INDICE

[0]

LOGICA SEQUENZIALE

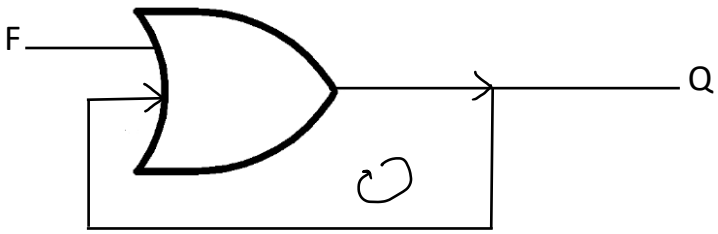
CIRCUITO SEQUENZIALE -> Circuito dove i valori dipendono anche da valori precedenti (quindi STATO INTERNO) e non solo dagli ingressi correnti, a differenza di una rete combinatoria.

Q
- OUTPUT
- STATO INTERNO (capacità di ricordare!)

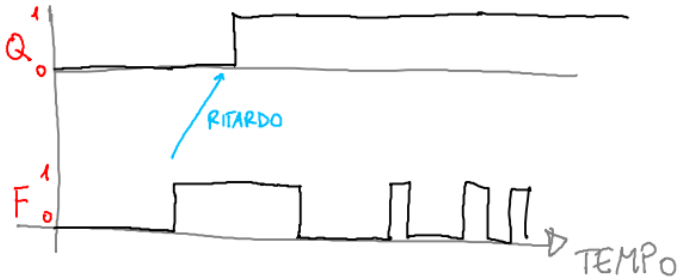
INPUT	STATO	NUOVO STATO
F	Q	Q'
0	0	0
0	1	1
1	0	1
1	1	1

***ESEMPIO luce delle scale**:*

è a 0, appena premo diventa e resta a 1 anche se smetto di premere l'interruttore

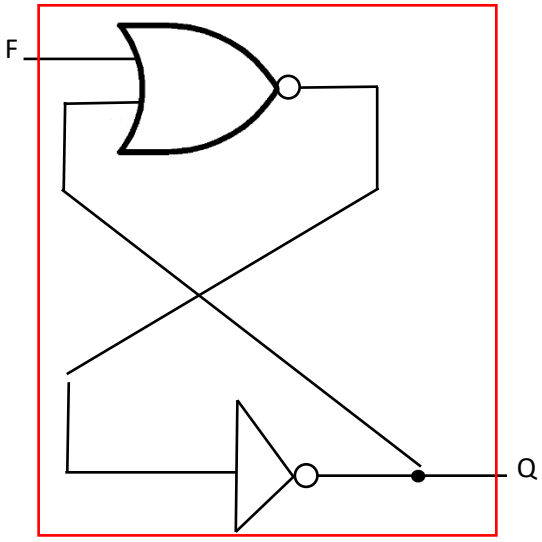


CICLO
FEEDBACK



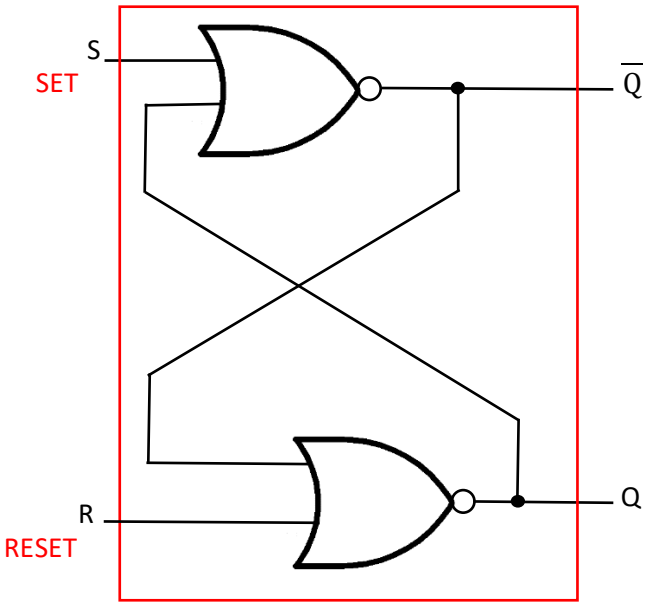
Se stato iniziale 0 e INPUT 0 → sempre STABILITA'

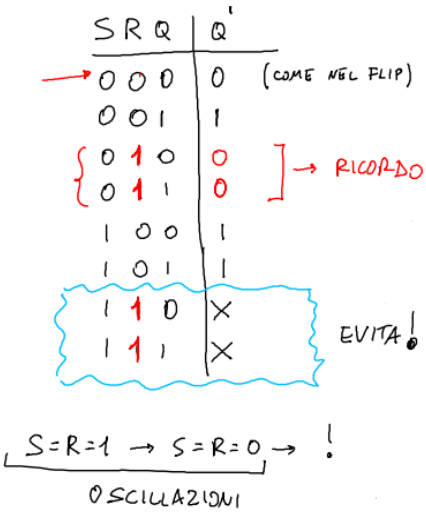
FLIP



DUE NEGAZIONI!

FLIP FLOP SR





IN SINTESI

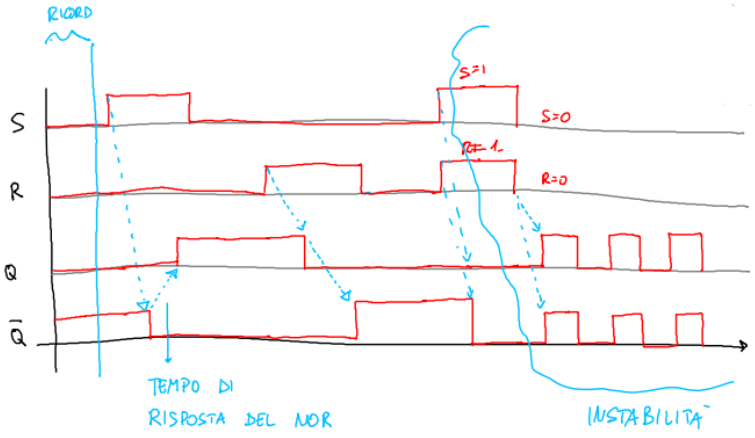
	S	R	$\overline{Q}$
RICORDA	0	0	Q
SET	1	0	1
RESET	0	1	0
INSTABILE	1	1	X

Flip-flop SR [modifica | modifica wikitesto]

È il flip-flop più semplice dal punto di vista circuitale e fu anche il primo ad essere realizzato. La versione attiva alta ha due ingressi S (*Set*) e R (*Reset*, detto anche *Clear*) e due uscite Q e $\bar{Q}$. È una rete sequenziale asincrona che si evolve in accordo alle seguenti specifiche: quando lo stato d'ingresso è s=0 e r=1 il flip-flop si resetta, cioè porta a 0 il valore della variabile d'uscita Q e a 1 la variabile d'uscita $\bar{Q}$; quando lo stato d'ingresso è s=1 e r=0 il flip-flop si setta, cioè porta a 1 il valore della variabile d'uscita Q e a 0 la variabile d'uscita $\bar{Q}$; quando lo stato d'ingresso è s=0 e r=0, il flip-flop conserva, cioè mantiene inalterato il valore di entrambe le variabili d'uscita. La combinazione s=1 ed r=1 non viene utilizzata in quanto instabile (il risultato dipende infatti da quale delle porte che compongono il circuito interno del flip flop viene commutata prima).

Q \ SR	00	01	11	10
0	0	0	X1	1
1	1	0	X1	1

$Q' = S + \overline{R}Q$



RETE SEQUENZIALE

↓

HA STATO INTERNO

RETE SINCRONA

↓

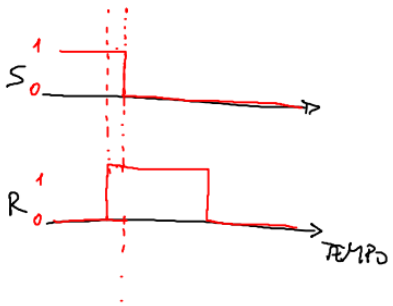
GESTISCE IL TEMPO

Esempi visti fin'ora:

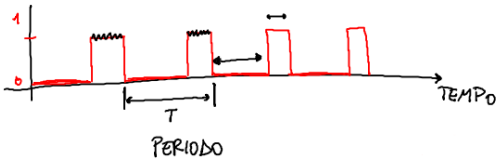
HALF ADDER / FULL ADDER -> COMBINATORIA ASINCRONA

FFSR -> SEQUENZIALE ASINCRONA

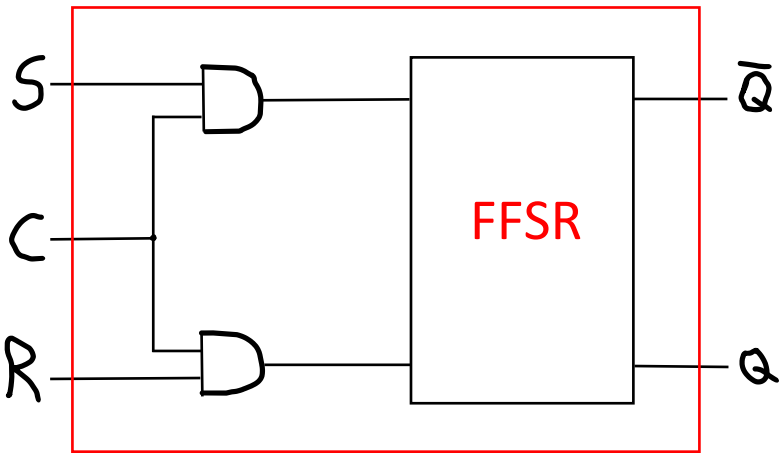
Nei FFSR c'è il PROBLEMA del TEMPO:



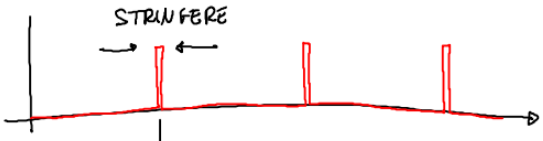
Per risolvere ciò ci "affidiamo" al CLOCK:

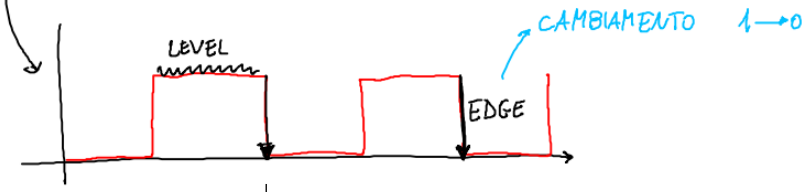


FLIP FLOP SR LEVEL TRIGGERED



CLOCK	INPUT		Q^1	
C	S	R		
0	0	0	Q	NON GUARDO L'INPUT
0	0	1	Q	
0	1	0	Q	
0	1	1	Q	
1	0	0	Q	SOLITO COMPORTAM
1	0	1	0	
1	1	0	1	
1	1	1	X	

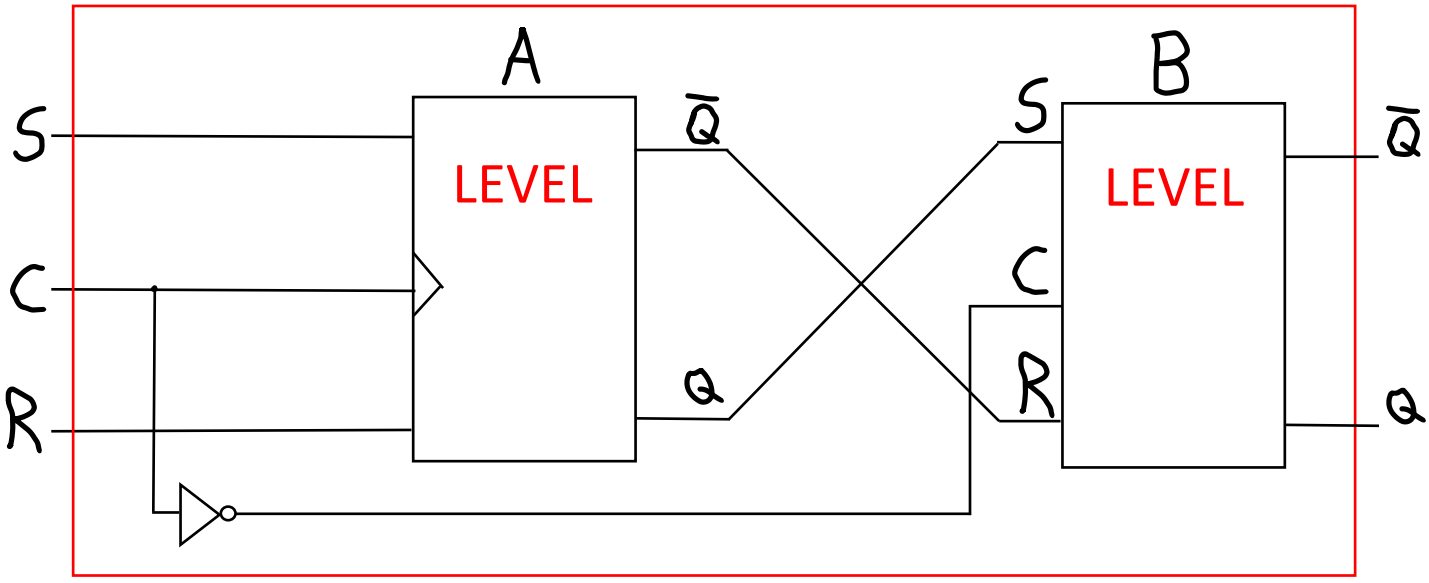




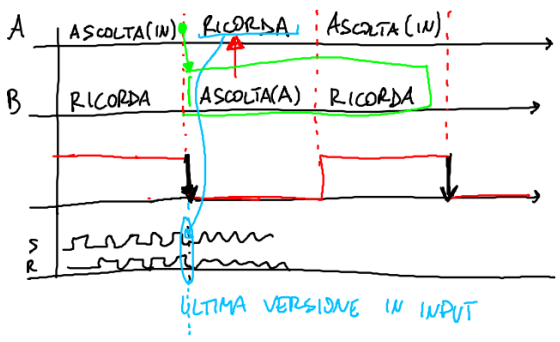
MOMENTO PRECISO
IN CUI ASCOLTARE

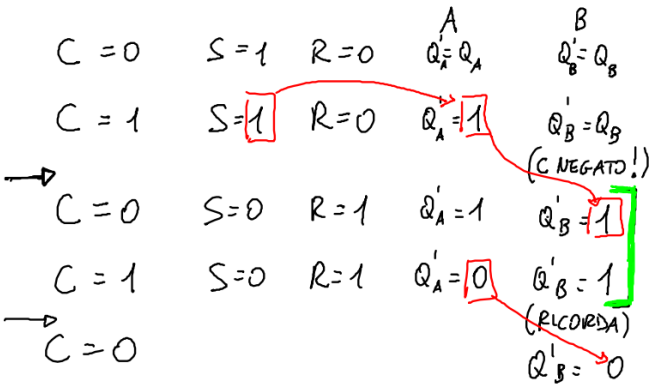
EDGE
(bordo del segnale)

FLIP FLOP SR EDGE TRIGGERED

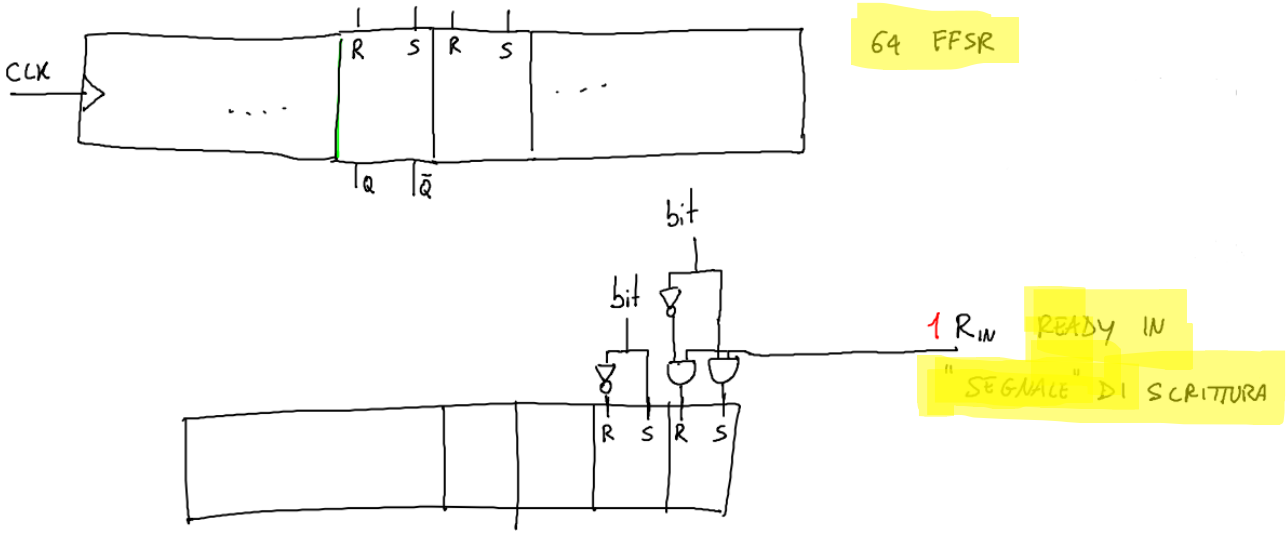


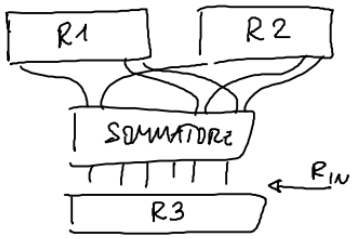
- MOLTO COSTOSO IN TERMINI DI “PORTE”
- SINCRONIA NEL TEMPO





REGISTRO


CLK
R
S
R
S
Q
$\bar{Q}$
64 FFSR
bit
bit
1 R_{IN}
READY IN
"SEGNALE" DI SCRITTURA
R
S
R
S


R1
R2
SOMMATORE
R_{IN}
R3

[1]

MEMORIE

SOLA LETTURA (es. ROM)

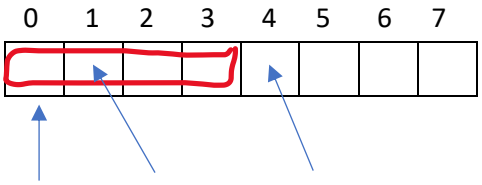
LETTURA e SCRITTURA (es. FFSR)

TECNOLOGIA → ELETTRONICA, MAGNETICA, OTTICA

MODALITA' DI ACCESSO → DIRETTA = RANDOM ACCESS (tempo costante); SEQUENZIALE (dipende da posizione)

DIMENSIONE VELOCITA' = TEMPO DI ACCESSO, QUANTI BIT/s TRASFERIRE

[Disco 10 MB/s, SSD 50 MB/s, RAM 1 GB/s]



CELLA PAROLA INDIRIZZO = 1234 (INDIRIZZI = ID = UINT)

CELLA : contiene un byte (8 bit, 256 elementi possibili)

WORD = PAROLA : n bit elaborati/trasferiti contemporaneamente (8/16/32/64)

oggi siamo qua

01001110011….

32 bit

12 3F AB 00 ESADECIMALE (4 bytes)

INTEL

BIG ENDIAN (a sx byte che vale di +) 12 3F AB 00

LITTLE ENDIAN (a sx byte che vale di -) 00 AB 3F 12

RAM = *Random Access Memory* → memoria elettronica di lettura/scrittura

La LETTURA richiede che venga fornito l'indirizzo della cella da leggere e venga trasmesso alla memoria il relativo comando. In risposta la MEMORIA restituisce il contenuto della cella indirizzata.

La SCRITTURA richiede che venga fornito il dato e l'indirizzo della cella da scrivere e venga quindi trasmesso alla memoria il relativo comando.

SRAM = Static RAM

+ velocità

+ costosa

+ consumo

impiegata per

Cache / Registri di CPU (es. FFSR)

→ MB

DRAM = Dynamic RAM

- velocità

- costosa

- consumo

impiegata per

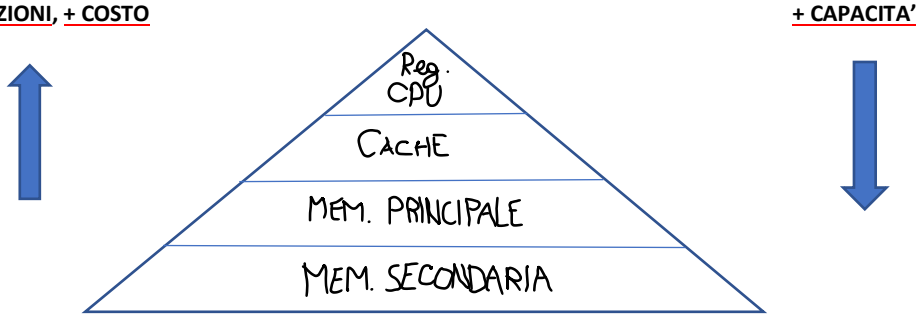
Memoria Principale (RAM)

→ GB

GERARCHIA di MEMORIA

+ PRESTAZIONI, + COSTO

+ CAPACITA'



ISTRUZIONI

Le istruzioni sono salvate nella RAM.

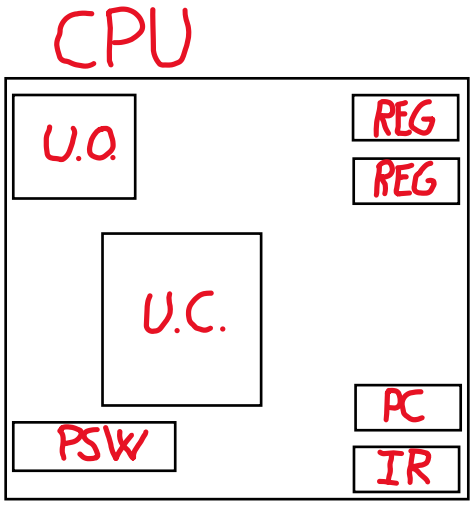
La CPU, grazie ai registri, esegue un'istruzione alla volta.

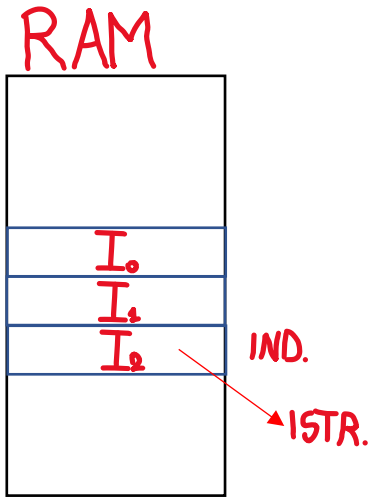
Trasferimenti: **REG ←→ RAM ←→ I/O**

PC (Program Counter): [DOVE] Contiene l'indirizzo della RAM nella quale è salvata l'istruzione che si vuole eseguire.

IR (Instruction Register): [COSA] Una volta prelevata l'istruzione viene messa in questo registro, per poter essere analizzata dalla CPU.

PSW (Process Status Word): è un registro nel quale vengono salvati dei bit detti flag ognuno dei quali rappresenta qualcosa che può essere successo durante le operazioni (divisioni per zero, overflow, numero negativo etc).





ESECUZIONE ISTRUZIONE:

- FETCH (PRELIEVO)

  1) La ***CPU*** preleva l'*istruzione* dalla ***RAM***. L'*indirizzo* al quale accedere è contenuto nel registro ***PC***.

  2) L'*istruzione* viene copiata nell'***IR***.

- EXECUTE
  La ***CPU*** interpreta l'***IR*** e le *azioni* vengono gestite ed effettuate dall'***Unità Operativa*** (grazie all'ALU, etc..).
- AGGIORNA PC++ / ALTERO FLUSSO (es. tramite un IF)

[2]
STACK

STACK (PILA)

Si usa la metodologia STACK, in memoria, per evitare la ridondanza di registri contenenti indirizzi di memoria [nel PC] (es. in un programma ad alto livello).

LIFO = Last IN First OUT

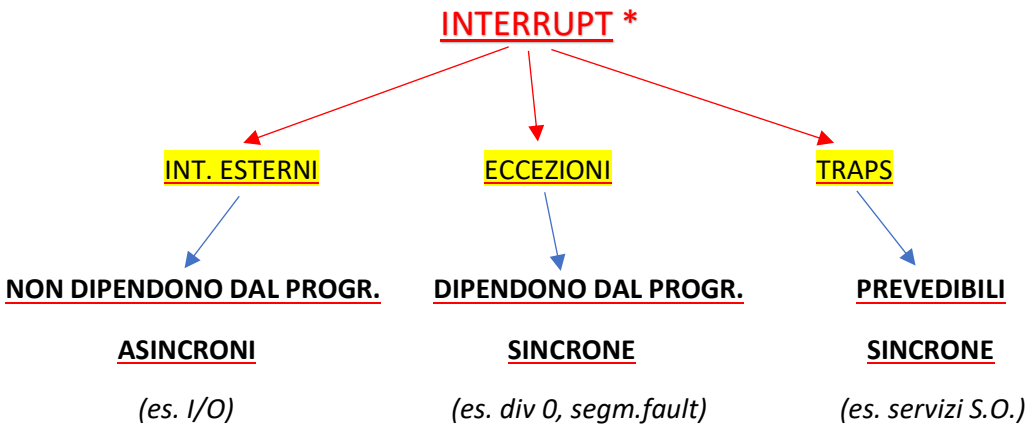
PUSH = "inserisco"

POP = "restituisco"

[inserisco ABCD, restituisco DCBA]

~~E~~
D
C
B
A

PUSH E = restituisco E



* [superutente; "RET. I" quando torno da un interrupt]

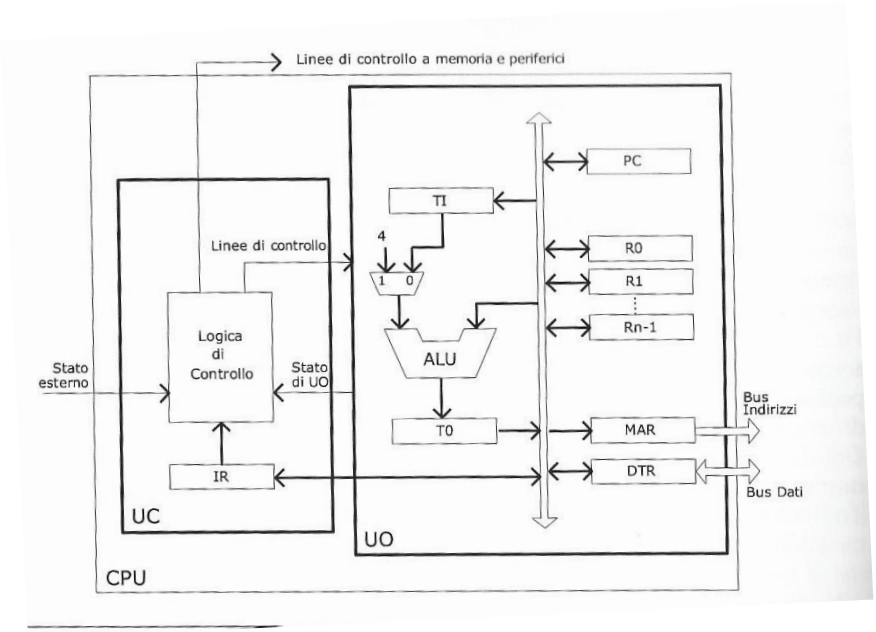
PROCESSORI

INSTRUCTION SET

- **RISC** (reduced) → POCHE e SEMPLICI (programmi lunghi) → ISTR. DIM. FISSA
- **CISC** (complex) → TANTE e COMPLESSE (programmi corti) → ISTR. DIM. VAR.

INTEL

[3]
PARTI COMPONENTI di una CPU



La CPU è scomposta in due parti (sono RETI SEQUENZIALI SINCRONE):

UNITA' OPERATIVA

- Contiene le reti logiche per l'elaborazione (l'ALU, la quale effettua operazione logiche sui dati).
- Aggiorna il PSW (Present Status Word, il quale è un registro con bit che identificano "errori").
- Informa l'Unità di Controllo.

UNITA' DI CONTROLLO

- Gestisce istruzioni e comandi.
- Interpreta il codice operativo contenuto nelle istruzioni e lo traduce in sequenze temporizzate di segnali di comando per l'UO.

ALU: *(Arithmetic Logic Unit)* – Può essere considerata come una rete combinatoria con due ingressi e un' uscita. Contiene le reti che elaborano operazioni aritmetico/logiche.

BUS: Canale di comunicazione, sposto 1 word.

MAR: *(Memory Address Register)* – Contiene l'indirizzo della locazione di memoria da leggere o scrivere. Trasferimento sempre, tramite bus, CPU → RAM.

DTR: *(Data Transfer Register)* – Registro tramite il quale viene scambiata l'informazione tra RAM e CPU.

MMU: *(Memory Management Unit)* – Unità di gestione della memoria principale (RAM)

ESECUZIONE DELLE ISTRUZIONI:

PC → MAR

Il contenuto di PC viene trasferito in MAR.

RAM READ

Viene letta la RAM.

ISTRUZIONE → DTR

Il contenuto della cella indirizzata viene caricato su DTR.

DTR → IR

Il contenuto di DTR viene copiato su IR per essere presentato alla decodifica sull'U.C.

TIN = R2 R3 → ALU

Se devo sommare R2+R3 (in R1), R2 viene trasferito su TIN, R3 viene presentato via bus all'altro ingresso di ALU.

ADD R1, R2, R3

Viene asserito il comando ADD all'ALU.

SOMMA → TOUT

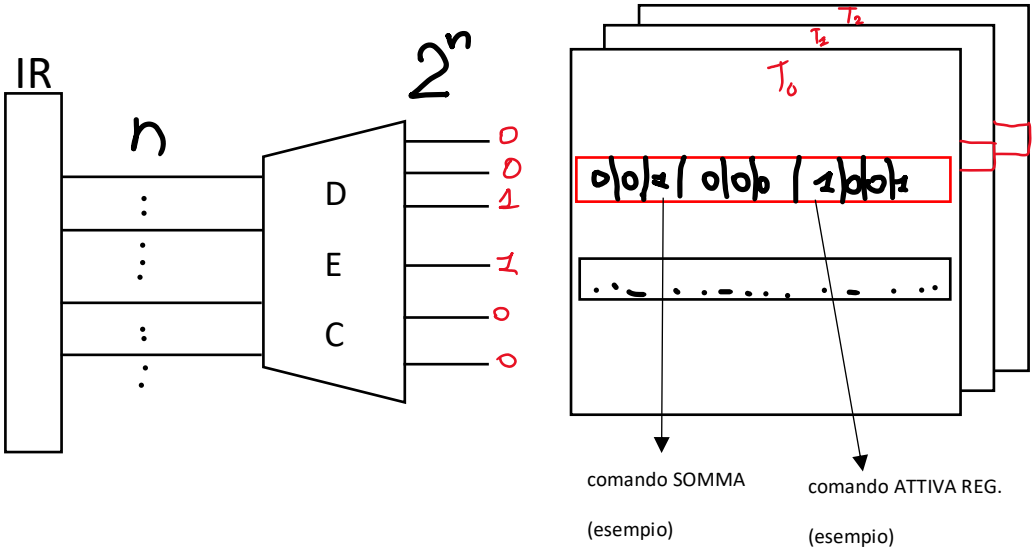
L'output finisce in TOUT.

TOUT → R1

TOUT finisce nel risultato, ovvero R1, tramite bus.

[4]
LOGICA CABLATA/MICROPROGRAMMAZIONE

LOGICA CABLATA



+ TABELLE → TEMPORIZZAZIONE (ogni tabella mi dice ciò che c'è da fare)

VELOCITA' MAGGIORE rispetto alla MICROPROGRAMMAZIONE

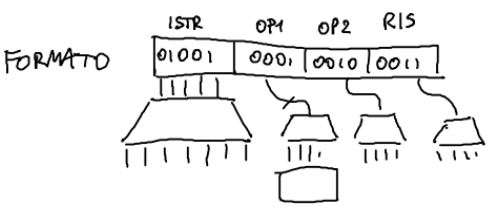
IR → DECODING

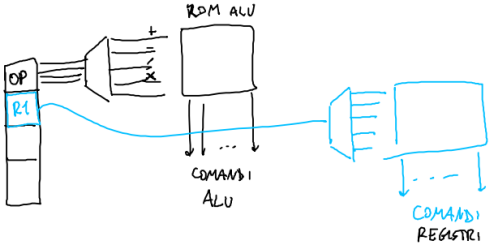
RIGA → corrisponde alla specifica istruzione

COLONNA → specifico comando (es. somma)

+ 0 CHE 1 ! (E' DIFFICILE FAR COMBACIARE TANTI COMANDI CONTEMPORANEAMENTE!)

+ ISTRUZIONI ? → Scrivere ROM diventa DELICATO (+ istruzioni comportano + decoder, e fare modifiche in corsa è quasi impossibile)



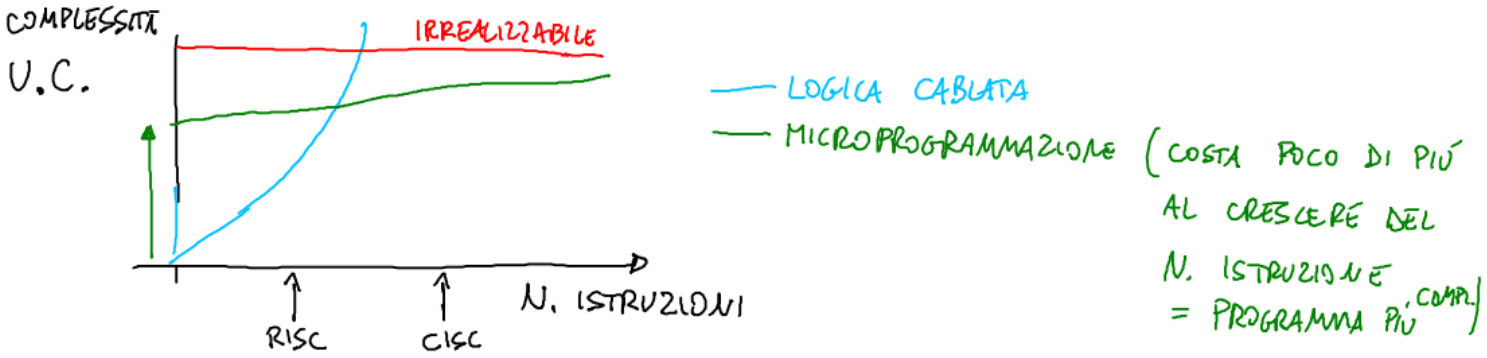


PROBLEMI LOGICA CABLATA:

- CRESCE (ROM, DECODER) AL CRESCERE DELLE ISTRUZIONI
- RIGIDA (0/1 nella ROM)
- NON SI PUO' CAMBIARE

L'unità di controllo cablata è un'unità che utilizza unità logiche combinatorie, dotate di un numero finito di porte in grado di generare risultati specifici in base alle istruzioni utilizzate per richiamare tali risposte. L'unità di controllo microprogrammata è un'unità che contiene microistruzioni nella memoria di controllo per produrre segnali di controllo.

Per apportare modifiche in un'unità di controllo cablata, l'intera unità deve essere riprogettata. Nell'unità di controllo microprogrammata, le modifiche possono essere implementate cambiando le microistruzioni nella memoria di controllo. Pertanto, **l'unità di controllo microprogrammata è più flessibile**.

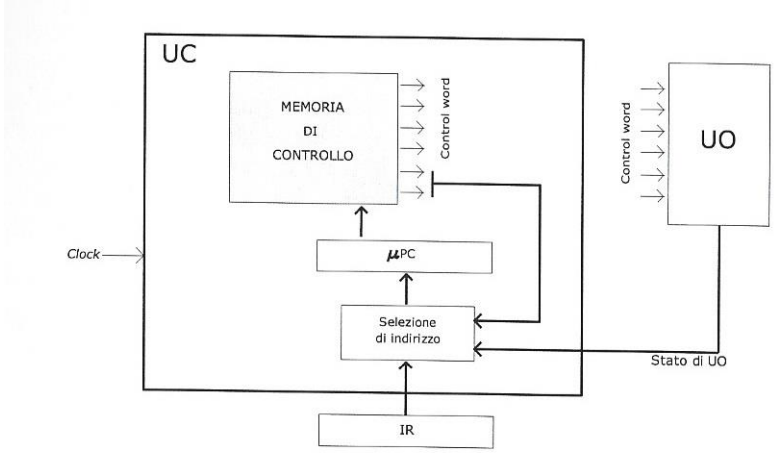


MICRO PROGRAMMAZIONE

Uso microprogramma per calcolare i comandi.

MICROPROGRAMMA → si può CAMBIARE ✓

1 istr. CPU → **tante** microistruzioni che calcolano i comandi



[5]
STORIA LOGICA CABLATA/MICROPROGR.

STORIA

INIZI '60 → **LOGICA CABLATA**

ANNI '80 → **CISC** (*programmi CORTI!*) **MICROPROGRAMMAZIONE**

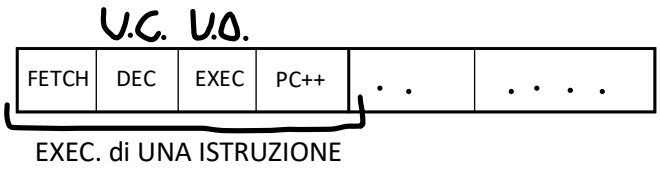
- Gestione delle istruzioni + flessibile
- Istruzioni + complesse

ANNI '80 → **MEMORIE + GRANDI** → **RISC ADATTI!**

COMPILATORI → **OTTIMIZZAZIONE CODICE**

Con i RISC si ha una struttura regolare e programmi semplici.

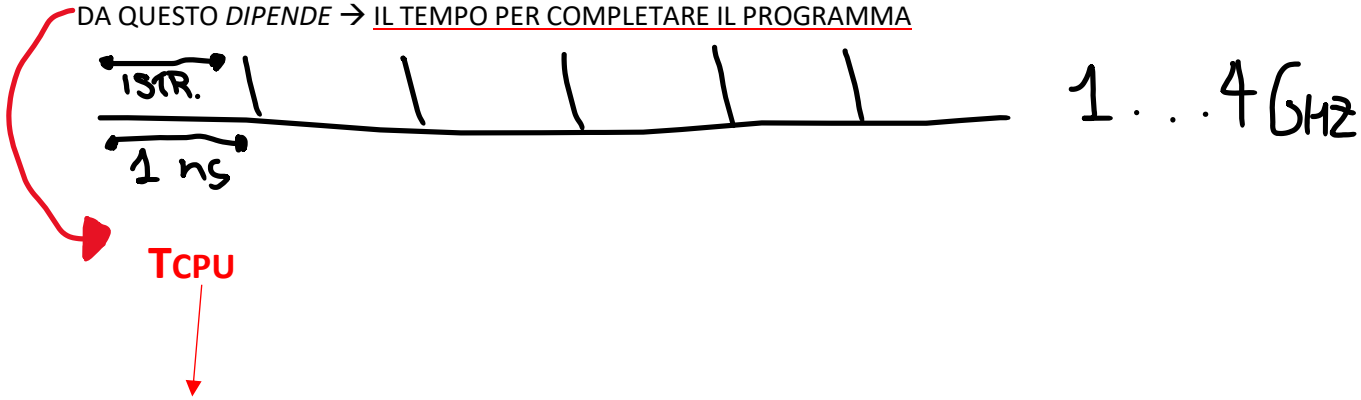
2000 → **ARCHITETTURE PARALLELE**



PRESTAZIONI CPU

- FREQUENZA DI CLOCK (quanto frequentemente vengono eseguite le istruzioni + semplici)

DA QUESTO *DIPENDE* → IL TEMPO PER COMPLETARE IL PROGRAMMA



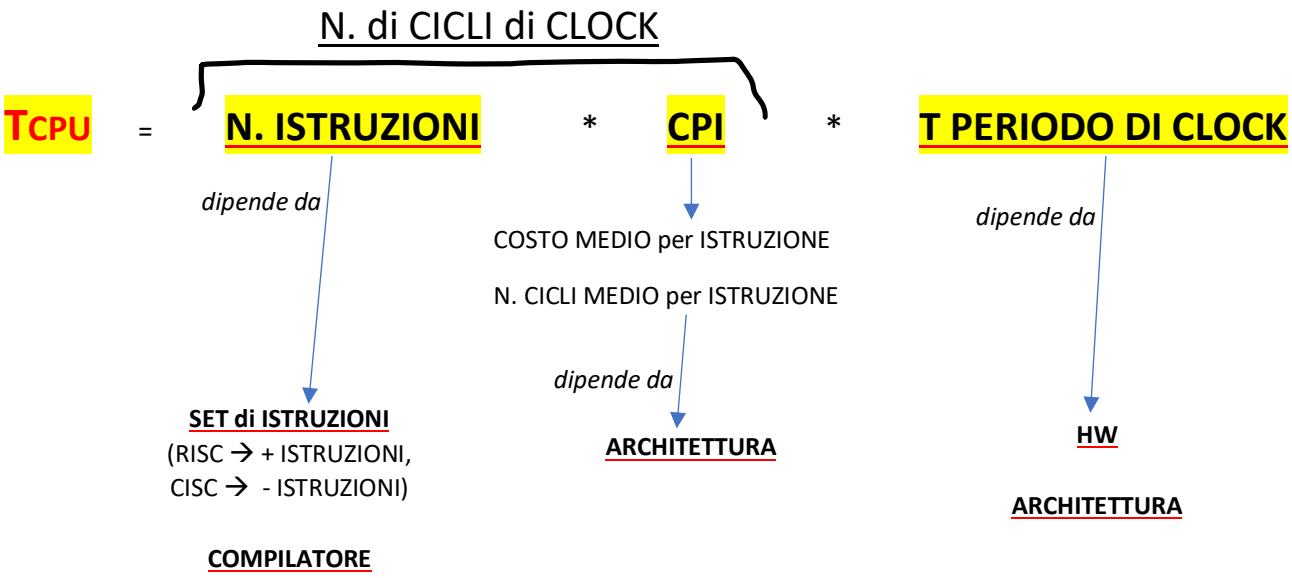
$$\frac{\text{N. CICLI di CLOCK}}{\text{FREQ. di CLOCK}} = \text{N. CICLI di CLOCK} * \text{PERIODO di CLOCK}$$

(es. 1k ISTR. * 1 ns)

P1 (PROCESSORE 1) : FREQ. ALTA * TANTI CICLI di CLOCK

P2 (PROCESSORE 2) : FREQ. BASSA * TANTI CICLI di CLOCK

} IL RISULTATO POTREBBE ESSERE SIMILE!



SPEED UP / ACCELERAZIONE = $\frac{T_{VECCHIO}}{T_{NUOVO}}$ (VARIAZIONE)

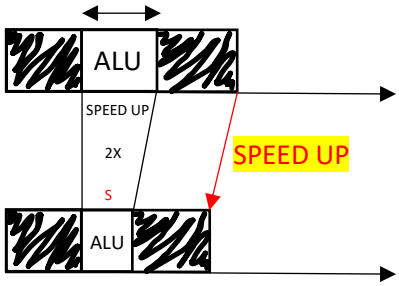
INDICI

- MIPS = MILIONI di ISTRUZIONI per SECONDO
- FLOATING POINT → MFLOPS 10^6
  GFLOPS 10^9
  TFLOPS 10^{12}
  PFLOPS 10^{15}
  EFLOPS 10^{18}

FLOPS = OPERAZIONI FLOATING POINT AL SECONDO

LEGGE DI AMDAHL

Definita nel 1967 da Gene Amdalh, definisce accelerazione (speedup) il rapporto tra le prestazioni ottenute con un miglioramento e le prestazioni di prima del miglioramento.



FU = **FUTILIZZO** = % TEMPO IN CUI SI USA IL COMPONENTE CHE MIGLIORO (tra 0 e 1)

FI = **FINUTILIZZO** = 1 - FUTILIZZO

$$\text{SPEED UP} = \frac{1}{(FI + \frac{FU}{S})} \quad \begin{matrix} = T_{VECCHIO} \\ = T_{NUOVO} \end{matrix}$$

Esempio:

- FU = 0.1 FI = 0.9 S = 2

  $\text{SPEED UP} = \frac{1}{0.9 + 0.1/2} = \frac{1}{0.95} = 1.05$

  IL COMPONENTE BENIFICIA 2X
  MA IL COMPLESSIVO POCO!

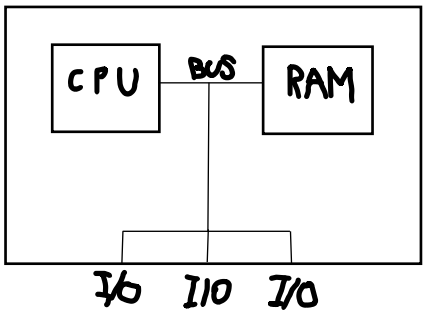
HA SENSO, QUINDI, MIGLIORARE LE ATTIVITA' CON FREQUENZA di UTILIZZO ALTA

(es. in un programma in c++ vado a migliorare i for)

PRESTAZIONI ELABORATORE

- MODELLI
- SIMULAZIONI
- BENCHMARK = programmi che "mixano" pezzi di codice tipici di applicazione
  (panoramica tipica di quell'elaboratore)
  lanciati su architetture diverse

[6]
INPUT/OUTPUT



INPUT: TASTIERA, MOUSE, MICROFONO

OUTPUT: CASSE AUDIO, MONITOR, STAMPANTE

INPUT/OUTPUT: CHIAVETTA, DISCO, SSD, BLUETOOTH, RETE

CPU ---------- *BUS* ---------- INTERFACCIA ---------- DISPOSITIVO I/O

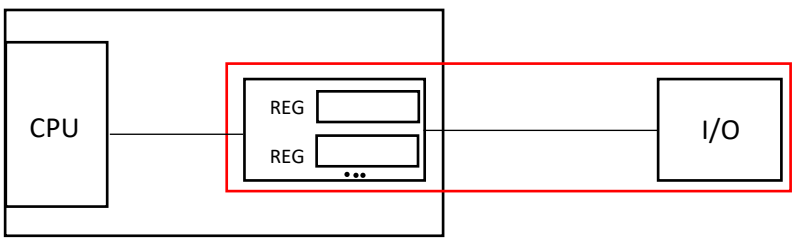
ADATTA LA COMUNICAZIONE FISICA e LOGICA

- COME RAPPRESENTO I DATI NEL MONDO FISICO
- ORGANIZZAZIONE DEI DATI

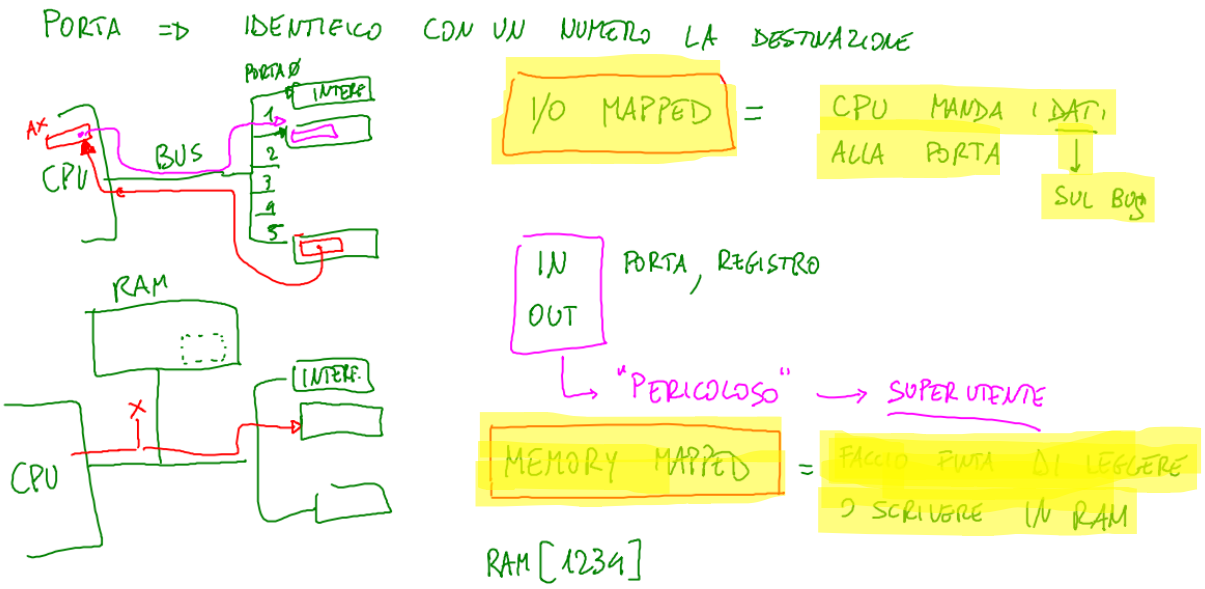
Un dispositivo lavora in modo asincrono (rispetto alla CPU)

L'interfaccia sincronizza la comunicazione

RICORDARE → REGISTRI
- DATI
- COMANDI
- STATO (errori, es. si inceppa la carta della stampante)



I/O MAPPED & MEMORY MAPPED

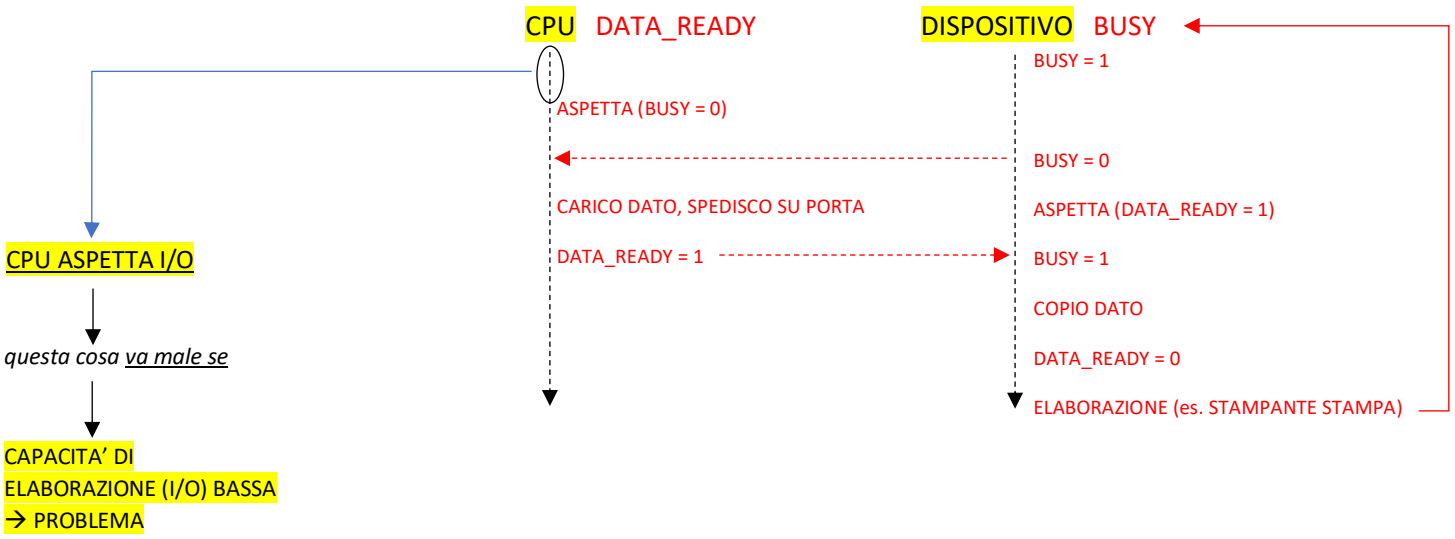


SINCRONIZZAZIONE CPU ←→ I/O

- CONTROLLO DI PROGRAMMA → PROGRAMMA CHE REGOLAMENTA LO SCAMBIO

PROTOCOLLO: HAND SHAKING

DATI
SEGNALI DI CONTROLLO (BIT)
REGOLE

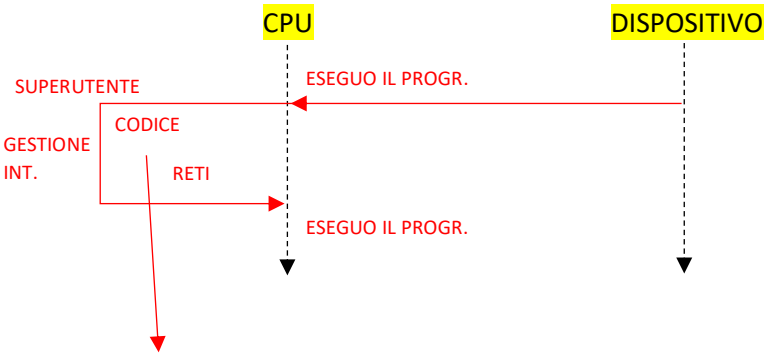


Come migliorare l'HANDSHAKING? ?

↓

- INTERRUZIONI / INTERRUPT

CPU LAVORA, VIENE AVVISATA QUANDO ARRIVA UN EVENTO!



DRIVER

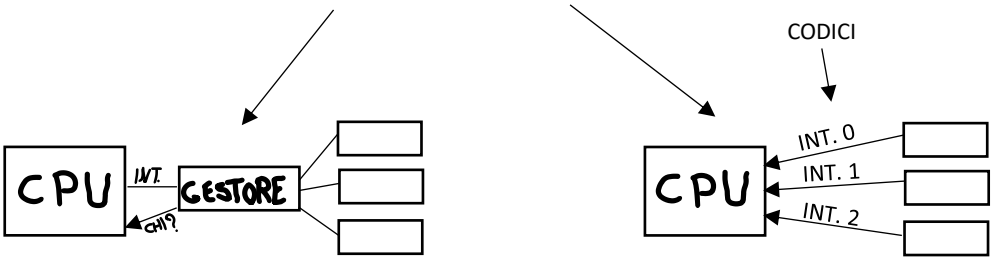
- ➔ CODICE INTEGRABILE con S.O.
- ➔ NELLE ROUTINE per INT.

INTERRUPT
- MASCHERABILI
- NON MASCHERABILI: EVENTI GRAVI (es. corrente salta)

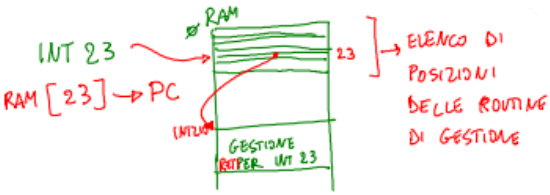
- *CHI SOLLEVA L' INTERRUPT?*
- *CHIAMARE LA ROUTINE GIUSTA*
- *PRIORITA'*
- *UN INTERRUPT PUO' INTERROMPERNE UN ALTRO*

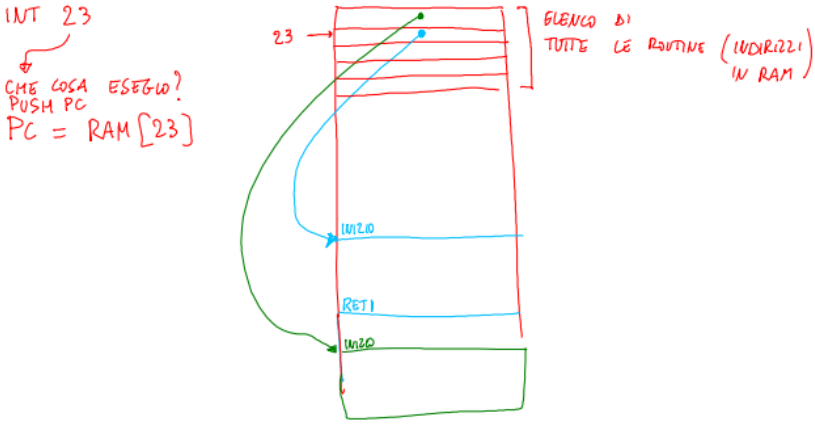
INTERRUPT VETTORIZZATO

Codici assegnati a dispositivi (assegno un numerino)



POCO GESTIBILE!





INT 23

↓

CHE COSA ESEGUO?

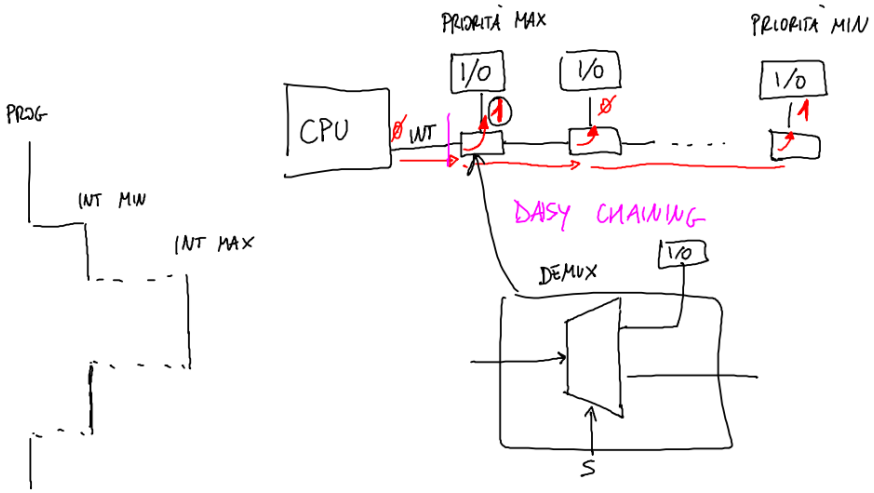
Va nella cella 23 e lì trova l'indirizzo

↓

PC = RAM[23]

[7]
DAISY CHAIN, DMA CONTROLLER, ASSEMBLY

PRIORITA' INTERRUPT



Questo sistema di controllo, grazie alla priorità, fa parlare la CPU col giusto I/O.

CPU chiede "chi ha chiesto attenzione?" e vede l'I/O acceso ad 1 con priorità maggiore.

Quando finisce l'I/O in questione, questo va a 0 e la CPU va avanti a vedere qual è il prossimo 1.

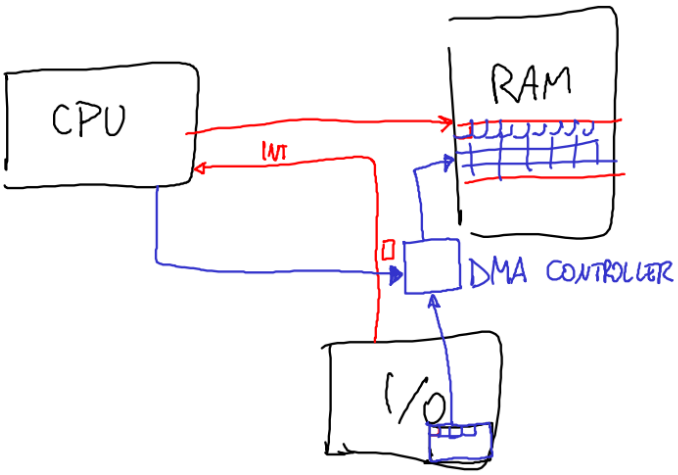
Ma se arriva un interrupt più importante, la CPU "corre" lì, quindi vince sempre quello con priorità max.

Tanti dati da TRASFERIRE?

DMA CONTROLLER

Il **DMA** (**Direct Memory Access**, "*accesso diretto alla memoria*") di un computer è quel meccanismo che permette ad altri sottosistemi, quali ad esempio le periferiche, di accedere direttamente alla memoria interna per scambiare dati, in lettura e/o scrittura, senza coinvolgere l'unità di controllo (CPU) per ogni byte trasferito tramite l'usuale meccanismo dell'interrupt e la successiva richiesta dell'operazione desiderata, ma generando un singolo interrupt per blocco trasferito.

Il DMA, tramite il controllore di accesso diretto (DMAC), ha quindi il compito di gestire i dati passanti nel BUS permettendo a periferiche che lavorano a velocità diverse di comunicare senza assoggettare la CPU a un enorme carico di interrupt che ne interromperebbero continuamente il rispettivo ciclo di elaborazione.



Invece che trasferire un byte alla CPU, gestire l'interrupt e poi mandarlo alla RAM

VORREI

Prendere tutto il blocchetto del file e trasferirlo direttamente alla RAM

DIRECT MEMORY ACCESS

(DMA)

(senza CPU)

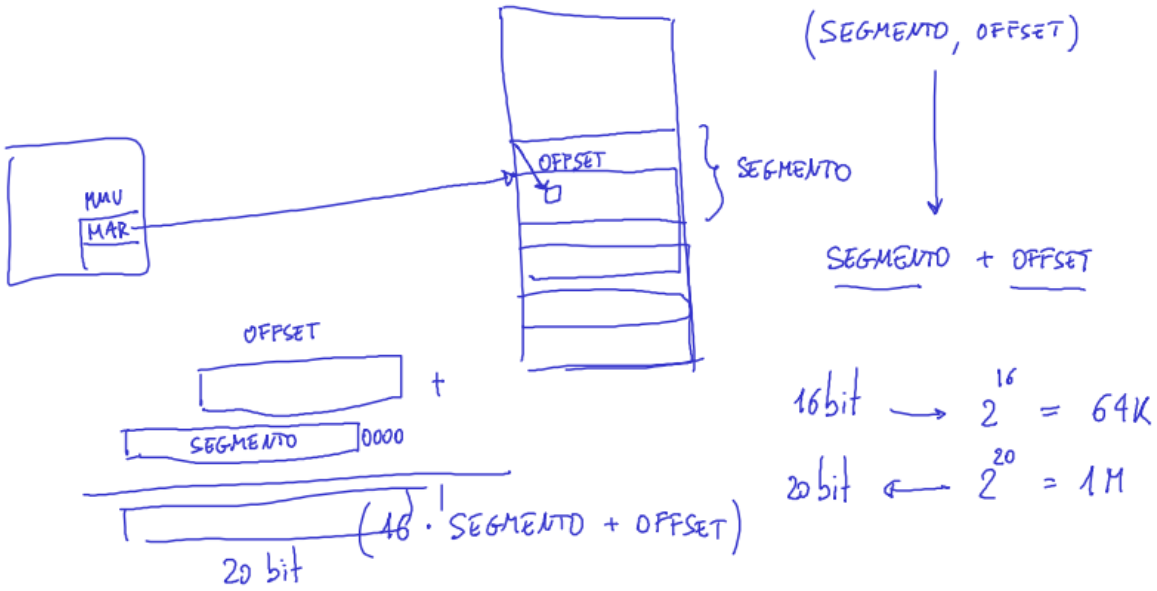
A OGNI BYTE

NO INT PER OGNI BYTE

INFORMAZIONI
- SORGENTE (I/O, INDIRIZZO)
- DESTINAZIONE (RAM, INDIRIZZO)
- N. BYTE

ATTIVITA'
- CICLO DI COPIA
- INT ALLA CPU ("HO FINITO"

CPU nel frattempo può lavorare, ma non può usare il BUS perchè è impegnato, dovrà aspettare.



- IP = Instruction Pointer (sarebbe il Program Counter)
- CS = Segmento
- DS = Data Segment

ISTRUZIONI ASSEMBLY (i numeri sono in complemento a 2)

- **INC AX** (1 byte) : incremento il registro AX (e IP viene incrementato)
- **DEX AX** (1 byte) : decremento il registro AX (se arriva a 0 allora Z si accende a 1, se è pari P a 1)
- **CMP AX, BX** (2 bytes) : (Z=0? I due numeri non sono uguali) (C, S a 1 se AX <= BX)
- **JLE 0110** (2 bytes) : se AX <= BX, altrimenti va all'istruzione dopo (JNE fa "not equal")
- **MOV AL, [0000]** : AL è la parte bassa di AX (a differenza di AH), ciò che in memoria a 0000 → AL

esempio FOR:

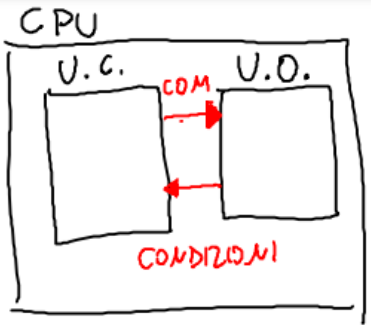
```
0108  MOV [di], AL
      INC  di
      INC  al
      DEC  cx
      JNE  0108 → (se CX non è zero torno su)
```

- **ADD AL, 20** : incrementa AL di 20
- **REP MOVSB** : muove singolo byte
- **CALL 0114** : chiamo funzione a 0114
- **RET** : ritorna da funzione e fa POP per andare all'istruzione giusta
- **PUSH 1234**
- **POP AX**

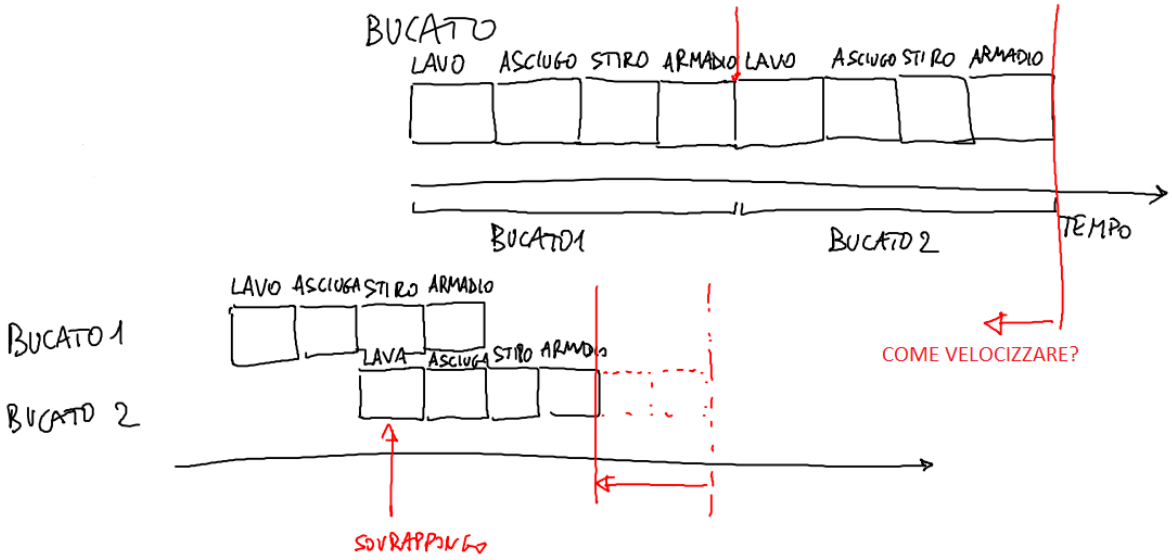
] istruzioni sullo stack

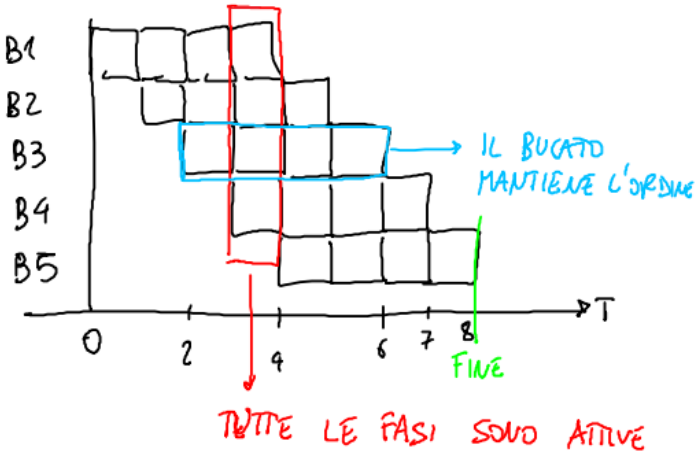
[8/9]

PARALLELIZZAZIONE ATTIVITA' INDIPENDENTI/PIPELINE



La maggior parte dei componenti è ferma!

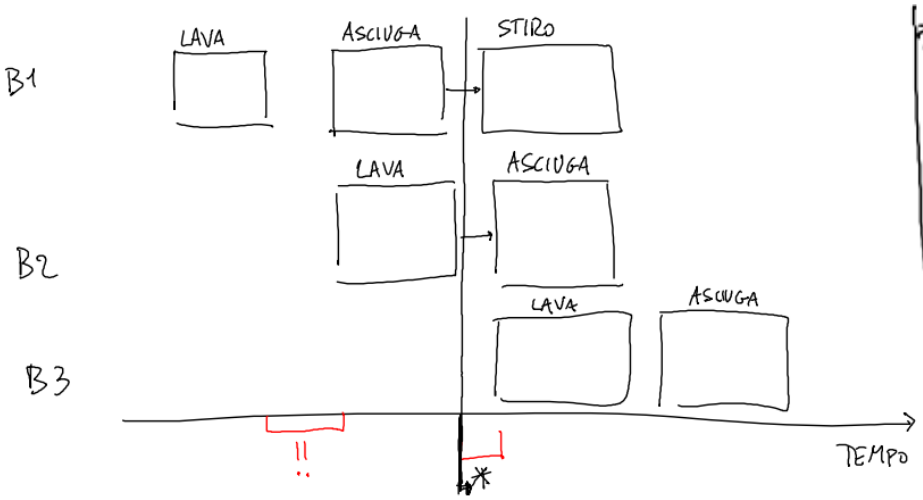


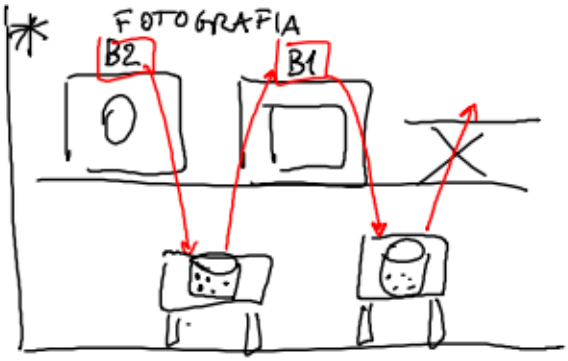


$$\text{SPEEDUP} = \frac{T_{\text{VECCHIO}}}{T_{\text{NUOVO}}} = \frac{5 \times 4 \text{ FASI}}{8} = \frac{20}{8} = 2.5$$

CIRCA IL N. DELLE FASI (STESSA DURATA INDIPENDENTI) $= \frac{M \times 4}{M + 3} \simeq 4$

Dagli anni ’80 si è deciso di “seguire” questo parallelismo



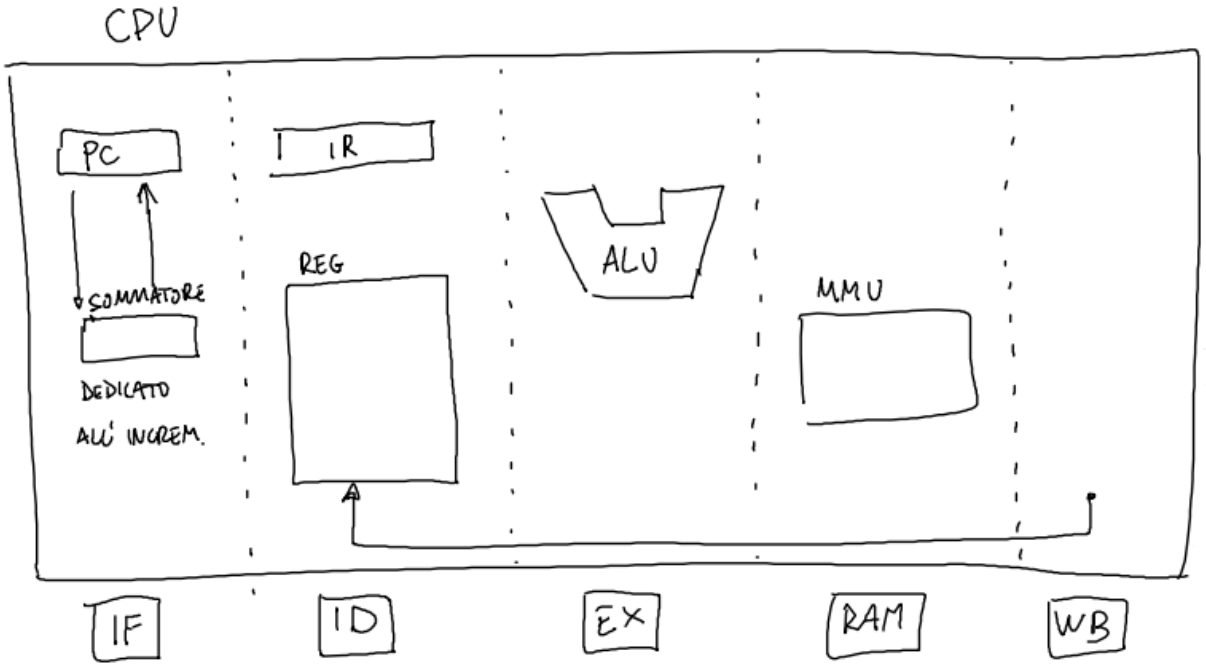


AD OGNI PASSAGGIO, APPOGGIO I VESTITI SU UN CESTO

PROCESSORE PIPELINE CON 5 STADI

Il concetto di pipeline viene utilizzato per indicare un insieme di componenti software collegati tra loro in cascata, in modo che il risultato prodotto da uno degli elementi (output) sia l'ingresso di quello immediatamente successivo (input).

1) [IF] FETCH : PC → RAM → IR , AGGIORNO PC
2) [ID] DECODIFICA & LETTURA REGISTRI : IR → COMANDI
3) [EX] EXECUTE : ES. ALU FA I CONTI
4) [RAM] ACCESSO RAM
5) [WB] WRITE BACK : AGGIORNO I REGISTRI CON I RISULTATI



CRITICITA' PIPELINE

Le criticità sorgono nelle architetture con pipelining quando non è possibile eseguire un'istruzione nel ciclo immediatamente successivo.

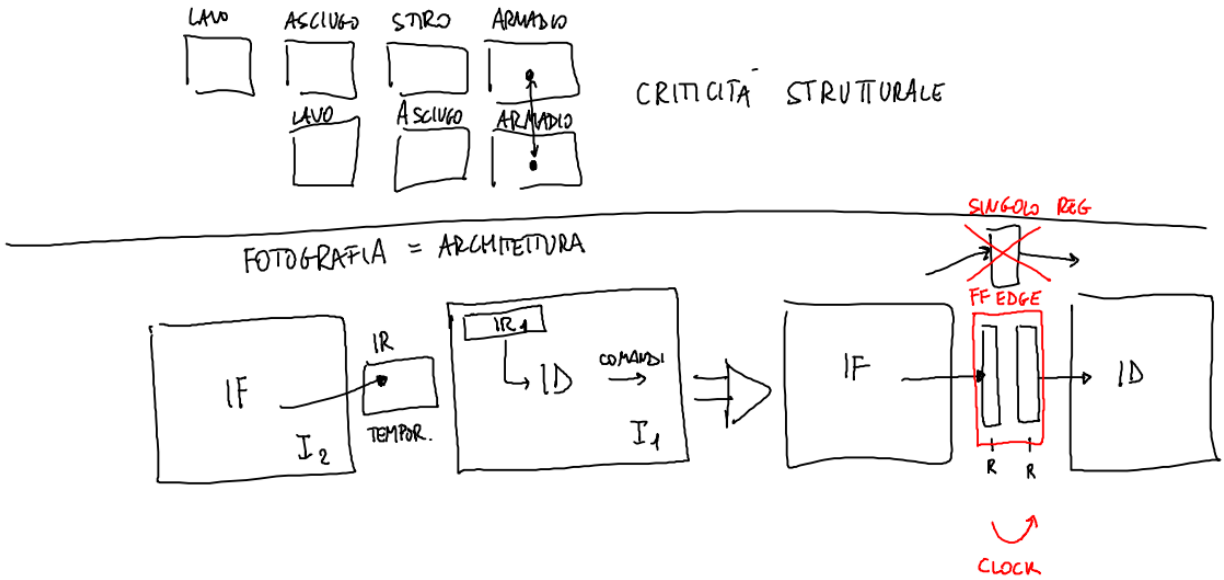
3 tipi di criticità:

- Criticità STRUTTURALI
- Criticità SUI DATI
- Criticità SUL CONTROLLO DI FLUSSO

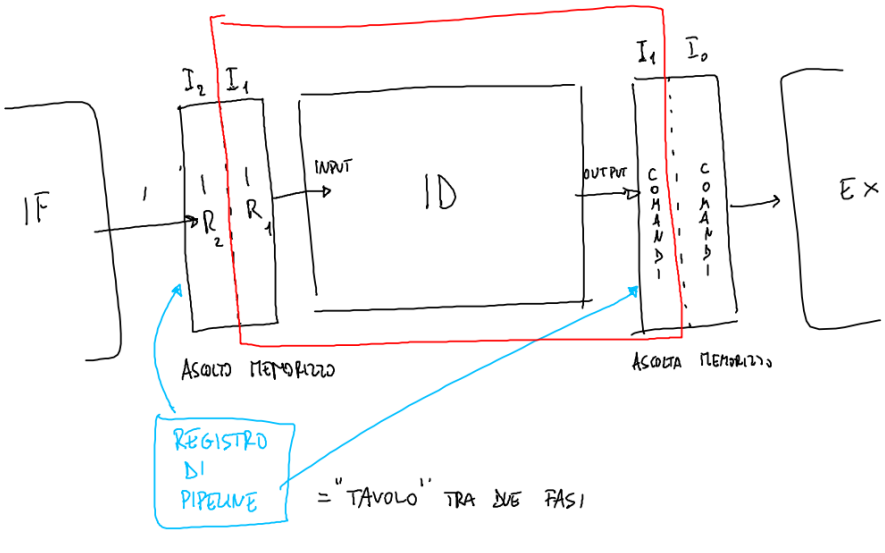
CRITICITA' STRUTTURALE

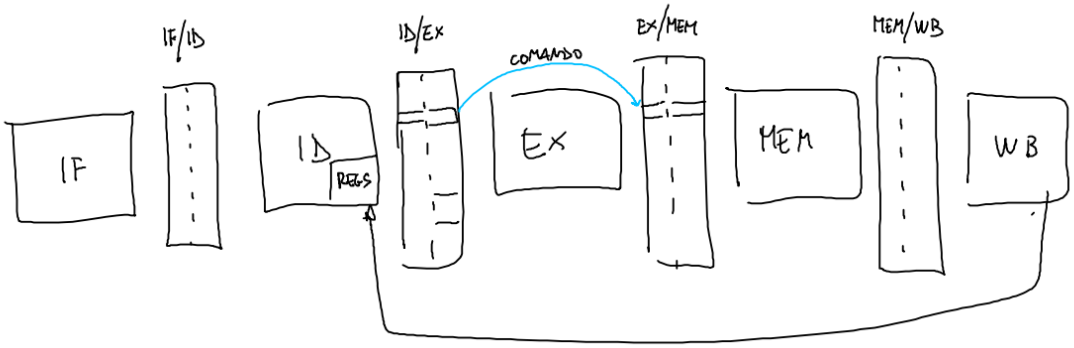
Tentativo di usare la stessa risorsa hardware da parte di diverse istruzioni in modi diversi nello stesso ciclo di clock.

1- COMPONENTE USATO DA 2 STADI
2- STADIO RICHIESTO CONTEMPORANEAMENTE



istruzioni passano dal 1° al 2° registro e poi il 1° registro "ascolta" la nuova fase di fetch





NOTE A PENNA:

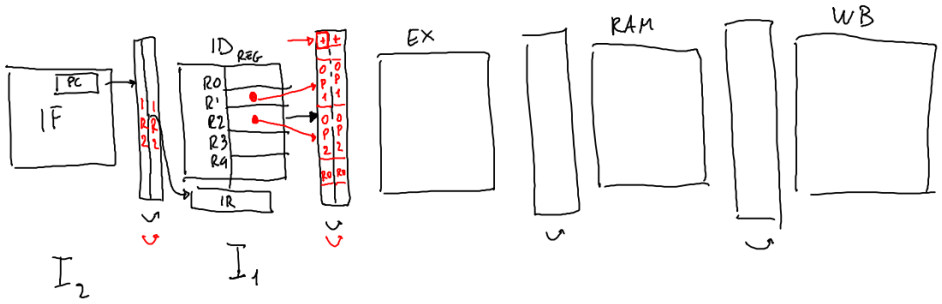
CRITICITA' SUI DATI

Tentativo di usare un risultato prima che sia disponibile.

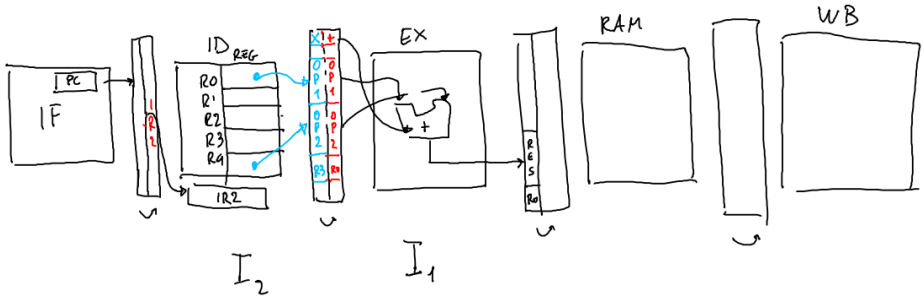
(es.: istruzione che dipende dal risultato di un'istruzione precedente che è ancora nella pipeline)

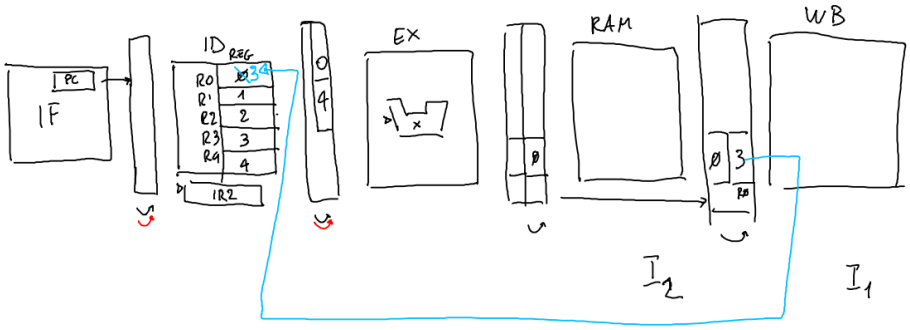
ISTRUZIONE 1 fa somma	I1 ADD R0, R1, R2	R1+R2 → R0	SCRIVO R0
ISTRUZIONE 2 fa moltiplicazione	I2 MUL R3, R0, R4	R0*R4 → R3	LEGGO R0

dipendenza tra le due istruzioni:
I2 deve aspettare che I1 abbia finito



PASSO INFORM. LATO DESTRO DEI REG PIPELINE = CAMBIO STADIO





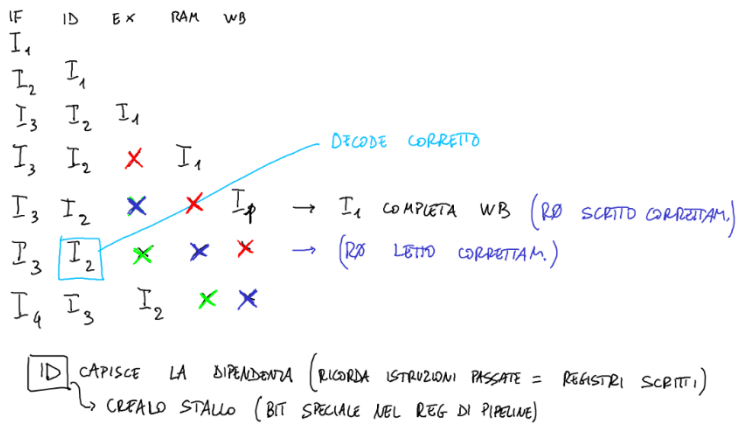
PROBLEMA!

Il risultato (1+2) arriva tardi per I2 che ha fatto 0*4 invece che 3*4

NOTE A PENNA:

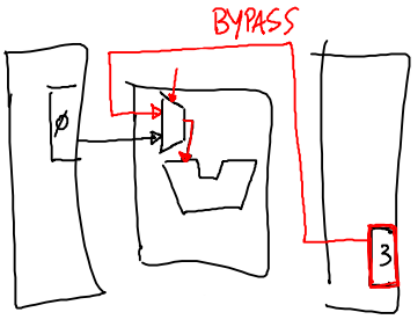
1) Per risolvere ciò peggioro le prestazioni ma garantisco la correttezza dei dati, tenendo ferma I2 nella fase di decode, mandando avanti I1. **STALLO BUBBLE**

riempiamo quello stadio
e non faccio niente



2) E se non voglio stalli? BYPASS

So dove prendere il valore corretto e introduco un multiplexer. Prelevo il dato corretto e non ancora scritto e lo uso al posto di quello decodificato.



I1 MOV AX, [1234] LETTURA RAM → REG

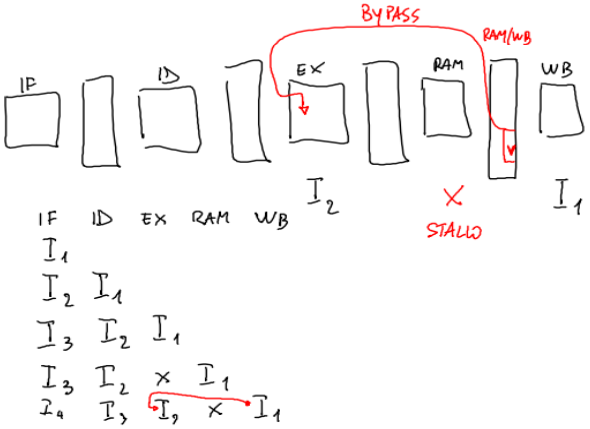
I2 ADD BX, AX

↓

BYPASS + STALLO

Serve 1 stallo e 1 bypass da RAM a EX perché non sono stadi in successione.

Devo aspettare che la RAM sia terminata, poi posso usare il valore → inserisco 1 STALLO



COMPILATORE → INDIRIZZI INRAM

A = B + E (12, 4, Ø)

C = B + F (16, 4, 8)

I1	MOV	AX, [Ø]	AX ← E
I2	MOV	BX [4]	BX ← B
I3	ADD	AX, BX	AX = B + E
I4	MOV	[12], AX	A ← B+E
I5	MOV	AX, [8]	AX ← F
I6	ADD	AX, BX	AX ← B + F
I7	MOV	[16], AX	C ← B + F

IF ID EX RAM WB

I4 I3 X I2 I1

I5 I4 I3 X I2

COMPILATORE → INDIRIZZI INRAM

A = B + E (12, 4, Ø)

C = B + F (16, 4, 8)

ANTICIPO ISTRUZIONE INDIPENDENTE

I1	MOV	AX, [Ø]	AX ← E
I2	MOV	BX [4]	BX ← B
I3	ADD	AX, BX	AX = B + E
I4	MOV	[12], AX	A ← B+E
I5	MOV	CX, [8]	CX ← F
I6	ADD	CX, BX	AX ← B + F
I7	MOV	[16], CX	C ← B + F

IF ID EX RAM WB

I4 I3 I5 I2 I1

I6 I4 I3 I5 I2

Ci sarebbe uno stallo perciò in I5, I6, I7 uso CX invece che AX e anticipo

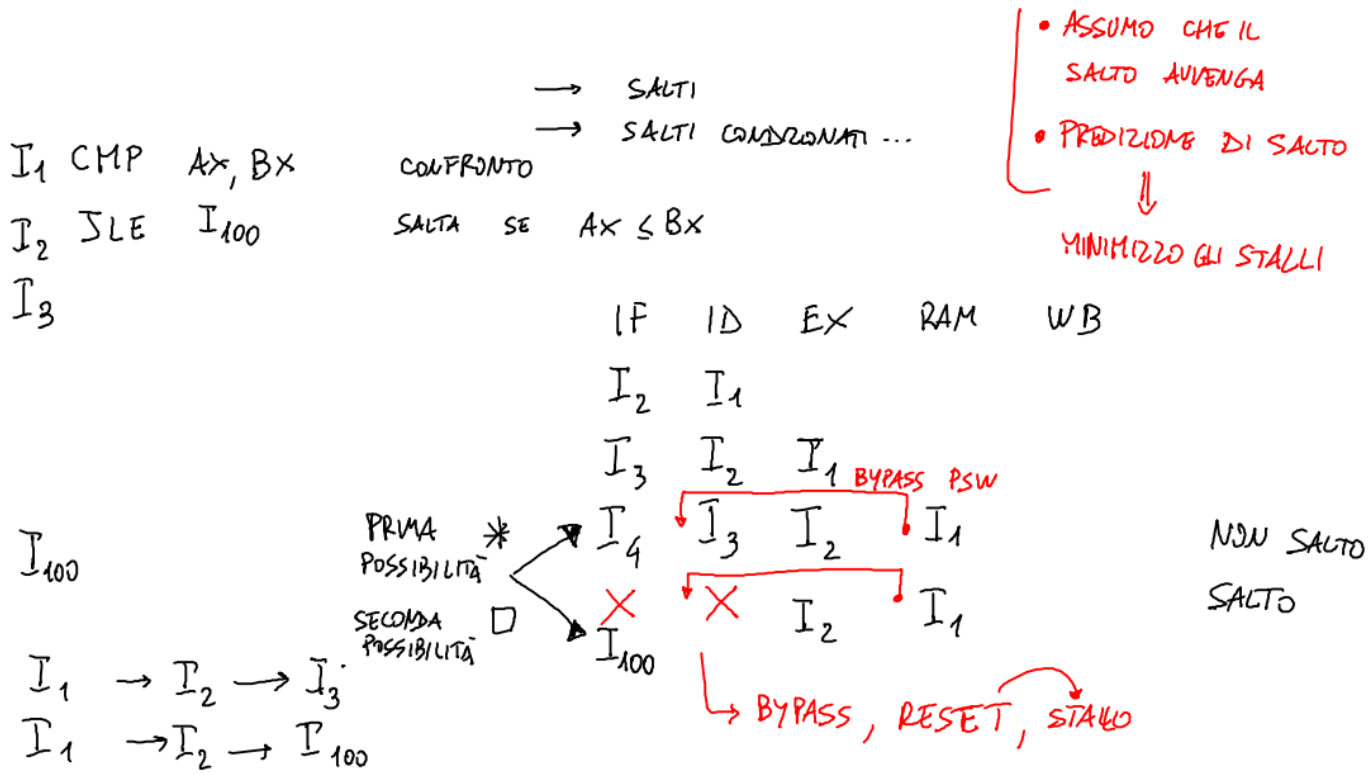
→

I5 dopo I2

BYPASS ✓

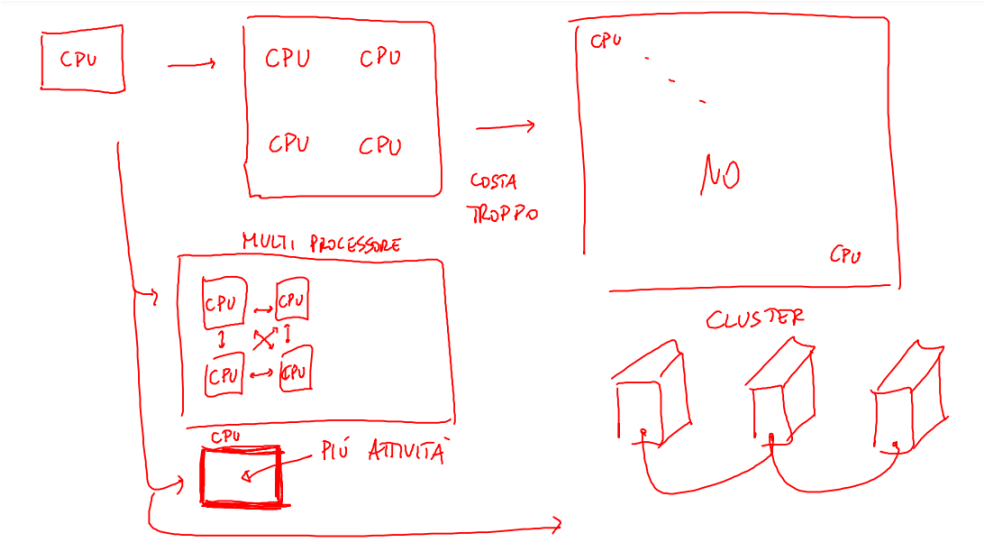
STALLI ✗

CRITICITA' SUL CONTROLLO DI FLUSSO



NOTE A PENNA:

[10]
ARCHITETTURE PARALLELE



Meglio tante CPU piccole, che una grande!

- LAVORO DISTRIBUITO a TANTI PROCESSORI

FLYNN → TIPOLOGIE PROCESSORI PARALLELI

ISTRUZIONI → SINGOLE/MULTIPLE
DATI → SINGOLI/MULTIPLI

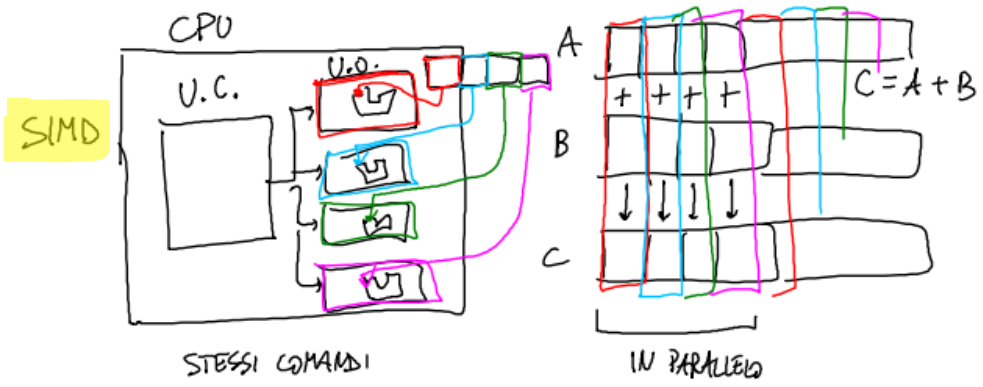
ISTR \ DATI	S	M
S	SISD	SIMD
M	MISD	MIMD

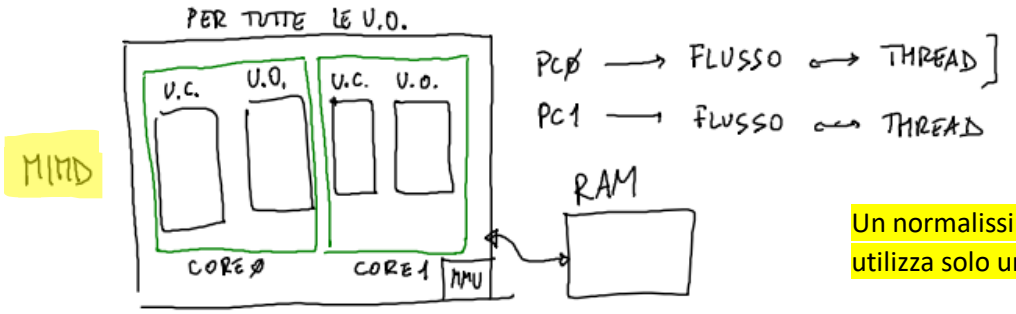
SISD → Processore Classico (non si trova più..)

MISD → ?? (poco interessanti

SIMD → Vettoriale (es. stessa op. su + dati in parallelo)

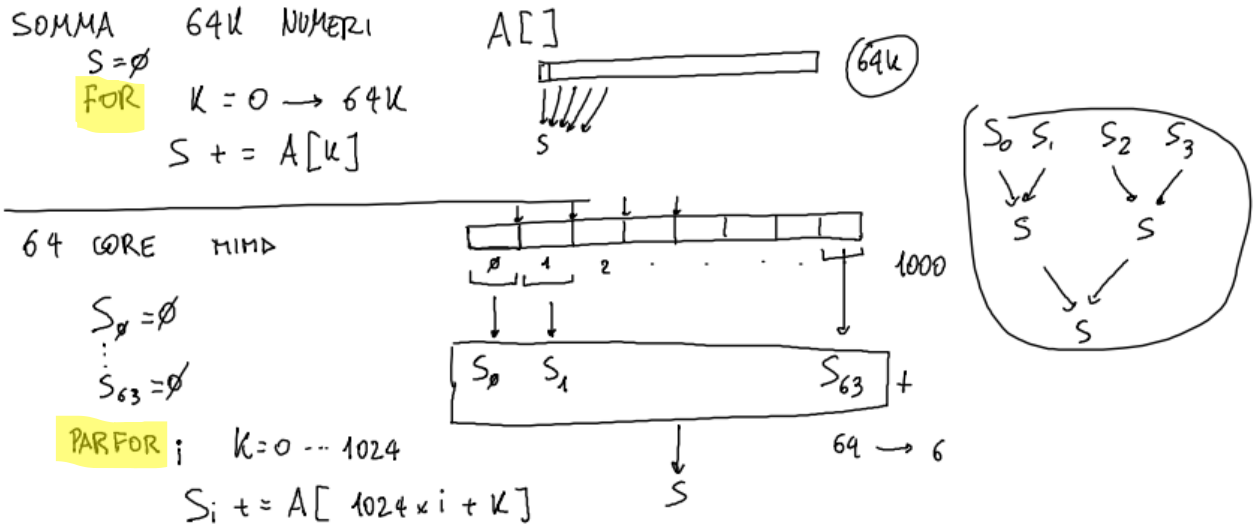
MIMD → Più flussi di esecuzione





Un normalissimo programma in C utilizza solo un CORE

Esempio di programma parallelo per la somma di un array con riduzione parallela

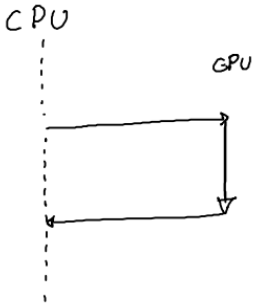


GPU

GPU → Graphics Processing Unit

I/O

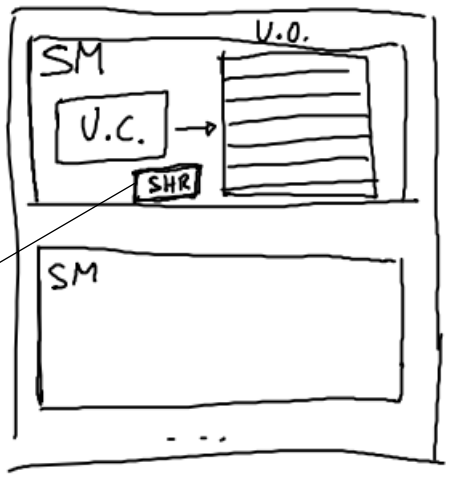
'97 GPGPU (GP^2U) → GENERAL PURPOSE GPU (programmabile, posso far conti)



CPU può chiedere aiuti a GPU per velocizzare alcuni conti

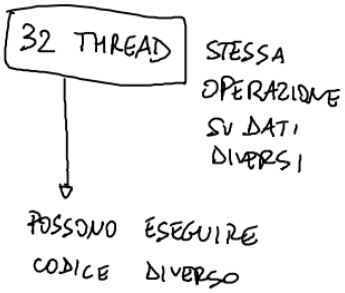
CPU		GPU	
LATENZA	BASSA	LATENZA	ALTA (ci metto tanto per arrivare ai dati)
BANDA	BASSA	BANDA	ALTA (veloce a trasferire i dati)
CACHE			

SM = Streaming Multiprocessor / Shared Processor



SHR: memoria condivisa (shared)

TANTI REGISTRI



SIMT = Single Instruction Multiple Thread