

Architettura degli Elaboratori

Progetto CPU (multiciclo)

slide a cura di Salvatore Orlando e Marta Simeoni

Problemi con progetto a singolo ciclo

Problemi del singolo ciclo

- Ciclo di clock **lungo**
- Istruzioni potenzialmente più veloci sono rallentate
 - impiegano lo stesso tempo dell'istruzione più lenta
- Unità funzionali e collegamenti del Datapath sono **replicati**
 - dobbiamo poter eseguire in parallelo tutti i passi computazionali necessari per l'esecuzione di qualsiasi istruzione dell'ISA

Problemi con progetto a singolo ciclo

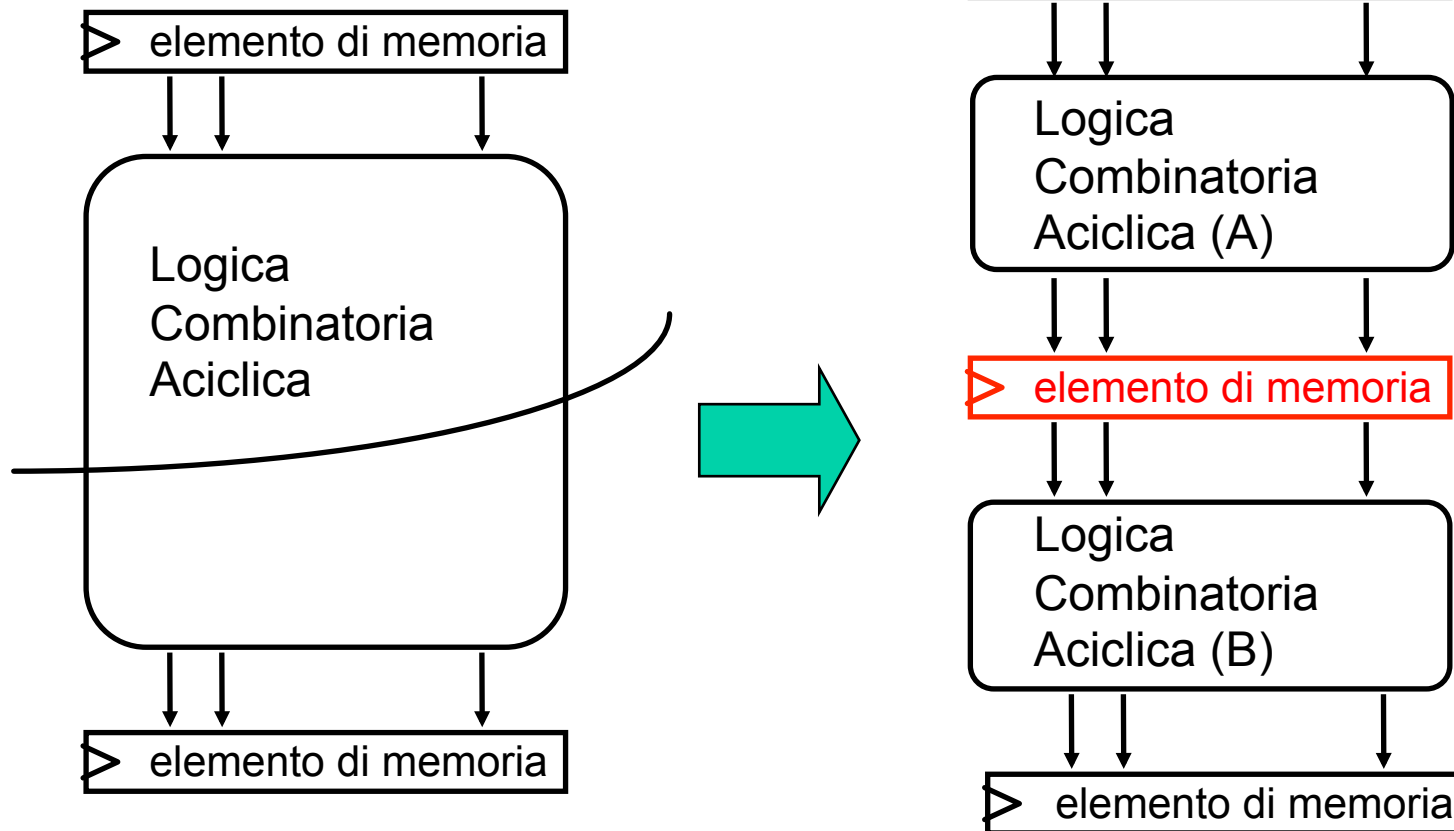
Possibile soluzione:

- datapath “multiciclo”
- usiamo un ciclo **più corto**
- istruzioni differenti impiegano un numero di cicli diversi
- unità funzionali possono essere usate più volte per eseguire la stessa istruzione \Rightarrow meno *replicazione*
 - basta usarle in cicli differenti
- registri aggiuntivi
 - usati per memorizzare i risultati parziali nell'esecuzione delle istruzioni

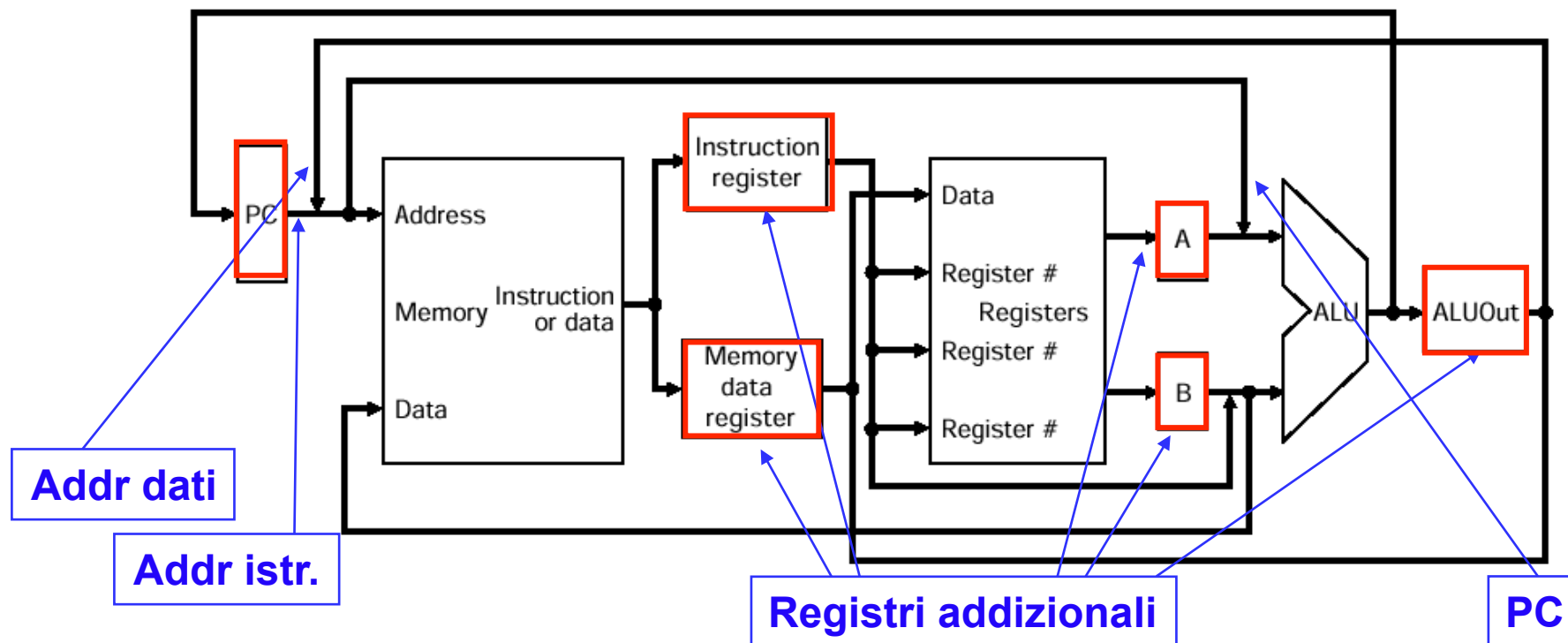
Esempio di riduzione del ciclo di clock

Effettua un *taglio* su grafo diretto aciclico corrispondente al circuito combinatorio, e inserisci un registro nel mezzo

Fai lo stesso lavoro di prima in 2 cicli più corti (più veloci), invece che in 1 singolo ciclo più lungo (più lento)



Datapath multiciclo



Registri interni addizionali usati per memorizzare valori intermedi, da usare nel ciclo di clock successivo per continuare l'esecuzione della stessa istruzione

- IR, MDR, A, B, ALUOut

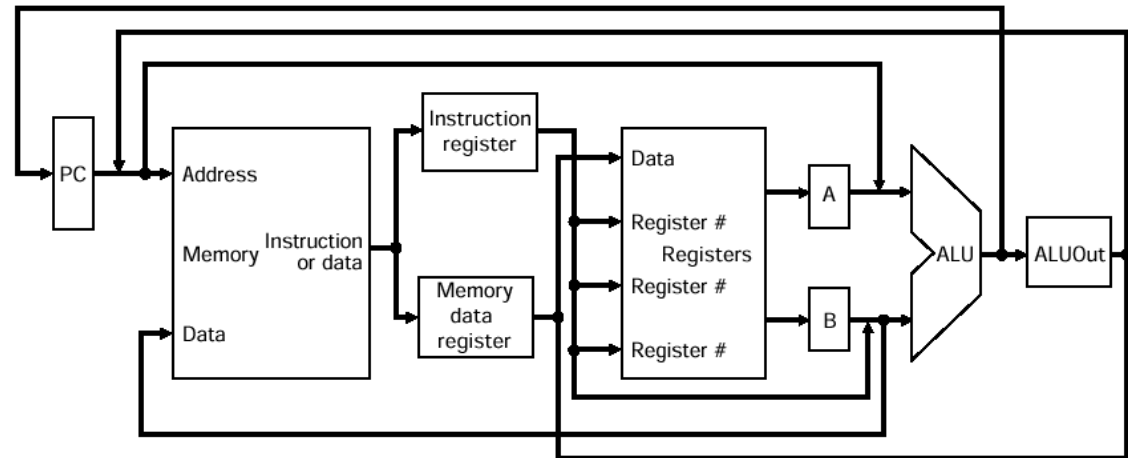
Riuso di unità funzionali

- ALU usata anche per calcolare l'indirizzo dei salti e incrementare il PC
- Memoria usata sia per leggere le istruzioni che per leggere/scrivere i dati

Architettura degli Elaboratori

Suddivisione in passi del Datapath

Nell'inserire i registri
addizionali, abbiamo
pensato di **spezzare**
l'esecuzione delle
istruzioni in passi



- ogni **passo** da eseguire in un **ciclo di clock** (*ciclo più corto rispetto alla CPU a ciclo singolo*)
- importante il bilanciamento della quantità di lavoro eseguito nei vari passi, perché dobbiamo fissare un ciclo di clock unico
 - determinato sulla base del passo più lungo, ovvero più costoso dal punto di vista computazionale

Al termine di ogni ciclo i valori *intermedi* sono memorizzati nei **registri interni addizionali**: da impiegare nei cicli successivi della stessa istruzione

Register File e **PC** sono invece impiegati per memorizzare valori da usare per l'esecuzione di una nuova istruzione

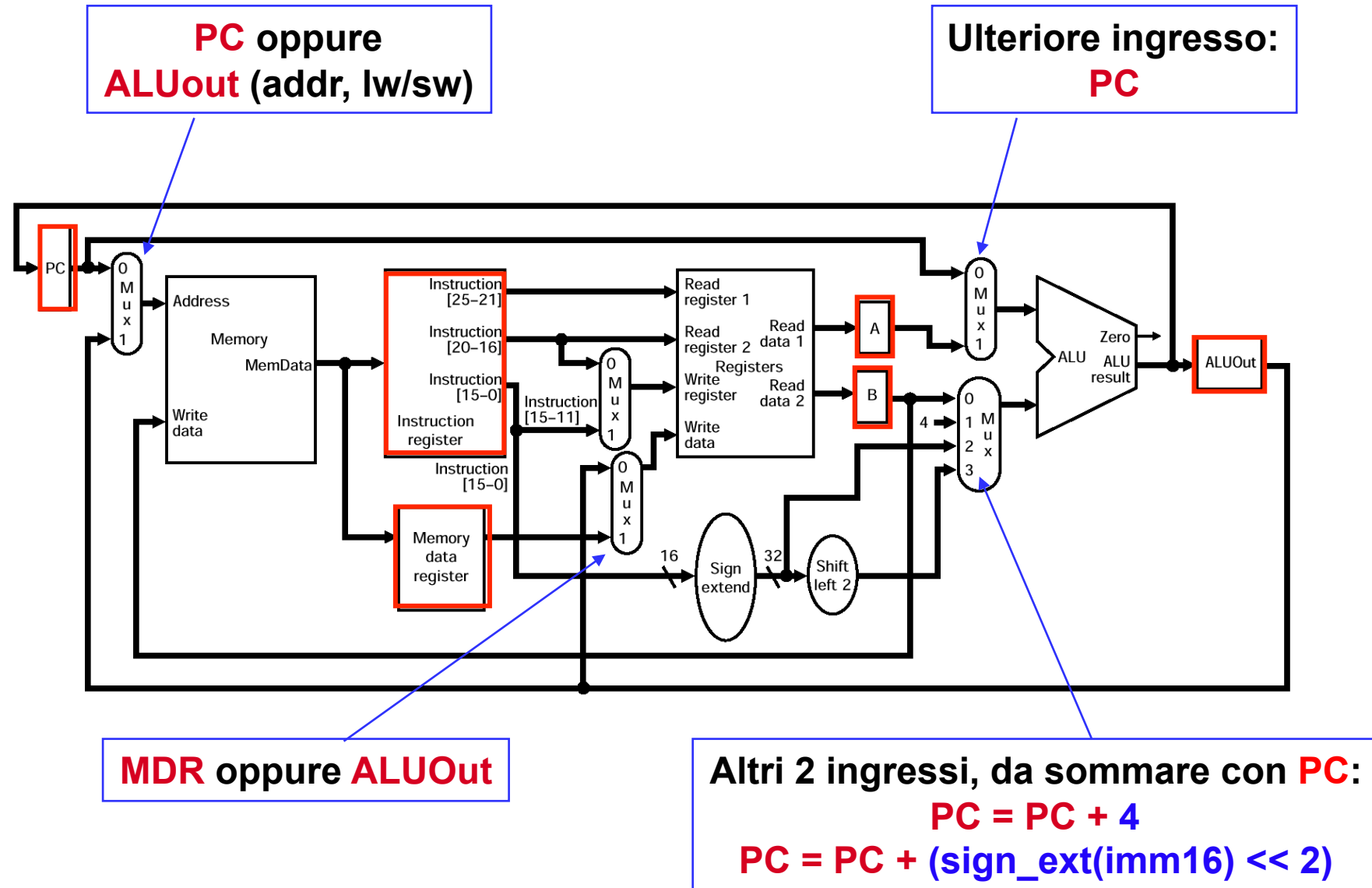
Sequenza dei cinque passi di esecuzione

1. Fetch dell'istruzione *e* Incremento PC
2. Decodifica dell'istruzione *e* Lettura dei regs. (*e* Addr. Branch)
3. R-type exe *o* Calcolo Indirizzo Memoria *o* Completa Branch *o* Completa Jump
 - dipende dal tipo di istruzione
4. Accesso alla memoria *o* Completa R-type (scrittura regs)
 - dipende dal tipo di istruzione
5. Write back (scrittura reg: solo LW)

OGNI PASSO ESEGUITO IN UN CICLO DI CLOCK

LE ISTRUZIONI IMPIEGANO DA 3 A 5 CICLI

Inseriamo i multiplexer



Controllo

I segnali di controllo alle varie unità funzionali e ai multiplexer non dipendono solo dal **tipo istruzione**, ma anche dallo specifico **passo di esecuzione**

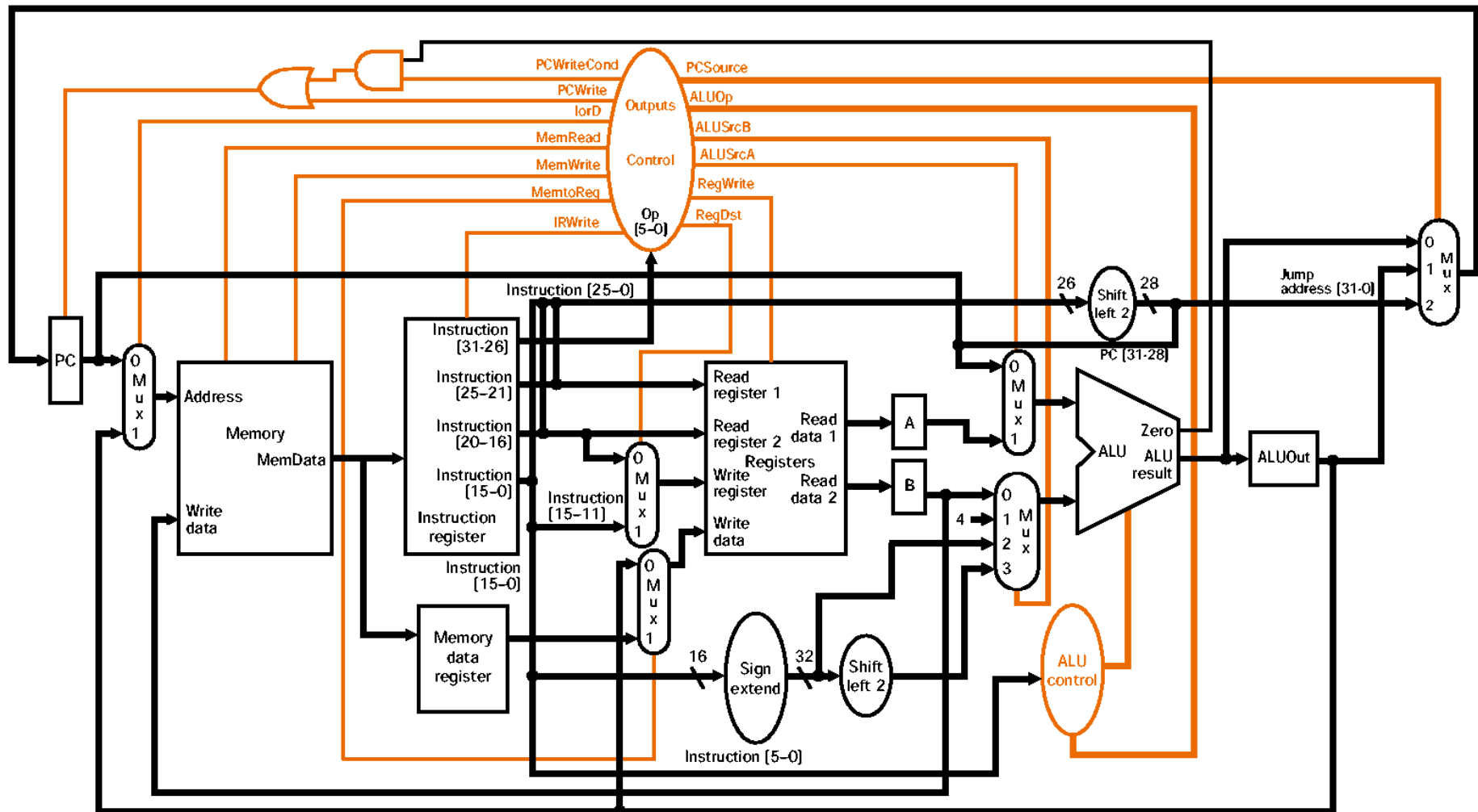
- es.: rispetto ad una **sub**, l'ALU dovrà essere usata, in cicli di clock differenti, per
 - $PC = PC + 4;$ (somma)
 - $R[rd] \leftarrow R[rs] - R[rt];$ (sottrazione)

Quindi i segnali di controllo dovranno essere diversi durante i vari passi (cicli di clock) necessari per l'esecuzione dell'istruzione

Il *controllo* sarà infatti implementato come circuito sequenziale

- l'output del circuito (segnali di controllo) dipenderà dallo **stato interno al circuito al tempo t_i**
- **stato del circuito sequenziale** = **passo di esecuzione di un'istruzione**

Datapath e Controllo multi-ciclo



Segnali di controllo di 1 bit

Segnale	Effetto se uguale a 0	Effetto se uguale a 1
RegDst	Reg. dest. proviene da <i>rt</i>	Reg. dest. proviene da <i>rd</i>
Reg Write	Nessuno	Scrittura in Reg. File
AluSrcA	1° ingresso dell'ALU è <i>PC</i>	1° ingresso dell'ALU è <i>reg. A</i>
MemRead	Nessuno	Lettura da Mem. in <i>reg. MDR</i>
MemWrite	Nessuno	Scrittura in <i>Mem.</i>
MemtoReg	Reg. scritto proviene da <i>ALUOut</i>	Reg. scritto proviene da <i>MDR</i>
lorD	Addr.della Mem. proviene da <i>PC</i>	Addr.della Mem. proviene da <i>ALUOut</i>
IRWrite	Nessuno	Scrittura in <i>IR</i> (proviene da <i>Mem.</i>)
PCWrite	Nessuno	Scrittura in <i>PC</i>
PCWriteCond	Nessuno	<i>PC</i> viene scritto se è anche vero che <i>Zero = 1</i> (beq)

Per istruzioni di **beq**: **PCWriteCond=1** e **PCWrite=0**

Il segnale di scrittura di PC è infatti calcolato come:

$$\text{PCWrite} + (\text{PCWriteCond} \cdot \text{Zero})$$

Se **Zero=0** il valore di PC che punta alla prossima istruzione rimane invariato

Segnali di controllo di 2 bit

Segnale	Valore	Effetto
ALUOp	00	ALU calcola somma (<i>lw, sw, PC + ...</i>)
	01	ALU calcola sottrazione (<i>beq</i>)
	10	ALU calcola l'operazione determinata da funct
ALUSrcB	00	2 ^a ingresso dell'ALU è reg. B
	01	2 ^a ingresso dell'ALU è costante 4
	10	2 ^a ingresso dell'ALU è sign_ext(imm16) (<i>lw/sw</i>)
	11	2 ^a ingresso dell'ALU è sign_ext(imm16) << 2 (<i>beq</i>)
PCSource	00	In PC viene scritto l'uscita dell'ALU (PC+4)
	01	In PC viene scritto ALUOut (<i>beq</i>)
	10	In PC viene scritto PC[31-28] sign_ext(imm26) << 2 (<i>jump</i>)

Passo 1: Fetch dell'istruzione

Usa PC per prelevare l'istruzione dalla memoria e porla nell'Instruction Register (IR)

Incrementa PC di 4, e rimetti il risultato nel PC

Passo identico per tutte le istruzioni

Usando la notazione RTL:

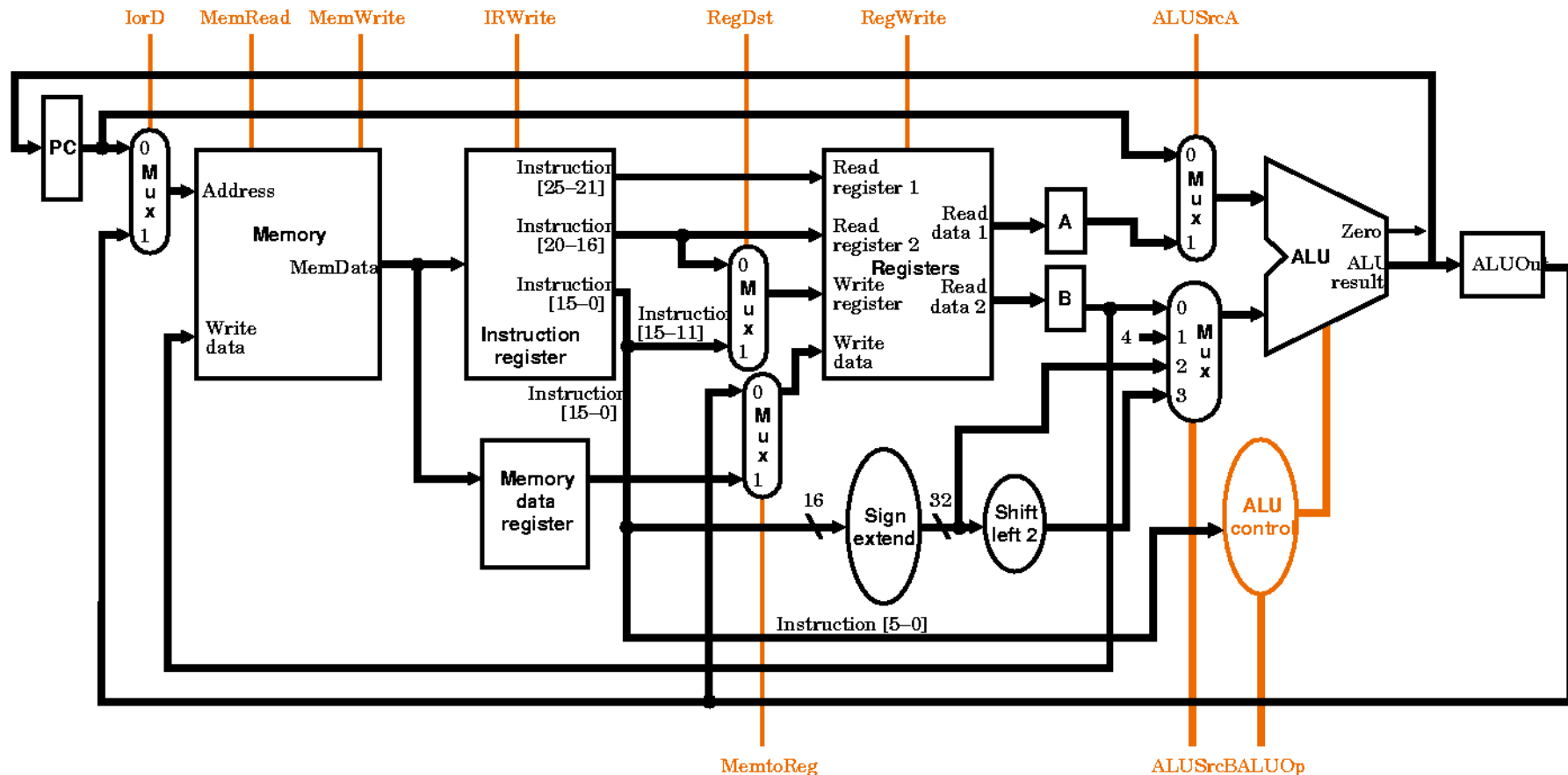
$IR = M[PC];$
 $PC = PC + 4;$

Durante questo passo (stesso ciclo di clock) usiamo:

- Memoria
- ALU

Vediamo in dettaglio i valori dei segnali di controllo

Passo 1: Fetch dell'istruzione


$$IR = M[PC];$$

- MemRead $\leftarrow 1$
- IRWrite $\leftarrow 1$
- lorD $\leftarrow 0$

$$PC = PC + 4;$$

- ALUSrcA \leftarrow 0
- ALUSrcB \leftarrow 01
- ALUOp \leftarrow 00 (somma)

- PCWrite $\leftarrow 1$
- PCSource $\leftarrow 00$

Passo 2: Decodifica istruzione & Lettura registri

Leggi i registri **rs** e **rt**, e calcola l'indirizzo del salto di **beq**

IR (**op**) viene inviato al **controllo** per la decodifica e la determinazione dei passi successivi \Rightarrow **Decodifica dell'istruzione**

RTL:

```
A = Reg[ IR[25-21] ];  
B = Reg[ IR[20-16] ];  
ALUOut = PC + (sign-ext( IR[15-0] ) << 2);
```

Passo identico per tutte le istruzioni , ma potremmo anticipare del lavoro non necessario.

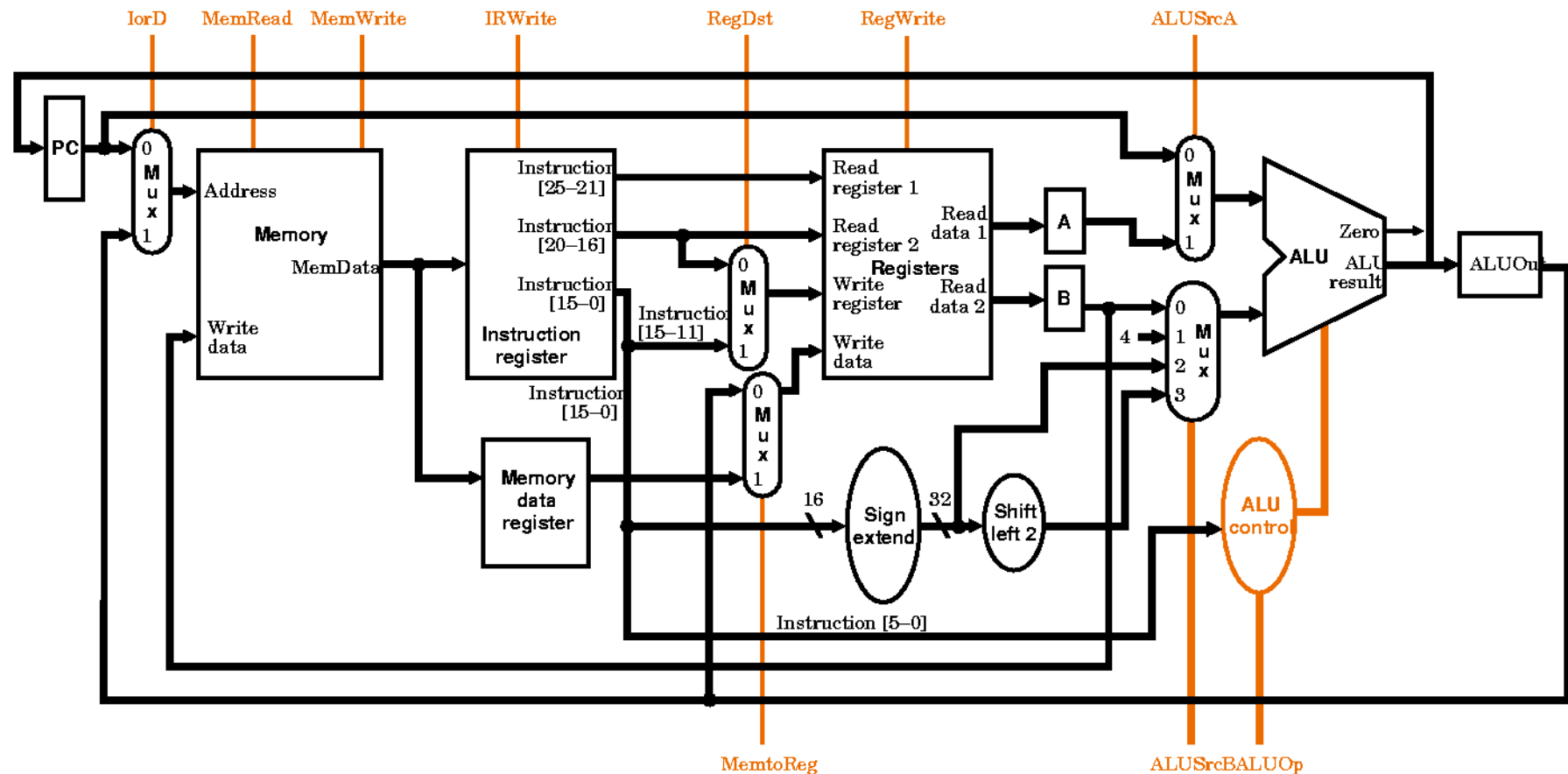
- per certe istruzioni, i due campi (**rs**, **rt**) potrebbero essere non significativi
- calcoliamo l'indirizzo a cui saltare, come se l'istruzione fosse **beq**, ma il campo **imm16** potrebbe essere non significativo

Quali i vantaggi di questo lavoro anticipato?

Durante questo passo (durante lo stesso ciclo di clock) usiamo: Register File e ALU

Vediamo i valori dei segnali di controllo...

Passo 2: Decodifica istruzione & Lettura registri



$A = \text{Reg}[\text{IR}[25-21]];$

$B = \text{Reg}[\text{IR}[20-16]];$

- A e B sovrascritti per ogni tipo di istruzione

$\text{ALUOut} = \text{PC} + (\text{sign-ext}(\text{IR}[15-0]) \ll 2)$

- $\text{ALUSrcA} \leftarrow 0$
- $\text{ALUSrcB} \leftarrow 11$
- $\text{ALUOp} \leftarrow 00$ (somma)

Passo 3: (dipende dall'istruzione)

Usiamo l'ALU in dipendenza del tipo di istruzione

Il **controllo**, avendo già decodificato l'istruzione letta al passo precedente, può già decidere i segnali da inviare al Datapath in relazione al tipo di istruzione

R-type exe:

$$ALUOut = A \text{ op } B;$$

Calcolo Indirizzo Memoria (load/store)

$$ALUOut = A + \text{sign-ext}(IR[15-0]);$$

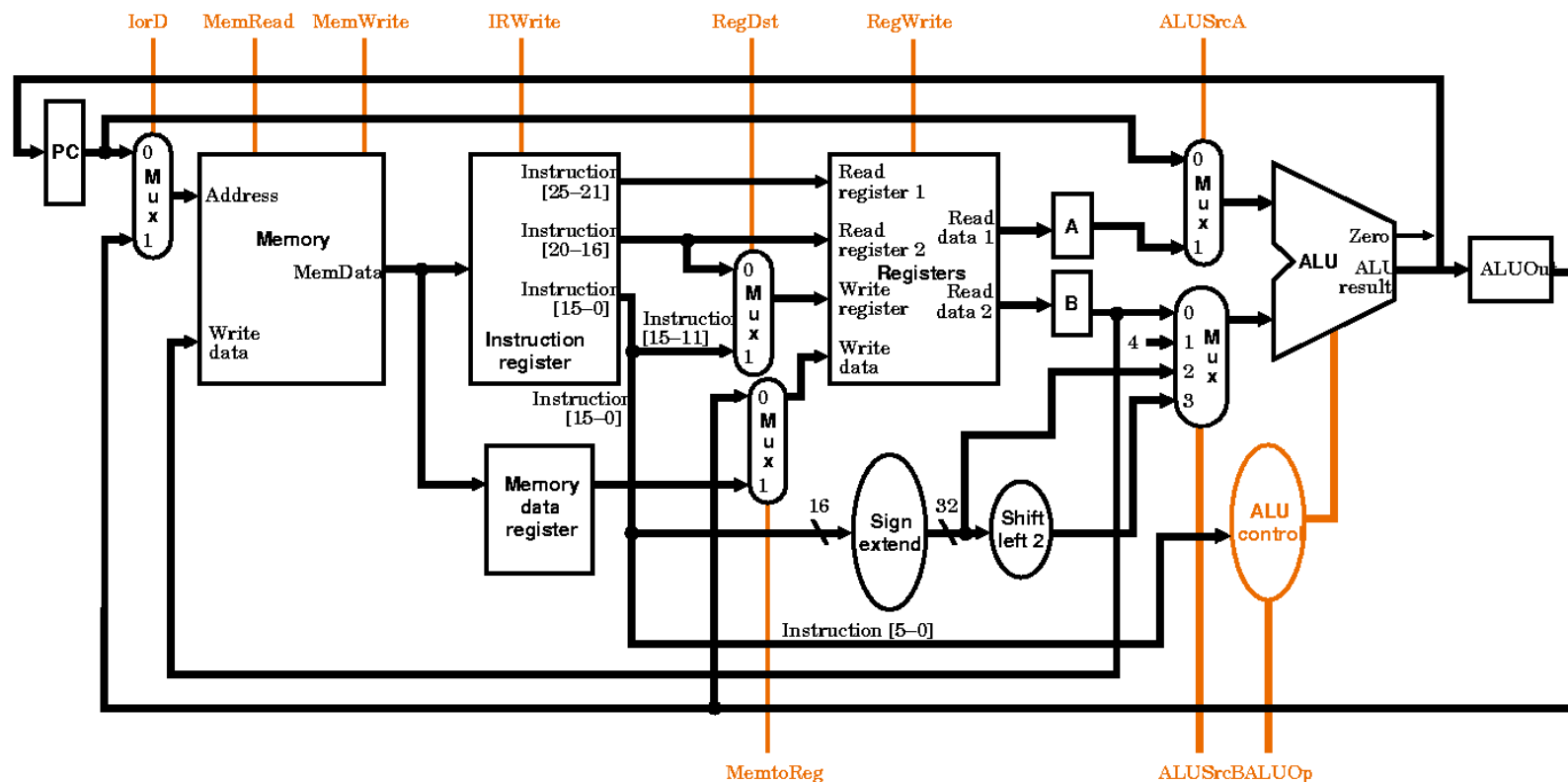
Completa Branch

$$\text{if } (A == B) \text{ then } PC = ALUOut;$$

Completa Jump

$$PC = PC[31-28] \parallel (IR[25-0] \ll 2);$$

Passo 3: (dipende dall'istruzione)



R-type exe:

$ALUOut = A \text{ op } B$; $ALUOut$ riscritto ad ogni ciclo

$ALUSrcA \leftarrow 1$

$ALUSrcB \leftarrow 00$

$ALUOp \leftarrow 10$ (campo FUNCT)

LOAD / STORE:

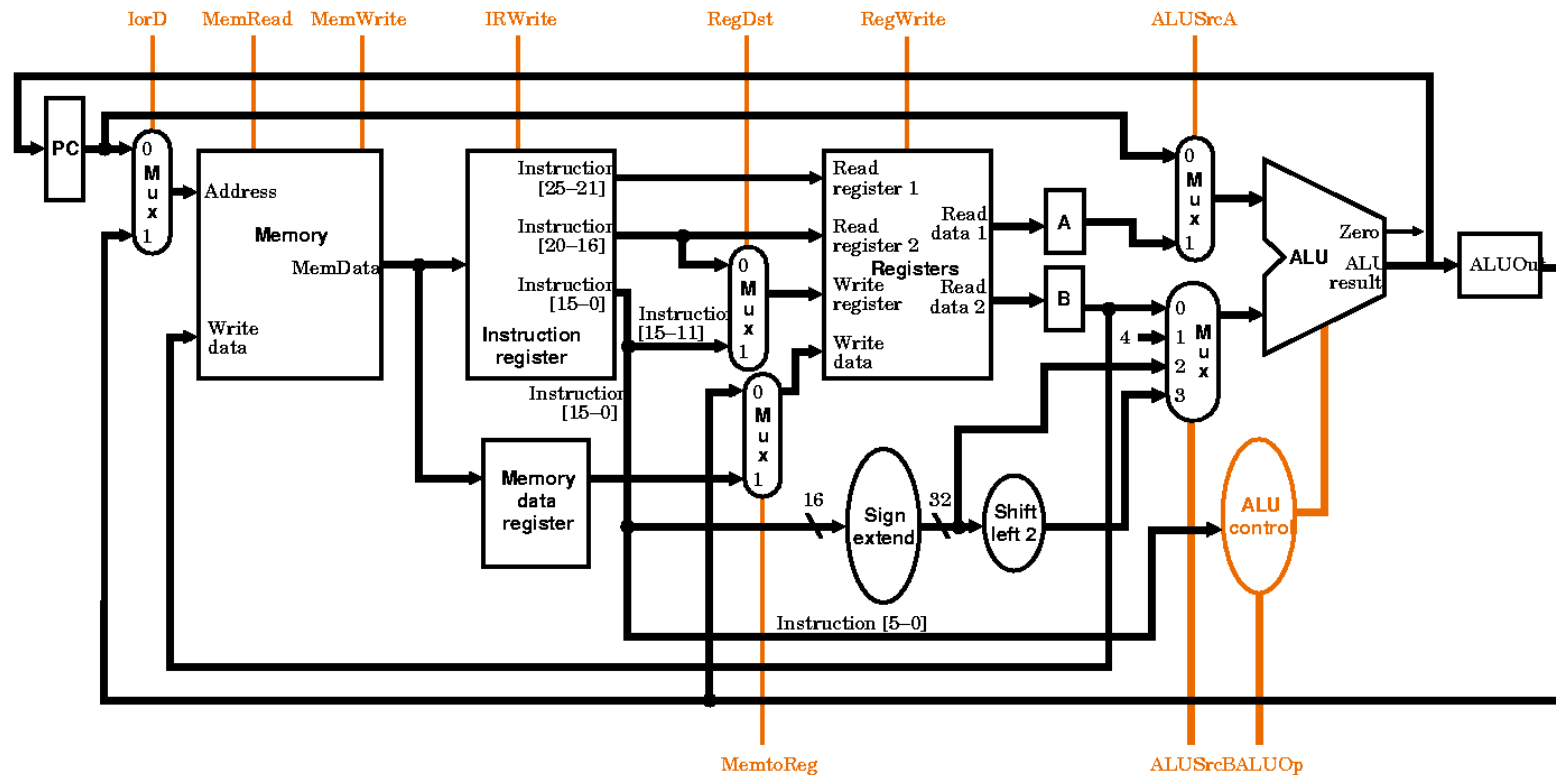
$ALUOut = A + \text{sign-ext}(IR[15-0])$;

$ALUSrcA \leftarrow 1$

$ALUSrcB \leftarrow 10$

$ALUOp \leftarrow 00$ (somma)

Passo 3: (dipende dall'istruzione)



Completa Branch: if (A == B) then PC = ALUOut;

$ALUSrcA \leftarrow 1$ $ALUSrcB \leftarrow 00$ $ALUOp \leftarrow 01$ (sottr.)

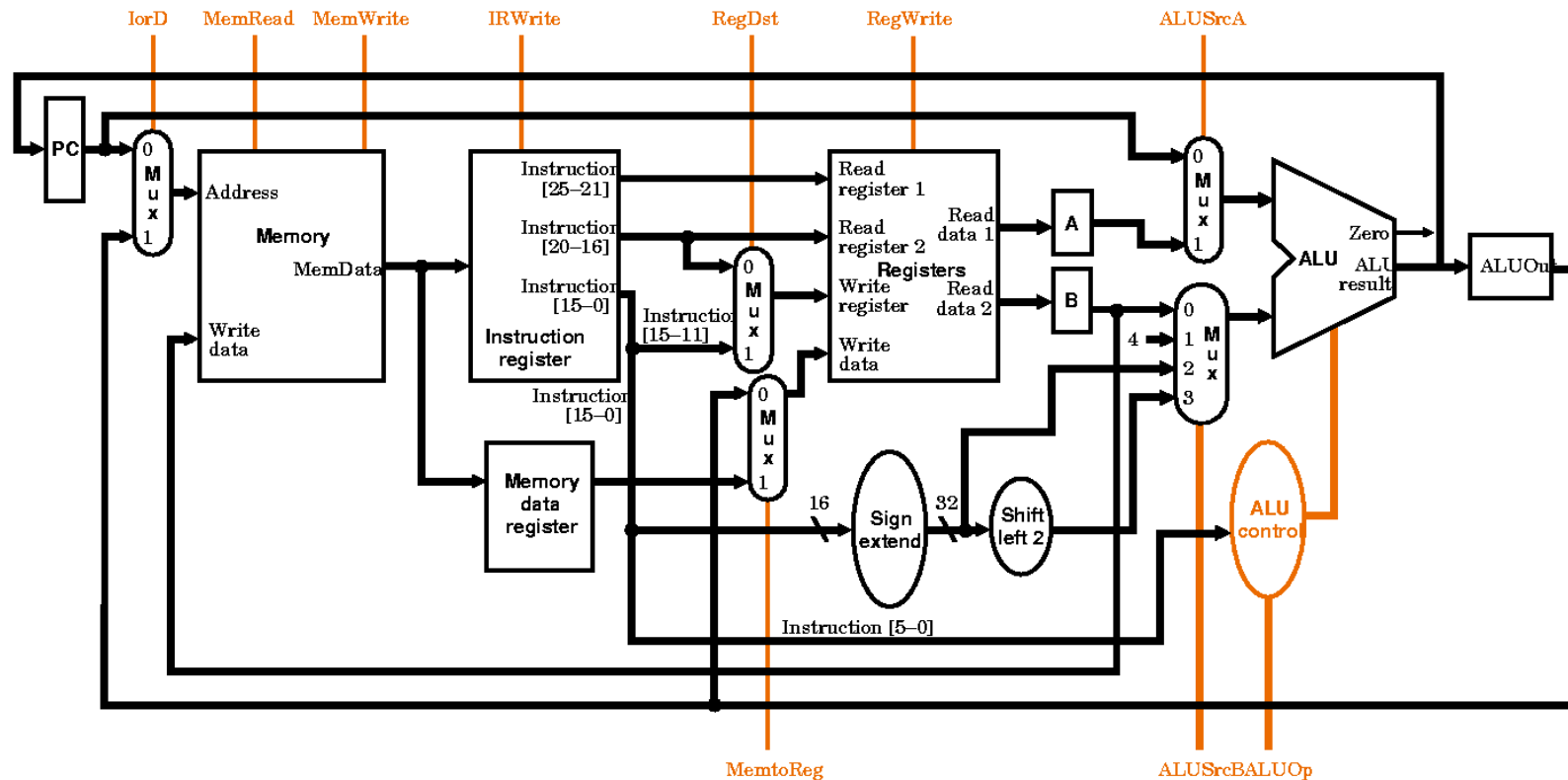
(sottrazione registri A e B, check salto sulla base di *Zero*)

Per abilitare la scrittura nel PC del valore precedentemente calcolato (ALUOut), necessari altri segnali di controllo non illustrati:

$PCWrite \leftarrow 0$ $PCWriteCond \leftarrow 1$ $PCSource \leftarrow 01$

Architettura degli Elaboratori

Passo 3: (dipende dall'istruzione)



Completa Jump: $PC = PC[31-28] \parallel (IR[25-0] \ll 2);$

Per abilitare la scrittura nel PC, i segnali di controllo, non illustrati in figura, sono:
 $PCWrite \leftarrow 1$ $PCSource \leftarrow 10$

Passo 4: (dipende dall'istruzione)

LOAD e STORE accedono alla memoria

MDR = Memory[ALUOut];
or
Memory[ALUOut] = B;

Terminazione istruzioni R-type

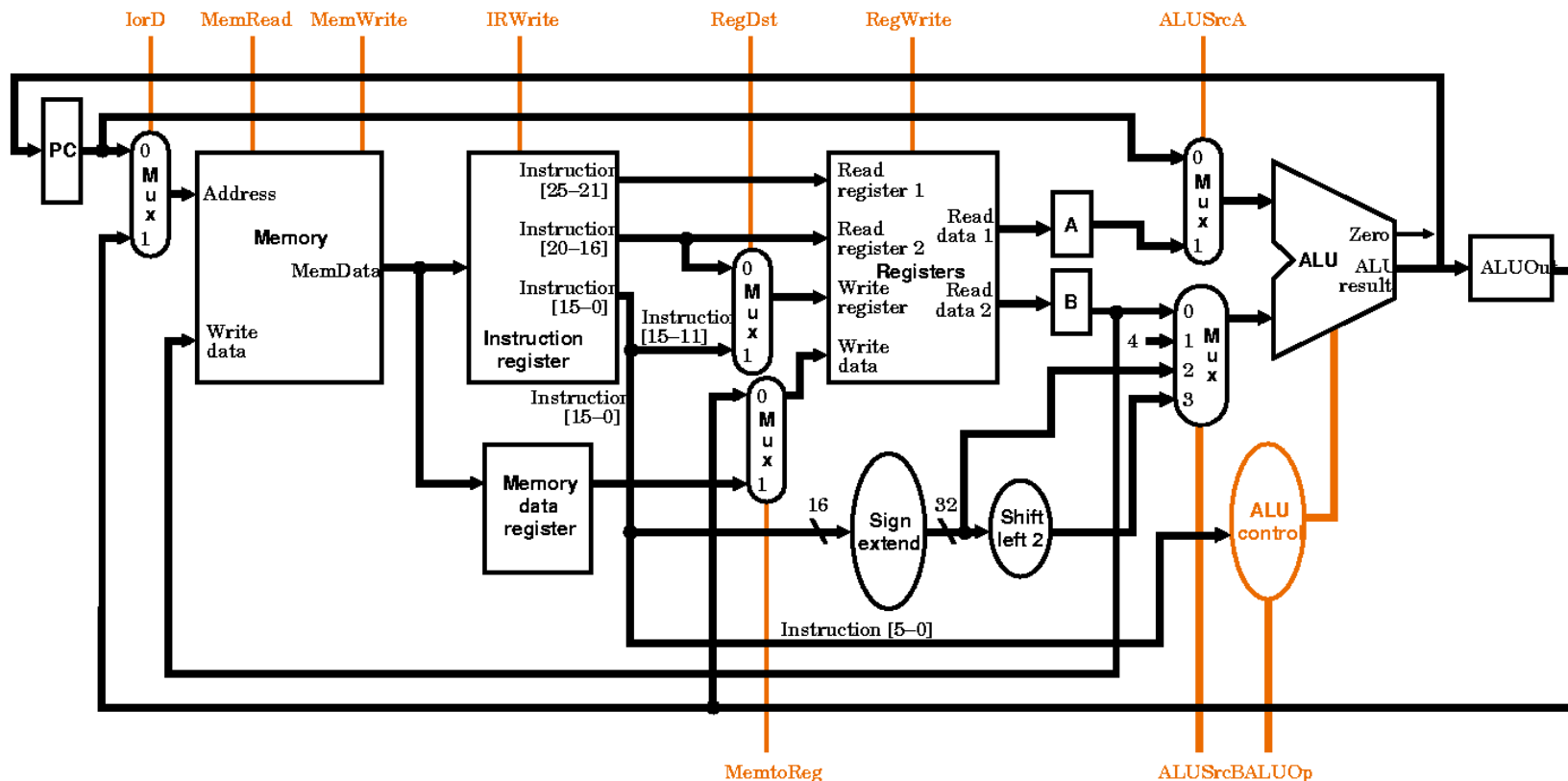
Reg[IR[15-11]] = ALUOut;

Durante questo passo usiamo:

- Register File (Write) *oppure* Memoria

Vediamo i segnali di controllo

Passo 4: (dipende dall'istruzione)



Load:

$MDR = Memory[ALUOut];$

- $lorD \leftarrow 1$
- $MemRead \leftarrow 1$

Store:

$Memory[ALUOut] = B;$

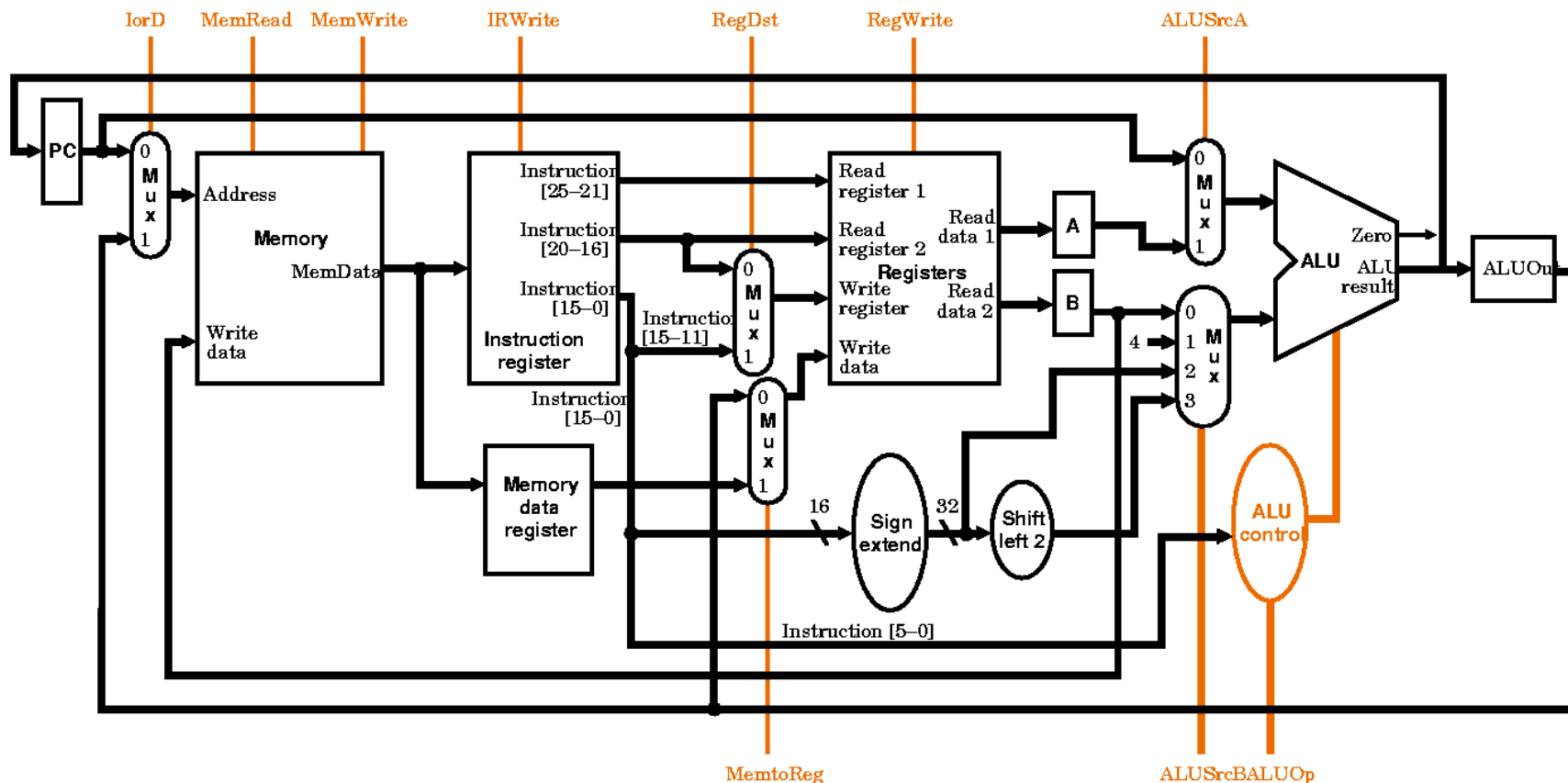
- $lorD \leftarrow 1$
- $MemWrite \leftarrow 1$

R-type:

$Reg[IR[15-11]] = ALUOut;$

- $RegDst \leftarrow 1$
- $RegWrite \leftarrow 1$
- $MemtoReg \leftarrow 0$

Passo 5: Write-back (LOAD)



Load:

$\text{Reg[IR[20-16]]} = \text{MDR};$

- $\text{RegDst} \leftarrow 0$
- $\text{RegWrite} \leftarrow 1$
- $\text{MemtoReg} \leftarrow 1$

NOTA: Le altre istruzioni non giungono al passo 5

Riassumendo

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	$R = \text{Memory}[PC]$ $PC = PC + 4$			
Instruction decode / register fetch/ branch addr. comp.	$A = \text{Reg}[\text{IR}[25-21]]$ $B = \text{Reg}[\text{IR}[20-16]]$ $\text{ALUOut} = PC + (\text{sign-extend}(\text{IR}[15-0]) \ll 2)$			
Execution, address computation, branch/ jump completion	$\text{ALUOut} = A \text{ op } B$	$\text{ALUOut} = A + \text{sign-extend}(\text{IR}[15-0])$	if $(A == B)$ then $PC = \text{ALUOut}$	$PC = PC[31-28] \parallel (\text{IR}[25-0] \ll 2)$
Memory access or R-type completion	$\text{Reg}[\text{IR}[15-11]] = \text{ALUOut}$	Load: $\text{MDR} = \text{Memory}[\text{ALUOut}]$ or Store: $\text{Memory}[\text{ALUOut}] = B$		
Memory read completion		Load: $\text{Reg}[\text{IR}[20-16]] = \text{MDR}$		

Alcune semplici domande

Quanti cicli sono necessari per eseguire questo codice?

<code>lw \$t2, 0(\$t3)</code>	5
<code>lw \$t3, 4(\$t3)</code>	5
<code>beq \$t2, \$t3, Label #assume not</code>	3
<code>add \$t5, \$t2, \$t3</code>	4
<code>sw \$t5, 8(\$t3)</code>	<u>4</u>
<code>Label: ...</code>	21

Cosa accade durante l'8° ciclo di esecuzione?

- **Calcolo dell'indirizzo della 2ª lw**

In quale ciclo avviene effettivamente la somma tra `$t2` e `$t3` ?

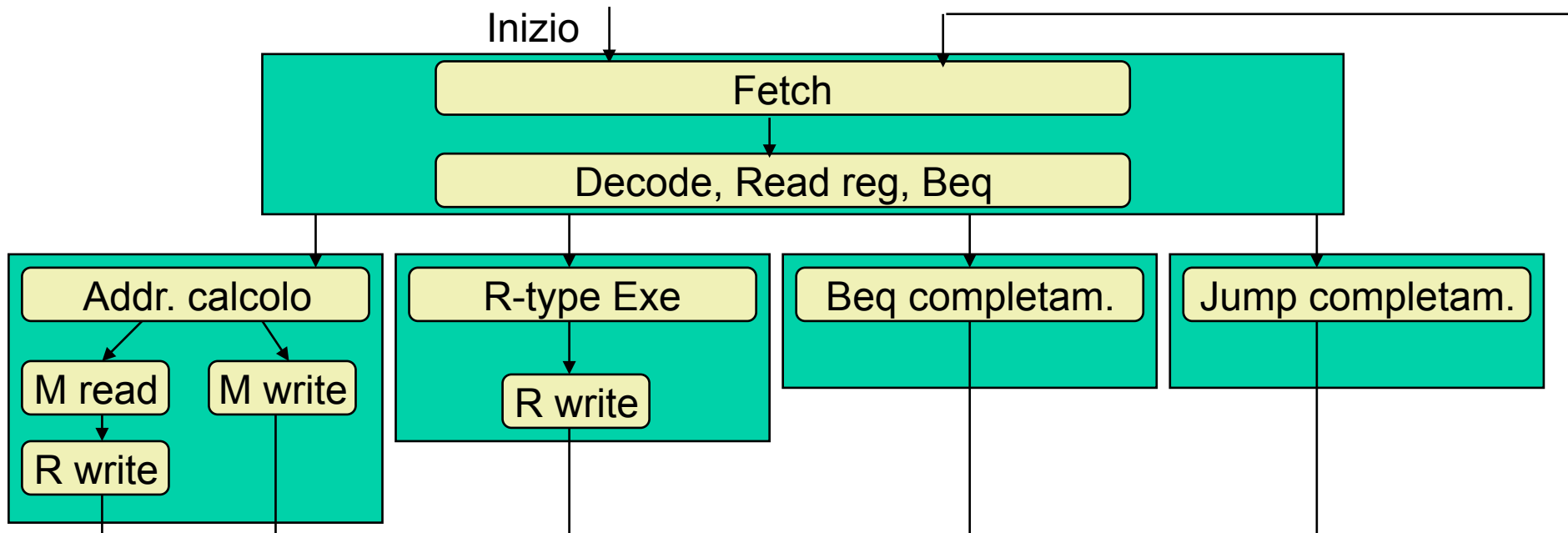
- **Nel 16-esimo ciclo**

Definizione del controllo

Possiamo implementare il **controllo** della CPU come un **circuito sequenziale di Moore**, modellato con un automa a stati finiti

Automa

- ogni nodo corrisponde ad uno stato differente del circuito, in corrispondenza di un certo ciclo di clock
- gli output del **controllo** (segnali di controllo) dipendono dallo stato corrente
- da 3 a 5 stati devono essere attraversati (ovvero, da 3 a 5 cicli di clock)



Automa completo

Etichette interne ai nodi

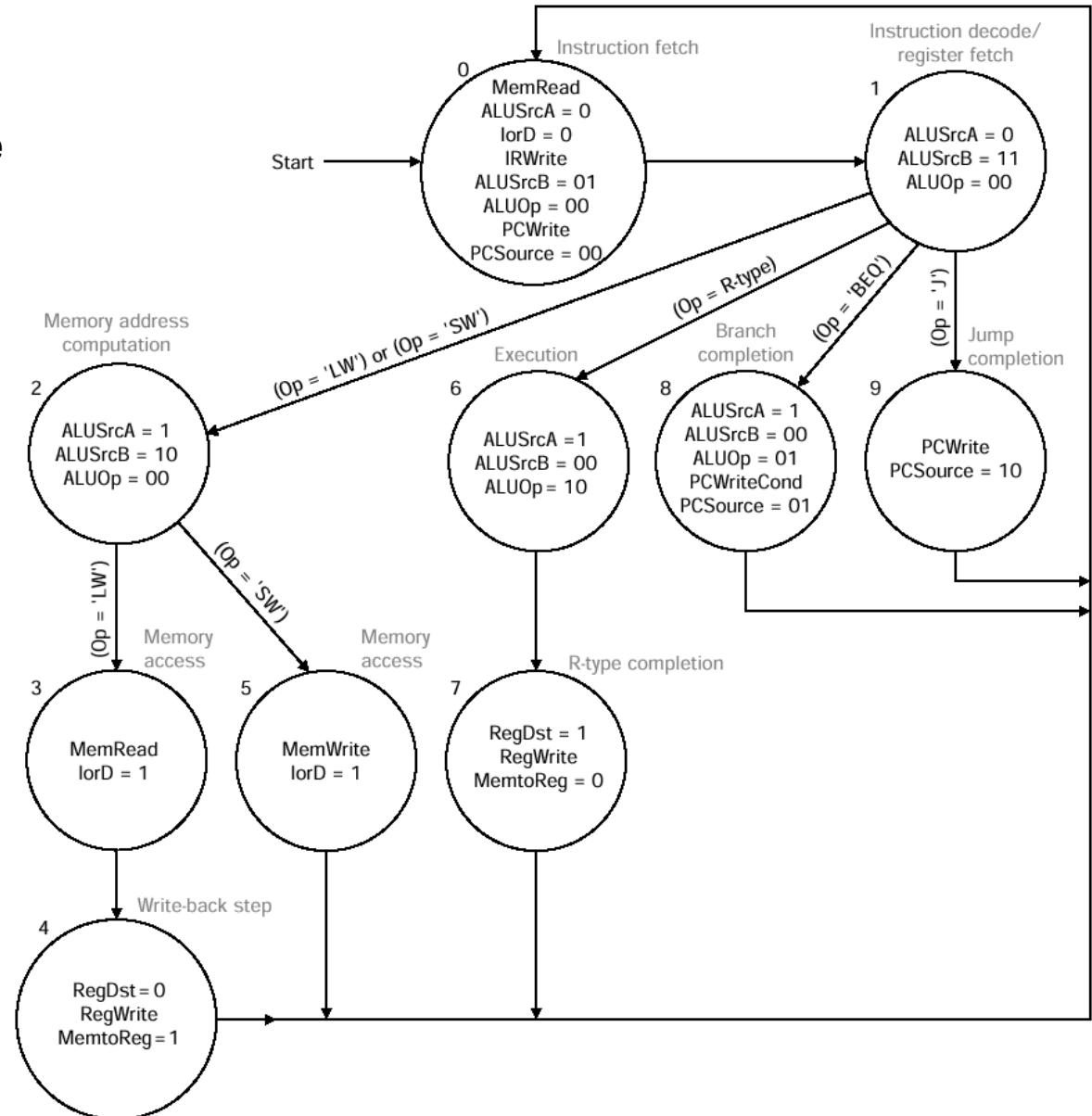
- corrispondono ai segnali che il Controllo deve inviare al Datapath

Etichette sugli archi

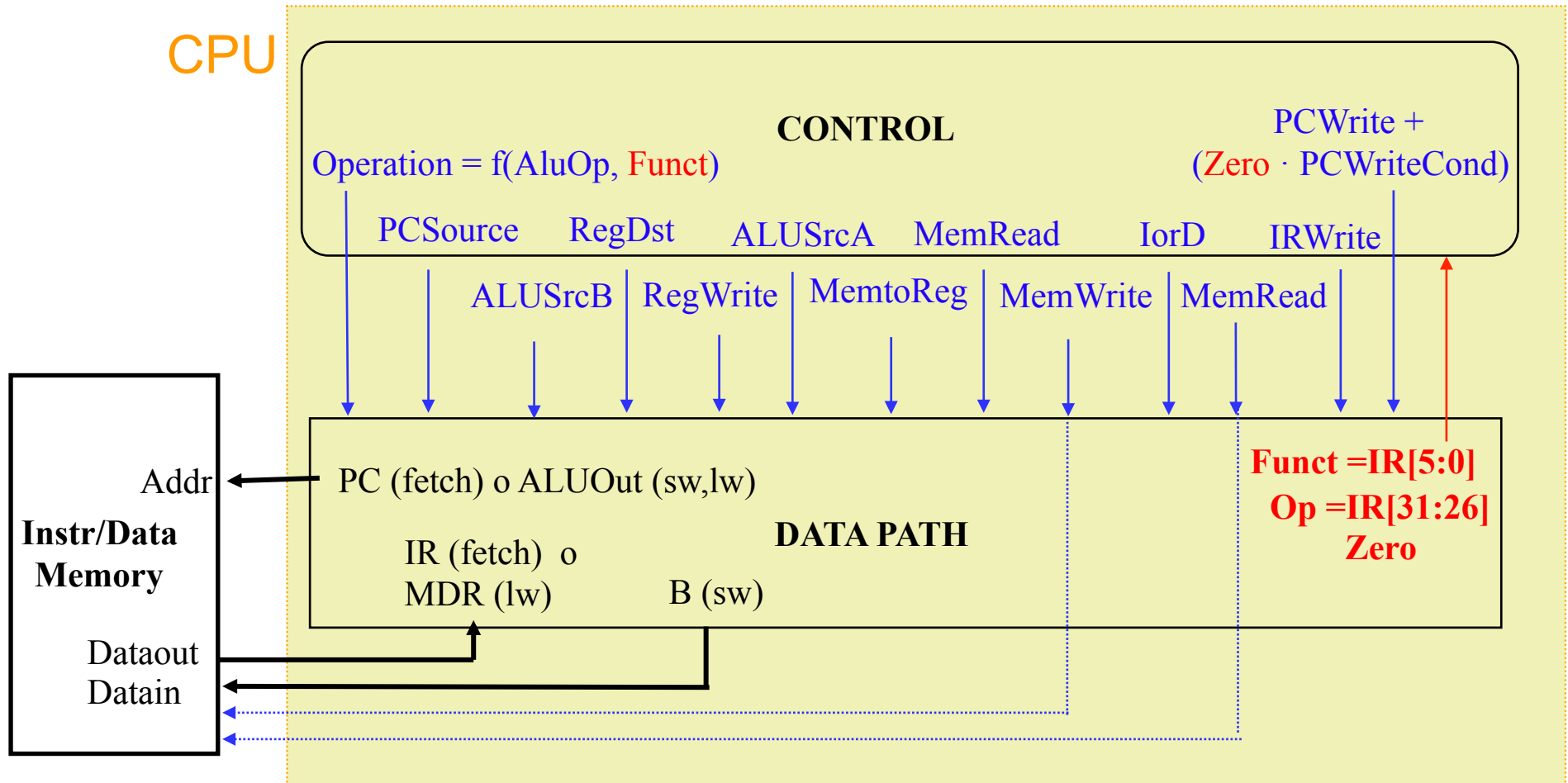
- dipendono dagli input del Controllo
- ovvero, dal valore di del campo **Op** dell'istruzione letta

10 stati

- ogni stato associato con un'etichetta mnemonica, e anche con un identificatore numerico
- quanti bit sono necessari per il **registro di stato**?



Componenti CPU (Datapath+Control) e Memoria



Nota: tra i segnali provenienti dal Datapath, solo **Op** è usato per selezionare il prossimo stato

Dimensionamento ciclo di clock

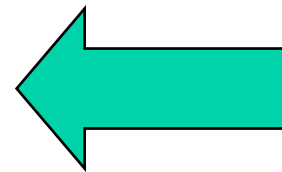
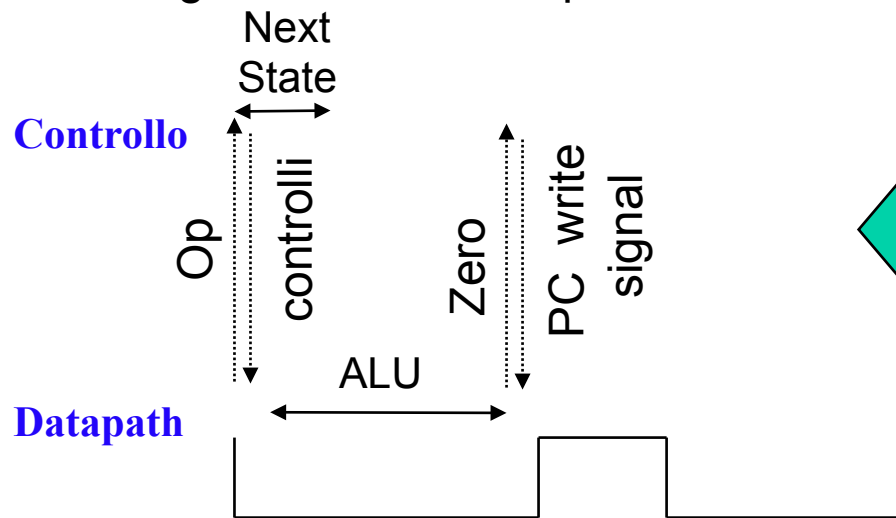
Ipotizziamo gli stessi costi precedenti (in ns) per le varie componenti

Mem. Istr/Dati: 2 ns Reg. File: 1 ns ALU: 2 ns

- non li usiamo mai in sequenza \Rightarrow possiamo ipotizzare un ciclo di 2 ns

Più in dettaglio, per determinare il ciclo di clock, consideriamo il diagramma di sotto, che si riferisce al 3° passo della BEQ

- poiché il controllo è di Moore, l'output (controlli) dipende solo dallo stato (veloce)
 - decodifica del controllo dell'ALU più complessa (2 livelli):
$$\text{Operation} = f(\text{AluOp}, \text{Funct})$$
- l'input del controllo, importante per la transizione di stato, è Op
 - Op è un campo del registro IR del Datapath (non è necessario calcolarlo)
- il segnale di Zero è importante nel caso di BEQ ...



Esempio di diagramma temporale per il 3° passo di esecuzione di un'istruzione

Costo istruzioni

Per le varie istruzioni, possiamo impiegare un numero differente di cicli

- introduciamo il concetto di **CPI** (Cicli Per Istruzione)

Quant'è il CPI delle varie istruzioni rispetto all'architettura multi-ciclo ?

- R-type, sw: 4 cicli (tempo: 8 ns)
- lw: 5 cicli (tempo: 10 ns)
- beq, jump: 3 cicli (tempo: 6 ns)

L'istruzione **lw** impiega ben 10 ns invece degli 8 ns dell'architettura a singolo ciclo

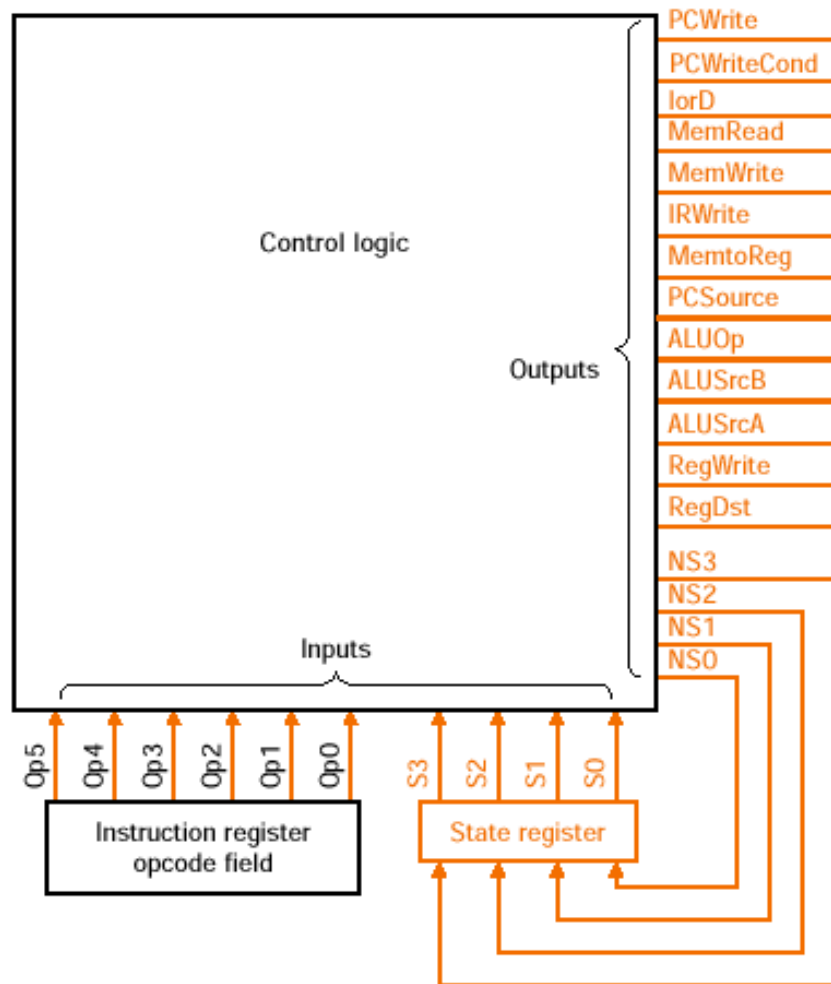
- purtroppo ciò è dovuto alla necessità di fissare il ciclo di clock abbastanza lungo da permettere l'esecuzione di uno qualsiasi dei passi previsti per le varie istruzioni
- il 5° passo della lw, anche se usa solo il Register File (latenza 1 ns), viene comunque eseguito in un ciclo di clock da 2 ns

Costo istruzioni

Abbiamo ottenuto un risparmio solo per le istruzioni di `beq` e `jump`

- se avessimo considerato istruzioni molto più lunghe (come quelle FP), non avremmo osservato questo *apparente* decadimento di prestazione nel passare all'architettura multi-ciclo
- in quel caso, la scelta del ciclo singolo ci avrebbe costretto ad allungare a dismisura il ciclo di clock per eseguire le istruzioni FP

Circuito sequenziale che implementa il controllo



Controllo a due livelli

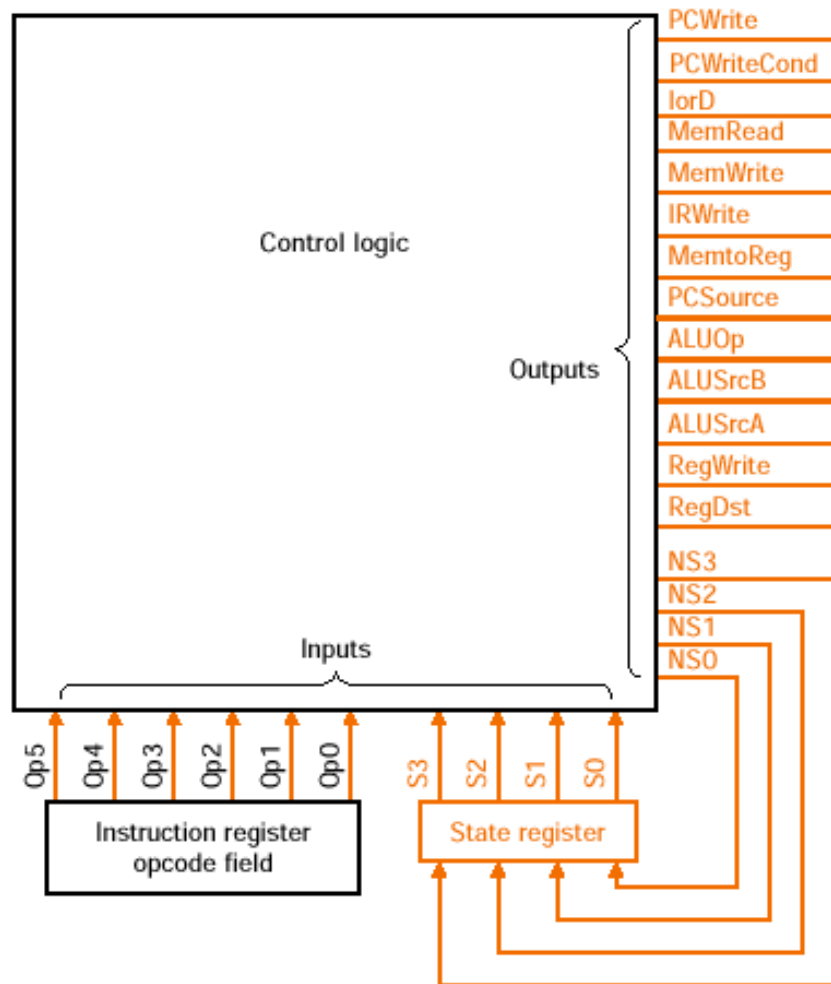
ALUOp calcolato sulla base di **Op**, combinato con **Func** (IR [5:0]) per generare il segnale a 3 bit (**Operation**) da inviare all'ALU

PCWrite e **PCWriteCond** usati assieme a **Zero** proveniente dal Datapath, per generare il segnale a 1 bit che permette la scrittura di PC

Nota

- *blocco combinatorio* per calcolare NEXT_STATE & OUTPUT
- *state register* per memorizzare lo stato corrente

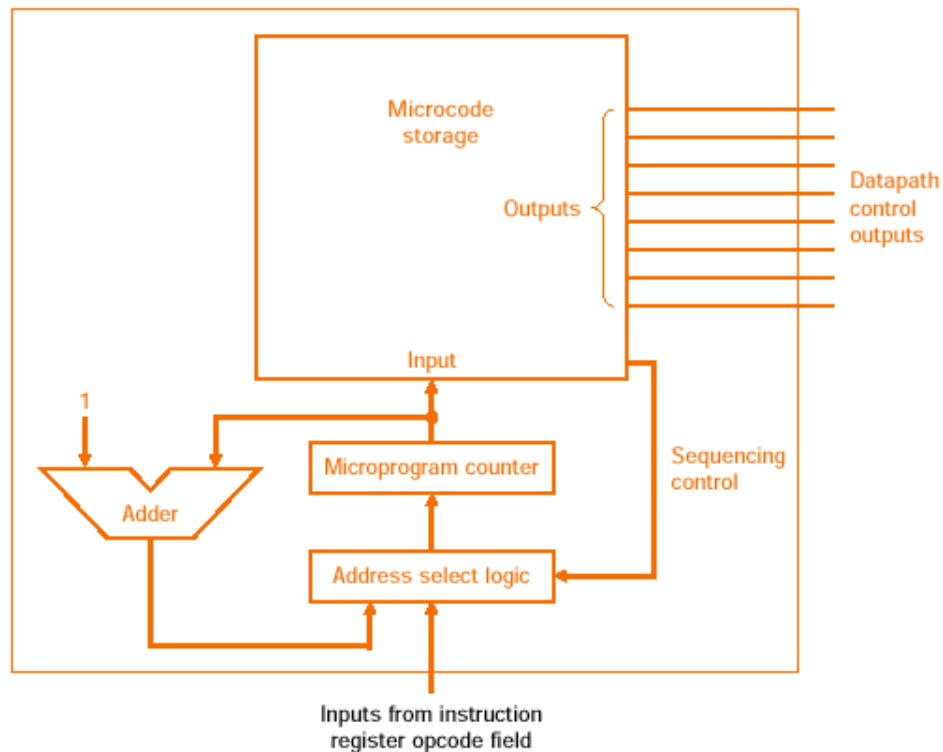
Realizzazione del blocco combinatorio



Blocco combinatorio realizzato con:

- PLA (buona minimizzazione ma non modificabile)
- ROM (tutte e due le tabelle di verità di OUTPUT e NEXT-STATE memorizzate in ROM → niente minimizzazione, ma si può modificare il circuito cambiando la ROM)

Automa rappresentato da un microprogramma



Soluzione alternativa per realizzare l'intero circuito sequenziale: usare un **microprogramma** per rappresentare l'automa a stati finiti → soluzione più flessibile

Storicamente, le implementazioni del controllo microprogrammato impiegano

- ROM per memorizzare microistruzioni
- Incrementatore esplicito e logica di sequenzializzazione per determinare la prossima microistruzione da eseguire