Architetture degli elaboratori II

Docenti: Alberto Borghese, Federico Pedersini
Dipartimento di Informatica
Università degli Studi di Milano

Turno 2 (Cognomi G–Z)

Prof. Federico PEDERSINI
pedersini@di.unimi.it

© F. Pedersini – Dip. Informatica, Università degli studi di Milano

Architetture Elaboratori II (6 CFU)

Orario lezioni turno B (G–Z):

- mercoledì 10.30-12.30 Aula **G12** (via Golgi)
- venerdì 10.30-12.30 Aula **G21** (via Golgi)

Laboratorio:

Turno B (G-Z): venerdì 13:30 – 15:30 Aula **309**

Orario di ricevimento:

- durante il corso: dopo le lezioni
- resto dell'anno: su appuntamento

Modalità d'esame:

– Prova scritta + prova orale
– Progetto di laboratorio: **progetto in Assembly MIPS**

© F. Pedersini – Dip. Informatica, Università degli studi di Milano

Materiale didattico

MATERIALE DIDATTICO

- **Appunti + slide**
  - Slide delle lezioni: homes.dsi.unimi.it/~pedersin/AE-INF.html
  - Le slide da sole NON BASTANO!

- **Testo di riferimento:**

  Edizione originale:
  D.A. Patterson, J.L. Hennessy,
  "Computer Organization & Design: The Hardware/Software Interface",
  Morgan Kaufmann, 2005.

  Edizione in italiano:
  D.A. Patterson, J.L. Hennessy,
  "Struttura e Progetto dei Calcolatori",
  Zanichelli, 2006

Obiettivo del corso

MISSIONE: capire come funziona un elaboratore elettronico

LEZIONE

Fondamenti di elettronica digitale
- Algebra di Boole
- Logica combinatoria
- Macchine sequenziali
- La CPU

Il linguaggio del calcolatore
- Programmazione Assembly
- Linguaggio macchina

Come funziona un calcolatore moderno
- **Strutture di CPU evolute**
- memorie (cache)
- I/O (bus, interrupt e DMA, ...)
- Architetture moderne (multiprocessore, multicore, GPU)

LABORATORIO

I.
Progettazione di circuiti digitali

LOGISIM: ambiente di progettazione e simulazione di circuiti digitali
http://sourceforge.net/projects/circuit/

II.
Programmazione CPU in Assembly

SPIM: ambiente di simulazione elaboratore con CPU MIPS32
http://spimsimulator.sourceforge.net

Architettura degli Elaboratori II

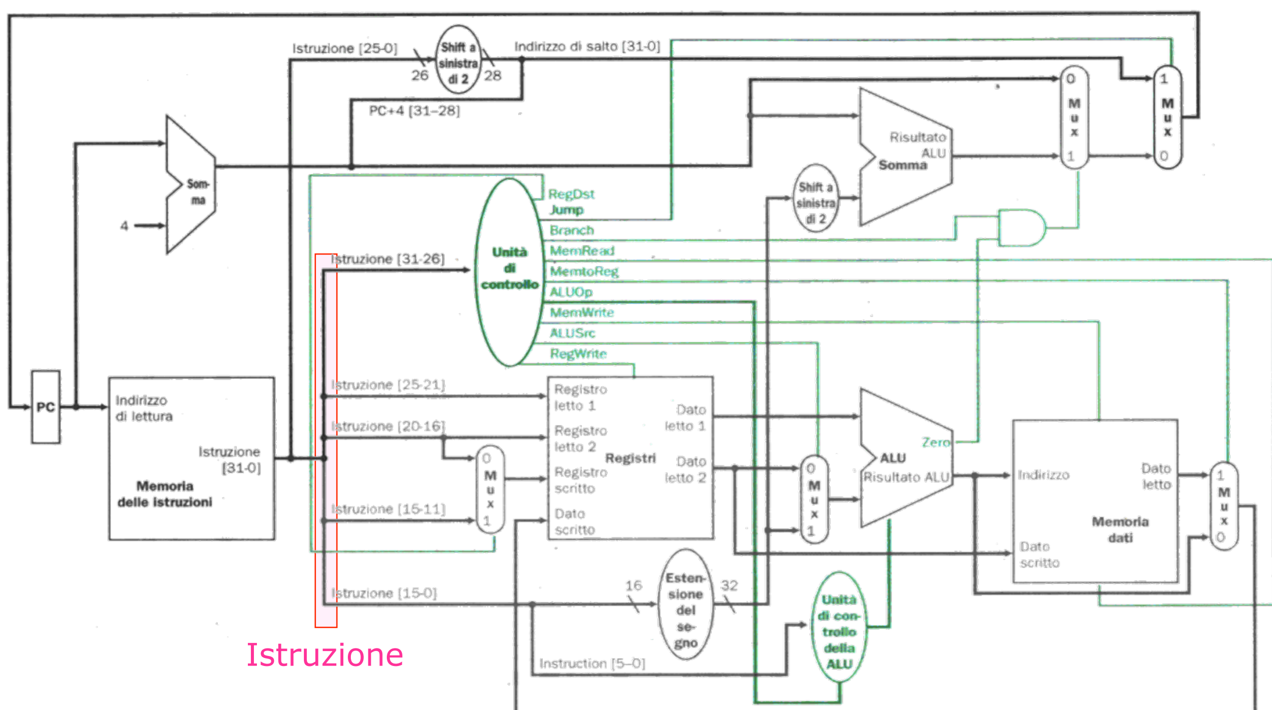
Lezione 1

CPU a ciclo multiplo

Proff. A. Borghese, F. Pedersini

Dipartimento di Informatica
Università degli Studi di Milano

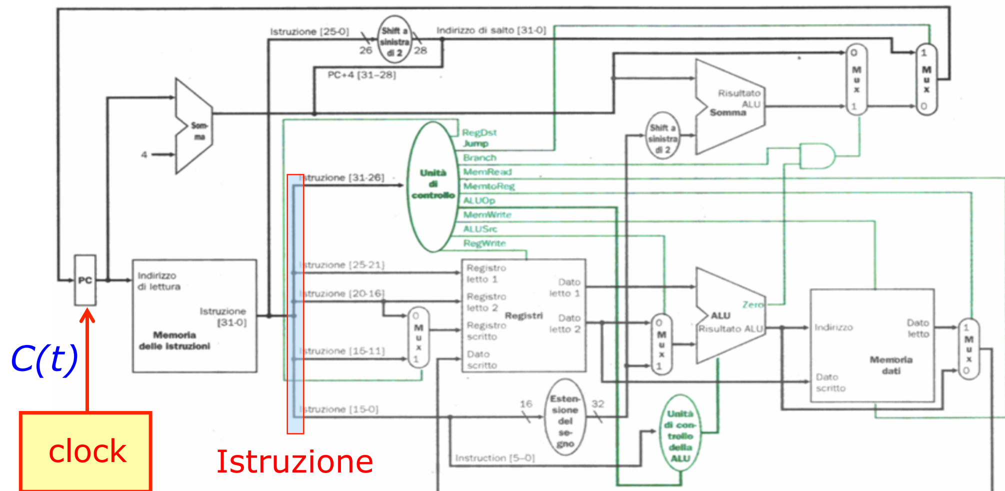
La CPU a singolo ciclo, completa

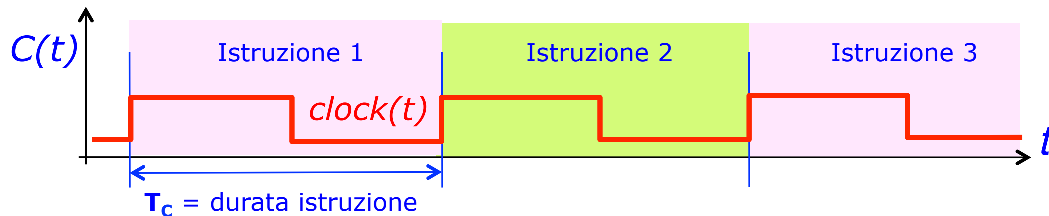


CPU a Ciclo singolo

❖ CPU a CICLO SINGOLO:

Ad ogni ciclo di clock viene eseguita un'istruzione completa.





CPU singolo ciclo: caratteristiche

Peculiarità: **semplicità**, di struttura e di funzionamento

- Il segnale di CLOCK viene fornito solo al registro Program Counter
- L' Unità di Controllo è una rete combinatoria

Vantaggi:

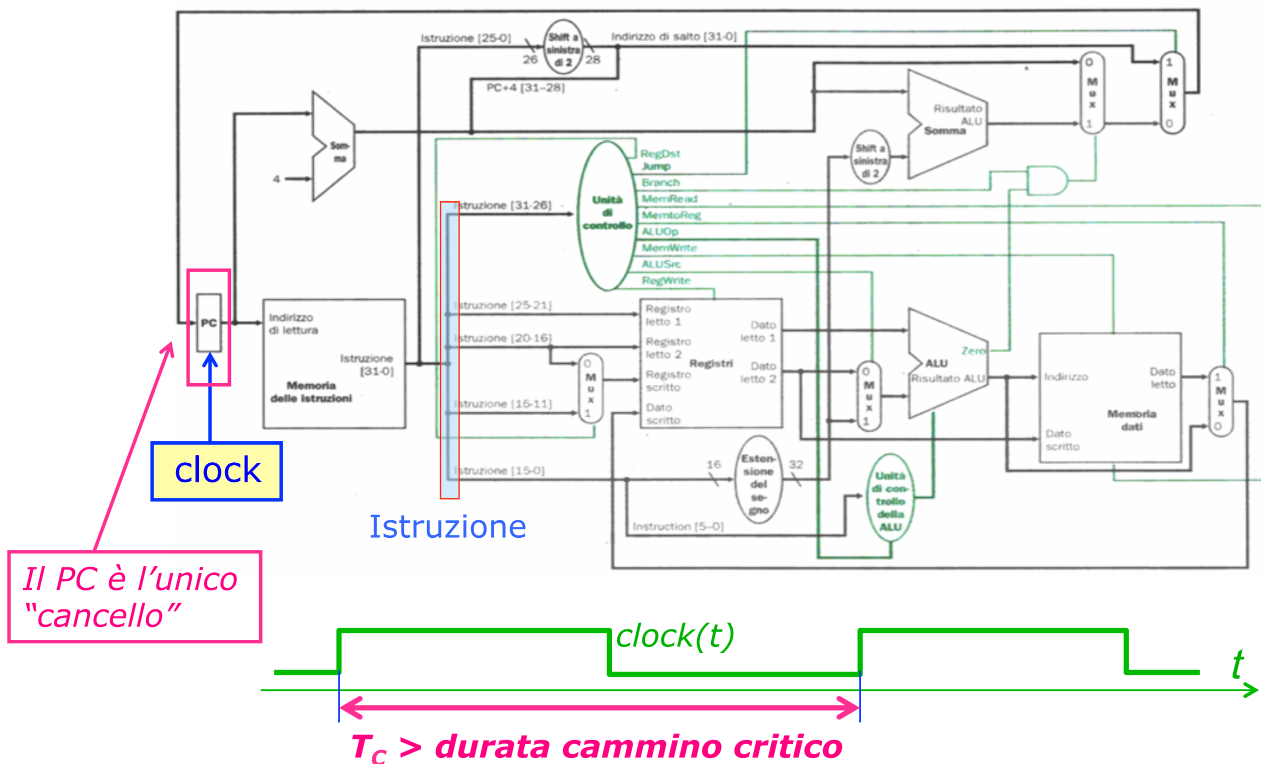
- Semplicità circuitale e di funzionamento
- Velocità di esecuzione, fissato T_C: **1 istruzione / ciclo di Clock**

Limiti:

- Utilizzo **non efficiente** delle risorse (3 ALU, 2 memorie) utilizzate più volte:
  – Memoria Istruzioni in fase di fetch, memoria Dati in fase di accesso memoria (`lw,sw`)
  – ALU PC (fetch), ALU branch (decodifica), ALU principale operazioni/test (esecuzione)
- **Tutte le istruzioni**, semplici o complesse, **durano** sempre T_C

T_C è dimensionato sul cammino critico dell'istruzione che dura di più

CPU a Ciclo singolo



 © F. Pedersini – Dip. Informatica, Università degli studi di Milano

CPU a ciclo singolo: analisi prestazioni

Qual è il **cammino critico** della **CPU singolo ciclo**?

Ogni istruzione ha un **datapath differente**
➔ cammino critico differente:

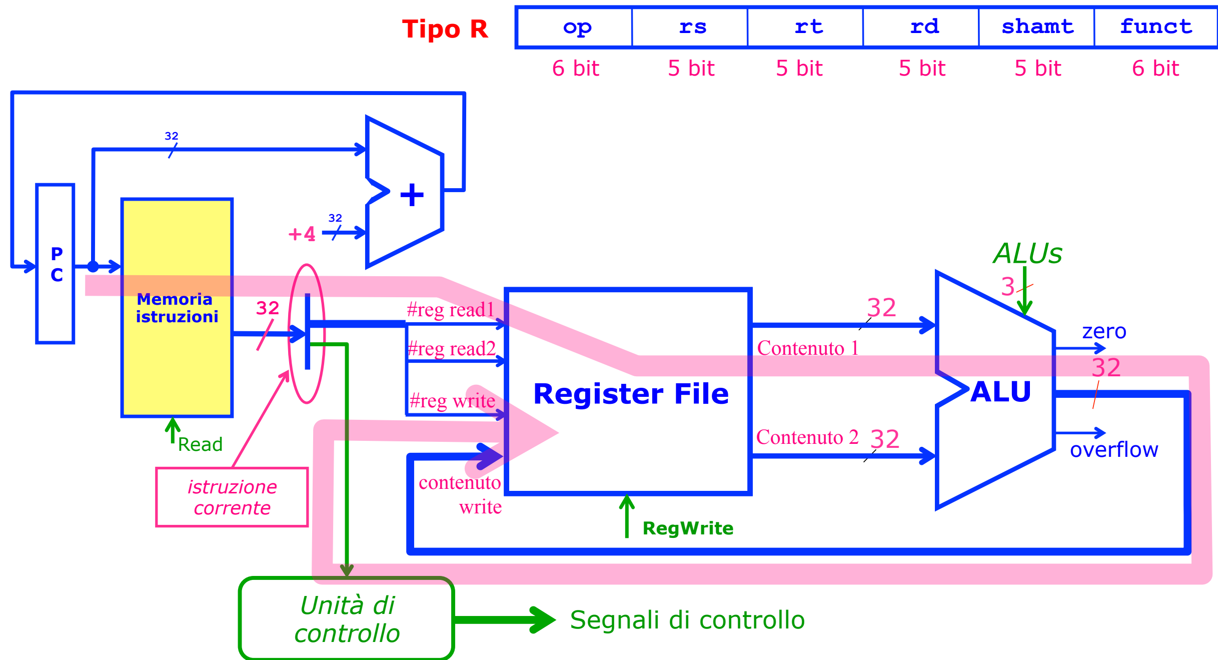
- **Aritmetico/Logica** (tipo R o I)
- Lettura da memoria (**lw**)
- Scrittura in memoria (**sw**)
- Salto condizionato (**beq**)
- Salto incondizionato (**j**)

Cammino critico della CPU: quello massimo!

 © F. Pedersini – Dip. Informatica, Università degli studi di Milano

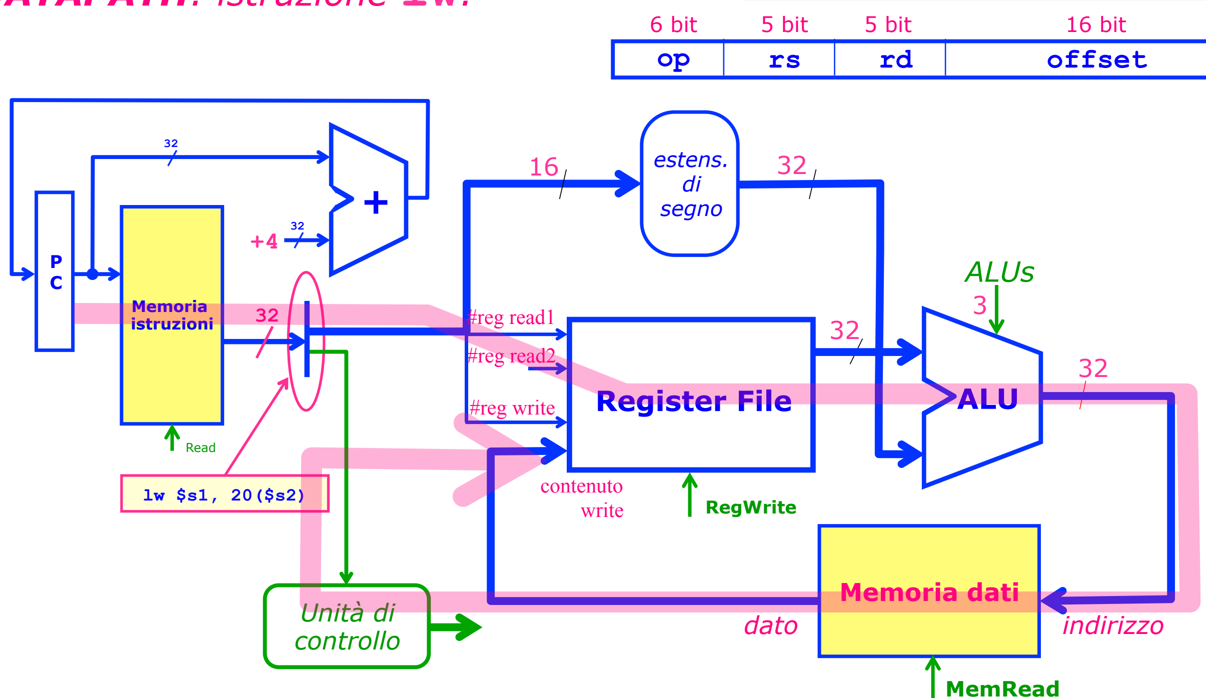
Datapath – istruzione A/L tipo R

DATAPATH: *istruzione **aritmetico/logica** tipo R:*



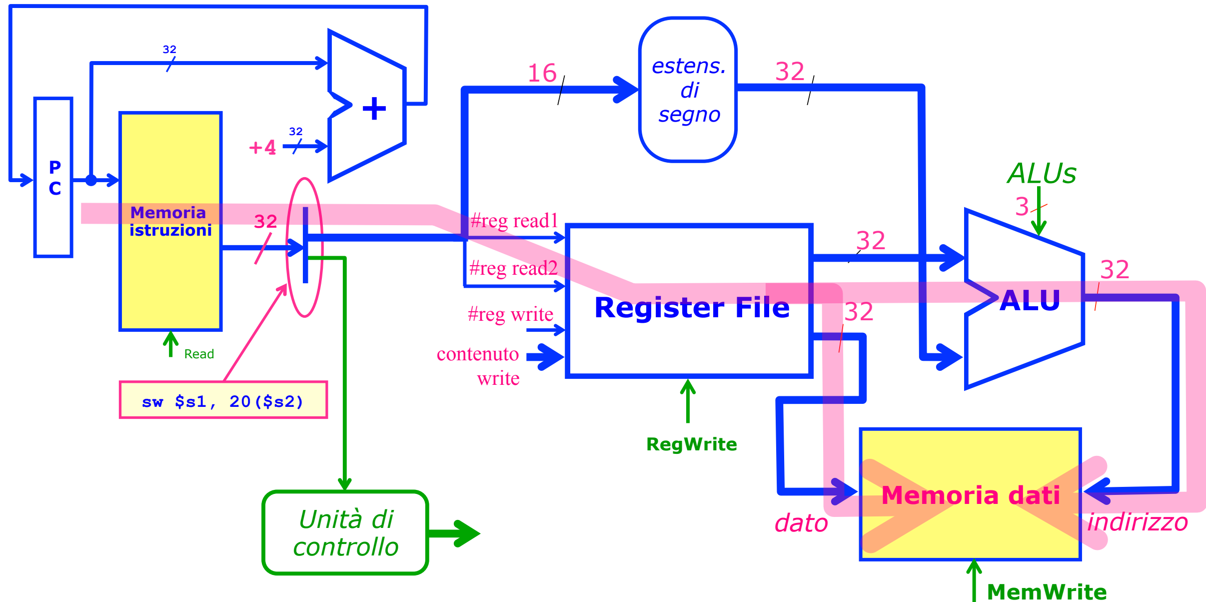
Datapath – esecuzione di una `lw`

DATAPATH: *istruzione* `lw`:



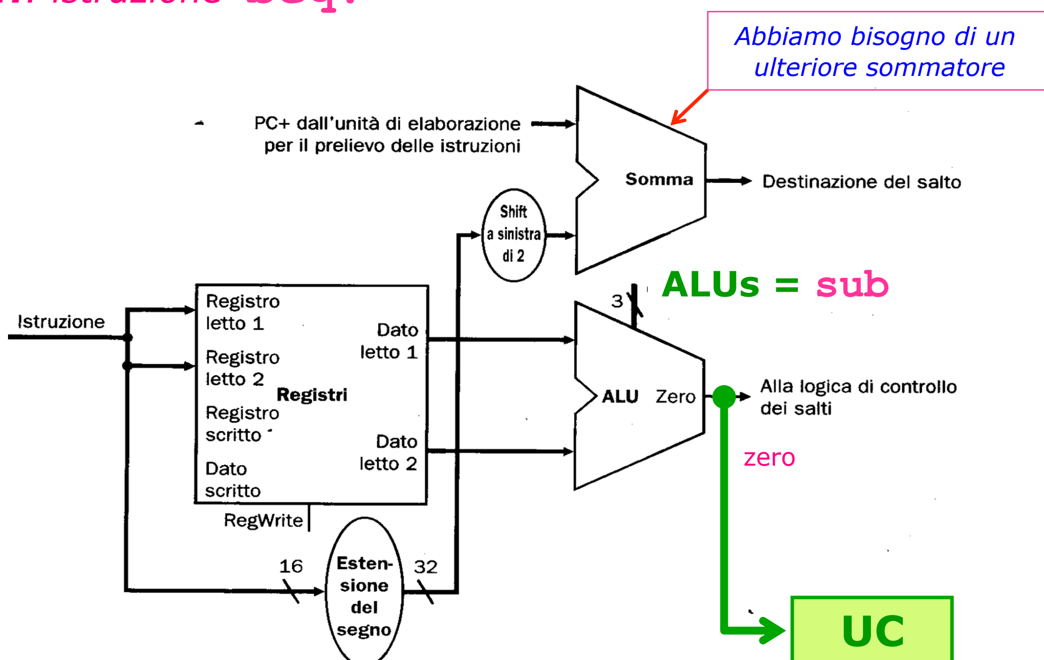
Datapath – esecuzione di una sw

DATAPATH: *istruzione* **sw**:



Circuito di esecuzione: beq

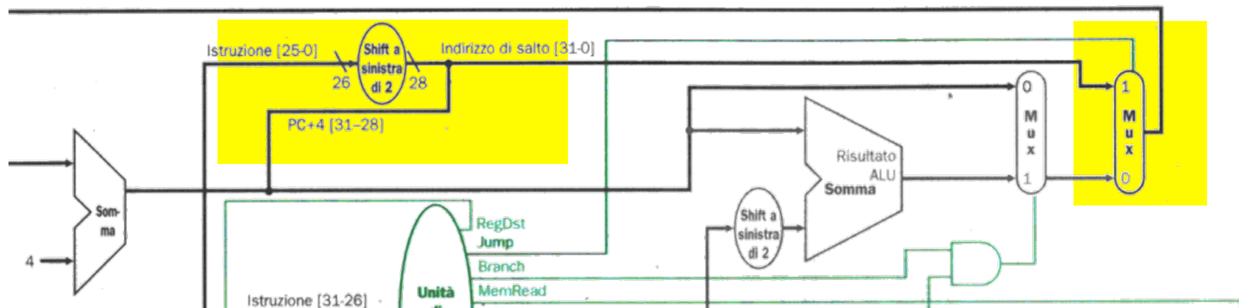
DATAPATH: *istruzione* **beq**:



Circuito di esecuzione: `j`

***DATAPATH**: istruzione* `j`:

Cammino critico molto breve (1 **MUX**!)



Esecuzione in un singolo ciclo di clock

- La **durata** del ciclo di **clock** T_C va dimensionata sul **caso peggiore**
  → istruzione di maggior durata
  → percorso logico più lungo (**cammino critico**)
  - **Accesso memoria: 2 ns** **ALU e sommatori: 2 ns**
  - **lettura/scrittura registri: 1 ns** **decodifica: 2 ns**
  - tempi trascurabili per gli altri elementi della CPU
  - componenti indipendenti possono lavorare in parallelo

Istruzione	Memoria istruzioni (fetch)	Lettura registri (decodifica)	Operazioni ALU	Accesso Memoria Dati	Write-back nel RF	**Totale**
`Tipo R`	2	2	2	0	1	**7 ns**
`lw`	2	2	2	2	1	**9 ns**
`sw`	2	2	2	2	0	**8 ns**
`beq`	2	2	2	0	0	**6 ns**
`j`	2	2	0	0	0	**4 ns**

Percorso più lungo: caricamento da memoria – `lw`

Valutazione prestazioni: CPU a singolo ciclo

- Nella CPU a singolo ciclo la durata delle istruzioni è costante, e pari alla durata massima necessaria
- La durata delle istruzioni è maggiore della durata teorica media
  - In alcuni casi (caso II) decisamente maggiore della media pesata sulla durata delle istruzioni

La CPU a singolo ciclo è **inefficiente** (c'è uno spreco di tempo)

Istruzione:	lw	sw	beq	j	R	fp (add)	fp (mul)	Durata MAX	Durata media
Durata effettiva	**9 ns**	**8 ns**	**6 ns**	**4 ns**	**7 ns**	**12 ns**	**20 ns**		
Caso I	24%	12%	18%	2%	44%			**9 ns**	**7.36 ns**
Caso II (con **FPU**)	31%	21%	5%	2%	27%	7%	7%	**20 ns**	**8.98 ns**

CPU multi-ciclo

Soluzione: **CPU a ciclo multiplo**

Idea di base: si spezza l'istruzione in **più passi parziali**

- ciascun passo parziale impiega lo stesso tempo.

- in un ciclo di clock viene eseguito un singolo passo parziale di istruzione (non l'intera istruzione)
- le istruzioni possono durare un numero diverso di cicli di clock

- Conseguenze:
  - Diventa possibile riutilizzare le unità funzionali, in cicli di clock diversi
  - Necessità di registri di memoria temporanea: devono memorizzare lo stato delle unità funzionali, cioè l'informazione che può servire ai passi successivi.
  - L'unità di controllo diventa una FSM.
    - Nella CPU a singolo ciclo era una Rete Combinatoria

Valutazione prestazioni: CPU multi-ciclo

CPU multi-ciclo:

Ogni istruzione dura **solo il numero di cicli di clock necessario**

- Durata effettiva = media pesata della durata delle istruzioni
- La CPU multi-ciclo consente una riduzione del tempo di esecuzione
- Dipende dal tipo di programma (frequenza delle diverse categorie di istruzione)

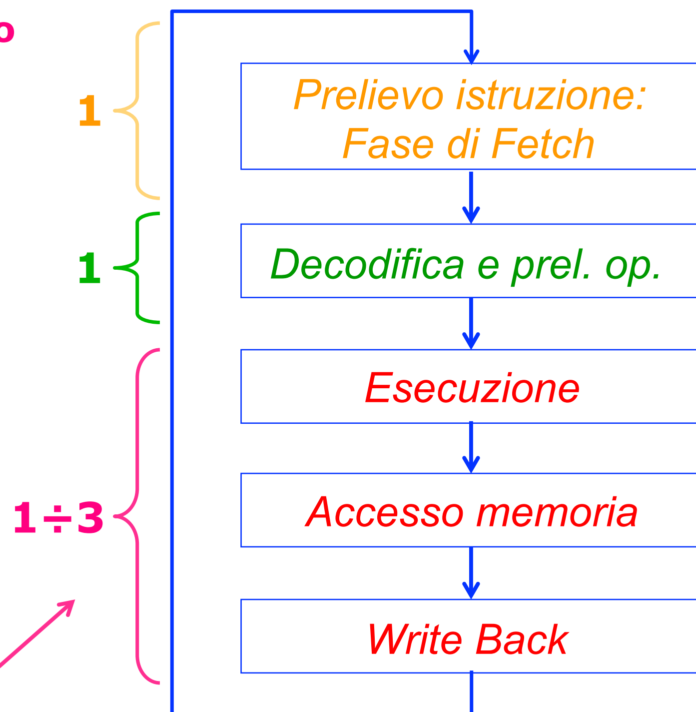
Istruzione:	`lw`	`sw`	`beq`	`j`	`R`	`fp (add)`	`fp (mul)`	Durata MAX	Durata media
Durata effettiva	**9 ns**	**8 ns**	**6 ns**	**4 ns**	**7 ns**	**12 ns**	**20 ns**		
Caso I	24%	12%	18%	2%	44%			**9 ns**	**7.36 ns**
Caso II (con **FPU**)	31%	21%	5%	2%	27%	7%	7%	**20 ns**	**8.98 ns**

Ciclo di esecuzione di un'istruzione

Passi parziali = fasi del ciclo di esecuzione istruzioni

- Le diverse fasi del ciclo di esecuzione vengono eseguite in momenti diversi, in successione
- L'esecuzione di ogni istruzione richiederà da 3 a 5 cicli di clock

Totale:
3÷5 cicli clock



Progetto CPU multi-ciclo

CPU Multi-ciclo – Idea di progetto:

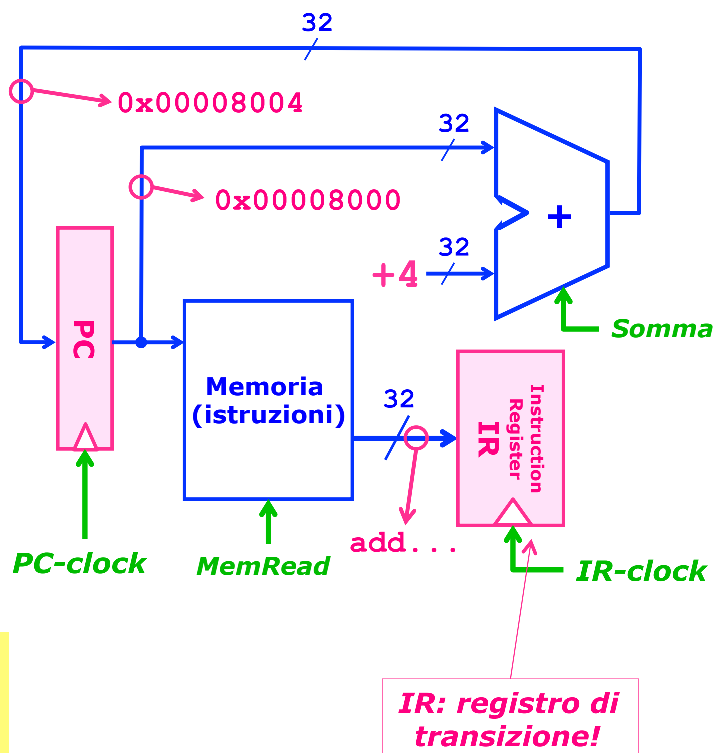
- *Progetto ogni singola fase in modo che avvenga in un ciclo di clock*
- *devo organizzare il passaggio di informazioni alla fase successiva:*
  - → *devo "memorizzare informazioni per il futuro"*
  - → *ho bisogno di* ***registri di transizione***
- *(posso riutilizzare una risorsa in tempi diversi!)*

Circuito della *fase di fetch multi-ciclo (fase 1)*

Fase di FETCH:

Cosa cambia?

- Devo memorizzare l'istruzione → **IR: Instruction Register**
- **Clock** differenti per **PC** e **IR**
- Comando esplicito **MemRead** per la **memoria istruzioni**




```
8000:  add $s1, $s2, $s3
8004:  sub $s4, $s1, $t1
.....
```

Sommario

Fasi successive al FETCH:

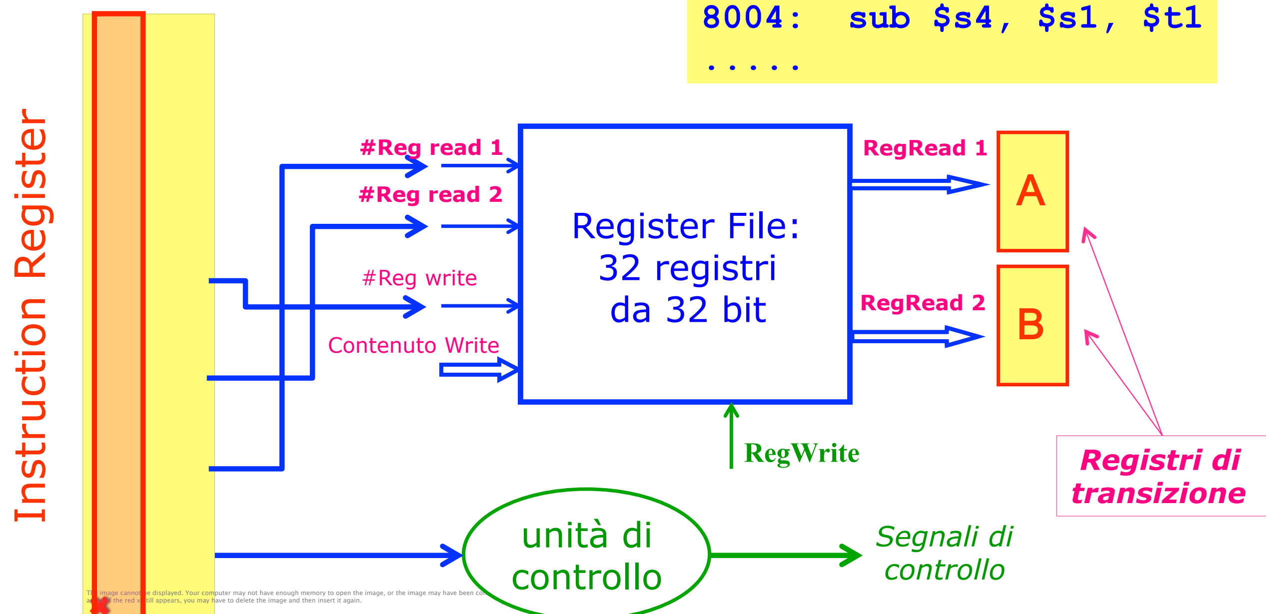
Dalla fase successiva a quella di Fetch, consideriamo la situazione istruzione per istruzione:

- Esecuzione multi-ciclo delle istruzioni A/L
- Esecuzione multi-ciclo delle istruzioni **lw/sw**
- Esecuzione multi-ciclo dei salti

Decodifica e lettura dei registri (*fase 2*)

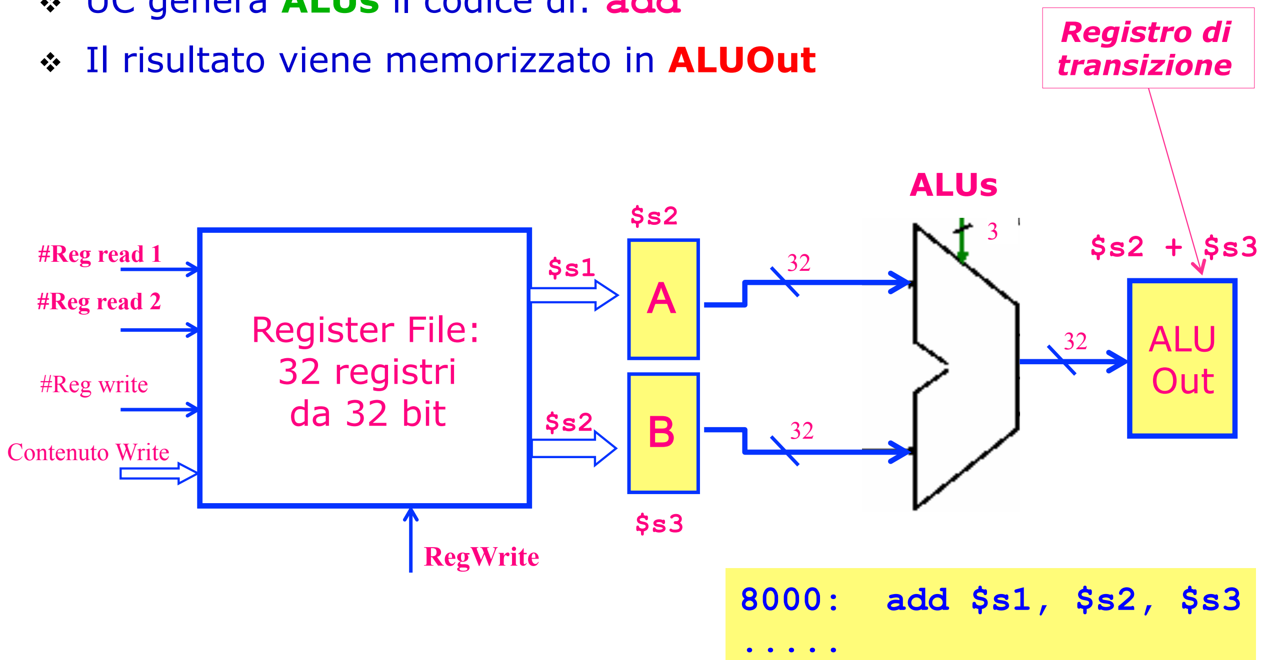
1. Leggo l'istruzione e genero i segnali di controllo opportuni.
2. Leggo il contenuto dei registri.

```
8000:  add $s1, $s2, $s3
8004:  sub $s4, $s1, $t1
.....
```



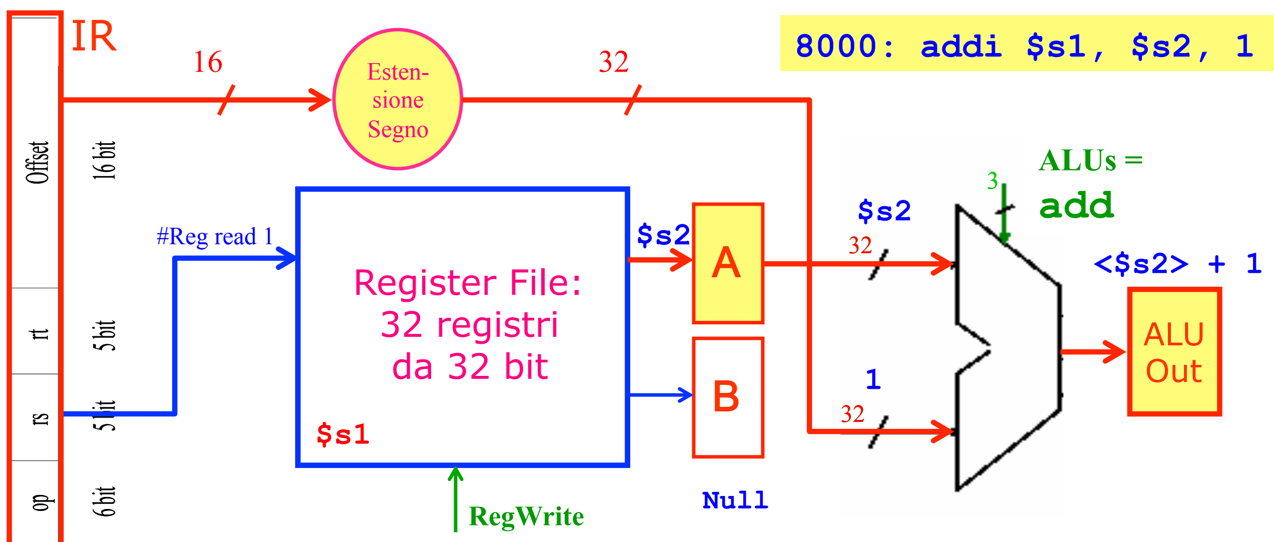
Fase di esecuzione **A/L – tipo R (fase 3)**

- Il contenuto dei registri **A** e **B** viene mandato all'ALU
- UC genera **ALUs** il codice di: **add**
- Il risultato viene memorizzato in **ALUOut**



Fase di esecuzione **A/L – tipo I (fase 3)**

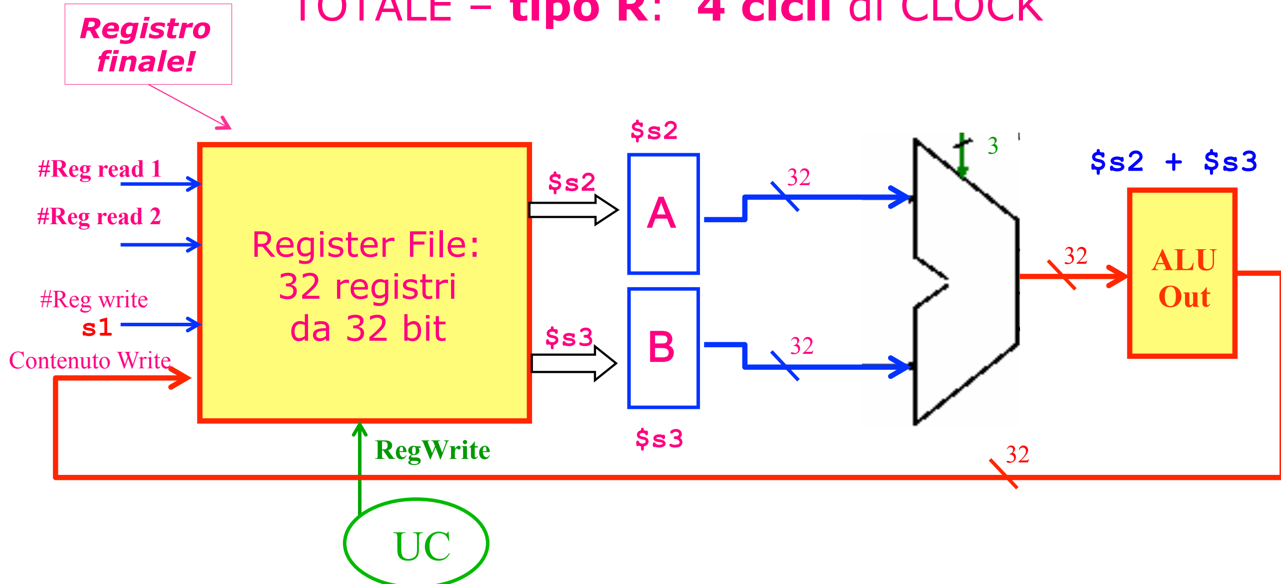
- Il contenuto di **A** e l' **immediato** (su 32 bit) arrivano all'ALU
- UC genera **ALUs** il codice di: **add**
- Il risultato viene memorizzato in **ALUOut**



Fase di Write-back – A/L (fase 4)

- Il contenuto di **ALUOut** viene mandato alla porta di scrittura del **RF**
- UC genera il segnale di: **RegWrite**

TOTALE – **tipo R**: **4 cicli** di CLOCK



Sommario

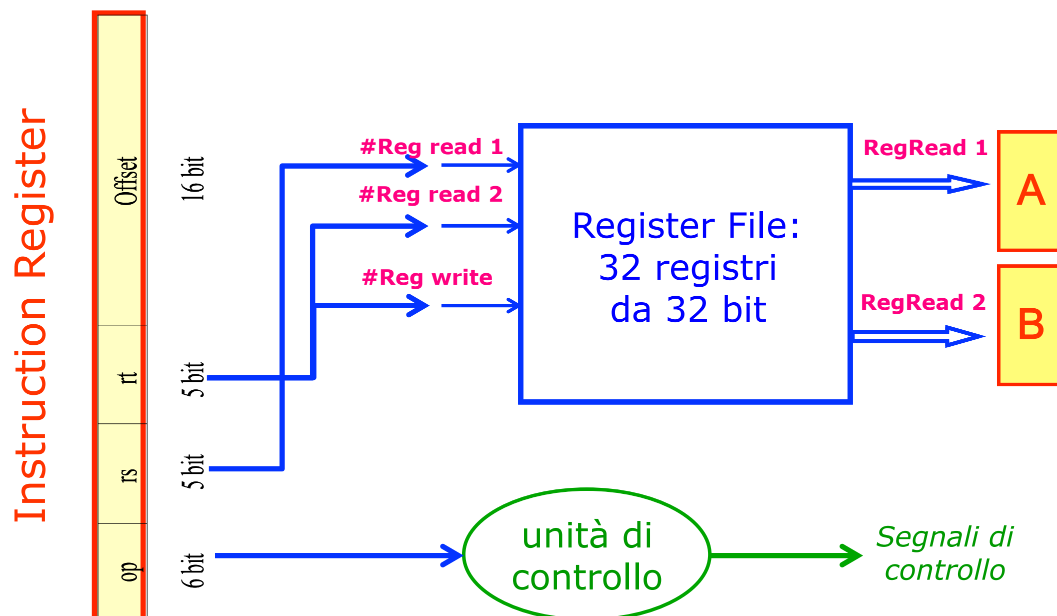
Fasi successive al FETCH:

Dalla fase successiva a quella di Fetch, consideriamo la situazione istruzione per istruzione:

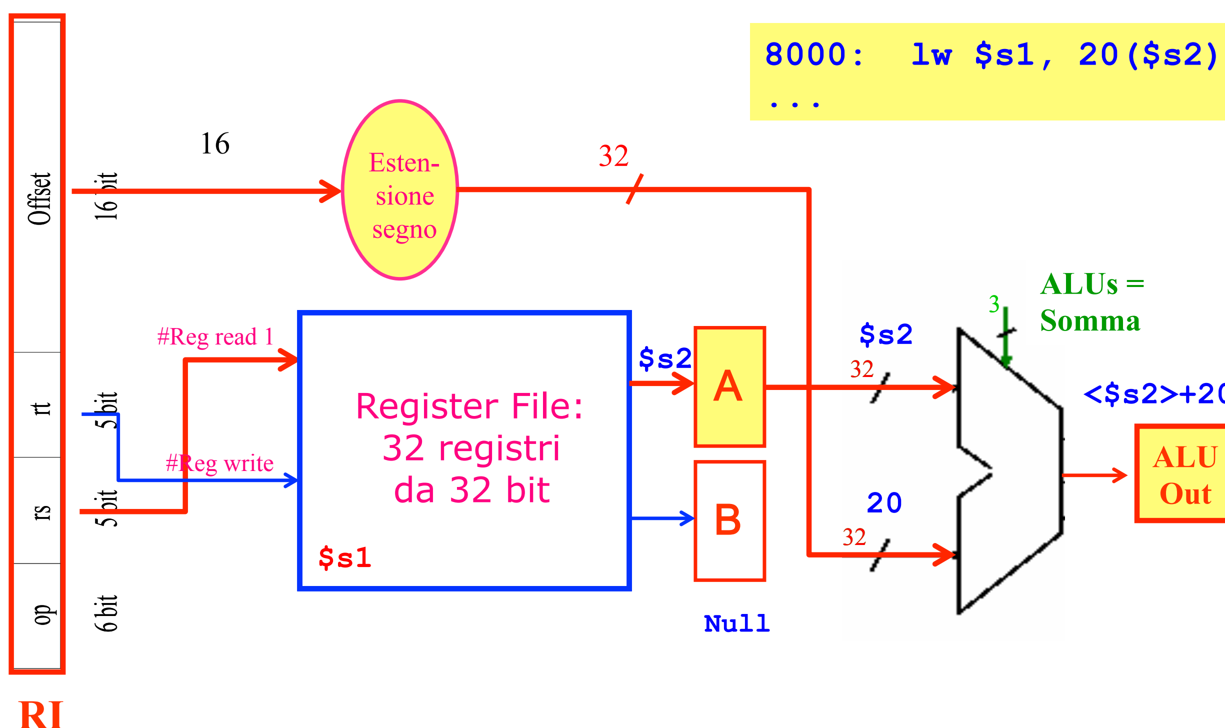
- Esecuzione multi-ciclo delle istruzioni A/L
- Esecuzione multi-ciclo delle istruzioni `lw/sw`
- Esecuzione multi-ciclo dei salti

Decodifica e lettura dei registri (*fase 2*): *tipo I*

1. Leggo l'istruzione e genero i segnali di controllo opportuni.
2. Leggo il contenuto dei registri.



Fase 3: esecuzione (tipo I: `lw`)

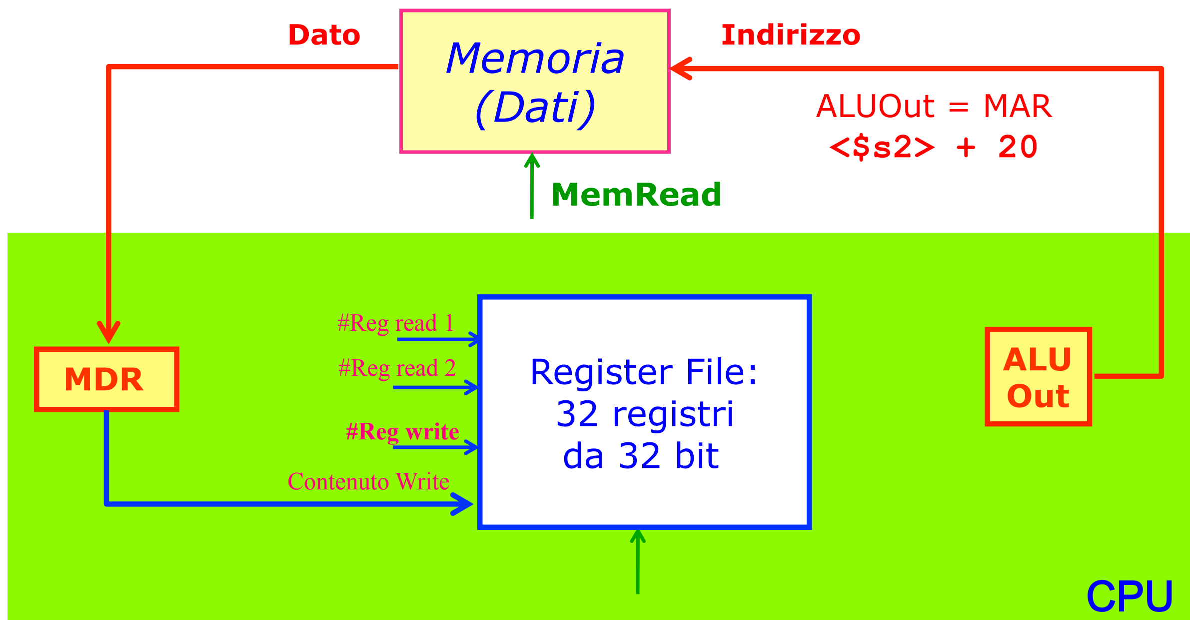


lw – Lettura memoria + write-back (fase 4,5)

FASE 4:

ALUOut → Mem → MDR

```
8000:  lw $s1, 20($s2)
...
```

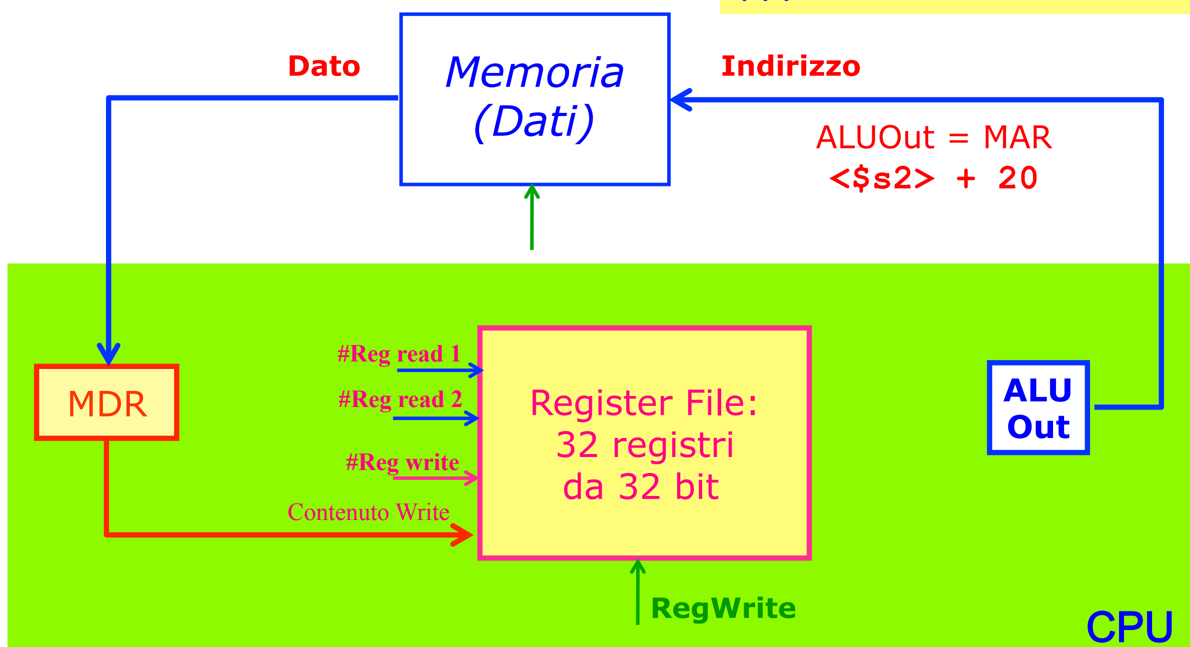


lw – Lettura memoria + write-back (fase 4,5)

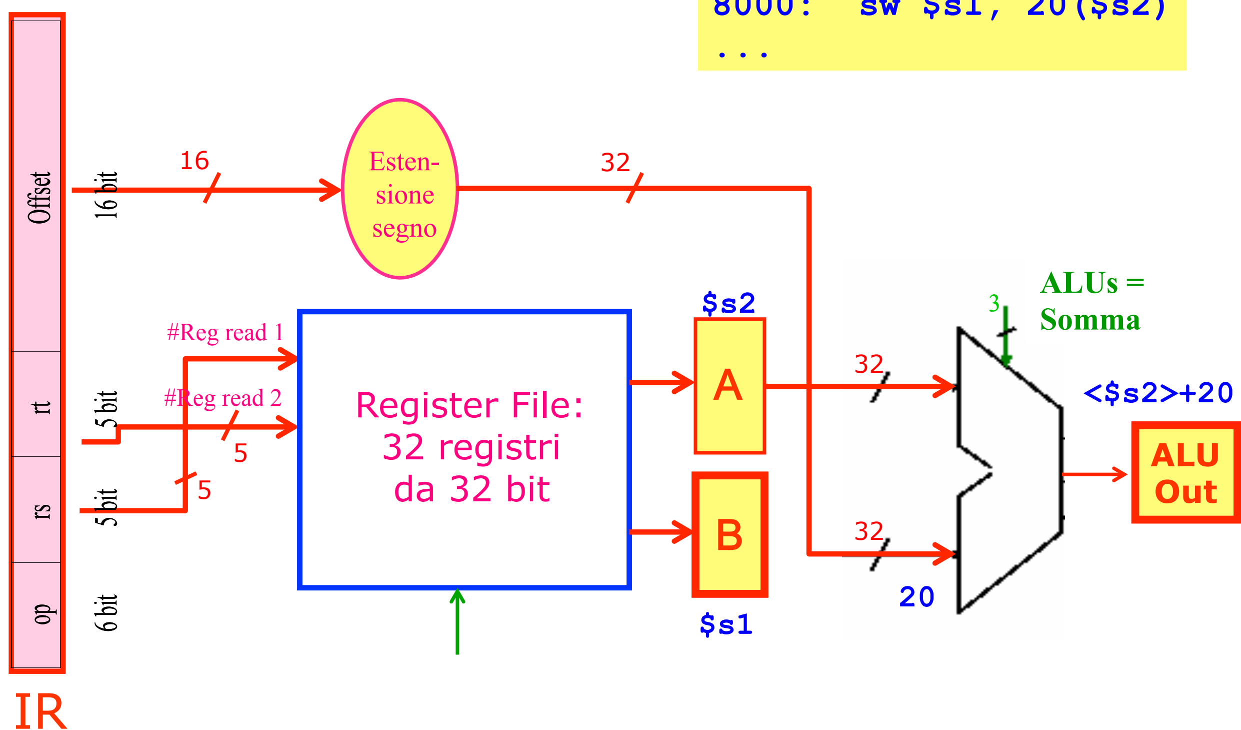
FASE 5: MDR → RegFile

TOTALE: 5 CC

```
8000:  lw $s1, 20($s2)
...
```



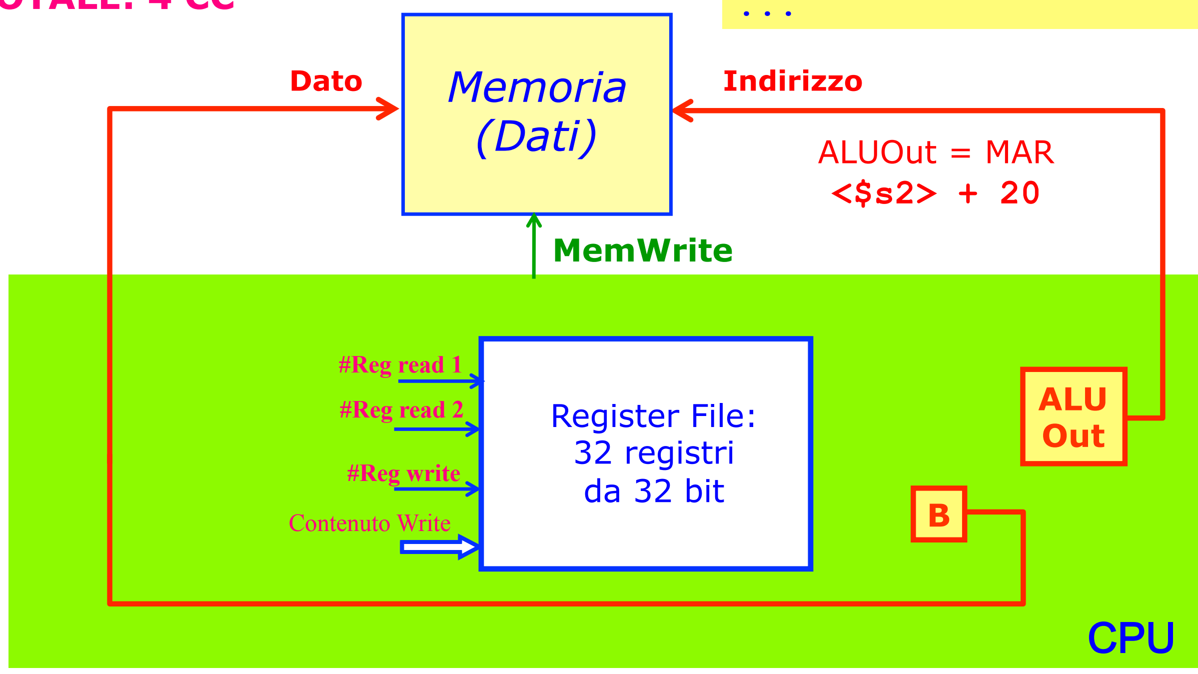
`sw`: *Fase 3: esecuzione (tipo I)*



`sw` : *Scrittura nella memoria*

1. ALUOut → Mem || MDR → Mem

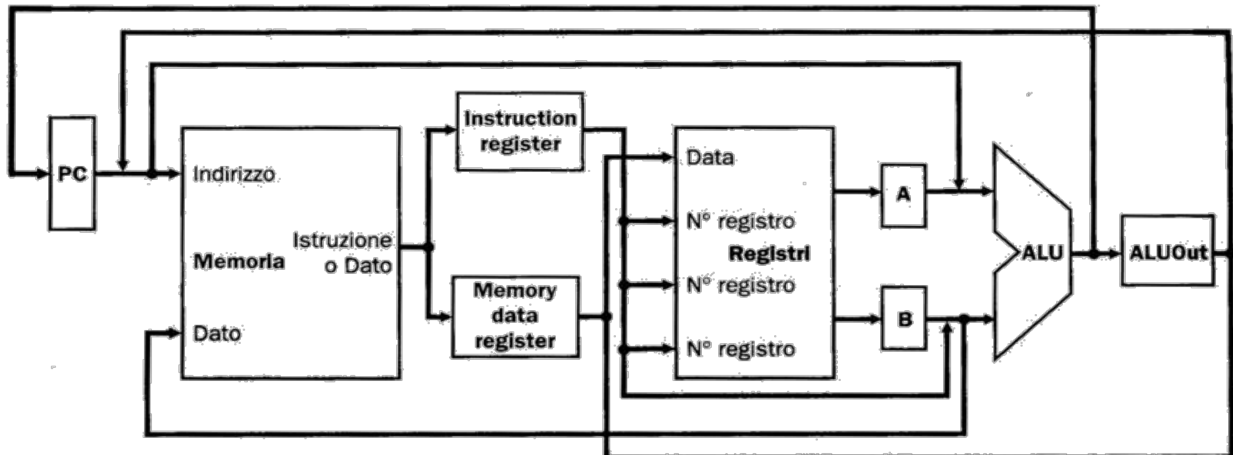
TOTALE: 4 CC



CPU multi-ciclo per istruzioni: A/L (tipo R), `lw/sw`

2 memorie → 1 memoria
a patto di creare 2 registri:
un **registro istruzione (RI)** ed un **registro dati (MDR)**

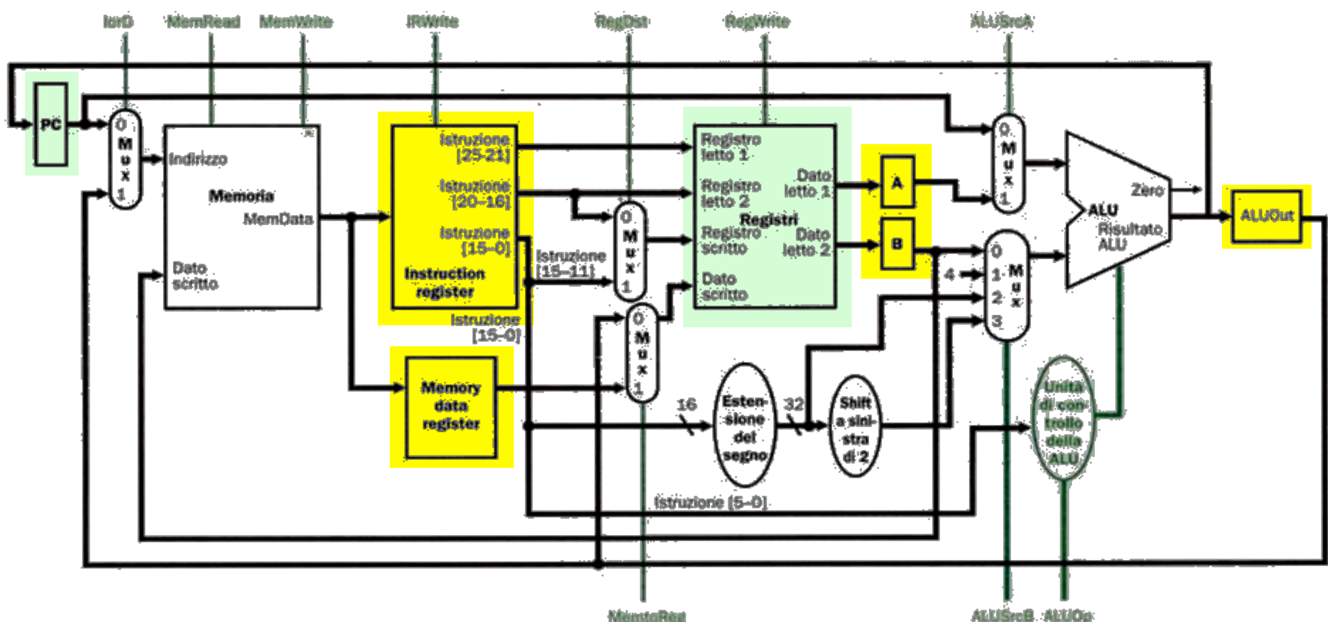
3 ALU → 1 ALU se le 3 funzioni vengono svolte in 3 momenti diversi:
- fase 1: fetch: aggiornamento PC (PC ← PC+4)
- fase 2: branch/lw/sw: `offset + rs`
- fase 3: operazioni A/L tipo R



CPU multi-ciclo per istruzioni: tipo R, `lw/sw`

Una sola ALU, una sola memoria (data + istruzioni)

5 nuovi registri: IR, MDR, A, B, ALUOut



Analisi dell'utilizzo dei registri

CPU Multi-ciclo: registri

- **PC** (Program Counter)**:** sincronizza il ciclo di esecuzione di un'istruzione. Viene scritto in fase di Fetch. Dall'inizio della fase di decodifica, contiene l'indirizzo dell'istruzione successiva.
- **IR** (Instruction Register)**:** viene aggiornato in fase di Fetch. Mantiene per tutte le fasi, il numero dei registri su cui operare.
- **A, B:** Memorizzano il contenuto dei registri letti 1 e 2
- **ALUOut:** Memorizza il risultato della ALU; viene letto durante le prime 3 fasi:
  - **FF** – contiene PC + 4
  - **DECOD** – contiene PC + Offset * 4
  - **EXEC** – contiene RegRead1 + [RegRead2 *oppure* (Offset x 4)]
- **MDR:** viene aggiornato solo nelle istruzioni **`lw/sw`**
- **Register File:** viene scritto nella fase di *WriteBack*, nelle istruzioni **R** e **`lw`**

Scrittura nei registri della CPU multi-ciclo

Quando vanno scritti i registri?

- **PC**: fase di ***Fetch***
  mantiene poi il suo valore per tutta la durata dell'istruzione
- **IR**: fase di ***Fetch***
  mantiene poi il suo valore per tutta la durata dell'istruzione
- **Register File**: fase di **Writeback**
  ma viene attivata soltanto nelle istruzioni: **A/L** e **`lw`**
- **Tutti gli altri** registri (**A**, **B**, **ALUout**, **MDR**):
  vengono scritti **ad ogni fase** (colpo di clock), quindi...
  non serve sintetizzare un **segnale di scrittura** di questi registri

  ➔ ***Comando di scrittura = clock!***

Esecuzione multi-ciclo delle *istruzioni di salto*

istruzione di salto → riscrittura del PROGRAM COUNTER

Il **PC** viene riscritto in **tre** diverse occasioni (e 3 diversi momenti):

1: incremento: **PC ← PC+4** **Fase 1** (tutte le istruzioni)

2: indirizzo di branch: **PC ← rs+**offset **Fase 3** (**branch**)

3: indirizzo di jump **PC ← IR+…** **Fase 3** (**jump**)

possibile conflitto di risorse!
ALU serve in fase 3 anche per il TEST

Cosa devo fare?

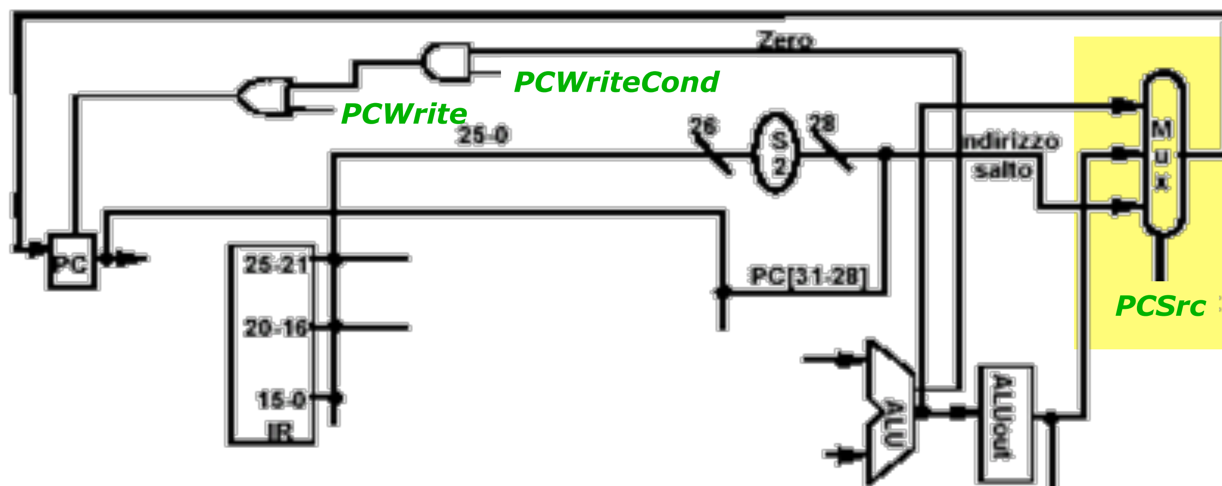
- aggiungere un circuito di selezione di operandi all'ingresso del PC
  → MUX in ingresso al PC
- comandare opportunamente la riscrittura del PC → segnale ***PCWrite***
- Utilizzare l'ALU in momenti diversi!

MUX: 3 possibili valori per PC → segnale **PCSrc** (2 bit)

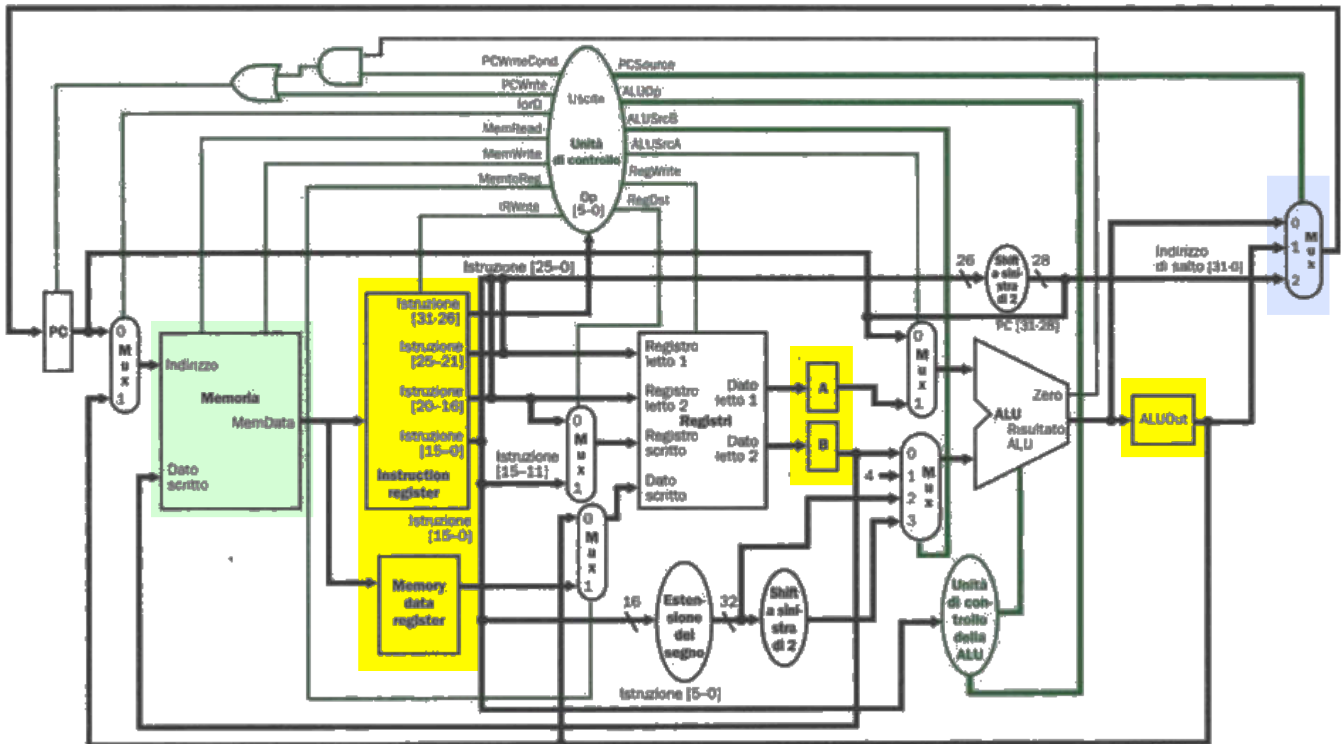
- Incremento PC (fetch) (fetch: PC ← PC+4)
- Indirizzo ottenuto dall'ALU (**branch**)
- Indirizzo ottenuto dall'IR (istr. tipo J) (**jump**)

PCWrite, PCSrc e PCWriteCond sono coordinati.

I 3 valori sono calcolati **in sequenza**.



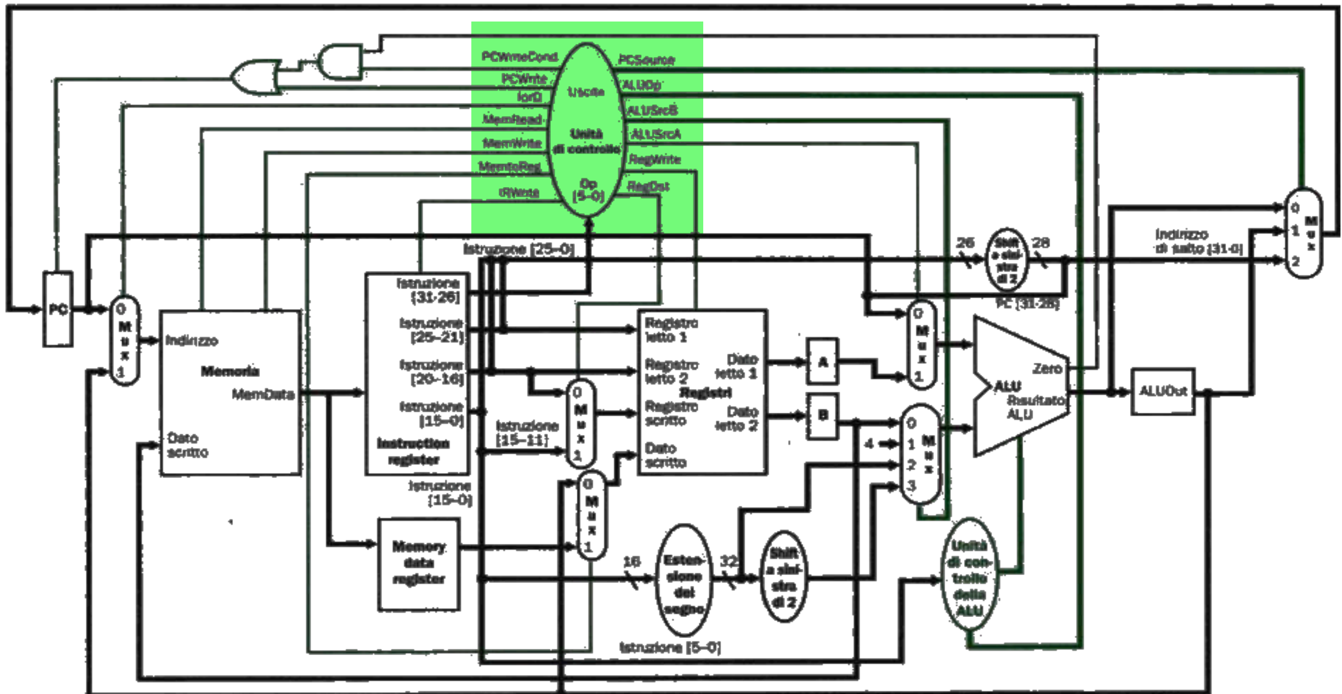
CPU multi-ciclo: schema completo



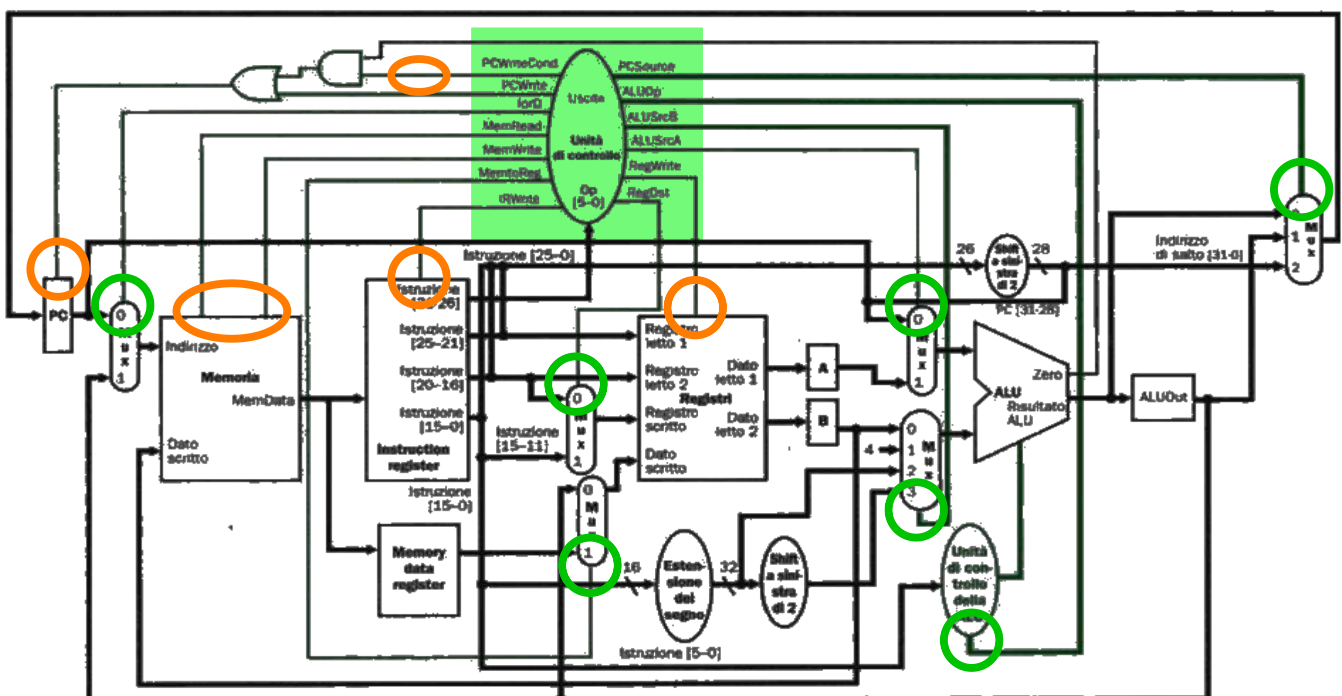
Sommario

- I segnali di controllo della CPU multi-ciclo
- Sintesi dell’Unità di Controllo

Struttura della CPU multi-ciclo



Struttura della CPU multi-ciclo



CPU Multi-ciclo: segnali di controllo

❖ ***Nuovi segnali***, rispetto alla CPU singolo ciclo:

Segnale	Valore	Effetto
ALUSrcA	**0**	1° operando è il valore attuale del PC
	1	1° operando proviene dalla porta di lettura 1 del RF
ALUSrcB	**00**	2° operando proviene dalla seconda porta di lettura del RF
	01	2° operando è la costante: **+ 4**
	10	2° operando è l'estensione del segno del campo **offset**
	11	2° operando: **offset** → estensione segno → shift sx di due posizioni
IorD	**0**	L'indirizzo della memoria proviene dal PC
	1	L'indirizzo della memoria proviene dalla ALU (**ALUOut**)
PCSrc	**00**	In PC viene scritta l'uscita della ALU (**PC + 4**)
	01	In PC viene scritta il contenuto di **ALUOut** (indirizzo di una **branch**)
	10	In PC viene scrittto l'indirizzo di destinazione della **jump**
PCWrite	**0**	–
	1	Viene scritto il registro **PC** (indirizzo in PC controllato da **PCSrc**)
PCWriteCond	**0**	–
	1	Il **PC** viene scritto se anche l'uscita **zero** della ALU è affermata

Segnali di controllo

❖ Segnali di **selezione**:

Segnale	Valore	Effetto
ALUSrcA	**0**	1° operando è il valore attuale del PC
	1	1° operando proviene dalla porta di lettura 1 del RF
ALUSrcB	**00**	ALU_b = seconda porta di lettura del RF
	01	ALU_b = **+ 4**
	10	ALU_b = **offset** → estensione segno
	11	ALU_b = **offset** → estensione segno → shift sx di due posizioni
IorD	**0**	L'indirizzo della memoria proviene dal PC
	1	L'indirizzo della memoria proviene dalla ALU (**ALUOut**)
PCSrc	**00**	In PC viene scritta l'uscita della ALU (**PC + 4**)
	01	In PC viene scritta il contenuto di **ALUOut** (indirizzo di una **branch**)
	10	In PC viene scrittto l'indirizzo di destinazione della **jump**
RegDst	**0**	Il registro da scrivere è definito nel campo **rt** (bit 20–16)
	1	Il registro da scrivere è definito nel campo **rd** (bit 15–11)
MemToReg	**0**	Il contenuto da scrivere nel RF è preso **dall'uscita della ALU**
	1	Il contenuto da scrivere nel RF è preso **dalla memoria (MDR)**

Segnali di controllo nella CPU multi-ciclo

- **7 Segnali di SELEZIONE**
  - **ALUSrcA:** Scelta del primo operando ALU (ALU **A**)
  - **ALUSrcB:** Scelta del secondo operando ALU (ALU **B**)
  - **IorD:** Selettore accesso memoria: Mem. Dati: **ALUOut** / Mem. Istruzioni: **PC**
  - **PCSrc:** Selettore del valore d'ingresso al PC
  - **RegDst**: Scelta del campo istruzione (**`rt/rd`**) che definisce il #RegWrite
  - **MemtoReg**: Selettore alla porta di scrittura del RF (**MDR / ALUOut**)
  - **AluOp:** Scelta dell'operazione nell'ALU

- **6 Segnali di COMANDO** (0: inattivo, 1: attivo)
  - **RegWrite**: Comando di scrittura nel RF (istruzioni con WriteBack)
  - **MemRead:** Comando di lettura Memoria
  - **MemWrite**: Comando di scrittura Memoria
  - **IRWrite:** Comando di scrittura dell' IR (durante la fase di fetch)
  - **PCWrite:** Comando di scrittura del PC
  - **PCWriteCond:** Abilitazione aggiornamento PC con indirizzo di salto

- Scrittura nei registri **A, B, ALUOut, MDR:**

  Comandata dal clock → non serve un segnale di controllo

Ciclo di esecuzione di un'istruzione

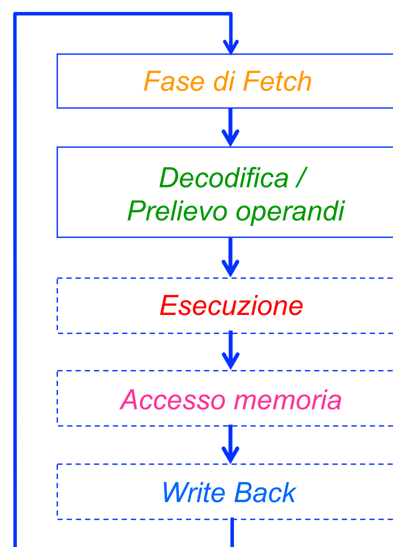
Le diverse **fasi del ciclo** vengono eseguite in **momenti diversi**

- L'esecuzione del ciclo richiede da **3 a 5 cicli di clock**
- L' **UC** deve "ricordare" tutte le fasi di ogni istruzione

UC** è una **macchina a stati finiti

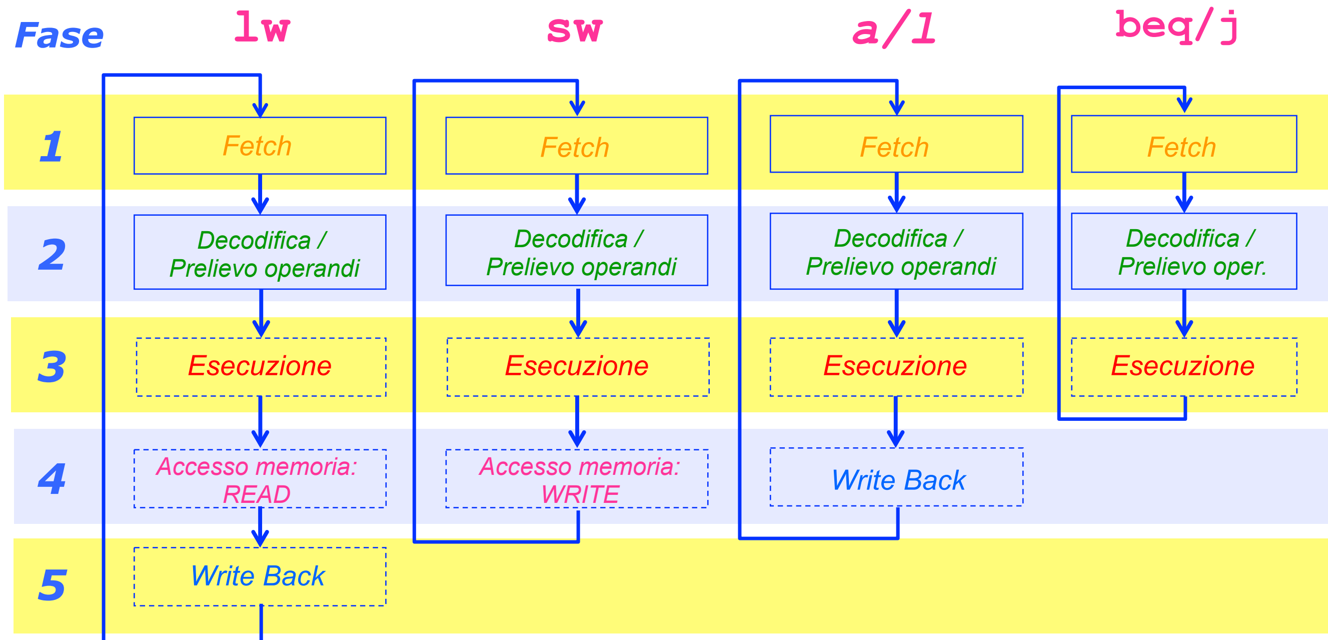
Criterio di suddivisione in fasi:

- Le operazioni elementari che hanno bisogno di unità funzionali diverse possono essere eseguite in **parallelo** (contemporaneamente, **nello stesso passo**
- Le operazioni elementari che hanno bisogno della stessa unità funzionale devono essere eseguite in **sequenza** (in **passi successivi** del ciclo di esecuzione)



Ciclo di esecuzione istruzioni

Suddivisione in fasi delle istruzioni:

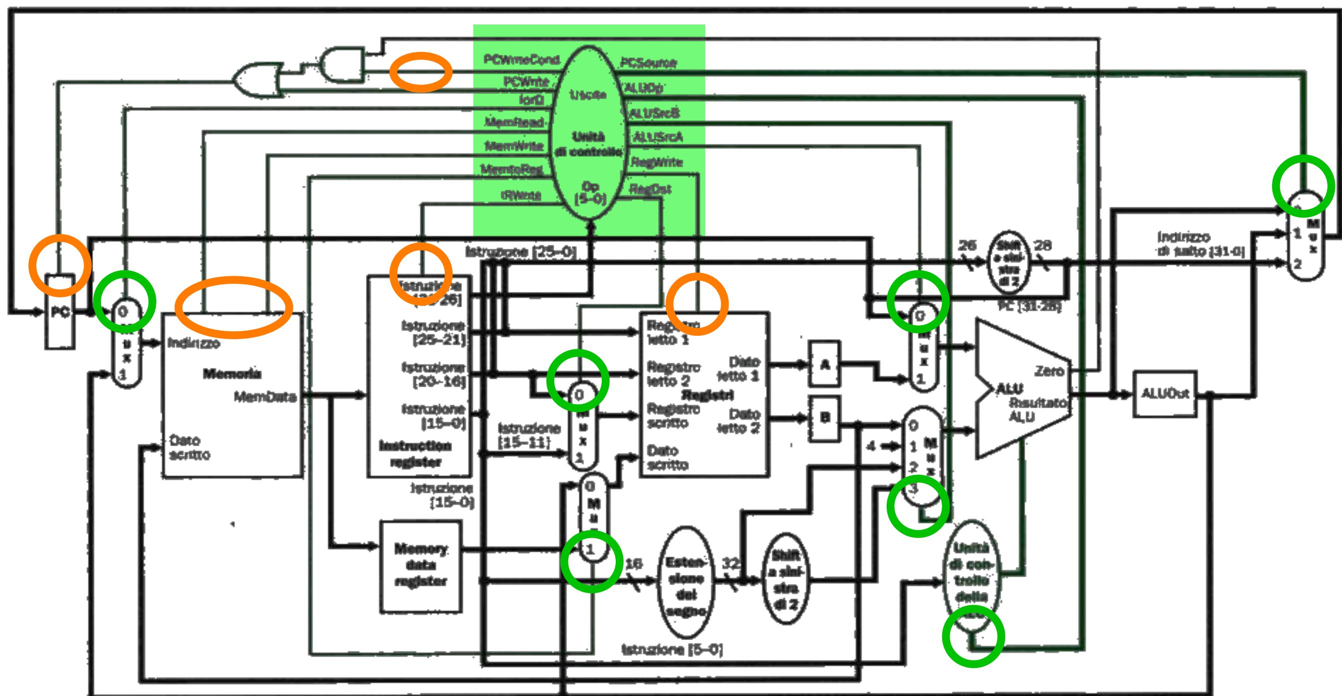


Riassunto dell'esecuzione – descrizione RTL

istruzioni→ fase ↓	Istruzioni A/L Tipo R	Istruz. di accesso a memoria (lw, sw)	Salto condizionato	Salto non condizionato
Fetch	IR ← Memory[PC] PC ← PC + 4			
Decodifica / Prelievo dati	A ← Reg[IR[25-21]] B ← Reg[IR[20-16]] ALUout ← PC + (sign_ext(IR[15-0]) << 2)			
Esecuzione	ALUOut ← A oper B	ALUOut ← A + sign_ext(IR[15-0])	If (A==B) then PC ← ALUOut	PC ← PC[31-28] || IR[25-0]<<2
lw/sw: mem tipo R: WB	Reg(IR[15-11])← ALUOut	lw: MDR←Memory[ALUOut] sw: Memory[ALUOut] ← B		
lw/sw: WB		lw: Reg[IR[20-16]] ← MDR		

Le istruzioni richiedono effettivamente da 3 a 5 cicli di clock

Struttura della CPU multi-ciclo



Macchina a Stati Finiti (di Moore)

- Una Macchina a Stati Finiti (MSF) è definita dalla quintupla:

$$< X, I, Y, f(\cdot), g(\cdot) >$$

X: insieme degli stati (in numero finito).

I: alfabeto di ingresso: l'insieme dei simboli che si possono presentare in ingresso. Con **n** ingressi, avremo 2^n possibili configurazioni.

Y: alfabeto di uscita: l'insieme dei simboli che si possono generare in uscita. Con **m** uscite, avremo 2^m possibili configurazioni

$f(\cdot)$: funzione stato prossimo: $X^* = f(X, I)$
Definisce l'evoluzione della macchina nel tempo, in modo deterministico

$g(\cdot)$: funzione di uscita: $Y = g(X)$ (macchina di **Moore**)
$Y = g(X, I)$ (macchina di **Mealy**)

- Per il buon funzionamento della macchina è previsto uno **stato iniziale**, al quale la macchina può essere portata mediante un comando di **reset**.

Sintesi della FSM della CPU

Ingressi (I) → codice operativo (OpCode), ALU.zero

Uscita (Y) → segnali di controllo

Stato (X) → istruzione corrente + fase di esecuzione

- I valori dei segnali di controllo (uscita) dipendono:
  - dalla fase di esecuzione dell'istruzione corrente → STATO

- Il passo successivo dell'istruzione (stato prossimo) dipende:
  - dal codice operativo → INGRESSO
  - dalla fase di esecuzione corrente → STATO

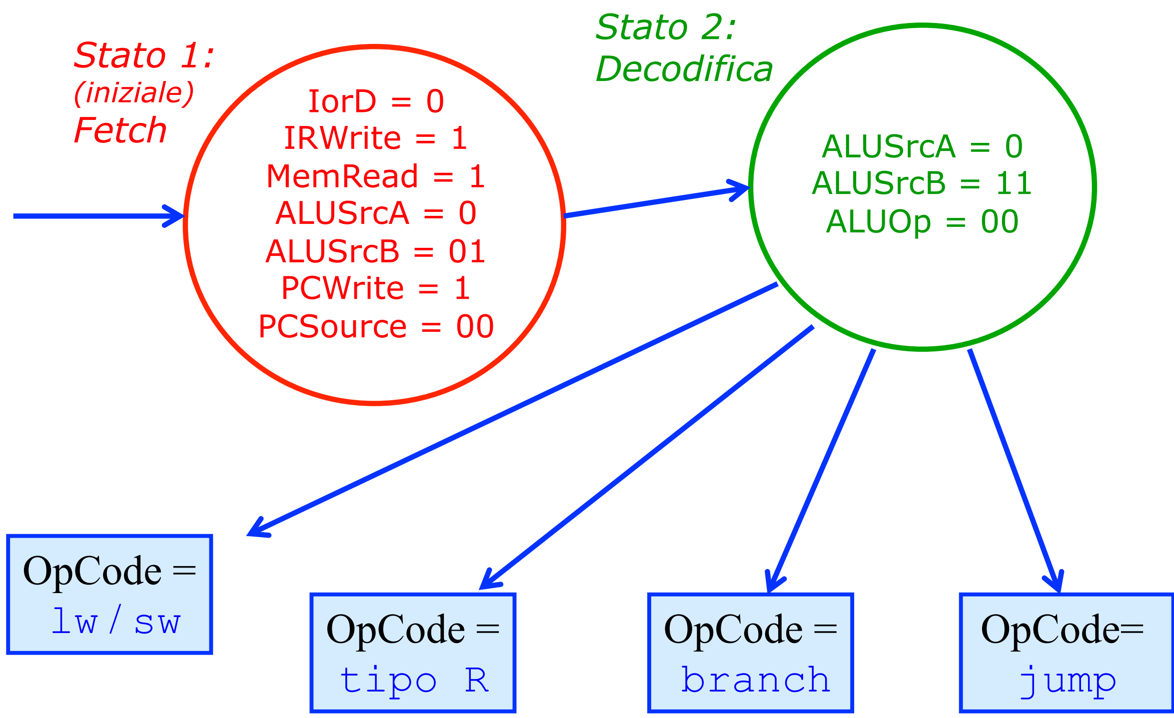
$Y = f(X)$
$X^* = f(I, X)$
→ Macchina di MOORE

Riassunto dell'esecuzione

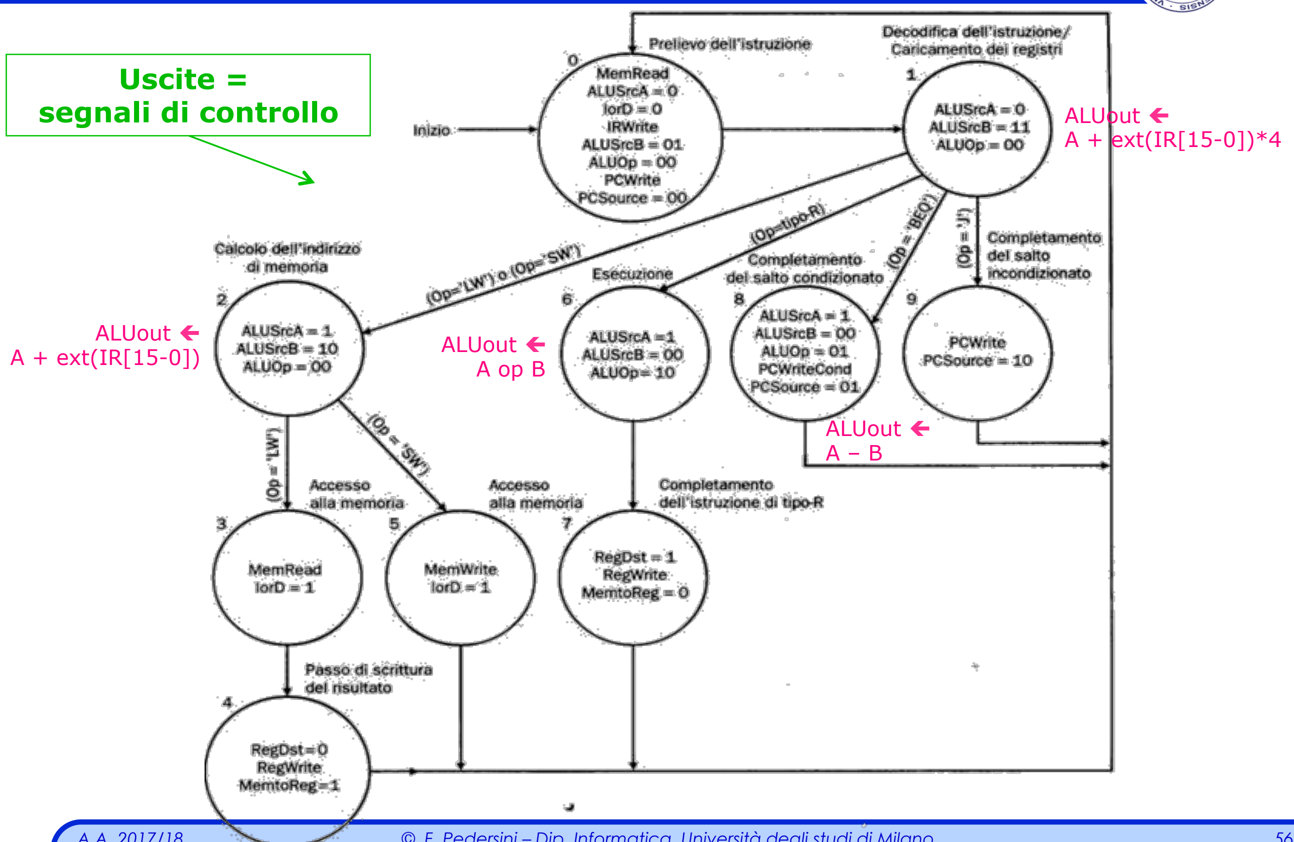
istruzioni→ fase ↓	Istruzioni A/L Tipo R	Istruz. di accesso a memoria (lw, sw)	Salto condizionato	Salto non condizionato
Fetch	IR ← Memory[PC] PC ← PC + 4			
Decodifica / Prelievo dati	A ← Reg[IR[25-21]] B ← Reg[IR[20-16]] ALUout ← PC + (sign_ext(IR[15-0]) << 2)			
Esecuzione	ALUOut ← A oper B	ALUOut ← A + sign_ext(IR[15-0])	If (A==B) then PC ← ALUOut	PC ← PC[31-28] || IR[25-0]<<2
lw/sw: mem tipo R: WB	Reg(IR[15-11])← ALUOut	lw: MDR←Memory[ALUOut] sw: Memory[ALUOut] ← B		
lw/sw: WB		lw: Reg[IR[20-16]] ← MDR		

Le istruzioni richiedono effettivamente da 3 a 5 cicli di clock

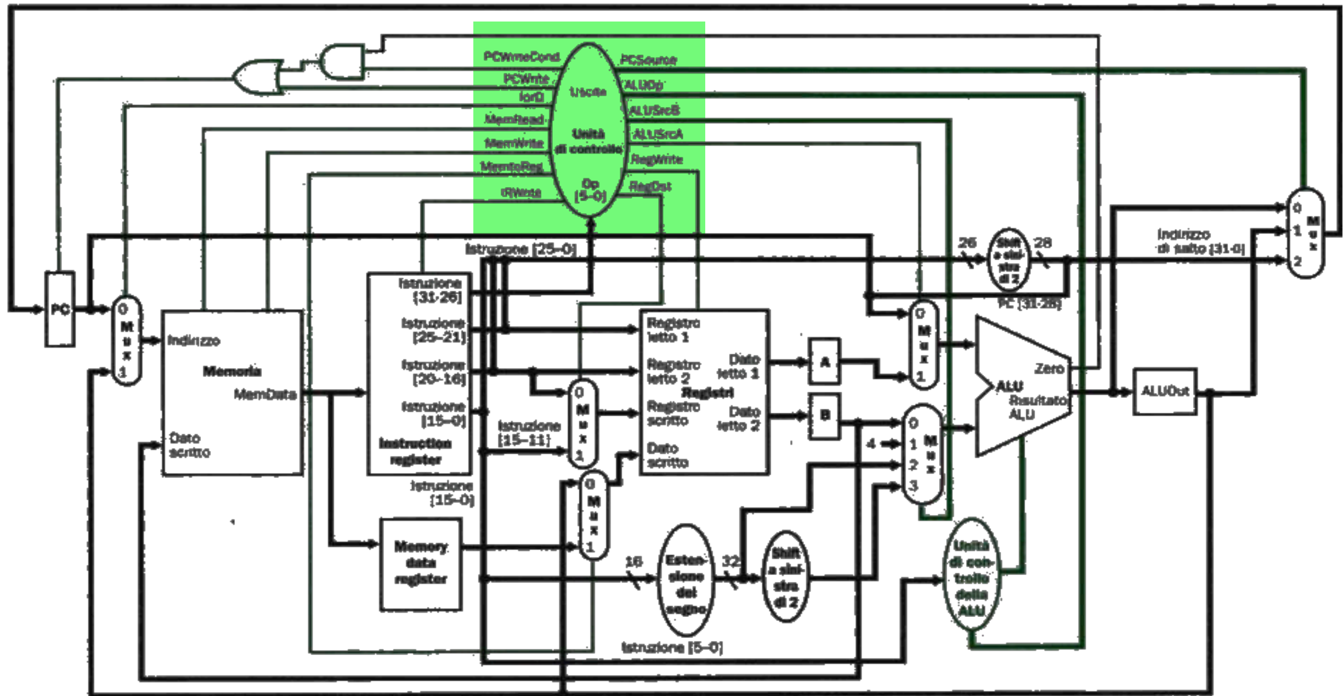
Fase di fetch e decodifica



FSM – State Transition Graph (STG)



CPU multi-ciclo



© F. Pedersini – Dip. Informatica, Università degli studi di Milano

Segnali di controllo: funz. uscita **g(X)** della FSM

Segnale controllo ➔ / Passo esecuzione ⬇	IorD	MemRead	MemWrite	IRWrite	ALUSrcA	ALUSrcB	ALUop	PCSource	PCWrite	PCWrCond	RegWrite	RegDst	MemtoReg
Fase fetch	**0**	**1**	0	**1**	0	01	00	00	**1**	0	0	X	X
Decodifica	X	0	0	0	0	11	00	X	0	0	0	X	X
Exec I - beq	X	0	0	0	1	00	01	01	0	**1**	0	X	X
Exec I – j	X	0	0	0	X	X	X	10	**1**	0	0	X	X
Exec I - R	X	0	0	0	1	00	10	X	0	0	0	X	X
Exec I – lw	X	0	0	0	1	10	00	X	0	0	0	X	X
Exec I - sw	X	0	0	0	1	10	00	X	0	0	0	X	X
Exec II – sw	**1**	0	**1**	0	X	X	X	X	0	0	0	X	X
Exec II – R	X	0	0	0	X	X	X	X	0	0	**1**	**1**	**0**
Exec II – lw	**1**	**1**	0	0	X	X	X	X	0	0	0	X	X
Exec III – lw	X	0	0	0	X	X	X	X	0	0	**1**	**0**	**1**

I segnali di selezione possono essere indifferenti, i segnali di comando no!

© F. Pedersini – Dip. Informatica, Università degli studi di Milano

FSM – State Transition Table (*STT*)

Ingressi (I) + Segnali (Y) → Passo esec. = Stato (X) ↓		OpCode = R	OpCode = sw	OpCode = lw	OpCode = beq	OpCode = j	IorD	MemRead	MemWrite	IRWrite	ALUScrA	ALUSrcB	ALUop	PCSource	PCWrite	PCWriteCond	RegWrite	RegDst	MemtoReg
Fase fetch	**0**						0	1	0	1	0	01	00	00	1	0	0	X	X
Decodifica	**1**						X	0	0	0	0	11	00	X	0	0	0	X	X
Exec I – beq	**8**						X	0	0	0	1	00	01	01	0	1	0	X	X
Exec I – jump	**9**						X	0	0	0	X	X	X	10	1	0	0	X	X
Exec I – a/l R	**6**						X	0	0	0	1	00	10	X	0	0	0	X	X
Exec I – lw/ sw	**2**						X	0	0	0	1	10	00	X	0	0	0	X	X
Exec II – sw	**5**						1	0	1	0	X	X	X	X	0	0	0	X	X
Exec II – R	**7**						X	0	0	0	X	X	X	X	0	0	1	1	0
Exec II – lw	**3**						1	1	0	0	X	X	X	X	0	0	0	X	X
Exec III – lw	**4**						X	0	1	0	X	X	X	X	0	0	1	0	1

FSM – State Transition Table (STT)

Ingressi (I) + Segnali (Y) → Passo esec. = Stato (X) ↓		OpCode = R	OpCode = sw	OpCode = lw	OpCode = beq	OpCode = j	IorD	MemRead	MemWrite	IRWrite	ALUScrA	ALUSrcB	ALUop	PCSource	PCWrite	PCWriteCond	RegWrite	RegDst	MemtoReg
Fase fetch	**0**	1	1	1	1	1	0	1	0	1	0	01	00	00	1	0	0	X	X
Decodifica	**1**	6	2	2	8	9	X	0	0	0	0	11	00	X	0	0	0	X	X
Exec I – beq	**8**	X	X	X	0	X	X	0	0	0	1	00	01	01	0	1	0	X	X
Exec I – jump	**9**	X	X	X	X	0	X	0	0	0	X	X	X	10	1	0	0	X	X
Exec I – a/l R	**6**	7	X	X	X	X	X	0	0	0	1	00	10	X	0	0	0	X	X
Exec I – lw/sw	**2**	X	5	3	X	X	X	0	0	0	1	10	00	X	0	0	0	X	X
Exec II – sw	**5**	X	0	X	X	X	1	0	1	0	X	X	X	X	0	0	0	X	X
Exec II – R	**7**	0	X	X	X	X	X	0	0	0	X	X	X	X	0	0	1	1	0
Exec II – lw	**3**	X	X	4	X	X	1	1	0	0	X	X	X	X	0	0	0	X	X
Exec III – lw	**4**	X	X	0	X	X	X	0	1	0	X	X	X	X	0	0	1	0	1

UC: *sintesi funzione uscita: Y = g(X)*

Segnali (Y) → **Stato (X) ↓**	$x_3\,x_2\,x_1\,x_0$	IorD	MemR	MemW	IRWrite	ALUSrcA	ALUSrcB	ALUop	PCSrc	PCWrite	PCWrCond	RegWrite	RegDst	MemtoReg
Fase fetch	**0: 0000**	**0**	**1**	**0**	**1**	**0**	**01**	**00**	**00**	**1**	**0**	**0**	X	X
Decodifica	**1: 0001**	X	**0**	**0**	**0**	**0**	**11**	**00**	X	**0**	**0**	**0**	X	X
Exec I – beq	**8: 1000**	X	**0**	**0**	**0**	**1**	**00**	**01**	**01**	**0**	**1**	**0**	X	X
Exec I – jump	**9: 1001**	X	**0**	**0**	**0**	X	X	X	**10**	**1**	**0**	**0**	X	X
Exec I – R	**6: 0110**	X	**0**	**0**	**0**	**1**	**00**	**10**	X	**0**	**0**	**0**	X	X
Exec I – lw,sw	**2: 0010**	X	**0**	**0**	**0**	**1**	**10**	**00**	X	**0**	**0**	**0**	X	X
Exec II – sw	**5: 0101**	**1**	**0**	**1**	**0**	X	X	X	X	**0**	**0**	**0**	X	X
Exec II – R	**7: 0111**	X	**0**	**0**	**0**	X	X	X	X	**0**	**0**	**1**	**1**	**0**
Exec II – lw	**3: 0011**	**1**	**1**	**0**	**0**	X	X	X	X	**0**	**0**	**0**	X	X
Exec III – lw	**4: 0100**	X	**0**	**0**	**0**	X	X	X	X	**0**	**0**	**1**	**0**	**1**

Esempi:

MemtoReg: (Stato=4)

RegWrite: (Stato=7) **or** (Stato=4)

ALUSrcA: (Stato=8) **or** (Stato=6) **or** (Stato=2)

$$MemtoReg = \overline{x_3} x_2 \overline{x_1}\,\overline{x_0} \quad \rightarrow \quad MemtoReg = \overline{x_0}$$

$$RegWrite = x_2\left(\overline{x_1}\,\overline{x_0} + x_1 x_0\right) = x_2\left(\overline{x_1} \oplus x_0\right)$$

$$ALUsrcA = \left(\overline{x_3} x_1 + x_3 \overline{x_2}\,\overline{x_1}\right)\overline{x_0} \quad \rightarrow \quad = x_3 + x_1$$

Sintesi stato futuro

❖ **Funzione stato futuro: X* = *f* (X , I)**

➢ devo codificare gli ingressi (OpCode) e lo stato (passo esec.):

OpCode (I) → **(X) ↓**		R: 000000	sw: 101011	lw: 100011	beq: 000100	jump: 000010
Passo esec.	$x_3 x_2 x_1 x_0$	$x_3^* x_2^* x_1^* x_0^*$				
Fase fetch	0: 0000	0001	0001	0001	0001	0001
Decodifica	1: 0001	0110	0010	0010	1000	1001
Exec I – beq	8: 1000	X	X	X	0000	X
Exec I – jump	9: 1001	X	X	X	X	0000
Exec I – a/l R	6: 0110	0111	X	X	X	X
Exec I – lw/sw	2: 0010	X	0101	0011	X	X
Exec II – sw	5: 0101	X	0000	X	X	X
Exec II – R	7: 0111	0000	X	X	X	X
Exec II – lw	3: 0011	X	X	0100	X	X
Exec III – lw	4: 0100	X	X	0000	X	X

STT Codificata – sintesi "stato futuro": X* = f(X,I)

Sintesi di $x_i^* = f_i(X,I), \quad i = 0..3: \quad X = \langle x_3, x_2, x_1, x_0 \rangle, \quad I = \langle i_5, i_4, i_3, i_2, i_1, i_0 \rangle$

OpCode (I) → (X) ↓		R: 000000	sw: 101011	lw: 100011	beq: 000100	jump: 000010
Passo esec.	$x_3x_2x_1x_0$	$x_3^*x_2^*x_1^*x_0^*$				
Fase fetch	0: 0000	0001	0001	0001	0001	0001
Decodifica	1: 0001	0110	0010	0010	1000	1001
Exec I – beq	8: 1000	X	X	X	0000	X
Exec I – jump	9: 1001	X	X	X	X	0000
Exec I – a/l R	6: 0110	0111	X	X	X	X
Exec I – lw/sw	2: 0010	X	0101	0011	X	X
Exec II – sw	5: 0101	X	0000	X	X	X
Exec II – R	7: 0111	0000	X	X	X	X
Exec II – lw	3: 0011	X	X	0100	X	X
Exec III – lw	4: 0100	X	X	0000	X	X

Sintesi di: x_0^*

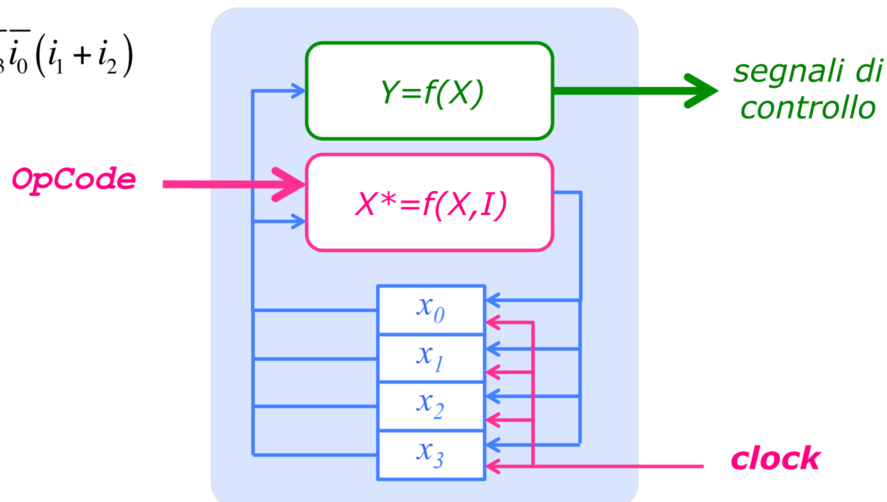
Sintesi: X* = f(X , I)

Sintesi di $x_i^* = f_i(X,I), \quad i = 0..3: \quad X = \langle x_3, x_2, x_1, x_0 \rangle, \quad I = \langle i_5, i_4, i_3, i_2, i_1, i_0 \rangle$

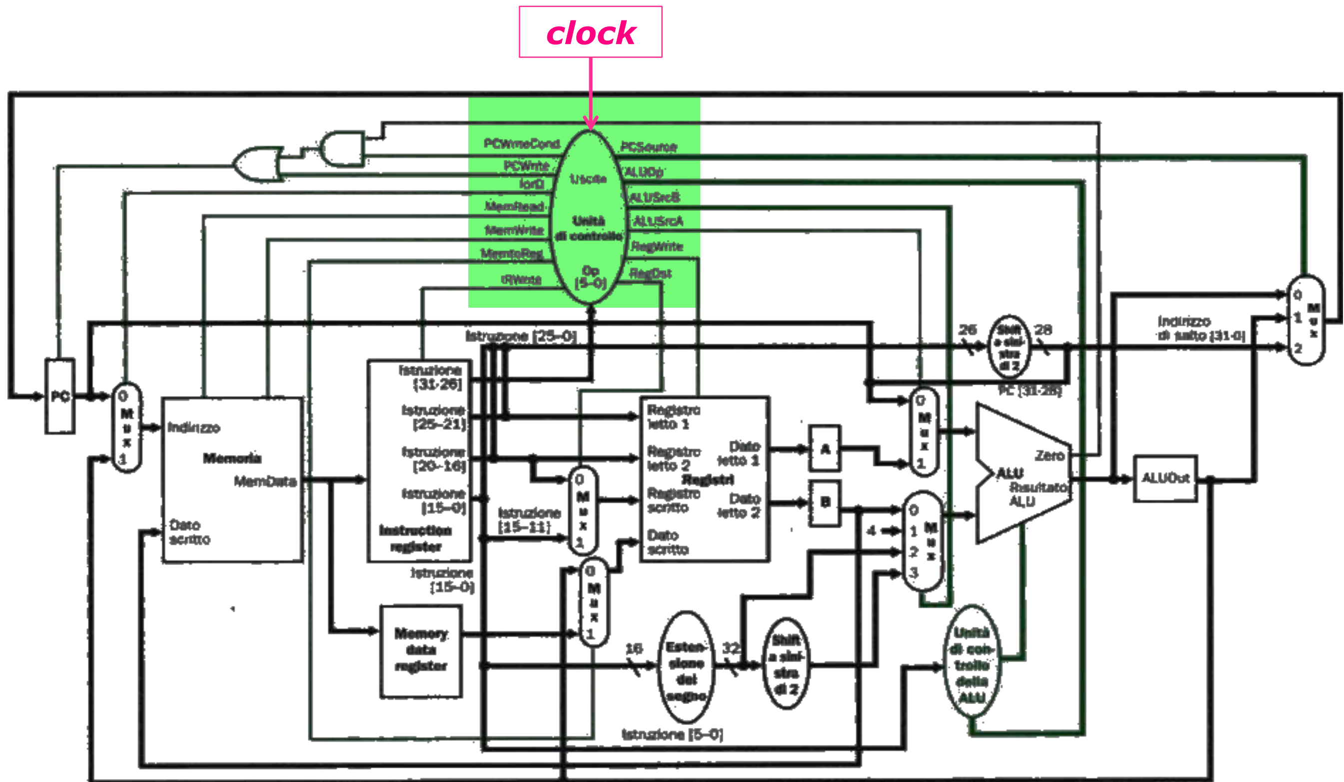
$$x_0^* = \overline{x_0}\,\overline{x_1}\,\overline{x_2}\,\overline{x_3} + \overline{x_0}x_1\overline{x_3} + \overline{i_0}i_1\overline{x_3} = \overline{x_3}\left(\overline{x_0}\left(\overline{x_1}\,\overline{x_2} + x_1\right) + \overline{i_0}i_1\right) = \overline{x_3}\left(\overline{x_0}\left(\overline{x_2} + x_1\right) + \overline{i_0}i_1\right)$$

$\vdots$

$$x_3^* = x_0\overline{x_1}\,\overline{x_2}\,\overline{x_3}\,\overline{i_0}\left(i_1 + i_2\right)$$



CPU multi-ciclo – struttura finale



Eccezioni ed Interrupt

Eccezioni in una CPU

ECCEZIONE: evento che altera l'esecuzione di un programma

- **Eccezioni** – Generate internamente al processore (es. ALU overflow)
- **Interrupts** – Generati esternamente al processore (es. segnali da periferiche)

Tipo di evento	Provenienza	Terminologia MIPS
Uso di un'istruzione non definita	interna	*Eccezione*
Overflow da istruzione aritmetica	interna	*Eccezione*
Chiamata al S.O. da parte di un programma (es. `syscall`)	interna	*Eccezione*
Richiesta di un dispositivo di I/O	esterna	*Interrupt*
Malfunzionamento hardware	int./est.	*Eccezione / Interrupt*

Come si deve comportare la CPU a fronte di eccezioni?

Azioni in risposta ad un'eccezione

Come reagisce la CPU ad un'eccezione?

Operazioni da svolgere:

1. *Salvataggio **stato corrente** della CPU (e cioè?)*
2. *Riconoscimento eccezione*
3. *Gestione eccezione*
3. *Ripristino dello "stato corrente"*

Gestione dell'eccezione: è un'operazione "delicata" da svolgere via software.
➔ (per questo le eccezioni vengono gestite dal sistema operativo)

Come avviene il "passaggio" alla gestione dell'eccezione?

1. Salvataggio dell'indirizzo attuale, dell'istruzione coinvolta
   indirizzo istruzione corrente: **`$PC – 4`**
2. Trasferimento (➔ jump) del controllo a un altro programma
   di solito parte del Sistema Operativo – tale programma deve gestire la condizione di eccezione e quindi ripristinare il programma interrotto.

Gestione eccezioni: esempi

Esempio eccezioni MIPS

Esempio gestione 2 tipi di eccezione interna:

1. **Istruzione non valida** (che non esiste nel Set Istruzioni):
   - Non esiste, al passo 2 (stato 1 – Decodifica), uno stato futuro valido.
   - Nuovo stato futuro denominato: "**Invalid OpCode**"
     - Tale stato viene raggiunto al verificarsi dell'eccezione.
2. **Overflow aritmetico**:
   - Al passo 3 di esecuzione dell'operazione
     (**ALU** ha eseguito l'operazione – manca la fase di Write-back nel RF)
     lo stato futuro è scelto in funzione del segnale di Overflow
   - **Overflow** → input aggiuntivo alla FSM della **UC**
   - Necessità di uno stato aggiuntivo: "**Overflow**"

Hardware addizionale:

Ci servono componenti hardware aggiuntivi:

- **Registro EPC** (Exception Program Counter)
  - registro a 32 bit utilizzato per memorizzare l'indirizzo dell'istruzione in esecuzione al momento dell'eccezione/interruzione

- **Registro Causa**
  - registro utilizzato per memorizzare la **causa** dell'eccezione; (MIPS: 32 bit)

    RC_0 = 0 → Causa: Istruzione indefinita (*Illegal OpCode*)

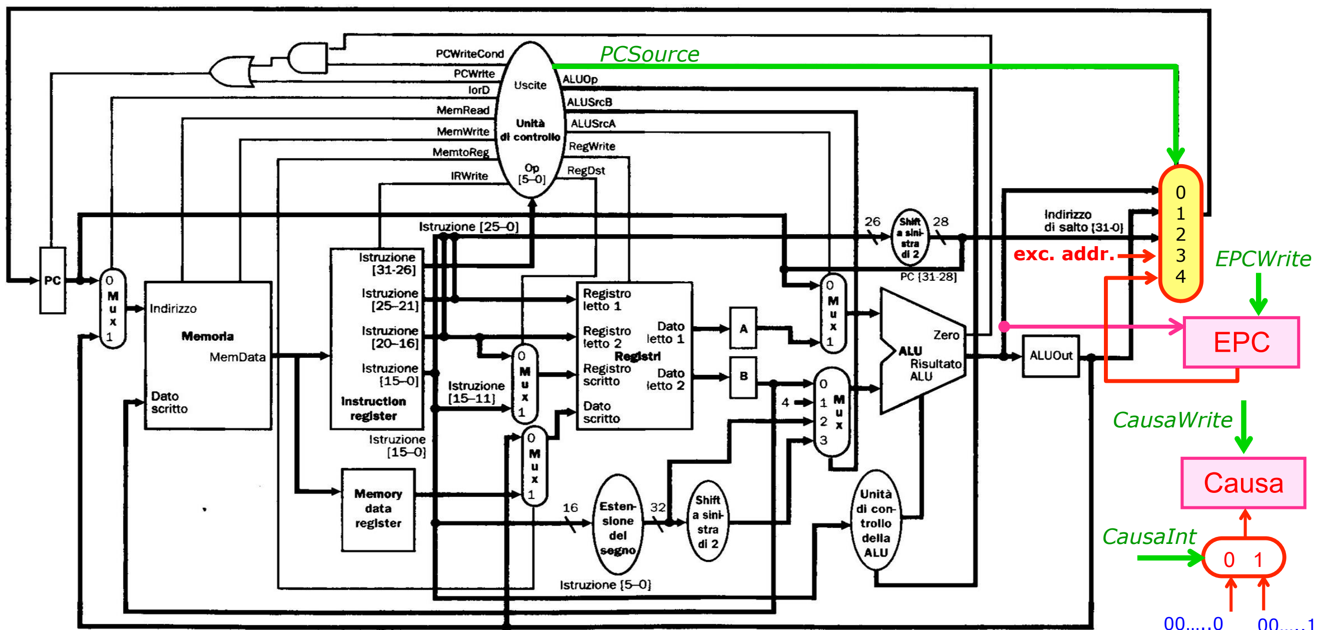
    RC_0 = 1 → Causa: Overflow aritmetico

- Segnali di controllo aggiuntivi
  - ***EPCWrite*** – scrittura nel registro EPC.
  - ***CausaWrite*** – scrittura nel registro Causa.
  - ***CausaInt*** – dato per il registro Causa.

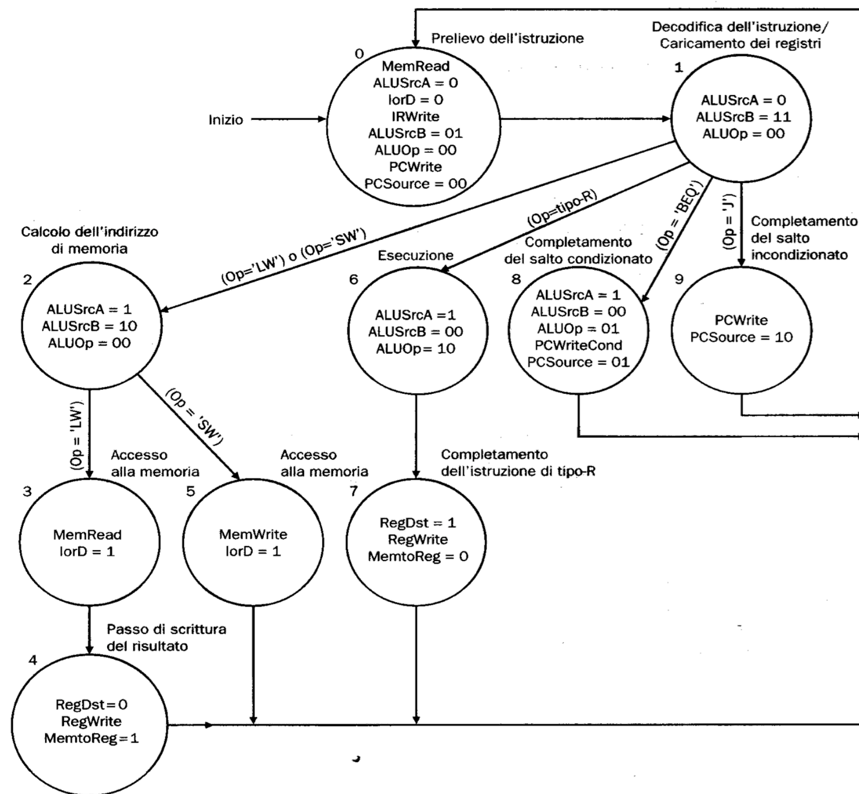
- Ingressi aggiuntivi all'Unità di Controllo
  - Aggiunta di **Overflow** (proveniente dalla ALU)

HW per risposta ad eccezioni

- Salvataggio del PC:
  - ***EPC*** ← PC – 4;
  - Selezione della **causa** (0…0 , 0…1)
  - **PC** ← **exception address**;



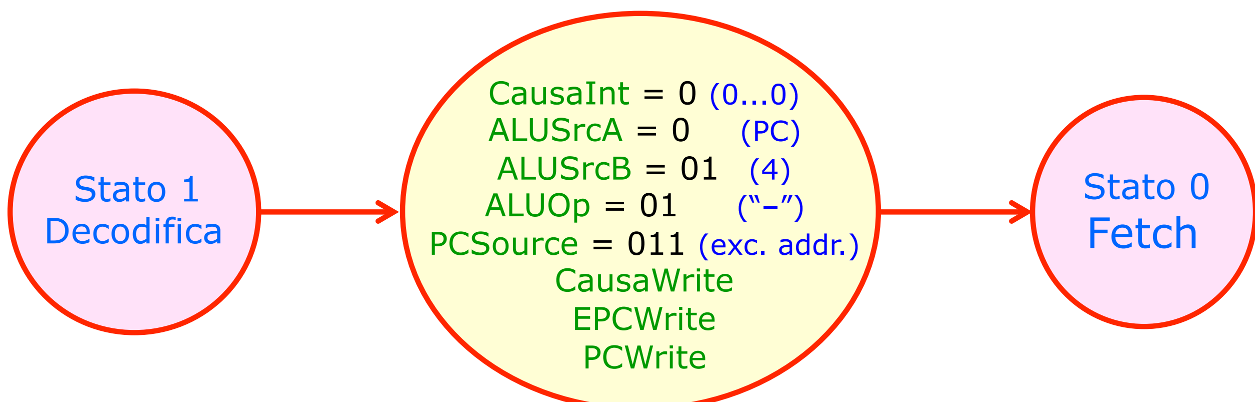
FSM di CPU Multi-ciclo: – STG



Stato 10: *Invalid OpCode*

- Nuovo stato nella FSM dell'unità di controllo:
  - **Stato: "Invalid OpCode" – stato 10**

Stato 10:
invalid OpCode



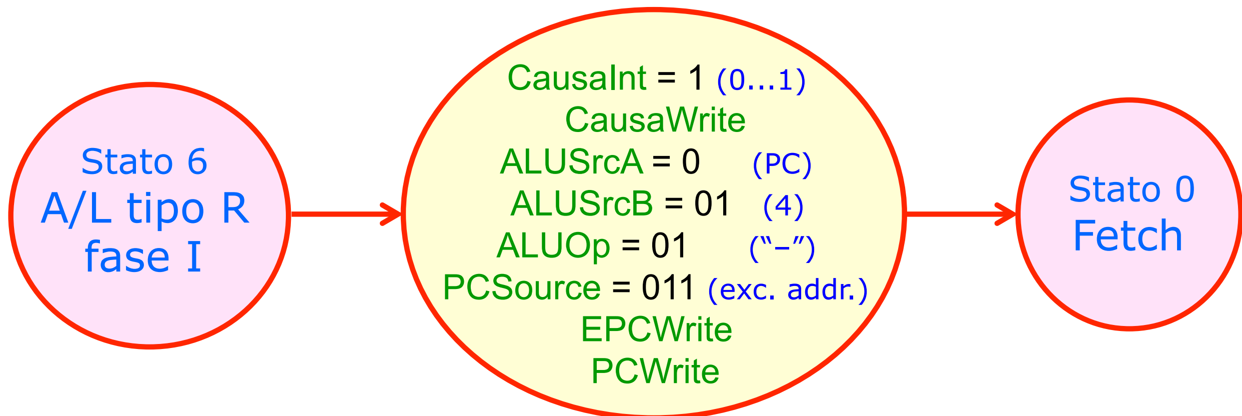
Stato11: Overflow

- Nuovo stato nella FSM dell'unità di controllo:

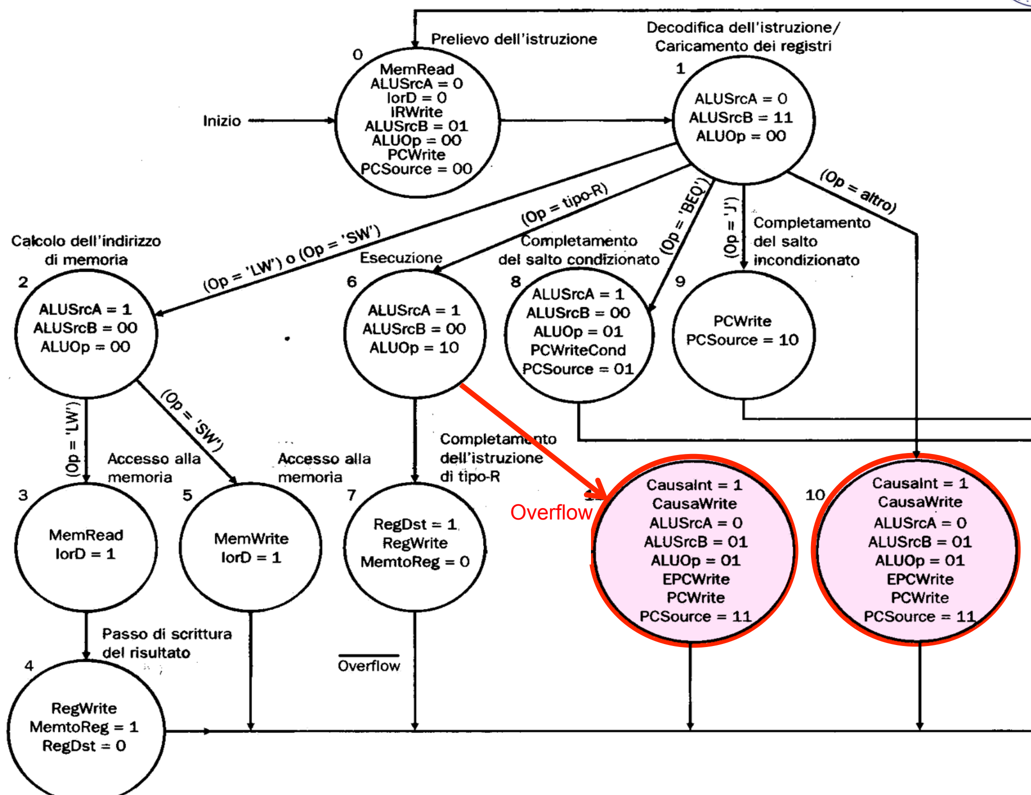
  "Overflow": stato 11

  Raggiungibile solo per istruzioni A/L, dopo l'esecuzione del calcolo (Esecuzione – fase II)

Stato 11: Overflow



CPU multi-ciclo con gestione eccezioni: FSM



Risposta ad un'eccezione

Eccezione → risposta software, da parte del programma di gestione della/delle eccezioni (spesso nel S.O.)

Tecniche di gestione della risposta all'eccezione:

- **A registro** (MIPS: ***registro causa***):
  - Il SO ha un unico *entry point* per la gestione delle eccezioni.
  - La prima istruzione è di decodifica della causa dell'eccezione
  - L' *entry point* è forzato tramite: `PCSource = 011`
- **Vettorializzata:**
  - Ogni eccezione rimanda ad un indirizzo diverso, dove si trovano funzioni diverse di servizio all'eccezione (nel Sistema Operativo).
  - Il diverso indirizzo definisce implicitamente la causa dell'eccezione
  - Gli indirizzi sono in genere equispaziati (es. ogni 256 parole)

Eccezione	Vettore di salto (al S.O.)
"Invalid OpCode"	0x 0000 0100
"Overflow"	0x 0000 0200

Risposta ad un'eccezione

Eccezione → risposta **software, da parte del programma di gestione**

Ritorno dall'eccezione

- Al termine della procedura di gestione da parte del S.O., occorre tornare a eseguire l'istruzione originariamente interrotta.

MIPS32: istruzione `rfe` – Return from Exception

`rfe`

010000	10 ... 0	100000
6	20	6

- Effettua un salto incondizionato all'indirizzo contenuto nel registro Exception Program Counter (EPC)
- Datapath RTL: **PC ← EPC**

MIPS32: Co-processore 0

Coprocessore 0 – gestore eccezioni, presente nelle architetture MIPS

- Raccolta e gestione delle informazioni e gli indirizzi per la gestione delle eccezioni

- Registri:

Registro	N. registro di **C0**	Utilizzo / Contenuto
Bad/Addr	**8**	Indirizzo di memoria (dati) a cui si è fatto riferimento
Stato	**12**	Maschera delle interruzioni e bit di abilitazione
Causa	**13**	Tipo di interruzione / interruzione pendente
EPC	**14**	Indirizzo istruzione che ha causato l'interruzione

- Istruzioni che interessano il coprocessore 0 (`c0`):
  - Lettura/scrittura da **memoria** a **registri di `c0`**

```
lwc0 $<reg_c0> <offset>($reg)
swc0 $<reg_c0> <offset>($reg)
```

  - Move from/to, tra **registri** general-purpose e **registri di `c0`**

```
mfc0 $<reg>, $<reg_c0> (pseudo-istruzione)
mtc0 $<reg_c0>, $<reg> (pseudo-istruzione)
```