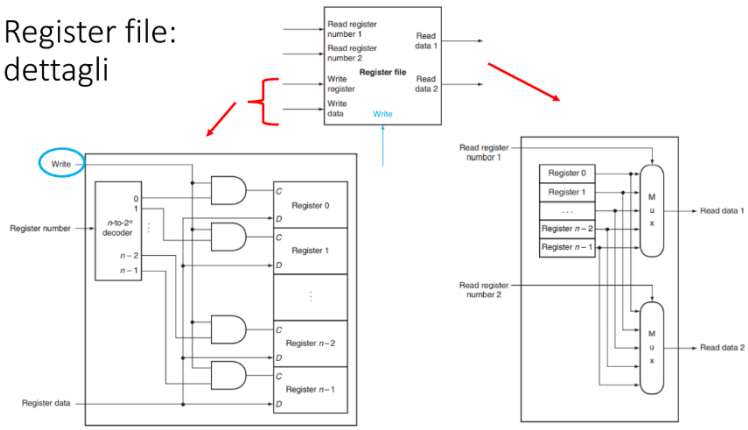
L'**ADD** è una ALU configurata in modo da eseguire sempre la somma dei suoi due ingressi a 32 bit, e fornisce in uscita il risultato della somma, ossia l'indirizzo della prossima istruzione (PC + 4).



I due elementi richiesti per implementare le operazioni di tipo R sono il register file e la ALU.

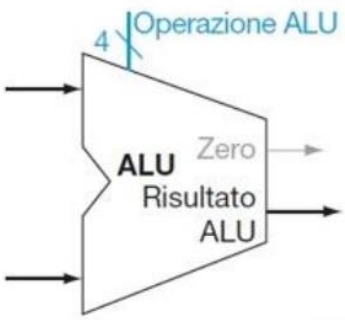


Il **register file** prende in input due registri di lettura, un registro di scrittura, un dato da scrivere e un RegWrite. I due registri di lettura non vengono modificati, vengono soltanto letti e mandati direttamente in output. Il registro di scrittura viene modificato in base al dato da scrivere soltanto quando il RegWrite assume valore 1, altrimenti non modifica il registro (quando assume valore 0). La scrittura avviene sul fronte attivo del clock, quindi tutti gli ingressi interessati devono essere validi sul fronte attivo del clock. La lettura e scrittura viene eseguita nello stesso ciclo di clock.

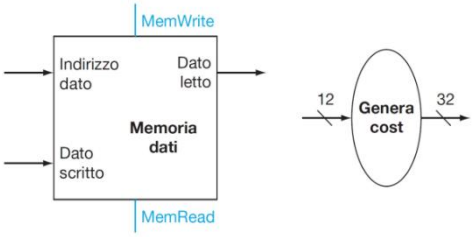


La **ALU** prende in input i valori dei registri di lettura del register file e ci fa delle operazioni. In base all'ingresso di controllo alla ALU, le operazioni saranno somma (0010), sottrazione (0110), and (0000) e or (0001).

C'è anche un segnale "Zero" (zero detector), che useremo per decidere se l'esito del confronto in una beq è positivo o meno. Controlla se tutti i bit dell'output della sottrazione sono 0, e se A == B, la flag è 1, se no la flag è 0. Quindi, "Zero" vale 1 quando il risultato è zero!



Le due unità necessarie per l'implementazione delle istruzioni di load e di store sono un'unità di memoria dati e un'unità di estensione del segno.



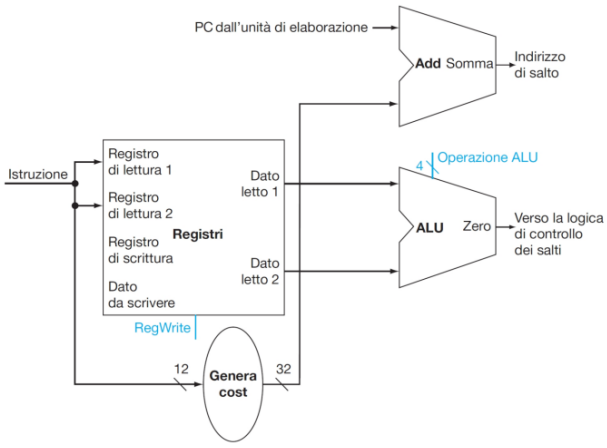
L'**unità di memoria** è un elemento di stato che ha come ingressi l'indirizzo del dato e il dato da scrivere, e possiede una sola uscita per il dato letto. I segnali di controllo per la lettura e la scrittura sono separati, anche se all'interno di un ciclo di clock solo uno dei due può essere asserito.

- **Indirizzo dato**: indirizzo della locazione di memoria a cui il processore desidera accedere. L'indirizzo di memoria viene solitamente fornito da un registro di processore (PC).
- **Dato scritto**: se l'operazione è una scrittura della memoria (store), l'unità riceve i dati che il processore desidera scrivere nella memoria esterna.
- **MemWrite** e **MemRead**: segnali di controllo che indicano il tipo di operazione di memoria che si desidera eseguire, ad esempio se si tratta di una lettura (load) o una scrittura (store).
- **Dato letto**: se l'operazione è una lettura della memoria (load), l'unità di memoria dati restituisce i dati leti dalla posizione di memoria specificata. Questi dati vengono poi inviati a un registro del processore o a un altro componente del sistema.

L'**unità di estensione del segno** (unità di generazione della costante) prende un numero con meno bit (12) e lo trasforma in un numero con più bit, conservandone il segno. Quindi serve per supportare i numeri negativi.

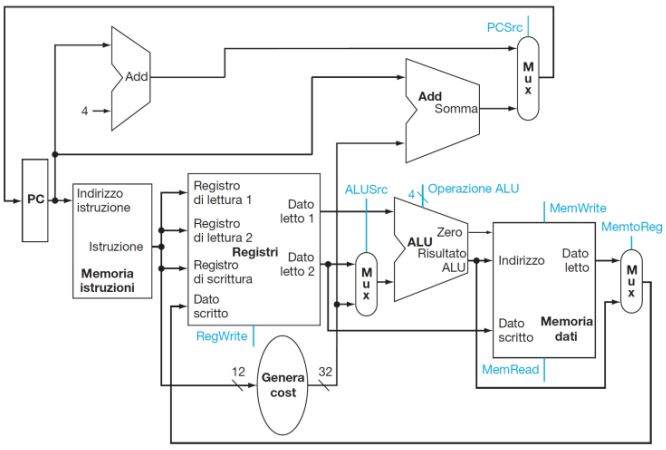
BEQ

L'istruzione entra nel register file. La ALU confronta i valori del Dato Letto 1 e il Dato Letto 2. Fa la sottrazione rs1 – rs2, e il risultato è 0, allora vuol dire che rs1 == rs2, e il segnale mandato dalla ALU è il segnale 'Zero'. L'immediato dell'istruzione entra nel Genera Cost, e questo offset viene sommato al PC tramite il blocco ADD, per ottenere l'indirizzo del salto. Quindi, se i registri sono uguali, la logica salta all'indirizzo calcolato, se no continua normalmente (con PC + 4)



Il risultato di tutti questi elementi è il **Datapath**.

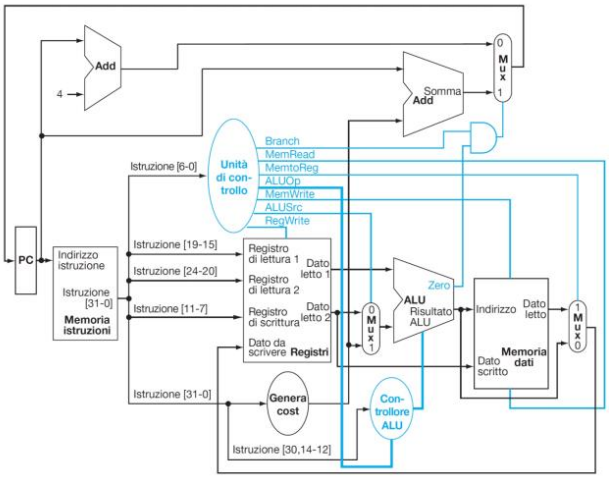
È una semplice unità di elaborazione per il nucleo dell'architettura RISC-V che combini tra loro gli elementi richiesti dalle istruzioni appartenenti ai diversi tipi. Questa unità di elaborazione può eseguire le istruzioni di base (load/store, ALU, salti condizionati) in un singolo ciclo di clock. Per integrare i salti condizionati si è resa necessaria solamente l'aggiunta di un altro multiplexer (MUX).



Unità di controllo

L'unità di controllo (control unit) è la logica di comando della CPU: decide quali segnali attivate in ogni momento in base all'istruzione corrente. È responsabile della gestione del flusso di istruzioni all'interno del processore.

Riceve in input il codice operativo (CodeOP) dell'istruzione e a volte anche funct3 e funct7 (per distinguere ad esempio add da sub). Produce in output segnali come RegWrite, ALUSrc, MemRead, MemWrite, Branch, MemtoReg, ecc.



Nome del segnale	Effetto quando non asserito	Effetto quando asserito
RegWrite	Nullo	Il dato viene scritto nel register file nel registro individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall'estensione del segno dei 12 bit del campo immediato dell'istruzione
PCSrc	Nel PC viene scritta l'uscita del sommatore che calcola il valore di PC + 4	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto
MemRead	Nullo	Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea "Dato letto"
MemWrite	Nullo	Il contenuto della memoria nella posizione puntata dall'indirizzo viene sostituito con il dato presente sulla linea "Dato scritto"
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati

Figura 4.16 Effetto di ciascuno dei sette segnali di controllo. Quando il segnale di controllo a 1 bit di un multiplexer a due vie è asserito, il multiplexer seleziona l'ingresso etichettato con 1; in caso contrario, cioè se il segnale di controllo non è asserito, il multiplexer seleziona l'ingresso etichettato con 0. Si ricordi che tutti gli elementi di stato ricevono il clock come ingresso implicito e che il clock controlla le operazioni di scrittura. Far commutare il clock mediante un segnale esterno a un elemento di stato potrebbe causare problemi di temporizzazione (*vedi* l'Appendice A per la discussione di questo problema).

CPU con pipeline

L'architettura vista finora a singolo ciclo funziona benissimo ma è inefficiente. Il periodo di clock è proporzionale al cammino di esecuzione più lungo (LD), quindi l'intero ciclo di clock deve essere abbastanza lungo da coprire anche l'istruzione più lenta. Questo tempo lo usano però anche le istruzioni più semplici, che non ne avrebbero bisogno. Quindi, anche se CPI = 1, ogni ciclo è troppo lungo, e il processore spreca tempo.

Equazione delle prestazioni

$$T_{CPU} = C_{CPU} \cdot T_C = \frac{C_{CPU}}{fc} = \boxed{\frac{N_{CPU} \cdot \overline{CPI}}{fc}}$$

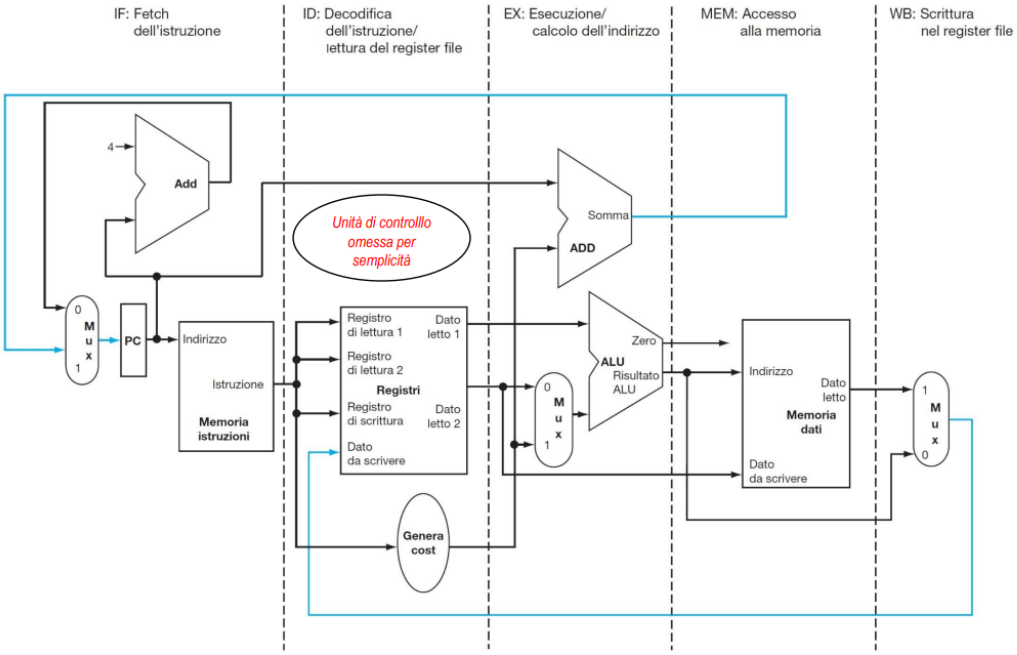
- **T_C è il periodo di clock**
- **f_C è la frequenza di clock == 1/ T_C**
- **C_{CPU} sono i cicli di clock**
- **N_{CPU} è il Numero di istruzioni eseguite DINAMICAMENTE**
- **$\overline{CPI}$ è il numero di Cicli Per Istruzione (medio)**

Esempio:
bne → 3 cicli
add → 2 cicli
2 istruzioni, 5 cicli → /CPI=5/2=2.5

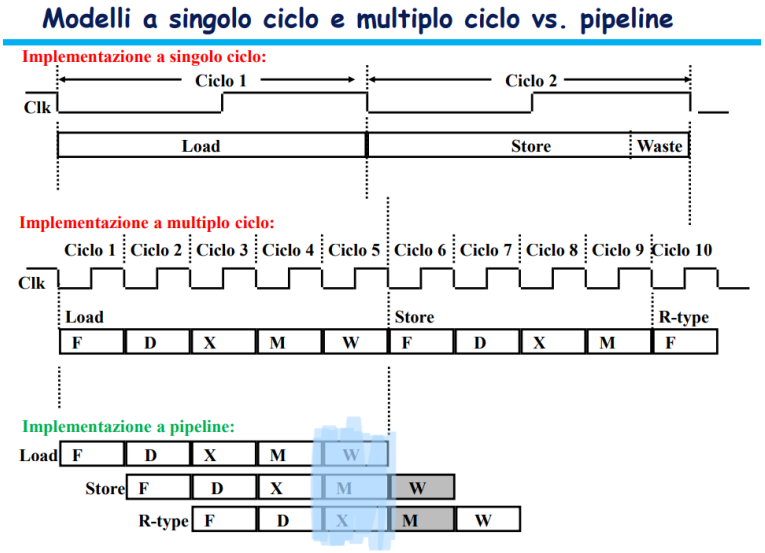
La soluzione è la pipeline, una tecnica di esecuzione delle istruzioni che suddivide il ciclo di esecuzione in fasi sequenziali, consentendo il parallelismo e migliorando l'efficienza del processore. Una CPU moderna spezza ogni istruzione in più fasi (pipeline) così che più istruzioni siano in esecuzione contemporaneamente e ogni fase duri poco (non si aspetta la più lenta).

- Il pipelining non riduce il tempo di esecuzione di ogni singola istruzione, bensì aumenta il loro throughput. Ogni istruzione impiega sempre gli stessi cicli, quindi la latenza (quanto impiega una singola istruzione dall'inizio alla fine) resta uguale. Ma la pipeline permette di lavorare su più istruzioni contemporaneamente; quindi, il throughput (numero di istruzioni completate per unità di tempo) aumenta tantissimo.
- Tutti gli stadi lavorano in contemporanea utilizzando risorse separate, ossia ogni stadio della pipeline ha il proprio hardware separato (come una catena di montaggio).
- Incremento potenziale di velocità = numero di stadi della pipeline.
- La velocità della pipeline è limitata dallo stadio più lento: ogni stadio dura un ciclo di clock; quindi, il clock deve essere abbastanza lungo per coprire lo stadio più lento. Gli stadi della pipeline non bilanciati riducono l'incremento di velocità; quindi, la CPU deve comunque aspettare lo stadio più lento che farà sprecare tempo agli altri.
- Il tempo per riempire la pipeline ed il tempo per svuotarla riducono l'incremento di velocità.
- Stallo per dipendenze: a volte un'istruzione dipende dal risultato della precedente, che non ha ancora completato la pipeline. Quindi la seconda istruzione aspetta fino a che la prima non è completa.

Idea di base: suddivisione della CPU in stadi



La CPU viene divisa in stadi: Fetch (IF), Decode (ID), Execute (EX), Memory (MEM) e Write Back (WB). Ogni stadio fa una parte dell'istruzione e lavora in parallelo con altri.

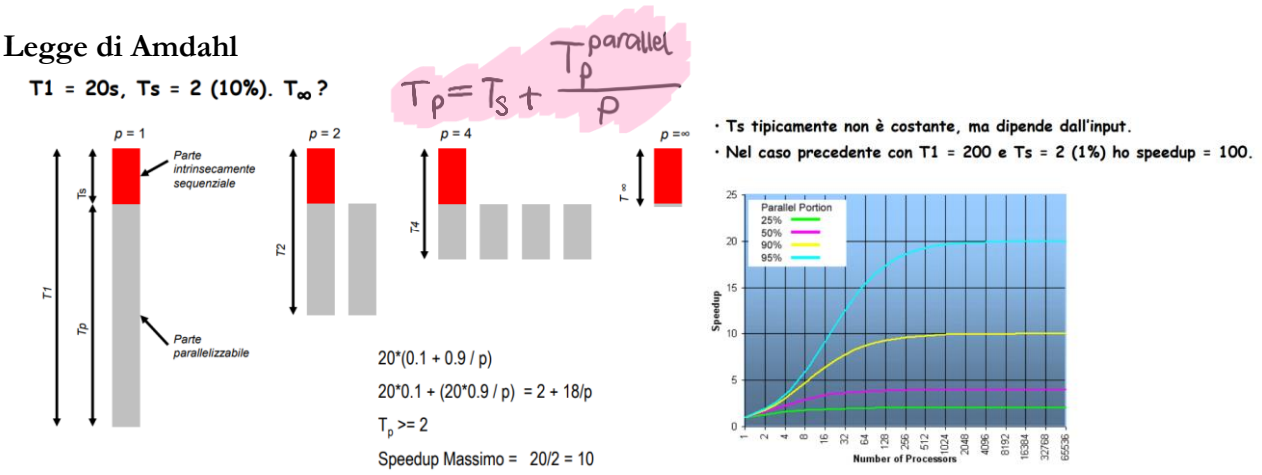


Nell'implementazione a singolo ciclo, ogni istruzione viene completata in un solo ciclo di clock. Tutte le fasi (fetch, decode, execute, mem, writeback) avvengono in quel singolo ciclo, che deve quindi essere molto lungo per contenere anche l'istruzione più lenta. Dopo ogni istruzione c'è uno spreco se la successiva è più veloce (WASTE nello schema).

Nell'implementazione a multiplo ciclo, ogni fase dell'esecuzione prende un ciclo di clock diverso. Le fasi sono fatte in sequenza, ma ogni ciclo è più corto, in quanto fa solo una fase. Il ciclo di clock è quindi più veloce e le istruzioni usano solo i cicli di cui hanno bisogno. Non c'è però nessun parallelismo: solo un'istruzione attiva alla volta.

Nell'implementazione a pipeline ogni fase è uno stadio della pipeline. Le istruzioni si sovrappongono: mentre la prima è in Write Back, la seconda è in mem, la terza in ex, e così via, come una catena di montaggio. Quindi, dopo il primo riempimento, ogni ciclo produce un risultato.

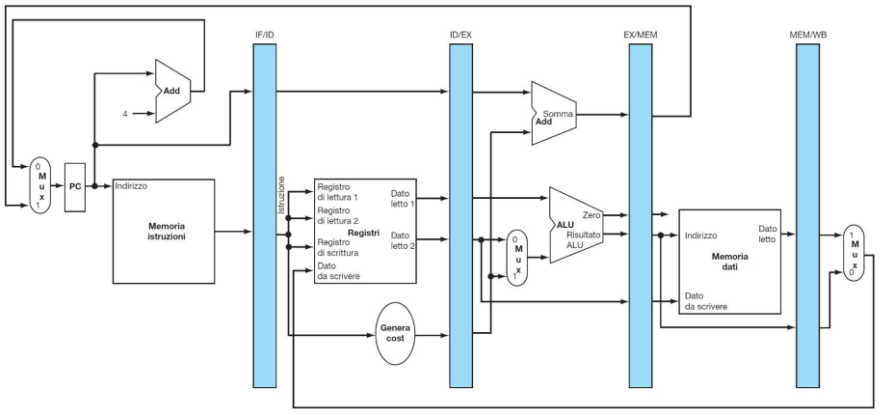
Legge di Amdahl



È una formula che ci dice quanto possiamo velocizzare un programma se rendiamo una parte di esso parallela (eseguibile su più core) e il resto rimane sequenziale (eseguibile da un solo core, intrinseca, c'è sempre).
Solo il codice parallelizzabile trae vantaggio dei processori in più; la parte sequenziale è un limite rigido che blocca lo speedup massimo.

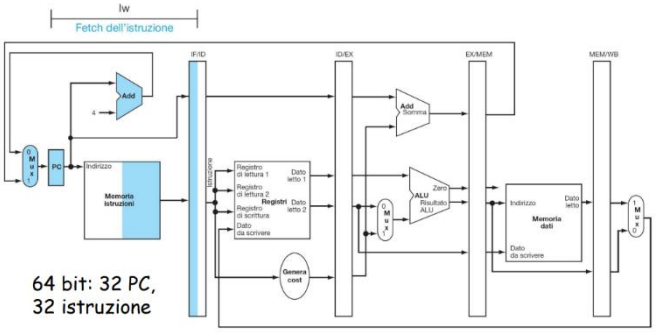
Registri aggiuntivi

Per preservare il valore dell'istruzione in esecuzione, il PC ed altri valori ci servono dei registri specifici per la pipeline. Per etichettarli si usa il nome dei relativi stadi che separano: ad esempio, il primo registro viene etichettato con IF/ID. I registri devono avere un'ampiezza sufficiente a memorizzare tutti i dati corrispondenti alle linee che li devono attraversare. Ad esempio, il registro IF/ID deve essere ampio 64 bit, poiché deve contenere sia l'istruzione prelevata dalla memoria su 32 bit che il contenuto del Program Counter incrementato di 4, su 32 bit. Gli altri 3 registri contengono 128 bit (ID/EX), 97 bit (EX/MEM) e 64 bit (MEM/WB).



Fetch

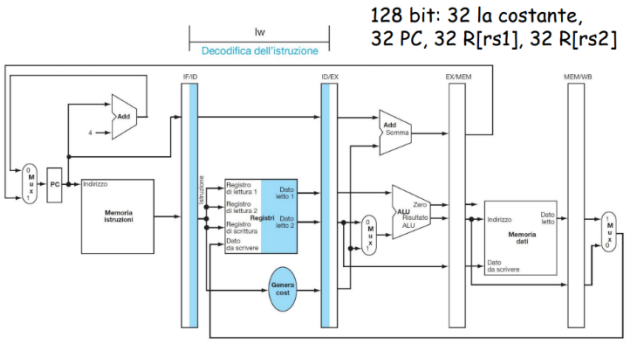
Nel fetch, il PC viene incrementato e riscritto, ma il vecchio valore va salvato in IF/ID perché può essere necessario più avanti, ad esempio nella beq.



Decodifica

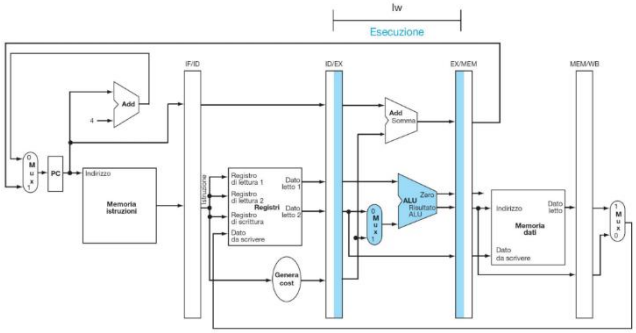
Prende l'istruzione dal fetch (32 bit), e viene smezzata nei suoi campi, quindi rs1, rs2, rd (registro destinazione), imm (costante), opcode, funct3, funct7. Così l'istruzione è decodificata

Nel registro ID/EX vengono salvati il valore di rs1 (32 bit), il valore di rs2 (3d bit), costante (32 bit) e PC corrente (32 bit), per un totale di 128 bit trasportati allo stadio EX per eseguire.



Esecuzione aritmetica o calcolo indirizzo

L'obbiettivo è effettuare il calcolo necessario per l'istruzione, quindi un'operazione aritmetica tra due registri (add, sub, ecc), la somma tra registro base e offset per ottenere l'indirizzo e il confronto tra due registri. Il MUX decide cosa mandare come secondo operandi all'ALU, se il secondo registro (operazioni o beq) o la costante offset (per lw e sw).



È 97 bit perché: 32 risultato, 32 nuovo PC, 32 secondo registro e 'Zero'.

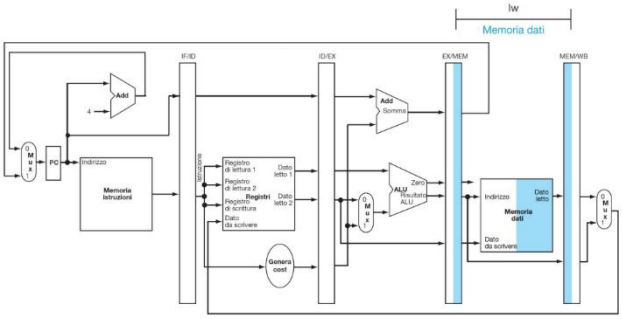
Accesso alla memoria

L'obbiettivo di MEM:

se l'istruzione è lw, legge dalla memoria dati un valore usando l'indirizzo calcolato nello stadio EX. Il valore letto viene poi salvato nel registro MEM/WB, pronto per il Write Back.

se l'istruzione è sw, scrive un valore in memoria all'indirizzo calcolato. Dopo lo store, non c'è Write Back, che verrà quindi 'saltato' (non farà nulla).

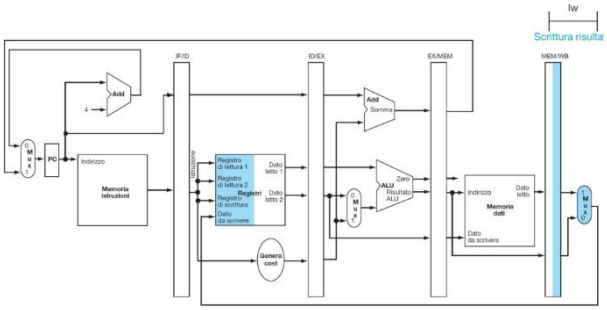
Le altre istruzioni passano comunque da qui ma non fanno nulla.



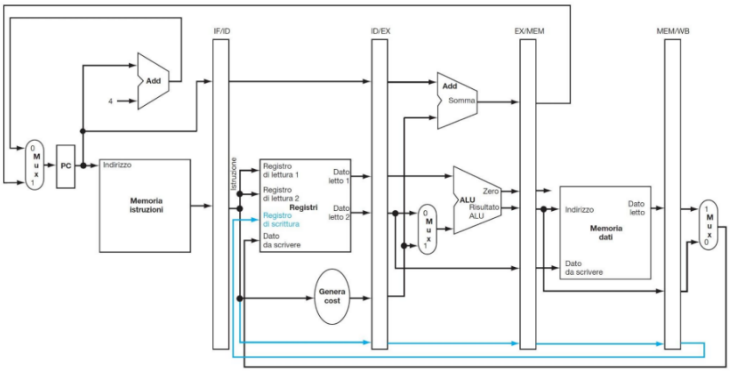
Scrittura nel register file

L'obbiettivo del Write Back è quello di scrivere il risultato, proveniente dalla memoria o dall'ALU, nel registro di destinazione rd.

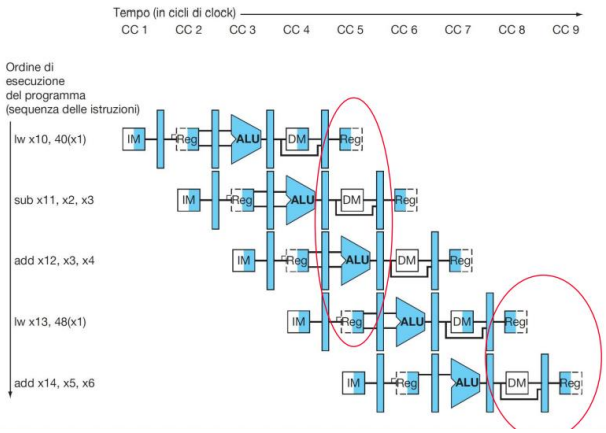
Il circuito ha il dato letto dalla memoria (lw) o il risultato dell'ALU, ma non sa dove scriverlo perché manca l'informazione rd, che dice quale registro va aggiornato.

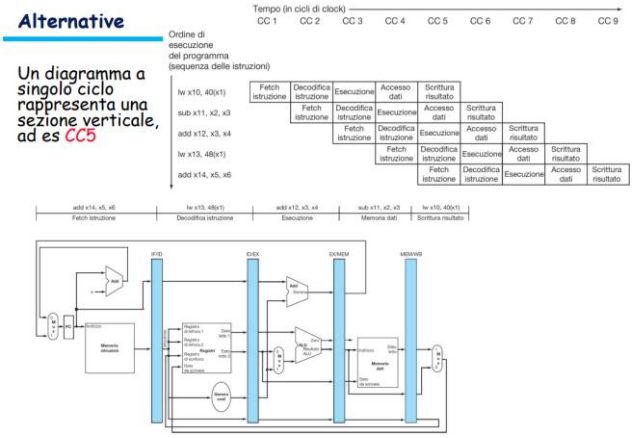


Per risolvere questo problema, l'informazione sul registro di destinazione rd viene ora trasportata lungo tutta la pipeline, fino allo stadio MEM/WB. I 5 bit (registri RISC-V sono 32 e $2^5 = 32$) che identificano rd sono ora inclusi nei registri interstadio (ID/EX, EX/MEM, MEM/WB).



Rappresentazione grafica della pipeline



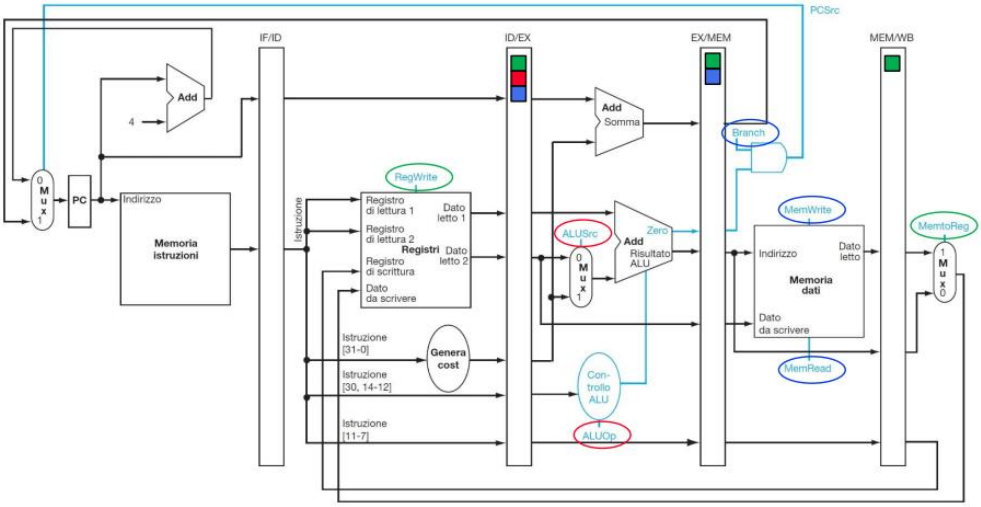


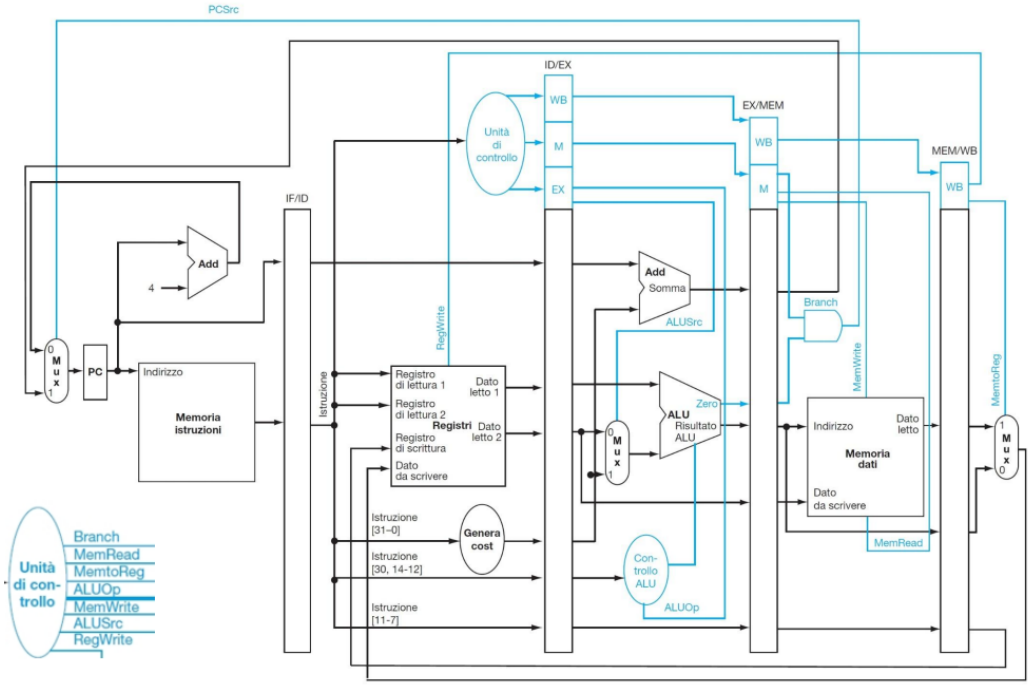
Al datapath, dobbiamo aggiungere l'unità di controllo

Ogni segnale è associato ad un componente e ciascun componente è attivo in uno/due stadi della pipeline. L'unità di controllo genera i segnali per: dire all'ALU che operazione fare, dire se attivare lettura o scrittura in memoria, dire se scrivere nel registro, indicare le sorgenti degli operandi.

	EX		MEM			WB	
Istruzione	**Segnali di controllo dello stadio di esecuzione/calcolo dell'indirizzo**		**Segnali di controllo dello stadio di accesso alla memoria dati**			**Segnali di controllo dello stadio di scrittura**	
	ALUOp	**ALUSrc**	**Branch**	**MemRead**	**MemWrite**	**RegWrite**	**MemtoReg**
Tipo R	10	0	0	0	0	1	0
ld	00	1	0	1	0	1	1
sd	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

Nello schema, devo quindi nuovamente estendere i registri, visto che i segnali sono definiti in ID.





I salti condizionati (beq) in pipeline introducono un hazard di controllo. Non basta azzerare i segnali di controllo, quindi la soluzione è il flush della pipeline.

Criticità

Il pipelining può provocare problemi?

Sì: Criticità (Hazard) nelle pipeline, conflitti quando più istruzioni vengono eseguite in parallelo

- Criticità strutturali: tentativo di utilizzare le stesse risorse (hardware) due volte allo stesso tempo
- Criticità sui dati: tentativo di utilizzare un dato prima che sia prodotto (da istruzione precedente)
- Criticità sul controllo: tentativo di saltare (if) pur avendo caricato istruzioni del ramo non-preso (ossia il processore non sa ancora se prenderà un ramo o un altro)

Le criticità si risolvono tutte aspettando, oppure modificando l'architettura.

Il controllo della pipeline deve:

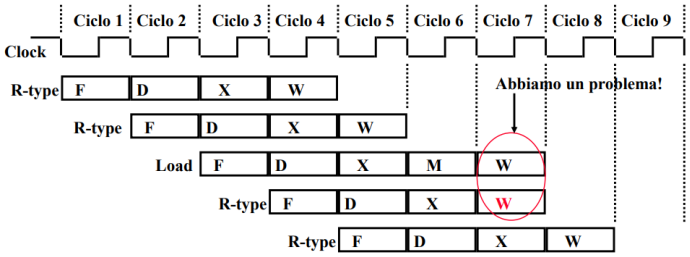
- Rilevare la presenza di criticità
- Gestirla, fermando un'istruzione, rimandandola, o adattando il flusso.

Un flusso senza stalli nella pipeline è molto difficile: anche se l'idea è quella di far lavorare tutto in parallelo, nella pratica ci sono sempre momenti in cui qualcosa deve aspettare.

Criticità strutturali

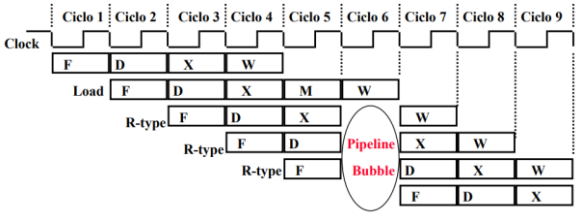
Le criticità strutturali sono situazioni in cui due istruzioni cercano di usare la stessa risorsa hardware nello stesso ciclo di clock, ma questa risorsa non è duplicata.

Facciamo un esempio, dove abbiamo 4 istruzioni di tipo R (quindi add, sub, ecc) a 4 cicli, e scrivono nel registro al ciclo 4. Abbiamo poi una load che scrive al ciclo 5 (più lunga rispetto alle altre istruzioni).

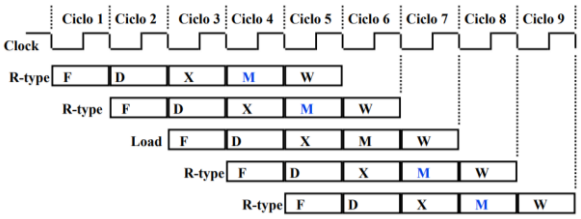


Abbiamo una criticità strutturale perché due istruzioni stanno tentando di scrivere nei registri contemporaneamente, anche se la porta di scrittura è unica.

La prima soluzione è quella di inserire una bolla, che blocca l'ingresso di una nuova istruzione per un ciclo, creando un vuoto nella pipeline. Questa è la NOP, un'istruzione speciale che appunto non fa nulla. Si crea impostando a 0 tutti i segnali, rimandando l'istruzione.



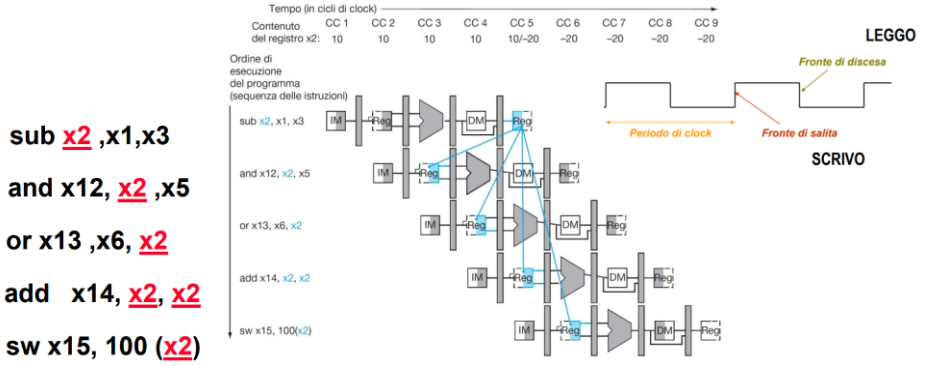
La seconda soluzione è far sì che anche le istruzioni di tipo R scrivano nello stadio 5 come la load. In questo modo, nessuna istruzione cerca di scrivere nello stesso ciclo. Le istruzioni di tipo R non faranno nulla nello stadio MEM, ma la pipeline rimane ordinata. Questo è il metodo che usa RISC-V.



Criticità sui dati

La criticità sui dati avviene quando un'istruzione ha bisogno di un dato che viene prodotto da un'istruzione precedente, ma quel dato non è ancora pronto.

Facciamo un esempio:



x2 viene calcolato alla prima istruzione (sub), ma viene subito nelle successive 4. Abbiamo quindi una catena di dipendenze sul registro x2. x2 viene scritto solo nel ciclo 5; se la and lo legge nel ciclo 2, prende quindi il valore sbagliato, perché non è ancora stato aggiornato.

La criticità dei dati può essere classificata:

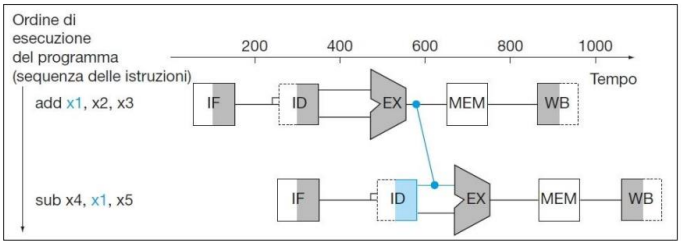
- **RAW (Read After Write) o FLOW dependency:** un'istruzione successiva legge un registro prima che l'istruzione precedente l'abbia scritto. È il caso dell'esempio.
- **WAR (Write After Read) o ANTI dependency:** un'istruzione successiva cerca di scrivere un registro che è stato letto dall'istruzione precedente
- **WAW (Write After Write) o OUTPUT dependency:** due istruzioni consecutive scrivono lo stesso registro

Le soluzioni sono:

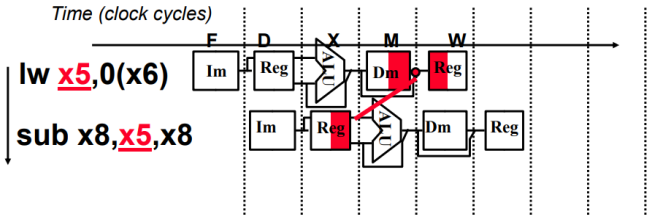
- **Inserimento di bolle**
- **Propagazione (o bypassing):** passo il dato direttamente, senza aspettare WB
- **Riordinamento istruzioni:** sposto il codice per evitare che le istruzioni dipendenti siano troppo vicine

Propagazione

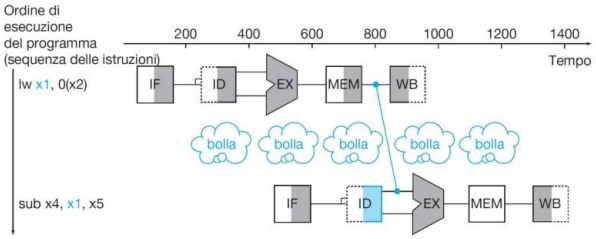
Quando possibile, il dato può essere 'anticipato' e passato dallo stadio EX o MEM direttamente al prossimo EX. In questo modo non si aspetta WB e si evita di introdurre una bolla.



La propagazione non basta se ci troviamo in un caso di load/use hazard.

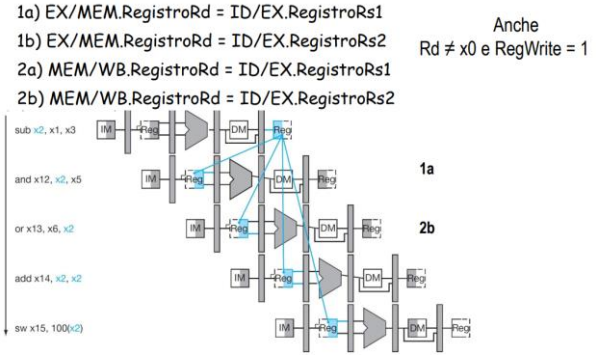


In questo caso, il dato è disponibile solo a fine MEM, quindi le istruzioni che dipendono da una load devono essere ritardate e mandate in stallo via hardware, chiamato **interlock**, che inserisce una bolla automatica per aspettare il dato.

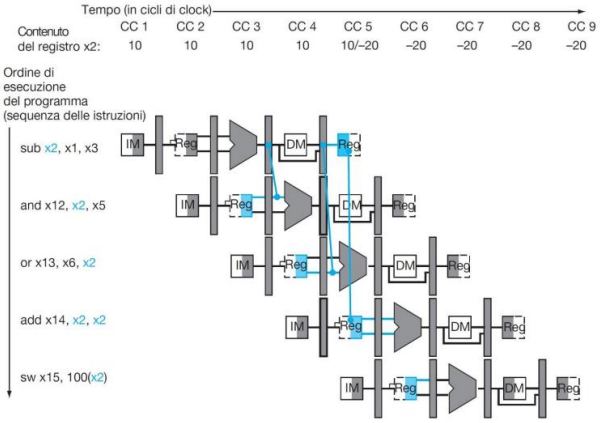


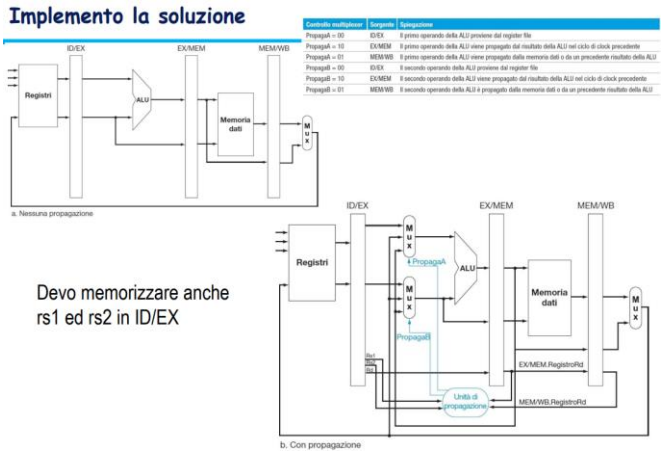
Rilevamento delle dipendenze

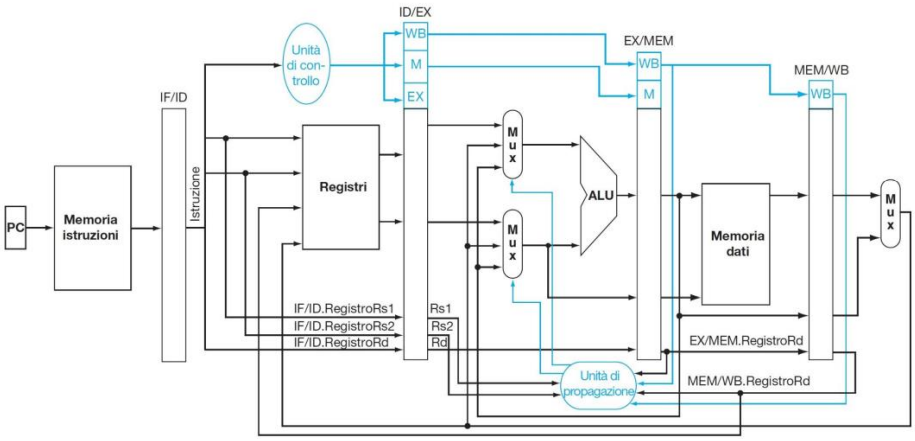
L'obbiettivo è quello di capire quando due istruzioni in pipeline stanno accedendo allo stesso registro e quindi potrebbero causare una criticità (hazard di tipo RAW). Per capire se c'è una RAW hazard bisogna confrontare i registri sorgente (rs1, rs2) dell'istruzione attualmente in fase di Decode (ID/EX) con il registro destinazione (rd) delle istruzioni nelle fasi successive (EX/MEM o MEM/WB).



La soluzione

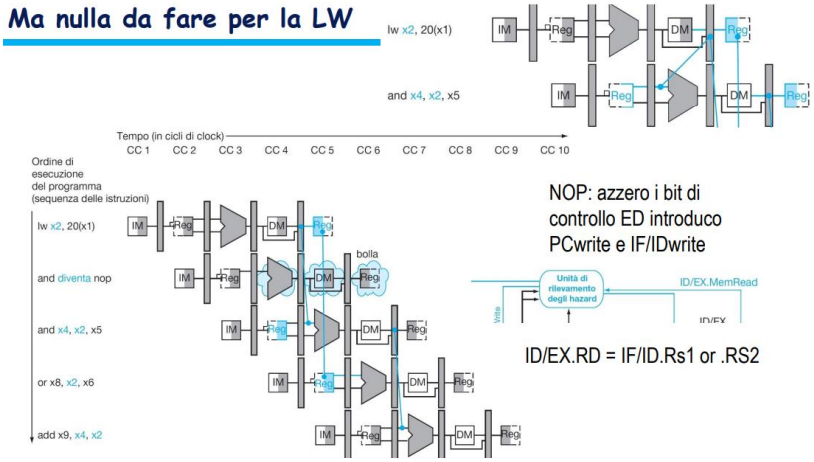




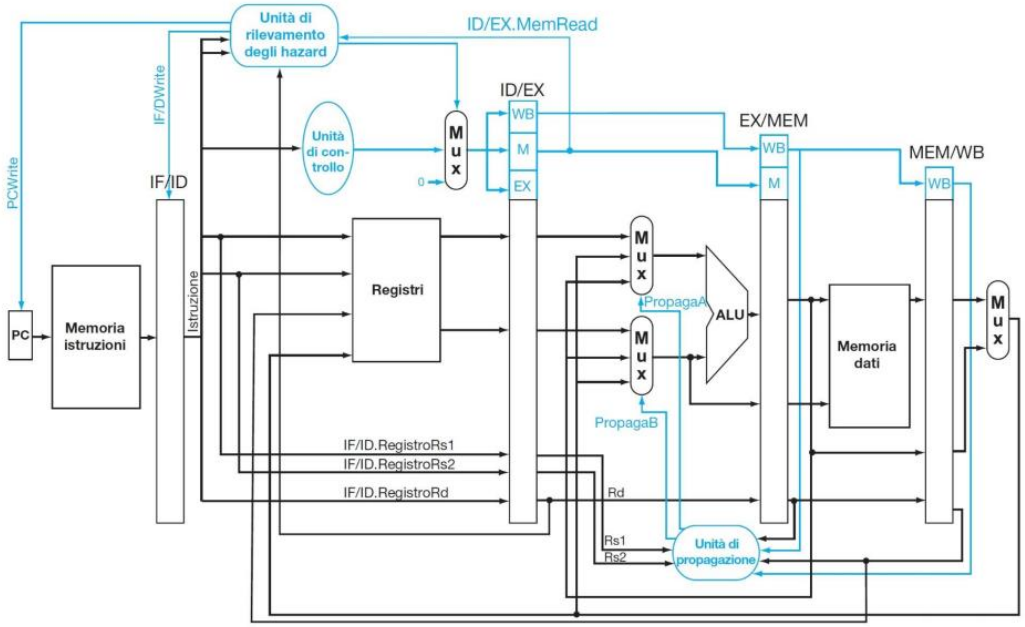


La fowarding unit (unità di propagazione) prende in ingresso i valori rs1, rs2, rd e RegWrite da ID/EX, EX/MEM e MEM/WB, e decide, tramite circuiti logici, se fare forwarding da EX/MEM, da MEM/WB o non fare nulla.
La Forwarding Unit è utilizzata per gestire le criticità sui dati senza dover inserire NOP. Questa unità permette di "inoltrare" i dati appena calcolati da uno stadio della pipeline ad uno stadio precedente che ne ha bisogno. In altre parole, se un'istruzione successiva ha bisogno del risultato di un'istruzione precedente che non ha ancora terminato la sua esecuzione, la Forwarding Unit può prelevare il risultato intermedio e fornirlo all'istruzione successiva, evitando così di dover aspettare che il dato passi attraverso tutti gli stadi della pipeline. Esempio: se l'istruzione `ADD` produce un risultato che è immediatamente necessario per un'istruzione `SUB`, la Forwarding Unit preleverà il risultato dell'`ADD` dallo stadio di Execution (EX) e lo inoltrerà direttamente allo stadio di Decode (ID) dell'istruzione `SUB`.

Il load/use hazard non può comunque essere risolto solo con il forwarding, perché lw produce il dato solo a fine MEM e se l'istruzione successiva vuole già usarlo in EX, il dato non è ancora disponibile. Serve quindi comunque uno stallo NOP.



La versione finale è quindi così:



L'Hazard Detection Unit (Unità di Rilevamento delle Criticità) ha il compito di rilevare quando si verifica una criticità nella pipeline.

Criticità sul controllo

In uno schema base (quello che abbiamo visto finora), la decisione sul salto (branch) viene presa solo nello stadio MEM:

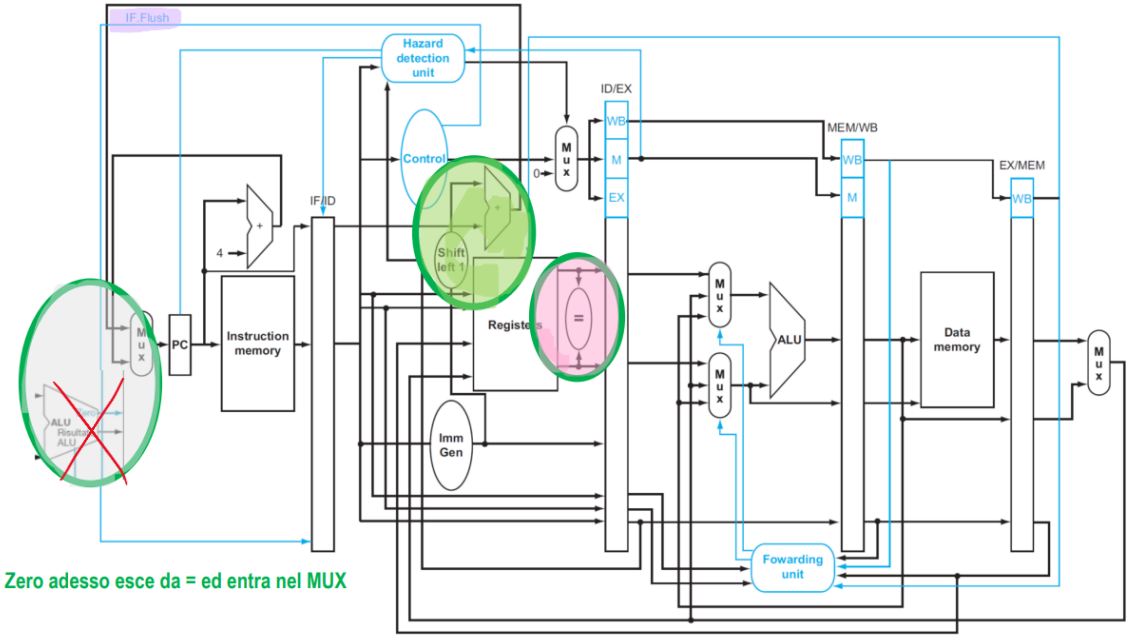
- Quando arriva un'istruzione beq, nel frattempo sono già entrate nella pipeline 3 nuove istruzioni (PC+4, PC+8, PC+12)
- Se il branch viene preso, queste 3 istruzioni sono sbagliate, e bisogna quindi annullarle

Quindi il processore perderebbe tre cicli per ogni salto, e dato che la frequenza delle operazioni di salto è di circa il 15-17%, l'impatto sulle prestazioni è notevole.

Le soluzioni sono tipicamente due, che però non risolvono completamente il problema:

- Prevedere quale ramo verrà preso
- Anticipare la valutazione del saldo nello stadio Decode

Ottimizzazione 1: branch anticipato nello stadio ID



Quindi calcola nello stato ID se rs1 == rs2 (e quindi beq deve saltare). Calcola già in questo stadio l'indirizzo target (PC+offset) e se Zero == 1, il salto è da fare, quindi il PC si aggiorna direttamente. Questo serve per ridurre il numero di cicli persi nelle istruzioni di salto.

Il comparatore (=) confronta i due registri letti (rs1 e rs2) e se sono uguali produce il segnale Zero = 1. Il comparatore sostituisce quindi l'ALU usata in MEM per il confronto.

Il generatore dell'indirizzo del salto (shift left 1 + somma) prende l'offset dal campo immediato (ImmGen), lo shifta a sinistra di 1 bit (perché lavoriamo con indirizzi in byte) e lo sommiamo a PC per ottenere PCtarget.

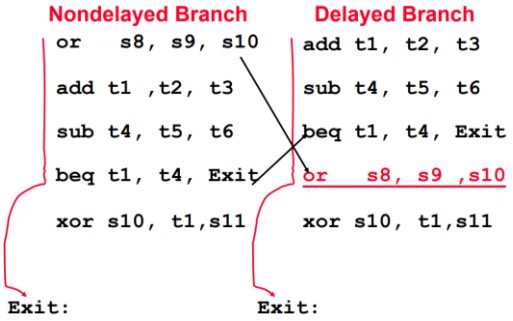
Il MUX a sinistra del PC aggiorna il PC normalmente a PC+4 se il salto non viene fatto, mentre se il salto viene fatto (Zero = 1), il PC si aggiorna a PCtarget.

Se il salto viene preso, la pipeline va pulita, le istruzioni sbagliate devono essere eliminate. Si attiva quindi il segnale IF.Flush che azzera i registri interstadio (NOP).

Ottimizzazione 2: ridefinire i salti (Branch-delay slot)

Sia che si faccia il salto, sia che non si faccia, viene eseguita l'istruzione che segue immediatamente il salto. Nel caso peggiore, si può sempre inserire una NOP nel branch-delay slot; nel caso migliore si può trovare un'istruzione prima del branch che può essere messa nel branch-delay slot senza influenzare il flusso di programma: il riordinamento delle istruzioni è un metodo comunemente utilizzato per velocizzare i programmi, fatto dal compilatore, che può trovare istruzioni indipendenti appropriate.

Quindi, il branch delay slot fa eseguire un'istruzione utile immediatamente dopo un'istruzione di salto, che può essere riordinata dal compilatore.

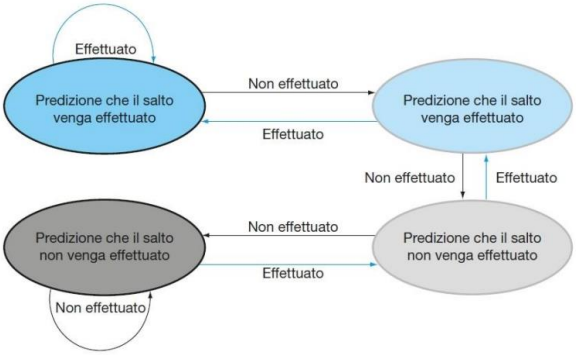


Predizione del salto

Posso predirlo in due modi:

- Posso definire in modo statico che un salto sia sempre eseguito o no, quindi suppongo sempre che il salto venga preso o non venga preso
- Dinamicamente, vedo come si è comportato il codice l'ultima volta: tramite una branch history table, tengo traccia del comportamento passato del salto, uso una parte dell'indirizzo dell'istruzione come indice + un bit (80% accuratezza) o due bit (90% accuratezza) per ricordarmi se ho effettuato il salto o no.

Utilizzo tecniche di predizione per minimizzare gli stalli causati dalle istruzioni di salto.



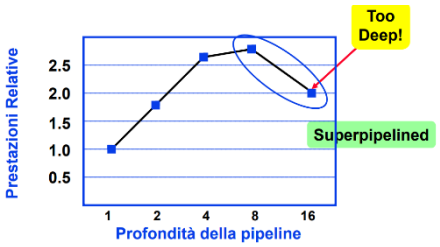
Punti principali del pipelining

- Le pipeline fanno passare informazioni di controllo al loro interno nello stesso modo in cui fanno passare i dati
- Propagazione e stalli sono gestiti dal controllo locale
- Le criticità limitano le prestazioni
    - Strutturali: richiedono più risorse hardware
    - Dati: richiedono propagazione, schedulazione dal compilatore
    - Controllo: valutazione preliminare, delayed branch, predizione
- Aumentando la lunghezza della pipe aumenta l'impatto delle criticità; il pipelining migliora il throughput delle istruzioni, ma non la loro latenza
- I compilatori riducono il costo delle criticità sui dati e sul controllo: Load delay slots, Branch delay slots Branch prediction

Profondità della pipeline

La pipeline non può essere troppo profonda.
Se aumento troppo il numero di stadi:

- Aumentano i cicli persi per stadi
- Serve più logica di controllo per gestire hazard
- Oltre un limite, non migliora più la prestazione



Eccezioni

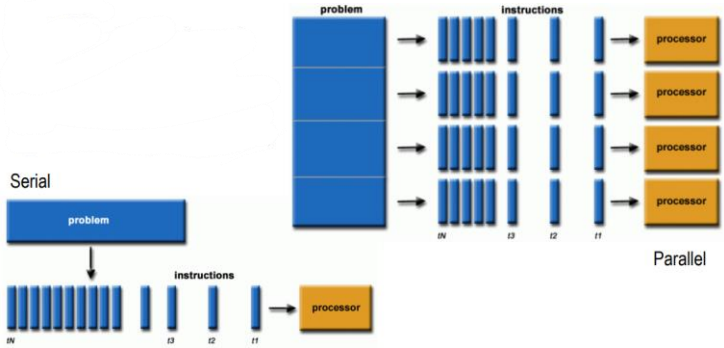
Le eccezioni sono tipicamente eventi non previsti che alterano il normale flusso delle istruzioni: I/O, malfunzionamento hardware, overflow aritmetico.
Vengono distinte dagli interrupt in quanto sono eventi esterni, mentre questi sono interni.
Sono trattate come gli hazard sul controllo e richiedono un lavoro congiunto tra hardware e sistema operativo per gestirle:

1. Blocco l'istruzione responsabile dell'eccezione prima che venga terminata
2. Concludo le istruzioni precedenti
3. Elimino le istruzioni successive dalla pipeline
4. Scrivo la causa dell'eccezione nel registro SCAUSE
5. Salvo l'indirizzo dell'istruzione responsabile dell'eccezione in SEPC
6. Trasferisco il controllo ad un indirizzo preassegnato corrispondente alla prima istruzione della procedura del SO che la gestisce
7. Il SO la analizza e, a seconda della causa, la gestisce (I/O) o killa il programma (malfunzion.)

Parallelismo a livello di singola CPU

Instruction Level Parallelism (ILP)

L'HPC (High Performance Computing) è la pratica di aggregare la potenza di calcolo in modo da ottenere prestazioni superiori a quelle che si potrebbero ottenere da un tipico computer desktop per risolvere grandi problemi nel campo della scienza, ingegneria o attività commerciale.



L'instruction Level Parallelism è qualunque forma di esecuzione simultanea di più di un'istruzione per ciclo di clock. La pipeline è la prima forma di ILP: ogni stadio lavora su un'istruzione diversa, e maggiore il parallelismo = minore tempo di esecuzione. Per andare oltre, si aumentano i componenti interni per eseguire più istruzioni nello stesso stadio e per ciclo di clock, impacchettandole in slot di esecuzione. In questo stadio non parlo più di CPI ma di IPC (Istruzioni per ciclo).
La parallelizzazione dell'esecuzione può essere:

- Statica: le decisioni sono prese durante la compilazione (compilatore)
- Dinamica: le decisioni sono prese durante l'esecuzione (hardware)
- Ibrida: una combinazione delle due, la più usata

Speculazione

La speculazione è una tecnica che permette al processore di anticipare l'esecuzione di istruzioni successive basandosi su previsioni, anziché attendere il completamento delle istruzioni precedenti.

Se la previsione è corretta, si ottiene un miglioramento delle prestazioni, ma se è errata, si devono scartare le istruzioni speculativamente eseguite, comportando un costo in termini di prestazioni. La verifica dell'accuratezza della previsione però complica la pipeline, perché si deve svuotare e tornare indietro.

Nella speculazione statica il compilatore inserisce istruzioni aggiuntive per controllare e "riparare" all'errore. Nella speculazione dinamica i dati vengono memorizzati in un buffer e copiati nel register file o memoria solo in caso di speculazione corretta.

Parallelizzazione statica dell'esecuzione

Le decisioni sono prese durante la compilazione (direttamente nell'Hardware), dove il compilatore si occupa di formare i pacchetti di istruzioni (come una singola istruzione lunga, con campi predefiniti, stile Very Long Instruction Word VLIW) e prevenire o ridurre gli hazard, anche con la predizione.

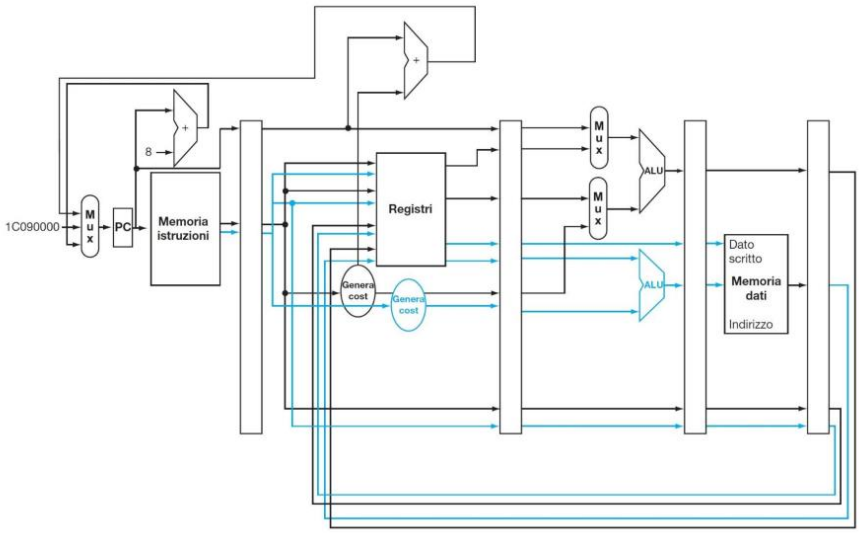
Consideriamo un'estensione della pipeline a due vie:

- Una per le operazioni aritmetiche e condizionali (ALU, branch)
- Una per load/store

Le istruzioni sono allineate e accoppiate in parole da 64 bit, e se non posso accoppiare due istruzioni uso una NOP.

Tipo di istruzione	Stadi della pipeline						
Istruzione ALU o di salto	IF	ID	EX	MEM	WB		
Istruzione di load o di store	IF	ID	EX	MEM	WB		
Istruzione ALU o di salto		IF	ID	EX	MEM	WB	
Istruzione di load o di store		IF	ID	EX	MEM	WB	
Istruzione ALU o di salto			IF	ID	EX	MEM	WB
Istruzione di load o di store			IF	ID	EX	MEM	WB

Per supportare queste tecniche servono componenti extra nell'hardware:



- Bit addizionali dalla memoria istruzioni
- Due porte di lettura e una di scrittura per il register file
- Una ALU che si occupa del calcolo degli indirizzi per il trasferimento dati

Loop unrolling: È una tecnica che consiste nel sostituire le istruzioni all'interno di un ciclo con più istruzioni ripetute, in modo da ridurre il numero di volte che il ciclo deve essere iterato e aumentare così il parallelismo. Per esempio, se un ciclo esegue un'operazione dieci volte, utilizzando il loop unrolling, vengono generate dieci copie dell'istruzione all'interno del ciclo, consentendo al processore di eseguire più istruzioni in parallelo senza dover ripetere il ciclo dieci volte.

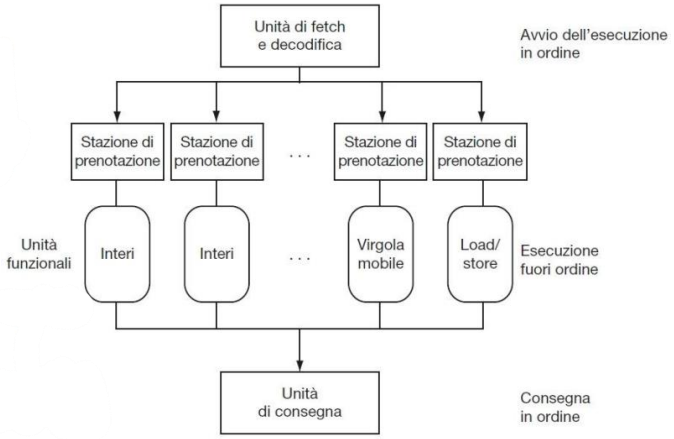
Parallelizzazione dinamica dell'esecuzione

Le istruzioni sono eseguite in ordine (quello deciso dal compilatore) e in esecuzione il **processore** decide se mandarne in esecuzione una alla volta, due o più (direttamente nel Software). Le componenti necessarie sono:

- Dynamic scheduling: per riordinare le istruzioni in tempo reale
- Più unità funzionali (ALU, load, store, …)
- Una stazione di prenotazione: ogni istruzione aspetta lì finchè non ha tutti gli operandi
- Unità di consegna (commit unit): gestisce il completamente delle istruzioni
- Buffer di riordino nell'unità di consegna: tiene traccia delle istruzioni speculative e dei risultati temporanei

La ridenominazione dei registri per evitare conflitti, usando nomi temporanei, è data dalle stazioni di prenotazione insieme al buffer di riordino.

I processori superscalari si basano su questa tecnica. La differenza tra il processore superscalare e quello VLIW è che nei primi è l'hardware che garantisce la correttezza dell'esecuzione, mentre qui è il processore



- Ogni istruzione eseguita (solo quando gli operandi sono disponibili) va nella stazione corrispondente, e solo quando gli operandi sono disponibili (e.g. load, propagation) l'istruzione può essere eseguita
- Le istruzioni possono essere eseguite fuori ordine MA saranno consegnate sempre in ordine
- Le unità funzionali eseguono le istruzioni non appena i dati sono disponibili
- I dati possono arrivare dal register file, dalla memoria dati o dal buffer dell'unità di consegna
- Se la speculazione è errata cancello i dati dal buffer di riordino

Alcune operazioni, come MEM e ALU, non sempre durano un ciclo, creando latenze, ossia tempi di attesa. Il modello superscalare serve proprio per nascondere queste latenze grazie all'esecuzione parallela e fuori ordine.

Un esempio di ILP dinamico è la CPU intel i7.

Sandy Bridge

Intel Sandy Bridge è una microarchitettura sviluppata da Intel e rilasciata nel 2011, comprende una grafica integrata, prestazione della CPU potenziate e nuove estensioni dell'insieme di istruzioni.

Il Front-End è la parte della CPU che prepara le istruzioni per essere eseguite. Funziona così:

1. Fetch delle istruzioni: prende le istruzioni x86 dalla memoria. Queste sono complicate (lunghezza variabile).
2. Pre-decode buffer:
    - Capisce dove inizia e finisce ogni istruzione.
    - Le istruzioni vengono convertite in unità chiamate Macro-Operations (MOP).
3. Branch Prediction Unit (BPU):
    - Tenta di indovinare il prossimo salto (es. if/else, loop).
    - Se sbaglia, bisogna svuotare la pipeline, perdendo più di 150 istruzioni in volo.
4. Le MOP finiscono in una Instruction Queue, dove si fa anche una fusione (es. un'istruzione confronto + salto può diventare un'unica macro-operazione più efficiente).

In breve: il front-end traduce il linguaggio complicato del software in un formato semplice e veloce da eseguire per l'hardware.

Il motore di esecuzione è dove le istruzioni vengono effettivamente eseguite. Ecco come funziona:

1. Ogni ciclo, il front-end può inviare fino a 4 µOPs (micro-operazioni) al motore.
2. A questo punto:
    - I registri logici (quelli che vede il programmatore) sono mappati su registri fisici reali.
    - Vengono prenotate risorse per load/store e porte del scheduler.
3. Il scheduler trattiene i µOP finché:
    - Gli operandi non sono pronti (es. devono ancora arrivare dalla memoria o da altre µOP).
    - L'unità funzionale richiesta è libera.
4. 6 porte di esecuzione:
    - Porte 0, 1, 5: eseguono operazioni di calcolo.
    - Porte 2, 3, 4: per caricare/salvare da memoria.
5. Quando una µOP è completata, può essere ritirata (retired) → il risultato diventa ufficiale.

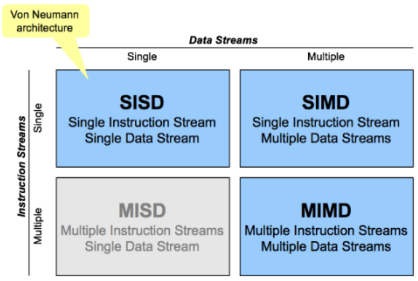
Sandy Bridge supporta anche AVX, un'estensione SIMD per operare su più numeri alla volta (es. vettori di 256 bit).

Parallelismo a livello di parola (vettorizzazione)

La tassonomia di Flynn distingue computer con architetture multi-processore in base a come possono essere classificati tra le due dimensioni indipendenti di Instruction Stream (flusso di istruzioni) and Data Stream (flussi di dati).

Ci sono 4 categorie:

- SISD (Single Instrucition, Single Data): CPU classica, architettura Von Neumann
- SIMD (Single Istruction, Multiple Data): vettorizzazione
- MISD (Multiple Instruction, Single Data): raro
- MIMD (Multiple Istruction, Multiple Data): multithreading



Quando carico un byte o una halfword (dati piccoli) 'spreco' mezzo registro. Sono rare le operazioni ALU su questi tipi di dati, mentre sono molto frequenti le operazioni su vettori di dati in ambito multimediale. Però nel multimedia (audio, video) uso tanti dati piccoli (8, 16 bit).

In questo caso la vettorizzazione è utile.Nel calcolo vettoriale: uso registri più lunghi per ospitare più dati insieme ed eseguire la stessa operazione su v[i], v[i+1] …v[i+n] (quindi su tutti) in un solo ciclo di clock.

SIMD

Una sola istruzione agisce su più dati in parallelo.

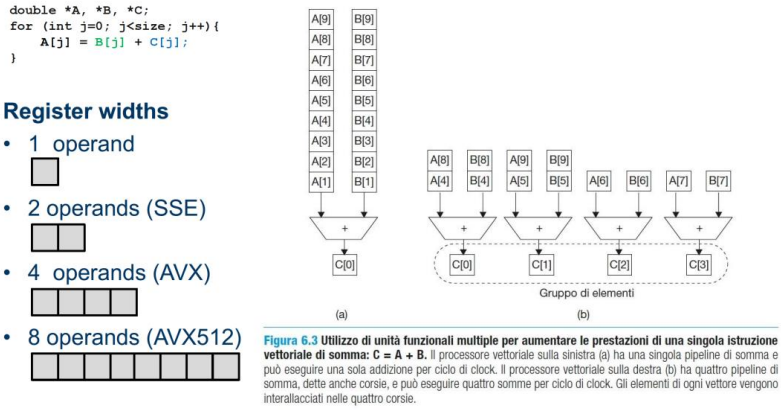
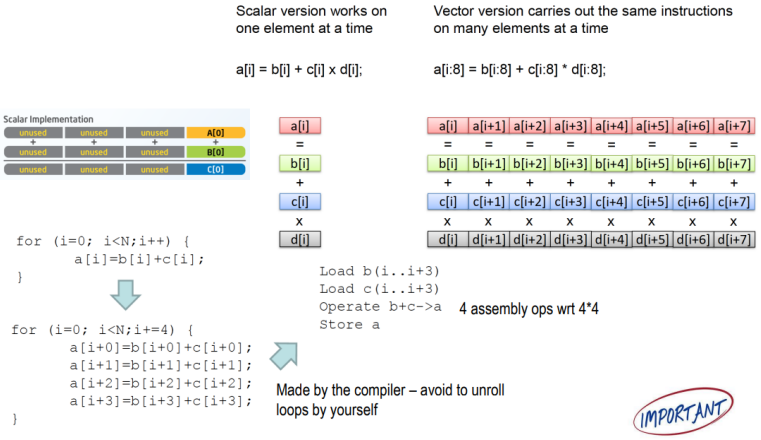


Figura 6.3 Utilizzo di unità funzionali multiple per aumentare le prestazioni di una singola istruzione vettoriale di somma: C = A + B. Il processore vettoriale sulla sinistra (a) ha una singola pipeline di somma e può eseguire una sola addizione per ciclo di clock. Il processore vettoriale sulla destra (b) ha quattro pipeline di somma, dette anche corsie, e può eseguire quattro somme per ciclo di clock. Gli elementi di ogni vettore vengono interallacciati nelle quattro corsie.

Secondo Intel, vettorizzare significa srotolare un ciclo (loop unrolling) combinato con la generazione di istruzioni SIMD impacchettate dal compilatore, che operano su array di dati. Visto che le istruzioni impacchettate operano in più di un elemento di dato alla volta, il ciclo può essere eseguito più efficientemente.

A volte viene chiamato auto-vettorizzazione (auto-vectorization), per enfatizzare che il compilatore identifica e ottimizza i cicli ottimizzabili da solo, senza aver bisogno di azioni dal programmatore: il compilatore fa tutto da solo, evita che tu debba scrivere versioni 'a mano' del codice.



Nomi dei registri vettoriali:

- SSE: registri da 128 bit, da xmm0 a xmm15 (usati in modalità scalare)
- AVX2: registri da 256 bit, da ymm0 a ymm15
- AVX512: registri da 512 bit, da zmm0 a zmm31

Queste sono estensioni SIMD (SSE, AVX, AVX2, AVX512), che possono arricchire l'architettura x86. Servono ad indicare quante operazioni vettoriali può eseguire una CPU in parallelo. Più l'estensione è avanzata e più è larga la dimensione del vettore (da 128 a 512 bit).

Requisiti per la vettorizzazione

- Contabile al runtime
    - Il numero di iterazioni deve essere noto prima di iniziare
    - Non ci devono essere terminazioni condizionali (break, continue, return) dentro al ciclo

- Deve avere un flusso di controllo singolo
    - Non ci devono essere switch statements (switch, goto)
    - Gli if sono possibili se implementati come 'masked assignments' (if … var=x else var=y)
- Deve essere il ciclo più interno se sono cicli annidati
- Non ci devono essere chiamate di funzione
    - Sono possibili funzioni matematiche come pow(), sin(), ecc

La vettorizzazione cambia l'ordine della compilazione rispetto al caso sequenziale; quindi, il compilatore deve essere in grado di provare che la vettorizzazione produrrà un risultato corretto.
Un problema tipico è quello dell'aliasing dei puntatori: se due puntatori puntano alla stessa memoria, vettorizzare potrebbe rompere il programma.
Se puoi eseguire un ciclo al contrario (for i = n; i > 0; i--) senza cambiare risultato, probabilmente si può vettorizzare.
Anche se il ciclo sembra 'giusto', a volte il compilatore non lo vettorizza, magari perché ha un dubbio su possibili dipendenze o ambiguità.

Data dependencies between iterations
- Proven Read-after-Write data (i.e., loop carried) dependencies
- Assumed data dependencies
  - Aggressive optimizations (e.g., IPO) might help

Vectorization won't be efficient
- Compiler estimates how better the vectorized version will be
- Affected by data alignment, data layout, etc.

Unsupported loop structure
- While-loop, for-loop with unknown number of iterations
- Complex loops, unsupported data types, etc.
- (Some) function calls within loop bodies
  - Not the case for SVML functions

RaW dependency

```
for (int i = 0; i < N; i++)
    a[i] = a[i-1] + b[i];
```

Inefficient vectorization

```
for (int i = 0; i < N; i++)
    a[c[i]] = b[d[i]];
```

Function call within loop body

```
for (int i = 0; i < N; i++)
    a[i] = foo(b[i]);
```

Loop-Carried Dependence vs. Loop Independent Dependence

Sono determinati dalla relazione tra statement e iterazione di un loop.

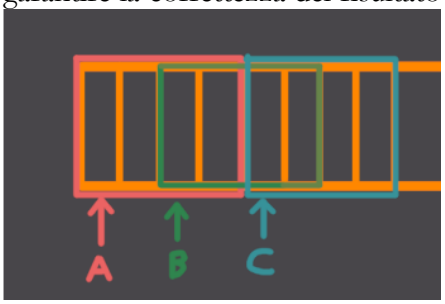
- LCD: quando uno statement in un'iterazione di un loop dipende in qualche modo ad uno statement di una iterazione diversa dello stesso loop (un'iterazione dipende da una precedente)
- LID: se uno statement in un'iterazione di un loop dipende solo da uno statement nella stessa iterazione del loop E due accessi di memoria sono dipendenti tra loro se riferiscono allo stesso luogo di memoria (memory location) e una delle referenze è un write (ogni iterazione è indipendente)

Tipi di dipendenze tra accessi di memoria:

- **RAW:** variabile scritta prima e poi letta, non vettorizzabile
- **WAR:** variabile prima letta poi scritta, vettorizzabile
- **WAW:** variabile scritta e poi riscritta, non vettorizzabile
- **RAR:** non è una vera dipendenza, quindi è sempre vettorizzabile

Aliasing and correctness

Se due array si sovrappongono in memoria, il compilatore non può garantire la correttezza del risultato che vettorizza.



Esempio: C[i] = A[i] + B[i], ma B[0-1] è A[2-3] e C[0-1] è B[2-3]

- Nella versione scalare, scriviamo prima in C[0] nella prima iterazione (i==0) e poi la leggiamo (using B[2]) nella terza iterazione (i==2)
- Nella versione vettorizzata, la prima iterazione copre 0-4. Leggeremo il vecchio valore di B[2], prima di scrivercelo: quindi è sbagliato.

In questi casi il compilatore non può vettorizzare. Se invece non si sovrappongono, possiamo dirlo al compilatore tramite pragmi o attributi.

Un loop si può rendere vettorizzabile se:

- Si separa (distribuisce) gli statement non in cicli
- Rimuovere le dipendenze
- Congelare i loop, semplificare il numero di iterazioni
- Cambiare l'algoritmo, ad esempio da ricorsivo a iterativo

In generale, se un'istruzione (statement) è dentro ad un loop che non è un ciclo di dipendenze (LCD), può essere vettorizzata.

Further Hinting and Pragmas

Il compiler Intel è capace di riconoscere che alcune condizioni creano dipendenze RAW. Ma il compilatore non è interamente impossibilitato dal vettorizzare un loop. Al suo posto, produce un codice multiversioned che può fare branch su un loop vettorizzato o non vettorizzato, in base al valore della condizione; poi durante l'esecuzione sceglie la versione giusta.

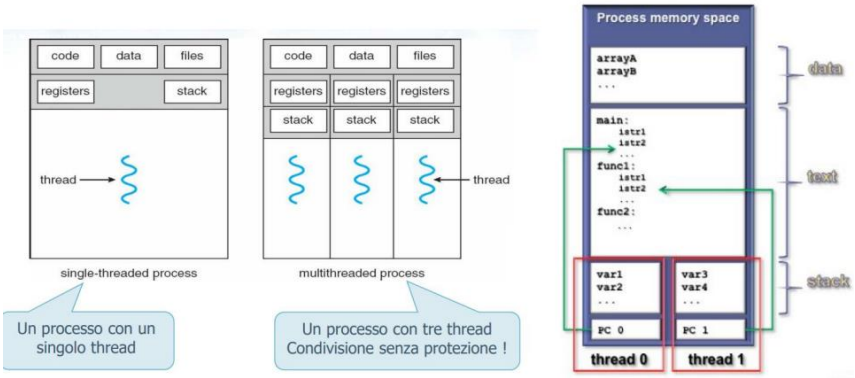
Parallelismo nel multithreading

Come abbiamo detto prima, la categoria del multithread è la MIMD.

<u>Cos'è un thread?</u> Un thread è un sottoprocesso, una sezione di processo, un flusso di esecuzione lanciato da un processo principale. L'esecuzione del thread è indipendente dal processo che "padre" l'ha lanciato e dagli eventuali fratelli.

POSIX è l'acronimo di Portable Operating System Interface, dove la X simboleggia l'eredità UNIX delle API. Il compito dello standard POSIX è quello di definire alcuni concetti base che vanno seguiti durante la realizzazione del sistema operativo.

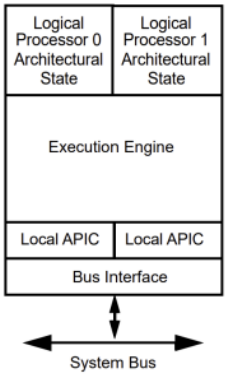
I POSIX Threads, solitamente denominati pthreads, sono un modello di esecuzione che esiste indipendentemente da un linguaggio di programmazione, nonché un modello di esecuzione parallelo. Consente a un programma di controllare più flussi di lavoro diversi che si sovrappongono nel tempo. Ogni flusso di lavoro viene definito thread e la creazione e il controllo di questi flussi si ottiene effettuando chiamate all'API dei thread POSIX.



Un processo ha uno spazio di memoria separato, mentre i thread condividono la memoria e le risorse con gli altri thread dello stesso processo.

Lo scopo è quello di condividere le unità funzionali di un singolo core per ottimizzarne l'uso, condividendole appunto tra più thread.

La tecnica chiave è SMT (Simultaneous Multithreading). Nei processori superscalari ho "troppe" unità funzionali per un singolo thread, per cui mixo istruzioni di thread diversi continuamente. Ho però bisogno di supporto hardware: tutti i registri sono duplicati, così si possono intervallare le istruzioni di thread diversi in pipeline. Senza il supporto hardware ho il multithreading software, richiede context switch. Il multithreading hardware viene chiamato da Intel HyperThreading.



Context switch: passaggio da un thread all'altro. Comporta:

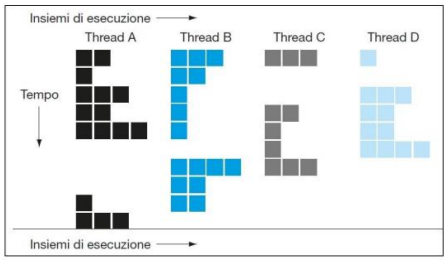
- Sostituzione del valore del PC (Program Counter)
- Sostituzione dei valori dei registri nel register file
- Ogni thread ha il suo spazio dedicato nello stack
- Può essere leggero (SMT – HT cambia la copia del registro) o oneroso (se SW MT)
  (SMT ha più copie dei registri, quindi non deve salvare nulla e basta cambiare quale set di registri sta usando. In SWMT (software multithreading) la CPU ha una sola copia dei registri, quindi quando cambia thread deve salvare tutti i registri del thread A da qualche parte e caricare quelli di B)

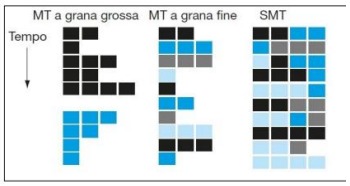
Multithreading a grana fine:

- Ad ogni istruzione si passa da un thread all'altro saltando quelli che sono in stallo
- Nascondo possibili perdite di throughput
- Il context switch dev'essere molto veloce, es due copie di TUTTI i registri
- Il singolo thread però viene ritardato anche se non è in stallo

Multithreading a grana grossa:

- Il thread in esecuzione cambia solo quando si presenta uno stallo importante (L3 miss)
- Non è necessario un context switch veloce





Multicore: il passo successivo oltre il multithreading

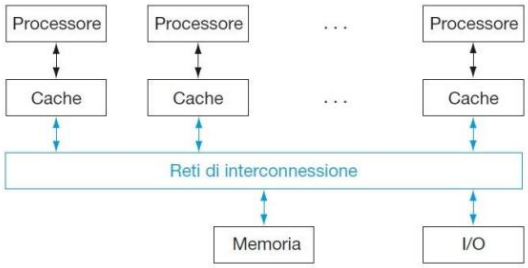
La tecnologia Multithreading hardware simula due processori in un singolo die. Avere due o più core fisici è molto meglio. Un processore multi-core, come implica il nome, contiene due o più core in un singolo die. Dall'esterno appare come una CPU single-core, ma all'interno possono esserci da due a novantasei core. Un processore multi-core fornisce (virtualmente) tutti i vantaggi di avere più processori fisici separati, con costi generali decisamente inferiori. AMD e Intel hanno introdotto i primi processori dual-core x86 per PC desktop nel maggio 2005. La prima soluzione di AMD è stata l'Athlon 64 X2, mentre i primi processori dual-core di Intel rispondono al nome di Pentium Extreme Edition 840 e Pentium D.

Die: Il die è la sottile piastrina di materiale semiconduttore sulla quale è stato realizzato il circuito elettronico del circuito integrato, a sua volta realizzato attraverso un procedimento litografico. Il die sigillato nel suo contenitore, chiamato col termine inglese package (letteralmente "confezione", "pacchetto"), forma un componente elettronico (il circuito integrato). Il die comunica con gli altri terminali tramite i "piedini" (o pin).

I motivi del multicore

Il clock fa in modo che le parti del processore lavorino all'unisono. Più tick hai al secondo e più veloce sarà il lavoro. Ma non può salire troppo, perché più velocità significa anche più consumo e calore da dover dissipare. Per questo. È meglio avere due core più lenti che uno solo più veloce. Underclocking un singolo core del 20% salva metà del consumo, sacrificando solo il 13% della performance. Quindi, se dividi il lavoro tra due core che lavorano all'80% del loro stato, avrai una performance migliore del 73% per la stessa energia; inoltre, il calore verrà dissipato in due posti al posto di uno.

SMP, NUMA e UNA



Un multiprocessore/multicore a memoria condivisa (SMP, Shared Memory Processor) fornisce un unico spazio di indirizzamento, per cui i programmi/thread possono essere allocati liberamente.

Tutti i processi possono accedere alla memoria principale indipendentemente da dove la parola è stata allocata. Il costo di un accesso potrebbe essere sempre lo stesso (UNA, Uniform Memory Access) oppure no (Non UMA, il tempo di accesso cambia a seconda di dove si trova fisicamente la memoria).

Operazioni atomiche

La vettorizzazione ci ha presentato un esempio di parallelizzazione sui dati, il data parallelism. Spesso però i task/thread devono cooperare. Ad esempio, nella ricerca di un valore massimo o minimo, aggiornare un contatore globale, ecc.

Nelle architetture i meccanismi di sincronizzazione sono tipicamente basati su istruzioni hardware in grado di leggere e modificare una locazione di memoria atomicamente.

(Ad esempio Thread1 esegue i+=2, Thread2 esegue i+=3. Quale sarà il risultato finale? RACE CONDITION)

Memory Access – Distributed Memory

I processors hanno la loro propria memoria locale. Gli indirizzi di memoria di un processo non mappano ad altri processi; quindi, non esiste il concetto di spazi globali tra più processi.

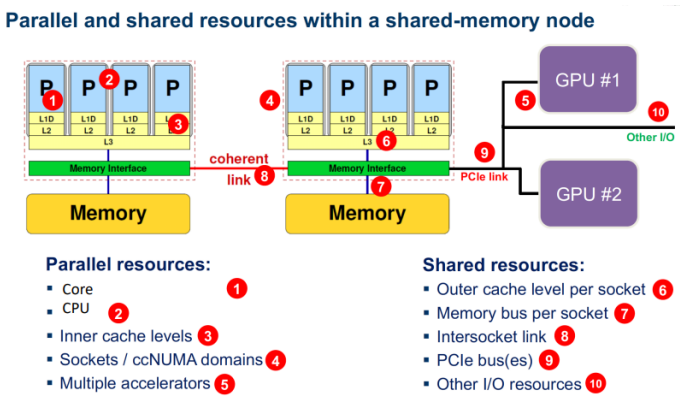
Ogni processo lavora nella sua memoria locale e opera indipendentemente. Quando un processore ha bisogno di dati da un altro processore, è di solito compito del programmatore di definire dove e quando questo dato sarà comunicato. La sincronizzazione tra compiti è quindi responsabilità del programmatore.

I network per trasferire i data sono vari, ma possono essere semplici come l'Ethernet.

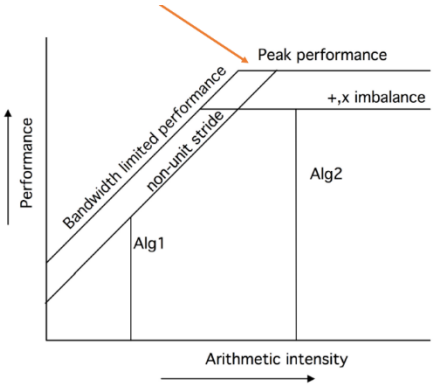
Levels of parallelism

- Instruction level (es FMA= fused multiply and add, a*b+c)
- Vector processing (SIMD, vettorizzazione)
- Hyperthreading (es 4 hardware threads/core for Intel KNL, 8 for PowerPC)
- Cores / processor (es 18 for Intel Broadwell)
- Processors (or sockets) / node
- Processors + accelerators (es CPU+GPU)
- Nodes in a system
- Many systems (es Cloud)

Per arrivare al massimo delle performance di un parallel computer, tutti i livelli di parallelismo devono essere usati. Ogni livello aggiunge più potenza di calcolo, ma anche più complessità alla programmazione.



Roofline model



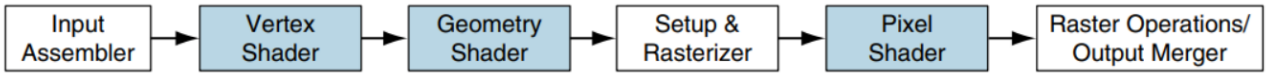
Il roofline model serve a valutare le prestazioni di un programma su una specifica microarchitettura.
Il concetto chiave è quello dell'intensità aritmetica: sia n il numero di elementi di input (di una operazione/funzione/programma), f(n) il numero delle operazioni, l'intensità aritmetica è pari ad f(n)/n (numero operazioni / numero dati usati). Il modello correla intensità aritmetica, prestazioni di picco ottenibili della microarchitettura e prestazioni ottenute effettivamente.
Le prestazioni di picco sono un limite assoluto della microarchitettura. È un limite teorico massimo, in quanto si può ottenere se e solo se tutte le caratteristiche della microarchitettura sono usate in modo perfetto. Tale numero si può ottenere o come calcolo teorico o tramite benchmark. I fattori che impediscono di raggiungere il picco sono load umbalancing (sbilanciamento del carico), utilizzo della cache subottimale…

La GPU

La GPU (Graphics Processing Unit) è un processore altamente parallelo con migliaia di core e memoria privata ad altissima banda (circa 900GB/s). Nata negli anni 90 per grafica 3D ad alta definizione (per gaming, animazione, ecc).
Le GPU sono progettate per renderizzare scene 3D complesse, composte da milioni di dati, punti e vertici ad un alto frame rate (60-120 FPS). Il processo di rendering richiede trasformazioni basati su operazioni di algebra lineare e filtri. Lo stesso set di operazioni è applicato ad ogni data point della scena; ogni operazione è indipendente rispetto ai dati; le operazioni sono performate in parallelo, usando un grande numero di threads che processano tutti i dati indipendentemente.

Graphics Logical Pipeline

L'applicazione 3D manda alla GPU una sequenza di vertici raggruppati in primitive geometriche: punti, linee, triangoli e poligoni (polygons). Il vertex shader mappa la posizioni dei vertici dei triangoli per cambiare posizione, colore e orientamento; la geometry shader prende primitive geometriche, come linee e triangoli, e crea primitive addizionali; il pixel shader frammenta ogni shade in pixel.
L'hardware delal GPU crea un nuovo thread indipendente per eseguire i programmi vertex, geometry e pixel shader per ogni vertice, primitiva e ogni frammento di pixel.

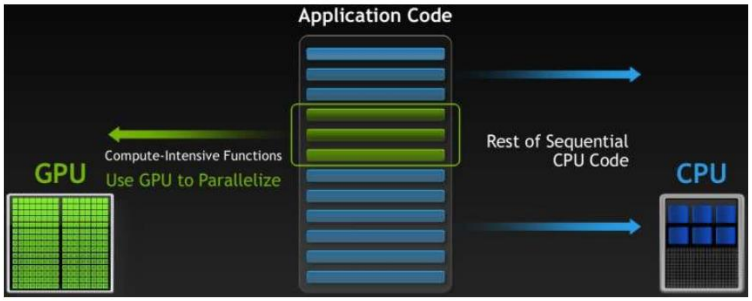


Le architetture GPU unificate sono basate su un array parallelo di molti processori programmabili. Il processo di vertex, geometry e pixel shader e parallel computing sono unificati nello stesso processore, ma un tempo erano in due processori separati.

Modello di programmazione GPGPU

La programmazione GPGPU (General Purpose GPU) usa il potere computazionale della GPU per solvere problemi oltre la grafica. CPU e GPU sono separati e hanno memorie separate. GPU è vista come un coprocessore ausiliario, equipaggiato con migliaia di core e memoria ad alta banda. Quando lavorano insieme, le performance migliorano.

La CPU (host) gestisce le sezioni seriali del programma, o quelli con un basso livello di parallelismo; la GPU (device) gestisce le sezioni data-parallel, che sono computazionalmente intensive. Una funzione che runna sulla GPU è chiamata kernel.



Le GPU sono state create per fare calcoli numerici, e non performeranno bene in altre task. Le applicazioni dovrebbero usare sia la CPU che la GPU.

Modello di programmazione CUDA (linguaggio)

CUDA (Compute Unified Device Architecture) è stato introdotto da NVIDIA nel 2007, e supporta l'esecuzione parallela di applicazioni su CPU e GPU.

È un insieme di estesioni ad un linguaggio di programmazione ad alto livello per usare la GPU come un coprocessore per eseguire compiti paralleli pesanti.

Le sezioni seriali del codice sono eseguite dalla CPU (host) e le parallele dalla GPU (device) in mod SIMD come kernel CUDA. Host e device hanno spazi di memoria separati e il programmatore trasferisce i dati tra CPU e GPU in maniera simile al passaggio di messaggi 'one-sided'.

Un linguaggio alternativo è OpenCL: standard aperto, multi piattaforma ma più basso livello.

Tra software e hardware: memoria, cache e sw/hw multithreading

Gerarchie di memoria

I sistemi di memorizzazione sono organizzati gerarchicamente, secondo velocità, costo e volatilità. La memoria principale è la RAM (Random Access Memory) ed è la memoria che la CPU può accedere direttamente; la memoria secondaria è quella su Disk (Hard Disk, SSD, ecc), ossia un'estensione della memoria principale che fornisce una memoria non volatile, ossia permanente.

- **SRAM (Static Random Access Memory)**
  È un'unità che memorizza un gran numero di parole in un insieme di flip-flop, opportunamente connessi, mediante un sistema di indirizzamento e trasferimento (lettura/scrittura) di parole. I dati memorizzati vengono mantenuti per un tempo arbitrario, finché c'è alimentazione. Il tempo di accesso tipico è di circa 10-30 nanosecondi.
- **DRAM (Dynamic Random Access Memory)**
  Nelle DRAM i bit vengono espressi sotto forma di stato di carica di un transistor, che va rinfrescato periodicamente con un circuito di controllo integrato sul componente di memoria. Usando meno transistor, la densità di bit memorizzati in una DRAM è molto più alta rispetto ad una SRAM e quindi la DRAM ha un costo inferiore. Il tempo di accesso tipico è di circa 60-70 nanosecondi.
- **HDD (Hard Disk)**
  È una memoria persistente con dischi magnetici e testine di lettura/scrittura. Il costo per bit è molto più basso con grande capacità di memorizzazione. Il tempo di accesso tipico è di circa 10 millisecondi.
- **SSD/Flash (Solid State Drive)**
  È una tecnologia basata su celle di semiconduttori che non richiede spostamenti meccanici e che mantiene lo stato anche senza alimentazione. Il tempo di accesso tipico è di circa 0.1 millisecondi.

Principio di località

- <u>Località temporale:</u> quando si fa riferimento ad un elemento di memoria, tendenzialmente si effettueranno subito altre operazioni sullo stesso elemento (esempio: lettura e scrittura dell'indice in un ciclo for).
- <u>Località spaziale:</u> quando si fa riferimento ad un elemento di memoria, tendenzialmente si effettueranno subito dopo operazioni anche su altri elementi vicini (esempio: lettura di celle di un array).

Durante l'esecuzione di un programma, solitamente si passa da una località all'altra.

Ogni coppia di livelli di una gerarchia è formata da un livello superiore, con accesso più rapido e capacità minore, e da un livello inferiore, con accesso più lento e capacità maggiore.
L'unità di memorizzazione nella cache viene detta linea o blocco di cache.

Cache hit/miss

In lettura si cerca prima il dato nel livello superiore:

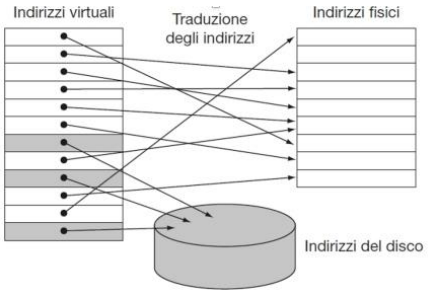
- Se il riferimento (indirizzo memoria) è già presente nella cache (cache hit), si procede con la lettura della linea di cache
- Altrimenti occorre trasferire i dati dal livello inferiore (cache miss) ed inserire una nuova linea di cache, eventualmente sovrascrivendo altre linee di cache (riferimenti e dati usati meno recentemente)

Un'operazione di scrittura può generare una cache miss nel caso in cui il dato non sia presente in cache. La scrittura pone anche il problema della consistenza tra dati in cache e dati nel livello inferiore.
Si possono usare diverse strategie:

- Scrivere in cache e in memoria (**write-through**)
- Scrivere in cache e schedulare scrittura su disco tramite buffer gestito da controller dedicato (**write buffer**)
- Scrivere solo nella cache e copiare i dati in memoria solo quando la linea o il blocco deve essere sovrascritto (**write back**) a causa di cache miss di altri riferimenti

RAM come cache per pagine di memoria di processi

La RAM può contenere diversi programmi in esecuzione e quindi lo spazio di memoria è suddiviso tra di essi. Lo spazio di indirizzamento di un programma è virtuale e più grande dello spazio allocato nella RAM. La RAM funge quindi da cache per i programmi in esecuzione (processi): lo spazio di indirizzamento dei processi è diviso in pagine caricate quando necessario.



Ci sono due classi di MIMD: Shared Memory MIMD e Distributed Memory MIMD

Shared Memory MIMD

Le architetture Shared Memory MIMD sono la base del multicore programming. Nelle architetture multicore i processi hanno diversi livelli di cache e condividono la memoria principale e un buffer di I/O. Diversi programmi, eseguiti simultaneamente, possono condividere dati in memoria principale attraverso tecniche di **threading** hardware e software. In questo caso la gestione della cache diventa molto complessa e richiede la garanzia della coerenza tra i valori mantenuti nelle cache dei diversi processori e quelli in memoria.

Multithreading

<u>Software</u>

Il multithreading a livello di software viene gestito a livello di sistema operativo. In questo contesto un singolo programma ha diversi flussi di istruzioni (servono diversi program counter) che condividono la stessa memoria ed effettuano operazioni simultaneamente (parallelismo reale o virtuale). Lo scheduler del sistema operativo si occupa di definire l'ordine delle istruzioni da eseguire.
Se i dati su cui operano i diversi flussi di istruzioni dello stesso programma sono indipendenti tra loro (esempio: parti di matrici), ogni thread potenzialmente può essere eseguito in parallelo con altri thread sullo stesso processore (parallelismo virtuale) o su diversi core (parallelismo reale).

<u>Hardware</u>

Chip-level multiprocessing (CMP o multicore) integra due o più processori in un chip, e ognuno esegue thread indipendentemente.
I processori superscalari con ILP (Instructio Level Parallelism) e SMT supportano hardware multithreading: in caso di cache miss (stallo) è possibile passare all'esecuzione di un altro thread sullo stesso core.

Intel Hyper-Threading: tecnologia di SMT introdotta da Intel nel 2002; si trattava di un tentativo di creare un processore di transizione tra i tradizionali single core e i successivi dual core, non inserendo 2 interi core all'interno di un unico package, ma duplicando solo alcune aree 'sensibili' del singolo core. Grazie all'Hyper-Threading un singolo core era comunque in grado di gestire due thread in contemporanea. Il sistema operativo riconosce quindi un physical core come 2 virtual (logical) core.

Programmi, processi e thread

Un processo è un'attività controllata da un programma che si svolge su un processore (o più semplicemente un programma in esecuzione). Un processo non è un programma. Un programma specifica la sequenza di esecuzione di un insieme di istruzioni ma non la distribuzione dell'esecuzione. Un processo rappresenta il modo in cui un programma viene eseguito nel tempo. Più processi possono eseguire lo stesso programma (varie istanze dello stesso programma). Ogni istanza viene considerata come un processo separato anche se possono condividere lo stesso codice mente i dati, l'immagine e lo stato rimangono separati.

Dal codice sorgente al codice eseguibile

Compile-time

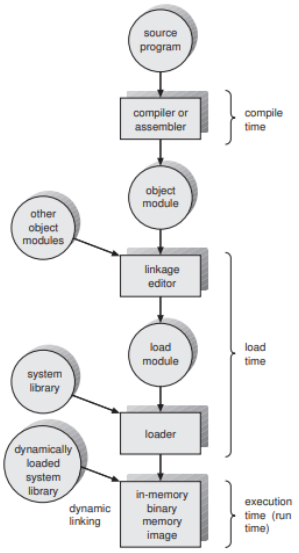
- Compilazione: il codice scritto nel linguaggio di alto livello viene tradotto in codice in linguaggio macchina
- Linking: il codice delle librerie importate viene incluso nel codice oggetto

Loading-time

- Il codice binario viene caricato in memoria principale

Execution-time

- Il codice oggetto viene eseguito dalla CPU (che può inviare richieste di lettura/scrittura alla memoria principale)



Indirizzi logici e indirizzi fisici

Spazio di indirizzamento logico

- Ogni processo è associato ad uno spazio di indirizzamento logico
- Gli indirizzi usati in un processo sono indirizzi logici, ovvero riferimenti a questo spazio di indirizzamento
- In generale indirizzi logici = indirizzi di memoria generati dalla CPU

Spazio di indirizzamento fisico

- Ad ogni indirizzo logico corrisponde un indirizzo fisico
- Il sistema operativo (e/o hardware) si occupa della traduzione da indirizzi logici a indirizzi fisici

Binding

Con il termine binding si indica l'associazione di indirizzi di memoria ai dati e alle istruzioni di un programma. Il binding può avvenire durante il caricamento o durante l'esecuzione.

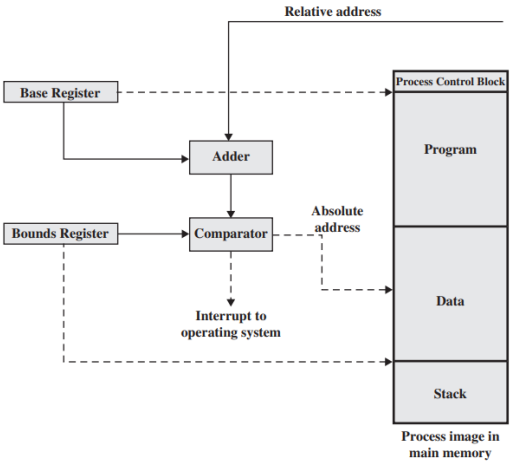
- **Binding durante il caricamento:** il codice generato dal compilatore non contiene indirizzi assoluti ma relativi: questo tipo di codice viene detto rilocabile. Il programma 'loader' si preoccupa di aggiornare tutti i riferimenti agli indirizzi di memoria coerentemente al punto iniziale del

caricamento. Permette di gestire multiprogrammazione e non richiede l'uso di hardware particolare. Richiede la traduzione degli indirizzi da parte del loader, quindi un formato particolare dei file eseguibili.

- **Binding durante l'esecuzione:** l'individuazione dell'indirizzo di memoria effettivo viene effettuata durante l'esecuzione da un componente hardware apposito chiamato memory management unit (MMU). Il programma può essere spostato da una zona all'altra della memoria durante l'esecuzione.

MMU

Un esempio di MMU si basa sui registri di riallocazione e di limite; il registro di limite viene utilizzato per implementare meccanismi di protezione della memoria. Ad esempio, se l'indirizzo logico è maggiore del valore del registro limite si genera un errore. Se si passa il test con il registro limite si utilizza il registro di rilocazione per calcolare l'indirizzo fisico: se il valore del registro di riallocazione è R, uno spazio logico 0,…,max viene tradotto in uno spazio fisico R,…,R+max.



Loading (caricamento) dinamico

Consente di poter caricare alcune routine di libreria solo quando vengono richiamate. Tutte le routine a caricamento dinamico risiedono sul disco (codice rilocabile). Le routine poco utili non vengono caricate in memoria al caricamento dell'applicazione. Il sistema operativo fornisce semplicemente una libreria che implementa le funzioni di caricamento dinamico.

Linking (collegamento) dinamico

<u>Linking statico:</u> le routine di libreria vengono copiate in ogni programma che le usa (es. printf in tutti i programmi C)

<u>Linking dinamico:</u> è possibile posticipare il linking delle routine di libreria al momento del primo riferimento durante l'esecuzione e quindi consente di avere eseguibili più compatti. In questo caso le librerie vengono implementate come codice reentrant: esiste una sola istanza della libreria in memoria e tutti i programmi eseguono il codice di questa istanza.

Processi e sistema operativo

Il sistema operativo tiene traccia di tutti i processi del sistema tramite la tabella dei processi. Le entry della tabella, Process Control Block (PCB), mantengono informazioni sui singoli processi. Lo scheduler gestisce l'avvicendamento dei processi in CPU: decide quale processo deve essere in esecuzione ogni istante; interviene quando viene richiesta un'operazione di I/O e quando un'operazione di I/O termina, ma anche periodicamente per assicurare il buon funzionamento del sistema.

Switch di contesto

Quando la CPU passa da un altro processo, il sistema deve salvare lo stato del vecchio processo e caricare quello del nuovo processo. Lo stato del processo viene salvato nel PCB. Il tempo di context-switch porta un certo overhead; il sistema non fa lavoro utile mentre passa di contesto. Questo può essere un collo di bottiglia per sistemi operativi ad alto parallelismo. Il tempo impiegato per lo switch dipende dal supporto hardware (per il salvataggio dei valori dei registri, ecc)

Software Multithreading

Riassumendo, i processi dei sistemi operativi le seguenti caratteristiche:

- Rappresentano unità di allocazione risorse: codice eseguibile, dati allocati staticamente (variabili globali) ed esplicitamente (heap), risorse mantenute dal kernel (file, I/O, working dir), controlli di accesso…
- Rappresentano unità di esecuzione: un percorso di esecuzione attraverso uno o più programmi: stack di attivazione (variabili locali), stato (running, waiting, ready, …), priorità, parametri di scheduling, …

Queste due componenti si possono considerare in maniera separata.

Un **thread** (o processo leggero, lightweight process) è un'unità di esecuzione che prevede program counter (insieme registri), stack del processore, stato di esecuzione. Il thread condivide con i thread suoi pari un'unità di allocazione risorse, che contiene il codice eseguibile, i dati e le risorse richieste al sistema operativo.

task = unità di risorse + thread che vi accedono

Condivisione di risorse tra i thread

Vantaggi:

- Maggiore efficienza
- Creare e cancellare thread è più veloce (100-200 volte): meno informazione da duplicare/creare/cancellare
- Lo scheduling tra thread dello stesso processo è molto più veloce che tra processi
- Cooperazione di più thread nello stesso task porta maggiore throughput e performance (es. in un file server multithread, mentre un thread è bloccato in attesa di I/O, un secondo thread può essere in esecuzione e servire un altro client)

Svantaggi:

- Maggiore complessità di progettazione e programmazione: i processi devono essere pensati paralleli, sincronizzazione tra thread, gestione dello scheduling, …
- Inadatto per situazioni in cui i dati devono essere protetti

Ottimi per processi cooperanti che devono condividere strutture dati o comunicare (e.g., produttore–consumatore, server, …): la comunicazione non coinvolge il kernel.

Esempi di applicazioni multithread

- <u>Lavoro foreground/background:</u> mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background
- <u>Elaborazione asincrona:</u> operazioni asincrone possono essere implementate come thread. Es: salvataggio automatico
- <u>Task intrinsecamente paralleli:</u> vengono implementati ed eseguiti più efficientemente con i thread. Es: file, http, ftp server, …

User Level Thread (ULT)

Stack, program counter e operazioni su thread sono implementati in librerie a livello utente. Vantaggi:

- Efficiente: non c'è il costo della system call
- Semplici da implementare su sistemi preesistenti
- Portabile: possono soddisfare lo standard POSIX 1003.1c (pthread)
- Lo scheduling può essere studiato specificatamente per l'applicazione

Svantaggi:
- Non c'è scheduling automatico tra i thread
- L'accesso al kernel è sequenziale
- Non sfrutta sistemi multiprocrssore
- Poco utile per processi I/O bound

Kernel Level Thread (KLT)
Il kernel gestisce direttamente (o almeno in parte) i thread. Le operazioni sono ottenute attraverso system call.
Vantaggi:
- lo scheduling del kernel è per thread, non per processo ⇒ un thread che si blocca non blocca l'intero processo
- Utile per i processi I/O bound e sistemi multiprocessor

Svantaggi:
- meno efficiente: costo della system call per ogni operazione sui thread
- necessita l'aggiunta e la riscrittura di system call dei kernel preesistenti
- meno portabile
- la politica di scheduling è fissata dal kernel e non può essere modificata

Esempi: thread in Unix e Linux (parzialmente)

Processi in UNIX
Ogni processo UNIX (identificato dal PID - process identifier) ha uno spazio di indirizzamento separato e quindi non vede le zone di memoria dedicate agli altri processi. Un processo UNIX ha tre segmenti: stack, dati – divisi in Dati statici e Heap – e codice. L'indirizzamento è virtuale: codice, dati e heap sono memorizzati nella parte iniziale della memoria virtuale, lo stack nella parte finale.
L'unico modo di creare nuovi processi in Unix è attraverso la funzione di sistema **fork**. La chiamata fork() dal processo P (padre) crea un nuovo processo F (figlio) che viene eseguito in parallelo con il padre. Dopo la creazione del figlio, padre e figlio condividono lo stesso codice, inoltre il figlio ha una copia dei dati e del program counter del padre. Il figlio proseguirà l'esecuzione a partire dall'istruzione che segue la fork (cioè lo stesso punto nel quale si trova il padre).

Per distinguere padre e figlio se la chiamata fork() ha successo allora restituisce:
- il PID del figlio al processo padre
- 0 al figlio

Il valore di ritorno di fork viene usato quindi per ridefinire il comportamento del figlio. Tipicamente, infatti, il figlio sostituirà l'immagine del processo padre con un nuovo programma attraverso la chiamata di sistema alla funzione execve che prende come parametro il nome di un file eseguibile.
La funzione execve() sostituisce dati, codice e stack con quelli del nuovo programma da eseguire nel contesto del processo figlio. Il padre può aspettare la terminazione del figlio utilizzando la funzione di sistema waitpid.

Creazione di un processo
La fork alloca spazio per il processo figlio: nuove tabelle per la memoria virtuale; nuova memoria per segmenti dati e stack; dati, stack e u-structure vengono copiati da quelli del padre preservando file aperti,

UID e GID, ecc.; il codice viene condiviso. La execve non crea nessun nuovo processo: semplicemente, i segmenti dati, codice e stack vengono rimpiazzati con quelli del nuovo programma.

User and Kernel Mode

I processi Unix operano in modo user e kernel: cioè il kernel esegue nel contesto di un processo le operazioni per gestire chiamate di sistema e interrupt. Alla partenza del sistema il codice del kernel viene caricato in memoria principale. Un processo in esecuzione in modo user non può accedere allo spazio di indirizzi del kernel. Quando un processo passa ad eseguire in modo kernel tale vincolo viene rilasciato per eseguire codice kernel (es. routine di gestione di interrupt/codice di una chiamata di sistema) nel contesto del processo utente.
Contesto utente: codice, dati, stack, registri
Contesto Kernel: entry nella tabella dei processi, u-area, stack kernel

Livelli di contesto

La parte dinamica del contesto di un processo (kernel stack, registri salvati) è organizzata a sua volta come stack con un numero di posizioni che dipende dai livelli di interrupt diversi ammessi nel sistema. Ad esempio, se il sistema gestisce interrupt software, interrupt di terminali, di dischi, di tutte le altre periferiche, e di clock: avremo al pi`u sette livelli di contesto.
Livello 0: User
Livello 1: Chiamate di sistema
Livelli 2-6: Interrupt (l'ordine dipende dalla priorità associata alle interrupt)

Esempio di esecuzione nel contesto di un processo

- Il processo esegue una chiamata di sistema: il kernel salva il suo contesto (registri, program e stack pointer) nel livello 0 e crea il contesto di livello 1
- La CPU riceve e processa un interrupt di disco (il controllo viene fatto prima dell'esecuzione della prossima istruzione): il kernel salva il contesto di livello 1 (registri, stack kernel) e crea il livello 2 nel quale si esegue la routine di gestione dell'interrupt di disco
- La CPU riceve un interrupt di clock: il kernel salva il contesto di livello 2 (registri, stack kernel per la routine di gestione dell'interrupt disco) e crea il livello 3 nel quale si esegue la routine di gestione dell'interrupt di clock
- La routine termina l'esecuzione: il kernel recupera il livello di contesto 2 e così via
- Tutti questi passi vengono fatti sempre all'interno dello stesso processo: cambia solo la sua parte di contesto dinamica

Algoritmo di gestione delle interruzioni

L'algoritmo del kernel per la gestione di un'interrupt consiste dei seguenti passi:

- salva il contesto del processo corrente
- determina fonte dell'interrupt (trap/interrupt I/O/ ecc)
- recupera l'indirizzo di partenza della routine digestione delle interrupt (dal vettore delle interrupt)
- invoca la routine di gestione dell'interrupt
- recupera il livello di contesto precedente

Per motivi di efficienza parte della gestione di interruzioni e trap viene eseguita direttamente dalla CPU (dopo aver seguito un'istruzione): il kernel dipende quindi dal processore.

Trap/Interrupt vs Context switch
Il modo user/kernel permette al kernel di lavorare nel contesto di un altro processo senza dover creare nuovi processi kernel. Con questo meccanismo un processo in modo kernel può svolgere funzioni logicamente collegate ad altri processi (ad es. la gestione dei dati restituiti da un lettore di disco) e non necessariamente collegate al processo che ospita momentaneamente il kernel.
Nota: La gestione di trap/interrupt si basa su una sorta di context switch all'interno di un processo: il controllo non passa ad un'altro processo ma è necessario salvare la parte corrente del contesto dinamico del processo all'interno dello stesso processo

Context Switch
Il kernel vieta context switch arbitrari per mantenere la consistenza delle sue strutture dati. Il controllo può passare da un processo all'altro in quattro possibili scenari:
- Quando un processo si sospende
- Quando termina
- Quando torna a modo user da una chiamata di sistema ma non è più il processo a più alta priorità
- Quando torna a modo user dopo che il kernel ha terminato la gestione di un'interrupt a non è più il processo a più alta priorità

In tutti questi casi il kernel lascia la decisione di quale processo da eseguire allo scheduler.

Programmazione multithread

Programmazione sequenziale
Il modello di esecuzione sequenziale prevede:
- Programmi formati da sequenze di istruzioni (quindi totalmente ordinate)
- Un'unica memoria (virtuale) per i dati dichiarati staticamente e creati dinamicamente
- L'esecuzione di una singola istruzione alla volta seguendo l'ordine del programma

Supponiamo che il nostro elaboratore fornisca una qualche forma di parallelismo.
Parallelismo reale: multi-processore, sistema distribuito
Parallelismo virtuale: multi-programmazione con scheduling
Come possiamo sfruttare questa possibilità a livello di (linguaggio/sistema di) programmazione?

Programmazione Concorrente
- Rilassiamo il modello di calcolo in modo da prevedere l'esecuzione concorrente di un insieme di istruzioni
- Nel nuovo modello le istruzioni di un programma possono essere parzialmente ordinate

Esempi: Calcolare (A + B) ∗ (C + D), calcoli su matrici, schemi divide-et-impera, rendering di un'immagine sullo schermo
Vantaggi:
- Sfruttare le prestazioni di architetture multi-processore
- Migliorare la reattività delle interfacce grafiche
- Ottenere maggiore naturalezza per alcune tipologie di algoritmi e applicazioni (browser, server, giochi, simulatori)
- Migliorare design e comprensione del codice dei sistemi operativi

Definizioni di thread

```
Thread T_1 { Prog_1 }
....
Thread T_n { Prog_n}
```

Il frammento di codice Prog_i è un programma sequenziale e può quindi contenere una sequenza con assegnamenti, if-then-else, repeat, while, ecc.
Ogni thread viene eseguito in concorrenza con gli altri condividendo le variabili globali. L'esecuzione in parallelo non implica l'esecuzione ripetuta dei sottoprogrammi: ad esempio, se Prog_i non ha cicli allora verrà eseguito una sola volta in parallelo con gli altri programmi.

Esempio 1:

```
Risorse
var x=0

Thread P1 { x:=500; }
Thread P2 { x:=0;   }
Thread P3 { write(x); }
```

Nell'esempio 1 ci sono due possibili output da write (del thread P3): 0 o 500. L'esecuzione di questo programma da origine ad un solo valore, ma esecuzioni diverse possono dare origine a risultati diversi. Un programma concorrente definisce una funzione che, dato un input, produce un insieme di possibili output e non un singolo valore come nel caso sequenziale (non determinismo).

Esempio 2:

```
var x=0

Thread P1 {
  while (true)   x:=500;   endwhile;
}

Thread P2 {
  while (true)   x:=0;      endwhile;
}

Thread P3 {
  while (true)  write(x); endwhile;
}
```

Nell'esempio 2 vi sono un numero infinito di possibili output. Ognuno di essi è a sua volta una sequenza finita o infinita $v_1 v_2 v_3 \ldots$ con $v_i \in \{0, 500\}$ per $i \geq 0$. Potrei eseguire solo P1 e P2 e mai P3, eseguire una volta P1 e poi una volta P3, ecc. Ad esempio, un possibile output è 0 500 0 500 0 500… L'esecuzione di tale programma da origine ad una sola sequenza di output (una tra tutte quelle possibili). Esecuzioni diverse possono dare origine a risultati diversi.

Esempio 3:

```
Risorse
var x=100

Thread P1 {
  x:=x+1;
}

Thread P2 {
  x:=x-1;
}
```

- Se le istruzioni sono eseguite in sequenza alla fine x = 100
- Se eseguo P1 fino alla valutazione di x + 1 e prima dell'assegnamento, poi eseguo P2 completamente ed infine eseguo l'assegnamento in P1 posso ottenere x = 101
- Se eseguo P2 fino alla valutazione di x − 1 e prima dell'assegnamento, poi eseguo P1 completamente ed infine eseguo l'assegnamento in P2 posso ottenere x = 99

Problema del produttore-consumatore

Tipico paradigma dei thread cooperanti:

- Il thread produttore produce informazione che viene consumata da un thread consumatore
- Soluzione a memoria condivisa: tra i due thread si pone un buffer di comunicazione di dimensione fissata (Produttore-consumatore con buffer limitato)

Dati condivisi tra i thread

```
type item = ... ;
var buffer: array [0..n-1] of item;
in, out: 0..n-1;
counter: 0..n;
in, out, counter := 0;
```

Thread Produttore

```
while (true) do
    ...
    produce un item in nextp
    ...
    while counter = n do skip endwhile;
    buffer[in] := nextp;
    in := in + 1 mod n;
    counter := counter + 1;
endwhile
```

Thread Consumatore

```
while (true) do
    while counter = 0 do skip endwhile;
    nextc := buffer[out];
    out := out + 1 mod n;
    counter := counter − 1;
    ...
    consuma l'item in nextc
    ...
end
```

Le istruzioni '*counter := counter + 1;*' e '*counter := counter – 1;*' devono essere eseguite atomicamente. Se eseguite non atomicamente possono portare ad inconsistenze.

Race condition

Il Race condition avviene quando **più thread accedono concorrentemente agli stessi dati**, e il risultato dipende dall'ordine di esecuzione di istruzioni selezionate dai diversi thread.

Le race condition si possono verificare frequentemente nei sistemi operativi multitasking, sia per dati in user space sia per strutture in kernel. Sono estremamente pericolose: portano al malfunzionamento dei thread cooperanti, o anche (nel caso delle strutture in kernel space) dell'intero sistema. Sono difficili da individuare e riprodurre: dipendono da decisioni dello scheduler, carico del sistema, utilizzo della memoria, numero di processori, ecc.

Memoria virtuale, Cache e Multicore

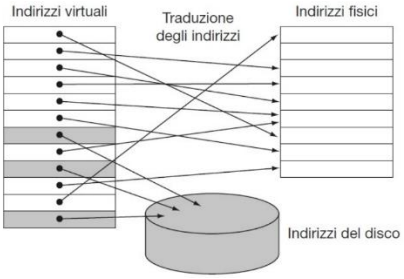
Memoria virtuale

È la tecnica che permette l'esecuzione di processi che non sono completamente caricati in memoria principale. Permette di eseguire in concorrenza processi che nel loro complesso (o anche singolarmente) hanno necessità di una memoria maggiore di quella disponibile. Ciò permette di separare la memoria logica vista dall'utente dalla memoria fisica.

Le istruzioni da eseguire e i dati su cui operano devono essere in memoria principale (Von Neumann); tuttavia, non è necessario che l'intero spazio di indirizzamento logico di un processo sia nella memoria principale, in quanto i processi non utilizzano tutto il loro spazio di indirizzamento contemporaneamente.

Per implementare una memoria virtuale:

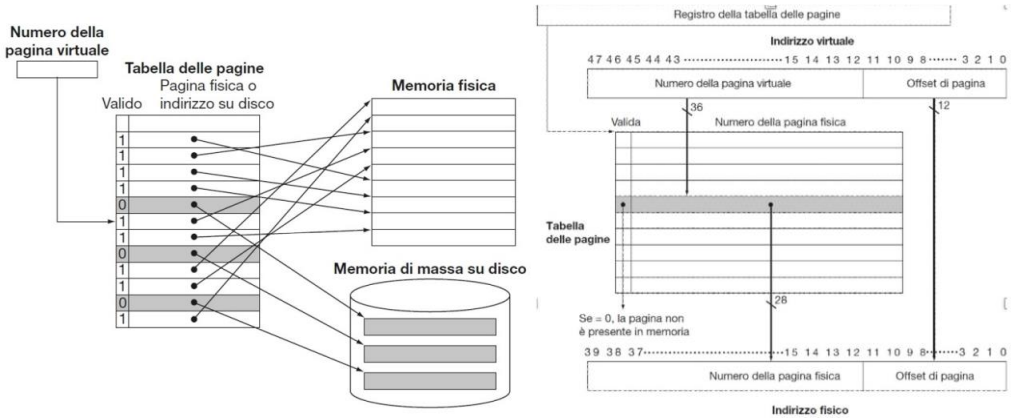
- Paginazione su richiesta (demand paging): si utilizza la tecnica della paginazione, ammettendo però che alcune pagine possano essere in memoria secondaria
- Nella tabella delle pagine di ogni singolo processo si utilizza un bit, il bit di validità, che indica se la pagina è presente o meno nella memoria principale
- Quando un processo tenta di accedere ad una pagina non in memoria, il processore genera una trap (page fault) e un componente del SO, il pager, si occupa di caricare la pagina mancante e di aggiornare di conseguenza la tabella delle pagine



Nella memoria virtuale i blocchi della memoria (pagine) vengono mappati da un insieme di indirizzi (detti virtuali) a un altro insieme di indirizzi (detti fisici). Il processore produce indirizzi virtuali mentre l'accesso alla memoria avviene utilizzando gli indirizzi fisici. Sia la memoria virtuale sia quella fisica sono suddivise in pagine, in modo tale che una pagina virtuale viene mappata su una pagina fisica. Se una pagina virtuale non è presente nella memoria principale (e quindi non può essere mappata su una pagina fisica), si troverà sul disco. Le pagine fisiche possono essere condivise facendo in modo che due indirizzi virtuali puntino allo stesso indirizzo fisico. Questa possibilità permette a due diversi programmi di condividere dati e parti di codice.

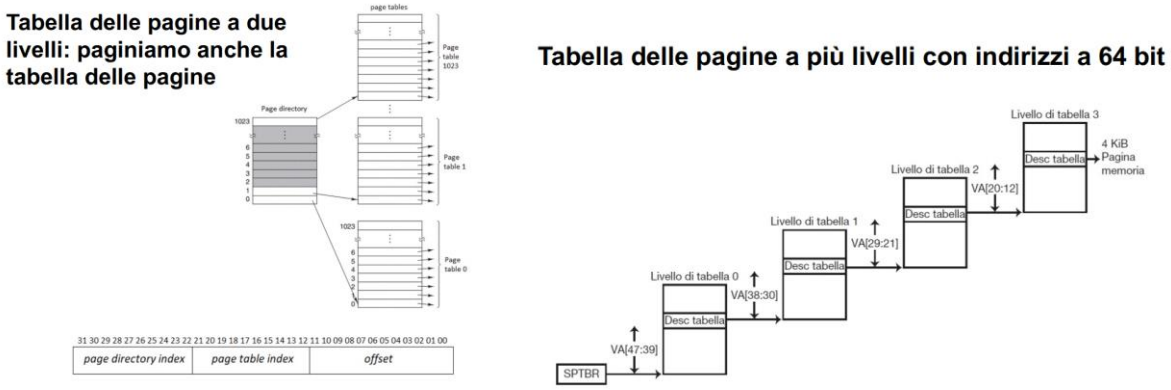
Gli indirizzi virtuali sono da 48 bit (36 bit per l'indirizzo di pagina e 12 per l'offset), e gli indirizzi fisici sono da 40 bit (28 bit per l'indirizzo di pagina e 12 per l'offset).

Tabella delle pagine: Per gestire mappatura, il sistema operativo mantiene una tabella delle pagine per ogni processo con un bit di validità.



Indirizzi fisici da 40 bit → indirizzo di pagina 12 bit

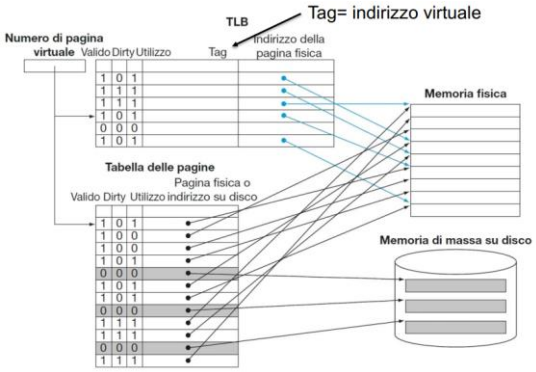
Indirizzi virtuali da 48 bit → indirizzo di pagina 36 bit



TLB: Caching applicato alla tabella delle pagine

La paginazione della Page Table è utile per razionalizzare l'uso della memoria principale ma introduce overhead nella fase di risoluzione degli indirizzi virtuali.

Viene utilizzata quindi una memoria cache dedicata chiamata Transaction Lookahead Buffer (TLB) per mantenere accesso diretto ad un numero fisso di elementi della tabella delle pagine. Nella TLB si associa ad un indirizzo virtuale, il corrispondente indirizzo fisico e dei flag che indicano se la pagina è stata usata di recente (use bit) o se ci sono state modifiche recenti (dirty bit).



Quando la CPU vuole accedere ad un indirizzo virtuale, prima controlla nella TBL. Se trova la corrispondenza (TLB hit) può saltare tutta la page table. Se non la trova, TLB miss. Con le TLB bisogna gestire due diversi tipi di miss:

miss nella TLB (indirizzo logico non è nella cache) → page table

miss nella tabella delle pagine (validità=0) → page fault

Se l'indirizzo virtuale non è presente nella TBL, si va a cercare nella Page Table (tabella delle pagine) la traduzione dell'indirizzo virtuale. Se la voce nella page table è valida (valid bit = 1), si ottiene l'indirizzo fisico e si aggiorna la TLB inserendo la nuova voce (così che la prossima volta sia più veloce).

Quando accedi ad un indirizzo virtuale che non ha mappatura nella tabella delle pagine (es. non è in RAM ma sul disco), il sistema genera un Page Fault, ossia un blocco momentaneo in cui carica la pagina dal disco alla RAM. Aggiorna poi la tabella delle pagine e poi il TLB

Memoria virtuale e cache lavorano insieme: un dato non può trovarsi nella cache se non è presente in memoria (il sistema operativo mantiene consistenza) e di conseguenza vengono modificate Page Table e TLB (per generare page fault al prossimo accesso).

Esistono diverse strategie per decidere quali indirizzi mantenere in una cache (es TLB) che vengono chiamate politiche di rimpiazzamento: ad esempio LRU (Least Recently Used) seleziona indirizzi in TLB con bit di uso a 0 eventualmente combinata con altre strategie (es una casuale) in caso tutte le pagine fossero indicate come in uso.

Conflitti nell'uso della cache

Se il blocco da portare in cache deve essere sovrascritto (sulla base della funzione di mapping) su un altro blocco di dati già presente in cache, cosa ne faccio del vecchio blocco?

- Se il vecchio blocco è stato solo acceduto in lettura (Read), possiamo semplicemente rimpiazzarlo
- Se il vecchio blocco è stato modificato (Read/Write), dipende dalle politiche di coerenza tra i livelli di memoria

Write through (scrivo sia in cache che in memoria)

Write back (scrivo solo in cache, e ritardo la scrittura in memoria)

Con la politica Write through il blocco in conflitto può essere rimpiazzato senza problemi; con la politica Write back prima di rimpiazzare il blocco in conflitto, questo deve essere scritto in memoria.

Write buffer

Le write pongono problemi con politica write-through. Le write diventano più lunghe, anche in presenza di Write Hit.

Il Write buffer funziona come memoria tampone «veloce» tra cache e memoria, per nascondere la latenza di accesso alla memoria: i blocchi sono scritti temporaneamente nel write buffer, in attesa della scrittura asincrona in memoria. Il processore può quindi proseguire senza attendere, a meno che il write buffer sia pieno.

Problema della coerenza della cache

Cache L1: La cache L1 (nota anche come cache primaria o cache di livello 1) è la cache più alta nella gerarchia dei livelli di cache di una CPU. È la cache più veloce nella gerarchia. Ha una dimensione minore e un ritardo minore (stato di attesa zero) perché di solito è integrato nel chip.

Cache L2: La cache L2 (nota anche come cache secondaria o cache di livello 2) è la cache che si trova accanto a L1 nella gerarchia della cache. Di solito si accede a L2 solo se i dati cercati non si trovano in L1. L2 viene solitamente utilizzato per colmare il divario tra le prestazioni del processore e la memoria.

Coerenza della memoria

Un sistema di memoria è coerente se la lettura di un dato restituisce il valore che è stato scritto più di recente in quella cella di memoria. Abbiamo visto che nel multicore programming si possono verificare data race. In presenza di più livelli di cache le cose si complicano ulteriormente.

Passo	Evento	Contenuto della cache della CPU A	Contenuto della cache della CPU B	Contenuto della memoria nella locazione X
0				0
1	CPU A legge X	0		0
2	CPU B legge X	0	0	0
3	CPU A scrive 1 in X	1	0	1

Figura 5.40 Il problema della coerenza delle cache, illustrato con una locazione di memoria (X), che viene letta e scritta da due processori (A e B). Supponiamo inizialmente che nessuna delle due cache contenga dei dati e che la locazione X della memoria principale contenga il valore 0. Supponiamo anche che la cache sia di tipo write-through; una cache di tipo write-back aggiunge alcuni problemi, ma della stessa natura. Dopo che la CPU A ha scritto il valore 1 nella locazione X della memoria, la memoria e la cache A contengono entrambe il valore aggiornato, mentre la cache B contiene ancora 0: se B richiedesse il contenuto della locazione X, leggerebbe quindi il valore 0!

Funzionalità richieste ad un processore multicore

- Migrazione: un dato viene trasferito da una cache all'altra (ad esempio se CPU A scrive X trasferisco nuovo dato alla cache della CPU B)
- Replicazione: quando un dato viene letto si memorizza una copia in ogni cache

Snoopy Cache Coherence Protocol: l'idea è quella di assicurare che un processore abbia accesso esclusivo ad un dato prima di modificarlo (mutua esclusione come per controllo delle data race nei thread Possibile soluzione: write invalidate protocol: Quando la CPU A deve modificare X, manda un segnale di invalidazione per tale dato (linea di cache, quindi con match su tag) a tutte le altre cache prima di effettuare la scrittura. In questo modo esiste sempre una sola copia valida del dato nella cache della CPU A.