

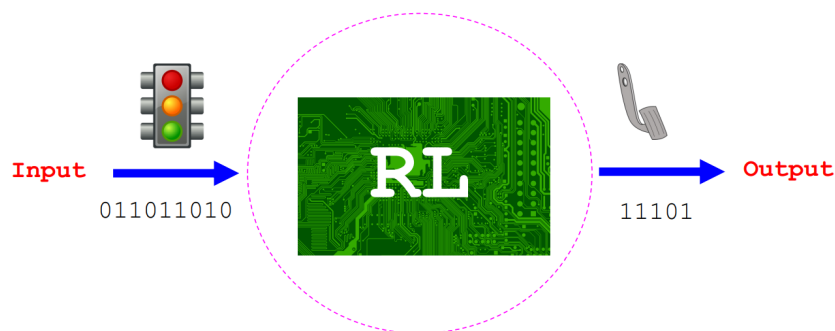
# Calcolatori elettronici

## ▼ 1.0 - Introduzione ai calcolatori elettronici

### Sistema di elaborazione

Un sistema di qualsiasi natura per l'elaborazione delle informazioni isolato dall'esterno, ovvero senza input e output, è inutile.

In questo corso siamo interessati a comprendere il funzionamento e la progettazione di **sistemi di natura elettronica per l'elaborazione delle informazioni**. Un qualunque sistema di elaborazione di questo tipo ha la seguente struttura:



Sistema di elaborazione (RL: rete logica).

### Informazione digitale vs analogica

L'informazione digitale e quella analogica sono due modi distinti di rappresentare e trasmettere dati:

L'**informazione digitale** è una rappresentazione dei dati che utilizza simboli discreti (scatti/livelli), solitamente bit, per codificare informazioni. Un'informazione di questo tipo è caratterizzata dalla sua precisione limitata e dalla capacità di essere manipolata e elaborata facilmente da computer e dispositivi elettronici.

**L'informazione analogica** è una rappresentazione dei dati che utilizza una scala continua di valori per codificare informazioni. Questo significa che l'informazione è rappresentata senza interruzioni e può variare su una scala infinita di livelli.

Il vantaggio di un'informazione analogica è quello di rappresentare direttamente ciò che c'è in natura, mentre uno svantaggio è quello di rappresentare anche dei disturbi.

Ad esempio i nostri sensi ricevono informazioni dalla natura e le inviano in maniera analogica, mentre per le reti logiche servono input e output di tipo digitale.

Questo rappresenta un primo problema da gestire per il corretto funzionamento di un sistema di elaborazione di natura elettronica, in quanto molti input e output in natura non sono di tipo digitale, ma analogico. Nella maggior parte dei sistemi di elaborazione di natura elettronica troviamo infatti in ingresso e in uscita dei convertitori da analogico a digitale e viceversa.

## **Protocollo di una rete logica**

Un altro problema da dover gestire per quanto riguarda l'utilizzo di una rete logica all'interno di un sistema di elaborazione delle informazioni è quello che una rete logica è spesso poco flessibile per quanto riguarda il modo in cui le informazioni le vengono fornite in input e vengono comunicate in output.

Se si desidera interagire con un sistema di elaborazione digitale occorre dunque adeguarsi a tali standard utilizzati dalla rete logica che costituiscono dunque il **protocollo** di tale rete.

Tale protocollo viene stabilito dal ciclo di bus di tale rete, descritto in dettaglio sui datasheet che il produttore del sistema rende sempre disponibili.

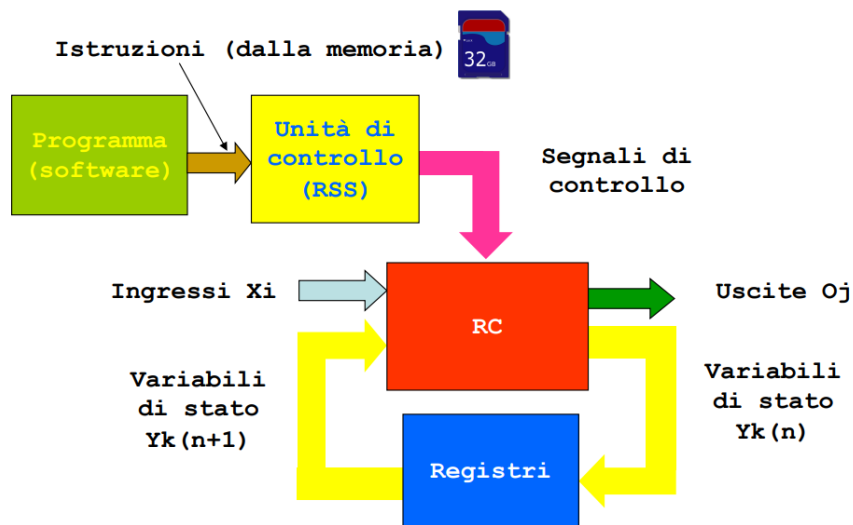
## **Quale rete logica utilizzare?**

Esistono diverse reti logiche da poter utilizzare, le quali vengono scelte in base al contesto applicativo.

Esiste una tipologia di architettura, o rete logica, molto flessibile e utilizzabile, anche se non sempre con risultati ottimali, in ogni contesto. Questa è basata

sul **modello di Von Neumann**, il quale funzionamento è dato dalle istruzioni (programma) inserite in un supporto di memoria. Cambiando tali istruzioni viene modificato il comportamento della rete logica, in modo tale da poter trattare problemi di varia natura, solitamente però con un'efficienza minore rispetto a quella ottenibile tramite una rete logica che può svolgere solamente un compito.

Il sistema di elaborazione si compone dunque di una rete logica sincrona che prende in input un **programma** che consente di variare il funzionamento della rete in base alle esigenze. La rete combinatoria è inoltre connessa ad un'**unità di controllo** che governa tutte le reti logiche presenti nel sistema, abilitando ad esempio gli ingressi e le uscite quando necessario.



Sistema di elaborazione.

## Modello di esecuzione del programma

Il programma risiede in memoria ed è costituito da istruzione codificate in forma binaria. Tali istruzioni vengono eseguite in maniera sequenziale dalla CPU, la quale si dota di un clock per sincronizzare tutta la rete logica.

Ad un livello molto astratto è possibile dire che il funzionamento del sistema può essere descritto tramite **due stati**:

- **Fetch-decode**

Stato in cui la macchina legge in memoria la prossima istruzione da eseguire (fetch) e in cui la rete di controllo analizza l'istruzione letta, prepara le periferiche utili per eseguire l'istruzione e crea una serie di comandi da essere eseguiti (decode).

- **Execute**

Stato in cui la CPU esegue i comandi generati dopo le fasi di instruction fetch e decode.

## **RISC vs CISC**

Esistono due tipologie di CPU:

- **RISC** (es. ARM)

Formate da poche e semplici istruzioni, presentano reti logiche semplici e veloci.

- **CISC** (es. Intel e AMD)

Costituite da molte istruzioni, alcune molto complesse, presentano reti logiche complicate.

Tipicamente a una sola istruzione CISC corrispondono più istruzioni RISC, dunque il codice per programmare architettura di tipo RISC è più denso, ma nonostante ciò l'esecuzione è spesso più veloce, in quanto le reti logiche che eseguono tali istruzioni sono più semplici.

Inoltre, per via della maggiore semplicità delle reti logiche, lo spazio utilizzato da queste è minore, e il restante silicio a disposizione può essere utilizzato per altre finalità, come registri e cache.

Attualmente i processori RISC in circolazione sono diffusi perlopiù nei dispositivi mobili, come smartphone e tablet, e in alcuni computer, come quelli di casa Apple, che stanno adottando questa nuova tecnologia per sfruttarne tutti i suoi vantaggi.

D'altro canto, i processori CISC sono diffusi nella maggior parte dei computer fissi e portatili soprattutto per via dei software già esistenti che dovrebbero essere ricompilati per girare su architetture RISC.

## **Linguaggio assembly**

Siccome per l'umano è spesso incomprensibile il codice macchina, viene solitamente utilizzato un linguaggio chiamato **assembly** per fornire istruzioni in maniera diretta alla CPU.

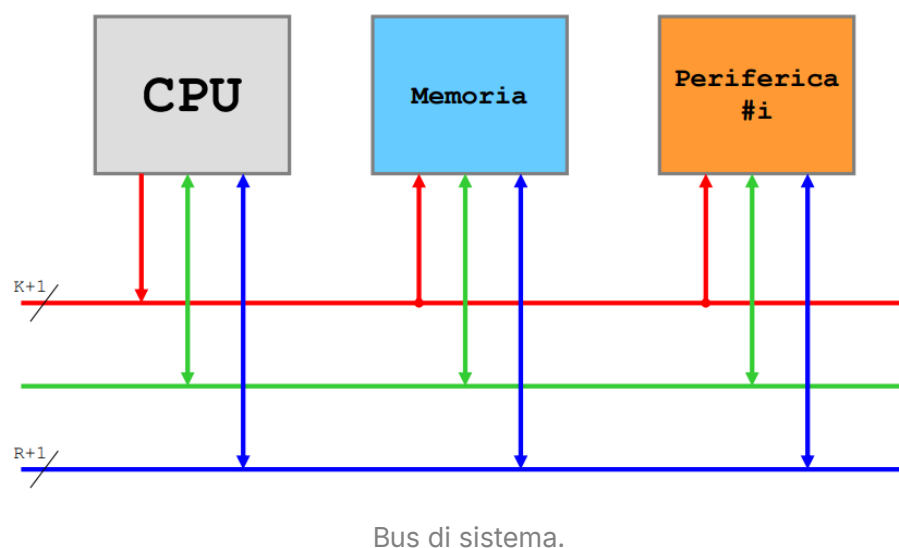
Ogni singola istruzione in linguaggio assembly, più comprensibile per l'umano rispetto al codice binario, può essere tradotta direttamente in una singola istruzione in linguaggio macchina, e questa operazione viene svolta dal **traduttore assembler**.

Normalmente però un programmatore scrive codice in linguaggio ad alto livello, il quale viene poi convertito direttamente dal compilatore in linguaggio macchina bypassando il livello del codice assembly.

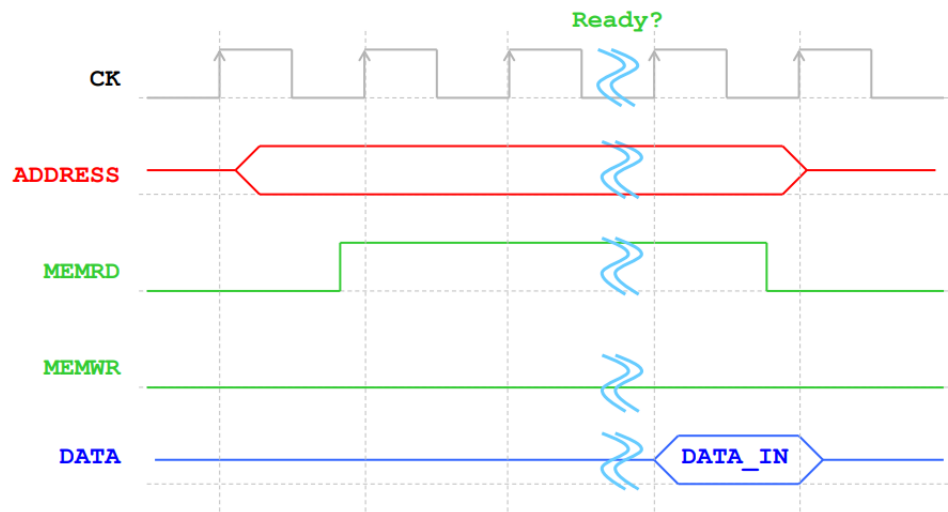
## Ciclo di bus

La lettura e la scrittura di una dato in memoria, le principali operazioni svolte dalla CPU, vengono governate da dei segnali predefiniti con un ben definito andamento temporale, i quali scandiscono il cosiddetto **ciclo di bus**. La durata di un ciclo di bus è indipendente dalla durata del clock, in quanto una singola operazione di scrittura/lettura svolta durante un ciclo di bus ha spesso una durata maggiore di un singolo clock.

Durante tale ciclo i dati (indirizzi e informazioni) viaggiano all'interno dei bus di sistema.

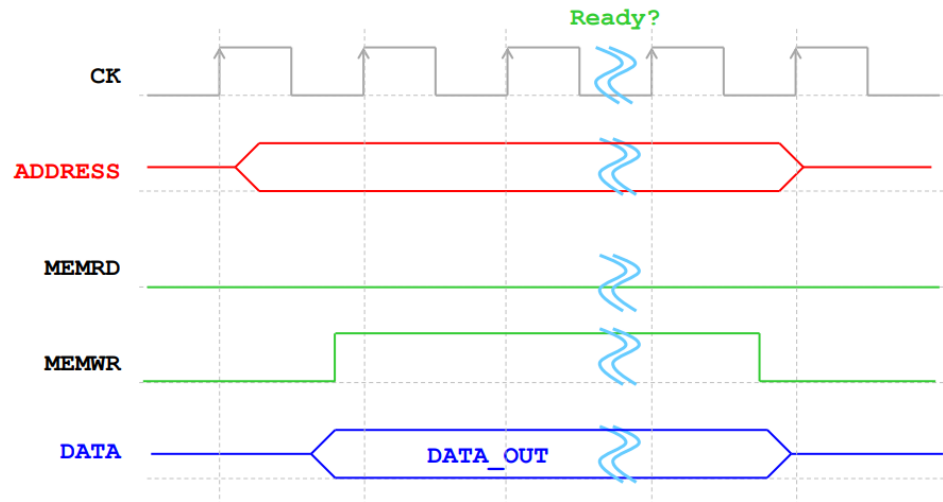


Esempio di ciclo di bus di lettura:



Esempio di ciclo di bus di scrittura:

### Esempio di ciclo di scrittura



La durata di un ciclo di bus è governata da un segnale detto **ready**, generato da una rete logica basata su un contatore. Se ready è a 0 si aspetta un altro ciclo di clock e si ricontra il ready, mentre quando finalmente il ready è a 1 viene eseguita l'istruzione e il ciclo di bus viene chiuso al prossimo clock.

## ▼ 2.0 - Mapping e decodifica

## ▼ 2.1 - Informazioni preliminari

### Spazio di indirizzamento

Una CPU emette un indirizzo per comunicare con le periferiche esterne ed effettuare operazioni di lettura e scrittura. Tale indirizzo viene codificato in codice binario tramite una sequenza di  $n$  bit. Come sappiamo, una sequenza di  $n$  bit è in grado di generare  $2^n$  numeri differenti, dunque un'indirizzo codificato con una sequenza di  $n$  bit può assumere  $2^n$  valori differenti, e tale numero viene detto **spazio di indirizzamento**.

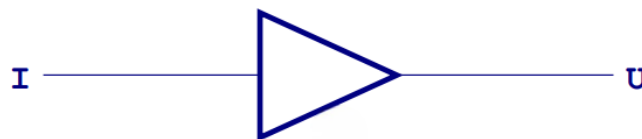
A ciascun indirizzo dello spazio di indirizzamento è associato 1 byte di informazione.

### Driver 3-state

Il driver 3-state è un dispositivo utile per il collegamento/scollegamento delle periferiche al bus dati.

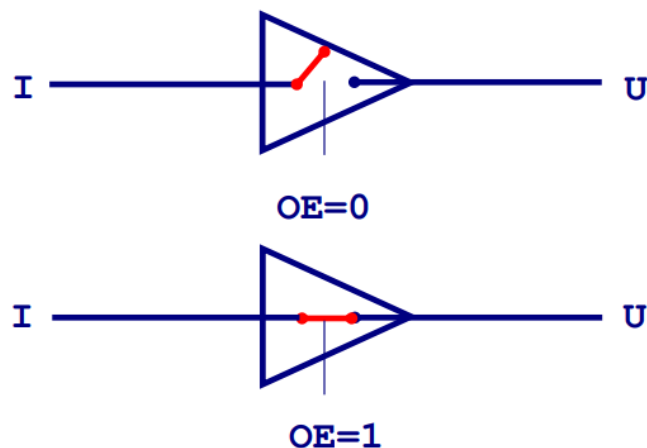
Partiamo dalla definizione di driver:

Il **driver** è un dispositivo che ha come unica utilità quella di rigenerare, fornendo energia, il segnale dato in input e propagarlo in output.



Driver.

Il **driver 3-state** è un driver con due segnali di input. Uno di questi, chiamato **OE**, se impostato a 1 invia il segnale proveniente dall'altro input e lo propaga in output come accadrebbe per un semplice driver, mentre se messo a 0 scollega l'input dall'output, causando il driver ad essere in uno stato detto 3-state, in cui il valore dell'output non può essere determinato.



OE	I	U
1	0	0
1	1	1
0	0	Z
0	1	Z

Driver 3-state.

## Mapping

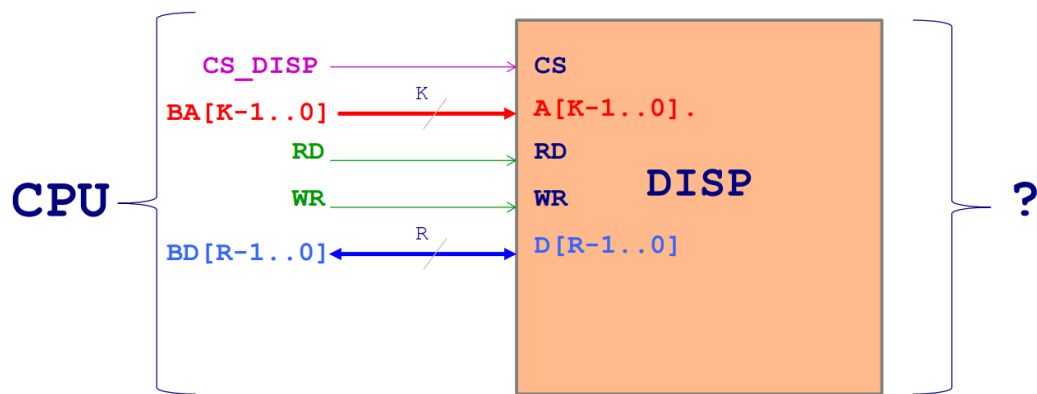
Il **mapping** di un dispositivo consiste nell'associare a tale periferica una finestra di indirizzi dello spazio di indirizzamento tramite la quale la CPU è in grado di comunicare con esso utilizzando il ciclo di bus.



Siccome nello spazio di indirizzamento di un calcolatore viene solitamente mappato più di un dispositivo, ogni volta che viene generato un indirizzo occorre comprendere a quale periferica appartiene in modo da connetterla al bus e disconnettere le altre. Questo viene fatto tramite la cosiddetta decodifica di **primo livello**, e solo successivamente sarà individuato l'elemento specifico all'interno di tale periferica al quale punta l'indirizzo, tramite la decodifica di **secondo livello**.

## Generico dispositivo

Un qualsiasi dispositivo comunica con la CPU tramite la seguente interfaccia standard a sinistra.



La comunicazione del dispositivo con l'esterno avviene invece secondo modalità che sono specifiche del dispositivo e dunque non standard.

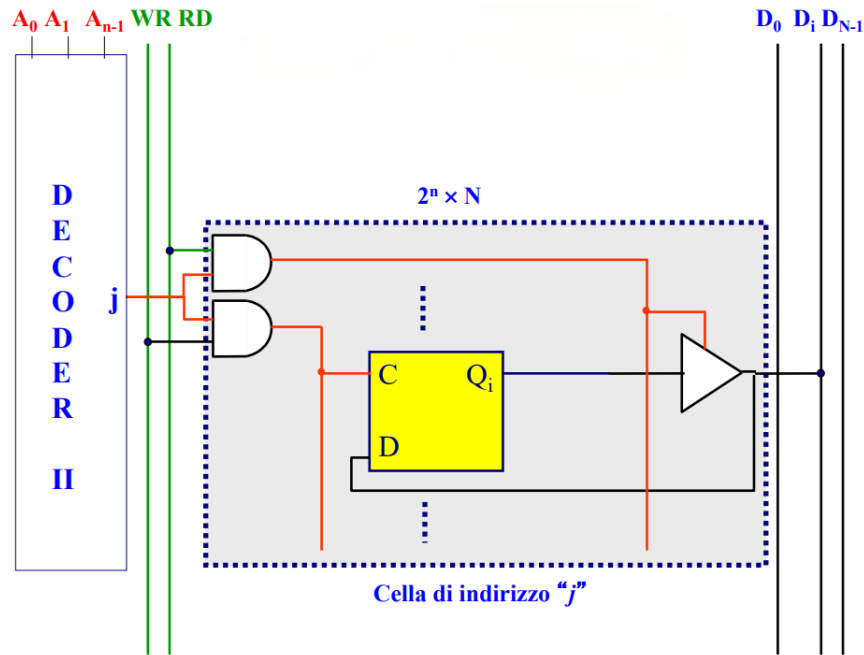
## Tipologie di memorie

### Memorie EPROM

Le **memorie EPROM** sono memorie non volatili a sola lettura. Hanno una capacità a multipli di 2 (32K, 64K, 128K, 256K ecc.).

### Memorie RAM

Le **memorie RAM** sono memorie volatili, leggibili e scrivibili. Hanno una capacità a multipli di 4 (8K, 32K, 128K, 512K ecc.).



La cella di una RAM.

## Circuiti integrati notevoli

### 244

Il circuito **"244"** è un driver 3-state ad 8-bit che connette 8 input ad 8 output. Quando  $OE = 1$  i dati possono passare dagli input agli output, altrimenti gli output si trovano in 3-state.

### 245

Il circuito **"245"** è un driver 3-state bidirezionale ad 8-bit. Quando  $OE = 1$  i dati possono passare dagli input agli output o dagli output agli input, a seconda del valore dell'input DIR, altrimenti gli input e gli output si trovano in 3-state.

## 373

Il circuito "373" è un latch a 8-bit con uscite 3-state.

## 374

Il circuito "374" è un registro edge-triggered, ovvero che assume il valore logico in input durante i fronti di salita del clock, con uscite 3-state.

### ▼ 2.2 - Mapping e decodifica di dispositivi a 8 bit

## Mapping

Consideriamo dispositivi con porta dati a 8 bit, per il mapping di tali dispositivi devono essere rispettate le seguenti condizioni:

- La dimensione della finestra di indirizzi associata a un dispositivo è una **potenza di 2**.
- La finestra di indirizzi è composta da **indirizzi contigui**.

Sia  $k$  il numero di bit di indirizzamento interni ad un dispositivo, solitamente esso occupa  $n = 2^k$  posizioni nello spazio di indirizzamento. Un qualunque dispositivo ha al suo interno un decoder di secondo livello di  $k$  variabili che seleziona i singoli oggetti indirizzabili.

- Si dice che un dispositivo è **mappato** all'indirizzo  $A$  se gli indirizzi dei byte del dispositivo sono compresi tra  $A$  e  $A + (n - 1)$ , cioè se  $A$  è l'indirizzo più basso tra tutti gli indirizzi associati al dispositivo.
- Si dice che un dispositivo è **allineato** se  $A$  è un multiplo di  $n$ . Inoltre se un dispositivo è allineato allora i  $k$  bit meno significativi di  $A$  sono uguali a zero (es. un dispositivo da 8 byte è allineato se è mappato ad un indirizzo il cui valore binario termina con 3 zeri).

## Lettura e scrittura

La **lettura** e la **scrittura** in un dispositivo avvengono tramite l'utilizzo degli input RD e WR:

- **RD**, detto anche Output Enable, è il comando di lettura. Quando CS e RD sono attivi, il dispositivo espone su  $BD[7..0]$  il contenuto della cella indirizzata.
- **WR** è il comando di scrittura. Quando CS e WR sono attivi, il dato presente su  $BD[7..0]$  viene memorizzato nella cella indirizzata durante il fronte di discesa di WR.

## Decodifica

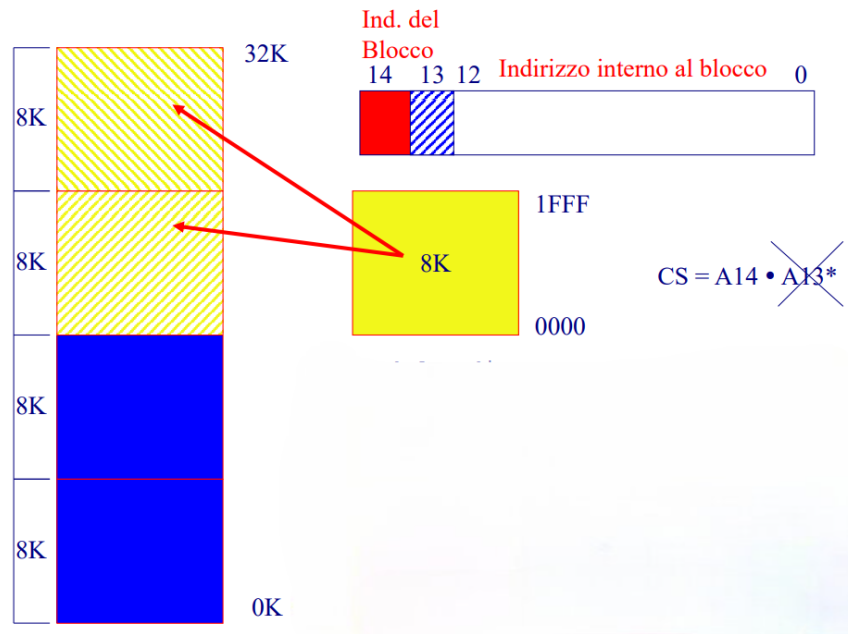
Consideriamo un dispositivo di  $2^k$  byte mappato in uno spazio di indirizzamento. Per individuare se un indirizzo  $ai = \alpha \# i$  si riferisce al dispositivo e, in tal caso, a quale oggetto del dispositivo si riferisce, occorre effettuare le seguenti 2 decodifiche:

- **Decodifica di primo livello**

Tale decodifica è utilizzata per decodificare  $\alpha$ , che confrontato con l'indirizzo in cui il dispositivo è mappato si riesce a stabilire se l'indirizzo è interno ad esso.

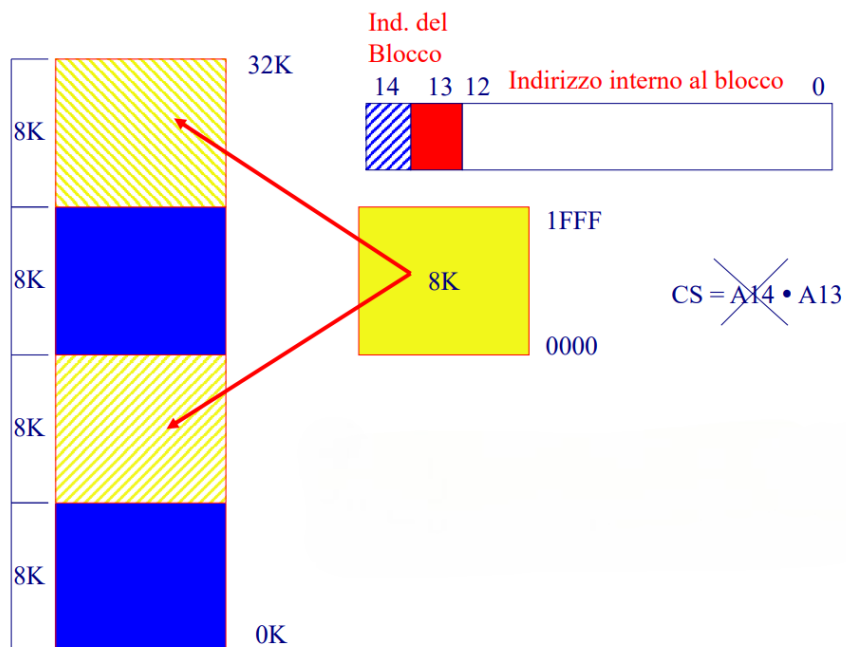
Per questa decodifica sono necessari al massimo  $20 - k$  bit più significativi dell'indirizzo. La decodifica è **completa** se vengono utilizzati tutti i  $20 - k$  bit per decodificare  $\alpha$ , **semplificata** se si utilizza un sottoinsieme dei  $20 - k$  bit.

Mettiamo infatti caso che in uno spazio di indirizzamento da  $32k$  in cui gli indirizzi da  $16k$  al  $32k$  sono liberi, deve essere mappato un dispositivo da  $8k$ . È possibile in questo caso assegnare tutti i  $16k$  liberi al dispositivo, rendendo l'espressione del CS più semplice.



Decodifica semplificata allineata.

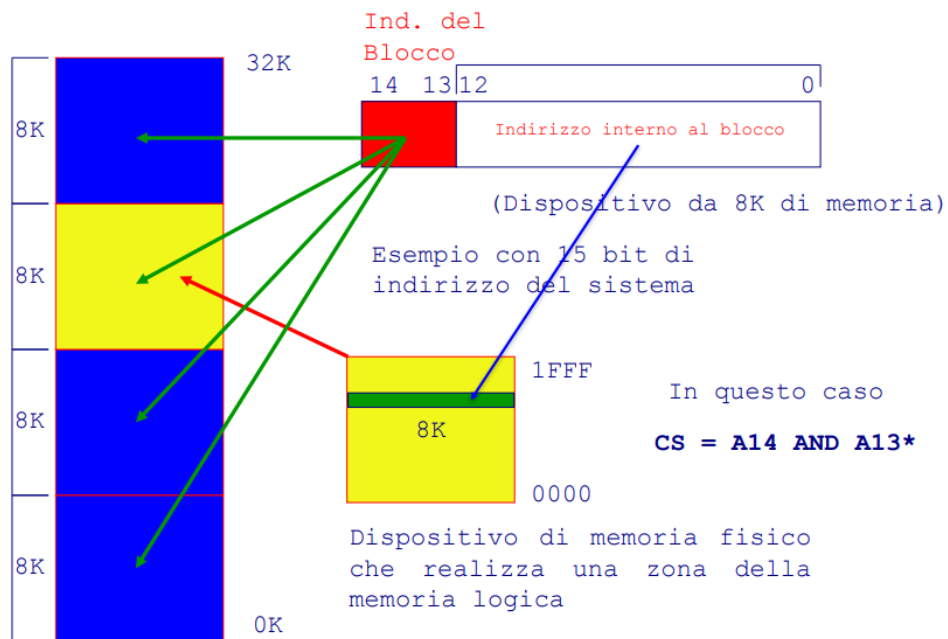
È possibile anche effettuare una decodifica semplificata assegnando al dispositivo due o più finestre di indirizzi non allineate.



Decodifica semplificata non allineata.

- **Decodifica di secondo livello**

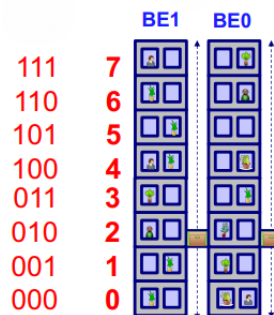
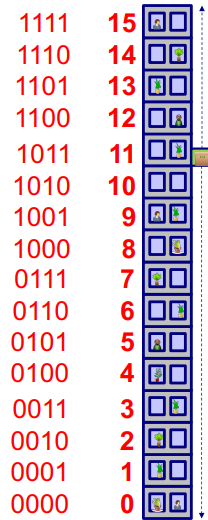
Tale decodifica è utilizzata per decodificare  $i$ , ovvero la posizione dell'oggetto indirizzato all'interno del dispositivo. Per questa decodifica sono necessari i  $k$  bit meno significativi dell'indirizzo.



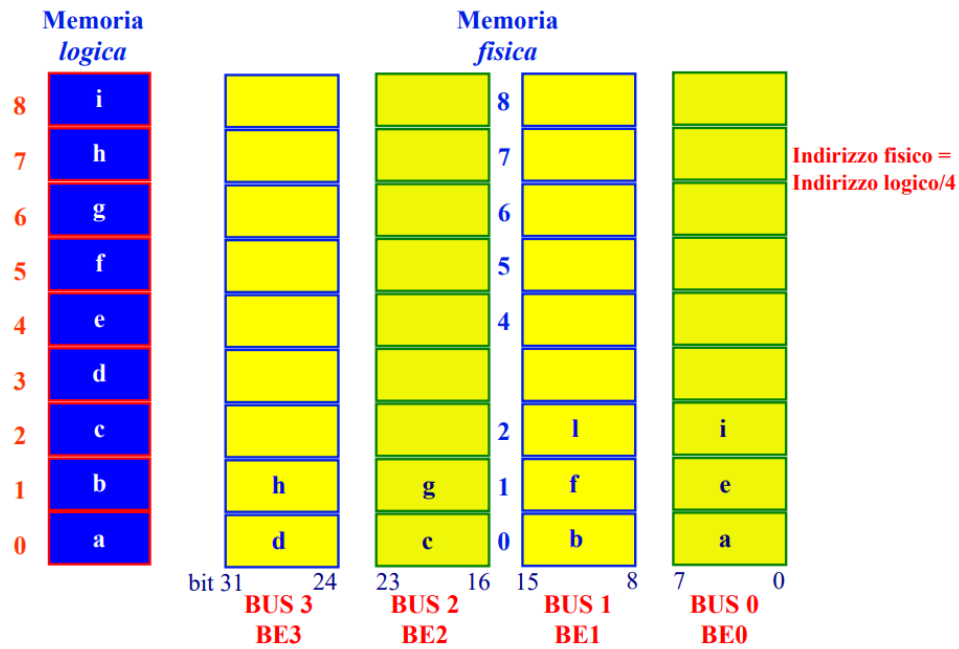
Decodifica completa.

### ▼ 2.3 - Parallelismo dei dati

Ogni trasferimento di dati dalla CPU a una periferica esterna o viceversa richiede un ciclo di bus. Per trasferire più di un elemento alla volta occorre dunque suddividere la memoria in memorie più piccole, potendo così accedere a più memorie durante lo stesso ciclo di bus. In questo modo vengono utilizzati meno bit (1 in meno per ogni suddivisione a metà della memoria) per indicare l'indirizzo a cui accedere e ad ogni indirizzo sono associate più celle di memoria, dunque occorre utilizzare ulteriori bit chiamati ByteEnable per specificare quali memorie vengono utilizzate durante il trasferimento e dunque far funzionare correttamente il chip select.



Tramite il parallelismo dei dati si distingue tra due tipologie di indirizzi, quello logico, dato da come è strutturata logicamente la memoria e quindi da come la vede il programmatore, e quello fisico, dato dall'effettivo indirizzo di ogni cella all'interno della sua memoria.



Memoria logica vs memoria fisica.

Nell'immagine appena mostrata si nota un caso di parallelismo 32 bit. I ByteEnable in tale caso sono i seguenti:

BE3	BE2	BE1	BE0	
1	1	1	1	Word 32 bit
0	0	1	1	Half word bassa
1	1	0	0	Half word alta
0	0	0	1	Byte 0-7
0	0	1	0	Byte 15-8
.....				
				<i>etc.</i>

### ▼ 3.0 - Linguaggio macchina

## Instruction Set Architecture

L'**Instruction Set Architecture** comprende l'insieme delle istruzioni e dei registri di una CPU. Mediante l'ISA è possibile accedere alle risorse interne (es. registri) ed esterne (es. memorie).



Quasi ogni CPU possiede una proprio ISA, che varia in base al numero di istruzioni possibili e dei registri accessibili dal programmatore. A proposito coesistono due scuole di pensiero:

- RISC: insieme ridotto di istruzioni con molti registri interni.
- CISC: insieme ampio di istruzioni con pochi registri interni.

L'obiettivo di un ISA è quello di minimizzare la seguente formula, ovvero il tempo di esecuzione del codice ( $CPI_{medio}$ : numero medio di clock necessari per eseguire una singola istruzione):

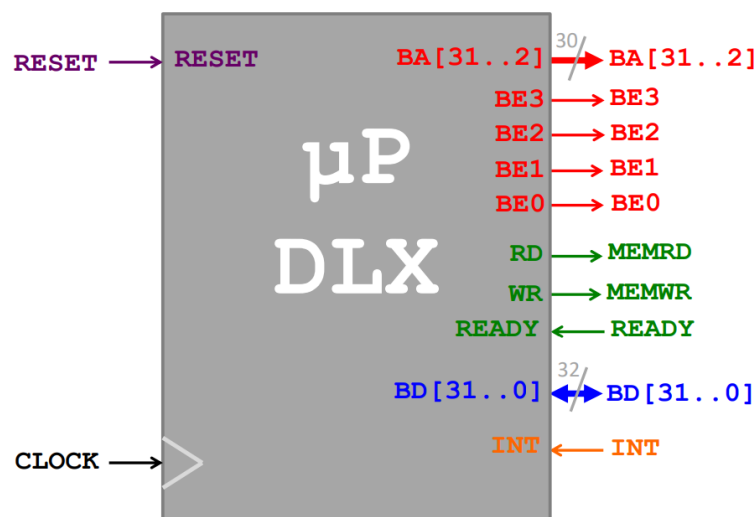
$$CPU_{Time} = N_{istruzioni} * CPI_{medio} * T_{CK}$$

Nei processori di tipo RISC  $N_{istruzioni}$  è più grande,  $CPI_{medio}$  è minore in quanto le istruzioni sono più semplici e dunque più veloci, e il  $T_{CK}$  è minore in quanto reti logiche più semplici sono solitamente più veloci.

Esiste un progetto, chiamato RISC-V, il quale mira a creare un ISA unico e open source, consentendo la compatibilità dei programmi nelle diverse architetture e facilitando la scrittura di codice. Tale ISA è molto simile a quella del DLX che studieremo in questo corso.

## Segnali del processore DLX

I segnali in input e in output del processore DLX sono i seguenti:

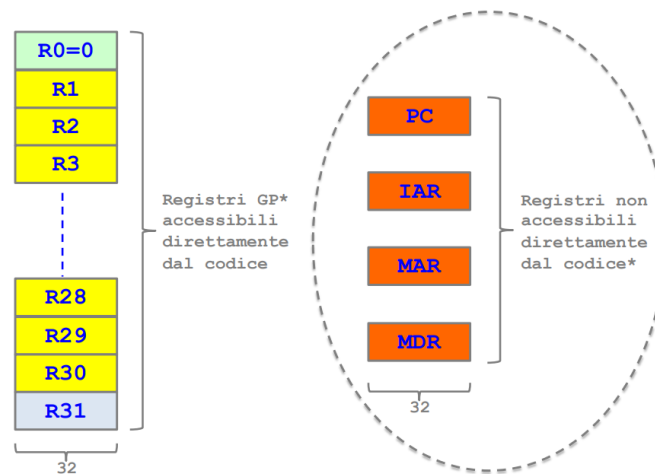


Il segnale RESET è asserito all'avvio da una rete esterna. Anche i segnali READY e INT sono generati da reti esterne.

## Caratteristiche dell'ISA DLX

Le caratteristiche dell'ISA del DLX sono le seguenti:

- Unico spazio di indirizzamento di 4GB.
- 32 registri da 32 bit (R0 ... R31) sono accessibili dal codice.
  - R0 = 0.
  - In R31 viene salvato l'indirizzo di ritorno nel caso di salti.
- La codifica delle istruzioni ha lunghezza costante.
- Ci sono 3 tipologie di istruzione: I, R e J.
- L'unica modalità di indirizzamento in memoria è quella indiretta, tramite registro + offset.
- Le operazioni aritmetico/logiche sono eseguite solo tra registri, non tra registri e memorie.



Registri del DLX.

## Tipi di dato ed estensione del segno

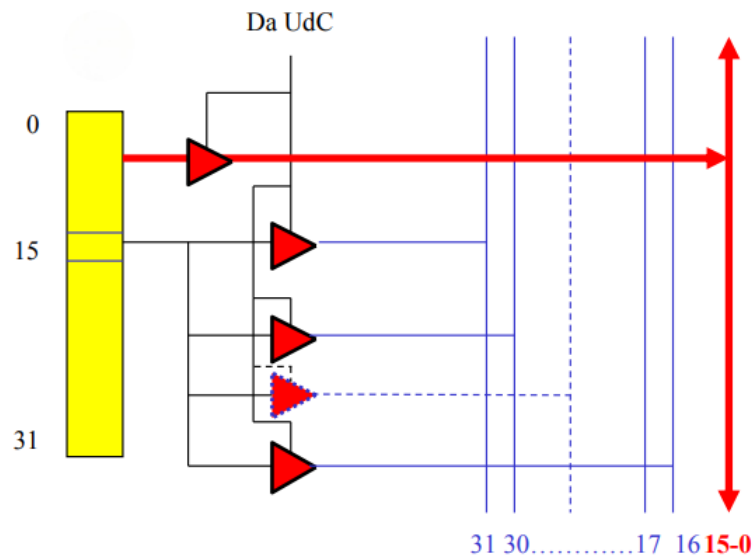
Nel DXL sono disponibili 3 tipologie di dato:

- **BYTE** (8 bit)

- **HALF-WORD** (16 bit)
- **WORD** (32 bit)

I dati di dimensione inferiore a 32 bit, dunque BYTE e HALF-WORD, una volta letti dalla memoria devono essere estesi a 32 bit durante il caricamento nei registri.

Questa operazione può essere fatta in 2 modalità, con segno o senza.



Estensione di una HALF-WORD con segno.

## Esempi

- Assumiamo che dalla memoria sia stato preso il seguente dato di tipo BYTE: 10110101.

Per trasferirlo nei registri l'estensione a 32 bit può avvenire in 2 modi:

- Senza segno. In questo caso l'estensione avviene aggiungendo 24 zeri.

00000000000000000000000010110101

- Con segno. In questo caso l'estensione avviene replicando 24 volte il bit di segno.

11111111111111111111111111110110101

## Il set di istruzioni del DLX

Le principali istruzioni di **trasferimento dati**:

- Load byte signed e unsigned (LB, LBU), load halfword signed e unsigned (LH, LHU), load word (LW).
- Store byte (SB), store halfword (SH) e store word (SW).
- Copia un dato da un registro GP a un registro speciale (MOVI2S) e viceversa (MOVS2I).

Le principali istruzioni **aritmetico-logiche**:

- Istruzioni logiche (anche con operatore immediato): AND, ANDI, OR, ORI, XOR, XORI.
- Istruzioni aritmetiche: ADD, ADDI, SUB, SUBI.
- Istruzioni di shift: SLL, SRL, SRA.
- Istruzioni di set condition: Sx, con  $x = \{EQ, NE, LT, GT, LE, GE\}$ .

Le principali istruzioni di **trasferimento del controllo**:

- Istruzioni di salto condizionato: BNEZ, BEQZ.
- Istruzioni di salto incondizionato PC-relative (J) e con registro (JR).
- Istruzioni di chiamata a procedura jump and link PC-relative (JAL) e con registro (JALR). L'indirizzo di ritorno viene automaticamente salvato in R31.
- Istruzione di ritorno dalla procedura di servizio delle interruzioni: RFE.

Data Transfer	Aritmetiche/logiche	Controllo
<b>LW</b> $Ra, Imm_{16bit} (Rb)$ <b>LB</b> $Ra, Imm_{16bit} (Rb)$ <b>LBU</b> $Ra, Imm_{16bit} (Rb)$ <b>LH</b> $Ra, Imm_{16bit} (Rb)$ <b>LHU</b> $Ra, Imm_{16bit} (Rb)$ <b>SW</b> $Ra, Imm_{16bit} (Rb)$ <b>SH</b> $Ra, Imm_{16bit} (Rb)$ <b>SB</b> $Ra, Imm_{16bit} (Rb)$  <b>MOV2SI</b> $Ra, Rs^*$ <b>MOVI2S</b> $Rs^*, Ra$ Special register $Rs^*$ (IAR)	<b>ADD</b> $Ra, Rb, Rc$ <b>ADDI</b> $Ra, Rb, Imm_{16bit}$ <b>ADDU</b> $Ra, Rb, Rc$ <b>ADDUI</b> $Ra, Rb, Imm_{16bit}$ <b>SUB</b> $Ra, Rb, Rc$ <b>SUBI</b> $Ra, Rb, Imm_{16bit}$ <b>SUBU</b> $Ra, Rb, Rc$ <b>SUBUI</b> $Ra, Rb, Imm_{16bit}$ <b>SLL</b> $Ra, Rb, Rc$ <b>SLLI</b> $Ra, Rb, Imm_{16bit}$ <b>SRL</b> $Ra, Rb, Rc$ <b>SRLI</b> $Ra, Rb, Imm_{16bit}$ <b>SRA</b> $Ra, Rb, Rc$ <b>SRAI</b> $Ra, Rb, Imm_{16bit}$ <b>OR</b> $Ra, Rb, Rc$ <b>ORI</b> $Ra, Rb, Imm_{16bit}$ <b>XOR</b> $Ra, Rb, Rc$ <b>XORI</b> $Ra, Rb, Imm_{16bit}$ <b>AND</b> $Ra, Rb, Rc$ <b>ANDI</b> $Ra, Rb, Imm_{16bit}$  <b>LHI</b> $Ra, Imm_{16bit}$	<b>Sx</b> $Ra, Rb, Rc$ <b>SxI</b> $Ra, Rb, Imm_{16bit}$ <b>BEQZ</b> $Ra, Imm_{16bit}$ <b>BNEZ</b> $Ra, Imm_{16bit}$ <b>J</b> $Imm_{26bit}$ <b>JR</b> $Ra$ <b>JAL</b> $Imm_{26bit}$ <b>JALR</b> $Ra$  <i>x può essere:</i> <i>LT, GT, LE, GE, EQ, NE</i>  <b>Gestione interrupt:</b> <b>RFE</b>

Per le istruzioni aritmetiche: l'immediato a 16 bit è esteso senza segno se di tipo U (unsigned) altrimenti con segno.

Per istruzioni logiche, sempre estensione senza segno.

$Ra \in \{R0^+, R1, \dots, R30, R31\}$   
 $Rb \in \{R0, R1, \dots, R30, R31\}$   
 $Rc \in \{R0, R1, \dots, R30, R31\}$

\*Ra non può essere R0 come registro destinazione di istruzioni load, MOV2SI, aritmetico/logiche, LHI e SET

Set di istruzioni del DLX.

## Esempi

- Data transfer
  - $LW\ R1, 0x0040(R3)$  significa  $R1 \leftarrow_{32} M[40 + R3]$
  - $LB\ R1, 0x0040(R3)$  significa  $R1 \leftarrow_{32} (M[40 + R3]_7)^{24} \#\# M[40 + R3]$
  - $LBU\ R1, 0x0040(R3)$  significa  $R1 \leftarrow_{32} (0)^{24} \#\# M[40 + R3]$
  - $SB\ R1, 0x0040(R3)$  significa  $M[40 + R3] \leftarrow_{32} R1$
- Aritmetico/logiche
  - $ADD\ R1, R2, R3$  significa  $R1 \leftarrow R2 + R3$  (formato R)
  - $ADDI\ R1, R2, 3$  significa  $R1 \leftarrow R2 + 3$  (formato I)
  - $LHI\ R1, 8420$  significa  $R1 = 8420 \#\# 0000h = 84200000h$
- Controllo

- Set
  - $SLT\ R1, R2, R3$  significa  $R1 \leftarrow 1$  se  $R2 < R3$  altrimenti  $R1 \leftarrow 0$  (formato R)
  - $SLTI\ R1, R2, 3$  significa  $R1 \leftarrow 1$  se  $R2 < 3$  altrimenti  $R1 \leftarrow 0$  (formato I)
- Salti incondizionati
  - $J\ offset$  significa  $PC = PC + 4 + (offset[25])^6 \##\ offset$  (formato J)
  - $JR\ R3$  significa  $PC = R3$  (formato R)
  - $JAL\ offset$  significa  $R31 = PC + 4$  e  $PC = PC + 4 + (offset[25])^6 \##\ offset$  (formato J)
  - $JALR\ R5$  significa  $R31 = PC + 4$  e  $PC = R5$
  - $JR\ R31$  significa  $PC = R31$  (istruzione per tornare da una procedura)
- Salti condizionati
  - $BEQZ\ R4, Imm_{16}$  significa  $PC = PC + 4 + Imm_{16}[15]^{16} \##\ Imm_{16}$  se  $R4 == 0$  altrimenti  $PC = PC + 4$

## Codifica binaria delle istruzioni

Ogni istruzione al livello ISA deve essere convertita in codice binario per essere eseguita dalla CPU e tale codifica in binario deve contenere tutte le informazioni necessarie all'unità di controllo per poter eseguire l'istruzione.

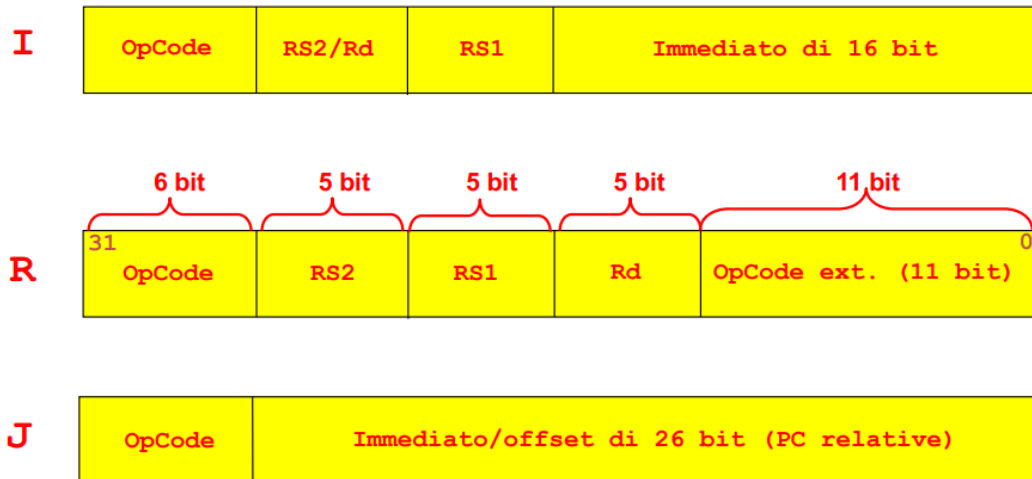
Esistono CPU con codifica delle istruzioni a lunghezza costante e variabile.

### Notazione per la costruzione di configurazioni binarie

- $<< n$ : traslazione logica a sinistra di  $n$  bit (inserendo "0" a destra).
- $>> n$ : traslazione logica a destra di  $n$  bit (inserendo "0" a sinistra).
- $\##$ : concatenazione di 2 campi.
- $(x)^n$ : ripetizione  $n$  volte di  $x$ .

- $x_n$ :  $n$ -esimo bit di una configurazione binaria  $x$ .
- $x_{n..m}$ : selezione di un campo in una stringa di bit  $x$ .

## Codifica binaria nel DLX



Nota: "RS" sta per registro sorgente, "Rd" sta per registro destinazione.

## Modalità di accesso alla memoria

Ogni ISA dispone di istruzioni per accedere alla memoria in lettura e scrittura. I due principali metodi di accesso alla memoria sono:

- **Diretto**

Con questo modalità l'istruzione contiene al suo interno un valore che specifica l'indirizzo di accesso alla memoria.

Es. l'istruzione "LB R7, 0800h" presenta un indirizzamento diretto e dice di leggere un byte all'indirizzo 0800h e di memorizzarlo nel registro R7.

- **Indiretto**

Con questa modalità l'indirizzo di accesso alla memoria è ottenuto sommando un valore costante al contenuto di un registro.

Es. l'istruzione "LB R7, 0800 (R2)" presenta un indirizzamento indiretto e dice di leggere un byte all'indirizzo  $R2 + 0800h$  e di memorizzarlo nel registro R7.

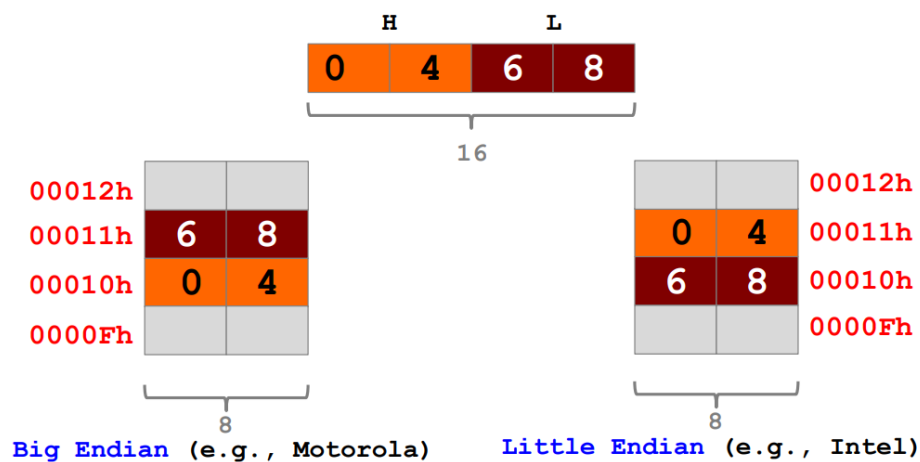
La differenza tra i due indirizzamenti è notevole in quanto con quello indiretto è possibile ad esempio creare dei loop che ad ogni iterazione leggono o scrivono un dato in una posizione diversa della memoria cambiando ogni volta l'indirizzo di memoria dal contenuto di un registro, mentre tramite indirizzamento diretto ciò non è possibile.

L'ISA del DLX consente solo l'indirizzamento indiretto, e l'indirizzo a 32 bit è sempre ottenuto sommando un registro a 32 bit con un valore immediato a 16 bit esteso a 32 con segno.

## Big Endian vs Little Endian

In un sistema con bus dati maggiore di 8 bit le informazioni possono essere memorizzate seguendo due modalità:

- **Big endian:** il byte più significativo viene memorizzato nella cella di memoria con indirizzo più basso.
- **Little endian:** il byte meno significativo viene memorizzato nella cella di memoria con indirizzo più basso.



Esempio di memorizzazione Big Endian vs Little Endian.

## ▼ 4.0 - Interruzioni

### Gestione degli eventi

In un sistema a microprocessore è di fondamentale importanza poter gestire eventi che si verificano al di fuori del programma che si sta eseguendo, come la pressione di un pulsante sulla tastiera o lo spostamento del mouse.



È possibile effettuare ciò tramite due tecniche:

- **Polling**

Questa tecnica, poco efficiente e poco usata, consiste nel controllare periodicamente se tali eventi si sono verificati. In questo modo la CPU spende molti cicli di bus per effettuare tali controlli rallentando l'esecuzione generale.

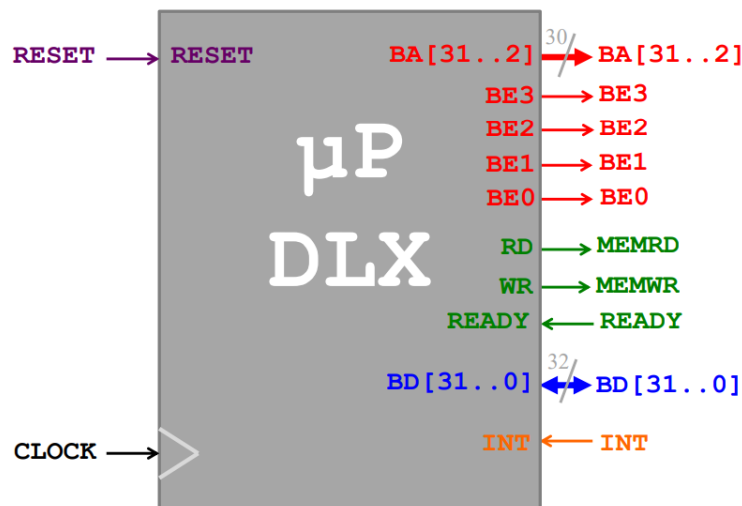
- **Interrupt**

Questa tecnica consiste nel segnalare alla CPU che si è verificato un evento che merita immediata attenzione e che quindi blocca il regolare flusso di esecuzione del codice. Una volta che un interrupt viene segnalato alla CPU, quest'ultima esegue automaticamente una porzione di codice chiamata **interrupt handler** al fine di gestire tale evento.

Gli eventi da gestire possono essere relativi a fattori esterni (es. pressione di un tasto) o interni (es. divisione per zero, overflow), e in quest'ultimo caso si parla di eccezioni.

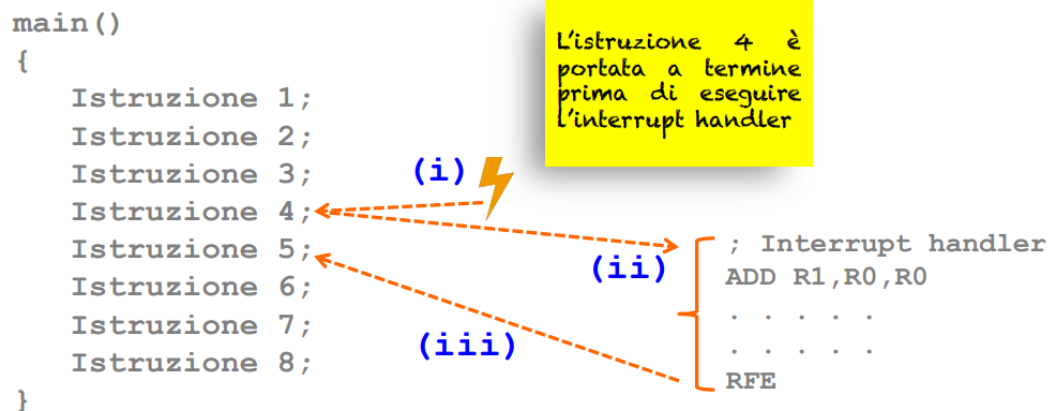
## Gestione interrupt nel DLX

In ogni processore, come anche nel DLX, è presente un segnale (es. INT) per gestire le interruzioni. In molti casi, ma non nel DLX, è presente anche un ulteriore segnale (es. NMI) per gestire interruzioni che non possono essere ignorate.



Nel caso in cui avviene un evento che la CPU deve gestire immediatamente, il segnale di INT viene portato a 1 e viene portata a termine l'operazione che si stava eseguendo per poi, subito dopo, viene assegnato il valore 0 al PC, al fine di eseguire il codice presente all'indirizzo 0 della memoria, ossia l'**Interrupt Handler**.

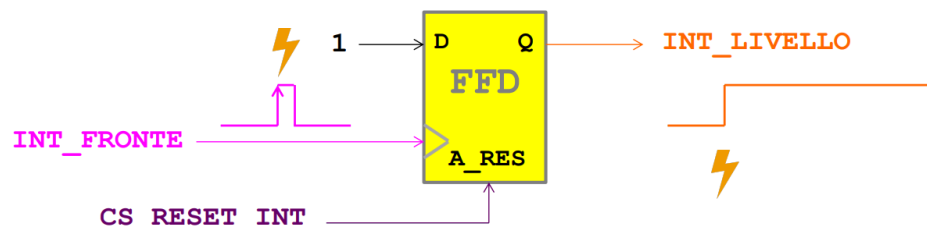
Qui vengono eseguite le operazioni utili a gestire le interruzioni e al suo termine, stabilito dalla presenza dell'istruzione RFE, il PC viene riportato all'indirizzo della prossima istruzione da eseguire, il quale era stato salvato nel registro IAR prima di eseguire l'Interrupt Handler.



### Trasformazione del segnale di interrupt da fronte a livello

Il DLX è sensibile al livello del segnale, dunque una volta che un segnale di interrupt viene portato a 1 questo rimane tale finchè la causa che lo ha generato non sia stata gestita dalla CPU.

Può però accadere che il dispositivo che genera l'interrupt assuma che la CPU sia sensibile ai fronti del segnale. In questo caso occorre eseguire una trasformazione del segnale da fronte a livello utilizzando ad esempio un FFD:



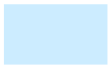
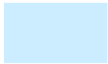
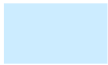
Trasformazione del segnale di interrupt da fronte a livello.

Così facendo il livello logico del segnale INT\_LIVELLO deve essere riportato a zero da un opportuno comando software che asserisce il segnale CS\_RESET\_INT.

### Consistenza dei dati

È importante che al termine della gestione dell'interruzione i registri vengano portati al valore che avevano prima dell'arrivo dell'interrupt, in quanto è possibile che le prossime operazioni debbano utilizzare dei valori salvati in precedenza nei registri.

Per questo motivo, all'inizio e alla fine dell'Interrupt Handler occorre inserire il codice utile per salvare in memoria il contenuto dei registri e ripristinarlo una volta terminato.

```
00000000h   ;Istruzioni che salvano i registri  
                                     ; modificati dalle istruzioni seguenti  
  
           ;codice di risposta alla richiesta  
                                     ;di interruzione  
  
           ;istruzioni di ripristino dei registri  
                                     ;modificati in precedenza  
  
XXXXXXXXXh  RFE    ; ritorno dall'interrupt (PC ← IAR)
```

## Gestione di interruzioni multiple

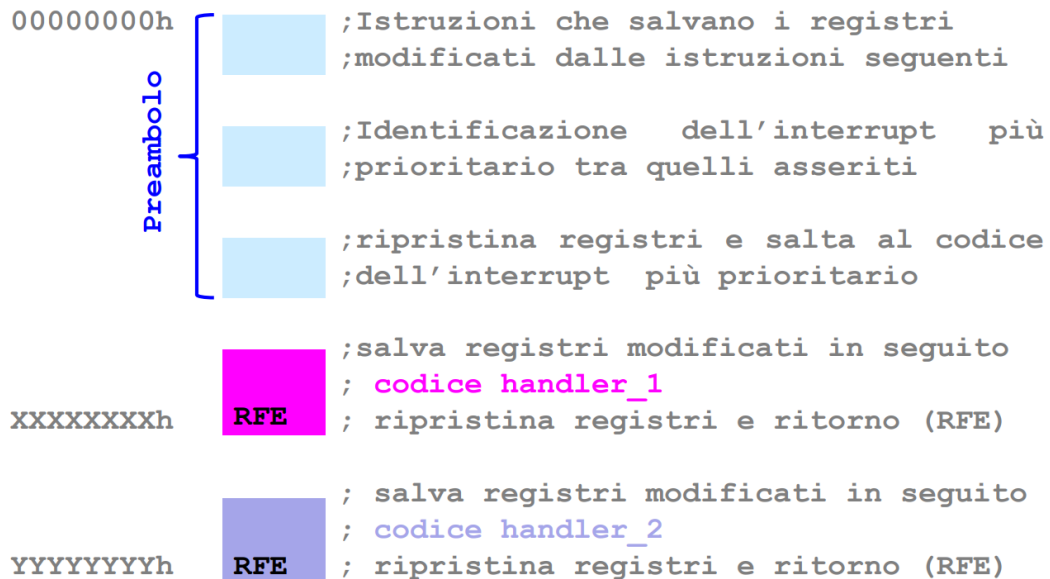
Avendo il DLX un singolo segnale INT, si convogliano tutti gli interrupt verso tale segnale e li si gestisce uno dopo l'altro, valutandone la priorità tramite software o hardware (PIC).

### Gestione software

Tramite software viene identificata l'interruzione che ha maggiore priorità e si salta all'indirizzo in cui vi è salvato il codice utile per gestire tale interrupt.

È possibile anche che un'interruzione arrivi durante la gestione di un altro interrupt. In questo caso, non avendo il DLX di base una gestione dello stack, occorre gestirlo tramite software.

La struttura di un Interrupt Handler con più sorgenti di interruzione diventa dunque la seguente:

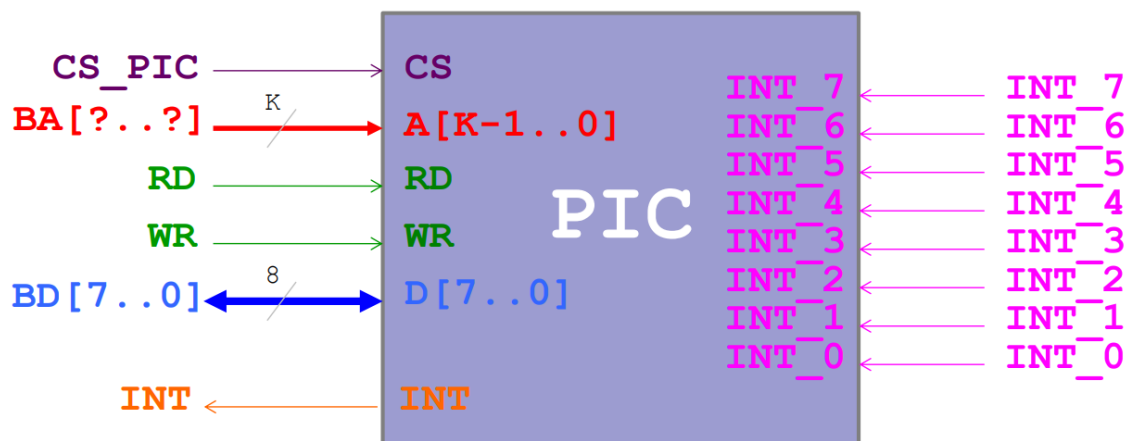


## Gestione hardware

La gestione hardware di interruzioni multiple può essere effettuata tramite un dispositivo chiamato **PIC (Programmable Interrupt Controller)**.

Tale dispositivo si occupa di ricevere in input diversi segnali di interrupt e restituire il codice del segnale con maggiore priorità.

La struttura di un tale dispositivo è la seguente:



Notiamo che in input è presente anche il segnale WR in quanto è un dispositivo programmabile al fine di stabilire in partenza la priorità di ciascun interrupt, dunque tramite alcune istruzioni software è possibile fare ciò.

Il segnale D[7..0] trasporta in uscita il codice del segnale di interrupt attualmente attivo con la maggiore priorità, mentre il segnale INT viene connesso a quello del DLX per scatenare un'interruzione al seguito della valutazione delle priorità.

## ▼ 5.0 - Handshake

### Problemi nell'utilizzo di dispositivi I/O

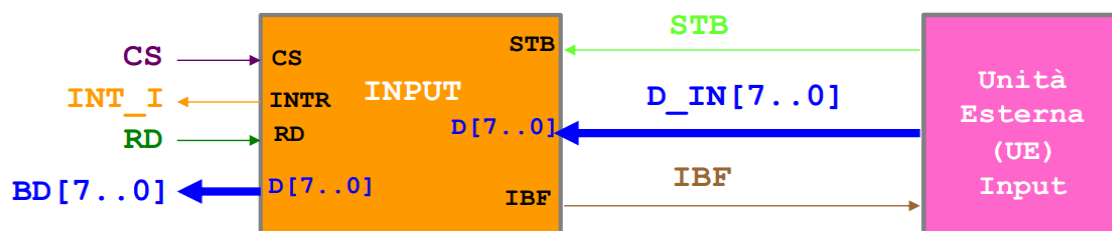
L'utilizzo di dispositivi esterni I/O che inviano e ricevono dati a/da la CPU fanno sorgere alcuni problemi di gestione:

- Come può la CPU sapere che un nuovo dato è stato inviato da una porta di input?
- Come può una porta di input sapere che la CPU è pronta per leggere un nuovo dato?
- Come può una porta di output sapere che un nuovo dato è stato inviato dalla CPU?
- Come può la CPU sapere che una porta di output è pronta per ricevere un nuovo dato?

Per questo motivo si introducono le **porte I/O**, le quali si basano sul protocollo di **handshake** per facilitare la comunicazione tra la CPU e un dispositivo I/O.

### Protocollo di handshake in input

I **segnali** che vengono utilizzati da una **porta di input** che utilizza il protocollo di handshake sono i seguenti:



- **IBF** (Input Buffer Full) indica a UE se la porta è piena.  
IBF = 0 → la porta non è piena, dunque UE può scrivere.

IBF = 1 → la porta è piena, dunque UE non può scrivere.

- **STB** indica alla porta se UE sta scrivendo.

STB = 0 → UE non sta scrivendo.

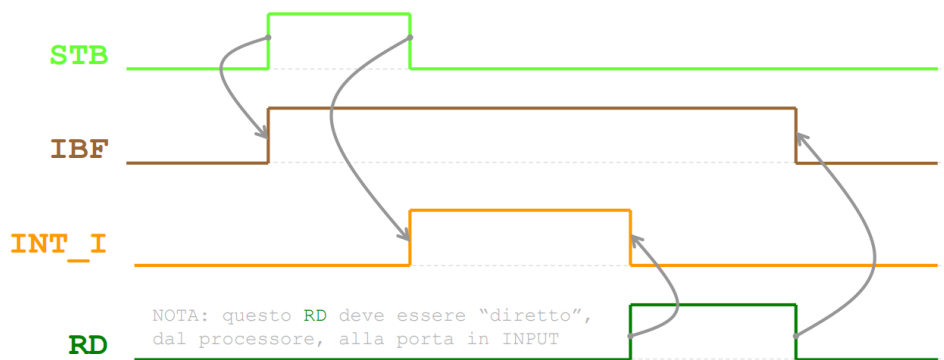
STB = 1 → UE sta scrivendo.

- **INT\_I** (Interrupt Request) indica se c'è un interrupt request inviata alla CPU che deve essere gestita.

Il **procedimento di handshake in input** si basa sui seguenti punti:

1. UE, controllando che IBF = 0, per scrivere nella porta invia il dato in D\_IN[7..0] e imposta STB = 1, che di conseguenza porta IBF a 1.
2. Una volta che la scrittura è terminata STB viene portato a 0 e di conseguenza la porta di input attiva INT\_I mettendo il dato in D[7..0].
3. Quando possibile la CPU andrà a gestire la richiesta di interrupt leggendo il dato in input, e una volta terminato la porta di input porterà IBF a 0.

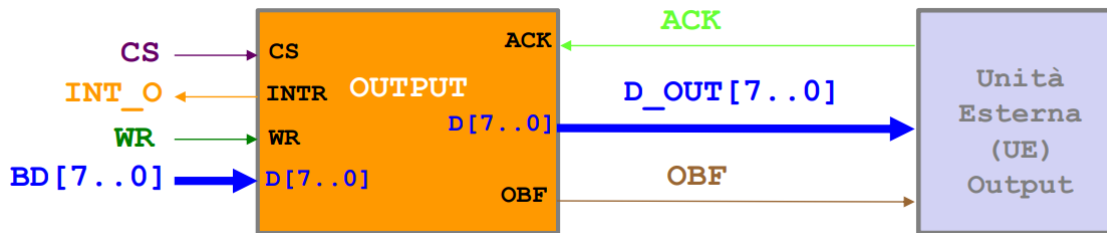
Le **forme d'onda dei segnali** durante il protocollo di handshake sono le seguenti:



Segnali del protocollo di handshake per una porta di input.

## Protocollo di handshake in output

I **segnali** che vengono utilizzati da una **porta di output** che utilizza il protocollo di handshake sono i seguenti:

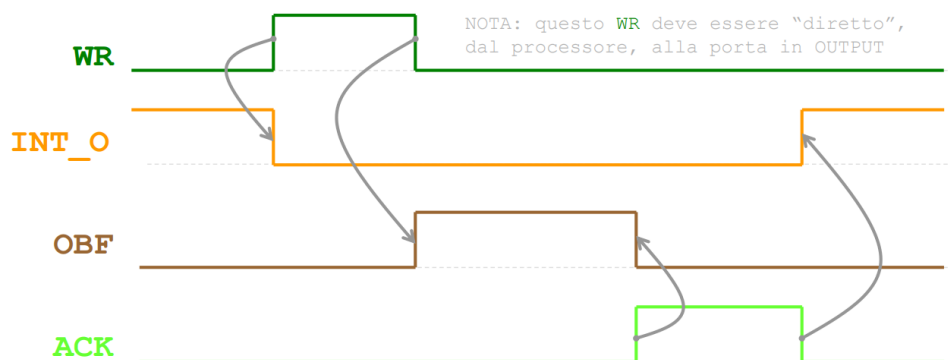


- **OBF** (Output Buffer Full) indica a UE se c'è un dato da leggere.  
OBF = 0 → UE non ha un dato da leggere.  
OBF = 1 → UE ha un dato da leggere.
- **ACK** (acknowledge) indica alla porta se UE ha letto il dato in output.  
ACK = 0 → UE non ha ancora letto.  
ACK = 1 → UE ha letto.
- **INT\_O** indica alla CPU che la porta può accettare un nuovo dato.

Il **procedimento di handshake in output** si basa sui seguenti punti:

1. La CPU, se INT\_O = 1, gestisce quando possibile la richiesta di interrupt scrivendo nella porta tramite l'invio del dato in D[7..0].
2. Una volta che la scrittura è terminata la porta attiva OBF e mette il dato in D\_OUT[7..0].
3. Quando possibile UE legge il dato scritto dalla CPU e attiva ACK.

Le **forme d'onda dei segnali** durante il protocollo di handshake sono le seguenti:



Segnali del protocollo di handshake per una porta di output.

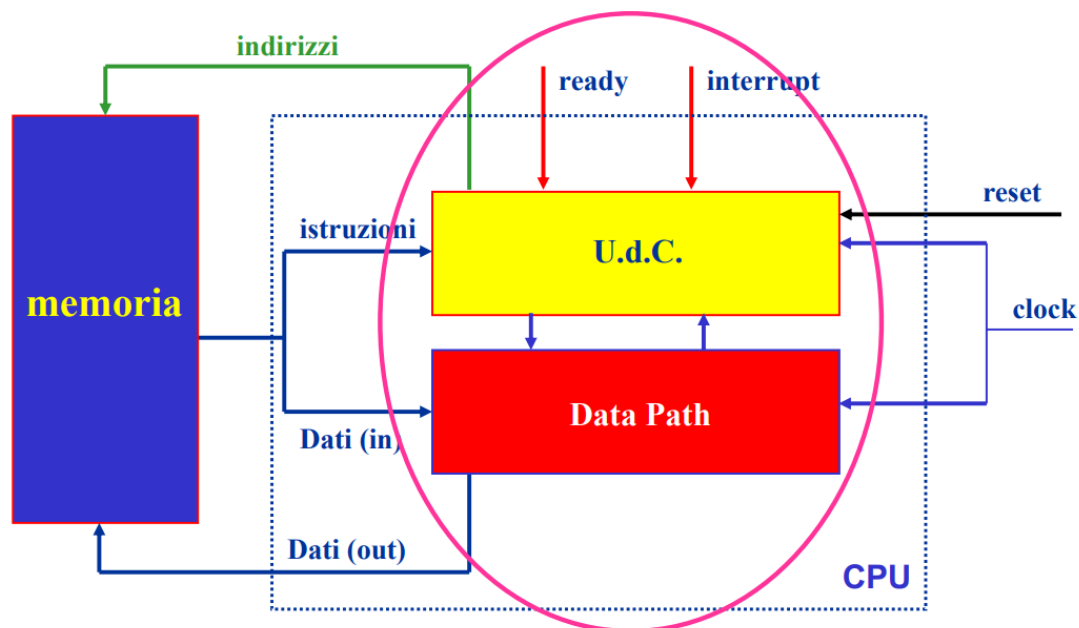
## ▼ 6.0 - DLX sequenziale

### ▼ 6.1 - Struttura del DLX

#### Struttura di una CPU

La struttura di una qualsiasi CPU può essere divisa in due blocchi: l'**unità di controllo** e il **datapath**.

A questi due componenti viene affiancata una **memoria esterna** sulla quale risiedono il programma e i dati.



Struttura di una CPU.

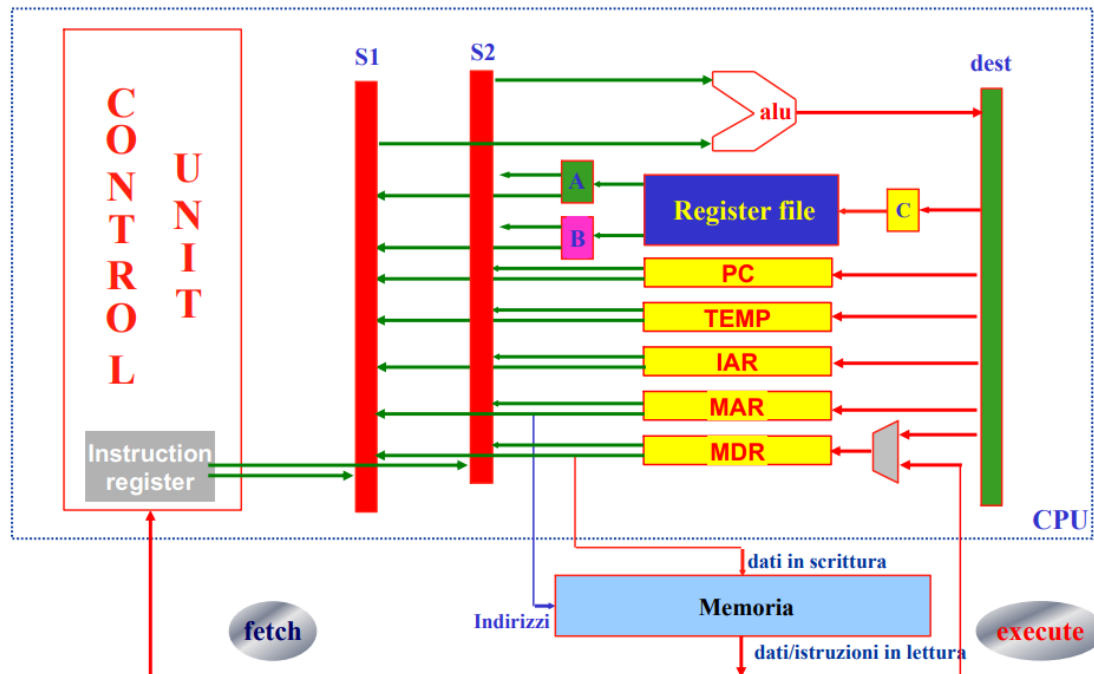
L'**unità di controllo** consiste in una RSS che ad ogni ciclo di clock invia un insieme di segnali al datapath che specificano una determinata **micro-operazione** da dover eseguire, ovvero un'operazione che verrà eseguita all'interno del datapath in un singolo ciclo di clock. Ogni istruzione appartenente all'ISA è eseguita mediante una successione di micro-operazioni.

Il **datapath** contiene invece tutte le unità di elaborazione ed i registri necessari per l'esecuzione delle micro-operazioni dettate dall'unità di controllo.



## Struttura del DLX

La **struttura logica del DLX** è la seguente:



### ▼ 6.2 - Datapath del DLX

## Datapath del DLX

### Registri del DLX

I **registri** che fanno parte del DLX sono i seguenti:

- **Register File:** contiene 32 registri general purpose R0, ..., R31 con R0 = 0.
- **IR (Instruction Register):** contiene l'istruzione attualmente in esecuzione.
- **PC (Program Counter):** contiene l'indirizzo della prossima istruzione da eseguire.
- **TEMP (Temporary Register):** può contenere risultati temporanei utili per le operazioni.

- **IAR (Interrupt Address Register)**: contiene l'indirizzo di ritorno in caso di interruzione.
- **MAR (Memory Address Register)**: contiene l'indirizzo del dato da scrivere o leggere in memoria.
- **MDR (Memory Data Register)**: contiene dati in transito da e per la memoria.
- **A e B**: sono i registri di uscita dal Register File.

Ogni registro campiona sul fronte di salita del clock e hanno:

- Due porte di uscita **O1** e **O2** (oppure **A** e **B** per il Register File) per connettersi ai bus S1 e S2.
- Un ingresso di controllo **WE** per scrivere all'interno del registro.
- Due ingressi di controllo **OE1** e **OE2** uno per ogni bus S1 e S2.

### Operazioni e flag della ALU

Le **operazioni** che può eseguire la ALU sono le seguenti:

Dest (uscite) - 4 bit di comando	
S1 + S2	
S1 - S2	
S1 and S2	
S1 or S2	
S1 exor S2	
Shift S1 a sinistra di S2 posizioni	
Shift S1 a destra di S2 posizioni	
Shfit S1 aritmetico a destra di S2 posizioni	
S1	
S2	
0	
1	

I **flag** che vengono rilasciati dalla ALU sono:

- Zero.
- Segno negativo.

- Carry.

### Frequenza massima del datapath

Per valutare la frequenza massima a cui è possibile far funzionare il datapath occorre introdurre le seguenti definizioni:

- $T_C(max)$ : ritardo massimo tra il fronte positivo del clock e l'istante in cui i segnali di controllo generati dall'unità di controllo sono validi.
- $T_{OE}(max)$ : ritardo massimo tra l'arrivo del segnale OE e l'istante in cui i dati del registro sono disponibili sul bus.
- $T_{ALU}(max)$ : ritardo massimo della ALU.
- $T_{SU}(min)$ : tempo di set-up minimo dei registri.

Abbiamo dunque che il tempo di clock minimo sarà dato dalla somma di tutte queste temporizzazioni, e la frequenza massima sarà il reciproco del tempo di clock minimo:

$$T_{CK}(min) = T_C(max) + T_{OE}(max) + T_{ALU}(max) + T_{SU}(max)$$

$$f_{CK}(max) = \frac{1}{T_{CK}(min)}$$

## ▼ 6.3 - Unità di controllo del DLX

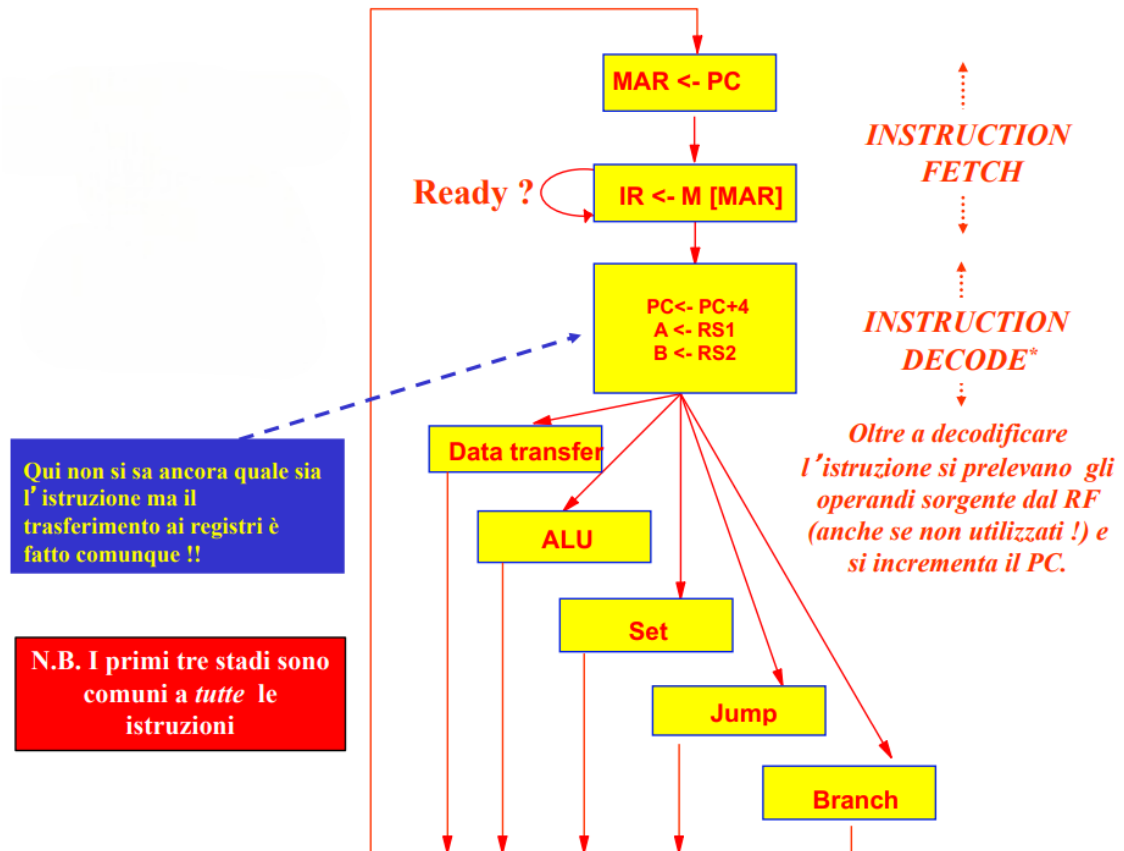
### Unità di controllo del DLX

L'**unità di controllo (controller)** di una CPU viene progettata in seguito alla definizione del set di istruzioni e del datapath.

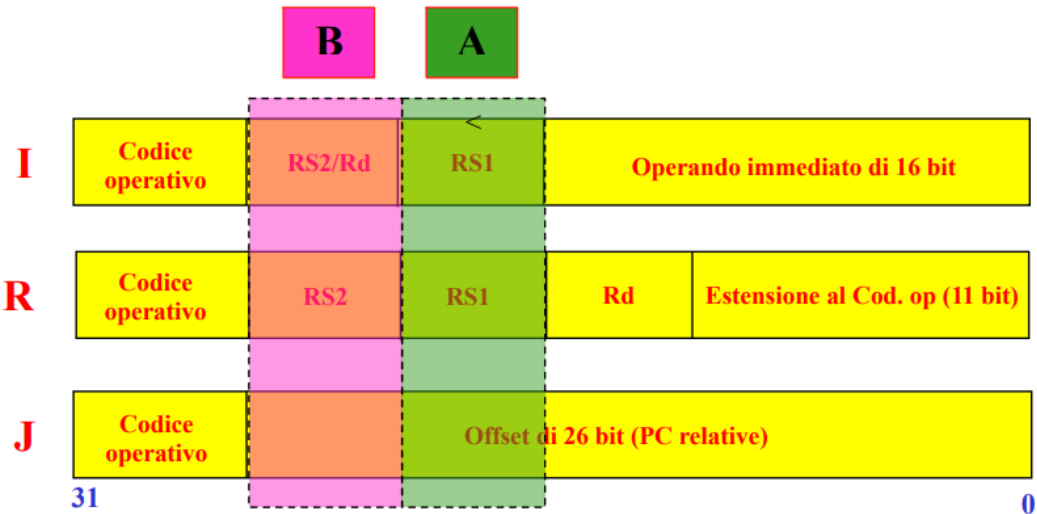
Il suo funzionamento può essere specificato tramite un **diagramma degli stati**. Il controller permane infatti in uno stato per un ciclo di clock e transita da uno stato all'altro in corrispondenza dei fronti del clock. Il diagramma degli stati descrive dunque le micro-operazioni che il datapath deve eseguire.

#### Il diagramma degli stati del controller

Il **diagramma degli stati del controller** più generale è il seguente:



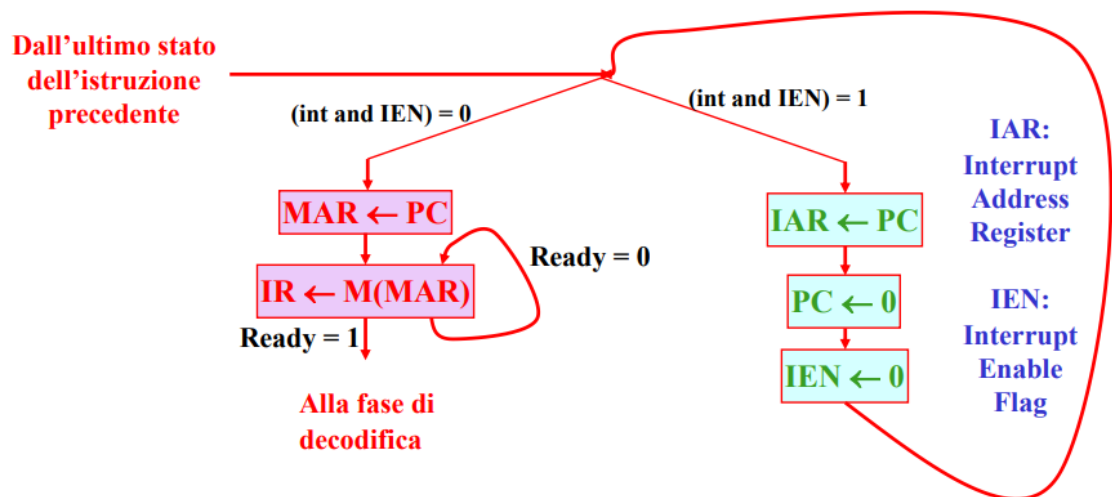
Come possiamo notare dal diagramma, durante la fase di decode vengono utilizzati 5 + 5 bit di istruzione per estrarre preventivamente dal Register File due registri da mettere in A e B. Queste micro-operazioni vengono fatte senza conoscere ancora che tipologia di operazione è stata letta dalla memoria, dunque può accadere che, ad esempio per le istruzioni di jump, tale estrazione dei registri sia stata inutile, ma vale comunque la pena farla in quanto questa casistica è poco frequente.



Estrazione preventiva dei registri dalla codifica delle istruzioni.

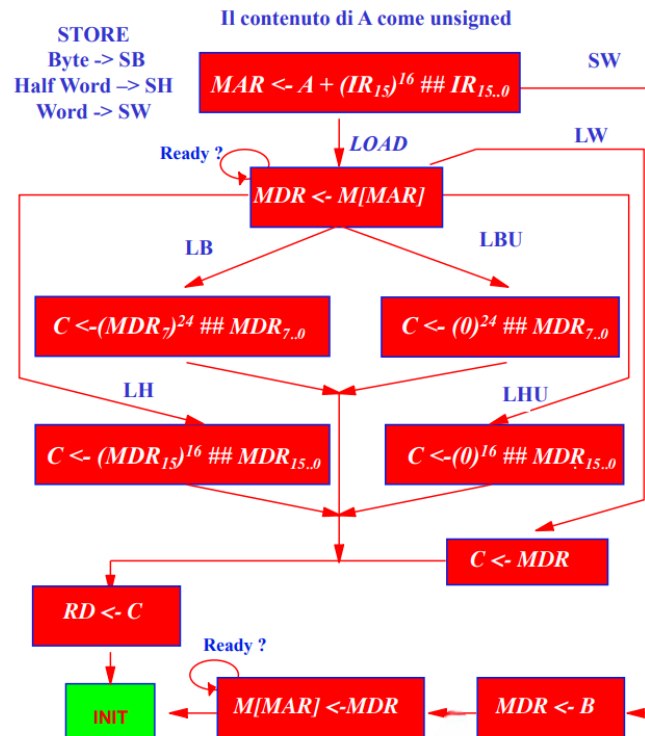
### Fase di fetch

La **fase di fetch** del diagramma degli stati può in realtà essere di due tipologie, una svolta nel caso in cui si presenta un interrupt e l'altra altrimenti. Si controlla dunque se l'interrupt è presente e può essere servito ( $IEN = \text{true}$ ), e in tal caso si esegue l'istruzione di chiamata a procedura all'indirizzo 0, e si salva l'indirizzo di ritorno nell'apposito registro IAR, altrimenti, nel caso in cui l'interrupt non è presente o le interruzioni non sono abilitate, si va a leggere in memoria la prossima istruzione da eseguire.



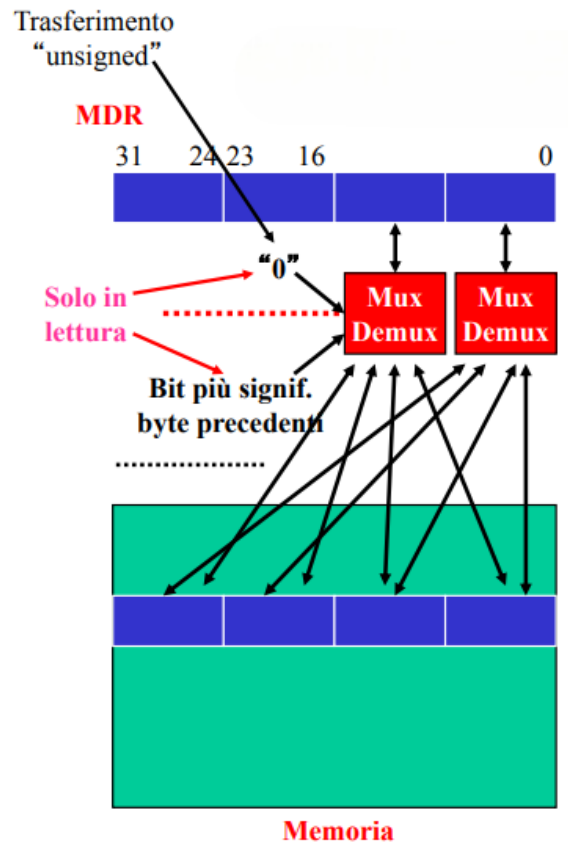
Gli stati della fase di fetch.

## Diagramma degli stati per le istruzioni di Data transfer



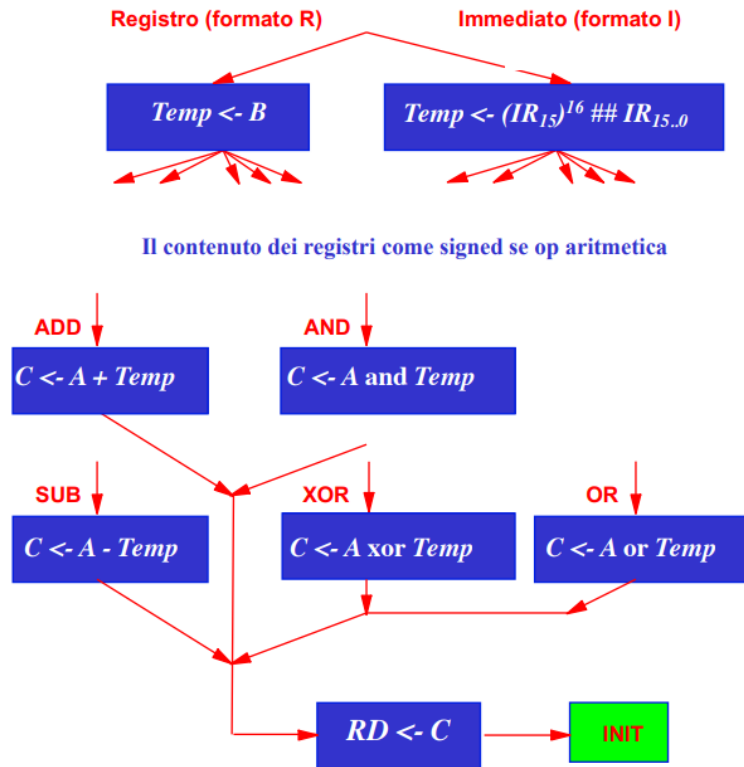
Occorre notare che durante le operazioni di Data transfer vengono effettuati degli spostamenti dei dati tra registri e memorie, i quali però spesso non sono **allineati**. Per questo motivo vengono posti dei **Mux/Demux** tra i registri e le memorie al fine di indirizzare in maniera allineata i byte. Ricordiamo inoltre che esistono nei vincoli nello spostamento dei dati:

- I trasferimenti di byte sono sempre considerati allineati, qualunque sia l'indirizzo di partenza e di arrivo.
- I trasferimenti di half-word possono avvenire solamente a indirizzi multipli di 2.
- I trasferimenti di word possono avvenire solamente a indirizzi multipli di 4.



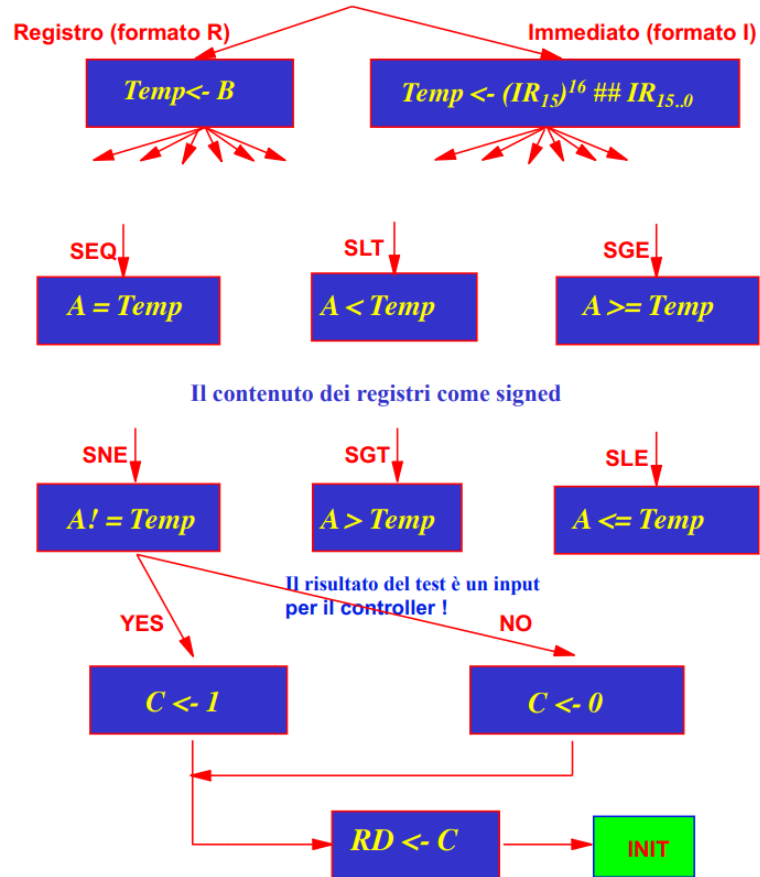
Mux e Demux per l'allineamento dei dati durante il trasferimento tra registri e memorie.

## Diagramma degli stati per le istruzioni ALU

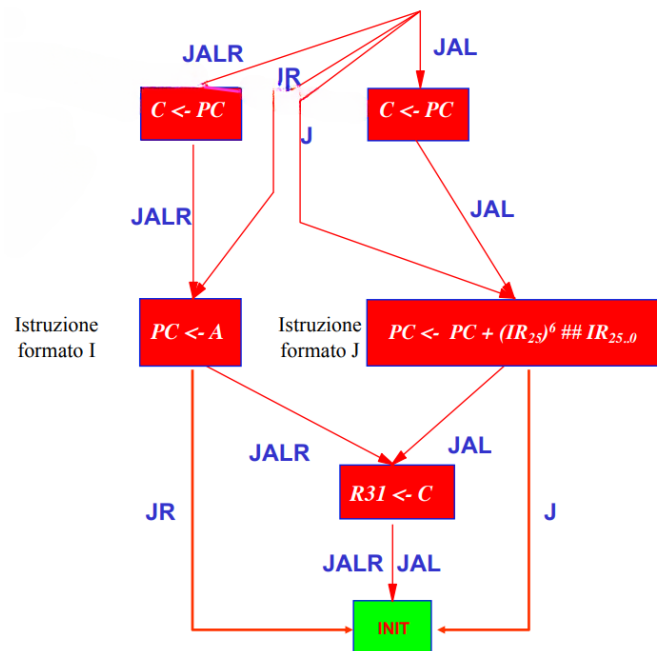


**Diagramma degli stati per le istruzioni di Set**



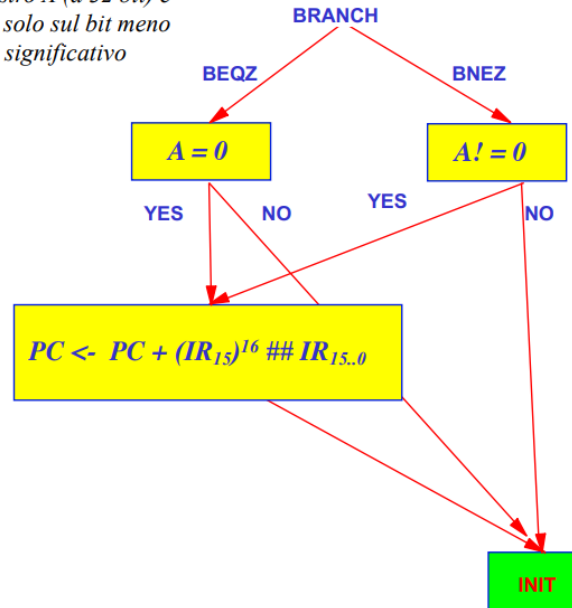


## Diagramma degli stati per le istruzioni di JUMP



## Diagramma degli stati per le istruzioni di BRANCH

*Il controllo se 0 (o !=0)  
è fatto sull'intero  
registro A (a 32 bit) e  
non solo sul bit meno  
significativo*



### ▼ 7.0 - DLX pipelined

#### ▼ Introduzione al DLX pipelined

### Calcolo del *CPI* del DLX pipelined

Oltre al tempo di clock minimo, per valutare la **velocità di una CPU** occorre tenere in considerazione anche il *CPI*, ovvero il numero medio di cicli di clock che servono per completare un'istruzione. Questo numero si basa su due fattori, ovvero il numero di cicli di clock necessari per completare tutte le differenti operazioni possibili e la quantità in percentuale delle diverse operazioni che vengono effettuate. Per questo motivo per effettuare i benchmark si utilizza solitamente un codice di esempio che viene eseguito su diverse CPU, in quanto codici con operazioni diverse non porterebbero a risultati comparabili.

Valutiamo ora il *CPI* del DLX sequenziale. Inseriamo in una tabella il numero di cicli di clock necessari per eseguire le differenti tipologie di operazioni.

Istruzione	Cicli	Wait	Totale
Load	6	2	8
Store	5	2	7
ALU	5	1	6
Set	6	1	7
Jump	3	1	4
Jump and link	5	1	6
Branch (taken)	4	1	5
Branch (not taken)	3	1	4

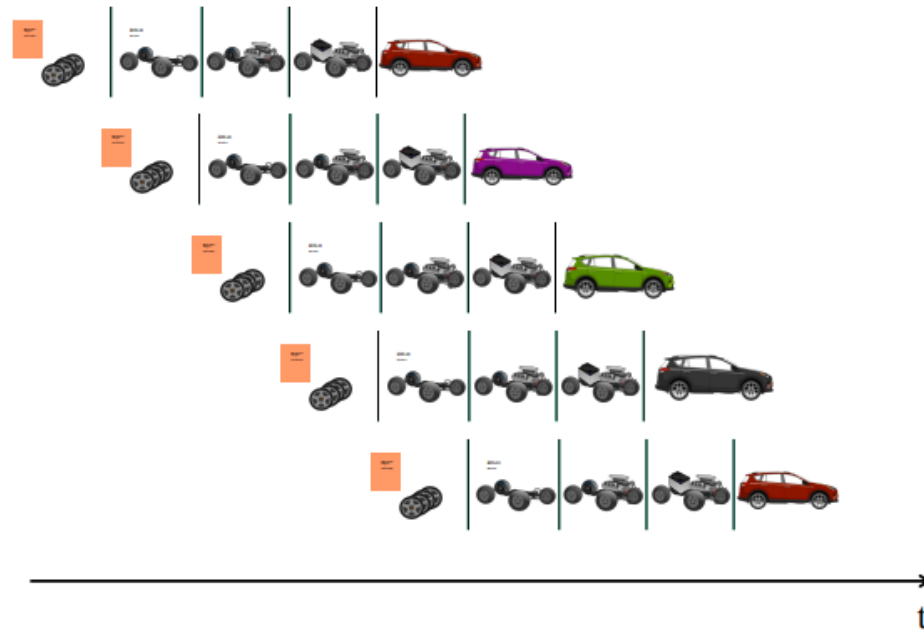
Il *CPI* si calcola nel seguente modo:

$$CPI = \sum_{i=1}^n (CPI_i * \frac{N_i}{\text{numero totale di istruzioni}})$$

Su un codice generico con le seguenti percentuali per tipologia di operazione, Load: 21%, Store: 12%, ALU: 37%, Set: 6%, Jump: 2%, Branch (taken): 12%, Branch (not-taken): 11%, il *CPI* è di 6.3, il che significa che per completare un'operazione il DLX impiega in media 6.3 cicli di clock. Questo numero è decisamente elevato per una CPU, in quanto porterebbe a una lentezza generale di esecuzione molto elevata. Per questo motivo viene introdotta una nuova tipologia di processore basato sul concetto di pipeline, il quale ha un *CPI* minore.

## Introduzione al DLX pipelined

Il concetto del **pipelining** deriva dal mondo delle industrie, nelle quali c'era la necessità di aumentare il **throughput**, ovvero la frequenza con la quale vengono completate le attività, al fine di migliorare la produttività e l'efficienza. Per fare ciò venne dunque introdotto il concetto di **catena di montaggio**:



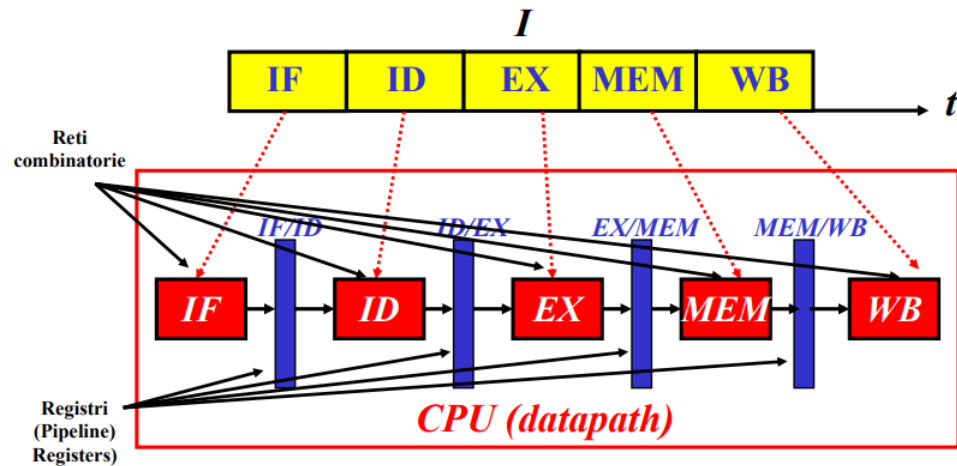
Catena di montaggio.

Tale concetto venne introdotto anche nel mondo delle CPU in quanto gran parte di esse, come il DLX, si basano sui seguenti **passi o stadi di esecuzione delle istruzioni**:

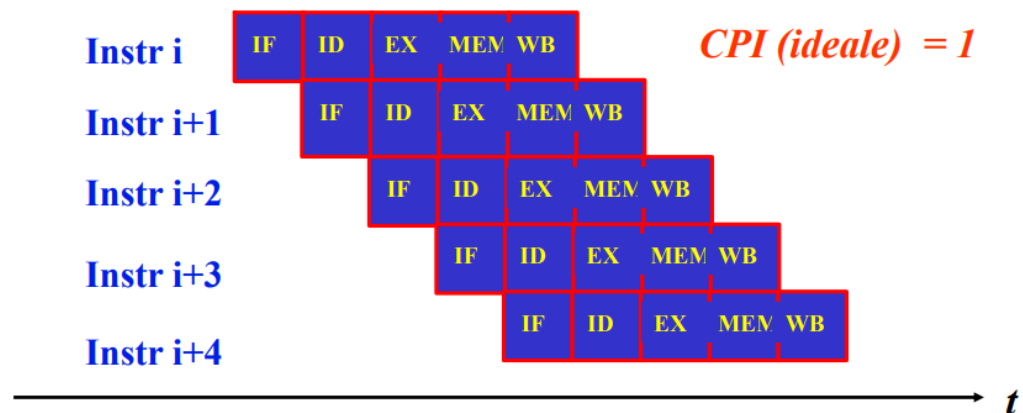
- **Fetch**: l'istruzione viene prelevata dalla memoria e posta in IR.
- **Decode**: l'istruzione in IR viene decodificata e vengono prelevati gli operandi sorgente dal Register File.
- **Execute**: elaborazione aritmetica o logica mediante la ALU.
- **Memory**: accesso alla memoria e, nel caso di BRANCH, aggiornamento del PC (branch completion).
- **Write-Back**: scrittura sul Register File.

Per questo motivo è dunque possibile suddividere la CPU parti distinte, ognuna addetta a uno specifico passo di esecuzione, e svolgere più istruzioni consecutive in contemporanea al fine di aumentare il throughput. Lo svolgimento di ogni operazione passa dunque per tutti i passi di esecuzione, i quali impiegano un singolo ciclo di clock per essere eseguiti,

e nel passaggio da un passo al successivo le informazioni utili vengono salvate all'interno di registri appositi chiamati **pipeline registers**.



L'esecuzione di istruzioni nel DLX pipelined si presenta dunque nel seguente modo:



Notiamo che un lato negativo dell'utilizzo del pipeline è quello dell'aumento del tempo di clock minimo  $T_{CK}(min)$ , in quanto questo diventa la somma del tempo necessario allo stadio più lento per completare l'operazione e dei due tempi necessari rispettivamente per prendere le informazioni necessarie dai registri a monte ed effettuare il set up dei registri a valle una volta terminato lo stadio, ovvero:

$$T_{clk} = T_d + T_P + T_{su}$$

*Clock Cycle*      *Ritardo registro a monte*      *Ritardo stadio combinatorio più lento*      *Set-up registro a valle*

Nonostante ciò il tempo perso dall'aumento del tempo di clock è molto inferiore rispetto al tempo guadagnato dall'aumento della throughput, dunque l'utilizzo del pipeline aumenta l'efficienza generale.

## Requisiti per l'implementazione del DLX pipelined

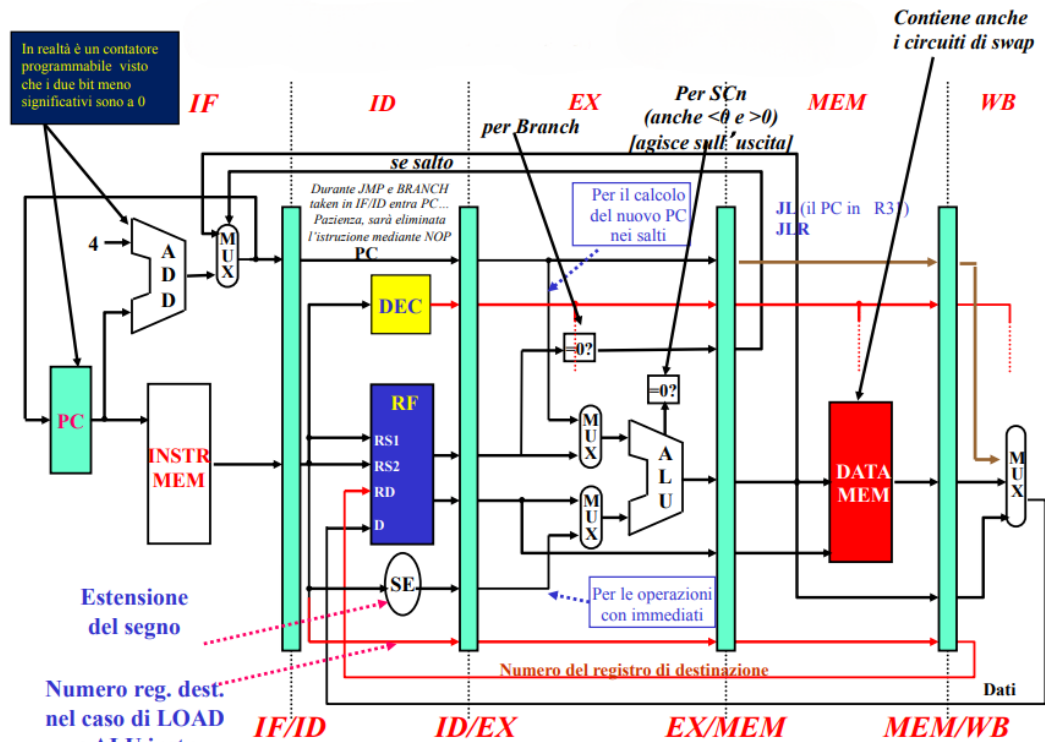
Per l'implementazione del DLX pipelined occorre rispettare i seguenti **requisiti**:

- Il PC deve essere **incrementato durante la fase di fetch** invece che di decode, dunque è necessario introdurre un adder nello stadio IF.
- Sono necessari **due MDR** (che chiameremo **LMDR** e **SMDR**) per gestire il caso di un'operazione Load seguita immediatamente da una Store che porterebbe ad avere due dati in attesa di essere scritti, uno in memoria e uno nel Register File.
- È necessario avere **due memorie**, una per le istruzioni (**Instruction Memory**) e una per i dati (**Data Memory**), come previsto dalla cosiddetta architettura Harvard, in quanto nello stesso ciclo di clock possono essere fatti due accessi in memoria, uno durante lo stadio IF per effettuare il fetch dell'istruzione e uno durante lo stadio MEM per memorizzare un dato in memoria. Siccome abbiamo visto che il tempo di clock minimo è determinato dallo stadio più lento della pipeline è preferibile utilizzare delle memoria cache per Instruction Memory (IM) e Data Memory (DM).

### ▼ Datapath del DLX pipelined

## Datapath del DLX pipelined

Analizziamo ora il **datapath del DLX pipelined**.



Datapath del DLX pipelined.

- **IF**

Nello stadio di Instruction Fetch viene utilizzato il contenuto di PC per andare in Instruction Memory e leggere la prossima istruzione da eseguire.

È presente inoltre un adder, che nella realtà è un semplice contatore programmabile, che effettua l'operazione  $PC + 4$  e mette il risultato in un Mux assieme al risultato della ALU al fine di valutare il caso di un possibile salto. L'uscita del Mux viene poi collegata all'ingresso del PC al fine di modificarlo al prossimo clock.

- **ID**

Nello stadio di Instruction Decode avviene la decodifica dell'istruzione letta nello stadio precedente. Vengono dunque utilizzati i bit dell'istruzione per prendere dal Register File i giusti registri da essere poi messi, nello stadio successivo, all'ingresso della ALU. Nella stessa fase viene esteso di segno l'eventuale immediato presente nell'istruzione.

- **EX**

Nello stadio di Execute vengono connessi i due corretti operandi da inserire all'ingresso della ALU, i quali vengono scelti tramite dei Mux tra S1, S2, il risultato del Mux proveniente dallo stadio IF (per il caso di salti in cui occorre calcolare il valore dell'indirizzo di destinazione) e l'immediato esteso di segno. Inoltre viene controllato se S1 è  $\neq 0$  per il caso di salti condizionati, e il risultato di tale controllo viene inviato al Mux nello stadio di IF.

- **MEM**

Nello stadio di MEM viene connesso il risultato della ALU all'ingresso della Data Memory come indirizzo con il quale leggere o scrivere nella memoria in caso di Load o Store.

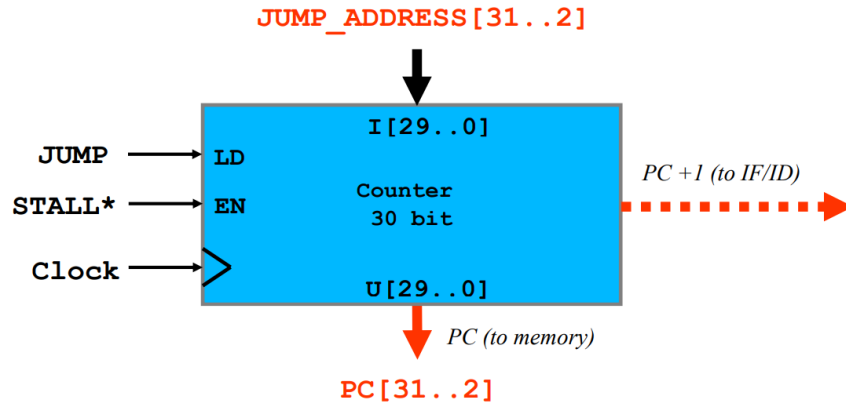
- **WB**

Nello stadio di WB viene utilizzato un Mux con in ingresso il risultato della ALU (per operazioni aritmetico logiche che vanno salvate in un registro), il contenuto di  $PC + 1$  (nel caso di salti in cui il contenuto di  $PC + 1$  viene salvato in R31 come indirizzo di ritorno) e del risultato della lettura dalla Data Memory nello stadio precedente (nel caso di Load) per scegliere quale dato inviare al File Register per effettuare il salvataggio in un registro in caso di operazioni che richiedono un Write Back.

### **Stadio di fetch con contatore**

Al posto dell'adder e del Mux in figura nello stadio di fetch viene utilizzato un **contatore a 30 bit**, in quanto i due bit meno significativi dell'indirizzo sono superflui visto che il DLX esegue il fetch ad indirizzi multipli di 4.

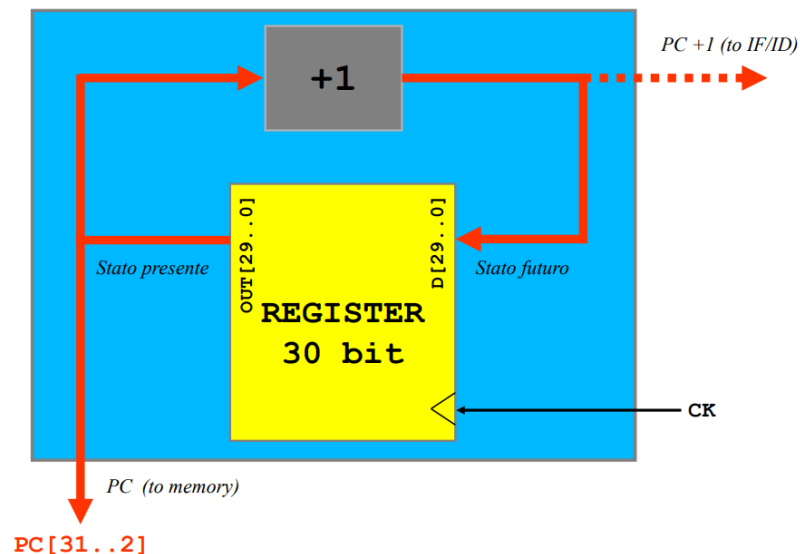




Notiamo che come enable viene inserito il segnale di STALL negato, il quale viene generato dall'unità di controllo quando lo stadio IF deve essere bloccato.

Come load viene invece inserito il segnale di JUMP, proveniente dallo stadio MEM, il quale segnala la presenza di un salto e in tal caso memorizza nel counter il contenuto di JUMP\_ADDRESS[31..2], anch'esso proveniente dallo stadio MEM.

Notiamo inoltre che per inviare agli stadi IF/ID  $PC + 1$  la logica che viene utilizzata è la seguente.

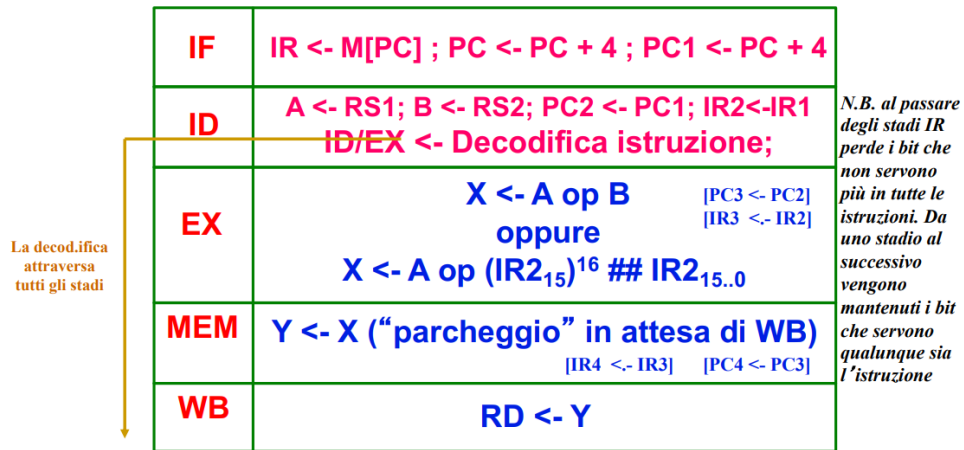


## Esecuzione in pipeline di una istruzione

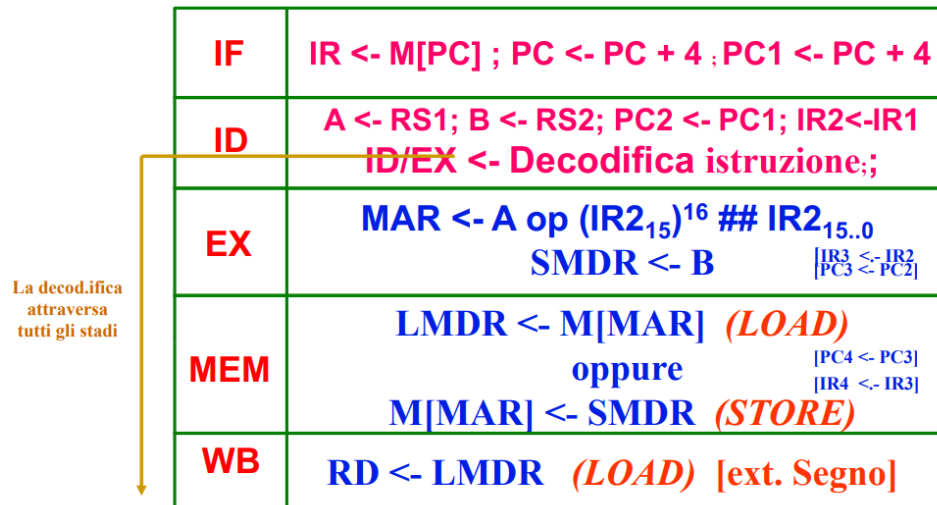
Vediamo ora le operazioni che vengono effettuate in ogni stadio della pipeline per ogni tipologie di istruzione:

- Istruzione **ALU**

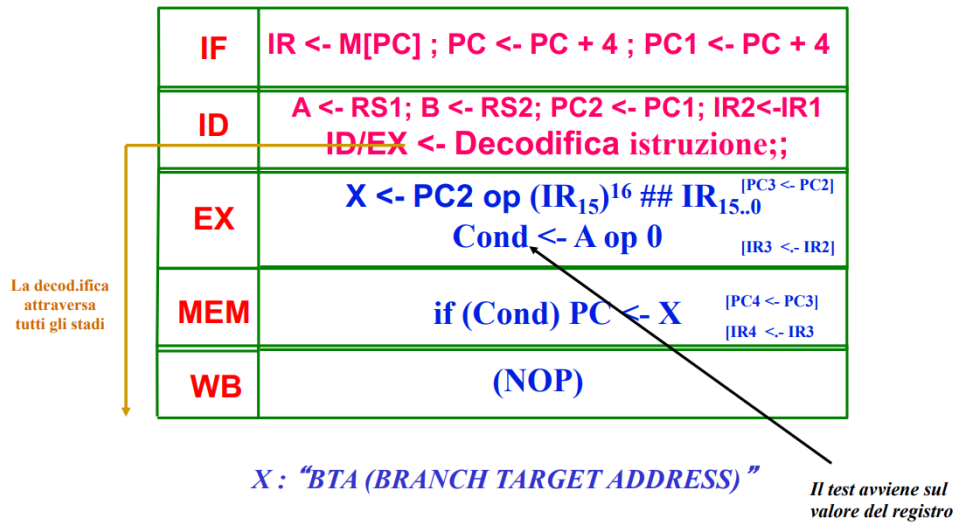
*NB in questa come nelle altre istruzioni RD (RS2) è trasferita fino allo stadio WB*



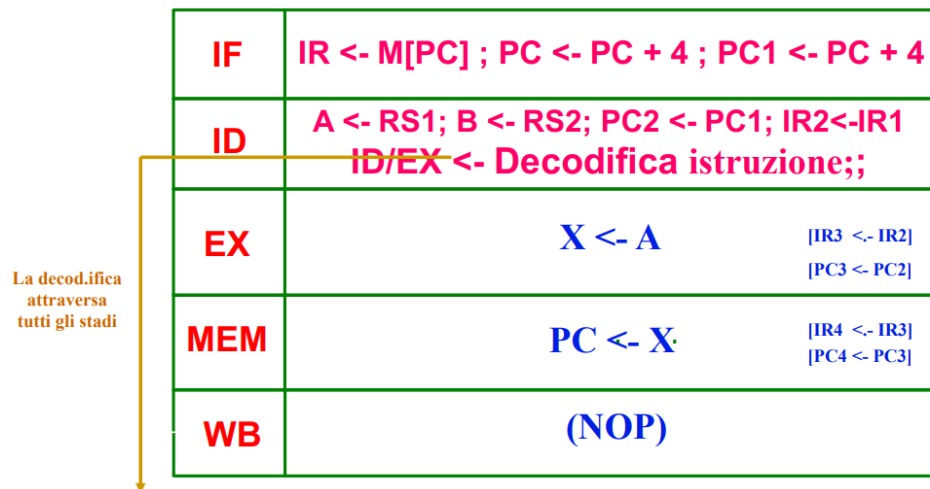
- Istruzione **MEM**



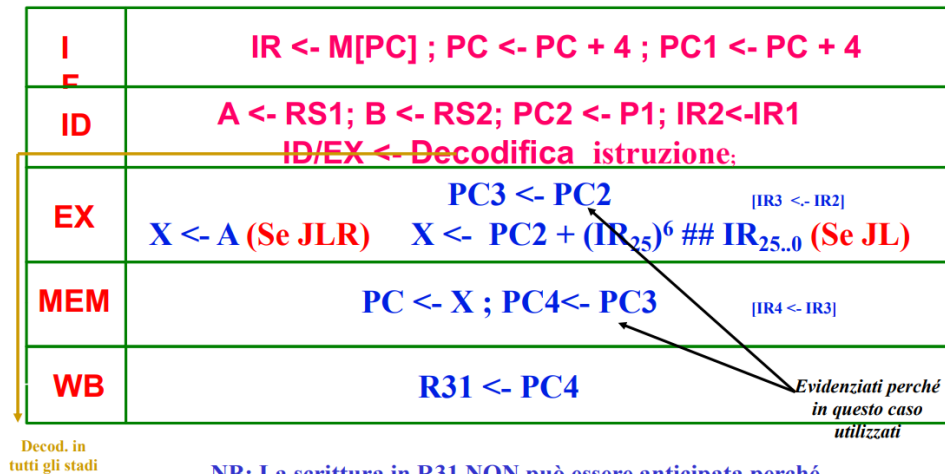
- Istruzione **BRANCH**



- Istruzione **JR**



- Istruzione **JL/JLR**



**NB:** La scrittura in R31 NON può essere anticipata perché potrebbe sovrapporsi ad altra scrittura di registro

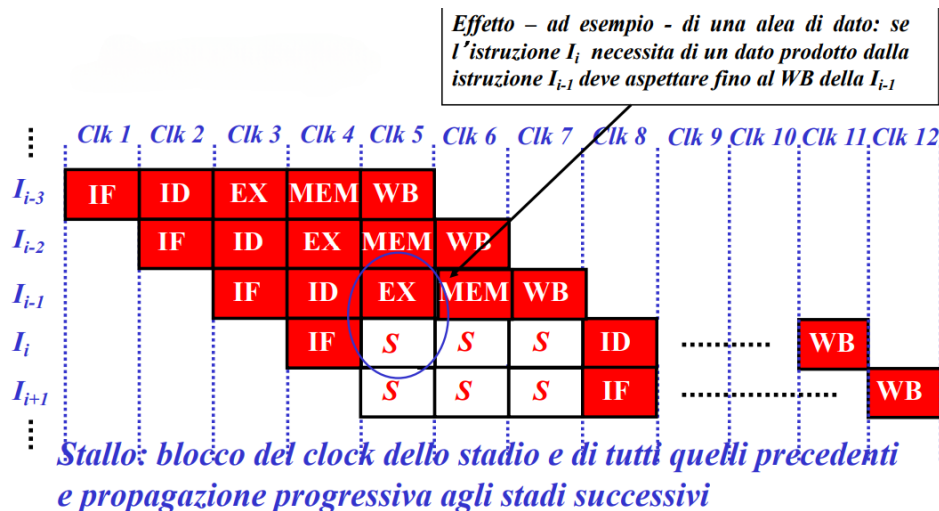
## ▼ Alee nella pipeline

### Introduzione alle alee nella pipeline

Si verifica una situazione di **alea** quando in un determinato ciclo di clock un'istruzione presente in uno stadio della pipeline non può essere eseguita in quel clock per diversi motivi. A seconda di questi le alee vengono divise in:

- Alee **strutturali**: una stessa risorsa è condivisa tra due stadi della pipeline, in questo caso tali stadi non possono essere eseguiti contemporaneamente.
- Alee di **dato**: dovute a dipendenze fra le istruzioni. Ad esempio può capitare quando un'istruzione legge un dato da un registro scritto dall'istruzione precedente.
- Alee di **controllo**: le istruzioni successive ad un'istruzione BRANCH dipendono dal risultato di tale BRANCH (taken/not taken), dunque non possono essere caricate con certezza sulla pipeline.

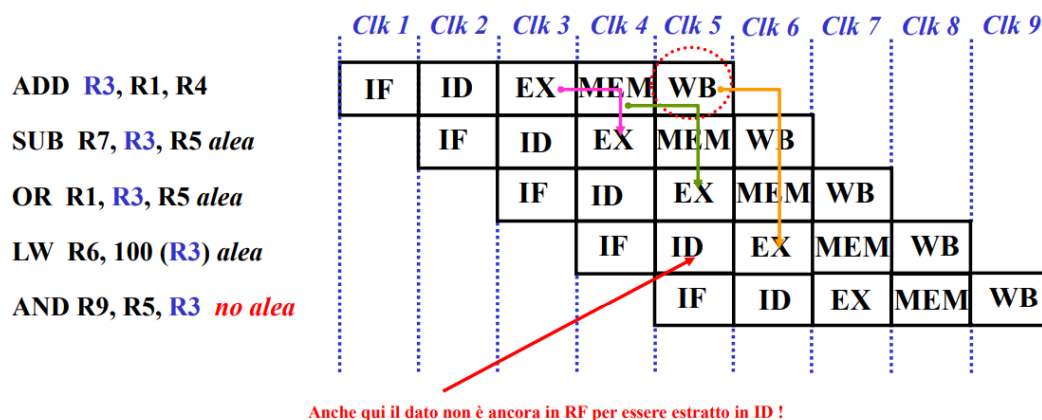
Nel caso in cui si verifica un'alea l'istruzione che non può essere eseguita viene bloccata assieme a tutte le istruzioni che la seguono, mentre le istruzioni precedenti avanzano normalmente.



Nel  $CPI$  effettivo si deve dunque tenere conto anche del numero di stalli che si verificano nell'esecuzione del codice. Per diminuire il  $CPI$  è utile dunque evitare o ridurre al minimo le alee, e questo si può fare utilizzando diverse tecniche.

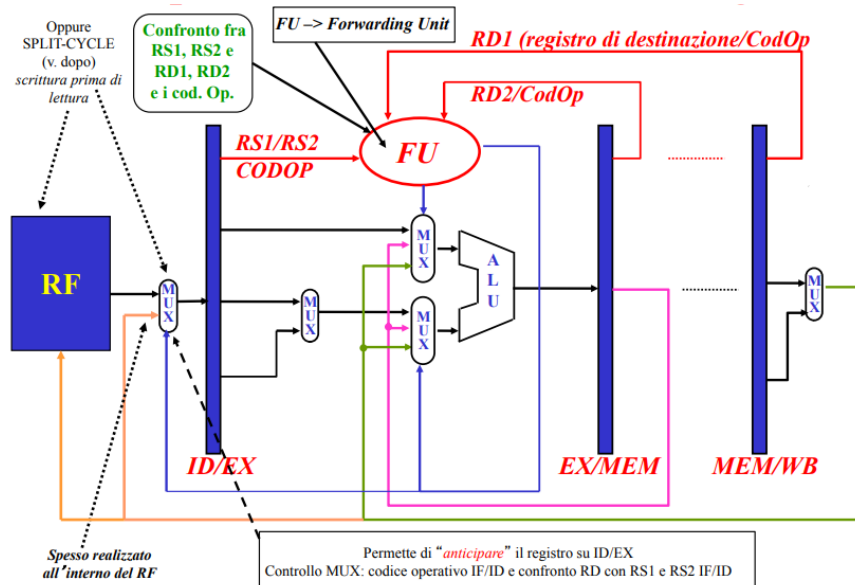
## Forwarding

La tecnica del **forwarding** consente di evitare le alee di dato, in quanto consiste nel propagare il risultato dello stadio EX agli stadi precedenti per fare in modo che le istruzioni successive abbiano il dato aggiornato con cui poter lavorare nello stadio di EX senza dover aspettare che la prima istruzione termini lo stadio di WB.

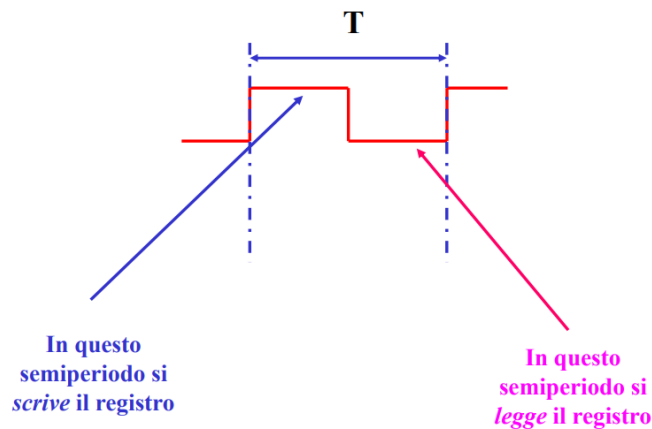


Forwarding.

Nel DLX il forwarding viene realizzato connettendo i registri destinazione presenti negli stadi MEM e WB e il codice operativo dell'istruzione ad un'unità chiamata **Forwarding Unit** presente nello stadio EX, la quale controlla se tali registri sono uguali ai registri sorgente da inserire come ingresso della ALU. In caso affermativo viene connesso all'ingresso della ALU il contenuto del registro destinazione, il quale è aggiornato.



## Split-cycle



Un'alea di dato che il forwarding **non** può risolvere è quella causata da un'istruzione di Load. Questa infatti legge il dato dalla memoria durante lo stadio MEM, ma non all'inizio dello stadio in quanto prima deve essere

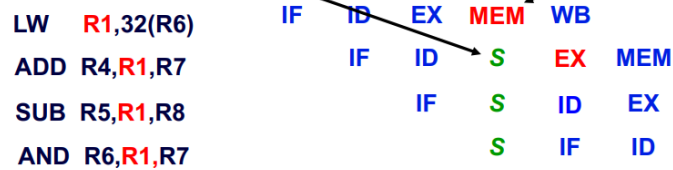
effettuato l'accesso in memoria, dunque all'inizio dello stadio non si ha ancora il dato pronto da propagare all'istruzione successiva nello stadio EX. In questo caso occorre stallare la pipeline, ovvero bloccare di un clock le istruzioni successive a quella di Load.

*Di fatto non viene generato il clock.  
Il blocco di un clock si propaga  
lungo la pipeline uno stadio alla  
volta.*



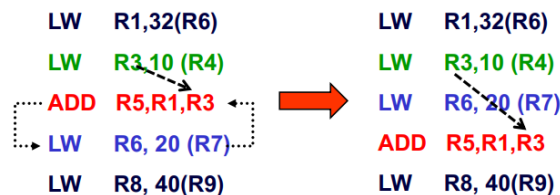
**E' necessario stallare la  
pipeline**

*Dalla fine di  
questo stadio in  
poi normale  
forwarding*



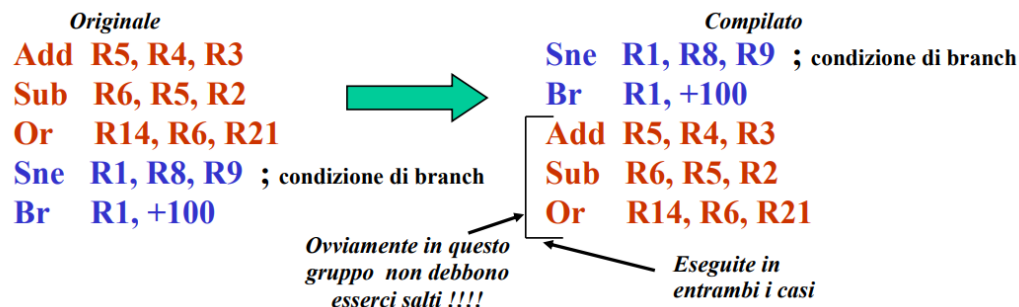
Stallo della pipeline causato da un'alea di dato.

In alcuni casi lo stallo della pipeline causato da una tale alea non viene gestito via hardware ma **via software dal compilatore**, scambiando l'istruzione successiva a quella di Load con un'altra successiva, oppure, nel caso in cui ciò non sia possibile, inserendo l'istruzione NOP, la quale non fa nulla.



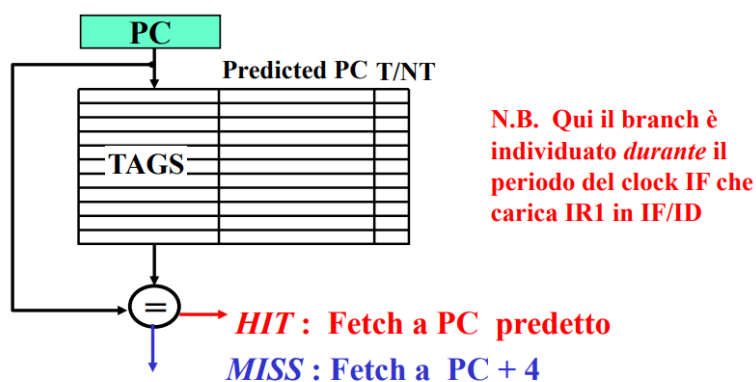
## Delayed Branch o Dynamic Prediction

Il **Delayed Branch** è una tecnica utilizzata per gestire alee di controllo che consiste nello scambiare via software alcune istruzioni al fine di evitare lo stallo della pipeline a seguito di un'istruzione di Branch o Jump, oppure, nel caso in cui ciò non sia possibile, asserendo le istruzioni successive come NOP.



Delayed Branch.

Un'altra tecnica per gestire alee di controllo è quella della **Dynamic Prediction**, la quale utilizza un **Branch Target Buffer**, ossia un record di tutti i salti precedenti in cui viene memorizzato l'indirizzo dell'operazione che contiene un salto e l'indirizzo della prossima istruzione (in modo da capire se il salto è taken/untaken). Così facendo, quando avviene un salto, si controlla se tale istruzione è già presente nel record, e in tal caso si inserisce nella pipeline le istruzioni a partire dall'indirizzo della prossima istruzione presente nel buffer, comportandosi in modo da emulare l'ultima volta in cui l'istruzione è stata eseguita. L'utilizzo di questa tecnica nella realtà varia da CPU a CPU, e solitamente viene tenuta segreta dai produttori in quanto è di grande importanza nel diminuire il *CPI*.



Dynamic prediction.

Notiamo che l'utilizzo della Dynamic Prediction ad **un solo bit**, ovvero che memorizza solo l'ultimo salto come taken/untaken, può non essere molto efficiente nel caso di cicli annidati. Consideriamo ad esempio un ciclo esterno loop1 eseguito 5000 volte e uno interno loop2 eseguito 1000 volte.



Utilizzando 1 solo bit nel Branch Target Buffer ciò comporterebbe che, quando il loop2 termina, la prediction sbaglia dopo non aver sbagliato per 1000 volte consecutive, ma sbaglia di nuovo quando ricomincia il loop2, dunque avvengono **2 errori consecutivi** per 5000 volte. Per evitare ciò è possibile memorizzare in altri bit se il salto è taken/untaken in altre esecuzioni precedenti e così effettuare la prediction con il dato che risulta più preponderante.

```
for (i=0; i<5000; i++)
  for (j=0; j<1000; j++)
  {
      x[i,j] = i*j + i + j;
      ...
      ...
  }
```