SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Architettura degli Elaboratori

Introduzione della pipeline. Data e control hazards

Prof. Andrea Sterbini – sterbini@di.uniroma1.it

Argomenti

- Le 5 fasi dell’istruzione
- Introduzione della pipeline
  - Solo una fase è attiva in ogni istante
  - Pipeline per processare più istruzioni contemporaneamente
  - Hazards sui dati e sul controllo
- Soluzione esercizio

Argomenti

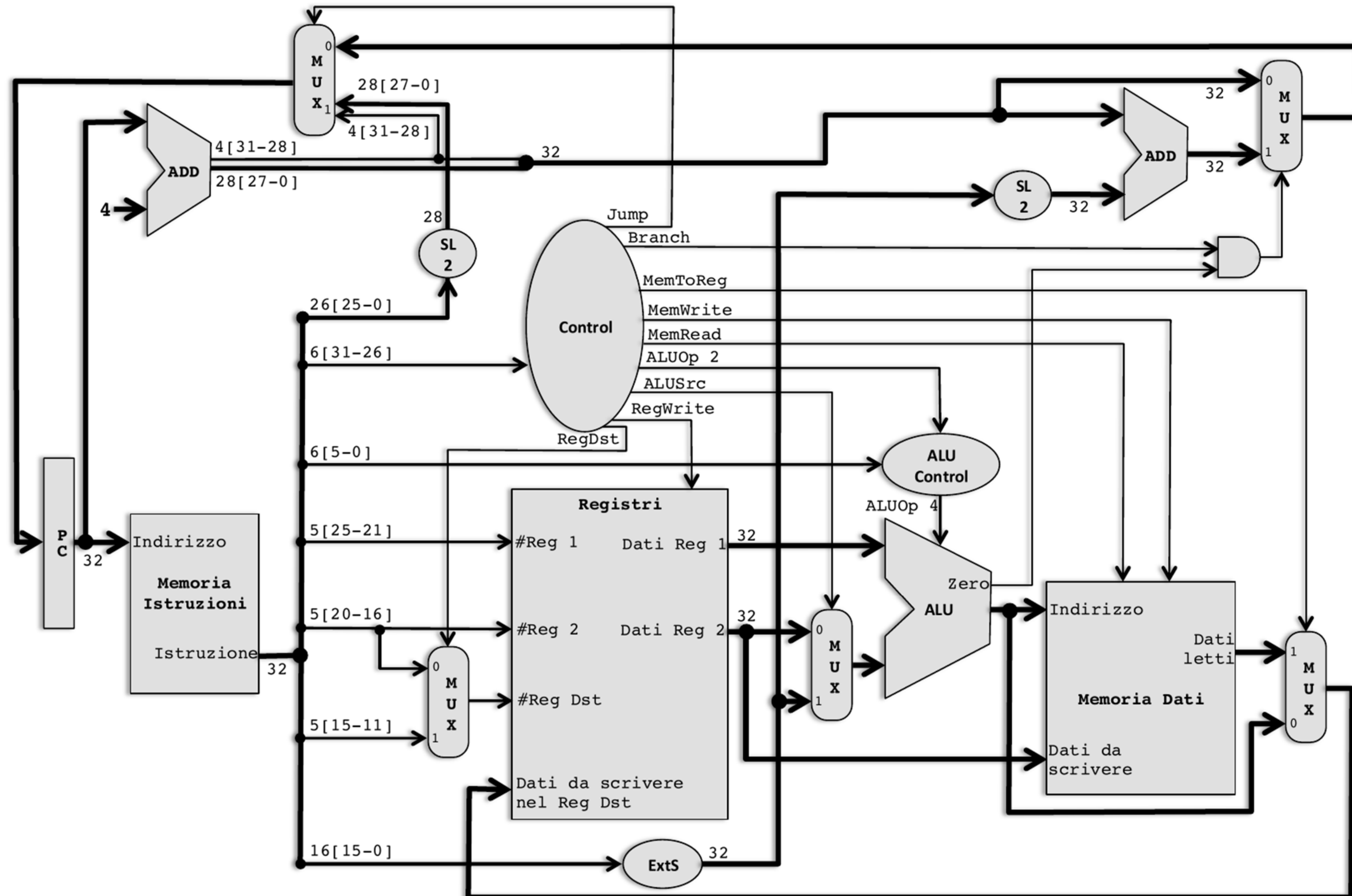
- Le 5 fasi dell'istruzione
- Introduzione della pipeline
  - Solo una fase è attiva in ogni istante
  - Pipeline per processare più istruzioni contemporaneamente
  - Hazards sui dati e sul controllo
- Soluzione esercizio

Fasi dell'istruzione:

- **Instruction Fetch**: si carica l'istruzione dalla memoria
- **Instruction Decode**: la CU decodifica l'istruzione e vengono letti gli argomenti dai registri
- **EXEcution**: l'ALU fa il calcolo necessario (tipo R o accesso alla memoria o branch)
- **MEMory access**: viene letta/scritta la memoria (lw e sw)
- **Write Back**: il risultato dell'ALU o quello letto dalla memoria viene messo nel registro destinazione

CPU MIPS a un ciclo di clock

Implementazione ad un ciclo di clock di MIPS (solamente le istruzioni: `add`, `sub`, `and`, `or`, `xor`, `slt`, `lw`, `sw`, `beq`, `j`)



La CPU è per l’80% inutilizzata!

- In ogni momento SOLO una unità funzionale è attiva e **le altre non fanno nulla!**:

- I.Fetch:	**Memoria Istruzioni (e aggiornamento PC)**
- I.Decode:	**Blocco registri (e CU)**
- Execute:	**ALU**
- MEM:	**Memoria dati**
- Write Back:	**Blocco registri**

La CPU è per l’80% inutilizzata!

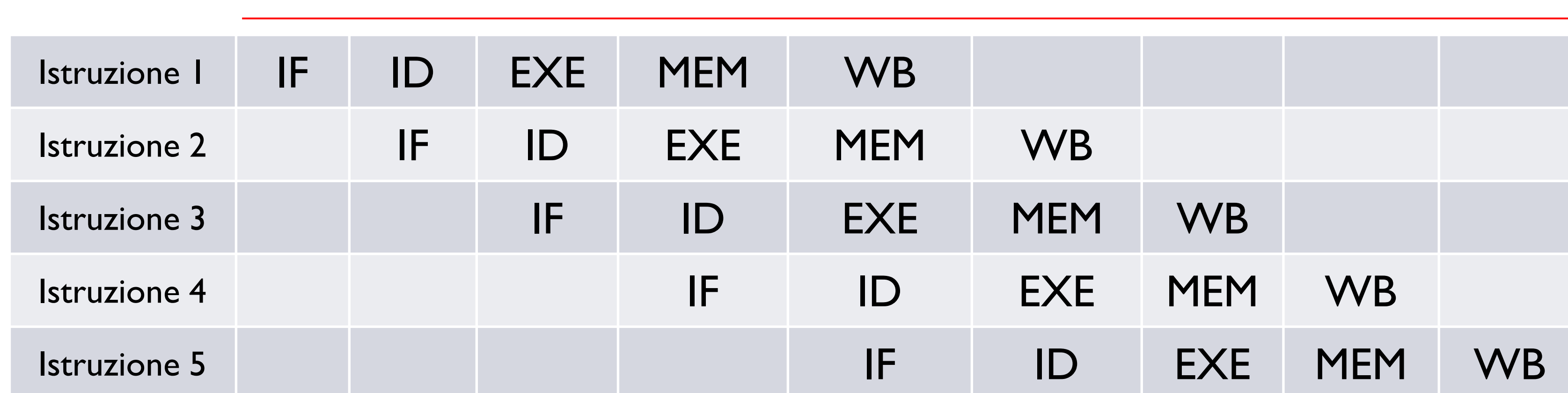
- In ogni momento SOLO una unità funzionale è attiva e **le altre non fanno nulla!**:
- I.Fetch: **Memoria Istruzioni (e aggiornamento PC)**
- I.Decode: **Blocco registri (e CU)**
- Execute: **ALU**
- MEM: **Memoria dati**
- Write Back: **Blocco registri**

Idea: trasformiamo la CPU in una CATENA DI MONTAGGIO, in cui ogni unità funzionale elabora la fase che gli corrisponde E POI PASSA L’ISTRUZIONE ALLA UNITA’ SUCCESSIVA

La CPU è per l'80% inutilizzata!

- In ogni momento SOLO una unità funzionale è attiva e **le altre non fanno nulla!**:
- I.Fetch: **Memoria Istruzioni (e aggiornamento PC)**
- I.Decode: **Blocco registri (e CU)**
- Execute: **ALU**
- MEM: **Memoria dati**
- Write Back: **Blocco registri**

Idea: trasformiamo la CPU in una CATENA DI MONTAGGIO, in cui ogni unità funzionale elabora la fase che gli corrisponde E POI PASSA L'ISTRUZIONE ALLA UNITA' SUCCESSIVA



Istruzione 1	IF	ID	EXE	MEM	WB				
Istruzione 2		IF	ID	EXE	MEM	WB			
Istruzione 3			IF	ID	EXE	MEM	WB		
Istruzione 4				IF	ID	EXE	MEM	WB	
Istruzione 5					IF	ID	EXE	MEM	WB

La CPU è per l'80% inutilizzata!

- In ogni momento SOLO una unità funzionale è attiva e **<u>le altre non fanno nulla!</u>**:
- I.Fetch: **Memoria Istruzioni (e aggiornamento PC)**
- I.Decode: **Blocco registri (e CU)**
- Execute: **ALU**
- MEM: **Memoria dati**
- Write Back: **Blocco registri**

Idea: trasformiamo la CPU in una CATENA DI MONTAGGIO, in cui ogni unità funzionale elabora la fase che gli corrisponde E POI PASSA L'ISTRUZIONE ALLA UNITA' SUCCESSIVA

Istruzione 1	IF	ID	EXE	MEM	WB				
Istruzione 2		IF	ID	EXE	MEM	WB			
Istruzione 3			IF	ID	EXE	MEM	WB		
Istruzione 4				IF	ID	EXE	MEM	WB	
Istruzione 5					IF	ID	EXE	MEM	WB

In questo modo <u>in ogni istante</u> **TUTTE le unità funzionali sono occupate** e svolgiamo <u>fino a 5 istruzioni contemporaneamente</u> (in fasi diverse)

Incremento della velocità

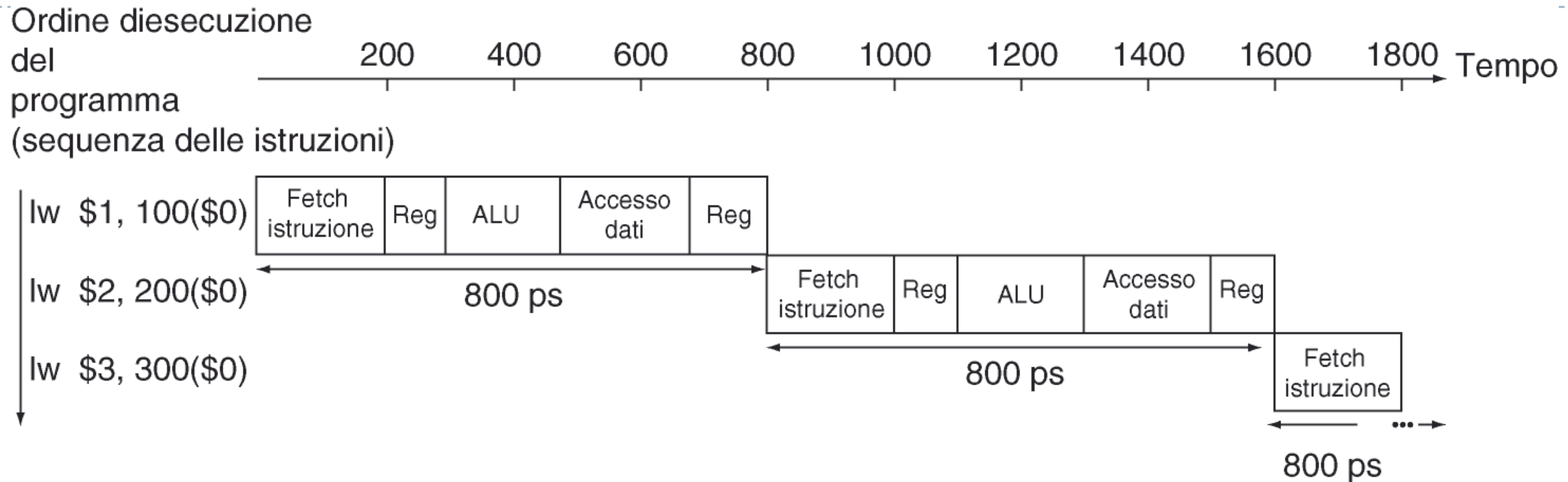
- Dato che le 5 fasi devono sovrapporsi dobbiamo individuare il periodo di clock uniforme che permette di svolgerle tutte (ovvero la durata della fase più lenta)

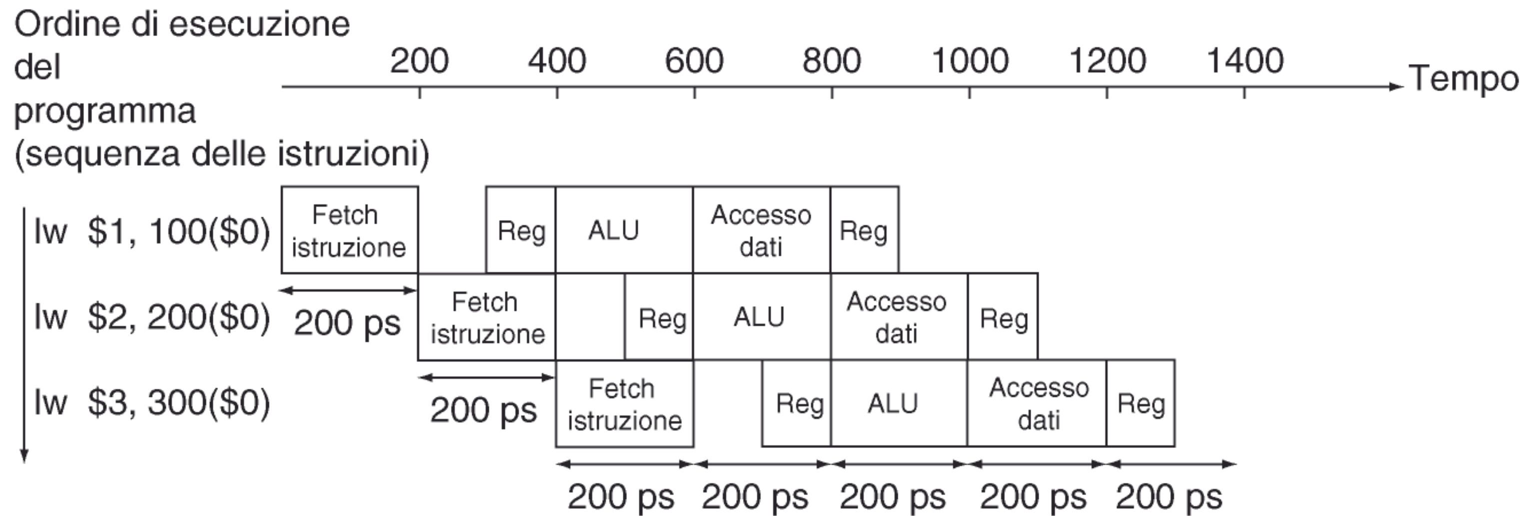
Classe dell'istruzione	Lettura dell'istruzione	Lettura dei registri	Operazione con la ALU	Accesso ai dati in memoria	Scrittura del register file	Tempo totale
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
Formato R (add, sub, AND, OR, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Salto condiziona-to (beq)	200 ps	100 ps	200 ps			500 ps

- Possiamo ridurre il periodo di clock da **800ps** (durata massima di una istruzione) a **200ps** (durata massima di una fase), ovvero quadruplicare la velocità del clock
- (in pratica si cerca sempre di progettare la CPU in modo che le fasi abbiano tempi simili)
- Dato che ad ogni colpo di clock una istruzione viene completata dalla pipeline, abbiamo **quadruplicato la velocità di esecuzione! (in condizioni ideali)**

Patterson et al., STRUTTURA E PROGETTO DEI CALCOLATORI, 3/E, Zanichelli editore S.p.A. Copyright © 2010

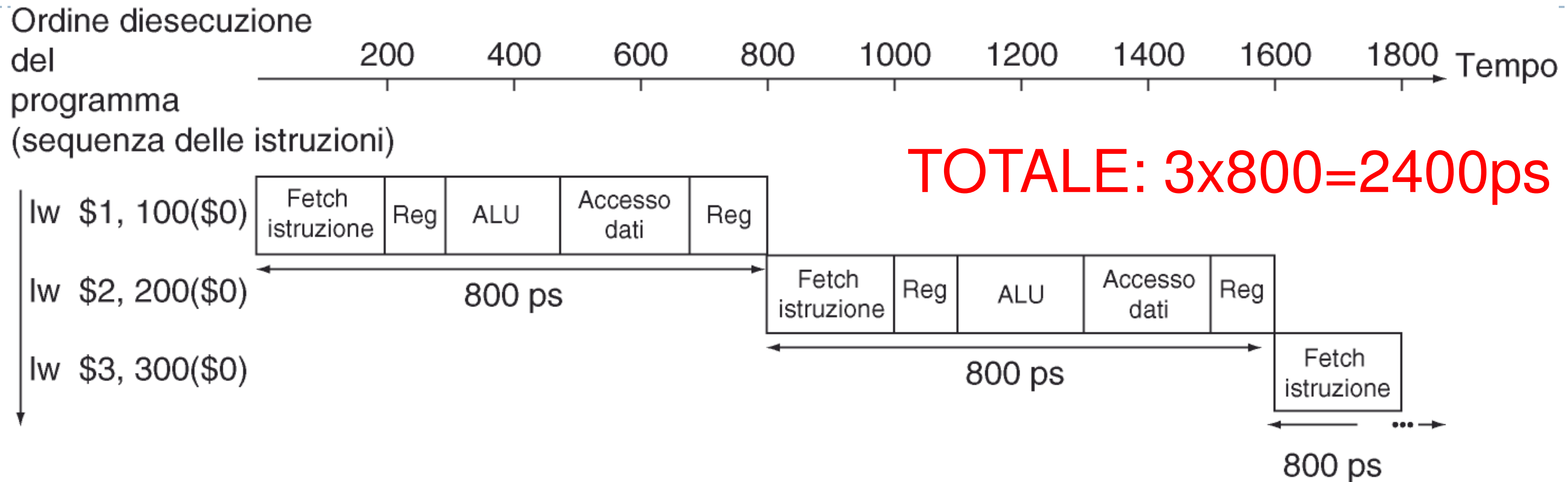
Esempio di esecuzione

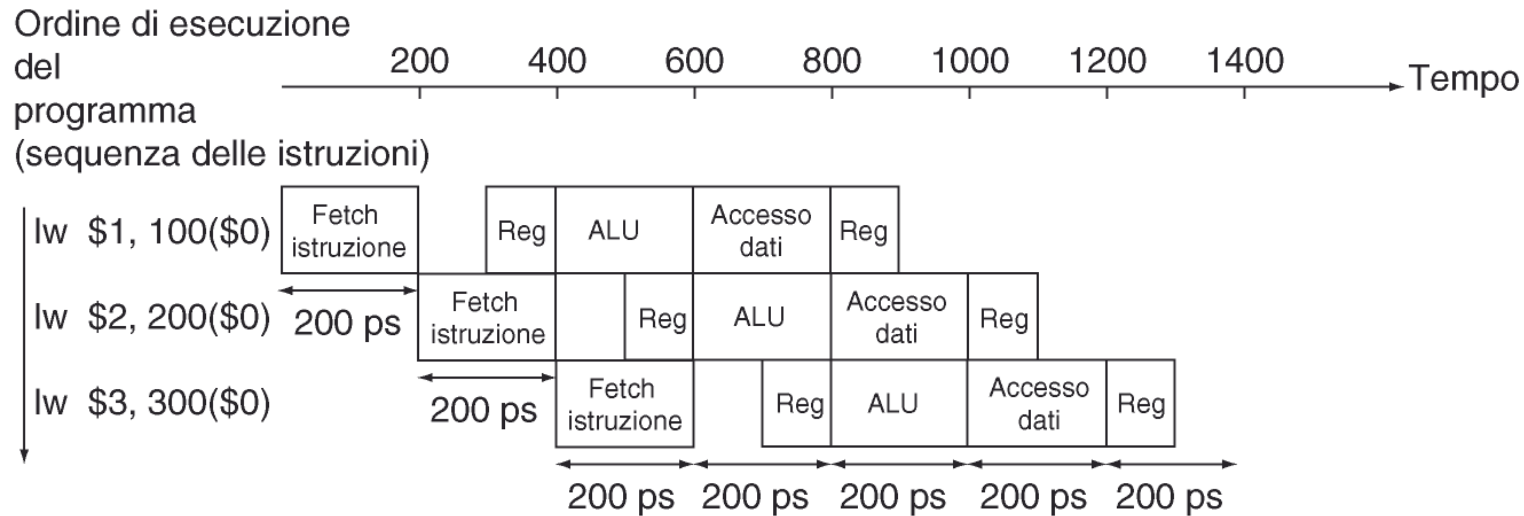




Patterson et al., STRUTTURA E PROGETTO DEI CALCOLATORI, 3/E, Zanichelli editore S.p.A. Copyright © 2010

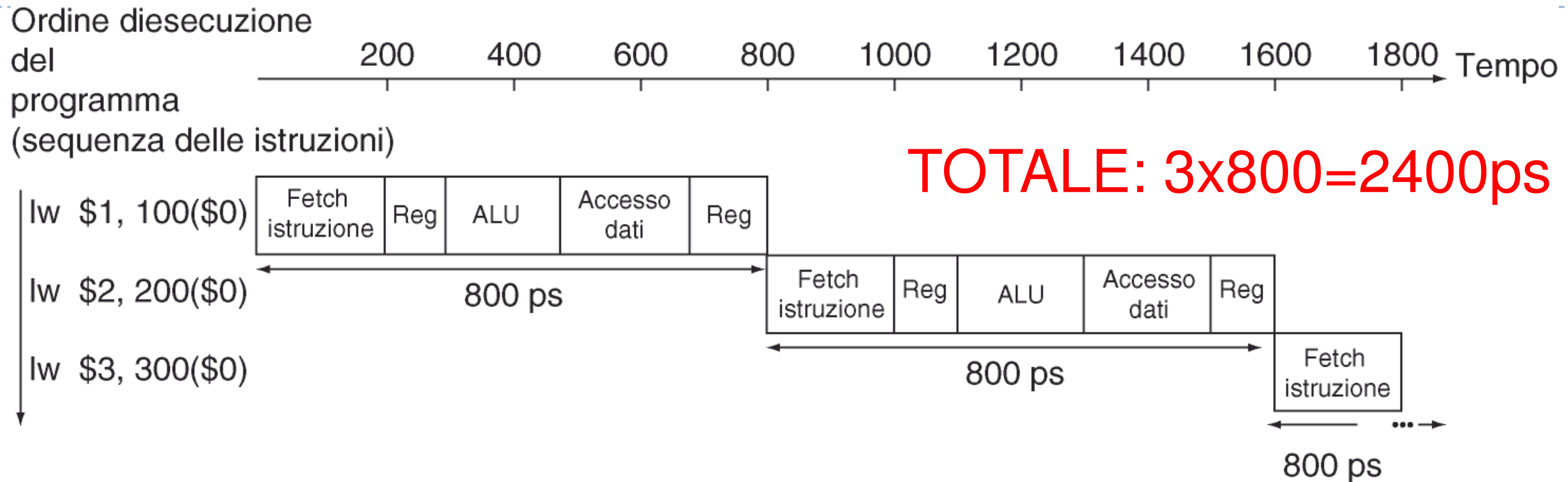
Esempio di esecuzione

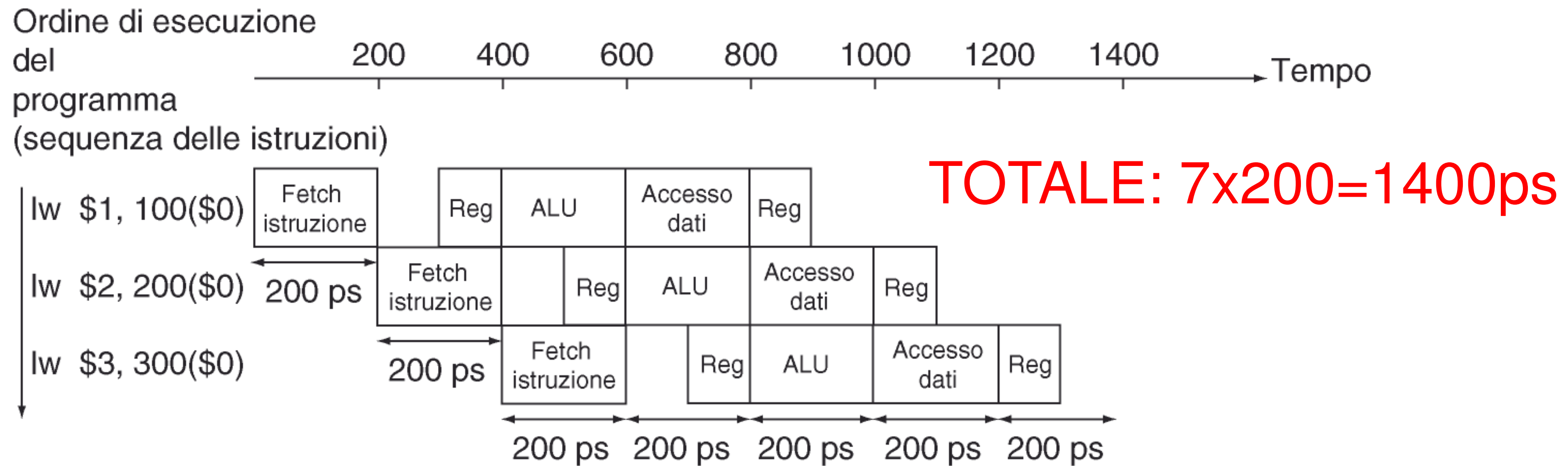




Patterson et al., STRUTTURA E PROGETTO DEI CALCOLATORI, 3/E, Zanichelli editore S.p.A. Copyright © 2010

Esempio di esecuzione





Patterson et al., STRUTTURA E PROGETTO DEI CALCOLATORI, 3/E, Zanichelli editore S.p.A. Copyright © 2010

Dimensioni fisse vs variabili

Dimensioni fisse vs variabili

- La fase di Fetch può sovrapporsi a Istruction Decode SOLO perché l'istruzione ha DIMENSIONE UNIFORME e quindi possiamo calcolare il prossimo PC senza sapere di che tipo di istruzione si tratta.
Nelle architetture con dimensione delle istruzioni variabile è necessario aspettare anche la fase di Istruction Decode e quindi la velocità tende ad essere più bassa

Dimensioni fisse vs variabili

- La fase di Fetch può sovrapporsi a Istruction Decode SOLO perché l'istruzione ha DIMENSIONE UNIFORME e quindi possiamo calcolare il prossimo PC senza sapere di che tipo di istruzione si tratta.

CISC									
IF	ID	altre	fasi	…					
		IF	ID	altre	fasi	…			
				IF	ID	altre	fasi	…	
RISC									
IF	ID	EX	MEM	WB					
	IF	ID	EX	MEM	WB				
		IF	ID	EX	MEM	WB			

Criticità/Hazard

- Come progettare le istruzioni **per realizzare una pipeline efficiente**:
  - **Lunghezza uniforme**: fetch senza necessità di decodifica
  - Istruzioni con **formati simili**: lettura dei registri già durante la decodifica
  - **Accesso alla memoria separato da ALU** (che viene usata per indirizzamento)
    - Un solo accesso alla memoria per istruzione
  - Dati e istruzioni **allineati in memoria**: un solo accesso x leggere una word o una istruzione

- **Criticità nella esecuzione (Hazard)**
  - **STRUTTURALI**: le risorse HW non sono sufficienti (es. l'ALU non è libera)
  - **SUL CONTROLLO:** un salto (jump/beq) cambia il flusso di esecuzione delle istruzioni
  - **SUI DATI**: necessità di inserire una attesa perché il dato necessario non è ancora pronto

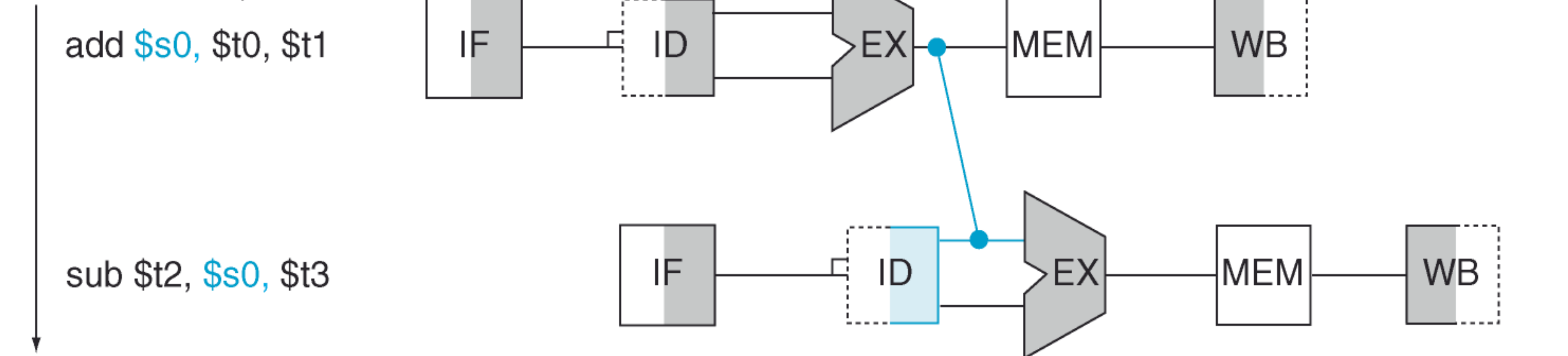
- Esempio (**data hazard sul registro $s0**):

```
add $s0, $t0, $t1     IF   ID   EX   MEM  WB
sub $t2, $s0, $t3          →    →    IF   ID   EX   MEM  WB   (2 stalli)
```

- La seconda istruzione non può eseguire la lettura degli argomenti se la prima non fa WB
- (<u>le due fasi WB e ID possono essere sovrapposte</u> se WB avviene nella prima metà e ID nella seconda metà del periodo di clock)

Propagazione/Forwarding

- In alcuni casi l'informazione necessaria è già presente nella pipeline ben prima del WB

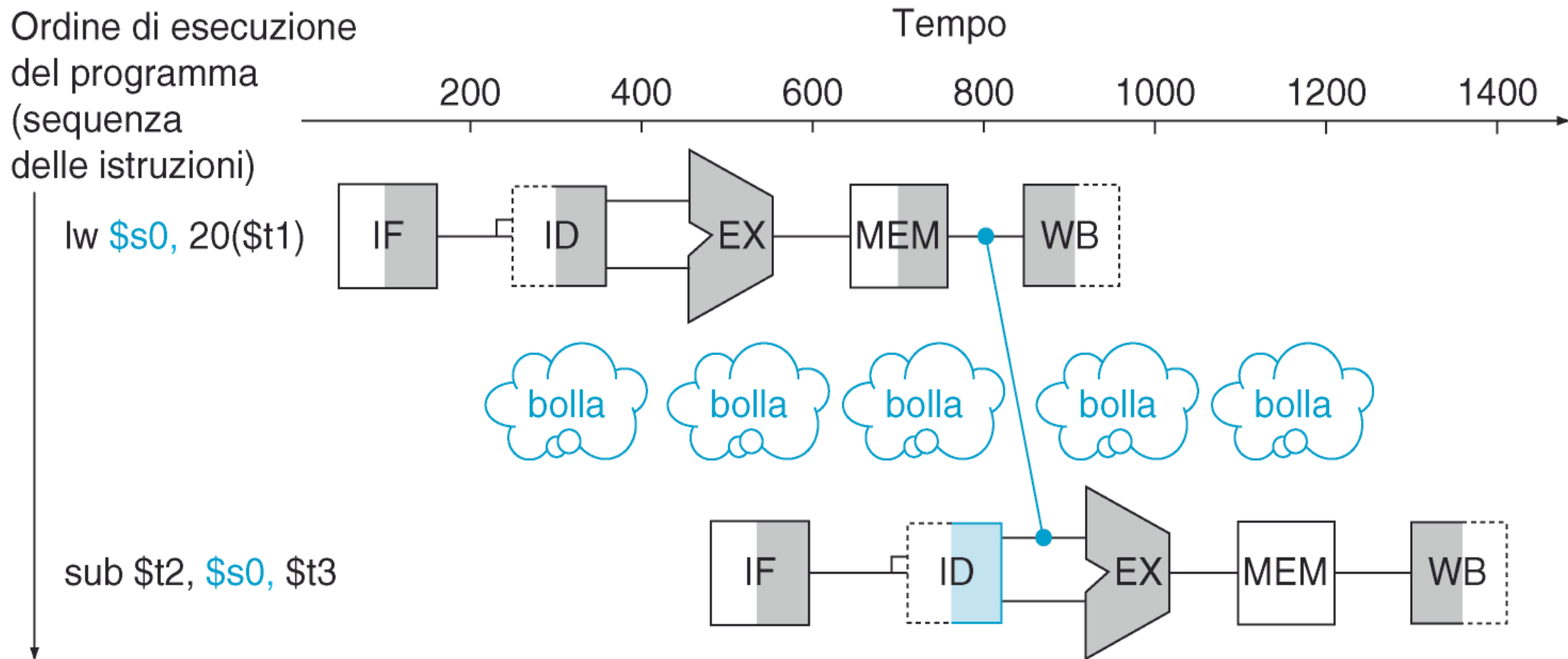


- Possiamo **inserire nel datapath** delle **«scorciatoie»** che recapitano il dato all'unità funzionale che ne ha bisogno **senza aspettare la fase di WB**
- In questo esempio NON è necessario aspettare 2 colpi di clock come nel precedente.
- Questo è possibile quando lo stadio che deve ricevere il dato è SUCCESSIVO nel tempo a quello che lo produce (nel diagramma temporale della pipeline)

Patterson et al., STRUTTURA E PROGETTO DEI CALCOLATORI, 3/E, Zanichelli editore S.p.A. Copyright © 2010

Talvolta il forwarding è impossibile

- Se la fase che ha bisogno del dato si trova PRIMA (nel tempo) di quella che lo produce sarà NECESSARIO inserire una attesa (stallo o bolla). **Esempio:** lw seguito da sub



Patterson et al., STRUTTURA E PROGETTO DEI CALCOLATORI, 3/E, Zanichelli editore S.p.A. Copyright © 2010

Esempio: A=B+E, C=B+F

Possiamo **evitare stalli riordinando le istruzioni** MA **bisogna mantenere la stessa semantica**

```
lw   $t1,   0($t0)      IF ID EX MM WB
lw   $t2,   4($t0)         IF ID EX MM WB
add  $t3, $t1, $t2            → IF ID EX MM WB                 (1 stallo)
sw   $t3, 12($t0)                    IF ID EX MM WB
lw   $t4,   8($t0)                      IF ID EX MM WB
add  $t5, $t1, $t4                         → IF ID EX MM WB (1 stallo)
sw   $t5, 16($t0)                                 IF ID EX MM WB
```

Esempio: A=B+E, C=B+F

Possiamo **evitare stalli riordinando le istruzioni** MA **bisogna mantenere la stessa semantica**

```
lw   $t1,   0($t0)      IF ID EX MM WB
lw   $t2,   4($t0)         IF ID EX MM WB
add  $t3, $t1, $t2            → IF ID EX MM WB            (1 stallo)
sw   $t3, 12($t0)                   IF ID EX MM WB
lw   $t4,   8($t0)                     IF ID EX MM WB
add  $t5, $t1, $t4                        → IF ID EX MM WB (1 stallo)
sw   $t5, 16($t0)                              IF ID EX MM WB
```

Esempio: A=B+E, C=B+F

Possiamo **evitare stalli riordinando le istruzioni** MA **bisogna mantenere la stessa semantica**

```
lw   $t1,  0($t0)    IF ID EX MM WB
lw   $t2,  4($t0)       IF ID EX MM WB
add  $t3, $t1, $t2         → IF ID EX MM WB               (1 stallo)
sw   $t3, 12($t0)                IF ID EX MM WB
lw   $t4,  8($t0)                   IF ID EX MM WB
add  $t5, $t1, $t4                     → IF ID EX MM WB (1 stallo)
sw   $t5, 16($t0)                            IF ID EX MM WB
```

```
lw   $t1,  0($t0)   IF ID EX MM WB
lw   $t2,  4($t0)      IF ID EX MM WB
lw   $t4,  8($t0)         IF ID EX MM WB
add  $t3, $t1, $t2           IF ID EX WB
sw   $t3, 12($t0)               IF ID EX MM WB
add  $t5, $t1, $t4                 IF ID EX MM WB
sw   $t5, 16($t0)                     IF ID EX MM WB (2 di meno)
```

Esempio: A=B+E, C=B+F

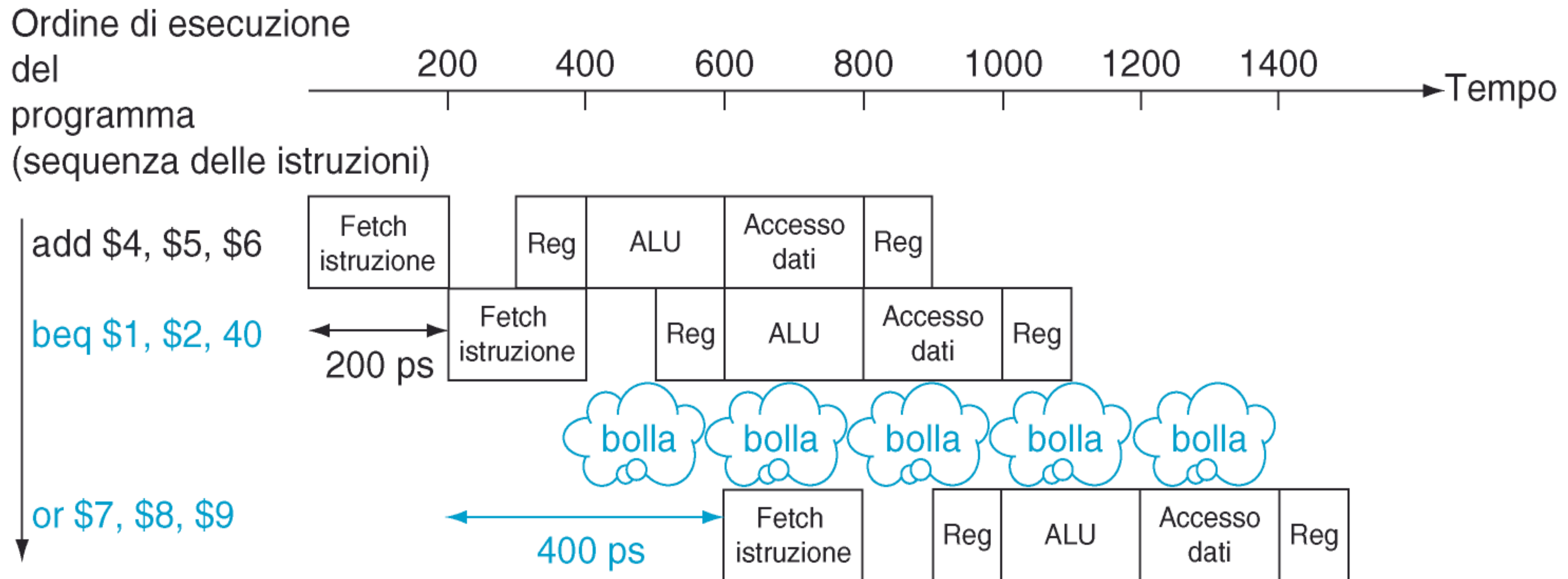
Possiamo **evitare stalli riordinando le istruzioni** MA **bisogna mantenere la stessa semantica**

```
lw   $t1,  0($t0)     IF ID EX MM WB
lw   $t2,  4($t0)        IF ID EX MM WB
add  $t3, $t1, $t2          → IF ID EX MM WB              (1 stallo)
sw   $t3, 12($t0)                  IF ID EX MM WB
lw   $t4,  8($t0)                     IF ID EX MM WB
add  $t5, $t1, $t4                      → IF ID EX MM WB (1 stallo)
sw   $t5, 16($t0)                              IF ID EX MM WB
```

```
lw   $t1,  0($t0)    IF ID EX MM WB
lw   $t2,  4($t0)       IF ID EX MM WB
lw   $t4,  8($t0)          IF ID EX MM WB
add  $t3, $t1, $t2            IF ID EX WB
sw   $t3, 12($t0)                IF ID EX MM WB
add  $t5, $t1, $t4                  IF ID EX MM WB
sw   $t5, 16($t0)                      IF ID EX MM WB (2 di meno)
```

Hazard sul controllo (beq)

- Quando il salto condizionato di una beq viene eseguito l'istruzione seguente (che è già in pipeline) dev'essere scartata per eseguire quella giusta (la destinazione del salto)



- La prossima istruzione può essere caricata solo DOPO che il salto è stato DECISO, quindi **sono necessari 2 stalli se beq viene decisa in EXE o 1 se viene decisa in ID**
- (lo schema qui sopra tiene assume che BEQ sia decisa nella fase ID usando HW aggiuntivo)

Patterson et al., STRUTTURA E PROGETTO DEI CALCOLATORI, 3/E, Zanichelli editore S.p.A. Copyright © 2010

Come mitigare i control hazard

- **Anticipare la decisione:**
- se beq viene calcolata dalla ALU nella fase EXE ci saranno 2 istruzioni da «buttare» perché abbiamo aspettato la fine della 3° fase
- se invece beq viene **decisa nella fase ID** è sufficiente «buttare» 1 istruzione

- **Ritardare il salto:**
- se (inoltre) l'istruzione che segue la beq viene SEMPRE eseguita anche se il salto viene fatto, si elimina di fatto lo stallo

- **Branch prediction:** (previsione del salto)
- la CPU osserva i salti eseguiti e cerca di pre-caricare l'istruzione (seguente o di destinazione) che viene eseguita più spesso

- La implementazione della beq che stiamo studiando assume che il salto NON venga fatto e carica sempre l'istruzione seguente

Soluzione esercizio per casa

- Si vuole aggiungere alla CPU l'istruzione vectorized jump (**vj**), di tipo **I** e sintassi assembly
- `vj $indice, vettore`
- che salta all'indirizzo contenuto nell'elemento $indice-esimo del *vettore di word*.
- **Esempio:** se in memoria si è definito staticamente il
- ***vettore:*** **.word 15, 24, 313, 42**
- e nel registro **$t0** c'è il valore **3** allora `vj $t0, vettore` salterà all'indirizzo **42** (che è l'elemento con indice 3 del vettore)
- a) si disegnino le modifiche necessarie a realizzare la funzione, aggiungendo tutti gli eventuali MUX, segnali di controllo, bus, ALU e sommatori (ecc) che ritenete necessari.
- b) indicate i valori di tutti i segnali di controllo, in modo da eseguire l'istruzione **vj.**
- c) tenendo conto che il tempo di accesso ai registri (sia in lettura che scrittura) è di **25ps**, l'accesso alla memoria impiega **75ps**, la ALU e i sommatori impiegano **100ps** e ignorando gli altri ritardi, calcolate il tempo di esecuzione minimo della istruzione **vj** e indicate se è necessario aumentare il periodo di clock della CPU per poter svolgere questa nuova istruzione.

Modifiche per vj $rs, vettore

- L'istruzione equivale alle istruzioni
  - **sll $at, $rs, 2**
  - **lw $at, vettore($at)**
  - **jr $at**
- Quindi dobbiamo:
  - eseguire le operazioni di una lw ma **con registro base moltiplicato per 4**
  - usare il risultato per fare il salto
- Serve una **unità di shift a sinistra di 2** da inserire sul primo argomento dell'ALU (con un MUX), Inoltre l'output della memoria deve essere inserito nel PC (con un secondo MUX)

- I tempi sono quelli di una **lw**
- I segnali da inviare dalla CU sono simili alla lw:

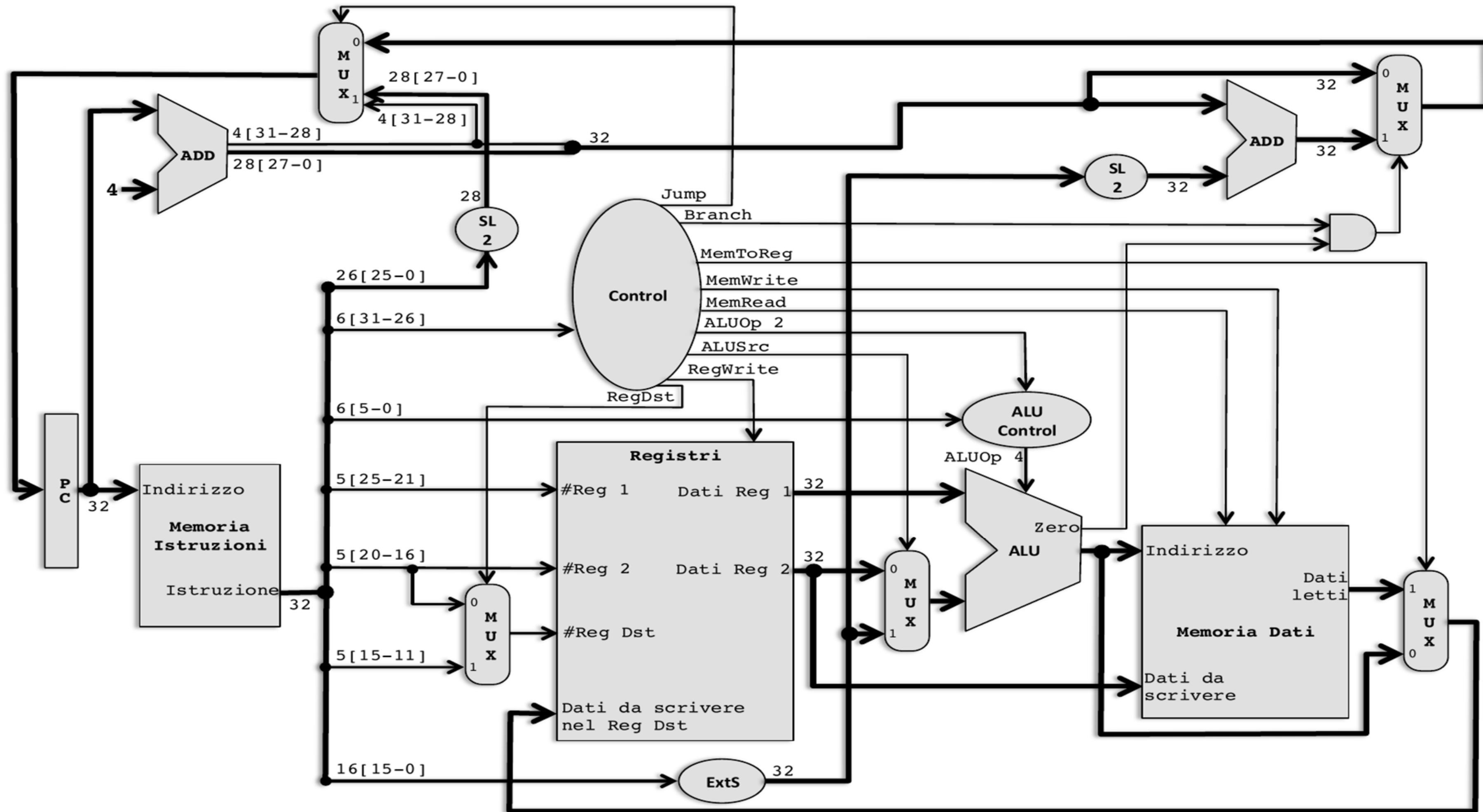
Modifiche per vj $rs, vettore

- L'istruzione equivale alle istruzioni
  - **sll $at, $rs, 2**
  - **lw $at, vettore($at)**
  - **jr $at**
- Quindi dobbiamo:
  - eseguire le operazioni di una lw ma **con registro base moltiplicato per 4**
  - usare il risultato per fare il salto
- Serve una **unità di shift a sinistra di 2** da inserire sul primo argomento dell'ALU (con un MUX), Inoltre l'output della memoria deve essere inserito nel PC (con un secondo MUX)

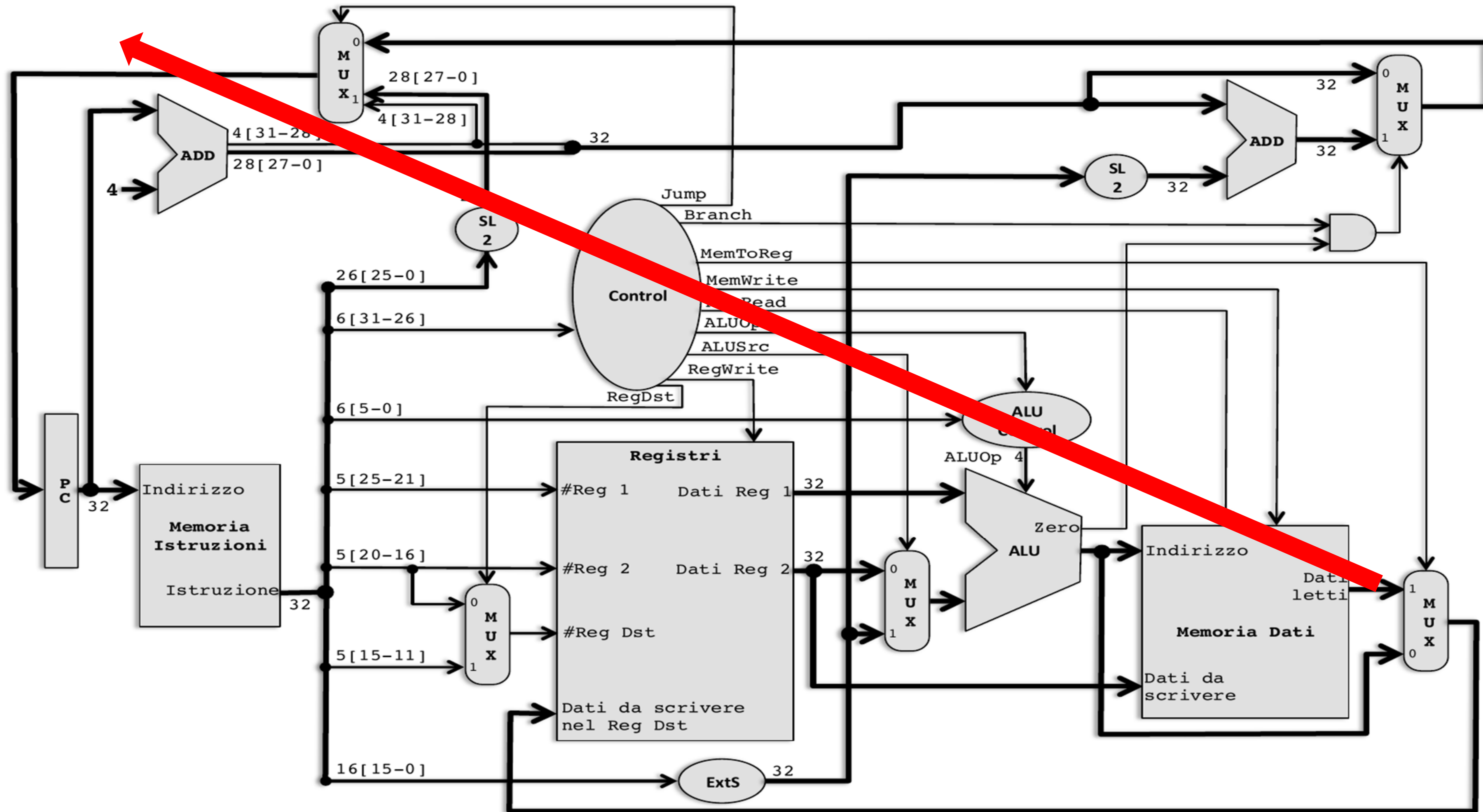
vj	Reg Dest	ALU Src	Mem toReg	Reg Write	Mem Read	Mem Write	Branch	Jump	ALU Op1	ALU Op0
1	X	1	X	0	1	0	X	X	0	0

- I tempi sono quelli di una **lw**
- I segnali da inviare dalla CU sono simili alla lw:

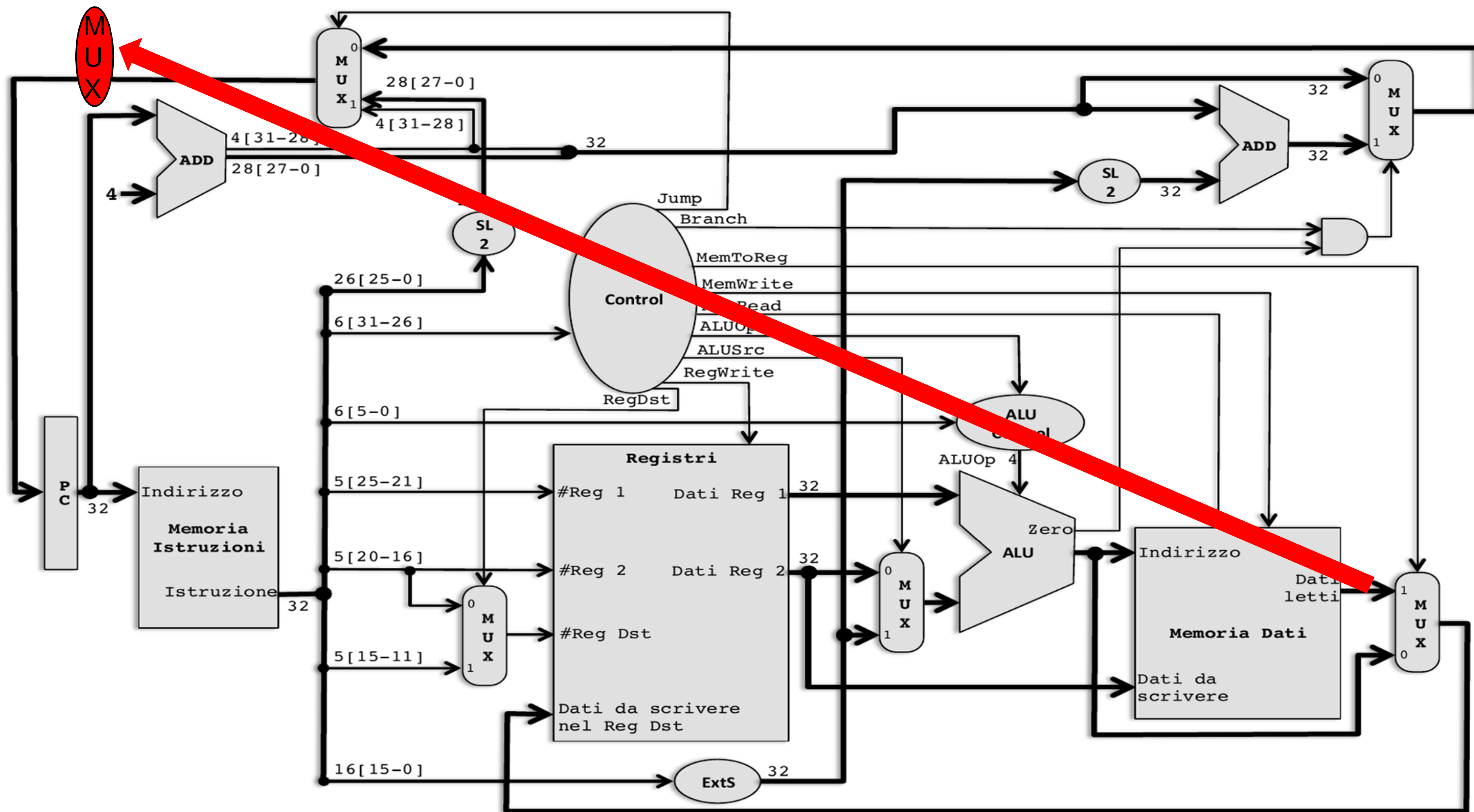
Modifiche per vj $rs, vettore



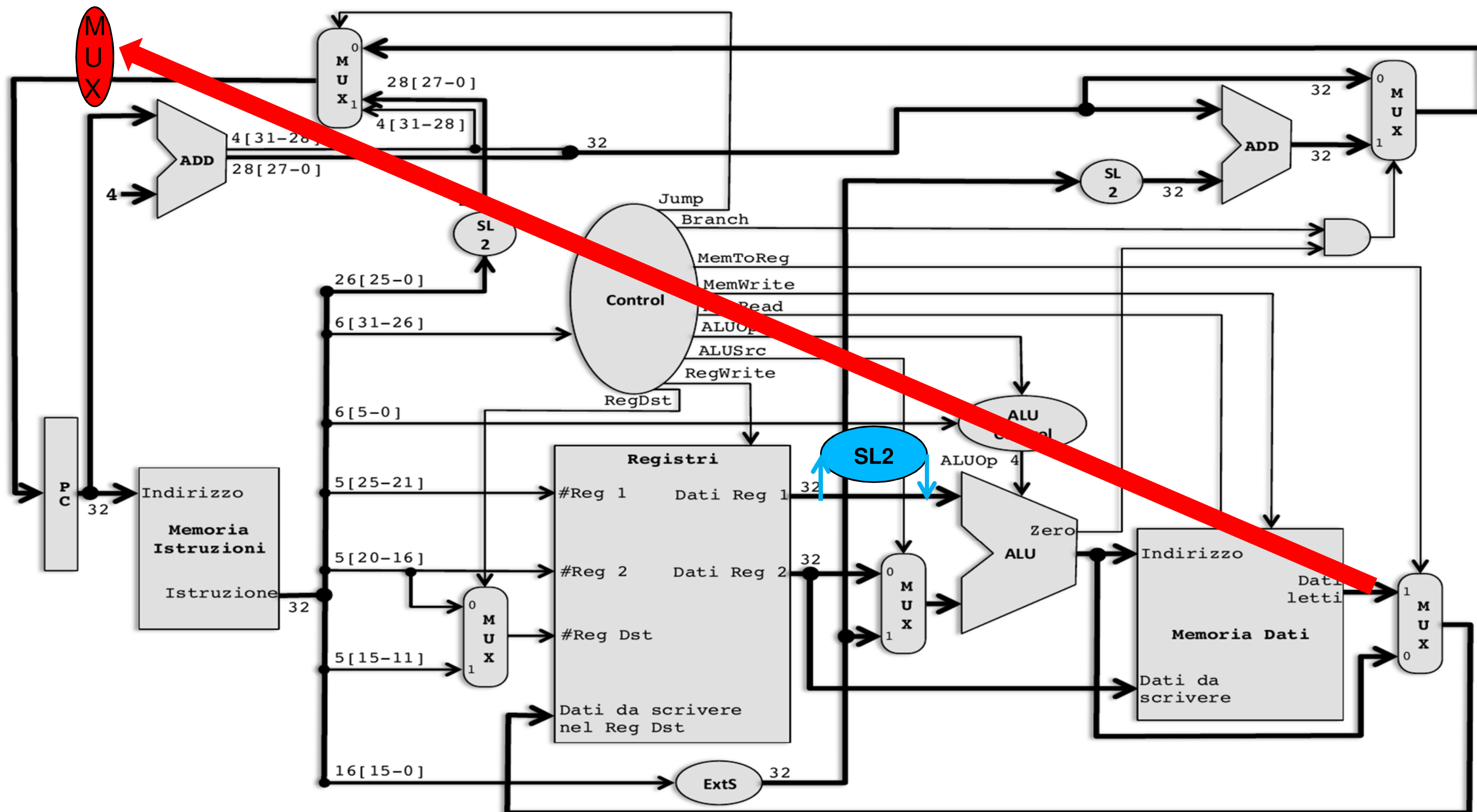
Modifiche per vj $rs, vettore



Modifiche per vj $rs, vettore

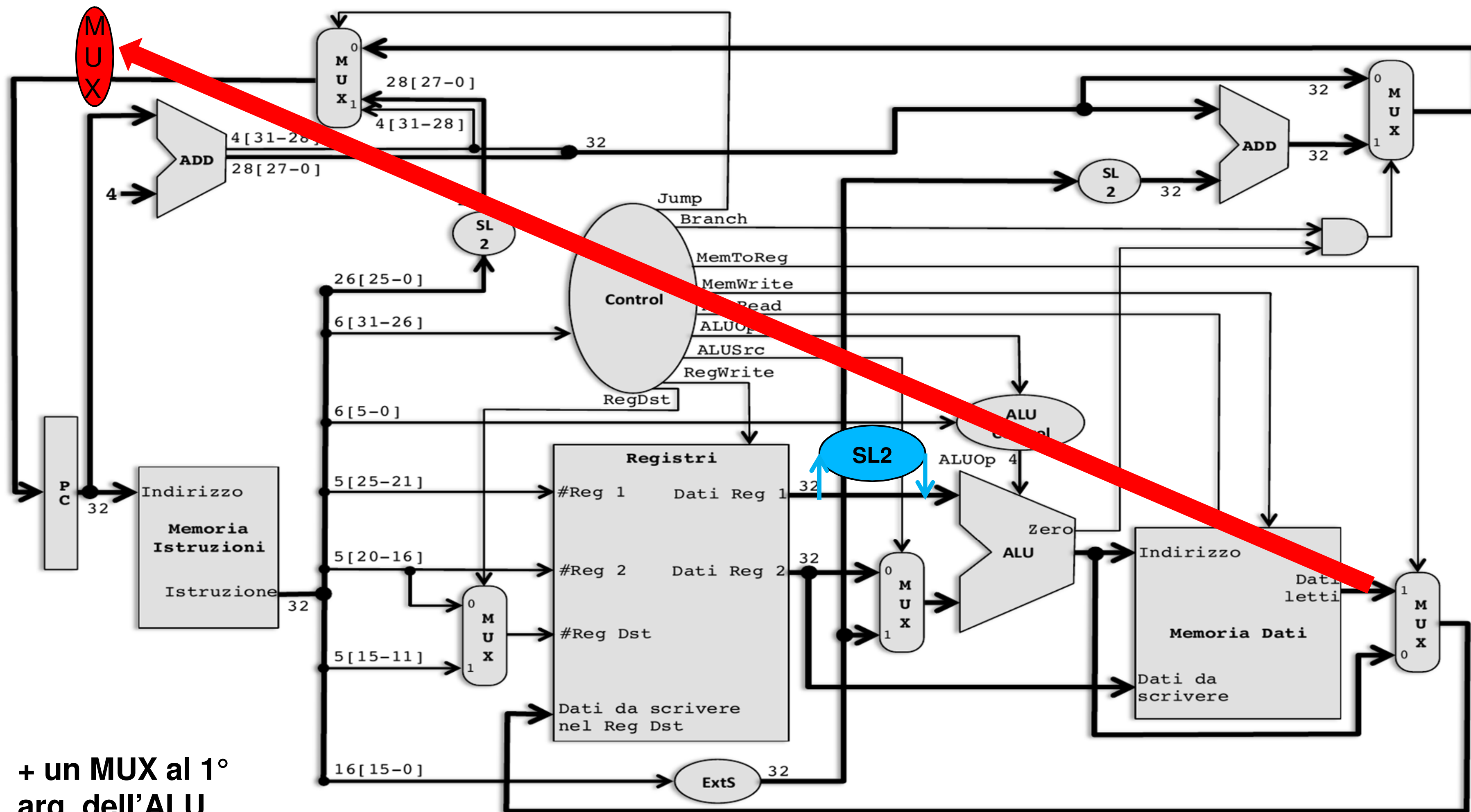


Modifiche per vj $rs, vettore



Modifiche per vj $rs, vettore

Implementazione ad un ciclo di clock di MIPS (solamente le istruzioni: add, sub, and, or, xor, slt, lw, sw, beq, j)



+ un MUX al 1° arg. dell'ALU

Esercizio

- Calcolate il numero di cicli necessari a eseguire le istruzioni seguenti:
  - individuate i **data e control hazard**
  - per **determinare se con il forwarding possono essere risolti** tracciate il diagramma temporale della pipeline
  - determinate **quali non possono essere risolti e necessitano di stalli** (e quanti stalli)
  - tenente conto del tempo necessario a **caricare la pipeline**
- Assumete:
  - che la beq salti alla fine della fase EXE,
  - che il salto beq non sia ritardato,
  - che j non introduca stalli

```
# Sommo un vettore di word
sommavettore:
        li   $t0,   0     # somma
        li   $t1,   40    # fine
        li   $t2,   0     # offset
ciclo:  beq  $t2, $t1, fine
        lw   $t3,   vettore($t2)
        add $t0, $t0, $t3
        addi $t2, $t2, 4
        j ciclo
fine:   li   $v0,   10
        syscall
```