



UNIVERSITÀ DEGLI STUDI DI PARMA

Corso di Laurea in

"Ingegneria delle Tecnologie Informatiche" (9 CFU)

"Ingegneria Informatica, Elettronica e delle Telecomunicazioni" (6 CFU)

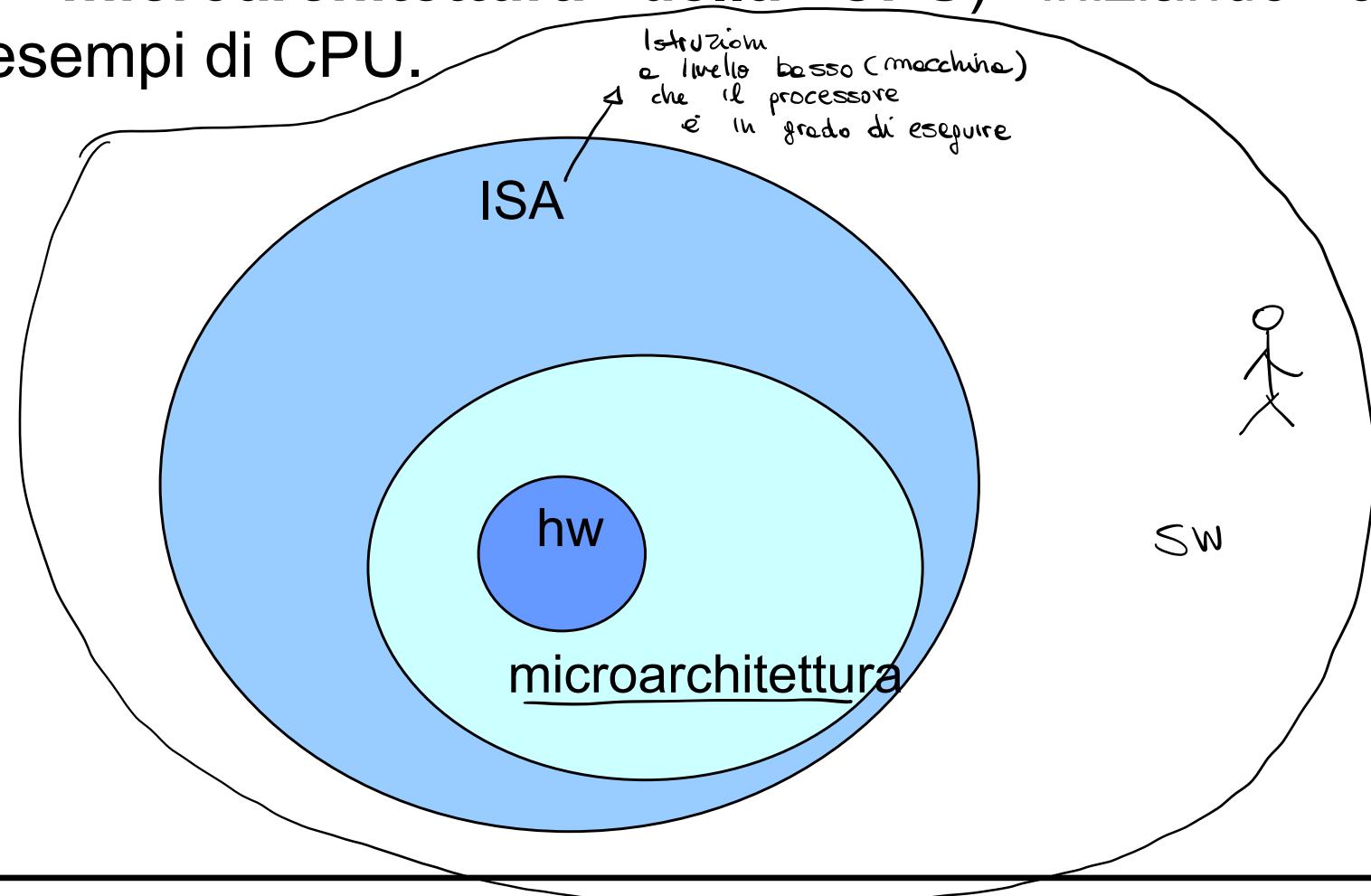
Architettura dei Calcolatori Elettronici

Microarchitettura della CPU

Andrea Prati

Microarchitettura della CPU

- Come visto il processore (o CPU) è l'elemento più importante del calcolatore. Cominciamo quindi a vederne le caratteristiche principali (spesso ci si riferisce con il termine **microarchitettura della CPU**) iniziando con alcuni esempi di CPU.



Progetto di una CPU

Diverse fasi

Architetture
Diverse → ISA Compatibili
↑

- 1) **definizione dell'ISA.** Più CPU sono compatibili a livello di ISA, ma non di architettura interna (ad es: Intel Pentium, Intel Celeron, AMD, ...)
- 2) **definizione dei blocchi logici della CPU a livello di microarchitettura** (come realizzare la pipeline, la superscalarità)
- 3) **definizione di blocchi aggiuntivi** per migliorare le prestazioni (ad es. branch prediction table)
- 4) **come progettare i singoli blocchi** (come realizzare le ALU, i registri e soprattutto l'unità di controllo)
non lo vedremo
- 5) **come interfacciare la CPU con l'esterno** (interfaccia con la memoria e struttura dei bus)

Se poi il sistema è multiprocessor le scelte progettuali sono ancora più numerose.

Instruction Set Architecture

ISA (Instruction Set Architecture) definisce:

- come la macchina è vista dal programmatore
- come la macchina è vista dal compilatore → *che traduce il linguaggio di alto livello in quello macchina (basso livello)*
- indipendentemente da come è fisicamente realizzata (dall'hardware e dalle microarchitetture)
- Più l'ISA è semplice (vicino all'hardware) → *he esistono alcune più vicine all'hw. o più vicine al programmatore*
 - maggior lavoro è deputato al programmatore;
 - più veloce e semplice è la rete logica che la implementa
- Evoluzione delle CPU dipende dalla evoluzione:
 - dell'architettura
 - della tecnologia
 - dei linguaggi di programmazione di alto livello
 - del costo del software

ISA ed Evoluzione delle CPU

1) Anni '70: Sviluppo delle macchine poi denominate "**CISC**" Complex instruction set computer

- Aumento del set di istruzioni (ISA complesso)
- Trasferimento della complessità nel microcodice (più difficile decodificarli)
- Anche istruzioni complesse
- Programmi più corti

- VAX 11/760 1978 300 istruzioni
- 68020 18 modi di indirizzamento (60% di silicio per il microcodice)

2) 1980 (Diezel Patterson) progetto "**RISC**" Reduced Instruction Set Computer:

- non si ottimizzano le prestazioni portando il set di istruzioni verso le applicazioni ed aumentandone la complessità ma creando un set di istruzioni che esegue **EFFICIENTEMENTE** le istruzioni critiche

→
Invece di introdurre istruzioni per la moltiplicazione
Continuiamo a farle come sequenze di somme ma ottimizziamo le singole somme per renderle



- I calcolatori detti **RISC** sono caratterizzati da una ISA molto semplificata (poche e veloci istruzioni)
- Comportamento dei programmi:
 - ⇒ *I'80% delle istruzioni eseguite corrispondeva al solo 20% del repertorio.*
 - ⇒ *conviene investire nella riduzione dei tempi di esecuzione di quel 20%, anziché aggiungere raffinate istruzioni, quasi mai usate, ma responsabili dell'allungamento del tempo di ciclo di macchina*
 - ⇒ *conviene costruire processori molto veloci, necessariamente con repertori semplici, e contare sull'ottimizzazione del compilatore*
- Le macchine **CISC**, invece, hanno una ISA più complessa, comprendenti anche istruzioni poco frequenti, “vicina” ai linguaggi di alto livello
- **Paradigma RISC:**
 1. studiare istruzioni più frequenti
 2. ottimizzare l'hardware per queste istruzioni
 3. aggiungere ciò che non modifica le prestazioni del punto precedente

porta ad una lettura più veloce → "spreco" della memoria"

RISC vs. CISC

(tipica domanda d'esame)

- RISC:**

- minor numero di tipi di istruzioni nell'ISA
- istruzioni di lunghezza fissa (es. 6 bit come nel nostro caso) → un approccio ESTREMO/TOTALEMENTE RISC avrebbe istruzioni a n bit fissi
- decodifica più semplice
- unità di controllo cablata → implementata su silicio
- programmi più lunghi
- pochi metodi di indirizzamento
- memoria allineata
- molti registri, di lunghezza fissa e ortogonali
- processori load/store

- CISC:**

- molte istruzioni, specializzate
- codice operativo e istruzioni di lunghezza variabile
- decodifica complessa, che richiede più cicli di clock
- unità di controllo microprogrammata → non posso renderla cablata x la sua complessità → tempi di decodifica più lunghi
- programmi più corti
- molta flessibilità nei metodi di indirizzamento → flessibilità ed accedere ai dati in memoria
- memoria non allineata → più cicli di accesso alla memoria
- pochi registri, di varie lunghezze e non ortogonali → es. - 4 o 16 bit
8 o 8 bit

CPU RISC

$$T_{cpu} = (\sum_i N_{I_i} CPI_i) T_{ck}$$

Istruzioni *

NCC

1 f_{ck}

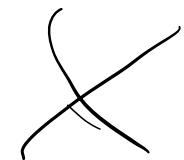
- Obiettivi:
 - eseguire la maggior parte delle istruzioni in un solo ciclo di clock (diminuire il CPI_i)
 - rendere l'implementazione più semplice in modo da minimizzare la durata del ciclo di clock (T_{ck})
- Caratteristiche dell'ISA delle CPU RISC:
 - poche istruzioni disponibili (MIPS 31, RISC II 39)
 - set di registri ortogonale
 - aumento del numero dei registri interni
 - architettura register/register
 - trasferimento di funzioni al compilatore
 - tutte le istruzioni della stessa lunghezza
- ciò si traduce in alcune regole di implementazione, comuni (assenza di microcodice, microcontrollo hardwired, uso di pipeline, visibilità del parallelismo interno,...)

* $N_I = 100$

	N_{I_i}	CPI_i	Ciclo di Clock
ADD	50	1	
LD	10	10	
ST	5	10	
SUB	20	1	
ML	15	5	
)
			quanto Tempo l'istruzione ci mette ad essere eseguite

Due esempi speculari

- Famiglia “x86”:
 - Intel
 - PC e altro
 - CISC
- Famiglia ARM:
 - ACORN → ARM
 - Sistemi “embedded” e “mobile”
 - RISC



Intel 4004

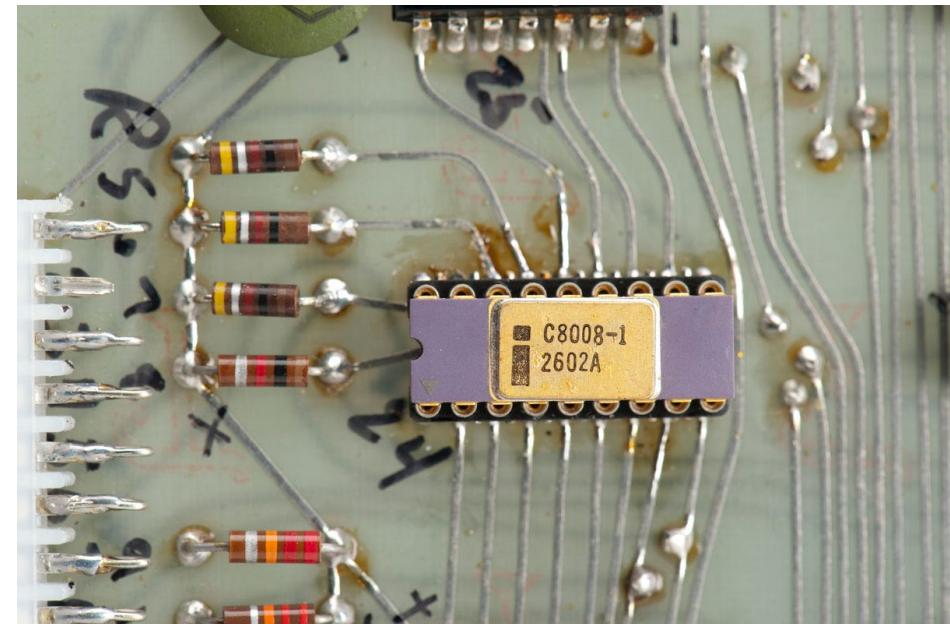
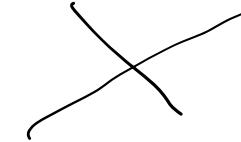
- Nasce nel 1968
 - Produrre circuiti integrati per memorie
 - Nel 1969 gli commissionano 12 chip per un calcolatore
 - Crea un singolo chip il 4004
 - Processore a 4 bit
 - 2300 transistor
 - Faggin

X



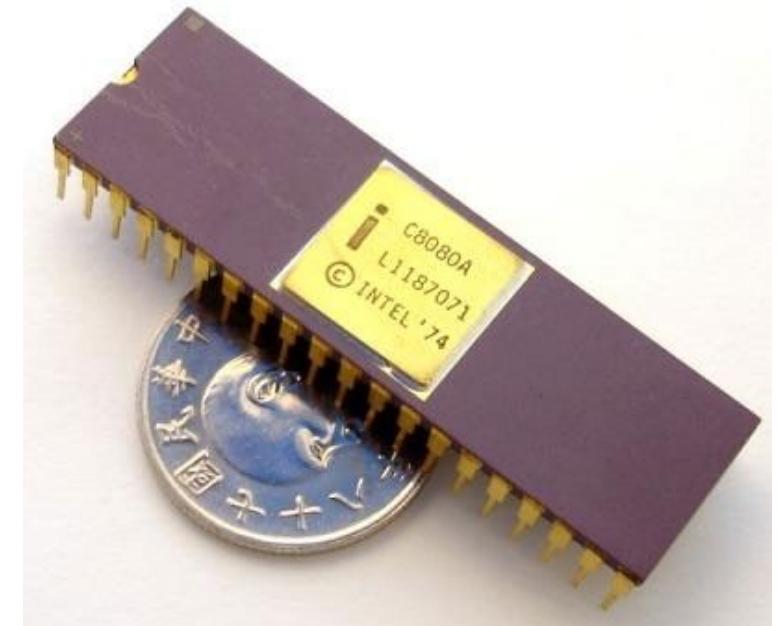
Intel 8008

- Evoluzione ad 8 bit del 4004
- In realtà più lento ma più memoria
 - 16 kbyte
- Successo inaspettato

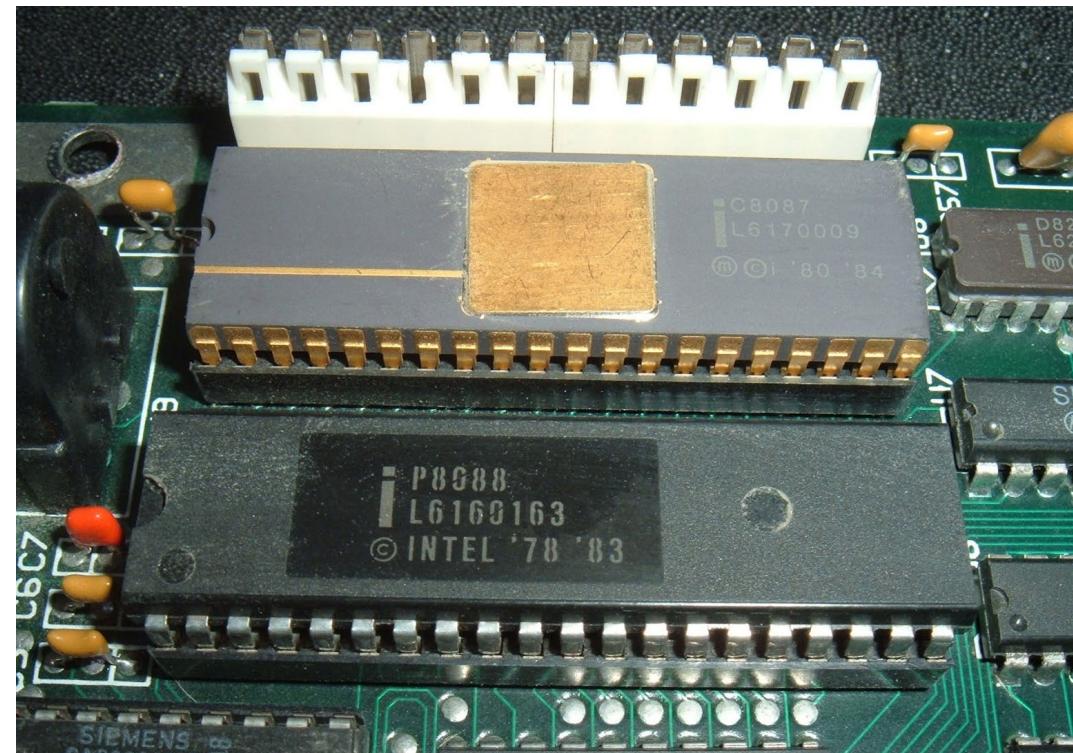


Intel 8080

- Evoluzione 8008
 - 2 MHz
 - Superato limite 16 kbyte
(64 kbyte)
 - Ne vendono milioni
 - “Nasce” il Personal Computer



Intel 8086/8088 (8087)



- CPU a 16 bit
 - 8086 bus a 16 bit
 - Olivetti M24
 - 8088 bus a 8 bit
 - Scelta più economica e vincente
 - IBM PC
 - 8087 primo coprocessore matematico
- 1 MB memoria

Intel 80286

- Nasce per superare limite 1 Mbyte
 - 16 Mbyte
- Retrocompatibile ma gestione memoria complessa
- Fondamentalmente considerato come un 8088 più veloce



Intel 80386

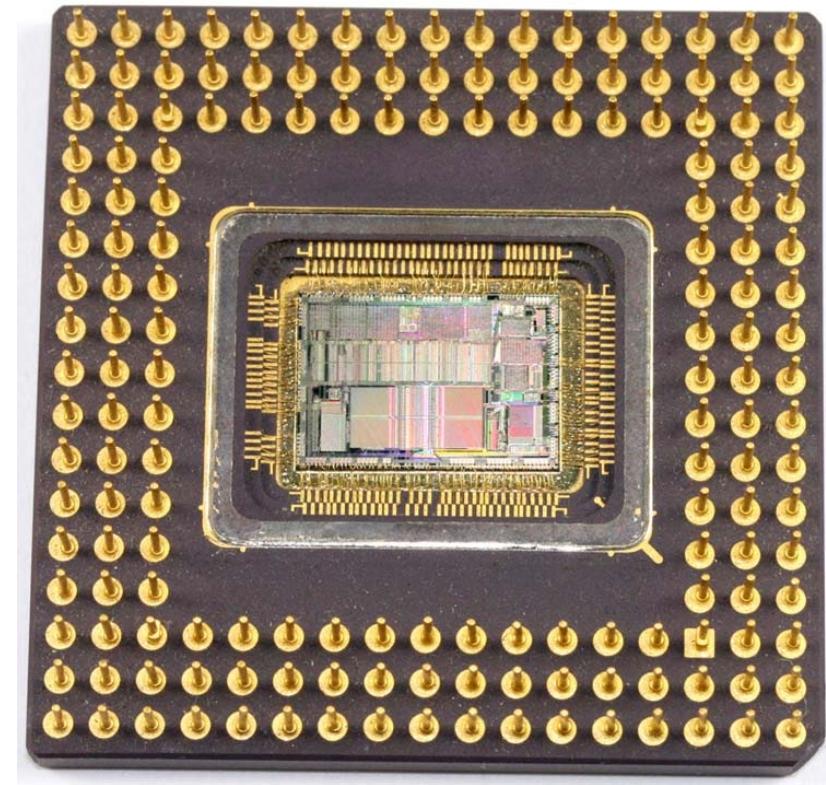
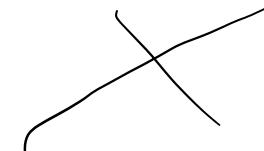
- Passaggio a 32 bit
- Supporto SO moderni
- Versioni DX/SX/SL
- Sempre retrocompatibile
 - Pro e contro

X



Intel 80486

- 80386 più veloce
- Integra definitivamente coprocessore
- Cache interna
- Pipeline



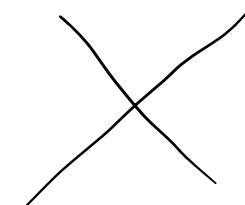
Intel Pentium

- La Intel scopre che i numeri non sono brevettabili
- Doppia pipeline
- Tecnologia MMX



Intel Pentium Pro

- Ramo collaterale del pentium
- Orientato ai server
- Ottimizzato per codice a 32 bit
- 5 pipeline
- Doppia cache interna
 - Doppio chip
- No MMX



Intel Pentium II e III

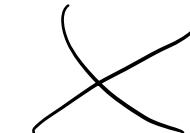
- Doppia cache su singolo chip
- Da MMX a SSE

X



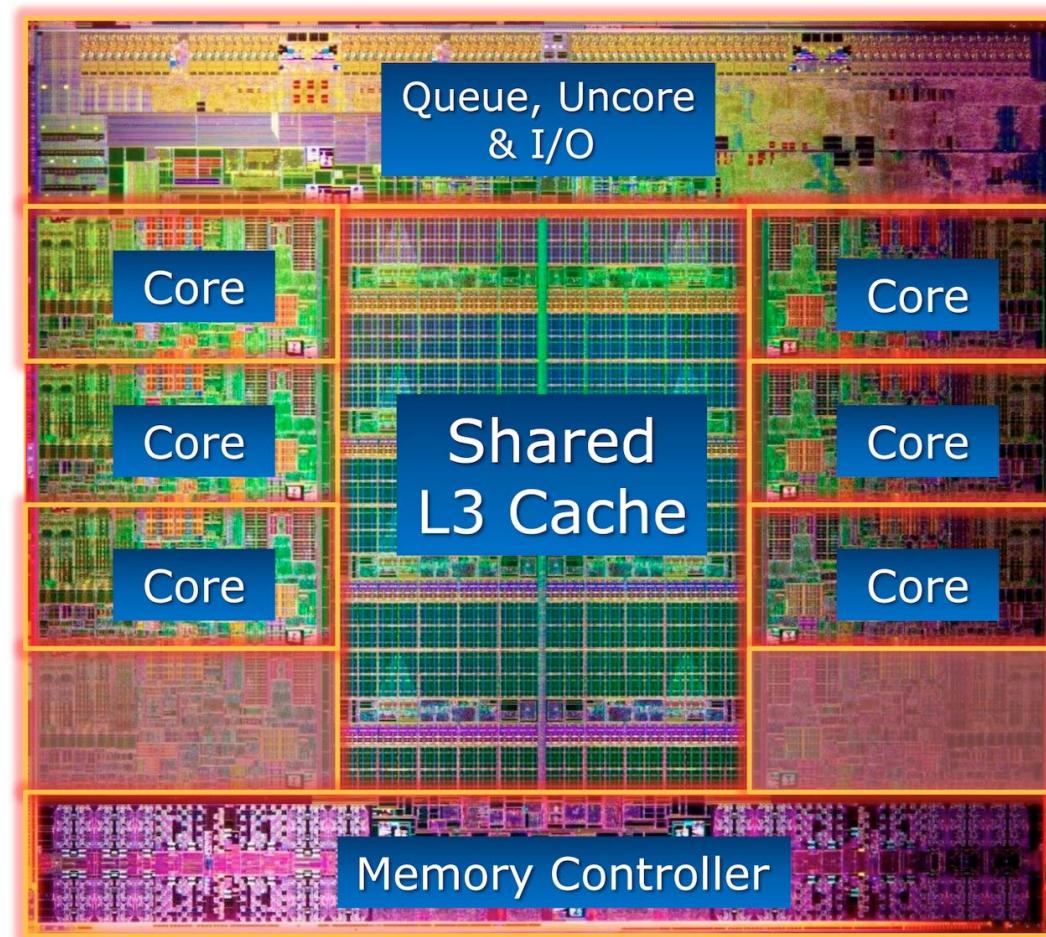
Intel Pentium 4

- SSE2
- EM64T - Extended Memory 64 Technology
- Nuovo sviluppo mirato ad aumentare la frequenza
 - Ma c'è un limite!
- Hyperthreading
 - Precede il multicore



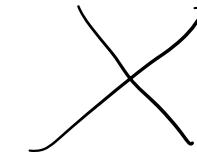
Intel Core 2/ix

- Piena tecnologia multicore



Panoramica non completa

- Versioni mobile o economiche
 - Celeron
- Versioni native a 64 bit (IA-64)
 - Itanium
 - Itanium 2
- Intel compatibili
 - AMD
 - Cyrix
 - ...



Intel: riepilogo

4004	4/1971	0.108	2300	640	First microprocessor on a chip
8008	4/1972	0.108	3500	16 KB	First 8-bit microprocessor
8080	4/1974	2	6000	64 KB	First general-purpose CPU on a chip
8086	6/1978	5–10	29,000	1 MB	First 16-bit CPU on a chip
8088	6/1979	5–8	29,000	1 MB	Used in IBM PC
80286	2/1982	8–12	134,000	16 MB	Memory protection present
80386	10/1985	16–33	275,000	4 GB	First 32-bit CPU
80486	4/1989	25–100	1.2M	4 GB	Built-in 8-KB cache memory
Pentium	3/1993	60–233	3.1M	4 GB	Two pipelines; later models had MMX
Pentium Pro	3/1995	150–200	5.5M	4 GB	Two levels of cache built in
Pentium II	5/1997	233–450	7.5M	4 GB	Pentium Pro plus MMX instructions
Pentium III	2/1999	650–1400	9.5M	4 GB	SSE Instructions for 3D graphics
Pentium 4	11/2000	1300–3800	42M	4 GB	Hyperthreading; more SSE instructions
Core Duo	1/2006	1600–3200	152M	2 GB	Dual cores on a single die
Core	7/2006	1200–3200	410M	64 GB	64-bit quad core architecture
Core i7	1/2011	1100–3300	1160M	24 GB	Integrated graphics processor

Intel: ultimi processori (5a generazione)

Processor Number	Cores/Threads	Base Freq (GHz)	Intel Turbo Boost Technology 2.0			Graphics	Graphics Base / Max Freq (MHz)	LPDDR3 Memory Speed Support (MHz)	DDR3L Memory Speed Support (MHz)	L3 Cache	TDP	cTDP Down	Tj (deg)	Intel SIPP	Intel Technologies				Package Type	1ku Pricing	
			Max Single Core Turbo (GHz)	Max Dual Core Turbo (GHz)	Max Quad Core Turbo (GHz)										Intel vPro	Intel TXT	Intel VT-d	Intel VT-x	AES-NI		
Intel Core™ i7 Processors (U-Processor Line)																					
i7-5650U	2/4	2.2	3.2	3.1	N/A	Intel® HD Graphics 6000	300/1000	1866	1600	4MB	15W	9.5W	105	2015	✓	✓	✓	✓	✓	BGA	\$426
i7-5600U	2/4	2.6	3.2	3.1	N/A	Intel® HD Graphics 5500	300/950	1600	1600	4MB	15W	7.5W	105	2015	✓	✓	✓	✓	✓	BGA	\$393
i7-5550U	2/4	2.0	3.0	2.9	N/A	Intel® HD Graphics 6000	300/1000	1866	1600	4MB	15W	9.5W	105			✓	✓	✓	✓	BGA	\$426
i7-5500U	2/4	2.4	3.0	2.9	N/A	Intel® HD Graphics 5500	300/950	1600	1600	4MB	15W	7.5W	105			✓	✓	✓	✓	BGA	\$393
Intel Core™ i5 Processors (U-Processor Line)																					
i5-5350U	2/4	1.8	2.9	2.7	N/A	Intel® HD Graphics 6000	300/1000	1866	1600	3MB	15W	9.5W	105	2015	✓	✓	✓	✓	✓	BGA	\$315
i5-5300U	2/4	2.3	2.9	2.7	N/A	Intel® HD Graphics 5500	300/900	1600	1600	3MB	15W	7.5W	105	2015	✓	✓	✓	✓	✓	BGA	\$281
i5-5250U	2/4	1.6	2.7	2.5	N/A	Intel® HD Graphics 6000	300/950	1866	1600	3MB	15W	9.5W	105			✓	✓	✓	✓	BGA	\$315
i5-5200U	2/4	2.2	2.7	2.5	N/A	Intel® HD Graphics 5500	300/900	1600	1600	3MB	15W	7.5W	105			✓	✓	✓	✓	BGA	\$281

ARM

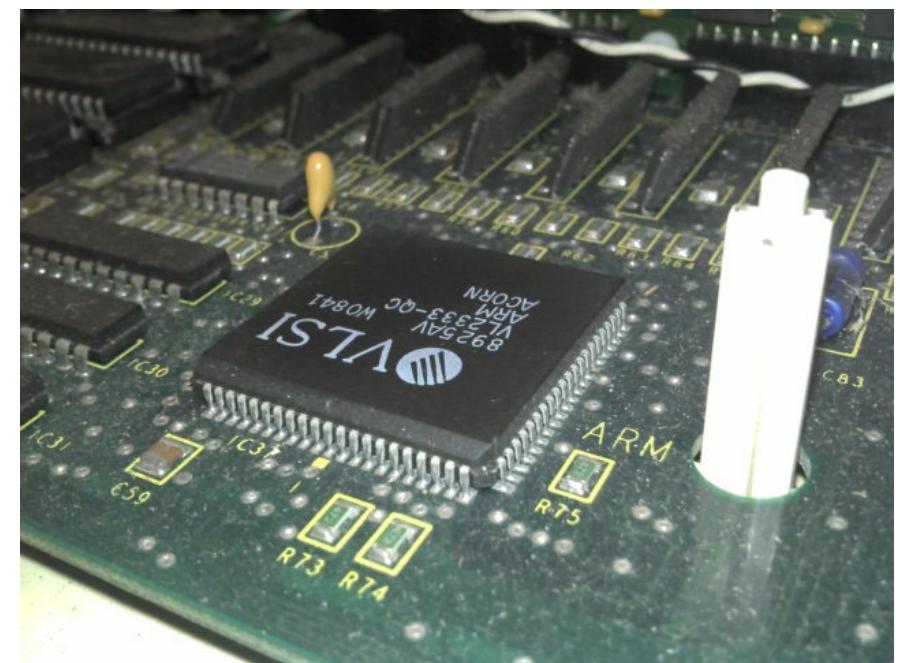
- Acorn Computer
 - Nasce negli anni 80
 - Sviluppano le idee di Berkeley RISC
 - Inizialmente: Acorn RISC Machine
 - Non sono produttori
 - Intel

X

ARM1 e ARM2

- ARM1
 - Architettura a 32 bit
 - Indirizzamento a 26 bit
 - 64 Mbyte di memoria
- ARM2
 - Acorn Archimedes
 - Buon successo

X



ARM 610/710 / ARMv3

- 1993
- 32 bit
- 33 MHz
- 4 kbyte cache L1
- Advanced RISC Machine
 - Apple Newton

X



StrongARM

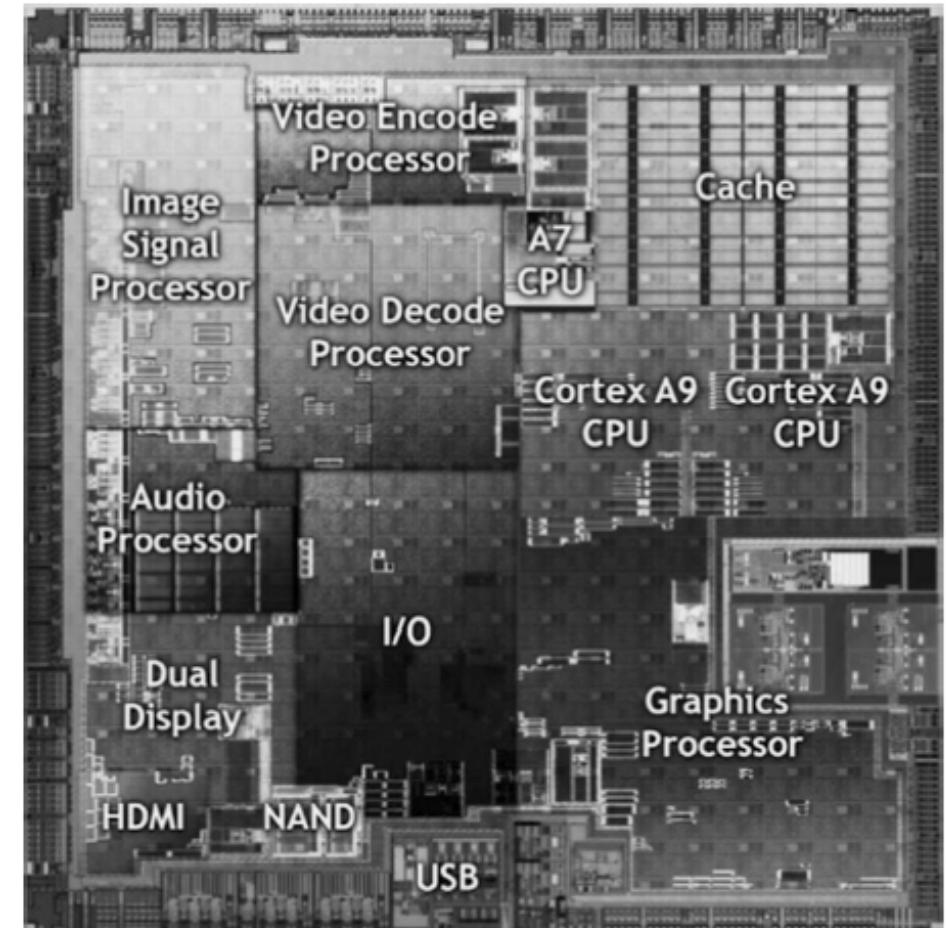
- 1995
- Cooperazione con DEC
- IS anche a 16 bit
- Passo fondamentale verso ambienti embedded/mobile
- 233 MHz
- 1 Watt
 - PDA
 - Router
 - Dispositivi multimediali/Game Boy

X



ARM Cortex

- 2005
- Blocchi funzionali
 - Non sempre tutti attivi
- Unità SIMD
- Floating Point migliorato
- Versioni singolo e multicore
- Pipeline
- Versioni Microcontroller,
- Application, RealTime
- Tablet/Cellulari
- Harvard/Von Neumann



SoC Tegra2 NVIDIA

Esempio di microarchitettura di CPU

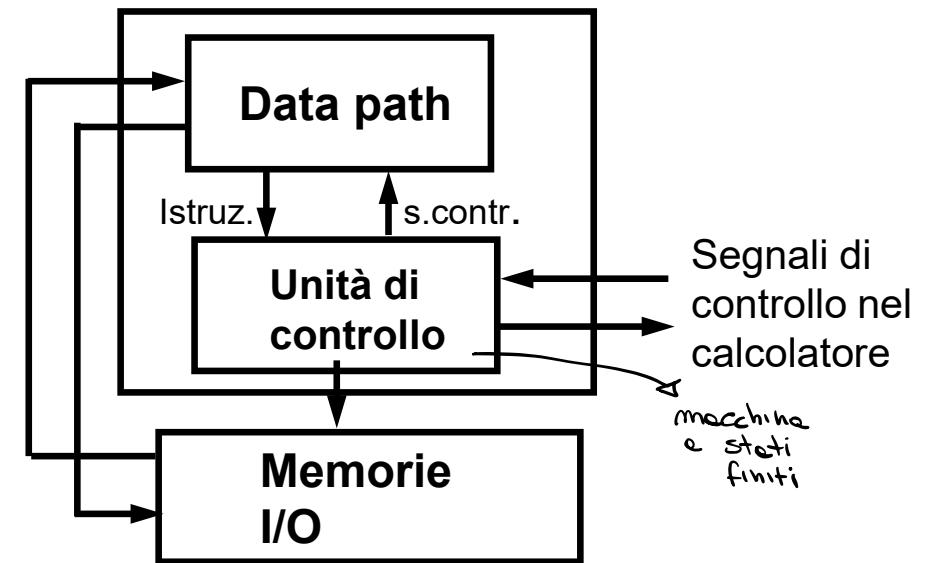
Nel seguito della lezione vedremo lo sviluppo e l'evoluzione dell'architettura di una semplice CPU.

L'evoluzione avviene attraverso tre passi successivi che consentono un significativo aumento delle prestazioni:

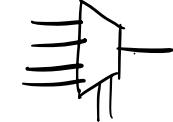
- Architettura monociclo
- Architettura multiciclo
- Architettura pipeline.

La CPU

- Dal punto di vista strutturale la CPU è composta da alcune parti **combinatorie**:
 - ALU, decodifica degli indirizzi per i registri interni
 - decodifica delle istruzioni,...
- e da alcune parti **sequenziali**
 - REGISTRI di dato e controllo
 - Unità di Controllo (Control Unit)
- Dal punto di vista funzionale si divide in
 - **Data path** (o **execution unit**): il percorso dei dati
 - **Control unit** (l'unità di controllo)
- l'unità di elaborazione acquisisce istruzioni e dati, passa le istruzioni alla unità di controllo, esegue le operazioni di ALU, genera i risultati, calcola il prossimo indirizzo...
- la unità di controllo è una FSM che controlla l'unità di elaborazione ed i segnali con l'esterno
- L'implementazione a livello di reti logiche e la progettazione a livello RTL (corrispondente alla microarchitettura) è dipendente dalla famiglia delle CPU e dalla casa costruttrice

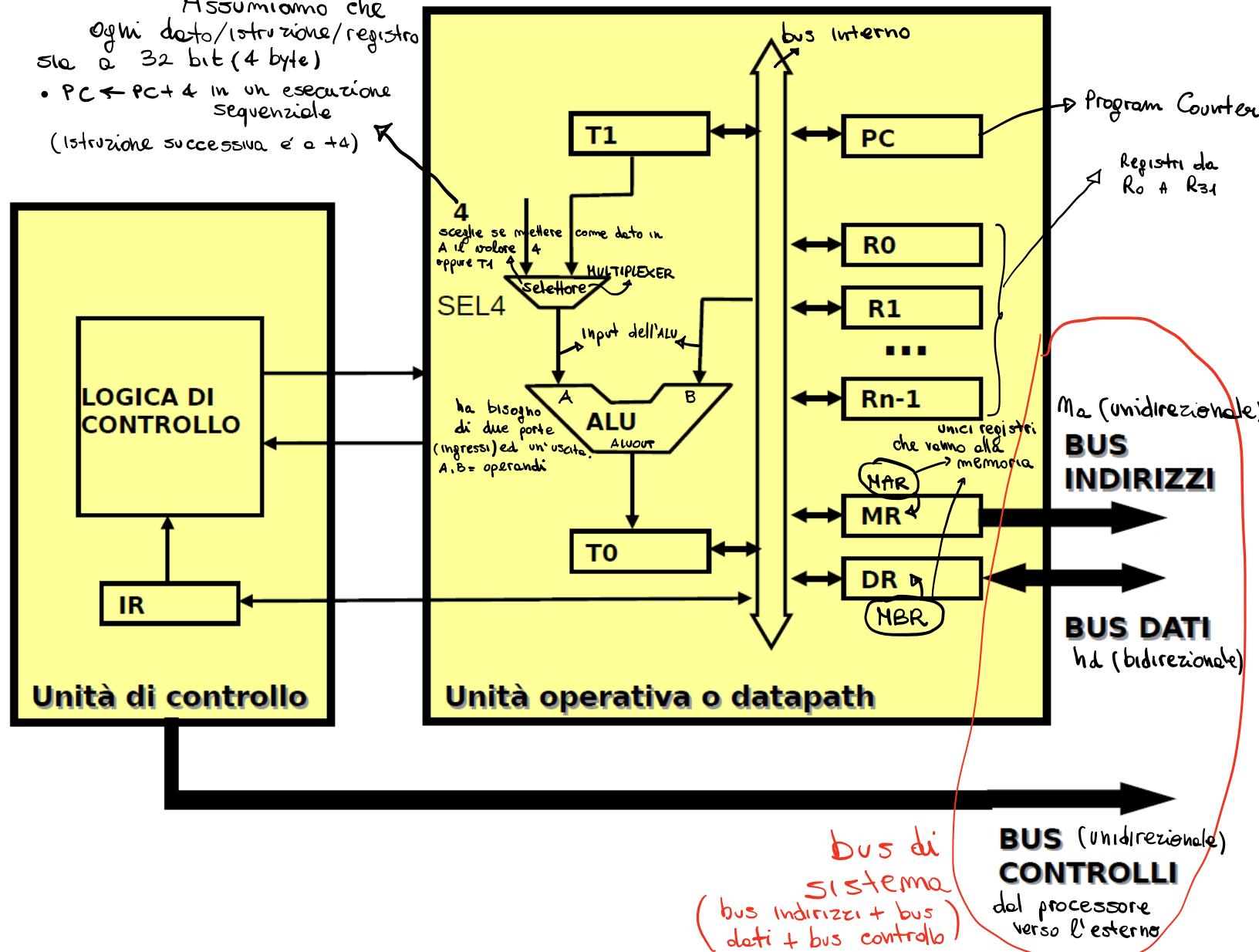


MULTIPLEXER 4:1



Esempio di microarchitettura di CPU – RISC PURO

Assumiamo che
ogni dato/istruzione/registro
sia a 32 bit (4 byte)
• $PC \leftarrow PC + 4$ in un'esecuzione
sequenziale
(istruzione successiva è a +4)

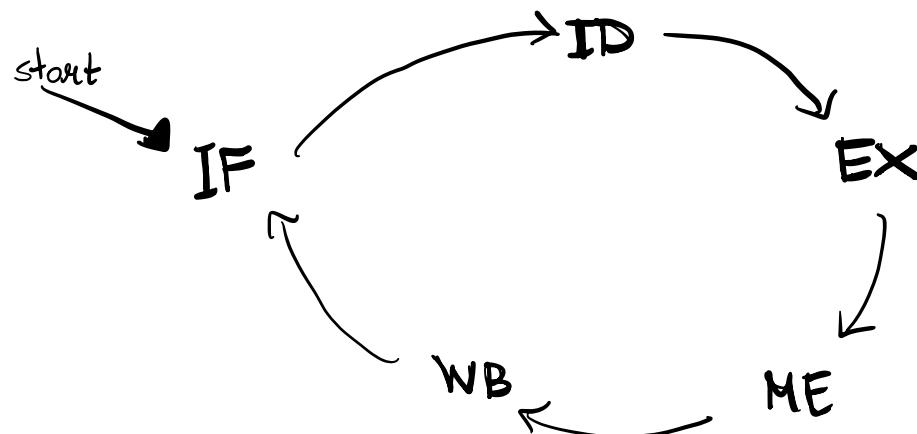


Architettura di riferimento

- Architettura di tipo RISC a 32 bit
 - Gli indirizzi di memoria sono riferiti ai byte
 - Istruzioni e dati occupano sempre e solo una parola di 32 bit
 - Istruzioni e dati si trovano in memoria allineate a indirizzi multipli di 4 (0, 4, 8, 12, ...) → ortogonali
 - La regola di aggiornamento del registro PC è:
 - $PC = PC + 4$ (salti esclusi)
 - 32 registri di uso generale di 32 bit (R0, R1, ..., R31)
 - Register file
 - Repertorio di istruzioni estremamente ridotto

Fasi di esecuzione

- Macchina a stati , ciclico :
 - **IF** (fetch): l'istruzione viene caricata nella memoria
 - **ID** (decode): l'istruzione si decodifica (combinatoria)
 - **EX** (esecuzione): calcolo/azione (datapath + ALU)
 - **ME** (memoria): accesso ai dati in memoria (Read o Write)
 - **WB** (write-back): operazione di scrittura su registro

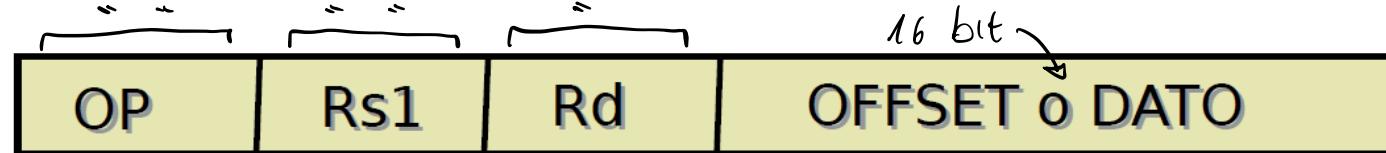


Formato istruzioni

- Un possibile formato istruzioni è il seguente:



Es: ADD R4,R2,R5 ; R5 <- R4 + R2



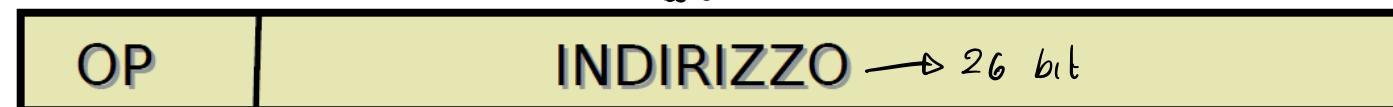
entro il
suo codice
operativo

Es: SUBI R4,R2,4 ; R2 <- R4 - 4 \rightsquigarrow R₂ = R_d
R_{S1} = R₄
DATO = 4

LOAD R1,20(R6) ; R1 <- M[20 + R6]
R_d offset + Reg. Base \rightsquigarrow legge in memoria ciò che c'è in R₆ + 20 e salva in R₁

soltanto condizionato
(come se fosse
un if)

JE R2,R3,offset ; se (R2=R3) allora PC <- PC+offset
if se il loro valore



Es: JP 804 ; PC <- 804

JP = soltanto incodizionato
di quanto? In base ai 26 bit
più significativi

Semplificazioni

- Assumiamo:
 - L'assenza di stack
 - Che la chiamata a funzioni avvenga mediante salto specifico che memorizzi il PC in uno dei registri
 - $JAL XYZ \rightarrow R31=PC, PC=XS(XYZ)$
 - Che gli indirizzi siano quelli effettivi
 - In realtà poco efficiente visto che la memoria viene indirizzata a 4

Memoria LIFO

si solleverà il
registro X il return f in R31
prima di
soltare JAL

(JP (Incondizionato)
me con RETURN)

Istruzioni aritmetico/logiche

- Supponiamo:
 - OP è univoco
 - Coinvolti solo registri (no operazioni aritmetico/logiche con la memoria → macchine **load-store**)
 - Funzione ALU contiene la specifica operazione:
 - ADD, SUB, MUL, DIV, IDIV, AND, ...
- non eccedono
alla memoria*



$A, B \Rightarrow$ porte d'ingresso
e 32 bit

ALU

OPALU

Dato

32

A

 $\begin{cases} Zf = 1 \\ Sf = 0 \end{cases}$

> segno positivo

Dato

32

B

 $\begin{cases} Zf = 0 \\ Sf = 1 \end{cases}$

<

 $\begin{cases} Zf = 0 \\ Sf = 0 \end{cases}$

≠



- Flag - Zero

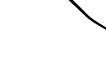
- lo 0 diventa 1 se l'ultima operazione ha dato 0
- lo 1 viceversa

- Flag - Segno

- - se l'ultima operazione positiva
- + viceversa

FLAG - singoli bit (0 oppure 1)

 Zero
Segno

 per il
 $>, <, =, \neq$
 \geq, \leq


Risultato

32

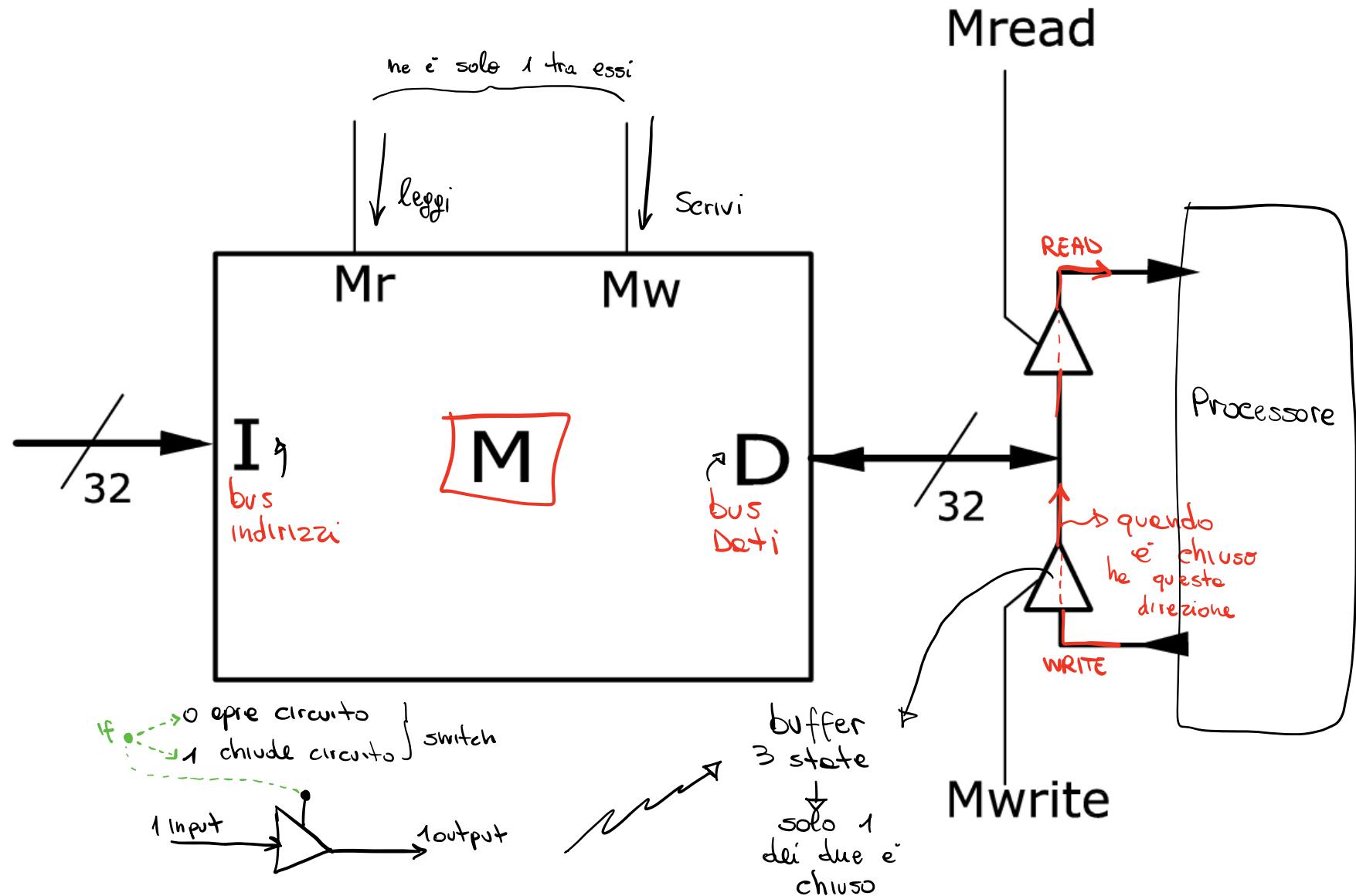
 ALUOUT
(o 32 bit)

Load (LD) e Store (ST)

- Permettono lettura e scrittura da memoria
- R_b = registro base
- R_{ds} = registro destinazione/sorgente
- Offset = scostamento rispetto a R_b
 - LOAD R13, R6, 100 $R6 \leftarrow M[R13+100]$
 - ST R4, R2, 20 $M[R4+20] \leftarrow R2$



Accesso alla memoria



Istruzioni di controllo

- Salti (e chiamate a funzioni)
- Se ne devono prevedere di differenti
 - JP: Salto incondizionato
 - JAL: Chiamata a funzione
 - JE: Salto condizionato egualanza (0 FLAG)
 - JS: Salto condizionato a segno ($SIGN \text{ FLAG}$)
 - JR: Salto relativo → salta all'indirizzo
contenuto in un indirizzo
 - ...

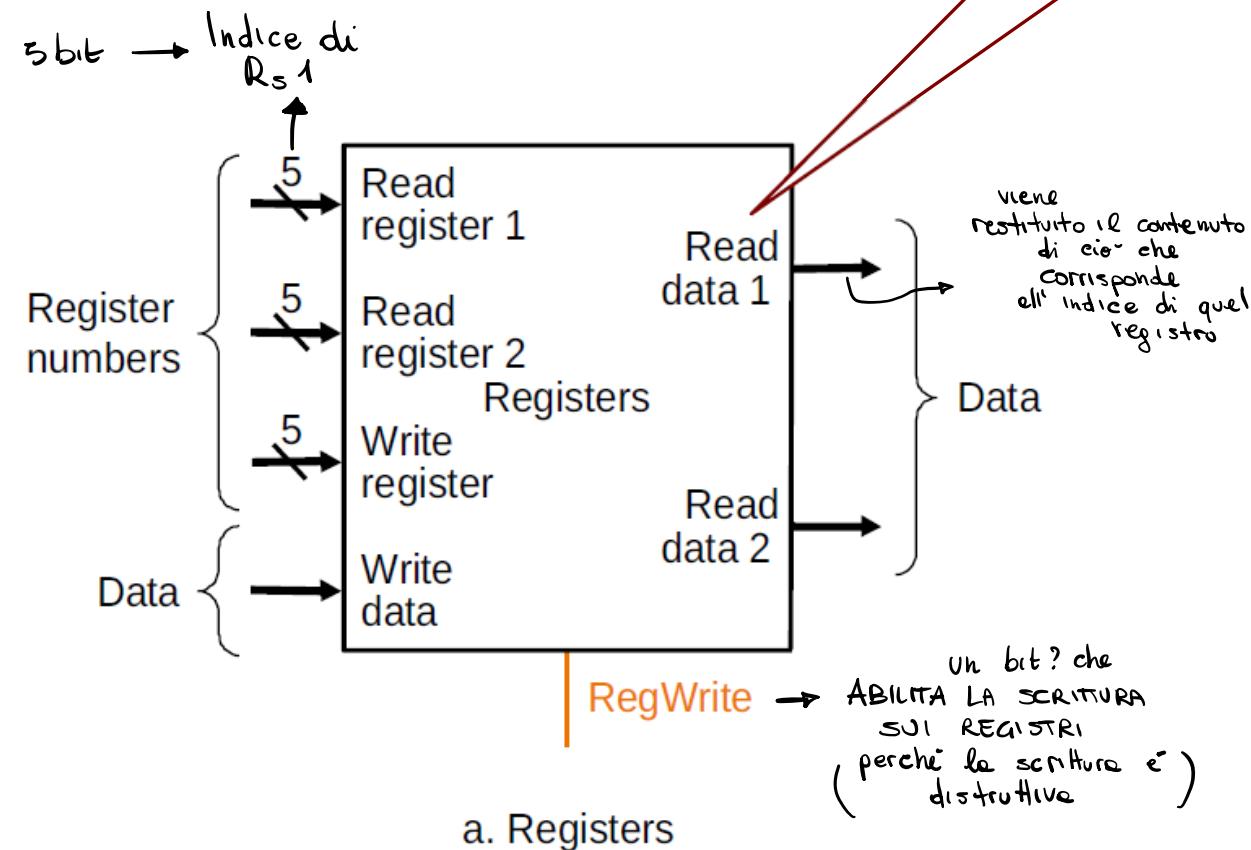
x operatori
relazionali

Register file

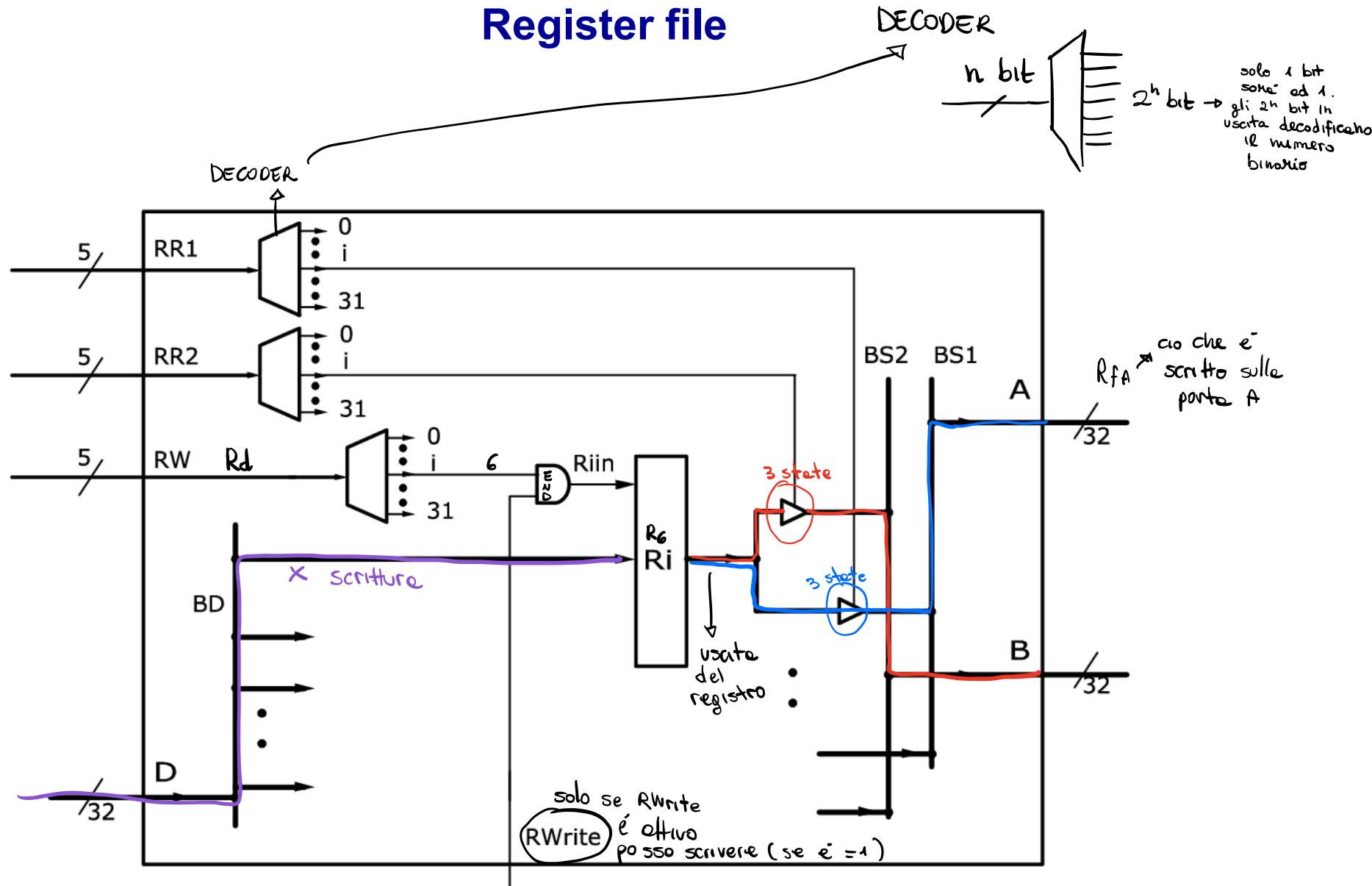
- Array di registri
- Tipicamente indifferenziati

- Array di Registri (32 Reg. con 32bit)
- Rete Logica di Registra

Read e Write sono aggettivi, non verbi



Register file



Approcci possibili

- **CPU monociclo**

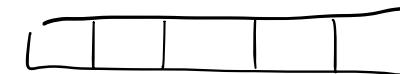
- Ogni istruzione richiede 1 ciclo di clock
 - Semplice <> Lenta



- **CPU multiciclo**

- Ogni fase di istruzione richiede 1 ciclo di clock
 - Complessa <> Ottimizza tempi

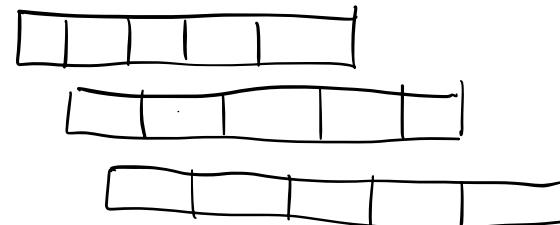
→ FETCH, DECODE, ECC ...



- **CPU pipeline**

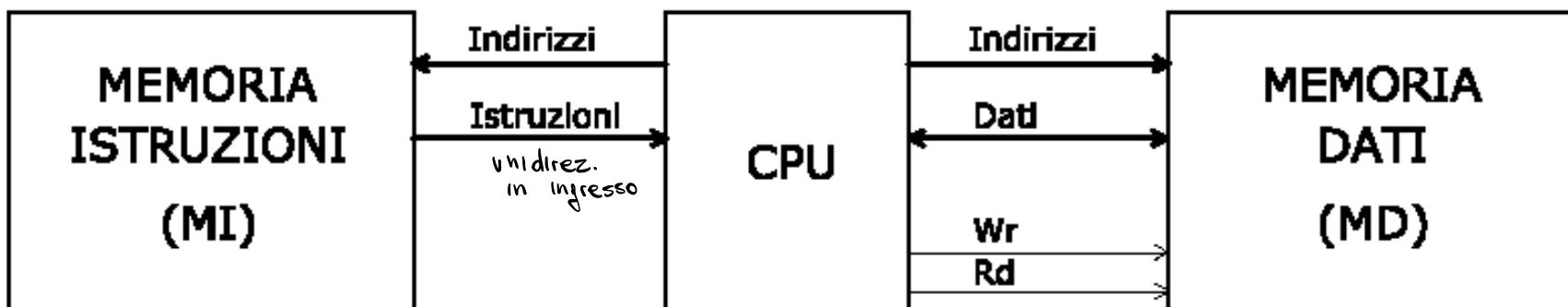
- Evoluzione precedente
 - Non si aspetta il completamento di una istruzione
 - Fasi eseguite in parallelo

perizie
parallelismo
istruzioni



CPU monociclo

- Esegue istruzioni in singolo colpo di clock
- Inizialmente: modello di Harvard



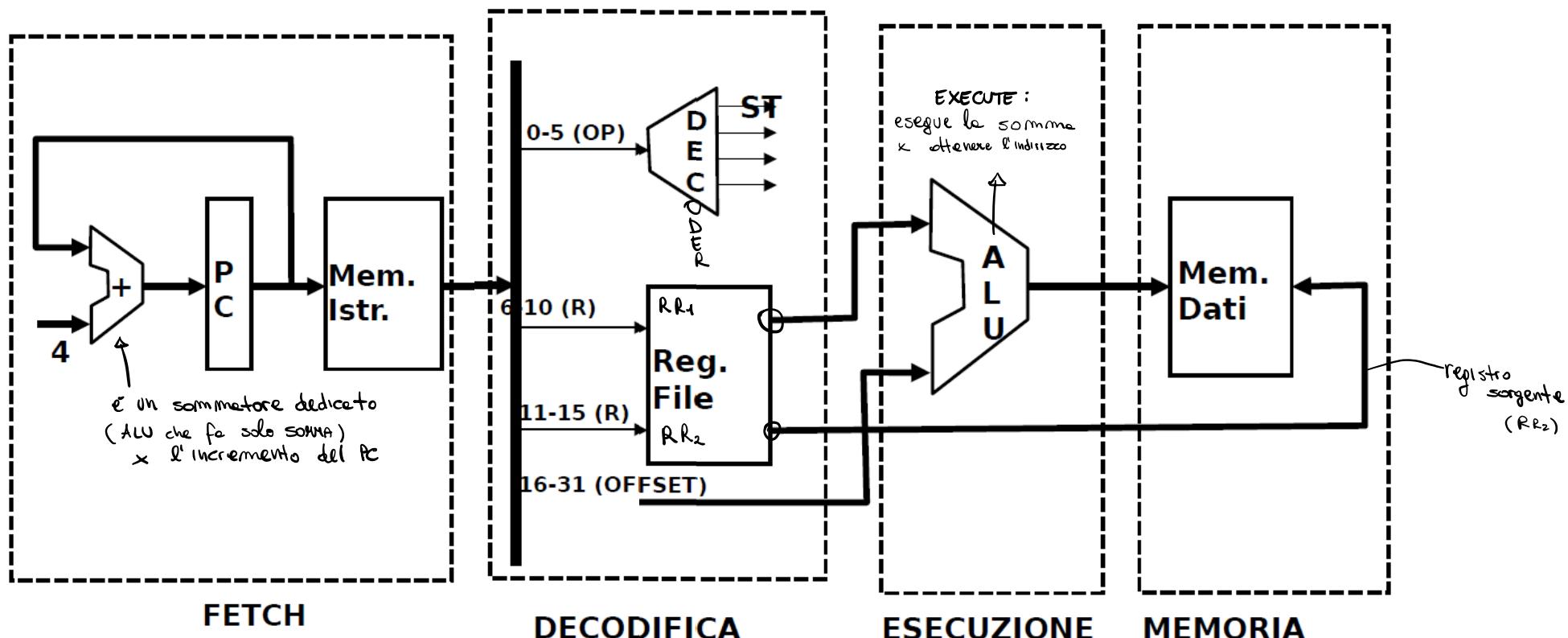
Architettura monociclo: esempio

- Supponiamo di dover fare: ST R6, R1, 20
- Mostrati solo gli elementi interessati

(store) $\Rightarrow M[R6+20] \leftarrow R1$

offset
registro base
o sorgente

\Rightarrow tutto questo è eseguito in un ciclo di clock

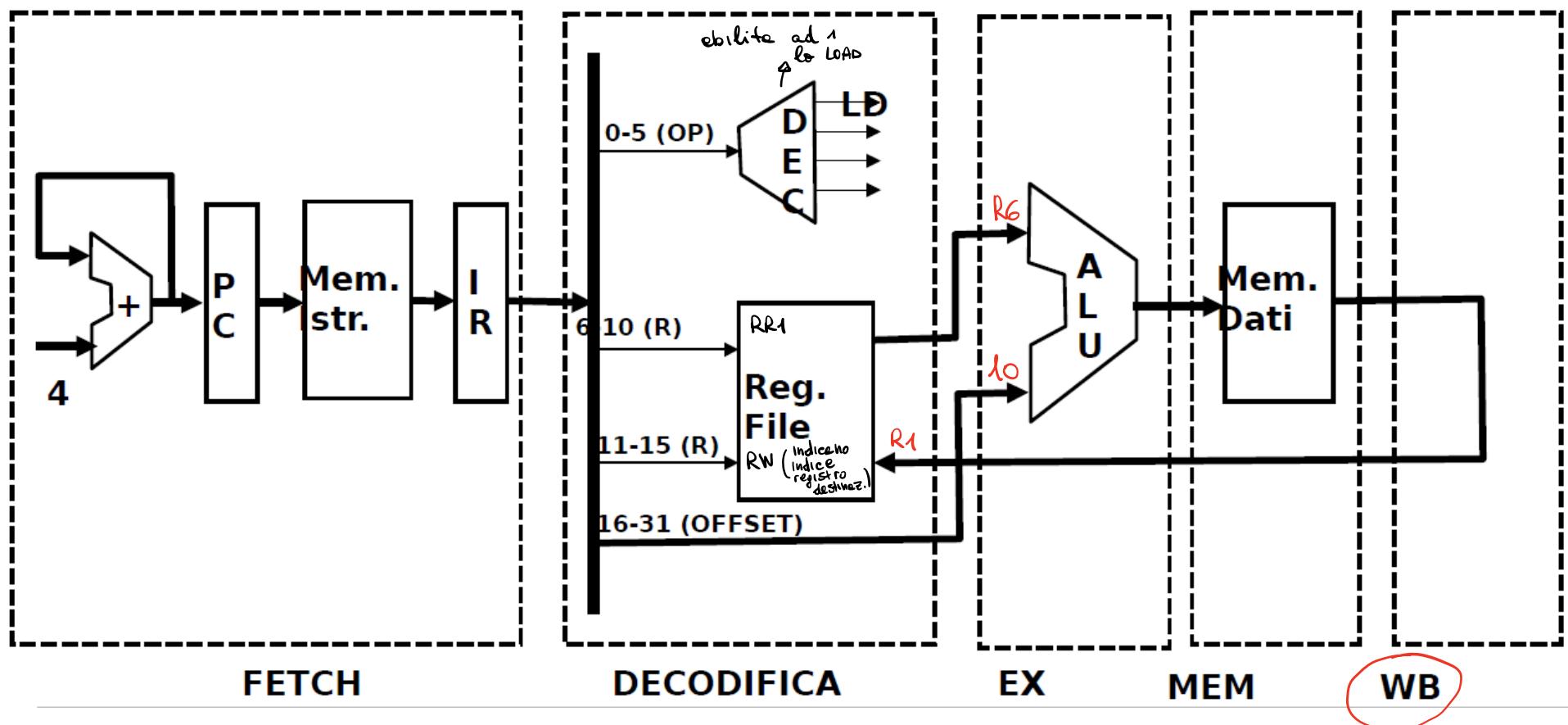


$\begin{cases} IR \leftarrow M[PC] \\ PC \leftarrow PC + 4 \end{cases}$
 presupponendo
 che non ci sia
 un salto

lo metto sul
bus degli indirizzi

Architettura monociclo: memoria unificata

- L'esempio visto ricalca l'architettura di Harvard. Vediamo un altro esempio: LD R6, R1, 10 (Load) $R_6 \leftarrow M[R_6 + 10]$



Architettura monociclo: considerazioni

- Non tutte le istruzioni richiedono gli stessi moduli
- Esempio realistico (ns)

usa tutte le fasi ←

	FETCH	DECOD	ALU	MEM	WB	TOTALE
ARITM	30	5	12		5	52
LD → la più complessa	30	5	12	30	5	82
ST	30	5	12	30		77
JP COND.	30	5	12			47
JMP (incond.)	30	5				35
JAL/CALL (solto l'indir. coh (ritorno in R31))	30	5			5 x scrivere l'indice per R31	40

- Alcune note: *salvo che sempre uguali*
 - ARITM non richiede MEM perchè load-store
 - ST non richiede WB perchè memorizza il risultato in memoria
 - JMP modifica PC direttamente nella fase DECOD
 - JP COND richiede fase ALU per verificare la condizione
 - JAL ha anche WB per scrivere nel registro interno l'indirizzo di ritorno

Dato un programma con N istruzioni il tempo di esecuzione sarà $T_{CPU} = N \cdot 82 \text{ ns}$ (*non la soluzione migliore, la più semplice ma la più lenta*)

Architettura multiciclo

- In una architettura multiciclo ogni stadio opera in un ciclo di clock e il periodo viene dimensionato in riferimento allo stadio più lento.
punte da queste idee
- Non tutte le istruzioni utilizzano lo stesso numero di cicli. Alcune si concludono in un numero inferiore rispetto a quelle che attraversano tutti gli stadi (es: la ST si conclude dopo la scrittura in memoria, no WB)
- In altre parole, dato $\sum T_{multi}$ il tempo per eseguire un'istruzione nell'architettura multiciclo e T_{mono} il tempo per l'architettura monociclo, ci saranno istruzioni in cui $\sum T_{multi} < T_{mono}$ e altre in cui $\sum T_{multi} > T_{mono}$
(sicuramente l'istruzione più lenta, cioè quella che richiede tutti i cicli di clock)

caso

T_{multi} (prestazioni migliori)

Architettura multiciclo

- I vantaggi della multiciclo dipendono fortemente dalla scelta degli stadi. Noi vediamo un semplice esempio con i 5 stadi visti sinora (T1 per IF, T2 per ID, T3 per EX, T4 per ME e T5 per WB)
- Non sommatore dedicato per fare $PC \leftarrow PC + 4$
- L'unità di controllo è un automa a stati finiti
- Le varie sotto-unità possono essere utilizzate in fasi diverse:
 - L'ALU è unica per le operazioni di calcolo e per incrementare il PC.
 - La memoria non deve essere separata perché fetch e lettura e scrittura sono eseguite in fasi diverse (architettura di Von Neumann, diversamente da prima!)

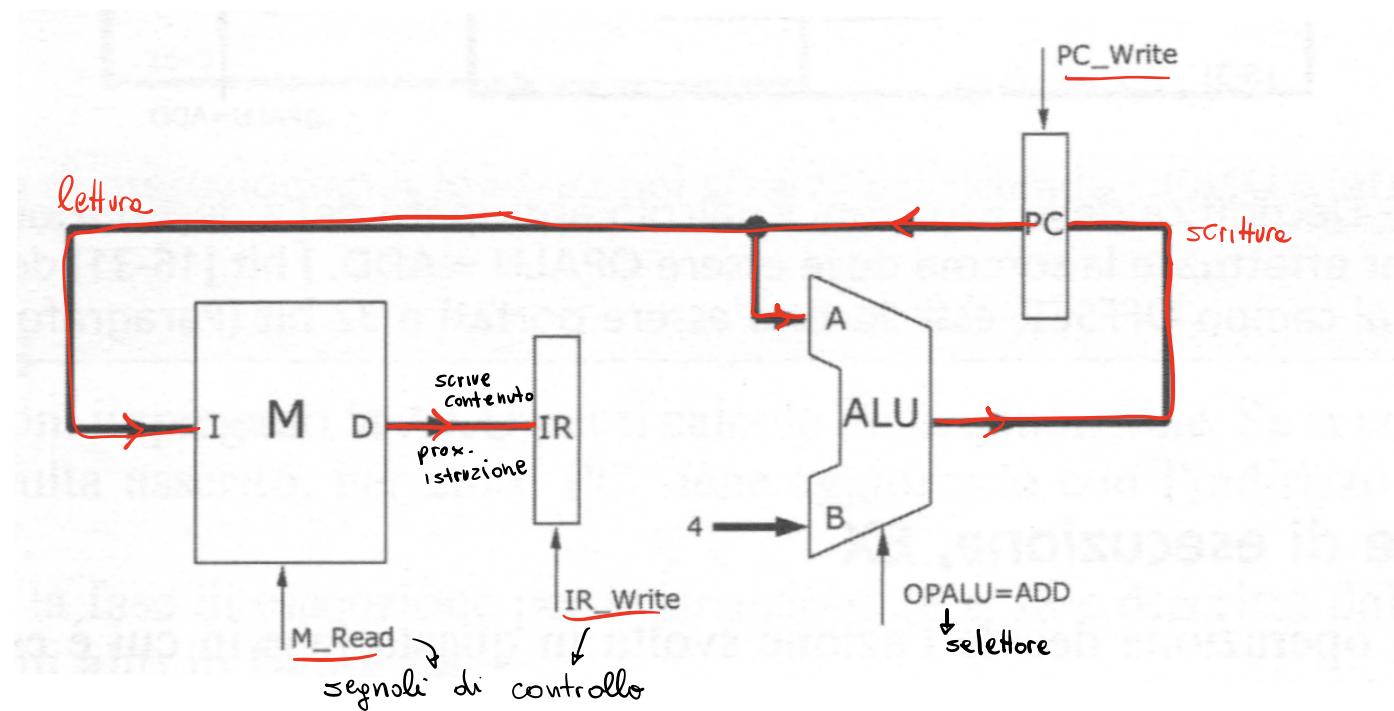
Architettura multiciclo: IF

- Leggo in memoria quanto puntato da PC (program counter)
- Per ora non consideriamo i salti (che modificano PC)
- 2 operazioni:

$\begin{cases} \text{IR} \leftarrow M[PC] \\ PC \leftarrow PC + 4 \end{cases}$ ← approfitto che l'ALU in T1 non lavora

*pereché non
sta facendo nient'altro*

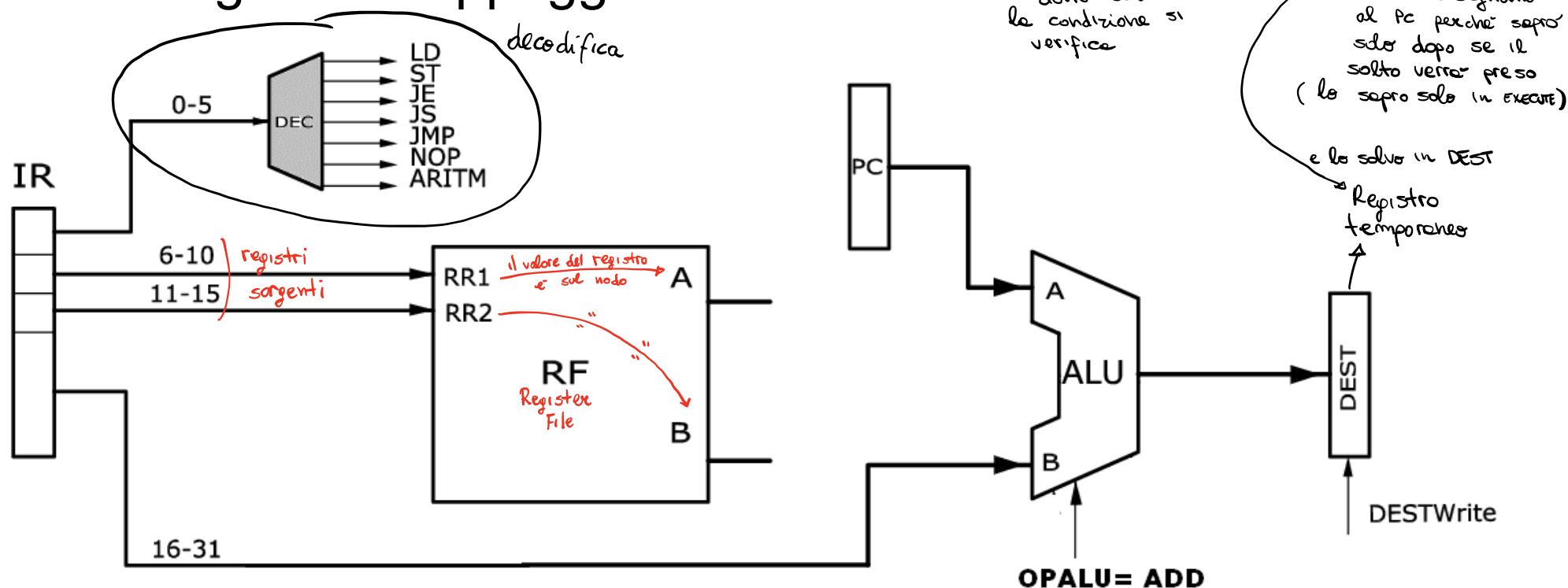
FETCH



Decode

Architettura multiciclo: ID

- Decodifica OP (come precedente)
- Codifica operandi presentata a RF che rende disponibili valori subito
- Sfrutto inoperatività ALU per calcolo salti condiz. JE/JJS
 - Registro di appoggio DEST



Architettura multiciclo: EX_{execute}

- Dipende da OP

– Aritmetiche : $ALUout \leftarrow A fALU B$

– LD/ST: $ALUout \leftarrow A + OFFSET$

– JE: $(\begin{matrix} \text{non lo} \\ \text{uso} \\ \text{questo} \\ \text{risultato} \end{matrix}) ALUout \leftarrow A - B; \text{ if Zero then } PC := DEST$
cioè che c'è sulla parte A,B dell'ALU

– JS: $ALUout \leftarrow A - B; \text{ if Segno then}$

$PC := DEST$

verifica
lo zero
FLAg

– JMP: $PC \leftarrow IND$

– JR : $PC \leftarrow RF.B$

– JAL : $PC \leftarrow IND$

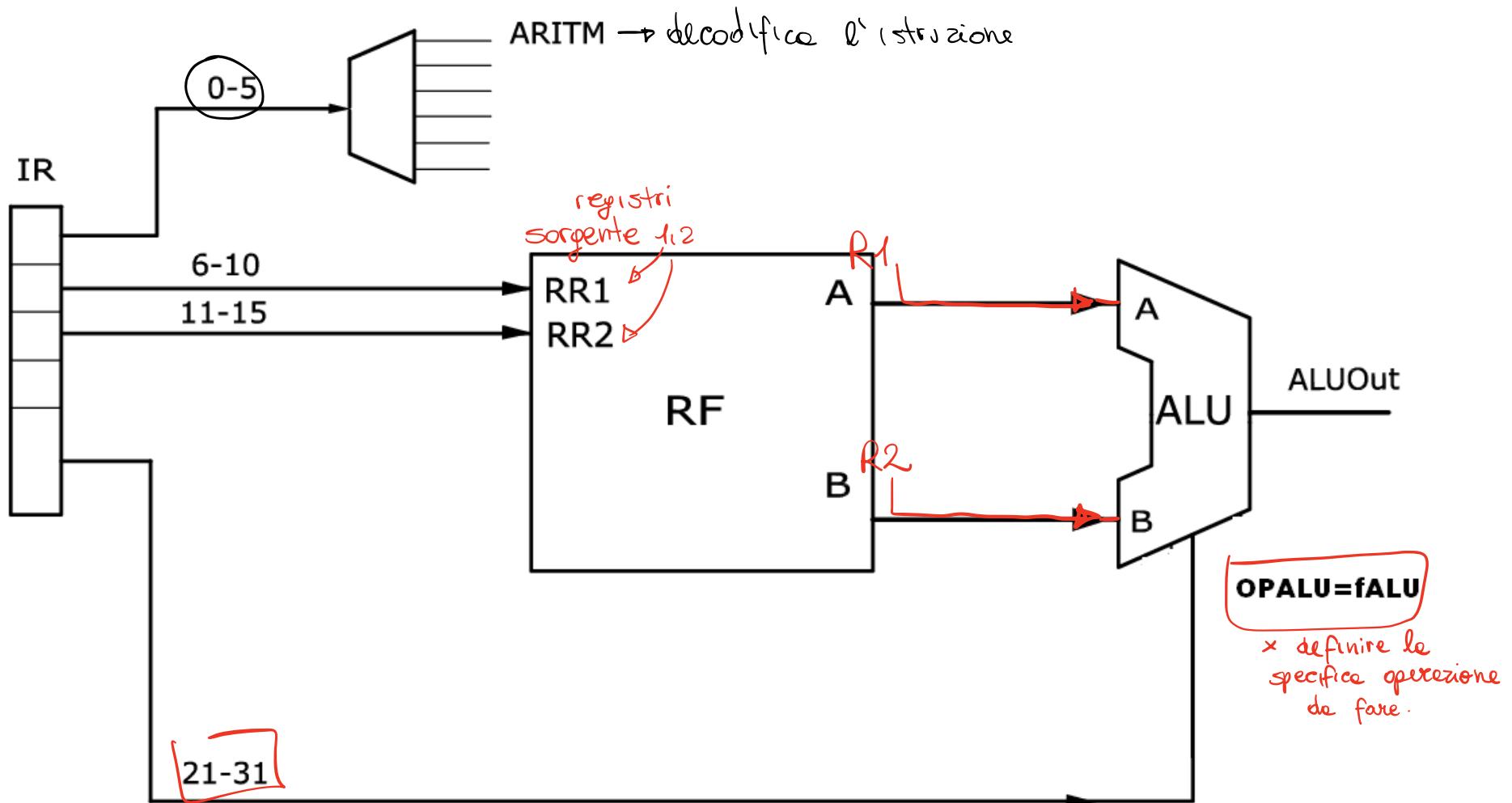
verifica
il sign Flag

Architettura multiciclo: EX

- Operazione di tipo aritmetico

$$\text{ALUout} \leftarrow A \text{ fALU } B$$

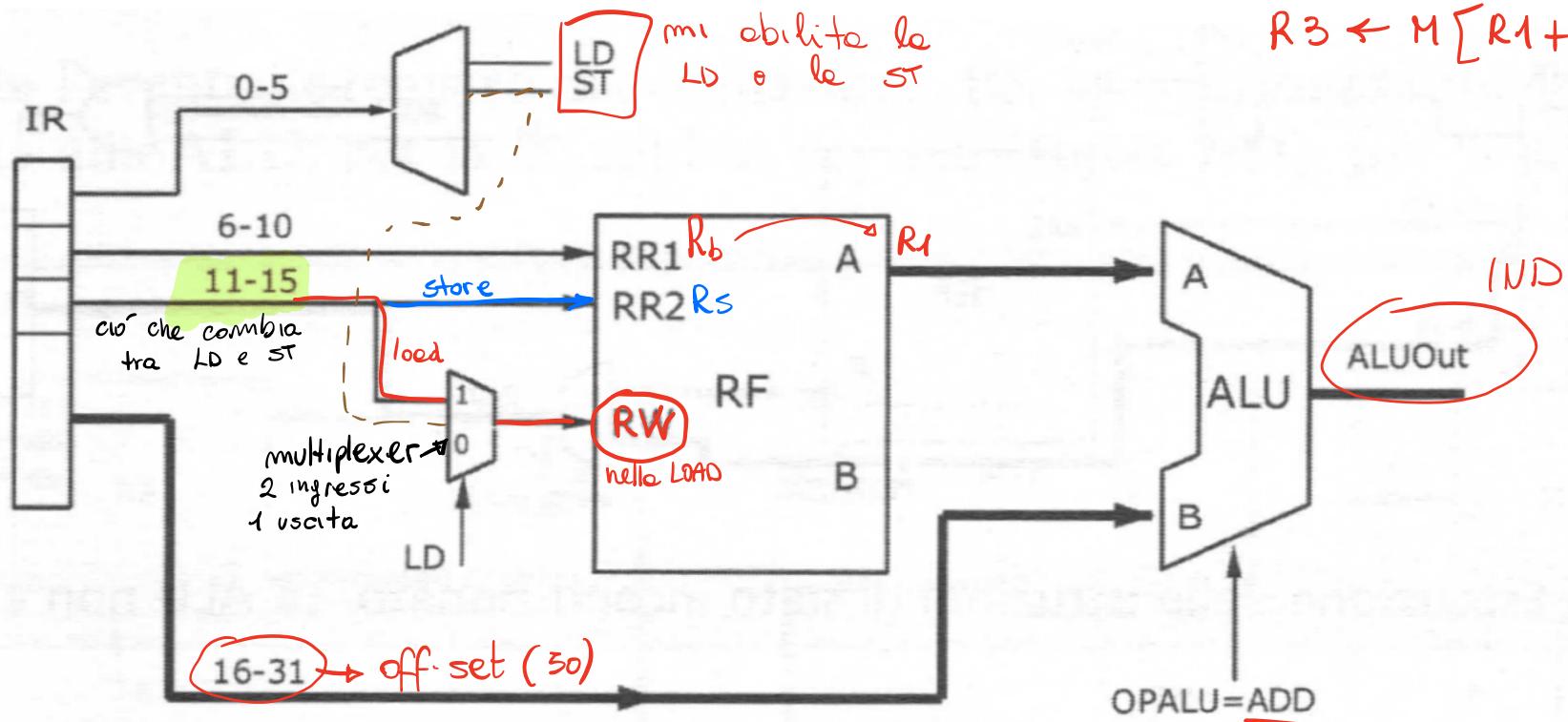
ADD R1, R2, R3
R3 \leftarrow R1 + R2



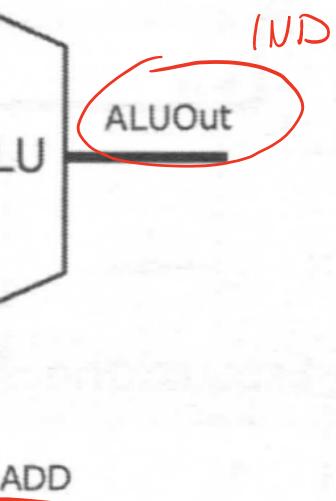
Architettura multiciclo: EX

- LOAD o STORE

$$\text{ALUout} \leftarrow A + \text{OFFSET}$$



LD R1, R3, 50
R3 $\leftarrow M[R1 + 50]$

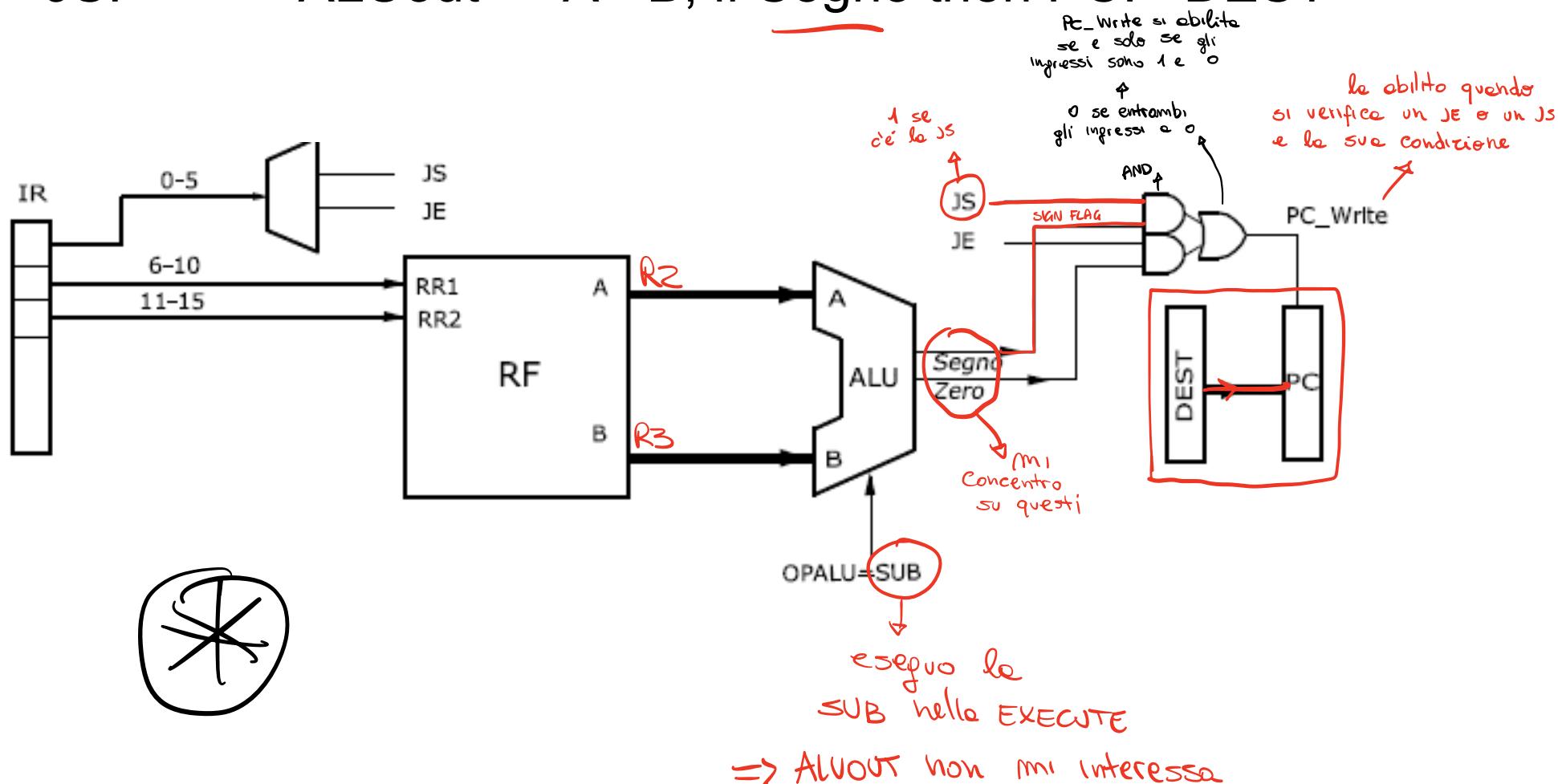


Architettura multiciclo: EX

- Salto condizionato JE/JJS

JE: $\text{ALUout} \leftarrow A - B$; if Zero then $\text{PC} := \text{DEST}$

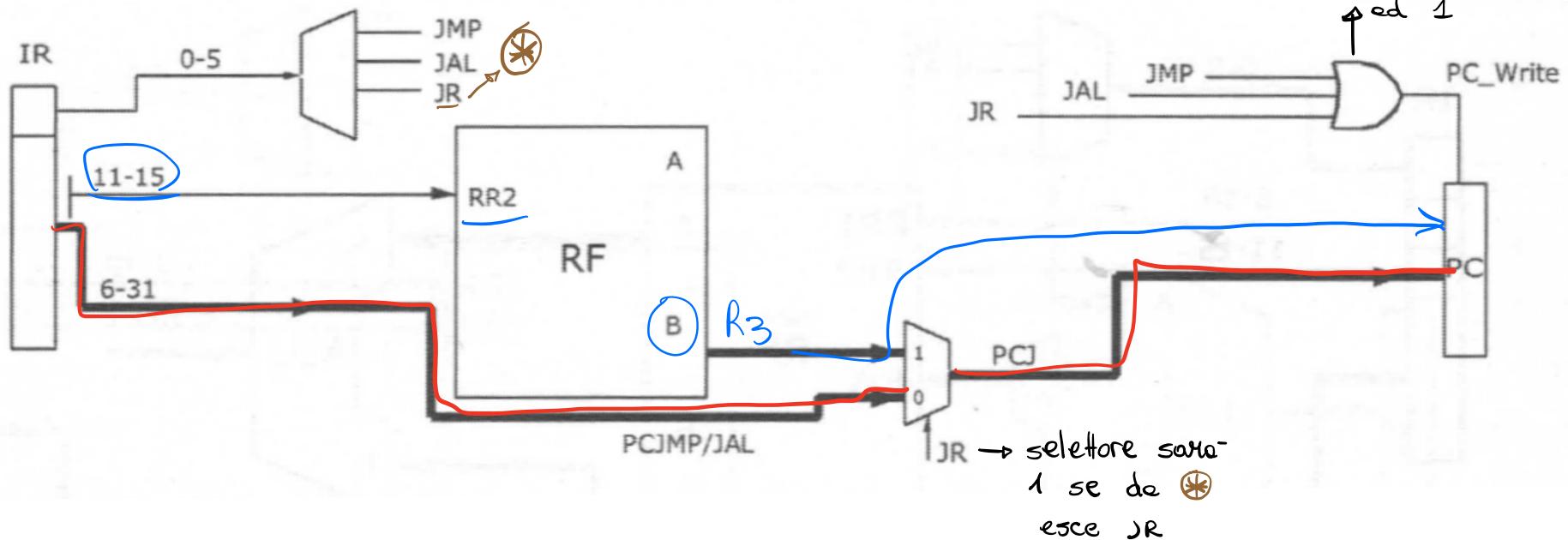
JS: $\text{ALUout} \leftarrow A - B$; if Segno then $\text{PC} := \text{DEST}$



Architettura multiciclo: EX

- Altri salti incondizionati

- JMP: $PC \leftarrow IND$
- JR : $PC \leftarrow RF.B$
- JAL : $PC \leftarrow IND$

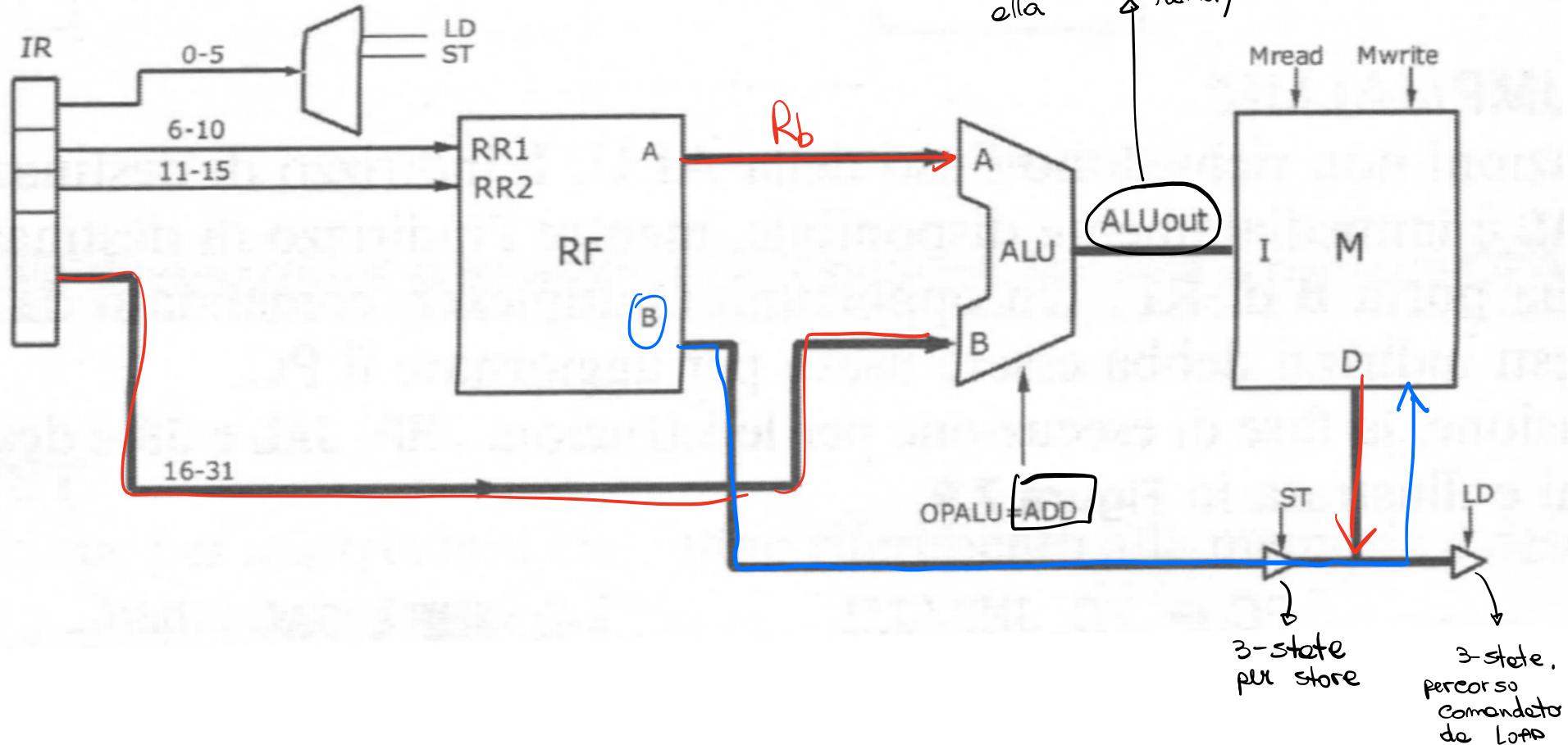


Architettura multiciclo: ME

- LOAD : Linea Dati $\leftarrow M[ALUout]$
- STORE $M[ALUOut] \leftarrow R$ (+ —)

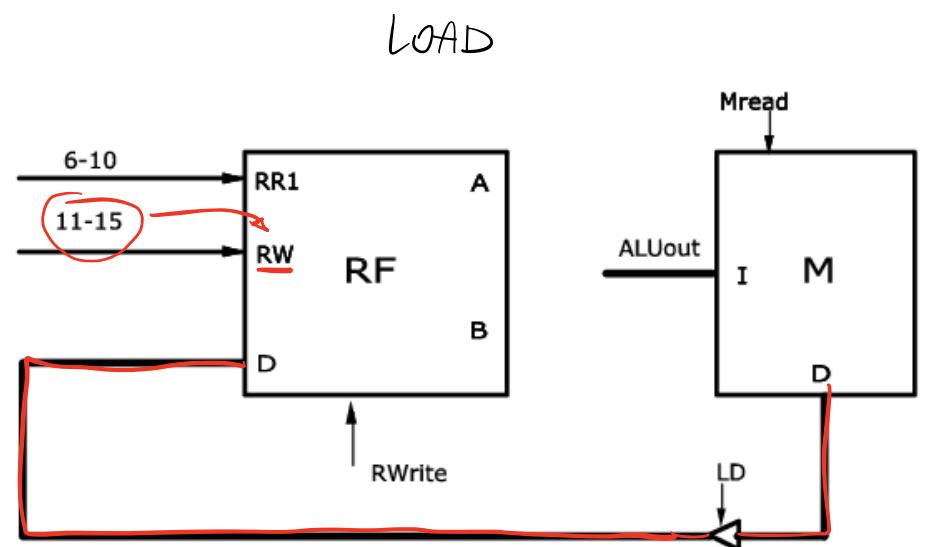
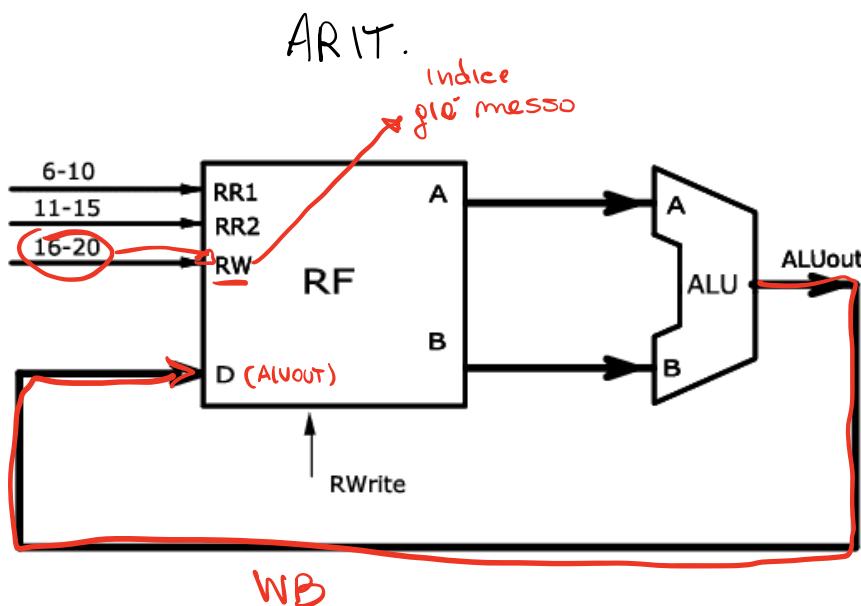
load
store] uniche che
hanno accesso
alla memoria

calcolato in fase
EXECUTE x accedere
alla memoria

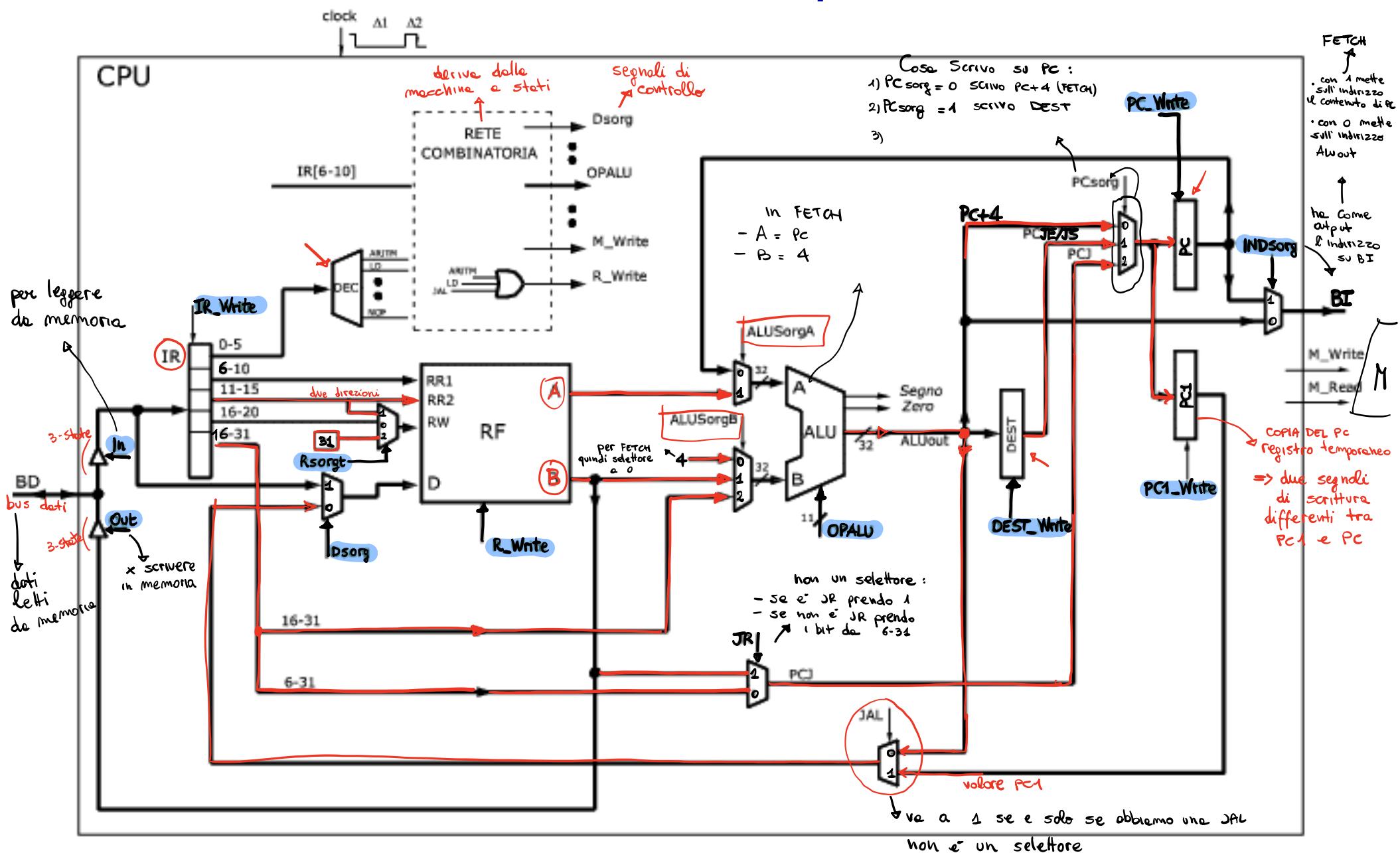


Architettura multiciclo: WB

- Aritmetica o LOAD

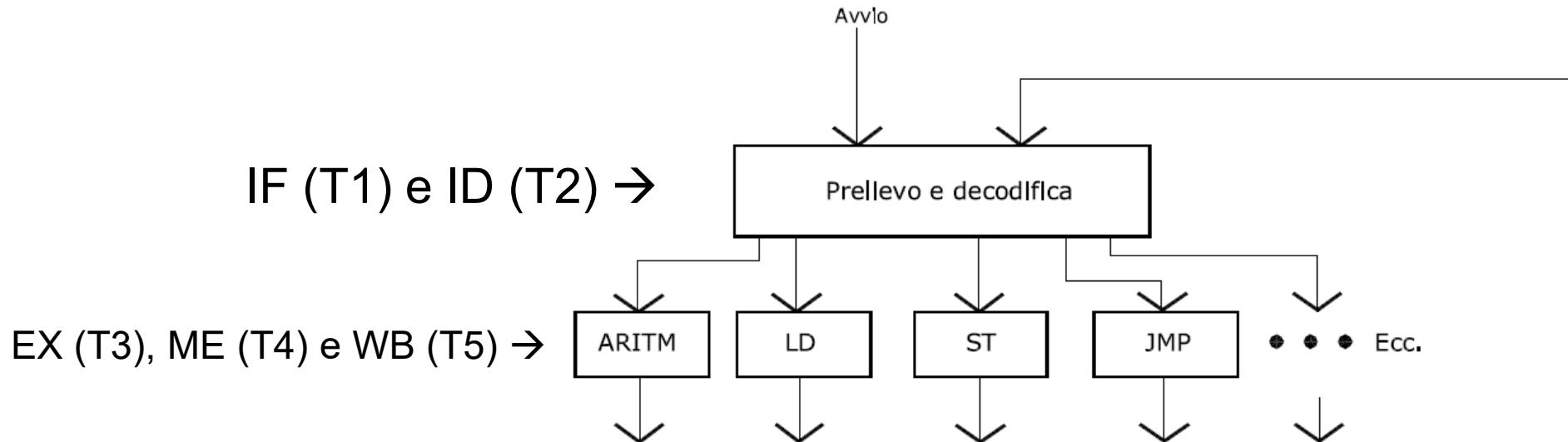


Architettura multiciclo: panoramica



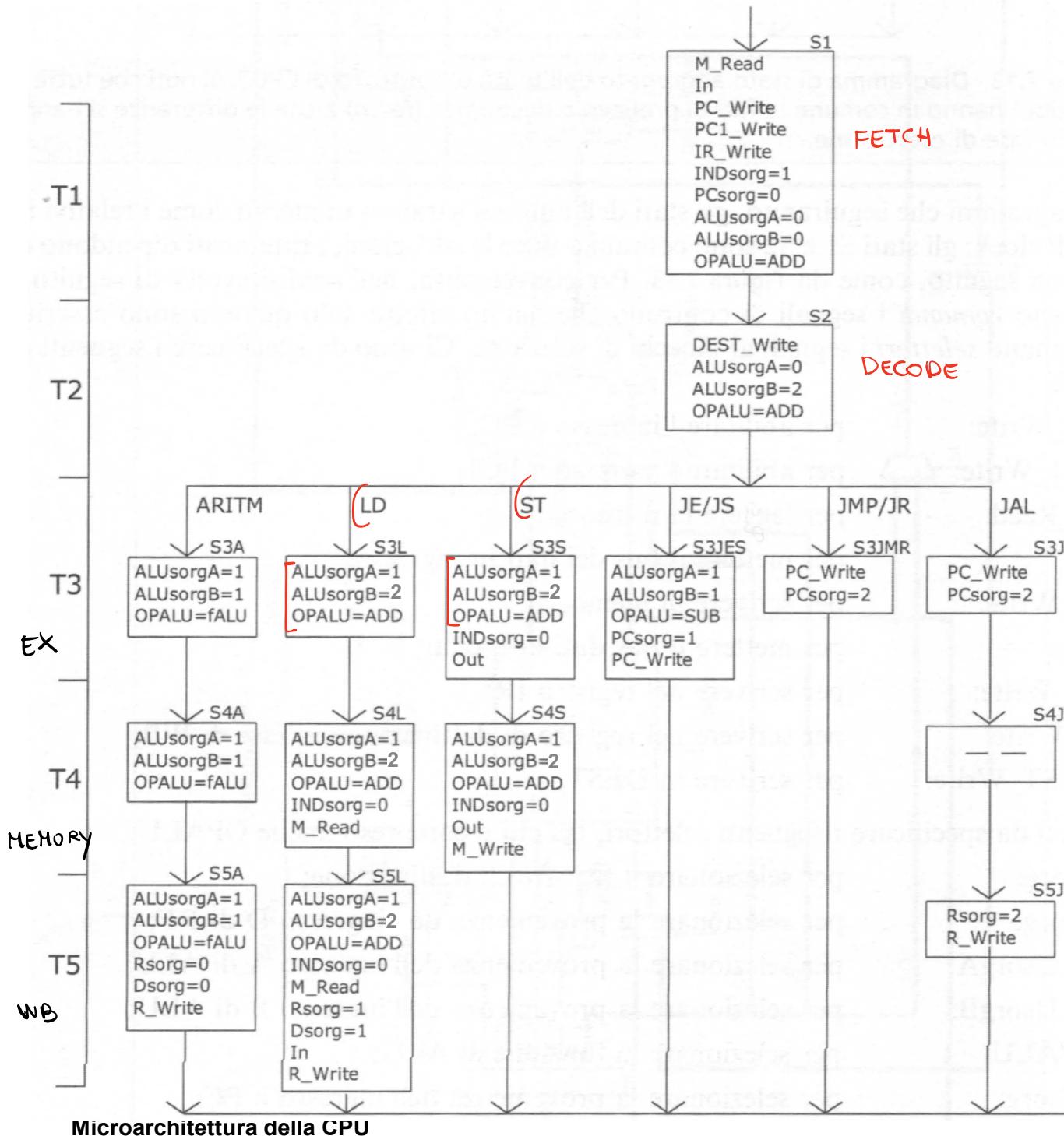
Architettura multiciclo: comandi e selettori

- L'unità di controllo è un automa a stati finiti (macchina di Moore, le cui uscite sono i segnali di controllo)



- Vediamo più nel dettaglio la macchina a stati finiti risultante

Architettura multiciclo: diagramma di stato



Architettura multiciclo: prelievo IF – **stato S1** – **fase T1**

- Devo asserire questi segnali:
→ viene attivato (ASSERIRE) e il segnale
può essere Asserito
basso o alto

FETCH

- M_read: per leggere dalla memoria
- In: per mettere il bus dei dati in ingresso
- PC_write: per abilitare la scrittura su PC
- PC1_write: per abilitare la scrittura su PC1
- IR_write: per memorizzare in IR l'istruzione letta
- Inoltre i selettori devo assumere i seguenti valori:
 - INDsorg = 1 per trasmettere alla memoria in contenuto di PC (e non ALUOut)
 - Pcsorg = 0 per selezionare PC+4 come ingresso a PC
 - ALUsorgA = 0 per selezionare PC come ingresso A della ALU
 - ALUsorgB = 0 per selezionare 4 come ingresso B della ALU
 - OPALU = ADD per eseguire la somma con la ALU

Architettura multiciclo: decodifica ID – stato S2 – fase T2

- Devo asserire questo segnale:
 - DEST_write: per memorizzare in DEST l'indirizzo di destinazione dell'eventuale JE/JS $DEST \leftarrow PC + Offset$
- Inoltre i selettori devo assumere i seguenti valori:
 - ALUsorgA = 0 per selezionare PC come ingresso A della ALU
 - ALUsorgB = 2 per selezionare il campo OFFSET dell'eventuale JE/JS come ingresso B della ALU
 - OPALU = ADD per eseguire la somma con la ALU

Architettura multiciclo: esecuzione EX – istruzioni aritmetiche

- Qui le fasi cambiano da istruzione a istruzione. Per le **istruzioni aritmetiche** abbiamo la fase T3 (S3A) e la fase T5 (S5A). La fase T4 (ME) è di sola transizione e i segnali di T3 sono confermati. → in T4
- In **T3 (stato S3A)** i selettori devono essere:
 - ALUsorgA = 1 per selezionare l'uscita A di RF come ingresso A della ALU
 - ALUsorgB = 1 per selezionare l'uscita B di RF come ingresso B della ALU
 - Istruzione quella codificata negli 11 bit meno significativi dell'istruzione
 - OPALU = fALU per effettuare l'operazione prevista nel CODOP
- In **T5 (stato S5A)** si deve asserire:
 - R_Write: per scrivere nel registro destinazione in RF (abilitare scrittura)
- e i selettori devono essere:
 - Rsorg = 0 per scrivere nel registro Rd indicato nell'istr.
 - Dsorg = 0 per selezionare la provenienza da ALU del dato da scrivere

=> è sottointeso che JAL = 0 (in basso)

Architettura multiciclo: esecuzione EX – istruzione LD

- In **T3 (stato S3L)** i selettori devono essere:
 - ALUsorgA = 1 per selezionare l'uscita A di RF come ingresso A della ALU
 - ALUsorgB = 2 per selezionare il campo OFFSET dell'istruzione come ingresso B della ALU
 - OPALU = ADD per calcolare l'indirizzo di memoria
- In **T4 (stato S4L)** si deve asserire:
 - M_Read: per comandare di leggere da memoria
- Inoltre i selettori della fase T3 devono rimanere tali per mantenere stabile l'indirizzo di memoria in uscita dalla ALU. Inoltre si seleziona:
 - INDsorg = 0 per trasmettere alla memoria ALUout
- In **T5 (stato S5L)** il dato letto va scritto nel registro destinazione, quindi sono asseriti:
 - In: per mettere il bus dei dati in ingresso
 - R_Write: per scrivere nel registro destinazione
- e i seguenti selettori:
 - Rsorg = 1 per scrivere nel registro Rd
 - Dsorg = 1 per selezionare il dato proveniente dalla memoria

ALUout =
Rb +
Offset

Architettura multiciclo: esecuzione EX – istruzione ST delle LOAD

- In T3 (stato S3S) vengono asseriti gli stessi segnali di S3L e in più devo fare quanto segue (e lo faccio in T3 in modo da essere certo che in T4 quando effettuo la scrittura in memoria gli ingressi alla memoria siano stabili):
 – Out asserito: per mettere il bus dei dati in uscita
 – INDsorg = 0 per trasmettere in anticipo alla memoria l'indirizzo calcolato dalla ALU
- In T4 (stato S4S) si devono assegnare:
 – M_Write: per comandare di scrivere in memoria
 – Out: per mantenere stabile il dato in ingresso alla memoria durante la scrittura
- Inoltre in T4 devo continuare a mantenere i valori dei selettori messi in T3 (quindi S3S che eredita alcuni da S3L).
- Non esiste per l'istruzione ST una fase di WB (T5)

“Osservazione”:

⇒ M_Write non si può anticipare ma oltre cose è sempre meglio anticiparla

Architettura multiciclo: esecuzione EX – istruzioni salto condiz.

- Le istruzioni di salto condizionato (JE/JJS) richiedono un solo ciclo per il loro completamento.
- Infatti l'indirizzo di destinazione è già stato calcolato nella fase di ID (T2) e memorizzato nel registro DEST. In T3 devo solo calcolare la condizione e aggiornare, eventualmente, il PC
- In T3 (stato S3JES) è asserito il comando:
 - PC_Write: per abilitare l'ingresso a PC **se e solo se la condizione è vera**
- e devono essere usati i seguenti valori per i selettori:
 - PCsorg = 1 per selezionare DEST come ingresso al PC
 - ALUsorgA = 1 per mantenere selezionata l'uscita A di RF come ingresso A della ALU
 - per i due Registri Sorg.
 - ALUsorgB = 1 per mantenere selezionata l'uscita B di RF come ingresso B della ALU
 - OPALU = SUB per confrontare i due registri
 - per differenza

Con refe logica slide 56

Architettura multiciclo: esecuzione EX – istruzioni JMP e JR

- Anche queste istruzioni possono essere completate in T3
- In **T3 (stato S3JMR)** è asserito il comando:
 - **PC_Write**: per abilitare l'ingresso a PC
- Poichè la linea JR opera come selettore tra PC_JMP/JAL e l'uscita B di RF, il solo selettore di interesse è:
 - **PCsorg = 2** per selezionare il corretto ingresso ingresso al PC

JMP \rightsquigarrow PC \leftarrow IND(6-31)
JR \rightsquigarrow PC \leftarrow R.FB

Architettura multiciclo: esecuzione EX – istruzione JAL

- Siccome questa istruzione richiede la scrittura nel registro R31 deve proseguire fino alla fase T5.
- Più specificatamente, nella fase T3 si comporta come il JMP/JR precedente, in T4 non compie alcuna azione, mentre in T5 scrive in R31 il contenuto di PC1.
- Quindi in T5 (stato S5JL) è asserito il comando:
 - R_Write: per scrivere nel registro destinazione
- e inoltre i seguenti valori vanno posti nei selettori:
 - Dsorg = 0 per selezionare la provenienza da PC1 del dato da scrivere
 - Rsorg = 2 per scrivere il contenuto di PC1 nel registro R31
- Si osservi che il selettore JAL=1 in automatico smista la provenienza del dato da scrivere dal PC1.
- **Attenzione:** questa architettura semplificata non permette chiamate innestate (né tantomeno ricorsive) di funzioni perchè sovrascriverebbero il valore di R31

Architettura multiciclo vs monociclo

- Il clock (T_{multi}) va scelto in base allo stadio più lento. Se indichiamo con τ_k il tempo richiesto dal generico stadio k, con n (5 nel nostro esempio) il numero di stadi, si ha che

$$T_{multi} = \max_{i=1, \dots, n} \{\tau_k\}$$

- Multiciclo fornisce migliori prestazioni ?
 - Dipende dal codice ! \Rightarrow +. di esecuzione
dipende dall'istruzione

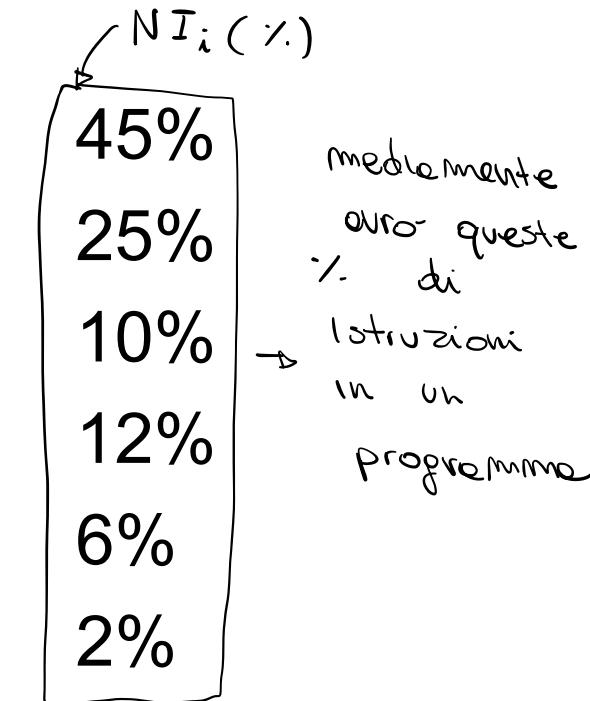
	FETCH	DECOD	ALU	MEM	WB	TOTALE	TEMPO ESECUZIONE
ARITM	30	5	12		5	52	
LD	30	5	12	30	5	82	150 ns
ST	30	5	12	30		77	
JP COND.	30	5	12			47	90 ns
JMP	30	5				35	
JAL/CALL	30	5			5	40	

Architettura multiciclo vs monociclo

- Accesso in memoria e fetch stadi più lenti, quindi $T_{multi} = 30$

- Durata e possibile consistenza numerica

– Aritm (5 stadi):	150 ns
– LD (5 stadi):	150 ns
– ST (4 stadi):	120 ns
– JE/JJS (3 stadi):	90 ns
– JMP/JR (3 stadi):	90 ns
– JAL (5 stadi):	150 ns



- $0,45 \times 150 + 0,25 \times 150 + 0,10 \times 120 + 0,12 \times 90 + 0,06 \times 90 + 0,02 \times 150 = 136,2 \text{ ns (!)}$
- Peggioramento del 66,10 % (monociclo 82 ns) → $100 \times (136,2 - 82) / 82 = 66,10$

Miglioramenti possibili

- Chiaramente questa architettura mult ciclio non è efficiente. Ci sono più modi per migliorarla:
 - 1° – Scegliere un periodo di clock più breve (<30) introducendo eventuali stati di attesa per le fasi più lunghe
 - 2° – Anticipare per quanto possibile le operazioni, in modo da ridurre il numero medio di clock per istruzione
 - 3° – Compattare le fasi distinte ed aggiustare opportunamente il clock
- Vediamo alcuni aggiustamenti per migliorare le prestazioni

1° APPROCCIO**Aumento della granularità del clock**

- Avendo preso come T_{multi} il tempo di accesso alla memoria, nelle fasi in cui questa non è utilizzata c'è un grande spreco di tempo
- L'ideale per un'architettura multiciclo è che tutti gli stadi richiedano circa lo stesso tempo.
- Se prendiamo ad esempio $T_{multi} = 12$ (tempo dell'ALU) allora gli stadi di accesso alla memoria richiederanno 3 cicli di clock ($12 \times 3 = 36 > 30$) e le durate diventano:
 - Aritm: $12 \times 3 + 12 + 12 + 12 + 12 = 84$ ns → prima era 150 ns
 - LD: $12 \times 3 + 12 + 12 + 12 \times 3 + 12 = 108$ ns → " " 150 ns
 - ST: $12 \times 3 + 12 + 12 + 12 \times 3 = 96$ ns → " " 120 ns
 - JE/JS: $12 \times 3 + 12 + 12 = 60$ ns → " "
 - JMP/JR: $12 \times 3 + 12 + 12 = 60$ ns
 - JAL: $12 \times 3 + 12 + 12 + 12 + 12 = 84$ ns
- Questo porta ad un tempo medio di 86,88 ns che risulta in un miglioramento del 36,21% rispetto alla versione multiciclo originale, ma questa nuova CPU multiciclo (**CPU A**) è ancora più lenta del 5,95% rispetto alla monociclo

1° APPROCCIO

Aumento della granularità del clock

- Il problema della CPU A è che il tempo a 12 ns fa sprecare ben 6 ns nelle operazioni di accesso alla memoria (36 ns quando ne basterebbero 30).
- Vediamo una nuova versione (CPU B) con $T_{multi} = 5$. La fase di IF impiega 6 periodi di clock (per arrivare a 30 visto che accede alla memoria), ID ne richiede 3 per via della ALU, EX ne richiede 3 per le istruzioni che usano la ALU mentre 1 per le altre, ME 6 per quelle che accedono alla memoria mentre 1 per le altre, WB solo 1. Quindi:

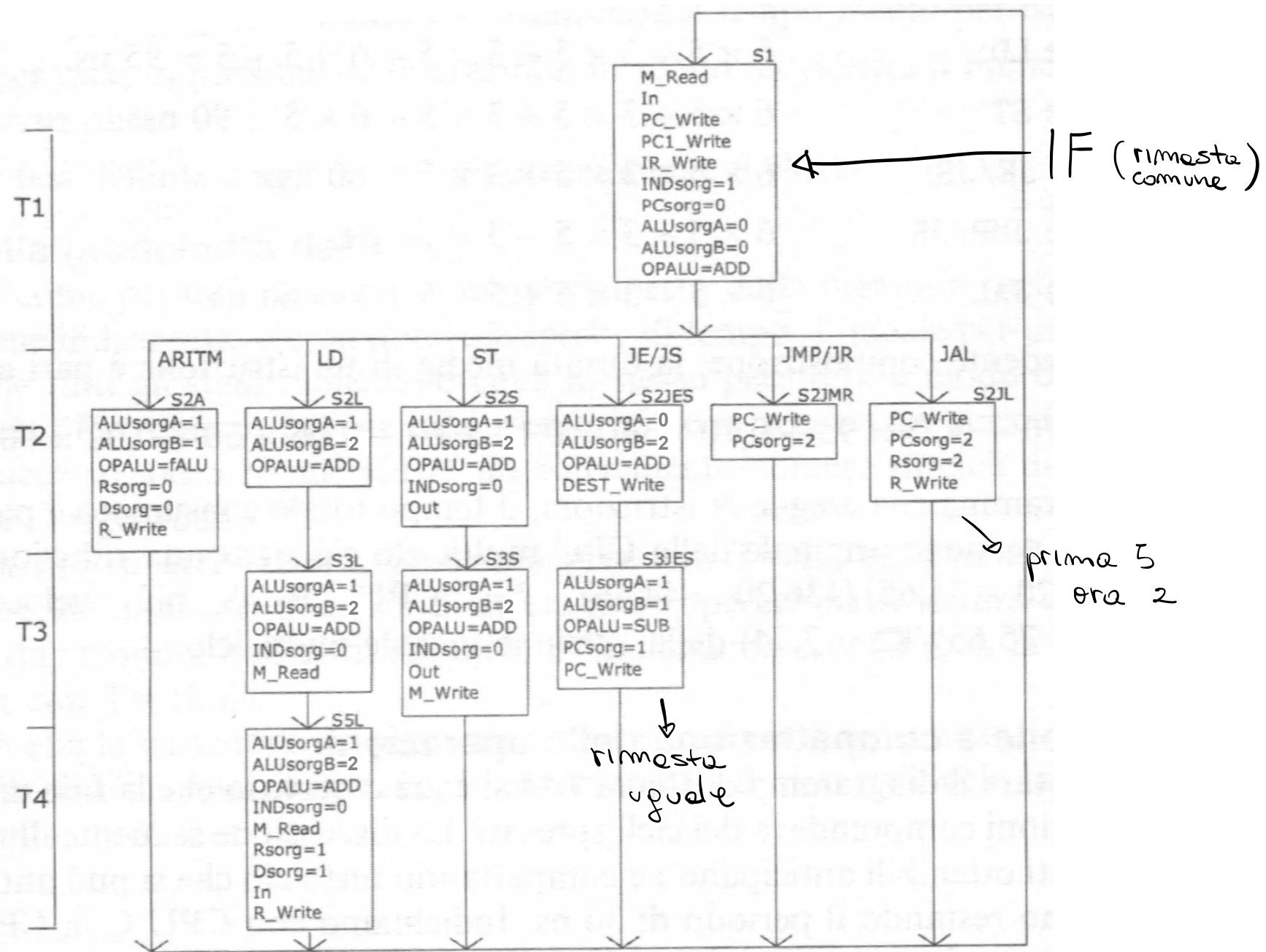
- Aritm: $5 \times 6 + 5 \times 3 + 5 \times 3 + 5 + 5 = 70 \text{ ns}$
- LD: $5 \times 6 + 5 \times 3 + 5 \times 3 + 5 \times 6 + 5 = 95 \text{ ns}$
- ST: $5 \times 6 + 5 \times 3 + 5 \times 3 + 5 \times 6 = 90 \text{ ns}$
- JE/JS: $5 \times 6 + 5 \times 3 + 5 \times 3 = 60 \text{ ns}$
- JMP/JR: $5 \times 6 + 5 \times 3 + 5 = 50 \text{ ns}$
- JAL: $5 \times 6 + 5 \times 3 + 5 + 5 + 5 = 60 \text{ ns}$

- Tempo medio = 75,65 ns (miglioramento del 44,31% rispetto alla versione multiciclo originale; miglioramento del 7,5% rispetto alla monociclo)

\Rightarrow IPOTESI di eseguire
un'istruzione in
un ciclo di clock
Cambiato

- ## 2° APPROCCIO Anticipazione e compattazione delle operazioni
- Agiamo ora invece sul diagramma degli stadi visto in precedenza cercando di anticipare e compattare alcune operazioni. Torniamo per ora a $T_{multi} = 30$ per confronto e chiamiamola **CPU C**.
 - Nell'anticipare partiamo dall'osservazione che in T2 si esegue un passo (costoso perchè usa la ALU) che serve solo a JE/JS. Possiamo quindi differenziare la fase T2 a seconda dell'istruzione
 - Inoltre visto che il periodo è molto lungo possiamo accorpare più operazioni nello stesso clock:
 - Aritmetiche:** dopo il fetch, richiedono decodifica, operazione ALU e WB. Totale: $5+12+5=22 \rightarrow$ mi basta una sola fase (T2)
 - LD:** il calcolo dell'indirizzo può essere fatto in T2, la lettura della memoria in T3 e la scrittura nel registro destinazione in T4 \Rightarrow anticipa tutto di una fase
 - ST:** come per LD anticipiamo il calcolo in T2 e quindi la scrittura in memoria può avvenire in T3 \Rightarrow anticipa tutto di 1 stadio
 - JE/JS:** nessun anticipo possibile \Rightarrow solo = PC + Offset, verifica condizione con SUB, quindi non possiamo anticipare nulla
 - JMP/JR:** aggiornamento PC in T2 \rightarrow T1, T2
 - JAL:** aggiornamento PC e scrittura R31 in T2 (elimino PC1)

Anticipazione e compattazione delle operazioni



Anticipazione e compattazione delle operazioni

- Risultano quindi queste durate:
 - Aritm (2 stadi): $30+30=60$ ns
 - LD (4 stadi): $30+30+30+30=120$ ns
 - ST (3 stadi): $30+30+30 = 90$ ns
 - JE/JS (3 stadi): $30+30+30 = 90$ ns
 - JMP/JR (2 stadi): $30+30 = 60$ ns
 - JAL (2 stadi): $30+30 = 60$ ns
- Tempo medio = 81,60 ns (miglioramento del 40,09% rispetto alla versione multiciclo originale; praticamente la stessa velocità della monociclo!!)

3° APPROCCIO Anticipazione e compattazione delle operazioni

- Ultimo caso (**CPU D**) lo vediamo riportando il tempo di ciclo a 5 ns e cercando di compattare ove possibile.
- Le fasi **IF** e **ME** richiedono 6 periodi di **clock**. Per ottimizzare i tempi conviene ipotizzare che la fase ID sia uguale per tutte le istruzioni e si riduca alla semplice decodifica del codice operativo, così da poter durare un solo periodo.
- Le istruzioni si differenziano a partire da **EX**. Per aritmetiche, **LD/ST** e **JE/JS** (che ora calcolano l'indirizzo in questa fase) lo stato **S3** richiede 3 cicli di clock per via della ALU. **JAL** può avere il **WB** nello stato **S3**. **JMP** e **JR** possono concludersi in **T2**. Su **S4** le aritmetiche richiedono un solo ciclo (per **WB**), **LD/ST** ne vogliono 6, **JE/JS** 3 per l'uso di ALU e scrittura nel PC.

Anticipazione e compattazione delle operazioni

- Risultano quindi queste durate:
 - Aritm:
 - LD: IF
Comune
e tutte
 - ST:
 - JE/JS:
 - JMP/JR:
 - JAL:

ID comune a tutte

$5 \times 6 + 5 \times 3 + 5 \times 3 + 5 + 5 = 70$	55 ns
$5 \times 6 + 5 \times 3 + 5 \times 3 + 5 \times 6 + 5 = 95$	85 ns
$5 \times 6 + 5 \times 3 + 5 \times 3 + 5 \times 6 = 90$	80 ns
$5 \times 6 + 5 \times 3 + 5 \times 3 + 5 \times 3 = 60$	65 ns
$5 \times 6 + 5 \times 3 + 5 = 50$	35 ns
$5 \times 6 + 5 \times 3 + 5 + 5 + 5 = 60$	40 ns
- Tempo medio = 64,76 ns (miglioramento del 52,50% rispetto alla versione multiciclo originale; miglioramento del 21,10% rispetto alla monociclo!!)

Riassunto finale

	%	monociclo	CPU 0	CPU A	CPU B	CPU C	CPU D
ARITM	45	82	150	84	70	60	55
LD	25	82	150	108	95	120	85
ST	10	82	120	96	90	90	80
JE/JS	12	82	90	60	60	90	65
JMP/JR	6	82	90	60	50	60	35
JAL	2	82	150	84	70	60	40
TMI		82	136,2	86,88	75,85	81,60	64,76