

ARCHITETTURE DEI CALCOLATORI

In informatica una **macchina** è una entità in grado di eseguire istruzioni (fisiche o virtuali) appartenenti ad un linguaggio, una **architettura** descrive le relazioni tra le parti di un elaboratore elettronico.

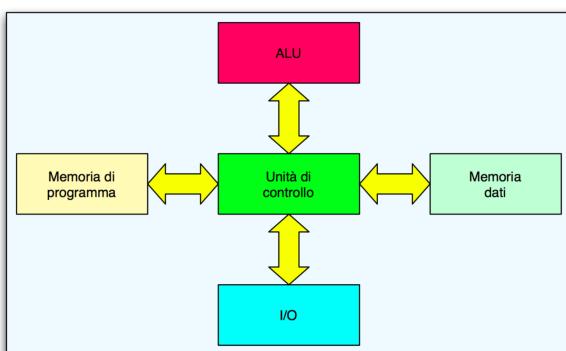
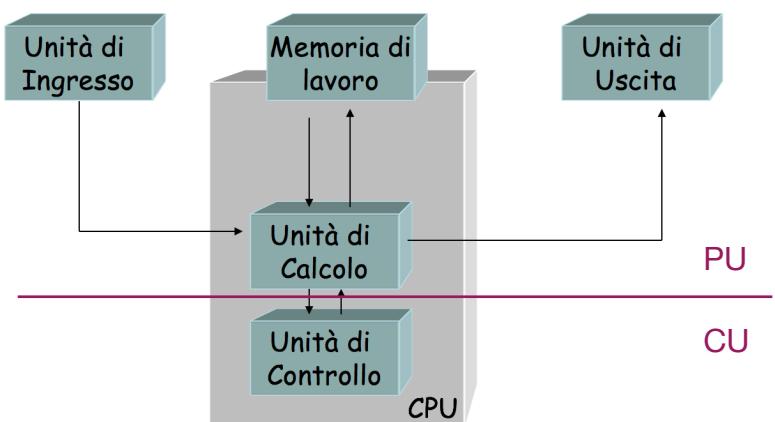
Il **firmware** consiste nel software direttamente connesso al componente e permette di interpretare il linguaggio macchina.

Architettura di Von Neuman

- Nella memoria di lavoro risiedono sia dati che programmi.
- Parte della memoria consiste nei registri contenuti nella CPU.

PU => processing unit (SCA)

CU => control unit (SCO)



Architettura Harvard

Evoluzione dell'architettura di Von Neuman, utilizza blocchi di memoria di versi per dati e programmi.

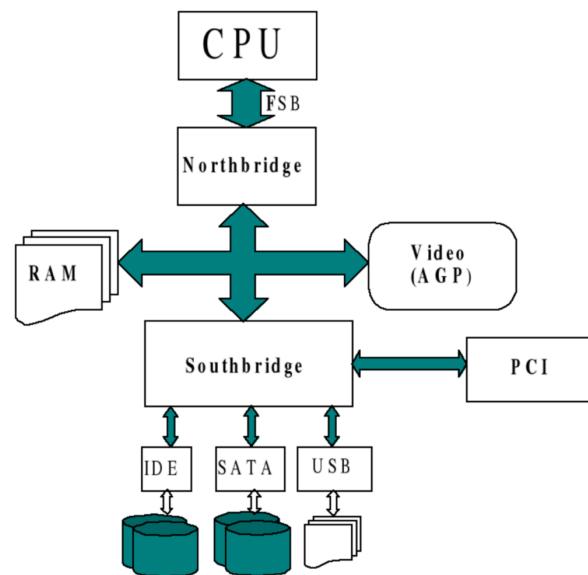
OSS permette di limitare il problema di virus avendo che la memoria dei programmi non può auto sovrascriversi.

Architettura IBM

Architettura reale dei calcolatori moderni, tiene conto delle differenti velocità dei componenti.

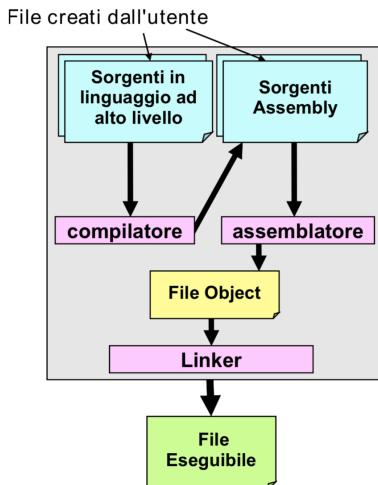
Definizioni preliminari

- **Asincrono** => dispositivo indipendente lavora senza sincronizzazione con altri
- **Sincrono** => dispositivo che lavora seguendo una coordinazione temporale con altri dispositivi



Generazione dei programmi

- I. **Compilazione** => traduzione linguaggio di alto livello in codice Assembly
- II. **Assemblaggio** => generazione di uno o più file contenenti le rappresentazioni binarie delle istruzioni
- III. **Collegamento** => traduzione di indirizzi, collegamento funzioni, ...



$$a = b + c$$

```

movw b, %ax
movw c, %bx
addw %ax, %bx
movw %bx, a

```

```

000101..0101001
1011101..010100
01011..11101010
010..1110101010

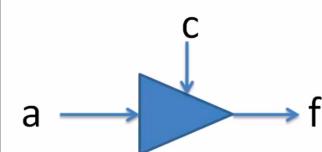
```

Algebra di Boole

Utilizzata per descrivere i comportamenti delle porte logiche¹ che compongono il circuito:

AND Gate		OR Gate																															
Boolean Expression	$Y = A \cdot B$	Boolean Expression	$Y = A + B$																														
$Y = A \cdot B$	<table border="1"> <tr><td>A</td><td>B</td><td>Y</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	A	B	Y	0	0	0	0	1	0	1	0	0	1	1	1	Boolean Expression	<table border="1"> <tr><td>A</td><td>B</td><td>Y</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	1
A	B	Y																															
0	0	0																															
0	1	0																															
1	0	0																															
1	1	1																															
A	B	Y																															
0	0	0																															
0	1	1																															
1	0	1																															
1	1	1																															
NOT Gate		NAND Gate																															
Boolean Expression	$Y = A'$	Boolean Expression	$Y = (A \cdot B)' = A' + B'$																														
$Y = A'$	<table border="1"> <tr><td>A</td><td>Y</td></tr> <tr><td>0</td><td>1</td></tr> <tr><td>1</td><td>0</td></tr> </table>	A	Y	0	1	1	0	Boolean Expression	<table border="1"> <tr><td>A</td><td>B</td><td>Y</td></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	A	B	Y	0	0	1	0	1	1	1	0	1	1	1	0									
A	Y																																
0	1																																
1	0																																
A	B	Y																															
0	0	1																															
0	1	1																															
1	0	1																															
1	1	0																															
NOR Gate		XOR Gate																															
Boolean Expression	$Y = (A + B)'$	Boolean Expression	$Y = A \oplus B$																														
$Y = (A + B)'$	<table border="1"> <tr><td>A</td><td>B</td><td>Y</td></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	0	Boolean Expression	<table border="1"> <tr><td>A</td><td>B</td><td>Y</td></tr> <tr><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>1</td></tr> <tr><td>1</td><td>0</td><td>1</td></tr> <tr><td>1</td><td>1</td><td>0</td></tr> </table>	A	B	Y	0	0	0	0	1	1	1	0	1	1	1	0
A	B	Y																															
0	0	1																															
0	1	0																															
1	0	0																															
1	1	0																															
A	B	Y																															
0	0	0																															
0	1	1																															
1	0	1																															
1	1	0																															
XNOR Gate																																	
Boolean Expression	$Y = A \odot B$																																
$Y = A \odot B$	<table border="1"> <tr><td>A</td><td>B</td><td>Y</td></tr> <tr><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>0</td></tr> <tr><td>1</td><td>1</td><td>1</td></tr> </table>	A	B	Y	0	0	1	0	1	0	1	0	0	1	1	1																	
A	B	Y																															
0	0	1																															
0	1	0																															
1	0	0																															
1	1	1																															

THREE-STATE BUFFER



Aggiunge un terzo stato chiamato
“**stato di alta impedenza elettrica**”
che permette di scollegare l’uscita
dal resto del circuito.

ATTENZIONE non è una porta logica

¹ porte logiche => circuiti digitali di base con 1 uscita ed 1 o più ingressi

Reti combinatorie

Una rete combinatoria (o logica) è un **circuito elettronico digitale** capace di realizzare una o più funzioni di commutazione.

Ogni funzione può essere espressa come **somma canonica**, ovvero è possibile realizzarla con 2 livelli di porte logiche AND - OR.

Mappe di karnaugh

Tabelle che permettono rappresentazione e semplificazione delle funzioni di commutazione, ogni casella rappresenta l'uscita corrispondente nella tabella.

La funzione sarà data da vari OR tra gli AND delle variabili aventi 1 come uscita.

Con più di 4 variabili in ingresso bisogna creare una nuova tabella per ognuna di esse.

OSS nel caso in cui la combinazione tra alcune variabile non rientra tra quelle utili nelle specifiche di progetto queste vengono segnate come “-” e possono essere usate per la riduzione come se fossero 1.

Tabella di verità

X ₀	X ₁	Y
0	0	0
0	1	1
1	0	2
1	1	3

X ₀	X ₁	Y
0	00	0
0	01	1
1	01	3
1	10	2
0	10	4
0	11	5
1	11	7
1	10	6

X ₀	X ₁	Y
00	00	0
00	01	1
01	01	3
01	10	2
11	00	4
11	01	5
11	11	7
11	10	6
10	00	12
10	01	13
10	11	15
10	10	14
00	10	8
00	11	9
01	10	11
01	11	10

ATTENZIONE utilizzano codice Gray, ovvero 11 e 00 sono invertiti.

Realizzare un circuito che riconosca se un numero compreso tra 0 e 9 sia divisibile per 3.

x ₃ x ₂ x ₁ x ₀	f
0 0 0 0	1
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
0 1 0 0	0
0 1 0 1	0
0 1 1 0	1
0 1 1 1	0
1 0 0 0	0
1 0 0 1	1
1 0 1 0	d.c.c.
1 0 1 1	d.c.c.
1 1 0 0	d.c.c.
1 1 0 1	d.c.c.
1 1 1 0	d.c.c.
1 1 1 1	d.c.c.
ex.	

$Y = \bar{x}_0 \bar{x}_1 \bar{x}_2 \bar{x}_3 + \bar{x}_2 x_1 x_0 + x_1$

X ₀	X ₁	Y
00	00	1
00	01	0
01	01	0
01	10	1
11	-	-
11	10	0
10	01	1
10	10	0

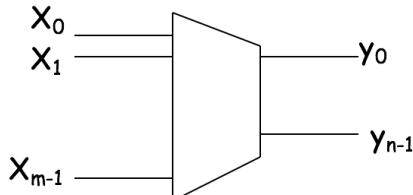
Moduli di base

è possibile dotare entrambi di un ingresso Enable, la codifica avviene solo se E=1

Codificatore

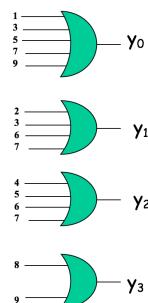
Realizza funzione di **codifica binaria**, ovvero associa ad ogni elemento di un insieme Γ composto da m simboli, una sequenza distinta di n bit $\Rightarrow (2^n \geq m)$

OSS nella realtà è molto costoso e difficilmente viene implementato in questo modo



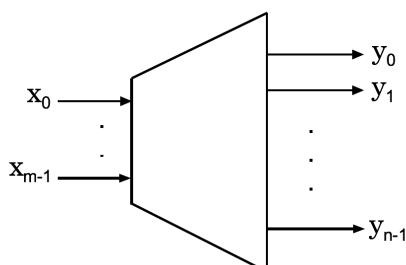
• Codifica cifre decimali in BCD

$y_3y_2y_1y_0$
0 0000
1 0001
2 0010
3 0011
4 0100
5 0101
6 0110
7 0111
8 1000
9 1001



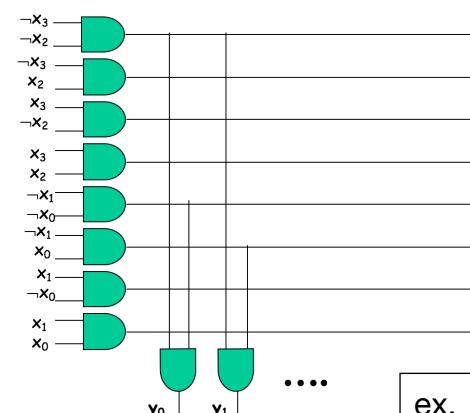
Decodificatore

Funzione inversa del codificatore, passa da codifica binaria a insieme di simboli



• Decoder BCD-Cifre decimali (seconda realizzazione)

$x_3x_2x_1x_0$	$y_9y_8y_7y_6y_5y_4y_3y_2y_1y_0$
0000	0 0 0 0 0 0 0 0 0 1
0001	0 0 0 0 0 0 0 0 1 0
0010	0 0 0 0 0 0 0 1 0 0
0011	0 0 0 0 0 0 1 0 0 0
0100	0 0 0 0 0 1 0 0 0 0
0101	0 0 0 0 1 0 0 0 0 0
0110	0 0 0 1 0 0 0 0 0 0
0111	0 0 1 0 0 0 0 0 0 0
1000	0 1 0 0 0 0 0 0 0 0
1001	1 0 0 0 0 0 0 0 0 0

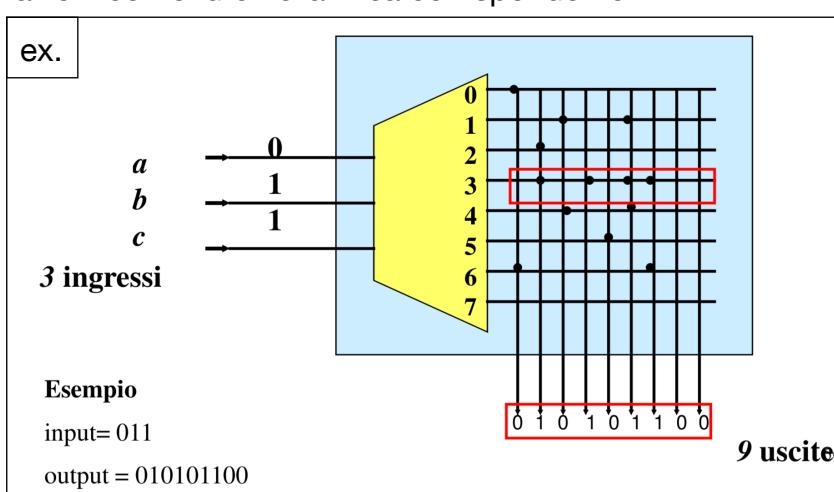


ex.

ROM

Circuito combinatorio con n ingressi (linee di indirizzamento) e m uscite (linee dati).

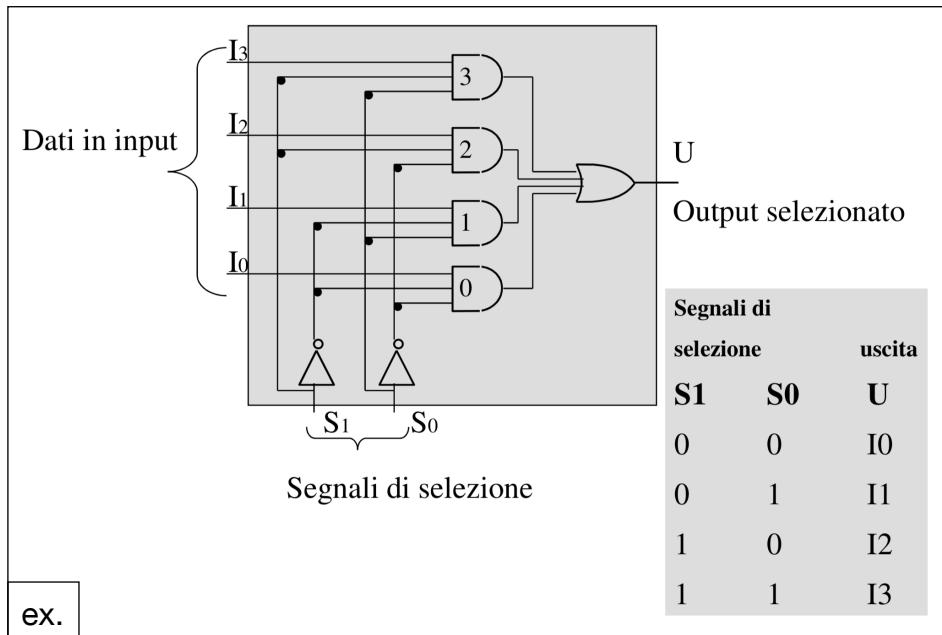
Mandata una sequenza in input (equivalente ad un indirizzo) è possibile leggere le informazioni contenute nella linea corrispondente.



Multiplexer (MUX)

Circuito combinatorio con 2^n linee in ingresso (I_0, I_1, \dots) e n segnali di selezione (S_0, S_1, \dots) che convergono in 1 unica uscita (U).

Permette di selezionare quale ingresso va in uscita tramite i segnali di selezione.

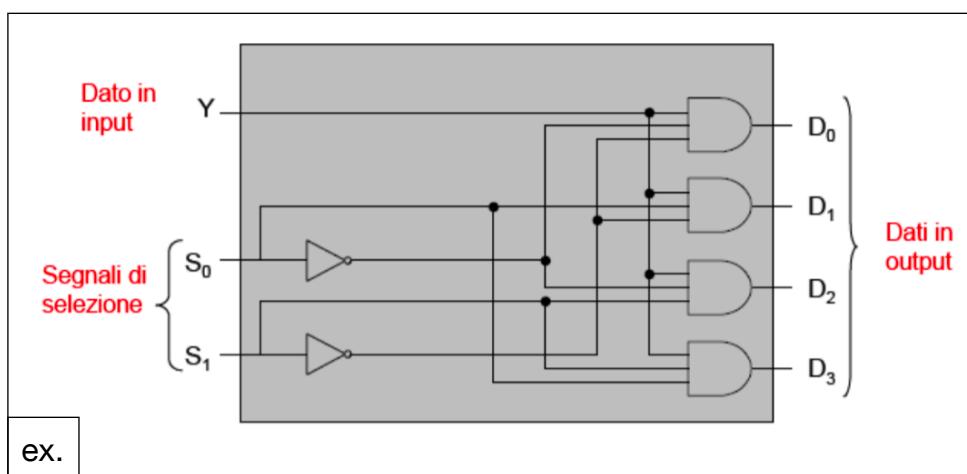


ATTENZIONE il numero di segnali di selezione dipende dal numero di segnali in input.
 $2^n \Rightarrow n$

Demultiplexer (DEMUX)

Circuito combinatorio con $n+1$ linee di ingresso, di cui 1 dato in input (Y) ed n segnali di selezione (S_0, S_1, \dots) che selezionano una delle 2^n uscite (O_0, O_1, \dots).

Funzione inversa al MUX, ricevuto un dato in ingresso seleziona su quale linea mandarlo. Tipicamente utilizzato come decodificatore.

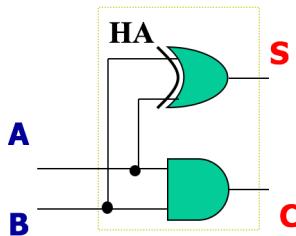


Half adder (semisommatore)

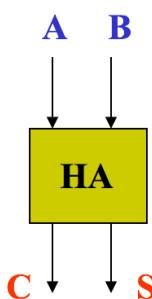
Realizza somme senza considerare il riporto precedente.

$$S = (\text{not}A)B + A(\text{not}B) = A \oplus B$$

$$C = AB$$



$$\begin{array}{rcl} A+ \\ B= \\ \hline C \quad S \end{array}$$



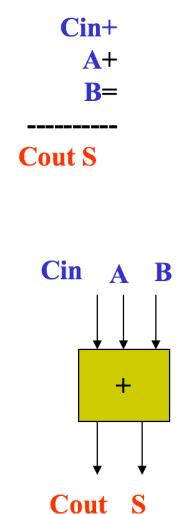
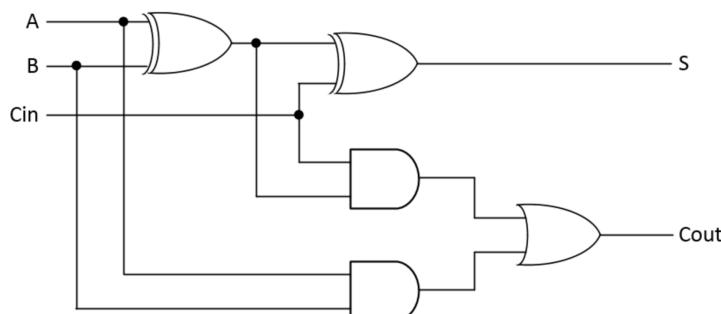
In		Out	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Full adder

Realizza somme considerando anche il **Carry in** input.

$$S = A \oplus B \oplus C$$

$$C_{out} = AB + CB + CA$$

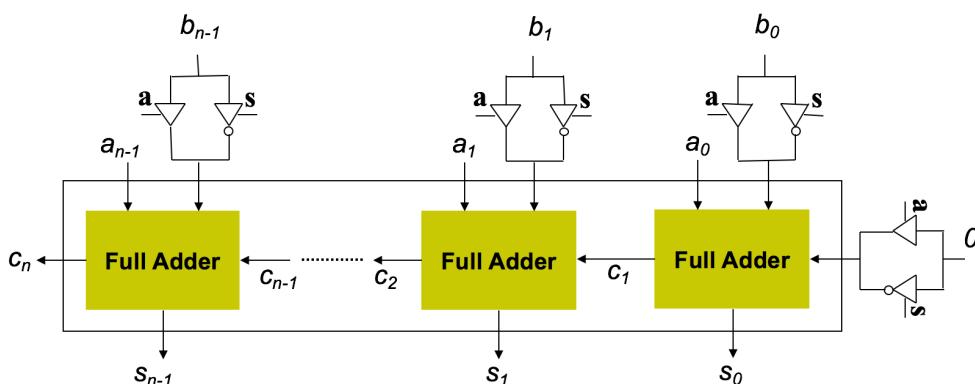


In		Out		
A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Ripple carry adder

Un adder singolo può sommare solo su 2 bit ed eventuale carry, è possibile lavorare su più cifre mettendo più adder in parallelo.

Il risultato ha $n+1$ bit, dove n indica il numero degli adder contenuti.



ATTENZIONE questo sistema soffre di ritardi nella propagazione del riporto

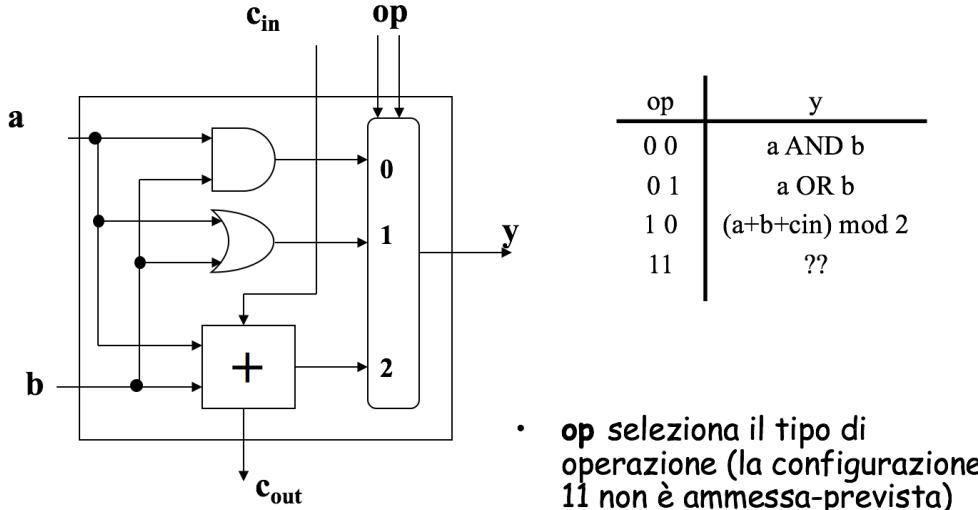
OSS la **sottrazione** si effettua con gli stessi circuiti combinatori tramite il **complemento a 2**, posizionando buffer ed inverter three state in ingresso.

ALU (unità logico-aritmetica)

Permette di implementare nello stesso blocco più tipi di operazioni aritmetiche e logiche, il numero di operazioni dipende dal tipo di componenti inclusi in fase di progettazione.

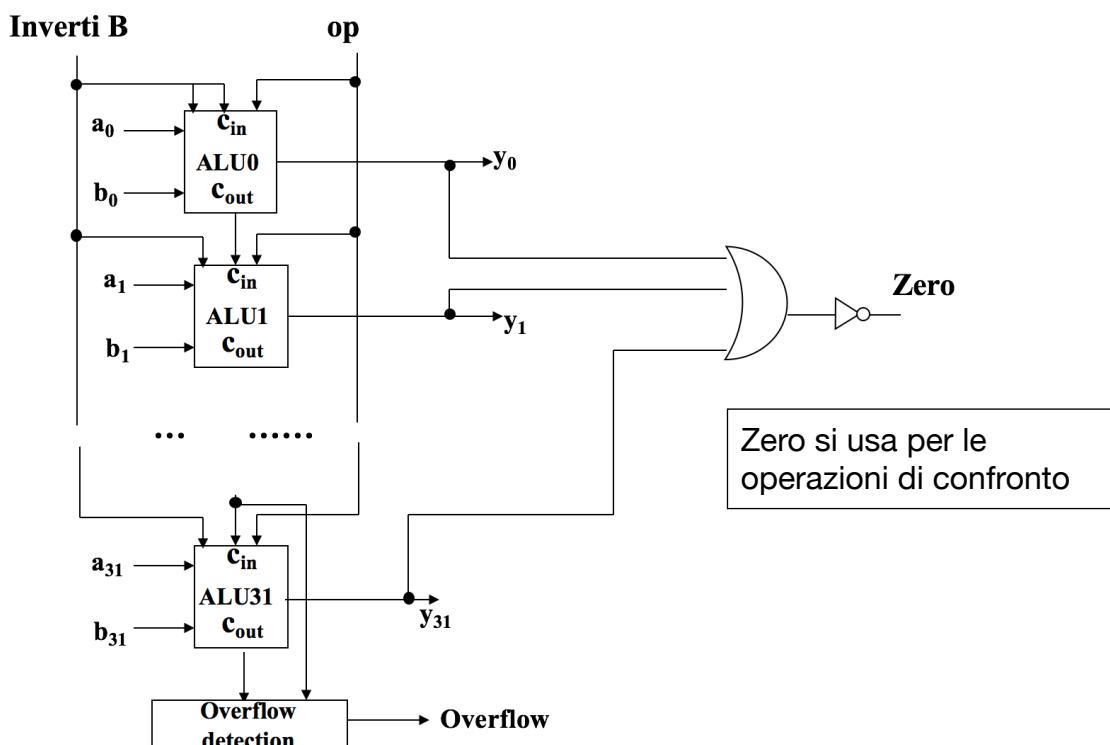
La scelta dell'operazione viene effettuata tramite un multiplexer.

ATTENZIONE permette solo operazioni su interi, operazioni in virgola mobile con FPU



Una ALU permette operazioni su un solo bit, quindi per numeri con più cifre si usano più ALU in parallelo.

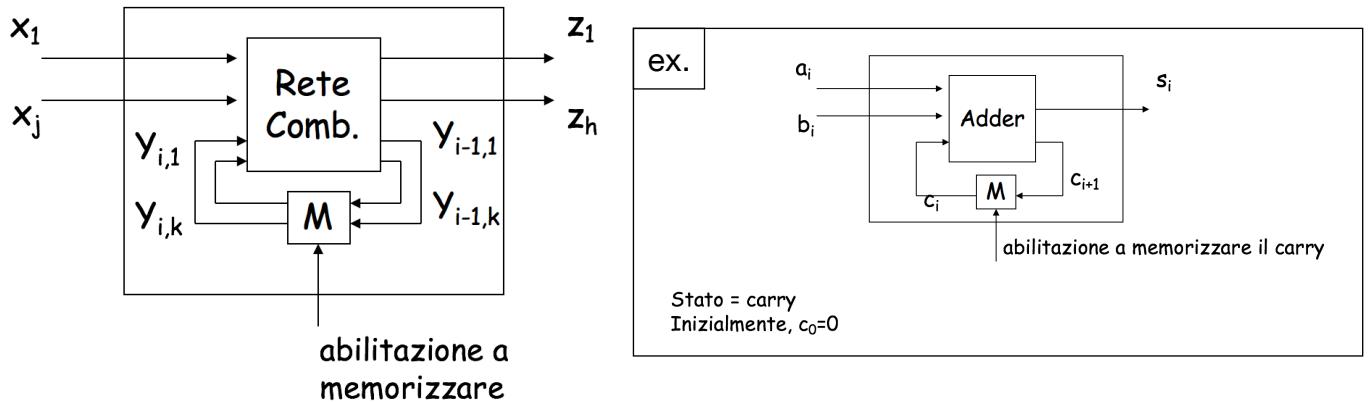
Anche in questo caso la sottrazione si effettua tramite complemento a 2 invertendo uno degli operandi, moltiplicazioni e divisioni si implementano con addizioni e sottrazioni successive.



Macchine sequenziali

Dal circuito combinatorio alla macchina sequenziale

Le macchine sequenziali sono composte da reti combinatorie alle quali vieni aggiunte una memoria per gli stati precedenti. (generalmente un flip/flop)



Rete combinatoria => valore in uscita dato dai valori in ingresso in quel momento

Rete sequenziale => valore in uscita dipende da valori in ingresso in quel momento e valori precedentemente memorizzati.

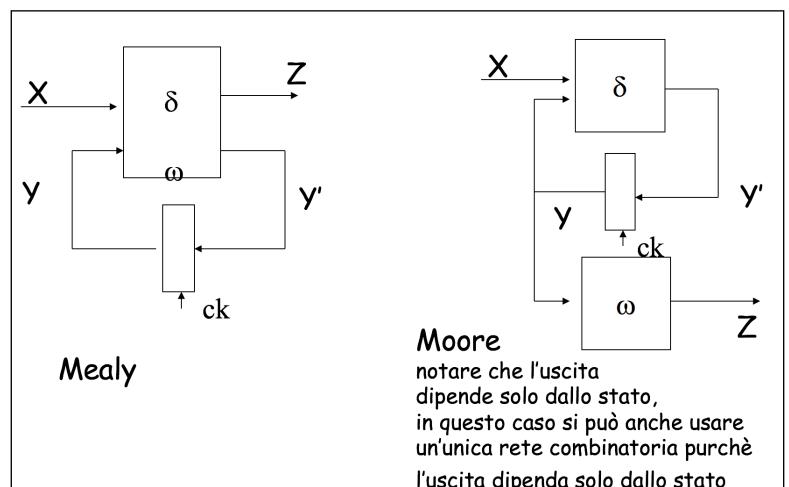
si possono essere:

- sincrone (non realizzabili)
- asincrone
- **sincrone impulsive²** (reti LLC)
- asincrone impulsive

Definizione

Una macchina sequenziale è una quintupla $MS = (I, S, O, \delta, \omega)$

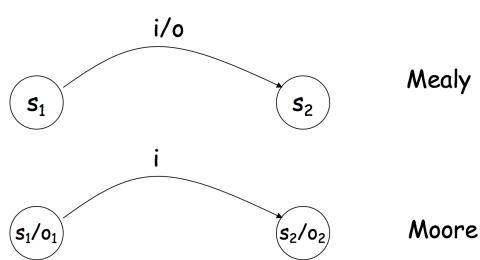
- I alfabeto in ingresso
 $I = \{i_1, \dots, i_m\}$
- S insieme degli stati
 $S = \{s_1, \dots, s_n\}$
- O alfabeto in uscita
 $O = \{o_1, \dots, o_q\}$
- δ funzione di stato successivo
 $\delta : S \times I \rightarrow S$
- ω funzione di uscita
 $\omega : S \times I \rightarrow O$ (Mealy)
 $\omega : S \rightarrow O$ (Moore)



² impulsive => scandite da clock

Rappresentazioni

- **diagramma degli stati** => grafo orientato etichettato $G(V,A,L)$



- vertiti V => insieme dei nodi (stati)
- archi A => insieme delle transazioni di stato
- etichette L => insieme delle etichette

OSS Mealy segna lo stato su ogni arco, Moore dentro lo stato

- **Tabelle degli stati/uscite** => matrici contenenti stati e uscite

- Mealy => $|S|$ righe x $|I|$ colonne
l'elemento in posizione h,k contiene prossimo stato ed uscita
- Moore => $|S|$ righe x $(|I| + 1)$ colonne
l'elemento in posizione $h,(|I| + 1)$ contiene il prossimo stato, l'uscita è contenuta in $h,(|I| + 1)$

Macchina di Mealy

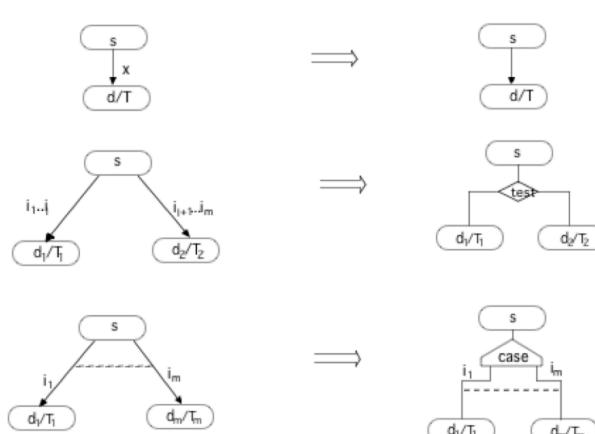
	i_1	i_2	-----	i_k	-----	i_m	
s_1				:			
s_2				:			
:				:			
s_h	---	---	-----	$\delta(i_k, s_h) / \omega(i_k, s_h)$			
:							
s_n							

Macchina di Moore

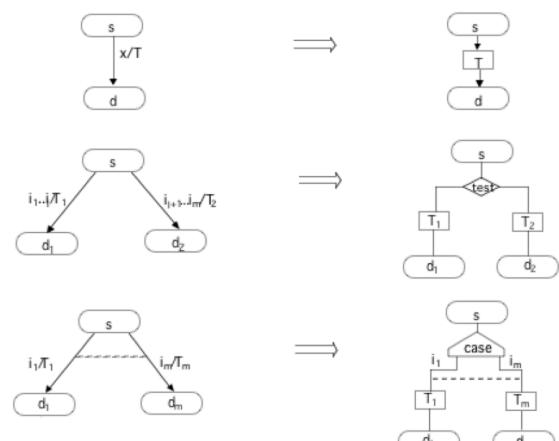
	i_1	i_2	-----	i_k	-----	i_m	ω
s_1				:			:
s_h	---	---	-----	$\delta(i_k, s_h)$			$\omega(s_h)$
s_n							

- **Algorithm state machine** => diagramma degli stati

Trasformazione del grafo in ASM: caso Moore

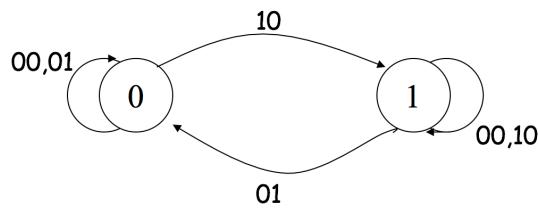


Trasformazione del grafo in ASM: caso Mealy



ex.1 Interruttore flip/flop S-R

- Ingresso: Set - Reset (S-R) - solo uno dei due ingressi può essere pari ad uno.
- Stati: 0, 1



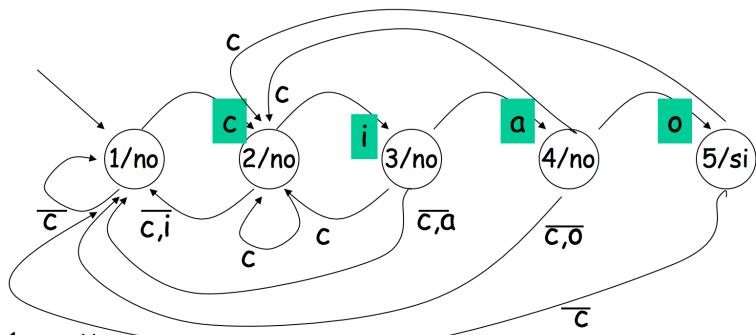
Ingressi S-R	Stato attuale	Stato succ.	Uscita
0 0	0	0	0
0 0	1	1	1
0 1	0	0	0
0 1	1	0	0
1 0	0	1	1
1 0	1	1	1

ex2. Riconoscitore sequenza "ciao" (tabella omessa per semplicità)

Alfabeto in ingresso => $I = \{a, b, \dots, z\}$

Uscite => $O = \{\text{si}, \text{no}\}$

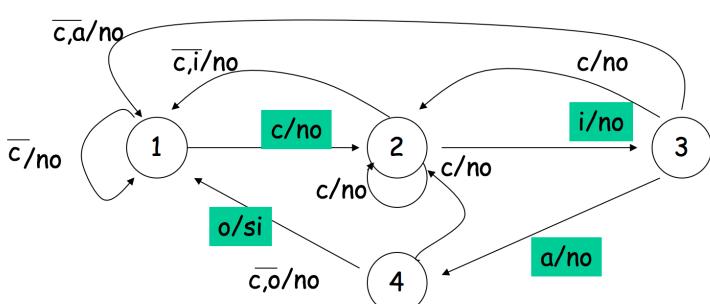
Diagramma degli stati (Moore)



- 1: aspetto c
- 2: aspetto i
- 3: aspetto a
- 4: aspetto o
- 5: parola completa

OSS Nel caso in cui la stringa debba essere riconosciuta alla prima immissione di ogni lettera, invece di avere più archi che tornano indietro, l'immissione di una lettera diversa porta in uno stato separato dal quale non è possibile uscire

Diagramma degli stati (Mealy)

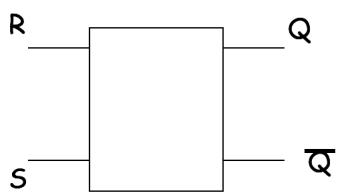
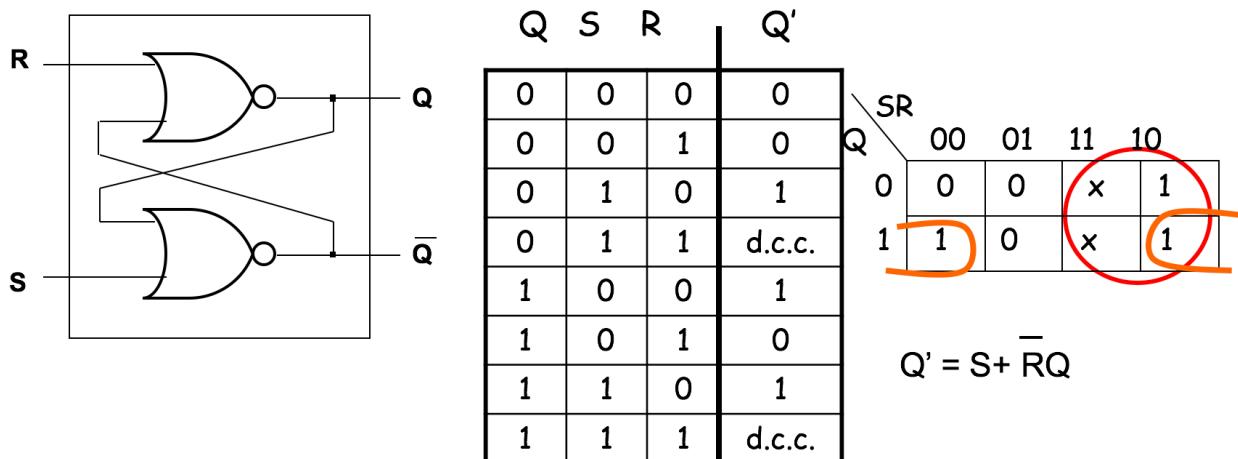


- 1: attesa c
- 2: attesa i
- 3: attesa a
- 4: attesa o

OSS per realizzare una macchina sequenziale e determinare le funzioni di eccitazione dei F/F è necessario il passaggio alla codifica in binario, effettuabile associando ad ogni simbolo una sequenza di bit.

Flip/flop (bistabile)

Il F/F è una macchina (di base asincrona) utilizzata per la memorizzazione dei dati, detto bistabile poiché ha 2 stati stabili, ovvero $\bar{Q} \neq Q$ sempre



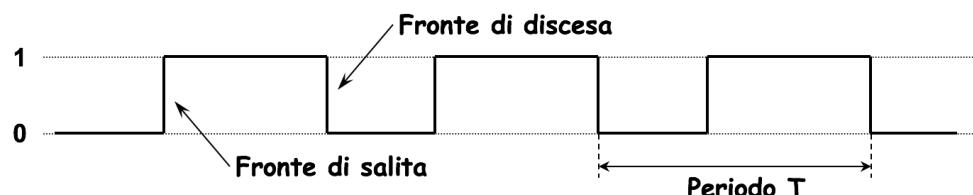
SR	Q'
00	Q
01	0
10	1
11	?

Configurazione 11 non consentita poiché creerebbe situazione di instabilità nel circuito

I F/F sincronizzati

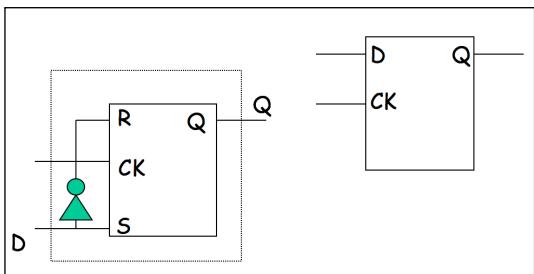
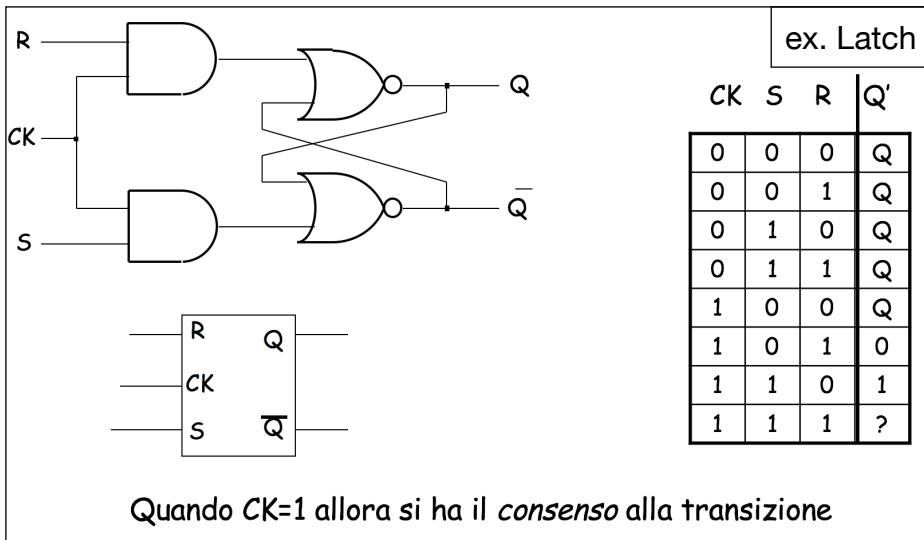
Ottenuti aggiungendo un segnale di sincronizzazione (clock) ai bistabili asincroni, l'abilitazione alla commutazione del F/F può essere fatta in vari modi:

- **Level-triggered o Latch** (abilitazione sul livello) => uscita cambia quando il segnale è a livello, ovvero Clock fermo ad 1 oppure 0 (usato per reti LLC)
- **Positive edge triggered** (abilitazione sul fronte di salita) => uscita varia solo su fronte di salita, ovvero durante passaggio del clock da 0 a 1
- **Negative edge triggered** (abilitazione sul fronte di discesa) => uscita varia solo su fronte di discesa, ovvero durante passaggio del clock da 1 a 0
- **Master slave** => ingresso si considera solo su fronte di salita, uscita cambia su fronte di discesa



OSS l'abilitazione sul fronte permette stabilizzazione degli ingressi che altrimenti potrebbero essere ancora in stato di transizione quando Clock è a livello

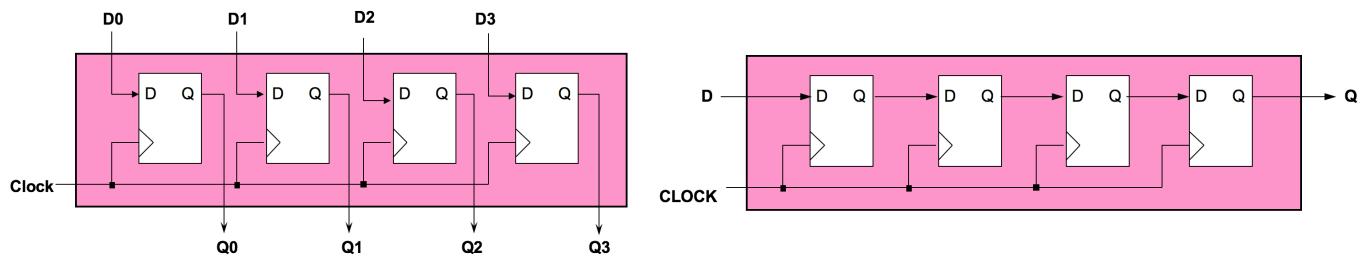
OSS dual channel delle memorie ram permette di trasferire dati sia su fronte di risalita che su fronte di discesa



EXTRA il F/F Delay latch permette di avere un unico ingresso per i dati D mettendo un inverter a monte dell'ingresso R

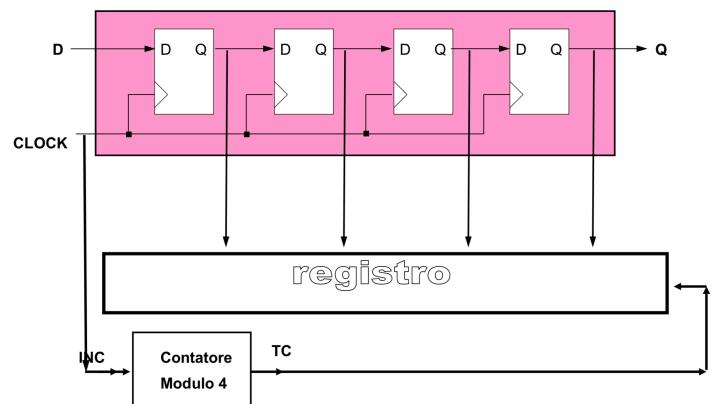
I registri

Elementi di memoria realizzati da un insieme di F/F collegati in parallelo, collegati in serie possono essere usati per creare shift register.



Una combinazione di entrambi è utilizzata per convertire i segnali da seriale a parallelo e viceversa.

- Seriale-parallelo => dati vengono ricevuti un bit alla volta e poi shiftati fino al completamento, infine salvati in parallelo su altro registro.
- Parallelo-seriale => dati inseriti in shift register ed inviati u bit alla volta shiftando ad ogni invio



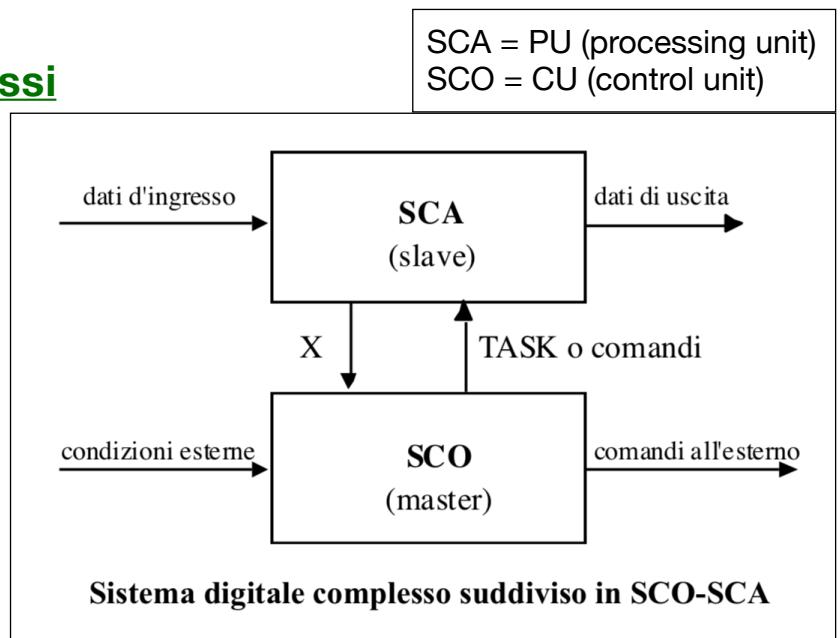
Sistemi digitali complessi

Quando si parla di sistemi complessi bisogna far attenzione a separare le unità di controllo (SCO) dalle unità di elaborazione (SCA).

SCA

Comprende tutti gli elementi utili per l'elaborazione dei dati:

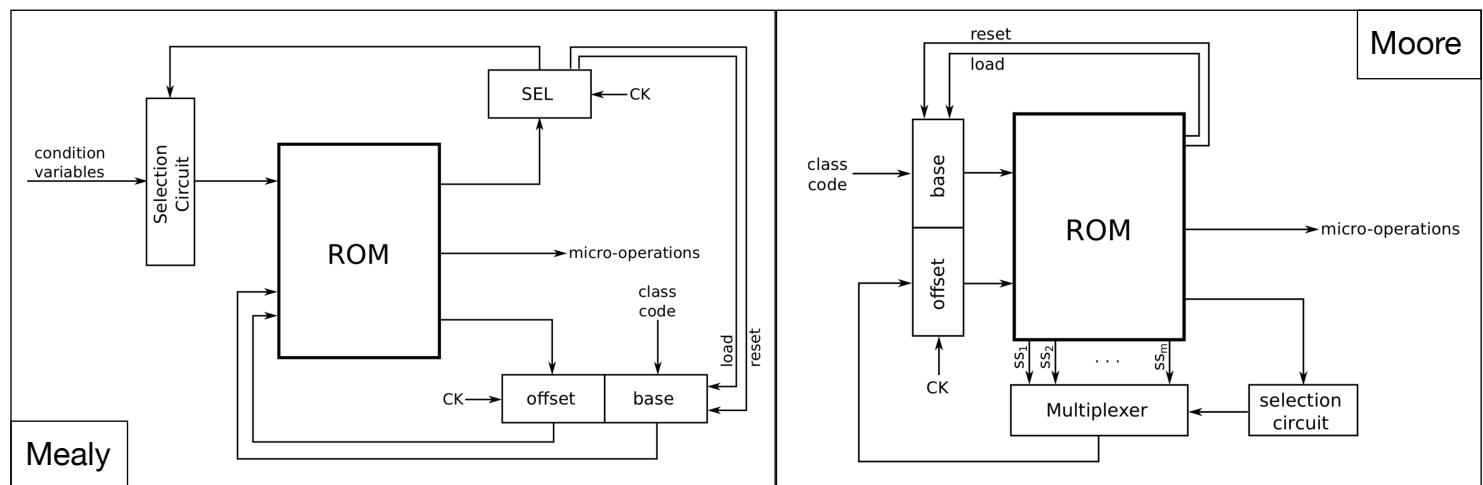
- Registri
- Operatori
- Strutture di interconnessione
- Moduli vari (MUX, DEMUX, ...)



SCO

Struttura di controllo che permette di effettuare la microprogrammazione dei componenti. Basate su macchine di Moore e Mealy ed inizialmente implementate tramite ROM.

Permette implementazione della semantica del microcodice di ogni istruzione, i circuiti di selezione servono per ottimizzare il costo del microcodice.

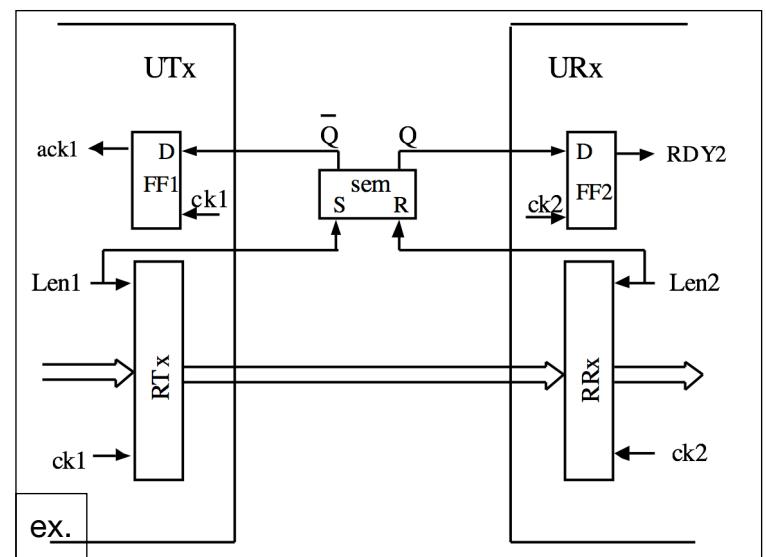


Protocollo di handshaking

Permette di simulare una connessione sincrona tra due dispositivi indipendenti che lavorano in maniera asincrona uno rispetto all'altro. Questo è possibile grazie ad un F/F comune, detto **F/F di handshaking**, che funge da semaforo.

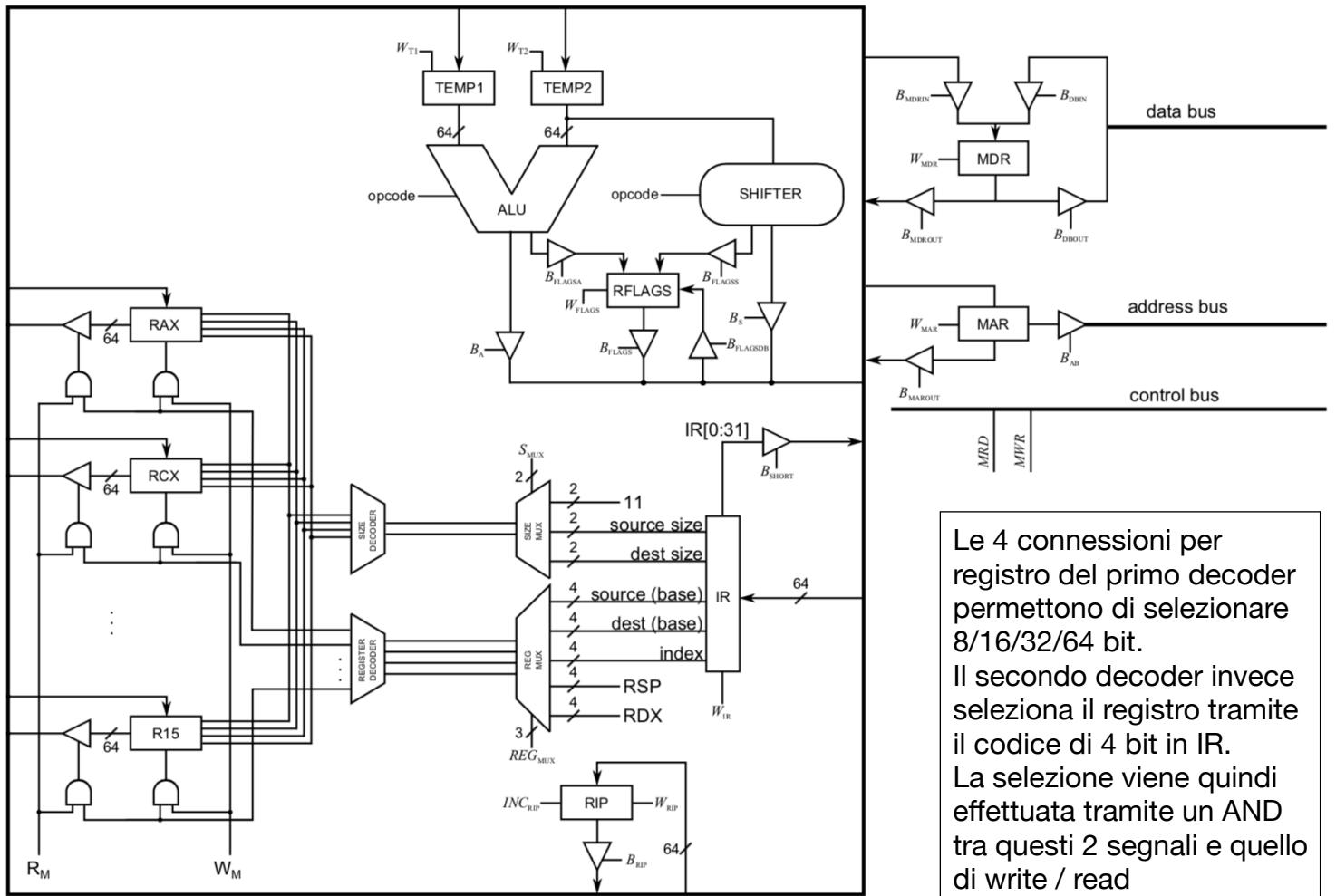
ex. Al termine di ogni operazione il dispositivo UTx invia i dati a URx ed un segnale, detto Ack.

URx ricevuto il segnale legge il dato ed una volta finite le sue operazioni invia un nuovo ack a UTx che riprenderà le sue funzioni ed invierà un nuovo dato



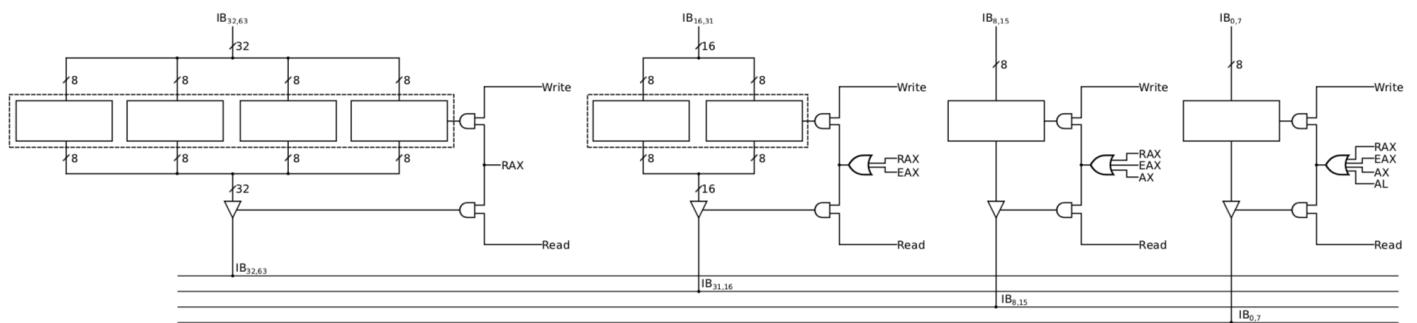
Processore Z64

Processore didattico a 64 bit basato su architettura x86 (usata da processori Intel / Amd)

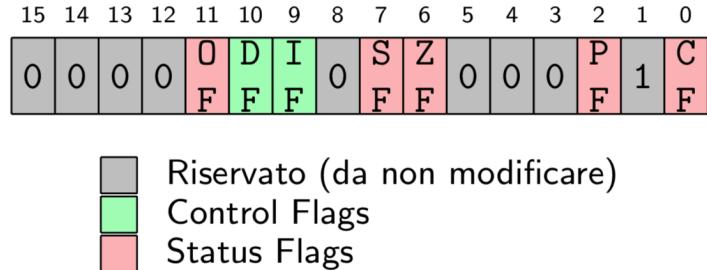


I Registri

- **TEMP1 / TEMP2** => registri nascosti che permettono di mantenere le variabili stabili durante il processamento nella ALU o shifter (sempre in funzione)
OSS si dicono nascosti poiché non visibili al programmatore
- **IR** => Instruction register, contiene dati istruzione in esecuzione
- **RIP (PC)** => program counter, contiene indirizzo prossima istruzione, modificabile da istruzioni di salto
- **MAR** => memory address register, indirizzo cella di memoria con cui interagire
- **MDR** => memory data register, contenuto della cella o dati da mandare alla cella
- **FLAGS** => contiene informazioni su esito ultima operazione (status) o flags per variare il modo di esecuzione, i suoi componenti vengono usati per condizioni di salto
- **RAX ... R15** => registri di uso generale per il salvataggio dei dati



Il registro FLAGS



- **carry (CF)**: vale 1 se l'ultima operazione ha prodotto un riporto
- **parity (PF)**: vale 1 se nel risultato dell'ultima operazione c'è un numero pari di 1
- **zero (ZF)**: vale 1 se l'ultima operazione ha come risultato 0
- **sign (SF)**: vale 1 se l'ultima operazione ha prodotto un risultato negativo
- **overflow (OF)**: vale 1 se il risultato dell'ultima operazione supera la capacità di rappresentazione
- **interrupt enable (IF)**: indica se c'è la possibilità di interrompere l'esecuzione del programma in corso
- **direction (DF)**: modifica il comportamento delle operazioni su stringhe

Il BUS

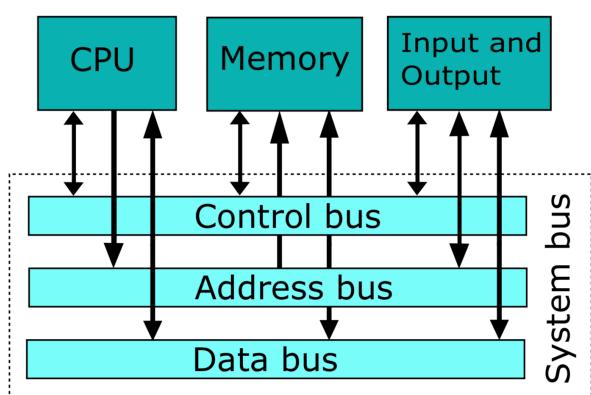
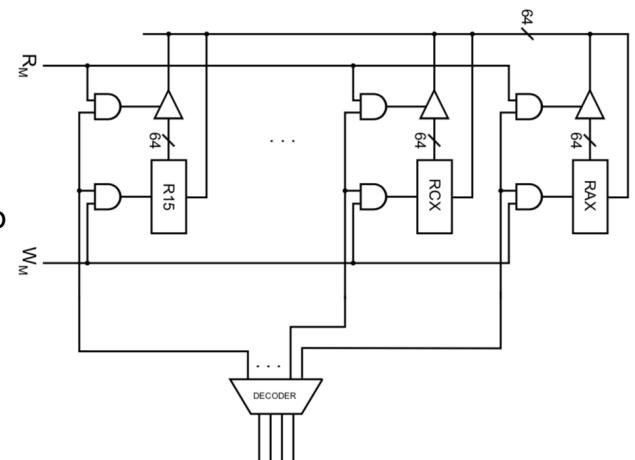
Struttura di collegamento tra i vari componenti, ogni elaboratore ha 2 tipi di bus, uno interno al processore ed uno esterno per connessione con memoria e periferiche.

Lo Z64 è dotato di un unico BUS interno, quindi un solo registro alla volta può scrivere su di esso, i dati vengono mantenuti sul bus finché un altro registro li sovrascrive.

Permette di effettuare 2 tipi di operazioni:

- **Trasmissione dati** => contenuto del registro posto sul bus, segnale tramite un **buffer three state**
- **ricezione dati** => bit presenti sul BUS vengono salvati nel registro, segnale **write enable**

OSS teoricamente sarebbe possibile scrivere su più registri contemporaneamente ma non esiste istruzione assembly



Anche il BUS esterno è unico ma diviso in 3 componenti:

- Control bus => specifica lettura / scrittura
- Address bus => specifica indirizzo
- Data bus => contiene dati

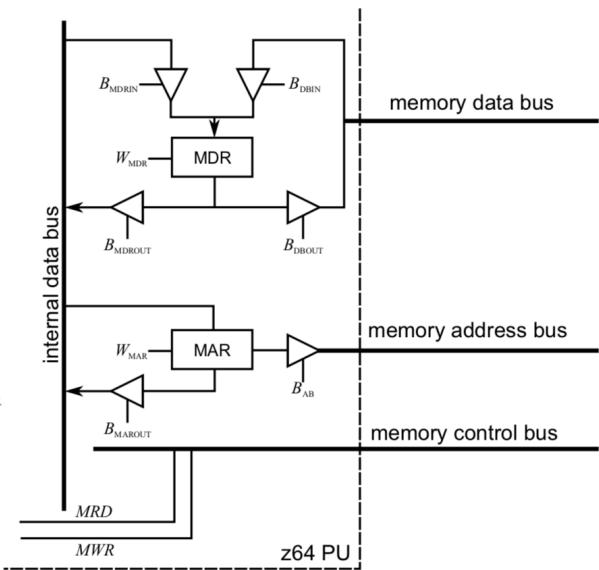
Interazione con la memoria

La memoria può essere vista come un lungo nastro diviso in celle (8 bit per cella), ognuna delle quali identificata da un numero intero progressivo detto indirizzo.

Contiene sia dati che istruzioni.

Per interagire con essa si utilizzano i due registri tampone MAR e MDR che permettono al processore di non bloccarsi in attesa dei dati in memoria pur avendo un solo bus interno.

Ad ogni interazione il processore posiziona indirizzo in MAR e manda segnale lettura o scrittura tramite memory control bus, in caso di scrittura carica dati su MDR, altrimenti lo utilizza per recuperare le informazioni una volta completate le operazioni della memoria.



OSS la linea di ritorno dal MAR serve per il calcolo degli indirizzi (vedere microistruzioni)

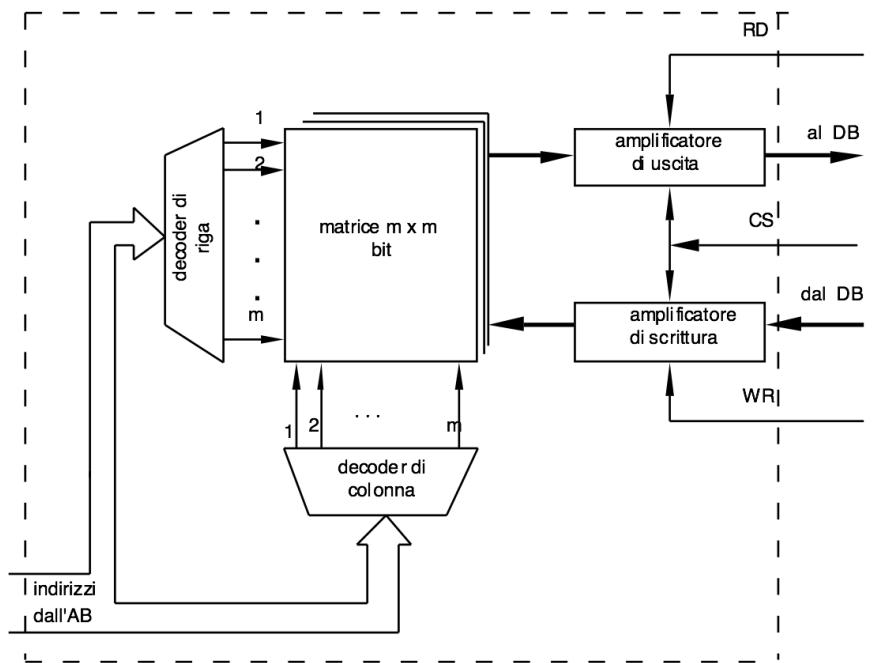
La memoria

Siccome la capacità delle memorie dei calcolatori moderni è molto elevata, queste vengono divise in moduli contenenti a loro volta blocchi di memoria.

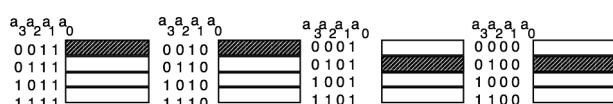
Il modo in cui vengono divisi i moduli cambia a seconda dell'elaboratore.

I dati vengono caricati o immagazzinati tramite una matrice di righe e colonne.

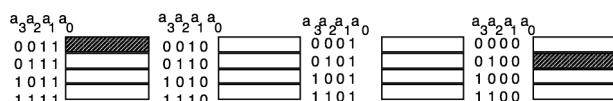
Per capire in quale modulo sono salvati i dati richiesti si fa riferimento ad un determinato numero di bit dell'indirizzo che dipende dalla quantità di moduli.



Esempio di memorizzazione di una informazione di quattro byte allineati sullo stesso indirizzo di riga.



Esempio di memorizzazione di una informazione di quattro byte disallineati



Esempio di memorizzazione di una informazione di due byte disallineati

Allineamento

A causa della divisione in moduli un dato più lungo della riga del modulo deve essere diviso su più moduli.

La memoria si dice allineata quando tutti i componenti di un dato si trovano sulla stessa riga in ogni modulo.

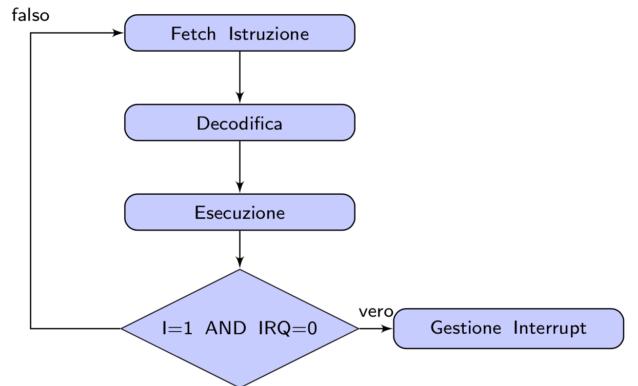
Il processamento delle istruzioni

Un ciclo istruzione si può dividere in 3 fasi:

- **FETCH** => l'istruzione contenuta in memoria viene copiata nel registro IR.
OSS indirizzo istruzione in RIP viene incrementato automaticamente dopo ogni fetch ancor prima di eseguire il decode
- **DECODE** => identificazione del tipo di istruzione da eseguire (microcodice)
- **EXECUTE** => control unit implementa la semantica dell'istruzione

Alla fine di ogni ciclo istruzione il processore controlla lo stato del interrupt flag e la presenza di richieste di interrupt, se le condizioni non sono verificate ricomincia, altrimenti gestisce l'interrupt.

Questo ciclo viene ripetuto all'infinito finché la macchina non viene spenta o riceve l'istruzione **HLT** che manda il processore in IDLE, ovvero rimane in attesa di interrupt senza eseguire alcuna operazione.

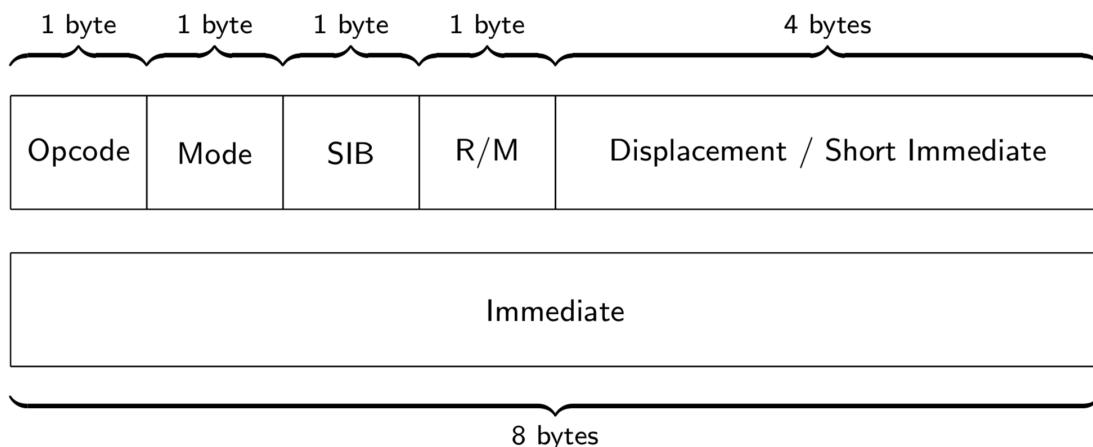


Formato istruzioni

Le istruzioni eseguibili dal processore si possono dividere in classi:

0. Controllo dell'hardware
1. Spostamento dati
2. Aritmetiche e logiche
3. Rotazione e shift
4. Operazioni su bit di flags
5. Controllo flusso di esecuzione del programma
6. Controllo condizionale del flusso di programma
7. Ingresso / uscita dati

Ognuna di esse per essere processata viene caricata nel IR secondo questo formato:

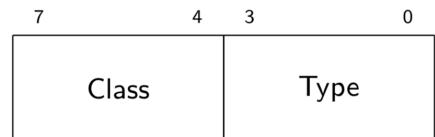


OSS quindi operazioni possono avere formato fino a 128 bit a seconda del tipo di istruzione, l'unico campo sempre presente è opcode

OSS da notare come ogni registro ha un codice di 4 bit per essere identificato dal decoder che si occupa della selezione degli operandi

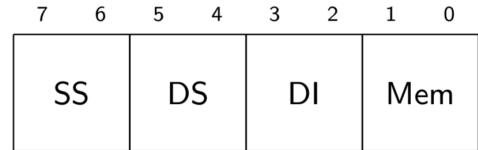
Opcode

- **Class** => codice di 4 bit, famiglia istruzione
- **Type** => identifica istruzione nella classe



Mode

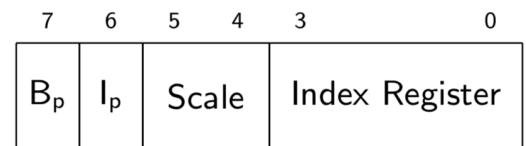
- **SS** => dimensione della sorgente
- **DS** => dimensione della destinazione
- **DI** => indica presenza di displacement o immediato
- **Mem** => indica se e quale tra sorgente e destinazione sono presenti in memoria



SIB (scala, indice, base)

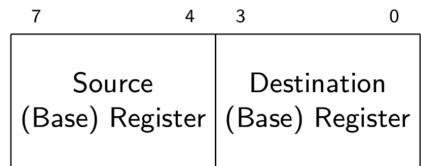
Presente solo nel caso di indirizzamento in memoria

- **B_p** => se 1 indirizzamento usa base
- **I_p** => se 1 indirizzamento usa indice
- **Scale** => valore della scala
(00 = 1) (01=1) (10=2) (11=4)
- **Index register** => codifica binaria registro indice



R/M

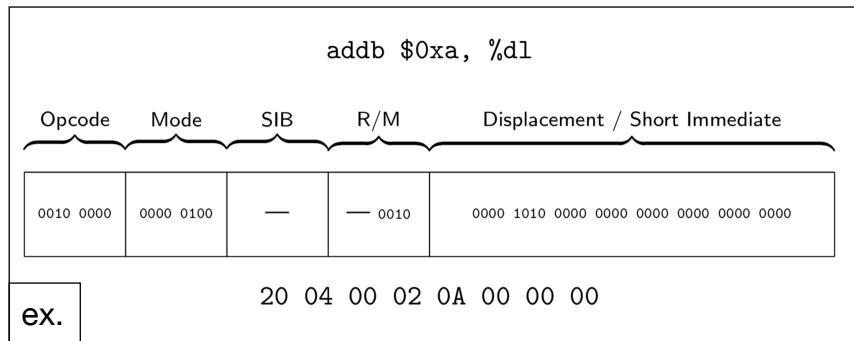
- **src register** => codifica registro sorgente
 - **Dest register** => codifica registro destinazione
- OSS possono essere anche basi di indirizzamento



Displacement / immediate

Contengono valori \$immediate rispettivamente a 32 e 64 bit

ATTENZIONE indirizzamento usa l'immediate a 64 bit, quindi si possono caricare dati fino ad un massimo di 32 bit



Endianess

Ordine in cui vengono salvati i byte di un dato

- **little-endian** => da byte meno significativo a più significativo, usato da processori x86 poiché utile per il cast
- **Big-endian** => da byte più significativo a meno significativo, usato per il protocollo "network-byte order" di internet

OSS ogni volta che un processore x86 invia dati sulla rete deve cambiare l'ordine dei byte
ATTENZIONE la codifica avviene a gruppi di 2 byte

Le microoperazioni

Lo Z64 è un processore mult ciclico, ovvero ad ogni ciclo istruzione corrispondono più cicli macchina (operazioni elementari eseguite in un solo ciclo di clock).

Costo del microcodice

Ipotizzando di non eseguire ottimizzazioni:

- $N_0 \Rightarrow$ stato necessario per il fetch
- $k \Rightarrow$ stati necessari per tutti i sottoprogrammi $N_i \quad \forall i \in [1, M] - M_i$

$$k = \log_2 \sum_{i=0}^M N_i$$

- $m \Rightarrow$ numero segnali in input
- $r \Rightarrow$ numero variabili in output

La ROM corrispondente sarà

- **Mealy => 2^k parole di $k2^n+2$ bit**
- **Moore => 2^{m+k} parole di $k+2$ bit**

Per evitare un costo troppo alto del microcodice è possibili utilizzare circuiti di selezione che permettono di mascherare segnali superflui. (vedere pag. 13)

Ad esempio in questo modo invece di associare ogni tipo di variabile di condizione ad ogni istruzione posso limitare ognuna di esse ad avere solo quelle realmente necessarie.

Inoltre, avendo che ogni istruzione corrisponde ad una linea della ROM, questa viene divisa in blocchi di classi dove la base corrisponde alla prima istruzione della classe, e le altre istruzioni vengono indicate come spiazzamento da essa.

Questo però comporta spazi vuoti poiché non tutte le classi hanno lo stesso numero di istruzioni e non tutte le istruzioni sono della stessa lunghezza.

OSS nei processori moderni le ROM non vengono comunque utilizzate poiché anche con il massimo delle ottimizzazioni occuperebbero 1/4 del processore

Microcodice FETCH

1. $\text{MAR} \leftarrow \text{RIP}$
 2. $\text{MDR} \leftarrow \text{MAR}; \text{RIP} \leftarrow \text{RIP} + 8$
 3. $\text{IR} \leftarrow \text{MDR}$
-
1. $B_{\text{RIP}} = 1; W_{\text{MAR}} = 1$
 2. $INC_{\text{RIP}} = 1; B_{\text{AR}} = 1$
 $B_{\text{AR}} = 1; MDR = 1$
 $B_{\text{AR}} = 1; MDR = 1; B_{\text{DBIN}} = 1; W_{\text{MDR}} = 1$
 3. $W_{\text{IR}} = 1; B_{\text{MDROUT}} = 1$

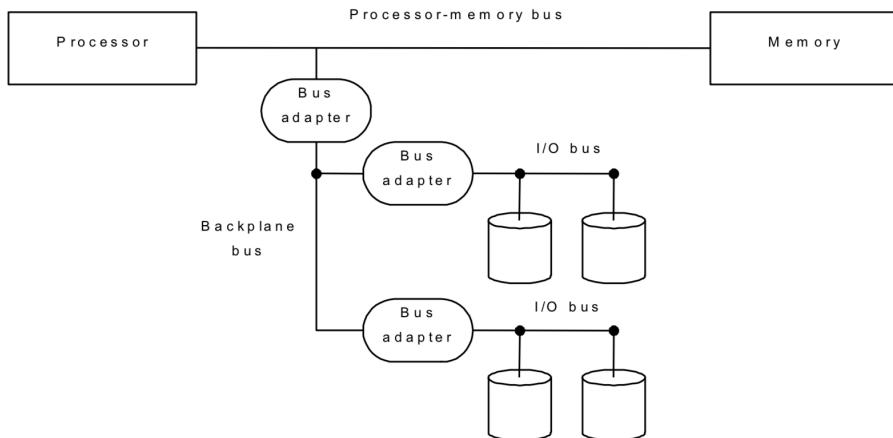
I dispositivi di I/O

Le istruzioni di classe 7 permettono di interfacciare il processore con periferiche di I/O. Questo può avvenire utilizzando un BUS separato (con I/ODR e I/OAR corrispondenti) oppure utilizzando lo stesso BUS usato per le memorie al quale viene aggiunto un segnale I/O che indica se l'indirizzo appartiene alla memoria oppure ad una periferica.

ATTENZIONE Tutti i segnali esterni al processore lavorano in logica negata (invertiti)

Tipologie di BUS

- I. **BUS processore-memoria** => alta velocità ma lunghezza ridotta, sincrono
- II. **BUS di I/O** => più lenti ma di lunghezza maggiore e permettono diversi tipi di connessioni, asincroni
- III. **BUS backplane** => struttura intermedia tra bus processore-memoria e bus I/O



BUS SINCRONO:
protocollo scandito da cicli di clock, più veloce ma necessita sincronizzazione di tutti i dispositivi collegati

BUS ASINCRONO:
Usa protocollo di handshaking, più lungo ma velocità dipende dal dispositivo

OSS è inoltre possibile dividere i bus in **bus paralleli** che hanno una linea per ogni bit e **bus seriali** che inviano tutti i dati sulla stessa linea

Istruzioni di input / output

Input => **inX %dx, %?a?**

Output => **outX %?a?, %dx**

Per utilizzare queste istruzioni bisogna prima salvare l'indirizzo del componente della periferica con la quale si vuole interagire in **%dx**.

In caso di output bisogna salvare il dato nel registro accumulatore **%?a?**, in caso di input il dato verrà scritto in questo registro.

Invece per trasferire un numero maggiore di dati con istruzione unica si possono usare:

Input => **insX %dx**

Output => **outsX %dx**

Il numero di dati da trasferire è contenuto in **%RCX**, la prima locazione di memoria in cui salvare o dalla quale caricare i dati in **%RDI**, ovviamente sono istruzioni bloccanti.

OSS l'utilizzo di un registro a 16 bit permette indirizzamento fino 65536 dispositivi

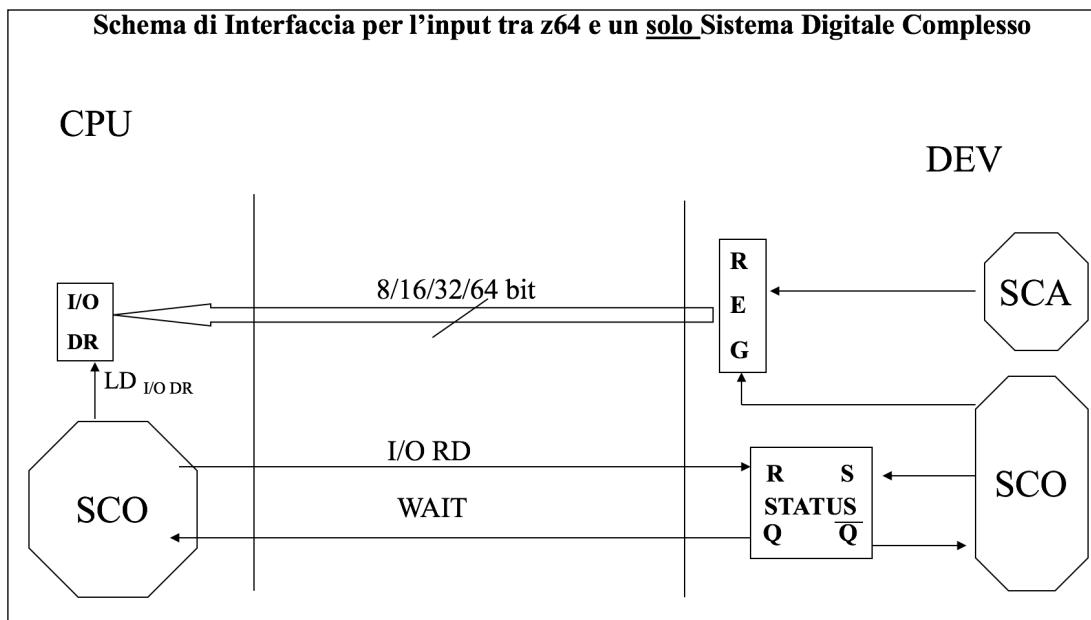
ATTENZIONE Le istruzioni della classe 7 sono implementate a firmware e non sono disponibili in user space, inoltre x86 permette trasmissione di dati fino a massimo 32 bit

Tipologie di interazione tra CPU e periferiche

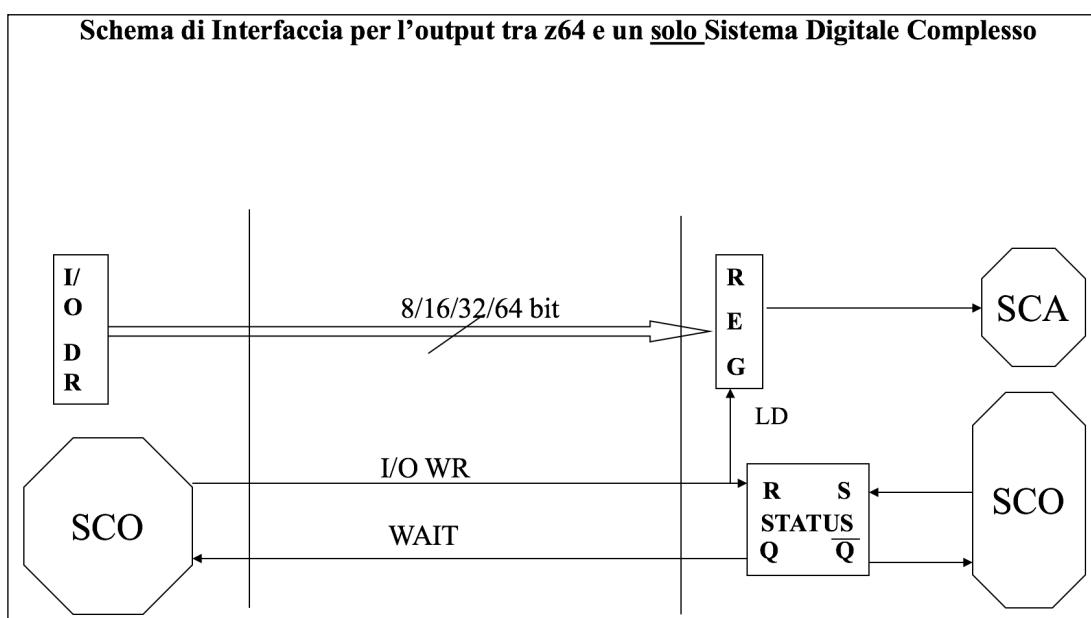
In tutti gli schemi si fa riferimento a registri I/ODR e I/OAR ma il protocollo è equivalente usando un solo BUS e quindi i registri MAR e MDR

Busy waiting (firmware, 1 dispositivo)

Basato su protocollo di handshaking, rende trasferimento semi-sincrono bloccando la CPU per tutto il tempo necessario per lo scambio dei dati.

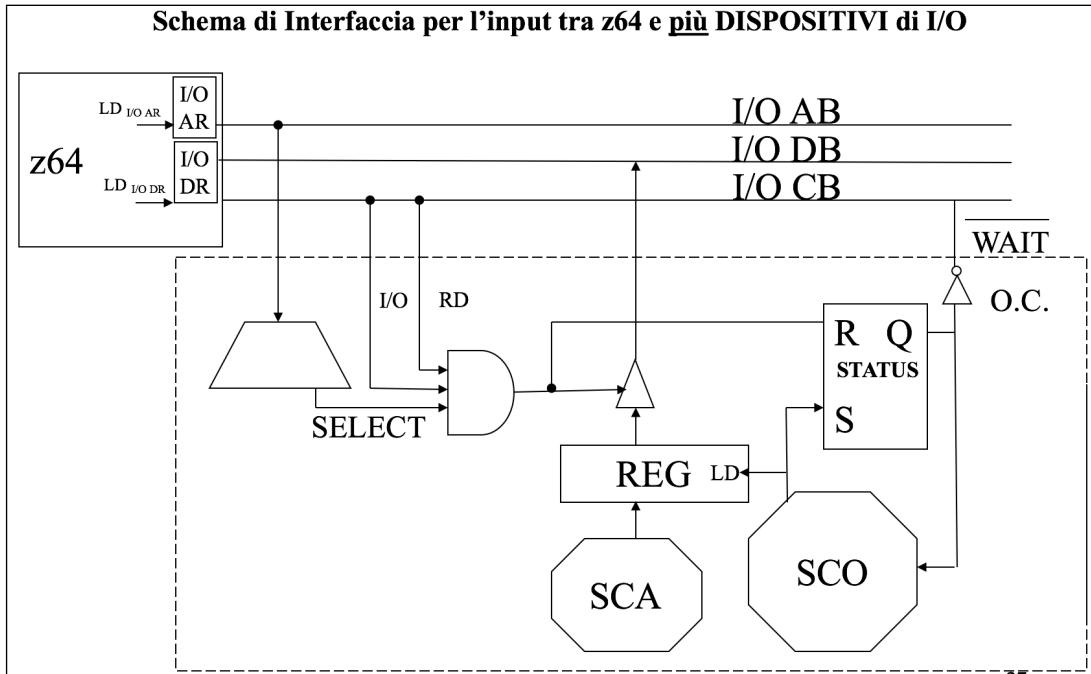


1. Processore richiede dato tramite I/O RD
2. F/F status invia segnale WAIT
3. SCA dispositivo produce dato e lo posiziona nel suo REG
4. SCO dispositivo disabilita WAIT e CPU legge dato

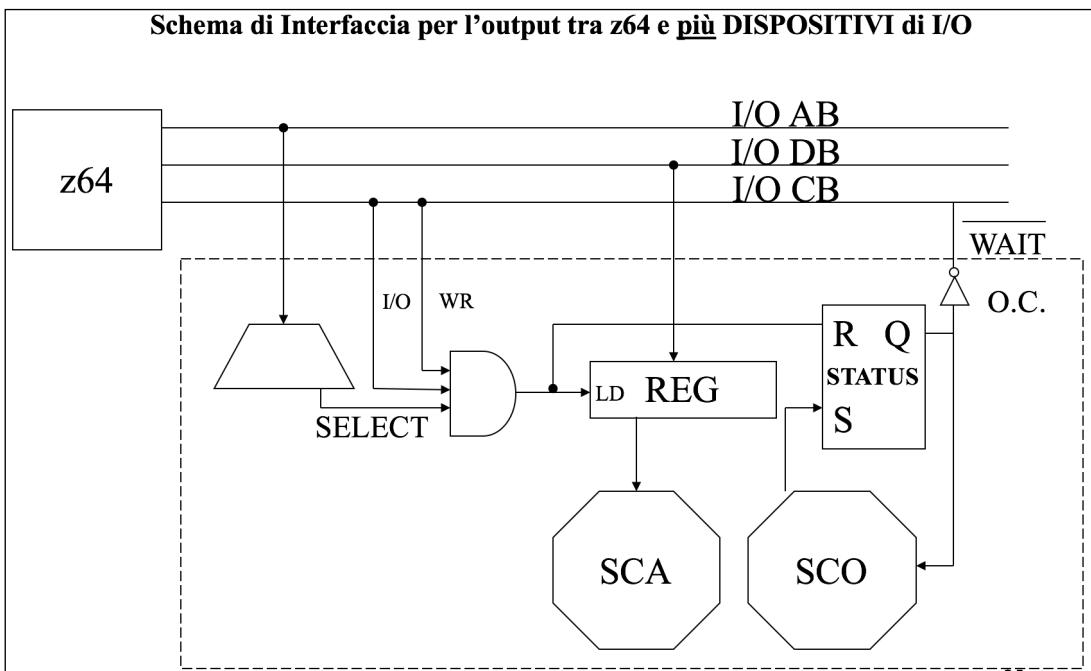


1. CPU invia segnale I/O WR e carica dato in REG
2. F/F status invia segnale di WAIT
3. SCA dispositivo legge il dato
4. SCO dispositivo disabilita segnale WAIT

Busy waiting (firmware, più dispositivi)



1. CPU invia segnale RD (con eventuale I/O) e tramite AND con uscita del decoder seleziona la periferica
2. Si aziona il buffer-three state del REG e F/F status invia segnale WAIT
3. Quando SCA ha prodotto il dato la SCO attiva scrittura su REG e disabilita WAIT



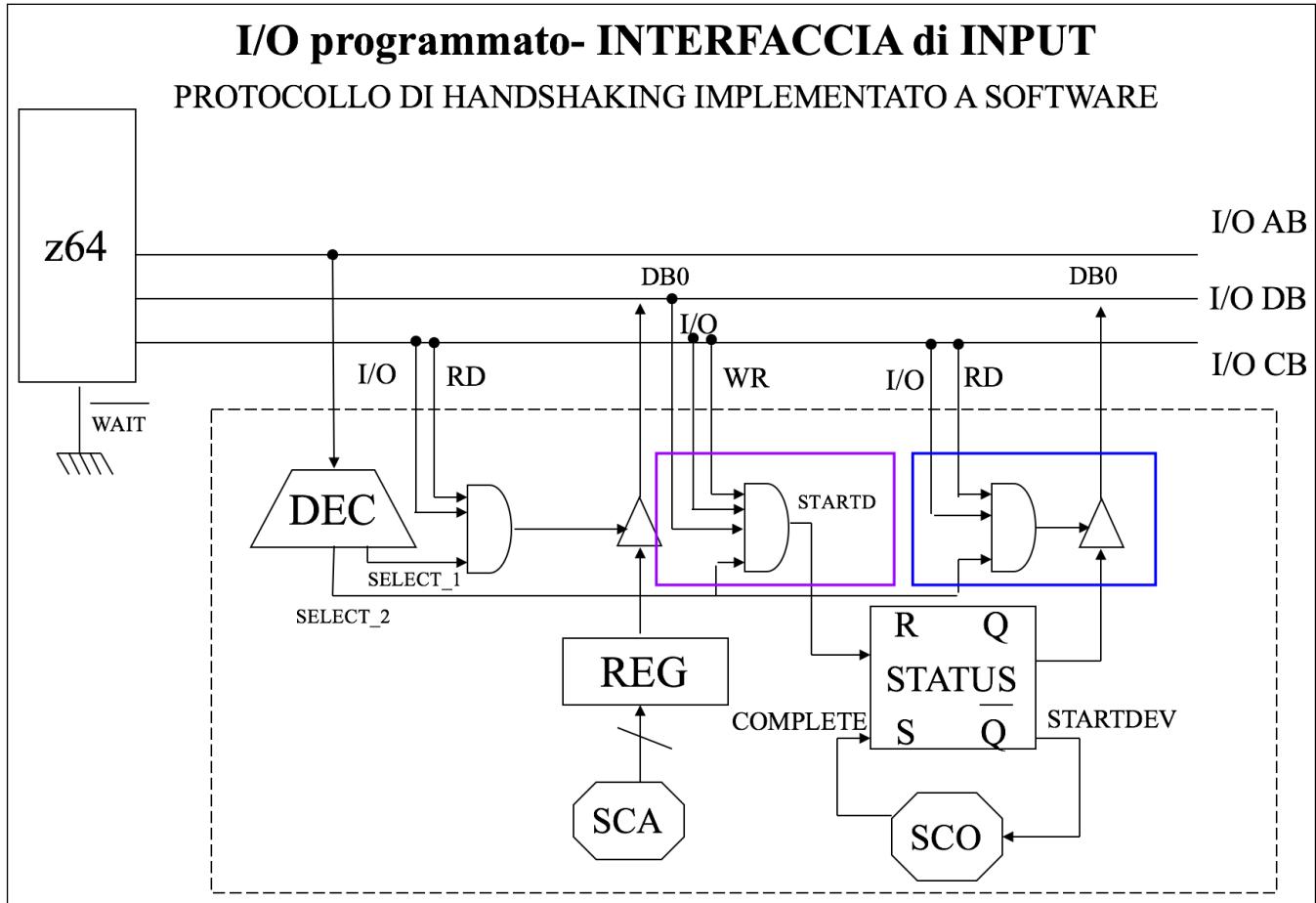
1. CPU invia segnale WR (con eventuale I/O) e tramite AND con uscita del decoder seleziona la periferica
2. Si abilita scrittura su REG e F/F status invia segnale WAIT
3. Finita la trasmissione SCO disabilita segnale WAIT

OSS nella realtà non si usa un decoder poiché troppo costoso

OSS gli indirizzi vengono assegnati dal calcolatore durante la fase di boot dello stesso

Busy waiting (software)

Per implementare le stesse funzioni via software il WAIT viene messo a terra per eliminarne gli effetti, il controllo si effettua via software.



1. (hardware viola) CPU invia segnale WR (con eventuale I/O) e scrive 1 sul data bus, tramite AND con uscita del decoder seleziona la periferica e la avvia
2. (hardware blu) Mentre SCA produce il dato la CPU inizia ciclo di controllo su F/F status usando segnale RD
3. IF (status == 0) ricomincia ciclo
4. IF (status == 1) seleziona indirizzo per buffer-three state e preleva dato da REG

CODICE ASSEMBLY:

```

        movw $FF_STATUS, %dx          #codice per avvertire periferica
        movb $1, %al
        outb %al, %dx

.aspetta:    inb   %dx, %al      #preleva status
            btb   $0, %al      #copia bit 0 di al nel carry flag

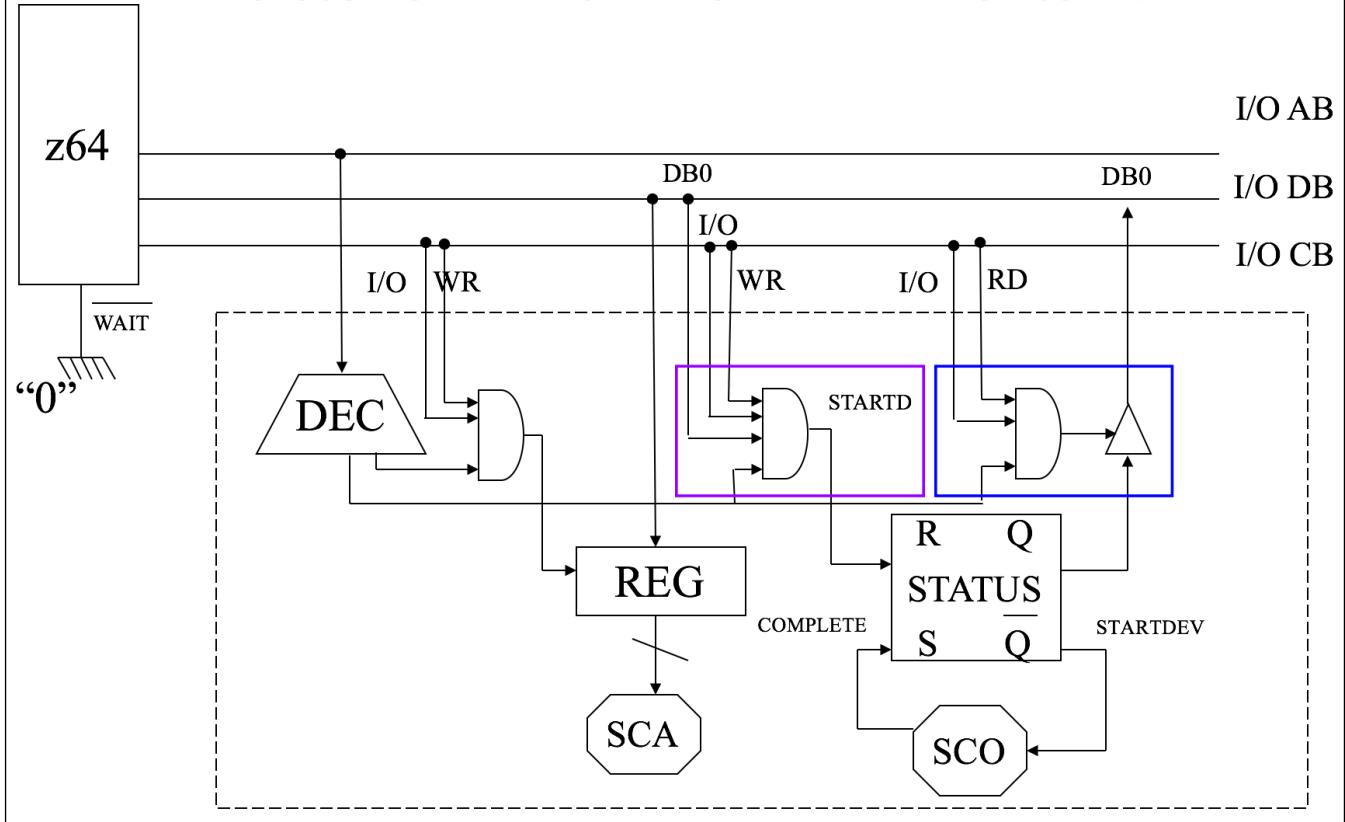
            jnc   .aspetta     #salta su aspetta se carry flag = 0
                                #se carry = 1 il ciclo termina

            movw $DEVICE_IN, %dx
            inl   %dx, %eax     #preleva dato richiesto da periferica

```

I/O programmato- INTERFACCIA di OUTPUT

PROTOCOLLO DI HANDSHAKING IMPLEMENTATO A SOFTWARE



1. CPU invia segnale WR (con eventuale I/O), tramite AND con indirizzo seleziona la periferica e scrive dato su registro REG
 2. (**hardware viola**) CPU invia segnale WR (con eventuale I/O) e scrive 1 sul data bus, tramite AND con uscita del decoder seleziona la periferica e la avvia
ATTENZIONE indirizzo cambia da punto 1 a punto 2
 3. (**hardware blu**) Mentre SCA preleva il dato la CPU inizia ciclo di controllo su F/F status usando segnale RD (
 4. IF (status == 0) ricomincia ciclo
 5. IF (status == 1) scambio dati terminato

CODICE ASSEMBLY:

Polling

Utilizzando il Busy waiting per il trasferimento di molti dati su un'unica periferica si mantiene la CPU in blocco per lunghi periodi, per risolvere questo problema si introduce il polling che permette al processore di interrogare ciclicamente tutte le periferiche (invece che sempre la stessa) per sapere quando una di queste ha un dato pronto.

CODICE ASSEMBLY:

```
.poll:      movw $STATUS_DEV1, %dx      #carica indirizzo prima periferica
            inb   %dx, %al      #preleva status
            btb   $0, %al      #copia bit 0 di al nel carry flag
            jc    .dev1        #se carry = 1 salto al codice di DEV1

            movw $STATUS_DEV2, %dx      #carica indirizzo seconda periferica
            inb   %dx, %al      #preleva status
            btb   $0, %al      #copia bit 0 di al nel carry flag
            jc    .dev2        #se carry = 1 salto al codice di DEV2

            #.....
            jmp  .poll         #ricomincia polling
```

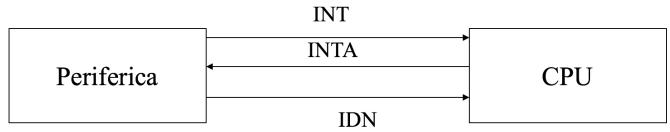
Interrupts

Metodo che permette gestione intelligente delle risorse affidando alle periferiche il compito di avvertire il processore una volta terminate le operazioni interne. Permette inoltre gestione delle urgenze tramite priorità.

Il metodo può essere suddiviso in 5 fasi:

1. **abilitazione/disabilitazione interrupti** => settando l'Interrupt flag tramite le istruzioni assembly **CLRI** (pone IF = 0) e **SETI** (pone IF = 1), bloccare gli interrupt serve per evitare che questi danneggino la corretta esecuzione dei programmi.
2. **Verifica richieste di interrupt** => le richieste **IRQ** (interrupt request) avvengono in modo asincrono rispetto alla normale esecuzione dei programmi, il processore controlla la presenza di richieste alla fine di ogni ciclo istruzioni.
3. **Identificazione sorgente interrupt** => ricevuta la IRQ il processore genera segnale **INTA** (interrupt acknowledgement) per avvertire periferica che è pronto.

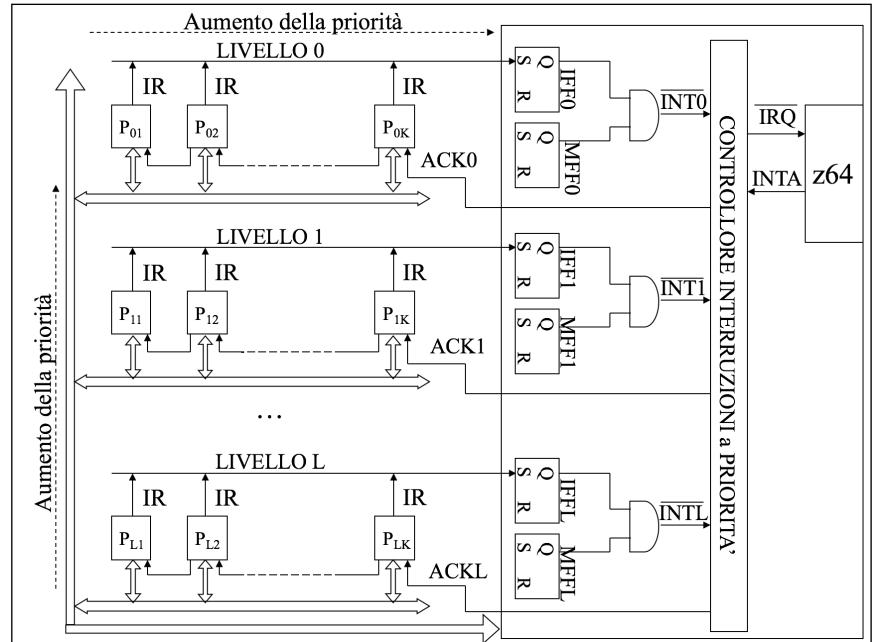
Periferica riceve INTA ed invia **IDN** (interrupt descriptor number) a CPU.
CPU tramite IDN identifica il driver della periferica contenuto nella **IDT³** (interrupt descriptor table), ogni dispositivo conosce solo il proprio IDN.



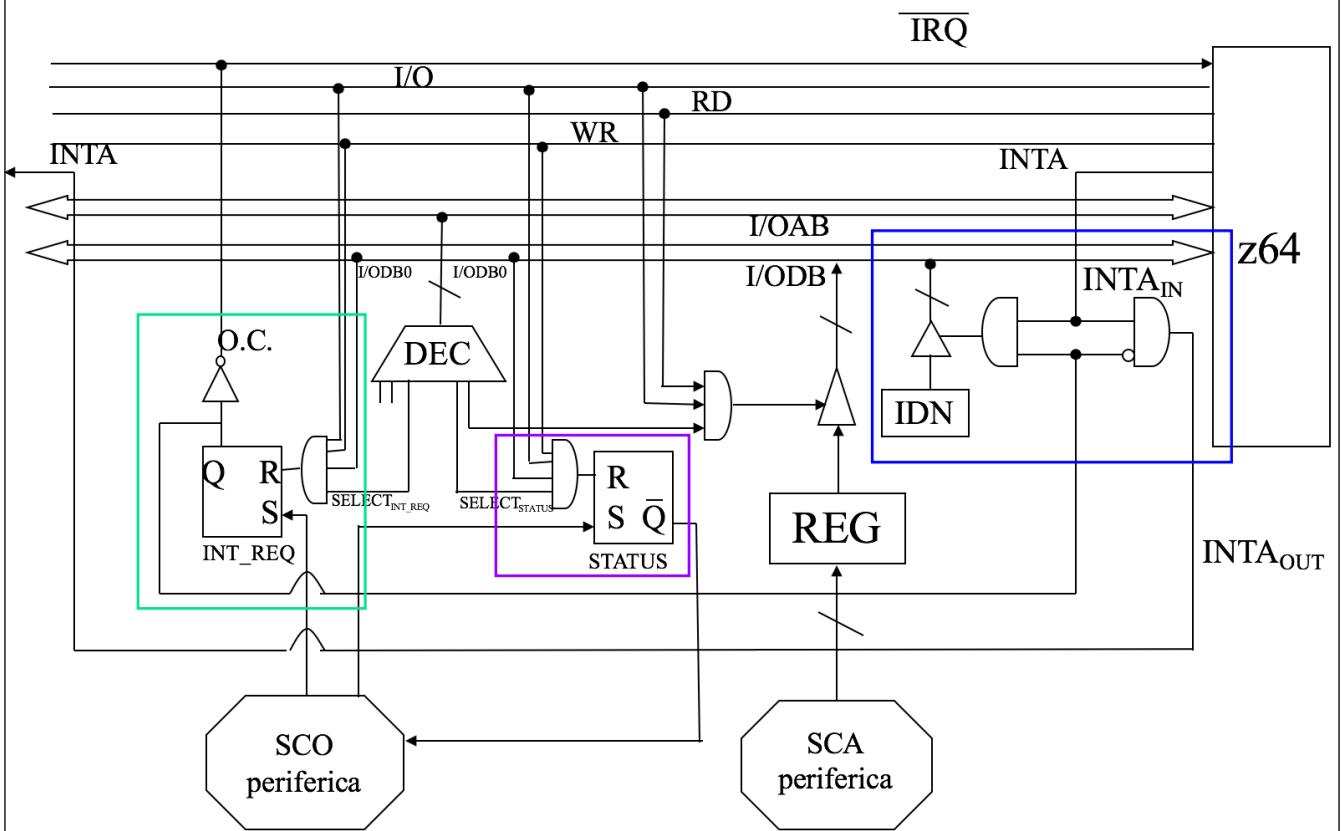
4. **Context switch** => processore salva registri su stack del processo in esecuzione ed esegue le altre operazioni per il cambio di contesto
ATTENZIONE importare disattivare interrupt flag durante questa fase
5. **Esecuzione driver** => finite le operazioni preliminari il processore esegue il driver del dispositivo, alla fine delle operazioni ripristina lo stato precedente tramite **IRET**.
OSS se vengono effettuate più richieste queste vengono messe in coda e servite una dopo l'altra prima di ritornare al normale flusso di programma

³ IDT = è una locazione di memoria (array) visibile solo al kernel nel quale vengono salvati tutti i puntatori ai driver dei dispositivi collegati all'elaboratore

EXTRA le operazioni di determinate periferiche possono essere più importanti di altre, per questo è stato introdotto il concetto di gestione delle priorità. Le periferiche vengono suddivise in livelli (0 più importante), se due periferiche mandano la richiesta insieme viene servita prima quella con priorità più alta. Se sono allo stesso livello si usa una DAISY-CHAIN (vedere arbitraggio bus)

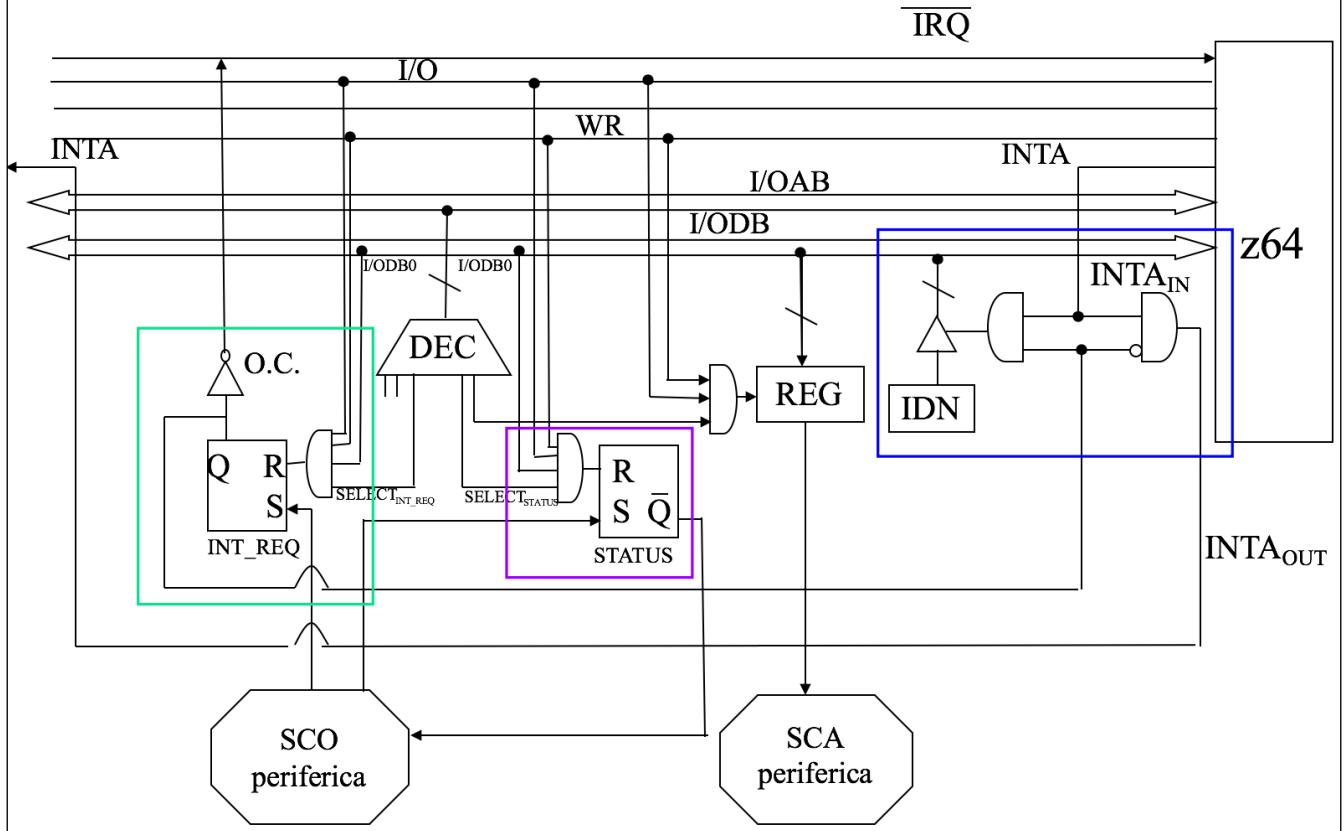


Interfaccia per *lettura* dati da periferica con interrupt



1. (**hardware viola**) CPU invia segnale WR (con eventuale I/O) e scrive 1 sul data bus, tramite AND con uscita del decoder setta il F/F status ed avvia la periferica.
Finita questa operazione ritorna al normale flusso d'esecuzione.
2. (**hardware verde**) in maniera asincrona rispetto alle operazioni del processore, la SCO della periferica invia segnale IRQ tramite F/F INT_REQ I
3. La CPU alla fine del ciclo istruzione se IF = 1 invia INTA a periferiche
4. (**hardware blu**) periferica riceve segnale INTA ed invia IDN
5. Ricevuto IDN processore consulta IDT, esegue context switch ed avvia il driver corrispondente che può essere implementato in diversi modi
6. Driver lancia IRET e processore ritorna al normale flusso d'esecuzione

Interfaccia per *Scrittura* dati su periferica con interrupt



1. Uguale a lettura, cambia solo il driver

DMAC

Il DMAC è un processore dedicato per il trasferimento di dati tra memorie e/o periferiche. Per effettuare i trasferimenti necessita programmazione delle sue componenti:

- Direzione di trasferimento, da o verso memoria
- Indirizzo iniziale della memoria, salvato in **CAR** (current Address register)
- Formato dati e lunghezza file, salvata in **WC** (word counter)
- modalità di acquisizione dei dati, **BURST⁴** o **BUS-STEALING⁵**
- Indirizzo della periferica

Finita la programmazione si avvia tramite il classico F/F STATUS

A differenza delle modalità viste in precedenza il DMAC può effettuare le trasmissioni ad ogni ciclo macchina invece di attendere la conclusione di un ciclo istruzioni.

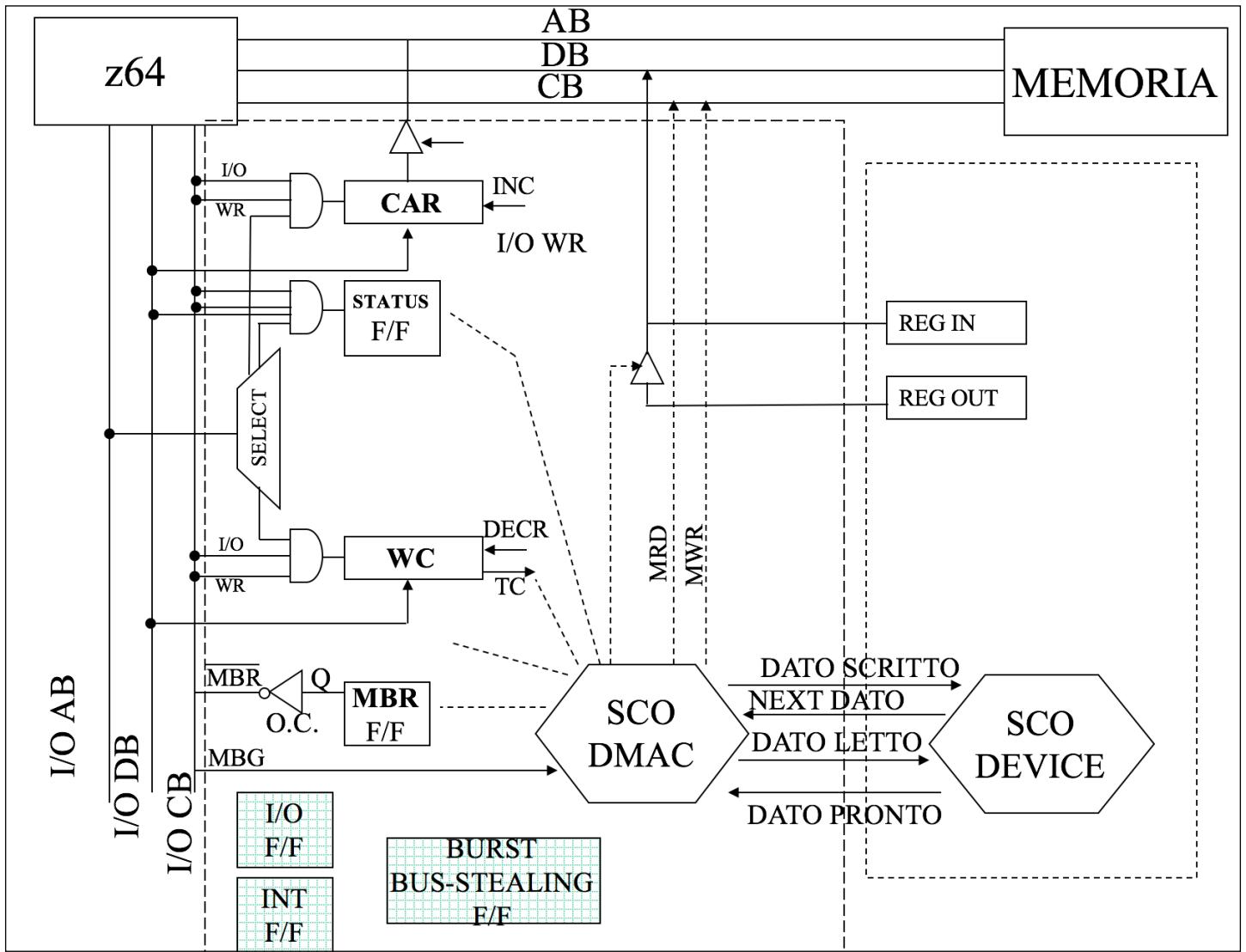
Avendo comunque un bus comune ad ogni interazione il DMAC avverte il processore inviando un **MBR** (memory bus request), se il processore è pronto a ricevere invia in risposta un **MBG** (memory bus grant) e mette in alta impedenza le sue uscite ad eccezione dei segnali di controllo.

OSS è visto dal processore come una normale periferica al quale nasconde tutti i dispositivi per i quali agisce da tramite.

Le caratteristiche interne delle varie periferiche non variano, tranne per il F/F status che diventa in comune per tutti i dispositivi di I/O collegati ad esso.

⁴ BURST => DMAC prende il controllo dei bus per tutto il tempo necessario per il trasferimento con conseguente blocco della CPU, equivalente a Busy waiting (utile per cache miss)

⁵ BUS-STEALING => DMAC suddivide dati in blocchi inviandoli in più brevi intervalli



1. Vedere slides macchina a stati finiti

CODICE ASSEMBLY: (nella realtà operazioni eseguite direttamente da ins / out)

```

movw $WC, %dx          #inizializza WC a $NUM_DATI
movl $NUM_DATI, %eax
outl %eax, %dx

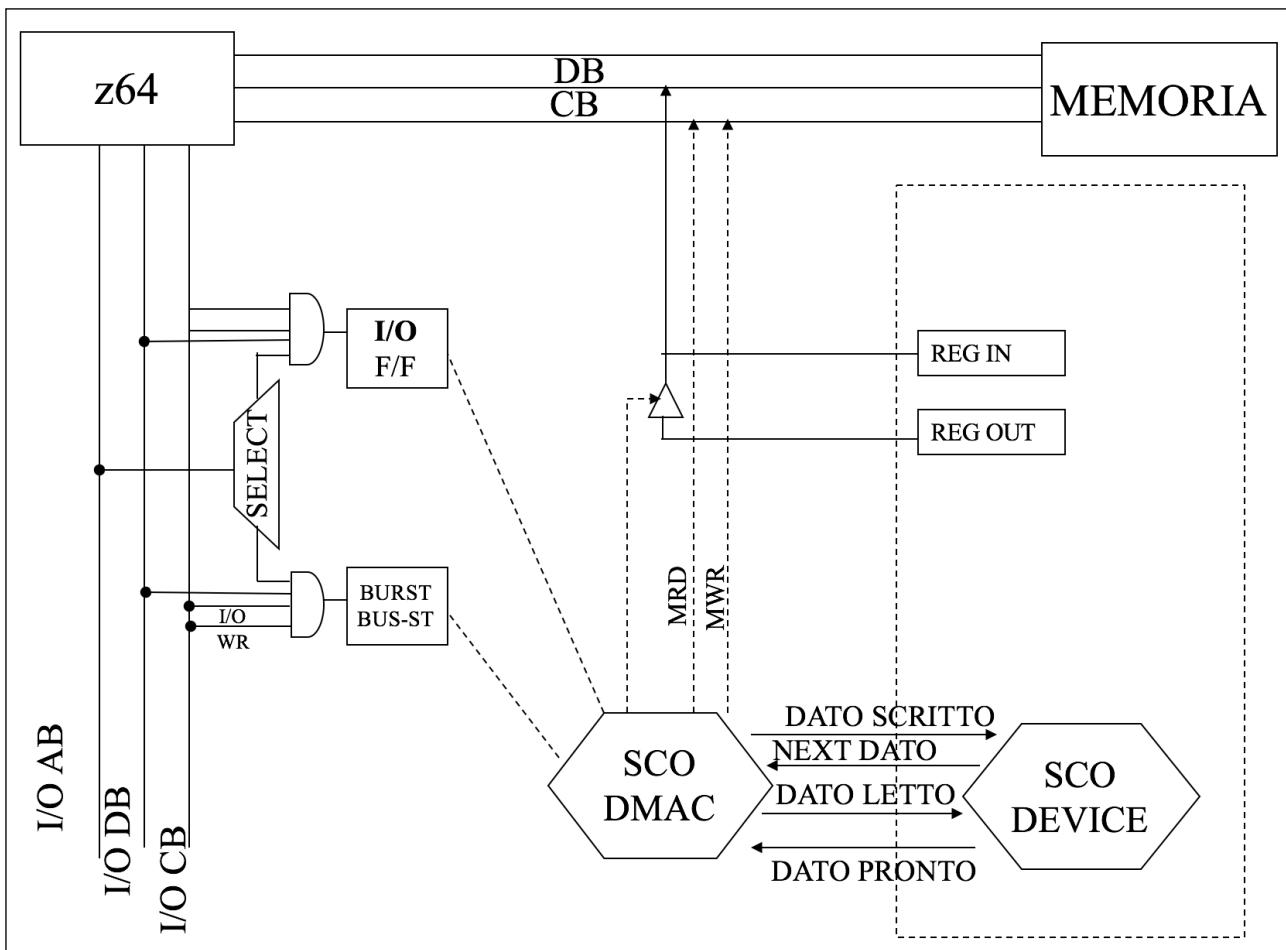
movw $CAR, %dx          #inizializza CAR a $ID_PERIFERICA
movl $ID_PERIFERICA, %eax
outl %eax, %dx

movw $DMACI/O, %dx      #setta F/F I/O per la scrittura (1)
movl $1, %eax
outl %eax, %dx

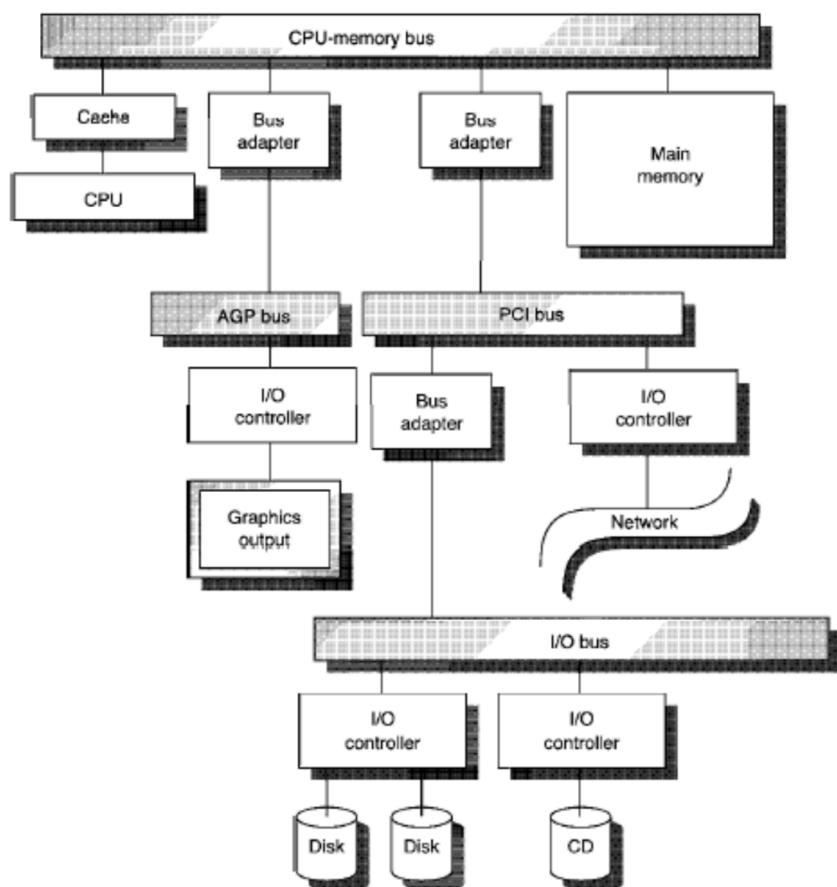
movw $DMACB-ST, %dx      #setta F/F BURST BUS-ST
movl $0, %eax            #per lavorare in burst ()
outl %eax, %dx

movw $DMAC_STATUS, %dx    #avvia il DMAC
movl $1, %eax
outl %eax, %dx

```



Architettura di un elaboratore moderno



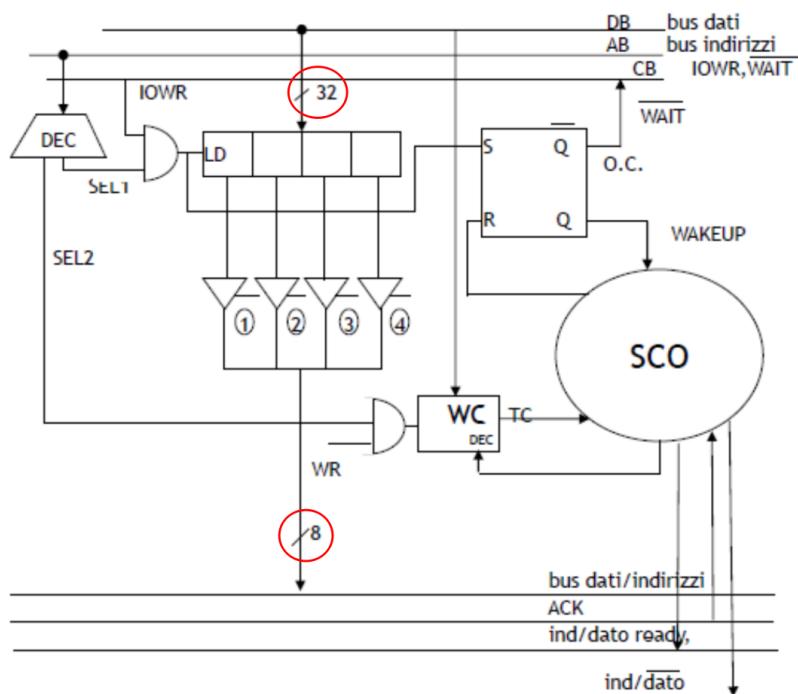
Adattatori di BUS

Gli adattatori consentono la trasmissione tra tipologie differenti di BUS, permettono quindi di ricevere dati secondo un protocollo e rinviarli utilizzando un nuovo protocollo.

OSS nel caso in cui i due bus lavorino a velocità differenti una volta connessi saranno entrambi costretti a lavorare alla velocità del più lento.

Per evitare questo rallentamento è possibile utilizzare lo “**store and forward**”, ovvero è possibile integrare un buffer nell’adattatore in modo che immagazzini i dati e li invii al buffer più lento utilizzando la sua velocità senza rallentare l’altro.

ex. Da bus sincrono a bus asincrono



SEL1 => indirizzo del registro a 32 bit per i dati
SEL2 => indirizzo del WC

F/F status => funge da semaforo, invia segnale WAIT ogni volta che il buffer riceve un dato e lo disabilita appena la periferica ha finito di leggere tutte e 4 le componenti del "messaggio"

protocollo:

1. Primo pacchetto (32 bit) indica numero di parole del messaggio, salvato in WC
 2. Secondo pacchetto (32 bit) viene inviato nel buffer e contiene identificativo della periferica connessa al bus asincrono
 3. Avendo che il bus asincrono è da 8 bit vengono selezionati solo gli 8 bit meno significativi (tramite segnale “4”) e vengono inviati sul bus per selezionare la periferica
 4. Finiti i passaggi preliminari il terzo pacchetto contiene i dati che vengono divisi in blocchi di 8 bit selezionati dal SCO tramite i segnali “1”, “2”, “3”, “4”.
 5. Una volta letti tutti i componenti del buffer WC viene decrementato di 1 e viene disabilitato segnale WAIT, in questo modo il buffer viene ricaricato con un nuovo dato e si ripete il passo 4
 6. Quando WC arriva a 0 il ciclo termina
-

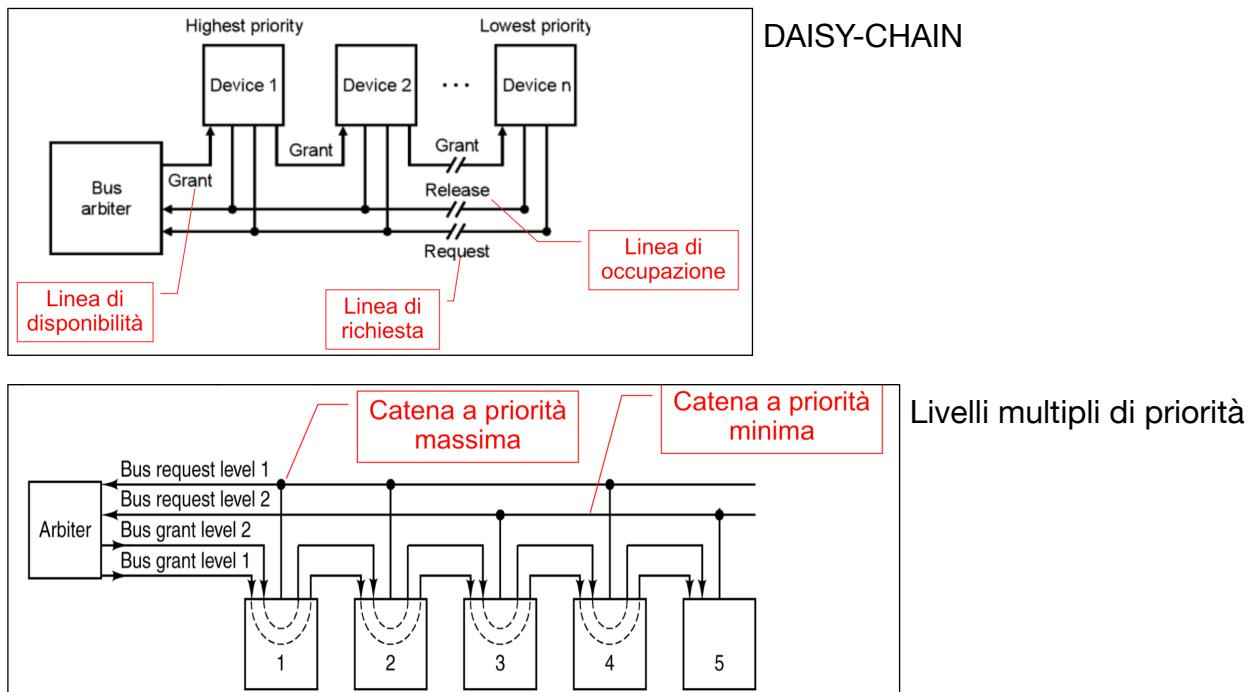
Arbitraggio dei bus

Serve per decidere quale sarà il prossimo dispositivo autorizzato ad usare il bus evitando situazioni di attesa indefinita o paralisi del sistema, esistono 2 tipologie:

Centralizzati

Un controllore decide a chi assegnare il bus

- **DAISY CHAIN** => sceglie il dispositivo con la priorità maggiore, ovvero quello più vicino al dispositivo d'arbitrio, non garantisce la fairness⁶
- **Livelli multipli di priorità** => le linee sono divise in livelli di priorità, in caso di conflitto vince il dispositivo nella linea con priorità più alta



Distribuiti

I dispositivi seguono un algoritmo per il controllo dell'acceso e cooperano per la condivisione del bus

- **Round-robin** => disciplina circolare, scambio ciclico di segnale di disponibilità tra gli utilizzatori
- **Rilevamento delle collisioni** => esiste una linea che indica lo stato libero / occupato del bus, se due dispositivi occupano il bus insieme bisogna rilevare la collisione ed effettuare una nuova trasmissione

⁶ fairness = tutti i dispositivi hanno la stessa importanza, non rispettare la fairness vuol dire favorire alcuni dispositivi rispetto ad altri

Banda passante di un BUS

Indica la frequenza con cui il bus passa dal trasferire un dato ad un altro, la banda teorica si ricava tramite: $\text{max banda} = \text{frequenza} * \text{numero linee} \text{ [MB/sec]}$

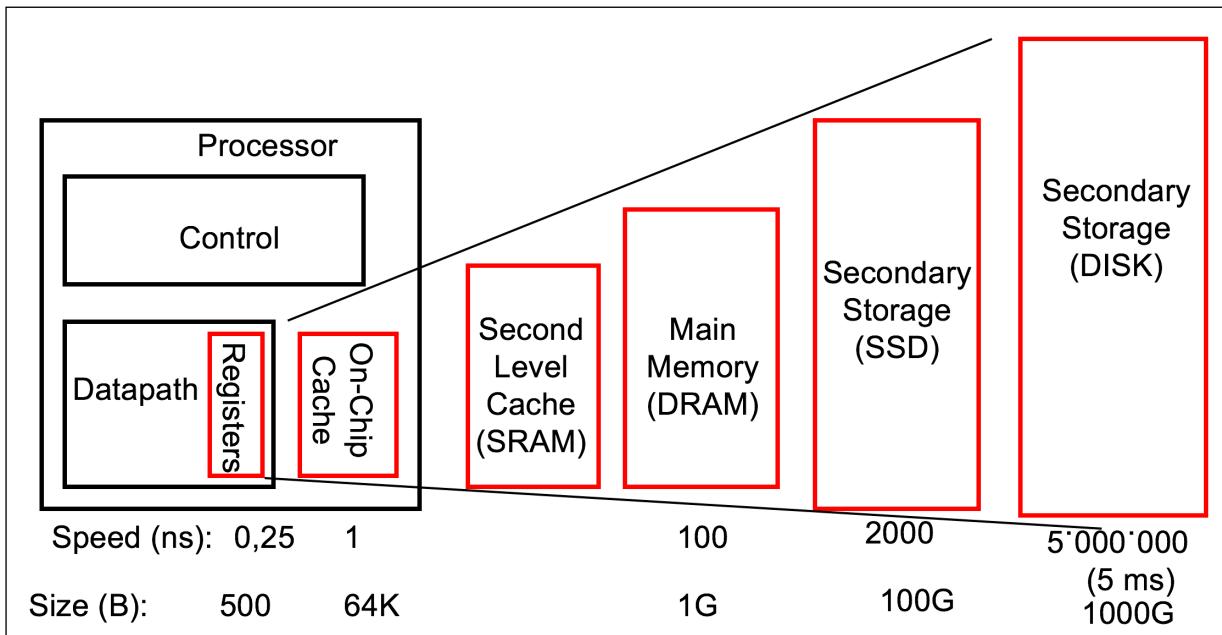
Limitazioni aumento della frequenza

- alte frequenze creano disturbi
- Ritardi nel segnale
- Bus skew (linee diverse che viaggiano a velocità differenti)

Tecniche per aumentare la banda passante

- parallelismo delle linee dati (aumentare numero linee)
- Separare linee dati e linee indirizzi
- Trasferimento di dati a blocchi (riduce tempo di risposta)
- Protocollo **split transaction** => si divide la transazione in transazione di richiesta e transazione di risposta, tra l'una e l'altra il bus viene rilasciato in modo da evitare periodi di inutilizzo del bus (implica 2 competizioni per il bus)

Gerarchie di memoria e cache



Ogni livello è basato su una tecnologia differente e contiene un sottoinsieme dei dati del livello inferiore, questo rende possibili ottimizzazioni grazie ai principi di località:

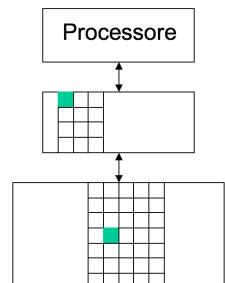
- **località temporale** => se accedo ad un elemento probabilmente vi accederò nuovamente (ex. ciclo)
- **località spaziale** => se accedo ad un elemento probabilmente accederò ad un elemento vicino ad esso (ex. array)

Quindi ad ogni accesso al livello inferiore copio un intero blocco dei dati (spaziale) e lo mantengo finché non è necessario riutilizzare lo spazio che occupa (temporale).

Migrazione delle informazioni

Ad ogni accesso ad una gerarchia si possono avere:

- **Hit** => informazione richiesta presente nel livello acceduto
- **Miss** => informazione assente, bisogna accedere a livello inferiore



definizioni:

- Hit rate => numero hit / numero accessi
- Miss rate => 1 - hit rate
- Hit time => tempo di accesso in caso di hit
- Miss penalty => tempo di trasferimento blocco da gerarchia inferiore a superiore
- Miss time => miss penalty + hit time (tempo per ottenere elemento in caso di miss)
- AMAT (tempo medio di accesso) => $c + (1 - h) * m$
c = hit time, h = miss rate, m = miss penalty

Struttura di una cache



Ogni entry (posizione) di una cache contiene:

- Valid bit => indica se la entry contiene informazioni (all'avvio tutti 0)
- Tag (etichetta) => identifica univocamente l'indirizzo in memoria corrispondente
- Campo dati => contiene la copia del blocco di dati

OSS numero di blocchi di una cache = dimensione cache / dimensione blocco

Cache a indirizzamento diretto

Ha la struttura di una normale memoria, ogni blocco al suo interno è accessibile direttamente tramite indirizzamento ed ha una posizione univoca nella cache.

Avendo però che la cache è più piccola della memoria si utilizza solo una parte dell'indirizzo (che forma l'index), quindi nel caso di elementi con bit meno significativi uguali (il tag) è richiesta comunque la sostituzione della linea con lo stesso indice anche se ci sono altre linee ancora libere.

Vantaggi:

- Veloce
- Richiede poco spazio
- Semplice da implementare

Svantaggi:

- Non si ha località temporale

INDIRIZZO			DATO
TAG	INDEX		
00	00	SOLE	
00	01	MARE	
00	10	CASA	
00	11	LUCE	
01	00	ABCD	
01	01	ROMA	
01	10	BARI	
01	11	ANNA	
10	00	SALE	
10	01	COMO	
10	10	PARI	
10	11	PEPE	
11	00	MANO	
11	01	BIRO	
11	10	DUCA	
11	11	SARA	

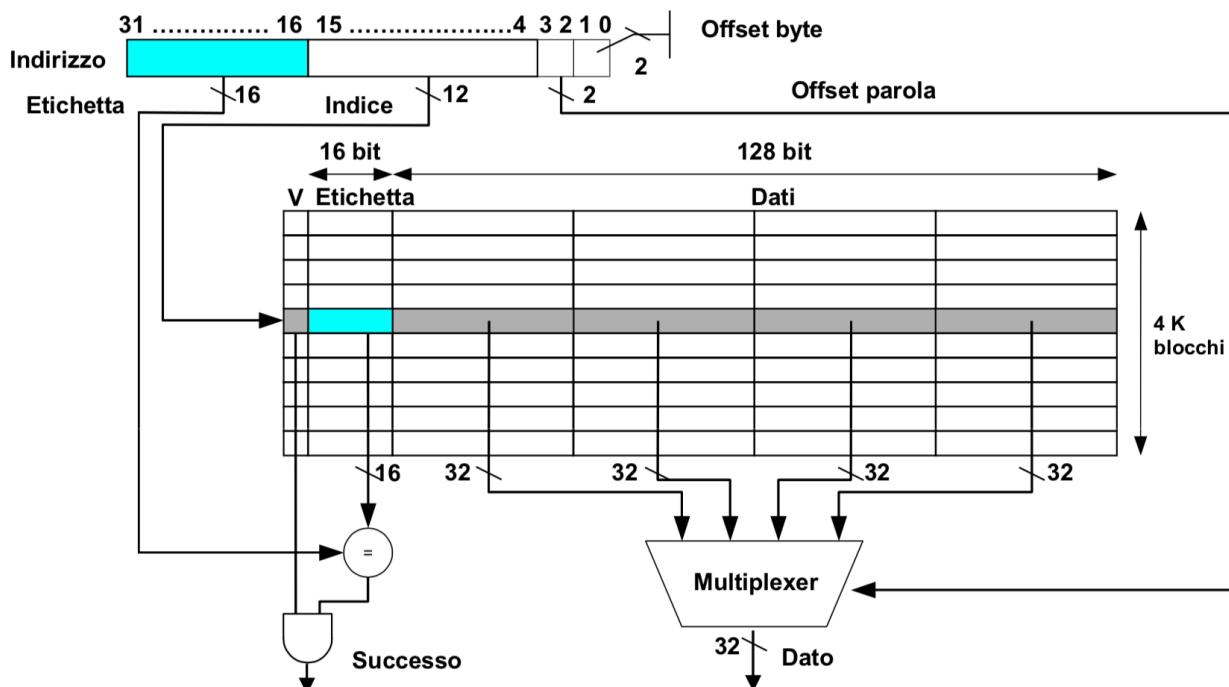
indirizzamento:

- sp. Byte => spiazzamento del byte nella parola
- sp. Parola => spiazzamento della parola nel blocco
- Indice => identifica il blocco
- Etichetta => identificativo corrispondente al tag nella struttura della cache

Etichetta	Indice	Sp. parola	Sp. byte

- Indirizzo in memoria = 32 bit
- Cache da 64 kb e blocco 128 bit (4 parole da 32 bit = 16 byte)
- Numero blocchi = 64kb / 16 byte = 4k blocchi
- Struttura => sp. Byte = 2 bit
sp. Parola = 2 bit
indice = 12
etichetta = 16 (indirizzo - (sp. Byte + sp. Parola + indice))

ex.



Cache completamente associativa

Permette di salvare i blocchi in una qualsiasi posizione della cache utilizzando l'indirizzo completo in memoria e non solo una parte, non avendo posizioni prestabilite ad ogni accesso bisogna esaminare tutte le linee per trovare quella cercata.

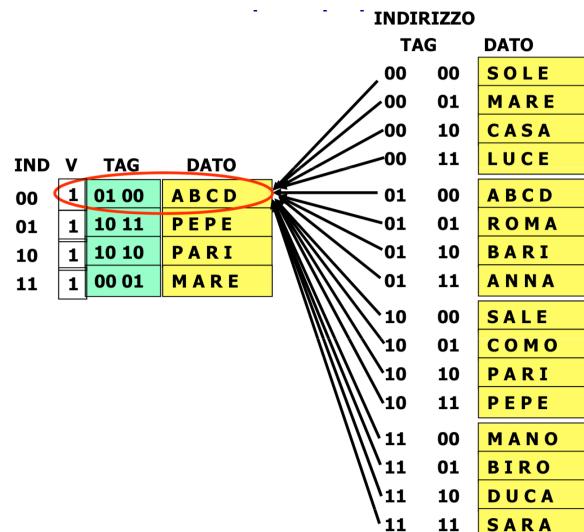
Il tag è quindi composto da tutto l'indirizzo.

Vantaggi:

- Si ha località temporale
- Non ha problemi di conflict miss⁷

Svantaggi:

- hardware più complesso
- possibilità di salvare meno blocchi nella stessa cache (causa tag più lunghi)

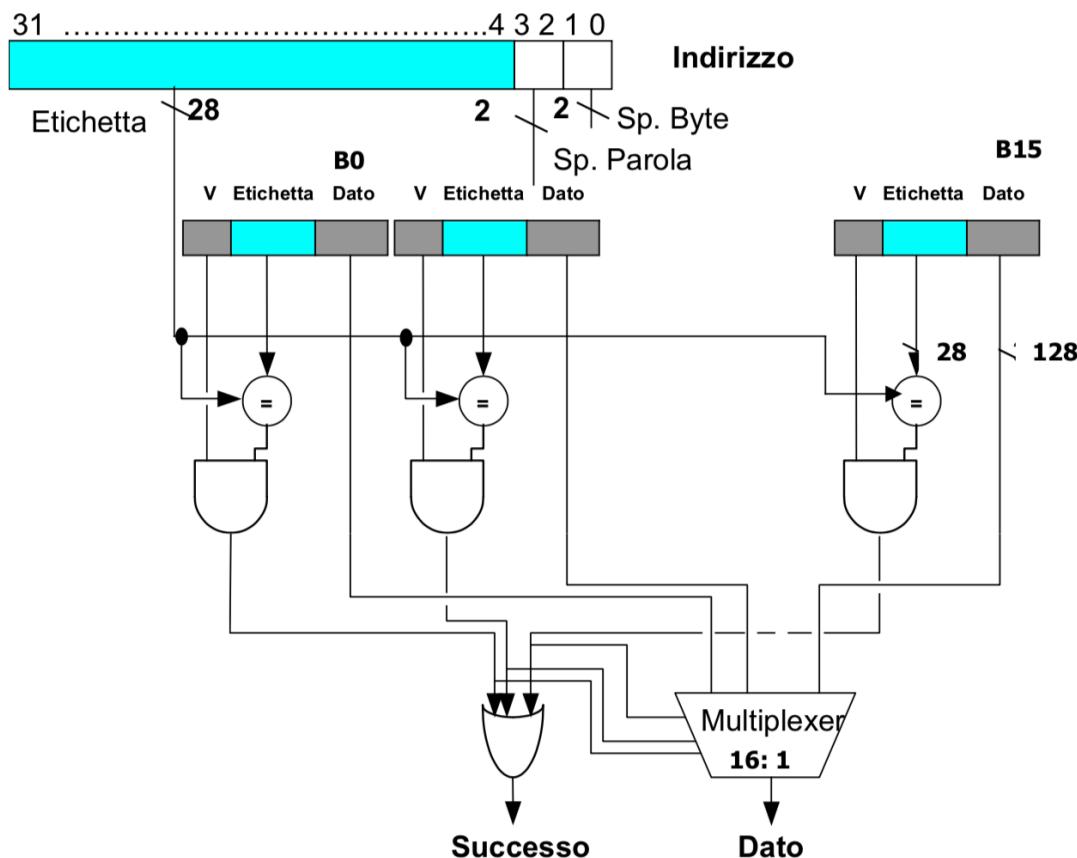


Indirizzamento:

- sp. Byte => spiazzamento del byte nella parola
- sp. Parola => spiazzamento della parola nel blocco
- Etichetta => indirizzo completo

Cache da 256 byte e blocco da 128 bit

ex.



⁷ conflict miss = miss dovuti al conflitto sugli indirizzi nelle cache a indirizzamento diretto

Cache set-associative ad N vie

Versione intermedia, basata su una cache ad indirizzamento diretto divisa in N vie parallele => *numero blocchi in una via = numero blocchi / N*.

Il blocco può quindi essere posizionato in una qualsiasi delle N vie rispettando però l'indirizzamento dato da index.

La ricerca si effettua accedendo contemporaneamente allo stesso indirizzo in ogni via, gli N blocchi letti formano un SET (insieme)
 $Set = \text{dim. cache} / (\text{dim. blocco} * n)$

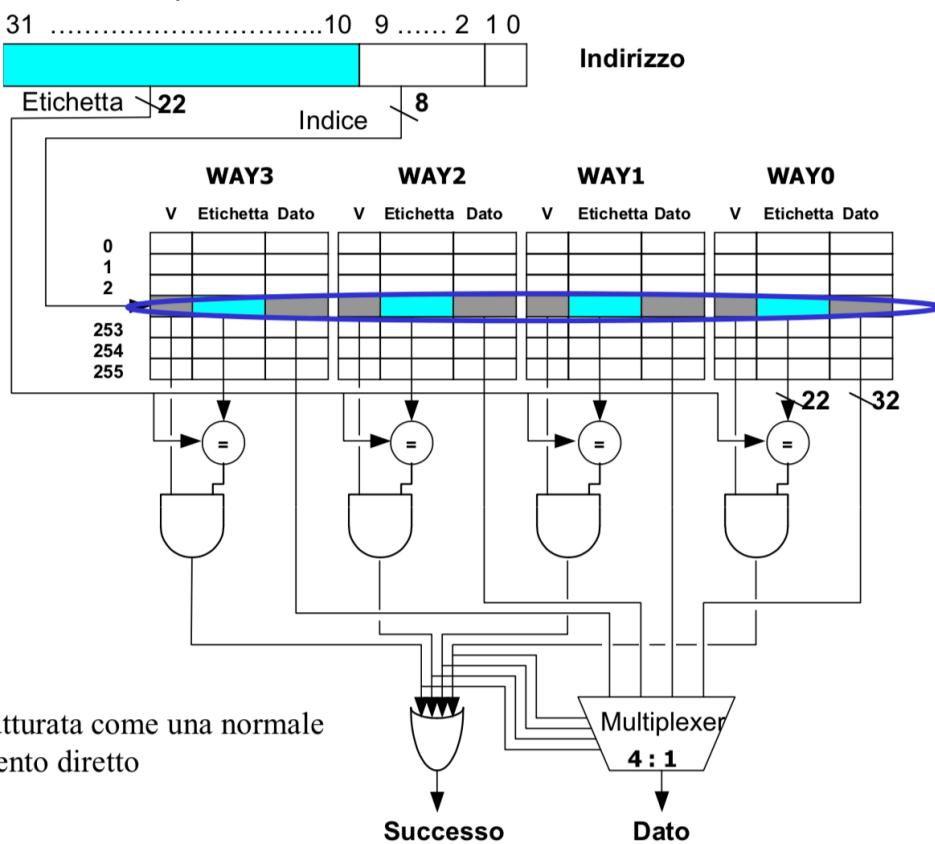
Politiche di sostituzione blocchi:

- sostituzione random
- **LRU** (last recently used) =>
viene associato contatore ad ogni blocco incrementato ad ogni accesso ad altro blocco, si sostituisce blocco con numero contatore più alto (costoso)
- **FIFO** (first in first out) =>
primo blocco ad entrare nella cache è primo ad uscire (poco costoso e molto efficiente)

INDIRIZZO		TAG	INDEX	DATO
000	0	SOLE		
000	1	MARE		
001	0	CASA		
001	1	LUCE		
010	0	ABCD		
010	1	ROMA		
011	0	BARI		
011	1	ANNA		
100	0	SALE		
100	1	COMO		
101	0	PARI		
101	1	PEPE		
110	0	MANO		
110	1	BIRO		
111	0	DUCA		
111	1	SARA		

- Indirizzo in memoria = 32 bit
- Cache da 4 kb a N = 4 vie e blocco 32 bit (1 parola = 4 byte)
- Numero blocchi = 4kb / 4 byte = 1k blocchi
- Numero di set = 4 kb / (4 byte * 4) = 2^8 insiemi
- Struttura => sp. Byte = 2 bit, sp. Parola = 0, bit indice = 8, etichetta = 16

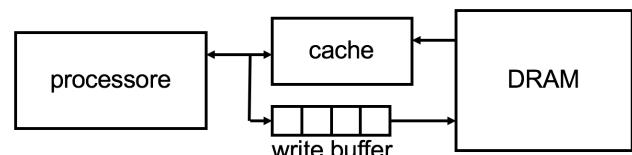
ex.



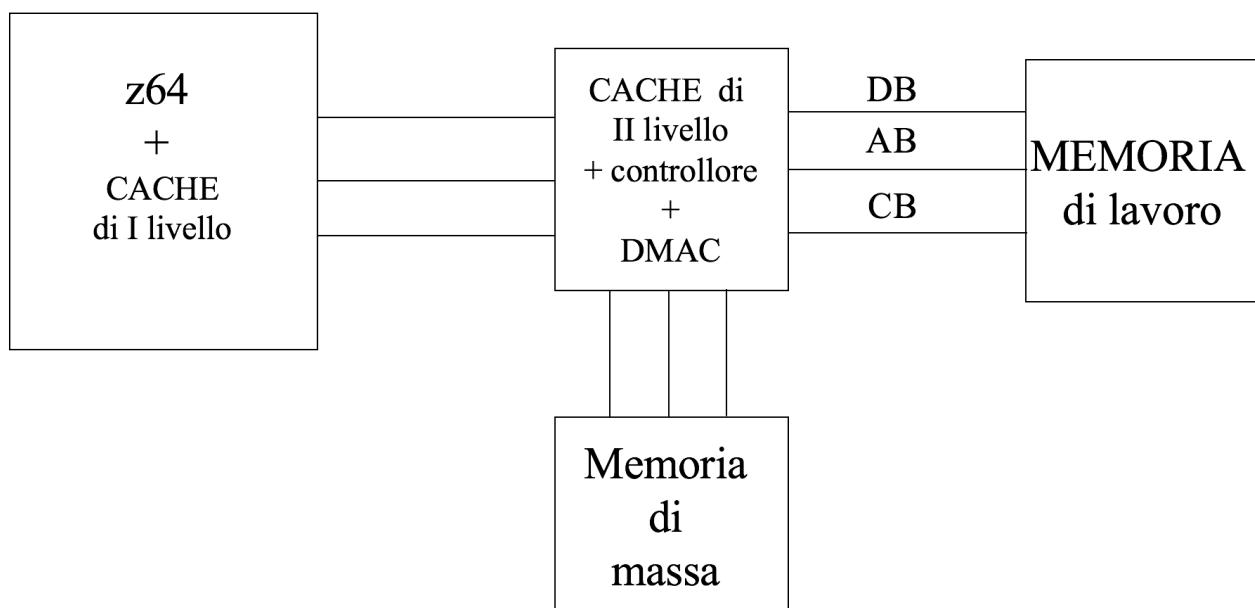
Strategie di scrittura su cache

In caso lettura basta accedere a cache e scaricare dati, in caso di scrittura nasce il problema della coerenza dei dati tra cache e gerarchia inferiore, risolvibile con:

- **Write-through** => immediata, scrittura contemporanea su cache e livello inferiore.
Vantaggi: semplice da implementare e mantiene coerenza
Svantaggi: operazioni di scrittura alla velocità di livello inferiore
- **Write-back** => dati salvati solo su cache, salvati su gerarchi superiore solo in caso di sostituzione del blocco.
Vantaggi: scrittura veloce su cache, unica scrittura su livello inferiore
Svantaggi: sostituzione blocco lenta e niente system recovery⁸
- **Write buffer** => scrittura contemporanea su cache e buffer, il buffer memorizza ed invia i dati a gerarchia inferiore contemporaneamente alle normali operazioni della CPU.
Le velocità non possono essere troppo diverse o si ha saturazione del buffer.



Schema di principio di una cache



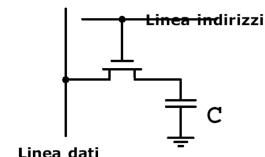
Nel caso di cache ad indirizzamento diretto gli indirizzi di cache level 1 e cache level 2 differiscono dal numero di bit usati per tag e index.

OSS la cache level 2 essendo più grande usa più bit per index

RAM (random access memory)

- SRAM (static ram) => basata su F/F, veloce ma costosa ed a bassa densità.
tecnologia complicata che permette la permanenza dei dati, costituita da 6 transistor mos.
- DRAM (dinamic ram) => basata su immagazzinamento di cariche elettriche, più lenta ma meno costosa e ad alta densità.
costituita da un condensatore, carico = 1 e scarico = 0.
richiede di rinfrescare la carica periodicamente

OSS SDRAM => synchronous ram, segnale di clock su fronte attivo



⁸ system recovery = doppio salvataggio dei dati per evitare perdite

Architetture RISC

Famiglie istruzioni CPU

- CISC (complex instruction set architecture) => set istruzioni più ampio e con formato variabile di byte, ha lo svantaggio che le singole istruzioni sono più lente (x86)
- RISC (simple instruction set architecture) => set istruzioni ristretto ma singole istruzioni più veloci, nel caso di operazioni più complesse implica l'utilizzo di più istruzioni (arm)

Set istruzioni processore RISC didattico a 32 bit

- Istruzioni logico/aritmetiche

Istruzione	Sintassi	Semantica
Somma	add regsorg1, regsorg2, regdest	(regdest) = (regsorg1) + (regsorg2)
Sottrazione	sub regsorg1, regsorg2, regdest	(regdest) = (regsorg1) - (regsorg2)
Prod. Logico	and regsorg1, regsorg2, regdest	(regdest) = (regsorg1) and (regsorg2)
Som. Logica	or regsorg1, regsorg2, regdest	(regdest) = (regsorg1) or (regsorg2)
Neg. logica	not regsorg1, regdest	(regdest) = not (regsorg1)

opcode	regsorg1	regsorg2	regdestL/A	non utilizzati
31-26	25-21	20-16	15-11	10-0

Ex. Somma registro 2 (00010) con registro 3 (00011) e metti risultato in 1 (00001)

000001	00010	00011	00001	-----
--------	-------	-------	-------	-------

- Caricamento memorizzazione (load / store)

Istruzione	Sintassi	Semantica
Caricamento di parola	load regdest, offset(regbase)	(regdest) = memoria[offset+(regbase)]
Memorizzazione di Parola	store regsorgM, offset(regbase)	memoria[offset+(regbase)] = (regsorg)

<i>Formato dell'istruzione load</i>			
opcode	regbase	regdestC	Offset
31-26	25-21	20-16	15-0
<i>Formato dell'istruzione store</i>			
opcode	regbase	regsorgM	Offset
31-26	25-21	20-16	15-0

Ex. (store) Trasferire registro 7 (00111) nella locazione di memoria con indirizzo di memoria dato da contenuto registro 4 (00100) + 16 (00...0010000)

000111	00100	00111	00000000000010000
--------	-------	-------	-------------------

- Salto condizionato

Istruzione	Sintassi	Semantica
salta se flag X ==1 (dove X può essere uno dei flag del registro di stato)	jumpX indirizzo	se flag X ==1 allora PC=PC+S+indirizzo (dove S è un intero che dipende da come è organizzata la memoria, per esempio in un processore a 32 bit con il banco di memoria della cache costituita con moduli di memoria di 8 bit il suo valore è pari a 4)

opcode	flag	Indirizzo
31-26	25-23	22-0

Ex. IF (carry == 1) => salva in PC il valore (PC) + S + 1026

001000	001	00000000000000001000000010
--------	-----	----------------------------

- NOP (istruzione non operativa)

Istruzione	Sintassi	Semantica
Nulla	Nop	Non modificare nulla

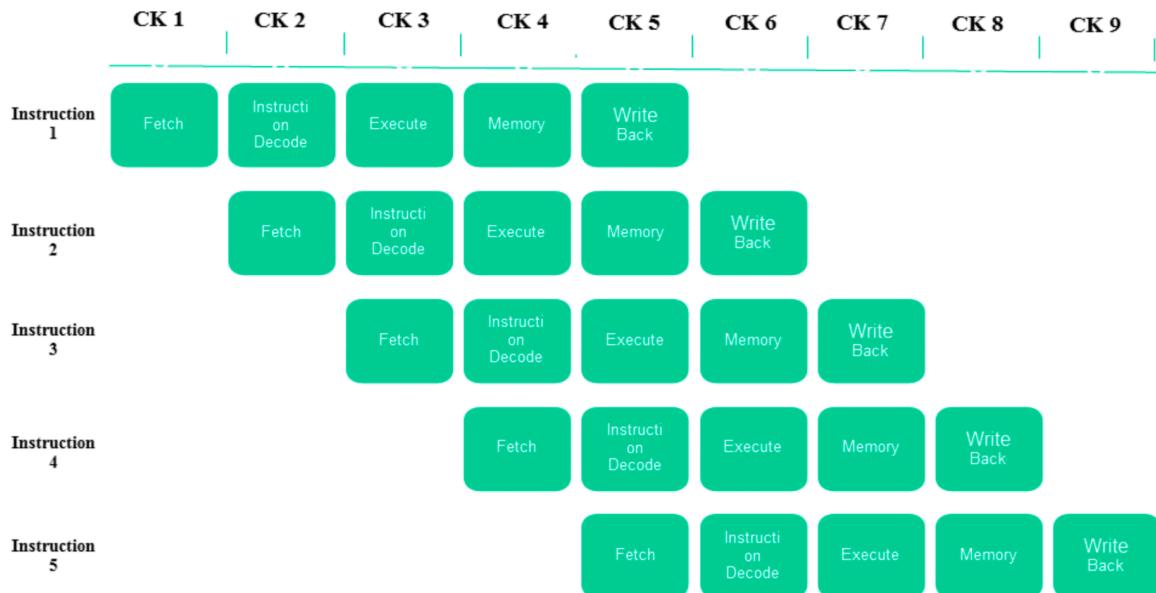
opcode			
31-26		25-0	

EX. Non fare nulla

000000	0000	0000	00000000000000000000
--------	------	------	----------------------

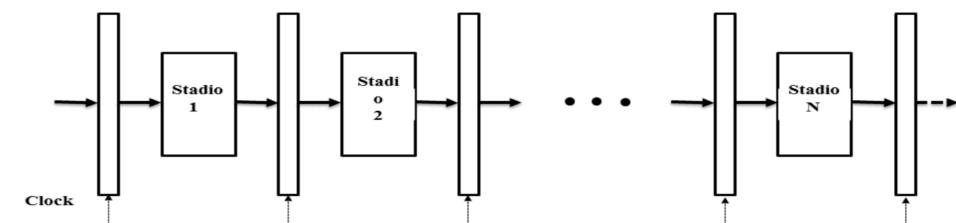
Pipelining

Basata sul presupposto che per l'esecuzione di ogni istruzione il processore deve passare per più stadi diversi ed ognuno di essi richiede componenti diversi, permette di eseguire più istruzioni contemporaneamente associando ogni istruzione ad un diverso stadio della struttura in ogni momento.



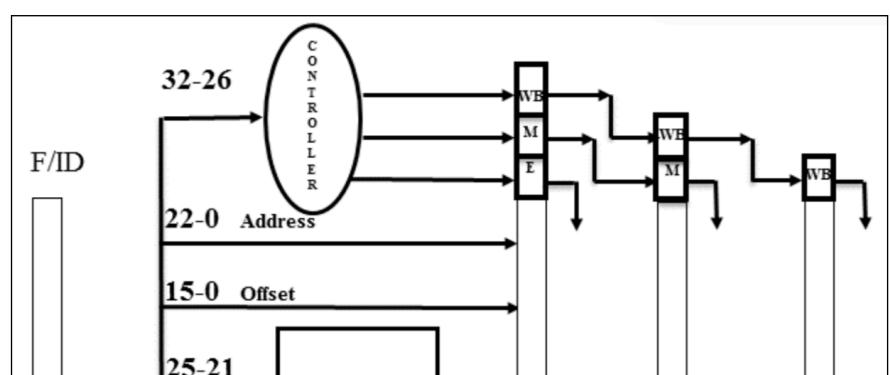
- fetch => si preleva l'istruzione, uguale per ogni tipo di istruzione
- Decode => decodifica istruzione e prelievo registri
- Execute => si effettuano operazioni logico/aritmetiche (calcolo indirizzi load/store/jump)
- Memory access => accesso a memoria per prelevare o memorizzare dati
- Write back => si memorizzano risultati in registri destinazione

OSS Per parallelizzare le operazioni è necessario contrapporre tra ogni stadio dei registri di pipeline che salvino i dati e una volta completate tutte le istruzioni su ogni stadio li trasmettono a quello successivo.

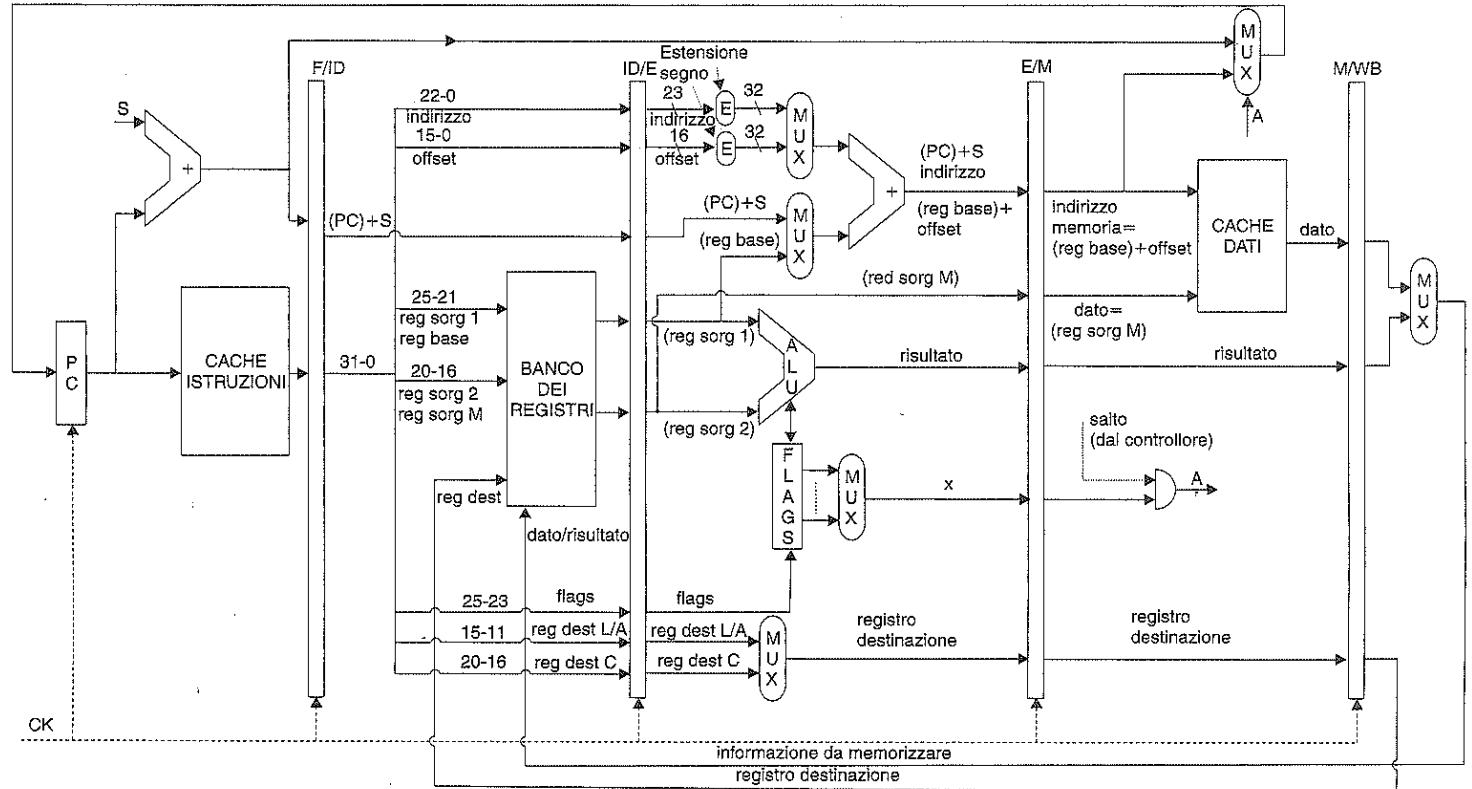


OSS dallo stadio decode oltre al passaggio dei dati i registri si occupano anche del passaggio delle informazioni di controllo per l'esecuzione delle operazioni.

Non servono segnali esplicativi per le scritture poiché si utilizza il segnale di clock principale



Architettura del processore didattico e segnali di controllo



- Stadio execute:

- M1 => pilota primo mux per operando ALU tra memoria e offset
- M2 => pilota secondo mux per operando ALU tra reg_base e PC+...
- M3 => pilota terzo mux seleziona tra reg_dest_L/A e reg_dest_C
- ALU_OPCODE => opocode per operazioni ALU

- Stadio memory:

- DRC => abilita lettura dato in memoria
- DCW => abilita scrittura dato in memoria
- JMP => consente salto della sequenza

- Stadio write back:

- MWB => pilota mux per selezione tra dato e risultato
- RW => abilita scrittura su banco registri (reg selezionato da mux)

Instruction	OPCODE	M1 M2 M3 OP ₃ OP ₂ OP ₁	DCW DCR JMP	MWB RW
ADD	000001	- - 0 0 0 1	0 - 0	1 1
SUB	000010	- - 0 0 1 0	0 - 0	1 1
OR	000011	- - 0 0 1 1	0 - 0	1 1
AND	000100	- - 0 1 0 0	0 - 0	1 1
NOT	000101	- - 0 1 0 1	0 - 0	1 1
LOAD	000110	1 1 1 - - -	0 1 0	0 1
STORE	000111	1 1 - - - -	1 0 0	- 0
JMP	001000	0 0 - - - -	0 0 1	- 0
NOP	000000	- - - 0 0 0	0 - 0	- 0

Prestazioni pipeline

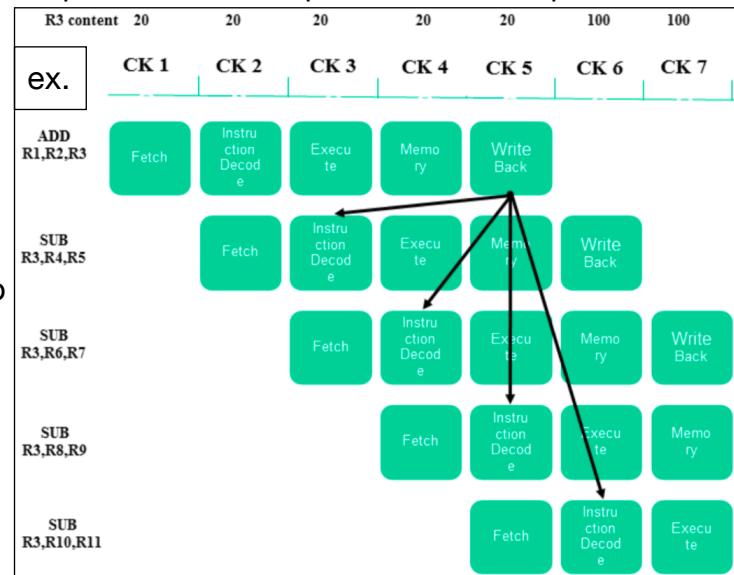
- vantaggi => potendo eseguire più istruzioni contemporaneamente aumenta il throughput⁹ ideale
- Svantaggi => il tempo di latenza delle singole istruzioni potrebbe aumentare dovendo sincronizzare tutti gli stadi (istruzioni che potrebbero essere eseguite più velocemente devono comunque viaggiare a velocità dell'istruzione più lenta)

Quindi si hanno vantaggi reali solo nel caso di molte istruzioni consecutive senza salti, nel caso ideale usando una pipeline con k stadi per completare n istruzioni occorrono $k + (n-1)$ cicli di Clock, nella realtà lo speedup è minore (soprattutto a causa delle criticità)

Criticità pipeline

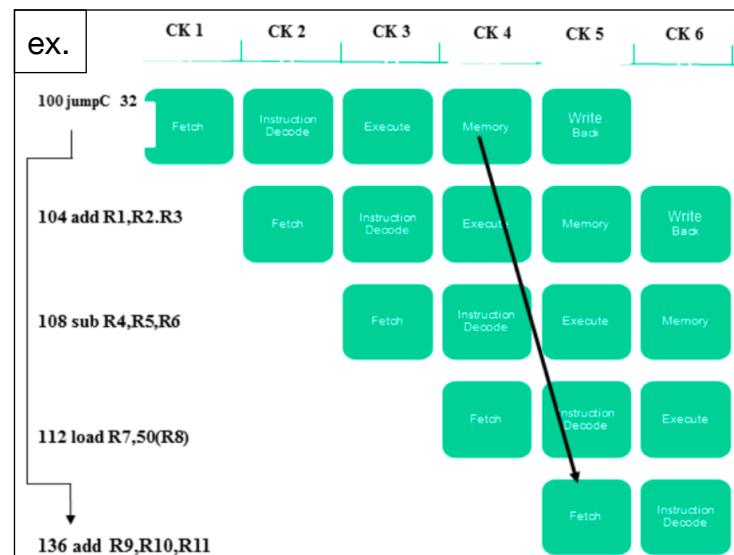
Sorgono quando non è possibile eseguire l'istruzione successiva senza cambiare la semantica del programma, possono essere:

- **strutturale** => tentativo di utilizzare la stessa risorsa hardware da due istruzioni differenti nello stesso momento (ex. Lettura e scrittura dello stesso registro, un'istruzione esegue decode sul registro e l'altra write back sullo stesso momento)
- **Sui dati** => tentativo di utilizzare un dato prima che sia disponibile, di due tipi:
 - **Define use** => istruzione usa risultato istruzione precedente
ex. add R1, R2, **R3**
sub **R3**, R4, R5
 - **Load use** => la prima istruzione utilizza una load su un registro, la seconda usa lo stesso registro come operando
ex. load **R3**, 122(R1)
sub **R3**, R5, R6



- **Sul controllo** => dati dai salti condizionali, siccome le istruzioni di salto verificano le condizioni solo durante la fase di memory intanto la pipeline si riempie di altre istruzioni che potrebbero andar perse.

OSS inoltre la pipeline soffre di problemi con la FPU e per la gestione di interrupt



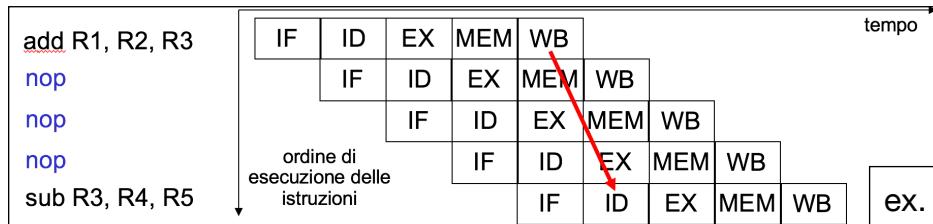
⁹ throughput = numero istruzioni eseguite per unità di tempo

$$136 = 100 + 32 + 4 \text{ (S)}$$

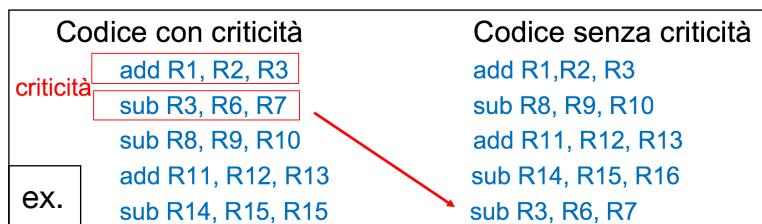
Soluzioni criticità sui dati

Software:

- **Inserire nop** => inserire una nop permette di rimandare l'utilizzo del registro fino a quando questo è nuovamente disponibile, evita il conflitto ma impone stalli



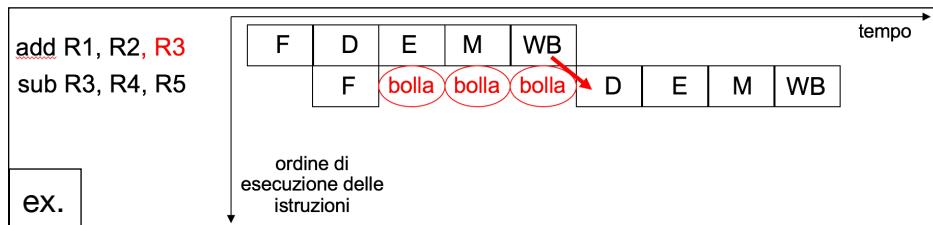
- **Riordinare le istruzioni** => programmatore o compilatore riordinano le istruzioni ponendo istruzioni indipendenti tra quelle che creano il conflitto, non crea stalli



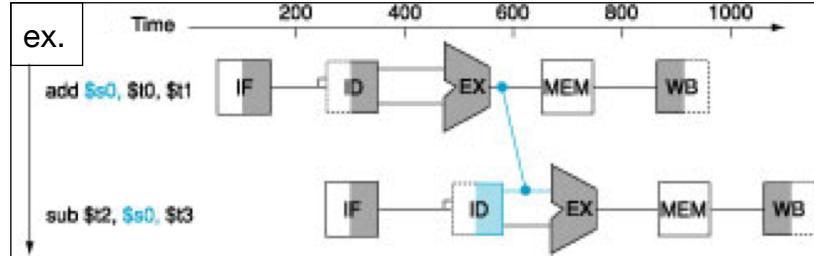
Hardware:

Implementate tramite l'**hazard detection unit**, un circuito combinatorio che effettua comparazioni sui registri ed implementa le soluzioni se rileva criticità.

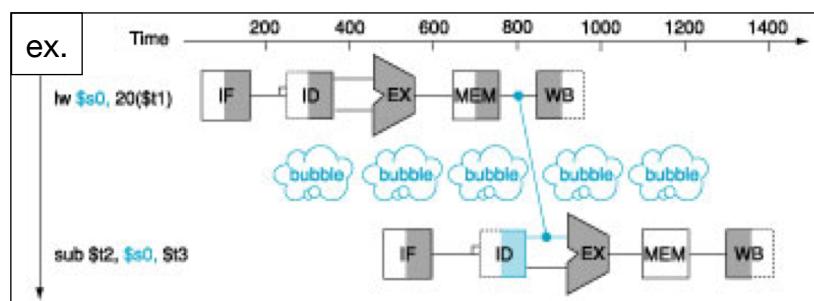
- **Inserimento di bubble (bolle)** => equivalente al nop, inserisce stalli tra le istruzioni



- **Forwarding (propagazione)** => si propagano i dati in avanti verso l'istruzione che lo richiede prima ancora di eseguire gli altri stadi che sarebbero necessari per renderla disponibile



ATTENZIONE nel caso di criticità load use bisogna usare anche stall



Soluzioni criticità sul controllo

- **Approccio pessimistico** => non si esegue nessuna istruzione fino al completamento della scelta, si effettua inserendo 3 nop (software) o 3 bolle (hardware)

Codice iniziale	Codice modificato
100 jumpC 32	100 jumpC 44
104 add R1,R2.R3	104 nop
108 sub R4,R5,R6	108 nop
112 load R7,50(R8)	112 nop
136 add R9,R10,R11	116 add R1,R2.R3 120 ub R4,R5,R6 124 load R7,50(R8) . . 148 add R9,R10,R11

100 jumpC 32	IF	ID	EX	MEM	WB						
104 add R1,R2.R3	bolla	bolla	bolla	IF	ID	EX	MEM	WB			
108 sub R4,R5,R6				IF	ID	EX	MEM	WB			
112 load R7,50(R8)					IF	ID	EX	MEM	WB		

- **Approccio ottimistico** => si ipotizza che il salto non venga effettuato e le istruzioni vengono eseguite comunque, si possono quindi avere 2 casi:
 - Non si doveva effettuare il salto => le istruzioni continuano la normale esecuzione
 - Si doveva effettuare il salto => i registri di pipeline vengono svuotati, si salta alla nuova istruzione e si ricomincia a riempire la pipeline

ATTENZIONE quindi in caso di salti diminuisce il throughput

OSS i processori implementano tecniche di predizione dei salti come la Tecnica bimodale, utile nel caso di loop, associa un contatore (ultimi 2 bit dell'istruzione) che indica quanto è consigliato effettuare il salto

Architetture di calcolo avanzate basate su pipelining

- **Superpipeline** => si incrementa il numero degli stadi che richiedono tempi d'esecuzione più lunghi in modo da aumentare la frequenza

Fetch	Instruction Decode	Execute	Execute	Execute	Memory	Write Back
-------	--------------------	---------	---------	---------	--------	------------

- **Superscalari** => si leggono ed seguono 2 istruzioni contemporaneamente, possibile solo alternando operazioni su ALU o salti con operazioni di load o store.

Fetch	Instruction Decode	Execute	Memory	Write Back
Fetch	Instruction Decode	Execute	Memory	Write Back