

Architettura degli elaboratori

Sistemi numerici

Per far capire ai computer una sequenza di byte, sono stati definiti degli **standard di codifica**, ovvero regole che vengono utilizzate nella rappresentazione dei dati in formato binario.

Dati k bit, il numero di configurazioni possibili è 2^k . Per definire meglio questi standard di codifica, sono stati introdotti vari termini:

Bit	Valore 0 o 1	Nibble	4 bit
Byte	8 bit	Halfword	16 bit
Word	32 bit	Doubleword	64 bit

Nei sistemi numerici, sono presenti due "oggetti": l'**entità**, ovvero il valore che vogliamo gestire, e la sua eventuale **rappresentazione** nel sistema numerico dato.

Esempio: Dato il valore "sedici", la rappresentazione decimale è 16_{10} , la sua rappresentazione binaria è 10000_2 . Anche se hanno diverse rappresentazioni, sono la stessa entità.

Sistema posizionale

Un sistema posizionale è un sistema dove ogni cifra del numero dato assume un valore in base alla posizione all'interno del numero.

Un esempio è il **sistema decimale**, dove, a seconda della posizione, una cifra può essere unità, decina, centinaia, migliaia, etc...

Più in specifico:

$$N = d_{n-1}, d_{n-2}, \dots, d_1, d_0, d_{-1}, \dots, d_{-m}$$

$$N = d_{n-1} * r^{n-1} + \dots + d_0 * r^0 + d_{-1} * r^{-1} + \dots + d_{-m} * r^{-m}$$

$$N = \sum_{i=-m}^{n-1} d_i * r^i$$

Esempio: $R = 10, d = 0, 1, \dots, 9$

$$123,45 = 1 * 10^2 + 2 * 10^1 + 3 * 10^0 + 4 * 10^{-1} + 5 * 10^{-2}$$

Positivi: NO VIRGOLA | Negativi: CON VIRGOLA

Vediamo i vari sistemi numerici più conosciuti:

Sistema decimale

- **Cifre:** 10
- **Posizionale?** Si
- **Lettere?** No
- **Base:** 10

Sistema romano

- **Cifre:** I, V, X, L, M, C
- **Posizionale?** No – un simbolo ha un significato indipendentemente dalla posizione in cui esso compare
- **Lettere?** Si

Sistema binario

- **Cifre:** 2 – 0 e 1
- **Posizionale?** Si
- **Lettere?** No
- **Base:** 2
- Primo bit: **MSB** (Most Significant Bit)
- Ultimo bit: **LSB** (Least Significant Bit)

Se devo rappresentare un numero con un numero di bit più elevato rispetto a quelli a disposizione, allora **aggiungo zero all'inizio**

Somma

La somma tra due numeri binari è definita su tre elementi: **due addendi e il riporto** (carry). Per eseguire la somma, si effettuano operazioni di somma tra i bit di *pari ordine*, dal LSB al MSB.

Esempio: $010011 + 01000$

Riporto	1	0	0	1	1	
Primo addendo	0	1	0	0	1	1
Secondo addendo	0	1	0	0	0	1
Somma	1	0	0	1	0	0

Esempio:

Il rapporto si somma alla prima cifra, poi il risultato di questo al secondo.

Raggiunto il numero di bit a disposizione, l'ultimo rapporto viene semplicemente scartato

$$\begin{aligned} 1+1 &= 0 \quad \text{e} 1 \\ 1+0 &= 1 \\ 0+1 &= 1 \\ 0+0 &= 0 \end{aligned}$$

Sottrazione

La sottrazione funziona in modo analogo alla somma. È definita sempre su tre elementi: **minuendo, sottraendo e prestito** (borrow).

Il prestito funziona come il riporto nelle sottrazioni a colonna: nelle operazioni dove il risultato verrebbe negativo, viene richiesto un prestito dal bit superiore, proprio come nelle operazioni a colonna.

Il problema risale quando si arriva al MSB con un riporto; se lo spazio a disposizione non è abbastanza (e.g. 6 bit ma c'è bisogno di 7 bit), si va in **errore di overflow**; questo errore si presenta quando il computer va nell'impossibilità di rappresentare il risultato di un'operazione con il numero di cifre a disposizione.

Esempio: $010101 - 110100$

Prestito	-prestito					
Minuendo	0	1	0	1	0	1
Sottraendo	1	1	0	1	0	0
Differenza	-	0	0	0	0	1

In questo caso, ci sarebbe bisogno di un prestito, ma non posso
Bit finiti, quindi **overflow**

Rappresentazione dei numeri negativi

Con i numeri binari, non è possibile rappresentare i numeri con i segni. Per questo, viene usato un *trick*: viene adoperato il MSB di un numero per indicare che segno abbia. Questa si tratta di una convenzione ovviamente. Vediamo ora i tre metodi per rappresentare un numero negativo.

Modulo e segno

Supponendo di avere 1 byte, si usano i primi **7 bit** da destra per il valore assoluto del numero, e il **MSB** per indicare il segno: 1 se il numero è negativo, 0 se è positivo.

Esempio: $-4_{10} \rightarrow 10000100$, **MSB di segno**

Con n bit, è possibile rappresentare $[-(2^{n-1}-1), +2^{n-1}+1]_{10}$. Questa rappresentazione, però, risulta alquanto problematica per il numero zero, per il fatto che **esistono due rappresentazioni diverse dello zero**, il che è impossibile. Si ha 0000_2 , equivalente a "+0", e 1000_2 , equivalente a "-0".

Somma

Usando Modulo e Segno, bisogna innanzitutto confrontare i bit di segno dei due numeri.

- Se i bit di segno sono uguali:
 - o Il bit di segno risultante sarà il bit di segno dei due addendi
 - o La somma sarà eseguita bit a bit

- Possibilità di overflow
- Se i bit di segno sono diversi:
 - Viene confrontato i valori assoluti dei due addendi
 - Il bit di segno risultante sarà il bit di segno dell'addendo con valore assoluto maggiore
 - La differenza sarà eseguita bit a bit

Sottrazione

Usando Modulo e Segno, bisogna innanzitutto confrontare i bit di segno dei due numeri.

- Se i bit di segno sono uguali:
 - Il bit di segno risultante sarà uguale al bit di segno dell'operando a modulo maggiore
 - Il risultato avrà modulo pari al modulo della differenza dei moduli degli operandi.
- Se i bit di segno sono diversi:
 - Il bit di segno risultante sarà uguale al bit di segno del minuendo
 - Il risultato avrà modulo pari alla somma dei moduli dei due operandi
 - Possibilità di overflow

Complemento a 1

Questo metodo si basa sull'operazione di complemento, ovvero l'operazione che associa ad un bit il suo opposto.

- Se il numero è positivo, lo si converte in binario con il metodo tradizionale.
- Se il numero è negativo, basta convertire in binario il suo modulo, e poi eseguire l'operazione di complemento sulla codifica binaria effettuata.

Questo metodo presenta, comunque, lo stesso problema del metodo MS, ovvero la **duplicata rappresentazione dello zero**.

$$Es. 3_{10} = 0011_2 \quad -3_{10} = 0011 - \text{complemento} \rightarrow 1100_2$$

Complemento a 2

Il complemento a 2 è stato introdotto per risolvere il problema degli altri due metodi, ovvero la duplicata rappresentazione dello zero.

Se il numero è **positivo**, allora rimane invariato. Se invece è **negativo**:

- Effettuare complemento a 1 sul valore da codificare
- Somma +1 al risultato ottenuto con CA1

Con questo metodo, i valori negativi hanno MSB = 1, e dati n bit, si possono rappresentare $[-(2^{n-1}), +2^{n-1}]_{10}$.

Per il calcolo del complemento, abbiamo **tre metodi**:

1. Definizione di complemento alla base $CA2(X) = 2^n - x$
2. Calcolo CA1 + somma 1
 - a. $CA1(X) = (2^n - 1) - X$
 - b. $CA2(X) = CA1(X) + 1$
3. Regola pratica
 - a. Si parte da destra, si trascrivono tutti gli zero fino ad incontrare il primo uno, e si trascrive esso.
 - b. Si complementa a 1 per tutti i bit restanti.

$$Es. CA2 a 4 bit di -7$$

1. Definizione
 - a. $2^4 - 7 = 10000 - 111 = 1001(CA2)$ → si eleva a quanti bit sono a disposizione
2. Passaggio da CA1
 - a. $111_2 = 0111 - \text{complemento} \rightarrow 1000(CA1) - \text{somma} +1 \rightarrow 1001(CA2)$
3. Regola pratica
 - a. $111_2 = 0111(CA2) = 1001(CA2)$

Da CA2 a decimale

Se il numero è positivo (MSB = 0), allora si converte in base decimale usando il numero binario puro.

Se invece il numero è negativo (MSB = 1), allora si applica l'operazione di CA2 ottenendo la rappresentazione del corrispondente positivo, si converte il risultato come numero in binario puro, e si aggiunge il segno meno.

Somma

Per effettuare la somma con CA2:

- Viene effettuata la somma su tutti i bit degli addendi, segno compreso
- Eventuale riporto oltre al bit di segno viene scartato
- Nel caso che gli operandi siano di segno concorde, occorre verificare la presenza o meno di overflow
 - o L'overflow non si presenta quando si sommano operandi di segno opposto

Es. $1111 + 1111$ overflow

+3	0000	0011
+(-8)	1111	1000
-5	1111	1011

Sottrazione

La sottrazione non viene mai effettuata, ma invece la sottrazione viene trasformata in somma, applicando **A-B = A + (-B)**, ovvero $A-B=A+CA2(B)$.

Innanzitutto, bisogna verificare l'assenza di overflow. Non si ha overflow se gli operandi hanno segno discorde, mentre si ha overflow se gli operandi hanno segno concorde.

Gli operandi devono essere rappresentati con lo stesso numero di bit. Nell'ipotesi di avere X in CA2 su n bit, e di volerne ricavare la rappresentazione, sempre in CA2 su m bit, si attua l'**estensione**: si replica MSB negli m-n bit più a sinistra.

Operazione shift

Il sistema binario include un'altra operazione speciale, chiamata "shift", che consiste nello spostare verso destra o verso sinistra la posizione delle cifre di un numero, espresso in una base qualsiasi, inserendo uno zero nelle posizioni lasciate libere. Verso **sinistra** equivale a *moltiplicare il numero per la base*, mentre verso **destra** equivale a dividere il numero per la base.

Left

Equivale alla moltiplicazione. L'ultimo numero fa uno shift a sinistra, e il segno si replica.

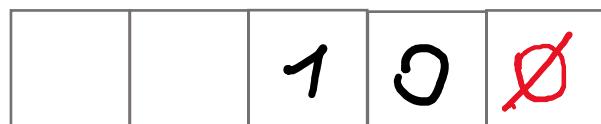


Scarico del bit: se il MSB è uno, allora va in overflow; se non va, allora si rimuove e si aggiunge uno zero alla fine.

Con il CA2, se il nuovo MSB è diverso dal precedente, allora va in **overflow**. Si shifta a left, e si introduce lo zero in coda, in fondo a destra.

Right

Equivale alla divisione. Elimino l'ultimo numero, e introduco lo zero prima del numero.



Si elimina l'ultimo numero, e si introduce lo zero prima del numero.

Con il CA2, si scarta il bit a destra, e si continua a replicare il segno. Si continua a replicare il MSB.

Sistema totale

- **Cifre:** 8, da 0 a 7
- **Posizionale?** Sì
- **Lettere?** No
- **Base:** 8

Sistema esadecimale

- **Cifre:** 16 – da 0 a 9 + da A a F
- **Posizionale?** Sì
- **Lettere?** Sì
- **Base:** 16
- **Notazioni particolari:** h 0x(numero)
- **Conviene?** È compatto, 4 numeri in base binaria sono raggruppati in un singolo numero esadecimale

Cifre:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9
A (10), B (11), C (12), D (13), E (14), F (15)

Conversione

Per convertire da un sistema numerico a un altro, basta rappresentare, come prima, le varie cifre del numero, **moltiplicate seguendo le regole viste precedentemente**.

$$Es. 1010_2 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 10_{10}$$

Per convertire, invece, **da base 10 a qualsiasi base s**, occorre:

- Dividere N per la base r fino a quando l'ultimo quoziente è minore della base stessa
- Prendere l'ultimo quoziente, e i resti delle divisioni. Procedendo dall'ultimo resto, li scriviamo da sinistra verso destra.

Es. $12_{10} \rightarrow$

$12 : 2 = 6$	resto 0	↑	si prendono i resti dal fondo verso l'alto
$6 : 2 = 3$	resto 0		
$3 : 2 = 1$	resto 1		
$1 : 2 = 0$	resto 1		

non posso più andare avanti

↳ divisione sempre per la base data

decimale → hex

↳ dividere per 2

↳ binario → hex

↳ 4 bit alla volta

Come metodo più veloce, possiamo anche:

- Approssimare il numero che abbiamo a una somma delle potenze della base; questi numeri devono essere il più vicino possibile.
- Successivamente, trasformare queste basi in **potenze**, e inserire zero nei "posti delle potenze mancanti"

$$Es. 35_{10} = 32 + 2 + 1 = 2^5 + 2^1 + 2^0 = 1 * 2^5 + 0 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 100011$$

Nel caso di una rappresentazione binaria, si userà la base 2; nel caso delle altre rappresentazioni, si userà la base relativa alla rappresentazione che si vuole.

Rappresentazione di numeri reali

Esistono due metodi per rappresentare un numero reale: rappresentarlo in **virgola fissa**, o in **virgola mobile**.

Virgola fissa

Un sistema in **virgola fissa**:

- Riserva un numero fisso di bit per parte intera e parte frazionaria
- La posizione della virgola decimale è **implicita**
- La posizione della virgola decimale è uguale in tutti i numeri
 - o $I < N$ bit per rappresentare la parte intera del numero
 - o $D = N - I$ bit per rappresentare la parte decimale del numero
 - o Intervallo di numeri interi rappresentabili: $[0, 2^I - 1]$
 - o Intervallo di numeri decimali rappresentabili: $[0, 2^D - 1]$
- Possiamo decidere noi **dove far "cadere" la virgola, basta che poi sia fissa.**
 - o Avendo per esempio 8 bit, si può dedicare: *1 bit al segno, 3 bit alla parte intera, 4 bit alla parte decimale*, o scegliere un'altra combinazione.
 - o La scelta quindi varia se si vuole rappresentare più numeri interi, o invece numeri decimali più precisi.

Signed virgola fissa

Un sistema a signed fixed point ci permette di segnare numeri negativi o positivi, ovvero con il segno.

- Riserva un bit per il segno
- Abbiamo $I < (N-1)$ bit per la parte intera
- $D = N - (I+1)$ bit per la parte decimale
- Intervallo di numeri interi rappresentabili: $[-2^{I-1} - 1, 2^{I-1} - 1]$
- Intervallo di numeri decimali rappresentabili: $[0, 2^D - 1]$

Virgola mobile

La rappresentazione a virgola mobile ci permette di usare **1 bit** per il segno, un numero variabile di bit per l'**esponente**, e un numero variabile di bit per la **mantissa** (parte frazionaria). Proprio per questo motivo, è chiamato "virgola mobile", ovvero perché possiamo muovere come vogliamo noi la virgola, ed inoltre rappresentare un range di numeri molto più ampi.

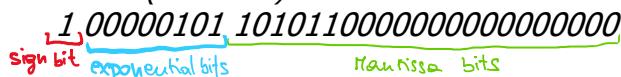
$$\text{Mantissa} * \text{base}^{\text{esponente}}$$

Esistono due forme di rappresentazione della virgola mobile:

- **Non-normalizzata:** la prima cifra della mantissa è uguale a zero
- **Normalizzata:** la prima cifra della mantissa è diversa da zero

Esempio: -53.5 in binario:

-53.5 — rappresentazione binaria $\rightarrow (-110101.1)_2$ — notazione scientifica $\rightarrow (-1.101011) * 2^5$


 1 00000101 10101100000000000000
 sign bit exponential bits mantissa bits

IEEE 754

Per la notazione a virgola mobile, si è visto che è reso necessario definire uno standard per la rappresentazione dei numeri in virgola mobile, per definire la semantica delle istruzioni. IEEE definisce lo **Standard for Binary Floating Arithmetic**, noto anche come **IEEE 754-1985** (sostituito poi nel 2008 dal IEEE 754-2008).

È un formato *non proprietario*, ossia non dipendente dall'architettura del calcolatore. Questo standard definisce il formato per la rappresentazione dei numeri in virgola mobile, compreso lo zero, i numeri denormalizzati, gli infiniti e i NaN, e un set di operazioni effettuabili su questi.

- Per i numeri a **32 bit**:
 - o 1 (indice 31) bit di segno, 8 (indice 30-23) bit di esponente, 23 (indice 22-0) bit di mantissa
 - o Numero:

$$(-1)^{\text{segno}} * 2^E * M$$
 - o Se l'esponente è negativo: **sottrarre 127 al risultato ottenuto**
- Per i numeri a **64 bit**:
 - o 1 (indice 63) bit di segno, 8 (indice 62-51) bit di esponente, 51 (50-0) bit di mantissa

Errore assoluto e relativo

Rappresentando un numero reale n in virgola mobile, si commette un errore di approssimazione. In realtà, viene rappresentato un numero razionale n' con un numero limitato di cifre significative. Per fare fronte a questo, ci sono due tipi di errori:

- **Errore assoluto:** $E_A = N - N'$
 - o **Ordine di grandezza** dipende dal numero di cifre significative e dall'ordine di grandezza del numero.
- **Errore relativo:** $E_R = E_A/N = (N - N')/N$ (indica "quanto è grave l'errore che sto commettendo rispetto all'originale").
 - o **Ordine di grandezza** dipende solo dal numero di cifre significative

Errore assoluto

Data una cifra n :

1. Prendo il numero approssimato più vicino
2. $E_A = n - n_{\text{approssimato}}$

Errore relativo

Dato un intervallo $(n_{\text{approx}} - n)$:

1. Misuro l'ampiezza dell'intervallo
2. Ampiezza/n $\rightarrow 7 \text{ bit} = 128$

Altre rappresentazioni

Possiamo non solo rappresentare numeri, ma anche caratteri, suoni, etc... concentrando sui caratteri primariamente, possiamo associare a ogni carattere un numero, tramite quattro standard:

- **ASCII standard:** 1 carattere è rappresentato con 7 bit, per un totale di 128 simboli
 - o 26 lettere maiuscole + 26 lettere minuscole
 - o 10 cifre decimali
 - o Segni di interpunkzione
 - o Caratteri di controllo
- **ASCII estesa:** 1 carattere è rappresentato con 8 bit, rappresentabili fino a 256 simboli.
- **Unicode:** 1 carattere è rappresentato con un numero maggiore di bit, che varia tra 8 e 32 bit per carattere, a seconda di quanto "pesi" il carattere
- **UTF:** intermedio

Circuiti logici

I **circuiti logici** sono realizzati come circuiti integrati realizzati su chip di silicio. Porte e fili sono depositati su un chip di silicio, inseriti in un package e collegati all'esterno con un certo insieme di pin. I circuiti integrati si distinguono per grado di integrazione.

Si parla di **segnale basso** quando il voltaggio è compreso tra zero e uno, mentre si parla di **segnale alto** quando il voltaggio è più di uno.

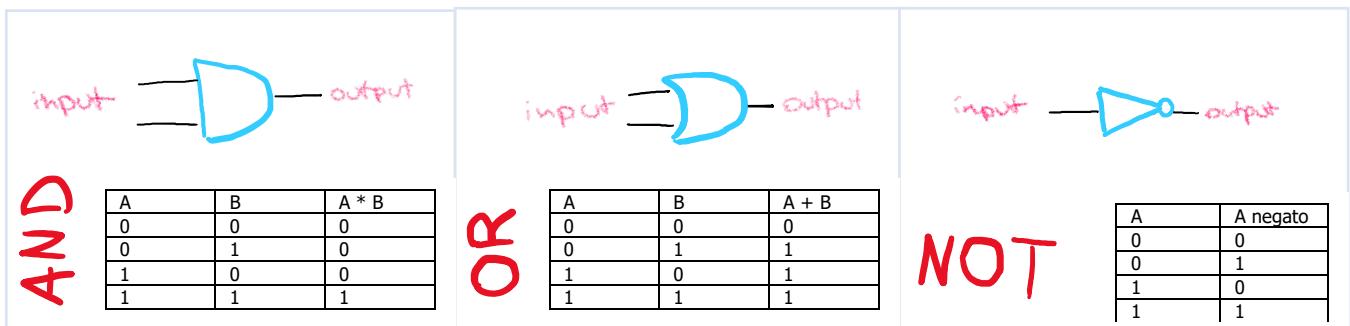
Un **circuito combinatorio** è quel circuito il cui lo stato delle uscite dipende solo dalla funzione logica applicata allo stato istantaneo delle sue entrate; dati degli input, elabora e da degli output.

Un **circuito sequenziale** è quel circuito il cui stato delle uscite non dipende solo dalla funzione logica applicata ai suoi ingressi, ma anche sulla base di valori pregressi collocati in memoria; dati degli input, elabora e considera lo stato attuale del sistema, per poi produrre degli output.

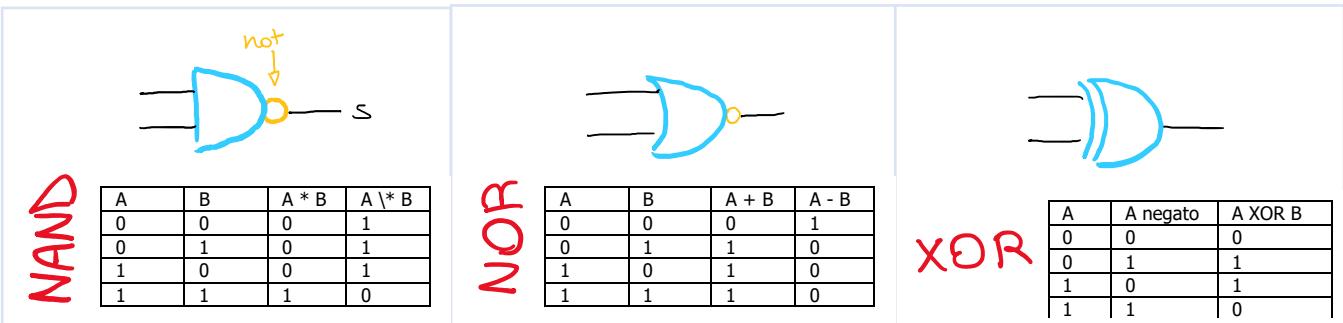
Porte logiche

Le porte logiche sono i componenti elettronici che permettono di svolgere le operazioni logiche primitive e derivate. È un circuito elettronico che, dati dei segnali **0 e 1 in input**, produce un segnale in output ottenuto effettuando una operazione booleana sugli ingressi.

Le **porte logiche fondamentali** sono AND, OR, e NOT, che svolgono le funzioni principali legate all'algebra booleana.



Le **porte logiche derivate** sono invece NAND, NOR e XOR.



Queste porte accettano 2 input e un output; si possono però combinare molteplici porte logiche per avere più *opzioni*.

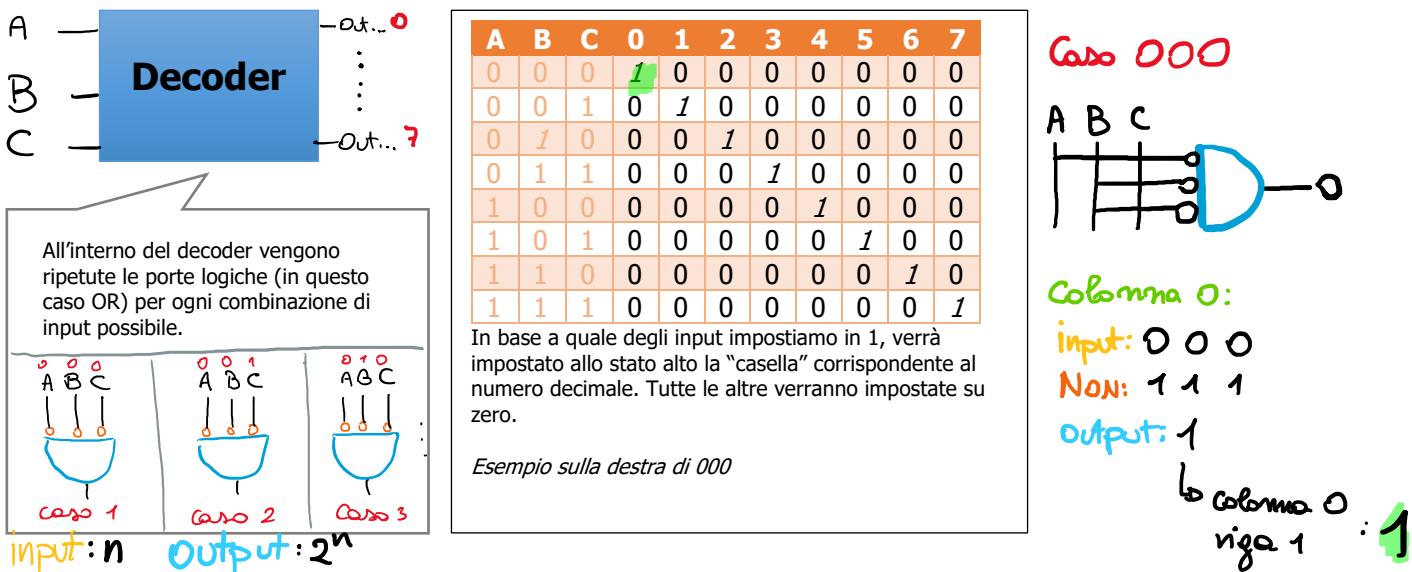
I circuiti notevoli

Un circuito notevole è un circuito che ha proprietà e caratteristiche distintive, che lo rendono utile per determinati scopi o applicazioni. Sono di solito usati come **blocchi di costruzione per creare circuiti più complessi e avanzati**.

Decoder

È un componente elettronico caratterizzato dall'avere n ingressi e 2^n uscite. Il suo scopo è **impostare allo stato alto l'uscita corrispondente alla conversione in base 10 della codifica binaria a n bit in input**, e di impostare allo stato basso tutte le altre.

Questo significa che, a seconda degli input dati, verrà impostato su **1** la "casella" corrispondente al numero decimale corrispondente a quello, mentre a tutte le altre colonne viene impostato il valore **0**.



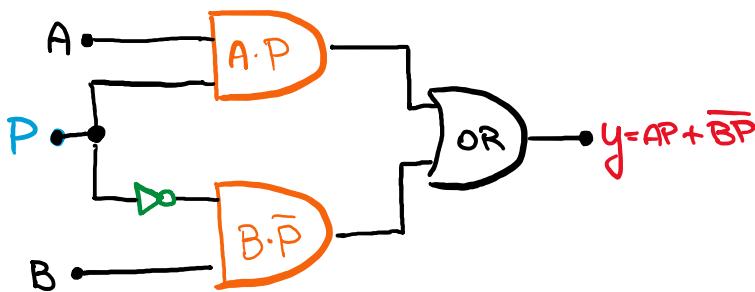
Gli n input sono interpretati come un numero *unsigned*. Se questo numero rappresenta il numero i , allora **solo il bit in output di indice i verrà posto ad 1**, mentre tutti gli altri verranno posti a 0.

Multiplexor

Un multiplexor, o selettore, è un componente elettronico con **2ⁿ entrate principali, n entrate di controllo, e 1 uscita**. Consente di selezionare uno tra diversi ingressi e di indirizzarlo all'uscita del circuito, usando un segnale di selezione.

Il valore del selettore determina quale input diviene output.

Se un multiplexor ha più porte entranti, allora ha bisogno di **log₂n selettori**.



A	B	P	$A \cdot P$	$B \cdot \bar{P}$	OR	Y
0	0	1	0	0	0	0
1	0	1	1	0	1	1
1	1	1	1	0	1	1
0	1	1	0	0	0	0

Un multiplexor funziona in base a quale selettore si imposta, ovvero la **P**. In questo caso, impostando il selettore su 1, possiamo vedere come **il risultato sia lo stesso di A**.

Quindi, in questo caso, quando il selettore è su 1, allora il risultato sarà sempre su 1.

Logiche a due livelli

La logica a due livelli, o combinatoria, è un tipo di circuito digitale in cui l'uscita del circuito **dipende solo dal valore degli ingressi del circuito in un determinato momento**. L'uscita del circuito viene quindi generata istantaneamente, in base ai valori degli ingressi senza dipendere da alcuno stato interno.

Possiamo quindi creare logiche a due livelli, utilizzando le porte logiche AND, OR e NOT. Possiamo quindi creare:

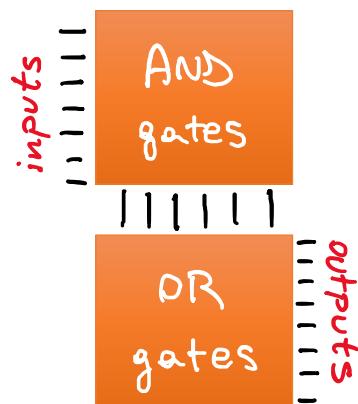
- *Somma di prodotti*: somma logica (OR) di prodotti (AND)
- *Prodotto di somme*: prodotto (AND) di somme (OR)

Come visto in questi due esempi, combinando assieme operazioni più semplici possiamo infine avere operazioni più complesse.

Programmable Logic Array

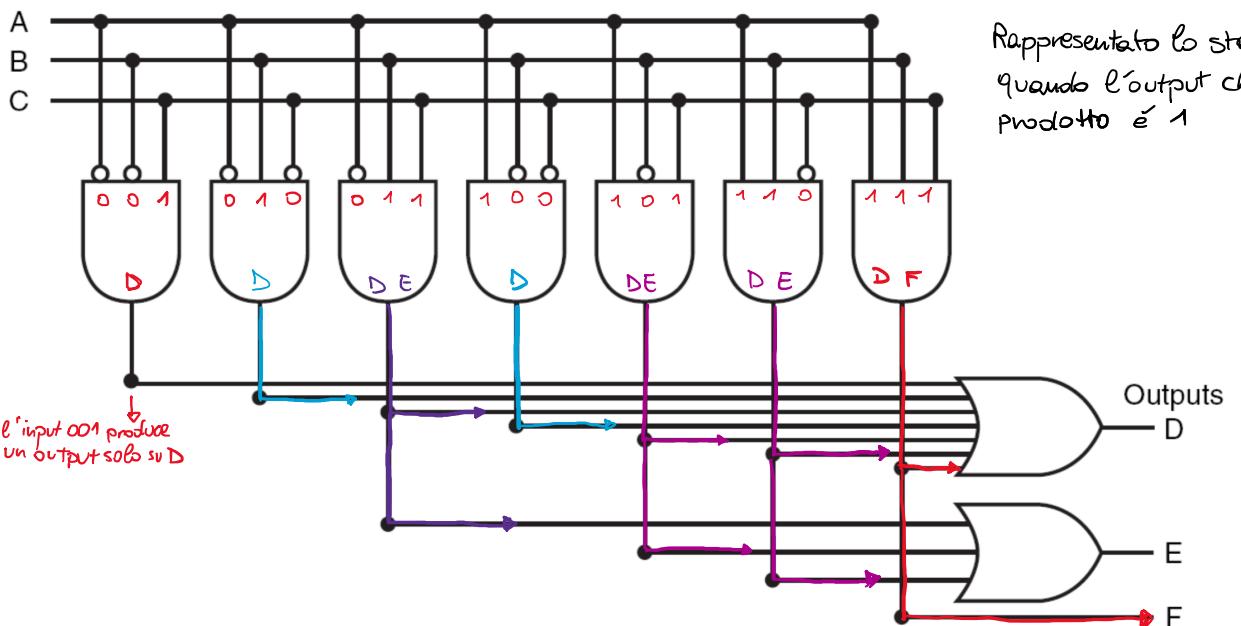
La somma di prodotti detta prima corrisponde ad un'implementazione chiamata PLA, dove abbiamo:

- Un insieme di **input**
- I corrispondenti input complementari mediante inverter, per poter gestire più uscite
- Una logica a due stage
 - o **Primo stage**: un array di porte logiche AND (prodotto)
 - o **Secondo stage**: un array di porte logiche OR (somma)



Possiamo vedere un esempio:

Inputs



Array di elementi logici

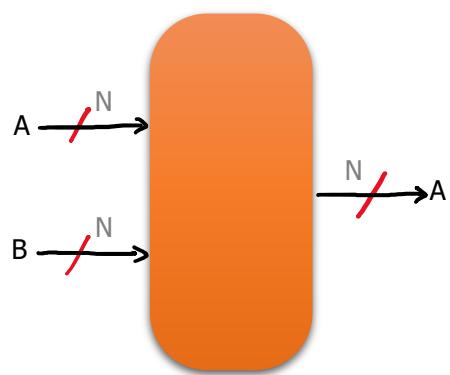
La maggior parte delle operazioni vengono svolte su 32 bit, mettendo in luce la necessità di creare **array di elementi logici**.

Un **bus** è una collezione di linee di input, che verranno trattate come un singolo segnale.

Prendendo un qualsiasi circuito notevole, possiamo per esempio condensare **n righe di input e output** in una sola riga, aggiungendoci **n** sulla freccia che rappresenta. I segnali possono essere trasmessi in parallelo o in serie.

Rappresentando il circuito in questo modo, si traduce come in una cascata di circuiti notevoli che lavorano **linea per linea** (es. Linea A1, B1, C1 – A2, B2, C2, etc...).

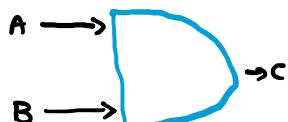
Di solito, è un insieme di circuiti logici che rappresentano un insieme di bit di informazione, tipicamente organizzati in una griglia o una matrice.



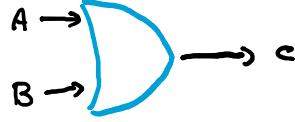
ALU (Arithmetic Logical Unit)

L'ALU è la parte del processore che svolge le operazioni aritmetico-logiche. È un insieme di circuiti combinatori che implementa operazioni aritmetiche, e operazioni logiche. Sono costituiti da dei blocchi base:

AND ($c = a * b$)



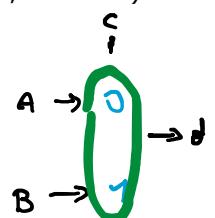
OR ($c = a + b$)



Inverter ($c = \bar{a}$)



Multiplexor (if $d == 0$, $c = a$; else $c = b$)



A	B	$C = a * b$
0	0	0
0	1	0
1	0	0
1	1	1

A	B	$C = a + b$
0	0	0
0	1	1
1	0	1
1	1	1

A	$C = A$
0	1
1	0

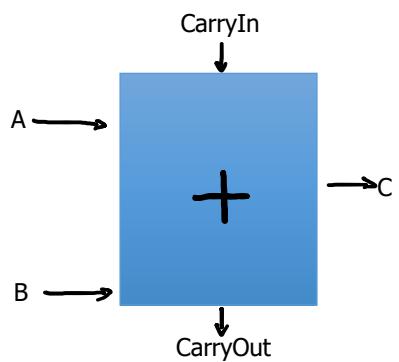
C	d
0	A
1	B

Addizionatore

È detto anche "sommatore", è un circuito logico che esegue l'operazione di somma su due numeri binari. L'addizionatore può essere progettato per operare su bit singoli, o su parole binarie composte da più bit. Un addizionatore a più bit è composto da più addizionatore a 1 bit **collegati a cascata**.

L'addizionatore ha anche due "Carry", che si riferiscono alla propagazione di un bit di carry attraverso una sequenza di operazioni di addizione o sottrazione in una ALU. Questi sono **CarryIn** e **CarryOut**.

- CarryIn: segnale di ingresso che indica se un bit di carry deve essere propagato in una determinata operazione di addizione o sottrazione
- CarryOut: segnale di uscita che indica se il risultato dell'operazione di addizione o sottrazione ha generato un bit di riporto.



Il CarryOut prodotto dal primo addizionatore viene usato come CarryIn dal secondo addizionatore, e così via, fino a quando tutti i bit vengono sommati. L'uscita è quindi composta dalla **somma dei bit corrispondenti**, e dal **CarryOut prodotto dall'ultimo addizionatore**.

Possiamo fare una tabella di verità per esempio di come funzionerebbe un addizionatore.

A	B	CarryIn	CarryOut	Somma
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

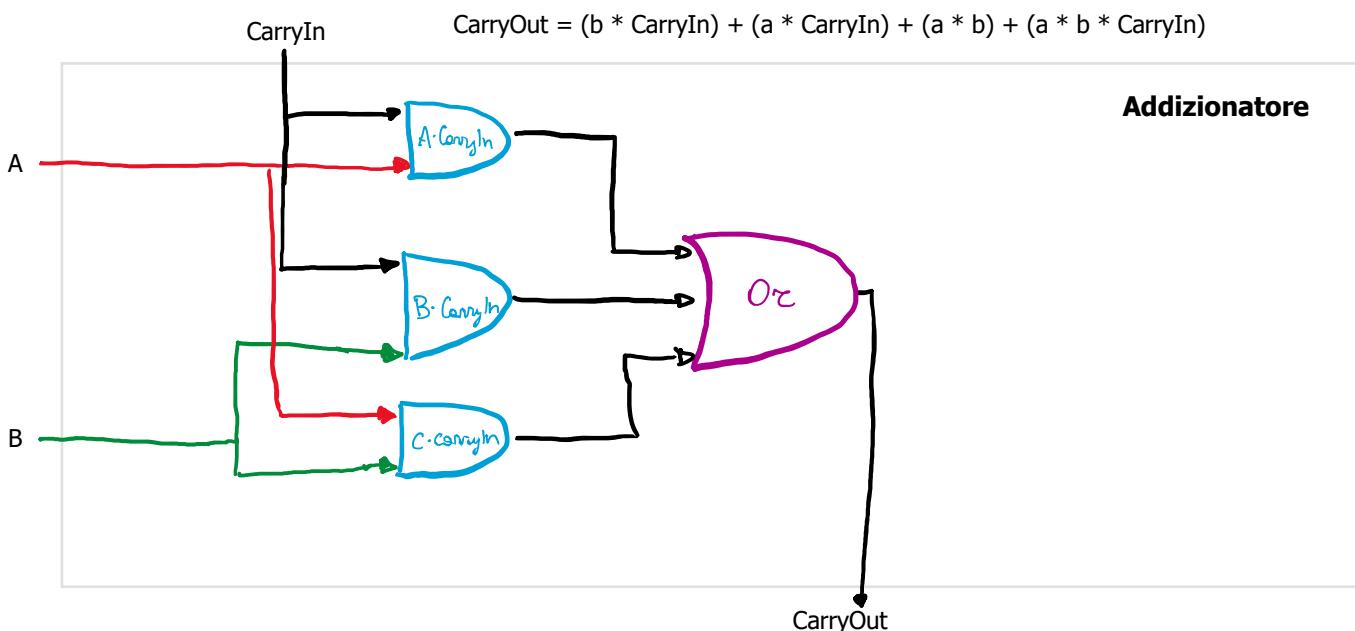
Operazione effettuata:
A + B + CarryIn

CarryOut: riporto

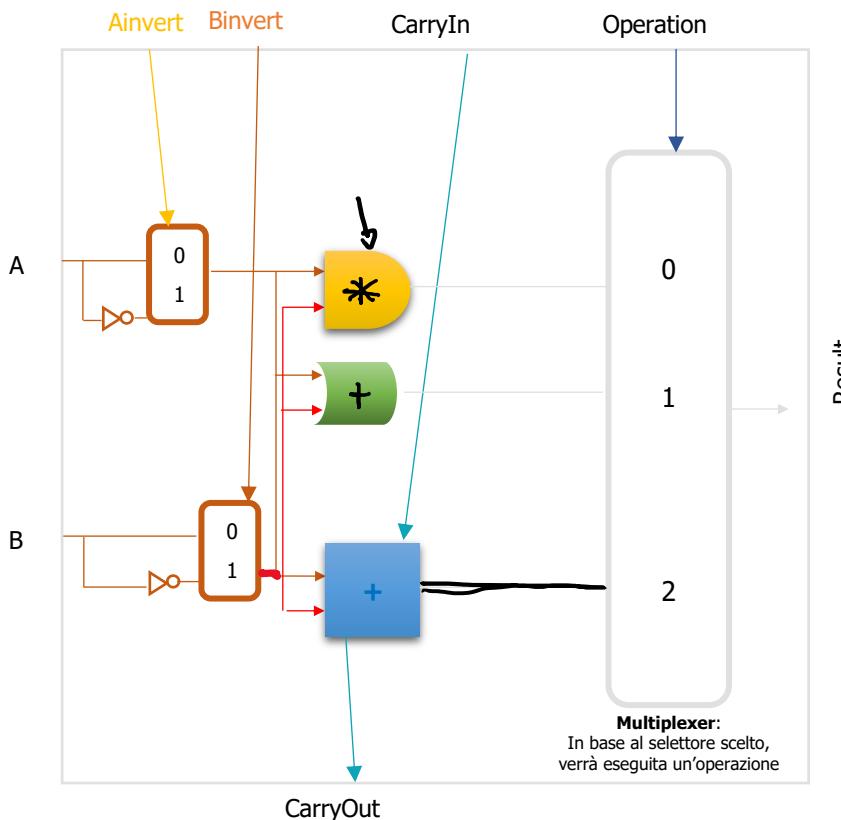
$1+1=0 \text{ } 2 \rightarrow$
CarryOut

Il CarryIn cambia per far vedere ogni esempio

Possiamo rappresentare questa tabella di verità sotto forma di circuito con una ALU ad 1 bit.



Ora che abbiamo un addizionatore sotto forma di un circuito, possiamo inserirlo all'interno di un circuito esistente. Come esempio, prendiamo **un circuito che implementa un multiplexor per sapere quale operazione verrà eseguita**.



CarryIn = 1

Aggiungendo $\triangleright O$ alla **B un not e un multiplexor**, è possibile scegliere se vogliamo che B sia positiva o negativa (0 se positiva, 1 se negativa), e quindi ottenere la sottrazione di numeri.

$$A - B = A + (B + 1)$$

Legge di De Morgan

Aggiungendo $\triangleright O$ alla **A un not e un multiplexor**, è possibile scegliere se vogliamo che A sia positiva o negativa (0 se positiva, 1 se negativa), e quindi ottenere la sottrazione di numeri, implementando effettivamente le leggi di De Morgan:

$$\begin{aligned} NOT(a \text{ OR } b) &= NOT a \text{ AND } NOT b \rightarrow NOT(a+b) = NOT a * NOT b \\ NOT(a \text{ AND } b) &= NOT a \text{ or } NOT b \rightarrow NOT(a * b) = NOT a + NOT b \end{aligned}$$

Operazioni di confronto

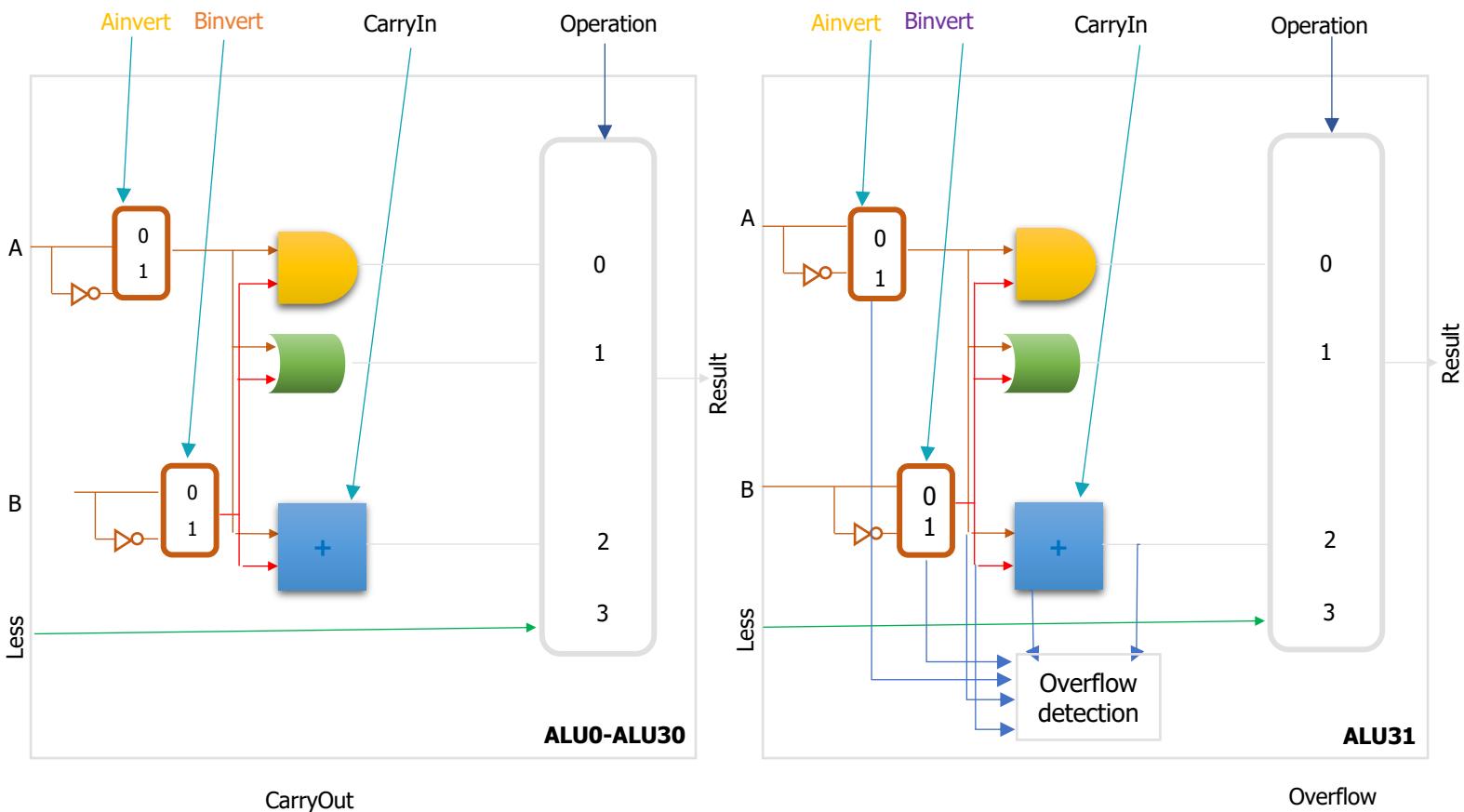
Le ALU possono avere speciali "pezzi di circuiti" che confronta due valori e restituisce un risultato basato sulla relazione tra i valori. Di solito, viene solitamente eseguita su numeri interi memorizzati nei registri della CPU.

SLT – Set on Less Than

È un'operazione che ha come risultato se, dati due numeri a e b:

- **1** se $a < b$ ($a-b < 0$)
- **0** se $a > b$ ($a - b > 0$)

Per poter eseguire questa istruzione, si devono poter azzerare tutti i bit dal bit-1 al bit-31, ed assegnare al bit-0 il valore del risultato.



Test di overflow

Ci possono essere dei casi dove il circuito può andare in overflow. In questo caso, abbiamo due casi:

- $(A-B) > 0$ e bit-31 di $(A-B) = 1$ $\text{bit-31}(A) = 0, \text{bit-31}(B) = 1 \quad \text{bit-31}(A-B) = 1$
- $(A-B) < 0$ e bit-31 di $(A-B) = 0$ $\text{bit-31}(A) = 1, \text{bit-31}(B) = 0 \quad \text{bit-31}(A-B) = 0$

Per questo, si esegue un controllo di overflow, che corrisponde a: $\text{overflow} = \underline{a} * b * \underline{\text{Result}} + a * \underline{b} * \underline{\text{Result}}$

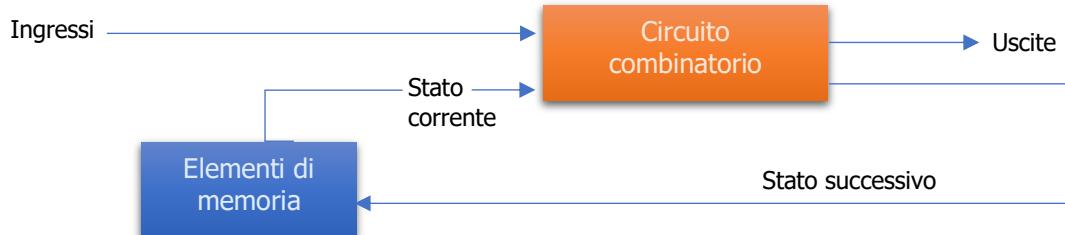
Circuiti sequenziali

I circuiti combinatori visti in precedenza sono in grado di calcolare funzioni che dipendono solo dai dati in input.

I **circuiti sequenziali**, invece, sono circuiti che dipendono da uno **stato**, ovvero delle informazioni memorizzate in elementi di memoria interni. L'uscita quindi non dipende solo dagli stati di ingresso attuali, ma anche dallo stato interno del circuito, che è **influenzato dagli stati di ingresso precedenti**.

Avendo una memoria interna, consente loro di "ricordare" gli stati di ingresso precedenti, e utilizzarli per determinare l'uscita attuale.

Un circuito di questo tipo ha, in ogni dato istante, uno stato determinato dai bit memorizzati. È quindi necessario un elemento di memoria per **memorizzare lo stato**; questo elemento si chiama **latch**.

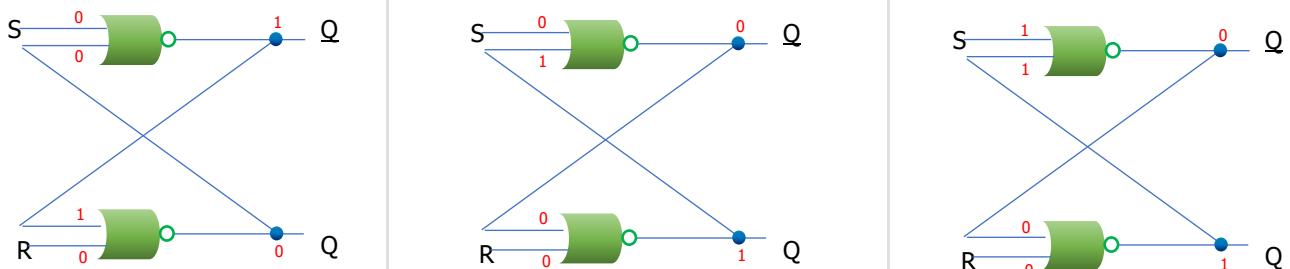
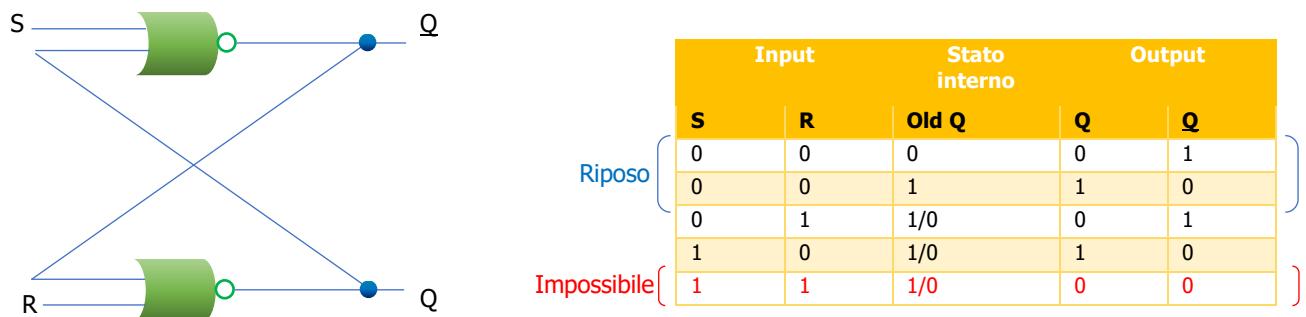


Lo stato corrente viene usato come input assieme agli ingressi nel circuito combinatorio

S-R latch (Set-reset)

Il Set-Reset latch è un circuito, composto da 2 porte NOR concatenate. In questo circuito:

- L'output del circuito diventa stabile dopo un certo intervallo di tempo che dipende dal numero di porte traversate.
- Bisogna evitare che durante questo intervallo, gli output intermedi del circuito vengano memorizzati.
- È **bistabile**: è un circuito che ha due stati stabili. *Mantiene lo stesso livello di tensione all'uscita finché non riceve un impulso esterno che lo fa cambiare.*
 - Potrebbe essere usato insieme a un clock per far cambiare lo stato, ma sarebbe un **latch sincrono**, che cambia il suo stato solo quando riceve un segnale di abilitazione insieme agli ingressi.
 - Un latch **asincrono** cambia il suo stato appena gli ingressi S e R variano.
- Ha due ingressi: **S** (Set) e **R** (Reset), che determinano lo stato dell'uscita. L'ingresso S imposta l'uscita a 1, mentre l'ingresso R Reset a 0.
- **Se entrambi gli ingressi sono attivi, il circuito è in uno stato indefinito.**

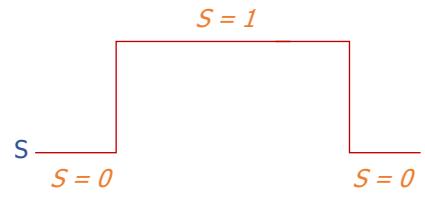


Uscita del latch

Come visto prima, un S-R latch (a porte NOR) presenta due uscite, Q e \bar{Q} . Possiamo applicare dei segnali ad S e R per ottenere dei risultati, **evidenziati nella tabella prima presentata**.

Per capire come vengono attribuiti i valori S ed R, e quindi come si apriranno i latch, possiamo usare un clock sia per S che per R (che verrà presentato al prossimo paragrafo).

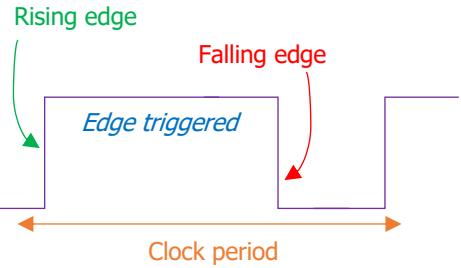
- Se un segnale è allo **stato alto**, allora il valore collegato a quel clock sarà **1**
- Se un segnale è allo **stato basso**, allora il valore collegato a quel clock sarà **0**



Clock

Il clock è un segnale periodico che viene utilizzato per sincronizzare le operazioni all'interno del circuito. Determina il momento in cui i **dati vengono letti o scritti dai registri**, e quando i segnali di controllo vengono attivati.

Viene quindi usato per **abilitare la scrittura nei latch**, e determina il ritmo dei calcoli e delle relative operazioni di memorizzazione. Usando questo, il circuito diventa **sincrono**.



È generalmente una forma di onda rettangolare, con un'alternanza periodica tra **alto** e **basso**. Il bordo di salita, o di discesa, del segnale di clock viene spesso utilizzato per attivare le operazioni del circuito.

Consente di **sincronizzare le operazioni del circuito**, evitando problemi come il "metastability", ovvero una condizione in cui il *segnale di uscita da un registro è instabile e può assumere valori intermedi*. Per questo, bisogna scegliere un periodo abbastanza grande, per **assicurare la stabilità degli output**.

Il cambiamento di stato *avviene sui "gradini" di questo clock*, e possiamo appunto scegliere se lavorare quando il clock è allo stato **edge triggered**, o allo stato 0.

D-Latch

Il D-Latch è un latch sincronizzato con il clock; in questo modo, si garantisce che il latch cambi solo in certi momenti.

Ha due ingressi, detto anche **latch** o **flip-flop**, ed è utilizzato per memorizzare **un bit di informazione** (0 o 1), e mantenere lo stato di uscita finchè non viene modificato da un segnale di ingresso.

1. Quando il segnale di controllo, il "clock", è alto, l'ingresso D viene memorizzato nella memoria del latch, l'uscita rimane costante anche se l'ingresso D cambia.
 - a. Il valore di D si propaga quasi **immediatamente** all'uscita Q
 - b. Anche le eventuali variazioni di D si propagano quasi immediatamente, con il risultato che Q può variare più volte durante lo stato asserted.
2. Quando il segnale di controllo è *basso*, il latch si disabilita e mantiene l'ultimo valore di uscita memorizzato
3. S = 0 e R = 0 → mantenuto il valore precedentemente memorizzato.
 - a. Solo in questo stato si stabilizza

Il segnale **D**, ottenuto come output di un circuito combinatorio, deve:

- Essere già stabile, quando il Clock diventa "asserted"
- Rimanere stabile, per tutta la durata del livello alto di Clock (setup time)
- Rimanere stabile per un altro periodo di tempo per evitare malfunzionamenti (hold time)

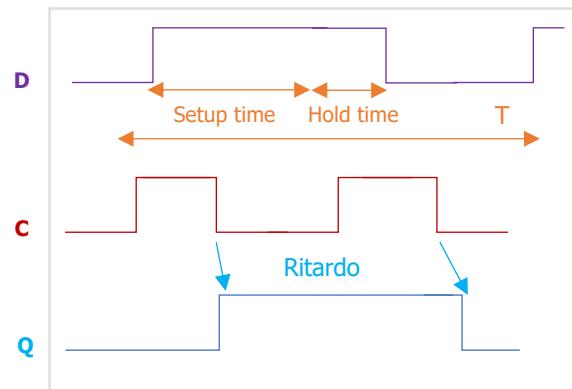
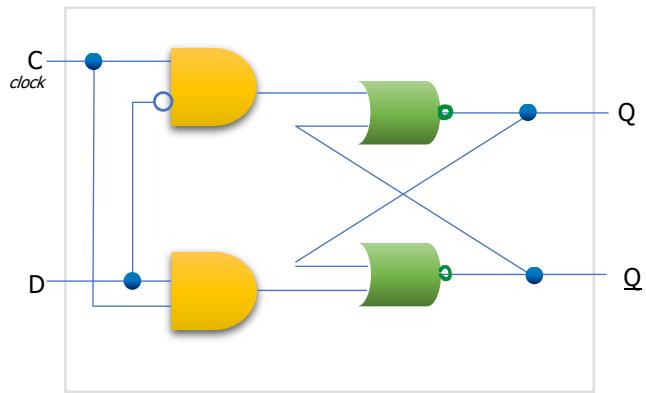
L'elemento di memoria è usato sia come input che come output nello stesso ciclo di clock; il segnale di clock rimane alto per molto tempo. In questo caso, potrebbe succedere che nel circuito **entri un segnale che fa in modo che il circuito non segni il segnale corretto**; rischiamo quello che è chiamato un **segnale "sporco"**.

L'output del precedente ciclo di clock viene usato nel seguente ciclo di clock.

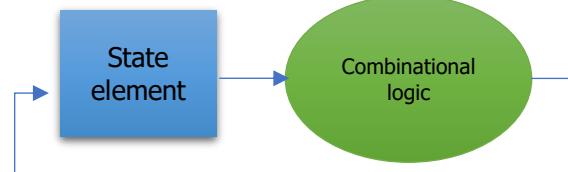
Flip-Flop

Finora, questi metodi sono detti **level triggered**, ovvero che avvengono nel livello alto o basso del clock. Esiste invece una metodologia **edge triggered**, che avviene sul fronte di salita o discesa del clock. Con questo metodo, *la memorizzazione avviene istantaneamente*, e l'eventuale segnale di ritorno sporco non fa in tempo ad arrivare a causa dell'istantaneità della memorizzazione.

In un circuito edge-triggered, il flip-flop (o latch) viene attivato dal fronte di salita o discesa del segnale di clock. In questo modo, il segnale di clock viene utilizzato come un "segnale di campionamento" per campionare l'ingresso del circuito in un determinato istante di tempo. Il segnale di uscita del flip-flop viene quindi mantenuto costante fino al successivo fronte di salita o discesa del segnale di clock, momento in cui il flip-flop viene attivato nuovamente.



Memoria usata come input e output



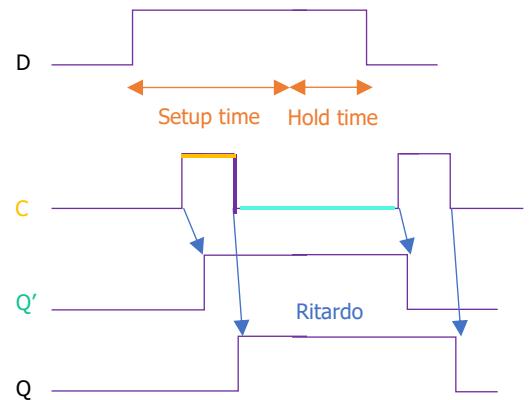
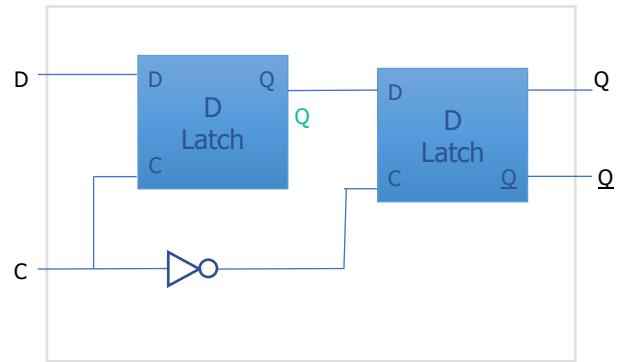
In un circuito level-triggered, invece, il segnale di clock viene utilizzato come un segnale di abilitazione che attiva il circuito in presenza di un segnale logico a livello alto o basso. In questo caso, il segnale di uscita del flip-flop rimane costante fintanto che il segnale di abilitazione rimane attivo.

Un flip-flop è un dispositivo di memoria digitale a circuito di elementi logici che, oltre a conservare un bit di dati, può essere utilizzato come elemento di sincronizzazione in circuiti sequenziali. I flip-flop possono **registrare i livelli di tensione**, consentendo così il supporto persistente del dato e la possibilità di una sequenza predefinita. Il flusso del segnale in un circuito sequenziale è regolato da una serie di **cambi di stato, o flip-flop**, i quali cambiano quando un certo evento si verifica.

Uno di questi tipi è il **D Flip-Flop**, che è usabile come input e output durante lo stesso ciclo di clock. Viene realizzato ponendo in serie 2 D-latch: il primo viene detto "master", il secondo viene detto invece "slave".

Immaginando di lavorare sul fronte di discesa:

1. Il primo latch è aperto e pronto per memorizzare D
 - a. Il valore memorizzato Q' fluisce fuori, ma il secondo latch è chiuso
2. Quando il segnale di clock scende, il secondo latch viene aperto per memorizzare il valore di Q'
3. Il secondo latch è aperto, memorizza D e fa fluire il nuovo valore Q nel circuito a valle. Il primo latch è chiuso, e non memorizza niente.



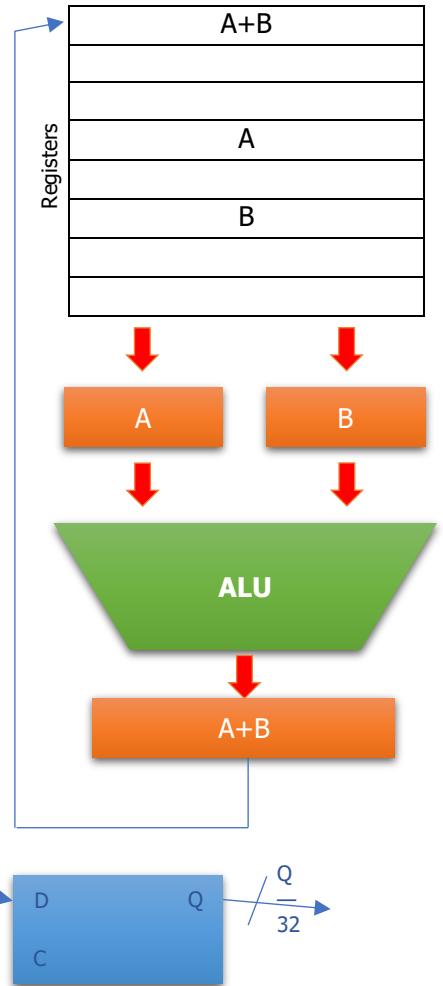
Datapath

Il datapath ci permette di gestire, oltre alle ALU, anche i registri di memoria. È un insieme di unità di calcolo, come ad esempio le unità di elaborazione, i registri e i moltiplicatori necessari per l'esecuzione delle istruzioni nella CPU.

Quando la ALU dovrà eseguire un'operazione, prenderà da un elemento chiamato **register file** quali saranno gli *indirizzi di memoria* dove sono contenuti i dati con cui deve eseguire l'operazione. Successivamente, questo risultato verrà salvato nella **ALU output register**, e successivamente salvato in un registro. Il passaggio di due operandi attraverso la ALU e la memorizzazione del risultato in un nuovo registro viene detto "ciclo di data path", ma questo è detto di qualsiasi sequenza di operazioni che si verificano all'interno del processore. L'obiettivo è *garantire che ogni istruzione venga eseguita in modo corretto ed efficiente usando le risorse disponibili all'interno del processore*.

Il clock non entra direttamente nei vari flip-flop, ma viene messo in **AND** con un segnale di controllo "Write". Questo segnale determina se, in corrispondenza del fronte di discesa del clock, il valore D debba o meno essere memorizzato nel registro.

- Se $write = 0 \rightarrow$ read register1, read register2 \rightarrow read data1, read data2
- Se $write = 1 \rightarrow$ write register \rightarrow write data
 - o Questo viene eseguito **solo sul fronte alto e se l'ALU ha fatto un'operazione**.



Register file

Un **register file** è un'area di memoria che contiene un insieme di registri a disposizione del processore. Questi sono *piccole aree di memoria interne usate per l'archiviazione temporanea dei dati*. Ogni registro ha un identificativo univoco, noto come **numero di registro**, che può essere acceduto e manipolato.

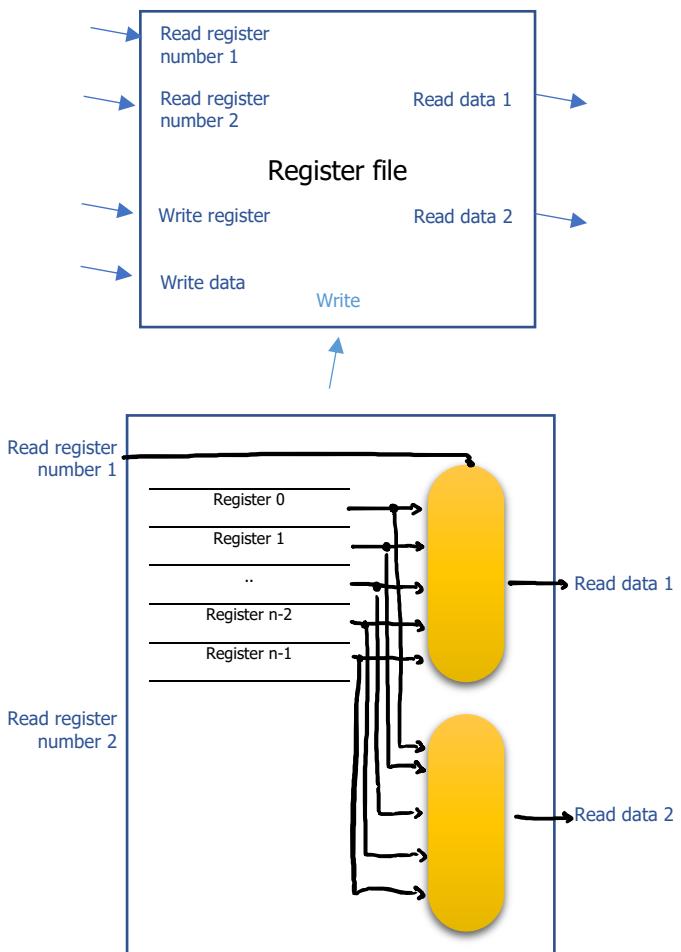
Un registro è costituito da **n flip-flop**; il register file permette inoltre la lettura di 2 registri, e la scrittura di uno.

Lettura

Utilizza due segnali che indicano i registri da leggere (Read Reg1, Read Reg2). Usa 2 multiplexer, ognuno con 32 ingressi e un segnale di controllo, che sceglie **da quale registro deve passare**.

Il register file fornisce sempre in output una coppia di registri. In generale:

1. *Selezionare il registro da cui si desidera leggere i dati, tramite il segnale di controllo*
2. *Impostare i segnali di controllo appropriati per abilitare la lettura dal registro selezionato*
3. *I dati contenuti saranno disponibili sul bus di lettura, e potranno essere usati come input.*



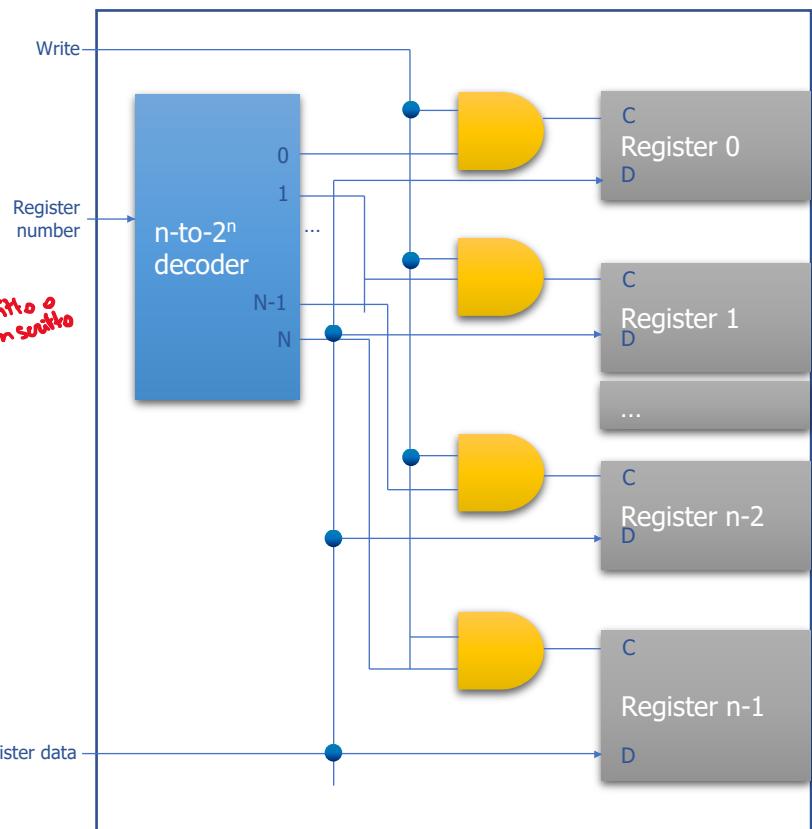
Scrittura

Utilizza tre segnali:

- Il registro da scrivere (Register Number)
- Il valore da scrivere (Register Data)
- Il segnale di controllo (Write)

Usa un **decoder** per decodificare il numero del registro da scrivere (*Write register*); il segnale Write (già in AND con il clock, ovvero sul fronte di salita) diventa in AND con l'output del decoder. Se invece Write non è affermato, allora nessun valore sarà scritto nel registro. In generale, può essere scritto come:

1. *Selezionare il registro di destinazione (tramite il segnale di controllo) in cui si scriveranno i dati*
2. *Impostare i segnali di controllo appropriati per abilitare la scrittura nel registro selezionato*
3. *Fornire i dati da scrivere tramite il bus di scrittura e collegarli ai bit del registro corrispondenti*
4. *Il registro selezionato memorizza i dati forniti.*



Memoria

Oltre alle piccole memorie implementate, per mezzo di registri e file di registri, esistono altri tipi di memorie che possiamo distinguere in base a diversi **parametri**:

1. **Dimensione**: quantità di dati memorizzabili
2. **Velocità**: l'intervallo di tempo tra la richiesta del dato e il momento in cui è disponibile
3. **Consumo**: potenza assorbita
4. **Costo**: costo per bit

Non è possibile avere un'unica memoria con tutte le caratteristiche ideali, ma possiamo organizzarle in una **gerarchia**:

- Le memorie più piccole, veloci, costose e che consumano di più sono poste ai **livelli alti**, quindi vicino alla CPU
- Le memorie ampie, più lente, meno costose e che consumano di meno sono poste ai **livelli bassi**.

Macchina a stato finito

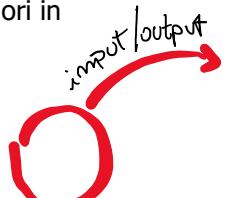
Una macchina a stato finito è un modello matematico che descrive un sistema che passa da uno stato ad un altro in risposta ad eventi specifici. La memoria è **cruciale** per le FSM, in quanto il sistema deve mantenere informazioni sullo stato corrente (e le condizioni precedenti), e di reagire di conseguenza quando si verificano determinati eventi. Sono sincronizzati con il clock

Sono composte da un set di stati e due funzioni:

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

Esistono due tipi di FSM:

- **FSM di tipo Mealy** produce un'uscita in base allo stato corrente e all'input corrente

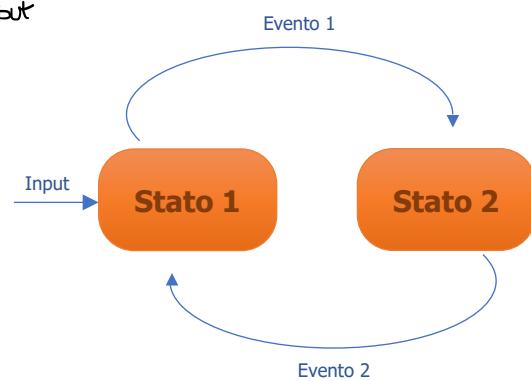


- **FSM di tipo Moore** produce un'uscita solo in base allo stato corrente.

Disegnare una FSM

Ecco un procedimento generale step-by-step:

1. Identificare gli **input e output** della FSM *input/output*
2. Identificare gli **stati**
3. Identificare gli **eventi** che causano una transizione da uno stato all'altro
4. Disegnare i cerchi per ogni stato
5. Etichettare ogni stato con un nome e, se necessario, con un'etichetta che descrive l'azione associata allo stato
6. Disegnare le frecce tra i cerchi per rappresentare le transizioni tra gli stati
7. Etichettare ogni freccia con l'evento che causa la transizione
8. Identificare eventuali **condizioni di output** associati a ogni stato o a ogni transizione e rappresentarle nella FSM.

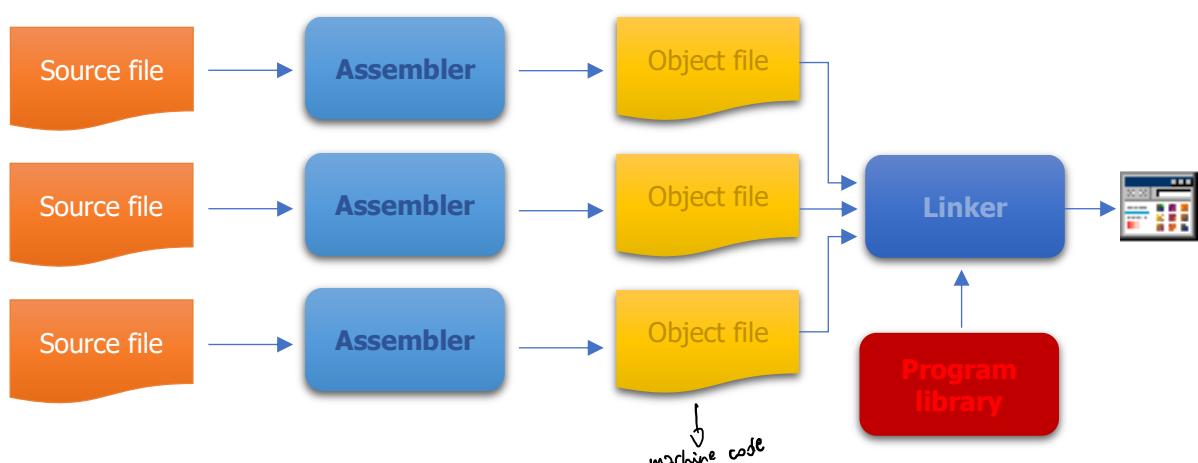


Assembly

Assembly è la **rappresentazione simbolica** del codice binario di un computer, chiamato anche **linguaggio macchina**. L'assembly è più leggibile del linguaggio macchina, perché usa simboli invece che bits. Inoltre, ci sono anche numerose keyword, come per esempio *opcodes* e *register specifiers*. Inoltre, possono essere usate *labels* per identificare, e attribuire un nome particolare a certe parti del codice.

L'assembly è inoltre fondamentale per la creazione di file eseguibili su un computer. Il processo si dota di vari passaggi:

- **Source file**: file di linguaggio assembly
- **Assembler**: traduce linguaggio assembly in istruzioni binarie. Legge un singolo file assembly, e produce un **object file** che contiene numerose istruzioni macchina, ed altre istruzioni che aiutano per **combinare più object file in un singolo programma**
 - o La maggior parte del programma è formato da più file, chiamati *moduli*, che sono scritti, compilati e assemblati indipendentemente.
 - Contiene riferimenti a subroutine e dati riferiti in altri moduli e librerie.
 - o Un programma potrebbe usare anche routine già scritte, fornite da una *program library*
- **Linker**: combina gli object e i library files in un singolo *executable file*.



Talvolta, potrebbe essere anche usato un **compilatore**, che non ha bisogno di un assembler prima, e quindi che traduce automaticamente il codice ad alto livello in codice a linguaggio macchina. Questi compilatori normalmente eseguono istruzioni molto più velocemente, rispetto a quelli che usano anche un assembler; in questo caso, però, il compilatore dovrà prendersi carico di una serie di azioni che di solito esegue l'assembler, come **risolvere indirizzi** e **codificare istruzioni in binario**.

Quando si usa l'assembly?

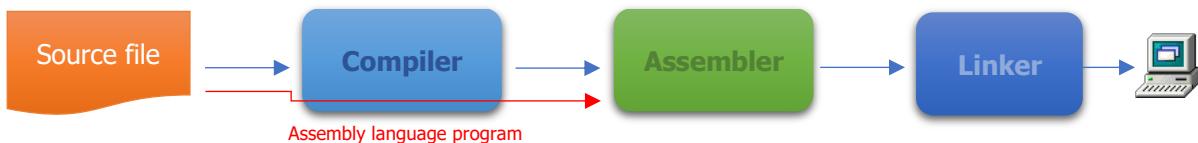
L'assembly, generalmente, gioca due ruoli. Il primo è che è **il linguaggio di output dei compilatori**, ovvero un programma che traduce un programma scritto in un codice ad alto livello in un programma uguale ma scritto in linguaggio macchina od assembly.

L'altro ruolo era quello di letteralmente scrivere programmi, che è una parte comunque importante in ambiti dove la velocità di esecuzione o la dimensione dell'eseguibile sono importanti, o per sfruttare funzionalità hardware che non sono accessibili con linguaggi ad alto livello.

Un programmatore assembly ha un controllo ben saldo su quali istruzioni vengono eseguite. In certi ambiti come l'embedded, **ridurre la dimensione di un programma**, o **velocizzarne l'esecuzione** diventa indispensabile. Molte volte, i programmatori possono anche usare un approccio *ibrido*, dove le parti di programma che devono avere un controllo maggiore sull'hardware, o essere eseguiti più velocemente, vengono scritti in assembly, mentre il resto viene scritto in un linguaggio ad alto livello.

I programmi scritti in assembly, però, sono **specifici per il computer per il quale si è scritto**, e hanno bisogno di una riscrittura per poter essere usati su altre architetture. Altro svantaggio è il fatto che i programmi possono essere molto più **lunghi**, rispetto ai loro corrispettivi in un linguaggio ad alto livello. Costrutti del linguaggi ad alto livello, come frasi if-then e loop, devono essere costruiti manualmente usando *branches* e *jumps*. Il programma risultante è, pertanto, difficile da leggere.

In base al linguaggio di programmazione che si usa, cambia anche la "catena di programmazione": se un programma è scritto in un linguaggio ad alto livello, allora passerà prima da un **compiler**; invece, se è direttamente creato in assembly, allora passerà direttamente a un assembler.



Avendo tre file con una dimensione di testo, e una dimensione dei suoi dati:

- La **dimensione del testo** equivale alla dimensione delle istruzioni assembly
- La **dimensione dei dati** equivale alla dimensione delle variabili, costanti e altre informazioni necessarie per l'esecuzione.

L'indirizzo base è sempre 0x10000000, e ogni sezione inizia a un indirizzo di memoria che dipende dalla sequenza di linkaggio.

Assembler

L'assembler traduce un file con istruzioni assembly in un file con istruzioni in **linguaggio macchina**, e data binari. Gestisce in automatico anche le etichette, le pseudo istruzioni, e i numeri in base diversa. Il processo di traduzione ha due step:

1. **Trova le aree di memoria con dei labels**, cosicché la relazione tra i nomi simbolici e gli indirizzi sono già conosciuti, mentre le istruzioni vengono tradotte.
2. **Traduce ogni istruzione assembly**, combinando assieme gli opcodes, le specifiche dei registri, e labels in una singola istruzione

L'assembler produce in output un **object file**, che contiene le istruzioni macchine. Tipicamente, questo file non può essere eseguito, perché contiene riferimenti a procedure o dati in file esterni. Di solito, una label è **esterna** se l'oggetto a cui è stata applicata può essere referenziato da altri file. Una label è invece **locale** se l'oggetto può essere usato soltanto all'interno del file in cui è stato definito (di solito, le labels sono locali come valore di default).

Visto che l'assembler processa ogni file individualmente, conosce soltanto gli indirizzi di memoria delle label locali. Per questo, l'assembler dipende su un altro strumento, chiamato **linker**, che combina vari *object files* e *library* in un singolo file eseguibile; questo processo, inoltre, automaticamente risolve problemi che possono sorgere da label esterne.

Le label (variabili) in assembly, però, hanno una particolare proprietà che i linguaggi di programmazione ad alto livello non posseggono: **le label possono essere usate prima di essere definite**. Questo forza l'assembler a prima processare tutto il programma alla ricerca di tutte le label, e quindi successivamente di produrre istruzioni.

Ogni modulo assemblato **di default parte dall'indirizzo 0**; nei sistemi dotati di meccanismi di memoria virtuale, tali indirizzi sono indirizzi virtuali.

Analisi più in dettaglio

Il primo passaggio di un assembler legge ogni riga del file assembly, e la divide nei suoi vari componenti; ognuno di questi componenti è chiamato **lessemi** (*lexemes*), che sono le singole parole, punteggiatura e i numeri.

```
ble      $t0, 100, Loop      ->    ble    $t0    ,    100    ,    Loop
```

Se una linea inizia con un label, allora l'assembler segna questa label all'interno della sua **symbol table** il nome della label, e l'indirizzo della memory word che l'istruzione sta occupando.

L'assembler, successivamente, **calcola quante parole di memoria** occuperà l'istruzione che ha appena analizzato. Tenendo traccia dell'area di memoria che ogni istruzione deve avere, potrà capire quale area di memoria deve essere attribuita alla prossima istruzione.

Il MIPS usa le **fixed-length instructions**, che hanno bisogno semplicemente di un'analisi della riga in cui correntemente sono.

Quando l'assembler arriva alla fine di una linea, la symbol table registra la posizione di tutte le label nel file. Questa informazione verrà poi successivamente usata di nuovo dall'assembler per la seconda passata, dove verrà veramente prodotto codice macchina. Viene analizzato di nuovo linea per linea, e se la linea contiene un'istruzione, l'assembler combina **la rappresentazione binaria degli opcode e degli operandi** in un'istruzione legale.

Divisione righe in lessemi

Se la linea inizia con una label
Salvata in symbol table + address

Registro nel file della posizione
delle label dalla symbol table

Seconda analisi: combina binary
di opcode e operandi in
istruzione

Tabella dei simboli

La tabella dei simboli dell'assembla contiene i riferimenti simbolici presenti nel modulo da tradurre e, al termine del primo passo, conterrà gli **indirizzi numerici di tutti i simboli**, tranne quelli esterni al modulo in esame.

Se ci sono etichette che definiscono costanti simboliche, viene creata la coppia *<etichetta, valore>*, e in ogni istruzione che fa riferimento al simbolo viene sostituito il valore.

Se ci sono etichette che definiscono variabili (spazi di memoria), l'assemblatore riserva lo spazio necessario, inizializza la zona di memoria, e crea la coppia *<etichetta, indirizzo>*.

Se ci sono etichette che definiscono istruzioni di salto, l'assemblatore deve generare un riferimento all'indirizzo dell'istruzione destinazione di salto.

Debugger

Il debugger è uno strumento che non viene usato dal computer nel compilare un programma, ma viene invece usato dallo sviluppatore. Consente di eseguire in modo controllato un programma per la **ricerca di un malfunzionamento**. Ha funzioni come l'ispezione del valore di variabili ed espressioni, breakpoint, interruzioni in caso di modifica del valore di determinate variabili (*watchpoint*), visualizzazione degli indirizzi di memoria etc...

Compilatore

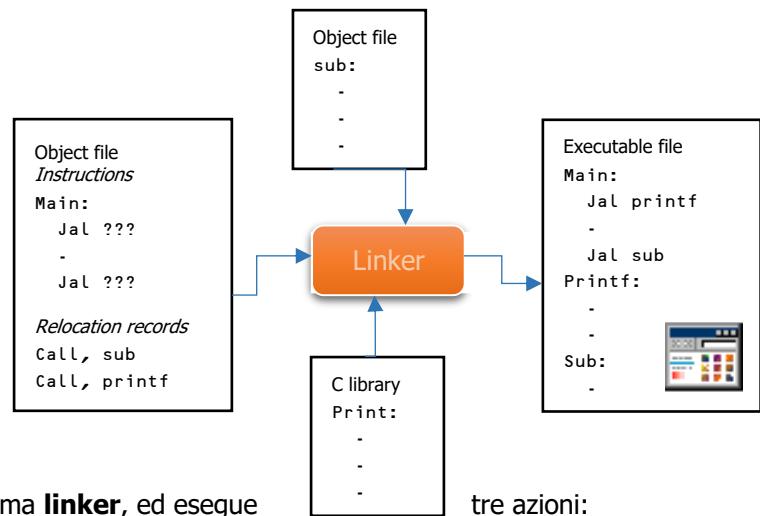
Un programma ad alto livello viene tradotto in linguaggio assembly usando questo strumento; è diverso da un assembler, perché **non traduce in linguaggio macchina**, ma semplicemente in assembly. Quindi, il programma creato con un compilatore dovrà essere successivamente lavorato anche da un assembler.

Il compiler permette, inoltre, di ottimizzare automaticamente il codice, rimuovendo parti che possono essere considerate ridondanti.

Linker

Compilazioni separate permettono a un programma di essere separato in file diversi; ogni file contiene una collezione di subroutine e strutture dati che formano un **modulo** in un programma più largo.

Un file, successivamente, può essere compilato e assemblato indipendentemente da altri file, cosicché **cambiamenti a un modulo non richiedono un'intera nuova ricompilaione**.



Lo strumento che mette assieme questi file si chiama **linker**, ed esegue

- Cerca nel programma per riferimenti a routine della libreria usate questo è fatto per cercare quali subroutine predefinite e strutture dati sono usate dal programma
 - o Cerca nel programma che non ci siano label che siano non definite. Il linker fa in modo di collegare i simboli esterni e le reference non risolte da un programma.
 - o Un simbolo esterno in un programma risolve una reference in un altro file, se tutti e due referenziavano una label con lo stesso nome.
 - o **Reference non collegate significa che un simbolo è stato usato, ma non definito nel programma.**
- Estrae i codici delle varie routine, e lo incorpora nel segmento di testo del programma. Questa nuova routine, a sua volta, potrà dipendere da altre routine, e quindi il linker continua a fare questo lavoro fino a quando tutti i collegamenti non sono stati effettuati.
- Determina l'area di memoria che ogni modulo occuperà. Visto che i file vengono assemblati separatamente, l'assembler **non può conoscere l'area di memoria delle istruzioni o data rispetto ad altri moduli**.
 - o Quando il linker piazza un modulo in memoria, tutte le reference assolute devono essere rilocate per riflettere la loro vera posizione.
 - o Visto che il linker ha tutte le informazioni di filo azione, allora può facilmente trovare e mettere a posto queste reference.

tre azioni:
dal programma;

Loader

Il loader si trova alla fine della catena di programmazione, e permette di eseguire il file. Legge l'**intestazione** del file eseguibile, per determinare la lunghezza delle istruzioni e delle variabili. Quindi, crea uno spazio di indirizzamento sufficiente a contenere testo e dati, copia le istruzioni e dati dal file in memoria. Eventuali **parametri** verranno copiati nello stack; questi parametri saranno poi da passare al programma principale.

Successivamente, **inizializza i registri**, imposta lo stack pointer, e salta a una procedura di **startup**: questa procedura copia i parametri nei registri argomento, e chiama la procedura principale del programma.

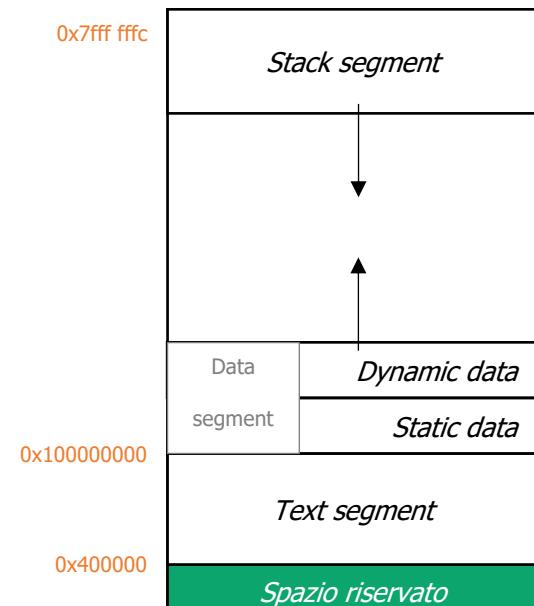
Quando la procedura principale restituisce il controllo, la procedura di startup termina il programma chiamando la funzione di sistema **exit**.

Semplicemente, è uno strumento che analizza lo header del file eseguibile, alloca le risorse necessarie in memoria centrale, e gestisce l'esecuzione del programma.

Uso della memoria

I sistemi basati su processore MIPS tipicamente dividono la memoria in **tre parti**:

1. Vicino al fondo dello spazio degli indirizzi (parte a 0x4000000) è il **text segment**, ovvero la parte dove sono contenute le istruzioni del programma
2. **Data segment**, che è diviso a sua volta in due parti:
 - a. **Static data** (0x10000000) che contiene gli oggetti la quale dimensione è conosciuta dal compilatore, e durata è pari al tempo di esecuzione del programma.
 - b. **Dynamic data**, che è allocata da un programma direttamente all'esecuzione stessa. Visto che un compilatore non può predire prima quanta memoria possa servire a un programma, il sistema operativo estende questa area in base alle necessità.
3. **Stack segment** (parte a 0x7fffffff); anche in questo caso, la dimensione dello stack di un programma non si può sapere prima. Mano a mano che il programma manda valori sullo stack, il sistema operativo allarga la dimensione dello stack verso il basso.



La posizione così particolare dello stack e del dynamic data (che sono agli estremi) permette al sistema di operativo di non farli coincidere, e quindi non causare problemi.

Convenzione delle chiamate di procedura

Un compilatore deve conoscere quali registri può usare, e quali registri sono invece riservati ad altre procedure. Per stabilire questo, è stata creata una **convenzione** seguita dal software. Un processore MIPS contiene **32 registri "general-purpose"**, divisi in:

- **\$0**: valore forzato di 0
- **\$at** (1), **\$k0** (26), **\$k1** (28) sono riservati all'assembler e al sistema operativo
- **\$a0-\$a3** (4-7) sono usati per passare i primi quattro argomenti alle routine (le rimanenti sono passate allo stack)
- **\$v0** (2), **\$v1** (3) sono usati per far "ritornare" i valori da funzioni
- **\$t0-\$t9** (8-15, 24, 25) sono **registri salvati dal chiamante** (caller-saved registers), e sono usati per tenere in memoria temporaneamente dati
- **\$s0-\$s7** (16-23) sono **registri salvati dal chiamato** (callee-saved registers), e sono usati per tenere in memoria dati che devono essere salvati durante multiple chiamate
- **\$gp** (28) è un global pointer, che punta nel mezzo di un blocco 64K nel static data.
- **\$sp** (29) è lo **stack pointer**, che punta all'ultima posizione sullo stack
- **\$fp** (30) è il **frame pointer**
- **\$ra** (31) è usato di solito da *jal*, e viene usato come indirizzo di ritorno da una chiamata di procedura.

Register name	Number	Usage
\$zero	0	Constant 0
\$at	1	Reserved for assembler
\$v0	2	Results of a function
\$v1	3	Results of a function
\$a0	4	Argument1
\$a1	5	Argument2
\$a2	6	Argument3
\$a3	7	Argument4
\$t0	8	Temporary
\$t1	9	Temporary
\$t2	10	Temporary
\$t3	11	Temporary
\$t4	12	Temporary
\$t5	13	Temporary
\$t6	14	Temporary
\$t7	15	Temporary
\$s0	16	Saved temporary
\$s1	17	Saved temporary
\$s2	18	Saved temporary
\$s3	19	Saved temporary
\$s4	20	Saved temporary
\$s5	21	Saved temporary
\$s6	22	Saved temporary
\$s7	23	Saved temporary
\$t8	24	Temporary
\$t9	25	Temporary
\$k0	26	Reserved for kernel
\$k1	27	Reserved for kernel
\$gp	28	Pointer to global area
\$sp	29	Stack pointer
\$fp	30	Frame pointer
\$ra	31	Return address

Chiamate di procedura

Quando una procedura (il *caller*) chiama un'altra procedura (il *callee*), viene eseguita una **chiamata di procedura**. Questa chiamata è centrata attorno a un'area di memoria chiamata **procedure call frame**, che è usata per vari scopi:

- Per tenere valori che devono essere passati come argomenti ad una procedura
- Per salvare registri che una procedura potrebbe modificare, ma che la procedura del chiamante non vuole cambiare
- Per fornire spazio alle variabili locali di una procedura

Nella maggior parte dei linguaggi di programmazioni, le chiamate di procedura e i *return* seguono la regola **LIFO**, cosicché la memoria può essere allocata e deallocated da uno stack. Sulla maggior parte delle macchine MIPS, questa procedura viene effettuata immediatamente **prima** che il chiamante invochi il chiamato, appena il callee parte, e subito **dopo** che il callee ritorni al caller.

1. **Passa gli argomenti.** Secondo convenzione, i primi quattro argomenti sono passati ai registri \$a0-\$a3. Ogni argomento rimanente è mandato sullo stack, e sono all'inizio dello stack frame della procedura del chiamante.
2. **Salva i registri salvati del chiamante.** La procedura chiamata può usare i registri \$a0-\$a3 e \$t0-\$t9, senza dover salvare prima il loro valore. Se il chiamante pensa di usare uno di questi registri dopo una chiamata, deve salvare il suo valore prima della chiamata.
3. **Esegue un'istruzione jal**, che fa saltare l'esecuzione alla prima chiamata del callee, e salva l'indirizzo di ritorno nel registro \$ra.

Prima che una routine chiamata possa partire, deve impostare il suo stack frame:

1. Allocare memoria per il frame, sottraendo la dimensione del frame dallo stack pointer
2. Salva i "callee-saved registers" nel frame. Un callee deve salvare i suoi valori nei registri \$s0-\$s7, \$fp e \$ra, prima di poterli modificare, perché il chiamante si aspetta che questi registri non siano modificati. Però, solo \$ra deve essere salvato, se il callee fa una chiamata.

3. Stabilire il frame pointer, aggiungendoci la dimensione dello stack frame – 4 a \$sp, e poi salva il risultato in \$fp.

Finalmente, il callee ritorna al caller facendo:

1. Se il callee è una funzione che *ritorna un valore*, e mette il valore ritornato nel registro \$v0.
2. Ripristina tutti i “callee-saved registers” che sono stati salvati quando la procedura è iniziata
3. Pop lo stack frame aggiungendoci la dimensione del frame a \$sp
4. Ritorna, con un salto, all’indirizzo \$ra.

Il Program Counter è un registro della CPU che conserva l’indirizzo di memoria della prossima istruzione, in linguaggio macchina, da eseguire. Viene sempre incrementato di 4 byte dopo l’esecuzione di ogni istruzione; questo si

Input e output

SPIM, il simulatore MIPS per architettura x86, simula un singolo dispositivo I/O: una console memory-mapped sulla quale il programma può leggere e scrivere caratteri. Quando un programma sta girando, SPIM connette il proprio terminale al processore. Per interromperlo, l’utente può premere **command+C**; questo comando non è passato al programma, ma semplicemente ferma l’esecuzione del programma. Quando l’esecuzione è fermata, il terminale è riconnesso a SPIM, cosicché si possono scrivere comandi di SPIM.

Il dispositivo del terminale ha due unità indipendenti: il **receiver** (che legge i caratteri da tastiera) e un **transmitter** (che mostra i caratteri sullo schermo).

Un programma controlla il terminale con quattro “memory-mapped” registri di dispositivo, ovvero che sono in una specifica e speciale area di memoria.

SPIM

SPIM è il simulatore dell’architettura MIPS32, che esegue codice assembly. Specificamente, supporta **MIPS32 Release 1**, con una mappatura della memoria fixata, no cache, e soltanto il coprocessore 0 e 1. SPI è un “self-contained system” per far girare programmi MIPS; contiene un debugger, e ha qualche funzione OS. SPIM è molto più lento di un computer reale, ma viene usato comunque rispetto a un processore con architettura x86-64 poiché l’architettura MIPS è molto più regolare e facile da capire e programmare rispetto a un’architettura Intel 80x86.

L’assembler implementa una **macchina virtuale**; questo computer ha dei *nondelayed branches*, caricamenti, e un *instruction set* migliore rispetto a un vero computer. L’assembler riorganizza le istruzioni per riempire i *delay slots*, e questo computer virtuale provvede anche per le *pseudoinstructions*, che comunque appaiono come normalissime istruzioni.

SPIM non simula la cache, o varie latenze della memoria, e non rispetta perfettamente le operazioni floating-point, o i delay nelle istruzioni di moltiplicazione o divisione. Le istruzioni floating-point inoltre ignorano qualche condizione di errore.

Ordine dei bit

I processori MIPS possono operare sia con **big-endian** (0,1,2,3) o **little-endian** (3,2,1,0) come ordine.

Chiamate di sistema

SPIM ha delle chiamate *syscall*. Per richiedere un servizio, un programma carica il codice per la chiamata del sistema nel registro \$v0, e gli argomenti nei registri \$a0-\$a3 (o \$f12 per le variabili floating-point). Le chiamate di sistema che ritornano valori posizionano i propri risultati nel registro \$v0 (o \$f0 per le variabili floating-point).

Alcuni servizi possono essere *print_int*, *print_float*, *print_double*, *print_string*, *read_int*, *read_float*, *read_double*, *read_string*, *sbrk*, *exit*, *print_char*, *read_char*, *open*, *read*, *write*, *close*, *exit2*.

Linguaggio MIPS R2000

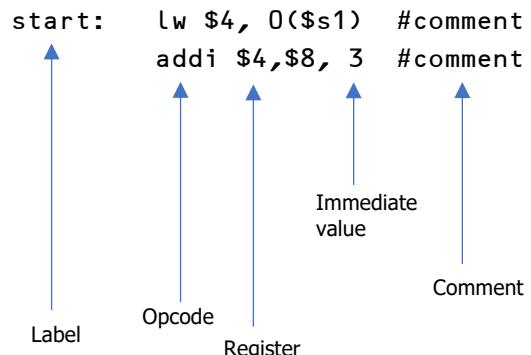
SPIM simula due coprocessori. Il coprocessore 0 gestisce le eccezioni e le interruzioni. Il coprocessore 1 invece è la **floating-point unit**.

MIPS è un'architettura load-store, ovvero che soltanto queste due istruzioni possono accedere alla memoria. La maggior parte delle funzioni load e store operano soltanto su dati allineati; un dato è **allineato** se l'indirizzo di memoria è un multiplo della sua dimensione in bytes.

Sintassi

L'assembly segue una propria sintassi:

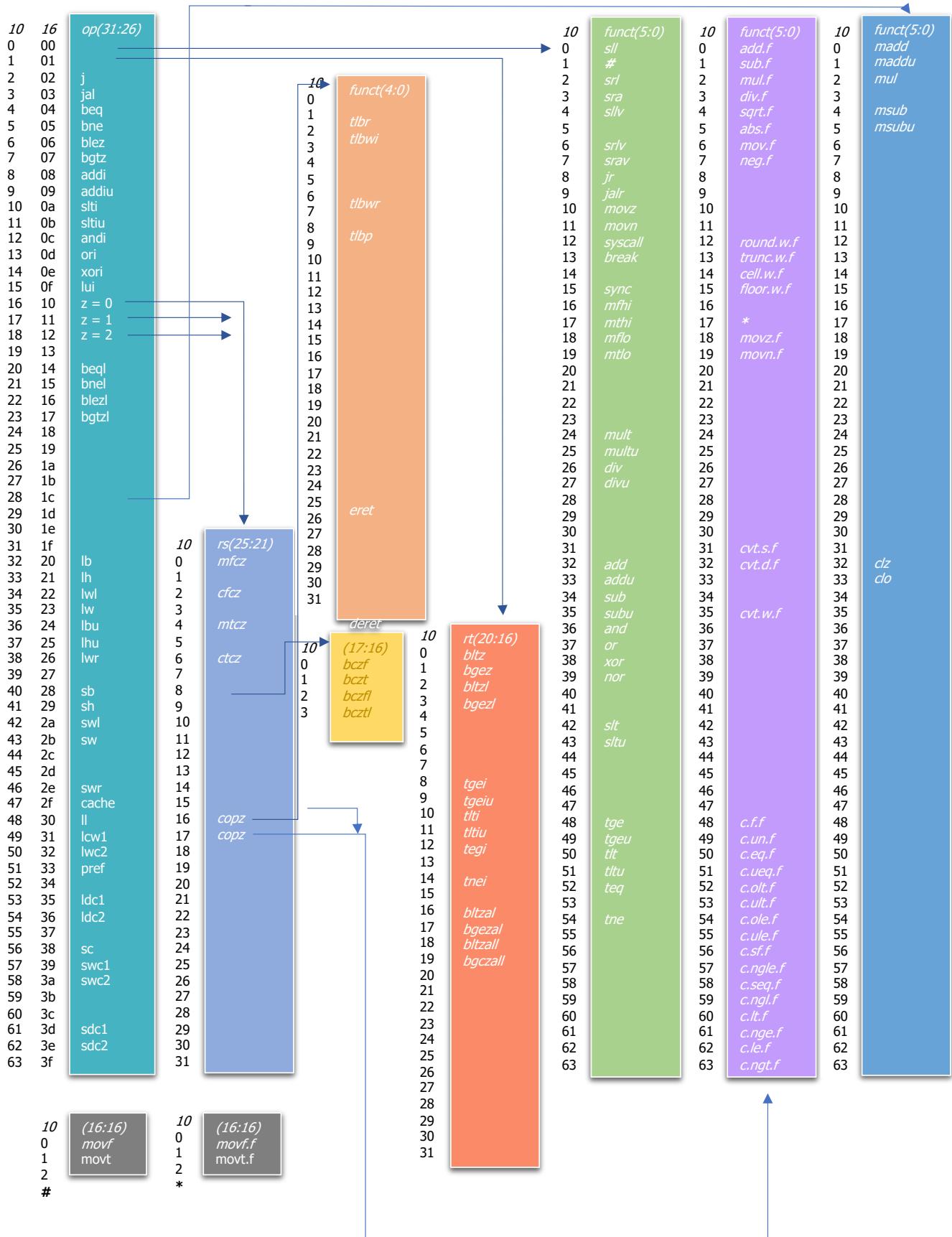
- I nomi che iniziano con un . sono **direttive assembler**, che dicono all'assembla come tradurre un programma, ma non producono codice macchina.
- I nomi che sono seguiti da una , sono labels che nominano la prossima area di memoria
- L'inizio di un'istruzione assembly è generalmente l'**opcode**, che dice alla macchina quale operazione deve eseguire
 - o Di solito, questi nomi sono derivati dalle funzioni che eseguono
- I numeri che iniziano (di solito hexadecimale) che iniziano con \$ sono registri
- I commenti sono rappresentati da #; ogni cosa che sta dopo questo simbolo verrà ignorata.
- **Identifiers** è una sequenza di caratteri alfanumerici che non iniziano con un numero..
- I numeri sono in **base 10**; se prima ci sta 0x, allora sono esadecimale.
- Le stringhe sono tra ", e possono essere inseriti i caratteri speciali \n, \t e \"



Ogni colonna contiene encoding per le istruzioni per un field, da un'istruzione. I numeri che sono sulla sinistra sono valori per dei field. Il testo all'inizio della colonna imposta un nome a un field, e specifica quali bit occupa in un'istruzione.

Istruzioni del MIPS

Le istruzioni del MIPS si basano generalmente sulle stesse condizioni: opcode, registri o variabili, e una variante.



Title

instruction	Opcode	Rs	Rt	Rd	0	variant
	6	5	5	5	5	6

Description

multiply (without overflow)

`mult rdest, rsrc1, src2`

Frase pseudonima

Addition (with overflow)

Add rd, rs, rt	0	Rs	Rt	Rd	0	0x20
	6	5	5	5	5	6

Il registro "rd" è dove verrà memorizzato il risultato dell'addizione. Visto che l'opcode è 0, allora andiamo a "cercare" nella tabella il corrispettivo 0x20, che è "32"

Addition (without overflow)

Addu rd, rs, rt	0	Rs	Rt	Rd	0	0x21
	6	5	5	5	5	6

Metti la somma dei registri rs e rt nel registro rd.

Addition immediate (with overflow)

addi rt, rs, imm	8	Rs	Rt	imm	
	6	5	5	16	

Metti la somma del registro rs e del valore immediato nel registro rt.

Addition immediate (without overflow)

Addiu rt, rs, imm	Opcode	Rs	Rt	Rd	
	6	5	5	16	

Metti la somma del registro rs e del valore immediato nel registro rt.

AND

and rd, rs, rt	0	Rs	Rt	Rd	0	0x24
	6	5	5	5	5	6

Metti il risultato dell'operazione logica AND tra I registri rt e rs nel registro rd.

AND immediate

andi rt, rs, imm	0xc	Rs	Rt	Rd	
	6	5	5	16	

Metti il risultato dell'operazione logica AND tra il registro rs, e il valore immediato con l'aggiunta degli zeri, nel registro rt.

Count leading ones

<code>clo rd, rs</code>	0x1c	Rs	0	Rd	0	0x21
	6	5	5	5	5	6

Conta il numero di uno che ci sono all'inizio della parola nel registro rs, e metti il risultato nel registro rd. Se una parola è tutti uno, allora il risultato è 32.

Count leading zeros

<code>clz rd, rs</code>	0x1c	Rs	0	Rd	0	0x20
	6	5	5	5	5	6

Conta il numero di zeri che ci sono all'inizio della parola nel registro rs, e metti il risultato nel registro rd. Se una parola è tutti zeri, il risultato è 32.

Dividi (with overflow)

<code>div rs, rt</code>	0	Rs	Rt	0	0x1a
	6	5	5	10	6

Dividi il registro rs per il registro rt. Metti il quoziente nel registro **lo**, e il resto nel registro **hi**. Se un operando è negativo, il resto non è specificato dall'architettura MIPS, e dipende dalla convenzione che è attualmente in uso.

Divide (without overflow)

<code>divu rs, rt</code>	Opcode	Rs	Rt	0	0x1b
	6	5	5	10	6

Dividi il registro rs per il registro rt. Metti il quoziente nel registro **lo**, e il resto nel registro **hi**. Se un operando è negativo, il resto non è specificato dall'architettura MIPS, e dipende dalla convenzione che è attualmente in uso.

Divide (with overflow) pseudoinstruction

`div rdest, rsrc1, src2`

Metti il quoziente della divisione tra rsrc1 e src2 nel registro rdest.

Divide (without overflow) pseudoinstruction

`divu rdest, rscr1, scr2`

Metti il quoziente della divisione tra rscr1 e scr2 nel registro rdest.

Multiply

<code>mult rs, rt</code>	0	Rs	Rt	0	0x18
	6	5	5	10	6

Moltiplica i registri rs e rt. Lascia la parola in low-order del prodotto nel registro **lo**, e invece quella in high-order nel registro **hi**.

Unsigned multiply

<code>multu rs, rt</code>	0	Rs	Rt	0	0x19
	6	5	5	10	6

Moltiplica I registri rs e rt. Lascia la parola in low-order del prodotto nel registro **lo**, e invece quella in high-order nel registro **hi**.

Multiply (with overflow) pseudoinstruction

mulo rdest, rsrcl, src2

Metti i 32-bit in low-order del prodotto tra I registri rsrcl e src2 nel registro rdest.

Unsigned multiply (with overflow) pseudoinstruction

mulou rdest, rsrcl, src2

Metti i 32-bit in low-order del prodotto tra I registri rsrcl e src2 nel registro rdest.

Multiply (without overflow)

mul rd, rs, rt

0x1c	Rs	Rt	Rd	0	2
6	5	5	5	5	6

Metti i 32-bit in low-order del prodotto tra I registri rs e rt nel registro rd.

Multiply add

madd rs, rt

0x1c	Rs	Rt	0	0
6	5	5	10	6

Moltiplica I registri rs e rt e aggiungi il prodotto risultante a 64-bit al valore a 64-bit nei registri concatenati **hi** e **lo**.

Unsigned multiply add

maddu rs, rt

0x1c	Rs	Rt	0	1
6	5	5	10	6

Moltiplica I registri rs e rt e aggiungi il prodotto risultante a 64-bit al valore a 64-bit nei registri concatenati **hi** e **lo**.

Multiply subtract

msub rs, rt

0x1c	Rs	Rt	0	4
6	5	5	10	6

Moltiplica I registri rs e rt e aggiungi il prodotto risultante a 64-bit al valore a 64-bit nei registri concatenati **hi** e **lo**.

Unsigned multiply subtract

msubu rs, rt

0x1c	Rs	Rt	0	5
6	5	5	10	6

Moltiplica I registri rs e rt, e aggiungi il prodotto risultante a 64-bit al valore a 64-bit nei registri concatenati **hi** e **lo**.

Negate value (with overflow) pseudoinstruction

neg rdest, rsrcl

Metti il valore negativo del registro rsrc nel registro rdest.

Negate value (without overflow) pseudoinstruction

`negu rdest, rsrc`

Metti il valore negativo del registro rsrc nel registro rdest.

NOR

<code>nor rd, rs, rt</code>	0	Rs	Rt	Rd	0	0x27
	6	5	5	5	5	6

Metti il risultato del NOR logico tra I registri rs e rt nel registro rd.

NOT pseudoinstruction

`not rdest, rsrc`

Metti il risultato dell'operazione logica not (a seconda dei bit) del registro rsrc nel registro rdest.

OR

<code>or rd, rs, rt</code>	0	Rs	Rt	Rd	0	0x25
	6	5	5	5	5	6

Metti il risultato di un OR logico tra I registri rs e rt nel registro rd.

OR immediate

<code>ori rt, rs, imm</code>	0xd	Rs	Rt	imm
	6	5	5	16

Metti Il risultato di un OR logico tra il registro rs, e il valore immediato (esteso con gli zeri) nel registro rt.

Remainder pseudoinstruction

`rem rdest, rsrc1, rsrc2`

Metti il resto del registro rsrc1 diviso per il registro rsrc2 nel registro rdest.

Unsigned remainder pseudoinstruction

`remu rdest, rsrc1, rsrc2`

Metti il resto del registro rsrc1 diviso per il registro rsrc2 nel registro rdest.

Shift left logical

<code>sll rd, rt, shamt</code>	0	Rs	Rt	Rd	shamt	0
	6	5	5	5	5	6

Shifta il registro rt verso sinistra per la distanza indicata dalla variabile immediata "shamt", e metti il risultato nel registro rd. (in questo caso rs è zero)

Shift left logical variable

sllv rd, rt, rs

0	Rs	Rt	Rd	0	4
6	5	5	5	5	6

Shifta il registro rt verso sinistra per la distanza indicata dal registro rs, e metti il risultato nel registro rd.

Shift right arithmetic

sra rd, rt, shamt

0	Rs	Rt	Rd	shamt	3
6	5	5	5	5	6

Shifta il registro rt verso destra per la distanza indicata dalla variabile immediata shamt, e metti il risultato nel registro rd.

Shift right arithmetic variable

sra v rd, rt, rs

0	Rs	Rt	Rd	0	7
6	5	5	5	5	6

Shifta il registro rt verso destra per la distanza indicata dal registro rs, e metti il risultato nel registro rd.

Shift right logical

srl rd, rt, shamt

0	Rs	Rt	Rd	shamt	2
6	5	5	5	5	6

Shifta il registro rt verso destra per la distanza indicata dalla variabile immediata shamt, e metti il risultato nel registro rd.

Shift right logical variable

srl v rd, rt, rs

0	Rs	Rt	Rd	0	6
6	5	5	5	5	6

Shifta il registro rt verso destra per la distanza indicata dal registro rs, e metti il risultato nel registro rd.

Rotate left pseudoinstruction

rol rdest, rsrc1, rsrc2

Ruota il registro rsrc1 a sinistra in base alla distanza indicata da rsrc2, e metti il risultato in rdest.

Rotate right pseudoinstruction

ror rdest, rscr1, rscr2

Ruota il registro rsrc1 a destra in base alla distanza indicata da rsrc2, e metti il risultato in rdest.

Subtract (with overflow)

sub rd, rs, rt

0	Rs	Rt	Rd	0	0x22
6	5	5	5	5	6

Metti la differenza tra i registri rs e rt nel registro rd.

Subtract (without overflow)

<code>subu rd, rs, rt</code>	0	Rs	Rt	Rd	0	0x23
	6	5	5	5	5	6

Metti la differenza dei registri rs e rt nel registro rd.

Exclusive OR

<code>xor rd, rs, rt</code>	0	Rs	Rt	Rd	0	0x26
	6	5	5	5	5	6

Metti il risultato di un'operazione logica XOR tra i registri rs e rt nel registro rd.

XOR immediato

<code>xori rt, rs, imm</code>	0xe	rs	Rt	Imm		
	6	5	5	16		

Metti il risultato di un'operazione logica XOR tra i registri rs e il valore immediato (con zero estesi) nel registro rt.

Set less than

<code>slt rd, rs, rt</code>	0	Rs	Rt	Rd	0	0x2a
	6	5	5	5	5	6

Imposta il registro rd a 1 se il registro rs è minore di rt, o 0 per l'opposto.

Set less than unsigned

<code>sltu rd, rs, rt</code>	0	Rs	Rt	Rd	0	0x2b
	6	5	5	5	5	6

Imposta il registro rd a 1 se il registro rs è minore di rt, o 0 per l'opposto.

Set less than immediate

<code>slti rt, rs, imm</code>	0xa	Rs	Rt	imm		
	6	5	5	16		

Imposta il registro rt a 1 se il registro rs è minore del valore immediato (sign-extended), o 0 per l'opposto.

Set less than unsigned immediate

<code>sltiu rt, rs, imm</code>	Opcode	Rs	Rt	Imm		
	6	5	5	16		

Imposta il registro rt a 1 se il registro rs è minore del valore immediato (sign-extended), o 0 per l'opposto.

Set equal pseudoinstruction

`seq rdest, rsrc1, rsrc2`

Imposta il registro rdest a 1 se il registro rsrc1 è uguale a rsrc2, altrimenti a 0.

Set greater than equal pseudoinstruction

`sge rdest, rsrc1, rsrc2`

Imposta il registro rdest a 1 se il registro rsrc1 è uguale a rsrc2, altrimenti a 0.

Set greater than equal unsigned pseudoinstruction

```
sgeu rdest, rsrc1, rsrc2
```

Imposta il registro rdest a 1 se il registro rsrc1 è uguale a rsrc2, altrimenti a 0.

Set greater than pseudoinstruction

```
sgt rdest, rsrc1, rsrc2
```

Imposta il registro rdest a 1 se il registro rsrc1 è maggiore di rsrc2, altrimenti a 0.

Set greater than unsigned pseudoinstruction

```
sgtu rdest, rsrc1, rsrc2
```

Imposta il registro rdest a 1 se il registro rsrc1 è maggiore di rsrc2, altrimenti a 0.

Set less than equal pseudoinstruction

```
sle rdest, rsrc1, rsrc2
```

Imposta il registro rdest a 1 se il registro rsrc1 è minore di rsrc2, altrimenti a 0.

Set less than equal unsigned pseudoinstruction

```
sieu rdest, rsrc1, rsrc2
```

Imposta il registro rdest a 1 se il registro rsrc1 è minore di rsrc2, altrimenti a 0.

Set not equal pseudoinstruction

```
sne rdest, rsrc1, rsrc2
```

Imposta il registro rdest a 1 se il registro rsrc1 è minore di rsrc2, altrimenti a 0.

Istruzioni branch

Le istruzioni branch usano una istruzione signed a 16-bit chiamata **offset field**; per questo, possono saltare avanti di $2^{15} - 1$ istruzioni, o 2^{15} istruzioni indietro. Un'istruzione jump invece contiene un **address field** di 26-bit.

In un processore MIPS, queste istruzioni sono ritardate, ovvero non ridanno controllo fino a quando l'istruzione dopo il branch è stata eseguita. Queste istruzioni ovviamente modificano anche il calcolo dell'offset, visto che deve essere calcolata relativamente all'indirizzo dell'istruzione ritardata (PC + 4), ovvero quando viene eseguito il branch. SPIM non simula questo aspetto.

Gli offsets non sono specificati come numeri. Invece, si esegue un branch a una **label**, e l'assembler calcola la distanza dal branch all'istruzione target. Tutte le operazioni branch condizionali hanno una variante simile, che non esegue l'istruzione nello slot delay del branch, se il branch non è finito.

Branch pseudoinstruction

b label

0	Rs	Rt	Rd	0	0x23
6	5	5	5	5	6

Senza alcuna condizione, salta all'istruzione dove c'è la label.

Branch coprocessor false

bclf cc label

0x11	8	cc	0	Offset
6	5	3	2	16

Condizionalmente fa un branch del numero di istruzioni specificate dall'offset, se la condition flag del coprocessore del floating point chiamato **cc** è falso. Se cc non è inserito all'interno dell'istruzione, allora di default cc = 0.

Branch coprocessor true

bclt cc label

0x11	8	cc	1	Offset
6	5	3	2	16

Condizionalmente fa un branch del numero di istruzioni specificate dall'offset, se la condition flag del coprocessore del floating point chiamato **cc** è falso. Se cc non è inserito all'interno dell'istruzione, allora di default cc = 0.

Branch on equal

beq rs, rt, label

4	Rs	Rt	offset
6	5	5	16

Condizionalmente fa un branch del numero di istruzioni specificate dall'offset se il registro rs è uguale al registro rt.

Branch on greater than equal zero

bgez rs, label

1	Rs	1	offset
6	5	5	16

Condizionalmente fa un branch del numero di istruzioni specificate dall'offset se il registro rs è maggiore o uguale a 0.

Branch on greater than equal zero and link

bgezal rs, label

1	Rs	0x11	offset
6	5	5	16

Condizionalmente fa un branch del numero di istruzioni specificate dall'offset se il registro rs è maggiore o uguale a 0. Salva poi l'indirizzo della prossima istruzione nel registro 31.

Branch on greater than zero

bgtz rs, label

7	Rs	0	offset
6	5	5	16

Condizionalmente fa un branch del numero di istruzioni specificate dall'offset se il registro rs è maggiore di zero.

Branch on less than equal zero

blez rs, label

6	Rs	0	offset
6	5	5	16

Condizionalmente fai un branch del numero di istruzioni specificate dall'offset se il registro rs è minore o uguale a zero.

Branch on less than and link

bltzal rs, label

1	Rs	0x10	offset
6	5	5	16

Condizionalmente fai un branch del numero di istruzioni specificate dall'offset se il registro rs è minore di zero. Salva l'indirizzo dell'istruzione dopo nel registro 31.

Branch on less than zero

bltz rs, label

1	Rs	0	offset
6	5	5	16

Condizionalmente fai un branch del numero di istruzioni specificate dall'offset se il registro rs è minore di zero.

Branch on not equal

bne rs, rt, label

4	Rs	Rt	offset
6	5	5	16

Condizionalmente fai un branch del numero di istruzioni specificate dall'offset se il registro rs non è uguale al registro rt.

Branch on equal zero pseudoinstruction

beqz rsrc, label

Condizionalmente fai un branch all'istruzione alla label se rsrc è uguale a 0.

Branch on greater than equal pseudoinstruction

bge rsrc1, rsrc2, label

Condizionalmente fai un branch all'istruzione alla label se rsrc1 è più grande o uguale a rsrc2.

Branch on greater than equal unsigned pseudoinstruction

bgeu rsrc1, rsrc2, label

Condizionalmente fai un branch all'istruzione alla label se rsrc1 è più grande o uguale a rsrc2.

Branch on greater than pseudoinstruction

bgt rsrc1, src2, label

Condizionalmente fai un branch all'istruzione alla label se rsrc1 è più grande di src2.

Branch on greater than unsigned pseudoinstruction

bgtu rsrc1, src2, label

Condizionalmente fai un branch all'istruzione alla label se rsrc1 è più grande di src2.

Branch on less than equal pseudoinstruction

`ble rsrc1, src2, label`

Condizionalmente fai un branch all'istruzione alla label se rsrc1 è più piccolo o uguale di src2.

Branch on less than equal unsigned pseudoinstruction

`bleu rsrc1, src2, label`

Condizionalmente fai un branch all'istruzione alla label se rsrc1 è più piccolo o uguale di src2.

Branch on less than pseudoinstruction

`blt rsrc1, src2, label`

Condizionalmente fai un branch all'istruzione alla label se rsrc1 è più piccolo di src2.

Branch on less than unsigned pseudoinstruction

`bltu rsrc1, src2, label`

Condizionalmente fai un branch all'istruzione alla label se rsrc1 è più piccolo di src2.

Branch on not equal zero pseudoinstruction

`bnez rsrc, label`

Condizionalmente fai un branch all'istruzione alla label se il registro rsrc non è uguale a 0.

Jump

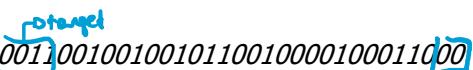
`j target`

2	target
6	26

Esegui un jump all'istruzione nel target.

Jump con program counter

Il "target" di 26 bit, sono I primi **4 bit del Program Counter**, concatenati a 26 bit del target, concatenati a 00.

Es. 

Per poter calcolare il valore massimo che un'istruzione jump può farci eseguire, dobbiamo inserire **26 bit a 1** – concatenando il target, I 26 bit, e gli ultimi "00", otteniamo l'indirizzo di memoria più alto al quale l'istruzione può fare un salto.

Jump and link

`jal target`

4	target
6	26

Esegui un jump all'istruzione nel target, e salva l'indirizzo dell'istruzione dopo nel registro \$ra.

Jump and link register

jalr rs, rd

0	Rs	0	rd	0	9
6	5	5	5	5	6

Esegui un jump incondizionato all'istruzione il quale indirizzo è nel registro rs. Salva l'indirizzo della prossima istruzione nel registro rd.

Jump register

jr js

0	Rs	0	8
6	5	15	6

Salta incondizionalmente all'istruzione il quale indirizzo è nel registro rs.

Trap if equal

teq rs, rt

0	Rs	Rt	0	0x34
6	5	5	10	6

Se il registro rs è uguale al registro rt, allora esegui un'eccezione Trap.

Trap if equal immediate

teqi rs, imm

1	Rs	0xc	imm
6	5	5	16

Se il registro rs è uguale al valore sign-extended imm, allora esegui un'eccezione Trap.

Trap if not equal

teq rs, rt

0	Rs	Rt	0	0x36
6	5	5	10	6

Se il registro rs non è uguale al registro rt, allora esegui un'eccezione Trap.

Trap if not equal immediate

teqi rs, imm

1	Rs	0xe	imm
6	5	5	16

Se il registro rs non è uguale al valore sign-extended imm, allora esegui un'eccezione Trap.

Trap if greater equal

tge rs, rt

0	Rs	Rt	0	0x30
6	5	5	10	6

Se il registro rs è più grande o uguale al registro rt, esegui un'eccezione Trap.

Trap if greater equal immediate

tgei rs, imm

1	Rs	8	imm
6	5	5	16

Se il registro rs è più grande o uguale al valore sign-extended imm, esegui un'eccezione Trap.

Trap if greater equal unsigned immediate

tgeiu rs, imm

1	Rs	9	imm
6	5	5	16

Se il registro rs è più grande o uguale al valore sign-extended imm, esegui un'eccezione Trap.

Trap if less than

tlt rs, rt

0	Rs	Rt	0	0x32
6	5	5	10	6

Se il registro rs è minore del registro rt, allora esegui un'eccezione Trap.

Unsigned trap if less than

tltu rs, rt

0	Rs	Rt	0	0x33
6	5	5	10	6

Se il registro rs è minore del registro rt, allora esegui un'eccezione Trap.

Trap if less than immediate

tlti rs, imm

1	Rs	a	imm
6	5	5	16

Se il registro rs è uguale al registro rt, allora esegui un'eccezione Trap.

Trap if less than immediate unsigned

tltiu rs, imm

1	Rs	a	imm
6	5	5	16

Se il registro rs è uguale al registro rt, allora esegui un'eccezione Trap.

Load address pseudoinstruction

la rdest, address

Carica l'*address*, e non i contenuti dell'area di memoria, nel registro rdest.

Load byte

lb rt, address

0x20	Rs	rt	Offset
6	5	5	16

Carica il byte all'*address* nel registro rt. Il byte è "sign-extended".

Load unsigned byte

lbu rt, address

0x24	Rs	rt	offset
6	5	5	16

Carica il byte all'*address* nel registro rt. Il byte è "sign-extended".

Load halfword

lh rt, address

0x24	Rs	rt	offset
6	5	5	16

Carica una halfword (16-bit) che è ad *address* nel registro rt. L'halfword è "sign-extended".

Load unsigned halfword

lhu rt, address

0x25	Rs	rt	offset
6	5	5	16

Carica una halfword (16-bit) che è ad *address* nel registro rt.

Load word

lw rt, address

0x23	Rs	rt	offset
6	5	5	16

Carica una word (32-bit) all'*address* nel registro rt.

Load word coprocessor 1

lwcl ft, address

0x31	Rs	ft	offset
6	5	5	16

Carica la word all'indirizzo *address* nel registro ft nella floating-point unit.

Load word left

lwl rt, address

0x22	Rs	rt	offset
6	5	5	16

Load the left bytes from the word at the possibly unaligned address into register rt.

Load word right

lwr rt, address

0x26	Rs	rt	offset
6	5	5	16

Load the right bytes from the word at the possibly unaligned address into register rt.

Load doubleword pseudoinstruction

ld rdest, address

Carica la doubleword a 64-bit all'indirizzo *address* nel registro rdest e rdest+1.

Unaligned load halfword pseudoinstruction

ulh rdest, address

Carica la halfword a 16-bit nell'indirizzo non allineato *address* nel registro rdest. La halfword è estesa.

Unaligned load halfword unsigned pseudoinstruction

ulhu rdest, address

Carica la halfword a 16-bit nell'indirizzo non allineato *address* nel registro rdest. La halfword è estesa.

Unaligned load word pseudoinstruction

ulw rdest, address

Carica una word (32-bit) nell'indirizzo non allineato *address* nel registro rdest.

Load linked

ll rt, address

0x30	Rs	rt	offset
6	5	5	16

Carica la word (32-bit) all'indirizzo *address* nell'indirizzo rt, e inizia un "atomic read-modify-write operation". L'operazione è completata grazie a un'istruzione *sc*, che fallisce se un altro processo scrive nell'area di memoria che contiene la parola caricata. Visto che SPIM non simula più processi insieme, questa operazione non fallisce mai.

Load word coprocessor 1

lwcl ft, address

0x31	Rs	ft	offset
6	5	5	16

Carica la word all'indirizzo *address* nel registro ft nella floating-point unit.

Store byte

sb rt, address

0x28	Rs	rt	offset
6	5	5	16

Salva il low byte dal registro rt all'indirizzo *address*.

Store halfword

sh rt, address

0x29	Rs	rt	offset
6	5	5	16

Salva il low halfword dal registro rt all'indirizzo *address*.

Store word

sw rt, address

0x2b	Rs	rt	offset
6	5	5	16

Salva il low byte dal registro rt all'indirizzo *address*. **L'offset deve essere specificato in esadecimale.**

Store word coprocessor 1

swc1 ft, address

0x31	Rs	ft	offset
6	5	5	16

Salva il valore floating-point nel registro ft del coprocessore floating-point all'indirizzo *address*.

Store double coprocessor 1

sdc1 ft, address

0x3d	Rs	ft	offset
6	5	5	16

Salva il valore double-word floating-point nel registro ft e ft+1 del coprocessore floating-point all'indirizzo *address*. Il registro ft deve essere pari.

Store word left

swl rt, address

0x2a	Rs	rt	offset
6	5	5	16

Salva i byte sinistri dal registro rt all'indirizzo non allineato *address*.

Store word right

swr rt, address

0x2e	Rs	rt	offset
6	5	5	16

Salva i byte destri dal registro rt all'indirizzo non allineato *address*.

Store doubleword pseudoinstruction

sd rsrc, address

Salva la quantità a 64 bit nel registro rsrc e rsrc+1 nell'indirizzo *address*.

Unaligned store halfword pseudoinstruction

ush rsrc, address

Salva la halfword bassa dal registro rsrc all'indirizzo *address* non allineato.

Unaligned store word

usw rsrc, address

Salva la word dal registro rsrc all'indirizzo *address* non allineato.

Store conditional

sc rt, address

0x38	Rs	rt	offset
6	5	5	16

Salva la word (32-bit) dal registro rt nella memoria all'indirizzo *address*, e completa un'operazione "atomic read-modify-write". Se questa operazione va a buon fine, la parola nella memoria è modificata, e il registro è impostato a 1. Se l'operazione fallisce perché un'altro processo ha scritto in un'area contenente la parola copiata, allora questa operazione non modifica la memoria e scrive 0 nel registro rt.

Datapath

Il datapath è il processore stesso, che può eseguire una serie di istruzioni aritmetico-logiche (add, sub, and, or, slt), interagire con la memoria (lw e sw), e di salto (anche dette control-flow instruction, usando beq e j).

Realizzazione di un datapath

Per la realizzazione di un datapath, occorre seguire una serie di istruzioni:

- Stabilire il set di istruzioni da implementare
- Identificare i componenti del datapath (alu, register file, etc)
- Stabilire la metodologia di clocking
- Assemblare il datapath, ed identificare i segnali di controllo
- Analizzare l'implementazione di ogni istruzione per determinare il setting dei segnali di controllo
- Assemblare la logica di controllo

Le parti del datapath

Il datapath è diviso in alcuni componenti:

- **Memoria:** contiene istruzioni e dati. Sulla base dell'indirizzo in input, restituisce in output l'istruzione o il dato letto.
- **Register file:** 32 registri che contengono i dati utilizzati nel corso dell'esecuzione delle istruzioni. Restituisce in output il contenuto dei registri letti (ReadRegister1 e ReadRegister2) e scrive, se il segnale di scrittura è attivo, il dato in input WriteData nel registro WriteRegister
- **ALU:** per tutte le istruzioni, incrementa il PC e calcola il valore di branch.
 - o Istruzioni r-type: esegue operazioni aritmetico-logiche su due operandi in funzione del function_code indicato dai 6 bit meno significativi dell'istruzione
 - o Accesso a memoria: calcola indirizzo di memoria a cui accedere
 - o Branch: confronta i due registri in input
- **Sign extended:** opera l'estensione di segno dai 16 bit in input ai 32 bit in output
- **Shift left 2:** opera uno shift a sinistra dei bit in input (con effetto di moltiplicare per 4 l'input)

I registri

- **Program Counter (PC):** contiene i 32 bit che indicano l'indirizzo dell'istruzione da eseguire
- **Instruction register:** contiene i 32 bit che corrispondono alla codifica dell'istruzione prelevata da memoria per l'esecuzione
- **Memory data register:** contiene il dato letto da memoria prima della sua scrittura nel registro destinazione
- **A e B:** contengono i valori letti dai registri del RegisterFile
- **ALUOut:** contiene output della ALU

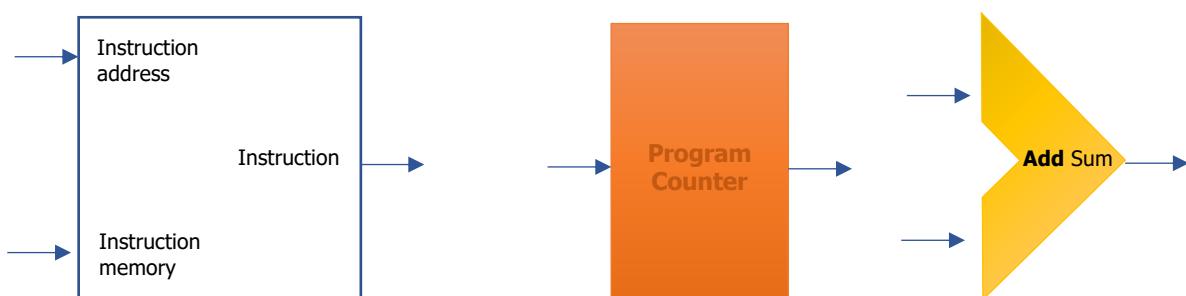
Eseguire un'istruzione

Per ogni istruzione, i primi due passi sono gli stessi:

1. Invia il *program counter* (PC) alla memoria che contiene il codice ed esegue il **fetch** dell'istruzione dalla memoria
2. Legge uno o due registri, usando i field dell'istruzione per scegliere i registri da leggere. Per le istruzioni *load word*, dobbiamo leggere soltanto un registro, ma per il resto delle istruzioni abbiamo bisogno di due.

Affinchè un'istruzione venga eseguito all'interno di un processore MIPS, c'è bisogno di tre categorie:

1. **Fetch:** legge l'istruzione dalla memoria e la salva nell'Instruction Register; l'indirizzo di memoria che indica l'istruzione da leggere si trova nel **Program Counter**. Dopo la lettura dell'istruzione in PC, viene incrementato di 4 per indicare la prossima istruzione da leggere usando l'ALU. Il Program Counter è un registro a 32-bit che sarà scritto alla fine di ogni ciclo di clock, e quindi non ha bisogno di un segnale di controllo write.



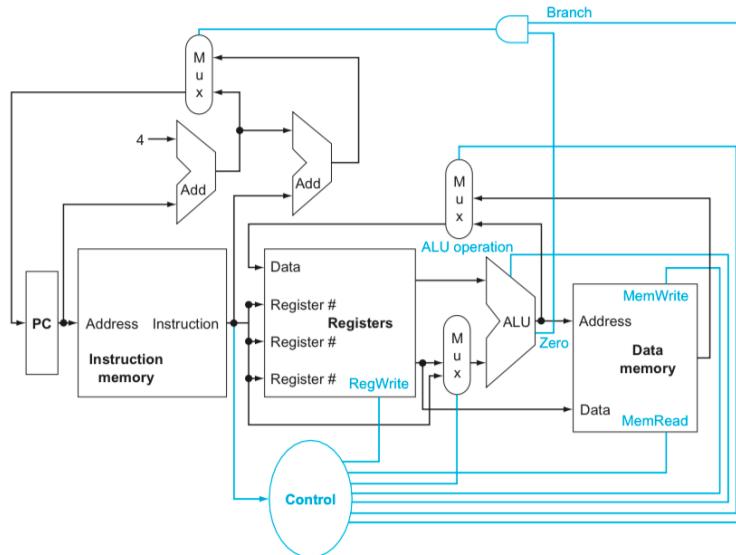
2. **Decode**: "decodifica" i vari campi dell'istruzione per decidere quali sono i passi necessari per l'esecuzione. Il MIPS legge i vari campi dell'istruzione, ed identifica il tipo di istruzione da eseguire. Esegue il calcolo del branch.
3. **Execute**: esegue i passi necessari per eseguire l'istruzione. Troviamo un register file, che contiene tutti i registri e due porte "read", una porta "write". Il register file manda sempre come output i contenuti del registro corrispondente al registro Read in input. Al contrario, quando deve scrivere, il registro deve essere esplicitamente indicato **modificando il segnale di controllo Write**. Le scritture sono sempre **edge-triggered**, quindi tutti gli input di scrittura devono essere validi al "clock edge": visto questo, questo design può leggere e scrivere valori in uno stesso ciclo di clock. *La lettura avrà il valore scritto nel ciclo di clock precedente, e il valore scritto sarà disponibile alla lettura il clock dopo*. All'interno della ALU, come prima, si usa per incrementare l'istruzione, e si usa l'operazione Zero per implementare i branches.



Note

All'interno del datapath, i dati possono arrivare da due o più sources. Per esempio, il valore scritto all'interno del PC può arrivare da **due adders**, i dati scritti all'interno del register file può arrivare dall'ALU o dalla memoria dati, e il secondo input all'ALU può arrivare da un registro, o da un *immediate* di un'istruzione. Per permettere questo, dobbiamo applicare un **multiplexor**, in modo che si possa scegliere da quale *sources* fare arrivare queste tipologie di dati.

Dobbiamo implementare anche un altro multiplexor e un'unità di controllo, poiché non tutte le operazioni eseguono le stesse operazioni. Un **control unit**, che ha come input l'istruzione, è usato per determinare come impostare le linee di controllo per l'unità funzionale, e due dei multiplexor. Il terzo multiplexor, che determina **quale istruzione del PC scrivere nel PC** (se PC+4, o il branch) è impostato, basato sull'output **Zero** della ALU, che è usata per effettuare una comparison di un'istruzione beq.



Convenzioni di logic design

Gli elementi del datapath nell'implementazione MIPS consistono di due diversi tipi di elementi logici che operano su valori dei dati ed elementi che **contengono uno stato**. Gli elementi che operano sui valori dei dati sono “**combinazionali**”, che vuol dire che i loro output dipendono soltanto sugli input correnti. Dato uno stesso input, un elemento combinazionale *produce sempre lo stesso output*, poiché non ha nessuna memoria interna.

Altri elementi all'interno del design non sono combinazionali, ma invece contengono uno stato, ovvero ha una memoria interna; questi sono chiamati **elementi di stato** poiché, se togliessimo l'elettricità dal computer, potremmo riprendere lo stato che c'era prima caricando gli elementi di stato con i valori che c'erano prima. Ha almeno due input e un output; gli input sono i valori dei dati che saranno scritti in un elemento e il clock (che determina *quando* i dati saranno scritti), mentre l'output **da il valore che era scritto il ciclo di clock precedente**.

L'implementazione MIPS usa due tipi di elementi di stato: *memorie* e *registri*, mentre il clock è usato per determinare quando un elemento di stato deve essere scritto; un elemento di stato può essere sempre letto. Questi componenti che contengono uno stato sono chiamati **sequenziali**, perché il loro output dipende sia sui suoi due input e i contenuti del suo stato.

Metodologia di clocking

Una metodologia di clocking definisce quando i segnali possono leggere, e quando possono essere scritti. È importante specificare questo aspetto, perché **se un segnale è scritto allo stesso tempo di quando è letto**, il valore della lettura potrebbe corrispondere al valore vecchio, al valore nuovo, o a un mix.

Per semplicità, si userà una **metodologia di clock edge-triggered**, che vuol dire che *ogni valore salvato in un elemento logico sequenziale sono aggiornati soltanto a un clock edge*, che è una veloce transizione da low a high, o viceversa. Visto che soltanto gli elementi di stato possono salvare un valore, ogni collezione di logica combinatoria deve avere i suoi input da un set di *elementi di stato*, e i suoi output devono essere scritti in un set di *elementi di stato*. Gli input sono *valori che erano stati scritti nel ciclo precedente*, gli output sono *valori che possono usati in un ciclo successivo*.



In questo design, ci sono due elementi di stato, intorno a un blocco di logica combinatoria, che funziona in un singolo ciclo di clock. Se un elemento di stato non è aggiornato in qualsiasi clock, allora ci deve essere un **write control signal** esplicito.

Questa metodologia ci permette di leggere i contenuti di un registro, mandare i valori attraverso la logica combinatoria, e poi scrivere quel dato nello stesso ciclo di clock. Non è importante che assumiamo che tutte le scritture siano eseguiti sul fronte rising o falling, perché gli input del blocco della logica combinatoria non può cambiare, eccetto sul clock edge scelto.

Singolo ciclo

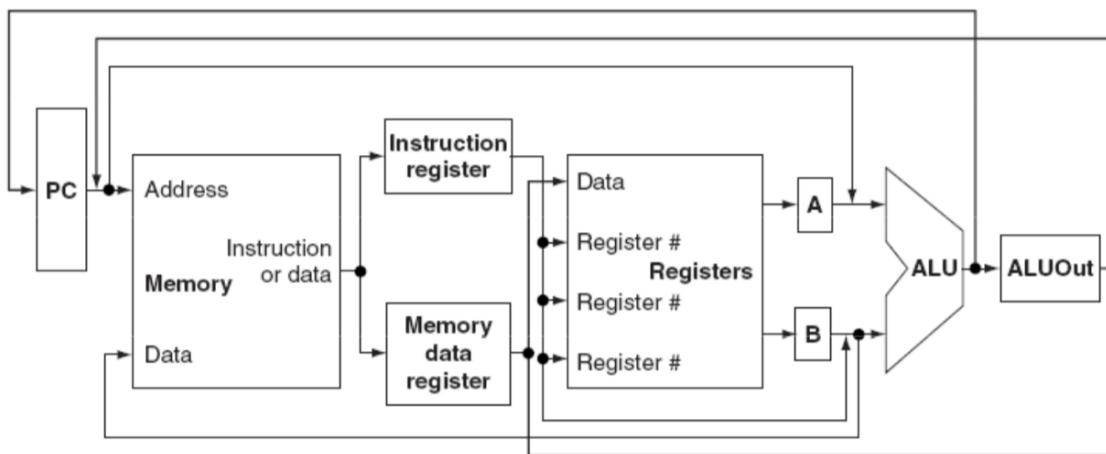
Ogni ciclo è un ciclo singolo di lunghezza fissa uguale al tempo necessario per eseguire l'istruzione più lunga. Ogni istruzione viene eseguita in un ciclo di clock, ma è uno spreco di tempo.

Multi-ciclo

Il ciclo di lunghezza fissa è più corto. Ogni istruzione viene eseguita in più cicli di clock (quindi, istruzioni di tipo diverso sono eseguite in un numero di cicli di clock diverso).

Le **unità funzionali vengono usate più volte** durante l'esecuzione della stessa istruzione in cicli di clock differenti (quindi, abbiamo meno replicazione). Si usano dei **registri aggiuntivi** per memorizzare i risultati parziali nell'esecuzione delle istruzioni:

- Memorizzano valori intermedi che vengono usati nel ciclo di clock successivo per continuare l'esecuzione della stessa istruzione (IR: Instruction Register, MDR: Memory Data Register, A – B: registri tra Register File e ingresso ALU, ALUout: output dell'ALU).
- L'ALU è usata non solo per le operazioni aritmetico-logiche, ma anche per calcolare l'indirizzo dei salti e incrementare il Program Counter. La memoria è usata sia per leggere le istruzioni che per leggere/scrivere i dati.



Eseguire le istruzioni in multi-ciclo

Questa è la lista di come le istruzioni vengono eseguite con un multi-ciclo:

- Ogni istruzione si esegue in più passi
- Ogni passo si esegue in un ciclo di clock
- Importante il bilanciamento della quantità di lavoro in ogni passo
- Una unità funzionale viene usata solo una volta durante lo stesso ciclo di clock
- Al termine di ogni ciclo di clock i valori intermedi vengono memorizzati nei registri addizionali e restano disponibili per il ciclo successivo

Per eseguire qualsiasi istruzione, dobbiamo iniziare eseguendo il fetch dell'istruzione a memoria. Per preparare l'esecuzione della prossima istruzione, dobbiamo anche incrementare il PC cosicché può puntare alla prossima istruzione, ovvero 4 byte dopo.

R-format

Prendendo come esempio le istruzioni R-format, tutte le istruzioni di questo tipo leggono due registri, esegue un'operazione con l'ALU sui contenuti dei registri, e scrivere i risultati in un registro. I 32 registri di "general purpose" sono all'interno di una struttura chiamata **register file**; è una collezione di registri in cui ogni registro può essere letto o scritto, specificato dal numero di registri nel file.

Questo tipo di istruzioni ha tre operandi, quindi avrà bisogno di leggere due parole di dati dal register file, e scrivere una parola di dati nel register file per ciascuna istruzione. Per ciascuna parola di dati che deve essere letta dai registri, abbiamo bisogno di mandare un input al register file, per specificare che registro deve essere letto, e un output dal register file che porterà il valore che è stato letto dai registri. Per scrivere poi una parola di dati, avremmo bisogno di un input per specificare il numero di registro, e un altro per i dati che devono essere scritti all'interno del registro.

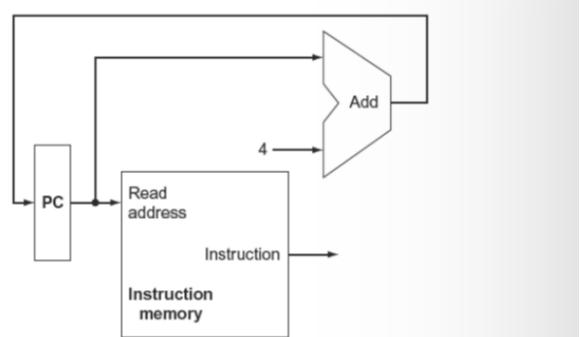


FIGURE 4.6 A portion of the datapath used for fetching instructions and incrementing the program counter. The fetched instruction is used by other parts of the datapath.



a. Registers

b. ALU

Il register file da sempre l'output dei contenuti di qualsiasi registro è scritto nel Read register input. Writes, invece, sono controllati dal registro di controllo write, che deve essere impostato su **asserted** per fare in modo che sia scritto. Alla fine, quindi, abbiamo bisogno di quattro input e due output.

Aggiuntivamente, avremmo bisogno anche di un'unità per eseguire un'operazione **sign-extend** dell'offerta field a 16-bit nell'istruzione, a un valore 32-bit signed, e una data memory unit da cui leggere, o in cui scrivere. La data memory deve essere scritta con le istruzioni **store**; quindi, data memory ha dei controlli di segnale per read e write, un imput per l'indirizzo, e un input per I dati che devono essere scritti in memoria.

Branch format

In un'istruzione branch, ci sono due dettagli a cui dobbiamo porre attenzione:

- L'instruction set architecture specifica che la base per il calcolo dell'indirizzo branch è l'indirizzo dell'istruzione subito dopo il branch. Visto che usiamo **PC+4** nel segmento fetch, è facile usare questi valori come la base per sapere quale è il target address di un'indirizzo branch.
- L'architettura definisce anche che l'offset field è spostato di sinistra di due bit, cosicché ha un offset di una word; questa operazione incrementa il range del field offset di un fattore di 4.

Dobbiamo anche determinare se l'istruzione dopo è l'istruzione che segue sequenzialmente, o l'istruzione che è al branch target address. Quando questa condizione è vera, allora il *branch target address diventa il nuovo PC*, e diciamo quindi che il **branch è stato eseguito**. Se gli operandi non sono uguali, allora il PC incrementato dovrà rimpiazzare il PC corrente, e quindi il branch non è stato eseguito. Quindi, il datapath del branch fa due operazioni: analizza l'indirizzo target del branch, e mette in confronto i contenuti dei registri.

Visto che l'ALU da un segnale di output che indica se il risultato è 0, possiamo mandare i due operandi dei registri alla ALU con il control set, per fare un'operazione di sottrazione. Se il segnale Zero della ALU è asserted, allora sappiamo che i due valori sono uguali.

Fetch

Durante il fetch, viene eseguito il fetch dell'istruzione, più incrementazione del Program Counter.

Operazioni	Segnali di controllo
IR \leftarrow M [PC]	Per leggere della memoria: <i>MemRead</i>
PC \leftarrow PC+4	Per scrivere IR: <i>IRWrite</i>

	Per indicare l'indirizzo da dove leggere dalla memoria: <i>IorD</i>
	Per incrementare il PC: <i>ALUSrcA, ALUSrcB, ALUOp</i>
	Per salvare il nuovo valore del PC in PC: <i>PCWrite</i>

Decode

Durante la decodifica, decodifica l'istruzione, legge i registri, e calcola l'indirizzo per un eventuale branch.

Operazioni	Segnali di controllo
A <- Reg[IR[25:21]]	Per il calcolo di un eventuale indirizzo di branch: <i>ALUSrcA, ALUSrcB, ALUOp</i>
B <- REG [IR[20:16]]	
ALUOut <- PC+(sign-extend (IR[15:0]) << 2	

Execute

Completa R_type, oppure accede alla memoria. Solo per l'istruzione lw, viene eseguita la scrittura del registro. **L'istruzione più lunga si esegue in 5 passi**, mentre la più corta è in 3 passi.

Per le istruzioni lw e sw:

Operazioni	Segnali di controllo
ALUOut <- A+(sign-extend (IR[15:0]))	Per il calcolo di un eventuale indirizzo di branch: <i>ALUSrcA, ALUSrcB, ALUOp</i>

\ \ \ /

Operazioni	Segnali di controllo
MDR <- M[ALUOut]	Per indicare l'indirizzo di memoria: <i>IorD</i>
M[ALUOut] <- B	Per leggere dalla memoria (lw): <i>MemRead</i>

\ \ \ / (lw)

Operazioni	Segnali di controllo
Reg[IR[20:16]] <- MDR	Per poter scrivere nel Register File: <i>RegWrite</i>
	Per indicare il registro da scrivere: <i>RegDest</i>
	Per scrivere il valore dalla memoria: <i>MemToReg</i>

Per le istruzioni R-Type aritmetico-logiche:

Operazioni	Segnali di controllo
ALUOut <- A op B	Per il calcolo aritmetico o logico: <i>ALUSrcA, ALUSrcB, ALUOp</i>

\ \ \ /

Operazioni	Segnali di controllo
Reg[IR[15:11]] <- ALUOut	Per poter scrivere nel Register File: <i>RegWrite</i>
	Per indicare il registro da scrivere: <i>RegDest</i>
	Per scrivere il valore nel ALUOut: <i>MemToReg</i>

Per le istruzioni beq:

Operazioni	Segnali di controllo
If A == B then PC <- ALUOut	Per la comparazione tra A e B: <i>ALUSrcA</i> , <i>ALUSrcB</i> , <i>ALUOp</i>
	Per scrivere PC: <i>PCWriteCond</i> , <i>PCSource</i>

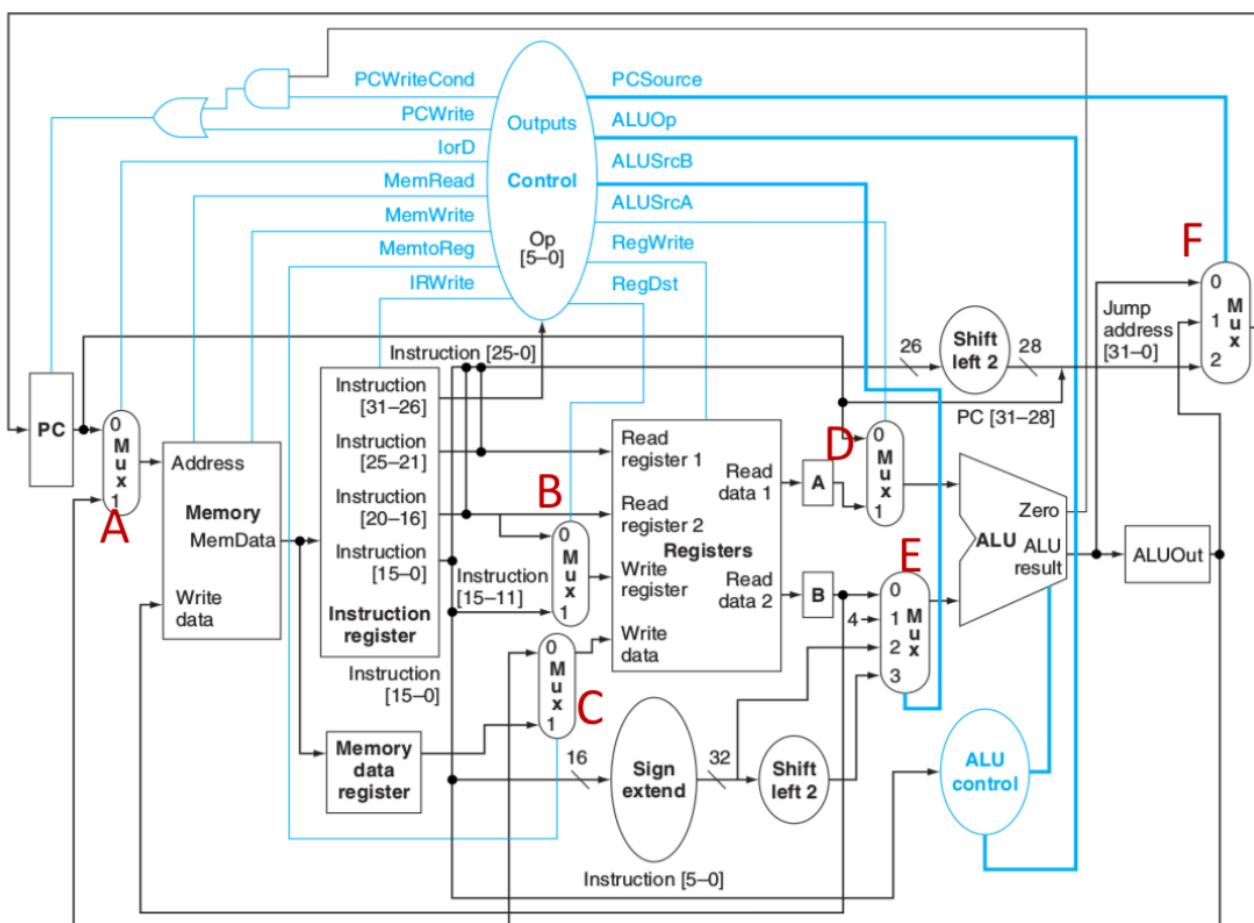
Per le istruzioni jump:

Operazioni	Segnali di controllo
PC <- (PC[31:28], IR[25:0]<<2)	Per scrivere PC: <i>PCWrite</i> , <i>PCSource</i>

Creare un datapath

Gli elementi principali di cui abbiamo bisogno sono una unità di memoria per immagazzinare le istruzioni di un programma, e per mostrare l'istruzione, dato un indirizzo. Abbiamo bisogno del **program counter** (PC), che è un registro che tiene conto dell'indirizzo dell'istruzione corrente. Abbiamo bisogno infine di un **adder** per incrementare il PC per andare all'istruzione dopo; questa è combinatoria, può essere costruita da una ALU collegando le control lines, cosicché il controllo specifica sempre un'operazione di aggiunta.

Funzioni del datapath



Dato un datapath, possiamo distinguere dei multiplexor:

- **A**: seleziona l'indirizzo di memoria a cui accedere tra PC (per leggere), o output della ALU (esecuzione di load)
- **B**: seleziona un gruppo di bit dell'Instruction Register che indica il registro in cui scrivere per i due tipi di istruzione I-type (IR[20:16]) e R-type(IR[15:11])

- **C**: seleziona la sorgente per il dato da scrivere in memoria ALUOut (per istruzioni r-type) e MemoryDataRegister (per istruzioni load)
- **D**: seleziona il primo operando dell'operazione aritmetico-logica eseguita dalla ALU tra Program Counter e registro A
- **E**: seleziona il secondo operando dell'operazione aritmetico-logica eseguita dalla ALU (B per r-type, 4 per aggiornamento PC, sign-extended IR[15:0] per istruzioni di accesso a memoria lw/sw, sign-extended 2-shifted per j)
- **F**: seleziona il valore dell'aggiornamento del Program Counter. L'output dell'ALU è PC+4, il contenuto di ALUOut è 2-shifted 26-bit beq field, jump target address (IR[25:0] shifted left 2 bits e concatenato con PC+4[31:28])

Il controllo ALU

Nel MIPS, l'ALU definisce la combinazione di 4 input di controllo. In base alla classe dell'istruzione, l'ALU avrà bisogno di fare una di queste cinque funzioni.

Per le istruzioni **load word** e **store word**, l'ALU verrà usata per calcolare l'indirizzo di memoria addizionando. Per le istruzioni **r-type**, l'ALU avrà bisogno di fare un AND, OR, subtract, add, slt (set on less than), in base al valore del field 6-bit function.

ALU control lines	Function
0000	AND
0001	OR
0010	Add
0110	Subtract
0111	Set on less than
1100	NOR

Possiamo generare i 4 bit necessari come input di controllo per l'ALU usando un piccolo control unit che ha come input il function field dell'istruzione, e un field per il controllo di due bit, chiamato **ALUOp**. Questo indica se l'operazione che deve essere effettuata deve essere add (00, per load e store), subtract (01, per beq), o determinata dall'operazione nel field *funct* (10).

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

Disegnando il Main Control Unit

Possiamo iniziare identificando i field di un'istruzione, e le linee di controllo di cui avremmo bisogno. Ci sono alcune osservazioni che dobbiamo fare dei formati delle istruzioni:

- L'op field è chiamato **opcode** ed è sempre nei bit 31:26. È riferito come *Op[5:0]*
- I due registri che devono essere letti sono sempre specificati da rs e rt, alle posizioni 25:21 e 20:16. Questo è vero per le istruzioni R-type, branch, equal e store
- Il registro base per load e store è sempre nella posizione 25:21 (Es)
- L'offset a 16 bit per il branch equal, load e store è sempre nella posizione 15:0
- Il registro di destinazione può essere in due spazi. Per il load è in 20:16 (rt), per un'istruzione R-type è in 15:11 (rd). Avrà bisogno quindi di un multiplexor per selezionare in che posizione dobbiamo prendere.

Possiamo quindi aggiungere un instruction label, e un altro multiplexor al datapath; visto che ogni multiplexor ha due input, ognuno di loro ha bisogno di un a singola linea di controllo.

La tabella di verità dei 4 bit di controllo della ALU, chiamato **Operation**, è la seguente:

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

Inoltre, abbiamo sette linee di controllo a singolo bit, e 2 bit per l'ALUOp. Questi sono utili per definire cosa fanno gli altri sette segnali di controllo, prima di determinare come impostare questi segnali di controllo durante l'esecuzione dell'istruzione. Di seguito, possiamo vedere una tabella che illustra cosa fa ciascuna di questa control line:

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register destination number for the Write register comes from rt (20:16)	The register destination number for the Write register comes from rd (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

La control unit può impostare tutti meno uno dei controlli di segnale, basandosi soltanto sull'opcode dell'istruzione. Il **PCSrc** è l'eccezione; questa control line deve essere su asserted, se l'istruzione is *branch on equal*, e se l'output Zero della ALU (usata per fare i confronti) è anch'essa su asserted. Per generare questo segnale, avremmo bisogno di eseguire un AND tra un segnale dalla control unit chiamato *Branch*, e il segnale *Zero* della ALU. Gli altri nove segnali di controllo possono essere quindi impostati sulla base dei sei segnali di input, che sono gli opcode 31:26.

Ciascuna operazione cambierà queste control lines per poter fare un'operazione diversa:

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	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

Input ed output

Esistono oltre a CPU e RAM delle periferiche, il cui bus è sempre "shared" con la memoria. L'interfaccia è fatta sempre con indirizzi mappati; bisognerà fare in modo che la periferica mandi dati alla ram passando dal processore, o senza passare dal processore. Il processore comunque **rimane sempre il referente principale**.

L'I/O è gestito dal controllo di un programma. La collaborazione tra cpu e periferica sarà fatta tramite registri della periferica. Il processore decide e si occupa sia del controllo che del trasferimento dati, mentre **la periferica ha un ruolo passivo**. La cpu predisponde il controllore della periferica all'esecuzione dell'I/O, la CPU si ferma e **interroga il registro di stato della periferica**, in attesa che sia pronta.

Periferiche

Le periferiche sono dispositivi per I/O di informazioni; sono collegate alla CPU tramite il **bus di sistema e/o interfacce**. Queste interfacce sono standardizzate per la comunicazione, e hanno una componente *hardware* (es. Controllore della periferica), e una componente *software* (es. Driver). Contengono comunque registri di dati (o buffer), e hanno associato dei registri di stato.

Periferiche mappate in memoria

Una parte della memoria è dedicata alla gestione delle periferiche. Ogni periferica collegata al sistema ha uno spazio riservato nella memoria, identificato da un

identificatore univoco; questo spazio è usato per comunicare con le periferiche.

Gli elementi di controllo delle periferiche, che sono i **registri dell'interfaccia**, sono mappati nella memoria, e non accessibili direttamente da un programma utente. Pertanto, devono passare per forza dal sistema operativo.

L'accesso ai registri delle periferiche è simile all'accesso alla memoria principale, da parte della CPU; vengono usate **istruzioni lw e sw**, per leggere e scrivere dati nei registri delle periferiche riservate. Questo garantisce che i programmi utente non accedano direttamente ai registri, e che la comunicazione con le periferiche avvenga in modo controllato.

Registri di stato della periferica

Questo è un componente che **rappresenta lo stato corrente della periferica**; viene letto dalla CPU per ottenere informazioni sull'avanzamento delle operazioni nella periferica; "dice" quindi se la periferica è pronta per ricevere dati, o se ci sono dati disponibili per il trasferimento.

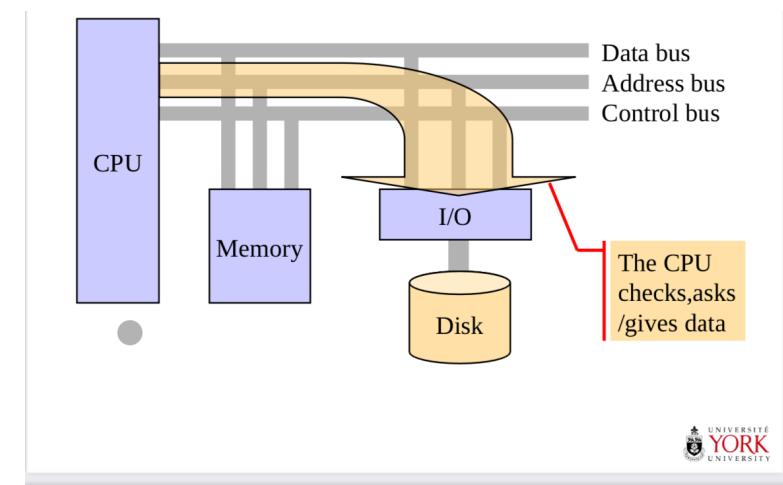
Il registro dei **dati** della periferica è responsabile per la memorizzazione dei dati in ingresso o in uscita, a seconda del tipo di periferica; viene usato per trasferire dati tra la periferica e la CPU.

Passi di I/O

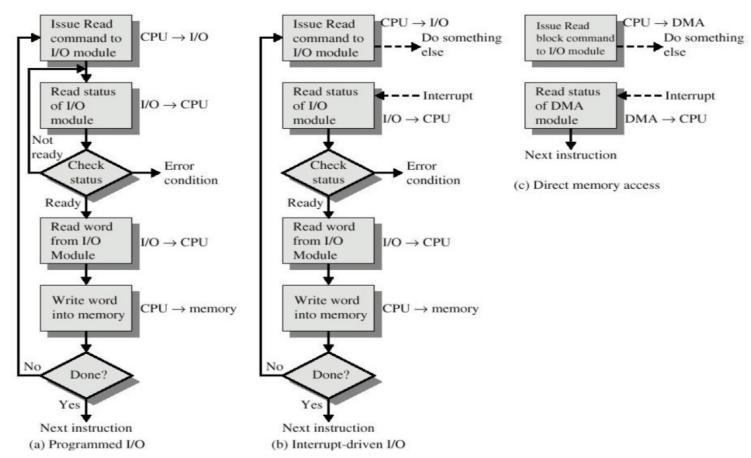
La CPU comunica con la periferica seguendo questi passaggi:

1. La CPU **interroga lo stato** della periferica per conoscere il suo stato corrente
2. La periferica **restituisce il suo stato** alla CPU, che può essere ad esempio pronta a trasmettere o ricevere dati.
3. Se la periferica è **pronta a trasmettere o ricevere dati**, la CPU richiede il trasferimento dei dati.
4. La CPU invia i dati alla periferica se deve trasmettere dati, o riceve i dati dalla periferica se deve ricevere dati.

In questo modo, la CPU comunica in modo bidirezionale.



UNIVERSITÉ
YORK
UNIVERSITY



Banda passante e latenza

Sono misure che permettono di **valutare l'efficienza della gestione delle operazioni** di I/O. La **banda passante** rappresenta la quantità di dati che si può trasferire per unità di tempo, e rappresenta una *misura di flusso*. La **latenza** rappresenta il tempo che intercorre tra l'istante in cui una periferica è pronta per il trasferimento, e l'istante in cui il dato viene trasferito; è quindi una misura di tempo.

Interrupt

Le interruzioni guidate da hardware sono tipicamente causate da dispositivi hardware esterni, come periferiche di input/output che sono collegate al processore attraverso *linee di controllo di bus*. Quando una periferica è pronta per eseguire un'operazione, **invia un segnale di interruzione alla CPU** tramite una specifica linea di interruzione.

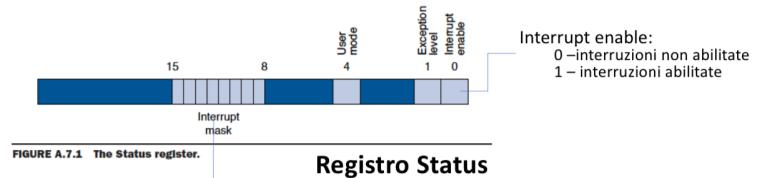


FIGURE A.7.1 The Status register.

Registro Status

Interrupt mask – interrupt abilitati

Un bit per ognuna delle 6 linee delle periferiche e 2 bit per interrupt a livello sw

Una volta che la CPU rileva il segnale di interruzione, se le interruzioni sono abilitate nel sistema, **interrompe il flusso normale di esecuzione**, e inizia a gestire l'interruzione. Durante questa fase, la CPU salva lo stato corrente dell'esecuzione del programma, al fine di poter riprendere l'esecuzione. La CPU poi può avviare la procedura di risposta all'interruzione specifica; questo può comportare l'esecuzione di operazioni richieste dalla periferica, l'aggiornamento dei registri di stato, o l'avvio di altre azioni correlate. Una volta completata la procedura di risposta, la CPU riprende l'esecuzione del programma interrotto, ripristinando lo stato precedente. Questo consente alla CPU di gestire efficacemente e tempestivamente le richieste delle periferiche esterne, migliorando l'efficienza e consentendo un'elevata velocità di trasferimento dei dati.

Interrupt abilitati

All'interno del registro di maschera degli interrupt, presente nel registro di stato, ogni bit corrisponde all'abilitazione di una specifica linea di interruzione collegata a una o più periferiche. **Se il bit è impostato a 1**, significa che L'interrupt corrispondente è abilitato, e la periferica che solleva l'eccezione può generare l'interruzione, permettendo alla CPU di percepire l'evento.

Se il bit è impostato a 0, l'interrupt corrispondente non è abilitato. Anche se la linea di interruzione viene attivata dalla periferica, la CPU non genererà l'eccezione corrispondente, e non interromperà il flusso di esecuzione corrente.

Quando si verifica un'interruzione abilitata, e si solleva un'eccezione, nel registro **Cause** viene impostato a 1 il **bit corrispondente all'interruzione specifica**. I 5 bit del codice di eccezione vengono impostati a 0, indicando che l'*eccezione è causata da un'interruzione* piuttosto che da altri eventi anomali o errori.

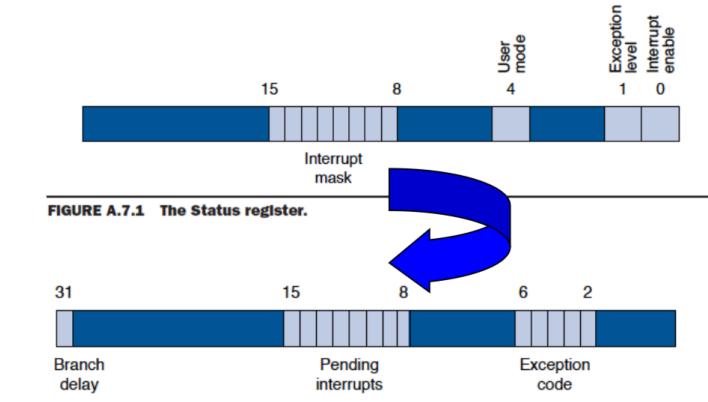
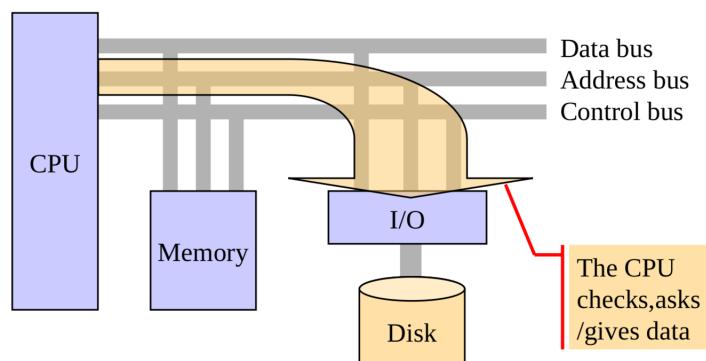


FIGURE A.7.2 The Cause register.

Tecniche di gestione I/O

Programmed I/O

Nel caso in cui l'I/O sia gestito dal controllo di programma, la CPU assume un ruolo attivo e si occupa sia del **controllo e trasferimento dei dati della periferica**. La CPU predisponde il controllore della periferica per eseguire l'I/O, entra in uno stato di attesa e interroga periodicamente il registro di stato della periferica.



Uno dei vantaggi è che **la CPU può rispondere immediatamente alla disponibilità** della periferica, ma uno svantaggio è che la CPU rimane bloccata in uno stato di "busy waiting", fino a quando la periferica non è pronta; perde quindi tempo di elaborazione.

Questo approccio offre una **banda passante elevata** (la CPU trasferisce immediatamente i dati e la gestione della periferica richiede poche istruzioni), ma la **latenza potrebbe essere elevata** nel caso in cui la periferica richieda un tempo considerevole per diventare pronta. Nel caso peggiore, la CPU potrebbe attendere un ciclo completo. I bit usati sono il bit di stato, bit di controllo, bit di dati, bit di interrupt.

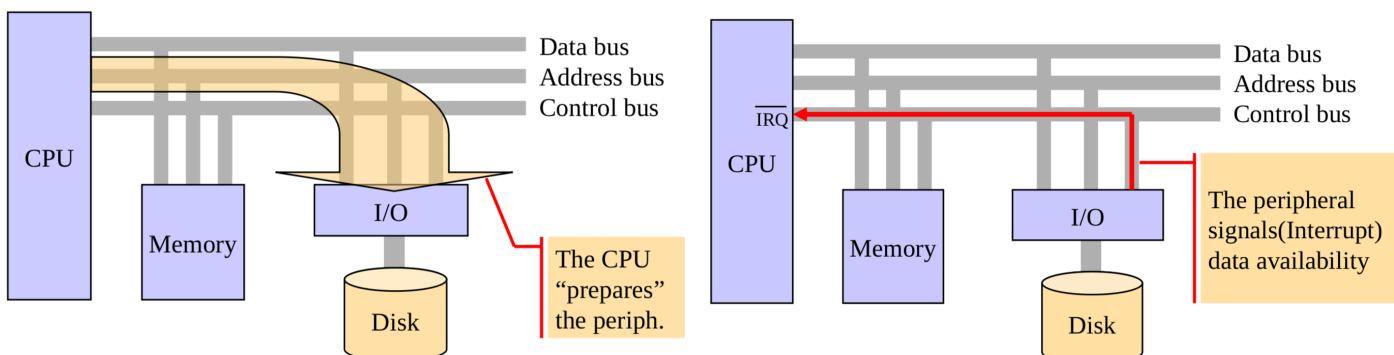
I/O guidato da interrupt

In questo, si verifica un **evento asincrono** che interrompe il normale funzionamento del processore. La periferica segnala alla CPU che richiede attenzione mediante un segnale sul bus di controllo, noto come **interrupt request** (IRA). Quando il processore rileva tale segnale durante una *fase di fetch*, risponde alla periferica con un segnale di **interrupt acknowledge** (IRQ ACK). La CPU quindi interrompe l'esecuzione del programma corrente, salvando il contesto dell'esecuzione del programma, in modo da poter riprenderne l'esecuzione. Viene quindi eseguita la procedura di risposta all'interruttore, che può essere una *routine predefinita o specifica* per la periferica.

Una volta terminata l'esecuzione della procedura di interrupt, la CPU riprende l'esecuzione del programma interrotto dal punto in cui si era interrotto, e il programma utente può continuare la sua esecuzione senza essere influenzato dall'interruttore.

Un vantaggio è che **la CPU non è più coinvolta in una fase di busy waiting**; viene solo notificata quando una periferica richiede effettivamente attenzione; uno svantaggio è che **la CPU deve comunque gestire le operazioni di trasferimento dati** associate all' I/O.

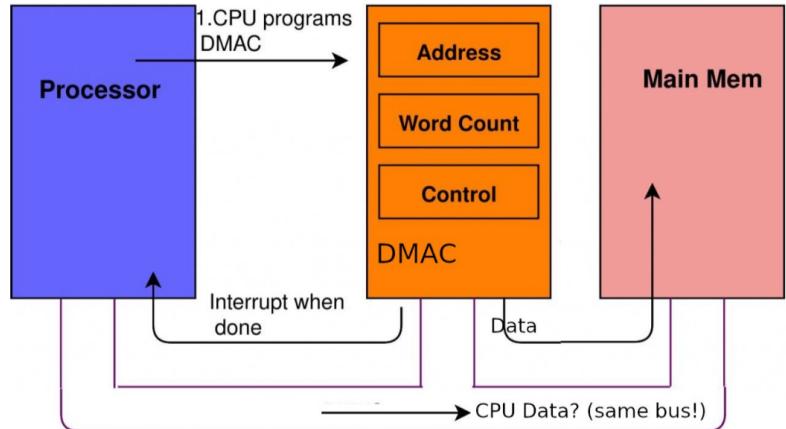
La **banda passante è generalmente inferiore**, perché il trasferimento di ogni dato richiede più tempo; questo è dovuto al fatto che la CPU deve gestire le operazioni di trasferimento durante l'evento di interrupt. Può avere una **latenza maggiore**, poiché richiede più operazioni da eseguire. Solo dopo aver completato la procedura, la CPU riprende l'esecuzione del programma interrotto.



Direct Memory Access

Per evitare che la CPU sia direttamente coinvolta nel trasferimento dei dati, è stato introdotto il **DMA**, che permette alla periferica di accedere direttamente alla memoria centrale senza l'intervento continuo della CPU. La periferica diventa **autonoma** nell'accesso alla memoria.

Richiede l'uso di due registri aggiuntivi per ogni periferica, oltre ad altri registri come il registro di stato e il registro dei dati. Uno di questi registri indica **l'indirizzo di memoria da cui o a cui trasferire dati**, mentre l'altro registra la quantità di dati da trasferire. Anche questi due registri sono mappati nella memoria del sistema, consentendo alla CPU di **comunicare con la periferica per configurare i trasferimenti DMA**.



Una volta che la periferica ha completato il trasferimento dei dati, invia un segnale di interrupt alla CPU per segnalare il completamento; la CPU può essere informata del termine del trasferimento, senza dover eseguire istruzioni aggiuntive o controllare continuamente lo stato.

Il vantaggio è la **massima banda passante**, in quanto la CPU non deve eseguire alcuna istruzione durante il trasferimento dei dati. Questo consente un rapido ed efficiente spostamento dei dati tra la periferica e la memoria. Inoltre, **la latenza è minima** poiché non sono necessarie istruzioni CPU per gestire il trasferimento dei dati.

Memoria mappata:

Dalla periferica arriva un segnale di ready; se è ready, allora la CPU scrive o legge quello di cui la periferica ha bisogno.

Interruzione dell'esecuzione

Durante l'esecuzione delle istruzioni, si possono verificare degli eventi inattesi che interrompono l'esecuzione temporaneamente.

Eccezione

È un evento sincrono rispetto al programma in esecuzione, generato all'interno del processore, ed è provocato da problemi nell'esecuzione di un'istruzione. La condizione di eccezione deve essere risolta da un **gestore di eccezioni**. Se la condizione di eccezione è risolubile, allora il programma può riprendere l'esecuzione; altrimenti, il programma termina prima della sua fine.

Gestire un problema

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

- Interruzione dell'esecuzione del programma corrente
- Salvataggio parziale dello stato di esecuzione corrente, per riprendere eventualmente l'esecuzione del programma corrente, se possibile
- Salto a una routine del sistema operativo per gestire l'eccezione/interruzione
- Esecuzione della routine del sistema operativo
- Se possibile, ripristino dello stato di esecuzione del programma, e continuazione dell'esecuzione del programma
-

Gestire l'eccezione

Per poter gestire l'eccezione, abbiamo due possibili soluzioni:

- Usando un indirizzo fisso, o un **registro dedicato**. Il controllo della CPU, prima di saltare all'handler del sistema operativo, deve salvare in un registro interno un identificatore numerico del tipo di eccezione verificatosi. L'handler accederà al registro interno, per determinare la causa dell'eccezione.
- Usando delle **interruzioni vettorizzate**. Esistono handler diversi per eccezioni/interruzioni diversi. Il controllo della CPU sceglie l'handler corretto, saltando all'indirizzo corretto. A questo scopo, viene predisposto un vettore di indirizzi, uno per ogni tipo di eccezioni/interruzioni, da indirizzare tramite il codice numerico dell'eccezione

Nel MIPS, viene adottata la prima soluzione, usando un registro chiamato **Cause**, per memorizzare il motivo dell'eccezione. L'indirizzo dell'istruzione corrente che ha causato l'eccezione viene salvato nel registro **Exception Program Counter** (EPC). Per ora, consideriamo solo come casi di eccezione le situazioni quando si ha un'*istruzione non valida* (generata di controllo a valle della decodifica dell'istruzione), e un *overflow* (generata dall'ALU nella fase execute di un'istruzione R-type).

Questi sono i passaggi che esegue:

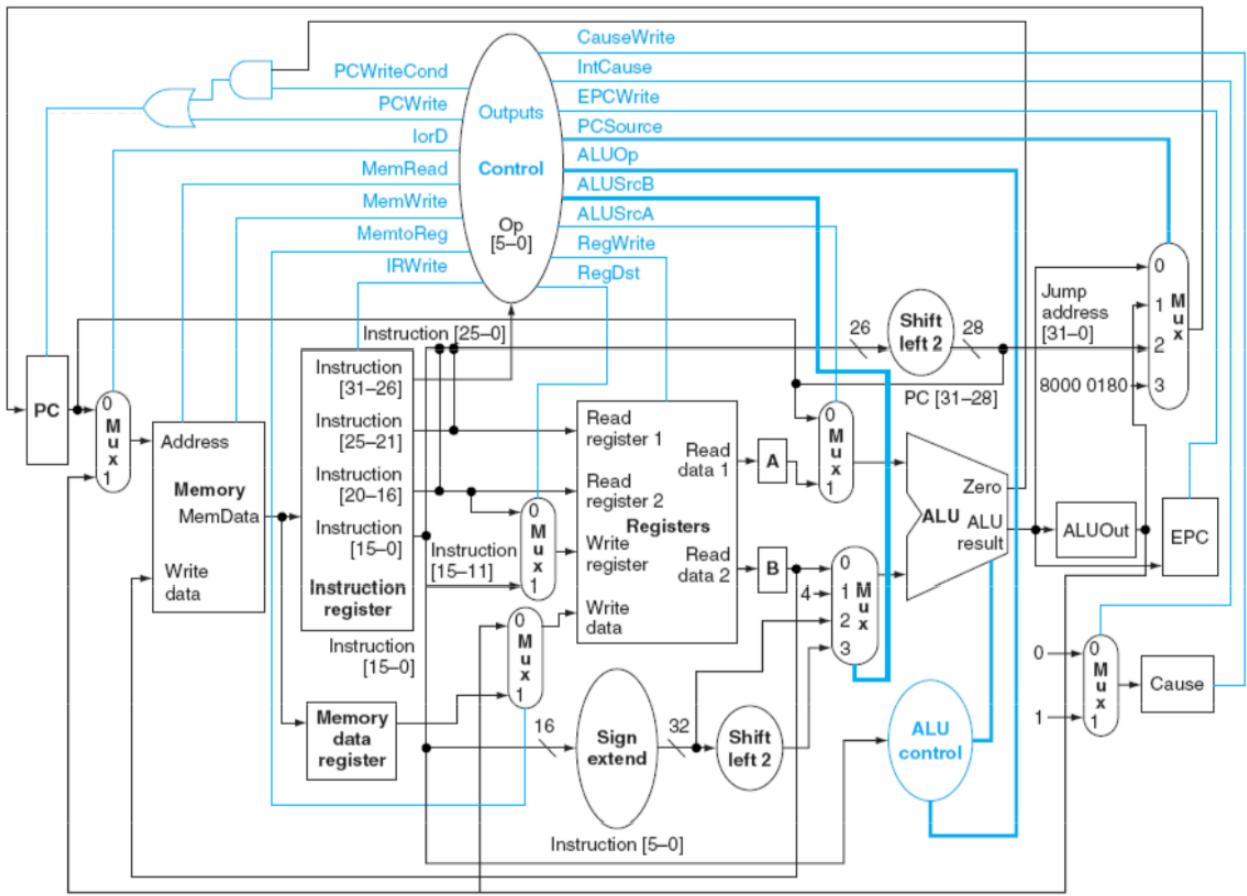
- **Individua l'evento inatteso**, la causa dell'eccezione, e la salva in un registro dedicato denominato Cause
- **Interrompe l'esecuzione corrente**
- **Salva** l'indirizzo dell'istruzione corrente nel EPC (**EPC <- PC-4**)
- **Salta a un gestore delle eccezioni** del sistema operativo, che si trova a un indirizzo fisso per gestire l'eccezione.

In aggiunta, se l'eccezione è un'**istruzione non valida**, allora l'unità di controllo rileva tale eccezione sulla base dell'OPCODE di un'istruzione. Nel caso di un overflow:

- Segnale che arriva all'unità di controllo della ALU
- Bisogna aggiungere 2 nuovi stati alla FSM del datapath:
 - Salva in EPC il valore PC-4
 - Salva nel registro Cause la causa dell'eccezione (0 o 1, poiché abbiamo due esempi)
 - Salva in PC l'indirizzo del gestore delle eccezioni

Quando succede un'eccezione, il MIPS non salva nessun altro registro oltre al PC. È compito della routine salvare altre porzioni dello stato corrente del programma, se necessario. Esistono, inoltre, CPU dove questo salvataggio viene prima di saltare alla routine.

Inoltre, sarà opportuno effettuare qualche modifica al datapath, come **introdurre nuove linee di controllo** per gestire le eccezioni, che saranno CauseWrite, IntCause, ed EPCWrite.



10

Gestione della memoria

Avere a disposizione una quantità illimitata di memoria e contemporaneamente veloce sarebbe la situazione ideale, ma questo è ovviamente impossibile.

Un programma non accede a tutte le sue istruzioni e a tutti i suoi dati contemporaneamente con la stessa probabilità; la soluzione per questo è una **gerarchia di memoria**, che consiste in un insieme di livelli di memoria, ciascuno caratterizzato da una diversa velocità e dimensione. A parità di capacità, le memorie più veloci hanno un *costo più elevato* per singolo bit di quelle più lente.

La **memoria interna** alla CPU è costituita dai registri, ed è caratterizzata da alta velocità e limitate dimensioni. La **memoria centrale** è caratterizzata da dimensioni molto maggiori della memoria interna alla CPU, ma con tempo di accesso più elevati. È accessibile in modo diretto tramite indirizzi; nei sistemi attuali, un livello di *memorie cache* è stato inserito tra CPU e memorie centrali. Le **memorie secondarie** sono ad alta capacità, bassi costi, e non volatili.

Principio di località

Un programma, in un certo istante di tempo, accede soltanto a una porzione relativamente piccola del suo spazio di indirizzamento.

Rappresenta la base del comportamento dei programmi in un calcolatore, e ci sono due tipi di località:

- **Temporale:** quando si fa riferimento a un elemento c'è la tendenza a fare riferimento allo stesso elemento dopo poco tempo
- **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

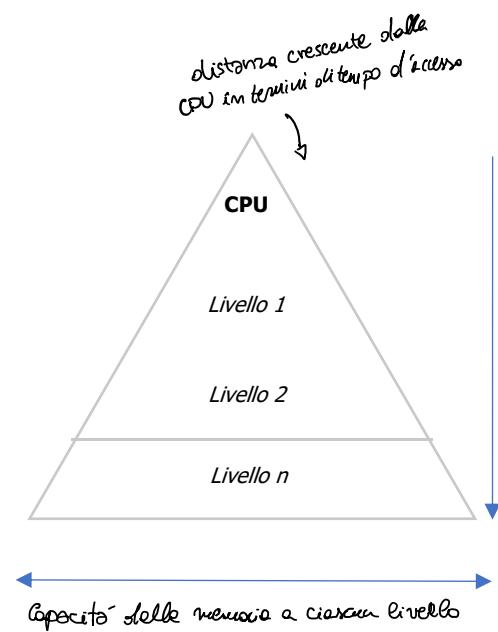
I programmi non vedono la gerarchia, ma referenziano i dati come se fossero sempre in memoria centrale. Questo principio viene sfruttato strutturando la memoria in modo **gerarchico**, in base alla velocità e alla dimensione.

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

Definizioni

Ecco una selezione di definizioni utili:

- **Blocco/linea:** la più piccola quantità di informazione che può essere presente/assente in una gerarchia di memoria
- **Hit (successo nell'accesso):** l'informazione richiesta dal processore si trova in uno dei blocchi nel livello superiore di memoria
 - o **Hit rate (frequenza di hit):** frazione degli accessi alla memoria nei quali l'informazione richiesta è stata trovata nel livello superiore di memoria
 - o **Tempo di hit:** tempo di accesso al livello superiore della memoria; comprende anche il tempo necessario a stabilire se il tempo di accesso si risolva in un successo o in un fallimento
 - o **Frequenza di hit:** accessi risolti con successo/numero totale di accessi
- **Miss (fallimento nell'accesso):** il dato non è presente nel livello immediatamente superiore, ed occorre accedere al livello più distante
 - o **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)
 - o **Tempo di miss:** il tempo necessario a sostituire un blocco del livello superiore con un nuovo blocco dal livello inferiore della gerarchia, e trasferire i dati di questo blocco al processore
 - o **Frequenza di miss:** accessi risolti senza successo/numero totale di accessi
- **Remind:** una gerarchia di memoria può essere controllata da più livelli, ma i dati vengono trasferiti solo tra due livelli vicini.



Cache

La cache è il livello della memoria gerarchica che si trova tra il processore e la memoria principale. La memoria principale e il suo utilizzo sono generalmente trasparenti al programmatore, quindi nascosta. L'algoritmo di caching si basa sui **principi di località spaziale e temporale**; mantiene i dati richiesti recentemente vicino alla CPU (località temporale), e muove blocchi contigui di memoria che contendono il dato richiesto (località spaziale).

Tipi di cache

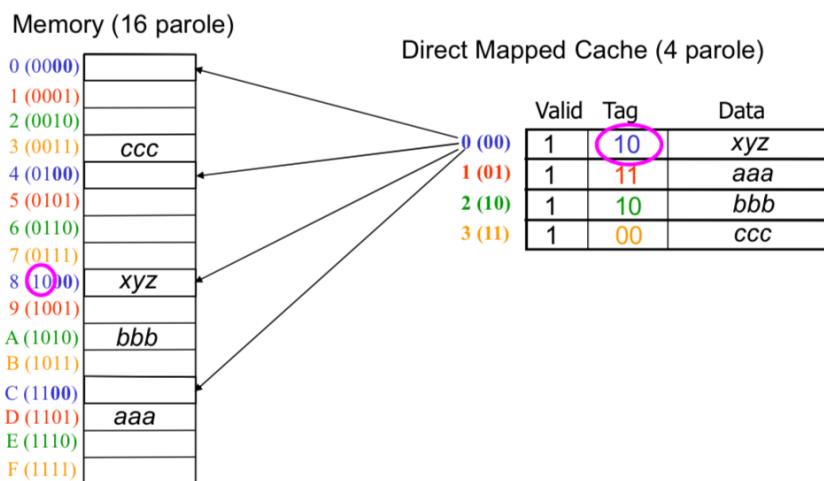
Ci sono di vari tipi:

- **Direct Mapped:** a ciascun blocco della memoria corrisponde una specifica locazione nella cache ha tre definizioni: **tag** (contiene informazioni necessarie a verificare se una parola della cache corrisponde o meno alla parola cercata), **indice** (utilizzato per selezionare il blocco della cache), **offset** (bit necessari per selezionare il byte richiesto nella parola).
 - o Dato che il numero di blocchi nella cache è una potenza di 2, la posizione corrispondente della parola in cache è data dai **log2(numero elementi nella cache)** bits meno significativi dell'indirizzo in memoria principale.
 - o Associa una sola locazione della cache a ogni parola della memoria, definendo una corrispondenza tra l'indirizzo in memoria della parola e la locazione nella cache
 - o L'indirizzo
- **Fully Associative:** ogni blocco può essere collocato in qualsiasi locazione della cache. Per ricercare un blocco nella cache, è necessario cercarlo in tutte le linee della cache. La ricerca sequenziale è troppo lenta, quindi si fa una *ricerca in parallelo*.
- **Set Associative:** soluzione intermedia tra direct mapped e fully associative. Ciascun blocco della memoria ha a disposizione un numero fisso di locazioni in cache.

Contenuto di una linea di cache

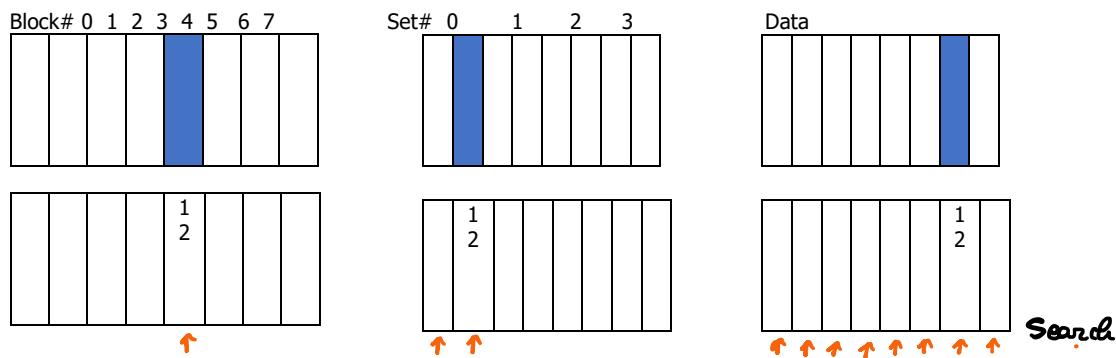
In una cache ad indirizzamento diretto, ogni linea di cache include:

- Il **bit di validità** indica se i dati nella linea di cache sono valido. All'avvio, tutte le linee sono non valide
- Il **tag**, che consente di individuare in modo univoco il blocco in memoria che è stato mappato nella linea di cache
- Il **blocco di dati** vero e proprio, formato da una o più parole



Da cache direct a mapped

Prendiamo come esempio il fatto che vogliamo trovare un **blocco di memoria con tag 12**; il metodo di ricerca cambia in base a come si vuole cercare il blocco corrispondente, e dal tipo di cache:



Per la memoria DMA (prima), c'è soltanto un singolo blocco cache dove il blocco 12 può essere trovato, e il numero del blocco è dato da **(12 modulo 8) = 4**.

Nella set associativa (seconda), ci saranno 4 set, e il blocco di memoria 12 deve essere impostato in **(12 mod 4) = 0**; non importa se a sinistra o a destra del set.

In una fully associative, il blocco di memoria per l'indirizzo del blocco 12 **può apparire in ognuno degli 8 blocchi della cache**.

Indirizzo

I bit meno significativi dell'indirizzo del dato richiesto indicano la posizione del dato nella cache. I bit più significativi dell'indirizzo del dato richiesto rappresentano il Tag.

Indirizzo 00101 → **Tag: 00** **indice: 101**
Tag indice

Prendiamo un esempio: dato un **indirizzo su 32 bit**, una cache DMA, la dimensione della cache è di 2^n blocchi (n sono usati per l'indice), la *dimensione del blocco della cache* è di 2^m parole (2^{m+2} byte). 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 $32 - (n+m+2)$, mentre il **numero totale di bit contenuti in una cache DMA** è: $2n * (\text{dimensione_blocco} + \text{dimensione_tag} + \text{bit_validità})$.

Tag

Il tag è un'etichetta o un identificatore associato a ogni blocco; questo viene usato per verificare se il dato richiesto dal processore è presente nella cache o no.

Quando il processore cerca un dato nella cache, l'indirizzo di memoria associato al dato viene diviso in **tag**, **indice** ed **offset**. Il processore confronta il tag dell'indirizzo richiesto con i tag dei blocchi di cache presenti nel set corrispondente. Se il tag corrisponde, si verifica un **hit**, altrimenti si verifica un **miss**, e il processore deve quindi accedere alla memoria principale per recuperare il dato richiesto.

Bit di validità

Il bit di validità indica se il **contenuto del blocco è valido o non valido**, e il suo scopo principale è identificare se i dati presenti nella cache sono stati caricati correttamente, e sono aggiornati rispetto ai dati presenti nella memoria principale.

Quando un blocco di cache viene caricato per la prima volta, o modificato a seguito di un'operazione di scrittura, il bit di validità viene impostato su *valida*. Se un blocco di cache è vuoto o non contiene dati validi, viene impostato invece su *non valido*.

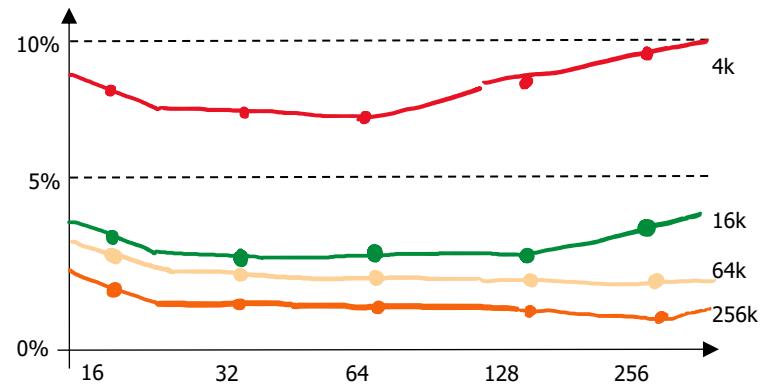
Mappatura di un indirizzo

Considerando ora una cache con 64 blocchi di 16 byte, un indirizzo n (in byte) del blocco può essere ricavato come: **(indirizzo del blocco) * modulo * (numero di blocchi nella cache)**

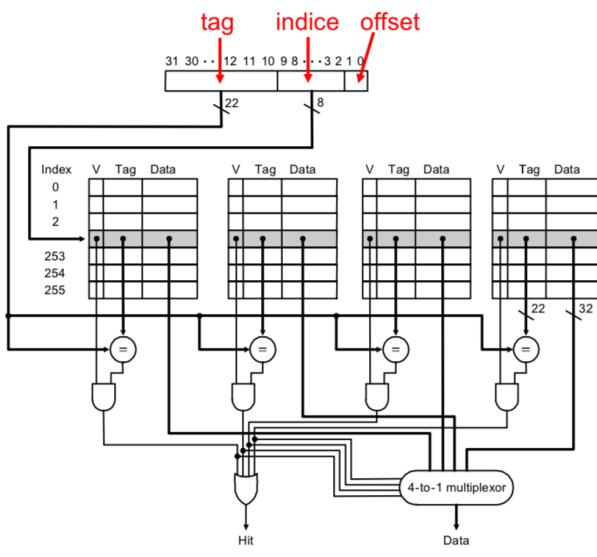
$$\text{Es. indirizzo del blocco} = \frac{\text{indirizzo del dato in byte}}{\text{Byte per blocco}} = \frac{1200}{16} = 75 \quad \left[\begin{array}{l} 64 \text{ blocchi di 16 byte} \\ \text{Indirizzo dato in byte: 1200} \end{array} \right]$$

Dimensione del blocco di cache

La dimensione di un blocco, insieme alla capacità totale della cache e alla politica di sostituzione dei blocchi, influisce sulla probabilità di verificarsi di un miss nella cache. *Un blocco più grande può contenere più dati e istruzioni*, aumentando la probabilità che un dato richiesto dal processore sia presente nella cache, **riducendo la frequenza dei miss**. Un blocco più grande però richiede anche più spazio di memoria.



Un'ampia dimensione per il blocco permette di sfruttare la località spaziale, ma blocchi di grossa dimensione comportano maggiori **miss penalty**, ed è necessario più tempo per trasferire il blocco. Se la dimensione del blocco è troppo grossa rispetto alla dimensione della cache, il *miss rate* aumenta, e il numero di blocchi nella cache è insufficiente.



all'indirizzo.

Lo svantaggio è che un ulteriore ritardo è fornito dal multiplexer, e il blocco è disponibile dopo la decisione Hit/miss (con accesso diretto, è disponibile prima) e la selezione del set.

Gestione di cache hit e cache miss

In caso di hit, il processore **continua il processamento**, accedendo al dato dalla cache dati, e all'istruzione dalla cache istruzioni.

In caso di miss invece, succede lo **stallo del processore** (come nel pipelining), in attesa di ricevere l'elemento dalla memoria. C'è successivamente un invio dell'indirizzo al controller della memoria, un reperimento dell'elemento dalla memoria, il caricamento dell'elemento in cache, e infine la ripresa dell'esecuzione.

Cache set-associative

I blocchi appartenenti alla cache sono raggruppati in **set**; in una cache di questo tipo ad N vie, ogni set raggruppa N blocchi. Ogni blocco di cache all'interno di un set ha un'**etichetta** (tag), che identifica il dato memorizzato e un dato effettivo (dati nella cache). Ogni indirizzo di memoria **corrisponde ad un unico set della cache**, e può essere ospitato in un blocco qualunque appartenente a quel set. Stabilito il set, per determinare se un certo indirizzo è presente in un blocco del set è necessario confrontare in parallelo i tag di tutti i blocchi.

Questo metodo offre una maggiore flessibilità rispetto alla cache diretta, in quanto può memorizzare più dati rispetto a una DMA, e riduce la complessità di ricerca rispetto a una cache associativa piena, in quanto il processore controlla solo il set specifico correlato

Gestione dei miss in lettura

Se un blocco non è presente nella cache, bisogna mettere in **stallo** l'intera CPU. Al verificarsi di un miss, bisogna:

1. Inviare PC (uscita della ALU) alla memoria: è l'uscita della ALU perché il ciclo prima, l'ALU calcola il PC+4 —> *bisogna inviare questo*
2. Nella memoria principale, viene effettuata un'**operazione di lettura** per recuperare l'istruzione mancante.
3. Dopo aver letto l'istruzione mancante, questa viene scritta nella cache delle istruzioni. La scrittura nella cache coinvolge l'inserimento del dato, del tag e di un bit di validità associato al blocco di cache.
4. Una volta che l'istruzione è stata caricata correttamente, viene riavviata l'esecuzione dell'istruzione che ha causato il miss.

Algoritmi per la sostituzione di blocchi

Nella cache ad accesso diretto, se il blocco di memoria è mappato in una linea di cache già occupata (**conflict miss**), si elimina il contenuto precedente della linea, e si rimpiazza con il nuovo blocco. Nel caso di cache completamente associativa, *ogni blocco è un potenziale candidato per la sostituzione*. In caso di cache set-associativa a N vie, bisogna scegliere tra gli N blocchi del set.

Possiamo quindi avere tre politiche di sostituzione:

- **Random**: scelta casuale
- **Least Recently Used** (LRU): sfruttando la località temporale, il blocco sostituito è quello che non si utilizza da più tempo. A ogni blocco si associa un contatore all'indietro, che viene portato al valore massimo in caso di accesso, e decrementato di 1 ogni volta che si accede ad un altro blocco.
- **First In First Out** (FIFO): seleziona il blocco più vecchio anziché quello non usato da più tempo.

Accesso in scrittura

Quando si scrive un dato nella cache di un sistema di memorizzazione, è necessario assicurarsi che tutti i **livelli inferiori della gerarchia di memoria** vengano aggiornati correttamente. Se non si effettuano gli aggiornamenti, si possono avere situazioni di incoerenza: se un dato viene scritto nella cache senza aggiornare il resto, crea un'incoerenza. Questo perché se si modifica un dato nella cache e non in memoria, e per qualche motivo si verifica un miss/**altre istruzioni hanno bisogno di quel dato in memoria**, non troveranno in memoria il nuovo dato ma quello vecchio, e quindi il programma non funziona.

Index	V	Tag	Data	Address	Data
...				...	
110	1	11010	42803	11010110	42803
...				...	

$$\text{Mem}[1101\ 0110] = 21763$$

Write-through

È un metodo di scrittura utilizzato nella cache, che comporta la **scrittura simultanea dei dati** sia nella cache che nella memoria principale. Quando un dato viene scritto nella cache, viene anche aggiornato nella memoria principale contemporaneamente.

Es. Supponiamo di avere una cache di dimensione 4 blocchi e una memoria principale. La cache è inizialmente vuota, e contiene solo dati invalidi. Si ha:

1. Si effettua una richiesta di lettura per un dato specifico
2. La cache controlla se il dato è presente nella cache
3. Se il dato è presente (hit), viene restituito dalla cache
4. Se il dato non è presente (miss), viene effettuata una richiesta di lettura alla memoria principale per ottenere il dato
5. Il dato viene caricato nella cache e restituito al richiedente

Write-through con write buffer

Con questo metodo, i dati vengono scritti sia nel buffer che nella cache contemporaneamente. Implementa un **write buffer**, che funziona come un'area temporanea per memorizzare le operazioni di scrittura in attesa di essere propagate alla memoria centrale. Quando il buffer è pieno, o raggiunge un certo limite, le operazioni di scrittura vengono inviate alla memoria centrale in un'unica volta in modo efficiente, *riducendo il numero di accessi alla memoria*, e migliorando le prestazioni complessive del sistema. Può essere che le operazioni di scrittura *vengono inviate alla memoria centrale uno alla volta*,

Write-back

È un protocollo di coerenza della cache utilizzato per gestire le modifiche dei dati nella cache, e la loro propagazione alla memoria principale in modo efficiente. Invece di aggiornare immediatamente la memoria principale ogni volta che viene modificato un dato nella cache, queste metodi **posticipa l'aggiornamento fino a quando il blocco di cache modificato non viene sostituito**.

Esempio:

4. Cache contiene un blocco di dati, chiamato "Blocco A" con un dato "1" all'indirizzo di memoria corrispondente.
5. Viene eseguita un'operazione che modifica il dato nel blocco A, sostituendolo con un nuovo valore "2".
6. Invece di aggiornare la memoria, il write-back tiene traccia del cambiamento.
7. Viene eseguita un'istruzione successiva che richiede il dato all'indirizzo corrispondente al blocco A. Trova quindi "2", perché è stato aggiornato.
8. Solo quando il blocco A viene rimosso dalla cache, o sostituito da un altro blocco, il write-back esegue la scrittura nella memoria principale.

Il write-back minimizza il numero di operazioni di scrittura verso la memoria principale, riducendo il traffico sulla memoria, e migliorando le prestazioni complessive del sistema. Questo può comportare però anche incoerenze.

Write-back con write buffer

È un metodo di scrittura che comporta la posticipa dell'aggiornamento della memoria principale, fino a quando **non è necessario**, o fino a quando il buffer di scrittura è pieno. Se il dato è presente, viene aggiornato solo nella cache; se invece non è presente, si effettua una richiesta di lettura alla memoria principale, e poi viene aggiornato nella cache e memorizzato nel buffer di scrittura.

Il buffer di scrittura tiene traccia delle modifiche apportate ai dati nella cache. Quando il buffer di scrittura è pieno, o viene richiesta una nuova lettura che coinvolge i dati nel buffer, **l'intero contenuto viene scritto nella memoria principale**. I dati modificati nella cache vengono aggiornati anche nella memoria principale.

Questo metodo aiuta a ridurre il numero di scritture effettuate direttamente sulla memoria principale, ottimizzando le prestazioni e riducendo il traffico di scrittura sulla memoria. *Riduce quindi la latenza*.

Write miss

Le scritture (sacre?) possono indurre un write miss. Abbiamo due soluzioni possibili:

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

Queste due soluzioni sono solitamente in combinazione con altri metodi:

- **Write back con Write allocate:** quando si verifica un write miss, il blocco di dati corrispondente viene caricato nella cache. La scrittura viene effettuata solo la cache, e questi verranno aggiornati nella principale solo quando il blocco viene eliminato dalla cache, o quando il buffer di scrittura è pieno.
- **Write through con Write Not Allocate:** quando si verifica un write miss, il dato viene scritto nella memoria senza essere trasferito nella cache. La cache funge solo da **cache di lettura**, e non memorizza i dati scritti. Ogni scrittura è fatta nella memoria principale, garantendo la coerenza dei dati tra la cache e la memoria principale. Questa è di solito considerata la soluzione più semplice.