Appunti di Architettura

Architettura degli Elaboratori (prof. Fersini) - CdL Informatica Unimib - 23/24

Federico Zotti

2024-07-01

Indice

1 Sistemi Numerici

1.1 Introduzione

I calcolatori utilizzano, a differenza di noi, il **sistema binario**. Questo perché la corrente elettrica può rappresentare solo due stati: acceso (*1*) e spento (*0*).

Sono stati definiti degli **standard di codifica**: regole che vengono utilizzate nella rappresentazione dei dati in formato binario.

Con il termine **bit** definiamo l'**unità di misura dell'informazione**. Combinando tra loro più bit si ottengono strutture più complesse. In particolare:

- Nybble (o nibble): 4 bit
- Byte: 8 bit
- Halfword: 16 bit
- Word: 32 bit
- Doubleword: 64 bit

Dati k bit, il numero di configurazioni ottenibili è pari a 2^k.

Una **rappresentazione** è un modo per descrivere un'entità. Bisogna distinguere l'entità (o valore) e la sua rappresentazione.

1.2 Vari sistemi numerici

Il **sistema numerico decimale**:

- Usa 10 cifre
- È un **sistema posizionale**: ogni cifra assume un valore diverso a seconda della posizione che occupa all'interno del numero

Il **sistema romano** invece non è posizionale (il valore della cifra non dipende dalla sua posizione).

Nei sistemi numerici posizionali un valore numerico N è caratterizzato dalla seguente rappresentazione:

$$N = d_{n-1}\, d_{n-2} \dots d_1\, d_0\,,\, d_{-1} \dots d_{-m}$$

$$N = d_{n-1} \cdot r^{n-1} + \dots + d_0 \cdot r^0 + d_{-1} \cdot r^{-1} + \dots d_{-m} \cdot r^{r-m}$$

$$N = \sum_{i=-m}^{n-1} d_i \cdot r^i$$

Dove: - d è la singola cifra - r la radice o base del sistema - n numero di cifre della parte intera (sinistra della virgola) - m numero di cifre della parte frazionaria (destra della virgola) - N è la rappresentazione del numero

1.2.1 Sistema Binario

Un **byte** è una sequenza di 8 bit consecutivi. Il bit *"più a sinistra"* è chiamato **MSB** (most significant bit) ovvero il bit che rappresenta la cifra con il valore più grande. Il bit *"più a destra"* è chiamato **LSB** (least significant bit) ovvero il bit che rappresenta la cifra con il valore più piccolo.

2 Rappresentazione di numeri interi con segno

2.1 Operazioni aritmetiche

2.1.1 Somma

La somma di due sequenze di bit è la somma tra i bit di pari ordine:

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 0$ con riporto 1 sul bit di ordine superiore

- $1 + 1 + 1 = 1$ con riporto 1 sul bit di ordine superiore

Dunque la somma è definita su 3 elementi:

- I due addendi
- Il riporto (carry)

2.1.2 Sottrazione

La sottrazione di due sequenze di bit è la sottrazione tra i bit di pari ordine:

- $0 - 0 = 0$
- $1 - 0 = 1$
- $1 - 1 = 0$
- $0 - 1 = 1$ con prestito 1 dal bit di ordine superiore

Dunque la sottrazione è definita su 3 elementi:

- Minuendo e sottraendo
- Il prestito (borrow)

2.2 Modulo e Segno

Supponiamo di avere a disposizione 1 Byte per rappresentare numeri sia positivi che negativi.

Con il metodo del **modulo e segno** utilizzeremo:

- I primi 7 bit per il valore assoluto del numero
- Il bit più a sinistra *(MSB)* per indicare il segno (1 se il numero è negativo, 0 se è positivo)

Con n bit totali si possono rappresentare i numeri interi nell'intervallo

$$\left[-(2^{n-1} - 1), +(2^{n-1} - 1)\right]_{10}$$

I problemi di questa rappresentazione sono che esistono due diverse rappresentazioni dello 0 e che un bit viene "speso" solo per il segno.

2.2.1 Somma

Confronto i bit di segno dei due numeri:

- Se i bit di segno sono uguali:
  - Il bit di segno risultante sarà il bit di segno dei due addendi
  - Eseguo la somma bit a bit (a meno di overflow)
- Se i bit di segno sono diversi:
  - Confronto i valori assoluti dei due addendi
  - Il bit si segno risultate sarà il bit di segno dell'addendo con valore assoluto
  - Eseguo la somma bit a bit

2.2.2 Sottrazione

Confronto i bit di segno dei due numeri:

- Se i bit di segno sono uguali:
  - Il bit di segno risultante sarà uguale al bit di segno dell'operando a modulo maggiore
  - Il risultato avrà modulo pari al modulo della differenza dei moduli degli operandi
- Se i bit di segno sono diversi
  - Il bit di segno risultante sarà uguale al bit di segno del minuendo
  - Il risultato avrà il modulo pari alla somma dei moduli dei due operandi

2.2.3 Overflow

Si può avere overflow solo quando:

- Si sommano due operandi con segno concorde
- Si sottraggono due operandi con segno discorde

2.3 Complemento a 1

Questo metodo si basa sull'**operazione di complemento**. Con complemento si intende l'operazione che associa ad un bit il suo opposto.

Il metodo del **complemento a 1** è molto semplice:

- Se il numero da codificare è **positivo**, lo si converte in binario con il metodo tradizionale
- Se il numero è **negativo**, basta convertire in binario il suo modulo e quindi eseguire l'operazione di complemento sulla codifica binaria effettuata

Anche in questo caso sussiste il problema delle due diverse rappresentazioni dello 0.

2.4 Complemento a 2

Il complemento a 2 è un altro metodo di codifica usato per rappresentare i numeri interi sia positivi che negativi. È basato sul complemento a 1.

A differenza del complemento a 1 i numeri negativi dopo aver complementato il numero vengono incrementati di 1.

Dati n bit si possono rappresentare i numeri interi nell'intervallo

$$\left[-(2^{n-1}), +(2^{n-1}-1)\right]_{10}$$

2.4.1 Conversione da CA2 a decimale

Se il numero è positivo (MSB = 0), si converte in base decimale usando il numero binario puro. Se il numero è negativo (MSB = 1), si applica l'operazione di CA2 a questo valore ottenendo la rappresentazione del corrispondente positivo, si converte il risultato come numero in binario puro e si aggiunge il segno meno.

2.4.2 Somma

1. Si esegue la somma su tutti i bit degli addendi, segno compreso
2. Un eventuale riporto oltre il bit di segno (MSB) viene scartato
3. Nel caso gli operandi siano di segno concorde occorre verificare la presenza o meno di overflow (si presenta solo se $(+A)+(+B) = -C$ o $(-A)+(-B) = +C$)

2.4.3 Sottrazione

La sottrazione tra due numeri in CA2 viene trasformata in somma applicando la regola

$$A - B = A + (-B)$$

2.5 Shift

Nel caso lo shift sia con un numero MS il segno non viene considerato e viene mantenuto.

Nel caso lo shift sinistro sia con un numero CA2 il segno non viene considerato (si sposta come tutti gli altri bit) e se il nuovo MSB è diverso dal precedente c'è un overflow. Nello shift destro il segno viene replicato.

3 Rappresentazione numeri reali e altre informazioni

I numeri reali possono essere rappresentati sia in virgola fissa *(fixed point)* che in virgola mobile *(floating point)*.

3.1 Numeri in virgola fissa

Un sistema di numerazione in **virgola fissa** è quello in cui si riserva un numero fisso di bit per la parte intera e la parte frazionaria. La posizione della virgola è **implicita** e uguale per tutti i numeri.

3.1.1 Unsigned fixed point

Per i numeri **unsigned** fixed point, dati n bit a disposizione

- $i < n$ bit per rappresentare la **parte intera** del numero
- $d = n - i$ bit per rappresentare la **parte decimale** del numero

Con questo metodo l'intervallo di numeri interi rappresentabili è

$$[0, 2^i - 1]$$

e l'intervallo rappresentabile dalla parte decimale è

$$[0, 2^d - 1]$$

3.1.2 Signed fixed point

Per i numeri **signed** fixed point, dati n bit a disposizione

- Un bit per il segno del numero da rappresentare
- $i < (n-1)$ bit per rappresentare la **parte intera** del numero
- $d = n - (i+1)$ bit per rappresentare la **parte decimale** del numero

Con questo metodo l'intervallo di numeri interi rappresentabili è

$$\left[-2^{i-1} - 1, \quad 2^{i-1} - 1\right]$$

e l'intervallo rappresentabile dalla parte decimale è

$$\left[0, 2^{d} - 1\right]$$

3.2 Numeri in virgola mobile

Nella notazione in **virgola mobile** un numero N è esprimibile come

$$N = \pm M \cdot B^{\pm E}$$

Vengono usati:

- Un bit per il segno
- n bit per la mantissa
- m bit per l'esponente

Dunque la vera rappresentazione in binario sarà

$$N = (-1)^{S} \cdot M \cdot B^{E}$$

Il numero rappresentato deve essere **normalizzato**, ovvero viene trasformato utilizzando solo una cifra intera:

$$1101.10011 \rightarrow 1.110110011$$

Dunque essendo il primo bit (la parte intera) sempre 1, non viene memorizzato e viene definito come **"bit nascosto"**.

Per rappresentare -53.5 in floating point 32 bit:

$$-53.5 = (-110101.1)_2 = (-1)^{(1)_2} \cdot (1.101011)_2 \cdot 2^{(101)_2}$$

```
|   1    |  00000101   |  10101100000..00   |
| Segno  |  Esponente  |      Mantissa      |
```

Lo standard che definisce la rappresentazione dei numeri in virgola mobile è lo **IEEE 754** (1985). Specifica il formato, le operazioni e le conversioni tra i diversi formati floating point e quelle tra i diversi sistemi di numerazioni.

Secondo questo standard l'esponente ha 8 bit (rappresentato in eccesso 127, polarizzato). I valori estremi -127 e 128 sono riservati. Dunque il numero più grande che può essere rappresentato (dall'esponente) è `111...111` e il più piccolo `000...000`. Per confrontare due esponenti basta considerarli interi senza segno.

Quindi in IEEE 754 un numero N in virgola mobile viene rappresentato come

$$N = (-1)^S \cdot (1 + 0.M) \cdot 2^{E-127}$$

3.2.1 Errore assoluto e Errore relativo

Rappresentando un numero reale n in virgola mobile si commette un errore di approssimazione. Questo perché viene rappresentato un numero razionale n' con un numero limitato di cifre significative.

L'errore assoluto è definito come

$$e_A = n - n'$$

e l'errore relativo come

$$e_R = \frac{e_A}{n} = \frac{n - n'}{n}$$

L'ordine di grandezza dell'errore assoluto dipende dal numero di cifre significative e dall'ordine di grandezza del numero. L'ordine di grandezza dell'errore relativo dipende solo dal numero di cifre significative.

3.3 Rappresentazione di caratteri

Possiamo associare ad ogni carattere un numero. Dunque possono essere rappresentati con diverse codifiche:

- ASCII standard: un carattere è rappresentato con 7 bit (128 simboli totali)
- ASCII esteso: un carattere è rappresentato con 8 bit (256 simboli totali)
- Unicode: un carattere è rappresentato con un numero maggiore di bit (da 8 a 32 bit per carattere)

4 Logica combinatoria

Un **circuito combinatorio** è un circuito nel quale lo stato delle uscite dipende solo dalla funzione logica applicata allo stato istantaneo (cioè in un determinato istante di tempo) delle sue entrate. Questo si differenzia dal **circuito sequenziale** nel quale lo stato delle uscite non dipende solo dalla funzione logica, ma anche sulla base di valori pregressi collocati in memoria.

4.1 Porte logiche

Le **porte logiche** sono i componenti elettronici che permettono di svolgere le operazioni logiche primitive oltre che a quelle direttamente derivate. Le porte

logiche hanno n input e generalmente un output. A ogni combinazione di valori in ingresso corrisponde una e solo una combinazione di valori in uscita.

Le porte logiche **fondamentali** sono:

- AND
- OR
- NOT

Le porte logiche **derivate** sono:

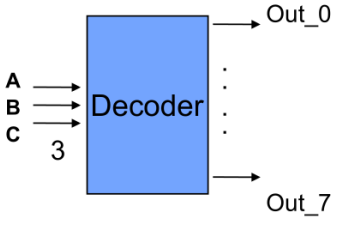
- NAND
- NOR
- XOR

Ad eccezione della porta NOT, tutte le altre porte possono esistere anche ad n ingressi collegando **a cascata** tra loro porte a due ingressi.

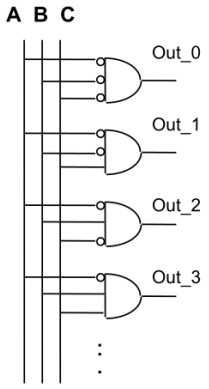
Le porte NOR e NAND che svolgono la funzione di inverter sono definite **universali**:

4.2 Decoder

Un **decoder** è un componente elettronico che ha n ingressi e 2^n uscite. Il suo scopo è di impostare allo stato alto l'uscita corrispondente alla conversione in base 10 della codifica binaria a n bit ricevuta in input (e di impostare allo stato basso tutte le altre).



A	B	C	0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

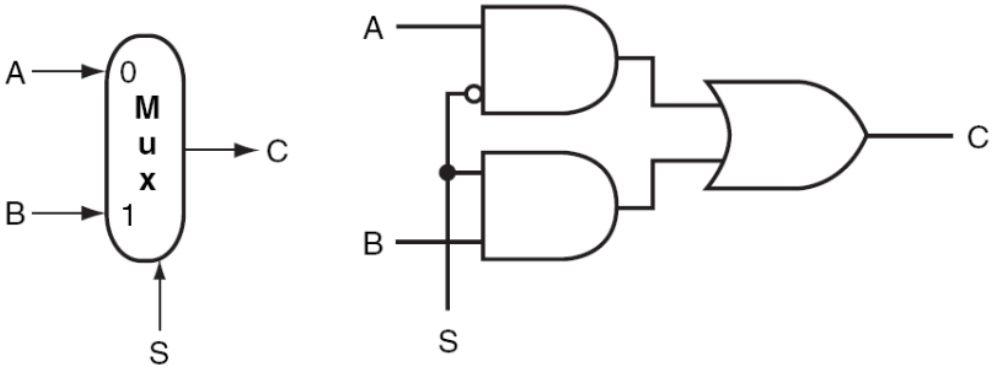


4.3 Multiplexer

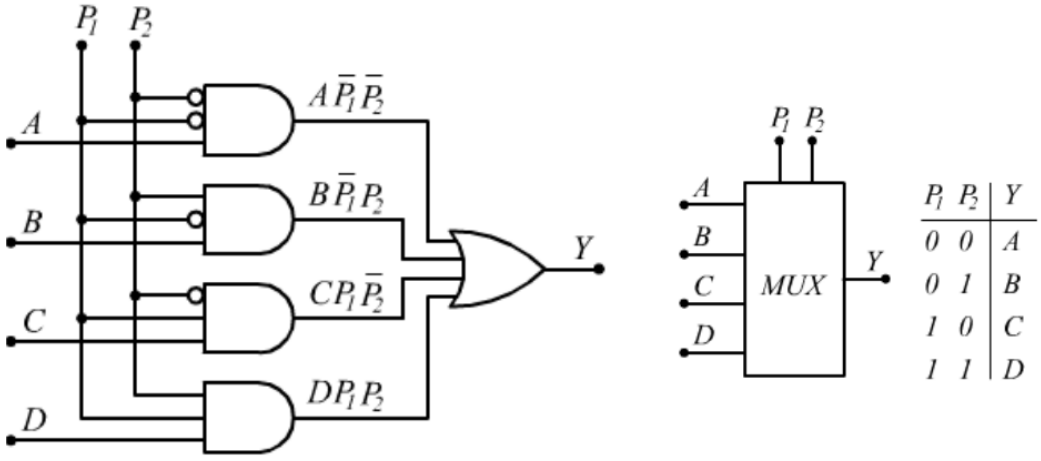
Un **multiplexer** (selettore) è un componente elettronico caratterizzato da 2^n entrate principali, n entrate di controllo (*selettore*) e un uscita. Il valore del selettore determina quale input diviene output.

Esempio con selettore ad un bit

$$C = (A \cdot \bar{S}) + (B \cdot S)$$



Esempio con selettore a due bit



Se un multiplexer riceve in input n segnali, esso necessiterà di $\log_2 n$ selettori, e consisterà di un decoder che genererà n segnali, un array di n porte logiche AND e

un'unica porta logica OR.

4.4 Logiche a due livelli e PLA

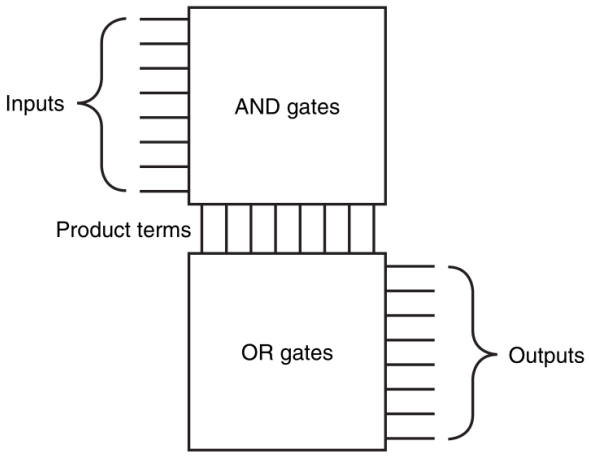
Utilizzando le porte logiche AND, OR e NOT è possibile implementare qualunque funzione logica più complessa. Dunque possiamo creare logiche a due livelli:

- Somma *(OR)* di prodotti *(AND)*
- Prodotto *(AND)* di somme *(OR)*

Per esempio possiamo rappresentare una qualsiasi tabella di verità D come somma di prodotti

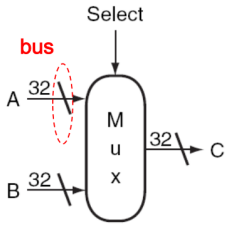
$$D = (\bar{A} \cdot \bar{B} \cdot C) + (\bar{A} \cdot B \cdot \bar{C}) + (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot B \cdot C)$$

La somma di prodotti corrisponde ad una implementazione comunemente nota come **Programmable Logic Array** (PLA)

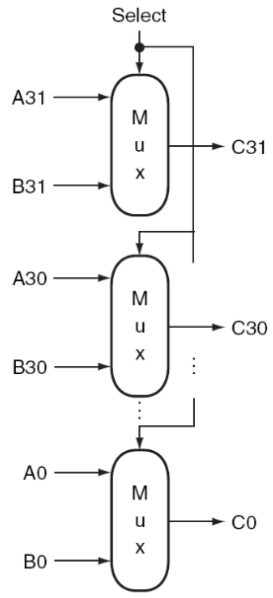


4.5 Array di elementi logici

La maggior parte delle operazioni vengono svolte su 32 bit. Questo implica la necessita di creare **array di elementi logici**. Un **bus** è una collezione di linee di input che verranno trattate come un singolo segnale:



Per esempio un multiplexer a 32 bit corrisponde ad un array di 32 multiplexer a 1 bit

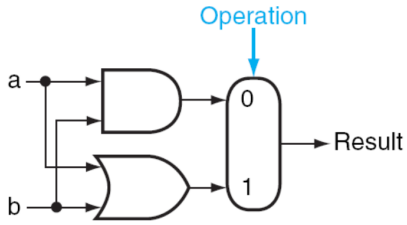


4.6 ALU

L'**Arithmetic Logic Unit** è la parte del processore che svolge le operazioni aritmetico-logiche. Essa è un insieme di circuiti combinatori che implementa operazioni aritmetiche (somma e sottrazione) e operazioni logiche (AND e OR).

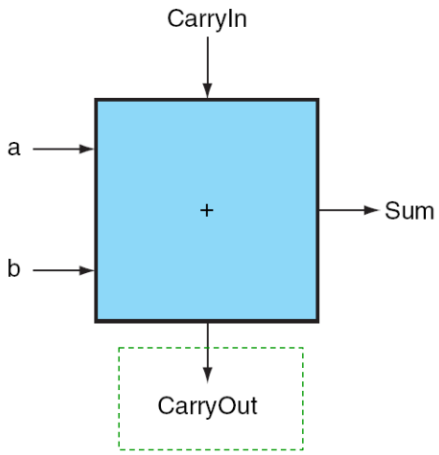
4.6.1 Operazioni logiche

ALU ad un bit che implementa AND e OR:



4.6.2 Addizione

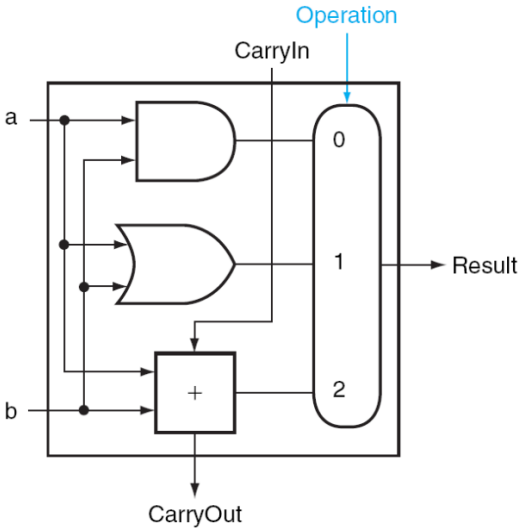
Per l'addizione è sufficiente implementare una PLA con input i due bit da sommare con il carry in ingresso e in output la somma con il carry in uscita.



$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

$$\begin{aligned}\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) \\ + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})\end{aligned}$$

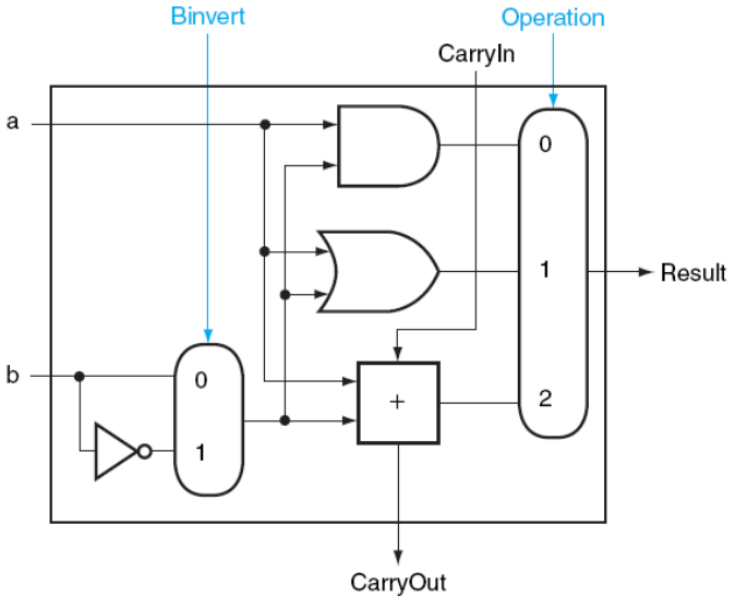
Questo blocco si può aggiungere all'ALU:



4.6.3 Sottrazione

Per aggiungere l'operazione di sottrazione si può approfittare della codifica CA2:

$$a - b = a + (\bar{b} + 1)$$



Bisogna solo ricordarsi di settare il `CarryIn` e `Binvert` a 1.

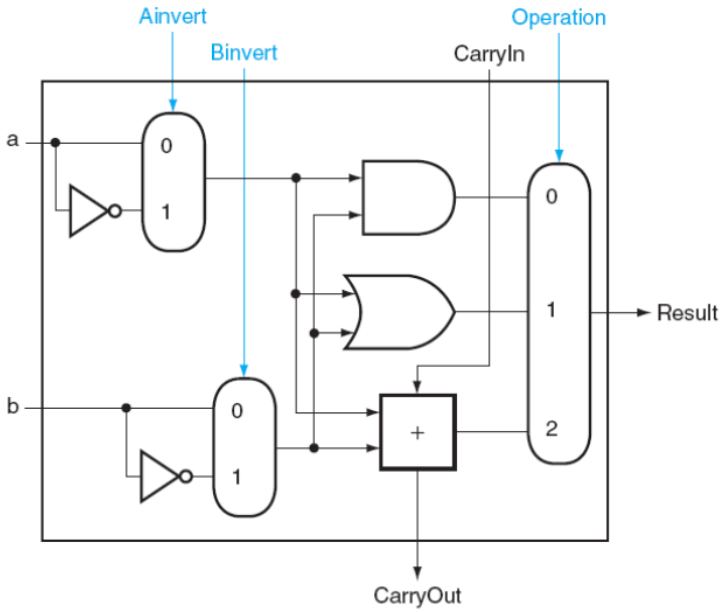
4.6.4 Porta NOR

Per implementare la porta NOR si possono sfruttare le leggi di De Morgan:

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

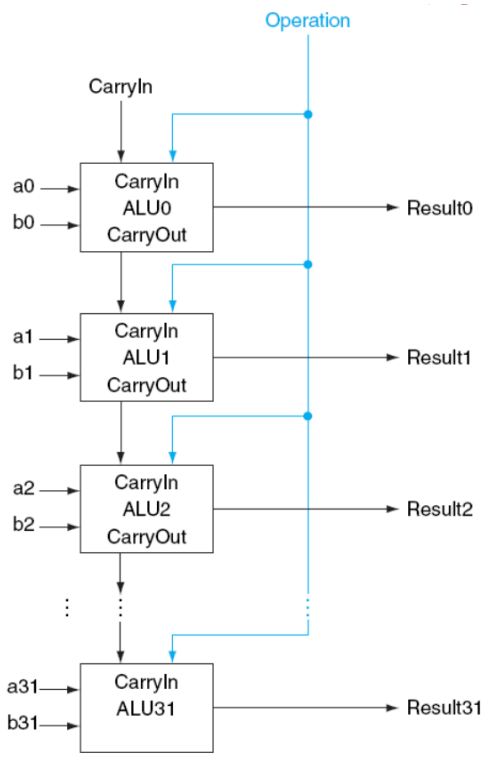
$$\overline{(a \cdot b)} = \bar{a} + \bar{b}$$

Dunque si può aggiungere un input `Ainvert` che inverte il bit `a`



4.6.5 ALU a 32 bit

Per rendere un ALU a 32 bit basta concatenare molte ALU a un bit:



4.6.6 Operazione Set-on-less-then (SLT)

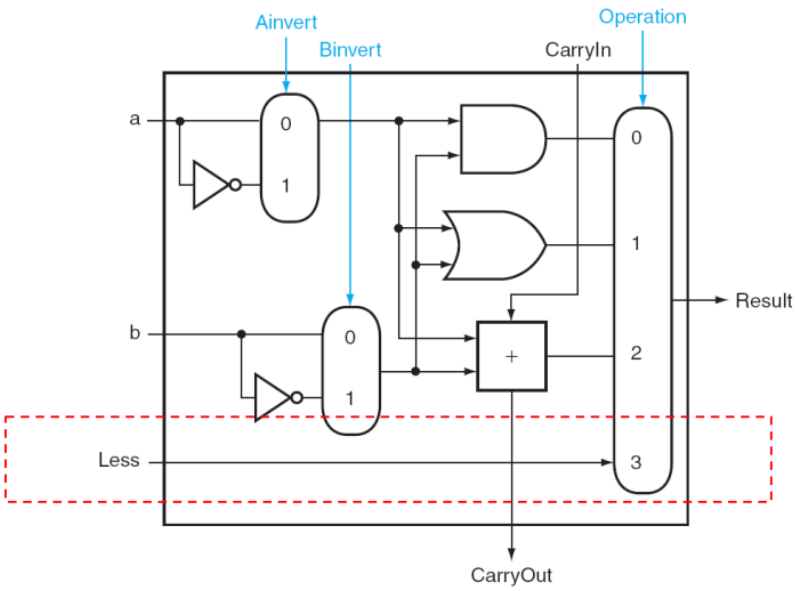
Questa operazione ritorna 1 se $a < b$, altrimenti 0. Per eseguire questa istruzione si devono poter azzerare tutti i bit dal bit 1 al bit 32 ed assegnare al bit 0 il valore del risultato.

$$\begin{aligned} a < b &\iff a - b < 0 \\ &\iff (a - b) \text{ è negativo} \\ &\iff \text{bit 31 di } (a - b) = 1 \end{aligned}$$

Per poter effettuare il confronto si sottrae a e b.

- Se $a - b$ è minore di 0, allora $a < b$: il risultato sarà $00 \dots 01$
- Se $a - b$ è maggiore di 0, allora $a > b$: il risultato sarà $00 \dots 00$

Dunque per implementare questa istruzione all'ALU si aggiunge l'input `Less`:

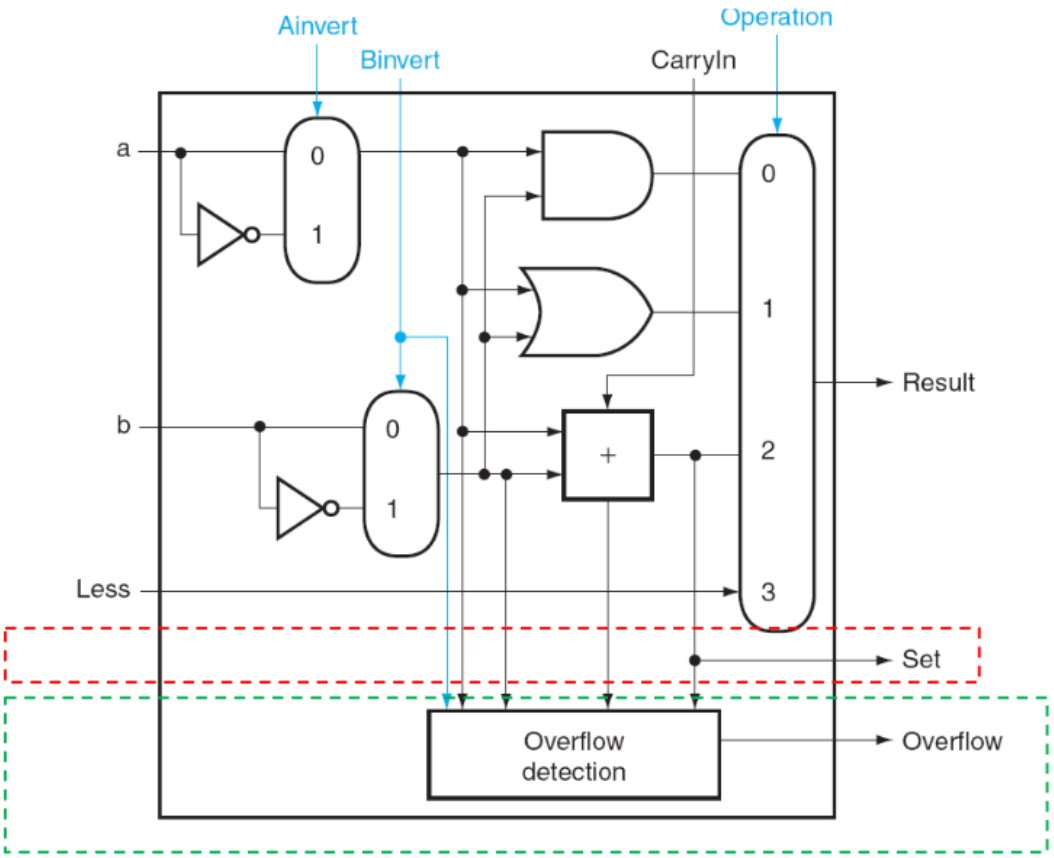


Per le ALU dal bit 1 *(il secondo)* al bit 31 *(l'ultimo)* i collegamenti sono:

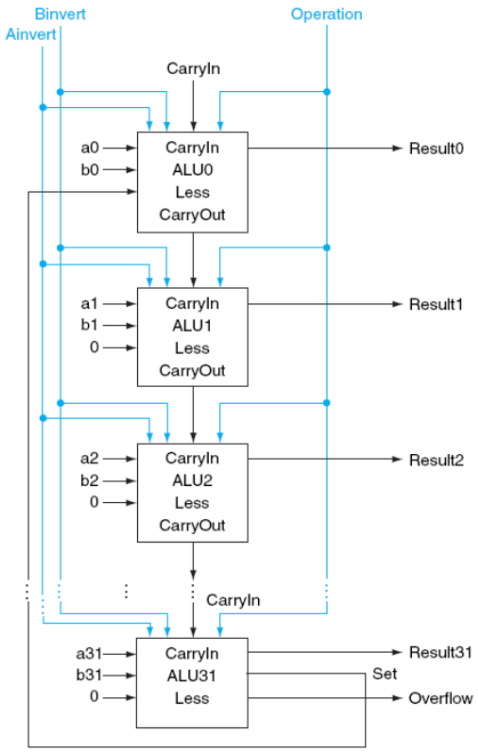
- $\texttt{CarryIn}_i = \texttt{CarryOut}_{i-1}$
- $\texttt{Less} = 0$

Per l'ALU del bit 31 inoltre si aggiunge l'output `Set` e `Overflow`:

- `Set` rappresenta il segno del numero $a - b$, dunque va collegato al `Less` dell'ALU al bit 0 *(il primo)*; in questo modo l'output complessivo dell'ALU sarà il bit di segno di $a - b$ al primo posto seguito da tutti gli zeri
- `Overflow` indica se c'è stato un overflow nell'ALU: $\bar{a} \cdot b \cdot \text{Result} + a \cdot \bar{b} \cdot \overline{\text{Result}}$



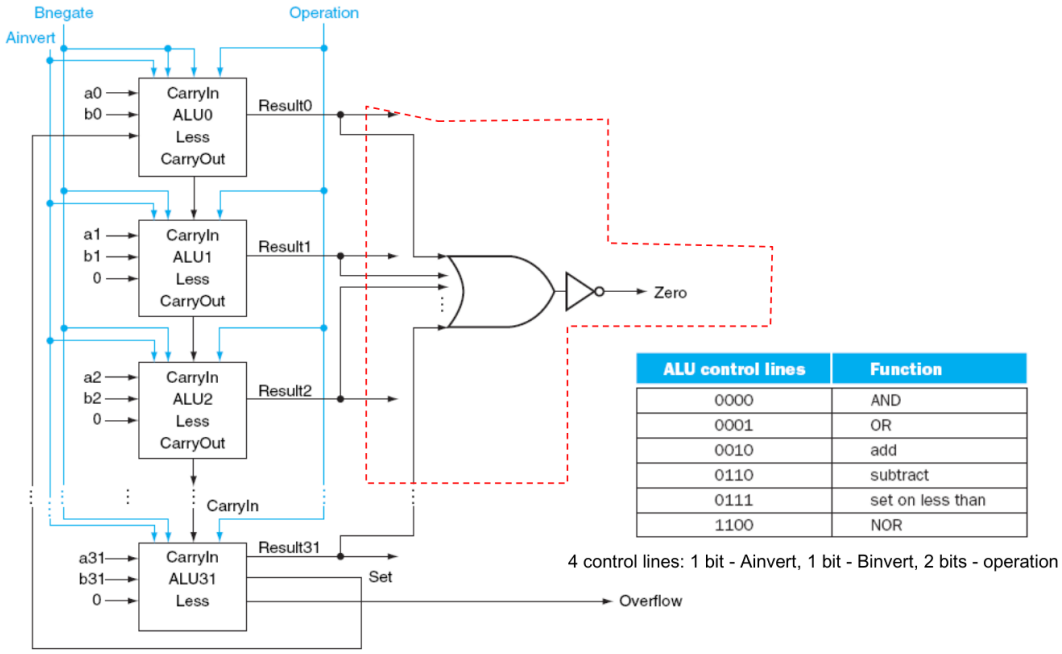
Tutto questo diventa



- Per sottrarre: `Binvert` e `CarryIn` a 1
- Per addizioni e operazioni logiche: `Binvert` e `CarryIn` a 0
- Per unificare i due segnali in uno: `Bnegate` a 1

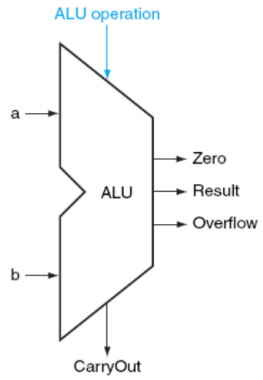
4.6.7 Operazione Branch-on-equal (BEQ)

Questa istruzione ritorna 1 se i numeri sono uguali. Per farlo viene eseguita una sottrazione $a - b$, vengono moltiplicate (OR) tutte le uscite e il risultato viene negato. Viene dunque aggiunta un'altra uscita `Zero`.



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

4.6.8 Rappresentazione simbolica



5 Logica sequenziale

Come già detto in precedenza, un **circuito sequenziale** si differenzia da un circuito combinatorio perché lo stato delle uscite non dipende solo dalla funzione logica, ma anche sulla base di valori pregressi collocati in memoria. In generale la funzione calcolata da un circuito sequenziale in un certo momento dipende dalla sequenza temporale dei valori in input al circuito. La sequenza temporale determina il valore

memorizzato nello stato.

5.1 Clock

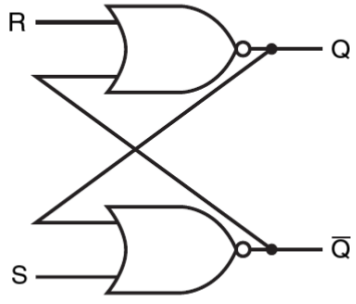
Il segnale di **clock** è fondamentale per le reti sequenziali che sono caratterizzate da uno **stato**. Il clock è definito come un segnale (onda quadra) con un periodo predeterminato e costante. Il periodo di clock deve essere abbastanza grande da assicurare la stabilità degli output di un circuito. Questo perché i transistor che compongono le porte logiche non sono istantanei, ma impiegano un po' di tempo per "commutarsi". Il clock determina quando il contenuto di un elemento che rappresenta lo stato è aggiornato. Determina il ritmo dei calcoli e delle operazioni di memorizzazione. Con il clock un circuito diventa **sincrono**.

Esistono due famiglie di circuiti digitali sequenziali:

- **Asincroni**: non fanno uso di clock
- **Sincroni**: necessitano di un clock

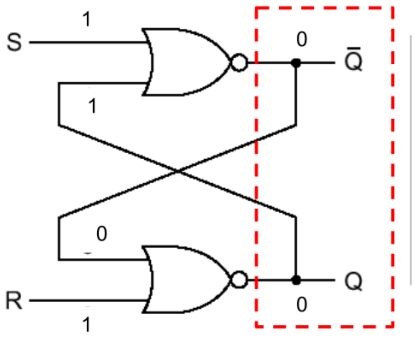
5.2 S-R Latch

L'**S-R Latch** è un circuito che ci permette di memorizzare un singolo bit (stato). È composto da due porte NOR concatenate. **S** significa Set, mentre **R** significa Reset.



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Da notare che la configurazione $S = 1$ e $R = 1$ viola la proprietà di complementarietà degli output e può portare ad una configurazione instabile.



Input S	Input R	Stato Interno Old Q	Output Q	Output $\bar{Q}$
0	0	0	0	1
0	0	1	1	0
0	1	1/0	0	1
1	0	1/0	1	0
1	1	1/0	0	0

Generalmente gli input S e R sono calcolati da un circuito combinatorio. L'output del circuito diventa però stabile dopo un certo intervallo di tempo che dipende dal numero di porte attraversate e bisogna evitare che durante questo intervallo gli output intermedi del circuito vengano memorizzati. La soluzione è **aggiungere un clock** al latch. Aggiungendo il clock questo circuito diventa **sincrono**.

5.3 D Latch

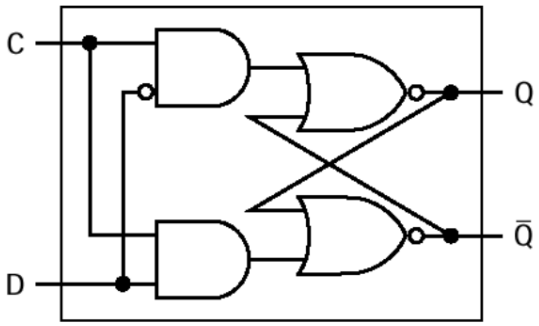
Il **D Latch** è un latch sincronizzato da un clock. Questo clock garantisce che il latch cambi stato solo in certi momenti.

D=1 corrisponde al *setting*

- S=1 e R=0

D=0 corrisponde al *resetting*

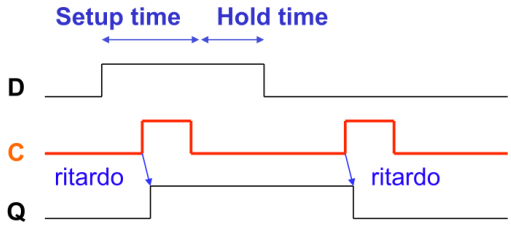
- S=0 e R=1



Quando il clock è **deasserted** (valore 0) non viene memorizzato nessun valore (viene mantenuto il valore precedentemente memorizzato). Quando il clock è **asserted** (valore 1) invece viene memorizzato un valore in funzione del valore di D.

Il segnale D, ottenuto come output di un circuito combinatorio deve:

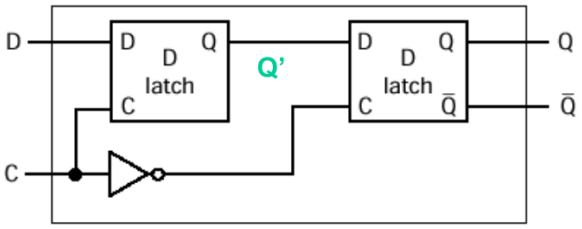
- Essere già stabile quando il clock diventa asserted
- Rimanere stabile per tutta la durata del livello alto del clock *(setup time)*
- Rimanere stabile per un altro periodo di tempo per evitare malfunzionamenti *(hold time)*



5.4 D Flip-Flop

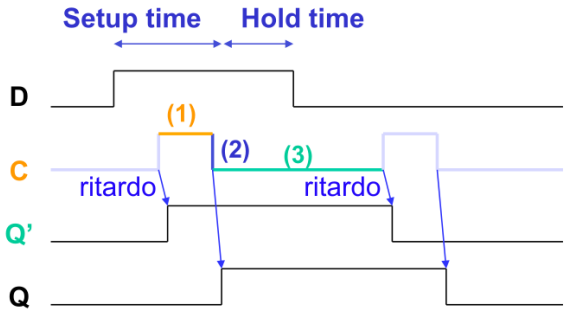
A differenza del D Latch che funziona con una metodologia detta **level-triggered** (ovvero la memorizzazione avviene per tutta la durata del livello alto del clock), il **D Flip-Flop** è un circuito che lavora con una metodologia detta **edge-triggered**, nel quale la memorizzazione avviene sul **fronte di salita** *(rising edge)* (o più raramente di discesa) del clock.

Il D Flip-Flop è utilizzabile come input e output durante lo stesso ciclo di clock. Viene realizzato ponendo in serie due D Latch: il primo viene detto **master** e il secondo **slave**.



(1) Il primo latch è aperto e pronto per memorizzare D. Il valore memorizzato Q' fluisce fuori, ma il secondo latch è chiuso

- => nel circuito combinatorio a valle entra ancora il vecchio valore di Q



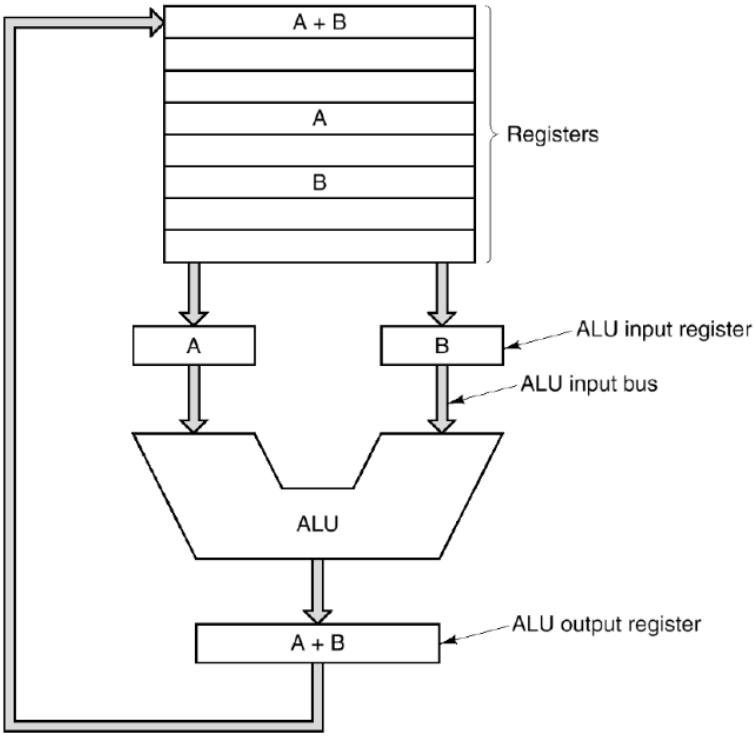
(2) Il segnale del clock scende, e in questo istante il secondo latch viene aperto per memorizzare il valore di Q'

(3) Il secondo latch è aperto, memorizza D (Q'), e fa fluire il nuovo valore Q nel circuito a valle. Il primo latch è invece chiuso, e non memorizza niente

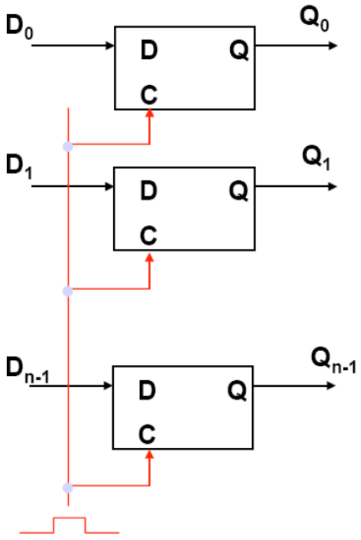
Il segnale D deve essere attivo per un periodo abbastanza lungo (setup time + hold time).

6 Register file

Un **registro** è costituito da n flip-flop. Nel MIPS ogni registro è di 1 word (32 bit). I registri sono organizzati in un **register file**. Il register file del MIPS ha 32 registri (1024 flip-flop totali). Esso permette la lettura di 2 registri e la scrittura di 1 registro.



Nel **datapath** della CPU il clock non entra direttamente nei vari flip-flop, ma viene messo in AND con un segnale di controllo `Write`. Il segnale `Write` determina se, in corrispondenza del fronte di discesa del clock, il valore `D` debba (o meno) essere memorizzato nel registro.



Rappresentazione del register file:

Read Reg1 # (5 bit)

- numero del 1° registro da leggere

Read Reg2 # (5 bit)

- numero del 2° registro da leggere

Read data 1 (32 bit)

- valore del 1° registro, letto sulla base di Read Reg1 #

Read data 2 (32 bit)

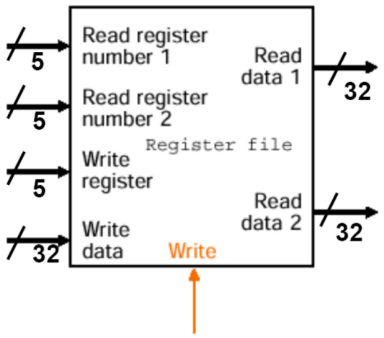
- valore del 2° registro, letto sulla base di Read Reg2 #

Write Reg # (5 bit)

- numero del registro da scrivere

Write data (32 bit)

- valore da scrivere nel registro Write Reg #

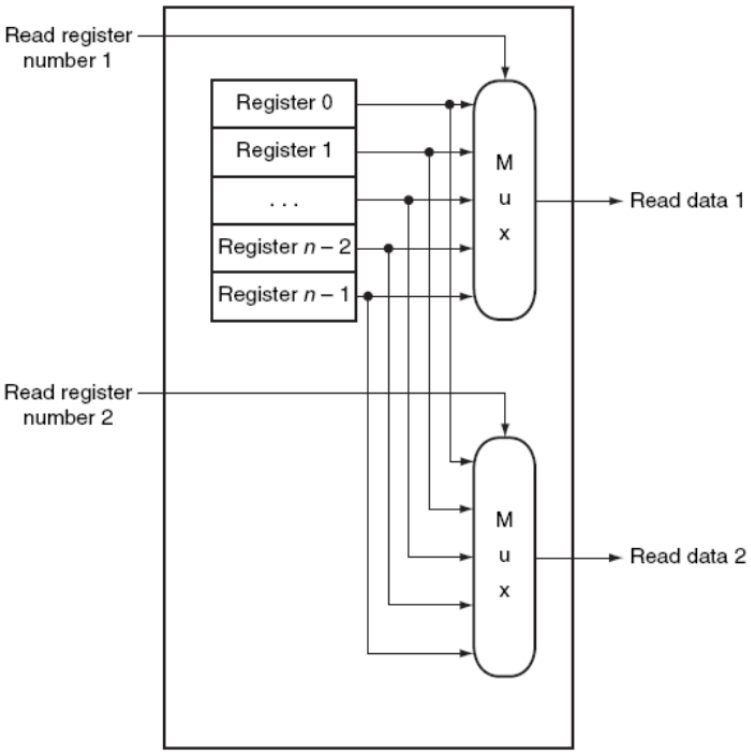


Write

- segnale di controllo messo in AND con il *clock*
- solo se Write=1, il valore di Write data viene scritto in uno dei registri

6.1 Lettura dal register file

Il register file utilizza due segnali che indicano i registri da leggere (`Read Reg1` e `Read Reg2`) e vengono collegati a due multiplexer. Il register file fornisce **sempre** in output una coppia di registri.

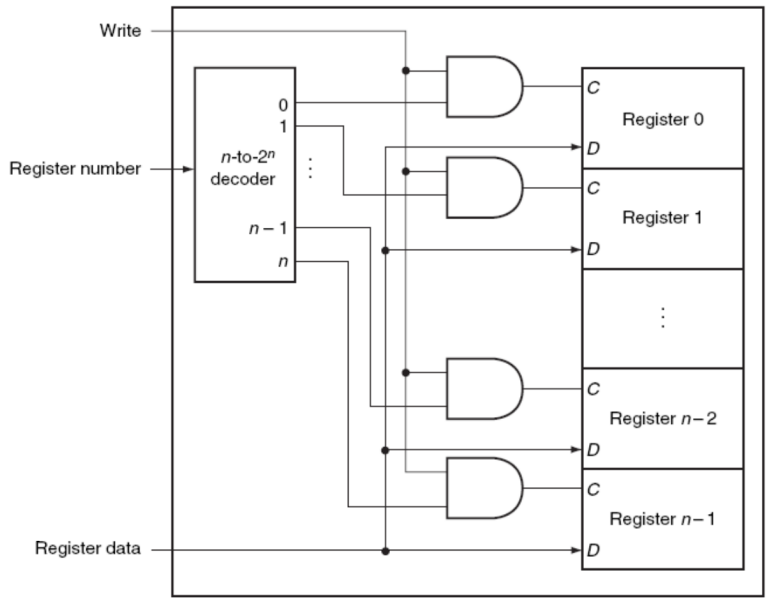


6.2 Scrittura nel register file

Per la scrittura nel register file vengono utilizzati tre segnali:

- Il registro da scrivere (`Register number`)
- Il valore da scrivere (`Register data`)
- Il segnale di controllo (`Write`)

Viene utilizzato un decoder che decodifica il numero del registro da scrivere. Se `Write` non è a 1 nessun valore sarà scritto nel registro.

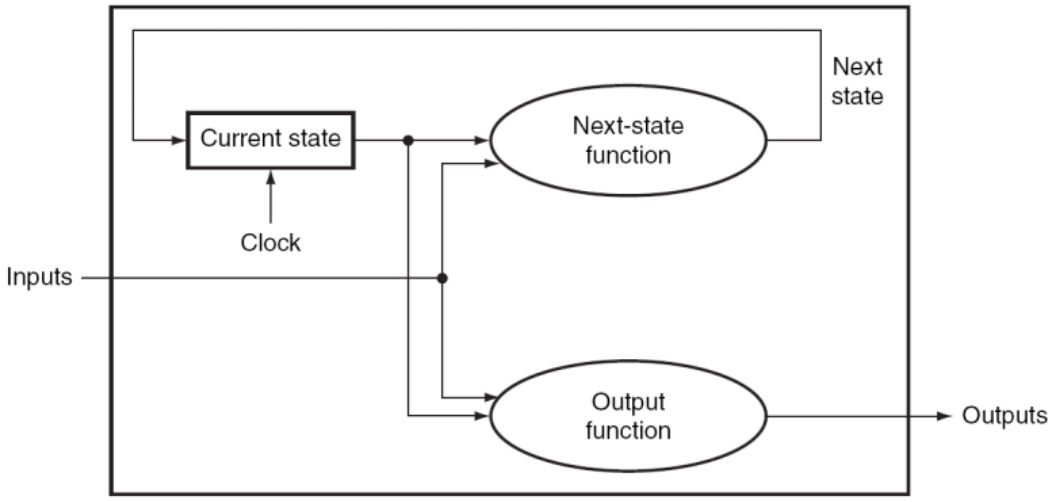


7 Macchine a stati finiti

Le **macchine a stati finiti** *(finite state machine/automata)* sono usate per descrivere i circuiti sequenziali. Sono composte da un set di stati e due funzioni:

- **Next state function**: determina lo stato successivo partendo dallo stato corrente e dai valori in ingresso
- **Output function**: produce un insieme di risultati partendo dallo stato corrente e dai valori in ingresso (solo nel caso di una *FSM Mealy*)

In questo corso vengono utilizzate solo **FSM Moore** che, a differenza delle FSM Mealy, non utilizzano i valori in ingresso nella funzione di output. Esse sono sincronizzate con il clock.



7.1 Esempio: Semaforo

In questo esempio viene rappresentato un semaforo a due colori: rosso e verde.

Segnali di input:

- `NScar`
- `EWcar`

Segnali di output:

- `NSlite`
- `EWlite`

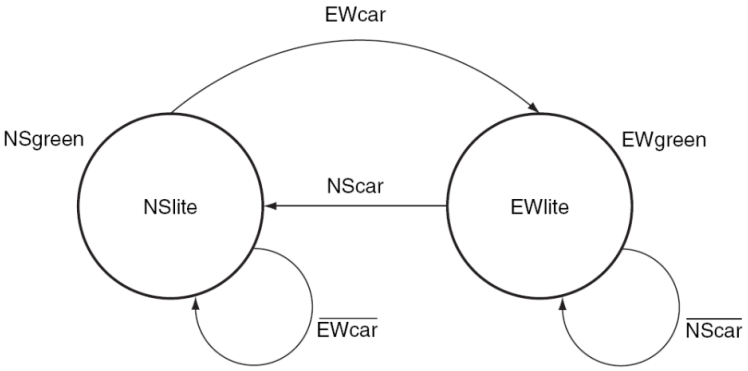
Possibili stati:

- `NSgreen`
- `EWgreen`

Lo stato cambia quando arrivano macchine al semaforo.

	Inputs		
	NScar	EWcar	Next state
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1



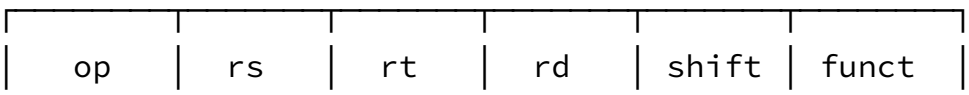
8 ISA

> Noi useremo l'ISA di **MIPS IV**.

Contenuto del processore:

- 32 registri
- …

8.1 Istruzioni R-Type



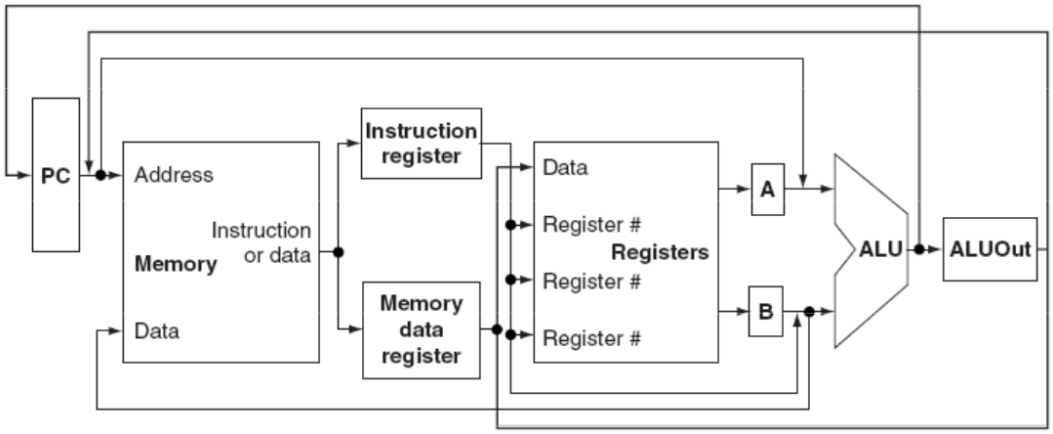
000000	01000	01001	01010	00000	100000
6b	5b	5b	5b	5b	6b

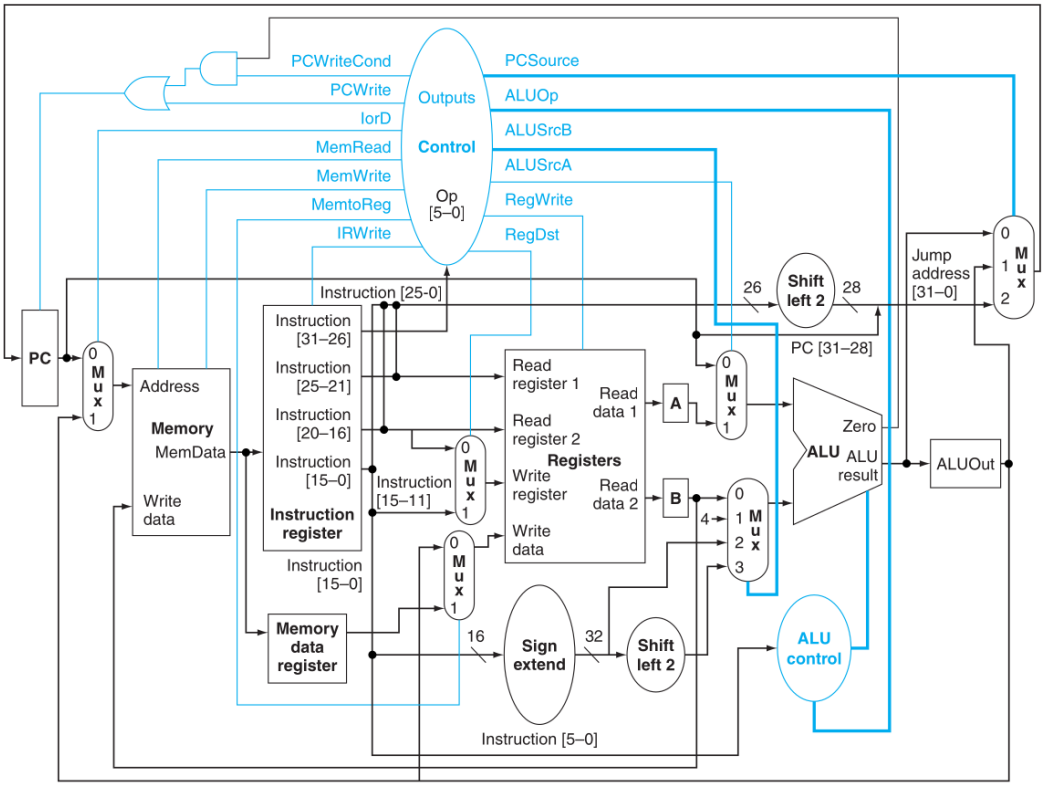
8.2 Istruzioni I-Type

op	rs	rt	immediate
000000	01000	01001	0101000000100000
6b	5b	5b	16b

9 Datapath

> Nel pdf viene trattato il datapath “singolo ciclo”. Noi invece studiamo il datapath “multi ciclo” (presente in un pdf vecchio).





9.1 Segnali di controllo

9.1.1 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

9.1.2 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSource	00	Output of the ALU (PC + 4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address (IR[25:0] shifted left 2 bits and concatenated with PC + 4[31:28]) is sent to the PC for writing.

9.2 Esecuzione di un'istruzione

Ogni istruzione viene eseguita utilizzando da 3 a 5 cicli di clock.

1. **Fetch**: fetch dell'istruzione dalla memoria e incremento di PC
2. **Decode**: decodifica l'istruzione, legge i registri e calcolo dell'indirizzo di un eventuale branch
3. **Execute 1**: esegue operazioni (*R-type*), o calcolo di memoria, o completa un branch, o completa una jump
4. **Execute 2**: completa le operazioni (*R-type*), o accesso alla memoria

5. **Execute 3**: scrittura registro (*lw*)

9.2.1 Fetch

Operazioni:

- `IR = Mem[PC]`
- `PC = PC + 4`

Segnali di controllo utilizzati:

- **`MemRead`** per leggere la memoria
- **`IRWrite`** per scrivere `IR`
- **`IorD`** per indicare l'indirizzo da dove leggere la memoria
- **`ALUSrcA`**, **`ALUSrcB`**, **`ALUop`** per incrementare il `PC`
- **`PCWrite`** per salvare il nuovo valore del `PC`

9.2.2 Decode

Operazioni:

- `A = Reg[ IR[25:21] ]`
- `B = Reg[ IR[20:16] ]`
- `ALUOut = PC + (sign-extend (IR[15:0])) << 2`

Segnali di controllo utilizzati:

- **`ALUSrcA`**, **`ALUSrcB`**, **`ALUop`** per il calcolo di un eventuale indirizzo di branch

9.2.3 Execute 1 (`lw`, `sw`)

Operazioni:

- `ALUOut = A + (sign-extend (IR[15:0]))`

Segnali di controllo utilizzati:

- **ALUSrcA**, **ALUSrcB**, **ALUop** per il calcolo dell'indirizzo di memoria per `lw` o `sw`

9.2.4 Execute 1 (R-type aritmetico-logiche)

Operazioni:

- `ALUOut = A op B` *(con op l'operazione da eseguire)*

Segnali di controllo utilizzati:

- **ALUSrcA**, **ALUSrcB**, **ALUop** per il calcolo aritmetico o logico

9.2.5 Execute 1 (beq)

Operazioni:

- `if A == B then PC = ALUOut`

Segnali di controllo utilizzati:

- **ALUSrcA**, **ALUSrcB**, **ALUop** per la comparazione tra A e B
- **PCWriteCond**, **PCSource** per scrivere il PC

9.2.6 Execute 1 (jump)

Operazioni:

- `PC = (PC[31:28], IR[25:0] << 2)`

Segnali di controllo utilizzati:

- **PCWrite**, **PCSource** per scrivere il PC

9.2.7 Execute 2 (`lw`, `sw`)

Operazioni:

- `MDR = Mem[ALUOut]`

oppure

- `Mem[ALUOut] = B`

Segnali di controllo utilizzati:

- **`IorD`** per indicare l'indirizzo di memoria
- **`MemRead`** per leggere dalla memoria (nel caso di `lw`)
- **`MemWrite`** per scrivere nella memoria (nel caso di `sw`)

9.2.8 Execute 2 (R-type aritmetico-logiche)

Operazioni:

- `Reg[ IR[15:11] ] = ALUOut`

Segnali di controllo utilizzati:

- **`RegWrite`** per poter scrivere nel *Register File*
- **`RegDest`** per indicare il registro da scrivere
- **`MemToReg`** per scrivere il valore da `ALUOut`

9.2.9 Execute 3 (`lw`)

Operazioni:

- `Reg[ IR[20:16] ] = MDR`

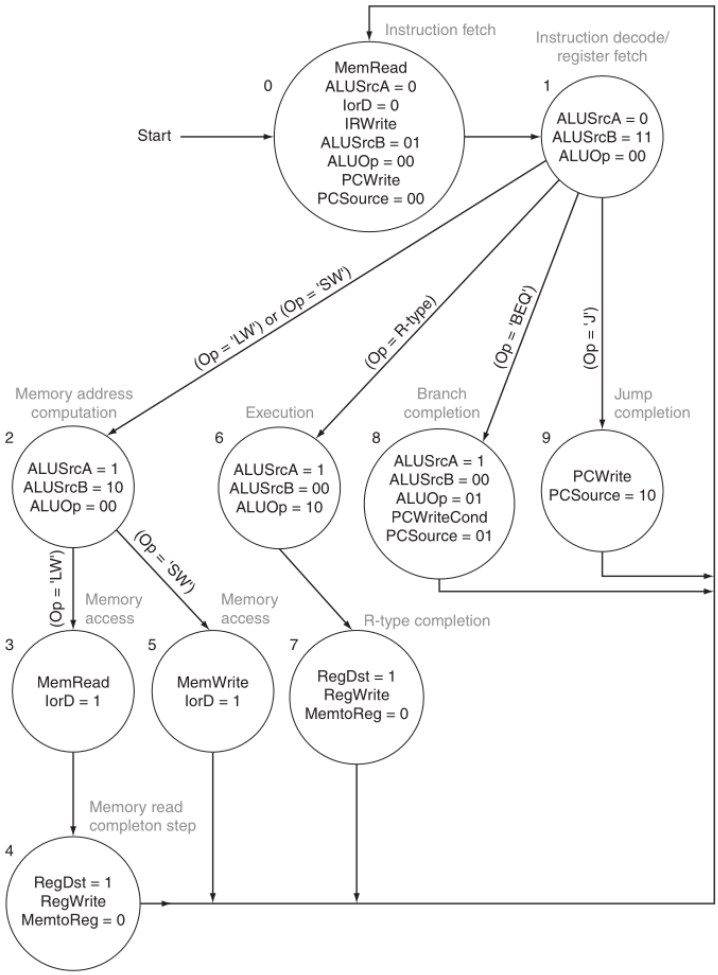
Segnali di controllo utilizzati:

- **`RegWrite`** per poter scrivere nel *Register File*
- **`RegDest`** per indicare il registro da scrivere
- **`MemToReg`** per scrivere il valore dalla memoria

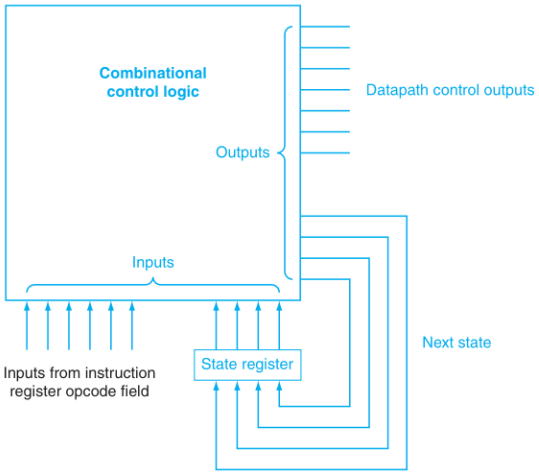
9.2.10 Riassunto

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]),2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

9.3 Control Unit (FSM)



Per implementare questa macchina a stati finiti viene utilizzata un **combinational control logic unit**



10 Eccezioni

Durante l'esecuzione delle istruzioni si possono verificare eventi inattesi:

- **Eccezioni** *(exception)*: evento sincrono generato all'interno del processore e provocato da problemi nell'esecuzione di un'istruzione
- **Interruzione** *(interrupt)*: evento asincrono che giunge dall'esterno del processore, di solito da un'unità di I/O utilizzato per comunicare alla CPU il verificarsi di certi eventi

Il controllo del processore deve gestire gli eventi inattesi e esegue i seguenti passi per gestirli:

- Interruzione dell'esecuzione del programma corrente
- Salvataggio parziale dello stato di esecuzione corrente (per riprendere l'esecuzione del programma corrente se possibile)
- Salto a una routine del sistema operativo per gestire l'eccezione/interruzione *(handler)*
- Esecuzione della routine
- Se possibile, ripristino dello stato di esecuzione del programma e continuazione della sua esecuzione

Per capire l'evento inatteso verificatosi è possibile utilizzare due metodi:

- **Indirizzo fisso** (registro dedicato): il controllo della CPU, prima di saltare alla routine (ad un indirizzo fisso), deve salvare in un registro interno (`Cause`) un identificatore numerico del tipo di eccezione avvenuta
- **Interruzioni vettorizzate**: esistono handler diversi per eccezioni/interruzioni differenti. Vengono salvate in un vettore di indirizzi

> Nel MIPS viene utilizzata la prima soluzione.
>
> La causa dell'eccezione viene salvata in `Cause`, mentre l'indirizzo dell'istruzione corrente che ha causato l'eccezione viene salvato in `Exception Program Counter` (`EPC`).

10.1 Esempi

Consideriamo solo 2 cause di eccezione:

- Istruzione non valida
- Overflow

Passi da eseguire:

1. Individuare l'evento inatteso, la causa dell'eccezione e salvarla in un registro dedicato denominato `Cause`
2. Interrompere l'esecuzione corrente
3. Salvare l'indirizzo dell'istruzione corrente in `EPC` (`EPC <- PC - 4`)
4. Saltare a un gestore dell'eccezione del SO che si trova a un indirizzo fisso per gestire l'eccezione

Osservazioni:

- Il MIPS non salva nessun altro registro oltre al PC
- È compito della routine salvare altre porzioni dello stato corrente del programma se necessario

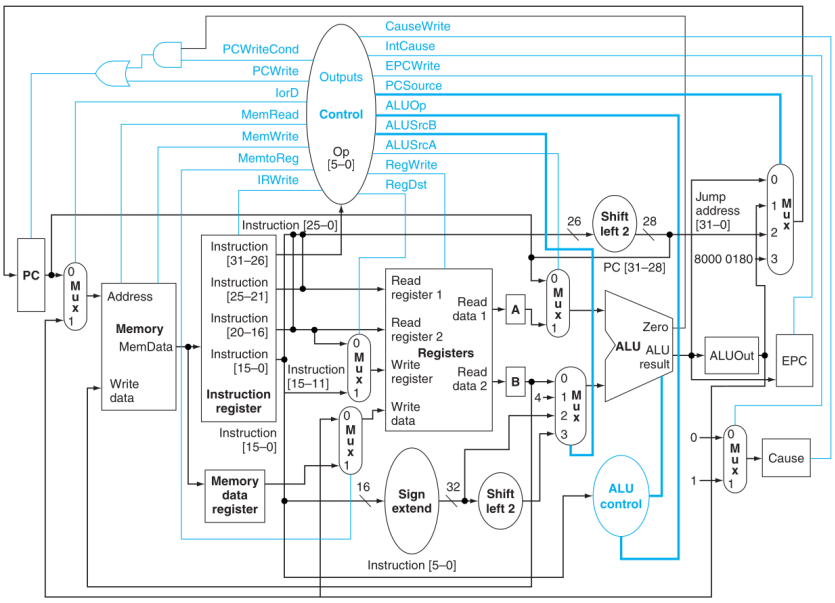
- Esistono CPU dove questo salvataggio viene prima di saltare alla routine

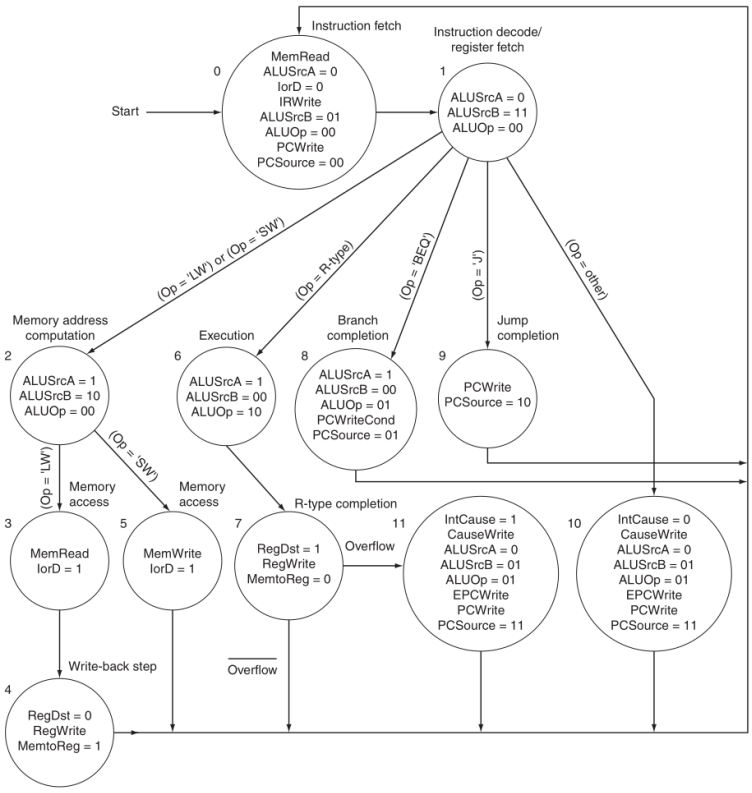
Nel caso di *istruzione non valida*, l'unità di controllo rileva tale eccezione sulla base dell'op code di un'istruzione.

L'*overflow* invece è un segnale che arriva all'unità di controllo dall'ALU.

Per gestirle bisogna aggiungere due nuovi stati alla FSM del datapath. Tali stati devono:

- Salvare in EPC il valore `PC - 4`
- Salvare nel registro `Cause` la causa dell'eccezione (`0` o `1` in questo caso perché consideriamo solo due esempi)
- Salvare in PC l'indirizzo del gestore delle eccezioni





11 Gestione dell'I/O

Esistono vari tipi di bus nei computer moderni. In questo corso vedremo il bus comune (di sistema) che collega la CPU sia con la memoria, sia con le periferiche.

11.1 Bus di sistema

Il **bus di sistema** è composto da:

- **Bus dei dati**: le linee per trasferire dati e istruzioni da e verso altri dispositivi
- **Bus di controllo**: trasporta le informazioni per la definizione delle operazioni da compiere e per la sincronizzazione tra i dispositivi
- **Bus degli indirizzi**: indica gli indirizzi di memoria che la CPU trasmette e che identificano i dati da leggere e scrivere dalla memoria o dalle periferiche

11.2 Periferiche

Le **periferiche** sono collegate alla CPU tramite il bus di sistema e/o **interfacce**.

Le interfacce sono standardizzate e hanno sia una componente hardware (controller) che una componente software (driver). Sono composte da:

- Registri di dati (buffer): rappresentano i dati in input o output in base al tipo di periferica
- Registri di stato: rappresentano lo stato della periferica

Queste periferiche utilizzano una parte di RAM riservata alle loro interfacce *(memory mapped)*. Ad ogni periferica viene assegnato un identificatore unico (indirizzo). I suoi registri sono quindi accessibili come aree di memoria. Queste aree di memoria sono accessibili soltanto dal sistema operativo, dunque se un utente deve leggere o scrivere deve passare dal SO.

11.3 Tecniche di gestione I/O

> - I/O gestito dal programma *(Programmed I/O)*
> - I/O guidato da interrupt *(Interrupt Driven I/O)*
> - Accesso diretto alla memoria *(Direct Memory Access - DMA)*

11.3.1 I/O gestito dal programma

Se l'I/O viene gestito dal programma in esecuzione *(Programmed I/O)*, la CPU si occupa del controllo e del trasferimento dei dati. La periferica ha un ruolo passivo. La CPU predispone il controllore della periferica all'esecuzione dell'I/O, interroga la periferica e aspetta la ricezione dei dati *(busy waiting)*.

Il vantaggio è una risposta molto veloce al ready bit, ma lo svantaggio è che il processore rimane bloccato in attesa della risposta.

Prestazioni:

- Banda passante: alta perché la CPU trasferirà subito il dato e la gestione della periferica richiede poche istruzioni
- Latenza: minima in quanto la CPU noterà subito lo stato della periferica

11.3.2 I/O guidato da interrupt

Se l'I/O viene guidato da interrupt *(interrupt driven I/O)*, la periferica segnala alla CPU di aver bisogno di attenzione tramite un interrupt attivato da un segnale di controllo nel bus. Quando il processore se ne accorge (in una fase di fetch) informa la periferica con un segnale di **interrupt acknowledge**. La CPU dunque interrompe la normale esecuzione del programma ed esegue la procedura di risposta all'interrupt.

Questo permette di poter eseguire altre istruzione mentre si attende il ready bit. Però la CPU deve comunque trasferire i dati quando avviene l'interrupt.

11.3.3 Direct Memory Access

Il **Direct Memory Access** *(DMA)* è un metodo per trasferire dati senza l'utilizzo del processore. La periferica diventa autonoma e gestisce il trasferimento di dati alla memoria. Questo permette di trasferire velocemente grandi quantità di dati.

Necessità di due registri in più per ogni periferica oltre a quelli già presenti:

- Un registro che indichi l'indirizzo di memoria da/dove trasferire i dati
- Un registro che indichi la quantità dei dati da trasferire

Anche questi due registri vengono mappati in memoria. Alla fine del trasferimento la periferica invia un interrupt per segnalare il completamento del trasferimento.

12 Cache

All'interno di un computer ci sono diversi tipi di memoria, disposti in una gerarchia dal più veloce al più lento:

- Registri
- Cache
- RAM
- HDD

> I programmi non vedono questa gerarchia, ma referenziano i dati come se fossero sempre in memoria centrale (ovviamente fino alla RAM, perché gli HDD hanno interfacce particolari).

In questa parte del corso analizzeremo la **cache**.

12.1 Principio di località

Un programma, in un certo istante, accede soltanto a una porzione relativamente piccola del suo spazio di indirizzamento.

Si hanno due tipi di località:

- **Temporale**: la tendenza a far riferimento allo stesso elemento dopo poco tempo
- **Spaziale**: la tendenza a far riferimento a elementi vicini dopo poco tempo

12.2 Definizioni

- **Blocco**/**linea**: la più piccola quantità di informazione che può essere presente/assente in una gerarchia di memoria
- **Hit**: l'informazione richiesta dal processore si trova in uno dei blocchi di memoria

- **Miss**: il dato non è presente nel blocco ed occorre accedere al livello sottostante
- **Hit rate**: frazione degli accessi alla memoria nei quali l'informazione richiesta è stata trovata nel livello di memoria
- **Miss rate**: frazione degli accessi alla memoria nei quali l'informazione richiesta non è stata trovata nel livello di memoria $(1 - \textit{Hit rate})$
- **Tempo di hit**: tempo di accesso al livello della memoria (comprende anche il tempo necessario a stabilire se il tempo di accesso si risolva in un hit o miss)
- **Tempo di miss**: tempo necessario a sostituire un blocco del livello superiore con un nuovo blocco dal livello inferiore della gerarchia, e trasferire i dati di questo blocco al processore

12.3 Tipologie di cache

Esistono tre tipologie di cache:

- **Direct mapped**: a ciascun blocco della memoria corrisponde un a specifica locazione nella cache
- **Fully-associative**
  - Ogni blocco può essere collocato in qualsiasi locazione della cache
  - Per ricercare un blocco nella cache è necessario cercarlo in tutte le linee della cache
  - La ricerca sequenziale è troppo lenta e la ricerca in parallelo è una soluzione molto costosa
- **Set-associative**
  - Soluzione intermedia tra direct mapped e fully associative
  - Ciascun blocco della memoria ha a disposizione un numero fisso (≥ 2) di locazioni in cache

12.3.1 Direct mapping

La **Direct Mapped Cache** associa una sola locazione della cache a ogni word della memoria definendo una corrispondenza tra l'indirizzo in memoria della parola e la locazione nella cache.

L'indirizzo è così composto:

- **Tag**: contiene le informazioni necessarie a verificare se una parola della cache corrisponde o meno alla parola cercata
- **Indice**: utilizzato per selezionare il blocco della cache
- **Offset**: bit necessari per selezionare il byte richiesto nella parole

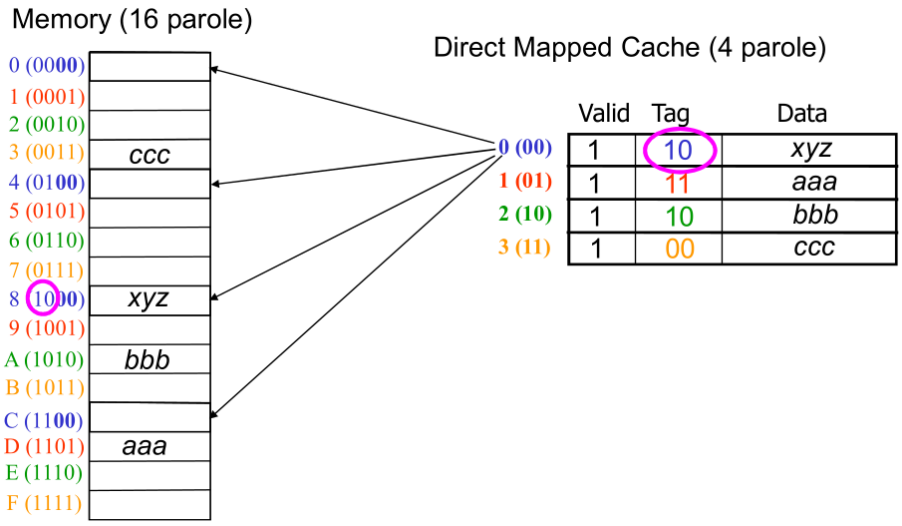
Dato che il numero di blocchi nella cache è una potenza di 2, la posizione corrispondente della parola in cache è data dai $\log_2(\text{n. elementi nella cache})$ bits meno significativi dell'indirizzo in memoria principale.

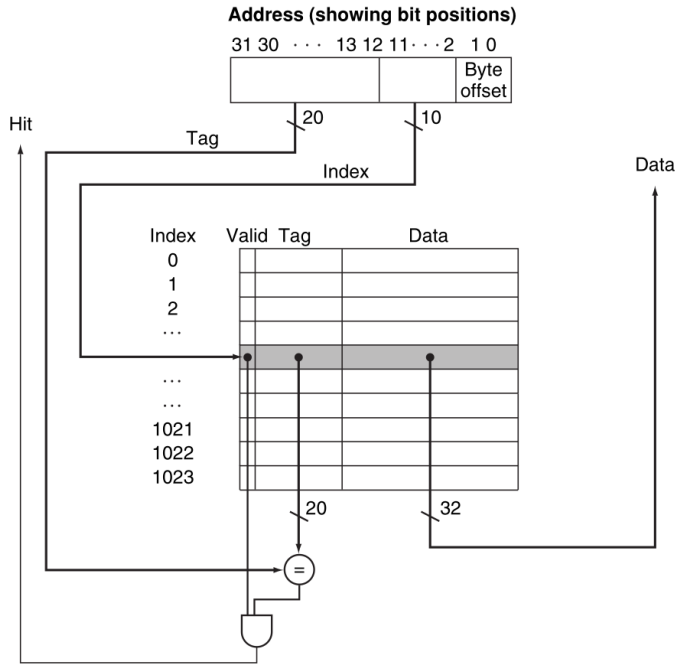
Esempio:

- Numero di elementi nella cache: 8
- Bit per indirizzare una locazione della cache: $\log_2 8 = 3$
- Indirizzo della word in memoria = `0111 0101 0010 0100`
- Posizione in cache: `100` (4)

In una *DMC* ogni linea di cache include:

- Il **bit di validità** che indica se i dati nella linea di cache sono validi
- Il **tag** che consente di individuare in modo univoco il blocco in memoria che è stato *mappato* nella linea di cache
- Il **blocco di dati** vero e proprio





In caso di hit, il processore continua l'esecuzione con l'accesso al dato dalla cache dati e accesso all'istruzione dalla cache istruzioni.

In caso di miss:

1. Si ha uno stallo del processore (come nel pipelining) in attesa di ricevere l'elemento dalla memoria
2. Invio dell'indirizzo al controller della memoria (una MMU, simile ad un DMAC)
3. Reperimento dell'elemento dalla memoria

4. Caricamento dell'elemento in cache
5. Ripresa della normale esecuzione del programma

Esempio: si consideri la seguente situazione

- Indirizzo su 32 bit
- Cache a mappatura diretta
- La dimensione della cache è di 2^n blocchi, dove n bit vengono usati per l'indice
- La dimensione del blocco della cache è di 2^m word, ossia 2^{m+2} byte, per cui m bit vengono usati per individuare una word all'interno di un blocco, mentre 2 bit per individuare un byte all'interno di una word

La dimensione del campo **tag** è

$$\dim(\text{tag}) = 32 - (n + m + 2)$$

Il numero totale di bit contenuti in una cache a mappatura diretta è

$$\begin{aligned} \text{n bit} &= 2^n \cdot \big(\dim(\text{blocco}) + \dim(\text{tag}) + \text{n bit validità} \big) \\ &= 2^n \cdot \big(2^m \cdot 23 + (32 - (n + m + 2)) + 1 \big) \end{aligned}$$

Esempio: consideriamo ora una cache con 64 blocchi di 16 byte ciascuno. L'indirizzo 1200 (in byte) può essere ricavato come:

$$\begin{aligned} &(\text{indirizzo del blocco}) \mod (\text{n blocchi}) \\ &\frac{1200}{16} = 75 \mod 64 \end{aligned}$$

e l'indirizzo del blocco si ricava come

$$\text{indirizzo del blocco} = \frac{\text{indirizzo del dato in byte}}{\text{byte per blocco}}$$

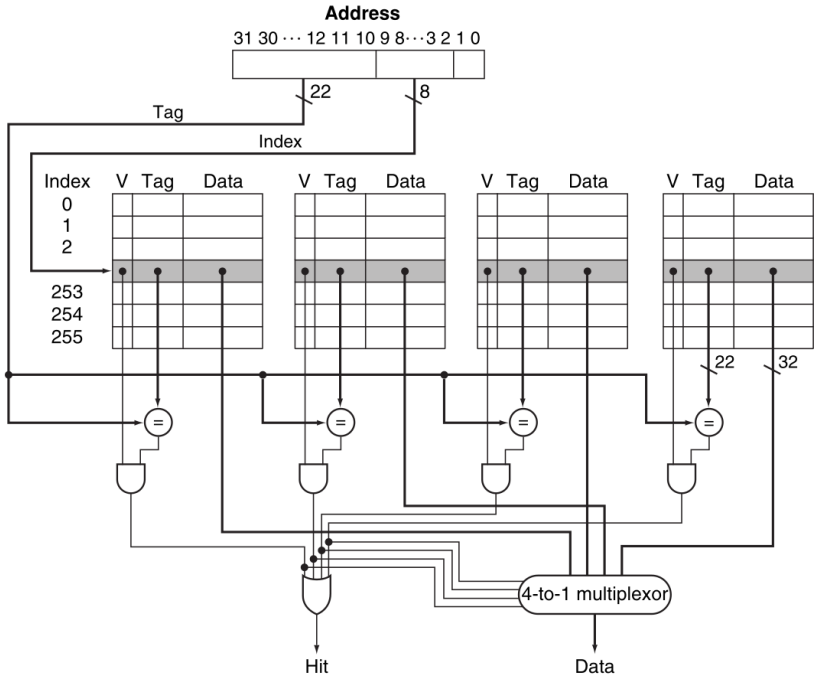
12.3.1.1 Scelta della dimensione del blocco

Un'ampia dimensione per il blocco permette di sfruttare la località spaziale, ma:

- Blocchi di grossa dimensione comportano maggiori **miss penalty** *(differenza tra il tempo di accesso al livello inferiore e il tempo di accesso alla cache)* ed è necessario più tempo per trasferire il blocco
- Se la dimensione del blocco è troppo grossa rispetto alla dimensione della cache, il **miss rate** aumenta (il numero di blocchi nella cache è insufficiente)

12.3.2 Cache set-associative

Nelle **cache set-associative a n vie** i blocchi sono raggruppati in **set** (ogni set raggruppa n blocchi). Ogni indirizzo di memoria corrisponde ad un unico set della cache (accesso diretto tramite indice) è può essere ospitato in un blocco qualunque appartenente a quel set. Stabilito il set, per determinare se un certo indirizzo è presente in un blocco del set è necessario confrontare in parallelo i tag di tutti i blocchi.

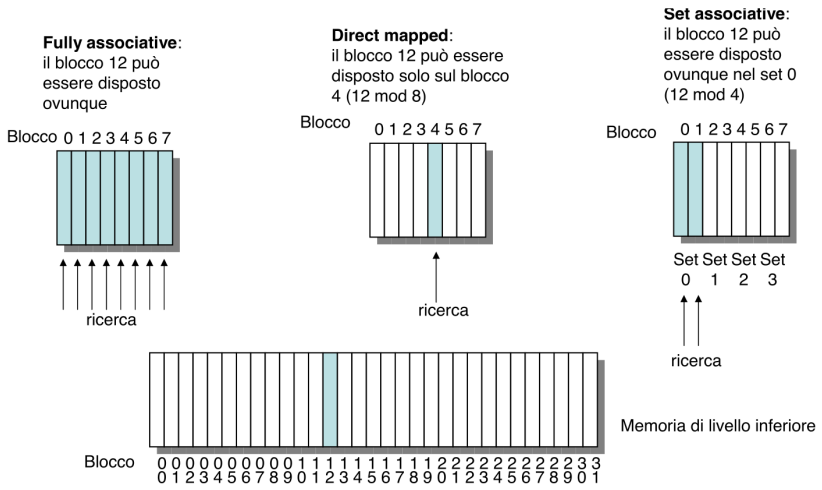


Svantaggi della cache set-associative: le cache -way set-associative a confronto

con cache ad accesso diretto:

- Utilizzano n comparatori invece di 1
- Hanno un ulteriore ritardo fornito dal multiplexer
- Il blocco è disponibile dopo la decisione hit/miss e la selezione del set (in una cache ad accesso diretto, il blocco è disponibile prima della decisione hit/miss)

Nell'incremento dell'associatività si ha il principale vantaggio nella diminuzione del miss rate, ma ad un maggior costo implementativo. La scelta tra cache ad indirizzamento diretto, set-associative e full-associative dipende dal costo dell'associatività rispetto alla riduzione del miss rate.



12.4 Gestione della cache

Per gestire una cache bisogna prendere 4 decisioni:

1. Dove posizionare un blocco *(tecnica di indirizzamento)*
2. Come reperire un blocco *(tecnica di indirizzamento)*
3. Quale blocco sostituire in corrispondenza di un miss *(algoritmo di sostituzione)*
4. Come gestire un'operazione di scrittura *(strategia di aggiornamento)*

Le prime due sono già state viste con le varie tipologie di cache.

12.4.1 Algoritmi per la sostituzione di blocchi

Nella cache ad accesso diretto se il blocco di memoria è mappato in una linea di cache già occupata (**conflict miss**), si elimina il contenuto precedente della linea e si rimpiazza con il nuovo blocco.

In caso di **(capacity) miss** nelle cache fully-associative ogni blocco è un potenziale candidato per la sostituzione. Invece nelle cache set-associative a n vie, bisogna scegliere tra gli n blocchi del set.

Ci sono diverse politiche di sostituzione:

- **Random**: scelta casuale
- **Least Recently Used (LRU)**: sfruttando la località temporale, il blocco sostituito è quello che non si utilizza da più tempo (ad ogni blocco si associa un contatore all'indietro, che viene portato al valore massimo in caso di accesso e decrementato di 1 ogni volta che si accede ad un altro blocco)
- **First-in First-out (FIFO)**: si approssima la strategia LRU selezionando il blocco più vecchio anziché quello non usato da più tempo

12.4.2 Gestione dei miss in lettura

Se un blocco non è presente nella cache bisogna mettere in stallo l'intera CPU. In generale al verificarsi di un miss nella cache delle istruzione sono necessari i seguenti passi:

1. Inviare PC (uscita dall'ALU) alla memoria
2. Lettura dalla memoria
3. Scrittura nella cache (dato, tag e bit di validità)
4. Riavviare l'esecuzione dell'istruzione che ha causato il miss

12.4.3 Accesso in scrittura

Scrivere un dato nella cache significa creare un'incoerenza, se non si aggiornano i livelli inferiori della gerarchia di memorie. Tale aggiornamento richiede lo stallo della CPU. Si hanno tre tecniche risolutive:

- **Write-through**
- **Write-back**
- **Write-through** con un write buffer

12.4.3.1 Write-through

Con la tecnica del **write-through** i dati sono scritti nel blocco della cache e nel blocco del livello inferiore allo stesso momento.

Vantaggi:

- È la soluzione più semplice da implementare
- Si mantiene la coerenza delle informazioni nella gerarchia di memorie

Svantaggi:

- Le operazioni di scrittura vengono effettuate alla velocità della memoria di livello inferiore (dunque diminuiscono le prestazioni)
- Aumenta il traffico sul bus di sistema

12.4.3.2 Write-back

Con la tecnica del **write-back** (o copy-back) i dati sono scritti solo nel blocco della cache. Il blocco modificato viene scritto nel livello inferiore della gerarchia solo quando deve essere sostituito. Questo implica che subito dopo la scrittura, cache e memoria di livello inferiore sono inconsistenti, e viene mantenuto un **dirty bit**.

Vantaggi:

- Le scritture avvengono alla velocità della cache
- Scritture successive sullo stesso blocco alterano solo la cache

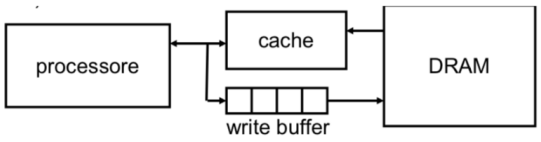
Svantaggi:

- Ogni sostituzione del blocco può provocare un trasferimento in memoria

12.4.3.3 Write-through con write buffer

Posizionando un **write buffer** tra la cache e la memoria di livello inferiore il processore scrive i dati sia nella cache che nel buffer e il controller della memoria scrive il contenuto del buffer in memoria.

Il write buffer è gestito in modalità FIFO con un numero tipico di elementi nel buffer uguale a 4. Il buffer è efficiente se la frequenza di scrittura $< \frac{1}{\text{DRAM write cycle}}$. Altrimenti il buffer può andare in saturazione ed il processore deve aspettare che le scritture giungano a completamento (**write stall**).



12.4.4 Write miss

Un secondo scenario di inconsistenza potrebbe accadere se cerchiamo di scrivere in un indirizzo che non è ancora contenuto nella cache (**write miss**).

Si hanno due possibili soluzioni:

- **Write allocate**: il blocco viene caricato in cache e si effettua la scrittura
- **No-write allocate**: il blocco viene scritto direttamente nella memoria di livello inferiore, senza essere trasferito in cache

Tipicamente troviamo le seguenti combinazioni:

- Write back con write allocate
- Write through con no-write allocate

12.5 Osservazioni

Una cache con blocchi di dimensioni maggiori sfruttano maggiormente la località spaziale diminuendo la frequenza di miss. La frequenza di miss torna a crescere se la dimensione dei blocchi diventa troppo grande rispetto alla dimensione della cache.

Quindi i blocchi vengono scaricati dalla cache prima ancora che molti dati in essi contenuti siano stati utilizzati. Quindi la località spaziale tra le parole di un blocco diminuisce e il miglioramento legato alla frequenza di miss si riduce.

Inoltre, cresce anche il costo di una miss: la penalità di una miss è determinata dal tempo necessario a prelevare un blocco dal livello sottostante e a scriverlo nella cache. Il tempo per prelevare un blocco è dato dalla somma tra la latenza per ottenere la prima parola del blocco e il tempo di trasferimento del resto del blocco.

