Appunti di Architettura

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

Federico Zotti

26 Mar 2024

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

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

e l'intervallo rappresentabile dalla parte decimale è

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

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:

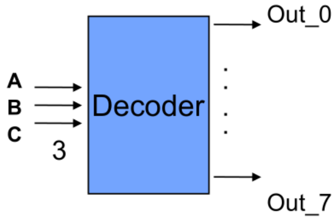
- NAND
- NOR
- XOR

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

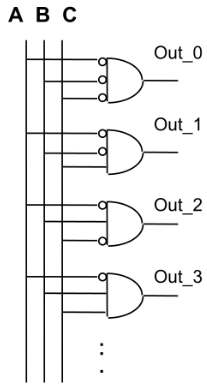
Le porte NOR e NAND che svolgono la funzione di inverter sono definite **universali**:

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

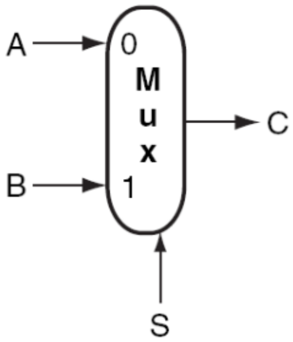


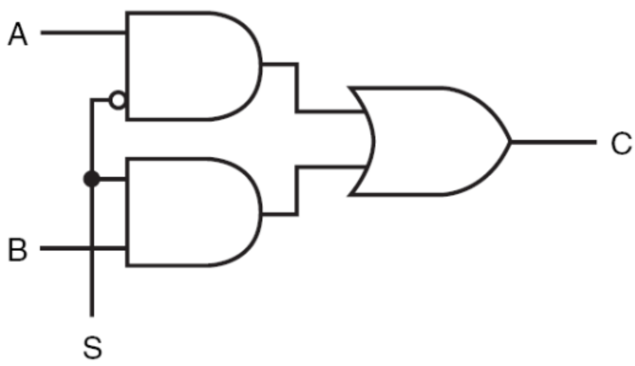
4.3 Multiplexer

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.

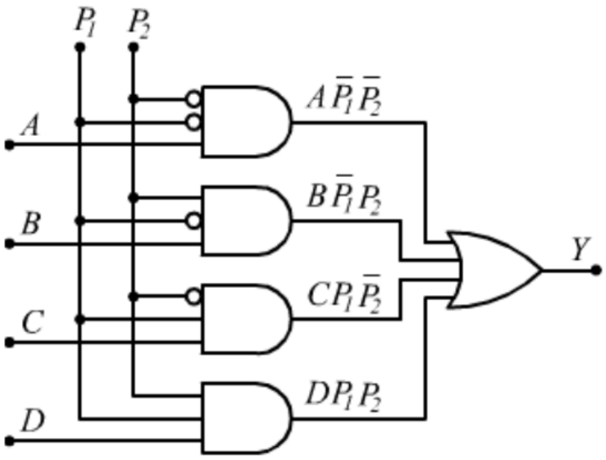
Esempio con selettore ad un bit

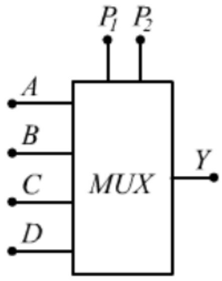
$$C = (A \cdot \bar{S}) + (B \cdot S)$$





Esempio con selettore a due bit





P_1	P_2	Y
0	0	A
0	1	B
1	0	C
1	1	D

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

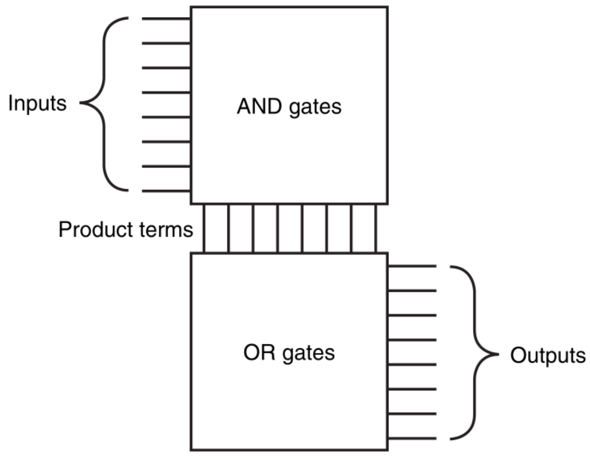
Utilizzando le porte logiche AND, OR e NOT è possibile implementare qualunque funzione logica più complessa. Dunque possiamo creare logiche a due livelli:

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

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

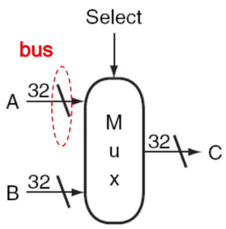
$$D = (\bar{A} \cdot \bar{B} \cdot C) + (\bar{A} \cdot B \cdot \bar{C}) + (A \cdot \bar{B} \cdot \bar{C}) + (A \cdot B \cdot C)$$

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

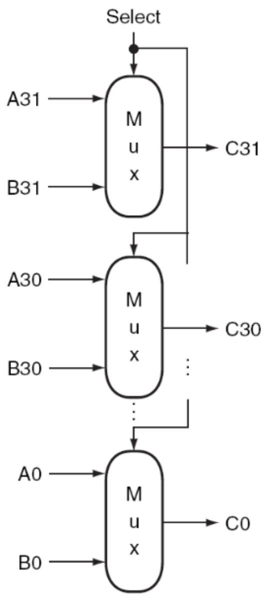


4.5 Array di elementi logici

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

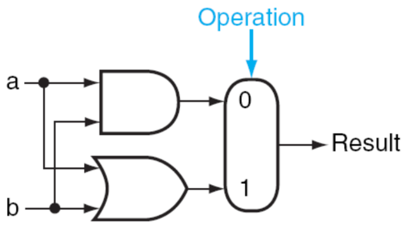


4.6 ALU

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).

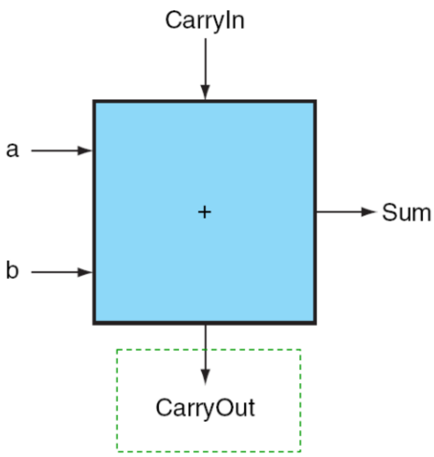
4.6.1 Operazioni logiche

ALU ad un bit che implementa AND e OR:



4.6.2 Addizione

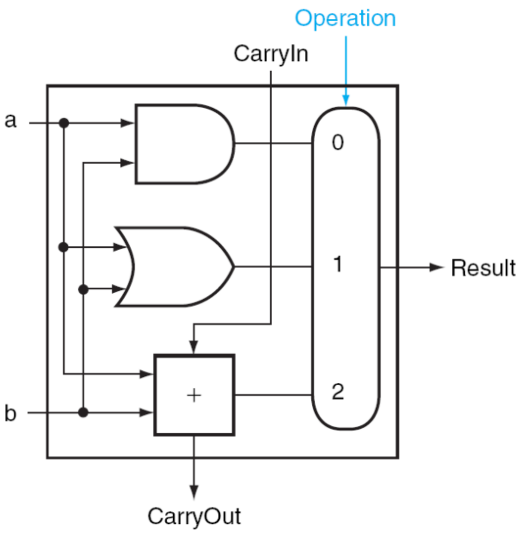
Per l'addizione è sufficiente implementare una PLA con input i due bit da sommare con il carry in ingresso e in output la somma con il carry in uscita.



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

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

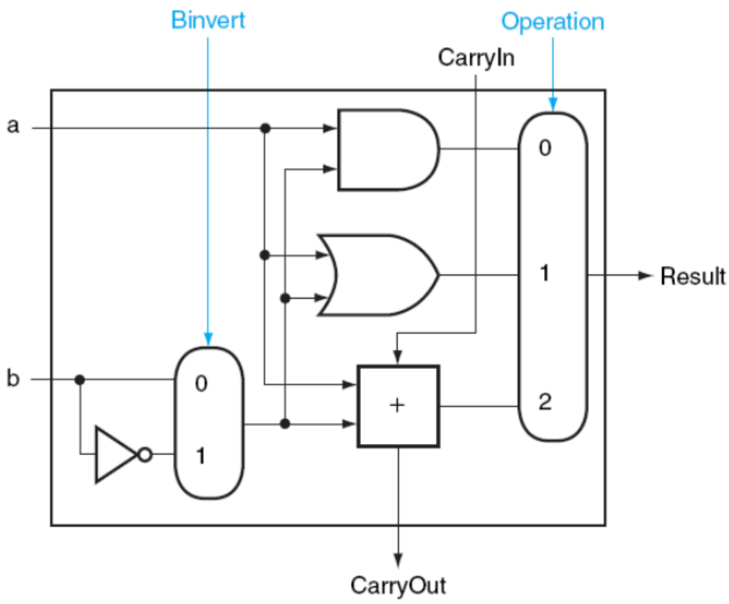
Questo blocco si può aggiungere all'ALU:



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.

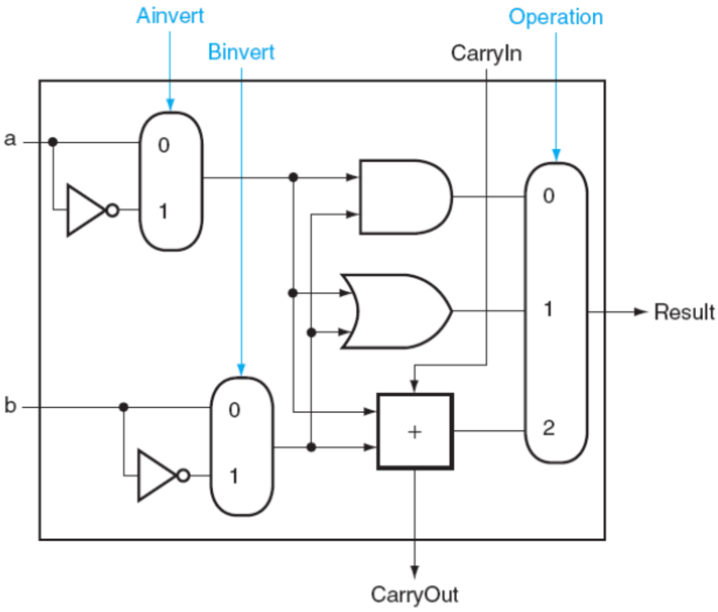
4.6.4 Porta NOR

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

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

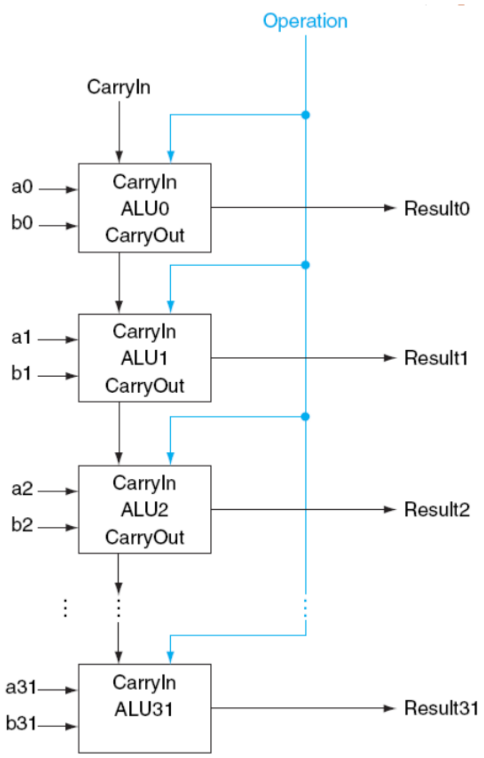
$$(\overline{a \cdot b}) = \bar{a} + \bar{b}$$

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)

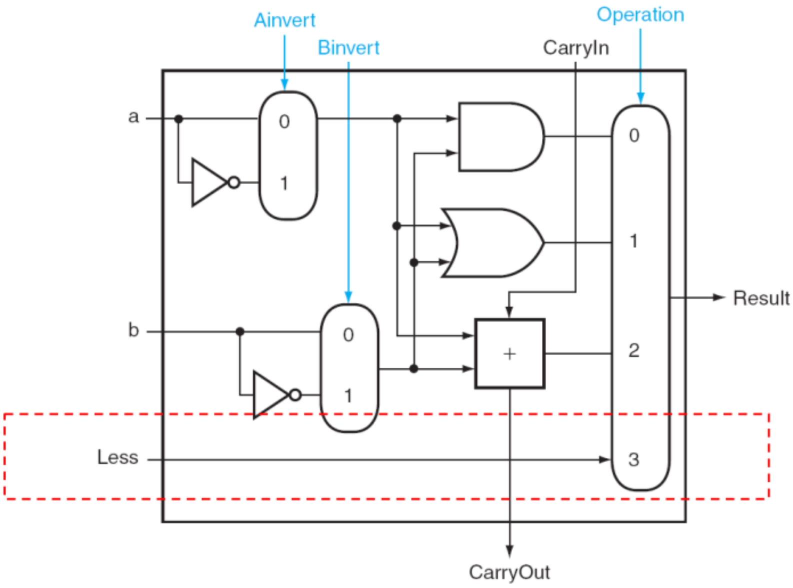
Questa operazione ritorna 1 se $a < b$, altrimenti 0. Per eseguire questa istruzione si devono poter azzerare tutti i bit dal bit 1 al bit 32 ed assegnare al bit 0 il valore del risultato.

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

Per poter effettuare il confronto si sottrae a e b.

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

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

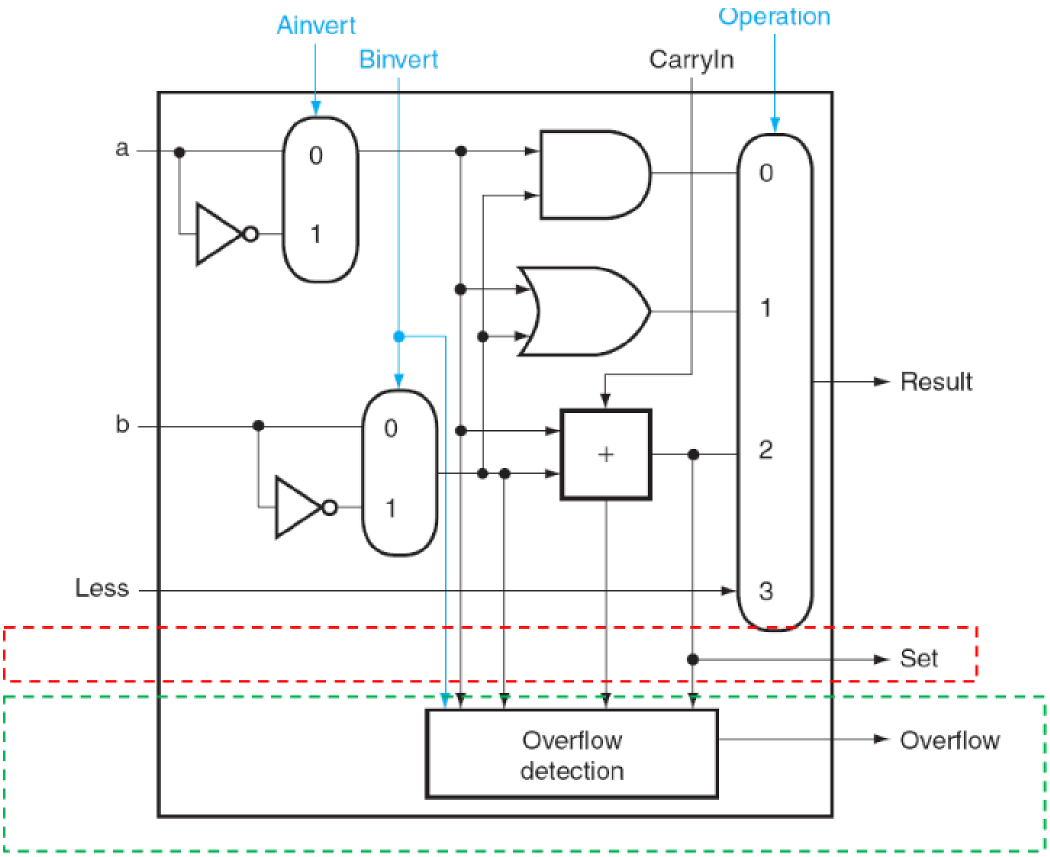


Per le ALU dal bit 1 *(il secondo)* al bit 31 *(l'ultimo)* i collegamenti sono:

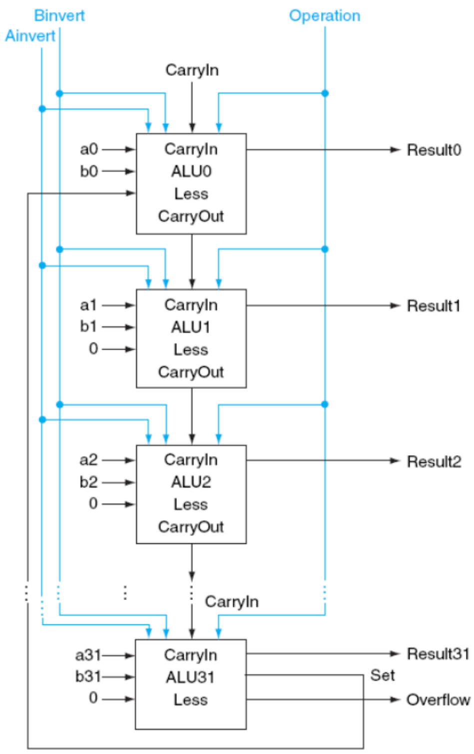
- `CarryIn`$_i$ = `CarryOut`$_{i-1}$
- `Less` $= 0$

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: $\bar{a} \cdot b \cdot \text{Result} + a \cdot \bar{b} \cdot \overline{\text{Result}}$



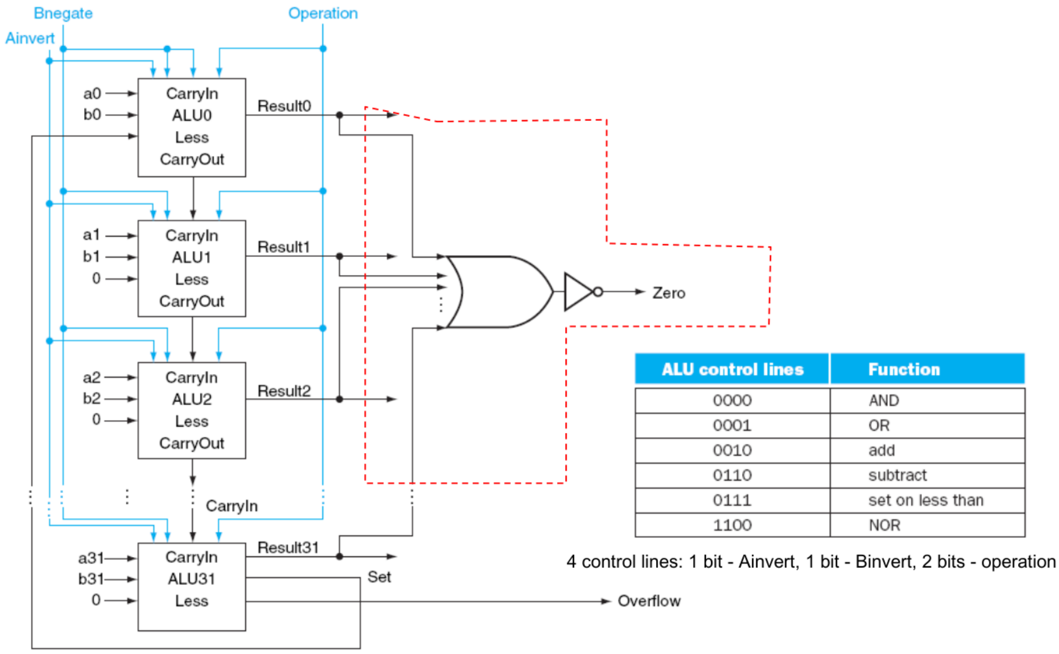
Tutto questo diventa



- 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

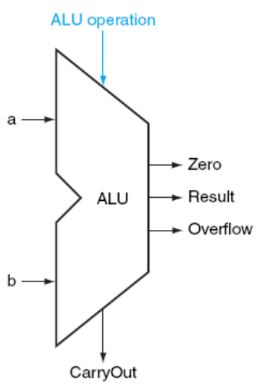
4.6.7 Operazione Branch-on-equal (BEQ)

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

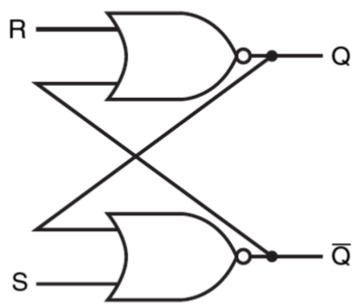
Il segnale di **clock** è fondamentale per le reti sequenziali che sono caratterizzate da uno **stato**. Il clock è definito come un segnale (onda quadra) con un periodo predeterminato e costante. Il periodo di clock deve essere abbastanza grande da assicurare la stabilità degli output di un circuito. 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

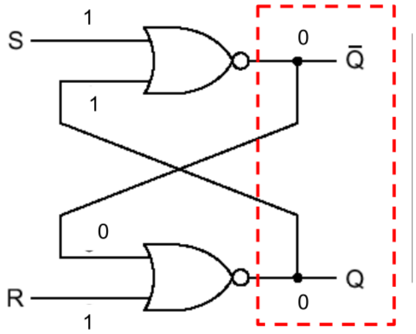
5.2 S-R Latch

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.

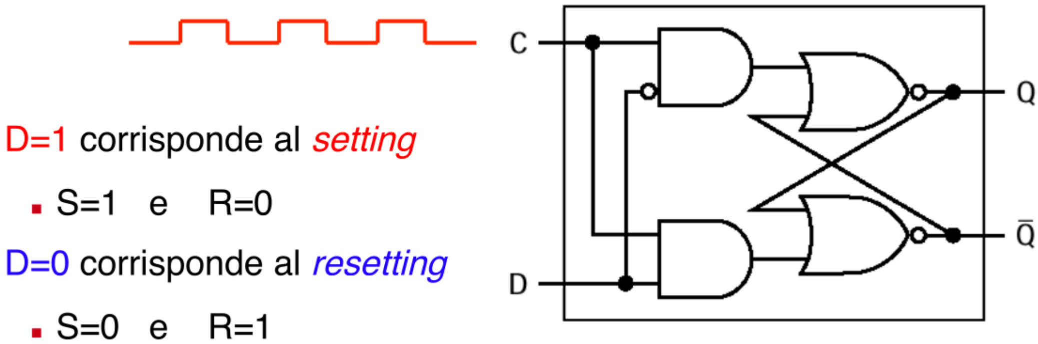


Input		Stato Interno	Output	
S	R	Old Q	Q	$\overline{Q}$
0	0	0	0	1
0	0	1	1	0
0	1	1/0	0	1
1	0	1/0	1	0
1	1	1/0	0	0

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

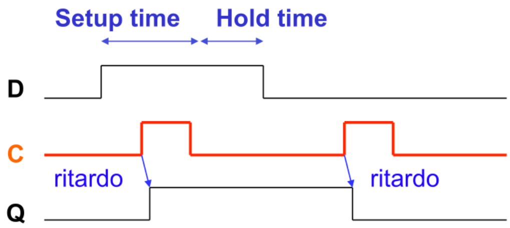
Il **D Latch** è un latch sincronizzato da un clock. Questo clock garantisce che il latch cambi stato solo in certi momenti.



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:

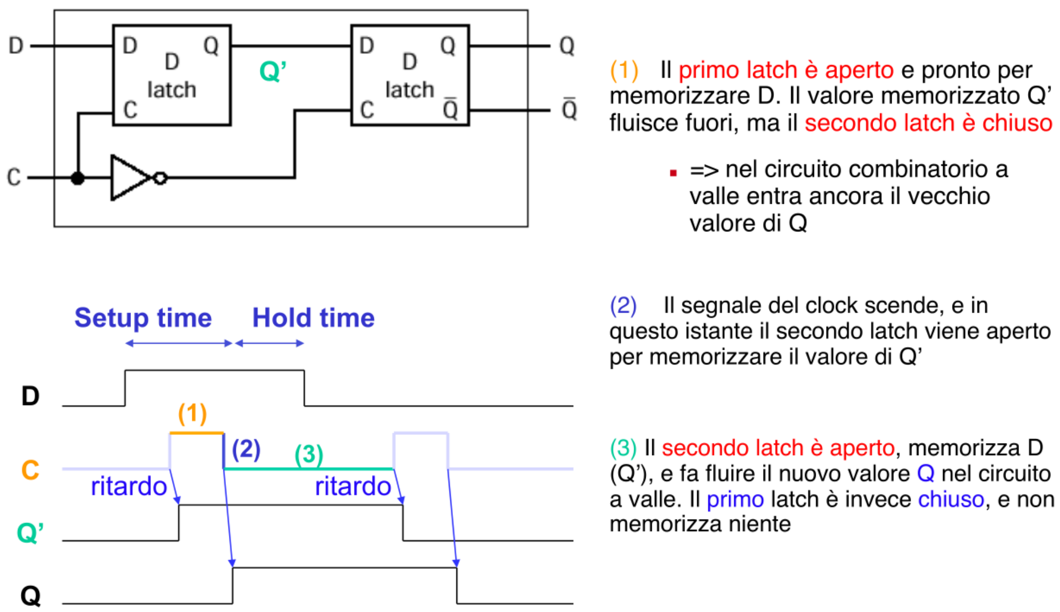
- Essere già stabile quando il clock diventa asserted
- Rimanere stabile per tutta la durata del livello alto del clock *(setup time)*
- Rimanere stabile per un altro periodo di tempo per evitare malfunzionamenti *(hold time)*



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.

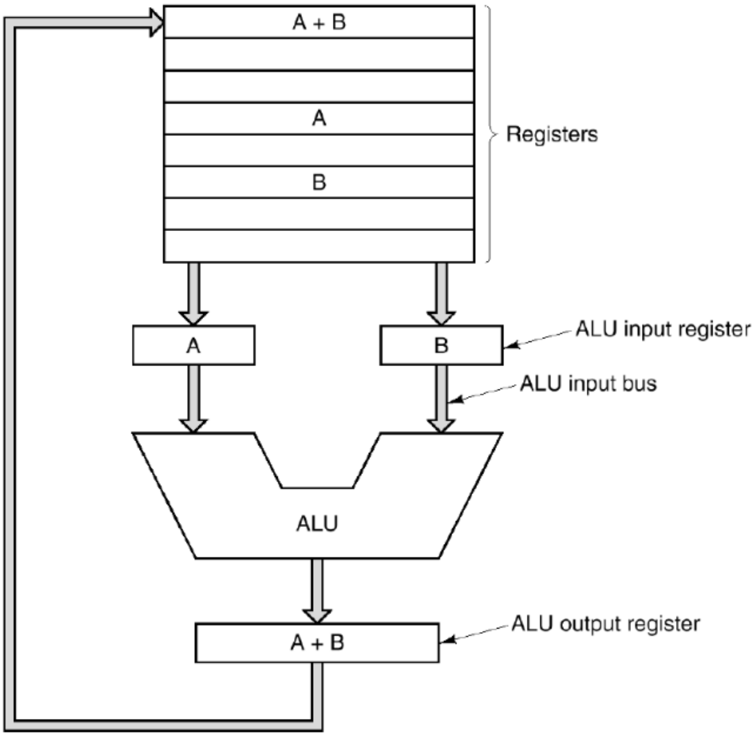
Il D Flip-Flop è utilizzabile come input e output durante lo stesso ciclo di clock. Viene realizzato ponendo in serie due D Latch: il primo viene detto **master** e il secondo **slave**.



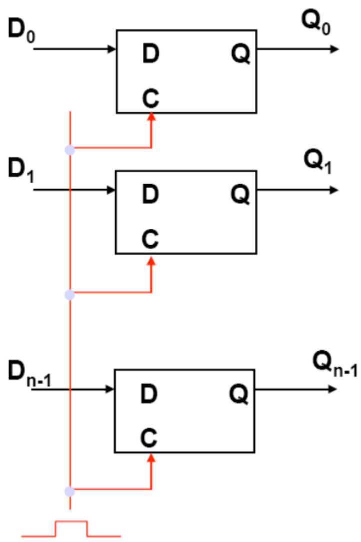
Il segnale D deve essere attivo per un periodo abbastanza lungo (setup time + hold time).

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 1º registro da leggere

Read Reg2 # (5 bit)

- numero del 2º registro da leggere

Read data 1 (32 bit)

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

Read data 2 (32 bit)

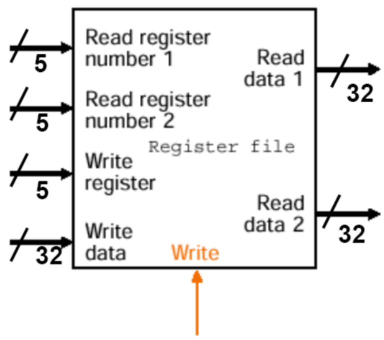
- valore del 2º registro, letto sulla base di Read Reg2 #

Write Reg # (5 bit)

- numero del registro da scrivere

Write data (32 bit)

- valore da scrivere nel registro Write Reg #

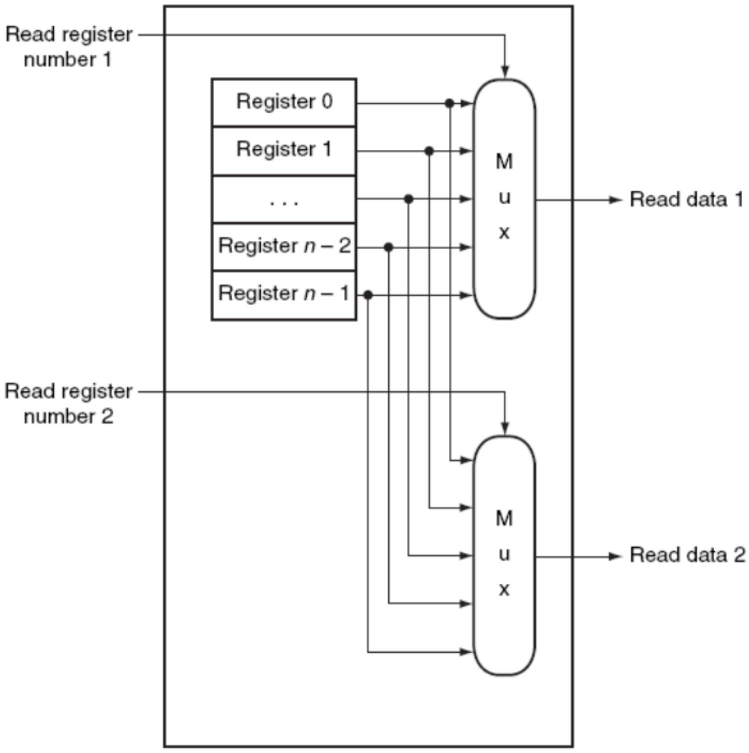


Write

- 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.

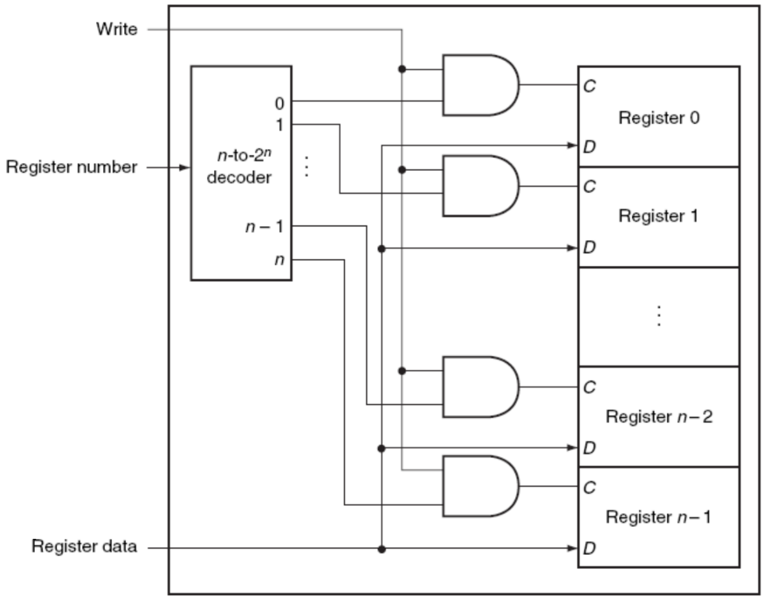


6.2 Scrittura nel register file

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.

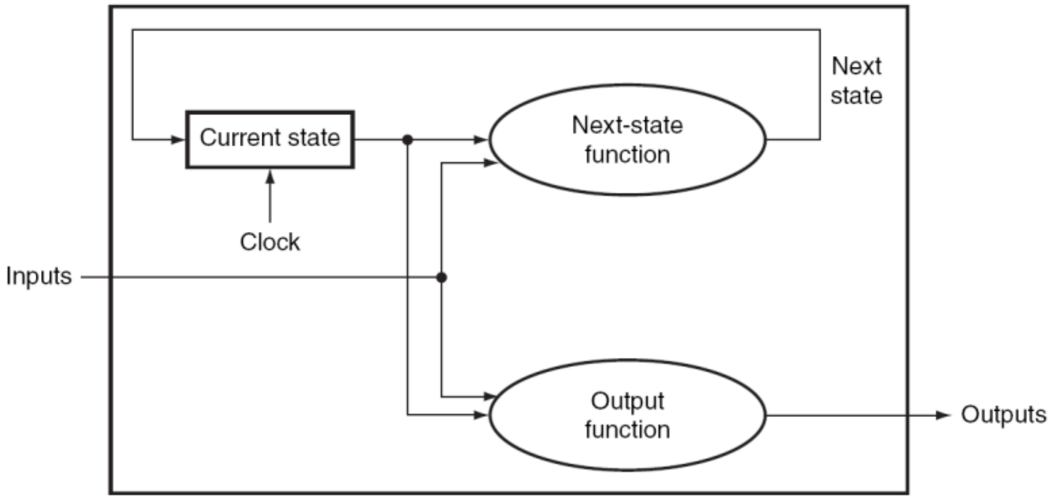


7 Macchine a stati finiti

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`

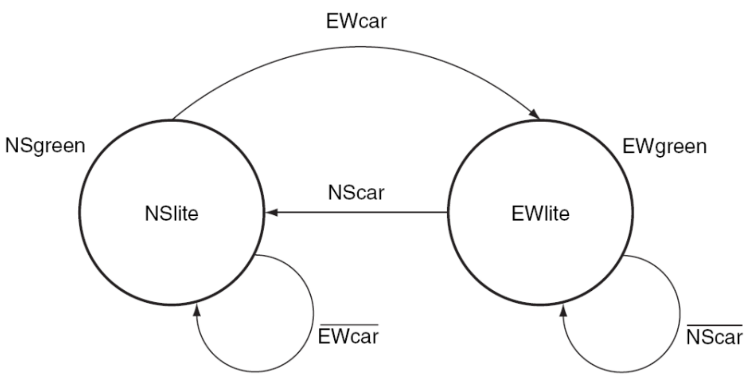
Possibili stati:

- `NSgreen`
- `EWgreen`

Lo stato cambia quando arrivano macchine al semaforo.

	Inputs		Next state
	NScar	EWcar	
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

```
┌──────┬─────┬─────┬─────┬─────┬──────┐│

  op   │  rs │  rt │  rd │shift│funct ││
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
          │└─────────┴─────────┴─────────┴───────────────────────┘
```