Introduzione alle pipeline e all'architettura RISC

Architettura del processore

Una possibile unità di misura delle prestazioni di un sistema di elaborazione è il tempo. Per incrementare le prestazioni del processore, e conseguentemente ridurre il tempo di elaborazione, la prima soluzione che ci viene in mente è quella di aumentare la frequenza a cui il sistema lavora. Ma la frequenza di clock è legata alla tecnologia e all'organizzazione architetturale della CPU, e non può essere alzata a dismisura in quanto nascerebbero dei problemi di origine elettrica, quali ad esempio l'alta corrente che verrebbe a scorrere nelle linee di trasmissione, e di origine termodinamica. Una possibile alternativa sarebbe quella di far eseguire al processore le istruzioni in parallelo, anzichè in modo sequenziale, incrementando in questo modo il numero di istruzioni eseguite per unità di tempo. A questo scopo analizziamo come esegue le istruzioni il processore visto a Reti logiche (con riferimento a [1]), ricordando che, malgrado sia una semplificazione dei processori in produzione, ha comunque un set di istruzioni vasto, con diverse modalità di esecuzione e che le istruzioni sono di lunghezza variabile, il che implica che sono possibili accessi al bus per individuare gli operandi.

// Descrizione del processore in Verilog

// Fase di fetch

```
F0:       begin MAR<=mml(CS,IP); IP<=IP+1; MR_<=0; STAR<=F1; end
F1:       begin STAR<=F2; end
F2:       begin OPCODE<=d7_d0; MR_<=1; MJR<=estrai_tipo(d7_d0);
STAR<=F3; end
```

// Fase di decodifica

```
F3:       begin MJR<=decode(OPCODE); STAR<=MJR; end
```

/* Le operazioni che vengono ora dipendono dal tipo di istruzione e sono variabili, in generale comprendono una fase di lettura dell'operando in memoria, una di esecuzione e una di scrittura del risultato in memoria */

E' di seguito riportata una tabella riassuntiva delle varie fasi che il processore attraversa nell'eseguire un'istruzione e le rispettive risorse utilizzate.

Fetch	Decode	Read	Execute	Write
Bus OPCODE MJR	OPCODE MJR	Bus Registri Operativi	ALU FLAGS Registri Operativi	Registri Operativi Bus

Nota: I registri operativi sono registri invisibili al programmatore che sono utilizzati dal processore come supporto alle sue operazioni.

Una primo miglioramento delle prestazioni si ottiene suddividendo la parte controllo, che invece ora abbiamo considerato monolitica, in più unità che comunicano tra loro e che hanno funzioni specifiche: l'unità di accesso al bus lavora nelle fasi di fetch, scrittura e lettura, l'unità di decodifica in fase di decodifica, e la ALU nella fase di esecuzione. La figura fornisce una possibile realizzazione.

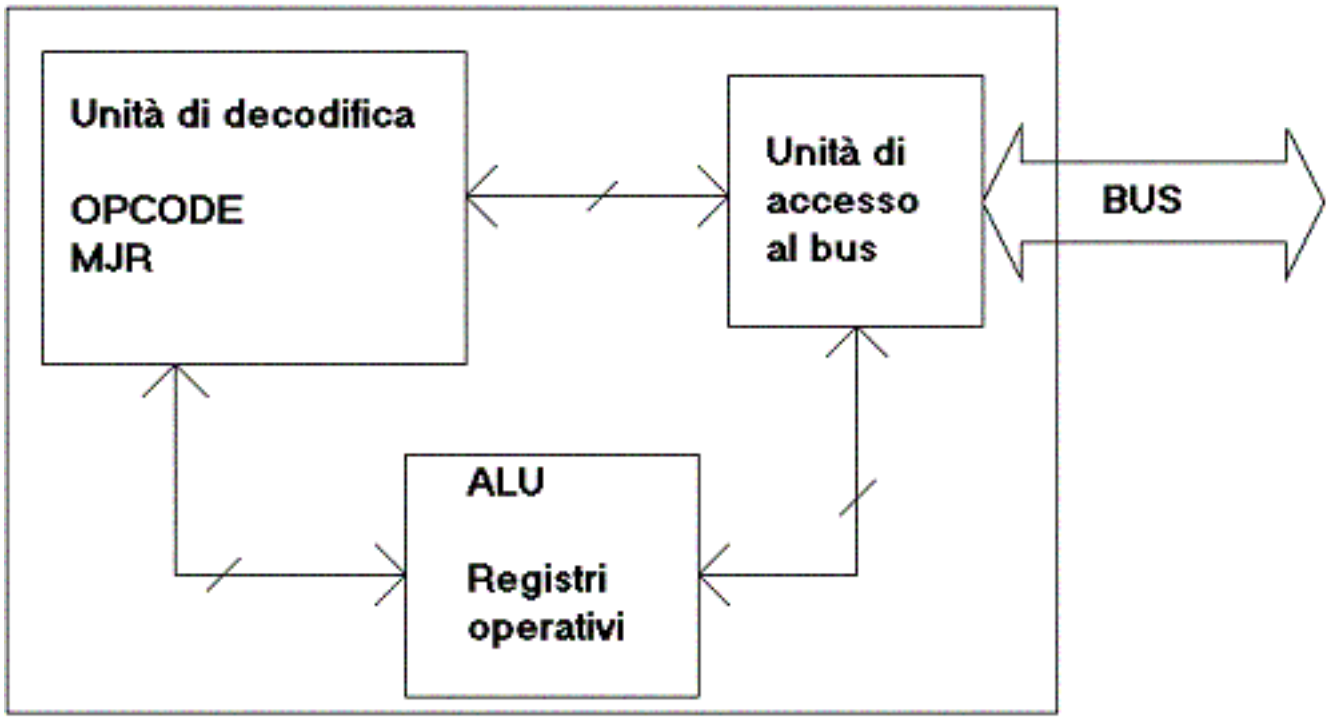

Fig. 1

Già a partire da questo schema si può implementare una sorta di parallelismo nell'esecuzione delle istruzioni, sovrapponendo le fasi nelle quali operano unità diverse, come illustrato nella figura che segue.

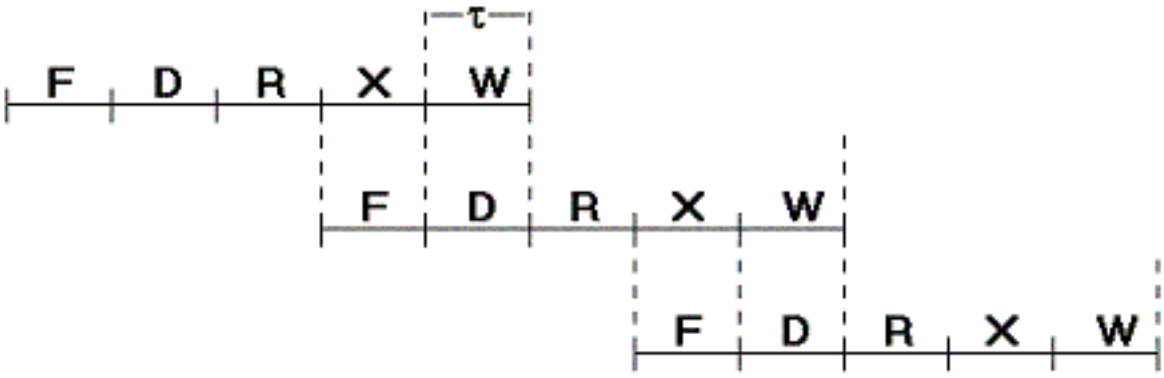

Fig. 2

La sovrapposizione delle fasi è possibile in quanto le unità che operano in contemporanea, utilizzano risorse diverse, ad esempio la fase di fetch (F) necessita del bus, del registro OPCODE e del registro MJR, mentre la fase di esecuzione (X) fa uso della ALU, dei FLAG e dei registri operativi.

Torna su

Pipeline

L'organizzazione dell'esecuzione di un'istruzione come una catena di stadi viene detta pipeline, e il concetto è analogo a quello della catena di montaggio in un' industria, dove il prodotto (nel nostro caso l'istruzione) attraversa diverse fasi prima di essere completato, e la catena viene continuamente alimentata in modo che ogni unità lavori costantemente. Il vantaggio di questo tipo di organizzazione sta nel fatto che il tempo necessario e terminare un prodotto (ad eseguire completamente un'istruzione) non è dato dalla somma dei tempi di attraversamento delle singole fasi, ma, idealmente, dal tempo di un singolo stadio. Con riferimento allo schema visto in precedenza un'istruzione per essere eseguita attraversa cinque stadi, quindi con la suddivisione della parte controllo avremo un'istruzione completata ogni tre cicli di clock (fig. 2), tenendo la struttura monolitica una ogni cinque. Si definsce throughput la frequenza con la quale viene completata un'istruzione: nel caso di pipeline ideale (un'istruzione per ciclo di clock) il throughput sarebbe di $1/\tau$, dove τ rappresenta il periodo di clock, con la pipeline della Fig. 2 il throughput è di $1/3\tau$, mentre senza pipeline è di $1/5\tau$. La fase di fetch non può essere ulteriormente anticipata perchè andrebbe in conflitto con la fase di read dell'istruzione precedente poichè entrambe richiedono l'utilizzo del bus. Questo è un problema perchè mi limita un ulteriore aumento del throughput, con conseguente limite alle prestazioni del sistema. Le soluzioni possibili sono due:

1. **Architettura Harvard:** Questa soluzione parte dalla considerazione che la fase di read legge un operando, mentre durante la fase di fetch viene prelevata un'istruzione. Il sistema CPU-memoria viene organizzato in maniera differente rispetto al classico modello di Von Neuman: sono infatti presenti due memorie distinte, una destinata a contenere i dati, mentre l'altra contiene le istruzioni.

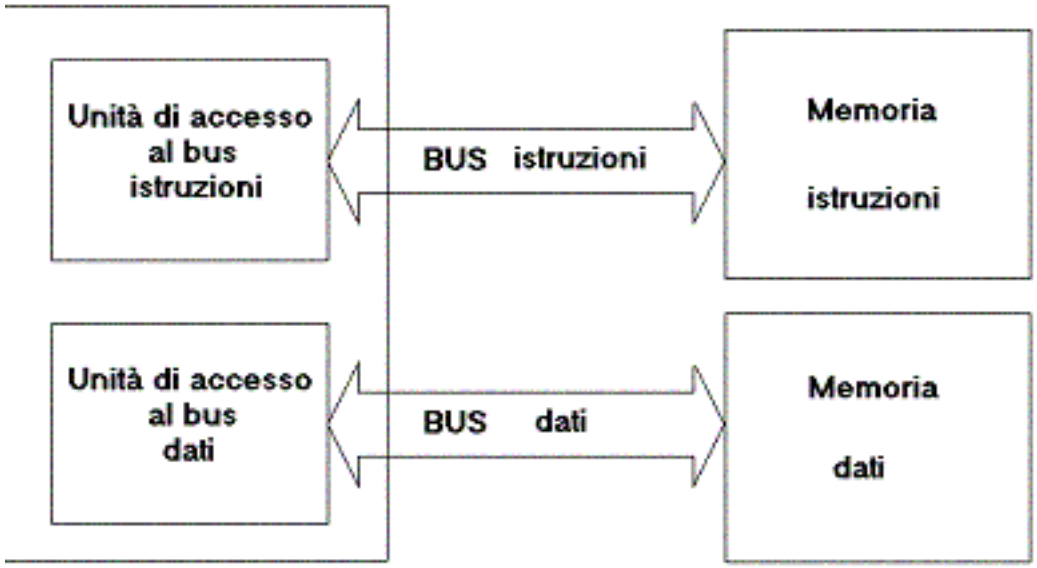

Fig.3

Le presenza di due memorie e di due bus distinti permette di anticipare la fase di fetch e di renderla parallela alla fase di read dell'istruzione precedente. La pipeline assume ora la struttura della figura.

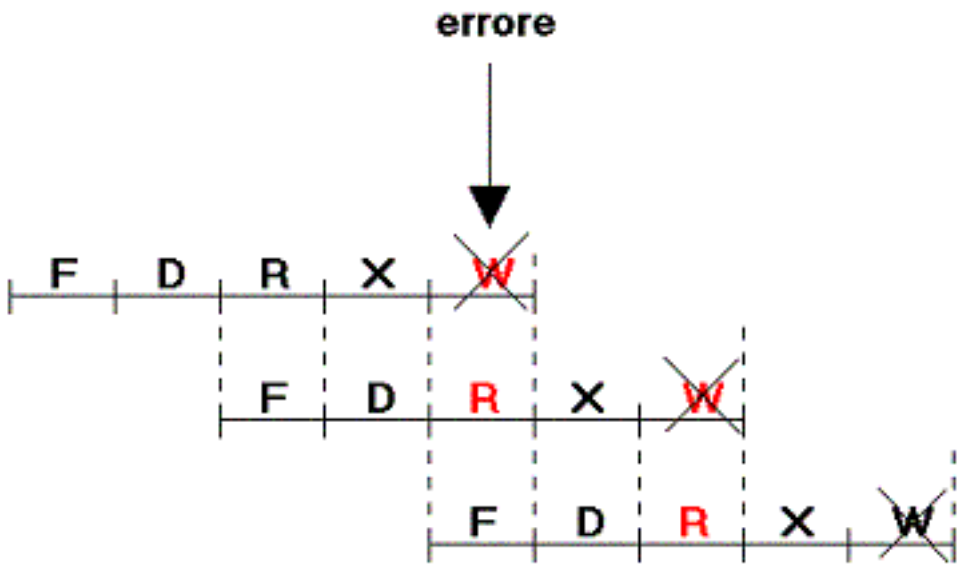

Fig. 4

Si nota però che c'è la sovrapposizione della fase di write (W) con quella di read (R), che effettuano entrambe accessi al bus dati. La soluzione a questo problema è un po' drastica: si elimina la fase di write di un'istruzione, ma alla stesso tempo si aumenta il numero di registri del processore nei quali possono essere immagazzinati operandi e risultati, e si realizzano delle istruzioni apposta per salvare in/caricare da memoria il contenuto di questi registri; è inevitabile che durante la loro esecuzione le prestazioni degradano in quanto si è prevesta una

fase di accesso in memoria. Il throughput di un processore organizzato con una pipeline come quella in fig. 4 è generalmente uguale a $1/2\tau$, tranne quando vengono eseguite le istruzione per salvare il contenuto dei registri in memoria, caso in cui abbiamo già visto che le prestazioni sono leggermente degradate. Un'ultima osservazione è che una struttura di questo tipo è costosa e viene realizzata solo per calcolatori dedicati.

2. **Coda di prefetch:** si aggiunge al calcolatore una nuova unità, la Prefetch Unit, che, quando il bus non è utilizzato, ne aprofitta per prelevare istruzioni e le memorizza in un apposito buffer detto coda di prefetch.

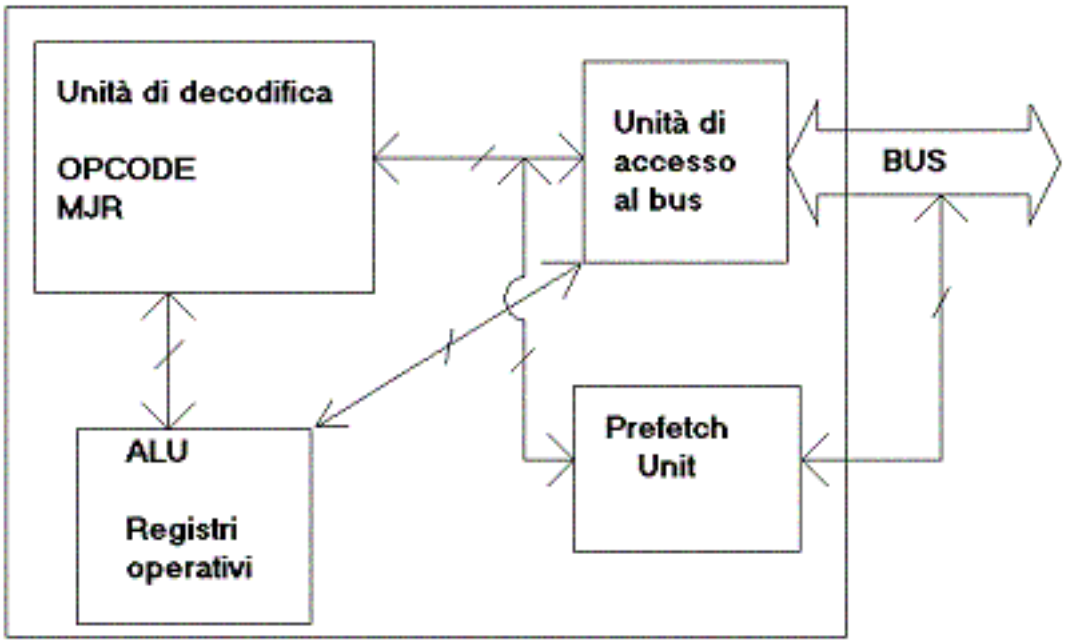

Fig. 5

In questo modo la fase di fetch non ha bisogno di effettuare un accesso al bus per prelevare un'istruzione in quanto questa risiede nella coda di prefetch, e può essere realizzata in parallelo alla fase di read dell'istruzione precedente. C'è inoltre da notare che la fase di prefetch (P) può essere realizzata in parallelo con la fase di decodifica, perchè a quest'ultima interessa il contenuto del registro OPCODE, mentre la prefetch unit memorizza l'istruzione nella sua coda interna, e non nel registro.

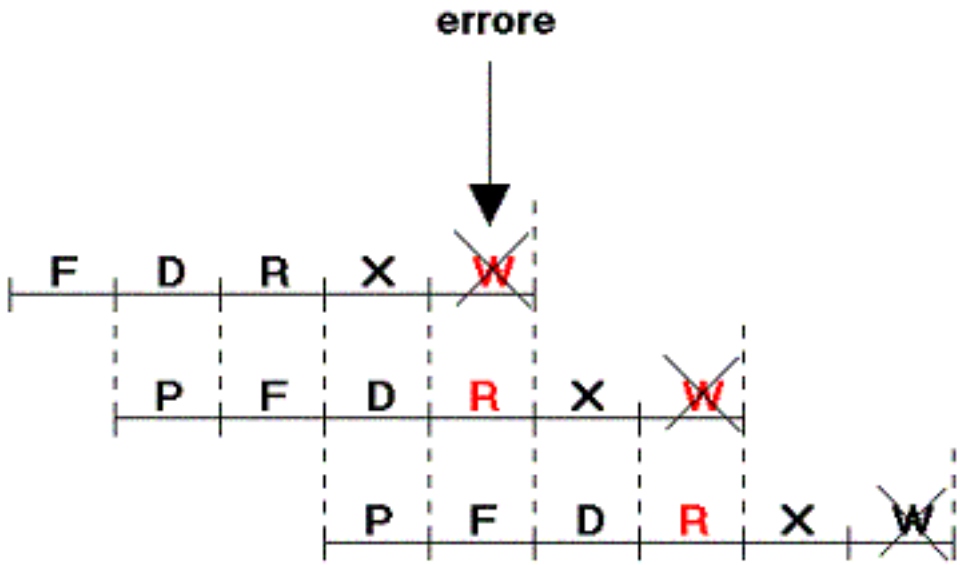

Fig. 6

Anche in questo caso c'è la sovrapposizione di due fasi (lettura e scrittura) che accedono al bus e la soluzione a analoga al caso precedente, eliminare nell'istruzione la fase di scrittura, incrementare il numero di registri presenti nel processore e realizzare istruzioni che salvano in/caricano da memoria il contenuto di questi registri.Per quanto riguarda le prestazioni a prima vista si può dire che abbiamo in genere abbiamo un throughput di $1/2\tau$, tranne quando vengono eseguite le istruzioni che accedono in memoria, analogo a quanto trovato per l'architettura Harvard. Questo risultato è vero se consideriamo, come abbiamo fatto finora, le varie fasi della pipeline di durata costante e uguale tra loro. Questa ipotesi non rispecchia il comportamento reale, in particolare la fase di decodifica è più veloce di quella di prefetch, in quanto quest'ultima prevede un accesso al bus, quindi le prestazioni risultano degradate. Per cercare di ovviare il problema, quando viene fatto il prefetch si cerca di memorizzare nella coda più di un'istruzione alla volta. Se ora si considerano la fasi di durata non costante, o comunque diversa tra loro, si deve prevedere un meccanismo che ritardi le fasi successive, in modo da garantire il corretto funzionamento del parallelismo, degradando un po' le prestazioni.

Torna su

Processori RISC

Il problema visto in precedenza era quello della durata delle fasi non costante a seconda

dell'istruzione eseguita. Per ovviare a questo inconveniente si cerca di avere un formato di istruzione standard, con l'operando immediato, oppure istruzioni con il solo compito di caricare/salvare un operando in memoria. L'idea di base è quella di delegare ad ogni singola istruzione un compito ben preciso e limitato; non è però necessario che il programmatore se ne serva, è il compilatore che effettua la traduzione in modo appropriato. Uno studio ha dimostrato che una piccola percentuale delle istruzioni (nell'ordine del 20%) viene largamente utilizzata (circa per l'80%), mentre ci sono delle istruzioni quasi mai impiegate e quasi superflue. Nasce quindi l'idea di ridurre il numero delle istruzioni e di suddividere le istruzioni in più operazioni semplici. Sono queste le considerazioni alla base dell'architettura RISC (Reduced Instruction Set Computers), che nasce con i seguenti obiettivi :

1. Fasi di durata costante;

2. Semplificazione delle istruzioni in modo da eliminare la decodifica e l'elaborazione di tipo: una volta che ho caricato l'istruzione ho già tutto quello che mi serve (nel caso di operandi o devono essere immediati, e quindi nell'istruzione, o precedentemente caricati in uno dei registri da un semplice istruzione di trasferimento);

3. Limitare gli accessi in memoria.

Soluzioni introdotte dall'architettura RISC:

a. Il codice operativo ha lunghezza costante (ad esempio 32 bit); in questo modo è soddisfatta la condizione 1;

b. Le istruzioni possono operare solo sui registri, e sono presenti due semplici istruzioni (LD e ST) dedicate al trasferimento del contenuto dei registri in memoria e viceversa: vengono soddisfatti così gli obiettivi 2 e 3;

c. E' presente un elevato numero di registri, il che rende possibile la soluzione del punto b.

Tornando alle istruzioni, in genere hanno una struttura come la seguente (le istruzioni che caricano/ salvano il contenuto di un registro in memoria hanno un formato diverso).

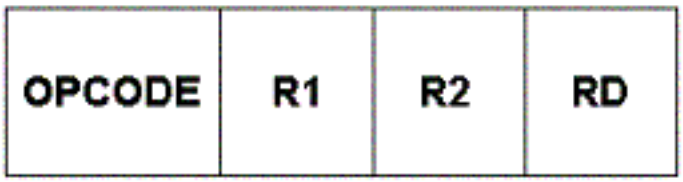

Fig. 7

Si può notare come il campo OPCODE sia ridotto, e questo grazie alla riduzione dell'insieme delle istruzioni. Inoltre sono presenti tre registri, due indicano l'operando sorgente e uno l'operando destinatario.

Con le soluzioni apportate dai processori RISC, in linea di principio l'istruzione viene completata in due fase: viene infatti eliminata la fase di decodifica, la fase di lettura non è più necessaria perchè gli operandi o sono immediati o risiedono in qualche registro del processore, e non si ha neppure la fase di scrittura, in quanto il destinatario è un registro. La pipeline assume quindi la seguente forma:

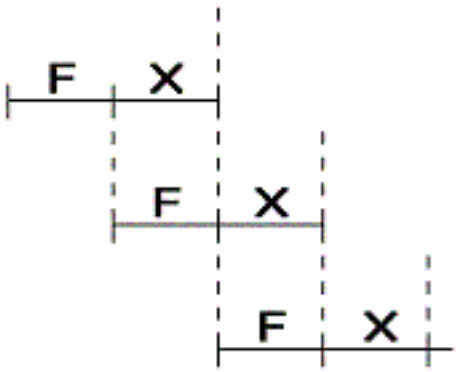

Fig. 8

Questo è un risultato molto importante, in quanto siamo arrivati al caso ideale di un'istruzione completata per ciclo di clock, con un throughput di $1/\tau$.

Nello schema precedente abbiamo considerato però che le fasi di fetch e di esecuzione abbiano durata uguale; non è detto che la fase di fetch duri un solo ciclo di clock, in quanto per procurarsi l'istruzione si deve accedere al bus di sistema che va ad una velocità minore del processore. Nell'architatture Harvard avevamo un bus dedicato, mentre ora il bus è unico e può essere utilizzato da meccanismi quali il DMA che lo bloccano fino a quando hanno finito le loro operazioni. Una possibile soluzione è quella di avere una fase di fetch che duri due cicli di clock: con un bus dati a 64 bit, tenendo presente che le istruzioni hanno lunghezza costante e pari a 32 bit, si prelevano due istruzioni, e prevedendo una coda di prefetch, in cui memorizzo l'istruzione che non viene subito eseguita il problema sembra risolto. In realtà non è detto che il bus abbia finito il prelievo dell' istruzione perchè la frequenza a cui opera il bus deve essere un quarto la frequenza del processore (o anche meno). Dal punto di vista elettrico il bus si può schematizzare come segue.

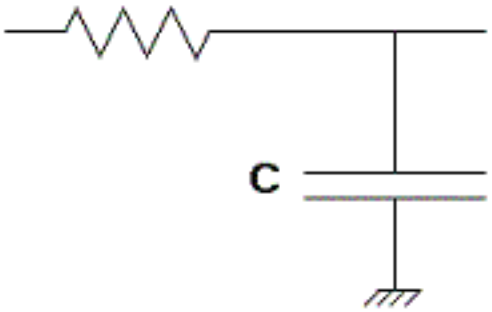

Fig. 9

La corrente che scorre in un condensatore è legata alla (variazione della) tensione con la seguente formula: i=Δv/Δt. Per valori tipici quali C=30*10^(-12)F, Δt=10^(-9)sec, Δv=5V (corrisponde alla transizione dal livello alto a quello basso o vicersa), la corrente che scorre in ogni linea è i=150mA: in un bus con circa cento linee (ne abbiamo ipotizzato 64 solo per i dati), se la frequenza è alta la potenza assorbita diventa notevole. Queste considerazioni ci fanno notare che il bus è il collo di bottiglia che rallenta le prestazioni tanto da arrivare ad pipeline come la seguente, con evidente diminuzione del throughput.

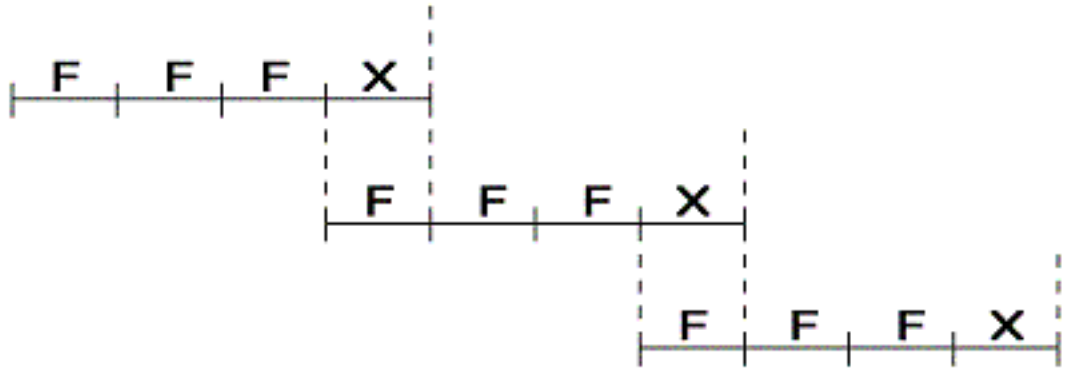

Fig. 10

La soluzione è di utilizzare una memoria cache dedicata alle istruzioni tra il bus e il processore, chiamata istruction cache (I-cache). Una possibile implementazione della I-cache è riportata di seguito.

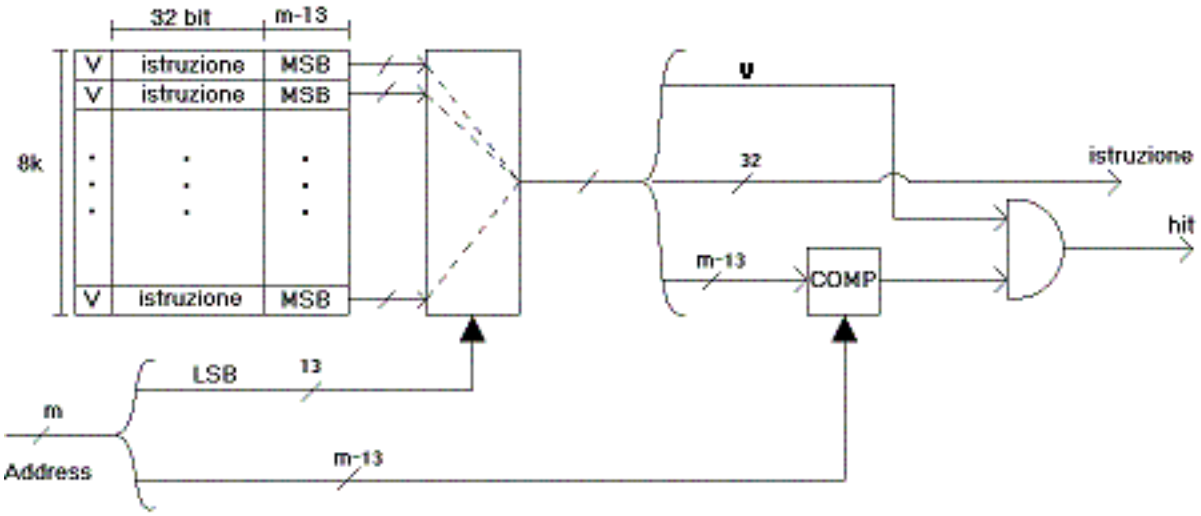

Fig. 11

Dal punto di vista logico la I-cache si può pensare come un array di elementi, ognuno composto da tre campi: un bit di validità (V), un campo per l'istruzione e un campo contenente gli m-13 bit più significativi (MSB) dell'indirizzo dell'istruzione. La I-cache in figura contiene 8k elementi, che vengono individuati dai 13 bit meno significativi dell'indirizzo dell'istruzione. Una volta individuato l'elemento viene confrontato il suo campo MSB con gli m-13 bit più significativi dell'indirizzo e l'uscita di questo comparatore viene mandata in una porta AND con il bit di validità. In caso di successo (HIT) il campo istruzione contiene proprio l'istruzione che volevamo, in caso contrario si deve accedere al bus per effettuare il prelievo dell'istruzione. In un ciclo di clock si riesce a capire se l'istruzione è presente o meno nella I-cache; nel caso di insuccesso l'indirizzo viene mandato sul bus vero e proprio e si ha un degrado delle prestazioni; ma si aprofitta del fatto che si deve accedere al bus per caricare un blocco di istruzioni nella I-cache (traserimento a burst: un indirizzamento, più trasferimenti). Risulta quindi fondamentale individuare un sitema di gestione della I-cache che mi permetta di avere un probabilità di successo alta. Questo discorso può essere ampliato per i dati e implementare una memoria cache dedicata, D-cache. Ma la questione è un po' più complicata rispetto alla I-cache, perchè nella D-cache ci si può anche scrivere, e si deve garantire la coerenza del dato propagando le modifiche anche nella memoria.

Si analizza ora la fase di esecuzione delle istruzioni. A volte può succedere che la fase di esecuzione di un'istruzione sia un po' più lunga del normale; in questo caso si deve aspettare ad eseguire l'istruzione successiva, ci sono dei tempi morti che posso sfruttare ad esempio per fare prefetch. L'unità che si occupa dell'esecuzione è la ALU, che sarebbe un circuito combinatorio monolitico, ma che di fatto viene modularizzato per semplificare il progetto a discapito dei tempi di attraversamento.

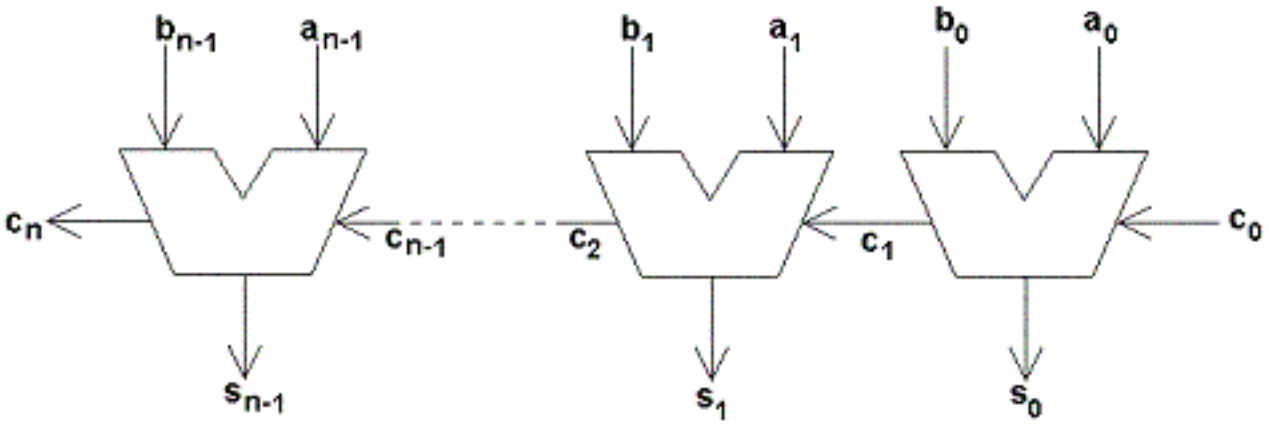

Fig. 12

Nasce l'idea di sfruttare la modularità per realizzare una pipeline interna delle istruzioni da eseguire. Ma abbiamo detto che la ALU è una rete combinatoria, quindi è trasparente e non si possono cambiare gli ingressi prima della fine dell'elaborazione. Si riesce comunque a suddividere la fase di esecuzione in sottofasi inserendo elementi non trasparenti, come i registri, tra un modulo e l'altro della ALU, che viene ad assumere una struttura come in figura.

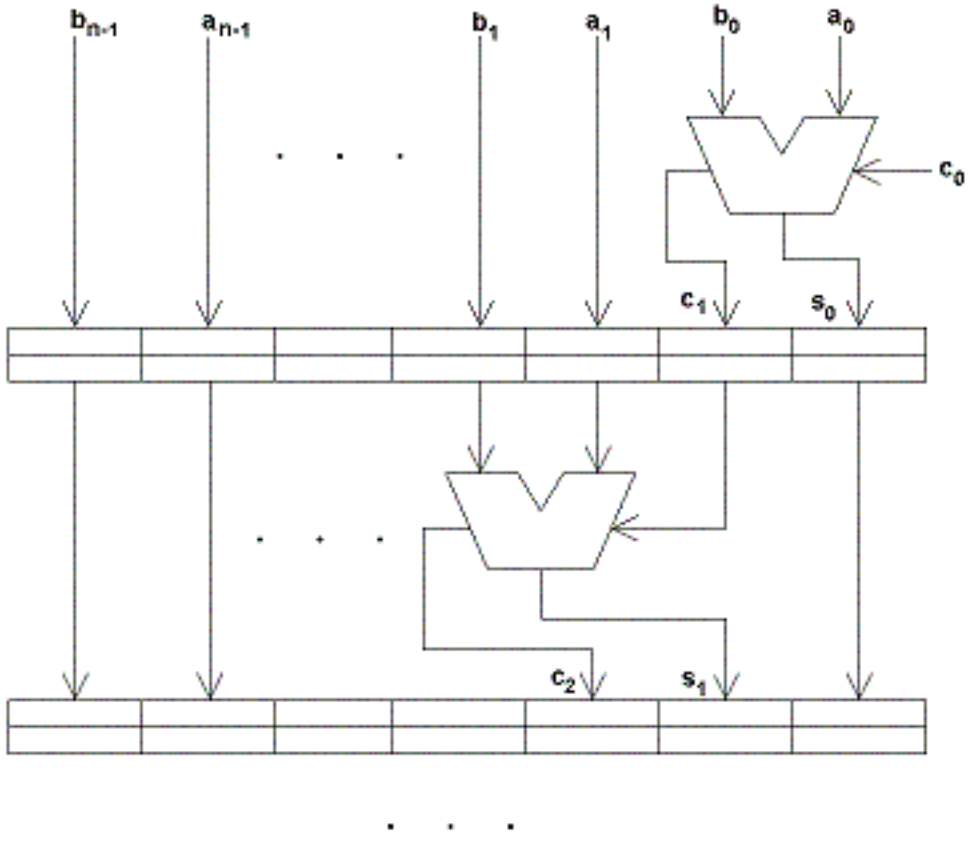

Fig. 13

Implementando la ALU in questo modo, la fase di esecuzione di un'istruzione può iniziare prima che sia terminata l'esecuzione precedente, e la pipeline assume questa forma (si fa riferimento ad una ALU composta da tre stadi)

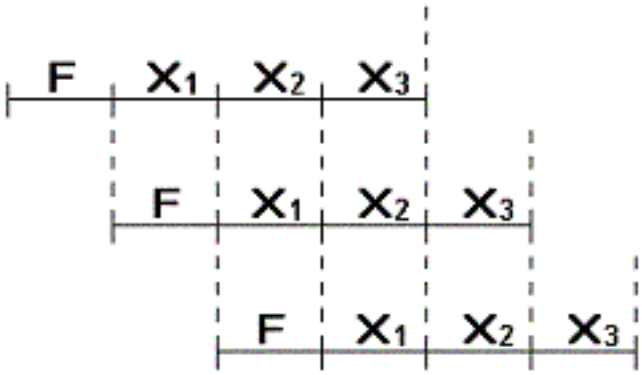

Fig. 14

C'è da notare che il tempo di esecuzione ora è diventato standard ed è superiore rispetto al caso senza registri, ma l'indice che a noi interessa maggiormente è il throughput che arriva a 1/τ. Il periodo del clock nel caso di ALU con registri ha durata 2T' (T' deve tenere conto, oltre del tempo di attraversamento T anche del Tsetup e del Thold degli elementi non trasparenti), mentre senza i registri il periodo di clock doveva comprendere il tempo di attraversamento degli n moduli, e valeva 2nT. In altre parole realizzare la ALU modularizzata e inserendo degli elementi non trasparenti consente di scegliere un clock minore rispetto alla soluzione della ALU trasparente.
Un'ultima osservazione riguarda il seguente pezzo di codice:

```
ADD     R1,R2,R3
ADD     R4,R5,R6
MOV     R3,R7
```

L'istruzione MOV ha bisogno del contenuto del registro R3, che però è pronto solo quando la prima istruzione finisce: si deve quindi inserire uno stato di attesa (che via software può essere realizzato inserendo una operazione NOP prima della MOV) dopo il quale la MOV è in grado di accedere correttamente al contenuto di R3.

Torna su

Salti nelle pipeline

Abbiamo visto che tramite l'utilizzo delle pipeline, con lo scopo di migliorare le prestazioni, viene anticipata la fase di fetch: questo però può risultare un problema quando l'istruzione successiva da eseguire non è quella successiva nel codice, ad esempio a causa di un'istruzione di salto, condizionato o incondizionato.

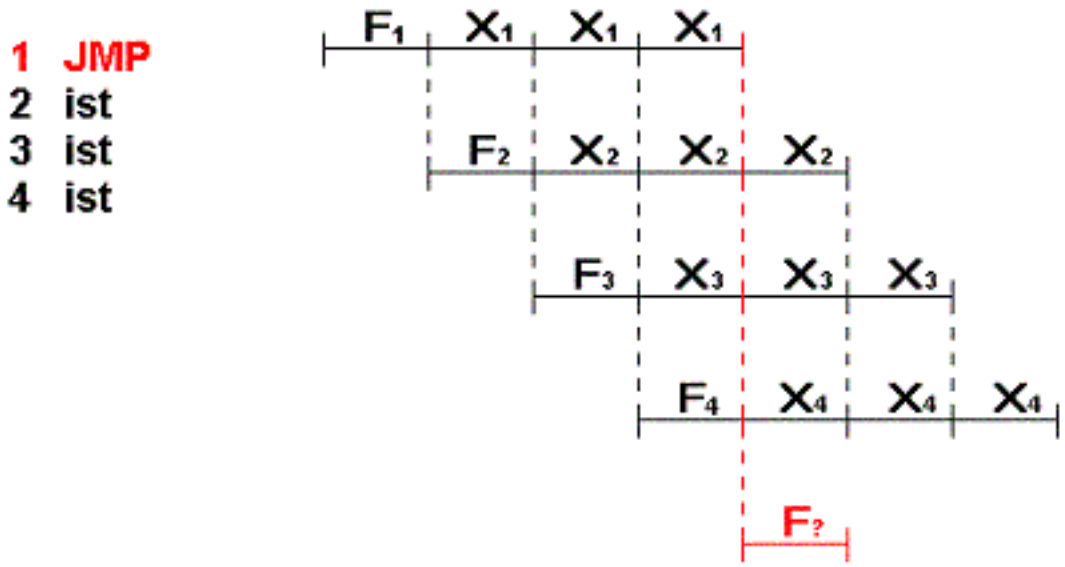

Fig. 15

Come si vede nell'esempio quando l'indirizzo di salto è pronto, sono già parzialmente eseguite tre istruzioni che non è detto che dovessero essere eseguite.

Ci sono diversi approcci per risolvere il problema dei salti nelle pipeline:

- **Pipeline Flushing**: Le istruzioni vengono caricate normalmente e quando l'istruzione di salto è terminata (è pronto l' indirizzo dell'istruzione successiva) viene abortita l'elaborazione parziale, ripulendo (flush) così la pipeline.
  Pre-condizioni: l'elaborazione parziale non deve avere effetti collaterali, non deve cambiare lo stato del processore, e questo si può ottenere scrivendo nei registri solo alla fine;
  Svantaggio: Le istruzioni di salto sono abbastanza frequenti, questa soluzione è poco efficiente; ha però il vantaggio di poter essere applicata a tutti i tipi di salto, sia condizionati che incondizionati.

- **Bubble**: Questa tecnica prevede l'inserimento di istruzioni NOP, che vengono dette bolle (bubble), in modo da non caricare le istruzioni successive ed evitare il flushing. Si potrebbe far in modo che sia il processore stesso ad inserire le bolle, ma per fare ciò in fase di fetch deve essere in grado di capire che si tratta di un'istruzione di salto. Un alternativa è quella di inserire le bolle via software, ad esempio in fase di compilazione: così non ho bisogno che sia il processore a farlo, ma ha lo svantaggio di essere poco efficace nel caso di salti condizionati.

- **Delayed branch**: questa soluzione nasce dalla considerazione che le istruzioni prima del salto vengono sicuramente eseguite, quindi si anticipa l'istruzione di salto, anche se ora il programma non ha un flusso completamente sequenziale. Di quanto si debba anticipare l'istruzione di salto dipende dalla profondità della pipeline. Ad esempio nel caso di una pipeline con una fase di fetch e tre fasi di esecuzione, il salto deve anticipare tre istruzioni, come mostrato in figura.

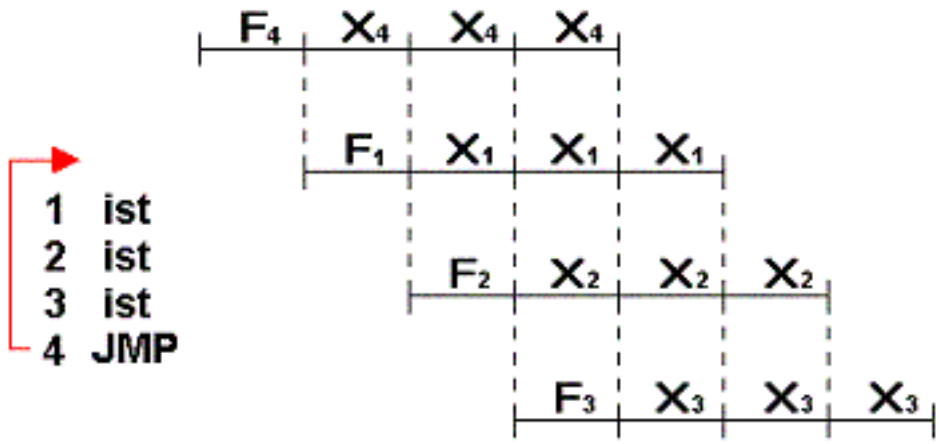

Fig.16

Questa tecnica prende il nome di delayed branch (salto ritardato), e non sempre è fattibile: se il salto è incondizionato e non dipende dai registri non ci sono problemi, se invece l'esecuzione dell'istruzione di salto dipende dall'esecuzione delle tre istruzioni precedenti allora non si può fare. L'operazione di anticipare le istruzioni di salto viene in genere fatta dal compilatore e non dal programmatore. Questo metodo funziona anche per salti condizionati, con problemi analoghi nel caso in cui il salto o la sua condizione dipendano dalle istruzioni precedenti.

- **Dual Pipeline**: si ha a disposizione una doppia pipeline, quando ho terminato la fase di fetch di un' istruzione di salto condizionato attivo l'esecuzione nella doppia pipeline dei due flussi possibili; poi una volta controllata la condizione, si decide quale pipeline scegliere e quell'altra viene svuotata, con il meccanismo del flushing. Per implementare questo approccio alla fine della fase di fetch devo sapere che si tratta di istruzione di salto condizionato e devo conoscere l'indirizzo del possibile salto. Comunque questa soluzione è poco utilizzata perchè è costosa.

- **Branch Prediction**: questa tecnica consiste nel fare una previsione della condizione e si sceglie tra i due rami quello più probabile. Se il salto è incondizionato non serve fare delle previsioni, so di sicuro dove salto. Per quanto riguarda i salti condizionati, la scelta si basa sulla considerazione che in un programma sono presenti molti cicli. Prendiamo in esame la seguente porzione di codice:

```
for   (i=0; i<n; i++;){
      ...
}
```

Quando verrà eseguito, la condizione i<n viene valuta n volte, e darà luogo a n-1 salti in indietro e solo alla fine avremo un salto in avanti. La branch prediction prevede sempre, per i salti condizionati, i salti all'indietro e se poi, una volta valutata la condizione si scopre che il salto era in avanti, si fa pipeline flushing. Questa tecnica funziona a patto che in fase di fetch si capisca che si tratti di un'istruzione di salto e si abbia l'indirizzo di salto.

Torna su

APPENDICE A: STORIA

Trattiamo ora l'evoluzione dei processori, in particolare i RISC (si è preso spunto da [2]).
I primi sistemi di calcolo vennero costruiti con logica cablata, cioè le varie unità venivano progettate come reti sequenziali sincronizzate. Questo approccio presentava però degli svantaggi, quali la complessità della logica, gli alti costi per apportarvi modifiche, e verso gli anni settanta si affermò la microprogrammazione, grazie alla quale l'unità di controllo viene vista come una sorta di elementare calcolatore nel calcolatore, dotato a sua volta di una memoria, detta memoria di controllo. In questo modo erano progettati macchine quali 8080, 8086, 86000, le quali erano caratterizzate da un insieme di istruzioni molto ampio e molto vario, tale da farle rientrare nella categoria di macchine CISC (Complex Instruction Set Computers). In questo periodo si riteneva vantaggioso avere un set di istruzioni particolarmente esteso e la tendenza era quella di arrivare a far eseguire alla macchina istruzioni simili a quelle di un linguaggio ad alto livello. Inoltre era di fondamentale importanza un uso efficente della memoria centrale, risorsa molto costosa a quei tempi, e grazie ad un repertorio di istruzioni esteso i programmi sarebbero stati più corti, e avrebbero occupato meno spazio. Proprio quando la microprogrammazione sembrava la soluzione definitiva, iniziarono a prendere campo le memorie a semiconduttore che, insieme all'introduzione della memoria cache decrementarono notevolmente il tempo di esecuzione dei programmi riducendo il vantaggio della memoria di controllo.
Ci si rese presto conto che, malgrado il repertorio di istruzioni fosse vasto, il 20% delle istruzioni costituiva ben l'80% dei programmi, e quindi che alcune istruzioni erano praticamente inutilizzate. Queste considerazioni, unite all'affermarsi delle architetture RISC, furono alla base del ritorno verso gli anni ottanta alla logica cablata. Proprio agli inizi del 1980 in due università della Californi vennero sviluppati due prototipi di macchine RISC. I risultati di questi studi vennero convogliati nello sviluppo della CPU SPARC, utilizzata dalla SUN Microsystem. Agli inizi degli anni novanta, si formò un consorzio tra IBM, Apple e Motorola per la progettazione e la produzione di una nuova CPU di tipo RISC, chiamata PowerPC; l'obiettivo di questa unione era quello battere il prodominio sul mercato dell'architettura Intel. A proposito di Intel, i primi processori (come l'8086) sono da considerarsi macchine CISC, e per mantenere la compatibilità all'indietro anche quelli moderni rientrano in questa categoria, ma grazie a degli accorgimenti il funzionamento è sempre più simile a quello delle macchine RISC

Torna su

Bibliografia

---

[1] Paolo Corsini, "Dalle porte and or not al sistema calcolatore", Edizioni ETS

[2] Giacomo Bucci, "Architetture dei calcolatori elettronici", McGraw-Hill

Torna su