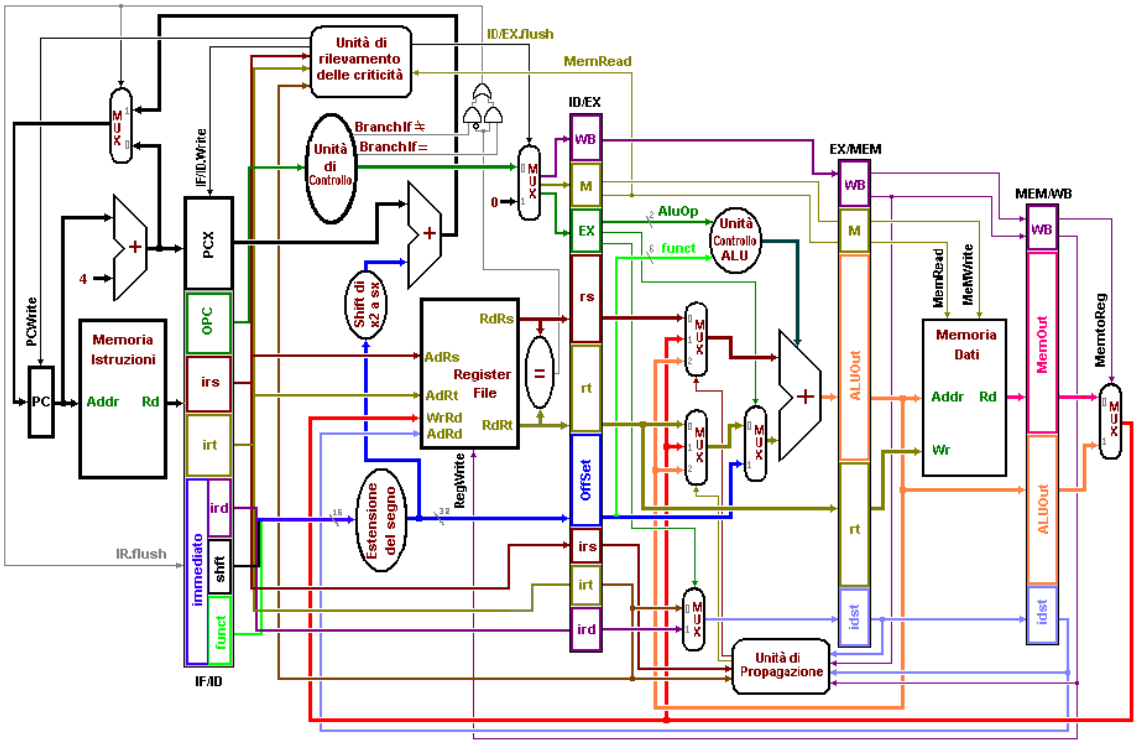
Esercitazione del 05/05/2010 - Soluzioni

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 nel *register file*, 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 lunga è la *pipe*, più alto è il numero di istruzioni teoricamente eseguibili contemporaneamente, più alta è la probabilità che insorgano criticità che possono rallentare la *pipe*.

Criticità nei dati risolvibili

Consideriamo una CPU *pipelined* come segue:



Consideriamo inoltre il seguente segmento di codice in cui sono presenti un certo numero di criticità sui dati risolvibili:

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

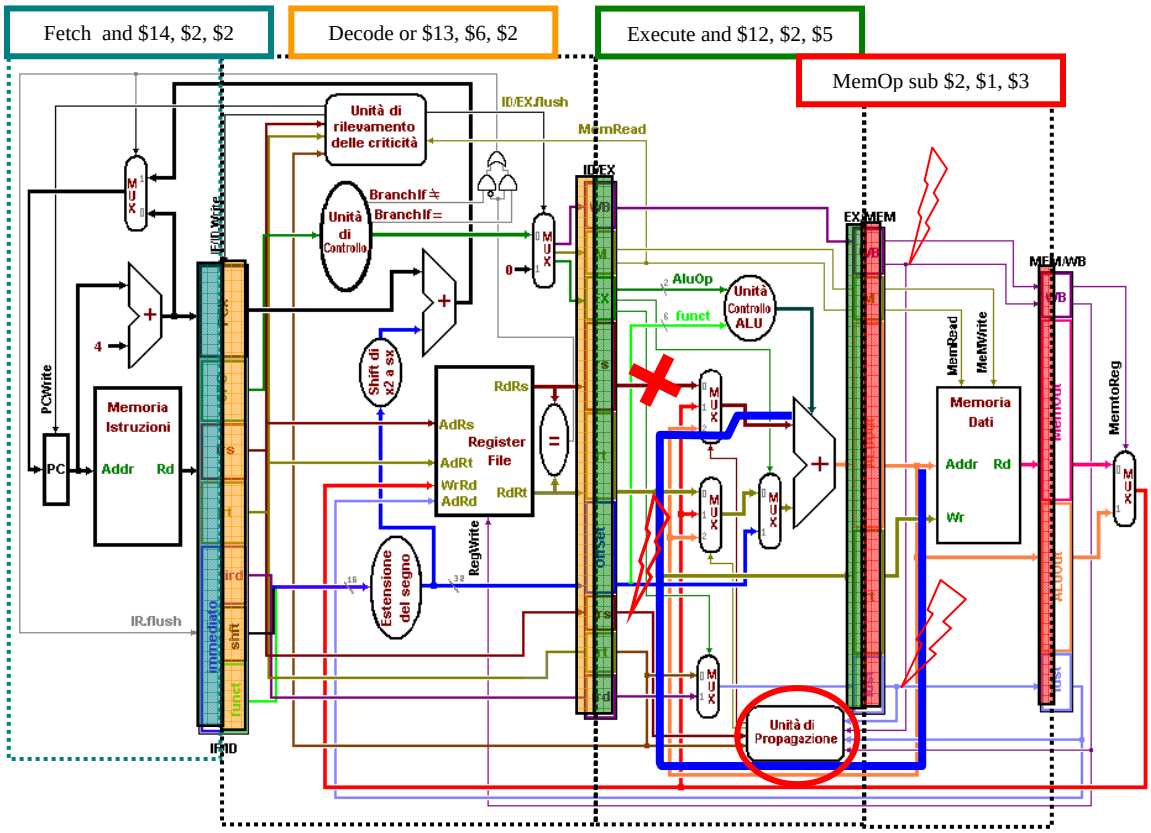
Nell'architettura *pipeline* dell'esempio l'istruzione **sub $2,$1,$3** dopo essere stata decodificata richiede 3 cicli per scrivere il risultato nel *register file*. Ne segue che, a meno di qualche accorgimento le tre istruzioni seguenti leggeranno un dato in $2 che non corrisponde a quello corretto[1]. 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 con più attenzione il flusso dei dati, 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 giunto al *register file*. 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**[2] 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.

[2] O se preferite, *anticipare in avanti (feed-forward).*

Criticità nei dati risolvibile dallo stadio MemOp

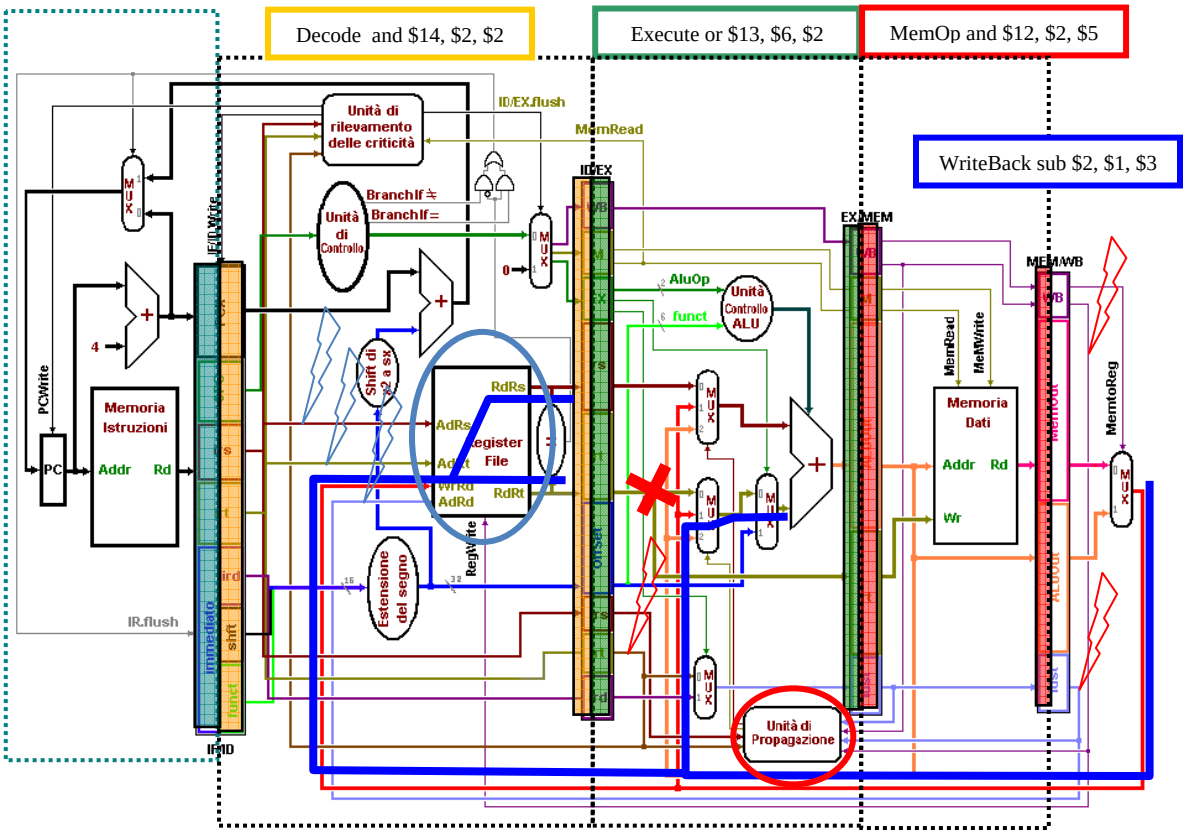
Questa è la situazione al quarto ciclo di clock dell'esempio considerato. Il primo stadio esegue la fase di *fetch* della quarta istruzione, il secondo stadio esegue la decodifica della terza istruzione, il terzo stadio esegue la seconda istruzione mentre il quarto stadio esegue l'operazione sulla memoria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



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, un valore in generale diverso da quello calcolato dall'istruzione precedente. 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.

Criticità nei dati risolvibile dallo stadio di WriteBack

Al quinto ciclo di clock dell'esempio si verifica ancora una criticità risolvibile: l'istruzione **or $13,$6,$2** sta per computare il proprio risultato usando il valore di $2 presente nel *register file* al ciclo di clock precedente e non ancora aggiornato.



Come prima il dato corretto è già presente all'interno della *pipe*, stavolta nel registro temporaneo **MEM/WB.** 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**). Confrontando il registro **rd** di destinazione e gli indici propagati lungo la *pipe*, l'unità di propagazione è in grado di anticipare il risultato dell'operazione di **sub** alla **or**, senza generare stalli.

Nello stesso ciclo di clock l'istruzione **add $14,$2 $2** sta caricando in **ID/EX** il valore del registro **$2**. Progettando opportunamente il *register file* è possibile fare in modo che un dato scritto possa essere propagato all'occorrenza in uscita entro il ciclo di clock. Ad esempio, è possibile realizzare le scritture nel primo semi-ciclo e le letture nel secondo semi-ciclo scritture. Un'altra possibilità è confrontare l'indice del registro che si sta scrivendo con i due registri richiesti in lettura e, nel caso di identità, anticipare in uscita il valore in ingresso in maniera simile a come agisce l'unità di propagazione.

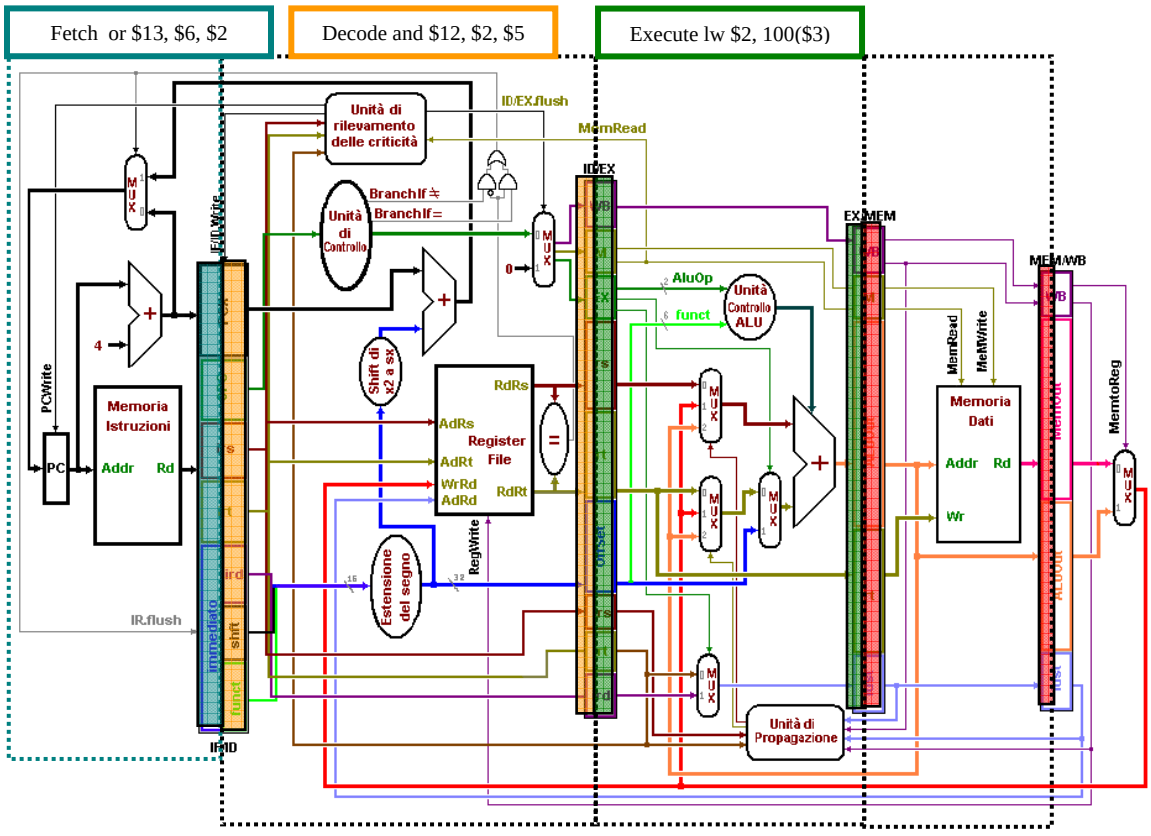
Criticità nei dati non risolvibili

A volte non è possibile anticipare il risultato di un istruzione perché 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. Se lasciamo procedere la *pipe* senza modifiche, al momento dell'esecuzione dell'istruzione **and** il dato della cella **100($3)** non è ancora stato estratto dalla memoria e quindi non è possibile anticiparne il valore attraverso un opportuno *data-path* (criticità non risolvibile). Occorre quindi fermare la *pipe* almeno per un ciclo in attesa che il dato divenga disponibile per l'anticipazione nello stadio di *WriteBack*.
Questa condizione di criticità può essere rilevata un ciclo prima che si verifichi il problema nello stadio di *Execute*, quando l'istruzione **Add** è ancora in fase di *Decode*. Se l'istruzione che è presente nello stadio di *Execute* è una **Load** che ha criticità con l'istruzione che segue, allora si può risolvere la situazione bloccando l'avanzamento della *pipe* negli stadi *Fetch* e *Decode* ed inserendo una **NOP** (detta *bubble*) all'interno della *pipe* al posto della **Add**.

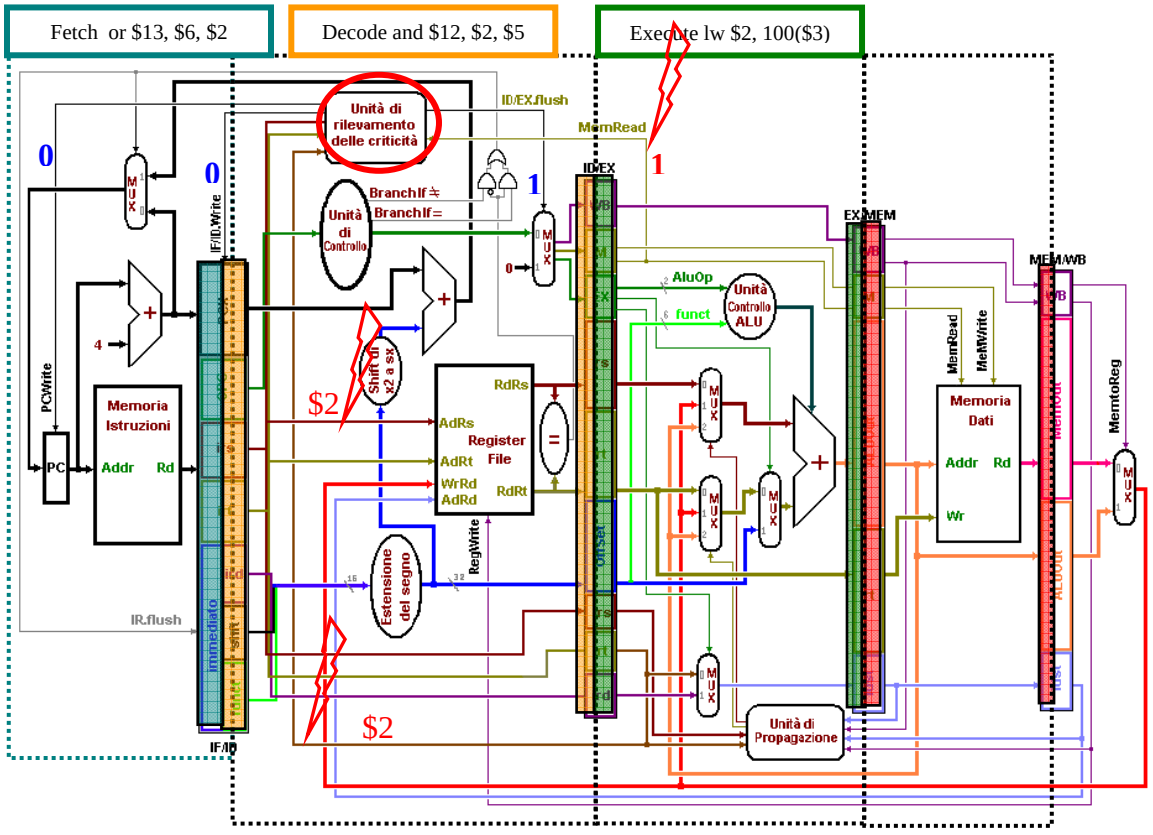
Questa è la situazione al terzo ciclo di clock dell'esempio considerato. Senza accorgimenti al successivo ciclo il primo stadio eseguirebbe il fetch della terza istruzione, il secondo stadio eseguirebbe la decodifica della seconda istruzione mentre il terzo stadio eseguirebbe la prima istruzione come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



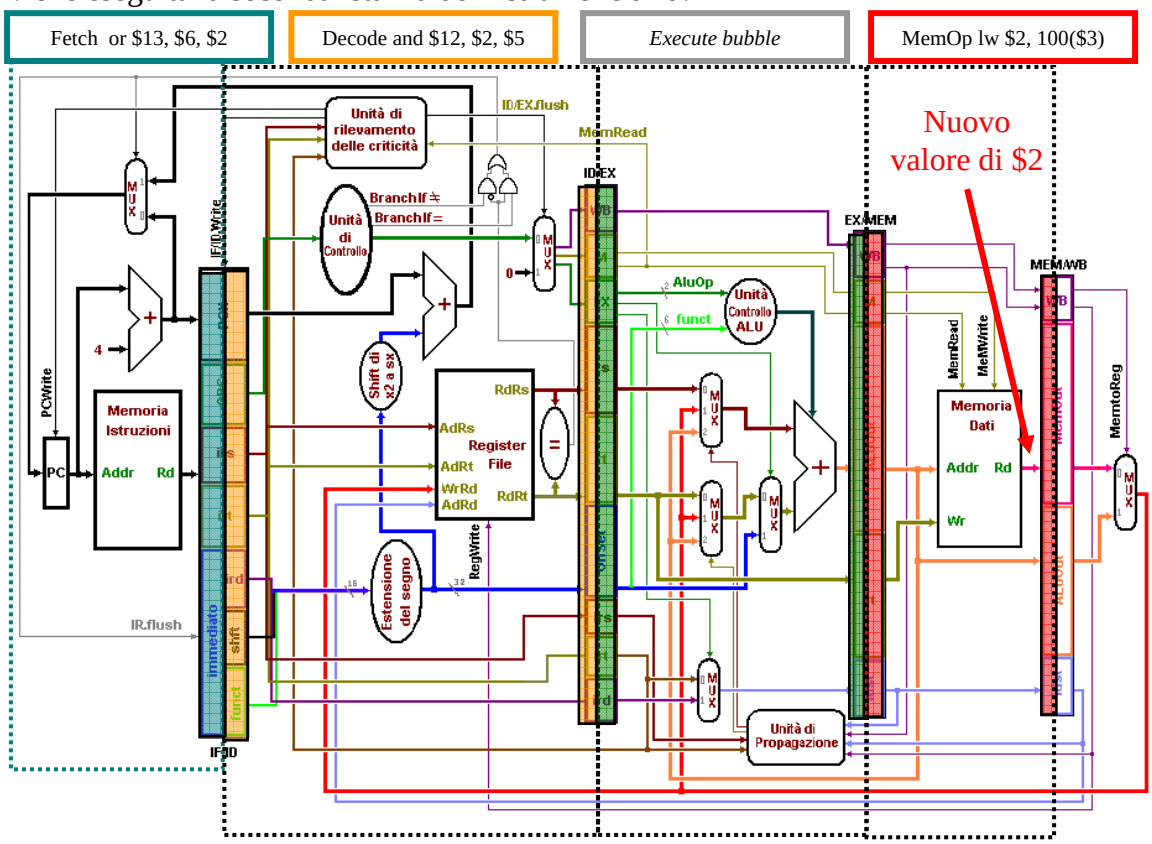
Come abbiamo descritto sopra, un opportuno circuito (**unità di rilevamento delle criticità**) è in grado di rilevare la criticità futura dello stadio di *Execute* esaminando l'indirizzo di destinazione della **lw** presente in **ID/EX,** il relativo segnale di scrittura **WB** presente anch'esso in **ID/EX,** il registro richiesto in lettura dalla istruzione **and** presente nello stadio di decodifica. Una volta rilevata la criticità è possibile inserire nella *pipe* una una *bubble* in modo da ritardare l'esecuzione della **and** di un ciclo di clock. Questo consente di estrarre il dato richiesto dalla memoria e di renderlo disponibile all'unità di propagazione.
La **NOP** viene realizzata fermando per un ciclo di clock l'avanzamento della pipe ed invalidando il risultato dell'istruzione and già entrata nella pipe.
L'unita di rilevamento una volta rilevata la criticità:

- inibisce l'aggiornamento del **PC** al nuovo valore. Questo provoca la rilettura dell'istruzione attualmente nella fase di *Fetch*, nell'esempio la **or**.
- inibisce l'aggiornamento del registro **IF/ID**. Questo permette di rilanciare la decodifica dell'istruzione che ha subito la criticità con un ritardo di un ciclo, nell'esempio la **and**.
- inibisce gli effetti dell'istruzione che ha subito la criticità azzerando i segnali di controllo nel registro **ID/EX.** L'istruzione con i dati errati percorrerà comunque la pipe senza però generare effetti sulla memoria dati o sui registri.

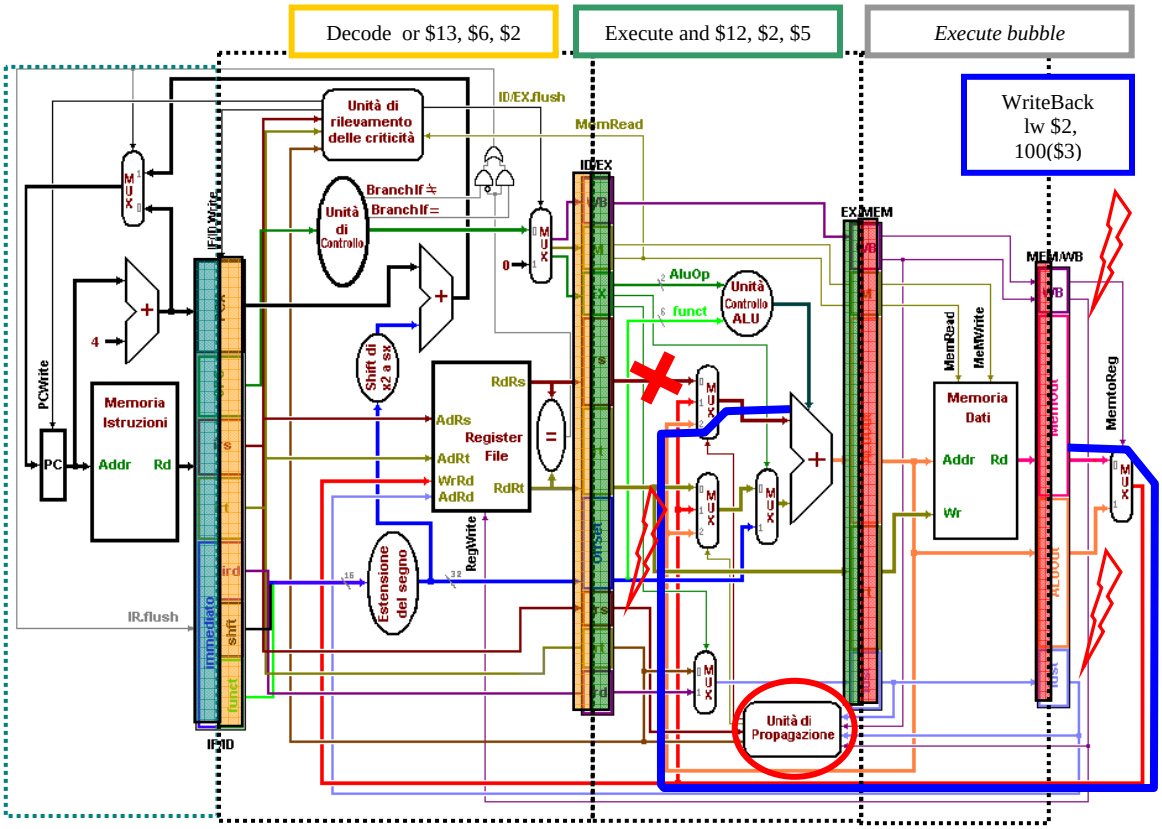
Di seguito è riportato lo stato della pipe alla fine del terzo ciclo di clock



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 può 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	**lw $2, 100($3)**
lw $2, 100($3)	**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.*

Nel caso la **lw** sia seguita da un'istruzione di salto condizionato dipendente dalla **lw**, occorre mettere in stallo la *pipe* per due cicli in attesa che il dato sia letto dalla memoria.

```
        lw $4, 50($7)
        beq $1,$4, JUMP
        ….
```

Criticità di flusso

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

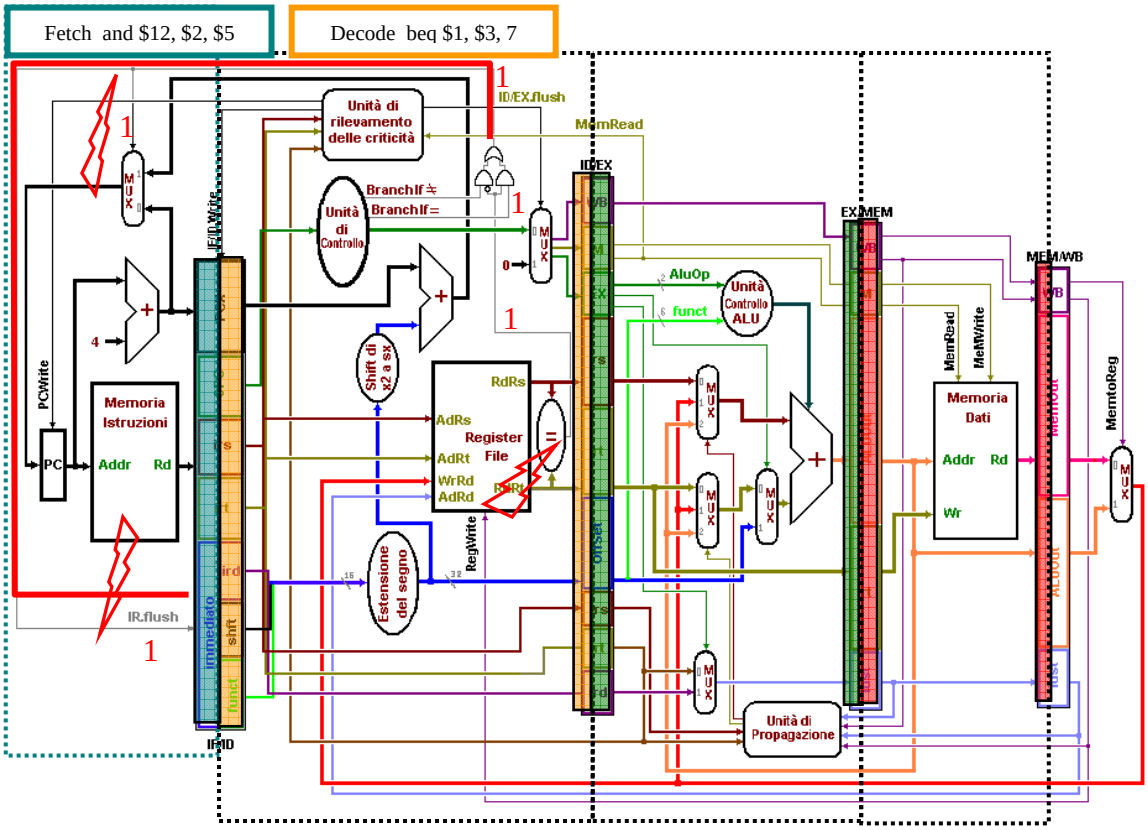
Si consideri il seguente esempio:

```
        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 no[3]. 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. Se si verifica il salto, questa istruzione "spuria" può alterare l'esecuzione corretta del programma *assembly*..
La soluzione più semplice a questo problema consiste nel introdurre appena dopo il salto sempre un'istruzione **NOP** (*bubble*) 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* perché 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.

[3] Un circuito che realizza un confronto di uguaglianza è più semplice da realizzare rispetto ad 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 $4,50($7).** Il primo stadio della *pipe* comunque è impegnato ad eseguire la fase di *fetch* dell'istruzione **and $12,$2,$5.**

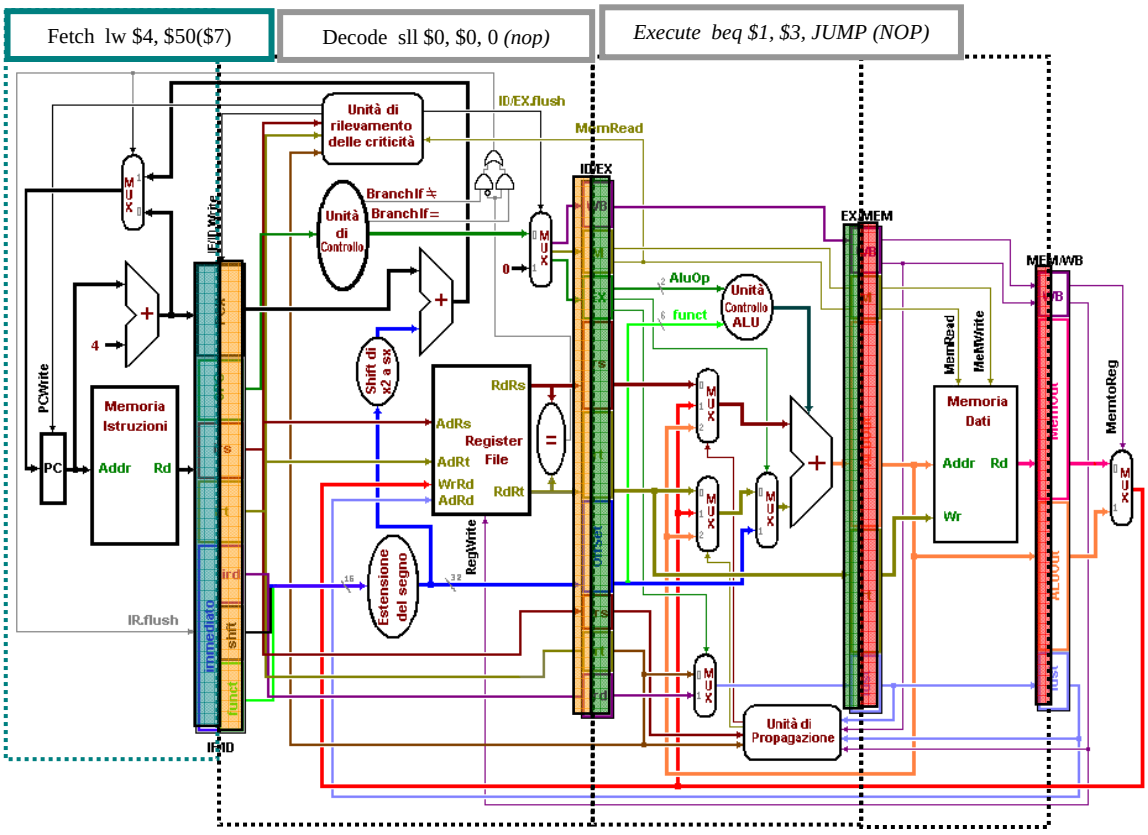


Massimo Marchi – marchi@dsi.unimi.it

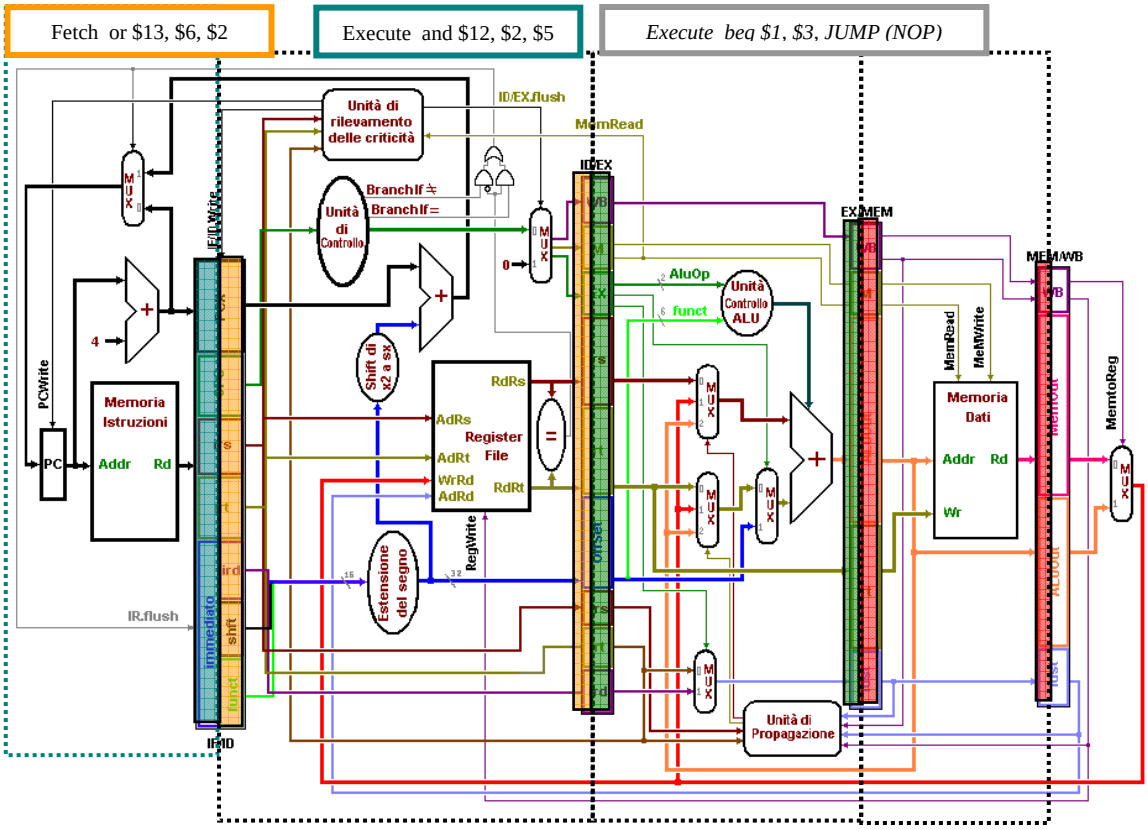
Nell'ipotesi fatta che si verifichi il salto, le linee di controllo dell'esempio:

- attraverso il segnale IR.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.



Se al contrario non si verifica la condizione di salto, allora la *pipe* procede regolarmente:



Rappresentazione algebrica dell'unità 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 (*ForwardA* e *ForwardB* rappresentano rispettivamente i segnali di controllo dei MUX di anticipazione per *rs* e *rt*):

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

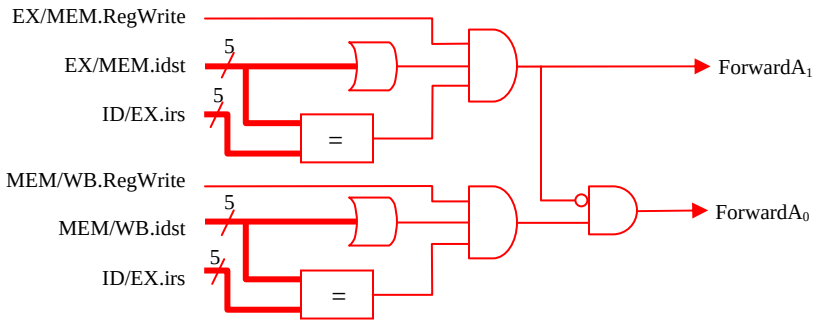
La condizione della seconda criticità nei dati descritta (detta anche *MEM hazard* o *2-type hazard*), quella che può avvenire tra un'istruzione di tipo R ed l'istruzione seguente quella successiva, può essere espressa in termini algebrici come segue:

```
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:[4]



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

[4] Il comparatore può essere implementato con XOR bit a bit e una OR a 5 ingressi.

Rappresentazione algebrica della unita 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 [(ID/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
}
```

Una possibile implementazione può essere la seguente: