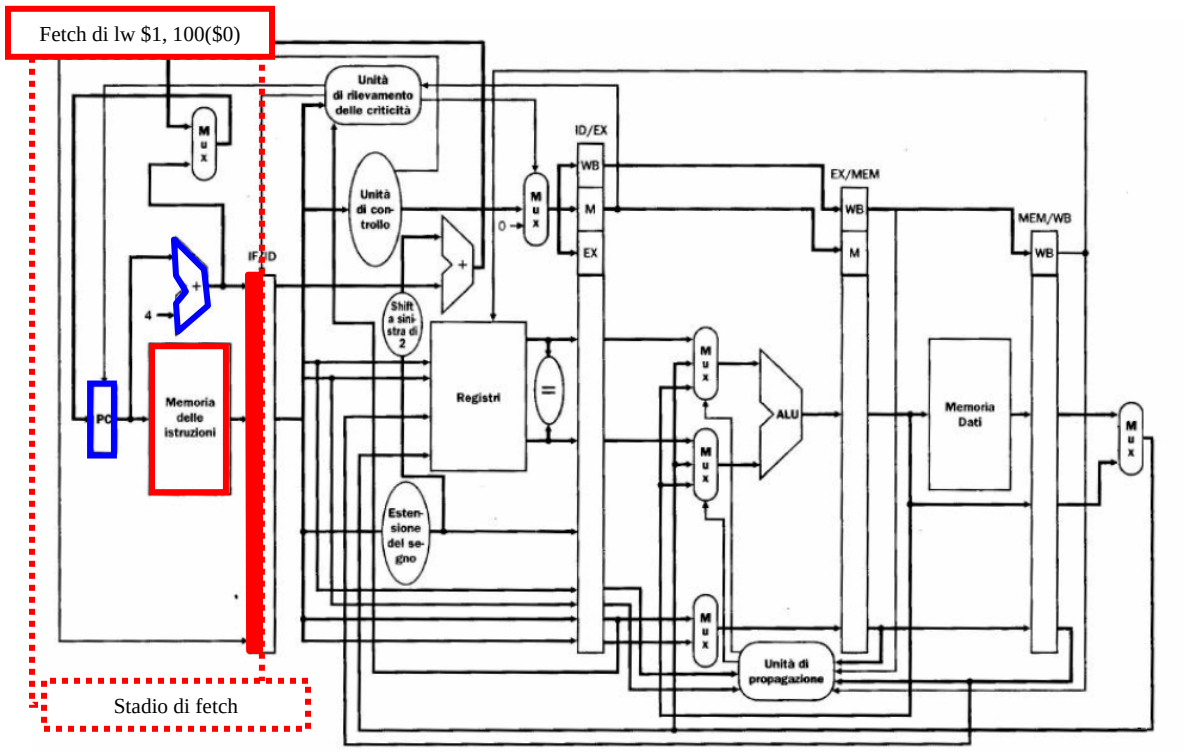
Esercitazione del 12/05/2005 - Soluzioni

Una CPU a ciclo singolo come pure una CPU multi ciclo eseguono una sola istruzione alla volta. Durante l'esecuzione poi, alcuni stadi della CPU rimangono inutilizzate in attesa che dagli stadi precedenti arrivino i dati da elaborare. In una CPU pipeline ogni stadio è separato dal successivo tramite un registro temporaneo. Ogni stadio, ad ogni ciclo di clock, legge il registro di separazione con lo stato precedente, elabora i dati e scrive i risultati nel registro di separazione dallo stato successivo. Al successivo ciclo di clock ogni stadio potrà occuparsi dell'istruzione che segue senza attendere la fine di quella corrente. Se una CPU contiene n stadi potrà, in linea teorica, mantenere in esecuzione contemporaneamente n istruzioni ognuna ad un diverso stadio dell'esecuzione.

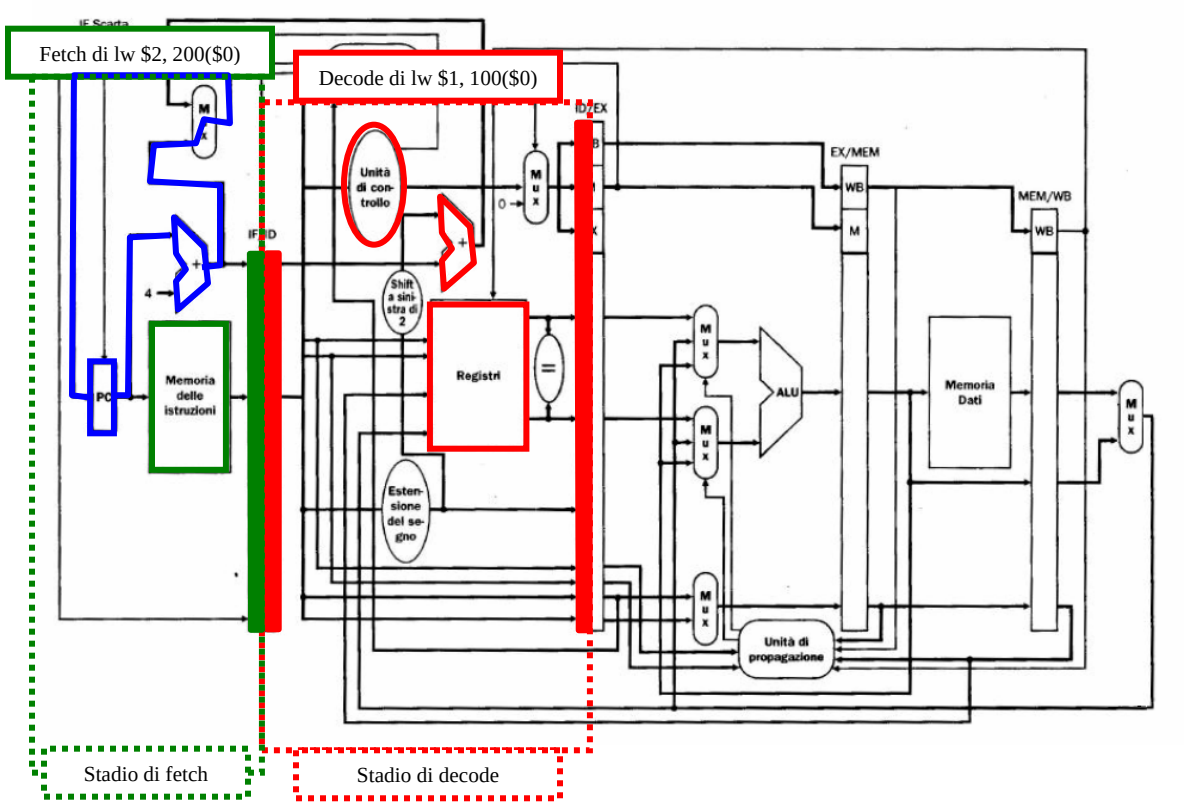
Consideriamo la sequenza di istruzioni:

lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
lw $4, 400($0)

Al primo ciclo di clock viene caricata la prima istruzione nel registro **IF/ID**. IlPC è incrementato di 4.



Al secondo ciclo di clock il primo stadio esegue la fase di fetch della seconda istruzione mentre il secondo stadio esegue la decodifica della prima istruzione. Il **PC** come prima è incrementato di 4. L'operazione di fetch (scrittura in **IF/ID**) non interferisce con l'operazione di decodifica (lettura da **IF/ID**) poiché i registri sono realizzati con architettura Master/Slave.

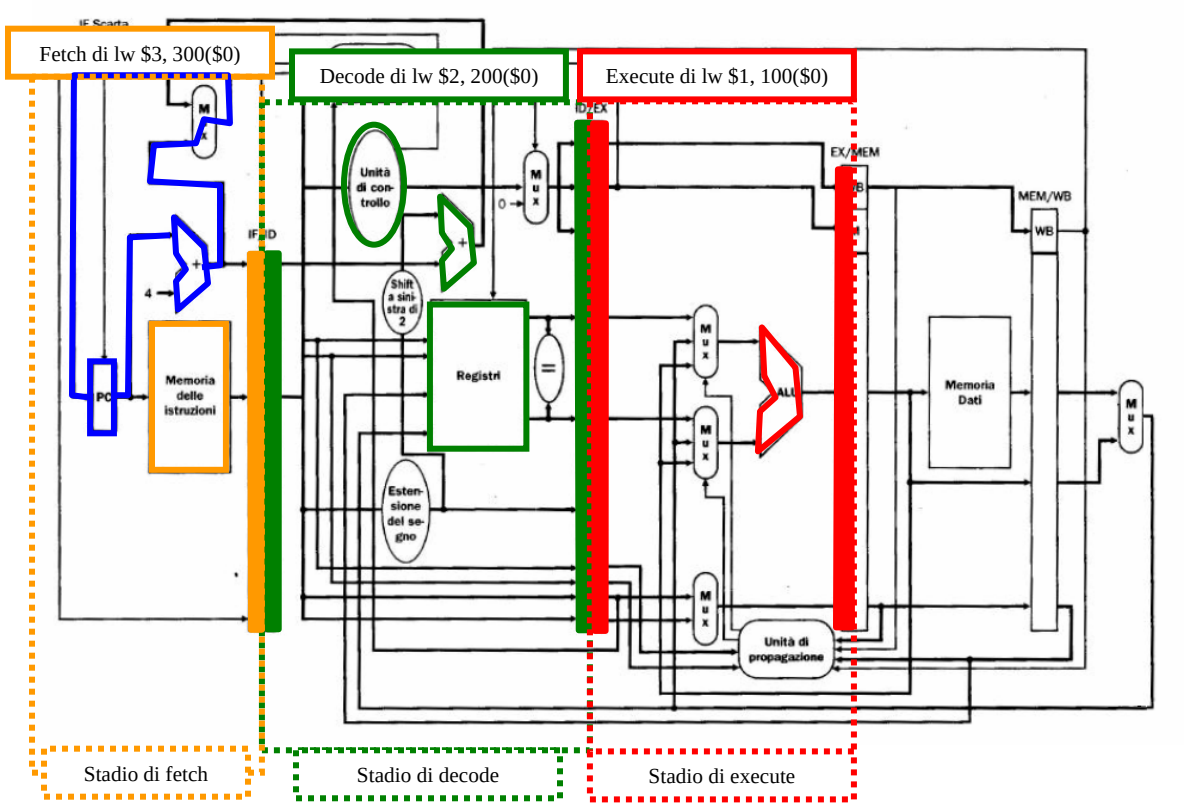


Nello stadio di decodifica vengono eseguite tutte le operazioni che si possono eseguire senza sapere ancora di che tipo di istruzione si tratta, come nel caso della CPU mono/multi ciclo. Tra queste la lettura preventiva dei registri, il calcolo degli indirizzi di salto condizionato ed incondizionato. Tutti i dati letti vengono passati allo stadio successivo tramite il registro ID/EX. Al prossimo ciclo l'unità di controllo dovrà decodificare la seconda istruzione. Ne segue che i segnali di controllo generati ma non ancora utilizzati saranno persi. Ad esempio, l'istruzione lw prevede al quinto ciclo di clock che il dato letto da memoria venga scritto nel registro indicato dal campo **rt.** Questo valore però non potrà essere recuperato da **IF/IR** in quanto al quinto ciclo di clock conterrà il codice operativo della quarta istruzione seguente. Analogamente il segnale di abilitazione alla scrittura nel register file andrà applicato al quinto ciclo di clock. E' necessario quindi far percorrere la pipeline insieme ai dati temporanei anche i segnali di controllo usati negli stadi seguenti. Vengono aggiunti nei registri temporanei i seguenti campi di controllo:

- **WB** viene usato per trasportare il segnale di scrittura nel register file
- **M** controlla le operazioni sulla memoria dati.
- **EX** controlla le operazioni eseguite dalla **ALU**

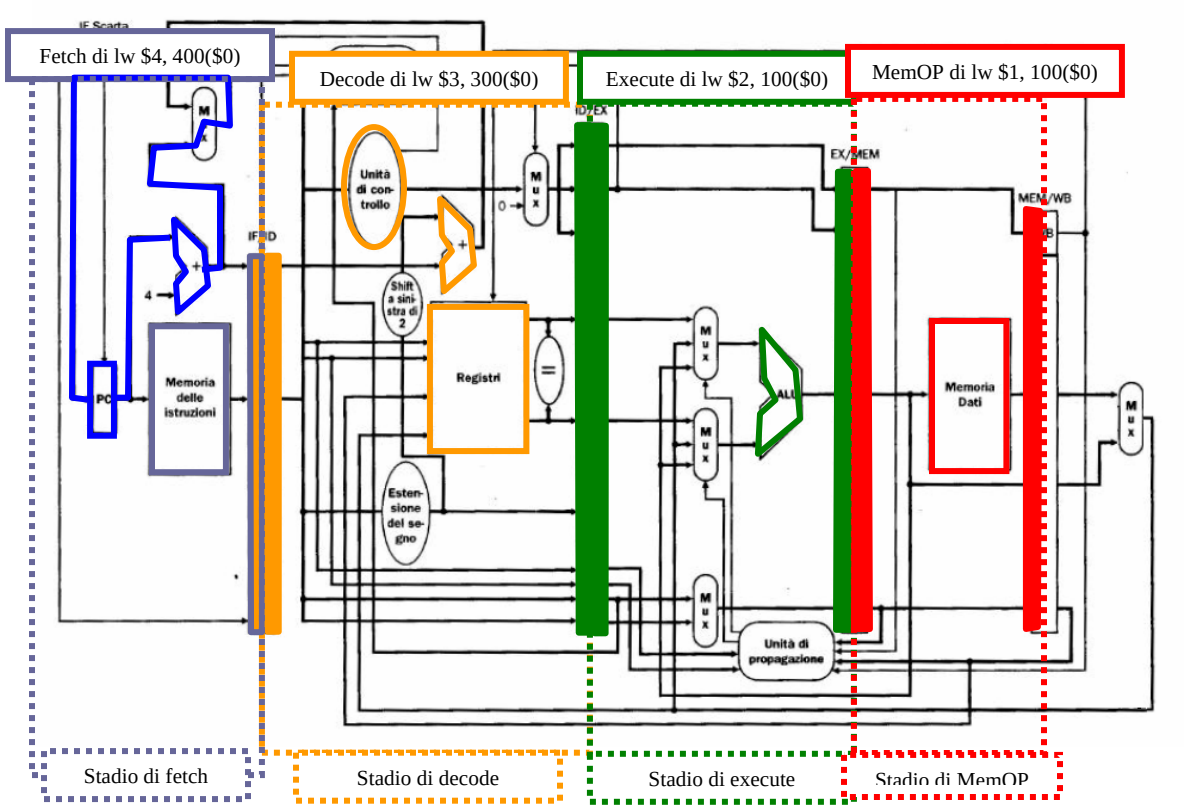
In maniera analoga andranno trasferiti anche i segnali di controllo per i multiplexer presenti nei vari stadi in modo da realizzare i giusti data path.

Al terzo ciclo di clock il primo stadio esegue la fase di fetch della terza istruzione dell'esempio, il secondo stadio esegue la decodifica della seconda istruzione mentre il terzo stadio esegue la prima istruzione. Il **PC** come prima è incrementato di 4.



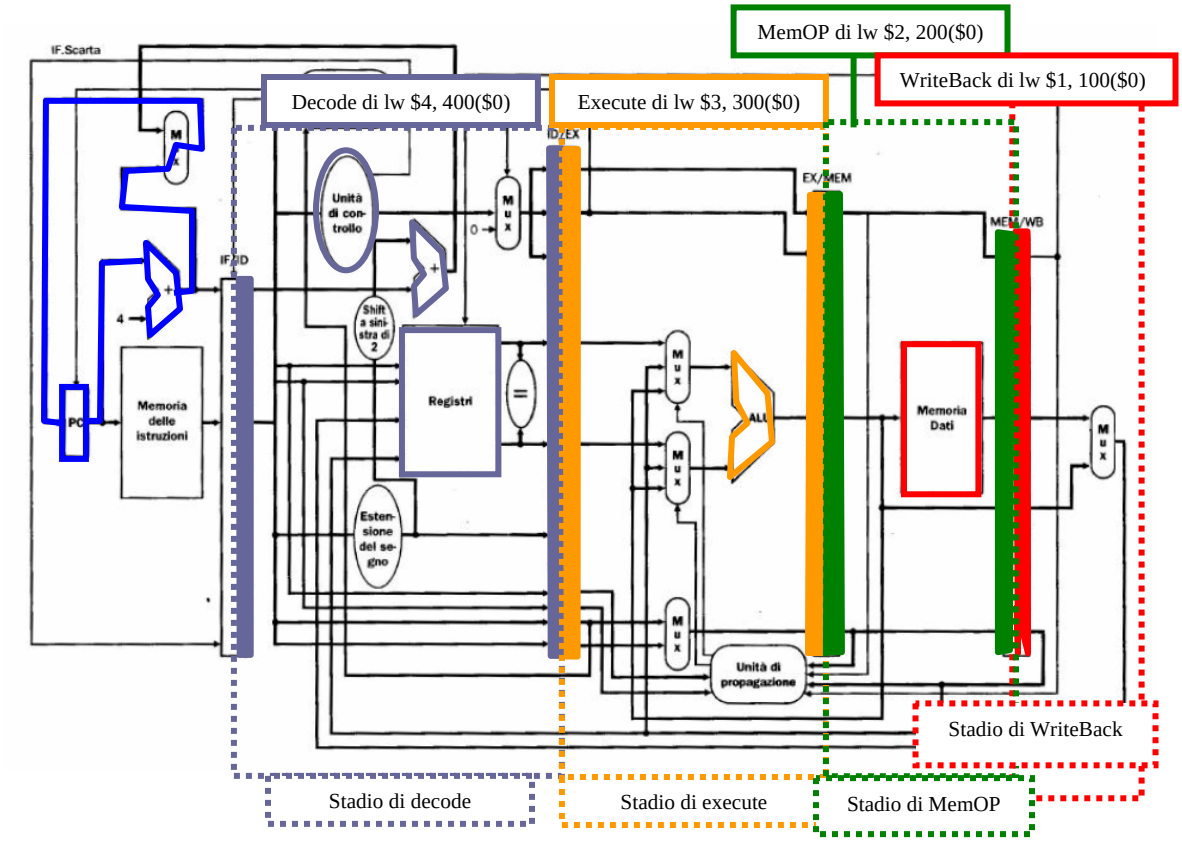
Nel registro **EX/MEM** il terzo stadio memorizza oltre all'indirizzo **100($0)** calcolato dalla **ALU** anche l'indice del registro **rt** (**$1**) passatogli dallo stadio precedente, i segnali di controllo **WB** ed **M** generati nel secondo ciclo dalla **UC**.
Nel registro **ID/EX** il secondo stadio memorizza il contenuto dei registri **rs**, **rt** ed **rd** (rd non verrà usato) estratti dal register file relativamente alla seconda istruzione, nonché i segnali di controllo opportuni per eseguirla.
Nel registro **IR/ID** il primo stadio memorizza il codice operativo della terza istruzione.

Al quarto ciclo di clock 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 seconde istruzione mentre il terzo stadio esegue le operazione sulla memoria dati relative alla prima istruzione. Il **PC** come prima è incrementato di 4.



Nel registro **MEM/WB** il quarto stadio memorizza oltre al dato contenuto nella cella di memoria dell'indirizzo **100($0)** anche l'indice del registro **rd** passatogli dallo stadio precedente ed il segnale di controllo **WB**.
Nel registro **EX/EM** il terzo stadio memorizza l'indirizzo **200($0)** calcolato dalla **ALU**, l'indice del registro **rt** (**$2**) passatogli dallo stadio precedente, i segnali di controllo **WB** ed **M** generati nel terzo ciclo dalla **UC**.
Nel registro **ID/EX** il secondo stadio memorizza il contenuto dei registri **rs**, **rt** ed **rd** (rd non verrà usato) estratti dal register file relativamente alla terza istruzione, nonché i segnali di controllo opportuni per eseguirla.
Nel registro **IR/ID** il primo stadio memorizza il codice operativo della quarta istruzione.

Al quinto ciclo di clock è possibile terminare la prima istruzione; il quinto stadio scrive il dato letto dalla memoria nel registrer file. Questa operazione, usando un'register file che consente letture e scritture contenporanee, può avvenire in contemporanea alla operazione di decodifica/lettura dei registri relativa alla quarta istruzione. Nello stesso ciclo il primo stadio esegue la fase di fetch dell'istruzione seguente alla quarta istruzione (non riportata), il terzo stadio esegue la terza istruzione mentre il quarto stadio esegue le operazione sulla memoria dati relative alla seconda istruzione. Il **PC** come prima è incrementato di 4.



Sfruttando l'indirizzo **rd** trasportato dalla pipe relativo alla prima istruzione il quinto stadio *scrive indietro* nel register file il contenuto della memoria letto nello stadio precedente.
Nel registro **MEM/WB** il quarto stadio memorizza il dato contenuto nella cella di memoria dell'indirizzo **200($0)** ed anche l'indice del registro **rd** passatogli dallo stadio precedente ed il segnale di controllo **WB**.
Nel registro **EX/EM** il terzo stadio memorizza l'indirizzo **300($0)** calcolato dalla **ALU**, l'indice del registro **rt** (**$3**) passatogli dallo stadio precedente, i segnali di controllo **WB** ed **M** generati nel quarto ciclo dalla **UC** relativi alla terza istruzione.
Nel registro **ID/EX** il secondo stadio memorizza il contenuto dei registri **rs**, **rt** ed **rd** (rd non verrà usato) estratti dal register file relativi alla quarta istruzione, nonché i segnali di controllo ottenuti dalla decodifica dell'istruzione..

L'elaborazione continua in maniera analoga; nel successivo ciclo di clock termina l'esecuzione della seconda istruzione, e cosi via. Il risultato globale è l'esecuzione di un

istruzione per ogni ciclo di clock. Ogni istruzione richiederà esattamente 5 cicli di clock per essere eseguita totalmente ma grazie al parallelismo dei 5 stadi si potranno eseguire 5 istruzioni contemporaneamente. Tutto funziona in questo modo finche non insorgono delle criticità nel flusso dei dati o nel flusso di esecuzione. Può infatti capitare che un'istruzione richieda un dato che è stato elaborato da un'istruzione precedente ma che non è ancora disponibile, o perchè non è stato ancora computato (criticità nei dati non risolvibile) o perché è ancora in viaggio all'interno della pipe (criticità nei dati non 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 alto l'insorgere di criticità che fanno rallentare la pipe.

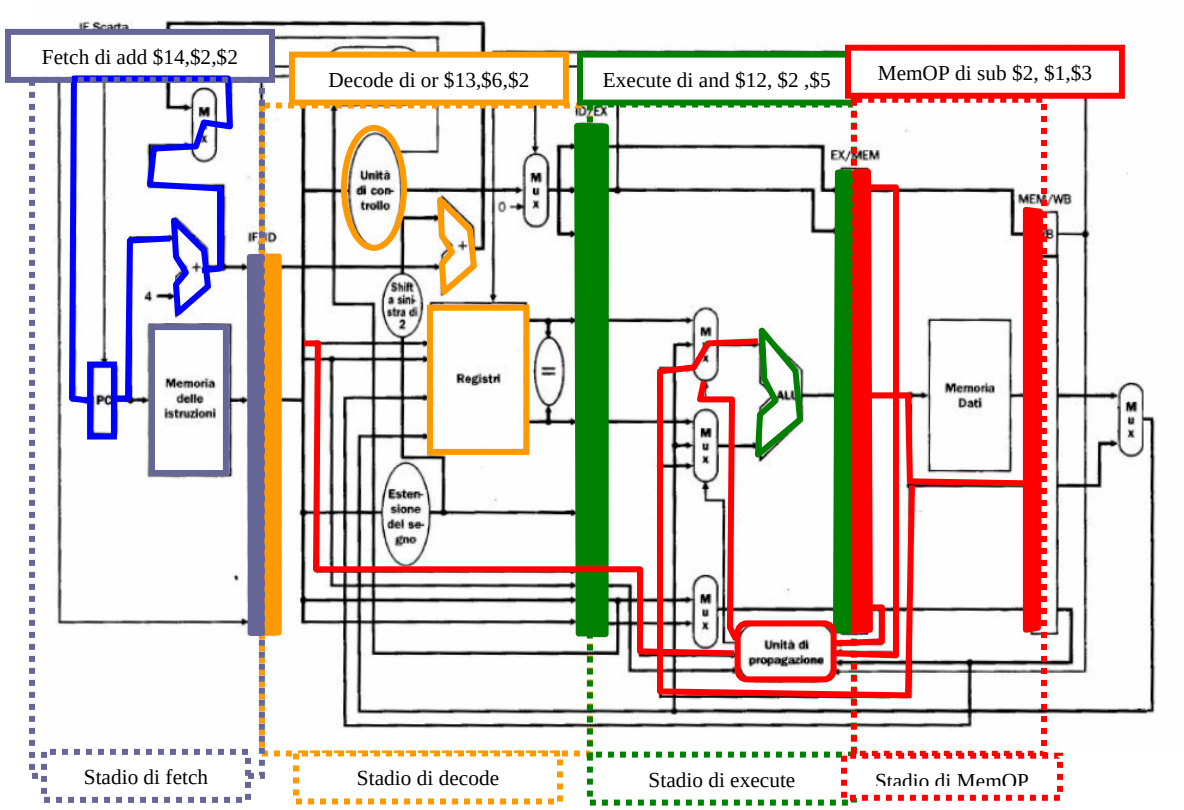
Consideriamo un caso di criticità sui dati risolvibile:

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 richiede 5 cicli per scrivere il risultato nel registro $2. 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 il flusso dei dati all'interno della CPU comunque si può notare che il valore corretto di $2, al momento di eseguire la seconda istruzione, è già presente all'interno della pipe anche se non è ancora stato scritto indiettro al registro $2. Si può quindi pensare anticipare la consegna del dato all'istruzione che segue costruendo un opportuno data path. In altre parole, nel caso che si verifichi questa criticità risolvibile, si può pensare di **propagare** il risultato all'indietro in anticipo.

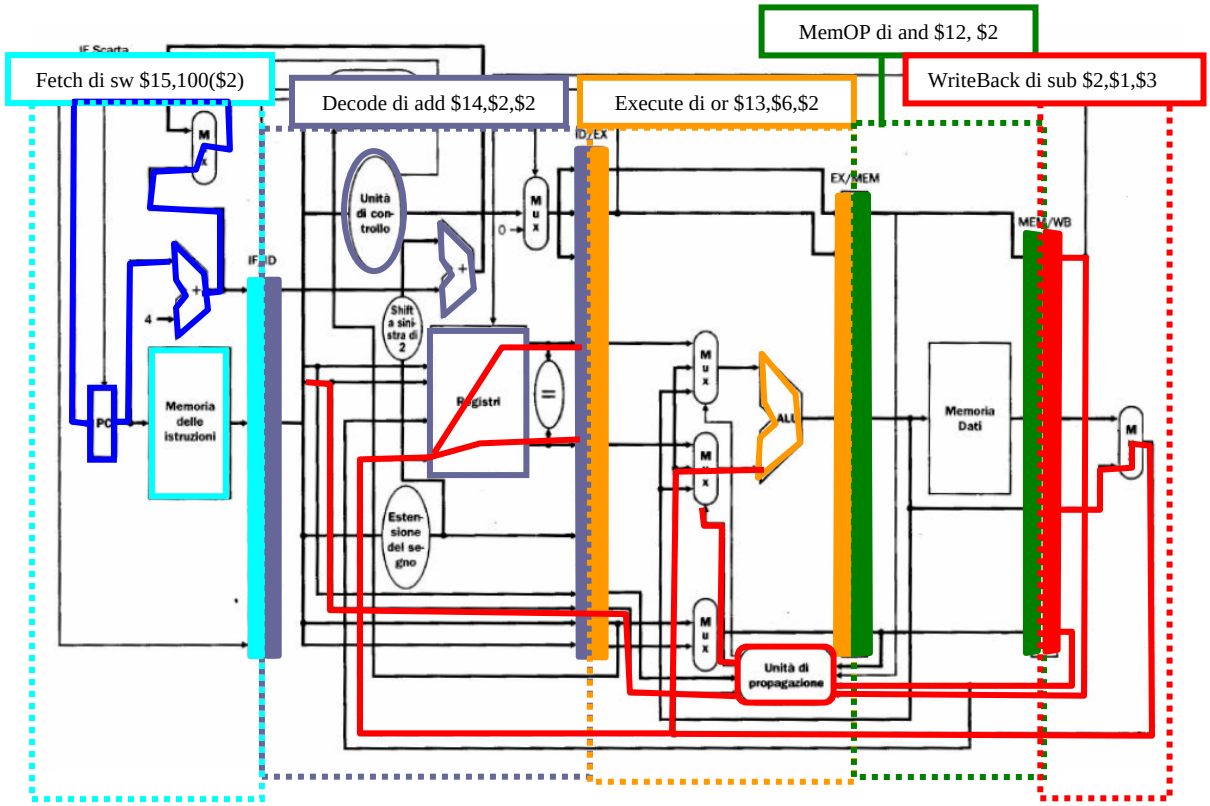
[1] Questo vale almeno secondo la semantica normalmente adottata per i programmi imperativi. In un programma ci si aspetta che ogni istruzione abbia effetto sulle istruzioni che seguono se esistono delle dipendenze. Nulla vieta comunque cambiare la semantica e lasciare al programmatore l'onere di gestire 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 seconde istruzione mentre il terzo stadio esegue le operazione 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** userebbe il dato **$2** estratto dal register file 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.

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 e non ancora aggiornato.

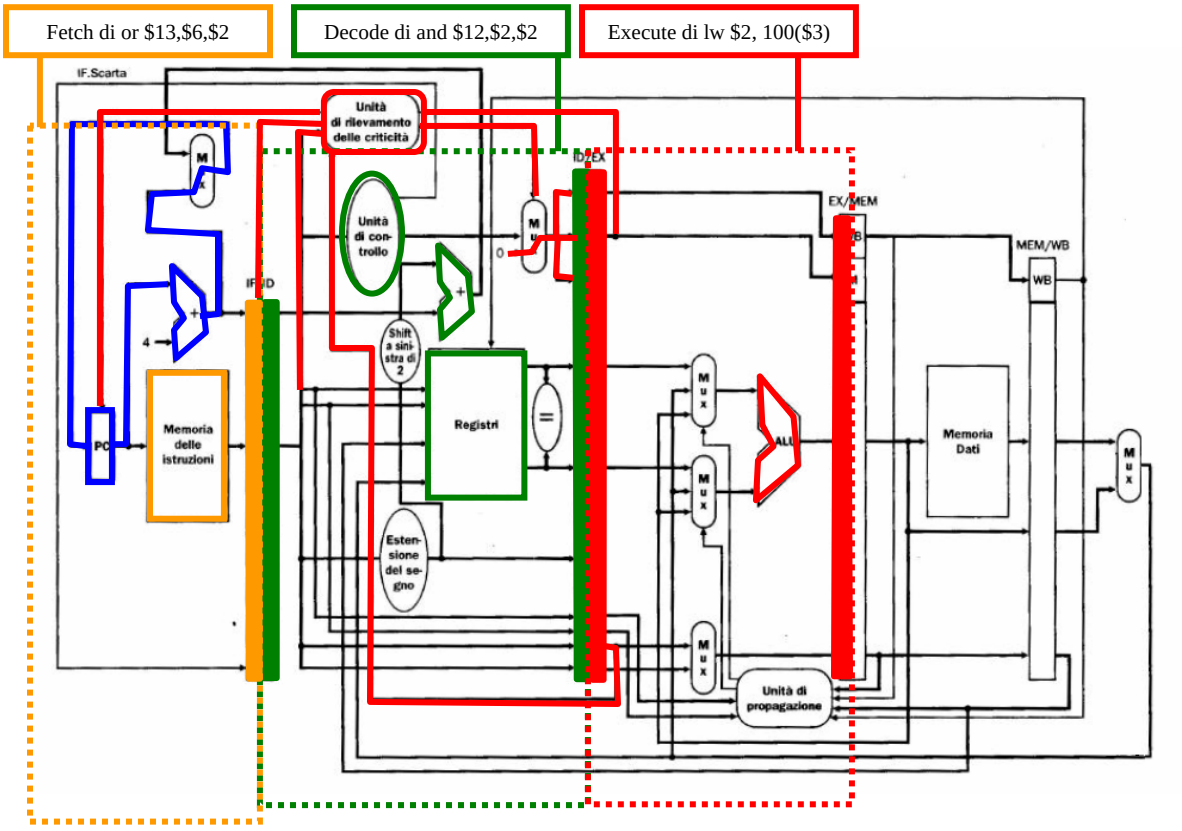


Come prima Il dato corretto è già presente all'interno della pipe, stavolta nel registro temporaneo **MEM/WB.** Nello stesso registro è anche presente l'indice del registro di destinazione dell'istruzione di **sub** (**$2**) nonché l'informazione che si tratta di una scrittura (**WB**). Confrontando 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 rallentare la pipe.
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 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'unitaà di propagazione.

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

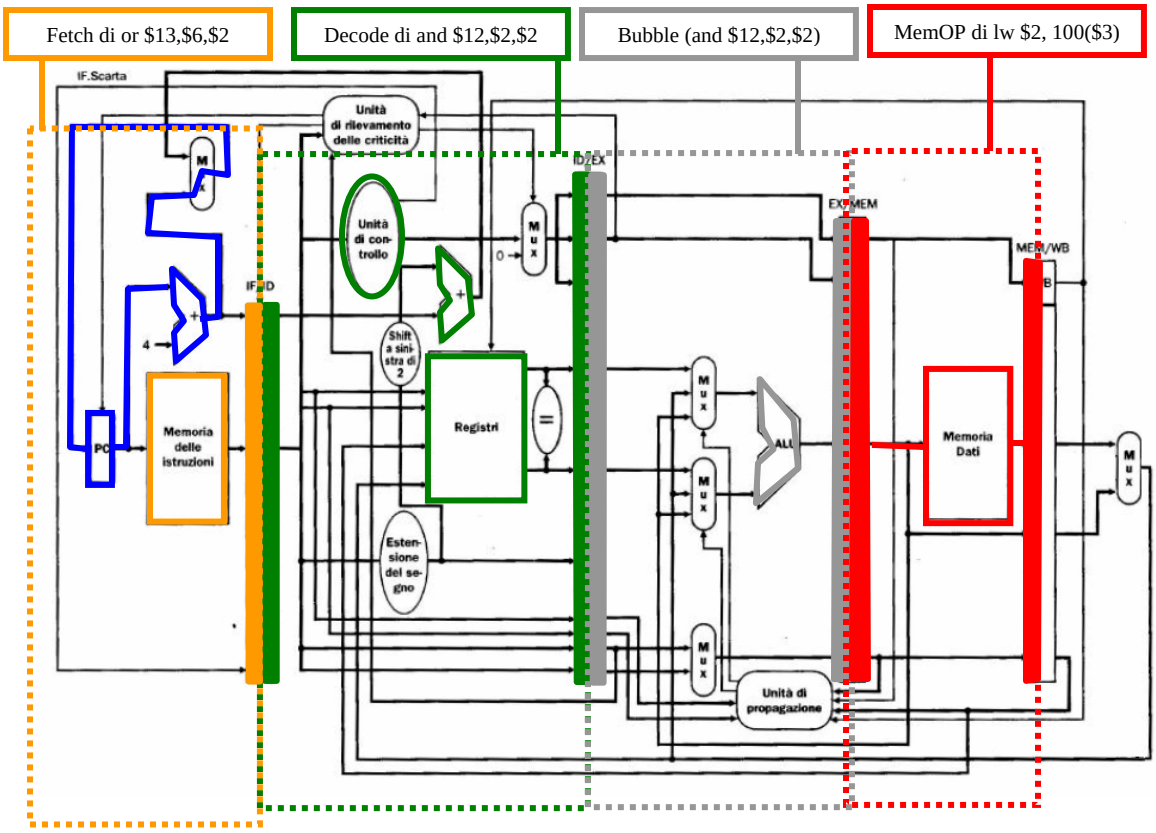
Come ne caso precedente esiste una dipendenza tra l'istruzione **lw** e le istruzioni direttamente seguenti. In questo caso però al momento dell'esecuzione dell'istruzione **and** il dato della cella **100($3)** non è ancora stato estratto dalla memoria e quindi non è possibile anticiparne il risultato come nel caso precedente (criticità non risolvibile). Occorre quindi fermare la pipe almeno per un ciclo in attesa che il dato divenga disponibile.
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.



A questo punto però un opportuno circuito (**unità di rilevamento delle criticità**) è in grado di rilevare la criticità 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 introdurre una NOP virtuale (una *bolla*) nella pipe in modo

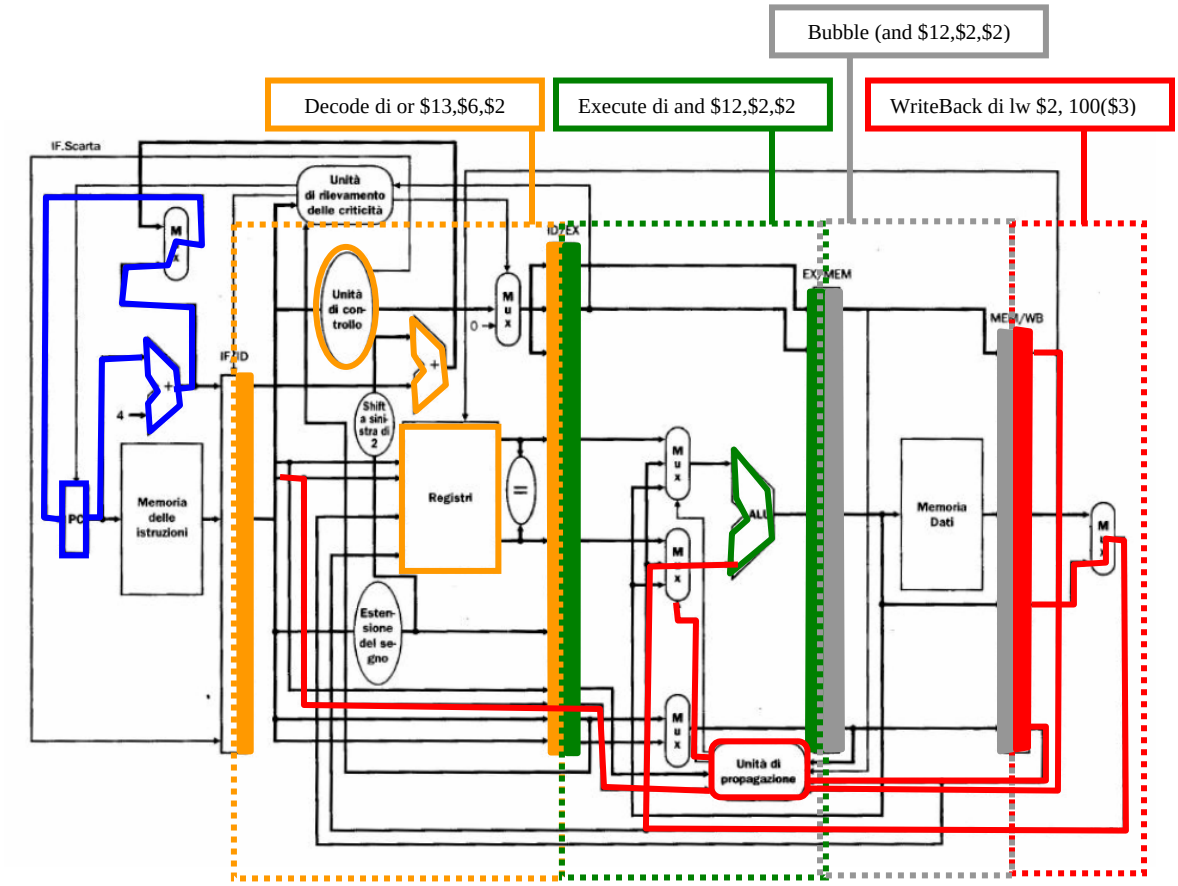
da ritardare l'esecuzione della **and** di un ciclo di clock. Questo consente di estrarre il dato richiesto dalla memoria e di renderlo disponibile all'unità di propagazione.
La NOP virtuale viene realizzata fermando per un ciclo di clock l'avanzamento della pipe ed invalidando il risultato dell'istruzione and già entrata nella pipe. L'unita di rilevamento delle 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 l'istruzione che ha causato la criticità con un ritardo di un ciclo, nell'esempio la **and**.
- inibisce gli effetti dell'istruzione che ha causato la criticità azzerando i segnali d controllo nel registro **ID/EX.** L'istruzione con i dati errati percorrerà comunque la pipe senza però generare effetti sull'esecuzione delle istruzioni.



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**

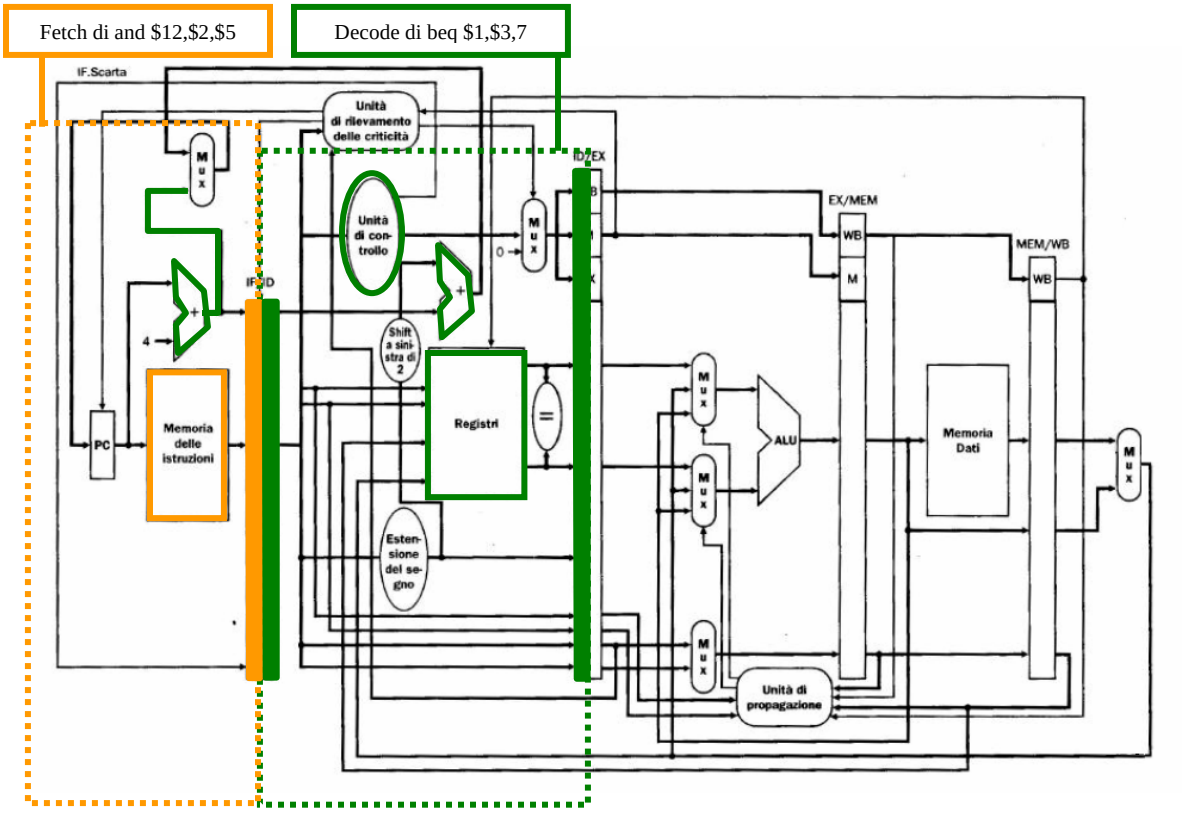


L'unità di rilevamento delle criticità permette al programmatore di non preoccuparsi di verificare la correttezza dell'esecuzione. Ciò non toglie l'introduzione delle NOP nella pipe riduce il numero di istruzioni eseguite dalla CPU. Un compilatore intelligente potrebbe in questi casi tentare di riordinare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

Le criticità nel flusso di esecuzione si verificano in presenza di salti condizionati. Si consideri il seguente esempio:

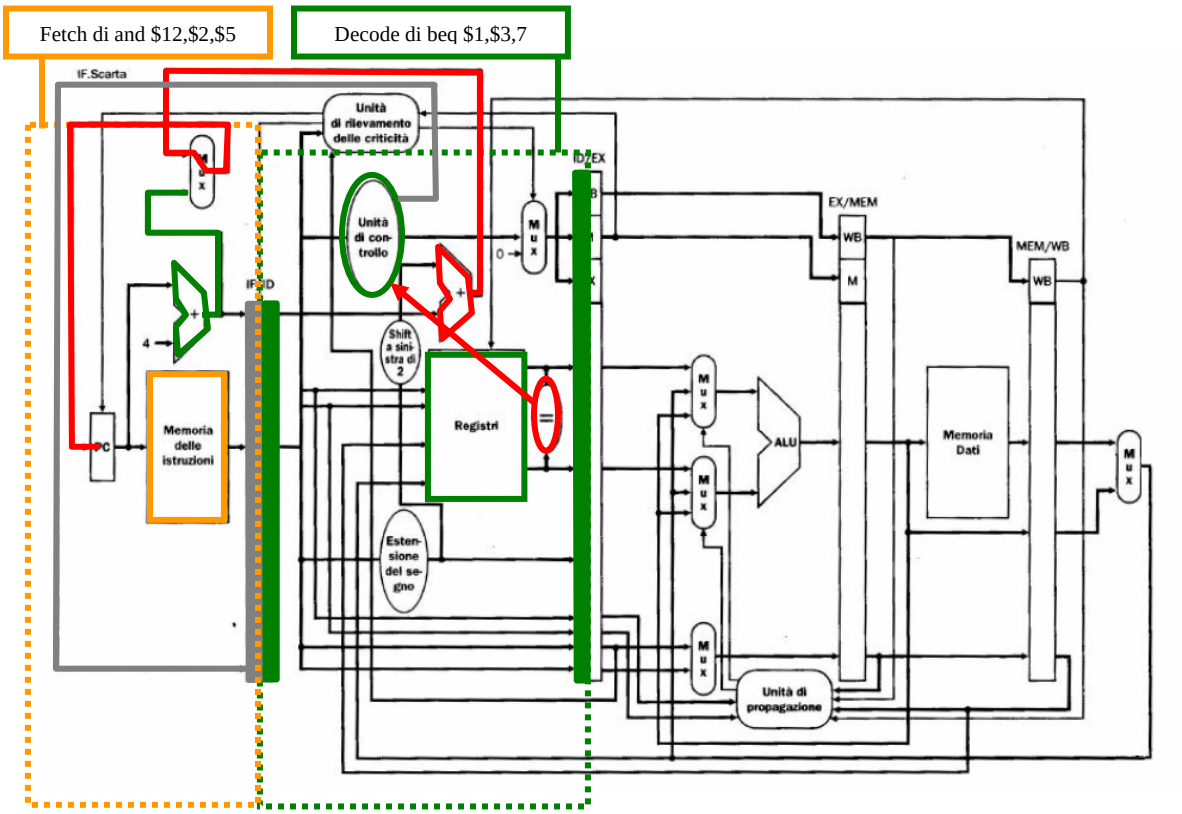
```
beq $1,$3, 7
and $12, $2, $5
or $13, $6, $2
and $14, $2, $2
lw $4, 50($7)
```

Ottimizzando il circuito è possibile eseguire il salto condizionato in 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 l salto si verifichi, questa istruzione "spuria" potrebbe alterare l'esecuzione corretta del programma assembly. La soluzione più semplice consiste nel introdurre appena dopo il salto sempre una istruzione NOP (una *bolla*) in modo da mettere in stallo la CPU in attesa che il risultato su salto sia deciso.
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 "spuria". Questo nel caso non si verifichi il salto fa guadagnare nell'esecuzione un ciclo di clock.



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**

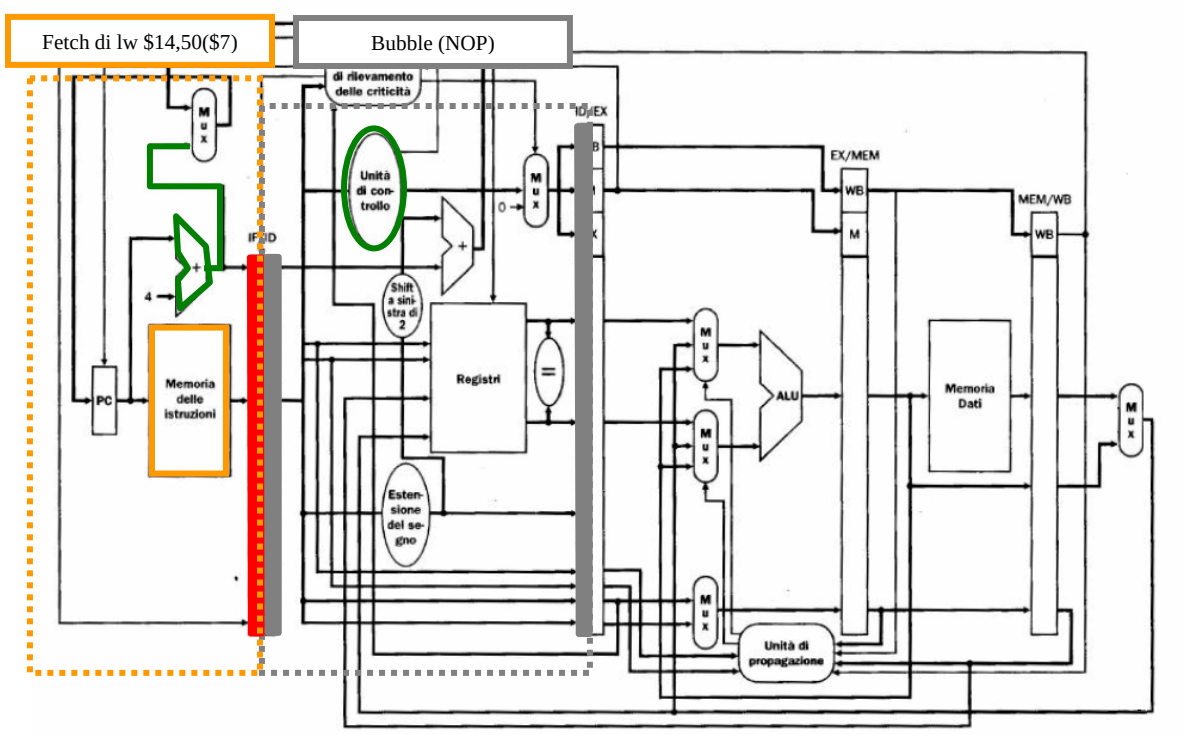
coincide o no[2]. L'unità di controllo quindi può decidere in fase di decodifica quale nuovo indirizzo caricare nel PC e, nel caso il salto si verifichi, può sostituire l'istruzione spuria entrata nella pipe con un istruzione di NOP.
Supponiamo che al tempo di decodifica dell'istruzione **beq $1,$3,7** 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 pronto ad eseguire la fase di fetch dell'istruzione **and $12,$2,$5.**



Nell'ipotesi che si verifichi il salto, l'unità di controllo dell'esempio:

- attraverso il segnale if.Scarta 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.

[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.



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