PROCESSORI RISC

Dal fetch al completamento dell'istruzione in un processore richiede un certo numero di cicli di clock cadenzati ad una certa frequenza —> processori CISC.

Si cerca di ridurre il numero di cicli di clock. Allo stesso modo è aumentato il numero di istruzioni eseguibili dal processore.

Processori RISC, si modifica l'architettura e anche il set di istruzioni supportate, cercando di ridurle. Si evolve verso architettura a pipeline.
Nell'81 nasce il progetto MIPS, che punta a definire un'architettura RISC. Questo sarà la base dei processori ARM e RISC-5 (alternativa ARM).

I processori RISC sono la base dei SUPERSCALARI.

MIPS

L'instruction set è lo stesso di un processore RISC. **Le istruzioni sono tutte su 32 bit**.
Architettura load-store, la generica istruzione lavora su registri, per accedere in memoria si usa solo load-store, in questa maniera sfrutto le pipeline.

La versione MIPS64 utilizza registri a 64bit.

Il numero di registri è elevato, circa 32, perché architettura load-store.

R0-R31 registri per interi, R0 solo in lettura, contiene la costante 0.

L'indirizzamento è anche più semplice rispetto ai CISC (i quali lavorano anche su dati in memoria e non solo su registri). Esiste un unico modo di indirizzamento usato dalle load-store (perché sono le uniche che accedono in memoria).

LD R1,30(R2) prende la costante 30 la somma al valore di R2 e ottiene la cella di memoria a cui accedere.
R1 <— MEM[R2+30]

Da questo singolo modo di indirizzamento posso ricondurmi ad alcuni dei 7 modi di indirizzamento 8086:

Se costante 0 —>	LD R1,0(R2)	mi riconduco all'indirizzamento 8086
Se utilizzo R0 —>	LD R1,30(R0)	mi riconduco all'indirizzamento 8086

Tre tipi di istruzioni:

Immediate:	6 bit opcode 5 bit registro sorgente 5 bit registro destinazione 16 bit per operandi (immediate)	operazione tra registro e costante
Register	6 bit opcode 5 bit registro operando 5 bit registro operando 5 bit registro risultato 5 bit 6 bit funzione	operazione tra registro e registro
Jump	6 bit opcode	

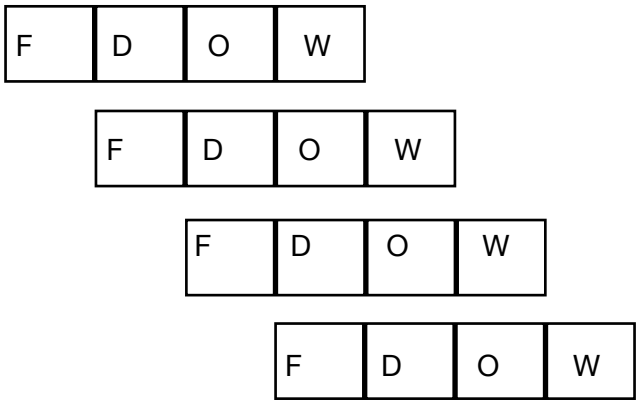
26 bit da sommare al program counter per fare il salto

Sonza-venerdì 12 ottobre 2018

PIPELINE

Senza pipeline caricamento, fetch, scrittura risultati hanno tutti moduli differenti. Quando un modulo lavora gli altri non fanno nulla —> perdita tempo.

Con pipeline ad ogni periodo di clock faccio partire il fetch di una nuova istruzione, ogni istruzione si troverà processata da un modulo diverso e tutti i moduli lavoreranno sempre.



Throughput
numero di istruzioni che completo nell'unità di tempo.

Unico segnale di clock che fa a tutti gli stadi —> sincronizzazione. Alla fine del periodo di clock tutti gli stadi devono avere completato il loro lavoro. *La frequenza del ciclo di clock è quindi determinata dallo stadio più lento nel caso peggiore.*

CPI
Clock cycle per instruction. Nel caso ottimale il CPI sarebbe 1, perché ad ogni colpo di clock esce un'istruzione. 1000 istruzioni, 1000 colpi di clock —> CP1=1. Nella realtà CP1>1.

Se non avessi pipeline CPI=4, cioè 4 colpi di clock per ogni istruzione.

Pipeline ideale
Throughput(pipelined)=Throughput(no pipelined)* n dove n numero di stadi.

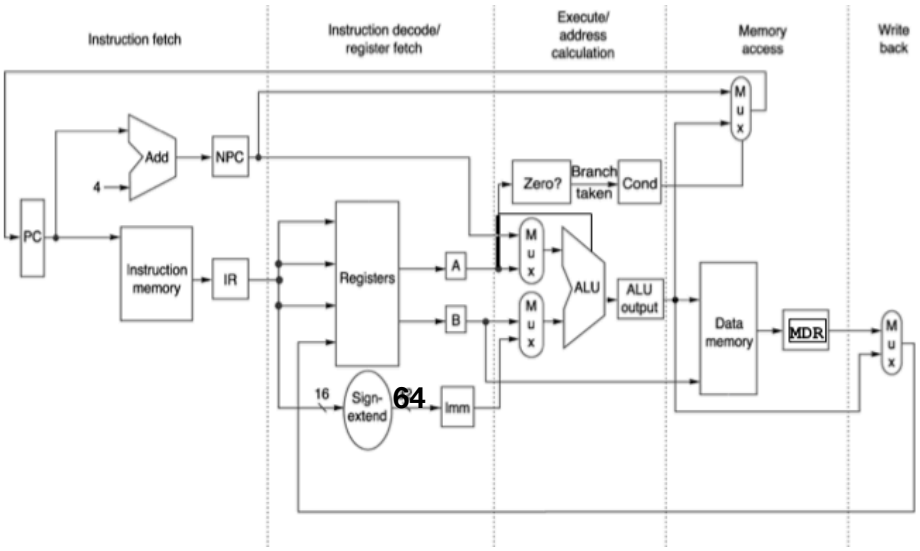
Aumentando il numero di stadi aumento il throughput ma diventa sempre più difficile far lavorare la pipeline come ideale.

Il MIPS ha 5 fasi, come il RISC-5:

FETCH	accede in memoria, carica in IR l'istruzione, aggiorna program counter.
INSTRUCTION DECODE	guarda codice macchina in IR e attiva segnali di controllo. In più il processore capisce che istruzione ha davanti, capisce gli operandi e li prepara. Gli operandi, ad accezione di load-store, sono registri e il processore sposta il loro valore nei registri temporanei per la fase successiva.
EX	esegue e salva risultato in registro temporaneo.
MEM	
WRITE BACK	prende il valore dal registro temporaneo e lo salva nel registro specificato.

Se senza pipeline CPI=5.

Architettura a ciclo singolo (no pipelined)



FETCH
IR <— MEM[PC]
NPC —> NPC+ 4 perché istruzione su 4 byte.

INSTRUCTION DECODE
Unità di controllo guarda primi 6 bit, capisce istruzione. Il suo compito è quello di prelevare i valori dai registri, se dovesse capitare un'istruzione salto non fa praticamente nulla. Si entra nel registerFile con l'indice del registro e come risultato ho il valore del registro.

EXECUTION
La ALU riceve dei segnali di controllo che sono stati generati al colpo di clock precedente dall'unità di controllo, guardando i primi 6 bit dell'istruzione.
L'unità di controllo poi gira i due MUX in modo opportuno, a seconda di avere register register o register immediate.
Il sommatore formato dai due MUX viene usato anche nelle istruzioni di salto, caricando in A o B il valore dell'offset e sommarlo al program counter (già aggiornato).

In caso di salto condizionato viene effettuata da ZERO? COND testando il registro A. COND è un FF che contiene 0-1 a seconda di condizione vera o falsa. Per semplicità l'indirizzo a cui saltare viene sempre calcolato, anche se condizione falsa. Se condizione vera il MUX aggiorna NPC e si salta alla nuova istruzione.

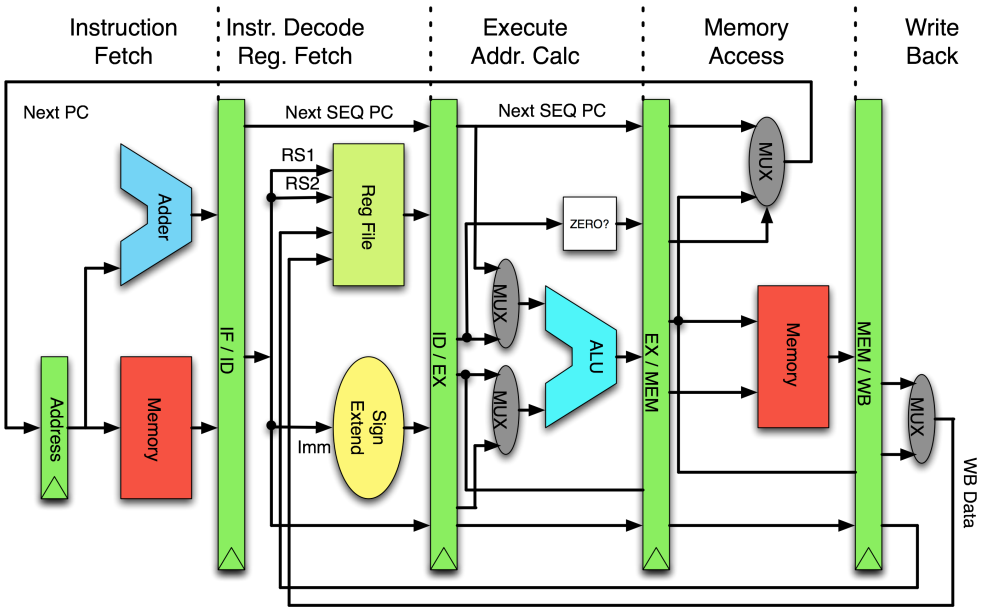
Se l'istruzione è di load-store calcola in A il registro da utilizzare, in B l'immediato da sommare e come output avrò l'indirizzo a cui accedere in memoria.

MEM
Nel caso di load-store prendo indirizzo calcolato in EXE, accedendo in memoria in lettura (load) o scrittura (store).
Il MUX poi salva il dato ottenuto in caso di (load) nel registro corretto (WRITE BACK).

In caso di salti condizionati in questa fase se la condizione è vera prendo output ALU e lo metto in PC altrimenti prendo NPC e lo metto in PC.

WRITE BACK
Prendo valore prelevato e lo metto nel registro corretto.

PIPELINING



I registri come NPC, LPD ecc.. sono andati a finire in appositi registri di pipeline. I registri pipeline ricevono tutti lo stesso segnale di Clock , caricando il valore che hanno sugli ingressi. Ad ogni colpo di Clock prendono il risultato della parte a monte e lo caricano nella parte a valle. Ciascuno stadio riceve input soltanto dallo stadio precedente, cioè dal registro di pipeline a monte.

Nella versione senza pipeline ho meno vincoli sul clock, perchè diversi registri possono avere Clock diversi.
Mentre nella pipeline tutti gli stadi devono avere lo stessa frequenza di Clock —> aggiungo vincoli, perdo da qualche altra parte. Solitamente clock pipelined leggermente più lento di quello senza pipelined.

Nel caso ideale ho 1 istruzione per ogni colpo di Clock, mentre prima avevo 5 colpi di Clock per ogni istruzione.
È difficile però raggiungere le condizioni ideali per diversi fattori, chiamati pipeline hazards. Ci sono tre tipi:

1) ***structure*** *legato a come è fatta la pipeline*, alcune memorie posso diventare collo di bottiglia perchè risorse condivise, cioè stadi diversi posso avere la necessità di fare accesso a quell'unica risorsa. La risorsa può non essere in grado di gestire più di una richiesta insieme. Esempio accesso al registerFile contemporaneamente in modalità scrittura e lettura. Uno stadio dovrà quindi fermarsi —> STALLO. Uno stadio quindi non completerà il suo lavoro in un periodo di clock. Analogo esempio è

l'accesso alle memorie. Esistono quindi memorie dual Port che permettono due accessi contemporanei.

Soluzione —> aumento il numero di periodi di Clock concessi allo stadio in stallo. Se dopo questo tempo non ha ancora svolto il suo compito, anche gli altri stadi andranno in stallo —> diminuisce throughput.

2) ***data*** Succede che ci siano istruzioni diverse che senza pipeline venivano eseguite correttamente e che ognuna richiede il risultato prodotta da un altra. Se eseguite in pipeline può capitare che i dati prodotti non vengano forniti in tempo.

Soluzione 1 *implemento una tecnica di forwarding o bypassing*, che fa si che quando necessario , l'input di un determinato stadio, non venga presa dal registerFile ma direttamente dal registro di pipeline che lo contiene. (implementato tramite multiplexer). Tutti i RISC adottano questa tecnica.

Soluzione 2 *mando in stallo le istruzioni che necessitano di un dato che deve essere ancora calcolato*, introducendo una bolla. —> perdo colpi di clock, scende troughput.

RAW, read after write, esempio di data hazard.

Lo stesso problema posso averlo anche in memoria, non solo con registri. Esempio Load e store di seguito (RAW in memoria).

3) ***control*** *Legati ai salti.* La pipeline ad ogni periodo di Clock carica una nuova istruzione, aumentando di 4 il Program Counter e al periodo di Clock successivo accede e carica l'istruzione successiva in memoria. Il salto modifica il PC 3 colpi di Clock dopo ma contemporaneamente la pipeline ha caricato le 3 istruzioni successive(ignorando il salto che doveva ancora aggiornare il PC).

Soluzione 1 —> fare in modo che il PC venga aggiornato subito e non 4 di colpi di Clock dopo. Sposto nel secondo stadio(D) la parte che calcola la condizione di JUMP. Non è una soluzione molto efficiente, ho solo ridotto di due stadi il problema, verrà caricata una sola istruzione prima dell'aggiornamento del PC.

Soluzione 2 —> quando carico istruzione di salto condizionato, presuppongo che la condizione non sia verificata e non modifico PC. Quando la condizione è verificata, devo aggiornare PC e sono nel problema precedente.

Soluzione 3 —> quando carico istruzione di salto condizionato presuppongo che la condizione sia verificata.

Soluzione 4 —> quando carico istruzione di salto condizionato mando in stallo la pipeline fino a che la condizione non è verificata.

Soluzione 5 —> *delayed branch*, non è la soluzione migliore perché con l'aumentare della profondità della pipeline aumenta il branch Delay slot.

Delayed Brach -Branch delay slot

In un processore RISC il compilatore può contribuire alle prestazioni. Quando si ha un salto, se non faccio niente, la pipeline carica istruzione sbagliata che sarà da buttare, perdendo un colpo di clock —> **branch Delay slot =1.**

Se il compilatore sa di questo rischio di caricare istruzione inutile, controlla se prima del salto ci siano istruzioni che posso spostare dopo il salto e che andrebbero comunque eseguite.

Queste istruzioni non devono essere ovviamente quelle che modificano la condizione.

```
LOOP    ld R4,values(R2)
        dadd R3,R3,R4
        daddi R2,R2,8        posso spostarla nel branch Delay slot, non ha
        daddi R1,R1,-1       dipendenze
        bnez R1,loop
        daddi R2,R2,8
```

BRANCH DELAY SLOT = 1, istruzione che carico dopo il salto, in questo modo invece di caricare un'istruzione sbagliata, caricherò sempre un'istruzione presa dal ciclo e non spreco quel cc.

Esempio dipendenze-hazard

Scandire vettore 10 interi e calcolare somma

```
MAIN    daddui  R1,R0,10    Somma R0 +10 e metti risultato in R1. R0 contiene
                             sempre 0. Ho ridotto numero istruzioni, non faccio una
                             MOV ma una somma con zero per implementare la
                             MOV. Daddui se uso costanti, dadd se uso registri.
        dadd   R2,R0,R0      dadd per registri
        dadd   R3.R0,R0
LOOP    ld R4,values(R2)     load, accede in memoria prende R2 che era inizializzato
                             a zero, lo somma a costante indirizzo values e carica
                             risultato in R4. È una mov da memoria a registro.
        dadd R3,R3,R4
        daddi R2,R2,8
        daddi R1,R1,-1
        bnez R1,loop         solitamente i RISC non usano flag ma verificano
                             condizione tramite registro e poi eventuale salto.
        sd R3,result(R0)     scrivo R3 in result all'indirizzo ottenuto sommando R0 a
                             result.
```

Analisi di esecuzione ESERCIZIO D'ESAME

```
MAIN    daddui  R1,R0,10    5
        dadd    R2,R0,R0    1
        dadd    R3.R0,R0    1

LOOP    ld R4,values(R2)
        dadd R3,R3,R4       —> aggiungo colpo di clock
        daddi R2,R2,8       —> stallo aggiunto da quello precedente
        daddi R1,R1,-1
        bnez R1,loop        —> aggiungo colpo di clock

        sd R3,result(R0)
```

ripetizione ciclo

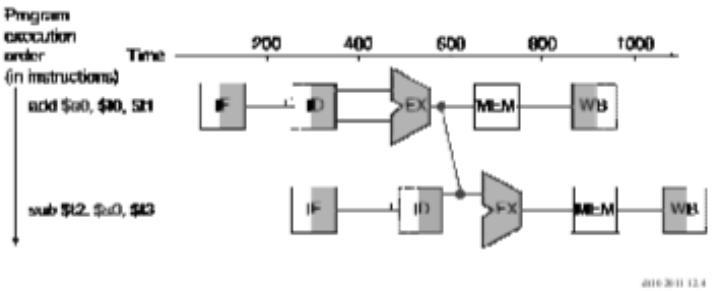
```
LOOP    ld R4,values(R2)
        dadd R3,R3,R4       —> aggiungo colpo di clock
```

Read after write Hazard perchè daddi modifica R1 e Benz la dovrebbe leggere subito. Daddi modifica nello stato di execute (perchè prodotto dalla ALU) ma la BNEZ legge R1 in fetch ma R1 non è ancora stato aggiornato. Bisogna inserire un nuovo stallo.

Bisogna anticipare il più possibile la verifica della condizione.
3 colpi di clock persi per ogni esecuzione del ciclo.
5 colpi per aspettare che la prima istruzione esca, 1 per ogni istruzione, più 3 persi, 8 totali per ogni esecuzione del ciclo.
Dipendenza di dato che crea problemi, load scrive in R4 in fase write la dadd legge R4 ma leggerebbe R4 nella fase di Decode, quindi sicuramente avremo un problema perchè la dadd legge il valore non aggiornato. —> si risolve con ***DATA FORWARDING***, processore attiva multiplexer che attivano direttamente l'unità di accesso in memoria LOAD, quindi già nella fase di MEM prendo i valori ricavati dal registro di pipeline senza aspettare che vengano scritto nel registro opportuno nella fase di writeback. Quindi nella fase di MEM della LD posso avere già valore aggiornato di R4 senza aspettare writeBack. ***Quindi utilizzo il valore quando è stato calcolato e non quando viene effettivamente salvato nel registro.***

Forwarding or bypassing

- use result when it is computed
  - do not wait for it to be stored in a register
  - requires extra connections in the datapath



ECCEZIONI

Somma di due tipi di eventi

- **interrupt esterni** periferiche
- **eventi interni** gestiti in maniera analoga agli interrupt esterni

C'è un pezzo di codice che gestisce le interruzioni del programma corrente, per poi tornare a lui una volta finita la gestione dell'eccezione.
Nei processori RISC questa gestione è molto complicata.
Mentre nei CISC quando arriva interrupt, salva il contento (PC, PSW) esegue routine e ritorna al programma originario ripristino il PC.
Nei RISC abbiamo più istruzioni parallelo che lavorano, devo quindi interromperle tutte e poi farle ripartire tutte dopo le gestione.

Eventi scatenanti:

- interrupt esterni — periferiche
- system call — INT genera interrupt via software
- debug — eccezione scatenato al termine dell'esecuzione di ogni istruzione
- Breakpoint — come debug
- Artimetica — verificarsi di overflow o underflow
- Artmetica floating
- Page fault — MMA o memoria virtuale, quando processore cerca di accedere a zona di memoria non in memoria principale , la MMA dice di fermarsi perchè la caricherà da disco e genera PAGE FAULT che è un'eccezione.
- Gestione indirizzi — esempio indirizzi non multipli di 4
- Memory protection — programma in esecuzione deve rispettare i vincoli sulle zone di memoria a cui può accedere.
- Undefined istruction — unità decodifica si prende istruzione caricata da fetch, poi prende valore e cerca di capire che istruzione ha davanti. Capità spesso che un valore non corrisponda a nessuna istruzione, genera eccezione.
- Hardware malfunction — le memorie hanno codice di protezione. Per ogni accesso ci dice se stiamo leggendo valore corretto. Quando si rileva valore sbagliato genera eccezione.
- Power failure

Caratteristiche eccezione

- *sincrone/asincrone* — Sincrone, arrivano alla fine di ciascun periodo di clock per esempio. Asincrone, solitamente esterne, arrivano in un momento qualunque.
- *user required/coerced* — richieste dall'utente/scatenate da qualche evento hardware, non dipendono dall'utente.
- intra/between instruction — se eccezione scatenata durante un'istruzione o tra due istruzioni.
- *maskable/unmaskable* — eccezione mascherate che voglio ignorare, forzo l'hardware a non gestire l'eccezione.
- *resume/terminate* — eccezioni che richiedono che programma interrotto venga ripreso o terminato.

Stopping the execution - GESTIONE PRECISA

Quando arriva eccezione nella pipeline dovrebbe identificare a quale istruzione è legata l'eccezione. Posso anche associare all'interrupt esterno un'istruzione (esempio quelle eseguite prima o dopo la gestione)
Tutte le istruzioni che vengono prima di quella associata all'eccezione dovrebbero essere completate. Tutte quelle dopo devono essere abortite, cancellando le scritture che hanno fatto, e poi eseguite quando e se si ritorna dalla gestione dell'eccezione.
Questo però non è cosi semplice da fare.

Esempio criticità eccezioni

Esempio1)

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

LD scatena eccezione quando accede in memoria (MEM). Con i dati su cui lavora la DADD questa scatena eccezione durante calcolo (EX). Nello stesso colpo di clock due istruzione generano eccezioni. Quale gestisco prima? Quella scatenata da LD.
SEGUE ORDINE CODICE

Esempio2)

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

LD scatena eccezione quando accede in memoria (MEM). Con i dati su cui lavora la DADD questa scatena eccezione durante fetch ID. Nello stesso colpo di clock due istruzione generano eccezioni. Quale gestisco prima? Quella scatenata da LD. Se non faccio nulla invece il processore gestirebbe prima l'eccezione scatenata dalla DADD.
* **NON SEGUE ORDINE CODICE MA ORDINE DI GENERAZIONE**

Exception order
Si evita di scatenare subito la gestione delle eccezioni. Quando eccezione viene scatenata non viene gestita subito, si setta STATUS FLAG. Se lo STATUS FLAG è dettato l'eccezione non potrà eseguire nessuna operazione di scrittura. Quando l'istruzione arriva al suo ultimo stadio viene gestita l'eccezione. SI VEDRÀ PIU AVANTI
* **Si torna a gestire le eccezioni nell'ordine che compaiono nel codice e non nell'ordine in cui vengono generate.**

Questo è tanto più facile quanto più l’instruction set permette di ritardare la scrittura del risultati di ogni istruzione.

MULTICYLES OPERATIONS

Le unità floating Point riguardano la fase di execute. L'unità intera posso renderla capace di eseguire tutte le operazioni in un solo colpo di clock ma non posso fare la stessa cosa con unità floating-point.

L'unità floating Point all'interno dello stato di execute introduce la necessità di avere più di un colpo di clock.
Si passa ad un architettura diversa

F	ID	EX somma int	MEM	WB
		EX somma float		
		EX moltiplicazione		
		EX divisione		

Perl l'unità floating Point ad ogni colpo di clock può far partire nuova somma. Posso avere architettura con o senza pipeline.

Latenza Numero di colpi di clock presenti tra un'istruzione che produce un risultato e un'altra istruzione che lo legge.

Initiation interval Numero di colpi di clock che devo avere tra il fetch di due istruzioni dello stesso tipo sulla stessa unità.

A seconda della latenza , le istruzioni di MUL e DIV richiedono più colpi di clock

F	ID	EX somma int	MEM	WB
		EX somma float		
	Latenza	EX moltiplicazione	Latenza	
Latenza	Latenza	EX divisione	Latenza	Latenza

A causa della nuova struttura della pipeline l'impatto delle dipendenze di dato è maggiore.

Ci sono nuove dipendenze di dato perchè le istruzioni raggiungono stato WRITEBACK in un ordine non facilmente prevedibile.

A causa di questa struttura posso avere diversi Hazard dovuti a:

- **Contemporary WB**
  scrittura contemporanea in un registro (WB nello stesso istante).
  Si risolvono andando in stallo nella fase di DECODE oppure prima di entrare nella fase di MEM o WB.
- operazioni con lunghe latenze possono introdurre stalli che durano per più tempo (si ripercuotono) nella pipeline.
- **WAW - ordine delle scritture**
  Istruzioni non raggiungono più il WB in ordine. Quindi prima di far passare un'istruzione allo stato di EX guardo se sta scrivendo in un registro che sta già utilizzando un'altra istruzione già nella fase di EX. Se così inserisco stallo nella nuova istruzione.

WAW HAZARD nuova dipende di dato, *Write after write hazard, due istruzioni che vogliono scrivere nella stessa destinazione*. Seguendo una logica come risultato dovrei aver avuto il risultato dell'ultima istruzione ma può succedere che il valore sia quello scritto dalla prima.
Questi hazard non esistono nell'architettura a pipeline che abbiamo visto prima del processo floating-point perchè tutte le istruzioni avevano EXE lunghe uguali.

Quindi devo aggiungere una logica che si accorga di questa situazione e aggiungere degli stalli.
Quindi l'utilizzo di un processore floating si aumenta le funzionalità ma introduce nuovi stalli.

Stopping the execution - GESTIONE IMPRECISA
Questo ha un impatto anche sulle eccezioni perchè diventa più difficile garantire le eccezioni precise. ***ECCEZIONI IMPRECISE***

```
DIV F0,F2,F4        —> DIV deve ancora fine la sua esecuzione
ADD F10,F10,F8
SUB F12,F12,F14     —> SUB genera eccezione
```

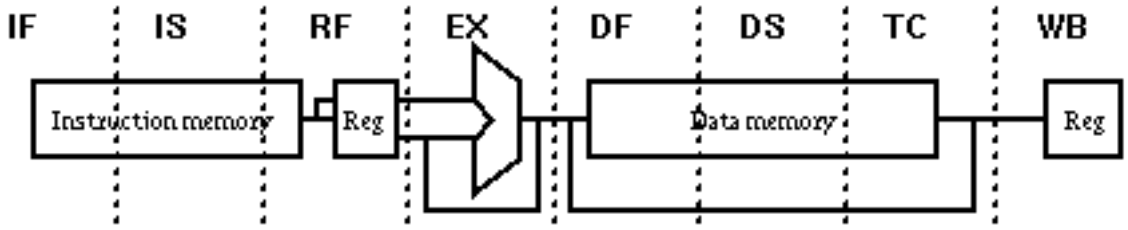
La Add e la Sub potrebbero finire prima della DIV perchè la div richiede molti CC. Se la sub generasse un'eccezione si ha un'eccezione imprecisa perchè tutte le istruzioni prima non sono ancora terminate.

Soluzioni

- Accettare le eccezioni imprecise
- **Bufferizzare i risultati di ogni istruzione fino a che tutte quelle prima non sono state completate.**
- Forzare le FP units a determinare anticipatamente quando un'istruzione può causare un eccezione in modo tale da fare il fetch di quella istruzione solo dopo che quelle prima sono completate. Cerco quindi di ritardare il fetch delle istruzioni che potrebbero creare eccezioni.

MIPS R4000 PIPELINE

Processore con pipeline a 8 stadi.



REG controlla se istruzione arrivata ha delle dipendenze di dato che richiede di mandarlo in stallo. Se si viene inserito stato a valle. Questo avviene nello stadio RF.

FETCH e SCRITTURA MEMORIA sono entrambe gestire con pipeline, ad ogni colpo di clock posso fare nuovo fetch e scrittura in memoria —> initiation interval = 1.

Questa nuova architettura essendo più profonda richiede numero di istruzioni maggiore tra LOAD e ISTRUZIONE GENERICA CHE LEGGE registro caricato. Questo prende il nome di LOAD DELAY SLOT.

È importante perchè rappresenta un margine su cui può lavorare il compilatore per migliorare le prestazioni.

LD R1 carico

LOAD DELAY SLOT

ADD …,…,R1 leggo

Nella pipeline vecchia si andava in stallo inserendo un colpo di clock di stallo.

Nella nuova pipeline vengono inseriti invece due stalli, LOAD DELAY SLOT. PROBLEMA Inoltre il BRANCH DELAY SLOT DIVENTA DI 3 cicli.

SOLUZIONE LOAD DELAY SLOT
Se nella LD ci fossero però due istruzioni io non perdo nulla. Riprende il concetto di latenza, se LD fosse seguita da due istruzioni non perdo colpi di clock. Quindi cerco due istruzioni prima di LD da mettere dopo LD senza che mi modifichi il programma.
Il compito di un buon compilatore, a valle di ciascuna load, controlla se c'è istruzione che legge il registro caricato (che andrebbe in stallo) dalla load e riempire il LOAD DELAY SLOT.

Più allungo la profondità della pipeline più il LOAD DELAY SLOT diventa lungo, più aumenta la penalità del salto.

Nell'architettura vecchia il BRACNH DELAY SLOT era 1, perchè a valle di un salto c'è un colpo di clock in cui processore carica istruzione, ora invece diventa di 3.

Esempio
for(i=0,i<100,i++)
Y[i]=X[i]^2 +X[i]/Z[i]

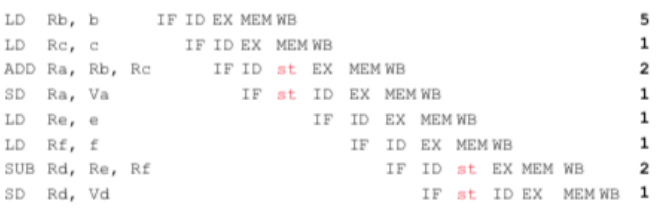
Z[i] sempre diverso da 0 —> non si generano eccezioni divisione per 0.

```
LOOP  Ld      F1,0(R2)
      Ld      F2,0(R3)
      mul.d   F3,F1,F1      —> qui non si creano problemi, F1 pronto quando serve a MUL
      div.d   F4,F1,F2      —> qui non si creano problemi
      add.d   F5,F3,F4      —> problema, risultati della mul e della div non disponibili subito
                               perchè mul e div ci mettono più colpi di clock. Stallo 1 colpo.
      s.d     F5,0(R4)      —> F5 crea problemi, non ancora disponibile, subisce
                               stalli precedenti più suo
      Daddi   R2,R2,8
      Daddi   R3,R3,8
      Daddi   R4,R4,8
      Daddi   R1,R1,8
      BNZ     R1,LOOP       —> crea problemi R1, stallo 1 colpo.
```

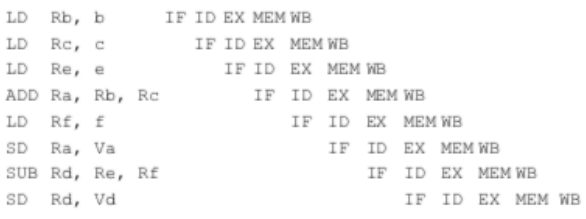
Quanti colpi di clock richiede esecuzione programma?
add.d richiede 1 colpo di clock extra, 1 suo + 1 stallo

venerdì 16 novembre 2018

```
LOOP  Ld      F1,0(R2)
      Ld      F2,0(R3)
      mul.d   F3,F1,F1
      div.d   F4,F1,F2
      Daddi   R2,R2,8       —> spostata da ???, guadagno un colpo di Clock
      add.d   F5,F3,F4
      s.d     F5,0(R4)
      ????????????
      Daddi   R3,R3,8
      Daddi   R4,R4,8
      Daddi   R1,R1,8
      BNZ     R1,LOOP       —> crea problemi R1, stallo 1 colpo.
```



14 clock cycles are required.



12 clock cycles are required.

INSTRUCTION LEVEL PARALLELISM-ILP

Quando passiamo da CISC a RISC con pipeline, che esegue istruzioni in parallelo, dobbiamo ipotizzare che le varie istruzioni possano essere eseguite in parallelo e cercare un ordine di esecuzione che massimizzi parallelismo.
Questo parallelismo deve esistere dentro un programma, parallelismo tra istruzioni.

Per sfruttare questo parallelismo ho due approcci:

- **Statico** il compilatore conosce la pipeline, analizza il codice e cerca l'ordine ottimale delle istruzioni che minimizza numero di stalli, cioè numero di colpi di clock. ***Statico perchè è il compilatore che esegue tutto.***
- **Dinamico** Il compilatore si limita a generare le istruzioni assembler, ***l'ordine viene deciso a run time dal processore*** (superscalare). Sposto l'intelligenza dal compilatore al processore, hardware quindi più complicato ma in questo modo massimizzo la portabilità del codice.

APPROCCIO STATICO
Il compilatore può prendere le istruzioni, le scrive in Assembler e le riordina nei BASIC BLOCKS, blocchi contenuti tra due salti. È complesso spostare istruzioni a cavallo di un salto. Le istruzioni all'interno del Basic Block sono sicuro che verranno eseguite in sequenza basta che rispettino le dipendenze di dato.

BasicBlock Parallelism

BRANCH-entry

B	**B**
A	**L**
S	**O**
I	**C**
C	**K**

BRANCH-exit

Il **BASIC BLOCK** è un sequenza di istruzioni con

- nessun salto in mezzo ad eccezzione della entry
- nessun salto fuori ad eccezione dell'out

All'interno di un basic Block il compilatore riordina le istruzioni, RESCHEDULING

Senza ordinare:
1. LD Rb,b
2. LD Rc,c
3. ADD **Ra**,Rb,Rc
4. SD **Ra**,Va
5. **LD Re,e**
6. **LD Rf,f**
7. SUB **Rd**,Re,Rf
8. SD **Rd**,Vd

Riordinando elimino gli stalli:
1. LD Rb,b
2. LD Rc,c
3. **LD Re,e**
4. ADD Ra,Rb,Rc
5. **LD Rf,f**
6. SD Ra,Va
7. SUB Rd,Re,Rf
8. SD Rd,Vd

Loop Parallelism

Più il basic block è grande più il compilatore ha possibilità di ottimizzazione. Solitamente hanno meno di dieci istruzioni quindi è difficile da sfruttare il meccanismo di ottimizzazione.

Devo cercare di aumentare la dimensione dei Basic block. Posso sfruttare i cicli facendo ***LOOP UNROLLING:***

```
for(i=0,i<N;i++)
        body
```

```
for(i=0,i<N/4;i++)
        body
        body
        body
        body
```

—> aumento numero di istruzioni, aumento di 4 volte il Basic block.

Vantaggi

- si riduce il numero di iterazioni e quindi il controllo
- aumenta la capacità del compilatore di poter ordinare le istruzione all'interno del Basic Block.

La dimensione del codice però aumenta. Importante nei sistemi embedded, costo memorie elevato.

Quando il compilatore riordina il Basic block deve tenere conto delle dipendenze.

Analisi delle dipendenze

L'analisi delle dipendenze è molto importante. Se due istruzioni sono indipendenti allora possono essere eseguite in parallelo senza nessuno stallo. Altrimenti dobbiamo identificare le dipendenze per capire come comportarci.

- **A** dipendenze di dato (registri, memoria)
- **B** dipendenze di controllo
- **C** dipendenze di nome

La dipendenza è una proprietà del programma. Le dipendenze sono il risultato di un'analisi del codice, non sempre creano problemi.
Le dipendenze sono diverse dagli hazards, che sono delle caratteristiche che derivano dall'architettura della pipeline.
Dietro ogni hazards c'è una dipendenza ma non sempre una dipendenza crea un Hazard.

A Dipendenza di dato - Registri

Si ha quando un'istruzione produce un risultato che è usato da un'altra istruzione.
Si ha quando un istruzione dipende da un'altra istruzione che a sua volta dipende da un'altra istruzione.

```
LD.D    F0,0(R1)
ADD.D F4,F0,F2
SD.D    F4,0(R1)
```

—> richiede uno stallo

Fino ad adesso abbiamo fatto attenzione alle dipendenze tra registri, ma ci sono anche **dipendenze di dato (memoria).**

A Dipendenze di dato - Memoria

Identificare le dipendenze al tempo di esecuzione è impossbile, utilizzo approccio conservativo, cioè assumo per esempio che se ci sono LOAD e STORE uno dietro l'altra, identifico sempre una dipendenza anche se in realtà poi magari non c'è (cioè operano su celle diverse).

IL COMPILATORE NON CONOSCE IL VALORE DEI REGISTRI, non può sapere se effettivamente gli indirizzi di memoria contenuti nei registri coincideranno o meno.

C Dipendeze di nome

Dipendenze associate a coppie di istruzioni tra cui non c'è flusso di dati ma entrambe le istruzioni fanno riferimento allo stesso registro o indirizzo in memoria (name).

Ne esistono di due tipi:

- **antidipendenze** quando un'istruzione scrive in un registro o locazione di memoria che un'altra istruzione, che verrà eseguita prima di quella che scrive, legge.
- **dipendenza di output** quando due istruzioni scrivono lo stesso registro o stessa locazione di memoria

LD	F0,0(R1)		DIPENDEZA DI OUTPUT
ADD	F4,**F0**,F2	add dovrebbe leggere R0	ANTIDIPENDEZA
SD	F4,0(R1)		
LD	**F0**,-8(R1)	R0 è scritto da LD	
ADD	F4,F0,F2		
SD	F4,-8(R1)		
LD	F0,-16(R1)		

ADD-LD *Si verifica **ANTIDIPENDENZA**, non ha un flusso di dati ma utilizzano entrambe la stessa risorsa. Non posso quindi sicuramente spostare la LD prima della ADD perchè sicuramente verrebbe eseguita prima.*

LD-LD *Si verifica dipendenza di **OUTPUT**, tutte e due scrivono nella stessa risorsa. Queste sono dipendenze di nome.*

*Questo tipo di dipendenze possono essere risolte tramite **REGISTER RENAMING.** Alcuni compilatori usano il register renaming per ridurre il numero di Hazards legati alla dipendenza e quindi di stalli.*

DIV	F0,F2,F4		DIV	F0,F2,F4	
ADD	**F6**,F0,**F8**		ADD	**S**,F0,**F8**	
SD	F6,0(R1)		S	**S**,0(R1)	
SUB	**F8**,F10,F14		SUB	**T**,F10,F14	uso registro temporaneo T
MUL	**F6**,F10,F8		MUL	F6,F10,**T**	

Tra SUB e ADD ho un ANTIDIPENDENZA perchè se la SUB fosse spostata prima della ADD dal compilatore starei violando la dipendenza e il risultato non sarebbe corretto. Altrimenti se SUB finisce prima di ADD stesso problema. Se uso un registro temporaneo anche se fosse spostata prima non creerebbe problemi perchè a questo punto lavorano su registri diversi.

Tra ADD e MUL ho un OUTPUT DEPENDENCE perché entrambe scrivono in F6. Se MUL fosse spostata prima della ADD dal compilatore avrei un errore del risultato perchè starei violando la dipendenza. Altrimenti se MUL finisse prima di ADD avrei lo steso problema. Rinomino come fatto prima il Registro F6.

Legame tra Hazard e dipendenze di dato
Ci sono Hazard diversi tipi:

- RAW Read after write
  DADD R1,R2,R3
  DSUB R4,R5,R1
- WAW write after write ***RISULTATO DIPENDENZE OUTPUT***
  LW R1,0(R2)
  DADD R1,R2,R3
- WAR write after read ***RISULTATO ANTIDIPENDENZE***

- RAR non corrisponde mai ad un Hazard.

B Dipendenze di controllo

Dipendenza di controllo introdotta dai salti.
Il basic block tendenzialmente a monte hanno un'istruzione di salto che potrebbe essere condizionato.

If p1 — *P1 determina se il basic block s1 verrà eseguito*. S1 è control-dependent da p1.

s1

If p2 — *P2 determina se il basic block s2 verrà eseguito.* S2 è control-dependent da p2.

s2

Dipendenza di controllo è legata ai salti condizionati nelle loro varie forme. Le dipendenze posso essere modellate attraverso dei grafi.

Se prendo un'istruzione di s1 e la porto fuori dal suo basic block perdo l'effetto della condizione, violo le dipendenze di controllo perchè portandola fuori dal Basic Block verrebbe eseguita indipendentemente dalla condizione. Le dipendenze di controllo hanno un significato e se voglio essere sicuro di non cambiare il risultato del programma non devo violarle.

Parte delle tecniche illustrare sono basate sulla violazione delle dipendenze di controllo. Se non violo dipendenza di controllo sono sicuro che il comportamento è quello previsto.
Ci sono però modifiche che nonostante violino le dipendenze di controllo non cambiano il risultato del programma.

Se cambio l'ordine di esecuzione delle istruzioni posso cambiare ordine in cui le eccezioni sono attivate all'interno del programma —> può creare problemi.

La correttezza di un programma si basa su:

- ***A Comportamento eccezioni***
  NON POSSO SPOSTARE ISTRUZIONI VIOLANDO UNA DIPENDENZA DI CONTROLLO PERCHÈ CREA PROBLEMI SULLE ECCEZIONI.
- **B *Data Flow***
  Inoltre il DATA FLOW è da considerare. La sequenza di istruzioni di salto condizionato determina la sequenza di istruzioni che vanno a scrivere i dati. Se modifico l'ordine i dati saranno diversi. Devo verificare che il flusso quindi sia corretto a valle della modifica.

A - PROBLEMA DEL COMPORTAMENTO ECCEZIONI

```
DADDU R2,R3,R4
BEQZ  R2,L1
LD    R1,0(R2)
```

La LD dipende dal risultato del salto condizionato. Può quindi scatenare eccezioni. Se sposto LD prima della branch creo delle eccezioni che non verrebbero scatenate se lasciata al suo posto —> non posso farlo.

Violo dipendenza di controllo e creo problema sulla generazione delle eccezioni.

B - PROBLEMA SUL FLUSSO DEI DATI.

```
        DADDU R1,R2,R3              DADDU R1,R2,R3
        BEQZ  R4,L                  DSUB R1,R5,R6
        DSUB  R1,R5,R6      —>      BEQZ R4,L
        DADDU                       DADDU R1,R2,R3
L       OR R7,R1,R8                 OR
```

Se sposto DSUB prima del salto condizonato verrà eseguita sempre. Generà pero eccezioni? No solitamente la sub non genera eccezioni. Se la sposto quindi prima non cambio il comportamento delle ECCCEZIONI ma cambio il comportamento del FLOW. A volte è accettabile se necessario. In questo caso posso spostare perchè il risultato dipende dalla DADDU e non dalla DSUB.

Violo dipendenza di controllo, non genero problema sulla generazione delle eccezioni ma genero problema sul data flow.

BRANCH PREDICTION TECNHIQUE

I salti creano problemi perchè il processore si accorge solo dopo la fase di fetch di aver decodificato un salto. Se salto condizionato ci sarà un momento successivo in cui valuterà condizione e a valle della valutazione verà modificato il PC. —> crea problemi alla pipeline perchè nel mentre processore ha caricato altre istruzioni nella pipeline che il salto abortirà. —> problemi di prestazione, riducono CPI.

La previsione che un salto condizionato sia preso o non preso è importante perchè se riesco a farlo il prima possibile posso influenzare il comportamento della pipelline sulle istruzioni successive. Non ho la certezza ma è una previsione.

Anche il compilatore e non solo il processore sfrutta le previsioni del salti.

Tecniche previsione:

- ***statiche*** la previsione viene fatta dal compilatore, guarda il codice e prova a fare delle previsioni sulla condizione del salto. Analisi statica fatta a monte dell'esecuzione.
- ***dinamiche*** fatta run-time dal processore nel momento in cui si trova davanti all'istruzione di salto codinzionato. La previsone può cambiare nel tempo, dinamica, perchè fatta a run time.

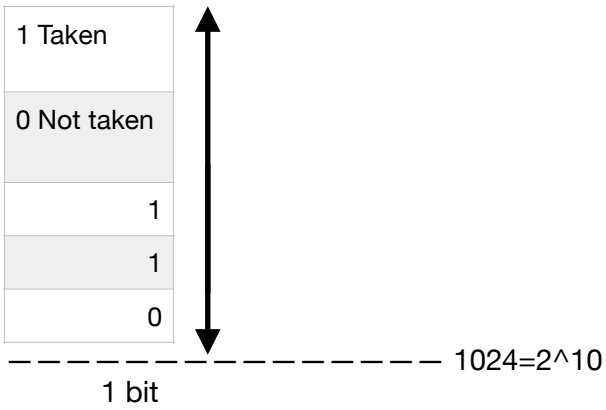
STATICHE

Si basano su tre approcci:

1) assumo salti sempre presi
2) compilatore può fare analisi e classificare un salto come preso o non preso sulla base del fatto che sia un salto in avanti o indietro. I salti in avanti solitamente sono non presi e quelli indietro solitamente presi (cicli). Alcuni processori permettono al compilatore di generare istruzioni diverse per salti presi o non presi (due diverse per esempio BEQZ una con preso e una con non preso). Serve per far capire al processore come il compilatore ha classificato il salto.
3) Viene eseguito più volte il codice con dati causali. Più costosa perchè ho analisi + simulazione. L'accuratezza dipende dal numero di simulazioni e quanto i dati in ingresso siano significativi.

DINAMICHE
Implementate in hardware, diversi approcci:

- **Branch history table**
  Per ogni salto condizionato , quando viene decodificato si accede alla BHT e si legge la predizione. Si aggiorna quindi il PC in relazione alla predizione che ho trovato. Nei processori MIPS la verifica della condizione di salto è effettuata nella fase di decodifica quindi la BHT alla fine non porta nessun vantaggio. Questo approccio richiederebbe una tabella (in hardware) tanto lunga quanti sono i salti. Per ovviare a questo problema si prendono i bit bassi dell'indirizzo dell'istruzione i salto. (dell'istruzione non di dove si andrà a saltare). Con quei 10 bit bassi accedo alla tabella e vedo se è taken o not taken.



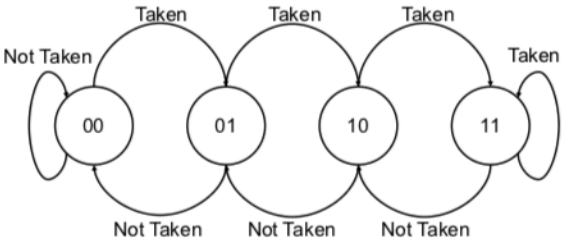
Indirizzo istruzione salto

	10 bit bassi Usati per accedere nella tabella

problemi:
- quando accedo tabella non è detto che prenda l'esito dello stesso salto alla volta precedente, potrei prendere quello di un altro perchè codifico il salto solo con i 10 bit bassi e due salti diversi possono averli uguali.
- Inoltre la predizione potrebbe essere sbagliata.

Quando processore parte cosa c'è in tabella?
- valori predefiniti, tutti 0 per esempio.
- uso predizioni statiche per inizializzare la tabella

Bimodal Branch prediction
Questa tecnica può essere migliorata se utilizzo due bit invece di uno. Per ogni salto classifico il salto come : frequentemente preso, frequentemente non preso o situazione intermedia.



Se contatore = 0 o 1 —> salto NT
Se contatore = 2 o 3 —> salto T

11 frequentlyTaken
00 FrequentlyNotTaken

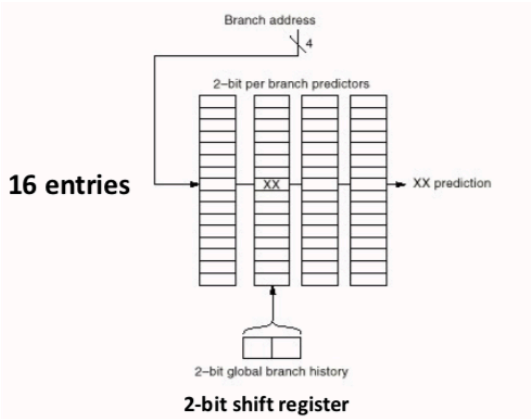
Ogni volta che eseguo il salto, sulla base del risultato, aggiorno i due bit.
I problemi restano gli stessi di quelli prima ma funziona meglio.

- **Predittori correlati.**
  Quando processore si trova davanti a salto condizionato, l'esito del salto dipende dal contesto nel quale il salto in una certa esecuzione viene eseguito. Si guardano i salti precedentemente eseguiti dal processore che esito hanno dato, due massimo tre. L'esito dei salti precedenti mi creano diversi scenari, per ognuna di queste condizioni vado a fare una predizione di come si era comportato il mio salto nello stesso scenario in passato.
  BRANCH HISTORY TABLE (matrice)
  (m,n)
  (2 ,2) —> ogni volta che eseguo salto per fare la predizione mi baso sui due salti precedenti (4 possibili scenari) , per ciascuno degli scenari ricordo cosa era successo nel passato nello stesso scenario. Si guarda l'esito degli ultimi m salti per scegliere tra 2^m predittori ognuno dei quali è un n-bit predictor.

00	01	10	11	*global branch history*
11 frequentlyTaken				
00 FrequentlyNotTaken				
11				
1				
0				

branch predictor

Per accedere alla tabella uso una concatenazione dei bit bassi dell'istruzione di salto più gli m bit della storia (m-bit dello shift register, global history)



Branch Target Buffer

Risolve il calcolo dell'indirizzo a cui saltare. Calcola perchè somma un offset al PC. Si cerca di predire qual'è l'indirizzo a cui saltare se la condizione di taken è soddisfatta.

Ora nella tabella non avremo taken o not taken ma l'indirizzo a cui saltare.
Questa operazione viene eseguita non dopo che il processore ha decodificato l'istruzione riconoscendola di salto condizionato, ma in parallelo con il fetch.
Quindi calcola possibile indirizzo a cui saltare alla fine della fase di FETCH, prima di sapere che quello sia un salto o meno.

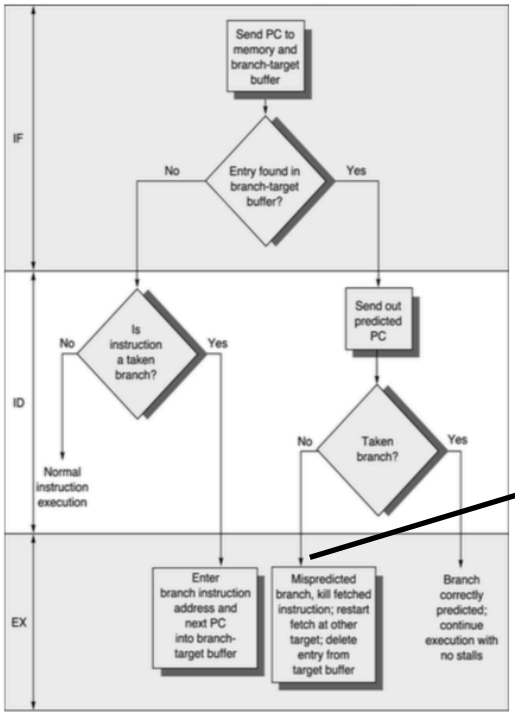
Riduciamo le penalità non solo legate a quelle di salto condizionato ma anche per quelle di salto incondizionato.

Nel branch target buffer non c'è solo l'indirizzo a cui saltare ma anche l'indirizzo dell'istruzione di salto.
In questo modo non ho più il problema che avevo nella BHT per il quale due salti con gli stessi bit bassi accedono alla stessa cella della BHT perchè ora posso fare il confronto anche sui restanti bit alti dell'istruzione di salto.

ADDRESS	TARGET	TAKEN/UNTAKEN
Indirizzo dell'istruzione di salto.	Indirizzo predetto del salto	*Questa colonna in realtà non serve perchè se il salto non viene preso non ci sarà proprio la riga con l'indirizzo a cui saltare perchè il branch viene scritto solo se il salto viene eseguito*

Per accedere al branch target buffer uso I 10 bit bassi dell'istruzione di salto, ma adesso una volta che ho indicizzato la tabella posso verificare se quello sia davvero il salto tramite il campo address.

Algoritmo

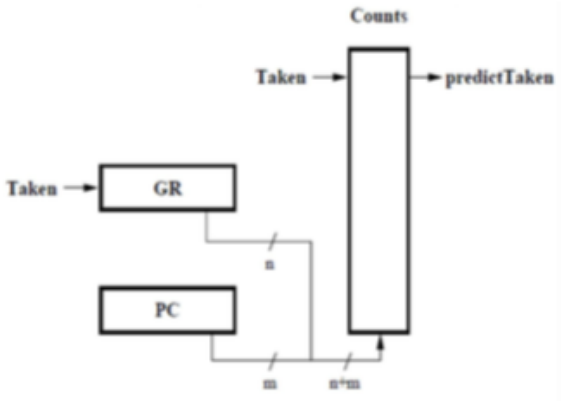


Se trovo entry ma salto non è taken allora è sbagliato perchè i salti not taken non ci dovrebbero essere nel Branch target buffer.

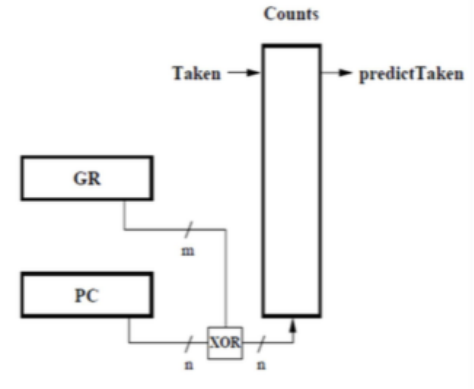
Se l'istruzione invece non è di salto non ci sarà la entry nella tabella ma comunque devo verificarlo.
Il branch target buffer se sbaglia la predizione deve essere aggiornato.

Normalmente le istruzioni di chiamata a procedura possono usare branch target buffer per eseguire salto.

- *Gloabal branch prediction* La BHT diventa uno shift register GR in cui salva l'esito degli n salti recenti. La difficoltà nell'uso di questo modello è che è più difficile identificare nel GR quale salto abbia preso perchè non vengono salvate tutte le informazioni del salto. Per questo motivo viene aggiunto l'indirizzo del salto PC.



index selection (g select)



index sharing (g sharing)

Branch Address	Global History	Gselect 4/4 (AND)
0000 0000	0000 0001	0000 0001
0000 0000	0000 0000	0000 0000
1111 1111	0000 0000	1111 0000
1111 1111	1000 0000	1111 0000

Branch Address	Global History	Gselect 4/4 (AND)	Gshare 8/8 (XOR)
0000 0000	0000 0001	0000 0001	0000 0001
0000 0000	0000 0000	0000 0000	0000 0000
1111 1111	0000 0000	1111 0000	1111 1111
1111 1111	1000 0000	1111 0000	0111 1111

DYNAMIC SCHEDULING TECHNIQUES

Passiamo da processore a pipeline ad uno dove l'ordine di esecuzione non è definito dal codice. Cerchiamo di sfruttare il parallelismo delle istruzioni riducendo il numero di stalli nella pipelne. Il processore durante l'esecuzione in hardware decide l'ordine di esecuzione delle istruzioni, decide dinamicamente le varie fasi di una generica istruzione se devono andare avanti o aspettare.

Vantaggi

- **In questo modo il processore riconosce per esempio le dipendenze di memoria, che il compilatore non poteva conoscere perchè non poteva vedere il valore degli indirizzi all'interno dei registri.**
- **Lavoro del compilatore semplificato**
- **Permette al processore di tollerare ritardi imprevedibili**
- **Permette il run dello stesso codice su diversi processori**

Out-of-order execution ordine diverso deciso dal processore e non dal codice. L'ordine dipende dalle dipendenze di dato ecc. Questo funzionamento potrebbe introdurre:

- **A.**WAR,
- **B.**WAW,
- **C.**eccezioni imprecise.

Esempio
DIV F0,F2,F4
ADD F10,F0,F8
SUB F12,F8,F14

La pipeline va in stallo perchè c'è una dipendenza tra la DIV e ADD (RAW). La Sub nonostante non sia coinvolta in nessuna dipendenza con le istruzioni precedenza anche lei va in stallo perchè gli stalli si ripercuotono.
Potremmo migliorare performance abilitando la out-of-order execution e mandando in esecuzione la SUB prima della DIV.

A WAR Hazard
DIV F0,F2,F4
ADD F6,F0,F8
SUB F8,F9,F14

Se la SUB fosse eseguita prima delle ADD , dato che out-of-order execution è abilitata, il risultato della ADD sarebbe sbagliato. Questo è un caso in cui non devo sfruttare la out-of-order execution.

B WAW Hazard
DIV F0,F2,F4
ADD F6,F0,F8
SUB F8,F9,F14
MUL F6,F10,F8

Se la MUL fosse eseguita prima della ADD, dato che out-of-order execution è abilitata, il risultato sarebbe sbagliato (perchè dipenderebbe dalla ADD invece che dalla MUL). Questo è un caso in cui non devo sfruttare la out-of-order execution.

C Eccezioni imprecise
Se l'out-of-order execution è abilitata è impossible gestire in modo preciso le eccezioni perchè quando un'eccezione è innescata è possibile che un'istruzione prima di quella che ha innescato

l'eccezione debba essere ancora completata o un'istruzione dopo quella che ha innescato l'eccezione sia già stata completata.

In entrambi questi casi ritornare al programma principale dopo la gestione dell'eccezione risulta difficile.
Verranno introdotte successivamente dei metodi per risolvere questo problema (esempio REODER BUFFER CON COMMIT IN ORDINE).

Splitting ID Stage

Per permettere out-of-order execution devo dividere in due parti lo stadio di decodifica.

Precedentemente MIPS decodificava istruzione con relativi segnali di controllo e caricava a monte operandi per l'execute.

Ora decodifica e accesso agli operandi vengono separati perchè devo fare decodifica, capire istruzione, capire a quale unità funzionale l'istruzione verrà caricata e poi accedere agli operandi.

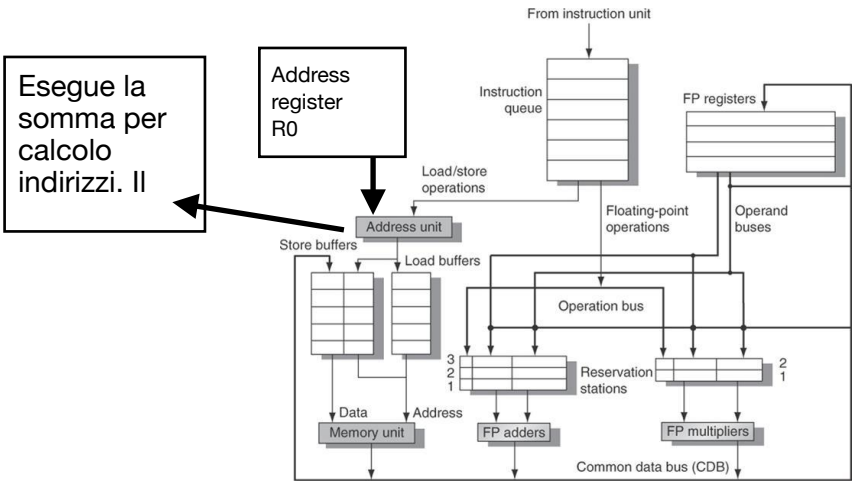
- **ISSUE** (structural Hazard) capisce quale istruzione ha davanti e quale unità funzionale avrà bisogno. C'è la logica di controllo per dipendenze strutturali, Se unità funzionale occupata e arriva istruzione che deve utilizzarla allora l'istruzione aspetta. L'ISSUE delle istruzioni avviene sempre in ordine.

- **READ OPERANDS** (Data Hazard) accesso ai registri per prendere operandi per l'istruzione. Quando operandi disponibili l'istruzione va nella fase di EX. *In questo stadio le istruzioni possono fermarsi se operandi non disponibile o passare davanti ad altre. Quindi ci può essere out-of-order execution.* Nella read operands va a finire la logica per la gestione dei data hazards.

- **EXECUTE** Posso avere anche due unità diverse che fanno la stessa cosa, quindi due istruzioni che vengono eseguite nello stesso momento. Quando due istruzioni entrano in due unità funzionali diverse, l'ordine di ingresso non per forza è uguale a quello di uscita.

Hardware schemes for dynamic scheduling
- Scoreboading
- Tomasulo's algorithm verificare la disponibilità del operandi e introduce register renaming.

Tomasulo Architecture



Le idee principali introdotte da Tomasulo sono

- Verificare la disponibilità degli operandi
- Register Renaming

Reservation station buffer in cui si accumulano gli operandi delle istruzioni che dovranno essere eseguite da quella unità funzionale. Dentro ciascun elemento della reservation station se operandi non ancora disponibili perchè istruzioni in corso stanno producendo quei valori, ci sarà scritto identificatore dell'istruzione (cioè il codice della reservation station che la sta memorizzando).

Register Renaming in Tomasulo

Quando l'operando che deve usare un'istruzione non è ancora disponibile , al suo posto viene scritto il nome della reservation station in cui è contenuta l'istruzione che sta aggiornando il valore di quell'operando. L'operando non è più un valore in un registro ma è il valore risultato di un'istruzione. —> ***alternativa al register renaming In questo modo si eliminano gli Hazard WAW and WAR.***

Quando istruzione assegnata ad un unità funzionale si assegna all'istruzione uno slot della reservation station dove vanno scritti i valori degli operandi. Per fare questo si accede a registerFile in modo tale da saper dire se c'è un'istruzione che sta calcolando un nuovo valore di quell'operando che sto guardando.

Common Data Bus - CDB

Quando i risultati sono pronti vengono passati direttamente alle unità funzionali tramite il common data bus senza invece passare attraverso i registri.
Per esempio le LOAD posso passare direttamente il valore dell'indirizzo di memoria a cui si trova il dato tramite il common data bus alla reservation station invece di salvarlo prima in un registro.

Unità load/store *(memoria)* Calcola indirizzo con cui accedere in memoria e poi accede.
Se uno store e poi una load è possibile che vada in esecuzione in ordine opposto —> può creare read after write hazard. (cosa che non succedeva prima senza scheduling dinamico).
Devo bloccare l'istruzione a seconda che ci sia o non ci sia la dipendenza di dato.
Se istruzione di load si va a vedere se c'è istruzione precedente di store che lavora sullo stesso indirizzo. Dentro unità di load e store c'è un buffer in cui sono memorizzati gli indirizzi su cui lavorano le varie istruzioni di load e store in attesa di essere completate.
Se load e storie lavorano sulla stesso indirizzo la load si ferma. —> nella reservation station devo avere anche l'indirizzo su cui le istruzioni lavorano.

Instruction Steps

Tutte le istruzioni seguono la fase di

- ISSUE
- EXECUTE
- WRITE RESULT

- ISSUE — l'istruzione viene prelevata dall'instruction queue. Se non c'è nessun reservation station libera per salvarla si va in stallo per Hazard Strutturale. Se reservation station libera si va a salvare l'istruzione e se anche gli operandi sono disponibili vengono salvati nella RS altrimenti si salvano le unità funzionali che stanno producendo gli operandi. (Register Renaming).

- EXECUTION

Normal instructions
Quando gli operandi sono pronti vengono scritti sul Common data bus e le reservation station possono prenderli direttamente da li invece che passare per i registri. *Solo quando tutti gli operandi sono disponibili l'istruzione può essere eseguita —> si evitano gli hazard RAW. Non vengono però eliminati perché perdo comunque CC perchè istruzione ferma.*

Load/Store
Per un'istruzione di Load/store per prima cosa si calcola l'indirizzo tramite la Address unit e lo si salva nel load/store buffer. La load viene eseguita non appena la memoria è disponibile. La store aspetta che gli operandi vengano scritti in memoria e la sua esecuzione avviene come per la load non appena la memoria è disponibile.

Branches
Per preservare il corretto comportamento delle eccezioni, nessuna istruzione può essere eseguita finché tutti i salti che la precedono sono completati. Quindi ogni volta che arriva all'unità di ISSUE un salto deve bloccare tutte le istruzioni successive. Come avevamo detto all'inizio! Se si usa dynamic branch prediction posso eseguirle solo dopo la predizione. (l'esecuzione speculativa porterà poi vantaggi)

- WRITE RESULT — Quando un'istruzione ha calcolato il risultato lo scrivo sul CDB e da qui passa alle unità funzionali che ne avevano bisogno e ai registri.

Store
Le store scrivono in memoria in questo stadio.

Reservation Station
Nelle reservation station ogni istruzione è associata ad un ID. Questo ID serve anche per identificare gli operandi che lei sta producendo —> register renaming.

Op Istruzione	**Vj** Valori operandi sorgente	**Vk** Valori operandi sorgente	**Qj** Id della reservation station che produrrà operando sorgente se non ancora disponibile	**Qk** Id della reservation station che produrrà operando sorgente se non ancora disponibile	**A** Usato solo nel load/ store buffer.	**Busy** Stato della reservation station e unità funzionale associata

giovedì 6 dicembre 2018

Register File

Ogni elemento nel register file contiene l'elemento Qj che contiene l'ID della reservation station che contiene l'istruzione che sta producendo il risultato che andrà in quel registro.
Se Qi è NULL nessuna istruzione in questo momento sta producendo un risultato in quel registro.

Tomasulo Loop

Con l'architettura di Tomasulo non è più necessario il loop Unrolling perchè le istruzioni di loop vengono automaticamente eseguite in parallelo.

VANTAGGI TOMASULO	SVANTAGGI
La logica per l'identificazione degli hazard è distribuita.	Alta complessità dell'hardware
WAW e WAR sono eliminati	CDB può essere collo di bottiglia.
	RAW evitati ma impossibile da eliminari, dovrò sempre aspettare gli operandi. Garantisco solo correttezza del risultato.

Load and Store order

Le istruzioni di load e store possono essere eseguite in qualsiasi ordine a patto che non facciano riferimento allo stesso indirizzo in memoria. In questo caso

- Se la load era prima della store può esserci WAR hazard in memoria.
  Perchè la STORE scrive in memoria dopo che la LOAD ha letto allo stesso indirizzo. Quindi se cambiassi l'ordine potrei avere un hazard.

- Se la store era prima della load può esserci un RAW hazard in memoria.
  Perchè la LOAD legge da un indirizzo di memoria che prima ha scritto la SD.

Per evitare hazard dovuti al riordino delle load e delle store il processore deve

- calcolare l'indirizzo di memoria in ordine
- effettuare dei controlli per ogni load e store

Per le load ogni volta che una load è pronta per l'issue

- si controlla lo store buffer per vedere se ci sono istruzioni di store che stanno agendo sullo stesso indirizzo di memoria. Se ci sono la load non è mandata al buffer finché la store che lavorava sullo stesso indirizzo non viene completata per evitare RAW.

  Non si controlla se ci siano altre LOAD perchè il RAR non è un mai un hazard.

Per le store ogni volta che una store è pronta per l'issue

- si controlla sia lo store che il load buffer per vedere se ci sono altri istruzioni di store che lavorano sullo stesso indirizzo. Se ci sono la store non è mandata al buffer finché tutte le store (WAW) e le load (WAR) precedenti che lavoravano sullo stesso indirizzo sono state completate.

  Devo controllare sia se ci sono LOAD perchè nel caso avrei un WAR sia se ci sono altre store perchè nel caso avrei un WAW.

Conlusioni

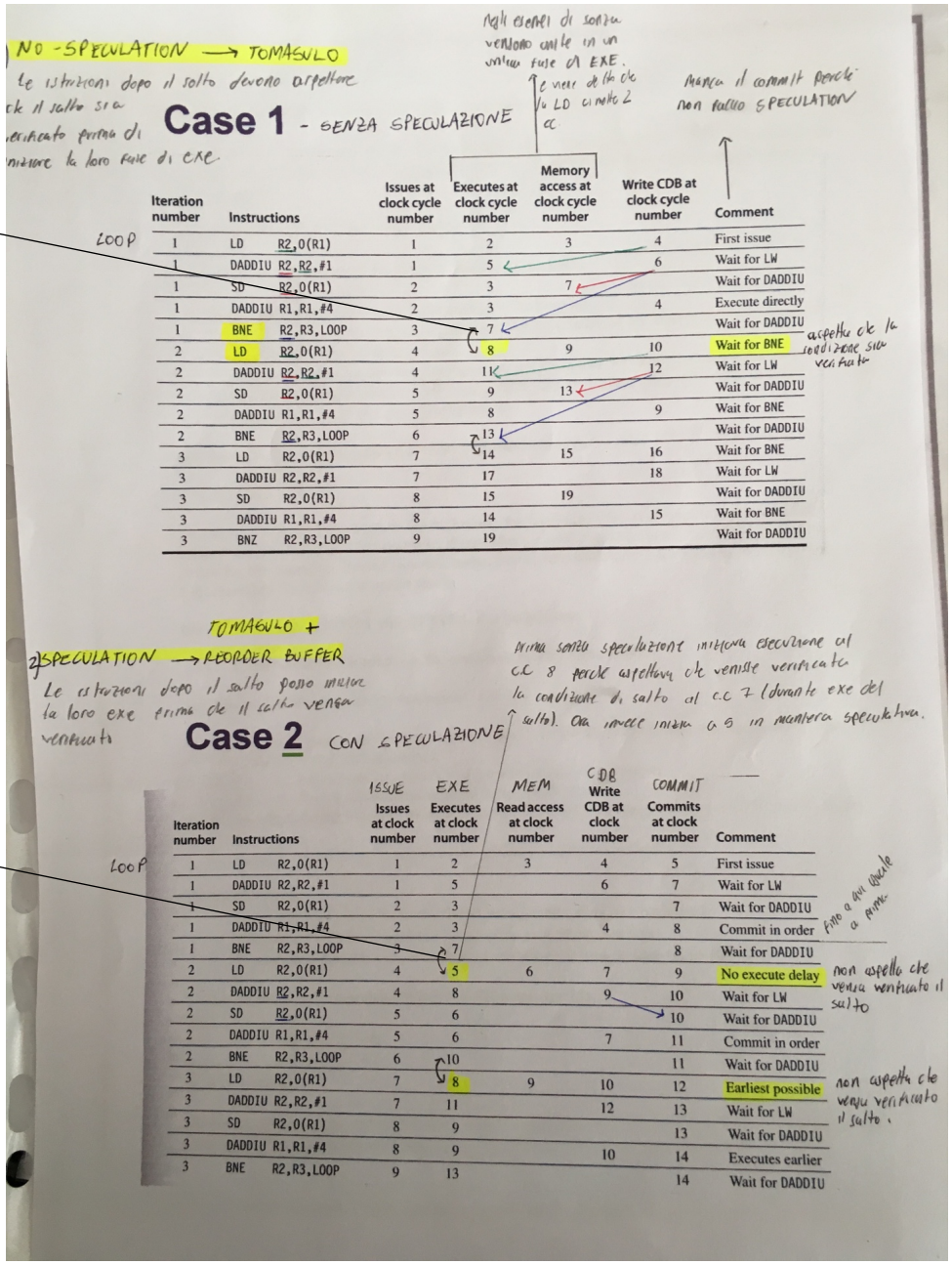
In tomasulo abbiamo l'ISSUE delle istruzioni in ordine, l'execution out-of-order ed anche il write-result è out-of-order perché quando un'istruzione ha calcolato il risultato lo scrive direttamente sul CommonDataBus.

Con l'introduzione del reorder buffer invece aggiungeremo anche il commit delle istruzioni in ordine e quindi anche la loro scrittura portando notevoli migliorie nella gestione dei salti e nella gestione delle eccezioni.

Gestione dei salti in Tomasulo con e senza speculazione

L'istruzione dopo il salto non viene eseguita finché il salto precedente non è completato. Nella fase di exe si saprà se predizione corretta o no.

L'istruzione dopo il salto viene eseguita in maniera speculativa, se la predizione fosse sbagliata si svuota reorder buffer.



NO-SPECULATION → TOMASULO

Le istruzioni dopo il salto devono aspettare che il salto sia verificato prima di iniziare la loro fase di exe

Case 1 - SENZA SPECULAZIONE

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SD R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LD R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SD R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LD R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SD R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

TOMASULO + SPECULATION → REORDER BUFFER

Le istruzioni dopo il salto possono iniziare la loro exe prima che il salto venga verificato

Case 2 CON SPECULAZIONE

Iteration number	Instructions	ISSUE Issues at clock number	EXE Executes at clock number	MEM Read access at clock number	CDB Write CDB at clock number	COMMIT Commits at clock number	Comment
1	LD R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SD R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SD R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SD R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	13			14	Wait for DADDIU

HARDWARE BASED SPECULATION

C'è un'ipotesi forte sulla capacità di fare predizioni. Abortire le istruzione prese dopo la decisione di un salto può essere critico.

Si implementa la speculation la cui idea di base è di dire che se non voglio peggiorare le prestazioni per un salto, faccio la previsione sapendo che le istruzioni che sto eseguendo posso essere quelle giuste oppure no. La correttezza delle istruzioni dipende dalla previsione -> speculation.

Speculation esecuzione di istruzioni che dipendono da una previsione.

Se le istruzioni che eseguono dipendono da una previsione sbagliata devo poterle abortire facilmente cancellando tutti i loro effetti.

Per gestire questo si introduce ulteriore struttura dati nella quale vanno a finire le istruzioni al termine della loro esecuzione —> REORDER BUFFER.

Al termine del completamento le istruzioni finiscono nel reorder buffer, struttura in cui ci sono tutte le informazioni necessarie per poter scrivere il risultato prodotto da quelle istruzioni , che però non è ancora scritto. La scrittura non avviene finché non viene effettuato il commit dell'istruzione , sempre che quelle precedente abbiano fatto commit, sennò non posso.

Se invece si scopre che la predizione era sbagliata si svuota ReorderBuffer e il risultato di quelle istruzioni tanto non era ancora stato scritto —> meglio.

Il reorder buffer garantisce il commit in ordine. *TOMASULO CON SPECULATION*

Prima invece se avessi dovuto abortire delle istruzioni loro avrebbero già scritto il risultato nel registerFile, con il reorder buffer no. *TOMASULO SENZA SPECULATION*

ROB and registerFile - RAT

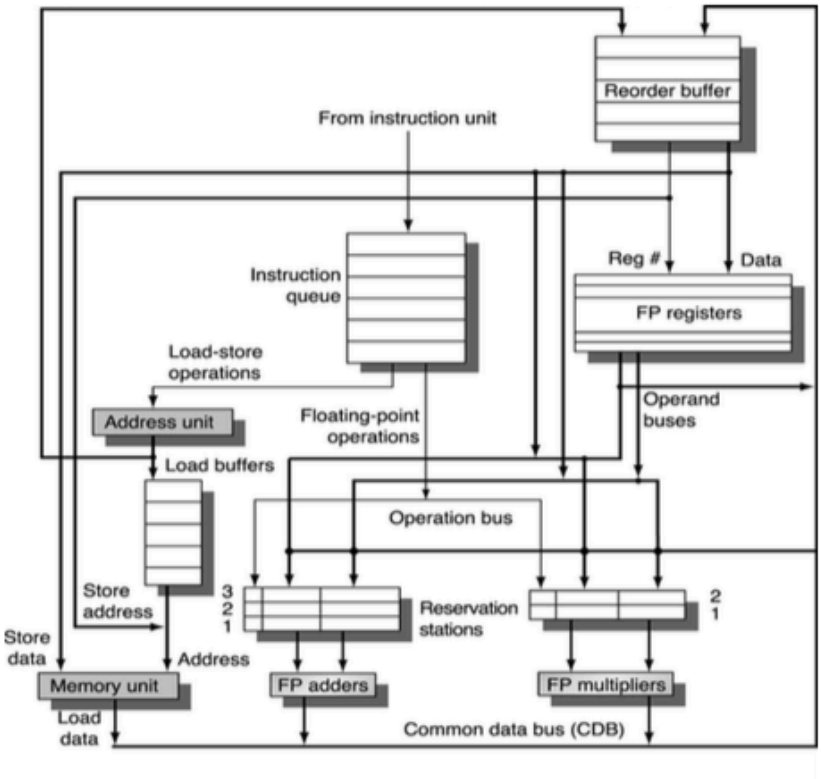
Si identifica l'istruzione attraverso lo slot del ReorderBuffer a cui viene assegnata l'istruzione. Questo perchè i dati non vanno presi più dal commonDataBus ma direttamente dal ReorderBuffer.

Quando operandi non disponibili quindi scriverò l'identificatore dello slot del reorder buffer.
Mentre prima l'identificativo dell'istruzioni era l'identificativo dello slot della reservation station.

RAT Register alias table, si occupa del register renaming dei registri con l'identificativo del ROB. Se più istruzioni voglio scrivere nello stesso registro sovrascrivo l'alias.

Architettura del ROB

Instruction Type	**Destination**	**Value**	**Ready**
branch, store or register	Register or Memory address Indica il registro risultato dell'istruzione.	Contiene il risultato dell'istruzione che deve ancora fare commit (infatti si trova nel reorderBuffer)	Indica che l'istruzione ha finito la sua fase di execute ed è pronta a fare commit.

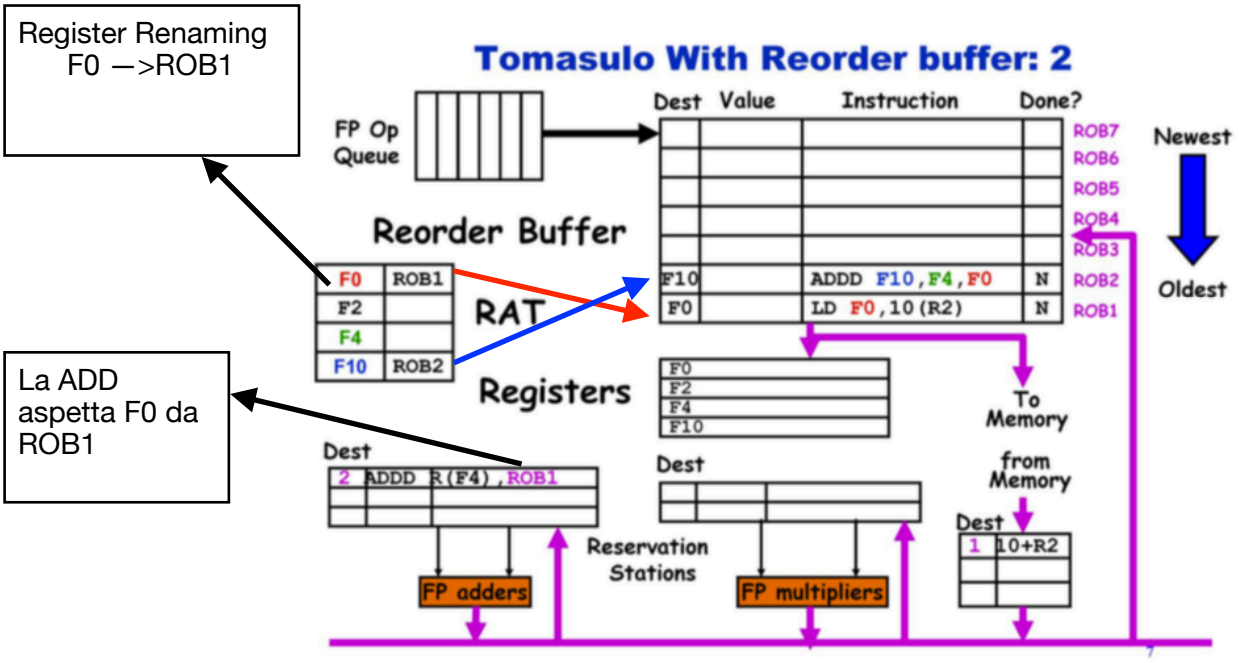


Instruction Execution Steps

- Issue istruzioni estratta da Instruction queue se c'è un posto libero in una reservationStation e un posto libero nel reoderBuffer, altrimenti si va in stallo.
  Gli operandi vengono inviati alla reservationStation se si trovano nel r registerFile o nel ReoderBuffer. Lo slot del ReoderBuffer viene usato per identificare l'istruzione all'interno della reservationStation.
- Execute *quando istruzione ha tutti gli operandi e unità funzionale libera. Con la reservation station evitiamo RAW.* La durata dipende dal tipo di istruzione e dal tipo di unità funzionale. Gli operandi posso essere presi anche direttamente dal commonDataBus.
- WriteResult La scrittura avviene sul common data bus. Colpo di clock in cui unità funzionale termina esecuzione, valore prodotto disponibile viene messo su common data bus con identificatore del reorder buffer. In questa fase il reader buffer viene scritto con i valori prodotti

dall'istruzione.

- Commit svuoto reorderBuffer se predizione era sbagliata. Oltre a far questo devo anche bloccare le unità funzionali che stanno lavorando sulle istruzioni ancora in esecuzione.



WAW and WAR hazards

Non posso più avere WAW and WAR perché grazie al dynamic renaming e al fatto che le scritture in memoria avvengono in ordine, grazie al reorderBuffer.

RAW hazard

Memory RAW, vengono evitati imponendo

- **ordine tra varie load che calcolano il proprio indirizzo**
- **impedire alla load di iniziare la sua seconda fase se qualsiasi altro slot del ROB attivo è occupato da una store che lavora sulla stessa destinazione.**

Register RAW, vengono evitate tramite

- **Reservation station e register renaming**

Store instructions

Scrivono in memoria solo dopo aver fatto il commit quindi hanno bisogno dell'operando nella fase di commit e non in quella di writeResult.

Exception Handling

Non servo istruzione quando viene scatenetata eccezione, *ma segno un flag nel reorder buffer segnando quale istruzione ha scatenato l'eccezione.*

***L'eccezione non viene servita finché l'istruzione non fa commit* e tutte le istruzioni successive vanno abortite finché non si ritornerà dalla routine di gestione. In questo modo ho la sicurezza che tutte le istruzioni precedenti sono state completate.**

Se eccezione asincrona nei processori CISC viene associata all'ultima istruzione di cui ho fatto fetch.

Nei processori RISC invece dipende dalla latenza dell'interrupt oppure associo alla prossima istruzione che farà commit.

Expensive events

Solitamente se devo eseguire un evento molto costoso in termini di tempo, aspetto che non diventi più speculativo, cioè aspetto di avere la certezza che quelle istruzioni debbano essere davvero eseguite.

Se devo eseguire un evento non costoso in termini di tempo posso anche eseguirlo in maniera speculativa.

MULTIPLE ISSUE PROCESSORS

Processore che limita l'effetto delle dipendenze di dato e di controllo perchè minimizzo penalità legate ai salti tramite scheduling dinamico effettuando contemporaneamente l'ISSUE di due istruzioni per colpo di clock.
Per le dipendenze strutturali non posso fare molto se non aumentare numero di unità funzionali.

Posso quindi ottenere un processore che riduce il numero di stalli, mi avvicino il più possibile a CPI=1.

I processori superscalari **ottengono un CPI<1, completano più istruzioni in un colpo di clock. Questo viene eseguito permettendo al processore di eseguire il *commit di due istruzioni in un solo colpo di clock.***

I due parametri fondamentali sono

- numero di istruzioni di cui faccio issue per cc
- numero di istruzioni di cui faccio commit per cc

Il fatto di poter fare issue e commit di più istruzioni in un solo cc può essere fatto con diversi processori:

- **processori superscalari** (con scheduling sia statico che dinamico)
- **very long instruction word VLIW**

Tipo di scheduling:

- STATICO processore più semplice
- DINAMICO processore più complicato

Processore Superscalare con Scheduling Statico

Posso fare ISSUE di due istruzioni per cc se:

- una è una load, store, branch o integer ALU
- l'altra è una operazione FP

Se rispetto questi vincoli posso fare fetch e il decode di due istruzioni alla volta. Le due istruzioni viaggiano in pacchetti chiamati *ISSUE PACKET.* Una volta le due istruzioni all'interno del pacchetto dovevano essere ordinate (sempre prima le operazioni tra interi) invece ora non devono più seguire un ordine preciso.

Il fetch viene eseguito prendendo dall'instruction memory (cache) due istruzioni. Se le istruzioni appartengono a cache diverse faccio il fetch di una sola istruzione.

Ideal situation

FP instructions are all assumed to last for 3 clock cycles

Instruction type	Pipe stages							
Integer instruction	IF	ID	EX	MEM	WB			
FP instruction	IF	ID	EX	EX	EX	WB		
Integer instruction		IF	ID	EX	MEM	WB		
FP instruction		IF	ID	EX	EX	EX	WB	
Integer instruction			IF	ID	EX	MEM	WB	
FP instruction			IF	ID	EX	EX	EX	WB
Integer instruction				IF	ID	EX	MEM	WB
FP instruction				IF	ID	EX	EX	EX

Quando la prima istruzione è un'istruzione di load, store o move ci può essere una contesa per una porta del FP registerFile.

Le possibili soluzioni sono:

- forzare la prima istruzione ad essere eseguita da sola
- dare al FP registerFile una porta aggiuntiva

Quando la prima istruzione è un'istruzione di load, store o move e la seconda legge il suo risultato ci può essere un hazard RAW. In questo caso bisogna ritardare la seconda istruzione inserendo uno stallo.

Con MIPS superscalare statico sia il LOAD DELAY SLOT che il BRANCH DELAY SLOT diventano di 3 istruzioni.

Processore Superscalare con Scheduling Dinamico

Può essere ottenuto adottando un'architettura simile a quella di tomasulo. Per semplicità le istruzioni non vengono mai issued out-of-order.

Il alcuni casi posso avere due istruzioni che vogliono scrivere nello stesso istante sul common data bus. Una possibile soluzione è di duplicare il CBD.

VLIW PROCESSORS

Sono processori in cui non si vuole che il processore, cioè l'HW

- rilevi e gestisca le dipendenze
- implementi scheduling dinamico
- implementi branch predicton e speculation.

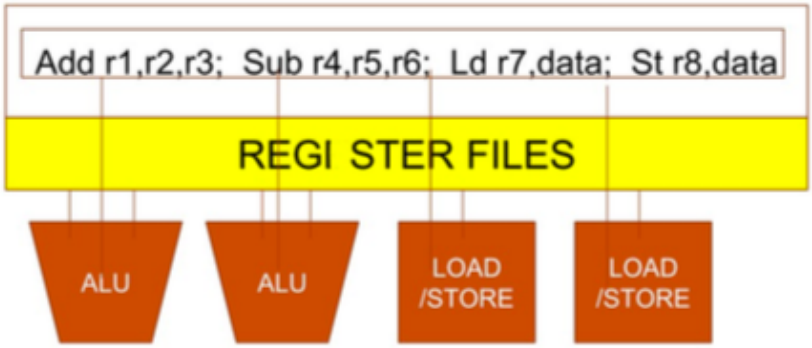
Si torna quasi a processore CISC ma con tante unità funzionali in parallelo.

Ciascuna istruzione è composta da più istruzioni. Ne codifico 4 in una sola. (esempio).

Chi produce VLIW offre una piattaforma software con diverse unità funzionali e facendo girare il mio codice capisco di quali e quante unità funzionali avrò bisogno.

Il processore diventa solo un esecutore delle istruzioni tutto il lavoro (impachettare istruzioni insieme, unrolling di loops, scheduling code ecc) viene svolto dal compilatore che diventa quindi più complicato quindi Software più complicato.

L'hardware diventa però più semplice perchè non fa nessun controllo sulle possibile dipendenze tra istruzioni.



Stalli

Quando un'operazione richiede uno stallo, vengono mandate in stallo tutte le istruzioni del pacchetto in modo da preservare il dataFlow deciso dal compilatore.

Il vantaggio principale dei VLIW è che l'harware come detto prima è più semplice, quindi l'HW non ha bisogno di scegliere l'istruzione da essere eseguita in parallelo tenendo conto delle dipendenze.

Le principali limitazioni di questi processori sono dovute a

- difficile produzione dell'hardware
- limitazioni specifiche di ogni processore VLIW
- Istrunction level Parallelism I.L.P

Limitazioni

Instruction level parallelism

è difficile trovare un numero indipendente di istruzioni da essere eseguite in parallelo.
Ed è ancora più difficile se devo anche parallelizzare le unità funzionale con latenze maggiori di 1. Solitamente per evitare stalli devo trovare un numero di istruzioni indipendente pari a

(pipeline depth) x (functional units)

Hardware

Aumentando il numero di unità funzionali c'è anche un aumento degli accessi al registerFile e alla memoria. Questo vuol dire aumentare la complessità dell'HW e eventualmente diminuire le prestazioni.

Le possibili soluzioni sono:

- memory interleaving
- multiport memories
- multiple access per clock cycle memories

Dimensione codice

In un processore VLIW il codice è più grande per due motivi:

- i loop sono solitamente sono unrolled per aumentare il parallelismo
- slot vuoto per la decodifica delle istruzioni (UNFILLED OPCODE)

Solitamente le istruzioni sono salvate in un formato compresso in memoria e poi espanse quando caricate dal processore.

Accesso memoria

In un processore VLIW l'accesso in memoria è spesso un collo di bottiglia.
Questo perchè come detto prima aumentano gli accessi dovuti al maggior numero di unità funzionali e gli stalli dovuti ai miss nella cache causano lo stallo dell'intero processore perchè tutte le istruzioni sono sincronizzate con un ordine scelto dal compilatore.

Binary code compatibility

Ogni cambiamento in un processore VLIW come modificare la latenza di un'unità funzionale richiede la ricompilazione del codice.
É un grande svantaggio in confronto ai processori superscalari i quali potevano essere facilmente essere resi compatibili con versioni precedenti.

VANTAGGI VLIW	SVANTAGGI VLIW
Le dipendenze sono determinate dal compilatore in quale schedula e impacchetta le istruzioni in accordo con la latenza delle unità funzionali	Maggior complessità del compilatore (software)
Le unità funzionali sono assegnate dal compilatore e corrispondono alla posizione all'interno del pacchetto che contiene le istruzioni	Problema della compatibilità. Per ogni cambiamento c'è bisogno della ricompilazione del codice.
Ridotta complessità dell'hardware	Cache miss mandano in stallo intero processore
La decodifica, riconoscimento delle dipendenze di dato, issue delle istruzioni diventano più semplici	In caso di UNFILLED OPCODE viene sprecato molto spazio in memoria e parte dell'istruzione viene sprecata.
Clock rate più elevato	Il codice diventa più grande.
Basso consumo di energia	

Applicazioni

Utili per digital Signal processing. Elaborazione di data Coe compressione e decompressione di immagini.

Esempi architetture

- **TRIMEDIA TM32** ogni istruzione contiene 5 operazioni. Scheduling statico e non vengono identificati hazard. Sviluppato da NXP (Philips semiconductor).
- **TRANSMETA CRUSOE** Ottimo per tecnologie mobili a basso consumo di energia. Garantisce compatibilità con instruction set x86. Ogni istruzioni può contenere 2 o 4 operazioni.
- **EPIC** **Explicit parallel instruction computing.** Architettura VLIW ma molto simile ad un processore superscalare.

ARM

	Grouping	Fn unit asgn	Initiation
Superscalar	Hardware	Hardware	Hardware
EPIC	Compiler	Hardware	Hardware
Dynamic VLIW	Compiler	Compiler	Hardware
VLIW	Compiler	Compiler	Compiler

CORTEX-A8, INTEL CORE i7

martedì 18 dicembre 2018

ARM CORTEX-A8 implementa scheduling statico perchè tanto va bene per applicazioni Embedded.

INTEL CORE i7 usa scheduling dinamico con architettura di tomasulo.

ARM CORTEX A8

Primo superscalare di ARM con scheduling statico. Unità di issue si basa sul lavoro del compilatore. Utilizza branch target buffer.

Usato su dispositivi mobile, smartphone e tablet.

Numero elevato di stadi di pipeline

- 3 stadi di fetch
- 5 stadi di decode
- 6 stadi di esecuzione

Pipeline molto profonda. Ad ogni colpo di clock in processore fa il fetch di due istruzioni, quelle che sono in memoria una dopo l'altra. È il compilatore che riordina il codice —> scheduling statico, cerca di evitare hazard e quindi stalli.

La decode unit è un'unità che richiede 5 colpo di clock con architettura a pipeline, ad ogni cc due istruzioni entrano dentro decode unit.

La MUL ha architettura a pipeline.

Stalli per

- hazard strutturali — due istruzioni che usano stessa unità funzionale
- data hazard
- control hazard — misprediction penality 13 cc, devo svuotare pipeline

ARM CORTEX A9

Superscalare con scheduling dinamico.

INTEL i7

Introdotto da intel a 64 bit nel 2008. Speculazione, scheduling dinamico, out-of-order. Basato su X-86 quindi con instruction set CISC che però viene trasformato in hardware in istruzioni RISC (microOp istruction).

THREAD LEVEL PARALLELISM

Thread processo che posso trattare in maniera autonoma con sue variabili e sua zona di memoria. Il thread può essere inteso come
- processo all'interno di una programma costituito da più processi
- un processo indipendente

ILP E TLP

ILP e TLP posso essere combinati, un processore ILP oriented può usare TLP come una sorgente di istruzioni indipendenti da assegnare alle varie unità funzionali il più possibile.

Multhithreading

Permette a diversi threads di condividere le unità funzionali di un singolo processore.

Richiede

- Salvataggio dello stato di ogni thread (register file, PC, page table ecc) PSW
- Accessi alla memoria indipendenti (virtual memory)
- Meccanismi efficienti per passare da un thread all'altro

Eseguo più thread contemporaneamente. Si salva indipendentemente dagli altri il contesto di ogni processo.

Si cerca di evitare che il processore non faccia nulla quando un processo lo manda in stallo. Potrei fare contextSwitch molto più veloce, anche ad ogni CC. Devò però riuscire a salvare il contesto altrettanto velocemente.

Posso investire in HW in maniera tale da rendere men costoso possibile in termini di tempo il passaggio da un thread all'altro, ContestSwitching.

Ogni volta che faccio contextSwitch devo cambiare PC, cambiare zona di memoria e i registri.

Multithreading type

- Fine-grained Lo switch tra un thread ed un altro avviene ad ogni colpo di clock.
- Coarse-grained il passaggio da un thread all'altro non avviene il CC di clock successivo allo stallo, ma solo quando mi accorgo che il thread resterà in stallo per un pò. Guardo quanto è grave la penalità.

- Simultaneous (SMT) — Cerco di costruire architettura che non fa threadSwitching ma che esegue davvero istruzioni in parallelo prese da vari thread. È un ibrido tra fine grained multithreading e i principi dei processori superscalari.

Fine grained

L'esecuzione dei vari thread è intervallata. La CPU deve essere in grado di vare lo switch ad un qualsiasi colpo di clock.

Vantaggi
Quando un istruzione va in stallo ne trovo subito un altra da far partire in altri thread senza perdita di performance.

Svantaggi
I thread potrebbero essere rallentati dall'esecuzione concorrente di altri thread.

Coarse grained

Lo switch viene eseguito solo dopo stalli molto costosi.

Vantaggi
????

Svantaggi
La perdita di performance causata da stalli corti non può essere evitata

Simultaneous Multithreading-SMT

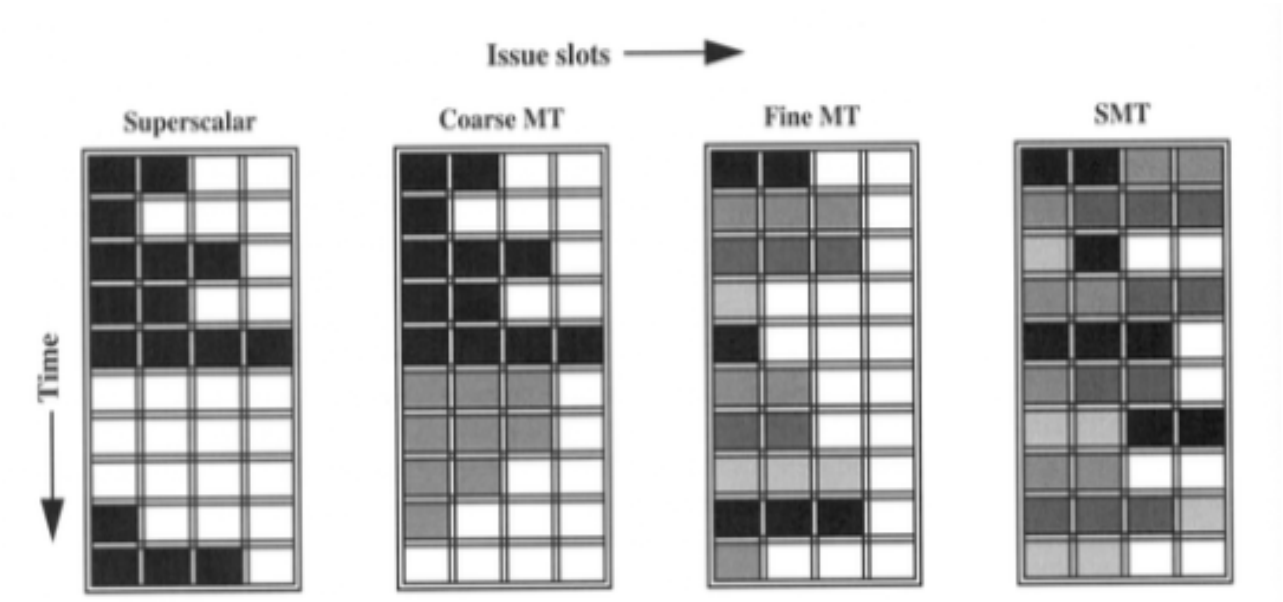
Tipo particolare di fine grained multithreading che permette ad un processore superscalare di combinare ILP e multitrheading contemporaneamente.

Diverse istruzioni prese da thread indipendenti possono essere issued tramite dynamic scheduling , usando questa capacità per risolvere le dipendenze esistenti tra di loro.
Il fatto di poter gestire istruzioni prese da più thread contemporaneamente è dovuto al fatto di avere :

- un grande numero di registri disponibili
- register renaming

Il sistema operativo fa in modo di sincronizzare i processi tramite timeSlices relativamente piccoli, così sembra che i processi vadano avanti in parallelo. ContextSwitch, si passa da un processo all'altro. SMT può essere implementato su un processore che supporti out-of-order avendo:

- renaming table per ogni ogni thread
- PC diversi per ogni trhead
- diversi ROB per ogni thread così faccio commit di istruzioni di thread diversi.



Istruzioni fanno parte della stessa esecuzione del processo. Se vado in stallo, aspetto e ricomincio	Si fa cambio di thread quando lo stallo è troppo lungo	Ad ogni CC si fa cambio di thread	Esecuzione di diversi thread in parallelo.

Alternativa SMT

Invece di investire per cercare di fare processori più veloci che implementino SMT si cerca di mettere sullo stesso dispositivo più cores in parallelo.

SUN T1 8 processori ognuno FineGrained.

Quando ho multicore con cache condivise il problema pè che quando un cores aggiorna una cache i valori aggiornati devono subito essere pronti per gli altri cores.

GPU-GRAPHIC PROCESSING UNIT

La CPU ha iniziato a vedere la GPU come un coProcessore a cui lasciare i compiti della parte grafica. È il chip che si trova nelle schede video.

GPU si occupa della parte grafica di un sistema di elaborazione. C'è la necessità di lavorare su grandi matrici, devo fare molti calcoli su matrici. Si cercano di rendere il più veloce possibile le operazioni aritmetiche su matrici.

Oltre che alla grafica potrei usare le GPU per applicazioni scientifiche, ci sono le GPGPU usate per HPC (high performance computing) o per assistenza alla guida.
Le GPGPU sono trattate come coprocessori a cui affidare i compiti più costosi.

Architettura GPU

Tante unità funzionali, quindi tante ALU ognuno con una sua memoria locale e con memorie condivise. Sfruttate soprattuto per programma con istruzioni aritmetiche intense e ratio elevato tra operazioni aritmetiche e operazioni in memoria.

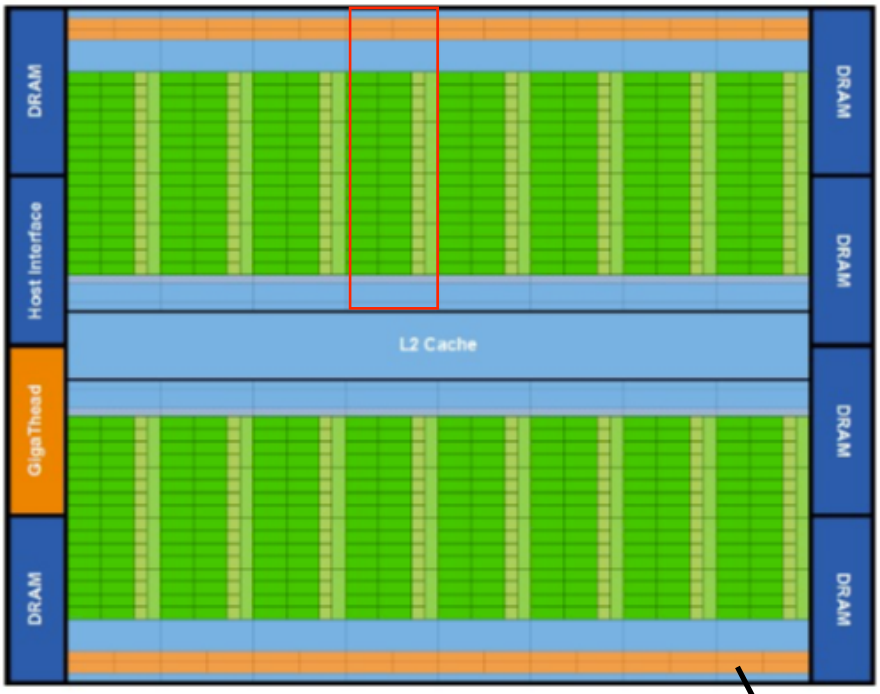
CUDA - Compute Unified Device Architecture

Architettura Hardware e Software per GPU NVIDIA per eseguire programmi in diversi linguaggi. Il concetto principale è che l'hardware supporta gerarchie di thread.

Fermi Architecture

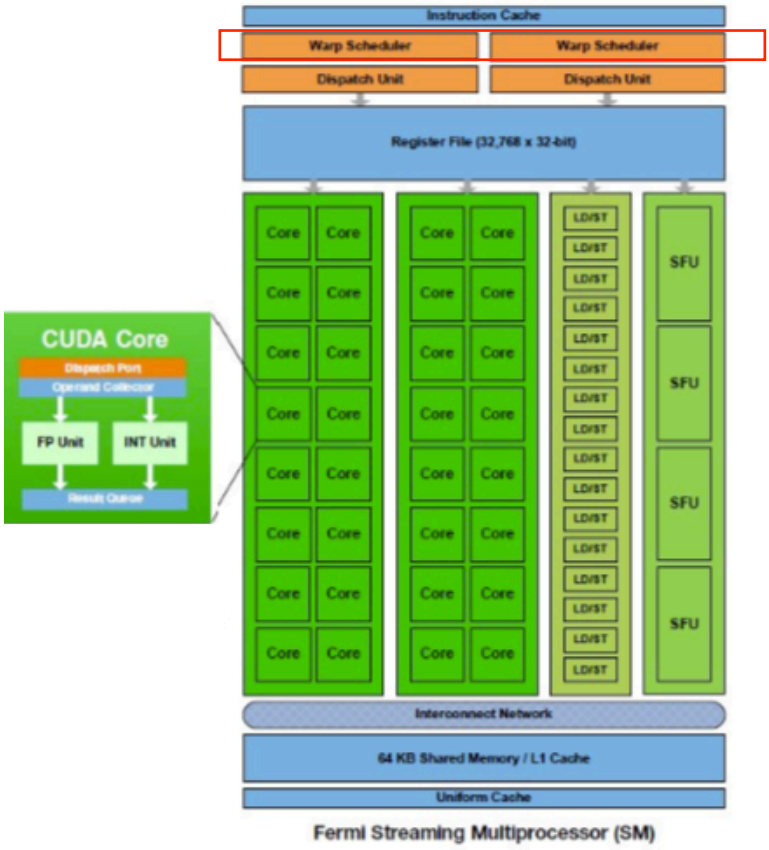
Architettura utilizzata nella famiglia NVIDIA T20.

- 16 stream-multiprocessors ognuno da 32 CUDA cores (rettangolo rosso)
  Ogni processore può eseguire una FP o integer instruction ad ogni CC per un thread.



Fermi Streaming Multiprocessor

32 cuda Cores con ALU pipelined e FPU (floating Point unit)
Esegue gruppo di 32 thread chiamati ***warp****.*



Fermi Streaming Multiprocessor (SM)

SIMT and WARP Scheduler

SIMT (single instruction multi thread, modello di esecuzione)

- tutti i thread dello stesso warp partono allo stesso PC ma sono liberi di eseguire salti indipendentemente.
- un WARP esegue un'istruzione comune comune a tutti i thread. SIMD (single instruction multiple Data)

Ci sono due Warp scheduler per ogni streaming multiprocessor, (rettangolo rosso)

CPU and GPU

La CPU da i lavori più costosi alla GPU, durante i quali la CPU è Busy e controlla l'esecuzione da parte della GPU.
Ci possono essere casi di collo di bottiglia nella connessione tra memoria centrale e la memoria della GPU

- I dati devono essere copiati nella GPU per essere usati e i risultati devono essere copiati in memoria centrale per essere salvati

Le GPU inoltre hanno dei vincoli quali

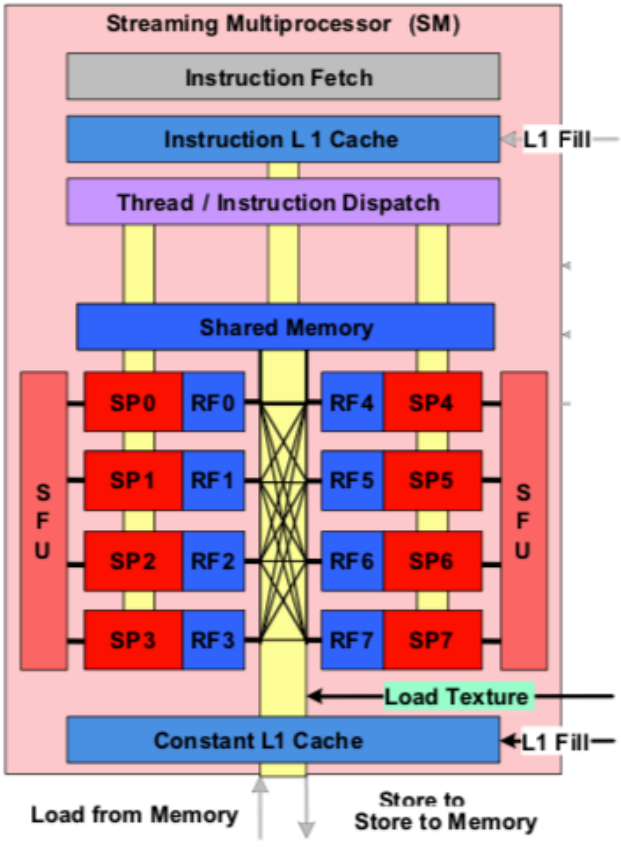
- **Dover usare dei modelli di programmazione appositi per le GPU come CUDA C o OpenCL.**
- **Avere capacità e risorse limitate perchè il codice è dipendente dalla piattaforma.**

Streaming Multiprocessor 8800 GTX GPU

Architettura precedente a quella della famiglia T20 Fermi.

Caratteristiche

- 8 streaming processor
- 2 super function units SFU, per calcolo di funzioni particolari come sin, cos log ecc.
- local register file RF, nell'architettura di fermi invece ne avevamo uno condiviso
- 16KB shared memory. CUDA sfrutta la memoria condivisa come risorsa visibile da tutti i thread all'interno dello stesso thread block. Tutti avranno accesso a scrittura e lettura.

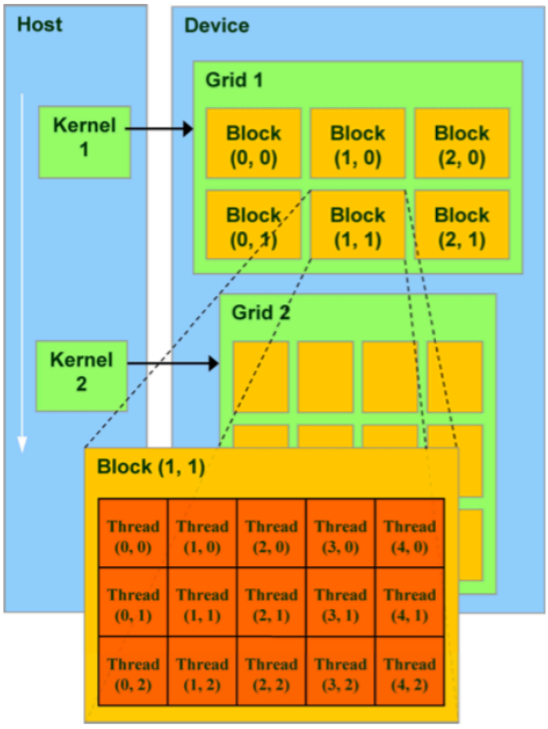


GPGPU

Si adotta una modello di programmazione general purpose, in cui la GPU è vista come un "Device" in grado eseguire blocchi di thread.

Usate per HPC (high performance computing) o per assistenza alla guida.
Le GPGPU sono trattate come coprocessori a cui affidare i compiti più costosi.

Cuda Programming model for GPGPU



La GPU è vista come un *Device* che

- è un coprocessore della CPU, che viene chiamata *host* (la CPU)
- ha la sua DRAM chiamata *Device memory*
- esegue diversi thread in parallelo

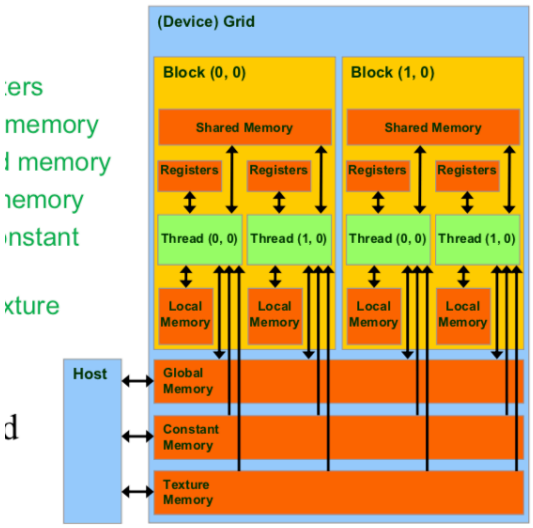
Le parti che possono essere eseguite in parallelo di una applicazione sono eseguite sul Device come *Kernel*, e vengono eseguite in parallelo su diversi thread.

Thread Batching

Un kernel è eseguito come una griglia di blocchi di thread, *Grid*, i quali condividono la stesso spazio in memoria.

Il thread Block è un blocco di thread che possono cooperare reciprocamente sincronizzando la loro esecuzione, altrimenti si avrebbero delle pessime performance.

Due thread di due blocchi diversi non possono cooperare.



I thread e I blocchi sono identificati tramite degli ID che semplificano così l'indirizzamento in memoria.

Ogni thread può:

- *scrivere e leggere* i propri registri
- *scrivere e leggere* la memoria locale
- *scrivere e leggere* la memoria condivisa
- *scrivere e leggere* la memoria globale
- *leggere* la memoria costanti
- *leggere* la texture memory

L'host può:

- scrivere e legger la memoria globale, costanti e texture.

Memoria globale
Principale mezzo di comunicazione tra host e Device. Il suo contenuto è visibile da tutti i thread.
Texture e memoria costanti
La memoria constanti viene inizializzata dall'host. Il contenuto è visibile da tutti i thread.