

Architettura degli Elaboratori

UniShare

Davide Cozzi
@dlcgold

Indice

1	Introduzione	3
2	Rappresentazione dell'informazione	4
2.0.1	Tipi di Dati	10
2.0.2	Varie Rappresentazioni	11
2.1	Esercizi	12
3	Il computer	16
3.1	MIPS32	17
4	Reti Sequenziali e Combinatorie	24
4.1	ALU	43
4.1.1	ALU a 1bit	43
4.1.2	ALU a 32bit	45
4.2	Esercizi	48
5	Assembly	54
6	Datapath	72
6.0.1	Unità di elaborazione	75
6.0.2	Performance singolo ciclo	89
6.0.3	Implementazione Multiciclo	91
6.0.4	CPI	103
7	Eccezioni	105
8	Gestione I/O	109
9	Cache	126
9.1	Cache	127
10	Tip & Tricks	130

11 Approfondimento	132
11.1 IEE754	132
11.1.1 Operazioni in virgola mobile	134
11.1.2 Virgola Mobile in MIPS	140
11.2 Pipeline	141
11.2.1 Pipelined Datapath	147

Capitolo 1

Introduzione

Questi *NON* sono appunti presi a lezione ma sono bensì un riassunto tratto da libro e slide del corso (la parte riguardante I/O deve essere ancora ultimata, attualmente sono presenti quasi unicamente suddette slide). Si assicura la probabile presenza di errori. Per eventuali proposte di correzione effettuare una pull request. Link: <https://github.com/dlcgold/Appunti>.

Grazie mille e buono studio!

Capitolo 2

Rappresentazione dell'informazione

Innanzitutto si ha che quando si hanno n bit si possono avere 2^n configurazioni diverse di bit. Quando dichiaro una variabile sto dichiarando una zona di memoria che contiene i bit di quel valore. Il numero di bit è finito.

Nota 1. Per indicare una base si usa la notazione $|_{base}$

Esempio 1. $n = 8$ comporta 256 codici, quindi non permette la rappresentazioni di numeri negativi o maggiori di 255. Si avranno quindi le seguenti rappresentazioni numeriche in 8 bit:

- $00000000 = 0|_{10}$
- $00000001 = 1|_{10}$
- $11111111 = 255|_{10}$

Esempio 2. Proviamo ora la conversione di numeri binari:

da decimale a binario:

convertendo $140|_{10}$ in binario:

$$\frac{140}{2} = 70 \text{ con resto } 0$$

$$\frac{70}{2} = 35 \text{ con resto } 0$$

$$\frac{35}{2} = 17 \text{ con resto } 1$$

$$\frac{17}{2} = 8 \text{ con resto } 1$$

$$\frac{8}{2} = 4 \text{ con resto } 0$$

$$\frac{4}{2} = 2 \text{ con resto } 0$$

$$\frac{2}{2} = 1 \text{ con resto } 0$$

$$\frac{1}{2} = 0 \text{ con resto } 1$$

quindi si ha: $10001100|_2$ **da binario a decimale:**
converti $10011|_2$ in decimali:

$$1 \times 2^0 = 1 +$$

$$1 \times 2^1 = 2 +$$

$$0 \times 2^2 = 0 +$$

$$0 \times 2^3 = 0 +$$

$$1 \times 2^4 = 16 =$$

$$19|_{10}$$

Il binario si trova nei contenitori di bit e lì non ha rappresentazione da destra a sinistra bensì ha una rappresentazione data dall'hardware (dalle saldature). Se prendo $n = 8$ so che i valori che si possono rappresentare sono nel range $[0, 255]$ e negli intervalli prima e dopo non ho modo di rappresentare nulla con 8 bit, si ha quindi un'*eccezione*, in questo caso un'*overflow*. Inoltre si possono avere problemi nel rappresentare numeri che sono risultato di operazioni aritmetiche (soprattutto la divisione). Si possono inoltre avere problemi causati dalla precedenza degli operatori

Modulo e segno

Per poter rappresentare i numeri negativi, utilizzando sempre 8 bit, si utilizza 1 bit per rappresentare il segno, cambiando così il range dei valori rappresentabili, che diventa:

$$[-(2^{n-1} - 1), (2^{n-1} - 1)]$$

Si ha però un problema: lo zero viene rappresentato da due codici:

- $0 = \overbrace{0}^{\text{segno}} 0000000$

- $0 = \overbrace{1}^{segno} 0000000$

Per ovviare a questo problema sono nate 2 soluzioni:

1. Complemento a 2:

Questa codifica consente nel prendere il binario di un valore, complementarne i valori (il cosiddetto *complemento a 1*) ovvero dove avevo 1 scriverò 0, e viceversa, e infine sommo 1 al risultato, ottenendo così l'opposto del valore di partenza. Si ottiene così un nuovo range per i valori:

$$[-2^{n-1}, 2^{n-1} - 1] = [-128, 127]$$

Inoltre una tecnica di questo tipo, oltre a togliere il problema del doppio zero, permette all'hardware che sa far le somme di saper fare anche le differenze (senza aver quindi necessità di cambiare hardware)

Esempio 3. prendo 01101010, ne faccio il complemento a 1 e ottengo 10010101. Ora sommo 1:

$$\begin{array}{r} 10010101 + \\ 00000001 = \\ \hline 10010110 \end{array}$$

2. Eccesso:

Si hanno i numeri da $256 = 11111111$ a $0 = 00000000$, le traslo verso il basso di 128, ottenendo così: $127 = 11111111$ e $-128 = 00000000$, eliminando quindi il problema del doppio zero.

Entrambe le tecniche portano a definire 0 come numero positivo.

Rappresentazione Esadecimale

I binari sono spesso espressi in esadecimali, ovvero in base 16. Si ha quindi la notazione $|_{16}$. In questa notazione si usano 16 simboli grafici: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$. Si ha per esempio

altra notazione per gli esadecimali

$$11|_{10} = A|_{16} \text{ e } 16|_{10} = 10|_{16} = \overbrace{0x16}$$

Esempio 4. Proviamo ora la conversione di numeri esadecimali:

da esadecimale a decimale:

convertendo $2A1_{16}$ in decimali:

$$1 \times 16^0 = 1 +$$

$$A \times 16^1 = 10 \times 16 = 160 +$$

$$2 \times 16^2 = 2 \times 256 = 512 =$$

$$673|_{10}$$

da decimale a esadecimale:

convertendo $6574|_{10}$ in esadecimali:

$$\frac{6574}{16} = 410 \text{ con resto } 14$$

$$\frac{410}{16} = 25 \text{ con resto } 10$$

$$\frac{25}{16} = 1 \text{ con resto } 9$$

$$\frac{1}{16} = 0 \text{ con resto } 1$$

e mi fermo perché ho quoziente 1

abbiamo quindi:

$$1\ 9\ 10\ 14 = 19AE|_{16}$$

Errori e Precisione

La precisione dei numeri interi è 1. Ci sono numeri, come quelli appartenenti a \mathbb{Q} e \mathbb{R} , che non sono rappresentabili dagli interi. Si cerca quindi un'approssimazione di quella cifra scegliendo il codice intero più vicino, e l'errore sarà la distanza tra il codice e il valore vero. Per la rappresentazione dei numeri non interi si hanno due modi:

1. Numeri in Virgola Fissa:

Si ha quando conosco i codici vicini al valore che cerco e posso quindi suddividere l'intervallo tra i due codici in parti uguali. In termini pratici assegno m bit prestabiliti alla parte decimale e i restanti alla parte intera. Per esempio dividendo l'intervallo per 1000 avrò la precisione di un millesimo, con un errore di approssimazione costante. Quindi

dati n bit ne uso uno per il segno, m per la parte decimale e il restante per la parte intera. La parte decimale è decisa a priori e si rappresenta come un numero naturale ma con esponenti negativi della base.

Esempio 5. trasformo $101.01|_2$ in decimali:

$$101 \rightarrow 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$$

$$01 \rightarrow 0 \times 2^{-1} + 1 \times 2^{-2} = 0.25$$

quindi $101.01|_2 \rightarrow 5.25|_{10}$

Esempio 6. trasformo $5.125|_{10}$ in binario:

$$\frac{5}{2} = 2 \text{ con resto } 1$$

$$\frac{2}{2} = 1 \text{ con resto } 0$$

$$\frac{1}{2} = 0 \text{ con resto } 1$$

quindi 101.

Passo alla parte decimale:

$$0.125 \times 2 = 0.25 \text{ ovvero } 0 \text{ con resto } 0.25$$

$$0.25 \times 2 = 0.5 \text{ ovvero } 0 \text{ con resto } 0.5$$

$$0.5 \times 2 = 1 \text{ ovvero } 1 \text{ con resto } 0$$

Quindi $5.125|_{10} \rightarrow 101.001|_2$

Si possono avere rappresentazioni periodiche. Si possono rappresentare pochi numeri e spesso in maniera non completa, con uno spreco di memoria per rappresentare gli zeri.

2. Numeri in Virgola Mobile:

Si ha quando non si conosce l'intervallo tra due codici, e quindi l'intervallo non può essere suddiviso in parti uguali. In termini di alto livello stiamo parlando di *Float* e *Double*. Con la notazione in virgola mobile si ha che la spaziatura dei vari intervalli cresce proporzionalmente. Si ha quindi un errore assoluto (modulo della differenza tra valore reale e valore trovato) crescente e non costante mentre l'errore relativo (rapporto tra errore assoluto e media delle misurazioni) è costante.

Un numero si dice *normalizzato* se, in virgola mobile, non possiede zeri iniziali (esempio $1,1 \times 10^{-9}$).

Per la rappresentazione si cerca un compromesso tra la **mantissa** (il valore compreso tra 0 e 1 che viene posto nel campo dopo la virgola) e l'**esponente**, aumentando la prima si aumenta l'accuratezza mentre aumentando il secondo si aumenta il range di numeri rappresentabili. Nel MIPS si ha la seguente rappresentazione:

1 bit segno + 8 bit esponente + 23 bit mantissa

ovvero:

$$(-1)^S \times F \times 2^E$$

con F contenuto del campo mantissa e E contenuto del campo esponente. Nel MIPS si possono rappresentare numeri frazionari piccoli quanto $2|_{10} \times 10^{-38}$ e grandi $2|_{10} \times 10^{38}$, ma se l'esponente è troppo grande si finisce in *overflow* come si può finire in *underflow* se la parte frazionaria è troppo piccola. Questa appena spiegata è la rappresentazione in *singola precisione*. Si può avere la rappresentazione in *doppia precisione* che in MIPS richiede 2 parole, una con un bit per il segno, 11 per l'esponente e 20 per la mantissa mentre l'altra dedica 32 bit alla mantissa. Si ottiene così una mantissa a 52 bit con la possibilità di rappresentare numeri frazionari piccoli quanto $2|_{10} \times 10^{-308}$ e grandi $2|_{10} \times 10^{308}$. Inoltre lo standard IEE 754 prevede una *polarizzazione* pari a 127 per la singola precisione in modo da ottenere l'esponente più negativo con 00...00 e quello più positivo 11...11. Si ha quindi che -1 è rappresentato da $-1 + 127|_{10} = 126|_{10} = 01111110|_2$ e $+1$ con $1 + 127|_{10} = 128|_{10} = 10000000|_2$. Per la doppia precisione si usa un esponente di 1023. Con la polarizzazione si ha:

$$(-1)^S \times (1 + \text{mantissa}) \times 2^{\text{esponente} - \text{polarizzazione}}$$

così, in singola precisione, si va da $\pm 1,000\dots|_2 \times 2^{-126}$ a $\pm 1,111\dots|_2 \times 2^{+127}$

Esempio 7. Si mostra la rappresentazione binaria IEEE 754 di $-0,75|_{10}$:
 $-0,75$ può essere scritto come:

$$-\frac{3}{4}|_{10} = -\frac{3}{2^2}|_{10}$$

quindi sapendo che $3|_{10} = 11_2$ ho:

$$-\frac{11|_2}{2^2|_{10}} = -0,11|_2$$

in notazione scientifica è $-0,11|_2 \times 2^0$ e in notazione scientifica normalizzata $-1,1|_2 \times 2^{-1}$.

Si ha quindi:

$$(-1)^1 \times (1 + 0,1000\dots|_2) \times 2^{126-127}$$

con tanti 0 nella mantissa quanti i bit, nella doppia precisione si avrà la stessa cosa con più zeri e con $2^{1022-1023}$

Esempio 8. Trovo il decimale dal seguente numero in virgola mobile: 1100000010100....0.

Il bit di segno è uguale a 1, il campo esponente è $10000001 = 129|_{10}$ e il campo mantissa contiene $1 \times 2^{-2} = 0.25|_{10}$. Ora uso l'equazione:

$$(-1)^1 \times (1 + 0,25) \times 2^{129-127} = -1 \times 1.25 \times 4 = -5,0$$

2.0.1 Tipi di Dati

Tutte le moderne CPU usano lo standard **IEEE754** che permette alla mantissa di essere di 24 bit in quanto si rende implicito il primo bit (1):

- Single (float) → 32 bit
- Double (double) → 54 bit
- Extended → 80 bit

Dati (di ogni tipo) e ed istruzioni sono solo configurazioni di bit Un Single (32 bit), per esempio, è così scomposto (da sinistra verso destra):

- **msb** (*most significant bit*) → 1 bit (0 per il segno positivo, 1 per il segno negativo)
- **exp** (*exponent*) → 8 bit
- **frac** (*fraction*) → 22 bit
- **lsb** (*least significant bit*) → 1 bit (ultima cifra della parte frazionaria, mi dice se il numero è pari o dispari)

32 bit			
msb	exp	frac	lsb
1 bit	8 bit	22 bit	1 bit
segno	esponenziale	parte decimale	

Esempio 9 (???). Posso effettuare la trasformazione:

$$001101011100 \times 2^0 \rightarrow 001, \overbrace{1010111100}^{significand} \times 2^{1001}$$

A seguito la tabella per la codifica **EEE754**

Single Precision	Double Precision	Object Represented		
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1-254	Anything	1-2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	\pm NaN (<i>Not a Number</i>)

Forma Denormalizzata

Posso sfruttare degli spostamenti per rappresentare ulteriori numeri limitando la precisione, aumentando l'errore relativo. Nell'esponente dei denormalizzati c'è una configurazione di zeri.

2.0.2 Varie Rappresentazioni

Rappresentazione di Caratteri

Si ha:

- **ASCII**, a 7 bit, quindi $2^7 = 128$ caratteri
- **UNICODE**, a 16 bit, quindi $2^{16} = 65536$ caratteri

Rappresentazioni di Audio e Immagini

- Il suono viene convertito in tensione/grandezze elettriche analogiche e convertito con il campionamento binario
- L'immagine viene convertita come per il suono ma converte i vari colori, in una determinata posizione, in dati binari

2.1 Esercizi

Esercizio 1. converto $1101101|_2$ in esadecimale:
prima converto in decimali:

$$1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 64 + 32 + 8 + 4 + 1 = 109|_{10}$$

ora converto $109|_{10}$ in esadecimali:

$$\frac{109}{16} = 6 \text{ con resto di } 13$$

$$\frac{6}{16} = 0 \text{ con resto di } 6$$

quindi $1101101|_2 = 6D|_{16}$

Esercizio 2. converto $52|_{16}$ in binario:
innanzitutto lo converto in decimale:

$$2 \times 16^0 = 2 +$$

$$5 \times 16^1 = 80 =$$

$$82|_{10}$$

ora converto $82|_{10}$ in binario:

$$\frac{82}{2} = 41 \text{ con resto } 0$$

$$\frac{41}{2} = 20 \text{ con resto } 1$$

$$\frac{20}{2} = 10 \text{ con resto } 0$$

$$\frac{10}{2} = 5 \text{ con resto } 0$$

$$\frac{5}{2} = 2 \text{ con resto } 1$$

$$\frac{2}{2} = 1 \text{ con resto } 0$$

$$\frac{1}{2} = 0 \text{ con resto } 1$$

quindi $52|_{16} = 1010010|_2$

Esercizio 3. converto $-20|_{10}$ in binario a 8 bit con il complemento a 2:

$$\frac{20}{2} = 10 \text{ con resto } 0$$

$$\frac{10}{2} = 5 \text{ con resto } 0$$

$$\frac{5}{2} = 2 \text{ con resto } 1$$

$$\frac{2}{2} = 1 \text{ con resto } 0$$

$$\frac{1}{2} = 0 \text{ con resto } 1$$

quindi $20|_{10} = 10100|_2$, che in 8 bit è 00010100. Ora procedo col complemento a 1 e ottengo 01011, in 8 bit 11101011. Infine sommo 1:

$$11101011 +$$

$$00000001 =$$

$$\hline 11101100$$

Se procediamo col metodo modulo e segno si ha che si parte da 10100, si converte in 7 bit 0010100 e si aggiunge il MSB (essendo negativo si mette 1) ottenendo 10010100

Esercizio 4 (???). siano 01001 e 10001 prima una rappresentazione in modulo e segno e poi in complemento a 2. Definire nei due casi quale è maggiore. Parto dal modulo e segno (si vede che il primo è maggiore anche dal fatto che è positivo mentre l'altro è negativo):

01001 sarebbe 1001 quindi $+1001|_2 = +9|_{10}$ 10001 sarebbe 0001 quindi $-0001|_2 = -1|_{10}$

quindi 01001 è maggiore.

Passiamo al complemento a 2:

01001 sarebbe $01001 - 1 = 01000$ a cui applico il complemento a 1 ottenendo $10111|_2$. Lo converto in decimale:

$$2^0 + 2^1 + 2^2 + 2^4 = 1 + 2 + 4 + 16 = 23|_{10}$$

quindi stiamo ragionando del numero $-23|_{10}$

10001 sarebbe $10001 - 1 = 10000$ cui applico il complemento a 1 ottenendo

$01111|_2$. Lo converto in decimale:

$$2^0 + 2^1 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15|_{10}$$

quindi stiamo ragionando del numero $-15|_{10}$.
quindi 10001 è maggiore.

Esercizio 5. Sommo 0x789 con 0x987:

$9 + 7 = 16$ quindi 0 col riporto di 1 $8 + 8 + 1 = 17$ quindi 1 col riporto di 1
 $7 + 9 + 1 = 17$ quindi 1 col riporto di 1 $1 + 0 = 1$ quindi:

$$\begin{array}{r} 789 + \\ 987 = \\ \hline 1110 \end{array}$$

Esercizio 6. Rappresento -20 per eccesso 128, ovvero, dato che voglio $128 = n^{k-1}$ avrò $k=8$ bit:

$$-20 + 128 = 108|_{10} = 01101100|_2$$

Esercizio 7. sia $00011100|_2$, con eccesso 128, trovo il decimale corrispondente:

$00011100 = 28|_{10}$ quindi so che $x + 128 = 28$ quindi $x = -100$, che è il decimale corrispondente a $00011100|_2$

Esercizio 8. Innanzitutto si riporta la tabella ascii:

ASCII Code Chart																
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	!	"	#	\$	%	&	'	()	*	+	,	-	.	/	
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-
6	.	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Con-

verto la parola Bit in decimale e esadecimale:

esadecimale = 426974

decimale = 0661051,116

Esercizio 9. Scrivo 13,25 in virgola mobile:

trasformo 13 in binario: 1101.

Passo alla parte decimale:

$$0,25 \times 2 = 0,5 \text{ ovvero } 0 \text{ col resto di } 0,5$$

$$0,5 \times 2 = 1 \text{ ovvero } 1 \text{ col resto di } 0$$

quindi avrò 1101.01 ovvero $1101.01 \times 2^0 = 1.10101 \times 2^3$. Si avrà:

$$(-1)^0 \times (1 + 0,1010100...0) \times 2^{130-127}$$

e quindi la rappresentazione in virgola mobile su 32 bit sarà:

segno positivo = 0

esponente = 130 = 10000010

mantissa= 101010000000000000000000

quindi:

$$13,25 = 0\ 10000010\ 101010000000000000000000$$

Esercizio 10. Ricavo il decimale della seguente notazione in virgola mobile:

1 10000000 11000000000000000000000000000000:

segno negativo = 1

esponente = $10000000|_2 = 128|_{10}$ quindi avrà $2^{128-127} = 2$

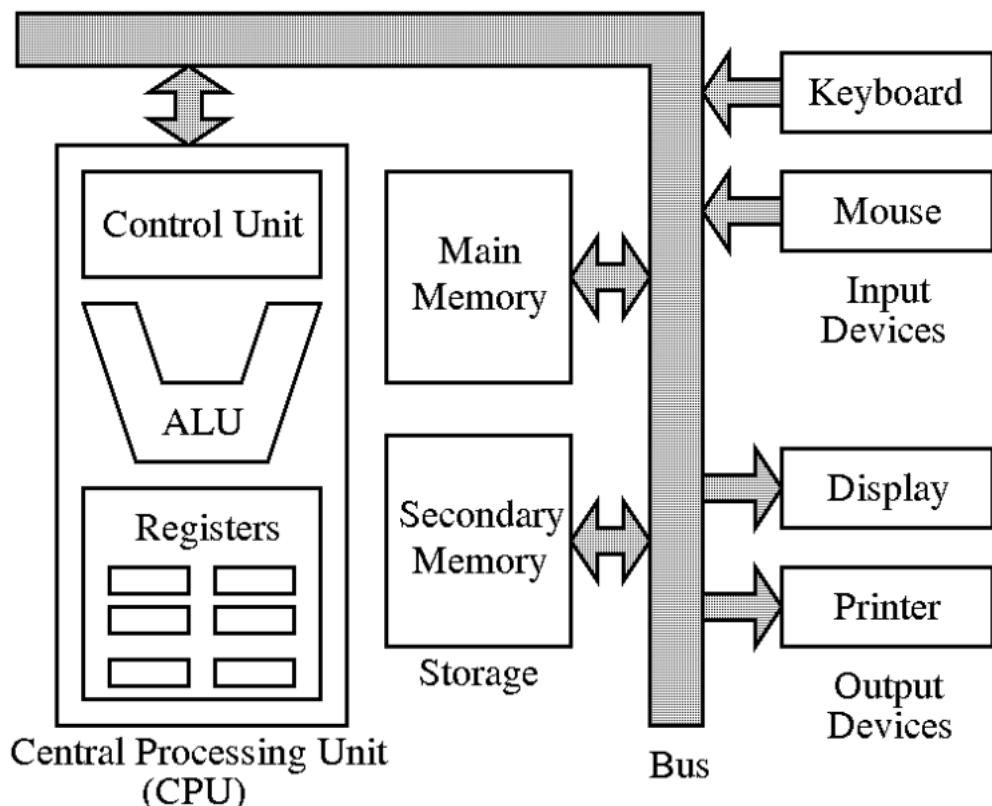
parte frazionaria mantissa = $11000000000000000000000000000000 = 0,75|_{10}$ quindi avrà:

$$(-1)^{-1} \times (1 + 0,75) \times 2 = -3,5$$

Quindi $1\ 10000000\ 11000000000000000000000000000000 = -3,5|_{10}$

Capitolo 3

Il computer



Si introducono le componenti base di un calcolatore che permettono di realizzare le funzioni di input, output, elaborazione e memorizzazione dati:

- **Unità di Controllo:** è la componente del processore che invia i comandi all'unità di elaborazione dati, alla memoria e ai dispositivi di I/O secondo le istruzioni del programma

- **Register File:** l'elemento che contiene i registri ($R0...Rn$) che possono essere letti o scritti fornendo il numero del registro
- **Memoria:** è dove vengono messi i programmi quando sono in esecuzione insieme ai dati da loro richiesti
- **DRAM:** i chip di DRAM costituiscono la memoria, se ne possono quindi usare più di una per memorizzare le istruzioni di un programma. DRAM sta per *memoria ad accesso casuale*. Si differenzia dalla normale memoria ad accesso sequenziale per il fatto che sono memorie il cui accesso richiede lo stesso tempo indipendentemente dalla particolare area di memoria
- **Memoria Cache:** una memoria piccola ma veloce che agisce come "tampone", *buffer*, nei confronti della DRAM. Essa è formata da SRAM, *memoria statica ad accesso casuale*
- **Memoria di Massa:** detta anche *memoria secondaria* è una memoria non volatile dove si conservano i dati tra un'esecuzione e l'altra (HDD e SSD)
- **Datapath:** detta anche *unità di elaborazione dati*, *ALU* è la componente che esegue le operazioni logico-aritmetiche. Recupera i dati dai registri del processore, li processa nell'accumulatore e salva i risultati nel registro d'uscita
- **Program Counter** è un registro del processore che conserva l'indirizzo di memoria della istruzione successiva. Si usa in un ciclo *fetch-execute* che sarebbe una ripetizione infinita del caricamento di un'istruzione dal Program Counter, l'aggiornamento di quest'ultimo con l'istruzione successiva e l'esecuzione della stessa

3.1 MIPS32

Nell'architettura MIPS abbiamo tre tipi di istruzioni: *R-Type*, *I-type* e *J-Type*. Andiamo ad analizzare il loro linguaggio macchina.

R-Type

Soltamente rappresenta quelle operazioni che hanno due dati da elaborare e un risultato, per esempio *add*, *sub*, *mul*, *div*, *and*, *or*, Soltamente sono così formate:

$$6\text{bit}(op) + 5\text{bit}(rs) + 5\text{bit}(rt) + 5\text{bit}(rd) + 5\text{bit}(shamt) + 6\text{bit}(funct)$$

Nel dettaglio si ha:

- **op:** *operation code* identifica l'operazione, 000000 è quello dell'R-Type
- **registri:** *rs* è il primo registro sorgente, *rt* è il secondo registro sorgente e *rd* è il registro destinazione
- **Shamt:** *shift amount* numero di posizioni di scorrimento, indica lo shift
- **funct:** specifica la variante dell'operazione base

le istruzioni possono essere scritte in esadecimali. Per convertire in binario basta scrivere ogni singolo elemento del'esadecimale in binario (se per esempio c'è *e* avrò 1110) e viceversa. Si ha la seguente tabella:

BINARIO	DECIMALE	ESADECIMALE
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Si aggiungono le tabelle coi registri e le operazioni base:

Register name	Number	Usage
\$zero	0	constant 0
\$at	1	reserved for assembler
\$v0	2	expression evaluation and results of a function
\$v1	3	expression evaluation and results of a function
\$a0	4	argument 1
\$a1	5	argument 2
\$a2	6	argument 3
\$a3	7	argument 4
\$t0	8	temporary (not preserved across call)
\$t1	9	temporary (not preserved across call)
\$t2	10	temporary (not preserved across call)
\$t3	11	temporary (not preserved across call)
\$t4	12	temporary (not preserved across call)
\$t5	13	temporary (not preserved across call)
\$t6	14	temporary (not preserved across call)
\$t7	15	temporary (not preserved across call)
\$s0	16	saved temporary (preserved across call)
\$s1	17	saved temporary (preserved across call)
\$s2	18	saved temporary (preserved across call)
\$s3	19	saved temporary (preserved across call)
\$s4	20	saved temporary (preserved across call)
\$s5	21	saved temporary (preserved across call)
\$s6	22	saved temporary (preserved across call)
\$s7	23	saved temporary (preserved across call)
\$t8	24	temporary (not preserved across call)
\$t9	25	temporary (not preserved across call)
\$k0	26	reserved for OS kernel
\$k1	27	reserved for OS kernel
\$gp	28	pointer to global area
\$sp	29	stack pointer
\$fp	30	frame pointer
\$ra	31	return address (used by function call)

MIPS Instruction Types

Type	-31-	format (bits)					-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)	
I	opcode (6)	rs (5)	rt (5)	immediate (16)			
J	opcode (6)	address (26)					

R-Type Instructions (Opcode 000000)

Main processor instructions that do not require a target address, immediate value, or branch displacement use an R-type coding format. This format has fields for specifying of up to three registers and a shift amount.

For instructions that do not use all of these fields, the unused fields are coded with all 0 bits. All R-type instructions use a 000000 opcode. The operation is specified by the function field.

Instruction	Function	
add	rd, rs, rt	100000
addu	rd, rs, rt	100001
and	rd, rs, rt	100100
break		001101
div	rs, rt	011010
divu	rs, rt	011011
jalr	rd, rs	001001
jr	rs	001000
mfhi	rd	010000
mflo	rd	010010
mthi	rs	010001
mtlo	rs	010011
mult	rs, rt	011000
multu	rs, rt	011001
nor	rd, rs, rt	100111
or	rd, rs, rt	100101
sll	rd, rt, sa	000000
sllv	rd, rt, rs	000100
slt	rd, rs, rt	101010
sltu	rd, rs, rt	101011
sra	rd, rt, sa	000011
srav	rd, rt, rs	000111
srl	rd, rt, sa	000010
srlv	rd, rt, rs	000110
sub	rd, rs, rt	100010
subu	rd, rs, rt	100011
syscall		001100
xor	rd, rs, rt	100110

I-Type Instructions (All opcodes except 000000, 00001x, and 0100xx)

I-type instructions have a 16-bit immediate field that codes an immediate operand, a branch target offset, or a displacement for a memory operand. For a branch target offset, the immediate field contains the signed difference between the address of the following instruction and the target label, with the two low order bits dropped. The dropped bits are always 0 since instructions are word-aligned.

For the bgez, bgtz, blez, and bltz instructions, the rt field is used as an extension of the opcode field.

opcode (6)	rs (5)	rt (5)	immediate (16)
Instruction	Opcode	Notes	
addi	rt, rs, immediate	001000	
addiu	rt, rs, immediate	001001	
andi	rt, rs, immediate	001100	
beq	rs, rt, label	000100	
bgez	rs, label	000001	rt = 00001
bgtz	rs, label	000111	rt = 00000
blez	rs, label	000110	rt = 00000
bltz	rs, label	000001	rt = 00000
bne	rs, rt, label	000101	
lb	rt, immediate(rs)	100000	
lbu	rt, immediate(rs)	100100	
lh	rt, immediate(rs)	100001	
ihu	rt, immediate(rs)	100101	
lui	rt, immediate	001111	
lw	rt, immediate(rs)	100011	
lwc1	rt, immediate(rs)	110001	
ori	rt, rs, immediate	001101	
sb	rt, immediate(rs)	101000	
slti	rt, rs, immediate	001010	
sltiu	rt, rs, immediate	001011	
sh	rt, immediate(rs)	101001	
sw	rt, immediate(rs)	101011	
swc1	rt, immediate(rs)	111001	
xori	rt, rs, immediate	001110	

J-Type Instructions (Opcode 00001x)

The only J-type instructions are the jump instructions `j` and `jal`. These instructions require a 26-bit coded address field to specify the target of the jump. The coded address is formed from the bits at positions 27 to 2 in the binary representation of the address. The bits at positions 1 and 0 are always 0 since instructions are word-aligned.

When a J-type instruction is executed, a full 32-bit jump target address is formed by concatenating the high order four bits of the PC (the address of the instruction following the jump), the 26 bits of the target field, and two 0 bits.

opcode (6)	target (26)
---------------	----------------

Instruction	Opcode	Target
j label	000010	coded address of label
jal label	000011	coded address of label

Coprocessor Instructions (Opcode 0100xx)

The only instructions that are described here are the floating point instructions that are common to all processors in the MIPS family. All coprocessor instructions instructions use opcode 0100xx. The last two bits specify the coprocessor number. Thus all floating point instructions use opcode 010001.

The instruction is broken up into fields of the same sizes as in the R-type instruction format. However, the fields are used in different ways.

Most floating point instructions use the format field to specify a numerical coding format: single precision (.s), double precision (.d), or fixed point (.w). The `mfc1` and `mtc1` instructions use the format field as an extension of the function field.

opcode (6)	format (5)	ft (5)	fs (5)	fd (5)	function (6)
---------------	---------------	-----------	-----------	-----------	-----------------

Instruction	Function	Format
add.s	fd, fs, ft	000000 10000
cvt.s.w	fd, fs, ft	100000 10100
cvt.w.s	fd, fs, ft	100100 10000
div.s	fd, fs, ft	000011 10000
mfc1	ft, fs	000000 00000
mov.s	fd, fs	000110 10000
mtc1	ft, fs	000000 00100
mul.s	fd, fs, ft	000010 10000
sub.s	fd, fs, ft	000001 10000

OPCODE map

Table of opcodes for all instructions:

	000	001	010	011	100	101	110	111
000	R-type		j	jal	beq	bne	blez	bgtz
001	addi	addiu	slti	sltiu	andi	ori	xori	
010								
011	llo	lhi	trap					
100	lb	lh		lw	lbu	lhu		
101	sb	sh		sw				
110								
111								

FUNC map of R-type instructions

Table of function codes for register-format instructions:

	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srav	
001	jr	jalr						
010	mfhi	mthi	mflo	mtlo				
011	mult	multu	div	divu				
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

Some R-type instructions grouped by their familiarity:

Instruction	Function
add	rd, rs, rt
addu	rd, rs, rt
sub	rd, rs, rt
subu	rd, rs, rt
and	rd, rs, rt
or	rd, rs, rt
xor	rd, rs, rt
nor	rd, rs, rt
slt	rd, rs, rt
sltu	rd, rs, rt
sll	rd, rt, sa
srl	rd, rt, sa
sllv	rd, rt, rs
srav	rd, rt, rs
sra	rd, rt, sa

per esempio:

000000 01000 01001 01010 00000 100000

che in esadecimali è $0x01095020$ in quanto possiamo notare come: 0000 0001 0000 1001 0101 0000 0010 0000 sia facilmente convertibile con la tabella sopra. Si riconoscono:

- $op = 000000$ quindi si ha un R-type
- $rs = 01000$ che è $8|_{10}$ e quindi il registro $\$t0$ è il primo registro sorgente
- $rt = 01001$ che è $9|_{10}$ e quindi il registro $\$t1$ è il secondo registro sorgente
- $rd = 01010$ che è $10|_{10}$ e quindi il registro $\$t2$ è il registro destinazione
- $shamt = 00000$
- $funct = 100000$ che è $32|_{10}$ e indica l'operazione *add*

quindi avremo:

`add $t0, $t1, $t2`

I-type

Si usa per eseguire operazioni con un dato contenuto in un registro e un numero costante (detto *valore immediato*). Si hanno operazioni come *addi*, *lw*, *sw*, Solitamente sono così formate:

$$6bit(op) + 5bit(rs) + 5bit(rt) + 16bit(imm)$$

Nel dettaglio si ha:

- **op:** *operation code* identifica l'operazione, come faceva *funct* per l'R-Type. Può quindi usare tutti gli opcode tranne 000000, 000010 e 000011
- **registri:** *rs* è il registro sorgente e *rt* è il registro destinazione.
- **imm:** identifica un valore immediato o un offset (immediato)

Per esempio:

001000 01000 01010 0000000000000000100

ovvero:

- $op = 001000$ che identifica *addi*
- $rs = 01000$ che è $8|_{10}$ e quindi il registro $\$t0$ è il registro sorgente
- $rt = 01001$ che è $9|_{10}$ e quindi il registro $\$t1$ è il registro destinazione
- $imm = 00000000000000100$ identifica il valore 4

quindi si avrà:

`addi $t0, $t1, 4`

J-Type

Si usa per identificare i salti ed è molto semplice:

$$6bit(op) + 26bit(address)$$

ovvero:

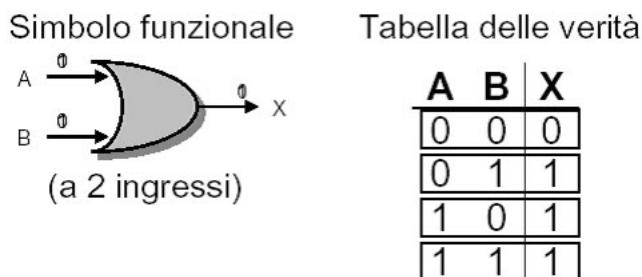
- **op:** *operation code* 000010 identifica *j*, il salto *incondizionato* e 000011, il *jal* che identifica il salto condizionato
- **address:** *rs* è il registro a cui saltare.

Capitolo 4

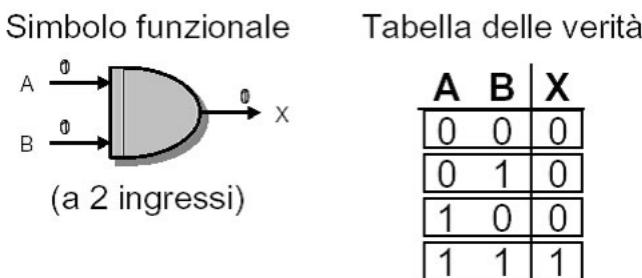
Reti Sequenziali e Combinatorie

Le parti principali di un computer sono costruite da circuiti digitali che elaborano segnali logici identificati da 0 e 1. Alla base dei circuiti si hanno le porte logiche:

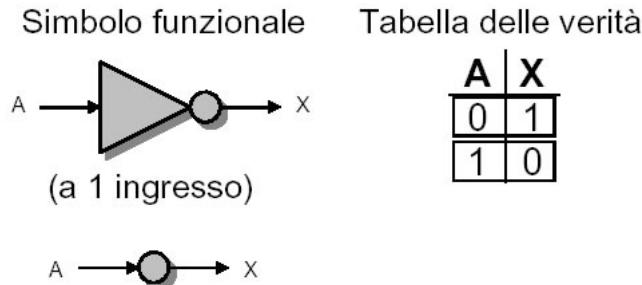
- **Somma Logica OR:** se almeno una variabile di ingresso è 1 la variabile di uscita sarà 1



- **Prodotto Logico AND:** la variabile di uscita è 1 sse entrambe quelle di ingresso sono 1:



- **Negazione NOT:** l'uscita sarà la variabile opposta a quella d'ingresso:



Essa può essere aggiunta alle porte OR e AND ottenendo NOR e NAND che avranno le uscite opposte a OR e AND. Per il NOR si ha che se almeno una variabile di ingresso è 1 la variabile di uscita sarà 0 e per il NAND che la variabile di uscita è 0 sse entrambe quelle di ingresso sono 1 (usando solo la porta NAND si possono ottenere le porte AND, OR e NOT, si chiama *universalità di NAND*). Si ha inoltre la porta XOR (*OR esclusivo*) che ha come uscita 1 sse le variabili d'ingresso sono diverse:

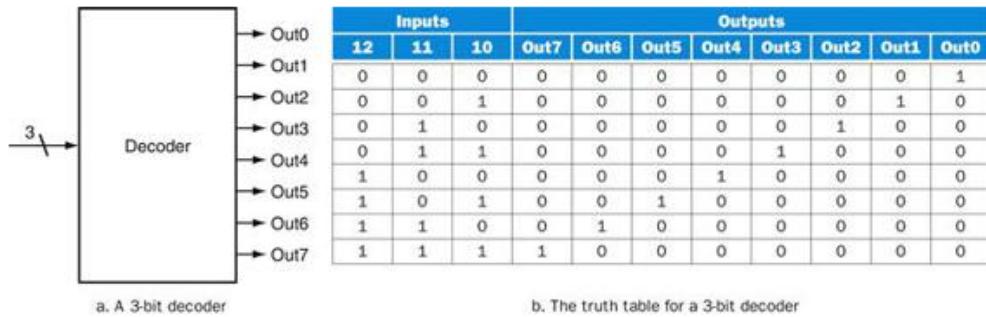
<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th style="text-align: center; width: 33px;">A</th> <th style="text-align: center; width: 33px;">B</th> <th style="text-align: center; width: 33px;">$f(A,B) = A \oplus B$</th> </tr> </thead> <tbody> <tr><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td></tr> <tr><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td></tr> <tr><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td></tr> <tr><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td></tr> </tbody> </table> EX-OR 	A	B	$f(A,B) = A \oplus B$	0	0	0	0	1	1	1	0	1	1	1	0	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th style="text-align: center; width: 33px;">A</th> <th style="text-align: center; width: 33px;">B</th> <th style="text-align: center; width: 33px;">$f(A,B) = \overline{A \cdot B}$</th> </tr> </thead> <tbody> <tr><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td></tr> <tr><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">1</td></tr> <tr><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td></tr> <tr><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td></tr> </tbody> </table> NAND 	A	B	$f(A,B) = \overline{A \cdot B}$	0	0	1	0	1	1	1	0	1	1	1	0	<table border="1" style="border-collapse: collapse; width: 100%;"> <thead> <tr> <th style="text-align: center; width: 33px;">A</th> <th style="text-align: center; width: 33px;">B</th> <th style="text-align: center; width: 33px;">$f(A,B) = \overline{A + B}$</th> </tr> </thead> <tbody> <tr><td style="text-align: center;">0</td><td style="text-align: center;">0</td><td style="text-align: center;">1</td></tr> <tr><td style="text-align: center;">0</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td></tr> <tr><td style="text-align: center;">1</td><td style="text-align: center;">0</td><td style="text-align: center;">0</td></tr> <tr><td style="text-align: center;">1</td><td style="text-align: center;">1</td><td style="text-align: center;">0</td></tr> </tbody> </table> NOR 	A	B	$f(A,B) = \overline{A + B}$	0	0	1	0	1	0	1	0	0	1	1	0
A	B	$f(A,B) = A \oplus B$																																													
0	0	0																																													
0	1	1																																													
1	0	1																																													
1	1	0																																													
A	B	$f(A,B) = \overline{A \cdot B}$																																													
0	0	1																																													
0	1	1																																													
1	0	1																																													
1	1	0																																													
A	B	$f(A,B) = \overline{A + B}$																																													
0	0	1																																													
0	1	0																																													
1	0	0																																													
1	1	0																																													

Collegando queste porte posso ottenere tutti i circuiti digitali. Si possono ottenere:

- **Circuiti Combinatori** dove le uscite dipendono unicamente dalle entrate
- **Circuiti Sequenziali** dove le uscite dipendono anche dallo stato interno del circuito

Vediamo alcuni circuiti logici famosi:

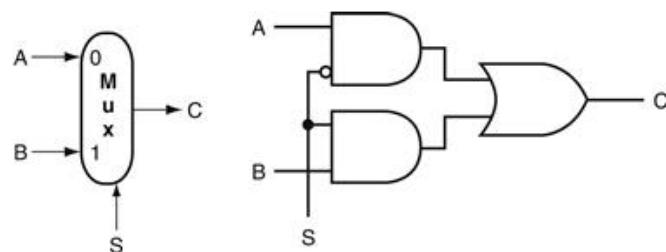
- **Decoder:** il più comune decoder prende in ingresso n bit e ha 2^n uscite dove però viene presa una sola uscita data dalla combinazione delle entrate. Il decoder traduce il segnale d'ingresso su n bit in un segnale che corrisponde al binario dell'ingresso. Gli output sono nominati $out_0, \dots, out_{2^n-1}$. Se il valore in ingresso è x si avrà true sull'uscita out_x e false su tutte le altre. Esiste anche l'**Encoder**, che prende in ingresso 2^n bit e ne ha n in uscita. Ecco un esempio con 3 bit in ingresso, e quindi $2^3 = 8$ bit in uscita:



Si nota che se per esempio di ha in ingresso $101|_2 = 5|_{10}$ si avrà 1, ovvero true, solamente su out_5

- **Multiplexor o Multiplexer:** si ha un'uscita uguale ad uno degli ingressi, scelto mediante un segnale di controllo. Si può usare un numero arbitrario di dati in ingresso, se ci sono due ingressi si ha un solo segnale di controllo, se se ne hanno n si hanno $\log_2(n)$ segnali di controllo. Ecco un esempio di un Multiplexor a due ingressi (A e B), con il segnale di controllo S , che rappresenta la seguente operazione, di uscita C :

$$C = (A \cdot S) + (B \cdot \bar{S})$$



se fosse stato formato da n input si avrebbe avuto la seguente composizione:

1. un decoder che genera n segnali, ciascuno corrispondente ad un diverso valore dell'ingresso di selezione
 2. una serie di n porte AND ciascuna che combina uno degli input con un segnale dal decoder
 3. una singola porta OR che raccoglie le uscite delle porte AND
- **Two-Level-logic e PLA (Programmable Logic Arrays):** si è visto che ogni funzione logica si può rappresentare con le porte OR, AND e NOT . Si può avere la TWO-Level-Rappresentazione usando appunto unicamente le porte AND e OR, con al più un NOT finale. Per l'equazione

$$E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot \overline{(A \cdot B \cdot C)}$$

si ha:

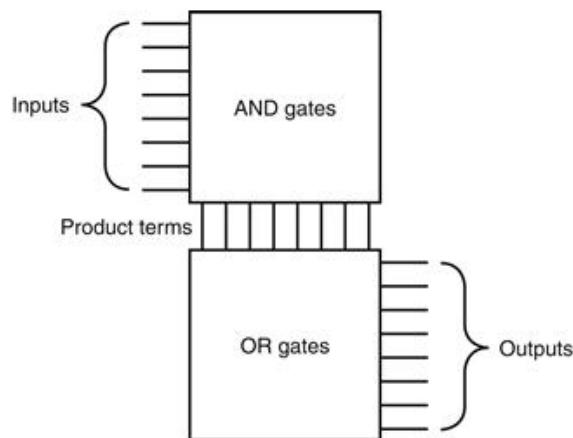
1. **Somma di Prodotti:** è la rappresentazione della somma logica (OR) mediante i prodotti logici (AND):

$$E = (A \cdot B \cdot \overline{C}) + (A \cdot C \cdot \overline{B}) + (B \cdot C \cdot \overline{A})$$

2. **Prodotto di Somme:** è l'esatto opposto della Somma di prodotti:

$$E = \overline{((\overline{A} + \overline{B} + C) \cdot (\overline{A} + \overline{C} + B) \cdot (\overline{B} + \overline{C} + A))}$$

Si userà soprattutto la *Somma di Prodotti* che viene rappresentata dalla PLA. Una PLA ha un insieme di inputs e i relativi complementi (ottenuti grazie a degli invertitori) e due stadi di logica. Il primo stadio è un array di porte AND che formano un insieme di *termini prodotto* (detti *ninterms*), ciascuno dei quali può consistere in uno qualsiasi degli input o dei loro complementi. Il secondo stadio è un array di porte OR ciascuna delle quali effettua una somma logica di un certo numero di *termini prodotto*. Ecco una rappresentazione schematica:



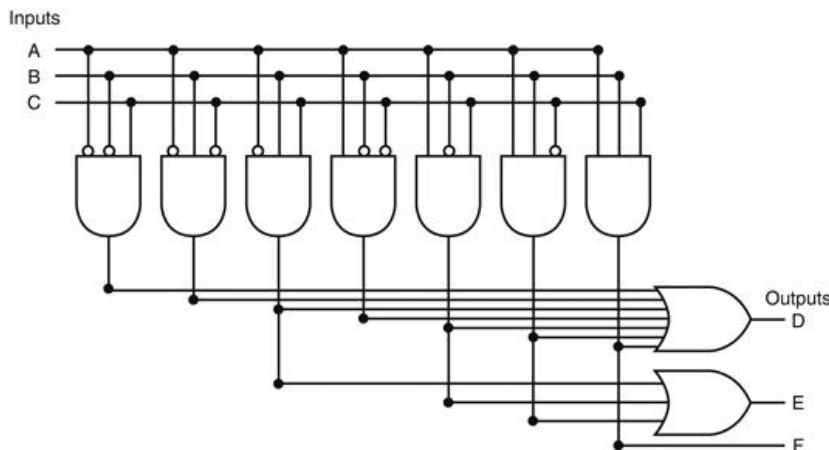
Una PLA può implementare direttamente la tavola di verità di una certa funzione logica, con molteplici input e output. Dato che ogni ingresso per il quale l'output è true richiede un *termine prodotto* ci sarà una riga corrispondente nella PLA. Ogni output corrisponde ad una certa riga di porte OR nel secondo stadio. La grandezza della PLA è data dalla somma delle lunghezze dei due array di porte AND e OR (dette *AND plane* e *OR plane*). La *AND plane* sarà pari a *numero input* \times *numero ninterms differenti* mentre la *OR plane* sarà pari a *numero output* \times *numero ninterms differenti*. Nella PLA solo la tavola di verità che produce true ha delle porte logiche associate e esiste un solo ingresso per ogni minterm differente (due *termini prodotto* uguali non hanno due ingressi diversi).

Capitolo 4. Reti Sequenziali e Combinatorie

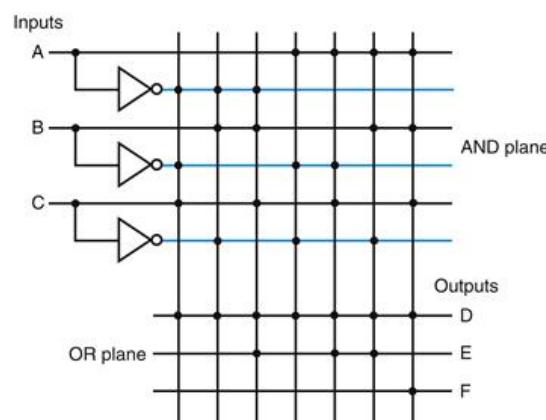
Vediamo un esempio dove si hanno A,B,C come input la cui tabella rappresenta lo stadio AND) e D,E,F come output la cui tabella rappresenta lo stadio OR). Ecco la tavola di verità:

Inputs			Outputs		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

che può essere rappresentata con le porte logiche (l'ingresso $A = 0$, $B = 0$, $C = 0$ non è presente perché non da output):



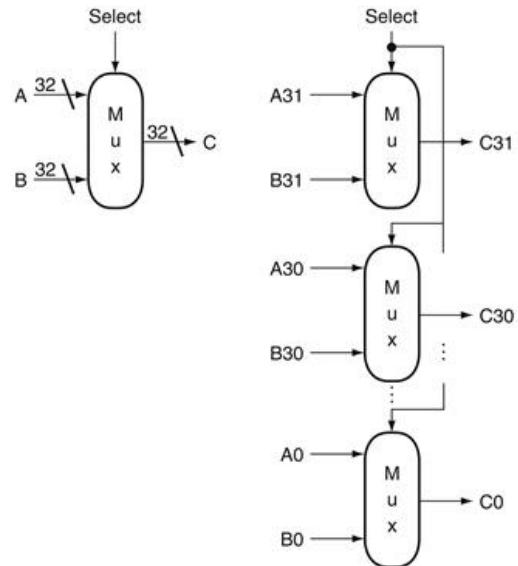
o mediante uno schema dove i pallini scuri sono gli 1 e si ha una rappresentazione mediante linee verticali e orizzontali:



- **ROM (Read Only Memory):** una ROM è detta memoria in quanto è un'insieme di locazioni che possono essere lette, tuttavia il contenuto è fisso e deciso nel momento della produzione. Esistono poi le PROM, ovvero le ROM *programmable* che possono essere programmate elettronicamente dopo il momento della produzione. Infine esistono le EPROM, ovvero le PROM *erasable*, che possono essere riscritte, cancellando con la luce ultravioletta il contenuto precedente. Sono quindi memorie di sola lettura (tranne durante la produzione e la fase di debug). Una ROM è formata da un insieme di funzioni logiche che forniscono gli indirizzi di ingresso e le uscite. Il numero di bit presenti in ciascun elemento indirizzabile determina il numero di bit d'uscita ed è detto *ampiezza* della ROM, per n linee di input si ha un'ampiezza pari a n . Se la ROM contiene 2^n elementi indirizzabili (detta *altezza* della ROM) si hanno n linee di input. Altezza e larghezza definiscono lo *shape (forma)* della ROM. Una ROM può codificare una collezione di funzioni logiche direttamente dalla tabella di verità, per esempio con m funzioni e n input ci serve una ROM con n linee di indirizzo e $2n$ ingressi, con ogni ingresso largo m bit. Gli ingressi nella tavola di verità sono gli ingressi nella ROM, mentre le uscite della tavola sono il contenuto della ROM. Se la tabella di verità è organizzata in modo tale che le entrate sono una sequenza di binari allora l'output farà sì che il contenuto della ROM avrà lo stesso ordine. A differenza della PLA la ROM è totalmente codificata e quindi ha più entrate (che crescono in maniera esponenziale) e più termini prodotto (che crescono in numero più lentamente). Le ROM possono implementare qualsiasi funzione logica abbinando input e output. Diventa così semplice cambiare il contenuto della ROM se cambia la funzione logica, dato che la grandezza della ROM non cambia

Si elencano ora altri aspetti, strutture o circuiti:

- **Bus:** è una collezione di linee di informazione unite da un singolo segnale logico. Per esempio nelle istruzioni MIPS il risultato di un'istruzione può essere scritto in un registro provenendo da una di due sorgenti. Un multiplexor sceglie quale dei due Bus (ciascuno largo 32bit) sarà scritto nel registro del risultato. Ovviamente il multiplexor a 1bit visto sopra dovrà essere replicato 32 volte. Nei disegni si rappresenta una linea diagonale sopra quella dell'input per rappresentare il numero di strutture (numero da indicare a fianco) necessarie, come in figura:

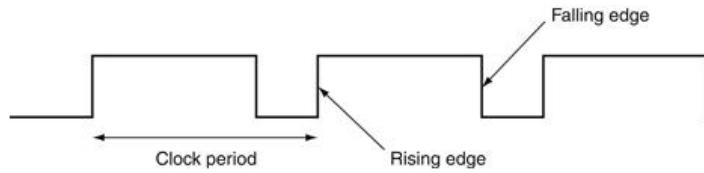


a. A 32-bit wide 2-to-1 multiplexor

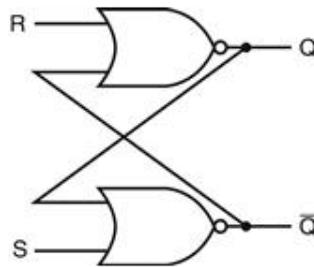
b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

- **Carry In e Carry Out:** il carry In rappresenta il riporto dell'operazione aritmetica precedente, il Carry Out rappresenta il riporto dell'operazione appena eseguita e diventa il Carry In di quell'operazione seguente.
- **Segnale di Clock:** nella logica sequenziale i segnali di clock sono necessari per determinare quando devono essere aggiornati gli elementi. Si ha quindi un segnale che evolve indipendentemente nel tempo con un certo periodo T costante. Si ha la frequenza di clock data da $\nu = \frac{1}{T}$. Il ciclo di clock si divide in 2 parti, quando il clock è alto e quando è basso, tra basso e alto si ha il *fronte (edge) di salita* e tra l'alto e il basso si ha il *fronte (edge) di discesa*. Tutti i cambi di stato avvengono in corrispondenza di un fronte.

Ecco una figura che semplifica quanto detto:



- **SR Latch:** è uno degli elementi di memoria più semplici, detto anche *Set and Reset Latch*, è costituito da una coppia di porte NOR e il bit immagazzinato è portato all'output Q e al suo complementare \bar{Q} . Se R e S non sono assegnate la coppia incrociata di coppie NOR agisce come un invertitore e salva i valori precedenti di Q e \bar{Q} . Ecco un'immagine dell'SR Latch:

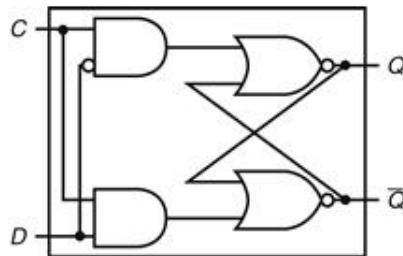


Si ha il seguente schema:

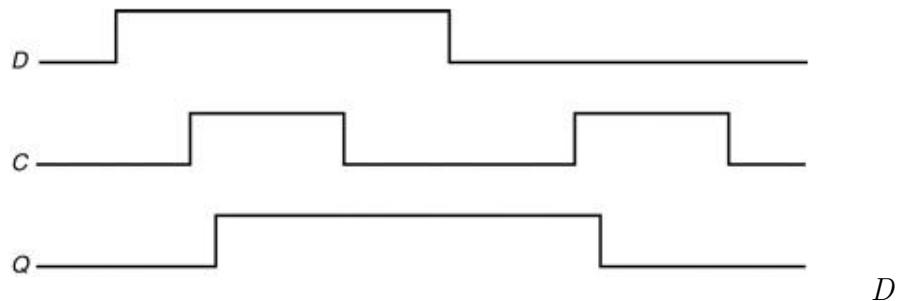
S	R	funzione
0	0	Latch (memorizzazione)
0	1	Reset: $Q = 0$ e $\bar{Q} = 1$
1	0	Set: $Q = 1$ e $\bar{Q} = 0$
1	1	Errore, operazione non consentita

- **D Latch:** ha 2 input e 2 output, dove gli input sono il valore da memorizzare, chiamato D, e il segnale di clock, chiamato C, che indica quando il latch deve leggere il valore sull'ingresso D e memorizzarlo. Gli output sono semplicemente il valore dello stato interno Q e il suo complemento \bar{Q} . Quando il valore di clock C è affermato (1) si ha che il Latch è *aperto* e il valore di Q diventa il valore di D. Quando il clock non è affermato (0) si dice che il Latch è *chiuso* e il valore di Q è rimane lo stesso dell'ultima volta che il Latch è stato *aperto*. Dato che Q varia ogni volta che varia D questa struttura è anche detta *Latch trasparente*.

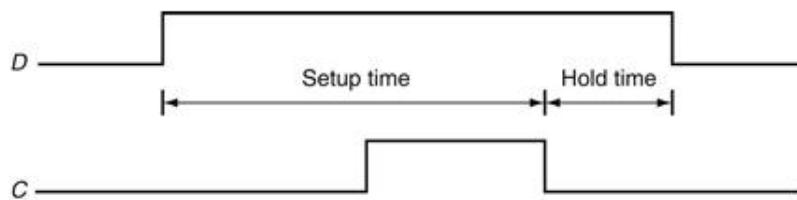
Ecco un'immagine che rappresenta il D Latch:



ed ecco cosa succede schematicamente a D e Q in corrispondenza del periodo di clock:

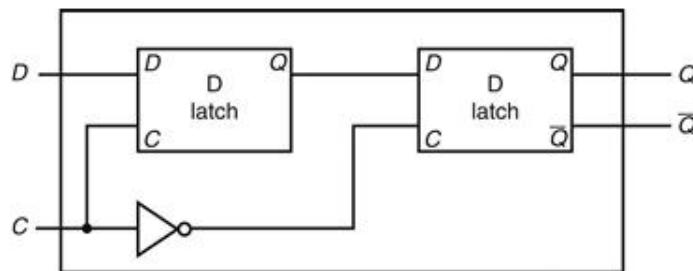


assume un valore, quando si ha il fronte di C lo "scrive" in Q, poi assume un altro valore che viene scritto non appena si ha un altro fronte di C. Il tempo minimo per cui l'ingresso D deve rimanere valido prima del fronte di clock è detto **tempo di set-up (tempo di preparazione)** mentre il tempo minimo per cui deve rimanere valido dopo il fronte è detto **tempo di hold (tempo di mantenimento)**, che solitamente è circa 0. Fallimenti di questi tempi minimi possono comportare output non prevedibili. Il valore salvato e l'output Q cambiano entrambi quando C è alto



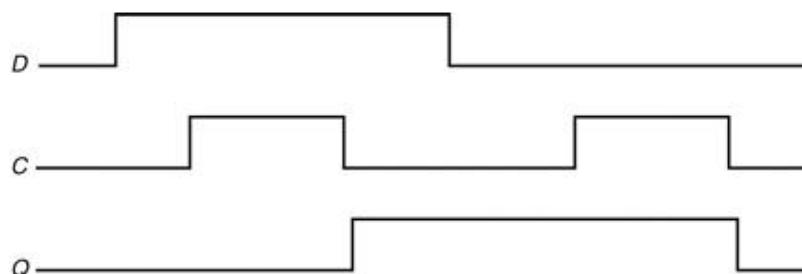
- **D Flip-Flop** si ha che un Flip-Flop è un elemento di memoria dove l'output è uguale al valore dello stato dentro l'elemento e si ha che lo stato cambia solo in un fronte di clock, può essere costruito in modo che l'azione venga svolta sia nel fronte di salita che in quello di discesa del clock (o uno o l'altro). I Flip-Flop non sono *trasparenti*, l'output cambia solo su un fronte di clock. Un D Flip-Flop è un Flip-Flop con una solo input di informazioni che salva il valore del segnale in input quando si ha un fronte di clock.

Ecco un'immagine del D Flip-Flop che agisce nel fronte di discesa costruito da una coppia di D Latch:



*Il primo Latch, chiamato **master**, è aperto quindi segue il valore di D solo se C è affermato. Quando il clock cade si chiude il primo Latch ma il secondo, chiamato **slave** è aperto e prende l'input dall'output del primo Latch*

ed ecco cosa succede schematicamente a D e Q in corrispondenza del periodo di clock:



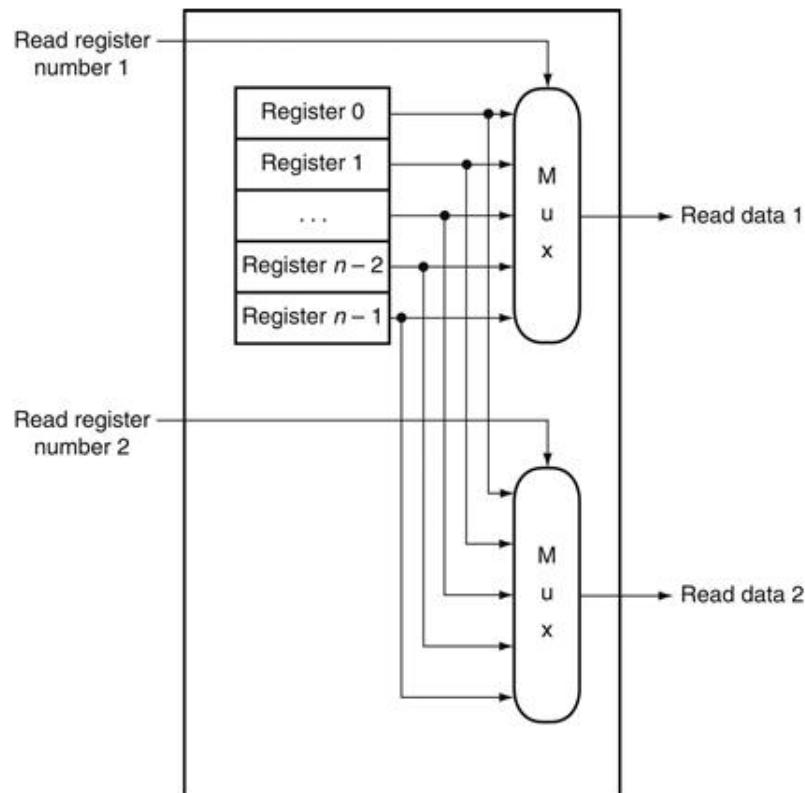
Quando C scende l'output di Q salva il valore dell'input D

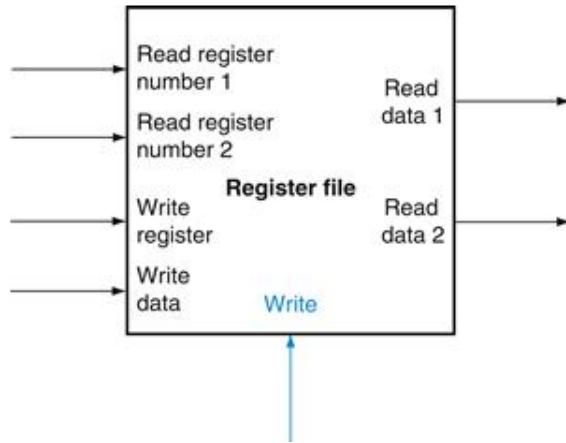
- **Register File:** la struttura principale del nostro datapath è il Register File, che consiste da un insieme di registri che possono essere letti e scritti fornendo il numero del registro da utilizzare. Si può implementare con un decoder per ogni porta di lettura o scrittura e con una matrice di registri fatti da dei D Flip-Flop. Dato che leggere un registro non cambia nessuno stato basta fornire in ingresso un indirizzo e si avrà come output l'informazione contenuta nel registro. Per scrivere un registro si avrà bisogno di 3 input:

1. il numero del registro
2. l'informazione da scrivere
3. un segnale di clock che controlli l'operazione di scrittura

Le porte di lettura invece si possono implementare con 2 multiplexor n -to-1, ciascuno di ampiezza pari al numero di bit presenti in ogni registro del Register File.

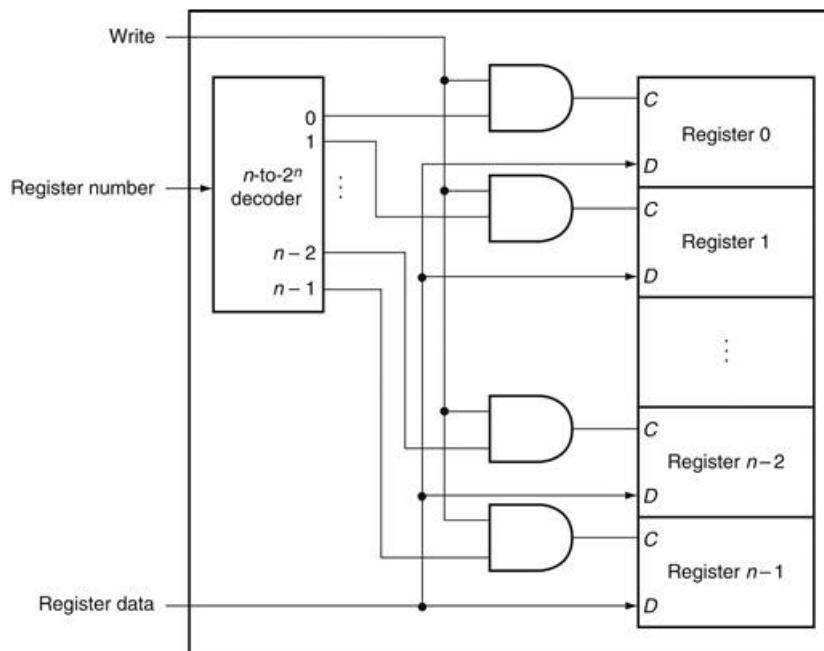
A seguito un'immagine che rappresenta l'implementazione di 2 porte di lettura di registri per un Register File ampio 32 bit:





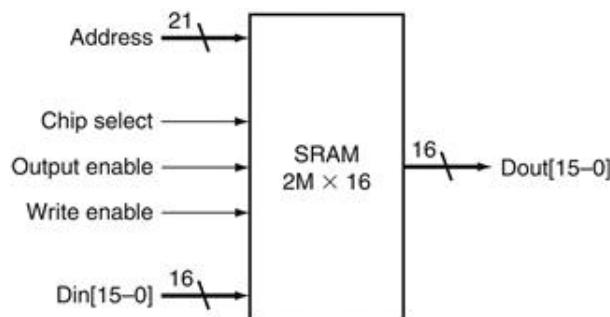
un Register File con due porte di lettura e una di scrittura ha 5 input
e 2 output

ecco invece l'implementazione di una porta di scrittura (un po' più complessa a causa del fatto che possiamo cambiare il contenuto solo del registro scelto). Per ottenerla si usa un decoder $n\text{-to-}2^n$ che genera un segnale indicante il registro da usare. Tutti e tre gli input hanno *setup time* e *hold time* per consentire la corretta esecuzione dell'operazione:



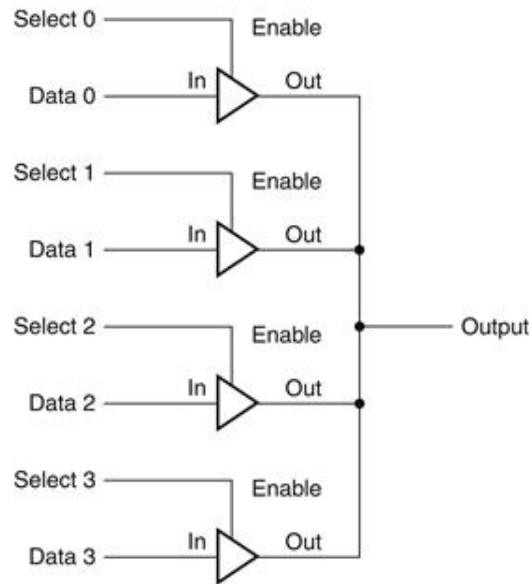
Se lo stesso registro viene letto e scritto durante un ciclo di clock si ha che, dato che la scrittura avviene su un fronte del clock, il registro sarà valido durante la scrittura e il valore letto sarà quello scritto in un ciclo di clock precedente. Per leggere il valore in corso di scrittura servirebbero delle implementazioni aggiuntive alla logica del Register File. Si hanno i seguenti:

1. **Read Reg #1:** numero del primo registro da leggere
 2. **Read Reg #2:** numero del secondo registro da leggere
 3. **Read Data 1:** valore del primo registro letto sulla base del *Read Reg #1*
 4. **Read Data 2:** valore del secondo registro letto sulla base del *Read Reg #2*
 5. **Write Reg#:** numero del registro su cui si deve scrivere
 6. **Write Data:** valore del registro da scrivere in base a *Write Reg#*
- **SRAM (Static Random Access Memory):** è una memoria dove i dati sono salvati in maniera statica (come in un Flip-Flop). Più veloce della DRAM ma più costosa. Si hanno dei semplici circuiti integrati che sono gli array della memoria con solitamente una sola porta d'accesso, che provvede sia alla lettura che alla scrittura. Si ha un tempo di accesso fisso, anche se ci sono differenze tra lettura e scrittura. Si hanno inoltre configurazioni specifiche per il numero di locazioni indirizzabili e per l'ampiezza di ciascuno di essi. Per esempio una SRAM $4M \times 8$ provvede con 4M entrate, ciascuna di 8 bit, con quindi 22 linee di accesso ($4M = 2^{22}$), un output a 8bit e un singolo dato in input da 8bit. Come nelle ROM si ha che il numero di locazioni indirizzabili è chiamato *altezza* e il numero di bit per ciascuna unità è detto *larghezza*. Vediamo lo schema di una SRAM $2M \times 16$, con i 21 indirizzi, i 16 dati in input, i 3 controlli e i 16 dati in output:

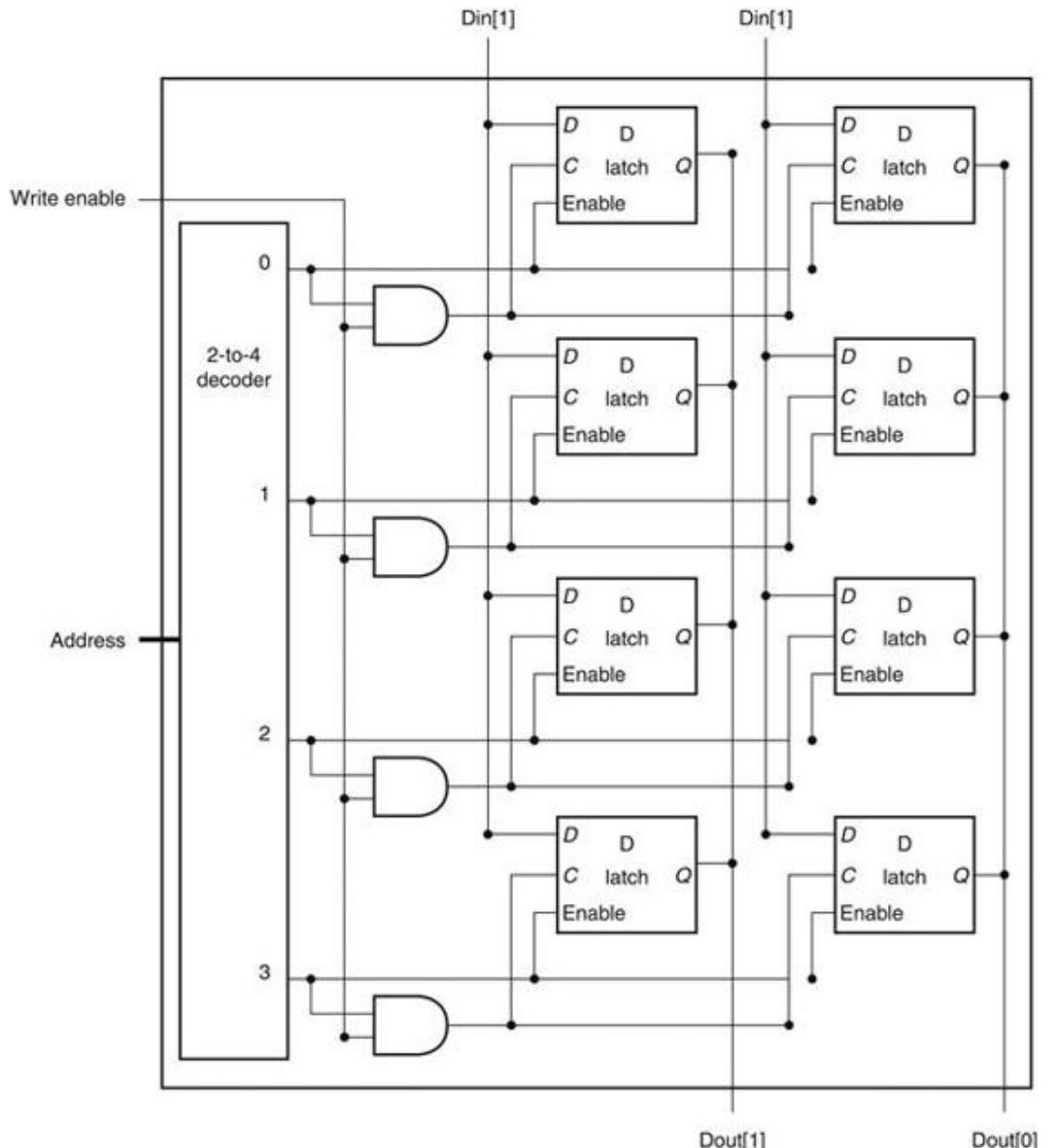


Per iniziare a leggere o scrivere il *Chip Select* deve essere attivo, per leggere bisogna anche attivare il segnale dell'output per indicare se il dato selezionato dall'indirizzo è attualmente guidato sui pin. L'attivare l'output è comodo per connettere più memorie verso un solo bus di output e per capire verso quale memoria indirizzare il bus. Il tempo di accesso in lettura è solitamente indicato dal ritardo (*delay*) tra il tempo in cui l'output è abilitato e l'indirizzo è valido e il tempo che l'informazione si trova nell'output. I tempi di accesso variano tra i 0,5 e i 2,4 ns per le parti più veloci e piccole (CMOS, *complementary metal-oxide semiconductor*, si hanno tempi leggermente maggiori (tra i 2 e i 12 ns) per le parti più grandi. Sono state prodotte anche SRAM a basso consumo, sia in stand-by che in uso, generalmente dalle 5 alle 10 volte più lente. Sono state sviluppate anche SRAM sincrone. Per scrivere servono il dato da scrivere e l'indirizzo. Quando sia il *write enable* e il *chip select* sono settati true il dato viene scritto nell'indirizzo specificato. Ovviamente si hanno i soliti *setup time* e *hold time*. Inoltre il *write enable* non è controllato dal clock ma da un impulso *pulse* con requisiti minimi di ampiezza. Il tempo totale della scrittura è dato dai due *time* e da questo impulso. Grandi SRAM non possono essere costruite come il Register File a causa del 64k-to-1 multiplexor necessario per una 64K×1 SRAM. Memorie più grandi usano quindi una linea di output condivisa, chiamata **bit line**. Per permettere a più sorgenti di veicolare su una linea singola si usa un *three-state buffer* (*detto anche tristate buffer*) che è dotato di due input (un segnale con l'informazione e un *output enable*) e un singolo output (uno tra: *asserted*, *deasserted* e *high impedance*). L'output del *tristate buffer* è uguale all'input del segnale con l'informazione, o *asserted* o *deasserted*, se l'*output enable* è *asserted*, ed è altrimenti in uno stato di *high impedance* che permette ad un altro *tristate buffer*, con *output enable asserted*, di determinare il valore dell'output condiviso. I *tristate buffer* sono incorporati nei Flip-Flop che formano le celle base della SRAM.

Ecco una un gruppo di 3 *tristate buffer* che formano un multiplexor con un decoder come input:

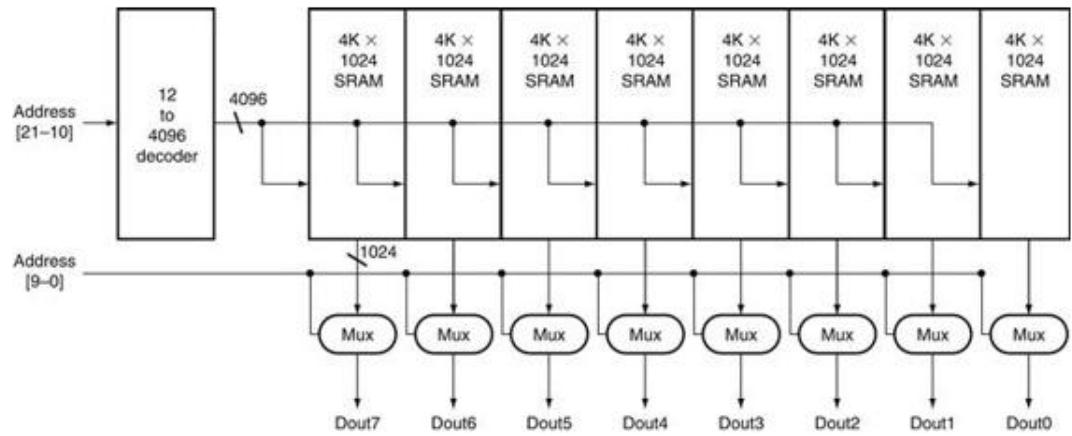


ed ecco una piccola SRAM 4×2 , costruita con un D Latch con un input chiamato *enable* che controlla il *three state output*:



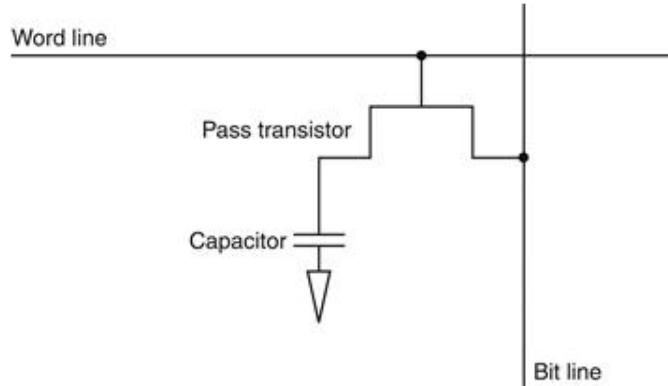
solo uno dei 4 input può essere asserito. Il decoder seleziona le celle da attivare, che a loro volta usano un three state output connesso alla linea verticale che fornisce l'informazione richiesta. L'indirizzo che seleziona la cella è mandato ad uno degli insiemi di linee orizzontali dette word lines. Si elimina l'uso di enormi multiplexor ma si ha ancora necessità di un grande decoder (per una $4m \times 8$ serve un

decoder 22-to-4M) e tante word lines. per ovviare a questo problema le grandi memorie vengono organizzate in un array di rettangoli come nell'esempio seguente:



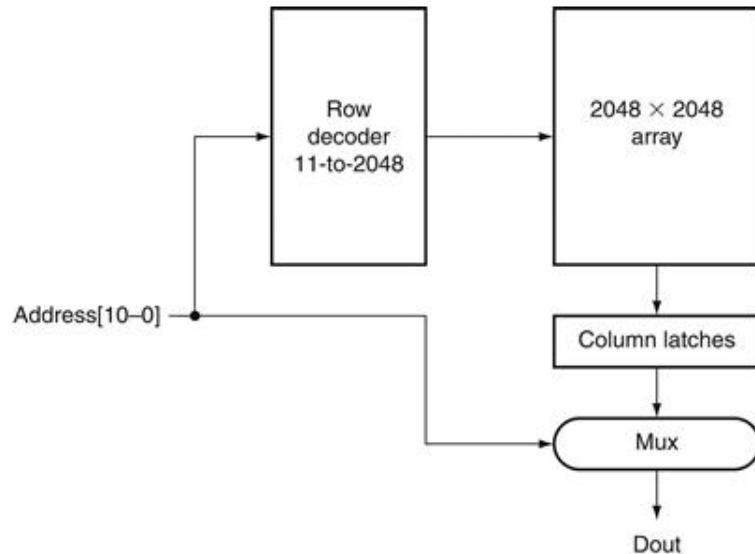
- **DRAM (Dynamic Random Access Memory):** in queste memorie il dato è salvato come una carica in un condensatore. Un singolo transistor è usato per accedere a questa carica salvata, sia per leggere che per sovrascrivere. Queste memorie sono più economiche (la SRAM richiede da 4 a 6 transistor per bit, la DRAM 1). A causa del fatto che salva la carica in un condensatore la DRAM deve essere periodicamente *refreshata* ed è per questo che è definita memoria *dinamica*. Per il refresh bisogna leggere il dato e riscriverlo, comunque la carica si mantiene per molti millisecondi, quindi per milioni di cicli di clock (sono in uso ora anche DRAM che refreshano senza basarsi sul clock del processore). Per permettere l'accesso nonostante i continui refresh la DRAM usa una struttura di decodifica a due livelli che permette di refreshare un'intera riga (*word line*) con un ciclo di lettura subito seguito da un ciclo di scrittura, occupando l'1%, massimo il 2%, dei cicli della DRAM, lasciando tutti gli altri per la lettura e la scrittura dei dati.

Il transistor dentro la cella è uno *switch*, chiamato *pass transistor*, come quello in figura:



quando il segnale sulla *word line* è *asserted* lo switch è chiuso, connettendo il condensatore alla *bit line*. Nel caso della scrittura si ha che il valore da scrivere si mette nella *bit line*, se è 1 il condensatore si carica, altrimenti si scarica. La lettura è invece più complessa in quanto la DRAM deve leggere cariche molto piccole del condensatore. Dopo aver attivato la *word line* per la lettura la *bit line* si carica a metà e la carica del condensatore viene letta nella *bit line* che si muove in alto o in basso. Questo cambiamento viene captato da un amplificatore che individua piccoli cambiamenti di voltaggio.

La DRAM usa un decoder a due livelli, prima un accesso sulla riga e poi uno sulla colonna, come mostrato in figura:



L'entrata sulle righe sceglie un numero di righe e attiva il corrispondente *word line*, il contenuto di tutte le colonne nelle righe attive viene

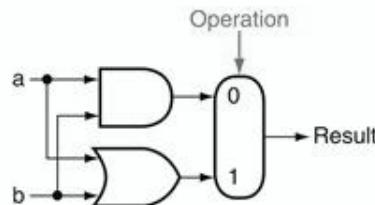
salvato nei Latch. Due segnali, il RAS, *Row Access Strobe*, e il CAS, *Columns Access Strobe*, segnalano quando righe e colonne sono state rifornite con i dati. Questo sistema rende l'accesso alla DRAM più lungo, tra i 50 e 970 ns. Una DRAM 64M×4 tiene 8K accessi su ogni riga, ma ne salva solo 4 con l'accesso alle colonne

4.1 ALU

arithmetic logic unit è l'unità che esegue le operazioni aritmetica (*add*, *sub*, etc...) e quelle logiche (*and*, *or*). La ALU è composta da porte OR, porte NAD, multiplexor e inverter.

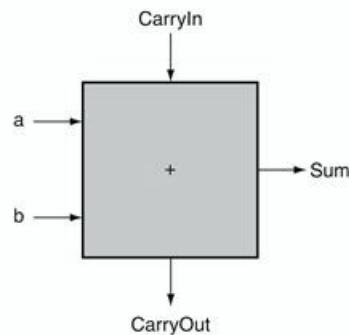
4.1.1 ALU a 1bit

ovviamente le operazioni logiche sono più semplici, come si può qui vedere:



si ha un multiplexor a destra che sceglie AND o OR in base al valore di *operation*, 0 per AND e 1 per OR.

Vediamo l'addizione. Un sommatore deve avere 2 input e un bit di output per la somma. C'è poi un secondo output, il *CarryOut*, contenente il resto della somma che è collegato come terzo input, *CarryIn*, ad un sommatore vicino (schema in figura). CarryOut e somma possono essere espresse come espressioni logiche, secondo la seguente tabella:



Inputs			Outputs		Comments
a	b	CarryIn	CarryOut	Sum	
0	0	0	0	0	$0 + 0 + 0 = 00_{\text{two}}$
0	0	1	0	1	$0 + 0 + 1 = 01_{\text{two}}$
0	1	0	0	1	$0 + 1 + 0 = 01_{\text{two}}$
0	1	1	1	0	$0 + 1 + 1 = 10_{\text{two}}$
1	0	0	0	1	$1 + 0 + 0 = 01_{\text{two}}$
1	0	1	1	0	$1 + 0 + 1 = 10_{\text{two}}$
1	1	0	1	0	$1 + 1 + 0 = 10_{\text{two}}$
1	1	1	1	1	$1 + 1 + 1 = 11_{\text{two}}$

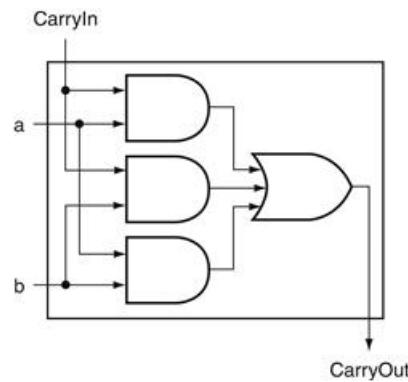
Si ha la seguente espressione dalla tabella per il CarryOut:

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b) + (a \cdot b \cdot \text{CarryIn})$$

Se $a \cdot b \cdot \text{CarryIn}$ è true allora anche gli altri tre termini devono esserlo quindi possiamo semplificare:

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

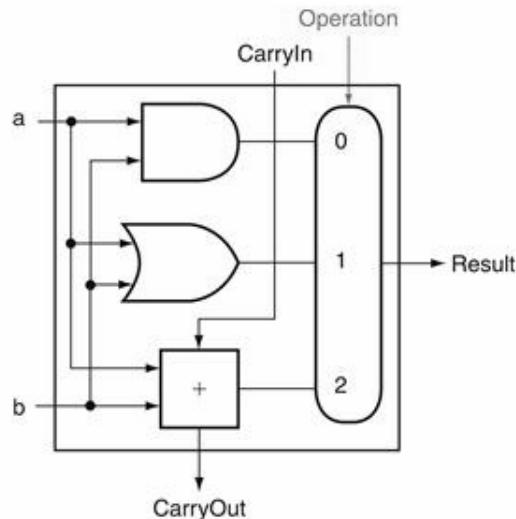
ottenendo questo circuito per ottenere il CarryOut:



La somma normale invece è settata in modo da avere 1 in output solo se 1 o 3 ingressi sono 1 con la seguente equazione:

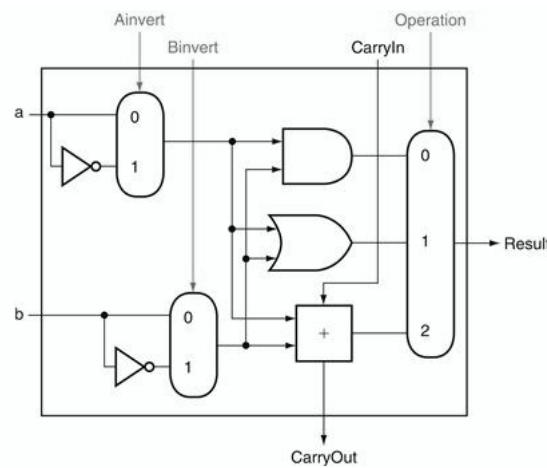
$$sum = (a \cdot \bar{b} \cdot \overline{CarryIn}) + (\bar{a} \cdot b \cdot \overline{CarryIn}) + (\bar{a} \cdot \bar{b} \cdot CarryIn) + (a \cdot b \cdot CarryIn)$$

Aggiungendo quindi il sommatore del CarryOut allo schema per le equazioni logiche si ottiene:

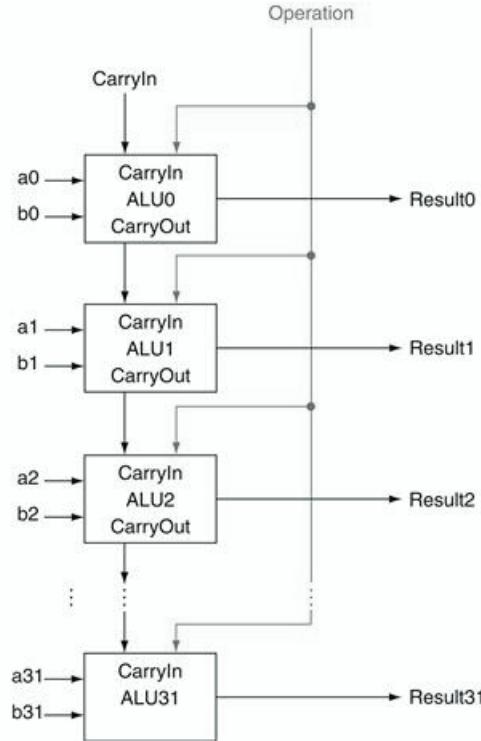


4.1.2 ALU a 32bit

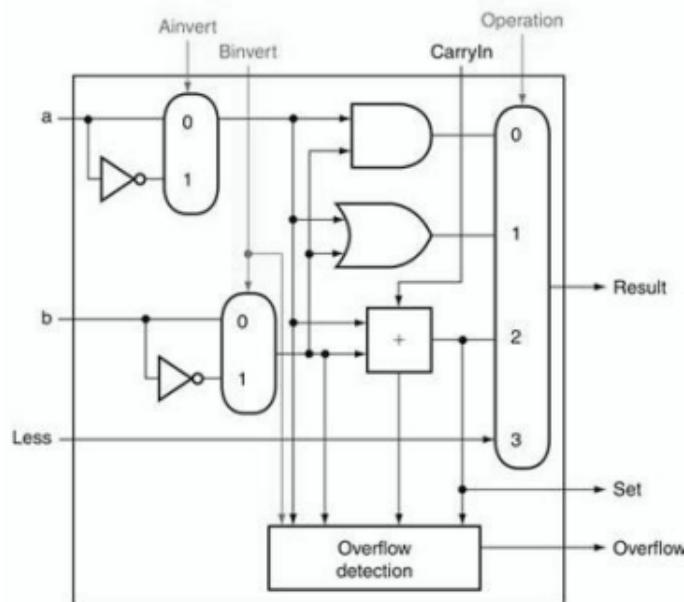
Si ottiene collegando 32 ALU a 1bit ottenendo così 32 risultati possibili. Si può per esempio avere la sottrazione, che consiste nel sommare l'inverso di un operando, aggiungendo 1 per il *complemento a 1*. Per invertire l'operando si aggiunge semplicemente un multiplexor 2:1 che sceglie, per esempio tra b e \bar{b} , ottenendo:



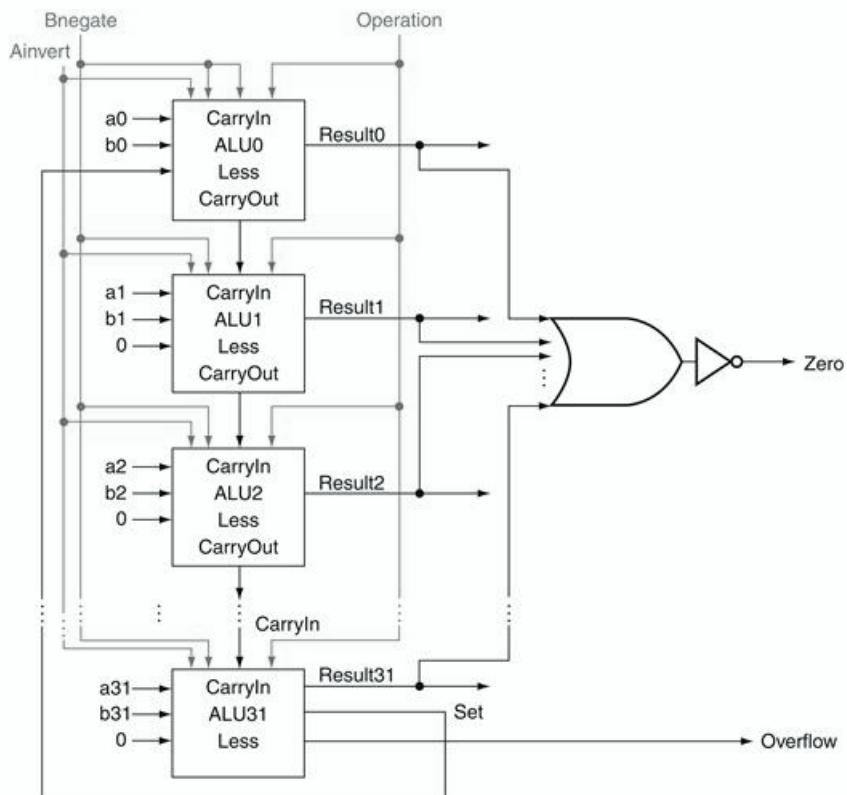
e nel complesso la seguente rappresentazione dell'ALU a 32bit:



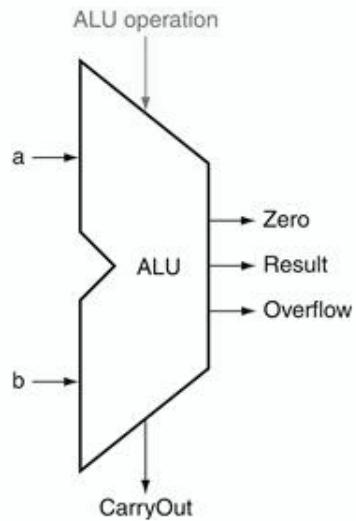
Aggiungiamo ora un'operazione di comparazione, la *slt*, che ha 1 come output se $rs < rt$ o 0 se viceversa. Per farlo si aggiunge l'input *Less* usato solo per questa istruzione, e si connette 0 al bit meno significativo della ALU, si usa così il bit meno significativo unicamente per questa istruzione. Inoltre ora si può calcolare anche un valore negativo, usando il bit più significativo per identificarlo, 1 per un numero negativo e 0 per uno positivo. Dato che il bit più significativo per la *slt* non è l'output del sommatore si dovrà usare il valore di input *Less*. Si necessita quindi di un bit aggiuntivo per il bit più significativo che ha un extra output: l'output del sommatore. Si crea quindi un nuovo output del sommatore, detto *set*, usato solo dalla *slt*. Inoltre si aggiunge l'*overflow detector* apposta per questo bit Più significativo della ALU. Si ha quindi:



e nel complesso ecco l'ALU a 32bit, con l'aggiunta del controllo che il risultato sia zero, per testare l'uguaglianza ($\text{zero} = \sum_{i=0}^{31} \overline{\text{result} - i}$):

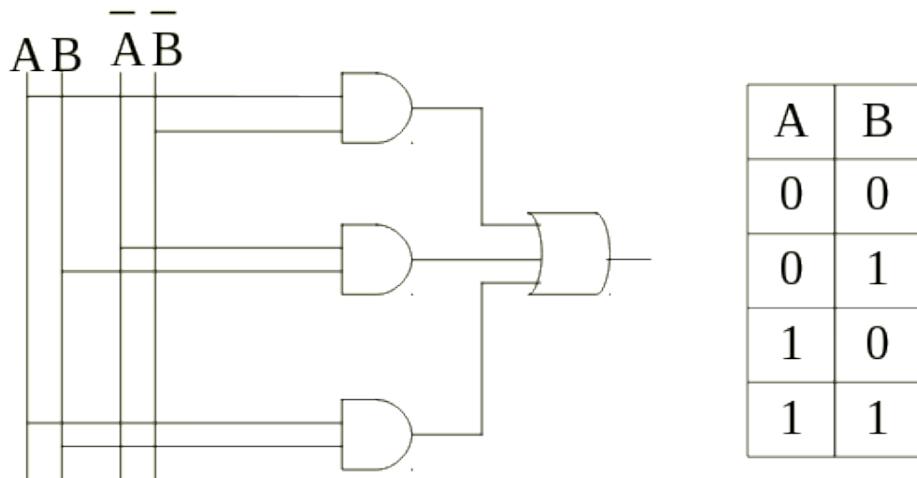


solitamente indicata col seguente simbolo:



4.2 Esercizi

Esercizio 11. Dato il seguente circuito con la seguente tabella di ingressi definire l'output:

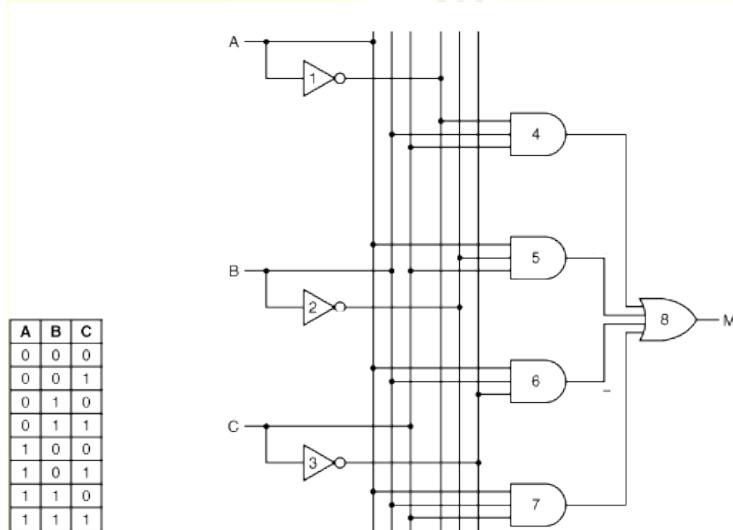


Si ha la seguente tabella della verità:

A	B	\bar{A}	\bar{B}	$A \wedge \bar{B}$	$\bar{A} \wedge B$	$A \wedge B1$	$final \vee$
0	0	1	1	0	0	0	0
0	1	1	0	0	1	0	1
1	0	0	1	1	0	0	1
1	1	0	0	0	0	1	1

Quindi come output si avrà in ordine $M = [0, 1, 1, 1]$

Esercizio 12. Dato il seguente circuito con la seguente tabella di ingressi definire l'output:



Si ha la seguente tabella della verità:

A	B	C	\bar{A}	\bar{B}	\bar{C}	$\bar{A} \wedge B \wedge C$	$A \wedge \bar{B} \wedge C$	$A \wedge B \wedge \bar{C}$	$A \wedge B \wedge C$	$final \vee$
0	0	0	1	1	1	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0
0	1	0	1	0	1	0	0	0	0	0
0	1	1	1	0	0	1	0	0	0	1
1	0	0	0	1	1	0	0	0	0	0
1	0	1	0	1	0	0	1	0	0	1
1	1	0	0	0	1	0	0	1	0	1
1	1	1	0	0	0	0	0	0	1	1

Quindi come output si avrà in ordine $M = [0, 0, 0, 1, 0, 1, 1, 1]$

Esercizio 13. Data la seguente tabella con input e output definire la funzione di D (usando gli OR di AND) e disegnare il circuito:

Ingressi			Uscita
A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Iniziamo scrivendo la tabella di verità con ciò che conosciamo:

A	B	C	\bar{A}	\bar{B}	\bar{C}	$\wedge\wedge$	$\wedge\wedge$	$\wedge\wedge$	final \vee
0	0	0	1	1	1				0
0	0	1	1	1	0				1
0	1	0	1	0	1				1
0	1	1	1	0	0				0
1	0	0	0	1	1				1
1	0	1	0	1	0				0
1	1	0	0	0	1				0
1	1	1	0	0	0				0

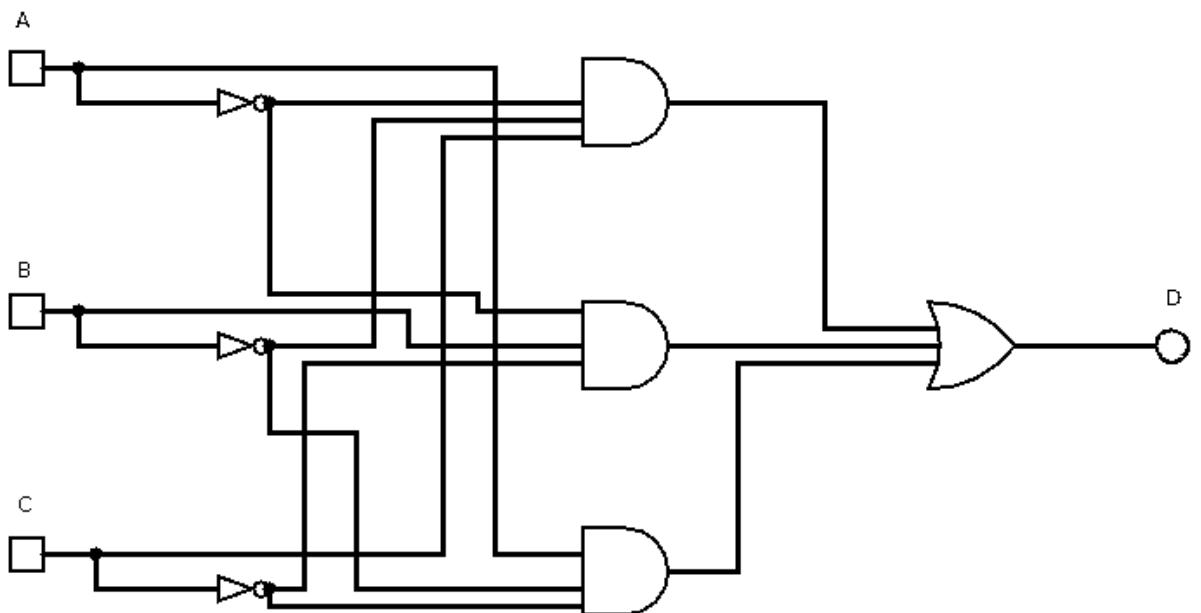
si avranno 3 doppi AND tra 3 degli elementi delle prime 6 colonne. Dato che l'uscita è formata dall'OR delle prime tre colonne posso intuire che per avere 0 devo avere tutti e tre i campi 0, si ottiene quindi:

A	B	C	\bar{A}	\bar{B}	\bar{C}	$\wedge\wedge$	$\wedge\wedge$	$\wedge\wedge$	$final \vee$
0	0	0	1	1	1	0	0	0	0
0	0	1	1	1	0				1
0	1	0	1	0	1				1
0	1	1	1	0	0	0	0	0	0
1	0	0	0	1	1				1
1	0	1	0	1	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0
1	1	1	0	0	0	0	0	0	0

quelli con 1 in uscita comportano la presenza di almeno un 1 come risultato di uno dei tre doppi and, per avere 1 in uscita da un doppio end si necessita di avere le 3 entrate uguali a 1. Si nota così che per la seconda riga servono \bar{A}, \bar{B}, C , ottenendo 1,0,0 per la terza \bar{A}, B, \bar{C} , ottenendo 0,1,0 per la quinta A, \bar{B}, C , ottenendo 0,0,1 e avendo infine:

A	B	C	\bar{A}	\bar{B}	\bar{C}	$\bar{A} \wedge \bar{B} \wedge C$	$\bar{A} \wedge B \wedge \bar{C}$	$A \wedge \bar{B} \wedge \bar{C}$	$final \vee$
0	0	0	1	1	1	0	0	0	0
0	0	1	1	1	0	1	0	0	1
0	1	0	1	0	1	0	1	0	1
0	1	1	1	0	0	0	0	0	0
1	0	0	0	1	1	0	0	1	1
1	0	1	0	1	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0
1	1	1	0	0	0	0	0	0	0

Ed ecco il circuito:



- Esercizio 14.** • un multiplexor a 32 ingressi quanti ingressi di selezione deve avere?

Si devono avere $\log_2(32) = 5$ ingressi di selezione

- un decoder con 1024 possibili uscite ha un ingresso di quanti bit?
Si ha $2^x = 1024$ quindi $x = 10$ bit in ingresso
- Considerando una RAM contenente 4096 Kbit la cui parola di memoria è di 32 bit, un indirizzo per tale memoria da quanti bit è composto?

$$4096 \text{ Kbit} = 2^{12} \text{ Kbit} = 2^{22} \text{ bit}$$

L'altezza della memoria:

$$\frac{2^{22}}{32} = 2^{17}$$

Ogni indirizzo di questa memoria è rappresentato su 17 bit

- Considerando una RAM contenente 10 bit e un indirizzo di memoria di 3 bit, da quanti bit è composta una parola di memoria?
Larghezza ram = dimensione / numero di parole indirizzabili:

$$\frac{10}{2^3} = 1,2$$

quindi una word ha 2 bit (la memoria è composta da 5 parole di memoria e 3 indirizzi non vengono usati)

- Con 30 bit per un indirizzo di memoria e una parola di memoria di 64 bit, qual è la dimensione massima di memoria indirizzabile in MB?

$$\frac{x}{2^{30}} = 64 \longrightarrow x = 64 \cdot 2^{30} = 6,87 \times 10^{10} \text{ bit}$$

divido per 8 e ottengo i byte, per 1024 e ottengo i Kilobyte e per 1024 ancora e ottengo 8192 Megabyte

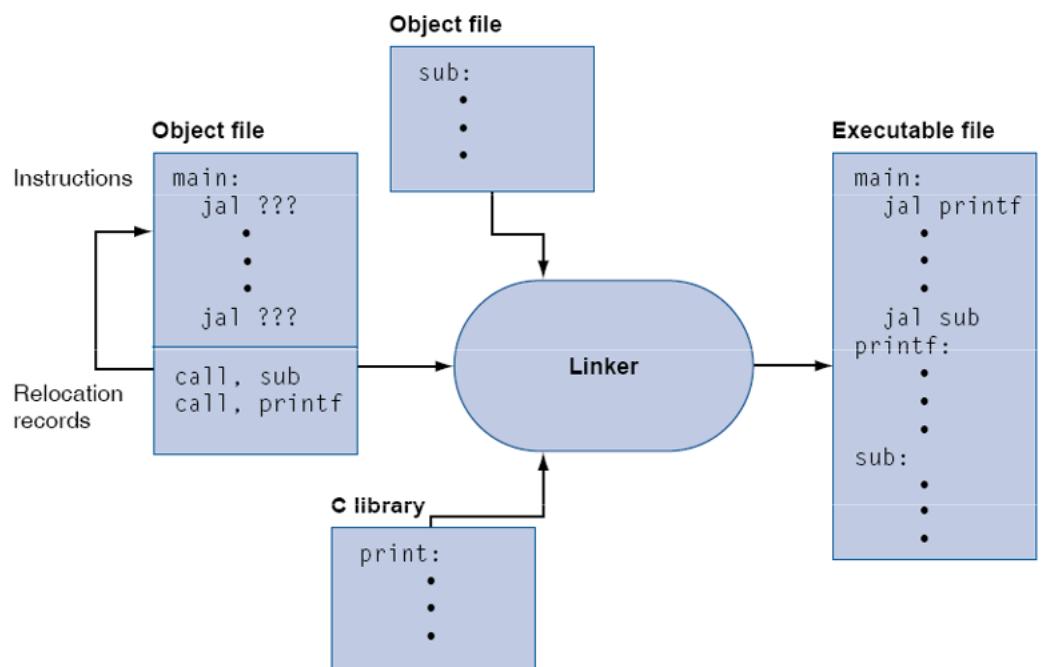
Capitolo 5

Assembly

Prima di tutto analizziamo cosa accadde quando si compila un programma di alto livello:

- il compilatore trasforma il programma in linguaggio assembler, una forma simbolica di ciò che il calcolatore è in grado di interpretare
- l'assemblatore traduce il linguaggio assembler in linguaggio macchina. L'hardware MIPS si assicura che \$zero contenga sempre il valore 0 e permette il caricamento di un registro di costanti a 32bit, nonostante il limite a 16bit per le I-Type. L'assemblatore accetta numeri in diverse basi. Si converte quindi il programma assembler in un *file oggetto*, ovvero una sequenza di istruzioni in linguaggio macchina, dati e informazioni necessarie a collocare le istruzioni in memoria nella posizione corretta. L'assemblatore deve determinare gli indirizzi corrispondenti a tutte le etichette, e ne tiene traccia nei salti e nei trasferimenti dati scrivendole in una *tabella dei simboli* che conterrà coppie di tipo *simbolo-indirizzo*
- per evitare di dover ripetere da 0 tutte le operazioni indicate ogni volta che si cambia qualcosa nel sorgente. Per ovviare a questo problema si è deciso di compilare separatamente ogni procedura e si è creato un programma di sistema chiamato *Linker (Link Editor)* che prende tutte le procedure e le unisce. Avvengono 3 passaggi:
 1. il Linker inserisce in memoria in modo simbolico il codice e i moduli dati
 2. determina gli indirizzi dei dati e delle etichette che compaiono nelle istruzioni
 3. corregge i riferimenti interni ed esterni

il Linker utilizza le informazioni di rilocazione e la tabella dei simboli di ciascun modulo oggetto per risolvere le etichette non definite, che si trovano nei salti e nei salti incondizionati. Nel complesso si può dire che trova gli indirizzi vecchi e li sostituisce con quelli nuovi. Si ha che è più veloce correggere in questa maniera il codice piuttosto che ricompilerlo e riassemblarlo. Dopo che tutti i riferimenti esterni sono stati risolti il Linker determina le locazioni di memoria che ciascun modulo dovrà occupare. Quando un Linker inserisce un modulo in memoria tutti i riferimenti *assoluti* (ovvero quelli non definiti in relazione ad un registro) devono essere rilocati in modo da poter riflettere la loro posizione reale. Il Linker produce un *file eseguibile* eseguibile su un calcolatore, che ha lo stesso formato di un *file oggetto* ma non contiene più riferimenti irrisolti (questo passaggio può essere eseguito in maniera parziale con indirizzi non risolti, come nel caso delle librerie).



- una volta che l'eseguibile è salvato su disco il sistema operativo può leggerlo, trasferirlo in memoria ed eseguirlo. Si ha il programma di caricamento, il Loader, che effettua i seguenti passaggi:
 1. lettura dell'intestazione dell'eseguibile per determinare la lunghezza del segmento testo e del segmento dati
 2. creazione di uno spazio di indirizzamento sufficiente per testo e dati

3. copia di istruzioni e dati dal file alla memoria
4. copia nello stack degli eventuali parametri passati al programma principale
5. inizializzazione dei registri del calcolatore e impostazione dello stack pointer affinché punti alla prima locazione libera
6. salto ad una procedura di avvio, *startup*, che copia i parametri nei registri argomento e chiama la procedura principale del programma (*main*), quando quest'ultima restituisce il controllo la procedura di *startup* termina il programma con una chiamata alla funzione di sistema *exit*

Tornando a parlare di CPU si hanno 2 filosofie di progetto:

- **RISC (Reduced Instruction Set Computing)**: qui si hanno poche istruzioni semplici, una struttura circuitalmente semplice, l'esecuzione veloce di ogni singola istruzione ma occorrono più istruzioni per fare cose anche semplici. Come esempi abbiamo MIPS, ARM o PowerPc.
- **CISC (Complex Instruction Set Computing)**: qui si hanno istruzioni complesse, struttura circuitale complicata, esecuzione più lenta di ogni singola istruzione ma occorrono meno istruzioni. Come esempi abbiamo Intel x86 (Pentium etc..)

Iniziamo a vedere MIPS. Si hanno 32 registri di 32bit e istruzioni di 32bit. Si hanno manipolazioni di dati solo sui registri, si può avere il trasferimento dei dati dalla memoria nei registri e si possono avere alterazioni del flusso di controllo con i *salti*.

Iniziamo con qualcosa di semplice: $somma = a + a + a + a$ e a si trova nel registro \$8:

```
add $9, $8, $8 # metto nel registro $9 la somma a+a  
add $9, $9, $9 # metto nel registro $9 la somma (a+a)+(a+a),  
               in quanto (a+a) e nel registro $9
```

vediamo anche le operazioni semplici di *load word*:

```
lw $10, 4($8) # carica nel registro $10 il contenuto della  
parola (a 32bit) contenuta all'indirizzo di  
memoria ottenuto come somma del registro $8  
e dell'offset immediato 4
```

e *store word*:

```
sw $10, 4($8) # memorizza il contenuto del registro $10  
parola (a 32bit) all'indirizzo di  
memoria ottenuto come somma del registro $8  
e dell'offset immediato 4
```

vediamo un esempio più complesso:

i valori delle variabili a,b,c e d sono memorizzati nella memoria partendo dall'indirizzo specificato dal registro \$s0. Aggiungo il valore immediato 10 a tutte le variabili e le salvo in memoria:

```
lw $t0, 0($s0)      # carico in $t0 il contenuto di $s0, senza  
nessun offset, ora $t0 è a  
addi $t0, $t0, 10 # aggiungo 10 ad a  
sw $t0, 0($s0)      # salvo il valore in memoria nel registro  
$s0  
lw $t0, 4($s0)      # carico in $t0 il contenuto del registro  
ottenuto con un offset di 4 da $s0,  
ora $t0 è b  
addi $t0, $t0, 10 # aggiungo 10 a b  
sw $t0, 4($s0)      # salvo il valore in memoria nel registro  
$s0 con un offset di 4  
lw $t0, 8($s0)      # carico in $t0 il contenuto del registro  
ottenuto con un offset di 8 da $s0,  
ora $t0 è c  
addi $t0, $t0, 10 # aggiungo 10 a c  
sw $t0, 8($s0)      # salvo il valore in memoria nel registro  
$s0 con un offset di 8  
lw $t0, 12($s0)     # carico in $t0 il contenuto del registro  
ottenuto con un offset di 12 da $s0,  
ora $t0 è d  
addi $t0, $t0, 10 # aggiungo 10 a d  
sw $t0, 8($s0)      # salvo il valore in memoria nel registro  
$s0 con un offset di 12
```

ecco un altro breve codice: *calcolo la somma del valore 10, memorizzato in \$s1 e del valore 20 memorizzato nella allocazione di memoria specificata da \$s0 con un offset di 56 byte. Il risultato va memorizzato nella memoria che si trova 3 word più avanti rispetto alla locazione attuale del secondo operando:*

```
lw $t0, 56($s0)    # carico in $t0 il contenuto del registro  
                    ottenuto con un offset di 56 byte da $s0  
add $t1, $s1, $t0 # faccio la somma tra 10, contenuto in $s1  
                    e 20, caricato nella riga precedente in $t0  
                    e metto il risultato in $t1  
sw $t1, 68($s0)    # salvo la somma nell'indirizzo di memoria  
                    ottenuto con un offset di 56 byte, quello  
                    del secondo operando, sommato di 3 word  
                    ovvero sommato di 12 byte, quindi avrà  
                    un offset di 56+12=68
```

introduciamo i flussi di controllo *if-else*:
scrivo in MIPS il seguente codice java:

```
if(i==j)
    f=g+h
else
    f=g-h
```

considerando che le 5 variabili sono agli indirizzi compresi tra \$s0 e \$s4.
L'indirizzo della prima istruzione è 80000:

```
80000 bne $s3, $s4, ELSE      # branch if not equal, se $s3 (i)
                                # e $s4 (j) non sono uguali salta
                                # all'etichetta ELSE
80004 add $s0, $S1, $s2      # se non si è saltato a ELSE,
                                # ovvero i valori in $s3 e $s4 sono
                                # uguali, esegui la somma tra $s1 (g)
                                # e $s2 (h) e salvala in $s0 (f)
80008 j EXIT                 # se è stata effettuata la somma salta
                                # all'istruzione exit
80012 ELSE: sub $s0, $S1, $s2 # l'etichetta ELSE riporta la sua
                                # istruzione, ovvero la sottrazione
                                # tra gli stessi indirizzi della
                                # somma che si avrebbe avuto
                                # se fossero stati uguali i e j.
                                # Si arriva a questa istruzione
                                # grazie al primo bne, se i e j sono diversi
80016 EXIT: ...              # si esce dal programma, si arriva
                                # qui col jump alla terza riga
                                # se si è effettuata la somma o
                                # dopo la sottrazione
```

elenchiamo le istruzioni I-Type di tipo *branch*:

- **beq**: *branch on equal*, va al target definito dall'etichetta se i due registri contengono lo stesso valore
- **bne**: *branch if not equal*, va al target definito dall'etichetta se i due registri contengono valori diversi
- **bgez** *Branch on greater than or equal to zero* va al target definito dall'etichetta se il valore è maggiore uguale di zero

- **bgtz** *Branch on greater than zero* va al target definito dall'etichetta se il valore è maggiore di zero
- **blez** *Branch on less than or equal to zero* va al target definito dall'etichetta se il valore è minore uguale di zero
- **bltz** *Branch on less than zero* va al target definito dall'etichetta se il valore è minore di zero
- **bgezal** *Branch on greater than or equal to zero and link* va al target definito dall'etichetta se il valore è maggiore uguale di zero e salva l'indirizzo del return in \$31
- **bltzal** *Branch on less than or equal to zero and link* va al target definito dall'etichetta se il valore è minore uguale di zero e salva l'indirizzo del return in \$31

Prima di continuare è necessario analizzare gli scopi dei vari registri. Ecco una comoda tabella:

Nome Simbolico	Numero	Uso
\$zero	0	Costante 0
\$at	1	Assembler temporary
\$v0-\$v1	2-3	Functions and expressions evaluation
\$a0-\$a3	4-7	Arguments
\$t0-\$t7	8-15	Temporaries
\$s0-\$s7	16-23	Saved Temporaries
\$t8-\$t9	24-25	Temporaries
\$k0-\$k1	26-27	Reserved for OS kernel
\$gp	28	Global pointer
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Return address

Tutti i registri il cui uso è scritto in grigio chiaro sono usati da assembler, compilatore e sistema operativo (ovvero tutti tranne i \$tn, parte dei \$sn e \$zero) e sono da trattare con cautela.

Programmando in assembly si possono avere le direttive date all’assembler per consentirgli di, per esempio:

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

vediamo quale esempio di direttive:

- *.data <addr>*, quel che segue va nel segmento dati, eventualmente all’indirizzo addr
- *.byte b1,...,bn* inizializza i valori in byte successivi
- *.word w1,...,wn* inizializza i valori in word successive
- *.text <addr>*, quel che segue va nel segmento text (programma), eventualmente all’indirizzo addr

vediamo un esempio ”completo”:

```
.data
item: .word 1
      .text
      .globl main
main: lw $t0,item
```

analizziamo ora qualche istruzione aritmetica e logica:

- *add rd, rs, rt*: è l’addizione, $rd=rs+rt$, può andare in overflow
- *addi rd, rs, imm*: è l’addizione immediata, $rd=rs+imm$ (con imm che può essere negativo), può andare in overflow
- *and rd, rs, rt*: è l’*and bit a bit* di *rs* e *rt*, col risultato salvato in *rd*
- *andi rd, rs, imm*: è l’*and bit a bit* di *rs* e *imm* (esteso anche a 0), col risultato salvato in *rd*
- *or rd, rs, rt*: è l’*or bit a bit* (logical *or*) di *rs* e *rt*, col risultato salvato in *rd*
- *ori rd, rs, imm*: è l’*or bit a bit* (logical *or*) di *rs* e *imm* (esteso anche a 0), col risultato salvato in *rd*

- *sll rd, rt, shamt*: è lo *shift left* di *rd* della distanza *shamt*, col risultato in *rd*
- *lui rt, imm*: è *load upper immediate*, carica i 16bit del valore immediato nei 16bit più significativi di *rt* e i rimanenti 16bit meno significativi vengono messi a 0. Se si vuole salvare un valore a 32bit prima si fa *lui \$s0, imm1*, con *imm1* decimale che rappresenta i primi 16bit del valore, caricando quindi i 16bit più significativi (quelli a sinistra), e poi si fa *ori \$s0, \$s0, imm2*, con *imm2* che rappresenta il decimale dei secondi 16bit, caricando i 16bit meno significativi (quelli a destra)

Si hanno poi le *pseudo istruzioni*, istruzioni assembly che non hanno una corrispondente istruzione macchina che vengono tradotte dall'assembler in una sequenza di istruzioni. Per esempio di può avere *li rd, imm*, ovvero *load immediate* che carica una costante di 32bit in *rd* (esempio *li \$s0, 4000000* che l'assemblatore traduce in *lui \$s0, 61; ori \$s0, \$s0, 2304*). Un altro esempio può essere *move \$t0, \$t1*, che sposta il contenuto di *\$t1* in *\$t0*, che l'assemblatore traduce con *add \$t0, \$zero, \$t1*, somma a *\$t1* zero e lo salva in *\$t0*. Vediamo un esercizio:

si leggano 3 valori a,b e c memorizzati all'etichetta valori nei registri \$t0, \$t1 e \$t2. Calcolo la somma dei 3 in \$s0 se il primo è positivo, il prodotto se il primo è negativo e l'AND se è uguale a 0. memorizzo il risultato nella locazione subito dopo il terzo valore

```

.data
valori: .word 0, 5, 2
        .word 0
.text
.globl main

main:   la $t7, valori      # i valori si troveranno a partire dal registro $t7

        lw $t0, 0($t7)      # carico i valori in $t0, $t1 e $t2
                            usando i vari offset a step di 4
        lw $t1, 4($t7)
        lw $t2, 8($t7)

        bgt $t0, $zero, than # se il valore di $t0 è positivo
                            vado all'etichetta than
        blt $t0, $zero, else # se il valore di $t0 è negativo
                            vado all'etichetta else

```

```

        and $s0, $t1, $t2      # se arrivo qui significa che $t0 è 0
                                quindi calcolo l'AND tra $t1 e $t2
                                e salvo in $s0
        j fine
else:   mul $s0, $t0, $t1      # dopo il calcolo dell'and salto all'etichetta fine
                                # calcolo il prodotto dei contenuti dei
                                # tre registri se $t0<0
        mul $s0, $s0, $t2
        j fine
than:   add $s0, $t0, $t1      # calcolo la somma dei contenuti dei
                                # tre registri se $t0>0
        add $s0, $s0, $t2
        j fine
fine:   sw $s0, 12($t7)      # alla fine salvo nella locazione dopo
                                il terzo valore, quindi con un offset di 12
                                da $t7

```

Ecco un altro esercizio: *calcolo la lunghezza di una stringa memorizzata dall'indirizzo **stringa**. La stringa finisce col carattere \0. Salvo la dimensione della stringa in memoria dopo la stringa in un word*:

```

.data
stringa: .asciiz "Hello World!" # definisco la stringa
                                13 byte per la stringa
                                12 byte per la lunghezza
dim:     .word 0
charfine: .asciiz ""           # char finale
.text
.globl main

main:    la $t0, stringa      # carico in $t0 l'indirizzo della stringa
        add $t2, $zero, $zero # definisco un contatore
        la $s5, charfine      # carico in $t0 l'indirizzo di charfine
        lb $s1, 0$(s5)         # load byte, metto in $s1 il contenuto
                                # di $s5: ""

ciclo:
        lb $s7, 0($t0)        # carico il $s7 il char 0 della stringa
        beq $s7, $s1, fine     # se $s7 è uguale a $s1, ovvero a ""
                                # salta a fine
        addi $t2, $t2, 1        # incrementa il contatore
        addi $t0, $t0, 1        # incremento di un char
        j ciclo                 # torno ad inizio ciclo
                                dove in $s7 ci sarà il char successivo

```

```
fine:    la $t3, dim          # carico in $t3 l'indirizzo di dim
         sw $t2, 0($t3)      # carico il risultato in $t3
```

Un altro esercizio: *Calcolo il minimo tra a,b e c memorizzati nella memoria all'etichetta "numeri". Salvo all'etichetta "minimo":*

```
.data
numeri: .word 9, 7, 8
minimo: .word 0
.text
.globl main

main:   .la $s0, numeri
        .la $s1, minimo
        .lw $t0, 0($s0)
        .lw $t1, 4($s0)
        .lw $t2, 8($s0)

        blt $t0, $t1, AminB # verifico se a<b, se vero vado a AminB
        blt $t1, $t2, BminC # verifico se b<c, se vero vado a BminC
        move $t3, $t2      # se arrivo qui ho b<a e c<b, quindi c
                           # è il minimo, e lo salvo in $t3
        j salvamin
BminC:  move $t3, $t1      # se in main arrivo a BminC
                           # significa che b<a e b<c
                           # quindi b è il minimo e lo salvo
        j salvamin
AminB:  blt $t0, $t2, AminC # verifico se a<c, se vero vado a AminC
        move $t3, $t2      # se arrivo qui ho a<b e c<a,
                           # c è il minimo e lo salvo
        j salvamin
AminC:  move $t3, $t0      # se arrivo qui ho a<b e a<c,
                           # a è il minimo e lo salvo
salvamin: sw $t3, 0($s1)  # salvo $t3 in minimo
```

Iniziamo a vedere come funzionano le procedure. Introduciamo delle istruzioni utili:

- *jal <IndirizzoProcedura>: jump and link*, salva nel registro 31 \$ra, detto *return address* l'indirizzo a cui tornare dopo la procedura, ovverlo l'indirizzo successivo (+4) a quello della *jal*

- *jr <registro>: jump register*, salta all'indirizzo contenuto in un registro, costituito da saltare a qualsiasi locazione di memoria. *jr \$ra* è uno degli utilizzi tipici, esegue il ritorno da una procedura saltando all'indirizzo precedentemente salvato da *jal*

Si hanno delle convenzioni base, si hanno infatti dei registri usati apposta dalle procedure:

- *\$a0-\$a3* sono i registri argomento per il passaggio di parametri
- *\$v0-\$v1* sono i registri valore per la restituzione dei risultati

Un parametro può essere sia un dato che un indirizzo. Vediamo qualche esempio elementare:

programma che definisce 4 numeri, definisce una procedura somma1 che riceve due numeri e fa la somma, viene chiamata prima tra num1 e num2 salvando in result1 e poi tra num3 e num4 salvando in result2

```
.data
num1:    .word 50
num2:    .word 14
result1: .word 0
num3:    .word 50
num4:    .word -66
result2: .word 0
        .text
        .globl main

main:    lw $a0, num1      # carico in $a0 num1
        lw $a1, num2      # carico in $a1 num2
        jal somma1       # l'indirizzo della prossima istruzione viene
                           # salvato in $ra e si salta alla procedura
        sw $v0, result1   # salvo result1

        lw $a0, num3      # carico in $a0 num3
        lw $a1, num4      # carico in $a1 num4
        jal somma1       # l'indirizzo della prossima istruzione viene
                           # salvato in $ra e si salta alla procedura
        sw $v0, result2   # salvo result2
somma1:  add $v0, $a0, $a1 # secondo le convenzioni viste sopra
        j $ra             # $ra contiene l'indirizzo di ritorno
```

In una situazione reale si deve tenere conto anche del sistema operativo, che è un insieme di programmi che stanno in un'area protetta della memoria e svolgono funzioni di utilità generale (esempio I/O), richiamabili dai programmi utente. Alcune funzioni base possono essere richiamate col meccanismo di *syscall*, concettualmente analogo alle procedure. Si ha la seguente tabella con le convenzioni per le syscall:

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	

si imposta i \$v0 il codice della chiamata e i parametri nei registri \$a0-\$a3. Facciamo qualche esempio:

codice che stampa "la risposta è 5":

```

.data
str:    .asciiz "la risposta è"
numero: .word 5

.text
li $v0, 4      # codice per la syscall print_str
la $a0, str    # indirizzo della stringa da stampare
                # passato per indirizzo
syscall        # stampa la stringa

li $v0, 1      # codice per la syscall print_int
lw $a0, numero # numero da stampare passato per valore
syscall        # stampa il numero

```

Torniamo a parlare dei registri, soprattutto dell'uso dei registri all'interno di procedure. Si hanno convenzioni per i \$t e i \$s:

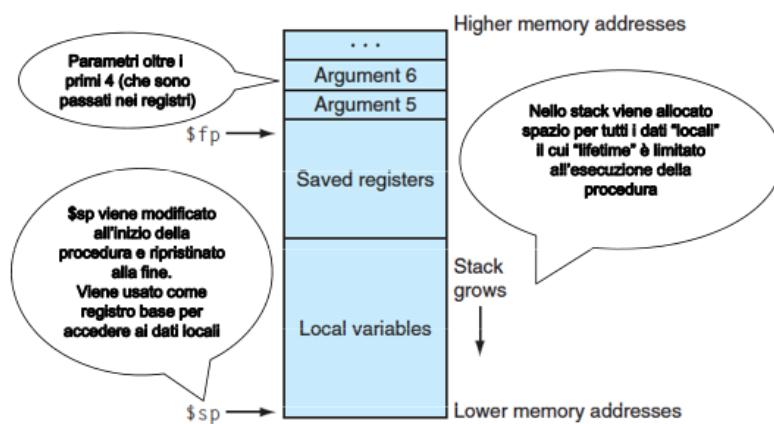
- *\$t*: sono i registri *temporary* e non sono salvati dalla procedura. Il chiamante non si può aspettare di trovarli immutati dopo una procedura e devono essere salvati prima della chiamata a procedura
- *\$s*: sono i registri *saved* e sono salvati dalla procedura. Il chiamante ha diritto di aspettarsi che siano immutati dopo una chiamata a procedura. Se la procedura usa questi registri si deve salvarne il contenuto all'inizio e ripristinarlo dopo, grazie all'uso dello stack

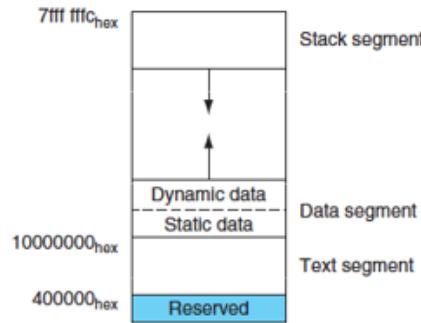
Si distinguono due tipi di procedure:

1. *procedure foglia*: procedure che non chiamano altre procedure
2. *procedure non foglia*: procedure che chiamano altre procedure, in questo caso bisogna far sì che una procedura *non foglia* salvi il contenuto in \$ra e lo ripristini prima del ritorno

Vediamo il comportamento di una procedura dal punto di vista dello *stack*, infatti alloca spazio nello *stack*:

- decrementa \$sp per lasciare nello *stack* lo spazio necessario al salvataggio, 1 word per ogni registro da salvare
- salva \$ra
- salva eventuali altri registri usando \$sp come registro base
- ripristina i registri
- incrementa \$sp per riportarlo alla situazione iniziale
- si torna alla procedura *jr \$ra*





Un frame di stack (*chiamata di procedura*) è un blocco di memoria associata alla procedura. \$sp punta alla prima parola del frame, \$fp all’ultima e un frame di solito è un multiplo della parola doppia (8 byte). Per esempio un frame di 32 byte:

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

Vediamo qualche esempio pratico:

vediamo un main che chiama proc1 che a sua volta chiama proc2, stampando i vari punti

```
.data
msg1: .asciiz "esecuzione del Main...\n"
msg2: .asciiz "esecuzione Proc1...\n"
msg3: .asciiz "esecuzione Proc2...\n"
msg4: .asciiz "fine Proc2.\n"
msg5: .asciiz "fine Proc1.\n"
msg6: .asciiz "fine Main.\n"
.globl main

.text
main:
    li $v0, 4          # stampa msg1
    la $a0, msg1
    syscall

    jal proc1         # $ra contiene l'indirizzo dell'etichetta
```

```

rientromain:
    li $v0, 4          # stampa msg6
    la $a0, msg6
    syscall
    li $v0, 10         # syscall exit
    syscall

proc1:
    li $v0, 4          # stampa msg2
    la $a0, msg2
    syscall
    addi $sp, $sp, -4 # faccio spazio sullo stack
    sw $ra, 0($sp)    # pusho $ra e salvo il punto di rientro
                       # del chiamante
    jal proc2

rientro1:
    lw $ra, 0($sp)    # recupero il valore corretto di $ra
    addi $sp, $sp, 4   # eseguo il pop dello stack
    li $v0, 4          # stampa msg5
    la $a0, msg5
    syscall
    jr $ra             # restituisco il controllo al chiamante

proc2:
    li $v0, 4          # stampa msg3
    la $a0, msg3
    syscall

    li $v0, 4          # stampa msg4
    la $a0, msg4
    syscall
    jr $ra             # restituisco il controllo al chiamante

```

Considero un array di 8 word, ogni elemento dell'array rappresenta una cifra binaria (0 o 1) di un numero binario. Implemento una procedura CA2 che lascia invariati tutti gli zeri e il primo 1 partendo dall'ultimo elemento dell'array e inverte tutti gli altri. Implemento una procedura INVERTI che inverte 0 in 1 e viceversa. CA2 chiama INVERTI:

```

.data
numero: .word 0, 1, 0, 1, 1, 0, 0, 0

```

```

.text
.globl main

main:   la $a0, numero
        jal CA2
        li $v0, 17
        syscall

CA2:    li $t5, 8
        addi $t0, $a0, 28

ciclo1: bltz $t5, fine1
        lw $t7, 0($t0)
        addi $t5, $t5, -1
        addi $t0, $t0, -4
        beq $t7, $zero, ciclo1 # se $t7 è 0 si riprende il ciclo1
                                # se no si va al ciclo2
        # se non ci sono più elementi si va alla fine

ciclo2  beqz $t5, fine1
        la $a0, 0($t0)
        addi $sp, $sp, -4
        sw $ra, 0($sp)
        # faccio spazio nello stack
        # pusho $ra

        jal INVERTI

        lw $ra, 0($sp)
        addi $sp, $sp, 4
        # recupero il valore corretto di $ra
        # eseguo il pop dello stack

        addi $t5, $t5, -1
        addi $t0, $t0, -4
        j ciclo2

fine1:  jr $ra
        # restituisco il controllo al chiamante

INVERTI:
        lw $t3, 0($a0)
        bgtz $t3, uno
        addi $t3, $t3, 1
        # carico il parametro passato
        # se il parametro è maggiore uguale
        # di zero vado alla procedura uno
        # se arrivo qua significa che il

```

Capitolo 5. Assembly

```
                                valore è 0, aggiungo 1 per invertire
j fine2
uno:    addi $t3, $t3, -1      # se arrivo qua significa che il
                                valore è 1, tolgo 1 per invertire

fine2:   sw $t3, 0($a0)        # ripristino $t3
          jr $ra                 # restituisco il controllo al chiamante
```

Capitolo 6

Datapath

Si vedrà come costruire un'unità di elaborazione dati (*datapath*) e l'unità di controllo associata a 2 diverse implementazioni hardware dell'insieme di istruzioni MIPS. Si vedrà innanzitutto un'interpretazione che comprende le istruzioni base del MIPS (*lw*, *sw*, *add*, *sub*, *and*, *or*, *slt*, *beq*, *j*). Si escludono molte istruzioni e si esclude l'uso di numeri in virgola mobile. Molti componenti necessari per realizzare queste istruzioni sono i medesimi, per esempio i primi due passi sono sempre identici:

1. inviare il contenuto del program counter (*PC*) alla memoria che contiene il programma e prelevare l'istruzione dalla memoria. Questa operazione è chiamata *fetch* ("raggiungere")
2. leggere il contenuto di uno o due registri utilizzando i campi dell'istruzione per selezionare in registri. Per *lw*, per esempio, serve un solo registro ma la maggior parte delle istruzioni ne richiede 2

dopo questi due passi le azioni richieste variano da istruzione a istruzione, dividendosi però in macrogruppi, dove le azioni sono simili:

- istruzioni di accesso alla memoria (*lw*, *sw*, *etc...*)
- istruzioni aritmetico logiche (*add*, *sub*, *and*, *or*, *slt*, *etc...*)
- istruzioni di salto (*beq*, *j*, *etc...*)

Tranne i salti incondizionati tutte le istruzioni usano l'unità aritmetico.logica (*ALU*) dopo aver letto i registri:

- le istruzioni di accesso alla memoria usano l'ALU per il calcolo dell'indirizzo

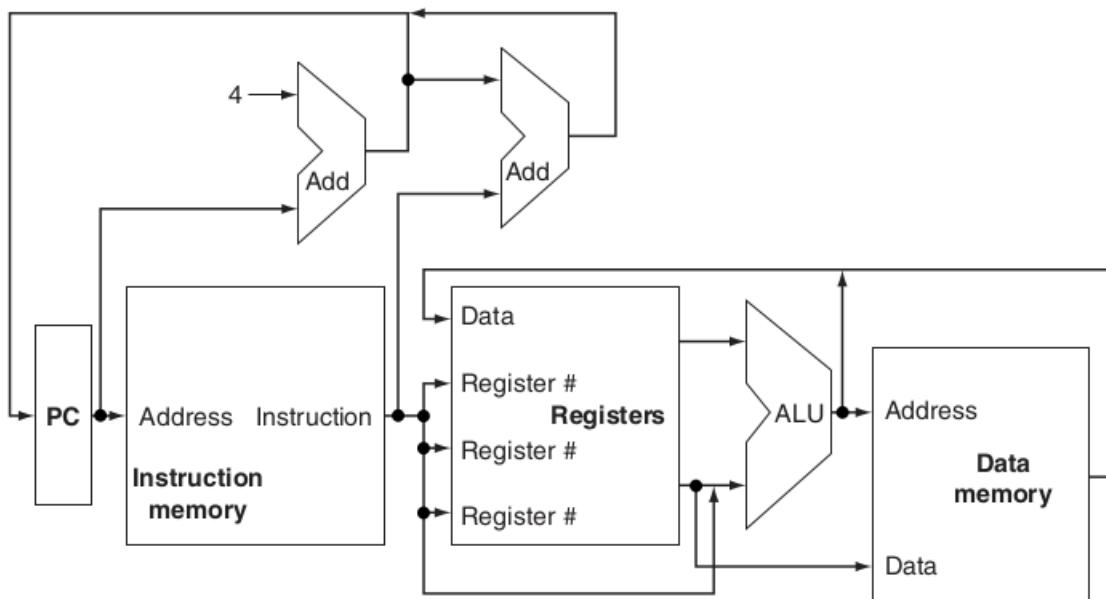
Capitolo 6. Datapath

- le istruzioni aritmetico-logiche usano l'ALU per l'esecuzione dell'operazione
- le istruzioni di salto usano l'ALU per i confronti

Dopo l'uso dell'ALU le azioni variano a seconda delle istruzioni.

- un'istruzione di accesso alla memoria dovrà accedere alla memoria per scrivere i dati (*sw*) o leggerli (*lw*)
- un'istruzione aritmetico-logica dovrà scrivere il risultato prodotto nell'ALU nel registro destinazione
- un'istruzione di salto condizionato l'indirizzo dell'istruzione successiva potrà cambiare in base al confronto, in assenza di salto il program counter verrà incrementato di 4 per puntare all'istruzione successiva

Ecco uno schema ad alto livello di astrazione dell'implementazione di un MIPS, focalizzato sulle varie unità funzionali e sulle loro interconnessioni:

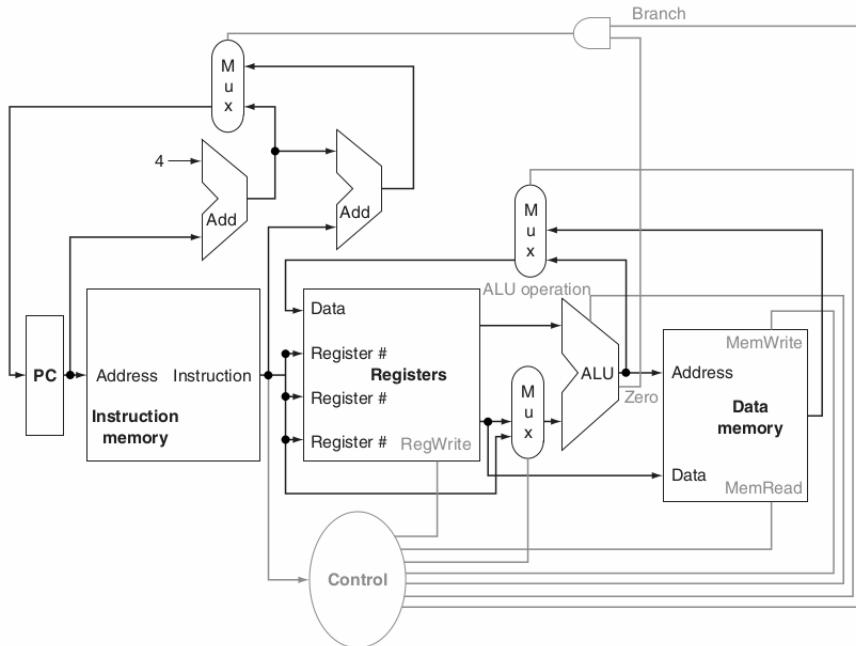


le linee rappresentano i bus (si hanno con dati maggiori di un bit), si vede il sommatore che aumenta il program counter di 4 e si vedono le uscite della ALU a seconda dell'istruzione

Sembra che la maggior parte del flusso dati scorra nel processore ma si omettono due aspetti importanti:

1. in diversi punti della figura si vede che i dati provengono da due sorgenti e arrivano alla stessa unità funzionale. Per esempio il valore scritto nel *PC* può provenire da uno dei due sommatori (*add*), il dato scritto nel register file può provenire sia dalla ALU che dalla memoria, il secondo input della ALU può provenire da un registro o dal campo *immediate* dell'istruzione. Queste linee multiple per i dati non possono essere semplicemente connesse ma bisogna introdurre un circuito logico che effettui una selezione connettendo la sorgente opportuna alla destinazione. Come circuito di selezione si usa solitamente un multiplexor (*MUX*), dove le linee di controllo sono impostate principalmente in base alle informazioni prelevate dall'istruzione stessa che viene eseguita
2. il secondo aspetto omesso è che i segnali di controllo di molte unità funzionali presenti nello schema dipendono dalle istruzioni, per esempio la memoria deve essere letta da una *load* e scritta da uno *store*, il register file deve essere scritto solo nel caso di una *load* o di un'operazione aritmetico-logica. La ALU poi deve poter scegliere l'operazione corretta tra le tante che può effettuare, anche qui la scelta viene effettuata mediante multiplexor.

Ecco l'unità di elaborazione con multiplexor e linee di controllo (in chiaro nell'immagine) per le principali unità funzionali:

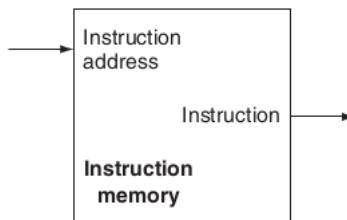


L'unità di controllo (visibile nell'immagine sopra), che ha come ingresso l'istruzione, viene utilizzata per determinare come impostare le linee di controllo per le unità funzionali e per due dei tre multiplexor. Il terzo multiplexor, che determina se nel *PC* vada scritto il valore $PC+4$ o l'indirizzo di destinazione del salto, viene controllato dall'uscita "zero" della ALU, uscita che è impostata a 1 quando l'uguaglianza degli operandi è verificata, e quindi riporta il risultato del confronto richiesto da *beq*. In questa CPU ogni istruzione inizia l'esecuzione su un fronte del segnale di clock e completa l'esecuzione sul fronte di clock del segnale successivo (*metodologia di temporarizzazione sensibile ai fronti*). Il ciclo di clock deve essere allungato per permettere a tutte le istruzioni di essere completate e quindi non si ha un approccio pratico.

6.0.1 Unità di elaborazione

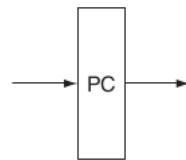
Iniziamo vedendo gli elementi base che servono ad eseguire i diversi tipi di istruzioni MIPS. Iniziamo vedendo quali elementi sono dell'unità di elaborazione (*datapath elements*) sono richiesti da ciascuna classe di istruzioni, procedendo per diversi livelli di astrazione.

- partiamo da un'immagine:



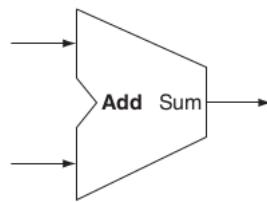
si mostra il primo elemento di cui abbiamo bisogno, un'unità di memoria dove salvare le istruzioni del programma che sia in grado fornire in uscita l'istruzione associata all'indirizzo dato in ingresso.

- si prosegue con:



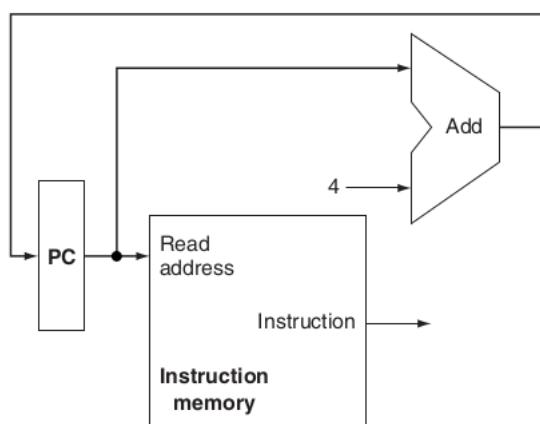
qui si mostra il program counter che è un registro che memorizza l'indirizzo dell'istruzione corrente

- si ha poi:



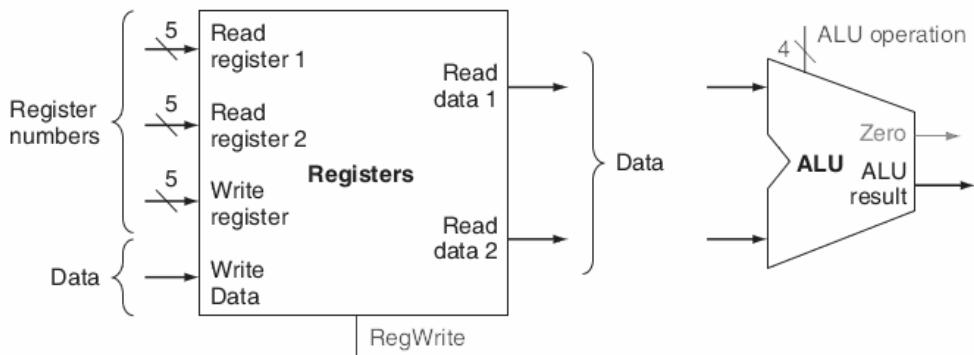
il sommatore, necessario per incrementare di 4 il *PC* e ottenere l'indirizzo dell'istruzione successiva, questo è un elemento combinatorio e può essere costruito a partire dall'ALU; è sufficiente impostare i segnali di controllo dell'ALU in modo che si abbia sempre una somma. Viene etichettato come *add* e può eseguire solo una somma.

Per eseguire qualsiasi istruzione occorre prelevare l'istruzione stessa dalla memoria. per prepararsi all'istruzione successiva si incrementa di 4 byte il *PC*, in modo che punti all'istruzione successiva. Nel complesso si ha il seguente schema:



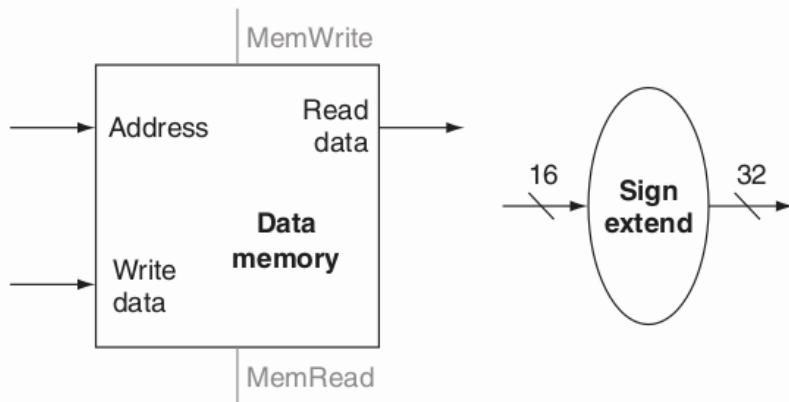
Considero ora le istruzioni R-Type (aritmetico-logiche). Tutte queste istruzioni leggono 2 registri e scrivono il risultato in un registro. I registri universali a 32bit del processore sono raccolti nel register file e si ha bisogno di un'ALU per operare sui valori letti dai registri. Per ogni istruzione si leggeranno due dati di una parola ciascuno dal register file e poi se ne scriverà il risultato. Per ogni parola letta bisogna prevedere un ingresso al register file che specifichi il numero del registro. Per scrivere un dato di una parola serviranno due ingressi, il primo che specifica il numero del registro di scrittura e l'altro deve fornire il dato da scrivere. Il register file fornisce in ogni momento in uscita il valore contenuto nel registro il cui numero d'ordine è specificato sull'ingresso relativo al numero di registro di lettura.

La scrittura viene controllata da un segnale di controllo esplicito, il *RegWrite*, che deve essere *asserted* perché il registro possa essere scritto in corrispondenza del fronte di clock. In totale si hanno quindi 4 ingressi (3 per i numeri dei registri e uno per il dato da scrivere) e due uscite per i dati. Gli ingressi che specificano il numero dei registri hanno ampiezza di 5bit, in modo da poter usare uno dei 32 registri ($2^5 = 32$) mentre i bus del dato in ingresso e di quelli in uscita sono di 32bit. L'ALU in questo caso riceve due ingressi a 32bit e produce un risultato su 32bit e produce un segnale di 1 bit se il risultato dell'operazione è zero. Ecco le immagini di quanto detto:



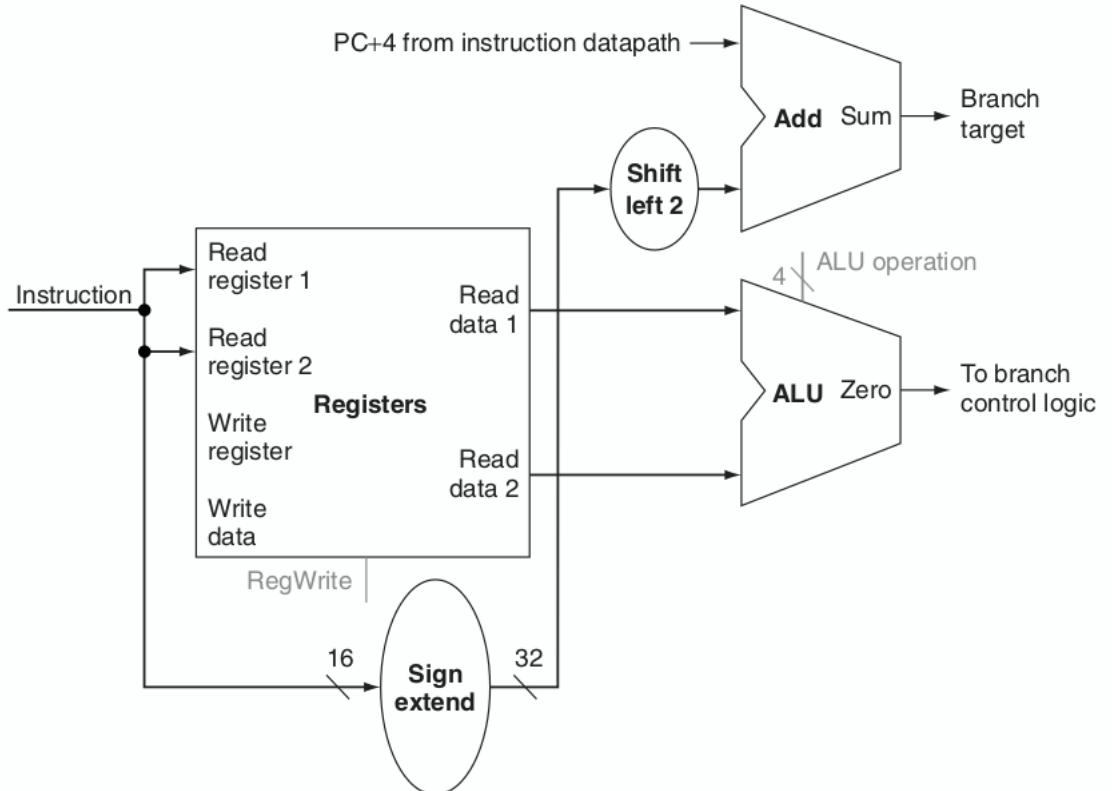
Passiamo alle istruzioni I-Type, in particolare *lw* e *sw* che ricordiamo sono della forma *lw/sw \$t1, valore_offset(\$t2)* e calcolano un indirizzo di memoria sommando il contenuto del registro base (nell'esempio *\$t2*) al campo offset di 16bit (valore immediato) considerato dotato di segno. Se l'istruzione è una store il dato da memorizzare deve essere letto dal register file (dove si trova *\$t1*) mentre se è una load il valore letto dalla memoria deve essere scritto all'interno del register file, nel registro specificato, *\$t1*. Per eseguire queste istruzioni servono sia ALU che register file, come nell'immagine precedente.

Inoltre sono necessarie un'unità per l'estensione del segno del campo offset, contenuto nei 16bit meno significativi, da estendere a 32bit e un'unità di memoria dati da cui prendere o su cui scrivere il dato, essa viene scritta dalle store e avrà segnali di controllo sia per la lettura che per la scrittura. Riceverà in ingresso l'indirizzo e il dato che deve essere scritto. Ecco un'immagine di questi elementi:



Vediamo ora l'istruzione *beq*. Si hanno tre operandi, due registri il cui contenuto viene confrontato per determinare se è uguale e un offset di 16bit utilizzato per calcolare l'indirizzo di destinazione del salto a partire da quello dell'istruzione stessa. La forma ricordiamo è *beq \$t1, \$t2, offset* e per eseguirla bisogna calcolare l'indirizzo di destinazione del salto, sommando il campo offset (esteso a 32bit con segno) al *PC*. Bisogna ricordare che l'architettura dell'insieme specifica che l'indirizzo di base per calcolare il salto è quello dell'istruzione che segue quella dopo il salto, ovvero *PC+4*. Inoltre l'architettura stabilisce che il campo offset sia spostato di 2bit a sinistra cosicché non codifichi lo spiazzamento in byte ma in numero di word, si aumenta quindi lo spazio di indirizzamento di un fattore 4 rispetto alla codifica in byte, per risolvere si fa scorrere l'offset a sinistra di 2bit. Dopo aver calcolato l'indirizzo di destinazione del salto bisogna determinare se l'istruzione da eseguire dopo quella sia presente nella posizione di memoria successiva o in quella alla destinazione del salto. Se gli operandi sono uguali quella del salto è vera e l'indirizzo di destinazione diventa il nuovo indirizzo del *PC* e si parla di *salto condizionato eseguito (branch taken)*, altrimenti si ha il solito *PC+4* e si ha il *salto non eseguito (branch not taken)*. Per i salti condizionati si hanno quindi due operazioni per l'unità di operazione: calcolare la destinazione e confrontare i registri. Inoltre i salti hanno effetto anche sulla parte dedicata al prelevare le istruzioni.

La parte dell'unità di elaborazione che gestisce i salti è quella in figura:



Per il calcolo dell'indirizzo di destinazione si hanno un'unità di estensione del segno e un sommatore, per il confronto si ha il register file che legge due registri con gli operandi ma non è necessario scriverci sopra il risultato. Il confronto viene eseguito dalla ALU inoltre, dato che fornisce un segnale in uscita se il risultato è zero si possono inviare gli operandi alla ALU specificando coi segnali di effettuare una sottrazione: se l'uscita zero viene asserita i due operandi sono uguali.

Si ha infine che l'istruzione del salto incondizionato *jump* opera sostituendo i 28bit meno significativi del *PC* coi 26bit meno significativi dell'istruzione fatti scorrere di 2 bit, che si ottiene concatenando 00 all'offset del salto.

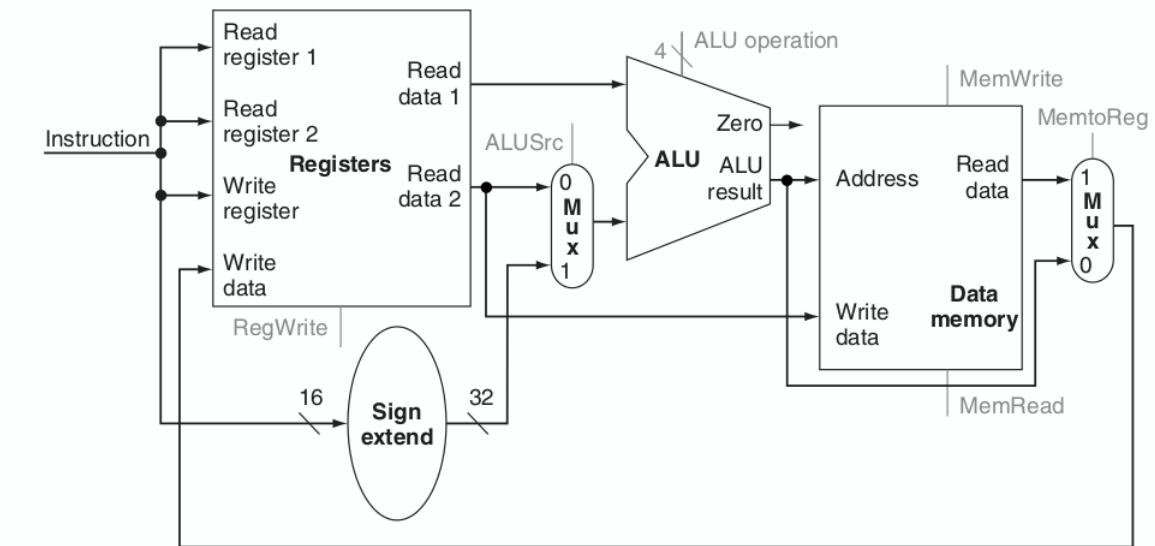
Passiamo ora alla costruzione di un datapath che possa eseguire tutte le istruzioni in un singolo ciclo di clock. Ogni unità può essere usata solo una volta per istruzione e si avrà bisogno quindi di duplicati e servirà una memoria delle istruzioni separata da quella dei dati. Comunque molte unità saranno condivise dal flusso di istruzioni diverse, grazie a collegamenti multipli in

ingresso all'elemento, gestibili con un multiplexor e i suoi segnali di controllo per poter selezionare un input su tanti.

le unità di elaborazione per le R-Type e per le istruzioni di accesso alla memoria sono simili, con solo le seguenti differenze:

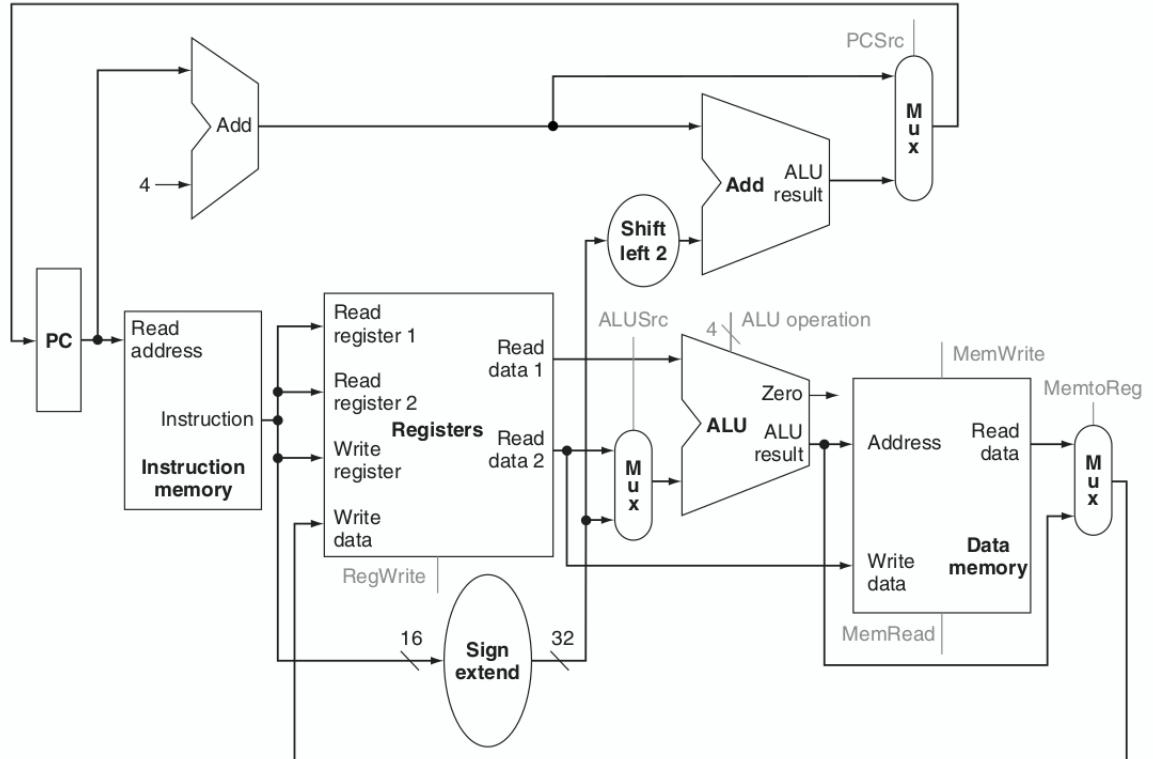
1. le istruzioni logico-matematiche usano la ALU con ingressi da 2 registri. Le istruzioni di memoria usano l'ALU prendono il secondo ingresso dal campo offset dell'istruzione stessa, con estensione di segno
2. il valore da scrivere nel registro proviene dalla ALU per le R-Type o dalla memoria per le load

Quindi per le R-Type e di accesso si ha:



Capitolo 6. Datapath

a cui si aggiunge la parte dedicata ai salti incondizionati ottenendo:



dove si ha l'istruzione di salto condizionato che usa la ALU per il confronto tra gli operandi, contenuti in 2 registri; si ha il sommatore per calcolare l'indirizzo di destinazione del salto e un altro multiplexor sceglie tra questo indirizzo destinazione o il $PC+4$.

Ora che abbiamo lo schema di base possiamo aggiungere l'unità di controllo che dovrà poter accettare dei valori di input e generare un segnale di scrittura per ciascun elemento di stato, un segnale di selezione per ogni multiplexor e i segnali di controllo per l'ALU.

Costruiamo ora un'implementazione che permetta l'esecuzione di *lw*, *sw*, *beq*, *add*, *sub*, *and*, *or*, *slt* (*set less than*) con l'aggiunta finale di *jump* (*j*). Per l'ALU si hanno le seguenti combinazioni dei 4 input di ingresso:

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

(si ricorda che *nor* non è tra le istruzioni MIPS prese in considerazione). Per le R-Type si eseguirà l'istruzione definita dal campo *funct*, i 6 bit meno significativi. Per generare i 4 bit di controllo dell'ALU si può usare una piccola unità di controllo che riceve in ingresso il campo *funct* e un campo di controllo su 2bit, chiamato ALUOp, secondo la tabella:

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

L'uscita dell'unità di controllo dell'ALU è un segnale di 4bit. L'utilizzo di livelli diversi di decodifica (l'unità di controllo principale imposta la ALUOp che a sua volta imposta l'ALU) è una tecnica frequente e si usa per ridurre le dimensioni dell'unità di controllo principale e aumentarne la velocità. Queste sono ottimizzazioni importanti visto che spesso il ciclo di clock è definito sull'unità di controllo. Vediamo come funziona la corrispondenza tra i 2bit del campo ALUOp e i 6bit di *funct* con i 4bit di controllo della ALU. Dato che ci interessa un piccolo sottoinsieme dei 64 possibili valori (ALUOp a 01 e non tutte le *funct*) possiamo ricorrere ad un piccolo circuito logico che riconosca i valori possibili e imposti i bit dell'ALU. Si costruisce quindi la seguente tabella di verità:

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

nella tabella sono riportate solo le combinazioni che ci servono (sarebbero $2^8 = 256$ elementi). Nella tabella il valore *X* indica che non si ha interesse del dato in ingresso infatti si avrà sempre lo stesso risultato, per esempio se l'ALUOp è 01 non si ha alcun interesse nei 6bit del campo *funct*. Da questa

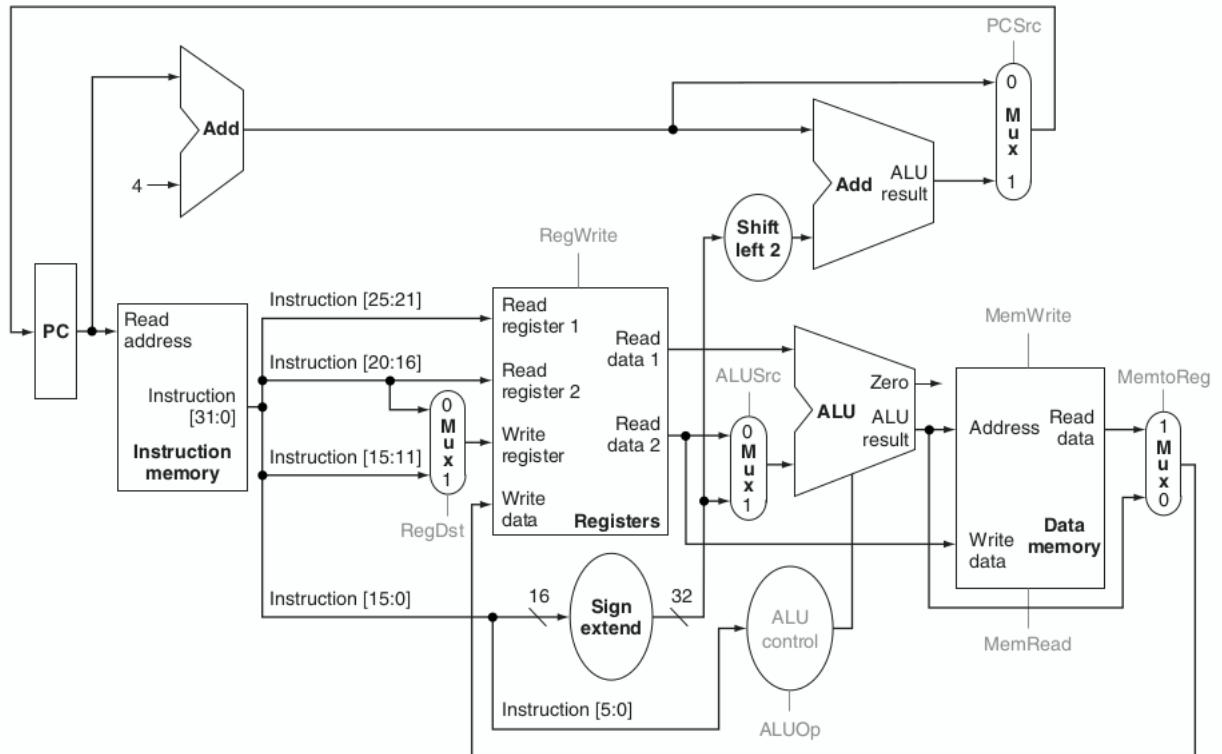
Capitolo 6. Datapath

tabella di potrà poi costruire un circuito logico.

Passiamo ora all'unità di controllo principale. Si ricorda che:

- l'opcode si trova sempre nei bit 31-26 e si indica con *CodOp[5-0]*
 - i 2 registri da leggere nelle R-Type sono nelle posizioni 25-21 e 20-16
 - il registro base per le *lw/sw* è nei bit 25-21
 - l'offset a 16bit delle I-Type è sempre nelle posizioni 15-0
 - il registro destinazione per una load è nelle posizioni 20-16, per una R-Type nelle posizioni 15-11, si necessiterà quindi di un multiplexor per decidere.

Possiamo quindi aggiungere all'implementazione precedente del datapath le etichette sui campi dell'istruzione è un multiplexor per il registro di scrittura del register file, oltre all'unità di controllo dell'ALU spiegata sopra, i segnali di scrittura per gli elementi di stato, il segnale di lettura della memoria dati e i segnali di controllo dei multiplexor, un segnale di controllo ciascuno in quanto hanno solo 2 ingressi, ovvero 7 segnali di controllo. Ecco un'immagine aggiornata dell'implementazione:



Capitolo 6. Datapath

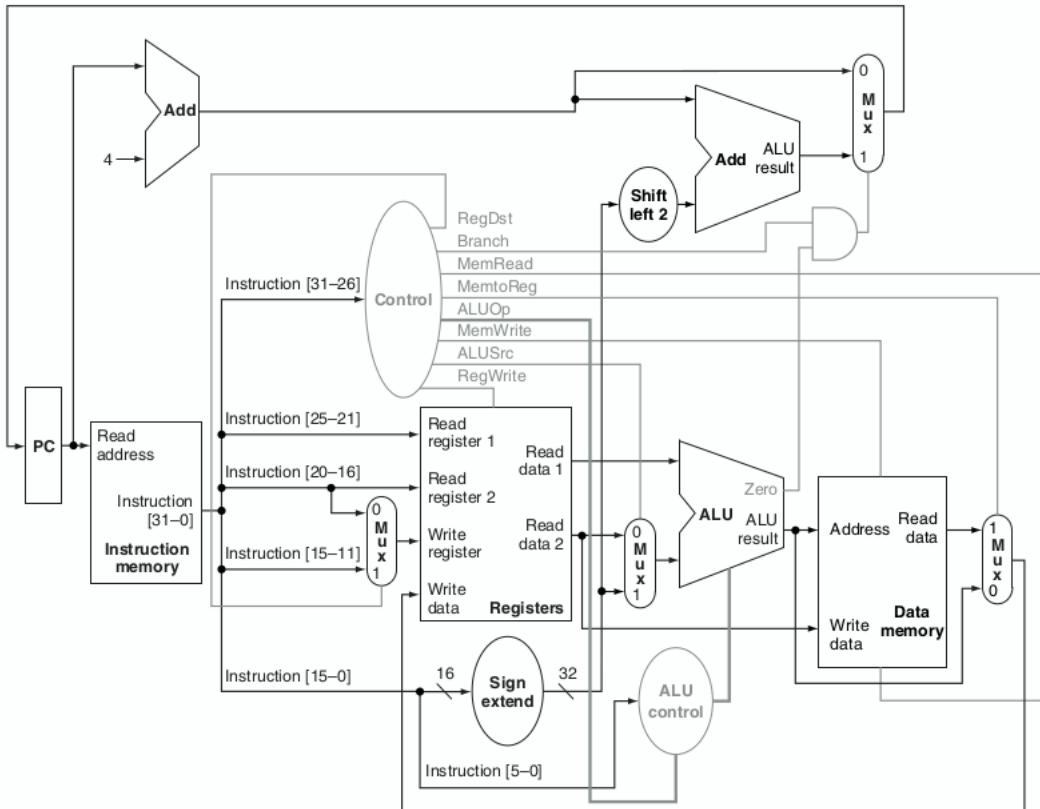
ed ecco una tabella che spiega nel dettaglio i 7 segnali di controllo:

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.

L'unità di controllo li può impostare tutti tranne 1, il segnale di controllo *PCSrc*, basandosi sul codice operativo dell'istruzione stessa. Quel segnale deve essere asserito se si ha una *beq* ma anche se l'uscita zero dell'ALU è vera. SI dovrà quindi collegare in AND un segnale proveniente dall'unità di controllo, detto *branch*, con l'uscita zero dell'ALU. SI hanno quindi in totale 9 segnali di controllo, i 7 in tabella più in due ALUOp, che possono essere impostati sulla base dei sei segnali di ingresso all'unità di controllo, che sono i 6bit dell'opcode.

Capitolo 6. Datapath

Ecco un'immagine completa dell'unità di elaborazione:



Nell'atto pratico si può ottenere questa tabella che identifica il valore dei segnali di controllo durante l'esecuzione delle istruzioni:

Instruction	RegDst	ALUSrc	MemtoReg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUOp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Vediamo ora 3 implementazioni (che avvengono in un ciclo di clock):

1. *add \$t1, \$t2, \$t3*:

- l'istruzione viene prelevata dalla memoria (*fetch*) e si incrementa il *PC*
- i due registri \$t2 e \$t3 vengono letti dal register file mentre l'unità di controllo principale calcola il valore da attribuire alle linee di controllo (*decode*)
- la ALU elabora i dati letti dal register file e il campo *funct* viene utilizzato per selezionare l'operazione (*execute*)
- il risultato è scritto nel register file selezionando il registro \$t1
- si modifica il *PC* e si reitera il ciclo

2. *lw \$t1, offset(\$t2)*:

- l'istruzione viene prelevata dalla memoria (*fetch*) e si incrementa il *PC*
- il registro \$t2 viene letto dal register file mentre l'unità di controllo principale calcola il valore da attribuire alle linee di controllo (*decode*)
- la ALU somma al valore letto dal register file i 16bit meno significativi dell'istruzione, dotati di segno ed estesi a 32bit (*execute*)
- la somma calcolata dalla ALU viene utilizzata come indirizzo per la memoria dati
- il dato proveniente dall'unità di memoria dati viene scritto nel register file nel registro \$t1
- si modifica il *PC* e si reitera il ciclo

3. *beq \$t1, \$t2, offset*:

- l'istruzione viene prelevata dalla memoria (*fetch*) e si incrementa il *PC*
- i due registri \$t1 e \$t2 vengono letti dal register file mentre l'unità di controllo principale calcola il valore da attribuire alle linee di controllo (*decode*)
- la ALU calcola la sottrazione del contenuto dei due registri letti dal register file. Il valore di *PC+4* viene sommato all'offset, (da 16bit esteso a 32bit con segno e spostati di 2 bit a sinistra). Il risultato è la destinazione del salto.

- la linea zero della ALU viene usata per determinare da quale sommatore prendere l'indirizzo successivo da scrivere nel *PC*

Si è così realizzata un'implementazione a singolo ciclo (ancora senza *jump*) della maggior parte delle istruzioni base del MIPS. Si ottiene quindi la seguente tabella di verità con tutte le uscite a seconda del codice operativo d'ingresso:

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Aggiungiamo quindi l'istruzione di *jump*. Questa istruzione assomiglia ad un branch ma non calcola la destinazione, non deve calcolare la destinazione. Si ha che, come per i salti condizionati, i 2bit inferiori dell'istruzione di salto sono 00. I successivi 26 bit dell'indirizzo di salto (da 32bit) provengono dal campo immediato di 26bit dell'istruzione e i 4bit superiori provengono dal *PC* incrementato di 4. Si ha quindi la *jump* scritta nel *PC* con la seguente concatenazione:

- i 4 bit più significativi del valore corrente sul *PC*, incrementato di 4, cioè i bit 31-28 dell'indirizzo dell'istruzione che segue la *jump*
- il campo immediato di 26bit della *jump*
- i bit 00

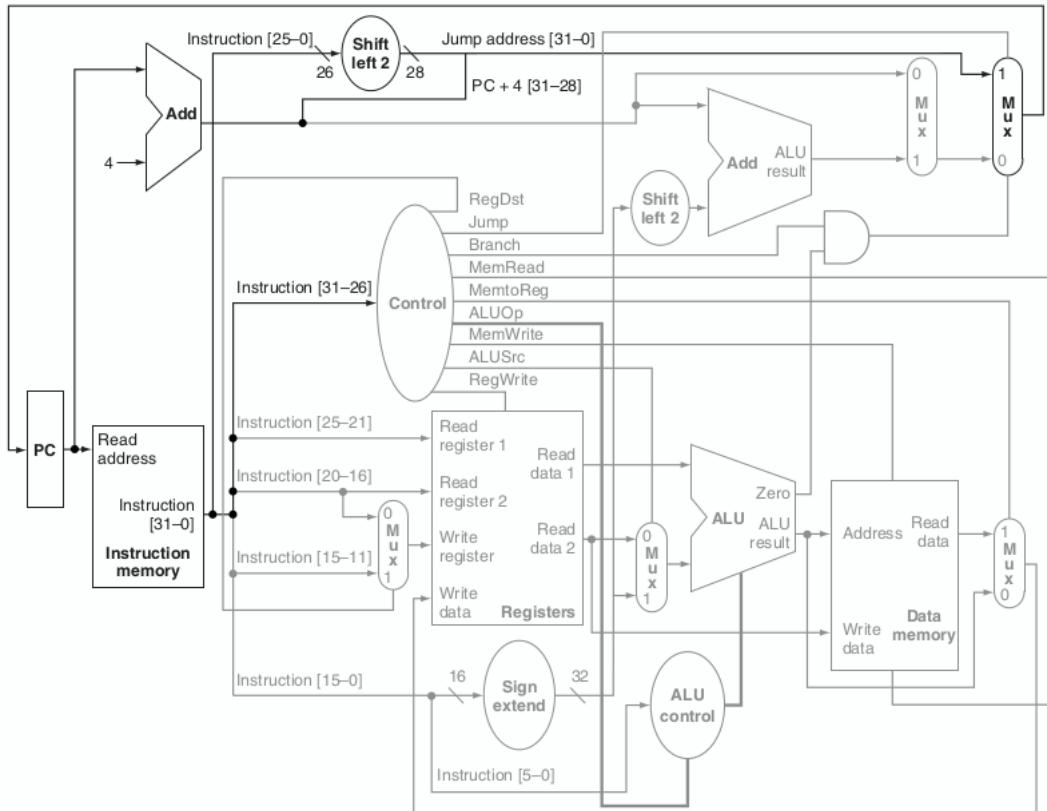
Si hanno dei segnali di controllo in più e si aggiunge un multiplexor per considerare questo possibile nuovo valore del *PC*, che può ora:

- provenire dal *PC* stesso con aggiunta di 4

Capitolo 6. Datapath

- provenire dall'indirizzo di destinazione di un branch
- provenire dall'indirizzo di destinazione di una jump

e serve un segnale di controllo per questo multiplexor. Questo segnale è chiamato *Jump* ed è asserito solo se si ha una jump, ovvero quando il codice operativo è 2. Ecco l'intera unità di elaborazione a singolo ciclo:



Ovviamente una implementazione singolo ciclo è inefficiente. Il periodo di clock deve avere la durata di tutte le operazioni e quindi è determinato sul cammino più lungo (*cammino critico*), solitamente quello della *load* che utilizza 5 unità funzionali.

6.0.2 Performance singolo ciclo

Esempio 10. Siano:

- *memory unit: 200 picosecond(ps)*
- *ALU e sommatori: 100ps*
- *register file (lettura e scrittura): 50ps*

si assuma che i multiplexor, l'unità di controllo, l'accesso al PC, l'unità del segno esteso e i collegamenti non abbiano delay. Valuto le performances:
Si hanno le seguenti percentuali di istruzioni: 25% load, 10% store, 45% istruzioni ALU, 15% branch e 5% jump.

$$CPU_execution_time = Instruction_count \times CPI \times Clock_{cycle_time}$$

CPI=cicli per istruzione

se assumiamo che ogni istruzione venga eseguita in un solo ciclo di clock si ha:

$$CPU_execution_time = Instruction_count \times 1 \times Clock_{cycle_time}$$

Aiutiamoci con due tabelle:

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

Capitolo 6. Datapath

Se si usa un singolo ciclo di clock per ogni istruzione si avrà il ciclo di clock determinato dall'istruzione più lunga (lw con 600ps). Se invece si usano più cicli di clock si avrà:

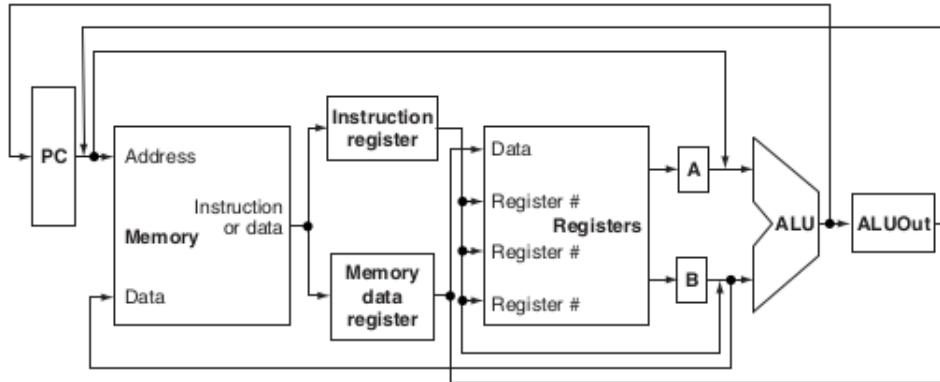
$$CPU_ClockCycle = 600 \times 25\% + 550 \times 10\% + 400 \times 45\% + 350 \times 15\% + 200 \times 5\% = 447,5ps$$

Inoltre:

$$\frac{Performance_single}{Performance_variable} = \frac{600}{447,5} = 1,34$$

6.0.3 Implementazione Multiciclo

In un'implementazione multiciclo ogni passo dell'esecuzione dell'istruzione richiede un ciclo di clock e permette ad ogni unità di essere usata più volte per istruzione, riducendo l'hardware necessario. Ecco un'astrazione dell'implementazione multiciclo:



dove si possono notare una singola unità di memoria, sia per istruzioni che dati, una singola ALU (anziché una ALU e due sommatori) e sono aggiunti uno o più registro dopo ogni unità per immagazzinare l'output di quella istruzione fino a che non debba essere utilizzata in un ciclo successivo. Dopo un ciclo di clock ogni informazione deve essere salvata in un elemento di stato. I dati usati da istruzioni seguenti devono essere salvati in un elemento di stato visibile al programmatore (register file, *PC* o memoria). Invece i dati usati dalla stessa istruzione in un ciclo successivo devono essere salvate in questi registri aggiuntivi. Si assume che ogni ciclo sia in grado di eseguire almeno un accesso alla memoria o uno al register file o un'operazione nella ALU. Si aggiungono i seguenti registri:

- *IR*, Instruction Register, e *MDR*, Memory Data Register, aggiunti per salvare rispettivamente l'output della memoria per un'istruzione letta e per un dato letto. Sono due registri diversi entrambi necessari durante lo stesso ciclo di clock
- i registri A e B usati per immagazzinare i valori operandi dei registri letti dal register file
- l'*ALUOut* per immagazzinare l'output dell'ALU

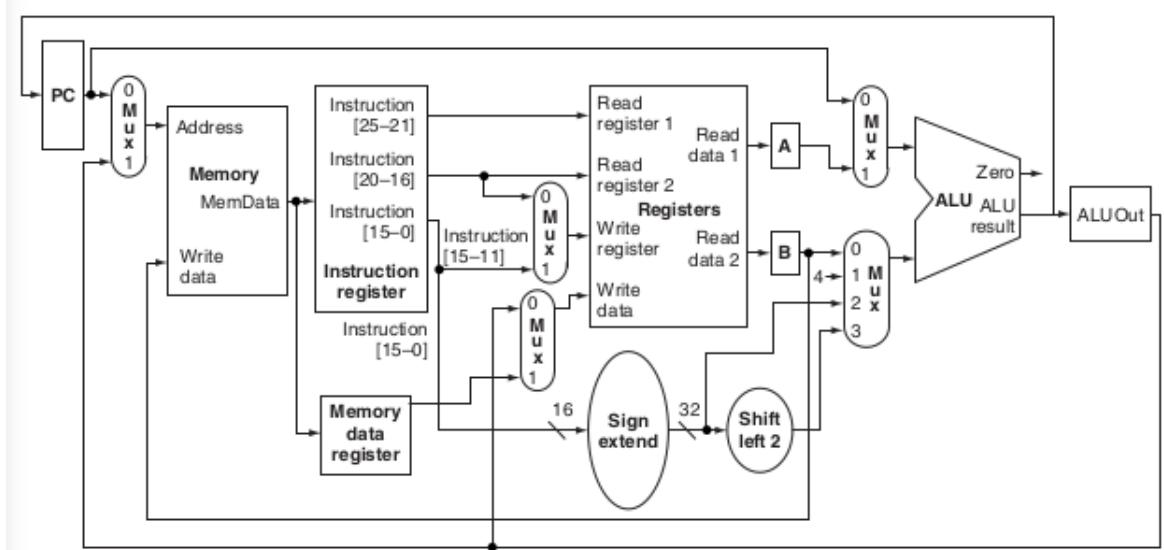
L'*IR* immagazzina l'istruzione fino alla fine dell'esecuzione della stessa e si richeide così un segnale di scrittura, non necessario per gli altri registri che

immagazzinano dati solo nel tempo intercorrente due cicli di clock. Si necessiterà inoltre di aggiungere multiplexor e ingrandire quelli esistenti a causa del fatto che molte unità sono usate per più scopi, per esempio ne servirà uno per decidere la sorgente per un indirizzo di memoria, tra il *PC* (per le istruzioni, e l'*ALUOut* (per i dati).

Si ha ora una singola ALU che svolge i lavori delle 3 ALU dell'implementazione a singolo ciclo. Per permettere questo ci servono altri due cambiamenti al datapath:

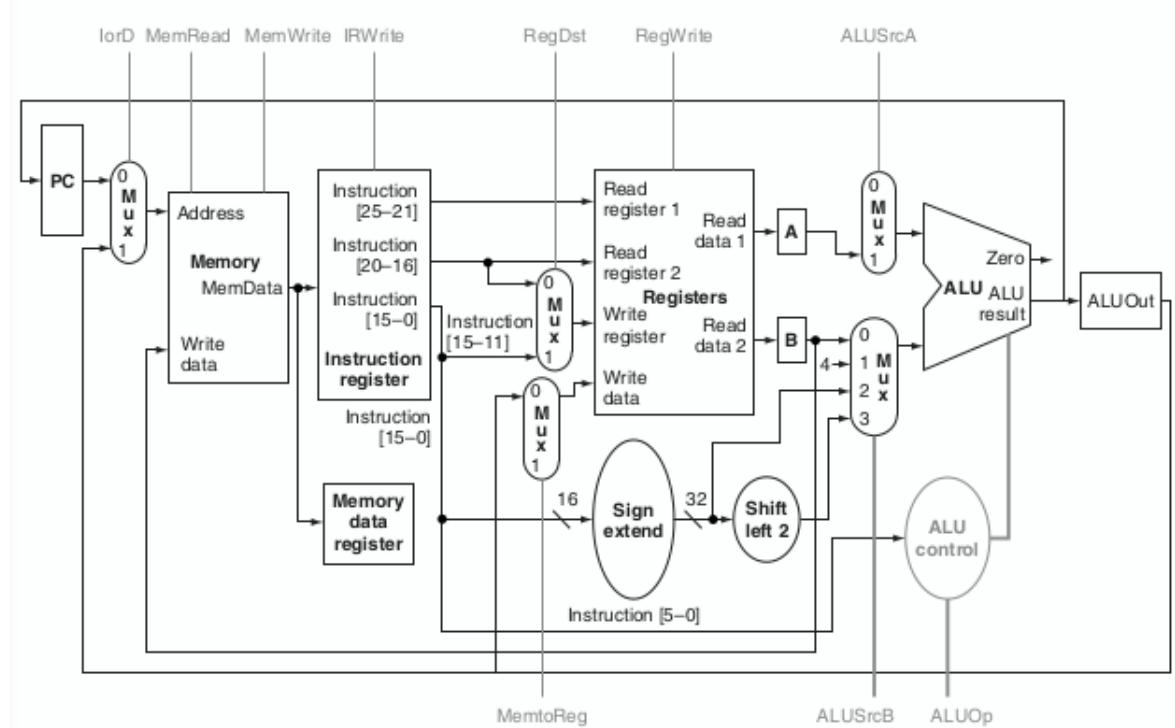
1. un multiplexor è aggiunto al primo input della ALU e sceglie tra il registro A il *PC*
2. il multiplexor del secondo input della ALU passa dall'avere 2 ingressi all'averne 4, i 2 input aggiuntivi sono la costante 4, per il *PC* l'offset (esteso e shiftato) per i branch

Si ottiene quindi il seguente datapath, con i multiplexor sistemati, con le unità di memoria ridotte e senza i due sommatori:



Capitolo 6. Datapath

Dato che si hanno più cicli di clock per istruzione si rendono necessari diversi segnali di controllo. Il *PC*, la memoria e i registri (anche l'*IR*) avranno bisogno di scrivere segnali di controllo (la memoria anche di leggerli). L'unità di controllo della ALU fatta per il singolo ciclo può essere usata anche nell'implementazione multiciclo, ogni multiplexor a 2 input avrà una singola linea di controllo mentre quello a 4 ne avrà 2. Si ottiene quindi:



Si dovrà aggiungere altro per permettere il funzionamento di branch e jump infatti con essi si hanno 3 possibili sorgenti per il valore da scrivere nel *PC*:

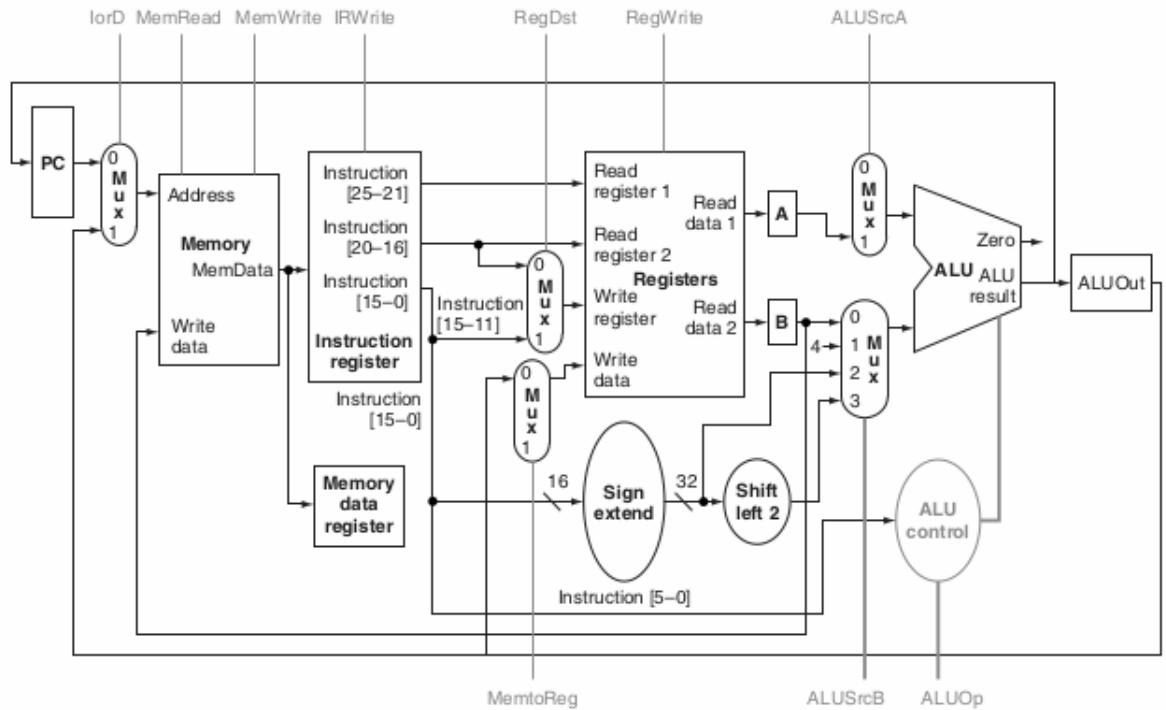
1. l'output della ALU, ovvero il $PC+4$ durante il fetch. Questo valore dovrebbe essere immagazzinato direttamente dal PC
 2. il registro ALUOut, ovvero dove saranno salvati gli indirizzi destinazione dei branch
 3. i 26bit meno significativi dell' IR shifati di 2 e concatenati con i 4bit più significativi del PC incrementato, che sono la sorgente nel caso di una jump

Se l'istruzione è un branch condizionato il *PC* incrementato è sostituito dal valore in *ALUOut* solo se i due registri designati sono uguali. La nostra implementazione ha quindi due segnali di controllo:

Capitolo 6. Datapath

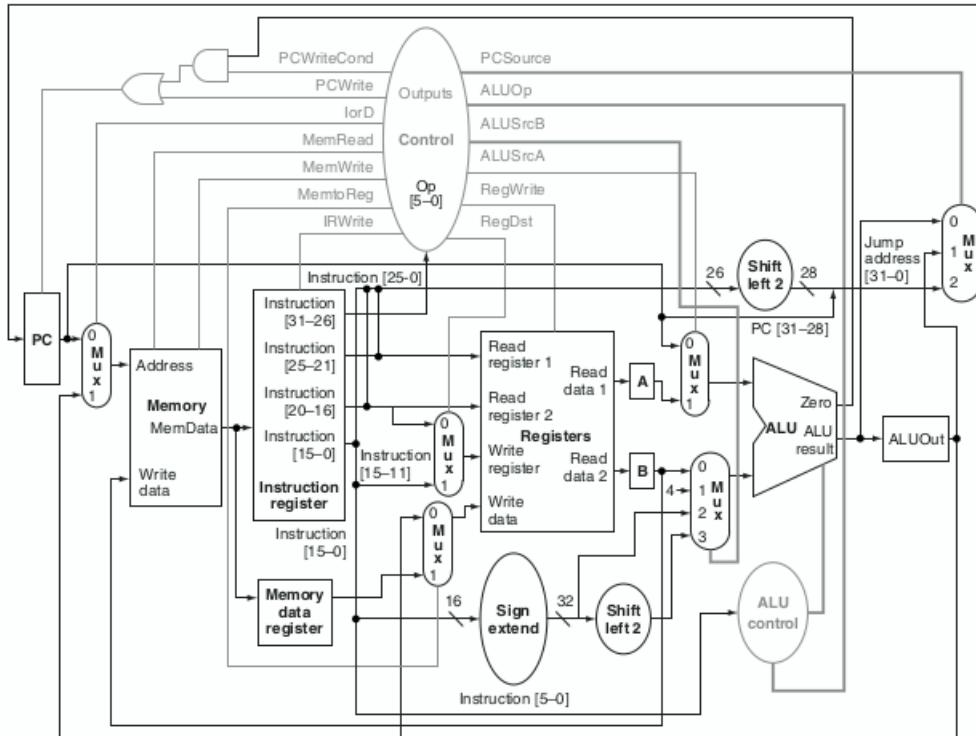
1. PCWrite che comporta una scrittura incondizionata del *PC*
2. PCWriteCond, che comporta la scrittura del *PC* se anche la condizione del branch è vera.

Si necessita di collegare questi due segnali di controllo a quello di scrittura del *PC*, si useranno poche porte per scegliere tra il segnale di PCWrite, PCWriteCond e il segnale zero della ALU. Per determinare se il *PC* deve essere scritto durante un branch mettiamo un AND tra il segnale zero e il PCWriteCond e l'output di questo AND viene messo in OR con il PCWrite e l'output è collegato con il segnale di scrittura del *PC*. Possiamo ottenere quindi un datapath multiciclo completo:



Capitolo 6. Datapath

e se aggiungiamo le linee di controllo otteniamo:



Capitolo 6. Datapath

e vediamo gli effetti dei vari segnali:

Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rd field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSrc.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Actions of the 2-bit control signals

Signal name	Value (binary)	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs a subtract operation.
	10	The funct field of the instruction determines the ALU operation.
ALUSrcB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shifted left 2 bits.
PCSrc	00	Output of the ALU ($PC + 4$) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC for writing.
	10	The jump target address ($IR[25:0]$ shifted left 2 bits and concatenated with $PC + 4[31:28]$) is sent to the PC for writing.

vediamo ora gli step principali (da 3 a 5) che compongono l'esecuzione di un'istruzione nell'implementazione multiciclo:

1. *fetch step*: si preleva l'istruzione dalla memoria e si elabora l'indirizzo dell'esecuzione successiva. Si manda il *PC* alla memoria (in lettura) come indirizzo e si scrive sull'*IR*, dove viene memorizzato e si incrementa il *PC* di 4. A livello pratico si asserta il MemRead e il IRWrite, si setta l'IorD a 0 e si seleziona il *PC* come sorgente (ALUSrcA settato a 0 e ALUSrcB a 01 e ALUOp a 00). Per scrivere l'istruzione successiva serve il *PC* a 00 il PCWrite settato

2. *decode and fetch step*: si leggono i registri rs e rt e si salvano in A e B. Si elabora anche l'indirizzo destinazione di un eventuale branch che viene salvato in ALUOut. Queste due operazioni potrebbero non essere necessarie ma riducono il numero di cicli necessari. Si ha quindi l'accesso al register file per leggere rs e rt e salvarli in A e B (azione da compiere ad ogni ciclo dato che A e B vengono sovrascritti). Si calcola la destinazione del branch e si salva nell'ALUOut dove sarà usata nel ciclo successivo (e si setta ALUSrcA a 0 cosicché il PC sia mandato alla ALU e l'ALUSrcB a 11, cosicché l'offset esteso e shifato sia mandato alla ALU e l'ALUOp a 00 così da permettere la somma). L'accesso al register file e il calcolo della destinazione del branch avvengono in parallelo
3. *execution, memory address computation, or branch completion*: è la prima parte definita dal tipo di istruzione e si ha l'ALU che opera di conseguenza:
 - *istruzioni per la memoria (lw, sw)*: l'ALU forma l'indirizzo di memoria. Serve l'ALUSrcA a 1 (così il primo insput sarà il registro A) e l'ALUSrcB a 10 (così si userà l'ououtput dell'unità del segno esteso come input). L'ALUOp sarà a 00 per sommare
 - *R-Type*: l'ALU esegue l'operazione richiesta sui due registri letti precedentemente dal register file (con ALUSrcA=0 e ALUSrcB=00, che permettono l'uso di A e B come input). L'ALUOp è a 10 per permettere l'uso del funct per decidere il segnale di controllo della ALU
 - *branch*: l'ALU verifica l'uguaglianza dei registri letti nel ciclo precedente (con ALUSrcA=0 e ALUSrcB=00, che permettono l'uso di A e B come input). L'ALUOp è a 01 per permettere alla alu di eseguire la sottrazione e il PCWriteCond sarà asserito per aggiornare il PC in caso di zero come output della ALU. Il PCSource sarà a 01 per permettere la scrittura nel PC dall'ALUOut con l'indirizzo del branch calcolato nel ciclo precedente. Per i branch condizionati si scrive due volte il PC, una volta con l'output della ALU (dal processo di decode) e l'altra con il valore dell'ALUOut (calcolato per il branch). L'ultimo dei due valori scritti sarà quello usato.
 - *jump*: il PC viene sovrascritto dall'indirizzo della jump. Il PC-Source è settato per indirizzare il valore dell'indirizzo della jump al PC e il PCWrite è asserito per scrivere l'indirizzo nel PC

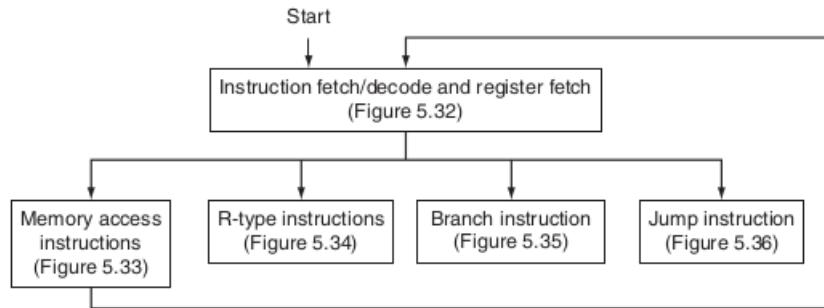
4. *Accesso alla memoria o completamento delle R-Type:* qui le istruzioni di load e store accedono alla memoria e le R-Type scrivono il risultato. Quando un valore è recuperato dalla memoria viene salvato nel MDR (*memory data register*) dove sarà usato nel prossimo ciclo. Se si ha una lod si prende il dato e si scrive nel MDR (scritto ad ogni ciclo, quindis enza segnali di controllo). Se si ha una store il dato viene scritto in memoria. Negli altri casi l'indirizzo è quello salvato nel ciclo precedente nell'ALUOut. Per la store si salva in B (che viene letto due volte, nel secondo e terzo step, ma fortunatamente il numero di registro non cambia). Il MemRead per la load e il MemWrite per la store sono asseriti e il IorD è a 1 per forzare l'arrivo dell'indirizzo di memoria dalla ALU e non dal *PC*. Per le R-Type si scrive il contenuto dell'ALUOut nel register file. Il RegDst è a 1 per forzare l'indirizzo rd come indirizzo dove scrivere e il RegWrite deve essere asserito e il MemtoReg a 0 per far sì che l'output della ALU sia scritto e non l'output della memoria.
5. *conclusione della lettura di memoria:* le load finiscono il loro processo scrivendo il dato letto, che si trova in MDR. Si necessita di MemtoReg=1 per scrivere il risultato dalla memoria, il RegWrite asserito per poter scrivere e RegDst=0 per scegliere rt come numero di registro

Macchina a stato finito

La macchina a stato finito è il primo metodo per definire il controllo multyciclo ed è un'insieme di stati e direzioni su come cambiano gli stati. Le direzioni sono definite dalle *next-state function* che indirizzano lo stato corrente e gli input verso un nuovo stato. Ogni stato definisce degli output che sono asseriti quando la macchina è in quello stato. I segnali che non sono asseriti sono deasseriti, non esiste il *don't care*. Nella macchina a stato finito si specificano sempre le implementazioni di tutti i multiplexor necessari. Ogni stato rappresenta un ciclo di clock e rappresentano i 5 step sopra (i primi 2 identici per ogni istruzione mentre le altre dipendono dall'opcode).

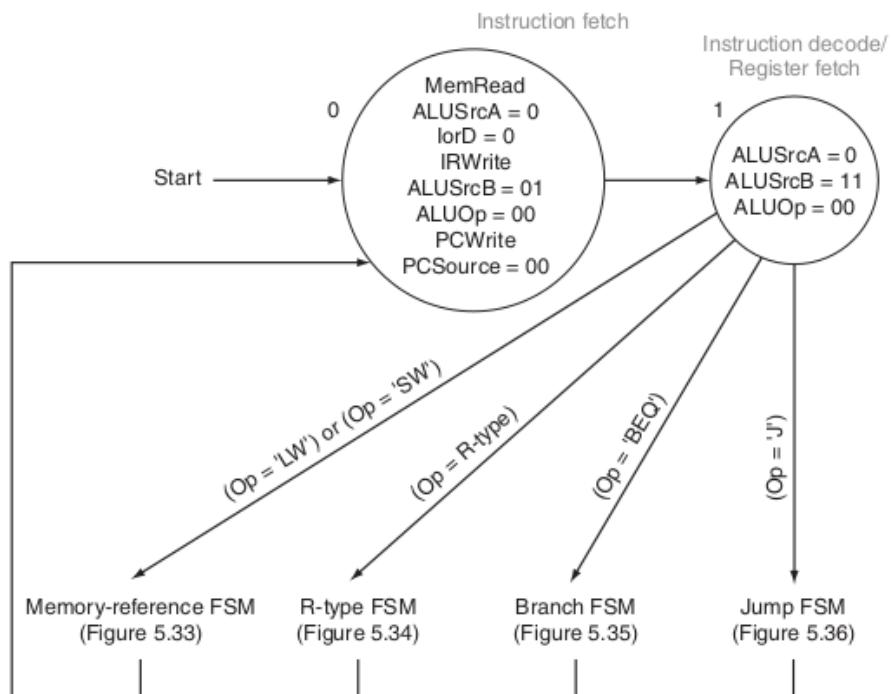
Capitolo 6. Datapath

Dopo l'ultimo step si ritorna all'inizio per un nuovo ciclo di clock, come in figura:

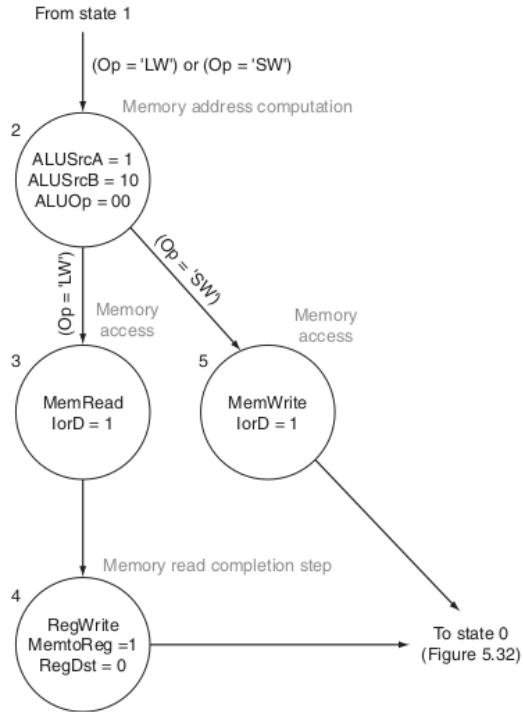


andremo ora ad analizzare i 5 step, ricordando che lo step 1 sarà lo state 0 e così via. I segnali asseriti saranno rappresentati con un cerchio, gli archi tra gli stati definiscono lo stato successivo e sono etichettati con le varie condizioni che selezionano un possibile stato successivo. Dopo lo stato 1 i segnali dipendono dalla classe dell'istruzione e la scelta viene effettuata nel processo di *decoding*:

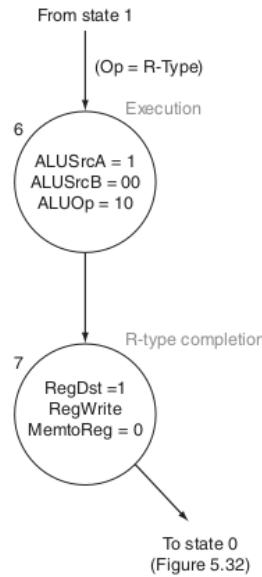
- *state 0 and state 1*: uguali per tutti



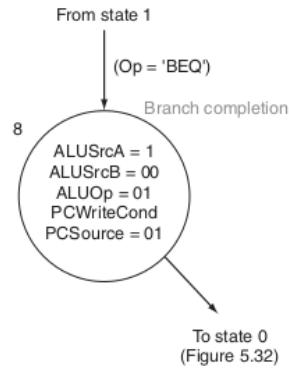
- *finite state machine for memory references (state 2- state 5)*: ecco i successivi stati per lw e sw:



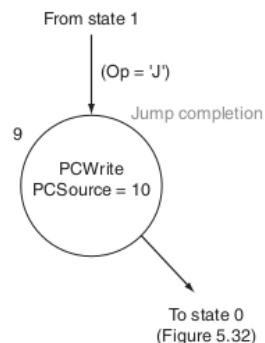
- *finite state machine for R-Type instructions (state 6- state 7)*: ecco i successivi stati per R-Type:



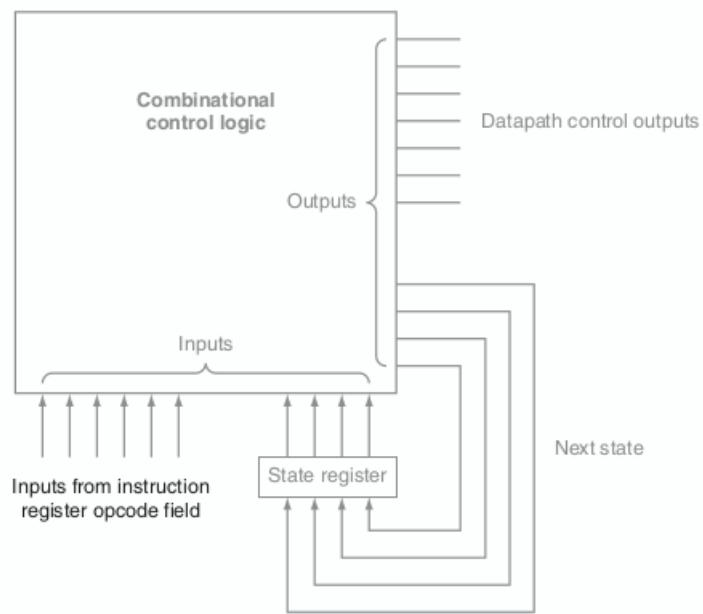
- *finite state machine for branch (state 8)*: ecco i successivi stati per i branch:



- *finite state machine for jump (state 9)*: ecco i successivi stati per i jump:

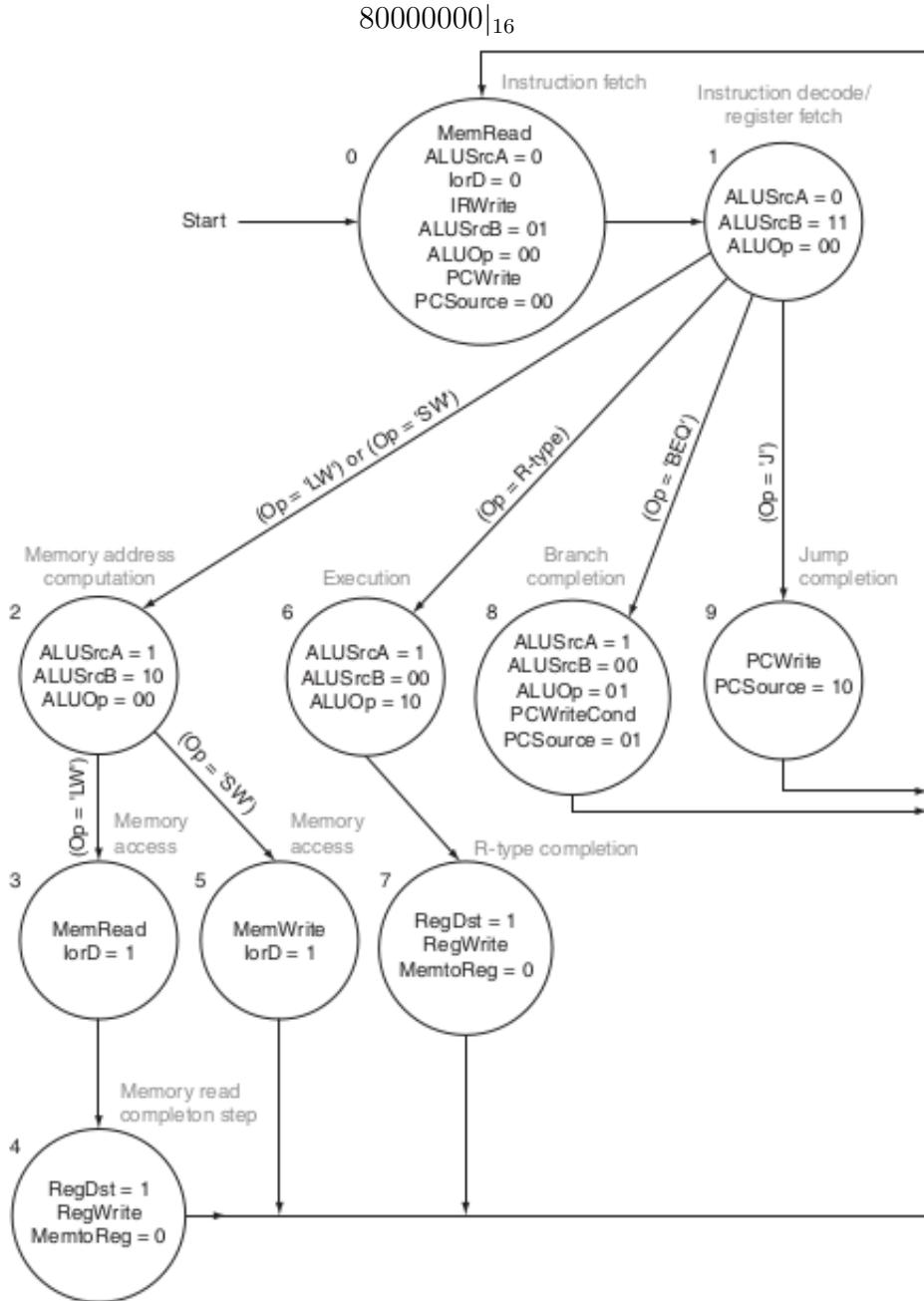


Una macchina a stato finito si può implementare con registri temporanei che contengono lo stato attuale e con un blocco di combinazioni logiche che permettono di determinare sia i degnali del datapath da asserire che lo stato seguente, in questa maniera (detta *Moore Machine*), *dove gli output dipendono unicamente dallo stato corrente. Si divide in 2, un blocco con il control output e solo l'input dello stato e un altro con solo l'output col prossimo stato:*



Capitolo 6. Datapath

Ed ecco la nostra macchina a stato finito completa:



6.0.4 CPI

Abbiamo il seguente mix di istruzioni: 25% loads (1% load byte + 24% load word) con 5 cicli, 10% stores (1% store byte + 9% store word) con 4 cicli,

Capitolo 6. Datapath

11% branches (6% beq , 5% bne) con 3 cicli, 2% jumps (1% jal + 1% jr) con 3 cicli, e 52% ALU con 4 cicli.

$$\begin{aligned} CPI &= \frac{CPU_Clock_cycles}{Instruction_{Count}} = \frac{\sum Instruction_{Count_i} \times CPI_i}{Instruction_{Count}} \\ &= \sum \frac{Instruction_{Count_i}}{Instruction_{Count}} \times CPI_i \end{aligned}$$

con $\frac{Instruction_{Count_i}}{Instruction_{Count}}$ che è semplicemente la frequenza dell'istruzione per ogni classe, quindi (esprimendo le percentuali in decimali):

$$CPI = 0,25 \times 5 + 0,10 \times 5 + 0,52 \times 4 + 0,11 \times 3 + 0,02 \times 3 = 4,12$$

Capitolo 7

Eccezioni

L'unità di controllo di controllo è la parte più difficile del processore da far funzionare correttamente e una delle cose più difficili sono la gestione delle eccezioni (origine interna) e gli interrupt (origine esterna), che sono eventi distinti dai salti che alterano il flusso sequenziale di esecuzione delle istruzioni. All'inizio sono state create per gestire eventi inaspettati come l'overflow aritmetico e poi lo stesso meccanismo è stato usato per gestire la comunicazione tra dispositivi di I/O e processore. Ecco una piccola tabella d'esempio:

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

Aggiungere a posteriori l'implementazione per le eccezioni può ridurre drasticamente le performance e rendere l'implementazione non funzionante. Due tipi di eccezioni possono essere l'esecuzione di un'istruzione non valida e l'overflow aritmetico. L'operazione fondamentale è salvare l'indirizzo dell'istruzione che genera l'istruzione nell'*EPC* (*exception program counter*) e trasferire il controllo ad un indirizzo preciso del sistema operativo, che a sua volta potrà intraprendere le giuste azioni, come fornire servizi al programma utente, svolger determinate azioni in caso dell'eccezione stabilita o determinare l'esecuzione segnalando un errore. A quel punto il sistema operativo può scegliere di interrompere l'esecuzione o di ripartire utilizzando l'*EPC* per determinare il punto da cui partire. Il sistema operativo, per poter gestire al

meglio l'eccezione, deve saperne la causa e deve sapere quale istruzione l'ha generata. Ci sono due metodi per comunicare la causa di un'eccezione:

1. il metodo usato nell'architettura del MIPS, dove si prevede un registro dedicato (detto *cause register*) contenente un campo che indica la causa
2. usare dei *interrupt vettorizzati* dove l'indirizzo a cui si deve trasferire il controllo viene determinato dalla causa dell'eccezione stessa (per esempio $80000000|_{16}$ per l'istruzione non definita e $80000180|_{16}$ per l'overflow aritmetico). Da questi registri il sistema operativo capisce la causa dell'eccezione. Questi registri sono separati da 32byte (l'equivalente di 8 istruzioni) con i quali il sistema operativo registra il motivo ed esegue alcune semplici elaborazioni. Se l'eccezione non è vettorizzata il sistema operativo deve comunque decodificare il registro causa.

Tutto questo può essere implementato aggiungendo alcuni registri e alcuni segnali di controllo ed estendendo l'unità di controllo. Come esempio implementiamo l'unico registro di risposta impostato a $80000180|_{16}$. Si aggiungono quindi:

1. *EPC*: un registro a 32bit che memorizza l'indirizzo dell'istruzione che genera l'eccezione. Serve anche nel caso di eccezioni vettorizzate.
2. *cause register*: registro che salva la causa. Nel MIPS è a 32bit sebbene alcuni bit non vengano utilizzati:

Quindi nel dettaglio si ha:

- *istruzione indefinita*: dopo lo stato 1 non si ha alcuna istruzione, il campo op non identifica alcun opcode e si avrà lo stato 10 (eccezione per istruzione indefinita)
- *overflow aritmetico*: l'ALU riesce autonomamente ad identificarlo e si ha un segnale chiamato *overflow* come output che porta allo stato 11 (eccezione per overflow aritmetico).

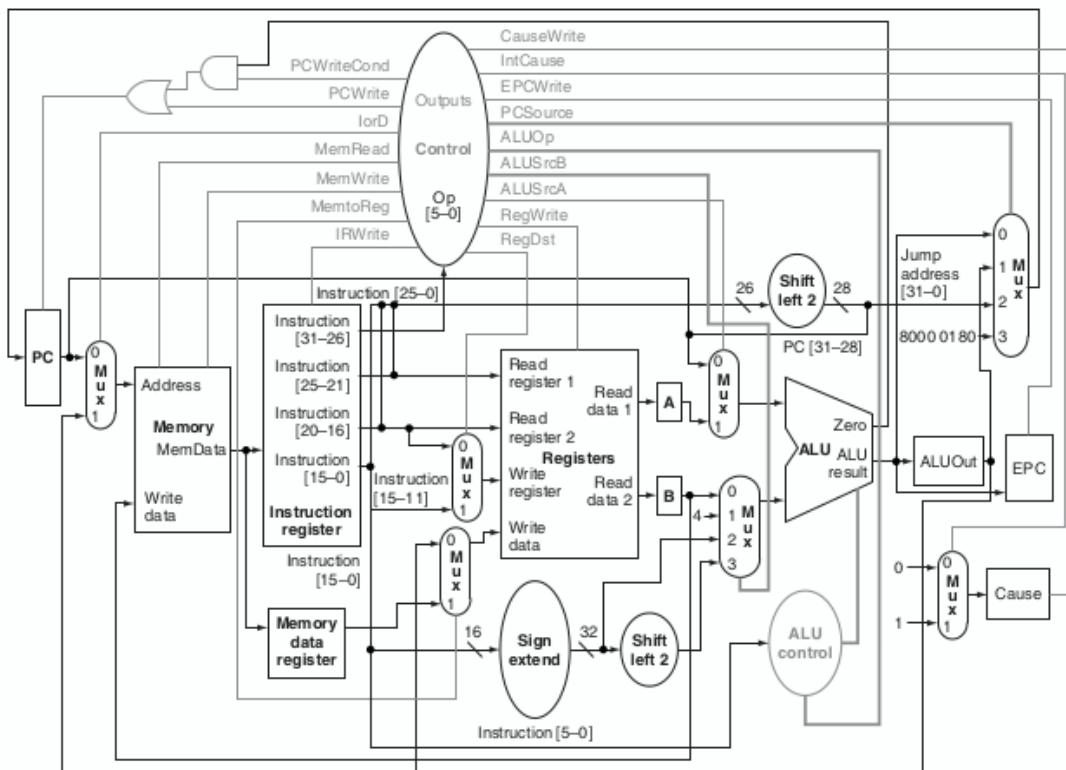
Si hanno anche 2 nuovi segnali di controllo:

1. *EPCWrite*
2. *CausaWrite*

inoltre servirà un segnale aggiuntivo (*PCsrc*) di 1bit per fornire il valore corretto al bit meno significativo del cause register, detto *CausaInt*. Infine si dovrà poter scrivere nel *PC* l'indirizzo iniziale della procedura di gestione

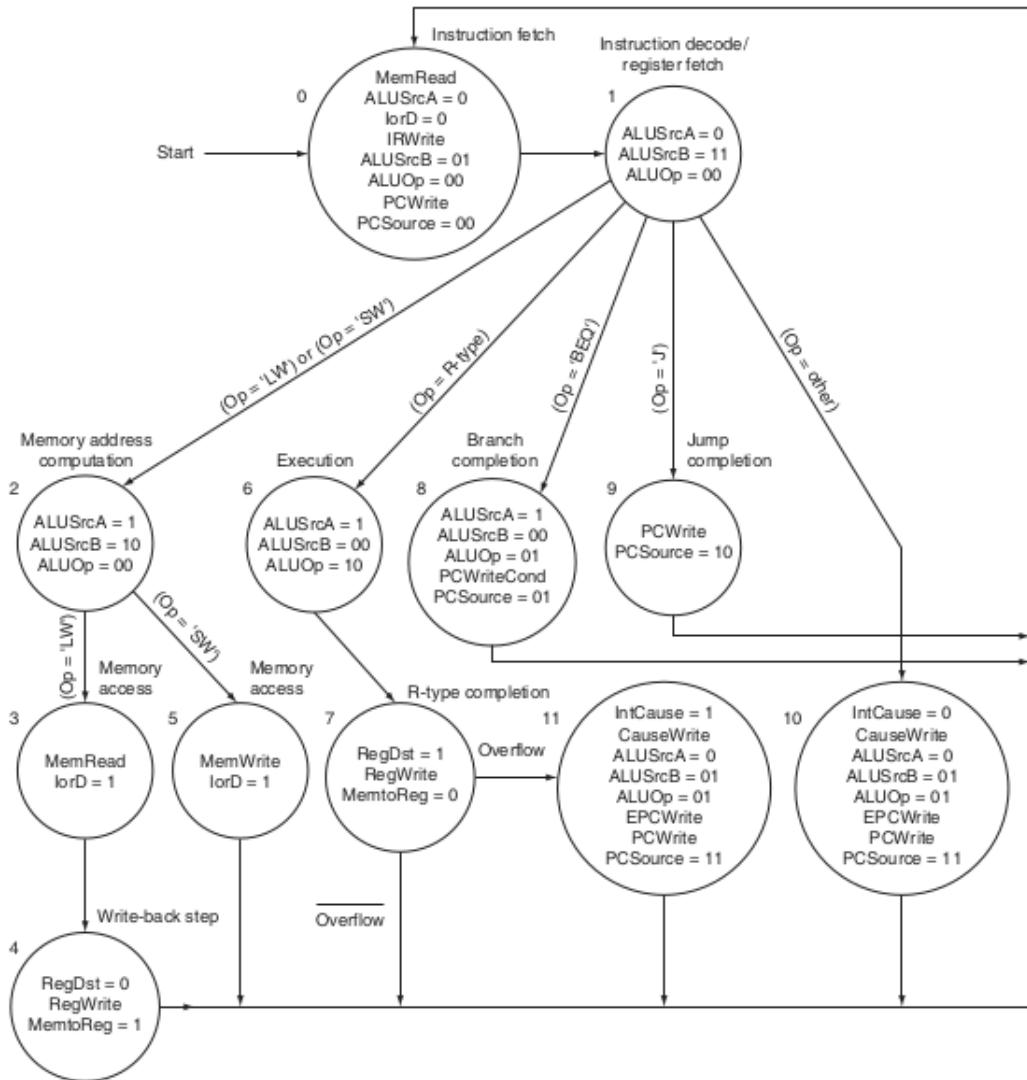
Capitolo 7. Eccezioni

delle eccezioni, esempio $C0000000|_{16}$. Si trasforma il multiplexor anteposto al PC in uno a 4 uscite con l'ingresso aggiuntivo connesso alla costante $C0000000|_{16}$ con un valore del PCSOURCE di 11_2 lo selezionerà. Siccome il PC viene incrementato nel primo ciclo non si può scrivere il valore del PC nell' EPC ma si dovrà usare l'ALU per togliere 4 al PC e scrivere il risultato nell' EPC , quindi l'input dell' EPC è collegata all'output della ALU. Nel momento di un'eccezione si salvano i dati nei registri per non mutarli durante la gestione dell'eccezione. In caso di eccezioni annidate si procede con la FIFO (*first in first out*). Si ottiene il seguente datapath multiciclo:



Capitolo 7. Eccezioni

con la seguente macchina a stato finito:

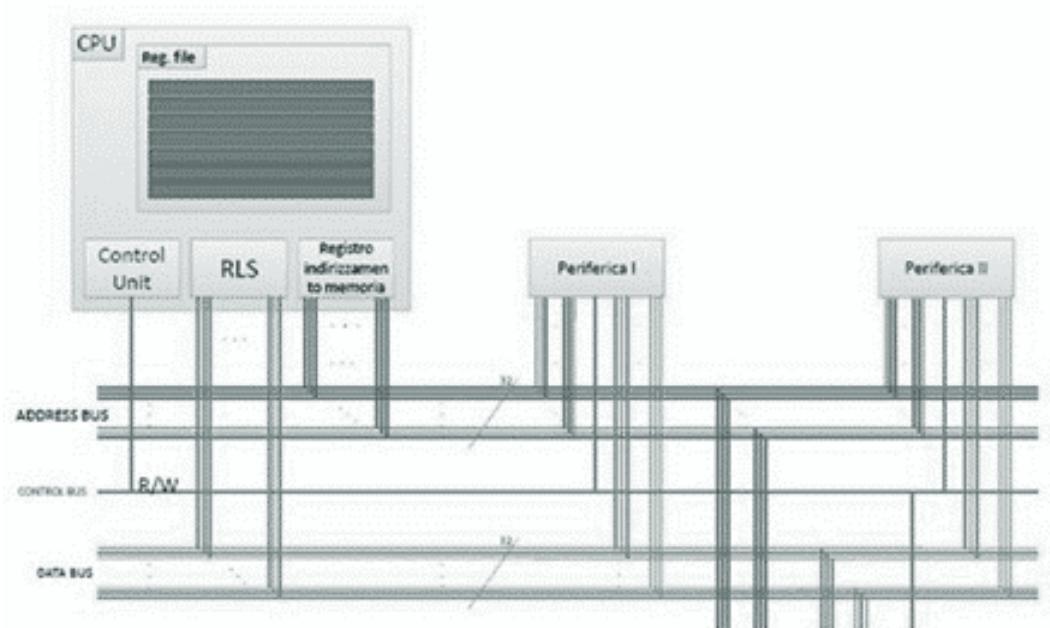


Capitolo 8

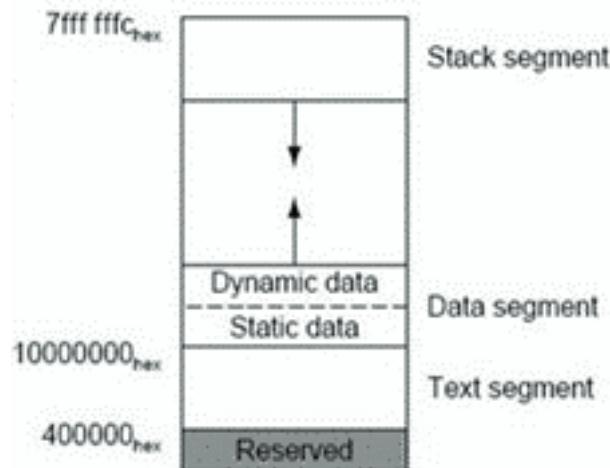
Gestione I/O

Si prende in considerazione un'architettura con un unico bus, sia per l'accesso alla memoria che per la comunicazione con le periferiche. Il bus si divide in:

1. *bus dati*
2. *bus di controllo*
3. *bus degli indirizzi*

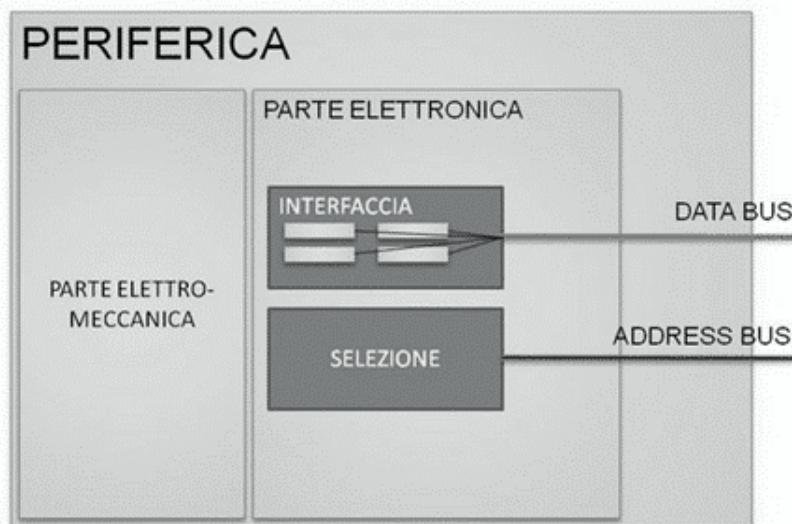


Per la gestione delle periferiche si usa la parte di memoria oltre la locazione $0x7FFFFFFC$, normalmente riservata al sistema:



dove ci saranno gli vari spazi dedicati alle varie periferiche e ogni registro della periferica appare anche in memoria in una locazione dedicata. Questa cosa permette le operazioni di accesso alla periferica esattamente come le letture e scritture in memoria.

Nel dettaglio una periferica è un componente elettromeccanico e la cpu ci comunica mediante un'interfaccia costituita da *registri periferica* e specifica la periferica coinvolta in una certa operazione mediante un indirizzo.



se nel bus si ha la giusta combinazione binaria si attiva la periferica corrispondente.

Si possono identificare un insieme minimale di registri di interfaccia nella comunicazione tra cpu e periferica:

- un *registro di stato* che rappresenta lo stato della periferica (se per esempio può ricevere dati in quel momento etc)
- un *registro dati* che è un registro di input output per la cpu a seconda della periferica (di ingresso, come la tastiera, o di uscita, come la stampante)

Si crea un problema: la cpu esegue ininterrottamente istruzioni ad una certa frequenza mentre le periferiche generano dati solo in certi momenti ad una certa frequenza (inferiore solitamente a quella della cpu) e si può avere un input umano che potrebbe arrivare dopo tempi molto lunghi. Per gestire il problema si introduce il *controllo di programma*. La cpu controlla il valore del registro di stato di una periferica e ne copia il valore dalla locazione nello spazio di indirizzamento in cui è mappato ad un registro della cpu, per poterlo usare successivamente in una comparazione. Se la periferica non è pronta la cpu riesegue il controllo sul registro di stato e finché la periferica non è pronta resta impegnata in questa operazione, sprecando energia elettrica e impedendo alla cpu di eseguire altro. Questa situazione è chiamata *BUSY WAIT*. Quando la periferica diventa disponibile si ha il trasferimento dei dati. Il trasferimento dal registro della periferica (mappata in memoria) alla memoria dati si ottiene caricandone prima il valore nei registri della cpu, una lw/sw da registro periferica e poi una lw/sw con la memoria.

Vediamo l'esempio di un caricamento da tastiera:

```
.text
.globl main

main:      la $t0, 0xfffff0000      # recupera l'indirizzo Receiver Control
           la $t2, 0xfffff0004      # recupera l'indirizzo Receiver Data

BusyWaitRead: lw $t1, 0($t0)          # legge Receiver Control Register
              andi $t1, $t1, 0x1      # tengo solo il bit 0
              beqz $t1, BusyWaitRead # se è 0 non è arrivato nulla
                                      continuo a controllare

              lb $t3, 0($t2)          # se arrivo qua significa
                                      che è arrivato un carattere
                                      e leggo il byte da
                                      Receiver Data register
              sb $t3, 0x10010000      # salvo il dato in memoria
```

Si hanno 2 metodi per valutare l'efficienza della gestione:

1. la *banda passante*: ovvero la quantità di dati che si può trasferire per unità di tempo. Si ha una misura di flusso
2. la *latenza*: ovvero è il tempo che intercorre tra l'istanza *Ready*, che indica che la periferica è pronta, e l'istante in cui il dato viene trasferito. Si ha una misura di tempo

Nel controllo di programma la latenza è minima in quanto la cpu noterà in fretta che la periferica è ready (il tempo peggiore è un ciclo completo di BUSY READ), si ha inoltre che la cpu trasferirà subito il dato e la banda passante sarà alta in quanto la gestione della periferica richiede poche istruzioni.
Vediamo ora come gestire il trasferimento multiplo di dati. Si avrà lo stesso nucleo di comunicazione visto sopra (chiamato *ciclo interno*) che dovrà essere ripetuto molte volte, in un *ciclo esterno* che comprenderà altre istruzioni ausiliarie (come decrementare il contatore dei byte rimasti o incrementare un puntatore all'area in cui depositare o prelevare i dati, verificando poi che siano stati tutti trasmessi).

Si ha il seguente diagramma di flusso (la parte a sinistra è il ciclo esterno, quella a destra quello interno):



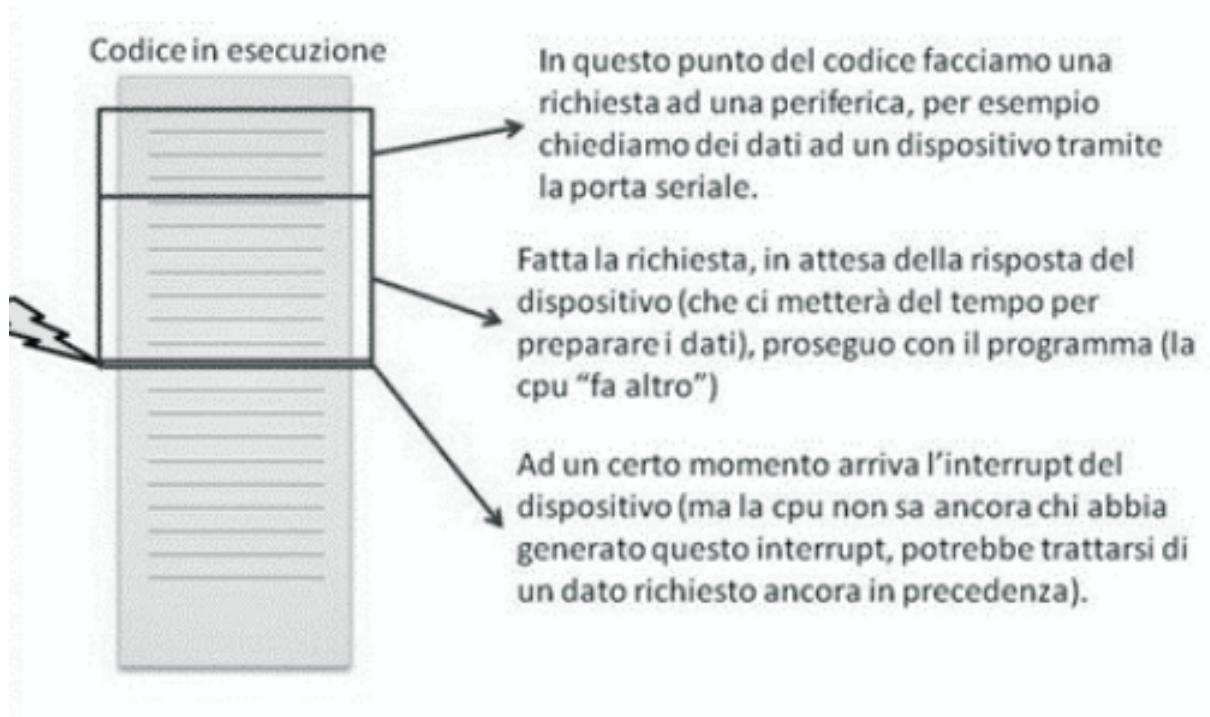
Si ha che alcune periferiche hanno più urgenza di trasferire i dati rispetto ad altre (per esempio se si leggono dati da un HDD si aumenta il rischio di perdere dati se la cpu non li legge abbastanza in fretta, si avrà infatti una sovrapposizione di essi).

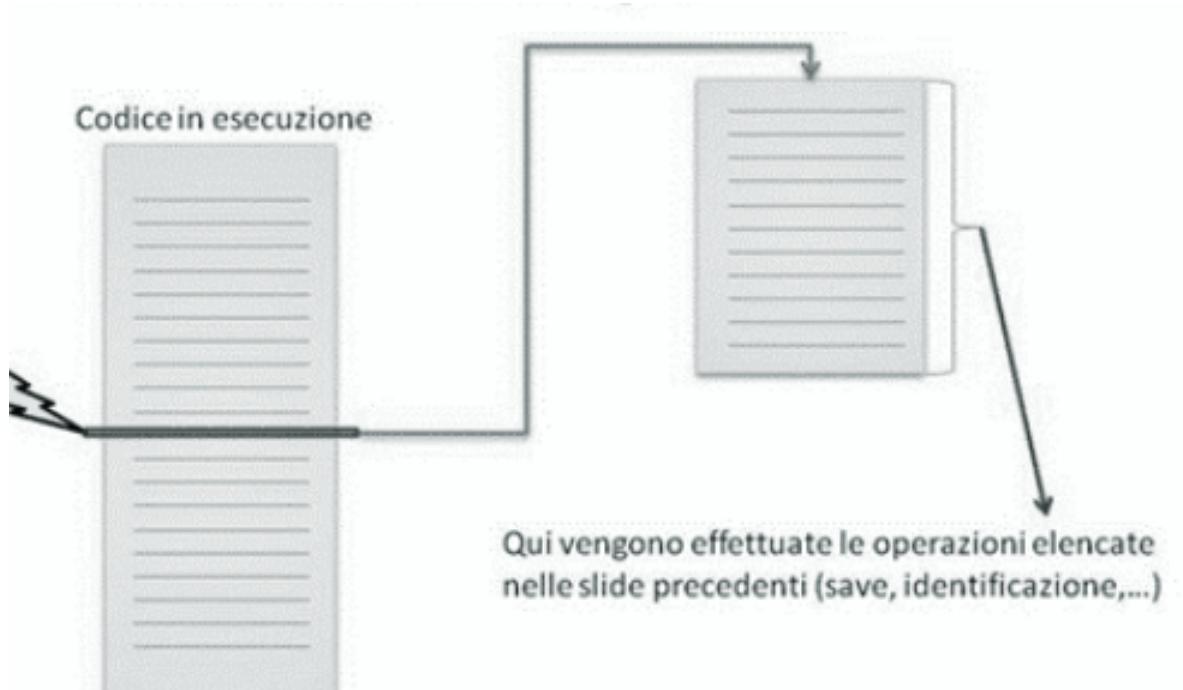
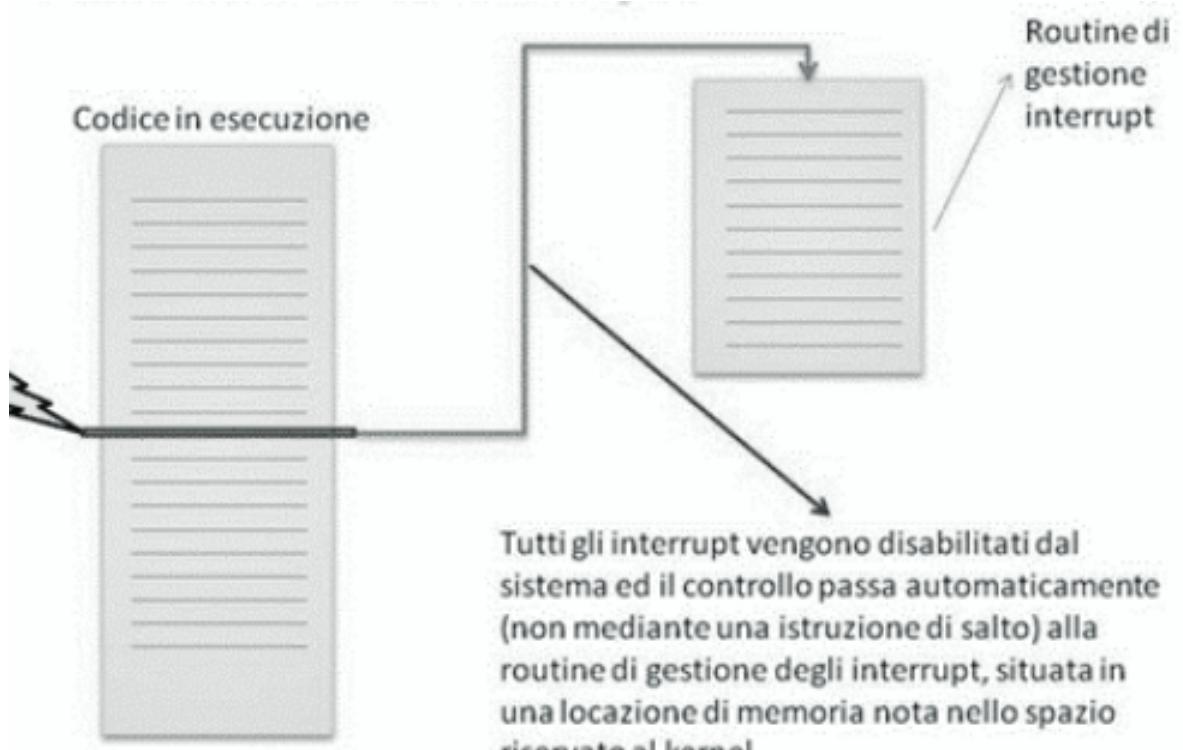
Analizziamo eventuali interruzioni durante l'input/output. Ci si riferisce alle interrupt. Si vuole infatti liberare la cpu dal BUSY WAIT e si cerca un meccanismo che faccia comunicare cpu e periferiche senza l'intervento diretto della cpu. Questo meccanismo si chiama interrupt e richiede dell'hardware aggiuntivo. Sul fronte del bus si ha l'aggiunta del bus di controllo che trasmette il segnale ready dalla periferica alla cpu. Questa linea è chiamata *linea di richiesta* dell'interruzione. 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:

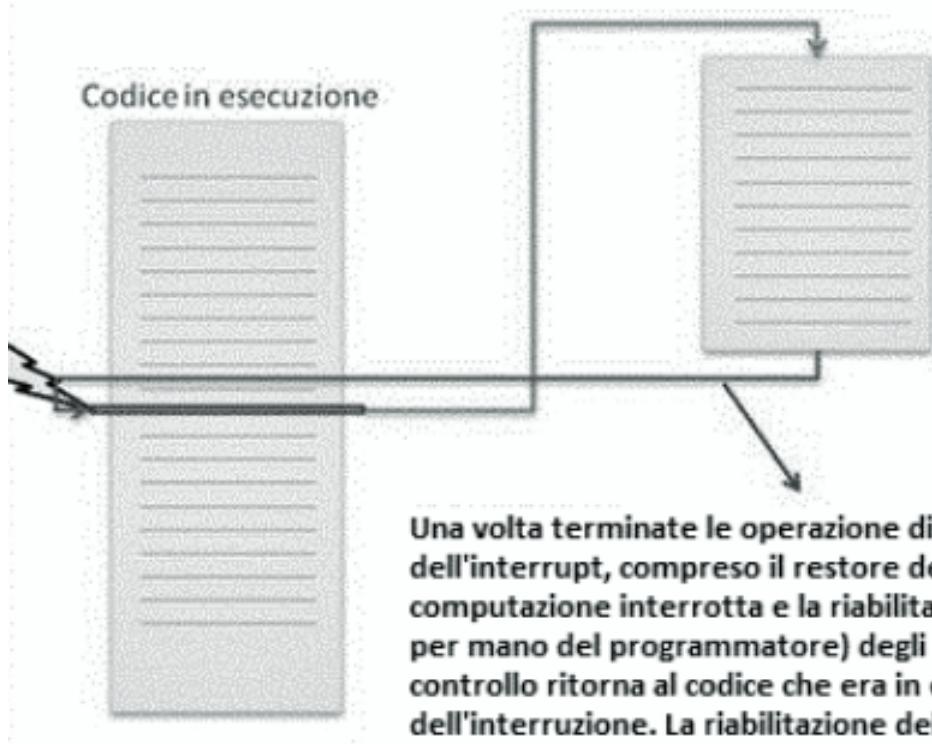
- *save* che salva lo stato della computazione al momento dell'interrupt
- identificazione della periferica interrompendente

- gestione della periferica, col trasferimento del dato
- *restore* che ripristina lo stato della computazione a prima dell'interruzione
- ritorno dall'interrupt (*eret*)

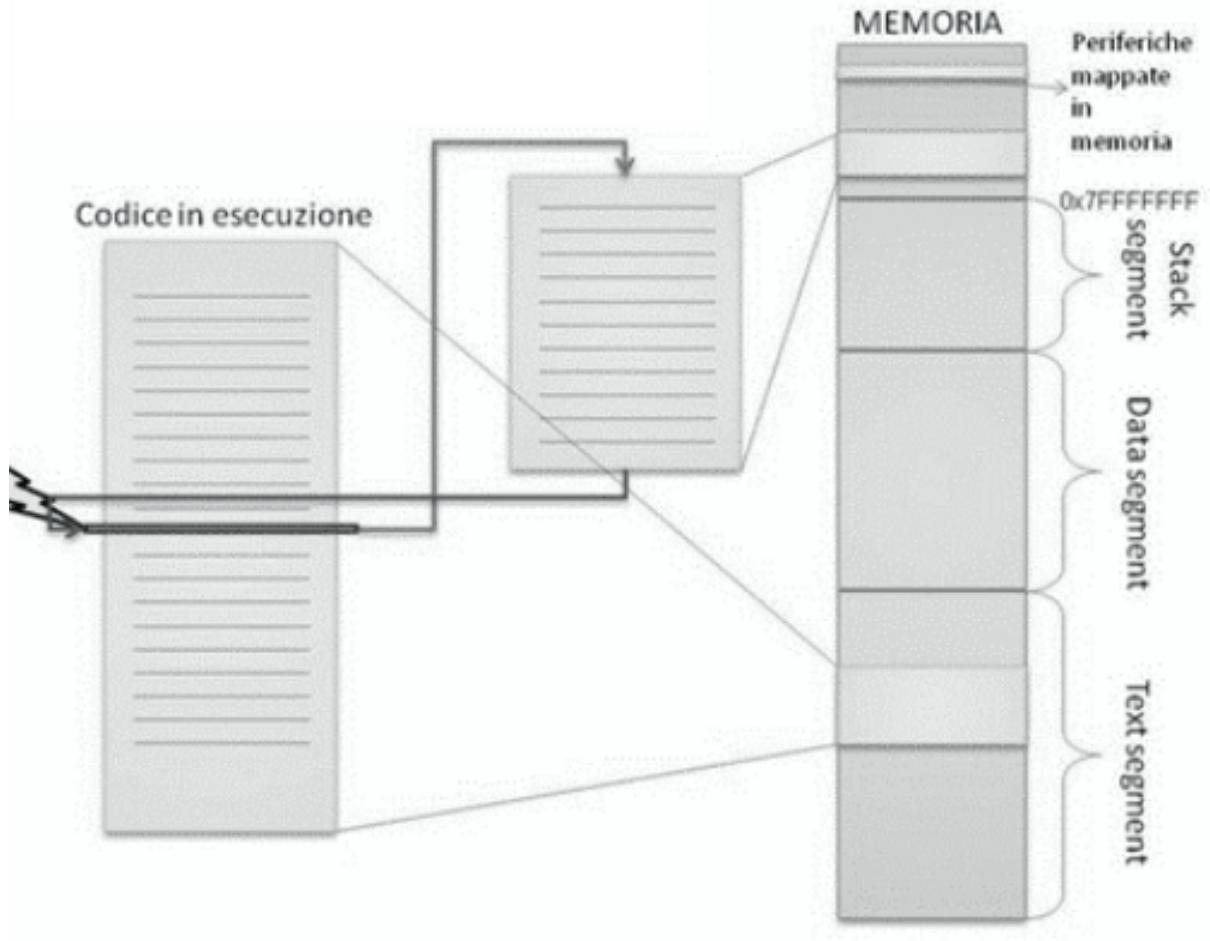
Vediamo un diagramma di flusso esplicativo:





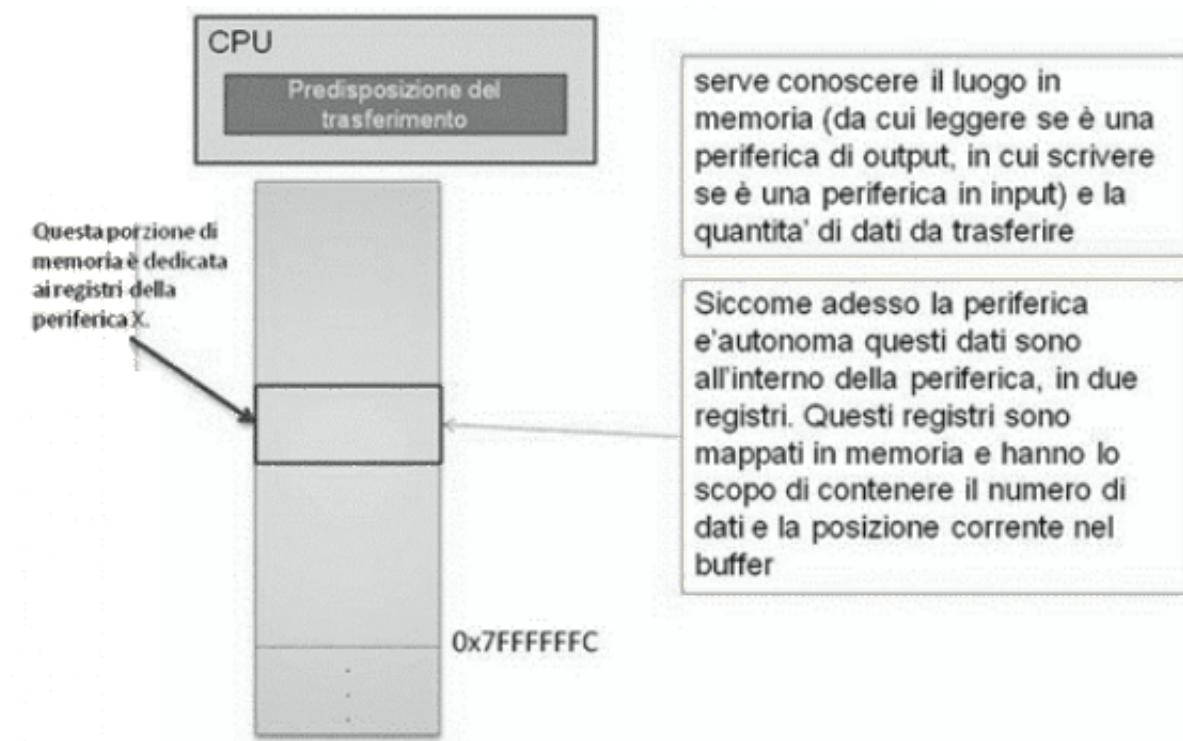


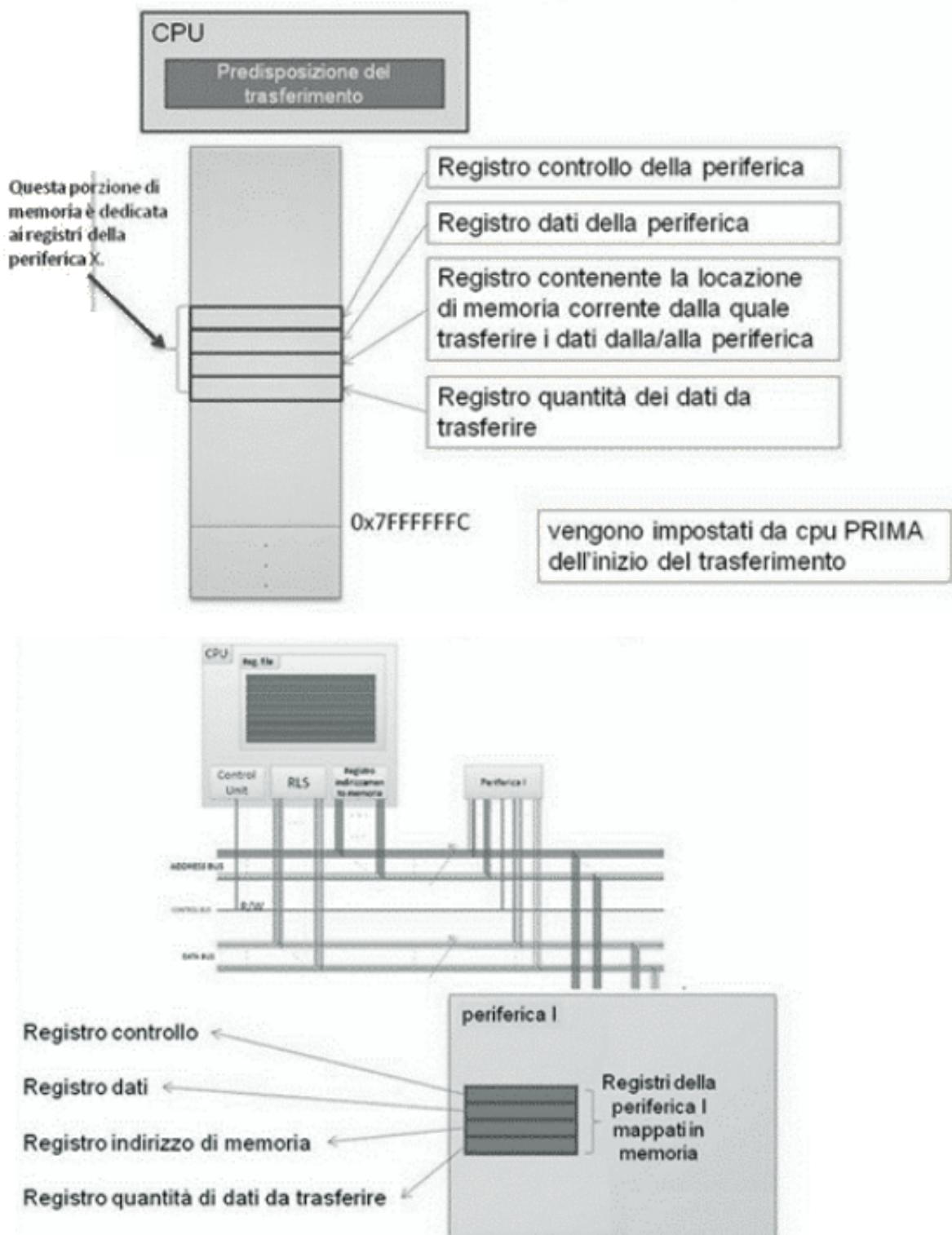
Una volta terminate le operazioni di gestione dell'interrupt, compreso il restore dello stato computazione interrotta e la riabilitazione (questa volta per mano del programmatore) degli interrupt, il controllo ritorna al codice che era in esecuzione prima dell'interruzione. La riabilitazione delle interruzioni e la alterazione del flusso di controllo sono svolte dall'istruzione eret.



Facciamo un esempio semplice, con un'unica linea di interruzione per tutte le periferiche. Si hanno quindi tutte le uscite delle periferiche saldate su un unico filo, *wired OR*. Il primo problema all'arrivo di un interrupt è capire quale periferica lo ha generato. Con una sola linea di interruzione bisognerà controllare il bit di stato nel registro di stato di ogni periferica, che è un'operazione onerosa. Si ha però un metodo più complesso per migliorare le performance, usare la vettorizzazione. L'interrupt genera un codice che la identifica su delle linee di bus dedicate che va a finire in un registro della cpu. A design time si sa un particolare indirizzo di memoria che è la base del vettore con le interruzioni. Quando arriva l'interrupt la cpu usa il codice come spiazzamento nel vettore interruzioni ed esegue il codice all'indirizzo "base del vettore+codice periferica". In MIPS32 si ha un unico gestore con una sola linea di interruzione ma c'è, nella cpu, modo di analizzare lo stato di alcune linee di interruzione grazie al cause register.

Passiamo alla gestione delle periferiche attraverso *DMA* (*Direct Memory Access*). Il DMA permette di superare il limite della banda bassa passante e della eccessiva latenza data dagli interrupt, permettendo inoltre di non eseguire il BUSY WAIT. Il DMA è utile per eseguire sequenze di trasferimento elementari e si usa solitamente per periferiche critiche (per le altre si usano gli interrupt). Con il DMA si rende la periferica autonoma nell'accesso alla memoria, così da renderla capace di gestire da sola i trasferimenti, liberando la cpu dal BUSY WAIT e ottenendo performance elevate a livello di banda passante e latenza. Per usare il DMA bisogna aggiungere 2 registri periferica, mappati in memoria, ai 2 precedentemente visti che servono per specificare l'indirizzo di memoria con il quale scambiare dati e per specificare la quantità dei dati da trasferire. Ecco una spiegazione della predisposizione:



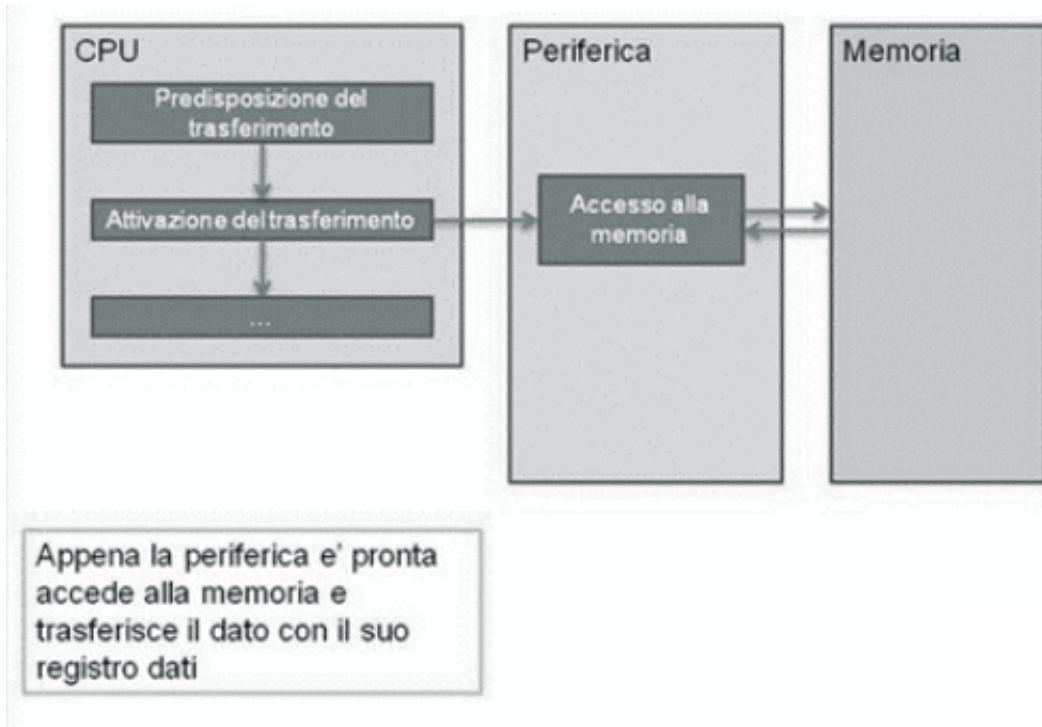


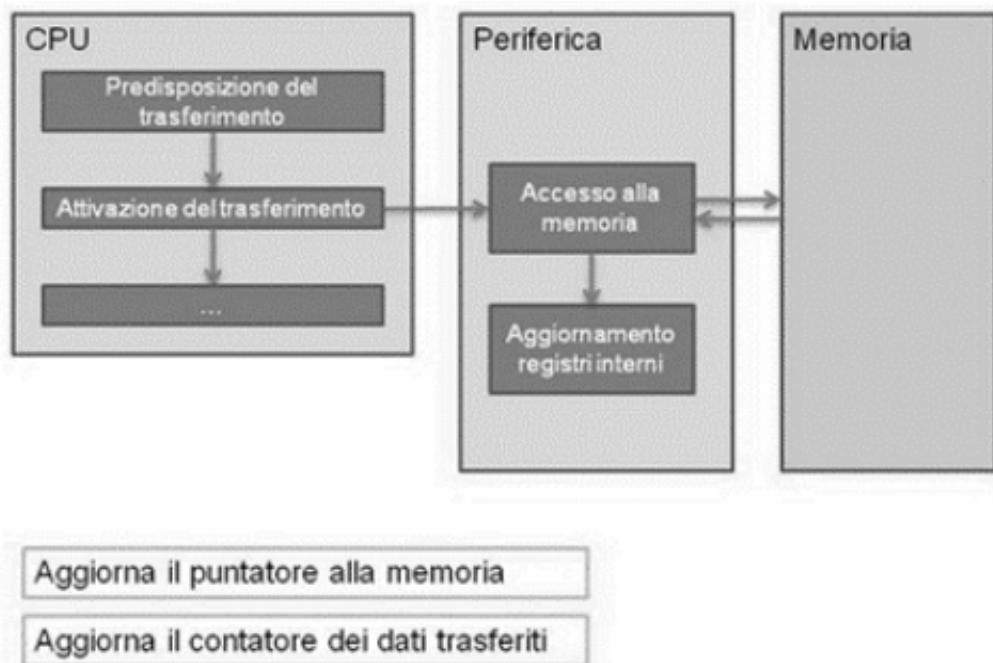
Ecco una spiegazione dell'attivazione:

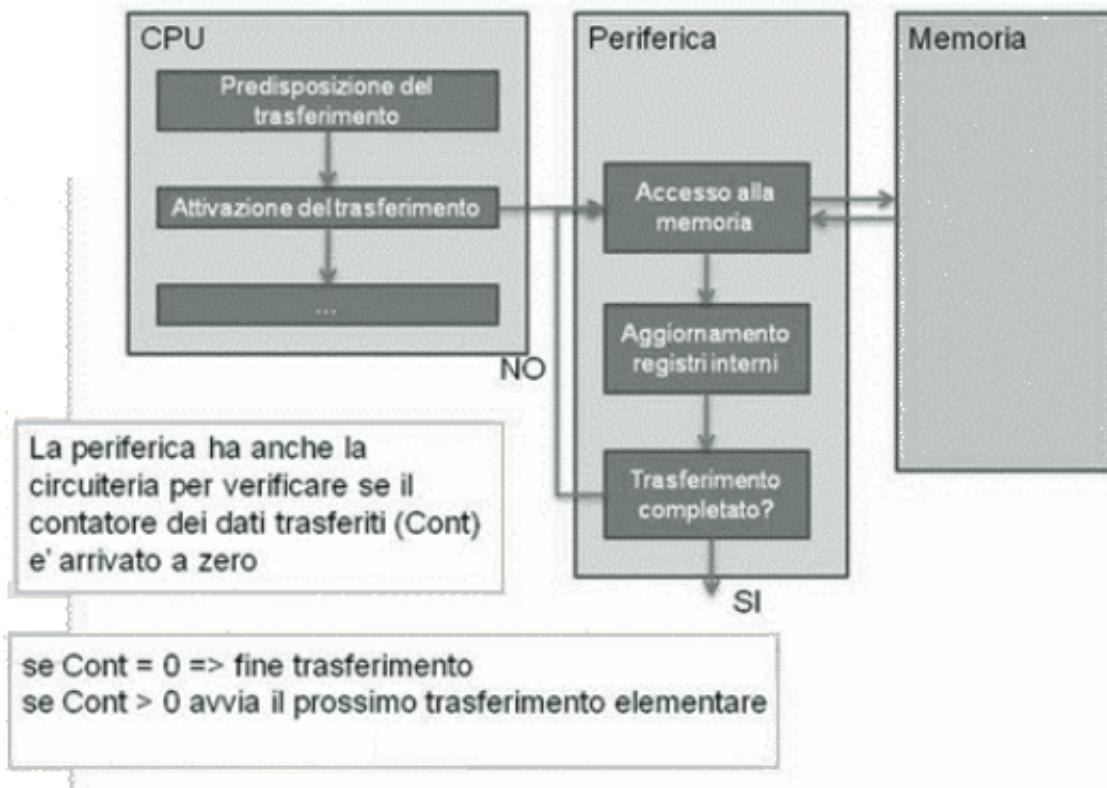


Da questo momento in poi la CPU puo' fare altro, mentre la periferica aspetta il tempo tra un trasferimento elementare ed il successivo

Ecco una spiegazione del trasferimento elementare:

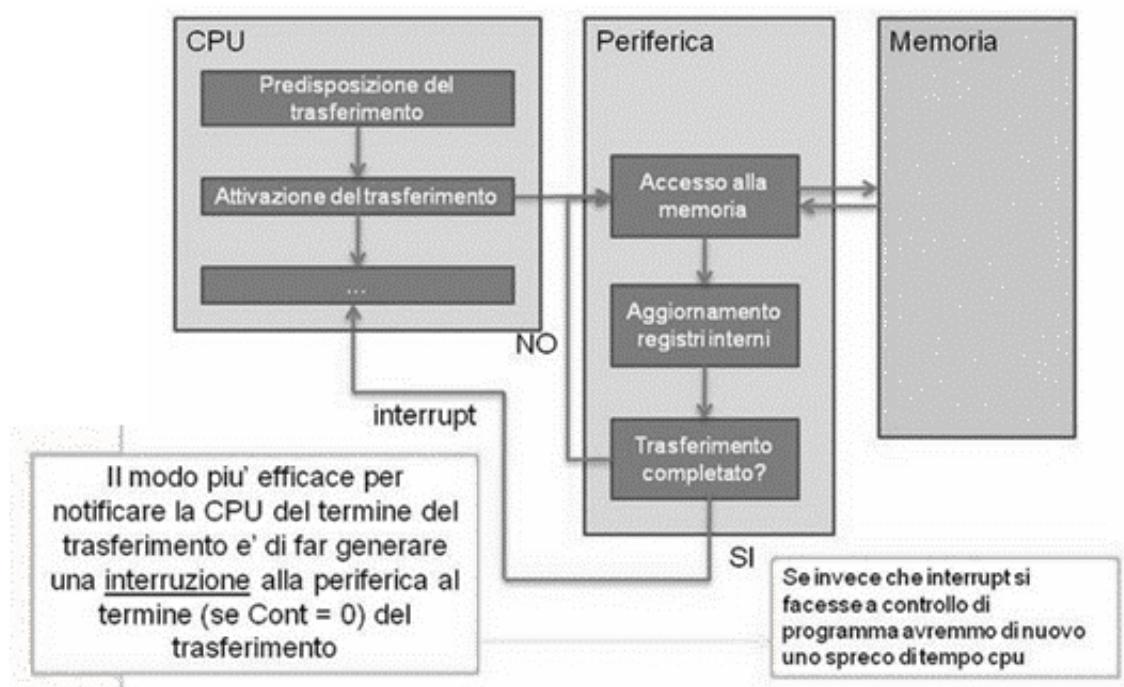
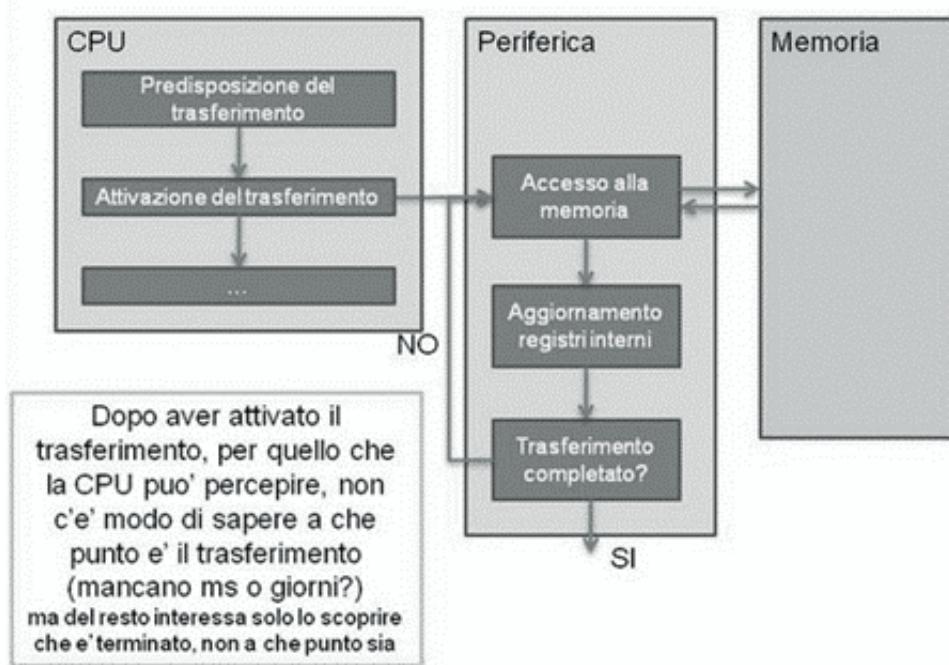






Si nota che servono linee di controllo per stabilire (ad arbitrio del bus) chi far accedere alla memoria tra cpu e periferica, nel caso vogliano accedere contemporaneamente.

Ecco una spiegazione dell'interruzione finale:



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

Capitolo 8. Gestione I/O

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 9

Cache

Si ha che un programma non accede a tutte le sue istruzioni e a tutti i suoi dati contemporaneamente con la stessa probabilità. Come soluzione si è trovata la *gerarchia di memoria* che consiste in un insieme di livelli di memoria, ciascuno caratterizzato da una diversa velocità e dimensione (ovviamente maggiore è la velocità maggiore è il costo). Un programma in un certo momento accede soltanto ad una piccola porzione del suo spazio di indirizzamento, è il *principio di località* ed è la base del comportamento di un calcolatore. Si hanno due tipi di località:

1. *temporale*: quando si fa riferimento a un elemento c'è la tendenza a fare riferimento allo stesso elemento dopo poco tempo
2. *spaziale*: quando si fa riferimento a un elemento c'è la tendenza a fare riferimento poco dopo ad altri elementi che hanno l'indirizzo vicino ad esso

questo principio struttura la memoria in modo gerarchico: più è veloce e più è vicina al processore, più è grande e più è lontana, più è costosa e più è vicina al processore e più è economica più è lontana. Un livello di memoria vicina alla cpu contiene dati memorizzati in ogni livello sottostante e tutti i dati sono nel livello più basso.

Si hanno le seguenti definizioni:

- *blocco/linea*: la più piccola quantità di informazione che può essere presente/assente in una gerarchia di memoria
- *hit*: l'informazione richiesta dal processore si trova in uno dei blocchi nel livello superiore di memoria
- *miss*: l'informazione richiesta dal processore non si trova in uno dei blocchi nel livello superiore di memoria

- *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($1 - HitRate$)
- *hit & miss rate*: usate per determinare le prestazioni
- *tempo di hit*: tempo di accesso al livello superiore della memoria, compreso il tempo necessario a stabilire se il tempo di accesso si risolva in un successo o in un fallimento
- *penalità 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

In una gerarchia di memoria ci sono più livelli ma i dati vengono trasferiti solo tra due livelli vicini

9.1 Cache

La cache (dal francese caché=nascosto, in quanto trasparente al programmatore) è il livello della memoria gerarchica che si trova tra il processore e la memoria principale. Si parlerà di *Direct Mapped Cache* che associa una sola locazione della cache ad ogni word della memoria definendo una corrispondenza tra l'indirizzo in memoria e la locazione nella cache. Si ha, per trovare il blocco che corrisponde ad un indirizzo della memoria principale:

indirizzo del blocco % numero di blocchi della cache

si hanno poi i seguenti campi:

- *tag*: etichetta, contiene informazioni per verificare se una word della cache corrisponde ad una word cercata
- *indice*: usato per selezionare il blocco della cache
- *blocco di validità*: indica se il blocco di memoria associato contiene (true) o no (false) un dato valido

inoltre si ha che:

- la dimensione della cache è 2^n blocchi, per cui n bit vengono usati per l'indice

- la dimensione della cache è di 2^m parole, ossia $m + 2$ byte, per cui m bit vengono usati per individuare una parola all'interno di un blocco, mentre 2 bit per individuare un byte all'interno di una parola
- la dimensione del campo tag è $32 - (n + m + 2)$
- il numero totale di bit in una cache a mappatura diretta è:

$$2^n \times (\text{dim_blocco} + \text{dim_cache} + \text{bit_validita}) = 2^n \times (2^m \times 32 + (32 - (n+m+2)) + 1)$$

anche se per convenzione si considera solo la dimensione dati

Esempio 11. da quanti bit è costruita una cache a mappatura diretta con 16kb di dati e blocchi da 4 word, con indirizzo di memoria a 32bit?

$$16KByte = 4k \text{ parole} = 2^{12} \text{ parole}$$

$$\text{un blocco} = 4 \text{ parole} = 2^2 \rightarrow 2^{10} \text{ blocchi nella cache}, m = 2 \text{ e } n = 10$$

$$\text{un blocco ha } 4 \times 32 = 128 \text{ bit}$$

$$\text{tag} = 32 - (10 + 2 + 2) = 18$$

$$\text{dimensione cache} = 2^{10} \times (4 \times 32 + 18 + 1) = 147Kbit$$

una cache con blocchi più grandi sfrutta maggiormente la località spaziale diminuendo la frequenza di miss che però torna a crescere se la dimensione dei blocchi è troppo grande rispetto alla dimensione della cache infatti diminuiscono i blocchi salvabili e cresce la competizione per essere salvati, i blocchi vengono scaricati nella cache prima ancora di utilizzare i dati e quindi diminuisce la località spaziale e il miglioramento della frequenza di miss cala. Inoltre cresce il costo di una miss e la penalità è data dal tempo necessario a prelevare un blocco dal livello sottostante e scriverlo nella cache (tempo dato dalla somma tra la latenza per ottenere la prima word del blocco e il tempo di trasferimento del resto del blocco). Questo incremento di penalità ha un impatto superiore al calo di frequenza e le prestazioni della miss calano

Esercizio 15. un programma impiega 200ms ad essere eseguito senza cache con un tempo di accesso alla ram di 50ns. Si aggiunge una cache con tempi di 2ns. La hit rate è di 0.9 quindi qual è il tempo di esecuzione? il numero degli accessi alla memoria è:

$$\frac{200ms}{50ns} = \frac{200 \times 10^{-3}}{50 \times 10^{-9}} = 4 \times 10^6$$

tempo di esecuzione = tempo cache + tempo RAM:

$$4 \times 10^6 \times 2ns \times 10^{-9} * 0.9 + 4 \times 10^6 \times 50ns \times (1 - 0.9) = 27.2ms$$

Esercizio 16. data una macchina con 5 cicli per accessi solo in cache (hit), 15 cicli di miss penalty e hit probability di 0,75 qual è il numero medio di cicli per accesso in memoria?

numero medio = hit × hit probability + miss × miss probability:

$$n = 5 \times 0,75 + 15 \times 0,25 = 7,5$$

Esercizio 17. Si consideri una cache con 64 blocchi da 16byte ciascuno. A quale blocco corrisponde l'indirizzo 1200, espresso in byte, della memoria principale?

il blocco contenente il dato sarà:

(indirizzo del blocco) modulo (numero dei blocchi nella cache)

$$\text{indirizzo_del_blocco} = \frac{\text{indirizzo_del_dato_in_byte}}{\text{byte_per_blocco}} = \frac{1200}{16} = 75$$

Si ricorda che le parole appartenenti al blocco individuato hanno indirizzo compreso tra:

$$\frac{\text{indirizzo_del_dato_in_byte}}{\text{byte_per_blocco}} \times \text{byte_per_blocco} = 1200$$

e

$$\frac{\text{indirizzo_del_dato_in_byte}}{\text{byte_per_blocco}} \times \text{byte_per_blocco} + (\text{byte_per_blocco} - 1) = 1215$$

si ha quindi:

(indirizzo del blocco) modulo (numero dei blocchi nella cache) = $75\%64 = 11$
sul blocco 11 vengono mappati tutti i dati tra l'indirizzo 1200 e il 1215 della memoria principale

Capitolo 10

Tip & Tricks

- i nanosecondi sono 10^{-9} secondi
- $T = \frac{1}{\nu}$
- l'intervallo di numeri rappresentabili in complemento a 2 con n bit è $[-2^{n-1}, 2^{n-1} - 1]$
- un carattere ascii è rappresentato da 7bit
- $(-1)^S \times (1 + \text{mantissa}) \times 2^{\text{esponente}-127}$
- il numero di bit che compongono un elemento della ROM è anche l'ampiezza
- il numero massimo di ingressi di un multiplexor a n linee di selezione è $\log_2(x) = n \rightarrow x = 2^n$ (si arrotonda per eccesso, se ho 12 ingressi, per esempio, conto 4 linee di selezione)
- il numero di linee di selezione in un decoder a n linee di uscita è $2^x = n \rightarrow x = \log_2(n)$
- se devo specificare il valore nei 26bit di una jump in esadecimale conto quante istruzioni ho fino alla label di destinazione, moltiplico per 4 trasformo il valore in hex. Sommo quel valore all'esadecimale dell'indirizzo di partenza. Converto il risultato in binario e tolgo i 4bit più significativi e i 2 meno significativi. Infine riconverto in hex.
- se devo specificare il valore nei 16bit in decimali di una beq conto quante istruzioni ho fino alla label di destinazione, moltiplico per 4 trasformo il valore in binario, tolgo i due bit meno significativi e ritrasformo in binario

- un'etichetta in assembly identifica una locazione di memoria
- con una procedura ho $a1, a2, a3, a4$ come indirizzi per il passaggio dei parametri ingresso e $v0, v1$ per quelli in uscita
- i registri per le eccezioni sono quelli del coprocessore 0
- se il cause register ha un valore in hex lo trasformo in binario. Prendo i 7 bit meno significativi e ci tolgo gli ultimi due. Traduco il rimanente (quei 5bit) in decimale e leggo la tabella
- $IndirizzoBlocco = \frac{IndirizzoDatoInByte}{BytePerBlocco}$ e
 $NumeroDBloccoMappatoInMemoriaPrincipale = IndirizzoBlocco \% NumeroBlocchi$
- $CPI_m = CPI \cdot HitProbability + CicliMissPenalty \cdot (1 - HitProbability)$
- $Velocita_m = \frac{FrequenzaInHz}{CPI_m}$ esprssa in $\frac{istruzioni}{secondi}$
- se si ha una cache a mappaggio diretto con x blocchi da y word e z cicli di loop e si cerca quanti miss si hanno per utilizzare delle certe locazioni di memoria si scrive da 0 a $(x \cdot y) - 1$ andando a capo ogni y (ogni riga sarà un blocco). Se si ha che una locazione di memoria non è stata contata tra i blocchi si fa +1 ai miss e si annulla il blocco (non ci saranno più miss lì), esempio se ho il blocco 0,1 e la locazione di memoria 1 faccio +1 a miss ma se quella dopo è 0 non faccio più +1 a miss
- se devo rappresentare in virgola fissa un numero x tra max e min in n bit faccio $\frac{max-min}{2^n}$ e verifico quante cifre decimali ho. Se x ha 3 cifre decimali e dal conto esce un numero con 2 allora l'errore di approssimazione sarà 0,00z, con z terza cifra decimale di x

Capitolo 11

Approfondimento

11.1 IEE754

Innanzitutto ricordiamo la formula generale per esprimere un numero in virgola mobile:

$$(-1)^S \times F \times 2^E$$

con:

- **S most significant bit.** Si usa per il segno
- **F mantissa** di *23 bit* che rappresenta il numero frazionario
- **E esponente** di *8bit* (incluso il segno)

I numeri in virgola mobile sono normalmente dei multipli della dimensione della parola. La rappresentazione di un numero in virgola mobile sul **MIPS** è quella appena specificata. Questa rappresentazione è chiamata modulo e segno, dal momento che il segno costituisce un bit separato dal resto del numero. Nel **MIPS32** si possono rappresentare numeri piccoli fino a $2|_{10} \times 10^{-38}$ e numeri grandi fino $2|_{10} \times 10^{38}$, nonostante questo si può comunque incappare in situazioni di **overflow** (ovvero che l'esponente è troppo grande per essere rappresentato). Si ha anche il cosiddetto **underflow** per quando si ottengono cifre così piccole da non poter essere rappresentate.

Un modo per ridurre questi problemi è aumentare l'esponente a *11bit* e ridurre la mantissa a *20bit* e aggiungere un intero word (*32bit*) per la mantissa (che quindi sarà su *52bit*). Questa è la cosiddetta **rappresentazione in virgola mobile in doppia precisione**. Ad alto livello quest'ultima rappresenta i double del C mentre la notazione precedente il float. Con la doppia precisione si possono rappresentare numeri piccoli fino a $2|_{10} \times 10^{-308}$ e numeri grandi fino $2|_{10} \times 10^{308}$ e si ha anche un aumento di precisione grazie alla

mantissa aumentata.

Questi formati sono importanti anche fuori dal mondo del MIPS e fanno parte dello standard **IEEE 754** per la virgola mobile, che si trova utilizzato praticamente in ogni calcolatore. Per far stare ancora più bit all'interno della mantissa, lo IEEE 754 rende implicito il bit a 1 iniziale dei numeri binari normalizzati. Quindi la mantissa è in realtà di *24bit* in singola precisione e *53bit* in doppia. Poiché lo O non ha 1 iniziali, gli si è assegnato il valore riservato con esponente nullo, cosicché non abbia un 1 iniziale.

Si ottiene quindi il seguente formato:

$$(-1)^S \times (1 + F) \times 2^E$$

dove i bit della mantissa rappresentano la frazione tra O e 1, ed E specifica il valore nel campo esponente.

Nonostante la formula l'ordine di rappresentazione in bit è $S + E + F$. posizionamento dell'esponente prima della mantissa semplifica inoltre l'ordinamento dei numeri in virgola mobile utilizzando le istruzioni di confronto tra interi, in quanto i numeri con esponente maggiore sono più grandi dei numeri con esponente minore, purché ambedue gli esponenti abbiano lo stesso segno. Gli esponenti negativi rappresentano un problema per l'ordinamento. Se si usa il complemento a due o una qualunque altra notazione in cui gli esponenti negativi hanno un 1 nel bit più significativo del campo esponente, un esponente negativo apparirà come un numero grande. Si procede quindi con la **notazione polarizzata** e la polarizzazione è il numero sottratto alla rappresentazione normale senza segno per determinare il valore reale. Lo IEEE 754 usa una polarizzazione pari a 127 per la singola precisione, così che -1 è rappresentato dalla combinazione di bit di valore $-1 + 127|_{10} = 126|_{10} = 01111110|_2$ e +1 è rappresentato da $1 + 127$, ossia $128|_{10} = 10000000$. Si ottiene quindi:

$$(-1)^S \times (1 + F) \times 2^{E-P}$$

Con P polarizzazione pari a 127 per la singola precisione e 1023 per la doppia. Quindi il formato IEEE 754 può essere processato dalle istruzioni di confronto tra interi per accelerare l'ordinamento dei numeri in virgola mobile

Esempio 12. Scrivo le due rappresentazioni di $-0.75|_{10}$: Innanzitutto $-0.75 = -\frac{3}{2^2}$ e $3|_{10} = 11_2$ essendo in base due mi sposto in base all'esponente 2, essendo il denominatore 2^2 mi sposto di 2, ottenendo $-0,11|_2$.

In notazione scientifica è quindi $-0,11|_2 \times 2^0$ che normalizzando diventa $-1,1|_2 \times 2^{-1}$. Quindi si ha che $E - 127 = -1$ e quindi $E = 126$ mentre la mantissa sarà $1 + F = |1,1|$ quindi F è la parte frazionaria di 1,1 e ovviamente $S = 1$ in quanto il numero è negativo.

Si ottiene quindi, essendo $E = 126_{10} = 01111110$ e $F = 1[0]22$:

1 01111110 10000000000000000000

in doppia precisione si ha $E - 1023 = -1$ e quindi $E = 1022$ mentre la mantissa sarà $1+F = |1,1|$ quindi F è la parte frazionaria di $1,1$ e ovviamente $S = 1$ in quanto il numero è negativo.

Si ottiene quindi, essendo $E = 1022|_{10} = 0111111110$ e $F = 1[0]51$:

Esempio 13. Ricavo il decimale di:

110000001010000000000000000000000000

Innanzitutto sappiamo di avere a che fare con un numero negativo, in quanto il MSB è 1. Il campo esponente è $10000001_2 = 129_{10}$ e la mantissa è $1 \times 2^{-2} = 0,25_{10}$ (per la mantissa si scala da 2^{-1}). Applico quindi la formula e ottengo:

$$(-1)^1 \times (1 + 0, 25) \times 2^{129-127} = -1, 25 \times 4 = -5, 0$$

11.1.1 Operazioni in virgola mobile

Somma

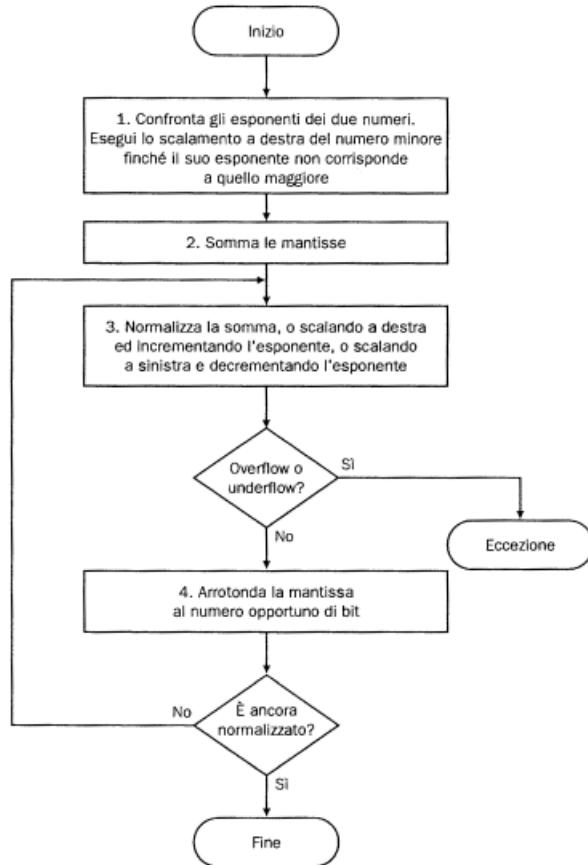
Innanzitutto ricordiamo come si procede per la somma in notazione scientifica in base 10. Vogliamo sommare $9,999 \times 10^1 + 1,610 \times 10^{-1}$. Innanzitutto si deve allineare la virgola del numero che ha l'esponente più piccolo. Quindi si deve trovare una forma per il numero minore che corrisponda in termini di esponente con il numero di esponente più grande. Quindi otteniamo che $1,610 \times 10^{-1} = 0,01610 \times 10^1$. Quindi, ragionando sui numeri in virgola mobile, **primo passo esegue lo scalamento a destra della mantissa del numero più piccolo, fino a che il suo esponente non coincide con quello del numero più grande.** Si possono però rappresentare solo 4 cifre decimali, quindi $0,016 \times 10^1$. Tornando ai nostri numeri in base 10 si ha ora la somma dei coefficienti: $9,999 + 0,016 = 10,015$ ovvero $10,015 \times 10^1$. Questo passaggio in virgola mobile è **la somma delle mantisse**. Tornando alla base 10 dobbiamo normalizzare il valore quindi: $10,015 \times 10^1 = 1,0015 \times 10^2$. Questa operazione non sempre è necessaria, **sistemando, anche in termini di operazioni in virgola mobile, l'esponente.** Ogni volta che l'esponente è incrementato o decrementato, si deve controllare se si è verificato un overflow o un underflow, cioè ci si deve accertare che l'esponente continui ad

essere rappresentabile all'interno del suo campo. Poiché si è assunto che la mantissa possa essere lunga soltanto quattro cifre (escludendo il segno), è necessario arrotondare il numero. Quindi: $1,0015 \times 10^2 = 1,002 \times 10^2$. *Si noti che esiste una caso sfortunato per l'arrotondamento, consistente nel dover sommare un 1 ad una stringa di 9: la somma può non essere più normalizzata ed occorre eseguire nuovamente la normalizzazione sistemando l'esponente.* A tal proposito si ricorda che per la singola precisione l'esponente più grande è 127 e quello più piccolo -126. I limiti per la doppia precisione sono 1023 e -1022. Viene quindi usato il **troncamento**, che è una delle quattro opzioni di arrotondamento previste dallo standard IEEE 754. Se ho uno 0 lascio zero se ho un 1 metto 1. Per esempio per arrotondare su 8 cifre decimali:

$$1,000000000001 \rightarrow 1,00000000$$

$$1,00000001111 \rightarrow 1,00000001$$

L'accuratezza del calcolo in virgola mobile dipende per una buona parte dall'accuratezza dell'arrotondamento e quindi, benché sia facile da realizzare, il troncamento porta lontani dall'accuratezza. Riassumendo:



Esempio 14. Si effettui $0,5|_{10} + (-0,4375|_{10})$.

Innanzitutto effettuiamo le dovute conversioni:

$$0,5|_{10} = \frac{1}{2}|_{10} = \frac{1}{2^1}|_{10} = 0,1|_2 = 0,1|_2 \times 2^0 = 1,000|_2 \times 10^{-1}$$

$$-0,4375|_{10} = -\frac{7}{16}|_{10} = -\frac{7}{2^4}|_{10} = \overbrace{-0,0111|_2}^{7|_{10}=111|_2} = -0,0111|_2 \times 2^0 = -1,110|_2 \times 2^{-2}$$

Procediamo ora con gli step sopra spiegati.

Innanzitutto scaliamo la mantissa del numero con esponente minore, per far coincidere l'esponente con quello dell'esponente maggiore:

$$-1,110|_2 \times 2^{-2} = -0,111 \times 2^{-1}$$

Sommiamo le due mantisse, per comodità $1|_2 = 1|_{10}$ e $0,111|_2 = 0,875|_{10}$:

$$1,000|_2 \times 2^{-1} + (-0,111|_2 \times 2^{-1}) = 0,001|_2 \times 2^{-1}$$

Normalizziamo controllando eventuali overflow e underflow:

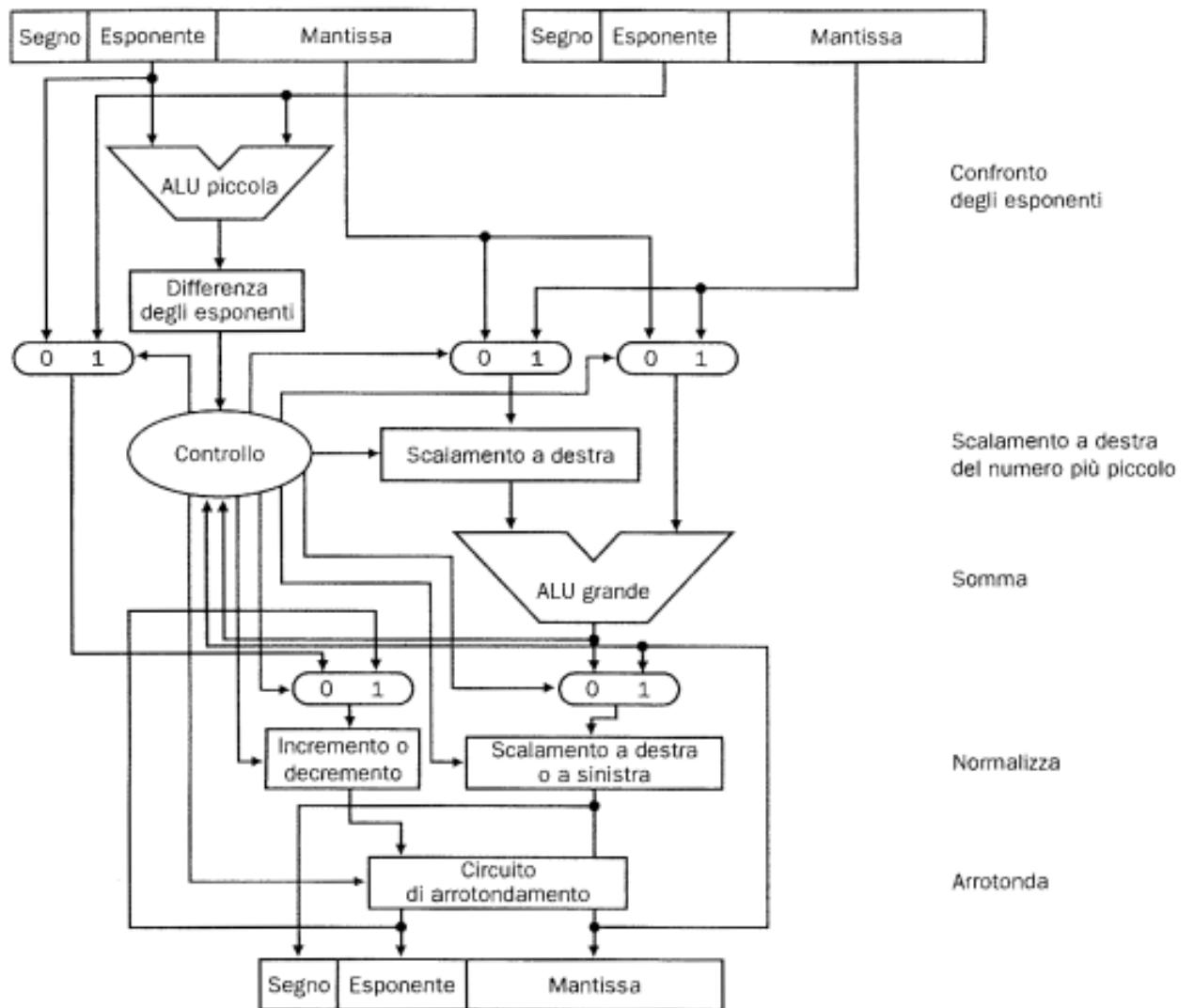
$$0,001|_2 \times 2^{-1} = 1,000|_2 \times 2^{-4}$$

e $-126 \leq -4 \leq 127$ e quindi non si ha ne overflow ne underflow. L'esponente polarizzato sarebbe $-4 + 127$, ossia 123, compreso tra 1 e 254, che sono l'esponente più piccolo e quello più grande tra quelli non riservati.

Infine si arrotonda la somma ma in questo caso è già in 4 bit quindi non si hanno modifiche.

La somma è quindi $1,000|_2 \times 2^{-4} = 0,0001|_2 = \frac{1}{2^4|_{10}} = \frac{1}{16|_{10}} = 0,0625|_{10}$

A livello hardware si ha:



Moltiplicazione e Divisione

Come prima procediamo con un esempio, moltiplicando $1,110|_{10} \times 10^{10}$ per $9,200 \times 10^{-5}$ e salvando 4 cifre per la mantissa e 2 per l'esponente.

Innanzitutto il **primo passo** consiste nel calcolare l'esponente del prodotto semplicemente sommando gli esponenti degli operandi. Nel nostro caso il nuovo esponente sarà $10 + (-5) = 5$.

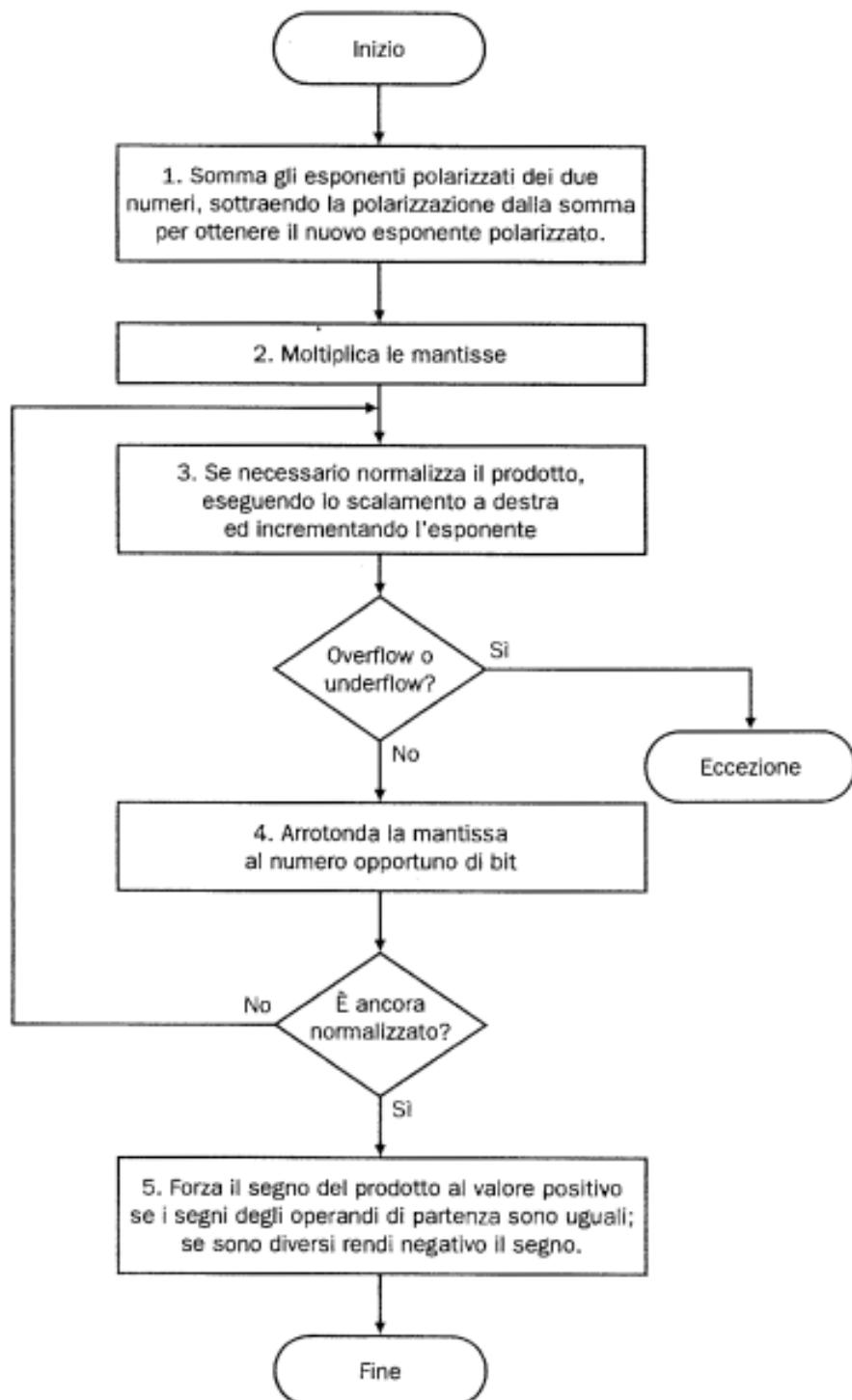
Si ripeta ora l'operazione con gli esponenti polarizzati, in maniera da essere sicuri di ottenere lo stesso risultato: $-10 + 127 = 137$ e $-5 + 127 = 122$, e quindi $137 + 122 = 259$ ma c'è un problema: il risultato è troppo grande per il campo esponente di 8 bit. Infatti quando si sommano numeri polarizzati, per ottenere il risultato corretto della somma si deve sottrarre la polarizzazione dalla somma:

$$137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

Si procede poi con la moltiplicazione delle mantisse: $1,110|_{10} \times 9,200|_{10} = 10,21200|_{10} = 10,212 \times 10^5|_{10}$. Si controllano eventuali underflow o overflow

Il prossimo passo consiste nella normalizzazione del risultato. Quindi $10,212 \times 10^5|_{10} = 1,0212 \times 10^6|_{10}$ ma da richiesta ci servono 4 cifre per la mantissa quindi $1,021 \times 10^6|_{10}$.

Il prossimo passo è studiare il segno che dipende dai segni dei due operandi come di consueto, ed essendo entrambi positivi nel nostro caso lo sarà anche il risultato che quindi sarà $1,021 \times 10^6|_{10}$.



Esempio 15. Moltiplichiamo $0,5|_{10}$ per $-0,4375|_{10}$.

Convertendo in binario si ha $1,000|_{-1}$ per $-1,110|_{-2}$.

Sommiamo gli esponenti $-1 + (-2) = -3$. Polarizzati sarebbero: $(-1 + 127) + (-2 + 127) - 127 = (-1 - 2) + (127 + 127 - 127) = -3 + 127 = 124$.

Moltiplichiamo le mantisse $1,000|_2 \times 1,110|_2 = 1,110000 \times 2^{-3}$ ma va ridotto a 4bit: $1,110 \times 2^{-3}$.

Essendo l'esponente $-126 \leq -3 \leq 127$, o normalizzato $1 \leq 124 \leq 254$, non si hanno underflow o overflow.

Non si hanno arrotondamenti e il segno sarà negativo in quanto gli operandi sono di segno opposto.

Quindi il risultato sarà $-1,110 \times 2^{-3}$.

Converto in decimale per sicurezza: $-1,110 \times 2^{-3} = -0,001110|_2 = -0,00111|_2 = -\frac{7}{2^5}|_{10} = -\frac{7}{32}|_{10} = -0,21875|_{10}$ che è esattamente il risultato in decimali cercato.

Per la divisione le uniche differenze sono la sottrazione degli esponenti (e non la somma) e la divisione delle mantisse (e non la loro moltiplicazione)

11.1.2 Virgola Mobile in MIPS

Il MIPS supporta i formati IEEE 754 per la singola e la doppia precisione con le seguenti istruzioni:

- somma in virgola mobile: addition_single (*add.s*) e addition_double (*add.d*)
- sottrazione in virgola mobile: subtraction_single (*sub.s*) e subtraction_double (*sub.d*)
- moltiplicazione in virgola mobile: multiplication_single (*mul.s*) e multiplication_double (*mul.d*)
- divisione in virgola mobile: division_single (*div.s*) e division_double (*div.d*)
- confronto in virgola mobile: comparison_single (*e.x.s*) e comparison_double (*e.x.d*), dove x può essere equal (*eq*), not equal (*neq*), less than (*lt*), less than or equal (*le*), greater than (*gt*), oppure greater than or equal (*ge*)
- salto in virgola mobile: branch_true (*belt*), branchJalse (*bclf*)

Il confronto in virgola mobile forza a vero o falso un bit, a seconda della condizione di confronto, ed un salto in virgola mobile decide

poi se saltare o meno, a seconda della condizione.

In MIPS i registri in virgola mobile sono nominati $\$f[1 - 9]*$ e si hanno operazioni separate di load e stare per i registri in virgola mobile: $lwc1$ e $swc1$. Un registro in doppia precisione in realtà corrisponde ad una coppia pari e dispari di registri per la singola precisione, che prende il nome dal registro pari.

Vediamo un esempio di codice:

```
lwcl $f4,x($sp) #carica il numero v.m. su 32 bit in F4
lwcl $f6,y($sp) # carica il numero v.m. su 32 bit in F6
add.s $f2,$f4,$f6 # F2=F4+F6 in singola precisione
swcl $f2,z($sp) #memorizza il numero v.m. su 32 bit da $f2
```

In caso di conti che comportano risultati indefiniti si ha un formato speciale: *NaN*, ovvero **Not A Number**.

Ricordiamo inoltre questa comoda tabella per capire come rappresentare le varie situazioni

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	\pm denormalized number
1–254	Anything	1–2046	Anything	\pm floating-point number
255	0	2047	0	\pm infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

FIGURE 3.13 EEE 754 encoding of floating-point numbers. A separate sign bit determines the sign. Denormalized numbers are described in the *Elaboration* on page 230. This information is also found in Column 4 of the LEGv8 Reference Data Card at the front of this book.

11.2 Pipeline

Con **pipeline** si indica una tecnica implementativa che sfrutta l'esecuzione in sovrapposizione di diverse istruzioni.

Si introduce quindi il **calcolo parallelo** in quanto le singole istruzioni impieghereanno lo stesso tempo ma si avranno più istruzioni svolte in parallelo. Le istruzioni MIPS richiedono tipicamente cinque passi:

1. prelievo dell'istruzione dalla memoria

2. lettura dei registri e decodifica dell'istruzione (il formato delle istruzioni MIPS permette la simultaneità tra la lettura e la decodifica)
3. esecuzione dell'operazione o calcolo dell'indirizzo
4. accesso ad un operando nella memoria dati
5. scrittura del risultato in un registro

Prima di procedere con la tecnica valutiamo un rapporto tra i tempi, se gli stadi sono perfettamente bilanciati si ha, in condizioni ideali:

$$t_{pipeline} = \frac{t_{noPipeline}}{numeroStadi}$$

considerando che la pipeline del MIPS ha 5 stadi in condizioni ideali è 5 volte più veloce dell'implementazione a singolo ciclo.

Nella realtà per non si hanno stadi bilanciati e si ha l'inserimento di overhead.

In definitiva il tempo per istruzione in un sistema con pipeline sarà maggiore del minimo possibile, per cui lo speedup sarà minore del numero di stadi della pipeline.

La tecnica delle pipeline migliora le prestazioni aumentando il throughput delle istruzioni, e non riducendo il tempo di esecuzione di ogni singola istruzione. Fortunatamente il throughput delle istruzioni è la metrica più importante, dal momento che i programmi reali eseguono miliardi di istruzioni.

Passiamo ora al MIPS.

Innanzitutto tutte le istruzioni MIPS hanno la stessa lunghezza: tale restrizione semplifica notevolmente il prelievo delle istruzioni nel primo stadio della pipeline e la loro decodifica nel secondo stadio.

In secondo luogo il MIPS ha un numero ridotto di formati delle istruzioni ed in tutti i formati i registri sorgente sono sempre posti nelle stesse posizioni, qualunque sia l'istruzione. Tale simmetria permette al secondo stadio di iniziare a leggere il register file nello stesso tempo in cui il circuito determina il tipo dell'istruzione reperita: se il formato delle istruzioni MIPS non fosse stato simmetrico, sarebbe stato necessario spezzare lo stadio 2, portando la pipeline ad un totale di 6 stadi (si vedrà tra non molto lo svantaggio di avere pipeline più lunghe).

In terzo luogo nel MIPS gli operandi di tipo memoria possono comparire solo nelle istruzioni di load e di store: questo vincolo permette di usare lo stadio di esecuzione per calcolare l'indirizzo di memoria, accedendo alla memoria nello stadio successivo altrimenti gli stadi 3 e 4 dovrebbero essere espansi in

uno stadio per l'indirizzo, uno per la memoria ed un terzo per l'esecuzione. Un quarto fattore è l'allineamento obbligatorio degli operandi in memoria non è necessario considerare casi in cui una singola istruzione di trasferimento dati richieda due accessi alla memoria, per cui i dati possono essere trasferiti tra il processore e la memoria in un solo stadio della pipeline. Nella pipeline si hanno i cosiddetti **hazard (criticità)**, per esempio dati dal fatto che l'istruzione successiva non può essere eseguita nel ciclo di dock immediatamente seguente. Si hanno 3 tipi di hazard principali:

1. **structural hazard**: indica che le risorse hardware presenti non sono in grado di supportare la combinazione di istruzioni che si vorrebbe eseguire nello stesso ciclo di clock.

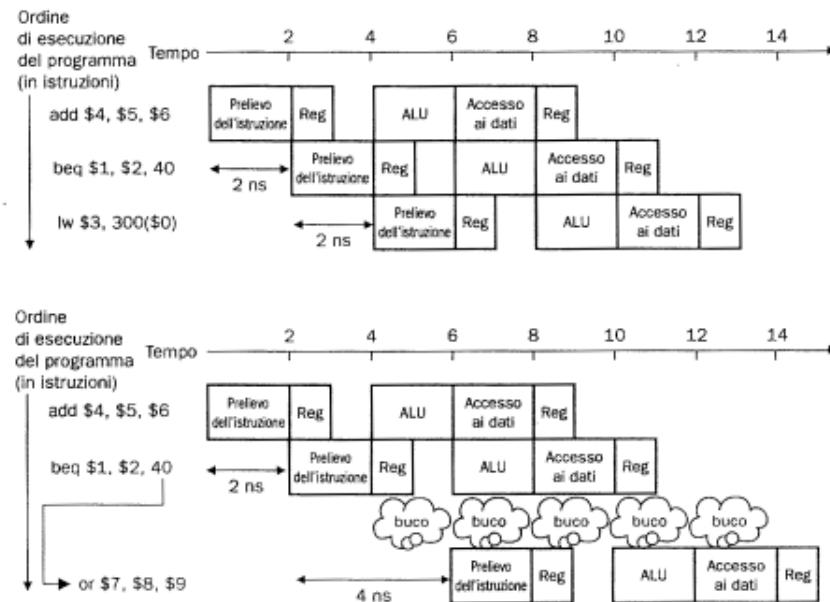
Il set di istruzioni di MIPS è stato progettato per supportare le pipeline senza structural hazards ma si potrebbero avere problemi con gli accessi in memoria

2. **control hazard**: che sono determinate dalla necessità di dover compiere alcune scelte in funzione del risultato di un'istruzione mentre altre sono in esecuzione.

si hanno due soluzioni per questo hazard:

- (a) **stall (stallo)**: ovvero lavorare in modo sequenziale fino a che non si ottiene il risultato corretto. Quest'opzione è **lenta**.

In MIPS si implementa con il salto condizionato; se l'elaboratore dovesse entrare in stallo sui salti condizionati dovrebbe sospendere l'esecuzione mediante pipeline procedendo in modo sequenziale fino al termine dell'istruzione di salto. Si ha quindi uno *stall della pipeline*, in gergo *bubble (buco)*. In pratica:



Abbiamo capito che non è questa la soluzione normalmente adottata nei calcolatori

- (b) **predict (predizione)**: ovvero se si è quasi sicuri di avere la corretta sequenza di istruzioni effettuare una sorta di, appunto, predizione.

In effetti i calcolatori usano la tecnica della predizione per gestire i salti condizionati. L'approccio più semplice consiste nel predire sempre che il salto non venga eseguito: in caso di predizione corretta la pipeline procede al massimo della velocità, mentre solo quando i salti vengono eseguiti si ha uno stall della pipeline.

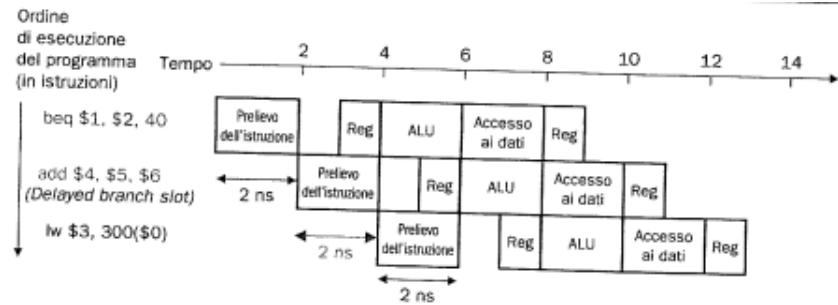
Una versione più sofisticata della predizione dei salti richiederebbe di predire alcuni salti come eseguiti ed altri come non eseguiti. Si considerino ad esempio i salti che si trovano al termine dei cicli che servono a ritornare all'inizio dell'iterazione successiva: poiché essi verranno probabilmente eseguiti e poiché saltano all'indietro, il processore potrebbe predire che i salti verso indirizzi minori verranno eseguiti. Non si tiene quindi conto dell'individualità delle specifiche istruzioni di salto.

I circuiti per la **predizione dinamica** dei salti, in netto contrasto, scelgono la predizione da fare in funzione del comportamento di ciascuna istruzione di salto, e possono modificare la predizione fatta per un certo salto durante l'esecuzione del programma. Un approccio per la predizione dinamica consiste nel memorizzare la *storia* di ogni salto, ricordando se è stato eseguito o meno, utilizz-

zando il *passato* per predire il *futuro* (cosa che avviene con una precisione del 90%). In caso di predizione errata la logica di controllo della pipeline deve garantire che le istruzioni che seguivano quella d1 salto non abbiano alcun effetto e deve far ripartire la pipeline dall'indirizzo corretto.

Il problema si aggrava all'aumentare della pipeline in quanto aumenta il costo di predizioni errate.

Esempio con il bubble dell'esempio sopra riempito da un *add*:



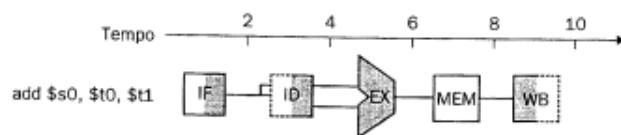
3. **data hazard:** ovvero quando un'istruzione dipende dal risultato di un'istruzione precedente che si trova ancora nella pipeline.

Vediamo un esempio:

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

Se non si prendessero contromisure, le criticità sui dati potrebbero causare dei seri stalli nella pipeline: l'istruzione *add* non scrive il proprio risultato prima del quinto stadio, per cui si dovrebbero aggiungere tre bubble alla pipeline.

Questo non è un problema che può risolvere il calcolatore con risultati apprezzabili. La soluzione più utilizzata è basata sull'osservazione che non è necessario attendere il termine dell'istruzione per cercare di risolvere la criticità sui dati: nella sequenza di codice appena vista, non appena la ALU genera il risultato della somma, questo può venire usato come ingresso per l'operazione di sottrazione. Si prende quindi in anticipo il dato mancante dalle *risorse interne*. Questa operazione viene detta **forwarding** (*propagazione*) o **bypassing** (*scavalcamiento*).

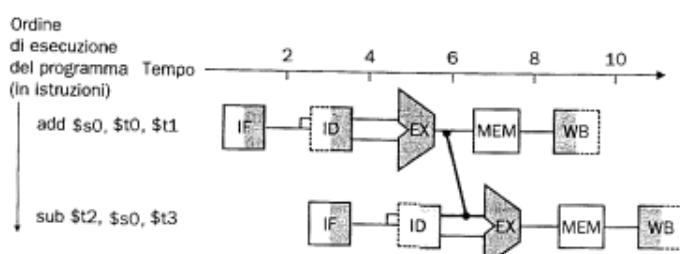


Con la seguente notazione:

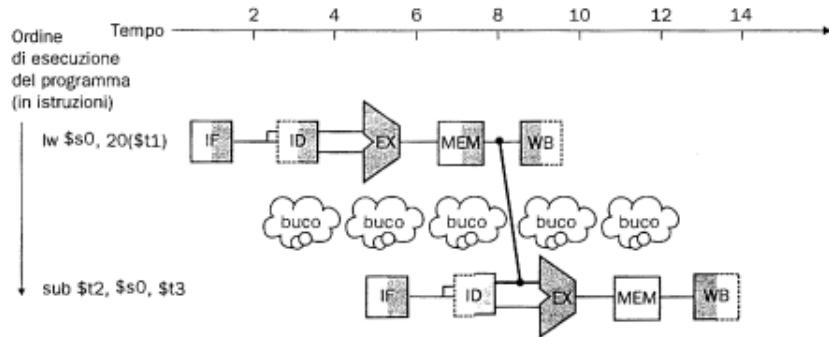
- **IF (*instruction fetch*)**, per lo stadio di prelievo dell’istruzione. Il quadrato rappresenta la memoria delle istruzioni.
- **ID (*instruction decode*)**, per lo stadio di decodifica delle istruzioni e lettura da *register file* in cui il simbolo rappresenta il register file che viene letto
- **EX**, per lo stadio di esecuzione, dove il disegno rappresenta la ALU
- **MEM** per lo stadio di accesso alla memoria, in cui il quadrato rappresenta la memoria dei dati
- **WB (*write back*)**, per lo stadio di scrittura dei risultati, , in cui il simbolo rappresenta il register file che viene scritto

Gli stadi evidenziati sono quelli utilizzati dall’istruzione (infatti MEM non viene usato). L’evidenziazione della metà destra del register file o della memoria indica che tale elemento letto in quello stadio, mentre l’evidenziazione della metà sionisrea si riferisce all’operazione di scrittura

Ecco invece i collegamenti necessari a propagare il valore di \$s O calcolato nello stadio di esecuzione dell’istruzione *add* fino ad arrivare all’ingresso dello stadio di esecuzione dell’istruzione *sub*. Il collegamento indica il cammino di propagazione dall’uscita dello stadio EX dell’istruzione *add* all’ingresso dello stadio EX dell’istruzione *sub*, sostituendo quindi il valore del registro \$sO letto nel secondo stadio della *sub*.



La tecnica del forwarding non previene però tutti gli stalli della pipeline. Un esempio lo si ottiene cambiando l’*add*, nell’esempio sopra, in un caricamento di \$s0; il dato voluto sarebbe disponibile solo dopo il quarto stadio della prima istruzione, ossia troppo tardi per essere usato come ingresso del terzo stadio della sottrazione. Si imporebbe uno stallo di uno stadio a causa del **load-use data hazard**:



Vediamo anche una piccola correzione di un codice per evitare stalli:

```
# nel registro $t1 c'è l'indirizzo di v[k]

lw    $t0, 0($t1)      # $t0 (temp) = v[k]
lw    $t2, 4($t1)      # $t2 = v[k+1]
sw    $t2, 0($t1)      # v[k] = $t2
sw    $t0, 4($t1)      # v[k+1] = $t0 (temp)
```

La criticità si presenta sul registro `$t2`, tra la seconda `lw` e la prima `sw`; lo scambio tra le due istruzioni `sw` elimina la criticità:

```
# nel registro $t1 c'è l'indirizzo di v[k]

lw    $t0, 0($t1)      # $t0 (temp) = v[k]
lw    $t2, 4($t1)      # $t2 = v[k+1]
sw    $t0, 4($t1)      # v[k+1] = $t0 (temp)
sw    $t2, 0($t1)      # v[k] = $t2 $
```

Si noti che in questo modo non si crea una nuova criticità in quanto vi è ancora un'istruzione tra la scrittura del registro `$t0` da parte della load e la lettura dello stesso registro da parte della store. In un calcolatore dotato di propagazione la sequenza precedente, riordinata, impiega dunque 4 cicli di clock.

Inoltre si noti che ciascuna operazione MIPS scrive un solo risultato e unicamente al termine della propria esecuzione. La propagazione sarebbe molto più difficile se vi fossero più risultati da propagare per ciascuna istruzione o se questi venissero scritti prima del termine dell'istruzione.

11.2.1 Pipelined Datapath