

# Calcolatori Elettronici

↗ exams overview

Calcolatori Elettronici T

@Francesca Guzzi

## 0. Fondamenti di reti logiche

- 0.1 Porte logiche di base
- 0.2 Componenti combinatori standard
- 0.3 Reti sequenziali asincrone notevoli

## 1. Caratteristiche dei calcolatori

- 1.1 Modello di riferimento
- 1.2 Componenti principali

## 2. Dispositivi notevoli

- 2.1 Memoria EPROM
- 2.2 Memoria RAM
- 2.3 Integrati notevoli
- 2.4 Registro Edge-Triggered con WE\*
- 2.5 Register File (1 read port, 1 write port)
- 2.6 Esempi utili

## 3. Mapping e decodifica

- 3.1 Mapping di dispositivi da 8 bit in sistemi con bus di dati da 8 bit
- 3.2 Caratteristiche di un dispositivo indirizzabile su finestra di  $n = 2^k$  byte
- 3.3 Mapping allineato
- 3.4 Decodifica completa
- 3.5 Decodifica non allineata
- 3.6 Decodifica parziale
- 3.7 Mapping, read, write e set/reset di un FFD
- 3.7 Mapping, read, write e set/reset di un latch
- 3.8 Memorie con processori a parallelismo >8

## 4. Linguaggio macchina

- 4.1 Codifica binaria delle istruzioni
- 4.2 Linguaggio assembly
- 4.3 Caratteristiche dell'ISA DLX
- 4.4 Formati di istruzioni
- 4.5 Linguaggio assembly DLX
- 4.6 Modalità di accesso alla memoria
- 4.7 Esempi di codice Assembly DLX

## 5. Interruzioni

- 5.1 Interrupt nel DLX
- 5.2 Programmable Interrupt Controller (PIC)

## 6. Periferiche di I/O con handshake

## 7. DLX Sequenziale

- 7.1 Funzioni della ALU
- 7.2 Trasferimento dati sul datapath
- 7.3 Progetto dell'Unità di Controllo
- 7.4 Numero di clock necessari per eseguire le istruzioni
- 7.5 Passi dell'esecuzione delle istruzioni

## 8. DLX Pipelined

- 8.1 Pipelining nella CPU DLX
- 8.2 Datapath in pipeline del DLX

## 0. Fondamenti di reti logiche

Classificazione delle reti logiche:

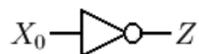
- *Combinatorie*, in cui lo stato d'uscita è funzione del solo stato d'ingresso presente in quell'istante;
- *Sequenziali*, nelle quali l'uscita ad un certo istante dipende sia dallo stato di ingresso *presente* che dagli stati di ingresso *precedenti*;
  - *Asincrone*: l'elaborazione avviene a *flusso continuo* → RSA → base di alcuni componenti fondamentali come latch e flip-flop;
  - *Sincrone*: l'elaborazione avviene ad *istanti discreti* → RSS → quelle che ci interessano e andremo a progettare, sono quelle con il clock.

### 0.1 Porte logiche di base

In generale con  $n$  ingressi si possono creare  $2^{2^n}$  funzioni.

Porte logiche **unarie**: (operano su un solo segnale, 4 funzioni che operano su un solo ingresso)

Porta Logica: NOT



Espressione:

- $Z = \overline{X_0}$
- $Z = \sim X_0$
- $Z = \neg X_0$
- $Z = !X_0$

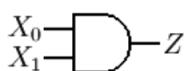
Tabella della verità:

$X_0$	$Z$
0	1
1	0

Descrizione:

L'operatore/porta NOT restituisce il valore inverso di quello in entrata.

Porte logiche **binarie**: (2 ingressi, 16 funzioni)

**Porta Logica: AND****Espressione:**

- $Z = X_0 X_1$
- $Z = X_0 \cdot X_1$
- $Z = X_0 \text{ AND } X_1$

**Tabella della verità:**

$X_1$	$X_0$	$Z$
0	0	0
0	1	0
1	0	0
1	1	1

**Descrizione:**

L'operatore/porta AND (letteralmente e in inglese) restituisce 1 (vero) se e solo se tutti gli operandi hanno valore 1 (vero), altrimenti restituisce 0 (falso). Tale operazione è anche detta prodotto logico.

**Porta Logica: OR****Espressione:**

- $Z = X_0 + X_1$
- $Z = X_0 \text{ OR } X_1$
- $Z = X_0 ? X_1$

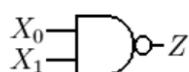
**Tabella della verità:**

$X_1$	$X_0$	$Z$
0	0	0
0	1	1
1	0	1
1	1	1

**Descrizione:**

L'operatore/porta OR (letteralmente o in inglese) restituisce 1 (vero) se almeno uno degli operandi è 1 (vero); ovvero restituisce 0 (falso) se e solo se tutti gli operandi sono 0 (falso). Tale operazione è anche detta somma logica.

Altre porte logiche (oltre queste 3 che sono quelle di base):

**Porta Logica: NAND****Espressione:**

- $Z = \overline{X_0} \overline{X_1}$

**Tabella della verità:**

$X_1$	$X_0$	$Z$
0	0	1
0	1	1
1	0	1
1	1	0

**Descrizione:**

L'operatore NAND (cioè la negazione del risultato dell'operazione AND) restituisce 0 (falso) se e solo se tutti gli elementi sono 1, mentre restituisce 1 (vero) in tutti gli altri casi.

**Porta Logica: NOR****Espressione:**

- $Z = \overline{X_0} + \overline{X_1}$

**Tabella della verità:**

$X_1$	$X_0$	$Z$
0	0	1
0	1	0
1	0	0
1	1	0

**Descrizione:**

L'operatore NOR, (cioè la negazione del risultato dell'operazione OR) restituisce 1 (vero) se e solo se tutti gli elementi sono 0, mentre restituisce 0 (falso) in tutti gli altri casi.

### Porta Logica: XOR



#### Espressione:

- $Z = X_0 \oplus X_1$

### Tabella della verità:

X <sub>1</sub>	X <sub>0</sub>	Z
0	0	0
0	1	1
1	0	1
1	1	0

### Descrizione:

L'operatore XOR (detto anche OR esclusivo o somma modulo 2) restituisce 1 (vero) se e solo se un unico dei due operandi è 1, mentre restituisce 0 (falso) in tutti gli altri casi. Osservando la tabella della verità dell'operatore XOR, si può riscrivere lo stesso XOR utilizzando le porte logiche di base:  
 $Z = X_0 \oplus X_1 = X_0 \bar{X}_1 + \bar{X}_0 X_1$

NB: esistono altri modi equivalenti per scrivere l'espressione

### Porta Logica: XNOR



#### Espressione:

- $Z = \bar{X}_0 \oplus \bar{X}_1$

### Tabella della verità:

X <sub>1</sub>	X <sub>0</sub>	Z
0	0	1
0	1	0
1	0	0
1	1	1

### Descrizione:

L'operatore XNOR (cioè la negazione del risultato dell'operazione XOR) restituisce 0 se e solo se un unico elemento dei due è uguale a 1 e tutti gli altri elementi sono 0. Osservando la tabella della verità dell'operatore XNOR, si può riscrivere lo XNOR stesso utilizzando le porte logiche di base:  
 $Z = X_0 \oplus X_1 = \bar{X}_0 \bar{X}_1 + X_0 X_1$

NB: esistono altri modi equivalenti per scrivere l'espressione

## 0.2 Componenti combinatori standard

### Encoder

L'Encoder ha la funzione di rilevare l'attivazione di una determinata linea d'ingresso e riportare sulle uscite il codice binario dell'entrata corrispondente. Il funzionamento del dispositivo è tale che attivando una delle n linee in ingresso, l'uscita assume una delle m configurazioni possibili, solo a titolo di esempio per riportarci al decoder,  $2^3$  ossia 8 ingressi avranno 3 uscite che in codice binario identificano i numeri da 0 a 7 tra ingresso e uscita non esiste però legame logico come nel decoder perché all'interno dell'encoder esistono delle allocazioni perenni di memoria (memorizzate dal costruttore) tali che il loro numero sia pari alle linee in ingresso (ogni linea attiva individua una locazione di memoria). Se gli ingressi attivati sono più di uno, l'uscita potrebbe assumere una configurazione binaria indesiderata. Per evitare che questo accada, i codificatori in commercio sono "con priorità": se si attiva più di una linea in ingresso, l'uscita assumerà la configurazione associata all'ingresso con più priorità, tra quelli attivati.

Conf.	I <sub>7</sub>	I <sub>6</sub>	I <sub>5</sub>	I <sub>4</sub>	I <sub>3</sub>	I <sub>2</sub>	I <sub>1</sub>	I <sub>0</sub>	A <sub>2</sub>	A <sub>1</sub>	A <sub>0</sub>
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	1	X	0	0	1
2	0	0	0	0	0	1	X	X	0	1	0
3	0	0	0	0	1	X	X	X	0	1	1
4	0	0	0	1	X	X	X	X	1	0	0
5	0	0	1	X	X	X	X	X	1	0	1
6	0	1	X	X	X	X	X	X	1	1	0
7	1	X	X	X	X	X	X	X	1	1	1

Esempio di codificatore ad 8 ingressi e 3 uscite con relativa tabella della verità

### Decoder

La sua funzione è opposta a quella dell'encoder: in base alla combinazione dei bit presenti ai suoi ingressi, attiva una corrispondente combinazione di bit sulle linee di uscita. In generale avendo  $n$  linee di ingresso, viene attivata esclusivamente una delle  $m$  linee di uscita con:  $m \leq 2^n$  e in base a questo il decodificatore viene detto  $n$  a  $m$ .

Esistono vari tipi di decodificatore: BCD-decimale (4 a 10), binario-ottale (2 a 8), binario-esadecimale (4 a 16), codice Gray-decimale, ecc. Per esempio un decodificatore binario-decimale può essere rappresentato con la seguente tabella di verità:

D C B A	0 1 2 3 4 5 6 7 8 9
0 0 0 0	1 0 0 0 0 0 0 0 0 0
0 0 0 1	0 1 0 0 0 0 0 0 0 0
0 0 1 0	0 0 1 0 0 0 0 0 0 0
0 0 1 1	0 0 0 1 0 0 0 0 0 0
0 1 0 0	0 0 0 0 1 0 0 0 0 0
0 1 0 1	0 0 0 0 0 1 0 0 0 0
0 1 1 0	0 0 0 0 0 0 1 0 0 0
0 1 1 1	0 0 0 0 0 0 0 1 0 0
1 0 0 0	0 0 0 0 0 0 0 0 1 0
1 0 0 1	0 0 0 0 0 0 0 0 0 1
1 0 1 0	0 0 0 0 0 0 0 0 0 0
1 0 1 1	0 0 0 0 0 0 0 0 0 0
1 1 0 0	0 0 0 0 0 0 0 0 0 0
1 1 0 1	0 0 0 0 0 0 0 0 0 0
1 1 1 0	0 0 0 0 0 0 0 0 0 0
1 1 1 1	0 0 0 0 0 0 0 0 0 0

In questo caso, solo le prime 10, delle 16 combinazioni possibili sui 4 fili di ingresso, danno luogo ad una corrispondente combinazione sui 10 fili di uscita, le 6 combinazioni successive non danno luogo ad un'uscita, sono ininfluenti.

- **Multiplexer**

Un **multiplexer** o **mux** o **selettor**e è un dispositivo capace di selezionare un singolo segnale elettrico fra diversi segnali in ingresso in base al valore degli **ingressi di selezione**. Esistono multiplexer sia per segnali digitali che per segnali analogici (**amux**).

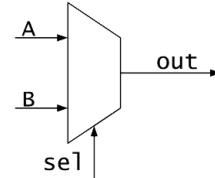
Per esempio, un multiplexer a 2 ingressi è una semplice porta logica la cui uscita Y assume il valore di uno dei due ingressi A o B in base al valore del terzo ingresso di selezione S. L'equazione booleana è:

$$Y = (A \text{ and not } S) \text{ or } (B \text{ and } S)$$

Che può essere espressa dalla seguente tabella di verità:

Questa tabella di verità mostra che quando il selettore è uguale a "0", allora Y è collegato ad A; mentre quando il selettore è uguale ad "1", Y è dipendente da B.

Input			Output
A	B	S	Y
1	1	0	1
1	0	0	1
0	1	0	0
0	0	0	0
1	1	1	1
1	0	1	0
0	1	1	1
0	0	1	0



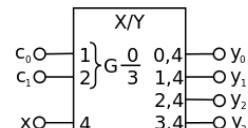
Sono di uso comune multiplexer con molte porte. Per esempio, un multiplexer a otto ingressi può smistare otto diversi segnali, tramite tre segnali logici di selezione. I segnali di ingresso sono numerati da  $X_0$  a  $X_7$ , e gli ingressi di selezione sono numerati  $S_2$ ,  $S_1$  e  $S_0$ . Se  $S_2$  e  $S_0$  sono a '1' e  $S_1$  è a '0', per esempio, l'uscita sarà uguale a  $X_5$ . In genere i multiplexer possono essere implementati con le porte logiche elementari, quali AND, OR, NAND ecc.

Di solito gli ingressi di selezione sono  $n$ , mentre le variabili in entrata sono  $2^n$  o meno.

#### • Demultiplexer

Il dispositivo complementare, il demultiplexer, ha un solo ingresso e diverse uscite. Un demultiplexer è un circuito logico la cui principale funzione è inversa a quella del Multiplexer. Esso è quindi una rete combinatoria con  $k$  ingressi (di selezione) e  $m = 2^k$  uscite, ciascuna delle quali è attiva soltanto in corrispondenza di uno dei  $2^k$  valori di ingresso.

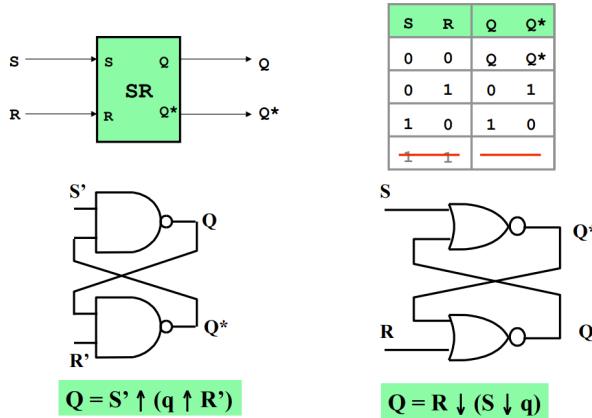
I piedini in basso (che si vedono anche in figura) sono detti di indirizzamento (o ingresso di selezione). In base al valore degli ingressi di selezione, l'ingresso viene collegato a una delle uscite. Per esempio, un demux a otto uscite ha un segnale di ingresso (X), tre ingressi di selezione ( $S_2$ ,  $S_1$  ed  $S_0$ ) e otto uscite (da  $A_0$  a  $A_7$ ). Se per esempio  $S_2$  e  $S_0$  sono a 1 e  $S_1$  è a 0 l'uscita  $A_5$  sarà uguale ad X e tutte le altre uscite saranno messe a 0.



Il demultiplexer ha la funzione esattamente inversa al multiplexer: il multiplexer infatti riunisce più entrate in un'unica uscita mentre il demultiplexer smista un ingresso in più uscite.

## 0.3 Reti sequenziali asincrone notevoli

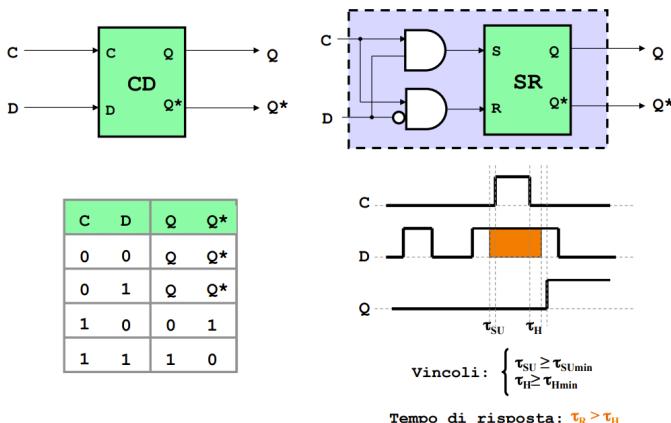
### 1. Latch SR



I comandi di set e reset devono avere una durata minima (vedi datasheet) per consentire il raggiungimento della condizione di stabilità

Dispositivo con due ingressi (set e reset) e la cui tabella presenta una configurazione vietata (set=1 e reset=1). La rete ha due uscite (vera e negata) e la rete funziona in modo tale che se set e reset sono entrambi 0 l'uscita non cambia e mantiene il valore, se set=0 e reset=1 la rete porta l'uscita a 0 (forza un reset della rete); se set=1 e reset=0 viene forzato un set → viene portata l'uscita Q a livello logico 1.

## 2. Latch CD



Latch CD: il problema/vantaggio delle "uscite trasparenti"

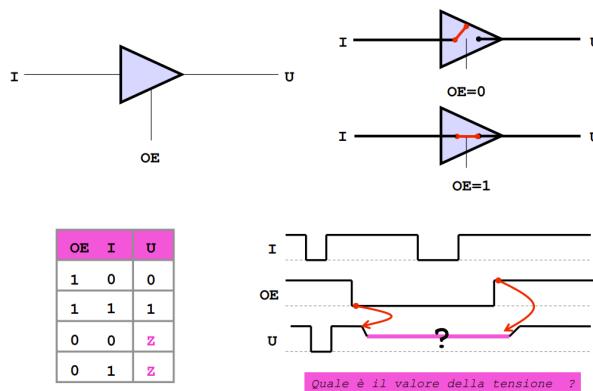
Il latch è un dispositivo di memorizzazione (reti senza CLK). Il latch CD ha due ingressi (C e D, dove C è un segnale di controllo sulla memorizzazione e D un segnale di dato che vogliamo memorizzare).

Se C=1 viene memorizzato sul dispositivo ciò che è presente sull'ingresso D, se C=0 indipendentemente dal valore di D la rete mantiene ciò che ha memorizzato l'ultima volta che C è stato a 1. Il segnale che andiamo a memorizzare non può cambiare in modo libero ma è vincolato ad essere stabile per un tempo di setup prima che arrivi il segnale C e un tempo di hold per la transizione di C da 1 a 0 → vincoli di setup e di hold.

Se C=1 ciò che si presenta sull'ingresso D appare sull'uscita Q (non istantaneamente) → se D cambia con C=1 l'uscita Q segue il segnale D con un certo ritardo, per questo uscite trasparenti.

## 3. Driver 3-state

Dispositivo che di fatto non fa nulla → esiste solo per amplificare/rigenerare un segnale (se deve passare da un livello di tensione a un altro oppure se ha ridotto o perso tensione).

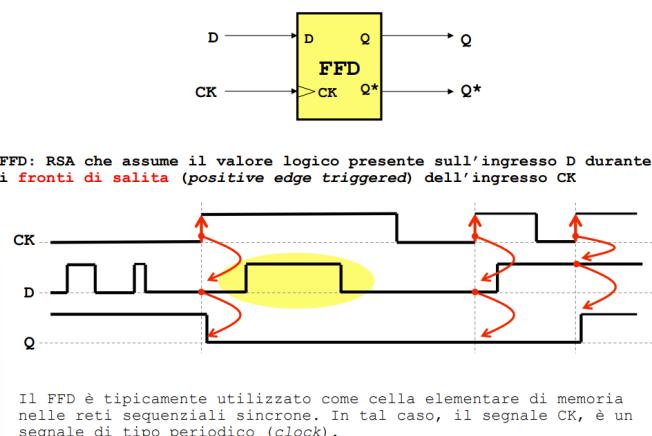


Se OE (output enable) è uguale a 0 la connessione al suo interno è interrotta, se OE=1 quella connessione non è interrotta. Quindi quando OE=1 l'uscita è identica all'ingresso, quando OE=0 l'uscita è z ovvero 3-state → alta impedenza → che valore assume? si può stabilire solo tramite una rete vicina o un disturbo.

Un segnale 3-state che entra in una rete non si propaga sull'uscita, quindi se abbiamo un 3-state in ingresso non comparirà sull'uscita → l'uscita del driver 3-state è un valore indefinito ma assumerà un certo valore comunque (non possiamo determinare se 0 o 1 però)

#### 4. Flip-flop D (delay)

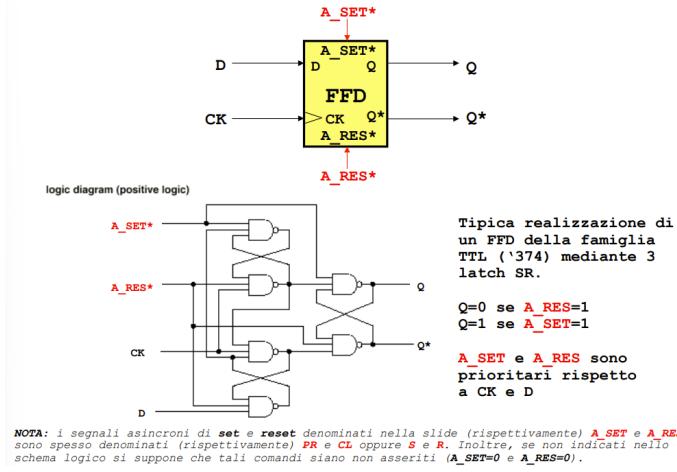
In questo caso abbiamo un CLOCK (il latch memorizza a livello di segnali, in particolare il latch CD se C=1, mentre il flip-flop memorizza a fronte di un CLK). Quando il CLK passa da 0 a 1 la rete memorizza ciò che è presente all'ingresso.



In seguito al fronte di salita del CK l'uscita Q assume ciò che è presente sull'ingresso D con un certo tempo di risposta. Cosa accade all'uscita se D cambia mentre CK è a 1 o a 0? Nulla!

Normalmente in un flip-flop sono presenti anche i segnali A\_RES e A\_SET, che agiscono in modo indipendente dal CK (se A\_RES=1 allora Q=0 indipendentemente dal fronte del CK e da D) → segnali prioritari.

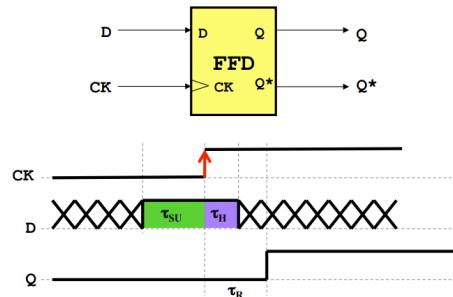
I FFD sono dotati di due ulteriori ingressi "asincroni" che consentono di settare (A\_SET) o resettare (A\_RES) Q indipendentemente da CK e D.



Questi due segnali di set e reset sono importanti per l'inizializzazione delle reti → all'avvio vengono generati dei segnali che resettano tutte le reti sequenziali sincrone che altrimenti partirebbero da uno stato casuale.

Per usare il flip-flop vanno rispettati certi vincoli → tempi di setup e tempi di hold:

Tempi di Setup ( $\tau_{SU}$ ), Hold ( $\tau_H$ ) e Risposta ( $\tau_R$ )



Il corretto funzionamento è garantito solo se  $\tau_{SU} \geq \tau_{S\text{min}}$  e  $\tau_H \geq \tau_{H\text{min}}$ .  
In caso contrario, metastabilità.

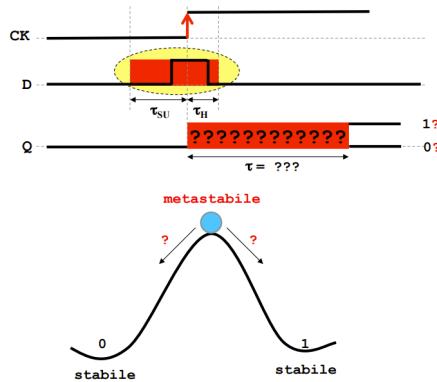
Cosa implicano i parametri  $\tau_{S\text{min}}$  e  $\tau_{H\text{min}}$  indicati nei datasheet ?

Tempo di setup → tempo entro il quale il segnale che vogliamo campionare deve rimanere stabile;

Tempo di hold → tempo per cui deve essere stabile dopo il fronte di salita del clock;

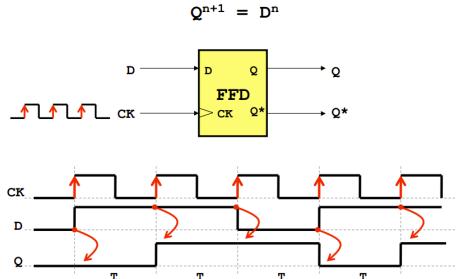
Metastabilità → fenomeno assolutamente da evitare che si verifica quando non vengono rispettati questi tempi minimi → immaginiamo il ff-D come un sistema meccanico e gli diciamo di passare da 0 a 1 ma non gli diamo il tempo di farlo, e lì la rete può fare di tutto → può stare per un certo tempo qui e poi andare in uno stato imprevedibile dopo un tempo imprevedibile → stato metastabile che non è né 0 ne 1.

Il mancato rispetto dei vincoli sul campionamento dei segnali porta a metastabilità.

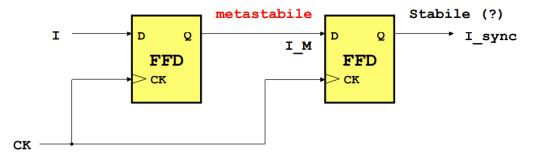


Un segnale sincrono è regolato dal clock, quindi varia al variare del clock.

Se all'ingresso CK viene inviato un segnale periodico (clock):  
il FFD ritarda (D = Delay) il segnale di uscita Q, rispetto al  
segnale di ingresso D, di un tempo pari al periodo di clock T



Normalmente i segnali provenienti dall'esterno (ma non solo) non sono sincroni con il clock della RSS. Questo è un problema molto comune.  
Come gestire potenziali situazioni di metastabilità che potrebbero compromettere il corretto funzionamento della RSS?



- La soluzione mostrata garantisce che l'uscita  $I_{sync}$  assume il valore di  $I$  nel momento in cui tale segnale è stato campionato?
- Sono sufficienti due livelli di FF?
- Quali sono gli effetti collaterali di questa soluzione?

Campionando due volte è probabile che la metastabilità si sia esaurita, partiamo da un segnale non sincrono e generiamo un segnale sincrono. Aumentando il numero di livelli è più probabile risolvere la metastabilità.

L'effetto collaterale è che in questo caso il segnale subirà due delay dei due clock, ma non è un problema.

Alcune considerazioni sulle RSS:

- Lo stato della rete cambia solo in corrispondenza dei fronti di salite del clock che si susseguono con periodo  $T$ ;
- La rete risponde ogni  $T \rightarrow$  se si desidera massimizzare la velocità di risposta è necessario adottare il modello di Mealy;
- La rete è svincolata dai ritardi della rete  $G \rightarrow$  quindi nessun problema di corse critiche (purchè  $T > \tau_{SUmin} + \tau_{Rmin}$ )
- All'interno di uno stesso progetto sono tipicamente presenti + RSS e non necessariamente per tutte il clock è lo stesso e/o coincide con il clock del processore.

**CLOCK GATING** → andare a mettere dei gate sul clock; in generale non è una buona strategia perché potrebbero esserci più fronti nel clock e il clock non arriva alle reti allo stesso modo, il clock non viene ricevuto dalla rete allo stesso istante e quindi la rete asincrona dà problemi → glitch sul clock.

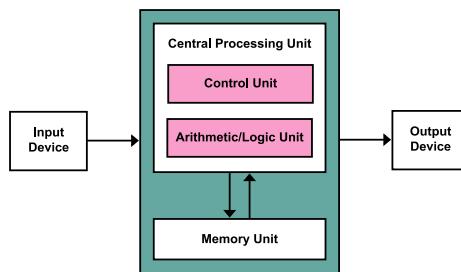


EVITARE SEMPRE CLOCK GATING!!!

## 1. Caratteristiche dei calcolatori

### 1.1 Modello di riferimento

L'hardware di un calcolatore si interfaccia con il software attraverso il suo set di istruzioni (linguaggio macchina);



Ogni blocco della struttura è costituito da circuiti elettrici digitali, e al suo interno tutte le informazioni sono codificate in forma binaria → nell'unità di elaborazione vengono elaborate variabili binarie, e in memoria dati ed istruzioni sono sotto forma di variabili binarie.

Bus → supporto tra i blocchi che costituiscono il calcolatore;

In memoria risiedono il programma (istruzioni codificate in forma binaria) e gli operandi delle istruzioni (dati elaborati e da elaborare) → le istruzioni vengono eseguite in sequenza dalla CPU (*macchina sequenziale sincrona*); a livello di massima astrazione il suo automa ha solo due stati:

- *Instruction Fetch* → la CPU legge in memoria la prossima istruzione da eseguire;
- *Execute* → la CPU esegue l'istruzione letta da IF;

### 1.2 Componenti principali

#### Memoria centrale

Durante l'esecuzione del programma, la CPU legge le istruzioni e scambia dati e altre informazioni con la memoria attraverso il bus. Tale memoria è divisa come un array  $M[0, 1, \dots, 2^{n-1}]$  di  $2^n$  elementi detti celle di memoria; l'array in sé spazio di archiviazione di memoria.

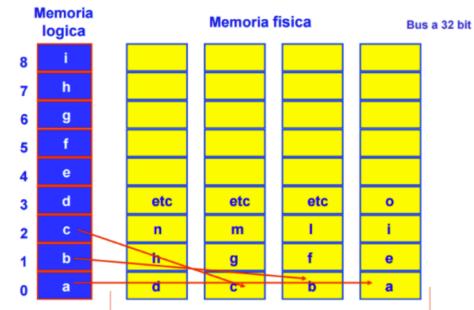
L'indice  $i$  che individua la cella  $M[i]$  si chiama indirizzo della cella → l'accesso da parte della CPU all'informazione residente in  $M[i]$  avviene sempre mediante il rispettivo indirizzo; pertanto sul bus di sistema non solo devono transitare i contenuti, ma anche gli indirizzi delle celle di memoria (contemporaneamente o in successione).

L'indirizzo di una cella in uno spazio di indirizzamento di  $2^n$  celle è una configurazione binaria di  $n$  bit; la dimensione di tale spazio in una CPU è uno dei parametri che ne caratterizza l'architettura.

Ogni cella è solitamente composta da 8 bit (1 byte) → in questo caso si dice che la memoria è organizzata in byte. Gli indirizzi di memoria si indicano in *codice esadecimale* e la dimensione di uno spazio di indirizzamento si può esprimere in KB ( $2^{10}$  byte), MB ( $2^{20}$  byte) e GB ( $2^{30}$  byte).

La memoria logica è un array che parte da 0 e finisce alla fine dello spazio di indirizzamento - 1 e gli elementi contenuti in essa sono gli elementi memorizzati in memoria. Come in questo caso (→) si può avere un parallelismo ovvero è possibile leggere e scrivere da 4 memorie contemporaneamente (+ vantaggioso!).

Se x esempio si vogliono scrivere 16 byte, senza parallelismo ci vogliono 16 cicli, mentre scrivendo 4 byte alla volta bastano 4 cicli.



Per poter fare ciò contemporaneamente si deve poter leggere e scrivere in quattro memorie fisicamente distinte → ogni memoria accetta un solo byte! → *indirizzi consecutivi e contigui nella memoria logica finiranno in memorie fisiche diverse.*

## Program counter

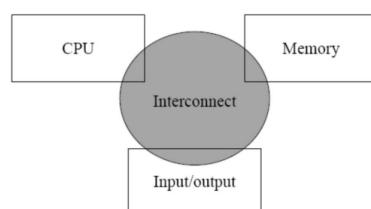
Per poter eseguire le istruzioni in sequenza, la CPU dispone al suo interno di un contatore detto *Program Counter* (PC) che viene incrementato ad ogni **FETCH**. Il PC contiene l'indirizzo di memoria della prossima istruzione da leggere sulla successiva fase di **FETCH**.

## Interfacce I/O

Così come i dispositivi di memoria, anche le interfacce di I/O sono mappate in uno spazio di indirizzamento, distinto da quello della memoria oppure nello stesso (*memory mapped I/O*).

Lo spazio di indirizzamento in I/O è solitamente < dello spazio di indirizzamento in memoria (64 KB). Anche se i due spazi di indirizzamento sono distinti, i segnali del bus che portano l'indirizzo sono comuni → la distinzione tra i due spazi di indirizzamento viene affidata ad appositi segnali del bus.

## Bus di sistema

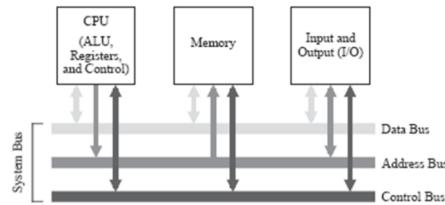


I segnali del bus di sistema sono suddivisi in tre gruppi:

- Bus dati → costituito dai segnali che portano istruzioni e operandi; il suo parallelismo è multiplo di un byte secondo una potenza di 2. Il parallelismo del bus di dati è un altro parametro caratteristico dell'architettura della CPU. Il bus di dati è identificato dal vettore di  $m$  bit  $M[m-1, \dots, 0]$ .
- Bus degli indirizzi → costituito dai segnali che identificano la posizione delle informazioni trasferite nello spazio di indirizzamento a cui si intende accedere; identificato dal vettore di  $n$  bit  $M[n-1, \dots, 0]$ .

$1, \dots, 0]$ .

- Bus dei segnali di comando → composto dai segnali che comandano i trasferimenti di dati sul bus.

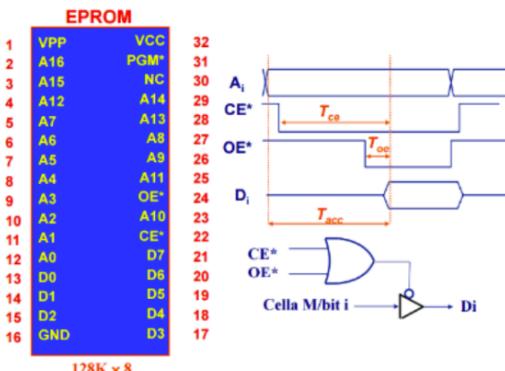


## 2. Dispositivi notevoli

### 2.1 Memoria EPROM

Una EPROM è una memoria a sola lettura (non volatile) che può essere programmata più volte in modo elettrico.

L'esempio mostra una memoria (non utilizzata) da 128K per 8 bit → ovvero 128KB organizzati per 8 ed è tipico per le memorie avere 8 bit. Una memoria di questo tipo dovrà avere  $\log_2 128K = 17$  indirizzi.



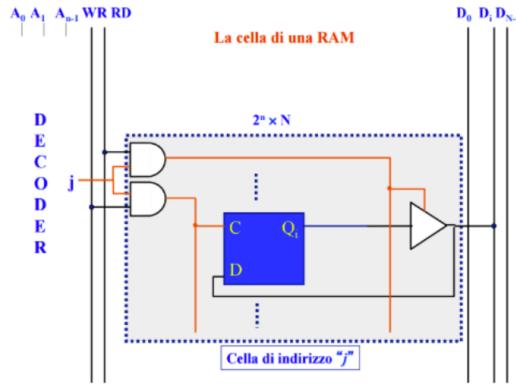
### 2.2 Memoria RAM

La memoria ad accesso casuale è un tipo di memoria volatile caratterizzata dal permettere l'accesso diretto a qualunque indirizzo di memoria con gli stessi tempi.

La RAM si contrappone alla memoria ad accesso sequenziale, in cui i dati sono disposti in modo sequenziale e per accedervi è necessario scorrere su tutti i dati precedenti.

Una possibile cella di memoria RAM è espressa come in figura. Dal processore, si hanno due segnali: *Write* (WR) e *Read* (RD); quando sono settati a 1 il processore vuole eseguire tali azioni in memoria. Non è possibile per configurazione che siano entrambi settati a 1 contemporaneamente, mentre se sono entrambi a 0 il processore nè legge nè scrive.

Alla destra della figura sono mostrati i bus di dati, e il dispositivo ha un decoder interno che ha  $N$  bit in ingresso e  $2^N$  connessioni che abiliteranno in modo esclusivo una determinata cella per decodificare qual è il bit o il byte che deve essere letto o scritto.



## 2.3 Integrati notevoli

- **244**

Integrato del Driver 3-State; presenta 8 buffer 3-state (8 ingressi e 8 uscite) con all'interno due segnali Enable (EN1 e EN2) che abilitano 4 buffer ciascuno.

- **245**

Integrato con buffer bidirezionale, in sostanza la composizione di due integrati 244 che si può utilizzare ogni volta che si avrà una periferica che si connette a un bus e tale periferica legge e scrive in entrambe le direzioni. Presente un segnale in ingresso in più chiamato DIR che specifica la direzione del trasferimento.

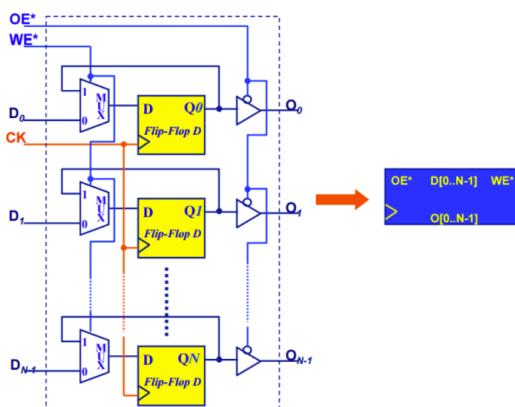
- **373**

Batteria di 8 latch CD; presenta un segnale CK che denota un comando (da non confondere con il segnale CK del clock). Il dispositivo ha anche la possibilità di andare in output enable con un 3-state sulle 8 uscite (se OE=1 le uscite saranno in 3-state).

- **374**

Batteria di 8 FFD; un solo segnale di clock è inviato a tutti i FFD → 8 uscite comandate dal segnale OE (se settato a 1 le uscite sono in 3-state).

## 2.4 Registro Edge-Triggered con WE\*



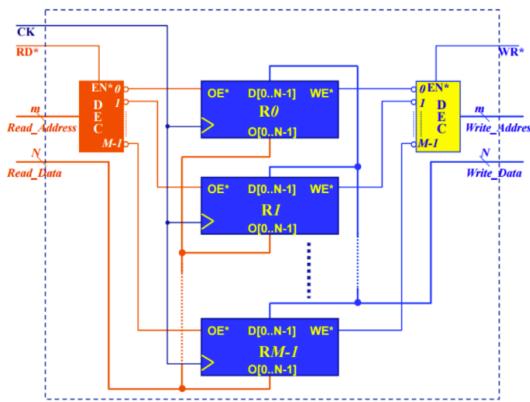
Un registro *edge-triggered* campiona informazioni in ingresso solo ai fronti di un segnale di clock CK (assumiamo durante i fronti di salita). Abbiamo bisogno di un registro a N bit in grado di:

- acquisire N dati sugli ingressi D[0...N-1] sul fronte di salita del segnale di clock CK;
- abilitare le uscite degli N flip-flop D (ossia mettere il dispositivo in 3-state o meno) → utilizziamo un registro edge-triggered con segnale di WE (*write enable*)

Nel nostro esempio, se WE=0 allora voglio scrivere ciò che è presente sugli ingressi al prossimo fronte di salita del segnale di clock CK; se WE=1 non voglio scrivere ciò che è presente sugli ingressi e negli N FFD verrà mantenuto il precedente valore. Poniamo WE\* come bit di indirizzo di un multiplexer (MUX) → anche WE\* è un ingresso. Il segnale D deve essere stabile per un periodo prima del fronte di salita del clock e anche dopo il fronte di salita del clock (rispetto dei vincoli temporali).

In un registro, oltre a poter decidere QUANDO poter scegliere (con WE\*) è utile poter mettere le uscite in 3-state (tramite l'OE, quando OE\*=0 le uscite  $Q_i$  sono direttamente elettricamente connesse agli  $O_i$ , quando OE=1 vanno in 3-state).

## 2.5 Register File (1 read port, 1 write port)



Consiste in una batteria di registri con WE\* → contiene al suo interno N registri, nei quali possiamo decidere anche da quale scrivere o leggere tramite le due porte (read-port e write-port) → spesso le due cose possono essere fatte contemporaneamente;

Si ha come ingresso un CK, un segnale Read, M bit di indirizzo (ovvero  $2^M$  registri), si ha un segnale WR, WR\_Address e WR\_Data. Ogni registro è dotato di un segnale WE.

Se è selezionato il dispositivo in basso e il suo WE\*=0 allora tale registro deve immagazzinare i dati che provengono dal bus dati (Write\_Data); viene selezionato questo dispositivo e non gli altri per mezzo del decoder che prende l'indirizzo in ingresso e capisce che il dispositivo è quello che deve essere abilitato alla scrittura. Se WE\*=1 non viene campionato nulla.

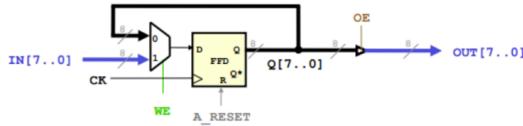
La lettura si può fare in qualsiasi momento, la scrittura invece è legata al fronte di salita del clock; si può leggere e scrivere in contemporanea ma da due diversi registri; mentre si è in lettura si può predisporre la scrittura.

## 2.6 Esempi utili

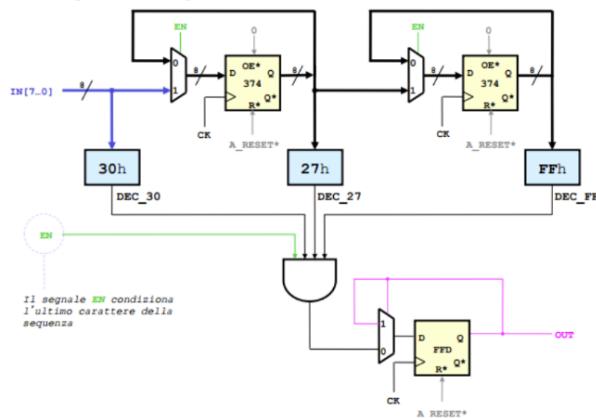
Progettare un registro a 8 bit con uscita 3-State utilizzando FFD positive edge triggered.

La rete, ad ogni fronte di salita del clock, memorizza il byte  $IN[7..0]$  in ingresso se  $WE = 1$  mentre mantiene il valore precedentemente memorizzato in caso contrario ( $WE = 0$ ).

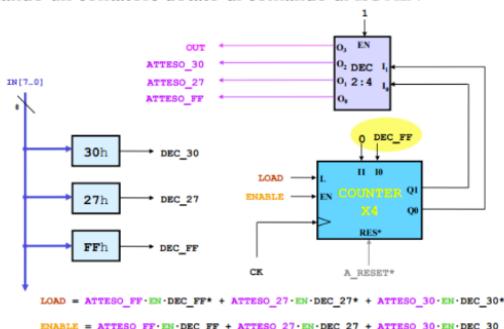
L'uscita  $OUT[7..0]$  della rete deve essere posta nello stato di alta impedenza quando il segnale  $OE = 0$ . Inoltre, la rete deve essere dotata di un ingresso asincrono di RESET ( $A\_RESET$ ) che, se 1, pone al livello logico 0 l'uscita  $OUT[7..0]$  indipendentemente dal valore dei segnali WE, IN e CK.



Progettare un rete che controlla se gli ultimi tre caratteri che si sono presentati sull'ingresso  $IN[7..0]$  mentre il segnale EN era a livello logico 1 sono stati  $FFh$  (primo carattere della sequenza),  $27h$  e  $30h$ . Nel caso sia rilevata la sequenza  $FF - 27 - 30$ , nel periodo di clock successivo a quello dell'ultimo carattere ricevuto ( $30h$ ), deve essere asserita l'uscita OUT e rimanere tale fino a che non viene asserito il segnale (asincrono) di reset  $A\_RESET$ . In seguito ad un reset deve riprendere immediatamente il controllo della sequenza in ingresso come se non fosse stato ricevuto alcun carattere.



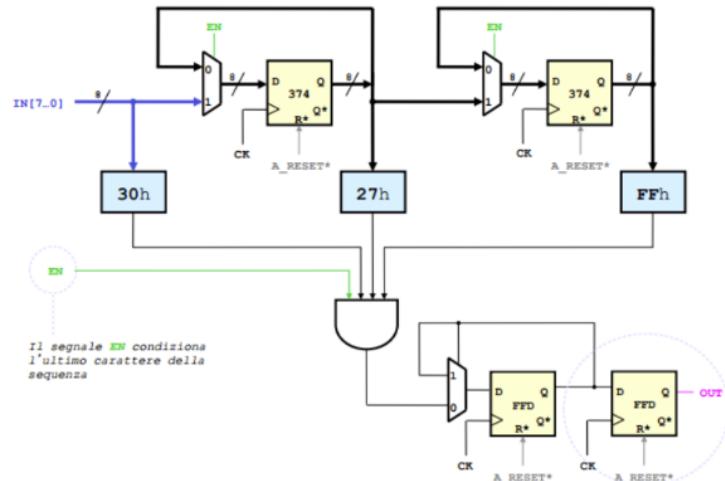
Una soluzione alternativa utilizzando un contatore dotato di comando di LOAD.



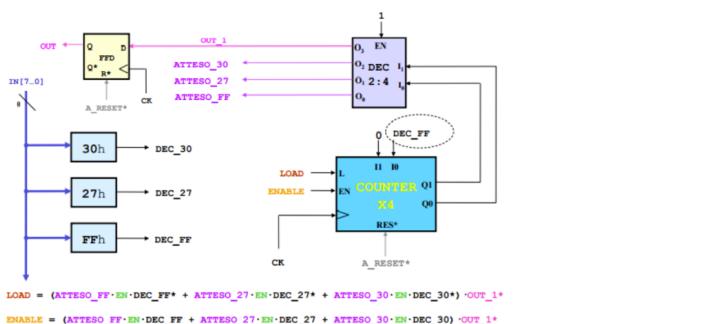
Progettare un rete che controlla se gli ultimi tre caratteri che si sono presentati in ingresso  $IN[7..0]$  mentre il segnale  $EN = 1$  sono stati  $FFh$  (primo carattere della sequenza),  $27h$  e  $30h$ . Nel caso sia rilevata tale sequenza, due periodi di clock successivi a quello dell'ultimo carattere della sequenza ricevuto deve essere asserita l'uscita  $OUT$  e rimanere tale fino a che il segnale di reset (asincrono)  $A\_RESET$  non assume il valore logico 1.

In seguito ad un reset (asincrono) la rete deve riprendere immediatamente il controllo della sequenza in ingresso come se non fosse stato ricevuto alcun carattere.

### Soluzione 1



### Soluzione 2



## 3. Mapping e decodifica

Per mapping di un dispositivo si intende l'operazione di collocazione della memoria all'interno del calcolatore. Per decodifica, invece, si intende l'operazione che consente di avvisare il dispositivo in particolare che risponde all'indirizzo associato. Tutte le comunicazioni che avvengono all'interno di un sistema a microprocessore avvengono attraverso i bus.

Il trasferimento di un'informazione tra agenti del bus avviene tramite una sequenza di eventi detta ciclo di bus.

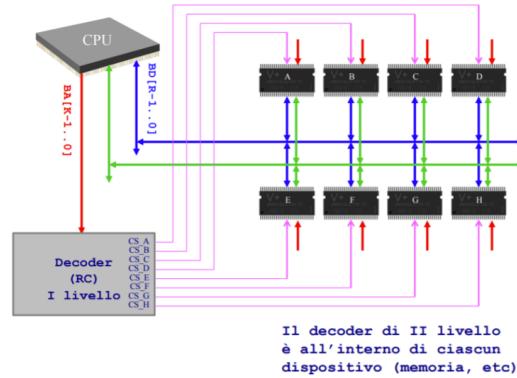
Ad esempio, per leggere in memoria all'indirizzo  $i$  la CPU (master del bus) genera un ciclo di bus costituito dalla seguente sequenza di bus:

1. mette sul bus di indirizzi l'indirizzo → 2. attiva un segnale di comando che identifica l'operazione desiderata → 3. attende che la memoria indirizzata (*slave*) metta l'informazione desiderata sul bus dati → 4. legge (cioè campiona) sul bus dati i segnali generati dalla memoria.

La condizione necessaria affinché un dispositivo fisico (memoria, interfaccia, o altre entità), sia accessibile al software è che il dispositivo deve essere mappato in uno spazio di indirizzamento, ovvero deve

essere associata al dispositivo una finestra di indirizzi di quello spazio di indirizzamento → condizione di visibilità.

Una CPU che emette un indirizzo a 20 bit ha uno spazio di indirizzamento di 1 MB ( $2^{20}$ ), 32 bit → 4 GB ( $2^{32}$ ).



### 3.1 Mapping di dispositivi da 8 bit in sistemi con bus di dati da 8 bit

Considerando solo dispositivi da 8 bit (con porta dati di 8 bit) si considera il parallelismo di tutti i dispositivi visti fin ora (memoria, latch, registri edge-trigger, driver, ...) ed imponiamo temporaneamente l'ulteriore condizione che il parallelismo del bus dati sia di 8 bit. In questa ipotesi, l'assegnamento a un dispositivo di una finestra di indirizzo in uno spazio di indirizzamento avverrà in generale, ma non necessariamente, nel rispetto delle due seguenti ulteriori condizioni restrittive:

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

Un dispositivo accessibile attraverso il bus, sia esso un dispositivo di memoria o un'interfaccia, occupa in generale  $n = 2^k$  posizioni nello spazio di indirizzamento →  $n$  rappresenta il numero di oggetti di 8 bit indirizzabili all'interno del dispositivo (es. numero di celle nella RAM) mentre  $k$  rappresenta il numero di bit di indirizzo interni al dispositivo (*fortemente variabile al variare del dispositivo* → generalmente estremamente piccolo per le interfacce e molto grande per le memorie). Esempio → per una RAM da 128KB si hanno 128k indirizzi, ossia  $2^{17}$  (17 sono i fili/bit di indirizzo).

Nota → i  $k$  bit sono utilizzati durante la decodifica di secondo livello, che è sempre più veloce possibile per identificare ciò che la CPU richiede durante quel ciclo di bus.

### 3.2 Caratteristiche di un dispositivo indirizzabile su finestra di $n = 2^k$ byte

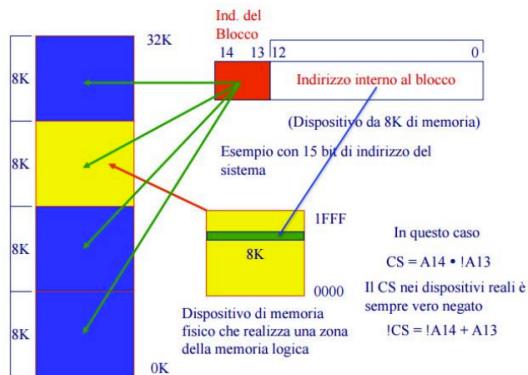
Qualunque dispositivo da 8 bit con all'interno  $n = 2^k$  elementi indirizzabili separatamente ha al suo interno un decoder di secondo livello di  $k$  variabili con:

- Ingresso di ENABLE che seleziona i singoli oggetti indirizzabili;
- Segnale di CHIP\_SELECT (detto anche chip\_enable) che è collegato alla rete combinatoria di decodifica di primo livello → serve a decodificare il dispositivo;
- $k$  linee di indirizzo di dimensione compatibile al numero di indirizzi che occupa quel dispositivo (vicino a  $2^k$ ) (indicati con  $A[k-1..0]$ )



- Bit RD e WR, il RD detto anche output-enable è il comando di lettura e se RD e CS sono attivi il dispositivo espone sul bus dati D[7..0] il contenuto della cella indirizzata; se CS e WR sono attivi viene campionato il dato presente su D[7..0]

Nota → i segnali di lettura e scrittura sono comandati dalla CPU, che emette indirizzi che vengono poi decodificati dal CHIP SELECT (potrebbe non emettere indirizzi, ma qualcosa sulle linee di indirizzo c'è sempre al massimo non succede nulla)



La decodifica di primo livello avviene, come ogni operazione, durante un ciclo di bus del processore. In questo caso vogliamo mappare un dispositivo da 8K di memoria → è buona norma mapparlo a indirizzi di taglia multipla di quelli del dispositivo → "scegliamo" uno spazio di memoria (indirizzamento) da 32K (8K x 4).

Con 32K →  $2^{15}$  → 15 bit di indirizzo del sistema → BA[14..0]

Ci sono dunque 4 slot possibili in cui mappare il dispositivo (32/8) → ogni slot ha uno spazio di 8K → i 4 slot ( $2^2$ ) sono rappresentabili con 2 bit → usiamo BA[14] e BA[13] → il **CHIP SELECT** sarà una rete combinatoria che utilizza questi due bit.

La decodifica di primo livello serve a identificare qual è il dispositivo e dov'è. In questo caso potrebbe trovarsi nelle posizioni identificate da:

- 00 ---- primo slot in basso ----- BA14 = 0, BA13 = 0 ----- BA14\*
- 01 ---- secondo slot in basso ----- BA14 = 0, BA13 = 1 ----- BA14\*
- 10 ---- terzo slot dal basso ----- BA14 = 1, BA13 = 0 ----- BA14
- 11 ---- quarto slot dal basso ----- BA14 = 1, BA13 = 1 ----- BA14

Il **CHIP SELECT** serve a identificare il blocco tra i 4 possibili → il nostro dispositivo è mappato nel terzo slot, ossia BA14 e BA13\* → **CS = BA14 AND BA13\*** → questi due bit vengono detti **componenti alpha**.

Una volta identificato il dispositivo, da dove vogliamo leggere o scrivere nello specifico dal dispositivo? da che indirizzo? → decodifica di secondo livello; → dobbiamo localizzare uno di quegli 8K, quanti bit sono necessari? 13 bit (per individuare una cella all'interno del dispositivo).

**?** perché  $2^{13} = 8192$  che è vicino a 8K (8000)

Avevamo "disponibili" 15 bit di cui ne usiamo 2 per la decodifica di 1 livello → BA[14,13] mentre i rimanenti 13 bit li utilizziamo per la decodifica di 2 livello → BA[12..0].

Le decodifiche in realtà non sono sequenziali, il processore emette i 15 bit contemporaneamente e ognuno svolge il suo ruolo.

### 3.3 Mapping allineato

Si consideri un dispositivo  $D$  di  $n = 2^k$  byte indirizzabili, si dice che  $D$  è mappato all'indirizzo  $A$  se gli indirizzi del byte di  $D$  sono compresi tra  $A$  e  $A + (n - 1)$  → cioè se  $A$  è l'indirizzo più basso tra tutti gli indirizzi associati a  $D$ .

Si dice che  $D$  è allineato se  $A$  è un multiplo di  $n$  (numero di bytes interni al dispositivo), cioè si avrà la condizione di allineamento se:

$$< \text{indirizzo più basso di } D > \bmod n = 0$$

Se  $D$  è allineato allora i  $k$  bit meno significativi di  $A$  sono uguali a 0.

Ad esempio:

- un dispositivo da 2 byte è allineato se è mappato ad un indirizzo pari;
- un dispositivo da 8 byte è allineato se è mappato a un indirizzo il cui valore in binario termina con 3 zeri;
- un dispositivo da 16 byte è allineato se il suo indirizzo iniziale in codice esadecimale ha la cifra meno significativa = 0;

Supponiamo di mappare un dispositivo  $D$  di  $2^k$  byte ( $k = 4$ ) a un indirizzo  $A$  allineato di uno spazio di indirizzamento di 1MB (bus di indirizzi di 20 bit).

Si introduce un operatore simbolico  $\#\#$  per concatenare, ovvero dire  $\alpha\#\#i$  significa concatena  $\alpha$  con  $i$ . Allora si può porre  $A = \alpha\#\#(0)^k$  dove  $\alpha$  è una configurazione binaria di 20K bit e gli indirizzi associati a  $D$  saranno compresi tra:

$$A_{min} = A = \alpha\#\#(0)^k$$

$$A_{max} = A_{min} + (2^k - 1) = \alpha\#\#(1)^k$$

Ad esempio:  $A_{min} = F8570$ ,  $A_{max} = F857F$

Dunque è possibile identificare l'indirizzo  $A_i$  dell'i-esimo byte di  $D$  come l'insieme di due campi concatenati  $A_i = \alpha\#\#i$  (ad esempio  $A_i = F8573$ ); dove  $\alpha$  individua tra le  $2^{20-k}$  finestre allineate di  $2^k$  byte presenti nello spazio di indirizzamento quella su cui è mappato (F857);  $i$  individua l'offset nel chip del byte indirizzato ( $i = 3$ ).

#### Esempio 1

Indirizzamento di un byte di una RAM all'indirizzo 40010H in uno spazio di indirizzamento di 1 MB nell'ipotesi di disporre di un chip da 128 KB mappato all'indirizzo 40000H. L'indirizzo viene suddiviso in due campi: il primo identifica la finestra di 128 KB in cui è mappata la RAM, il secondo identifica l'offset nella RAM.



#### Esempio 2

Indirizzamento di un byte all'indirizzo 1026H in un dispositivo di I/O di 16 byte mappato all'indirizzo 1020H di uno spazio di indirizzamento di 64 KB. L'indirizzo viene suddiviso in due campi: il primo identifica la finestra di 16 B in cui è mappato il dispositivo, il secondo identifica l'offset nel dispositivo.



### 3.4 Decodifica completa

Consideriamo uno spazio di indirizzamento di 1MB in cui sia mappato un dispositivo di  $2^k$  byte → per individuare una cella di indirizzo  $A_i = \alpha\#\#i$  possiamo decodificare tutti i 20 bit che compongono  $A_i$ .

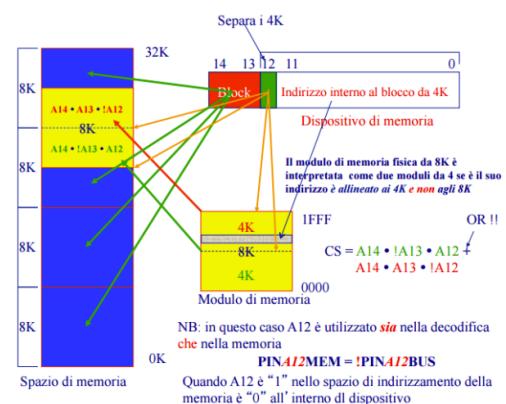
Questa decodifica è effettuata ricorrendo alla struttura dei decoder ad albero, con albero di due livelli:

- **Primo livello** → usato per decodificare  $\alpha$  (pos. in cui il chip è mappato) e servono 20-K variabili;
  - **Secondo livello** → usato per decodificare  $i$  (byte all'interno del chip, decoder di k variabili);

La decodifica si può dire completa se si utilizzano tutti i 20-k bit, semplificata se si utilizza solo un sottoinsieme minimo (minimo termine) di questi ultimi.

### 3.5 Decodifica non allineata

Che cosa succede se si va non più a piazzare il proprio blocco di memoria da 8K come da esempio, all'interno di uno spazio di indirizzamento di 32K? Cosa potrebbe succedere se lo si piazza in uno spazio non allineato, ovvero multiplo di 8K? Succede che il dispositivo lo si può piazzare a metà dell'indirizzo allineato (a cavallo dell'indirizzo), ovvero che deve essere mappato per metà sopra quell'indirizzo e per metà sotto. Nel caso in figura si ha un blocco da 8K, lo si mappa a cavallo dell'indirizzo tratteggiato, che è allineato, di conseguenza si avrà 4K sopra e 4K sotto.



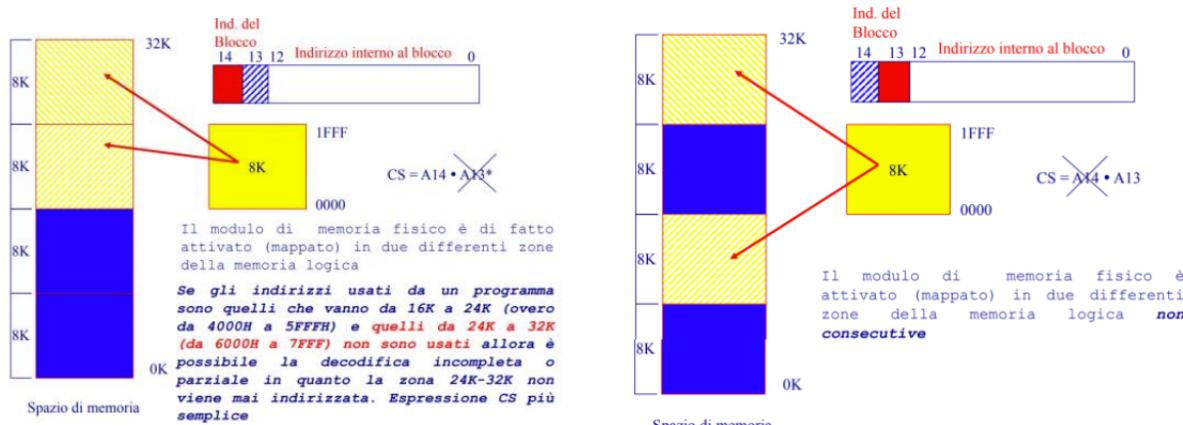
Se si hanno 15 bit di indirizzo la decodifica completa assume la seguente forma:

$$CS = A14 * !A13 * A12 + A14 * A13 * !A12$$

Significa che si deve abilitare il dispositivo giallo, solo quando si è nei secondi 4K di tale slot da 8K oppure solo nei primi 4K dello slot sopra da 8K.

### **3.6 Decodifica parziale**

Si supponga di essere in un sistema con bus 8 bit; utilizzando mapping di blocchi allineati → decodifica completa → si utilizzano tutti i bit che rimangono. Se si usano meno bit nella decodifica i blocchi si replicano con un certo pattern a seconda del bit che si va ad eliminare (più o meno significativo) → di conseguenza aumenta lo spazio che si assegna ad ogni dispositivo (però si utilizzano meno bit!).



## Esempio:

Si consideri un sistema con bus di indirizzi a 16bit e bus dati a 8bit. Scrivere le espressioni di decodifica completa e semplificata (quella da usare all'esame) nei seguenti casi:

1. Dispositivo di memoria da 16 KB mappato a 8000h;
2. Dispositivo di memoria da 8 KB mappato a 0000h;
3. Entrambi i dispositivi precedenti.

Risoluzione:

1.  $16 \text{ KB} = 2^{14} = 16.384 \rightarrow 14 \text{ bit per entrare nella memoria};$

Spazio di indirizzamento da 64 KB  $\rightarrow$  4 slot da 16KB  $\rightarrow$  2 bit ( $4 \rightarrow 2^2$ ) necessari per identificare lo slot in cui c'è il dispositivo ( $16 - 14 = 2$  bit per la decodifica di primo livello)  $\rightarrow$  per la decodifica completa servono 2 bit;

$8000 \text{ h} = 8000 \text{ esadecimale} = 1000\ 0000\ 0000\ 0000$

$8000 \text{ h}$  è il punto in cui è mappato  $\rightarrow$  il 3° slot ha indirizzo 8000h; il dispositivo sta in 10 (binario)  $\rightarrow$  espressione combinatoria del chip select:

$$CS = BA15 \text{ AND } BA14^*$$

Poichè il dispositivo è unico, posso posizionarlo in tutte e 4 le slot del mio spazio di indirizzamento  $\rightarrow$  l'espressione semplificata della decodifica del chip select è **CS = 1** (il disp. è ovunque  $\rightarrow$  la memoria risponde a 4 indirizzi [00,01,10,11]  $\rightarrow$  rete combinatoria più semplice  $\rightarrow$  risparmio).

2.  $8 \text{ KB} = 2^{13} = 8.152 \rightarrow 13 \text{ bit per entrare nella memoria};$

spazio di indirizzamento 64 KB  $\rightarrow$  8 slot da 8KB  $\rightarrow$  3 bit ( $2^3$ ) per identificare lo slot con D;  $16 - 13 = 2$  bit per la decodifica di primo livello  $\rightarrow$  per la decodifica completa servono 3 bit;

$0000h \rightarrow 0000\ 0000\ 0000\ 0000 \rightarrow$  1 slot ha indirizzo 0000h  $\rightarrow$  il dispositivo sta in 000 (binario)  $\rightarrow CS = BA15^* \text{ AND } BA14^* \text{ AND } BA13^*$   $\rightarrow$  espressione semplificata (dispositivo unico)  $CS = 1$ ;

3. Decodifica completa e semplificata sono uguali alle precedenti; per la decodifica semplificata possiamo dividere i 64 KB di memoria in 2 slot ( $2 = 2^1 \rightarrow 1$  bit necessario) avendo 2 dispositivi e quindi:

- a. assegnare metà spazio al dispositivo da 8KB  $\rightarrow CS = BA15^*$
- b. l'altra metà al dispositivo da 16KB  $\rightarrow CS = BA15$

## 3.7 Mapping, read, write e set/reset di un FFD

Il Flip-Flop D è un elementare dispositivo di memoria, con una CPU come possiamo scrivere e leggere istruzioni o settare/resettare in modo asincrono il FFD?

Consideriamo il caso di una CPU con bus dati a 8 bit con 20 bit di indirizzo, 64 K di RAM agli indirizzi alti e 64 K di EPROM agli indirizzi bassi.

20 linee di indirizzo  $\rightarrow 2^{20} = 1$  milione di indirizzi nello spazio di indirizzamento  $\rightarrow$  agendo sui due comandi asincroni A\_RES e A\_SET dobbiamo scrivere il chip select della RAM, della EPROM, per scrivere, leggere, impostare il FFD al livello logico 1 e al livello logico 0.



La RAM da 64K non esiste → 2 RAM da 32K → RAM LOW (L) e RAM HIGH (H)  
Le EPROM da 64K esistono!

**Se vogliamo leggere** da un FFD, il segnale Q deve essere collegato in qualche modo al bus dati BD[7..0] ma ci sono altri dispositivi già connessi, non è possibile una connessione diretta → dobbiamo inserire una rete con chip select tra Q e uno dei bit del bus dati → usiamo un 3-state con `MEM_RD = 1` perché ciclo di bus di lettura;

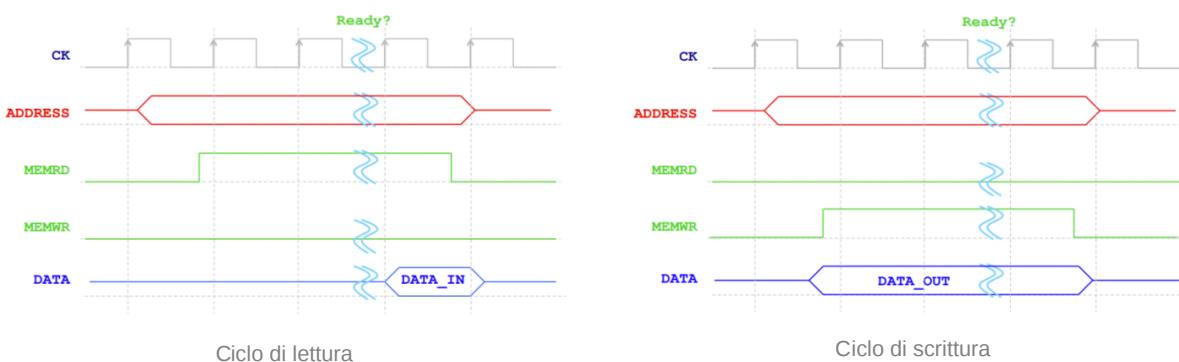
**Se vogliamo scrivere** su FFD dobbiamo prendere un segnale/bit dal bus dati BD[7..0] e collegarlo a D → sui fili del bus dati deve esserci solo il dispositivo che viene letto (tutte le altre uscite sono in 3-state) → ma dobbiamo porre un fronte di salita del clock quando c'è il dato in D nel ciclo di bus → rispettare tempi di setup e di hold → `MEM_WR = 1` perché ciclo di bus di scrittura (varia da 0 a 1 a seconda del fronte di salita-discesa) → useremo un segnale negato così che quando c'è un fronte di salita si capisca che si vuole scrivere.

Assumiamo che i comandi del FFD siano mappati nei seguenti indirizzi:

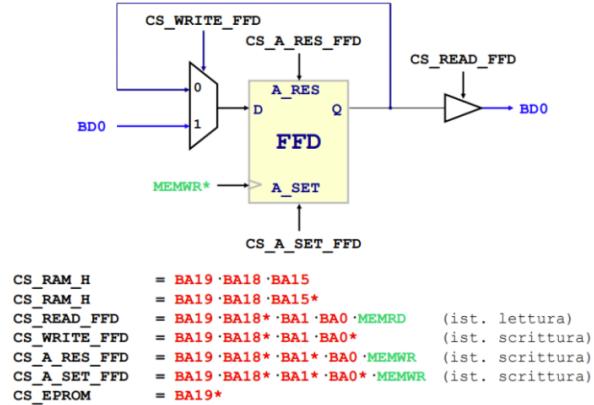
CS_READ_FFD	-> 80003h	-> 11
CS_WRITE_FFD	-> 80002h	-> 10
CS_A_RES_FFD	-> 80001h	-> 01
CS_A_SET_FFD	-> 80000h	-> 00

e assumiamo di utilizzare il segnale `BD0` del bus dati per leggere e scrivere il singolo bit di dato. Se un indirizzo è già usato da RAM o EPROM non dobbiamo utilizzarlo per altri CS → un determinato indirizzo indicato da un chip select dovrebbe rispondere a un solo determinato dispositivo.

Scegliamo indirizzi contigui per semplificare il CS (non necessario ma così si risparmia sulla rete) → ad esempio se voglio leggere dal FFD devo fare ciclo di bus di lettura all'indirizzo 80003h.



Soluzione: rete sequenziale sincrona con utilizzo di reti combinatorie (multiplexer) e 3-state:



(seconda riga in realtà è CS\_RAM\_L)

l'uscita Q del FFD collegata a BDO con un 3state attraverso il segnale di `CS_READ_FFD` che va a 1 quando l'indirizzo corrisponde all'indirizzo che abbiamo assegnato a quel dispositivo e (in contemporanea) quando il segnale `MEM_RD = 1`.

#### Mapping delle memorie:

- EPROM → possiamo assegnarle tutti i primi 512K →  $CS\_EPROM = BA19^*$  → la prima metà dello spazio di indirizzamento la assegniamo completamente alla EPROM;
- RAM → assegniamo la seconda metà alla RAM →  $CS\_RAM = BA19^*BA18$  e poi discriminiamo in base al bit BA15 (negato e non) → ma perchè?  
256K = 4 blocchi da 64 K → la RAM deve stare negli ultimi 64K (32+32) nelle ultime 2 delle 8 mini-slot  
→ usando BA17 e BA17\* avrei assegnato a quegli indirizzi qualcos'altro;

**Mapping dei 4 segnali** → 4 indirizzi contigui → i 4 segnali del chip select nei 256 mega definiscono le istruzioni;

BA1 e BA0 per selezionare tra i 4 indirizzi:

- `CS_READ_FFD` → all'interno dello slot da 256K devo identificare l'indirizzo 3 (80003h → 3) quindi BA1 AND BA0 (11) con segnale `MEM_RD` perchè è un'istruzione di lettura → è bene metterlo perchè se nei transistor ci fosse una scrittura a quell'indirizzo, il segnale potrebbe andare a 1 ma a noi serve che accada soltanto durante un ciclo di bus di lettura a quell'indirizzo;
- `CS_WRITE_FFD` → non vogliamo scrivere sul FFD quando c'è un ciclo di bus di scrittura generico (potrebbe riguardare la RAM o un altro dispositivo) → lo vogliamo solo quando c'è un ciclo di scrittura all'indirizzo a cui mappiamo `CS_WRITE_FFD` (80002h → 2 → BA1 AND BA0\* → 10 = 2) → ciò che è su BDO va sull'ingresso D del FFD → il segnale di riferimento è `MEM_WR` (non il clock) → non campioniamo sul fronte di salita di `MEM_WR` ma su quello di discesa (sul clock mettiamo `MEM_WR*`) se campionassimo sul fronte di salita daremmo poco tempo a ciò che proviene dal bus dati per arrivare all'ingresso D → evitiamo metastabilità;
- **Comandi asincroni** (tendenzialmente vogliamo evitarli, si utilizzano per settare la logica del sistema in uno stato iniziale definito da noi):
  - `CS_A_SET_FFD` mappato a 0 → 80000h ( $BA1^*BA0^*$ ) → condizionato a uno qualsiasi dei due segnali `MEMWR` o `MEMRD` (bisogna che si attivi se c'è anche un ciclo di bus di lettura/scrittura così evita di andare a 1 durante i transistor)
  - `CS_A_RESET_FFD` mappato a 1 → 80001h ( $BA1^*BA0$ );

### 3.7 Mapping, read, write e set/reset di un latch

Mentre il FFD campiona sui fronti di salita del segnale di clock, il latch CD non campiona ma memorizza le informazioni in ingresso solo se C = 1 (e mantiene le informazioni precedenti se C = 0). Considerando la stessa CPU del FFD, la soluzione è identica alla precedente con la differenza che il latch è mappato nello spazio di indirizzamento della CPU.

- scrittura == istruzione di STORE → durante il ciclo di bus gli indirizzi corrisponderanno a quelli a quelli indicati per capire dove si vuole scrivere (memory write = 1, memory read = 0);
- lettura == istruzione di LOAD → memory read = 1, memory write = 0;

### 3.8 Memorie con processori a parallelismo >8

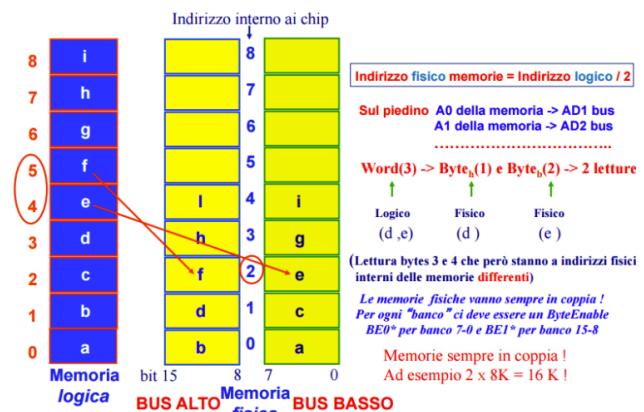
Aumentando il parallelismo dei dati è possibile ridurre la dimensione di ciascuna memoria e trasferire più dati nello stesso ciclo di bus. Nel caso di un trasferimento di 32 bit, avremo quindi 4 memorie che si prendono 8 bit dei 32 → il parallelismo di ciascuna memoria è sempre 8 bit → elementi contigui vanno su memorie diverse → quando devo leggerla o scriverla, il processore riassembra o disassembra le informazioni automaticamente.

E per il trasferimento di un dato non allineato? → meglio di no, altrimenti poi vanno fatti 2 cicli di bus allineati per ricostruirlo e accedere all'intero dato (i processori DLX si fermano per errore di esecuzione).

#### Caso a 16 bit

Abbiamo un banco di memoria da 16 bit, cioè con un bus di memoria a 16bit suddiviso in due bus dati da 8 bit (bus alto, bus basso).

Mappare un dispositivo di memoria in un sistema a 16 bit implica la suddivisione del blocco di memoria logica in due dispositivi fisici grandi la metà del dispositivo di memoria logica in modo tale che essi possano essere mappati rispettivamente sul bus alto e sul bus basso.



Per ognuno dei dispositivi mappati in memoria è richiesto un byte enable, ossia due segnali BE0 e BE1 che vanno ad agire rispettivamente sul bus basso e sul bus alto → la presenza di questi due bus dati permette il trasferimento contemporaneo di 16 bit se effettua l'accesso a indirizzi multipli di 2.

In logica negativa, attivando contemporaneamente BE0 e BE1 si permette il trasferimento di una WORD, se solo uno dei due è attivo si riferisce ad uno dei due.

Per quanto riguarda il mapping dei dispositivi in memoria, con un sistema a 8 bit si è costretti a mappare due dispositivi di dimensione dimezzata (con 128K di spazio di indirizzamento → due dispositivi da 64K l'uno mappati su bus alto e basso).

La decodifica viene effettuata come se si avesse a che fare con sistemi a bus dati a 8 bit → differenza data da BE0 e BE1.

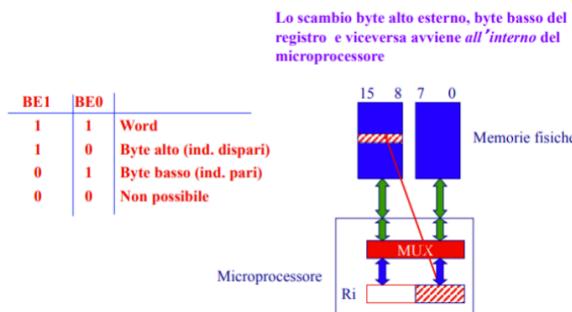


dividere per due = togliere ultimo bit → utilizzo n-1 bit

Il **bus dati basso** della memoria fisica contiene tutti i byte memorizzati in indirizzi pari della memoria logica;

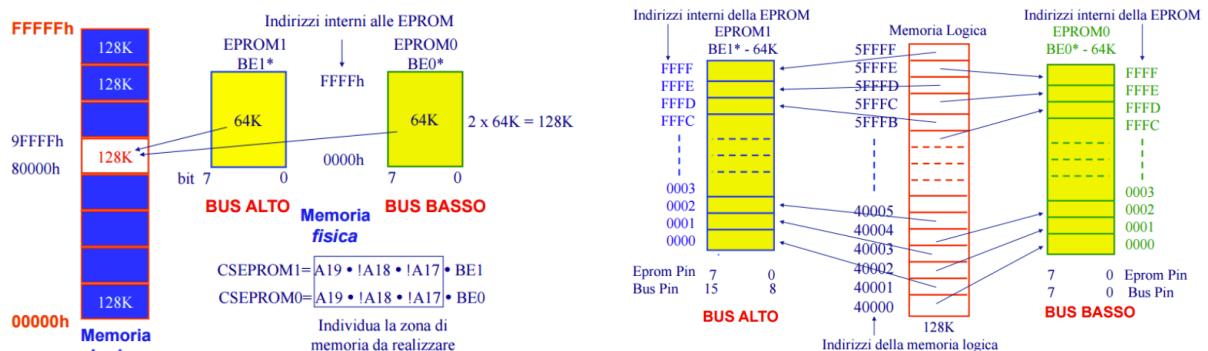
a-b, c-d sono allo stesso indirizzo fisico ma su 2 bus dati diversi → discriminati dal *byte enable* (BE0 = 1 per banco 7..0; BE1 = 1 per banco 15..8) → meglio memorizzare a indirizzi pari per far bastare un ciclo di bus (multipli di 2 considerando anche lo 0 come multiplo).

Il **bus dati alto** della memoria fisica contiene tutti i byte memorizzati in indirizzi dispari della memoria logica;



**BA0** del processore non viene generato (di fatto seleziona il banco - al suo posto BE0 e BE1)  
**BA1** del processore connesso ai piedini A0 delle memorie  
**BA2** del processore connesso ai piedini A1 delle memorie  
etc. etc.

Quando leggo a byte a indirizzo dispari, gli 8 bit più significativi finiranno nei bit meno significativi della CPU → il MUX nel processore può far sì che qualcosa collocato nel bus alto vada a finire nel bus basso.



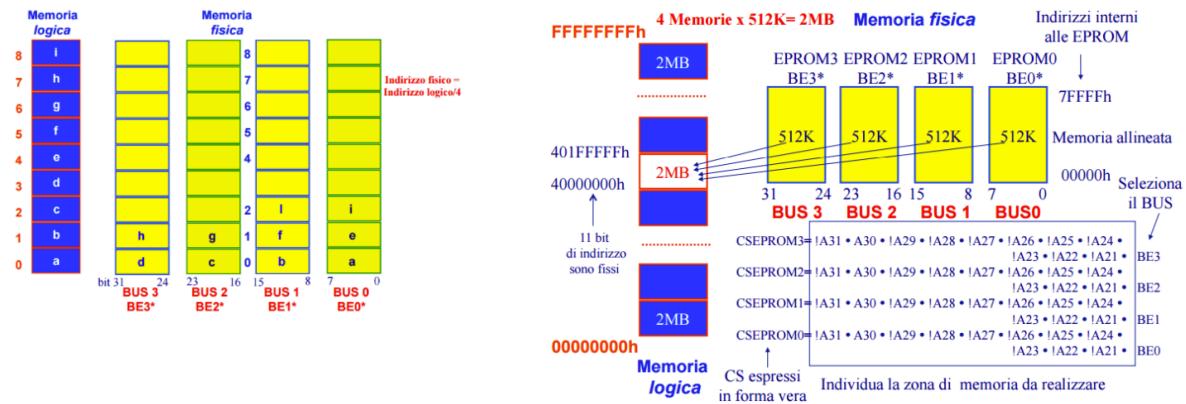
40000 bus dati basso, 40001 bus dati alto → una parte del chip select è la stessa tranne BE1 e BE0 → indirizzi continui, due memorie fisiche diverse

### Caso a 32 bit

Nel caso di un banco di memoria a 32 bit ritroviamo 4 bus di dati da 8 bit, per cui serviranno 4 segnali di *byte enable*. L'indirizzo fisico corrisponde all'indirizzo logico diviso 4 → il dato lo posso trasferire solo se è allineato e deve essere multiplo di 4 → se non è multiplo non hanno lo stesso indirizzo fisico!

In questo caso, attivando contemporaneamente i 4 BE si ha la possibilità di leggere una WORD da 32 bit. Attivando i primi due o gli ultimi due si può leggere una HALF WORD BASSA o una HALF WORD ALTA;

analogamente è possibile leggere i 4 byte uno ad uno.



Caso con 4 eprom e 2MB di memoria

Esempi:

Esempio: si vuole realizzare nel DLX (bus 32 bit) una memoria RAM da 256K posta all' indirizzo 84000000 (allineata). Campo di indirizzamento 84000000-8403FFFF. Dispositivi: 8 RAM da 32 K (le RAM da 64K stanno NON esistono !!!!)  
Di fatto quindi vi sono due banchi da 128K l' uno: il primo realizza la memoria da 84000000 a 8401FFFF e l' altro da 84020000 a 8403FFFF. I chip di memoria da 32 K utilizzano al loro interno (fisicamente) come indirizzi di selezione delle celle i pin A14-A0 che sono però collegati rispettivamente agli indirizzi emessi dal DLX. BA16-BA2 (ricordiamo infatti che BA1 e BA0 del DLX NON sono emessi e al loro posto vengono emessi BE3, BE2, BE1 e BE0). Si noti il ruolo dell' indirizzo DLX BA17 che divide i due banchi

*Primo banco (decodifica non semplificata)*  
CSRAM00=(BA31 • BA30! • BA29! • BA28! • BA27! • BA26 • .... • BA18!•BA17!) •BE0  
CSRAM01=(BA31 • BA30! • BA29! • BA28! • BA27! • BA26 • .... • BA18!•BA17!) •BE1  
CSRAM02=(BA31 • BA30! • BA29! • BA28! • BA27! • BA26•....•BA18!•BA17!) •BE2  
CSRAM03=(BA31 • BA30! • BA29! • BA28! • BA27! • BA26 • .... • BA18!•BA17!) •BE3  
*Secondo banco (decodifica non semplificata)*  
CSRAM10=(BA31 • BA30! • BA29! • BA28! • BA27! • BA26 • .... • BA18!•BA17!) •BE0  
CSRAM11=(BA31 • BA30! • BA29! • BA28! • BA27! • BA26 • .... • BA18!•BA17!) •BE1  
CSRAM12=(BA31 • BA30! • BA29! • BA28! • BA27! • BA26 • .... • BA18!•BA17!) •BE2  
CSRAM13=(BA31 • BA30! • BA29! • BA28! • BA27! • BA26 • .... • BA18!•BA17!) •BE3

Ovviamente nel caso di decodifica semplificata (memoria logica incompletamente realizzata fisicamente) le funzioni di decodifica vengono ridotte di complessità. Ove questi due banchi fossero gli unici da realizzare i CS dipenderebbero solo da BA17 e da BEi

Esempio: si vuole realizzare nel DLX una memoria RAM da 128K posta all' indirizzo 84010000. Campo di indirizzamento 84010000-8402FFFF (allineato a 64K). Dispositivi: 4 RAM da 32 K. Si noti che in questo caso il banco NON si trova a un multiplo di 128K (84010000 è multiplo di 64K e NON di 128 K). Ciascuno dei dispositivi fisici da 32 K deve essere considerato come la somma di due dispositivi da 16 K. I due banchi così risultanti 16K x 4 = 64 K sono entrambi allineati (rispettivamente 84010000-8401FFFF e 84020000-8402FFFF).

CSRAM0=(BA31•BA30!•BA29!•BA28!•BA27!•BA26•...•BA18!•BA17!•BA16 +  
BA31•BA30!•BA29!•BA28!•BA27!•BA26•...•BA18!•BA17!•BA16!) •BE0  
CSRAM1=(BA31•BA30!•BA29!•BA28!•BA27!•BA26•...•BA18!•BA17!•BA16 +  
BA31•BA30!•BA29!•BA28!•BA27!•BA26•...•BA18!•BA17!•BA16!) •BE1  
CSRAM2=(BA31•BA30!•BA29!•BA28!•BA27!•BA26•...•BA18!•BA17!•BA16 +  
BA31•BA30!•BA29!•BA28!•BA27!•BA26•...•BA18!•BA17!•BA16!) •BE2  
CSRAM3=(BA31•BA30!•BA29!•BA28!•BA27!•BA26•...•BA18!•BA17!•BA16 +  
BA31•BA30!•BA29!•BA28!•BA27!•BA26•...•BA18!•BA17!•BA16!) •BE3

Attenzione: i dispositivi hanno PINs di indirizzo A14...A0 che andrebbero rispettivamente collegati agli indirizzi del bus BA16-BA2 (BA1 e BA0 sono sostituiti da BE3-BE0). BA16 però è utilizzato anche nella decodifica in quanto l' indirizzo iniziale del banco NON è multiplo di 128K. Ne discende che il A14 della RAM deve essere collegato a ....

## 4. Linguaggio macchina

L'insieme delle istruzioni e dei registri di una CPU costituiscono l'**Instruction Set Architecture (ISA)** attraverso il quale è possibile accedere alle risorse interne (es. registri) ed esterne (es. memoria). Ogni CPU possiede un proprio ISA, ed esistono due linee di pensiero:

- **RISC (Reduced Instruction Set Computer)** → insieme ridotto di istruzioni semplici → molti registri interni (DLX, ARM, RISC-V, etc)
- **CISC (Complex Instruction Set Computer)** → insieme ampio di istruzioni complesse → pochi registri (Intel X86)

Un requisito fondamentale di un linguaggio macchina/ISA è quello di minimizzare il tempo di esecuzione del codice:

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

dove il  $CPI_{medio}$  è il numero medio di clock per l'esecuzione di un'istruzione e  $T_{CK}$  è il periodo di clock. Per minimizzare il tempo di esecuzione si può ridurre il periodo di clock (soluzione più semplice ma

aumenta i consumi di energia e in più bisogna stare attenti ai tempi di setup ed hold legati alla rete), il CPI medio o il numero di istruzioni (ciò va a sfavore dei processori RISC però, perché implica istruzioni più complesse).

Le **istruzioni eseguibili** da una CPU tipicamente sono molto semplici → operazioni aritmetiche, lettura/scrittura da memorie periferiche, confronti tra operandi e operazioni di salto condizionati e non ([go to](#)).

Ogni CPU possiede un certo numero di **registri** accessibili al programmatore; il numero e la dimensione dei registri dipendono dall'ISA (con impatto sulla rete logica risultante). Ovviamente avere molti registri general purpose è vantaggioso → meno accessi alla *lenta* memoria!



Meno operazioni di accesso alla memoria ci sono, meglio è → meglio avere tanti registri!

## 4.1 Codifica binaria delle istruzioni

Le istruzioni, per essere eseguite dalla rete logica CPU, devono essere codificate in binario secondo il datasheet del produttore. Esistono CPU con codifica delle istruzioni a lunghezza:

- costante (es. 32 bit nel caso RISC DLX e molti altri)
- variabile da istruzione a istruzione (Intel X86)

Esempio: LB R7,0800(R3) - "Leggi un BYTE (8 bit)  
all'indirizzo R3 + 0800h e trasferisci nel registro R7"

**LOAD BYTE** → accede in lettura allo spazio di indirizzamento

A	3	5	7	0	8	0	0
---	---	---	---	---	---	---	---

Ipotetica codifica dell'istruzione con 32 bit. I bit non utilizzati per codificare R3, R7 e 0800 rappresentano il codice operativo (op code)

## 4.2 Linguaggio assembly

Il linguaggio assembly ha lo scopo generale di consentire al programmatore di ignorare il formato binario del linguaggio macchina → ogni codice operativo del linguaggio macchina viene sostituito, nell'assembly, da una sequenza di caratteri che lo rappresenta in forma mnemonica. In secondo luogo i dati e gli indirizzi di memoria vengono scritti nella base numerica più consona ma anche in forma simbolica (identificatori). Il programma assembly (assembler è l'“assemblatore”) risulta così più leggibile mantenendo però completo isomorfismo con il linguaggio macchina.

Ogni CPU ha il suo proprio assembly.

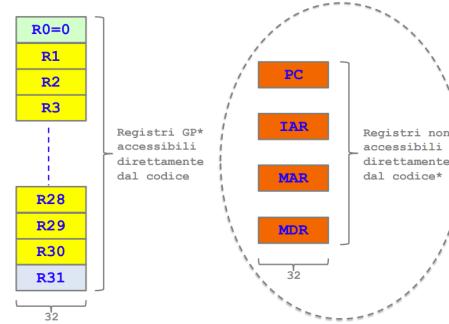
Macchina	->	Assembly
014FA27Dh	->	ADD R1,R2,R3 // R1 = R2 + R3

## 4.3 Caratteristiche dell'ISA DLX

Il DLX (*delux*) è un microprocessore caratterizzato da un **unico spazio di indirizzamento** di 4G (4 bus di dati) e da **32 registri da 32 bit general purpose**. Ogni registro è a 32 bit, ed il registro R0 è vincolato ( $R0 = 0$ ) a sola lettura → anche il registro R31 non si può scrivere (utilizzato per memorizzare gli indirizzi di ritorno delle istruzioni di tipo JAL e JALR).

Oltre ai registri base general purpose del DLX, sono presenti anche altri registri non accessibili direttamente tramite codice:

- **PC (Program Counter)** → è il registro che contiene l'indirizzo (puntatore) alla prossima istruzione da eseguire/su cui fare fetch;
- **IAR (Interrupt Address Register)** → registro di indirizzamento delle interruzioni, contiene l'indirizzo di ritorno dalle routine di interrupt, memorizza il program counter nel momento in cui si interrompe il main → serve per poi ritornarci;



- **MAR** → registro di indirizzamento della memoria → contiene l'indirizzo di memoria a cui accedere per la lettura o scrittura dei dati
- **MDR** → registro di lettura e scrittura della memoria → contiene i dati da leggere o scrivere;
- **TEMP** → registro temporaneo → memorizzazione dati intermedi;

Nel DLX (integer) sono disponibili tre tipi di dato:

- **byte** → 8 bit
- **half-word** → 16 bit
- **word** → 32 bit

In ogni registro devono esserci dati da 32 bit. Se i dati sono da 8 bit, devono essere **estesi mantenendo il segno** (dove presente) a 32 bit. Ad esempio:

```
Prendendo il byte 10110101
assumendo il dato unsigned, aggiungiamo 24 zeri:
00000000000000000000000010110101 oppure (0)24##10110101

assumendo il dato signed, l'estensione avviene replicando 24 volte il bit d
i segno:
1111111111111111111111110110101 -> (1)24##10110101

COMPLEMENTO A 2 NEL CASO BINARIO -> A = NOT(A) + 1
```

#### 4.4 Formati di istruzioni

Ogni istruzione è di lunghezza costante di 32 bit e deve essere allineata in memoria. Esistono 3 formati di istruzioni:

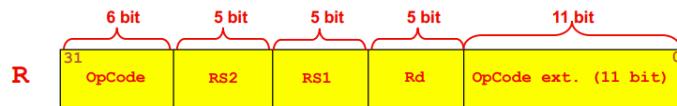
- **I-type (inmediate)** → istruzioni che utilizzano 2 operandi riferiti a registri e un operando immediato da 16 bit con segno → usate per l'accesso diretto alla memoria e il trasferimento dati verso/da i registri;





I = immediato (AND tra 1 registro e una costante a 16 bit) → senza I = tra registri

- **R-type (register)** → istruzioni che lavorano esclusivamente tra registri, utilizzano un operatore e 3 operandi riferiti a registri (32 bit di lunghezza) → usate per calcolo numerico su interi con **ALU**;



- **J-type (jump)** → istruzioni di salto che contengono un operatore e un offset da 26 bit con segno che, sommato all'indirizzo contenuto nel PC, fornisce l'indirizzo di destinazione del salto → usate per implementare il controllo del flusso di un programma.



Non sono presenti istruzioni per gestire lo stack, il DLX non ne ha il supporto hardware → si può fare manualmente utilizzando un registro tramite operazione di confronto;

Anche per quanto riguarda la gestione dei salti, nel caso di un jump if bisogna prima fare il confronto tra valori nei registri e salvarne l'esito in un altro registro, successivamente eseguire il salto condizionato (branch) → condizione e poi azione.



**NON** si possono fare operazioni direttamente tra registri e memoria; solo registri-registri o registro-costante

## 4.5 Linguaggio assembly DLX

Sintassi di base:

```
<command> <operand register> <immediate>
<command> <target register> <operand register> <operand register>
<move command> <target> <source>
<branch command> <label of place in code to go to>
```

### Istruzioni Aritmetico Logiche (ALU)

```
# Istruzioni di somma
ADD      Ra,Rb,Rc
ADDI     Ra,Rb,I   // I = immediato = costante a 16 bit
ADDU     Ra,Rb,Rc  // U sta per unsigned
ADDUI    Ra,Rb,I

# Sottrazione
SUB     Ra,Rb,Rc
```

```

SUBI      Ra,Rb,I
SUBU      Ra,Rb,Rc
SUBUI     Ra,Rb,I

# Moltiplicazione e divisione
DIV       Ra,Rb,Rc
DIVI      Ra,Rb,I
MUL       Ra,Rb,Rc
MULI      Ra,Rb,I

# Istruzioni di shift
SLL       Ra,Rb,Rc // shift logico a sinistra
SLLI     Ra,Rb,I
SRL       Ra,Rb,Rc // e a destra
SRLI     Ra,Rb,I
SRA       Ra,Rb,Rc // shift aritmetico a destra
SRAI     Ra,Rb,I

# Istruzioni logiche
OR        Ra,Rb,Rc
ORI      Ra,Rb,I
XOR      Ra,Rb,Rc
XORI     Ra,Rb,I
AND      Ra,Rb,Rc
ANDI     Ra,Rb,I

# Load High immediato
LHI      Ra,I
// crea un indirizzo in un registro -> piazza i 16 bit dell'immediato ndllz
// porzione più significativa del registro dest. e riempie il resto con 0
// utile x creare costanti a 32 bit in un registro in cui possiamo impostar
// e gli ultimi 16 bit con ADDI

# Istruzioni di SET CONDITION -> SETx, con x = {EQ,NE,LT,GT,LE,GE}
// confrontano le sorgenti e mettono a 1 o 0 la dest. in base al risultato
Sx       Ra,Rb,Rc // Ra non può essere R0 o R31
SxI      Ra,Rb,I

```

### Istruzioni per il **trasferimento dati**

```

# Istruzioni di load
LW      Ra,I(Rb) // load word - esempio: LW R8,0(R18)
// carica 32 bit dall'indirizzo R18+0 e li
mette in R8, deve essere
// allineato o eccezione! (R18 divisib

```

```

ile x 4 = ultimi 2 bit a 0)
LB      Ra,I(Rb) // load byte -> sempre allineato
LBU     Ra,I(Rb)
LH      Ra,I(Rb) // load halfword -> indirizzo deve essere pari
LHU     Ra,I(Rb)

# Istruzioni di store
SW      Ra,I(Rb) // store word
SH      Ra,I(Rb) // store halfword
SB      Ra,I(Rb) // store byte

# Istruzioni che copiano un dato da un registro GP a uno speciale e viceversa
MOVS2I Ra,Rs*    // special register Rs* (IAR)
MOVI2S Rs*,Ra

```

### Istruzioni per il trasferimento del controllo

```

# Istruzioni di salto condizionato (branch)
BEQZ    Ra,I    // I = target, se = 0 salta
BNEQZ   Ra,I    // se != 0 salta

// prima dell'istruzione di salto c'è un'istruzione di SET
// l'indirizzo di dest. è calcolato sommando a I esteso in segno il program counter + 4 -> il salto è relativo a PC+4
// se I < 0 saltiamo indietro, se I > 0 in avanti

# Istruzioni di salto incondizionato (jump)
J       I        // salto relativo a PC+4
J       Ra       // salto assoluto senza ritorno, va dritto all'indirizzo

# Istruzioni di jump and link (salto incondizionato con ritorno)
JAL    I
JAL    Ra

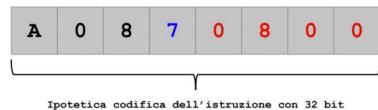
```

## 4.6 Modalità di accesso alla memoria

- Indirizzamento diretto

Con questa modalità l'istruzione contiene al suo interno un valore (cablato) che specifica l'indirizzo di accesso alla memoria.

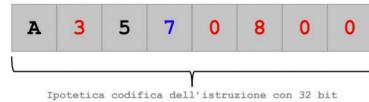
**LB R7, 0800h** → “leggi un byte all'indirizzo 0800h e memorizzalo nel registro R7”



- **Indirizzamento indiretto**

Con questa modalità l'indirizzo di accesso alla memoria è ottenuto sommando un valore costante presente nell'istruzione con il contenuto di un registro. **Indirizzo = costante + registro** → il registro è cablato nell'istruzione ma il suo contenuto può cambiare a runtime.

`LB R7, 0800(R3)` → "leggi un byte all'indirizzo R3 + 0800h e memorizzalo nel registro R7"



Chiaramente per alcune cose (ad esempio lettura di un array) l'indirizzamento diretto è molto più scomodo (LB per ogni elemento dell'array a quell'indirizzo) quindi conviene utilizzare l'indirizzamento indiretto + LOOP. Ad esempio:

```
XOR R9, R8, R8      ; R8 = 0 - inizializzo R8
ADDI R9, R8, 8       ; R9 = 8 - indice x accesso all'array

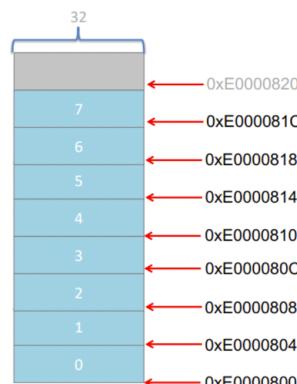
LOOP: SUBI R9, R9, 1    ; decremento l'indice - PC-16
      LBU R7, 0800h(R9) ; PC-12
      ADD R8, R8, R7      ; PC-8
      BNEZ R9, LOOP       ; se R9 != 0, salto a LOOP - PC-4
```

E come sono memorizzati i dati in memoria in un sistema con parallelismo >8? → *Big Endian* (indirizzi significativi a indirizzi superiori) e *Little Endian* (indirizzi significativi a indirizzi inferiori).

## 4.7 Esempi di codice Assembly DLX

### Esempio codice assembly DLX 1

Scrivere il codice assembler DLX per inserire in memoria, a partire dall'indirizzo E0000800h l'array di word indicato in figura.



```
init:   ADD R1, R0, R0      ; R1 = 0 -> valore da inserire
        ADDI R2, R0, 0x0008 ; R2 -> indice
        LHI R3, 0xE000          ; R3 = 0xE0000000 -> indirizzo
        ADDI R3, R3, 0x0800 ; R3 = 0xE0000800

loop:   SUBI R2, R2, 0x0001 ; R2 = R2 - 1
        SW R1, 0x0000(R3)
```

```

        ADDI R1,R1,0x0001 ; R1 = R1 + 1
        ADDI R3,R3,0x0004 ; R3 = R3 + 4 -> sommo 4 perchè sto
salvando una word (occupa 4 indirizzi)
        BNEZ R2,loop      ; R2 != 0 -> loop

```

### Esempio di codice assembler DLX 2

Il codice seguente è corretto?

```

LHI   R1,0x0000
ADDI  R2,R0,0x0081
SH    0x7FF1(R1),R2
LHU   R3,0x7FF1(R1)
LH    R4,0x7FF1(R1)

; fino alla seconda riga tutto ok, abbiamo R1 = 0x00000000, e R2 = 0x0000000
81
; salva halfword (16 bit) in R1 + 0x7FF1 però l'indirizzo non è pari -> err
ore
; idem per LHU ed LH non vengono proprio eseguiti

```

## 5. Interruzioni

In un sistema a microprocessore è di fondamentale importanza poter gestire eventi che si verificano all'esterno (ma non solo) della CPU. Una strategia poco efficiente è quella di interrogare le periferiche a **polling**, spendendo molti cicli macchina per la verifica.

Una strategia molto più efficiente, basata su una strategia “push”, consiste nell'uso di **interrupt**.

Un interrupt è un evento che interrompe la CPU durante il regolare flusso di esecuzione del codice → segnala che si è verificato un evento che merita immediata attenzione da parte della CPU → se abilitata, quest'ultima esegue un **interrupt handler**. Gli eventi possono essere interni (divisione per zero, overflow) o esterni (premuto un tasto) → **exceptions**. Nel DLX abbiamo un solo interrupt: INT.

Il **PIC (Program Interrupt Control)** è la rete che decide come gestire gli interrupt e aiuta la CPU. Noi invece gestiremo tutto via software. Quando un interrupt si verifica, il main viene messo in pausa “segnando” l'istruzione successiva a quella che è stata interrotta, poi il DLX fa fetch all'indirizzo zero e gestisce l'evento, dopodichè ritorna ad eseguire il main (prima si termina sempre l'istruzione in corso!) → una specie di jump and link → l'interrupt handler termina sempre con l'istruzione **RFE** return from exception → l'indirizzo di ritorno è salvato in IAR (Interrupt Address Register). → **nell'ISA DLX è gestito un solo indirizzo di ritorno.**

Nel caso del DLX, la CPU è sensibile al livello del segnale di interrupt (1 se c'è una richiesta pendente, 0 se l'evento è stato gestito o non ci sono interruzioni).

Come fare se il dispositivo che genera l'interruzione assume che la CPU sia sensibile ai fronti e non al segnale? Occorre eseguire una trasformazione da fronte a livello del segnale INT\_FRONTE (FFD con A\_RESET a CS\_RESET\_INT).

### 5.1 Interrupt nel DLX

Il DLX ha un solo segnale di interrupt → come fare allora a gestire multiple sorgenti di interrupt?  
 Convogliando tutti gli interrupt verso l'unico segnale INT presente nel DLX. In seguito è necessario però associare un livello di priorità a ciascuna interruzione e poter interrompere l'interrupt handler in esecuzione se giunge una richiesta di interrupt più prioritaria (*annidamento*).

L'annidamento **non è previsto** dal DLX base, per poter annidare gli interrupt sarebbe necessario uno *stack software* → **il DLX disabilita le interruzioni mentre esegue l'handler**.

Nell'handler, inoltre, è molto importante salvare e ripristinare i registri modificati dallo stesso handler per garantire che il codice del main mantenga la consistenza dei dati ed evitare che utilizzi un registro modificato dall'handler (non deve interferire!).

Quindi la struttura di un handler è tipo: istruzioni che salvano i registri (che verranno modificati dalle istruzioni seguenti) → handling → ripristino registri → RFE (PC ← IAR). Nel caso di multipli interrupt, semplicemente gli handler si susseguono.

## 5.2 Programmable Interrupt Controller (PIC)

Il PIC si occupa di gestire multiple sorgenti di interruzione e fornire direttamente alla CPU (su richiesta) qual è il codice/indirizzo dell'interrupt più prioritario tra quelli asseriti.

Le varie sorgenti di interruzioni `INT[7..0]` sono inviate al PIC che si occupa di inviare la richiesta sull'unico pin INT del DLX. Nei sistemi a microprocessore, inoltre, ci sono tipicamente dei timer sui primi 6 interrupt → nel PIC c'è anche un segnale di WRITE per consentirne la programmazione.

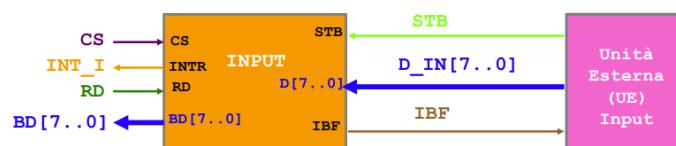
In una CPU (ma non nel DLX) può essere presente un ulteriore segnale (in input) denominato **NMI** (*Not Maskable Interrupt*) → tale segnale sono collegate un numero limitato di sorgenti di interruzioni particolarmente critiche (tipo imminente perdita di alimentazione elettrica). Una richiesta di interrupt inviata sul pin NMI non può essere ignorata e interrompe l'esecuzione di altri handler → priorità massima.

## 6. Periferiche di I/O con handshake

In precedenza abbiamo visto come la CPU è in grado di scambiare dati col mondo esterno tramite un buffer; tuttavia non vi era nessuna garanzia sul corretto esito dei trasferimenti → cosa accade se mentre la CPU scrive su una porta in output un dispositivo esterno legge dalla stessa porta? → i trasferimenti sono intrinsecamente esposti a errori.

La soluzione a questo problema è la **sincronizzazione** → protocollo di handshake.

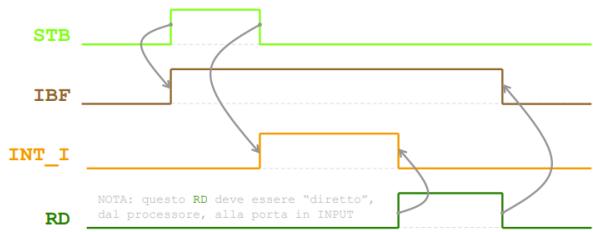
Nel caso di una comunicazione in **input**:



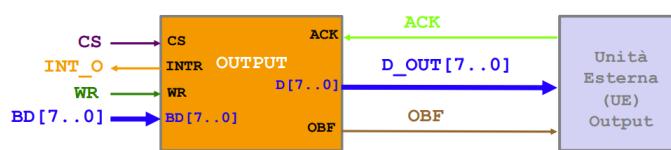
1. Se `IBF = 0`, quando possibile UE può scrivere il dato nel *buffer d'ingresso* della porta
2. UE, portando **STB** a 1, scrive il dato nella porta (fronte di discesa di strobe) che contemporaneamente asserisce **IBF (Input Buffer Full)** → (segnale a livello)
3. Quando UE porta STB a 0 (scrittura terminata), l'interfaccia attiva `INT_I` (*Interrupt Request*)

4. Quando possibile, la CPU andrà a leggere il dato scritto nella porta da UE. Al termine, IBF = 0 (mentre INT\_I = 0 dall'inizio della lettura)

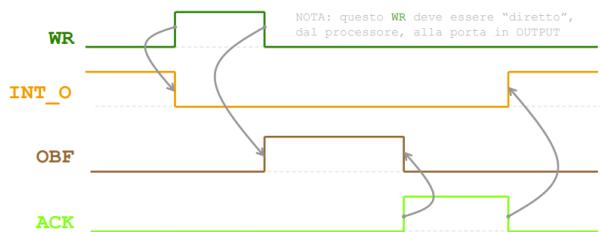
Il comportamento è asincrono!



Nel caso di comunicazione della CPU in **output**:

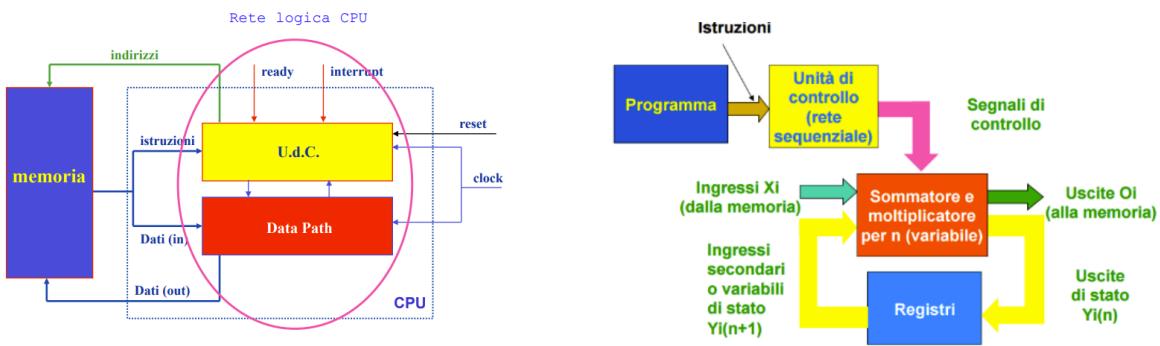


1. Il segnale **INT\_O** asserito comunica alla CPU che la porta può accettare un nuovo dato
2. In rispetto alla richiesta di interrupt la CPU scrive, quando possibile, il dato sul buffer della porta
3. L'interfaccia segnala a UE che è disponibile un nuovo dato attivando **OBF** (**Output Buffer Full**) → ha uno spazio di 8 bit
4. Quando possibile, UE legge il dato scritto dalla CPU asserendo ACK.

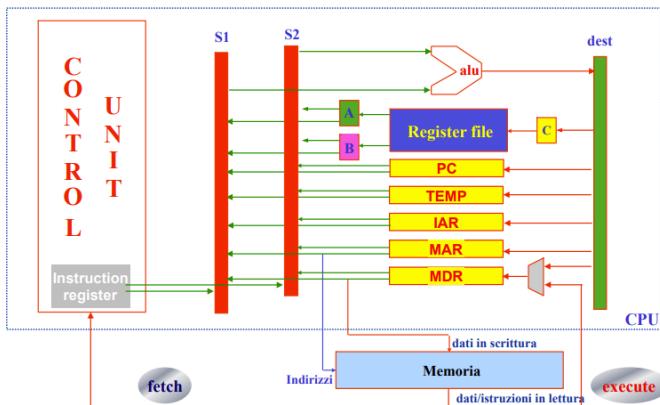


## 7. DLX Sequenziale

La struttura di una CPU, come tutte le reti logiche sincrone che elaborano dati, può essere strutturata in due blocchi: **Unità di Controllo** e **Datapath**. La CPU, per funzionare, ha bisogno della memoria esterna su cui risiedono il programma e i dati.



- **Datapath** → contiene tutte le unità di elaborazione e i registri necessari per l'esecuzione delle istruzioni della CPU. Ogni istruzione appartenente all'ISA è eseguita mediante una successione di operazioni elementari dette *micro-operazioni*.
  - **Micro-operazione** → operazione eseguita all'interno del datapath in un ciclo di clock (ALU, trasferimento di un dato da un registro a un altro)
- **Unità di Controllo** → è una rete sequenziale sincrona che in ogni ciclo di clock invia un ben preciso insieme di segnali di controllo al datapath al fine di specificare l'esecuzione di una determinata *micro-operazione*.



Parallelismo dell'architettura: 32 bit (bus, alu e registri hanno parallelismo 32)

La parte del Datapath è composta da:

- **Register file** → 32 registri general purpose R0..R31 con R0 = 0
- **PC** → Program Counter → contiene l'indirizzo dell'istruzione a cui la memoria deve fare il fetch
- **TEMP** → *Temporary Register* → contiene l'istruzione attualmente in esecuzione → non accessibile
- **IAR** → *Interrupt Address Register* → deposito dell'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* → registro di transito temporaneo dei dati da e per la memoria
- **IR** → *Instruction Register* → contiene l'istruzione attualmente in esecuzione
- **A e B** → registri di uscita dal Register File.

Si noti che se si vuole passare da un bus sorgente al bus destinazione, si deve **necessariamente passare per l'ALU**: per esempio, se si prende il contenuto del registro TEMP e lo si scrive dentro MAR, l'unità di controllo piloterà il 3-state per connettere TEMP a S2, abiliterà il WE sul registro MAR e farà sì che l'operazione ALU sia l'identità (fai passare ciò che sta nel bus2 → s2+0) arrivando sul bus DEST e vedendo WE sul MAR attivo il dato finirà dentro MAR.

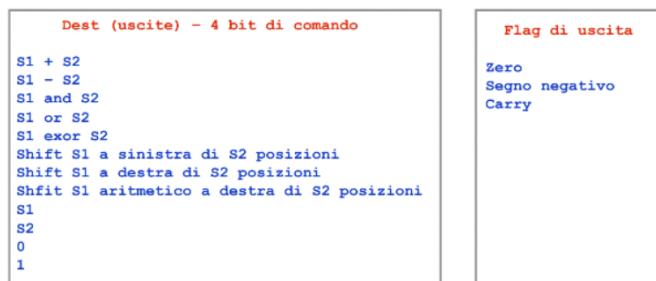
Qualsiasi dato che viene letto o scritto in memoria passa necessariamente da MDR.

▼ *Ma come fa il DLX a generare BE0, BE1, BE2, BE3 dal MAR?*

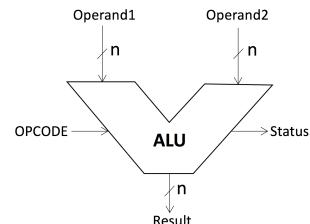
Un decoder non è sufficiente, è necessario conoscere che tipo di istruzione è in esecuzione → l'unità di controllo conosce il codice operativo dell'istruzione dell'instruction register:

- se word → unità di controllo si aspetta BA0 e BA1 = 0 e attiva BE0 BE1 BE2 e BE3
- se byte → unità di controllo attiva uno solo dei BE → gli ultimi 2 bit specificano quale dei 3
- se halfword → unità di controllo si aspetta BA0 = 0 e attiva due BE

## 7.1 Funzioni della ALU



La ALU (*Arithmetic Logic Unit*) è una rete puramente **combinatoria** in grado di eseguire operazioni aritmetiche e logiche sui due operandi in base al codice operativo (op-code) in ingresso. Oltre a fornire il risultato dell'operazione, indicano anche attraverso una serie di **flag** il verificarsi di particolari condizioni nel risultato stesso.



L'unità di controllo ha accesso totale ai flag, ma nel DLX non esiste un *registro di flag* che ne registri valori → questo è un vantaggio, perchè nei processori CISC (oltre ad esserci il problema della variabilità della lunghezza delle istruzioni e il non poter usare indiscriminatamente ogni registro) bisogna andare ad interrogare i registri di flag della ALU.

## 7.2 Trasferimento dati sul datapath

I bus S1 ed S2 sono multiplexati (3-state) con parallelismo a 32 bit, perchè in essi sono connessi più dispositivi.

I registri compaiono **sul fronte positivo** del clock, hanno due porte di uscita O1 e O2 per i due bus (o i registri A e B) e dispongono di **tre ingressi di controllo**:

- Un ingresso Write Enable →  $WE^*$
- Due ingressi Output Enable →  $OE^*$  → uno per ogni porta di uscita (bus S1 ed S2)

Al fine di valutare la massima frequenza a cui è possibile far funzionare il datapath è importante conoscere le seguenti temporizzazioni:

- $T_C(max)$  → ritardo max 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 max tra l'arrivo del segnale OE e l'istante in cui i dati del registro sono disponibili sul bus;
- $T_{ALU}(max)$  → ritardo massimo introdotto dalla ALU;
- $T_{SU}(min)$  → tempo di *set-up* minimo dei registri → requisito minimo per il corretto campionamento da parte dei registri;

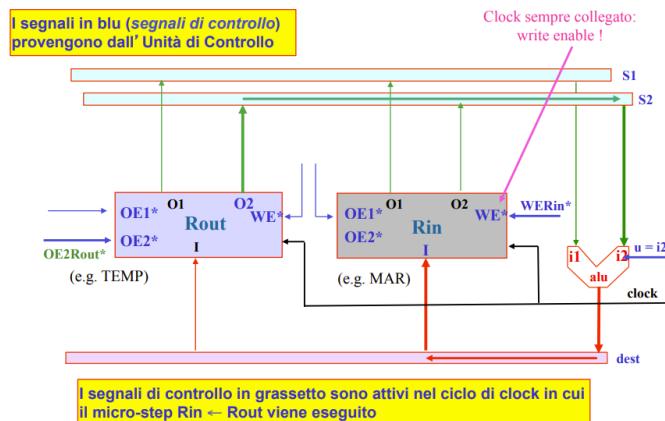
La massima **frequenza di funzionamento** del datapath si calcola come segue:

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

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

Per aumentare la frequenza massima si deve diminuire almeno uno dei vari tempi, usando (per esempio) una tecnologia migliore.

Analizzando la struttura del datapath, si visualizzano i due registri TEMP e MAR:



Se si vuole inviare un dato da ROUT a RIN si deve connettere O2 di ROUT al bus S2 e tale operazione viene fatta dall'unità di controllo togliendo il 3-state su O2. I dati circolano sul bus S2 ed entrano nella ALU, uscendo senza essere modificati; il dato arriva sul bus DEST e poi in ingresso su RIN → il dato arriva su tutti i registri collegati a DEST, ma RIN è l'unico su cui è stato abilitato WE.

### 7.3 Progetto dell'Unità di Controllo

Una volta definito il *set di istruzioni* e progettato il *datapath*, il passo successivo del progetto di una CPU è il progetto dell'Unità di Controllo. Il **CONTROLLER** è una RSS: il suo funzionamento può essere specificato tramite un *diagramma degli stati*.

Il controller, come tutte le RSS, permane in un determinato stato per un ciclo di clock e transita (*può transitare*) da uno stato all'altro in corrispondenza degli istanti di sincronismo (fronti del clock).

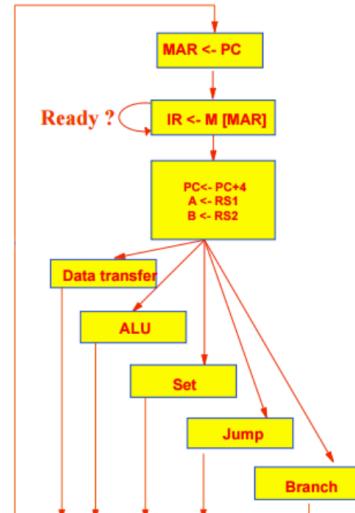
Ad ogni stato corrisponde quindi un ciclo di clock. Le *micro-operazioni* che devono essere eseguite in quel ciclo di clock sono specificate (in linguaggio RTL - *Register Transfer Language* → descrizioni di alto livello

di un circuito) nel diagramma degli stati che descrive il funzionamento del controller all'interno degli stati.

A partire dalla descrizione RTL si sintetizzano poi i segnali di controllo che devono essere inviati al datapath per eseguire le operazioni elementari associate ad ogni stato.

Ogni volta, riesegue le stesse operazioni: legge l'istruzione, la decodifica e una volta codificata la esegue.

Per prima cosa si deve trasferire il PC nel MAR, dopodiché il controller va a caricare (nell'Instruction Register) ciò che è all'indirizzo di memoria che punta il PC (che nello stato precedente si è trasferito nel MAR). A tal punto termina la fase di fetch.

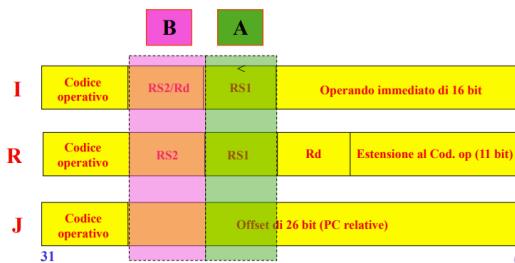


Esempio di diagramma degli stati di un controller

Mentre il controller decodifica l'istruzione, il DLX esegue l'operazione  $PC+4$ , ovvero punta all'istruzione successiva. Contemporaneamente mette nei registri A e B i due registri sorgente usati dal register file.

Una volta codificata l'istruzione e quindi stando sul successivo fronte di clock, il controller avrà terminato la decodifica e potrà passare alla fase di Execute, che dipende dal tipo di istruzione.

**Estrazione “automatica” dei registri durante la fase di decode di una qualsiasi istruzione:**



Questi 5+5 bit sono utilizzati per estrarre, preventivamente e ancor prima di conoscere che tipo di istruzione è stata letta dalla memoria, dal Register File due registri in A e B.

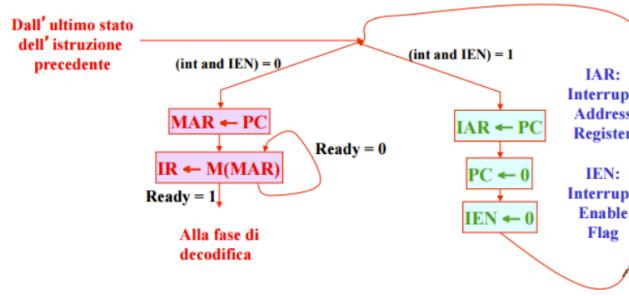
Nel caso di istruzione J non ci sono registri coinvolti e quindi saranno estratti i bit corrispondenti all'offset; nel caso di istruzione I, in B potrebbe finire il valore del registro destinazione. Infine i 5+5 bit rappresentano gli indici ma non il valore dei due registri contenuti nel Register File.

### Stati della fase di fetch

In questa fase si deve verificare se è presente un interrupt.

Se l'interrupt non è presente o le interruzioni non sono abilitate si va a leggere in memoria la prossima istruzione da eseguire il cui indirizzo è in PC (fetch tradizionale).

Se l'interrupt è presente e può essere usato (`IEN = true`), si esegue implicitamente l'istruzione di chiamata a procedura all'indirizzo 0 e si salva l'indirizzo di ritorno nell'apposito registro IAR.



Modificando il datapath in modo da poter indirizzare la memoria dal PC (senza passare per il MAR) si hanno meno stati ma maggiore complessità → tutte le istruzioni impiegano un clock in meno per essere eseguite → ma potenzialmente maggiore lentezza (minore frequenza di clock).

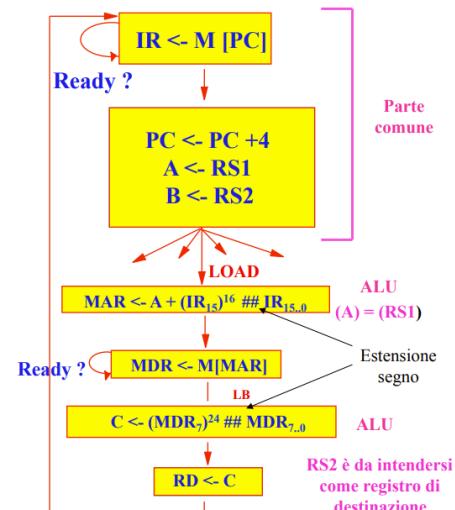
### Controllo per l'istruzione LB (Load Byte)

Avendo finito di decodificare l'istruzione, a questo punto l'unità di controllo sa che è una load byte e deve essere letto un byte all'indirizzo di memoria contenuto nell'istruzione stessa comando il registro sorgente con l'immediato esteso in segno → il dato che viene letto dalla memoria deve essere trasformato dagli 8 bit ai 31 bit (gli 8 bit devono essere interpretati con segno).

Quindi l'ALU prende A e l'unità di controllo va ad estendere il segno del proprio immediato che contemporaneamente viene sommato → il risultato viene memorizzato nel MAR.

Il contenuto della memoria all'indirizzo presente nel valore nel MAR viene letto (ready = 0 oppure = 1 in base alla situazione) e viene memorizzato nel MDR. Quando  $\text{ready} = 1$  → operazione terminata, altrimenti si fa un nuovo ciclo di clock.

A questo punto il dato viene letto dalla memoria e nel registro C vengono scritti gli 8 bit di dato letti dalla memoria concatenati con i 24 bit di segno appartenenti al dato letto.



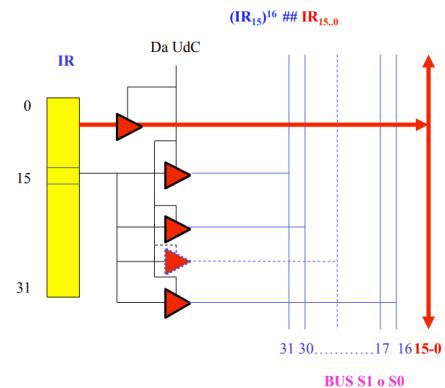
Nel clock successivo si entra nel register file e il dato C viene scritto nel registro destinazione (RD).



Non si può leggere il dato dalla memoria e scriverlo direttamente in RD!!! (prima del RF è presente il registro C)

Quest'istruzione ammette due estensioni di segno:  
una per calcolare l'indirizzo di lettura e uno per  
determinare il valore del dato letto dalla memoria  
→ viene eseguita dai 3-state che replicano 32  
volte il bit 15 che andrà connesso in modo  
opportuno al bus S1 o S2.

Non posso usare un solo 3-state altrimenti  
cortocircuito, si avrebbero tutti i fili a 0 o a 1.



### Controllo per le istruzioni di DATA TRANSFER

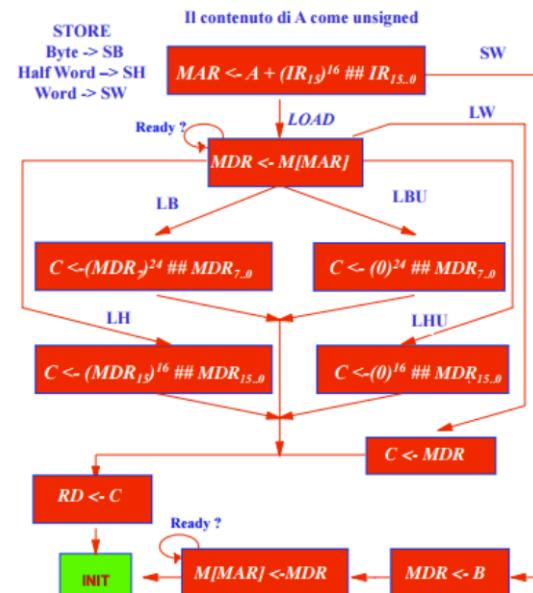
Prendendo in esame una SW (store word → scrive il contenuto del registro in memoria): viene inserito nel MAR un valore (RS1 + immediato esteso)

Quindi l'indirizzo è formato nel DLX per accedere alla memoria sia per una LOAD che per una STORE, sempre nello stesso modo.

Procedendo con il caso di SW, si vuole prendere un registro (contenuto nel RF) quindi si trasferisce B (che contiene SR2) in MDR, ovvero si scrive in MDR all'indirizzo di memoria dato da MAR.

Quest'operazione dura 1 clock se la memoria è pronta a ricevere il dato, altrimenti 2 o più clock.

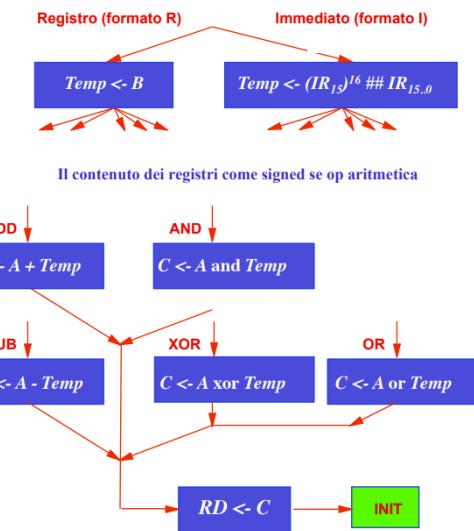
Nel caso di store non si ha mai estensione del segno!



### Controllo per le istruzioni ALU

Viene uniformato tutto sia per istruzioni di tipo R che per istruzioni di tipo I → si passa sempre per il registro TEMP. Per le istruzioni di tipo R semplicemente si trasferisce B in TEMP (non strettamente necessario), si esegue l'istruzione ALU e si memorizza in C il risultato dell'operazione.

Analogamente per un'operazione con immediato si trasferisce il valore esteso a 32 bit in TEMP, si esegue l'operazione tra A e TEMP e poi si scrive il contenuto di C nel RF (nel RD).

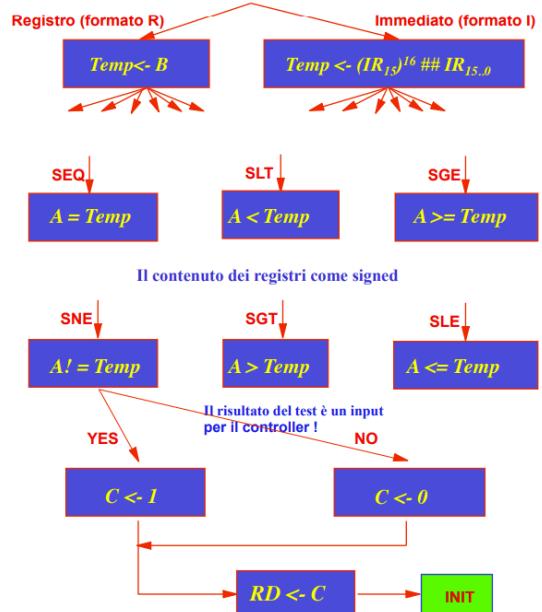


### Controllo per le istruzioni di SET - confronto

Anche qui si uniforma tutto sia per istruzioni R che I e si passa per TEMP. Si va a verificare qual è la condizione e in seguito l'unità di controllo, in base al contenuto del flag, va a capire come settare il RD in base all'esito dell'operazione.

I flag non vengono campionati ma vengono utilizzati dall'unità di controllo DLX per andare a definire se nel RD ci dovrà andare un 1 o uno 0.

Per fare un confronto tra registri sorgenti TEMP non è necessario, l'unità di controllo potrebbe programmare la ALU per fare un confronto → se invece l'istruzione prevede un immediato si passa da TEMP (che conterrà l'esteso immediato)

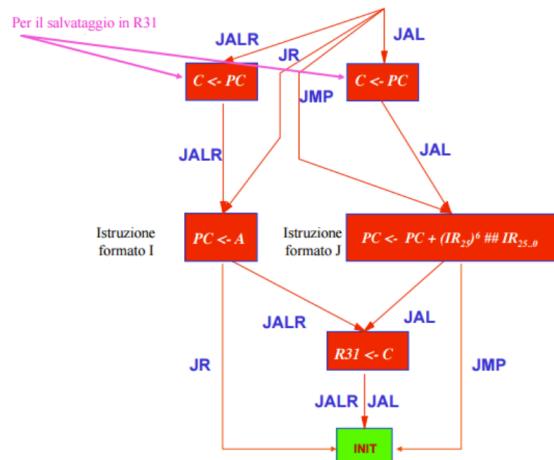


### Controllo per le istruzioni di JUMP

Abbiamo istruzioni di due tipi: JUMP e JUMP\_AND\_LINK → quest'ultima prevede che il PC sia salvato in R31.

Le jump non tornano quindi non salvano il PC in R31 → jmp altra notazione per j = salto incondizionato.

Il salto condizionato è il branch (equal or notequalzero)

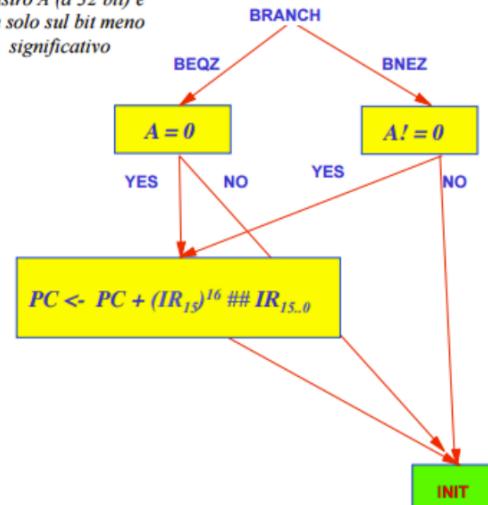


### Controllo per le istruzioni di BRANCH

In questo caso salto se il registro è o non è zero. Come si calcola l'indirizzo? PC + immediato esteso (con ALU) → estensione analoga a quella di prima con i 3-state. Dopodichè viene aggiornato il valore del PC (se la condizione è verificata, altrimenti non si aggiorna un bel niente).

All'esame BRANCH etichetta → poi scrivo etichetta = valore\_esadecimale

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



### 7.4 Numero di clock necessari per eseguire le istruzioni

I vari cicli di clock per ogni categoria di istruzione sono illustrati nella tabella:

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

Istruzione	Cicli	Wait	Totale
Branch (not taken)	3	1	4

Il numero di clock medio che all'interno di un programma serve per ogni singola istruzione si calcola:

$$CPI = \sum_{i=1}^n \left( CPI_i * \frac{N_i}{N_{tot\ istruzioni}} \right)$$

## 7.5 Passi dell'esecuzione delle istruzioni

Nel DLX, l'esecuzione di tutte le istruzioni può essere scomposta in **5 passi**, ciascuno eseguito in uno o più cicli di clock.

1. **FETCH** → l'istruzione viene prelevata dalla memoria e posta in IR.

```
MAR ← PC
IR ← M[MAR]
```

2. **DECODE** → l'istruzione in IR viene decodificata e vengono prelevati gli operandi sorgente dal RF.

```
A <- RS1, B <- RS2, PC <- PC+4
```

(eseguite in un solo ciclo di clock tutte e tre nel DLX sequenziale)

3. **EXECUTE** → elaborazione aritmetica o logica mediante la ALU, in base alle operazioni

- MEMORIA:  $MAR <- A + (IR_{15})^{16} \# IR_{15..0};$   
 $MDR <- B;$  (serve nello store, non in load)
- ALU:  $C <- A op B (oppure A op (IR_{15})^{16} \# IR_{15..0});$   
 $C <- \text{sign}(A op B (oppure A op (IR_{15})^{16} \# IR_{15..0}));$  se SCn
- BRANCH:  $TEMP <- PC + (IR_{15})^{16} \# IR_{15..0};$
- J e JAL:  $TEMP <- PC + (IR_{25})^6 \# IR_{25..0};$  → OCCHIO cambiano gli indici!!
- JR e JALR:  $TEMP <- A;$

4. **MEMORY** → accesso alla memoria e in caso di BRANCH aggiornamento del PC.

- MEMORIA:  $MDR <- M[MAR];$  (load)  
 $M[MAR] <- MDR;$  (store)
- BRANCH:  $\text{If(cond) } PC <- TEMP;$  [A] è il registro che condiziona il salto
- JAL e JALR:  $C <- PC;$

5. **WRITE-BACK** → scrittura sul register file.

- Istruzioni J, JR, JAL, JALR:  $PC <- TEMP;$   
 $RD <- C;$
- Altre istruzioni:  $C <- MDR;$  (se è una LOAD → due micropassi)  
 $RD <- C;$

---

## 8. DLX Pipelined

Il *pipelining* è oggi la principale tecnica di base impiegata per rendere “veloce” una CPU. Sapendo che il DLX esegue un certo set di istruzioni impiegando in media sei clock per eseguirle, si vuole ottenere un

Calcolatori Elettronici

45

processore compatibile ed equivalente che esegua lo stesso set di istruzioni ma con un'efficienza maggiore, avente una CPI = 1.

L'idea alla base del pipelining è quella della catena di montaggio: si suddivide il bene che si vuole produrre in tanti micro-passi che verranno eseguiti su un sistema distribuito.

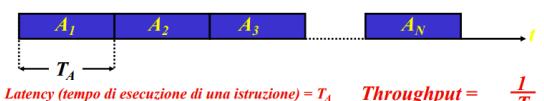
Avendo un sistema, S, deve eseguire N volte un'attività A, ritroviamo quindi due concetti fondamentali:

- **LATENCY** → tempo che intercorre tra l'inizio e il completamento dell'attività →  $T_A$
- **THROUGHPUT** → frequenza con cui vengono completate le attività →  $\frac{1}{T_A}$

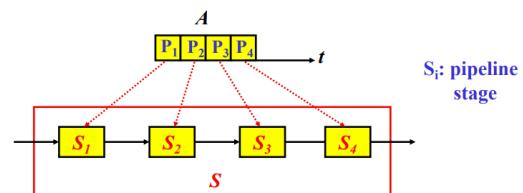
Uno dei nostri problemi potrebbe essere il bus dati → bisogna aspettare i dati, considerando un eventuale ritardo nell'arrivo delle informazioni → dipendenza dai dati che spesso sono in memoria.

Nel caso del DLX sequenziale, finché l'esecuzione di un'istruzione non finiva non poteva partire un'altra; nel caso del DLX pipelined bisogna fare in modo che più istruzioni siano in esecuzione in contemporanea ma nella sequenza inizialmente definita.

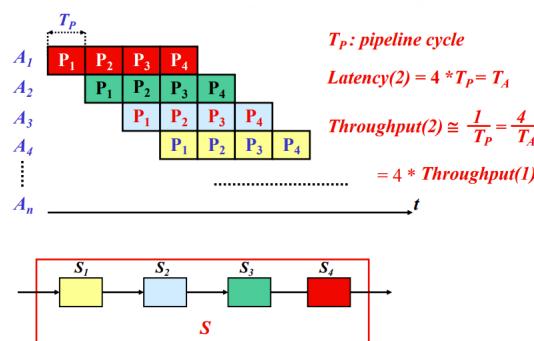
### 1) Sistema Sequenziale



### 2) Sistema in Pipeline

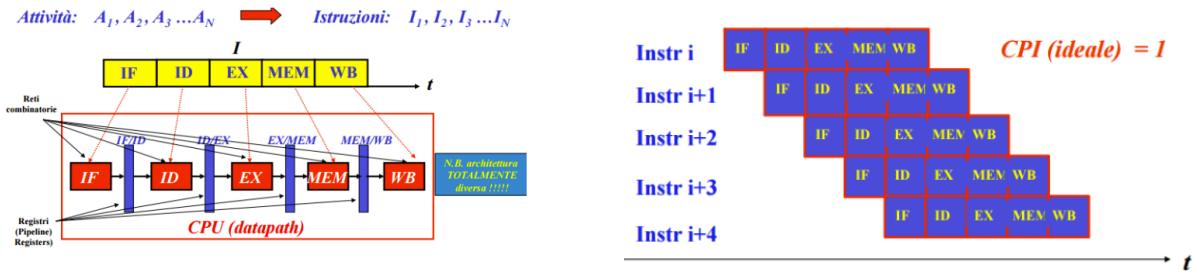


Nell'immagine sotto si ha che la prima attività parte in un istante  $t_0$  e termina in un altro istante  $t$ , si compara di 4 micro-operazioni; se al termine della prima micro-op si fa partire una seconda attività e la vado ad eseguire contemporaneamente ad un certo punto ci si troverà ad avere 4 attività che sono attive, ciascuna in 4 fasi diverse → in un determinato istante 4 attività in 4 fasi diverse. Ad ogni ciclo di clock e quando sono a regime è possibile che una di queste attività si conclude → si riesce ad avere un throughput = 1 sul tempo di ogni micro attività.



## 8.1 Pipelining nella CPU DLX

Prendendo in esame il DLX, è possibile pensare che il pipelining coinvolga 5 attività (fetch, decode, execute, memory, write-back) → tra ciascuna di queste attività si pone una barriera che contiene i registri dove sono memorizzate le informazioni che devono essere portate lungo la pipeline. In questo caso abbiamo un'architettura (ovviamente) diversa, ma ad ogni clock terminerà un'intera istruzione.

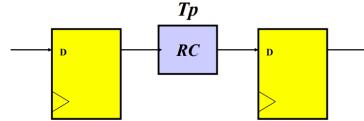


Le barriere blu sono registri edge-triggered per mantenere informazioni strettamente necessarie. Ogni fase dura un certo tempo minimo: frequenza massima → maggiori consumi; tutti gli stadi devono durare lo stesso tempo, per ridurlo al minimo dobbiamo considerare come tempo minimo il tempo maggiore dei 5 stadi della pipeline. Dopo essere andati a regime: *Pipeline Cycle* → *Clock Cycle* = **CPI = 1 (idealemente!)**

Essendo presenti i registri edge triggered (**Pipeline Registers**), ci sono dei tempi da rispettare per far sì che tutto funzioni correttamente → la frequenza massima è quindi condizionata da:

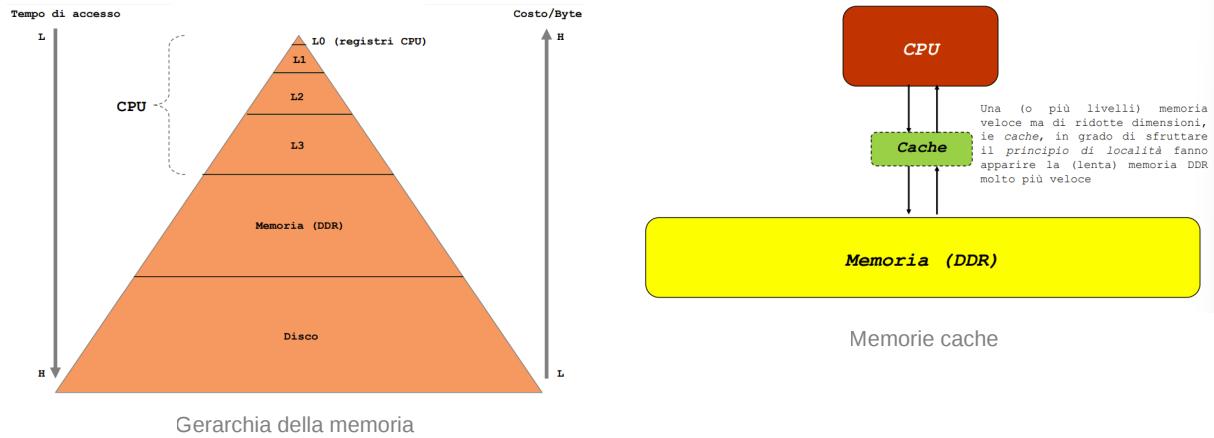
- tempo di ritardo del registro a monte (fornisce dati non al clock ma con un ritardo) →  $T_d$
- tempo dello stadio combinatorio più lento →  $T_p$
- tempo di setup del registro a valle (i ritardi dei registri sono tutti uguali) →  $T_{su}$

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

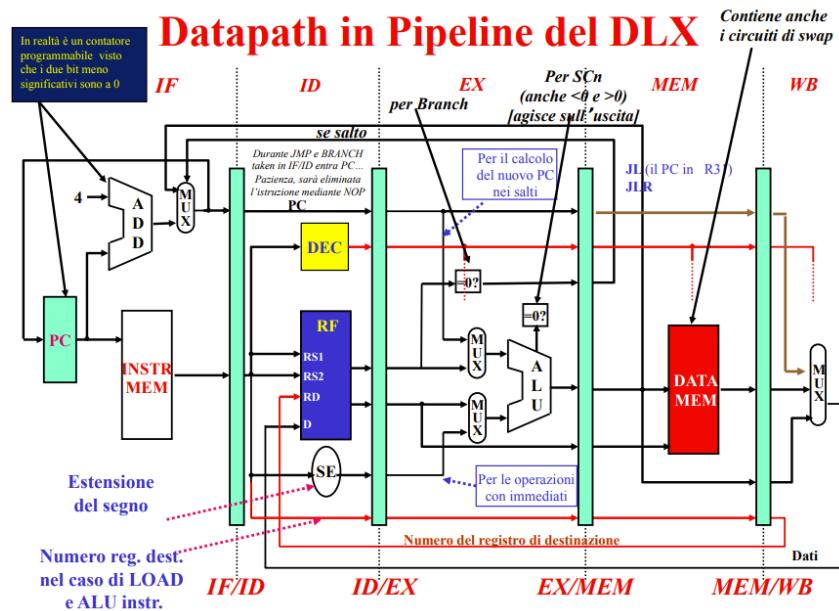


I requisiti per l'implementazione in pipeline sono:

- Ogni stadio deve essere attivo in ogni ciclo di clock;
- È necessario incrementare il PC in IF (invece che in ID), perché in un clock successivo ci si troverà nuovamente nello stadio di fetch;
- È necessario introdurre un ADDER direttamente nello stadio di fetch ( $\text{PC} \leftarrow \text{PC}$  oppure  $\text{PC} \leftarrow \text{PC} + 1$ );
- Sono necessari due MDR (chiamati LMDR e SMDR) per gestire il caso di una LOAD seguita immediatamente da una STORE (WB-MEM sovrapposti → sovrapposizione di due dati in attesa di essere scritti, uno in memoria e l'altro nel RF);
- In ogni ciclo di clock devono poter essere eseguiti 2 accessi alla memoria (IF, MEM) → Instruction Memory (IM) e Data Memory (DM) → **Architettura Harvard**
- Il clock della CPU è determinato dallo stadio più lento: IM e DM devono essere delle memorie cache (on-chip)
- I Pipeline Registers trasportano sia dati sia informazioni di controllo → l'unità di controllo è “*distribuita*” fra gli stadi della pipeline.



## 8.2 Datapath in pipeline del DLX



### Sezione IF - stadio di fetch

In questa fase il DLX incrementa il PC di 4 (tramite l'adder) e gestisce tramite il MUX l'eventuale presenza di salto o meno. Se il salto è preso, viene fornito l'indirizzo al quale saltare (nel PC → il PC è un registro edge triggered quindi l'aggiornamento avviene con un clock di ritardo rispetto all'uscita del PC).

Se il salto è preso, il PC fornisce un indirizzo “sbagliato” perché il DLX sta già facendo il fetch dell’istruzione che in realtà non doveva entrare nella pipeline → l’istruzione va eliminata → in questo caso bisogna cambiare il codice operativo e far transitare istruzioni innocue fino alla fine senza fare nulla → il DLX suppone che il salto non sia preso (quindi fa entrare le istruzioni nella pipeline ma poi devono scorrere perché non servono) → quando nello stadio MEM capiamo che il salto in realtà ha preso si eliminano le istruzioni con un NOP (*no operation*) tramite l’unità di controllo.

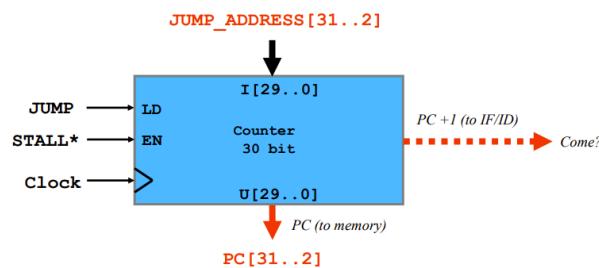
Le branch sono molto problematiche per un dlx pipelined!

Possiamo rimpiazzare lo schema basato su registro e multiplexer con un contatore a 30 bit di indirizzo:

- ▼ Perchè a 30 bit e non 32?

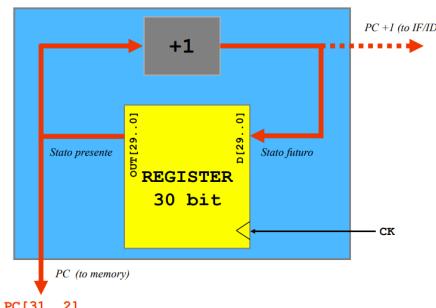
Perchè gli ultimi 2 bit (i meno significativi) sono sempre a 0 (essendo gli indirizzi allineati e le istruzioni lunghe sempre 32 bit) quindi facciamo  $PC = PC + 1$  con un counter a 30 bit.

**PC+1 a 30 bit == PC+4 a 32 bit**



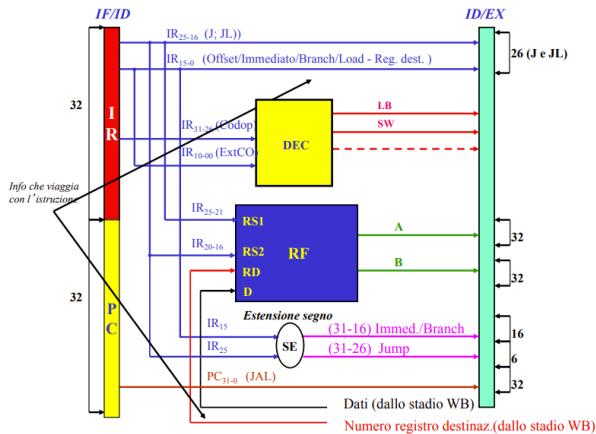
Il segnale JUMP codifica se il DLX deve saltare alla destinazione specificata da  $JUMP\_ADDRESS[31..2]$ . Entrambi i segnali sono inviati dallo stadio MEM; il segnale STALL è generato dall'unità di controllo quando lo stadio di IF deve essere bloccato → ad esempio se il dato non è presente nella cache (circostanze legate ad accesso alla memoria) → fa fermata la pipeline finchè  $ready = 1$ , e quando  $STALL = 1 \rightarrow ENABLE = 0$ .

In circostanze normali  $ENABLE = 1$ , ma il load è prioritario rispetto a enable.



Da questa fase si invia:  $PC + 1$  allo stadio ID per il prossimo fetch, e  $PC$  alla memoria per eventuali necessità.

## Sezione ID



Nello stadio di ID si hanno le informazioni ancora non elaborate → 32 bit di indirizzo (PC+4) → l'istruzione viene inviata alla rete combinatoria che elabora i cambi per decodificare che tipo di istruzione è nello stato ID.

Mentre avviene la codifica, contemporaneamente vengono fatte tante operazioni speculative:

- Vengono estratti 2 registri ipotetici con il metodo dei campi fissi → il register file ha 2 porte di uscita (A e B, RS1 e RS2) → se ci sono dei registri, ogni registro 5 bit+5 bit in posizione fissa ( $2^5 = 32$ ) → forse non serviranno, ma se servono nella fase di execute sono già stati estratti (efficienza!).
  - Si porta dietro PC+4 nel caso in cui l'istruzione sia una jump and link e sia necessario salvare in R31 l'indirizzo di ritorno
  - Viene fatta l'estensione del segno dell'eventuale immediato a 16bit per le load o branch replicando IR[15]
  - Viene fatta l'estensione del segno dell'eventuale immediato a 26bit per le jump replicando IR[25]
  - Viene decodificato il codice operativo, ossia IR[31..26] per capire di che tipo di istruzione si tratta
  - Viene decodificato IR[10..0] nel caso in cui l'istruzione sia di tipo R (ultimi 11 bit dedicati all'extCod → estensione del codice operativo)

- ▼ I bit di Extra Code sono utili per la decodifica solo nelle istruzioni che non usano immediati, negli altri casi vengono oscurati nel dec?

Si, perchè i bit di estensione del codice operativo se è necessario utilizzarli il decoder li usa, altrimenti li filtra tramite una rete combinatoria che (formato R) se quei bit sono necessari li decodifica.

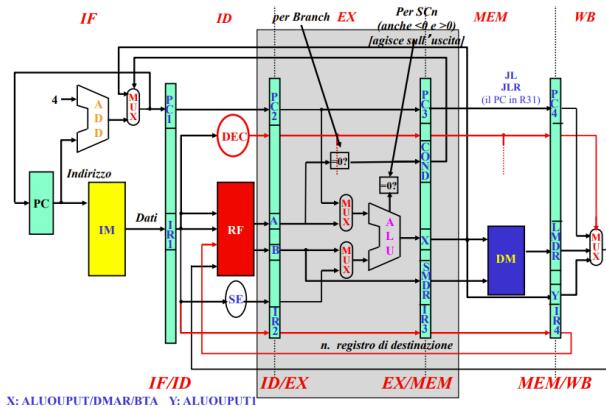
▼ Perchè c'è una doppia estensione?

Perchè noi non sappiamo ancora di che tipo è questa istruzione, e l'immediato può essere a 16 bit o 26 bit.

▼ Nel DLX pipelined A e B sono all'interno del register file?

No, sono all'esterno → il RF viene usato per estrarre i due indirizzi sorgente e dal *write back* per sovrascrivere eventualmente → istruzione in wb modifica un registro in destinazione, dice se si deve scrivere qualcosa e in quale registro.

Sezione EX



In questa fase il DLX conosce il tipo di istruzione che deve eseguire, è lo stadio in cui vengono svolte le operazioni ALU → sono presenti due MUX per selezionare le due opzioni che provengono dalla barriera ID/EX. Il calcolo combinatorio dell'ALU introduce un ritardo.

Durante l'operazione di Set viene verificato se una determinata condizione è vera o falsa e quindi in accordo viene settato il registro d'uscita e l'altra operazione rilevante che viene fatta è quella di andare a testare se un registro è 0, quindi una condizione per un determinato salto, si nota che la condizione viene campionata, quindi se il salto dev'essere preso o meno in tale registro viene campionata la condizione salto o non salto, in modo che quando l'istruzione passa nella fase successiva, ovvero nella fase di MEM, questa informazione viene retroazionata al MUX iniziale (stadio IF) che serve per pilotare la destinazione del salto data dal registro X (calcolata dall'ALU come registro + immediato).

## Sezione MEM

In questa fase i branch tornano al fetch → retroazione;

Le istruzioni che devono accedere alla memoria lo fanno → se il dato è nella cache l'accesso è rapidissimo, l'indirizzo della memoria è stato calcolato nella EX; molte istruzioni in questa fase non fanno nulla.

## Sezione WRITE-BACK

Viene terminata l'istruzione = si va ad aggiornare il registro destinazione per vari motivi → letto qualcosa dalla memoria, era un jump and link quindi va salvato PC in R31, ho usato la ALU, ciò che è stato elaborato durante EX va salvato.

Per scrivere nel registro destinazione servono i 5 bit dell'istruzione codificata a 32 bit presa dalla memoria → l'informazione che voglio scrivere in questo registro sta nel Register File.

## 8.3 Esecuzione in pipeline delle istruzioni

- Esecuzione in pipeline di istruzione ALU

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

<b>IF</b>	$IR \leftarrow M[PC]; PC \leftarrow PC + 4; PC1 \leftarrow PC + 4$
<b>ID</b>	$A \leftarrow RS1; B \leftarrow RS2; PC2 \leftarrow PC1; IR2 \leftarrow IR1$ <b>ID/EX &lt;- Decodifica istruzione;</b>
<b>EX</b>	$X \leftarrow A \text{ op } B$ [PC3 < PC2] oppure $X \leftarrow A \text{ op } (IR2_{15})^{16} \# IR2_{15..0}$
<b>MEM</b>	$Y \leftarrow X$ ("parcheggio" in attesa di WB) [IR4 < IR3] [PC4 < PC3]
<b>WB</b>	$RD \leftarrow Y$

La decodifica attraversa tutti gli stadi

N.B. al passare degli stadi IR perde i bit che non servono più in tutte le istruzioni. Da uno stadio al successivo vengono mantenuti i bit che servono qualunque sia l'istruzione

X: "ALUOUTPUT" (in EX/MEM), Y: "ALUOUTPUTI"

Se l'istruzione è ALU, nella fase di EX viene fatta l'operazione prevista dall'istruzione che a questo punto è decodificata. Nella fase di MEM non fa nulla di utile se non ricampionare ciò che ha prodotto in EX, e nella fase di WB se è presente un'operazione di scrittura del RD viene scritto il risultato e bufferizzato nel RD.

- Esecuzione in pipeline di istruzione MEM

<b>IF</b>	$IR \leftarrow M[PC]; PC \leftarrow PC + 4; PC1 \leftarrow PC + 4$
<b>ID</b>	$A \leftarrow RS1; B \leftarrow RS2; PC2 \leftarrow PC1; IR2 \leftarrow IR1$ <b>ID/EX &lt;- Decodifica istruzione;</b>
<b>EX</b>	$MAR \leftarrow A \text{ op } (IR2_{15})^{16} \# IR2_{15..0}$ SMDR $\leftarrow B$ [IR3 < IR2] [PC3 < PC2]
<b>MEM</b>	$LMDR \leftarrow M[MAR]$ ( <b>LOAD</b> ) oppure $M[MAR] \leftarrow SMDR$ ( <b>STORE</b> ) [PC4 < PC3] [IR4 < IR3]
<b>WB</b>	$RD \leftarrow MDR$ ( <b>LOAD</b> ) [ext. Segno]

La decodifica attraversa tutti gli stadi

Le istruzioni di accesso alla memoria possono essere di lettura o scrittura (LOAD o STORE). Nella fase di EX, viene eseguita la somma tra il registro e l'immediato.

Nella fase di MEM, sono svolte due operazioni possibili: se è una LOAD, viene eseguito l'accesso alla memoria all'indirizzo calcolato nello step precedente; se è una STORE viene scritto il contenuto di un registro sorgente, che era stato opportunamente campionato nello SMDR, all'indirizzo di memoria calcolato nello step precedente.

Nella fase di WB, le operazioni sono inerenti solo per istruzioni di tipo LOAD perché deve essere scritto nel RF il dato letto dalla memoria, quindi memorizzato in RD (portato fino alla fine): 5 bit sono portati fino alla fine per poter scrivere in memoria nel RF.

- Esecuzione in pipeline di istruzione BRANCH

<b>IF</b>	$IR \leftarrow M[PC]$ ; $PC \leftarrow PC + 4$ ; $PC1 \leftarrow PC + 4$
<b>ID</b>	$A \leftarrow RS1$ ; $B \leftarrow RS2$ ; $PC2 \leftarrow PC1$ ; $IR2 \leftarrow IR1$ <b>ID/EX</b> <- Decodifica istruzione;;
<b>EX</b>	$X \leftarrow PC2 \text{ op } (IR_{15})^{16} \# IR_{15..0}$ $Cond \leftarrow A \text{ op } 0$ [ $PC3 \leftarrow PC2$ ] [ $IR3 \leftarrow IR2$ ]
<b>MEM</b>	$\text{if } (Cond) PC \leftarrow X$ [ $PC4 \leftarrow PC3$ ] [ $IR4 \leftarrow IR3$ ]
<b>WB</b>	(NOP)

*X: "BTA (BRANCH TARGET ADDRESS)"      Il test avviene sul valore del registro*

Primi due stadi identici. Una istruzione di BRANCH deve, nello stadio di EX, fare due cose: verificare se il BRANCH deve essere preso e deve essere anche calcolato, come  $PC + \text{immediato}$  (con segno o senza segno), l'indirizzo di destinazione (nel caso che il BRANCH fosse preso) per indicare se il salto dev'essere in avanti o indietro.

In tale implementazione, la condizione sul registro viene campionata perché viene poi gestita nella fase successiva. Se la condizione di salto è vera allora rimpiazza il PC (agisce sul MUX nella fase di fetch) con il valore X calcolato nella fase precedente.

Nello stadio di WB, un'operazione di BRANCH non fa assolutamente niente.

- **Esecuzione in pipeline di istruzione JR**

<b>IF</b>	$IR \leftarrow M[PC]$ ; $PC \leftarrow PC + 4$ ; $PC1 \leftarrow PC + 4$
<b>ID</b>	$A \leftarrow RS1$ ; $B \leftarrow RS2$ ; $PC2 \leftarrow PC1$ ; $IR2 \leftarrow IR1$ <b>ID/EX</b> <- Decodifica istruzione;;
<b>EX</b>	$X \leftarrow A$ [ $IR3 \leftarrow IR2$ ] [ $PC3 \leftarrow PC2$ ]
<b>MEM</b>	$PC \leftarrow X$ [ $IR4 \leftarrow IR3$ ] [ $PC4 \leftarrow PC3$ ]
<b>WB</b>	(NOP)

Le istruzioni di JUMP, nel DLX, sono eseguite come le istruzioni di BRANCH. La differenza significativa è che il JUMP è un salto in cui si sa sin dall'inizio che dovrà essere preso, mentre nel BRANCH c'è un qualcosa che dipende da una condizione. Con tale istruzione è possibile saltare a qualsiasi indirizzo, quindi non è relativo al PC. Il registro sorgente serve per sovrascrivere il PC e questo è fatto nello stadio di MEM. Nello stadio di WB non viene eseguito niente.

- **Esecuzione in pipeline di istruzione JL o JLR**

<b>I</b>	$IR \leftarrow M[PC]$ ; $PC \leftarrow PC + 4$ ; $PC1 \leftarrow PC + 4$
<b>F</b>	
<b>ID</b>	$A \leftarrow RS1$ ; $B \leftarrow RS2$ ; $PC2 \leftarrow P1$ ; $IR2 \leftarrow IR1$ <b>ID/EX</b> <- Decodifica istruzione;;
<b>EX</b>	$PC3 \leftarrow PC2$ $X \leftarrow A \text{ (Se JLR)}$ $X \leftarrow PC2 + (IR_{25})^6 \# IR_{25..0} \text{ (Se JL)}$ [ $IR3 \leftarrow IR2$ ]
<b>MEM</b>	$PC \leftarrow X$ ; $PC4 \leftarrow PC3$ [ $IR4 \leftarrow IR2$ ]
<b>WB</b>	$R31 \leftarrow PC4$ Evidenziati perché in questo caso utilizzati

Decod. in tutti gli stadi  
NB: La scrittura in R31 NON può essere anticipata perché potrebbe sovrapporsi ad altra scrittura di registro

Primi due stadi identici. Calcolo dell'indirizzo, se è una JLR molto semplice, se è invece una JL si aggiorna il PC.

## 8.4 Alee nelle Pipeline

Si verifica una situazione di "alea" (*Hazard*) quando in un determinato ciclo di clock un'istruzione presente in uno stadio della pipeline non può essere eseguita in quel clock. Le varie alee che possono essere presenti, sono (tre tipologie di problemi):

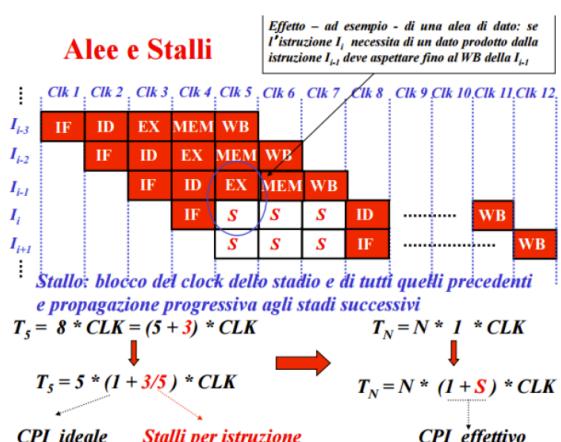
- **Alee strutturali** → una risorsa è condivisa fra due stadi della pipeline: le istruzioni che si trovano contemporaneamente in tali stadi non possono essere eseguite simultaneamente → esempio due stadi che vogliono accedere alla stessa risorsa di memoria e quindi bisogna decidere quale risorsa è primaria rispetto all'altra; un altro possibile esempio è duplicare l'ALU, una nel fetch per incrementare il PC, l'altra nella fase di EX per eseguire le varie tipologie di operazioni.
- **Alee di dato** → dovute a dipendenze fra le istruzioni → ci possono essere istruzioni che leggono dei dati dopo che sono stati modificati da delle istruzioni che sono più avanti nella pipeline, ovvero che vengono prima nel codice → tale problema viene denominato ***Read After Write (RAW)***.
- **Alee di controllo** → le istruzioni che seguono un BRANCH dipendono dal loro risultato (taken/not taken) quindi c'è il rischio di far finire altrove nella pipeline istruzioni che non dovrebbero entrare, ed è presente il problema di capire se il BRANCH è preso oppure no.

Se si presentano problemi di queste tipologie, è bene bloccare l'istruzione che non può essere eseguita ("**stallo della pipeline**"), insieme a tutte quelle che la seguono, mentre le istruzioni che la precedono avanzano normalmente (così da rimuovere la causa dell'alea).

Se per esempio si ha un accesso alla memoria che richiede 100 cicli di clock per andare a prelevare il dato nella fase di MEM, tutte le istruzioni che precedono MEM devono essere messe in stall.

Se si inseriscono degli stalli sulla pipeline, si avrà un effetto negativo sulla CPI perché aumenta di 1 il fattore proporzionale al numero di stalli che si va ad inserire nella pipeline.

Nello schema la prima istruzione ( $I_{i-3}$ ) è una determinata istruzione di qualsiasi tipo; entra nella pipeline  $I_{i-2}$ , poi  $I_{i-1}$  ed infine  $I_i$ . In tal caso, questa istruzione ha qualche problema ad avanzare perchē, ad esempio, potrebbe avere la necessità di utilizzare un dato che sarà modificato solo quando l'istruzione che la precede avrà aggiornato il RF → in questo caso l'istruzione  $I_{i-1}$  viene messa in stall per un numero di clock sufficienti a far sì che avvenga la scrittura nel RF.



### Stalli nel salto

Quando si è nei salti, quindi quando bisogna gestire un'alea di controllo, succede che nel proprio sistema entrano delle istruzioni errate.

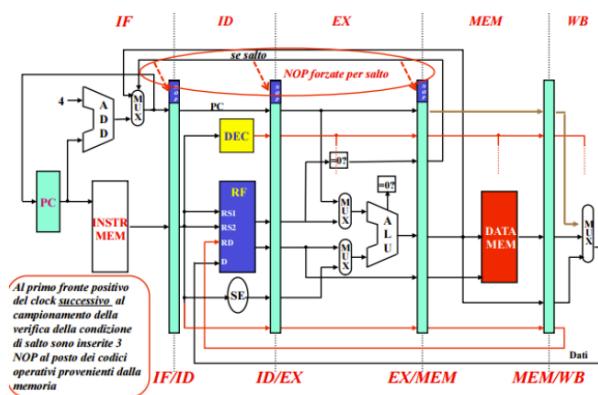
Primo problema che ci si pone è: se entrano delle istruzioni errate, significa una perdita di tempo perchē non entrano istruzioni utili, in modo da avere un CPI pari a 1 e se entrano istruzioni errate questo aumenta

e il numero di clock per istruzione diventa maggiore di 1 e le prestazioni peggiorano. Il secondo problema, nonché il peggiore è: se entrano delle istruzioni nella pipeline e queste non sono corrette, perché ad esempio c'è stato un salto, allora bisogna far sì che quelle istruzioni non facciano nessun danno e creare ulteriori problemi.

Una istruzione può creare dei danni quando, per esempio, modifica/aggiorna il contenuto di un registro (RD) il che avviene nella fase di WB; oppure che venga fatta una scrittura ad un indirizzo di memoria che in realtà non deve essere sovrascritto.

Si ricorda che il PC sa se un salto dovrà essere preso nello stadio di EX. Se sono entrate delle istruzioni all'interno della pipeline e queste sono errate, incluso il fetch di un'istruzione all'indirizzo sbagliato (in caso di BRANCH ovviamente), comunque, se ciò accade, l'unità di controllo può andare a disintegrale le istruzioni che sono in quel momento errate e presenti nella pipeline, e per fare questo è sufficiente andare a cambiare il codice operativo dell'istruzione, facendole diventare delle istruzioni innocue e nel DLX esiste il codice operativo per tale tipologia di istruzione, ovvero l'istruzione

**No OPeration (NOP)** che non fa nulla, passa nella pipeline senza fare nulla di problematico.



Un problema è quello di spostare il rilevamento del salto il prima possibile perché migliorerebbe le prestazioni. Se si vuole ridurre il numero di stalli e/o istruzioni errate (nel caso precedente erano presenti tre stalli) si va ad anticipare il rilevamento del salto dallo stadio di MEM allo stadio di EX, si ha ridotto di 1 le istruzioni errate nella pipeline o le NOP che si vuole mettere. Tale strategia però, è molto aggressiva e può creare problemi al timing.

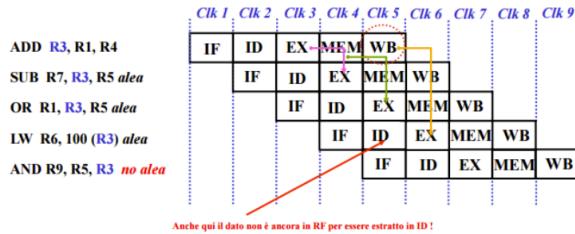
## Forwarding

Il forwarding consiste nel far avanzare le istruzioni all'interno della pipeline anche se hanno prelevato dal RF il valore non aggiornato del RS che è stato modificato.

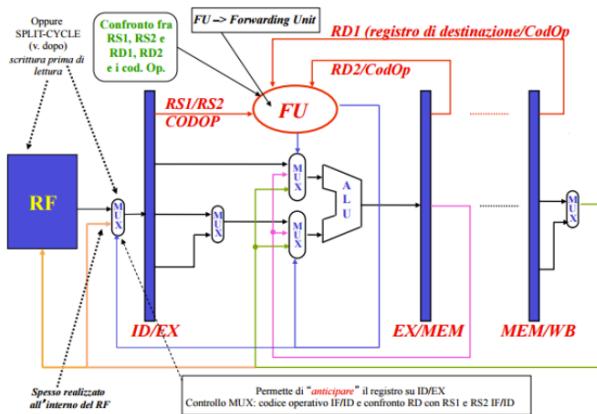
Come da esempio, prelevano e leggono R3 vecchio e nel momento in cui utilizzano quel valore, in particolare nello stadio di EX, una rete logica ad hoc (la **Forwarding Unit** per l'appunto) fa sì che il dato corretto arrivi nello stadio di EX dalla posizione in cui si trova. Quindi, in questo caso, l'istruzione di sottrazione, quando si troverà nello stadio di EX, la FU farà sì che R3 corretto (la sub entra nello stadio di EX con un valore di R3 estratto dal RF che non è ancora stato aggiornato all'interno del RF, perché l'istruzione precedente (ADD) non ha ancora terminato il WB), che a questo punto di trova nello stadio di MEM, venga in qualche modo inviato (retroazionato) nello stadio di EX, in modo che il valore vecchio di R3 sia rimpiazzato con il suo valore aggiornato

La stessa cosa, viene fatta con l'operazione successiva (OR), che quando si troverà nello stadio di EX, riceverà il dato aggiornato dallo stadio di WB; quindi, mentre la prima istruzione è nel WB e si sta portando

dietro il valore di R3 aggiornato, e al termine di quel ciclo, invierà al RF. L'istruzione di OR, può ricevere quel dato aggiornato dall'istruzione di ADD che è in WB.



Il forwarding consente quindi di eliminare quasi tutte le alee di tipo RAW della pipeline del DLX senza creare stalli (si ricorda che nel DLX si alterano i registri solo in WB).



#### ▼ A che serve il MUX prima di ID-EX?

Serve quando abbiamo un'istruzione in write-back che modifica il registro destinazione e c'è un'istruzione in ID che legge lo stesso registro → per evitare di leggere un dato vecchio, lo bypassa con il dato aggiornato che sta per essere scritto nel register file.