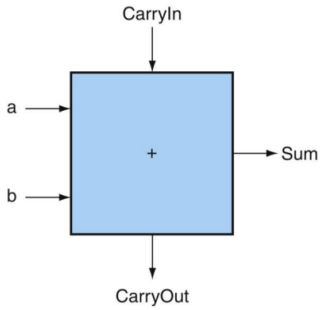
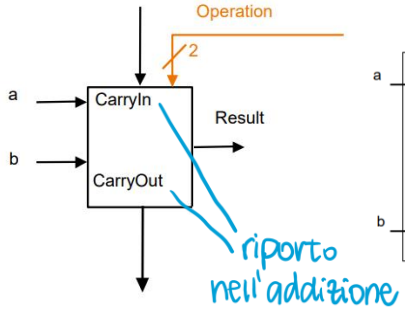
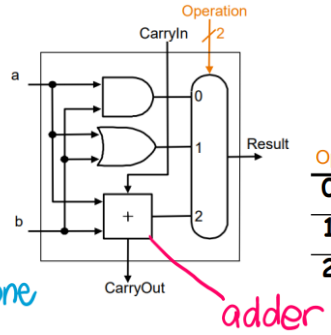
Costruzione di una Arithmetic Logic Unit (ALU)

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.

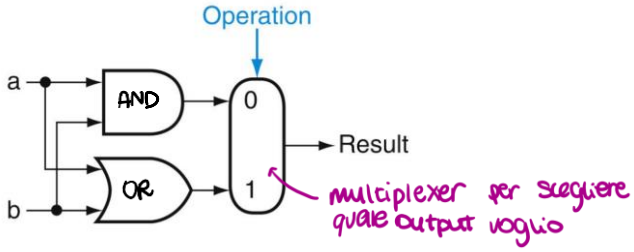




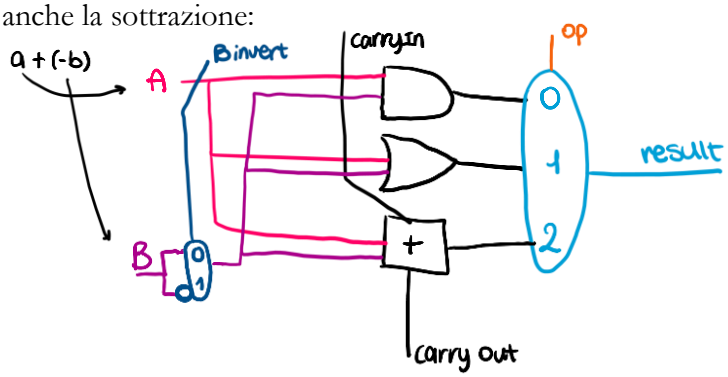


Operation	Function
0	AND
1	OR
2	ADD

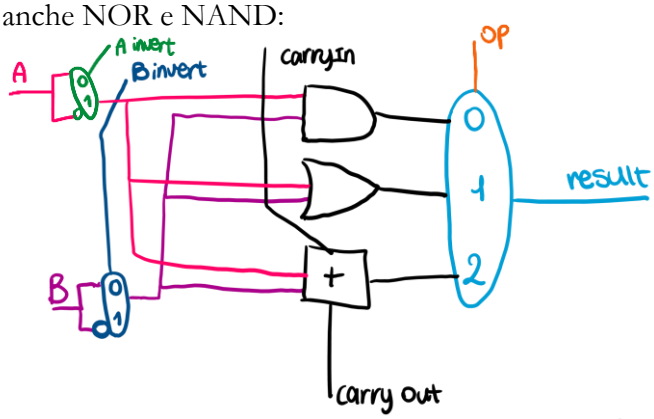
Questa è un'operazione AND e OR



Se voglio fare anche la sottrazione:



Se volessi aggiungere anche NOR e NAND:

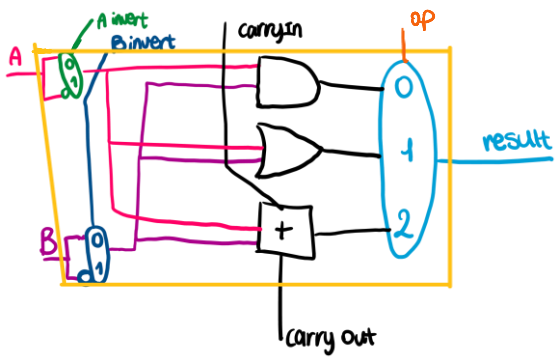


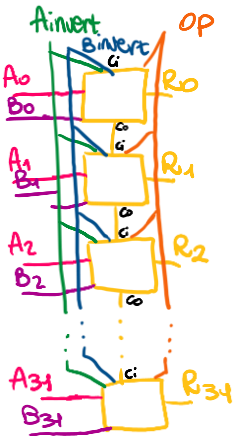
NOR $\rightarrow \neg(A \cup B) = \neg A \cap \neg B$ NAND $\rightarrow \neg(A \cap B) = \neg A \cup \neg B$

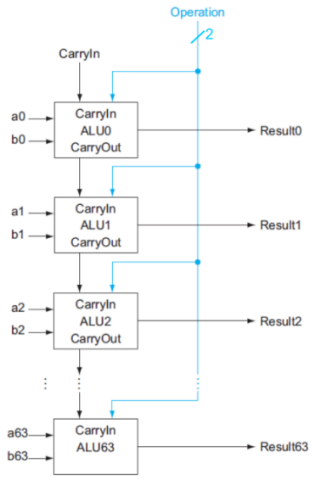
Un'ALU da **1 bit** può fare operazioni molto semplici: ad esempio, può sommare 1 + 1, risultato 0 con riporto 1: il bit meno significativo, quello a destra, è 0, e il riporto (carry) è 1, quindi va trasportato al bit successivo, creando 10, ossia 2 in binario.

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

Supporto per la slt

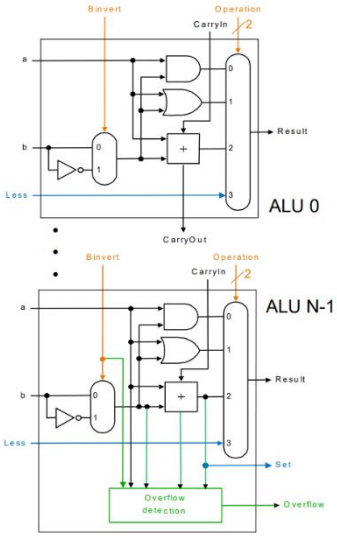
Funzione set-on-less-than (slt) funziona che: se a < b produce 1, altrimenti 0. Per farlo, basta usare la sottrazione: (a – b) < 0 implica a < b. Per vedere se (a-b) è negativo basta che verificare se il bit più significativo vale 1.

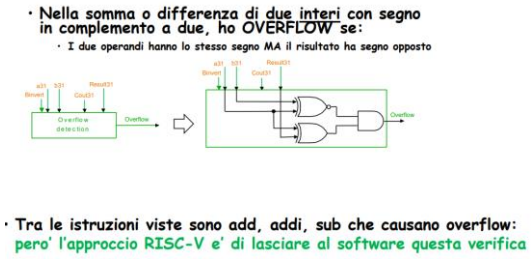
Supporto per il test di uguaglianza

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:





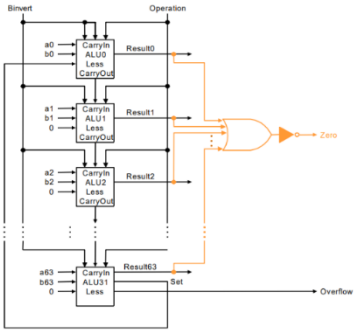
Se il confronto ha esito positivo il filo less invia sul bit meno significativo il risultato 1, numero negativo.

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.



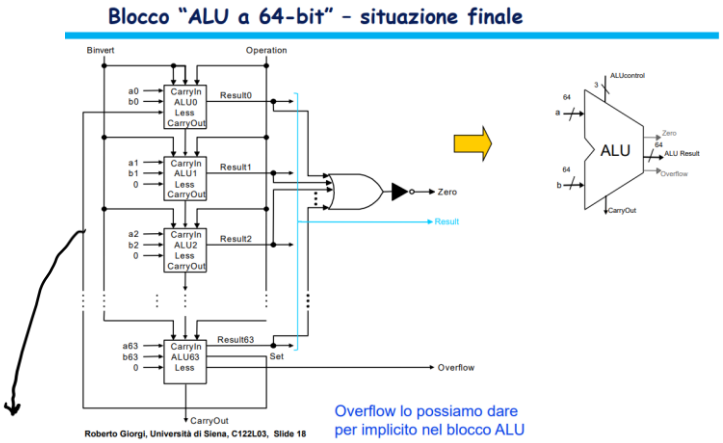
ALUcontrol

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 a destra.

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

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

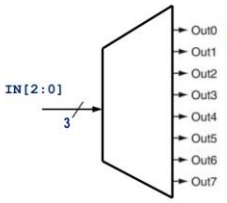
ALUcontrol	Function
0 00	AND
0 01	OR
0 10	ADD
1 10	SUB
1 11	SLT
1 10	EQ



L'immagine è una ALU da 64 bit, ed è adatta ad elaborare grandi quantità di dati, come numeri con virgola mobile, indirizzi di memoria più grandi ecc. Il compito generale di un'ALU è quello di eseguire operazioni, aritmetiche come somma e sottrazione, o logiche come and e or, o anche comparazioni per i salti (branch). Una ALU da 1 bit è formata da due operandi (a, b), collegati in un AND, OR e in un ADDER. Hanno un multiplexer collegato alle 3 operazioni e in base a quale operazione viene scelta (tramite la linea Operation), il risultato sarà quello dell'operazione scelta. L'adder ha un CarryIn e un CarryOut (ossia i riporti dell'addizione), e gli operandi hanno anche una versione invert per fare le sottrazioni, i NOR e i NAND. Una ALU da 64 bit è formata da 64 ALU da 1 bit, con il carryOut collegato come carryIn per l'ALU successiva; per il primo ALU, il carryIn è uguale alla linea Binvert. Nel caso dell'immagine però, è un'uguaglianza. Gli ALU da un bit eseguiranno tutti una sottrazione tra gli operandi. Tutti i risultati saranno poi portati in un controllo che controlla se tutti i risultati della sottrazione sono 0. Se è così vuol dire che lo 'Zero' varrà 1, e vorrà dire che tutti e due gli operandi di ogni ALU sono uguali. I risultati di tutti i 64 bit vengono raccolti insieme nel vettore Result[63:0] che rappresenta il risultato finale a 64 bit dell'operazione eseguita: questo vuol dire che per avere il risultato finale, si deve aspettare che tutti gli ALU da 0 a 63 abbiamo ricevuto il loro carry per eseguire la loro operazione.

Elementi Architetturali Principali: Reti Combinatorie Notevoli

Decoder ed encoder



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

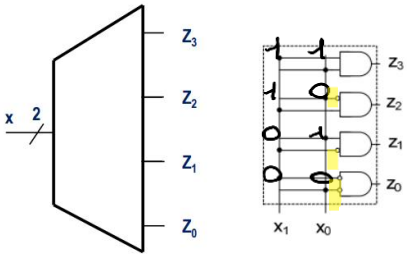
Questo esempio è un **decoder** a 3 bit, ossia da 3 a 8.

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).

Prende in ingresso un codice binario e attiva una sola uscita corrispondente a quel codice. Serve per selezionare una tra tante linee.

In generale, un decoder ha n bit in ingresso e 2^n uscite: ha n input e 2^n output. Quindi, un decoder da 2 ingressi ha 4 output.

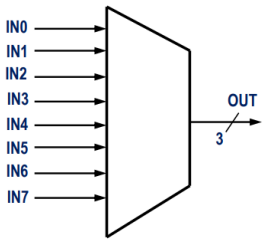
Si può sintetizzare in termini di porte logiche con 2^n porte AND, ognuna delle quali riconosce uno dei 2^n mintermini.



Es. decoder da-2-a-4

L'encoder effettua l'operazione inversa del decoder.

Un encoder prende come ingresso n linee, ognuna la quale rappresenta un segnale attivo, e restituisce in uscita una rappresentazione binaria dell'indice della linea attiva. Serve per ridurre il numero di segnali necessari a rappresentare una selezione. Ha 2^n ingressi e n output. Ad esempio 8 input e 3 output.



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

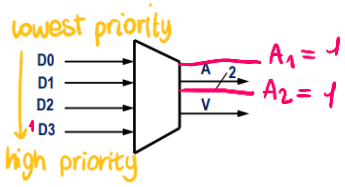
OUT2=IN4+IN5+IN6+IN7

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

Priority encoder

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



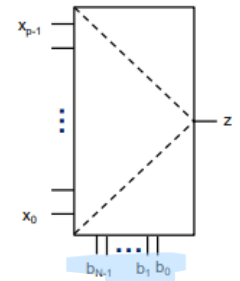
Un priority encoder è una variante dell'encoder che risolve un problema tipico, ossia quando più ingressi sono attivi contemporaneamente (portando ad un output ambiguo).

Il priority encoder funziona come un encoder normale, ma restituisce il codice binario dell'ingresso attivo con priorità più alta. 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.

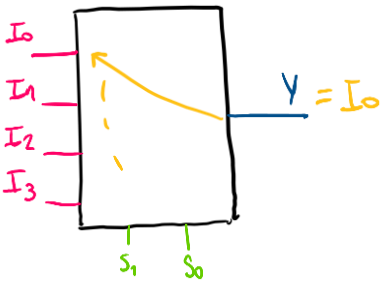
Quando D3 vale 0 ma D2 vale 1 gli altri bit a destra non contano e l'uscita indica priorità 2 e così via…
Oltre a 2^n ingressi e n uscite binarie, ha anche un bit V di validità, che è necessario per indicare se l'uscita è valida, ovvero se almeno un 1 è stato posto in ingresso (= se almeno un ingresso è attivo).

Multiplexer

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 ingressi, N variabili di comando (nella foto sono b) e 1 uscita. Specificando le variabili di comando, viene selezionato uno dei 2^N ingressi.

$n = 2^m$ → 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 perché base = argomento

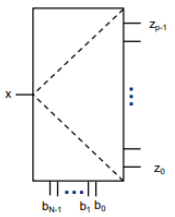
Quindi:

- 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**.

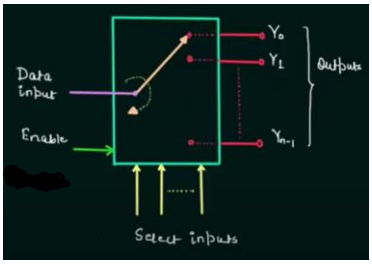
Ad esempio, un MUX 4:1 ha 4 ingressi (I0, I1, I2, I3), 2 bit di selezione (S0, S1) e 1 uscita (Y).
Se S1S0 = 10, il MUX collega l'ingresso I2 all'uscita Y.

Demultiplexer

Realizza la funzione inversa del multiplexer, ovvero data un'informazione x, la distribuisce su una delle 2^N uscite possibili.



Demultiplexer con 2^N uscite



Ha 1 ingresso, N variabili di comando e 2^N uscite. Specificando le variabili di comando, viene selezionato una dei 2^N uscite.

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.

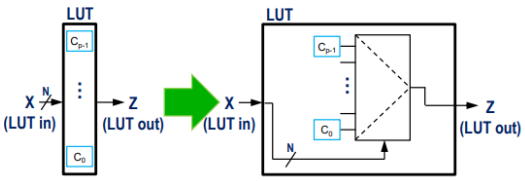
Ad esempio, un DEMUX 1:4 ha: 1 ingresso (D), 2 bit di selezione (S0, S1) e 4 uscite (Y0, Y1, Y2, Y3)
Se S1S0 = 01, il dato in ingresso D va in uscita Y1.

RIASSUNTO FINALE MULTIPLEXER E DEMULTIPLEXER

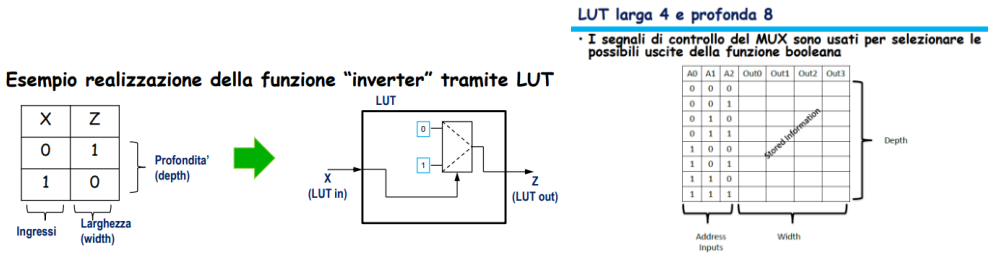
Un multiplexer è un circuito combinatorio che serve a selezionare uno tra diversi ingressi e a inoltrarlo in uscita. Seleziona dati presi da vari input e li trasmette in un'unica uscita. È un data selector perché seleziona un dato tramite select line/value e lo rende output, che è sempre 1. Ha N ingressi, un'uscita e $log2\ (N)$ bit di selezione (select lines), che determinano quale ingresso viene collegato all'uscita. È come se fosse un interruttore controllato digitalmente. Facendo un esempio, un MUX da 4:1 ha 4 ingressi, 2 bit di selezione e 1 uscita. I 2 bit di selezione creano il numero binario che corrisponde all'ingresso: con 00 verrà selezionato l'ingresso I0, con 01 l'ingresso I1, con 10 l'ingresso I2, e con 11 l'ingresso I3.
Un demultiplexer fa la funzione inversa: prende un unico ingresso e lo inoltra su una tra più uscite, in base ai bit di selezione. Si usa quando si vuole distribuire un segnale verso più destinazioni. Ha 1 ingresso, N uscite e $log2\ (N)$ bit di selezione, che determinano quale delle uscite sarà attiva. Facendo un esempio, un DEMUX da 1:4 ha 1 ingresso, 2 bit di selezione e 4 uscite. I 2 bit di selezione creano il numero binario che corrisponde all'output: con 00 verrà selezionato l'uscita I0, con 01 l'uscita I1, con 10 l'uscita I2, e con 11 l'uscita I3.

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.



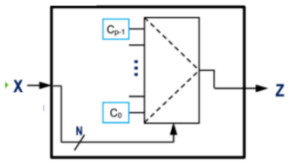
Una LUT è una struttura che memorizza i valori di uscita di una funzione logica per ogni possibile combinazione di ingressi. In pratica:

- 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.

---

4) Spiegar quali componenti include e a cosa potrebbe servire il seguente circuito:



4 punti

Il circuito rappresenta una Look Up Table (LUT), che utilizza un multiplexer per implementare funzioni logiche predefinite, imitando una tabella di verità. Memorizza i valori di uscita di una funzione logica per ogni possibile combinazione di ingressi. Gli ingressi del MUX contengono i risultati predefiniti della funzione logica, e i bit del MUX rappresentano le variabili logiche. Quindi, in base al valore binario delle variabili logiche, viene selezionato uno degli ingressi, che rappresenta l'output corrispondente nella tabella di verità.

Circuiti (o reti) sequenziali

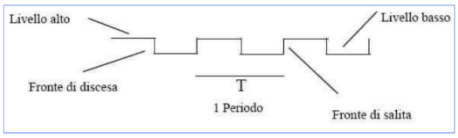
circuiti combinatori: sono un tipo di circuito elettronico digitale in cui l'uscita dipende esclusivamente dagli ingressi in un determinato istante di tempo, senza alcuna memoria interna o dipendenza dallo stato precedente del sistema. L'uscita cambia immediatamente quando gli ingressi cambiano e per 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)



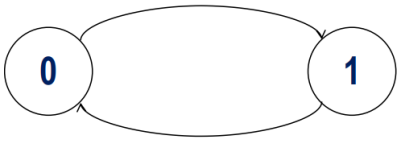
Il clock del processore funge da metronomo per questi cicli di esecuzione. Genera un segnale ritmico, un "ticchettio" regolare, che scandisce il tempo di ogni fase. Più alto è il clock, maggiore è la frequenza con cui il processore può completare un ciclo di esecuzione e, di conseguenza, più istruzioni 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

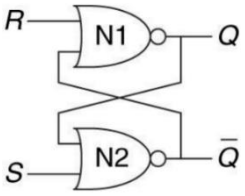
Il primo passo per la progettazione è comprendere il compito che il circuito deve svolgere. Per questo scopo si possono usare dei metodi di specifica dei requisiti del sistema basati ad esempio sulle macchine a strati finiti (FSM).

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.

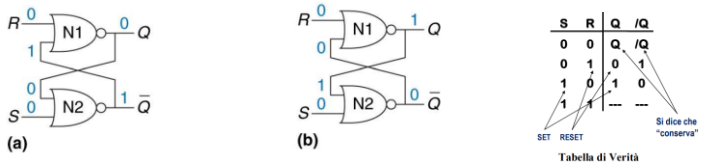


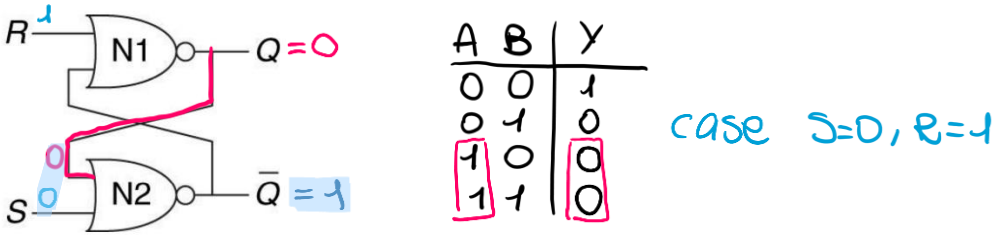
Latch SR



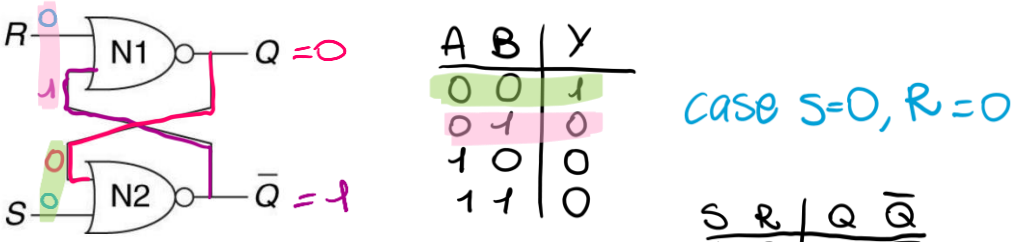
L'esempio di circuito di memoria ad 1 bit più semplice è il Latch SR: utilizza porte NOR e non è guidato da un segnale di «clock» (logica asincrona)
L'uscita dipende dai segnali presenti sugli ingressi e dai segnali precedentemente presentati in ingresso. (asincrono)

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





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

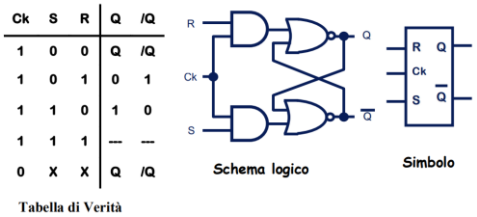


L'output, lo stored data, rimane sempre lo stesso, rimane in memory.
Non si può usare S = 1 e R = 1 perché Q e $\bar{Q}$ sarebbero entrambi 0.

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

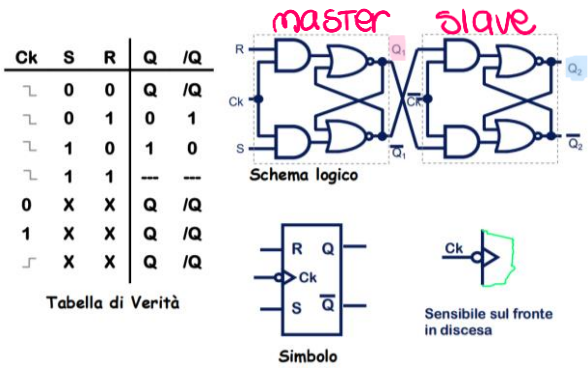
S-R clocked (sincrono)

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.



S-R edge triggered Flip-Flop

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



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.

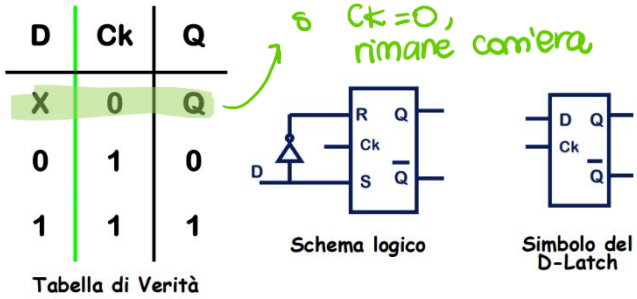
D-Latch (Delay/Data Latch) (serratura/chiavistello) o D-Trasparent

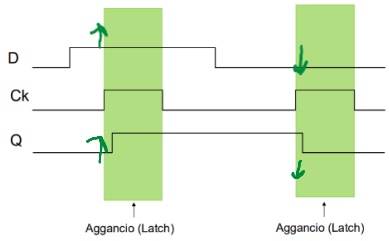
IL latch SR è problematico perché si comporta in modo impredicibile quando S R = 1 1

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)

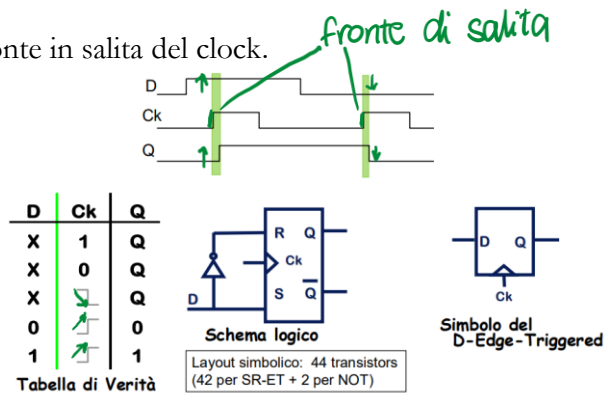




il grafico mostra perché viene anche chiamato transparent:
Le aree verdi indicano quando Ck = 1.
In quelle zone, Q **segue esattamente** D (dato da memorizzare)
Quando Ck = 0, anche se D cambia, Q rimane bloccato al valore che aveva.

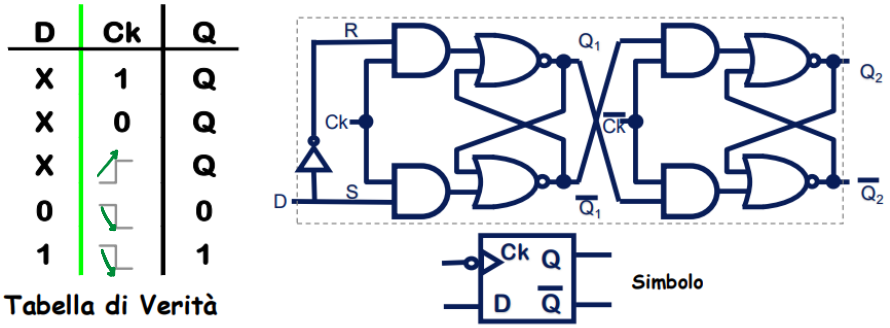
D-edge triggered

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



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).
Un flip-flop D edge triggered è un circuito sequenziale che cattura il valore presente in ingresso D al fronte di salita e mantiene il valore catturato stabile in uscita Q fino al prossimo fronte di clock; quindi, ignorando le altre variazioni di D se non eseguite al fronte di 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.



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.

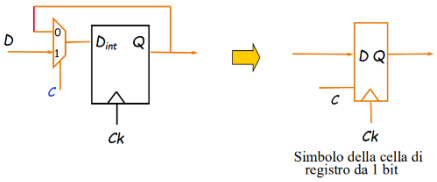
Confronto Latch vs. Flip-flops
Nel latch, l'uscita cambia non appena cambia l'ingresso e il clock ha valore alto.
Nel Flip-flop: l'uscita cambia solo su un fronte del clock (metodologia edge-triggered)
Gli elementi in comune sono:

- 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.

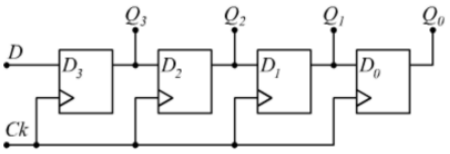
Cella di registro

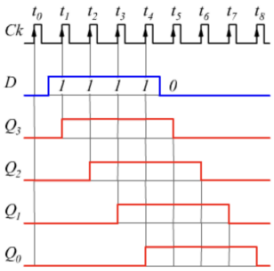
Una cella di memoria da un bit usa un D-Latch e un multiplexer, il cui controllo si chiama LE (Latch Enable), WE (Write Enable) o C (control) (sono tutti la stessa cosa)



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).

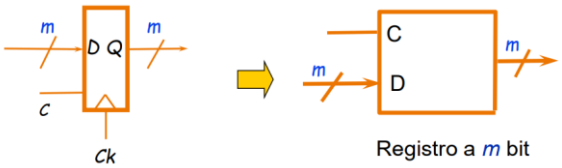
Registro con input in serie e output in parallelo





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.

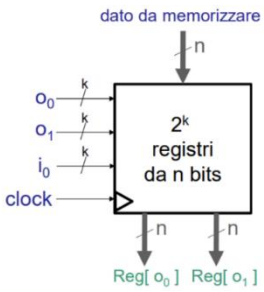
Registro a m bit

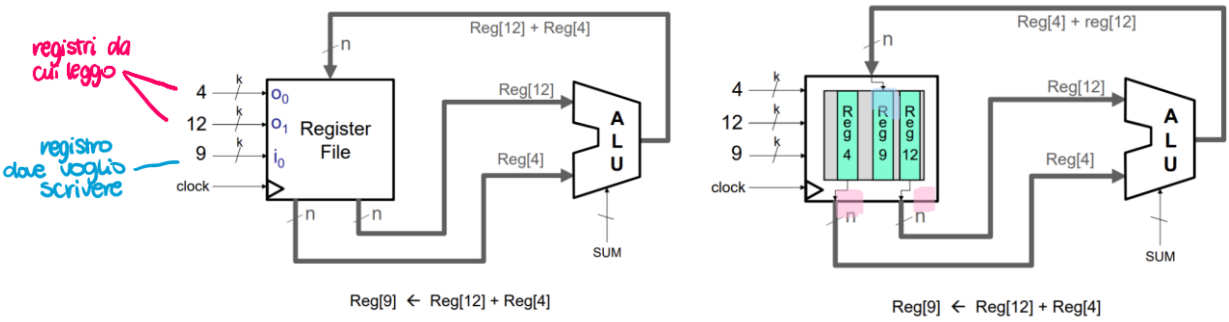


È un registro che memorizza m bit contemporaneamente. Il registro legge il valore su D e lo memorizza in Q solo se C è attivo e al fonte di salita del Ck. (nell'ultima rappresentazione si intende implicitamente che sia presente un segnale Ck per il clock).

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

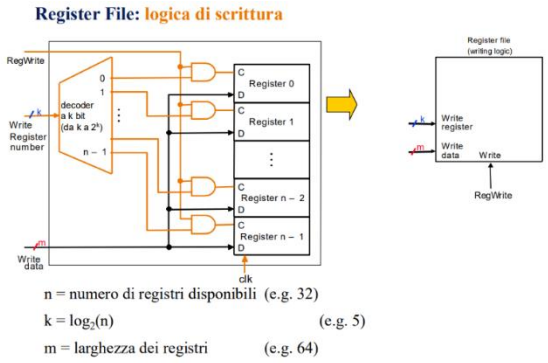
- contiene m = 2^k registri da n bit ciascuno, numerati da 0 a $2^k - 1$
- in output restituisce il valore attuale di due dei registri, specificati dagli indici o_0 e o_1
- in input, prende anche la parola di n bit che verrà memorizzata nel registro di indice i_0 (quando il clock è al fronte di salita)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





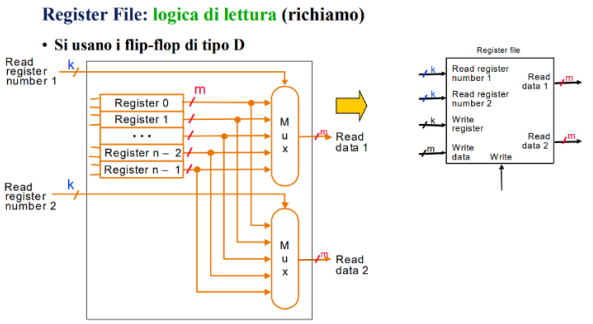
Reg[12] e Reg[4] vengono letti e inviati all'ALU. L'ALU esegue l'op somma (SUM) e il risultato della somma viene poi memorizzato in Reg[9] SOLO AL PROSSIMO FRONTE DEL CLOCK.

Logica di scrittura



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.

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:

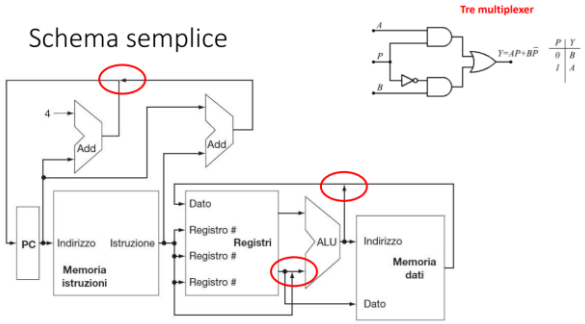
- **Instruction fetch:** prelevo la prossima istruzione da eseguire ed incremento il PC (Program Counter)
- **Instruction decode:** decodifico l'istruzione e leggo gli eventuali operandi dal register file
- **Execution:** uso la ALU per il calcolo. Un valore di un'operazione Aritmetico Logica o un indirizzo 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

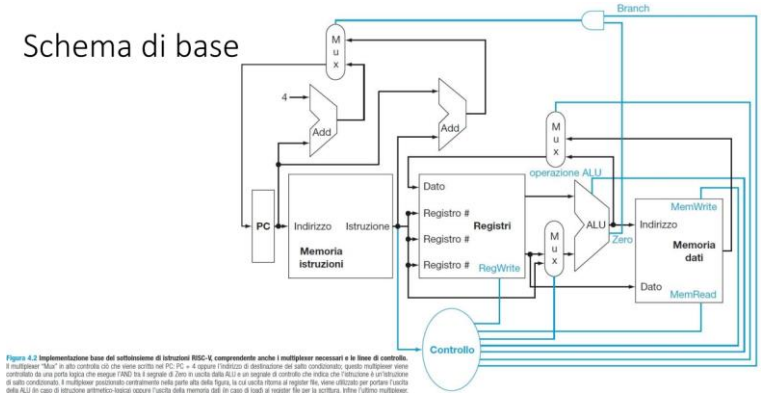
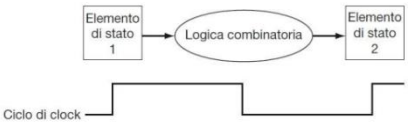


Figura 4.2 Implementazione base del sottoinsieme di istruzioni RISC-V, comprendente anche i multiplexer necessari e le linee di controllo. Il multiplexer "Mux" in alto controlla ciò che viene scritto nel PC: PC + 4 oppure l'indirizzo di destinazione del salto condizionato; questo multiplexer viene controllato 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-logicaoppure l'uscita della memoria dati in caso di loadi al register file per la scrittura. Infine l'ultimo multiplexer.

Temporizzazione

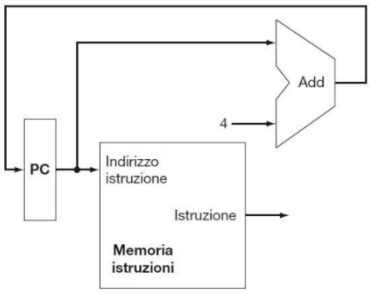
I valori da inserire in un elemento di stato provengono tipicamente da una rete combinatoria come elaborazione di altri dati. Bisogna quindi aspettare che il dato prodotto dalla rete combinatorio sia valido e che il suo valore non stia variando. Si usa quindi una temporizzazione edge-triggered, dove 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.

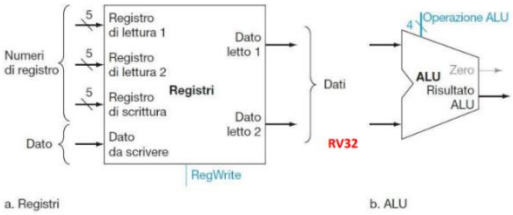
Il **program counter (PC)** è un registro da 32 bit (RV32) che conserva l'indirizzo di memoria della prossima istruzione da eseguire. Viene aggiornato al termine di ogni ciclo clock, quindi non richiede un segnale di controllo esplicito per la scrittura.



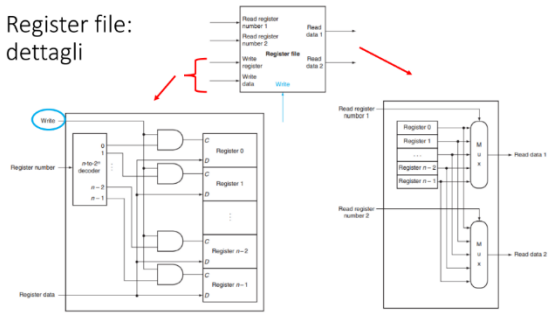
La **memoria delle istruzioni** fornisce solamente un accesso in lettura, perché essa non viene scritta dall'unità di elaborazione. Per questo, viene trattata come un circuito logico combinatorio

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.

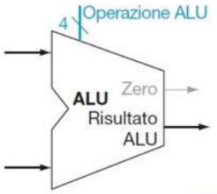


Il **register file** prende in input due registri di lettura, un registro di scrittura, un dato da scrivere e un RegWrite. I due registri di lettura non vengono modificati, vengono soltanto letti e mandati direttamente in output. Il registro di scrittura viene modificato in base al dato da scrivere soltanto 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.

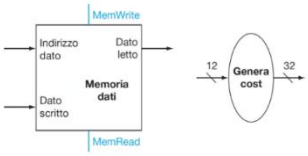


La **ALU** prende in input i valori dei registri di lettura del register file e ci fa delle operazioni. In base all'ingresso di controllo alla ALU, le operazioni saranno somma (0010), sottrazione (0110), and (0000) e or (0001).

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.

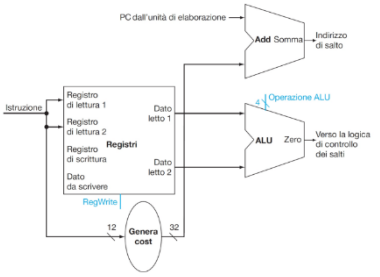


- **Indirizzo dato**: indirizzo della locazione di memoria a cui il processore desidera accedere. L'indirizzo di memoria viene solitamente fornito da un registro di processore (PC).
- **Dato scritto**: se l'operazione è una scrittura della memoria (store), l'unità riceve i dati che il processore desidera 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.

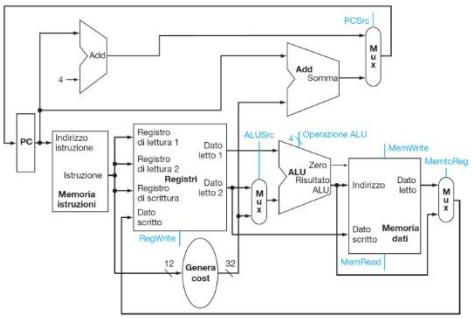
L'**unità di estensione del segno** (unità di generazione della costante) prende un numero con meno bit (12) e lo trasforma in un numero con più bit, conservandone il segno. Quindi serve per supportare i numeri negativi.

BEQ

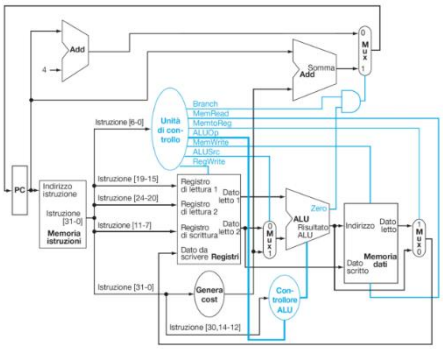
L'istruzione entra nel register file. La ALU confronta i valori del Dato Letto 1 e il Dato Letto 2. Fa la sottrazione rs1 – rs2, e il risultato è 0, allora vuol dire che rs1 == rs2, e il segnale mandato dalla ALU è il segnale 'Zero'. L'immediato dell'istruzione entra nel Genera Cost, e questo offset 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).



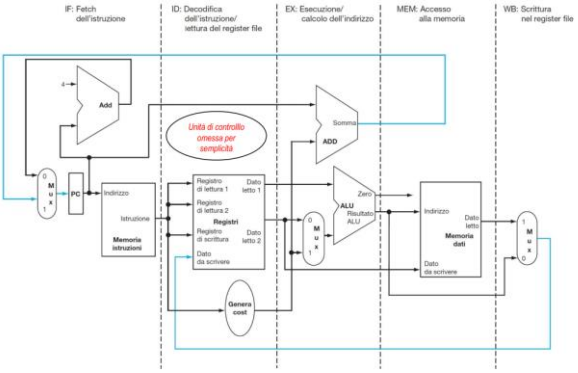
Unità di controllo
L'unità di controllo (control unit) è la logica di comando della CPU: decide quali segnali attivate in ogni momento in base all'istruzione corrente. È responsabile della gestione del flusso di istruzioni all'interno del processore.
Riceve in input il codice operativo (CodeOP) dell'istruzione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



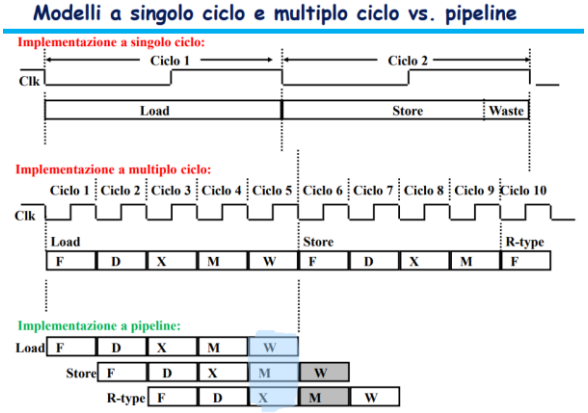
CPU con pipeline
L'architettura vista finora a singolo ciclo funziona benissimo ma è inefficiente. Il periodo di clock è proporzionale al cammino di esecuzione più lungo (LD), quindi l'intero ciclo di clock deve essere abbastanza lungo da coprire anche l'istruzione più lenta. Questo tempo lo usano però 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.
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.



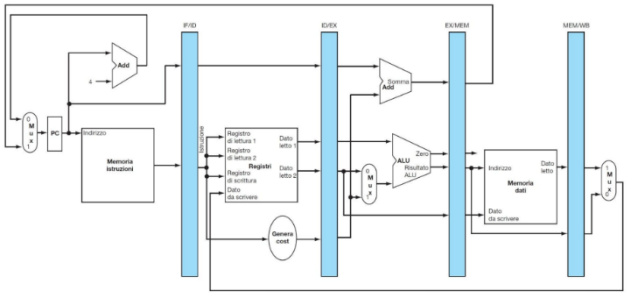
Nell'implementazione a singolo ciclo, ogni istruzione viene completata in un solo ciclo di clock. Tutte le fasi (fetch, decode, execute, mem, writeback) avvengono in quel singolo ciclo, che deve quindi essere molto lungo per contenere anche l'istruzione più lenta. Dopo ogni istruzione c'è uno spreco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

Registri aggiuntivi

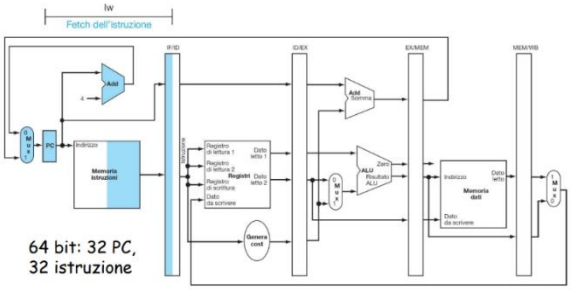
Per preservare il valore dell'istruzione in esecuzione, il PC ed altri valori ci servono dei registri specifici per la pipeline. Per etichettarli si usa il nome dei relativi stadi che separano: ad esempio, il primo registro viene etichettato con IF/ID. I registri devono avere un'ampiezza sufficiente 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

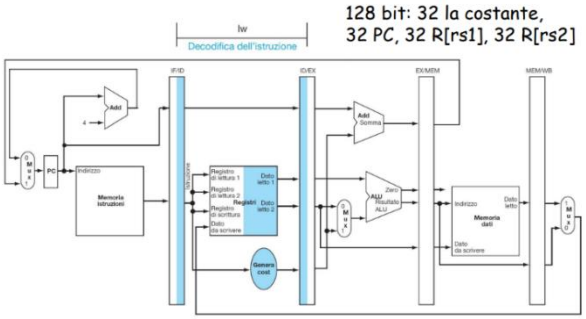
Nel fetch, il PC viene incrementato e riscritto, ma il vecchio valore va salvato in IF/ID perché può essere necessario più avanti, ad esempio nella beq.



Decodifica

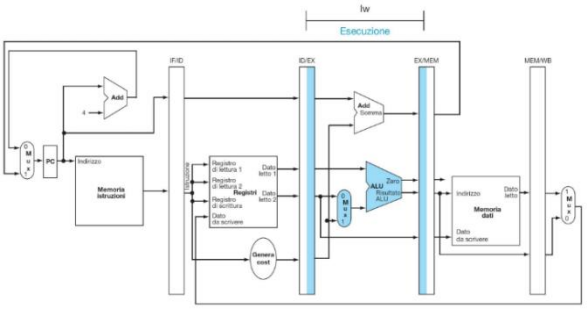
Prende l'istruzione dal fetch (32 bit), e viene smezzata nei suoi campi, quindi rs1, rs2, rd (registro destinazione), imm (costante), opcode, funct3, funct7. Così l'istruzione è decodificata

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'.

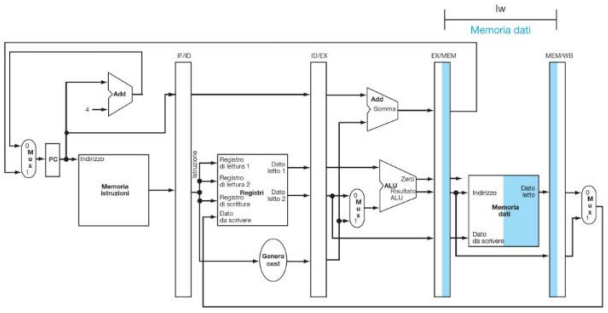


Accesso alla memoria

L'obbiettivo di MEM:

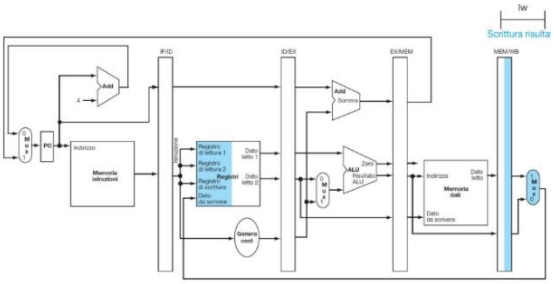
se l'istruzione è lw, legge dalla memoria dati un valore usando l'indirizzo calcolato nello stadio EX. Il valore letto viene poi salvato nel registro MEM/WB, pronto per il Write Back.

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.

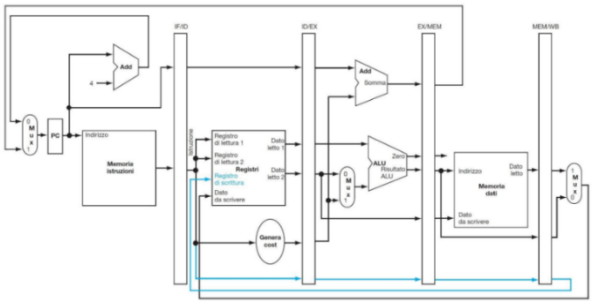


Scrittura nel register file

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.



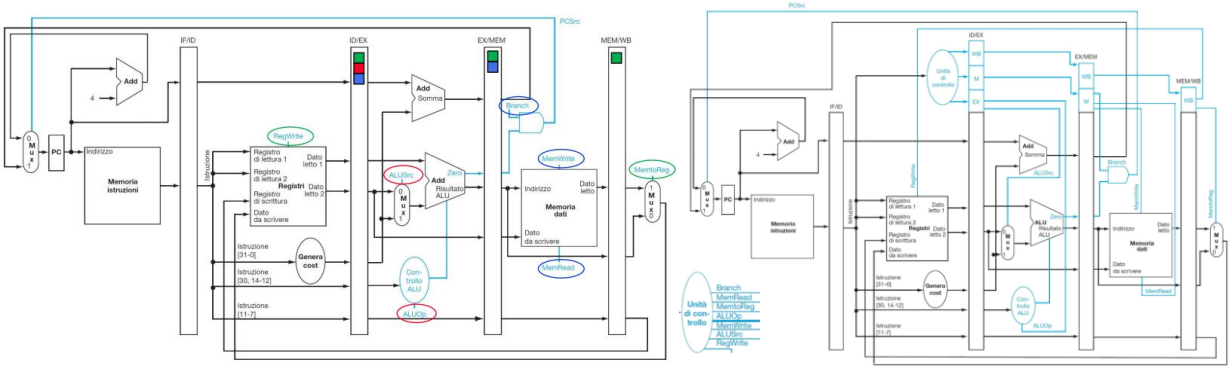
Per risolvere questo problema, l'informazione sul registro di destinazione rd viene ora trasportata lungo tutta la pipeline, fino allo stadio MEM/WB. I 5 bit (registri RISC-V sono 32 e $2^5 = 32$) che identificano rd sono ora inclusi nei registri interstadio (ID/EX, EX/MEM, MEM/WB).

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.

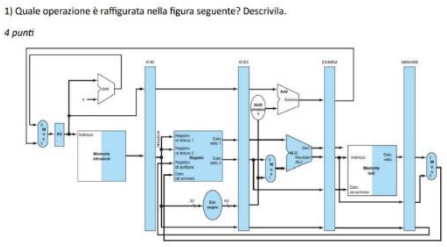
	EX		MEM			WB	
Istruzione	**Segnali di controllo dello stadio di esecuzione/calcolo dell'indirizzo**		**Segnali di controllo dello stadio di accesso alla memoria dati**			**Segnali di controllo dello stadio di scrittura**	
	ALUOp	**ALUSrc**	**Branch**	**MemRead**	**MemWrite**	**RegWrite**	**MemtoReg**
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.

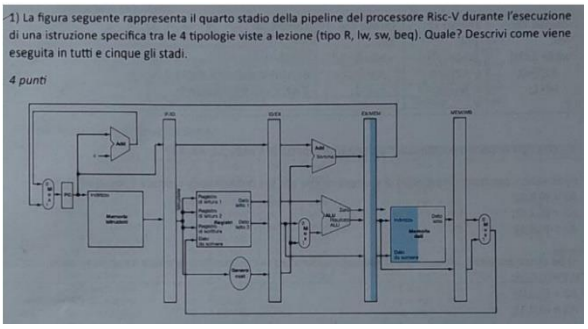
RISPOSTE VARIE ESAME



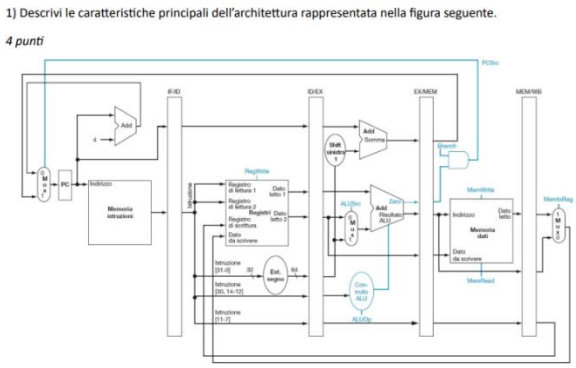
L'architettura rappresentata è un'architettura Pipeline a 5 stadi. Lo scopo della pipeline è quello di aumentare il numero di istruzioni eseguite in un certo tempo, eseguendole in parallelo. L'istruzione rappresentata è una lw (load word).

Parte dal PC (program counter), che prende l'istruzione dalla memoria istruzioni. ID (decodifica) prende l'istruzione dal registro IF/ID e la smezza; inoltre, prende l'offset costante e lo porta nel multiplexer dello stadio successivo. Nello stadio successivo EX, l'ALU calcola un indirizzo sommando un valore del registro (rs1) con quello dell'offset (il MUX gli manda la costante offset al posto del secondo registro di solito usato per le operazioni o il beq), per produrre l'indirizzo usato dalla memoria dati per leggere la parola. Nello stadio

MEM, prende il valore salvato nel registro EX/MEM e legge dalla memoria dati usando l'indirizzo calcolato da EX. Il valore viene salvato nel registro MEM/WB, Write Back lo prende e lo riporta nel registro di destinazione rd. Questo registro di destinazione viene trasportato fino a WB lungo tutta la pipeline nei registri interstadio (ID/EX, EX/MEM, MEM/WB), che avranno quindi 5 bit in più. Se così non fosse, WB non saprebbe il registro dove scrivere la parola.



In questa figura viene rappresentata l'istruzione sw (store word). Partendo da IF, l'instruction fetch, il PC punta all'istruzione dalla memoria delle istruzioni (e viene poi incrementato di 4 per l'istruzione successiva). In ID, instruction decode, si decodifica l'istruzione, quindi la base dell'indirizzo, il dato da salvare in memoria e l'offset, che viene estratto con il Generatore di Costante. Nell'EX, l'ALU calcola l'indirizzo dove deve essere salvato il dato, prendendo la base dell'indirizzo e sommandola con l'offset. MEM prende i dati e l'indirizzo ritornato dalla ALU dal registro EX/MEM (i registri sono tra ogni stadio e sono la memoria che fa in modo che un'istruzione possa avere i dati e risultati ricavati dallo stadio precedente) e scrive il dato nell'indirizzo. Il quinto stadio WB non fa nulla perché la funzione sw si ferma in memoria, facendo lo store del dato in memoria e non dovendo riportare nulla nei registri.



L'architettura rappresentata è un'architettura Pipeline a 5 stadi. Lo scopo della pipeline è quello di aumentare il troughput del processore, ossia il numero di istruzioni eseguite in un certo tempo, eseguendole in parallelo. Gli stadi sono IF (instruction fetch), ID (instruction decode), EX (execute), MEM (memory access) e WB (write back). L'IF legge l'istruzione dalla memoria istruzioni usando il program counter (PC); l'ID decodifica l'istruzione; EX esegue le operazioni tramite ALU, e calcola l'indirizzo in casi di accesso di branch, facendo PC + costante (offset decodificato da ID); MEM accede alla memoria dati in caso di load o store; WB scrive il risultato nel registro di destinazione. Nella foto ci sono 4 registri IF/ID (64 bit), ID/EX (128 bit), EX/MEM (97 bit) e MEM/WB (64 bit). I registri servono per mantenere i dati tra uno stadio e l'altro, permettendo così di eseguire più istruzioni contemporaneamente, ognuna in uno stadio diverso della pipeline.

Ci sono poi dei controlli (che di solito sono generati dalla control unit, che però non è nell'immagine), che nell'immagine sono ALUOp, ALUSrc, MemRead, MemWrite, Branch, … Controllo ALU, sotto all'ALU dell'EX, è il modulo che manda un segnale ad ALU per fargli capire quale operazione deve eseguire, basandosi sull'istruzione decodificata da ID. Molto importante è il branch che si trova nel'MEM. In caso di branch, l'ALU usa il segnale Zero per indicare che gli indirizzi sono uguali, e il nuovo indirizzo che è stato calcolato in EX viene portato allo stadio iniziale (IF). Lì si trova un MUX (multiplexer) che aggiorna il PC normalmente a PC + 4 (quindi al registro successivo) se branch non viene eseguito, mentre se Zero = 1, il PC si aggiorna al nuovo indirizzo calcolato prima. L'informazione del registro di destinazione deve essere trasportata fino a WB, se no Write Back non saprà dove riscrivere il risultato. Rd viene quindi trasportata lungo tutta la pipeline nei registri interstadio (ID/EX, EX/MEM, MEM/WB), che avranno quindi 5 bit in più.

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:

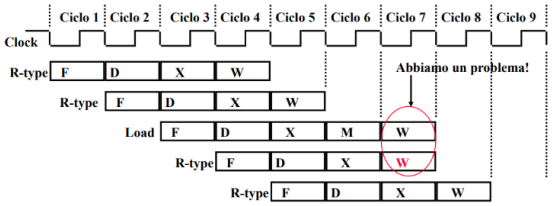
- Rilevare la presenza di criticità
- Gestirla, fermando un'istruzione, rimandandola, o adattando il flusso.

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

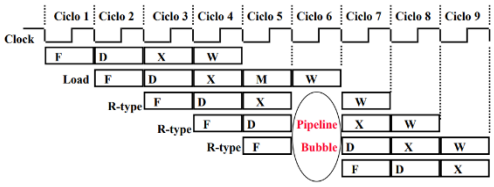
Le criticità strutturali sono situazioni in cui due istruzioni cercano di usare la stessa risorsa hardware nello stesso ciclo di clock, ma questa risorsa non è duplicata.

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).

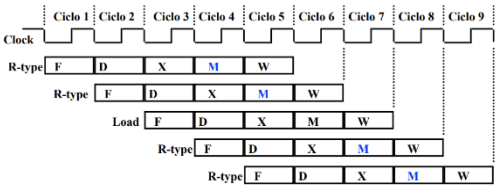


Abbiamo una criticità strutturale perché due istruzioni stanno tentando di scrivere nei registri contemporaneamente, anche se la porta di scrittura è unica.

La <u>prima soluzione</u> 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 <u>seconda soluzione</u> 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:

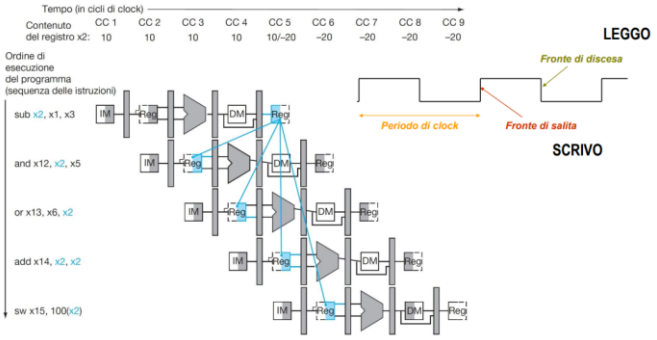
sub <u>x2</u> ,x1,x3

and x12, <u>x2</u> ,x5

or x13 ,x6, <u>x2</u>

add x14, <u>x2</u>, <u>x2</u>

sw x15, 100 (<u>x2</u>)



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:

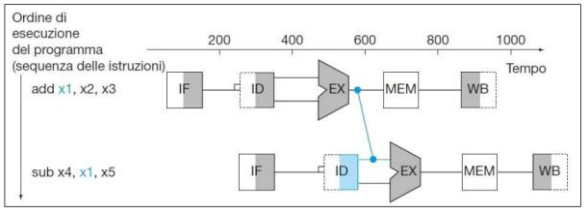
- **RAW (Read After Write) o FLOW dependency:** un'istruzione successiva legge un registro prima che l'istruzione precedente l'abbia scritto. È il caso dell'esempio.
- **WAR (Write After Read) o ANTI dependency:** un'istruzione successiva cerca di scrivere un registro che è stato letto dall'istruzione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

Le soluzioni sono:

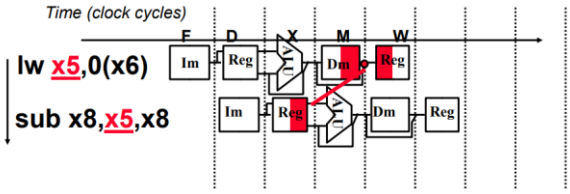
- **Inserimento di bolle**
- **Propagazione (o bypassing):** passo il dato direttamente, senza aspettare WB
- **Riordinamento istruzioni:** sposto il codice per evitare che le istruzioni dipendenti siano troppo vicine

Propagazione

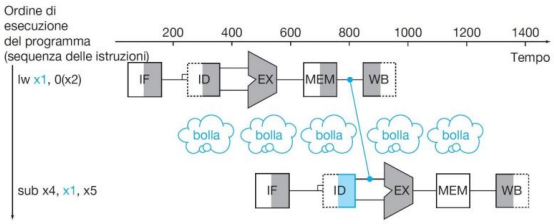
Quando possibile, il dato può essere 'anticipato' e passato dallo stadio EX o MEM direttamente al prossimo EX. In questo modo non si aspetta WB e si evita di introdurre una bolla.



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



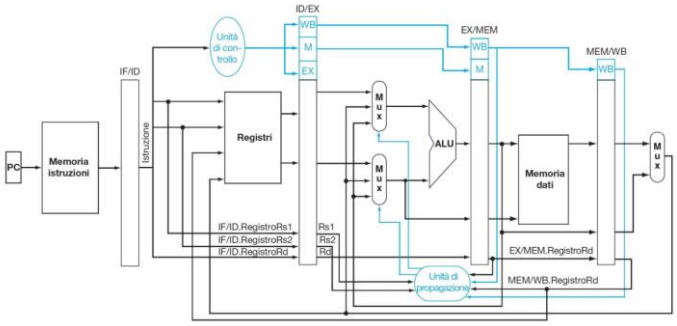
In questo caso, il dato è disponibile solo a fine MEM, quindi le istruzioni che dipendono da una load devono essere ritardate e mandate in stallo via hardware, chiamato **interlock**, che inserisce una bolla automatica per aspettare il dato.



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).

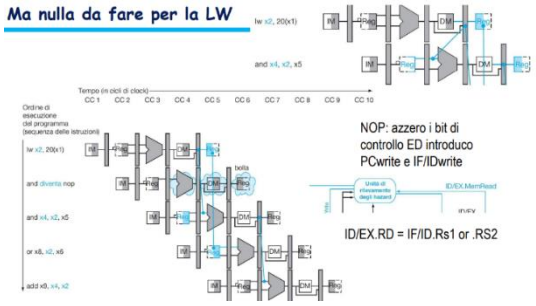
La soluzione



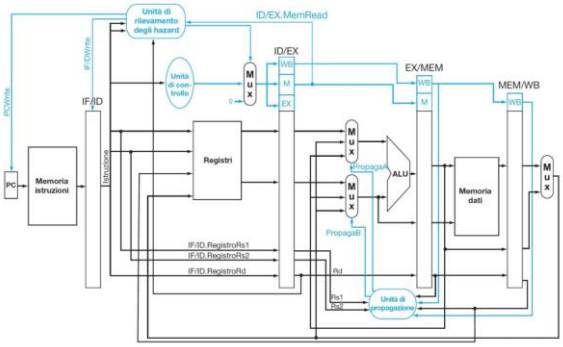
La fowarding unit (unità di propagazione) prende in ingresso i valori rs1, rs2, rd e RegWrite da ID/EX, EX/MEM e MEM/WB, e decide, tramite circuiti logici, se fare forwarding da EX/MEM, da MEM/WB o non fare nulla.

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

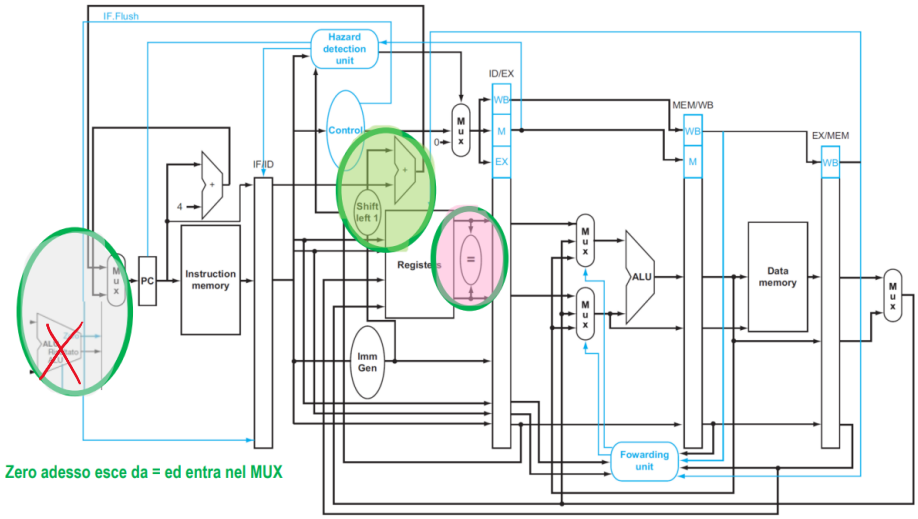
In uno schema base (quello che abbiamo visto finora), la decisione sul salto (branch) viene presa solo nello stadio MEM:

- 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

Eccezioni

Le eccezioni sono tipicamente eventi non previsti che alterano il normale flusso delle istruzioni: I/O, malfunzionamento hardware, overflow aritmetico.

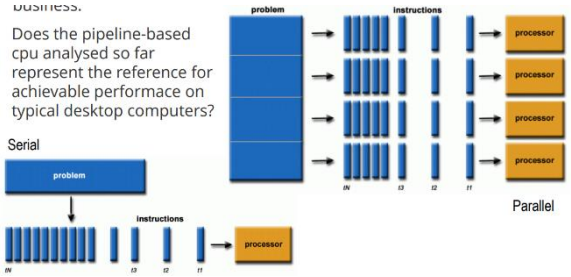
Vengono distinte dagli interrupt in quanto sono eventi esterni, mentre questi sono interni.

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:

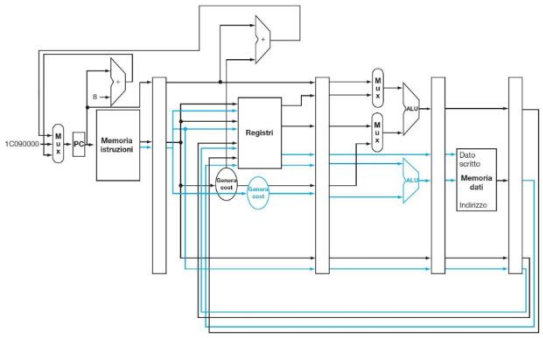
- Statica: le decisioni sono prese durante la compilazione (compilatore)
- Dinamica: le decisioni sono prese durante l'esecuzione (hardware)
- Ibrida: una combinazione delle due, la più usata

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.
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

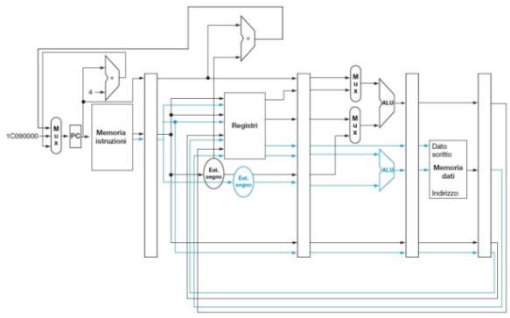
- Ogni istruzione eseguita (solo quando gli operandi sono disponibili) va nella stazione corrispondente, e solo quando gli operandi sono disponibili (e.g. load, propagation) l'istruzione può essere eseguita
- Le istruzioni possono essere eseguite fuori ordine MA saranno consegnate sempre in ordine
- 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.

RISPOSTE ESAME

1) Quale tipo di parallelismo a livello di istruzioni è raffigurata nella figura seguente? Descrivilo brevemente.

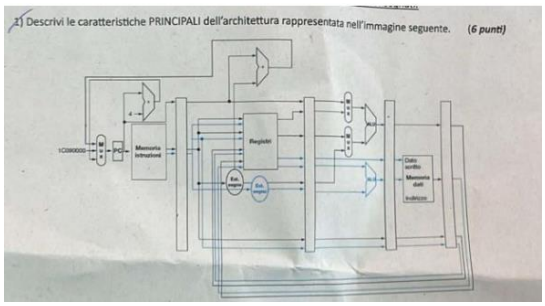
4 punti



Il tipo di parallelismo rappresentato è il parallelismo a livello di istruzioni (Instruction Level Parallelism, ILP), che è una forma di esecuzione simultanea di più di un'istruzione per ciclo di clock. La pipeline è la prima forma di ILP, dove ogni stadio lavora in un'istruzione diversa, e in cui maggiore il parallelismo, minore è il tempo di esecuzione.

Le decisioni sono prese durante la compilazione, direttamente nell'hardware, dove il compilatore si occupa di fornire i pacchetti di istruzioni e prevenire o ridurre gli hazard, anche con la predizione. La predizione prevede la speculazione sulle istruzioni successive, 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 eseguite sotto predizione e rieseguirle, svuotando la pipeline e tornando indietro. Nella speculazione dinamica quindi i dati vengono memorizzati nel buffer e copiati nel register file solo se la speculazione è corretta.

Le parti blu dello schema indicano i percorsi di forwarding (bypass), una tecnica usata per risolvere le dipendenze sui dati tra istruzioni consecutive, senza inserire stall. Questi percorsi prelevano i risultati non ancora scritti nei registri e li inoltrano direttamente alla ALU, rendendo la pipeline più efficiente.



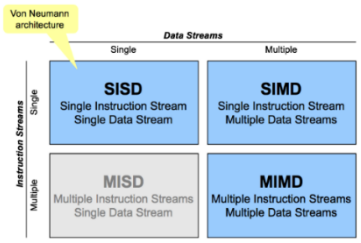
L'architettura rappresentata è un'architettura Pipeline a 5 stadi. Lo scopo della pipeline è quello di aumentare il numero di istruzioni eseguite in un certo tempo, eseguendole in parallelo. Gli stadi sono IF (instruction fetch), ID (instruction decode), EX (execute), MEM (memory access) e WB (write back). L'IF legge l'istruzione dalla memoria istruzioni usando il program counter (PC); l'ID decodifica l'istruzione; EX esegue le operazioni tramite ALU, e calcola l'indirizzo in casi di accesso di branch, facendo PC + costante (offset decodificato da ID); MEM accede alla memoria dati in caso di load o store; WB scrive il risultato nel registro di destinazione. Nella foto ci sono 4 registri IF/ID (64 bit), ID/EX (128 bit), EX/MEM (97 bit) e MEM/WB (64 bit). I registri servono per mantenere i dati tra uno stadio e l'altro, permettendo così di eseguire più istruzioni contemporaneamente, ognuna in uno stadio diverso della pipeline.
Le parti blu dello schema indicano i percorsi di forwarding (bypass), una tecnica usata per risolvere le dipendenze sui dati tra istruzioni consecutive, senza inserire stall. Questi percorsi prelevano i risultati non ancora scritti nei registri e li inoltrano direttamente alla ALU, rendendo la pipeline più efficiente.

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.
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.

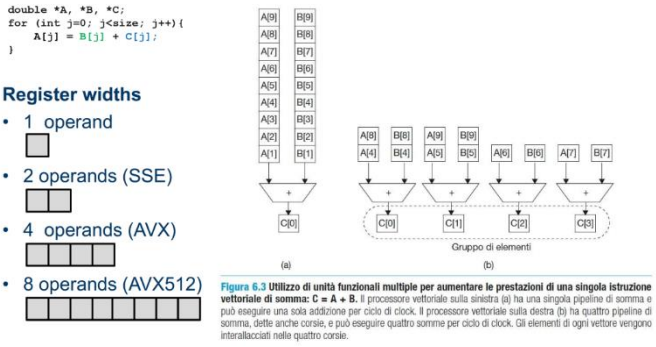


Figura 6.3 Utilizzo di unità funzionali multiple per aumentare le prestazioni di una singola istruzione vettoriale di somma: C = A + B. Il processore vettoriale sulla sinistra (a) ha una singola pipeline di somma e può eseguire una sola addizione per ciclo di clock. Il processore vettoriale sulla destra (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.

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).

---

4) Un processore che supporta AVX2 è un processore x86? E' teoricamente più performante di uno che supporta SSE4.2? Motiva in modo sintetico ma esaustivo. **(3 punti)**

Si, un processore che supporta AVX2 è un processore x86. È più performante di un SSE4.2 perché AVX2 introduce i registri a 256 bit, il doppio rispetto ai 128 bit di SSE4.2, facendo in modo di eseguire più operazioni SIMD in parallelo.

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.

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)

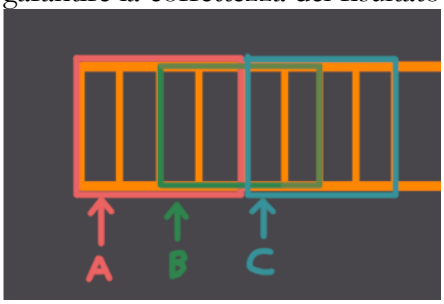
- LID: se uno statement in un'iterazione di un loop dipende solo da uno statement nella stessa iterazione del loop E due accessi di memoria sono dipendenti tra loro se riferiscono allo stesso luogo di memoria (memory location) e una delle referenze è un write (ogni iterazione è indipendente)

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.

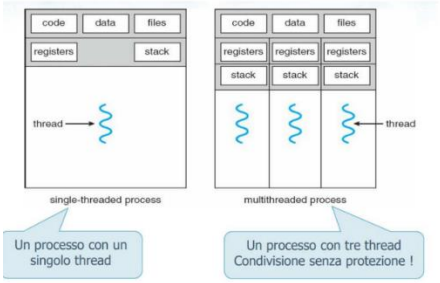
Parallelismo nel multithreading

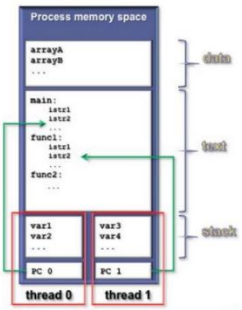
Come abbiamo detto prima, la categoria del multithread è la MIMD.

<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.

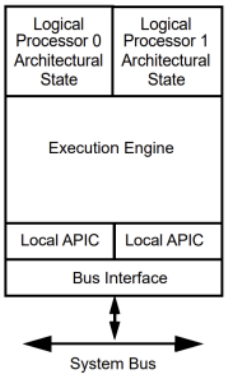




Un processo ha uno spazio di memoria separato, mentre i thread condividono la memoria e le risorse con gli altri thread dello stesso processo.

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:

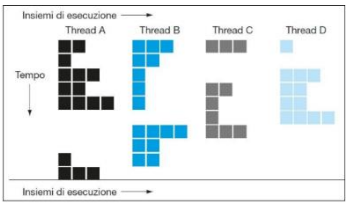
- Sostituzione del valore del PC (Program Counter)
- Sostituzione dei valori dei registri nel register file
- Ogni thread ha il suo spazio dedicato nello stack
- Può essere leggero (SMT – HT cambia la copia del registro) o oneroso (se SW MT)
  (SMT ha più copie dei registri, quindi non deve salvare nulla 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)

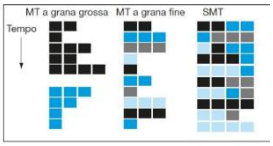
Multithreading a grana fine:

- 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

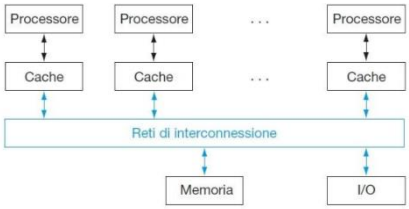
novantasei core. Un processore multi-core fornisce (virtualmente) tutti i vantaggi di avere più processori fisici separati, con costi generali decisamente inferiori. AMD e Intel hanno introdotto i primi processori dual-core x86 per PC desktop nel maggio 2005. La prima soluzione di AMD è stata l'Athlon 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.

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:

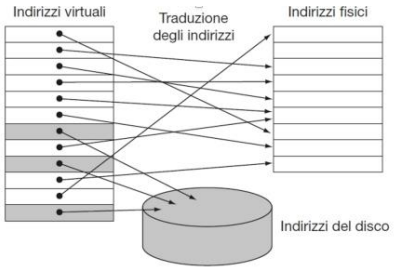
- Se il riferimento (indirizzo memoria) è già presente nella cache (cache hit), si procede con la lettura della linea di cache
- Altrimenti occorre trasferire i dati dal livello inferiore (cache miss) ed inserire una nuova linea di cache, eventualmente sovrascrivendo altre linee di cache (riferimenti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

<u>Intel Hyper-Threading</u>: 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.

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

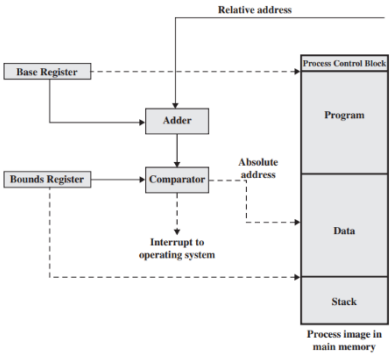
Binding

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

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

Linking dinamico: 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

- Lavoro foreground/background: mentre un thread gestisce l'I/O con l'utente, altri thread operano sui dati in background
- Elaborazione asincrona: operazioni asincrone possono essere implementate come thread. Es: salvataggio automatico
- Task intrinsecamente paralleli: 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.

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:

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).

```
Risorse
var x=0

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

Esempio 2:

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 ≥ 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.

```
var x=0

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

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

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

Esempio 3:

- 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

```
Risorse
var x=100

Thread P1 {
  x:=x+1;
}

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

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.

Risposta esame: Le variabili sono condivise tra i *n* thread, ma non c'è nessuna sincronizzazione, rendendo possibile l'accesso simultaneo e quindi race condition. L'ordine di esecuzione è non deterministico, può cambiare ogni volta che il programma viene eseguito, portando quindi a risultati diversi.

Memoria virtuale, Cache e Multicore

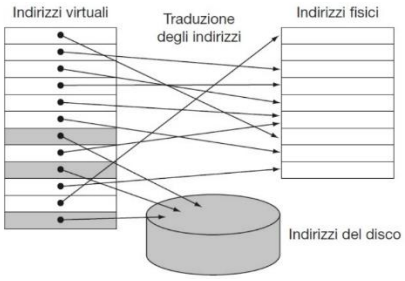
Memoria virtuale

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.

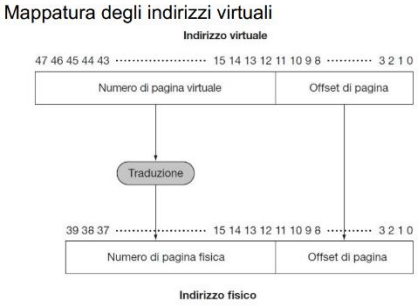
Per implementare una memoria virtuale:

- 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



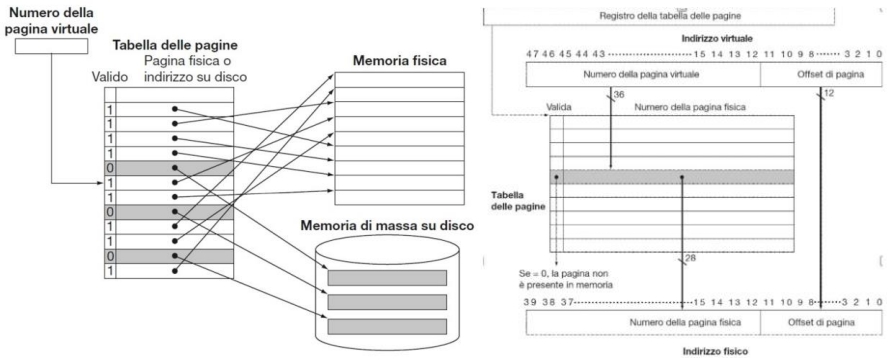
Nella memoria virtuale i blocchi della memoria (pagine) vengono mappati da un insieme di indirizzi (detti virtuali) a un altro insieme di indirizzi (detti fisici). Il processore produce indirizzi virtuali mentre l'accesso alla memoria avviene utilizzando gli indirizzi fisici. Sia la memoria virtuale 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.



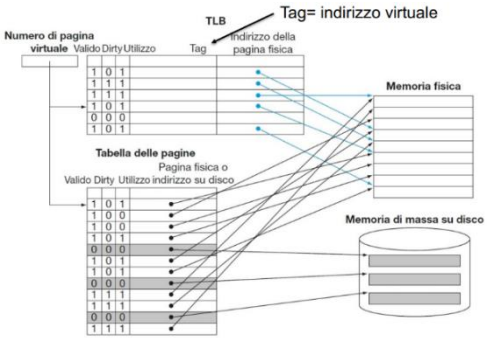
Indirizzi fisici da 40 bit → indirizzo di pagina 12 bit

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.

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.

7) Confrontare le strategie **write through** e **write back** usate nella gestione della cache. **(4 punti)**

La strategia write through, quando scrivo qualcosa di nuovo sulla cache, aggiorna subito anche la memoria principale. La write back, invece, scrive in cache, ma rimanda l'aggiornamento della memoria al momento in cui il blocco della cache deve essere sovrascritto.

6) Spiegare brevemente funzionamento, posizionamento all'interno di un calcolatore e scopo della TLB.

2 punti

La TBL (Translation Lookaside Buffer) è una cache specializzata che memorizza le traduzioni tra indirizzi virtuali e indirizzi fisici. Quando la CPU accede ad un indirizzo virtuale, deve tradurlo in indirizzo fisico usando la Page Table (tabella delle pagine). La TBL evita di accedere ogni volta alla tabella delle pagine: se la TBL contiene già la traduzione, allora TBL hit, e la conversione viene data alla CPU; se non è presente, TBL miss, il sistema operativo o l'hardware carica l'informazione da Page Table. La TLB si trova tra la CPU e la MMU, ma è molte volte integrata al processore. Il suo scopo è appunto quello di velocizzare la traduzione degli indirizzi virtuali in fisici, migliorando le prestazioni dell'accesso alla memoria in presenza di paginazione.

6) Descrivere scopo e funzionamento della tabella delle pagine.

2 punti

La tabella delle pagine è una struttura dati usata dal sistema operativo per gestire la memoria virtuale. Viene usata per tradurre indirizzi logici generati dai processi in indirizzi fisici effettivamente usati in RAM e per garantire l'isolamento tra processi. Ogni processo ha una sua tabella delle pagine e quando un processo accede ad un indirizzo virtuale, questo viene diviso in numero di pagina, che serve per cercare nella tabella quale frame fisico (blocco di RAM) contiene quella pagina virtuale, e offset, un puntatore all'interno della pagina per identificare la posizione esatta. Ogni voce nella tabella ha inoltre dei bit di controllo, usati dal sistema operativo, e sono i bit di validità (se la pagina è in memoria o no), il bit di protezione (quindi i permessi, solo lettura, scrittura, ecc), il bit di modifica (se è stato modificato di recente), il bit di accesso (se è stato usato di recente). Ha anche un caching chiamato TBL, che salva le pagine più utilizzate / utilizzate di recente per velocizzare il processo.

6) Spiegare il funzionamento della paginazione chiarendo bene il ruolo dell'hardware in una sua possibile realizzazione in un elaboratore.

2 punti

La paginazione è una tecnica di gestione della memoria che suddivide lo spazio degli indirizzi della memoria virtuale in blocchi di dimensione fissa chiamati pagine, mentre i blocchi della memoria fisica sono chiamati frame. Lo spazio di indirizzamento dei processi è suddiviso in pagine opportunatamente caricate quando necessario.
Ogni processo utilizza indirizzi virtuali, che devono essere tradotti in indirizzi fisici. La tabella delle pagine associa ogni pagina virtuale al corrispondente frame fisico.
L'hardware che esegue questi compiti è la MMU, Memory Management Unit. Esegue la traduzione automatica degli indirizzi virtuali in fisici, consultando la page table. Può anche usare una cache, la TLB (Translation Lookaside Buffer) per velocizzare l'accesso.