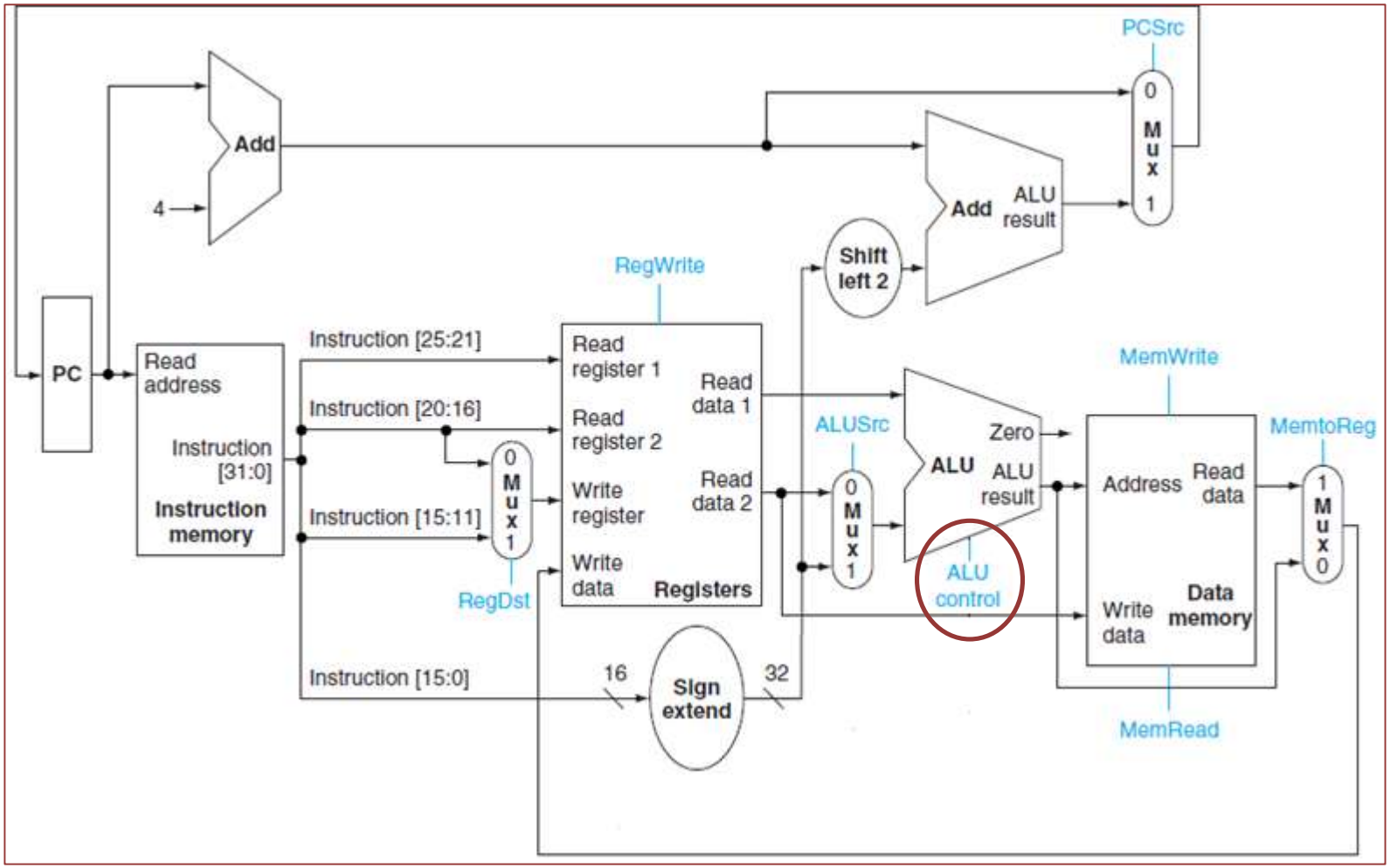
Logica di controllo della CPU MIPS a ciclo singolo

Michele Favalli

Logica di controllo

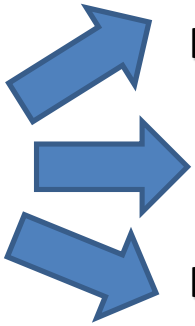
- Le operazioni svolte dal data-path per ciascun tipo di istruzione costituiscono la specifica della logica di controllo
- Tale logica deve fornire i valori dei bit di controllo per
  - ALU
  - Multiplexer
  - Registri e memoria

Controllo per la ALU



Controllo dell'ALU

- **Ricordiamo che la ALU compie operazioni diverse a seconda della CLASSE di istruzioni:**



Istruzioni di accesso alla memoria: fa una somma

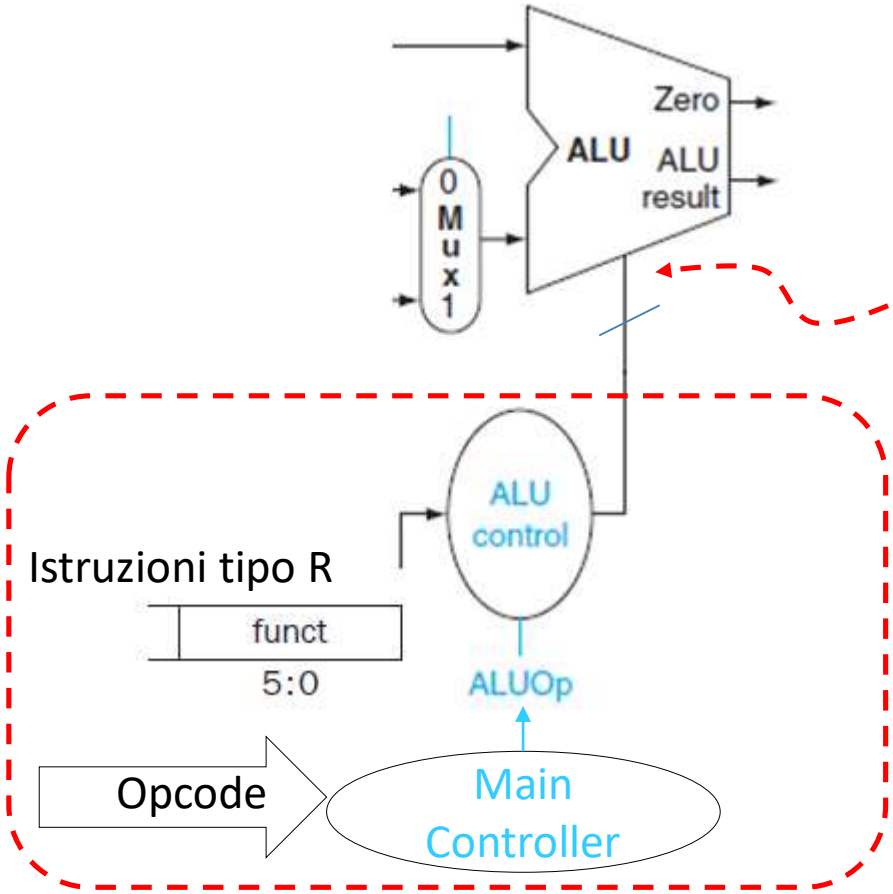
Istruzioni di BRANCH (beq): fa una sottrazione

Istruzioni Aritmetico-Logiche: dipende dal campo FUNCT

- **Consideriamo il seguente sottoinsieme delle operazioni aritmetico-logiche che una ALU può realizzare:** *and, or, add, sub, slt.*
- **Ipotizziamo che la ALU abbia 4 ingressi (faccia cioè 16 operazioni). Ne usiamo quindi 5 in questo esempio ridotto:**

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than

Controllo dell'ALU



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than

I 4 bit di controllo dipendono sia dal tipo di istruzione (fornito dai segnali ALUOp, provenienti dal MAIN CONTROLLER) sia dal campo FUNCT delle istruzioni di tipo R (**CONTROLLO GERARCHICO**)

ADD: eseguita con

- ✓ istruzioni aritmetiche e FUNCT=32
- ✓ istruzioni di accesso alla memoria

SUB: eseguita con

- ✓ istruzioni aritmetiche e FUNCT=34
- ✓ istruzioni di branch

ALUOp

- ALUOp deve codificare tre classi di istruzioni (mem, branch e R), nel nostro esempio ridotto. Dunque, servono 2 bit.
  - L'Opcode dell'istruzione determina automaticamente il valore di ALUOp.

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

Il CONTROLLO GERARCHICO è un modo per ottimizzare la velocità della logica di controllo, che è tipicamente critica ai fini della performance.

Sintesi del Controllore di Primo Livello

- Occorre determinare la tabella della verità ai fini della sintesi:
  - Ingressi: 8 bit (2 bit di ALUOp, 6 bit di FUNCT)
  - Uscite: 4 bit (i bit di comando dell'ALU)
  - Riportare solo le entry della tabella che interessano, sulle possibili 2^8=256,…
  - …senza dimenticare di mettere i don't care!

Se ho branch, load e store, il valore di FUNCT non importa

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Solo le righe che attivano la ALU

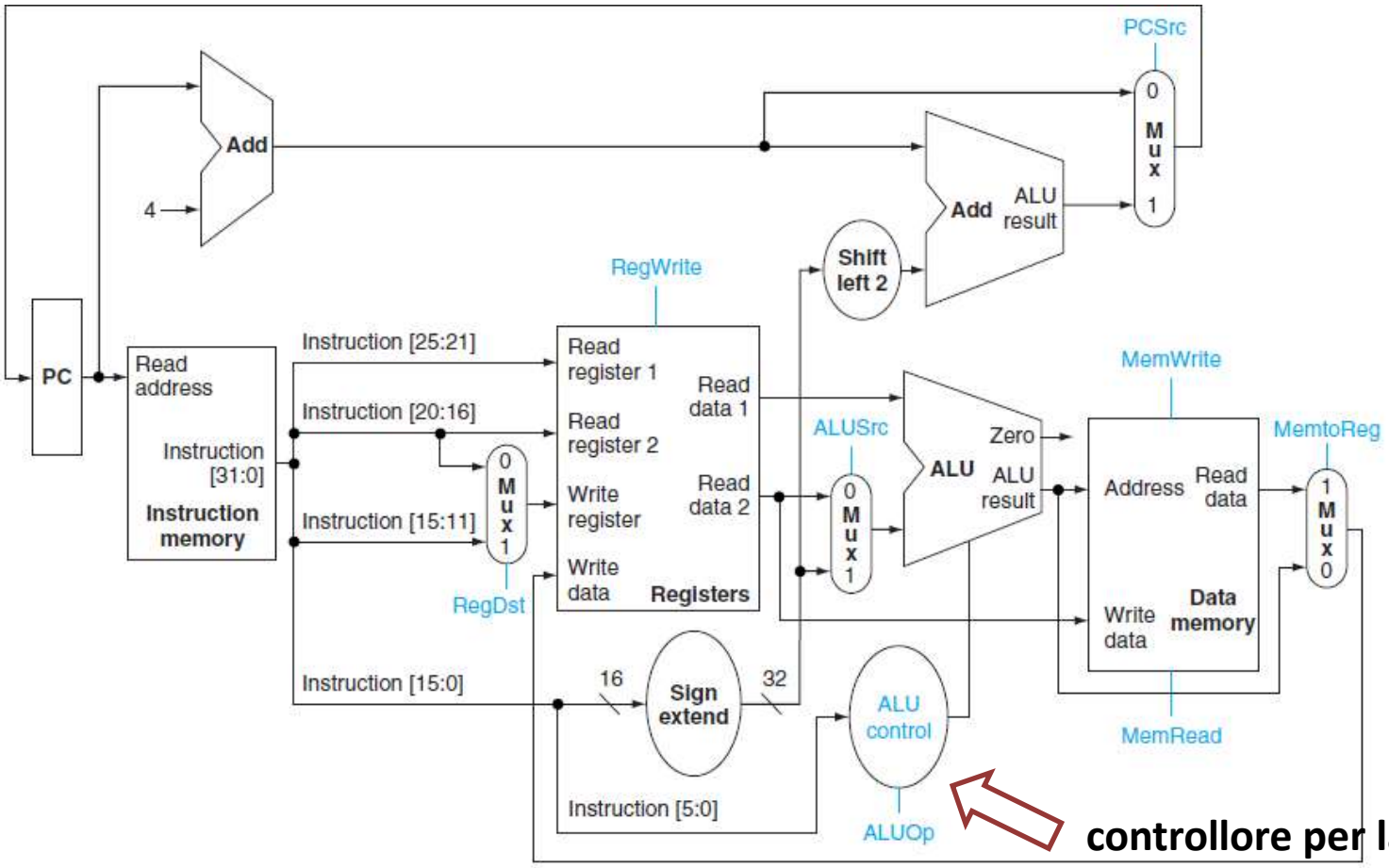
Questi due bit sono sempre «10», dunque irrilevanti

Dato che «11» non è ammesso, anziché indicare «10» posso indicare «1X»

Dato che «11» non è ammesso, se ho «X1» significa che è Branch!

Il problema è noto: sintesi di una rete logica di costo minimo da una tabella di verità con 8 ingressi e 4 uscite => potete provare a usare Logic Friday

Main controller



Il MAIN CONTROLLER può settare tutti i segnali di controllo tranne uno (PCSrc) basandosi solamente sul campo OPCODE dell'instruzione

Significato dei Segnali di Controllo

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Sorgente: J.L.Hennessy, D.A. Patterson, «Computer Organization and Design»

Main Controller

Opcode (6 bit)

Field	0					
Bit positions	31:26					

a. R-type instruction

Field	35 or 43					
Bit positions	31:26					

b. Load or store instruction

Field	4					
Bit positions	31:26					

c. Branch instruction

9 Segnali di Controllo (inclusi ALUOp)

Il MAIN CONTROLLER può settare tutti i segnali di controllo tranne uno basandosi solamente sul campo OPCODE dell'istruzione

Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Il problema è noto: realizzazione di costo minimo di una tabella di verità con 6 ingressi e 9 uscite.

Main Controller (PCsrc)

Opcode (6 bit)

Field	0
Bit positions	31:26

a. R-type instruction

Field	35 or 43
Bit positions	31:26

b. Load or store instruction

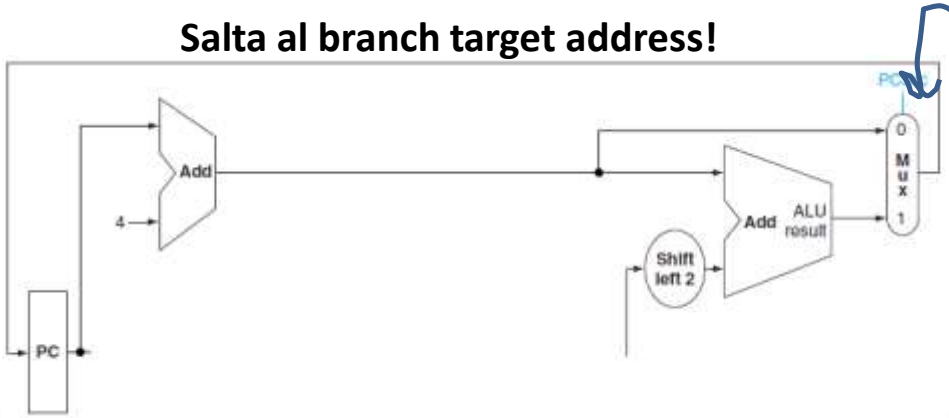
Field	4
Bit positions	31:26

c. Branch instruction

9 Segnali di Controllo (inclusi ALUOp)

Solo la selezione del branch target address tramite il segnale di controllo PCSrc richiede sia l'OPCODE sia l'uscita della ALU

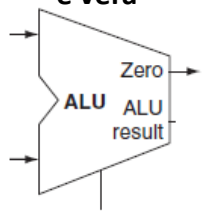
Salta al branch target address!



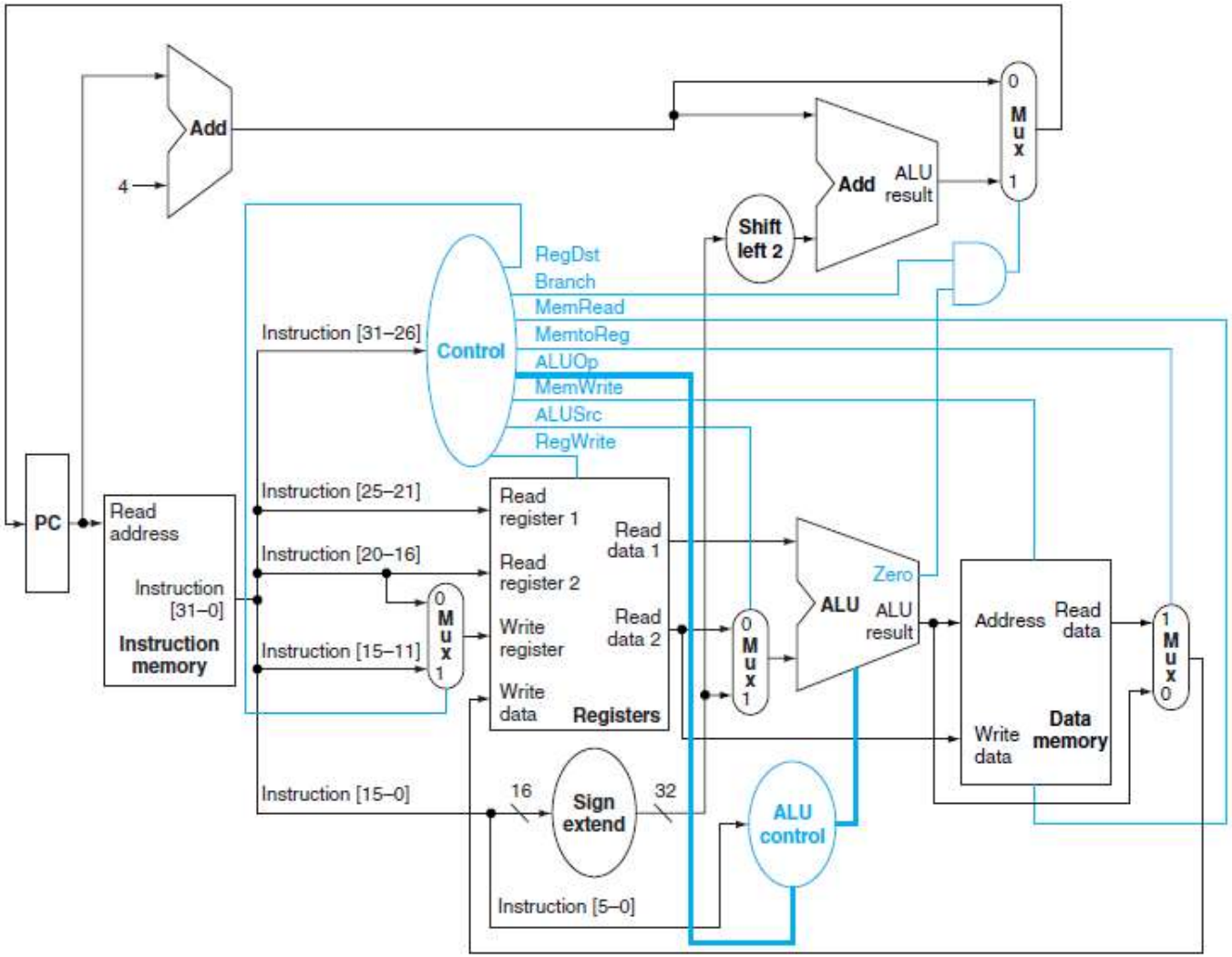
E' un branch

Global controller

La condizione è vera



Control Path e Data Path



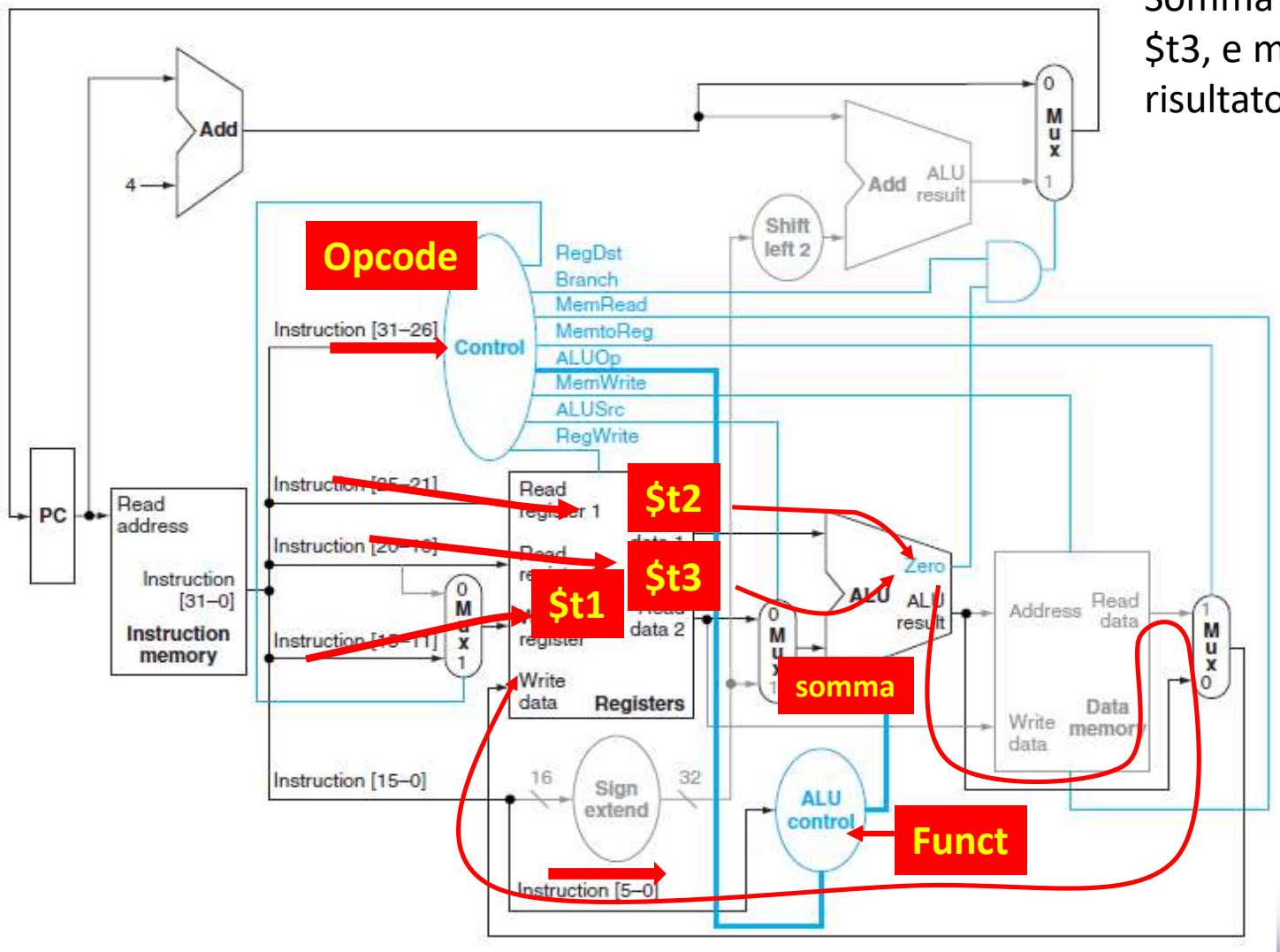
Segnali di controllo:

- 3 pilotano i multiplexers
- 3 segnali per controllare read e write nel register file e nella memoria dati
- 1 segnale che indica se si tratta di un branch oppure no
- 2 segnali per il controllo del controllore dell'ALU (ALUOp)

Istruzione di tipo R

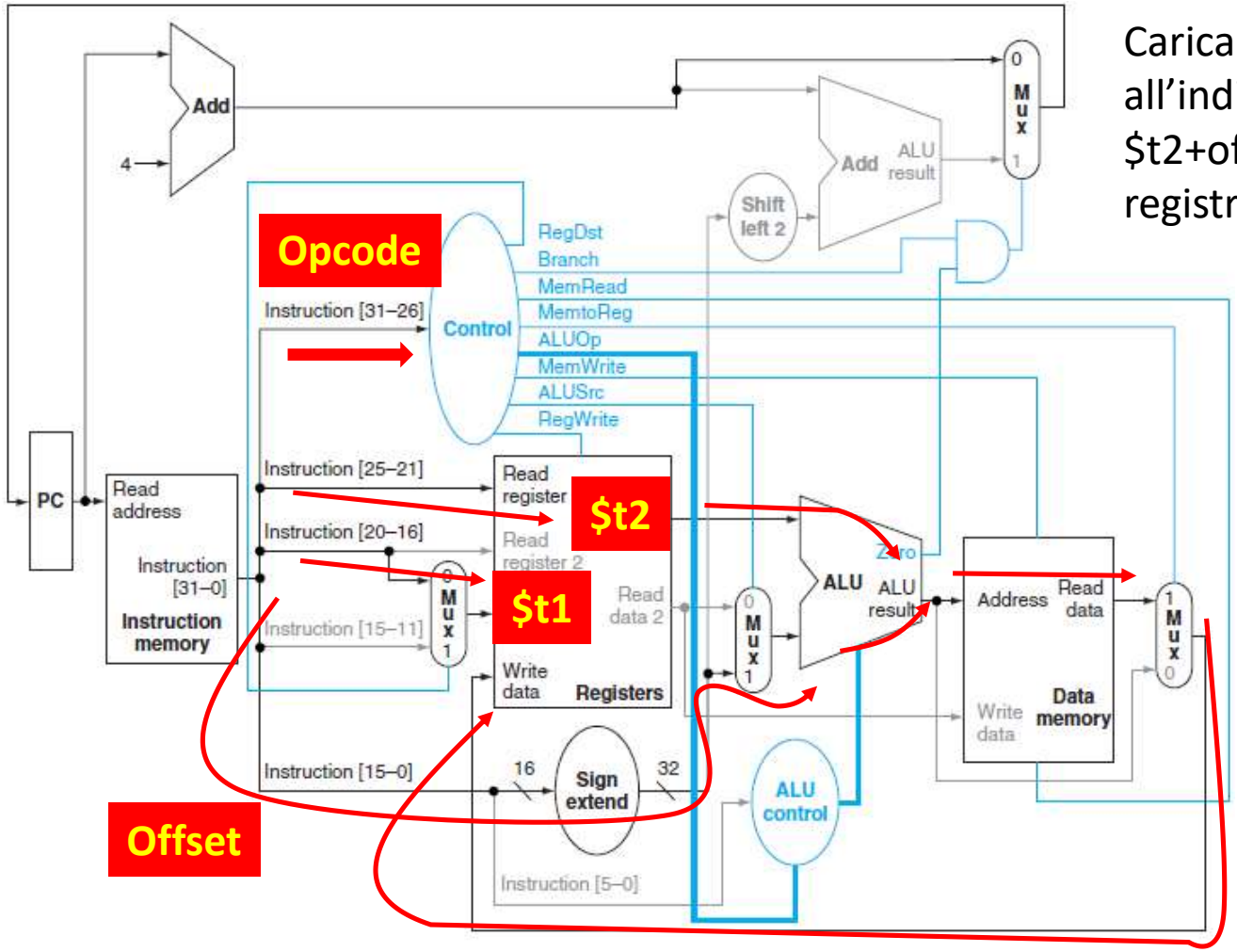
```
add $t1,$t2,$t3
```

Somma $t2 con $t3, e metti il risultato in $t1.



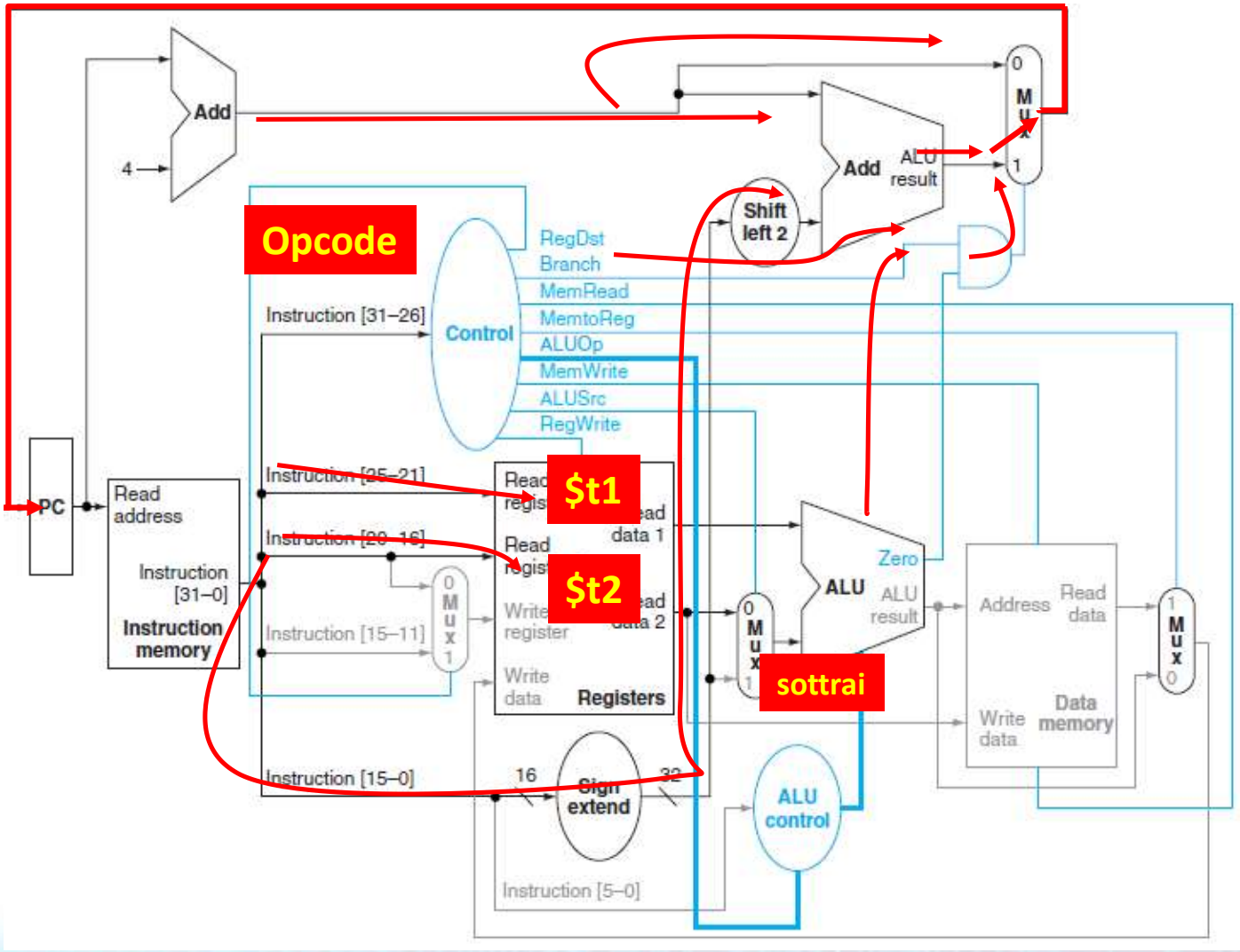
Istruzione Load

```
lw $t1, offset($t2)
```



Carica la parola all'indirizzo $t2+offset nel registro $t1

Istruzione «branch equal» `beq $t1,$t2,offset`



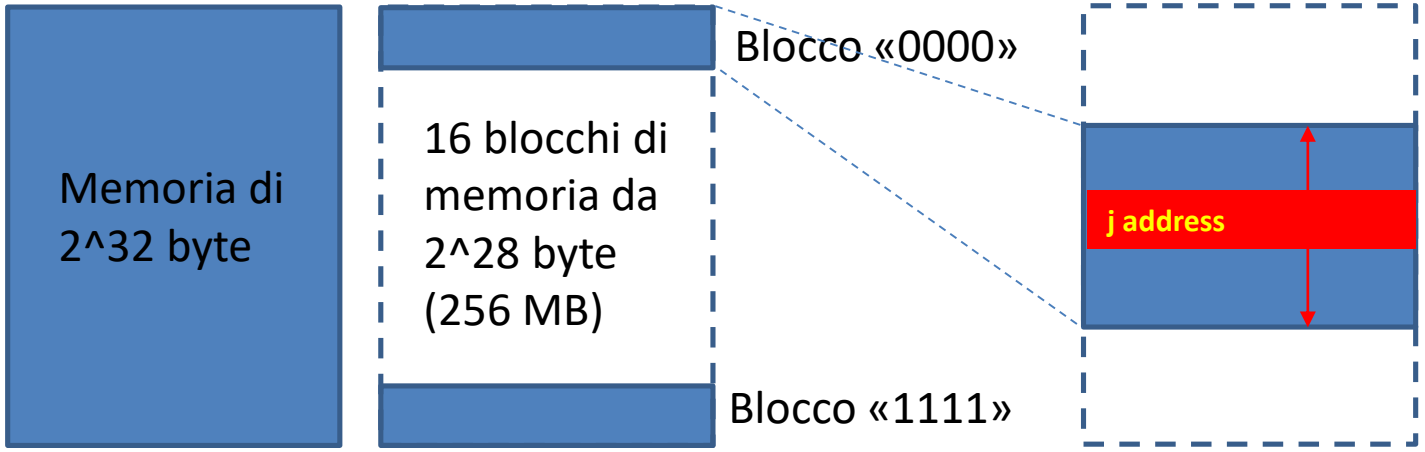
Salta a «PC+4+offset» se i contenuti dei registri $t1 e$t2 coincidono

Istruzione jump

j address

Field	000010	address
Bit positions	31:26	25:0

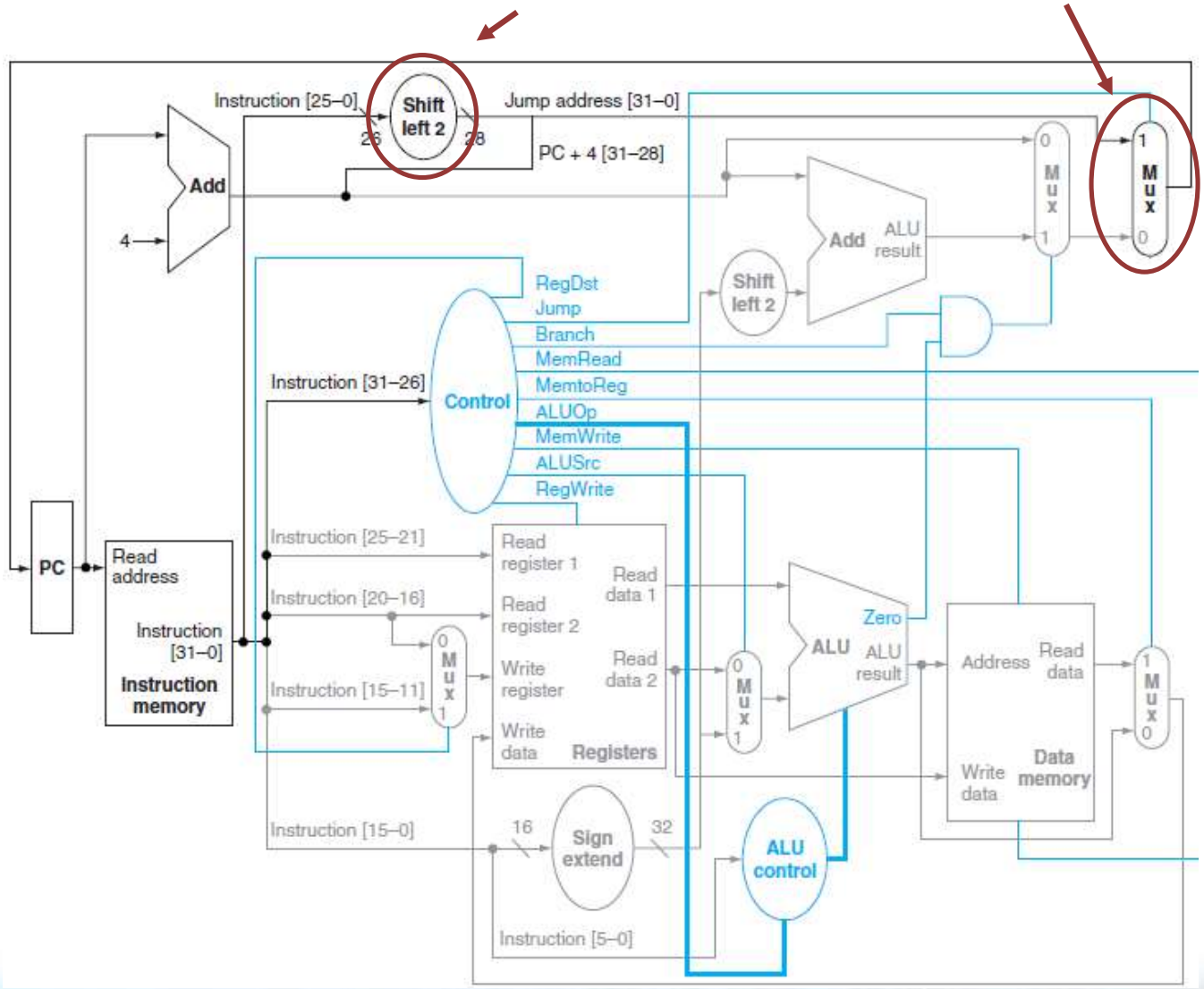
- Occorre comprendere bene l'indirizzamento di questa istruzione:
  - «address» è un indirizzo assoluto con la granularità della parola, non del byte
  - Occorre moltiplicarlo per 4 per ottenere l'indirizzamento a byte, cioè occorre fare lo «shift left» di 2 posizioni, inserendo «00» in posizione meno significativa.
  - Da 26 bit si passa così a 28 bit. Da dove vengono i rimanenti 4 bit?
  - I 4 bit rimanenti provengono dai **bit più significativi di PC+4**



- *Se i primi 4 bit più significativi sono quelli del PC, significa che il salto mi fa rimanere dentro al blocco corrente*
- *«address»*4 consiste di 28 bit, e identifica un byte qualsiasi all'interno del blocco.*

Per jump all'esterno del blocco di memoria, il compilatore deve cambiare tecnica di indirizzamento (es., jr)

Supporto dell'istruzione «jump»



A. Moltiplica i 26 bit dell'indirizzo per 4, ottenendo un indirizzo a byte

B. Concatena i 28 bit della moltiplicazione con i 4 bit più significativi da PC+4

C. Aggiorna il Program Counter

Implementazione a singolo ciclo di clock

Il ciclo di clock è costretto ad avere la stessa lunghezza per ogni istruzione

- Esso sarà determinato dal «percorso» più lungo attraverso il sistema microprocessore- memoria
- Quasi certamente, l'operazione più onerosa è la LOAD che usa: memoria istruzioni + register file + ALU + memoria dati + register file
- Le istruzioni di di tipo R usano: memoria istruzioni+register file+ALU + register file
- Le altre istruzioni sono potenzialmente più veloci, quindi a parità di ciclo di clock generano «idleness»

Altra Limitazione: 1 sola istruzione in esecuzione per ciclo di clock (throughput limitato)

Possibili miglioramenti di una CPU a ciclo singolo

- CPU multiciclo (CPI>1)
  - in questa CPU le istruzioni più veloci impiegano 1 ciclo di clock, mentre quelle più lente ne possono utilizzare più di uno
- Pipelining
  - tecnica implementativa nella quale diversi blocchi funzionali della CPU lavorano nello stesso ciclo di clock su istruzioni diverse

CPU multiciclo vs ciclo singolo – I

- Ipotesi con valori di ritardo arbitrari
  - Memorie (dati/istr.): 1.2 ns
  - ALU: 0.6 ns
  - Register file: 0.2 ns
- Load => ritardo di 3.2 ns
- R-type => ritardo di 2.2 ns
- A ciclo singolo deve essere T_{clock}>=max{delay}, quindi va bene usare T_{clock}=3.2 ns
  - *Idleness: la R butta via 1 ns*
- Nella multiciclo potrei usare T_{clock}=1.1 ns
  - 3 cicli per la load e 2 per le R-type

Ritardi dei multiplexer, shifter ….
Margini sui ritardi ….

CPU multiciclo vs ciclo singolo – I

- A ciclo singolo deve essere T_{clock}>=max{delay}, quindi va bene usare T_{clock}=3.2 ns
- Nella multiciclo potrei usare T_{clock}=1.1 ns
  - 3 cicli per la load e 2 per le R-type
- Tempo per eseguire 1 load e 1 R
  - nella CPU a ciclo singolo = 6.4 ns (CPI = 1)
  - nella CPU multiciclo = 5.4ns (CPI medio = #R*2+#load*3=2.5)
- Il controllo non è più combinatorio, ma è una FSM: i segnali di controllo dipendono da quale ciclo sto considerando