



# Hazard sul controllo

Prof. Alberto Borghese  
Dipartimento di Informatica  
[alberto.borghese@unimi.it](mailto:alberto.borghese@unimi.it)

Università degli Studi di Milano

Riferimento al Patterson: 4.8



# Sommario

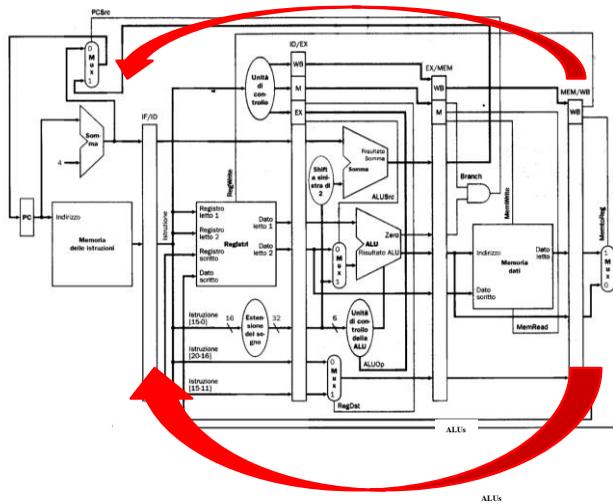
**Hazard sul controllo**

Anticipazione dei salti

Predizione dei salti



## Pipeline – criticità o hazard



La fase di WB genera un cammino da dx (WB) a sx (DEC) sul data path, serve una coppia di istruzioni.  
 beq genera un cammino da dx (MEM) a sx (IF) sul control path, a prescindere da altre istruzioni.



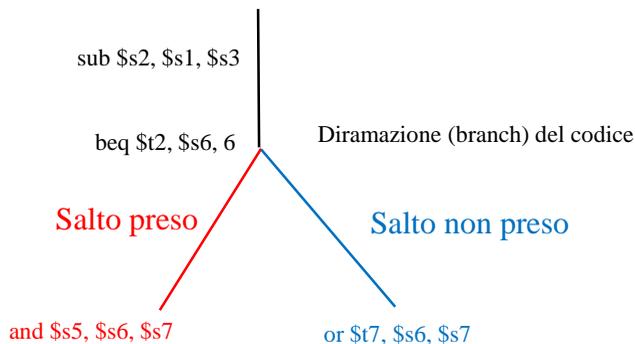
## Come affrontare gli hazard su controllo

- Si può risolvere l'hazard...
  - ...*aspettare*
    - si mette lo stage opportuno dell'istruzione dipendente dalla precedente in pausa
    - il controllo della pipeline deve individuare il problema prima che avvenga! **Stallo**.
  - ...*prevenire*
    - Il compilatore, come ottimizzazione, può **riordinare le istruzioni** in modo che il risultato sia lo stesso ma non ci siano hazard
  - ...*modificare la CPU*
  - ...*scartare istruzioni*
    - si butta via l'attuale lavoro della pipeline e si ricomincia ("flushing" della pipeline)
    - è sufficiente individuare il problema DOPO che è avvenuto
    - Es: l'istruzione successiva (a sx) ha usato in lettura un registro che è stato appena modificato dall'istruzione precedente (dx)? **flush**. Oppure: l'istruzione precedente (a dx) ha effettuato un salto e quindi successive (a sx) sta lavorando con un PC e IR sbagliato? **flush**.
  - ...*prevedere*
    - attraverso alcuni meccanismi di **predizione** preposti, l'architettura stessa tenta di predire su base statistica i risultati rilevanti dell'istruzione in corso (es il PC – "branch prediction", o il valori prodotti – "value prediction"). Se la predizione si rivela giusta: tutto ok. Se si rivela sbagliata: **roll-back**



## Hazard sul controllo

0x400 sub \$s2, \$s1, \$s3
0x404 beq \$t2, \$s6, 7
0x408 or \$t7, \$s6, \$s7
0x40C add \$t4, \$s8, \$s8
0x410 and \$s5, \$s6, \$s7
0x414 add \$t0, \$t1, \$t2
0x418 sw \$s3, 24(\$t1)
0x41C addi \$t7, \$s6, 10
0x420 add \$t8, \$s2, \$s2
0x424 and \$s5, \$s6, \$s7
0x418 add \$t0, \$t1, \$t2



## Esempio di Hazard sul controllo

0x400 sub \$s2, \$s1, \$s3	IF	ID	EX \$1-\$3	MEM	WB s->\$2			
0x404 beq \$t2, \$s6, 7		IF	ID	EX Zero if (\$s2 == \$s5)	MEM	WB		
0x408 or \$t7, \$s6, \$s7			IF	ID	EX	MEM	WB	
0x40C add \$t4, \$s8, \$s8				IF	ID	EX	MEM	WB
0x410 and \$s5, \$s6, \$s7					IF	ID	EX	MEM
0x414 add \$t0, \$t1, \$t2						IF	ID	EX

Quando sono in fase di fetch dovrei avere a disposizione l'indirizzo corretto. In caso di salto questo potrebbe esser disponibile nella PIPELINE solo all'inizio della fase di WB della beq.

**In caso di salto:** Ho 3 istruzioni sbagliate in pipeline: or, la add e la and.

NB Il PC è master/slave per cui occorre che l'indirizzo sia pronto prima dell'inizio della fase di fetch.



## Soluzione 1 – «aspettare» - stallo



0x400 sub \$s2, \$s1, \$s3

0x404 beq \$t2, \$s6, 7

0x408 or \$t7, \$s6, \$s7

0x40C add \$t4, \$s8, \$s8

0x410 and \$s5, \$s6, \$s7

0x414 add \$t0, \$t1, \$t2

0x418 sw \$s3, 24(\$t1)

0x41C addi \$t7, \$s6, 10

0x420 add \$t8, \$s2, \$s2

0x424 and \$s5, \$s6, \$s7

0x428 add \$t0, \$t1, \$t2

....

....

0x400 sub \$s2, \$s1, \$s3

0x404 beq \$t2, \$s6, 7

nop

nop

nop

0x408 or \$t7, \$s6, \$s7

0x40C add \$t4, \$s8, \$s8

0x410 and \$s5, \$s6, \$s7

0x414 add \$t0, \$t1, \$t2

0x418 sw \$s3, 24(\$t1)

0x41C addi \$t7, \$s6, 10

0x420 add \$t8, \$s2, \$s2

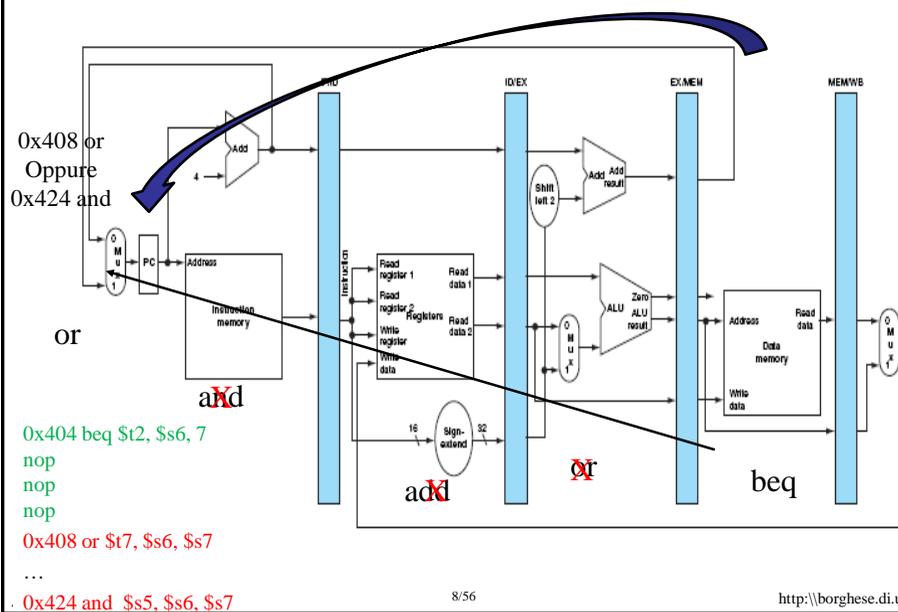
0x424 and \$s5, \$s6, \$s7

0x428 add \$t0, \$t1, \$t2

Con lo stallo spreco di 3 cicli di clock (in modo che la fase IF dell'istruzione successiva vada a coincidere con la fase di WB dell'istruzione beq. **Situazione troppo frequente perché la soluzione sia accettabile.**



## Soluzione con uno stallo di 3 cicli di clock





## Soluzione 2 – «prevenire»



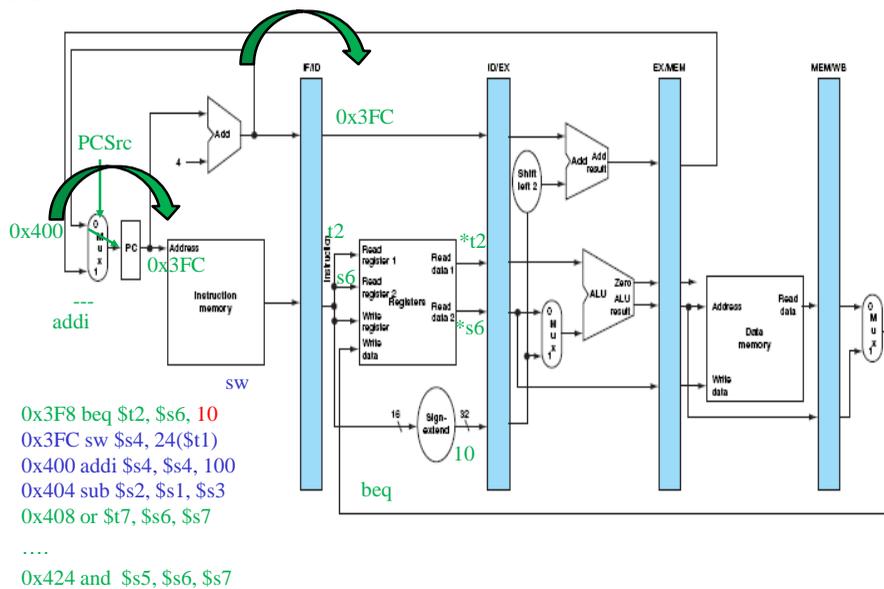
0x3F8 sw \$s4, 24(\$t1)	0x3F8 beq \$t2, \$s6, <b>10</b>
0x3FC addi \$s4, \$s4, 100	0x3FC sw \$s4, 24(\$t1)
0x400 sub \$s2, \$s1, \$s3	0x400 addi \$s4, \$s4, 100
0x404 beq \$t2, \$s6, 7	0x404 sub \$s2, \$s1, \$s3
0x408 or \$t7, \$s6, \$s7	0x408 or \$t7, \$s6, \$s7
0x40C add \$t4, \$s8, \$s8	0x40C add \$t4, \$s8, \$s8
0x410 and \$s5, \$s6, \$s7	0x410 and \$s5, \$s6, \$s7
0x414 add \$t0, \$t1, \$t2	0x414 add \$t0, \$t1, \$t2
0x418 sw \$s3, 24(\$t1)	0x418 sw \$s3, 24(\$t1)
0x41C addi \$t7, \$s6, 10	0x41C addi \$t7, \$s6, 10
0x420 add \$t8, \$s2, \$s2	0x420 add \$t8, \$s2, \$s2
0x424 and \$s5, \$s6, \$s7	0x424 and \$s5, \$s6, \$s7
0x428 add \$t0, \$t1, \$t2	0x428 add \$t0, \$t1, \$t2
....	....
....	....

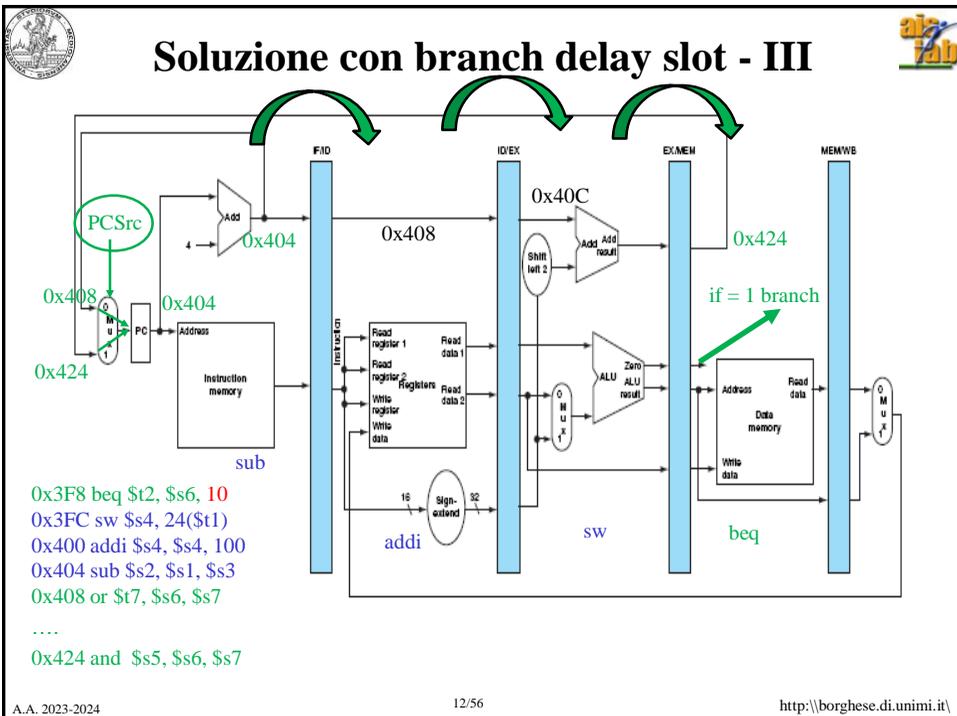
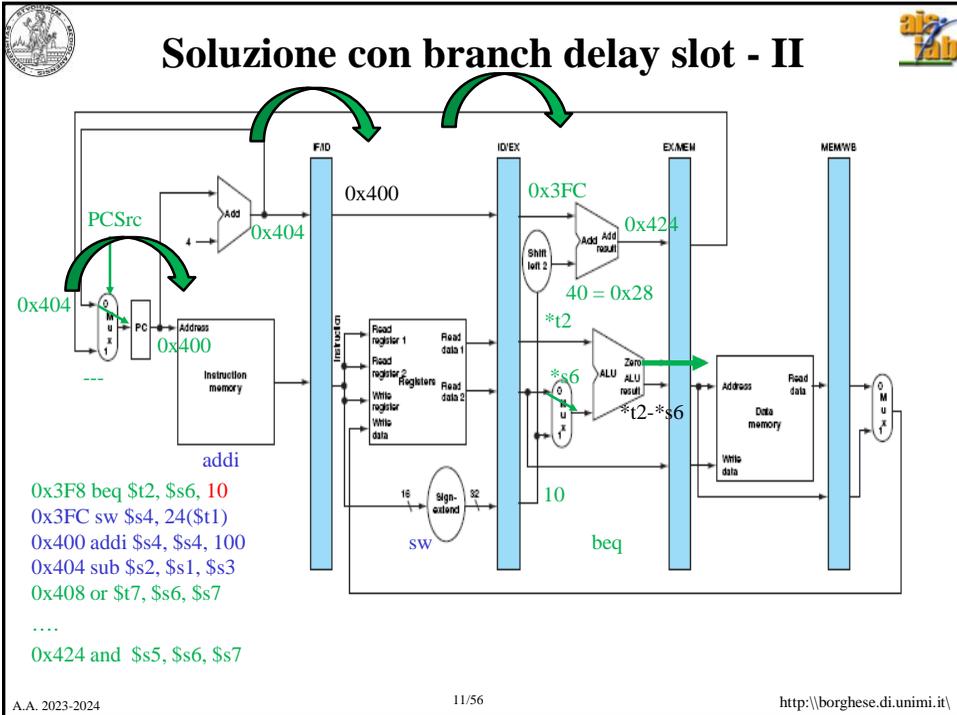
Branch delay slots

Le istruzioni immediatamente seguenti il salto vengono **sempre** eseguite (sia che il salto venga preso sia che non venga preso). **Non ci sono dipendenze con la beq.** Occorre aggiornare l'indirizzo di salto.



## Soluzione con branch delay slot - I







## Salto incondizionato – “prevenire” - I



Utilizzato all'interno dei cicli for / while. Non pone problemi. Si risolve con la riorganizzazione del codice.

396:	addi \$t6,\$t6,20	396:	j 80000
400:	add \$s0, \$s1, \$s2	400:	addi \$t6,\$t6,20
404:	j 80000	404:	add \$s0, \$s1, \$s2
Label 408:	and \$s1, \$s2, \$s3		
80000:	or \$t0, \$t1, \$t2	80000:	or \$t0, \$t1, \$t2
80004:	sub \$t3, \$t4, \$t5	80004:	sub \$t3, \$t4, \$t5

j “lavora” nella fase di decodifica. Vengono inserite due istruzioni: una in fase di decodifica e una in fase di fetch prima che il salto venga preso nella fase di EXE della jump: delayed jump. **Riempio tutti gli slot di esecuzione.**

L'esecuzione avviene fuori ordine, ma l'utente non vede differenze.

Come viene modificata la CPU (parte di datapath e parte di controllo)?



## Salto incondizionato – “prevenire” - II



Prendo le due istruzioni dalla destinazione del salto.

400:	add \$s0, \$s1, \$s2	400:	add \$s0, \$s1, \$s2
404:	j 80000	404:	j 80008
408:	and \$s1, \$s2, \$s3	408:	or \$t0, \$t1, \$t2
		412:	addi \$t6, \$t7, 100
80000:	or \$t0, \$t1, \$t2		
80004:	addi \$t6,\$t7,100		
80008:	sub \$t3, \$t4, \$t5	80008:	sub \$t3, \$t4,\$t5

Riempio tutti gli slot di esecuzione.

E' il compilatore a riorganizzare il codice. Non sono richieste modifiche alla CPU.

Si può fare di meglio con una fase di fetch evoluta. *L'istruzione di jump è necessaria?*



## Salto incondizionato



Utilizzato all'interno dei cicli for / while. Non pone problemi. Si risolve con la riorganizzazione del codice.

400:	add \$s0, \$s1, \$s2	400:	add \$s0, \$s1, \$s2
404:	j 80000	80000:	or \$t0, \$t1, \$t2
408:	and \$s1, \$s2, \$s3	80004:	sub \$t3, \$t4, \$t5
80000:	or \$t0, \$t1, \$t2		
80004:	sub \$t3, \$t4, \$t5		

L'esecuzione avviene fuori ordine, ma l'utente non vede differenze.



## Modifiche della CPU per i salti condizionati



Non sempre i delay slot si riescono a riempire

3 slot sono tanti

Il miglioramento dell'architettura è richiesto.



# Sommario



Hazard sul controllo

Anticipazione dei salti

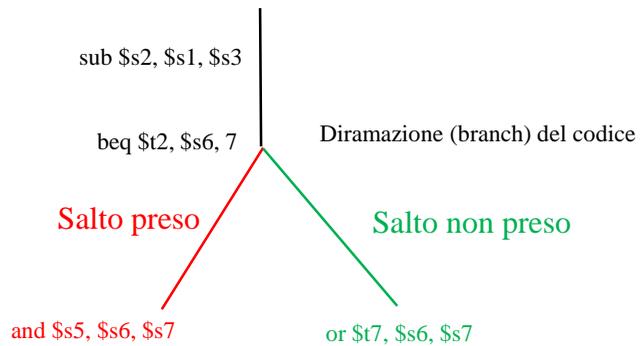
Predizione dei salti



## «predire»



0x400 sub \$s2, \$s1, \$s3
0x404 beq \$t2, \$s6, 7
0x408 or \$t7, \$s6, \$s7
0x40C add \$t4, \$s8, \$s8
0x410 and \$s5, \$s6, \$s7
0x414 add \$t0, \$t1, \$t2
0x418 sw \$s3, 24(\$t1)
0x41C addi \$t7, \$s6, 10
0x420 add \$t8, \$s2, \$s2
0x424 and \$s5, \$s6, \$s7
0x418 add \$t0, \$t1, \$t2



Supponiamo sia corretto continuare in sequenza (salto non preso). Scommettiamo.

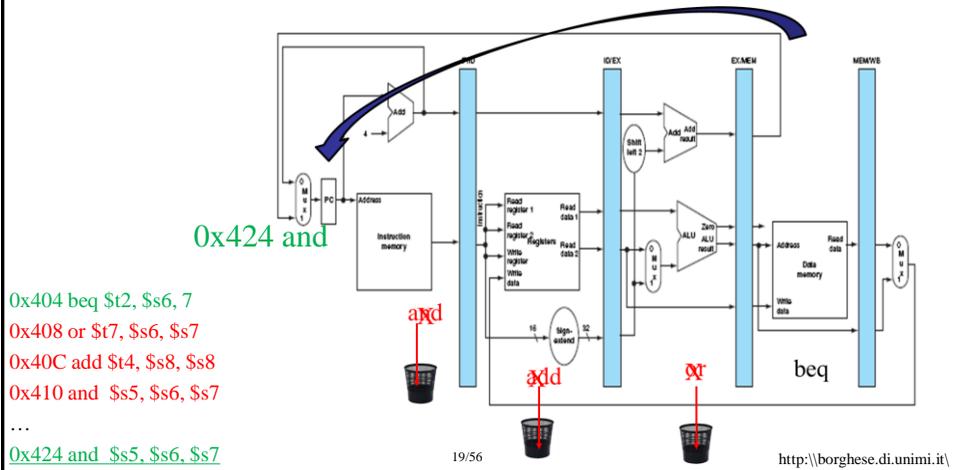
E se la supposizione risultasse **non corretta**?



# Flush della pipeline



- Sostituisco delle nop alle 3 istruzioni.
- 3 istruzioni sono tante. Cerchiamo di minimizzare l'impatto di una predizione errata.



# Modifica della CPU



Obiettivi:

- 1) **Identificare** l'hazard durante la fase ID di esecuzione della branch (possibile se condizioni semplici).
- 2) **Saltare all'istruzione** di destinazione del salto -> istruzione letta quando la beq è in fase di EXE.
- 3) **Scartare una sola istruzione** nel caso di predizione errata.

400:	sub \$s2, \$s1, \$s3	IF	ID	EX \$s1-\$s3	MEM	WB s->\$2			
404:	beq \$t2, \$s6, 7		IF	ID	EX Zero if (\$s2 == \$s5)	MEM	WB		
<del>408:</del>	<del>or \$t7, \$s6, \$s7</del>			<del>IF</del>	<del>ID</del>	<del>EX</del>	<del>MEM</del>	<del>WB</del>	
424:	and \$s5, \$s6, \$s7				IF	ID	EX	MEM	WB

A.A. 2023-

or -> e poi proseguo con la add regolarmente.



# 1) Identificare l'Hazard nella fase ID



La fase di ID è la prima fase in cui la CPU (la UC) sa che l'istruzione è una beq.

Supponiamo a-priori che il salto **non debba essere preso** -> continuo in sequenza.  
Scopro a-posteriori che il salto **deve essere preso**, ho creato un hazard che devo risolvere.

Come identifico che il salto deve essere preso?

Nella fase di DECODE: se “(il contenuto del registro source = contenuto del registro destinazione) e l'istruzione è una branch il salto doveva essere preso”.

```
If (*rs == *rt) & (branch) then
    hazard_controllo
```

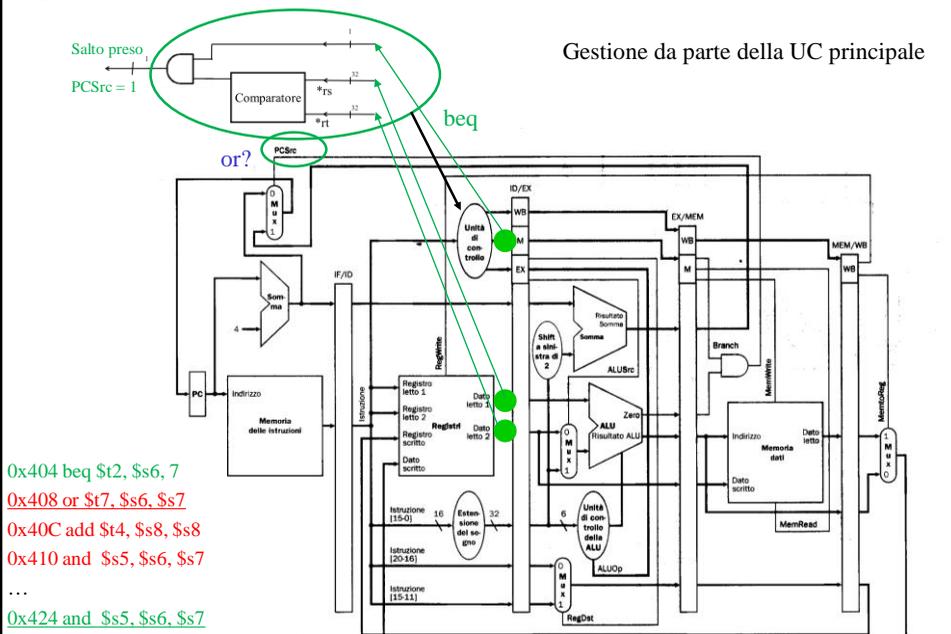
Qui tutti i dati che determinano l'hazard sono associate alla beq. L'hazard riguarda solo la branch a differenza degli hazard sui dati che coinvolgono coppie di istruzioni.

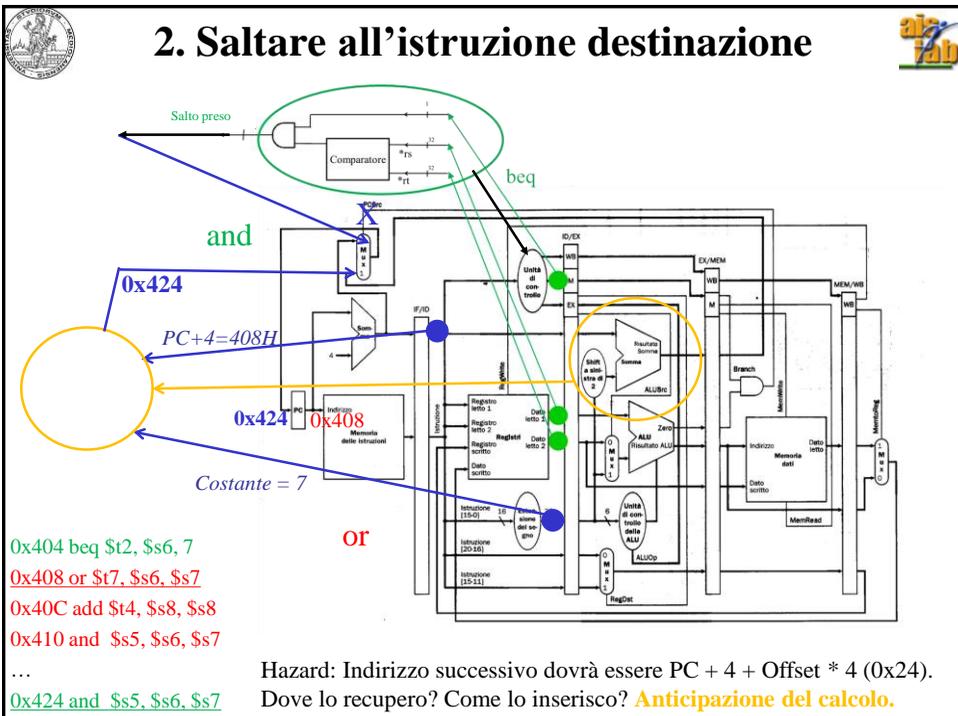
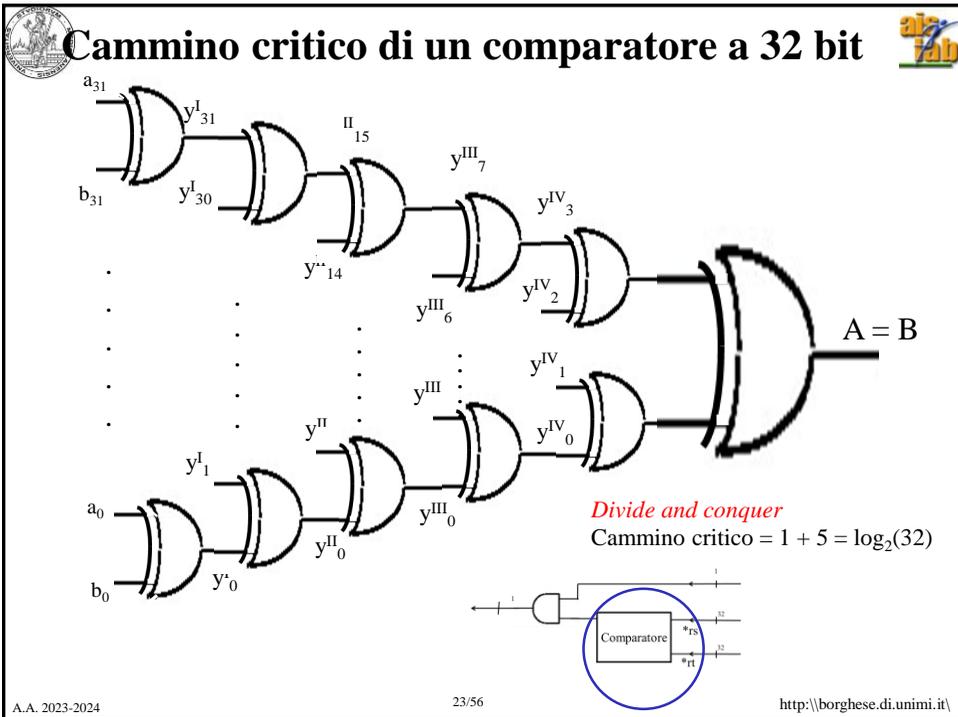


# UC che identifica hazard sul controllo in ID



Gestione da parte della UC principale



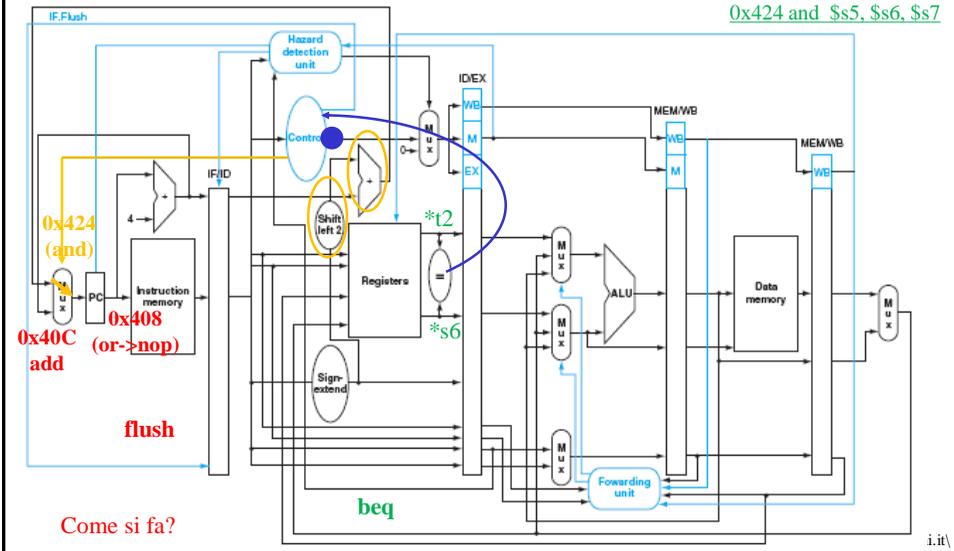




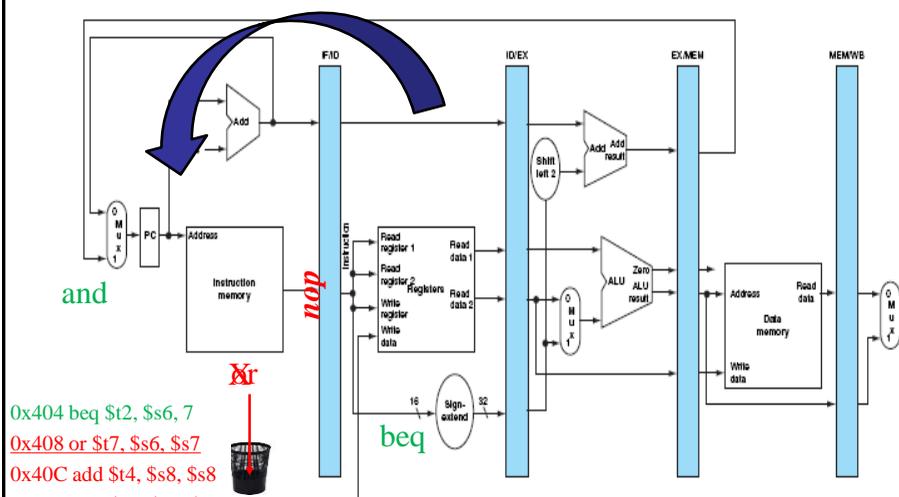
# CPU modificata

*Anticipazione della valutazione della branch.  
Anticipazione del calcolo dell'indirizzo di salto.*

0x404 beq \$t2, \$s6, 7  
0x408 or \$t7, \$s6, \$s7  
0x40C add \$t4, \$s8, \$s8  
0x410 and \$s5, \$s6, \$s7  
...  
0x424 and \$s5, \$s6, \$s7



# Flush dell'istruzione OR



*Nel caso in cui il salto venga preso, occorre trasformare la or (o una qualsiasi altra istruzione) in una nop durante la fase di fetch*



# Istruzione nop



Si sostituisce nella parte master di IF/ID un'istruzione nulla. Un'istruzione che non farà nulla.

Nome campo	op	rs	rt	rd	shamt	funct
Dimensione	6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
sll \$s1, \$s2, 7	000000	X	10010	10001	00111 (7)	000000

\$s1 = \$s2 = \$zero, shmt = 0

Nome campo	op	rs	rt	rd	shamt	funct
Dimensione	6-bit	5-bit	5-bit	5-bit	5-bit	6-bit
sll \$zero, \$zero, 0	000000	00000	00000	00000	00000 (0)	000000

L'istruzione sll con tutti «0» non fa «nulla»! Ma è un'istruzione regolare, che verrà eseguita regolarmente.



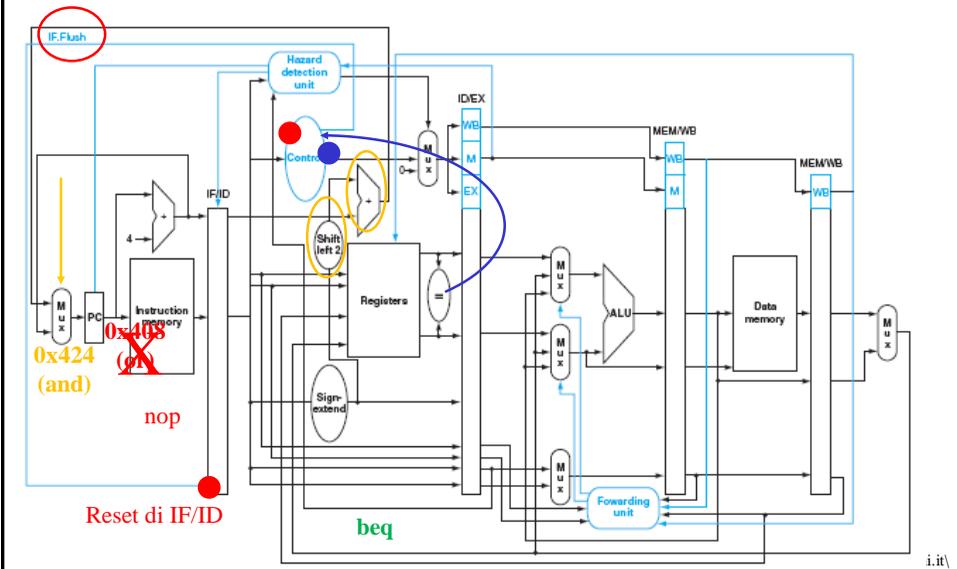
# CPU modificata



*Anticipazione della valutazione della branch.*

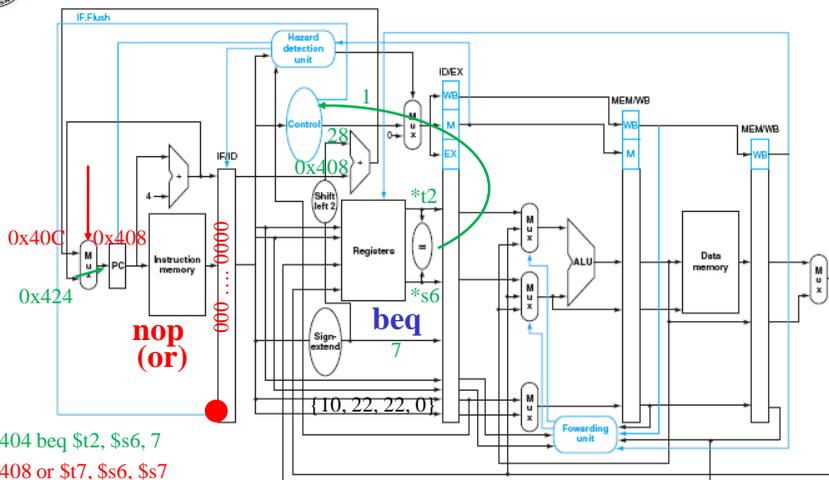
*Anticipazione del calcolo dell'indirizzo di salto (bypass verso l'UC).*

*Flush dell'istruzione or*





## Esempio (salto da prendere) - I



0x404 beq \$t2, \$s6, 7  
 0x408 or \$t7, \$s6, \$s7

...  
 0x424 and \$s5, \$s6, \$s7  
 0x428 addi \$s0, \$s1, 100  
 0x42C lw \$t2, 24(\$t1)  
 0x430 sw \$t2, 28(\$t1)

I segnali di controllo viaggiano “normalmente”. Manca il segnale di branch perchè generato e consumato nella fase di DEC.

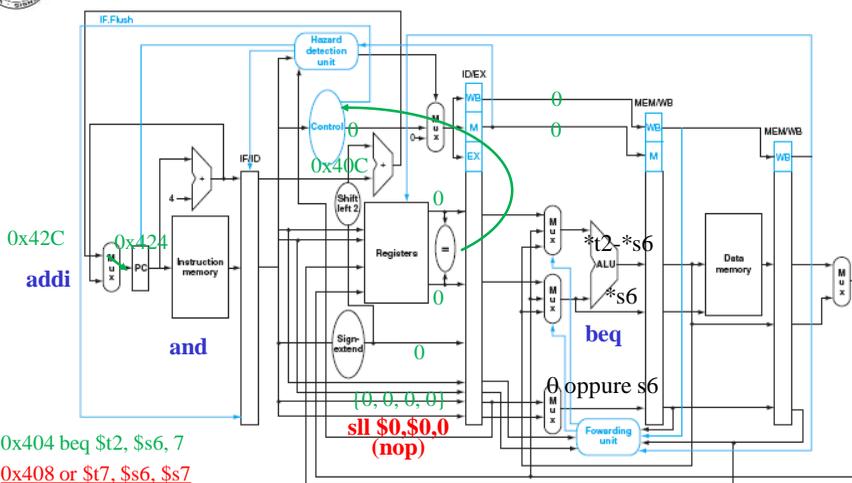
No hazard sui dati

29/56

<http://borghese.di.unimi.it/>



## Esempio (salto da prendere) - II



0x404 beq \$t2, \$s6, 7  
 0x408 or \$t7, \$s6, \$s7

...  
 0x424 and \$s5, \$s6, \$s7  
 0x428 addi \$s0, \$s1, 100  
 0x42C lw \$t2, 24(\$t1)  
 0x43C sw \$t2, 28(\$t1)

I segnali di controllo viaggiano “normalmente”. Manca il segnale di branch perchè generato e consumato nella fase di DEC.

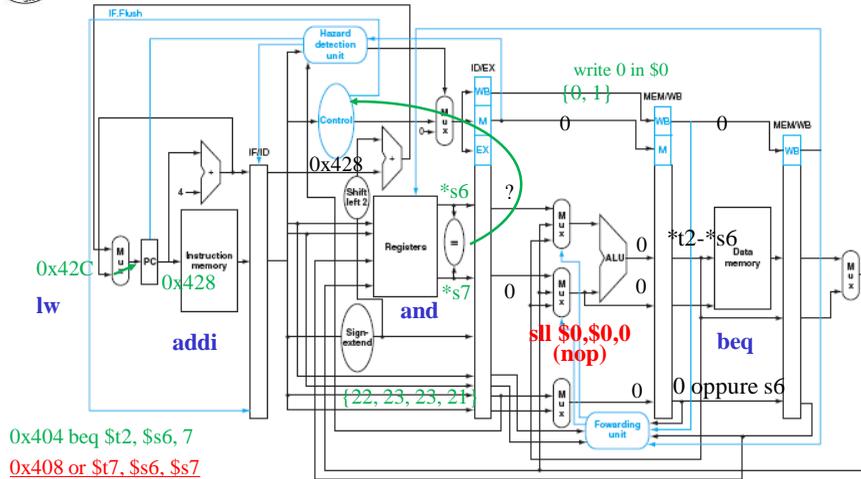
No hazard sui dati

30/56

<http://borghese.di.unimi.it/>



## Esempio (salto da prendere) - III



0x404 beq \$t2, \$s6, 7  
 0x408 or \$t7, \$s6, \$s7

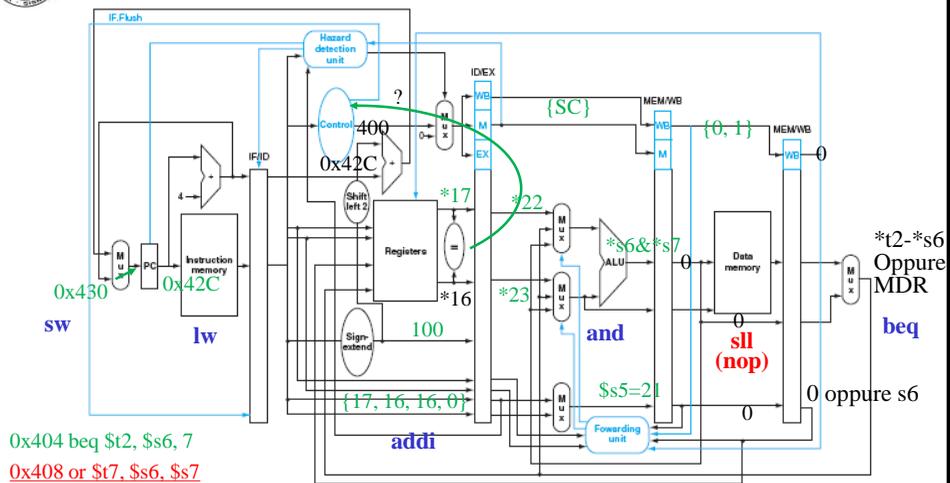
...  
 0x424 and \$s5, \$s6, \$s7  
 0x428 addi \$s0, \$s1, 100  
 0x42C lw \$t2, 24(\$t1)  
 0x430 sw \$t2, 28(\$t1)

I segnali di controllo viaggiano "normalmente". Manca il segnale di branch perchè generato e consumato nella fase di DEC.

No hazard sui dati



## Esempio (salto da prendere) - IV



0x404 beq \$t2, \$s6, 7  
 0x408 or \$t7, \$s6, \$s7

...  
 0x424 and \$s5, \$s6, \$s7  
 0x428 addi \$s0, \$s1, 100  
 0x42C lw \$t2, 24(\$t1)  
 0x430 sw \$t2, 28(\$t1)

I segnali di controllo viaggiano "normalmente". Manca il segnale di branch perchè generato e consumato nella fase di DEC.

No hazard sui dati



# Gestione della criticità



Decisione ritardata -> posso dovere eliminare 1 istruzione (flush).

### Soluzione SW:

Aggiunta di un "branch delay slot": l'istruzione successiva a quella di salto viene sempre eseguita indipendentemente dall'esito della branch.

Contiamo sul compilatore/assemblatore per mettere dopo l'istruzione di salto un'istruzione che andrebbe comunque eseguita indipendentemente dal salto (ad esempio posticipo un'istruzione precedente la branch).

### Soluzione HW:

Si introduce dell'hardware aggiuntivo per prendere la decisione sul salto già in fase di DEC della beq.

Ci si affida all'hardware della CPU per gestire l'eliminazione dell'istruzione in fase di fetch (flush).

PCSrc viene impostato per trasferire l'indirizzo di salto nel PC.

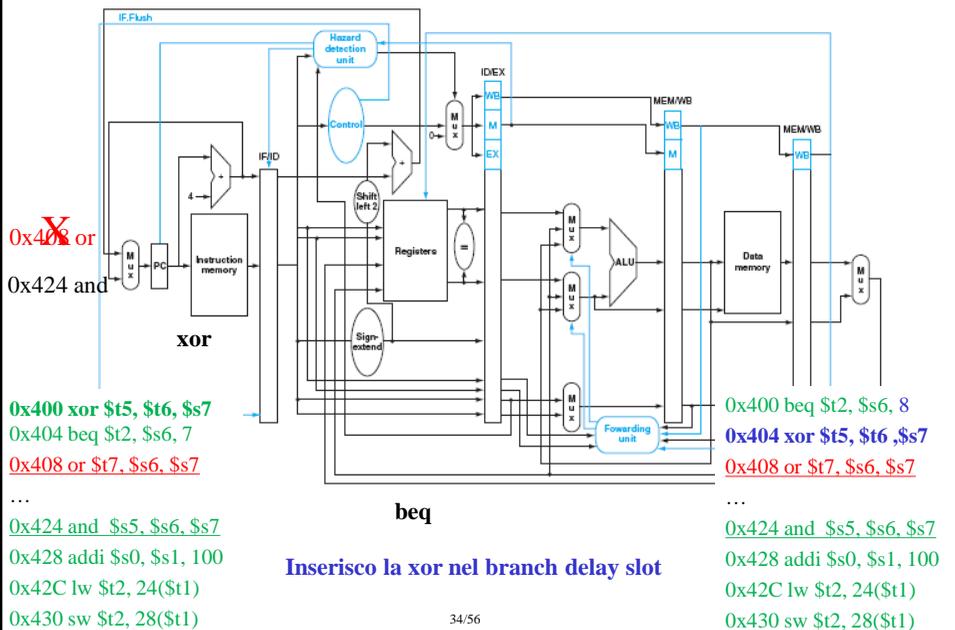
Flush è costoso su pipeline più profonde e ad esecuzione parallela -> si cerca di evitarlo.

**Investimento sui meccanismi di predizione.**

A.



# Branch delay slot





## Esecuzione condizionata



In questo caso la branch è una “normale” istruzione, il cui risultato viene eliminato se l’istruzione non doveva essere eseguita.

Esempio di un’implementazione MIPS:

- `movn $8, $11, $4`

Questa istruzione sposta il contenuto del registro 11 nel registro 8, se il contenuto del registro 4 non è zero.

- `movz $8, $11, $4`



ARM v7 ha esecuzione condizionata di molte istruzioni.

ARM v8 ha ridotto il numero di istruzioni che vengono eseguite in modo condizionato (true RISC)



## Sommario



Hazard sul controllo

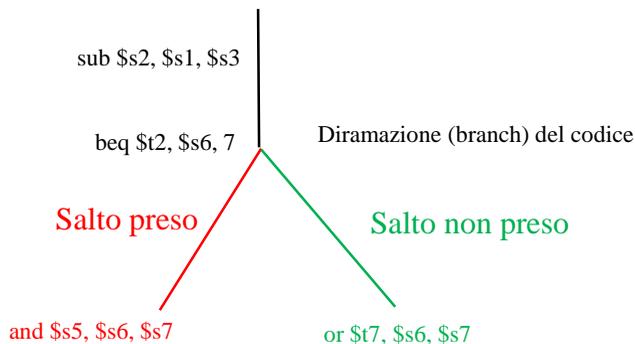
Anticipazione dei salti

Predizione dei salti



## «predire»

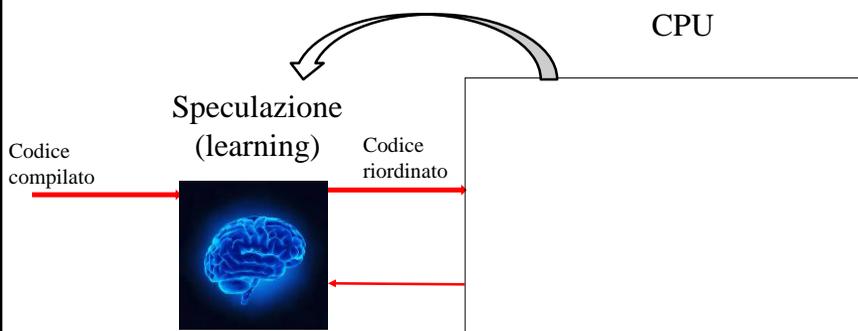
0x400 sub \$s2, \$s1, \$s3
0x404 beq \$t2, \$s6, 7
0x408 or \$t7, \$s6, \$s7
0x40C add \$t4, \$s8, \$s8
0x410 and \$s5, \$s6, \$s7
0x414 add \$t0, \$t1, \$t2
0x418 sw \$s3, 24(\$t1)
0x41C addi \$t7, \$s6, 10
0x420 add \$t8, \$s2, \$s2
0x424 and \$s5, \$s6, \$s7
0x418 add \$t0, \$t1, \$t2



Supponiamo sia corretto continuare in sequenza (salto non preso). **Su cosa basiamo la supposizione a-priori?**



## CPU con Predizione





## Riorganizzazione del codice a seconda della predizione



```
sub $s2, $s1, $s3
beq $t2, $s6, label:
```

```
or $t7, $s6, $s7
add $t4, $s8, $s8
and $s5, $s6, $s7
add $t0, $t1, $t2
add $t8, $s2, $s2
....
```

```
label: and $s5, $s6, $s7
add $t0, $t1, $t2
sw $s3, 24($t1)
addi $t7, $s6, 10
....
```

```
....
```

```
....
```

Codice compilato

Speculazione (learning)



Codice riordinato

CPU

39/56

```
sub $s2, $s1, $s3
bne $t2, $s6, label2
```

```
and $s5, $s6, $s7
add $t0, $t1, $t2
sw $s3, 24($t1)
addi $t7, $s6, 10
....
```

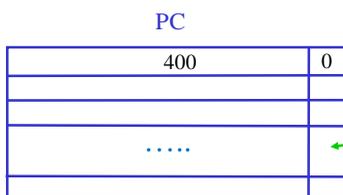
label2: or \$t7, \$s6, \$s7

```
add $t4, $s8, $s8
and $s5, $s6, $s7
add $t0, $t1, $t2
add $t8, $s2, $s2
....
```

```
....
```



## Branch prediction buffer



Bit meno significativi del PC

Bit che indica se l'ultima volta il salto era stato eseguito o meno.

Previsione relativa a una beq con gli stessi bit meno significativi del PC. E' un problema?

```
START: 0x400 beq $t0, $t1, SALTA    for (t0=0;t0=t1;t0++)
        0x404 add $s0, $s1, $s2    {
        0x408 sub $s3, $s4, $s5
        0x40C addi $t0 $t0, 1
        0x410 j START              }
SALTA: 0x414 and $t2, $t3, $t4
```

Predittore = 0 (non salta) finchè dentro il ciclo



## Problemi con questo BPB



```

START: 0x400 beq $t0, $t1, SALTA    for (t0=0;t0<=t1;t0++)
        0x404 add $s0, $s1, $s2      {
        0x408 sub $s3, $s4, $s5
        0x40C addi $t0 $t0, 1
        0x410 j START                }
SALTA: 0x414 and $t2, $t3, $t4

```

0x400	0
-------	---

Durante il ciclo

Predizione (a-priori) = non salto  
Esito (a-posteriori) = non salto (0->0)

0x400	1
-------	---

All'ultima iterazione

Predizione (a-priori) = non salto  
Esito (a-posteriori) = salto (0 -> 1)

0x400	0
-------	---

Alla prima iterazione successiva

Predizione (a-priori) = salto  
Esito (a-posteriori) = non salto (1->0)

**2 errori per ogni ciclo**

Algoritmi migliori di ottimizzazione dello scheduling per predizione ottima del salto.



## 1 errore per ogni ciclo



```

START: 0x400 beq $t0, $t1, SALTA    for (t0=0;t0<=t1;t0++)
        0x404 add $s0, $s1, $s2      {
        0x408 sub $s3, $s4, $s5
        0x40C addi $t0 $t0, 1
        0x410 j START                }
SALTA: 0x414 and $t2, $t3, $t4

```

La prima volta che la predizione non e' allineata con l'esito, non modifico la predizione.

0x400	0
-------	---

Durante il ciclo

Predizione (a-priori) = non salto  
Esito (a-posteriori) = non salto (0->0)

0x400	0
-------	---

All'ultima iterazione

Predizione (a-priori) = non salto  
Esito (a-posteriori) = salto (**non correggo la predizione, 1 solo errore**)

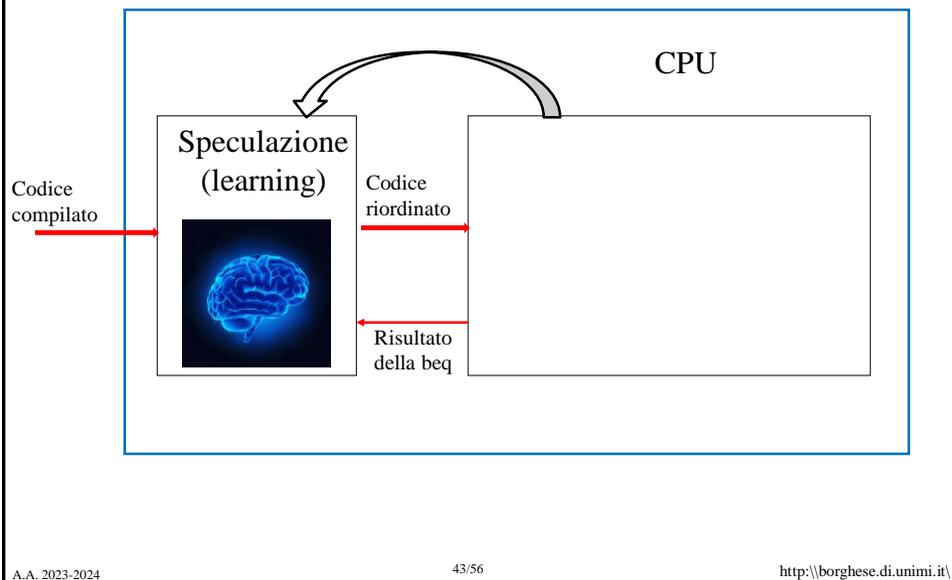
0x400	0
-------	---

Alla prima iterazione successiva

Predizione (a-priori) = salto  
Esito (a-posteriori) = non salto (0->0)



## CPU con Predizione



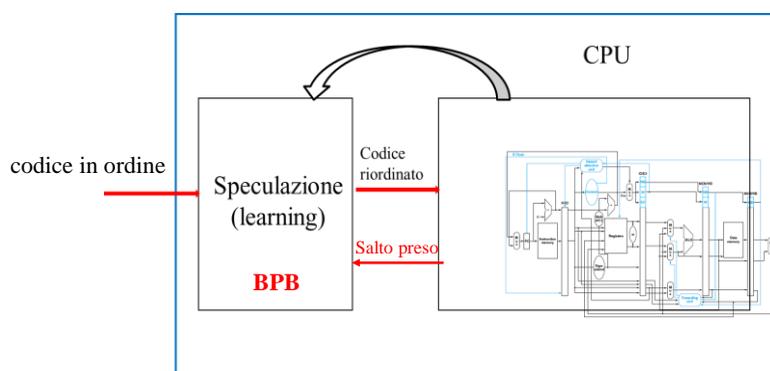
## Branch Prediction buffer a 2 bit - FSM



$\langle I, S, Y, f(S,I), Y(S), S_0 \rangle$

$I = \{ \text{salto\_non\_preso}, \text{salto\_preso} \}$

$Y = \{ \text{salto\_predetto\_non\_preso}, \text{salto\_predetto\_preso} \}$





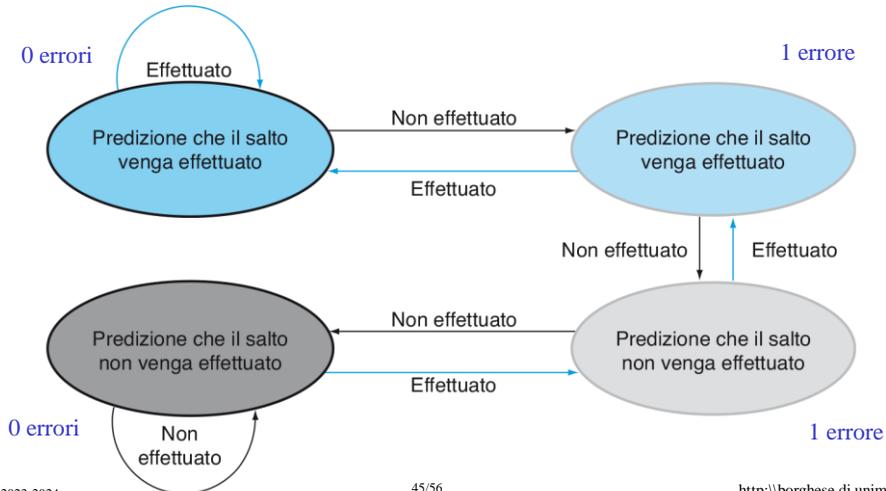
## Branch Prediction buffer a 2 bit - STG



$I = \{\text{salto\_non\_preso}, \text{salto\_preso}\}$

$Y = \{\text{salto\_predetto\_non\_preso}, \text{salto\_predetto\_preso}\}$

$S = \{\text{salto\_non\_preso\_0\_errori}, \text{salto\_non\_preso\_1\_errore}, \text{salto\_preso\_0\_errori}, \text{salto\_preso\_1\_errore}\}$



A.A. 2023-2024

45/56

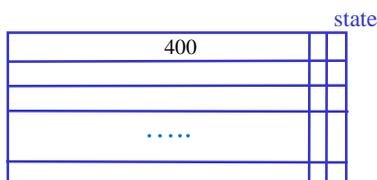
<http://borghese.di.unimi.it/>



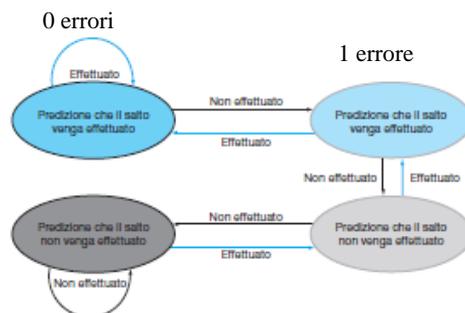
## Branch Prediction buffer a 2 bit - STT



	I=Taken	I=NotTaken	Y
S=Taken 0 errors	Taken 0 errors	Taken 1 error	Predict taken
S = Taken 1 error	Taken 0 errors	Not Taken 1 error	Predict taken
S=Not Taken 0 errors	Not Taken 1 error	Not Taken 0 errors	Predict Not taken
S = Not Taken 1 error	Taken 1 error	Not Taken 0 errors	Predict Not taken



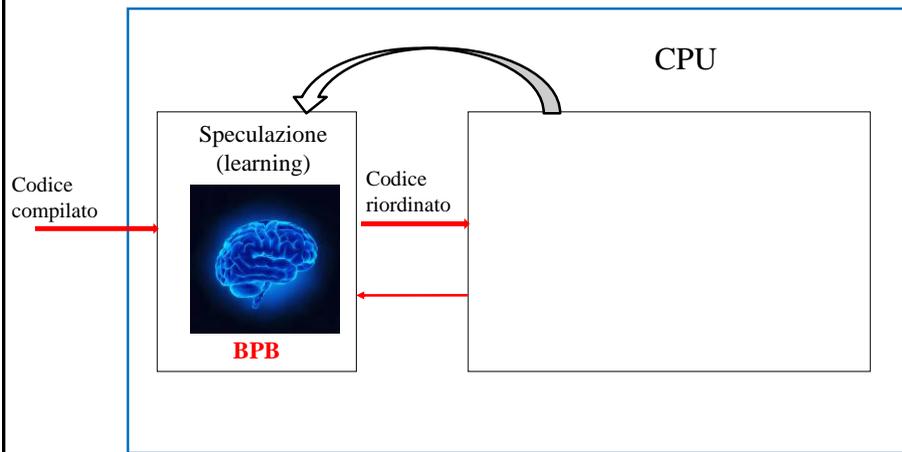
Bit meno significativi del PC



A.A. 2023-2024



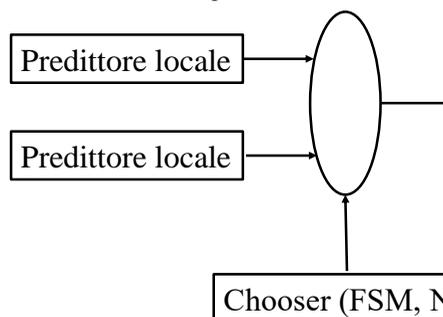
## CPU con Predizione



## Evoluzioni della branch prediction



- 1) **Correlating predictors.** Comportamento locale e globale dei salti. Tipicamente 2 predittori a 2 bit (1 locale e 1 globale). Viene scelto il predittore a seconda che l'ultima volta abbia «indovinato» o meno.
- 2) **Tournament predictors.** Vengono utilizzati predittori multipli a 1 o 2 bit. Anche qui vengono utilizzati spesso un predittore locale un predittore globale. Per ogni branch viene selezionato il predittore migliore in base alla storia di quel particolare salto e anche la scelta può essere effettuata da un predittore a 2 bit.



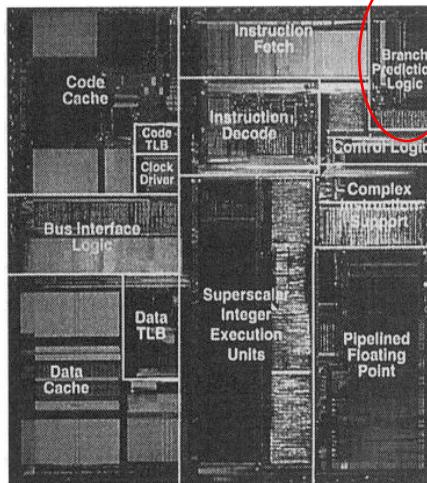
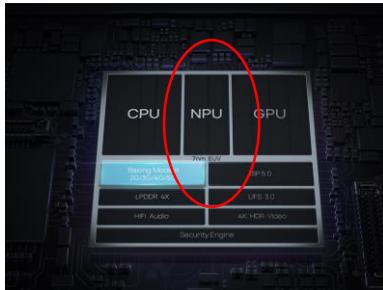


# Evoluzione della logica



**Branch prediction buffer (4 kbyte nella CPU del Pentium 4). Si trova nel circuito di speculazione.**

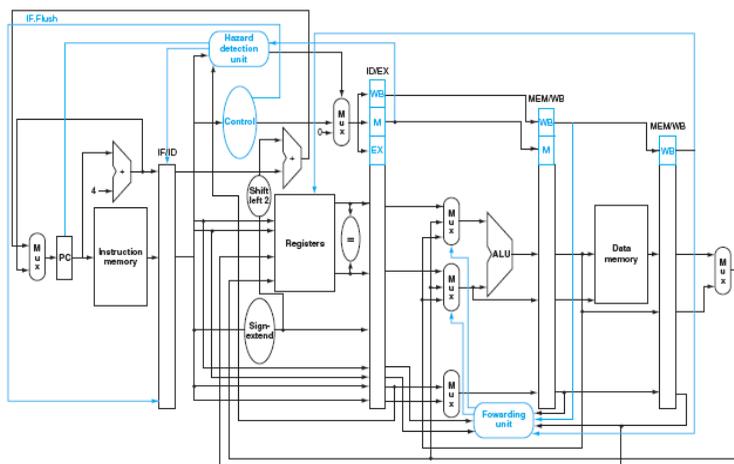
**Neural Processing CPU (anche messa a fuoco, riconoscimento facciale...)**



**Kirin 990, Hawei P40 Serie, 2020**



# Esempio



Data la CPU sopra, specificare il contenuto di **TUTTE** le linee (dati e controllo) quando è in esecuzione il seguente segmento di codice:

```
0x400 addi $t3, $t1, 32
0x404 sub $t4, $t1, $t1
0x408 add $t1, $t2, $t3
0x 40C beq $t2, $s0, 40
0x410 sw $s2, 64($s0)
```

quando l'istruzione di addi si trova in fase di WB. Specificare sullo schema (con colore o con tratto grosso) quali linee, all'interno dei diversi stadi, trasportino dati e segnali di controllo utili all'esecuzione dell'istruzione, riferendosi alla situazione in cui l'istruzione di addi è in fase di WB. **Supponi che il valore predetto per ogni ramo del test della branch di codice sia che il contenuto di \$t1 è \$s0 in un...**

