

# ADE - Architettura degli elaboratori

Elia Ronchetti

Marzo 2022

# Indice

<b>1 Introduzione e Argomenti</b>	<b>6</b>
1.1 Rappresentazione dell'informazione . . . . .	6
1.2 Circuiti logici . . . . .	6
1.3 Instruction Set Architecture (ISA) . . . . .	6
1.4 Linguaggio Assembly . . . . .	7
1.5 Datapath . . . . .	7
1.6 Gestione delle eccezioni . . . . .	7
1.7 Tecniche di gestione dell'ingresso/uscita . . . . .	7
1.8 Gerarchie di memoria: cache . . . . .	7
<b>2 Sistemi numerici</b>	<b>8</b>
2.1 Conversione tra basi . . . . .	9
2.2 Operazioni aritmetiche e Overflow . . . . .	9
2.3 Operazioni con segno . . . . .	9
2.3.1 Modulo e segno . . . . .	9
2.3.2 Operazioni aritmetiche con MS . . . . .	10
2.3.3 Complemento a 1 (CA1) . . . . .	10
2.3.4 Complemento a 2 (CA2) . . . . .	11
2.3.5 Operazioni aritmetiche con CA2 . . . . .	11
2.4 Riassunto Overflow . . . . .	12
2.5 Operazione di Shift . . . . .	13
2.6 Rappresentazione Eccesso 128 . . . . .	13
2.7 Rappresentazione con la virgola . . . . .	14
2.7.1 Virgola fissa . . . . .	14
2.7.2 Virgola mobile . . . . .	14
2.7.3 Errore assoluto ed errore relativo . . . . .	15
2.7.4 Codifiche . . . . .	15
<b>3 Circuiti logici</b>	<b>16</b>
3.1 Porte Logiche . . . . .	16
3.1.1 AND * . . . . .	16

3.1.2	OR + . . . . .	17
3.1.3	NOT . . . . .	17
3.1.4	Porte con più di due ingressi . . . . .	17
3.2	Porte logiche derivate . . . . .	18
<b>4</b>	<b>Circuiti combinatori</b>	<b>19</b>
4.1	Decoder . . . . .	19
4.2	Multiplexor . . . . .	19
4.2.1	Relazione importante per gli es . . . . .	20
4.3	Logiche a due livelli e PLA . . . . .	20
4.4	ROM (Read Only Memory) . . . . .	21
4.5	ALU . . . . .	22
4.5.1	Operazioni aritmetiche . . . . .	22
4.6	Operazioni di confronto - SLT e BEQ . . . . .	24
4.6.1	SLT . . . . .	24
4.6.2	BEQ - Branch on equal . . . . .	24
4.6.3	Overflow detection . . . . .	24
4.6.4	Carry Lookahead . . . . .	25
4.7	Note per gli esercizi . . . . .	25
4.7.1	Conversione da tabella a circuito . . . . .	25
4.7.2	Calcolo output circuito . . . . .	26
<b>5</b>	<b>Circuiti sequenziali</b>	<b>27</b>
5.1	Clock . . . . .	27
5.2	S-R Latch . . . . .	28
5.3	D Latch . . . . .	29
5.3.1	Edge-triggered clocking . . . . .	30
5.4	D Flip Flop . . . . .	31
5.5	Differenze tra Latch e Flip-Flop . . . . .	32
5.6	Datapath e Register File . . . . .	32
5.7	Memoria . . . . .	34
5.8	Macchine a stati finiti . . . . .	40
<b>6</b>	<b>Instruction Set Architecture (ISA)</b>	<b>42</b>
6.1	Filosofie di progetto della CPU . . . . .	42
6.1.1	RISC . . . . .	42
6.1.2	CISC . . . . .	43
6.2	Istruzioni MIPS . . . . .	43
6.2.1	R-Type . . . . .	43
6.2.2	I-type . . . . .	44
6.2.3	J-type . . . . .	47

6.3	Procedure e convenzioni registri . . . . .	48
6.3.1	Passaggio parametri - convenzioni base registri . . . . .	49
6.3.2	Convenzioni sui registri t e s . . . . .	50
<b>7</b>	<b>ASM - Catena Programmatica + Linguaggio Assembly</b>	<b>52</b>
7.1	Linguaggio Assembler - Vantaggi e Svantaggi . . . . .	52
7.2	Compiler, Assembler, Linker . . . . .	53
7.2.1	Compilatore . . . . .	53
7.2.2	Assembler . . . . .	54
7.2.3	Il processo di assemblaggio . . . . .	54
7.2.4	Linker (Link editor) . . . . .	55
7.2.5	Loader . . . . .	56
7.2.6	Pseudoistruzioni . . . . .	56
7.3	Uso dello stack . . . . .	57
7.3.1	Operazioni procedura chiamante . . . . .	58
7.3.2	Operazioni procedura chiamata . . . . .	58
7.4	Syscall . . . . .	58
7.5	Domande Esame . . . . .	59
7.5.1	Descrivere i passi per la gestione delle etichette irrisolte e il ruolo del linker in questa fase . . . . .	59
<b>8</b>	<b>Datapath</b>	<b>62</b>
8.1	Realizzare un datapath . . . . .	62
8.2	Passi per l'esecuzione di una istruzione - Fetch Decode Execute	63
8.3	Implementazione Fetch . . . . .	64
8.4	Datapath multi-ciclo . . . . .	68
8.5	Dettaglio esecuzione istruzioni MIPS . . . . .	69
8.5.1	Fetch e Decode . . . . .	70
8.5.2	3. Execution, memory address computation, or branch completion . . . . .	71
8.5.3	4. Memory access or R-type instruction completion step	72
8.5.4	Riassunto Esecuzione Istruzioni . . . . .	73
8.6	Datapath: Automa per il controllo . . . . .	73
8.7	Riassumendo il Datapath . . . . .	79
8.7.1	Funzioni dei registri . . . . .	80
8.7.2	Ruolo di ciascuno dei multiplexer . . . . .	80
8.8	Eccezioni . . . . .	83
8.8.1	Gestione di eccezioni e interruzioni . . . . .	84
8.8.2	Datpath e FSM con gestione eccezioni . . . . .	87

<i>INDICE</i>	5
<b>9 Gestione Input-Output</b>	<b>88</b>
9.1 Bus di Sistema . . . . .	88
9.2 Periferiche . . . . .	89
9.2.1 Passi di I/O . . . . .	90
9.2.2 Tecniche di gestione I/O . . . . .	90
<b>10 Gerarchie di memoria e cache</b>	<b>95</b>
10.1 Principio di località . . . . .	96
10.2 Definizioni . . . . .	96
10.3 Cache . . . . .	97
10.3.1 Tipi di cache . . . . .	98
10.3.2 Campi della cache . . . . .	100
10.3.3 Confronto tra le tecniche di indirizzamento . . . . .	102
10.4 Gestione della cache . . . . .	103
10.4.1 Algoritmi per la sostituzione di blocchi . . . . .	103
10.4.2 Gestione dei miss in lettura e Accesso in scrittura . . . . .	104
10.4.3 Write miss . . . . .	105
10.4.4 Riassumendo . . . . .	106

# Capitolo 1

## Introduzione e Argomenti

### 1.1 Rappresentazione dell'informazione

- Sistemi numerici
- Rappresentazione dei numeri interi con e senza segno
- Rappresentazione dei numeri in virgola fissa e mobile
- Rappresentazione dell'informatica non numerica

### 1.2 Circuiti logici

- Reti combinatorie
- Reti sequenziali e FSM (Finite State Machine)
- Rassegna di circuiti notevoli (decoder, multiplexor, register file, ALU, etc.)

### 1.3 Instruction Set Architecture (ISA)

- schema di von Neumann
- CPU, registri, ALU e memoria
- Ciclo fondamentale di esecuzione di una istruzione (fetch/decode/execute)
- Tipi e formati di istruzioni MIPS32

- Modalità di indirizzamento

## 1.4 Linguaggio Assembly

- Formato simbolico delle istruzioni
- Catena di programmazione (compilatore, assembler, linker, loader, debugger, etc.)
- Pseudo-istruzioni e direttive dell’assemblatore
- Scrittura di semplici programmi assembly
- Convenzioni programmatiche (memoria, nomi dei registri, etc.)

## 1.5 Datapath

- Percorsi dei dati per le diverse classi di istruzioni
- Controllo del percorso dei dati con FSM

## 1.6 Gestione delle eccezioni

- Tassonomia di eccezioni in terminologia MIPS32
- Modifiche alla FSM di controllo, registro Cause

## 1.7 Tecniche di gestione dell’ingresso/uscita

- Controllo di programma
- Interruzione di programma
- Accesso diretto alla memoria

## 1.8 Gerarchie di memoria: cache

- Cache a mappature diretta
- Cache fully associative
- Cache n-way set associative

# Capitolo 2

## Sistemi numerici

Con il termine bit definiamo l'unità di misura dell'informazione. Un bit può assumere solo il valore di 0 o 1.

Combinando tra loro più bit si ottengono strutture più complesse, per esempio:

- byte, 8 bit
- nybble, 4 bit
- word, 32 bit = 4 Byte

Una rappresentazione è un modo per descrivere un'entità

Il sistema numerico decimale:

- usa 10 cifre
- è un sistema posizionale: ogni cifra assume un valore diverso a seconda della posizione che occupa

Confronto tra Basi

<b>0<sub>hex</sub></b> = <b>0<sub>dec</sub></b> = <b>0<sub>oct</sub></b>	0 0 0 0
<b>1<sub>hex</sub></b> = <b>1<sub>dec</sub></b> = <b>1<sub>oct</sub></b>	0 0 0 1
<b>2<sub>hex</sub></b> = <b>2<sub>dec</sub></b> = <b>2<sub>oct</sub></b>	0 0 1 0
<b>3<sub>hex</sub></b> = <b>3<sub>dec</sub></b> = <b>3<sub>oct</sub></b>	0 0 1 1
<b>4<sub>hex</sub></b> = <b>4<sub>dec</sub></b> = <b>4<sub>oct</sub></b>	0 1 0 0
<b>5<sub>hex</sub></b> = <b>5<sub>dec</sub></b> = <b>5<sub>oct</sub></b>	0 1 0 1
<b>6<sub>hex</sub></b> = <b>6<sub>dec</sub></b> = <b>6<sub>oct</sub></b>	0 1 1 0
<b>7<sub>hex</sub></b> = <b>7<sub>dec</sub></b> = <b>7<sub>oct</sub></b>	0 1 1 1
<b>8<sub>hex</sub></b> = <b>8<sub>dec</sub></b> = <b>10<sub>oct</sub></b>	1 0 0 0
<b>9<sub>hex</sub></b> = <b>9<sub>dec</sub></b> = <b>11<sub>oct</sub></b>	1 0 0 1
<b>A<sub>hex</sub></b> = <b>10<sub>dec</sub></b> = <b>12<sub>oct</sub></b>	1 0 1 0
<b>B<sub>hex</sub></b> = <b>11<sub>dec</sub></b> = <b>13<sub>oct</sub></b>	1 0 1 1
<b>C<sub>hex</sub></b> = <b>12<sub>dec</sub></b> = <b>14<sub>oct</sub></b>	1 1 0 0
<b>D<sub>hex</sub></b> = <b>13<sub>dec</sub></b> = <b>15<sub>oct</sub></b>	1 1 0 1
<b>E<sub>hex</sub></b> = <b>14<sub>dec</sub></b> = <b>16<sub>oct</sub></b>	1 1 1 0
<b>F<sub>hex</sub></b> = <b>15<sub>dec</sub></b> = <b>17<sub>oct</sub></b>	1 1 1 1

## 2.1 Conversione tra basi

Svolti esercizi di conversione tra basi

## 2.2 Operazioni aritmetiche e Overflow

- Addizioni e sottrazioni
- Svolti esercizi con Overflow, bit di carry

L'overflow si verifica quando il risultato non può essere rappresentato con il numero di bit, che ho a disposizione e quindi ottengo un risultato sbagliato.

## 2.3 Operazioni con segno

Ci sono diverse modalità per rappresentare il segno in base 2

### 2.3.1 Modulo e segno

La rappresentazione modulo e segno divide i bit di rappresentazione in 2, nel caso di 8 bit, 7 sono utilizzati per rappresentare il valore assoluto e il bit

più significativo (MSB - Most significant bit), quello a sinistra, rappresenta il segno, 0 positivo, 1 negativo

$$1|0000100 \quad (2.1)$$

Questa rappresentazione è semplice e con n bit totali, si possono rappresentare i numeri interi nell'intervallo, ma ha alcuni problemi

- Esistono 2 rappresentazioni diverse per lo 0
- Un bit tra tutti i bit disponibili viene speso per il segno e questo è uno spreco, riduce inoltre la capacità di rappresentazione

### 2.3.2 Operazioni aritmetiche con MS

Possiamo avere overflow solo quando:

- si sommano due operandi con segno concorde
- si sottraggono due operandi con segno discorde

L'overflow si verifica quando c'è un riporto della cifra più significativa del modulo, cioè non si è nella condizione di rappresentare il risultato ottenuto.

### 2.3.3 Complemento a 1 (CA1)

È un'altra modalità di rappresentazione dei numeri interi con segno. Come indica il nome stesso, questo metodo si basa sull'operazione di complemento

**Complemento** è l'operazione che associa ad un bit (o ad ogni sequenza di bit) il suo opposto, cioè il valore ottento sostituendo tutti gli 1 con 0 e tutti gli 0 con 1

**Esecuzione** è semplice e diretta

1. Se il numero da codificare è positivo lo si converte in binario con il metodo tradizionale
2. Se il numero è negativo basta convertire in binario il suo modulo e quindi eseguire l'operazione di complemento sul numero appena convertito

**Problema** ancora doppia rappresentazione dello 0

### 2.3.4 Complemento a 2 (CA2)

Anche qui il MSB è 0 se  $x$  è positivo e MSB = 1 se  $x$  è negativo

#### Esecuzione

1. Se il numero  $X$  è positivo esso rimane invariato
2. Se il numero  $X$  è negativo
  - 2.1 Si effettua il complemento a 1 (CA1) sul valore da codificare
  - 2.2 Si somma +1 al risultato ottenuto con CA1

Così elimino la doppia rappresentazione dello zero. I valori negativi hanno MSB = 1.

### 3 Metodi per il calcolo di CA2

1. Definizione di complemento alla base
2. Per calcolare CA2 si calcola CA1 e si somma 1
3. Regola Pratica
  - 3.1 Si parte da destra, si trascrivono tutti gli 0 fino ad incontrare il primo 1 e si trascrive anch'esso
  - 3.2 Si complementano a 1 ( $0 \rightarrow 1$  e  $1 \rightarrow 0$ ) tutti i bit restanti

Distinzione tra Operazione CA2 e Rappresentazione CA2

- La rappresentazione - come sono organizzati i bit
- Il calcolo - procedura di trasformazione

### 2.3.5 Operazioni aritmetiche con CA2

#### Somma

1. Si esegue la somma su tutti i bit degli addendi, segno compreso
2. Un eventuale riporto (carry) oltre il bit di segno (MSB) viene scartato
3. Nel caso gli operandi siano di segno concorde occorre verificare la presenza o meno di overflow (il segno del risultato non è concorde con quello dei due addendi)

L'overflow non si presenta mai quando si sommano operandi di segno opposto.  
L'overflow si presenta se:

$$(+A) + (+B) = -C \quad (2.2)$$

oppure

$$(-A) + (-B) = +C \quad (2.3)$$

Un altro modo per vedere se c'è overflow è guardare i riporti nelle ultime due posizioni più significative, se sono diversi c'è overflow.

### Sottrazione

La sottrazione tra 2 numeri in **CA2** viene trasformata in somma applicando la seguente Regola

$$A - B = A + (-B) \quad (2.4)$$

Tradotto in termini di CA2

$$A - B = A + \text{CA2}(B) \quad (2.5)$$

Si effettua il complemento del sottraendo

**Overflow** Per assicurarsi della correttezza del risultato bisogna verificare l'assenza di Overflow:

1. Non si ha overflow se gli operandi hanno segno discorde
2. Si ha overflow se gli operandi hanno segno concorde e il segno del risultato è discorde con essi

Gli operandi devono essere rappresentati sempre con lo stesso numero di bit, per questo motivo in caso ci fossero meno bit si replica n volte il bit di segno (questo non altera il risultato)

## 2.4 Riassunto Overflow

In quali dei seguenti casi si può ottenere overflow?

- somma di due numeri con segno concorde? Sì
- somma di due numeri con segno discorde? NO
- sottrazione di due numeri con segno concorde? NO
- sottrazione di due numeri con segno discorde? Sì

## 2.5 Operazione di Shift

Consiste nello spostare (shift) verso destra (right) o verso sinistra (left) la posizione delle cifre di un numero, espresso in una base qualsiasi, inserendo uno zero nelle posizioni lasciate libere.

- Left equivale a moltiplicare il numero per la base
- Right equivale a dividere il numero per la base

## Rappresentazione Eccesso $2^n$

Un numero X è rappresentato come segue:

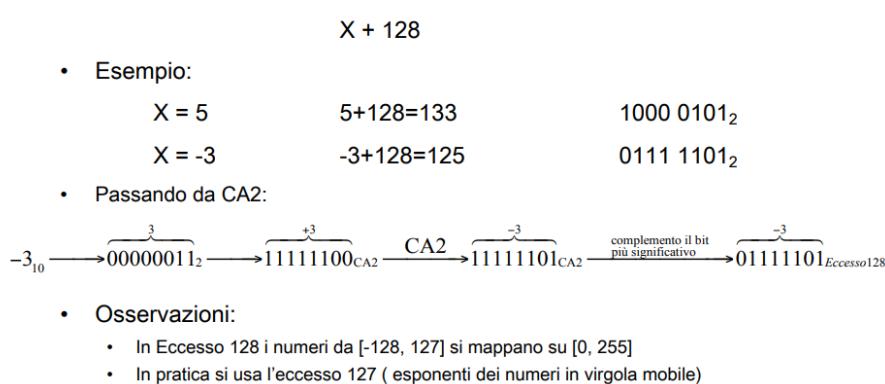
$$X + 2^{n-1} \quad (2.6)$$

Con n bit si rappresenta l'eccesso

**Regola pratica** I numeri in eccesso  $2^{n-1}$  si ottengono da quelli in CA2 complementando il bit più significativo

## 2.6 Rappresentazione Eccesso 128

Il numero è rappresentato come segue



## 2.7 Rappresentazione con la virgola

### 2.7.1 Virgola fissa

È il metodo più semplice, scegliamo dove mettere la virgola e la fissiamo. Il problema è che in base a dove posiziono la virgola ho diverse capacità di rappresentazione della parte intera o frazionaria.

- Più a sinistra, scarsa rappresentazione intera, alta rappresentazione frazionaria
- Più a destra, scarsa rappresentazione frazionaria, alta rappresentazione intera

Questo porta rigidità

### 2.7.2 Virgola mobile

- Usa 1 bit per rappresentare il segno s
- Usa altri bit per rappresentare la mantissa m
- Usa altri bit per codificare l'esponente e

Seguendo lo standard IEEE 754 la suddivisione è effettuata nella seguente modalità

#### 32 bit

- Segno - 1
- Esponente - 8
- Mantissa - 23

#### 64 bit

- Segno - 1
- Esponente - 11
- Mantissa - 52

### 2.7.3 Errore assoluto ed errore relativo

Rappresentando un numero reale  $n$  in virgola mobile si commette un errore di approssimazione, dato che viene rappresentato un numero razionale  $n$  con un numero limitato di cifre significative

#### Legenda

- $n$  - Valore che voglio rappresentare
- $n'$  - Rappresentazione effettiva
- $e_A$  - Errore assoluto
- $e_R$  - Errore relativo

#### Definizioni

- Errore assoluto - La differenza tra il numero che voglio rappresentare e la sua effettiva rappresentazione -  $e_A = n - n'$
- Errore relativo -  $e_R = \frac{e_A}{n}$

### 2.7.4 Codifiche

Spiegazione ASCII, Unicode. Vedere tabella per convertire stringhe in numeri.

# Capitolo 3

## Circuiti logici

Nell'elettronica digitale sia gli ingressi che le uscite possono assumere solo i valori di segnale alto (1 per convenzione) o basso (0).

Un circuito combinatorio è senza memoria. Il suo output dipende solo dall'input inserito da noi, non ci sono condizioni pregresse.

Un circuito sequenziale l'output non è influenzato solo dall'input, ma anche dallo stato in cui si trova il circuito. Vi è quindi la presenza di uno stato, che a livello elettronico implica la presenza di una memoria che memorizzi dei valori.

### 3.1 Porte Logiche

Le porte logiche sono i componenti elettronici che permettono di svolgere le operazioni logiche primitive oltre che a quelle direttamente derivate. Esse realizzano le operazioni principali dell'algebra booleana. Sono circuiti elettronici che dati dei segnali 0 e 1 in input producono un segnale in output ottenuto effettuando una operazione booleana sugli ingressi. Le porte logiche hanno n input e generalmente 1 output

#### 3.1.1 AND \*

Questa porta logica svolge l'operazione logica di AND tra due bit, detta anche **prodotto logico**.



A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

### 3.1.2 OR +

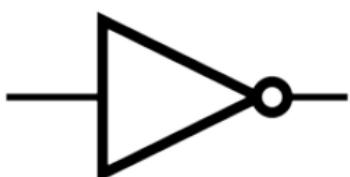
Svolge l'operazione logica di OR tra due bit, detta anche **somma logica**.



A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

### 3.1.3 NOT

Svolge l'operazione logica di NOT su un bit, detta anche **negazione logica**.



A	$\bar{A}$
0	1
1	0

### 3.1.4 Porte con più di due ingressi

Ad eccezione della porta NOT, le altre porte logiche possono esistere anche ad N ingressi, queste porte svolgono l'operazione logica associata su N bit invece che su 2.

Nei circuiti si possono realizzare porte a N ingressi collegando a cascata tra loro porte a 2 ingressi.

### **3.2 Porte logiche derivate**

Porta NAND → svolge l'operazione di NOT sul bit risultante dell'operazione di AND. (nega il risultato dell'and).

Porta NOR → svolge l'operazione di NOT sul bit risultante dall'operazione di OR. (nega il risultato dell'or).

Porta XOR → Opera come disgiunzione esclusiva tra due input, quando i 2 bit sono uguali produce in output un 1. Viceversa restituisce 0. Le porte NOR e NAND svolgono la funzione di inverter, sono definite universali.

# Capitolo 4

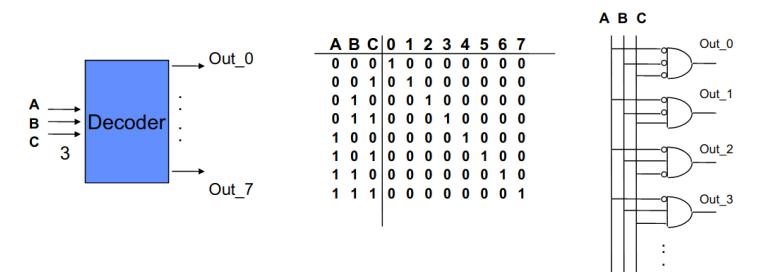
## Circuiti combinatori

Sono circuiti dove non sono presenti memorie, l'output di un circuito combinatorio dipende solo dall'input che stiamo dando al circuito.

### 4.1 Decoder

Componente elettronico caratterizzato dall'avere  $n$  ingressi e  $2^n$  uscite. Solo 1 valore è attivo per ogni combinazione di input, quindi l'ingresso seleziona una delle uscite, l'uscita selezionata ha valore 1 tutte le altre 0.

Nella foto il pallino vuoto corrisponde a 0, mentre senza pallino corrisponde a 1. Esiste anche il suo inverso denominato encoder che riceve  $2^n$  input e produce un output di  $n$  bit.



### 4.2 Multiplexor

Un multiplexor, detto anche selettore, è un componente elettronico caratterizzato da

- $2^n$  entrate principali

- n entrate di controllo (selettore)
- 1 uscita

Riceve n segnali in input e un solo segnale di output. Nel caso in cui riceva 2 input (A e B) + il selector (S), il valore di output sarà A o B in base al segnale del selettore (0 o 1, quindi True o False).

Un multiplexor a 32 bit corrisponde a un array di 32 multiplexor ad 1 bit.

#### 4.2.1 Relazione importante per gli es

La relazione tra il massimo numero di ingressi selezionabili (n) con m segnali/ingressi di selezione è:

$$m = \log_2 n \text{ ovvero } n = 2^m$$

Nel caso io abbia un multiplexor a 32 ingressi il numero di ingressi di selezione m sarà:

$$2^m = 32 \text{ ovvero } m = 5$$

### 4.3 Logiche a due livelli e PLA

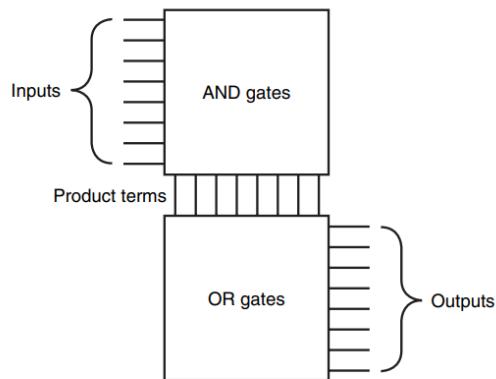
Possiamo creare logiche a due livelli:

- Somma di prodotti: somma logica (OR) di prodotti (AND)
- Prodotto di somme: prodotto (AND) di somme (OR)

Esercizio tabella di verità

**Programmable Logic Array** La somma di prodotti corrisponde ad una implementazione comunemente nota come **Programmable Logic Array**

- Insieme di input
- I corrispondenti input complementati (mediante inverter)
- Una logica a due stage:
  - Primo stage: un array di porte logiche AND (prodotto)
  - Secondo stage: un array di porte logiche OR (somma)



## 4.4 ROM (Read Only Memory)

Circuito combinatorio in cui ad ogni ingresso (indirizzo) corrisponde una uscita (contenuto della cella di memoria con quell'indirizzo).

- Input:  $n$  bit
- Output: una tra le  $2^n$  celle/locazioni di memoria

### ROM vs PLA

- ROM fully decoded
- PLA - partially decoded
- ROM dimensione più grande rispetto a PLA
- PLA più efficienti
- ROM possono implementare qualsiasi funzione logica
- PROM (Programmable ROM) → possono essere programmate
- EPROM (Erasable Programmable ROM)

Normalmente il bus è di 32 bit

## 4.5 ALU

L'Arithmetic Logic Unit è la parte del processore che svolge le operazioni aritmetico-logiche.

È un insieme di circuiti combinatori che implementa:

- Operazioni aritmetiche, es. somma e sottrazione
- Operazioni logiche: es AND e OR

Blocchi per la costruzione di una ALU

- AND
- OR

Una ALU a 32 è semplicemente l'interconnessione di 32 ALU ad 1 bit in cascata

### Blocchi base per costruire ALU

- AND gate
- OR gate
- Inverter
- Multiplexor

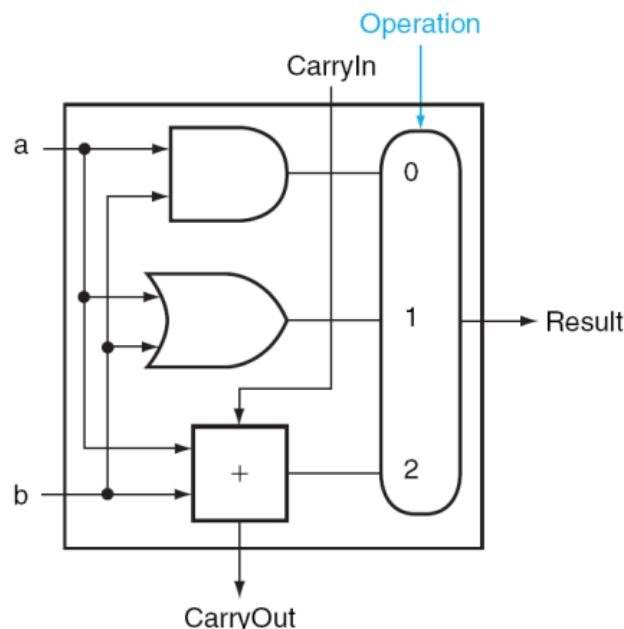
#### 4.5.1 Operazioni aritmetiche

##### Addizione

a	b	CarryIn	CarryOut	Somma
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>0</b>
1	0	0	0	1
<b>1</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>
1	1	0	1	0
<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>	<b>1</b>

$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

- ALU su 1 bit che esegue somma, AND, OR



**Sottrazione** Negando *b* possiamo ottenere una sottrazione (in CA2). Per questo si aggiunge un Binvert e lo si setta a 1 nel caso serva effettuare una sottrazione

**NOR** si implementa con De Morgan Connnettendo 32 ALU da 1 bit otteniamo una ALU a 32 bit. La connessione si effettua collegando il bit meno significativo con il CarryIn del bit più significativo. Questa organizzazione è chiamata ripple carry.

## 4.6 Operazioni di confronto - SLT e BEQ

Le operazioni di confronto all'interno di una ALU sono BEQ (Branch-on-equal) e SLT (set-on-less-than)

### 4.6.1 SLT

Risultato: 1 se  $a < b$ , 0 altrimenti.

Per eseguire questa istruzione si devono poter azzerare tutti i bit dal bit-1 alcuni bit-31 e assegnare al bit-0 il valore del risultato. Per poter realizzare il confronto si effettua la sottrazione tra a e b.

- Se  $a - b < 0$  allora  $a < b$  e il risultato sarà 000...01
- Se  $a - b > 0$  allora  $a > b$  e il risultato sarà 000...00

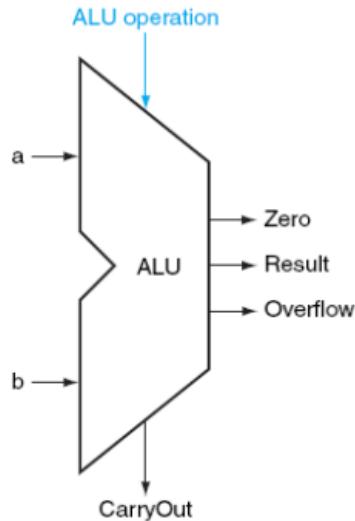
### 4.6.2 BEQ - Branch on equal

Per verificare l'uguaglianza di 2 elementi si fa la sottrazione fra di essi, se il risultato è zero i due elementi sono uguali. Logicamente si fa un OR di tutte le uscite delle ALU a 1 bit e poi si nega il risultato.

### 4.6.3 Overflow detection

Infine per completare la ALU è necessario inserire nell'ultima operazione un circuito per l'overflow detection. Se all'ultima operazione è presente un bit di CarryOut si è verificato un overflow.

Questa è il simbolo comunemente utilizzato per rappresentare la ALU.



#### 4.6.4 Carry Lookahead

Obiettivo: anticipare i bit di CarrIn per i bit più significativi per velocizzare le operazioni

Soluzione: passare tra meno porte

Teniamo conto che ALU ha 3 ingressi: 2 operandi e CarryIN (per la prima ALU su 1 bit)

$$\begin{aligned} \text{CarryOut}_0 &= \text{CarryIn}_1 = (b_0 * \text{CarryIn}_0) + (a_0 * \text{CarryIn}_0) + (a_0 * b_0) \\ \text{CarryOut}_1 &= \text{CarryIn}_2 = (b_1 * \text{CarryIn}_1) + (a_1 * \text{CarryIn}_1) + (a_1 * b_1) \end{aligned}$$

Possiamo quindi riscrivere CarryIn2 sostituendo CarryIn1 della prima equazione

$$\text{CarryIn}_2 = b$$

## 4.7 Note per gli esercizi

### 4.7.1 Conversione da tabella a circuito

È sufficiente prendere in considerazione solo le righe dove ho 1 come output, i valori relativi a quella riga vanno concatenati con degli AND, dato che solo se si verificano tutti ottengo 1 come output, mentre per concatenare le righe vere fra di loro utilizzo gli OR, dato che è sufficiente che si verifichi anche solo 1 di queste righe per poter ottenere 1 come risultato.s

### **4.7.2 Calcolo output circuito**

Se si tratta di somme di prodotti (and concatenati da or) è sufficiente identificare un valore che dà true dato che essendo una catenda di OR basta un solo true perchè l'espressione sia true. Mentre è necessario controllare che siano tutti false per affermare che l'intera espressione sia falsa.

Se si tratta di prodotti di somme (OR concatenati da AND) è necessario verificare che tutte le sotto espressioni siano vere per poter affermare che l'intera espressione sia vera. Mentre è sufficiente che una singola sotto espressione sia false per poter affermare che l'intera espressione sia falsa.

# Capitolo 5

## Circuiti sequenziali

Circuiti dove sono presenti delle memorie, per questo motivo l'output non dipende solo dall'input che stiamo dando, ma dipende anche da valori salvati precedentemente in memoria, questo concetto è chiamato **stato** nei blocchi logici.

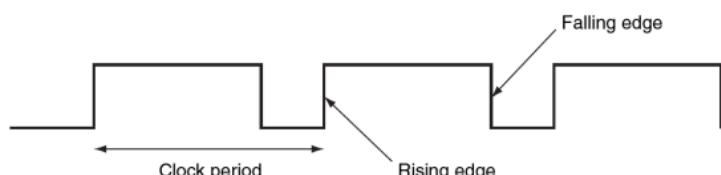
**Differenza con i circuiti combinatori** La sequenza temporale determina il valore memorizzato nello stato.

Sintetizzando, i circuiti sequenziali sono formati da:

- Elementi di memoria (di vario tipo) che memorizzano informazione
- Reti combinatorie che elaborano informazione

### 5.1 Clock

Il segnale di clock è fondamentale per le reti sequenziali caratterizzate da uno stato. Il clock è definito come un segnale (onda quadra) con un periodo predeterminato e costante.



Caratterizzato da un periodo  $T$  (oppure ciclo) di clock e dalla frequenza  $F$  definita come  $\frac{1}{T}$  e misurata in Hertz.

Il periodo di Clock è abbastanza grande da assicurare la stabilità degli output di un circuito e determina quando il contenuto di un elemento che

rappresenta lo stato è aggiornato. Sostanzialmente determina il ritmo dei calcoli e delle operazioni di memorizzazione.

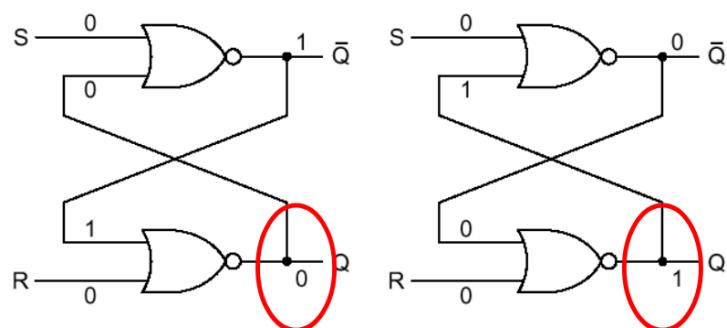
Con il clock il circuito diventa sincrono.

La tipologia di clocking studiata è la **edge-triggered clocking**, cioè tutti i cambiamenti avvengono sul bordo di un clock. Esistono altre tipologie di clock utilizzate, come la **level-triggered** (utilizzata nei primi esempi), ma noi studieremo solo edge triggered.

## 5.2 S-R Latch

**S-R Latch** è un circuito composto da 2 porte NOR concatenate dove

- S = Set
- R = Reset



Input		Stato Interno Old Q	Output	
S	R		Q	$\bar{Q}$
0	0	0	0	1
0	0	1	1	0
0	1	1/0	0	1
1	0	1/0	1	0
1	1	1/0	1	1

Come si può notare dalla tabella tabella di verità, il valore di set serve per salvare un nuovo valore, quello di reset per scartare il valore salvato. Se entrambi i valori sono a zero il valore salvato resta immutato.

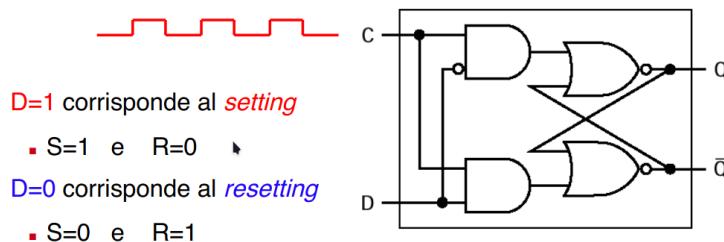
La combinazione set = 1 e reset = 1 va assolutamente evitata, dato che viola le proprietà di complementarietà e può portare ad una configurazione instabile. I valori S e R devono essere stabili e valere (1,0) o (0,1) per poter

memorizzare un valore corretto. Essi sono di solito calcolati da un circuito combinatorio.

Per evitare che vengano memorizzati i valori mentre il circuito combinatorio li sta calcolando viene utilizzato il clock. Esso determina il tempo in cui è possibile memorizzare un dato.

## 5.3 D Latch

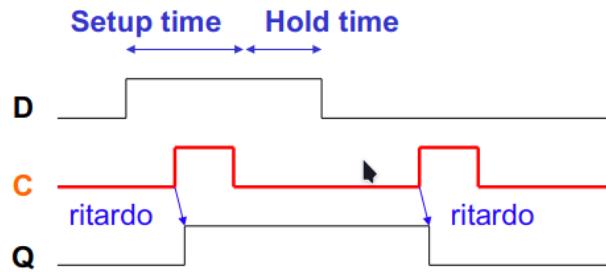
Caratterizzato da  $D =$  il valore da memorizzare,  $C =$  il segnale di clock, ovvero quando il latch deve leggere e memorizzare il valore. Gli output sono sempre i valori degli stati interni  $Q$  e il suo complementare  $\bar{Q}$ .



È un Latch sincronizzato con il clock, così il latch cambia stato in momenti opportuni Quando il clock è deasserted non viene memorizzato nessun valore, mentre quando è asserted viene memorizzato un valore (in funzione di D). Il segnale D, ottenuto come output di un circuito combinatorio deve:

- Essere già stabile quando C diventa asserted
- Rimanere stabile per tutta la durata del livello alto di C (Setup time)
- Rimanere stabile per un altro periodo di tempo per evitare malfunzionamenti

Nei circuiti teorici dove il ritardo è nullo il problema non si pone, ma nei circuiti reali il **ritardo NON è nullo**. Gli output possono temporaneamente cambiare da valori corretti a valori errati e ancora a valori corretti, questo fenomeno è denominato **Glitch**. Dopo un certo intervallo, con alta probabilità i segnali si stabilizzano.



Per questo motivo il clock T deve essere scelto abbastanza lungo affinchè l'output del circuito combinatorio si stabilizzi. Deve essere stabili un po' prima del periodo di apertura del latch (setup time) e lo deve rimanere per un certo tempo (hold time).

**Funzionamento D Latch** Quando il clock C è asserito allora il latch è aperto e si può memorizzare il valore può essere memorizzato (il valore di output diventa il valore di input D). Mentre quando il clock C non è asserito il latch viene detto chiuso e il valore di output Q è il l'ultimo valore memorizzato l'ultima volta che il latch era aperto.

### Il D-latch è caratterizzato dal seguente comportamento

- Durante l'intervallo alto del clock il valore del segnale di ingresso D viene memorizzato nel latch
- Il valore di D si propaga quasi immediatamente all'uscita Q
- Anche le eventuali variazioni di D si proaggano quasi immediatamente, con il risultato che Q può variare più volte durante l'intervallo alto del clock
- Solo quando il clock torna a zero Q si stabilizza
- Durante l'intervallo basso del clock il latch **non memorizza**

Fino ad ora abbiamo visto solo esempi con metodologia di clocking detta level-triggered, ora vedremo la **edge-triggered**.

#### 5.3.1 Edge-triggered clocking

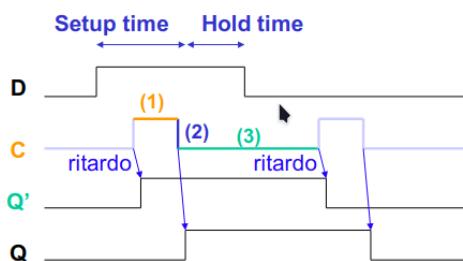
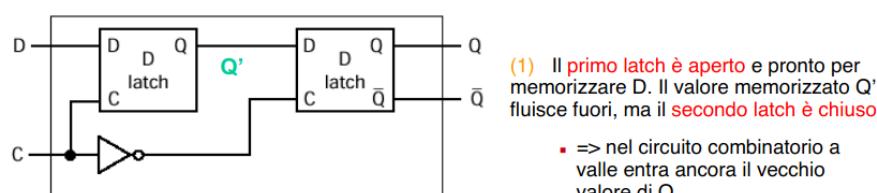
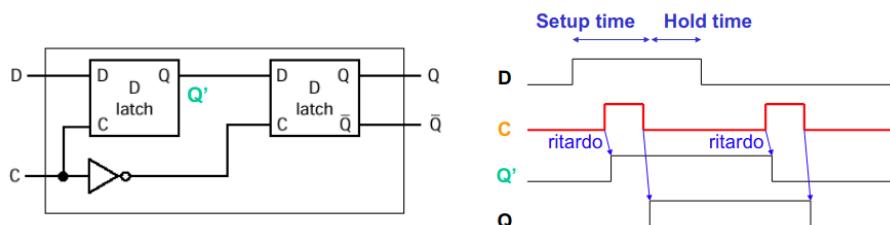
In questo caso il timing avviene sul fronte di salita (o di discesa) del clock, otteniamo così i seguenti vantaggi

- La memorizzazione avviene istantaneamente
- L'eventuale segnale di ritorno sporco non fa in tempo ad arrivare a causa dell'istantaneità della memorizzazione

Il Latch è trasparente, ma non può essere utilizzato sia come input che output all'interno dello stesso ciclo.

## 5.4 D Flip Flop

Il D Flip-Flop è utilizzabile come input e output durante lo stesso ciclo di clock. Realizzato ponendo in serie 2 D-latch: il primo viene denominato **master** il secondo **slave**.



Il Flip Flop non è trasparente, dato che il suo valore cambia solo sui on the clock edge.

## 5.5 Differenze tra Latch e Flip-Flop

La principale differenza è nel momento in cui questi due circuiti operano. Il latch memorizza quando riceve un valore da memorizzare e il clock è assegnato.

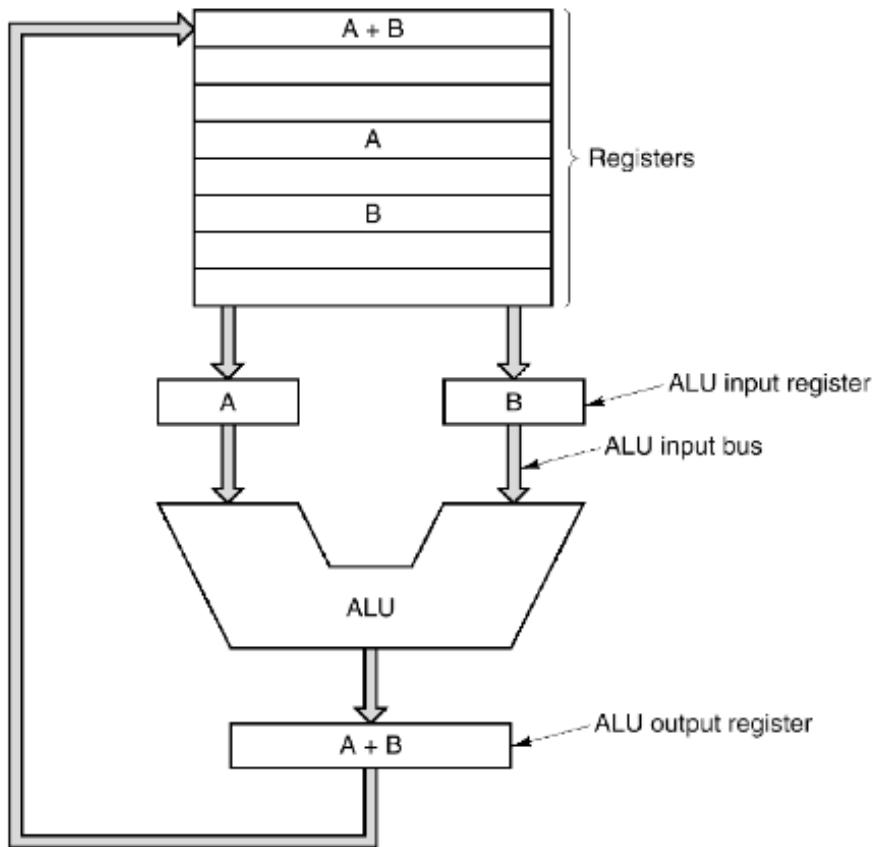
Flip Flop invece memorizza **solo** on a clock edge. Per questo ultimo motivo e per la possibilità di poter essere usato sia come input che come output all'interno di un solo ciclo utilizzeremo principalmente flip-flop.

## 5.6 Datapath e Register File

Il Datapath è un insieme di unità di calcolo, come ad esempio la ALU e i registri, necessari per l'esecuzione delle istruzioni nella CPU. I registri sono presenti all'interno di una CPU e costituiscono una delle strutture principali di un processore. Possono essere letti e modificati fornendo un register number a cui accedere.

### Implementazione

Un registro è costituito da n flip-flop. (in MIPS ogni registro è di 1 word = 4byte = 32 bit). I registri sono organizzati in un **Register File**. Il Register File del MIPS ha 32 registri ( $32 \times 32 = 1024$  flip-flop). Il register file permette la lettura di 2 registri e la scrittura di 1 registro.



Nel Datapath della CPU il clock non entra direttamente nei vari flip-flop, viene messo in AND con un segnale di controllo "Write", il segnale che determina se, in corrispondenza del fronte di discesa del clock, il valore D debba (o meno) essere memorizzato nel registro.

### Lettura dal Register File

- Utilizza 2 segnali che indicano i registri da leggere (Read Reg1, Read Reg2)
- Utilizza 2 multiplexer: ognuno con 32 ingressi e un segnale di controllo da 5 bit
- Il register file fornisce **sempre** in output una coppia di registri

Utilizza 3 segnali:

- Il registro da scrivere

- Il valore da scrivere
- Il segnale di controllo

Utilizza un decoder che decodifica il numero del registro da scrivere (Write Register)

Il segnale Write (già in AND con il clock) è in AND con l'output del decoder  
Se Write non è asserito nessun valore sarà scritto nel registro

### Alcune note

- Ogni registro può contenere valori di qualsiasi tipo purché usino il numero di bit a disposizione nel registro
- Spetterà a chi definisce il programma in binario usare i dati memorizzati nei registri in modo coerente con il loro tipo
- Il ciclo di clock deve essere sufficientemente lungo da permettere al segnale di propagarsi dall'output del Register File al suo input, compreso il computo della ALU.

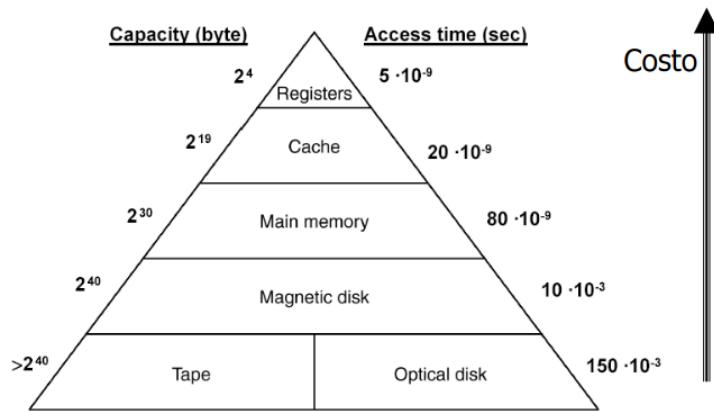
## 5.7 Memoria

Oltre alle piccole memorie dei registri e file di registro esistono altri tipi di memoria che possiamo distinguere in base a diversi parametri

1. Dimensione: quantità di dati memorizzati
2. Velocità: tempo tra richiesta e risposta
3. Consumo: potenza assorbita
4. Costo: costo per bit

Non è possibile avere una memoria con tutte le caratteristiche ideali.  
Possiamo però organizzare una gerarchia

- Memoria piccole e più veloci (costose) sono poste ai livelli alti
- Memorie ampie e più lente (meno costose) sono poste ai livelli più bassi



## RAM

La dimensione del register file è piccola, per memorizzare dati strutturati è necessaria una memoria più grande come la **RAM - Random Access Memory**, che è meno veloce della memoria dei registri, ma molto più capiente. La RAM è un insieme di celle da 1 byte ognuna individuata da un indirizzo. Le operazioni di lettura copiano il contenuto della cella di memoria nel registro dati, mentre le operazioni di scrittura copiano il contenuto del Registro Dati nella cella.

## SRAM e DRAM

### Static RAM

- è più veloce, tempi di accesso intorno a 0,5 - 2,5 ns
- vengono utilizzati dei latch per la realizzazione

### Dynamic RAM

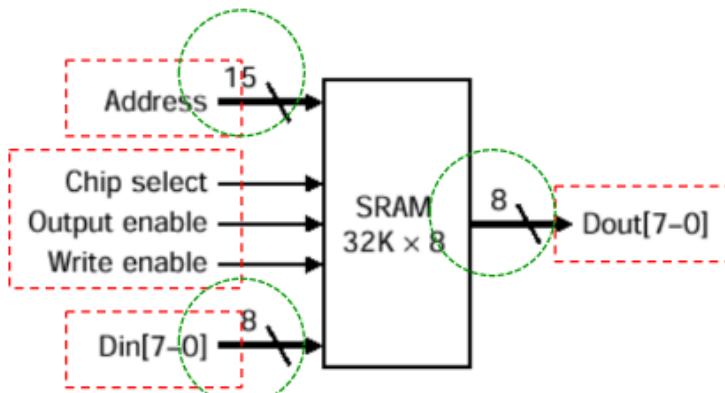
- Ogni bit è memorizzato tramite un condensatore
- tempi di accesso intorno a 50-70 ns
- è necessario rinfrescare il contenuto della DRAM a intervalli di tempo prefissati

## SRAM

È realizzata come matrice di latch

- Larghezza o ampiezza = numero di latch per ogni cella
- Altezza H = numero di celle indirizzabili
- Singolo indirizzo usato sia per lettura che per scrittura
- Non è possibile leggere e scrivere contemporaneamente

### Circuito SRAM



### Spiegazione circuito

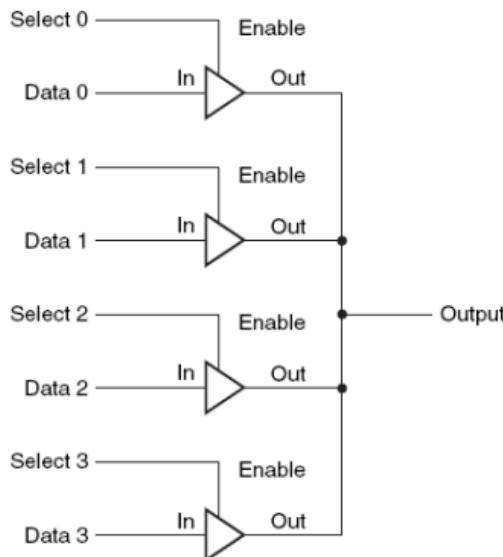
- Din e Dout
- Address
- Segnali di controllo
  - Chip select deve essere asserito per poter leggere o scrivere
  - Output enable deve essere asserito per poter leggere
  - Write enable deve essere asserito per poter scrivere

## Realizzazione SRAM

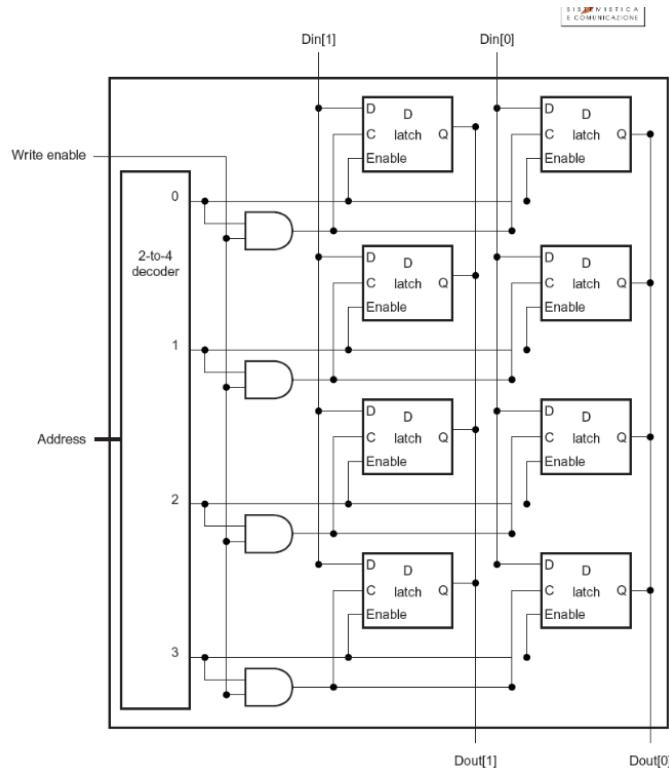
Non è possibile utilizzare decoder e multiplexer come per i registri perché data la dimensione notevolmente maggiore avremmo bisogno di enormi multiplexer e decoder, con molte porte AND, questo porterebbe all'introduzione di ritardi considerevoli agli accessi della memoria.

Per evitare multiplexer in uscita quindi:

- Possiamo usare una linea di bit condivisa su cui i vari elementi di memoria sono tutti collegati (OR)
- Il collegamento alla linea condivisa avviene tramite buffer a tre stati, che aprono o chiudono i collegamenti (se il controllo è asserito o meno)
- Il buffer ha due ingressi (dato e segnali di Enable) e una uscita
  - l'uscita è uguale al dato (0 o 1) se Enable è asserito
  - l'uscita viene impedita se Enable non è asserito



Esempio SRMA 4x2



I latch di una certa colonna sono collegati alla stessa linea in output (Dout[0] e Dout[1]).

Nell'esempio ogni D-latch ha un segnale di Enable che abilita il three-state buffer interno. Il decoder serve ad abilitare in lettura e scrittura una certa linea di memoria. Entrambi i segnali Write enable e Enable vengono abilitati su una sola linea di memoria (2 bit).

## SRAM a 2 livelli

Per diminuire la complessità dei decoder è opportuno suddividere gli indirizzabili in 2 blocchi:

1. parte alta per accedere a una riga
2. parte bassa per accedere a una specifica colonna

Nota che celle consecutive hanno indirizzi che solitamente differiscono solo per la parte bassa dell'indirizzo, la parte in alto è quindi comune.

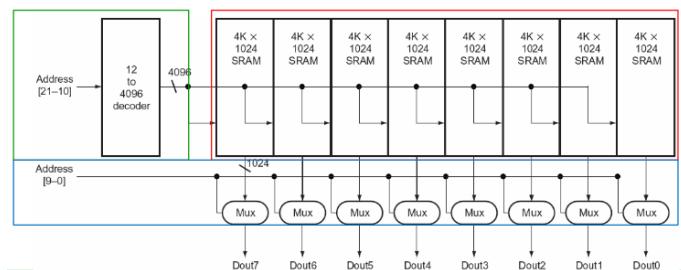
Le **Synchronous SRAM e DRAM (SSRAM e SDRAM)** permettono di aumentare la banda di trasferimento alla memoria sfruttando questa proprietà.

Synchronous perchè sono sincronizzate con un segnale di clock.

- è possibile specificare che vogliamo trasferire dalla memoria una **sequenza di celle consecutive (burst)**
- ogni burst è specificato da un indirizzo di partenza e da una lunghezza
- le celle del burst sono contenute all'interno di una stessa Riga, **selezionata una volta per tutte tramite decoder**
- La memoria fornisce una delle celle del burst a ogni ciclo di clock

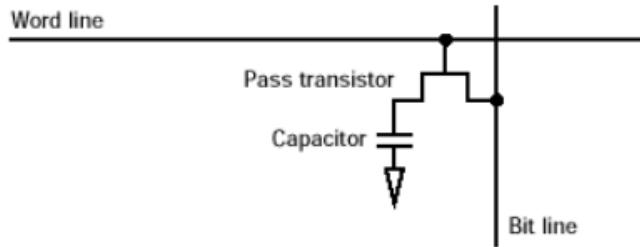
In pratica scegliendo una sequenza di celle consecutive si utilizza una volta sola il decoder, in seguito si utilizza solo il multiplexor. Questo migliora la banda di trasferimento, inoltre non è necessario ripresentare l'indirizzo per ottenere ogni cella del burst. Esempio di SRAM:  $4M \times 8 \rightarrow$  servono 22 linee di indirizzo ( $4M = 2^{22}$ )

- suddiviso in 8 blocchi da 4Mb
- parte alta dell'indirizzo [21-10] seleziona la medesima riga di ogni blocco attraverso un Decoder
- parte bassa dell'indirizzo [9-0] seleziona singoli bit dei 1024 bit in output dai vari blocchi attraverso una serie di 8 Multiplexer

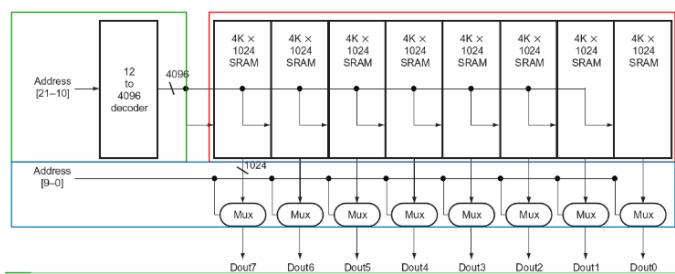


**Perchè usare le DRAM** Le DRAM sono meno costoso e più capienti rispetto alle SRAM, essendo realizzate con un solo transistor per bit e un solo condensatore occupano meno spazio per singolo bit, inoltre i condensatori hanno un basso costo. Purtroppo sono più lente rispetto alle SRAM (da 5 a 10 volte più lente!).

### Funzionamento DRAM



- Il condensatore possiede una carica (0/1)
- Il transistor viene chiuso, trasferendo il potenziale elettrico del condensatore sulla Bit line (output), grazie al segnale affermato sulla Word line
- la specifica Word line è attivata sulla base dell'indirizzo di memoria richiesto



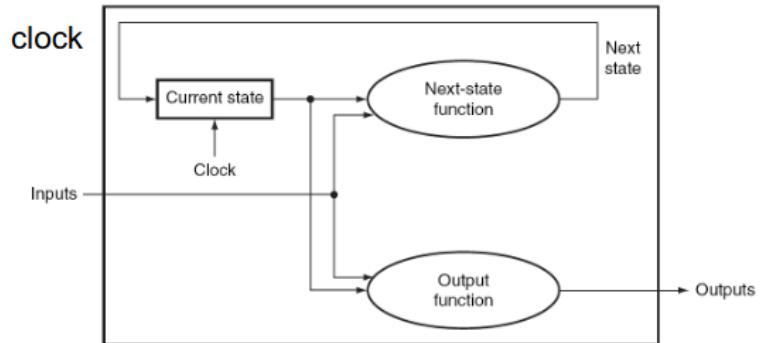
**Perchè usare le DRAM** Le DRAM sono meno costoso e più capienti rispetto alle SRAM, essendo realizzate con un solo transistor per bit e un solo condensatore occupano meno spazio per singolo bit, inoltre i condensatori hanno un basso costo. Purtroppo sono più lente rispetto alle SRAM (da 5 a 10 volte più lente!).

## 5.8 Macchine a stati finiti

Per la rappresentazione dei circuiti combinatori vengono utilizzate delle tabelle di verità, mentre per i circuiti sequenziali sono utilizzate le **Finite State Machine (FSM)**. Sono composte da un set di stati e 2 funzioni

- Next state function - determina lo stato successivo partendo dallo stato corrente e dai valori in ingresso
- Output function - produce un insieme di risultati partendo dallo stato corrente e dai valori di ingresso

Sono sincronizzate con il clock



**FSM - Moore vs Mealy** Se la macchina dipende dallo stato corrente e basta, quindi viene usata come controller parliamo di una **FSM Moore**. Se la macchina dipende dallo stato corrente e dagli input parliamo di una **FSM Mealy**. In questo corso si utilizza Moore.

# Capitolo 6

## Instruction Set Architecture (ISA)

Per analizzare l'architettura di un computer analizziamo tre aspetti fondamentali:

- Cosa fa (ISA)
- Come si programma (Assembly Language)
- Come è fatto (circuiti e Datapath)

### 6.1 Filosofie di progetto della CPU

Ci sono 2 principali filosofie di progetto della CPU

- RISC (Reduced Instruction Set Computing)
- CISC (Complex Instruction Set Computing)

#### 6.1.1 RISC

- Poche istruzioni semplici
- Struttura circuitamente semplice
- Esecuzione veloce di una singola operazione
- Occorrono più istruzioni per fare cose anche semplici

Ottimi esempi di questa tipologia di architettura sono MIPS, ARM e PowerPC.

**Implementazioni** MIPS è stata utilizzata per i processori del Nintendo64, mentre PowerPC per quanto riguarda PlayStation , Xbox 360, Nintendo Wii e GameCube.

### 6.1.2 CISC

- Istruzioni complesse
- Struttura circuitalmente complicata
- Esecuzione più lenta di una singola istruzione
- Occorrono meno istruzioni

Un esempio diffusissimo è Intel x86 e tutti i derivati.

## 6.2 Istruzioni MIPS

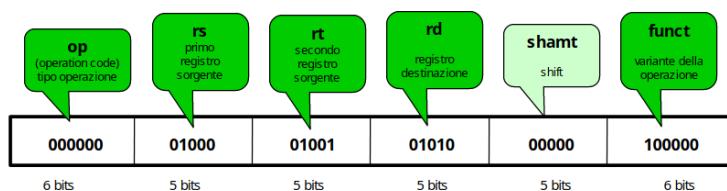
Noi utilizzeremo MIPS attraverso QTSPIM. Essendo RISC è semplice, uniforme e più semplice da capire. Utilizza 32 registri di 32 bit e istruzioni di 32 bit.

Ogni istruzione ha un formato diverso, quindi raggruppa i 32 bit in modo diverso. In MIPS le istruzioni sono divise in tre formati

- R-type - Istruzioni relative ai registri
- I-type - Istruzioni immediate (costanti passate direttamente nel codice dell'operazione)
- J-type

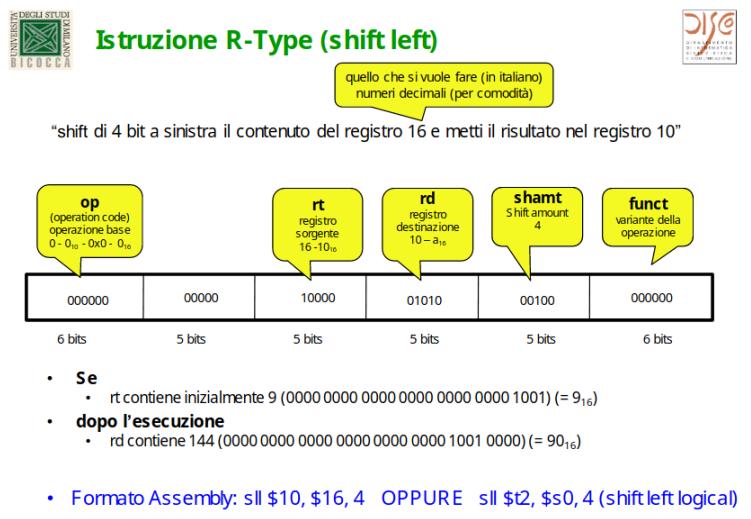
### 6.2.1 R-Type

Sono le istruzioni che operano sui registri Formato istruzione.



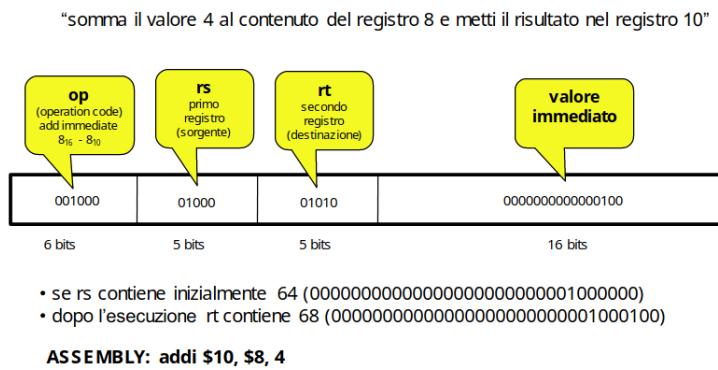
**OpCode** In questo caso l'opcode dice che è di tipo R dato che le R-type Instruction hanno tutte OpCode 000000. Per identificare l'istruzione è necessario controllare il campo funct. Per gli altri tipi di istruzione non esiste.

### Shift left



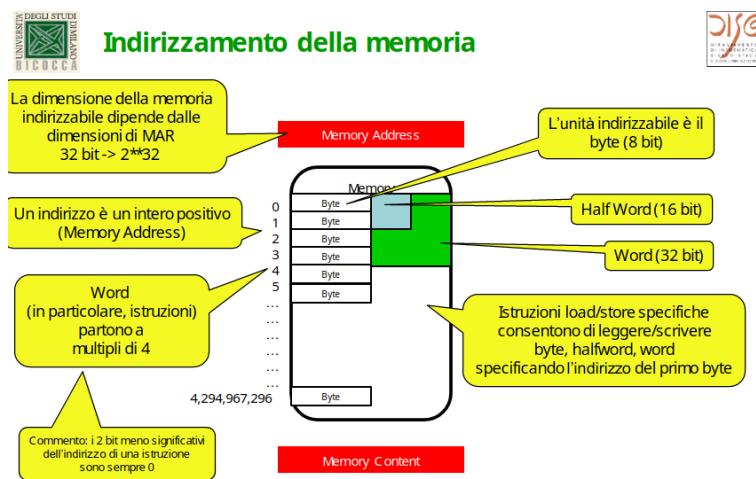
### 6.2.2 I-type

Istruzioni per la somma di valori ricorrenti (costanti), in questo caso la costante viene direttamente passata nel codice dell'operazione, riservando 16 bit per il valore (campo passato in complemento a 2). Vengono chiamate I-type perchè si tratta di operazioni immediate. Il valore è considerato come un intero con segno ( $2^{15}$  valori rappresentabili) Il formato è il seguente:



Chiaramente le stringhe di bit sono poco leggibili, per questo esiste assembly. Scrivo delle parole e attraverso un assembler lo trasformo in linguaggio macchina, traduco quindi il codice sorgente in eseguibile.

**Indirizzamento della memoria** Nota utile per la spiegazione della prossime istruzioni



### Load and Store

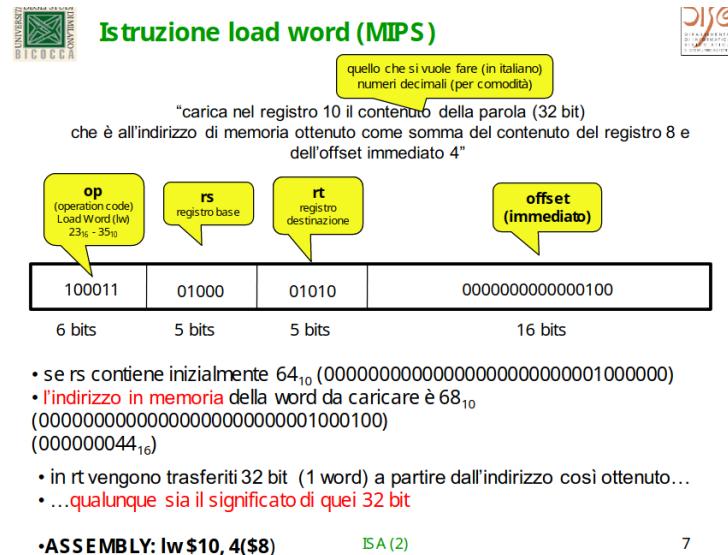
Istruzione per prelevare un valore dalla memoria RAM e salvarlo in un registro della CPU. Istruzione composta da:

- opcode
- Registro di destinazione
- Indirizzo di memoria il cui contenuto viene copiato nel registro destinazione

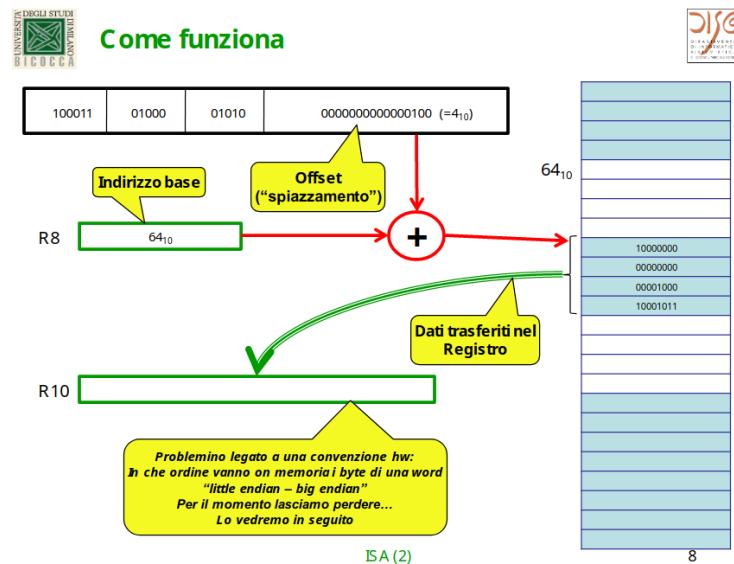
**Problema** La memoria è grande, in MIPS ho  $2^{32}$  locazioni, per specificare un indirizzo occorrono 32 bit, ma non ci stanno in una istruzione di appunto 32 bit.

**Soluzione** Usare il formato I-type (opcode, 2 registri, 1 immediato), dove un registro è la destinazione, un registro contiene un indirizzo base e il valore immediato è interpretato come l'offset (spiazzamento) rispetto all'indirizzo base. L'indirizzo a cui accedere è calcolato quindi come (base + offset).

### Esempio Load Word MIPS



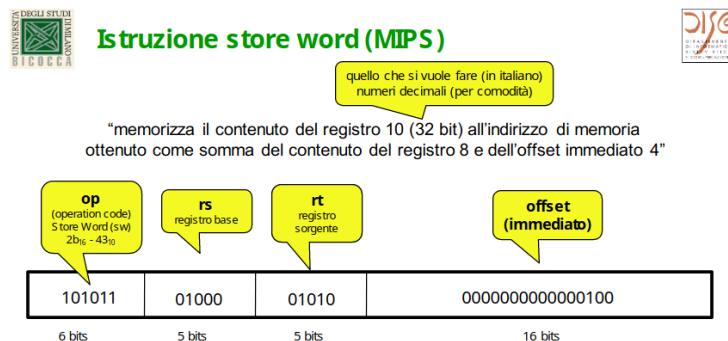
### Rappresentazione grafica esecuzione Load Word



Il problema del little endian - big endian è un problema di convenzione, alcuni calcolatori considerano i bit ricevuti in memoria dal più significativo al meno significativo, altri il contrario, più avanti si affronterà la soluzione a questo problema.

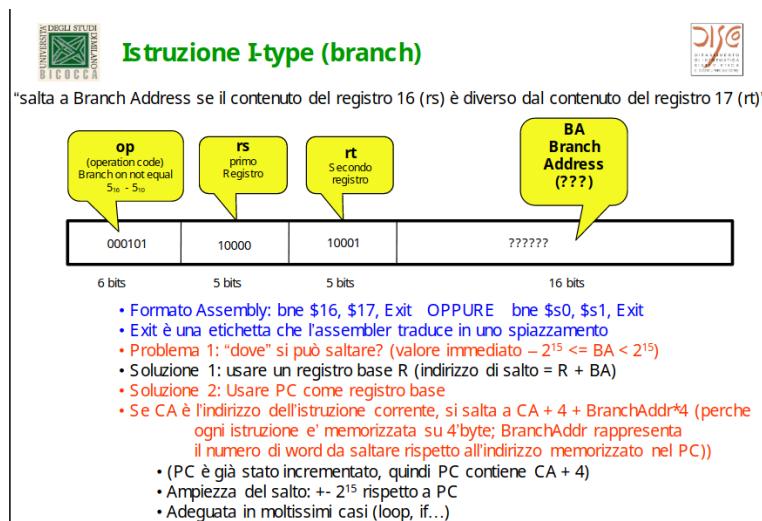
### Store word

In questo caso viene memorizzato il contenuto di un registro del processore direttamente in memoria RAM. Viene sempre utilizzata la tecnica della rappresentazione di un indirizzo di memoria come "indirizzo base + offset"



### Branch on not equal

Sempre istruzione I-Type, mi permette di saltare a Branch Address se il contenuto del primo registro è diverso dal contenuto del secondo.

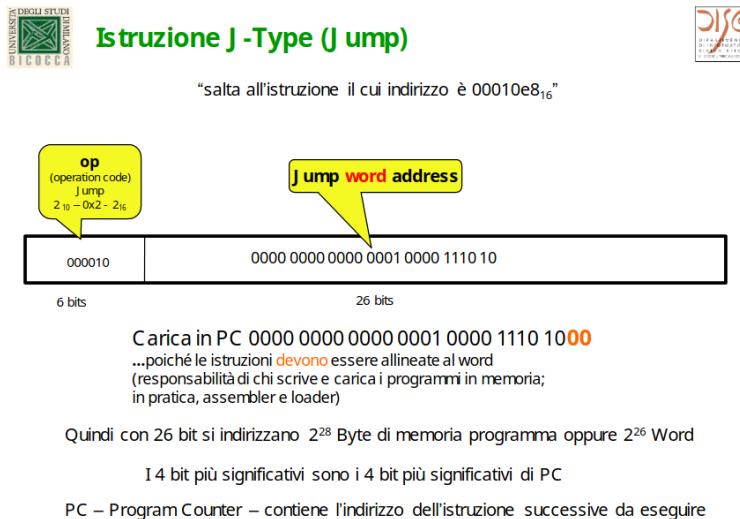


### 6.2.3 J-type

Istruzioni caratterizzate dal fatto che ci sia un salto (jump) a un determinato indirizzo.

## Jump

L'istruzione Jump mi permette di saltare a una specifica istruzione tramite indirizzo.



## 6.3 Procedure e convenzioni registri

Questa sezione tratterà il passaggio di parametri attraverso registri per esempio per il controllo del flusso del programma.

### Istruzione jal - jump and link

**Sintassi** jal "IndirizzoProcedura".

Questa istruzione salta a una procedura indicata nella stessa e contemporaneamente crea un collegamento a dove deve ritornare per continuare l'esecuzione del chiamante. Salva nel registro \$ra, return address situato nel registro 31, l'indirizzo a cui tornare dopo l'esecuzione della procedura. È l'indirizzo successivo a quello dell'istruzione jal, cioè l'indirizzo in cui si trova la jal + 4. Tale indirizzo si trova nel registro PC (Program Counter).

### jr - jump register

**Sintassi** jr "registro".

Salta all'indirizzo contenuto in un registro. È una istruzione di uso generale che consente di saltare a qualsiasi locazione di memoria.

**Utilizzo tipico di jr** jr \$ra. In questo modo realizzo il ritorno da procedura, saltando all'indirizzo precedentemente salvato da jal.

### 6.3.1 Passaggio parametri - convenzioni base registri

Riporto per chiarezza uno screenshot di QTSPIM per chiarire la denominazione dei registri della CPU.

<b>R0</b>	<b>[r0]</b>	= 0
<b>R1</b>	<b>[at]</b>	= 0
<b>R2</b>	<b>[v0]</b>	= 0
<b>R3</b>	<b>[v1]</b>	= 0
<b>R4</b>	<b>[a0]</b>	= 0
<b>R5</b>	<b>[a1]</b>	= 0
<b>R6</b>	<b>[a2]</b>	= 7fffffa64
<b>R7</b>	<b>[a3]</b>	= 0
<b>R8</b>	<b>[t0]</b>	= 0
<b>R9</b>	<b>[t1]</b>	= 0
<b>R10</b>	<b>[t2]</b>	= 0
<b>R11</b>	<b>[t3]</b>	= 0
<b>R12</b>	<b>[t4]</b>	= 0
<b>R13</b>	<b>[t5]</b>	= 0
<b>R14</b>	<b>[t6]</b>	= 0
<b>R15</b>	<b>[t7]</b>	= 0
<b>R16</b>	<b>[s0]</b>	= 0
<b>R17</b>	<b>[s1]</b>	= 0
<b>R18</b>	<b>[s2]</b>	= 0
<b>R19</b>	<b>[s3]</b>	= 0
<b>R20</b>	<b>[s4]</b>	= 0
<b>R21</b>	<b>[s5]</b>	= 0
<b>R22</b>	<b>[s6]</b>	= 0
<b>R23</b>	<b>[s7]</b>	= 0
<b>R24</b>	<b>[t8]</b>	= 0
<b>R25</b>	<b>[t9]</b>	= 0
<b>R26</b>	<b>[k0]</b>	= 0
<b>R27</b>	<b>[k1]</b>	= 0
<b>R28</b>	<b>[gp]</b>	= 10008000
<b>R29</b>	<b>[sp]</b>	= 7fffffa5c
<b>R30</b>	<b>[s8]</b>	= 0
<b>R31</b>	<b>[ra]</b>	= 0

- \$r0 (0): contiene sempre il valore 0
- \$at (1), \$k0 (26) e \$k1 (27): sono riservati per l'assembler e il sistema operativo

- \$a0 - \$a3 (4-7): registri argomento per il passaggio dei parametri
- \$v0, \$v1 (2,3): registri valore per la restituzione dei risultati
- \$t0 - \$t9 (8-15, 24, 25): chiamati **caller-saved register** sono utilizzati per memorizzare quantità temporanee che non serve che vengano preservate tra le calls
- \$s0 - \$s7 (16-23): chiamati **caller-saved register** utilizzati per contenere valori per un tempo lungo, valori che devono essere preservati attraverso le calls.
- \$gp (28): puntatore globale che punta a metà di un blocco di memoria da 64K
- \$sp (29): stack pointer, punta l'ultima posizione sullo stack
- \$fp (30): frame pointer
- \$ra (31): return address from a procedure call (l'istruzione jal scrive in questo registro)

### 6.3.2 Convenzioni sui registri t e s

**I registri \$t (temporary) non sono salvati dalla procedura**

- il chiamante non si può aspettare di trovare immutati i contenuti dei registri \$t dopo una chiamata a procedura
- I contenuti dei registri \$t devono essere salvati dal chiamante prima della chiamata a procedura

**I registri \$s (saved) sono salvati dalla procedura**

- Il chiamante ha il diritto di aspettarsi che i contenuti dei registri \$s siano immutati dopo una chiamata a procedura
- Se la procedura usa i registri \$s deve salvarne il contenuto all'inizio e ripristinarlo prima del ritorno

Il contenuto dei registri \$s viene salvato sullo **stack**.

**Procedura foglia e non foglia** Una procedura foglia non chiama altre procedure, mentre una procedura NON foglia chiama altre procedure. Se una procedura chiama un'altra procedura è necessario salvare il contenuto dei registri \$ ra e \$s sullo stack.

Per convenzione i registri \$a0 - \$a3 sono utilizzati per il passaggio dei parametri.

Mentre gli indirizzi \$v0 - \$v1 sono utilizzati per la restituzione dei risultati. Dal punto di vista hardware sono registri come tutti gli altri, ma il loro utilizzo per il passaggio di parametri e risultati è una convenzione programmatica che deve essere rispettata per consentire di scrivere procedure che possono essere scritte senza bisogno di sapere come è fatto il programma che le chiama e possono essere chiamate senza bisogno di sapere come sono fatte "dentro".

**NB un parametro può essere un dato o un indirizzo!**

### I 6 passi di una procedura

- Setting dei parametri in un luogo accessibile alla procedura
- Trasferire il controllo alla procedura e salvare l'indirizzo dell'istruzione dove tornare dopo la chiamata della procedura (usare jal)
- Acquisire risorse per l'esecuzione della procedura
- Eseguire il compito richiesto
- Mettere il risultato in un luogo accessibile al chiamante
- Restituire il controllo al punto di partenza (usare l'istruzione jr)

# Capitolo 7

## ASM - Catena Programmatica + Linguaggio Assembly

### 7.1 Linguaggio Assembler - Vantaggi e Svantaggi

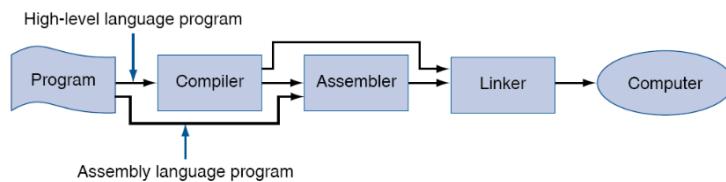
**Vantaggi** La dipendenza dall’architettura del calcolatore porta a numerosi vantaggi

- Più efficienza
- Programmi potenzialmente più compatti
- Massimo sfruttamento delle potenzialità dell’hardware sottostante
- Molto importante per programmare controller di processi e macchinari o per apparati limitati (embedded)

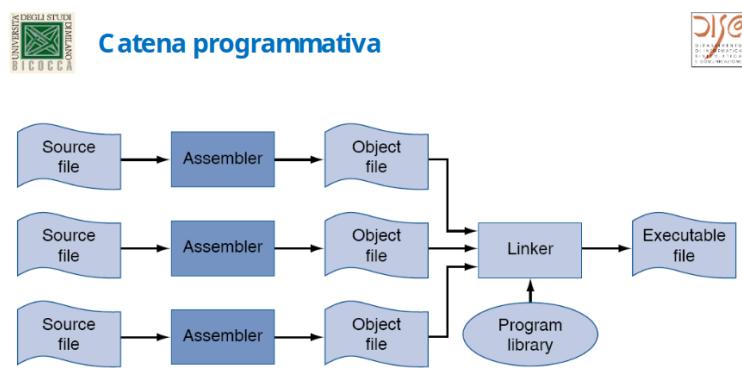
#### Svantaggi

- meno espressività (strutture di controllo limitate)
- Necessario conoscere i dettagli dell’architettura
- Mancaza di portabilità su architetture diverse
- Difficoltà di comprensione
- Lunghezza maggiore dei programmi

## 7.2 Compiler, Assembler, Linker



**FIGURE A.1.6** Assembly language either is written by a programmer or is the output of a compiler.



**FIGURE A.1.1** The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

**Debugger** Consente di eseguire il programma in modo controllato per la ricerca di un bug. Programma eseguito passo per passo e ispezione del valore di variabili ed espressioni. Possibilità di interrompere in punti predefiniti (breakpoint) e di interruzione in caso di modificati del valore di determinate variabili (watchpoint). Visualizzazione degli indirizzi di memoria delle variabili o delle istruzioni.

### 7.2.1 Compilatore

Un programma ad alto livello viene tradotto nel linguaggio assembly utilizzando un apposito programma detto compilatore. Dopo la fase di compilazione, il programma scritto in linguaggio assembly viene tradotto in linguaggio macchina da un apposito programma **l'assemblatore**.

Spesso con il termine compilazione si indica l'intero processo di traduzione da linguaggio ad alto livello a linguaggio macchina (essendo l'assemblatore spesso integrato con il compilatore).

### 7.2.2 Assembler

- Converte un programma assembler (file sorgente) in linguaggio macchina (file oggetto).
- Gestisce le etichette
- Gestisce pseudoistruzioni
- Gestisce numeri in base diverse

### 7.2.3 Il processo di assemblaggio

L’assemblaggio è un procedimento sequenziale che esamina, riga per riga il codice sorgente Assembly, traducendo ciascuna riga in un’istruzione del linguaggio macchina. Viene applicato modulo per modulo al programma e costituisce per ogni modulo la tabella dei simboli del modulo. 2 passi importanti

- Traduce i codici mnemonici (simbolici) delle istruzioni nei corrispondenti codici binari
- Traduce i riferimenti simbolici (variabili, registri, etichete di salto, parametri) nei corrispondenti indirizzi numerici.

Poichè le etichette di salto generano il problema dei riferimenti in avanti (ossia, riferimenti ad etichette successive o contenute in altri file), l’assemblatore deve leggere il programma sorgente due volte.

Ogni lettura del programma sorgente è chiamata passo e l’assemblatore è chiamato traduttore a 2 passi.

Ogni modulo assemblato di default parte dall’indirizzo 0

**Tabella dei simboli** Contiene i riferimenti simbolici presenti nel modulo da tradurre e al termine del primo passo, conterrà gli indirizzi numerici di tutti i simboli, tranne quelli esterni al modulo in esame. Per le etichette associate a direttive dell’assemblatore che definiscono costanti simboliche viene creata la coppia ”etichetta, valore” e in ogni istruzione che fa riferimento al simbolo viene sostituito il valore.

Discorso simile per le etichette che definiscono variabili (spazio di memoria + eventuale inizializzazione). In questo caso l’assemblatore riserva spazio, eventualmente inizializza la zona di memoria e crea nella tabella la coppia ”etichetta, indirizzo”. In ogni istruzione fa riferimento al simbolo (all’etichetta), il simbolo viene sostituito con l’indirizzo. Nelle etichette presenti nelle

istruzioni di salto, l'assemblatore deve generare un riferimento all'indirizzo dell'istruzione destinazione di salto.

**Osservazioni** Le etichette esterne (global) al modulo possono essere usate da moduli esterni, mentre le etichette interne (local) sono visibili solo all'interno di un modulo.

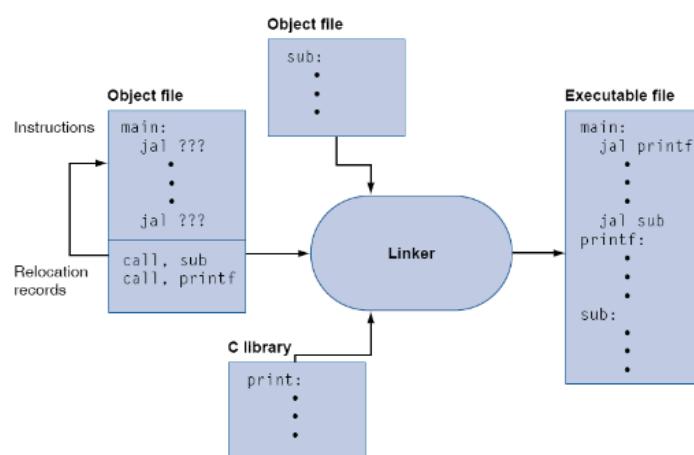
Le etichette esterne a un modulo rimangono non risolte, ma l'assembler non è disturbato da questo aspetto, dato che esse saranno risolte poi dal **linker**.

#### 7.2.4 Linker (Link editor)

Dopo che l'assembler ha tradotto il sorgente in vari moduli non resta che integrarli fra di loro e creare una singola unità eseguibile, a questo ci pensa il **linker**. Il linker principialmente effettua le seguenti operazioni:

- Inserisce in memoria in modo simbolico il codice e i moduli dati
- Determina gli indirizzi dei dati e delle etichete che compaiono nelle istruzioni
- Corregge i riferimenti interni ed esterni e risolve i riferimenti in sospeso (a etichette esterne)
- Genera il file eseguibile

Quindi ora abbiamo il file in linguaggio macchina pronto per essere eseguito.



**FIGURE A.3.1** The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

### 7.2.5 Loader

Quando eseguiamo un programma interviene il **loader**. È la parte che si occupa del caricamento del programma e delle relative librerie in memoria, prepara inoltre l'esecuzione da parte del sistema operativo. Nel dettaglio:

- Viene letta l'intestazione del file eseguibile per determinare la lunghezza del segmento di testo (cioè delle istruzioni) e del segmento dati (cioè le variabili)
- Viene creato uno spazio di indirizzamento sufficiente a contenere testo e dati
- Vengono copiate delle istruzioni e dati dal file eseguibile in memoria
- Vengono copiati nello stack eventuali parametri passati al programma principale
- Inizializzati i registri e impostato lo stack pointer affinchè punti alla prima locazione libera
- Salto a una procedura di startup la quale copia i parametri nei registri argomento e chiama la procedura principale del programma
- Quando la procedura principale restituisce il controllo, la procedura di startup termina il programma con una chiamata alla funzione di sistema exit

L'assemblatore riceve delle direttive in fase di assemblaggio, queste non corrispondono a istruzioni macchina e servono per:

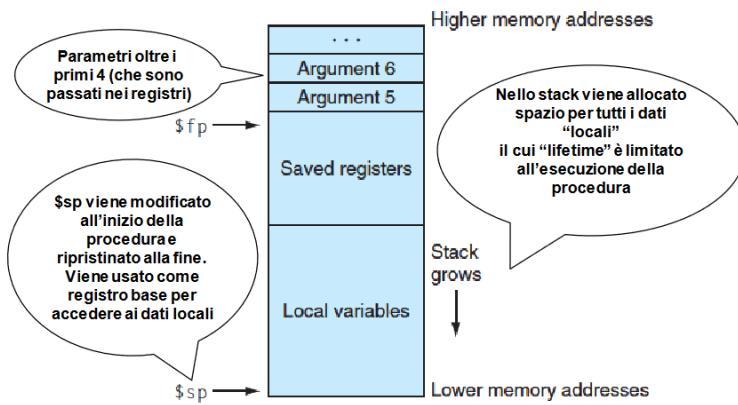
- associare etichette simboliche a indirizzi
- allocare spazio di memoria per le variabili
- decidere in quali zone di memoria allocare istruzioni e dati

### 7.2.6 Pseudoistruzioni

Istruzione assembly che non ha una corrispondente istruzione macchina, viene tradotta dall'assembler in una sequenza di istruzioni.

## 7.3 Uso dello stack

- Appendice A Hennessy-Patterson sez. A5 Fig. A.5.1



Lo stack è l'area di memoria dove vengono salvati gli indirizzi delle procedure che devono essere richiamate. Cosa fa la procedura? Alloca spazio nello stack

- Decrementa \$sp per lasciare in stack lo spazio necessario al salvataggio (1 word per ciascun registro da salvare)
- Salva \$ra
- Salva eventuali altri registri usando \$sp come registro base
- Ripristina i registri
- Incrementa \$sp per riportarlo alla situazione inizializzazione
- Jr \$ra (ritorno alla procedura)

I registri che interagiscono con lo stack sono:

- \$sp che punta alla prima parola del frame
- \$fp che punta all'ultima parola del frame

Un frame di solito è multiplo della parola doppia (8 byte).

Esempio: un frame di 32 byte

```

addi $sp, $sp, -32 # frame di stack di 32 byte
addi $fp, $sp, 28 # imposta il frame pointer
sw $ra, 0($fp) # salva l'indirizzo di ritorno
                # come primo word nel frame sullo stack

```

### 7.3.1 Operazioni procedura chiamante

Prima di chiamare una procedura, la procedura chiamante deve eseguire i seguenti passi al fine di non avere perdita di informazione:

- Impostare gli argomenti da passare alla procedura in \$a0 - \$a3, eventuali altri argomenti sono nella memoria o nello stack
- Salvare eventualmente i registri \$a0 - \$a3 e \$t0 - \$t7 in quanto la procedura chiamata può usare liberamente questi registri
- Infine chiamare la procedura tramite istruzione *jal "nome procedura"*

### 7.3.2 Operazioni procedura chiamata

La procedura chiamata chiamata deve eseguire i seguenti passi **appena viene chiamata**

- Allocare il suo stack frame (\$sp = \$sp - dimensione frame procedura)
- Salvare i valori disponibili nei registri \$s0 - \$s7, \$fp, \$ra se intende usare tali registri per la sua esecuzione, se per esempio la procedura non chiama un'altra procedura non è necessario salvare il registro \$ra
- Settare il frame pointer (che indica l'indirizzo dell'ultima parola del frame) \$fp = \$sp - dimensione frame procedura + 4

Quando **ha finito la sua esecuzione** deve eseguire i seguenti passaggi

- Mettere il valore di ritorno nei registri \$v0, \$v1
- Ripristinare i valori dei registri salvati sullo stack (\$s0 - \$s7, \$fp, \$ra)
- Liberare lo spazio sullo stack: \$sp = \$sp + dimensione frame procedura
- Eseguire l'istruzione *jt \$ra*

## 7.4 Syscall

In un sistema reale esiste il Sistema Operativo che è un insieme di programmi che:

- stanno in un'area (protetta) di memoria
- svolgono funzioni di utilità generale (in particolare I/O) richiamabili dai programmi utente

Qui tratteremo i meccanismi base di chiamata al SO. Il simulatore MIPS fornisce alcune funzioni elementare che simulano alcune funzionalità base del SO, richiamabili attraverso il meccanismo di **syscall**, concettualmente analogo a una chiamata a procedura.

Per effettuare una syscall è sufficiente inserire nel registro \$v0 il codice della syscall e nei registri \$a0-\$a3 gli argomenti.

### Syscall essenziali

- exit2 codice 17: terminazione del programma
- read\_int
- print\_int codice 1 (parametro passato per valore)
- read\_string
- print\_string (parametro passato per indirizzo)

Service	System call code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$a0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

## 7.5 Domande Esame

### 7.5.1 Descrivere i passi per la gestione delle etichette irrisolte e il ruolo del linker in questa fase

L'assembler traduce un file in linguaggio assembly in un file binari in linguaggio macchina e dati binari. La traduzione si divide in 2 parti:

1. Trovare spazi di memoria attraverso etichette così che la relazione tra nomi simbolici e indirizzi sia nota quando le istruzioni verranno tradotte
2. Tradurre ogni riga di codice assembly associando il corrispondente numero di opcodes, registri e etichette in un'istruzione corretta.

Il file prodotto dall'assembler sarà un file oggetto contenente le istruzioni macchina e i dati. Tuttavia il file non è ancora eseguibile dato che fa riferimento a procedure o dati contenuti in altri file. Un'etichetta (label) è esterna (global) se l'oggetto etichettato può far riferimento anche in altri file oltre a quello in cui l'etichetta è definita. Viceversa è locale se ci si può riferire ad essa solo in locale. Le **etichette locali (local labels)** nascondono il proprio nome dato che non devono essere viste esternamente, questo permette al compilatore di non aver bisogno di produrre nomi univoci in tutti i file.

Se una linea inizia con una label, l'assembler registra nella **symbol table** il **nome dell'etichetta e l'indirizzo di memoria della word che l'istruzione occupa**. L'assembler quindi calcola quante words of memory l'istruzione occuperà. Dopo aver registrato tutte le label e gli indirizzi l'assembler inizia il secondo passaggio, qui vengono usate le informazioni della **symbol table** per produrre codice assembly con tutti i riferimenti registrati nella tabella, i riferimenti esterni restano irrisolti.

**Object File Format** Quindi ora ci troviamo con degli object files contenenti 6 sezioni distinte

1. l'object file header, che descrive la dimensione e la posizione delle altre parti del file
2. il **text segment** che contiene il codice in linguaggio macchina per le routines del file sorgente. Queste routines potrebbero essere ineseguibili a causa dei riferimenti irrisolti.
3. Il **data segment** che contiene la rappresentazione binaria dei dati del file sorgente. Come per il text segment anche qua i dati potrebbero essere incompleti a causa dei riferimenti irrisolti.
4. **relocation information:** identifica istruzioni e data words che dipendono da indirizzi assoluti. Questi riferimenti devono cambiare se il programma è spostato in memoria.
5. **symbol table** che associa gli indirizzi con le etichette esterne nel file sorgente e lista i riferimenti irrisolti

6. debugging information: informazioni su come il programma è stato compilato

La produzione di diversi file oggetto è causata dalla compilazione separata, che permette di dividere un programma in pezzi che vengono immagazzinati in diversi file, questo permette di trattare il programma in moduli che possono essere compilati indipendentemente. Lo strumento che unisce insieme questi file è chiamato **linker** ed esegue 3 operazioni:

1. Cerca fra le librerie dei programmi per trovare le routines utilizzata dal programma in oggetto
2. Determina la locazione che il codice proveniente da ogni modulo occuperà e rialloca le istruzioni sistemando i riferimenti assoluti
3. Risolve i riferimenti fra i files

L'obiettivo principale del linker è quindi assicurarsi che il programma in questione non contenga etichette indefinite. Per effettuare questa risoluzione verifica se i file oggetto abbiano deli riferimenti a etichette con lo stesso nome. Riferimenti irrisolti indicano che il simbolo è usato, ma non definito nel programma. Per quanto riguarda le librerie il linker cerca nelle librerie di sistema per verificare se ci sono subroutine predefinite e strutture dati a cui il programma fa riferimento. Quando un programma usa una libreria il linker estrae il suo routine's code della libreria e lo incorpora nel text segment del programma. Queste nuove routine possono dipendere da altre librerie per questo il linker continua a cercare tra le altre librerie.

**File eseguibile** Se tutti i riferimenti sono risolti allora il linker determina lo spazio di memoria che occuperà ogni modulo, dato che l'assembler compilando i moduli singolarmente non può sapere quando spazio occuperanno in totale. Quando il linker alloca i moduli in memoria tutti i riferimenti assoluti vanno rallocati per far riferimento alla loro reale locazione. Infine viene prodotto un **file eseguibile** che può essere eseguito su un computer.

# **Capitolo 8**

## **Datapath**

Il datapath è un insieme di unità di calcolo, come ad esempio ALU, i registri e i moltiplicatori necessari per l'esecuzione delle istruzioni nella CPU. Il passaggio di due operando attraverso la ALU e la memorizzazione del risultato in un nuovo registro viene detto ciclo di data path. Tale ciclo definisce ciò che è in grado di fare una macchina: più veloce è questo ciclo, più veloce sarà la macchina.

### **8.1 Realizzare un datapath**

- Si stabilisce il set di istruzioni da implementare
- Si identificano i componenti del datapath (Alu, register file, ecc)
- Si stabilisce la metodologia di clocking
- Si assembla il datapath e si identificano i segnali di controllo
- Si analizza l'implementazione di ogni istruzione per determinare il setting dei segnali di controllo
- Si analizza l'implementazione di ogni istruzione per determinare il setting dei segnali di controllo
- Si assembla la logica di controllo

## 8.2. PASSI PER L'ESECUZIONE DI UNA ISTRUZIONE - FETCH DECODE EXECUTE63

### 8.2 Passi per l'esecuzione di una istruzione - Fetch Decode Execute

#### Cosa ne pensa il libro

1. Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
2. Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require that we read two registers.

#### Fetch

- Legge l'istruzione dalla memoria e la salva in un registro dedicato (Instruction Register)
- L'indirizzo di memoria che indica l'istruzione da leggere si trova nel registro Program Counter (PC)
- Dopo la lettura dell'istruzione in PC, viene incrementato di 3 per indicare la prossima istruzione (usando l'ALU)

#### Decode

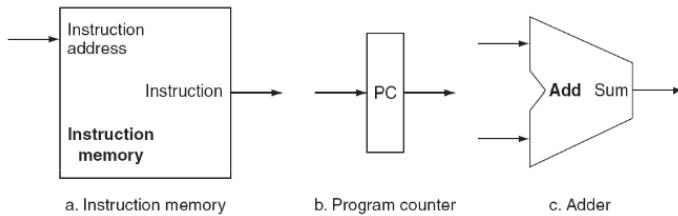
Decodifica i vari campi dell'istruzione per decidere quali sono i passi necessari per la sua esecuzione

#### Execute

Esegue i passi necessari per eseguire l'istruzione

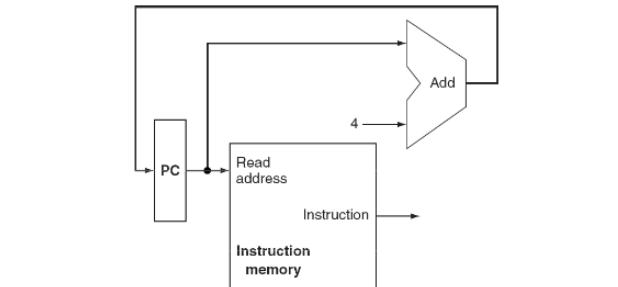
### 8.3 Implementazione Fetch

- Cosa serve per implementare il fetch?



**FIGURE 5.5 Two state elements are needed to store and access instructions, and an adder is needed to compute the next instruction address.** The state elements are the instruction memory and the program counter. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.) The program counter is a 32-bit register that will be written at the end of every clock cycle and thus does not need a write control signal. The adder is an ALU wired to always perform an add of its two 32-bit inputs and place the result on its output.

- Calcolo:  $PC+4$



**FIGURE 5.6 A portion of the datapath used for fetching instructions and incrementing the program counter.** The fetched instruction is used by other parts of the datapath.

Ciclo che ci permette in sostanza di caricare l'indirizzo di memoria iniziale e poi incrementare il PC di 4 per passare all'istruzione successiva

### Decode

Il processore MIPS legge i vari campi dell'istruzione e identifica il tipo di istruzione da eseguire (OPCODE e FUNC CODE se necessario)

### Execute

### Clocking

Nello schema è implicito il clock, ci sono 2 metodologie principali

### Singolo ciclo

- Ciclo singolo di lunghezza fissa uguale al tempo necessario per eseguire l'istruzione più lunga
- Ogni istruzione viene eseguita in un ciclo di clock
- Svantaggi: spreco di tempo, perché per far sì che tutte le istruzioni vengano eseguite in un clock solo devo avere un clock più lento.

Inoltre consideriamo come metodologia utilizzata la edge-triggered clocking methodology. Questo consente di effettuare i cambiamenti di stato su ogni picco del clock, è possibile quindi effettuare operazioni di lettura e scrittura sullo stesso registro all'interno dello stesso clock cycle.

### Codici controllo ALU

La ALU ha 4 input di controllo. Questi bit non sono codificati. Sono 6 delle 16 possibili combinazioni sono utilizzate per questo subset. La ALU MIPS ha le seguenti 6 combinazioni.

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

Dipendentemente dalla classe di istruzioni la ALU avrà bisogno di eseguire una di queste 5 funzioni (NOR serve per altre parti del MIPS instruction set). Per una load e una store word utilizziamo la ALU per calcolare l'indirizzo tramite add. Per le R-type la ALU necessita di una delle 5 funzioni (AND, OR, subtract, add o set on less than). Per Branch-equal la ALU necessita di utilizzare la subtract.

Come vediamo nelle seguenti tabelle quando ALUOp ha codice 10 per determinare che funzione deve utilizzare la ALU si considerano i bit del funct field (quindi significa che ho un'istruzione R-type). Quando il codice di ALUOp è diverso da 10 allora i bit del campo funct field non mi interessano, dato che non influenzano il risultato ("don't care term").

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	and	0000
R-type	10	OR	100101	or	0001
R-type	10	set on less than	101010	set on less than	0111

**FIGURE 5.12 How the ALU control bits are set depends on the ALUOp control bits and the different function codes for the R-type instruction.** The opcode, listed in the first column, determines the setting of the ALUOp bits. All the encodings are shown in binary. Notice that when the ALUOp code is 00 or 01, the desired ALU action does not depend on the function code field; in this case, we say that we “don’t care” about the value of the function code, and the funct field is shown as XXXXXX. When the ALUOp value is 10, then the function code is used to set the ALU control input.

ALUOp		Funct field							Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	0	X	X	X	X	X	X	0010	
X	1	X	X	X	X	X	X	0110	
1	X	X	X	0	0	0	0	0010	
1	X	X	X	0	0	1	0	0110	
1	X	X	X	0	1	0	0	0000	
1	X	X	X	0	1	0	1	0001	
1	X	X	X	1	0	1	0	0111	

**FIGURE 5.13 The truth table for the three ALU control bits (called Operation).** The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don’t-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don’t-care terms and are replaced with XX in the truth table.

Qui di seguito un remind del formato delle istruzioni R-type, Load or Store e Branch.

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0
a. R-type instruction						
Field	35 or 43	rs	rt	address		
Bit positions	31:26	25:21	20:16	15:0		
b. Load or store instruction						
Field	4	rs	rt	address		
Bit positions	31:26	25:21	20:16	15:0		
c. Branch instruction						

**FIGURE 5.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats.** The jump instructions use another format, which we will discuss shortly. (a) Instruction format for R-format instructions, which all have an opcode of 0. These instructions have three register operands: rs, rt, and rd. Fields rs and rt are sources, and rd is the destination. The ALU function is in the funct field and is decoded by the ALU control design in the previous section. The R-type instructions that we implement are add, sub, and, or, and s<sub>lt</sub>. The shamt field is used only for shifts; we will ignore it in this chapter. (b) Instruction format for load (opcode = 35<sub>ten</sub>) and store (opcode = 43<sub>ten</sub>) instructions. The register rs is the base register that is added to the 16-bit address field to form the memory address. For loads, rt is the destination register for the loaded value. For stores, rt is the source register whose value should be stored into memory. (c) Instruction format for branch equal (opcode = 4). The registers rs and rt are the source registers that are compared for equality. The 16-bit address field is sign-extended, shifted, and added to the PC to compute the branch target address.

## Funzione Multiplexor e segnali di controllo

Qui di seguito è riportato lo schema del Datapath MIPS con aggiunti i Multiplexor e i segnali di controllo.

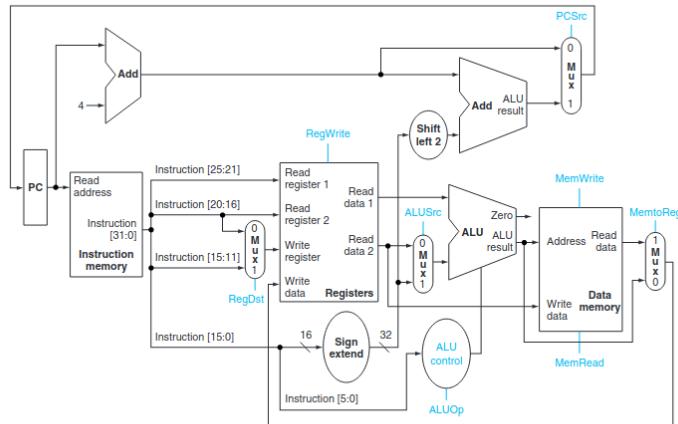


FIGURE 5.15 The datapath of Figure 5.12 with all necessary multiplexors and all control lines identified. The control lines are shown in color. The ALU control block has also been added. The PC does not require a write control, since it is written once at the end of every clock cycle; the branch control logic determines whether it is written with the incremented PC or the branch target address.

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from the rt field (bits 20:16).	The register destination number for the Write register comes from the rd field (bits 15:11).
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

FIGURE 5.16 The effect of each of the seven control signals. When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1. Otherwise, if the control is deasserted, the multiplexor selects the 0 input. Remember that the state elements all have the clock as an implicit input and that the clock is used in controlling writes. The clock is never gated externally to a state element, since this can create timing problems.

## Multi-ciclo

- Ciclo di lunghezza fissa più corretto

- Ogni istruzione viene eseguita in più cicli di clock
- Istruzioni di tipo diverso - eseguite in un numero di cicli di clock diverso
- Maggiore ottimizzazione rispetto al singolo ciclo
- Più complesso perchè devo orchestrare (unità di controllo) la gestione dei clock

## 8.4 Datapath multi-ciclo

Servono dei registri aggiuntivi (oltre ai 32 che abbiamo già visto)

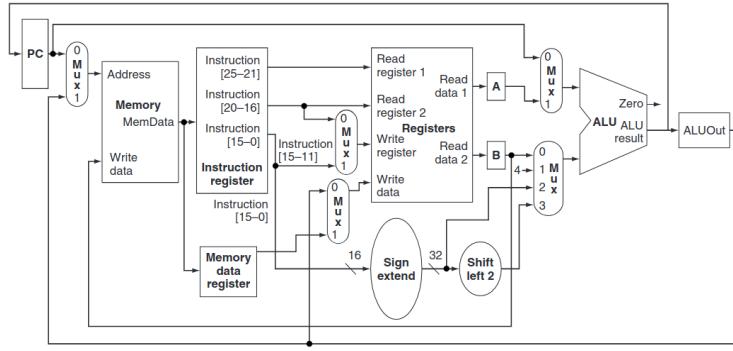
- Memorizzano valori intermedi che vengono usati nel ciclo di clock successivo per continuare l'esecuzione della stessa istruzione
- Instruction Register - Tiene in memoria la copia dell'istruzione che sto utilizzando
- MDR - Memory Data Register - Copia dei dati da elaborare prelevati dalla memoria
- A, B - registri tra register File e l'ingresso Alu - tengono in memoria i registri contenenti gli operandi
- ALUout - l'output della ALU (dato che l'ALU è un circuito combinatorio non è in grado di memorizzare un risultato)

Questo perchè non è più presente un ciclo di lunghezza pari all'istruzione più lunga, ma ho una lunghezza più breve, è necessario quindi spezzare l'istruzione in più cicli, preservando però le informazioni tra un ciclo e l'altro.

**1 ALU al posto di 3** Con la tecnica multiciclo è possibile eliminare i 2 adder e far svolgere le operazioni di add direttamente alla ALU. Per far questo è necessario aggiungere dei multiplexor.

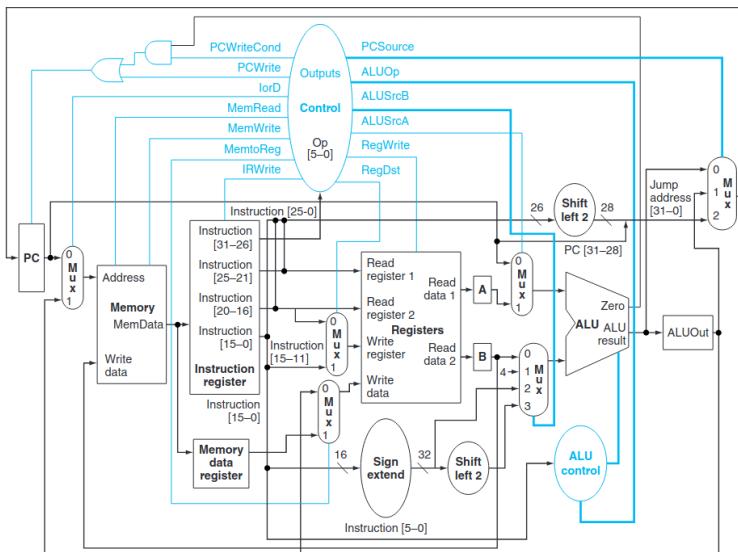
1. Per scegliere tra il valore A del registro o il PC
2. Il multiplexor della seconda ALU diventa da 2 a 4 vie - la prima via aggiuntiva è il numero costante 4 utilizzato per incrementare il PC, la seconda è relativa all'estensione di segno e lo shift utilizzato per le branch address.

Qui di seguito il Datapath senza control line e senza supporto alla jump



**FIGURE 5.26 Multicycle datapath for MIPS handles the basic instructions.** Although this datapath supports normal incrementing of the PC, a few more connections and a multiplexer will be needed for branches and jumps; we will add these shortly. The additions versus the single-clock datapath include several registers (IR, MDR, A, B, ALUOut), a multiplexor for the memory address, a multiplexor for the top ALU input, and expanding the multiplexor on the bottom ALU input into a four-way selector. These small additions allow us to remove two adders and a memory unit.

Qui di seguito sono state aggiunte le control line e il multiplexor per il supporto alla jump.



**FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines.** The control lines of Figure 5.27 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 5.27 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken. Support for jumps is included.

## 8.5 Dettaglio esecuzione istruzioni MIPS

Qualsiasi istruzione MIPS necessita dai 3 ai 5 step per essere completata.

- Fetch dell'istruzione e incremento  $PC + 4$ .  $IR = Memory[PC]$ ,  $PC = PC + 4$
- Decode: decodifica dell'istruzione E lettura registri E calcolo dell'indirizzo per un eventuale branch

- Execute: eseguire le operazioni relative a R-type OPPURE calcolo di memoria OPPURE completa branch OPPURE completa jump
- Execute: completa R-type OPPURE accesso alla memoria
- Execute: scrittura registro (solo per l'istruzione lw)

### 8.5.1 Fetch e Decode

Dato che non ho ancora capito che istruzione sto leggendo la Fetch e Decode vengono sempre eseguite nello stesso modo per tutte le istruzioni.

#### 1. Fetch

Nella Fetch salvo il PC nell'IR e incremento di 4 l'IR. Devo quindi settare l'ALU in modalità add.

**Dettaglio Fetch preso dal libro** To implement this step, we will need to assert the control signals MemRead and IRWrite, and set IorD to 0 to select the PC as the source of the address. We also increment the PC by 4, which requires setting the ALUSrcA signal to 0 (sending the PC to the ALU), the ALUSrcB signal to 01 (sending 4 to the ALU), and ALUOp to 00 (to make the ALU add). Finally, we will also want to store the incremented instruction address back into the PC, which requires setting PC source to 00 and setting PCWrite. The increment of the PC and the instruction memory access can occur in parallel. The new value of the PC is not visible until the next clock cycle. (The incremented PC will also be stored into ALUOut, but this action is benign.)

#### 2. Decode

Anche se non so ancora di che istruzione si tratta (dato che è proprio questo il passo che mi consentirà di capirlo) non posso effettuare letture che poi non posso scartare, dato che potrebbero darmi degli errori, posso però leggere dei dati che se mi servono (perchè sono corretti) posso tenere al limite verranno scartati dato che il multiplexor selezionerà solo le sezioni coerenti con l'istruzione in esecuzione. Quindi acquisisco i valori di Registro A e B e calcoliamo anche il branch target address con la ALU. In poche parole mi sto preparando sia per un'istruzione R-type che per una branch.

### 8.5.2 3. Execution, memory address computation, or branch completion

In questo caso l'esecuzione è determinata dall'OPCode dell'istruzione.

#### Memory Reference

$\text{ALUOut} \leq A + \text{sign-extend}(\text{IR}[15:0])$

Operation: The ALU is adding the operands to form the memory address. This requires setting ALUSrcA to 1 (so that the first ALU input is register A) and setting ALUSrcB to 10 (so that the output of the sign extension unit is used for the second ALU input). The ALUOp signals will need to be set to 00 (causing the ALU to add).

#### R-type (arithmetic-logical instruction)

$\text{ALUOut} \leq A \text{ op } B$

Operation: The ALU is performing the operation specified by the function code on the two values read from the register file in the previous cycle. This requires setting ALUSrcA = 1 and setting ALUSrcB = 00, which together cause the registers A and B to be used as the ALU inputs. The ALUOp signals will need to be set to 10 (so that the funct field is used to determine the ALU control signal settings).

#### Branch

if ( $A == B$ )  $\text{PC} \leq \text{ALUOut}$

Operation: The ALU is used to do the equal comparison between the two registers read in the previous step. The Zero signal out of the ALU is used to determine whether or not to branch. This requires setting ALUSrcA = 1 and setting ALUSrcB = 00 (so that the register file outputs are the ALU inputs). The ALUOp signals will need to be set to 01 (causing the ALU to subtract) for equality testing. The PCWriteCond signal will need to be asserted to update the PC if the Zero output of the ALU is asserted. By setting PCSource to 01, the value written into the PC will come from ALUOut, which holds the branch target address computed in the previous cycle. For conditional branches that are taken, we actually write the PC twice: once from the output of the ALU (during the Instruction decode/register fetch)

and once from ALUOut (during the Branch completion step). The value written into the PC last is the one used for the next instruction fetch.

### Jump

x, y is the Verilog notation for concatenation of bit fields x and y  
 $PC <= PC[31:28], (IR[25:0]), 2'b00;$

Operation: The PC is replaced by the jump address. PCSource is set to direct the jump address to the PC, and PCWrite is asserted to write the jump address into the PC.

### 8.5.3 4. Memory access or R-type instruction completion step

During this step, a load or store instruction accesses memory and an arithmetic-logical instruction writes its result. When a value is retrieved from memory, it is stored into the memory data register (MDR), where it must be used on the next clock cycle. Memory reference:

$MDR <= Memory[ALUOut];$

or

$Memory[ALUOut] <= B;$

Operation: If the instruction is a load, a data word is retrieved from memory and is written into the MDR. If the instruction is a store, then the data is written into memory. In either case, the address used is the one computed during the previous step and stored in ALUOut. For a store, the source operand is saved in B. (B is actually read twice, once in step 2 and once in step 3. Luckily, the same value is read both times, since the register number—which is stored in IR and used to read from the register file—does not change.) The signal MemRead (for a load) or MemWrite (for store) will need to be asserted. In addition, for loads and stores, the signal IorD is set to 1 to force the memory address to come from the ALU, rather than the PC. Since MDR is written on every clock cycle, no explicit control signal need be asserted.

**Arithmetic-logical instruction (R-type)**  $Reg[IR[15:11]] <= ALUOut;$   
 Operation: Place the contents of ALUOut, which corresponds to the output of the ALU operation in the previous cycle, into the Result register. The signal RegDst must be set to 1 to force the rd field (bits 15:11) to be used to select the register file entry to write. RegWrite must be asserted, and MemtoReg must be set to 0 so that the output of the ALU is written, as opposed to the memory data output.

## 5. Memory read completion step

During this step, loads complete by writing back the value from memory.

**Load**  $\text{Reg}[\text{IR}[20:16]] \leq \text{MDR};$

**Operation** Write the load data, which was stored into MDR in the previous cycle, into the register file. To do this, we set MemtoReg = 1 (to write the result from memory), assert RegWrite (to cause a write), and we make RegDst = 0 to choose the rt (bits 20:16) field as the register number. This five-step sequence is summarized in Figure 5.30. From this sequence we can determine what the control must do on each clock cycle.

### 8.5.4 Riassunto Esecuzione Istruzioni

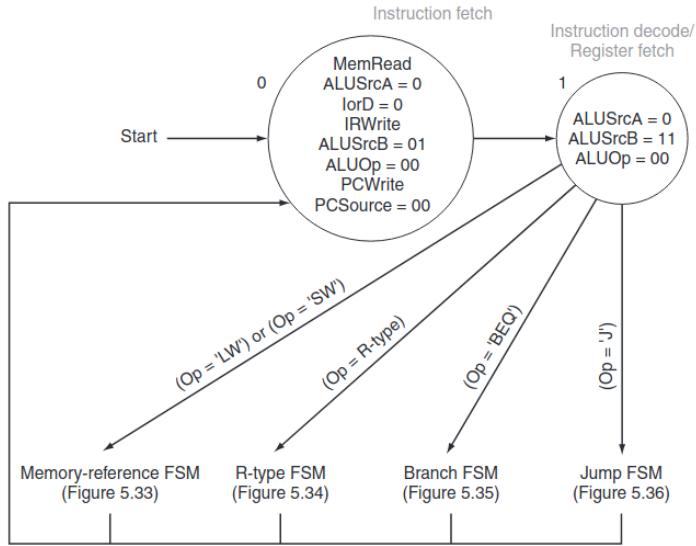
Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch		IR $\leftarrow \text{Memory}[\text{PC}]$ PC $\leftarrow \text{PC} + 4$		
Instruction decode/register fetch		A $\leftarrow \text{Reg}[\text{IR}[25:21]]$ B $\leftarrow \text{Reg}[\text{IR}[20:16]]$ ALUOut $\leftarrow \text{PC} + (\text{sign-extend}[\text{IR}[15:0]] \ll 2)$		
Execution, address computation, branch/jump completion	ALUOut $\leftarrow A \text{ op } B$	ALUOut $\leftarrow A + \text{sign-extend}[\text{IR}[15:0]]$	If (A == B) PC $\leftarrow \text{ALUOut}$ PC $\leftarrow (\text{PC}[31:28], [\text{IR}[25:0]], 2'b00)$	
Memory access or R-type completion	Reg [IR[15:11]] $\leftarrow$ ALUOut	Load: MDR $\leftarrow \text{Memory}[\text{ALUOut}]$ or Store: Memory [ALUOut] $\leftarrow B$		
Memory read completion		Load: Reg[IR[20:16]] $\leftarrow \text{MDR}$		

**FIGURE 5.30 Summary of the steps taken to execute any instruction class.** Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

## 8.6 Datapath: Automa per il controllo

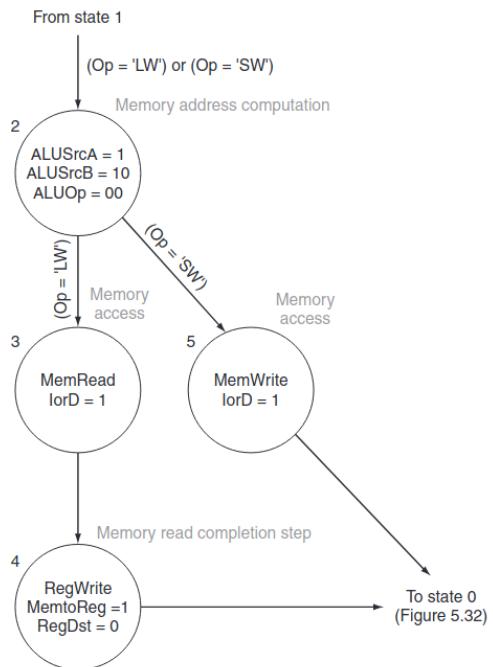
Un automa che ripete all'infinito la sua esecuzione, serve per controllare il datapath. Utilizzato per rappresentare l'unità dedicata ai segnali di controllo dei multiplexor, rappresenta nel dettaglio il ciclo Fetch Decode Execute, illustrando i 5 passaggi descritti nella sezione precedente.

Nell'automa a stati finiti ogni stato corrisponde a un clock.



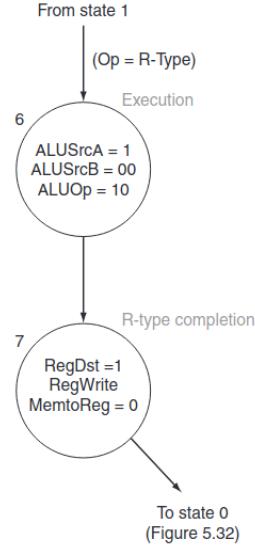
**FIGURE 5.32 The instruction fetch and decode portion of every instruction is identical.** These states correspond to the top box in the abstract finite state machine in Figure 5.31. In the first state we assert two signals to cause the memory to read an instruction and write it into the Instruction register (MemRead and IRWrite), and we set IorD to 0 to choose the PC as the address source. The signals ALUSrcA, ALUSrcB, ALUOp, PCWrite, and PCSource are set to compute PC + 4 and store it into the PC. (It will also be stored into ALUOut, but never used from there.) In the next state, we compute the branch target address by setting ALUSrcB to 11 (causing the shifted and sign-extended lower 16 bits of the IR to be sent to the ALU), setting ALUSrcA to 0 and ALUOp to 00; we store the result in the ALUOut register, which is written on every cycle. There are four next states that depend on the class of the instruction, which is known during this state. The control unit input, called Op, is used to determine which of these arcs to follow. Remember that all signals not explicitly asserted are deasserted; this is particularly important for signals that control writes. For multiplexors controls, lack of a specific setting indicates that we do not care about the setting of the multiplexor.

**Memory Reference** In questa sezione dell'automa notiamo che come nell'esecuzione la Load e la Store differiscono per il numero di passaggi eseguiti, infatti la Store ne richiede i totale 4 mentre la Load 5 (dato che deve leggere e poi scrivere un valore). Qui di seguito l'automa relativo a questa tipologia di istruzione.



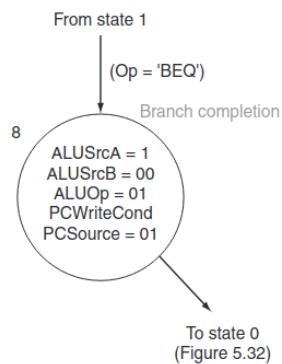
**FIGURE 5.33 The finite state machine for controlling memory-reference instructions has four states.** These states correspond to the box labeled “Memory access instructions” in Figure 5.31. After performing a memory address calculation, a separate sequence is needed for load and for store. The setting of the control signals ALUSrcA, ALUSrcB, and ALUOp is used to cause the memory address computation in state 2. Loads require an extra state to write the result from the MDR (where the result is written in state 3) into the register file.

**R-type instruction** In questo caso ho un singolo ramo dato che il numero di istruzioni da eseguire è identico per tutte le R-type e sarà il function code a comunicare alla ALU la funzione da eseguire.



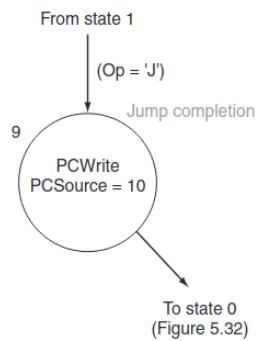
**FIGURE 5.34 R-type instructions can be implemented with a simple two-state finite state machine.** These states correspond to the box labeled “R-type instructions” in Figure 5.31. The first state causes the ALU operation to occur, while the second state causes the ALU result (which is in ALUOut) to be written in the register file. The three signals asserted during state 7 cause the contents of ALUOut to be written into the register file in the entry specified by the rd field of the Instruction register.

**Branch instruction** In questo caso è necessario un solo passaggio.



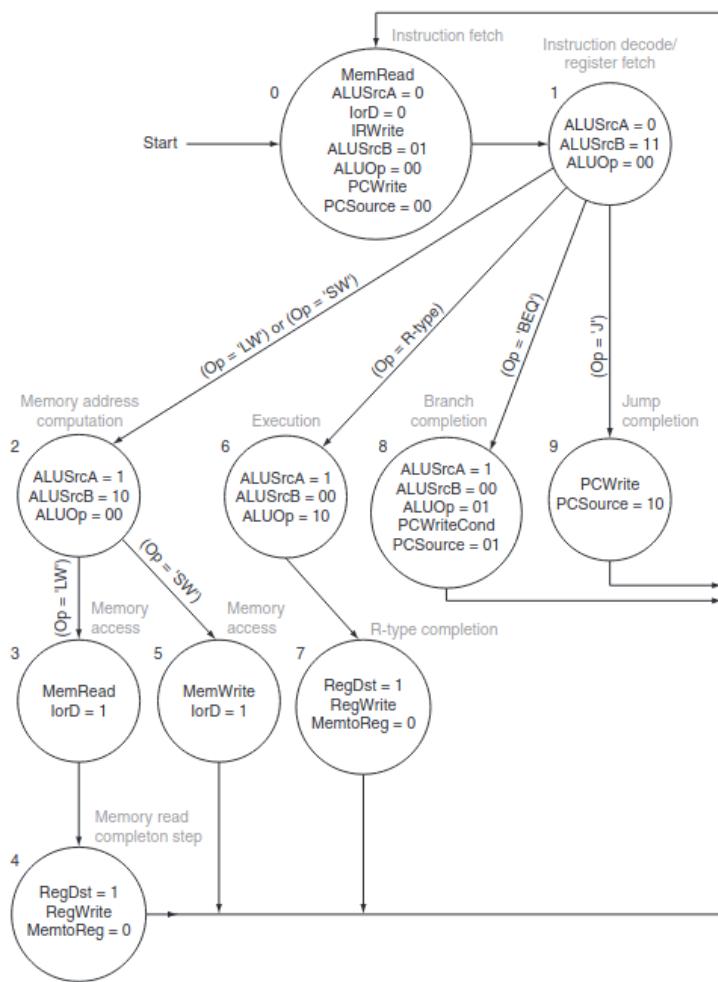
**FIGURE 5.35 The branch instruction requires a single state.** The first three outputs that are asserted cause the ALU to compare the registers (ALUSrcA, ALUSrcB, and ALUOp), while the signals PCSource and PCWriteCond perform the conditional write if the branch condition is true. Notice that we do not use the value written into ALUOut; instead, we use only the Zero output of the ALU. The branch target address is read from ALUOut, where it was saved at the end of state 1.

**Jump instruction** Anche in questo caso è necessario un solo passaggio dove asserisco lo shift dei due bit per la concatenazione con il PC.



**FIGURE 5.36** The jump instruction requires a single state that asserts two control signals to write the PC with the lower 26 bits of the instruction register shifted left 2 bits and concatenated to the upper 4 bits of the PC of this instruction.

## FSM control completo



**FIGURE 5.38 The complete finite state machine control for the datapath shown in Figure 5.28.** The labels on the arcs are conditions that are tested to determine which state is the next state; when the next state is unconditional, no label is given. The labels inside the nodes indicate the output signals asserted during that state; we always specify the setting of a multiplexer control signal if the correct operation requires it. Hence, in some states a multiplexer control will be set to 0.

## 8.7 Riassumendo il Datapath

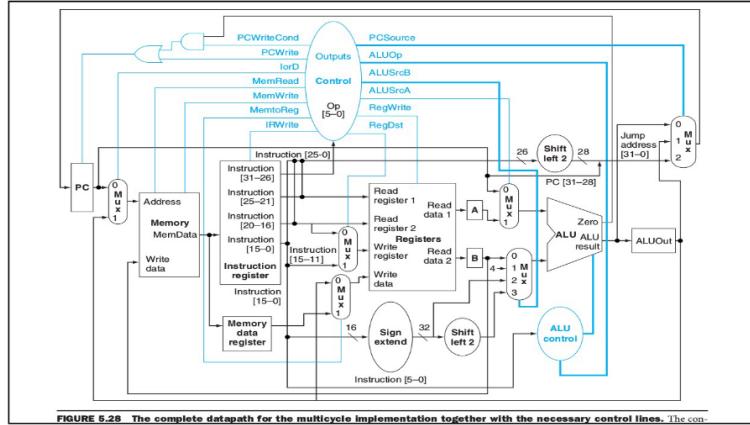


FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines. The con-

**Memoria** Contiene istruzioni e dati sulla base dell'indirizzo in input restituisce in output l'istruzione o il dato letto.

**Register File** 32 registri che contengono i dati utilizzati nel corso dell'esecuzione delle istruzioni, restituisce in output il contenuto dei due registri letti (indicati dai due input - ReadRegister1 e ReadRegister2) e scrivere, se il segnale di scrittura è attivo, il dato in input WriteData nel registro indicato dall'indirizzo in input (WriteRegister).

**ALU** per tutte le istruzioni incrementa il PC ( $PC+4$ ) e calcola il valore di branch (che sarà usato solo nel caso di istruzioni di branch). PC viene incrementato di 4 byte dato che le istruzioni hanno lunghezza 32 bit = 4 byte, per questo per passare alla prossima istruzione devo spostarmi di 32 bit sulla memoria per raggiungere l'inizio della prossima istruzione. Inoltre:

- Istr. R-Type: esegue operazioni aritmetico logiche su due operandi in funzione del "function code" indicato dai 6 bit meno significativi dell'istruzione
- Istr. accesso a memoria: calcola l'indirizzo di memoria cui accedere
- Istr. branch: confronta i due registri in input

**Sign Extended** : opera l'estensione di segno dai 16 bit in input ai 32 bit in output.

**Shift Left 2** opera uno shift a sinistra dei bit in input (con l'effetto di moltiplicare per 4 l'input)

### 8.7.1 Funzioni dei registri

**PC** : contiene i 32 bit che indicano l'indirizzo dell'istruzione da eseguire

**Instruction Register** Contiene i 32 bit che corrispondono alla codifica dell'istruzione prelevata da memoria per l'esecuzione

**Memory Data Register** contiene il dato letto da memoria prima della sua scrittura nel registro destinazione (e.g. nel caso di esecuzione di istruzione load)

**A e B** Contengono i valori letti dai registri del RegisterFile

**ALUOut** Contiene l'output della ALU

### 8.7.2 Ruolo di ciascuno dei multiplexer

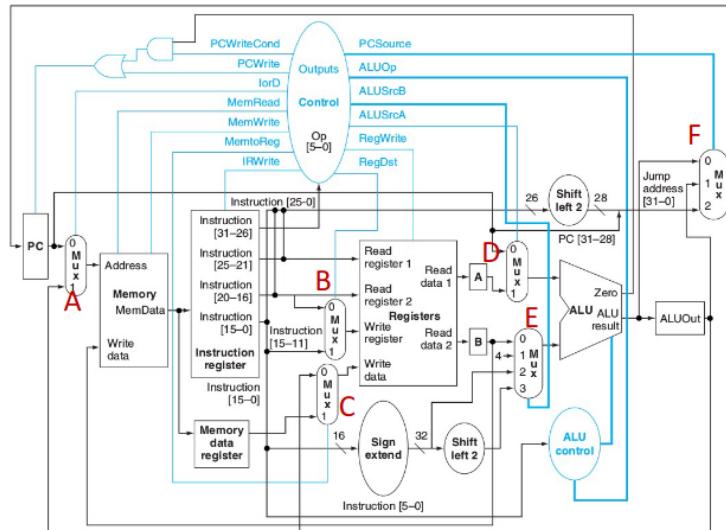


FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines. The con-

**A** seleziona l'indirizzo di memoria a cui accedere tra:

- PC (nel caso si voglia leggere un'istruzione)
- output della ALU (nel caso di esecuzione dell'istruzione "load")

**B** Seleziona il gruppo di bit dell'IR che indica il registro in cui scrivere i due tipi di istruzione I-Type (IT[20:16]) e R-type (IR[15:11])

**C** Seleziona la sorgente per il dato da scrivere in memoria tra:

- ALUOut (per istruzioni R-type)
- MemoryDataRegister (per istruzioni load)

**D** Seleziona il primo operando dell'operazione aritmetico-logica eseguita dalla ALU tra PC (per incremento +4) e registro A (per istruzioni R-type)

**E** Seleziona il secondo operando dell'operazione aritmetico-logica eseguita dalla ALU

- B (per istruzioni R-type)
- 4 (per aggiornamenti del PC)
- sign-extended IR[15:0] (per istruzioni di accesso a memoria - lw/sw)
- sign-extended 2-shifted (j)

**F** Seleziona il valore per l'aggiornamento del PC

- output dell'ALU: PC + 4
- contenuto di ALUOut: 2 shifted 26-bit beq field - 26 (jump address) + 2 (shift) + 4 (incremento PC) = 32
- jump target address (IR[25:0] shifted left 2 bits e concatenato con PC + 4)

### Segnali di controllo fetch, decode, execute

L'attivazione dei segnali di controllo per un datapath multyciclo può essere descritta da un **automa a stati finiti**.

I segnali di controllo (next state per l'automa) sono definiti **in funzione dello stato corrente e dell'Opcode**.

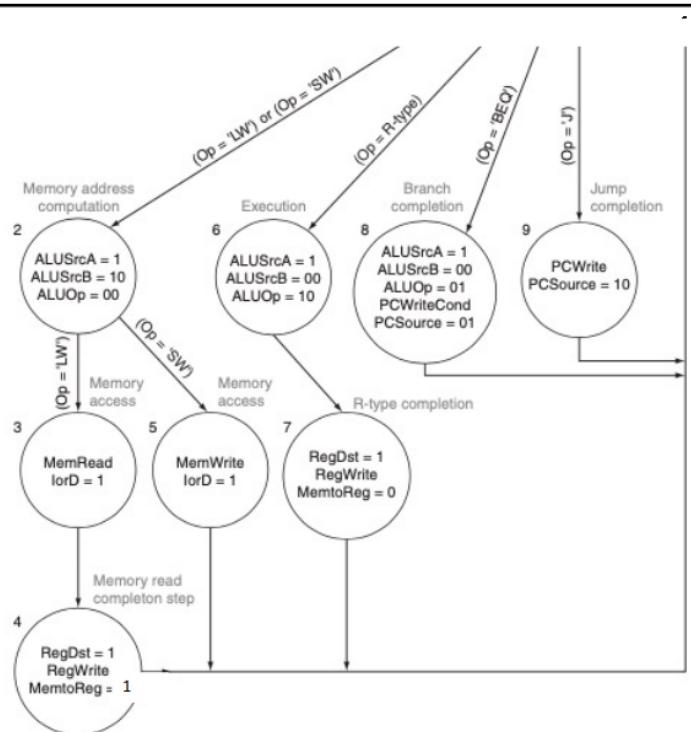
Per tutti i tipi di istruzioni, nella fase di fetch (stato 0) sono impostati i seguenti segnali:

- IorD = 0 per selezionare PC come sorgente per l'indirizzo di memoria

- MemRead: per leggere un'istruzione da memoria
- IRWrite per scrivere l'istruzione letta nell'Instruction register
- ALUSrcA, ALUSrcB, ALUOp, PCWrite e PCSource sono impostati per calcolare  $PC + 4$  e memorizzare il nuovo valore nel PC

**Per tutti i tipi di istruzioni**, nella fase di decode (stato 1) sono impostati i segnali ALUSrcA, ALUSrcB e ALUOp per calcolare il valore di branch  
Nella fase di execute sono impostati i segnali di controllo, **in funzione del tipo di istruzione in esecuzione**.

- Le istruzioni di **jump o branch**, dopo lo stato 1 richiedono un ciclo di clock per il loro completamento
- Le istruzioni di tipo R-type e le istruzioni di memorizzazione (store), richiedono due cicli di clock per l'esecuzione di una sequenza di due stati d'impostazione dei segnali
- Le istruzioni di lettura da memoria richiedono tre cicli di clock per l'esecuzione di una sequenza di tre stati d'impostazione dei segnali



## 8.8 Eccezioni

Durante l'esecuzione delle istruzioni si possono verificare eventi inattesi: eccezioni e/o interruzioni.

**Eccezione** Evento sincrono generato all'interno del processore e provocato da problemi nell'esecuzione di un'istruzione. Per esempio un overflow, una istruzione non valida, errori, pagine non presenti in memoria, ecc.

- Causate da eventi interno al processore
- Sincrone rispetto al programma in esecuzione
- La condizione di eccezione deve essere risolta da un gestore di eccezioni (es. handler)
- Se la condizione di eccezione è risolvibile il programma riprendere l'esecuzione, altrimenti il programma termina prima della sua fine.

**Interruzione** Evento asincrono che giunge dall'esterno del processore. Di solito arriva da un'unità di I/O utilizzato per comunicare alla CPU il verificarsi di certi eventi. Esempi: la terminazione di un'operazione di I/O la cui esecuzione era stata richiesta dalla CPU.

- Causate da eventi esterni al processore
- Asincrone rispetto al programma in esecuzione
- Sono gestite tra due istruzioni consecutive
- Si sospende l'esecuzione del programma utente, si gestisce l'interruzione e poi si riprende l'esecuzione del programma utente

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

### 8.8.1 Gestione di eccezioni e interruzioni

Il controllo del processore deve gestire gli eventi inattesi. Tutti i processori eseguono i seguenti passi per gestire un'eccezione/interruzione:

- Interruzione dell'esecuzione del programma corrente
- Salvataggio parziale dello stato in esecuzione corrente (e.g. PC) - per riprendere eventualmente l'esecuzione del programma corrente se possibile
- Salto a una routine del sistema operativo (SO) per gestire l'eccezione/interruzione (tale routine è chiamata di solito handler o gestore delle eccezioni)
- Esecuzione della routine del SO
- Se possibile, ripristino dello stato di esecuzione del programma e continuazione dell'esecuzione del programma

**Come capire l'evento inatteso verificatosi?** Due possibili soluzioni:

1. Indirizzo fisso - Registro dedicato (e.g., Cause) - il controllo della CPU, prima di saltare all'handler del SO (a un indirizzo fisso), deve salvare in un registro interno un identificatore numerico del tipo di eccezione verificatosi. L'handler accederà al registro interno per determinare la causa dell'eccezione
2. Interruzioni vettorizzate - esistono handler diversi per eccezioni/interruzioni differenti. Il controllo della CPU sceglie l'handler corretto, saltando all'indirizzo corretto. A questo scopo, viene predisposto un vettore di indirizzi, uno per ogni tipo di eccezione/interruzione, da indirizzare tramite il codice numerico dell'eccezione/interruzione

**MIPS** In MIPS viene adottata la prima soluzione, usando un registro, denominato Cause, per memorizzare il motivo dell'eccezione. L'indirizzo dell'istruzione corrente che ha causato l'eccezione viene salvato nel registro Exception Program Counter (EPC).

### Esempio di gestione delle eccezioni in MIPS

Passi da eseguire:

- Individuare l'evento inatteso, la causa dell'eccezione e salvarla in un registro dedicato denominato Cause
- interrompere l'esecuzione corrente
- Salvare l'indirizzo dell'istruzione corrente nel EPC (EPC = PC - 4)
- Saltare a un gestore delle eccezioni del SO che si trova a un indirizzo fisso per gestire l'eccezione

### Osservazioni

- Il MIPS non salva nessun altro registro oltre al PC
- è compito della routine salvare altre porzioni dello stato corrente del programma se necessario
- Esistono CPU dove questo salvataggio avviene prima di saltare alla routine (es approcci CISC)

### Necessario aggiungere 2 stati

- Devono salvare in EPC il valore PC - 4
- Salvare nel registro Cause la causa dell'eccezione (0 o 1 in questo caso perchè consideriamo 2 esempi di eccezioni)
- Salvare in PC l'indirizzo del gestore delle eccezioni

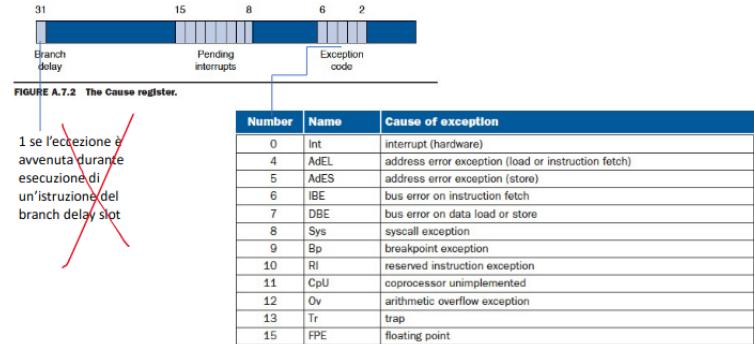
## Pratica MIPS

Per tutti i tipi di eccezione avviene

- Salvataggio dell'indirizzo dell'istruzione che ha causato l'eccezione (in EPC)
- Cambiamento del normale flusso di esecuzione di un programma: aggiornamento di PC con il valore 0x80000180 che corrisponde all'indirizzo dello spazio kernel text in cui ha inizio la procedura di gestore delle eccezioni - exception handler.

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

## Il registro Cause e possibile cause di eccezione



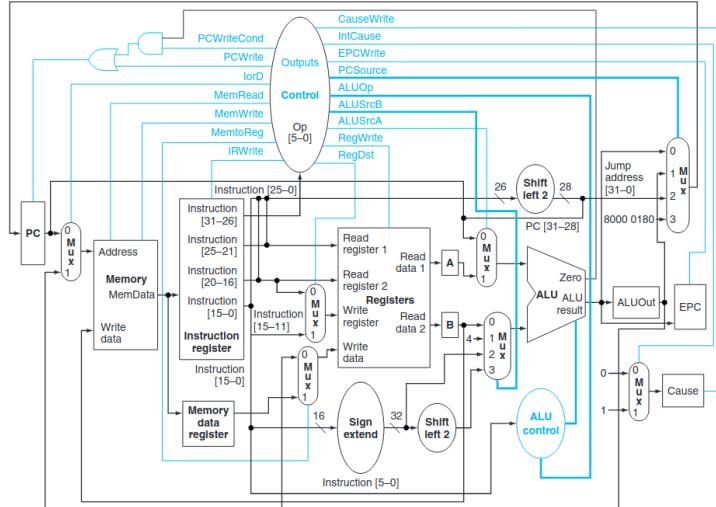
## Exception Handler

È la porzione di codice assembly che contiene il codice che gestisce le eccezioni e si trova alla locazione fissa 0x80000180 in area ktext riservata al sistema operativo. Quando si presenta un eccezione viene caricato nel PC l'indirizzo dell'Handler, così che venga eseguito il codice che si occupa di capire di che eccezione si tratta.

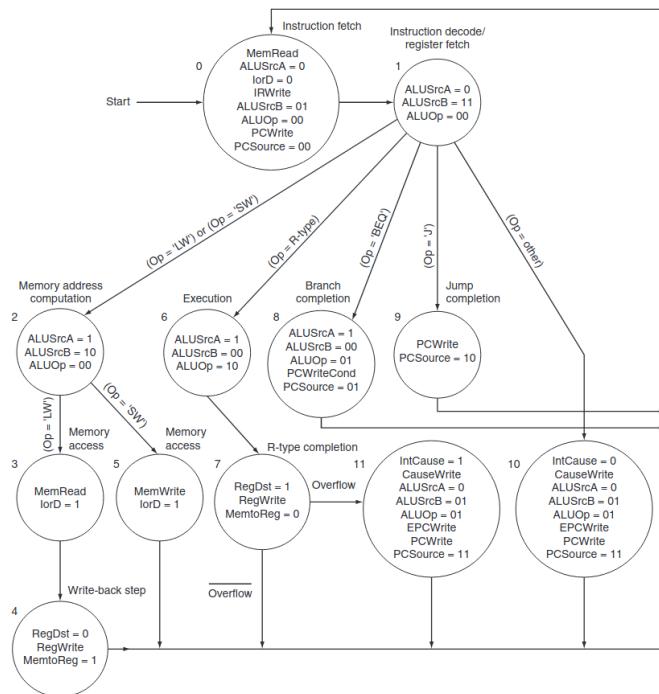
Dal momento che viene eseguito dopo l'interruzione di un programma utente in un punto qualsiasi, il programma è tenuto a **preservare TUTTI i registri macchina, NON è responsabilità dell'hardware**. Ad esso sono riservati i due registri \$k01, \$k02 che può utilizzare senza doverne preservare i contenuti. Al termine del trattamento dell'eccezione, l'exception handler ritorna il controllo al programma interrotto (istruzione eret). MIPS rileva e tratta eccezioni PRIMA del completamento dell'esecuzione della istruzione corrente.

- Se è una interruzione, dopo la sua gestione, l'esecuzione del programma deve riprendere dall'istruzione corrente (salvata nel registro EPC)
- se si tratta di una eccezione ed è possibile proseguire l'esecuzione deve riprendere dall'istruzione successiva (EPC+4).

### 8.8.2 Datpath e FSM con gestione eccezioni



**FIGURE 5.39** The multicycle datapath with the addition needed to implement exceptions. The specific additions include the Cause and EPC registers, a multiplexor to control the value sent to the Cause register, an expansion of the multiplexor controlling the value written into the PC, and control lines for the added multiplexor and registers. For simplicity, this figure does not show the ALU overflow signal, which would need to be stored in a one-bit register and delivered as an additional input to the control unit (see Figure 5.40 to see how it is used).



**FIGURE 5.40** This shows the finite state machine with the additions to handle exception detection. States 10 and 11 are the new states that generate the appropriate control for exceptions. The branch out of state 1 labeled ( $Op = \text{other}$ ) indicates the next state when the input does not match the opcode of any  $\text{lw}$ ,  $\text{sw}$ ,  $0$  (R-type),  $J$ , or  $\text{beq}$ . The branch out of state 7 labeled  $Overflow$  indicates the action to be taken when the ALU signals an overflow.

# Capitolo 9

## Gestione Input-Output

Con I/O definiamo l'insieme di architetture e dispositivi per il trasferimento di informazioni da e verso l'elaboratore. Si tratta di dispositivi eterogenei per

- Velocità di trasferimento
- Latenze
- Sincronizzazione
- Modalità di interazione (sia con l'uomo che con la macchina)

### 9.1 Bus di Sistema

Esistono vari tipi di bus nei computer di oggi. In questo corso tratteremo il bus di sistema che collega la CPU con la memoria e con le periferiche. Il bus di sistema è composto da:

- Bus di dati: le linee per trasferire dati e istruzioni da/verso dispositivi
- Bus di controllo: trasporta informazioni per la definizione delle operazioni da compiere per la sincronizzazione fra i dispositivi
- Bus degli indirizzi: la CPU trasmette gli indirizzi di memoria o di periferica che identificano i dati da leggere/scrivere dalla memoria/-periferiche

Tutte le unità dell'elaboratore sono connesse al bus

**Vantaggi** Elevata flessibilità, semplicità, basso costo

**Svantaggi** Gestione complessa del canale condiviso

## 9.2 Periferiche

Sono dispositivi per I/O di informazioni collegati alla CPU tramite il bus di sistema e/o interfacce, le interfacce sono standardizzate per la comunicazione e hanno una componente hardware (e.g. il controller della periferica) e una componente software (e.g. driver). La struttura di un'interfaccia contiene registri di dati (oppure buffer) e ha associato registri di stato.

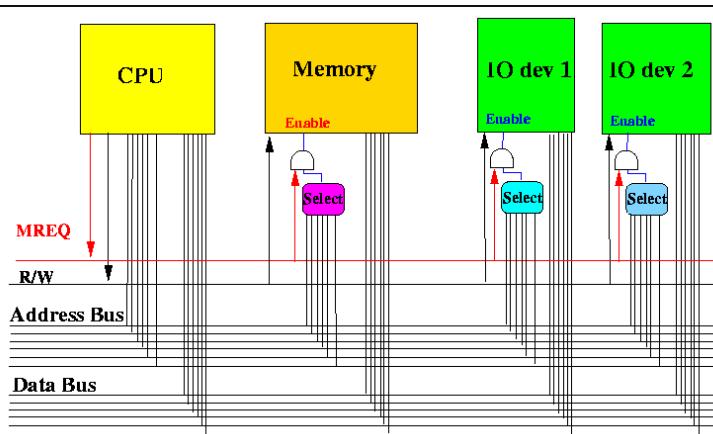
### Periferiche mappate in memoria

Una parte della memoria riservata al sistema viene usata per la comunicazione con le periferiche. Ogni periferica ha uno spazio dedicato nella memoria e viene assegnato un identificatore unico alla stessa (es. indirizzo unico).

I registri dell'interfaccia della periferica sono mappati in memoria, ma sono riservati, per questo non sono accessibili da un programma utente, per potervi accedere è necessario passare dal SO.

L'accesso alla periferica è simile all'accesso alla memoria (e.g. lw, sw) dal punto di vista della CPU. Non tutte le architetture degli elaboratori usano questa soluzione.

### Un selettore generale per mappare dispositivi in memoria



## Registri di interfaccia della periferica

- Il **registro di stato della periferica** rappresenta lo stato della stessa e viene letto dalla CPU. Un esempio di stato è "periferica pronta per ricevere dati", "dati richiesti disponibili per il trasferimento".
- Il **registro dei dati della periferica** rappresenta i dati di input o di output in base al tipo della periferica (di ingresso come per la tastiera o uscita come la stampante).

### 9.2.1 Passi di I/O

1. CPU interroga lo stato della periferica
2. La periferica restituisce il suo stato
3. Se la periferica è pronta per trasmettere/ricevere dati, la CPU richiede il trasferimento dei dati
4. La CPU invia o riceve i dati

### 9.2.2 Tecniche di gestione I/O

- I/O gestito da programma (Programmed I/O)
- I/O guidato da interrupt (Interrupt Driven I/O)
- Accesso Diretto alla Memoria (Direct Memory Access - DMA)

Per valutare le prestazioni del trasferimento dati è necessario ricorrere a delle misure che permettono di valutare l'efficienza della gestione delle operazioni di I/O.

- Banda passante - rappresenta la quantità dei dati che si può trasferire per unità di tempo; rappresenta una misura di flusso
- Latenza - rappresenta il tempo che intercorre tra l'istante in cui una periferica è pronta per il trasferimento e l'istante in cui il dato viene trasferito, è una misura di tempo

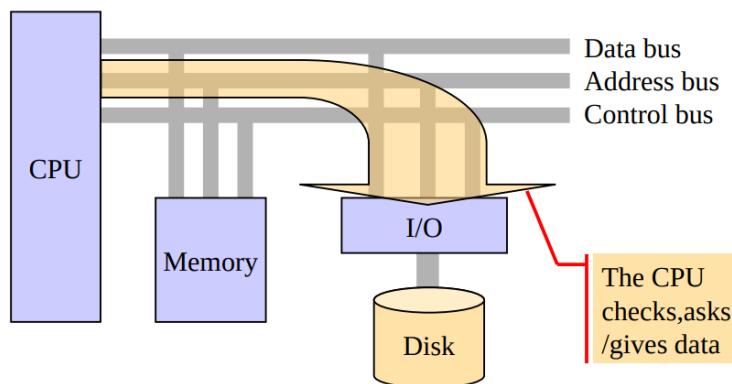
## I/O Gestito da programma

In questa casistica è il programma a gestire l'I/O e la periferica ha un ruolo passivo. La CPU si occupa sia del controllo sia del trasferimento dati, predisponendo il controllore della periferica all'esecuzione dell'I/O.

La CPU si ferma e interroga il registro di stato della periferica in attesa che sia pronta (e.g. ready bit assume un determinato valore).

**Vantaggio** Risposta veloce al ready bit

**Svantaggio** la CPU resta bloccata in stato di busy waiting



### Prestazioni I/O gestito da programma

- Banda passante alta perchè la CPU trasferisce subito il dato (molti dati per l'unità di tempo) e la gestione della periferica richiede poche istruzioni
- Latenza minima in quanto la CPU noterà subito lo stato della periferica (tempo massimo: un ciclo intero di busy waiting)
- CPU resta però in attesa che la periferica risponda, in uno stato di busy waiting

Esempio: se si trasferiscono più dati si ha un ciclo interno per il trasferimento di ogni byte e un ciclo esterno per il trasferimento di tutti i byte.

## I/O gestito da interrupt

L'Interrupt è un evento asincrono che genera l'interruzione del normale funzionamento del processore.

- La periferica segnala alla CPU di aver bisogno di attenzione mediante un segnale sul bus di controllo, tale segnale è una segnale **interrupt request**
- Quando il processore se ne accorge (in una fase di fetch) informa al periferica con un segnale interrupt acknowledge.
- La CPU interrompe l'esecuzione del programma corrente (salvando il contesto dell'esecuzione del programma per poter riprendere la sua esecuzione) ed esegue la procedura di risposta all'interrupt.
- Terminata l'esecuzione della procedura di interrupt, la CPU riprendere l'esecuzione del programma interrotto, infine il programma utente continua la sua esecuzione.

**Vantaggio** La CPU non fa più busy waiting come per I/O gestito da programma

**Svantaggio** La CPU deve comunque gestire le operazioni di trasferimento. Per evitare l'intervento della CPU nella fase di trasferimento dati è stato introdotto il protocollo di trasferimento DMA - Direct Memory Access.

Quando una periferica genera un interrupt la CPU esegue una serie di istruzioni predefinite (definite da chi ha creato il sistema) contenuta a partire dalla locazione di memoria prestabilita dentro quella dedicata al kernel che sono:

- Save che salva lo stato della computazione al momento dell'interrupt
- Identificazione della periferica interrompente
- Gestione della periferica, con il trasferimento del dato
- Restore che ripristina lo stato della computazione a prima dell'interruzione
- Ritorno dall'interrupt (eret)

### Prestazioni

- Banda passante: minor banda passante in quanto il trasferimento di ogni dato necessita di più tempo
- Latenza: maggior latenza per la maggior quantità di operazioni da eseguire

## DMA - accesso diretto alla memoria

Usato quando si trasferiscono velocemente grandi quantità di dati. Con **DMA** la periferica diventa autonoma negli accessi alla memoria dato che è essa a gestire i trasferimenti, non è più la CPU a intervenire. Necessità di 2 registri in più per ogni periferica oltre al registro di stato e registro dei dati.

- Un registro che indichi l'indirizzo di memoria da/dove trasferire i dati
- Un registro che indichi la quantità dei dati da trasferire

Anche questi 2 registri aggiuntivi sono **mappati in memoria**.

Alla fine del trasferimento la periferica invia un interrupt alla CPU per segnalare il completamento del trasferimento.

### Prestazioni

- Banda passante: massima perchè la CPU non deve eseguire nessuna istruzione
- Latenza: minima dato che nessuna istruzione è eseguita dalla CPU

## Implementazione DMA

4 Registri presenti nella periferica:

1. Registro controllo della periferica
2. Registro dati della periferica
3. Registro contenente la locazione di memoria corrente dalla quale trasferire i dati alla/dalla periferica
4. Registro quantità dei dati da trasferire

Tutti questi registri sono mappati in memoria e impostati dalla CPU prima del trasferimento dato che la periferica è autonoma.

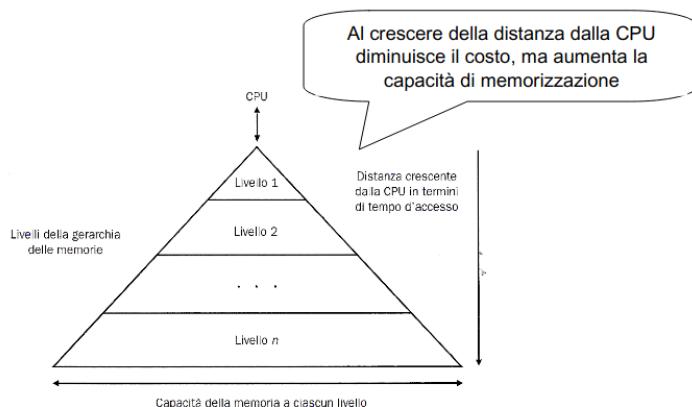
La **CPU si occupa solo di far partire il trasferimento** dato che la periferica è capace di controllare da sola se il trasferimento è ultimato o c'è altri dati, in quanto ha un contatore dei bit rimanenti. Una volta terminato il trasferimento è la periferica che notifica la CPU attraverso un interrupt.

Con il DMA si ha latenza minima e banda passante massima in quanto la CPU non deve eseguire istruzioni. L'attesa è quella dell'arbitraggio del bus, ovvero del blocco del bus da parte della CPU (nell'ordine del tempo di un'istruzione nel caso peggiore) e l'unica latenza è il tempo per ottenere l'accesso al bus.

# Capitolo 10

## Gerarchie di memoria e cache

L'ideale sarebbe avere a disposizione una quantità illimitata di memoria e contemporaneamente veloce, ma ciò non è possibile, dato che le memorie veloci costano molto di più a parità di bit rispetto alle memorie lente. Solitamente una memoria più capiente è più lenta, mentre una più veloce è meno capiente.



**CPU** La memoria interna alla CPU è costituita dai registri ed è caratterizzata da alta velocità e limitate dimensioni.

**Memoria centrale** La memoria centrale è caratterizzata da dimensioni molto maggiori della memoria interna alla CPU, ma con tempi di accesso più elevati. Accessibile direttamente tramite indirizzi.

Nei sistemi attuali è presente un livello intermedio in termini di capienza e velocità, tra CPU e memoria centrale, esso è **la memoria cache**.

**Memorie secondarie** Sono ad alta capacità, con bassi costi e non volatili.

## 10.1 Principio di località

Un programma in un certo istante di tempo, accede soltanto a una porzione relativamente piccola del suo spazio di indirizzamento e questa è la base del comportamento dei programmi in un calcolatore.

### Definizione

Durante l'esecuzione di una data istruzione presente in memoria, con molta probabilità le successive istruzioni saranno ubicate nelle vicinanze di quella in corso. Nell'arco di esecuzione di un programma si tende a fare riferimenti continui alle stesse istruzioni.  
*Wikipedia*

Esistono due tipi di località:

- Temporale: quando si fa riferimento a un elemento c'è tendenza a fare riferimento allo stesso elemento dopo poco tempo
- Spaziale: quando si fa riferimento a un elemento c'è la tendenza a fare riferimenti poco dopo ad altri elementi che hanno l'indirizzo vicino ad esso

**Importante** I programmi NON vedono la gerarchia ma referenziano i dati come Se fossero sempre in memoria centrale. Il principio di località viene sfruttato strutturando la memoria in modo gerarchico.

- Velocità - Più è veloce, più è vicina al processore, più è costosa
- Dimensione - Più è grande, più è lontana dal processore, meno è costosa

Un livello di memoria più vicino al processore contiene un sottoinsieme di dati memorizzati in ogni livello sottostante e tutti i dati si trovano nel livello più basso.

## 10.2 Definizioni

- **Blocco/linea** - La più piccola quantità di informazione che può essere presente/assente in una gerarchia di memoria
- **Hit (successo nell'accesso)** - L'informazione richiesta dal processore si trova in uno dei blocchi nel livello superiore della memoria

- **Miss (fallimento nell'accesso)** - Il dato non è presente nel livello immediatamente superiore di memoria e occorre accedere al livello più distante
- **Hit rate (frequenza di hit)** - Frazione degli accessi alla memoria nei quali l'informazione richiesta è stata trovata nel livello superiore di memoria
- **Miss rate (frequenza di miss)** - Frazione degli accessi alla memoria nei quali l'informazione richiesta NON è stata trovata nel livello superiore di memoria ( $1 - \text{HitRate}$ )
- **Tempo di hit** - Tempo di accesso al livello superiore della memoria  
Comprende anche il tempo necessario a stabilire se il tempo di accesso si risolva in un successo in un fallimento
- **Tempo di miss** - Il tempo necessario a sostituire un blocco del livello superiore con un nuovo blocco dal livello inferiore della gerarchia, e trasferire i dati di questo blocco al processore
- **Frequenza di hit** =  $\frac{\text{Accessi risolti con successo}}{\text{Numero totale di accessi}}$
- **Frequenza di miss** =  $\frac{\text{Accessi risolti senza successo}}{\text{Numero totale di accessi}}$

**Remind** Una gerarchia di memoria può essere composta da più livelli, ma i dati vengono trasferiti solo tra due livelli vicini.

## 10.3 Cache

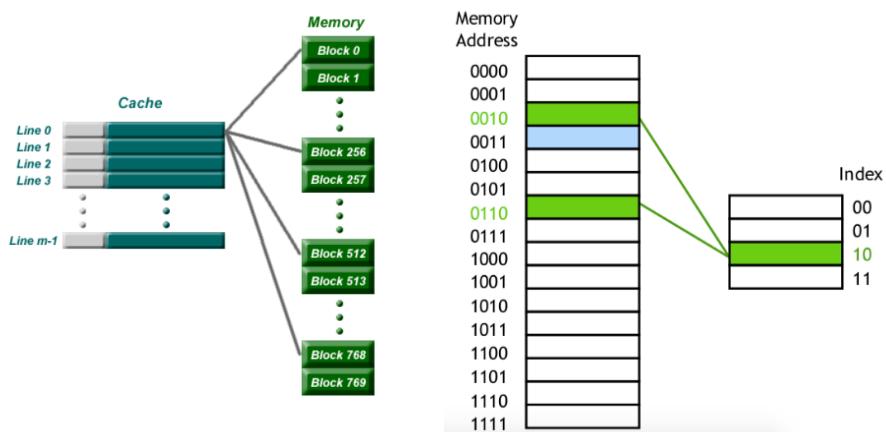
La Cache è il livello della memoria gerarchia che si trova tra il processore e la memoria principale. Proviene dal termine francese cachè, che significa nascosto. La memoria cache e il suo utilizzo sono generalmente trasparenti al programmatore (quindi nascosta). L'algoritmo per la gestione della cache, l'algoritmo di caching, si basa sui principi di località spaziale e temporale.

- Mantiene i dati richiesti recentemente "vicino" alla CPU (località temporale)
- Muove blocchi contigui di memoria che contendono il dato richiesto (località spaziale)

### 10.3.1 Tipi di cache

#### Direct Mapped

A ciascun blocco della memoria corrisponde una specifica locazione nella cache, viene direttamente assegnato uno specifico blocco. Associa una sola locazione della cache a ogni parola della memoria definendo una corrispondenza tra l'indirizzo in memoria della parola e la locazione nella cache.

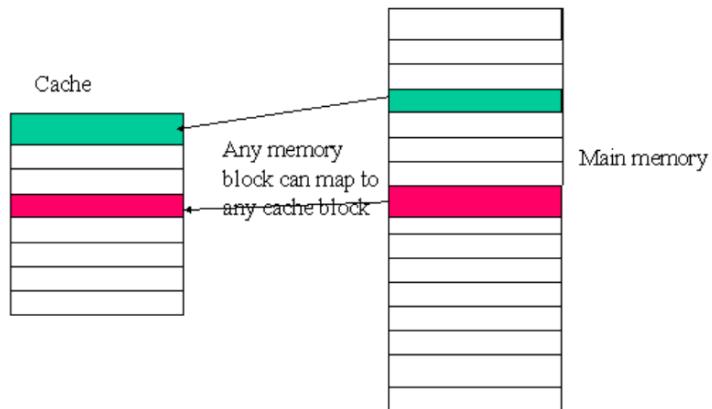


Nella prima figura viene illustrato il funzionamento delle direct mapped cache, a ciascun blocco di cache possono appartenere diversi blocchi di memoria. Nella seconda figura notiamo che se un programma utilizza 2 indirizzi diversi si verifica che lo stesso programma ha associati 2 indirizzi diversi, problema di associazione molti a uno.

#### Fully associative

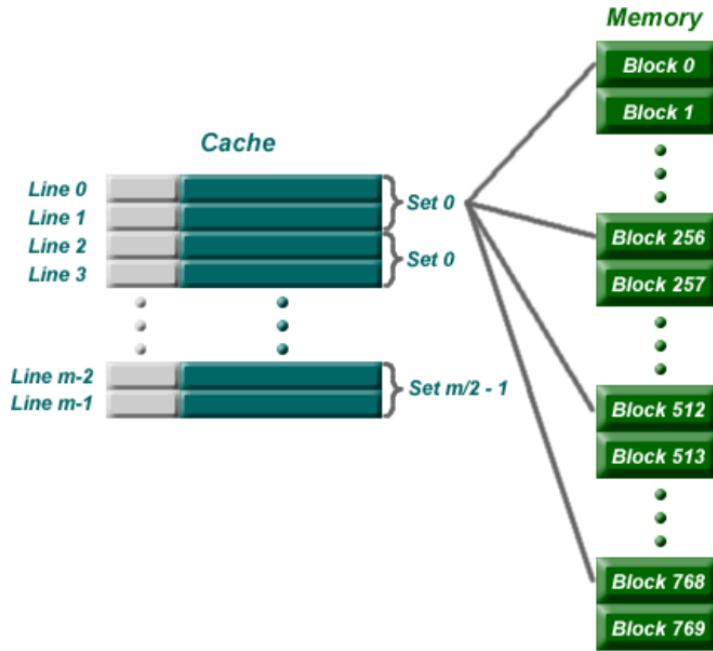
Una cache completamente associativa ci permette di memorizzare i dati in qualsiasi blocco inutilizzato della cache, anzichè forzare l'assegnazione di uno specifico blocco di memoria come nella direct mapped. Si risolve così la problematica del conflitto. In questo caso la corrispondenza è 1 a 1.

Il problema è che è molto costosa perché l'indirizzo viene unicamente riservato a un elemento, quindi la cache occupa più spazio rispetto al direct mapped. Inoltre per cercare un blocco nella cache è necessario cercarlo in tutte le linee della cache. La ricerca sequenziale è troppo lenta e la ricerca in parallelo è molto costosa.



## Set Associative Mapping

Ibrido tra Direct Mapped e Fully Associative. Un indirizzo di memoria viene mappato su un gruppo della cache detto Set. Si ragiona quindi su sottoinsiemi, non direttamente su indirizzo.



Ciascun blocco della memoria ha a disposizione un numero fisso ( $\geq 2$ ) di locazioni cache. La dimensione dell'insieme influenza la velocità di ricerca.

### 10.3.2 Campi della cache

Campi che mi permettono di verificare la presenza di un dato all'interno della cache.

- Tag (etichetta) - Contiene informazioni necessarie a verificare se una parola della cache corrisponde o meno alla parola cercata. I bit più significativi rappresentano questo campo
- Indice - Utilizzato per selezionare il blocco della cache, corrisponde all'indirizzo di memoria della cache. Rappresentato dai bit meno significativi.
- Bit di validità - Campo che indica se il blocco di memoria associato contiene un dato valido (true/false). Serve per verificare se la cache sta restituendo valori validi (per esempio in fase di accensione della CPU la cache è alimentata, ma non ha valori validi).

Se Tag (i Bit più significativi), Bit di validità e indirizzo di memoria (Bit meno significativi) coincidono genero un hit, quindi sono nel blocco di cache che mi serve.

**Esempio** Consideriamo la seguente situazione

- Indirizzo su 32 bit
- Cache a mappatura diretta
- Dimensione cache di  $2^n$  blocchi, dove n bit vengono usati per l'indice
- Dimensione del blocco della cache è di  $2^m$  parole, ossia  $2^{m+2}$  byte, per cui m bit vengono usati per individuare una parola all'interno di un blocco, mentre 2 bit per individuare una parola all'interno di un blocco, mentre 2 bit per individuare un byte all'interno della parola

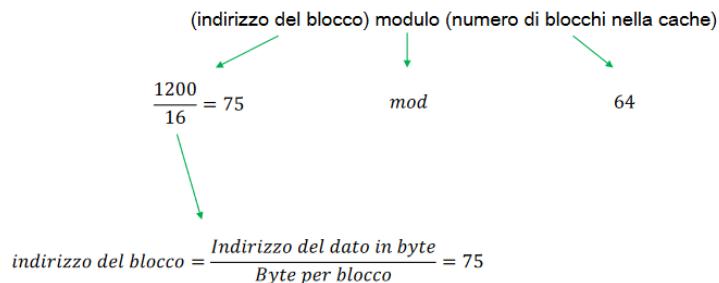
La dimensione del campo tag è:  $32 - (n + m + 2)$ .

Il numero totale di bit contenuti in una cache a mappatura diretta è:

$$2^n(\text{dimensione blocco} + \text{dimensione tag} + \text{bit validità})$$

$$2^n * (2^m * 32 + (32 - (n + m + 2)) + 1)$$

**Mappatura di un indirizzo** Consideriamo una cache con 64 blocchi di 16 byte ciascuno. L'indirizzo 1200 (in byte) può essere ricavato come:



## Scelta della dimensione del blocco di cache

La dimensione del blocco di cache è in stretta relazione con la frequenza di miss. Minore è il numero di blocchi, maggiore è la probabilità di generare un miss e viceversa. Quindi blocchi più grandi generano meno miss.

Un'ampia dimensione per il blocco permette di sfruttare la località spaziale, ottimo per esempio per le istruzioni di un programma che spesso sono fisicamente vicine fra loro.

## Svantaggi di un blocco grande

- Blocchi di grossa dimensione comportano maggiori miss penalty, è necessario quindi più tempo per trasferire il blocco
- Se la dimensione del blocco è troppo grossa rispetto alla dimensione della cache, il miss rate aumenta, dato che il numero di blocchi nella cache è insufficiente

**Miss Penalty** Differenza tra il tempo di accesso al livello inferiore e il tempo di accesso alla cache.

## Cache set-associative

I blocchi appartenenti alla cache sono raggruppati in set. In una cache set associative a N vie (**N-way set associative**) ogni set raggruppa N blocchi. Ogni indirizzo di memoria corrisponde ad un unico set della cache (accesso diretto tramite indice) e può essere ospitato in un blocco qualunque appartenente a quel set. Stabilito il set, per determinare se un certo indirizzo è presente in un blocco del set è necessario confrontare in parallelo i tag di tutti

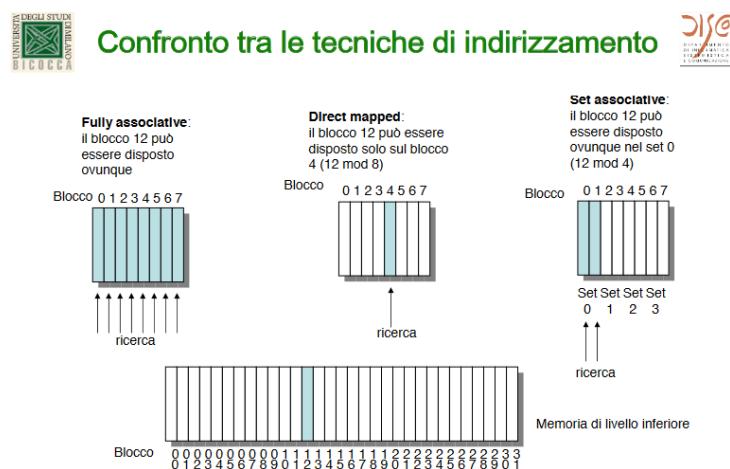
i blocchi. Ogni indirizzo di memoria corrisponde ad un unico set della cache (accesso diretto tramite indice) e può essere ospitato in un blocco qualunque appartenente a quel set. Stabilito il set, per determinare se un certo indirizzo è presente in un blocco del set è necessario confrontare in parallelo i tag di tutti i blocchi.

**Svantaggi** Cache N-way set associative a confronto con cache ad accesso diretto:

- N comparatori invece di 1
- ulteriore ritardo fornito dal multiplexer
- Il blocco è disponibile dopo la decisione Hit/Miss e la selezione del set

In una cache ad accesso diretto, il blocco è disponibile prima della decisione Hit/Miss.

### 10.3.3 Confronto tra le tecniche di indirizzamento



#### Osservazioni aumento grado di associatività

- Vantaggio principale - Diminuzione del miss rate
- Svantaggi - Maggior costo implementativo e incremento dell'hit time

La scelta tra il tipo di cache dipende dal costo dell'associatività rispetto alla riduzione del miss rate.

## 10.4 Gestione della cache

La gestione della cache consiste in queste 4 decisioni da prendere

1. Dove posizionare un blocco? - **Tecnica di indirizzamento**
2. Come reperire un blocco? - **Tecnica di indirizzamento**
3. Quale blocco sostituire in corrispondenza di un miss? - **Algoritmo di sostituzione**
4. Come gestire un'operazione di scrittura? - **Strategia di aggiornamento**

### 10.4.1 Algoritmi per la sostituzione di blocchi

#### Direct associative

Nella cache ad acceso diretto se il blocco di memoria è mappato in una linea di cache già occupata (conflict miss), si elimina il contenuto precedente della linea e si rimpiazza con il nuovo blocco.

#### Fully-associative e Set-associative

Nel caso delle fully-associative ogni blocco è un potenziale candidato per la sostituzione. Nel caso della set-associativa a N vie bisogna scegliere tra gli N blocchi del set

#### Politiche di sostituzione

**Random** (Scelta casuale) - Semplice da implementare

#### Least Recently Used (LRU)

- Sfruttando la località temporale il blocco sostituito è quello che non si utilizza da più tempo
- Ad ogni blocco si associa un contatore all'indietro, che viene portato al valore massimo in caso di accesso e decrementato di 1 ogni volta che si accede ad un altro blocco

**First In First Out (FIFO)** Si approssima la strategia LRU selezionando il blocco più vecchio anziché quello non usato da più tempo.

### 10.4.2 Gestione dei miss in lettura e Accesso in scrittura

Se un blocco non è presente nella cache bisogna mettere in stallo l'intera CPU. In generale al verificarsi di un miss nella cache delle istruzioni sono necessari i seguenti passi:

1. Inviare PC - 4 (uscita della ALU) alla memoria
2. Lettura della memoria
3. Scrittura nella cache (dato, tag e bit di validità)
4. Riavviare l'esecuzione dell'istruzione che ha causato il miss

Scrivere un dato nella cache significa creare un'incoerenza se non si aggiornano i livelli inferiori della gerarchia di memorie. Tale aggiornamento richiede lo stallo della CPU.

#### Tecniche scrittura Cache

- Write-through
- Write-back
- Utilizzo di un Write Buffer

#### Write-Through

Scrittura simultanea nel blocco della cache e nella memoria inferiore.

#### Vantaggi

- Soluzione più semplice da implementare
- Si mantiene la coerenza delle informazioni nella gerarchia di memorie

#### Svantaggi

- Le operazioni di scrittura vengono effettuate alla velocità della memoria di livello inferiore, quindi diminuiscono le prestazioni
- Aumento il traffico sul bus di sistema

## Write-back

(o copy-back): i dati sono scritti solo nel blocco della cache. Il blocco modificato viene scritto nel livello inferiore della gerarchia solo quando deve essere sostituito. Subito dopo la scrittura cache e memoria non sono consistenti, dato che per un breve lasso di tempo non sono coerenti. Il blocco cache può essere quindi in due stati: stato clean (non modificato), dirty (modificato). Il bit che denota questo stato è detto dirty bit.

### Vantaggi

- Le scritture avvengono alla velocità della cache
- Scritture successive sullo stesso blocco alterano solo la cache

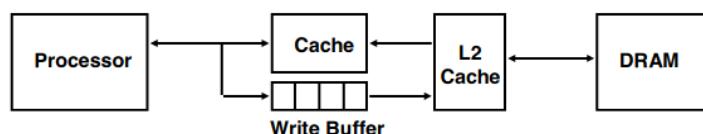
### Svantaggi

- Ogni sostituzione del blocco può provocare un trasferimento in memoria

## Write buffer

Viene interposto tra la cache e la memoria di livello inferiore un buffer di scrittura. Il processore scrive i dati sia nella cache che nel write buffer. Il controller della memoria scrive il contenuto del write buffer in memoria.

**Gestito in modalità FIFO** Il numero tipi di elementi del buffer è 4. Efficiente se la frequenza di scrittura <<  $\frac{1}{\text{write cycle della DRAM}}$ . Altrimenti il buffer può andare in saturazione e il processore deve aspettare che le scritture giungano a completamento (write stall).



Per ovviare al problema si inserisce una cache di secondo livello.

### 10.4.3 Write miss

Anche le scritture possono generare un write miss, si verifica quando si tenta di scrivere in un blocco che non è presente nella cache.

### Possibili soluzioni

- **Write allocate** - Il blocco viene caricato in cache e si effettua la scrittura
- **No-write allocate** - Il blocco viene scritto direttamente nella memoria di livello inferiore, senza essere trasferito in cache

Le cache Write back tendono ad utilizzare Write allocate.

Le cache Write-Through prediligono No-write allocate.

### 10.4.4 Riassumendo

- Una cache con blocchi di dimensioni maggiori sfrutta maggiormente la località spaziale diminuendo la frequenza di miss.
- La frequenza di miss torna a crescere se la dimensione dei blocchi diventa troppo grande rispetto alla dimensione della cache.
- Quindi i blocchi vengono scaricati dalla cache prima ancora che molti dati in essi contenuti siano stati utilizzati
- Quindi la località spaziale tra le parole di un blocco diminuisce e il miglioramento legato alla frequenza di miss si riduce
- Cresce anche il costo di una miss: la penalità di una miss è determinata dal tempo necessario a prelevare un blocco dal livello sottostante e a scriverlo nella cache
- Il tempo per prelevare un blocco è dato dalla somma tra la latenza per ottenere la prima parola del blocco e il tempo di trasferimento del resto del blocco

### Cose fondamentali

- Gerarchia memoria
- Definizioni miss
- Tecniche di indirizzamento
- Grafico hit miss
- Esercizi su Indirizzamento