



Politecnico  
di Torino

# Architetture e Sistemi di elaborazione

Anno accademico 2022-2023

Marco Lampis

6 febbraio 2023

# Indice

<b>0 Informazioni</b>	<b>1</b>
<b>1 Introduzione</b>	<b>3</b>
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 . . . . .	6
<b>2 Instruction Set</b>	<b>7</b>
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
<b>3 MIPS64</b>	<b>11</b>
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
<b>4 Pipeline</b>	<b>17</b>
4.1 Versione senza pipeline . . . . .	17

4.2	Versione pipelined . . . . .	18
4.2.1	Pipeline performance . . . . .	18
4.2.2	Pipeline hazards . . . . .	18
4.2.3	Control hazards . . . . .	21
4.3	Multicycle operations . . . . .	23
4.3.1	Floating Point operations . . . . .	23
4.3.2	Multi-cycle Hazards . . . . .	24
4.3.3	MIPS R4000 Pipeline . . . . .	25
4.4	Instruction level Parallelism . . . . .	25
4.4.1	Basic Block . . . . .	26
4.4.2	Loop level parallelism . . . . .	27
4.4.3	Dipendenze . . . . .	28
4.5	Exceptions and Interrupts . . . . .	30
4.5.1	Interrupt Protocol in 80x86 . . . . .	31
4.5.2	Interrupt Protocol in ARM . . . . .	32
4.5.3	Precise exceptions . . . . .	32
4.5.4	Imprecise exceptions . . . . .	32
4.5.5	MIPS: possibile sorgenti di eccezione . . . . .	33
4.5.6	Eccezioni contemporanee . . . . .	33
4.5.7	Instruction set complications . . . . .	34
<b>5</b>	<b>Cache</b> . . . . .	<b>35</b>
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 . . . . .	43
5.5.2	Write Through . . . . .	43
5.6	Cache Coherence . . . . .	43
5.7	Esempio . . . . .	44
<b>6</b>	<b>Branch Prediction Techniques</b> . . . . .	<b>45</b>
6.1	Static branch prediction . . . . .	45
6.2	Dynamic branch prediction . . . . .	46
6.2.1	Branch history table . . . . .	46

6.3	Two bit prediction schemes . . . . .	47
6.3.1	n-bit prediction schemes . . . . .	47
6.4	Impatto sulle performance . . . . .	48
6.5	Correlating Predictors . . . . .	48
6.5.1	(m, n) predictors . . . . .	48
6.5.2	(1,1) predictor . . . . .	49
6.5.3	(2,2) predictor . . . . .	49
6.6	Branch-target buffer . . . . .	50
6.7	gselect . . . . .	50
6.8	gshare . . . . .	52
<b>7</b>	<b>Dynamic scheduling Techniques</b>	<b>55</b>
7.1	Esecuzione fuori ordine . . . . .	55
7.2	Splitting ID-Stage . . . . .	55
7.3	EX stage . . . . .	56
7.4	Hardware Schemes for dynamic scheduling . . . . .	56
7.4.1	Tomasulo's algorithm . . . . .	56
<b>8</b>	<b>Speculation</b>	<b>61</b>
8.1	architettura . . . . .	61
8.2	Instruction Execution Steps . . . . .	62
8.2.1	Issue . . . . .	62
8.2.2	Execution . . . . .	63
8.2.3	Write result . . . . .	63
8.2.4	Commit . . . . .	63
8.3	WAW e WAR hazard . . . . .	63
8.3.1	RAW hazard . . . . .	64
8.4	istruzione di Store . . . . .	64
8.5	Exception Handling . . . . .	64
8.6	Speculative expensive events . . . . .	64
<b>9</b>	<b>Vliw</b>	<b>65</b>
9.1	Limitazioni . . . . .	65
9.2	EPIC . . . . .	66
9.3	Classificazione . . . . .	66
<b>10</b>	<b>Arm</b>	<b>67</b>
10.1	Data Processing . . . . .	68
10.1.1	Istruzioni reg-reg . . . . .	68

10.1.2 Istruzioni reg-imm . . . . .	69
10.2 Data Transfer Instructions . . . . .	70
10.3 Branch instructions . . . . .	70
10.4 branch and link instructions . . . . .	70
10.5 ARM Cortex M3 . . . . .	70
10.6 Programmer View . . . . .	72
10.7 Bus . . . . .	73
10.8 Memory map . . . . .	73
10.9 Exception . . . . .	75
10.10 Interrupt . . . . .	75
10.11 Clock distribution . . . . .	76
10.12 Power management . . . . .	76
10.13 Istruzioni . . . . .	76
10.13.1 Link register (R14) . . . . .	77
10.13.2 Stack pointer (R13) . . . . .	77
10.13.3 Program Status register (R12) . . . . .	77
10.13.4 Esecuzione condizionale . . . . .	77
10.14 Operazioni aritmetiche . . . . .	79
10.14.1 Moltiplicazione . . . . .	79
10.15 Moltiplicazione con accumulazione . . . . .	79
10.16 Divisione . . . . .	80
10.16.1 Divisione senza segno (unsigned) . . . . .	80
10.16.2 Divisione con segno (signed) . . . . .	80
10.17 Shift . . . . .	80
10.17.1 Logical Shift Left (LSL) . . . . .	80
10.17.2 Logical Shift Right (LSR) . . . . .	80
10.17.3 Arithmetic Shift Right (ASR) . . . . .	80
10.18 Rotazioni . . . . .	81
10.18.1 Rotate Right (ROR) . . . . .	81
10.18.2 Rotate Right with Extend (RRX) . . . . .	81
<b>11 Stack e subroutines</b> . . . . .	<b>83</b>
11.1 LDM e STM . . . . .	83
11.1.1 Ordine . . . . .	85
11.1.2 Metodi di indirizzamento . . . . .	85
11.1.3 PUSH e POP . . . . .	86
11.2 Subroutines . . . . .	86
11.2.1 Chiamate annidate a subroutines . . . . .	87

11.2.2 Passaggio di parametri . . . . .	87
11.3 ABI . . . . .	88
<b>12 Supervisor Calls (SVC)</b>	<b>91</b>
12.1 Stato delle eccezioni . . . . .	92
12.2 Supervisor Calls Syntax . . . . .	93
12.3 Exception return . . . . .	94
12.4 Processor Specific Configuration . . . . .	94
12.5 Processor Modes . . . . .	95
12.6 Control Register . . . . .	95
<b>13 Interrupt controller</b>	<b>97</b>
13.1 Polling . . . . .	97
13.2 Interrupt . . . . .	98
13.3 Controller . . . . .	98
13.3.1 NVIC . . . . .	98
<b>14 Timer</b>	<b>101</b>
14.1 Funzionamento . . . . .	101
14.2 LPC1768 . . . . .	102
14.2.1 Standard timers . . . . .	102
14.3 Utilità . . . . .	104
14.3.1 Accensione . . . . .	104
14.3.2 Abilitazione . . . . .	104
<b>15 Clocking and Power control functions</b>	<b>105</b>
15.1 Power control registers . . . . .	105
15.1.1 Sleep mode . . . . .	105
15.1.2 Deep sleep mode . . . . .	106
15.1.3 Power-down mode and Deep Power-down . . . . .	106
15.2 Peripheral power control . . . . .	106
<b>16 Switch bouncing</b>	<b>107</b>
16.1 Soluzione software . . . . .	107
16.2 Repetitive Interrupt Timer (RIT) . . . . .	109
<b>17 Touch Display</b>	<b>111</b>
17.1 Display controller - ILI9325 . . . . .	112
17.2 Touch Screen Controller - ADS7843 . . . . .	112



# 0 Informazioni

La seguente dispensa è stata realizzata nell'anno accademico 2022-2023 durante il corso di Architettura 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 o a eseguire una pull request su GitHub.

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 disponibili su GitHub al seguente link:

1 [https://github.com/Guray00/polito\\_lectures](https://github.com/Guray00/polito_lectures)



**Figura 1:** Repository GitHub

*La seguente dispensa è rilasciata sotto la Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International Public License.*





# 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 speedup 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 performance 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 opzione consiste nel calcolo della seguente formula:

$$\text{CPU time} = \left( \sum_{i=1}^n CPI_i * IC_i \right) * \text{clock cycle time}$$

- **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
- consumo 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 indirect**: ADD R1, @(R3)
- **autoincrement**: ADD R1, (R2)+
- **autodecrement**: ADD R1, -(R2)
- **scaled**: ADD R1, 100(R2)[R3]

Scegliere una metodologia 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 deve 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. È 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.

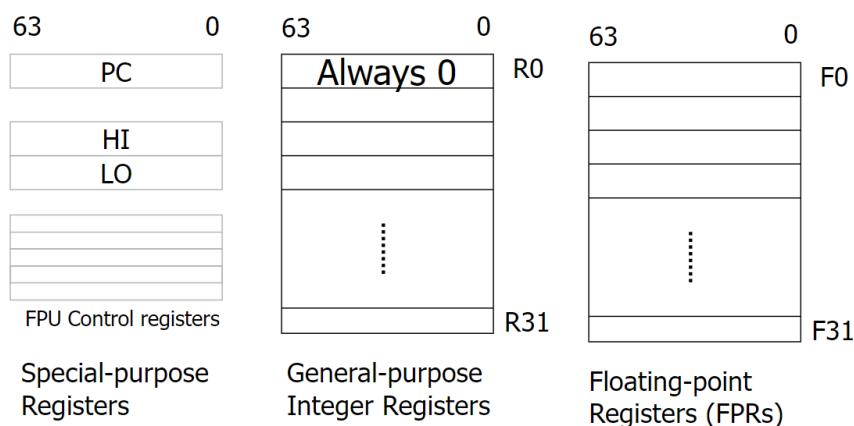
Le raccomandazioni sono di avere almeno 16 registri ed essere semplici e ortogonali.

## 3 MIPS64

Il MIPS prende il nome da ***Microprocessor without Interlocked Pipeline Stages***, 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 diminuendo 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 prefissa 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 single precision floating point
- 64 bit double precision floating point

### 3.1 Metodi di indirizzamento

#### 3.1.1 Indirizzamento immediato

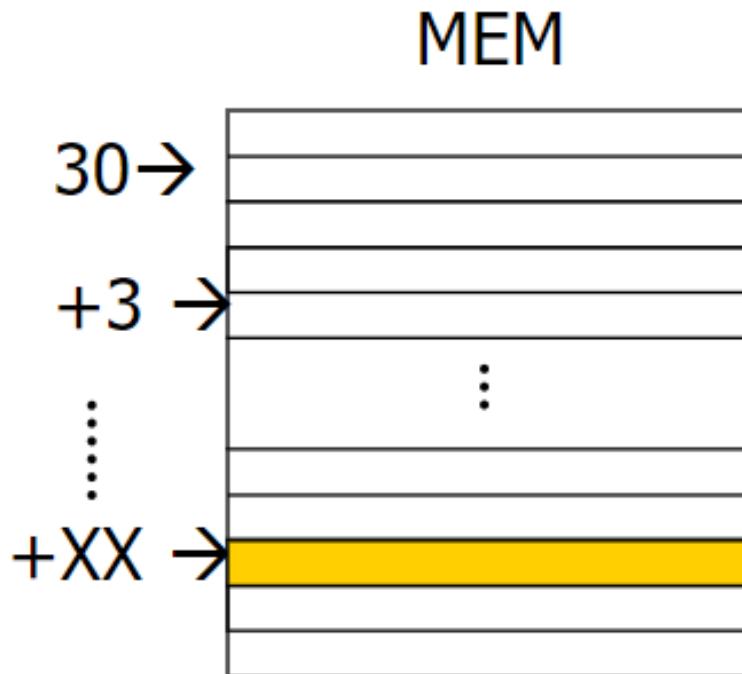
Viene utilizzato 16 bit di immediate field. Il primo registro è quello destinazione, mentre il secondo e terzo campo sono gli operandi.

#### 3.1.2 Indirizzamento con displacement

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

$R2 = XX$

$R1 \leftarrow MEM[30 + R2]$



**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 displacemnto 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**: double 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:

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

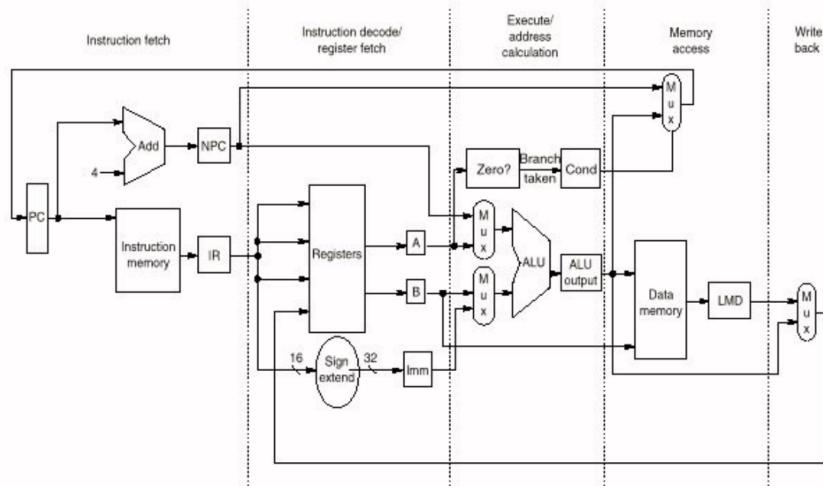
con  $n$  pari al numero di *stage*.

## 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 quelle di 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.

### 4.2.1 Pipeline performance

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

### 4.2.2 Pipeline hazards

Gli **hazards** sono situazioni che possono far sì 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

#### 4.2.2.1 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.

#### 4.2.2.2 Structural hazards

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

- una unità non è in grado di terminare 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.

#### 4.2.2.3 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 stall per i dati richiesti, utilizzandoli solo quando sono disponibili
- implementare un meccanismo di forwarding

#### 4.2.2.4 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 seleziona 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:

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

#### 4.2.2.5 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

#### 4.2.2.6 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]

#### 4.2.2.7 Introdurre stalli e forwarding

Data una istruzione attualmente 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 field del IR contenuto nel EX/MEM e MEM/WB registers con il source field del IR contenuto nel IF/IDm ID/EX, EX/MEM registers.

#### 4.2.3 Control hazards

Sono dovuti a salti (condizionali o meno) che possono cambiare il *program counter* 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:

```

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

```

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

## 4.3 Multicycle operations

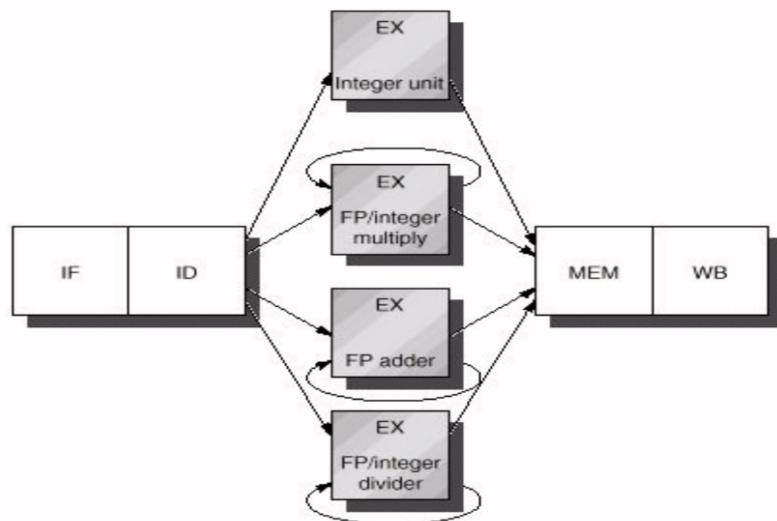
Fino ad ora abbiamo lavorato con istruzioni che richiedono un colpo di clock per essere eseguite. In questo capitolo vedremo come gestire istruzioni che richiedono più cicli di clock per essere eseguite.

### 4.3.1 Floating Point operations

Le operazioni sui numeri floating point è più complicato rispetto agli interi. Per questo motivo, per riuscire a svolgere tali operazioni in 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 parallelizza 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

### 4.3.2 Multi-cycle Hazards

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

#### 4.3.2.1 Structural hazards (multi-cycle)

A causa della 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 stall nella fase di decode
- le istruzioni sono messe in stall prima della fase di MEM o WB

#### 4.3.2.2 Data hazards (multi-cycle)

A causa di un più lunga 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.

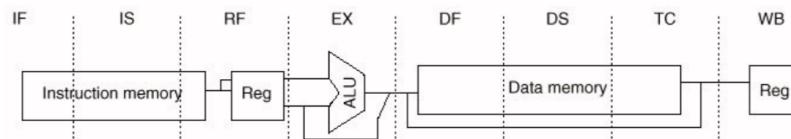
La soluzione è quello di controllare, prima di entrare in fase di esecuzione, se l'istruzione andrà a scrivere su un registro che è già in fase di esecuzione.

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.

### 4.3.3 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 slot aumentato, di 3 cicli

L'unità floating point 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

## 4.4 Instruction level Parallelism

Esistono due approcci per l'Instruction level parallelism:

- Dynamic approach:

- Static approach:

Più frequenti soprattutto nel mercato desktop e server, ma anche incluso in prodotti come:

- intel core serieses
- ..

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

#### 4.4.1 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:

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

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

```
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
```

```
1 LD Rb, b      IF ID EX MEM WB
2 LD Rc, c      IF ID EX MEM WB
3 ADD Ra, Rb, Rc   IF ID st EX MEM WB
4 SD Ra, Va      IF st ID EX MEM WB
5 LD Re, e        IF ID EX MEM WB
6 LD Rf, f        IF ID EX MEM WB
7 SUB Rd, Re, Rf    IF ID st EX MEM WB
8 SD Rd, Vd      IF st ID EX MEM WB
```

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

```
1 LD Rb, b      IF ID EX MEM WB
2 LD Rc, c      IF ID EX MEM WB
3 LD Re, e        IF ID EX MEM WB
4 ADD Ra, Rb, Rc   IF ID EX MEM WB
5 LD Rf, f        IF ID EX MEM WB
6 SD Ra, Va      IF ID EX MEM WB
```

7 SUB Rd, Re, Rf	IF ID EX MEM WB
8 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 tecniche di parallelizzazione come ad esempio quelle sui cicli.

#### 4.4.2 Loop level parallelism

Prendendo come esempio il seguente codice:

```
1 for (i=0; i < 1000; i++)
2     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

##### 4.4.2.1 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.

```
1 // originale
2 for (i=0;i<N;i++ ) {
3     body
4 }
```

```
1 // unrolled
2 for (i=0;i<N/4;i++ ){
3     body
4     body
5     body
6     body
7 }
```

Nell'esempio di prima otterremo:

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

```

4      x[i+2] = x[i+2]+ y[i+2];
5      x[i+3] = x[i+3]+ y[i+3];
6  }

```

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.

#### 4.4.2.2 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 (rispetto a 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.

#### 4.4.3 Dipendenze

Abbiamo 3 tipi di dipendenze di dato:

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

##### 4.4.3.1 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, @R1
2 ADD.D F4, F0, F2
3 S.D F4, @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.

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

**4.4.3.1.2 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 compilatore 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.

#### 4.4.3.2 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'istruzione 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.

**4.4.3.2.1 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

```

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

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

**4.4.3.2.4 WAR hazards** ...

### 4.4.3.3 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 dipendente da p2.

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

## 4.5 Exceptions and Interrupts

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

Le cause di eccezioni 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 instruction 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 controllo, salva immediatamente il Program Counter dell'istruzione che ha causato l'eccezione.

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

#### 4.5.1 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

#### **4.5.2 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'interruzione di tipo N, in tale posizione verrà messo il salto alla attuale Interrupt Service Routine

#### **4.5.3 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 termini di performance.

#### **4.5.4 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 precedenti sono completate senza problemi.
- Bufferizzare i risultati di ogni eccezione fino a quando una istruzione non è stata completata senza problemi.

#### 4.5.5 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

#### 4.5.6 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 prima 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

#### 4.5.7 Instruction set complications

Quando una istruzione è garantito che finisce 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.

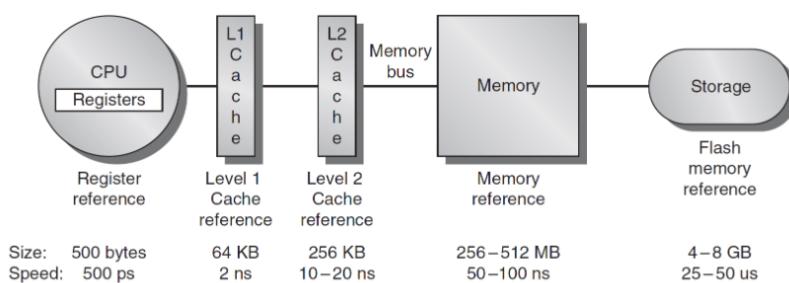
Instructions implicitly updating condition codes possono creare complicazioni:

- 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  $t_0$ , 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

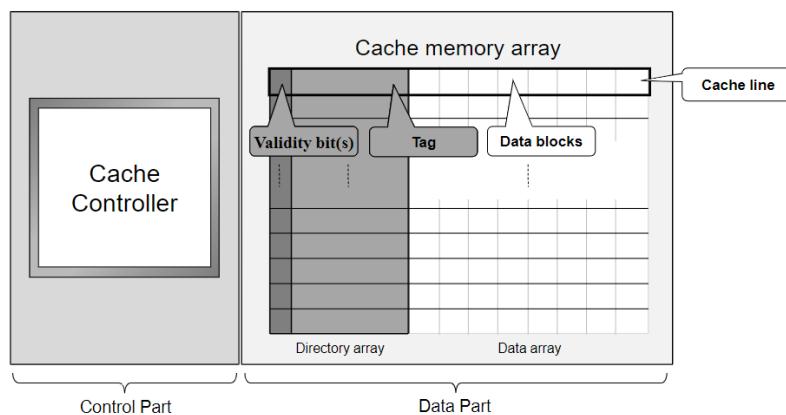
Il tempo di accesso alla memoria medio sarà:

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

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 è divisa in un directory array e un data array, in cui ogni entry è una cache line caratterizzata 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

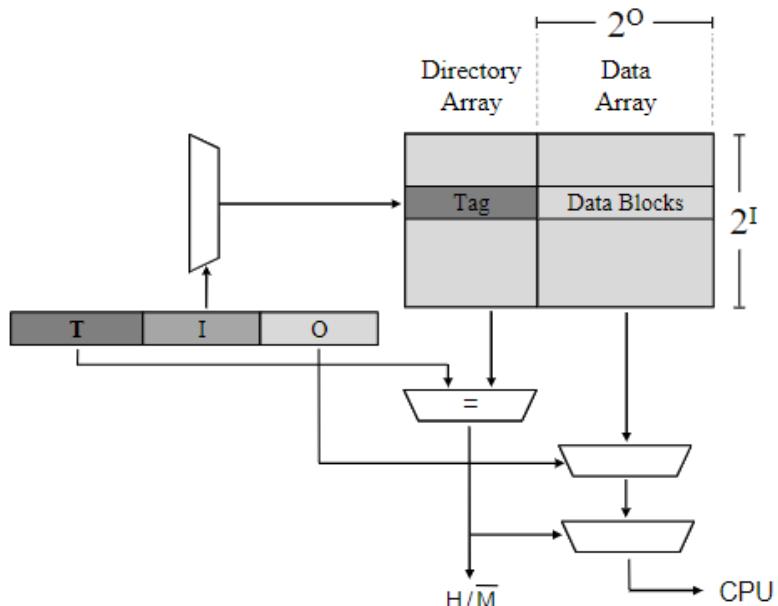
Ogni blocco di memoria e data block è pari a un byte, per cui se ho una memoria di 1024 byte avrò 1024 cache line, in quanto ogni cache ha un data block. Quando un dato della memoria deve essere indirizzato è composto da:

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



**Figura 5.3:** CPU address

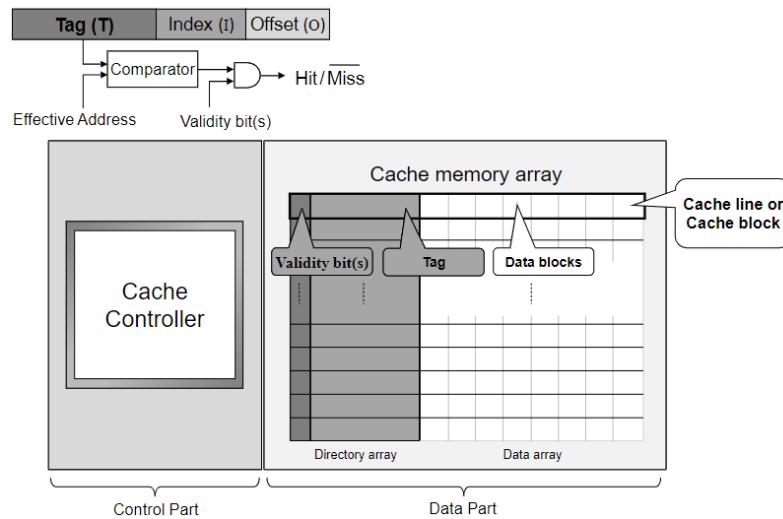
Per individuare un blocco in cache è sufficiente un multiplexer (decoder) che opera sull'indirizzo:



**Figura 5.4:** Logica combinatoria

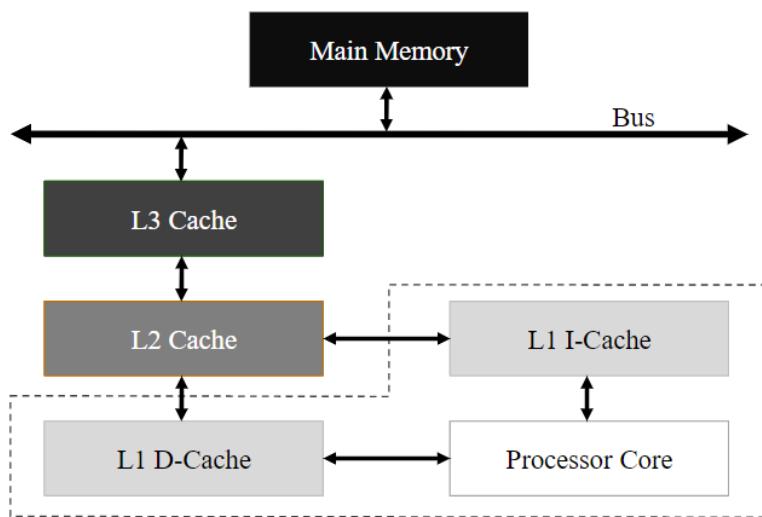
Nel caso di 1024 byte saranno sufficienti 10 bit per l'index, 3 bit per l'offset e 19 bit per il tag.

Dunque il diagramma totale appare come segue:



**Figura 5.5:** Recupero di un indirizzo

La cache è normalmente situata tra 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 più 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 associazività. 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 staticamente a un set  $k$  in cache utilizzando l'espressione:

$$k = \text{index} \bmod n$$

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 vantaggio 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:

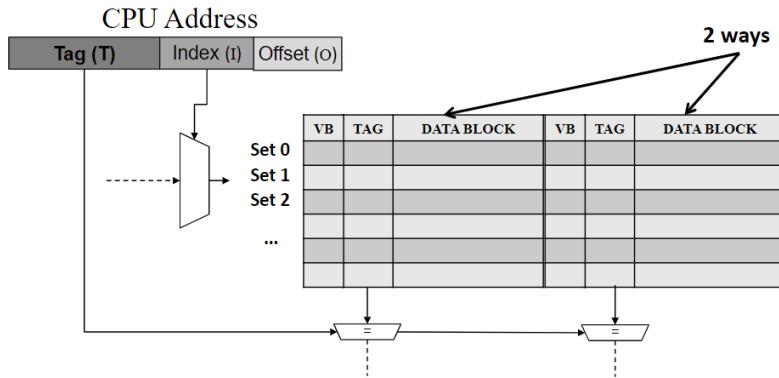
$$s = N/W$$

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

Un blocco è associato a un set  $k$  mediante:

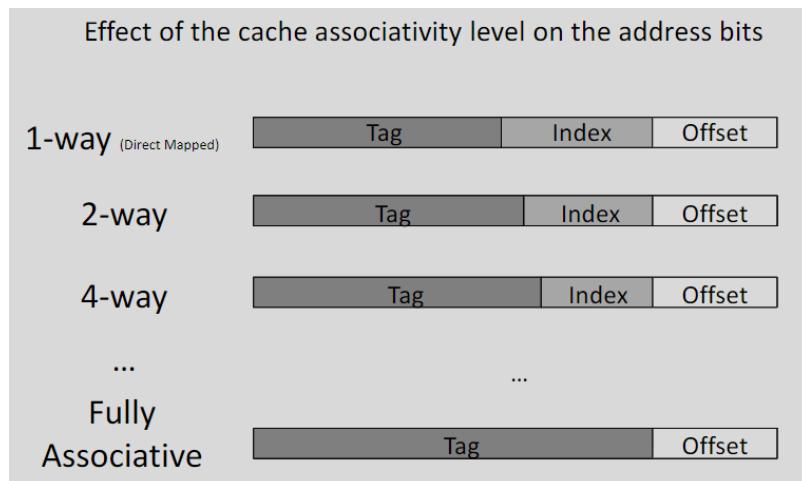
$$k = i \bmod s$$

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.



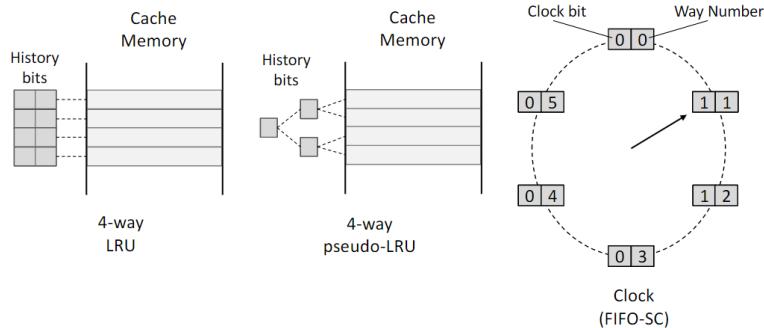
**Figura 5.7:** Fully Associative

## 5.4 Algoritmo di rimpiazzamento

Per sostituire le cache 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 frequently 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 approssimazione 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”, perché 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 principali 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.



**Figura 5.9:** AMD Zambesi

## 5.7 Esempio

Immaginiamo di avere una memoria cache con le seguenti caratteristiche:

- 64 kbyte di dimensione
- 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:

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

# 6 Branch Prediction Techniques

I salti possono potenzialmente impattare in modo molto rilevante sulle prestazioni delle pipeline. Per questo motivo è possibile ridurre la perdita di performance andando a eseguire delle predizioni su quello che sarà il risultato del salto.

Lo scopo è quello di eseguire correttamente il forecasting branches, riducendo le chance che il controllo dipenda da cause di stallo. Possiamo categorizzare gli schemi in due gruppi:

- **static techniques**: sono gestite dal compilatore come risultato di una analisi preliminare del codice.
- **dynamic techniques**: sono implementate in hardware in base al comportamento del codice.

L'accuratezza della predizione non aumenta in modo significativo a causa di un aumento della dimensione del buffer o del numero dei bit utilizzati per la predizione.

## 6.1 Static branch prediction

Lo **static branch prediction** può essere utile in combinazioni con altre tecniche statiche come l'**enabling delayed branches** e il **rescheduling to avoid data hazards**.

Il compilatore può prevedere il comportamento in modo differenti:

- predire sempre i branch come **presi**
- predire in base al **branch direction** (se siamo all'interno di un ciclo)
- predire in base alle informazioni di **profiling** dalle prime esecuzioni.

Nei programmi per *SPEC92* il 34% delle predizioni basate su *branch sempre presi* erano errate, con un rate molto variabile da 9% al 59%. Altre tecniche potrebbero avere un comportamento migliore nel caso medio, ma sono ancora con alte variazioni da programma a programma.

Nel caso di predizioni basate su branch direction si ha che i *forward branches* solitamente sono poco presi mentre i *backward branches* sono molto spesso presi (come ad esempio in un ciclo).

Il *profiling* è una tecnica che può essere utilizzata per prevedere una sequenza tipica di input che riceve il programma ed eseguire il programma un numero limitato di volte utilizzando tale input, eseguendo una statistica in base al comportamento dei branch.

## 6.2 Dynamic branch prediction

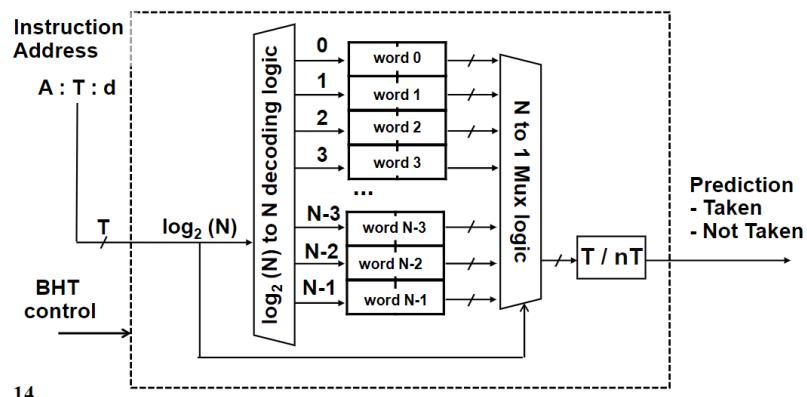
Gli *schemi dinamici* sono implementati in hardware, e utilizzano gli indirizzi delle istruzioni di *branch* per attivare meccanismi differenti di predizioni. Possono essere implementati seguendo tecniche differenti:

- *branch history table*
- *two level prediction schemes*
- *branch-target buffer*
- *altri...*

### 6.2.1 Branch history table

E' il metodo più semplice di *predizione dinamica*. La **branch History Table** (*BHT*) è una piccola memoria indicizzata dalla più piccola porzione di indirizzo dell'istruzione di branch. Ciascuna entry ha uno o più bit che registrano se il branch è stato preso o meno l'ultima volta che è stato eseguito.

Ogni volta che viene decodificata una istruzione di branch, viene eseguito un accesso alla *BTH* utilizzando la porzione di indirizzo per l'indice. La predizione viene salvata nella tabella utilizzata, e il nuovo program counter viene calcolato in accordo a tale predizione. Quando il risultato di un branch è noto, la tabella viene aggiornata.



**Figura 6.1:** BHT Implementazione

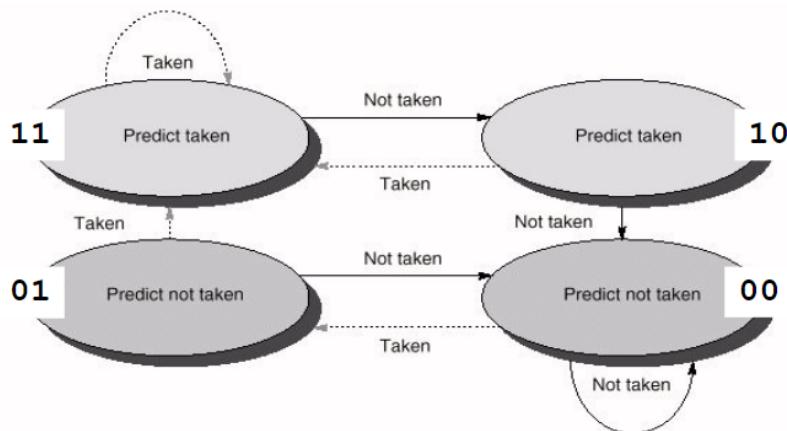
L'efficacia dipende dal metodo utilizzato, ma soffre del problema di **aliasing** (una linea della tabella potrebbe far riferimento a un altro salto e non quello che sto eseguendo) e dell'accuratezza della predizione.

Nei processori *MIPS*, la valutazione della condizione del branch viene eseguita quando viene identificata l'istruzione di branch, per tale motivo non porta nessun reale vantaggio.

Se prendiamo un esempio di un loop che è stato preso 9 volta di seguito e poi al decimo deve uscire. Se ipotizziamo che la entry non è condivisa con altri branch, la predizione utilizzando una BHT di 1bt, verrà sbagliata solo la prima e l'ultima iterazione con un successo del 80%, minore del fatto che venga assunto che venga sempre preso (90%).

### 6.3 Two bit prediction schemes

Gli schemi predittivi a 2 bit permettono una capacità di predizione più elevata. Per ogni branch vengono mantenuti due bit, la predizione è cambiata solo dopo due miss.



**Figura 6.2:** Two bit prediction schemes

**Attenzione:** Non si comporta come un contatore

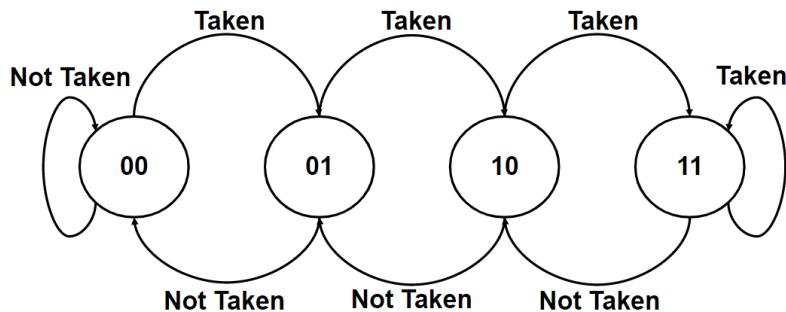
#### 6.3.1 n-bit prediction schemes

Gli schemi predittivi a n-bit permettono una capacità di predizione ancora più elevata, e rappresenta un caso più generale di quella precedente. Il counter, che si comporta effettivamente come un contatore, è aumentato di uno ogni volta che il branch viene preso e decrementato in caso contrario.

Quando il counter è maggiore di metà del suo valore massimo, il branch viene predetto come preso, altrimenti come non preso.

Alcuni esperimenti hanno mostrato come ci sia un vantaggio con  $n > 2$ .

Viene inoltre denominata **bimodal branch prediction**.



**Figura 6.3:** Saturating counter con 2 bit

## 6.4 Impatto sulle performance

Le performance dei branch vengono impattate da alcuni fattori:

- **accuratezza della precisione**
- **costo del branch** (penalità per predizione errata)
- **branch frequency** (minore per programmi in floating point)

## 6.5 Correlating Predictors

In questo tipo di predittori, la storia dei salti precedenti influenza la scelta attuale di predizione.. Questo approccio, denominato anche two-level predictors, è dunque basato sulla dipendenza tra i risultati degli ultimi branch.

### 6.5.1 ( $m, n$ ) predictors

Utilizzano il comportamento degli ultimi  $m$  branch per scegliere da  $2^m$  branch predictors, ciascuno ad  $n$ -bit.

L'hardware necessario per implementare lo schema è molto semplice:

- la storia dei più recenti  $m$  branches è registrato in uno shift register ad  $m$ -bit, dove ciascun bit registra se un salto è stato preso o meno.
- Il branch prediction buffer è indicizzato utilizzando una concatenazione di bit low order del branch address con  $m$  low order history bits.

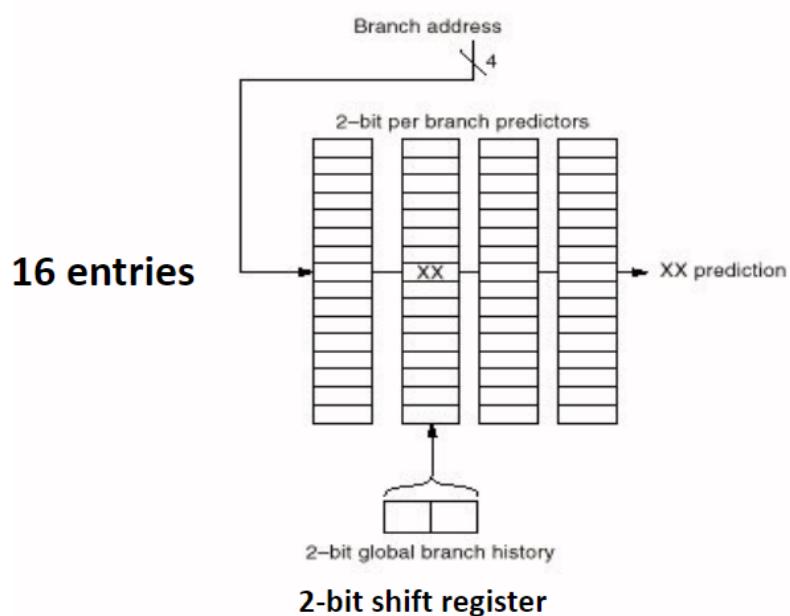
### 6.5.2 (1,1) predictor

In questo caso abbiamo  $m=1$  ed  $n=1$ . Ciascun branch è associato con  $2^m$  predictors di  $n$  bits:

- uno riporta la predizione nel caso il branch precedente è stato preso
- uno riporta la predizione nel caso il branch precedente non è stato preso

### 6.5.3 (2,2) predictor

Un altro esempio potrebbe essere con  $m=2$  ed  $n=2$ , con la seguente costruzione:



**Figura 6.4:** (2,2) Predictor

Come si vede è sufficiente uno shift register con due bit.

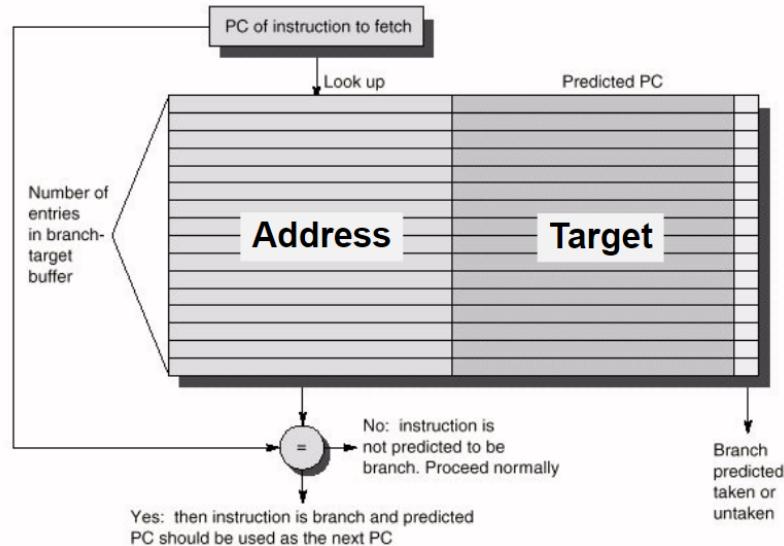
## 6.6 Branch-target buffer

Perciò ridurre il numero di effetti sul controllo delle dipendenze richieste da sapere appena possibile, come l'informazione che il branch sia preso o meno e il nuovo valore del *program counter* (se il branch è assunto di essere prese), vengono risolti attraverso l'introduzione del branch-target buffer (o cache).

Ciascuna entry del branch-target buffer contiene l'**indirizzo** del branch considerato e il **target** value da caricare nel program counter.

Utilizzando un branch-target buffer, il program counter è caricato con il nuovo valore alla fine dello stage di fetch, prima ancora che la branch instruction sia decodificata.

Questa tecnica è molto buona, ma ha però un costo molto elevato.

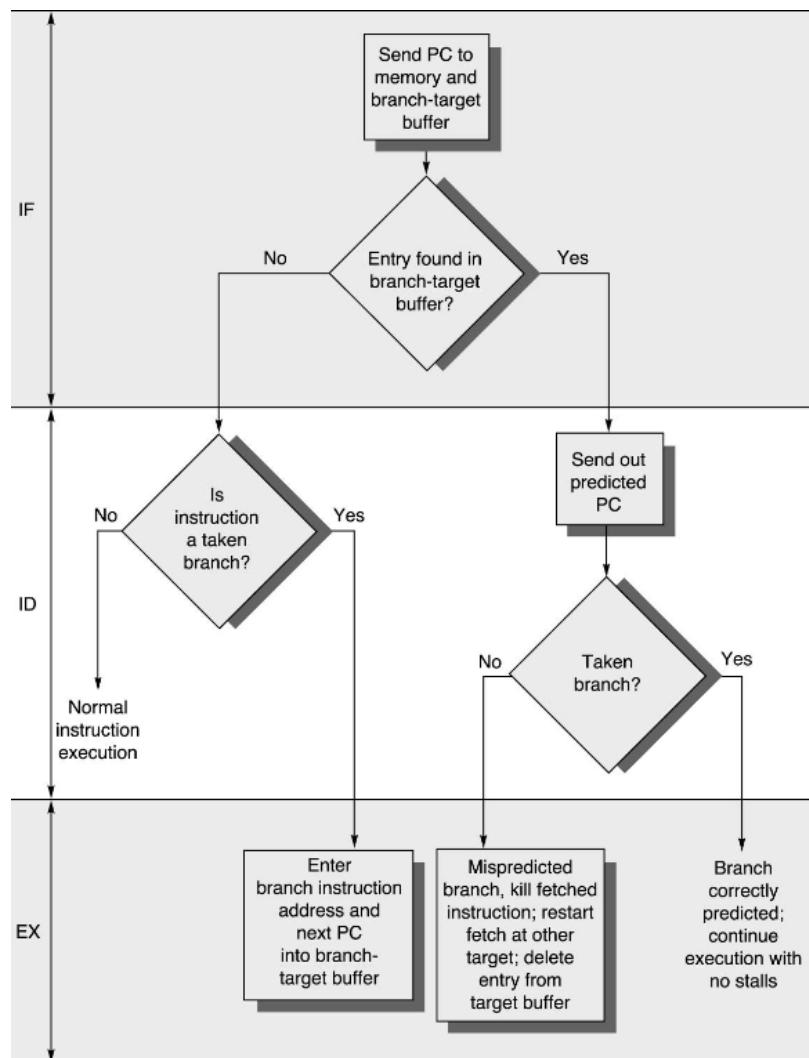


**Figura 6.5:** Architettura

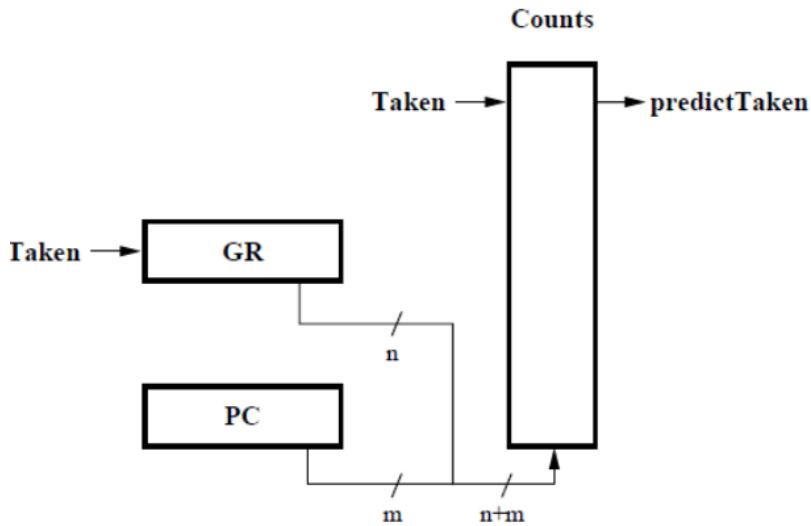
Se viene utilizzato un meccanismo con di predizione a 2-bit, è possibile combinare un branch-target buffer con un branch prediction buffer.

## 6.7 gselect

La tabella di predizioni mediante saturated counter è acceduta concatenando il branches global history (GR) con il branch address (PC).

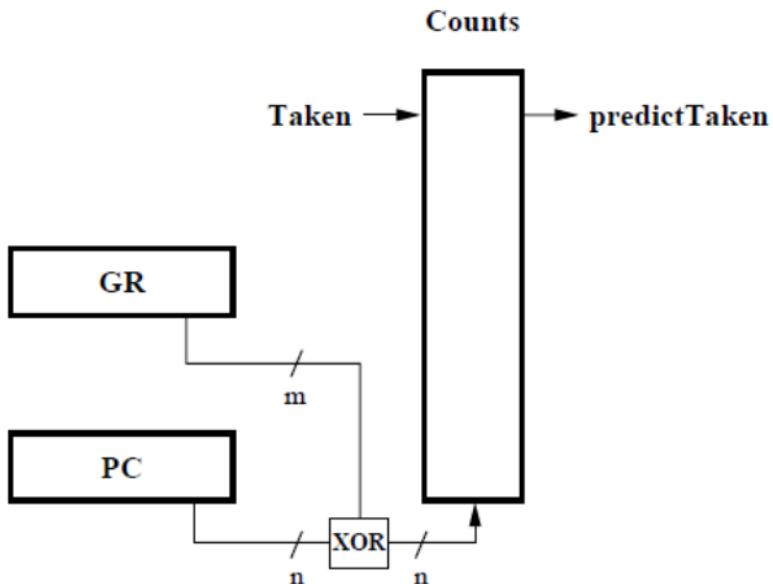


**Figura 6.6:** Comportamento

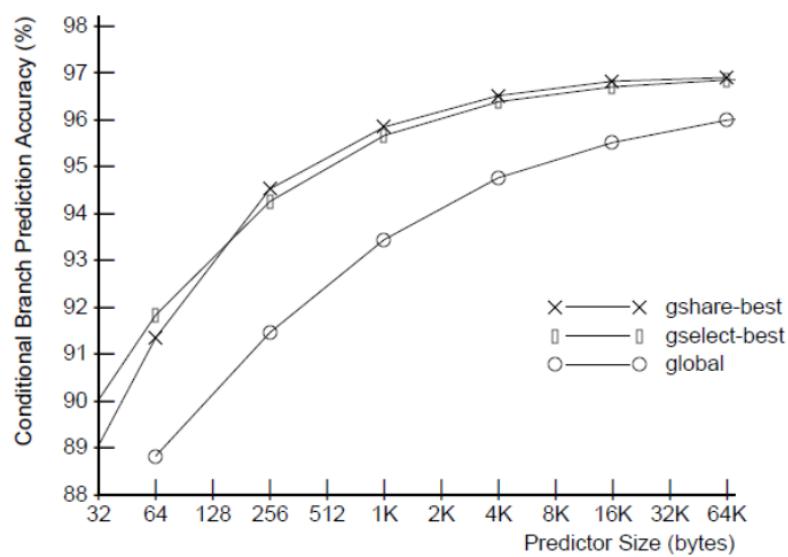


## 6.8 gshare

La tabella di predizione è ottenuta effettuando lo XOR tra il branches global history (GR) e il branch address (PC).



Incredibilmente funziona meglio del gselect.





# 7 Dynamic scheduling Techniques

Lo scheduling dinamico consente di identificare le dipendenze che sono sconosciute a tempo di compilazione. Semplifica il lavoro del compilatore e permette al processore di tollerare ritardi non prevedibili. Inoltre, permette di eseguire lo stesso codice su differenti processori.

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

La pipeline ha uno stallo dopo la `DIV.D` a causa della dipendenza tra `DIV.D` e `ADD.D`. La `SUB.D` si pone in pausa a sua volta, anche se in realtà non vi è nessuna dipendenza con le istruzioni precedenti. Potremmo riuscire ad aumentare le prestazioni rimuovendo la necessità di eseguire le operazioni in ordine.

## 7.1 Esecuzione fuori ordine

L'esecuzione fuori ordine prevede l'esecuzione di istruzioni che non sono state ancora eseguite. Questo può però comportare problemi, in particolare intruducendo:

- WAR hazards: write after read
- WAW hazards: write after write
- esecuzione imprecisa

Se l'esecuzione out-of-order è concessa, potrebbe diventare impossibile riuscire a fare una gestione precisa delle eccezioni. Questo potrebbe comportare che quando una eccezione è rilasciata, una istruzione precedente o successiva all'eccezione ha ancora da essere completata. In entrambi i casi potrebbe essere difficoltoso far ripartire il programma.

## 7.2 Splitting ID-Stage

Per consentire l'esecuzione fuori ordine, è necessario dividere l'ID-Stage in due parti:

- issue: decode instruction, controlla gli hazard strutturali

- read operands: aspetta fino a quando non ci sono data hazards, solo dopo legge gli operandi

La fase di issue legge l'istruzione da un registro o da una coda (scelta nella fase di fetch). Solo dopo si dovrà attendere per gli operandi e solo a quel punto entrare in fase di esecuzione.

Le istruzioni possono essere messe in stato di stallo o bypassare durante le fasi di lettura degli operandi, per questo motivo potrebbe entrare in una fase di esecuzione out of order

### 7.3 EX stage

Se il processore include più unità funzionali, molte istruzioni possono essere eseguite in parallelo. Anche in questo caso le istruzioni potrebbero bypassare durante le fasi di esecuzione, per questo motivo possono uscire dalla esecuzione out-of-order.

### 7.4 Hardware Schemes for dynamic scheduling

Alcune strategie hardware potrebbero essere applicate per risolvere i problemi di scheduling dinamico. In particolare:

- scoreboard
- Tomasulo's algorithm

#### 7.4.1 Tomasulo's algorithm

Roberto Tomasulo è stato architetto del processore IBM 360/91, le sue idee sono state pubblicate in un paper del 1967. Le stesse idee vennero poi riutilizzate per il primo processore superscalare costruito.

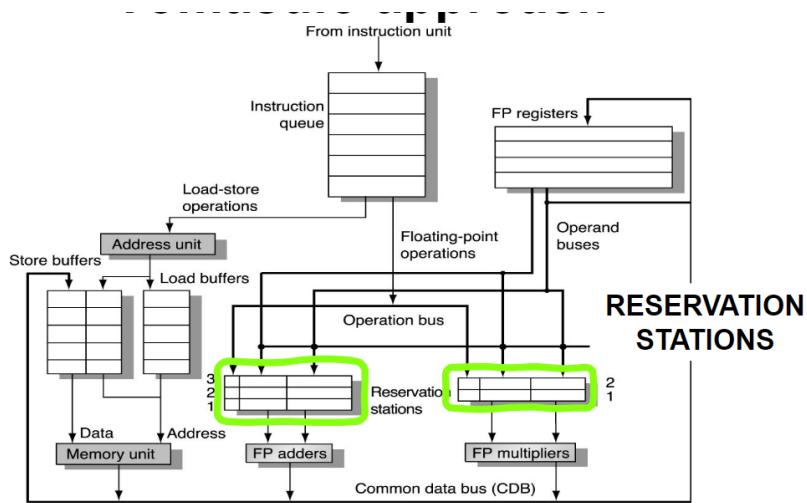
Le principali idee sono:

- tracciare la disponibilità degli operandi
- introduzione del *register renaming*

##### 7.4.1.1 Reservation stations

Le **reservation stations** sono la novità chiave nell'approccio di Tomasulo, hanno diverse funzioni:

- **bufferizzano gli operandi** delle istruzioni in attesa di eseguire; gli operandi vengono memorizzati nella stazione non appena come sono disponibili



**Figura 7.1:** MIPS FP unit structure using Tomasulo approach

- implementano la logica di emissione
- identificano univocamente un'istruzione nel pipeline: le istruzioni in attesa designano il'istazione di prenotazione che fornirà loro un operando di ingresso.

Ciascuna reservation station relativa a un'unità funzionale controlla quando una istruzione può iniziare l'esecuzione in quella unità.

#### 7.4.1.2 Register renaming

Ogni volta che viene impartita una istruzione, il registro specifica che gli operandi pendenti sono rinominato ai nomi delle reservation station in carica di calcolarli.

Questo implementa una strategia di register renaming, in grado di eliminare gli hazard WAW e WAR.

#### 7.4.1.3 Common Data Bus (CDB)

I risultati sono passati direttamente alle altre unità funzionali, piuttosto che andare nei registri. Tutti i risultati dalle unità funzionali e dalla memoria sono inviati sul Common Data Bus, il quale:

- goes everywhere (except to the load buffer) ??
- allows all units waiting for an operand to load it simultaneously when it is available.

#### 7.4.1.4 Instruction execution steps

Le istruzioni vengono eseguite in 3 fasi:

- Issue
- Execute
- Write result

Possono avere una lunghezza differente.

**7.4.1.4.1** Quando una istruzione viene presa dalla coda (strategia FIFO), se non sono disponibili reservation station allora avviene uno structural hazard e l'istruzione è posta in stallo fino a quando una reservation station non diventa disponibile. Se invece è disponibile una reservation station, vi viene inviata l'istruzione con gli operandi se sono disponibili, altrimenti si attende anche la loro disponibilità.

**7.4.1.4.2 Execute** Quando un operando appare sul CDB, viene letto dal reservation unit e appena tutti gli operandi dell'istruzione sono disponibili nella reservation unit, l'istruzione può essere eseguita. In questo modo sono eliminate le RAW hazards.

L'e istruzioni di load e store eseguite in due step:

- appena il base register è disponibile, l'effective address è calcolato e scritto nel buffer della load/store
- Nel caso della load è eseguita appena la memoria è disponibile mentre per la store si attende che gli operandi che devono essere scritti siano disponibili e l'esecuzione avviene appena la memoria è disponibile.

Per evitare di modificare il comportamento delle eccezioni, nessuna istruzione è autorizzata a iniziare l'esecuzione fino a quando tutti i branch precedenti non sono stati completati. Per questo motivo la **speculazione** potrebbe essere implementata per migliorare questo meccanismo.

**7.4.1.4.3 Write result** Quando il risultato dell'istruzione è disponibile, viene immediatamente scritto nel CDB, dove i registri e le unità funzionali attendono.

In questo step le istruzioni di step scrivono in memoria.

#### 7.4.1.5 Instruction identifiers

ciascuna reservation station è associata a un identificatore. Questo identifica anche gli operandi necessari per l'istruzione, è in questo modo che quest'ultima li riconosce.

Gli identifiers hanno anche il compito di implementare funzionare come virtual register che possono essere utilizzati per implementare il register renaming.

#### 7.4.1.6 Reservation station fields

Ogni reservation station ha i seguenti campi:

- Op: operazione da eseguire
- V<sub>j</sub>: valore del operando sorgente j
- V<sub>k</sub>: valore del operando sorgente k
- Q<sub>j</sub>: reservation station che produrrà un operando sorgente j
- Q<sub>k</sub>: reservation station che produrrà un operando sorgente k
- A: utilizzato dal buffer della load/store per memorizzare prima il campo immediato e poi l'effective address
- Busy: status della reservation station

Op	V <sub>j</sub>	V <sub>k</sub>	Q <sub>j</sub>	Q <sub>k</sub>	A	Busy
----	----------------	----------------	----------------	----------------	---	------

#### 7.4.1.7 Register file

Ogni elemento in un register file contiene un campo Q<sub>i</sub>. Questo, contiene il numero della reservation station che contiene l'istruzione il cui risultato dovrebbe essere salvato in un registro. Se il Q<sub>i</sub> è null, neon è attualmente attiva nessuna istruzione che sta calcolando un risultato per quel registro.

In questo modo la hazard detection logic è distribuita e gli stalli per WAW e WAR sono eliminati. Si ha però come contro una alta complessità hardware (includendo un buffer associativo per ogni reservation station) e il CDB potrebbe essere un bottleneck.

#### 7.4.1.8 Loop handling

Il loop unrolling non è necessario in questa architettura, in quanto possono essere eseguite normalmente in parallelo.



# 8 Speculation

La speculazione basata su hardware è una tecnica per ridurre l'effetto delle dipendenze di controllo in un processore che implementa il dynamic scheduling.

Se un processore supporta la branch prediction con dynamic scheduling, le istruzioni di fetch ed issue sono eseguite come se la predizione fosse sempre corretta.

Se un processore supporta questo tipo di speculazione, le esegue sempre.

Le idee sono principalmente tre

- dynamic branch prediction
- dynamic scheduling
- speculation

In questo modo il processore implementa una data flow execution, dove le operazioni sono eseguiti appena sono disponibili gli operandi.

## 8.1 architettura

Viene adottata l'architettura di Tomasulo, che viene però estesa per supportare la speculazione.

La fase di execute si separa nuovamente:

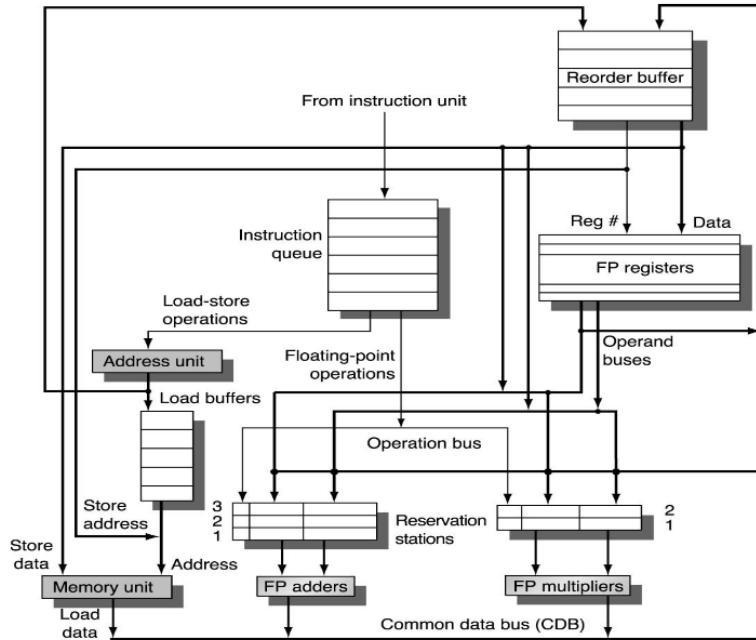
- il calcolo dei risultati
- l'aggiornamento del file register e della memoria, che viene eseguita quando una istruzione non è più speculativa (instruction commit).

A tal fine viene introdotto il **ReOrder Buffer** (ROB), una struttura dati che contiene i risultati delle istruzioni che non hanno ancora effettuato il commit. Si preoccupa di fornire ulteriori virtual register e integra lo store buffer presente nell'architettura originale di Tomasulo.

Con la speculazione, i dati potrebbero essere letti dal ROB se le istruzioni non hanno ancora eseguito commit oppure dal register file in caso contrario.

Ogni entry della ROB ha quattro campi:

- **Instruction type:** branch, store, or register
- **Destination:** register number, or memory address
- **Value:** contains the value when the instruction has completed but still didn't commit
- **Ready:** indicates whether the instruction completed its execution



## 8.2 Instruction Execution Steps

L'esecuzione delle istruzioni avviene in quattro fasi:

- Issue
- Execution
- Write result
- Commit

### 8.2.1 Issue

Una istruzione viene estratta dalla coda delle istruzioni se è disponibile una empty reservation station e un slot libero nel reorder buffer, in caso contrario l'istruzione viene posta in stallo.

Gli operandi delle istruzioni se presenti nel register file o nel reorder buffer, sono inviati alla riservetion station.

Il numero della entry del reorder buffer dell'istruzione viene vincolata alla reservation station per indicare l'istruzione.

### 8.2.2 Execution

L'istruzione è eseguita appena sono disponibili tutti gli operandi, in modo da evitare hazard di tipo RAW. Gli operandi sono presi, se possibile, dal CDB appena una istruzione la produce.

La lunghezza di questo step varia in base al tipo di istruzione (*ad esempio 2 per le load, 1 per operazioni intere, variabile per FP*).

### 8.2.3 Write result

Scriveremo nel command data bus appena il dato è libero e verrà inviato al reorder buffer.

Tutte le reservation station aspetteranno per il risultato prima di leggerlo, e ciascuna entry verrà poi segnata come disponibile.

### 8.2.4 Commit

Il reorder buffer viene ordinato in base all'ordine originale. Appena una istruzione raggiunge la testa del buffer:

- se è un branch non predetto correttamente, il buffer viene ripulito e l'esecuzione riparte con la corretta istruzione che segue
- se invece il branch è stato predetto correttamente, il risultato viene scritto nel registro o in memoria (nella store).

In entrambi i casi, l'entry del reorder buffer viene marchiata come libera.

Il reorder buffer è implementato come una **circular buffer**.

## 8.3 WAW e WAR hazard

In seguito alle precedenti implementazioni, gli hazard WAW e WAR non possono più avvenire in quanto il dynamic renaming è implementato e l'aggiornamento della memoria avviene in ordine.

### 8.3.1 RAW hazard

I raw hazards sono evitati in quanto avviene un enforcing del program order mentre i calcoli dell'effective address e la load wrt avvengono prima delle istruzioni di store (????).

Inoltre, una load non entra nel secondo step se qualche entry della ROB è occupata da una store che ha una destination field che è uguale della field che sta facendo la load.

## 8.4 istruzione di Store

Le istruzioni di store scrivono in memoria solamente quando è stato eseguito il commit. Per questo motivo, gli operandi in input sono richiesti al momento della commit piuttosto che nello stage di scrittura dei risultati. Questo significa che il ROB deve avere un ulteriore campo che specifica da dove gli operandi in input dovranno essere presi per ogni istruzione di store.

## 8.5 Exception Handling

Le eccezioni non sono eseguite appena vengono rilasciate, ma piuttosto quando sono aggiunte al reorder buffer.

Quando una istruzione effettua il commit, la possibile eccezione viene eseguita, e le istruzioni successive vengono eliminate dal buffer.

Se l'istruzione viene eliminata dal buffer, l'esecuzione viene ignorata. Per questo motivo la precise exception handling è supportata completamente.

## 8.6 Speculative expensive events

Quando avviene un evento costoso in termini di tempo (second-level cache miss, TLB miss) nel caso speculativo, alcuni processori aspettano per la sua esecuzione fino a quando l'evento non è più speculativo, invece gli eventi poco costosi sono normalmente eseguite in modo speculativo.

# **9 Vliw**

L'evoluzione dei processori superscalare ha portato a complessi processori composti da più unità funzionali, e tutti sono accomunati da una logica per rilevare e gestire le dipendenze, schedulare dinamicamente, previsione e speculazione su branch. Ciò ha comportato un aumento notevole della complessità dei processori.

Sono stati studiati anche approcci alternativi tra cui i VLIW.

I VLIW (Very Long Instruction Word) sono processori che hanno istruzioni molto lunghe e che hanno encode molte operazioni, che vengono caricate in parallelo. L'hardware include tante unità operazionali quanto quante sono richieste in una singola istruzione.

Questi processori sono molto diffusi per le applicazioni embedded.

Ciò ha comportato un software molto più complesso, in quanto è compito del compilatore decidere quali istruzioni impachettare insieme: exploding parallelism, unrolling loops, scheduling code in basic blocks, etc.

Si ha però una semplificazione dell'hardware, in quanto non è necessario effettuare alcun controllo di dipendenze tra le istruzioni e non è dunque necessario avere un'unità che si occupa di valutare quali istruzioni eseguire in parallelo.

Quando una operazione richiede uno stall, l'intero pacchetto di istruzioni viene posto in pausa in modo da preservare il flusso deciso dal compilatore.

## **9.1 Limitazioni**

Le performance che possono essere causate da un multiple issue processor sono limitate dalla limitazioni dei programmi ILP, difficoltà di costruire l'hardware, limitazioni specifiche di processori superscalari o vliw.

E' inoltre difficile trovare un numero sufficiente di istruzioni indipendenti da eseguire in parallelo, soprattutto se consideriamo le unità funzionali pipelined che abbiano una latenza minore di 1.

In generale, per evitare gli stalli sarebbe necessario avere un numero di operazioni indipendenti circa pari a:

avarage pipeline depth \* number of functional units

Con l'incremento di unità funzionali segue un aumento della bandwidth del file register e della memoria. Ciò significa un aumento della complessità hardware e una riduzione delle performance. Alcune soluzioni possibili sono:

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

La dimensione totale del codice è molto maggiore per i processori VLIW a causa di due fattori principali:

- i cicli sono srotolati in modo molto intensivo per aumentare il parallelismo
- emty slots nel encoding delle istruzioni

Spesso le istruzioni sono compresse in memoria e poi espansse quando caricate nel processore.

Un processore VLIW richiede spesso accesso alla memoria, che può essere un bottleneck in quanto la bandwidth è magigore e gli stalli per i miss in cash possono porre in attesa l'intero processore.

Un ulteriore problema è la compatibilità dei binari, che non può essere garantita in quanto ogni cambio di implementazione richiede una ricompilazione. Questo è uno dei principali svantaggi rispetto ai processori superscalari, che possono creare facilmente binari compatibili con processori precedenti. Object code translation o l'emulazione possono essere le soluzioni a questo problema.

## 9.2 EPIC

l'architettura EPIC (Explicitly Parallel Instruction Computing) fu introdotta nella fine degli anni 90 in alcuni processori HP e Intel, come Itanium. Lo scopo era quello di ottenere dei VLIW con una maggiore flessibilità, ottenendo successo nell'area dei processori high-end.

## 9.3 Classificazione

Le istruzioni processate in parallelo richiedono tre task principale:

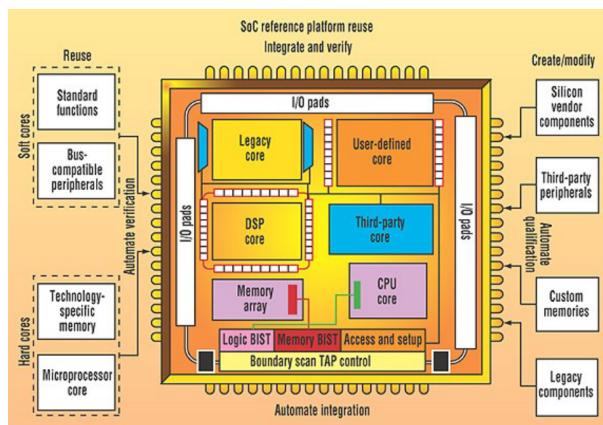
1. controllare le dipendenze tra le istruzioni per raggruppare le istruzioni per l'esecuzione parallela
2. Assegnare le istruzioni alle unità funzionali per inizializzarle insieme

# 10 Arm

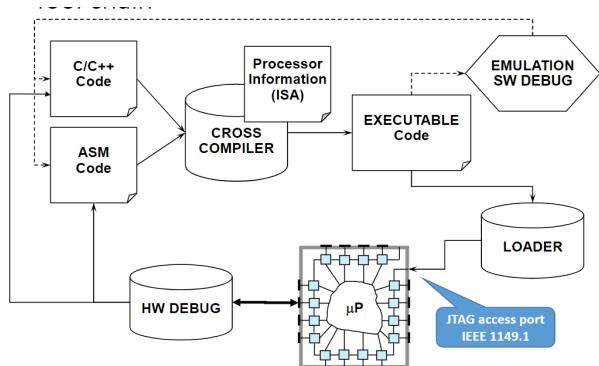
Arm è un architettura RISC che ha preso gran parte del segmento di processori in ambito mobile ed embedded.

L'architettura ARM si divide in:

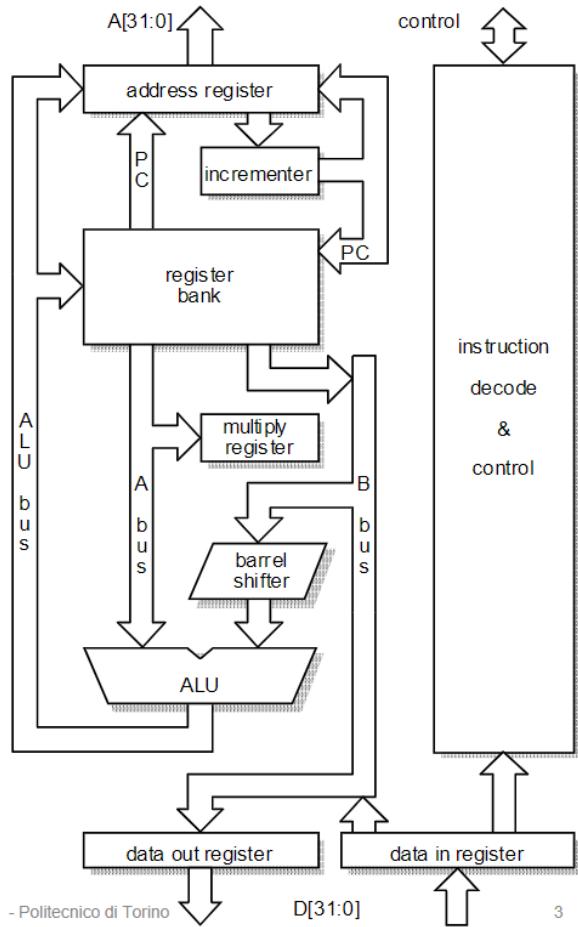
- ARM architecture embedded in System on chip (SoC)
- Arm operationg Systems
- ARM compile - support - debug tools



La toolchain di sviluppo arm è illustrata nella seguente immagine:



L'architettura generica di un processore arm è il seguente:



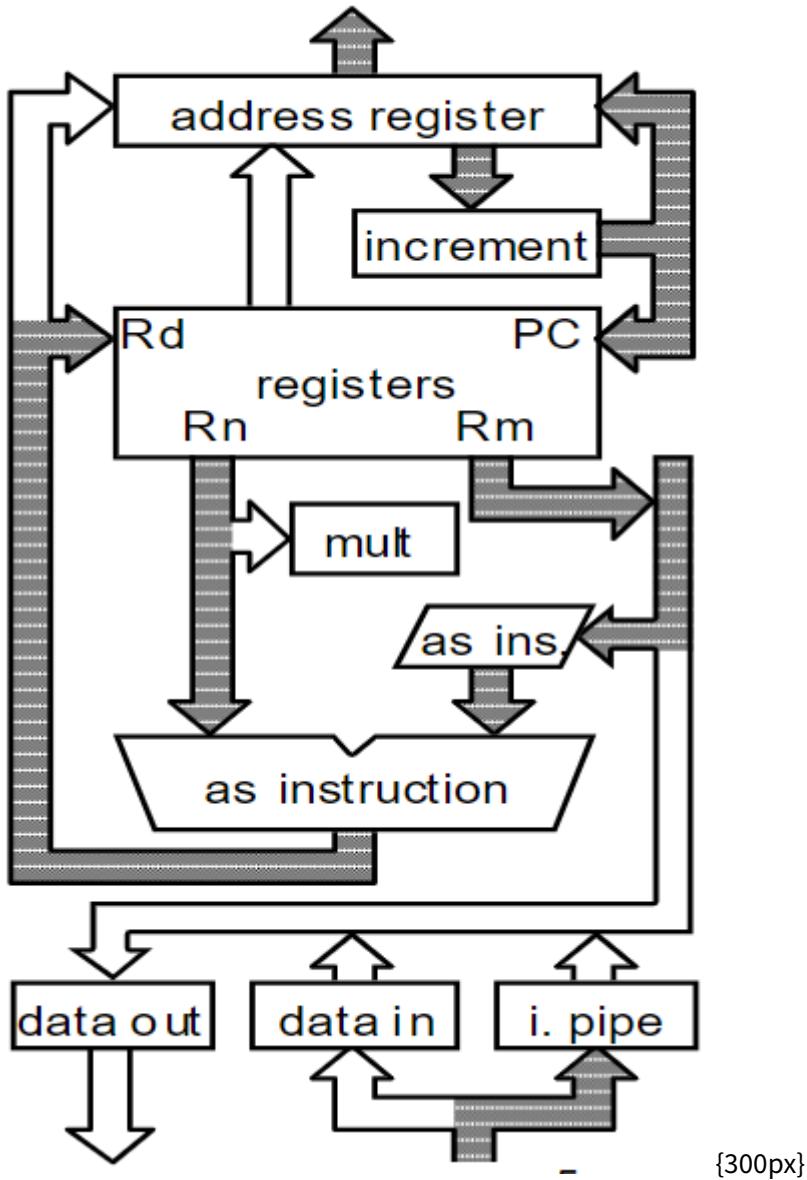
Il barrel shifter consente di eseguire gli shift in modo automatico del secondo registro di una operazione senza utilizzare una istruzione apposita.

## 10.1 Data Processing

### 10.1.1 Istruzioni reg-reg

Quando viene eseguita una istruzione da registro a registro:

- due operandi vengono letti dai registri **Rn** ed **Rm**
- un operando potrebbe essere “ruotato” ( $0x00000A \Rightarrow 0x000A00$ )
- la **ALU** genera il risultato
- il risultato viene scritto nel registro **Rd**
- la prossima istruzione viene recuperata dalla memoria
- il Program Counter viene aggiornato



### 10.1.2 Istruzioni reg-imm

Quando viene eseguita una istruzione registro-immediato:

- un operando viene letto dal registro **Rn**, l'altro è immediato
- un operando potrebbe essere “routato”
- la *ALU* genera il risultato
- il risultato viene scritto nel registro **Rd**
- la prossima istruzione viene recuperata dalla memoria

- il Program Counter viene aggiornato

## 10.2 Data Transfer Instructions

Le istruzioni di trasferimento dati richiedono due cicli di esecuzione nello stage di esecuzione. Nel primo viene calcolato l'indirizzo utilizzando un registro e un immediato, mentre nel secondo avviene un accesso in memoria nel quale viene trascritto il dato.

## 10.3 Branch instructions

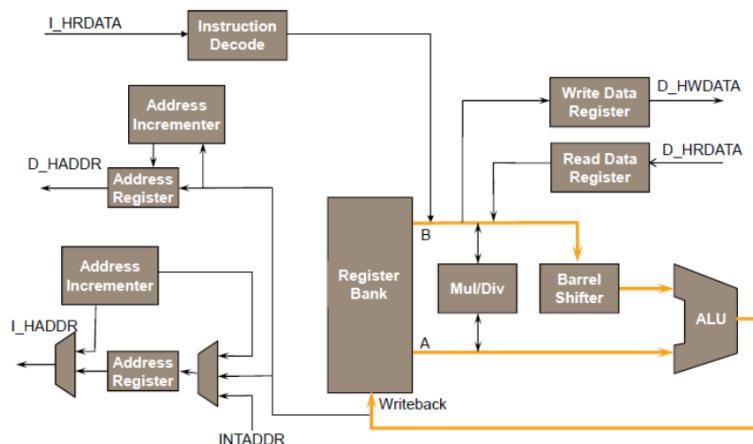
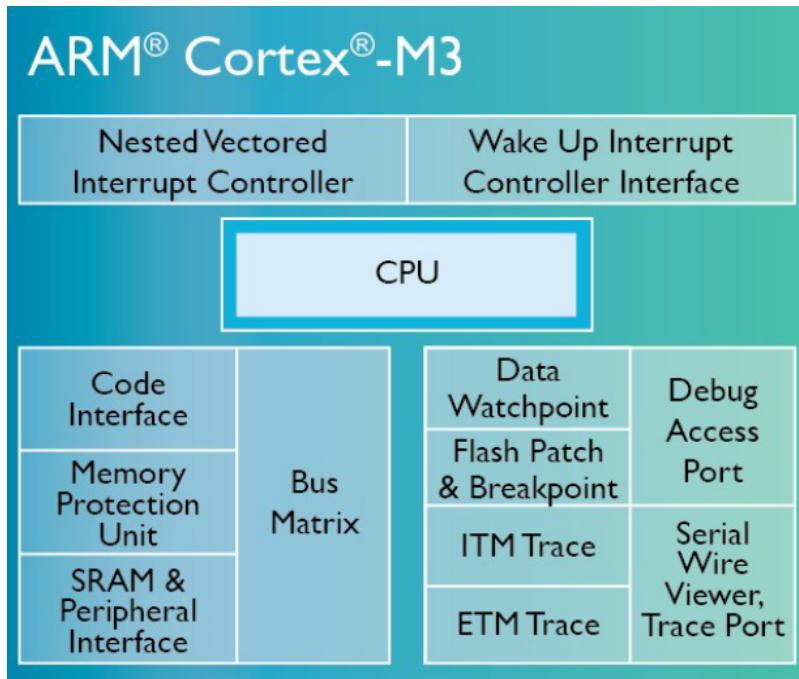
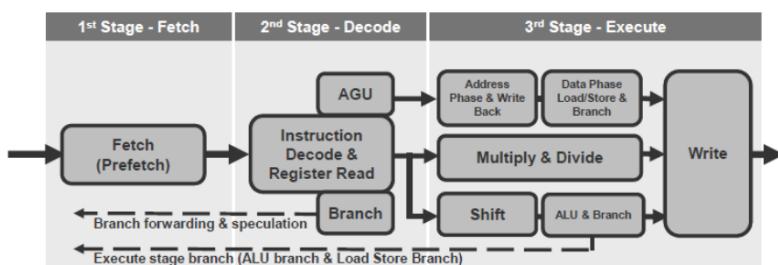
Prima viene calcolato l'indirizzo target, aggiungendo un immediato (shiftato di due posizioni) al program counter, dopo la pipeline viene svuotata e riempita nuovamente.

## 10.4 branch and link instructions

In questo caso, un clock aggiuntivo è necessario in quanto è necessario salvare l'indirizzo di ritorno in `r14`.

## 10.5 ARM Cortex M3

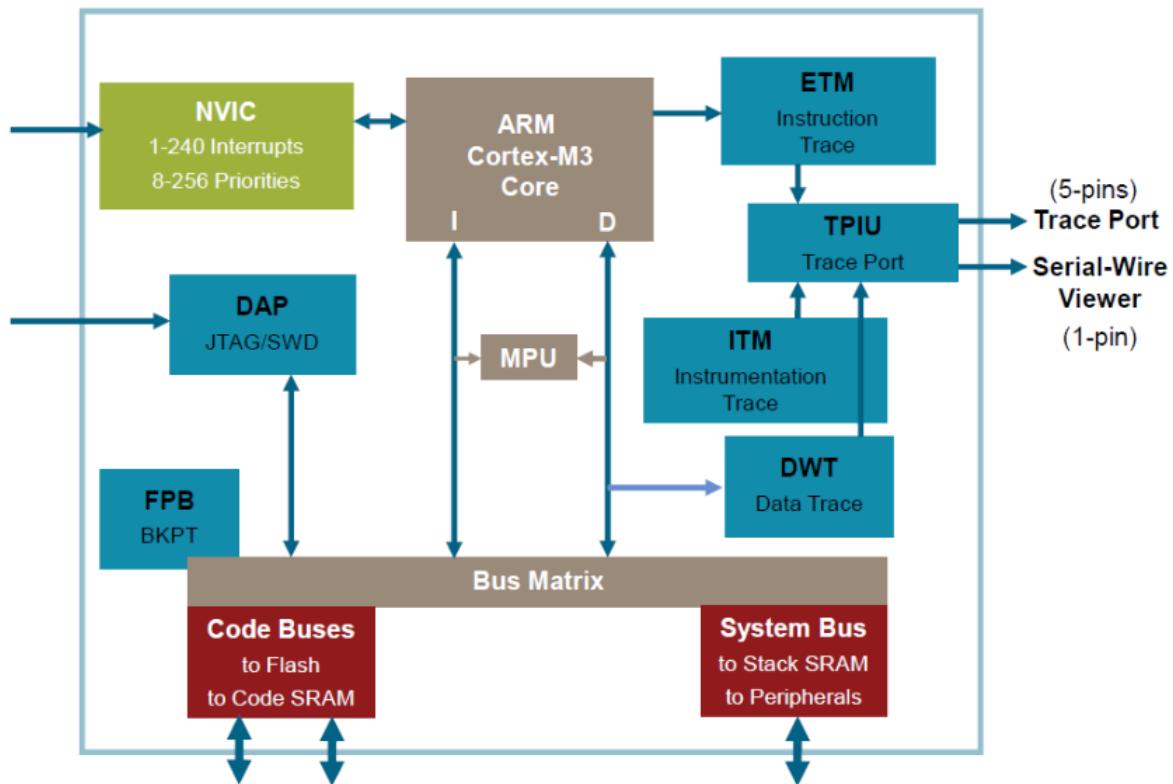
L'architettura del cortex M3 è rappresentata di seguito:

**Figura 10.1:** datapath

I salti sono un problema in quanto richiedono 3 cicli per essere completati. Nel caso peggiore, il sal-

to indiretto viene preso e avviene sempre che la pipeline viene svuotata e riempita. Inoltre, non è supportato il delayed branch mechanism.

Quando leggiamo della memoria perdiamo un ciclo di clock.



## 10.6 Programmer View

Sono messi a disposizione:

- 18 registri a 32 bit
- handling delle interruzioni efficiente
- power management enable idle mode
- efficient debug and development support features
- grande supporto del sistema (user/supervisor)
- realizzato per essere completamente programmabile in C

L'instruction set thumb (processori con **T** nell'acronimo) sono a 16 bit, sono meno potenti e in numero minore.

Thumb2 venne aggiunto da ARM nel 2003 come suoerset di Thumb (garantendo retrocompatibilità) e include nuove istruzioni a 16 bit e alcune a 32 bit. È più veloce della prima versione ma riesce a mantenere la dimensione del codice molto compatta.

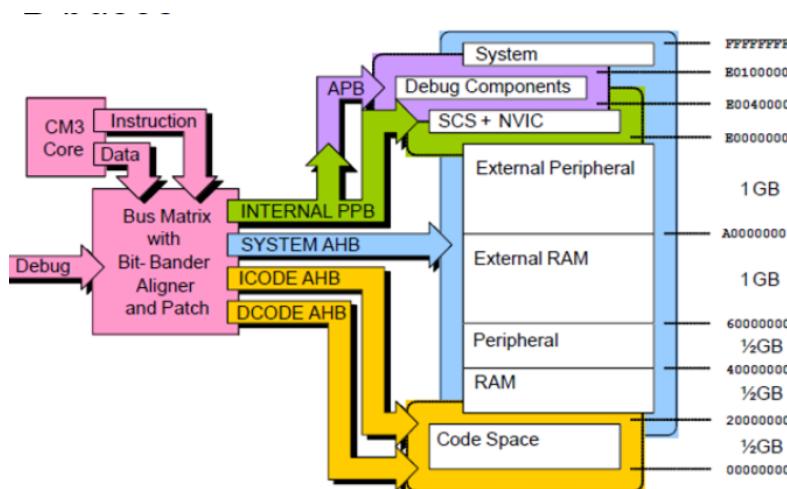
## 10.7 Bus

Il AMBA bus system prevede 3 bus:

- AHB: Advanced High Performance Bus
- ASB: Advanced System Bus
- APB: Advanced Peripheral Bus

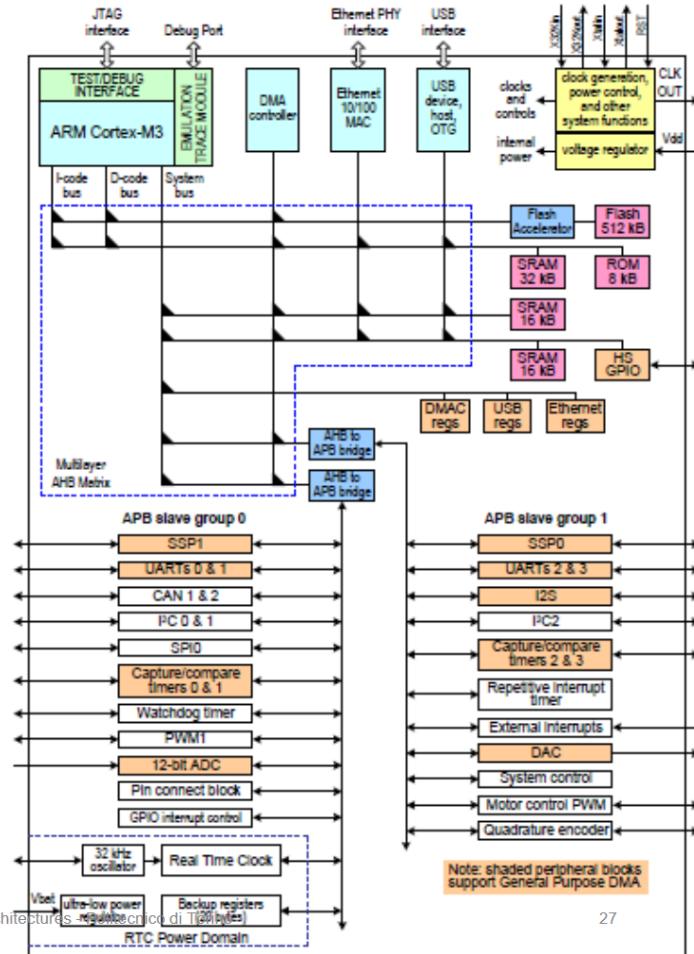
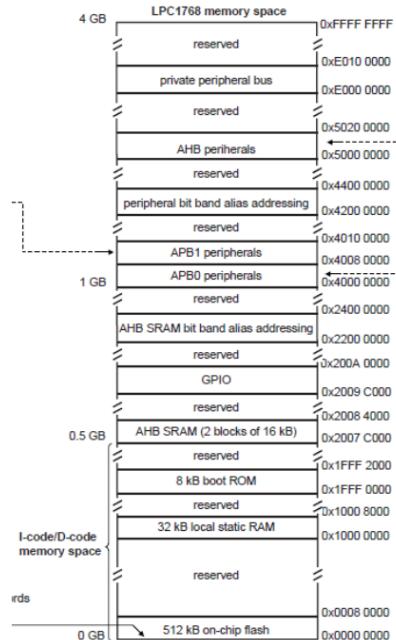
## 10.8 Memory map

La memoria è mappata in 4gb (derivanti dai 32 bit di indirizzi) e la bus matrix viene acceduta mediante i bus AHB e PPB.



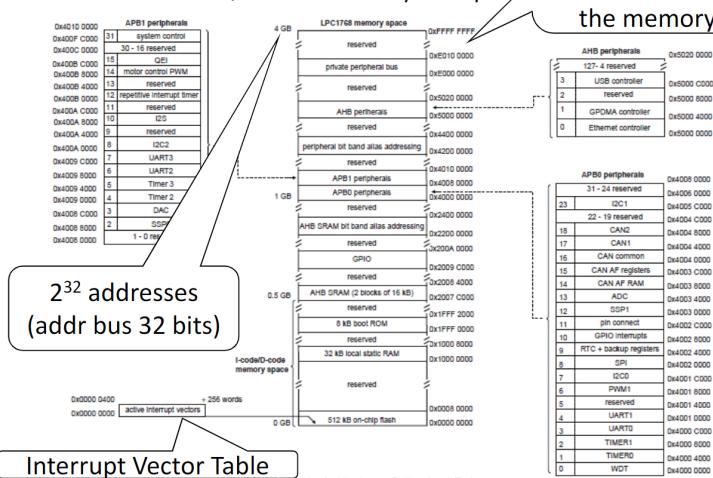
**Figura 10.2:** Memory map

## NXP LPC176x/5x block diagram and memory map



{width=400px}

### NXP LPC176x/5x memory map



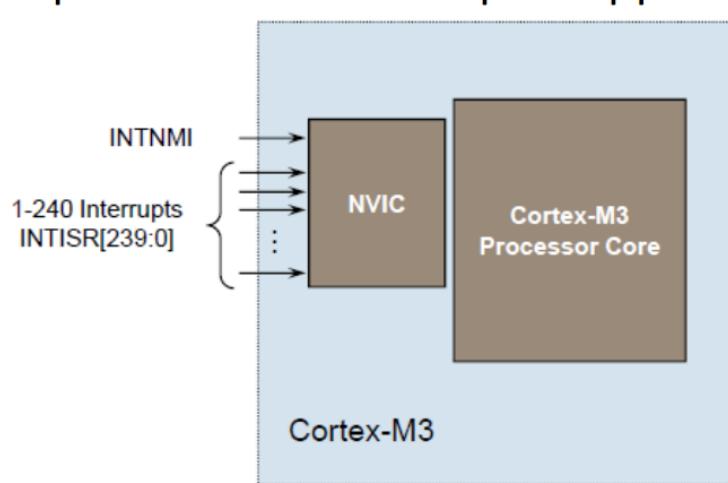
## 10.9 Exception

Le eccezioni sono le seguenti:

- Reset
- NMI
- Faults
  - hard fault
  - memory manage
  - bus galut
  - usage fault
- SVCall
- Debug Monitor
- PendSV
- SysTick Interrupt
- External Interrupt

## 10.10 Interrupt

E' supportata una interruzione non mascherabile INTNMI. Insieme al processore è presente un Nested Vectored Interrupt Controller (NVIC) che supporta fino a 240 interruzioni esterne.



## 10.11 Clock distribution

I sistemi come come arm v7-M hanno bisogno di due clock:

- uno ad alta frequenza per la CPU e le componenti veloci
- uno a bassa frequenza per i core periferici che necessitano di meno performance o che operano a velocità limitate.

Il clock della CPU (CCLK) e il clock periferico (PCLK) ricevono il clock input da un PLL (Phase Lock Loop), VPB (VLSI Peripheral Bus) Divider o da un clock esterno.

## 10.12 Power management

Dal punto di vista energetico sono supportate più modalità di sleep:

- sleep now
- sleep on exit
- deep sleep

## 10.13 Istruzioni

Il cortex M3 supporta il clock in tutte le modalità, anche da una fonte esterna. E' inoltre presente un Wake-UP Interrupt Controller (WIC), ovvero una fonte esterna di wake-up che permette al processore di spegnersi completamente. Effective with State-Retention / Power Gating (SRPG) methodology.

Le istruzioni sono a 32 (o 16) bit e possono essere eseguite in modo condizionale. E' presente una architettura di load/store, con la caratteristica che le istruzioni di processazione di dati avvengono solo su registri. Il formato utilizzato prevede 3 operandi e combina ALU e shifter. L'accesso alla memoria avviene con istruzioni dotate di auto-indexing.

L'instruction set può essere esteso attraverso dei coprocessori.

In particolare il cortex M3 prevede l'utilizzo di 18 registri a 32 bit che supportano come tipi di dato il byte (8 bit), halfword (16 bit) e word (32 bit).

Il program ounter è il registro [r15](#). Quando il processore è eseguito in ARM state tutte le istruzioni sono lunghe 32 bit e sono allineate a word. Per questo motivo, il PC value è salvato nei bit [31:2] con i bit [1:0] pari a 0.

A differenza di molti altri processori come 80x86, ARM consente la scrittura diretta del Program Counter attraverso il registro [R15](#).

### 10.13.1 Link register (R14)

Il link register è il registro `r14`, che viene utilizzato per memorizzare l'indirizzo di ritorno quando il branch con operazioni link vengono eseguite, calcolate a partire dal `PC`.

Per ritornare da un linked branch è sufficiente eseguire `MOV r15, r14` oppure `MOV pc, lr`.

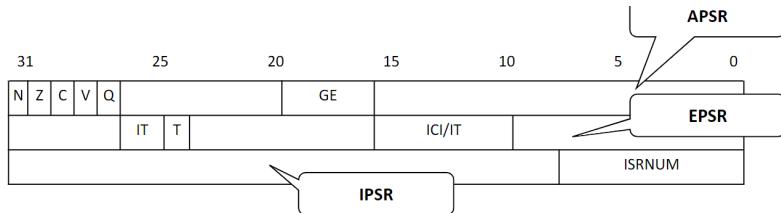
### 10.13.2 Stack pointer (R13)

Il registro `r13` viene utilizzato come stack pointer e viene aggiornato automaticamente, in particolare a tempo di boot viene recuperato dal Interrupt Vector Table oppure si aggiorna quando il programma esegue una istruzione stack oriented.

### 10.13.3 Program Status register (R12)

Il Program Status Register è suddiviso in 3 registri:

- Application Program Status Register (APSR)
- Execution Program Status Register (EPSR)
- Interrupt Program Status Register (IPSR)



I flag di stato sono:

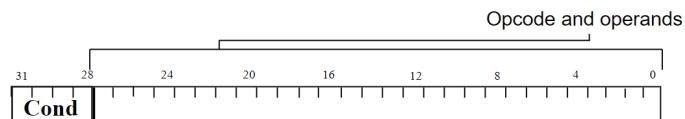
- N: Negative
- Z: Zero
- C: Carry
- V: oVerflow
- Q: Saturation

### 10.13.4 Esecuzione condizionale

La maggior parte delle set instruction permette ai branch di essere eseguite in modo condizionale.

Riutilizzando il condition evaluation hardware, ARM incrementa il modo significativo il numero di istruzioni. tutte le istruzioni contengono un campo condizionale che determina se la cpu le eseguirà o meno. Le istruzioni non eseguite richiedono un ciclo, in quanto necessitano di completare il ciclo per consentire il corretto fetching e decoding delle prossime istruzioni.

Questo permette di rimuovere la necessità di molti branch, che causano lo stallo delle pipeline (3 cicli per il refill). Ciò comporta un *very dense in-line code* e la penalità di non eseguire le istruzioni condizionali è compensata dal fatto che non è necessario eseguire il branch.



**Figura 10.3:** Condition filed

Codice	Significato
EQ	Equal
NE	Not equal
CS	Carry set (identical to HS), il carry vale 1
HS	Unsigned higher or same (identical to CS)
CC	Carry clear (identico a LO), il carry vale 0
LO	Unsigned lower (identical to CC)
MI	Minus or negative result
PL	Positive or zero result
VS	Overflow
VC	No overflow
HI	Unsigned higher
LS	Unsigned lower or same
GE	Signed greater than or equal
LT	Signed less than
GT	Signed greater than
LE	Signed less than or equal
AL	Always (this is the default)

To execute an instruction conditionally, simply postfix it with the appropriate condition:

- For example an add instruction takes the form:
- ADD r0,r1,r2 ;  $r0 = r1 + r2$  (ADDAL)
- To execute this only if the zero flag is set:
- ADDEQ r0,r1,r2 ; If zero flag set then... ; ...  $r0 = r1 + r2$
- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect).

To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an **S**. ADDS r0,r1,r2 ;  $r0 = r1 + r2$  ; ... and set flags

Molto importante!

## 10.14 Operazioni aritmetiche

### 10.14.1 Moltiplicazione

Moltiplicazione con risultato su 32 bit:

```
1 MUL <Rd>, <Rn>, <Rm>
```

Moltiplicazione con risultato **unsigned** su 64 bit:

```
1 UMULL <Rd1>, <Rd2>, <Rn>, <Rm>
```

Moltiplicazione con risultato **signed** su 64 bit:

```
1 SMULL <Rd1>, <Rd2>, <Rn>, <Rm>
```

**Nota:** Non ci sono differenze tra signed ed unsigned su 32 bit

**Attenzione:** tutti gli operandi devono essere registri.

## 10.15 Moltiplicazione con accumulazione

- MLA <Rd>, <Rn>, <Rm>, <Ra>  $Rd = Rn * Rm + Ra$
- MLS <Rd>, <Rn>, <Rm>, <Ra>  $Rd = Rn * Rm - Ra$
- UMLAL <Rd1>, <Rd2>, <Rn>, <Rm>  $Rd1, Rd2 = Rn * Rm + Rd1, Rd2$
- SMLAL <Rd1>, <Rd2>, <Rn>, <Rm> uguale a UMLAL, ma con valori con segno

## 10.16 Divisione

### 10.16.1 Divisione senza segno (unsigned)

```
1 UDIV <Rd>, <Rn>, <Rm>
```

### 10.16.2 Divisione con segno (signed)

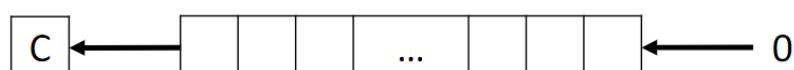
```
1 SDIV <Rd>, <Rn>, <Rm>
```

- If Rn is not exactly divisible by Rm, the result is rounded toward zero.
- UDIV and SDIV do not change the flags (the suffix ‘S’ can not be added)

## 10.17 Shift

### 10.17.1 Logical Shift Left (LSL)

```
1 LSL <Rd>, <Rn>, <op2>
```



**Figura 10.4:** Logical Shift Left

### 10.17.2 Logical Shift Right (LSR)

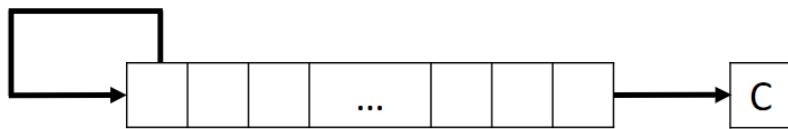
Lo shift avviene come ci aspettiamo, inserendo nel bit più significativo degli zeri e spostando il meno significativo nel carry.

```
1 LSR <Rd>, <Rn>, <op2>
```



### 10.17.3 Arithmetic Shift Right (ASR)

```
1 ASR <Rd>, <Rn>, <op2>
```

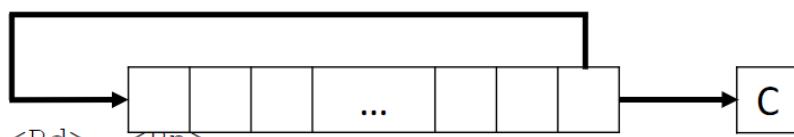


## 10.18 Rotazioni

Le istruzioni di rotazione operano nel seguente modo:

### 10.18.1 Rotate Right (ROR)

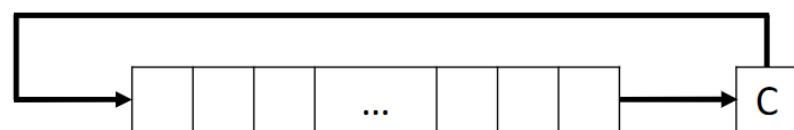
```
1 ROR <Rd>, <Rn>, <op2>
```



**Figura 10.5:** Rotate right

### 10.18.2 Rotate Right with Extend (RRX)

```
1 RRX <Rd>, <Rn>
```

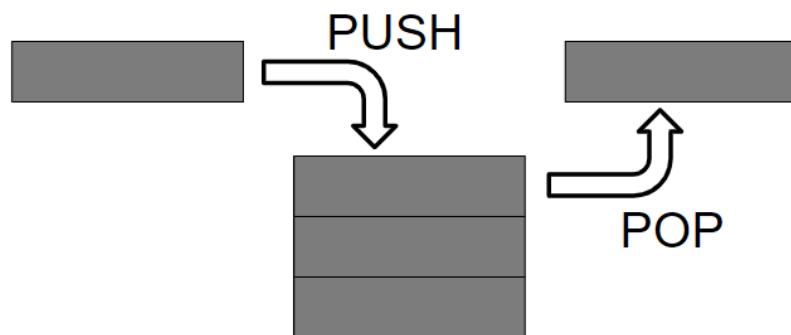


**Figura 10.6:** Rotate Right with Extend



# 11 Stack e subroutines

Lo stack ha una organizzazione di tipo *Last In - First out*, ovvero **LIFO**. I dati sono sempre inseriti (scritti) ed estratti (letti) dal top della pila. Lo **stack pointer** contiene l'indirizzo del top dello stack.



**Figura 11.1:** Stack

Lo *stack pointer* viene aggiornato dopo ogni push e può essere di due tipi:

- **descendente**: l'indirizzo del top della pila diminuisce dopo ogni push
- **ascendente**: l'indirizzo del top della pila aumenta dopo ogni push

Mentre il contenuto del top dello stack può essere anch'esso di due tipi:

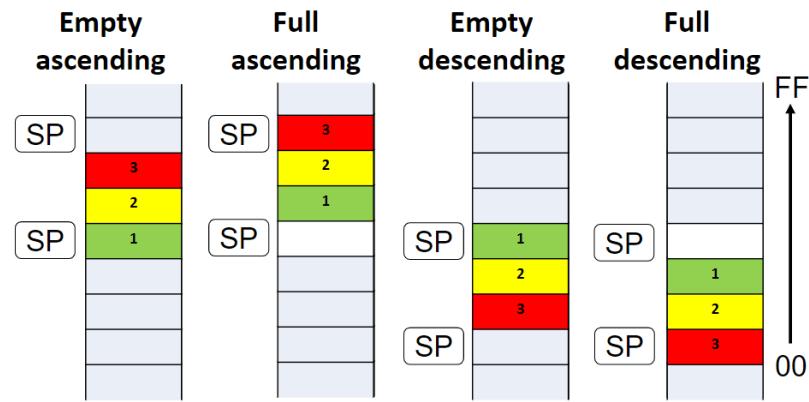
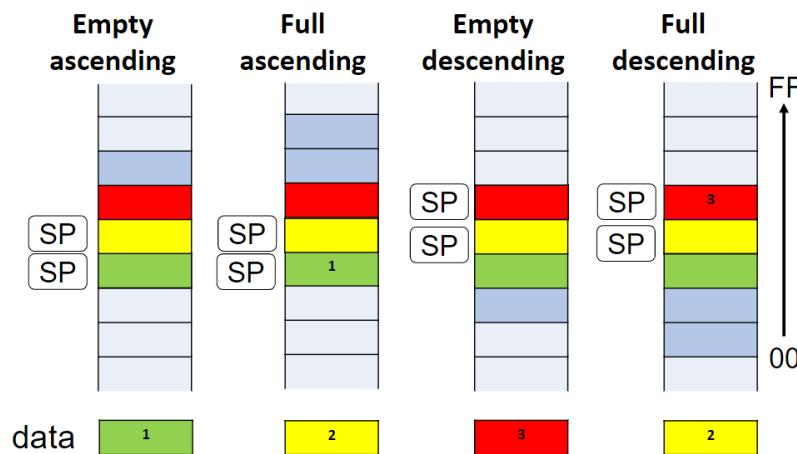
- **empty stack**: lo stack pointer punta all'entry dove il nuovo dato dovrà essere inserito
- **full stack**: lo stack pointer punta all'ultimo elemento inserito

**Nota:** il nostro stack è *full descending*.

## 11.1 LDM e STM

Le istruzioni di **LDM** e **STM** sono usate per caricare e salvare più registri contemporaneamente.

```
1 LDM{xx}/STM{xx} <Rn>{!}, <regList>
```

**Figura 11.2:** Esempio dopo 3 PUSH**Figura 11.3:** Esempio dopo 1 POP

Dove i parametri sono i seguenti:

- $Rn$  è il base register
- $xx$  specifica il metodo di indirizzamento, come e quando  $Rn$  viene aggiornato durante l'istruzione
  - con !:  $Rn$  viene settato al nuovo valore
  - senza !:  $Rn$  viene impostato al valore iniziale
- $regList$ : lista di registri

Una lista di registri possono essere:

- **consecutivi**: indicati separati dal valore iniziale e finale mediante una dash
- non **consecutivi**: registri separati con una ,

Esempi:

```
1 {r0-r4, r10, LR}
```

Indicano  $r0, r1, r2, r3, r4, r10, r14$ .

**Importante:** SP può non apparire nella lista mentre PC può essere presente solo con LDM e solamente in assenza di LR.

### 11.1.1 Ordine

L'ordine con cui vengono scritti i registri non è importante, questi sono automaticamente ordinati in ordine crescente:

- il registro più basso è salvato/caricato dal più basso indirizzo
- il registro più alto è salvato/caricato dal più alto indirizzo

Esempio:  $\{r8, r1, r3-r5, r14\}$  indica  $r1, r3, r4, r5, r8, r14$ .

### 11.1.2 Metodi di indirizzamento

I metodi di indirizzamento possibili sono due:

- IA: increment after (default)
  1. L'indirizzo a cui eseguire l'accesso è indicato nel base register
  2. Il base register è incrementato di 1 word (4 byte)

3. Se non ci sono registri nella lista, torna al punto uno
- **DB:** decrement before
  1. Il base register è decrementato di 1 word (4 byte)
  2. L'indirizzo a cui eseguire l'accesso in memoria è indicato nel base register
  3. Se non ci sono registri nella lista, torna al punto 1

### 11.1.3 PUSH e POP

Le istruzioni di PUSH e POP facilitano l'utilizzo di un full descending stack:

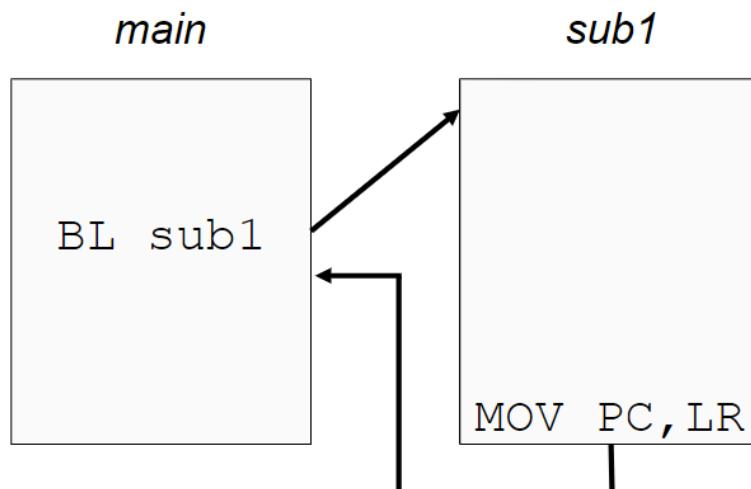
- **PUSH <regList>** ha il medesimo significato di **STMDB SP!, <regList>**
- **POP <regList>** ha il medesimo significato di **LDMIA SP!, <regList>**

## 11.2 Subroutines

Una subroutine viene chiamata con **BL <label>** e **BLX <Rn>**, in particolare viene scritto l'indirizzo della prossima istruzione in LR e il valore di label o Rn in PC.

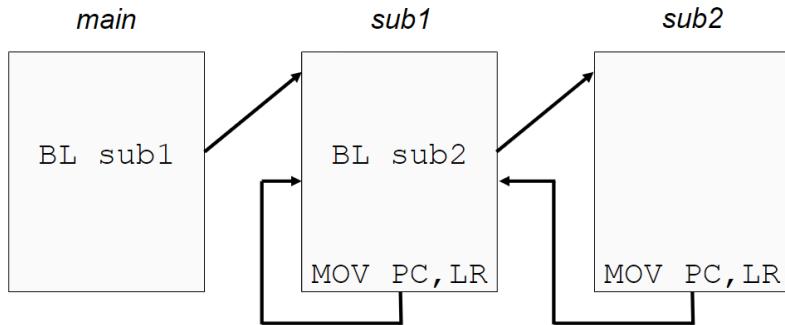
Una procedura **reentrant** finisce con un branch all'indirizzo salvato in **LR**.

E' possibile, facoltativamente, iniziare e terminare una subroutine con le direttive **PRC/FUNCTION** e **ENDP/ENDFUNC**.



**Figura 11.4:** Chiamata a una subroutine

Questo funzionamento genera però problemi nel caso in cui siano presenti delle chiamate annidate, in quanto il valore in **LR** viene **sovrascritte** non rendendo possibile per sub1 tornarne al main.



### 11.2.1 Chiamate annidate a subroutine

Invece di cambiare il valore di **LR** quando avvengono delle chiamate annidate, la nuova chiamata potrebbe cambiare il valore utilizzato nel registro della procedura precedente.

Tutte le subroutine dovrebbero salvare **LR** e gli altri registri utilizzati come prima istruzione e ripristinarli come ultima istruzione:

```

1 PUSH {regList, LR}
2 // ...
3 POP {regList, PC}
  
```

### 11.2.2 Passaggio di parametri

E' possibile passare alle funzioni dei parametri, in particolare sono possibili tre approcci:

- tramite **registro**
- tramite **referenza**, ad esempio un indirizzo salvato in un registro
- tramite lo **stack**

#### 11.2.2.1 Referenza

```

1      MOV r0, #0x34
2      MOV r1, #0xA3
3      LDR r3, =mySpace
4      STMIA r3, {r0, r1}
5      BL sub2
6      LDR r2, [r3]
7      ; r2 contains the result
  
```

```

8
9 sub2 PROC
10    PUSH {r2, r4, r5, LR}
11    LDMIA r3, {r4, r5}
12    CMP r4, r5
13    SUBHS r2, r4, r5
14    SUBLO r2, r5, r4
15    STR r2, [r3]
16    POP {r2, r4, r5, PC}
17    ENDP

```

### 11.2.2.2 Stack

```

1 MOV r0, #0x34
2 MOV r1, #0xA3
3 PUSH {r0, r1, r2}
4 BL sub3
5 POP {r0, r1, r2}
6 ; r2 contains the result
7 ...
8 stop B stop
9 ...
10
11 sub3 PROC
12 PUSH {r6, r4, r5, LR}
13 LDR r4, [sp, #16]
14 LDR r5, [sp, #20]
15 CMP r4, r5
16 SUBHS r6, r4, r5
17 SUBLO r6, r5, r4
18 STR r6, [sp, #24]
19 POP {r6, r4, r5, PC}
20 ENDP

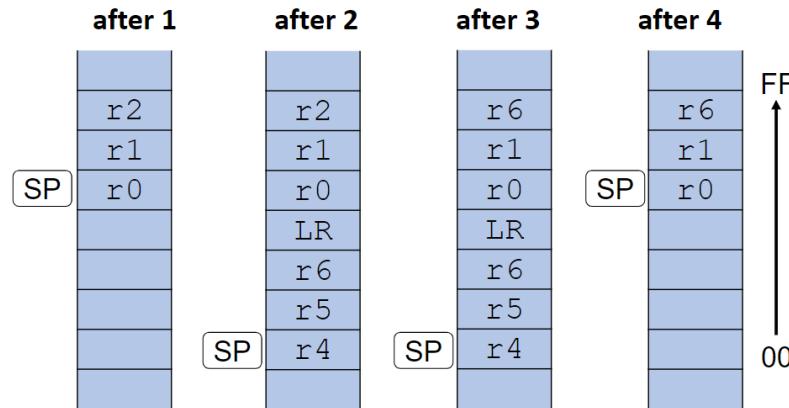
```

## 11.3 ABI

Un Application Binary Interface è un interfaccia che si pone tra due program binary modules (spesso una libreria e un programma eseguito da un utente).

Un aspetto comune di un ABI sono le **calling convention**, che determinano come i dati sono trasmessi in input oppure letti in output dalle *computational routines*.

I primi 4 registri r0-r3 (a1-a4) sono utilizzati come argomenti per le subroutine e il valore di ritorno di da una funzione. Una subroutine deve sempre preservare il contenuto dei registri **r4-r8, r10, r11** e **SP**.

**Figura 11.5:** Elementi nello stack

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Annotations on the right side of the table:

- A callout bubble points to r9 with the text: "Can be freely used to hold local variables".
- A callout bubble points to the bottom row with the text: "If there are more than 4 formal arguments, they have to be saved in the stack".

**Figura 11.6:** Calling conventions

Lo standard base proeede du passare gli arcomenti nei core registers (r0-r3) e nello stack. Per le subroutine che richiedono un piccolo numero di parametri, solo i registri sono utilizzati, riducendo in modo significativo l'overhead delle chiamate.

Come già detto lo stack è *full-descending*, con il current extent dello stack mantenuto nel registro **SP** ([R13](#)).

E' possibile creare delle variabili locali nello stack nello stesso modo in cui salviamo i dati, semplicemente sottraendo il numero di byte richiesti da ciascuna variabile dallo *stack pointer*.

## 12 Supervisor Calls (SVC)

I termini di *eccezioni* e *interruzioni* sono spesso confusi:

- **eccezione:** solitamente fa riferimento a un evento interno della CPU come un floating point overflow, MMU fault, trap (*soltanente quindi fa riferimento al sw*).
- **Interrupt:** fa riferimento a un evento esterno di U/O come I/O device request e reset (*soltanente quindi fa riferimento al hw*).

Nelle architetture arm le istruzioni ASM che rilasciano interruzioni software sono le **SVC**.

Non abbiamo in ARM istruzioni per la gestione di numeri reali.

- Reset
- NMI
- Faults
  - Hard Fault
  - Memory Management
  - Bus Fault
  - Usage Fault
- SVCall
- Debug Monitor
- PendSV
- SysTick Interrupt
- External Interrupt

La interrupt vector Table (IVT) è una tabella che permette di determinare quali procedure gestiscono ciascuna eccezione.

Una **Undefined Instruction** (UDF) è una istruzione non definita, che potrebbe essere voluto.

**SysTick** è una eccezione che viene rilasciata dal timer di sistema quando viene raggiunto lo zero. In un sistema operativo, il processore può utilizzare queste eccezioni come system tick.

Exception Type	Index	Vector Address
(Top of Stack)	0	0x00000000
Reset	1	0x00000004
NMI	2	0x00000008
Hard fault	3	0x0000000C
Memory management fault	4	0x00000010
Bus fault	5	0x00000014
Usage fault	6	0x00000018
SVcall	11	0x0000002C
Debug monitor	12	0x00000030
PendSV	14	0x00000038
SysTick	15	0x0000003C
Interrupts	≥16	≥0x00000040

**Figura 12.1:** IVT

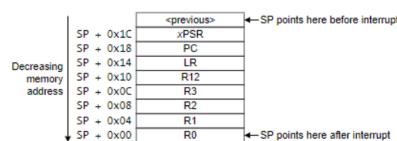
**Attenzione:** E' necessario sapere bene per l'esame come funziona SVC.

## 12.1 Stato delle eccezioni

Le eccezioni possono trovarsi in 3 differenti stati:

- **inattiva:** l'eccezione non è attiva o pendente
- **attiva:** l'eccezione è attualmente in servizio dal processore ma non è stata completata
- **pending:** l'eccezione è in attesa di essere servita dal processore

Quando una eccezione viene eseguita, il processore salva le informazioni nello stack corrente. Questa operazione è denominata **stacking** e la struttura di 8 parole è chiamata **stack frame**.

**Figura 12.2:** Stack Frame

Gli handler di default sono dichiarati come *weak symbols* per consentire all'application writer di installare i propri handler semplicemente implementando una funzione con il nome corretto. Se avviene una interruzione di cui non è stato definito un handler dall'application writer allora viene eseguito quello di default. Il default interrupt handler sono tipicamente implementati come loop infiniti. Se una funzione termina con un handler di default, è prima necessario determinare quale interrupt è in esecuzione.

```
SVC_Handler PROC
EXPORT SVC_Handler [WEAK]
B .
ENDP
```

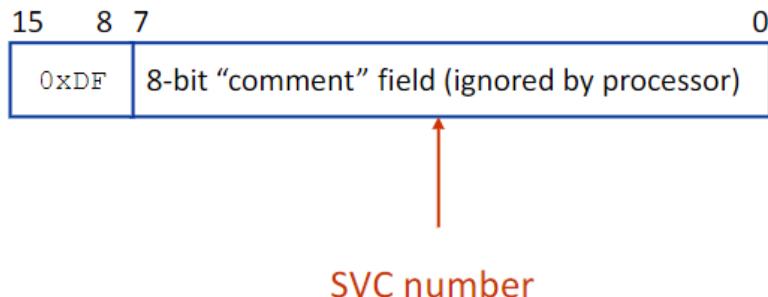
## 12.2 Supervisor Calls Syntax

Le supervisor calls sono normalmente utilizzate per fare richieste di operazioni privilegiato o accedere a risorse di sistema da un sistema operativo. Come gli ARM cores precedenti è presente una istruzione **SVC** (*Formalmente SWI*) che genera una supervisor call.

La chiamata viene fatta nel seguente modo:

```
1 {label} SVC immediate
```

L'istruzione SVC ha un numero al suo interno denominato **SVC number**. Questo viene utilizzato per indicare chi sta chiamando la richiesta ed è un numero intero su 8 bit da 0 a 255.



**Figura 12.3:** SVC number

Sui core ARM precedenti dovevi estrarre il numero SVC dall'istruzione utilizzando l'indirizzo di ritorno nel collegamento register e gli altri argomenti SVC erano già disponibili da R0 a R3.

Sul Cortex-M3, il core salva i registri degli argomenti nello stack sull'iniziale voce di eccezione. Qualsiasi valore restituito deve essere restituito al chiamante mediante modifica i valori del registro impiantati. Per fare ciò, deve essere presente un breve pezzo di codice assembly implementato come inizio del gestore SVC. Per identificare in quale stack sono stati salvati i registri per estrarre il numero SVC dall'istruzione.

## 12.3 Exception return

Il processore salva un **EXC\_RETURN** value nel [LR](#) una volta che l'eccezione comincia. Questo meccanismo di eccezione si basa sul valore determinato quando il processore ha completato l'exception handler. I bit [31:4] di un sono sempre [0xFFFFFFF](#).

<b>EXC_RETURN</b>	<b>Description</b>
<a href="#">0xFFFFFFF1</a>	Return to Handler mode. Exception return gets state from the main stack. Execution uses MSP after return.
<a href="#">0xFFFFFFF9</a>	Return to Thread mode. Exception Return get state from the main stack. Execution uses MSP after return.
<a href="#">0xFFFFFFFFD</a>	Return to Thread mode. Exception return gets state from the process stack. Execution uses PSP after return.
All other values	Reserved.

**Figura 12.4:** Exception return

Quando il processore carica nel PC un valore che fa il match con il pattern di una *EXC\_RETURN* riconosce che l'operazione non è un normale branch e, invece, che l'eccezione è stata completata. Per questo motivo, se se comincia ritorna una sequenza.

I bit [3:0] Del *ECX\_RETURN* indicano il modo in cui il processore deve ritornare al chiamante.

## 12.4 Processor Specific Configuration

L'istruzione che segue consente di abilitare l'aggiornamento di registri per utilizzo spaziale quando si ha un livello privilegiato:

```
1 MSR {cond} spec_reg, Rn
```

I valori sono i seguenti:

- **cond:** è un condition code opzionale
- **Rn:** specifica il registro sorgente

- **spec\_reg**: è uno dei seguenti registri PSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI\_MAX, FAULTMASK, CONTROL

## 12.5 Processor Modes

Esistono due modalità operative:

- **thread mode**: on reset o dopo una eccezione
- **handler mode**: quando avviene una eccezione

Esistono anche due livelli di accesso:

- **user level**: accesso limitato alle risorse
- **privileged level**: accesso a tutte le risorse

L'handler mode è sempre privilegiato.

## 12.6 Control Register

Il **control register** utilizza i seguenti bit:

- **CONTROL[2]**, solo su cortex-M4 e cortex-M7, se vale 0 allora FPU non è attiva altrimenti si
- **CONTROL[1]**:
  - se vale 0 in handler mode allora MSP è selezionato (non sono possibili alternate stack per l'handler mode).
  - Se vale 0 in thread mode allora il default stack pointer **MSB** verrà utilizzato
  - se vale 1 in thread mode allora l'alternate stack pointer **PSP** verrà utilizzato
- **CONTROL[0]**: non presente in cortex-M0, se vale 0 allora il processore è in thread mode in uno stato privilegiato, se vale 1 allora è in handler mode e in stato utente

A tempo di reset dopo le inizializzazioni è possibile impostare il processore in *user mode*.

```
1 MOV R0, #3
2 MSR CONTROL, R0
```

Questa istruzione porta il sistema in uno stato non privilegiato e thread mode, utilizzando il Process Stack Pointer (PSP). A causa dell'ingresso in una procedura di handling il sistema si sta muovendo a uno stato privilegiato e in handler mode, utilizzando il master stack pointer (MSP).

Un esempio:

**STACK segment**

```
1 Stack_Size EQU      0x00000200
2 AREA      STACK, NOINIT, READWRITE, ALIGN=3
3 SPACE     Stack_Size/2
4 Stack_Process SPACE   Stack_Size/2
5 __initial_sp
```

**CALLER**

```
1 MOV R0, #3
2 MSR CONTROL, R0
3 LDR SP, =Stack_Process
4 SVC 0x10
```

**HANDLER**

```
1 STMFD SP!, {R0-R12, LR}
2 MRS R1, PSP
3 LDR R0, [R1, #24]
4 LDR R0, [R0,#-4]
5 BIC R0, #0xFF000000
6 LSR R0, #16
7 LDMFD SP!, {R0-R12, LR}
8 BX LR
```

# 13 Interrupt controller

Per gestire le periferiche sono possibili due approcci:

- polling
- interrupt

Le categorie di eventi di sistema possono essere:

- eventi poco frequenti ma importanti
- I/O synchronization
- periodic interrupts
- data acquisition samples ADC

## 13.1 Polling

Il polling è un approccio in cui il processore controlla periodicamente lo stato di un dispositivo. Il polling è un approccio semplice ma inefficiente. Il polling è un approccio in cui il processore controlla periodicamente lo stato di un dispositivo. Il polling è un approccio semplice ma inefficiente.

Per determinare se è disponibile o meno si controlla lo status register (*best practice*) oppure verificando i data registers.

Spesso viene implementato direttamente in software mediante l'utilizzo di un ciclo che esegue una sequenza di check in modo più o meno frequente.

Le caratteristiche principali sono:

- la maggior parte del tempo è spesa in un ciclo software (inefficiente energetica)
- facili da implementare
- alta latenza
- difficile gestione delle richieste annidate (low performance)

## 13.2 Interrupt

Le periferiche possono comunicare direttamente con la CPU mediante le interruzioni, implementate in hardware. In questo modo è possibile entrare in uno stato di sospensione (idle) e il sistema si *risveglia* quando un evento di sistema si verifica.

Quando viene ricevuta una richiesta, la cpu ha bisogno di riconoscere la sorgente in modo da eseguire il corretto handler.

L'architettura attualmente presa in esame prevede l'implementazione di una gestione vettorizzata delle interruzioni mediante l'utilizzo della **IVT** (Interrupt Vector Table).

La CPU collabora con i dispositivi esterni mediante l'**interrupt controller**.

Per configurare la modalità interrupt è necessario a tempo di boot inizializzare le strutture dati come counters, pointers ed eventualmente specificare un flag che consente di abilitare le interrupt (semafori). Inoltre, è necessario configurare l'interrupt controller abilitando le sorgenti e impostando la priorità di ogni sorgente. Questo dovrà essere fatto in ogni routine di servizio per le interruzioni.

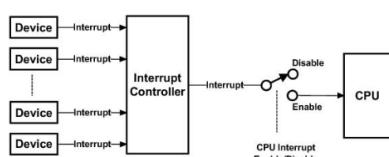
Anche a runtime è necessario un acknowledge pulendo i flag che indicano le interruzioni attive, che può essere effettuato in parti differenti della routine.

E' necessario mantenere il contenuto di R4-R8, R10-R11 (ABI APAPCS) e comunicando attraverso variabili globali condivise.

## 13.3 Controller

Un interrupt controller è un dispositivo che viene utilizzato per combinare diverse sorgenti di interruzioni in una o più linee della cpu, consentendo di gestire più livelli di priorità che possono essere assegnati agli interrupt outputs.

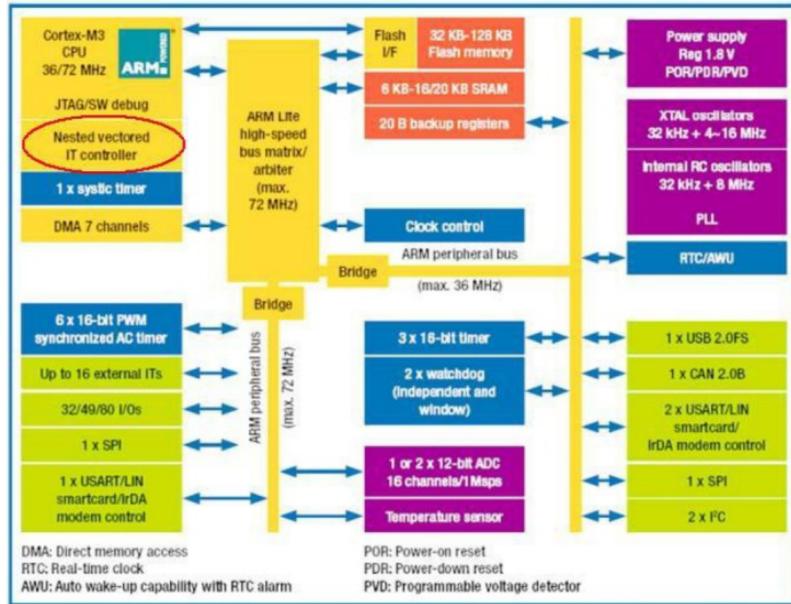
Consente la gestione dei segnali di interruzione ricevuti da più dispositivi combinandoli in un solo interrupt output.



### 13.3.1 NVIC

L'interrupt vettoriale annidato Il controller (NVIC) è parte integrante parte del Cortex-M3.

- Lo stretto accoppiamento con la CPU consente interruzioni basse latente ed efficiente elaborazione del ritardo in arrivo delle interruzioni
- gestisce 35 possibili interruzioni esterne



**Figura 13.1:** NVIC

### 13.3.1.1 Priorità - NVIC\_SetPriority

Per impostare la priorità abbiamo a disposizione la funzione `NVIC_SetPriority()`.

```
1 void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority);
```

Un valore minore di priorità significa una priorità superiore.



# 14 Timer

Un sistema potrebbe richiedere di aspettare per un periodo di tempo o eseguire operazioni a periodi regolari. Queste funzionalità potrebbero richiedere il supporto di periferiche denominate **timer**.

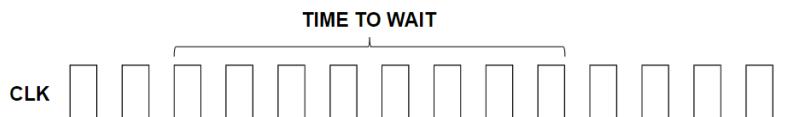
Il timer ha dunque lo scopo di consentire al programmatore di sincronizzare, basandosi sul conteggio, il sistema.

Soltamente quando un procedura di conteggio raggiunge la fine, il sistema regisce in qualche modo:

- tipicamente mediante un interrupt handler
- potrebbe entrare in una situazione di risparmio energetico da parte della cpu-

## 14.1 Funzionamento

I timer sono dotati di un clock signal dedicato, attraverso il quale il timer incrementa il suo contatore. Hanno dei registri che possono essere programmati con un numero di clock cycle da contare.



**Figura 14.1:** Clock del timer

Un timer potrebbe seguire differenti modalità di funzionamento:

- **decreasing count:** si interrompe quando il contatore raggiunge 0
- **increasing count:** si interrompe quando il contatore raggiunge un valore massimo

$$\text{time}[s] = \text{count} * \text{Clock_Period}[s]$$

$$\text{count} = \text{time}[s] / \text{Clock_Period}[s]$$

$$\text{count} = \text{time}[s] * \text{frequency}[1/s]$$

Se il tempo di attesa è troppo alto, potrebbe non rientrare all'interno del timer del registro. Per risolvere tale problema vengono, a seconda dei casi, adoperate soluzioni sia hardware che software:

- hw: cascade of counter
- hw: prescaler
- sw: handler software count of hw events

## 14.2 LPC1768

Normalmente i SoC implementano più timer:

- standard timers: programmati dall'utente per implementare intervalli regolari e ritardi
- operating system timers: utilizzati dal sistema operativo per gestire i processi
- extra timers: danno al sistema operativo funzionalità specifiche come ripetitive interrupt timer e PWM

### 14.2.1 Standard timers

Lo standrd timer/counter è realizzato in modo da contare i cicli del pheripherical clock (PCLK) o di un clock esterno. Inoltre, può generare facoltativamente interruzioni o eseguire altre azioni per valori specifici del timer in base a 4 match registers.

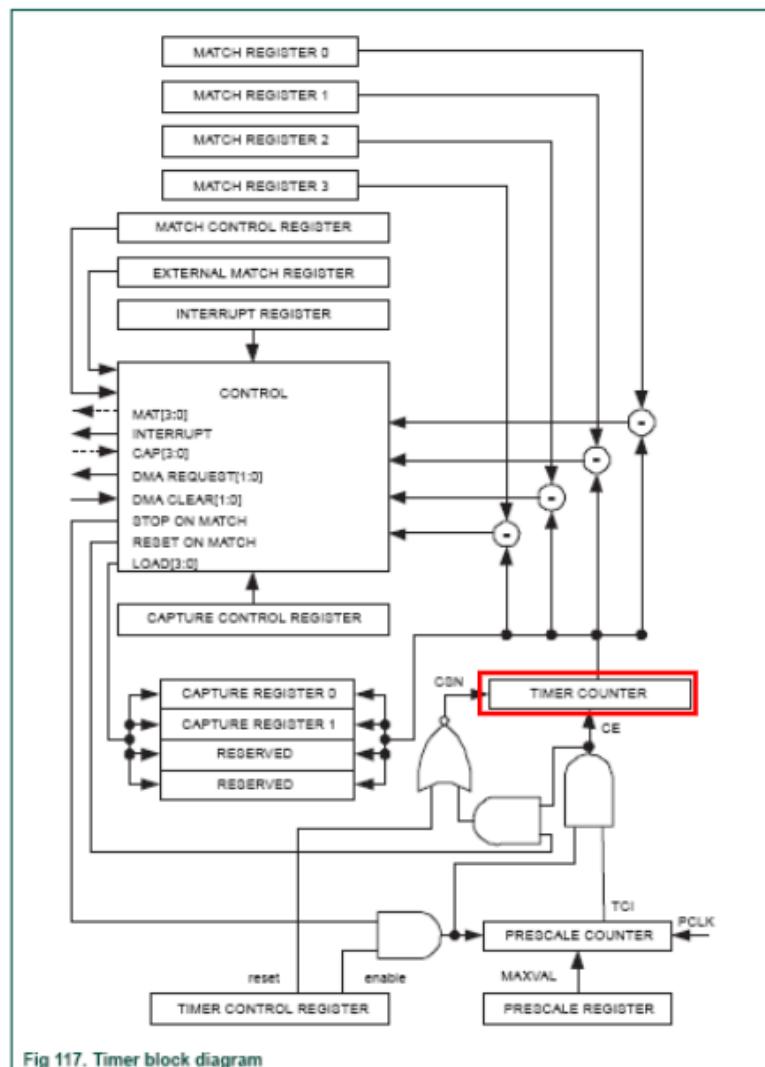
Prevede 4 capture inputs per trap the timer value quando avviene una transizione del segale, gene-rando opzionalmente una interruzione.

timer 0 e 1 sono di default.

#### 14.2.1.1 Match registers

Sono presenti **4** registri a **32 bit** che permettono:

- **operazioni continue** con un interrupt opzionale su match
- **fermare** il timer su match con un interrupt opzionale
- **resettere** il timer su match con un interrupt opzionale
- generazione di una **unique interrupt**, l'ISR deve capire quale dei 4 match registers ha generato l'interrupt (leggendo il registro IR).



**Figura 14.2:** Standard timers

#### 14.2.1.2 Capture registers

Una transizione può essere registrata da un pin configurato per caricare un dei capture registers. Con il valore del timer counter e opzionalmente genera un interrupt.

#### 14.2.1.3 External match output

Quando un match register (MR3:0) è uguale a un timer counter (TC) questo output può essere toggled, andare basso, andare alto o non fare niente.

### 14.3 Utilità

#### 14.3.1 Accensione

Per accendere il timer 2 è necessario:

- aprire il file `system_LPC17xx.c` contenuto nella cartella `lib_SoC_board`
- andare dalle tab in basso nel `configuration wizard`
- cercare la voce `PCTIM2` e metterla a 1

#### 14.3.2 Abilitazione

Per abilitare il timer 2 è necessario utilizzare la funzione `enable_timer(x);`, con `x` numero intero del timer da abilitare.

# 15 Clocking and Power control functions

## 15.1 Power control registers

Il **PCON** è un registro di controllo che consente di entrare nelle modalità di funzionamento energetico.

Il **PCONP** è il registro relativo all'alimentazione periferica, utile nel caso si voglia utilizzare il timer 2 e 3.

L'ingresso in modalità di consumo energetico ridotto inizia sempre con l'esecuzione di una WFI (wait for interrupt) o WFE (wait for Exception). Il cortex M3 supporta internamente due modalità di risparmio energetico: sleep e deep sleep. Invece, il power-down e il deep power-down sono selezionati dai bit del registro PCON.

Il registro PCON si occupa anche di gestire alcune modalità di risparmio energetico e altri controlli relativi all'alimentazione. Allo stesso modo, sono presenti dei flag che indicano quando si entra in una situazione di risparmio energetico.

Il bt PM1 e PM0 di PCON permettono di selezionare una modalità di risparmio energetico. Questi sono encodati in modo da garantire una compatibilità con i dispositivi precedenti che non supportavano sleep e power-down modes.

### 15.1.1 Sleep mode

Quando la **sleep mode** è attivata, il clock del core viene fermato e il **SMFLAG** bit in PCON viene settato. L'esecuzione delle istruzioni viene fermata fino a quando non avviene un reset o una interruzione (il wake up occorre quando qualsiasi interruzione occorre).

Le funzioni periferiche continuano a funzionare e potrebbero generare interruzioni che potrebbero risvegliare l'esecuzione. Lo sleep mode elmina il dynamic power utilizzato dallo stesso processore, la memoria e i relativi controlli oltre ai bus interni.

### 15.1.2 Deep sleep mode

Quando il chip entra in **deep sleep mode**, il clock del core viene fermato interrompendo l'oscillatore principale e il **DSFLAG** bit in PCON viene settato. L'IRC continua a eseguire e può essere configurato per guidare il watchdog timer, permettendo il watchdog di svegliare la cpu.

L'oscillatore a 32 kHz RTC oscillator non è fermato e il RTC potrebbe essere utilizzato come sorgente di risveglio mediante interruzione.

I PLL sono automaticamente spenti e disconnessi (CCLK e USBCLK vengono resettati a zero).

La memoria FLASH è lasciata in standby in modo da consentire un risveglio rapido.

Dal momento che tutte le operazioni dinamiche sui chip sono sospese, la deep sleep mode riduce il consumo energetico dei chip a un valore molto basso. Lo stato del processore e i registri, i registri periferici e la SRAM interna sono preservati e i livelli logici dei pin rimangono statici.

Il deep sleep mode può essere terminato da un reset o da una specifica interruzione in grado di funzionare senza l'utilizzo di clock (*Wake-up from Deep Sleep mode can be brought about by NMI or External Interrupts EINT0 through EINT3*).

### 15.1.3 Power-down mode and Deep Power-down

In power down mode avvengono le stesse cose del deep sleep mode ma con la differenza che anche la memoria flash viene posta spenta. Questo consente di risparmiare maggiore energia, ma richiede un tempo di attesa maggiore per il risveglio della memoria prima dell'esecuzione del codice o dell'accesso ai dati. L'ingresso in questo stato causa il set del bit PDFLAG in PCON.

In deep power-down mode l'alimentazione viene completamente spenta per il cheap ad esclusione del real-time clock, il reset pin, il WIC, il registro di backup RTC. L'ingresso causa il set del bit di **DPD-FLAG** in **PCON**.

## 15.2 Peripheral power control

Il power control periferico è gestito dal registro **PCONP**. Questo consente di spegnere singolarmente le periferiche che non sono necessarie per il funzionamento dell'applicazione, incrementando il risparmio energetico.

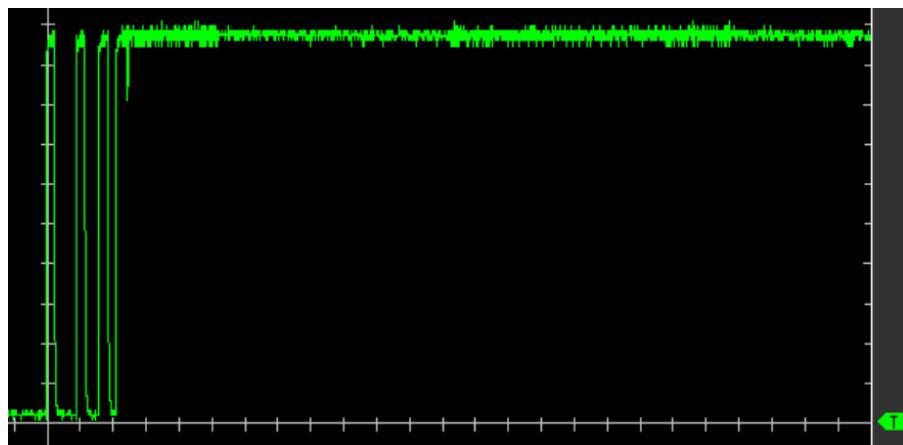
Questo risultato viene ottenuto spegnendo la sorgente del clock per determinate periferiche. Alcune funzioni periferiche non possono essere spente (watchdog timer, il pin connect block, system control block)

# 16 Switch bouncing

Quando un pulsante viene premuto oppure viene invertito un toggle switch, due parti metalliche si uniscono e apparentemente il contatto è immediato. Questo non è completamente corretto in quanto all'interno degli switch esistono parti che si muovono.

Quando un pulsante viene premuto, prima avviene il contatto con l'altra parte metallica in un piccolo intervallo di microsecondi. Successivamente esegue un ulteriore contatto leggermente più più lungo e così via, solo alla fine gli switch sono completamente chiusi. Lo switch sta dunque rimbalzando tra "in contatto" e "non in contatto".

Soltamente il SoC lavora più velocemente del bouncing e per tale motivo l'hardware pensa che il pulsante venga premuto più volte.



**Figura 16.1:** Bouncing

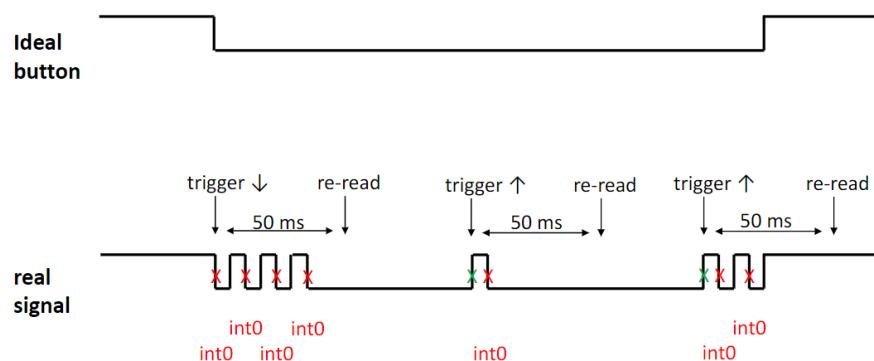
## 16.1 Soluzione software

UN modo comune per risolvere lo switch bouncing è di rileggere il valore del pin dopo 50ms dal primo bounce. Per i pulsanti si preferisce utilizzare le interruzioni (più efficiente energeticamente in quanto si può entrare in power down mode), se non è disponibile viene utilizzato il polling (un timer può essere utilizzato per svegliare il sistema a tempi regolari).

L'implementazione blocking delay non è consigliata, in particolare utilizzare for/while/do-while con blocchi vuoti è considerato profondamente errato.

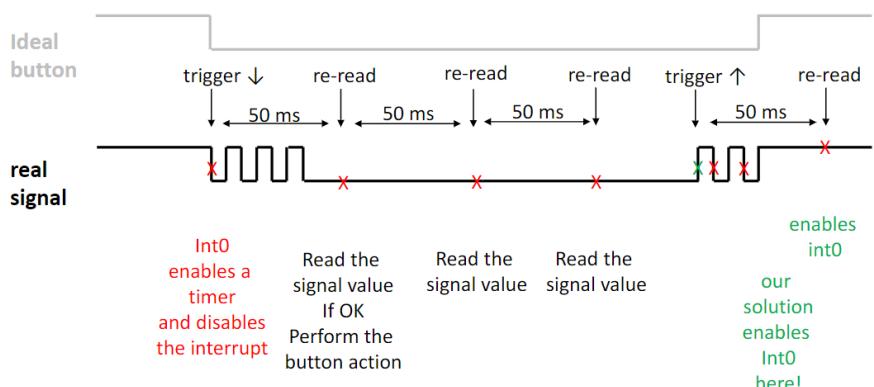
Se il pin di interrupt mode è settato, questo potrebbe non essere direttamente leggibile. Per tale motivo, per poter leggere il valore del pulsante è necessario disabilitare le interruzioni e accettare di leggere l'input value.

Quello che succede:



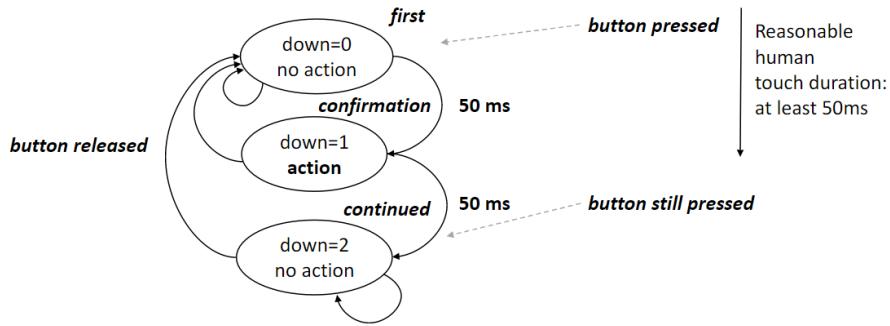
**Figura 16.2:** Soluzione visuale

Quello che vogliamo ottenere:



**Figura 16.3:** Soluzione

Quanto detto può essere schematizzato nella seguente macchina a stati:

**Figura 16.4:** Schema a stati finiti

## 16.2 Repetitive Interrupt Timer (RIT)

Il RIT è un timer che consente di generare interruzioni a specifici intervalli, senza utilizzare lo standard timer.

```

1 // RIT/IRQ_RIT.c
2 uint32_t init_RIT ( uint32_t RITInterval ){
3     LPC_SC->PCLKSEL1  &= ~(3<<26);
4     LPC_SC->PCLKSEL1 |= (1<<26); //RIT Clock = CCLK
5     ...
6     lib_RIT.c
7     RIT_cnt = 50ms * 100MHz
8     RIT_cnt = 5.000.000 = 0x4C4B40
9     ...
10 }
```

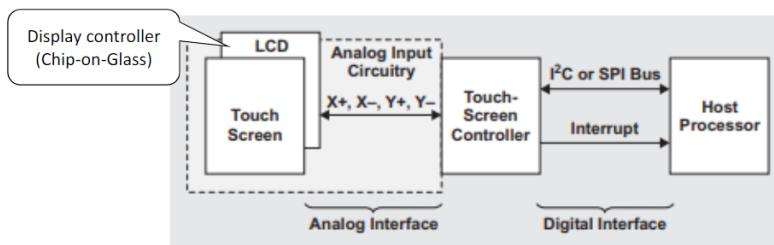
**Nota bene:** il RIT di default è spento.

Experiment switch bouncing with your board and try to mitigate Key bouncing: they must use the external interrupt functionalities Advanced -> Joystick: implement a «timer controlled polling strategy» also able to mitigate debouncing Quite Advanced -> can you manage the pressure of many buttons or the contemporary use of buttons and Joystick? Super-Advanced -> implement button and joystick debouncing by using the RIT only.



# 17 Touch Display

Esistono vari tipi di display, in particolare quella che andremo a vedere è un display dotato di touch-screen. Le due componenti sono in realtà indipendenti e vengono gestite da due interfacce differenti.



**Figura 17.1:** Interfacce

Le tipologie sono:

- LCD: liquid crystal display
- TFT LCD: thin film transistor liquid crystal display
- IPS LCD: in-plane switching liquid crystal display
- LED-backlit LCD: LCD con retroilluminazione a LED

Allo stesso modo anche le tecnologie relative al touchscreen sono diverse:

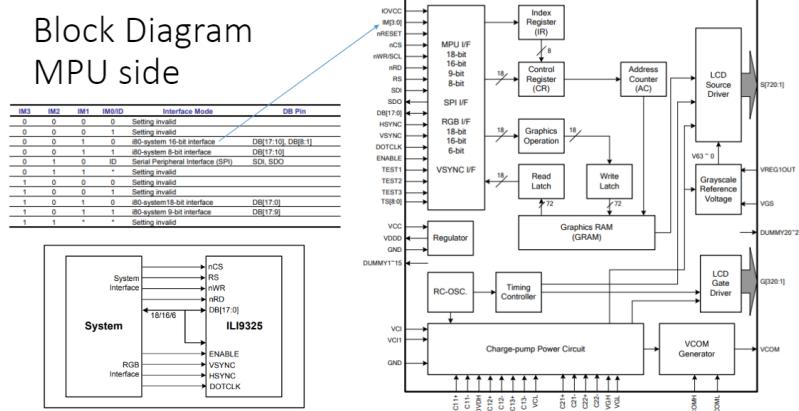
- resistivo
- capacitivo
- infrarossi
- surface acoustic wave
- near field imagining
- light pens

**Nota:** Lo schermo che viene utilizzato è capacitivo e di tipo TFT LCD.

## 17.1 Display controller - ILI9325

- Single chip solution for a TFT LCD display
- 240RGBx320-dot resolution capable with real 262,144 display color
- Incorporates 720-channel source driver and 320-channel gate driver
- Internal 172,800 bytes graphic RAM
- High-speed RAM burst write function
- System interfaces
- i80 system interface with 8-/ 9-/16-/18-bit bus width
- Serial Peripheral Interface (SPI)
- RGB interface with 6-/16-/18-bit bus width (VSYNC, HSYNC, DOTCLK, ENABLE, DB[17:0])
- VSYNC interface (System interface + VSYNC)

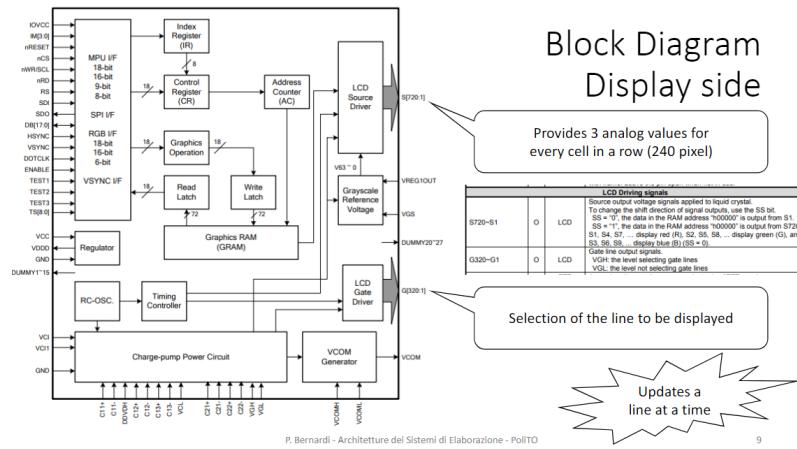
**Attenzione:** Non bisogna oltrepassare i 32kHz di frequenza di clock per il display.



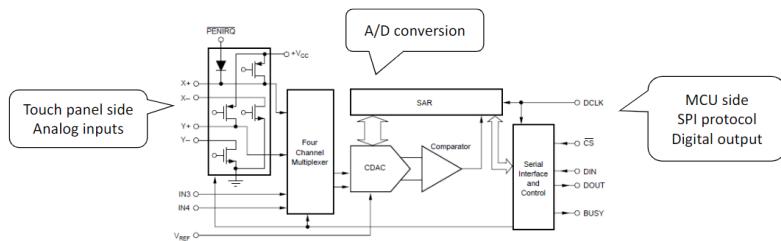
**Figura 17.2:** Block Diagram MPU side

## 17.2 Touch Screen Controller - ADS7843

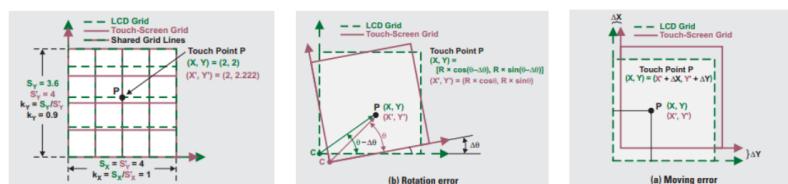
- 4-wire touch screen interface
- ratiometric conversion
- single supply: 2.7v to 5v
- up to 125kHz conversion rate
- serial interface
- programmable 8- or 12-bit resolution

**Figura 17.3:** Block Diagram Display Side

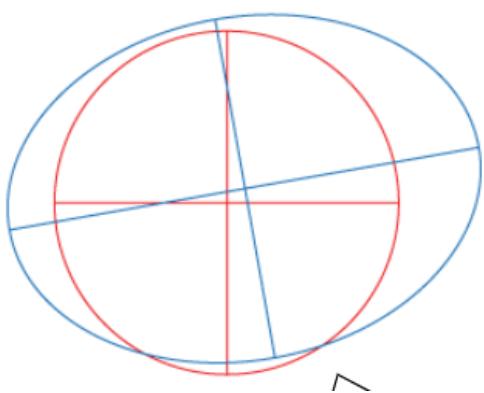
- 2 auxiliary analog inputs
- full power-down control

**Figura 17.4:** Diagramma a blocchi

Il display e il touch non sono perfettamente allineati e per tale motivo alla prima accensione deve essere calibrato.

**Figura 17.5:** Errore di calibrazione

Quando il dito viene tracciato dal display questo appare come un cerchio, ma in realtà il touch panel potrebbe salvare le coordinate come un ellisse. Tale problema può essere causato da alcune trasformazioni come: traslazione, rotazione e scalatura.



**Figura 17.6:** Calibratura