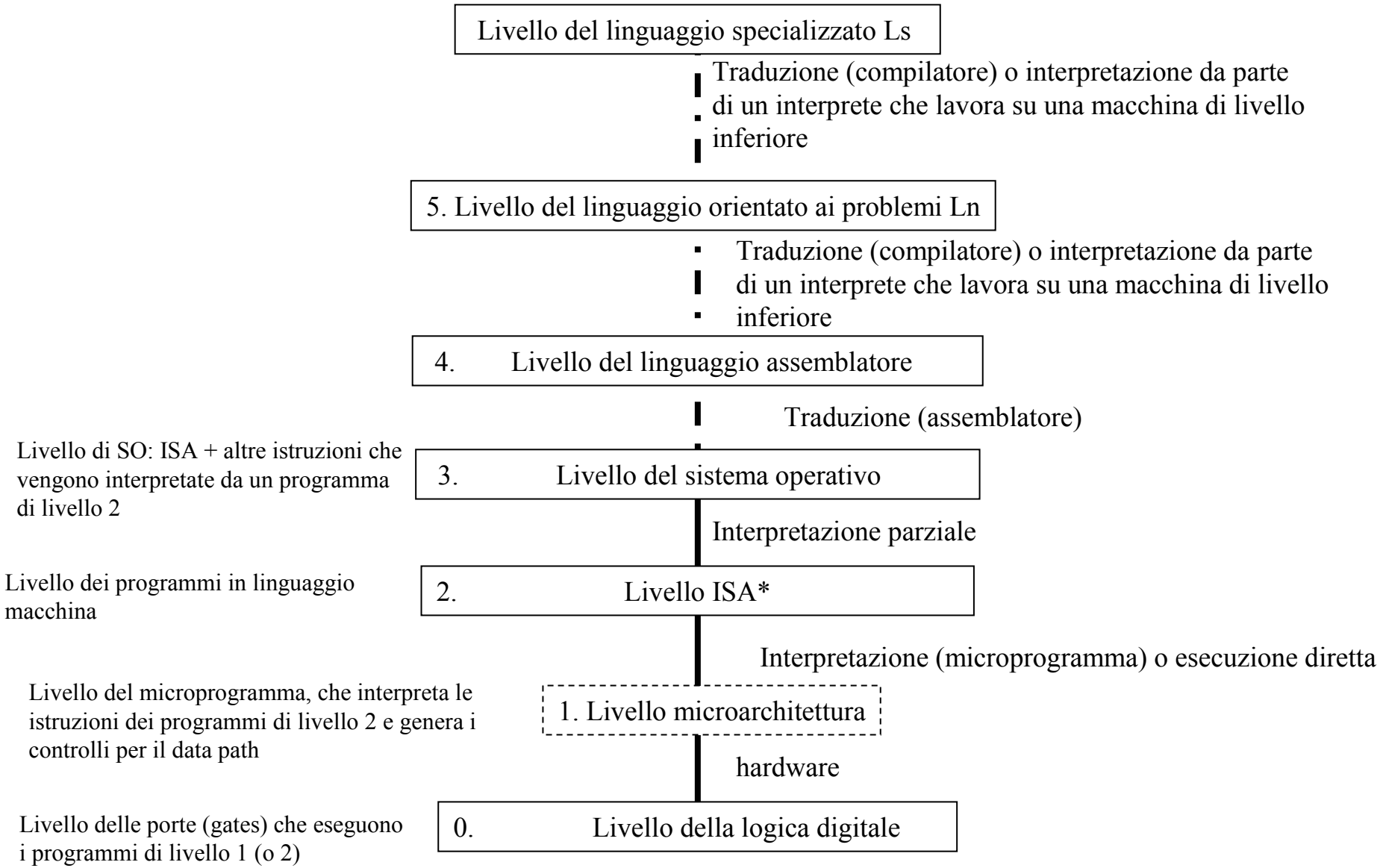
Calcolatori Elettronici B
a.a. 2005/2006

Tecniche di Controllo

Massimiliano Giacomin

Architetture

descrivono il calcolatore a diversi livelli di astrazione



*ISA: Instruction Set Architecture

Il calcolatore è progettato come una sequenza di macchine a diversi livelli, ognuna costruita sulle macchine definite ai livelli precedenti.
Ogni livello rappresenta una distinta astrazione, caratterizzata da differenti oggetti e operazioni.
I tipi di dati, le operazioni e le caratteristiche di ogni livello sono chiamate **architettura.**

- Dal livello 5 i linguaggi in cui esprimere i programmi sono chiamati linguaggi di alto livello: forniscono dati e operazioni per descrivere soluzioni di problemi in termini comprensibili per persone esperte in un certo campo.
- I programmi del livello 4 sono in forma simbolica; vengono generalmente tradotti (o a volte interpretati) da altri programmi.
- I programmi ai livelli 1, 2 e 3 sono sempre interpretati; la forma interpretata dalla macchina è sempre numerica.

I confini tra hardware e software sono sfumati:

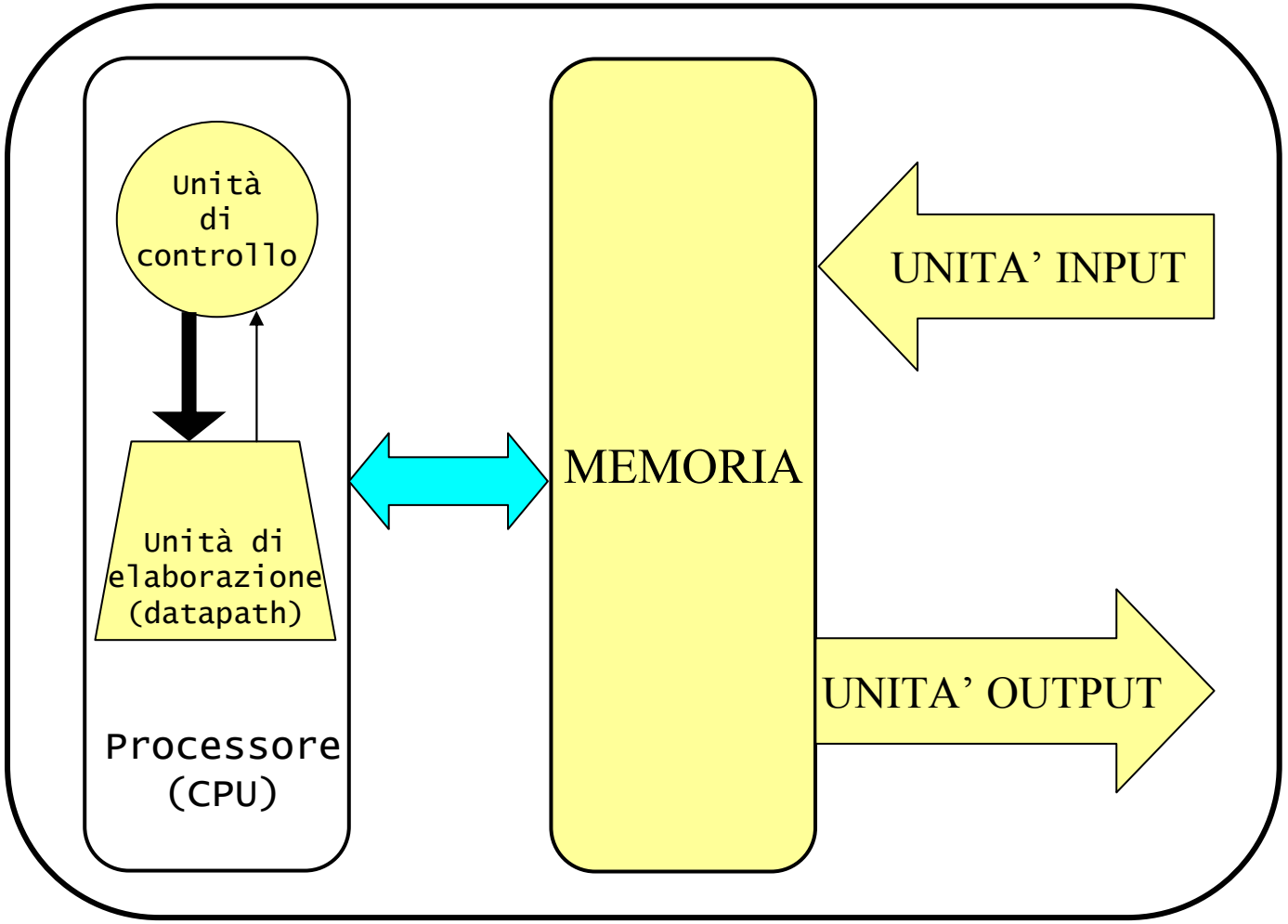
Hardware and and software are logically equivalent (Tanenbaum)
Hardware is just petrified software (K: Panetta Lenz)

ma “dovendo scegliere”:

> E' vero che il software non potrebbe esercitare i poteri della sua leggerezza se non mediante la pesantezza dell’hardware; ma è il software che comanda, che agisce sul mondo esterno e sulle macchine, le quali esistono solo in funzione del software, si evolvono in modo d'elaborare programmi sempre più complessi. (Italo Calvino)

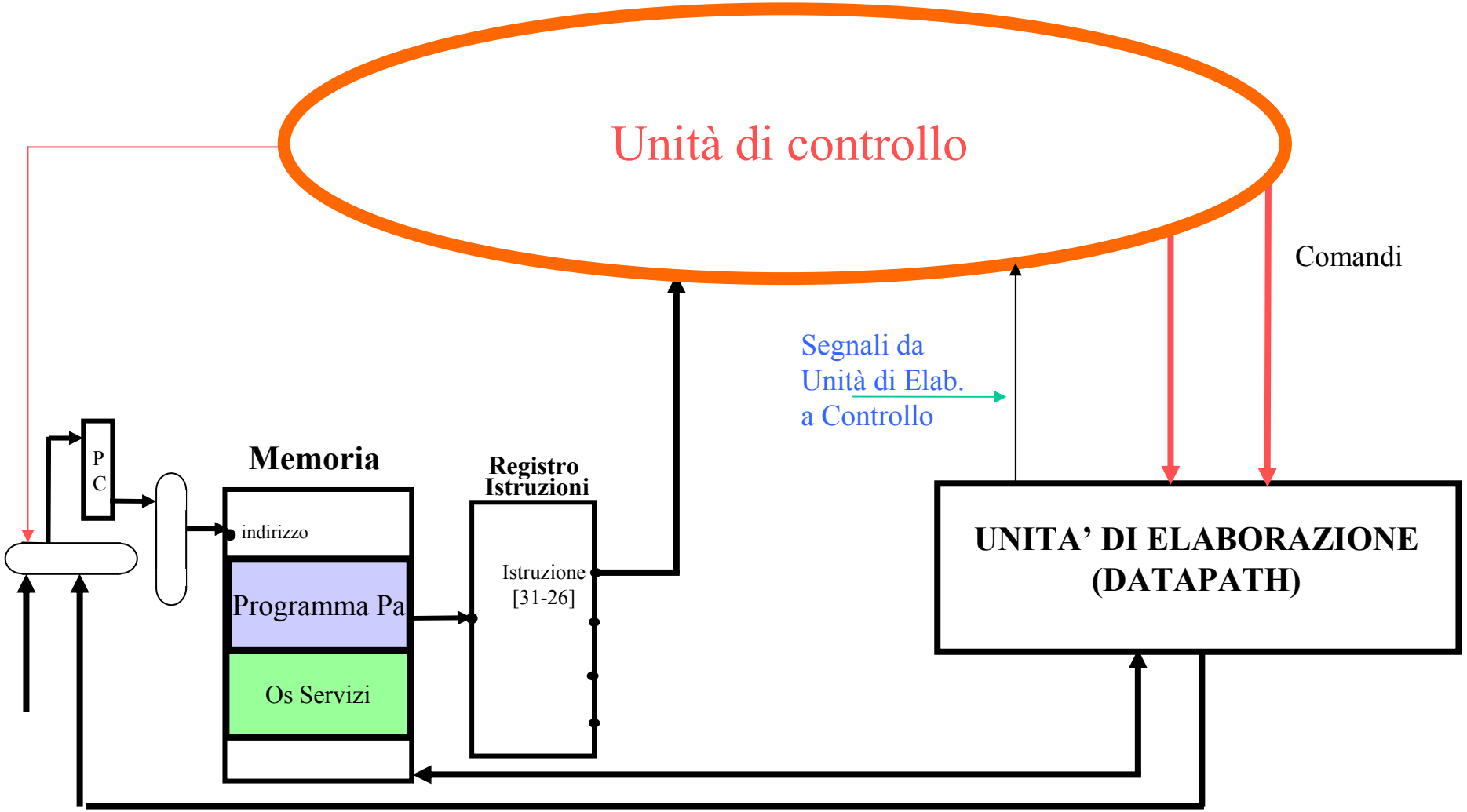
Schema molto semplificato del calcolatore

- Livelli 0-1 -



Schema del processore (e memoria)

Durante l'esecuzione di un programma applicativo Pa, i circuiti interpretano le istruzioni del programma in linguaggio macchina costituito dal < Pa (tradotto) ∘ i servizi OS>



Specifica e realizzazione del controllo

<u>Circuiti di controllo</u>: normalmente, sono presenti due parti

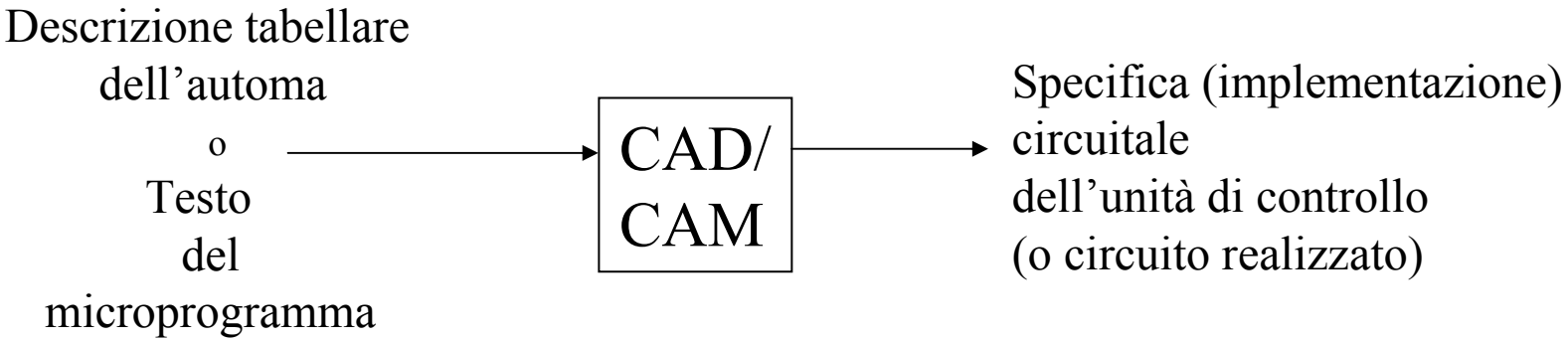
- **Combinatori** (reti combinatorie): per parte del processo di decodifica e di controllo (es. controllo della ALU), ma vedi anche controllo PIPELINE!
  - Specifica: Funzioni logiche | Tabelle di verità
  - Realizzazione: PLA | ROM
- **Sequenziali**: controllo "ad alto livello", p.es. sequenzializzazione operazioni nel controllo multiciclo.
  - Specifica: diagramma a stati finiti (FSM) | microprogramma
  - Realizzazione:

    Macchina a stati finiti (stato esplicito) | Sequenzializzatore (controllo microprogrammato)

(si può passare da una delle due tecniche di specifica ad una delle due tecniche di realizzazione)

Importanza della specifica

Data una specifica, esistono strumenti di progettazione assistita che ne permettono la traduzione in circuiti.
P.es. per il controllo multiciclo



NB: E' assolutamente necessario

- Aver bene in mente le differenze tra specifica ed implementazione
- Saper distinguere nei vari casi la natura sequenziale o combinatoria di una specifica unità di controllo (faremo vari esempi)

Faremo riferimento ad un sottoinsieme delle istruzioni MIPS:

- Istruzioni aritmetiche: add, sub, and, or, slt

  add rd, rs, rt // rd ← rs + rt

  slt rd, rs, rt // rd = 1 se rs < rt, 0 altrimenti

- Istruzioni di accesso a memoria:

  lw rt, offset(rs) // rt ← M[rs+offset]

  sw rt, offset(rs) // M[rs+offset] ← rt

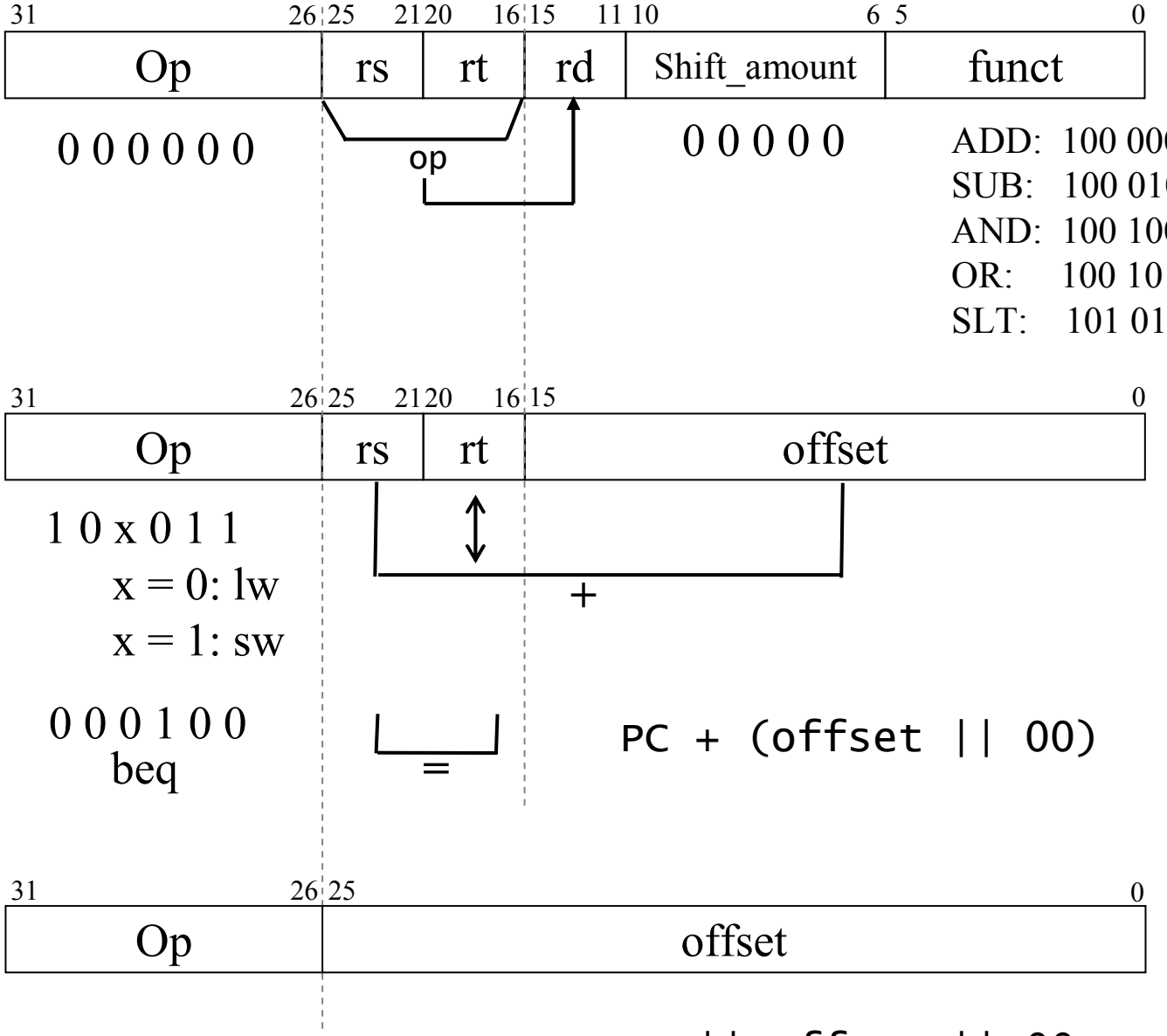
- Istruzioni di salto condizionato:

  beq rs, rt, offset // se rs=rt salta a offset *istruzioni* rispetto a PC
  (aggiornato a istruzione corrente + 4 bytes!)
  in bytes: PC + (offset || 00)

- Salto incondizionato:

  j offset // salta all'indirizzo *in istruzioni* ottenuto da:
  4 bit di PC || offset [30 bit]
  indirizzo in byte è la concatenazione di
  4 bit di PC || offset || 00 [32 bit]

Codifica delle istruzioni viste:

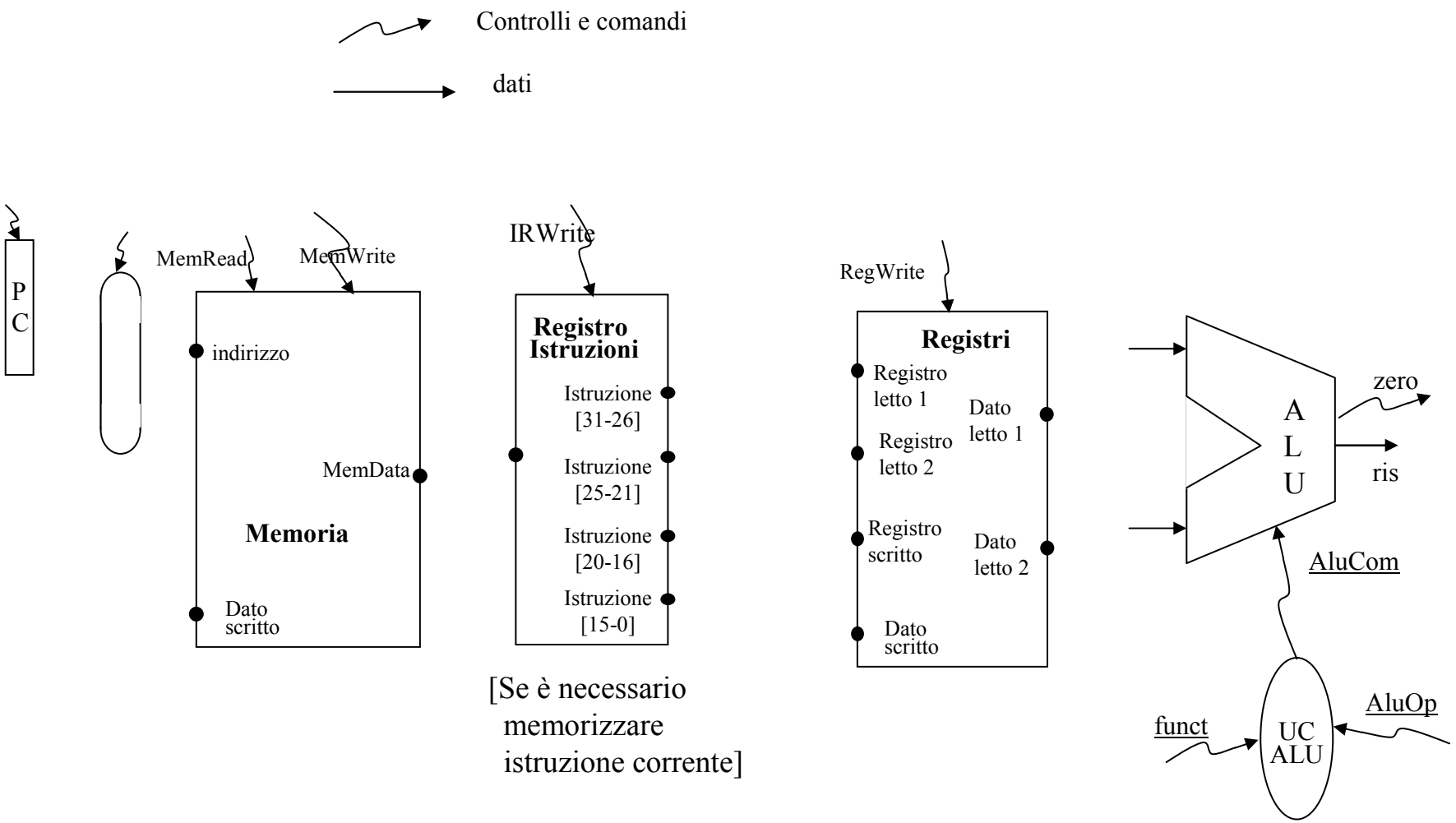


Aritmetiche: Tipo-R

lw, sw, beq: Tipo-I

J: Tipo-J

Glossario Visuale: indica le risorse individuate in base ad una prima analisi

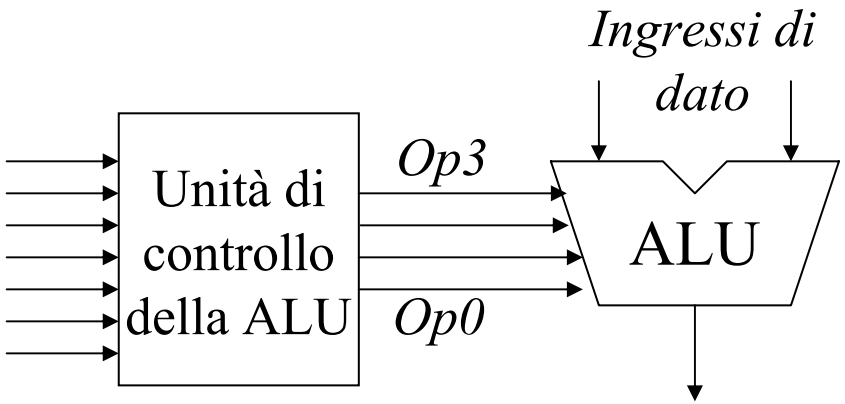


[Se è necessario memorizzare istruzione corrente]

Un esempio di controllo combinatorio: Il controllo della ALU

- Supponiamo che la ALU possa svolgere le 6 operazioni seguenti:
  - AND
  - OR
  - NOR
  - Somma
  - Sottrazione
  - Comparazione di minoranza (set on less than)
- e sia dotata di 4 ingressi di controllo (della ALU) per codificare le operazioni possibili
- Il blocco logico che implementa l'unità di controllo della ALU avrà 4 uscite, corrispondenti ai 4 bit di controllo che codificano una delle 6 operazioni possibili

Chiamiamo ***Op3***, ***Op2***, ***Op1***, ***Op0*** le 4 **uscite** del blocco di controllo della ALU (ovvero l'**ingresso di controllo** della ALU)



Assumiamo la seguente codifica

	Op3	*Op2*	*Op1*	*Op0*
AND	0	0	0	0
OR	0	0	0	1
Somma	0	0	1	0
Sottrazione	0	1	1	0
Set on less than	0	1	1	1
NOR	1	1	0	0

- Gli **ingressi** dell'unità di controllo della ALU derivano da:
  - Un campo di controllo di 2 bit (***ALUOp***) che identifica la classe dell'istruzione da eseguire come segue:

Classe dell'istruzione	*ALUOp*	*Operazione*
Load word, store word	00	somma
Branch on equal	01	sottraz.
Tipo-R	10	***funct***

  - Il campo funzione (***funct***) dell'istruzione (se Tipo-R)

- I bit ALUOp (ingressi dell'unità di controllo ALU) vengono generati dall'unità di controllo principale in base al codice operativo (***op***) della istruzione

Relazione fra ingressi di controllo della ALU e istruzioni

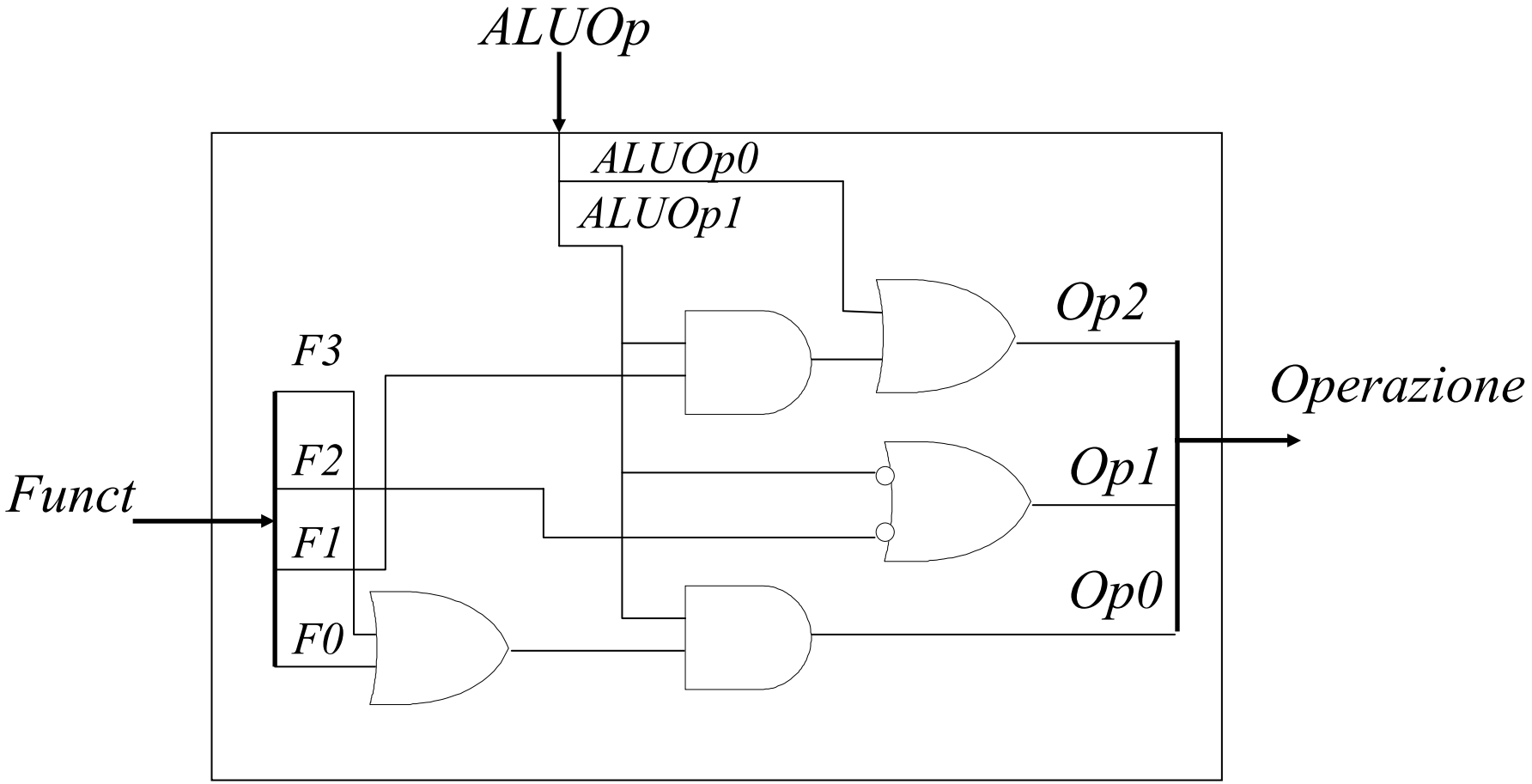
Trascurando la NOR (di cui il testo si dimentica!), le operazioni previste da gran parte delle istruzioni del MIPS possono essere eseguite da una ALU in grado di svolgere le 5 operazioni viste

NB: sono considerati solo Op2…Op0

Codice operativo	ALUOp	Operazione	Campo funct	Azione della ALU	Ingresso di controllo della ALU
LW	00	load word	XXXXXX	somma	010
SW	00	store word	XXXXXX	somma	010
Branch equal	01	branch equal	XXXXXX	sottrazione	110
Tipo-R	10	somma	100000	somma	010
Tipo-R	10	sottrazione	100010	sottrazione	110
Tipo-R	10	AND	100100	and	000
Tipo-R	10	OR	100101	or	001
Tipo-R	10	set on less than	101010	set on less than	111

INGRESSI **USCITE**

Una volta specificata la relazione (combinatoria!) tra ingressi e uscite mediante una tabella di verità, si possono usare le tecniche viste a Calcolatori A per realizzare la rete combinatoria:



NB: nella pratica, per passare da specifica a implementazione si usano strumenti CAD:

- semplificano il processo di realizzazione (riducendo errori)
- utilizzano insiemi strutturati di porte logiche;
  p.es. PLA utilizza approccio a due livelli, generalmente più efficiente

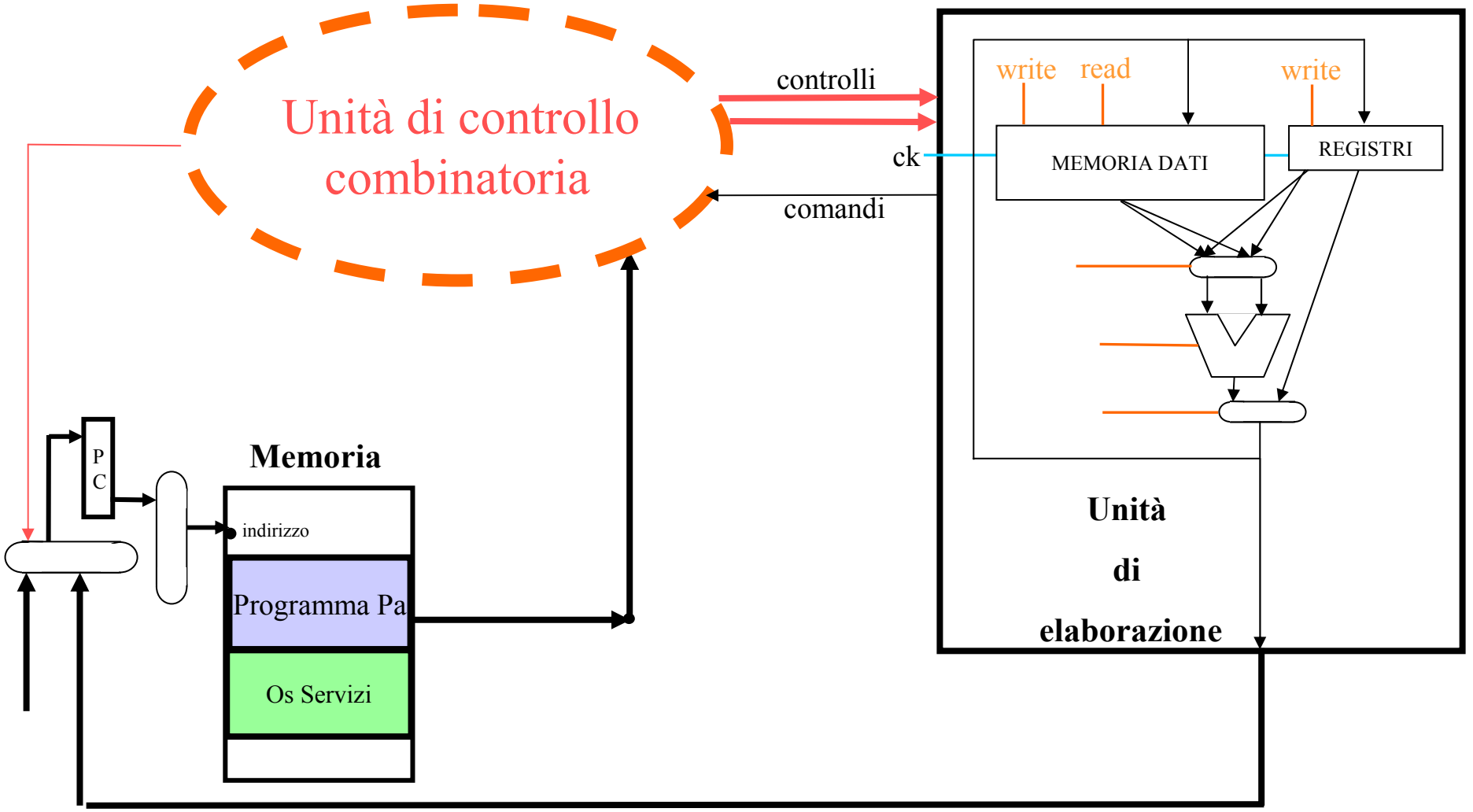
ORA CONSIDERIAMO IL CONTROLLO DEL PROCESSORE.

VEDREMO:

- SINGOLO CICLO
- MULTICICLO
  - FSM
  - MICROPROGRAMMATO
- CON PIPELINE

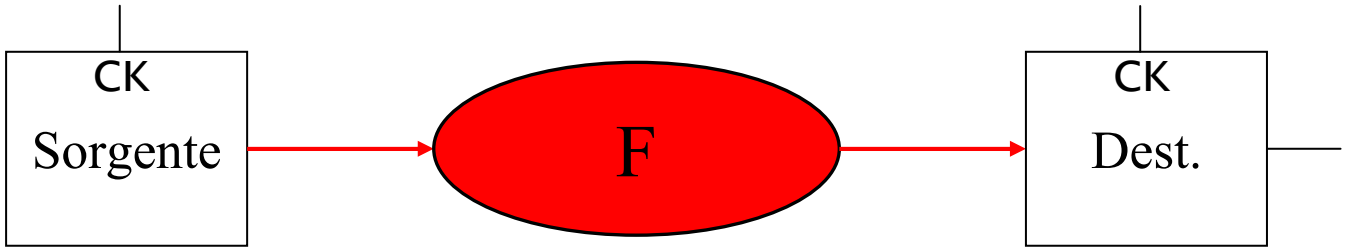
Controllo di un processore a singolo ciclo

NB: schema stilizzato (in particolare, non corrisponde al MIPS)



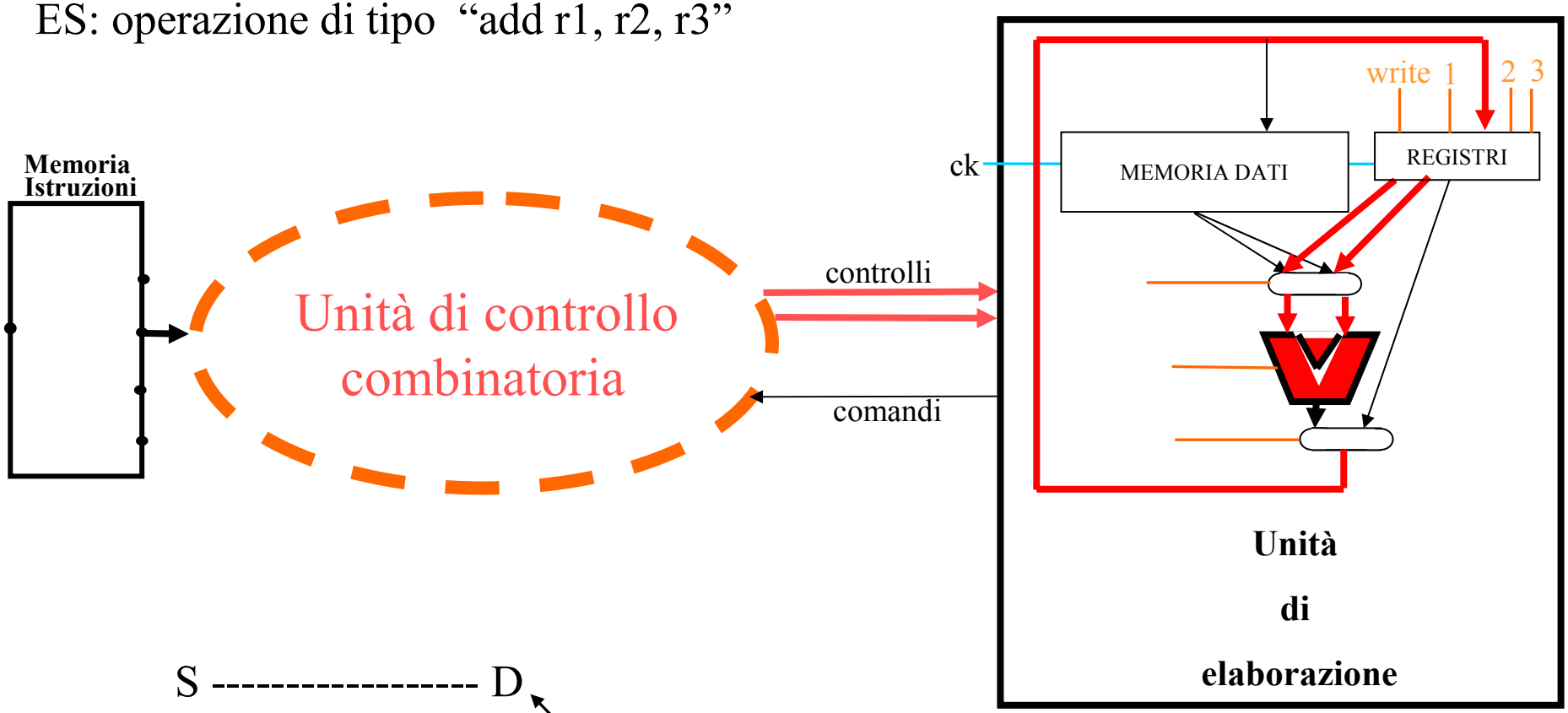
Idea di base

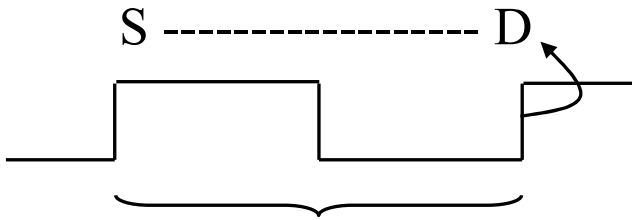
- Ad ogni ciclo di clock, il registro istruzioni contiene l'istruzione corrente
- L'unità di controllo è una rete combinatoria che:
  - riceve in input l'istruzione corrente
  - produce in output segnali di controllo all'unità di elaborazione:
    controllo multiplexer, read e write ad elementi di memoria, controllo ALU
- I segnali di controllo determinano, a seconda del tipo di istruzione:
  - il percorso sorgente-destinazione dei dati mediante:
    indirizzi e numeri registri + segnali di controllo ai multiplexer
  - le operazioni aritmetiche e logiche effettivamente svolte mediante:
    segnali di controllo alle ALU
  - se un elemento di memoria deve scrivere e/o leggere un dato mediante:
    segnali di tipo read/write
- Avremo quindi la determinazione di un "percorso" del tipo:



dove: - sorgente e destinazione possono coincidere
- valore sorgente disponibile "nel corso" del ciclo, destinazione scritta alla fine

ES: operazione di tipo “add r1, r2, r3”





- valori dei registri disponibili (poco dopo – T_{prop} – l’inizio del periodo)
  + si “crea” percorso sorgente-destinazione (con i comandi relativi)
- valori all’ingresso della destinazione (registri) stabili entro la fine del ciclo
- questi valori sono scritti nel fronte successivo – disponibili prossimo ciclo

Conseguenze:

• E' possibile condividere elementi tra istruzioni diverse
(eseguite in cicli diversi del clock!), però…

• Ogni elemento che venga usato più di una volta nell'esecuzione di una istruzione deve essere duplicato:

- Memoria dati (lw, sw)

≠

Memoria programmi (fase di fetch di tutte le istruzioni)

- ALU (lw, sw, beq, aritmetiche)

≠

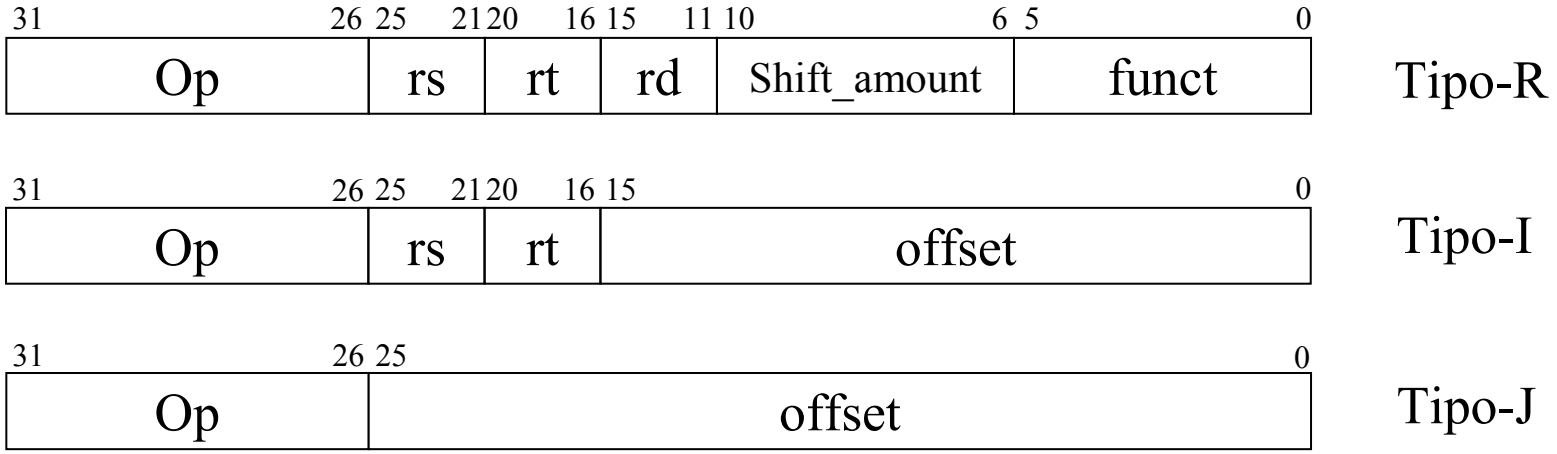
Sommatore PC + 4 (tutte le istruzioni)

≠

Sommatore PC+offset (beq)

Consideriamo il caso del MIPS, di cui si vogliono implementare le istruzioni:

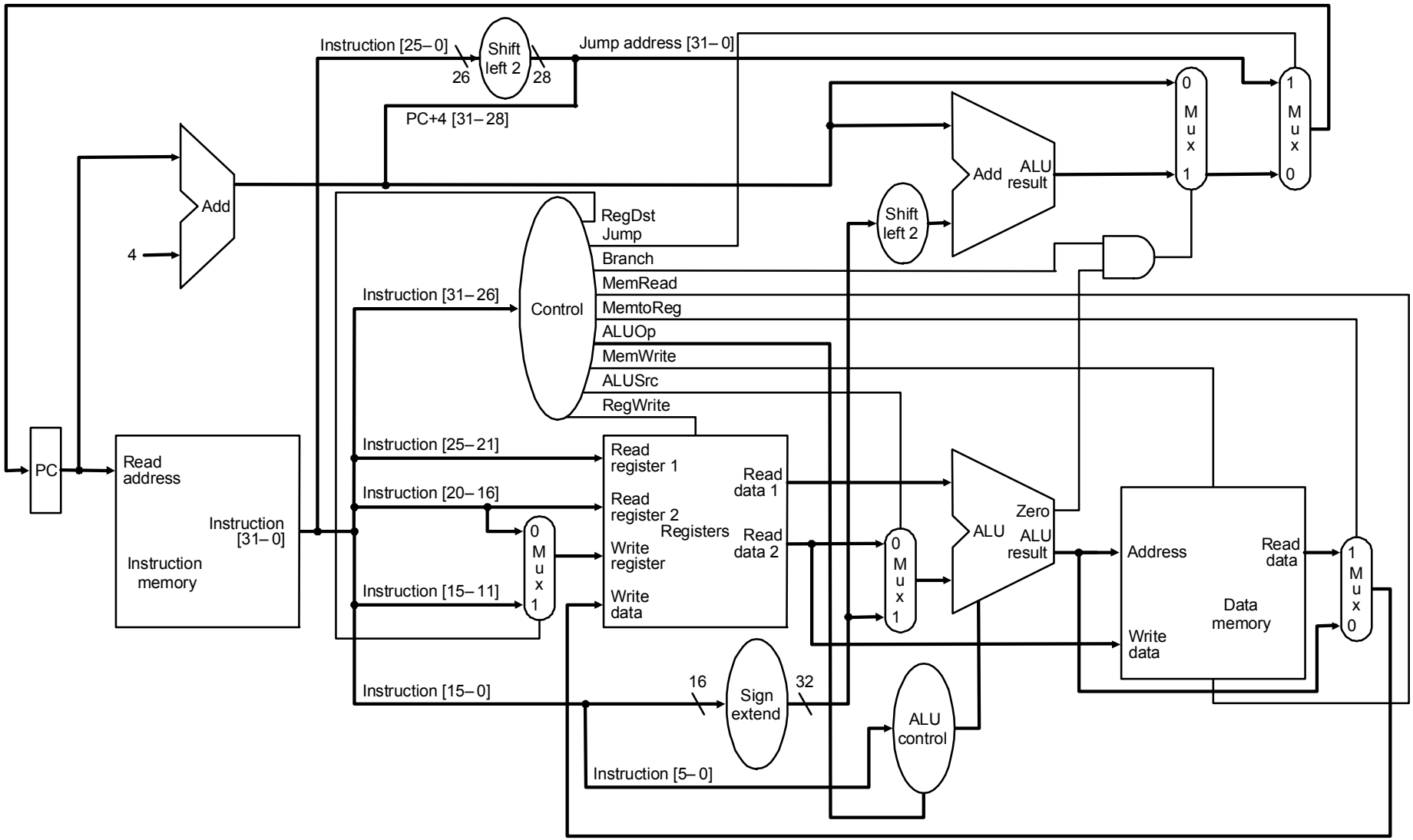
- Aritmetiche e logiche } Tipo-R
- lw, sw } Tipo-I
- beq } Tipo-I
- j } Tipo-J



La progettazione è facilitata dal fatto che:

- Campo Op sempre in [31-26]
- i registri da leggere sono sempre in rs e rt
- il registro base [da sommare] per lw e sw è rs [primo ingresso della ALU]
- l'offset a 16 bit da sommare per beq, lw, sw è in [15-0]

Il registro su cui scrivere può essere rd (per operaz. di TIPO-R) o rt (per lw)

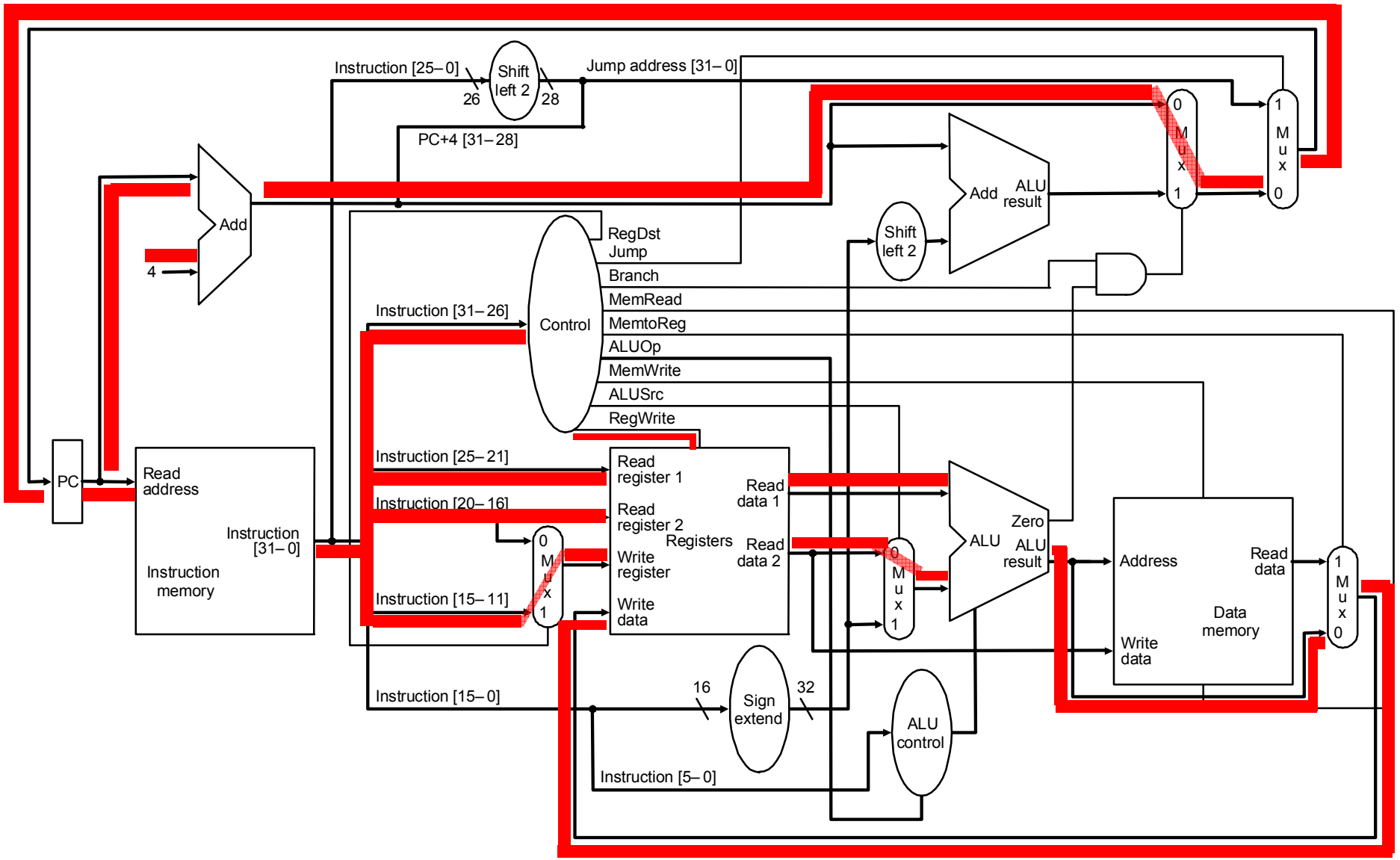

Instruction [25–0]
Shift left 2
26
28
Jump address [31–0]
PC+4 [31–28]
Add
4
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
Shift left 2
Add
ALU result
0
M
u
x
1
1
M
u
x
0
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU
Zero
ALU result
ALU control
Address
Read data
Data memory
Write data

Si vede che:

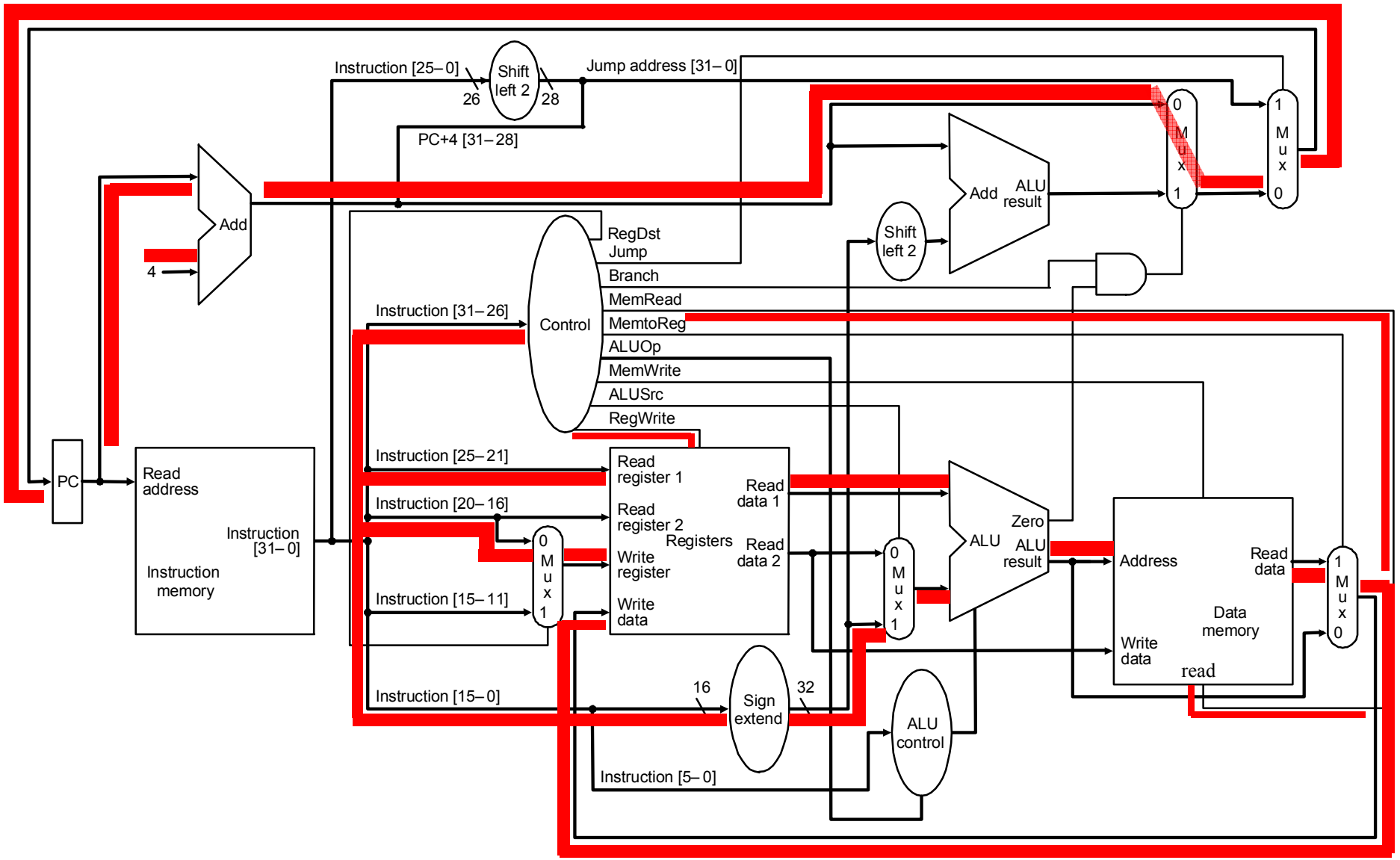
- i segnali di controllo sono determinati in modo “combinatorio” soltanto sulla base del campo Opcode
- non è in generale possibile prevedere l’ordine di arrivo dei segnali di controllo: le operazioni non sono eseguite “in sequenza”, controllo combinatorio (è necessario che T_{clock} sia sufficientemente lungo)

Vediamo allora alcuni esempi di esecuzione delle istruzioni…

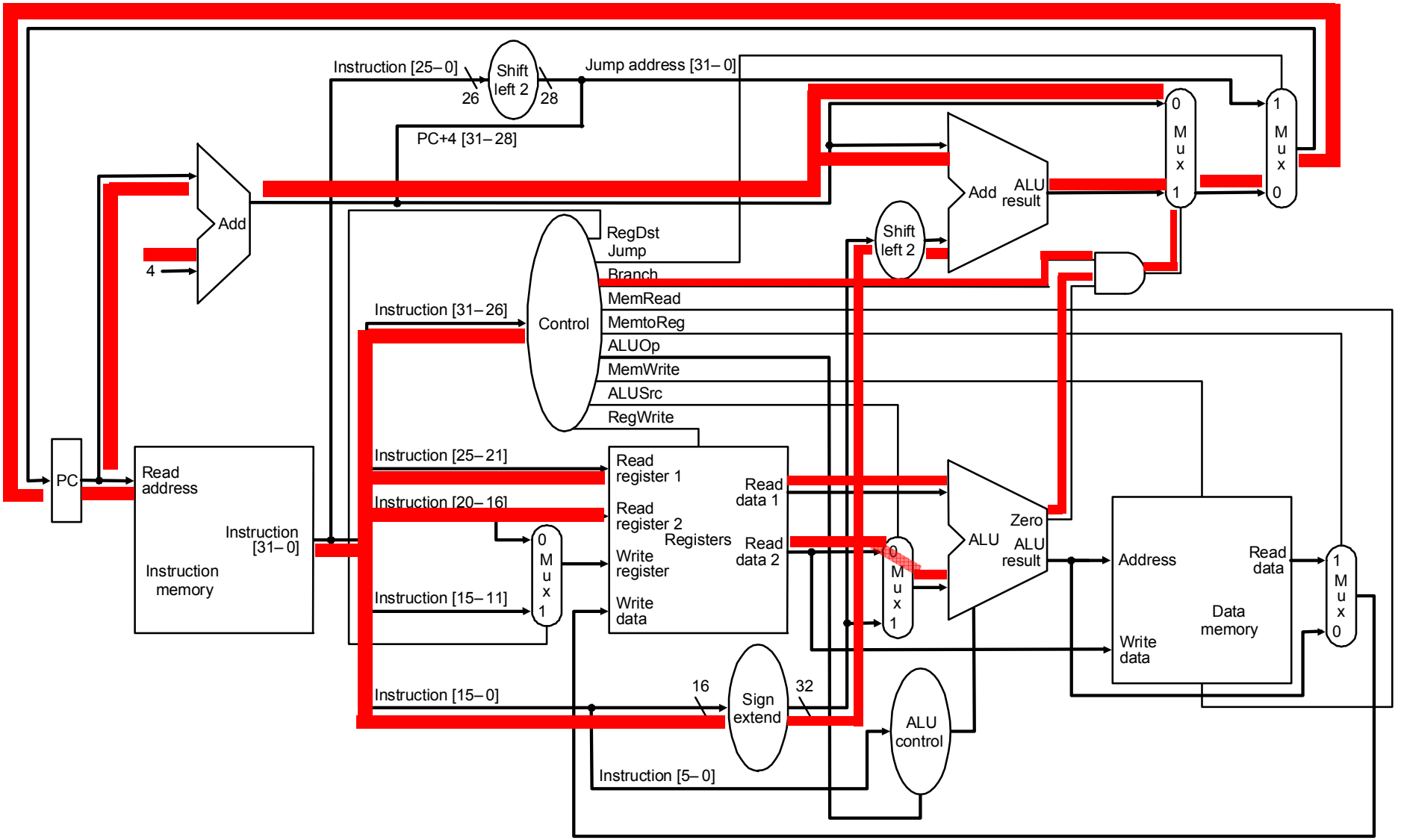
Esempio: istruzione di tipo-R



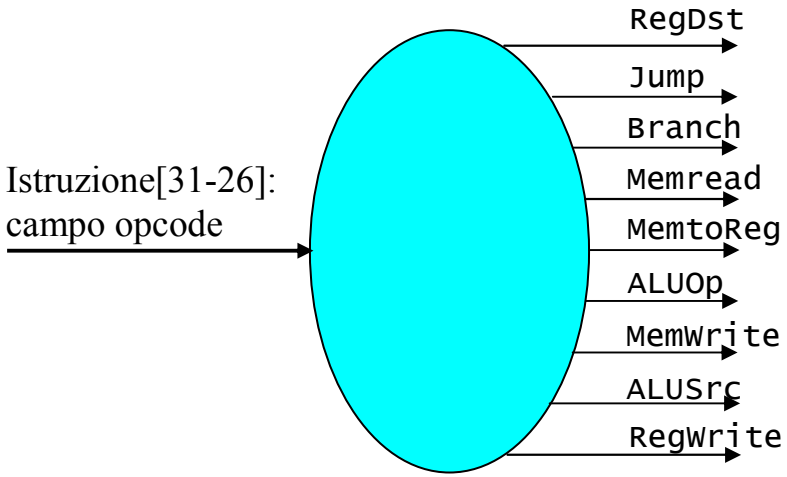
Esempio: istruzione di load



Esempio: istruzione beq



Progetto e realizzazione dell'unità di controllo principale



Progetto e realizzazione corrispondono a quelli di una rete combinatoria:

ISTRUZ	RegDst	Jump	Branch	Mem Read	Memto Reg	ALUOp	Mem Write	ALUSrc	Reg Write
Tipo-R	1	0	0	0	0	10	0	0	1
lw	0	0	0	1	1	00	0	1	1
sw	X	0	0	0	X	00	1	1	0
beq	X	0	1	0	X	01	0	0	0
j	X	1	X	0	X	XX	0	X	0

Opcode [6 bit]

Richiamo su prestazioni

Si ricorda che le prestazioni sono valutate sul tempo di esecuzione di un programma!
Per valutare soluzioni progettuali diverse, bisogna far riferimento
allo stesso programma (o classe di programmi!)

$$T_{esecuzione} = numero\_istruzioni * \underbrace{CPI * T_{clock}}$$

Tempo medio di esecuzione per istruzione

Fisse a parità di
Instruction Set Architecture

T_{clock} non varia al variare delle istruzioni

[progettazione di sistemi "a clock variabile" non adottata in pratica]

E' necessario conoscere la distribuzione delle singole istruzioni per valutare CPI

[il numero di cicli di clock impiegati varia da istruzione a istruzione]

Esempi di processori che usano controllo a singolo ciclo:

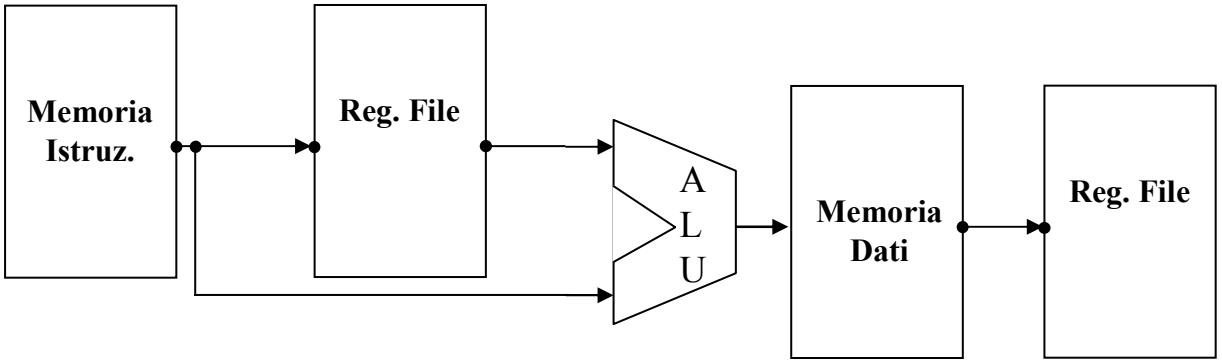
NESSUNO!

Perché?

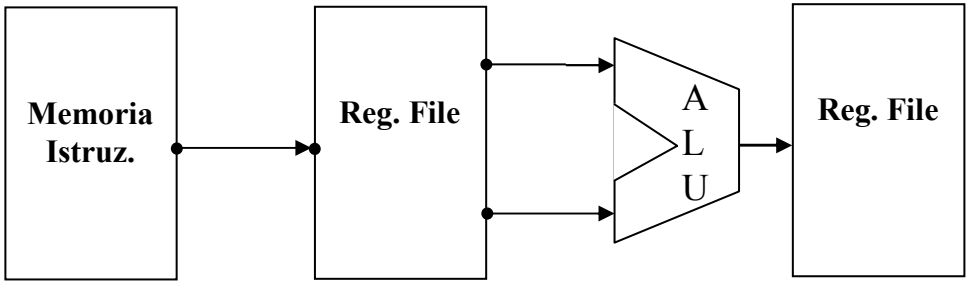
- Periodo di clock: abbastanza lungo per garantire la stabilità dei segnali attraverso il percorso più lungo ⇒ tempo di esecuz. costante per diverse istruz.

  ➡ Istruzioni "più lente" limitano le istruzioni "più veloci"!

Es. lw



Es. Tipo-R



Limitazione aggravata da

- istruzioni che utilizzano decine di unità funzionali in serie, ad esempio comprendenti calcoli in virgola mobile!

⇒ CPI = 1, ma T_{clock} alto, ovvero frequenza di clock molto bassa!

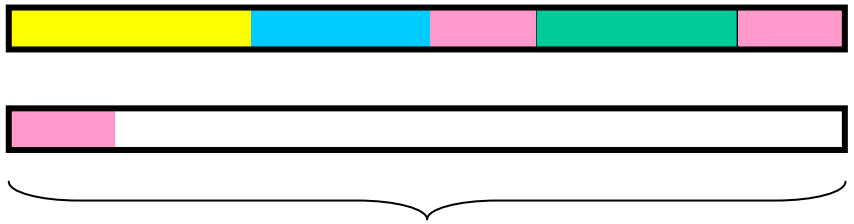
Nel complesso, il tempo di esecuzione di una istruzione è sempre pari al caso peggiore, ovvero a quello dell'istruzione più complessa e lunga

• Altro svantaggio: duplicazione dell'HW ⇒ costi elevati!

Nell'esempio del MIPS, come visto:

- occorrono due memorie diverse [dati e istruzioni]
  [NB: questo va comunque bene perché negli attuali calcolatori si usa memoria cache + pipeline]
- occorrono tre ALU, una per istruzioni aritmetiche e confronto operandi in beq, una per calcolare PC+4 ed una per calcolare indirizzo di salto beq
  [la cosa si complica considerando modalità di indirizzamento complesse, ad esempio quelle con autoincremento]

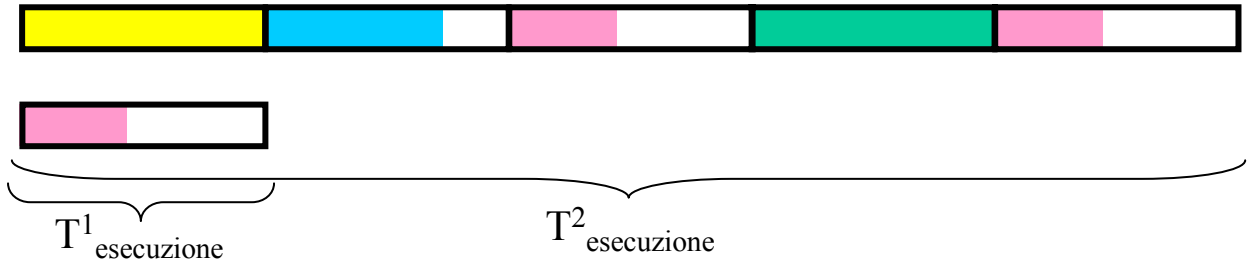
Una strategia alternativa



Istruzione lunga

Istruzione potenzialmente breve

Suddividere le istruzioni in “fasi”: un ciclo di clock per una fase!



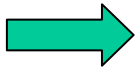
NB: $T^2_{esecuzione}$ (il caso peggiore) può in generale essere maggiore del precedente!

Tuttavia, le istruzioni più lunghe sono anche meno frequenti:

Principio “rendere più veloce l’evento più frequente” comporta un guadagno!

Controllo di un processore-multiciclo

- Ogni istruzione divisa in fasi
- Ad ogni ciclo di clock una fase: controllo sequenziale



- E' possibile utilizzare una stessa unità funzionale più volte nel corso di una istruzione:
  - unica memoria per istruzioni e dati
  - una sola ALU, che svolge le funzioni logico aritmetiche delle istruzioni di Tipo-R e beq (per confrontare rs e rt), somma PC+4 (per fetch), somma per calcolare indirizzi nei salti condizionati (beq)
- Necessità di memorizzare risultati intermedi in registri temporanei (non visibili al programmatore):
  - registri temporanei: salvano dati prodotti in un ciclo di clock e utilizzati dalla stessa istruzione in un ciclo di clock successivo
  - registri visibili al prog: dati utilizzati da istruzioni successive

Come visto, ciascuna fase deve essere sufficientemente breve (T_{clock} breve)

⇒ Bilanciare la suddivisione in fasi,
evitare di mettere in serie più unità funzionali lente.

⇒ Faremo in modo di non mettere in serie più di una delle operazioni:
- accesso in memoria (istruzioni o dati)
- accesso al register file (due letture e una scrittura)
- operazioni della ALU

(NB: accesso/scrittura in un registro singolo considerato non oneroso)

⇒ Ciascuna di queste unità [memoria, register file, ALU] necessita di registri temporanei per memorizzarne il risultato.

Sono inoltre necessari multiplexer aggiuntivi per instradare
i dati verso le unità funzionali riutilizzate nella stessa istruzione
[p.es. MUX per selezionare indirizzo memoria tra PC (per fetch)
o uscita ALU (per lw o sw)]

Esistono diverse modalità di specifica e realizzazione del controllo…

Specifica progressiva del sistema di calcolo di un sistema multiciclo

Di seguito sono schematizzate le attività che una macchina multiciclo svolge per eseguire le istruzioni del set del MIPS ridotto: le fasi rispettano i vincoli posti in precedenza sulle operazioni che si possono eseguire in serie.

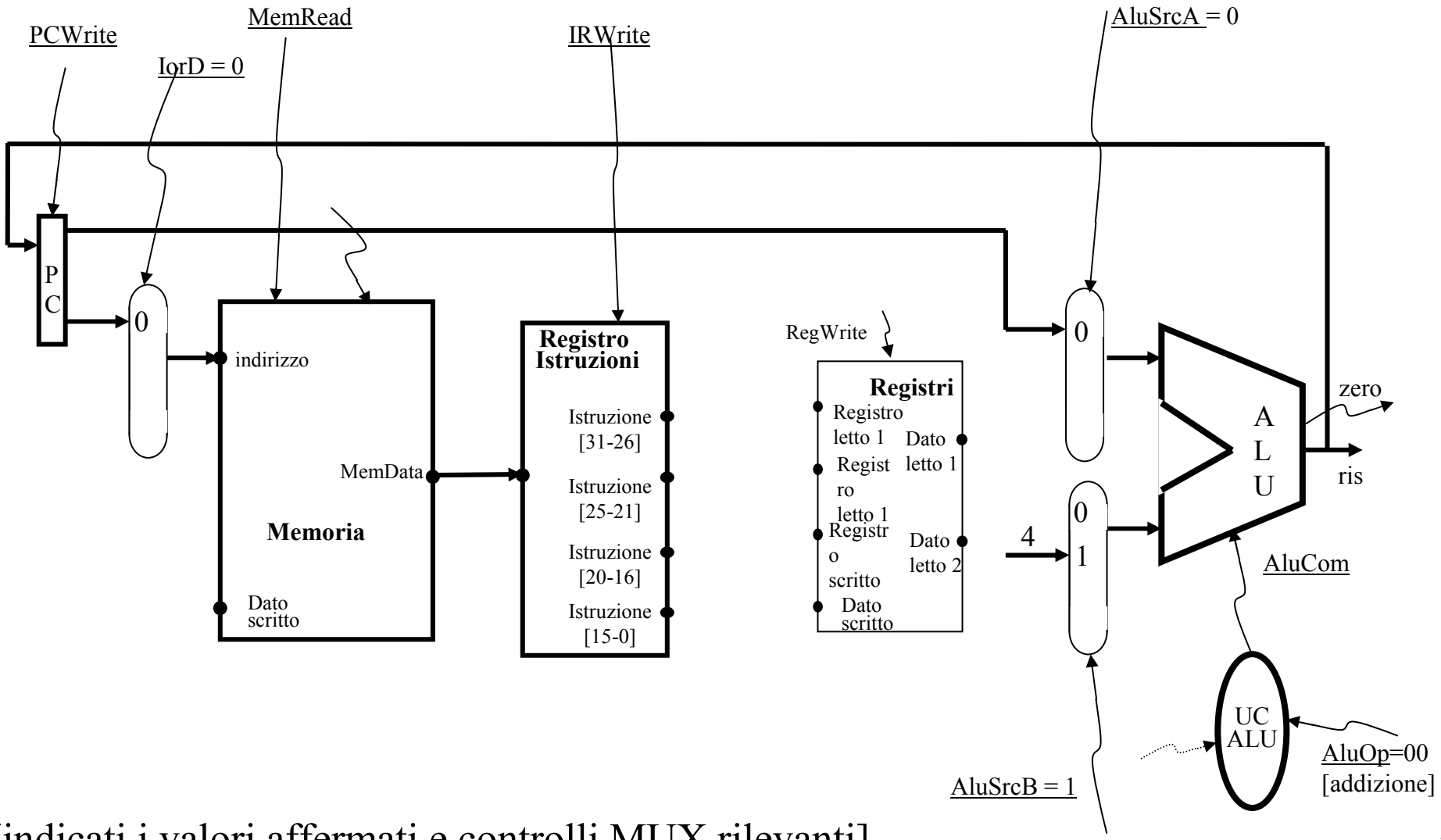
Nello specificare le diverse attività di calcolo si sono seguite le convenzioni adottate nel testo. In particolare, sono rese visibili le risorse impegnate nel ciclo in esame.

La combinazione delle immagini risulta nello schema 5-28 del testo.

I ciclo:Prelievo dell'istruzione

I ciclo: IR ← memoria [PC]

ottimistiche: PC ← PC + 4



[indicati i valori affermati e controlli MUX rilevanti]

NB:

- Durante il primo ciclo di clock vengono creati i due percorsi paralleli:
  - PC_out → Memoria → IR
  - PC_out → ALU → PC_in

  + vengono attivati i segnali di scrittura in IR e PC

- Alla fine del periodo di clock (fronte del clock):
  - PC e IR vengono scritti;

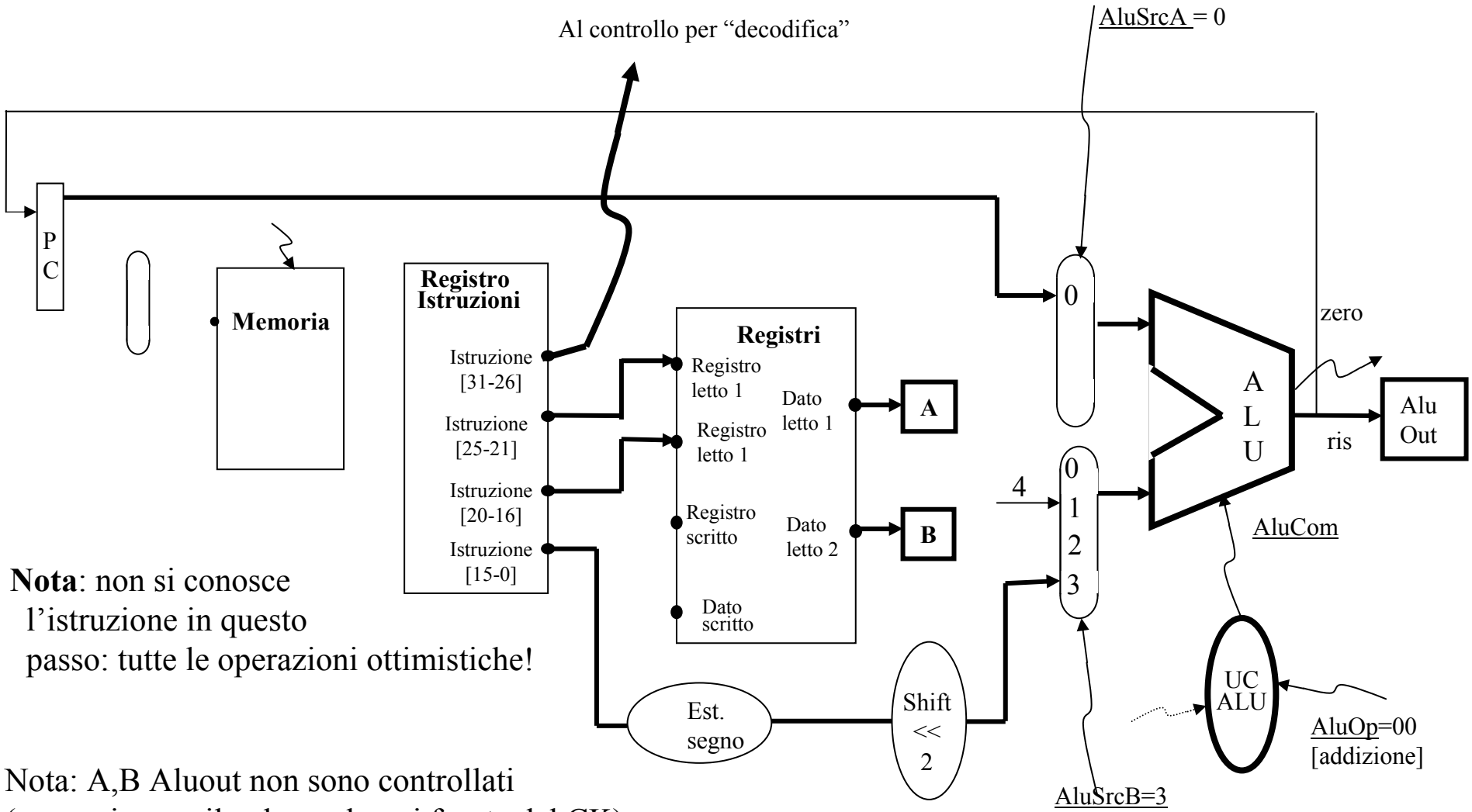
  i valori sono disponibili nel ciclo di clock successivo

Al secondo ciclo di clock IR ha il valore dell'istruzione da eseguire,
PC sarà già incrementato…

NB: si noti che ciascun percorso contiene soltanto un elemento "critico"
(memoria e ALU);
sorgente e destinazione infatti sono registri singoli (PC e ALU)

II ciclo:Decodifica e caricamento registri

ottimistiche: alimentazione registri: A← registro[IR[25-21]]; B ← registro[IR[20-16]]
calcolo indirizzo branch: AluOut ← PC + (sign_ext(IR[15-0]) << 2)

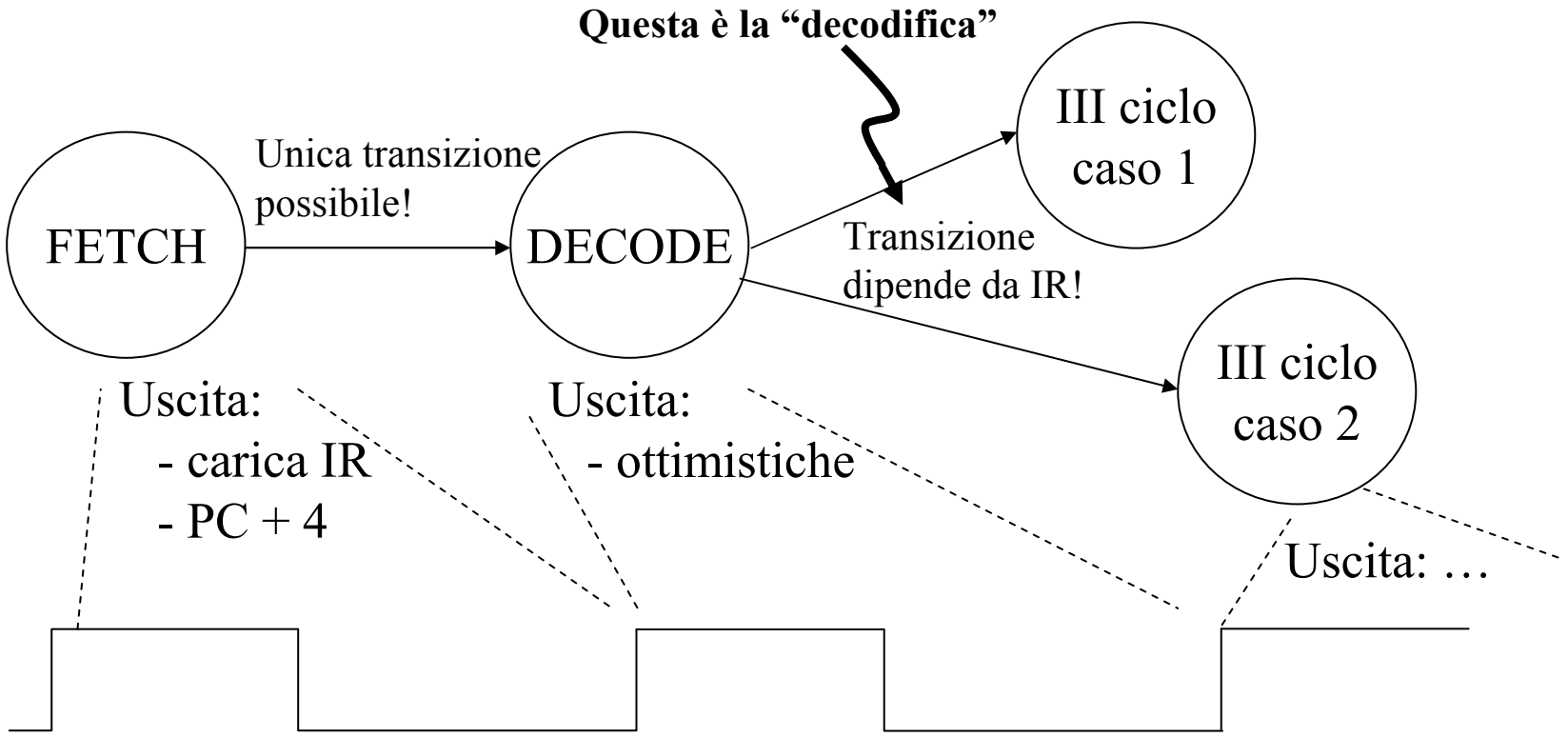


Nota: non si conosce l'istruzione in questo passo: tutte le operazioni ottimistiche!

Nota: A,B Aluout non sono controllati (memorizzano il valore ad ogni fronte del CK)

NB: si è detto che nel secondo ciclo è già disponibile l'istruzione in IR.
D'altra parte si dice anche che "non si conosce quale sia l'istruzione"
e quindi si effettua la decodifica + le uniche operazioni svolte sono "ottimistiche"

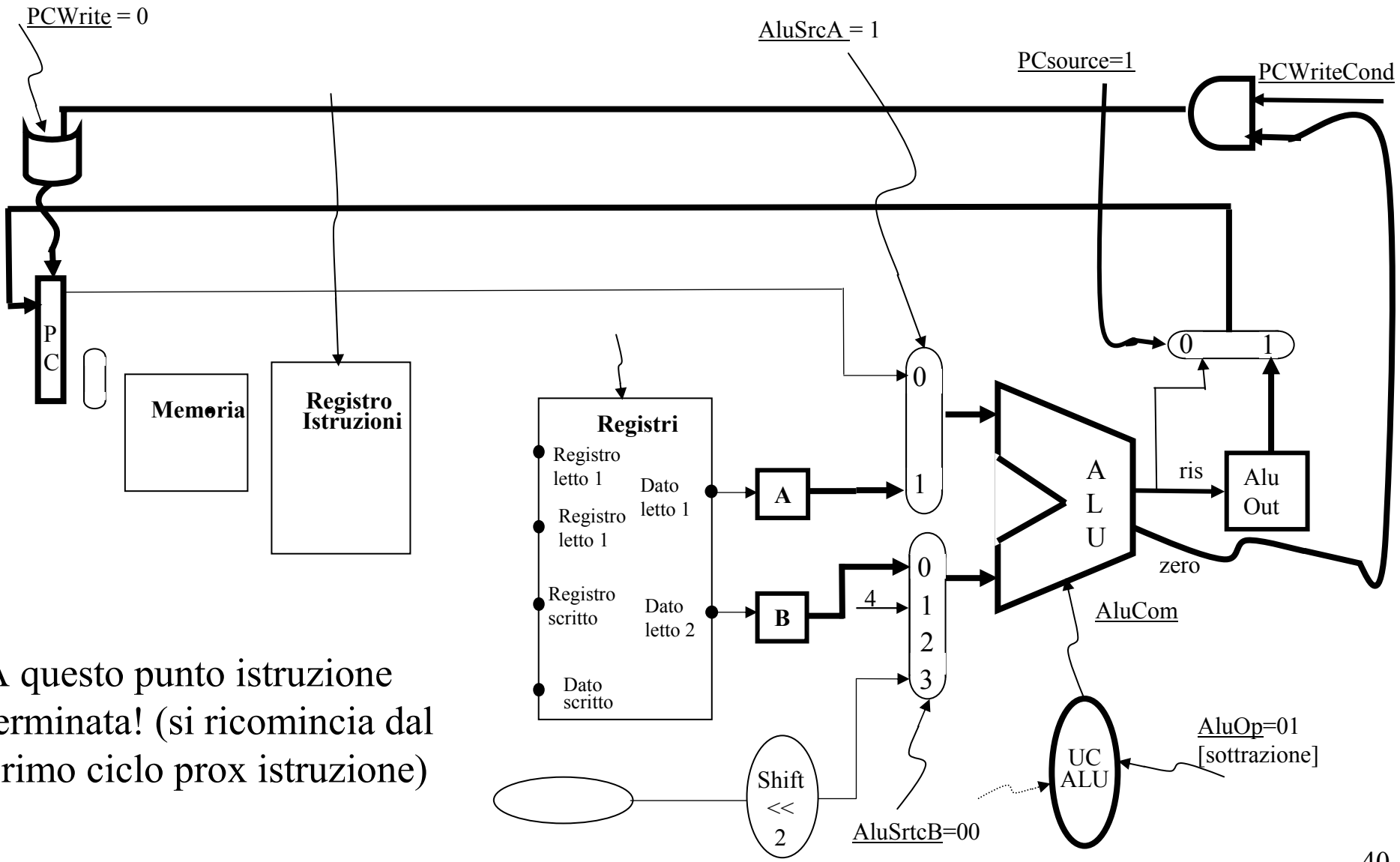
Ciò perché si usa il modello di Moore: uscite del sistema di controllo dipendono solo dallo stato corrente, non dagli ingressi [IR].



III ciclo:se è branch (beq)

IF ([A]= = [B]): PC← [AluOut]

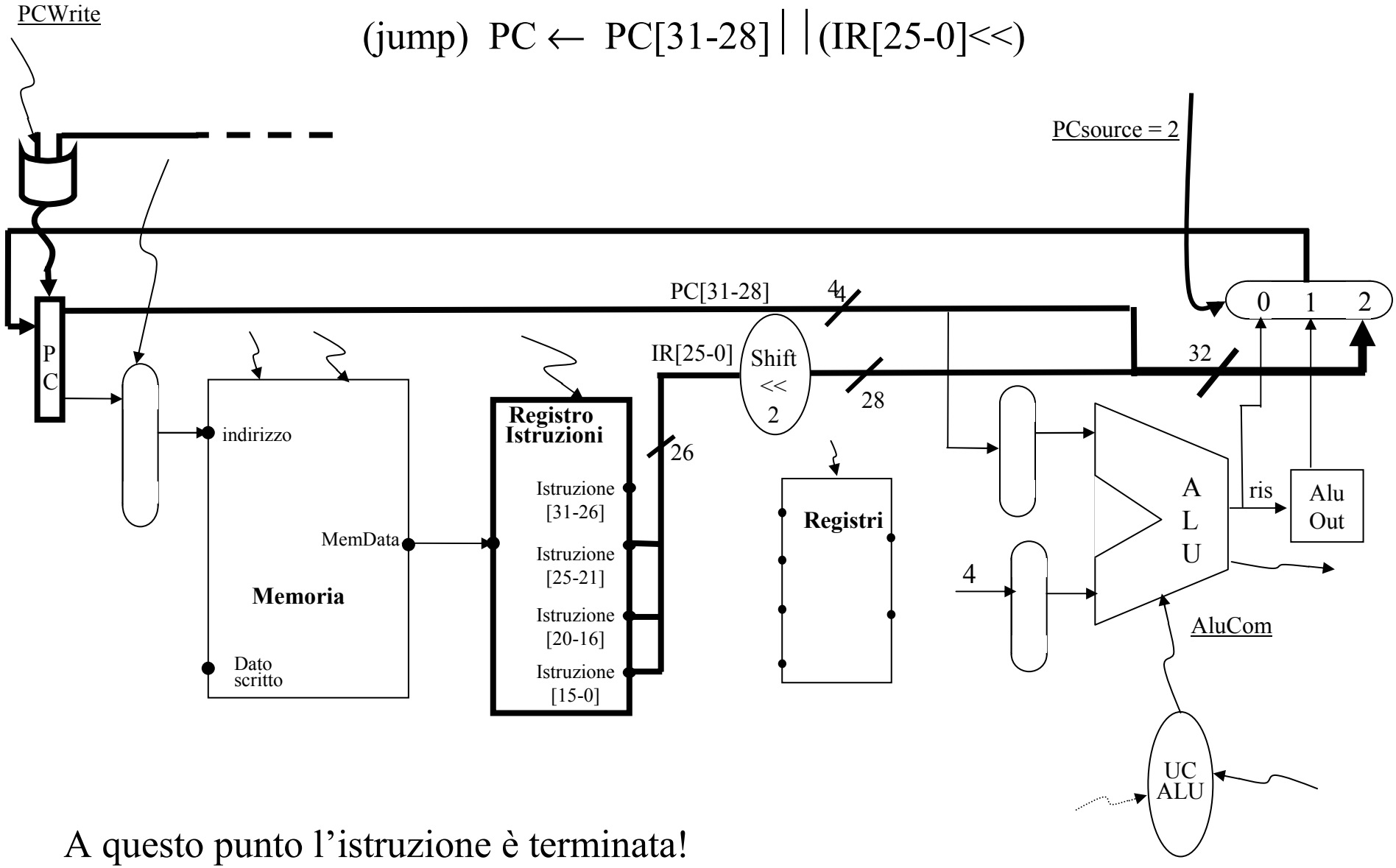
Attenzione: si ritocca il data path e si introduce PCsource



A questo punto istruzione terminata! (si ricomincia dal primo ciclo prox istruzione)

III ciclo: se è salto incondizionato

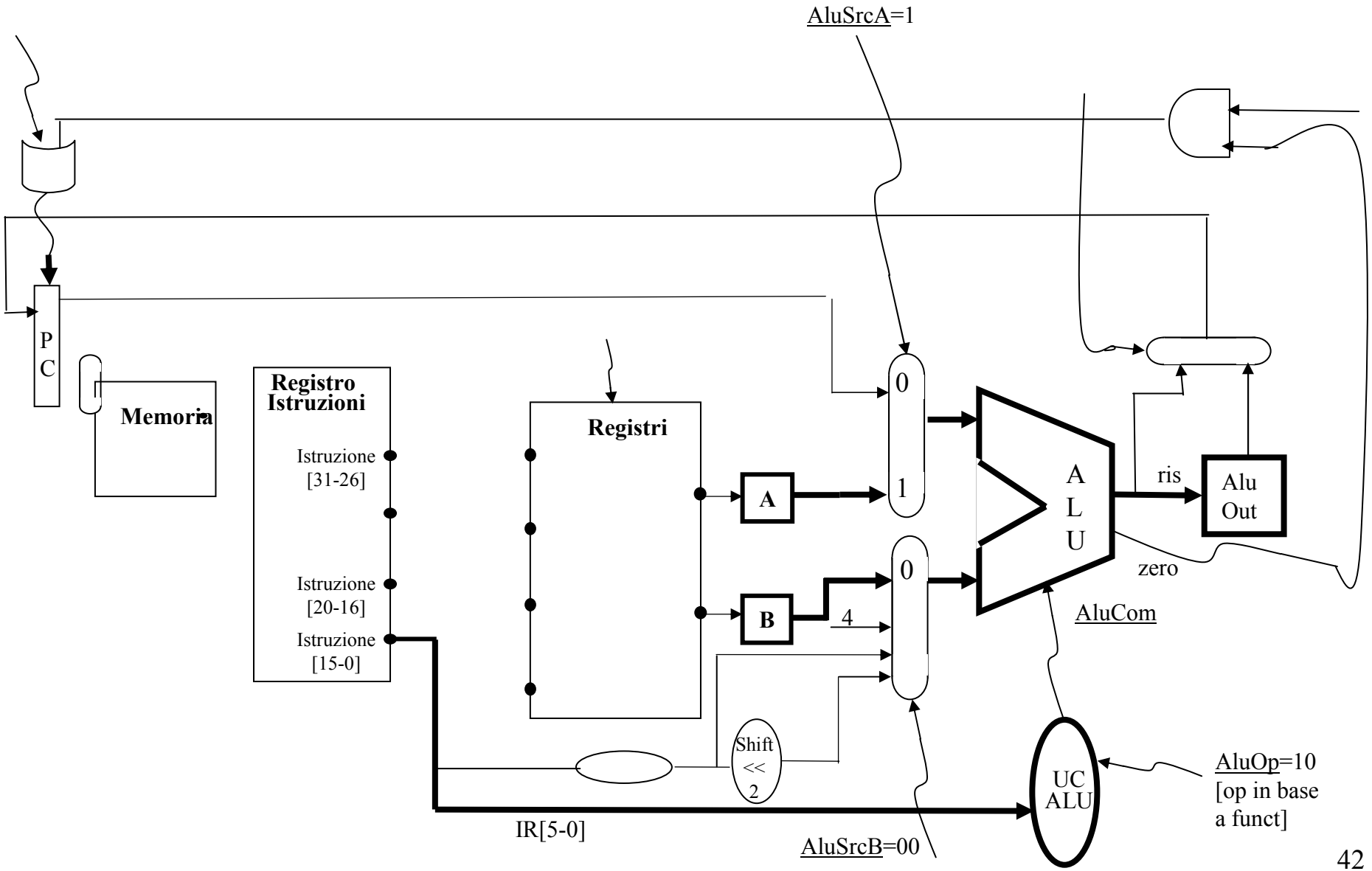
(jump) PC ← PC[31-28]||(IR[25-0]<<)



A questo punto l'istruzione è terminata!
(si ricomincia dal primo ciclo prossima istruzione)

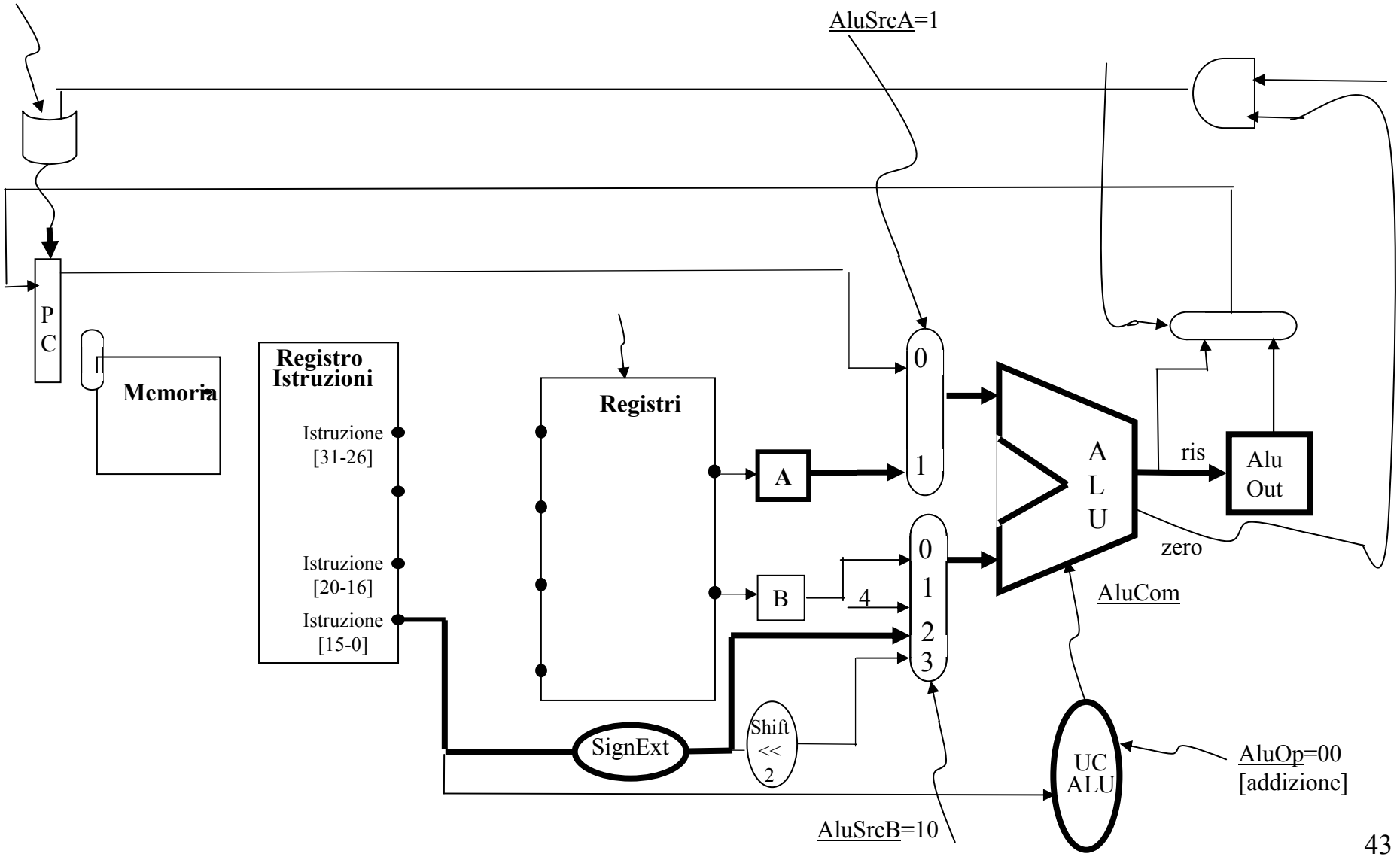
III ciclo:per tipo R

AluOut← A op B



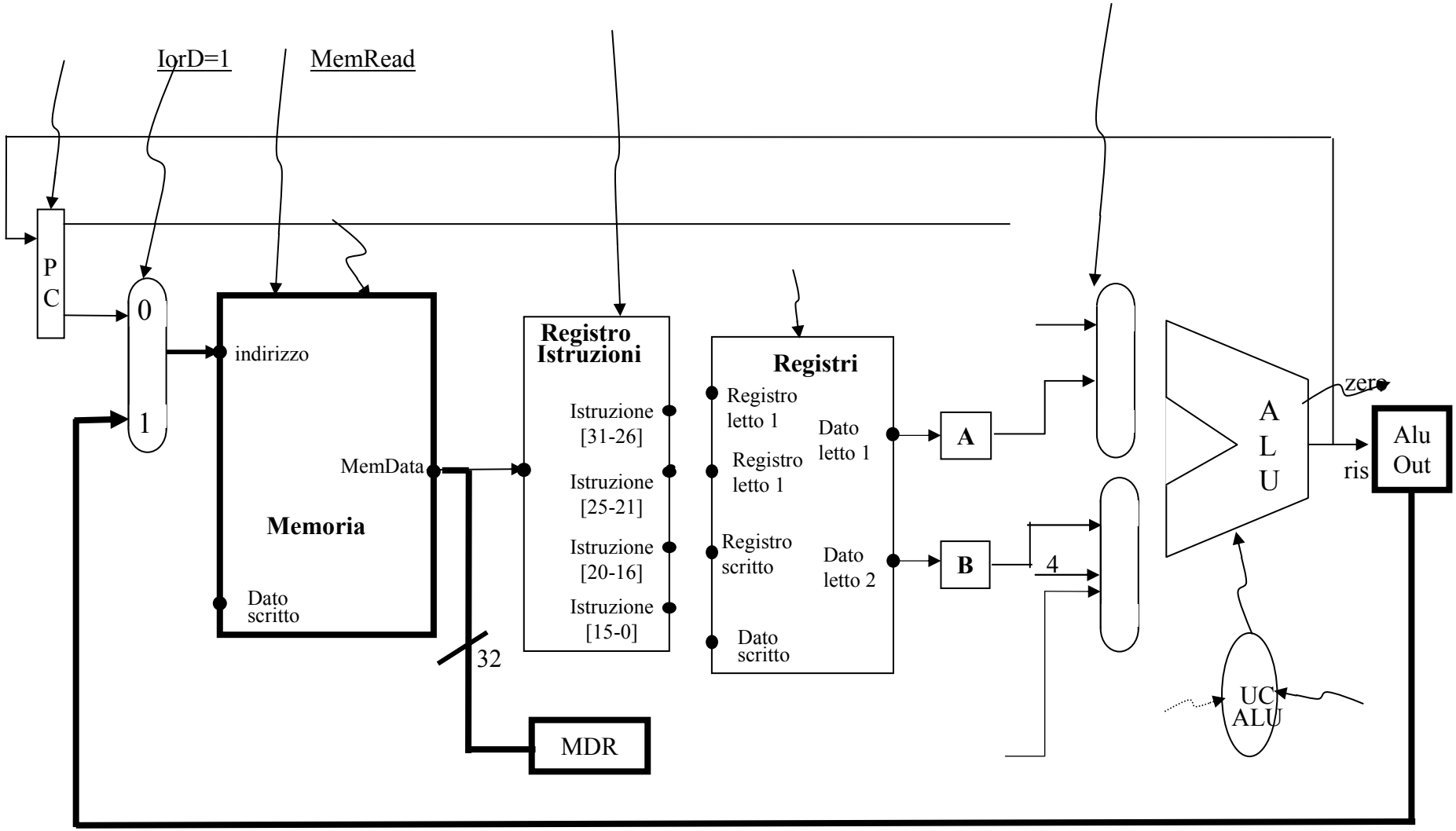
III ciclo:per tipo accesso a memoria

AluOut← [A]+ (signExt([IR[15-0]])



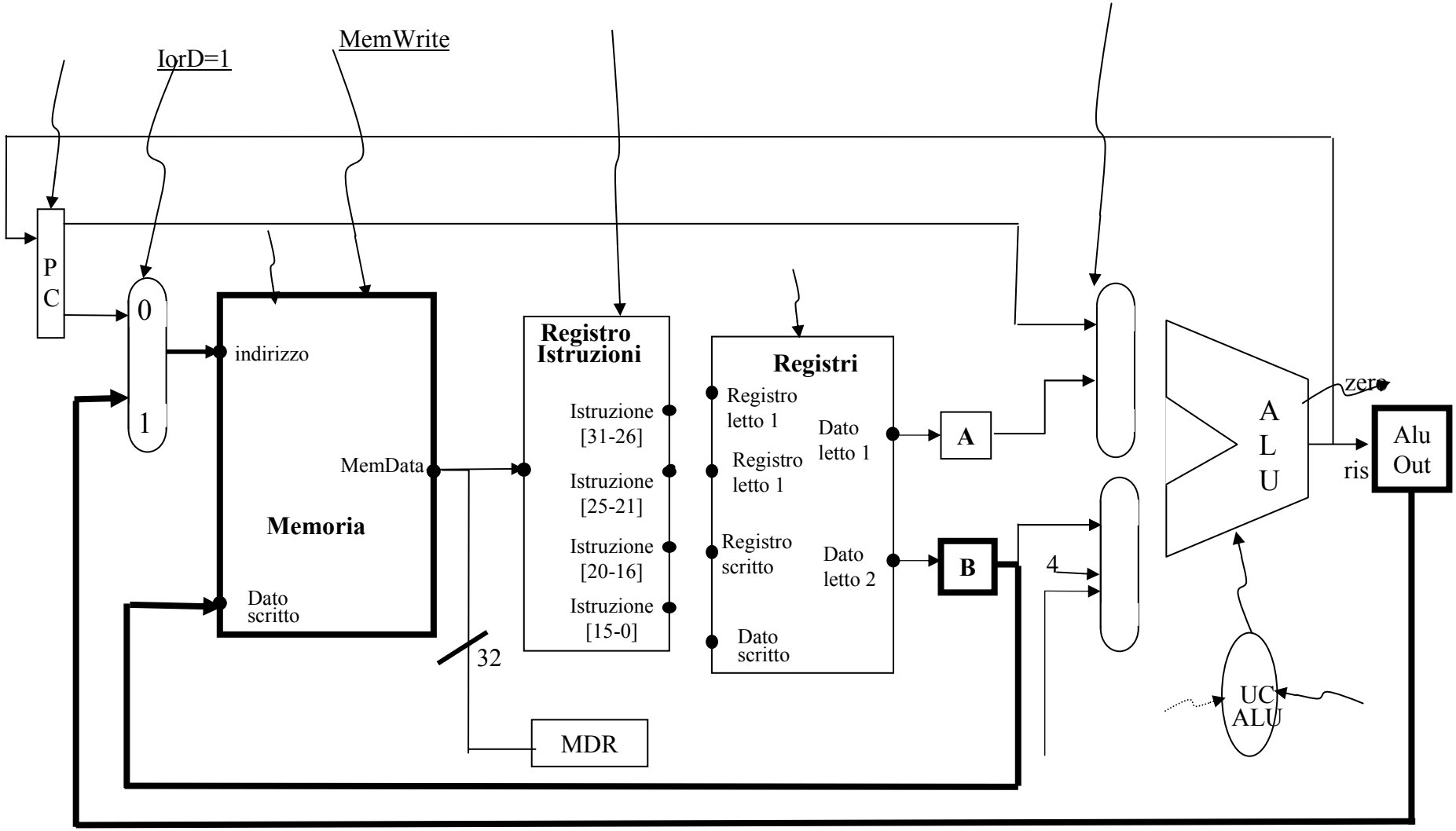
IV ciclo: carica parola (lw)

MDR ← memoria [AluOut]



IV ciclo: memorizza parola (sw)

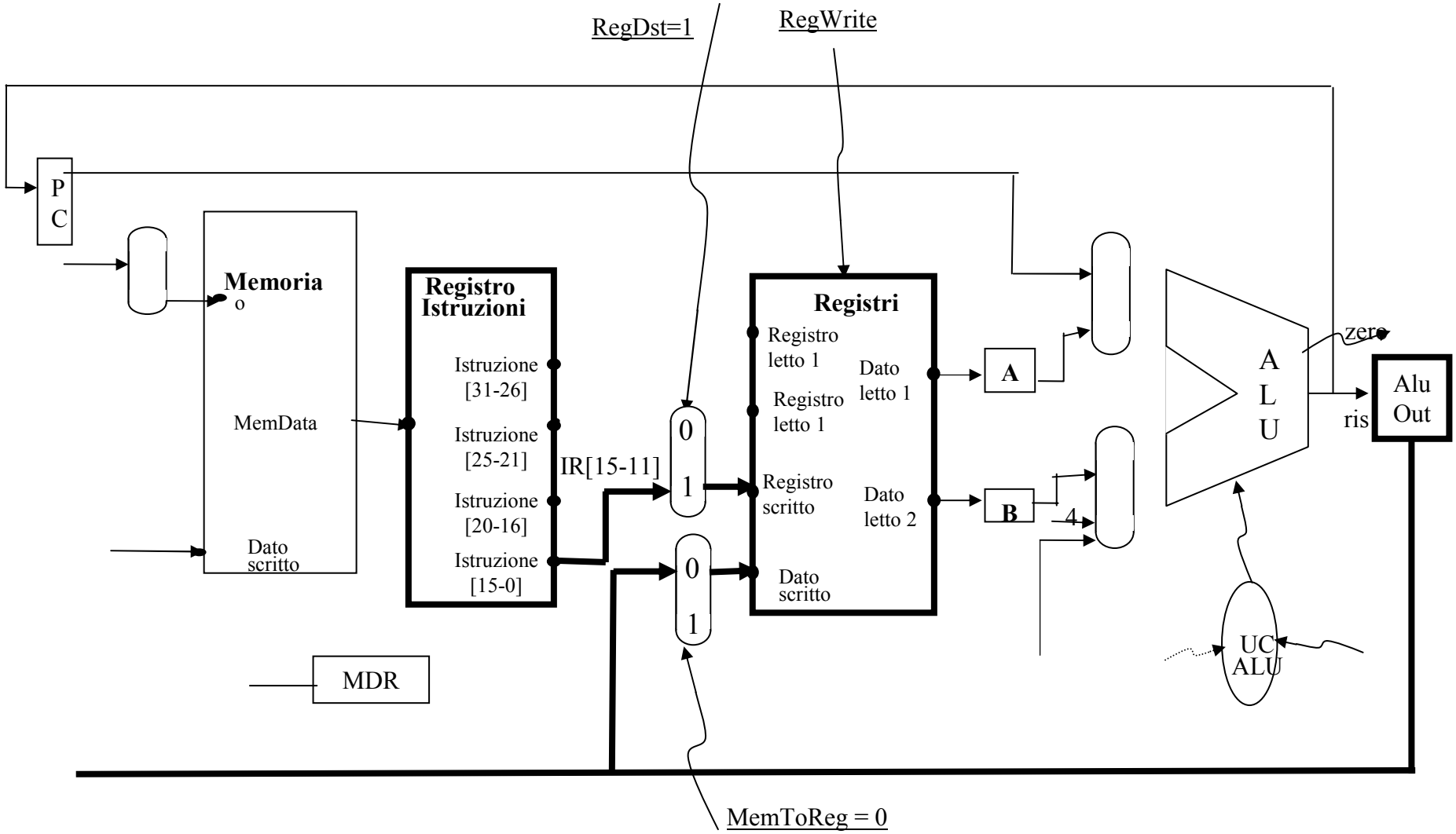
memoria [AluOut] ← B (NB: B è stato letto due volte…)



A questo punto l'istruzione è terminata!

IV ciclo: completamento istruzioni R

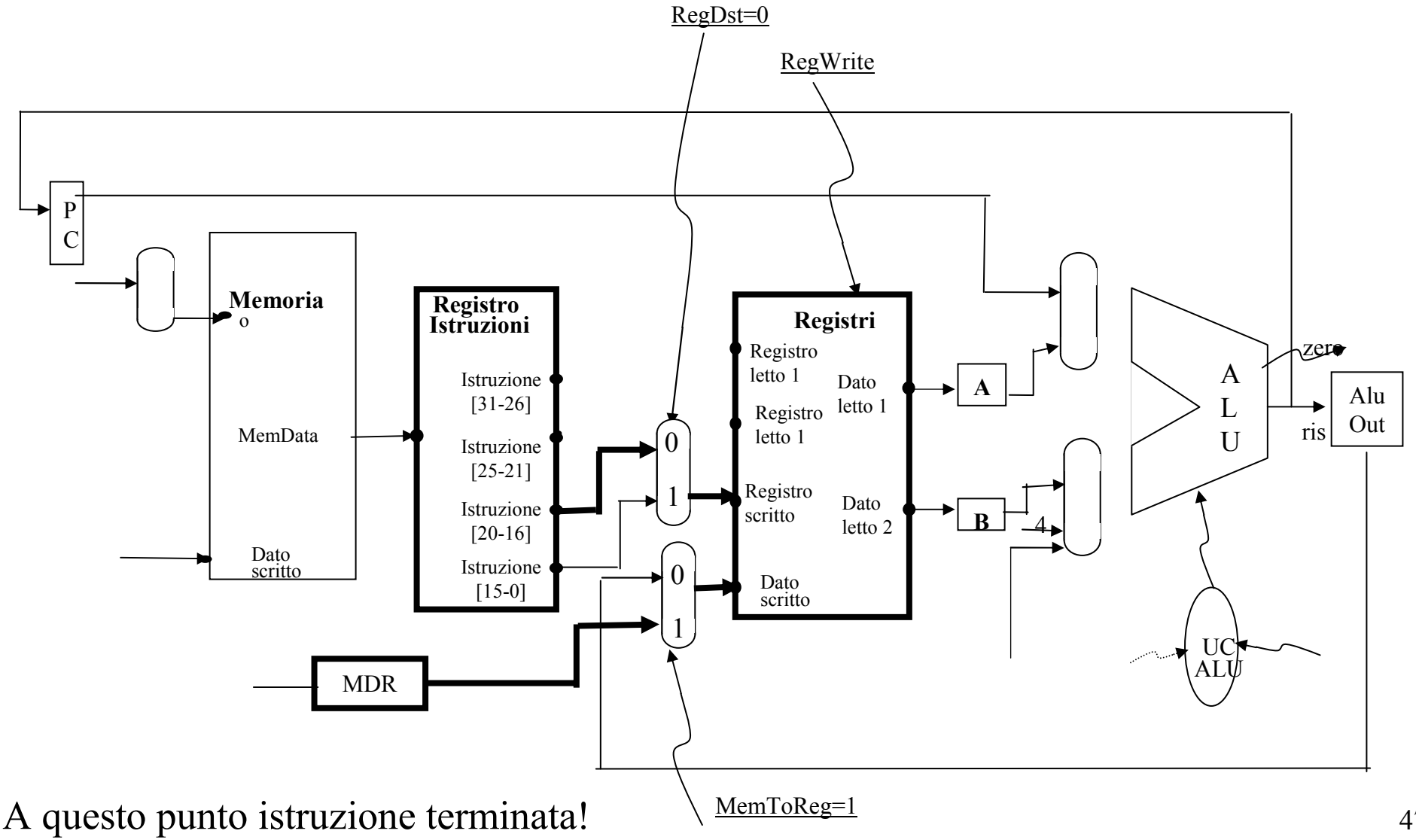
Reg [IR[15-11]] ← Aluout



A questo punto l'istruzione è terminata!

V ciclo: completamento lettura da memoria (lw)

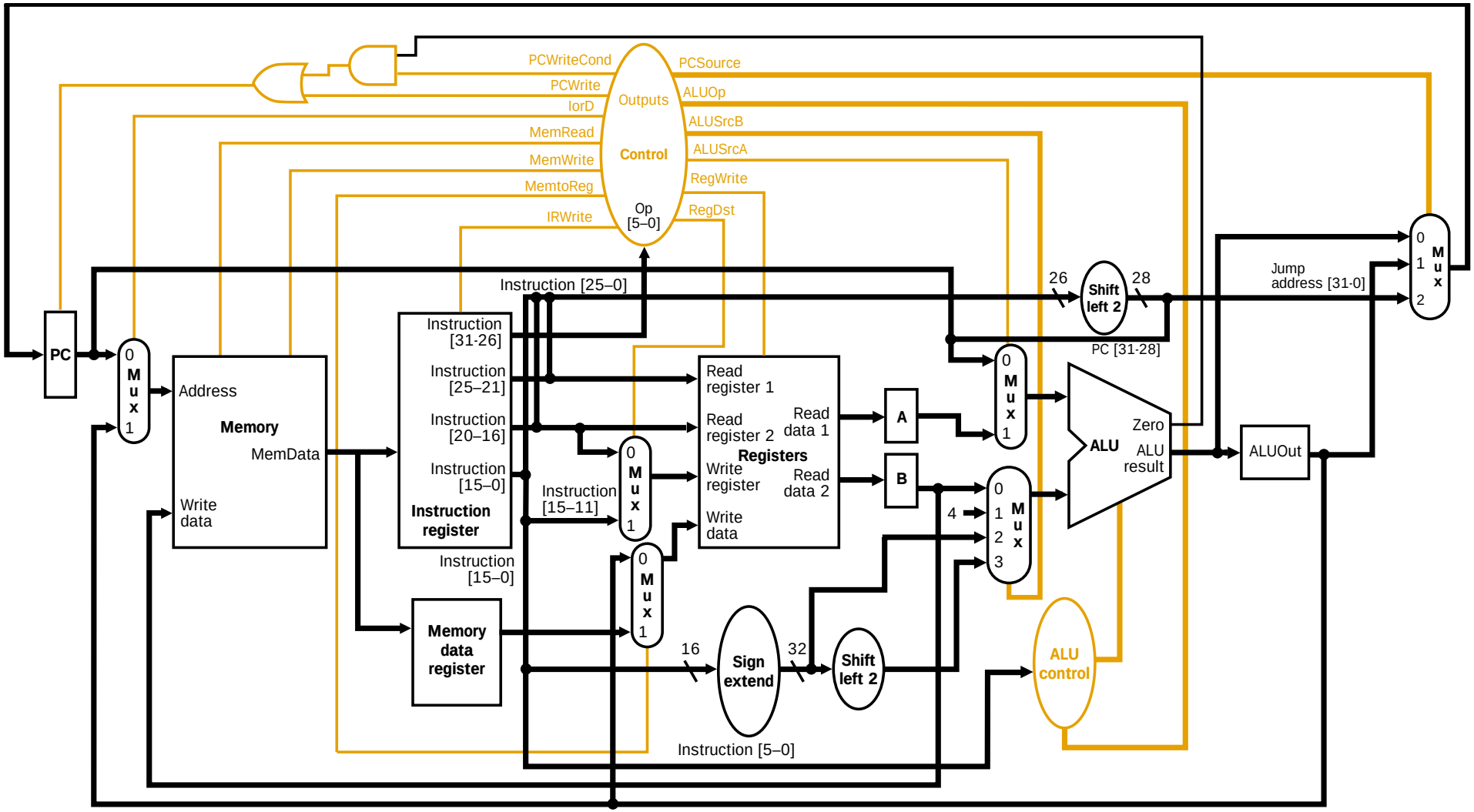
Reg [IR[20-16]] ← MDR



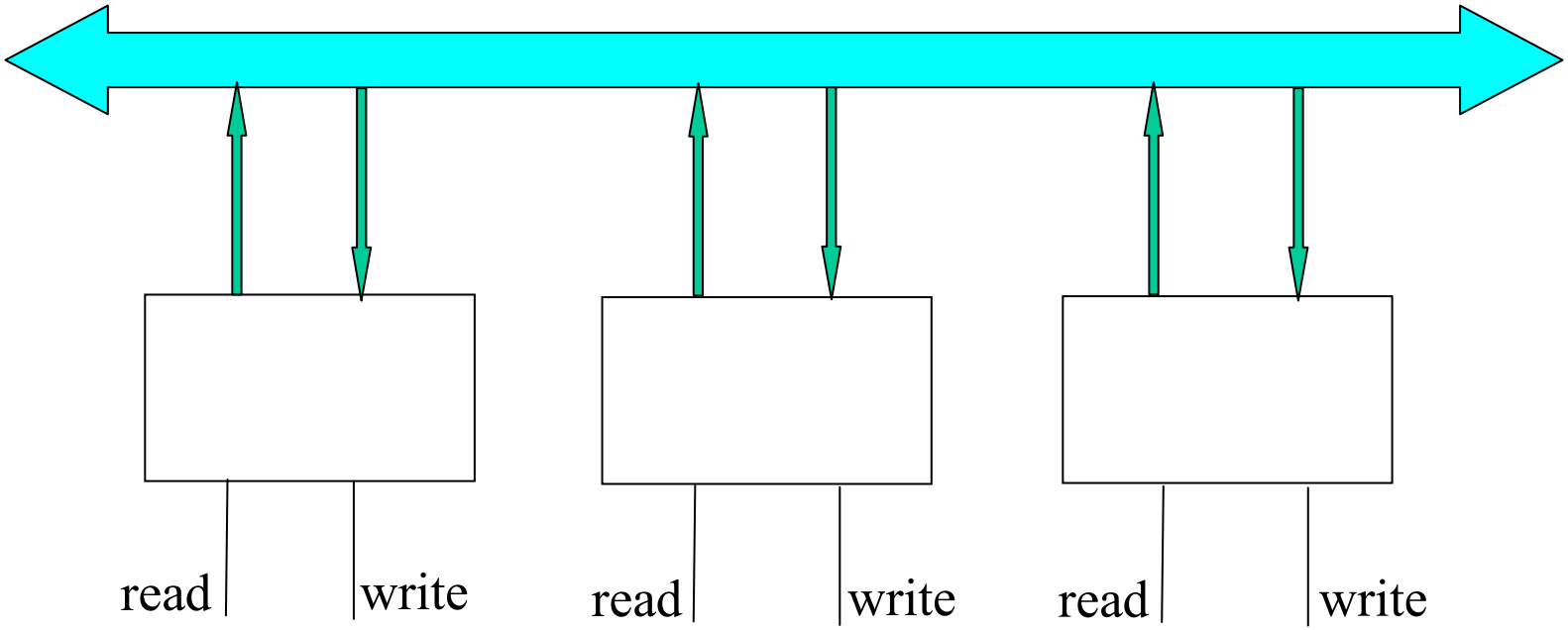
A questo punto istruzione terminata!

Da queste considerazioni risultano immediatamente:

- unità di elaborazione (datapath)
- specifica dell'unità di controllo



Soluzione alternativa (permette di risparmiare connessioni):
bus condiviso interno alla CPU



Permette di risparmiare delle linee di dato, ma comporta una riduzione delle prestazioni, perché un bus è meno veloce di una connessione punto-punto

→ Nel seguito, non considereremo questa soluzione
(si parla di bus interno alla CPU!)

Specifica dell'unità di controllo:

1) Come macchina a stati finiti (Modello di Moore)

2) Come microprogramma

Partiamo dalla prima modalità, ottenendo direttamente il diagramma a stati…

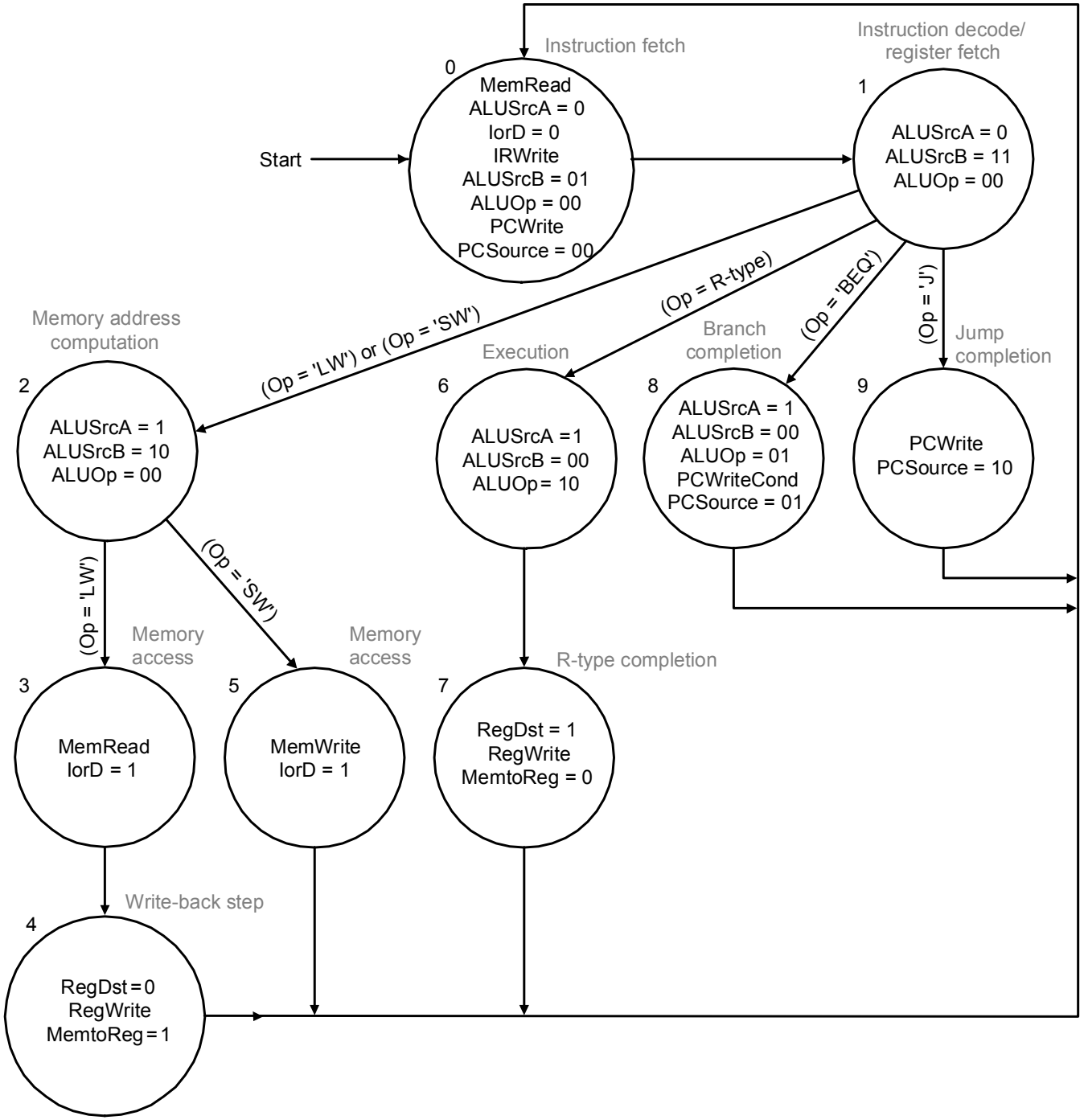
Ad ogni stato sono associate le uscite corrispondenti (cfr. lucidi precedenti):

- segnali di scrittura/lettura:

  non indicati $\Rightarrow$ 0

- segnali di controllo MUX:

  non indicati $\Rightarrow$ valore indifferente


Instruction fetch
Instruction decode/
register fetch
0
MemRead
ALUSrcA = 0
IorD = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
1
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address
computation
Execution
Branch
completion
Jump
completion
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
6
ALUSrcA =1
ALUSrcB = 00
ALUOp= 10
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
R-type completion
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
7
RegDst = 1
RegWrite
MemtoReg = 0
Write-back step
4
RegDst = 0
RegWrite
MemtoReg = 1

Come può essere realizzata la macchina corrispondente a questa specifica?

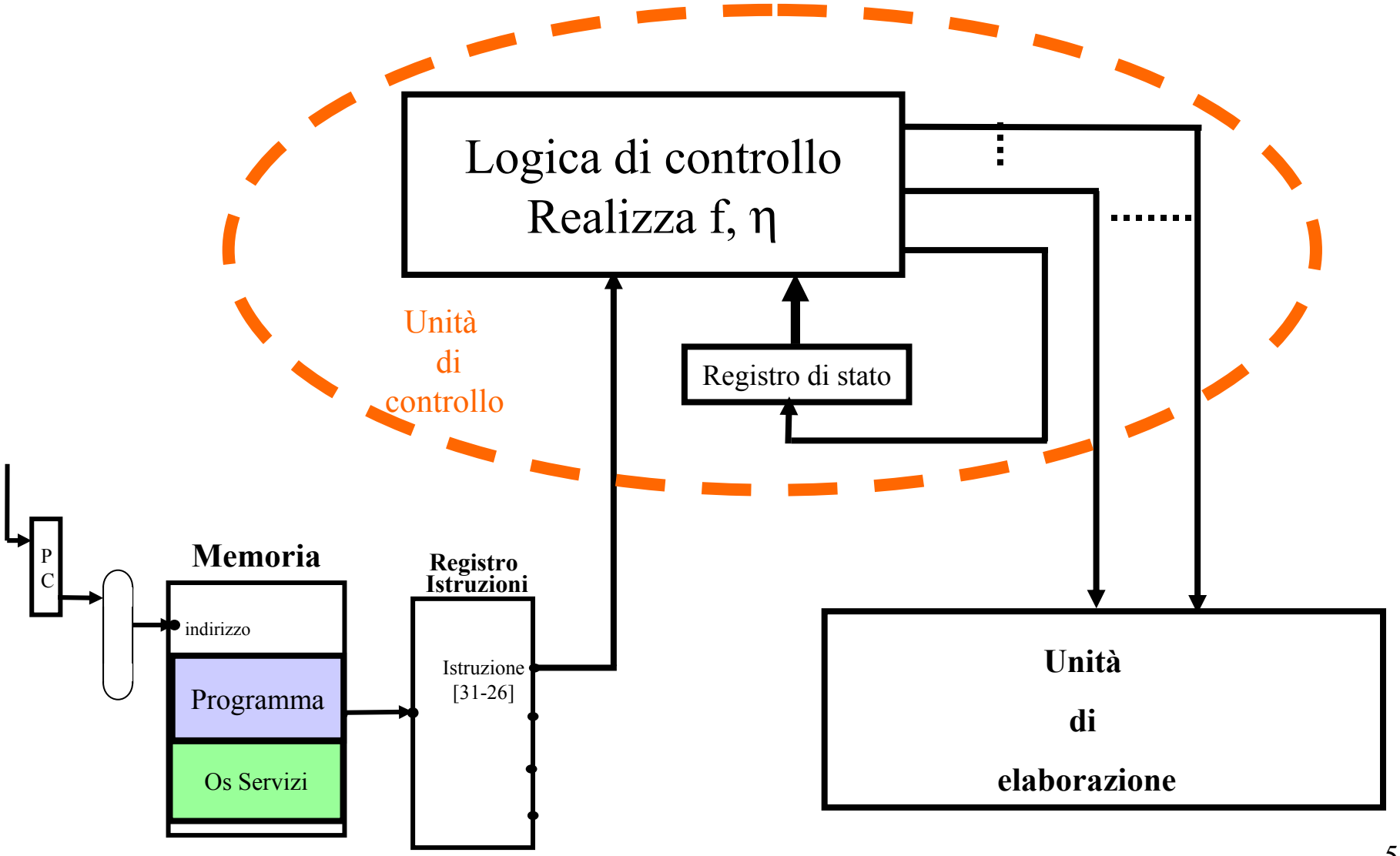
1) Con una macchina a stati finiti

2) Con tecnica di controllo microprogrammato

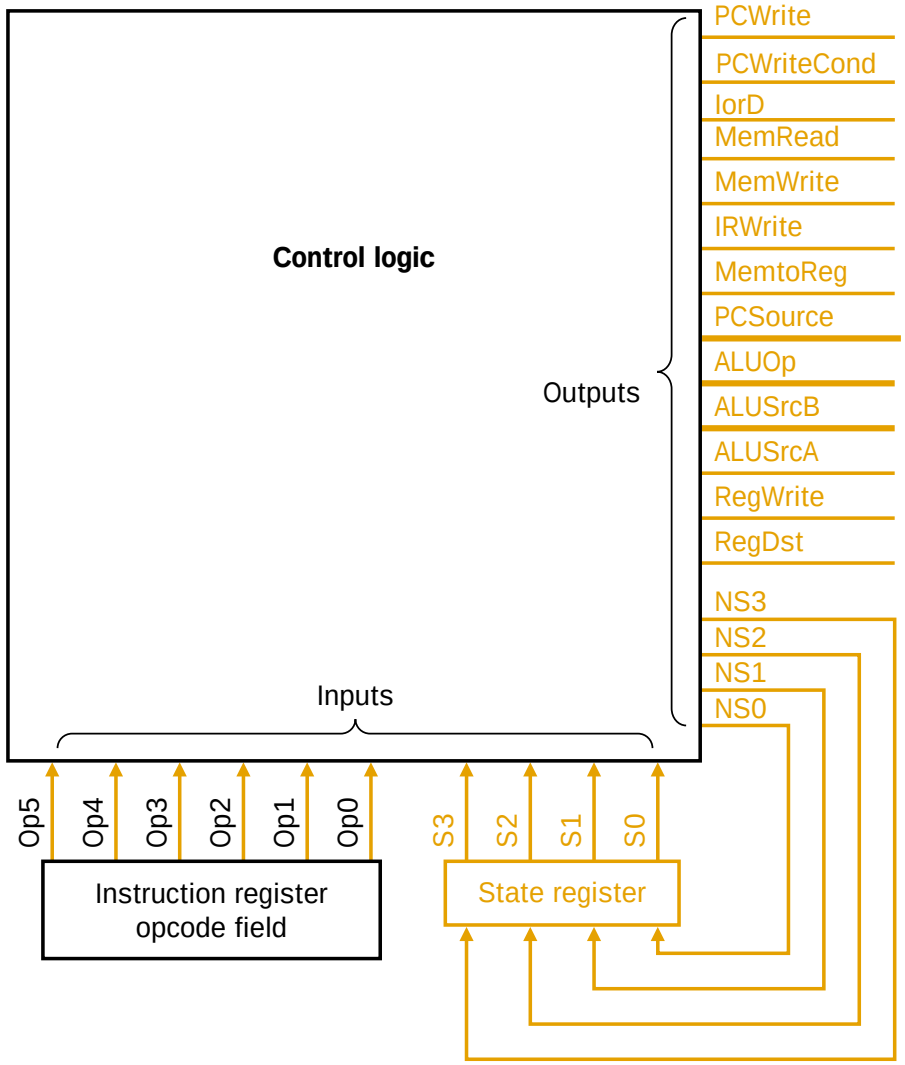
Vediamo la prima modalità di realizzazione…

Controllo con macchina a stati finiti

Durante l'esecuzione di un programma applicativo i circuiti interpretano le istruzioni:
del programma costituito dal< programma applicativo ∘ i servizi OS>



- Automa con 10 stati ⇒ 4 bit per codificarli (S3-S0)
  memorizzati con 4 flip-flop di tipo D
- Codifica: assegnamento di un valore a ciascuno stato (useremo numero progressivo)



Dobbiamo realizzare due funzioni logiche combinatorie:

- funzione di uscita $\eta(\underline{s})$
  restituisce 16 segnali di uscita
- funzione di stato futuro $f(\underline{s}, \underline{I})$
  restituisce NS3…NS0

dove $\underline{s} = (S3, S2, S1, S0)$

$\underline{I} = (OP5, \ldots, OP0)$

Tabelle di verità per le **uscite di controllo** (dipendono solo dallo stato):
per semplificarle vengono indicati solo gli stati in cui singole uscite = 1.

PCSource (PCSource1 – PCSource0)

Dal diagramma di stato, si vede che:
- PCSource1 vale 1 nello stato 9
- PCSource0 vale 1 nello stato 8

S3	S2	S1	S0	PCSource1
1	0	0	1	1

S3	S2	S1	S0	PCSource0
1	0	0	0	1

⇒ Equazioni logiche: $PCSource1 = S3\ \overline{S2}\ \overline{S1}\ S0$

$PCSource0 = S3\ \overline{S2}\ \overline{S1}\ \overline{S0}$

IorD

Dal diagramma di stato, si vede che
vale 1 negli stati 3 e 5

⇒ $IorD = \overline{S3}\ \overline{S2}\ S1\ S0 + \overline{S3}\ S2\ \overline{S1}\ S0$

S3	S2	S1	S0	IorD
0	0	1	1	1
0	1	0	1	1

Altre uscite ricavate con considerazioni analoghe…

NB: campi Op don't care (1 riga sta per $2^6 = 64$ righe!)

s3	s2	s1	s0
0	0	0	0
1	0	0	1

a. Tabella di verità per PCWrite

s3	s2	s1	s0
1	0	0	0

b. Tabella di verità per PCWriteCond

s3	s2	s1	s0
0	0	1	1
0	1	0	1

c. Tabella di verità per IorD

s3	s2	s1	s0
0	0	0	0
0	0	1	1

d. Tabella di verità per MemRead

s3	s2	s1	s0
0	1	0	1

e. Tabella di verità per MemWrite

s3	s2	s1	s0
0	0	0	0

f. Tabella di verità per IRWrite

s3	s2	s1	s0
0	1	0	0

g. Tabella di verità per MemtoReg

s3	s2	s1	s0
1	0	0	1

h. Tabella di verità per PCSource1

s3	s2	s1	s0
1	0	0	0

i. Tabella di verità per PCSource0

s3	s2	s1	s0
0	1	1	0

j. Tabella di verità per ALUOp1

s3	s2	s1	s0
1	0	0	0

k. Tabella di verità per ALUOp0

s3	s2	s1	s0
0	0	0	1
0	0	1	0

l. Tabella di verità per ALUSrcB1

s3	s2	s1	s0
0	0	0	0
0	0	0	1

m. Tabella di verità per ALUSrcB0

s3	s2	s1	s0
0	0	1	0
0	1	1	0
1	0	0	0

n. Tabella di verità per ALUSrcA

s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Tabella di verità per RegWrite

s3	s2	s1	s0
0	0	0	0

p. Tabella di verità per RegDst

Tabelle di verità per le **uscite di stato futuro** (dipendono da stato e campi Op)

NS0: pari a 1 quando stato futuro = 1, 3, 5, 7, 9

Dall'analisi del diagramma di stato, si vede che ad essi si può arrivare:

stato 1 : da stato 0 a prescindere da Op
stato 3 : da stato 2 con Op = "lw" [100011]
stato 5: da stato 2 con Op = "sw" [101011]
stato 7: da stato 6 a prescindere da Op
stato 9: da stato 1 con Op = "j" [000010]

Ciò corrisponde alla tabella di verità:

OP5	OP4	OP3	OP2	OP1	OP0	S3	S2	S1	S0	NS0
X	X	X	X	X	X	0	0	0	0	1
1	0	0	0	1	1	0	0	1	0	1
1	0	1	0	1	1	0	0	1	0	1
X	X	X	X	X	X	0	1	1	0	1
0	0	0	0	1	0	0	0	0	1	1

$$NS0 = \overline{S3}\ \overline{S2}\ \overline{S1}\ \overline{S0} + OP5\ \overline{OP4}\ \overline{OP2}\ OP1\ OP0\ \overline{S3}\ \overline{S2}\ S1\ \overline{S0} + \overline{S3}\ S2\ S1\ \overline{S0} + \overline{OP5}\ \overline{OP4}\ \overline{OP3}\ \overline{OP2}\ OP1\ \overline{OP0}\ \overline{S3}\ \overline{S2}\ \overline{S1}\ S0$$

Con considerazioni analoghe si ricavano le altre uscite di stato futuro

⇒ Abbiamo la specifica di entrambe le funzioni combinatorie

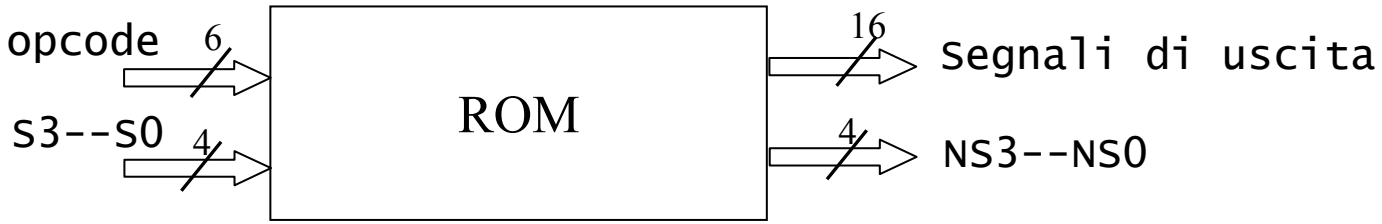
Realizzazione in 2 (*2) modi alternativi:

1) Tramite ROM
   - unica ROM per entrambe le funzioni
   - due ROM separate

2) Tramite PLA
   - unica PLA
   - due PLA separate

Confrontiamo le varie soluzioni

Unica ROM



Dimensione: $\mathbf{2^{10} * 20 = 20\ Kbit}$

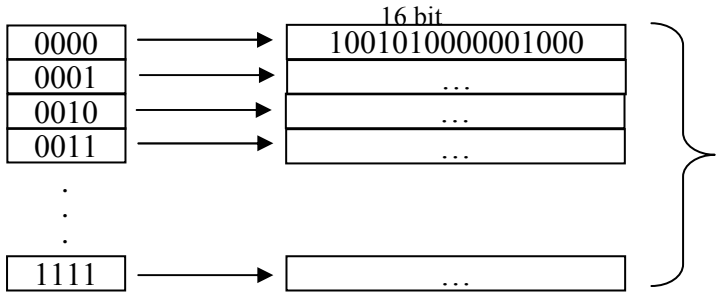
Parola di uscita

Combinazioni possibili dei 10 valori di ingresso

• Definendo un ordine dei bit di ingresso e di uscita, è possibile specificare il contenuto della ROM. Risulta che:

- 10 bit di ingresso (stato + opcode) → 4 bit di stato futuro

- 4 bit ingresso (stato) → 16 bit di controllo

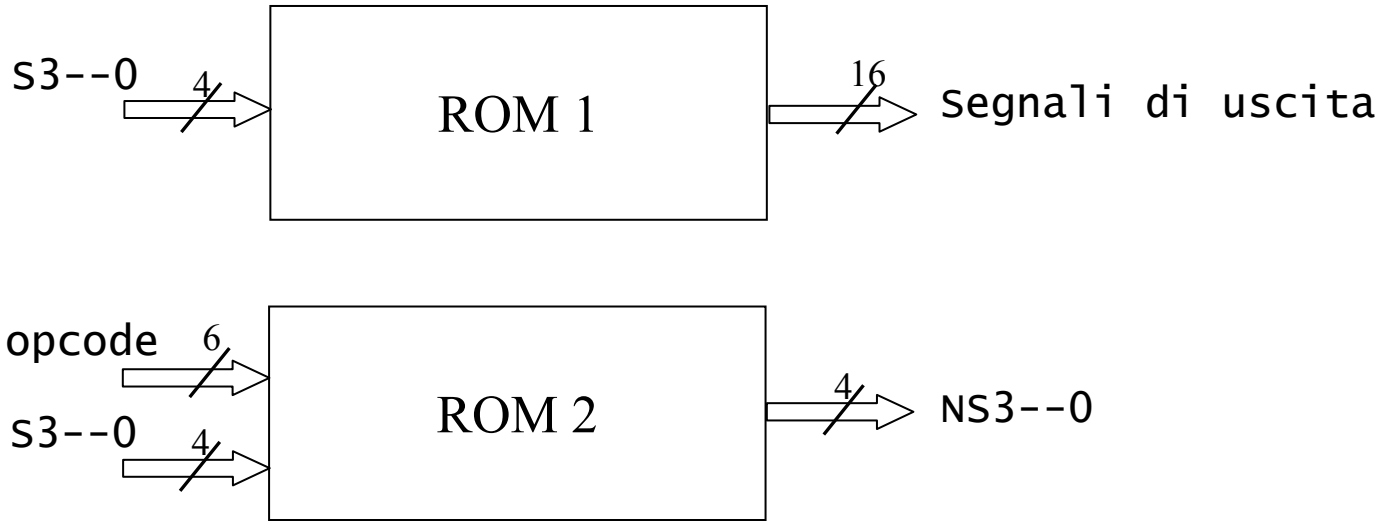


2^4 parole (di 16 bit)
ciascuna ripetuta 2^6 volte
(6 opcode don't care)

*ovvero: 2^4*2^6*16 al posto di 2^4*16*

Possiamo evitare lo spreco dovuto alle ripetizioni per 2^6 usando due ROM separate (stato futuro e uscite controllo):
la ROM per le uscite di controllo ha solo 4 ingressi (bit di stato) e quindi evita di ripetere la stessa parola di memoria per tutte le 2^6 combinazioni degli ingressi!

Due ROM separate



Dimensione: **2^4 * 16 + 2^{10} * 4 = 4,3 Kbit**

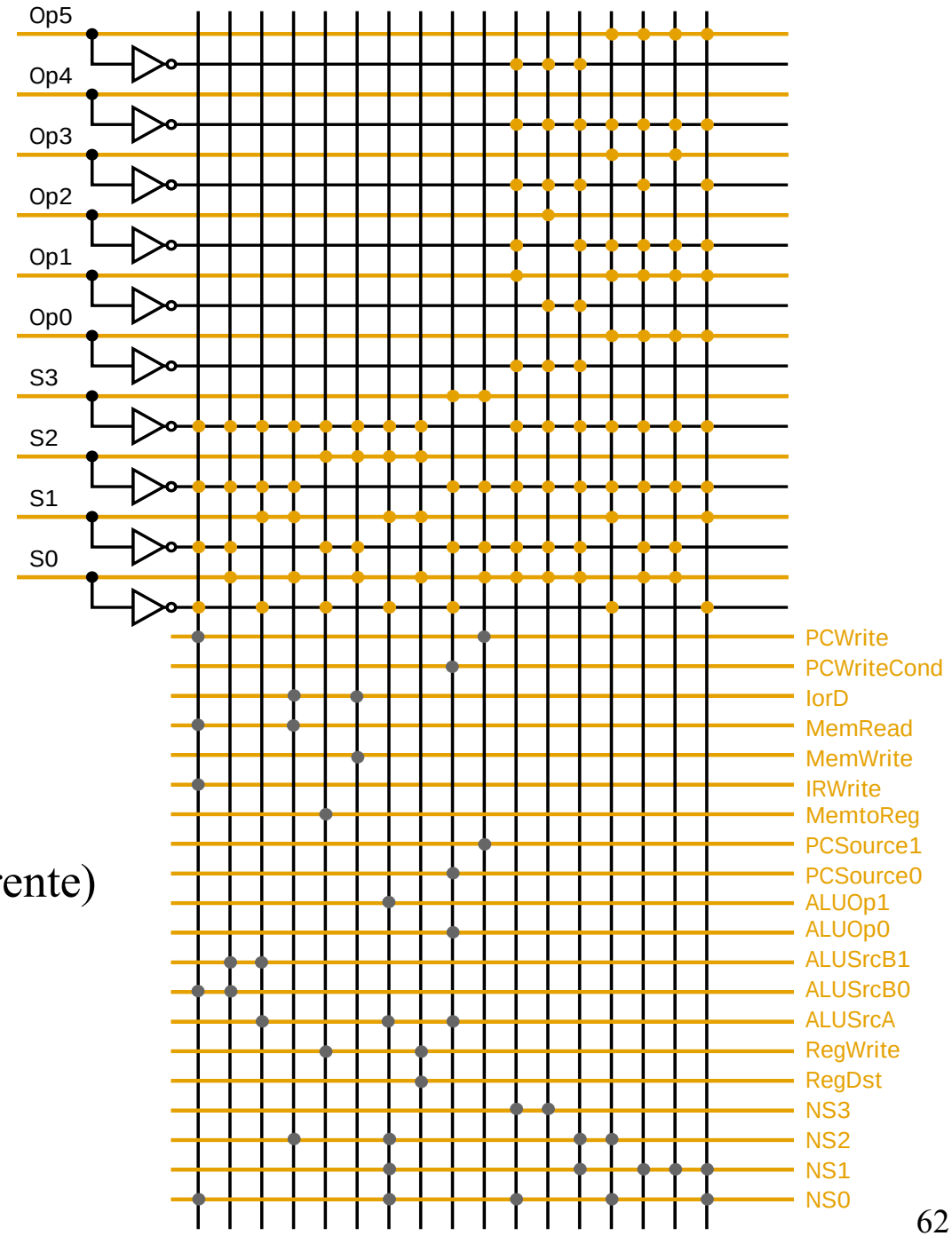
Nelle ROM ci sono comunque molti elementi duplicati:

- molte combinazioni dell'ingresso non si verificano (ROM le codifica tutte)
  p.es. ho 4 bit per 10 stati: 16-10=6 combinazioni non si verificano
- a volte le uscite non dipendono da [tutti] i valori dell'ingresso.
  p.es. da stato 0 si passa sempre a stato 1 per qualunque valore di opcode, mentre ROM 2 duplica la parola NS1 (0001) per 2^6 volte
- inoltre, ROM è "completamente codificata"

Unico PLA

Ciascuna uscita è OR di termini prodotto (mintermini)

- Rappresentate solo le righe della tabella di verità con uscita 1
- Ciascun mintermine può rappresentare più righe (input don't care)



In questo caso: 17 mintermini
(10 dipendono solo da stato corrente)

Dimensione: proporzionale a
10*17 + 17*20 = 510

Due PLA

1) PLA a 4 ingressi (stato corrente) e 16 uscite di controllo

10 mintermini (quelli che dipendevano solo da stato corrente, usati dalle uscite di controllo)

Dimensione: 4*10 + 10*16 = 200

2) PLA a 10 ingressi (stato e opcode) e 4 uscite di stato futuro

10 mintermini (cfr. figura del PLA: sono quelli usati da uscite di stato futuro, dei quali 7 dipendono da stato e opcode)

Dimensione: 10*10 + 10*4 = 140

⇒ Dimensione totale: proporzionale a

200 + 140 = 340

Riepilogo

	1	2	Riduzione a
ROM	20 Kbit	4.3 Kbit	1/5
PLA	510	340	3/5
Riduzione a	1/40	1/13	

FINORA ABBIAMO VISTO:

- Processore a singolo ciclo [Unità di controllo combinatoria]
  - specifica e implementazione: cfr. reti combinatorie
- Processore multi-ciclo [Unità di controllo sequenziale]
  - **specifica come FSM** e **implementazione come FSM**

    ↓

    Parti combinatorie:
    - specifica con tab. di verità | equazioni logiche
    - realizzazione con ROM |PLA (singole o doppie)

  - ORA VEDIAMO LA **SPECIFICA COME microprogramma**

Il controllo specificato come microprogramma

• Nella pratica, set istruzioni più numeroso o istruzioni più complesse (cfr. Intel)

⇒ drastico aumento del numero degli stati

⇒ impraticabile rappresentare gli stati esplicitamente!

• Molti di questi stati raggiungibili in modo sequenziale

(es. precedente: da stato_0 si passa sempre a stato_1 a prescindere da ingressi)

⇒ idea: utilizzare i concetti della programmazione

- Microistruzioni "rappresentano" segnali al datapath in un ciclo di clock
- Controllo del processore rappresentato da un microprogramma
  (microistruzioni eseguite in sequenza + salti)

VANTAGGI

• Nel caso di molti stati, porta ad una significativa semplificazione nella specifica del controllo!

• Inoltre, come vedremo questa specifica ha una "naturale" realizzazione che può essere più economica di una FSM in diversi casi

Nota bene:

Abbiamo:

un particolare datapath

ISA (che include il formato delle istruzioni macchina)

Dobbiamo specificare l'unità di controllo!

Solo dopo verrà illustrata la realizzazione.

Passi necessari:

1) **Definire Formato** (rappresentazione simbolica) delle **microistruzioni**:
   - Sequenza di campi distinti
   - Valori (simbolici) che ogni campo può assumere
   - Segnali di controllo associati a ciascun campo

   Obiettivi:
   - leggibilità del microprogramma
     ⇒ ciascun campo associato ad una funzione specifica
     [es. 1 campo per operaz. ALU, 3 per scelta sorgenti e destinazione]
   - impedire scrittura microistruzioni incoerenti / istruzioni non supportate
     ⇒ campi diversi per insiemi disgiunti di segnali di controllo;

Nome del campo	Valore del campo	Funzione del campo con tale valore
Label	Una stringa qualunque	Questo campo è utilizzato per specificare le etichette utilizzate per controllare la sequenza delle microistruzioni. Le etichette che terminano con 1 o 2 sono utilizzate durante gli smistamenti tramite tabelle di salto indicizzate sulla base del codice operativo; le altre etichette sono utilizzate come destinazioni dirette per i salti tra microistruzioni. Le etichette non generano direttamente segnali di controllo, ma sono utilizzate per definire il contenuto delle tabelle di smistamento e per generare i segnali di controllo relativi al campo Sequencing.
ALU control	`Add`	Indica alla ALU di sommare.
	`Subt`	Indica alla ALU di sottrarre; è utilizzato nell'implementazione del confronto per i salti condizionati.
	`Func code`	Utilizza il campo funct dell'istruzione per determinare i segnali di controllo della ALU.
SRC1	`PC`	PC è il primo ingresso della ALU.
	`A`	Il registro A è il primo ingresso della ALU.
SRC2	`B`	Il registro B è il secondo ingresso della ALU.
	`4`	Il valore 4 va sul secondo ingresso della ALU.
	`Extend`	L'uscita dell'unità di estensione del segno va sul secondo ingresso della ALU.
	`Extshft`	L'uscita dell'unità di scalamento va sul secondo ingresso della ALU.
Register control	`Read`	Legge due registri utilizzando i campi rs e rt di IR come numeri di registro, ponendo i dati letti nei registri A e B.
	`Write ALU`	Scrive nel register file utilizzando il campo rt di IR come numero di registro ed il contenuto di ALUOut come dato.
	`Write MDR`	Scrive nel register file utilizzando il campo rt di IR come numero di registro ed il contenuto di MDR come dato.
Memory	`Read PC`	Legge la memoria utilizzando PC come indirizzo; il risultato è scritto in IR (e nel MDR).
	`Read ALU`	Legge la memoria utilizzando ALUOut come indirizzo; il risultato è scritto in MDR.
	`Write ALU`	Scrive nella memoria utilizzando ALUOut come indirizzo; il dato è contenuto in B.
PCWrite control	`ALU`	Scrive l'uscita della ALU in PC
	`ALUOut-cond`	Se l'uscita Zero della ALU è attiva, scrive in PC il contenuto del registro ALUOut.
	`Jump address`	Scrive in PC l'indirizzo di salto ricavato dall'istruzione.
Sequencing	`Seq`	Seleziona sequenzialmente la microistruzione successiva.
	`Fetch`	Torna alla prima microistruzione, che inizia l'esecuzione di una nuova istruzione.
	`Dispatch i`	Esegue uno smistamento utilizzando la ROM specificata da `i` (1 o 2).

Notare che:

• In *Register control* sono previsti due possibili valori per la scrittura dei registri:

- Write ALU: scrive nel registro *rd* il contenuto di *ALUout*
  [usato nelle istruzioni di Tipo-R]
- Write MDR: scrive nel registro *rt* il contenuto di *MDR*
  [usato nell'istruzione lw]

Sono quindi escluse le combinazioni:

- scrittura in *rt* del contenuto di *ALUout*
- scrittura in *rd* del contenuto di *MDR*

possibili con il datapath progettato ma non usate in istruzioni MIPS considerate.

• In *Memory* i valori previsti non permettono ad esempio di:

- leggere la memoria indirizzata da *ALUout* e scrivere il valore letto in *IR*
- scrivere in memoria (il dato contenuto in *B*) nella locazione indirizzata da PC

A ciascun campo (a parte Sequencing) possiamo già associare segnali di controllo, il cui valore è determinato dal valore simbolico del campo stesso
(cfr. Assembler vs. Linguaggio Macchina)

Nome del campo	Valore	Segnali attivi	Commenti
Controllo della ALU	Add	ALUOp=00	Forza la ALU ad eseguire la somma.
	Subt	ALUOp=01	Forza la ALU ad eseguire la sottrazione; si implementa così anche il confronto per i salti.
	Codice func	ALUOp=10	Usa il codice funzione dell'istruzione per determinare il controllo della ALU.
SRC1	PC	ALUSrcA=0	Usa il PC come primo ingresso della ALU.
	A	ALUSrcA=1	Il registro A è il primo ingresso della ALU.
SRC2	B	ALUSrcB=00	Il registro B è il secondo ingresso della ALU.
	4	ALUSrcB=01	Si usa 4 come secondo ingresso della ALU.
	Extend	ALUSrcB=10	Si usa l'uscita dell'unità di estensione del segno come secondo ingresso della ALU.
	Extshft	ALUSrcB=11	Si usa l'uscita dell'unità di scalamento di due posizioni come secondo ingresso della ALU.
Controllo del registro	Lettura		Si leggono due registri usando i campi rs e rt di IR come numeri dei registri e si mette il dato nei registri A e B.
	Scrittura ALU	RegWrite, RegDst=1, MemtoReg=0	Si scrive un registro usando il campo rd di IR come numero del registro e il contenuto di ALUOut come dato.
	Scrittura MDR	RegWrite, RegDst=0, MemtoReg=1	Si scrive un registro usando il campo rt di IR come numero del registro e il contenuto di MDR come dato.
Memoria	Lettura PC	MemRead, IorD=0 IRWrite	Si legge la memoria usando PC come indirizzo; si scrive il risultato in IR (e in MDR).
	Lettura ALU	MemRead, IorD=1	Si legge la memoria usando ALUOut come indirizzo; si scrive il risultato in MDR.
	Scrittura ALU	MemWrite, IorD=1	Si scrive in memoria usando ALUOut come indirizzo, e il contenuto di B come dato.
Controllo della scrittura di PC	ALU	PCSource=00, PCWrite	Si scrive l'uscita della ALU in PC.
	ALUOut-cond	PCSource=01, PCWriteCond	Se è attiva l'uscita Zero della ALU si scrive in PC il contenuto del registro ALUOut.
	Indirizzo di salto	PCSource=10, PCWrite	Si scrive in PC l'indirizzo di salto contenuto nell'istruzione.

Comandi non specificati:

- a elementi di memoria (lettura e scrittura):
  = 0
- a MUX e controllo elementi combinatori (ALU):
  = don't care

2) **Creare Microprogramma**: rappresentazione simbolica del controllo traducibile automaticamente in circuiti logici di controllo.

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

NB: Si vede che il microprogramma corrisponde esattamente al diagramma degli stati individuato nella specifica del processore come FSM.

Ciò fa già capire come la realizzazione possa essere effettuata mediante:

- *sequenzializzatore*
- *macchina a stati finiti!*

Note:

Nel microprogramma, le microistruzioni sono disposte in sequenza.

Ciascuna istruzione può essere etichettata da una label (campo *Label*)
che serve ad identificarla nei salti.

Scelta della microistruzione successiva dipende da campo *Sequencing*:
- Seq: eseguita l'istruzione successiva nel microprogramma
- Fetch: eseguita la microistruzione di fetch, etichettata con label Fetch,
  che di fatto corrisponde all'esecuzione di una nuova istruzione MIPS
- Dispatch i: l'istruzione successiva viene scelta, sulla base degli ingressi
  (campo opcode di IR), entro la tabella di smistamento i.

Ciascuna tabella contiene la corrispondenza
ingresso (campo opcode) → label della istruzione cui saltare

Le label nel campo *Label* terminano con un numero *i* indicante
la tabella di smistamento in cui sono contenute (vedi microprogramma).

Oltre al microprogramma, dobbiamo definire le tabelle di smistamento (forma simbolica)

- sulla base di ciascun possibile valore di opcode, a quale istruzione si salta

Tabella di smistamento del microcodice 1

Campo codice operativo	Nome del codice operativo	Valore
000000	Formato-R	Rformat1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

Tabella di smistamento del microcodice 2

Campo codice operativo	Nome del codice operativo	Valore
100011	lw	LW2
101011	sw	SW2

Con ciò la specifica è completa:

- formato microistruzioni
- associazione valori campi di controllo - segnali controllo a datapath
- microprogramma
- tabelle di smistamento

} Cfr. Ling. Macchina vs. Ling. Assembler

REALIZZAZIONE DELLA SPECIFICA CON MICROPROGRAMMA

• Dobbiamo decidere come realizzare:
- funzione di sequenzializzazione (determina la sequenza di computazione)
- funzione di controllo (determina le uscite di controllo Data Path)

• Esistono due modalità di realizzazione:
- come macchina a stati finiti:

Funzione di controllo = funzione di uscita [con PLA o ROM]

Funzione di sequenzializzazione = funzione di stato futuro [con PLA o ROM]

$\Rightarrow$ funzione stato futuro codifica esplicitamente lo stato

NB: vedremo brevemente questa tecnica alla fine…

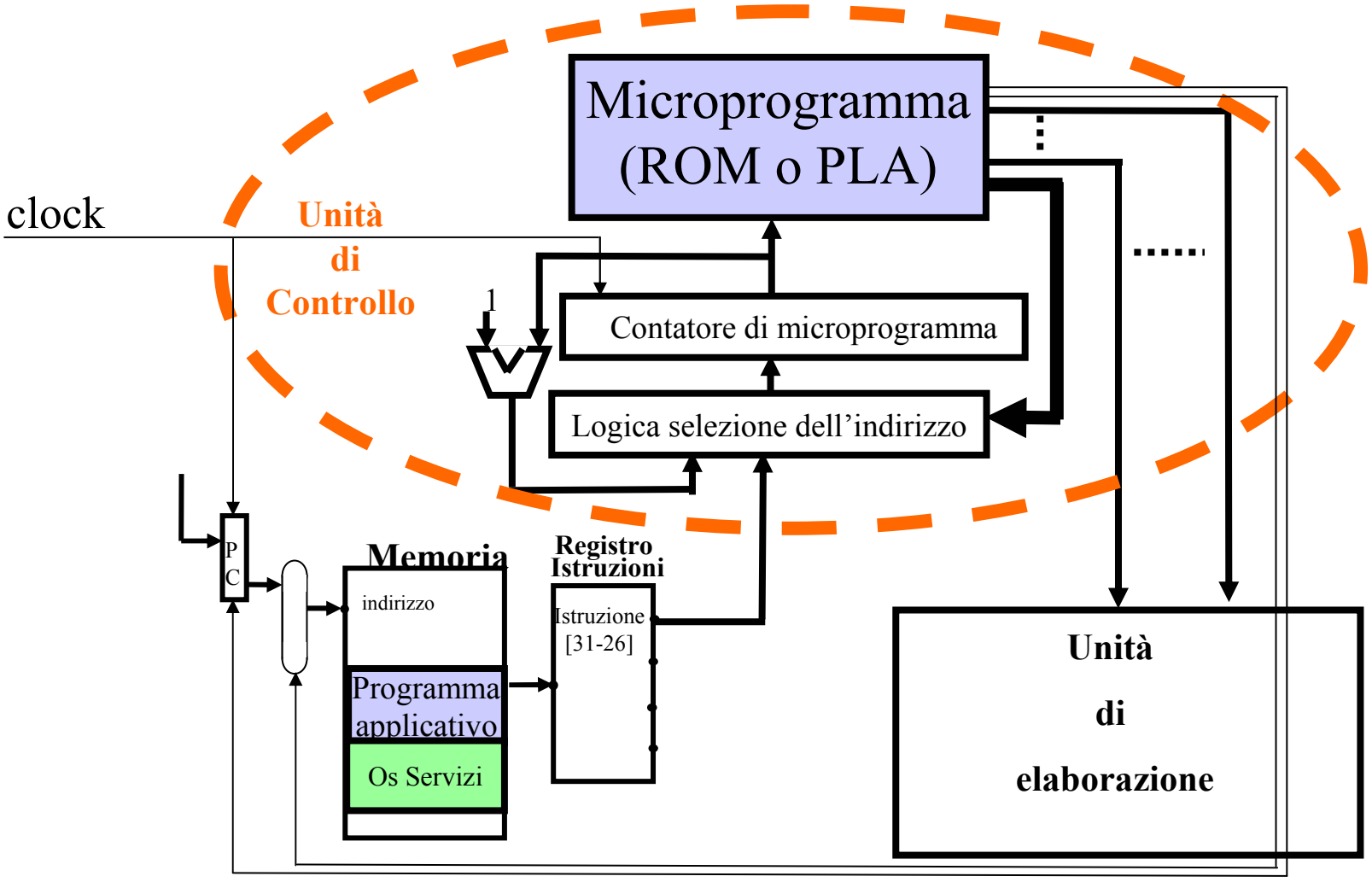
- Usando una ROM per memorizzare le microistruzioni (funzione di controllo) ed un contatore che fornisce in maniera sequenziale l'indirizzo microistruzione (stato futuro):

si elimina la codifica esplicita dello stato futuro all'interno dell'unità di controllo

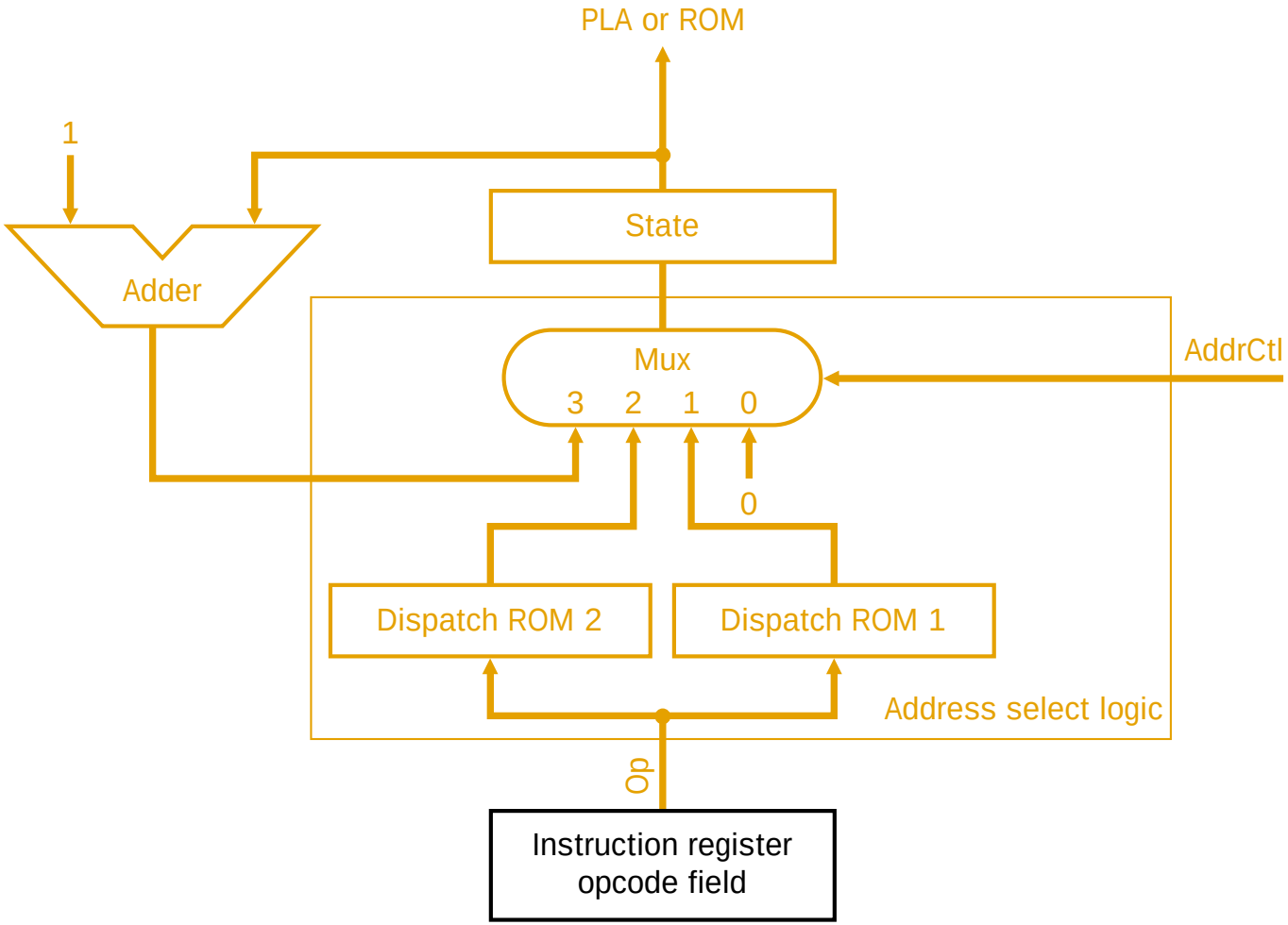
└ Controllo microprogrammato

Controllo Microprogrammato

Durante l'esecuzione di un programma applicativo i circuiti interpretano iterativamente il microprogramma sulle istruzioni del < programma applicativo ∘ i servizi OS> .



- Logica di selezione dell'indirizzo, sulla base del segnale di controllo proveniente da microistruzione corrente, sceglie l'indirizzo successivo tra:
  - stato (indirizzo) incrementato
  - stato 0 (di fetch)
  - stato determinato da tabelle di dispatch funzione di opcode
- Segnale di controllo AddrCtl a due bit (proveniente da microistruzione).

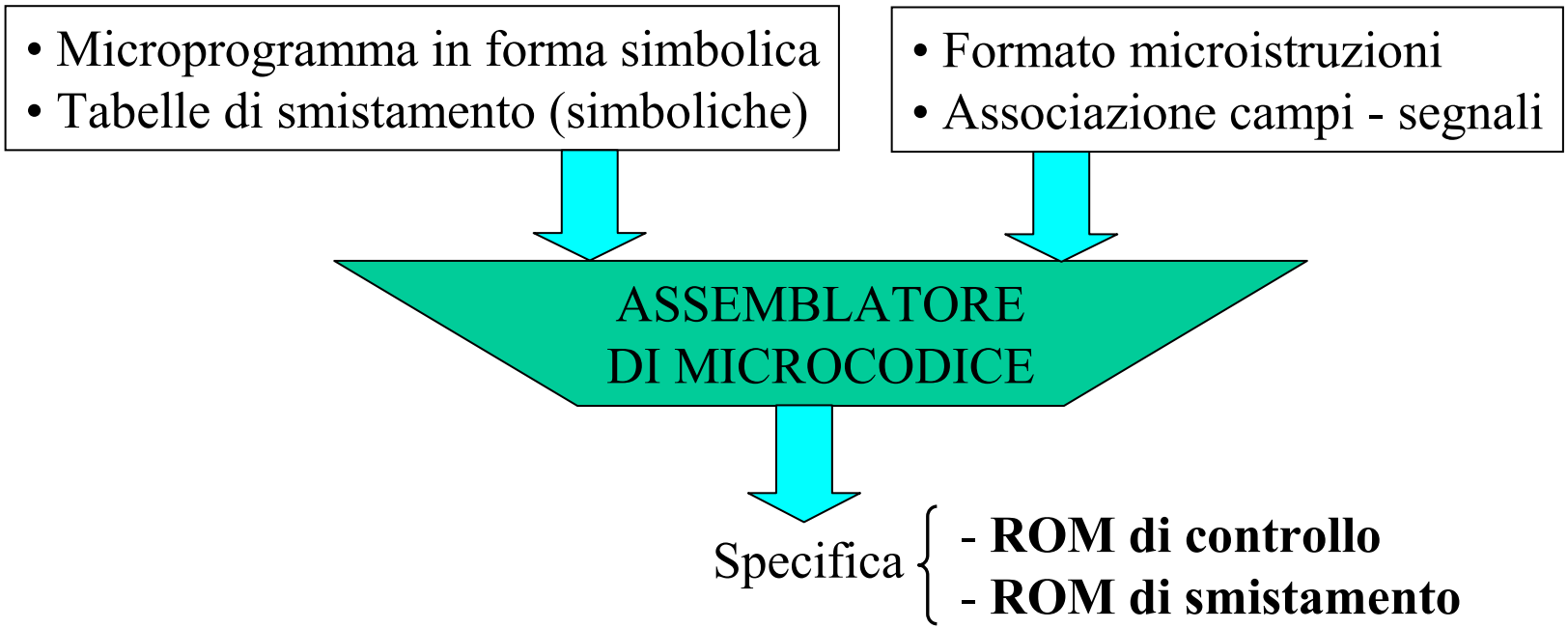


A questo punto possiamo associare i valori simbolici del campo Sequencing al segnale di controllo *AddCtl*:

- *Seq* *AddCtl* = 11
- *Fetch* *AddCtl* = 00
- *Dispatch 1* *AddCtl* = 01
- *Dispatch 2* *AddCtl* = 10

$\Rightarrow$ Abbiamo quindi la corrispondenza completa valori campi-segnali

Fase successiva: corrisponde alla "compilazione" del Linguaggio Assembler in LM



1) Traduzione dei campi di ogni microistruzione nei segnali corrisponenti:
da microprogramma + tabella di associazione campi-segnali

2) Assegnare indirizzi alle microistruzioni [mantenendo sequenzialità]

⇨ Specifica **ROM di controllo**: ogni parola corrisponde a una microistruzione

Mantenendo l'ordine nel microprogramma (coincidente con codifica stati FSM):

Numero dello stato	Bit 17-2 della parola di controllo	Bit 1-0 della parola di controllo
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000000	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

Bit 17-2: bit di controllo (NB: corrispondenti a ciascuno stato del diagramma)
Bit 1-0: bit di controllo indirizzo: associati a campo *Sequencing* della microistruzione

3) Specifica delle **ROM di Smistamento**

da tabelle di smistamento simboliche

\+ indirizzi assegnati alle istruzioni (Tabella label - indirizzo)

ROM di smistamento 1		
Op	**Nome del codice operativo**	**Valore**
000000	Formato-R	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

ROM di smistamento 2		
Op	**Nome del codice operativo**	**Valore**
100011	lw	0011
101011	sw	0101

NB: gli altri elementi delle ROM [ciascuna con 2^6 * 4 bit] sono vuote
in questo caso [perché si sono considerate solo poche istruzioni]

Controllo microprogrammato (realizzato con sequenzializzatore): valutazione

- Specifica come macchina a stati finiti: abbiamo visto che, nel caso delle 2 ROM:

$$\underbrace{2^4 * 16}_{\text{uscita}} + \underbrace{2^{10} * 4}_{\text{stato futuro}} = \underbrace{256}_{6\%} + \underbrace{4096}_{94\%}$$

- Istruzioni complesse (p.es. virgola mobile) sono caratterizzate da sequenze di stati indipendenti da ingressi (opcode); all'aumentare del set di istruzioni disponibili, queste sequenze diventano via via più frequenti
- Inoltre, complessità e numerosità delle istruzioni porta ad aumento numero stati totali

  ⇒ In una macchina a stati, la funzione di stato futuro deve comunque codificare tutte le transizioni stato-stato: anche per quelle indipendenti da ingressi lo stato destinazione deve essere esplicitamente codificato;
  in una tabella di verità, vi sarà una riga per ogni transizione [opcode don't care].

  Nel controllo microprogrammato, la funzione di controllo ha solo due bit in più per decidere la modalità di selezione: transizioni "sequenziali" non codificano esplicitamente lo stato destinazione, ma solo che bisogna scegliere il "seguente". Le ROM di smistamento memorizzano solo stati cui si arriva in dipendenza dall' ingresso!

  ⇒Stati in sequenza non codificati, semplificando unità di controllo wrt. FSM

VEDIAMO IN DETTAGLIO…

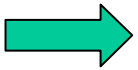
Implementazione di controllo e smistamento tramite ROM: dimensioni

ROM di controllo: 10 microistruzioni $\Rightarrow$ 4 bit di indirizzo
ciascuna parola (microistruzione) 16+2 = 18 bit

Dimensione: $2^4 * 18 = 288$ bit

ROM di smistamento: indirizzate da Opcode $\Rightarrow$ 6 bit di indirizzo
ciascuna parola ha 4 bit

Dimensione: $2^6 * 4 = 256$ bit



Dimensione totale = 288 + 2*256 = 800 bit

Implementazione di controllo e smistamento tramite PLA: dimensioni

PLA di controllo: funzione stato (4 bit) $\Rightarrow$ 16 + 2 bit di controllo di indirizzo

Avevamo visto che per 16 uscite si usano 10 mintermini (tutti i 10 stati); quindi i 2 bit di controllo di indirizzo useranno parte di questi mintermini

Dimensione: = 4*10 + 10*18 = 220

PLA di smistamento 1: funzione bit di opcode (6 bit) $\Rightarrow$ 4 bit di indirizzo [stato]
dalla specifica ROM di smistamento 1 si vede che ci sono 5 combinazioni di opcode che interessano: 5 mintermini

Dimensione: = 6*5 + 5*4 = 50

PLA di smistamento 2: funzione bit di opcode (6 bit) $\Rightarrow$ 4 bit di indirizzo [stato]
da specifica ROM di smistamento 2 si vede che ci sono 2 combinazioni di opcode che interessano: 2 mintermini

Dimensione: = 6*2 + 2*4 = 20

$\Rightarrow$ Dimensione totale = 220 + 50 + 20 = 290

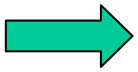
Confronto con implementazione “a stato esplicito”

Tramite ROM:

	Controllo	Stato futuro/ smistamento	Totale
Stato esplicito	256	4096	4.3K
Microprog.	288	512	800

Tramite PLA:

	Controllo	Stato futuro/ smistamento	Totale
Stato esplicito	200	140	340
Microprog.	220	70	290



Lieve aumento della dimensione per il controllo (dovuta all’introduzione dei segnali AddrCt per la scelta della microistruzione successiva) compensato dalla diminuzione della dimensione per stato futuro

Ottimizzazioni della logica di controllo

Favorite dall'uso di strumenti CAD ma anche da opportune scelte di progettazione:

- Minimizzazione logica delle funzioni di controllo e smistamento:
  - sfruttamento dei termini don't care
  - individuazione di segnali di ingresso che identificano istruzioni [nell'esempio lw e sw sono gli unici ad avere Op5=1] semplificando ROM/PLA di smistamento: favorita da scelta progettuale che assegna codici operativi correlati ad istruzioni che condividono stessi stati (stesse microistruzioni)

- Assegnazione degli stati:
  stati (indirizzi delle microistruzioni) scelti in modo da semplificare equazioni che legano gli stati alle uscite di controllo
  [l'assemblatore di microcodice può assegnare indirizzi in modo opportuno nel rispetto dei vincoli di sequenzialità delle microistruzioni]

• Riduzione della memoria di controllo mediante tecniche di *codifica*:

riduzione ampiezza delle microistruzioni:

- Gruppi di linee nelle microistruzioni possono essere codificate in un minor numero di bit.

Es. solo uno tra i 6 bit 13-8 della parola di controllo è attivo: codifica in 3 bit

Numero dello stato	Bit 17-2 della parola di controllo	Bit 1-0 della parola di controllo
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000000	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

NB: tempo di decodifica di solito trascurabile, costo HW decoder compensato da riduzione costo memoria di controllo

- uso di un campo *formato* delle microistruzioni (formati diversi per microistruzioni!)

[fornisce valori di default per segnali di controllo non specificati]

p. es. formato per microistruzioni di accesso in memoria e

formato per operaz. ALU (specifica implicitamente no read/write memoria)

In generale:

- Microcodice orizzontale (minimamente codificato)
  - più flessibile (può specificare qualunque combinazione di segnali)
  - aumenta costo dell'HW (memoria di controllo più ampia)
  - favorisce buoni T_{clock} e CPI [ma attenzione a memoria di controllo!]

VS.

- Microcodice verticale (massimamente codificato)
  - minor costo HW (costo decodificatori < riduzione costo memoria)
  - peggiori prestazioni (tempo di decodifica che può far aumentare T_{clock} + riduzione combinazioni possibili di segnali nella stessa microistruzione che può richiedere un maggior CPI)

NB: ABBIAMO VISTO

- Processore multi-ciclo [Unità di controllo sequenziale]
  - **specifica come FSM** e **implementazione come FSM**
  - **specifica come microprogramma** e **implementazione con sequenzializzatore**

Sono possibili anche gli altri due passaggi!

- **specifica come microprogramma** e **implementazione come FSM**

Ogni microistruzione corrisponde ad uno stato: codifica (numerica) dello stato

$\Rightarrow$ dai valori dei campi di controllo delle microistruzioni deriva la funzione:

stato $\rightarrow$ uscita [funzione di uscita]

$\Rightarrow$ dal campo *Sequencing* di una microistruzione (stato) + tabelle smistamento simboliche [che codificano istruzione futura sulla base di ingressi opcode] deriva la funzione:

stato*ingressi $\rightarrow$ stato futuro [funzione di stato futuro]

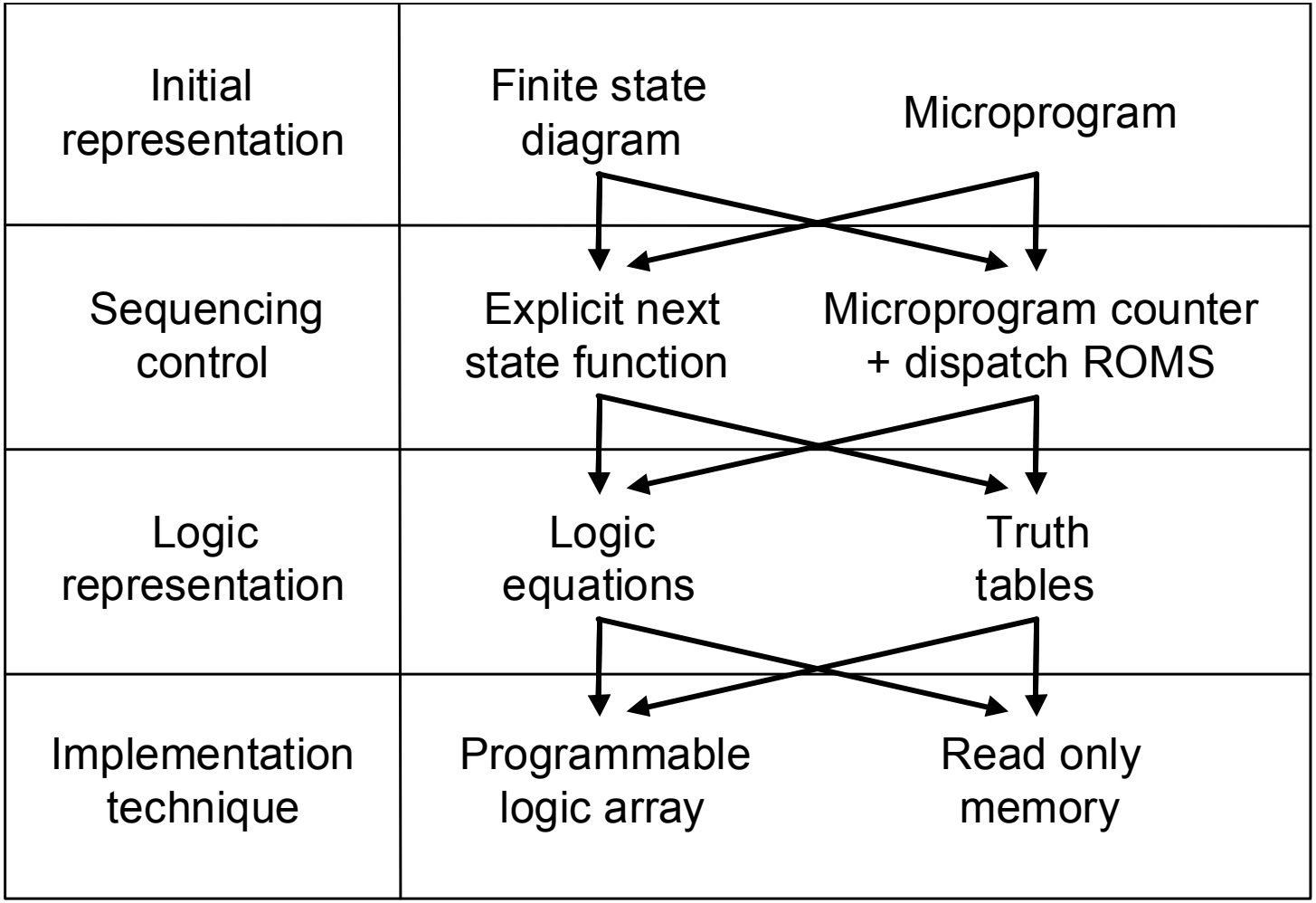
• **specifica come FSM** e **implementazione con sequenzializzatore**

Ogni stato corrisponde ad una microistruzione: assegnazione di un ordine agli stati
(conviene: transizioni tra stati indipendenti dall'ingresso ⇒ stati posti in sequenza!)

⇒ dalla funzione di uscita derivano per ogni microistruzione
i segnali di controllo da attivare
⇒ dalla funzione di stato futuro si può stabilire se vi sia un'unica transizione
verso la microistruzione seguente, verso la microistruzione di fetch o se
la transizione dipenda dall'ingresso [tabella di smistamento], ovvero:
campo *Sequencing*
tabelle di smistamento

NB: l'uso di strumenti automatici per la realizzazione (in grado di applicare tecniche di ottimizzazione) rende le fasi di specifica e realizzazione relativamente indipendenti! Si può scegliere la specifica che risulta più semplice e "gestibile"

Controllo di un processore-multiciclo: Riepilogo specifica e realizzazione



- Modalità di rappresentazione:

  scelte per ragioni di semplicità progettuale

- Realizzazione di stato futuro:

  sequenzializzatore più efficiente all'aumentare del numero di stati

  e delle sequenze composte da stati consecutivi senza salti

- Realizzazione logica di controllo [parti combinatorie]:

  PLA sono più efficienti (vantaggio non sussiste se ROM è densa)

  ROM è soluzione più adeguata se funzione di controllo è in un chip diverso rispetto a unità di elaborazione (possibilità di cambiare microcodice indipendentemente dal resto del processore)

  └── NB: di fatto per ragioni di efficienza l'unità di controllo è oggi sempre implementata nello stesso chip. Inoltre, con CAD difficoltà di progettazione PLA non sussiste più.

  NB2: CACHE $\Rightarrow$ ROM di microcodice non molto più veloce di RAM con LM: se microprogramma non è efficiente, sequenze di istruzioni macchina possono essere più efficienti di una corrispondente istruzione complessa!