7 – Le CPU ad elevate prestazioni

7.1 – Introduzione alle architetture avanzate

Nei paragrafi precedenti di questa Sezione si è fatto riferimento ad un'organizzazione di un processore che, rifacendosi direttamente alla macchina di Von Neumann, ha un comportamento di tipo sequenziale: in ogni istante viene processata una sola istruzione alla volta. In questo caso il tempo necessario per eseguire un programma dipende sia dal numero di cicli di clock del processore per eseguire il programma e sia dalla durata del periodo del clock, cioè:

$$T_{exec} = N_{cicli} T_{clock}$$

dove:
- T_{exec} è il tempo di esecuzione di un programma;
- N_{cicli} è il numero di cicli di clock del processore per eseguire il programma;
- T_{clock} è la durata del ciclo di clock.

Oppure, tenendo conto della frequenza del clock (f_{clock}) con cui è pilotato il processore, si ha che:

$$T_{exec} = \frac{N_{cicli}}{f_{clock}}$$

Il numero di cicli di clock del processore per eseguire il programma è pari a:

$$N_{cicli} = N_{istr} CPI$$

dove:
- N_{istr} è il numero di istruzioni del programma;
- *CPI* è il numero medio di cicli di clock per eseguire un'istruzione (Clocks Per Instruction).

Quindi per diminuire il tempo di esecuzione di un programma o si aumenta la frequenza del clock, e questo non è sempre possibile data una certa tecnologia d'integrazione, e comunque esiste sempre il limite della velocità di trasmissione della luce, o si diminuisce il numero medio di istruzioni di un programma o quello dei cicli di clock per istruzione. Un'altra possibilità potrebbe essere quella di riformulare il programma sequenziale in modo che possa essere eseguito da più processori contemporaneamente. In questo caso, teoricamente, la velocità di esecuzione del programma potrebbe essere aumentata di un fattore pari al numero di processori utilizzati. Questa scelta porta alla definizione di architetture parallele e/o vettoriali, conosciute anche come SIMD (Single-Instruction Multiple-Data) o MIMD (Multiple-Instruction Multiple-Data), secondo la classificazione di Flynn. E' da sottolineare che parecchie architetture avanzate, derivate dalle scelte che verranno esposte successivamente, fanno uso anche del precedente approccio (vedere le architetture pipeline superscalari).

Ritornando alle considerazioni di come aumentare le prestazioni di un singolo processore, la scelta di ridurre il numero medio di istruzioni utilizzate per scrivere un programma è la soluzione che adottano i progettisti delle architetture *CISC* (Complex Instruction Set Computer = calcolatore a repertorio di istruzioni complesso). Infatti, prevedendo un gran numero di tipi di istruzioni, di cui alcune molto complesse, e con molti modi di indirizzamento, si ha la possibilità di ridurre il numero di istruzioni dei programmi, con lo svantaggio, però, di dover aumentare il *CPI*.

Per ridurre il numero medio di cicli di clock per istruzione si possono eseguire più vie non in contrapposizione tra loro. Si ricorda che i cicli istruzione sono composti da uno o più cicli macchina ed ogni ciclo macchina, a sua volta, è composto di una fase di interazione del processore con il mondo esterno (memoria o dispositivi di ingresso/uscita) e di una fase di attività interne al processore (decodifica o esecuzione di una istruzione). A sua volta la fase di esecuzione può essere composta di più attività elementari. Quindi per diminuire il tempo di esecuzione di un programma si potrebbe intervenire a livello d'interazione processore-mondo esterno o a livello di attività interne del processore. Per quanto riguarda il miglioramento dell'interazione processore-dispositivi di ingresso/uscita normalmente si demanda ai canali di comunicazione (vedere paragrafi precedenti). Particolare attenzione comunque deve essere rivolta verso l'ottimizzazione dell'interazione con la memoria, anche perché l'esecuzione di ogni istruzione prevede una fase di *fetch*, in cui il codice dell'istruzione viene prelevato dalla memoria. Come abbiamo visto per ridurre i tempi di accesso alla memoria si possono utilizzare diverse soluzioni, come visto nei paragrafi/capitoli, quali l'utilizzo delle memorie cache e/o interleaving e il prefetching.

Un'altra possibilità di riduzione dei tempi di esecuzione dei programmi è quella di permettere l'esecuzione di più istruzioni dello stesso programma contemporaneamente, come avviene nei processori con architettura pipeline, superpeline e superscalare. Le soluzioni che permettono la presenza di più programmi contemporaneamente (il così detto parallelismo a livello di threads/processi) sono al di fuori degli obiettivi di questo Capitolo.

Di seguito vedremo come ridurre i tempi di esecuzione dei programmi tramite le architetture pipeline, super pipeline e superscalari, come evoluzioni della pipeline. In particolar modo si introdurrà prima una architettura pipeline di tipo RISC, il cui tipo non solo è stata la prima soluzione in cui sono state introdotte le architetture pipeline, ma ha rappresentato e rappresenta ancora una grande fetta del mercato dei processori. Una volta visto il metodo per realizzare tale tipo di architettura e analizzati i suoi lati positivi e negativi, si progetterà una soluzione pipeline del processore z64. Questa soluzione è più complessa di quella RISC, ciò è dovuto al fatto che il set di istruzioni utilizzato (che è un sottoinsieme dell'X-86) prevede dei metodi di indirizzamento molto più complessi di quelli delle soluzioni RISC. Inoltre per motivi pedagogici mentre per la prima soluzione prevediamo un set di istruzioni minimo, ma sufficiente per far vedere il metodo di progettazione e le relative problematiche, per la seconda soluzione si farà vedere una soluzione architetturale in grado di supportare un elevato numero di istruzioni dello z64.

7.2. - Architetture RISC

Le scelte architetturali fatte per i processori RISC nascono dalle seguenti considerazioni:

- nei codici oggetto prodotti dai compilatori per i processori CISC la distribuzione delle istruzioni macchina non è uniforme: circa il 10% delle istruzioni ricorrono nel 90% dei casi, mentre le più complesse vengono raramente usate; inoltre le istruzioni più usate sono anche quelle più semplici (come il trasferimento dati da memoria a registro e viceversa, operazioni tra il contenuto di registri);
- le istruzioni complesse, con differenti tipi di indirizzamenti, necessitano di una unità di controllo complessa, con conseguente rallentamento dell'esecuzione del ciclo macchina e considerevole occupazione di spazio fisico nel chip del processore;
- il tempo di esecuzione delle istruzioni macchina è fortemente dipendente dal tempo di accesso alla memoria di lavoro (RAM statiche o dinamiche) per prelevare le istruzioni (fetch) e per leggere/scrivere i dati;

- le istruzioni macchina che comportano solo operazioni tra il contenuto dei registri interni del processore, non dovendo interagire con la memoria di lavoro per la lettura/scrittura dei dati, hanno un tempo di esecuzione molto limitato;
- le limitazioni di velocità di accesso alla memoria esterna al processore sono principalmente dovute alle comunicazioni in-out tra chip (con conseguente uso di bus per poter utilizzare dispositivi standard – off the shelf);
- il miglioramento delle tecnologie di integrazione VLSI permette di integrare nello stesso dispositivo (chip) sia il processore sia le memorie cache;
- è possibile organizzazione il codice oggetto in modo che istruzioni consecutive non aggiornino lo stesso insieme di registri, ciò consente di poter eseguire contemporaneamente più istruzioni purché utilizzino sottosistemi indipendenti dello SCA del processore.

Queste considerazioni hanno portato a:
- utilizzare il minor numero di tipi di istruzione;
- utilizzare solo due tipi di istruzioni per la lettura e scrittura dei dati dalla memoria;
- utilizzare istruzione di manipolazione dei dati che facciano riferimento solo ad operandi memorizzati nei registri interni del processore;
- eseguire più istruzioni contemporaneamente;
- usare una struttura pipeline nel sottosistema di calcolo del processore (data path);
- utilizzare una cache per le istruzioni da eseguire ed una cache per i dati da elaborare.

Si vuole sottolineare che la scelta di usare un set ridotto di istruzioni è stata anche una necessità nella progettazione dei primi processori. Infatti, per ragioni di costo, quando ancora la tecnologia non era ai livelli di oggi, si dovevano usare pochi registri interni e unità di controllo con dimensioni fisiche ridotte. Naturalmente le istruzioni complesse erano emulate con un insieme di istruzioni più semplici. Oggi è possibile utilizzare un numero di registri più elevato, comunque si fa sempre ricorso ad una unità di controllo, la più semplice possibile, per migliorare al massimo le prestazioni.

Come si vedrà nel seguito le scelte progettuali elencate permettono di sfruttare al meglio la velocità intrinseca dei componenti del processore e di interagire il meno possibile con la memoria di lavoro e di massa e quindi di velocizzare l'esecuzione dei programmi. Comunque, a fronte di questi vantaggi occorre sottolineare anche qualche inconveniente dei processori RISC. Questi derivano dal numero ridotto di istruzioni; ciò comporta che il software prodotto per un processore RISC, a parità di funzione da espletare, occupa più memoria, sia staticamente che dinamicamente, rispetto a quello necessario per il software per un processore CISC. L'aumento statico di occupazione di memoria è dell'ordine del 30-50%, mentre quello dinamico (cioè il numero di bytes di codice effettivamente prelevato dalla memoria) è dell'ordine del 10-30%. Quest'aumento sensibile di codice comporta un incremento della memoria di lavoro e di memoria di massa, ma fortunatamente questo aumento di memoria è abbondantemente compensato dai continui miglioramenti della relativa tecnologia in termini di prestazioni, densità di memorizzazione e di costo. Per quanto riguarda l'aumento dinamico del codice da eseguire, questo è abbondantemente compensato dall'aumento del throughput.

Di seguito, dopo aver introdotto un set molto ridotto di istruzioni per un'architettura RISC tipica, introdurremo la tecnica del pipeline, dopodiché si evidenzieranno i problemi che possono insorgere nell'esecuzione dei programmi e delle possibili soluzioni, infine si daranno delle indicazioni su come modificare l'architettura di base per migliorare ulteriormente le prestazioni.

7.2.1. – Il set delle istruzioni

Il set delle istruzioni che si introduce, anche se costituito da un piccolo numero di istruzioni, è sufficiente a supportare un gran numero di algoritmi manipolanti numeri interi e nel contempo

permette di evidenziare quali scelte progettuali è necessario fare durante il progetto di un processore di tipo RISC e quali accorgimenti hardware e software è necessario utilizzare per evitare fenomeni indesiderati. Questo set non è un sottoinsieme in senso stretto delle istruzioni di un processore commerciale, anche se presenta le stesse caratteristiche. Le istruzioni a cui facciamo riferimento sono solo di 4 tipi:

- logiche/aritmetiche su numeri interi;
- caricamento/memorizzazione;
- salto condizionato;
- non operativa.

L'istruzione non operativa (che verrà identificata con *nop*) viene introdotta non per manipolare dati o modificare l'ordine di esecuzione delle istruzioni, ma per risolvere, come vedremo, a software alcuni problemi di correttezza di esecuzione di programmi scritti per processori multiciclo su sistemi di elaborazione utilizzanti processori ad architettura pipeline in cui non sono previsti accorgimenti hardware che rendono equivalente l'esecuzione dei programmi a quelli eseguiti sui processori multiciclo.

Nel presentare il formato delle istruzioni si ipotizza di avere un processore a 32 bit e con 32 registri visibili dal programmatore.

Istruzioni logiche/aritmetiche (istruzioni di tipo L/A)

Istruzione	*Sintassi*	*Semantica*
Somma	add regsorg1, regsorg2, regdest	(regdest) = (regsorg1) + (regsorg2)
Sottrazione	sub regsorg1, regsorg2, regdest	(regdest) = (regsorg1) - (regsorg2)
Prod. Logico	and regsorg1, regsorg2, regdest	(regdest) = (regsorg1) **and** (regsorg2)
Som. Logica	or regsorg1, regsorg2, regdest	(regdest) = (regsorg1) **or** (regsorg2)
Neg. logica	not regsorg1, regdest	(regdest) = **not** (regsorg1)

Formato delle istruzioni L/A

opcode	regsorg1	regsorg2	regdestL/A	non utilizzati
31-26	25-21	20-16	15-11	10-0

Notare che per uniformare il formato delle istruzioni di tipo L/A, l'identificativo del registro sorgente dell'istruzione "nop" viene duplicato. Inoltre si sono fissati a 6 i bit per il codice operativo perché si ipotizza che non ci siano solo le istruzioni elencate.

Esempi di codice

add 2, 3, 1 -- somma il contenuto dei registri 2 e 3, il risultato mettilo nel registro 1

000001	00010	00011	00001	----------

sub 5, 6, 4 -- sottrai il contenuto del registro 5 con quello di 6, il risultato mettilo nel registro 4

000010	00101	00110	00100	----------

and 2, 3, 1 -- fai l'and tra il contenuto dei registri 2 e 3, il risultato mettilo nel registro 1

000011	00010	00011	00001	----------

or 8, 9, 7 -- fai l'or tra il contenuto di registri 8 e 9, il risultato mettilo nel registro 7

000100	01000	01001	00111	----------

not 8, 7 -- fai la negazione del contenuto del registro 8, il risultato mettilo nel registro 7

000101	01000	01000	00111	----------

Istruzioni caricamento/memorizzazione (istruzioni di tipo C/M) (L/S)

Istruzione	*Sintassi*	*Semantica*
Caricamento di parola	load regdest, offset(regbase)	(regdest) = memoria[offset+(regbase)]
Memorizzazione di Parola	store regsorgM, offset(regbase)	memoria[offset+(regbase)] = (regsorg)

Formato dell'istruzione load

opcode	regbase	regdestC	Offset
31-26	25-21	20-16	15-0

Esempio di codice

load 1, 32(3) -- trasferisci il contenuto della locazione di memoria il cui indirizzo è dato dalla somma di 32 con il contenuto del registro 3 nel registro 1

000110	00101	00001	0000000000100000

Formato dell'istruzione store

opcode	regbase	regsorgM	Offset
31-26	25-21	20-16	15-0

Esempio di codice

Store 7, 16(4) -- trasferisci il contenuto del registro 7 nella locazione di memoria il cui indirizzo è dato dalla somma di 16 con il contenuto del registro 4.

000111	00100	00111	0000000000010000

Istruzioni di salto condizionato (istruzioni di tipo S) (J)

sintassi *semantica*

Istruzione	*Sintassi*	*Semantica*
salta se flag X = =1 (dove X può essere uno dei flag del registro di stato)	jumpX indirizzo	se flag X = =1 allora PC=PC+S+indirizzo (dove S è un intero che dipende da come è organizzata la memoria, per esempio in un processore a 32 bit con il banco di memoria della cache costituita con moduli di memoria di 8 bit il suo valore è pari a 4)

Formato dell'istruzione

opcode	flag	Indirizzo
31-26	25-23	22-0

Esempio di codice

jumpC 1026 -- se il bit di CARRY = = 1 allora metti il PC ad un valore pari a (PC)+S+1026.

001000	001	00000000000001000000010

Istruzione non operativa (NOP)

Istruzione	*Sintassi*	*Semantica*
Nulla	Nop	Non modificare nulla

Formato dell'istruzione nop

opcode			
31-26		25-0	

Esempio di codice

nop -- non fare nulla

000000	0000	0000	0000000000000000

E' importante fare il punto sulla disposizione dei campi nei formati istruzione elencati, che sarà utile nel paragrafo successivo:

- il campo codice-operativo è sempre contenuto nei bit 31-26;
- i due registri sorgente delle operazioni logiche/aritmetiche sono specificati, rispettivamente, nei bit 25-21 e 20-16, nella stessa posizione troviamo specificato anche il registro base e registro sorgente dell'istruzione store, mentre il registro base dell'istruzione load è specificato nei bit 25-21; questa disposizione semplificherà la progettazione del data path, in quanto, come si vedrà, sarà possibile accedere al banco dei registri per prelevare gli operandi indipendentemente dal tipo di istruzione;
- il registro destinazione delle operazioni logiche/aritmetiche è specificato nei bit 15-11, mentre quello dell'istruzione load nei bit 20-16; questo disallineamento purtroppo complicherà un po' l'architettura del data path;
- l'offset sia nelle istruzioni load che store è memorizzato nei bit 15-0.

7.2.2. – La tecnica del pipeline

Come accennato più volte per diminuire il tempo di esecuzione dei programmi è possibile progettare un processore con una struttura pipeline, che permette di eseguire contemporaneamente le attività relative ad ogni singola fase del processamento di ogni istruzione con quelle relative alle fasi di altre istruzioni. Una struttura pipeline è come una catena di montaggio in cui ogni operatore effettua una lavorazione elementare su un oggetto fornitogli dall'operatore precedente. In questo modo è possibile far uscire un oggetto dalla catena di montaggio nel tempo necessario per completare una singola fase di lavorazione. Naturalmente non si riduce il tempo di completamento di ogni singolo oggetto, ma si aumenta la frequenza con cui escono gli oggetti dalla catena di montaggio: se la catena è costituita di N stadi è possibile potenzialmente aumentare il throughput di N volte. Come in una catena di montaggio, per non rallentare il funzionamento, l'esecuzione di ogni singola fase, processata in uno stadio distinto, comporta l'utilizzazione di unità funzionale non condivisibili con altri stadi. Questa peculiarità comporta scelte architetturali contrarie a quelle effettuate per i processori multiciclo, in cui le unità funzionali potevano essere utilizzate durante l'esecuzione di differenti fasi, come ad esempio il bus interno, che poteva essere utilizzato sia per trasferire dati che codici "istruzione", o l'ALU che poteva essere utilizzata sia per le operazioni tra dati memorizzati nei registri che per incrementare il PC. Una volta che le attività di una fase sono completate è necessario trasferire il risultato delle attività allo stadio successivo e contemporaneamente prelevare l'informazione da manipolare dallo stadio precedente o dall'ingresso. Per disaccoppiare funzionalmente ed elettricamente i differenti stadi di una struttura pipeline si interpongono dei registri dove memorizzare l'informazione da elaborare, come specificato nella figura successiva (7.1). Scopo dei registri di interfaccia tra i vari stadi è quello di mantenere stabile l'informazione da manipolare per tutto il tempo necessario allo stadio. Il passaggio dell'informazione da uno stadio ad un altro è abilitato da un segnale di controllo periodico (il *clock*). Funzione del clock è quella di scandire gli istanti di tempo in cui gli stadi della struttura pipeline si scambiano le informazioni da elaborare, pertanto il periodo del clock deve essere sufficientemente lungo da permettere ad ogni singolo stadio di completare le proprie attività sulle informazioni passategli dallo stadio precedente. Quindi il periodo minimo del clock sarà pari al massimo tra i tempi di elaborazione di ogni singolo stadio.

Schema di principio delle architetture pipeline

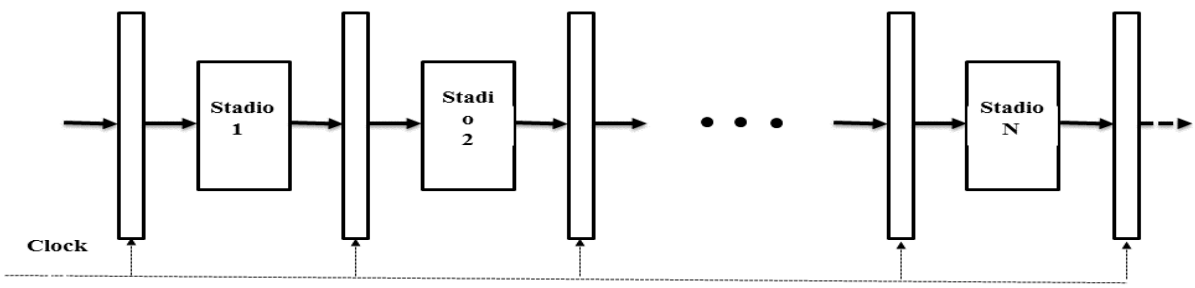


Fig. 7.1 - Schema di principio delle architetture pipeline.

Di seguito, dopo aver analizzato le fasi necessarie per processare ogni singola istruzione, si farà vedere l'hardware necessario per eseguirle ed infine si darà l'architettura completa.

7.2.2.1. Fasi di esecuzione delle istruzioni

Per eseguire un'istruzione logica/aritmetica è necessario:
- prelevare l'istruzione dalla memoria ed incrementare di S il PC;
- decodificare l'istruzione e leggere il contenuto dei registri sorgente;
- effettuare l'operazione logica o aritmetica specificata;
- memorizzare nel registro destinazione il risultato dell'operazione.

Per eseguire un'istruzione di caricamento è necessario:
- prelevare l'istruzione dalla memoria ed incrementare di S il PC;
- decodificare l'istruzione e leggere il contenuto del registro base;
- calcolare l'indirizzo della locazione di memoria da cui prelevare il dato;
- leggere il contenuto della locazione di memoria in cui è memorizzato il dato;
- memorizzare nel registro destinazione ciò che è stato letto dalla memoria.

Per eseguire un'istruzione di memorizzazione è necessario:
- prelevare l'istruzione dalla memoria ed incrementare di S il PC;
- decodificare l'istruzione e leggere il contenuto del registro sorgente del dato da memorizzare e il contenuto del registro base;
- calcolare l'indirizzo della locazione di memoria in cui scrivere il dato letto dal registro sorgente;
- memorizzare nella locazione di memoria ciò che è stato letto dal registro sorgente.

Per eseguire un'istruzione di salto condizionato è necessario:
- prelevare l'istruzione dalla memoria ed incrementare di S il PC;
- decodificare l'istruzione;
- calcolare l'indirizzo della locazione di memoria da cui prelevare l'istruzione successiva nel caso in cui il flag di stato selezionato è pari ad 1;
- dipendendo dal valore del flag selezionato aggiornare il PC o lasciarlo invariato.

Per eseguire un'istruzione nulla è necessario:
- prelevare l'istruzione dalla memoria ed incrementare di S il PC;
- decodificare l'istruzione.

Da quanto descritto si può notare che ogni istruzione può essere completata in più passi e che l'istruzione di caricamento (load) è quella che necessita del maggior numero di passi elementari: cinque. Di seguito vengono elencate le attività di ogni singolo passo in funzione del tipo di istruzione.
- Nel primo passo si preleva l'istruzione da eseguire (*fetch* dell'istruzione), questo passo è indipendente dal tipo di istruzione e pertanto è uguale per tutti e quattro i tipi di istruzione.
- Nel secondo passo si prevede la decodifica dell'istruzione (*instruction decode*) e il prelievo del contenuto di due registri (per le istruzioni di tipo L/A includendo anche l'istruzione not, in

quanto pur necessitando di un solo operando per semplificare il data path si accede due volte allo stesso valore), di un registro (per le istruzioni di tipo C/M) o di nessun registro (per le istruzioni di tipo S).

- Nel terzo passo (*execute*) ogni singolo tipo di istruzione effettua delle operazioni logiche o aritmetiche: le istruzioni di tipo L/A per eseguire l'operazione specificata dall'istruzione, le istruzioni di tipo C/M per calcolare l'indirizzo della locazione di memoria in cui andare a leggere/scrivere il dato, mentre l'istruzione di salto per determinare l'indirizzo dell'istruzione da eseguire, nel caso in cui sia necessario effettuare il salto.
- Nel quarto passo le attività dipendono fortemente dal tipo di istruzione e solo l'istruzione load necessita di un quinto passo. In particolare le istruzioni ***store*** memorizzano il dato in memoria (*memory*), le istruzioni ***load*** prelevano il dato dalla memoria (*memory*); le istruzioni di salto condizionato aggiornano o meno il PC con il valore identificato nel terzo passo; mentre le istruzioni logiche/aritmentiche devono scrivere il dato nel registro destinazione, ciò che è stato elaborato nel passo precedente (*write back*).
- Infine nel quinto passo l'istruzione load memorizza nel registro destinazione ciò che ha letto nella memoria nel quarto passo (*write back*).

7.2.2.2. Progetto del data path

Le istruzioni scelte vengono eseguite al massimo in cinque passi e, quindi, volendo associare ad ogni passo uno stadio, la struttura pipeline sarà costituita di cinque *stadi*. Anche se alcune istruzioni potrebbero essere eseguite in meno passi, per non complicare troppo l'architettura tutte le istruzioni sono eseguite in cinque cicli di clock, inoltre per semplificare ulteriormente il progetto ogni stadio è specializzato per una singola funzione: Fetch, Instruction Decode, Execute, Memory e Write Back. Ogni singolo passo di ogni istruzione è eseguito nello stadio corrispondente. Per esempio il quarto passo delle istruzioni di tipo L/A viene eseguito nel quinto stadio (Write Back) della struttura pipeline, ovviamente nel quarto stadio (Memory) questo tipo di istruzione non viene processata.

La sequenza dei passi per l'esecuzione di ogni singola istruzione è schematizzata in figura 7.2, che corrisponde anche all'organizzazione logica della struttura pipeline. Si rileva di nuovo che solo l'istruzione load è interessata a tutti e cinque gli stadi, mentre le altre solo ad un sottoinsieme.

In figura 7.3 viene mostrata la sovrapposizione temporale delle attività nell'esecuzione di più istruzioni. Dalla figura si può notare che mentre la prima istruzione della sequenza è nello stadio di Write Back, la seconda è nello stadio di Memory, la terza nello stadio di Execute, la quarta nello stadio di Instruction Decode e la quinta nello stadio di Fetch.

Fetch	Instruction Decode	Execute	Memory	Write Back

Fig. 7.2 - Sequenza dei passi per l'esecuzione di ogni istruzione - organizzazione logica della struttura pipeline.

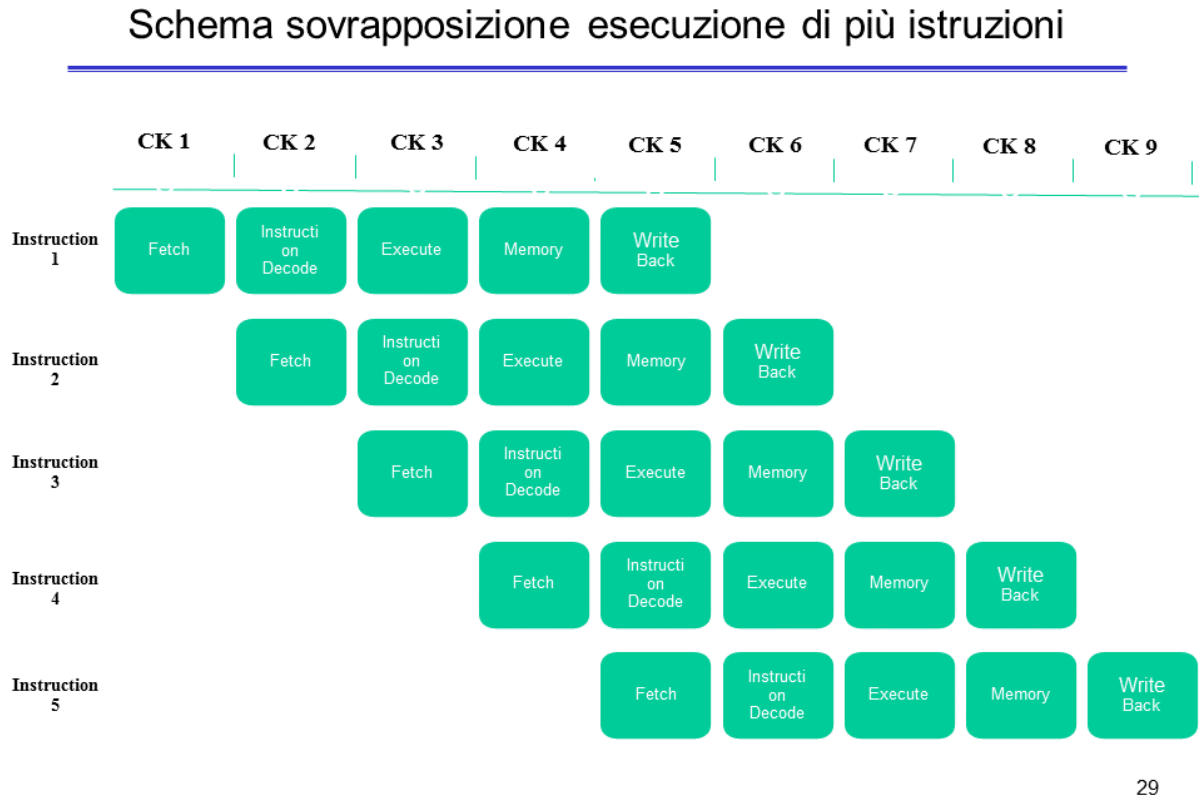


Fig. 7.3 - Schematizzazione della sovrapposizione delle attività nell'esecuzione di più istruzioni.

Volendo si potrebbe realizzare una struttura pipeline con un numero inferiore di stadi, facendo eseguire in un singolo stadio più di un passo delle istruzioni; ciò comporta, però, un allungamento del periodo di clock (il cui valore minimo, come già detto, è pari al massimo tra i tempi d'elaborazione di ogni singolo stadio) rallentando di conseguenza la velocità di passaggio dei dati da uno stadio all'altro.

Per l'individuazione del data path è sufficiente specificare per ogni singolo stadio l'hardware necessario per eseguire il relativo passo di ogni tipo di istruzione. I primi quattro stadi necessitano di un registro per disaccoppiare le loro attività. Per comodità di presentazione i cinque stadi vengono denominati con F, D, E, M,WB che rappresentano le lettere iniziali dei loro nomi; mentre i registri tra uno stadio e l'altro della struttura pipeline con F/D, D/E, E/M, M/WB.

Stadio di Fetch (F)

Nello stadio di fetch ogni singola istruzione è prelevata dalla memoria e il PC è incrementato di S unità. Come già accennato il valore di S dipende dal numero di bit del processore e dal numero di moduli della memoria cui il processore deve accedere in parallelo per prendere un'istruzione.

Per leggere i codici delle istruzioni è necessaria una memoria cache, un registro per memorizzare il PC (che identifica l'indirizzo di memoria da cui leggere le singole istruzioni) e un addizionatore per incrementare il contenuto del PC. Il data path può essere strutturato come in figura 7.4. Il data path verrà successivamente modificato per tenere conto delle istruzioni di salto

condizionato, infatti, in questo caso, se il flag selezionato è pari ad 1 occorrerà aggiornare il PC per saltare all'istruzione localizzata a: PC+S+indirizzo (indirizzo specificato nei bit 22-0 del formato istruzione).

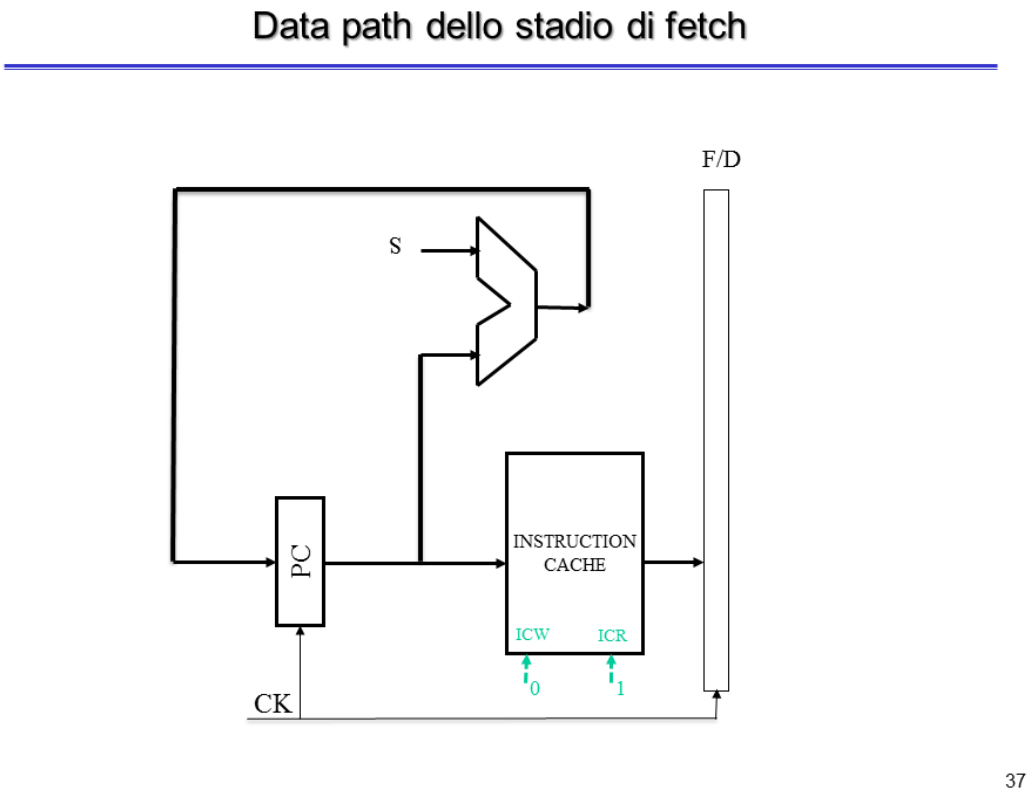


Fig. 7.4 - Data path dello stadio di Fetch.

Da notare che ipotizzando che la cache istruzioni abbia tutte le istruzioni del programma da eseguire questa debba essere solo letta. Pertanto la memoria cache è abilitata sempre in lettura; pertanto, ipotizzando di utilizzare due segnali di controllo, uno per la lettura (ICR) ed uno per la scrittura (ICW) (anche se sarebbe sufficiente un singolo segnale ipotizzando che quando asserito abiliti la lettura e quando negato la scrittura o viceversa), questi saranno cablati ad 1 e 0, rispettivamente (se in logica positiva 1 equivale alla connessione all'alimentazione e 0 alla connessione alla massa).
Inoltre il Program Counter deve essere modificato per l'esecuzione di ogni istruzione, per questo motivo l'abilitazione alla scrittura di quanto presente nel suo ingresso è effettuato dal segnale di Clock che abilita il trasferimento delle istruzioni da uno stadio all'altro.

Stadio di Instruction Decode (D)

In questo stadio il codice operativo dell'istruzione è inviato al controllore (vedere sottoparagrafo successivo) per la generazione dei comandi da inviare al data path degli stadi successivi, comandi (detti anche microrordini) necessari per completare l'esecuzione dell'istruzione. Inoltre per le istruzioni di tipo L/A si deve leggere il contenuto dei registri sorgenti, per quelle di tipo C/M il contenuto del registro base, mentre per quelle di salto condizionato non si deve leggere alcun

registro. Pertanto in questo stadio è sufficiente un banco di registri con due ingressi identificanti i registri da leggere. Poiché il controllore, a questo punto dell'esecuzione dell'istruzione, non l'ha ancora decodificata, esso non può ancora intervenire per comandare la lettura di zero, uno o due registri. Pertanto ci si pone nella condizione più sfavorevole, che in questo caso coincide con la lettura di due registri, lasciando agli stadi successivi la possibilità di filtrare eventualmente i dati non utili. Ciò comporta che il banco di registri è sempre abilitato per la lettura di due registri. Ricordando il formato delle istruzioni si può notare che per le istruzioni L/A l'identificativo del regsorg1 è disponibile nei bit 25-21 del formato istruzione e quello di regsorg2 nei bit 20-16, per le istruzioni C/M l'identificativo del registro di base è memorizzato nei bit 25-21, mentre quello del regsorgM dell'istruzione store nei bit 20-16.

Inoltre, è da notare che è necessario trasmettere agli stadi successivi tutte quelle informazioni utili per la corretta esecuzione delle istruzioni. In particolare per le istruzioni in cui è previsto lo stadio di Write Back è necessario trasferire l'identificativo del registro destinazione (bit 15-11 del codice istruzione per le istruzioni di tipo L/A e bit 20-16 per le istruzioni di caricamento). Inoltre, sono da trasferire i bit relativi alla codifica dei flags (bit 25-23) per le istruzioni di salto, i bit dell'offset per le istruzioni di tipo C/M (bit 15-0) e i bit dell'indirizzo per le istruzioni di salto (bit 22-0). Il data path può essere strutturato come in figura 7.5.

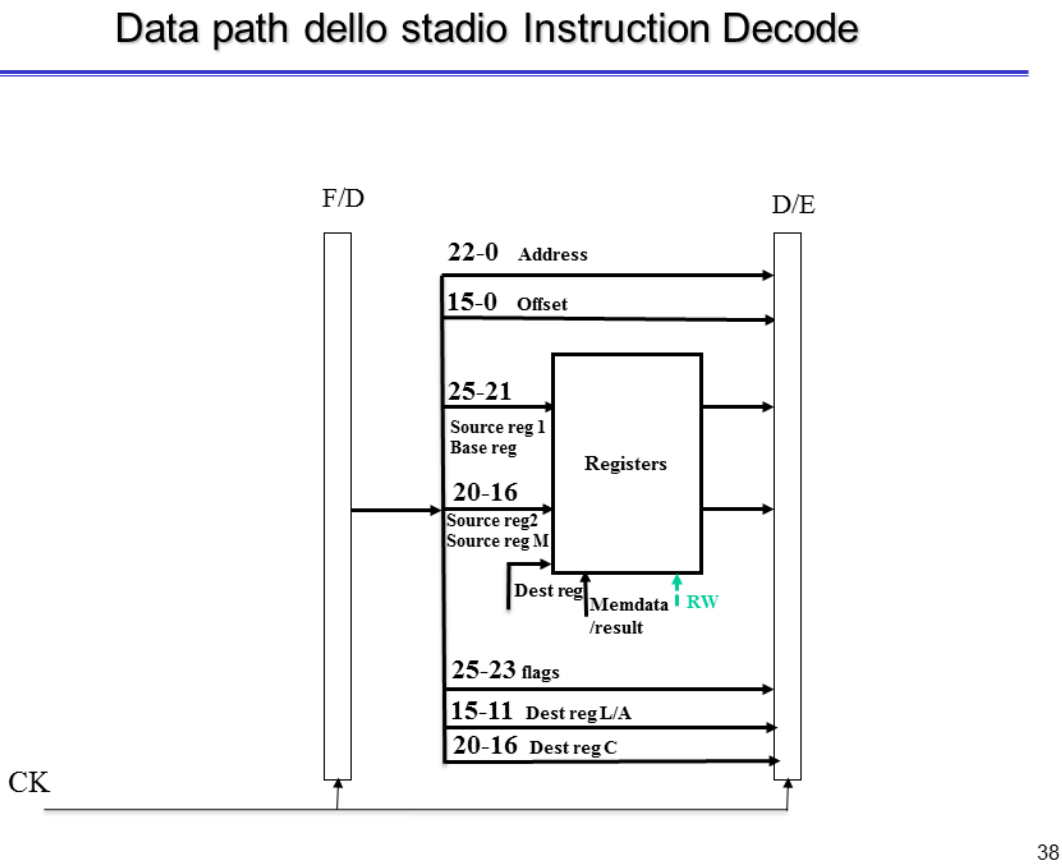


Fig. 7.5 - Data path dello stadio Instruction Decode.

E’ da notare che in questo stadio è necessario un solo segnale di controllo che permetta l’abilitazione in scrittura del registro destinazione delle operazioni logiche/aritmetiche o di load. Il segnale di controllo è denominato RW per Register Write e nel caso in cui è asserito permette la scrittura del dato/risultato nel registro destinazione, operandi che vengono propagati dagli stadi precedenti temporalmente la fase di Write Back.

Stadio di Execute (E)

In questo stadio le istruzioni di tipo L/A effettuano un’operazione logica o aritmetica in funzione del proprio codice operativo, le istruzioni di tipo C/M un’addizione per identificare la locazione di memoria in cui andare a leggere/scrivere il dato, mentre l’istruzione di salto un’addizione, per determinare l’indirizzo dell’istruzione da eseguire nel caso in cui è necessario effettuare il salto. Sembrerebbe quindi sufficiente un’ALU. Invece, si preferisce accoppiare all’ALU un addizionatore dedicato al calcolo degli indirizzi di memoria per non modificare i flags dell’ALU, che cosi’ dipendono solo dall’esecuzione delle istruzioni di tipo L/A, come deve avvenire correttamente per non modificare la semantica di un programma. E’ da notare che l’addizionatore dedicato per le istruzioni di tipo C/M deve sommare il contenuto del registro base con l'offset, mentre per le istruzioni di tipo S deve addizionare il valore aggiornato del PC con l'indirizzo, pertanto gli ingressi dell’addizionatore sono pilotati da due multiplexer comandati dal controllore. A tal fine il controllore, in funzione del tipo di istruzione da interpretare, utilizzerà due segnali di controllo per i due multiplexer: M1 e M2. Che porrà a 0 nel caso di istruzione di tipo S (salto) ed1 nel caso di istruzione di tipo C/M. L'addizionatore deve avere entrambi gli ingressi con lo stesso numero di bit, pertanto i bit dell'offset e dell'indirizzo sono estesi a 32 bit come quelli del contenuto del registro base e del PC. Nella figura i dispositivi estensori di bit sono identificati con la lettera E e sono posti prima dei relativi multiplexer.
Nel caso si esecuzione di istruzioni L/A (logiche/aritmetiche) l’ALU dovrà manipolare i bit degli operandi, propagati dallo stadio precedente e identificati dalle linee dati Sorg_reg_1 e Sorg_reg_2, in funzione del tipo di istruzione, per questo motivo il controllore dovrà generare i segnali di controllo appropriati (ALU_code) per comandare opportunamente l’ALU. Naturalmente tali segnali di controllo dovranno essere generati in funzione dell’OPCODE dell’istruzione da eseguire. Da notare che essendo l’ALU un circuito combinatorio la sua uscita sarà sempre presente, pertanto nel caso di esecuzione di istruzioni non L/A ci sarà sempre un’uscita, che verrà memorizzata nel registro di interfaccia E/M; sarà poi compito del controllore non utilizzare tali dati negli stadi successivi.

Nel caso di esecuzione di istruzioni di tipo S (salto), dato che il salto viene effettuato in funzione del valore del flag specificato nell’istruzione, è necessario verificare se tale bit è pari a 0 o ad 1, poiché il flip/flop specificato fa parte di un pool di flags, allocati nello Status Register (SR), è quindi necessario selezionarlo tra le uscite di SR, a tal fine si usa un multiplexer pilotato dai bit 25-23 del formato istruzione, in cui è codificato l’identificativo del bit di interesse.

Funzioni dello stadio di Execute sono anche quelle di propagazione delle informazioni necessarie agli stadi successivi anche se non necessitano di alcuna manipolazione. In particolar modo dal secondo stadio arriva l’identificativo del registro destinazione per le istruzioni che prevedono lo stadio di WB, pertanto quest'informazione deve essere ulteriormente propagata in avanti. E’ da notare però che mentre per le istruzioni di tipo L/A questo identificativo è memorizzato nei bit 15-11 del codice istruzione, nelle istruzioni di caricamento questo è memorizzato nei bit 20-16, pertanto è necessario un multiplexer (comandato dal controllore, tramite il segnale di controllo M3) per selezionare l’informazione corretta. Allo stadio Memory, infine, deve essere propagato

anche il contenuto del registro sorgente delle istruzioni di tipo store (Source_reg_M), che utilizza le stesse linee dati di Source_reg_2. Il data path può essere strutturato come in figura 7.6.

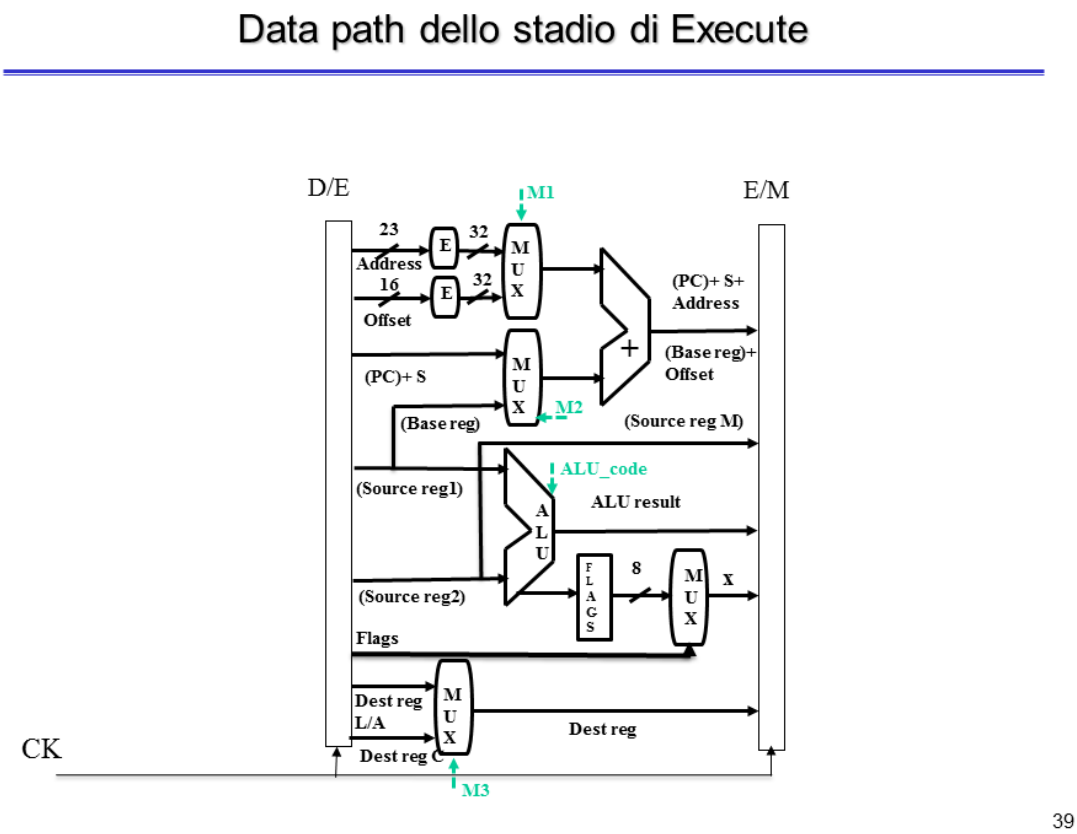


Fig. 7.6 - Data path dello stadio di Execute.

Stadio di Memory (M)

Questo stadio prende il nome dalle attività necessarie alle istruzioni di tipo C/M per interagire con la memoria. Però in questo stadio vengono eseguite anche le attività relative al quarto passo delle istruzioni di salto condizionato: propagazione o meno del PC con il valore calcolato nel terzo passo in funzione del valore del flag selezionato. Questo passo può anche essere eseguito nello stadio successivo, ma si preferisce eseguirlo prima per non dover propagare nel registro di pipeline successivo le informazioni necessarie per la sua esecuzione e per prendere al più presto la decisione dell'istruzione da eseguire. Come detto nel paragrafo relativo alla descrizione del data path dello stadio di Fetch, il PC dovrà esse aggiornato con il valore dell'indirizzo dell'istruzione successiva oppure, nel caso di salto, con l'indirizzo dell'istruzione identificata dall'istruzione di salto. Pertanto sarà necessario utilizzare un multiplexer che, in funzione del tipo di istruzione e dell'eventuale valore di un flags si stato, selezionerà uno dei due indirizzi precedenti. Da notare che mentre il primo valore è calcolato nel primo stadio, il secondo è calcolato nel quarto stadio.

Ciò potrà creare dei problemi di correttezza nell'esecuzione dei programmi, problema che verrà affrontato quando tratteremo della gestione dei rischi o hazard.

Nel caso di esecuzione di una istruzione di *store* o di *load* l'indirizzo di memoria a cui accedere viene propagato dallo stadio precedente, dove era stato determinato dall'addizionatore predisposto all'uopo. Nel caso di *store* il dato da memorizzare è quello che era stato propagato precedentemente anche dallo stadio Execute e proveniente dallo stadio di Instructione Decode ed in particolare dal banco dei registri (Source_reg_M), mentre nel caso di *load* il dato è quello acceduto in lettura dalla memoria cache utilizzando l'indirizzo propagato dallo stadio precedente. Per permettere la lettura o la scrittura dei dati in memoria cache il controllore genera due segnali di controllo: DCR (Data Cache Read) e DCW (Data Cache Write).
Anche in questo stadio, come nel precedente, alcune informazioni non debbono essere manipolate o utilizzate, ma solo propagate allo stadio successivo. In questo caso l'informazione da propagare è quella relativa all'uscita dell'ALU da utilizzare nel caso di esecuzione di istruzioni logiche/aritmetiche. Tale informazione è propagata tramite le linee identificate come ALU_result. Pertanto il data path dello stadio di Memory può essere strutturato come in figura 7.7.

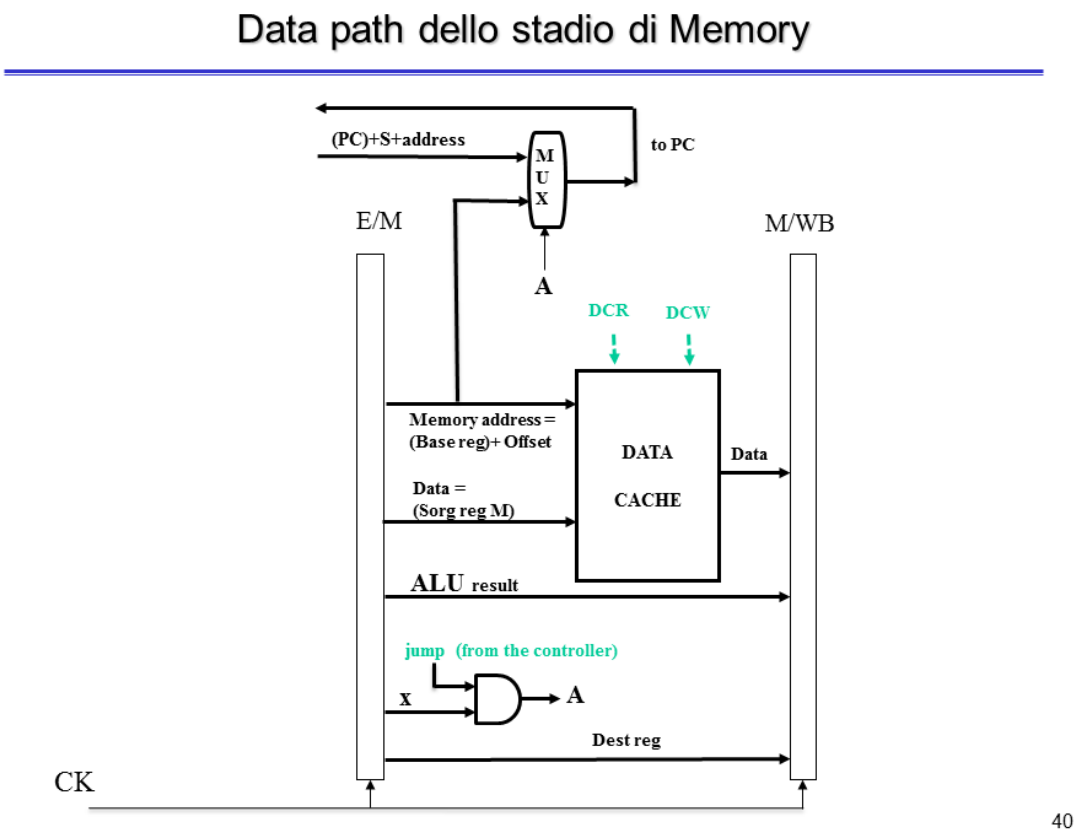


Fig. 7.7 - Data path dello stadio di Memory.

Come si vede dalla Figura 7.7 il multiplexer è pilotato dal segnale "A" che viene generato dall'AND tra il segnale di controllo *jump*, generato dal controllore a fronte della decodifica

dell'OPCODE relativo alle istruzioni di salto, e il valore del flip/flop di stato selezionato nello stadio precedente.

Stadio di Write Back (WB)

Questo stadio è necessario per completare l'esecuzione delle istruzioni di tipo L/A e di caricamento (*load*). Questo stadio, essendo l'ultimo della catena della struttura pipeline, non deve trasferire informazioni ad altri stadi e quindi non è necessario prevedere un ulteriore registro di interfaccia.

L'identificativo del registro destinazione è stato univocamente determinato nel terzo stadio, l'informazione da memorizzarvi invece ancora non lo è. Infatti, mentre nel caso di istruzioni di tipo L/A questo valore è pari all'uscita dell'ALU, nel caso di istruzioni di tipo *load* questo è l'uscita della memoria, pertanto è necessario un multiplexer (pilotato dal controllore, tramite il segnale di controllo MWB, Mux Write Back) per la loro selezione. L'abilitazione della scrittura nel registro destinazione è specificata dal controllore. Il data path può essere strutturato come in figura 7.8. L'architettura del banco dei registri deve permettere la lettura (effettuata nel secondo stadio) e la scrittura (effettuata nel quinto stadio) dei registri nello stesso ciclo di clock.

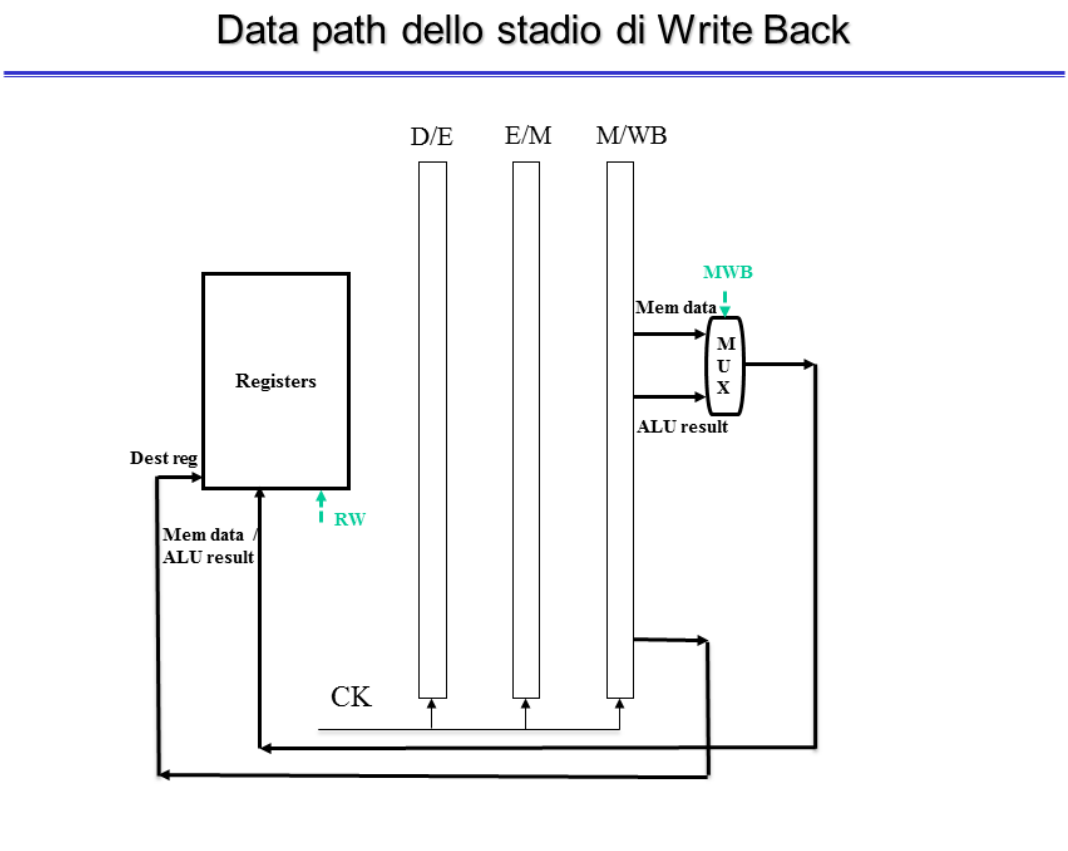


Fig. 7.8 - Data path dello stadio di Write Back.

Da notare che il segnale di controllo relativo alla scrittura del dato nel banco registri agisce nello stadio di Write Back, che è il quinto della struttura pipeline, mentre il banco dei registri è allocato nel secondo stadio, pertanto il valore nel registro destinazione potrà essere aggiornato solo dopo tre colpi di clock da quando sono stai prelevati gli operandi relativi alla generazione del corretto

valore da scrivere nel registro destinazione. Pertanto se nel frattempo ci sono state istruzioni che hanno utilizzato il vecchio valore del registro destinazione, cioè quello prima della corretta scrittura della fase di Write Back, queste istruzioni hanno utilizzato dati non corretti e quinid il programma risulterà scorretto. Fenomeni di questo tipo si chiamano Data Hazard, di tipo Define use o Load use, e verranno tratte in suno dei successivi paragrafi.

Architettura complessiva

Aggregando i cinque stadi ora esaminati si ottiene l'architettura schematizzata in figura 7.9. Notare che il PC è aggiornato ad ogni colpo di clock (con il valore (PC)+S o con (PC)+S+indirizzo, dove nel secondo caso il (PC) si riferisce a quello di 3 cicli precedenti. Quando è necessario caricarci il valore calcolato nel terzo stadio, perché si è verificato che il flag specificato da un'istruzione di salto è pari ad 1, è necessario prevedere l'annullamento delle attività relative alle istruzioni negli stati di E, ID e F. Questo fenomeno conosciuto come detto rischio (hazard) sarà trattato in un prossimo paragrafo.

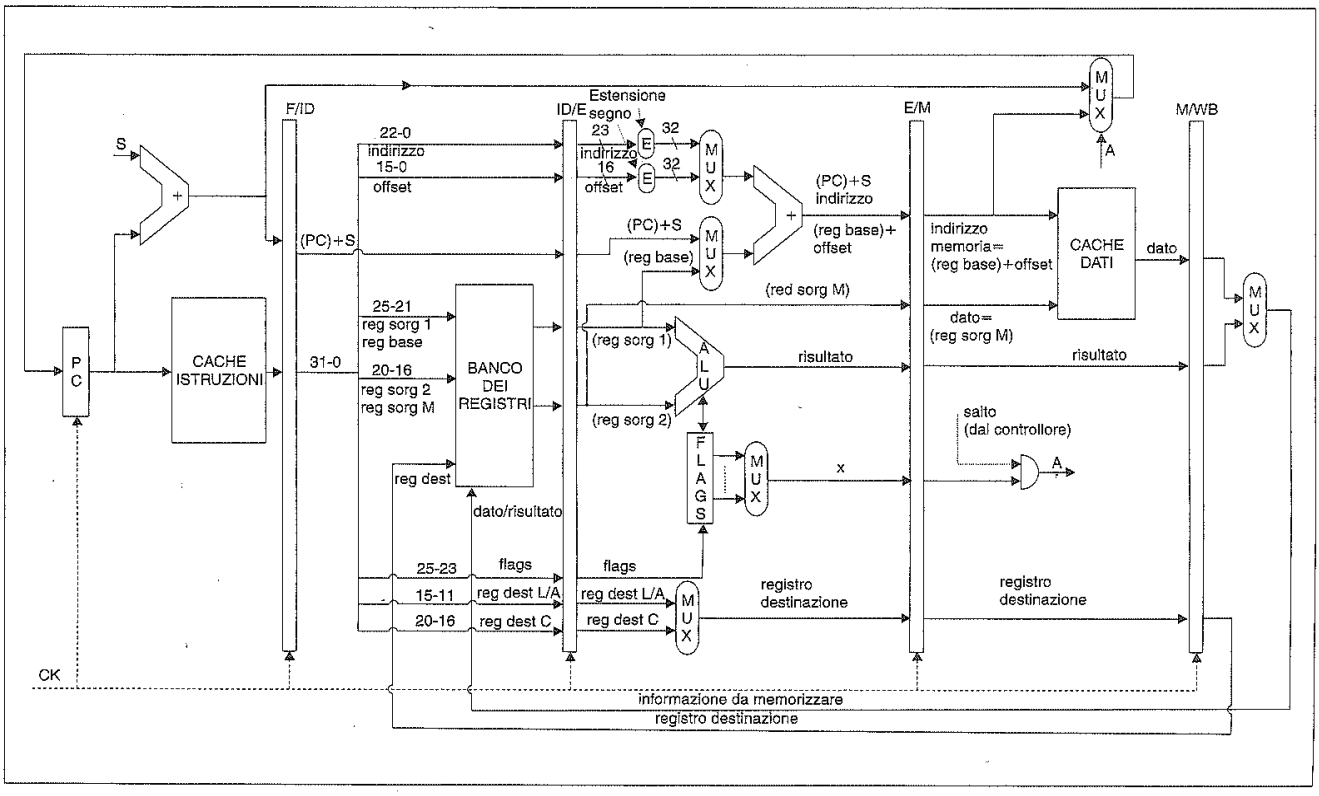


Fig. 7.9 - Data path completo dell'architettura pipeline.

7.2.2.3. Schema del controllore

Nella struttura pipeline sono presenti tanti stadi per quante sono le fasi previste per l'esecuzione di una istruzione. Nel caso specifico di 5 stadi, così come schematizzato in figura 7.9, le attività nel primo e nel secondo stadio sono indipendenti dal tipo di istruzione, cosa che non avviene negli altri tre stadi. Come già detto nel secondo stadio, dopo la fase di fetch, le istruzioni vengono decodificate, ciò viene fatto mandando il codice operativo (bit 31-26) al controllore. Normalmente il controllore è una ROM, che indirizzata dal codice operativo emette i comandi (microrordini) che devono essere eseguiti negli stadi successivi. Peculiarità della struttura pipeline è che ad ogni colpo di clock cambia l'istruzione nel secondo stadio (come in tutti gli altri stadi) e quindi cambia il codice operativo utilizzato dalla ROM per generare i segnali di controllo. Pertanto è necessario, ad ogni colpo di clock, dal secondo stadio non solo trasmettere gli operandi al terzo stadio, ma anche i segnali di controllo relativi a quegli elementi del sottosistema di calcolo necessari per completare la manipolazione degli operandi nel relativo stadio. In Figura 7.10 è schematizzata la tecnica per trasmettere i segnali di controllo da uno stadio al successivo. Per memorizzarli si usa lo stesso registro di pipeline utilizzato per gli operandi. Naturalmente, come si vede dalla Figura 7.10, i segnali di controllo necessari in uno stadio non vengono trasmessi allo stadio successivo, per ogni stadio vi sono tanti segnali di controllo per quanti sono i segnali di controllo necessari per far lavorare correttamente i dispositivi in quello stadio che devono essere pilotati dal controllore. In particolar modo:

- nello stadio di EXECUTE sono previsti:
  - M1 per pilotare il primo multiplexer nella selezione del primo operando da mandare all'addizionatore tra l'indirizzo di memoria e l'Offset;
  - M2 per pilotare il secondo multiplexer nella selezione del secondo operando da mandare all'addizionatore tra il valore di (PC)+ e (Base_reg):;
  - M3 per selezionare l'appropriato registro destinazione tra Dest_reg_L/A e Dest_reg_C
  - ALU_OPCODE per comandare l'ALU ad effettuare l'operazione appropriata in funzione del tipo di istruzione da eseguire.
- nello stadio MEMORY sono previsti:
  - DCR per abilitare la lettura del dato dalla memoria
  - DCW per abilitare la scrittura del dato in memoria
  - JMP per consentire il salto di sequenza nell'esecuzione di un programma
- nello stadio di WRITE BACK sono previsti:
  - MWB per pilotare il multiplexer nella selezione tra dato e risultato
  - RW per abilitare la scrittura nel banco dei registri di quanto selezionato dal multiplexer.

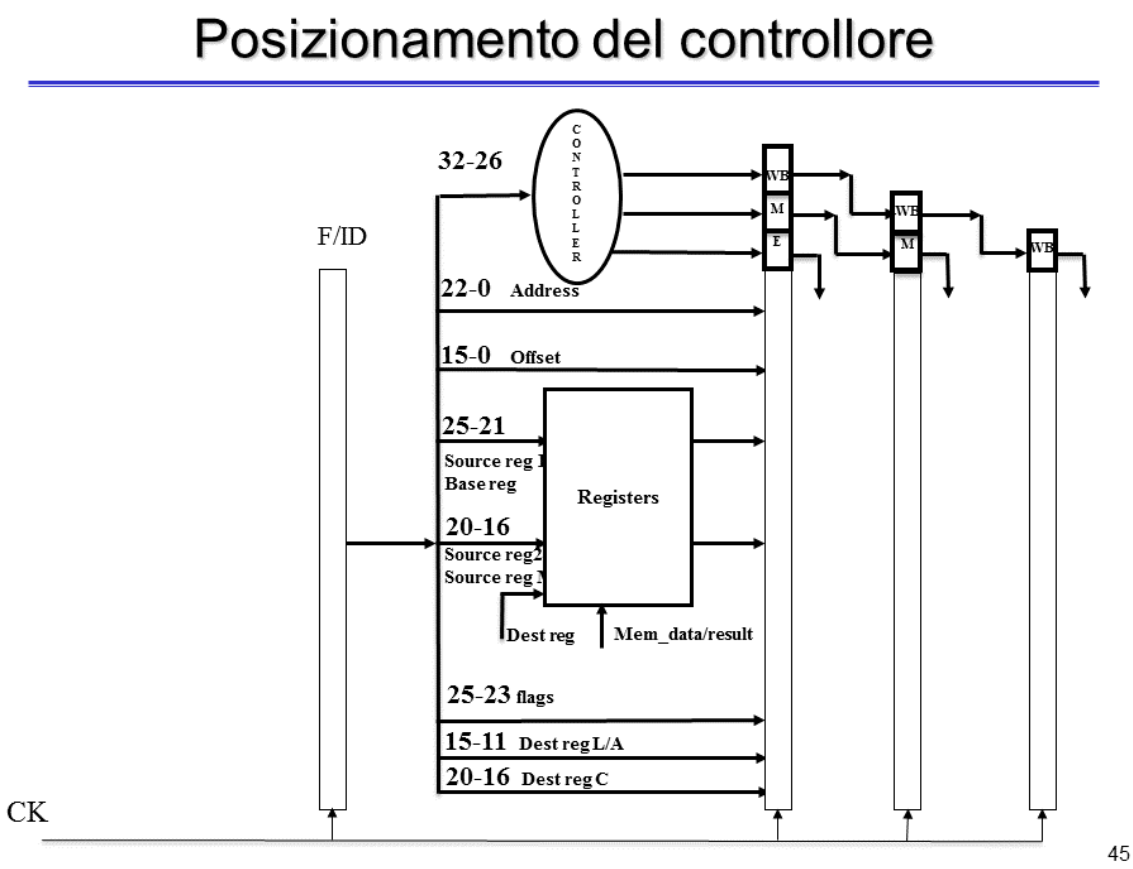


Figura 7.10 Posizionamento del SCO

Nella Figura 7.11 è riportata una tabella in cui è riportata una possibile codifica dei comandi che devono essere generati in funzione del tipo di istruzione da eseguire. Per esempio nel caso esecuzione di una addizione è necessario che generare il codice per l'addizione da mandare all'ALU, che quindi deve essere disponibile nello stadio di EXECUTE; non si deve abilitare alcun componente nello stadio di MEMORY, e quindi si metterà a zero il bit di controllo relativo alla scrittura della memoria e a quello necessario per verificare se si è creata una condizione per un salto, mentre il segnale di controllo relativo alla lettura della memoria può essere messo sia ad uno che a zero, dato che una eventuale lettura non avrebbe alcuna conseguenza (e quindi è stato messo come dcc); infine nello stadio di WRITEBACK viene messo ad uno il segnale di controllo MRW dato che si deve selezionare l'operando proveniente dall'ALU, operando che poi viene messo nel bamco dei registri dato che il segnale di controllo RW è messo pari ad uno.

I segnali di controllo (3/3)

		EXECUTE						MEMORY			WRITE BACK	
Instruction	OPCODE	M1	M2	M3	OP_3	$0P_2$	$0P_1$	DCW	DCR	JMP	MWB	RW
ADD	000001	-	-	0	0	0	1	0	-	0	1	1
SUB	000010	-	-	0	0	1	0	0	-	0	1	1
OR	000011	-	-	0	0	1	1	0	-	0	1	1
AND	000100	-	-	0	1	0	0	0	-	0	1	1
NOT	000101	-	-	0	1	0	1	0	-	0	1	1
LOAD	000110	1	1	1	-	-	-	0	1	0	0	1
STORE	000111	1	1	-	-	-	-	1	0	0	-	0
JMPC	001000	0	0	-	-	-	-	0	0	1	-	0
NOP	000000	-	-	-	0	0	0	0	-	0	-	0

48

Fig. 7.11 – Comandi del controllore.

7.2.3. – Conflitti e rischi: identificazione e risoluzione

L'esecuzione di un programma in un processore con architettura pipeline può essere schematizzato come in figura 7.12. Dalla figura si vede il sovrapporsi delle varie fasi e che, in assenza di interferenze tra le attività di ogni singola istruzione, dopo il transitorio è completata un'istruzione ad ogni periodo di clock. Come già detto il periodo del clock deve essere dimensionato per completare le attività dello stadio più lento. Inoltre, è da notare che alcune delle istruzioni potrebbero essere completate in un numero di passi inferiore a cinque, ma l'organizzazione della struttura pipeline è tale che, se anche alcune fasi sono inutili, ogni singola istruzione è eseguita in cinque passi. Ed è proprio per questo che nell'esempio di figura 7.12 non sono stati rappresentati i tipi di istruzione.

Fig. 7.12 - Schema di esecuzione di un programma senza conflitti.

Però lo schema rappresentato in figura 7.12 rappresenta l'esecuzione di un programma ideale in cui l'esecuzione di un'istruzione in uno stadio non influenza quelle delle istruzioni negli stati precedenti (istruzioni eseguite successivamente). Le interferenze o criticità (***hazard***) che si possono verificare possono essere:

- criticità strutturali;
- criticità sui dati;
- criticità sul controllo.

Si ha una criticità **strutturale** quando due o più istruzioni in stadi differenti tentano di usare contemporaneamente la stessa risorsa (per esempio una istruzione nella fase di decode ed una istruzione in write back che vorrebbero leggere e scrivere, rispettivamente, lo stesso registro; inoltre si potrebbero avere conflitti strutturali se invece di una cache per le istruzioni ed una per i dati ce ne fosse una sola per entrambi i tipi di informazioni).

Si ha una criticità **sui dati** quando si tenta di usare un dato prima che sia disponibile. Tale dato potrebbe essere o un dato memorizzato in memoria, ancora da leggere, o un dato ancora da produrre o prodotto, ma non memorizzato, dall'ALU. Il primo tipo di conflitto si chiama ***load use***, mentre il secondo ***define use***.

Si ha una criticità **sul controllo** quando si tenta di prendere una decisione sulla prossima istruzione da eseguire prima che la condizione sia valutata.

Per quanto riguarda le ***criticità strutturali*** si tenta di eliminarle ottimizzando la progettazione dell'hardware. In particolar modo, come già detto, nelle architetture pipeline si tiene distinta la cache istruzioni da quelle dei dati, proprio per evitare conflitti nell'accesso in memoria, inoltre il conflitto sul banco dei registri può essere eliminato progettando lo stesso in modo che la scrittura

di un dato sul banco dei registri venga fatta nel primo semiperiodo del clock di sincronizzazione della pipeline, mentre la lettura venga fatta nel secondo semiperiodo.

Define use

Un esempio di una criticità define use si ha nella figura 7.13, dove si può vedere un segmento di programma in cui l'istruzione "sub R3, R4, R5" necessita del risultato dell'istruzione "add R1, R2, R3", in quanto la seconda istruzione deve utilizzare il contenuto del registro R3 che deve essere aggiornato dalla prima istruzione.

```
add R1, R2, R3
sub R3, R4, R5
sub R3, R6, R7
add R3, R8, R9
sub R3, R10, R11
```

Fig. 7.13 - Esempio di frammento di un programma con conflitti di tipo define use.

Per evidenziare graficamente i conflitti si usa il data path stilizzato precedentemente, come quello rappresentato in figura 7.14, in cui si riportano le attività relative all'esecuzione del segmento di programma della figura precedente: la prima istruzione scrive nel registro R3, mentre le istruzioni successive leggono il contento dello stesso registro. L'ordine di esecuzione delle istruzioni del programma è rappresentato dall'alto verso il basso, in alto è rappresentato il trascorrere del tempo (in cicli di clock) ed una possibile evoluzione del contenuto del registro R3. Le dipendenze sono evidenziate con delle frecce in grassetto che mettono in correlazione la scrittura e la lettura del registro R3. Ipotizzando che la prima istruzione inizi ad essere eseguita nel primo ciclo di clock, si ha che modifica il contenuto del registro R3 alla fine del quinto colpo di clock, mentre la seconda, la terza, la quarta e la quinta istruzione leggono, rispettivamente, il contenuto del registro R3 nel terzo, quarto, quinto e sesto ciclo di clock. Dalla figura si può vedere che solo la quinta istruzione legge il corretto contenuto del registro R3, mentre le altre no.

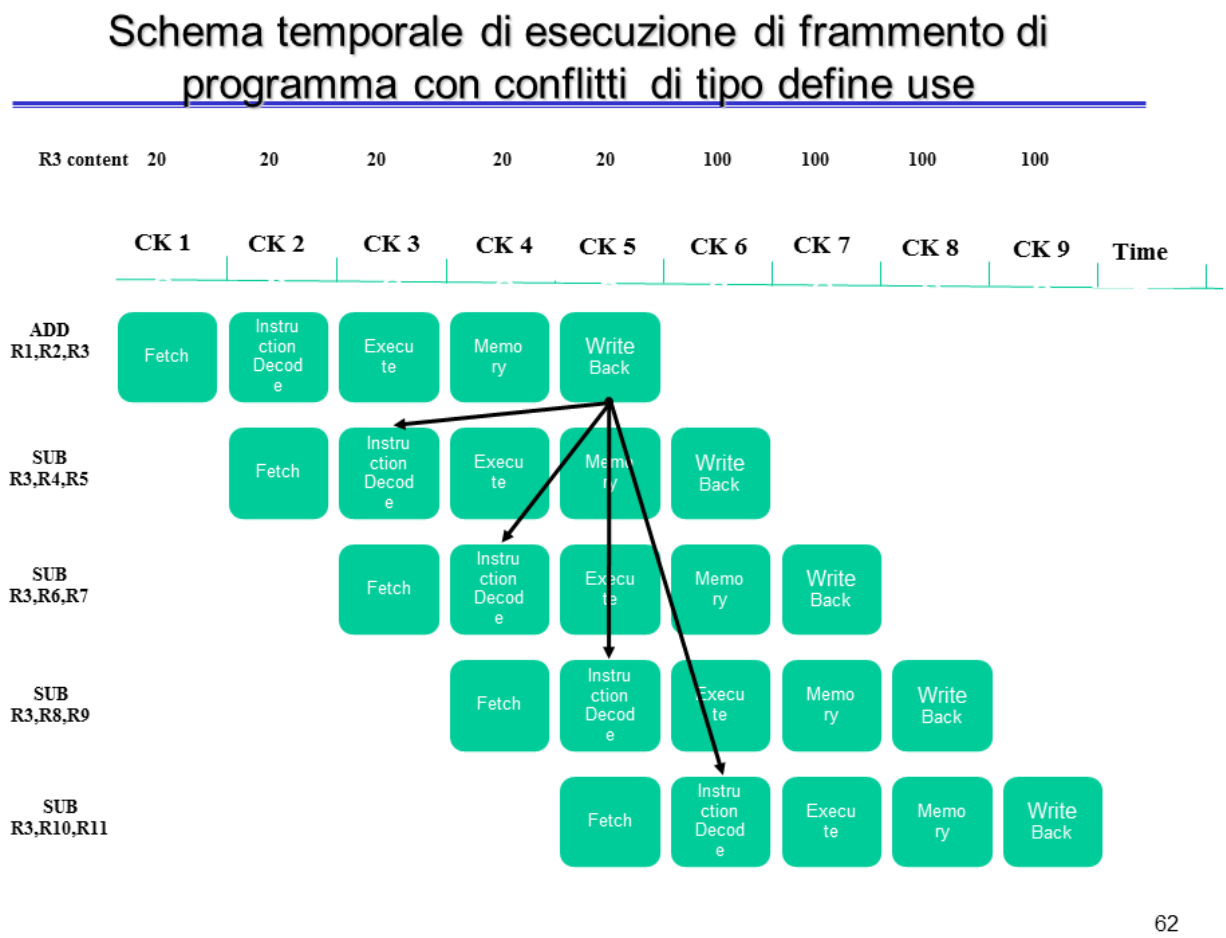


Fig. 7.14 - Schema temporale di frammento di un programma con conflitti di tipo define use

Per evitare i conflitti sui dati si potrebbe ritardare l'esecuzione delle istruzioni che hanno un conflitto con le istruzioni precedenti. Questo può essere fatto via software inserendo delle istruzioni non operative ("nop") o modificando il codice, cambiando l'ordine dell'esecuzione delle istruzioni (se possibile); oppure via hardware, inserendo delle "bolle" (annullamento delle attività degli stadi successivi, questo è fatto inserendo al posto dei microrordini generati dal controllore dei microrordini che non hanno effetto sulle unità funzionali del data path) o utilizzando la cosiddetta propagazione in avanti.
Di seguito per ogni opzione faremo vedere la relativa soluzione in dettaglio.

L'inserimento delle istruzioni di tipo "nop" può essere fatto dal programmatore, se lavora a livello assembler, o dal compilatore. Per verificare se esiste un conflitto è necessario confrontare i codici operativi delle istruzioni in sequenza e gli identificativi dei registri interessati.

Per esempio utilizzando il segmento di programma schematizzato in figura 7.13, si nota che se si inseriscono tre nop subito dopo la prima istruzione, come schematizzato nella figura successiva (7.15), si evitano conflitti, in quanto , come schematizzato in figura 7.16, il banco dei registri viene aggiornato nel passo write back della prima istruzione prima che la sub R3, R4, R5 legga il contenuto del registro R3. Naturalmente nel caso che il banco dei registri fosse progettato in modo da evitare conflitti strutturali allora di nop se ne sarebbero potute aggiungere solo due.

```
add R1, R2, R3
```

```
nop
nop
nop
sub R3, R4, R5
sub R3, R6, R7
add R3, R8, R9
sub R3, R10, R11
```

Fig. 7.15 - Esempio di frammento di un programma con nop

In figura 5.52 (delle fotocopie) si può vedere la rappresentazione grafica delle dipendenze sul registro R3, come è evidente le linee di dipendenza hanno tutte lo stesso verso da sinistra verso destra, indice di assenza di criticità.

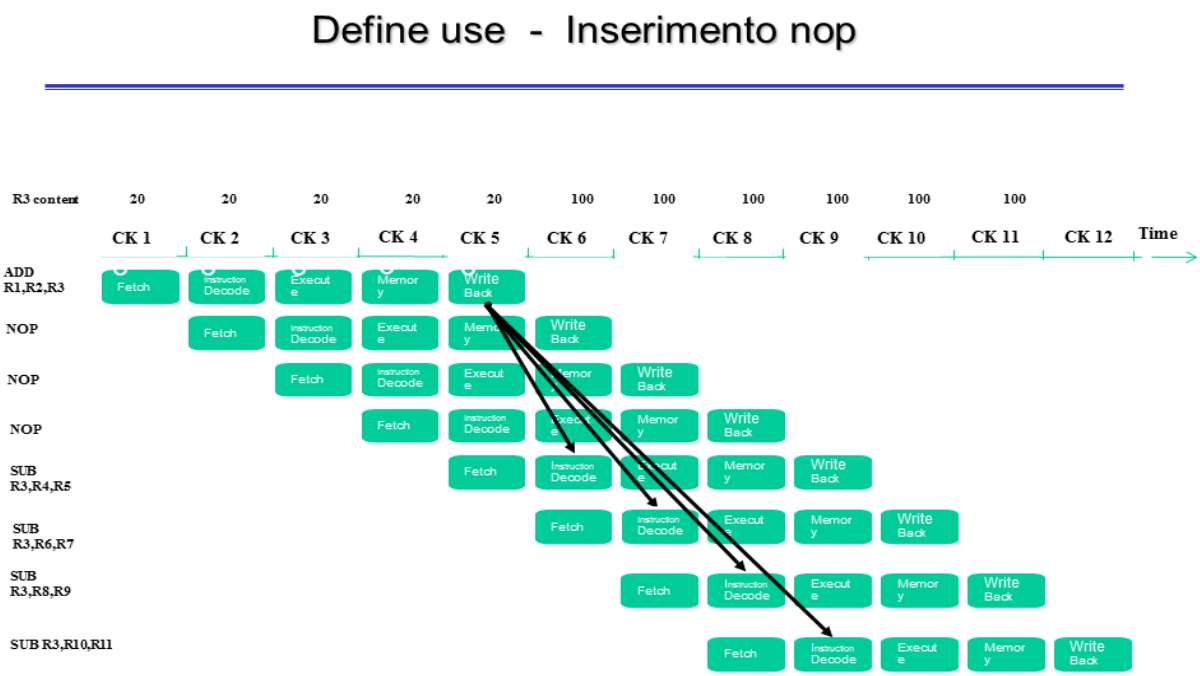


Fig. 7.16 Schema temporale di frammento di un programma senza conflitti di tipo define use per l'inserimento di nop

Naturalmente con questa soluzione si allungano i tempi di esecuzione del programma stesso. Una soluzione per evitare ciò è possibile tramite il riordino delle istruzioni, purché la semantica del programma rimanga la stessa. Nel segmento di programma schematizzato in figura 7.13 tutte le istruzioni dopo la prima utilizzano il registro R3 come registro sorgente, pertanto anche cambiando l'ordine delle istruzioni sarebbe sempre necessario inserire delle nop e comunque la prima deve sicuramente precedere tutte le altre, invece nel segmento di programma, rappresentato in figura 7.17.

```
add R1, R2, R3
sub R3, R6, R7
sub R8, R9, R10
add R11, R12, R13
sub R14, R15, R15
```

Fig. 7.17 – Secondo esempio di frammento di un programma con conflitti di tipo define use

Dal segmento di programma si può notare che solo la seconda istruzione usa il contenuto del registro R3 modificato dalla prima, mentre le altre istruzioni non hanno alcune dipendenza con utte e altre, pertanto modificando il codice come rappresentato in figura 7.18, si può notare che solo cambiando posto all'istruzione sub R3, R6, R7 non esistono più conflitti e quindi il programma può essere eseguito senza penalità temporali e senza modificarne la semantica.

```
add R1, R2, R3
sub R8, R9, R10
add R11, R12, R13
sub R14, R15, R15
sub R3, R6, R7
```

Fig. 7.18 – Codice del secondo esempio di frammento di un programma con riordino per evitare conflitti

Come detto, in alternativa alle soluzioni software si può intervenire modificando l'hardware del processore, in questo caso il programmatore, il compilatore o l'assemblatore non si dovranno preoccupare della gestione delle criticità di tipo define use.

La prima soluzione si basa sull'emulazione dell'inserimento delle istruzioni nop, in questo caso il si dovrà inserire una unità hardware, detta anche rivelatore di conflitti, che nel caso di conflitto dovrà propagare negli stadi successisi alla decode i microrordini relativi alla nop al posto di quelle relativi alla istruzione corrente. Inoltre si dovranno bloccare le istruzioni negli stadi di fetch e decode, che potranno propagarsi nella pipeline solo dopo che sarà risolto il conflitto.

Per poter far ciò si introduce l'hazard unit, che potrà identificare la presenza di un conflitto verificando se una delle due istruzioni (ipotizzando che sia stato risolto il conflitto strutturale sui registri) che sono negli stadi EX o MEM scriverà nel banco dei registri nella sua fase di write-back e se il relativo registro destinazione coincida con uno dei registri sorgenti dell'istruzione nello stato DEC. Per verificare che una istruzione scriverà nel banco dei registri è sufficiente controllare se il segnale di controllo RW (segnale che abilita la scrittura sul banco dei registri) sia pari ad 1.

In caso di conflitto, come detto, l'HAZARD UNIT dovrà:

- propagare nella pipeline i microrordini (segnali di controllo) relativi alla NOP e non all'istruzione corrente nella fase di decode, e questo lo farà pilotando un mux a prendere la codifica dei microrordini relativi alla NOP invece che i microrordini relativi all'istruzione corrente;
- bloccare l'istruzione nella fase di decode fino a che non sarà risolto il conflitto; questo può essere fatto disabilitando il registro di interfaccia IF/ID a memorizzare il codice dell'istruzione

nella fase di DECODE, ciò è fattibile mettendo in AND il clock della pipeline con un segnale di abilitazione che può generarsi dall'HAZARD UNIT (p.e se pari a 0 non ci deve essere memorizzazione);

- bloccare l'istruzione nella fase di fetch fino a che non sarà risolto il conflitto; questo può essere fatto disabilitando la scrittura del PC con il nuovo valore calcolato dall'addizionatore, ciò è fattibile mettendo in AND il clock con un segnale di abilitazione che può generarsi dall'HAZARD UNIT (p.e se pari a 0 non ci deve essere aggiornamento)

Per evitare il ritardo che ne consegue, dall'esempio di figura 7.14, si può osservare che alla fine dello stadio Execute della prima istruzione il valore corretto del registro destinazione (quello che poi sarà scritto nello stadio di Write Back) è già disponibile nel registro di pipeline E/M e quindi potrebbe essere utilizzato al posto del contenuto non aggiornato del registro R3 letto dalla seconda istruzione nello stadio ID. E' proprio questa osservazione che ha portato alla definizione dell'unità hardware "propagazione in avanti" o FORWARDING UNIT (anch'essa di tipo combinatoria). Unità che, come l'unità di rivelazione conflitti, per le proprie attività necessita di confrontare i codici operativi (o microrordini) delle istruzioni con gli identificativi dei relativi registri.

Compito di tale unità, in caso di rivelazione di conflitti, è quello di presentare come operando all'ALU non il valore errato letto dall'istruzione, ma quello nuovo elaborato dall'istruzione che lo avrebbe dovuto memorizzare nel banco dei registri nella fase di write back, come schematizzato in figura 7.19.

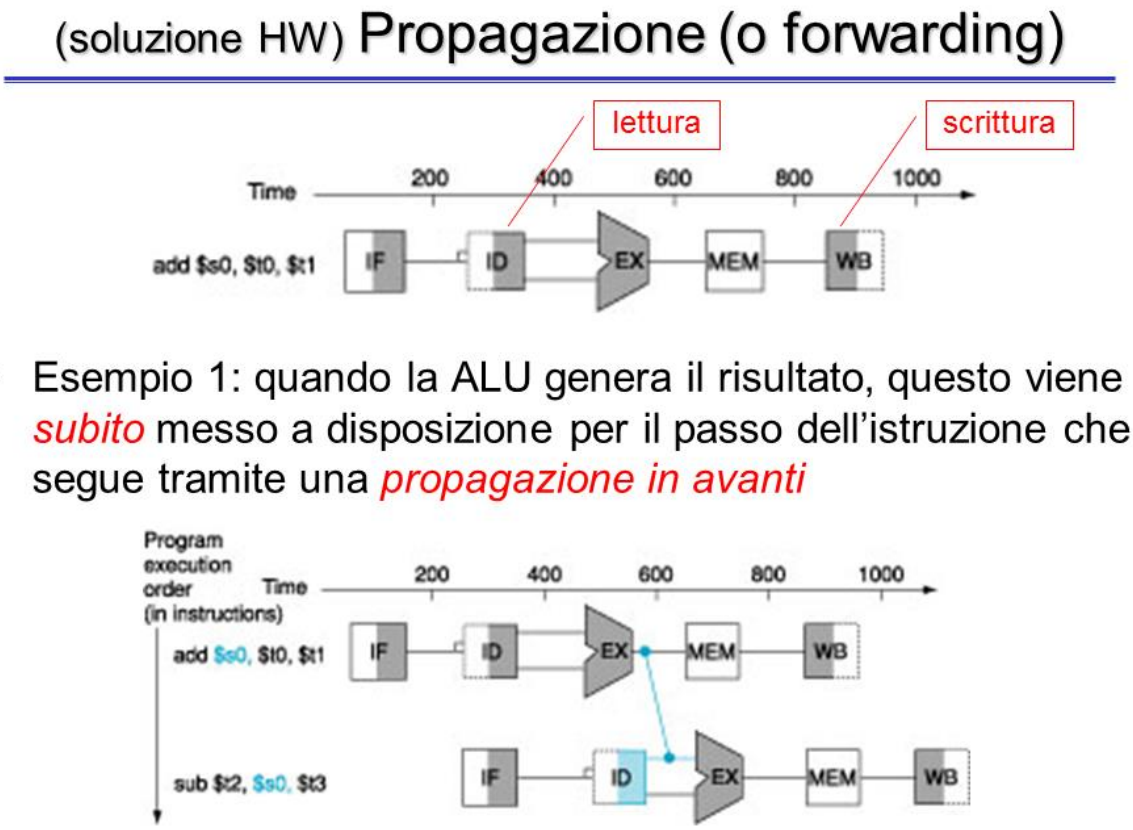


Figura 7.19 – Effetto dell'introduzione della forwarding unit

Load use

Un esempio di una criticità load use si ha nella figura 5.52a, dove si può vedere un segmento di programma in cui l'istruzione "sub R3, R4, R5" necessita del risultato dell'istruzione " load R3, 122(R1)", che la prima istruzione deve memorizzare nel registro R3 dopo aver acceduto alla memoria alla locazione di memoria data dalla somma di 122 con il contenuto del registro R1.

```
load R3, 122(R1)
sub R3, R5, R6
sub R3, R6, R6
add R3, R7, R8
sub R3, R9, R10
```

Fig. 7.20 - Esempio di frammento di un programma con conflitti di tipo load use.

Anche in questo caso per evidenziare graficamente i conflitti si usa il data path stilizzato precedentemente, come quello rappresentato in figura 7.21, in cui si riportano le attività relative all'esecuzione del segmento di programma della figura precedente: la prima istruzione scrive nel registro R3, mentre le istruzioni successive leggono il contento dello stesso registro. L'ordine di esecuzione delle istruzioni del programma è rappresentato dall'alto verso il basso, in alto è rappresentato il trascorrere del tempo (in cicli di clock) ed una possibile evoluzione del contenuto del registro R3. Le dipendenze sono evidenziate con delle frecce in grassetto che mettono in correlazione la scrittura e la lettura del registro R3. Ipotizzando che la prima istruzione inizi ad essere eseguita nel primo ciclo di clock, si ha che modifica il contenuto del registro R3 alla fine del quinto colpo di clock, mentre la seconda, la terza, la quarta e la quinta istruzione leggono, rispettivamente, il contenuto del registro R3 nel terzo, quarto, quinto e sesto ciclo di clock. Dalla figura si può vedere che solo la quinta istruzione legge il corretto contenuto del registro R3, mentre le altre no.

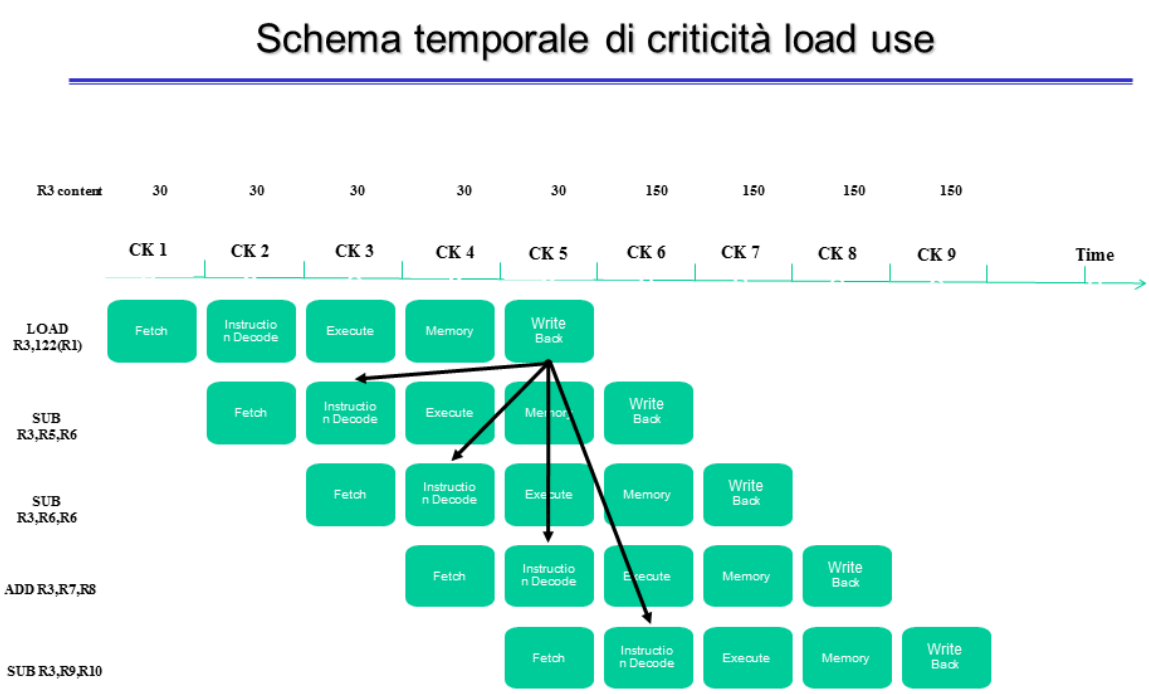


Fig. 7.21- Schema temporale di frammento di un programma con conflitti di tipo load use

Anche in questo caso, per evitare i conflitti sui dati, si potrebbe ritardare l'esecuzione delle istruzioni che hanno un conflitto con le istruzioni precedenti. Questo può essere fatto via software modificando il codice, cambiando l'ordine dell'esecuzione delle istruzioni (se possibile) o inserendo delle istruzioni non operative ("nop"), come schematizzato in figura 7.22, oppure via hardware, inserendo delle "bolle" (annullamento delle attività degli stadi successivi, questo è fatto inserendo al posto dei microrordini generati dal controllore dei microrordini che non hanno effetto sulle unità funzionali del data path) o utilizzando la propagazione in avanti. E' da notare che in questo caso non è possibile utilizzare la propagazione in avanti quando esiste un conflitto di tipo load use tra due istruzioni successive in quanto la seconda utilizzerà il dato "letto erroneamente" nell'ALU contemporaneamente alla lettura in memoria del dato corretto dall'istruzione che la precede. Quindi in questo caso sarà necessario inserire almeno una nop o una bolla tra le due istruzioni.

```
load R3, 122(R1)
nop
nop
nop
sub R3, R5, R6
sub R3, R6, R6
add R3, R7, R8
sub R3, R9, R10
```

Fig. 7.22 - Esempio di frammento di un programma senza conflitti di tipo load use con l'uso di nop

Criticità sul controllo

In figura 7.23 è riportata l'esecuzione di un frammento di programma in cui esiste un hazard sul controllo con le attività del data path stilizzato.

Fig. 7.23 – Esecuzione di un frammento di programma con hazard sul controllo

Dalla figura si può notare che solo nello stadio Memory è presa la decisione se effettuare o no il salto. Nel caso in cui non si deve saltare è corretto continuare il processamento delle istruzioni negli stadi precedenti, mentre nel caso opposto, schematizzato in figura, vi è un evidente errore. Per evitare quest'inconveniente si possono seguire due approcci: il primo ***pessimistico*** (ci si mette nella condizione che bisogna effettuare sempre il salto), l'altro ***ottimistico*** (ci si mette nella condizione che non si debba eseguire il salto). Nel primo caso si sospende l'esecuzione delle istruzioni successive, fino a che non si è sicuri se il salto deve essere effettuato o no. Ciò può essere fatto via software inserendo nel codice del programma delle istruzioni di tipo "nop", come schematzzato in figura 7.24, oppure via hardware inserendo delle "bolle". Nel secondo caso invece si può intervenire solo a livello hardware. In questo caso la difficoltà risiede nell'annullare gli effetti del processamento delle istruzioni, che non dovevano essere eseguite; in particolare facendo riferimento al set delle istruzioni della figura 7.23 sarà necessario annullare l'effetto delle istruzioni:

- add R1,R2, R3 nello stadio EX, che potrebbe aver modificato il valore del CARRY
- sub R$,R5,R6 nello stadio ID,
- load R7, 32(R8) nello stadio IF.

Però mentre la prima potrebbe aver modificato il valore del CARRY e degli altri flip/flop di stato, la seconda e la terza non modificano nulla, tranne che eventualmente propagare i segnali di controllo generati dallo SCA (relativi alla sub) o il proprio OPCODE (relativi alla load). Pertanto mentre per la prima è necessario ripristinare il valore dello STATUS REGISTER al valore precedente, e quindi c'è la necessità di avere una doppio banco di flags di stato, per la seconda e la terza è sufficiente annullare la loro presenza resettando i contenuti dei registri di interfaccia IF/ID e ID/EX.

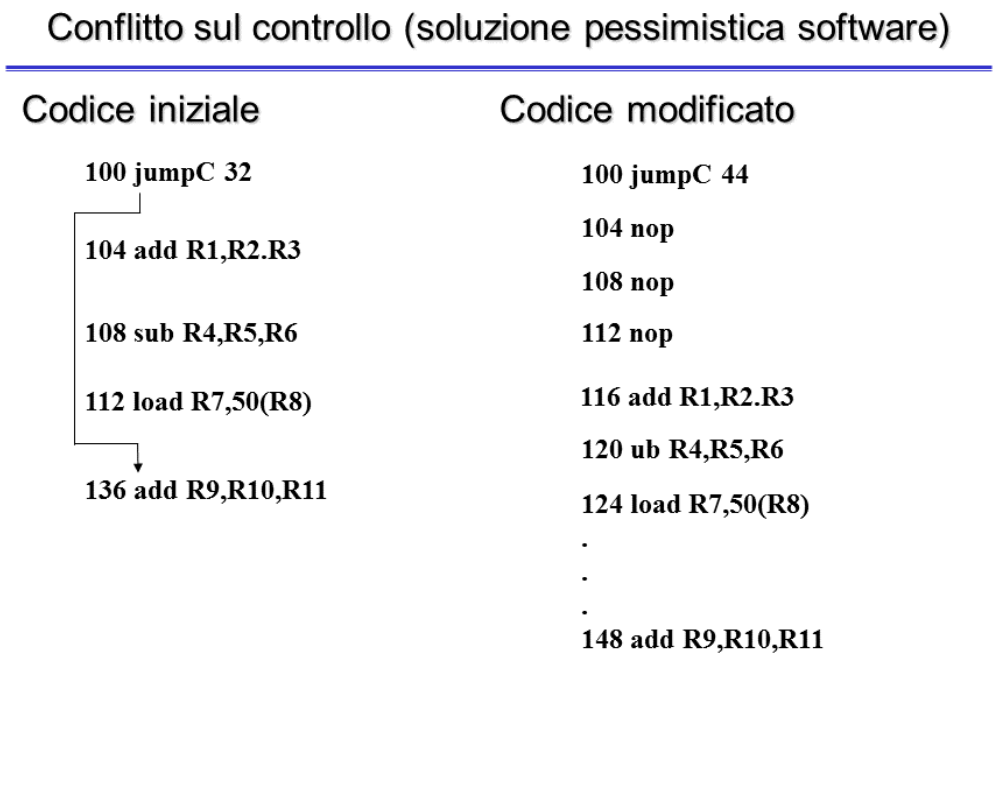


Fig. 7.24 – Soluzione pessimistica per evitare un hazard sul controllo usando delle nop

E' da notare che spesso i salti si riferiscono a dei loop, naturalmente se il numero delle iterazioni è molto alto è molto conveniente effettuare il salto, inoltre in altri contesti si potrebbe verificare che le condizioni di salto dipendono dalle condizioni valutate da poco. Per questo motivo si sono proposte soluzioni che tendono a predire se una condizione sia vera o meno. E' stato sperimentato, utilizzando dei benchmark accettati in letteratura, quali ad esempio SPEC, che alcune soluzioni riescono a predire in modo corretto più del 90% dei casi. I produttori di processori hanno proposto sia soluzioni locali che globali, dove per locali si intendono soluzioni che utilizzano informazioni di comportamento "vicine" all'indirizzo dell'istruzione interessata alla predizione, mentre per globali si intendono soluzioni che utilizzano informazioni di comportamento che interessano tutto il programma in esecuzione.

Un esempio di soluzione locale è la cosiddetta bimodale, in cui si utilizza una tabella di contatori, a cui si accede utilizzando i bit meno significativi dell'indirizzo delle istruzioni di salto, che deve verificare la condizione per il salto. Tali contatori, di due bit, possono assumere quattro valori:
- 00, fortemente non scelto;
- 01, debolmente non scelto;
- 10, debolmente scelto;
- 11, fortemente scelto.

Ad ogni predizione di salto, se la condizione è verificata si passa da un valore a quello crescente (e quindi se si è già nello stato 11 si rimane in tale stato), mentre se la condizione non è verificata si passa al valore inferiore (anche in questo caso se si è già nello stato 00 si rimane in tale stato).
Da notare che i contatori potrebbero interessare più istruzioni di salto se aventi gli stessi bit meno significativi. Questo tipo di soluzione locale è particolarmente adatta per la predizione dei cicli, naturalmente potrebbe fallire nel primo salto e nell'uscita del loop.

7.2.4. – Altri problemi delle architetture pipeline

Nel data path del processore pipeline presentato si è prevista una sola ALU, senza entrare nel merito se questa fosse in grado di gestire numeri interi e/o reali. Normalmente per l'esecuzione di operazioni tra numeri reali si utilizza un'unità aritmetica specializzata: la Float Point Unit - FPU, che avendo una certa complessità circuitale, ha dei tempi di esecuzione molto maggiori rispetto a quella delle unità aritmetiche per numeri interi. Per far convivere questi due tipi di unità aritmetiche si possono intraprendere diverse alternative:
- si impone che le attività di tutti gli stadi vengano eseguite con una durata sufficiente ad eseguire le operazioni floating point, con conseguente rallentamento di tutte le fasi di esecuzione delle istruzioni;
- si prevedono più stadi di Execute, che vengono sfruttati appieno solo dalle istruzioni che prevedono operazioni floating point, con conseguente rallentamento degli altri tipi di istruzione;
- si prevede una FPU superveloce, che possa eseguire le operazioni con una velocità confrontabile con le altre unità funzionali, con conseguente incremento del costo del processore e di dissipazione del calore;
- si prevede di eseguire le istruzioni con operandi floating point su uno stadio di Execute distinto da quello delle altre istruzioni (in questo caso normalmente si prevede che per gli operandi si utilizzino registri distinti da quelli degli altri operandi), con conseguente possibilità che non sempre le istruzioni sono completate nello stesso ordine con cui vengono iniziate.

Stessi problemi si sollevano nel caso in cui si volesse implementare all'interno della pipeline una unità specializzata nelle moltiplicazioni e divisioni. Infatti anche in questo caso, come nel caso di inserimento dell'unità floating point.

Nel seguito faremo vedere come sia possibile far convivere nello stadio Execute sia la normale ALU che una FPU e una unità moltiplicatore/divisore.

Altro problema sorge nella gestione delle eccezioni o delle interruzioni. Come ribadito più volte, peculiarità delle architetture pipeline è che nel processore vengono eseguite contemporaneamente tante istruzioni per quanti sono gli stadi. Pertanto, nel caso d'insorgenza di un'eccezione o dell'arrivo di un'interruzione non si può seguire la politica utilizzata per i processori a ciclo multiplo, in cui prima di eseguire il programma di gestione dell'evento asincrono si completa l'istruzione corrente. In questo caso all'arrivo di un evento asincrono, prima

dell'esecuzione del relativo programma di gestione, è necessario arrestare l'immissione di nuove istruzioni e completare le istruzioni già presenti nella struttura pipeline oppure si completano solo quelle negli stadi di Execute, Memory e Write-Back, annullando la presenza delle istruzioni nei primi due stadi del pipeline. Naturalmente dopo aver eseguito il programma di gestione dell'evento asincrono è necessario ripristinare il contesto del processo interrotto, per questo motivo è necessario prevedere almeno il salvataggio del registro di stato e dell'indirizzo dell'ultima istruzione eseguita prima del programma di gestione dell'evento.

Nel caso di gestione delle interruzioni, è evidente che essendoci più istruzioni ancora in esecuzione prima dell'attivazione del programma di gestione si potrebbero creare problemi di consistenza rispetto alla gestione delle interruzioni nel caso di esecuzione dello stesso programma in un processore multiciclo, in cui viene eseguita una istruzione alla volta. Per evidenziare questa possibilità di parla di gestione ***precisa*** delle interruzioni nel caso in cui terminata la gestione del programma di gestione dell'evento il comportamento del processore è consistente con quello di un processore sequenziale, mentre viene detta ***imprecisa*** se non viene garantita tale proprietà.

7.2.5. – Architetture RISC avanzate

L'esecuzione di segmenti di programmi in cui esiste poca interferenza tra le istruzioni può essere ulteriormente migliorata aumentando il numero di stadi della struttura in pipeline o mettendo più strutture pipeline in parallelo. Le architetture del primo tipo sono identificate come *superpipeline*, mentre le seconde come *pipeline superscalari*.

Le architetture superpipeline nascono dalla considerazione che il throughput delle strutture pipeline è proporzionale alla frequenza del clock, e quindi avendo un numero maggiore di stadi veloci è possibile aumentare la frequenza di funzionamento, mentre le superscalari dal fatto che avendo più processori in parallelo si possono eseguire più istruzioni contemporaneamente. Ovviamente, si possono avere processori che usano i due approcci simultaneamente.

Per quanto riguarda le strutture superpipeline bisogna identificare il numero ottimale degli stadi considerando il tempo di esecuzione necessario per ogni singola fase e il numero medio di istruzioni che non interferiscono tra loro per i programmi che si prevede saranno eseguiti dal processore. Per esempio se si dispone di memorie cache e banchi di registri con un tempo di accesso di 10 nesc., addizionatori con tempo di calcolo di 10 nsec. e ALU con tempo di calcolo di 30 nsec., trascurando tutti i tempi delle altre unità funzionali (registri di pipeline, mutiplexer, etc.) il periodo di clock della struttura pipeline a cinque stadi progettata precedentemente dovrebbe essere almeno di 30 nsec. Di conseguenza una singola istruzione viene eseguita, avendo un clock di 30 nsec., in 150 nsec, mentre il rate di uscita è pari a 33,3 MIPS. Se invece, fosse possibile suddividere l'ALU in tre stadi, ognuno con un tempo di calcolo di 10 nsec. si avrebbe un processore con 7 stadi (come schematizzato in figura 6.21), in grado di eseguire un'istruzione in 70 nsec. e con un rate di uscita pari a 100 MIPS. Rispetto alle architetture pipeline questa soluzione presenta una maggiore complessità circuitale e nel fatto che, essendoci più istruzioni eseguite contemporaneamente, è più alta la probabilità d'interferenza tra le stesse; quindi, questa soluzione potrebbe essere conveniente solo se il numero medio di conflitti e di rischi non inficia le sue potenzialità.

Fetch	Instruction Decode	Execute	Execute	Execute	Memory	Write Back

Esempio dell'organizzazione logica di una struttura superpipeline

Fetch	Instruction Decode	Execute	Memory	Write Back
Fetch	Instruction Decode	Execute	Memory	Write Back

Fig. 7.25 Esempi di organizzazioni logiche di strutture superpipeline e superscalare.

Anche per quanto riguarda le strutture pipeline superscalari bisogna identificare il numero ottimale delle strutture pipeline da porre in parallelo, considerando il tempo di esecuzione di ogni singola istruzione e il numero medio di istruzioni che non interferiscono tra loro per i programmi che si prevede saranno eseguiti dal processore. Normalmente le architetture superscalari prevedono due o quattro strutture pipeline in parallelo (nella figura 7.25 è rappresentata un'organizzazione logica per un'architettura a due vie). Anche in questo caso, rispetto alle architetture pipeline, questa soluzione presenta una maggiore complessità circuitale e nel fatto che, essendoci più istruzioni eseguite contemporaneamente, è più alta la probabilità d'interferenza tra le stesse. Quindi, la soluzione potrebbe essere conveniente solo se il numero medio di conflitti e di rischi non inficia le sue potenzialità. Di conseguenza è da notare che sono simili le difficoltà che possono incontrare i progettisti dei compilatori di entrambi i tipi di, in quanto, in entrambi i casi, si deve evitare il potenziale maggior numero possibile d'interferenze dovute al maggior numero di stadi rispetto alla soluzione pipeline di base.

Di seguito si darà un’idea di come è strutturata una architettura superscalare a due vie in grado di eseguire le istruzioni della soluzione pipeline vista precedentemente. Obiettivo dell’implementazione è quella di eseguire due istruzioni contemporaneamente in ogni stadio, a tal fine è necessario evitare conflitti strutturali, per questo ogni istruzione in esecuzione non deve interferire nell’utilizzazione dell’hardware dell’altra istruzione. In particolar modo il programma è scritto in modo da alternare istruzioni che utilizzano l’ALU o il suo status register (e quindi le istruzioni logiche aritmetiche e le jmpX) con quelle che accedono in memoria (la load e la store). In questo modo è possibile fare eseguire in parallelo due istruzioni contemporaneamente per ogni statio, purchè non ci siano conflitti sui dati o sul controllo.

Superscalare a due vie

- Si leggono da 2 istruzioni alla volta (dimensione complessiva dell'Instruction Cache pari a 64 bit)
- Ogni programa è organizzato in modo da alternare una istruzione logica/aritmetica o di salto con una di load o store

Istruzione	Stadi della pipeline							
ALU o branch	IF	ID	EX	MEM	WB			
Load o store	IF	ID	EX	MEM	WB			
ALU o branch		IF	ID	EX	MEM	WB		
Load o store		IF	ID	EX	MEM	WB		
ALU o branch			IF	ID	EX	MEM	WB	
Load o store			IF	ID	EX	MEM	WB	
ALU o branch				IF	ID	EX	MEM	WB
Load o store				IF	ID	EX	MEM	WB

93

Fig. 7.26 - Esempio di esecuzione di un programma in una architettura superscalare a due vie

Per poter far eseguire due istruzioni alla volta l'architettura è così modificata:

- la memoria cache istruzione invece di essere di 32 bit è strutturata a 64 bit, in modo che in ogni ciclo di fetch si leggono due istruzioni;
- il SCO è modificato per poter decodificare due istruzioni alla volta, inoltre deve essere cambiato il banco dei registri inquanto deve consentire alle istruzioni di poter leggere i propri operandi ed eventualmente poter permettere la scrittura c contemporanea di due registri nella fase di writeback
- non cambia il SCA dello stadio EX, così come quello dello stadio MEM, e WB, con l'unica accortezza che è necessario trasferire tra uno stadio e l'altro i dati e i segnali di controllo di due istruzioni contemporaneamente.

La figura successiva, Fig. 7.27, fa riferimento all'architettura di massima di un processore MIPS superscalare a due vie.

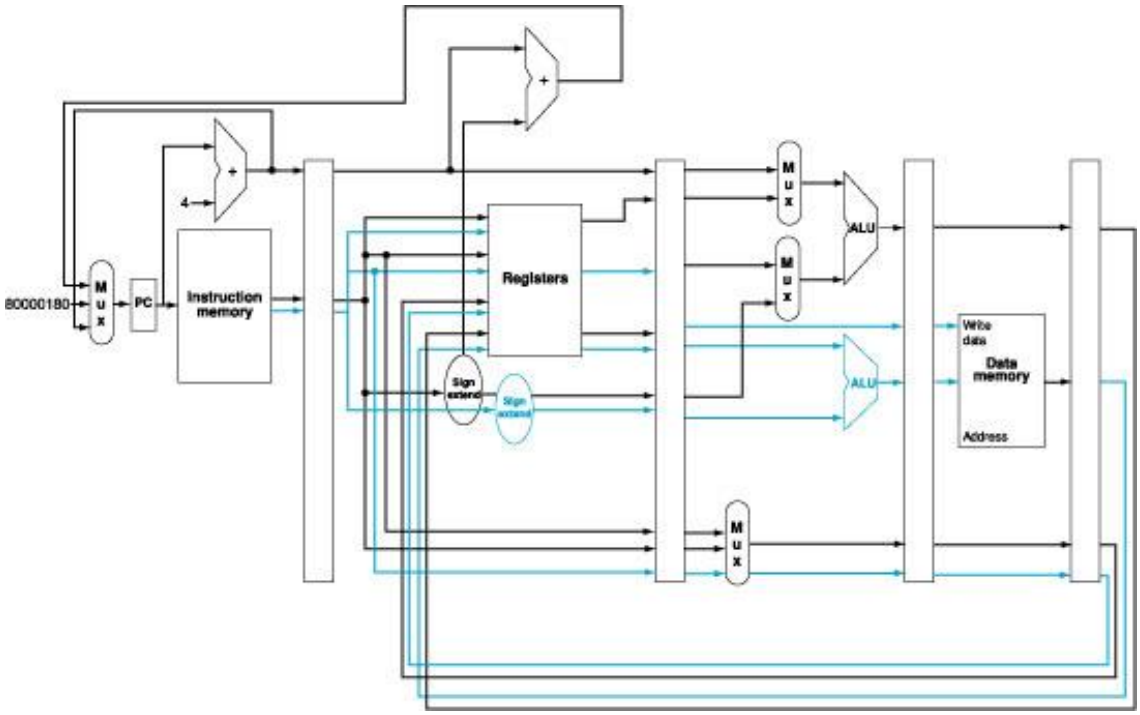
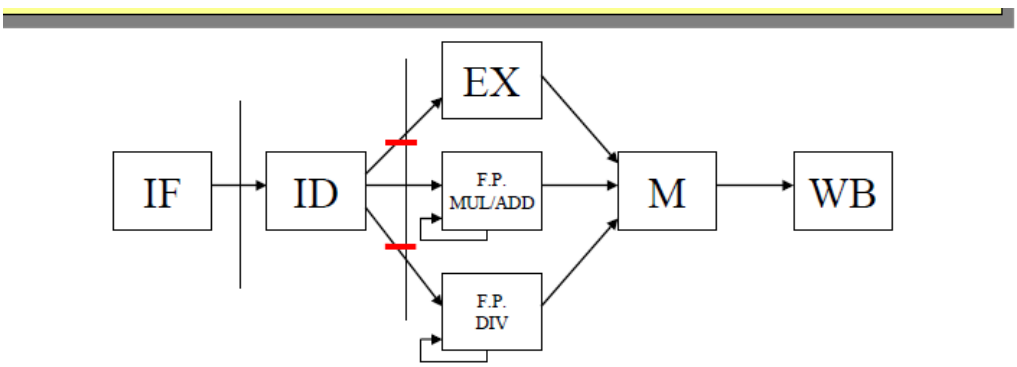


Fig. 7.27 - architettura MIPS superscalare a due vie

Di seguito si farà vedere come sia possibile far convivere nello stadio di EXECUTE sia una ALU che una FPU che una unità specializzata nelle divisioni e moltiplicazioni di numeri interi (MUL/DIV). Dal punto di vista circuitale la soluzione è banale identificando tre path distinti per le istruzioni che necessitano dell'ALU o della FPU o della MUL/DIV.



Ad esempio:

La soluzione implementativa più semplice è quella che prevede che nello stadio di EXECUTE ci sia una sola istruzione alla volta e quindi una istruzione che utilizza solo l'ALU, solo la FPU o

solo la MUL/DIV. In tal caso, però, lavorerebbe solo una unità di calcolo mentre le altre due sarebbero inutilizzate. Per permetter alle tre unità di calcolo di lavorare in parallelo, sfruttando così al massimo le potenzialità dello stadio di EXECUTE si potrebbe pensare, come nel caso della soluzione SUPERSCALARE di far eseguire contemporaneamente tre istruzioni che utilizzano in modo distinto le tre unità di calcolo. Problema di questa soluzione è che mentre l'istruzione che utilizza l'ALU è molto veloce le istruzioni che utilizzano le altre due unità di calcolo potrebbero essere molto lente. Per questo motivo si potrebbe pensare che nel frattempo che le unità di calcolo lente eseguono la loro istruzione nell'ALU potrebbero essere eseguite più istruzioni. Naturalmente questo complica molto l'architettura del sistema in quanto è necessario prevedere che l'ordine di esecuzione delle istruzioni possa essere differente da quello con cui sono state allocate in memoria dal programmatore o dall'assembler. Questo potrebbe essere consentito in alcuni casi ed in altri no, nel secondo caso sarebbe necessario inserire bolle o stalli per rispettare la semantica dei programmi.