

Esercitazione del 05/05/2010 - Soluzioni

1) Criticità nella distribuzione dei dati e dei flussi

Anche se in media una CPU pipelined richiede un ciclo di clock per istruzione, le singole istruzioni richiedono più cicli di clock per essere portati a termine. Può capitare che un'istruzione richieda un dato che è stato elaborato da un'istruzione precedente ma che non è ancora disponibile, o perché non è stato ancora computato/estratto dalla memoria (criticità nei dati non risolvibile) o perché è ancora in viaggio all'interno della pipe (criticità nei dati risolvibile). Oppure può capitare che l'esecuzione raggiunga un salto condizionato. In questo caso la criticità risiede nel fatto che finché non è stato eseguito il salto non si sa quale istruzione inserire nella pipe, se quella direttamente seguente o quella di destinazione del salto. Più è lunga la pipe, più alto è il numero di istruzioni teoricamente eseguibili contemporaneamente, più è alta la probabilità che insorgano criticità che possono rallentare la pipe.

2) Criticità risolvibili

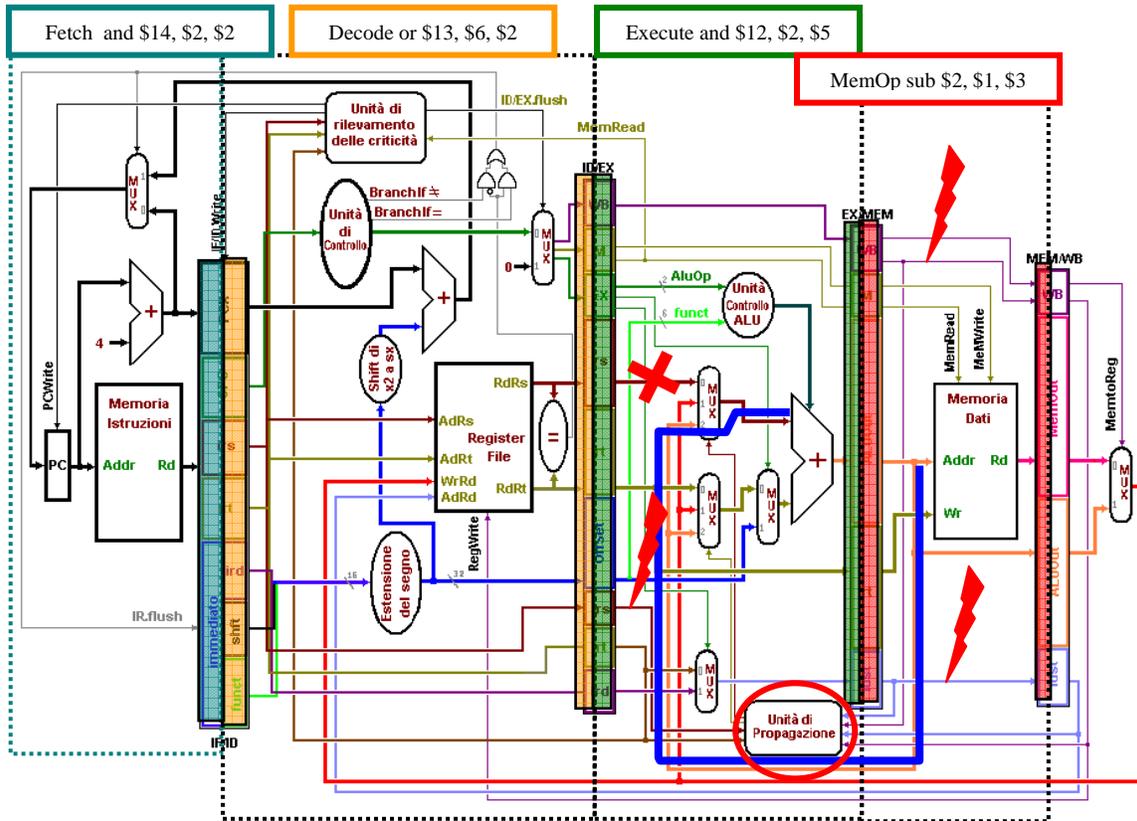
Consideriamo un caso di criticità sui dati risolvibile:

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
and $14, $2, $2
sw $15, 100($2)
```

Nell'architettura pipeline dell'esempio l'istruzione **sub \$2,\$1,\$3** dopo essere stata decodificata richiede 3 cicli per scrivere il risultato nel register file. Ne segue che, a meno di qualche accorgimento le tre istruzioni seguenti leggeranno un dato in \$2 che non corrisponde a quello corretto¹. Si potrebbe risolvere il problema inserendo tre istruzioni neutre NOP (*No Operation*) che ottengono l'effetto di mettere in stallo la pipe per tre cicli. Questo però rallenterebbe troppo la CPU. Esaminando il flusso dei dati all'interno della CPU comunque si può notare che il valore corretto di \$2, quando occorre alla seconda istruzione, è già presente all'interno della *pipe* anche se non è ancora stato scritto nel registro \$2. Si può quindi pensare anticipare la consegna del dato all'istruzione che segue, nello stadio in cui serve, costruendo un opportuno data-path. In altre parole, nel caso che si verifichi questa criticità risolvibile, si può pensare di **propagare all'indietro nella pipe** il risultato in anticipo rispetto allo svolgimento normale della *pipe*.

¹ Questo vale almeno secondo la semantica normalmente adottata per i programmi imperativi. In un programma ci si aspetta che l'effetto di un'istruzione si concretizzi completamente al momento dell'esecuzione dell'istruzione stessa, *senza ritardi di propagazione*. Nulla vieta comunque cambiare la semantica e lasciare al programmatore l'onere di gestire questi effetti inconsueti.

Questa è la situazione al quarto ciclo di clock dell'esempio considerato. Il primo stadio esegue la fase di *fetch* della quarta istruzione, il secondo stadio esegue la decodifica della terza istruzione, il terzo stadio esegue la seconda istruzione mentre il quarto stadio esegue le operazioni sulla memoria dati relative alla prima istruzione. Il PC come prima è incrementato di 4.



Senza accorgimenti, l'esecuzione dell'istruzione **and \$12, \$2, \$5**, presente nello stadio di *Execute*, userebbe il dato **\$2** estratto dal *Register File* al secondo ciclo di questo esempio ed in generale diverso da quello calcolato dalla prima istruzione. Il dato corretto comunque è già presente in questo ciclo di clock nel registro temporaneo **EX/MEM**. Nello stesso registro è anche presente l'indice del registro di destinazione dell'istruzione di **sub (\$2)** nonché l'informazione che si tratta di una scrittura (**WB**). Se propaghiamo lungo la pipe anche l'indice dei registri utilizzati dalla **ALU**, possiamo allora verificare tramite un opportuno circuito (unità di propagazione nello schema) quando si verifica questo tipo di criticità ed anticipare il risultato dell'operazione precedente a quelle che seguono.

3) Criticità non risolvibili

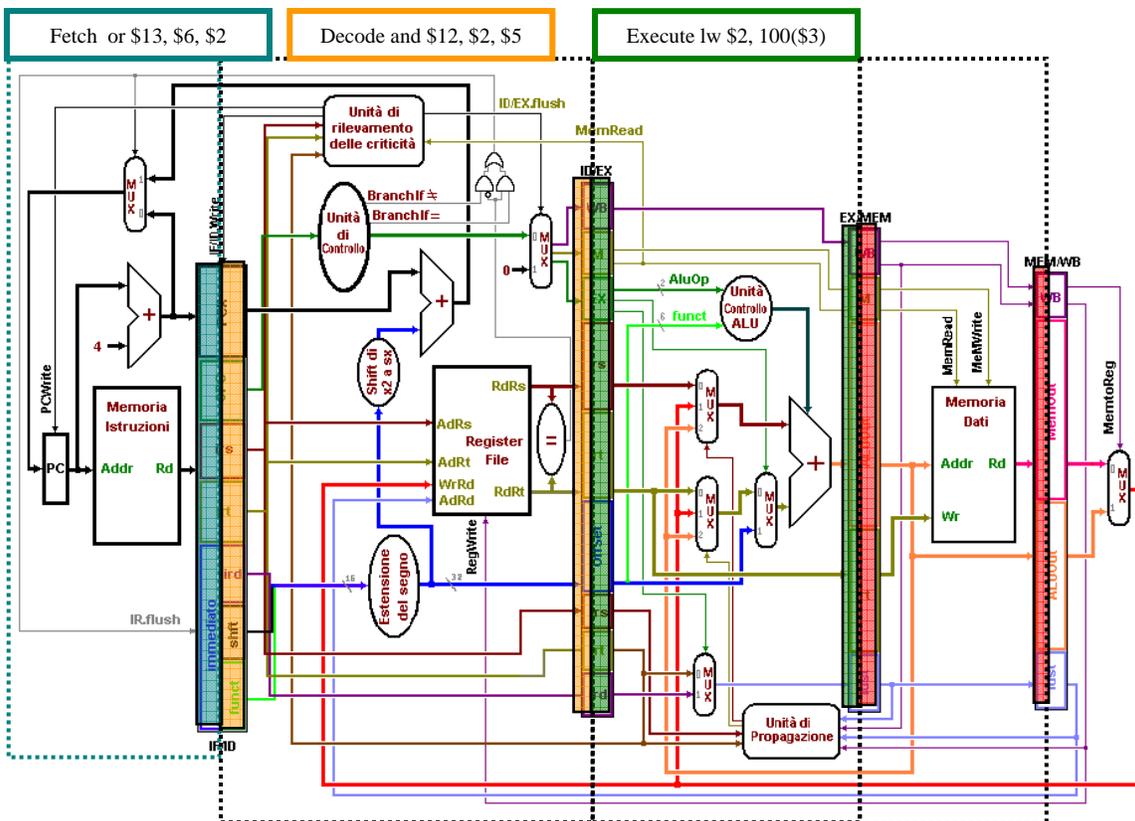
A volte non è possibile anticipare il risultato di un'istruzione perchè il dato non è ancora stato calcolato. Consideriamo la seguente sequenza di istruzioni:

lw \$2, 100(\$3)
and \$12, \$2, \$5
or \$13, \$6, \$2

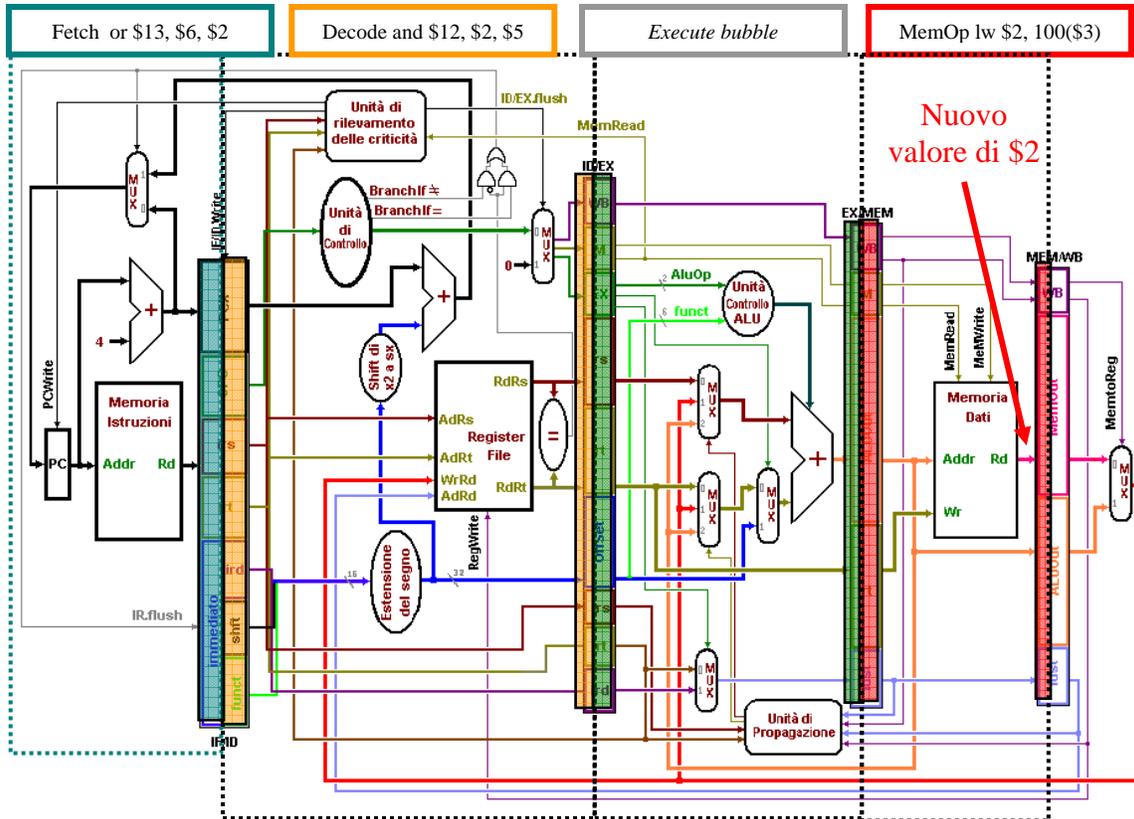
Come nel caso precedente esiste una dipendenza tra l'istruzione **lw** e le istruzioni direttamente seguenti. In questo caso, se lasciassimo procedere la pipe senza modifiche, al momento dell'esecuzione dell'istruzione **and** il dato della cella **100(\$3)** non sarebbe ancora stato estratto dalla memoria e quindi non sarebbe possibile anticiparne il risultato (criticità non risolvibile). Occorre quindi fermare la pipe almeno per un ciclo in attesa che il dato divenga disponibile nello stadio di *WriteBack*.

La condizione di criticità può comunque essere rilevata un ciclo prima che si verifichi il problema nello stadio di *Execute*, quando l'istruzione **and** è ancora in fase di *Decode*. Se l'istruzione che è presente nello stadio di *Execute* è una **Load** che ha conflitti con l'istruzione che segue, allora si può risolvere bloccando la pipe negli stadi *Fetch* e *Decode* e annullando la decodifica dell'istruzione che segue.

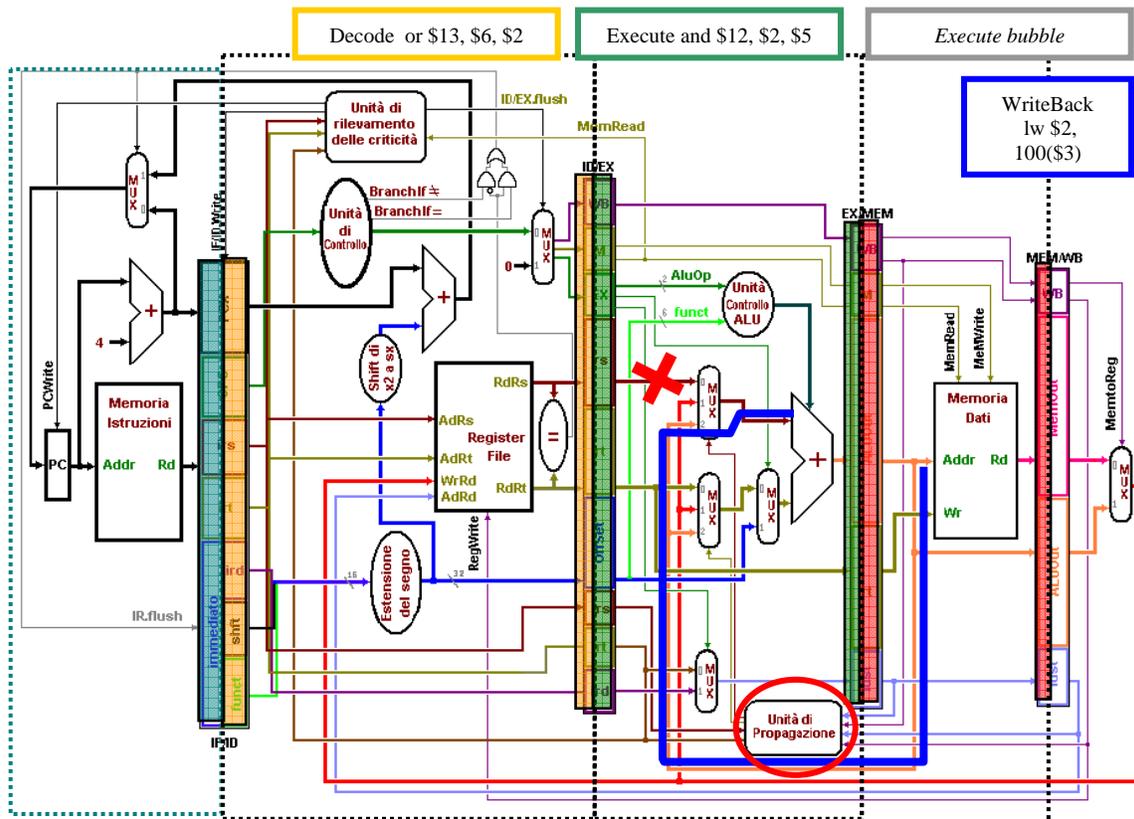
Questa è la situazione al terzo ciclo di clock dell'esempio considerato. Senza accorgimenti al successivo ciclo il primo stadio eseguirebbe il fetch della terza istruzione, il secondo stadio eseguirebbe la decodifica della seconda istruzione mentre il terzo stadio eseguirebbe la prima istruzione come visto nell'esempio precedente.



Al quarto ciclo di clock il dato è letto dalla memoria. Ne segue che è presente nella pipe nel registro **MEM/WB** e può essere quindi anticipato nel quinto ciclo di clock quando viene eseguita la seconda istanza dell'istruzione **and**.



Al successivo ciclo la pipe può proseguire normalmente. Il quinto stadio completa la scrittura dell'istruzione **lw**. Il quarto stadio resta inattivo a causa della NOP. Il terzo stadio esegue la seconda istanza della **and** sfruttando il dato anticipato dall'unità di propagazione mentre il secondo stadio esegue la decodifica dell'istruzione **or**



L'unità di rilevamento delle criticità permette al programmatore di non preoccuparsi di verificare la correttezza dell'esecuzione. Ciò non toglie che l'introduzione delle **nop** nella pipe riduce il numero di istruzioni eseguite nell'unità di tempo dalla CPU. Un compilatore intelligente potrebbe in questi casi tentare di riordinare le istruzioni (senza ovviamente cambiare la semantica del programma) in modo da evitare le criticità e quindi i ritardi nella pipe.

Ex:

add \$14, \$15, \$16	\rightarrow	lw \$2, 100(\$3)
lw \$2, 100(\$3)	\rightarrow	add \$14, \$15, \$16
and \$12, \$2, \$5		and \$12, \$2, \$5
or \$13, \$6, \$2		or \$13, \$6, \$2

L'istruzione **add** non ha dipendenze con le istruzioni successive quindi può essere spostata in avanti di una posizione senza alterare la semantica del programma. In questo modo la bolla necessaria per risolvere la criticità sulla **lw** viene riempita con un'istruzione utile.

4) Criticità di flusso

Le criticità nel flusso di esecuzione si verificano in presenza di salti condizionati.

Si consideri il seguente esempio:

```
                beq $1,$3, JUMP
                and $12, $2, $5
                or $13, $6, $2
                and $14, $2, $2
    JUMP        lw $4, 50($7)
```

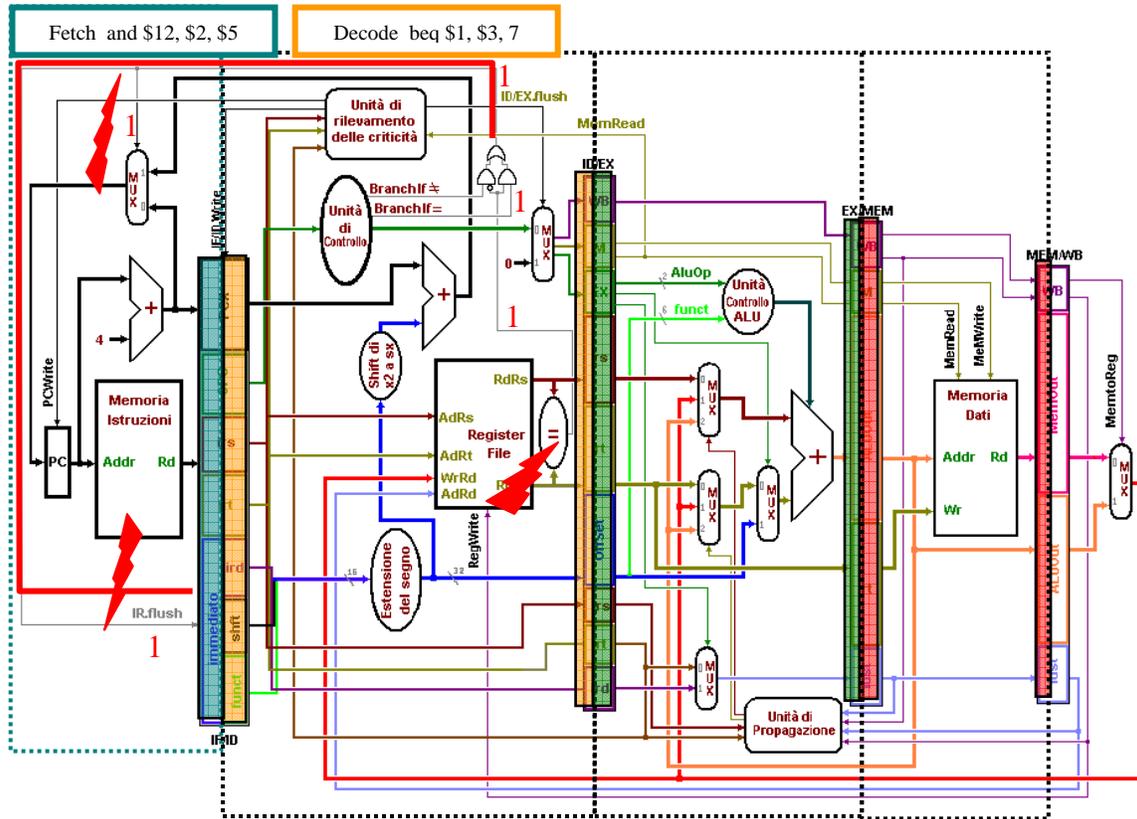
Limitando le condizioni di salto alla sola verifica di uguaglianza è possibile decidere se eseguire il salto nello stadio di decodifica; usando un circuito *ad-hoc* posto all'uscita del *Register File* è possibile decidere rapidamente se il contenuto dei due registri **rs** ed **rt** coincide o no². Un circuito così ottimizzato può quindi eseguire il salto condizionato in soli due cicli di clock.

Questo però non evita che nella pipe entri l'istruzione immediatamente seguente l'istruzione di salto, sia che il salto si verifichi sia che il salto non abbia effetto. Nel caso il salto si verifichi, questa istruzione "spuria" potrebbe alterare l'esecuzione corretta del programma *assembly*.

La soluzione più semplice a questo problema consiste nel introdurre appena dopo il salto sempre una istruzione **nop** (una *bolla*) in modo da mettere in stallo la CPU in attesa che la decisione sul salto sia presa. Una soluzione alternativa più efficiente ma più costosa in termini di hardware impiegato consiste nel lasciare entrare nella pipe l'istruzione critica ed attendere il completamento del salto per decidere se continuarne l'esecuzione o eliminarla dalla pipe perchè non appartenente al flusso di esecuzione. L'unità di controllo quindi può decidere in fase di decodifica quale nuovo indirizzo caricare nel PC e, nel caso il salto si verifichi, può sostituire l'istruzione spuria entrata nella pipe con un'istruzione di NOP. Questo comportamento fa guadagnare un ciclo di clock di effettiva elaborazione nel caso non si verifichi il salto.

² Un circuito che realizza un confronto di uguaglianza è più semplice circuitalmente da realizzare che un circuito che realizza un confronto di maggioranza per cui è necessario un sottrattore.

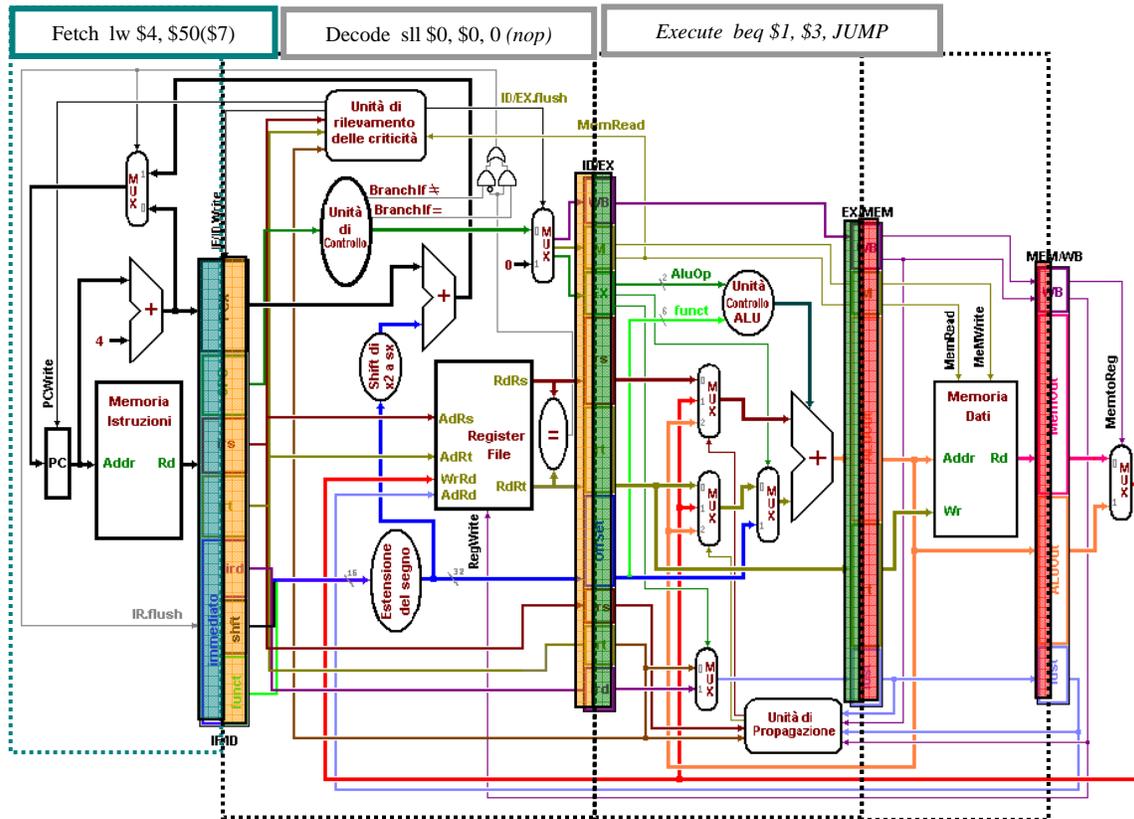
Supponiamo che al tempo di decodifica dell'istruzione **beq \$1,\$3,7** il valore di **\$1** e **\$3** sia uguale, cioè che si verifichi la condizione di salto. In questa situazione, l'istruzione semanticamente giusta da eseguire è **lw \$4,\$5(\$7)**. Il primo stadio della pipe comunque è pronto ad eseguire la fase di fetch dell'istruzione **and \$12,\$2,\$5**.



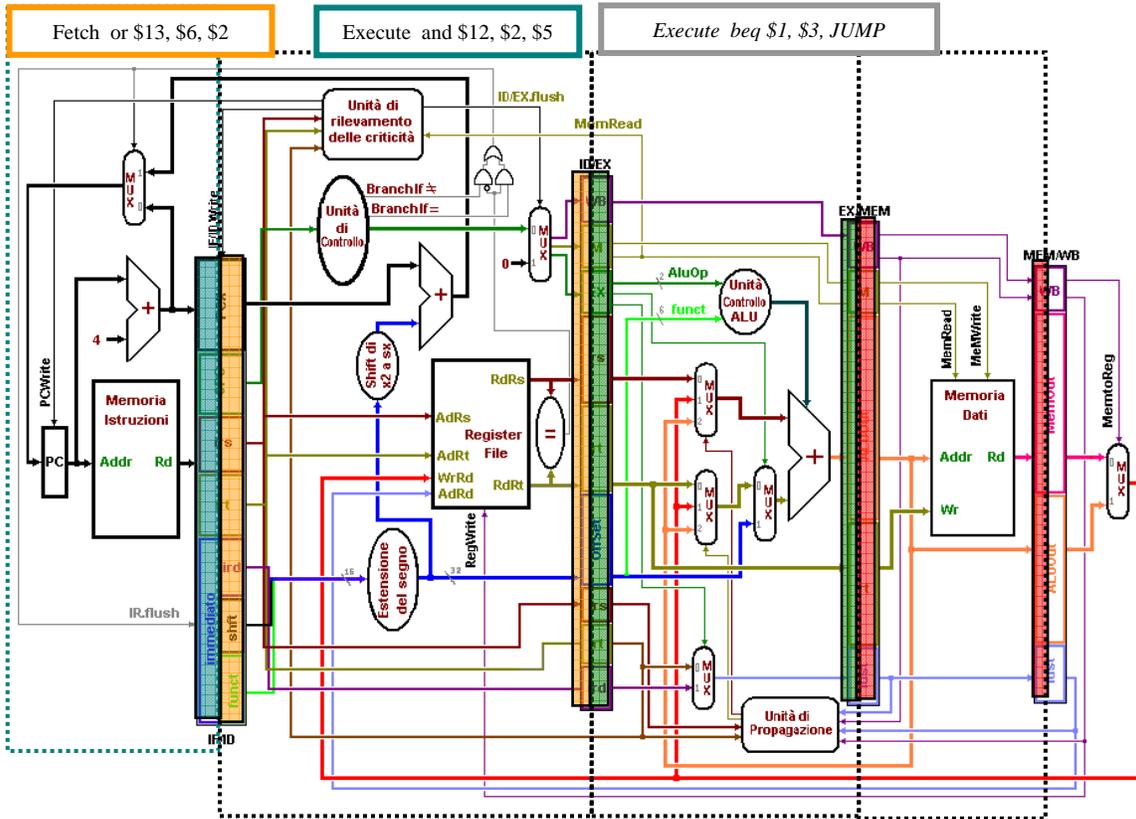
Nell'ipotesi fatta che si verifichi il salto, le linee di controllo dell'esempio:

- attraverso il segnale `if.Flush` sostituisce alla istruzione di **and** una istruzione di **nop** (in pratica si inserisce una *bolla* nella pipe).
- memorizza nel **PC** il risultato del sommatore ad-hoc presente nello stadio di decodifica che calcola l'indirizzo di destinazione del salto.

Questa è la situazione della pipe dopo che è stato eseguito il salto, al terzo ciclo di clock.



Nel caso contrario che non si verifichi la condizione di salto la pipe procede regolarmente:



5) Rappresentazione algebrica della unita di propagazione.

La condizione della prima criticità nei dati descritta (detta anche *EX hazard* o *1-type hazard*), quella che può avvenire tra un'istruzione di tipo R ed un'istruzione immediatamente seguente, può essere espressa in termini formali come segue:

```
if ( (EX/MEM.RegWrite = 1)
      and (EX/MEM.idst != 0)
      and (EX/MEM.idst != ID/EX.irs) ) ForwardA = 10
```

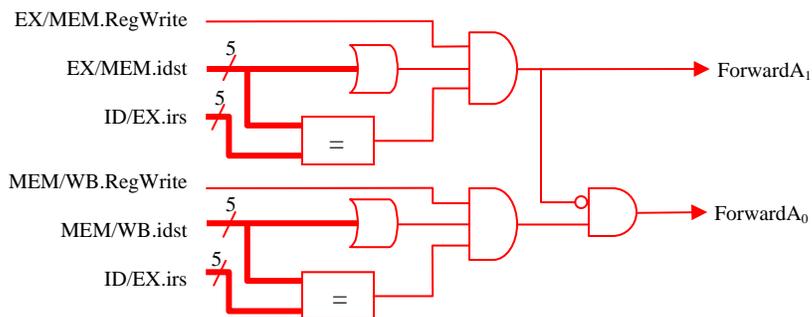
La condizione della seconda criticità nei dati descritta (detta anche *MEM hazard* o *2-type hazard*), quella che può avvenire tra un'istruzione di tipo R ed l'istruzione seguente quella successiva, può essere espressa in termini algebrici come segue:

```
if ( (MEM/WB.RegWrite = 1)
      and (MEM/WB.idst != 0)
      and (MEM/WB.idst != ID/EX.irs) ) ForwardA = 01
```

E' da notare che questa propagazione non va eseguita se si è verificata contemporaneamente la propagazione precedente. Ne segue che la formula che tiene conto di entrambe le criticità diventa:

```
if ( (EX/MEM.RegWrite = 1)
      and (EX/MEM.idst != 0)
      and (EX/MEM.idst != ID/EX.irs) ) ForwardA = 10
else if ( (MEM/WB.RegWrite = 1)
          and (MEM/WB.idst != 0)
          and (MEM/WB.idst != ID/EX.irs) ) ForwardA = 01
else ForwardA=00
```

Una possibile implementazione può essere la seguente:³



Si può definire una formula analoga per il secondo MUX considerando **irt** e **ForwardB** al posto di **irs** e **ForwardA**.

³ Il comparatore può essere implementato con XOR ibt a bit e na OR a 5 ingressi.

Fetch di lw \$4,
12(\$0)

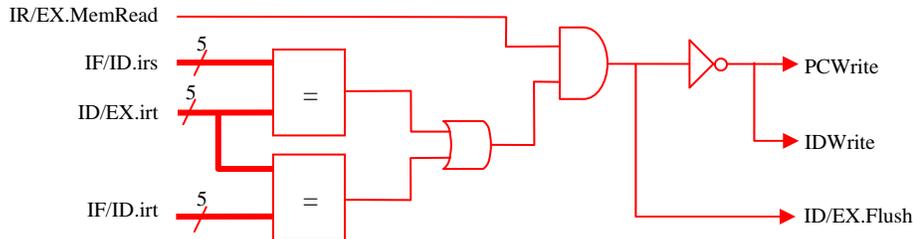
6) Rappresentazione algebrica della unità di rilevamento delle criticità.

La condizione della criticità nei dati non risolvibile descritta sopra, quella che può avvenire tra un'istruzione di tipo LW ed un'istruzione immediatamente seguente, può essere espressa in termini formali come segue:

```
if [(IR/EX.MemRead=1)
    and ((ID/Ex.irt==IF/ID.irs) or (ID/Ex.irt==IF/ID.irt)) ] {
    PCWrite=0
    IDWrite=0
    ID/Ex.flush=1
}
```

In realtà la condizione delineata è più stringente di rispetto alle reali necessità: se la lw non è seguita da una istruzione di tipo R può capitare che venga introdotta una bolla anche quando non serve (considerate una beq non eseguita la cui trama è tale che ID/EX.irt è uguale a IF/ID.irt).

Una possibile implementazione può essere la seguente;



Si può definire una formula analoga per il secondo MUX considerando **irt** e **ForwardB** al posto di **irs** e **ForwardA**.

4) Riorganizzazione del codice:

```
400:  add $t0, $t1, $t2
404:  j 80000
```

```
80000:  addi $t1, $t0, 100
80004:  addi $s5, $t0, 200
80008:  add $s6, $s7, $s7
8000C:  beq $s1, $s2, -2
80010:  add $t5, $t5, $t5
80014:  add $t6, $t6, $t6
80018:  sw $t4, 40($s5)
```

