Architettura degli elaboratori II

Un riassunto di
Paolo Alfano

Note Legali

Liberatoria, aggiornamenti, segnalazione errori:

Quest'opera viene pubblicata in formato elettronico senza alcuna garanzia di correttezza del suo contenuto. Il documento, nella sua interezza, è opera di Paolo Didier Alfano
e viene mantenuto e aggiornato dallo stesso, a cui possono essere inviate eventuali segnalazioni all'indirizzo *paul15193@hotmail.it*
Ultimo aggiornamento: 18 maggio 2017

Indice

1 Una semplice architettura RISC

1.1 Istruzioni e componenti

Nel seguito faremo molto spesso riferimento ad alcuni termini di cui diamo subito la definizione

- Instruction set Architecture(ISA): l'insieme delle istruzioni macchina di un processore. Due processori con stesso ISA possono avere struttura diversa
- Microarchitettura: l'architettura interna del processore a partire dal suo datapath
- Datapath: percorso che che compiono le istruzioni all'interno del processore per venire eseguite

Infatti diversi tipi di istruzioni possono dover attraversare componenti diverse del processore e in generale il tempo di esecuzione e' tanto maggiore quanto piu' e' lungo il datapath.
Ad ogni modo nel seguito intenderemo identici i termini microarchitettura, architettura o architettura interna. Inoltre intenderemo identici i termini ISA e Instruction set.

Il nostro punto di partenza consiste nel definire una semplice microarchitettura RISC in cui le istruzioni sono a dimensione fissata a 32 bit. L'idea che emergera' e' che le istruzioni macchina sono "semplici" e per questa ragione l'unita' che gestisce le varie componenti del datapath e' semplice a sua volta.
La nostra macchina RISC possiede 32 registri da 32 bit ognuno, e il registro R0 contiene sempre il valore zero. Inoltre la nostra macchina non considera operazioni in floating point.
All'interno della nostra macchina RISC avremo delle istruzioni a 32 bit la cui forma generica e' mostrata in Figura 1

Ad ogni modo vedremo quattro tipi di istruzioni:

- Istruzioni di tipo R: abbiamo due registri su cui operiamo e restituiscono il risultato in un terzo registro. Un esempio di istruzione R e' l'addizione:

  $DADD\ R1,\ R2,\ R3$
  la cui semantica e' $R1 = R2 + R3$
  Un altro esempio di istruzione R e' l'operazione di sottrazione:

  $DSUB\ R1,\ R2,\ R3$

op	reg_1	reg_2	reg_dest	shift	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Figura 1: Struttura di un'istruzione generica

Figura 2: Struttura di un'istruzione di tipo I

Figura 3: Struttura di un'istruzione di salto

la cui semantica e' $R1 = R2 - R3$
Notiamo che il prefisso D sulle istruzioni indica che operiamo sugli interi(per i numeri floting point useremo il prefisso F)

Notiamo che in questo caso il campo op indica che l'istruzione e' di tipo R, i tre campi successivi indicano dove prelevare e rilasciare i dati, il campo di shift non viene usato, mentre il campo funct definisce il tipo dell'operazione(somma, sottrazione..)

- Istruzioni che usano un valore immediato(tipo I): in questo caso cambia la struttura dell'istruzione, che viene mostrata in Figura 2

  Un esempio di istruzione di tipo I e' la somma con valori immediati:

  $DADDI\ R1,\ R2,\ 147$
  la cui semantica e' $R1 = R2 + 147$

- Istruzioni che operano sulla memoria: hanno la stessa struttura delle istruzioni di tipo I. I due esempi tipici di istruzioni di questo tipo sono load e store:

  $LD\ R_{dest},\ n(R1)$
  la cui semantica e' quella di inserire in R_{dest} 4 byte di memoria principale che si trovano a partire dall'indirizzo $n + val(R1)$ dove $val(R)$ indica il contenuto del registro R

  $SD\ R_{source},\ n(R1)$
  la cui semantica e' quella di mettere alla posizione $n+val(R1)$ il contenuto di R_{source}

- Istruzioni di salto: la struttura delle istruzioni di salto viene mostrata in Figura 3

  I salti possono essere condizionati o(branch) o incondizionati(jump)

Chiaramente considerare solo queste operazioni ci limita in qualche modo. Ad esempio non possiamo eseguire salti condizionati superiori a 2^{16} bit e non possiamo gestire le procedure e anche se in un'implementazione reale queste problematiche andrebbero risolte, non sono necessarie per capire il funzionamento

base di una CPU.
Solitamente i primi due passi da effettuare per eseguire un'istruzione sono:

1. Prelevare l'istruzione
2. Mentre decodifichiamo l'istruzione leggiamo uno o due registri relativi all'istruzione

Dopo, le azioni effettuate dipendono dal tipo di istruzione anche se tutte sono piu' o meno simili fra loro. Ad esempio tutte le istruzione eccetto la jump sfruttano la ALU.

Volendo realizzare una versione monociclo della nostra macchina ricordiamo che il datapath e' composto da due tipi di elementi logici:

- Elementi di stato: in grado di memorizzare un valore come registri e memorie. Ogni elemento di stato possiede almeno due segnali in input(il valore da scrivere e il clock) e un segnale di output(il valore da leggere). Al termine di ogni ciclo di clock viene restituito il valore contenuto nell'elemento di stato
- Elementi combinatori: elementi il cui output e' funzione solo dell'input all'istante corrente, come le ALU

Poiche' tutto funzioni e' sufficiente che il ciclo di clock sia abbastanza lungo da permettere che i segnali in input ad un elemento di stato si siano stabilizzati prima della fine del ciclo stesso.
Solitamente usiamo degli elementi di stato come input per degli elementi combinatori il cui output viene inserito all'interno di altri elementi di stato secondo:

$$ElemStato_1 \rightarrow ElemCombin \rightarrow ElemStato_2$$

Nulla esclude che invece gli elementi di stato in input agli elementi combinatori siano gli stessi su cui verra' prodotto l'output. Chiariamo questo concetto tramite un esempio

Esempio 1 : un semplice esempio di quanto appena detto sta nell'incremento del Program Counter, infatti ad ogni ciclo il PC viene letto e incrementato da una ALU dedicata il cui output viene inserito nuovamente nel PC! Questa situazione e' mostrata in Figura 4

1.2 Una semplice Control Unit

Per realizzare una Control Unit che ci permetta di dirigere la ALU e la memoria, dobbiamo prima di tutto considerare i 6 bit del campo *op* e generare in output i segnali per: dirigere la scrittura dei registri, la scrittura della memoria dati, i multiplexer e la ALU.
La nostra Control Unit sara' fondamentalmente il circuito logico che implementa la tabella di verita' i cui valori di ingresso sono i sei bit del campo *op* e il cui output sono i segnali per gestire il controllo del datapath MIPS. Nello specifico i segnali sono:

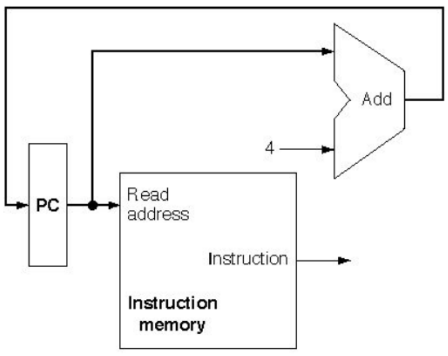


Figura 4: Incremento del Program Counter

- RegDst:gestisce i registri. Se abilitato indica che il numero del registro di destinazione per la scrittura proviene dal registro reg_dest, altrimenti il numero di registro di destinazione proviene da reg_2
- RegWrite: gestisce i registri. Indica che verra' effettuata la scrittura di un certo valore in una certa posizione
- ALUsrc: gestisce la ALU. Se asserito indica che il secondo operando della ALU sono i 16 bit inferiori dell'istruzione, altrimenti il secondo output proviene dalla seconda uscita dell'insieme dei registri
- PCsrc: gestisce il Program counter. Se asserito il nuovo valore di PC e' determinato dal salto, altrimenti e' determinato dal normale incremento della ALU.
- MemRead: gestisce la memoria. Se asserito indica che verra' letta una certa posizione il cui valore verra' inserito in un certo registro.
- MemWrite: gestisce la memoria. Se asserito indica che verra' scritto un certo valore in memoria presso una certa posizione
- MemToReg: gestisce la memoria. Se asserito indica che il valore da scrivere nei registri proviene dalla memoria, altrimenti viene proviene dalla ALU
- ALUop: gestisce la ALU. Indica quale operazione deve effettuare la ALU. E' un campo a due bit che ci permette di distinguere se la ALU deve eseguire un operazione di tipo R, una LOAD/STORE, oppure la verifica di un'operazione di salto condizionato. Nel caso l'operazione sia di tipo R sara' il campo funct a determinare precisamente quale operazione eseguire(somma, sottrazione..)

A titolo d'esempio riportiamo l'esecuzione di una istruzione di tipo R:

1. Tramite il PC preleviamo le istruzioni dalla Instruction Memory

2. Il campo *op* viene mandato alla Control Unit mentre i campi *reg_1* e *reg_2* vengono usati per indirizzare la memoria dei registri

3. La Control Unit tramite il campo *op* determina che l'istruzione e' di tipo R ed invia i segnali per gestire il datapath, incluso il segnale che specifichi alla ALU il tipo di operazione che deve eseguire

4. Nel frattempo la memoria dei registri produce in output i due valori relativi ai registri

5. La ALU produce il risultato che viene presentato in input alla memoria dei registri. Tutti i passi visti finora possono essere eseguiti in un solo ciclo di clock se esso ha una durata maggiore dei tempi di ritardo delle varie unita' funzionali. Notiamo che per ora nessuna scrittura e' stata fatta.

6. Scriviamo sui registri. Tale scrittura avverra' effettivamente alla fine del ciclo di clock

1.3 Architetture multiciclo

Anche se utilizzare una macchina che esegua ogni operazione in un singolo ciclo di clock possa funzionare, nella realta' questa soluzione non viene implementata poiche' dovendo garantire l'esecuzione dell'istruzione piu' lunga, l'esecuzione di ogni altra istruzione piu' corta comporta uno spreco di tempo. Per questo motivo si preferisce usare delle operazioni che vengano eseguite in piu' fasi, dove ogni fase impiega un ciclo di clock. In questo modo evitiamo sprechi di tempo e anche se impieghiamo n cicli di clock per eseguire una certa istruzione in una architettura ad n fasi, ognuna di queste fasi dura $1/n$ della lunghezza del ciclo di clock.
Mostriamo in Figura 5 l'architettura di una macchina multiciclo per andare a spiegare nel seguito ogni singola fase

Notiamo subito che abbiamo cinque fasi distinte. Vediamole una per una

1. Instruction Fetch: preleviamo l'istruzione dalla Instruction memory tramite il program counter. Nel frattempo incrementiamo il program counter. Salviamo l'istruzione e il nuovo program counter in due registri di appoggio invisibili al programmatore.

2. Instruction decode: all'inizio di questa fase non sappiamo ancora quale sia l'istruzione da eseguire. In questa seconda fase eseguiamo due operazioni in parallelo: preleviamo i registri dalla memoria e calcoliamo l'eventuale indirizzo di branch(anche se non e' detto che l'istruzione sia effettivamente di branch). Notiamo inoltre che prelevare i registri dalla memoria potrebbe non essere utile(non sappiamo se eseguiremo un'operazione di tipo R) ma leggerli non e' dannoso. Dunque li salviamo nei registri temporanei A e B. Notiamo infine che data la struttura irregolare delle istruzioni non

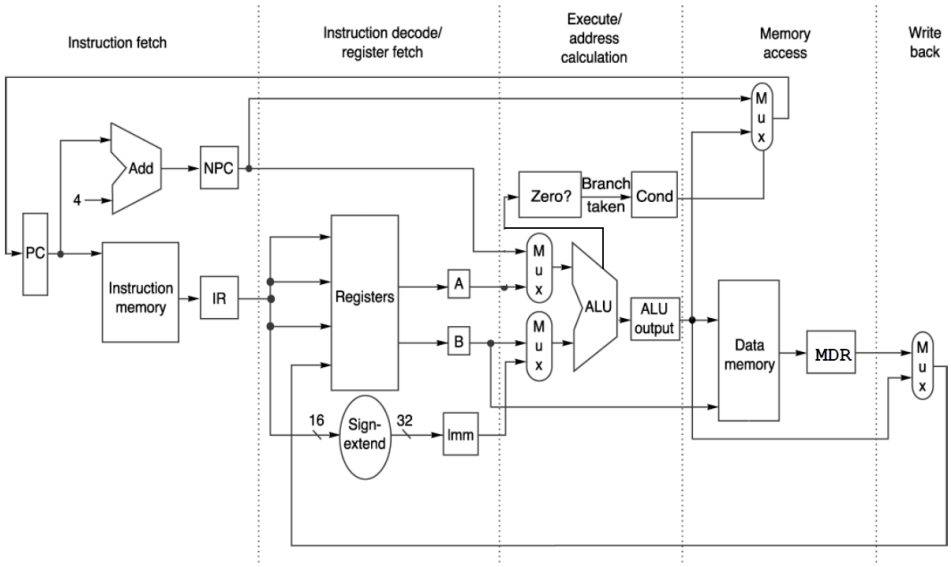


Figura 5: Architettura di una macchina MIPS multiciclo

possiamo prelevare gli operandi fino a quando non sappiamo che tipo di operazione stiamo eseguendo

3. Execution: la Control Unit dirige la ALU usando l'op code e le permette di eseguire una delle quattro istruzioni(istruzioni I, R, di salto condizionato o di salto incondizionato).

4. Memory Access: In questa fase vengono svolte in parallelo due mansioni. Se l'istruzione e' di load/store andiamo ad operare sulla Data Memory, se invece abbiamo un'istruzione di tipo I o R andiamo a salvare l'output dell'ALU nella memoria dei registri.
   Notiamo quindi che nel datapath dell'architettura MIPS esistono due memorie, una per i dati e una per le istruzioni che vengono accedute in momenti diversi. In pratica potrebbero essere un'unica memoria e difatti le RAM moderne contengono sia istruzioni che dati. Una memoria che contenga dati e istruzioni non puo' pero' esistere al livello del processore poiche' devono poter essere accedute nello stesso ciclo di clock a causa del pipelining(che poi vedremo). Difatti in ogni CPU moderna Instruction memory e Data memory sono distinte.

5. Write back register: in questa fase completiamo l'operazione di load andando a scrivere il contenuto di MDR nel corretto registro di destinazione

Quindi nell'implementazione che abbiamo fornito un branch richiede 3 cicli, un'istruzione di tipo I, R o una store richiedono 4 cicli, infine la load richiede 5 cicli. Valutando statisticamente le occorrenze di queste istruzioni di diversa

durata il numero di clock necessari per eseguire un'istruzion e' CPI=3.98
Abbiamo ottenuto una miglioria rispetto alla macchina monociclo visto che ogni ciclo di clock adesso dura 1/5 del ciclo di clock originale.
Come cambia la Control Unit? Chiaramente diventa piu' complessa perche' ad ogni ciclo dobbiamo specificare quali segnali inviare e dove. Anche in questo caso potremo rappresentare la Control Unit con una tabella di verita' che generi un output e stabilisca quale sia la successiva riga della tabella da eseguire. Ma quindi la nostra Control Unit e' una *macchina di Moore* in cui l'output dipende solo dallo stato corrente mentre la transizione nello stato successivo dipende sia dallo stato corrente che dall'input. Questo tipo di macchina puo' essere comodamente rappresentata tramite un automa a stati finiti. Che differenza c'e' a rappresentare una macchina con un microprogramma o con un automa? Nessuna, e' solo questione di comodita' per rappresentare macchine ben piu' complesse della nostra. Chiaramente non bisogna spingere troppo sulla complessita' dell'architettura altrimenti ci spostiamo sulle architetture CISC, di minore prestanza.

1.4 Contrapposizione RISC-CISC

All'inzio degli anni '50 viene introdotto il concetto di microprogramma per creare un livello che si collocasse tra le istruzioni macchina e la loro esecuzione. In quegli anni introdurre delle istruzioni molto complesse era giustificato poiche'

- Il tempo di accesso alla RAM era molto superiore al tempo di accesso alla ROM che conteneva i microprogrammi. Inoltre, non esistendo le cache l'accesso in memoria era costoso e ogni volta che vi si accedeva si cercava di fare una gran quantita' di lavoro con una sola istruzione(riducendo il numero di accessi in memoria)

- I compilatori moderni non esistevano e avere istruzioni complesse semplificava il lavoro del compilatore

- La RAM era costosa e avere istruzioni espressive permetteva di generare eseguibili piu' corti che permettevano di risparmiare spazio

- Era semplice aggiungere nuove istruzioni tramite microprogrammi. Infatti bastava scrivere il nuovo microprogramma e e inserirlo in una nuova ROM che ai tempi era facilmente sostituibile.

Questa versatilita' aveva pero' un costo: istruzioni macchina cosi' diverse fra loro avevano lunghezza variabile(in tal modo non si sprecava spazio). Visto che venivano prelevate in memoria quantita' fisse' di bit non si sapeva da subito se il blocco prelevato contenesse una nuova istruzione, una parte di una precedente o di una successiva. Inoltre poiche' le istruzioni potevano indirizzare liberamente la RAM, questa diviene velocemente un collo di bottiglia.
Esempi di set di istruzioni complesse sono l'ISA del Pentium 4, complesso e irregolare per mantenere la backward compatibility.

Mentre procede lo sviluppo delle architetture CISC, all'inizio degli anni '80 a Berkeley viene progettata una CPU la CU non e' descritta da un microprogramma. In questo momento viene coniato il termine RISC(e di conseguenza anche il termine CISC). Da questi progetti prendereanno vita le architetture SPARC della Sun Microsystem.
Nel frattempo a Stanford viene creata una macchina per sfruttare al massimo il pipelining(che vedremo in seguito). Tale macchina e' quella su cui ci siamo basati per sviluppare la nostra macchina delle pagine precedenti, e il suo nome era MIPS.
Questi due lavori paralleli sulle architetture RISC sono stati ispirati da vari studi(Cocke, Tanenbaum), che mostravano come spesso veniva usata solo una piccola parte di tutte le modalita' di indirizzamento messe a disposizione. Dunque la maggior parte dei programmi spendeva molto tempo ad eseguire istruzioni semplici ed era dunque inutile avere istruzioni tanto complesse. Grazie ai due lavori precedenti vengono quindi definiti i principi delle architetture RISC:

1. La CPU deve eseguire un numero limitato di istruzioni macchina semplici che possano essere portate a termine in un ciclo di clock di breve durata. Istruzioni semplici richiedono datapath piu' corti e dunque CU piu' semplici e cicli di clock piu' corti

2. L'accesso alla RAM va limitato il piu' possibile per cercare di eliminare il collo di bottiglia su di essa.

3. Dal punto precedente segue che le istruzioni debbano sfruttare argomenti contenuti nei registri e questo implica che ogni macchina debba avere a disposizione un numero elevato di registri.

Un esempio di famiglia architetturale che segue questi principi e' la famiglia SPARC con istruzioni a 32 bit e avente una forma piu' o meno regolare.
Piu' nel dettaglio, esistono degli ulteriori buoni principi per progettare una CPU moderna

- Rinunciare ad un microcodice complesso: questa scelta e' possibile se il maggior spazio occupato in RAM dal codice non rappresenta un problema

- Definire istruzioni di lunghezza fissa: questo permette di prelevare e decodificare molto piu' efficientemente ogni istruzione

- Solo Load e Store dovrebbero indirizzare la RAM

- Avere registri general purpose in abbondanza per memorizzare computazioni intermedie

- Sfruttare il piu' possibile il pipelining: ovvero dividere l'architettura in piu' parti e fare in modo che in ogni momento ogni parte dell'architettura sia impegnata con una istruzione diversa in una fase diversa. Studieremo approfonditamente questa caratteristica in seguito

Proviamo ad applicare i principi elencati in un esempio

Esempio 1 : supponendo di avere a disposizione l'istruzione CISC

$$ADDL3\ 42(R1),\ 56(R2),\ 0(R3)$$

il cui significato e' prendi la word all'indirizzo dato da $56+val(R2)$, prendi la word all'indirizzo $0+val(R3)$, somma i due valori e metti il risultato nella cella di memoria all'indirizzo $42+val(R1)$.
Possiamo trasformarlo in un codice RISC come quello che segue:

$LD\ R4,\ 56(R2)$
$LD\ R5,\ 0(R3)$
$ADD\ R6,\ R4,\ R5$
$SD\ R6,\ 42(R1)$

Come gia' detto il codice RISC e' piu' lungo e usa piu' registri ma opera sulla RAM solo tramite Load e Store

Ad ogni modo la vicenda RISC-CISC si e' risolta con la sconfitta delle architetture CISC e gia' nel 2000 oltre il 90% del mercato era coperto da architetture RISC. Delle architetture CISC solo la famiglia x86 sembra essere sopravvissuta. Comunque, anche se tale famiglia utilizza ancora istruzioni tipicamente CISC, esse vengono tradotte dopo la fase di fetch in piu' microistruzioni semplici di lunghezza fissa a 72 bit.
Questa scelta e' dettata dal fatto di voler mantenere una retrocompatibilita', infatti Intel ha anche dato vita a una linea di processori completamente RISC a 64 bit.
Ad ogni modo la distinzione tra architetture RISC/CISC non ha piu' molto senso e una misura che le distingue e' data dalla *CPI* ovvero le Clock Cycles Per Instruction. Ossia il numero di cicli di clock necessari per eseguire una certa istruzione.

2 Pipelining

2.1 Concetti di base

La struttura che abbiamo dato della macchina MIPS multiciclo ci permette di dividere un lungo ciclo di clock in cinque fasi distinte. Come dicevamo, suddividere il ciclo in fasi ci permette di sovrapporre l'esecuzione di istruzioni diverse che siano in fasi diverse. Oltre a questo occorre evitare di compiere contemporaneamente operazioni differenti che operino sullo stesso componente. Pero' nell'architettura che abbiamo definito le fasi di Instruction Decode e Write Back lavorano entrambe sulla memoria dei registri. In realta' questo fatto e' permesso se stabiliamo che ognuna delle due fasi sfrutti la memoria in un momento diverso del ciclo di clock(Ad esempio la WB nella prima meta' del ciclo mentre la ID nella seconda meta' del ciclo). Risolto questo inconveniente possiamo mostrare graficamente il procedimento di pipelining in Figura 6

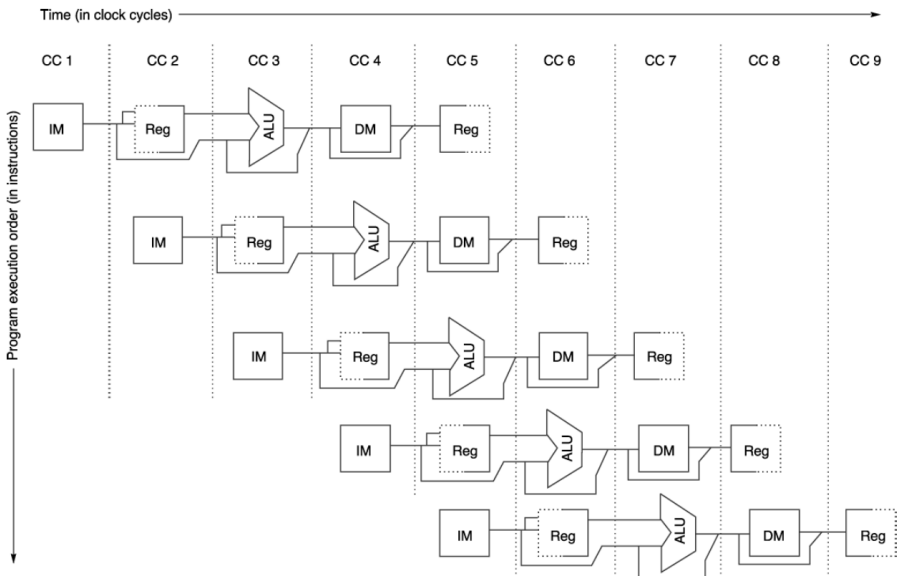
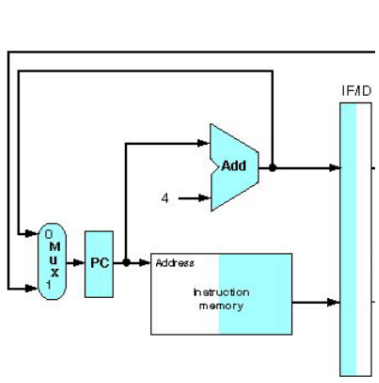


Figura 6: Pipelining

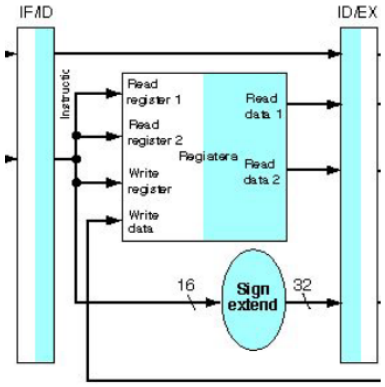
Notiamo quindi che il procedimento di pipelining puo' essere visto come una serie di datapath traslati nel tempo in cui necessitiamo di memorie distinte per i dati e per le istruzioni in modo da evitare conflitti. Notiamo inoltre che la Figura 6 non mostra la gestione del PC che deve essere incrementato ad ogni ciclo.
Nelle implementazioni reali ogni stage e' separato dal successivo da opportuni *registri della pipeline* che fungono da registri di output per la fase precedente e da registri di input per la fase successiva. Di fatto i registri della pipeline coincidono con quelli che avevamo introdotto per passare dalla macchina monociclo a multiciclo, ovvero IR, A, B... e proprio come questi sono invisibili al programmatore.
Chiameremo ogni insieme dei registri tra una fase e l'altra con il nome delle due fasi tra cui tali registri si collocano. Quindi avremo i registri di pipeline *IF/ID, ID/EX, EX/MEM, MEM/WB*.
Notiamo che dopo la fase finale non abbiamo ulteriori registri di pipeline poiche' alla fine della fase di write back le istruzioni devono alterare in qualche modo visibile al programmatore lo stato del processore.
Consideriamo adesso un esempio per mostrare il percorso seguito da un'istruzione all'interno del datapath della nostra architettura pipeline.

Esempio 2 : consideriamo le varie fasi che deve attraversare un'istruzione load. Mostriamo in Figura 7 l'intero procedimento diviso nelle sue fasi.

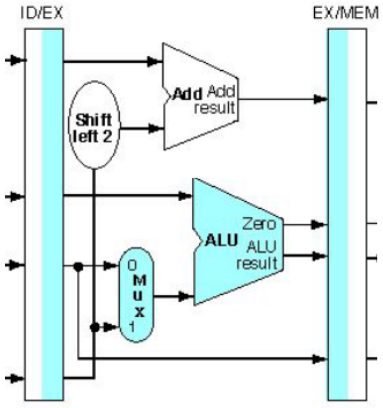
Nella fase di IF l'istruzione viene letta e posta nel registro IF/ID.
Nella fase di decode leggiamo da IF/ID la posizione da leggere nella me-



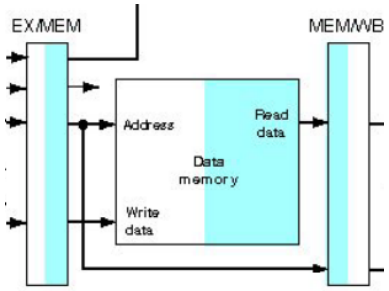
(a) Fase di IF



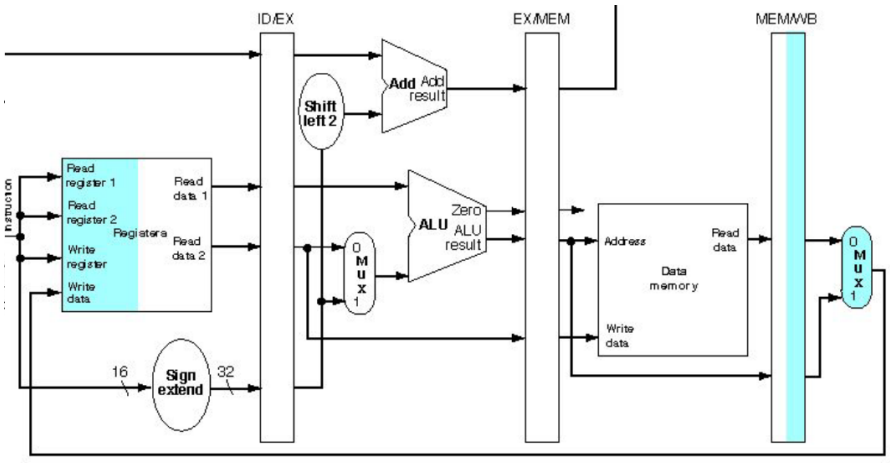
(b) Fase di ID



(c) Fase di EX



(d) Fase di MEM



(e) Fase di WB

Figura 7: Fasi del processo di pipelining

moria dei registri. Vengono letti i due registri anche se in realta' useremo solo il primo. I 16 bit del valore immediato vengono convertiti in un valore a 32 bit. I valori prelevati dai registri e il valore immediato vengono scritti in ID/EX
Nella fase di execution leggiamo da ID/EX e il contenuto del primo registro viene sommato al valore immediato. Il risultato viene scritto in EX/MEM.
Nella fase di memory leggiamo da EX/MEM la posizione da leggere nella memoria dati. Tramite essa leggiamo la memoria dati e salviamo cio' che abbiamo letto in MEM/WB.
Nella fase di write-back leggiamo da MEM/WB e andiamo a mettere il risultato letto nella memoria a registri. Per capire in quale posizione della memoria a registri andare a scrivere dobbiamo "portarci dietro" il valore R_{dest} per tutte le fasi dalla ID in poi.

Cerchiamo adesso di valutare quale e' l'incremento di prestazioni dovuto all'utilizzo della pipeline. Supponiamo che una pipeline che lavori a pieno regime esegua un'istruzione ad ogni ciclo di clock. Pero' a causa dell'overhead della pipeline(che e' una struttura piu' complessa) abbiamo un ulteriore ritardo di 0.2 cicli di clock. Nel complesso per ogni istruzione impieghiamo 1.2 cicli di clock.
Supponiamo invece che in una CPU senza pipeline le operazioni della ALU e le operazioni di branch impieghino 4 cicli di clock mentre quelle di accesso alla memoria impieghino 5 cicli di clock. Supponendo che rispettivamente l'occorrenza di tale operazioni sia 40%, 20% e 40%. Allora il tempo medio di esecuzione di un'istruzione sara':
$4 * 0.4 + 4 * 0.2 + 5 * 0.4 = 4.4$ cicli di clock
Se eseguiamo il rapporto
$\text{speedup} = \frac{4.4}{1.2} = 3.7$
otteniamo un incremento delle prestazioni di 3.7 volte.

2.2 Problematiche del pipeline

Purtroppo le cose non sono cosi' semplici visto che esistono tre classi di problemi che minano la produttivita' della pipeline. Quando si verifica uno di questi problemi e' necessario fermare(to stall) la pipeline in attesa che vengano risolti.

- Problemi strutturali: si verificano perche' alcune risorse harware all'interno del datapath vengono usate contemporaneamente da due istruzioni nella pipeline. Ad esempio una cache L1 unica per dati e istruzioni genererebbe moltissimi problemi strutturali. Per questo tale memoria e' divisa in due parti duplicate.
  La soluzione e' duplicare le unita' funzionali piu' usate come le ALU ma questo rende piu' complessa e costosa l'architettura, costringendo i pro-

gettisti a trovare un compromesso tra complessita', prestazioni, consumi e costi.

- Problemi sui dati: si verificano perche' le istruzioni di un programma non sono scorrelate fra loro e possono succedere casi come il seguente
  $DADD\ R1, R2, R3$
  $DSUB\ R4, R1, R5$
  in cui il risultato salvato in R1 dalla DADD deve essere usato subito dopo dalla DSUB che deve attendere che l'istruzione precedente scriva nella memoria registri il risultato. In questo lasso di tempo la DSUB deve necessariamente bloccarsi.
  Una semplice tecnica usata per risolvere questo tipo di problema e' il *forwarding*. L'idea e' di rendere il risultato dell'istruzione disponibile per le istruzioni successive sin da subito, prima della fase WB. Questo puo' essere fatto utilizzando i registri della pipeline che memorizzano i risultati intermedi.
  Questa tecnica pero' non risolve tutti i problemi perche' per esempio le istruzioni di load non possono essere pronte prima di essere passate dalla memoria, ovvero verso la fine del ciclo di clock. Fondamentalmente quindi, il forwarding funziona molto male per le istruzioni di load i cui argomenti devono essere utilizzati nelle istruzioni successive. Questo e' quello che viene definito come un caso di stalling

- Problemi di controllo: si verificano quando viene eseguita un'istruzione di branch. Infatti, cambiando il valore del PC l'istruzione successiva che intanto e' gia' stata prelevata non verra' eseguita. E dunque il suo prelevamento sara' stato inutile. Per capirlo meglio consideriamo:

  $LD\ R1, 0(R4)$
  $BNE\ R0, R1, else$
  $DADD\ R1, R1, R2$
  $else: DSUB\ R1, R1, R3$

  Quando sappiamo se il branch viene eseguito la DADD ha gia' completato la sua fase di IF, anche se non e' detto che poi venga eseguita.
  Una soluzione semplice a tale problema e' quella di procedere come se nulla fosse. Se il branch non viene effettuato allora non avremo problemi. Se invece il branch viene effettuato dovremo caricare la nuova istruzione e avremo perso un ciclo di clock.
  Con questa tecnica chiaramente si sprecano dei cicli di clock e quanti cicli sono sprecati dipende dalla frequenza delle istruzioni di branch.
  Un'altra soluzione consiste nel fare delle assunzioni a priori sulle istruzioni di salto. Questa tecnica viene detta *predizione statica dei branch*. Ad esempio potremmo assumere che i branch all'indietro sono sempre eseguiti. Questa semplice predizione e' di fatto molto utile in un loop il cui corpo viene eseguito n volte. Infatti tale predizione si rivelera' giusta per

ogni esecuzione del loop e sbagliata solo all'uscita dal loop dando un'accuratezza di previsione pari a $n/n+1$. Quindi questo ci permette di evitare di sprecare n cicli di clock.
Un'altra tecnica e' quella del *delayed branch*. Consiste nel piazzare subito dopo un branch una istruzione che deve comunque essere eseguita, indipendentemente dall'esito del branch. Eseguire tale istruzione ci permette nel frattempo di aggiornare correttamente il PC di modo che l'istruzione successiva venga prelevata ed eseguita correttamente. Questa tecnica richiede l'intervento del compilatore e non e' sempre facilmente applicabile.

Un capitolo a parte meriterebbe la gestione delle eccezioni come interrupts, trap e altro. In questo corso ci accontentiamo di sapere che gestirle non implica nulla di nuovo concettualmente ma rende l'intera architettura piu' complicata.

2.3 Versioni piu' sofisticate

Vediamo qualche architettura pipeline piu' avanzata.
Se una pipeline funziona bene perche' non usarne due in parallelo? Nel caso in cui si possano prelevare due istruzioni dalla memoria e che possano essere lanciate in parallelo avremo un guadagno computazionale e i conflitti anche in questo caso dovranno essere ridotti dal compilatore o dinamicamente(lo vedremo nei capitoli successivi).
Naturalmente potremmo pensare di utilizzare molte piu' pipeline in parallelo ma le architetture moderne non sono andate in questa direzione e hanno preferito replicare la fase piu' costosa di tutte, la EX.
Infatti le operazioni come moltiplicazioni e divisioni(in particolare su floating point) sono molto piu' costose delle altre. Certamente potremmo eseguirle in un unico ciclo di clock. Tale ciclo sarebbe pero' molto lungo sprecando il tempo delle altre fasi che completerebbero molto prima. L'idea ottimale e' quindi di usare una pipeline piu' breve per le operazioni semplici e una pipeline piu' lunga e complessa per le operazioni piu' complicate. Tale schema viene mostrato in Figura 8.

Architetture che replichino la propia fase EX sono dette *architetture superscalari*. Anche se formalmente una architettura e' superscalare se puo' mandare in esecuzione piu' di un'istruzione per ciclo di clock. Una tecnica avanzata che vedremo in seguito e' quella dello scheduling dinamico della pipeline Nel complesso le architetture RISC sono particolarmente adeguate per il pipelining perche':

- Le istruzioni hanno stessa lunghezza, semplificando il fetch e la decodifica
- Il numero di istruzioni ridotto permette di avere una struttura regolare che permette allo stadio di ID di iniziare a leggere la memoria registri in parallelo alla decodifica dell'istruzione. Se cio' non e' possibile dovremmo spezzare la fase ID in due: una fase di decode e una di fetch

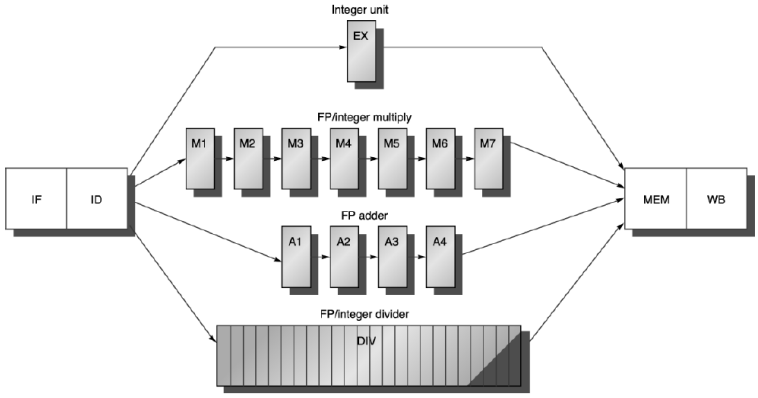


Figura 8: Architettura pipeline reale

- Gli operandi in memoria dati compaiono solo nelle load e nelle store. Questo ci permette di usare lo stadio EX per calcolare l'indirizzo di memoria e lo stadio successivo per accedere alla memoria.

3 ILP Dinamico

Abbiamo visto che le tecniche di pipelining sovrappongono almeno in parte l'esecuzione delle istruzioni. Questa forma di parallelismo viene detta *parallelismo a livello di istruzione*, o *Instruction level Parallelism*(ILP).
Abbiamo visto nella sezione precedente i tre tipi di problemi e se tali problemi non fossero presenti potremmo:

- Aumentare la frequenza di clock: vuol dire ridurre il ciclo di clock, ovvero suddividere il lavoro in piu' fasi. Ma se aumenta il numero di fasi, aumenta anche il numero di istruzioni eseguite in parallelo. Quindi esiste un legame evidente tra frequenza di clock e parallelismo. Ovviamente aumentare il numero di fasi e' una tecnica che non puo' essere sfruttata indefinitamente sia per problemi architetturali(complessita' della CU) sia per motivi tecnologici(piu' fasi $\rightarrow$ piu' componenti $\rightarrow$ piu' calore, piu' consumo)

- Mandare piu' istruzioni in esecuzione in parallelo: questa tecnica, nota come *multiple issue* e' sfruttata da tutti i processori moderni . Vengono mandate in esecuzione un'insieme di istruzioni indipendenti tra loro detto *issue packet*. Ogni issue packet viene mandato in esecuzione in un preciso lasso di tempo, detto *issue slot*. Le tecniche per essere sicuri che siano indipendenti fra loro le vedremo piu' avanti.
  Chiaramente non deve esistere alcun problema strutturale. Devono quindi esserci le unita' funzionali sufficienti per permettere l'esecuzione in parallelo(piu' ALU, una Instruction memory che permetta di prelevare piu' istruzioni in parallelo...).

In un'architettura pipeline senza multiple issue, il numero di istruzioni portate a termine per ogni ciclo dovrebbe essere
Clock Per Instruction(CPI) = 1
Nelle architetture multiple issue tale valore dovrebbe addirittura scendere sotto 1.
Vedremo invece come, anche in presenza di multiple issue, spesso il CPI medio sia un valore superiore a 1.
Abbiamo visto che per avere multiple issue dobbiamo capire quali siano le istruzioni che possiamo mandare in esecuzione. Per capirlo usiamo due tecniche:

- Multiple issue statico: ogni issue packet viene determinato dal compilatore staticamente. Lo vedremo nel prossimo capitolo
- Multiple issue dinamico: in questo caso e' il processore a determinare quali istruzioni eseguire a runtime. Lo vedremo nelle prossime pagine

Ad ogni modo queste due tecniche non sono del tutto distinte. Infatti nella realta' i processori di una categoria tendono ad usare anche qualche tecnica dell'altra.
Andiamo a vedere le principali tecniche dinamiche per evitare le perdite di tempo dette *alee*(o *hazard* in inglese)

- Scheduling dinamico della pipeline: supponiamo di avere il seguente codice
  $LDR4, 100(R2)$
  $DADDR10, R4, R8$
  $DSUBR12, R8, R1$
  L'esecuzione della DADD dipende dalla LD. Dunque fino a che la LD non e' completata blocchiamo l'esecuzione della pipeline, anche di quelle istruzioni come la DSUB che potrebbe anche essere eseguita poiche' del tutto indipendente. Comprendere questo fatto e cambiare l'ordine delle istruzioni a tempo di esecuzione viene detto scheduling dinamico della pipeline. Poter eseguire queste istruzioni *out of order* implica anche un completamento out of order.
  Va da se' che tutto questo deve accadere senza modificare il comportamento globale del programma.
- Branch Prediction: la CPU tiene traccia per ogni branch del risultato delle sue precedenti esecuzioni. Tramite esse puo' stabilire con buona probabilita' quali siano le successive istruzioni da eseguire. Questa tecnica e' cosi' utile che spesso viene usata anche nei processori a ILP statico
- Speculazione hardware: estensione del branch prediction. Richiede un hardware particolare e sofisticato. Per questo motivo non viene adottata da tutti i processori a ILP dinamico. La vedremo piu' nel dettaglio in seguito

3.1 Dipendenze e alee

Cerchiamo ora di formalizzare il concetto di dipendenza tra istruzioni. Infatti determinare se e come un'istruzione possa dipendere da un'altra e' fondamentale per capire se sia possibile mandarle in esecuzione in parallelo. Se due istruzioni sono indipendenti possono essere eseguite in qualunque ordine. Se invece hanno una qualche dipendenza possono essere sovrapposte solo in parte. I due principali tipi di dipendenze sono: *dipendenza sui dati* e *dipendenza sui nomi*:

- Dipendenza sui dati: diremo che un'istruzione j e' *data dependent* da un'istruzione i se vale una delle due condizioni

  1. j usa un risultato prodotto da i
  2. j e' data dependent da k e k e' data dependent da i

  La seconda condizione esplicita che possono esistere delle catene di dipendenze

- Dipendenza sui nomi: si verifica quando due istruzioni sfruttano uno stesso registro. La dipendenza sui nomi puo' essere di due tipi

  - Antidipendenza: tra i e j se i legge un registro che j deve scrivere. In questo caso i deve avere il tempo di leggere altrimenti leggera' il valore scritto da j che e' errato.
  - Dipendenza in output: se i e j scrivono successivamente nello stesso registro. In questo caso deve essere mantenuto l'ordine di scrittura affinche' il valore finale sia quello dell'ultima istruzione eseguita nel programma originale

Vediamo qualche esempio di dipendenza

Esempio 1 : il seguente codice

$$
\begin{array}{l}
loop : LD\ F0, 0(R1) \\
FADD\ F4, F0, F2 \\
SD\ F4, 0(R1) \\
DADD\ R4, R4, \# - 8 \\
BNE\ R4, R5, LOOP
\end{array}
$$

presenta tre dipendenze sui dati. La seconda istruzione dalla prima, la terza dalla seconda e la quinta dalla quarta.

Esempio 2 : il seguente codice

$$
\begin{array}{l}
DIV\ F0, F2, F4 \\
ADD\ F6, F0, F8 \\
SUB\ F8, F10, F14 \\
MUL\ F6, F10, F12
\end{array}
$$

presenta una antidipendenza tra ADD e la SUB. Presenta inoltre una dipendenza in output tra la ADD e la MUL

Ad ogni modo le dipendenze sui nomi non sono vere dipendenze visto che per risolverle basta cambiare il nome del registro usato in una delle due istruzioni. Per poterlo fare dobbiamo avere a disposizione qualche registro libero e tener traccia di tali modifiche per le istruzioni che seguiranno.

Nel complesso, supponendo di avere due istruzioni i e j dove i occorre prima di j, le dipendenze possono essere riassunte in tre categorie:

- RAW(read after write): j cerca di leggere un registro prima che i lo abbia scritto. È il caso della dipendenza sui dati
- WAR(write after read): j cerca di scrivere un registro prima che i lo abbia letto. In tal caso verrebbe letto da i un valore errato
- WAW(write after write): j cerca di scrivere un registro prima che i lo abbia fatto. Se permettessimo questa situazione alla fine avremmo il valore sbagliato nel registro. È il caso della dipendenza in output

La tecnica di base per lo scheduling dinamico venne inventata da Robert Tomasulo nel 1967. Nella sua architettura Tomasulo usava delle *stazioni di prenotazione* associate ad ogni unita' funzionale. In tali stazioni le istruzioni che non potevano ancora essere eseguite attendevano. In particolare quando un'istruzione non poteva essere eseguita, segnalava l'identificativo della stazione di prenotazione che conteneva l'operando mancante. Quando un'operazione terminava un *Common Data Bus* che connetteva tutte le unita' funzionali, trasferiva i dati necessari da una stazione di prenotazione a quelle in attesa dei risultati(oltre che alla memoria dei registri).
In particolare, l'esecuzione di un'istruzione nell'architettura di Tomasulo era divisa in tre parti

1. Issue: fase di prelievo, decodifica e inserimento dell'istruzione nella stazione di prenotazione associata all'unita' funzionale desiderata. Se gli operandi necessari sono disponibili vengono prelevati e mandati nella stessa stazione dell'istruzione. Se invece non sono ancora disponibili, come gia' detto veniva assegnato per ogni operando mancante un identificatore della stazione destinata a produrre l'output
2. Execute: fintanto che mancano degli operandi, monitoriamo il common data bus; quando tutti gli operandi sono disponibili l'istruzione viene inviata all'unita' funzionale. Se abbiamo piu' istruzioni disponibili vengono inviate una dopo l'altra possibilmente in pipeline. Nel caso le istruzioni da eseguire siano load o store vengono memorizzate in un'area di memoria detta load/store buffer. Nel caso siano load, possono completare appena l'unita' di accesso alla memoria e' disponibile. Nel caso siano store attendono il calcolo del risultato da memorizzare

3. Write Result: quando l'unita' termina l'esecuzione il risultato viene mandato sul bus alla memoria dei registri e alle eventuali stazioni in attesa del risultato. In questa fase le store scrivono in memoria e le load prelevano il dato.

Notiamo che i registri interni delle stazioni di prenotazione svolgono il compito di registri temporanei, effettuando una ridenominazione dei registri che risolve il problema delle dipendenze sui nomi.
Riportiamo un esempio di funzionamento dello schema di Tomasulo

Esempio 3 : supponiamo di voler eseguire il seguente codice:
$LD\ F6,\ 34(R2)$
$LD\ F2,\ 45(R3)$
$MUL\ F0,\ F2,\ F4$
$SUB\ F8,\ F2,\ F6$
$DIV\ F10,\ F0,\ F6$
$ADD\ F6,\ F8,\ F2$

Supponiamo di aver terminato l'esecuzione della prima load, e che la nostra architettura possieda: due unita' funzionali per le load, tre unita' funzionali per le addizioni/sottrazioni e due per le moltiplicazioni/divisioni. Le reservation station saranno occupate come segue:

Staz.	busy	Op	Qj	Qk	A
Load1	no				
Load2	yes	LD			$45 + Reg[R3]$
Add1	yes	SUB	Load2		
Add2	yes	ADD	Add1	Load2	
Add3	no				
Mult1	yes	MUL	Load2		
Mult2	yes	DIV	Mult1		

Dove i campi sono:
Staz e' il nome della stazione
Busy indica se la stazione e' occupata
Op indica l'operazione che verra' eseguita
Qj e Qk indicano le stazioni che possiedono gli operandi necessari alla stazione
A e' un campo particolare per le load/store che indica prima il valore immediato per la load/store e dopo l'indirizzo effettivo della RAM. Notiamo come questo schema risolva gli hazard WAW e WAR. Ad esempio il WAR tra la DIV e la ADD viene risolto con una ridenominazione dei registri. Infatti la DIV non preleva piu' il secondo operando da F6, bensi' da Vk e quindi la terminazione di ADD prima di DIV non crea problemi.

Esempio 4 : supponiamo di avere il seguente codice

$Loop:\ LD\ F0,\ 0(R1)$

$MUL\ F4,\ F0,\ F2$
$SD\ F4,\ 0(R1)$
$ADD\ R1,\ R1,\ -8$
$BNE\ R1,\ R2,\ Loop;$

Supponiamo di ignorare la ADD e di supporre che il salto venga sempre eseguito. Notiamo che in tal modo abbiamo sufficienti unita' funzionali per poter eseguire in contemporanea due iterazioni del loop!
Questa tecnica e' nota come *srotolamento dinamico*(o *dynamic unrolling*) e puo' fornire dei vantaggi in termini di tempo di esecuzione. Dobbiamo pero' fare attenzione all'ordine delle istruzioni LOAD STORE dei vari cicli. Infatti potremmo avere un hazard RAW se LOAD e STORE appartengono a cicli diversi e inoltre la LOAD viene dopo la STORE ma le due istruzioni vengono eseguite in ordine inverso. Potremmo invece avere un hazard WAR se la STORE viene dopo la LOAD ma le due istruzioni vengono eseguite in ordine inverso. Potremmo infine avere un hazard WAW se abbiamo due STORE eseguite in ordine inverso.
Per evitare questo problema la CPU deve verificare che i registri delle due istruzioni incriminate non siano in comune.

3.2 Branch prediction Dinamico

Ad ogni modo implementare uno schema di pipelining dinamico non e' facile e un'altra miglioria di cui abbiamo parlato e' dovuta ai branch. Una pipeline schedulata dinamicamente può fornire prestazioni molto elevate purché i salti vengano predetti in modo accurato.
Come gia' detto cerchiamo di fare un qualche tipo di predizione sul branch. Nel caso la predizione sia errata svuotiamo la pipeline e ricominciamo. Chiaramente in caso di errore lo spreco di cicli di clock e' tanto maggiore quanto piu' profonda e' la pipeline.
Per tutti questi motivi i processori moderni adottano delle tecniche di branch prediction dinamico, in cui l'esito di un branch viene predetto sulla base delle sue precedenti esecuzioni.
Solitamente viene usato un *branch prediction buffer* che contenga gli indirizzi delle istruzioni di branch. Ad ogni entry del buffer associamo un bit di predizione che indica se la volta precedente il salto era stato preso o meno.
Il problema di questo approccio e' che sbaglia piu' spesso di quanto si pensi, infatti immaginiamo di avere un loop, dopodiche' usciamo e rientriamo nel loop piu' tardi. Il bit singolo sbaglia due volte, sia quando usciamo, sia quando rientriamo.
Per evitare cio' si usa uno schema di predizione a 2 bit. In questo schema consideriamo una predizione sbagliata(e dunque cambiamo il valore di predizione) se sbagliamo a predire per due volte consecutive. Questo schema puo' essere riassunto con un'automa a 4 stati mostrato in Figura 9.

Questo schema, sebbene piu' complesso, porta a un miglioramento sensibile della performance. Visto che due bit predicono meglio di uno, potremmo

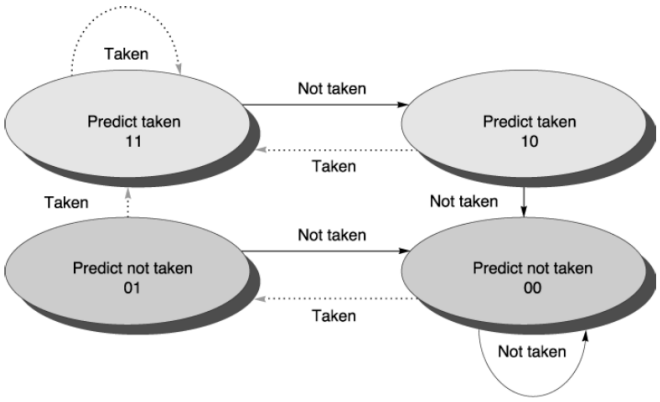


Figura 9: Schema della predizione a due bit

pensare di usare tre bit. In realta' aggiungere un nuovo bit non aiuta quanto potremmo sperare.
Inoltre l'efficacia dello schema dipende anche dalla dimensione del branch prediction buffer. E' stato mostrato statisticamente che una dimensione del buffer a 4096 entry e' ottimale. Una dimensione minore porta a un degrado della performance. Una dimensione maggiore non porta ulteriori benefici.

Vengono infine utilizzate due altre tecniche che migliorano ulteriormente la performance:

- Schema a predittori correlati: combina i predittori a due bit di piu' salti consecutivi andando cosi' ad analizzare il comportamento dei branch circostanti
- Schema a torneo: ogni salto ha due predittori diversi, a uno e a due bit. Ogni volta viene usato quello che si e' comportato meglio la volta precedente

Fin'ora abbiamo pero' ignorato un aspetto importante. Se un salto deve essere preso, non sappiamo immediatamente quale sia la l'indirizzo a cui spostare il program counter. Infatti dovremmo eseguire un instruction fetch...
Per questo motivo viene utilizzato un *branch target buffer*, ovvero una memoria che associa ad ogni branch l'indirizzo a cui trasferire il controllo nel caso in cui il branch venga eseguito.

3.3 Speculazione hardware

Come abbiamo gia' detto, quando la predizione di un salto e' sbagliato dobbiamo annullare le istruzioni che si trovano subito dopo il branch e che abbiamo erroneamente inserito nella pipeline. Consideriamo pero' il seguente codice:

$LD\ F4,\ 100(R4)$
$BLE\ F4,\ \#0.66666, jump$
$FADD\ F1,\ F1,\ \#0.5$
$DADD\ R1,\ R1,\ \#2$
$jump:\ FADD\ F1,\ F1,\ \#0.25$
$DADD\ R1,\ R1,\ \#1$

Nel caso in cui il valore caricato dalla load non sia in cache, potrebbero volerci decine di cicli di clock a recuperare il dato dalla RAM. In tal caso potremmo eseguire le istruzioni subito dopo il branch effettuando una speculazione.
Questa tecnica pero' introduce un problema. Infatti la $FADD\ F1,\ F1,\ \#0.25$ verra' sempre e comunque eseguita. Pero' tale istruzione utilizza valori modificati dalle due istruzioni precedenti che non sappiamo se verranno eseguite. Per questo motivo anche questa $FADD$ deve essere considerata un'istruzione speculativa.
Per questo motivo introduciamo un nuovo componente nello schema di Tomasulo: il *Reorder Buffer*(ROB).
Il ROB e' una speciale unita' in cui vengono temporaneamente sistemate le istruzioni in attesa di sapere se debbano essere effettivamente eseguite. Quando la CPU ne viene a conoscenza o le annulla o le committa.
Anche se le istruzioni possono essere eseguite out of order, devono essere committate in order. Per questo motivo il ROB e' ordinato come una coda FIFO. Ogni entry nel ROB possiede quattro campi: il *tipo di istruzione*(branch, ALU...), il campo di *destinazione*(la locazione del registro che viene modificato dall'istruzione), il *valore* che memorizza temporaneamente il risultato dell'esecuzione fino al commit, e infine il campo *ready* che segnala quando l'istruzione ha terminato la fase EX e il contenuto del campo valore e' valido. Le fasi di esecuzione di un'istruzione in questa nuova architettura sono adesso quattro.

1. Issue: dopo la IF e la ID l'istruzione I viene avviata ad una stazione di prenotazione e inserita in fondo al ROB. Se disponibili gli operandi di I vengono prelevati da uno dei registri, da un'altra stazione o dal ROB. Infine la locazione del ROB che contiene I viene scritta nella stazione di prenotazione di I per sapere quale locazione modificare del ROB successivamente.

2. Execute: invariata

3. Write Result: quando il risultato di I e' pronto viene scritto in memoria e inoltrato a chi lo aspettava. Tale risultato viene anche inserito nel ROB

4. Commit: se l'istruzione non e' di branch o e' un branch predetto correttamente, il contenuto del campo valore e' trasferito in memoria o in RAM e l'istruzione viene rimossa dal ROB. Se l'istruzione e' un branch con predizione sbagliata tutto il ROB viene svuotato e la computazione e' riavviata dall'istruzione corretta

3.4 Multiple Issue

Abbiamo parlato in precedenza delle caratteristiche strutturali di un processore multiple issue. Se possiede tali caratteristiche allora e' in grado di avviare piu' di una istruzione per ciclo di clock. Possiamo dunque pensare ad un processore superscalare come ad un processore che possieda piu' pipeline operanti in parallelo.
Notiamo pero' che in assenza di scheduling dinamico della pipeline il multiple issue e' fortemente limitato infatti un'istruzione C indipendente da due istruzioni A e B precedenti subisce comunque lo stall creato da A e B. Dunque diciamo subito che usare il multiple issue senza avere una schedulazione dinamica e' poco conveniente. Infatti processori multiple issue a scheduling statico avviano di solito non piu' di due istruzioni per volta.

Vediamo come funziona un processore multiple issue: le istruzioni vengono prelevate dalla IM e poste in una Instruction Queue(IQ) che viene analizzata dalla CU per rilevare le dipendenze. Quando la CU rileva istruzioni indipendenti fra loro le invia verso le stazioni di prenotazione per eseguirle. Solitamente le istruzioni spostate dalla IQ alle stazioni sono meno di quelle prelevate dalla IM(controllare le alee e' un'operazione piu' costosa di un semplice prelievo dalla IM). Per questo motivo la IQ tende a riempirsi. Quando questo succede non si prelevano istruzioni dalla IM fino a quando la IQ non possiede dello spazio libero.

Una domanda che potrebbe essere naturale porsi e': come mai usiamo delle tecniche dinamiche anche in casi tanto complessi? Non potremmo spostare queste operazioni in fase di compilazione? La risposta e' no per tre motivi:

- I miss cache non sono prevedibili staticamente
- I salti non sono prevedibili staticamente
- L'ILP statico funziona bene solo su una specifica architettura, quello dinamico no.

Ad ogni modo anche l'ILP dinamico ha i suoi problemi. Infatti limitando il numero di istruzioni prese in considerazione per valutare le dipendenze, l'ILP dinamico diventa molto meno sfruttabile.
Per questo motivo ad oggi vengono effettuate delle ricerche ad esempio sul *value prediction*, ovvero cercare di predire il risultato di un'istruzione. È chiaro che conoscendo a priori il risultato di una istruzione potremmo passarlo ad altre istruzioni in attesa eliminando le dipendenze. Chiaramente una tecnica di questo tipo viene considerata solo se abbiamo buone possibilita' di valutare correttamente il risultato.

3.5 Storia dell'ILP dinamico

La storia dell'ILP dinamico comincia con la microarchitettura Intel P6. Uno degli aspetti fondamentali di tale serie e' la backward compatibility. Questa

caratterstica e' stata mantenuta con grande sforzo progettuale, rendendo l'architettura poco elegante.
L'architettura P6 viene introdotta con il Pentium Pro che a differenza delle architetture precedenti(basate su piu' pipeline), decide di usare una struttura superscalare. I registri visibili dell'architettura P6 erano: un'insieme di registri general purpose, un'insieme di registri residuati da architetture precedenti e spesso inutilizzati, piu' otto registri floating point. L'architettura del P6 per la prima volta cerca di andare incontro alla filosofia RISC anche se e' ancora basata su istruzioni IA-32 tipicamente CISC che necessitano di essere tradotte in microistruzioni risc a 72 bit, dette *uops*. La maggior parte delle istruzioni IA-32 possono essere tradotte in 1-4 uops. Quelle che richiedono piu' uops vengono eseguite mediante microcodice memorizzato in una ROM. Ad ogni ciclo fino a 3 istruzioni IA-32 vengono prelevate e fino a 6 uops sono generate in ogni ciclo. La scheduazione delle uops e' dinamica e specula utilizzando un ROB. Inoltre fino a tre istruzioni possono eseguire il commit(graduation nella terminologia Intel). La pipeline del P6 e' divisa in 14 stage:

1. 8 stage per il fetch e la decodifica delle istruzioni
2. 3 stage per l'esecuzione out of order
3. 3 stage per il commit

I valori medi riscontrati utilizzando una architettura P6 sono:

- 1 uop commited per ciclo
- 1.37 uops per eseguire una istruzione IA-32
- 1.15 cicli di clock per completare un'istruzione IA-32
- 2 cicli di clock per completare un'istruzione IA-32 di un programma che operi su floating point

L'evoluzione dell'architettura P6 e' l'architettura NetBurst. Questa nuova architettura rompe rispetto alla P6 perche' riduce sensibilmente il ciclo di clock ottenendo un processore estremamente veloce ma con consumi molto alti e problemi di dissipazione. Usa una pipeline con almeno 20 stage.
Le differenze principali con l'architettura precedente sono: ciclo di clock piu' corto, la speculazione viene fatta con una tecnica diversa dal ROB che permette di avere fino a 128 istruzioni uop in attesa di commit, sono presenti piu' unita' funzionali per il calcolo. In particolare le ALU lavorano a velocita' doppia di quella del ciclo di clock. Inoltre viene usata una *trace cache* come cache L1, e una cahce L2 di dimensione maggiore. Ulteriori differenze stanno nell'incremento della dimensione del branch target buffer e l'introduzione dell'*hyper threading*.

Ad ogni modo il successo di NetBurst e' controverso perche' ad un aumento della frequenza di clock non e' corrisposto linearmente un aumento delle prestazioni. Si e' infatti verificato che con un aumento del 70% della frequenza di

clock, le prestazioni migliorano solo del 26%.
A questo dobbiamo aggiungere il problema degli elevati consumi dell'architettura NetBurst. Per questo motivo dal 2005 Intel cerca di proporre una versione aggiornata dell'architettura P6.
Nel gennaio del 2006 viene commercializzato il primo Core Duo a 65 nanometri e dual core. A fine luglio 2006 fa la comparsa il Core 2 Duo che segna il definitivo ritiro del brand Pentium. Per marcare tale rottura la microarchitettura del Core 2 Duo cambia nome diventando la *Intel64*. La pipeline e' a 14 stadi come nelle precedenti architetture P6 ma consente di prelevare quattro istruzioni IA-32, dicendo contemporaneamente addio all'hyper threading, ritenuto il responsabile degli elevati consumi dell'architettura NetBurst. I bassi consumi di questa nuova architettura derivano dallo "spegnimento" di alcune porzioni del datapath non usate durante l'esecuzione delle istruzioni.
Inoltre, nel 2007/2008 fanno la prima comparsa le architetture a 45 nanometri. Dalle architetture Core 2 Duo viene usato un nuovo set di istruzioni chiamate Intel64(da non confondere con l'IA-64). Questo nuovo set di istruzioni prevede una modalita' compatibile per le applicazioni a 32 e a 16 bit e ovviamente la possibilita' di far girare applicativi a 64 bit.

Visto che nel seguito parleremo spesso di architetture *a n nanometri*, diamone una definizione piu' precisa. Il valore n rappresenta la semidistanza tra due celle di memoria. Dunque tanto piu' e' piccolo, quante piu' componenti riesco a inserire in un chip.

Nella seconda meta' del 2008 fa la sua comparsa la microarchitettura Nehalem, ancora a 45 nanometri. Ma dal 2010 con la nuova architettura Westmere passiamo ai 32 nm. In questo contesto i processori vengono divisi in famiglie e prendono i nomi di *i3, i5, i7* a seconda del livello di prestazioni e della quantita' di cache.
Queste due nuove architetture reintroducono l'hyperthreading e un branch prediction basato su piu' predittori a 2 bit. Viene inoltre implementata una Translation Lookaside Buffer a due livelli. Una ulteriore novita' e' che introduce un terzo livello di cache. Grazie a tutte queste scelte i consumi si riducono del 30% e le prestazioni aumentano del 10%-25%.

Grazie a quanto visto finora e' possibile notare la strategia produttiva biennale della Intel detta "tick-tock".

- Tick: viene sviluppato un nuovo processo produttivo(meno nanometri, piu' transistor)
- Tock: viene sviluppata una nuova microarchitettura migliorata e basata sugli sviluppi dell'anno precedente

Nel 2011 viene introdotta una nuova architettura detta Sandy Bridge a 32 nm che riduce i tempi di accesso alla cache e integra nella CPU la GPU. Secondo Intel Sandy Bridge fornisce mediamente il 17% in piu' di prestazioni rispetto

alle architetture precedenti.
Nel 2012 viene introdotta l'architettura Ivy Bridge sostanzialmente identica alla Sandy Bridge ma con tecnologia a 22nm che migliora le prestazioni del 5%-15% e le prestazioni della GPU del 25%-65%.

Nel giugno 2013 Intel presenta la nuova architettura Hashwell sempre a 22nm che pero' differisce molto dalla Ivy Bridge. Ad un aumento delle prestazione del 3%-6% corrisponde un aumento dei consumi dell'8%.
Dal 2014 viene commercializzata la versione a 14nm della Haswell, detta Broadwell che riduce i consumi e lo spazio rispetto alla Hashwell.

Vediamo un po' piu' nel dettaglio il processore core i7. L'esecuzione e' out of order e la pipeline e' a 14 stadi che possiamo suddividere nelle seguenti fasi:

1. Fetch: un branch predictor a piu' livelli preleva 16 byte dalla cache delle istruzioni

2. I 16 byte vengono pre-decodificate e le istruzioni che possono essere fuse in un unica istruzione vengono fuse. Tale operazione e' detta *macro-op fusion*

3. Le istruzioni vengono tradotte in uops, ne vengono prodotte massimo quattro per ogni ciclo di clock e inserite nel *loop stream detect buffer*

4. All'interno del loop stream detect buffer vengono cercate piccole sequenze di uops che formano un loop, di modo che possano essere avviate direttamente dal buffer per tutta la durata del loop

5. Avviene l'esecuzione delle uops con uno schema simile a quello di Tomasulo(basato su reservation station e speculazione).

Veniamo invece alle architetture AMD. Nel 2007 viene presenta l'architettura AMD K10 su cui si baseranno varie architetture a 65nm prima, e a 45nm dopo. Nella AMD K10 vengono avviate 3 istruzioni per ciclo di clock e abbiamo anche in questo caso 3 livelli di cache.
Il processore Opteron e' basato proprio sull'architettura K10: e' compatibile con le istruzioni x86 visto che le traduce in un'insieme di istruzioni RISC-like simili alle uops della Intel.
La pipeline e' a 12 stadi(17 in caso di floating point) e per la speculazione viene usato un ROB.

All'inizio del 2011 AMD passa alla tecnologia a 32nm includendo nello stesso chip la CPU e la GPU. Come gia' visto per Intel, il risparmio energetico viene raggiunto spegnendo la parte del datapath che non viene utilizzata.
A fine 2011 viene commercializzata una nuova architettura detta bulldozer, ancora a 32nm la cui caratteristica piu' interessante consiste nell'avere due unita' integer per ogni bulldozer, rendendolo un chip dual core. Questa scelta architetturale e' la risposta alla reintroduzione dell'hyperthreading di Intel. Inoltre tale

scelta si basa sull'idea che nell'ambito server/workstation l'80% del lavoro sia svolto con operazioni su interi, fornendo quando opera solo su interi un aumento di prestazioni dell'80% rispetto ad una architettura multithread.
Tra il 2012 e il 2015 AMD rilascia tre architetture basate su bulldozer che portano la tecnologia a 28nm.
Verso la fine del 2016 AMD introduce una nuova architettura detta ZEN che migliora del 52% il numero di istruzioni eseguite per ciclo di clock e che porta la tecnologia a 14nm.

A questo punto riflettiamo un attimo su una scelta comune alle architetture Intel e AMD: integrare in un unico chip la CPU e la GPU. Questa scelta e' dettata dall'esigenza di operare su numeri floating point. Storicamente la GPU e' sempre stata una componente abbastanza rigida con un numero limitato di operazioni(texture mapping, rotazioni, traslazioni) eseguite in parte al livello hardware.
Tuttavia, recentemente le GPU sono diventate piu' flessibili e possono anche essere utilizzate per operare sui numeri in virgola mobile. Chiaramente passare alla GPU i calcoli in virgola mobile da eseguire puo' convenire solo nel caso in cui CPU e GPU risiedano sullo stesso chip.

Vediamo infine altri due grandi nomi nell'ambito dei processori: ARM e SUN.
La ARM e' una multinazionale di processori RISC che vende processori embedded molto diffusi. Nel solo 2010 ha venduto piu' di 6 miliardi di processori e il 95% degli smartphone utilizza un processore ARM. Nel 2012 ha venduto il 75% dei processori a 32 bit nel mondo. Noti processori ARM sono i processori della seria A della Apple.
Il processore A8(iPhone, iPad, Motorola) e' una classica architettura RISC con due pipeline a 13 stadi. Alcune istruzioni possono essere eseguite su entrambe le pipeline, altre(come la MUL) solo su una delle due pipeline. I maggiori problemi prestazionali nel processore A8 sono dovuti ad hazard funzionali e sui dati causati da uno scheduling statico della pipeline. Per questo motivo viene introdotto l'A9 che fornisce uno scheduling dinamico capace di migliorare del 28% le prestazioni.
Un'altra grande casa di produzione di processori e' la SUN che nel 2010 lancia il SUN SPARC T3, un processore multicore(16 core) in cui ogni core e' capace di eseguire fino a 8 thread in parallelo. La sua architettura e' basata su una pipeline a 8 stadi, esecuzione in order, possibilita' di avviare 4 istruzioni per ciclo di clock.
Nel 2011 viene lanciato il T4 che aumenta la frequenza di clock e introduce l'esecuzione out of order. Nel 2013 viene commercializzato il T5 che porta la tecnologia a 28nm e aumenta ulteriormente la frequenza di clock.

Concludiamo riportando qualche riflessione rispetto allo sviluppo delle architetture:

- A parita' di cache e di unita' funzionali disponibili, l'efficienza computa-

zionale single core puo' essere migliorata aumentando il ciclo di clock o aumentando il numero di istruzioni avviate per ciclo di clock(ILP).

- La tendenza e' quella di creare CPU sempre piu' veloci invece che migliorare l'ILP. La ragione fondamentale e' che un aumento lineare dell'ILP porta ad un aumento quadratico della dimensione dei circuiti, impattando sensibilmente anche sui consumi.
  Per questo si preferisce aumentare il multi threading, o migliorare il sistema di caching oppure sfruttando soluzioni architetturali nuove(integrazione di GPU...)

- Ad ogni modo tutte le architetture moderne implementano una qualche forma di ILP. È comunque interessante notare che molte delle idee sull'ILP nascono verso la meta' degli anni '60 e negli anni successivi, ma a quei tempi non esisteva la tecnologia per poterle sfruttare. Ad esempio negli anni '70 nascono i primi sistemi di branch prediction a due bit, ma anche i primi processori multiple-issue. Negli anni '80 invece emergono i primi processori multiple-issue dinamici. Tutte queste idee verranno messe insieme correttamente solo verso la fine degli anni '80, inizio anni '90.

4 ILP statico

Nel capitolo precedente abbiamo visto alcune tecniche per aumentare il parallelismo tra le istruzioni dinamicamente. Vediamo invece quali ottimizzazioni possiamo fare a tempo di compilazione, ovvero staticamente.

4.1 Riordinare le istruzioni

Una tecnica fondamentale e' quella di riordinare le istruzioni macchina in modo da eliminare o quanto meno ridurre gli stall. Naturalmente la manipolazione deve lasciare inalterato il funzionamento del programma.
Un modo per ridurre l'impatto degli stall, consiste infatti nel "tenere impegnata" la pipeline con altre istruzioni indipendenti mentre viene risolta la prima operazione della catena di dipendenze. Questa forma di scheduling dipende: dalla quantita' di ILP presente nel programma, dalla lunghezza delle pipeline e dal fatto che il compilatore conosca le informazioni della CPU per cui sta generando il codice.

Esempio 1 : supponiamo di voler eseguire il seguente codice:

$for(i = 1000; i > 0; i = i - 1)x[i] = x[i] + s;$
Che tradotto in codice MIPS diviene:

$LOOP:\ LD\ F0,\ 0(R1)$
$FADD\ F4,\ F0,\ F2$

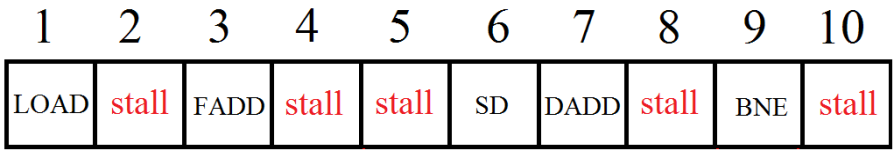


Figura 10: Esecuzione di un'iterazione del loop

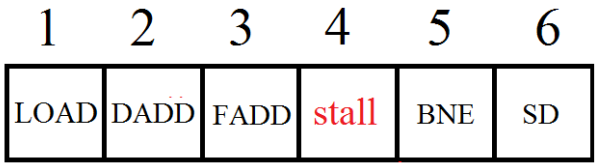


Figura 11: Esecuzione ottimizzata

$SD\ F4,\ 0(R1)$
$DADD\ R1,\ R1,\ \#-8$
$BNE\ R1,\ R2,\ LOOP$

L'esecuzione di tale codice MIPS viene mostrato in Figura 10.

Notiamo che abbiamo impiegato 10 cicli per eseguire una singola iterazione.
Un compilatore intelligente potrebbe riordinare le istruzioni come segue:

$LOOP:\ LD\ F0,\ 0(R1)$
$DADD\ R1,\ R1,\ \#-8$
$FADD\ F4,\ F0,\ F2$
$BNE\ R1,\ R2,\ LOOP$
$SD\ F4,\ 8(R1)$

In questo modo abbiamo eliminato gli stall tra la LOAD e FADD, tra DADD e BNE. Spostando inoltre la SD dopo la BNE abbiamo eliminato lo stall della BNE.
Notiamo che comunque questa forma di scheduling e' possibile solo nel caso in cui il compilatore conosca la latenza in cicli di clock di ogni istruzione che genera. L'esecuzione ottimizzata e' mostrata in Figura 11.

Notiamo che il costo di un'iterazione si e' ridotto notevolmente passando da 10 a 6 cicli di clock.

4.2 Loop unrolling

Un'altra tecnica che avevamo gia' nominato e' quella del loop unrolling. Supponendo che il numero di volte che dobbiamo eseguire un certo loop sia multiplo

di un certo valore n, allora potremo "srotolare" il contenuto del loop n volte per ogni iterazione del loop ed eseguire il loop $1/n$ volte

Esempio 2 : riprendiamo l'esempio visto poco fa. Essendo eseguito 1000 volte il loop, potremmo pensare di riscrivere il corpo del loop per quattro volte di seguito e poi andare ad eseguire il loop 250 volte soltanto. In questo caso il risultato sarebbe di 28 cicli di clock per un'iterazione del loop, che pero' contiene quattro volte il corpo del for. Dunque il costo sarebbe $28/4 = 7$ cicli.

Un modo per migliorare notevolmente la situazione consiste nello scrivere il codice del loop srotolato come segue:

$LD\ F0,\ 0(R1)$
$LD\ F6,\ -8(R1)$
$LD\ F10,\ -16(R1)$
$LD\ F14,\ -24(R1)$
$FADD\ F4,\ F0,\ F2$
$FADD\ F8,\ F6,\ F2$
$FADD\ F12\,F10,\ F2$
$FADD\ F16,\ F14,\ F2$
$SD\ F4,\ 0(R1)$
$SD\ F8,\ -8(R1)$
$DADD\ R1,\ R1,\ \#-32$
$SD\ F12,\ 16(R1)$
$BNE\ R1,\ R2,\ LOOP$
$SD\ F16,\ 8(R1)$

In questo modo, per eseguire quattro volte il corpo del loop impieghiamo 14 cicli di clock. Quindi $14/4 = 3.5$ cicli per ogni corpo del ciclo originario! Notiamo che abbiamo anche effettuato delle operazioni di ridenominazione dei registri per poter eseguire immediatamente tutte le load.

Ovviamente ottimizzazioni di questo tipo si pagano con un tempo maggiore per generare il codice in fase di compilazione. Ribadiamo che comunque queste tecniche prevedono che il compilatore sia a conoscenza delle caratteristiche della macchina su cui lavora. Inoltre vi sono alcuni problemi da considerare: quanti nomi di registri diversi si possono usare nel loop unrolling? Come gestire loop che non so quante volte verranno eseguiti? Inoltre dobbiamo considerare che il loop unrolling genera un codice piu' lungo dell'originale. Estremizzare questa tecnica porta quindi a un riempimento della cache, aumentando la probabilita' di cache miss.

4.3 Multiple Issue statico

Ad ogni modo, una miglioria che possiamo comunque cercare di applicare e' l'introduzione del multiple issue statico. Con questa tecnica il compilatore ge-

	1	2	3	4	5	6	7	8	9
Memory1	LOAD	LOAD	LOAD	LOAD	no-op	SD	SD	SD	SD
Memory2	LOAD	LOAD	FADD	no-op	no-op	SD	SD	SD	no-op
Floating1	no-op	no-op	FADD	FADD	FADD	FADD	no-op	no-op	no-op
Floating2	no-op	no-op	FADD	FADD	FADD	no-op	no-op	no-op	no-op
Integer/ Branch	no-op	no-op	no-op	no-op	no-op	no-op	DADD	no-op	BNE

Figura 12: Esecuzione che fa uso del multiple issue

nera una serie di pacchetti detti issue packet contenenti un certo numero di istruzioni indipendenti tra loro. Se in un dato momento non riesce a riempire completamente l'issue packet, vi inserisce l'istruzione speciale no-op che non effettua alcuna modifica.
Essendo una tecnica statica il contenuto dei pacchetti viene determinato prima dell'esecuzione ed e' quindi responsabilita' del compilatore stabilire l'ordine adatto delle istruzioni per massimizzare le prestazioni. Stabilire staticamente quali siano i pacchetti di istruzioni ha anche un altro vantaggio: ovvero che la CPU ha una architettura piu' semplice poiche' non deve controllare a run-time tutte le dipendenze fra le istruzioni visto che tale lavoro e' gia' stato fatto in parte dal compilatore.
Questo approccio viene spesso indicato col nome VLIW (Very Long Instruction Word) dal fatto che le prime architetture che lo adottavano usavano istruzioni molto lunghe, ognuna delle quali specificava piu' operazioni fra loro indipendenti.
Quanto più è alto il numero di istruzioni che un processore può avviare contemporaneamente, tanto più l'approccio VLIW è vantaggioso. Infatti l'overhead che un processore superscalare deve sopportare per i confronti a run-time cresce quadraticamente con il numero di istruzioni che deve controllare.

Esempio 3 : consideriamo un processore che possa avviare 5 istruzioni contemporaneamente con le seguenti unita' funzionali: una ALU per interi e per i branch, due unita' floating point, due unita' per i riferimenti in memoria.Consideriamo allora l'esempio gia' visto, di cui pero' facciamo un unrolling di 7 corpi del ciclo. L'esecuzione in questo caso viene mostrata in Figura 12

Notiamo che non tutti gli slot sono occupati. Questo e' dovuto alle dipendenze e mette in evidenza che non sempre e' possibile sfruttare a pieno il

multiple issue. Ad ogni modo per eseguire 7 iterazioni del for originario abbiamo impiegato 9 cicli di clock. Dunque $9/7 = 1.29$ cicli di clock per ogni iterazione del for originario!

Naturalmente e' anche possibile implementare tecniche piu' avanzate come:

- Static branch prediction: assumere che i branch si comportino sempre in uno stesso modo
- Loop level parallelism: evidenziare il parallelismo in iterazioni successive di un loop quando i cicli non sono indipendenti fra loro
- Symbolic loop unrolling: il loop non viene srotolato ma si cerca di mettere in ogni pacchetto istruzioni indipendenti tra loro anche appartenenti a cicli diversi
- Global code scheduling: compattare il codice proveniente da diversi blocchi di istruzioni condizionali

4.4 Istruzioni predicative

Le tecniche viste finora funzionano bene quando il comportamento dei branch e' facilmente predicibile. Quando il comportamento non e' ben chiaro le dipendenze dal controllo limitano la possibilita' di usare il parallelismo tra istruzioni. Per questo vengono introdotte le *istruzioni predicative* o *condizionali*.
Una istruzione predicativa e' un particolare tipo di istruzione formato da una parte condizionale e da una seconda parte che viene eseguita solo se la parte condizionale e' verificata. Quando la parte condizionale non e' verificata, l'istruzione viene trasformata in una no-op. Un esempio di istruzione di questo tipo e' la move condizionale. Vediamone un esempio

Esempio 4 : consideriamo l'istruzione $if(A == 0)S = T$. Il codice convertito normalmente sarebbe:

$BNEZ\ A,\ Jump$ // branch if not zero
$ADD\ S,\ T,\ R0$ // R0 always contains 0
$Jump$: *doSomething*

Usando la move condizionale possiamo tradurre come segue il nostro codice:

$CMOVZ\ S,\ T,\ A$ // mov R3 in R2 if R1 == 0

In questo modo la dipendenza sul controllo viene trasformata in un'eventuale dipendenza sui dati.

Eliminare questi tipi di branch puo' avere un impatto notevole sulle pipeline reali, in cui l'esecuzione del branch puo' richiedere molti cicli di clock. Infatti

i branch non sono facilmente predicibili e le istruzioni condizionali ci aiutano ad ovviare al problema. Notiamo che i branch possono diventare ancora piu' difficili da prevedere se dipendono gli uni dagli altri. Per questo motivo molte architetture non permettono di avviare piu' branch in un unico ciclo di clock.

Chiaramente con le move condizionali non eliminiamo tutti gli altri branch che non coinvolgano una move. Per questo, andando ad estremizzare il concetto di istruzione condizionale, alcune architetture implementano la *full predication*: tutte le istruzioni possono essere controllate da un predicato. Per funzionare correttamente, tali architetture usano dei *registri predicativi* ad un bit che memorizzano l'esito delle esecuzioni precedenti.
Nel complesso le istruzioni predicative:

- sono particolarmente utili per implementare piccoli if-then-else eliminando branch difficilmente predicibili
- devono comunque essere usate con criterio poiche' le istruzioni predicative annullate hanno comunque consumato risorse della CPU
- non sono comode da usare per combinazioni complesse di branch
- possono essere piu' lente delle corrispondenti istruzioni non condizionali. Per questo molte architetture implementano soltanto alcune semplici istruzioni condizionali, in particolare la move condizionale

4.5 Due famiglie di processori

Anche se inizialmente ILP statico e dinamico erano molto diversi fra loro, ad oggi questa differenza non e' piu' cosi' marcata. Bisogna comunque dire che i due tipi di ILP sono ancora pensati per scopi differenti: ILP dinamico sembra essere piu' adatto a processori general purpose, mentre quello statico sembra comportarsi meglio su processori che operano su applicazioni embedded in cui sono piu' importanti i consumi e dove i programmi da usare sono limitati.
Vediamo due famiglie di processori che vanno incontro a questi due approcci

- ILP dinamico, Itanium 2: insieme al suo predecessore e' la prima architettura Intel completamente RISC. Sfrutta comunque diversi principi statici che abbiamo visto: usando registri predicativi e andando a creare dei blocchi di istruzioni indipendenti detti instruction groups. La CPU e' libera di eseguire le istruzioni all'interno di un group come preferisce, ma diversi group vengono eseguiti in maniera sequenziale, separati da un'istruzione speciale di stop.
  La differenza principale con quanto abbiamo detto finora consiste in una maggiore flessibilita' visto che in realta' le istruzioni nell'Itanium sono organizzate in bundle da 128 bit: ogni bundle contiene tre istruzioni anche appartenenti a instruction groups diversi.
  Oltre alla full prediction l'Itanium supporta un'altra tecnica detta *advanced load*. L'idea e' la seguente: visto che le load possono impiegare molti

cicli di clock ad essere eseguite, in particolare se devono prelevare dati dalla RAM, potremmo pensare di anticipare le load il piu' possibile, di modo che se il dato non fosse presente in cache, si possa avere il tempo di recuperare i dati dalla RAM. L'idea e' buona ma cosa succede se anticipando la load la spostiamo prima di una store su quello stesso registro? È per questo che ogni volta che una load viene anticipata questo fatto viene segnalato con una entry in una speciale tabella. Quando una store deve essere eseguita controlla in tale tabella se una load sullo stesso registro era stata anticipata. In tal caso la load viene eseguita nuovamente.
Ad ogni modo il progetto Itanium e' controverso visti i costi elevati di un processore tanto complesso e la sua diffusione dedicata solo a progetti di fascia alta.

- ILP statico, TriMedia TM32: esempio di processore usato per applicazioni multimediali in lettori mp3, dvd, cd...
  Ha un clock relativamente basso e puo' avviare fino a 5 istruzioni per ciclo di clock. Supporta la predicazione di tutte le istruzioni e non esegue controlli di dipendenze a run-time, delegando tali compiti al compilatore. L'unico vero difetto di questo approccio e' che il codice generato occupa comunque molto piu' spazio di un codice equivalente RISC, e anche se viene applicata effettivamente una compressione del codice, occupa circa il doppio dello spazio di un generico codice RISC.

5 Caching

La necessita' di avere una memoria cache nasce dal fatto che da sempre la memoria principale e' piu' lenta della CPU, e tale differenza di prestazioni si e' acuita nel tempo.
Ad esempio un core i7 puo' generare riferimenti per 409.6 Gbyte al secondo. Per contro nello stesso tempo la RAM puo' fornire fino a 25 Gbyte di dati/istruzioni al secondo. Dovrebbe quindi essere chiaro che e' inutile progettare processori sempre piu' veloci se poi non possiamo reperire le informazioni sufficientemente in fretta.
Intanto dobbiamo differenziare tra DRAM e SRAM:

- DRAM(Dynamic RAM): indica la tecnologia con cui sono costruite le RAM usate nella memoria principale. Usano un condensatore che deve essere ricaricato ogni pochi millisecondi
- SRAM(Static RAM): indica la tecnologia con cui sono costruite le RAM usate nei vari livelli di cache. Usano un circuito formato da transistor che devono essere sempre alimentati. In generale consumano di piu' sono piu' complesse e costose, ma hanno tempi di risposta minori.

La piu' grande differenza tra la cache e la RAM e' che i dati di quest'ultima devono percorrere il BUS di sistema, e tale operazione puo' richiedere diversi cicli di clock.

Se ci pensiamo il concetto di caching e' estendibile a tutti i supporti di memoria visto che: i registri del processore fanno da cache per la cache hardware, la cache hardware fa da cache per la RAM, la RAM fa da cache per il disco rigido...
L'idea di cache nasce in contemporanea all'architettura di Von Neumann e l'idea fondamentale che vi sta dietro e': se una buona percentuale di dati e istruzioni sono presenti in cache, l'effetto della lentezza della RAM si riduce di molto. Questo meccanismo posa le sue basi sui *principi di localita'*:

- Localita' spaziale: quando il processore accede ad una locazione di memoria i è probabile che nel breve periodo debba accedere anche alle locazioni $i+1, i+2\ldots$
  Questo succede perché i programmi sono organizzati con strutture contigue, ad esempio gli array.

- Localita' temporale: quando il processore accede ad una locazione di memoria i è probabile che nel breve periodo debba accedere nuovamente alla locazione i

5.1 Funzionamento e tipi di cache

Affinche' la Cache possa funzionare, la RAM viene divisa in blocchi di dimensione fissa detti *cache lines*, o *linee*. Ogni linea e' formata da un numero prefissato di byte consecutivi della RAM. Ogni linea e' identificata da un indirizzo in RAM corrispondente al primo byte della linea. Supponendo quindi di avere una RAM suddivisa in 2^m blocchi ognuno formato da 2^n byte, allora sono sufficienti $m+n$ bit per identificare univocamente un dato presente in memoria. Inoltre, gli m bit piu' significativi identificheranno il blocco mentre gli n bit meno significativi identificheranno la locazione all'interno del blocco.
In ogni istante alcune linee di RAM sono contenute nella cache. Quando una word viene cercata dal processore, comincia dalla cache. Se il dato e' contenuto nella cache si verifica un *cache hit*, se non c'e' si verifica un *cache miss* e in quel caso il dato deve essere recuperato dalla RAM.

Vediamo adesso i principali tipi di cache. Il primo tipo e' la cache *direct-mapped*. Ogni entry memorizza 2^n byte consecutivi della RAM. Ad ogni entry sono associate due informazioni: il *bit di validita'* e il *Tag*. La loro utilita' verra' mostrata a breve.
La posizione della linea di RAM all'interno della cache e' univoca. Questa proprietà viene garantita grazie ad operazioni in modulo e a meccanismi simili secondo la formula

$$pos_{cache} = pos_{RAM} \ \% \ dim_{cache}$$

Chiaramente essendo che $dim_{RAM} > dim_{cache}$, diverse linee della RAM saranno mappate in una stessa linea di cache. Capiamolo meglio tramite un esempio, spiegando anche il ruolo del Tag in questa situazione.

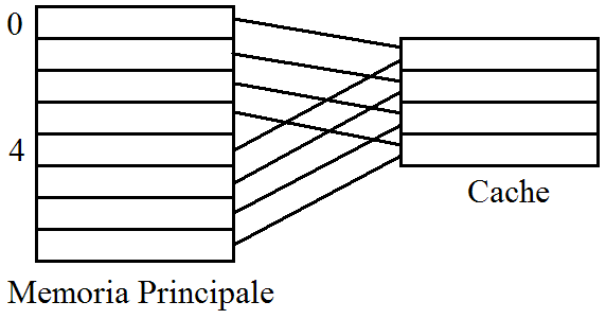


Figura 13: Corrispondenza memoria-cache dell'esempio 1

Esempio 1 : supponiamo di avere una memoria principale a 8 indirizzi e una cache da 4 linee.

Notiamo in Figura 13 che la linea 0 e la linea 4 della memoria principale quando vengono accedute vengono entrambe salvate, di volta in volta, nella prima linea della cache. Questo perché chiaramente

$$0 \equiv_4 4 \equiv_4 0$$

Poiché più di un elemento della memoria principale può essere associato ad una linea della cache, usiamo il Tag. Il Tag è un valore che viene associato ad ogni elemento della memoria principale e serve per risolvere i casi di ambiguità dovuti alle collisioni tra elementi che si possono sovrapporre sulla cache, proprio come quelli contenuti nella linea 0 e nella linea 4. In questo caso infatti la cache e' indirizzabile con 2 bit, mentre la RAM e' indirizzabile con 3 bit. Il numero di bit del Tag e' dato da $bit_{RAM} - bit_{cache} = 3 - 2 = 1$
Questo vuol dire che in questo caso ci e' sufficiente un solo bit per distinguere i casi di sovrapposizione. Infatti solo la posizione 0 e la posizione 4 possono collidere sulla posizione 0 della cache, quindi ci e' sufficiente un solo bit per capire quale delle due entry stia occupando la linea di cache.

Notiamo che all'aumentare della differenza di dimensione tra RAM e cache, aumentano il numero di sovrapposizioni e quindi aumenta anche il numero di bit del Tag. Ad ogni modo quale valore numerico dobbiamo assegnare al Tag per discriminare tra i vari indirizzi sovrapposti? In questo senso ci aiutano gli indirizzi della RAM. Infatti se abbiamo utilizzato l'operazione di modulo per stabilire la posizione delle linee m RAM all'interno della cache a c linee, allora i bit $[m, m-c]$ rappresenteranno il Tag. Ovvero il Tag sara' costituito dagli $m-c$ bit piu' significativi. In generale potremmo dividere un indirizzo in RAM in quattro parti distinte per reperire in modo univoco i dati dalle memorie. Mostriamo tale schema in Figura 14:

Figura 14: Suddivisione in quattro parti di una linea di RAM

- Byte: sono i bit meno significativi e identificano i singoli byte all'interno di una stessa word
- Word: sono bit adiacenti ai Byte, e identificano una word all'interno della linea
- Cache entry: sono i bit adiacenti ai Word, e indicano in quale linea della cache troveremo la linea corrente
- Tag: sono i bit piu' significativi e come illustrato poco fa permettono di risolvere le collisioni tra le diverse linee di RAM all'interno della cache

Come funzionano le letture e le scritture con la struttura che abbiamo dato?

- Lettura: quando la CPU cerca un dato, estrae i bit cache entry dall'indirizzo da cui dovrebbe prelevare i dati. A quel punto controlla il bit di validita' per verificare se i dati nella cache siano consistenti. Se sono consistenti controlla anche i bit del campo Tag, per evitare le collisioni. Se anche i bit del Tag sono identici si e' verificato un cache hit. Altrimenti abbiamo un cache miss. In tal caso la linea deve essere prelevata dalla RAM e copiata nella entry corretta della cache sovrascrivendo la linea e il Tag precedenti.
- Scrittura: la gestione delle scritture e' piu' delicata perche' dobbiamo mantenere la cache e la RAM coerenti tra loro. Per questo sono state sviluppate due strategie.
  La prima e' detta *write-through*: ogni volta che un dato viene modificato nella cache tale modifica viene riportata anche nella RAM. Questa tecnica prevede pero' un accesso in RAM ogni volta che un dato viene modificato.
  La seconda politica e' detta *write-back*: questa politica modifica i dati in RAM solo quando il dato modificato all'interno della cache viene rimosso dalla cache. Permette di risparmiare sul numero di scritture effettuate in memoria.

Possiamo anche fare un piccolo appunto sulla dimensione delle linee di cache. Quale e' la dimensione ideale? È chiaro che linee piu' grandi permettono di ridurre il numero di miss perche' capaci di sfruttare al meglio la localita' spaziale.
D'altronde pero', oltre una certa dimensione non conviene andare perche' se le linee sono troppo grandi la cache conterra' poche linee dando luogo a molti cache miss che comporteranno molte sostituzioni all'interno della cache.

Prima di vedere l'altro tipo di indirizzamento della cache studiamo meglio cosa succede nel caso si verifichi un cache miss.
In caso di miss infatti dovremo: inviare l'indirizzo alla RAM, accedere alla RAM e portare in cache la linea recuperata. L'accesso alla RAM e' l'operazione a maggior costo. Possiamo migliorare la situazione in due modi: allargando il bus(riduce i tempi di comunicazione tra cache e RAM), oppure possiamo allargare la banda passante della RAM, ossia la quantita' di dati prelevabili dalla RAM.
Allargare il BUS e' un'operazione complessa e che non ci da grandissimi margini di guadagno. È piu' interessante invece andare ad aumentare la banda passante della RAM, operazione piu' semplice e da cui possiamo trarre maggiori benefici.
Una soluzione semplice al problema della banda passante della RAM e' rendere la memoria RAM *interlacciata*. È uno schema simile a quello usato nei sistemi RAID: suddividiamo la RAM in un'insieme di banchi che possono essere letti in parallelo estraendo in una sola lettura una maggiore quantita' di dati proporzionale al numero di banchi in cui abbiamo diviso la RAM.

Vediamo adesso un altro tipo di cache: la cache *set-associativa*. Nelle cache direct-mapped suddividiamo la cache in un'insieme di linee e ad ogni linea corrispondono piu' linee della RAM. Nel caso delle cache set-associative dividiamo la cache in un certo numero di insiemi, ognuno costituito da un certo numero di linee. A questo punto quando viene prelevato qualcosa dalla RAM, viene inserito nella cache secondo

$$pos_{cache} = pos_{RAM} \ \% \ \#insiemi_{cache}$$

In questo modo quando due dati della RAM finiscono nello stesso insieme(collisione), controlliamo se esiste una posizione libera in tale insieme. Se c'e' non abbiamo la collisione, se invece non c'e' dobbiamo rimuovere qualcuno dall'insieme(tendenzialmente usando LRU).
La lettura delle cache set-associative e' comunque efficiente se consideriamo di poter leggere tutti gli elementi di un'insieme in un colpo solo.
Estremizzando questo ragionamento potremmo pensare di avere una cache come se fosse un unico grande insieme. In tal modo minimizzeremmo il numero di miss. Questo tipo di cache che viene detta *completamente associativa* non e' in realta' utilizzabile perche' la lettura in un colpo solo di tutta la cache sarebbe molto difficile e molto costosa dal punto di vista hardware da implementare.
Tuttavia il concetto di per se' resta valido: tanto piu' e' ampio l'insieme quanti meno cache miss avremo.

Adesso che abbiamo visto i diversi tipi di cache cerchiamo di fare una valutazione sul guadagno derivante dall'uso della cache. Supponendo che il tempo di esecuzione di una istruzione sia di 1 ciclo di clock, che il numero di riferimenti in memoria medio per istruzione sia 1.5 e che il tempo di accesso alla RAM sia di 100 cicli di clock, allora:

- Cache perfetta(miss rate = 0%): numero di clock per ogni istruzione(CPI) pari a $CPI = 1$
- Senza cache(miss rate = 100%): $CPI = 1 + 100 * 1.5 = 151$
- Cache(miss rate 2%): $CPI = 1 * 0.98 + 0.02 * 100 * 1.5 = 4$

Ad ogni modo, come ci possiamo immaginare un modo per ridurre drasticamente il numero di accessi alla RAM e' quello di usare un sistema di cache a piu' livelli. Ad oggi tutti i moderni multicore hanno due livelli di cache privati e condividono un terzo livello di cache.
Un altro aspetto di cui tener conto e' che l'efficienza degli algoritmi dipende dalla presenza/assenza della cache. Un esempio e' il Radix Sort, un algoritmo di ordinamento che per array di grande dimensione dovrebbe fornire prestazioni migliori del Quick Sort ma che in presenza di cache puo' comportarsi peggio di quest'ultimo.
Concludiamo dando una serie di acronimi tipici delle memorie

- DIMM(Dual Inline Memory Modules): moduli da 8 bytes che costituiscono le DRAM
- SDRAM(Synchronous DRAM): DRAM dotate di un ciclo di clock. In tal modo possono sincronizzarsi col clock del controller della RAM
- DDR: DRAM (in realtà, SDRAM) in cui è possibile trasferire dati sia sul fronte discendente che su quello ascendente del clock, raddoppiando le prestazioni rispetto a normali DRAM
- DDR2, DDR3, DDR4...: insieme di standard in cui viene man mano aumentata la frequenza del clock e vengono anche diminuiti i consumi
- GDRAM(Graphic Synchronous DRAM): particolari tipi di DDR adatti a gestire le richieste dei processori grafici trasferendo piu' bit in parallelo delle normali DRAM e collegate alla GPU direttamente mediante saldature che permette di diminuire la dispersione del segnale e aumentare il clock
- Flash memory: particolari tipi di memorie usate al posto degli hard disk poiche' sono statiche e in lettura sono molto piu' veloci di un hard disk. Ad ogni modo hanno un numero di cicli di riscrittura limitato a circa 100.000 per blocco

6 Architetture Parallele: multithreading

Fino ad ora abbiamo visto vari modi per sfruttare il parallelismo implicito nelle istruzioni macchina di un programma. Vediamo adesso delle tecniche esplicite di parallelizzazione.
A grandi linee, distinguiamo tre tipi di architetture parallele esplicite:

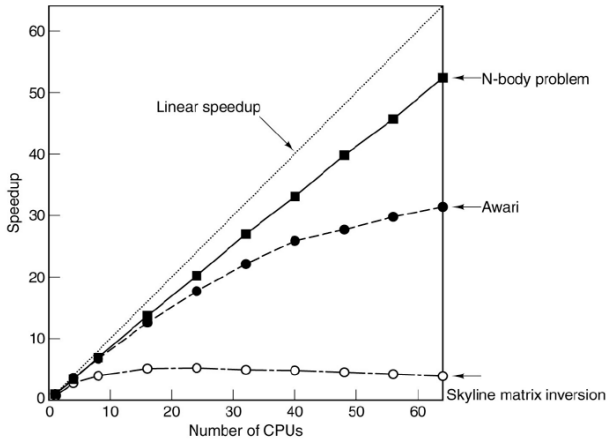


Figura 15: Speedup su problemi differenti

1. multi-threading
2. Sistemi a memoria condivisa
3. Sistemi a memoria distribuita

In questo capitolo studiamo il primo tipo di architettura esplicita.
Cominciamo dicendo che lo *speed-up*, ovvero l'incremento di prestazioni che possiamo ottenere usando piu' CPU, e' meno che lineare. Intuitivamente questo e' dovuto al fatto che non tutto il programma e' parallelizzabile(fase iniziale di inizializzazione, fase finale di sincronizzazione...). Consideriamo un esempio

Esempio 1 : consideriamo di eseguire

> gcc main.c function1.c function2.c –o output
>
> e che per una macchina monoprocessore servano 3 secondi per compilare main.c, 2 secondi per compilare function1.c, 1 secondo per compilare function2.c, 1 secondo per linkare gli oggetti (main.o, function1.o, function2.o). Il totale e' di 7 secondi.
> Avendo a disposizione 3 CPU impieghiamo comunque $3 + 1 = 4$ secondi. Dunque lo speedup e' pari a $7/4 = 1.75$ pur avendo triplicato il numero di processori.

Di fatto problemi diversi possono portare a uno speedup diverso che dipende da quanto sia parallelizzabile il problema. Questo fatto viene mostrato in Figura 15

Piu' formalmente quello che possiamo dire e' che:
dato un programma eseguibile in tempo t su un processore, detta f la parte di programma non parallelizzabile e $1 - f$ la parte parallelizzabile allora il tempo di esecuzione passa da $fT + (1 - f)T$ a $fT + (1 - f)T/n$ dove n e' il numero di processori disponibili. Nel complesso lo speedup e':

$$speedup = \frac{ft + (1-f)t}{ft + (1-f)t/n} = \frac{n(ft + (1-f)t)}{nft + (1-f)t} = \frac{nt}{t(1 + nf - f)} = \frac{n}{1 + (n-1)f}$$

Il risultato che abbiamo ottenuto viene detto *legge di Amdahl*.
Tramite questa legge risulta evidente che avremmo uno speedup perfetto solo nel caso in cui $f = 0$ ovvero solo nel momento in cui l'intero programma sia parallelizzabile. Infatti all'atto pratico i problemi delle computazioni esplicitamente parallele sono due

1. La quantita' limitata di parallelismo nei programmi
2. Gli elevati costi di comunicazione tra processi e memoria

Ad ogni modo la programmazione parallela e' una buona pratica per altri due motivi: la presenza di piu' processori aumenta l'affidabilita' del sistema(se si rompe un processore ne abbiamo altri che possono lavorare), inoltre poiche' i servizi sono spesso forniti su scala geografica, la programmazione parallela evita colli di bottiglia.

Vediamo adesso i vari metodi multithreading. Consideriamo inizialmente una CPU multihread che, in senso stretto, non e' un'architettura parallela in quanto il parallelismo viene realizzato con una singola CPU. Comunque questo tipo di architettura porta gia' il programmatore a concepire e sviluppare applicazioni parallelizzabili.
L'idea del multithreading nasce dal fatto che in presenza di un cache miss abbiamo una lunga attesa prima di poter proseguire. Per questo motivo possiamo nel frattempo eseguire un altro thread dello stesso processo. Perche' non possiamo eseguire thread di un processo differente? Semplicemente perche' il process switch puo' richiedere centinaia di cicli di clock per essere effettuato, dunque non avrebbe senso. Esistono tre tecniche di base per il multithreading:

- Fine-grained multithreading: lo switching tra i thread avviene ad ogni istruzione anche se il thread non ha generato un cache miss. La politica di scheduling a questo punto e' round robin. L'assunzione che viene fatta e' che il costo di thread switching sia minimo.
  Un aspetto particolarmente interessante e' che il fine-grained multithreading aiuta ad eliminare le dipendenze sui dati. Infatti anche se ci fosse una dipendenza, tra un'istruzione e la successiva trascorrono tanti cicli di clock quanti sono i thread, aumentando la probabilita' che tale dipendenza venga risolta. Il problema invece e' che con questa politica un thread viene fermato quando invece potrebbe proseguire nella sua esecuzione. Per questo motivo esiste l'approccio coarse-grained.

- Coarse-grained multithreading: lo switching avviene solo quando il thread genera uno stall. Questo genera un ciclo di clock sprecato ogni volta che abbiamo uno stall. Questo approccio spreca potenzialmente piu' cicli di

clock ma se ci sono pochi thread attivi possono gia' essere sufficienti a tenere la CPU sufficientemente impegnata. Inoltre con questa politica paghiamo molto piu' raramente il costo di thread-switching

- Medium-grained multithreading: via di mezzo tra le due politiche precedenti. Lo switching viene effettuato solo quando si sta per eseguire una istruzione che potrebbe generare uno stall di lunga durata(load o branch). L'istruzione viene comunque avviata, ma poi effettuiamo comunque lo switching. In tal modo non sprechiamo il ciclo di clock dell'eventuale cache miss.

Possiamo comunque chiederci come fare ad identificare a chi appartengono le istruzioni nella pipeline. Solitamente si utilizza un TID da associare ad ogni istruzione, oppure si svuota la pipeline ad ogni switch. Un'ultima assunzione comune a tutte le politiche viste finora e' che tutte le istruzioni dei vari thread siano disponibili in cache. Altrimenti ogni context switch genera un cache miss.

Le moderne architetture superscalari mettono insieme diversi principi. Infatti sfruttano il parallelismo insisto nelle istruzioni e a livello di thread. Tali architetture vengono dette architetture a *simultaneous multi threading*(SMT).
Nell'SMT ad ogni ciclo vengono avviate piu' istruzioni potenzialmente appartenente a thread diversi, aumentando l'utilizzo delle varie risorse della CPU. Infatti poiche' le istruzioni appartenenti a thread diversi sono quasi certamente indipendenti, e' possibile avviarle in parallelo.
Questo approccio e' potenzialmente migliore di quelli appena visti. Infatti le architetture senza multithreading possono subire lunghi stall. Le architetture coarse-grained e fine-grained mancano di parallelismo a livello di istruzione.
Ad ogni modo anche nell'SMT non si riesce sempre ad avviare il massimo numero possibile di informazioni per ciclo di clock a causa del numero limitato di unita' funzionali.

Per concludere vediamo due casi di multithreading reali

- Multithreading Intel: la Intel introduce l'hyperthreading nel 2002, ovvero la capacita' di eseguire due thread in modalita' SMT in contemporanea. La dimensione dell'area per supportare l'hyperthreading aumenta del 5% portando pero' un aumento delle prestazioni del 25%-30%. All'interno di questo modello dobbiamo stabilire dei metodi per gestire le risorse dei due thread. Intel individua 4 strategie
  - Duplicazione delle risorse: alcune risorse devono essere necessariamente duplicate, generando quell'aumento del 5% della superficie di cui abbiamo parlato poco fa
  - Partizionamento delle risorse: alcune risorse vengono rigidamente divise in due parti e ognuna di esse viene affidata ad uno dei due thread. Chiaramente questo modello puo' portare ad un sottoutilizzo delle risorse

  - Condivisione delle risorse: risorse completamente condivise. Il thread che ne prende prima il controllo le sfrutta. Questa strategia puo' portare ovviamente a situazioni di starvation e deadlock
  - Condivisione controllata: un thread puo' controllare una certa percentuale della risorsa(anche maggiore del 50%) ma non puo' mai controllarla tutta. In tal modo si evita la starvation. Questa soluzione e' piu' complessa perche' prevede un monitoraggio delle risorse, dunque altro hardware, un certo overhead...

  Ad ogni modo Intel abbandono' l'hyperthreading e lo reintrodusse nel 2008

- SUN UltraSPARC: processori della SUN implementano il fine-grained multi threading e sono maggiormente concetrati sul parallelismo a livello di thread. Lo switch avviene ad ogni ciclo di clock, saltando i thread in idle o in attesa di risolvere un cache miss. Il modello T1 in particolare sfruttava 8 cores e ogni core gestiva 4 threads. Le prestazioni ottenute dal T1 erano mediamente migliori delle altre architetture grazie all'utilizzo di 8 cores

7 Architetture parallele: multiprocessori a memoria condivisa

Una architettura con piu' CPU e che condivida la stessa memoria primaria viene detta multiprocessore. In un sistema di questo tipo tutte le CPU condividono uno spazio di indirizzamento logico mappato su una memoria fisica. La comunicazione tra le CPU avviene tramite letture/scritture in memoria.
Il problema di fondo dei sistemi multiprocessore a memoria condivisa e' la memoria stessa. Infatti diventa tanto piu' difficile da gestire quanti piu' sono i processori che vogliono usarla.
Tutti i moderni sistemi operativi prevedono la *multielaborazione simmetrica*. I processi possono essere inseriti in una coda *ready to run* per poi essere eseguiti. Tale coda puo' essere condivisa(ma dobbiamo fare attenzione ai problemi di concorrenza) o puo' essere specifica del processore.
Quando la coda e' propria del processore dobbiamo fare attenzione al *bilanciamento del carico*. Se le code ready to run sono fortemente sbilanciate e' opportuno ribilanciarne il carico. Tutti i sistemi operativi possiedono un meccanismo di questo tipo(Linux SMP bilancia il carico ogni 300ms o quando si svuota la coda di un processore). Alcuni sistemi operativi forniscono anche l'opportunita' di specificare che un processo non debba essere spostato.
Si distinguono di fatto tre diverse classi di multiprocessore:

1. Uniform memory access(UMA): la memoria e' condivisa fra tutti i processori, il tempo di accesso e' uguale per tutti i processori

2. Non Uniform memory access(NUMA): lo spazio logico e' ancora condiviso ma fisicamente la memoria e' distribuita tra le varie CPU e i tempi di accesso variano a seconda se i dati sono nella memoria locale o remota

3. Cache only memory access(COMA): i dati non hanno un loro luogo permanente in una qualche memoria. Bensi' possono migrare tra le memorie o essere replicati tra esse

Vediamo in maggior dettaglio queste classi di multiprocessore

7.1 Uniform memory access(UMA)

In questo modello abbiamo almeno due CPU che condividono una memoria. Quando una CPU vuole leggere /scrivere una locazione di memoria, verifica che il bus sia libero e invia la sua richiesta. Chiaramente la memoria puo' facilmente diventare un collo di bottiglia visto che tutti i processori devono sincronizzarsi sull'uso di essa. Nel seguito, per semplicita' supporremo che ogni nostra CPU sia single core.
Una soluzione che mitiga il collo di bottiglia della RAM consiste nel dare ad ogni CPU anche una cache privata, riducendo di molto gli accessi in memoria. Questa soluzione pone pero' un nuovo problema. Se un dato e' condiviso tra due processori, e uno dei due lo modifica nella sua cache privata, come fa l'altro ad accorgersene? Questo problema, che viene chiamato *problema di coerenza della cache*, ha diverse soluzioni tutte a livello hardware. La piu' comune e' basata su una tecnica di *snooping* del BUS, in cui il processore controlla le operazioni svolte dagli altri processori. Vediamo prima un protocollo write-through. Supponiamo di avere le seguenti quattro situazioni

- Read miss: il dato non e' in cache. La CPU preleva la linea mancante dalla memoria
- Read hit: invisibile agli altri processori
- Write miss: il dato non e' in cache. Prima scriviamo il dato in memoria poi lo carichiamo nella cache
- Write hit: la cache line viene aggiornata e l'aggiornamento viene anche propagato in RAM

Quando un processore vedra' un read miss di un altro processore non fara' nulla perche' questo non puo' invalidare i suoi dati nella cache. Quando invece abbiamo un operazione di write, se tale operazione viene fatta su un dato in comune, il processore che sta facendo snooping marca come invalida la propria linea di cache che contiene il dato comune. In questo modo nessuna cache puo' contenere dati inconsistenti. Il pregio fondamentale di questa tecnica e' la sua semplicita', che porta pero' ad una minora efficienza. Esistono infatti diversi protocolli write-back. Il piu' noto di questi e' il *protocollo MESI* in cui ogni entry della cache puo' essere in quattro possibili stati:

- Invalid: l'entry della cache non contiene dati validi
- Shared: piu' cache possono contenere la linea e la memoria RAM e' aggiornata

- Exclusive: nessun altra cache contiene la linea e la memoria RAM e' aggiornata
- Modified: la linea e' valida, la RAM contiene un valore vecchio

Purtroppo anche sfruttando il protocollo MESI l'uso di un BUS singolo crea un limite al numero di CPU che possiamo connettere. Per superare questo limite al numero di connessioni tra CPU e RAM introduciamo la *crossbar switch* un sistema a commutatori incrociati. Nella crossbar switch ogni CPU e' collegata ad ogni banco di memoria. Chiaramente questo comporta una crescita quadratica del numero di collegamenti, dunque anche questa soluzione non e' ottimale. Per connettere molte CPU si puo' usare un sistema basato su semplici switch in cui ogni ingresso puo' essere diretto verso ogni uscita. Per farlo devono essere specificate tramite un'opportuna stringa(divisa in quattro parti: modulo, indirizzo, codice dell'operazione e valore) quale sia la destinazione e quale sia l'operazione da fare.
Un'altra soluzione consiste nel collegare n CPU ad n memorie realizzando $log_2(n)$ stadi ognuno contenente $n/2$ switch. Nel complesso il numero di switch diventa $\frac{n}{2}log_2(n)$, che e' gia' molto meglio del valore n^2 che avevamo nelle crossbar switch. Notiamo pero' che questo approccio puo' portare a dei conflitti negli accessi. Per questo motivo le reti di questo ultimo tipo sono solitamente delle reti bloccanti.

7.2 Non uniform memory access(NUMA)

Il problema principale dei processori UMA e' che per realizzarli necessitiamo di un hardware complesso e costoso. Allo stato attuale non e' conveniente usare sistemi UMA con piu' di 256 processori. Per costruire sistemi piu' grandi dobbiamo accettare il compromesso che i tempi di accesso in memoria non siano uniformi.
Nei processori NUMA lo spazio di memoria e' uniforme dal punto di vista logico ma dal punto di vista fisico ogni processore e' dotato di un suo spazio di memoria locale. Questo comporta che programmi scritti per sistemi UMA possano funzionare anche su architetture NUMA.
Ad ogni modo e' sempre possibile, tramite load e store, accedere alla memoria di un altro processore(tale memoria viene detta *remota*). Possiamo comunque distinguere tra due tipi di architetture NUMA, *Non-Caching NUMA* e *Cache-Coherent NUMA*:

- Non-Caching NUMA(NC-NUMA): in questo sistema i processori non hanno una cache locale. In questo caso un MMU locale controlla se l'indirizzo riferito e' diretto alla memoria locale o a un modulo remoto. In questo caso il problema della coerenza della RAM rispetto alla cache e' risolto visto che la cache non c'e'. Quello che rimane pero' e' il problema dell'efficienza vista l'assenza di cache. In realta' anche tale problema e' in parte risolto introducendo una memoria locale privata che contenga solo dati privati del processore. Nonostante questo il tempo di accesso ai dati remoti resta molto alto

- Cache-Coherent NUMA(CC-NUMA): aggiungere una cache diminuisce i tempi di accesso ai dati remoti ma introduce il problema della coerenza della cache. Un modo per garantire la coerenza sarebbe ancora una volta lo snooping ma questa tecnica diventa troppo difficile oltre un certo numero di CPU. Il protocollo che invece utilizziamo e' a *directory based*.
  L'idea e' che ogni processore debba gestire una certa quantita' della RAM. Ad ogni processore allora associamo un database che contenga per ogni indirizzo di cui la CPU e' responsabile, le cache di quali altri processori contengano tale valore.

  Esempio 1 : supponiamo di avere una memoria RAM gestita da 256 processori e che ogni processore debba gestire 16MB di RAM. Lo spazio di indirizzamento e' unico. Consideriamo inoltre che una linea di RAM gestita da una certa CPU possa essere contenuta in una sola memoria cache. Abbiamo: 8 bit per individuare il processore responsabile di una certa linea, 18 bit per individuare la linea all'interno dei 16 MB di RAM di tale processore, e 6 bit di offset per indirizzare il byte all'interno della linea. Supponiamo che a questo punto la CPU 20 esegua una load sull'indirizzo 0x24000108. Traducendolo in binario otteniamo che tale indirizzo e' gestito dalla CPU 36, che la linea e' la 4 e che l'offset e' pari a 8. A questo punto la CPU 20 invia una richiesta alla CPU 36 per sapere in quale cache e' contenuto il valore dell'indirizzo 0x24000108. A questo punto la CPU 36 potrebbe verificare che tale valore non e' in nessuna cache. In tal caso il valore viene caricato direttamente dalla RAM e inviata alla CPU 20. A questo punto la CPU 36 inserisce nel suo database che il valore contenuto nell'indirizzo 0x24000108 e' contenuto nella cache della CPU 20. Se invece la CPU 36 avesse verificato che tale valore era gia' posseduto da qualcuno, ad esempio la CPU 82, allora avrebbe comunicato il numero 82 alla CPU 20 che a sua volta avrebbe prelevato il valroe dalla CPU 82. A questo punto il nodo 36 avrebbe dovuto cambiare il suo database dicendo che la versione piu' aggiornata dell'indirizzo 0x24000108 e' adesso posseduta dalla CPU 20.

  Abbiamo visto che nonostante un'architettura di questo tipo venga chiamata a memoria condivisa, opera molti scambi di messaggi.
  Ad ogni modo l'overhead e' abbastanza limitato vista la dimensione limitata dei messaggi.
  Nella realta' un' architettura di questo tipo sara' certamente piu' complicata visto che nell'esempio abbiamo considerato che una linea puo' essere contenuta in una sola cache.

In un sistema monoprocessore i vari processi si sincronizzano tra loro utilizzando opportune system call. Nei sistemi multiprocessore abbiamo bisogno di primitive di sincronizzazione simili. Ad esempio possiamo implementare un'operazione di exchange atomica costruendo un sistema di lock del tipo:

```
Shared var int lock = 0; // lock inizialmente accessibile
int v = 1;
repeat
   while ( v == 1) do exch (v, lock); //entry section
      critical section // qui dentro lock = 1
   lock = 0; // exit section
   altro codice non in mutua esclusione // il lock e' di nuovo libero
until false;
```

Tuttavia in un ambiente multiprocessore l'atomicita' sulle istruzioni non e' sufficiente. Infatti la mutua esclusione e' garantita sul singolo processore, ma succederebbe un disastro se nel frattempo un altro processore operasse sugli stessi dati. Potremmo disabilitare tutte le operazioni sulla memoria per garantire la mutua esclusione tra processori ma questo rallenterebbe troppo le operazioni delle CPU.
La soluzione adottata in molti processori sfrutta una coppia di istruzioni eseguite in sequenza: la prima cerca di portare in CPU il valore della variabile condivisa, la seconda cerca di modificarla e restituisce un valore che ci permette di capire se l'istruzione e' stata eseguita in modo atomico. Nel caso in cui l'esecuzione delle due istruzioni sia stata atomica non ci sono problemi. Se invece tra le due istruzioni c'e' stata un'interruzione la seconda viene di fatto annullata. Il meccanismo usato da questa coppia di istruzioni e' simile a quello di una *spin lock*. Se c'e' un fallimento le due istruzioni vengono eseguite di nuovo fino a quando l'esecuzione non va a buon fine. Questa soluzione e' in questo caso accettabile perche' le sezioni in spin lock sono molto corte e perche' essendo il sistema multi processore, la spin lock porta ad un deterioramento delle prestazioni minore rispetto a un sistema monoprocessore.
Chiaramente tutte queste operazioni sono piu' efficienti nel caso sia disponibile una cache. Infatti se la variabile di lock e' nella cache, puo' essere reperita in modo efficiente, altrimenti deve essere reperita in memoria.
Ad ogni modo queste tecniche vengono sfruttate soltanto quando il numero di CPU coinvolte non e' molto alto(10 o meno), altrimenti vengono usate delle tecniche piu' sofisticate.

A causa dei problemi di non determinismo sull'esecuzione delle istruzioni di un programma unite al fatto che all'interno di un calcolatore possono esserci diverse memorie inserite all'interno di una gerarchia, la coerenza della cache rispetto alla memoria puo' tradursi in un caos generale. Per questo motivo vengono implementate delle regole di gestione e comportamento della memoria che nel loro complesso costituiscono il *modello di consistenza della memoria*.
Questi modelli possono essere piu' o meno rigidi a seconda delle proprieta' che vogliano garantire. Abbiamo tre diversi tipi di modelli

- Consistenza stretta: la forma piu' ovvia di consistenza. Qualsiasi lettura ad una cella di memoria restituisce sempre il valore dovuto alla scrittura piu' recente. In realta' questa e' la forma di consistenza piu' difficile da

W100	W100	W200
W200	R3 = 100	R4 = 200
R3 = 200	W200	W100
R3 = 200	R4 = 200	R3 = 100
R4 = 200	R3 = 200	R4 = 100
R4 = 200	R4 = 200	R3 = 100

Figura 16: Possibili esecuzioni delle istruzioni nell'esempio 2

mantenere perche' causa un collo di bottiglia sulla memoria. Di fatto questo modello non viene implementato nella realta'

- Consistenza sequenziale: in questo meccanismo di consistenza, nel momento in cui abbiamo piu' letture e scritture su una stessa variabile, viene scelto un qualche ordine che magari non corrisponde a quello reale, ma tale ordinamento delle istruzioni viene percepito identicamente da tutti i processori

  Esempio 2 : supponiamo di avere quattro CPU. Supponiamo che CPU1 scriva 100 in una variabile x, dopo CPU2 scriva 200 in x, e che infine dopo tali scritture CPU3 e CPU4 leggano entrambe due volte x. Mostriamo in Figura 16 tre possibili esecuzioni delle istruzioni in oggetto.

  Tutte e tre le esecuzioni sono valide e possibili pero' soltanto la prima rispetta le proprieta' di consistenza sequenziale. Vediamo i vari casi:
  1)CPU 3 legge (200, 200) – CPU 4 legge (200, 200)
  2)CPU 3 legge (100, 200) – CPU 4 legge (200, 200)
  3) CPU 3 legge (100, 100) – CPU 4 legge (200, 100)
  È evidente che solo il caso 1) e' corretto perche' le due CPU non possono vedere ordini diversi in cui gli eventi si verificano

  Dunque, ancora una volta possiamo dire che la consistenza sequenziale permette di asserire che, nel caso di eventi multipli che avvengono in maniera concorrente, viene scelto un ordine preciso in cui questi eventi si verificano, e quest'ordine è osservato da tutti i processori del sistema. Sembra una proprieta' ovvia ma in realta' non e' cosi' facile da implementare visto che comunque imponiamo un ordinamento alle istruzioni. Per questo sono stati proposti altri modelli meno restrittivi

- Consistenza del processore: e' una consistenza meno vincolante che possiede due proprieta'

  - Le scritture di una qualsiasi CPU sono viste dalle altre CPU nell'ordine in cui sono state avviate

- Per ogni locazione di memoria qualsiasi CPU vede tutte le scritture effettuate da ogni singola CPU in quella locazione nello stesso ordine

Notiamo che la consistenza e' effettivamente piu' debole della consistenza sequenziale

Esempio 3 : supponiamo che una CPU1 scriva in x A, B e C. Concorrentemente una CPU2 scrive su x X, Y, Z. Nella consistenza sequenziale qualsiasi altra CPU che legga più volte x leggerà una qualche combinazione delle sei scritture, ad esempio X, A, B, Y, C, Z, e questa stessa sequenza verrà letta da ogni CPU nel sistema.
Secondo la consistenza del processore, diverse CPU che leggano più volte x potranno leggere sequenze diverse. Ciò che viene garantito è che nessuna CPU vedrà una sequenza in cui, ad esempio, B viene prima di A o Z viene prima di Y. L'ordine in cui ciascuna CPU esegue le sue scritture, viene visto allo stesso modo da tutte le altre.
Quindi, ad esempio, CPU3 potrà leggere: A, B, C, X, Y, Z. Mentre CPU4 potrà leggere X, Y, Z, A, B, C

Questo modello meno restrittivo e' adottato da molti sistemi multiprocessore.
Vediamo velocemente due architetture reali che implementano molto di quello che abbiamo appena detto

- Sun Fire E25K: formato da 72 CPU UltraSPARC IV(ogni UltraSparc IV e' formata da 2 UltraSparc III). È formato da 18 boardset ognuno dei quali contiene una scheda madre CPU-memoria(che contiene fino a 4 UltraSPARC IV), una scheda di input/output e una di espansione. Quindi e' un sistema che puo' contenere fino a 144 CPU. Ogni CPU ha una sua cache.
  Le comunicazioni all'interno di uno stesso boardset e' implementata tramite un BUS su cui e' implementata la logica di snooping, mentre la comunicazione tra i vari boardset e' implementata tramite 3 crossbar switch.
  Per la coerenza della cache viene usato un sistema combinato di snooping-directory based.
  Il Sun Fire raggiunge prestazioni di spostamento che toccano i 40 GB/sec
- : SGI ALTIX: usato in ambito militare/scientifico puo' essere costituito anche da 2048 processori Itanium 2 e 32 TB di RAM.
  Il sistema e' formato da un insieme di nodi detti *C-Brick* che possono contenere fino a 4 Itanium 2. Tali nodi possono pero' esser costituiti anche da sola memoria.
  In questo caso la coerenza della cache e' implementata sia tramite logica di snooping dei 4 Itanium all'interno di un C-Brick, sia tramite protocollo directory attraverso la rete di interconnessione

7.3 Multiprocessori COMA

Come abbiamo visto, nelle due architetture precedenti il problema e' che esiste un collo di bottiglia tra i processori e la memoria. Per aggirare questo problema nei sistemi COMA(Cache Only Memory Access) si usa un principio completamente diverso: l'intero spazio fisico viene visto come una grande cache. I dati possono migrare all'interno dell'intero sistema spostandosi da un banco all'altro di memoria. Ad ogni modo questo approccio e' ancora in fase di studio.

8 Architetture parallele: scambio di messaggi

Abbiamo visto come i sistemi ultiprocessore abbiano una scalabilita' limitata dovuta alle difficolta' di connettere tra loro molti core e al collo di bottiglia che viene a formarsi sulla memoria. Per questo motivo oltre un certo numero di processori dobbiamo rinunciare ad uno spazio di indirizzamento comune e accettare che la comunicazione tra le varie CPU avvenga tramite scambio di messaggi.
A causa di questa caratteristica queste architetture sono a volte chiamate *NO Remote Memory Access*(NORMA).
All'interno di queste architetture i programmi girano su CPU diverse scambiandosi informazioni tramite primitive del tipo send/receive. Ogni nodo e' composto da una o poche CPU che condividono della RAM.

8.1 Massively Parallel Processor

Un noto tipo di architettura NORMA sono le *Massively Parallel Processor*. Queste macchine sono spesso formate da centinaia o migliaia di CPU e un'alta capacita' di gestione dell'I/O. Inoltre le MPP devono avere un'elevata tolleranza ai guasti.
Vediamo un paio di architetture di questo tipo

- Blue Gene: nasce nel 1999 con l'obiettivo di essere il supercomputer con piu' potenza di calcolo ma allo stesso tempo il piu' efficiente energeticamente. Per questo usa solo processori a "bassa" frequenza di clock(700 MHz). I core sono aggregati in chip. Ogni chip contiene 2 core. Ogni chip possiede 512 MB di RAM. Due chip compongono una scheda custom e 16 schede custom compongono una motherboard. 32 motherboard compongono un cabinet. Il sistema completo e' formato da 64 cabinet. La rete di interconnessione tra i nodi e' un toroide a tre dimensioni. La comunicazione avviene tramite commutazione di pacchetto. Sono state realizzate piu' versioni successive di BlueGene

- Red Storm: l'unita' di base e' la scheda. Su ogni scheda giacciono quattro processori Opteron(single dual o quad core). Ogni core ha a disposizione 1 GB di RAM privata. Le schede sono impaccate a gruppi da 8 in una card cage. Tre card cage costituiscono un cabinet. L'intero sistema e'

formato da 108 cabinet. Anche in questo caso le CPU sono connesse tramite una rete toroidale, e la comunicazione viene gestita tramite dei processori Opteron dedicati

8.2 Cluster of Workstation

Un altro tipo di architettura a scambio di messaggi e' il *cluster of workstation*(COW). È formato da un'insieme di macchine connesse in rete. Quali sono allora le differenze con le architetture MPP? La distinzione non e' netta pero' possiamo dare qualche elemento:

- Nelle MPP le unita' computazionali sono tutte uguali dal punto di vista dell'architettura. Nelle COW le macchine possono essere molto diverse fra loro
- Nelle MPP le unita' comunicano mediante una rete ad alte prestazioni progettata ad hoc, nelle COW la rete e' solitamente Ethernet o Internet
- Nelle MPP la singola unita' computazionale non puo' funzionare senza le altre. In certi COW invece l'unita' computazionale puo' anche funzionare da sola

Un esempio di COW e' l'insieme delle macchine dei server di Google.
Google gestisce diversi centri di elaborazione sparsi nel mondo e quando un utente si collega tramite l'IP viene stabilito quale datacenter debba gestirne la richiesta. Ogni centro ha almeno una connessione a fibra ottica da 2488 Gbit che gestisce le query. È presente anche una connessione di backup.
La filosofia di Google e' di utilizzare processori a basso costo e aggiornarli solo quando il prezzo dei processori migliori cala. Un tipico PC Google possiede un processore di fascia bassa, qualche centinaio di Mega di RAM e qualche Giga di hard disk.
Ogni centro di elaborazione possiede 64 rack e ogni rack e' composto da 80 PC.
Partendo da questa architettura i principi fondamentali di Google sono tre:

- I componenti si guastano con frequenza proporzionale al loro numero. Il software deve essere tollerante ai guasti
- Le componenti hardware/software devono essere replicate
- Ottimizzare il rapporto costo/prestazioni

Come avvenga la replicazione hardware e' evidente. Per quanto riguarda il software Google esegue una scansione del Web e copia ogni pagina periodicamente. Le copie delle pagine web vengono divise in *shards* che vengono replicati tre volte su macchine non adiacenti.
Nel complesso, quando Google riceve una query vengono compiuti i seguenti passi: la query viene instradata ad un gestore di query, ad un analizzatore sintattico e a un server che gestisce la pubblicita' con cui Google si finanzia.

Viene consultato un server che contenga un indice per la query, e vengono reperiti i risultati piu' significativi. A questo punto il risultato viene mandato al client insieme alla pubblicita' e ad eventuali segnalazioni d'errore.

9 Processori vettoriali

Nella prima parte del corso abbiamo visto come migliorare le prestazioni di un processore single core utilizzando ILP, pipeline a molti stadi...
I processori vettoriali sono un'alternativa ai modelli visti precedentemente. Sono pensati per applicazioni che operino naturalmente su dati di carattere vettoriale come gli studi scientifici/meteorologici. Queste architetture mettono a disposizione istruzioni che operino direttamente su dati organizzati a vettori. Con questo approccio le istruzioni vettoriali hanno le seguenti caratteristiche:

- Una singola istruzione vettoriale specifica una gran quantita' di lavoro
- La computazione di ogni elemento del vettore e' indipendente dagli altri
- È necessario controllare le alee solo tra due interi vettori e non tra ogni coppia di elementi appartenenti ad essi
- Le istruzioni vettoriali accedono a locazioni adiacenti di memoria
- Poiche' una singola istruzione esegue un intero loop, scompaiono gli hazard sul controllo

L'architettura di un processore vettoriale viene mostrata in Figura 17

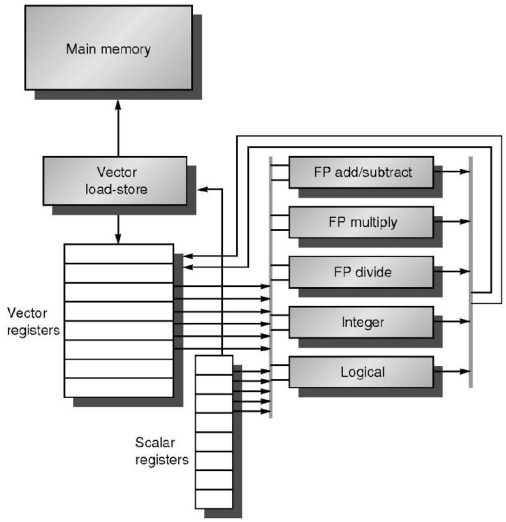


Figura 17: Architettura di un processore vettoriale

Ogni processore vettoriale e' formato da una parte scalare classica e da una parte vettoriale. Le unita' vettoriali sono capaci di operare sia su numeri interi che su numeri floating point e in binario. Le componenti principali sono: *registri vettoriali* che contengono ognuno 64 elementi, *unita' funzionali vettoriali* ovvero delle unita' pipelined che fanno partire una nuova operazione ad ogni ciclo di clock, *unita' vettoriali load/store* che spostano piu' celle adiacenti dalla RAM ai registri vettoriali e viceversa, infine dei normali registri scalari.
Le istruzioni piu' classiche su tali architetture sono:

- ADDV V1, V2, V3: somma gli elementi dei vettori V2 e V3, mette il risultato nel registro vettoriale V1 (ci saranno istruzioni analoghe per la moltiplicazione e la divisione)
- ADDVS V1, V2, F0: somma lo scalare F0 ad ogni elemento del registro vettoriale V2, metti il risultato in V1
- LV V1, R1: carica il contenuto della RAM a partire dall'indirizzo in R1 dentro V1
- EQ V1, V2: compara ogni elemento di V1 e V2 e metti il risultato in un vettore di bit.

A questo punto, vista la rimozione di tutti gli hazard di cui abbiamo parlato, potremmo chiederci come mai le architetture vettoriali non hanno preso il sopravvento negli anni. Vi sono vari motivi: prima di tutto l'architettura inerentemente piu' complessa limita la frequenza di clock di tali calcolatori. Infatti sono necessari registri piu' grandi, unita' funzionali piu' sofisticate, datapath piu' ampio e connessione con la memoria piu' sofisticata. Per queste ragion il costo dei processori vettoriali e' sensibilmente maggiore. Per questo motivo la domanda di tali processori e' sempre stata limitata e i costi di produzione non sono mai stati abbattuti. Inoltre l'aumento di prestazioni delle normali architetture ha permesso ai processori tradizionali di diminuire le differenze di prestazioni anche su problemi tipicamente vettoriali.
Ad ogni modo l'esperienza dei processori vettoriali e' stata in parte trasferita ai processori moderni che spesso hanno arricchito il loro set con istruzioni di tipo vettoriale per operare su dati naturalmente vettoriali.

10 Tassonomia delle architetture

Finora abbiamo visto diversi tipi di architetture. Possiamo classificarle in base a certe loro caratteristiche? Una classificazione dei diversi tipi di architettura un po' datata e' la *tassonomia di Flynn*. Considera ogni calcolatore come una macchina che deve operare su un flusso di istruzioni e su un flusso di dati. Suddivide le architetture considerando le combinazioni di questi due fattori:

- Single Instruction Single Data(SISD): classica architettura monoprocessore. Questa visione e' per lo meno superficiale e processori di questo tipo

ad oggi sono solo i processori piu' semplici come gli 8051, macchina a 8 bit, con una frequenza di qualche decina di MegaHertz e a basso costo. Considerare appartenente alla categoria SISD un processore superscalare multiple-issue sembra un po' una forzatura, che però di solito viene fatta: in fondo, i moderni processori sono comunque i naturali pronipoti della classica architettura Von Neumann

- Multiple Instruction Single Data(MISD): molti autori pensano che questa categoria non abbia neanche senso anche se alcuni osservano che i moderni processori pipelined sono vicini a tale categoria
- Single Instruction Multiple Data(SIMD): uno dei primi modelli di architettura proposti di cui l'Illiac IV fu capostipite. Anche i processori vettoriali ricadono in questa categoria
- Multiple Instruction Multiple Data(MIMD): tutte le architetture multicore multi processore e multi computer appartengono a questa categoria

Occorre sottolineare che comunque la tassonomia di Flynn sia abbastanza rozza e che diverse categorie non sono di facile classificazione. Ad esempio i processori multithread potrebbero essere considerati SISD o MIMD...

11 Programmare con l'assembler 8088

A livello piu' basso ogni programma e' una sequenza di istruzioni scritte in bit. Poiche' questo e' molto scomodo ogni processore permette di usare una rappresentazione simbolica di ogni istruzione. Possiamo anche usare nomi simbolici per le costanti e etichettare istruzioni per farvi riferimento in seguito. Possiamo inoltre usare delle pseudoistruzioni per guidare l'assemblatore o usare delle macro che servono per specificare sequenze di istruzioni piu' lunghe.
Ovviamente per poter programmare in assembler e' necessario avere una conoscenza approfondita della macchina, a differenza dei linguaggi ad alto livello in cui possiamo ignorare l'architettura del processore. Per questo motivo programmare in assembler su processori moderni e' di fatto molto complesso visto che tali processori sono assai sofisticati.
Visto che in caso di errore e' molto difficile capire dove stia il problema, spesso per iniziare si usa un simulatore che permetta di eseguire le istruzioni una alla volta, che permetta di fermare il programma...

Possiamo adesso parlare di un'architettura ben nota: l'8088. È uno degli antenati dei moderni processori Intel. Molte delle sue istruzioni possono essere ritrovate nei processori Intel piu' moderni. Il datapath dell'8008 e' molto semplice: preleva l'istruzione dalla RAM, incrementa il PC, decodifica l'istruzione, preleva gli operandi dalla memoria o dai registri, esegue l'istruzione e immagazzina il risultato in memoria o su un registro.
L'8008 possiede quattro registri generali a 16 bit

- AX:accumulator : usato per contenere di default il risultato di alcune operazioni come le MUL. Ad esempio:
  MUL (addr) //AX = AX · "dato il cui indirizzo è addr"

- BX: base register : se usato tra parentesi specifica la word in RAM il cui indirizzo e' contenuto in BX

- CX: counter register : contiene il contatore che gestisce i loop. Lo decrementa man mano.

- DX: data register : usato insieme ad AX nelle operazioni a 32 bit. Contiene i 16 bit piu' significativi

Lo stack dell'8008 e' una porzione di RAM da 64 KB e le chiamate di procedura vengono impilate dagli indirizzi a piu' alto valore 65536 fino all'indirizzo 0. In realta' anche i dati sono contenuti nello stack, a partire dall'indirizzo 0. Dunque i dati e i record di attivazione condividono lo stack.
Vengono usati anche quattro registri puntatori

- SP: stack pointer : punta la cima dello stack in RAM

- BP: base pointer : puo' contenere un indirizzo qualsiasi dello stack ma di solito viene usato per puntare all'inizio del record di attivazione sulla cima dello stack

- SI e DI: due registri che vengono usate per indirizzare i dati sullo stack, oppure usati in coppia con BX per indirizzare dati nel segmento dati del programma che e' in esecuzione

Esistono infine diversi registri speciali, ne elenchiamo giusto un paio

- IP: instruction pointer: di fatto e' il program counter

- Flag Register: e' costituito da un'insieme di flag che indicano se si sono verificate alcune condizioni particolari, ad esempio se il risultato dell'ultima operazione eseguita e' uno zero, se e' negativo, se ha generato un overflow...

Come viene gestita la memoria in assembler? Abbiamo registri da 16 bit che devono indirizzare 1 MB di RAM(2^{20} byte. L'indirizzo e' dunque da 20 bit ed e' formato dai 16 bit dei registri a cui concateniamo 4 bit a zero.
Ovviamente dobbiamo anche specificare quali siano i comandi che possono indirizzare la memoria. Nelle macchine RISC soltant le load e le store possono farlo mentre nelle macchine CISC anche le altre istruzioni.
Nel caso dello 8088 molte istruzioni indirizzano la RAM. Ad esempio $ADDCX(20)$ preleva i 2 byte contenuti in memoria alla locazione 20 e 21 e mette il risultato in CX.
Ad ogni modo le principali modalita' di indirizzamento della RAM sono:

- Diretto: l'istruzione contiene l'indirizzo dell'operando

- A registro indiretto: l'indirizzo dell'operando e' contenuto in un registro.
- Con spiazzamento: il valore contenuto nel registro viene sommato ad una costante numerica per formare l'indirizzo

Vediamo invece le principali classi di istruzioni per l'8088

- Trasferimento: la MOV e' di gran lunga la piu' usata e sposta i dati dalla memoria ad un registro o viceversa.
- Operazioni aritmetiche: questa categoria contiene le note operazioni ADD, SUB, MUL e DIV. Queste ultime due in particolare hanno un solo operando esplicito visto che sfruttano l'accumulatore
- Operazioni logiche e di shift
- Operazioni di test e manipolazione dei flag
- Cicli e operazioni su stringhe: i salti nei cicli devono essere a distanza 128 byte perche' il campo che specifica la lunghezza del salto e' ad 8 bit. Notiamo che 128 byte non implica quante istruzioni si possano saltare al massimo visto che la lunghezza delle istruzioni nell'x86 non e' fissa!
- Istruzioni di salto: effettuano i salti entro una distanza pari a 32768 byte. Se necessario e' possibile effettuare salti piu' lontani con la tecnica jump over jump in cui concateniamo piu' salti
- Chiamate di subroutine: tramite la CALL possiamo effettuare le chiamate di procedure. Quando la CALL viene eseguita deve essere salvato il PC per poter poi riprendere(return adress). A questo punto scriviamo nel PC il primo indirizzo della procedura. L'ultima istruzione della procedura invochera' RET che preleva l'indirizzo di ritorno posto sulla cima dello stack e lo scrive nel PC. Dovremo comunque fare attenzione al fatto che il codice chiamato puo' utilizzare i registri generali e se vi avevamo scritto qualcosa di importante e' necessario salvare il contenuto di tali registri prima che possano essere "sporcati"
- Subroutine di sistema: istruzioni per comunicare con l'esterno. Eventuali valori di ritorno di tali procedure vengono salvati in AX. Quando una subroutine di questo tipo termina lascia sullo stack le sue informazioni che possono essere utilizzate in seguito oppure cancellate cambiando il valore di SP

A questo punto possiamo vedere come viene davvero sviluppato un programma in assembler. Come gia' detto esistono dei simulatori che interpretano le istruzioni di un programma e ne tracciano l'esecuzione. Ovviamente questi simulatori svolgono spesso anche la funzione di debugger(permettendo di inserire breakpoint, visualizzare il contenuto in memoria...)
Il compilatore di un programma assembler viene spesso detto *assemblatore*, e il suo output viene detto *file oggetto*. L'assembler permette anche di specificare

delle etichette per indicare specifiche locazioni in memoria. Purtroppo usare le label introduce il problema che nel momento in cui una label fa riferimento ad una locazione avente un indirizzo maggiore, il compilatore non sa con cosa sostituire tale label. Per questo motivo la fase di compilazione si compone di due parti: nella prima ogni riga di codice viene analizzata e ad ogni label viene associato un valore numerico. Nella seconda passata puo' essere generato il codice binario.
Ad ogni modo come gia' detto il processo di compilazione dipende dalla macchina e anche dal compilatore stesso. Infatti ogni compilatore possiede un proprio insieme di pseudoistruzioni.
Ad esempio l'*Amsterdam Compiler Kit as 88* possiede un set di pseudoistruzioni i cui due tipi piu' importanti sono TEXT e DATA che servono a definire le sezioni di codice e di dati. Nell'ACK vengono usati due tipi di labels: *global* che devono essere uniche per tutto il programma, e *local* specifiche di una certa sezione di codice. Il tracer/debugger e' formato da sette finestre per visualizzare informazioni:

1. Processor with register: mostra i registri generali in decimale
2. Stack: visualizza il contenuto dello stack
3. Program text/source file: mostra la porzione di codice corrente in esecuzione
4. Subroutine call stack: le righe di codice relative alle chiamate piu' recenti di subroutine
5. Error output field: mostra eventuali errori del tracer e dati in input
6. Interpreter commands: gli ultimi comandi dell'interprete
7. Values of global variables/data segment: mostra una porzione del data segment

Un altro tipo di architettura e' la MIPS. In questo caso la memoria e' divisa in tre parti: codice, dati(statici e dinamici) e uno stack. Poiche' l'area dati inizia alla locazione 1000000_{hex} per poter indirizzare tale area si utilizza un global pointer che contiene l'indirizzo di partenza dell'area dati. Lo stack viene invece usato per il passaggio dei parametri, per salvare il contenuto dei registri che saranno sporcati dalla procedura chiamata e come spazio per le variabili locali. Esistono due puntatori detti *frame pointer* e *stack pointer* che delimitano lo stack.
Un'ultima cosa da dire e' che sebbe i registri di una macchina RISC siano generali, devono essere usati in modo specifico per rendere i programmi che li usano piu' efficienti.