Politecnico di Torino

Architetture e Sistemi di elaborazione

Anno accademico 2022-2023

Marco Lampis

28 dicembre 2022

Indice

0 Informazioni **1**

1 Introduzione **3**

1.1 Dependability evaluation . . . 3
1.2 Computer Performance . . . 3
1.3 Linee guida e principi per il computer design . . . 4
1.3.1 Legge di Amdahl . . . 4
1.3.2 CPU performance equation . . . 5

2 Instruction Set **7**

2.1 GPR Machines . . . 7
2.2 Memory Addresses . . . 8
2.3 Control Flow Instruction . . . 9
2.4 Tipo e dimensione degli operandi . . . 9
2.5 Instructions Set Characteristics . . . 10

3 MIPS64 **11**

3.1 Metodi di indirizzamento . . . 12
3.1.1 Indirizzamento immediato . . . 12
3.1.2 Indirizzamento con displacement . . . 12
3.2 Formato delle istruzioni . . . 13
3.2.1 Immediato . . . 13
3.2.2 Registro . . . 13
3.2.3 Salto . . . 13
3.3 Instruction Set . . . 14
3.3.1 Load and store . . . 14
3.3.2 Operazioni ALU . . . 14
3.3.3 Branch e jump . . . 15

4 Pipeline **17**

4.1 Versione senza pipeline . . . 17
4.2 Versione pipelined . . . 18

4.3 Pipeline performance . . . 18
4.4 Pipeline hazards . . . 18
4.4.1 Stalls . . . 19
4.4.2 Structural hazards . . . 19
4.4.3 Data hazards . . . 19
4.4.4 Control hazards . . . 21
4.5 Multicycle operations . . . 23
4.5.1 Floating Point operations . . . 23
4.5.2 Multi-cycle Hazards . . . 24
4.5.3 MIPS R4000 Pipeline . . . 25
4.6 Instruction level Parallelism . . . 25
4.6.1 Basic Block . . . 26
4.6.2 Loop level parallelism . . . 27
4.6.3 Dipendenze . . . 28
4.7 Exceptions and Interrupts . . . 30
4.7.1 Interrupt Protocol in 80x86 . . . 31
4.7.2 Interrupt Protocol in ARM . . . 32
4.7.3 Precise exceptions . . . 32
4.7.4 Imprecise exceptions . . . 32
4.7.5 MIPS: possibile sorgenti di eccezione . . . 33
4.7.6 Eccezioni contemporanee . . . 33
4.7.7 Instruction set complications . . . 34

5 Cache **35**
5.1 Organizzazione . . . 36
5.2 Trovare un blocco in cache . . . 37
5.3 Harvard Architecture . . . 39
5.3.1 Cache Size . . . 39
5.3.2 Mapping . . . 39
5.4 Algoritmo di rimpiazzamento . . . 41
5.5 Memory Update . . . 42
5.5.1 Write Back . . . 42
5.5.2 Write Through . . . 43
5.6 Cache Coherence . . . 43
5.7 Esempio . . . 43

0 Informazioni

I seguenti appunti sono stati presi nell'anno accademico 2022-2023 durante il corso di Architetture dei sistemi di elaborazione.

Il materiale non è ufficiale e non è revisionato da alcun docente, motivo per cui non mi assumo responsabilità per eventuali errori o imprecisioni.

Per qualsiasi suggerimento o correzione non esitate a contattarmi.

E' possibile riutilizzare il materiale con le seguenti limitazioni:

- Utilizzo non commerciale
- Citazione dell'autore
- Riferimento all'opera originale

E' per tanto possibile:

- Modificare parzialmente o interamente il contenuto

Questi appunti sono disponbili su GitHub al seguente link:

```
https://github.com/Guray00/polito_lectures/tree/main/Tecnologie%20e%20Servizi%20di%20Rete
```

Repository GitHub

1 Introduzione

Introduzione al corso

1.1 Dependability evaluation

L'affidabilità è spesso misurata utilizzando:

- MTTF, Mean Time To Failor, oppure in FIT, failure in one bilions hours.
- Mean Time Between Failurs, ovvero il tempo che intercorre tra i guasti
- Mean Time To Repair, ovvero il tempo che intercorre tra il guasto e la riparazione

Le tre misure sono legate dalla seguente formula:

$$MTTF = MTBF + MTTR$$

Per riuscire a garantire un rateo di "zero guasti" si studia la "bathtub curve", ovvero la curva che descrive il numero di guasti in funzione del tempo. La curva è caratterizzata da tre fasi:

- Infant mortality: fase iniziale in cui si verifica un numero elevato di guasti
- Normal life: fase in cui il numero di guasti è costante
- End of Life (EOL): fase in cui il numero di guasti aumenta

1.2 Computer Performance

La performance di un dispositivo può essere analizzata da due punti di vista:

1. **Utente**: la risposta nel tempo.
2. **System Manager**: Throughput, la quantità di lavoro che può essere svolta in una unità di tempo.

Il tempo che deve essere considerato per la performance sono:

- elapsed time: tempo che intercorre tra l'inizio e la fine dell'esecuzione di un programma

- cpu time: user cpu time e system cpu time

La valutazione della performance viene spesso effettuata eseguendo le applicazioni e valutando il loro comportamento. La scelta dell'applicativo inficia particolarmente sull'analisi, ma nel caso ideale si dovrebbe utilizzare un carico di lavoro paragonabile all'utilizzo utente. Per questo motivo si utilizzano i benchmark, ovvero del software su misura che simulano il comportamento di un utente.

I benchmark spesso vengono utilizzati eseguendo algoritmi (es quicksort molto grosso), programmi reali (compilatore C) o applicazioni apposite.

In particolare noi utilizzeremo ***MIBench*** che consente di eseguire test inerenti a vari tipi di applicazioni.

E' importante garantire la riproducibilità dei test, per questo motivo è importante utilizzare uno stesso hardware e software per tutti i test (oltre al programma di input).

Può essere interessante avere una media pesata dei risultati, in modo da poter valutare la performance in base al tipo di applicazione.

1.3 Linee guida e principi per il computer design

Le linee guida per la misurazione della performance si basano su due principi:

- legge di Amdahl
- CPU performance equation

1.3.1 Legge di Amdahl

La legge di Amdahl è una formula che descrive il miglioramento della performance in funzione del numero di processori. La formula è la seguente:

$$\text{speedup} = \frac{\text{performance with enhancement}}{\text{performance with without enhancement}}$$

Lo speedup risultante da un miglioramento dipende da due fattori:

- fraction enhanced: la frazione del tempo di computazione che può essere migliorata
- speedup enhanced: la dimensione del miglioramento che le parti ricevono.

$$\text{execution time new} = \text{execution time old} * ((1 - \text{fraction enhanced}) + \frac{\text{fraction enhanced}}{\text{speedup enhanced}})$$

$$\text{speedup overall} = \frac{\text{execution time old}}{\text{execution time new}} = \frac{1}{(1 - \text{fraction enhanced}) + \frac{\text{fraction enhanced}}{\text{speedup enhanced}}}$$

1.3.1.1 Esempio 1

Supponiamo di avere una macchina che è 10 volte più veloce nel 40% dei programmi che girano. Quale è lo speedpup totale?

$$\text{fraction enhanced} = 0.4$$

$$\text{speedup enhanced} = 10$$

$$\text{speedup overall} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56$$

1.3.1.2 Esempio 2

Sono disponibili due soluzioni per migliorare la performane di una macchina floating point:

- *soluzione 1*: aumentando di 10 le performance delle radici quadrate (circa il 20% del tempo di esecuzione) aggiungendo un hardware dedicato.
- *soluzione 2*: aumentare di 2 la performance di tutte le operazioni floating point (circa il 50% del tempo di esecuzione).

Quale soluzione rende più rapida la macchina? Per rispondere è sufficiente riapplicare la legge di Amdahl.

$$\text{speedup1} = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = 1.22$$

$$\text{speedup2} = \frac{1}{(1 - 0.5) + \frac{0.5}{2}} = 1.33$$

La soluzione 2 è più vantaggiosa.

1.3.2 CPU performance equation

Per misurare il tempo richiesto per eseguire un programma sono utilizzabili 3 approcci:

1. osservando il **sistema reale**

2. effettuando delle **simulazioni** (molto costoso)
3. utilizzando la **CPU performance equation**

La terza opzioni consiste nel calcolo della seguente formula:

$$\text{CPU time} = (\sum_{i=1}^{n} CPIi * ICi) * \text{clock cycle time}$$

- **CPI**: clock cycles per instruction
- **IC**: instruction count, ovvero il numero di istruzioni
- **clock cycle time**: è l'inverso della frequenza del clock

2 Instruction Set

L'instruction set Architecture (ISA) è come il computer è visto da un programmatore e dal compilatore. Ci sono molte alternative per un designer ISA, che possono essere valutati in base a vari criteri:

- performance del processore
- complessità del processore
- complessità del compilatore
- dimensione del codice
- consume energetico
- ecc

Le CPU sono spesso classificate in accordo al loro tipo di memoria interna:

- **stack**: unicamente dalla memoria
- **accumulatori**: dalla memoria e da un accumulatore, il secondo risulta sempre la destinazione
- **registri**
  - register-memory (utilizzo di registro e memoria)
  - register-register (unicamente mediante registri)
  - memory-memory

Stack	Accumulator	Register (register-memory)	Register (load-store)
`Push A`	`Load A`	`Load R1,A`	`Load R1,A`
`Push B`	`Add B`	`Add R1,B`	`Load R2,B`
`Add`	`Store C`	`Store C,R1`	`Add R3,R1,R2`
`Pop C`			`Store C,R3`

Figura 2.1: Esempio di codice

2.1 GPR Machines

Attualmente tutti i processori utilizzano General Purpose Register senza accedere direttamente alla memoria. Non hanno dunque dei ruoli specifici, anche se arm in alcuni casi fa eccezione. Questo è un

favore perché risultano più veloci rispetto alla lettura in memoria ed è più semplice per il compilatore per gestire le variabili.

2.2 Memory Addresses

Esistono due tipi di memorizzazione in accordo all'andianess:

- **big endian**: il byte con l'indirizzo più piccolo viene salvato nella posizione più significativa. L'indirizzo del dato è quello del most significant byte.
- **little endian**: il byte con l'indirizzo più piccolo viene salvato nella posizione meno significativa. L'indirizzo del dato è quello del least significant byte.

Dunque se leggiamo un dato nel modo sbagliato avremo i dati invertiti.

I dati possono essere salvati in modo:

- **allineato**: le letture allineate rappresentano una limitazione per l'accesso in memoria (nel nostro caso sarà sempre allineato)
- **non allineato**: è il caso di intel x86, dove le istruzioni possono avere lunghezza differente, causando overhead sia per le performance sia per l'hardware

La memoria può essere acceduta in tre differenti modi:

- **registro**: `ADD R4, R3`
- **immediato**: `ADD R4, #3`
- **displacement**: `ADD R4, 100(R1)`
- **indiretto**: `ADD R4, (R1)`
- **indexed**: `ADD R3, (R1+R2)`
- **diretto (o assoluto)**: `ADD R1, (1001)`
- **memory inderect**: `ADD R1, @(R3)`
- **autoincrement**: `ADD R1, (R2)+`
- **autodecrement**: `ADD R1, -(R2)`
- **scaled**: `ADD R1, 100(R2)[R3]`

Scegliere una metodologià piuttosto che un'altra può portare a ridurre il numero di istruzioni o aumentare la complessità dell'architettura CPU o aumentare l'average CPI. Quello più diffuso è sicuramente con displacement. La dimensione dell'indirizzo in modalità displacement dovrebbe essere tra 12 e 16 bit mentre la dimensione per la immediate field dovrebbe essere tra 8 e 16 bit.

2.3 Control Flow Instruction

Le istruzioni di controllo possono essere divise in quattro categorie:

- conditional branch
- jumps
- procedure calls
- procedure returns

Gli indirizzi di destinazione sono normalmente specificati come displacement rispetto al valore corrente del program counter. In questo modo:

- riduciamo il numero di bit, in quanto l'istruzione target è spesso molto vicina da quella sorgente
- il codice diviene indipendente dalla posizione

Le chiamate a procedure e i salti indiretti mediante registri consentono di scrivere codice che include salti che non sono conosciuti a tempo di compilazione e di implementare case o switch statements. Supporta le librerie condivise dinamicamente e le funzioni virtuali (chiamare differenti funzioni in base al tipo di dato)

Nel caso di utilizzo di procedure, alcune informazioni devono essere salvate:

- il return address
- i registri utilizzati
  - caller saving
  - callee saving

Riassumendo: Poche istruzioni sono realmente indispensabili come load, store, add subtract, move register-register, and, shift compare equal, compare not equal, branch, jump, call e return. Branch displacements relativo al program counter dovrebbe essere di almeno 8 bit, mentre register-indirect e PC-relative addressing possono essere utilizzati nelle chiamate alle procedure e ritorno.

2.4 Tipo e dimensione degli operandi

Gli operandi supportati sono:

- **char**: 1 byte
- **half word**: 2 byte
- **word**: 4 byte

- **double word**: 8 byte
- **single-precision floating point**: 4 byte
- **double-precision floating point**: 8 byte

2.5 Instructions Set Characteristics

L'encoding dell'instruction set dipende dalle istruzioni che compongono il set e dai metodi di indirizzamento supportati. Quando un gran numero di metodi di indirizzamento sono supportati, un indirizzo specificato in un campo è utilizzato per specificare l'addressing modo e il registro che potrebbe essere coinvolto. Quando il numero di è invece basso, possono essere encodati insieme all'opcode.

Il designer dove far attenzione a problemi di conflitto dovuti alla dimensione del codice o alla dimensione dell'instruction set, il numero di metodi di indirizzamento e il numero di registro, oltre alla complessità di fetch e decoding hardware.

Ormai sono poche le persone che sviluppano direttamente in assembly, in quanto ormai i programmi odierni fanno utilizzo di compilatori largamente ottimizzati. Si pongono allora alcuni problemi relativi alla allocazione delle variabili all'interno dei registri, fase cruciale in fase di ottimizzazione da parte di un compilatore. E' possibile ottimizzare il tempo di accesso alle variabili allocandole all'interno dei registri solo se queste sono salvate nello stack o sono variabili globali in memoria. Non è pertanto possibile per le variabili nello heap, a causa di problemi di allineamento.

Le raccomandazioni sono di avere almeno 16 registri ed essere semplici e ortogonali.

3 MIPS64

Il MIPS prende il nome da ***M**icroprocessor without **I**nterlocked **P**ipeline **S**tages*, e fa parte della famiglia di processori RISC. Il primo processore è stato inventato nel 1985 a cui si sono susseguiti ulteriori versioni. Quello che si è scoperto è che dimunendo la complessità di ogni passaggio si rendeva più veloce il funzionamento, dunque rimuovendo il sistema di interlock.

Questo tipo di processori quando eseguono un operazione in memoria si limitano a fare "solamente questo". Hanno un simple load-store instruction set. Sono pensati per l'efficienza delle pipeline, in particolare con una lunghezza di istruzioni prefissata e pensate per applicazioni a basso consumo energetico (a differenza dei processori SISC). Il misc per tanto potrebbe risultare più compatto in quanto ogni istruzione fa più operazioni, ma a costo di una maggiore complessità.

I registri sono a 64 bit e il registro 0 è sempre `0` (non `R0`). Questo consente di utilizzare metodi di indirizzamento alternativi rispetto a quelli già visti.

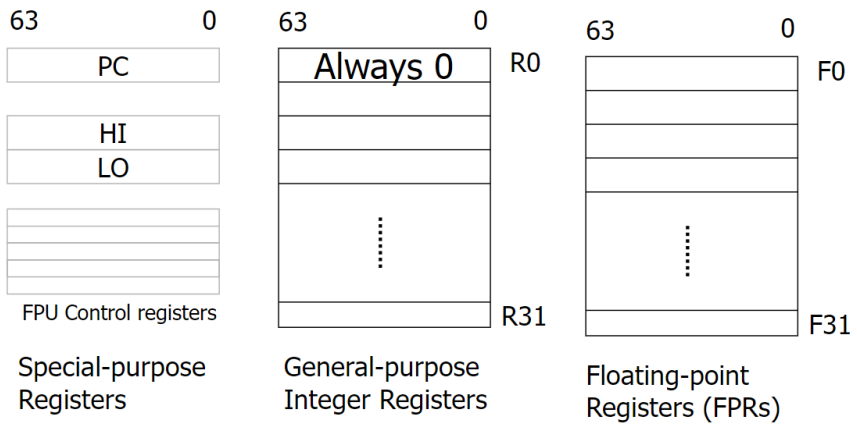


Figura 3.1: Architettura

I tipi di dato utilizzabili sono i classici:

- byte
- half word
- words

- double words
- 32-bit signle precisin floating point
- 64 bit double precision floating point

3.1 Metodi di indirizzamento

3.1.1 Indirizzamento immediato

Viene utilizzato 16 bit di immediate field. Il primo registro è quello destinazione, mentre il secondo e terzo campo sono gli operandi.

3.1.2 Indirizzamento con displacement

LD R1, 30(R2) // carica il valore di R2 + 30 in R1

R2 = XX

R1 <- MEM[30 + R2]

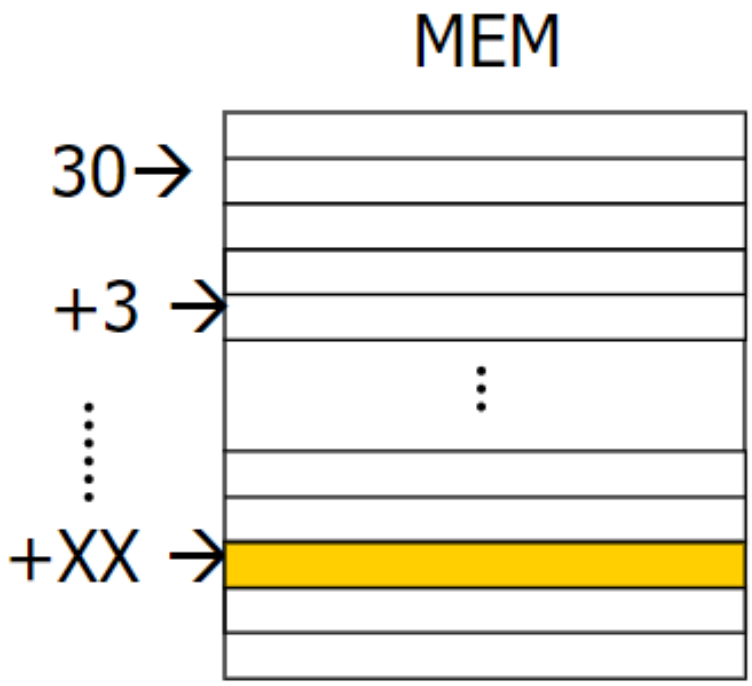


Figura 3.2: Indirizzamento

- registro indiretto
- indirizzamento assoluto

3.2 Formato delle istruzioni

Una istruzione CPU è un single 32 bit aligned word. Include un opcode di 6 bit iniziali. Le istruzioni sono in 3 formati:

- immediato
- registro
- salto (jump)

3.2.1 Immediato

Il primo tipo è quello immediato, caratterizzato da:

- **opcode**: 6 bit di opcode
- **Rs**: 5 bit di indirizzo sorgente *(source register)*
- **Rt**: 5 bit di indirizzo destinazione *(target register)*
- **Immediate**: 16 bit signed immediate utilizzati per gli operandi logici, aritmetici, signed operands, load/store address byte offsets, istruzioni di displacemnte rispetto a PC.

3.2.2 Registro

Il secondo tipo è quello registro, caratterizzato da:

- **opcode**: 6 bit di opcode
- **Rd**: 5 bit di indirizzo destinazione
- **Rs**: 5 bit di indirizzo sorgente
- **Rt**: 5 bit di indirizzo di indirizzo target
- **Sa**: 5 bit di shift amount, utile per fare shift a sinistra e a destra
- **Function**: 6 bit di funzione utilizzato per indicare le funzioni

3.2.3 Salto

Il terzo tipo è quello salto, caratterizzato da:

- **opcode**: 6 bit di opcode

- **offset**: Indice a 26 bit spostato a sinistra di due bit per fornire i 28 bit di ordine inferiore dell'indirizzo di destinazione del salto

3.3 Instruction Set

le istruzioni sono raggruppato per il loro funzionamento:

- Load and store
- operazioni ALU
- branches e salti
- floating point
- miscellanee

Nota: le istruzioni sono lunghe 32 bit.

3.3.1 Load and store

I processori MIPS utilizzano un architettura di caricamento e salvataggio, attraverso le quali avviene l'accesso alla memoria principale.

- **LB**: Carica un byte da memoria in un registro. `LB R1, 28(R8)`
- **LD**: Carica una doppia parola da memoria in un registro `LD R1, 28(R8)`
- **LBU**: Carica un byte senza segno da memoria in un registro `LBU R1, 28(R8)`
- **L.S**: Carica un floating point single precision in un registro `L.S F4, 46(R5).`
- **L.D**: Carica un floating point double precision in un registro `L.D F4, 46(R5)`
- **SD**: memorizza un double `SD R1, 28(R8)`
- **SW**: salvo una word `SW R1, 28(R8)`
- **SH**: salvo la half word più significativa `SH R1, 28(R8)`
- **SB**: salvo gli 8 bit meno significativi `SB R1, 28(R8)`
- stessa cosa con i reali

Ovviamente avviene l'estensione dei valori ripetendo il bit più significativo. Nel floating point il primo bit è il segno. Attenzione: per L.S abbiamo il risultato nella parte più significativa.

3.3.2 Operazioni ALU

Tutte le operazioni vengono eseguiti con operandi memorizzati nei registri. Le istruzioni possono essere di tipo immediato, con due operandi, shift, moltiplicazione, divisione, ecc. oltre ad aritmetica in complemento a due come somma, sottrazione, moltiplicazione, divisione.

3.3.2.1 ADD

- **DADDU**: dobule add unsigned `DADDU R1, R2, R3`
- **DADDUI**: double add unsigned immediate `DADDUI R1, R2, 74`
- **LUI**: load upper immediate `LUI R1, 0X47`

3.3.3 Branch e jump

- J Unconditional Jump `J name`
- JAL Jump and Link `JAL name`
- JALR Jump and Link Register `JALR R4`
- JR Jump Register
- BEQZ Branch Equal Zero
- BNE Branch Not Equal
- MOVZ Conditional Move if Zero
- NOP No Operation (nella realtà è uno shift a sinistra di 0 bit di R0 in R0)

4 Pipeline

La pipeline è un implementazione che consente di eseguire più istruzioni in modo sovrapposto durante l'esecuzione. In questo modo, differenti unità (chiamate pipe stages o segmenti) sono eseguite in parallelo ed eseguono parti differenti.

Il throughput rappresenta il numero di istruzioni che vengono processate per unità di tempo. Tutte gli stage sono sincronizzate e il tempo per eseguire il primo ste è chiamato machine cycle, e normalmente corrisponde a un ciclo di clock. La lunghezza del machine cycle è determinato dallo stage più lento. Siamo in grado di eseguire CPI (clock Cycles Per Instruction) clock cycles per istruzione.

In una pipeline ideale, tutti gli stage sarebbero perfettamente bilanciati e sarebbe dovuto a:

$$\text{throughput}_{pipelined} = \text{throughput}_{unpipelined} * n$$

con n pari al numero di stages.

4.1 Versione senza pipeline

Prendiamo come esempio una implementazione senza pipeline. L'esecuzione di ogni istruzione potrebbe essere composta di al più 5 clock cycles:

1. fetch (IF)
2. decode/register fetch (ID)
3. execution (EX)
4. Memory access (MEM)
5. Write back (WB)

Tutte le istruzioni richiedono dunque 5 clock cycle, tranne le istruzioni branch a cui ne bastano 4. Ottimizzazioni potrebbero essere fatte per ridurre il CPI medio: come esempio, le istruzioni alu potrebbero essere completate durante il cycle di MEM. Le risorse hardware potrebbero essere ottimizzato per eliminare duplicazioni. Si potrebbe prendere in considerazione un'architettura single clock alternativa. E' necessario l'utilizzo di un single control unit per produrre i segnali necessari al datapath.

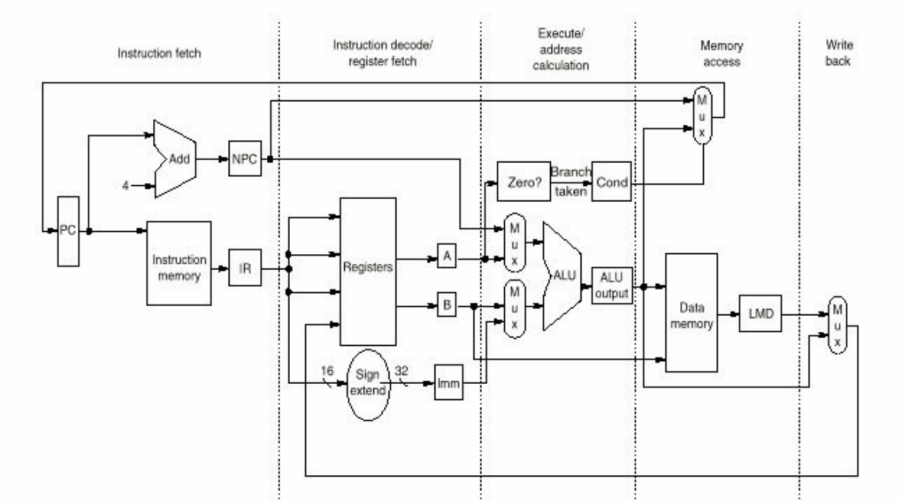


Figura 4.1: Datapath

4.2 Versione pipelined

Un esempio di una versione pipelined prevede l'avvio di una nuova istruzione per ogni clock cycle. Inoltre, differenti risorse lavorano mediante differenti istruzioni contemporaneamente. Per ciascun clock cycle, ciascuna risorsa può essere utilizzata per solo una richiesta; ciò significa che è necessario separare le istruzioni e la memoria dati e che il register file è utilizzato nello stadio di lettura in ID e per scrittura in WB. Deve dunque essere disegnato per soddisfare queste necessità nello stesso clock cycle. Il program counter deve essere cambiato nello stadio di IF (facendo attenzione nei casi dei salti). Infine, si vede necessario introdurre i **pipeline register**, ovvero dei registri intermedi.

Nota: si da per scontato che i dati necessari siano già stati caricati in cache.

4.3 Pipeline performance

La pipeline aumenta il throughput del processore senza dover rendere più veloce le singole istruzioni. Le istruzioni processato sono fatte rallentate da pipeline control overheads. La profondità della pipeline è limitata dalla necessità di bilanciare gli stati e dal overhead.

4.4 Pipeline hazards

Gli **hazards** sono situazioni che possono far si che un istruzione non venga eseguita come dovrebbe. Ci sono tre tipi di hazard:

- **structural hazards**: la causa sono relative a un conflitto tra le risorse
- **data hazards**: un istruzione dipende dall'esecuzione di una istruzione precedente
- **control hazards**: relative a salti condizionali e altre istruzioni che cambiano il program counter

4.4.1 Stalls

A causa dei pipeline hazards sono necessari gli stalli, dove si richiede ad alcuni processi di fermarsi per non causare problemi, per uno o più cicli di clock. Questo fa in modo che le istruzioni che verranno dopo una certa istruzione non vengano eseguite, mentre quelle indietro continuano ad essere eseguite. Gli stalli causano dunque l'introduzione di una sorta di "bolla" all'interno della pipeline.

4.4.2 Structural hazards

I structural hazards possono avvenire quando una unità della pipeline non è in grado si eseguire una certa operazione che era stata pianificata per quel cycle. Alcuni esempi potrebbero essere:

- una unità non è in grado di terminare un il suo task in un ciclo di clock
- La pipeline ha un solo register-file write port, ma non ci sono cicli in cui due register writers sono richiesti
- La pipeline fa riferimento a un single-port memory, e ci sono cicli in cui differenti istruzioni vorrebbero accedere alla memoria contemporaneamente.

La soluzione è inevitabilmente il miglioramento dell'hardware o l'acquisto di nuove componenti.

4.4.3 Data hazards

Gli hazard di dati sono quelli relativi alle dipendenze dei dati che vengono elaborati alterando, ad esempio, l'ordine di lettura e scrittura degli operandi e causando risultati sbagliati o non deterministici.

Se avviene un interruzione durante l'esecuzione di una porzione di codice critica la correttezza potrebbe essere ripristinata, ma causando quello che potrebbe essere un comportamento non deterministico.

Per risolvere questi problemi si potrebbe utilizzare:

- implementare uno stallo per i dati richiesti, utilizzandoli solo quando sono disponibili
- implementare un meccanismo di forwarding

4.4.3.1 Forwarding

Un hardware dedicato all'interno del datapath si occupa di rilevare quando un precedente operazione alu dovrebbe essere scritta nel registro che corrisponde al sorgente della operazione ALU. In questo caso, l'hardware selezione il risultato come input piuttosto che il valore nel registro. Deve, inoltre, essere in grado di fare il forwarding dei dati da ogni istruzione iniziata in precedenza e allo stesso modo di non fare forwarding se l'istruzione che segue è in stallo oppure è stata eseguita una interruzione.

Si hanno dunque data hazard quando vi è dipendenza tra le istruzioni e sono abbastanza vicine da essere sovrapposte a causa della pipeline. Questo generalmente avviene per operandi che sono sia registri che memoria, in particolare se la memoria subisce load e store non nello stesso stage o se l'esecuzione procede mentre un istruzione è in attesa di risoluzione di una cache miss.

Un esempio può essere il seguente:

```
; istruzioni     1 2 3 4 5 6 7 8
LD  R1, 0(R2)    F D E M W          -> 5
SUB R4, R1, R5     F D s E M W      -> 2
ADD R6, R1, Ry       F s D E M W    -> 1
```

4.4.3.2 Implementazione del controllo

Il controllo avviene nella fase di decodifica, e richiede di individuare un possibile data hazard relativa a una istruzione in fase di ID. Se venisse rilevata, due strade sono possibili:

- attivazione del forwarding
- l'istruzione viene posta in stallo prima di entrare nello stage in cui gli operandi non sono disponibili

4.4.3.3 Load interlock detection

Quando una istruzione di load va in fase di esecuzione e un'altra istruzione sta cercando di accedere al dato carico in fase di decode, dovranno essere eseguiti dei controlli per verificare se gli operandi fanno match, e nel caso rilevare il data hazard e come risultato l'unità di controllo deve inserire nella pipeline uno stallo per prevenire le istruzioni di fetch e decode di avanzare.

opcode di ID/EX	opcode di IF/ID	match?
Load	register-register alu	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	register-register alu	ID/EX.IR[rt] == IF/ID.IR[rt]

opcode di ID/EX	opcode di IF/ID	match?
Load	load, store, ALU immediate, branch	ID/EX.IR[rt] == IF/ID.IR[rs]

4.4.3.4 Introdurre stalli e forwarding

Data una istruzione attuamente in stage di decodifica, introdurre uno stallo in fase di esecuzione è possibile:

- forzando tutti zeri nella pipeline ID/EX register (corrisponde a una nop)
- forzando IF/ID register a contenere il valore corrente
- Fermare il program counter

Invece, l'introduzione di un forwarding può essere implementato:

- dal data memory output del ALU
- verso l'input della ALU, data memory inputs, o zero detection unit

Inoltre, deve necessariamente eseguire le comparazioni tra il destination fiel del IR contenuto nel EX/MEM e MEM/WB registers con il source fiel del IR contenuto nel IF/IDm ID/EX, EX/MEM registers.

4.4.4 Control hazards

Sono dovuti a salti (condizionali o meno) che possono cambiare il program couter dopo che l'istruzione ha già eseguito il fetch. Nel caso in cui siano condizionali, la decisione di come varia il program counter dipende da quale branch verrà eseguito. Nella implementazione MIPS, il PC è scritto con il target address (se è preso) alla fine della fase di decode.

Una possibile soluzione si basa sull'utilizzo di stalli appena l'istruzione di branch viene individuata (in fase di decode) e decidere anticipatamente se il salto avverrà o meno e calcolare in anticipo il nuovo valore del program counter.

Un esempio senza ottimizzazione:

```
1 ; istruzioni     1 2 3 4 5 6 7 8
2 nez  R1, cont    F D E M W
3 1                  F D
4 2                    F
5
6 :
7 1                      F D E M W
8 2
```

Con l'ottimizzazione:

```
; istruzioni     1 2 3 4 5 6 7 8
nez  R1, cont    F D E M W
1                  F
2
                                
:
1                    F D E M W
2
```

Il costo è l'aumento di hardware e fare attenzione ai registri che sono legate alle istruzioni di salto in modo che siano corretti.

Per evitare problemi può essere necessario introdurre stalli per avere consistenti i valori nei registri, riprendendo l'esempio di prima:

```
; istruzioni     1 2 3 4 5 6 7 8
addi R1, R1, 1   F D E M W
nez  R1, cont      F s D E M W
1                    s F
2
                                
:
1                      F D E M W
2
```

Ci sono varie tecniche per ridurre la degradazione delle performance dovute ai salti:

- freezing the pipeline
- predict untaken
- predict taken
- delayed branch

La prima alternativa, **freezing the pipeline**, è quella già proposta che prevede che la pipeline sia posta in stallo (o svuotata) appena l'istruzione di branch è rilevata e fino a quando non si conosce dove saltare. E' la soluzione più semplice da implementare.

La **predict untaken** assume che il branch non venga preso e evita qualsiasi cambio di stato della pipeline fino a quando il branch non ha compuuto la decisione. Inoltre, annulla le operazioni eseguite se invece il branch viene preso. Lo stesso approccio può essere utilizzato nel **predict taken** se si è a conoscenza del il target address prima del risultato del branch. In questi casi il compilatore utilizza delle ottimizzazioni interne per aumentare le probabilità da parte del processore di fare la giusta previsione (ad esempio realizzando strutture for più semplici da valutare).

Il **delayed branch** si basa sull'idea di riempire gli slot dopo l'istruzione di branch, denominati branch.delay slot, con istruzioni che devono essere eseguite indipendentemente dall'esito del branch.

E' compito del compilatore eseguire aggiungere le istruzioni corrette e non esegue nulla in particolare quando l'istruzione di branch è decodificata. A volte però non è semplice trovare le istruzioni di delayed slot, per cui negli ultimi anni è meno adottata.

4.5 Multicycle operations

Fino ad ora abbiamo lavorato con istruzioni che richiedono un colpo di clock per essere eseguite. In questo capitolo vedremo come gestire istruzioni che richiedono più cicli di clock per essere eseguite.

4.5.1 Floating Point operations

Le operazioni sui numeri floating point è più complicato rispetto agli interi. Per questo motivo, per riuscire a svolgere tali operazioni ini un solo ciclo di clock il designer sarebbe costretto:

- utilizzare un clock lento
- rendere complesse le unità (aggiungendo molta logica)

Nessuna di queste due è però realmente fattibile, per cui si cerca di scomporre l'operazione in più fasi che vengono eseguite in una sorta di pipeline. Otteniamo quindi una versione che paralelizza le diverse attività. Tutte le unità convergono nella fase di mem per poi concludere nella fase di write back. E' solo la fase di execute che risulta parallelizzata.

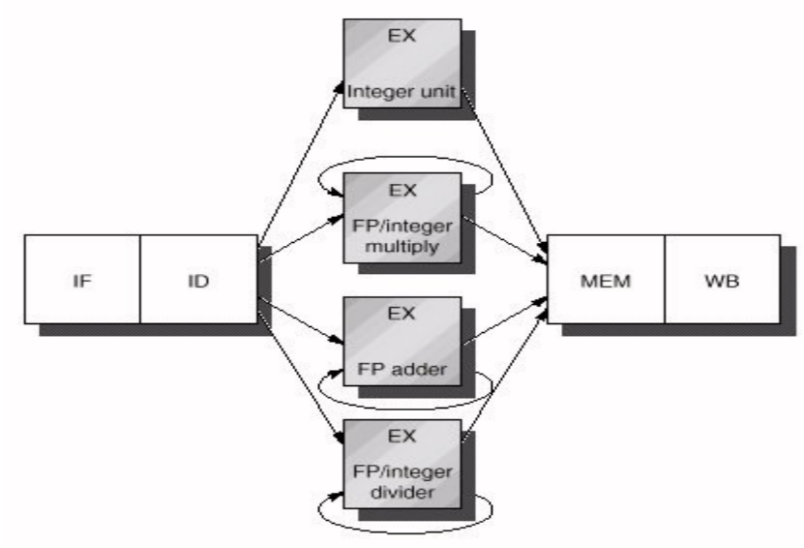


Figura 4.2: Parallelizzazione dell'esecuzione

Si vede necessario definire:

- **latenza**: il numero di cicli che devono essere eseguiti per concludere l'istruzione e produrre un risultato che possa essere riutilizzato
- **initiation interval** (intervallo di inizializzazione): il numero di cicli che devono passare tra due operazioni dello stesso tipo sulla stessa unità

4.5.2 Multi-cycle Hazards

Questo crea però come problema quello di una comparsa più frequente degli hazard.

4.5.2.1 Structural hazards (multi-cycle)

A causa della dell'impossibilità di utilizzare una pipeline per l'unità di divisione, molte istruzioni potrebbero necessitarne allo stesso tempo. Inoltre, è possibile ottenere risultati dalle diverse unità operazionali nello stesso tempo *(non è realmente possibile)*.

> Il simulatore funziona in modo leggermente diverso, per cui alcuni casi potrebbero comportarsi diversamente.

La soluzione è quello di utilizzare ulteriori write ports (però molto costosa) oppure forzare uno structural hazard:

- le istruzioni sono poste in stallo nella fase di decode
- le istruzioni sono messe in stallo prima della fase di MEM o WB

4.5.2.2 Data hazards (multi-cycle)

A causa di un più unga latenza nelle operazioni, gli stalli per i data hazards possono fermare una pipeline per un quantitativo di tempo maggiore.

Inoltre, nuovi tipi di data hazards sono possibili a causa dei tempi maggiori per raggiungere la write back.

La soluzione è quello di controllare, prima di entrare in fase di esecuzione, se l'istruzione andrà a scrivere su un registro che è già in fase di esecuzione.

> Non sempre ci si ferma in fase di decodifica, può succedere che si fermi anche in fase di execute.

> Si utilizza la S per gli stalli strutturali ed s per gli stalli dovuti a data hazards.

4.5.3 MIPS R4000 Pipeline

Il MIPS R4000 è un processore a 64 bit introdotto il 1991, con istruzioni simili al MIPS64. Utilizzava una pipeline a 8 stage per risolvere i problemi di cache dovuti agli accessi e una frequenza di clock più elevata. L'accesso alla memoria erano stati dunque divisi in più passi. Le pipeline più lunghe prendono spesso il nome di **superpipelines**.

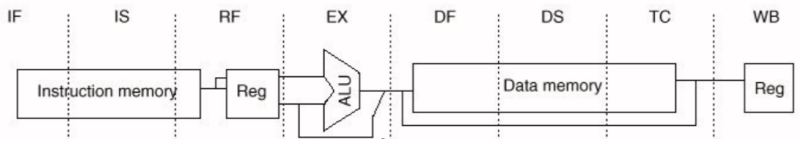


Figura 4.3: Mips R4000

Alcune caratteristiche sono:

- maggior forwarding
- load delay slot aumentato, di 2 cicli
- branch delay slod aumentato, di 3 cicli

L'unità floating poin è composta di 3 unità funzionali: divider, multiplier, adder. Era composta da 8 stage differenti:

- A adder Mantissa ADD stage
- D divider divide
- E multiplier exception test
- M multiplier multiplier I
- N multiplier multiplier II
- R adder rounding
- S adder operand shift
- U unpack numbers

4.6 Instruction level Parallelism

Esistono due approcci per l'Instruction level parallelism:

- Dynamic approach:
- Static approach:

Più frequenti soprattuto nel mercato desktop e server, ma anche inccluso in prodotti come:

- intel core seriees

- ..

L'approccio statico è meno frequente, ma prevalente in quello che è il mercato embedded.

4.6.1 Basic Block

Un basic block è una sequenza di istruzioni senza alcun branch-in, ad esclusione dell'ingresso, e senza branch-out, ad esclusione dell'uscita.

Il compilatore potrebbe rischedulare le istruzioni per ottimizzare il codice, ad esempio:

```
a = b + c
d = e - f
```

Assumendo che la load abbia una latenza di 1 clock cycle, un codice che implementa questo funzionamento sono:

```
LD  Rb, b
LD  Rc, c
ADD Ra, Rb, Rc
SD  Ra, Va
LD  Re, e
LD  Rf, f
SUB Rd, Re, Rf
SD  Rd, Vd
```

```
LD Rb, b    IF ID EX MEM WB
LD Rc, c       IF ID EX MEM WB
ADD Ra, Rb, Rc    IF ID st EX MEM WB
SD Ra, Va            IF st ID EX MEM WB
LD Re, e                IF ID EX MEM WB
LD Rf, f                   IF ID EX MEM WB
SUB Rd, Re, Rf                IF ID st EX MEM WB
SD Rd, Vd                        IF st ID EX MEM WB
```

Sono necessari 14 clock. Si potrebbe cambiare l'ordine delle operazioni ottenendo:

```
LD Rb, b    IF ID EX MEM WB
LD Rc, c       IF ID EX MEM WB
LD Re, e          IF ID EX MEM WB
ADD Ra, Rb, Rc       IF ID EX MEM WB
LD Rf, f                IF ID EX MEM WB
SD Ra, Va                  IF ID EX MEM WB
SUB Rd, Re, Rf                IF ID EX MEM WB
SD Rd, Vd                        IF ID EX MEM WB
```

In questo modo ottimizzato sono invece necessari solo 12 cicli di clock.

Per il MIPS le i basic block sono solitamente lunghi tra le 4 e le 7 istruzioni. Dal momento che le istruzioni potrebbero dipendere da altre, il parallelismo dei basic block è limitato, per tale motivo si utilizzano ulteriori tecnic di parallelizzazione come ad esempio quelle sui cicli.

4.6.2 Loop level parallelism

Prendendo come esempio il seguente codice:

```
for (i=0; i < 1000; i++)
    x[i] = x[i] + y[i];
```

Ogni iterazione del ciclo è indipendente dalle altre, quindi è possibile eseguire le istruzioni in parallelo. Esistono due modalità per fare ciò:

- loop unrolling (statico o dinamico)
- SIMD

4.6.2.1 Loop unrolling

Il loop unrolling è una tecnica che consiste nel duplicare il codice all'interno del ciclo, in modo da eseguire più istruzioni in parallelo.

```
// originale
for (i=0;i<N;i++ ) {
    body
}
```

```
// unrolled
for (i=0;i<N/4;i++ ){
    body
    body
    body
    body
}
```

Nell'esempio di prima otterremo:

```
or (i=0;i<N;i=i+4 ){
    x[i] = x[i]+ y[i];
    x[i+1] = x[i+1]+ y[i+1];
    x[i+2] = x[i+2]+ y[i+2];
    x[i+3] = x[i+3]+ y[i+3];
}
```

I vantaggi sono che riusciamo a ridurre il numero di controlli che vengono effettuate e aumentiamo le chance che il compilatore elimini gli stalli. Lo svantaggio è l'aumento della dimensione del codice.

4.6.2.2 SIMD

Le single instruction stream possono essere portate a multiple data streams (SIMD) adoperando vector processors, istruzioni vettoriali che lavorano su un set di dati (rispettoa d dati scalari), l'utilizzo di Graphics Processing Units e l'utilizzo di differenti unità funzionali per eseguire task simili in parallelo lavorando su multipli dati.

4.6.3 Dipendenze

Abbiamo 3 tipi di dipendenze di dato:

- dipendenze di dati
- dipendenze di nome
- dipendenze di controllo

4.6.3.1 Dipendenze di dati

Si dice che c'è una dipendenza di dati se una istruzione i dipende da una istruzione j con i che deve produrre un risultato che viene utilizzato da j oppure se l'istruzione j dipende da una istruzione k che è a sua volta dipendente da i.

Un esempio è mostrato di seguito:

```
1 Loop:   L.D F0, 0(R1)
2 ADD.D   F4, F0, F2
3 S.D     F4, 0(R1)
```

La seconda istruzione utilizza come sorgente il prodotto della prima, mentre la terza ugualmente riscrive nel medesimo registro causando delle dipendenze tra i dati.

4.6.3.1.1 Dipendenze e hazard Le dipendenze sono proprietà del programma, mentre gli hazards sono proprietà della pipeline. Gli stalli dipendono dal programma e dalla pipeline.

4.6.3.1.2 Dipendenze di memoria Rilevare le dipendenze che riguardano i registri è semplice, ma se sono prese in considerazione celle di memoria potrebbe essere molto più complicato in quanto l'accesso alla medesima cella potrebbe essere molto complicato. Se viene utilizzata una strategia

statica, il compile deve adottare un approccio conservativo assumendo che ogni istruzione di load faccia riferimento alla stessa cella o a una precedentemente salvata. Questo tipo di dipendenze possono essere rilevate solo a runtime.

4.6.3.2 Dipendenze di nome

Le dipendenze di nome avvengono quando due istruzioni fanno riferimento allo stesso registro o alla stessa locazione di memoria (nome) ma non c'è un flusso di dati associato al nome. Ci sono due tipi di dipendenze di nome tra una istruzione i e un istruzione j:

- **antidependence**: istruzione j scrive un registro o una locazione di memoria che l'struzione i deve leggere, ed i viene eseguita prima.
- **output dependence**: entrambe le istruzioni i e j scrivono nello stesso registro o porzione di memoria.

```
1 Loop: L.D F0, 0(R1)
2       ADD.D F4, F0, F2
3       S.D F4, 0(R1)
4       L.D F0, -8(R1)
5       ADD.D F4, F0, F2
6       S.D F4, -8(R1)
7       L.D F0, -16(R1)
8       ...
```

tra la riga 2 e la 4 c'è antidependence, mentre tra la 1 e la 4 c'è output dependence.

4.6.3.2.1 Register renaming Strategia secondo cui si cerca di identificare i casi in cui la dipendenza dei dati possa non esistere utilizzando un ulteriore registro.

- dinamico
- statico

```
1 DIV.D F0, F2, F4
2 ADD.D F6, F0, F8
3 S.D F6, 0(R1)
4 SUB.D F8, F10, F14
5 MUL.D F6, F10, F8
```

Potrebbe essere risolto nel seguente modo:

```
1 DIV.D   F0, F2, F4
2 ADD.D   F6, F0, F8
3 S.D     F6, 0(R1)
4 SUB.D   T, F10, F14
```

```
5 MUL.D   F6, F10, T
```

e inoltre:

```
1 DIV.D   F0, F2, F4
2 ADD.D   S, F0, F8
3 S.D     F6, 0(R1)
4 SUB.D   F8, F10, F14
5 MUL.D   S, F10, F8
```

4.6.3.2.2 RAW hazards Read after write hazards, corrispondono a una dipendenza di dati reale.

4.6.3.2.3 WAW hazards Sono possibili se le istruzioni scrivono in uno o più stae o

4.6.3.2.4 WAR hazards …

4.6.3.3 Dipendenza di controllo

Dipendenze che avvengono quando un istruzione dipende da un branch. Ad esempio:

```
1 if p1 {
2     S1;
3 }
4 j
5 if p2 {
6     S2;
7 }
```

S1 è dipendente da p1, ed S2 è dipendnente da p2.

4.6.3.3.1 Data flow Bisogna fare attenzione nel caso in cui avvengano modifiche che alterano il flusso di dati ed è fondamentale evitarlo.

4.7 Exceptions and Interrupts

Le eccezzioni sono eventi interni che modificano la normale esecuzione del programma. Nel caso invece di eccezzioni dovute a effetti esterni si parla di interruzioni.

Le cause di eccezzioni sono varie:

- I/O device request

- Operating system call by a user program
- Tracing instruction execution
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow or underflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory-protection violation
- Undefined instruction
- Hardware malfunction
- Power failure.

Le eccezioni si classificano in:

- sincrone e non sincrone
- Richieste dall'utente o coerced (forzate)
- mascherabili e non mascherabili
- in mezzo all'istruzione o tra due istruzioni
- con ritorno o senza ritorno

La maggior parte dei dispositivi sono catalogati come restartable machines, ovvero dato una eccezione è in grado di ripartire da dove si era bloccata.

Quando avviene una eccezione, la pipeline deve eseguire i seguenti step:

- forzare una trap instrcution nella pipeline nella prossima fase di fetch
- finchè non viene presa la trap, deve essere disabilitate tutte le scritture per le istruzioni che stanno causando la eccezione e per tutte le istruzioni nella pipeline.
- Quando la procedura di gestione dell'eccezione prende il controlo, salva immediatamente il Program Counter dell'istruzione che ha causato l'eccezione.

Una volta terminata la gesitone, una istruzione speciale fa tornare la macchina all'origine dell'eccezione e ricarica il PC originale facendo ripartire lo stream di istruzioni.

4.7.1 Interrupt Protocol in 80x86

1. Arriva un'eccezione da un dispositivo esterno
2. la CPU rileva l'interruzione
3. La CPU legge un numero dal databus di tipo N
4. La CPU salva il valore del processor status word (PSW) e dell'indirizzo di ritorno (Code Segnemnt and Instruction Pointer registers) nello stack

5. La CPU resetta l'interrupt flag disabilitando le interruzioni esterne e il trap mode
6. Utilizzando N come indice, il processore legge dalla Interrupt Vector Table l'indirizzo in cui saltare
7. La CPU salta alla Interrupt Service Routine

4.7.2 Interrupt Protocol in ARM

1. Arriva un'eccezione da un dispositivo esterno
2. La CPU rileva l'interruzione
3. La CPU esegue il push del current stack frame composto da 8 registri incluso Program Counter e il Processor Status Register e R0-R3 all'interno dello stack
4. La CPU aggiorna i flag del processore
5. La CPU salta all'indirizzo dato dall'interruzuone di tipo N, in tale posizione verrà messo il salto alla attuale Interrupt Service Routine

4.7.3 Precise exceptions

Un processore può gestire le eccezioni in modo preciso e in modo non preciso. Quando avviene una istruzione che rilascia eccezione tutte le precedenti devono essere completate, mentre quelle che seguono vengono rieseguite dall'inizio. Ripartire dopo una eccezione potrebbe essere molto complicato se non gestite in modo preciso, per questo è necessario nella maggior parte delle architetture (perlomeno per le istruzioni intere). Questo ha però un costo in termina di performance.

4.7.4 Imprecise exceptions

Garantire eccezioni precise è più complicato con le multiple cycle instructions. Un esempio è mostrato di seguito:

```
1 DIV.D F0, F2, F4
2 ADD.D F10, F10, F8
3 SUB.D F12, F12, F14
```

L'istruzione ADD.D e SUB.D sono completate prima di DIV.D (out-of-order completion). Se dovesse essere rilasciata una eccezione da parte di SUB.D, questa sarebbe gestita in modo impreciso.

La soluzione implementabili sono vari:

- utilizzare anche le eccezioni non precise
- fornire una versione operativa veloce e imprecisa ed eventualmente una versione precisa ma più lenta

- forzare l'unità FP a determinare se una istruzione causerà una eccezione, e nel caso procedere con le priossime istruzioni solo quando le precendenti sono completate senza problemi.
- Bufferizzare i risultati di ogni eccezione fino a quando una istruzione non è stata completata senza problemi.

4.7.5 MIPS: possibile sorgenti di eccezione

Pipeline stage	Cause of exception
IF	Page fault on instruction fetch, Misaligned memory access, Memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch, Misaligned memory access, Memory-protection violation
WB	None

4.7.6 Eccezioni contemporanee

Immaginiamo di avere una eccezione nella MEM di LD e nella EX di DADD:

```
1 LD   IF ID EX MEM WB
2 DADD IF ID EX MEM WB
```

Potrebbe avvenire una data page fault exception nella fase di MEM per LD e una eccezione di tipo aritmetico nello stage EX per DADD. La price eccezione viene processata e, se la causa dovesse essere rimossa, la seconda eccezione viene gestita.

Esistono però dei casi in cui le due eccezioni possono avvenire in ordine inverso a quelli a cui fanno riferimento:

```
1 LD IF ID EX MEM WB
2 DADD IF ID EX MEM WB
```

La soluzione potrebbe essere:

- aggiungere un flag di status per ogni istruzione della pipeline
- se avviene una eccezione, viene settato il flag di stato
- se il flag di stato è settato, l'istruzione non esegue operazioni di scrittura

- Quando una istruzione raggiunge l'ultimo stage, e il flag di stato è settato, viene rilasciata una eccezione

4.7.7 Instruction set complications

Quando una istruzione è garantito che finisca è detta **committed**. Alcune macchine hanno istruzioni che possono cambiare stato prima di essere *committed*. Se una di queste istruzioni venisse abortita a causa di una eccezione, lascerebbe la macchina in uno stato alterando e rendendo nuovamente di difficile implementazione le eccezioni precise.

Instructions implicitly updating condition codes possono creare complicazioni:

- possono causare data hazards
- richiedono di essere salvati e ripristinati in caso di eccezione
- rendono più difficoltoso per il compilatore riempire il possibili delay slot tra le istruzioni di scritture per i condition codes e i branch.

Le istruzioni complesse sono difficili da implementare nella pipeline, forzando ad avere la stessa lunghezza. Per questo motivo a volte i problemi sono risolti inserendo nella pipeline microistruzioni che implementano ciascuna istruzione.

5 Cache

Le prestazioni delle memorie cache sono molto più veloci rispetto alle memorie, sia hard disk che ram.

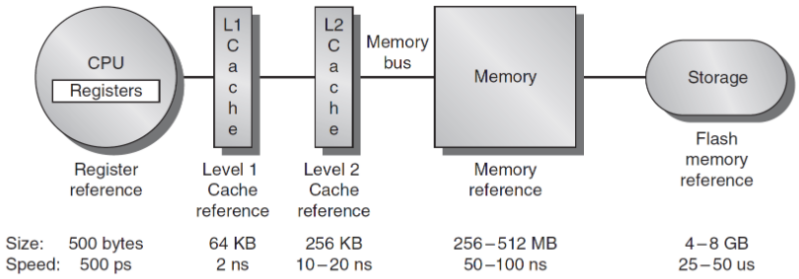


Figura 5.1: Dimensioni e velocità

Le cache funzionano secondo due principi:

- **principio di località temporale**: se a un tempo t il processore accede a una cella di memoria, è molto probabile che sarà necessario accedere nuovamente a quella cella in un tempo

$$t + \delta t$$

- **principio di località spaziale**: se a un tempo t il processore accede a una cella di memoria, è molto probabile che sarà necessario accedere nuovamente a celle di memoria vicine a quella cella.

Dunque se un blocco intero viene caricato in memoria cache a t0, è molto probabile che a un certo tempo

$$\delta t$$

il programma troverà in cache tutte le word necessarie.

E' necessario definire alcuni elementi:

- **h**: cache hit ratio
- **C**: cache access time

- **M**: memory access time quando il dato non è in cache

Il tempo di accesso alla memoria medio sarà:

$$t_{access} = h * C + (1 - h) * M$$

Normalmente i valori per h sono nell'ordine di 0.9. L'equazione non è però accurata in quanto prende in considerazione solo un tipo di cache.

5.1 Organizzazione

La cache si divide in una parte di data e una parte di controllo. La parte di data a sua volta e divisa in un directory array un data array., in cui ogni entry è una cache line caratterizzatà da un bit di validità, un tag e un blocco di dati.

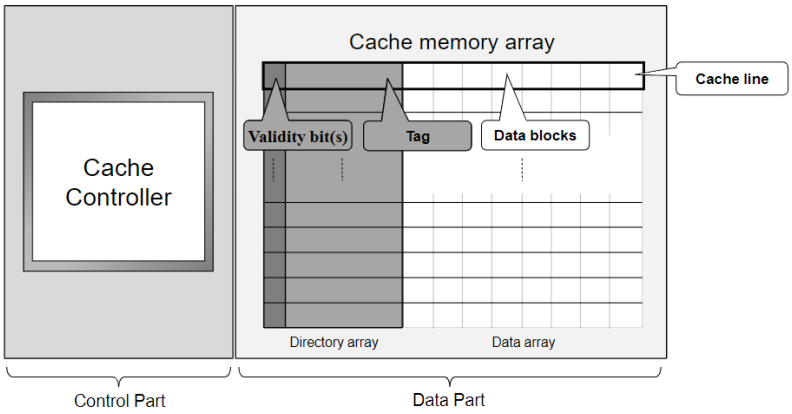


Figura 5.2: Organizzazione di una memoria cache

Ogni cache line può contenere un blocco di memoria con a sua volta più word. A ciascuna è associato un tag field che indica il blocco di memoria presente in quel momento. Inoltre, la cache contiene la logica ricevere gli indirizzi prodotti dal processore, controllare al suo interno per vedere se è presente e nel caso caricare il blocco.

Un data block può avere una dimensione differente e in numero differente all'interno di una cache line.

Il tag è una parte dell'indirizzo dove la linea di cache si trova in memoria, non è necessario salvare 32 bit dell'indirizzo.

I bit di validità invece indicano se il blocco è presente o meno. Il numero di bit di validità può avere più bit pari al numero di data block presenti.

Si parla di **cache hit** quando la richiesta di data, che viene ricevuta dalla cache, è presente all'interno della cache. Si parla di **cache miss** quando la richiesta di data non è presente all'interno della cache.

5.2 Trovare un blocco in cache

Ogni blocco di memoria e data block è pari a un byte, per cui se ho una memoria di 1024 byte avrò 1024 cache line, in quanto ogni cache ha un data block. Quando un dato della memoria deve essere indirizzato è composto da:

- **tag**: indica il blocco di memoria
- **index**: indica la cache line
- **offset**: indica la word all'interno del blocco

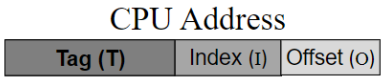


Figura 5.3: CPU address

Per individuare un blocco in cache è sufficiente un multiplexer (decoder) che opera sull'indirizzo:

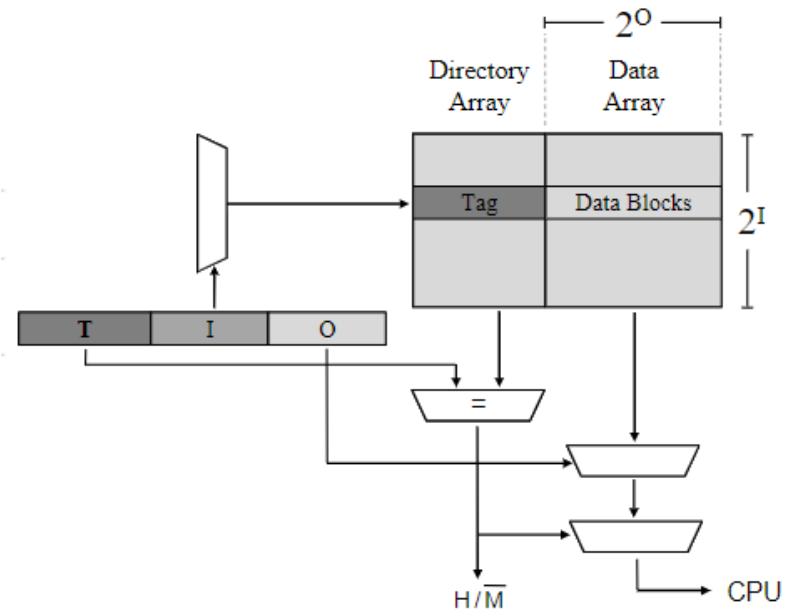


Figura 5.4: Logica combinatoria

Nel caso di 1024 byte saranno sufficienti 10 bit per l'index, 3 bit per l'offset e 19 bit per il tag.

Dunque il diagramma totale appare come segue:

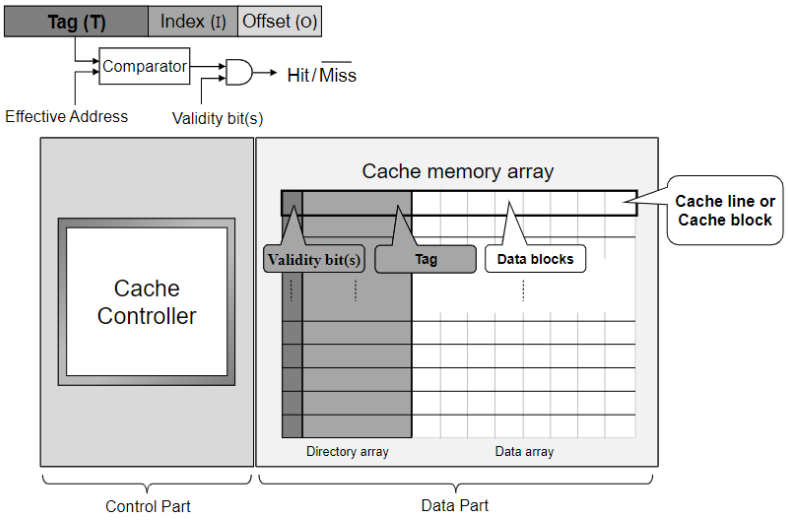


Figura 5.5: Recupero di un indirizzo

La cache è normalmente situata ta il CPU e il bus (oppure tra la memoria principale e il bus, ma non conviene). Ogni volta che il processore effettua un accesso in memoria allora la cache interpreta l'indirizzo e controlla se la word è già in cache o meno.

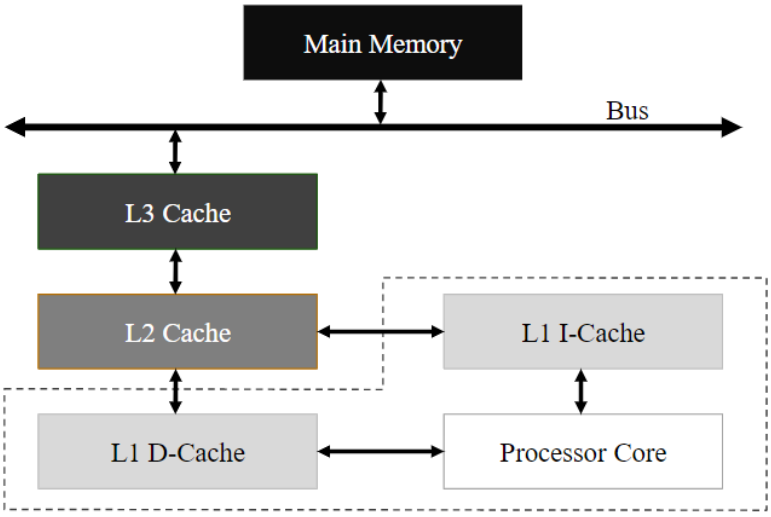


Figura 5.6: Cache position

Nel caso di un cache hit, la cache riduce il tempo di accesso alla memoria di un fattore dipendente dal ratio tra il tempo di accesso alla cache e alla memoria.

Nel caso di una miss, la cache può rispondere in due modi:

- Accede alla memoria carica il blocco mancante, successivamente risponde alla richiesta. Il tempo di accesso è maggiore rispetto a un sistema cache-free.
- Accede alla memoria e risponde immediatamente alla richiesta (load through o early restart). Questa tecnica richiede un costo maggiore in termini di hardware relativi alla cache, ma le miss hanno un impatto minore sulle performance della cache.

5.3 Harvard Architecture

Le cache oggi giorno sono separate in cache di istruzioni e cache di dati. La cache per le istruzioni è solitamente pià semplice da gestire rispetto a quella dei dati, in quanto le istruzioni non possono essere cambiate.

Se sono utilizzate due cache, l'architettura del sistema ricade nello schema denominato "Harvard architecture", caratterizzato dall'esistenza di due memorie separate tra dati e codice (in contrasto con quella di Von Neumann).

Le caratteristiche sono:

- cache size: dimensione della cache
- block size: dimensione di un blocco di memoria
- mapping: tipo di mappatura
- replacing algorithm: algoritmo di rimpiazzamento
- meccanismo di aggiornamento della memoria

5.3.1 Cache Size

La dimensione della cache è molto importante in termini di costi e performance. Man mano che la dimensione aumenta si ha un incremento dei costi, delle prestazioni di sistema ma una riduzione della velocità della cache. Le dimensioni solitamente variano da qualche kB a qualche MB.

5.3.2 Mapping

Il meccanismo attraverso cui una linea viene associata ad un blocco di memoria è detto mapping. E' importante assicurarsi che la verifica di presenza di un dato per un certo indirizzo sia sufficientemente veloce.

Il tipo di mappatura è detto modello di assocatività, e può essere:

- direct mapped: numero del blocco di memoria modulo totale dei blocchi in cache
- set associative: numero del blocco modulo totale dei blocchi in cache diviso livello di associatività. Permette di salvare ogni elemento della memoria in due linee della memoria (utile quando abbiamo elementi che vengono chiamati molto più spesso in modo da salvarli). Garantisce che i dati più utilizzati possono essere ritrovati in cache.
- Fully associative: ogni linea della memoria può essere salvata in qualsiasi posizione della cache. Salva l'indirizzo completo (tag scompare)

5.3.2.1 Direct Mapping

Ciascun blocco di memoria è associato staticamento a un set k in cache utilizzando l'espressione:

$$k = index \, mod \, n$$

dove n è il numero di linee nella cache. Il calcolo di k può essere fatto semplicemente prendendo i bit meno significativi dell'index.

Il vantagigo è che può essere implementato semplicemente in hardware, lo svantaggio è che se il programma accede frequentemente a due blocchi corrispondenti alla stessa cache line, avviene una miss a ciascun accesso in memoria.

5.3.2.2 Set associative Mapping

Per calcolare dove scrivere il valore è necessario trovare il numero di set:

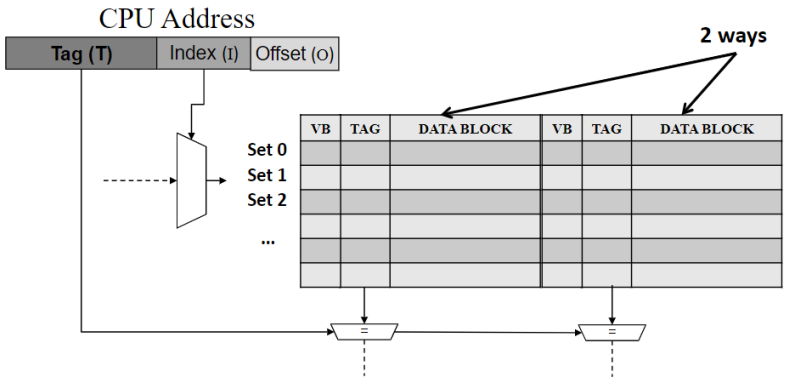
$$s = N/W$$

Dove N è il numero di cache lines e W il numero di word lines.

Un blocco è associato a un set k mediante:

$$k = i \, mod \, s$$

Il blocco `i` può essere inserito in una qualsiasi delle W linee del set k. Una cache set associative con W linee in ogni set è detta cache a W-linee. Solitamente W a un valore di 2 o 4.



5.3.2.3 Fully Associative Mapping

Ciascun blocco della memoria principale può essere messo in un blocco qualsiasi della cache. Il vantaggio è una maggiore flessibilità nello scegliere un blocco, ma a costo di una maggiore complessità hardware nella ricerca.

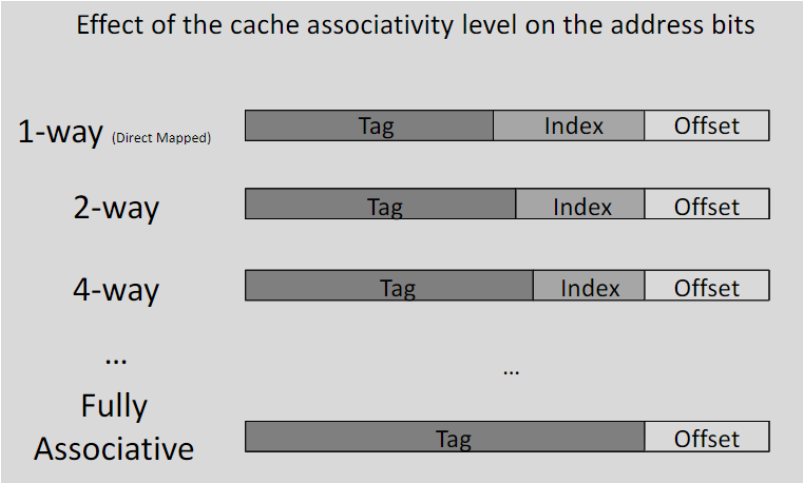


Figura 5.7: Fully Associative

5.4 Algoritmo di rimpiazzamento

Per sostituire le chace line deve essere utilizzato un algoritmo che individui quale rimuovere. Le scelte possibili sono:

- LRU: least recently used, la più utilizzata che scegli il rimpinzamento in base a quale sia stata la meno utilizzata recentemente.

- FIFO: first in first out, è la più semplice e scegli la prima che è stata utilizzata.
- LFU: least frequetly used, teoricamente la più efficace, sceglie quale rimpiazzare prendendo quella meno utilizzata.
- random: viene scelto casualmente quale cella utilizzare.

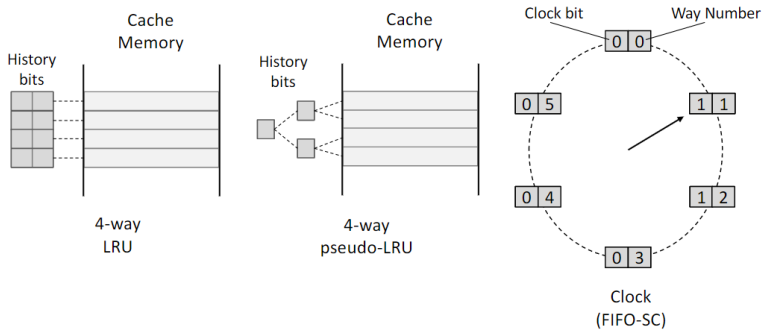


Figura 5.8: Algoritmi di rimpiazzamento

Esiste anche il pLRU che è un approssimazinoe efficiente di LRU. L'età di ciascuna via della cache è mantenuta in un albero binario, di cui ogni nodo rappresenta una "history bit". Quando avviene un accesso, viene fatto il toggle dei bit corrispondenti incontrati.

Nel caso FIFO viene utilizzato l'algoritmo **second chance**: ogni elemento ha un bit di utilizzo. Quando viene utilizzato un nodo viene posto a `1` il bit, dandogli una seconda "chance", perchè essendo appena stato letto potrebbe essere ancora utile. In modo sequenziale vengono controllati tutti i nodi, fino a quando non viene trovato uno di valore `0`, che viene rimosso. Se un nodo viene trovato con il bit a `1` viene posto a `0` e si continua a cercare. Se tutti i nodi sono stati utilizzati ha un comportamento FIFO.

5.5 Memory Update

Quando avviene una operazione di scrittura su un dato presente in cache, è necessario aggiornare anche il dato presente in memoria principale. Per fare ciò esistono due soluzioni:

- write back
- write through

5.5.1 Write Back

Per ogni cache block, un flag denominato **dirty bit**, indica se il blocco è stato cambiato o meno da quando è stato caricato in cache. La scrittura in memoria principale avviene solo quando il blocco

viene sostituito dalla cache ed è settato il dirty bit.

gli svantaggi di questo approccio sono:

- sostituzione più lenta in quanto a volte è necessario copiare in memoria principale il blocco.
- In un sistema multiprocessore ci potrebbero essere incosistenze tra le cache dei vari processori
- Potrebbe non essere possibile ripristinare il dato in memoria dopo un system failure.

5.5.2 Write Through

Ogni volta che la CPU effettua una operazione di scrittura, questo viene scritto sia in cache che in memoria principale. La conseguente perdita di efficienza è limitata dal fatto che le operazioni di scrittura sono solitamente molto meno numerose di quelle di lettura.

5.6 Cache Coherence

La coerenza tra le cache è uno dei problemi priincipali tra i sistemi multiprocessore con memoria condivisa, in cui ogni processore ha una propria cache. Lo stesso tipo di problema si verifica se è presente un DMA controller.

Per risolvere questo problema viene introdotto il **validity bit** per ogni cache line. Se è disabilitato, allora non è stato effettuato nessun accesso al blocco e deve dare una miss. All'avvio tutti i validity bit sono disabilitati.

Può essere conveniente utilizzare più bit di cache avere più livelli di cache:

- L1: primo livello, piccolo e veloce
- L2: secondo livello, lento ma capiente
- L3: terzo livello, molto lento ma molto capiente

Un esempio è AMD Sambezi, facente parte della AMD fusion family. Include 8 core con ciascuno una cache di livello 1. Ciascuna coppia di processori ha un secondo livello da 2 o 4 Mbytes e infine i core dello stesso device condividono un terzo livello di cache da 8 mbytes.

5.7 Esempio

Immaginiamo di avere una memoria cache con le seguenti caratteristiche:

- 64 kbyte di dimensione

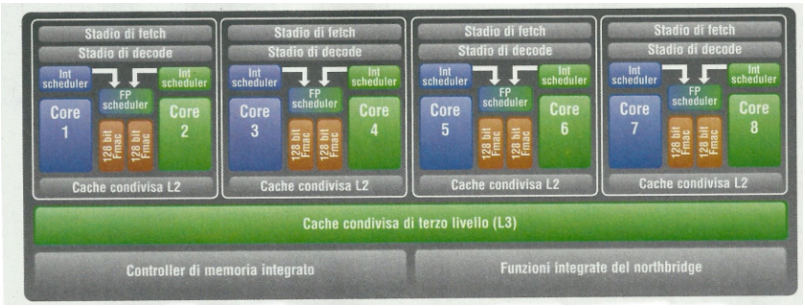


Figura 5.9: AMD Zambesi

- direct mapping
- 4byte blocks
- 32 bit di indirizzo

Determina la struttura della cache (numero di line, dimensione del tag field).

Ogni blocco è di 4 byte, quindi se ho 2^32 byte avrò 2^30 blocchi. Se la cache ha 64 kbyte, avrò 2^16 blocchi quindi la cache ha 2^14 linee. Il tag è composto da 30 bit, ma 14 fanno riferimento alla linea e dunque solo 16 sono per il tag field.

La dimensione totale della cache è dunque:

$$2^14 * (32 + 16) = 2^14 * 48 = 768kbit = 96kByte$$