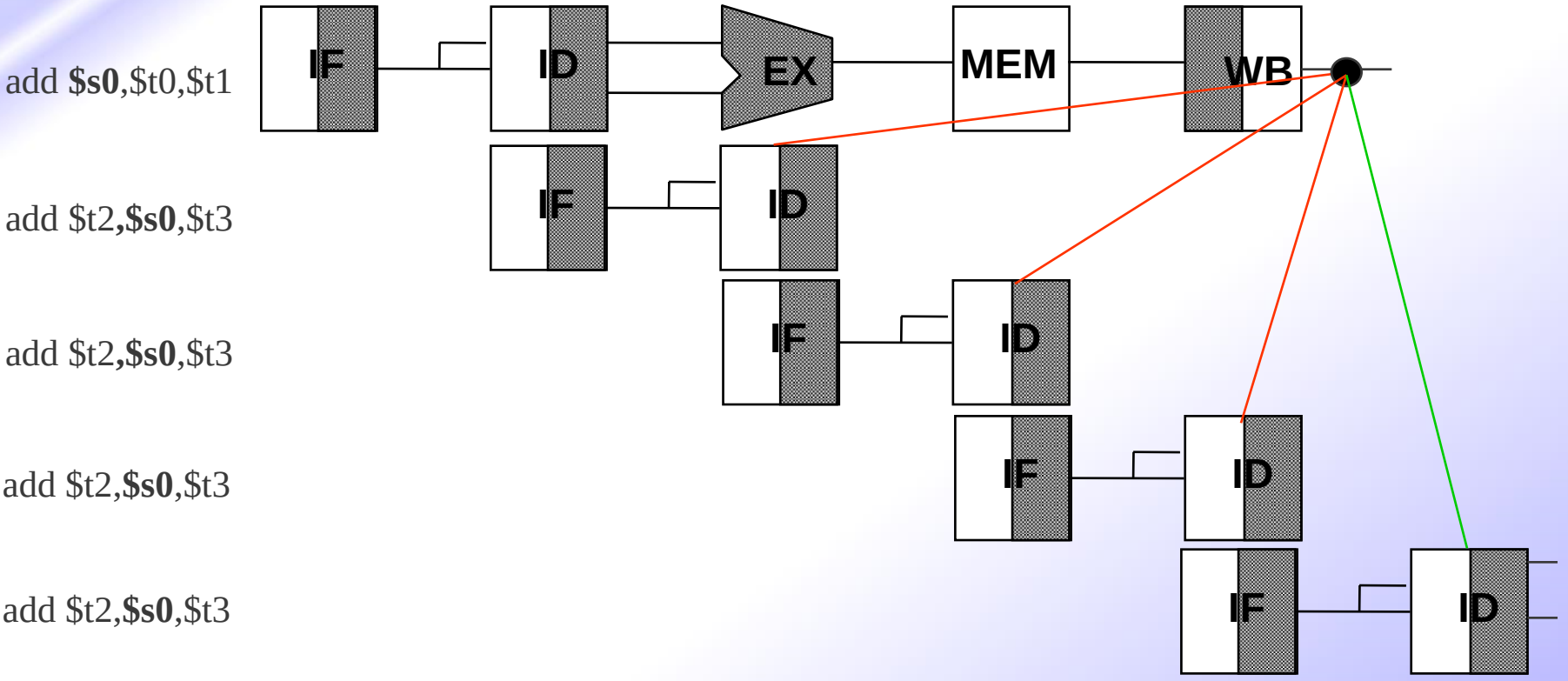
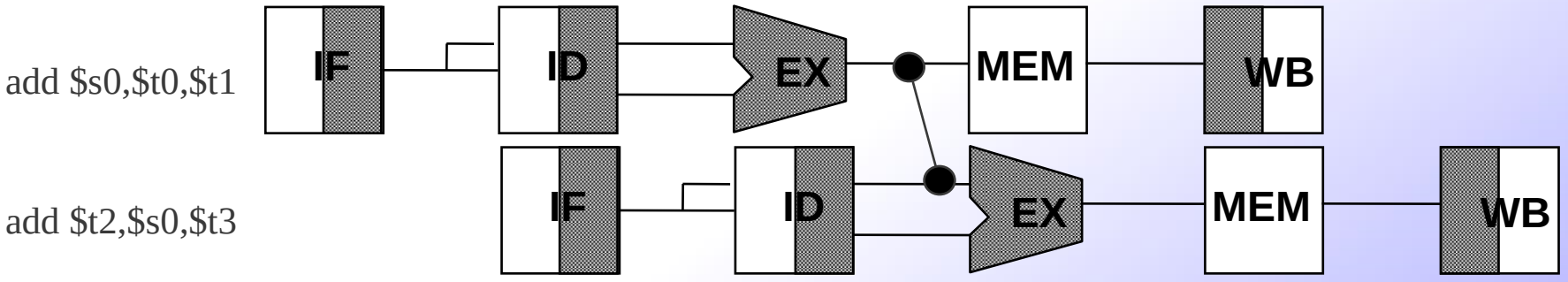
Criticità sui dati

- Un'istruzione dipende dal risultato di un'istruzione precedente ancora in pipeline



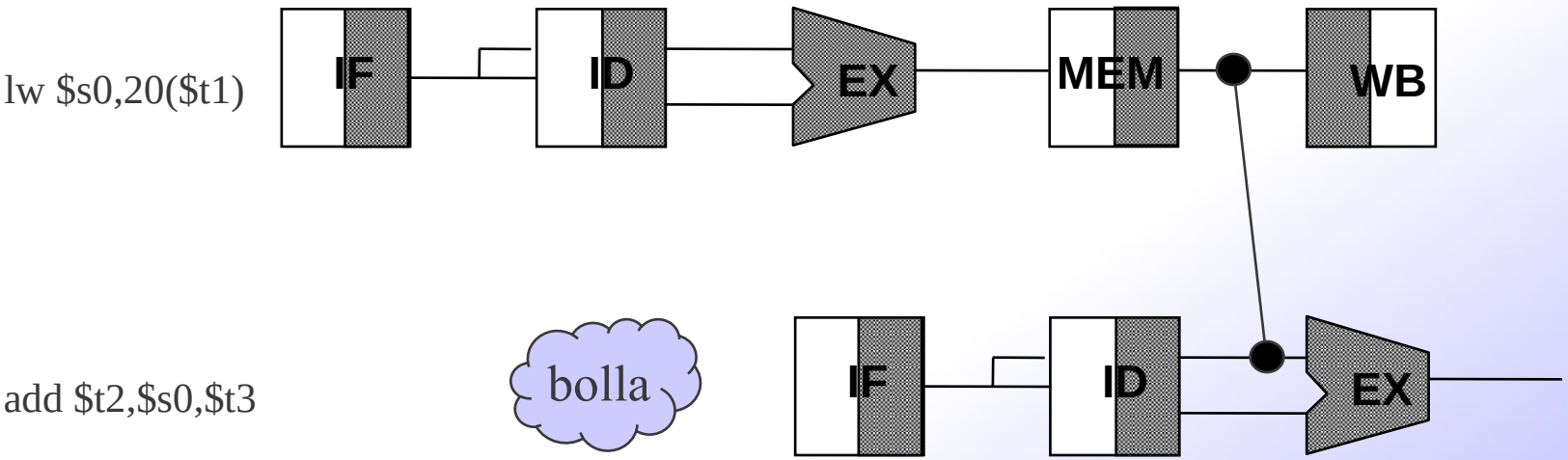
Propagazione *(forwarding)*

- Soluzione al problema delle criticità sui dati
- **Osservazione:**
  - Il risultato della EX può essere reso subito disponibile per la EX della fase successiva



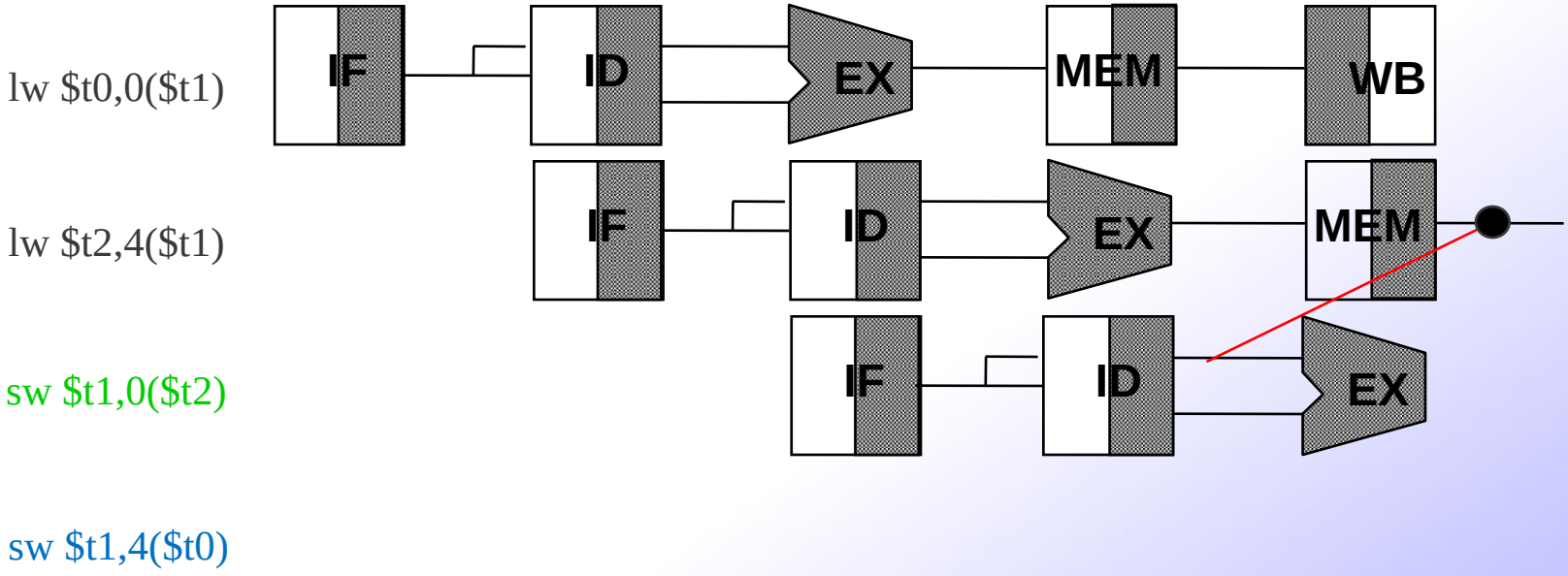
Ancora criticità dati: caso *load-use*

- Altro caso: istruzione di tipo R che segue una lw. In questo caso, il forwarding è da MEM a EX

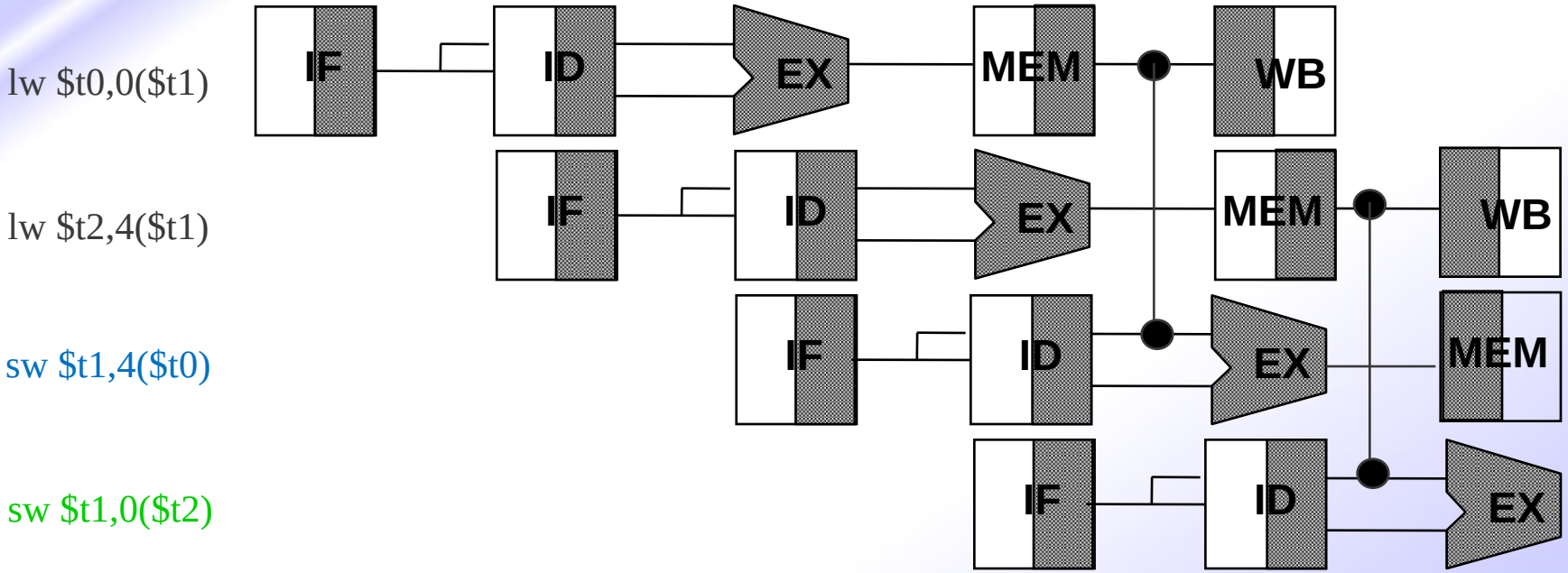


Forwarding + Riordino codice

- Soluzione al problema delle criticità sui dati



Forwarding + Riordino codice



- Nell'architettura MIPS, il compilatore cerca di evitare che l'istruzione successiva alla lw dipenda dalla lw stessa

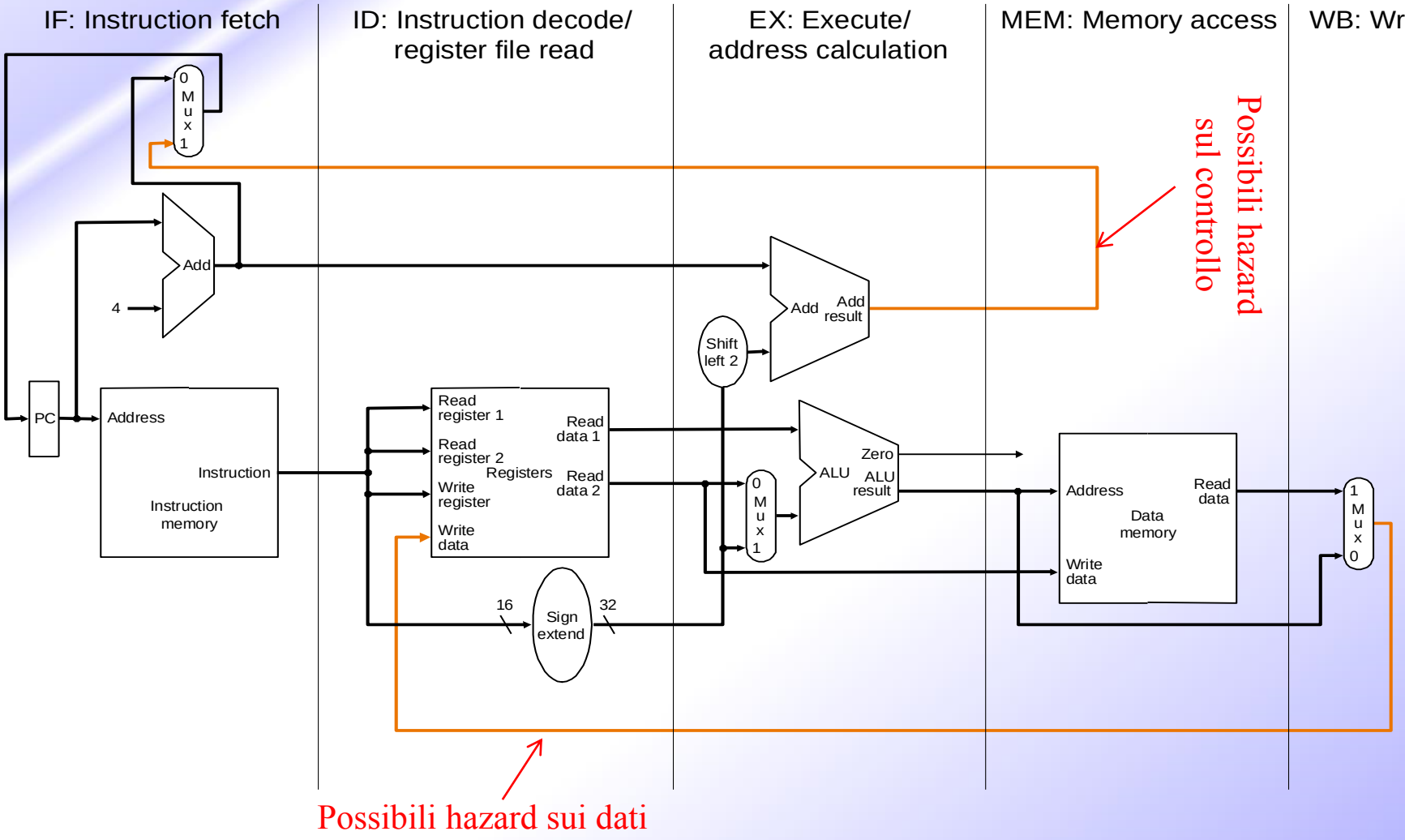
Cammino dei Dati

Semplificazioni

- Per il momento, ignoriamo le criticità

- Istruzioni considerate:
  - lw
  - sw
  - add, sub, or, and
  - beq

Cammino dati a ciclo singolo



Registri di Pipeline

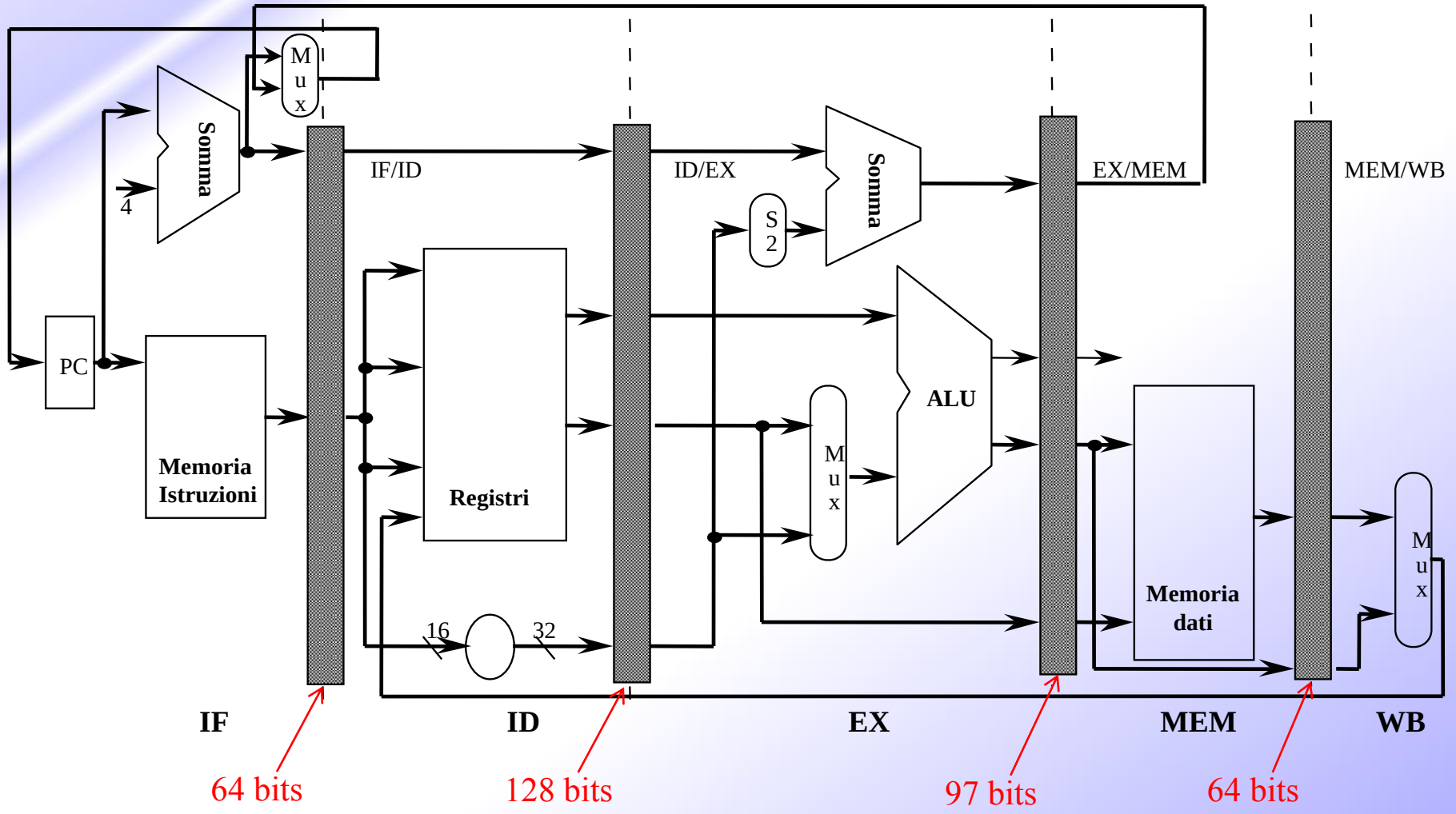
- Dopo aver letto l'istruzione *i*, IF è libero per fare il fetch della prossima istruzione, *i+1*
- Prima, è però necessario *salvare l'istruzione i* per usarla nei 4 stadi successivi (ID, EX, MEM, WB)
- Lo stesso principio si applica ad ogni altro stadio
- **Soluzione.** Registri tra uno stadio e l'altro (analogia: cesto tra una fase e l'altra della lavanderia):
  - IF/ID ID/EX EX/MEM MEM/WB
  - No, perchè WB scrive direttamente sul file registri

- Ad ogni ciclo di clock, ogni istruzione nella pipeline avanza di uno stadio da sinistra verso destra

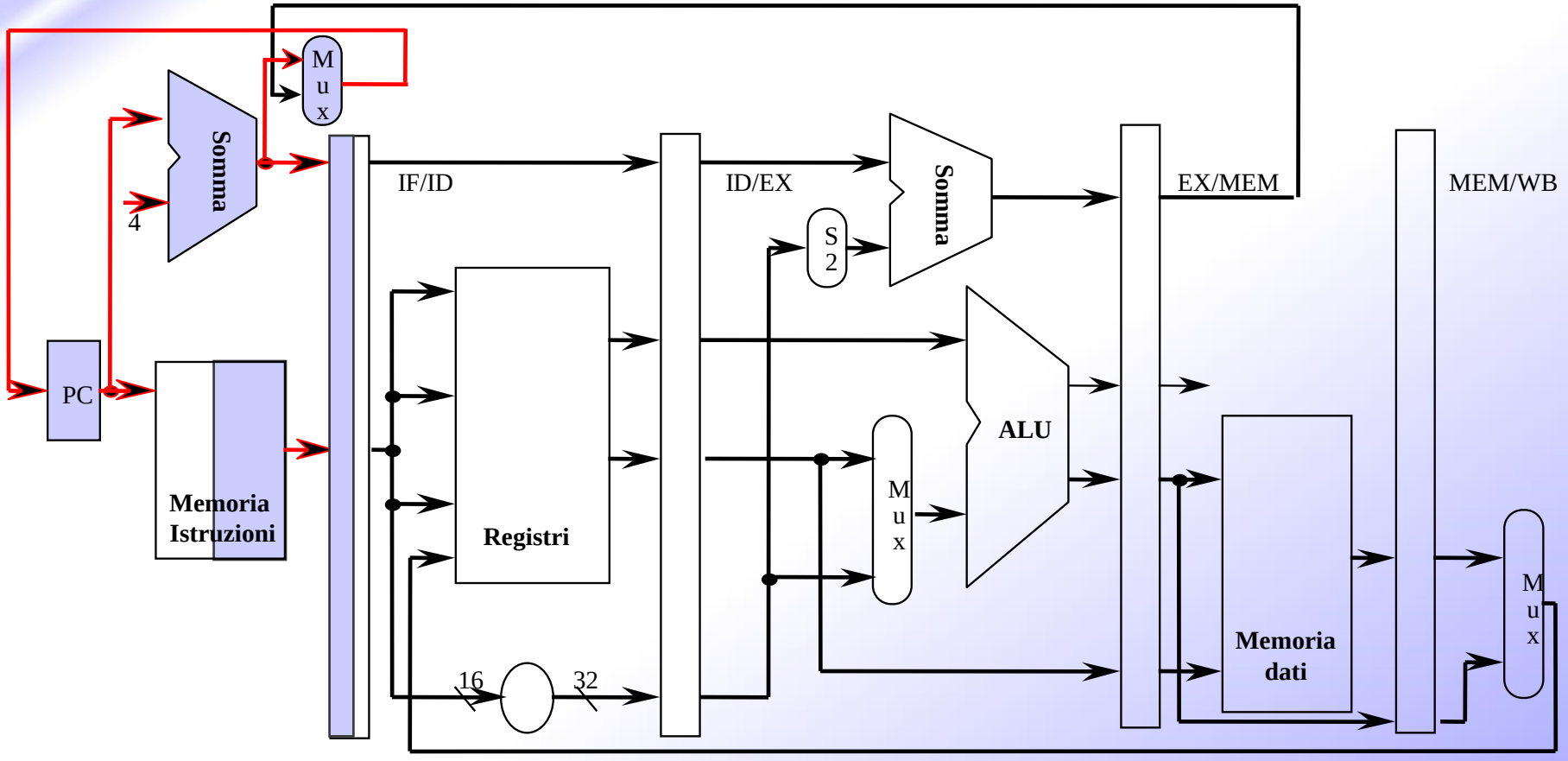
Un principio generale

- Ad ogni stadio, ogni informazione potenzialmente necessaria ad uno stadio successivo viene passata lungo la pipeline, attraverso i pipeline register

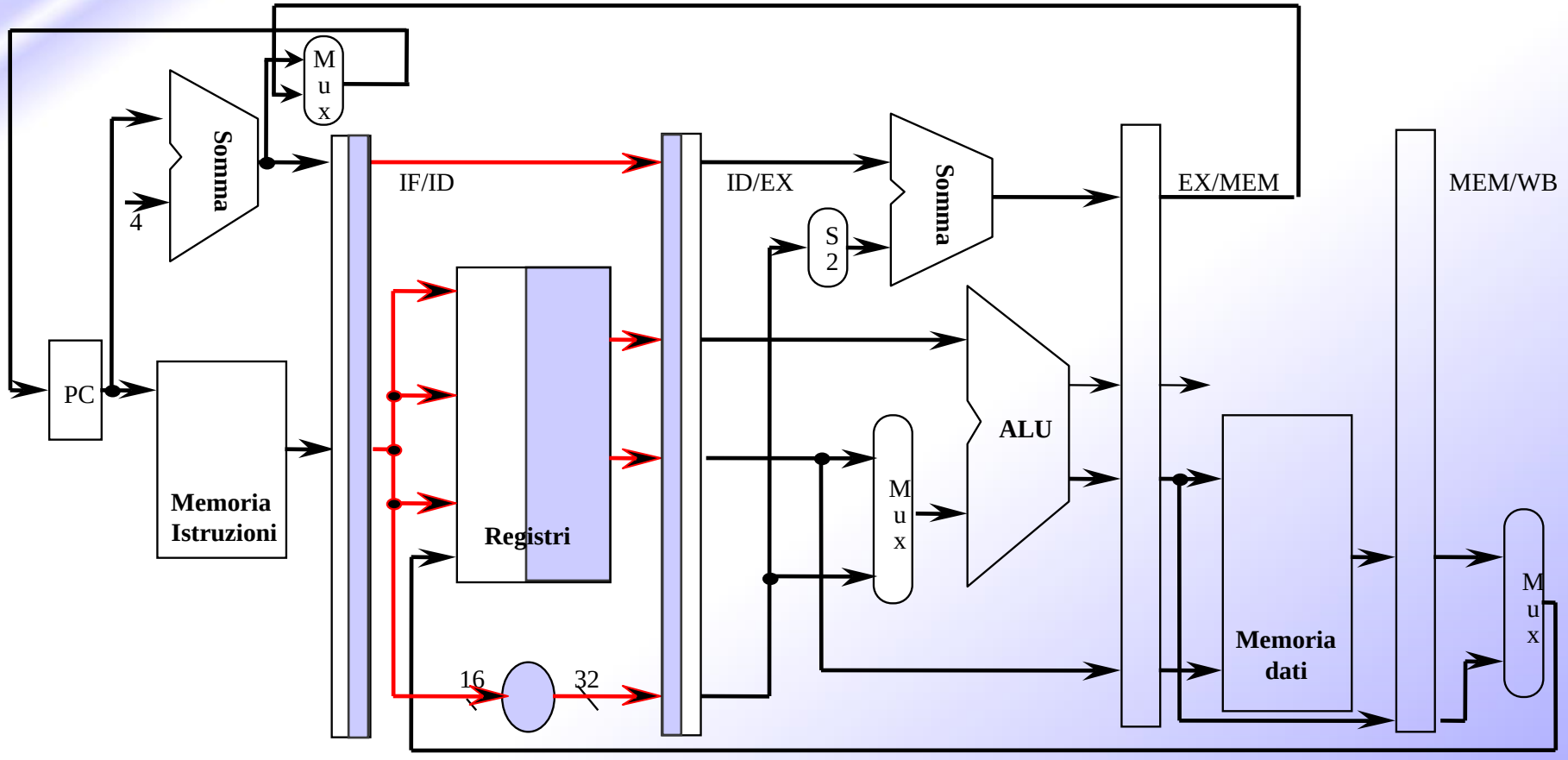
Il cammino dei dati con pipeline



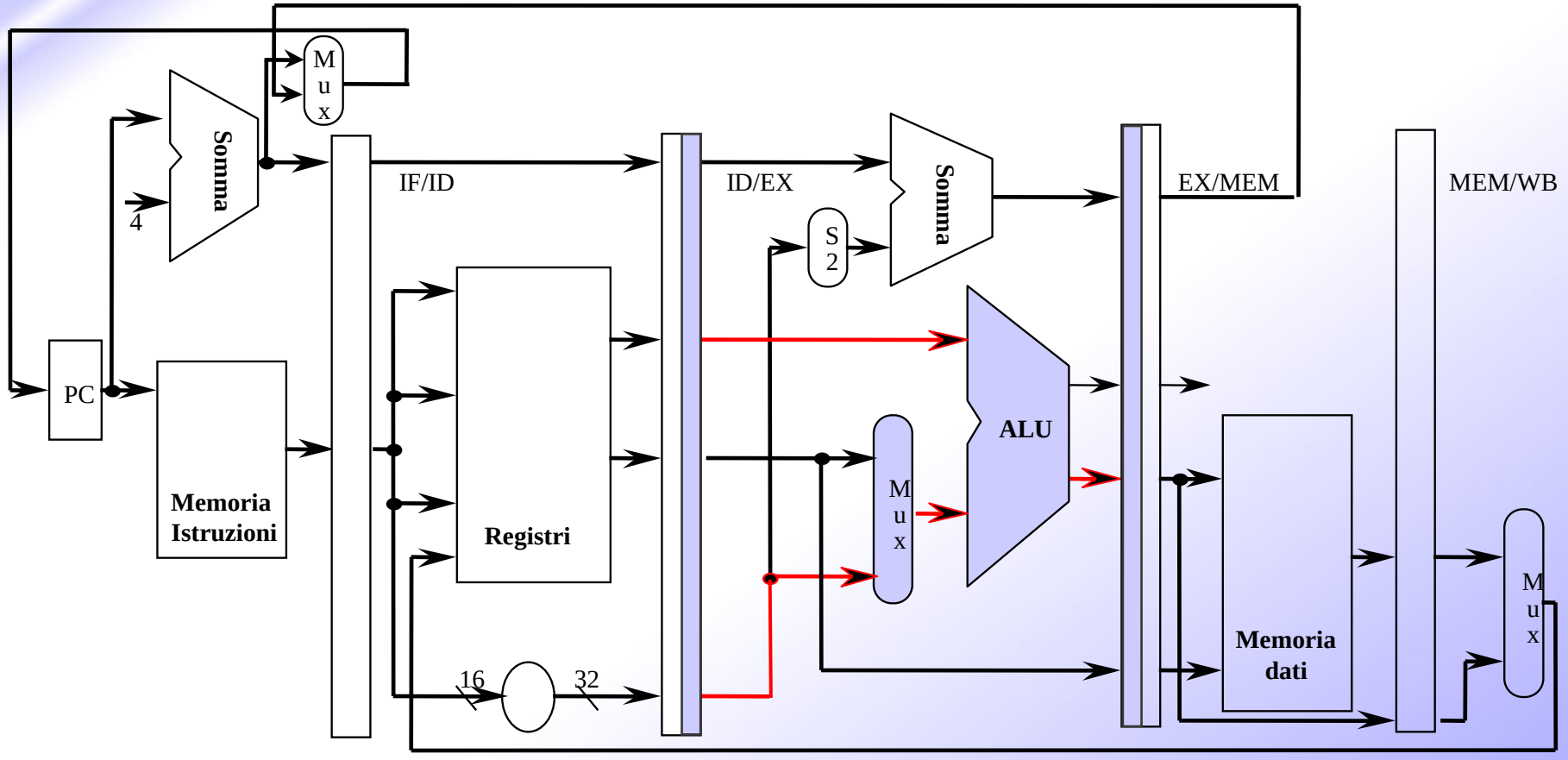
Esecuzione della lw: IF



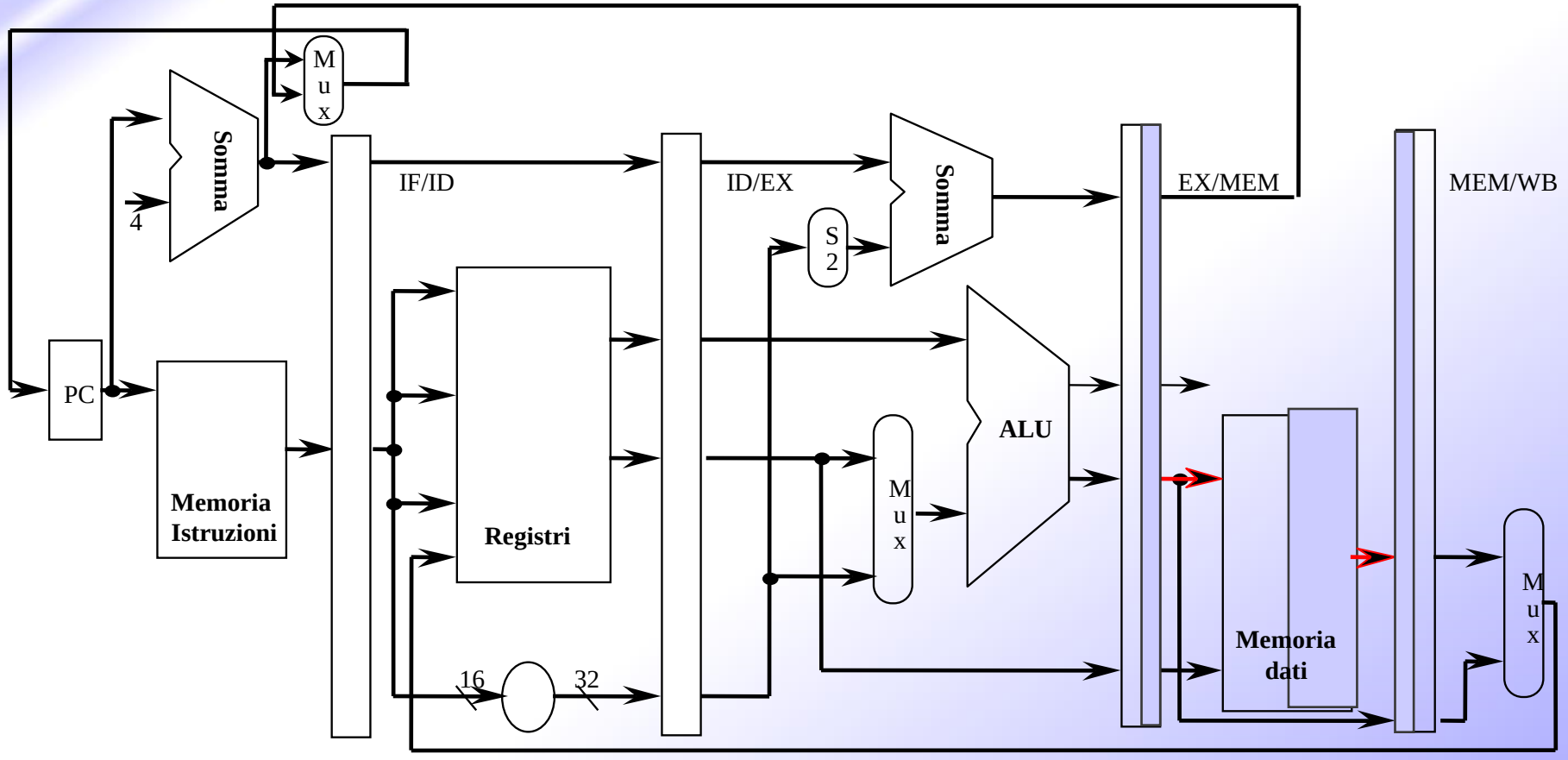
Esecuzione della lw: ID



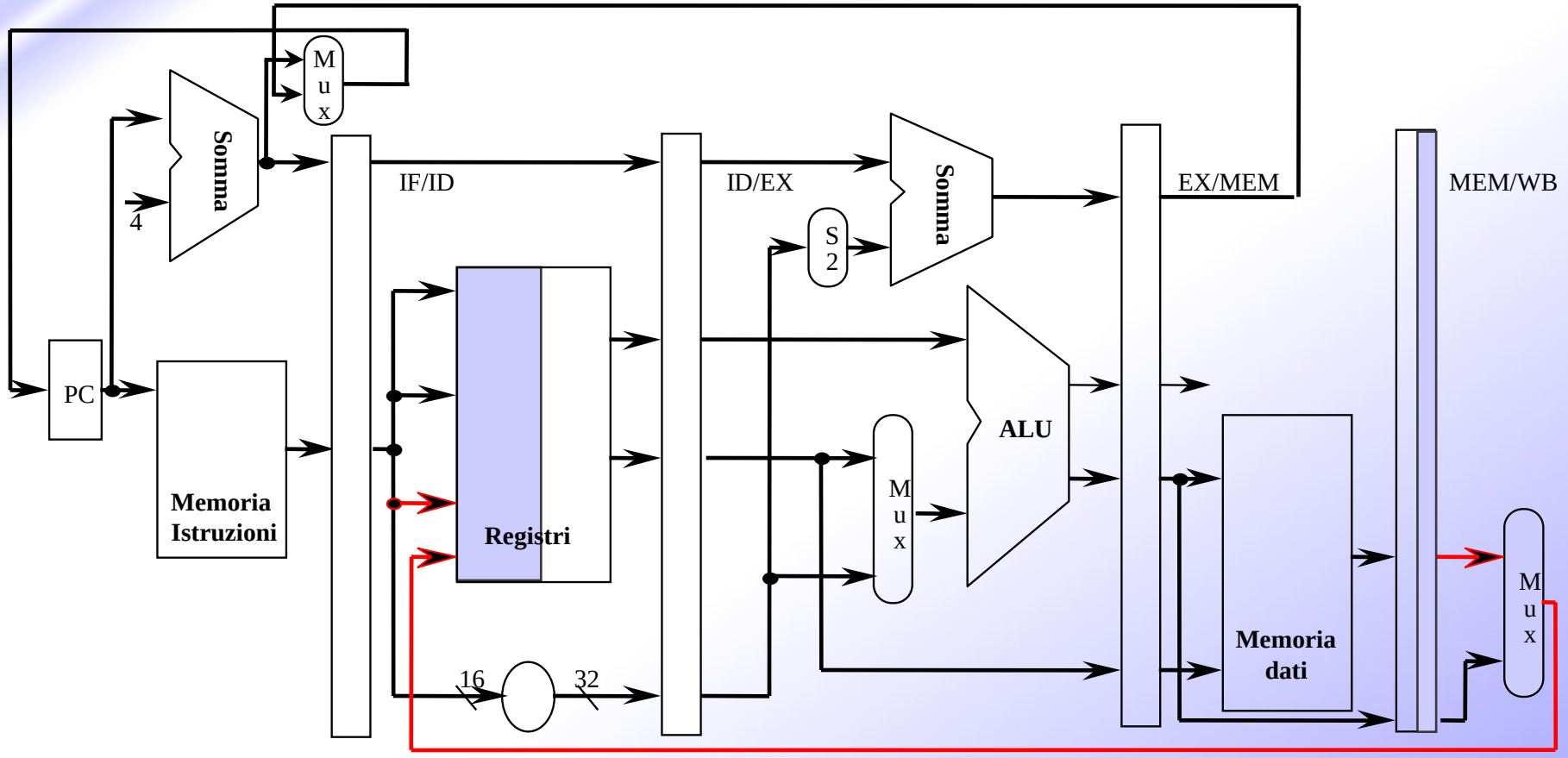
Esecuzione della lw: EX



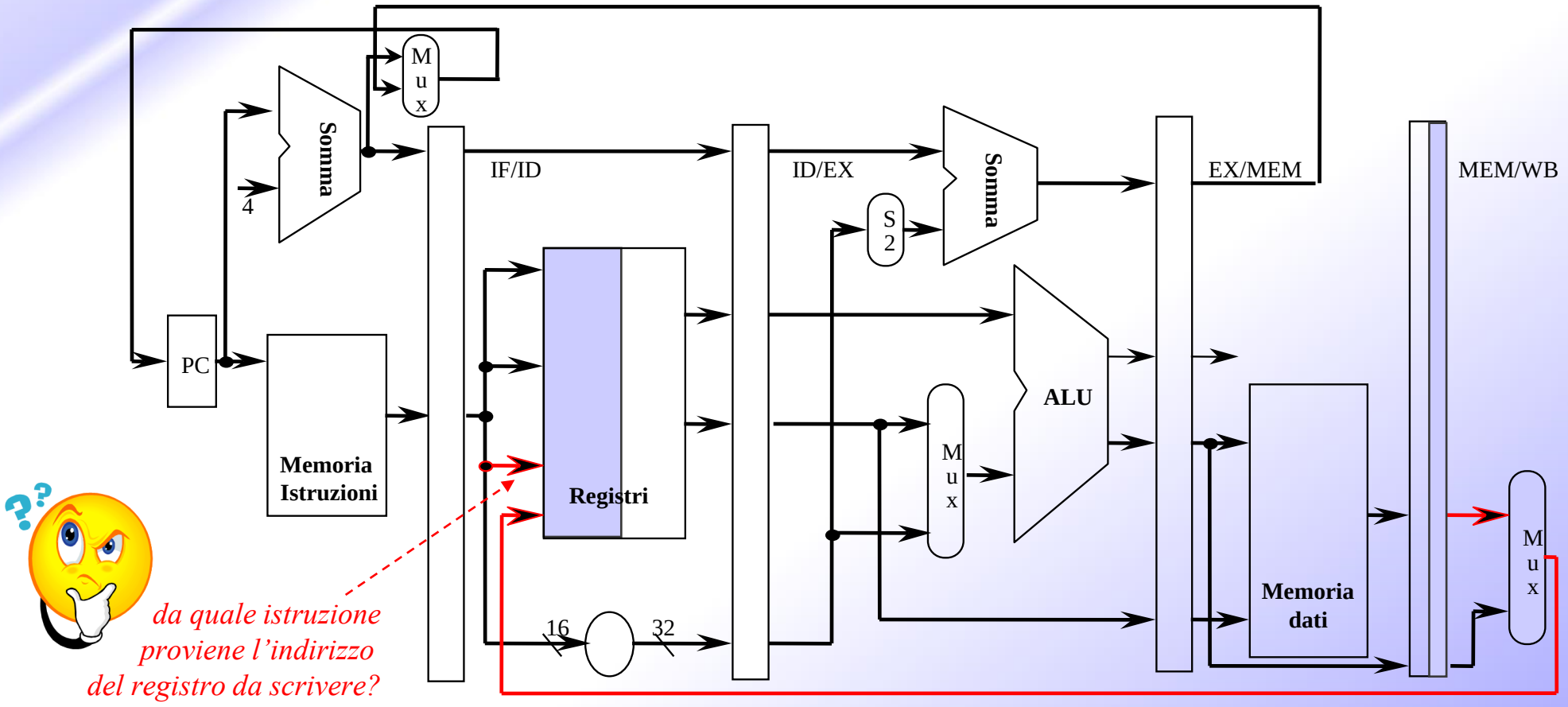
Esecuzione della lw: MEM



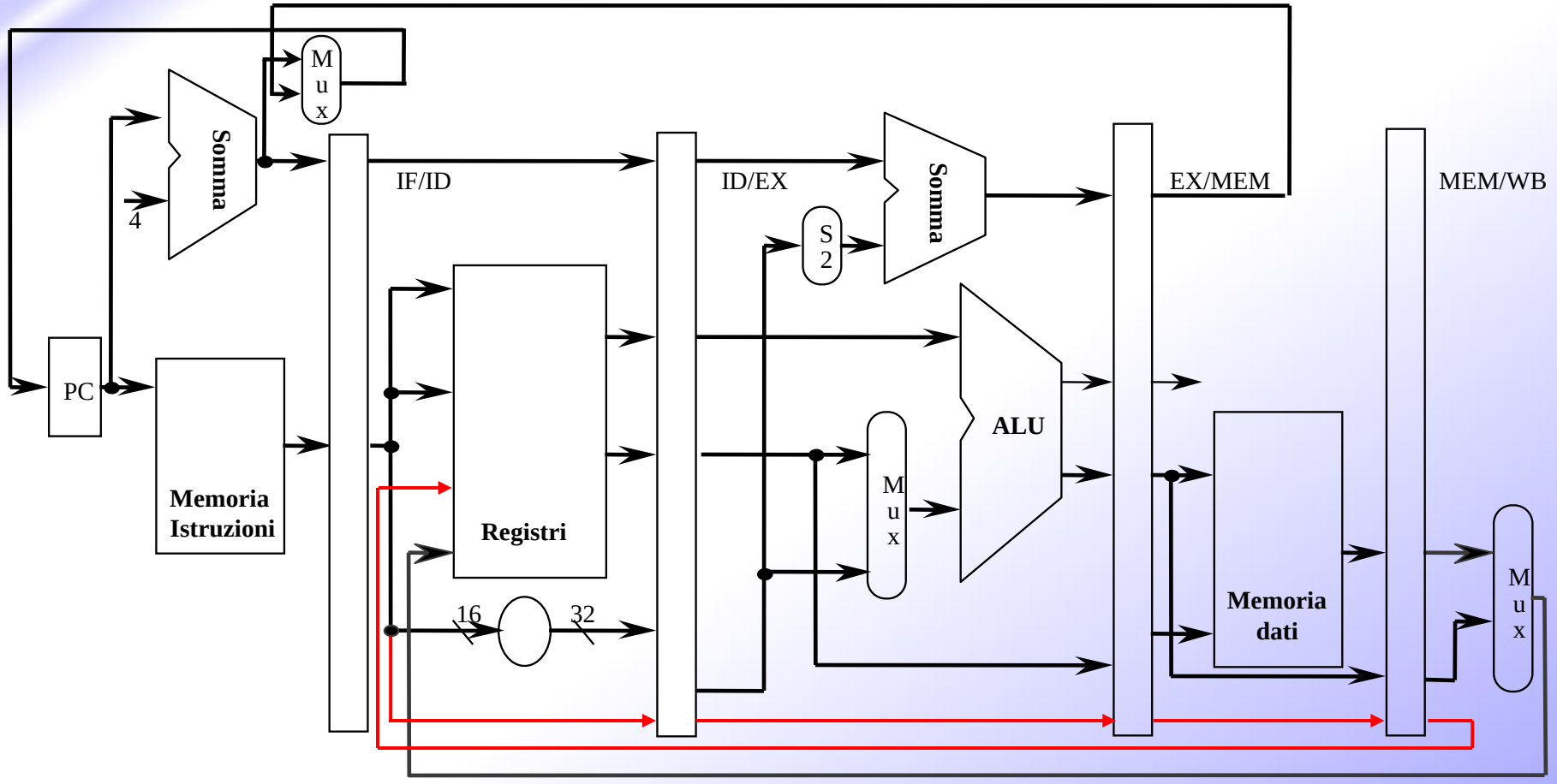
Esecuzione della lw: WB



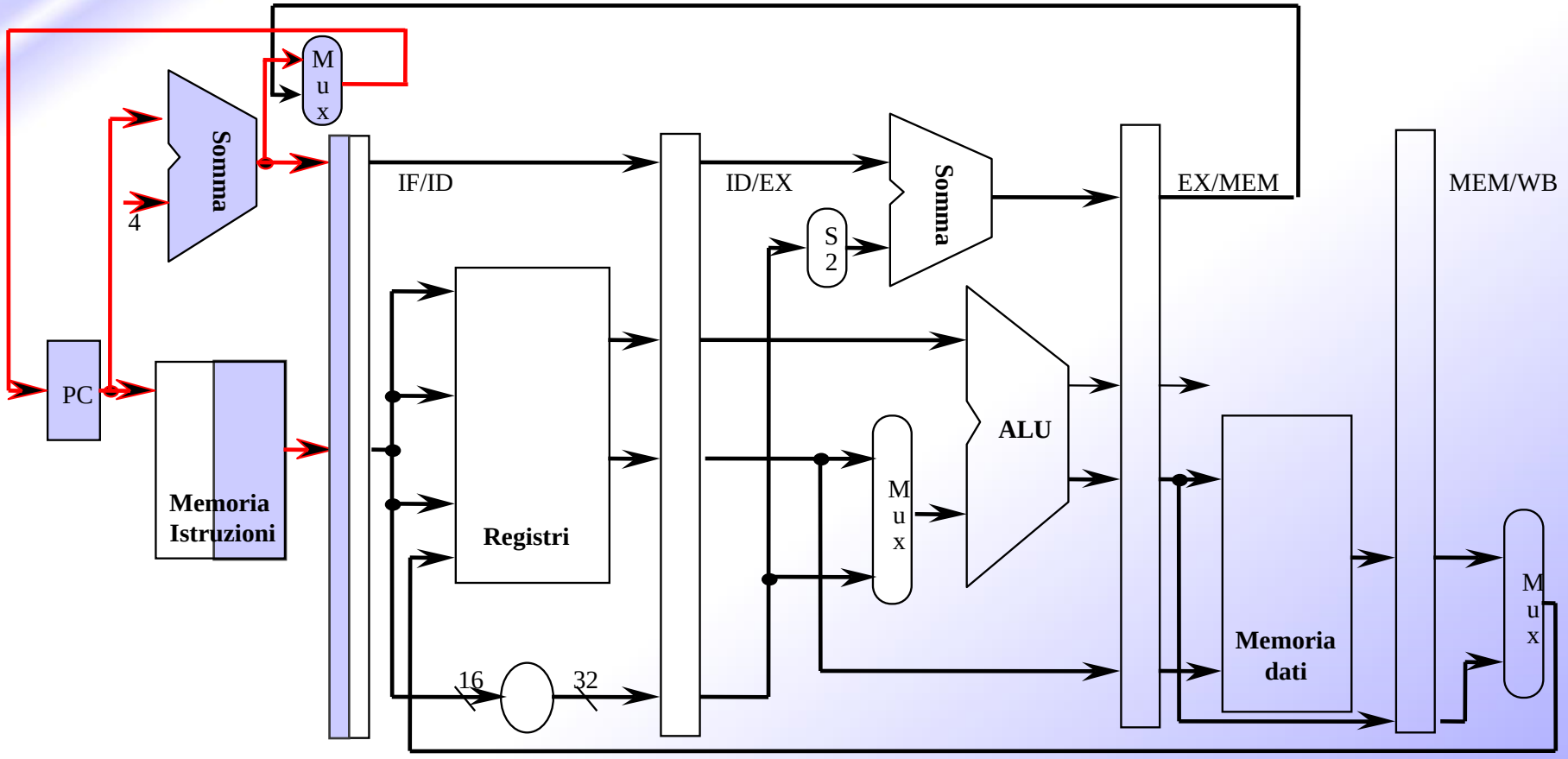
Errore nella WB della lw



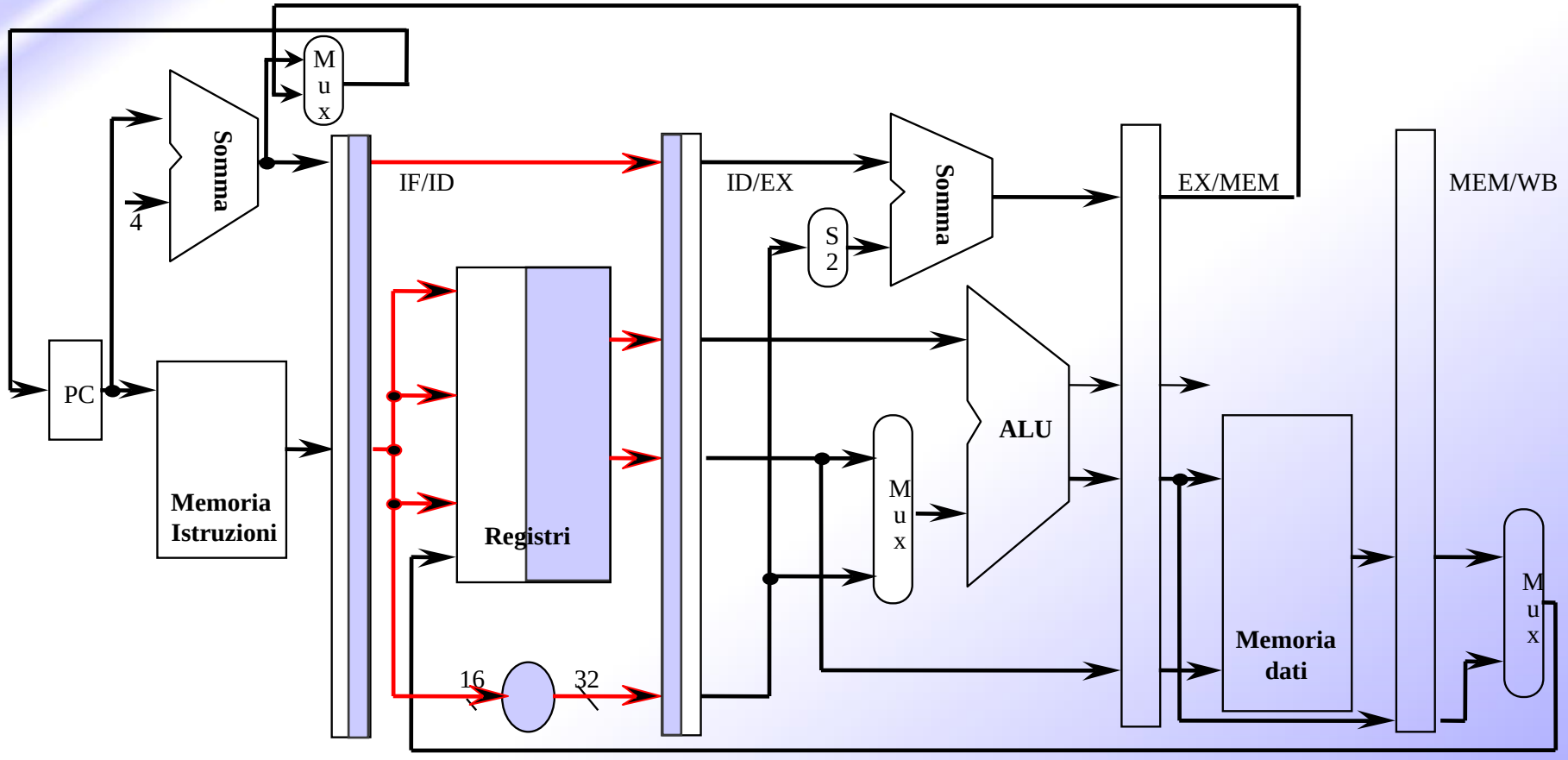
Cammino dei dati corretto



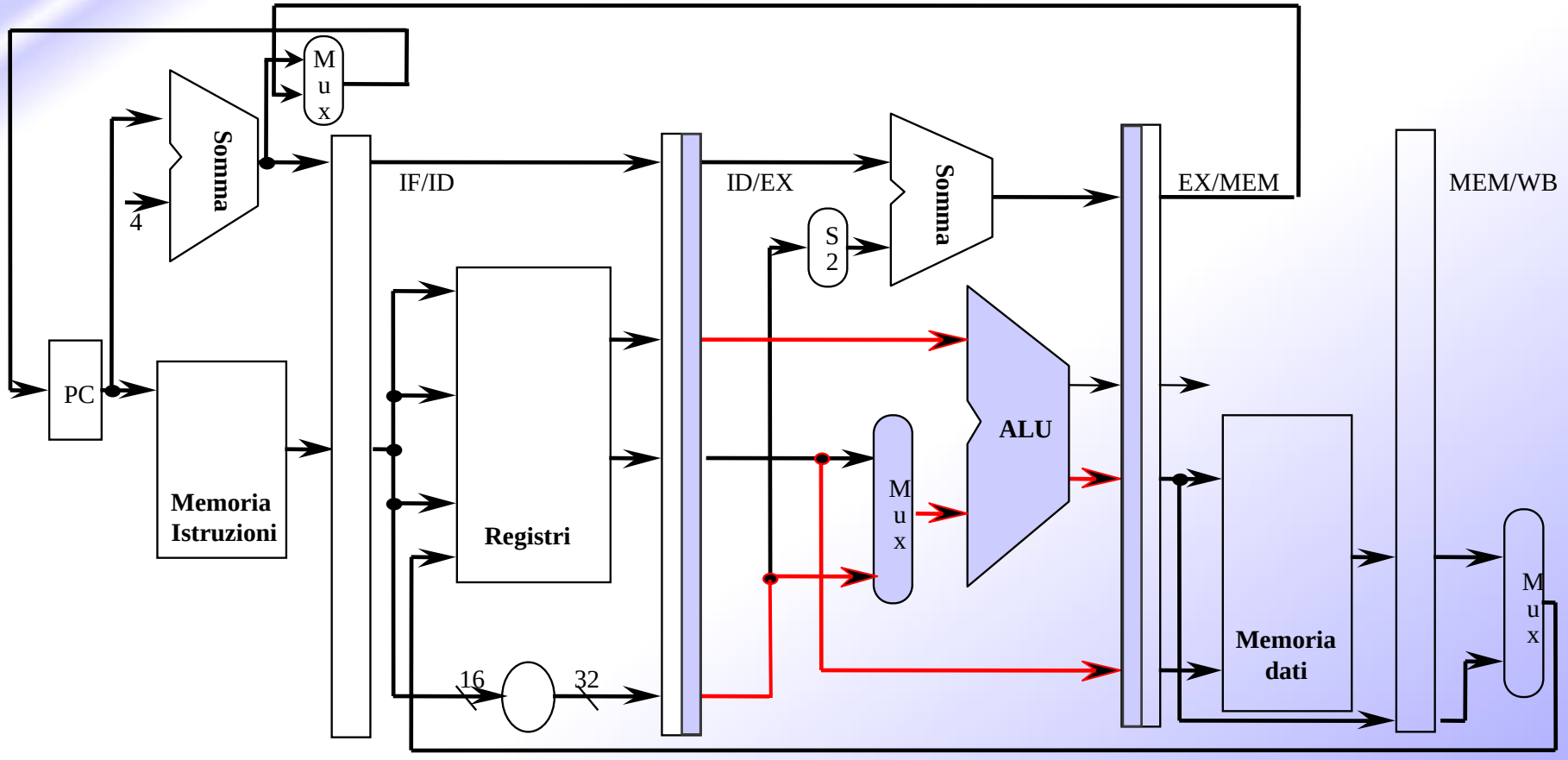
Esecuzione della sw: IF



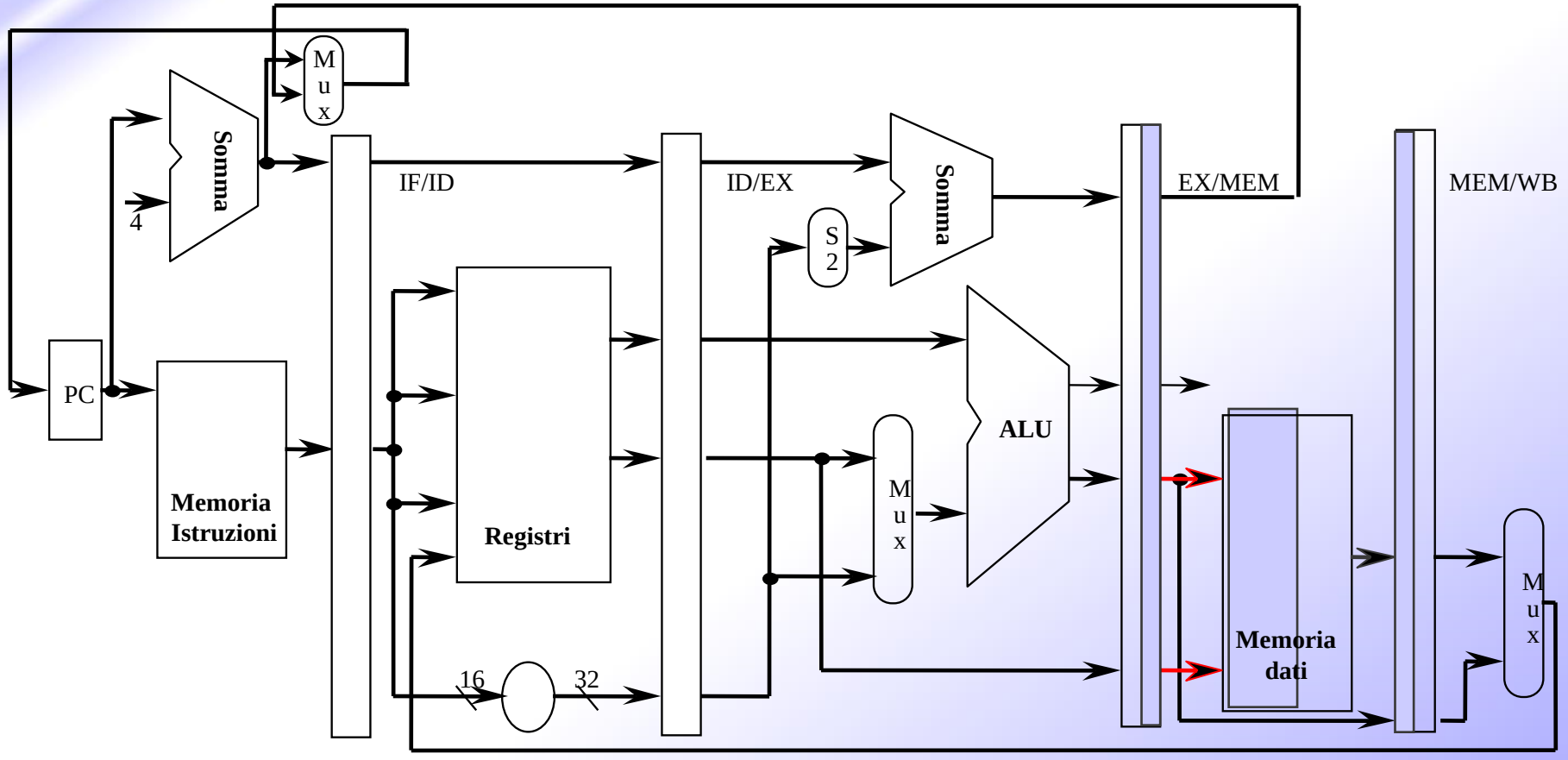
Esecuzione della sw: ID



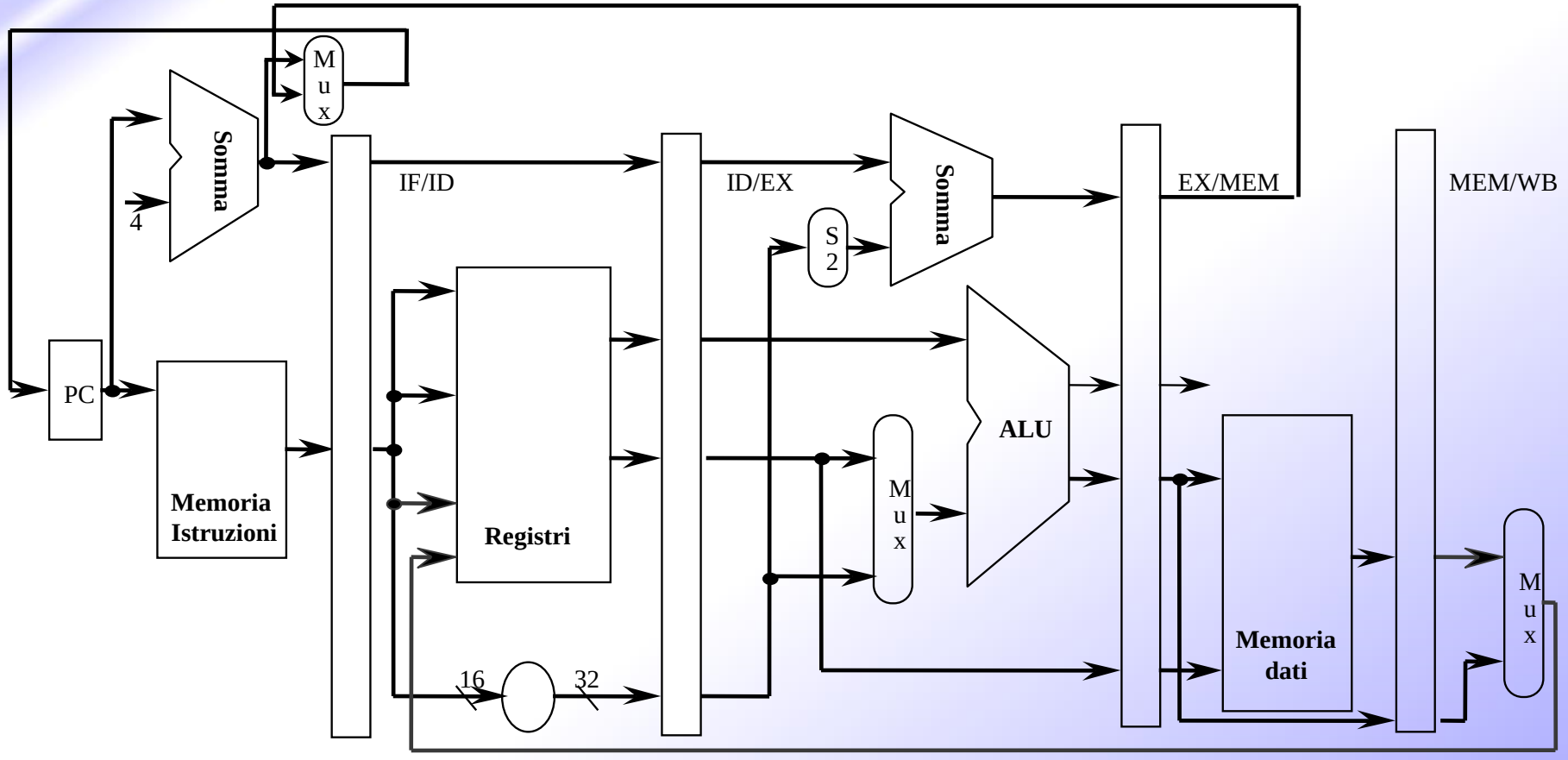
Esecuzione della sw: EX



Esecuzione della sw: MEM

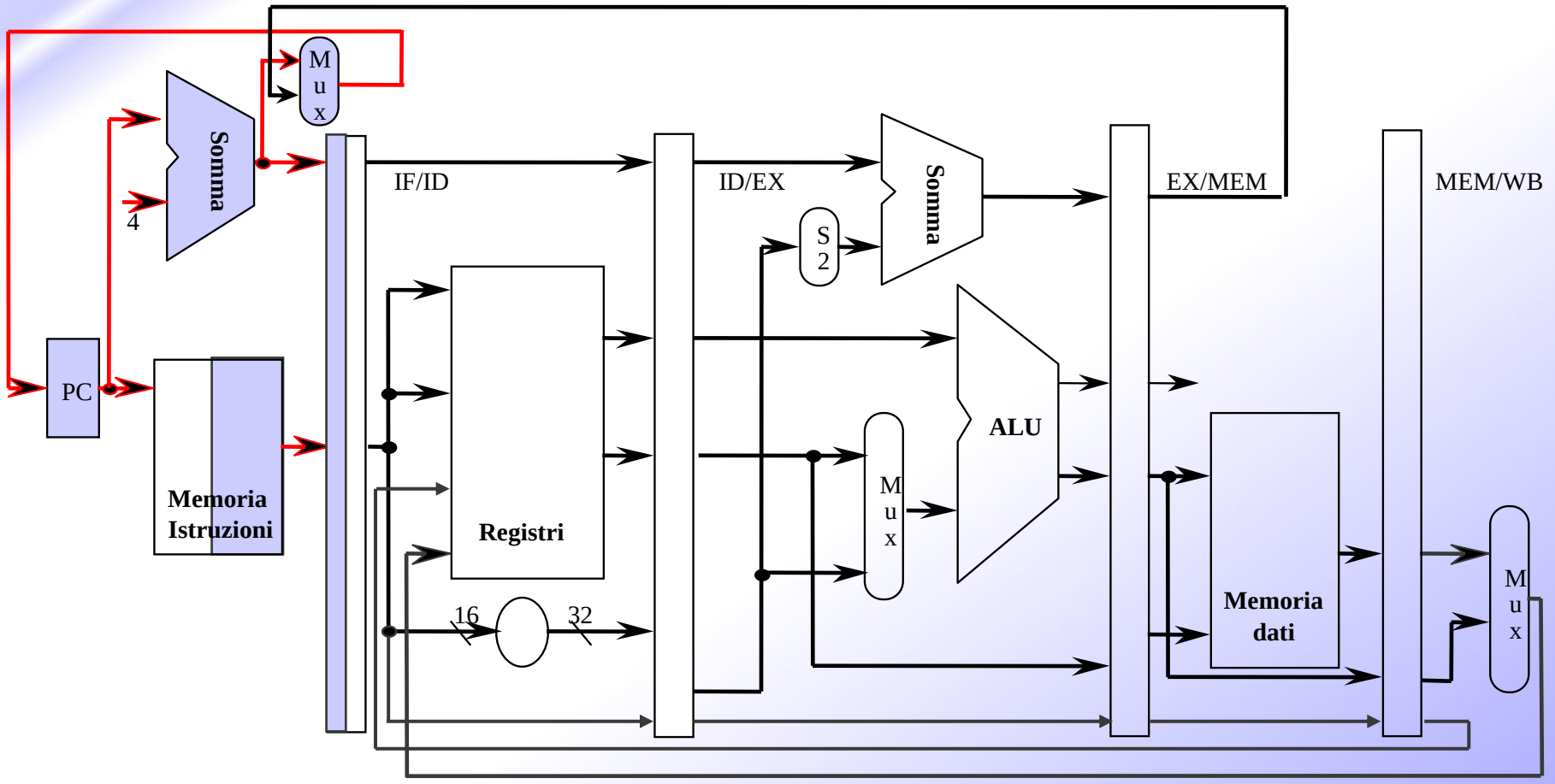


Esecuzione della sw: WB



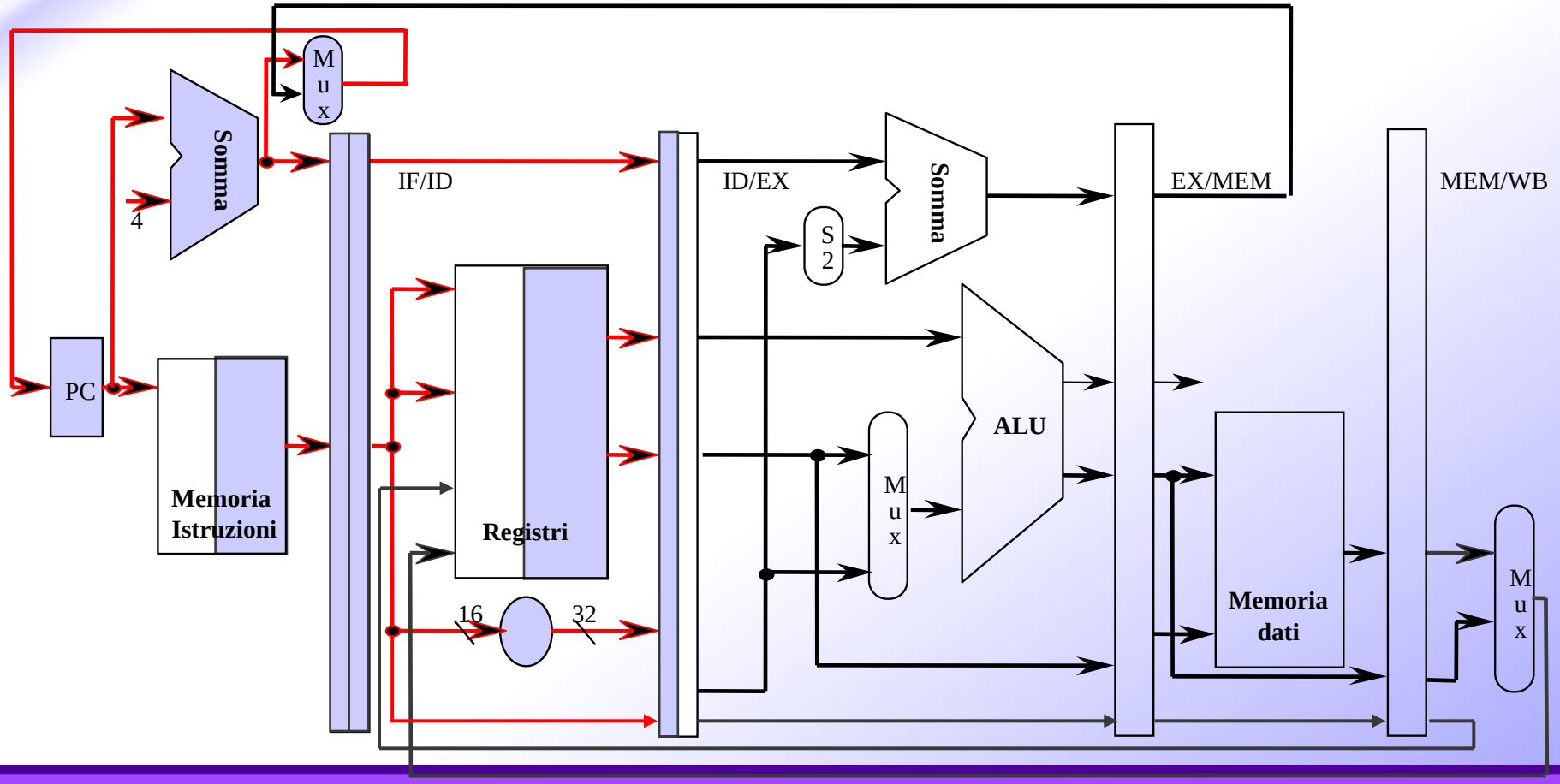
Esecuzione di 2 istruzioni: ciclo 1

lw $s4,20($s1) seguita da **sub $t3,$s2,$s3**



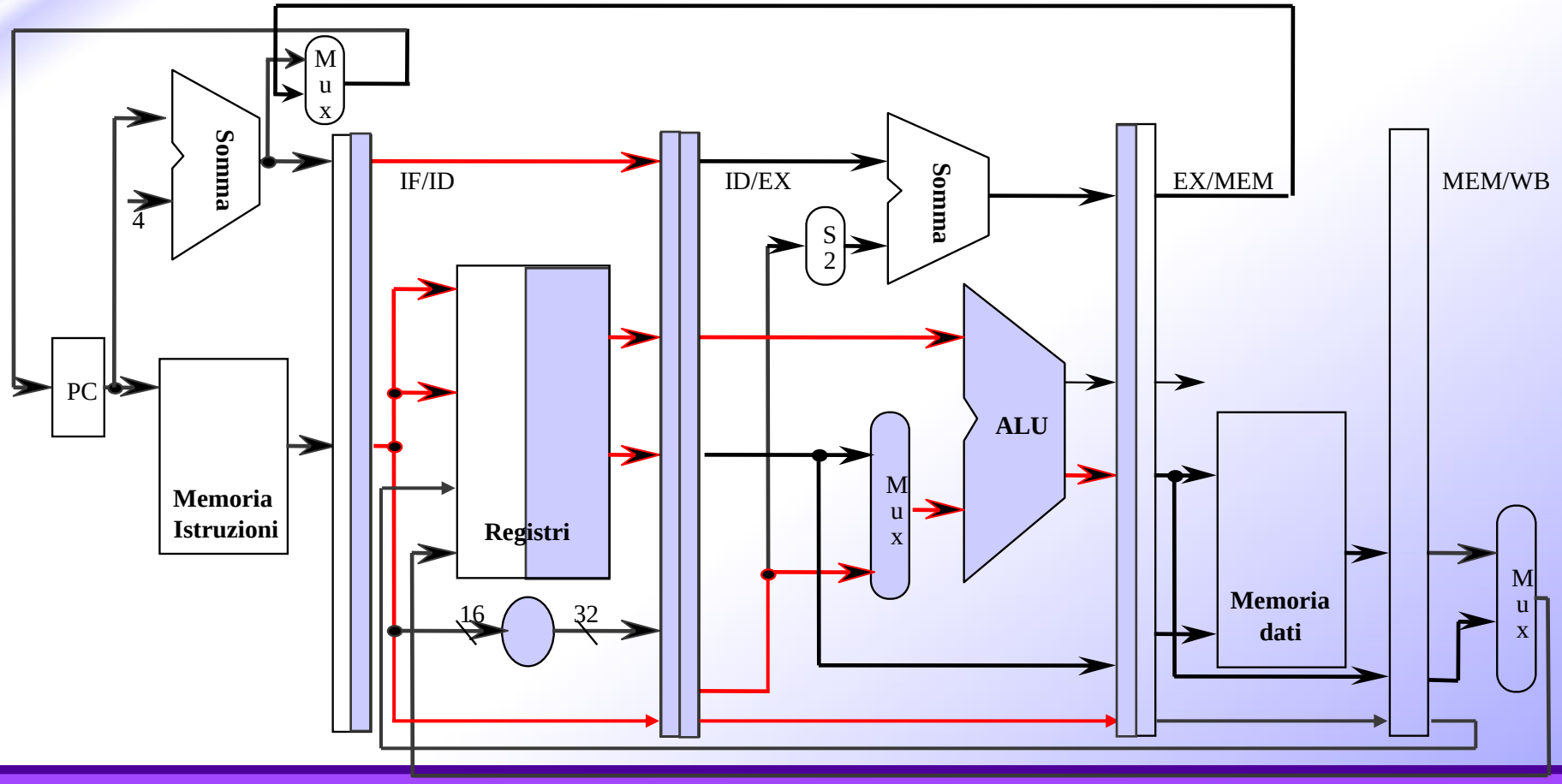
Esecuzione 2 istruzioni: ciclo 2

lw $s4,20($s1) seguita da **sub $t3,$s2,$s3**



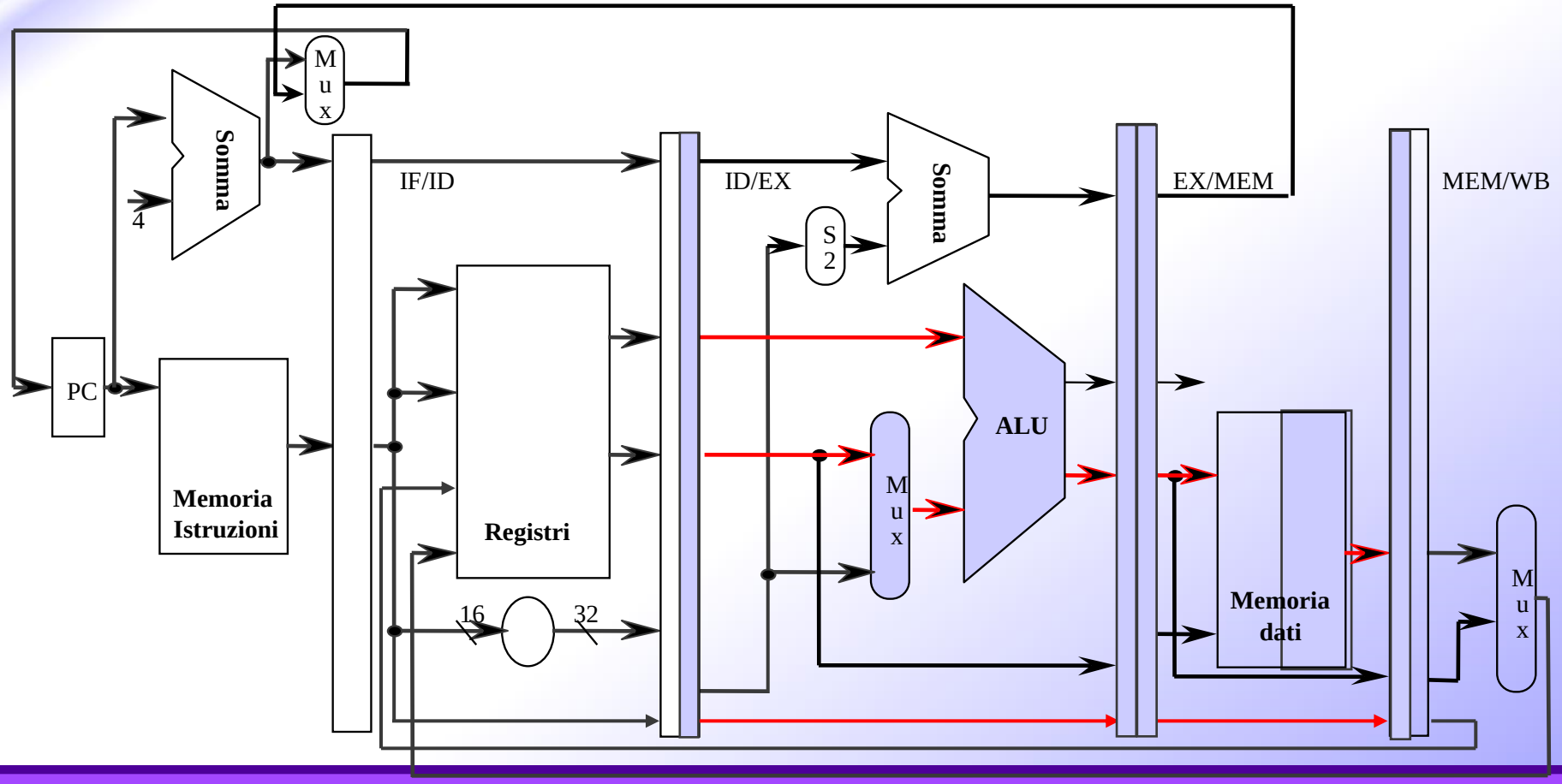
Esecuzione 2 istruzioni: ciclo 3

lw $s4,20($s1) seguita da **sub $t3,$s2,$s3**



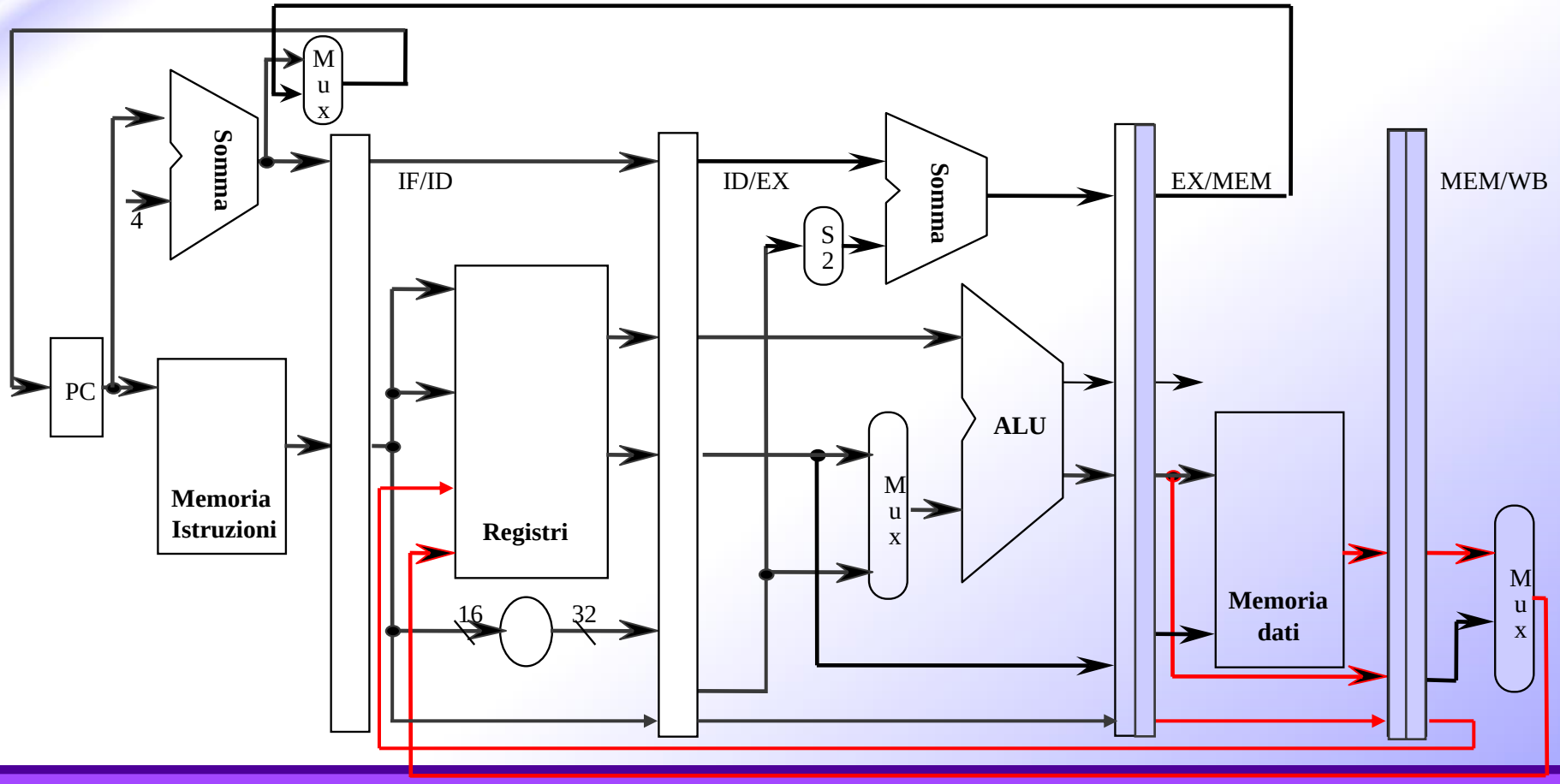
Esecuzione 2 istruzioni: ciclo 4

lw $s4,20($s1) seguita da **sub $t3,$s2,$s3**



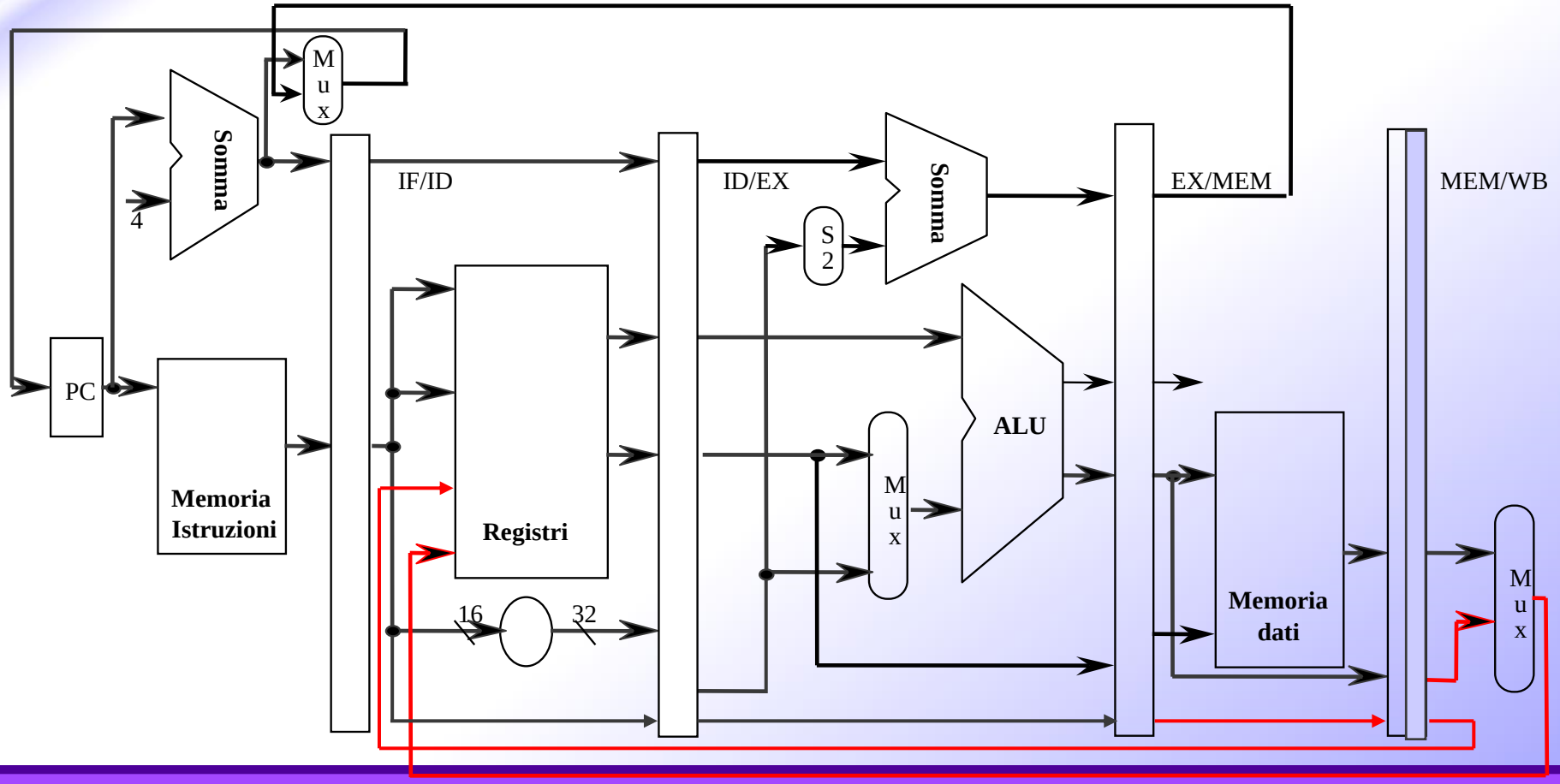
Esecuzione 2 istruzioni: ciclo 5

lw $s4,20($s1) seguita da **sub $t3,$s2,$s3**



Esecuzione 2 istruzioni: ciclo 6

lw $s4,20($s1) seguita da **sub $t3,$s2,$s3**



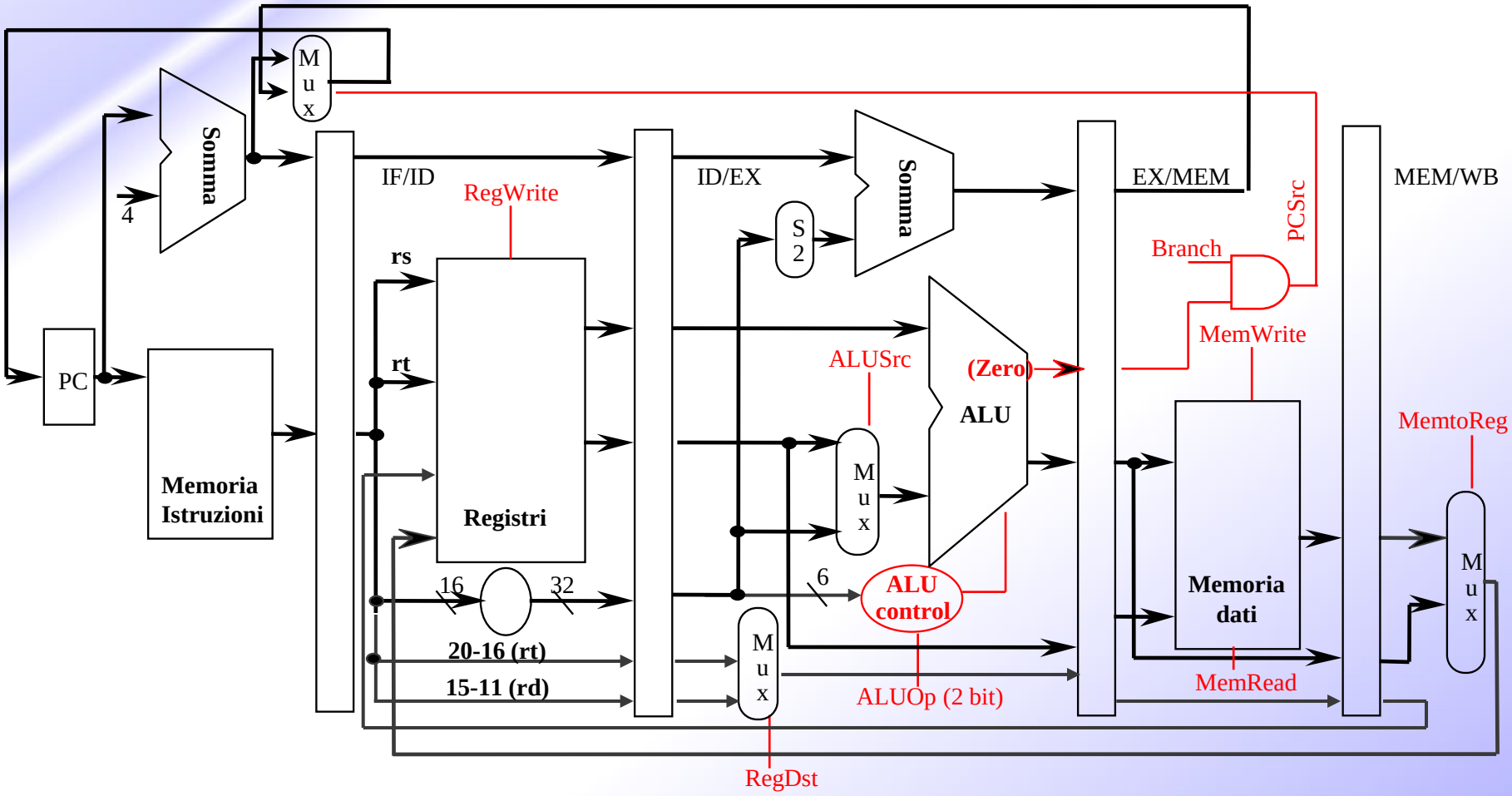
Controllo

Gli otto segnali di controllo

- Multiplexer
  - Input ALU (ALUSrc) – 1 bit
  - Input File Registri (RegDst) – 1 bit
  - Registro risultato (MemtoReg) – 1 bit
- Scrittura File Registri (RegWrite) – 1 bit
- Controllo ALU (ALUOp) – 2 bit
- Scrittura Memoria Dati (MemWrite) – 1 bit
- Lettura Memoria Dati (MemRead) – 1 bit
- Salto (Branch) – 1 bit

- Semplificazione:
  - Ignoriamo criticità per adesso
  - PC e pipeline registers scritti ad ogni ciclo di clock

Segnali di controllo per fase



Implementare il controllo/1

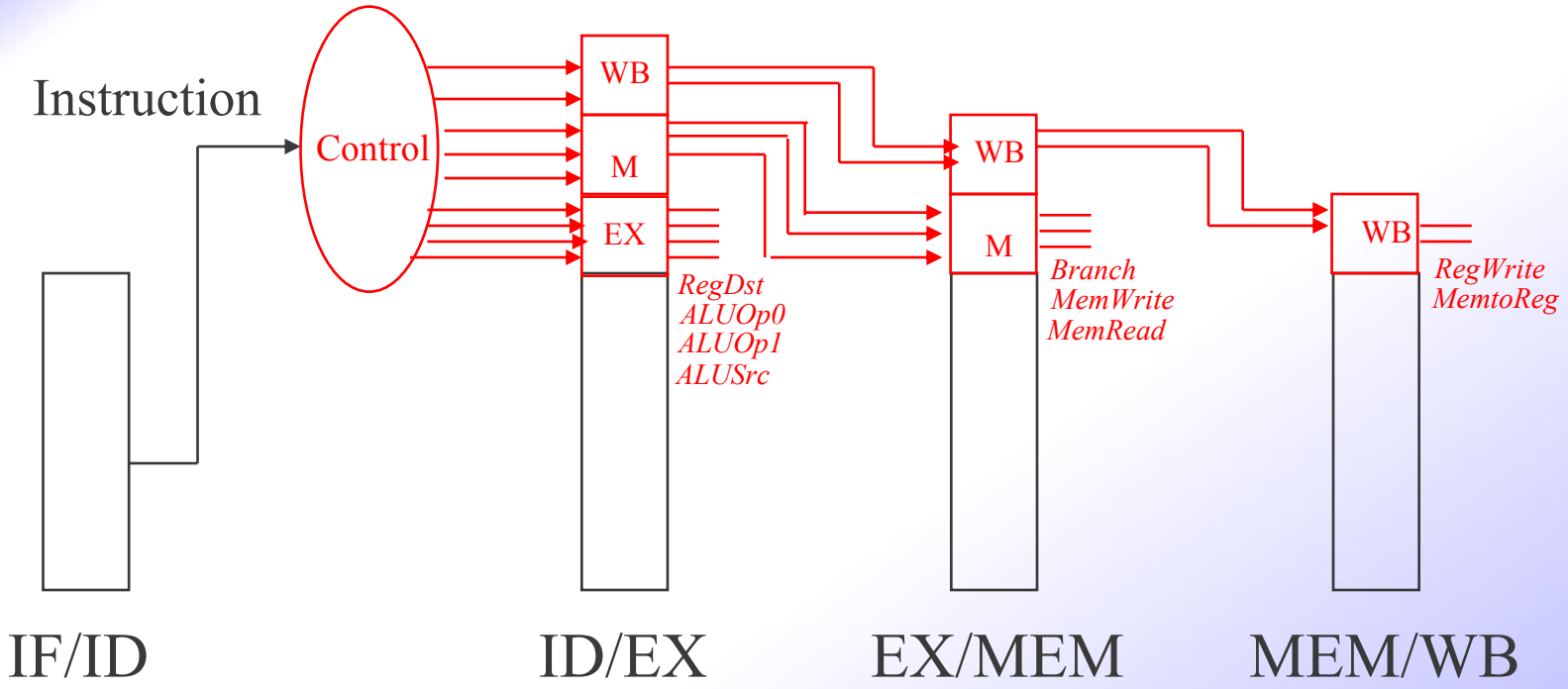
- La tabella di verità per i 9 bit di controllo è la stessa della CPU a ciclo di clock singolo

  Prende come input i campi Op (6 bit) e Funct (6 bit) dell'istruzione

- Tuttavia, i segnali vanno raggruppati in **tre insiemi**, corrispondenti alle **tre fasi** in cui essi sono usati: EX, MEM, WB

- Come tali, i segnali vengono **creati** nella fase di ID, e **passati** lungo la pipeline.
  - Questo richiede l'estensione dei pipeline registers.

Implementare il controllo/2



Esecuzione della pipeline

- Consideriamo le seguenti cinque istruzioni

```
lw      $s1,  20($t1)
sub     $s2,  $t2,  $t3
and     $s3,  $t4,  $t5
or      $s4,  $t6,  $t7
add     $s5,  $t8,  $t9
```

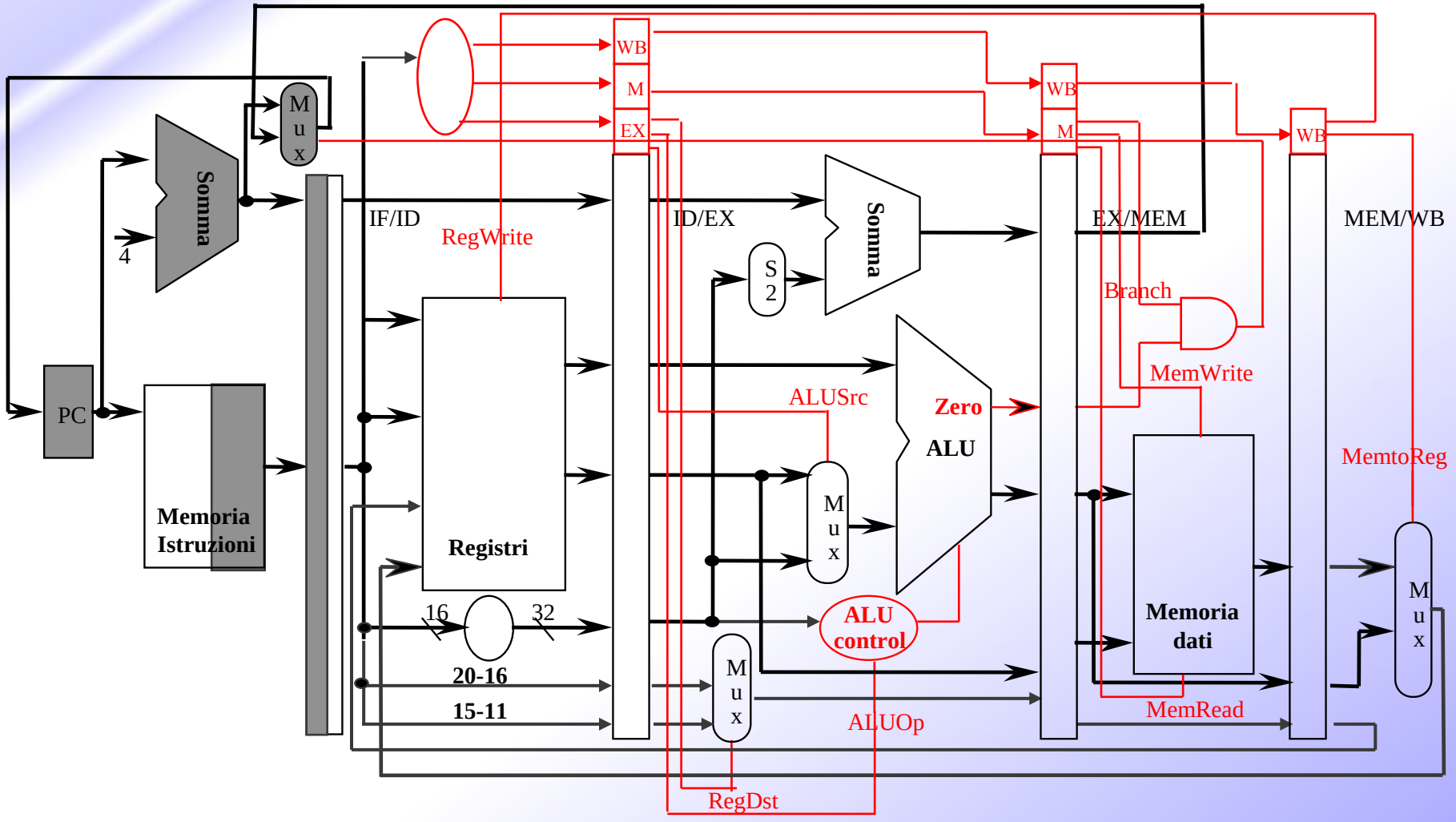
Ciclo 1

Stadio EX:
RegDst → 0
ALUOp1 → 0
ALUOp0 → 0
ALUSrc → 0

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 0

Stadio WB:
RegWrite → 0
MemtoReg → 0

lw $s1 20($t1)

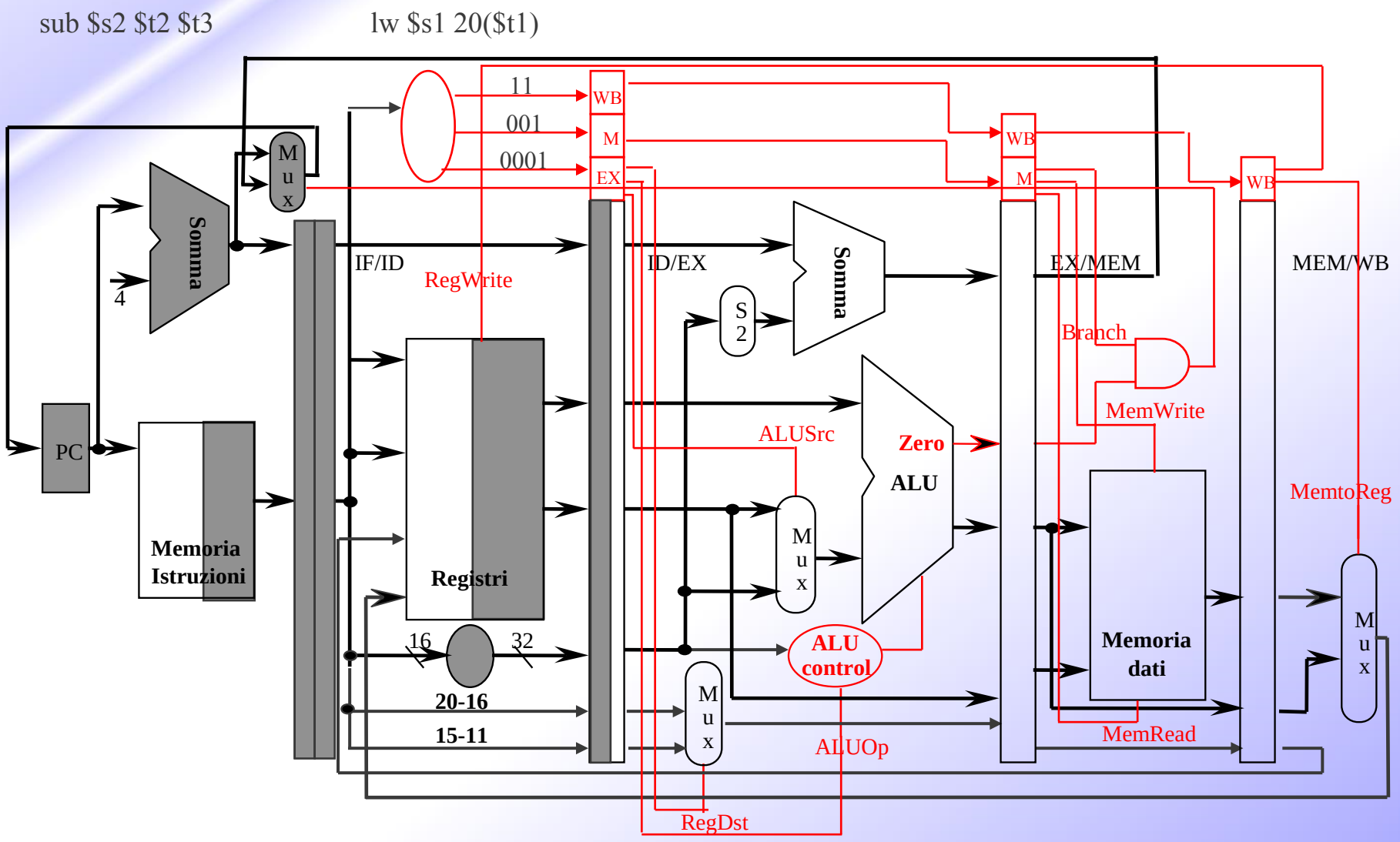


Ciclo 2

Stadio EX:
RegDst → 0
ALUOp1 → 0
ALUOp0 → 0
ALUSrc → 0

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 0

Stadio WB:
RegWrite → 0
MemtoReg → 0

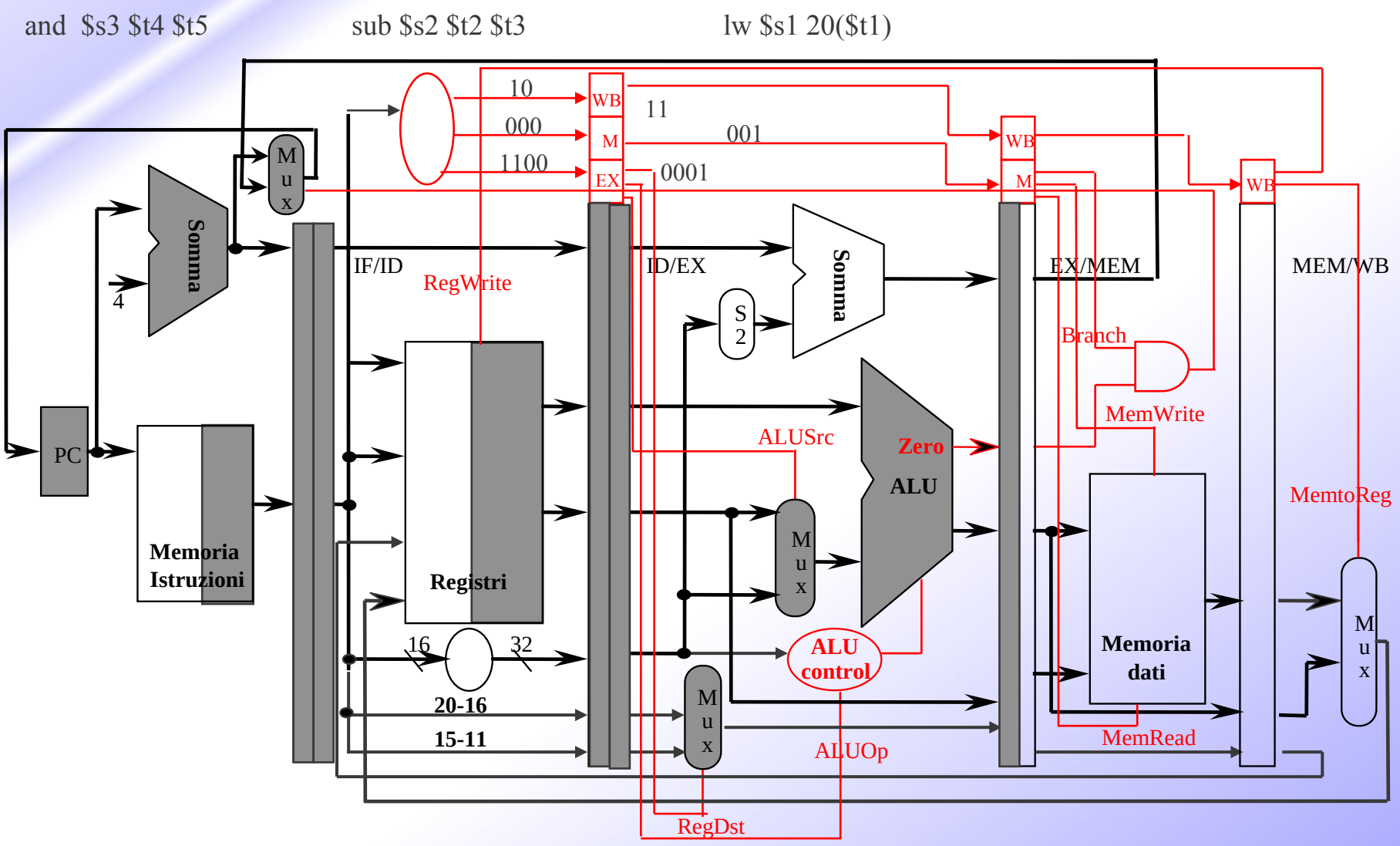


Ciclo 3

Stadio EX:
RegDst → 0
ALUOp1 → 0
ALUOp0 → 0
ALUSrc → 1

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 0

Stadio WB:
RegWrite → 0
MemtoReg → 0

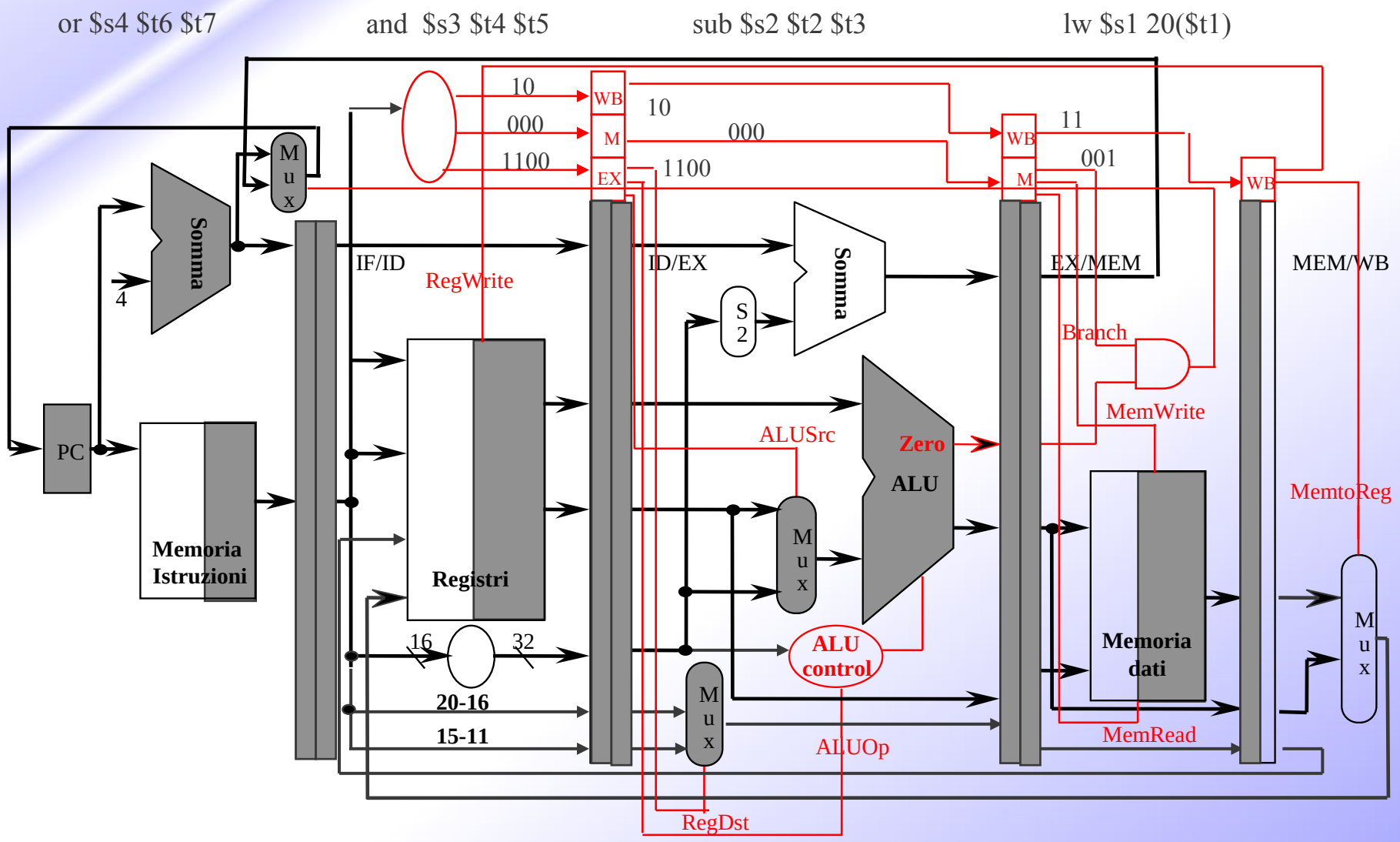


Ciclo 4

Stadio EX:
RegDst → 1
ALUOp1 → 1
ALUOp0 → 0
ALUSrc → 0

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 1

Stadio WB:
RegWrite → 0
MemtoReg → 0

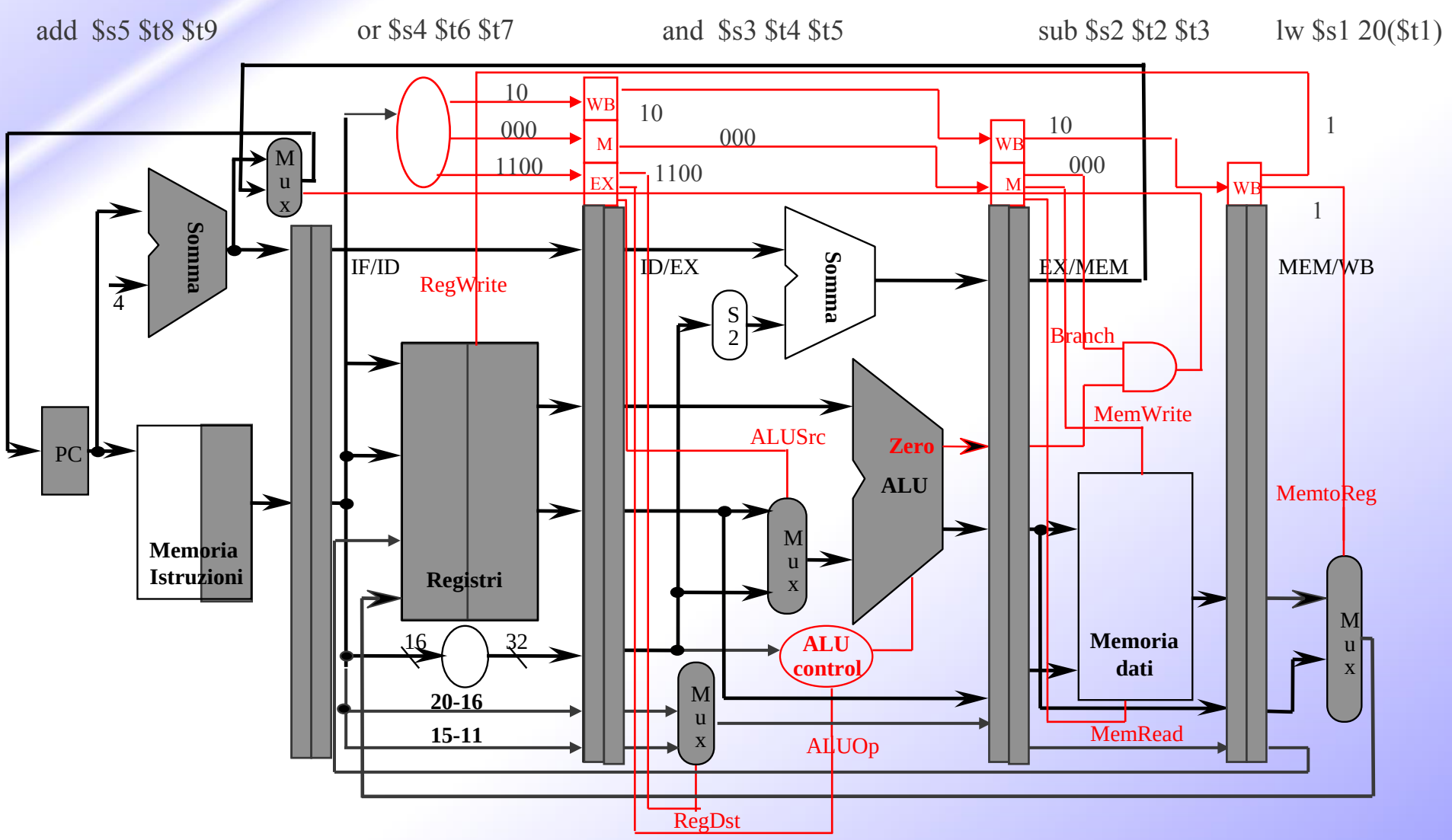


Ciclo 5

Stadio EX:
RegDst → 1
ALUOp1 → 1
ALUOp0 → 0
ALUSrc → 0

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 0

Stadio WB:
RegWrite → 1
MemtoReg → 1

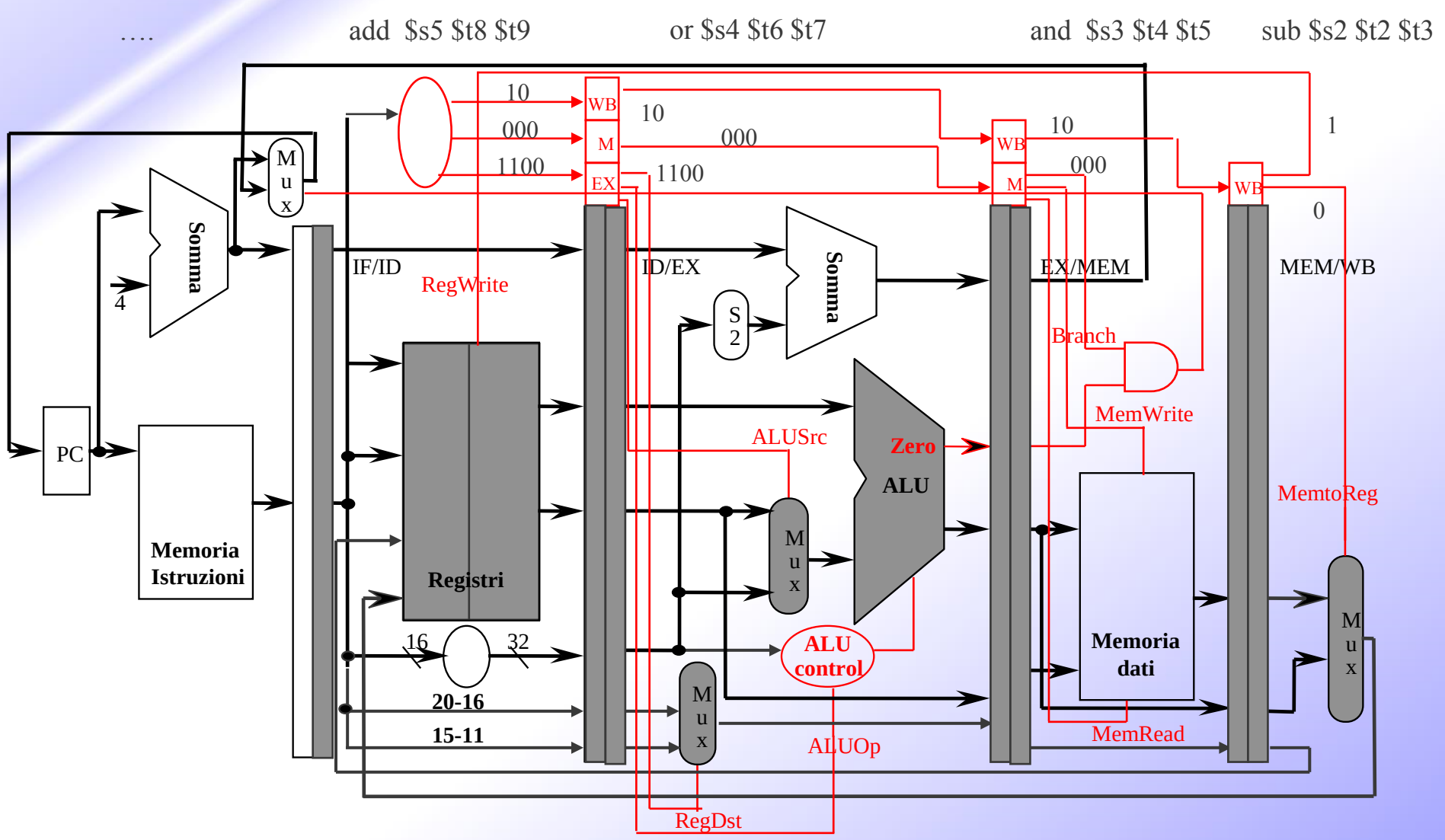


Ciclo 6

Stadio EX:
RegDst → 1
ALUOp1 → 1
ALUOp0 → 0
ALUSrc → 0

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 0

Stadio WB:
RegWrite → 1
MemtoReg → 0

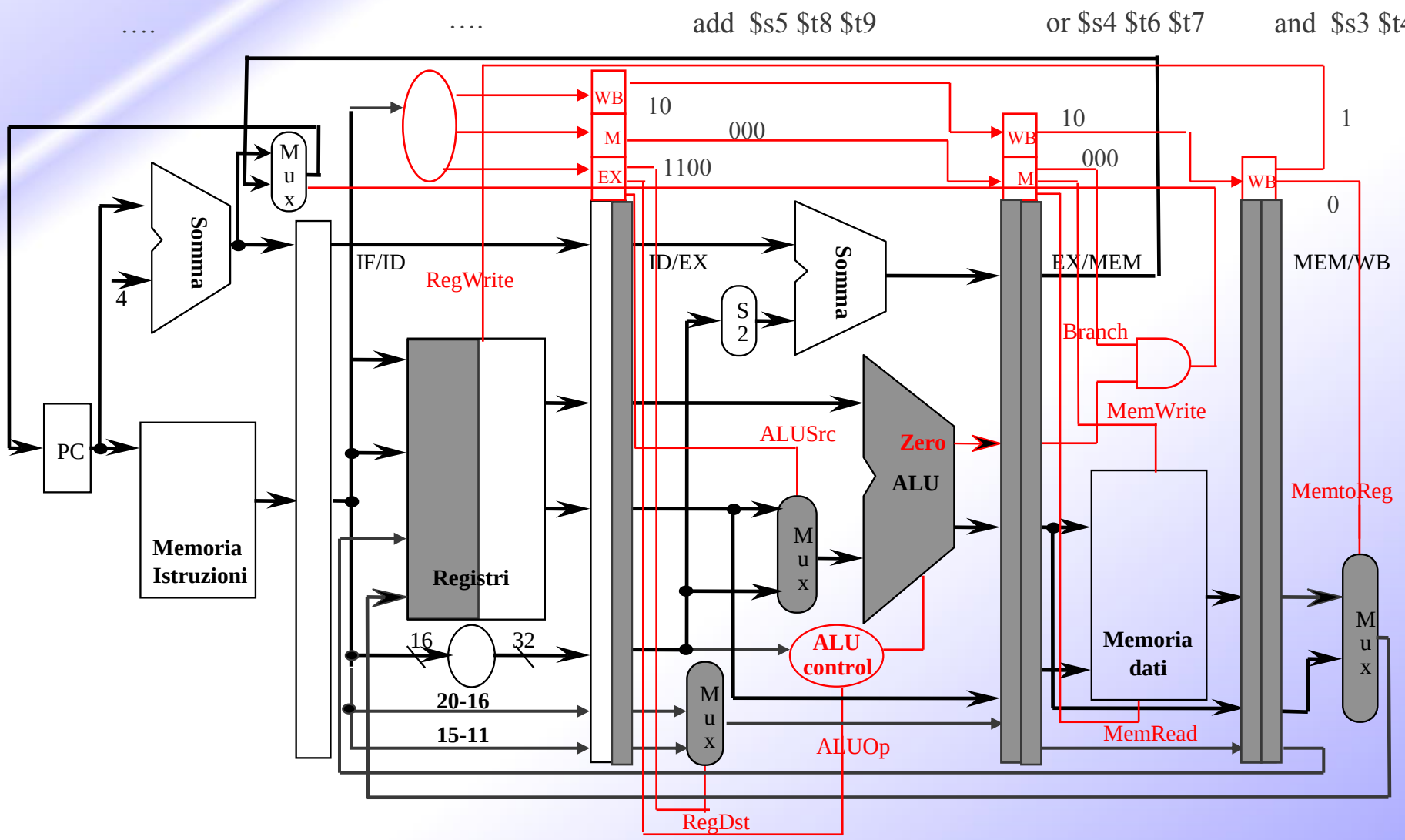


Ciclo 7

Stadio EX:
RegDst → 1
ALUOp1 → 1
ALUOp0 → 0
ALUSrc → 0

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 0

Stadio WB:
RegWrite → 1
MemtoReg → 0

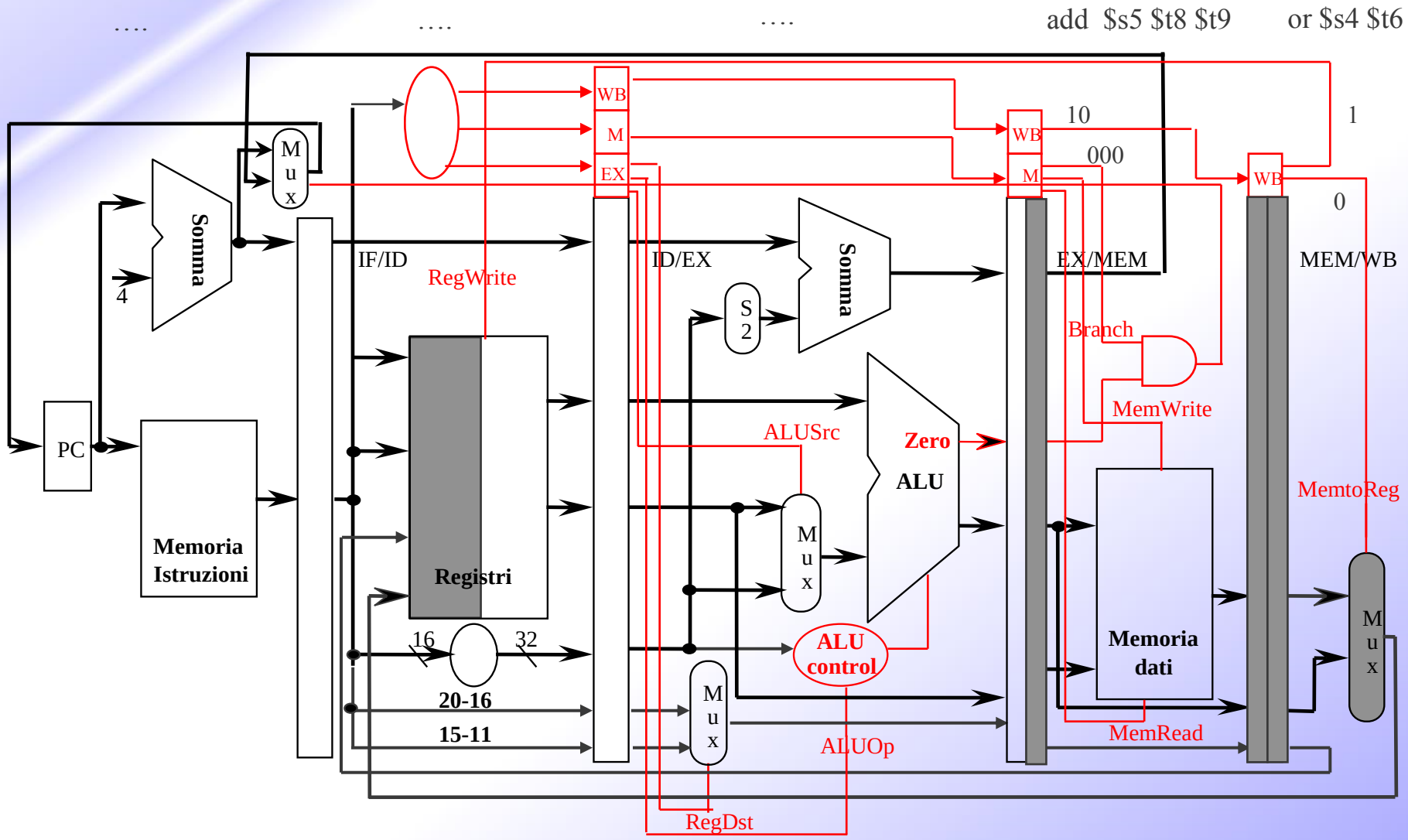


Ciclo 8

Stadio EX:
RegDst → 0
ALUOp1 → 0
ALUOp0 → 0
ALUSrc → 0

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 0

Stadio WB:
RegWrite → 1
MemtoReg → 0

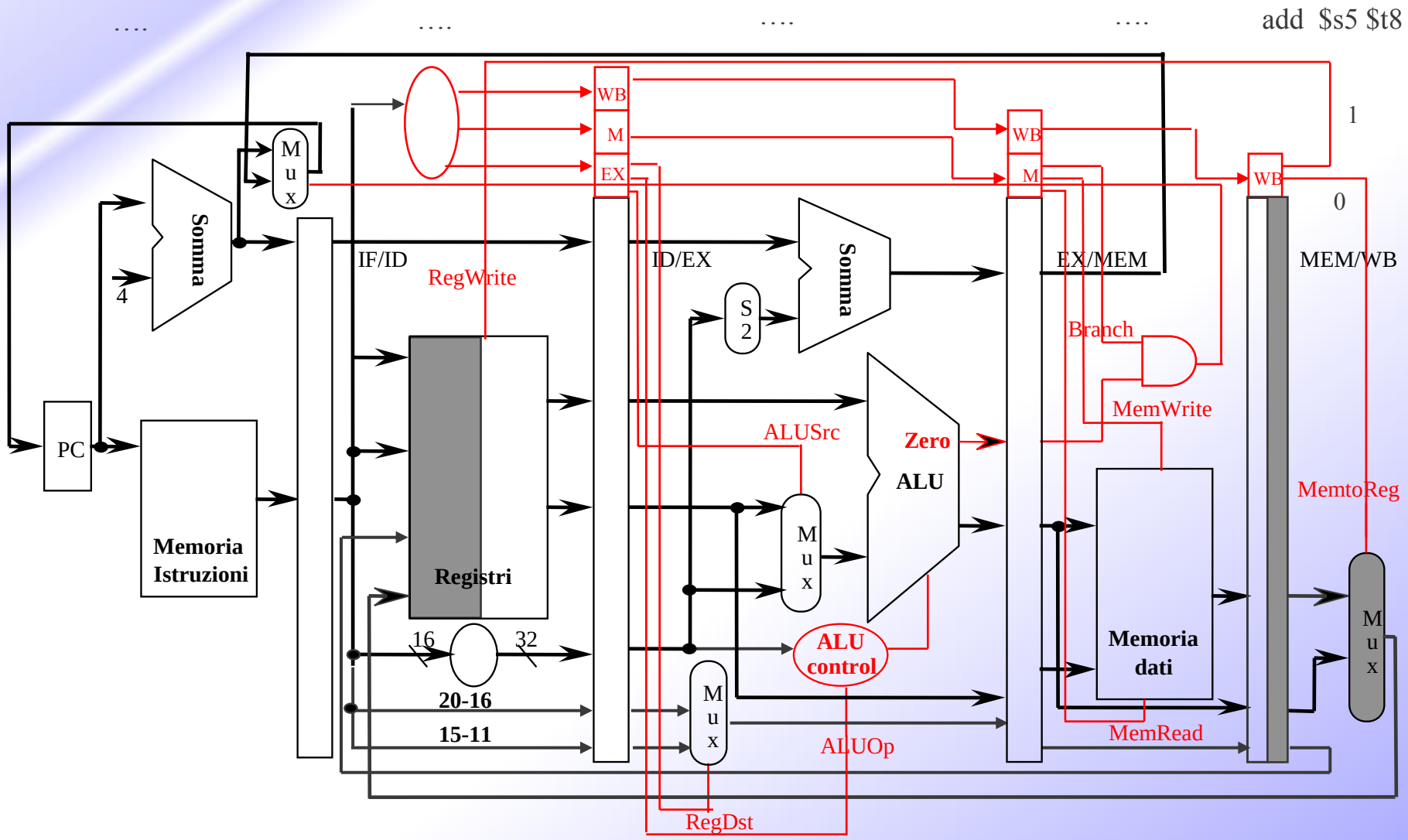


Ciclo 9

Stadio EX:
RegDst → 0
ALUOp1 → 0
ALUOp0 → 0
ALUSrc → 0

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 0

Stadio WB:
RegWrite → 1
MemtoReg → 0

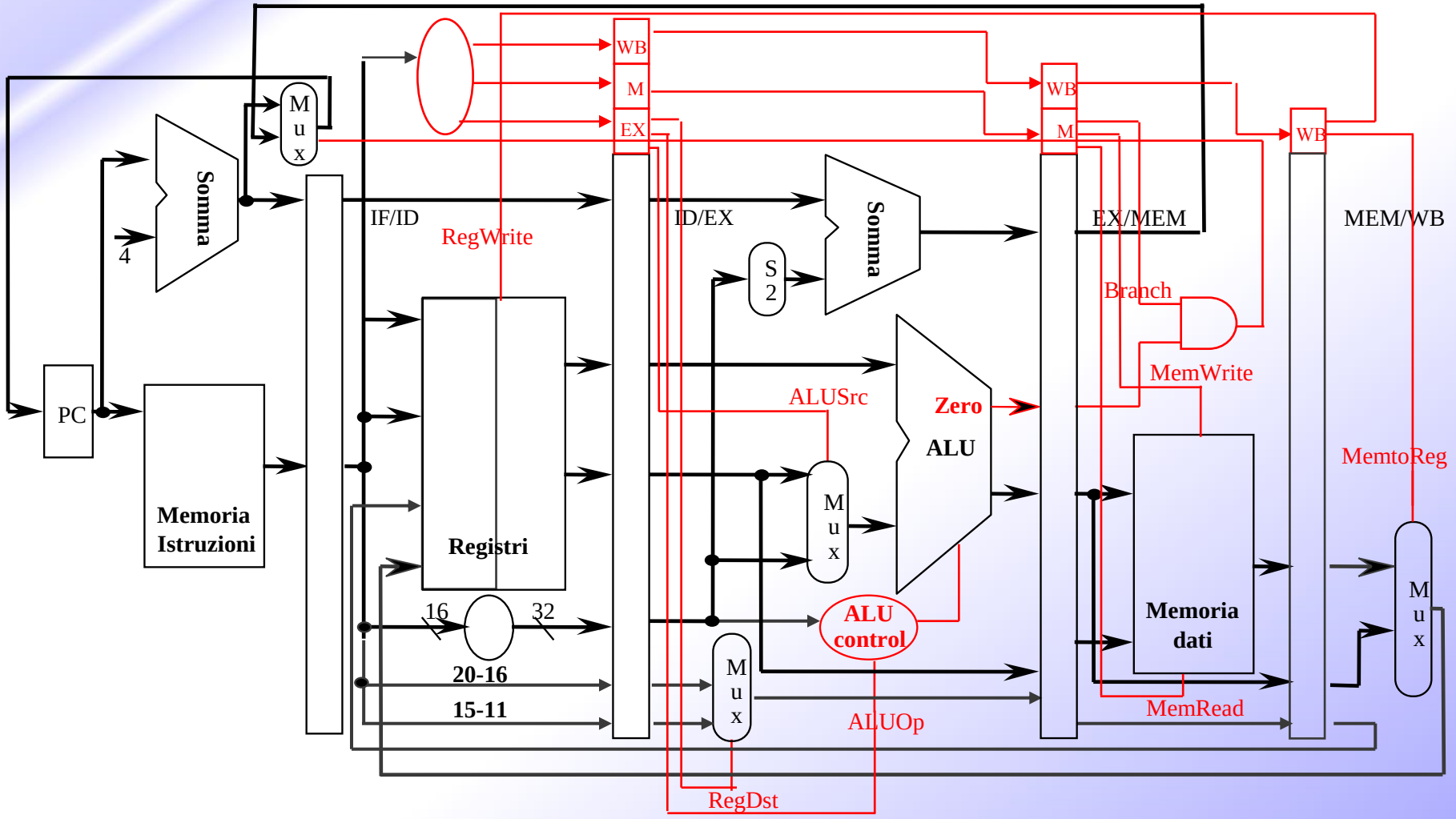


Ciclo 10

Stadio EX:
RegDst → 0
ALUOp1 → 0
ALUOp0 → 0
ALUSrc → 0

Stadio MEM:
Branch → 0
MemWrite → 0
MemRead → 0

Stadio WB:
RegWrite → 0
MemtoReg → 0

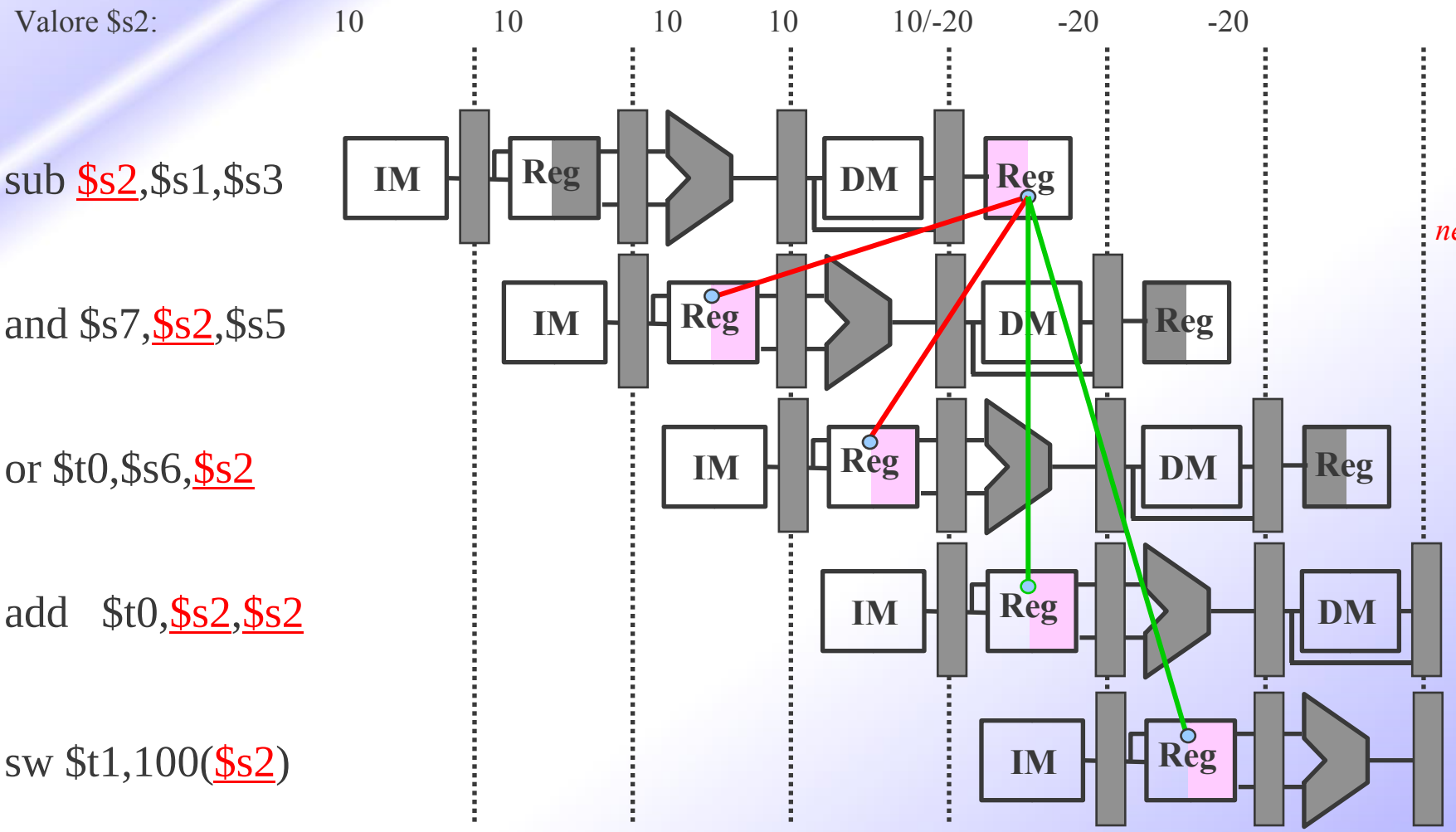


Propagazione e Stallo

Assumiamo che:
- $s2=10 prima della sub
- $s2=-20 dopo la sub

Criticità sui dati

Cosa c'e' di strano nella figura?



Hazard — No Hazard

Funzionamento File Registri

- Il File Registri viene
  - **scritto** nella **prima metà** del ciclo di clock e
  - **letto** nella **seconda metà** del ciclo di clock
- **Assunzione** di funzionamento: quando il F.R. viene letto, **il valore *aggiornato* è già disponibile**

- Dunque non c'è criticità nella 4^ istruzione

```
add  $t0, $s2, $s2
```

I Soluzione (sw): Riordino Codice

- Inserire istruzioni sicuramente indipendenti
  - Es., Istruzioni nop (no operation)
- Esempio precedente:

```
sub $s2,$s1,$s3
nop
nop
and $s7,$s2,$s5
or $t0,$s6,$s2
add  $t0,$s2,$s2
sw $t1,100($s2)
```

- In pratica, questo metodo non dà risultati soddisfacenti

II Soluzione (hw): Propagazione

Idea Base

Quando una istruzione intende scrivere un registro (add, lw), il risultato è già disponibile a partire dalla fine della fase EX (per la add) o MEM (per la lw) dell'istruzione

Propagazione

Senza aspettare il WB, quando necessario, passare il risultato all'input della fase EX (ALU) di una istruzione che segue

Propagazione: Esecuzione

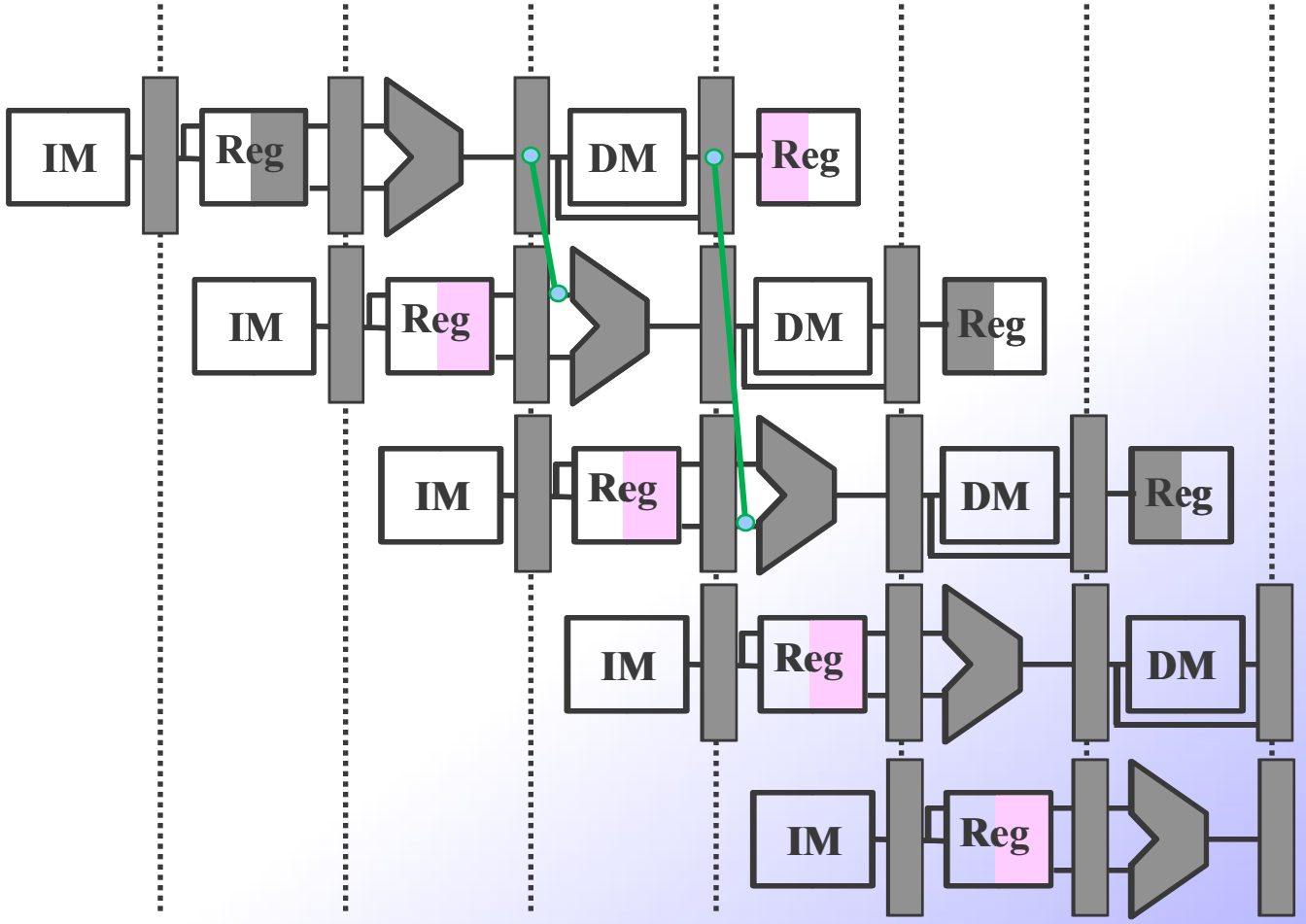
sub $s2,$s1,$s3

and $s7,$s2,$s5

or $t0,$s6,$s2

add $t0,$s2,$s2

sw $t1,100($s2)



Rilevare la Criticità Dati

- Notazione per specificare i campi dei registri di pipeline
  - Esempio:

    ID/EX.RegisterRs indica il campo Rs del registro ID/EX
- Le criticità è conveniente rilevarle **nella fase EX** di ogni istruzione. Condizioni che generano un hazard sui dati:

  - 1a: EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b: EX/MEM.RegisterRd = ID/EX.RegisterRt

  - 2a: MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b: MEM/WB.RegisterRd = ID/EX.RegisterRt

Esempio

sub $s2, $s1, $s3

and $s7, $s2, $s5

tipo 1a: EX/MEM.RegisterRd = ID/EX.RegisterRs

or $t0, $s6, $s2

tipo 2b: MEM/WB.RegisterRd = ID/EX.RegisterRt

add $t0, $s2, $s2

no hazard: il Register File fornisce i dati corretti già durante lo stadio ID dell'istruzione add

sw $t1, 100($s2)

no hazard: sw legge $s2 nel ciclo di clock successivo a quello in cui sub scrive in $s2

Casi speciali

Non si ha criticità, e dunque propagazione, nei seguenti casi:

- l'istruzione da cui il dato dipende non intende scrivere
  - => Controllare che il segnale RegWrite sia asserito (EX/MEM.RegWrite==1)
- scrittura sul registro $0 ($zero)
  - => Non propagare il valore potenzialmente diverso da 0 (EX/MEM.RegisterRd!=0)