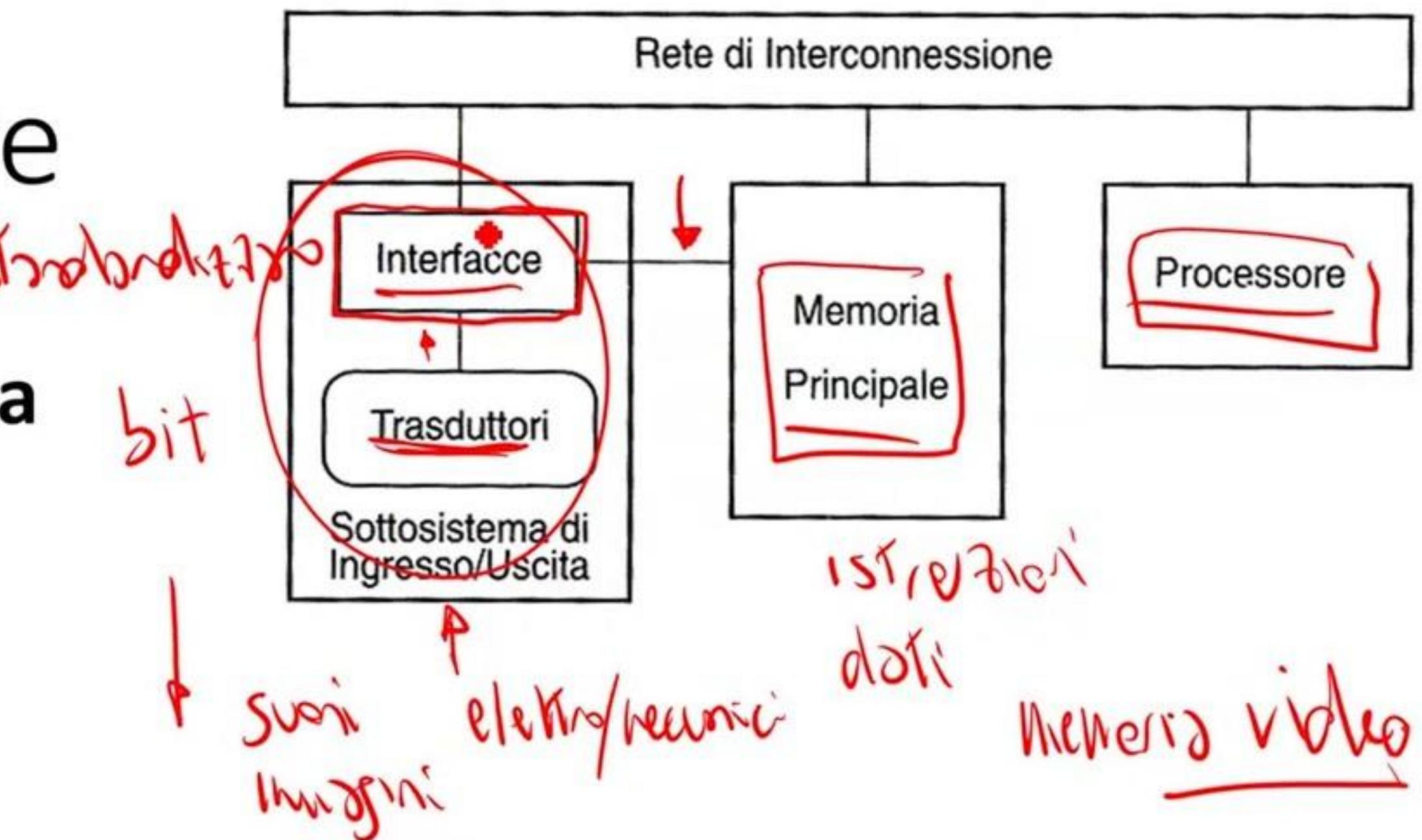


Materiale didattico

- Paolo Corsini, "Dalle porte AND, OR, NOT al Sistema calcolatore", edizioni ETS
- Struttura del calcolatore - appunti, versione 24/11/2020

Struttura del calcolatore

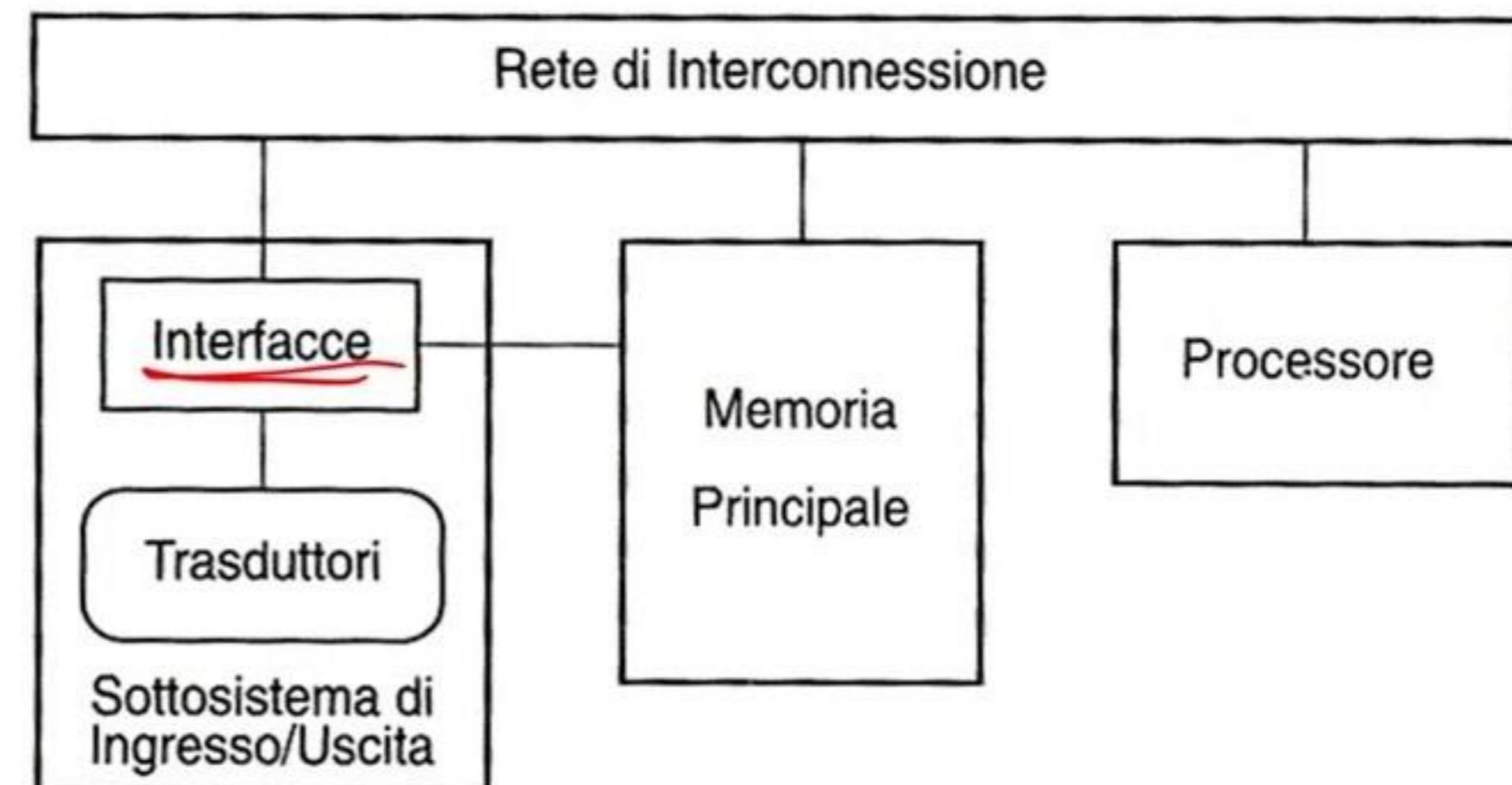
- Descriveremo in Verilog un **sistema completo**
 - Processore
 - Memoria
 - Interfacce
 - Dispositivi di I/O
- Mettendo insieme tutto quello che abbiamo già visto



Struttura del calcolatore

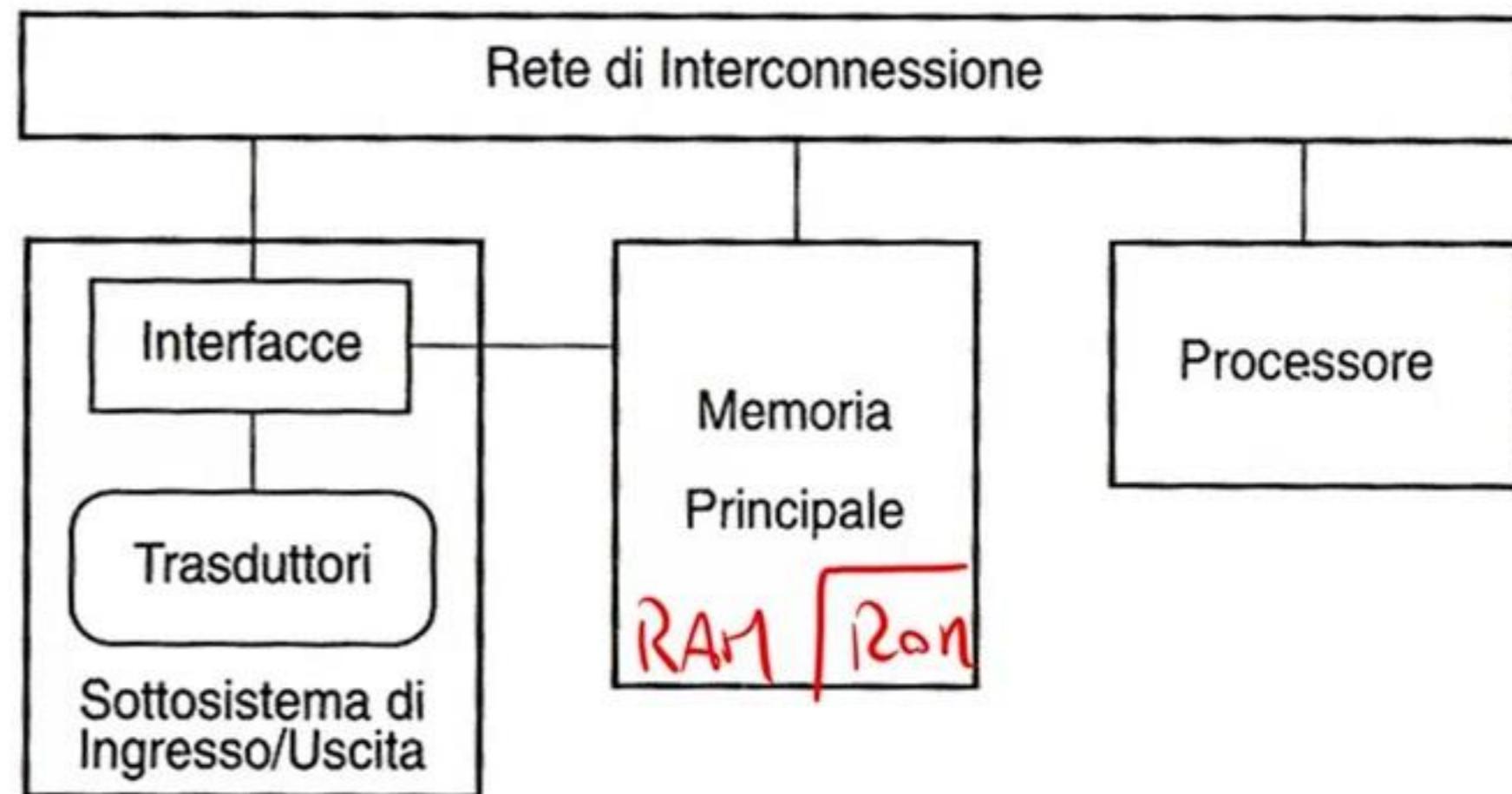


- **Sottosistema di ingresso/uscita (I/O)**
- gestisce la codifica delle informazioni ed il loro scambio con il mondo esterno.
- All'interno di questo sottosistema distinguiamo **interfacce e dispositivi**.
 - Dispositivi: effettuano la vera e propria codifica.
 - Interfacce: gestiscono i vari dispositivi, cioè fanno in modo che il colloquio tra questi ed il processore possa avvenire con modalità standard.
- Interfacce contengono un numero (piccolo) di registri di interfaccia, che il processore può leggere o scrivere (o, più raramente, leggere e scrivere)



Struttura del calcolatore

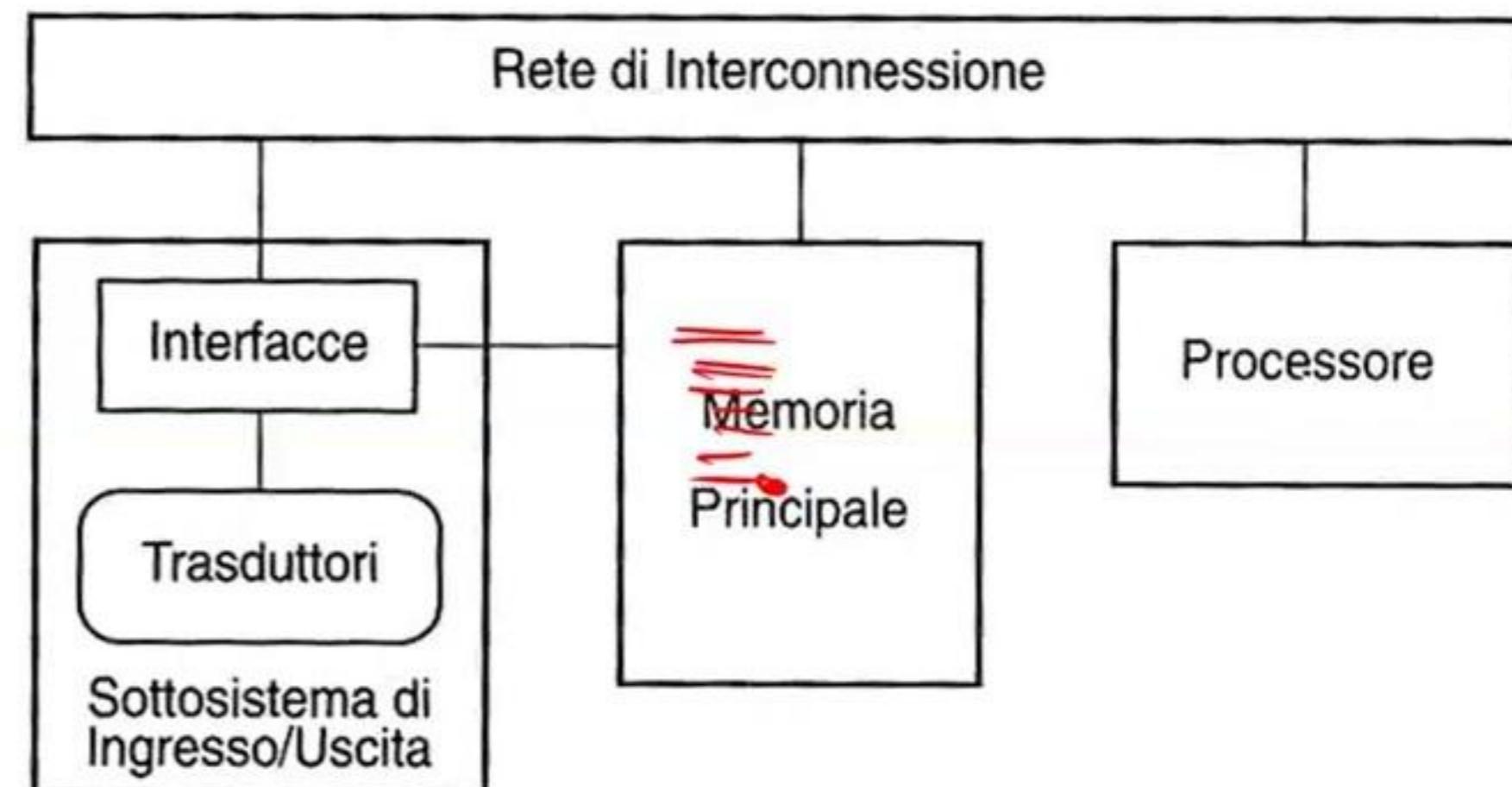
- **Memoria principale**
- contiene in ogni istante **le istruzioni e i dati** che il processore elabora
 - alcuni dati possono risiedere nel sottosistema di I/O.
- Una parte di questa memoria è adibita a **memoria video**, e contiene una replica dell'immagine che viene mostrata sullo schermo.
- Per questo motivo c'è un collegamento diretto tra la memoria ed un'interfaccia (video) nella figura.



Struttura del calcolatore

HLT

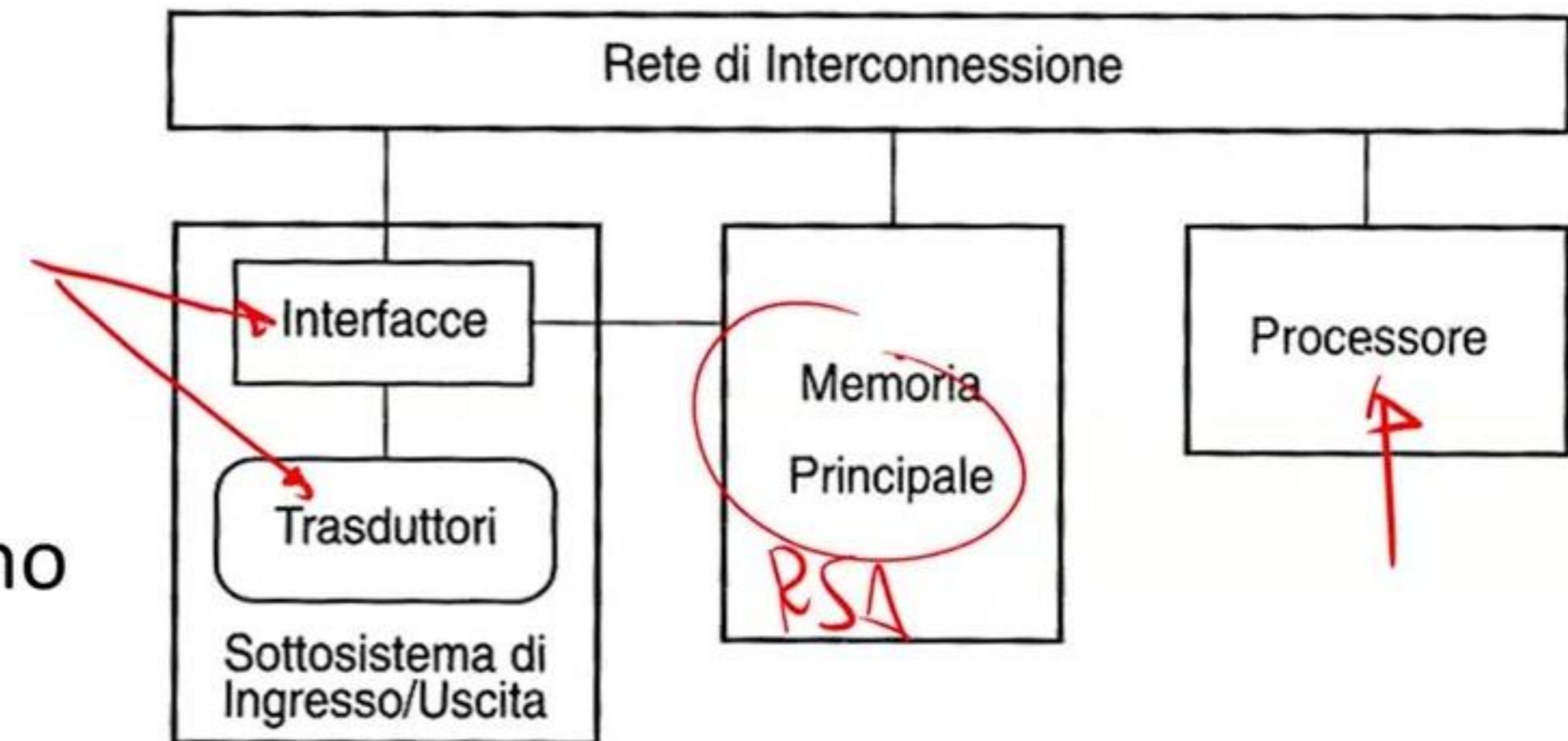
- Il **processore** ciclicamente
 - preleva un'istruzione dalla memoria (**fetch o chiamata**)
 - la esegue
- Le istruzioni si trovano, di norma, **in sequenza** nella memoria principale (**istruzioni operative**)
 - Le **istruzioni di controllo** alterano il flusso sequenziale
 - il prelievo di istruzioni riparte da una locazione diversa
- Deve partire al reset in modo consistente
 - Deve iniziare a leggere la memoria da una locazione ben precisa
 - dove deve essere scritto del codice, in maniera indeleibile
- Ciò si realizza facendo in modo che:
 - Al reset, il processore abbia inizializzato l'**instruction pointer** (ed altro) ad un valore opportuno
 - Parte della memoria sia implementata in EPROM, e contenga un programma **bootstrap**



- **sEP8 (8-bit simple Educational Processor)**
 - elabora dati a 8 bit
 - lavora in aritmetica in base 2 rappresentando gli interi in C2
 - indirizza una memoria di **16Mbyte** 2^4 2^{20} 2²¹ kili
 - Inventato per il corso di RL

Struttura del calcolatore

- Il calcolatore è una serie di RSS
 - Con l'eccezione dei dispositivi, che contengono parti elettromeccaniche (non oggetto della nostra trattazione)
 - La memoria è (in parte) una RSA
- Tutti i moduli RSS avranno una piedinatura per il reset
 - consente al calcolatore di partire in modo consistente
- Descriveremo il processore come RSS
 - Sintetizzabile come PO/PC



- Dobbiamo prima **darne una specifica**
 - come facciamo con qualunque RSS.
 - Con quali altre reti si interfaccia, e come;
 - quale è il suo comportamento osservabile (il suo linguaggio macchina)

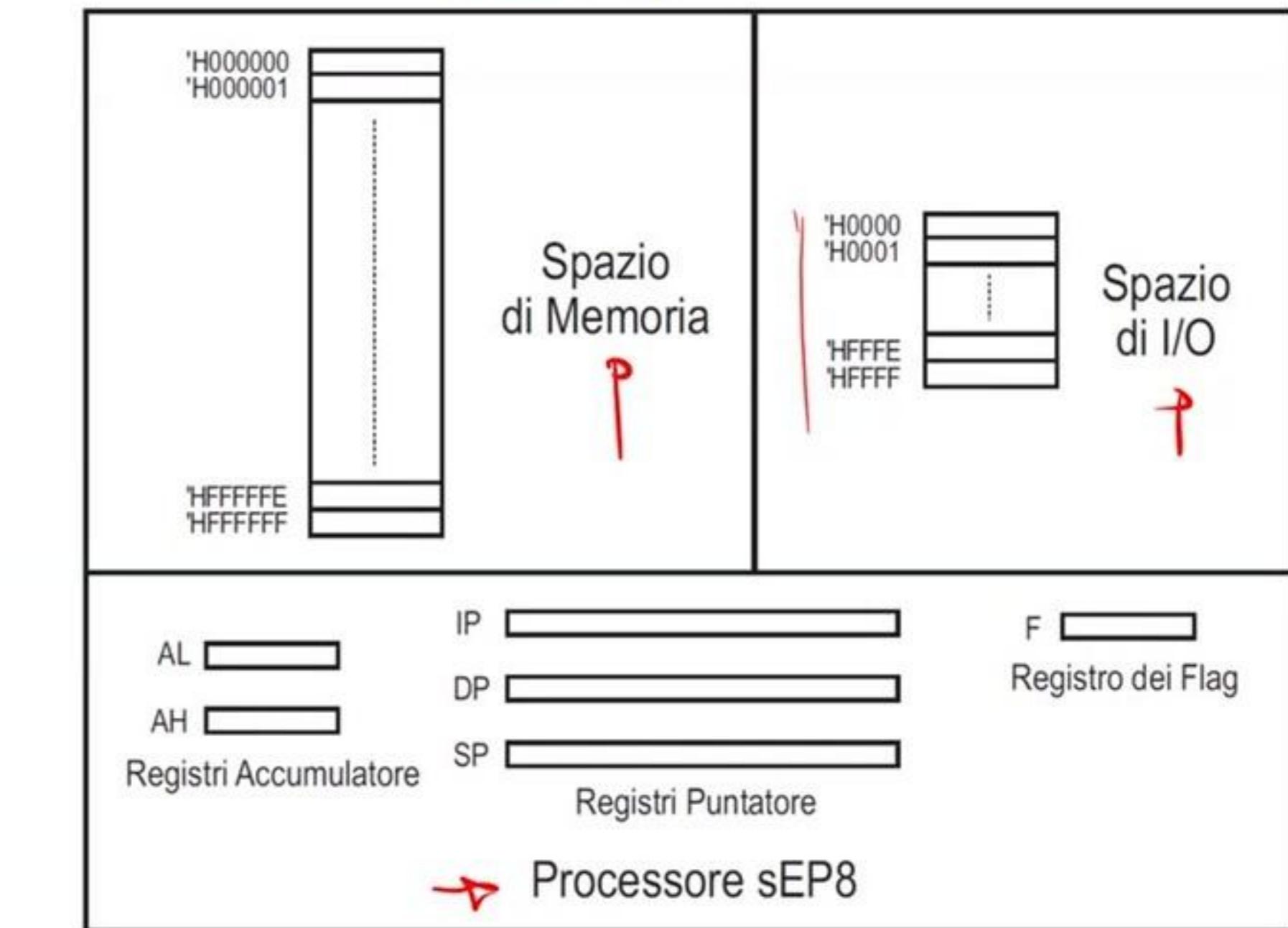
Il calcolatore visto dal programmatore

- **Memoria:** spazio lineare di 2^{24} locazioni da un byte (16 Mbyte).

- Indirizzi a 24 bit
- Lettura e scrittura di un byte

- **Spazio di I/O:** spazio lineare di $2^{16}=64K$ locazioni o porte da un byte

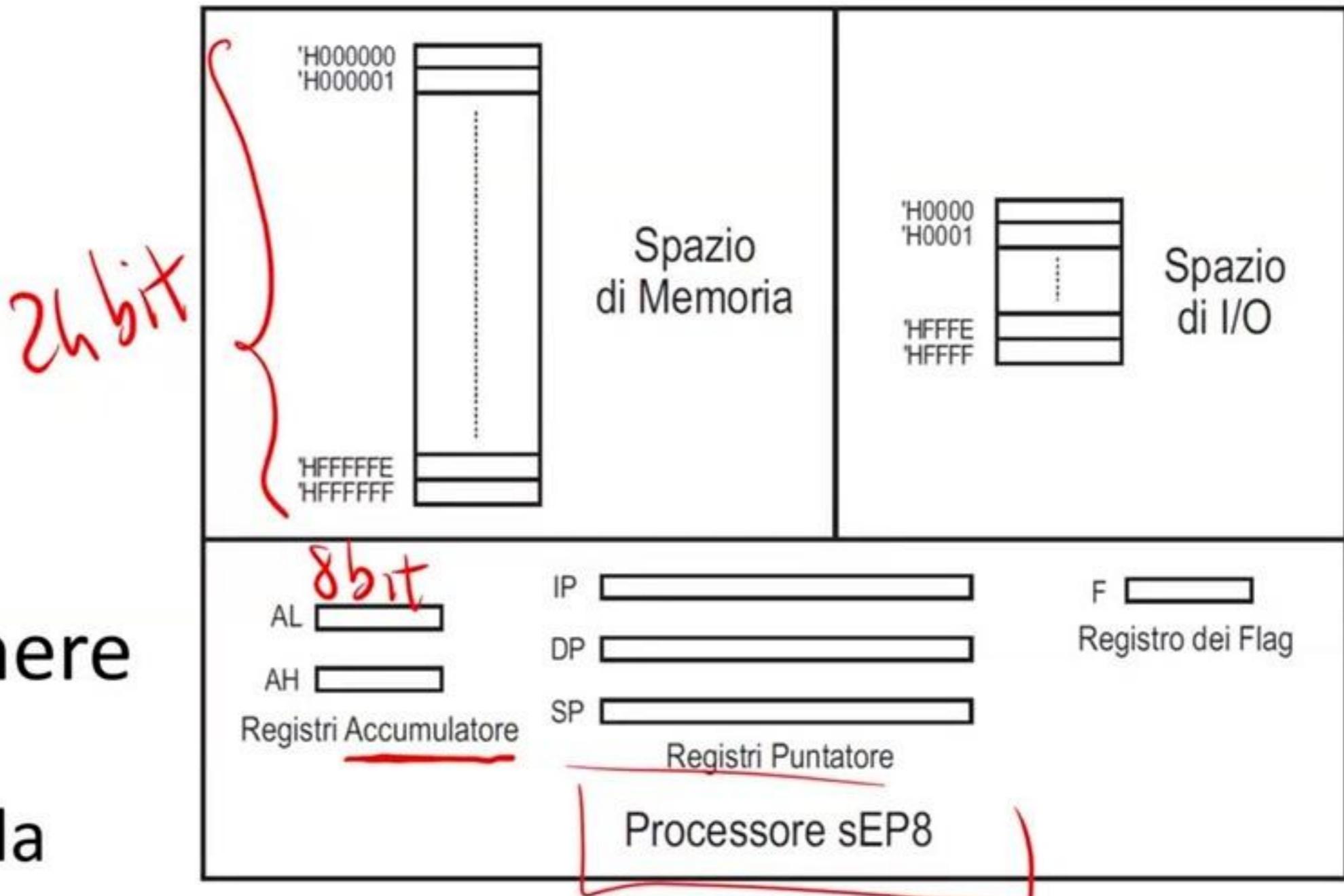
- Indirizzi a 16 bit
- Lettura e scrittura di un byte
- insieme dei **registri di interfaccia** che il processore può teoricamente indirizzare
- Pochi sono implementati fisicamente



10₁₀ ~61e³

Il calcolatore visto dal programmatore

- Il processore sEP8 ha **tre tipi di registri**
- Registri accumulatore: AH ed AL, 8 bit
 - contengono operandi di elaborazioni
- Registro dei flag: 8 bit
 - 4 bit significativi: CF (0), ZF (1), SF (2), OF (3)
- Registri puntatore: a 24 bit (devono contenere indirizzi di memoria)
 - IP (instruction pointer): contiene l'indirizzo della **prossima istruzione** da eseguire
 - SP (stack pointer): contiene l'indirizzo del **top della pila**
 - DP (data pointer): contiene l'indirizzo di operandi, a seconda della modalità di indirizzamento (che vedremo più avanti)



- Al reset:
 - $F \leftarrow 'H00 ;$
 - $IP \leftarrow 'HFF0000 ;$

NaN NaN, NL

Assembler

ssubroutine

LM

01C - — C10

Linguaggio macchina del processore sEP8

- Il linguaggio macchina di un **processore** è il suo comportamento osservabile.
- Noi esseri umani programmiamo però in **Assembler**, e non in **linguaggio macchina**.
- Conviene iniziare la descrizione del comportamento del processore sEP8 spiegando come un **programmatore Assembler** dovrebbe scrivere le sue istruzioni
 - successivamente discuteremo come queste si possano codificare in linguaggio macchina

Assembler del sEP8

- Per un programmatore **Assembler**, il formato delle istruzioni del processore sEP8 sarà il seguente:

~~opcode~~ ~~source~~, ~~destination~~

~~EXABCDE~~

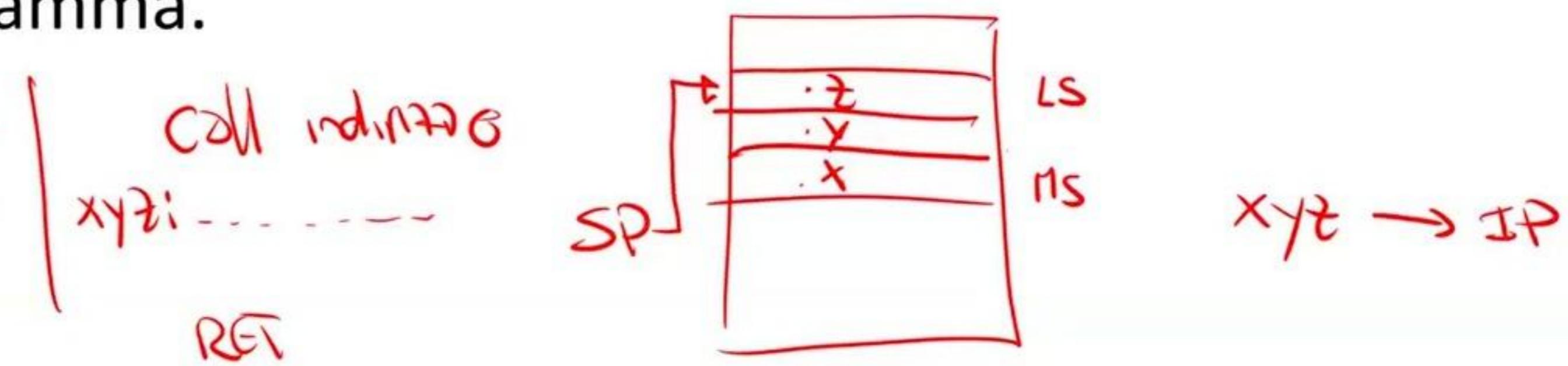
- OPCODE è il codice operativo dell'istruzione,
 - source e destination individuano i due operandi sorgente e destinatario.
 - In alcune istruzioni il campo source può mancare
 - In due istruzioni (NOP e HLT) mancano entrambi
-
- Le **modalità di indirizzamento** degli operandi sono simili a quelle dei processori Intel
 - Ripassiamole

Modalità di indirizzamento – ist. operative

- **Di registro:** uno o entrambi gli operandi sono nomi di registro
 - OPCODE AL, AH
 - OPCODE DP
- **Indiretto:** l'operando sorgente è specificato **nell'istruzione** come costante
 - OPCODE \$0x10, AL
- **Di memoria:** valido per il *sorgente* o per il *destinatario* (mai entrambi). Sono possibili due indirizzamenti di memoria
 - **diretto:** l'indirizzo è specificato direttamente nell'istruzione
OPCODE 0x1010FA, AL
 - **indiretto:** la locazione di memoria ha indirizzo contenuto nel registro DP
OPCODE (DP), AL
- **Delle porte di I/O:** le porte di I/O si indirizzano in modo **diretto**, specificando l'offset della porta dentro l'istruzione stessa
 - IN 0x1010, AL
 - OUT AL, 0x9F10

Modalità di indirizzamento – ist. controllo

- Salti, condizionati e non, chiamate di sottoprogramma ed istruzioni di ritorno da sottoprogramma.
 - JMP indirizzo
 - Jcon indirizzo
 - CALL indirizzo
 - RET
- Le prime tre devono specificare l'indirizzo di salto, che va a **sostituire il contenuto di IP** (3 byte)
- CALL e RET interagiscono con la **pila**:
 - CALL salva in pila il contenuto di IP (3 byte), cioè l'indirizzo della istruzione successiva alla CALL medesima (indirizzo di ritorno);
 - la RET preleva dalla pila un indirizzo (3 byte), e lo sostituisce ad IP.



Set di istruzioni del processore sEP8

- MOV source, destination

- **Formati**

- Registro Flag: inalterato

MOV AL, AH

MOV AH, AL

MOV \$operando, A^{8bit}~~*~~

(* = L, H)

MOV \$operando, DP^{24bit}

MOV \$operando, SP

MOV indirizzo, A^{*}

MOV indirizzo, DP

MOV DP, indirizzo

→ MOV (DP), A^{*}

MOV A^{*}, indirizzo

→ MOV A^{*}, (DP)

Set di istruzioni del processore sEP8 (cont.)

IN offset, AL

OUT AL, offset

PUSH AL

POP AL

PUSH AH

POP AH

PUSH DP

POP DP

- Registro Flag: inalterato

Set di istruzioni del processore sEP8 (cont.)

INC DP

INL DIV

- ~~ADD~~ source, destination
- ~~SUB~~ source, destination
- ~~CMP~~ source, destination

Flag CF, ZF, SF, OF
IN

Formati

XYZ \$operando, A*

(* = L, H)

XYZ indirizzo, A*

XYZ (DP), A*

Set di istruzioni del processore sEP8 (cont.)

SHL A*
SHR A*

Flag CF, ZF, SF, OF=0

NOT A*

AND source, destination
OR source, destination

Flag ZF, SF, CF=OF=0

Formati

XYZ \$operando, A*
XYZ indirizzo, A*
XYZ (DP), A*

Set di istruzioni del processore sEP8 (cont.)

JMP indirizzo

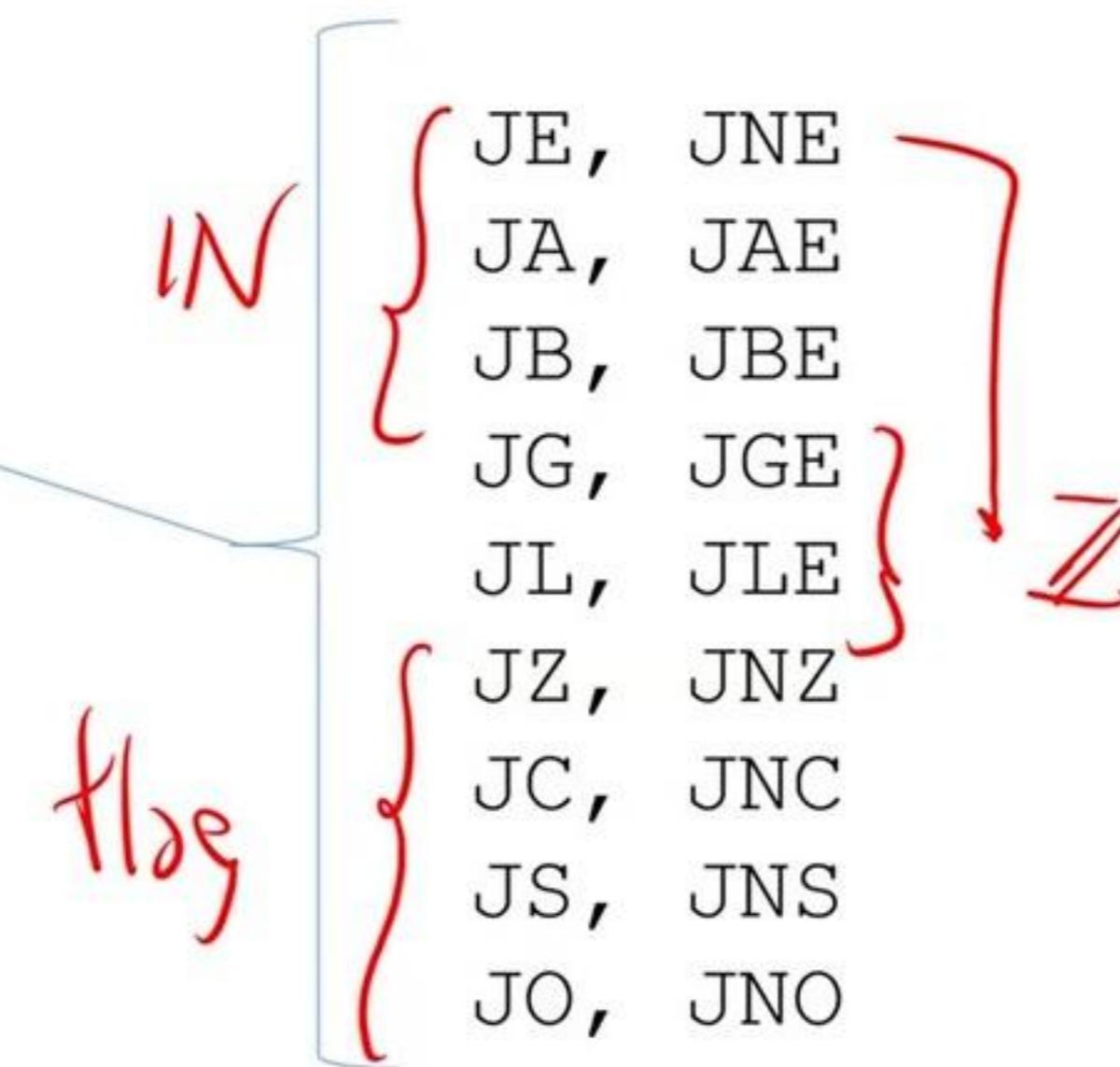
~~Jcon~~ indirizzo

CALL indirizzo

~~RET~~

NOP

HLT



Dall'Assembler al linguaggio macchina

- Devo tradurre un'istruzione Assembler:

OPCODE source, destination

- In una sequenza di zeri e uni con una certa **sintassi**.
- Questa **sintassi** costituisce il **linguaggio macchina** di quel processore, e deve essere **compatta** e facile da interpretare
 - «facile» per un ~~processore~~ processore, non necessariamente per noi.



Dall'Assembler al linguaggio macchina (cont.)

- Per gli esseri umani è dirimente il “tipo” di operazione
 - MOV: copia di informazione, qualunque siano gli operandi
 - Assembler: specifica prima il “tipo” dell’operazione, e successivamente gli operandi
- Per un processore, invece, è dirimente dove si trovino gli operandi.
- `MOV AH, AL` il processore gli operandi li ha già, perché sono contenuti nei registri
- ~~`MOV $0x10, AL`~~ il processore deve **leggere in memoria** l’operando sorgente, che è contenuto nell’istruzione medesima (l’istruzione si trova in memoria, ovviamente).
- `MOV (DP), AL`

Fetch
rite

Dall'Assembler al linguaggio macchina (cont.)

- **Fase di fetch**

- Il processore si deve procurare le informazioni necessarie ad eseguire l'istruzione
 - Il codice operativo
 - Gli operandi
-
- Per trovare gli operandi può essere necessario eseguire cicli di lettura in memoria
 - O nello spazio di I/O (IN, OUT)

MOV AH, AL
MOV \$0x10, AL
MOV (DP), AL

- Fasi di **fetch** differenti (perché operandi differenti)
- Fasi di **esecuzione identiche:**
AL<=operando;

Formato delle istruzioni macchina

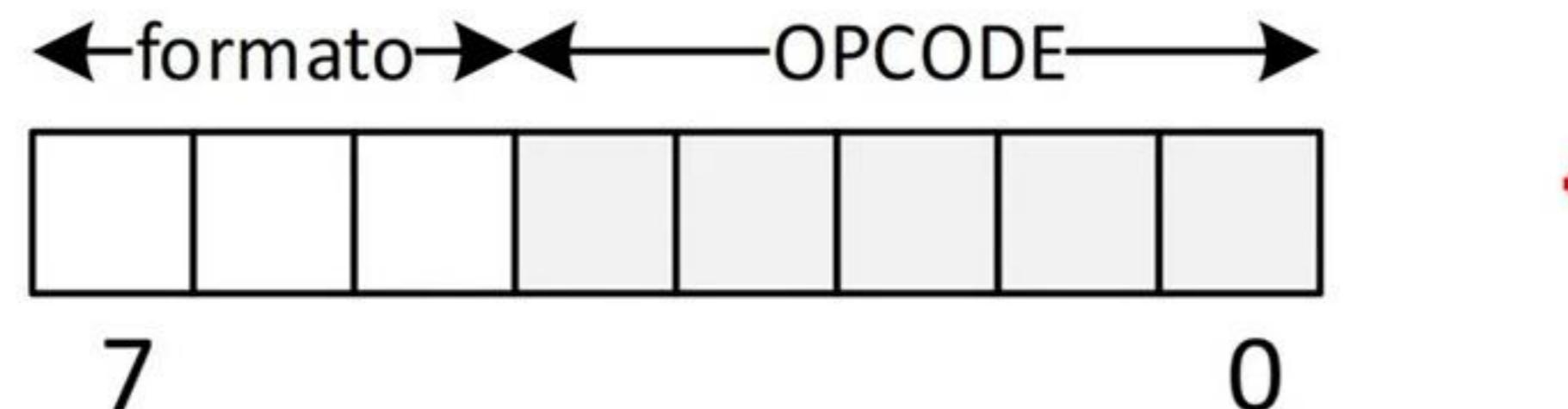
- Tutte le istruzioni del tipo:

XYZ \$operando, AL (MOV, ADD, SUB, AND, OR)

- Devono procurarsi l'operando sorgente nella stessa maniera
 - Leggendo **un byte** in memoria
- La loro **fase di fetch** può essere messa a comune
- Mentre **la fase di esecuzione** sarà differente
 - Dipende dall'operazione XYZ
- Conviene dividere le istruzioni in un certo numero di **formati**
 - Corrispondenti ai diversi modi di recuperare gli operandi

Formato delle istruzioni macchina

- Il primo byte di un'istruzione codifica contemporaneamente
 - Il formato dell'istruzione (primi tre bit -> 8 formati)
 - Il codice operativo dell'istruzione (ultimi 5 bit -> 32 opcode possibili per formato)



- Passiamo in rassegna i vari formati

Formato F0 (000)

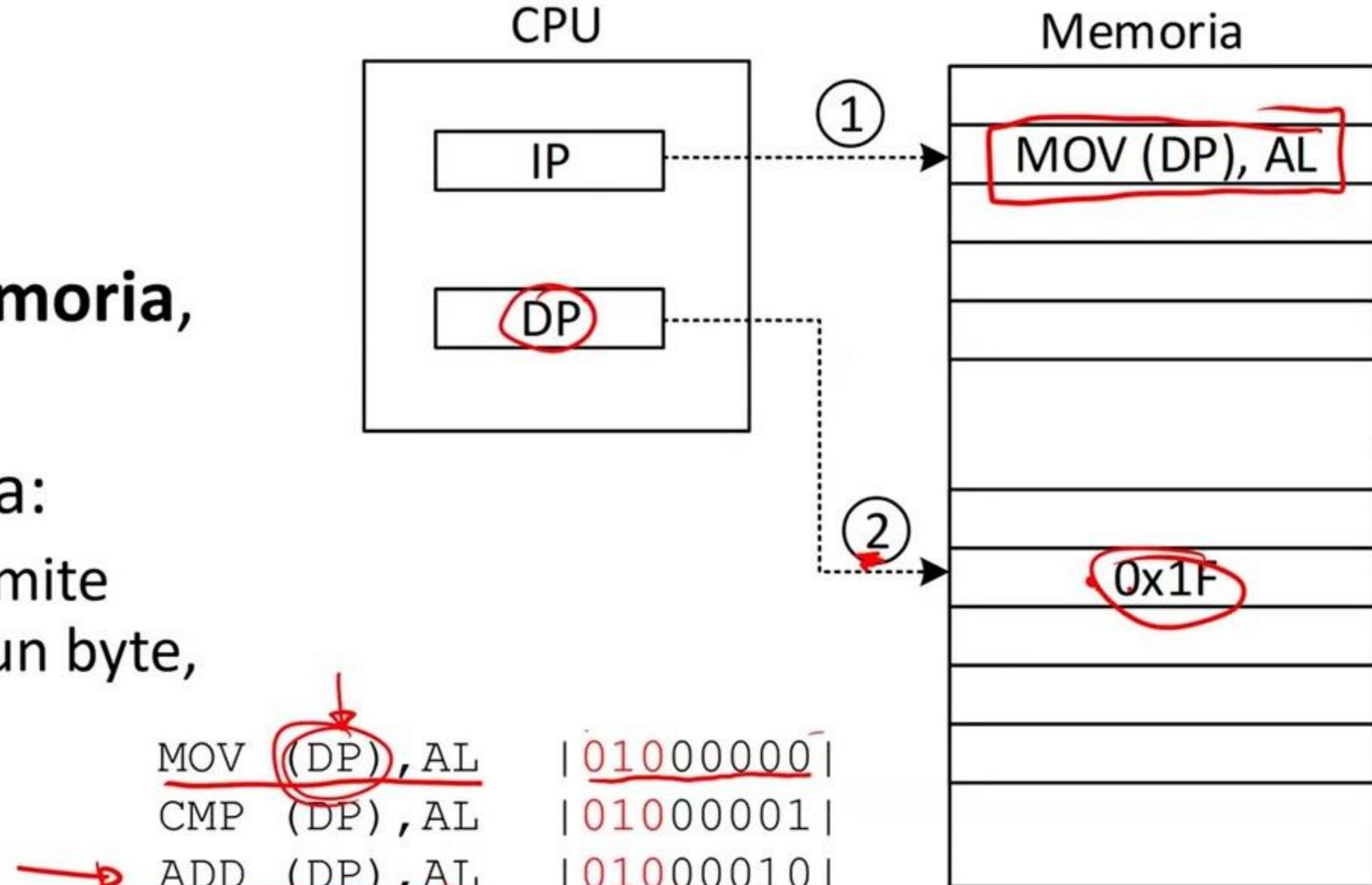
- **Formato F0 (000)**: istruzioni per le quali il processore non deve compiere **nessuna azione** per procurarsi gli operandi, in quanto:
 - gli operandi sono registri, oppure
 - le istruzioni non hanno operandi (HLT, NOP, RET)
- Istruzioni costituite **da un unico byte**.
- La fase di fetch consiste nella lettura di quest'unico byte, **all'indirizzo puntato da IP**

Codifica mnemonica	Codifica macchina
HLT	000 <u>00000</u>
NOP	000 <u>00001</u>
MOV AL, AH	<u>0000010</u>
MOV AH, AL	00000011
INC DP	00000100
SHL AL	00000101
SHR AL	00000110
NOT AL	00000111
SHL AH	<u>00001000</u>
SHR AH	000 <u>01001</u>
NOT AH	00001010
PUSH AL	00001011
POP AL	00001100
PUSH AH	00001101
POP AH	00001110
PUSH DP	00001111
POP DP	00010000
RET	00010001

Formato F2 (010)

- l'operando **sorgente** si trova in **memoria**, indirizzato tramite **DP**
- Tutta l'istruzione sta su un byte, ma:
 - l'operando sorgente va prelevato tramite una **ulteriore lettura in memoria** di un byte, all'indirizzo contenuto in DP
 - Va fatto durante la fase di fetch

Sorgente
→ dentro il processore



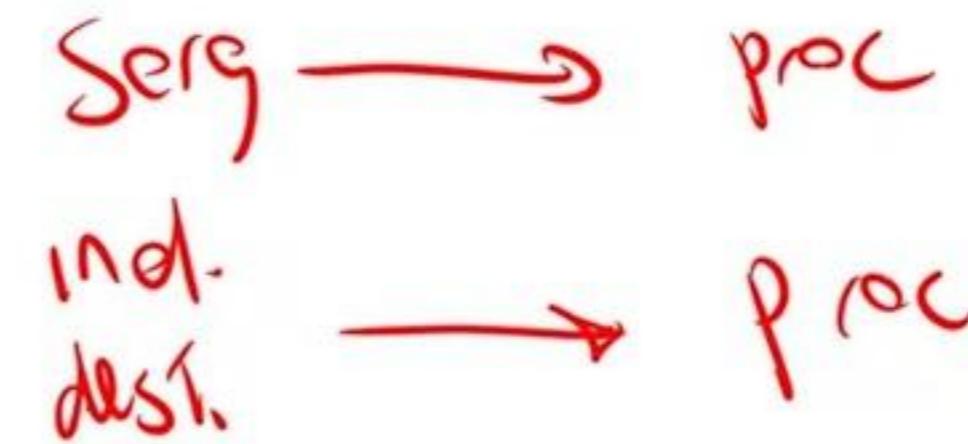
TEIP

AL <= AL + TEMP;

Formato F3 (011)

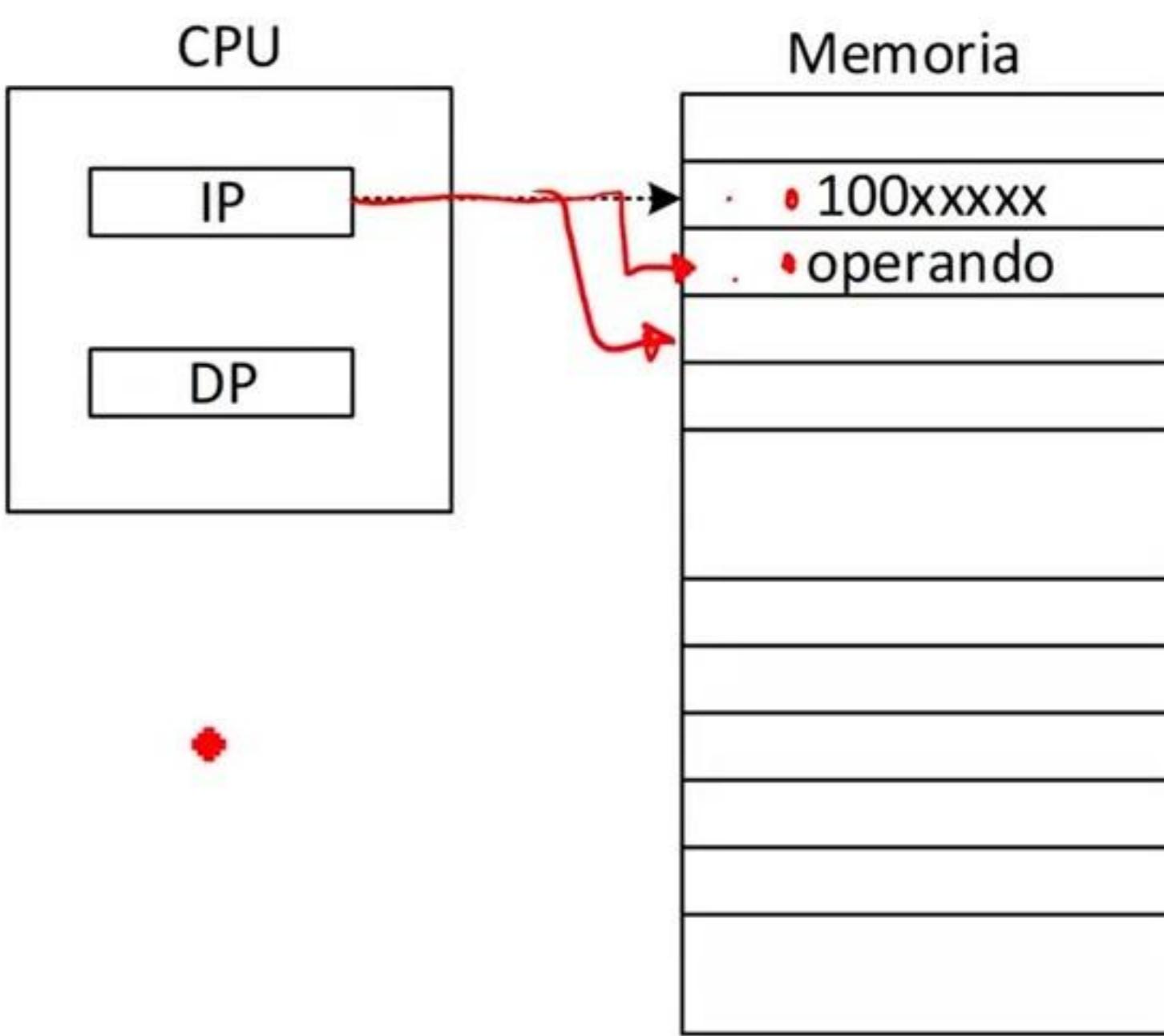
- Istruzioni in cui l'operando **destinatario** è indirizzato usando DP.
- Anche in questo caso tutta l'informazione relativa all'istruzione può stare su un singolo byte
- Per l'operando **destinatario**, è sufficiente avere l'indirizzo
 - La copia si fa in fase di esecuzione

MOV AL, (DP) |01100000|
MOV AH, (DP) |01100001|



Formato F4 (100)

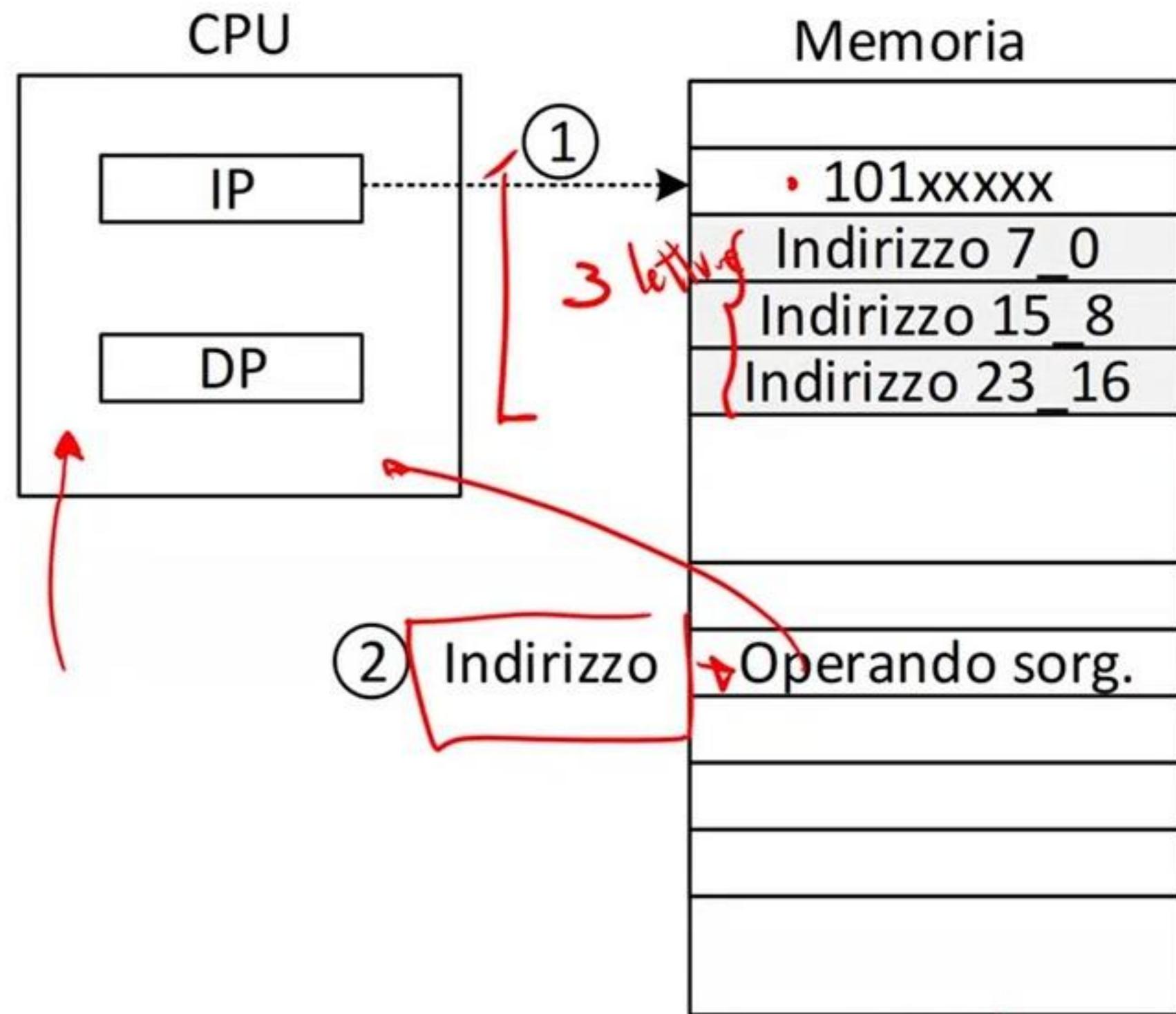
- Istruzioni in cui l'operando sorgente è indirizzato in modo immediato, e sta su 8 bit.
- L'istruzione è lunga **due byte**
 - il secondo contiene l'operando sorgente
- Pertanto, la fase di fetch dovrà leggere due byte in memoria, ad indirizzi consecutivi puntati dal **registro IP**.



MOV \$operando, AL | 10000000 | operando |
CMP \$operando, AL | 10000001 | operando |
ADD \$operando, AL | 10000010 | operando |
SUB \$operando, AL | 10000011 | operando |
AND \$operando, AL | 10000100 | operando |
OR \$operando, AL | 10000101 | operando |
MOV \$operando, AH | 10000110 | operando |
CMP \$operando, AH | 10000111 | operando |
ADD \$operando, AH | 10001000 | operando |
[...]

Formato F5 (101)

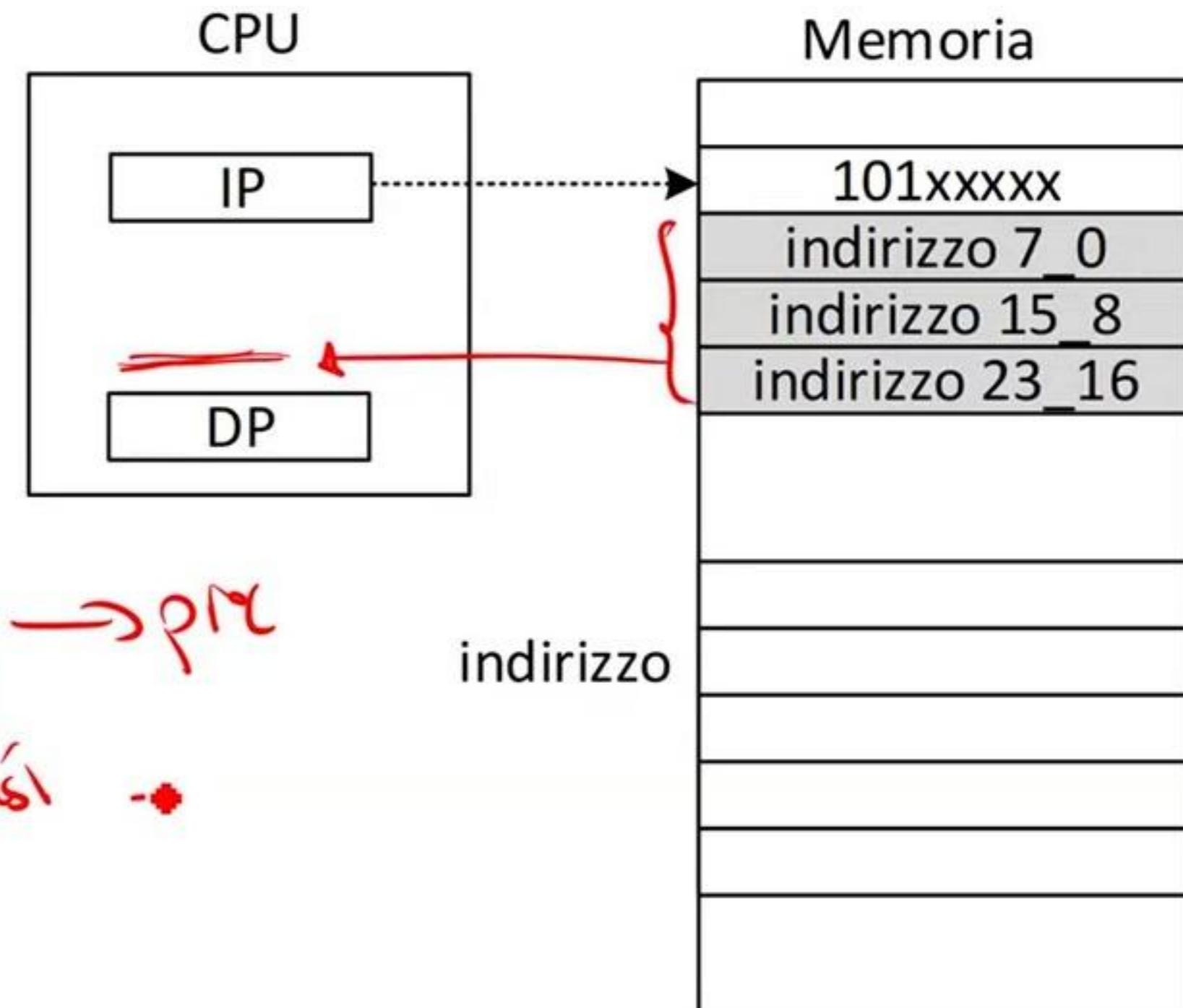
- Istruzioni in cui l'operando sorgente è indirizzato in modo diretto.
- Lunghe **4 byte**: **uno** di opcode e **tre** di indirizzo di memoria
 - lo spazio di memoria è a 24 bit.
- La fase di fetch dovrà quindi:
 - leggere in memoria 4 byte, a indirizzi consecutivi puntati dal registro IP
 - una volta recuperato l'indirizzo dell'operando sorgente, andare in memoria a leggere l'operando sorgente stesso



MOV	indirizzo, AL	10100000	<i>3 byte</i>	indirizzo
CMP	indirizzo, AL	10100001	indirizzo	
ADD	indirizzo, AL	10100010	indirizzo	
SUB	indirizzo, AL	10100011	indirizzo	
AND	indirizzo, AL	10100100	indirizzo	
OR	indirizzo, AL	10100101	indirizzo	
MOV	indirizzo, AH	10100110	indirizzo	
CMP	indirizzo, AH	10100111	indirizzo	
[...]				

Formato F6 (110)

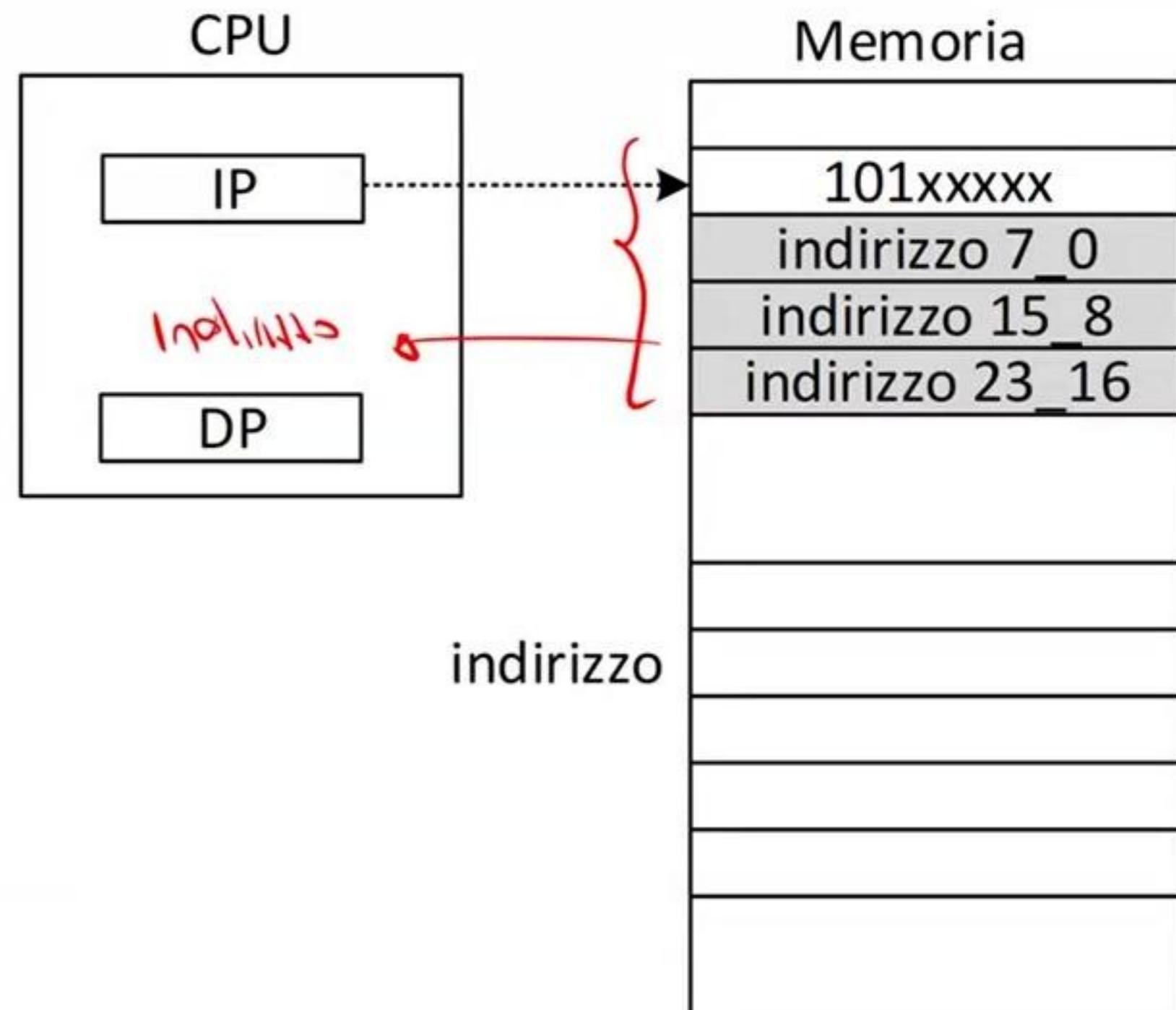
- Istruzioni in cui l'operando **destinatario** è in memoria, indirizzato in modo **diretto**.
- il processore dovrà leggere 4 byte in memoria, per procurarsi l'**indirizzo del destinatario**, a locazioni consecutive **puntate da IP**
 - L'indirizzo del destinatario è letto in memoria in fase di fetch
 - La scrittura del destinatario avviene nella fase di esecuzione



MOV AL, indirizzo |11000000| indirizzo |
MOV AH, indirizzo |11000001| indirizzo |

Formato F7 (111)

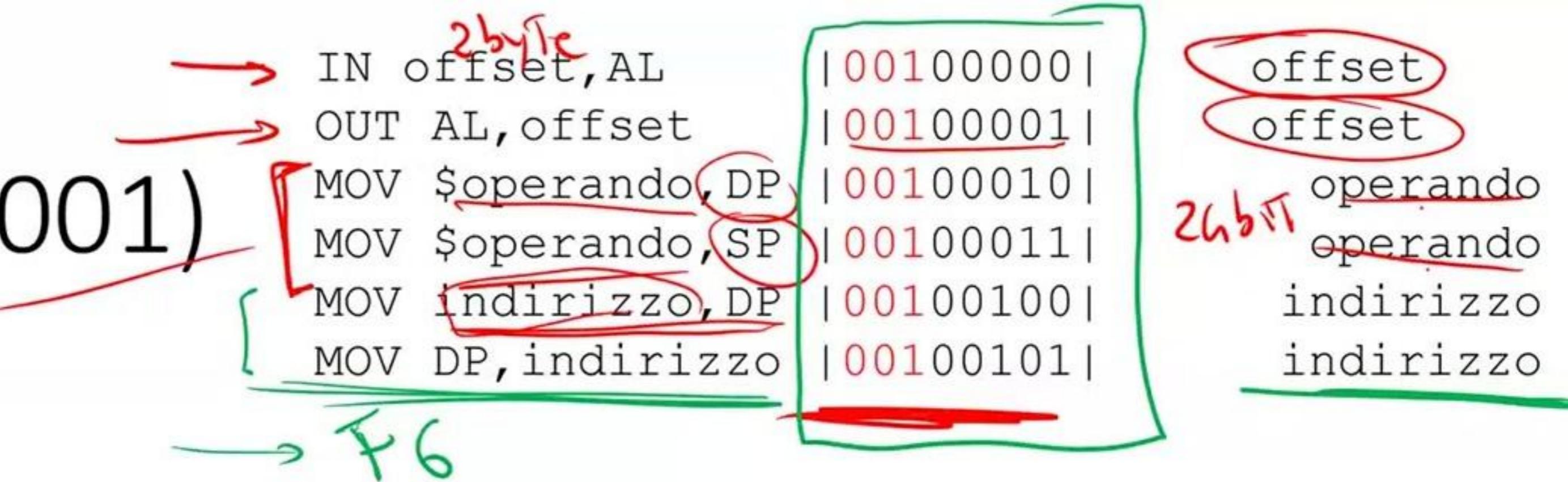
- Raggruppa le istruzioni di controllo (CALL, JMP, Jcon) in cui ho un indirizzo di salto, specificato in modo diretto nell'istruzione stessa, su 3 byte.
- Pertanto, in fase di fetch vanno letti 4 byte consecutivi, **puntati dall'indirizzo IP**.



	1	3	
JMP	indirizzo	11100000	indirizzo
JE	indirizzo	11100001	indirizzo
JNE	indirizzo	11100010	indirizzo
JA	indirizzo	11100011	indirizzo
JAE	indirizzo	11100100	indirizzo
JB	indirizzo	11100101	indirizzo
[...]			
CALL	indirizzo	11110011	indirizzo

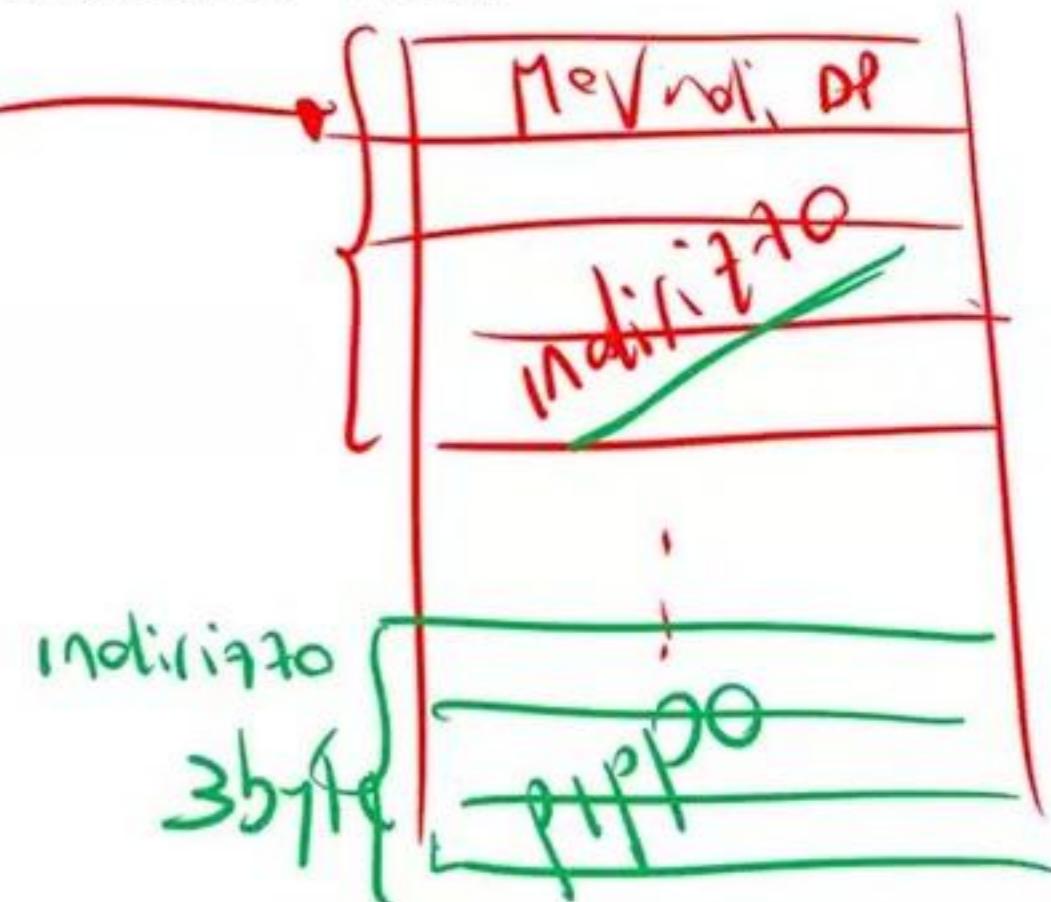
Formato F1 (001)

Fh



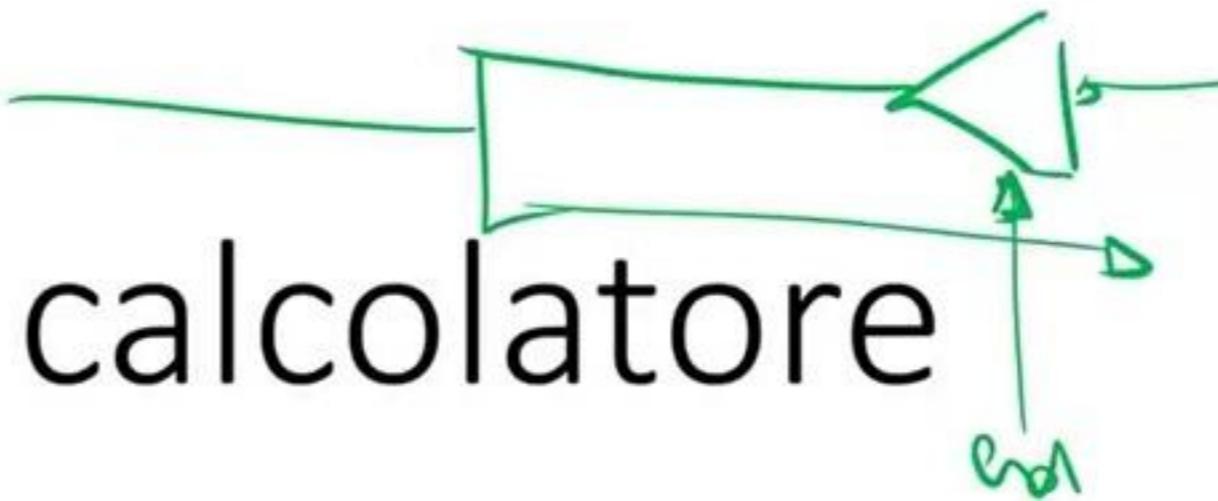
- Raggruppa tutte le istruzioni che non possono essere classificate nei precedenti formati

- Istruzioni di I/O
 - operando indirizzo a 16 bit, sorgente (IN) o destinatario (OUT)
- MOV con uno dei registri a 24 bit (DP o SP)
 - Operando a 24 bit sorgente, sia immediato che diretto, o destinatario



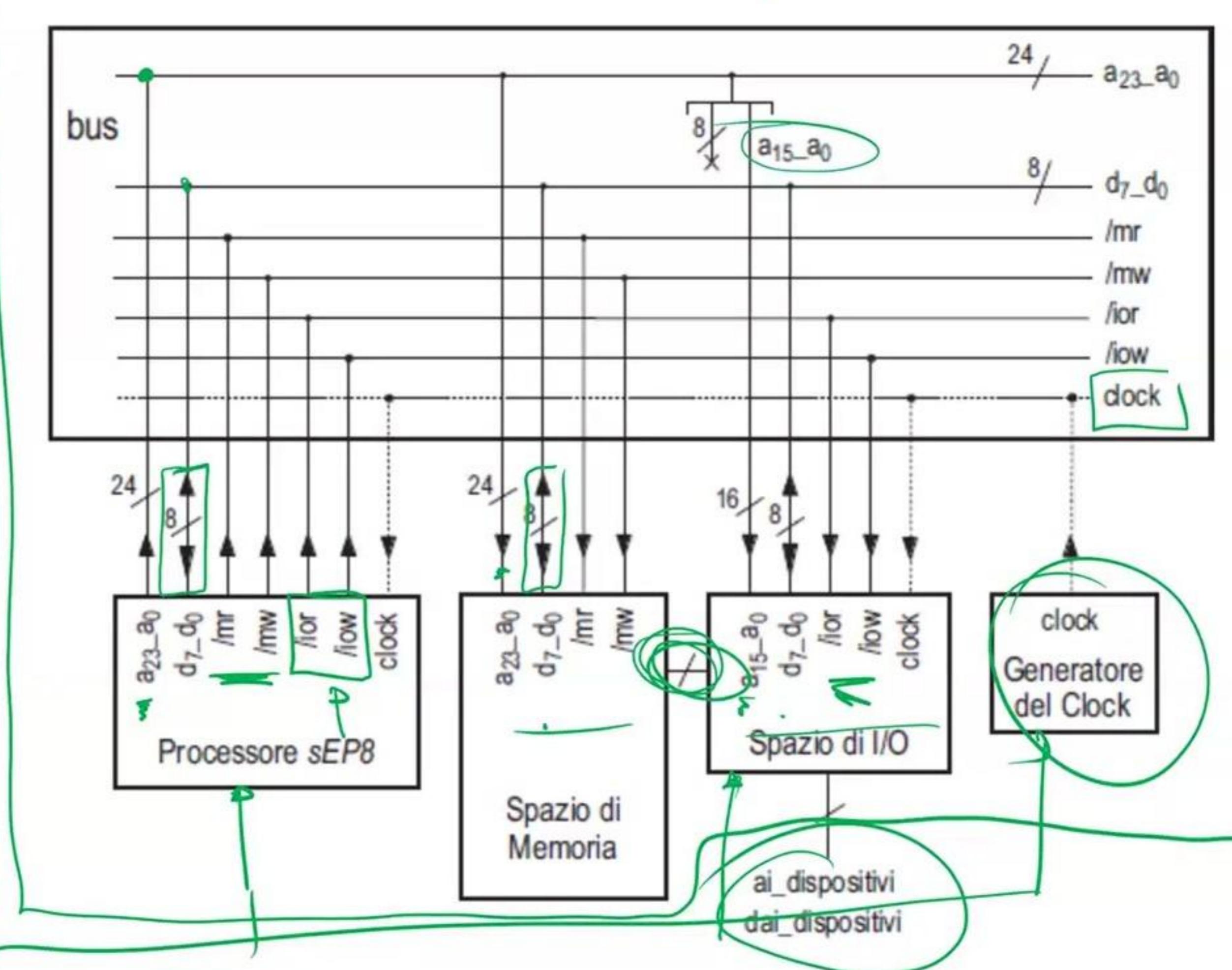
- Le azioni per procurarsi gli operandi sono diverse da un'istruzione all'altra
 - Meglio fare una fase di fetch «scarna» in cui prelievo solo l'opcode
 - Gestiremo gli operandi successivamente in fase di esecuzione
 - Poco pulito dal punto di vista concettuale, ma molto più semplice

Architettura hardware del calcolatore



- Bus
- **24 fili di indirizzo**
 - out per il processore, in per gli altri
- **8 fili di dati**
 - Proc. legge e scrive **byte**
 - In/out per processore. Forchetta.
- **Fili di controllo**
 - Tutti attivi bassi
 - /mr, /mw (per la memoria)
 - /ior, /iow (per le interfacce nello spazio di I/O)
- **Clock, reset**

1 rega



```

module Calcolatore(ai dispositivi,dai dispositivi);
  input [...:0] dai dispositivi;
  output [...:0] ai dispositivi;
  //bus
  wire [7:0] d7_d0;
  wire [23:0] a23_a0;
  wire mr_,mw_,ior_,iow_;
  wire clock, reset_;
  wire [15:0] a15_a0; assign a15_a0=a23_a0[15:0];
  //Collegamenti tra memoria video e adattatore grafico
  wire [...:0] a_mem_video;
  wire [...:0] da_mem_video;
  //Moduli costituenti il calcolatore
  Processore P(d7_d0,a23_a0,Mr_,mw_,ior_,iow_,clock,reset_);
  Spazio_di_Memoria SdM(d7_d0,a23_a0,Mr_,mw_,da_mem_video,a_mem_video);
  Spazio_di_IO SdIO(d7_d0,a15_a0,ior_,iow_,a_mem_video,da_mem_video,
                    ai_dispositivi,dai_dispositivi,clock,reset_);
  Generatore_del_Clock GC(clock);
endmodule

```

{ module Processore(...); ... endmodule
 module Spazio_di_Memoria(...); ... endmodule
 module Spazio_di_IO(...); ... endmodule
 module Generatore_del_Clock(...); ... endmodule
 module Gruppo RC con trigger di Schmitt(...); ... endr }

Spazio di memoria

26 bit

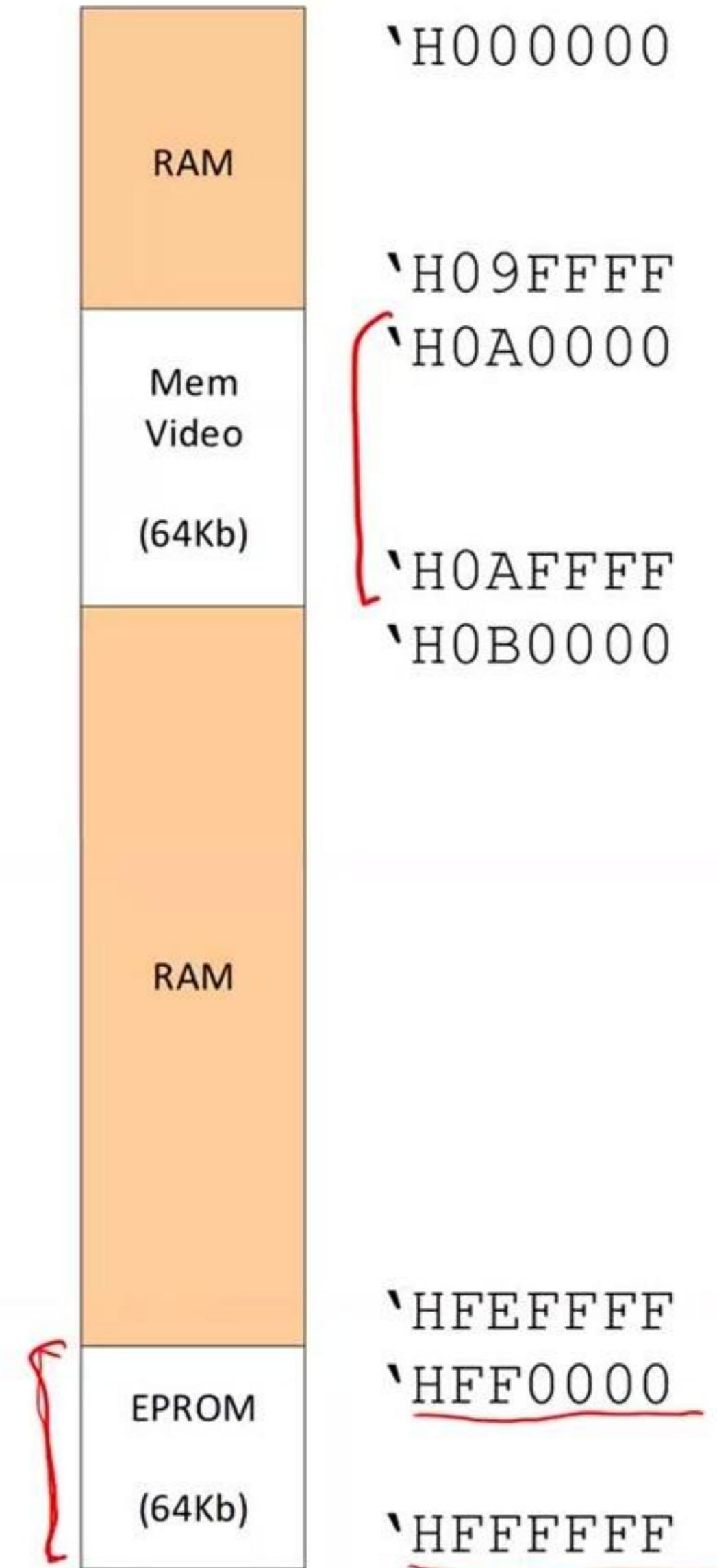
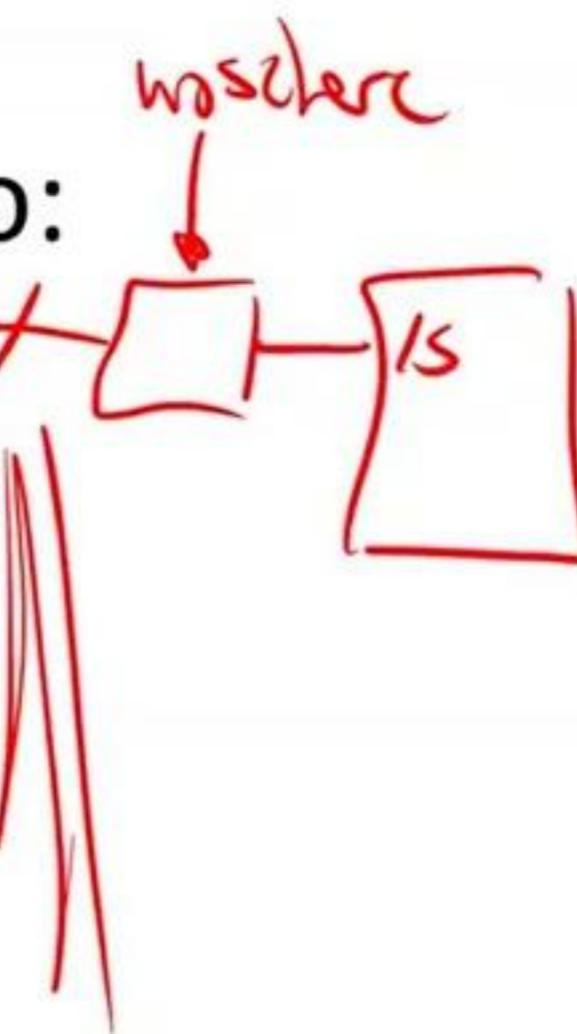
- 16Mbyte

- in parte con tecnologia RAM,
- in parte EPROM (la parte che contiene il bootstrap),
- in parte memoria video (di tipo diverso, che non vediamo)

$2^6 \cdot 2^{10}$

- Esempio: 64k di EPROM e 64k di mem. video:

- EPROM: [`'HFF0000`; `'HFFFFFFFFFF`]
- Memoria video [`'H0A0000`; `'H0AFFFF`]
00001010
- Il resto è RAM
 - [`'H000000`; `'H09FFFF`]
 - [`'H0B0000`; `'HFEFFFFF`]

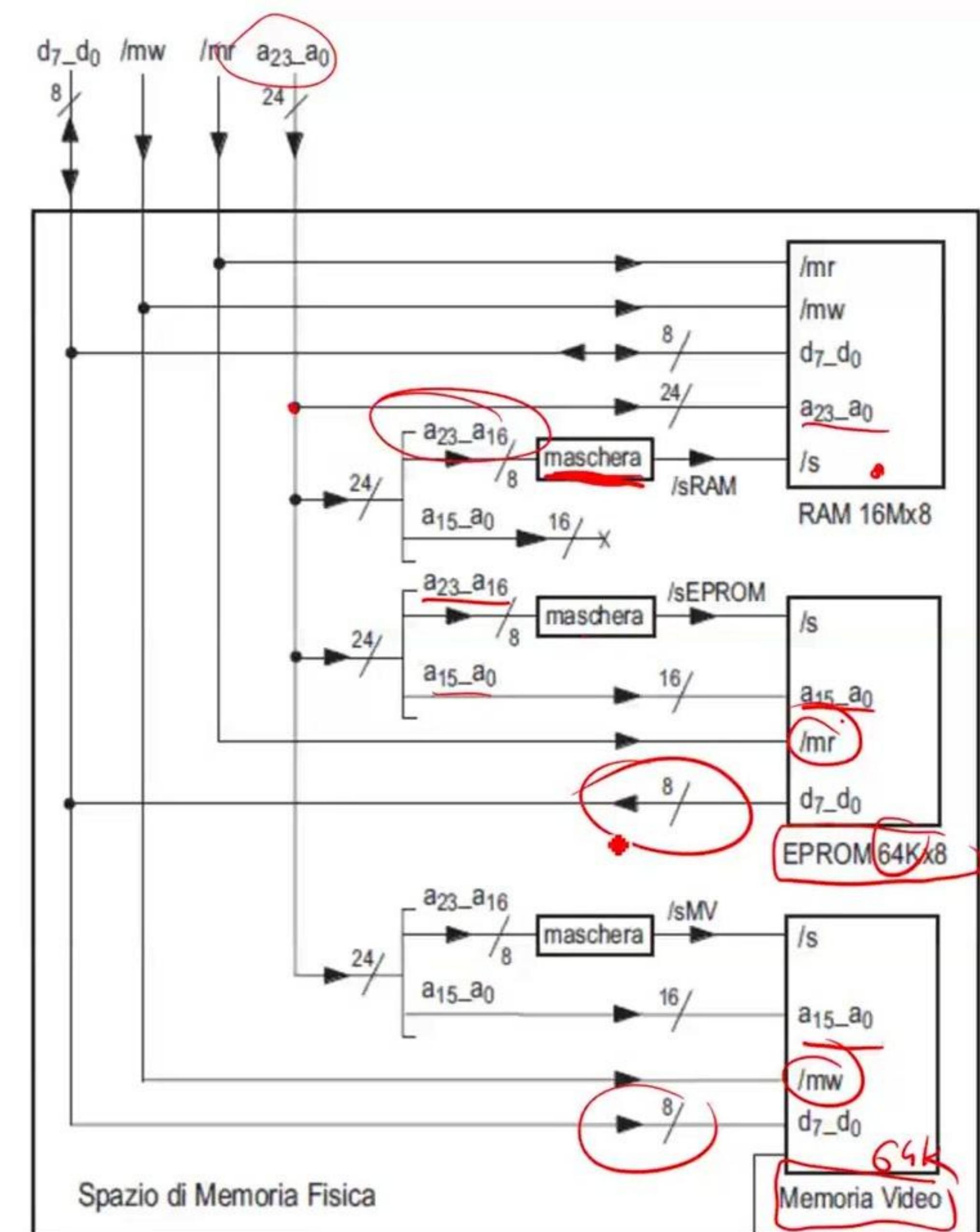


Spazio di memoria (cont.)

- I fili di indirizzo a_{10} - a_{16} entrano anche nella RAM, oltre che nella maschera.
- Il modulo RAM potrebbe coprire tutto lo spazio, ma **non è selezionato** quando il range di indirizzi è quello della EPROM o della mem. video.
- La EPROM non ha l'ingresso /mw

a_{23}	a_{22}	a_{21}	a_{20}	a_{19}	a_{18}	a_{17}	a_{16}	/sRAM	/sMV	/sEPROM
0	0	0	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	0

→ $\overline{a_{23} a_{22} a_{21}}$ altro $\overline{a_{19} a_{18} a_{17}}$ $\overline{a_{16}}$

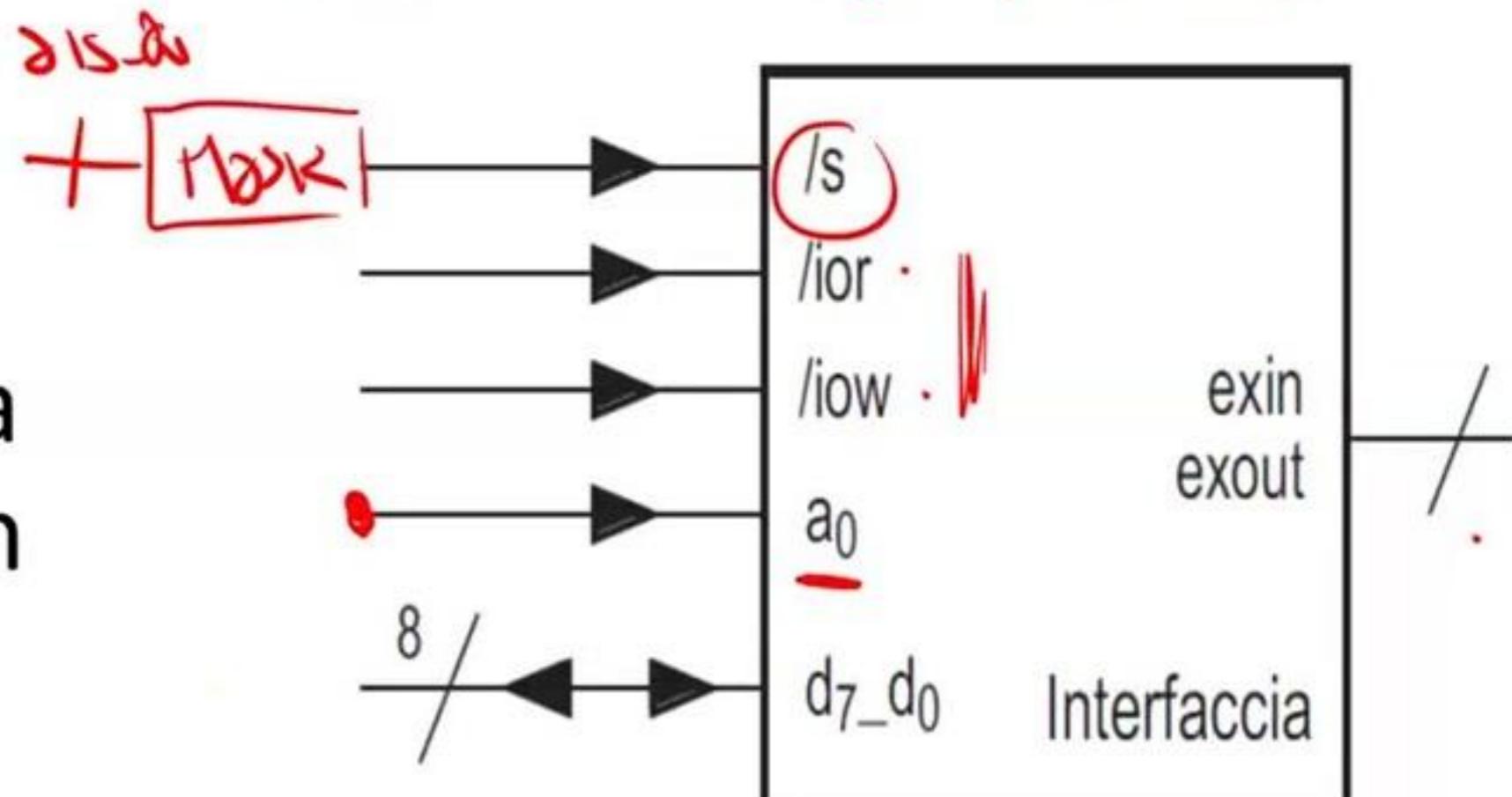


Spazio di I/O

2^{16} 64K

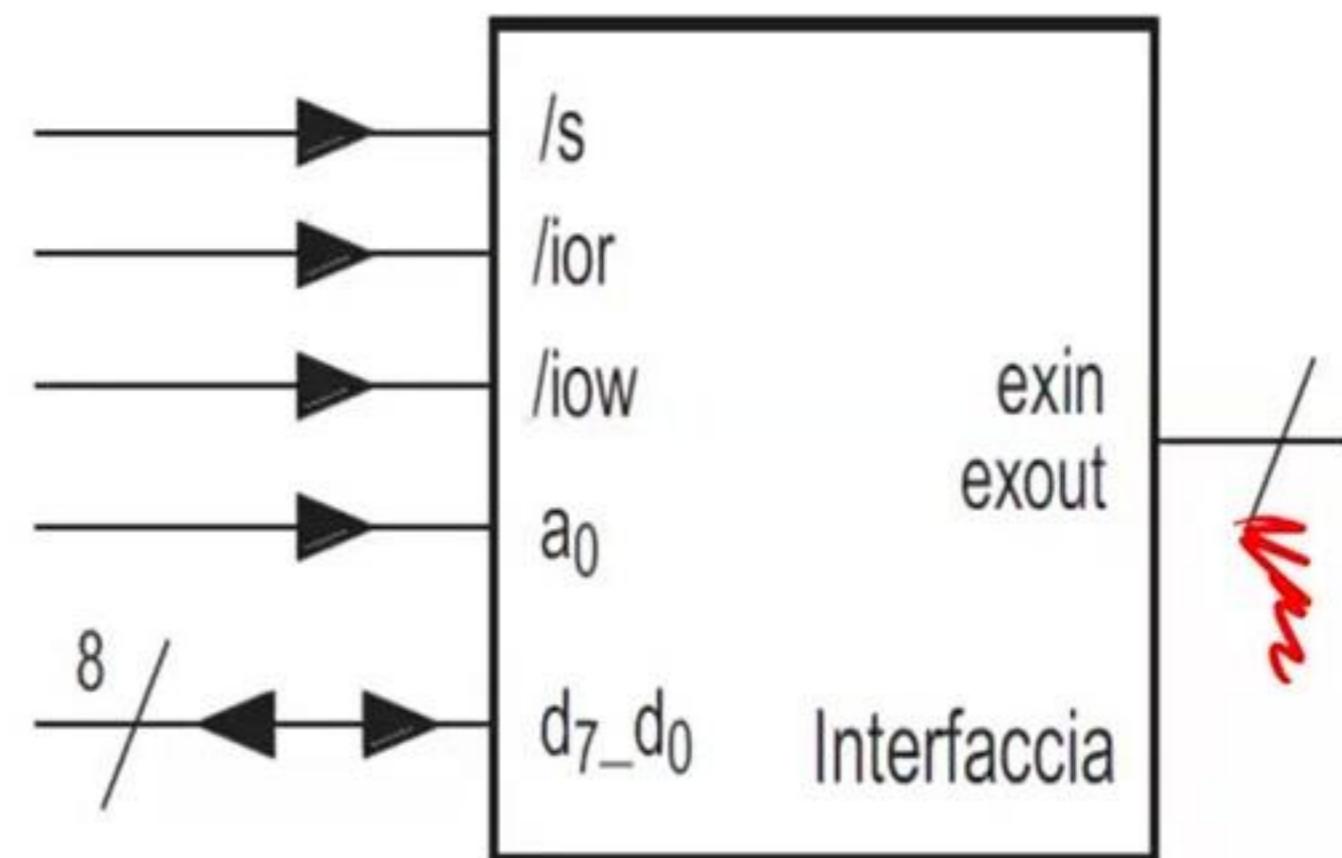
- Realizzato fisicamente tramite **interfacce**, che fungono da **raccordo tra il bus e i dispositivi di I/O**
- Un'interfaccia ha dei collegamenti sia “lato bus” che “lato dispositivo”.
- Dal lato del bus, collegamenti simili a quelli di una **piccola memoria RAM**, di poche locazioni (due, in questo esempio)
- Le locazioni che si trovano nelle interfacce prendono il nome di **porte** di ingresso e uscita.

/s	/ior	/iow	azione
1	-	-	nessuno
0	1	1	"
0	0	1	ciclo lettura
0	1	0	ciclo scrittura
0	0	0	non def.



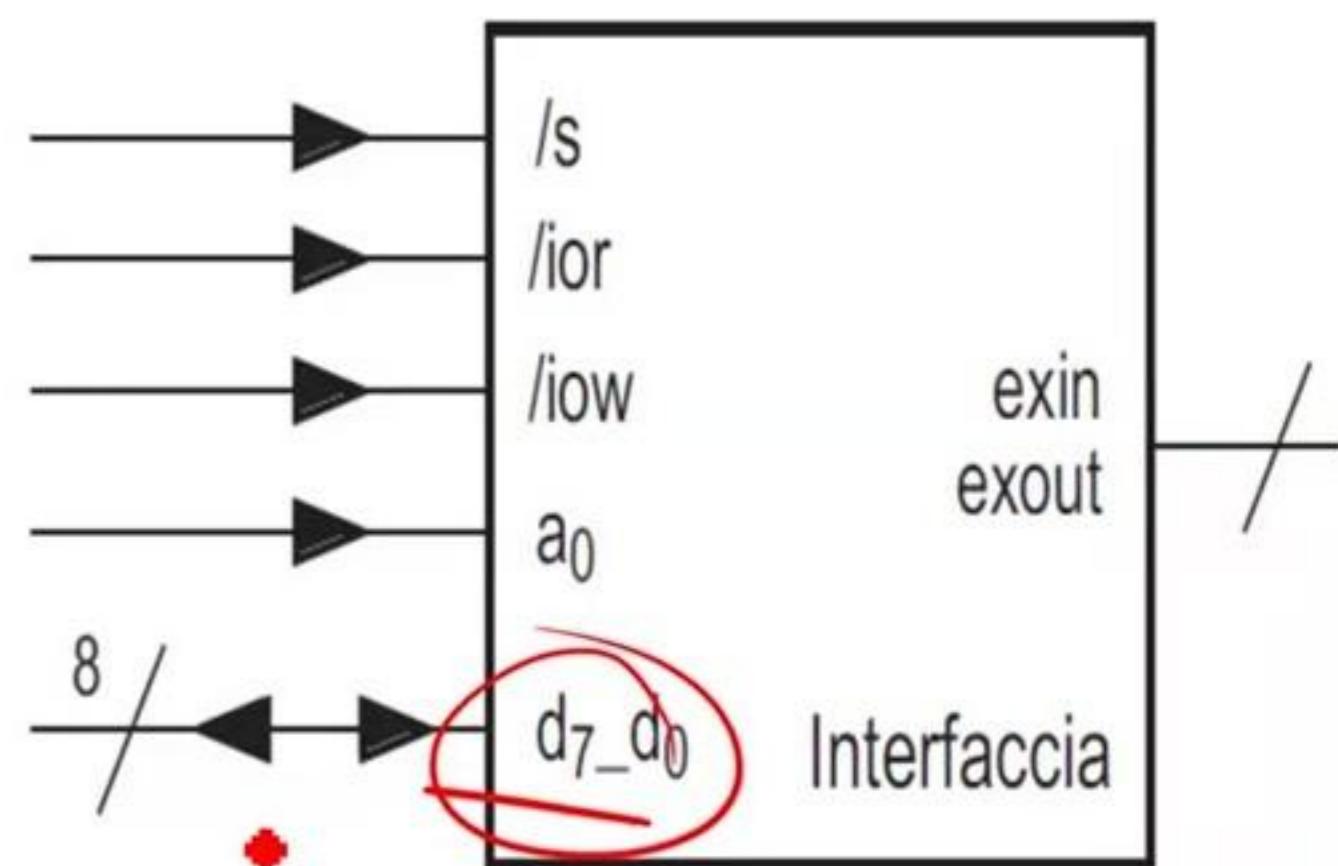
Spazio di I/O (cont.)

- **Collegamenti dal lato del bus**
- /ior, /iow: comandi di lettura e scrittura
 - Cicli di lettura/scrittura simili a quelli della memoria
- /s: proviene da una **maschera**, che sente alcuni fili di indirizzo
- a0: filo di indirizzo interno, seleziona una porta di due
 - Se l'interfaccia ha una sola porta, non necessario
- Dati bidirezionali
- **Collegamenti dal lato del dispositivo**
 - Variabili, dipendono dal dispositivo



Spazio di I/O (cont.)

- in una RAM si può leggere e scrivere qualunque locazione.
- Spesso in un'interfaccia una porta supporta
 - **soltanto lettura** (istruzione IN)
 - **soltanto scrittura** (Istruzione OUT)
- In una singola interfaccia ci sono spesso porte di entrambi i tipi



Interfacce e dispositivi

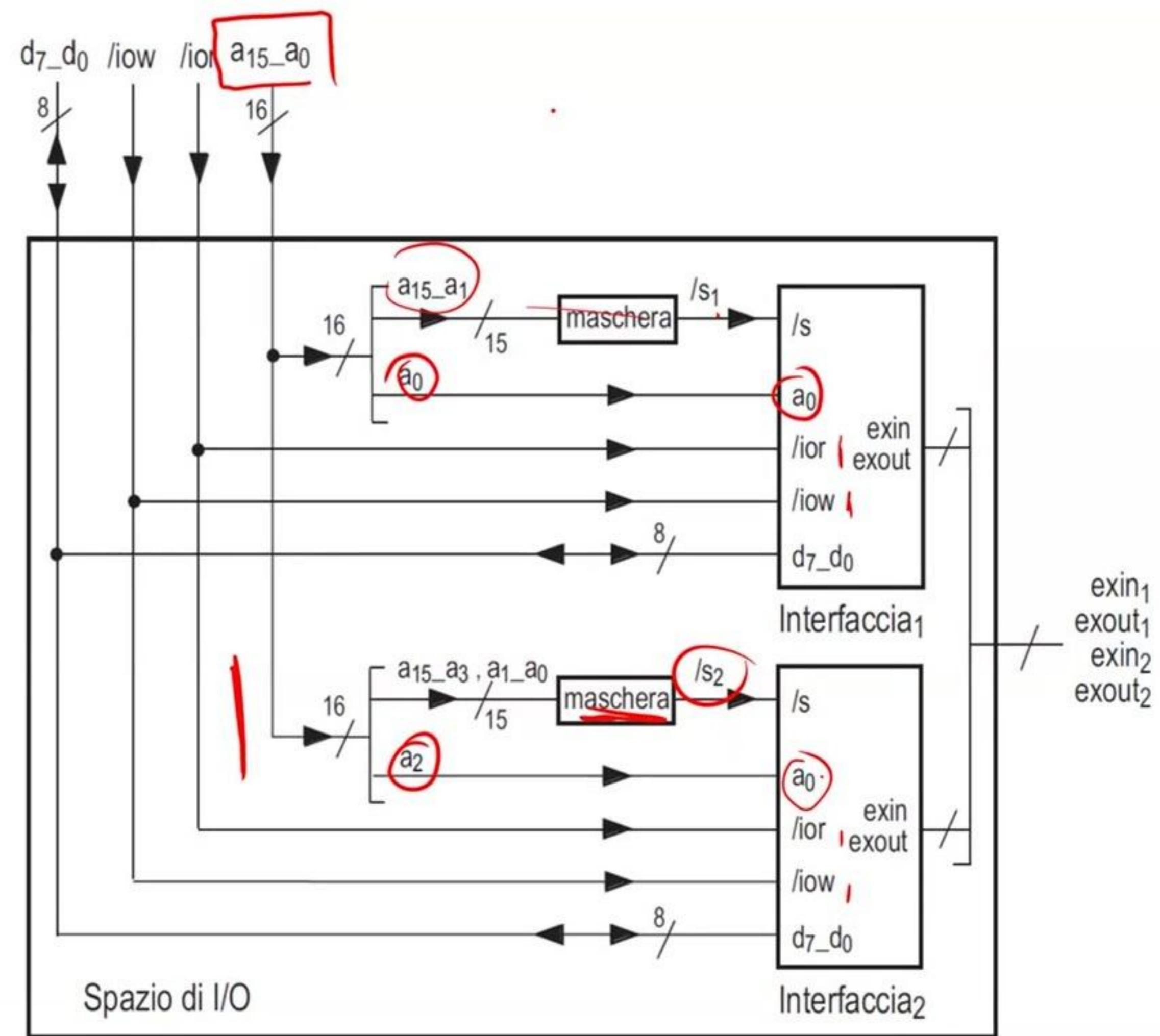
- Perché c'è bisogno delle interfacce?
- I dispositivi hanno **velocità molto diverse tra loro**
 - Diverse per ordini di grandezza
 - Sono spesso **molto più lenti del processore**
- Hanno **modalità di trasferimento dati** molto diverse
 - Alcuni trasferiscono **un bit alla volta** (seriali)
 - Altri **gruppi di bit** (e.g., byte).
- Le interfacce rendono i dispositivi visibili in modo **standard**
 - Temporizzazioni omogenee
 - Trasferimento dati omogeneo

Spazio di I/O (cont.)

- Due interfacce ciascuna a due porte (sulle quali si può leggere e scrivere)
- La prima dà corpo a due porte di offset 'H03C8, 'H03C9
- La seconda dà corpo a due porte di offset 'H0060, 'H0064 (**non contigui**)

H0060

0000 0000 0110 0000
 0100
 a2



Processore

- **Registri**

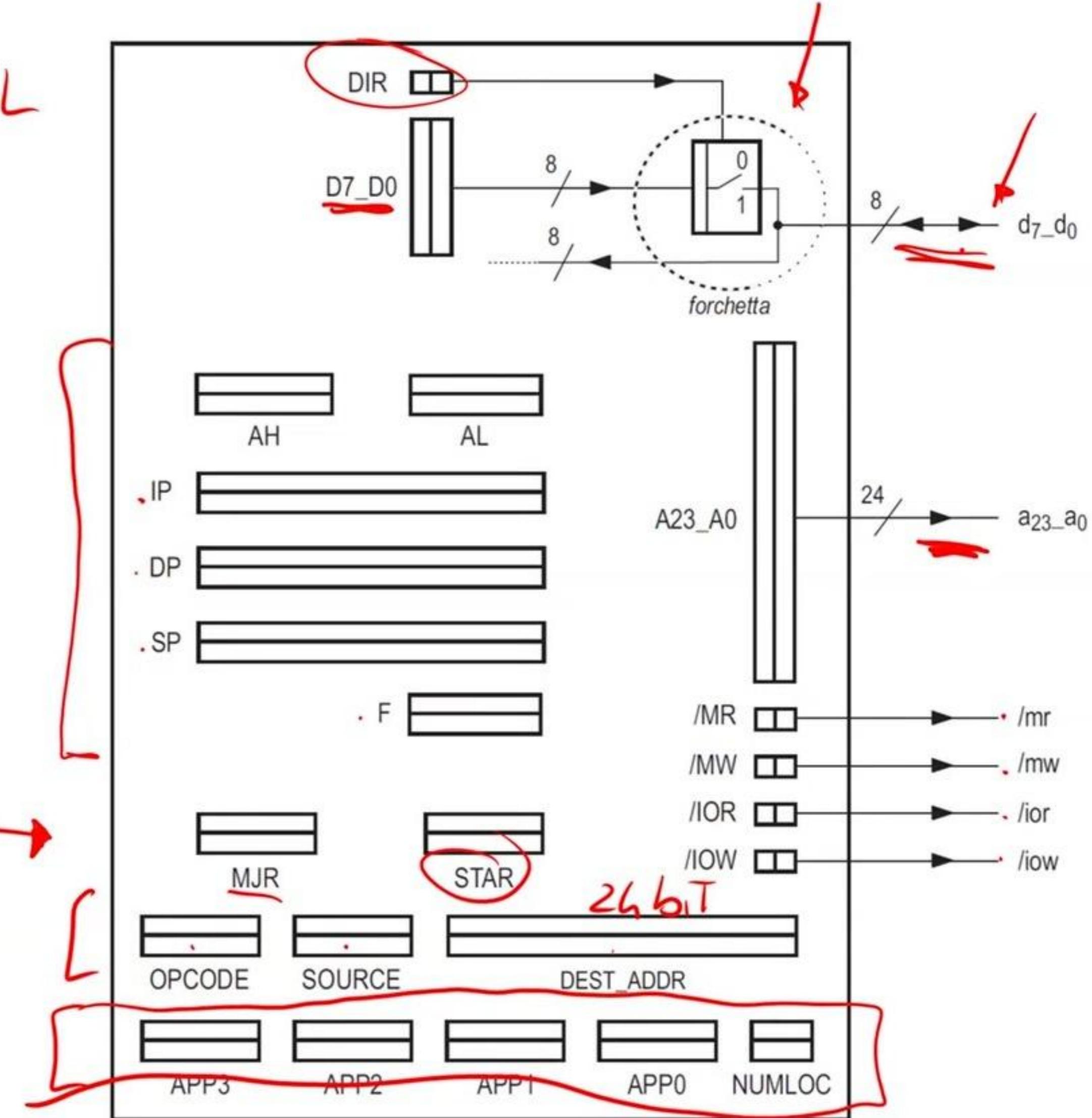
- Quelli visibili al programmatore
- Alcuni per sostenere **le uscite**
- STAR e MJR
- OPCODE, SOURCE, DEST_ADDR
- Altri come «appoggio» per operazioni

- **Dimensioni**

- 8 bit (operandi)
- 24 bit (indirizzi)
- 1 bit (fili di controllo del bus)

Mov Indirizzo, AL
26bit

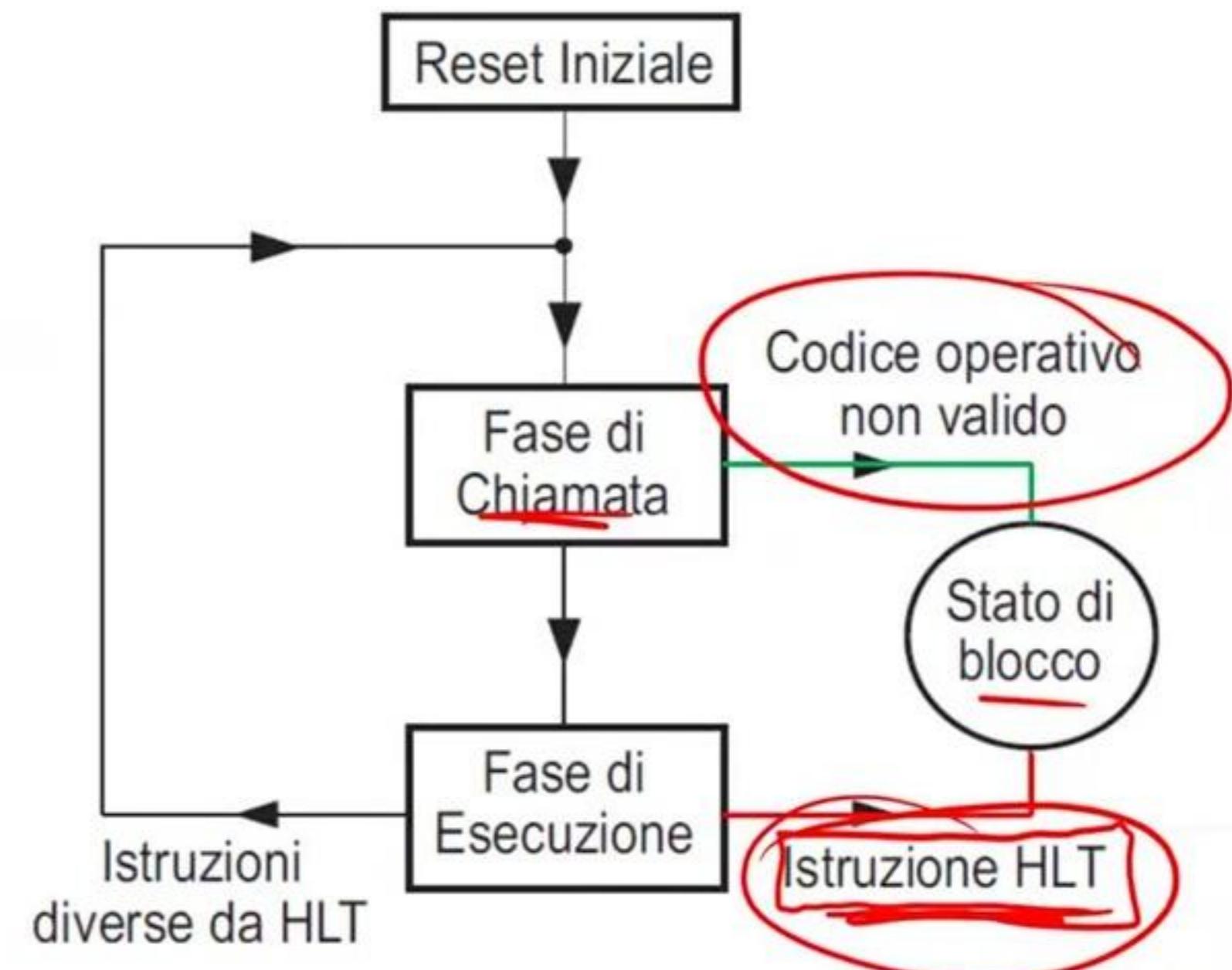
Instruction
Registers



Processore (cont.)

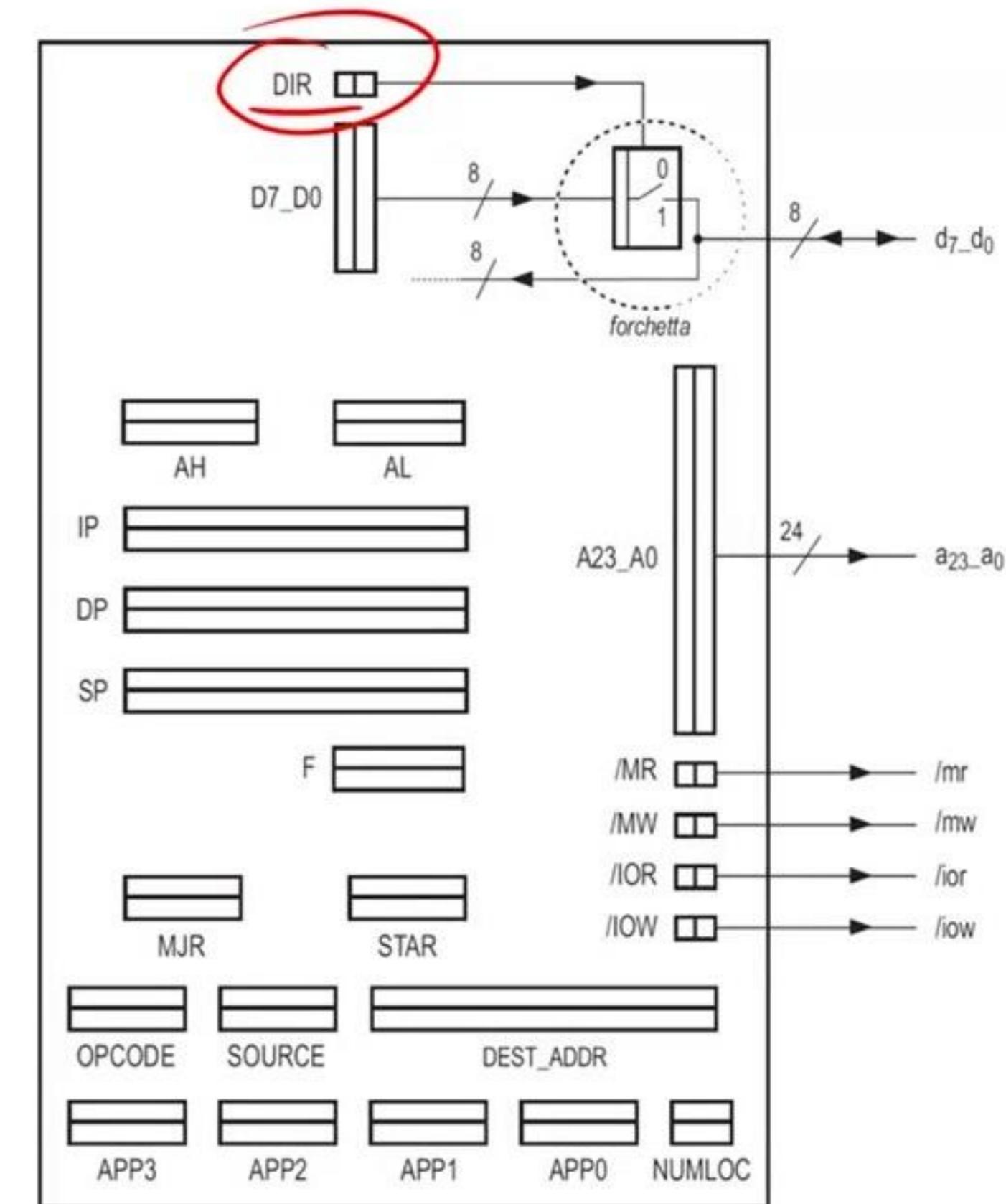
- Fase di reset
 - Inizializzazione registri (F, IP, altri)
- Fase di fetch
- Fase di esecuzione
- Stato di blocco
 - Si **esegue HLT**
 - Si **preleva un opcode non valido**
- Analizziamo le fasi in maggior dettaglio

HLT : STDR<=HLT;
[- - - - -]



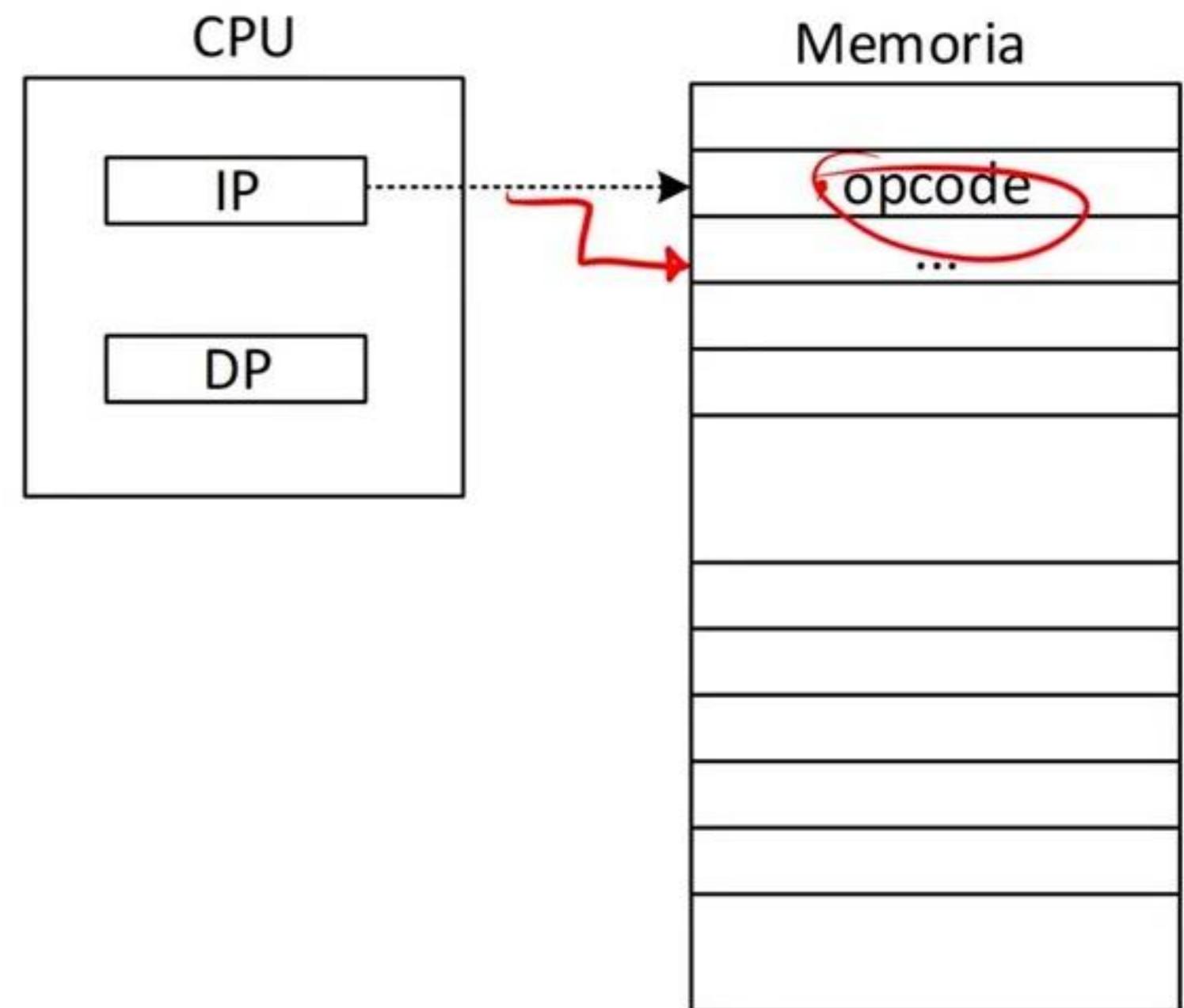
Fase di reset

- **IP** \leftarrow 'HFF0000, primo indirizzo del blocco EPROM
- **F** \leftarrow 0
- **Tutti i registri** che reggono variabili di controllo del bus dovranno essere inizializzati in modo coerente: **/MR, /MW, /IOR, /IOW** \leftarrow 1.
- Fili di dati **in alta impedenza**. **DIR** \leftarrow 0
 - DIR sarà **sempre a 0**, tranne quando devo scrivere
- **STAR** verrà inizializzato con l'etichetta del primo statement della fase di fetch
- Gli altri registri non hanno bisogno di essere inizializzati



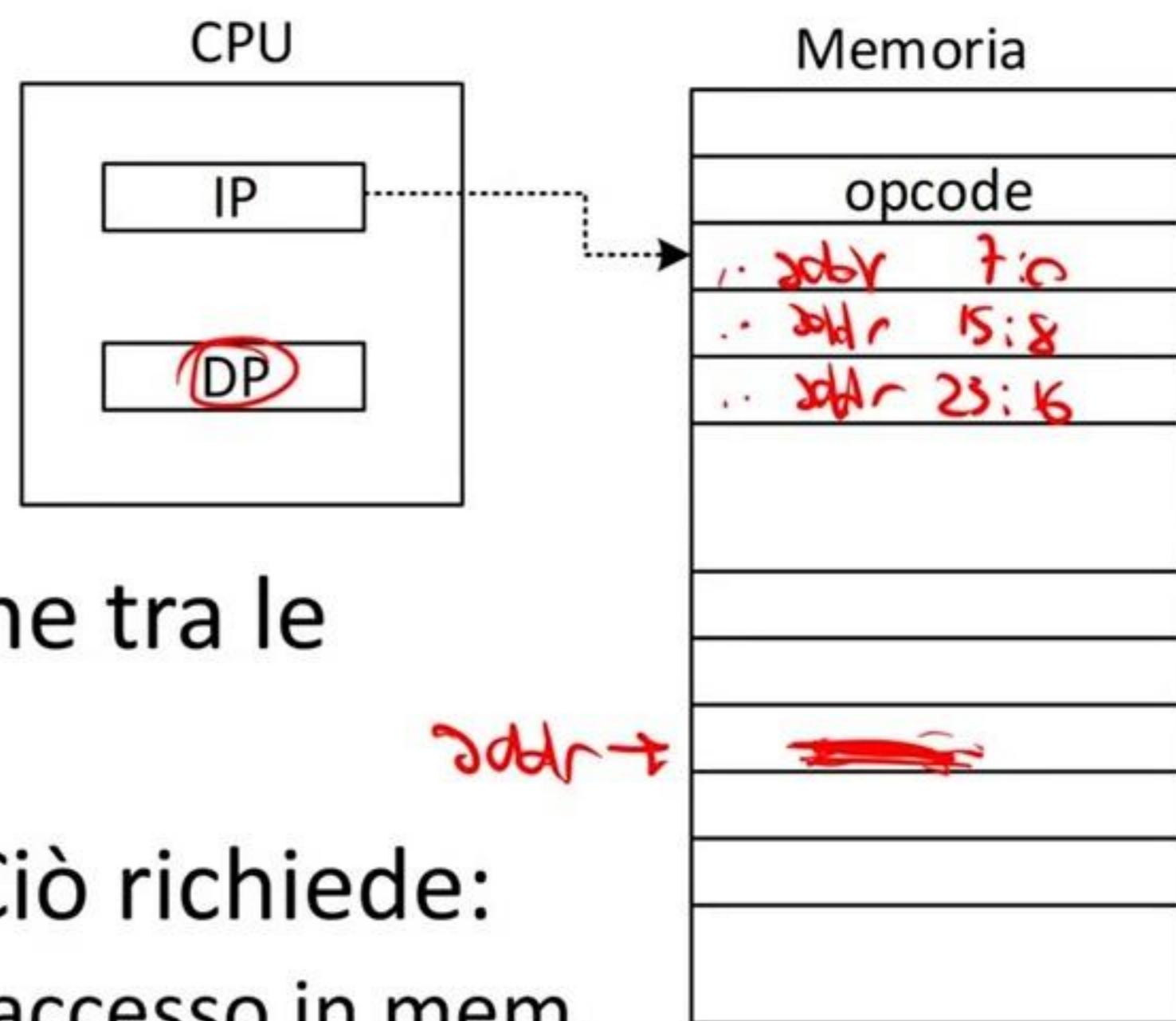
Fase di fetch

- Il processore:
 1. **preleva un byte** dalla memoria, all'indirizzo indicato in **IP**
 2. **incrementa IP** (modulo 2^{24})
 3. **controlla** che quel byte corrisponda all'opcode di una delle istruzioni che conosce. Se non è così si **blocca** (come se avesse eseguito una HLT)
 4. **inserisce** il byte letto nel registro **OPCODE**, e valuta il **formato dell'istruzione**.



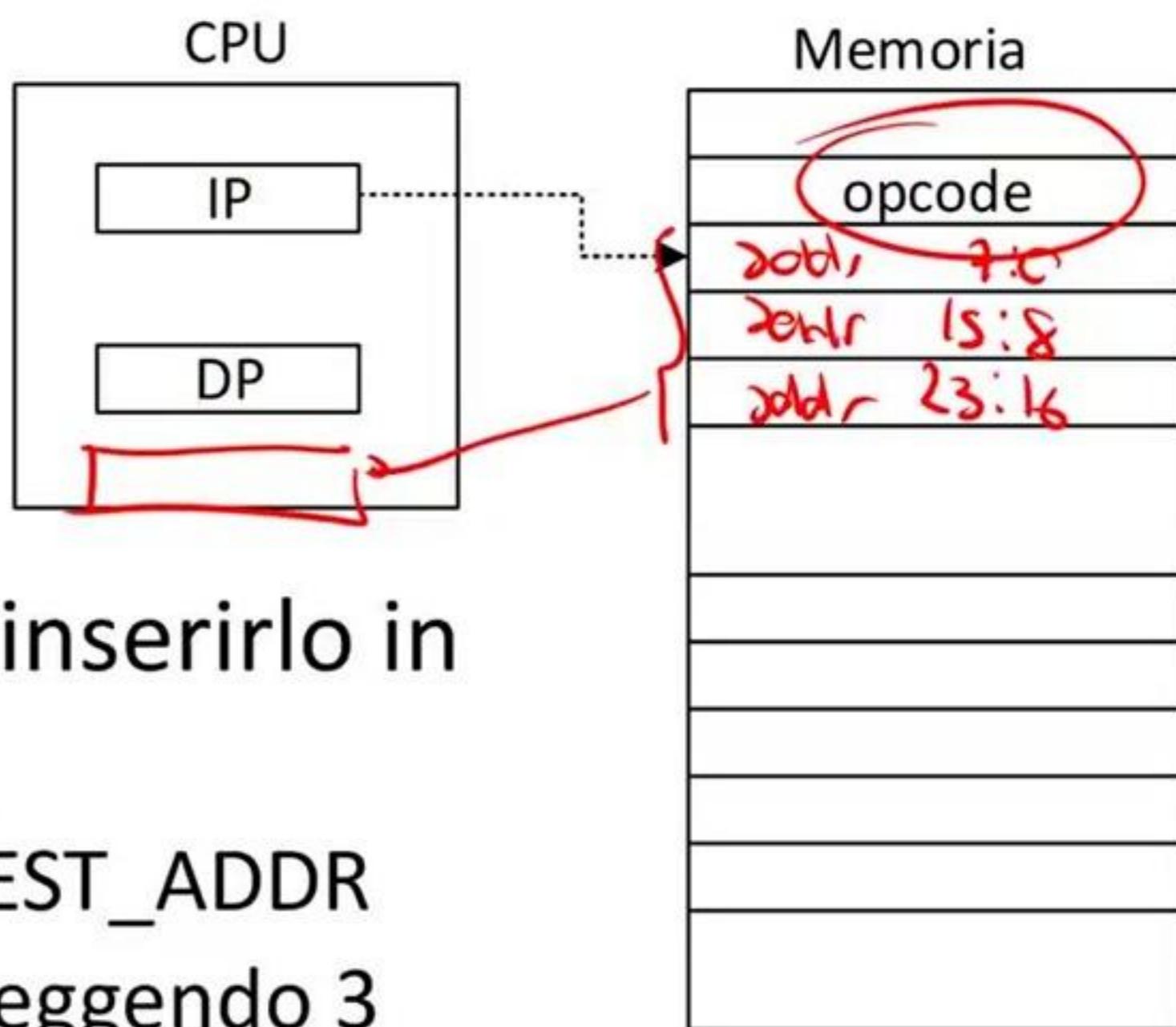
Fase di fetch (cont.)

Mov addr, AL



- A seconda del formato, il processore deve fare alcune tra le seguenti cose:
 - SOURCE \leftarrow operando sorgente a 8 bit (F2, F4, F5). Ciò richiede:
 1. F2, sorgente in mem. con indirizzamento indiretto: un accesso in mem. all'indirizzo contenuto in DP
 2. F4, sorgente immediato: un accesso in mem. all'indirizzo contenuto in IP
 3. F5, sorgente in mem. con indirizzamento diretto: **due accessi** in memoria:
 1. all'indirizzo contenuto in IP per procurarsi l'indirizzo (3 byte)
 2. all'indirizzo appena trovato (1 byte).
 - Nei formati F4 e F5 dovrà anche incrementare IP opportunamente
 - +1 (F4), +3 (F5)

Fase di fetch (cont.)



- Procurarsi l'**indirizzo dell'operando destinatario**, ed inserirlo in **DEST_ADDR** (formati F3, F6, F7).
 - nel formato **F3** l'indirizzo sta già in **DP**. Basta copiarlo in **DEST_ADDR**
 - nei formati **F6** ed **F7** devo andarlo a leggere in memoria, leggendo 3 byte puntati da **IP**, ed incrementando di 3 il valore di **IP**
- Nel formato **F0** non deve fare niente
- Nel formato **F1** il processore farà cose particolari (più avanti)
- Come ultima cosa, in fase di fetch si guarda il contenuto di **OPCODE**
 - Per capire quale è l'istruzione da eseguire

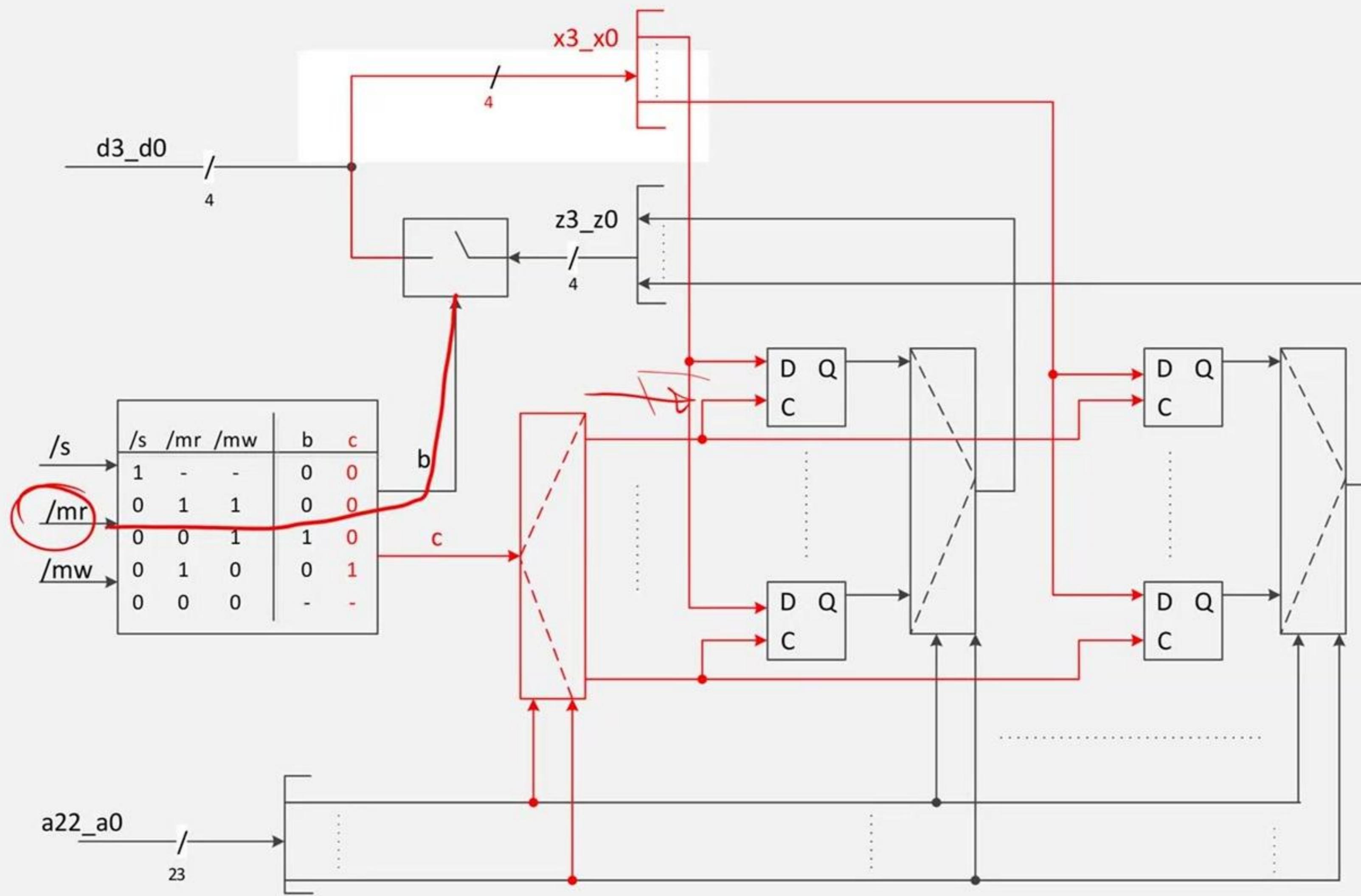
Fase di esecuzione

- Il processore **esegue** l'istruzione che ha decodificato
- torna nella fase di fetch
- a meno che non stia eseguendo l'istruzione di HLT, nel qual caso si blocca e potrà essere sbloccato soltanto da un nuovo reset

Letture e scritture in memoria e spazio di I/O

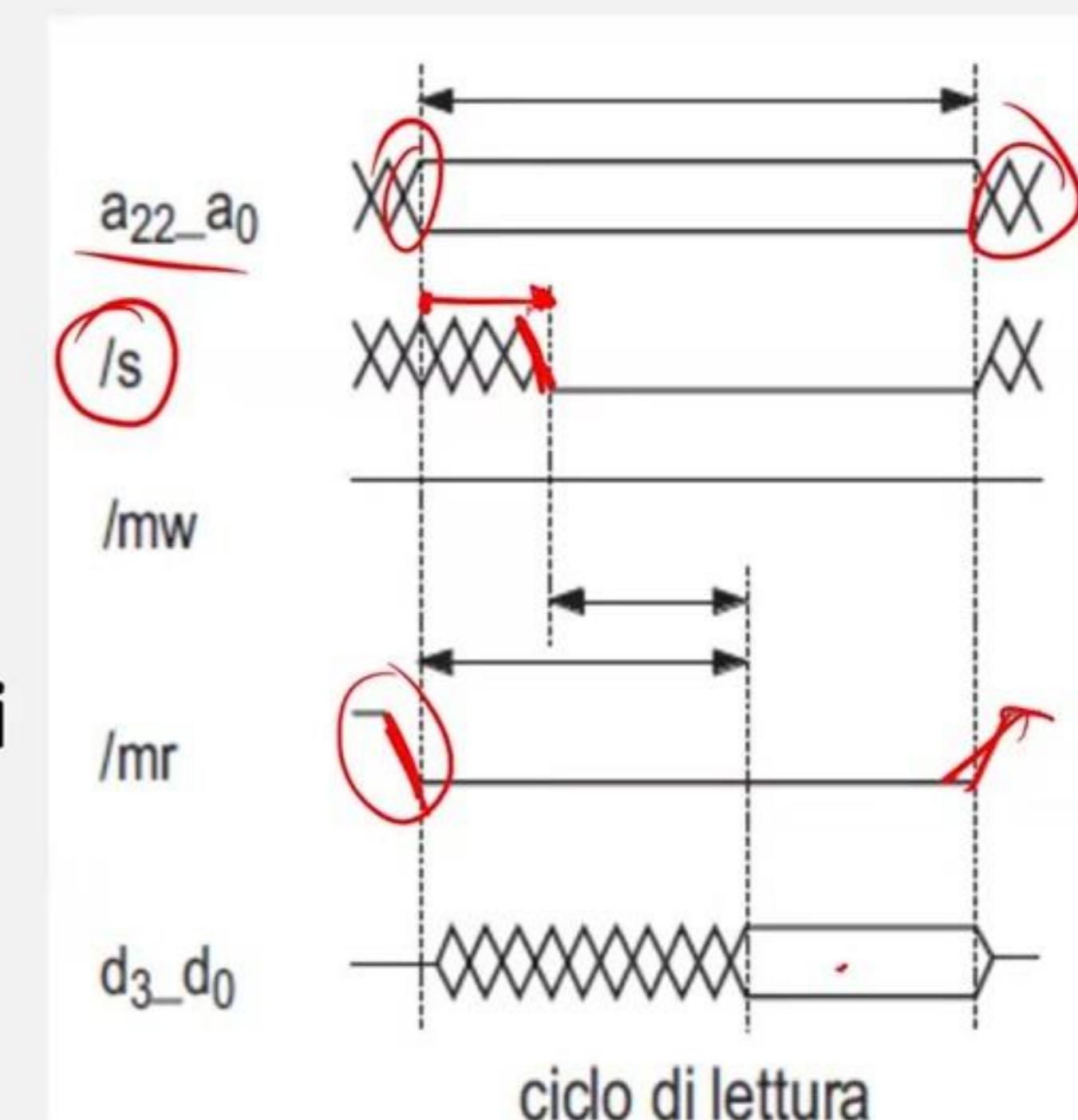
- Durante la fase di fetch, il processore **legge in memoria**
- Durante quella di esecuzione, il processore dovrà **leggere e scrivere in memoria (RET, MOV) o nello spazio di I/O (IN, OUT)**.
- Vediamo, a livello di microistruzioni, come si fa a scrivere un frammento di microprogramma **compatibile con le temporizzazioni viste a suo tempo** per i cicli di lettura e di scrittura.

Struttura della RAM statica



Temporizzazione delle RAM statiche - lettura

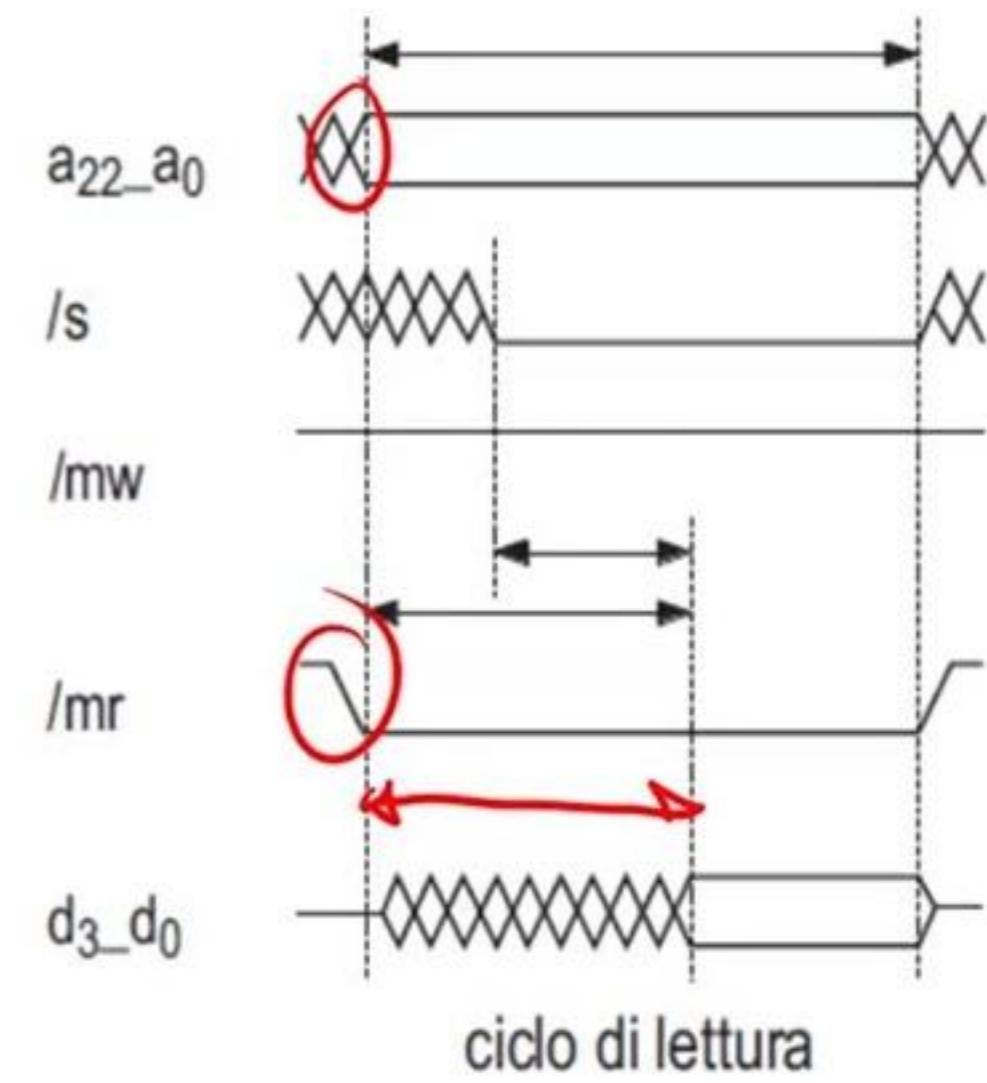
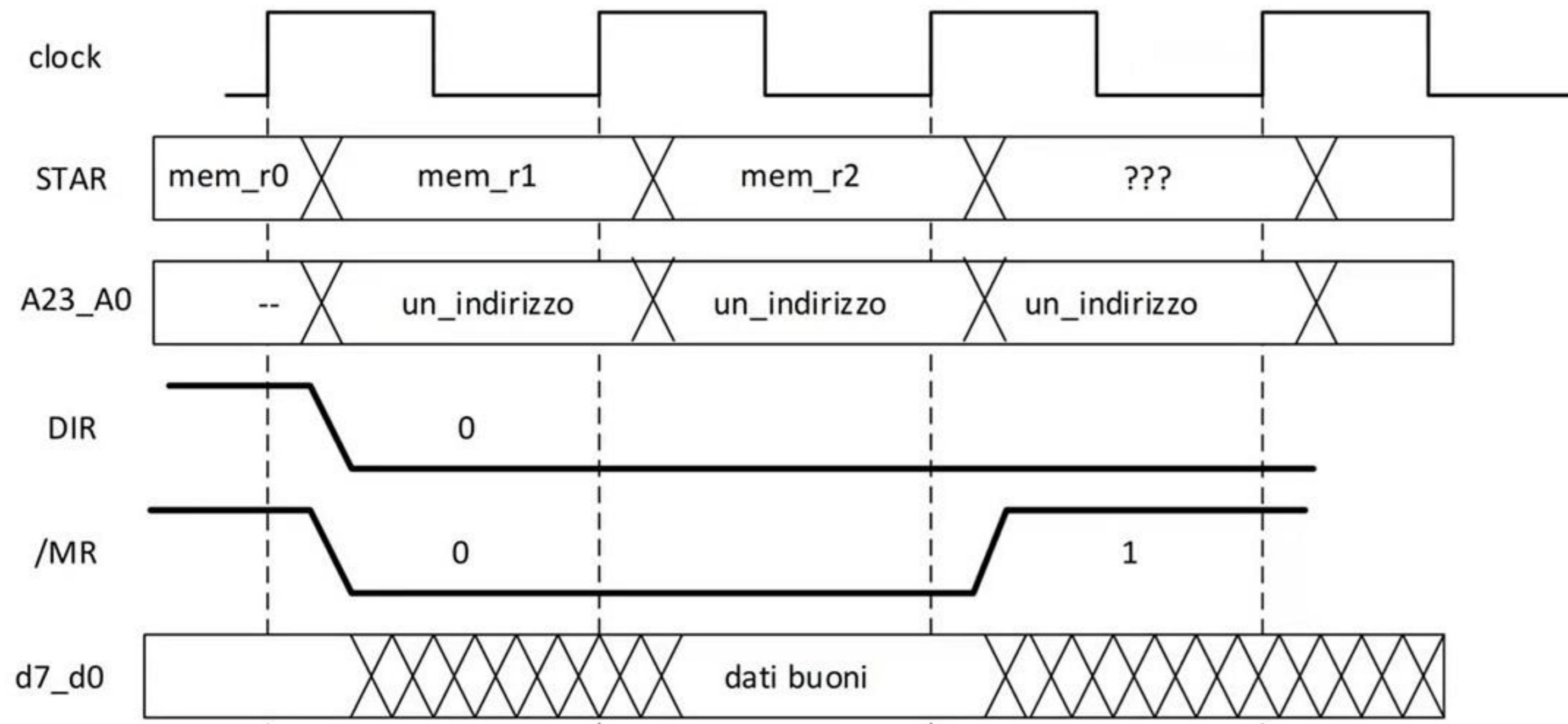
- Gli indirizzi si stabilizzano ed arriva il comando di **/mr**.
- Il comando di **/s** arriva con un po' di ritardo, e **balla** nel frattempo, (funzione combinatoria di altri bit di indirizzo)
- Quando **sia /s che /mr sono a 0**
 - le tri-state vanno in conduzione
- I multiplexer sulle uscite vanno a regime dopo gli indirizzi
 - Da lì in poi i dati sono buoni, e chi li ha richiesti li può prelevare
- Quando **/mr torna ad 1** (dopo che **chi voleva leggere i dati li ha prelevati**), i dati tornano in alta impedenza.
- A quel punto gli indirizzi e **/s** possono tornare a ballare



Ciclo di lettura in memoria – μ -istruzioni

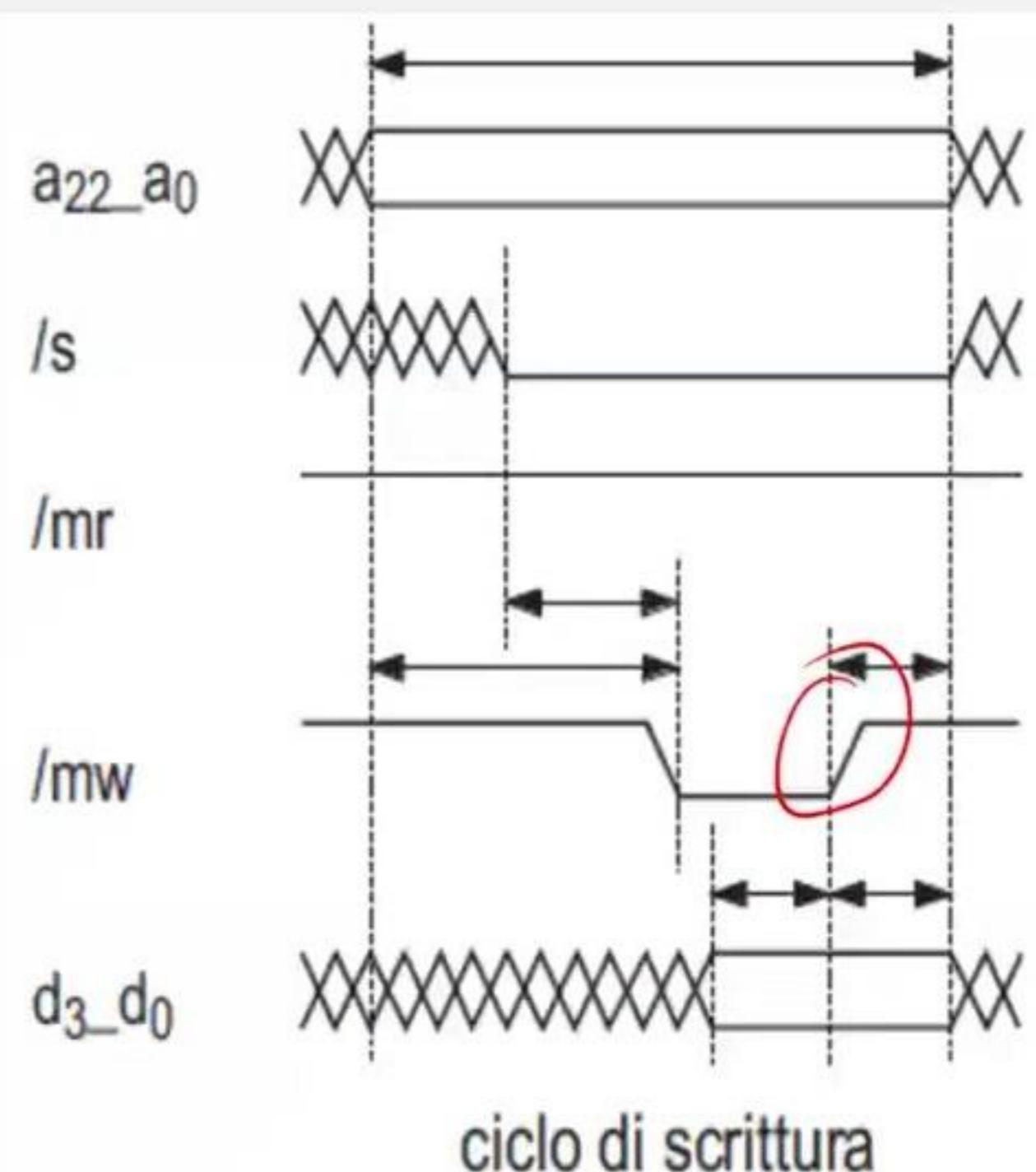
- Registri coinvolti : A23_A0, DIR, MR_

```
mem_r0: begin A23_A0<=un_indirizzo; DIR<=0; MR_<=0; STAR<=mem_r1; end  
mem_r1: begin STAR<=mem_r2; end //stato di wait  
mem_r2: begin QUALCHE_REGISTRO<=d7_d0; MR_<=1; ..... ; end
```



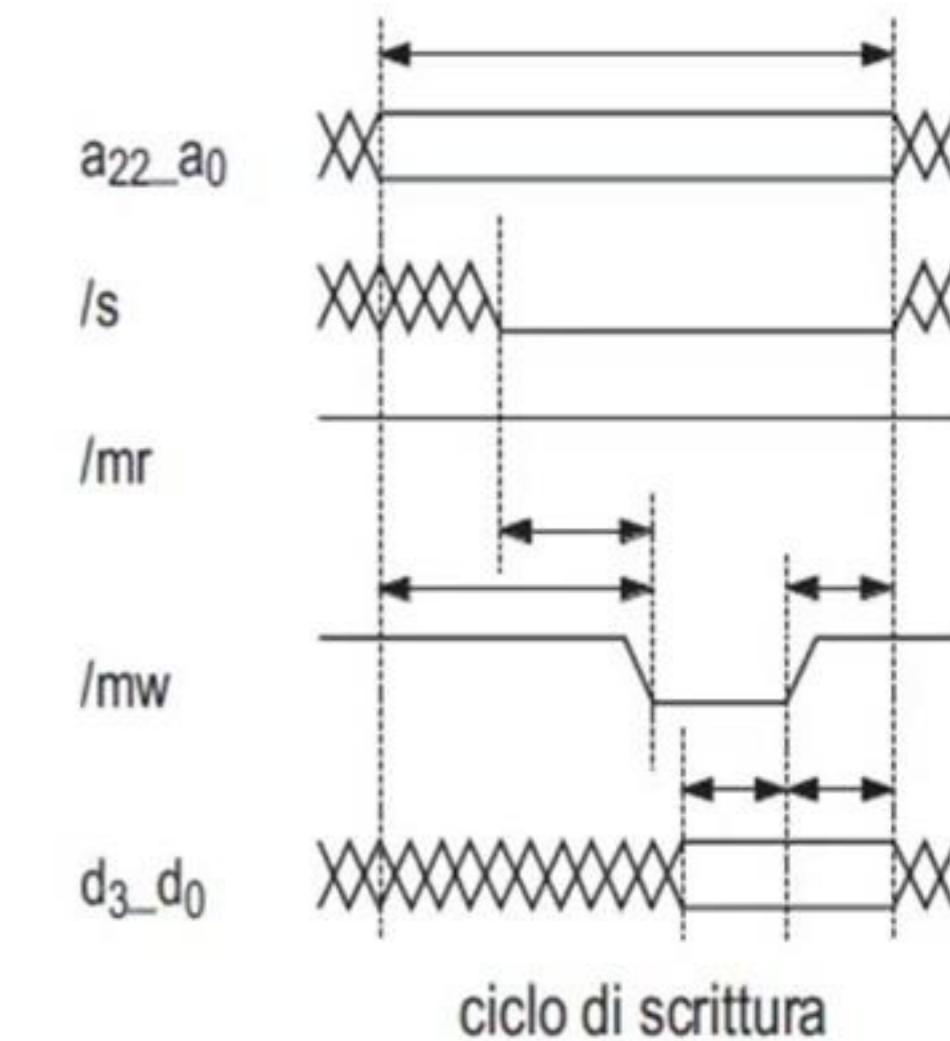
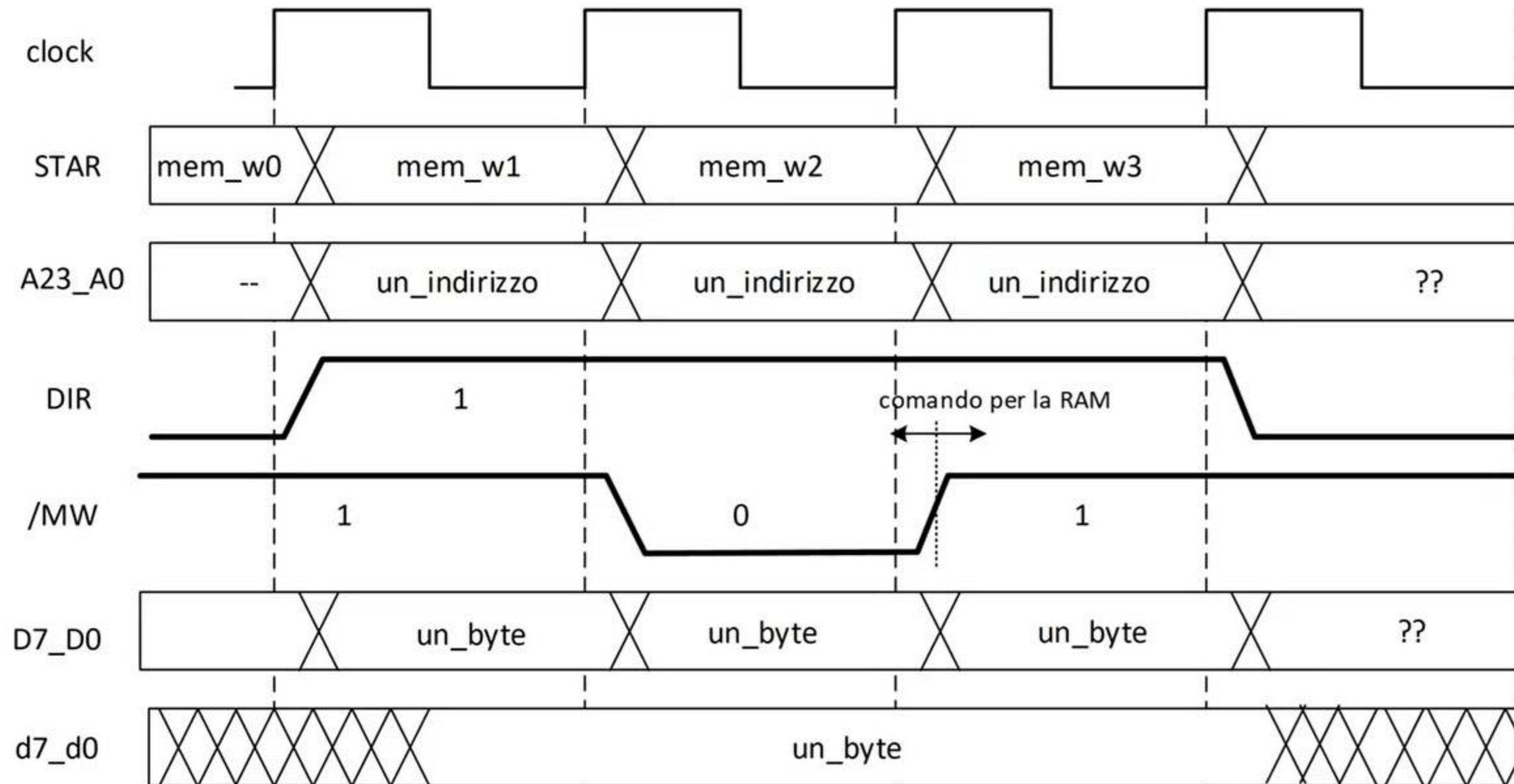
Temporizzazione delle RAM statiche - scrittura

- La **scrittura è distruttiva** (i D-latch sono in trasparenza)
- Devo **attendere che /s e gli indirizzi siano stabili** prima di portare giù **/mw**
- I dati devono **essere corretti a cavallo del fronte di salita di /mw**
 - Corrisponde (con un minimo di ritardo dovuto alla rete combinatoria ed al demultiplexer), al fronte di discesa dell'ingresso *c* dei D-latch



Ciclo di scrittura in memoria – μ -istruzioni

```
mem_w0: begin A23_A0<=un_indirizzo; D7_D0<=un_byte; DIR<=1; STAR<=mem_w1; end
mem_w1: begin MW_<=0; STAR<=mem_w2; end
mem_w2: begin MW_<=1; STAR<=mem_w3; end
mem_w3: begin DIR<=0; ..... ; end
```



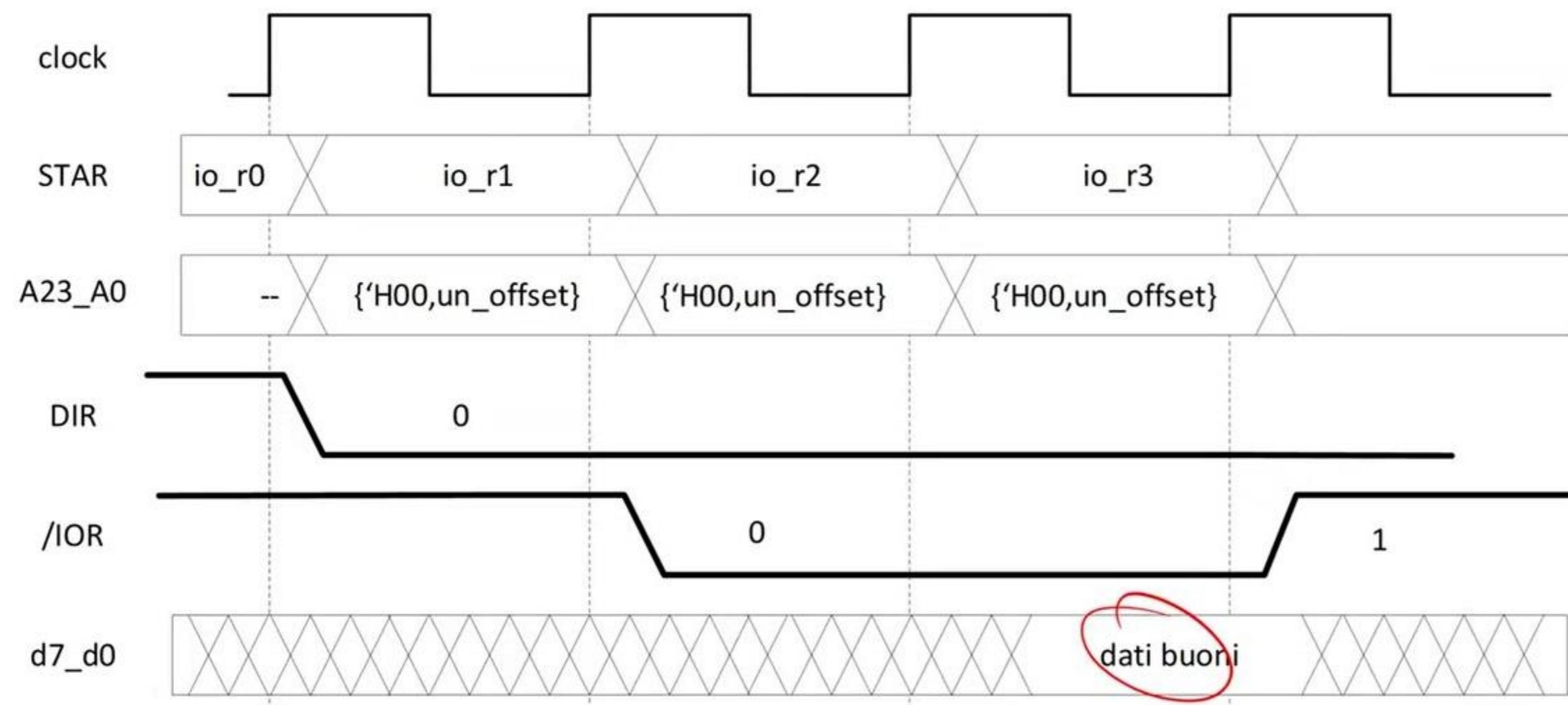
Lettura nello spazio di I/O

SPAZIO DI I/O

- a) Indirizzi a 16 bit
- b) Si usa /ior invece che /mr

- Registri coinvolti : A23_A0, DIR, IOR_

```
io_r0: begin A23_A0<={ 'H00,un_offset}; DIR<=0; STAR<=io_r1; end  
io_r1: begin IOR_<=0; STAR<=io_r2; end  
io_r2: begin STAR<=io_r3; end //stato di wait  
io_r3: begin QUALCHE_REGISTRO<=d7_d0; IOR_<=1; .... ; end
```

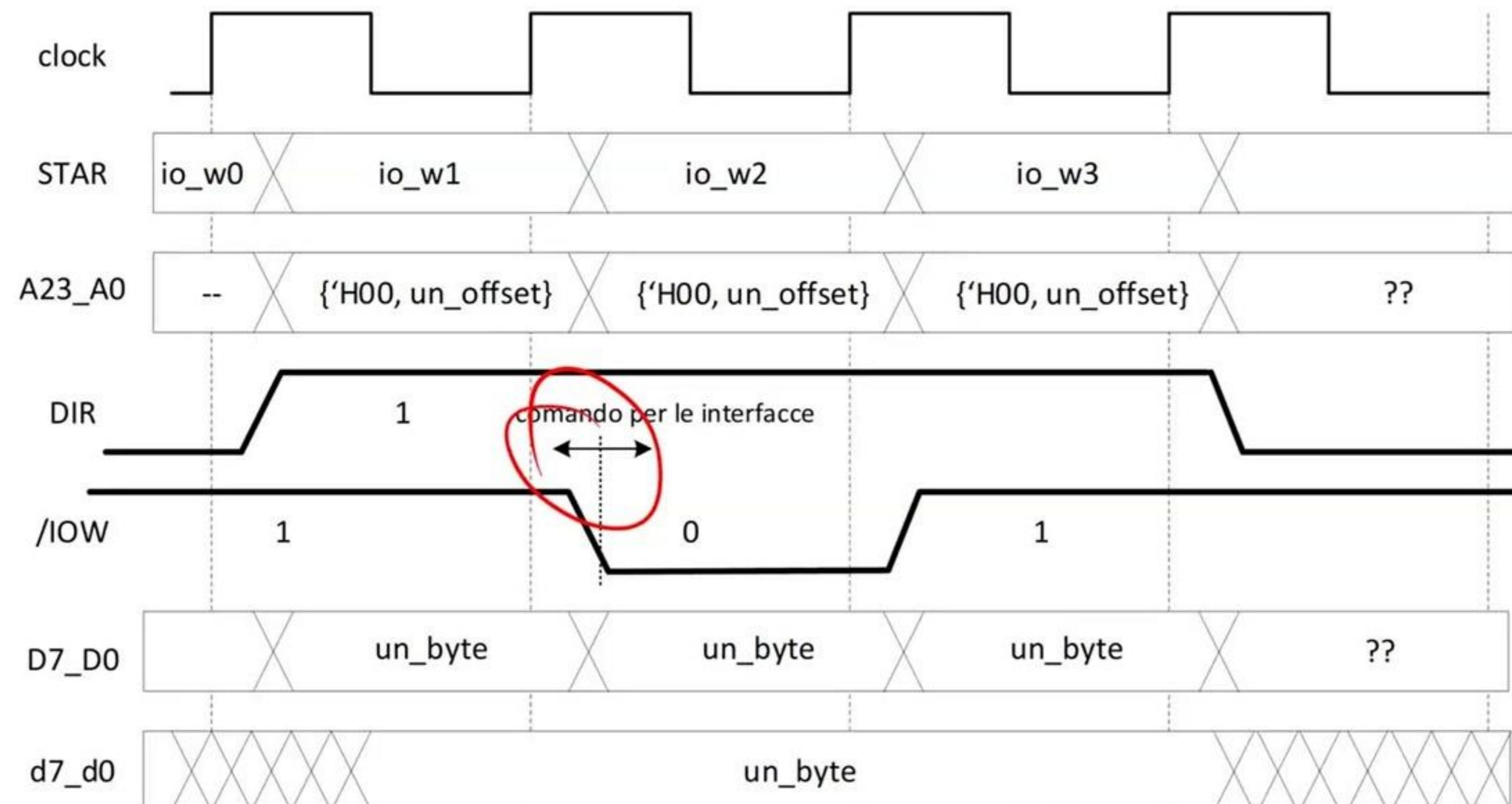


Scrittura nello spazio di I/O

SPAZIO DI I/O

- a) Indirizzi a 16 bit
- b) Si usa /iow invece che /mw

```
io_w0: begin A23_A0<={ 'H00,un_offset}; D7_D0<=un_byte; DIR<=1; STAR<=io_w1; end  
io_w1: begin IOW_<=0; STAR<=io_w2; end  
io_w2: begin IOW_<=1; STAR<=io_w3; end  
io_w3: begin DIR<=0; ..... ; end
```



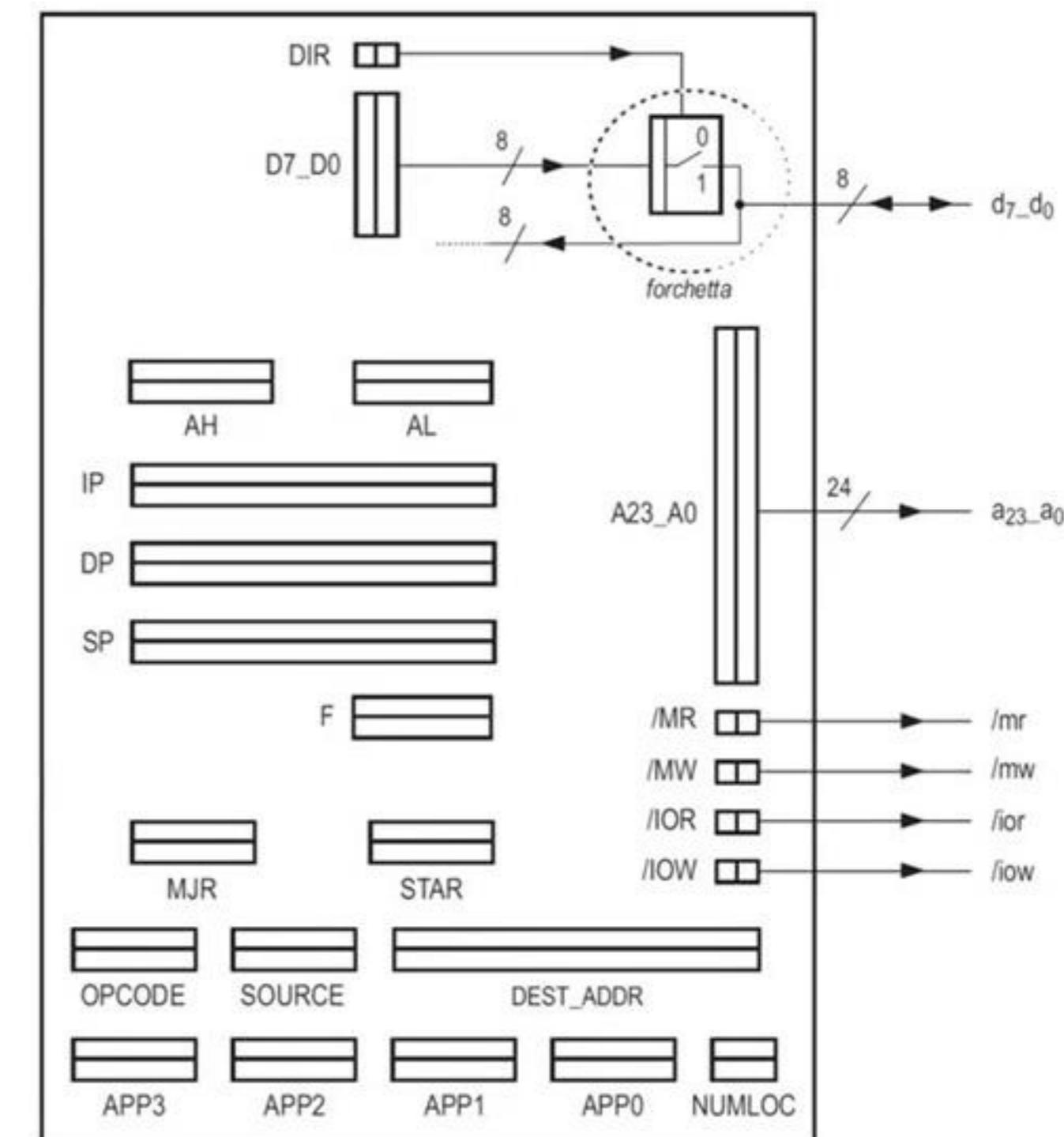
Accessi per più di un byte alla memoria

- Il processore ha bisogno di fare accessi in memoria non solo al byte, ma anche ad operandi a **2 e 3 byte**
 - per prelevare **indirizzi nello spazio di I/O e di memoria**, rispettivamente
 - Occasionalmente anche di 4 byte (non nella parte di corso che vedremo)
- Per fare questo in modo semplice, fa comodo dotarsi di **μ -sottoprogrammi** di lettura/scrittura modulari.
- Utilizziamo
 - il registro MJR, per contenere il μ -indirizzo di ritorno
 - il registro NUMLOC come contatore del numero di byte da leggere/scrivere
 - i registri APP0, APP1, APP2, APP3 per contenere i byte letti/da scrivere

Letture di più di un byte in memoria

```
// MICROPROGRAMMA PRINCIPALE  
Sx: begin ... A23_A0<=un_indirizzo; MJR<=Sx+1; STAR<=readB; end  
Sx+1: begin ... <utilizzo di APP0> end
```

- μ -sottoprogrammi
 - readB (1 byte), readW (2), readM (3), readL (4)
- Parametri di ingresso
 - ~~A23_A0~~ inizializzato con il primo indirizzo in memoria
 - Verrà modificato dai μ -sottoprogrammi
 - **DIR** a zero
- Parametri di uscita
 - APPj (j=0..3): contengono i byte letti



Lettura di più di un byte in memoria

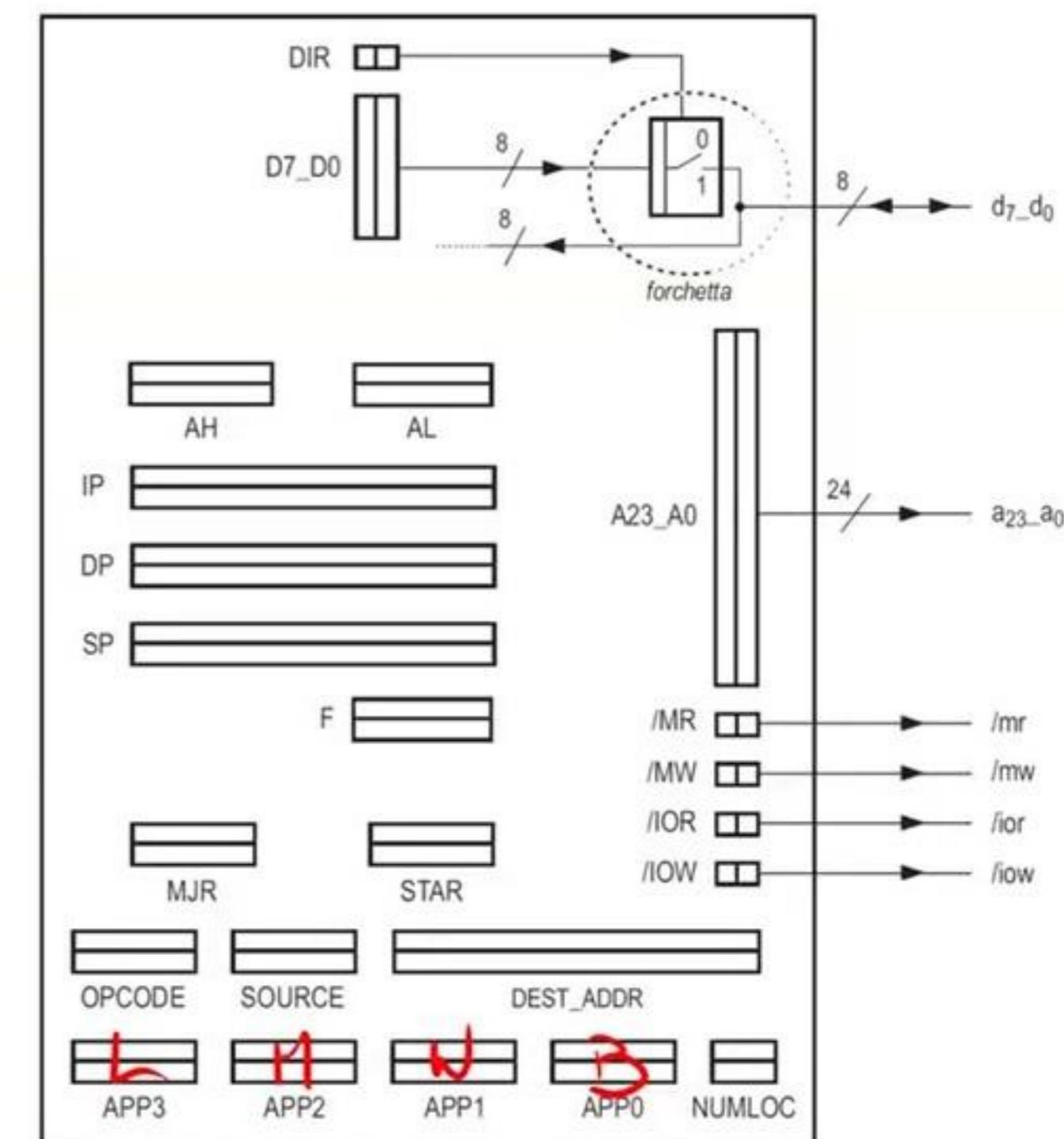
```
// MICROPROGRAMMA PRINCIPALE  
Sx: begin ... A23 A0<=un_indirizzo; MJR<=Sx+1; STAR<=readB; end  
Sx+1: begin ... <utilizzo di APP0> end
```

```
// MICROSOTTOPROGRAMMA PER LETTURE IN MEMORIA  
readB: begin MR_<=0; NUMLOC<=1; STAR<=read0; end  
readW: begin MR_<=0; NUMLOC<=2; STAR<=read0; end  
readM: begin MR_<=0; NUMLOC<=3; STAR<=read0; end  
readL: begin MR_<=0; NUMLOC<=4; STAR<=read0; end  
read0: begin APP0<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;  
       STAR<= (NUMLOC==1) ? read4 : read1; end  
read1: begin APP1<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;  
       STAR<= (NUMLOC==1) ? read4 : read2; end  
read2: begin APP2<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;  
       STAR<= (NUMLOC==1) ? read4 : read3; end  
read3: begin APP3<=d7_d0; A23_A0<=A23_A0+1; STAR<=read4; end  
read4: begin MR_<=1; STAR<=MJR; end
```

Scritture di più di un byte in memoria

```
// MICROPROGRAMMA PRINCIPALE  
Sx: begin ... APP1<=dato_16_bit[15:8]; APP0<=dato_16_bit[7:0];  
       A23_A0<=un indirizzo; MJR<=Sx+1; STAR<=writeW; end  
Sx+1: begin ... end
```

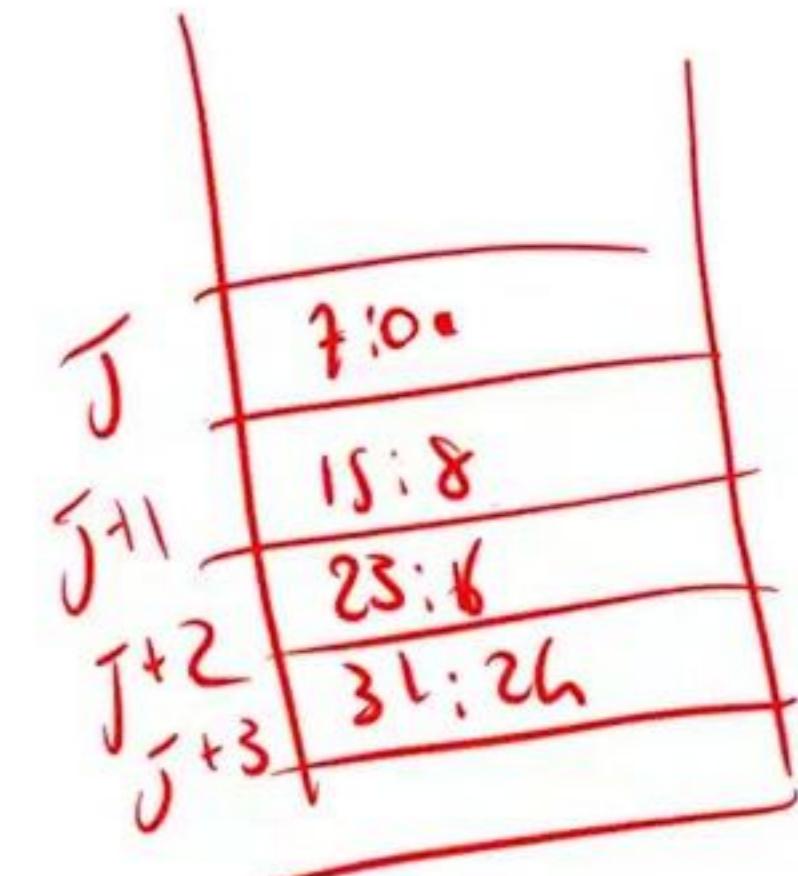
- μ -sottoprogrammi
 - writeB (1 byte), writeW (2), writeM (3), writeL (4)
- Parametri di ingresso
 - ~~A23-PC~~ ~~ADDR~~ inizializzato con il **primo** indirizzo in memoria
 - Verrà modificato dai μ -sottoprogrammi
 - **DIR** a zero
 - **APPj** (j=0..3): contengono i byte da scrivere



Scritture di più di un byte in memoria

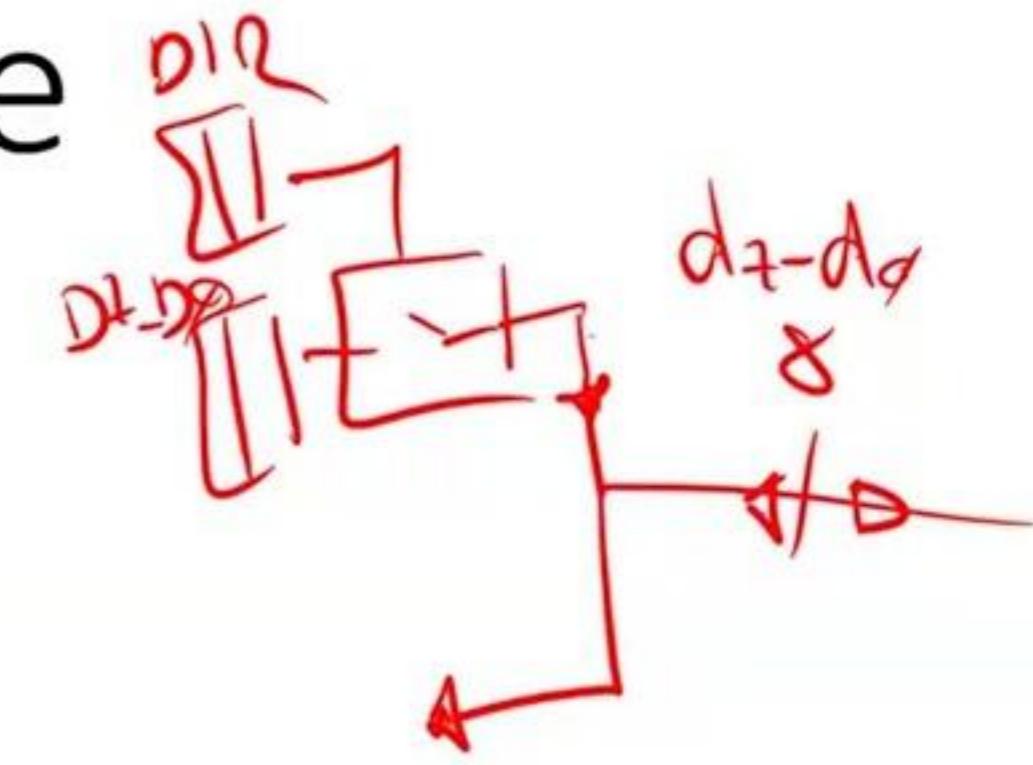
```
// MICROPROGRAMMA PRINCIPALE
Sx: begin ... APP1<=dato_16_bit[15:8]; APP0<=dato_16_bit[7:0];
        A23_A0<=un_indirizzo; MJR<=Sx+1; STAR<=writeW; end
Sx+1: begin ... end

// MICROSOTTOPROGRAMMA PER SCRITTURE IN MEMORIA
writeB: begin D7_D0<=APP0; DIR<=1; NUMLOC<=1; STAR<=write0; end
writeW: begin D7_D0<=APP0; DIR<=1; NUMLOC<=2; STAR<=write0; end
writeM: begin D7_D0<=APP0; DIR<=1; NUMLOC<=3; STAR<=write0; end
writeL: begin D7_D0<=APP0; DIR<=1; NUMLOC<=4; STAR<=write0; end
write0: begin MW_<=0; STAR<=write1; end
write1: begin MW_<=1; STAR<=(NUMLOC==1)?write11:write2; end
write2: begin D7_D0<=APP1; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1; STAR<=write3; end
write3: begin MW_<=0; STAR<=write4; end
write4: begin MW_<=1; STAR<=(NUMLOC==1)?write11:write5; end
write5: begin D7_D0<=APP2; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1; STAR<=write6; end
write6: begin MW_<=0; STAR<=write7; end
write7: begin MW_<=1; STAR<=(NUMLOC==1)?write11:write8; end
write8: begin D7_D0<=APP3; A23_A0<=A23_A0+1; STAR<= write9; end
write9: begin MW_<=0; STAR<= write10; end
write10: begin MW_<=1; STAR<= write11; end
write11: begin DIR<=0; STAR<=MJR; end
```



Descrizione in Verilog del processore

```
module Processore(d7_d0,a23_a0,mr_,mw_,ior_,iow_,clock,reset_);  
    input  
    inout [7:0] d7_d0;  
    output [23:0] a23_a0;  
    output mr_,mw_;  
    output ior_,iow_;  
  
    // REGISTRI OPERATIVI DI SUPPORTO ALLE VARIABILI DI USCITA E ALLE  
    // VARIABILI BIDIREZIONALI E CONNESSIONE DELLE VARIABILI AI REGISTRI  
    reg DIR;  
    reg [7:0] D7_D0;  
    reg [23:0] A23_A0;  
    reg MR_,MW_,IOR_,IOW_;  
    assign mr_=MR_;  
    assign mw_=MW_;  
    assign ior_=IOR_;  
    assign iow_=IOW_;  
    assign a23_a0=A23_A0;  
    assign d7_d0=(DIR==1)?D7_D0:'HZZ; //FORCHETTA
```



Descrizione in Verilog del processore (2)

```
// REGISTRI OPERATIVI INTERNI
reg [2:0] NUMLOC;
reg [7:0] AL,AH,F,OPCODE,SOURCE,APP3,APP2,APP1,APP0;
reg [23:0] DP,IP,SP,DEST_ADDR;

// REGISTRO DI STATO, REGISTRO MJR E CODIFICA DEGLI STATI INTERNI
reg [6:0] STAR,MJR;
parameter fetch0=0, .... write11=85;

// RETI COMBINATORIE NON STANDARD
function valid_fetch;    1: opcode bco
  input [7:0] opcode;   0: m bco
  ...
endfunction

function [7:0] first_execution_state;
  input [7:0] opcode;
  ...
endfunction

// REGISTRO DI STATO, REGISTRO MJR E CODIFICA DEGLI STATI INTERNI
reg [6:0] STAR,MJR;
parameter fetch0=0, .... write11=85;

// RETI COMBINATORIE NON STANDARD
function jmp_condition; J ✓
  input [7:0] opcode; Jt ✓
  input [7:0] flag; Jf ✗
  ...
endfunction

function [7:0] alu_result; J ✓
  input [7:0] opcode,operando1,operando2;
  ...
endfunction

function [3:0] alu_flag; J ✓
  input [7:0] opcode,operando1,operando2;
  ...
endfunction
```

Descrizione in Verilog del processore (3)

```
// ALTRI MNEMONICI
parameter [2:0] F0='B000, F1='B001, F2='B010, F3='B011,
                F4='B100, F5='B101, F6='B110, F7='B111;

//-----
// AL RESET_INIZIALE
always @ (reset_==0) #1 begin
    IP<='HFF0000; F<='H00; DIR<=0;
    MR_<=1; MW_<=1; IOR_<=1; IOW_<=1;
    STAR<=fetch0; end
    •

//-----
// ALL'ARRIVO DEI SEGNALI DI SINCRONIZZAZIONE
always @ (posedge clock) if (reset_==1) #3
    casex (STAR)
```

Descrizione in Verilog del processore (4)

```
// FASE DI CHIAMATA  
fetch0: begin A23_A0<=IP; IP<=IP+1; MJR<=fetch1; STAR<=readB; end  
fetch1: begin OPCODE<=APP0; STAR<=fetch2; end  
fetch2: begin MJR<= (OPCODE[7:5]==F0) ? fetchEnd :  
           (OPCODE[7:5]==F1) ? fetchEnd :  
           (OPCODE[7:5]==F2) ? fetchF2_0 :  
           (OPCODE[7:5]==F3) ? fetchF3_0 :  
           (OPCODE[7:5]==F4) ? fetchF4_0 :  
           (OPCODE[7:5]==F5) ? fetchF5_0 :  
           (OPCODE[7:5]==F6) ? fetchF6_0 :  
           /* default */      fetchF7_0;  
           STAR<=(valid_fetch(OPCODE)==1) ? fetch3:nvi; end  
fetch3: begin STAR<=MJR; end  
[...]  
fetchFx_y: begin [...] STAR<=fetchEnd; end  
[...]  
// TERMINAZIONE CON BLOCCO PER ISTRUZIONE NON VALIDA  
nvi: begin STAR<=nvi; end  
  
// TERMINAZIONE REGOLARE CON PASSAGGIO ALLA FASE DI ESECUZIONE  
fetchEnd: begin MJR<=first_execution_state(OPCODE); STAR<=fetchEnd1; end  
fetchEnd1: begin STAR<=MJR; end
```

Annotations:

- Red circles highlight conditions: `A23_A0<=IP`, `IP<=IP+1`, `MJR<=fetch1`, `STAR<=readB`, `OPCODE<=APP0`, `STAR<=fetch2`, `OPCODE[7:5]==F0`, `OPCODE[7:5]==F1`, `OPCODE[7:5]==F2`, `OPCODE[7:5]==F3`, `OPCODE[7:5]==F4`, `OPCODE[7:5]==F5`, `OPCODE[7:5]==F6`, `valid_fetch(OPCODE)==1`, `STAR<=MJR`, `STAR<=fetchEnd`, `STAR<=nvi`, `MJR<=first_execution_state(OPCODE)`, `STAR<=fetchEnd1`, `STAR<=MJR`.
- Red brackets group sections:
 - Fetch0, Fetch1, Fetch2, FetchEnd, FetchEnd1 are grouped by a large bracket on the right labeled `read & code`.
 - Fetch3 is grouped by a bracket on the right labeled `specchio termo`.
 - FetchFx_y is grouped by a bracket on the right labeled `valid`.
 - nvi is grouped by a bracket on the right labeled `exe`.

La slide 65 non l'ho trovata :(

Porzioni di fase di fetch specifiche dei vari formati

OpCode

F	Byte	OPCODE	SOURCE	DEST_ADDR
F0	1	readB @ IP	--	--
F1	1	readB @ IP	--	--
F2	1	readB @ IP	readB @ DP	--
F3	1	readB @ IP	--	DP
F4	2	readB @ IP	readB @ IP	--
F5	4	readB @ IP	readM @ IP, readB	--
F6	4	readB @ IP	--	readM @ IP
F7	4	readB @ IP	--	readM @ IP

Porzioni di fase di fetch specifiche dei vari formati

Formato F2:

```
fetchF2_0: begin A23 A0<=DP; MJR<=fetchF2_1;
              STAR<=readB; end
```

```
fetchF2_1: begin SOURCE<=APP0; STAR<=fetchEnd; end
```

Formato F3:

```
fetchF3_0: begin DEST_ADDR<=DP; STAR<=fetchEnd; end
```

Formato F4:

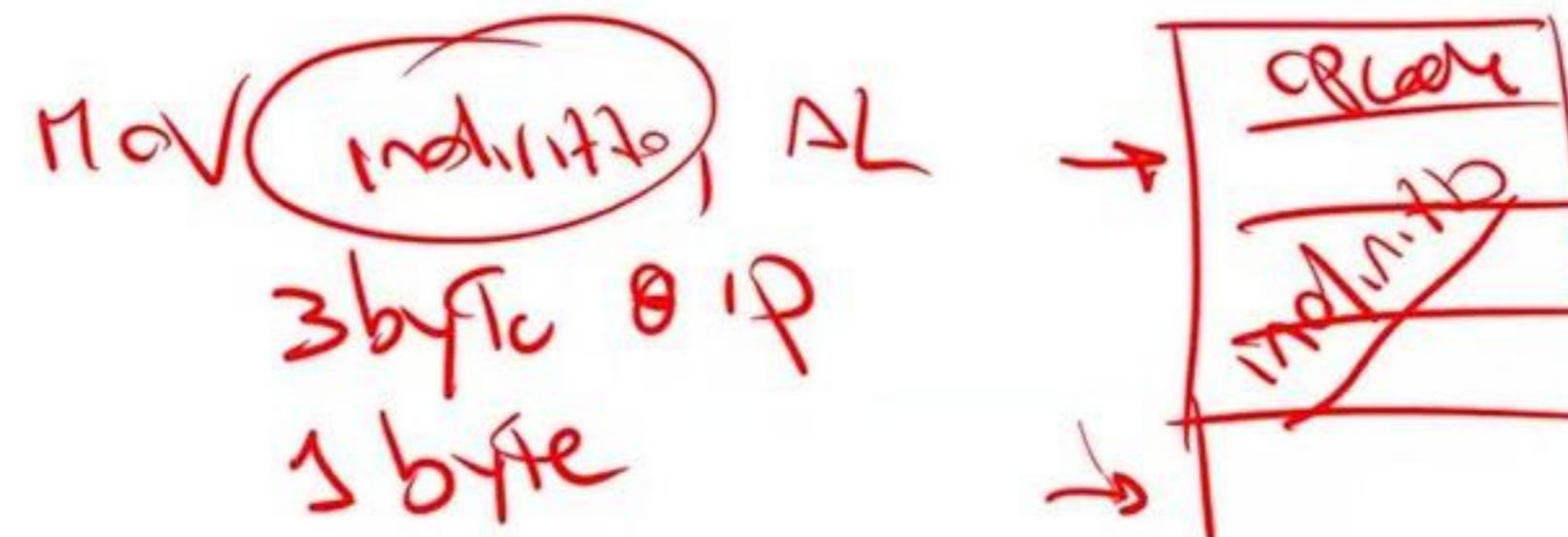
```
fetchF4_0: begin A23 A0<=IP; IP<=IP+1;
              MJR<=fetchF4_1; STAR<=readB; end
```

```
fetchF4_1: begin SOURCE<=APP0; STAR<=fetchEnd; end
```

F	Byte	OPCODE	SOURCE	DEST_ADDR
F0	1	readB @ IP	--	--
F1	1	readB @ IP	--	--
F2	1	readB @ IP	readB @ DP	--
F3	1	readB @ IP	--	DP
F4	2	readB @ IP	readB @ IP	--
F5	4	readB @ IP	readM @ IP, readB	--
F6	4	readB @ IP	--	readM @ IP
F7	4	readB @ IP	--	readM @ IP



Porzioni di fase di fetch specifiche dei vari formati



Formato F5:

```

fetchF5_0: begin A23 A0<=IP; IP<=IP+3; MJR<=fetchF5_1;
            STAR<=readM; end
fetchF5_1: begin A23 A0<={APP2,APP1,APP0};
            MJR<=fetchF5_2; STAR<=readB; end
fetchF5_2: begin SOURCE<=APP0; STAR<=fetchEnd; end

```

F	Byte	OPCODE	SOURCE	DEST_ADDR
F0	1	readB @ IP	--	--
F1	1	readB @ IP	--	--
F2	1	readB @ IP	readB @ DP	--
F3	1	readB @ IP	--	DP
F4	2	readB @ IP	readB @ IP	--
F5	4	readB @ IP	readM @ IP, readB	--
F6	4	readB @ IP	--	readM @ IP
F7	4	readB @ IP	--	readM @ IP

Formato F6/F7:

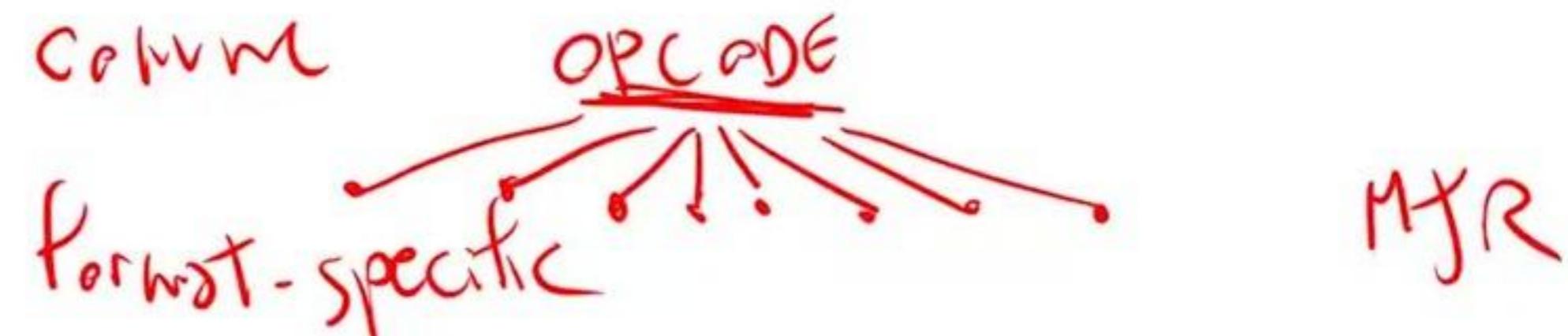
```

fetchF6_0: begin A23 A0<=IP; IP<=IP+3; MJR<=fetchF6_1; STAR<=readM; end
fetchF6_1: begin DEST_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; end
            .
            .
fetchF7_0: begin A23 A0<=IP; IP<=IP+3; MJR<=fetchF7_1; STAR<=readM; end
fetchF7_1: begin DEST_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; end

```

F6 MOV AL, [indirizzo]
F7 JMP [indirizzo]

Fine della fase di fetch



codice OPCODE → first exec.
 style

- All'uscita della fase di fetch:
 - OPCODE contiene il codice operativo dell'istruzione
 - Se l'istruzione ha un operando sorgente **immediato o in memoria**, questo sta in SOURCE (indipendentemente dalla modalità di indirizzamento usata dal programmatore nell'istruzione Assembler);
 - Se l'istruzione ha un operando destinatario **in memoria**, il suo indirizzo sta in DEST_ADDR (indipendentemente dalla modalità di indirizzamento usata dal programmatore nell'istruzione Assembler).
 - IP è stato incrementato del numero di byte necessario, e punta alla **prossima** istruzione da prelevare.

La fase di esecuzione

- Sarà abbastanza semplice, perché:
- una parte della complessità è stata gestita in fase di fetch
- la maggior parte delle istruzioni “complesse” (quelle logico-aritmetiche) sono implementate tramite **reti combinatorie**, e quindi si riducono ad un semplice assegnamento a registro

Fase di esecuzione (1)

```
//----- istruzione NOP -----  
nop: begin STAR<=fetch0; end  
  
//----- istruzione HLT -----  
hlt: begin STAR<=hlt; end  
  
//----- istruzione MOV AL,AH -----  
ALtoAH: begin AH<=AL; STAR<=fetch0; end 10  
  
//----- istruzione MOV AH,AL -----  
AHToAL: begin AL<=AH; STAR<=fetch0; end  
  
//----- istruzione INC DP -----  
incDP: begin DP<=DP+1; STAR<=fetch0; end
```

Fase di esecuzione (2)

//----- istruzioni *P2* MOV •(DP), AL -----
//
//
ldAL: begin AL<=SOURCE; STAR<=fetch0; end

//----- istruzioni •MOV (DP), AH -----
//
//
ldAH: begin AH<=SOURCE; STAR<=fetch0; end

//----- istruzioni |MOV AL, (DP) | DEST_ADDR
//
|MOV AL, indirizzo |
storeAL: begin A23_A0<=DEST_ADDR; APP0<=AL; MJR<=fetch0; STAR<=writeB; end ||

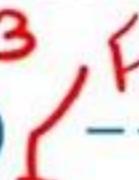
//----- istruzioni MOV AH, (DP) -----
//
MOV AH, indirizzo
storeAH: begin A23_A0<=DEST_ADDR; APP0<=AH; MJR<=fetch0; STAR<=writeB; end *s*

SOURCE



X12 (DP), AL

DEST_ADDR



Formato F1 (001)

IN offset, AL	00100000	offset	
OUT AL, offset	00100001	offset	
MOV \$operando, DP	00100010	operando	
MOV \$operando, SP	00100011	operando	
MOV indirizzo, DP	00100100	indirizzo	
MOV DP, indirizzo	00100101	indirizzo	

- Raggruppa tutte le istruzioni che non possono essere classificate nei precedenti formati
 - Istruzioni di I/O
 - operando indirizzo a 16 bit, sorgente (IN) o destinatario (OUT)
 - MOV con uno dei registri a 24 bit (DP o SP)
 - Operando a 24 bit sorgente, sia immediato che diretto, o destinatario
- Le azioni per procurarsi gli operandi sono diverse da un'istruzione all'altra
 - Meglio fare una fase di fetch «scarna» in cui prelievo solo l'opcode
 - Gestiremo gli operandi successivamente in fase di esecuzione
 - Poco pulito dal punto di vista concettuale, ma molto più semplice

Fase di esecuzione (3)

```

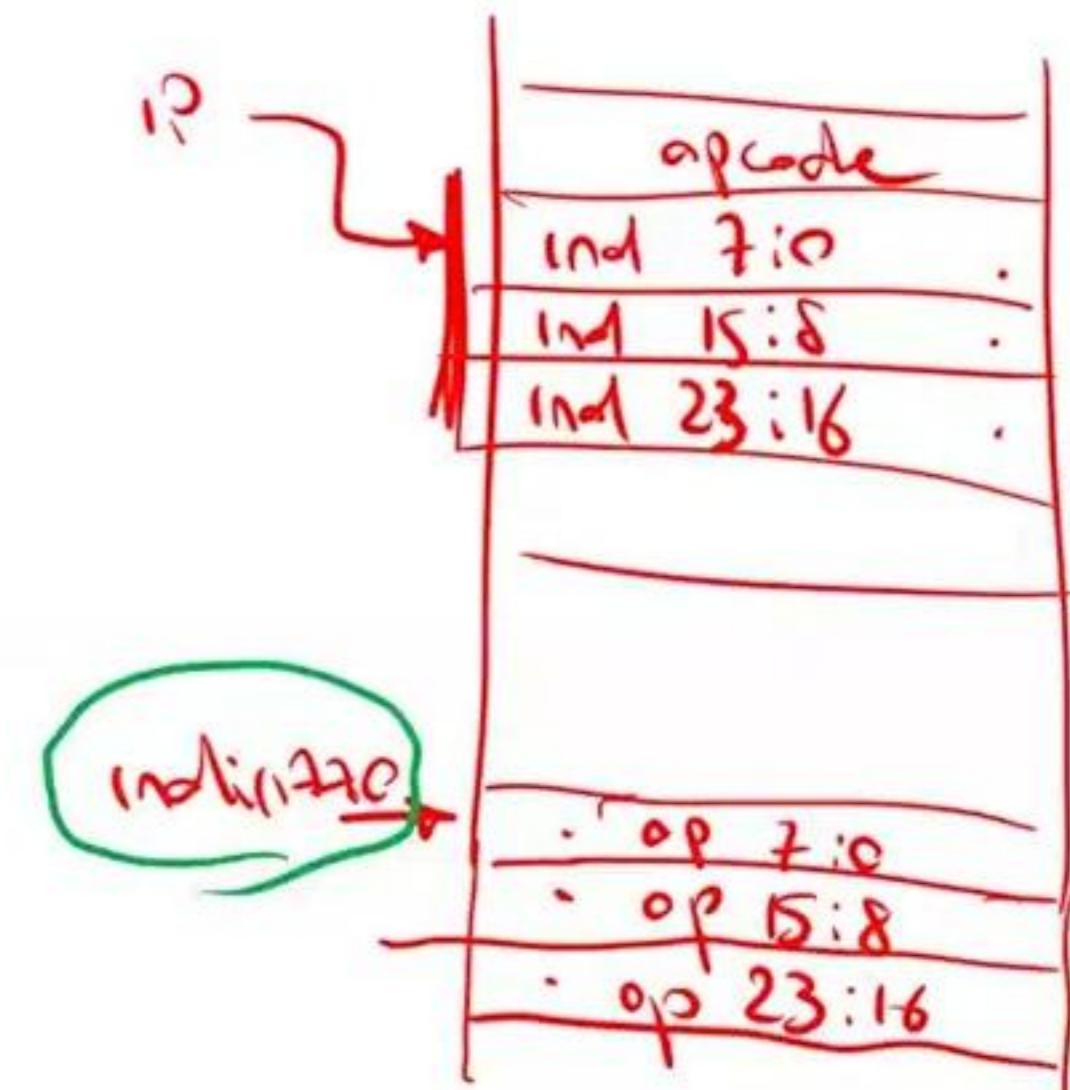
//----- istruzione MOV $operando, SP -----
ldSP: begin A23 A0<=IP; IP<=IP+3; MJR<=ldSP1; STAR<=readM; end
ldSP1: begin SP<={APP2,APP1,APP0}; STAR<=fetch0; end
26bit 3 ↓

//----- istruzione MOV $operando, DP -----
ldimmDP: begin A23 A0<=IP; IP<=IP+3; MJR<=ldimmDP1; STAR<=readM; end
ldimmDP1: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end
24bit 3 ↓

//----- istruzione MOV indirizzo, DP -----
lddirDP: begin A23 A0<=IP; IP<=IP+3; MJR<=lddirDP1; STAR<=readM; end
lddirDP1: begin A23 A0<={APP2,APP1,APP0}; MJR<=lddirDP2; STAR<=readM; end
lddirDP2: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end
3 ↓ ↑

//----- istruzione MOV DP, indirizzo -----
storeDP: begin A23 A0<=IP; IP<=IP+3; MJR<=storeDP1; STAR<=readM; end
storeDP1: begin A23 A0<={DEST_ADR}; {APP2,APP1,APP0}<=DP;
          MJR<=fetch0; STAR<=writeM; end
1 ↓

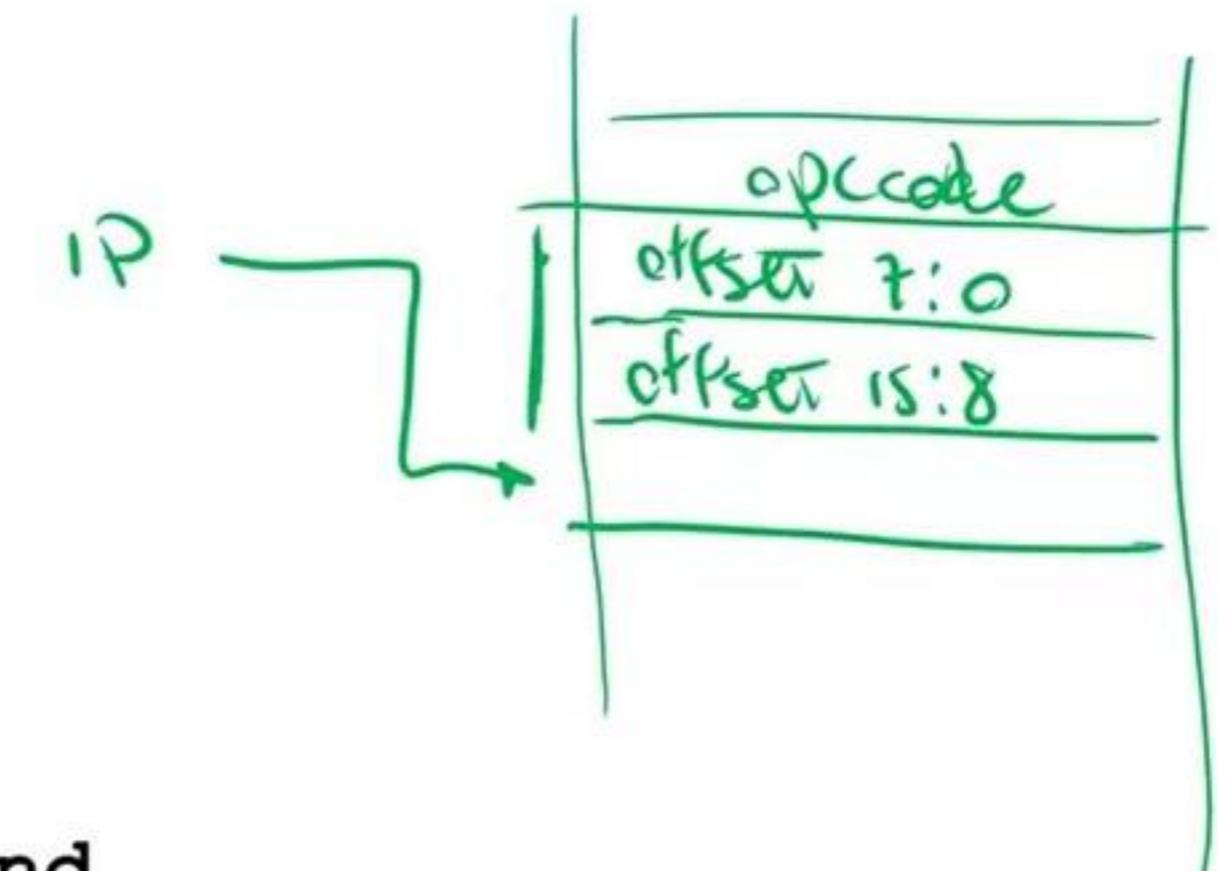
```



Fase di esecuzione (4)

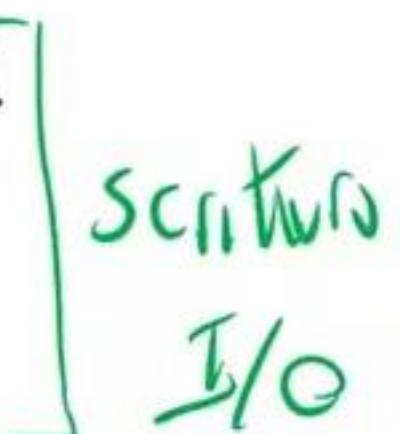
//----- istruzione IN offset, AL -----

```
in: begin A23_A0<=IP; IP<=IP+2; MJR<=in1; STAR<=readW; end
in1: begin A23_A0<={ 'H00,APP1,APP0}; STAR<=in2; end
in2: begin IOR_<=0; STAR<=in3; end
in3: begin AL<=d7_d0; IOR_<=1; STAR<=fetch0; end
```



//----- istruzione OUT AL, offset -----

```
out: begin A23_A0<=IP; IP<=IP+2; MJR<=out1; STAR<=readW; end
out1: begin A23_A0<={ 'H00,APP1,APP0}; D7_D0<=AL; DIR<=1; STAR<=out2; end
out2: begin IOW_<=0; STAR<=out3; end
out3: begin IOW_<=1; STAR<=out4; end
out4: begin DIR<=0; STAR<=fetch0; end
```

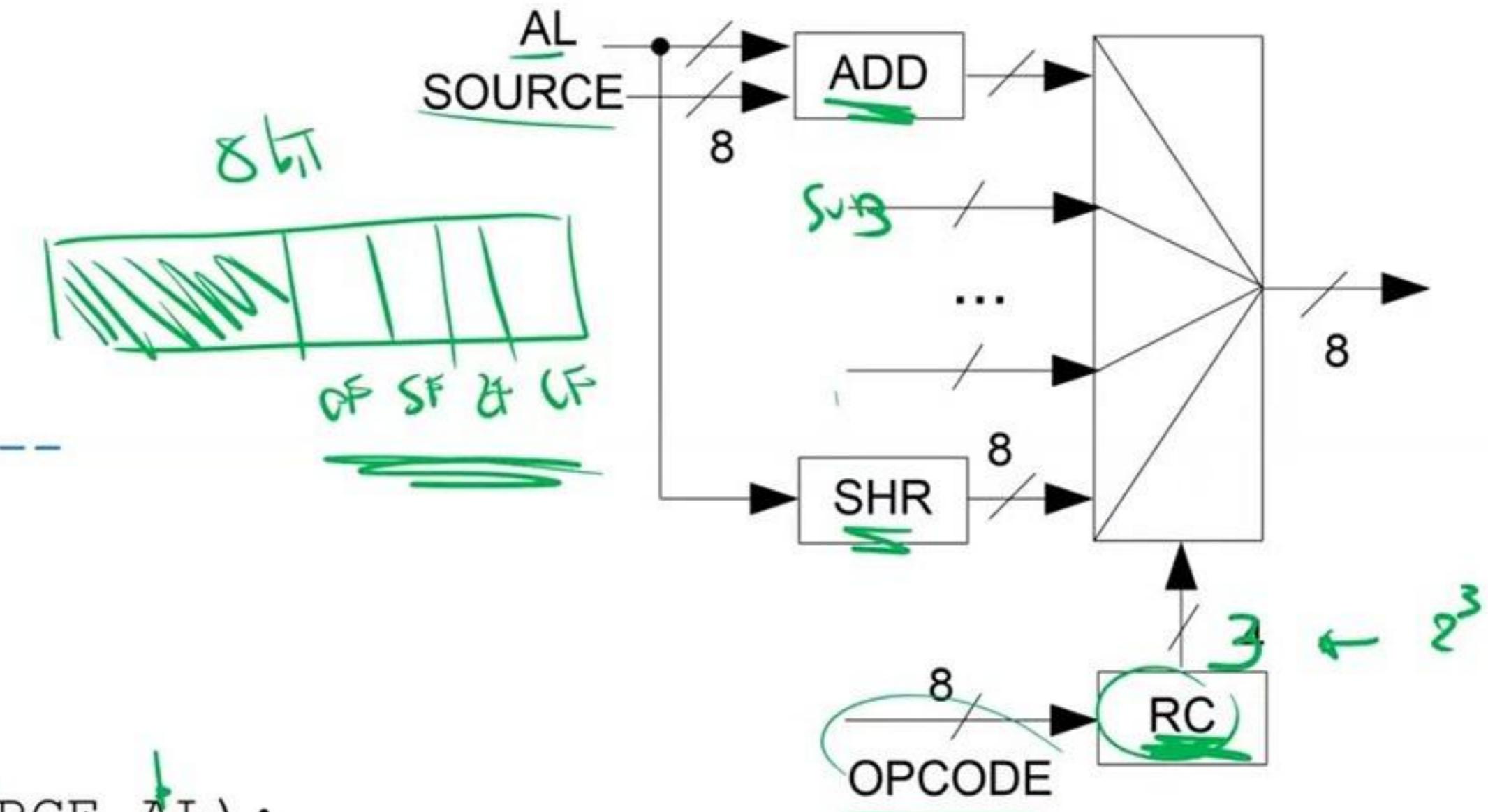


$T \geq T_{\text{idle}}$

Fase di esecuzione (5)

//----- istruzioni ADD (DP), AL -----
 //
 //
 //
 //
 aluAL: begin AL<=alu_result(OPCODE, SOURCE, AL);
 F<={F[7:4], alu_flag(OPCODE, SOURCE, AL)}; STAR<=fetch0; end

//----- istruzioni ADD (DP), AH -----
 //
 //
 //
 //
 aluAH: begin AH<=alu_result(OPCODE, SOURCE, AH);
 F<={F[7:4], alu_flag(OPCODE, SOURCE, AH)}; STAR<=fetch0; end



Fase di esecuzione (6)

//----- istruzioni JMP indirizzo -----

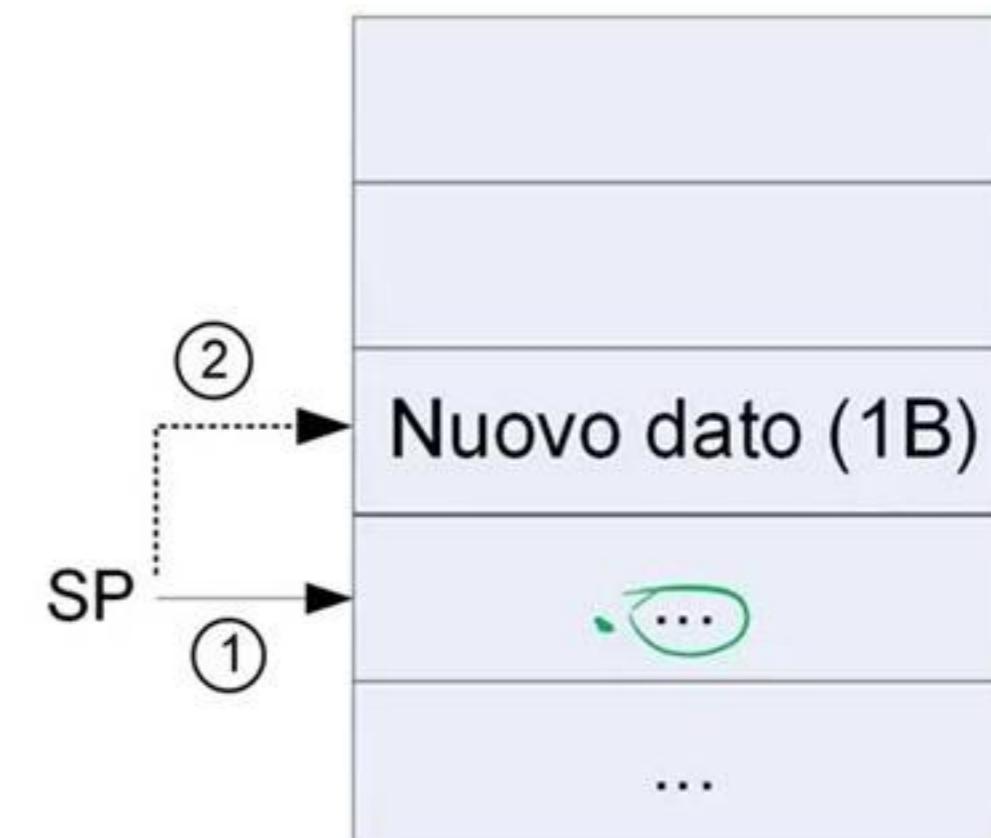
// JA indirizzo
// JAE indirizzo
// [...]
// JZ indirizzo

jmp: **begin** IP<=(jmp_condition (OPCODE, F)==1) ?DEST_ADDR:IP; STAR<=fetch0; **end**

JMP - *JZ F=1*

F
JZ F=1

Fase esecuzione (7)

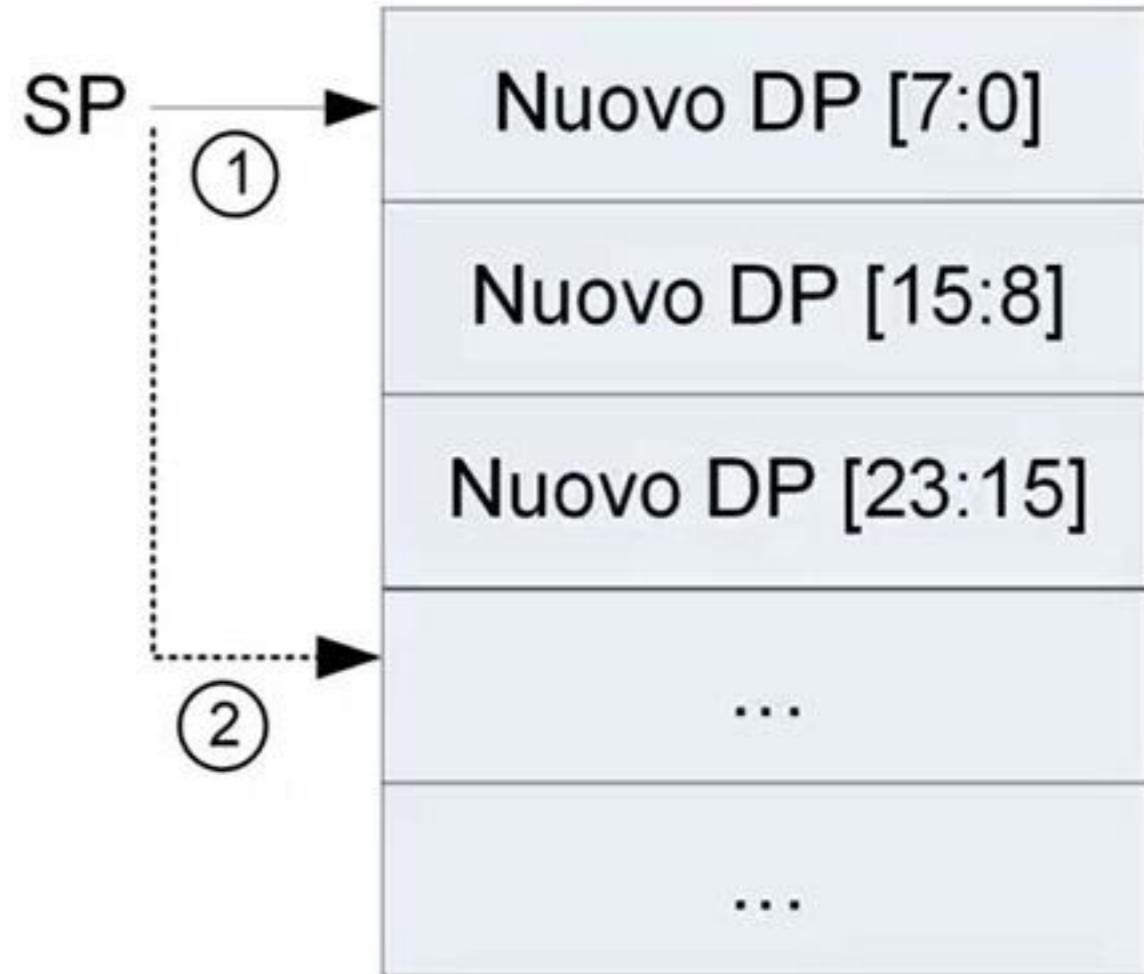


```
//----- istruzione PUSH AL -----  
pushAL: begin A23_A0<=SP-1; SP<=SP-1; APP0<=AL; MJR<=fetch0; STAR<=writeB; end  
  
//----- istruzione PUSH AH -----  
pushAH: begin A23_A0<=SP-1; SP<=SP-1; APP0<=AH; MJR<=fetch0; STAR<=writeB; end  
  
//----- istruzione PUSH DP -----  
pushDP: begin A23_A0<=SP-3; SP<=SP-3; {APP2,APP1,APP0}<=DP; MJR<=fetch0;  
STAR<=writeM; end
```

Annotations in green highlight specific fields in the assembly code:

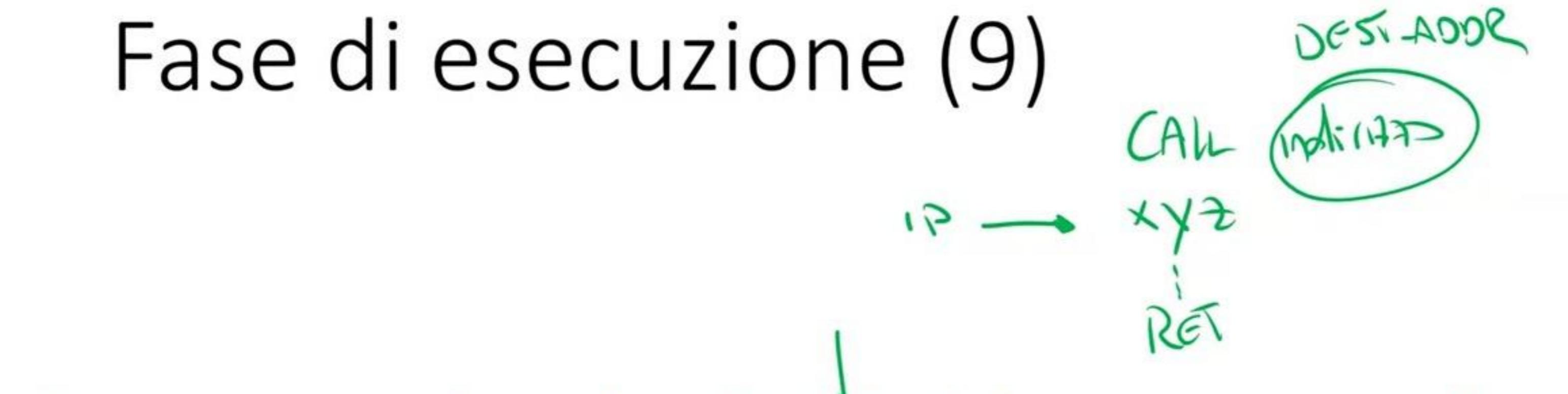
- A circled "AL" is above the first instruction.
- A circled "AH" is above the second instruction.
- A circled "DP" is above the third instruction.
- A circled "3byte" is written vertically next to the third instruction.
- A red arrow points from the circled "AL" to the circled "AL" in the first instruction.
- A red arrow points from the circled "AH" to the circled "AH" in the second instruction.
- A red arrow points from the circled "DP" to the circled "DP" in the third instruction.
- A red arrow points from the circled "3byte" to the circled "3byte" in the third instruction.
- A red arrow points from the circled "MJR" in the first instruction to the circled "MJR" in the second instruction.
- A red arrow points from the circled "STAR" in the first instruction to the circled "STAR" in the second instruction.
- A red arrow points from the circled "writeB" in the first instruction to the circled "writeB" in the second instruction.
- A red arrow points from the circled "APP0" in the first instruction to the circled "APP0" in the third instruction.
- A red arrow points from the circled "APP0" in the second instruction to the circled "APP0" in the third instruction.
- A red arrow points from the circled "APP0" in the first instruction to the circled "APP0" in the third instruction.
- A red arrow points from the circled "APP1" in the first instruction to the circled "APP1" in the third instruction.
- A red arrow points from the circled "APP2" in the first instruction to the circled "APP2" in the third instruction.
- A red arrow points from the circled "MJR" in the first instruction to the circled "MJR" in the third instruction.
- A red arrow points from the circled "STAR" in the first instruction to the circled "STAR" in the third instruction.
- A red arrow points from the circled "writeM" in the first instruction to the circled "writeM" in the third instruction.

Fase di esecuzione (8)



```
//----- istruzione POP AL -----  
popAL: begin A23_A0<=SP; SP<=SP+1; MJR<=popAL1; STAR<=readB; end  
popAL1: begin AL<=APP0; STAR<=fetch0; end  
  
//----- istruzione POP AH -----  
popAH: begin A23_A0<=SP; SP<=SP+1; MJR<=popAH1; STAR<=readB; end  
popAH1: begin AH<=APP0; STAR<=fetch0; end  
  
//----- istruzione POP DP -----  
popDP: begin A23_A0<=SP; SP<=SP+3; MJR<=popDP1; STAR<=readM; end  
popDP1: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end
```

Fase di esecuzione (9)



```

//----- istruzione CALL indirizzo -----
call: begin A23_A0<=SP-3; SP<=SP-3; {APP2,APP1,APP0}<=IP;
      MJR<=call1; STAR<=writeM; end
call1: begin IP<=DEST_ADDR; STAR<=fetch0; end

//----- istruzione RET -----
ret: begin A23_A0<=SP; SP<=SP+3; MJR<=ret1; STAR<=readM; end
ret1: begin IP<={APP2,APP1,APP0}; STAR<=fetch0; end

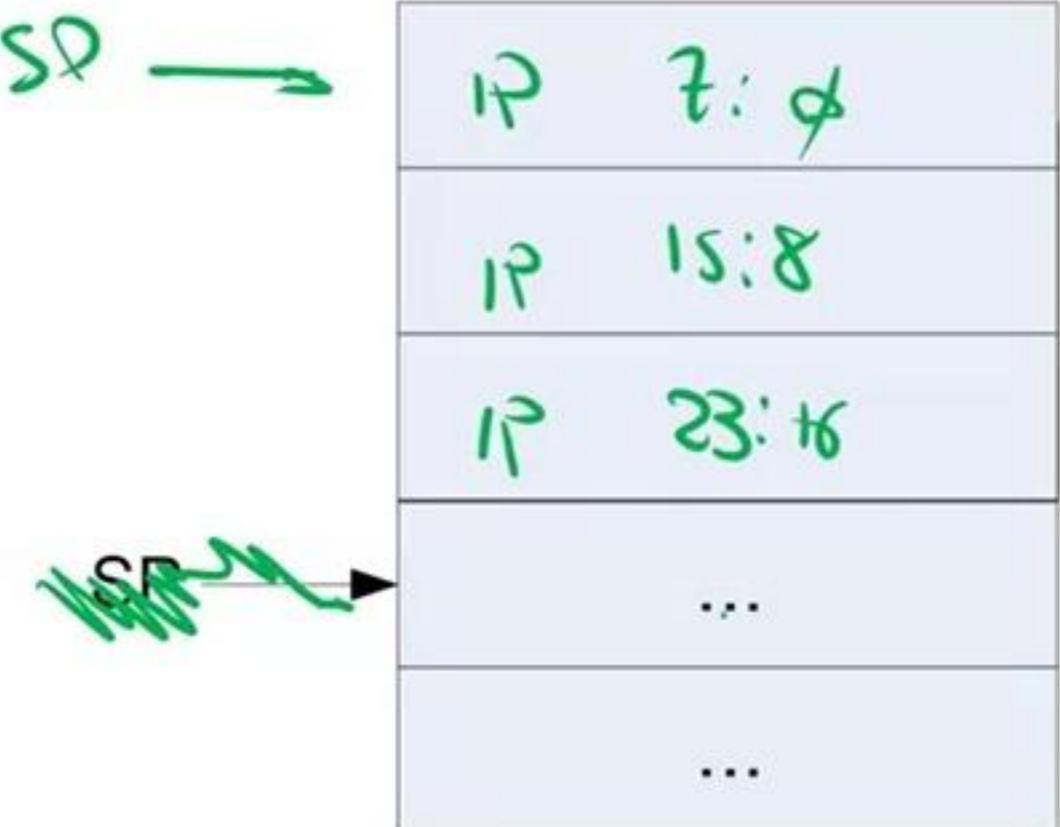
```

cond3

|

wileB

|



Conclusioni

- In ogni stato ci sono **μ -salti al massimo a due vie**
 - La maggior parte dei μ -salti è ad una sola via
 - μ -salti a due quasi esclusivamente nelle sottoliste di lettura/scrittura in memoria
- Le **reti combinatorie** riportate in cima alla descrizione (e.g., ALU, jmp_condition, etc.) contengono solo cose già viste a lezione
- Se vi venisse chiesto di **sintetizzare** questo processore in accordo al modello PO/PC, non ci sarebbero difficoltà concettuali.
- In soli due mesi di corso abbiamo acquisito la capacità di descrivere e sintetizzare **dell'hardware** capace di **eseguire programmi software** arbitrariamente complessi