



Hazard sul controllo

Prof. Alberto Borghese
Dipartimento di Informatica
alberto.borghese@unimi.it

Università degli Studi di Milano

Riferimento al Patterson: 4.8



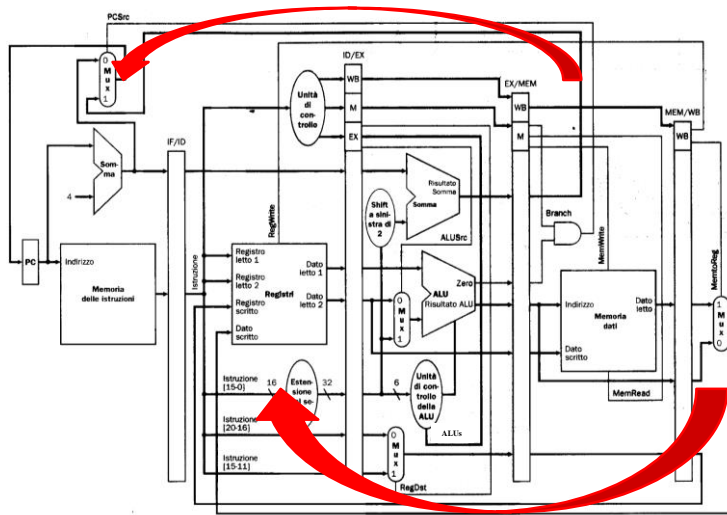
Sommario

Hazard sul controllo

Predizione dei salti



Pipeline – criticità o hazard



la fase di WB genera un cammino da dx (WB) a sx (DEC) sul data path
 beq genera un cammino da dx (MEM) a sx (FF) sul control path



Come affrontare gli *hazard su controllo*



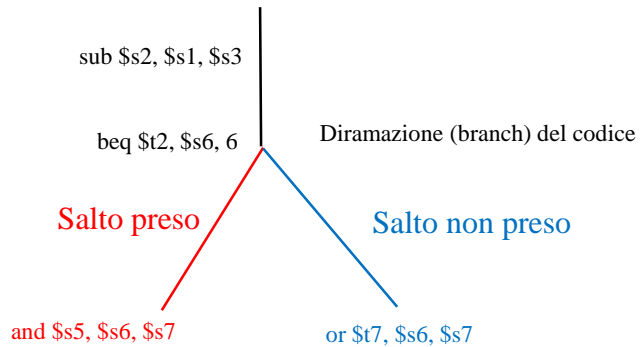
- Si può risolvere l'hazard...
 - *...aspettare*
 - si mette lo stage opportuno dell'istruzione dipendente dalla precedente in pausa
 - il controllo della pipeline deve individuare il problema prima che avvenga! **Stallo**.
 - *...prevenire*
 - Il compilatore o la CPU, come ottimizzazione, può riordinare le operazioni in modo che il risultato sia lo stesso ma non ci siano hazard (**branch delay slot**)
 - *...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 i 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.

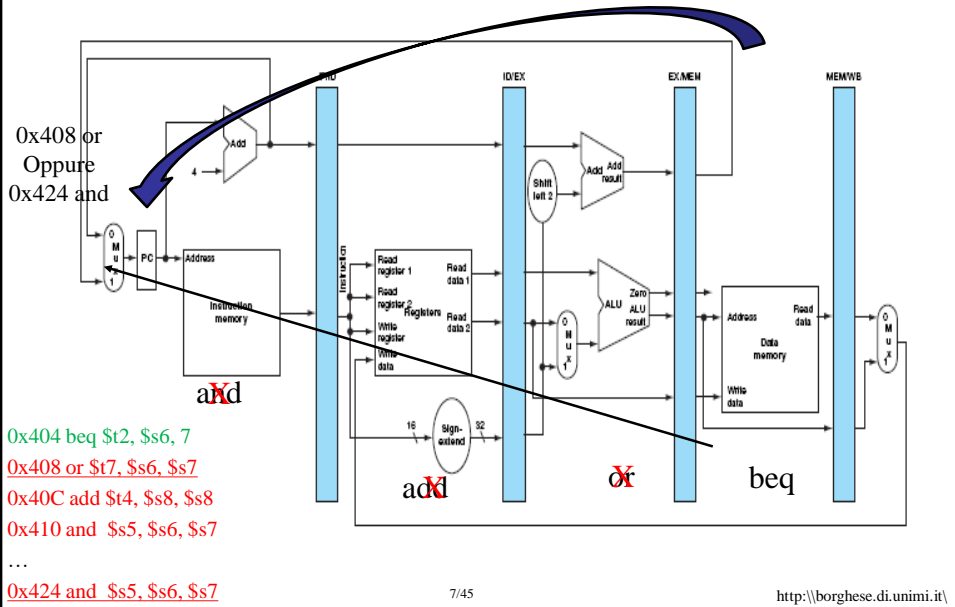
Non c'è nulla di veramente efficace contro gli hazard sul controllo (non si può propagare nulla).

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.



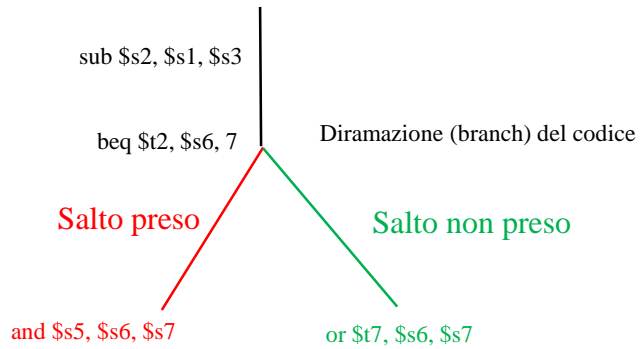
“Aspettare” 3 cicli di clock non è una buona idea



«predire»



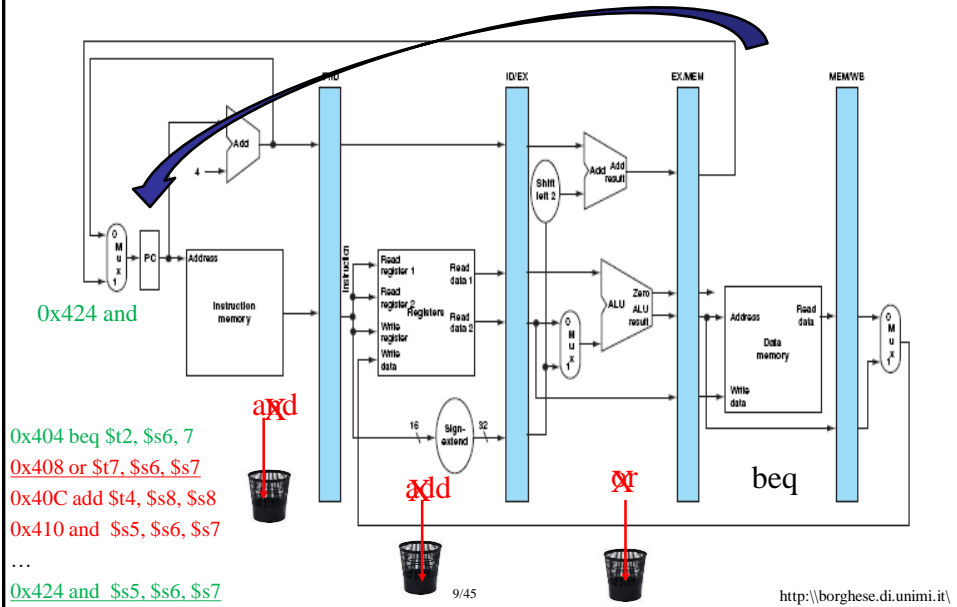
0x400 sub \$s2, \$s1, \$s3
0x404 beq \$t2, \$s6, 7
0x408 or \$t7, \$s6, \$s7
0x40C add \$t4, \$s8, \$s8
0x410 and \$s5, \$s6, \$s7
0x414 add \$t0, \$t1, \$t2
0x418 sw \$s3, 24(\$t1)
0x41C addi \$t7, \$s6, 10
0x420 add \$t8, \$s2, \$s2
0x424 and \$s5, \$s6, \$s7
0x418 add \$t0, \$t1, \$t2



Supponiamo sia corretto continuare in sequenza (salto non preso). E se la supposizione risultasse **non corretta**?



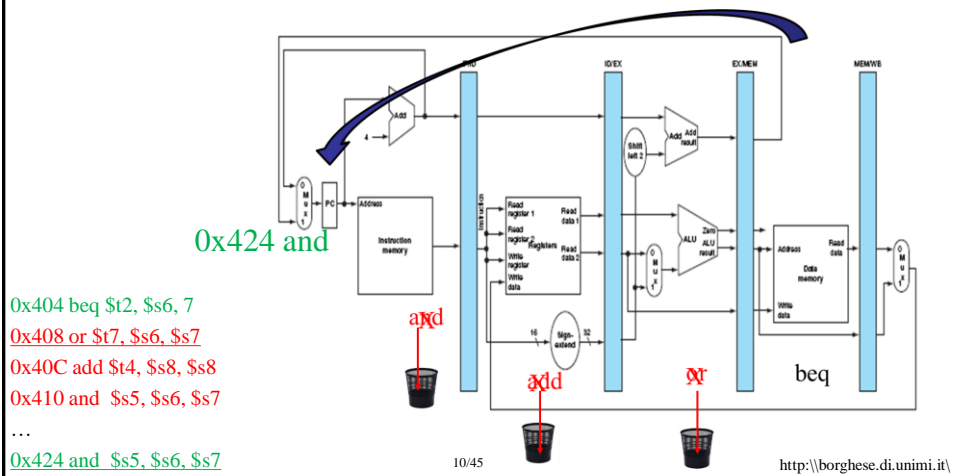
Flush delle 3 istruzioni



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



Obbiettivi:

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

800:	sub \$s2, \$s1, \$s3	IF	ID	EX \$s1- \$s3	MEM	WB s->\$2			
804:	beq \$t2, \$s6, 7		IF	ID	EX Zero if (\$s2 == \$s5)	MEM	WB		
808:	or \$t7, \$s6, \$s7			IF	ID	EX	MEM	WB	
824:	and \$s5, \$s6, \$s7				IF	ID	EX	MEM	WB

or ->  e poi proseguo con la add regolarmente.



1) Identificare l'Hazard nella fase ID



Supponiamo il salto non preso

Se il salto viene preso, ho creato un hazard che devo risolvere.

Come identifico che il salto doveva 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”.

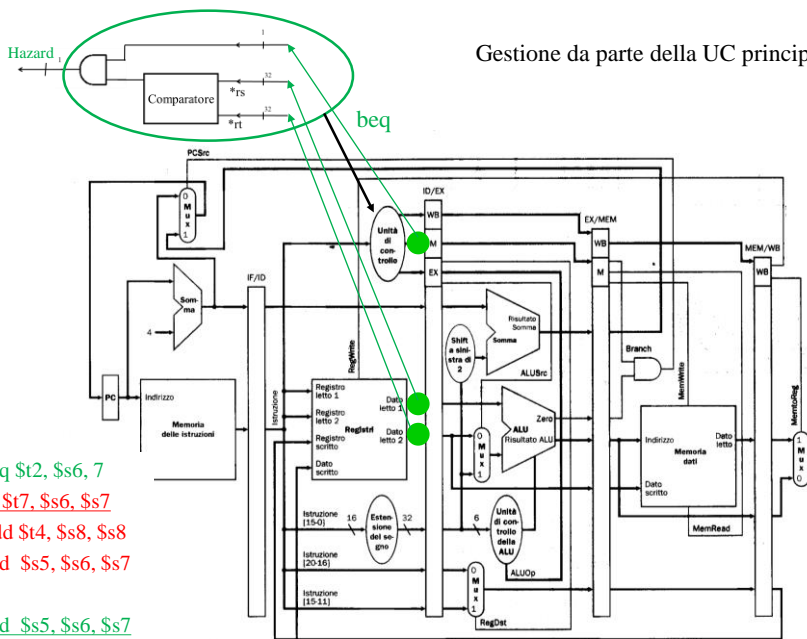
```
If (*rs == *rt) & (branch) then
    hazard
```



UC che identifica hazard sul controllo



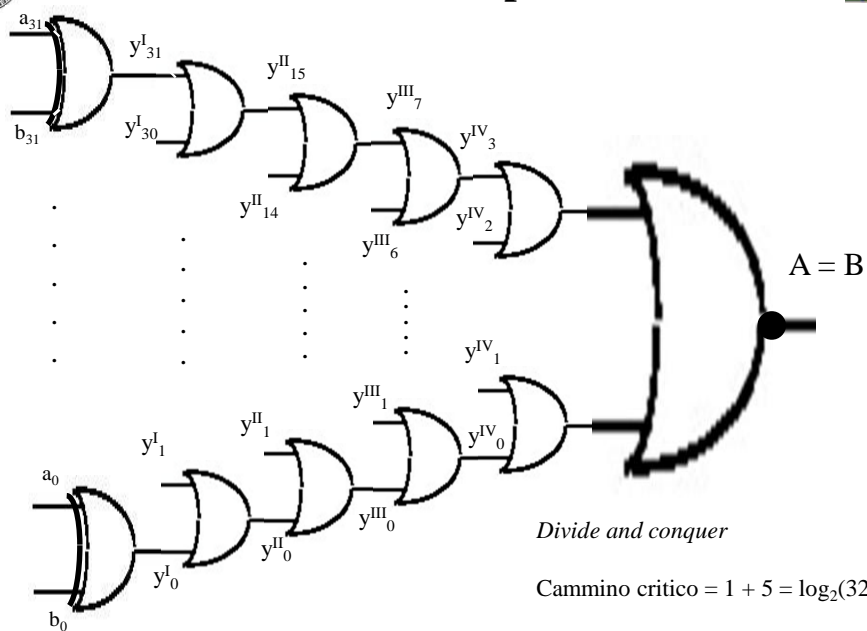
Gestione da parte della UC principale



- 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

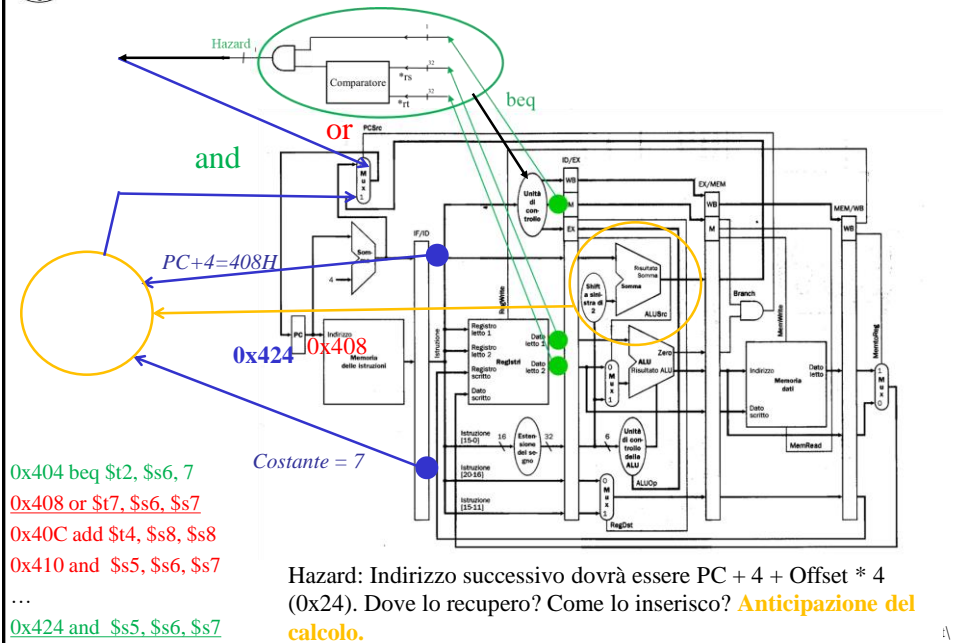


Cammino critico di un comparatore a 32 bit





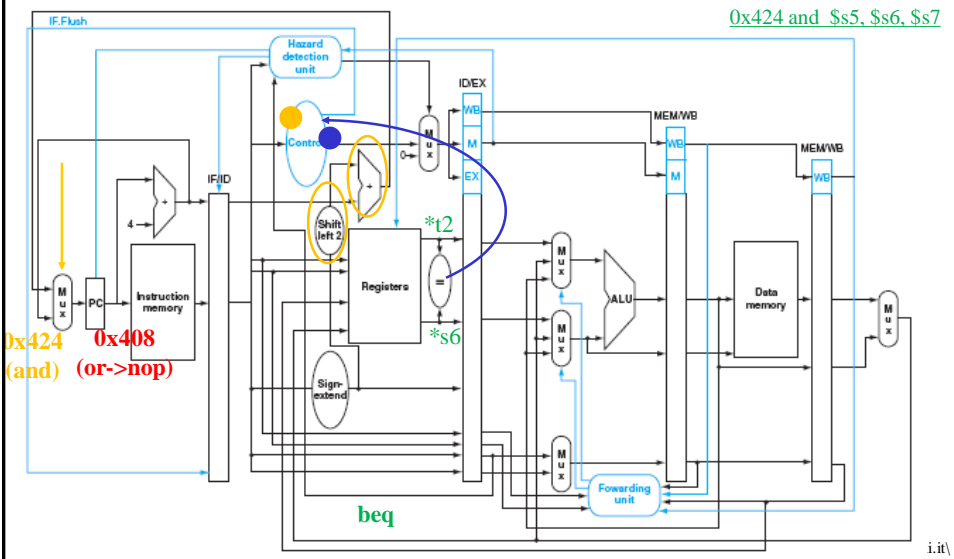
2. Saltare all'istruzione destinazione



CPU modificata

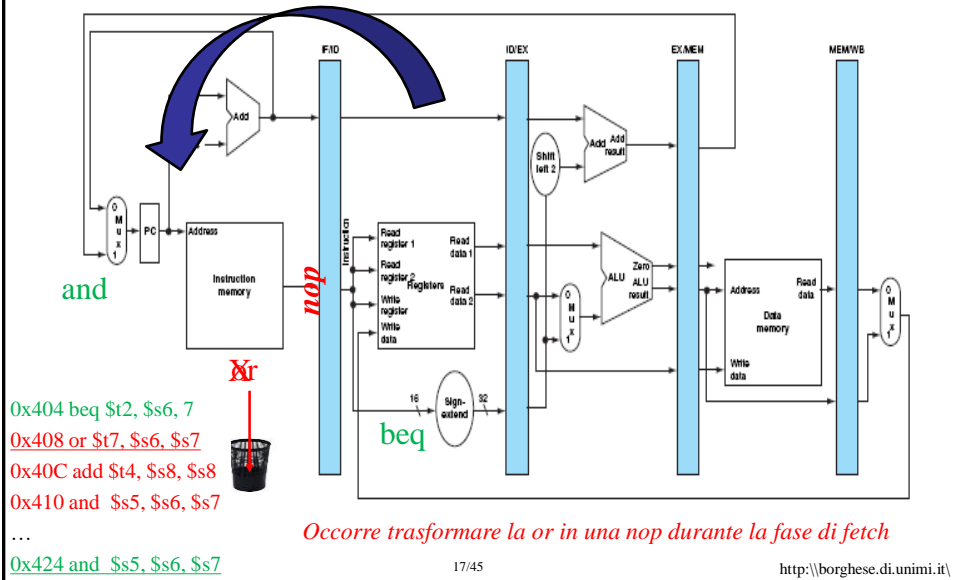
Anticipazione della valutazione della branch.
Anticipazione del calcolo dell'indirizzo di salto.

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

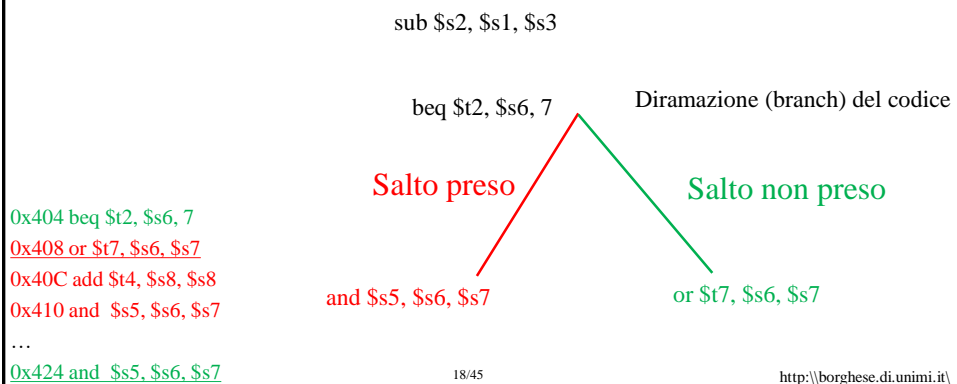


La situazione



Quando la beq è in fase di DECODE posso decidere:

- 1) Procedo in sequenza. La or procede all'interno della pipeline e l'istruzione successiva alla or sarà "regolarmente" la add.
- 2) Prendo il salto. La or deve essere eliminata dalla pipeline e l'istruzione eseguita successivamente sarà la and destinazione del salto.





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



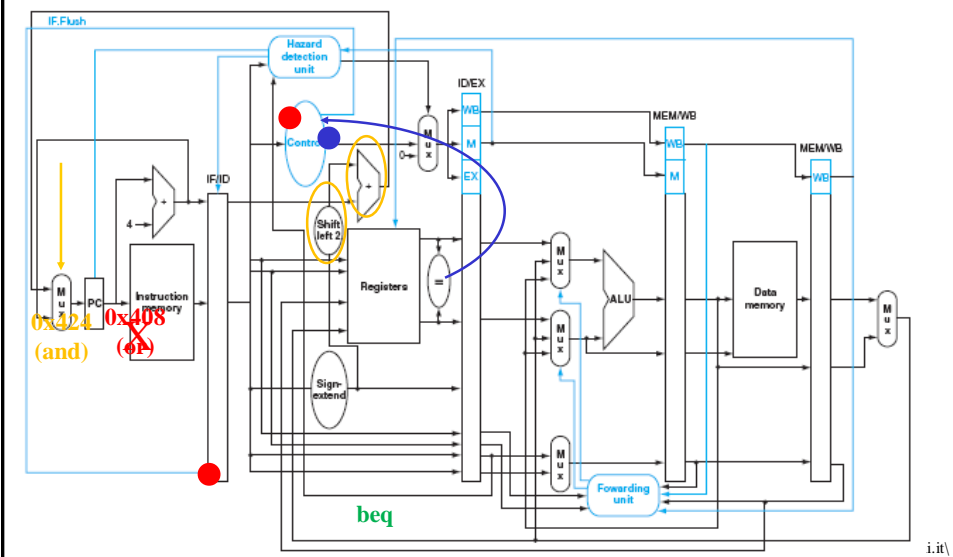
CPU modificata



Anticipazione della valutazione della branch.

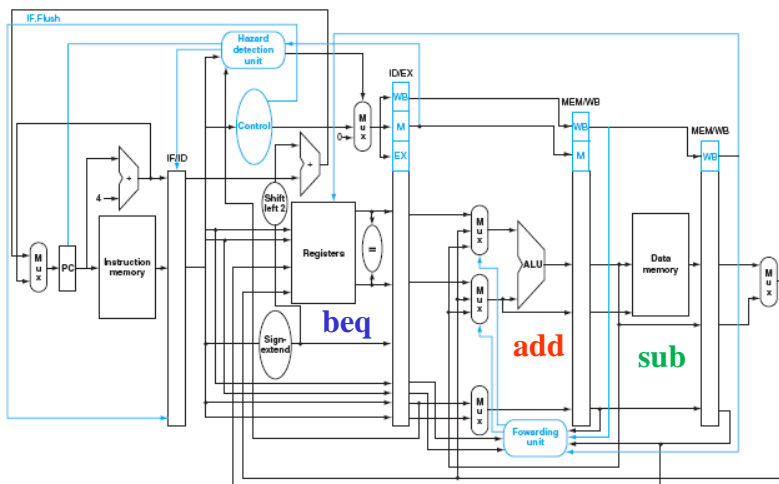
Anticipazione del calcolo dell'indirizzo di salto (bypass verso l'UC).

Flush dell'istruzione or





Criticità sui dati aggiuntive



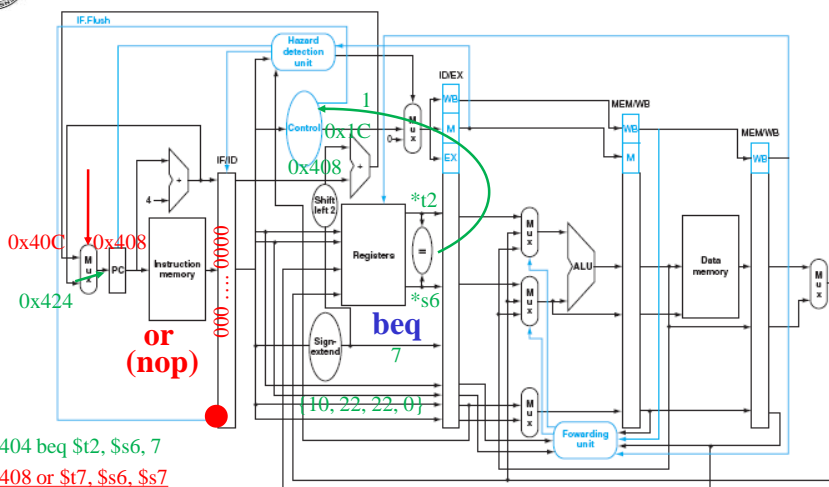
0x39C sub \$t2, \$s4, \$s3
 0x400 add \$t2, \$s4, \$s3
 0x404 beq \$t2, \$s6, 7

Forwarding unit revised:

- Stallo dell'istruzione di add
- Bypass fino al comparatore del risultato della sub



Esempio (salto da prendere) - I



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

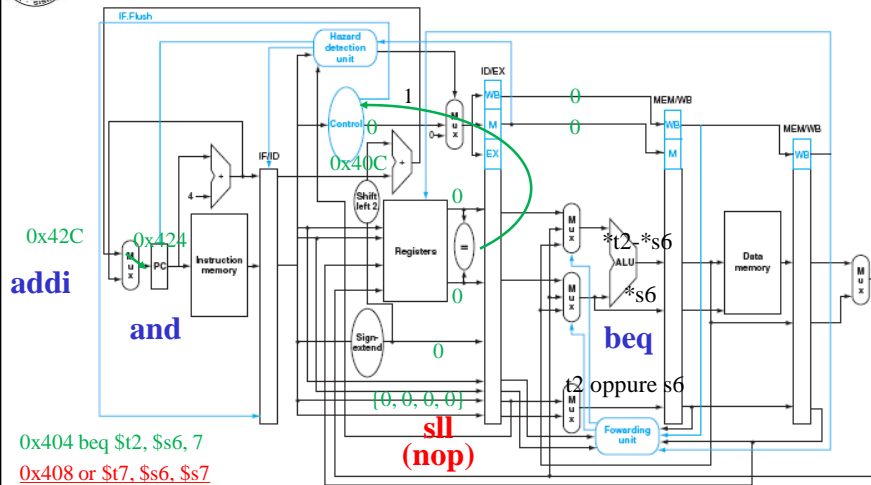
...
 0x424 and \$s5, \$s6, \$s7
 0x428 addi \$s0, \$s1, 100
 0x42C lw \$t2, 24(\$t1)
 0x430 sw \$t2, 28(\$t1)

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

No hazard sui dati



Esempio (salto da prendere) - II



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

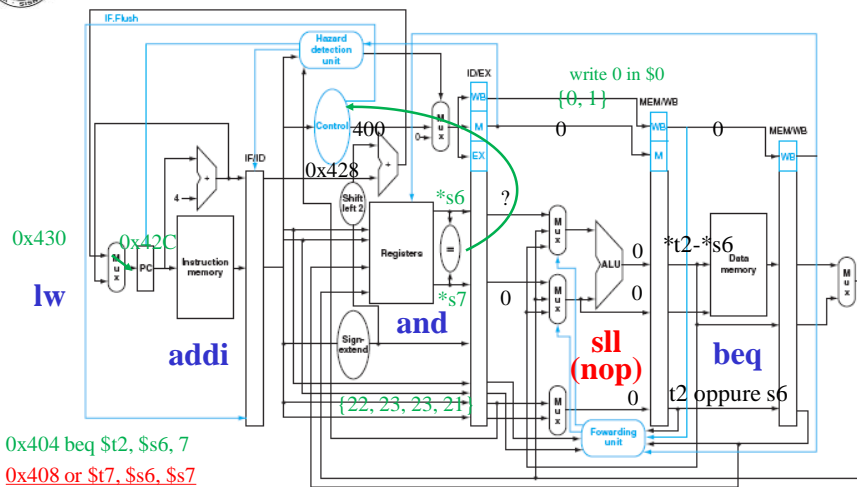
...
 0x424 and \$s5, \$s6, \$s7
 0x428 addi \$s0, \$s1, 100
 0x42C lw \$t2, 24(\$t1)
 0x43C sw \$t2, 28(\$t1)

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

No hazard sui dati



Esempio (salto da prendere) - III



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

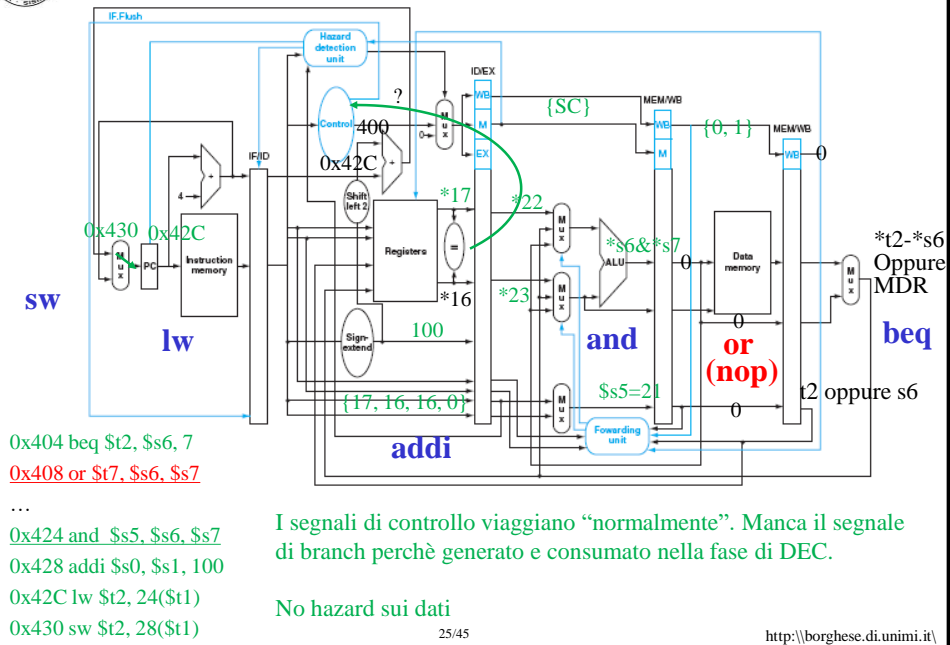
...
 0x424 and \$s5, \$s6, \$s7
 0x428 addi \$s0, \$s1, 100
 0x42C lw \$t2, 24(\$t1)
 0x430 sw \$t2, 28(\$t1)

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



Gestione della criticità



Decisione ritardata.

Soluzione SW:

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

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

Soluzione HW:

Ci si affida all'hardware della CPU per gestire l'eliminazione delle istruzioni (flush).

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

Investimento sui meccanismi di predizione.



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



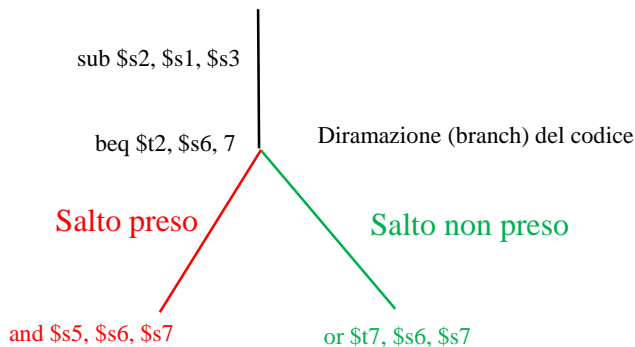
Hazard sul controllo

Predizione dei salti



«predire»

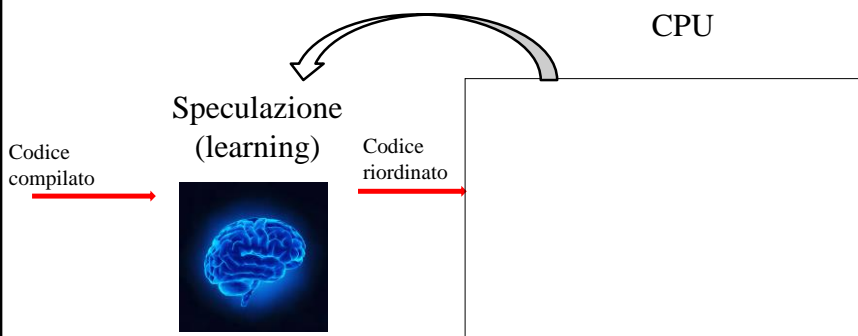
0x400 sub \$s2, \$s1, \$s3
0x404 beq \$t2, \$s6, 7
0x408 or \$t7, \$s6, \$s7
0x40C add \$t4, \$s8, \$s8
0x410 and \$s5, \$s6, \$s7
0x414 add \$t0, \$t1, \$t2
0x418 sw \$s3, 24(\$t1)
0x41C addi \$t7, \$s6, 10
0x420 add \$t8, \$s2, \$s2
0x424 and \$s5, \$s6, \$s7
0x418 add \$t0, \$t1, \$t2



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



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

31/45

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

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

```
....
```

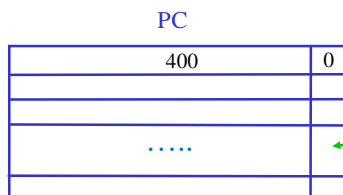
label2: or \$t7, \$s6, \$s7

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

```
....
```



Branch prediction buffer



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

Bit meno significativi del PC

Previsione relativa ad 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 = 1 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 = non salto
Esito = non salto (0->0)

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

All'ultima iterazione

Predizione = non salto
Esito = salto (0 -> 1)

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

Alla prima iterazione successiva

Predizione = salto
Esito = 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 = non salto
Esito = non salto (0->0)

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

All'ultima iterazione

Predizione = non salto
Esito = salto (**non correggo la predizione, 1 errore**)

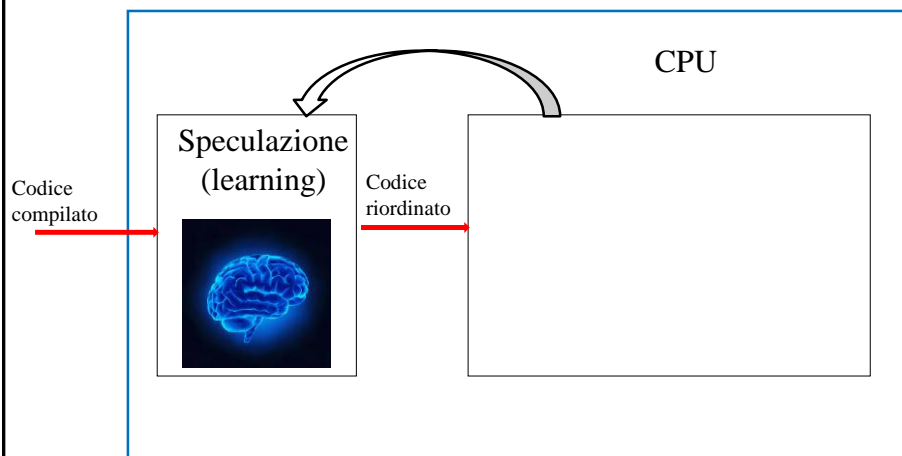
0x400	0
-------	---

Alla prima iterazione successiva

Predizione = salto
Esito = non salto (0->0)



CPU con Predizione



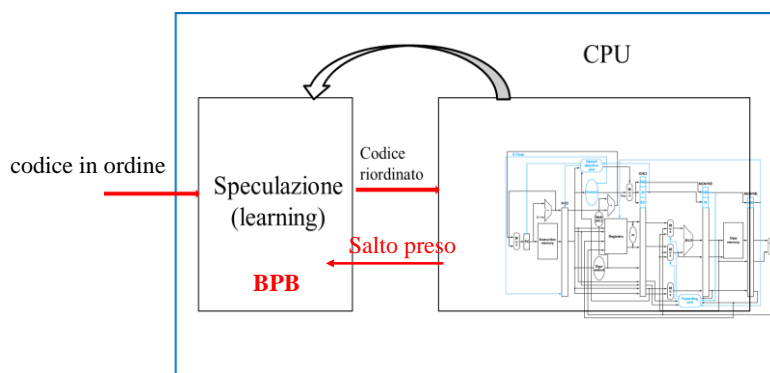
Branch Prediction buffer a 2 bit - FSM



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

$I = \{ \text{salto_non_preso}, \text{salto_preso} \}$

$Y = \{ \text{salto_predetto_non_preso}, \text{salto_predetto_preso} \}$





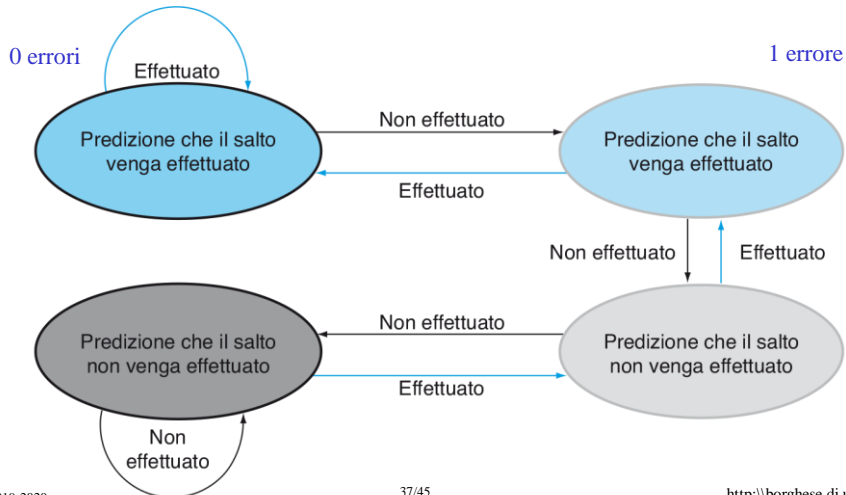
Branch Prediction buffer a 2 bit - STG



$I = \{\text{salto_non_preso}, \text{salto_preso}\}$

$Y = \{\text{salto_predetto_non_preso}, \text{salto_predetto_preso}\}$

$S = \{\text{salto_non_preso_0_errori}, \text{salto_non_preso_1_errore}, \text{salto_preso_0_errori}, \text{salto_preso_1_errore}\}$



A.A. 2019-2020

37/45

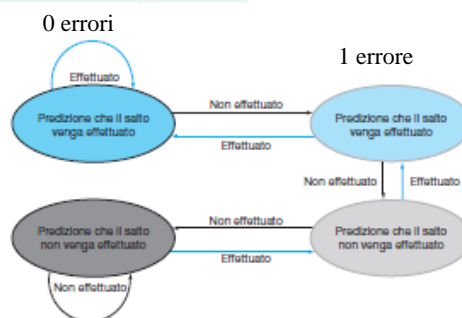
<http://borghese.di.unimi.it/>



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



A.A. 2019-2020



Salto incondizionato

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

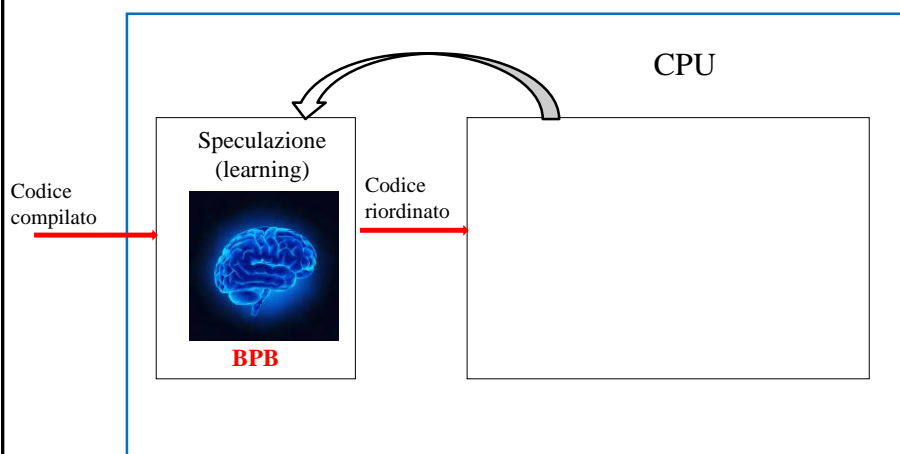
	400:	add \$s0, \$s1, \$s2		400:	add \$s0, \$s1, \$s2
	404:	j 80000		80000:	or \$t0, \$t1, \$t2
Label	408:	and \$s1, \$s2, \$s3		80004:	sub \$t3, \$t4, \$t5
	80000:	or \$t0, \$t1, \$t2			
	80004:	sub \$t3, \$t4, \$t5			

j “lavora” nella fase di decodifica. Viene eseguita un’istruzione prima del salto: delayed jump. **Riempio tutti gli slot di esecuzione.**

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



CPU con Predizione

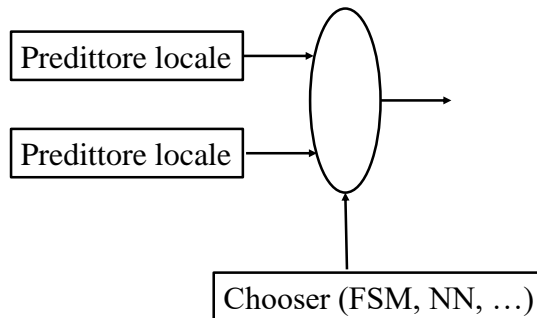




Evoluzioni della branch prediction



- 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 preduttore 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.



A.A. 2019-2020

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

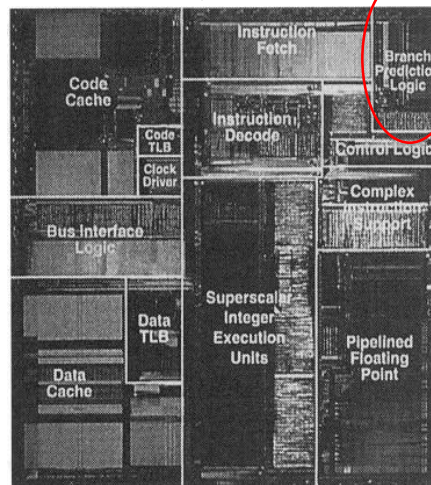
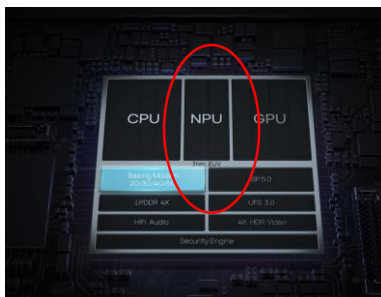


Evoluzione della logica



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...)

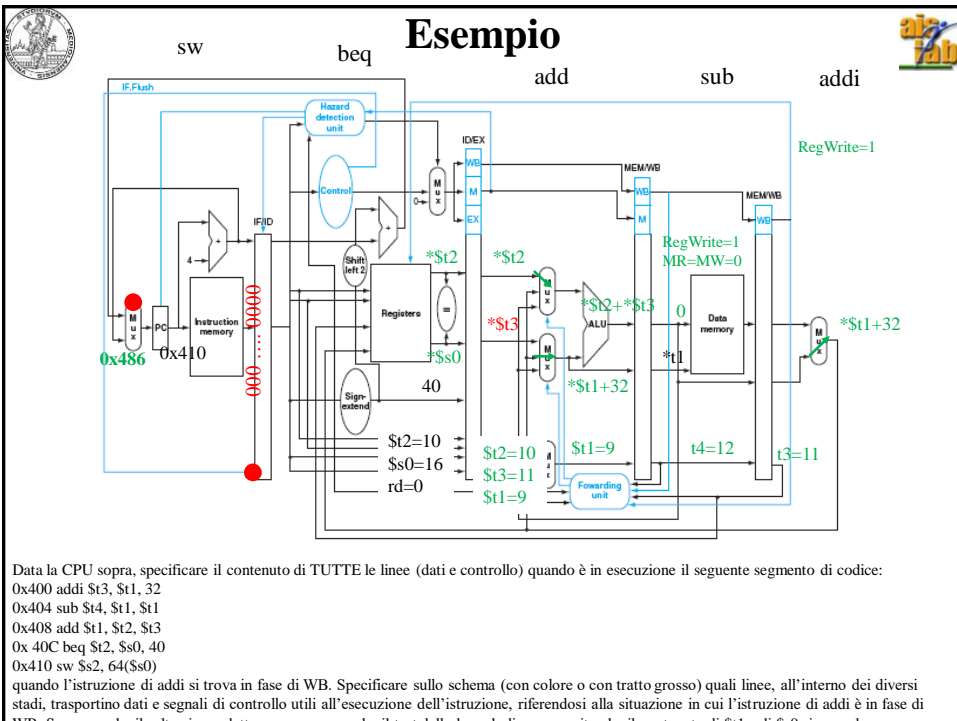
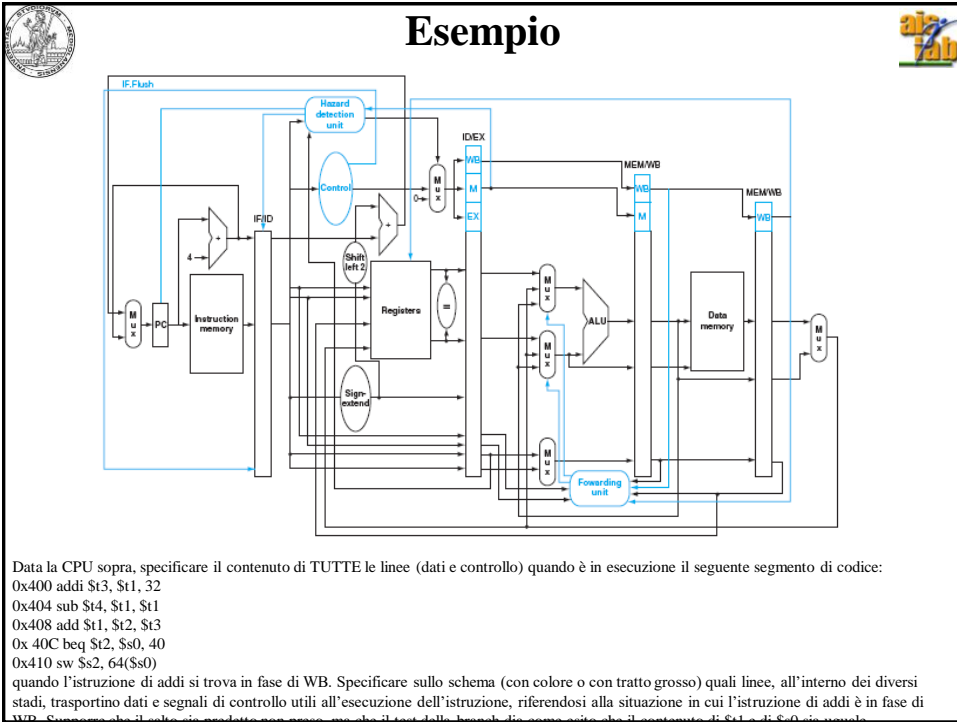


Kirin 990, Hawei P40 Serie, 2020

A.A. 2019-2020

42/45

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





Sommario



Hazard sul controllo

Predizione dei salti