



# Riassunto Calcolatori Elettronici

## 1. Il Modello di Von Neumann

Il modello di Von Neumann è lo schema progettuale di uno dei primi calcolatori realizzati, l'EDVAC, il quale evidenzia i principali nonché essenziali componenti di un generico calcolatore:

- Processore (**CPU**), composto a sua volta da un'unità di controllo e un'unità aritmetico-logica (ALU);
- Memoria centrale (**RAM**)
- Unità di **input/output**
- **Bus**: canali di comunicazione (solitamente si tratta di circuiti stampati) che collegano i vari componenti del calcolatore

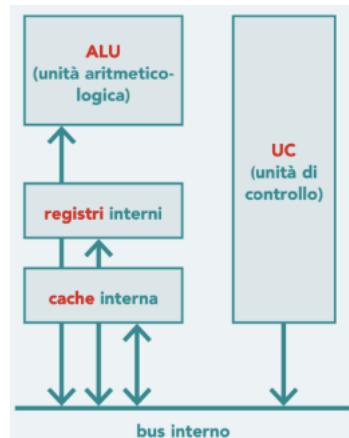
### 1.1 Il processore

Il **processore** o CPU (Central Processor Unit) ha il compito di eseguire i programmi immagazzinati in memoria centrale prelevando le istruzioni e i relativi dati, interpretandole ed eseguendole sequenzialmente.

La CPU internamente si compone di 4 componenti principali:

- **Unità di controllo** (UC)
- **Unità logico-aritmetica** (ALU)
- registri
- bus interni

È collegata alla memoria centrale mediante un bus.



#### 1.1.1 L'unità di controllo e il ciclo generale del processore

E' l'unità che si occupa di **dirigere e coordinare le attività interne** alla CPU che portano l'esecuzione di una istruzione. L'esecuzione di una determinata istruzione avviene attraverso alcune fasi:

- Fetch
- Decode
- Operand Assembly
- Execute

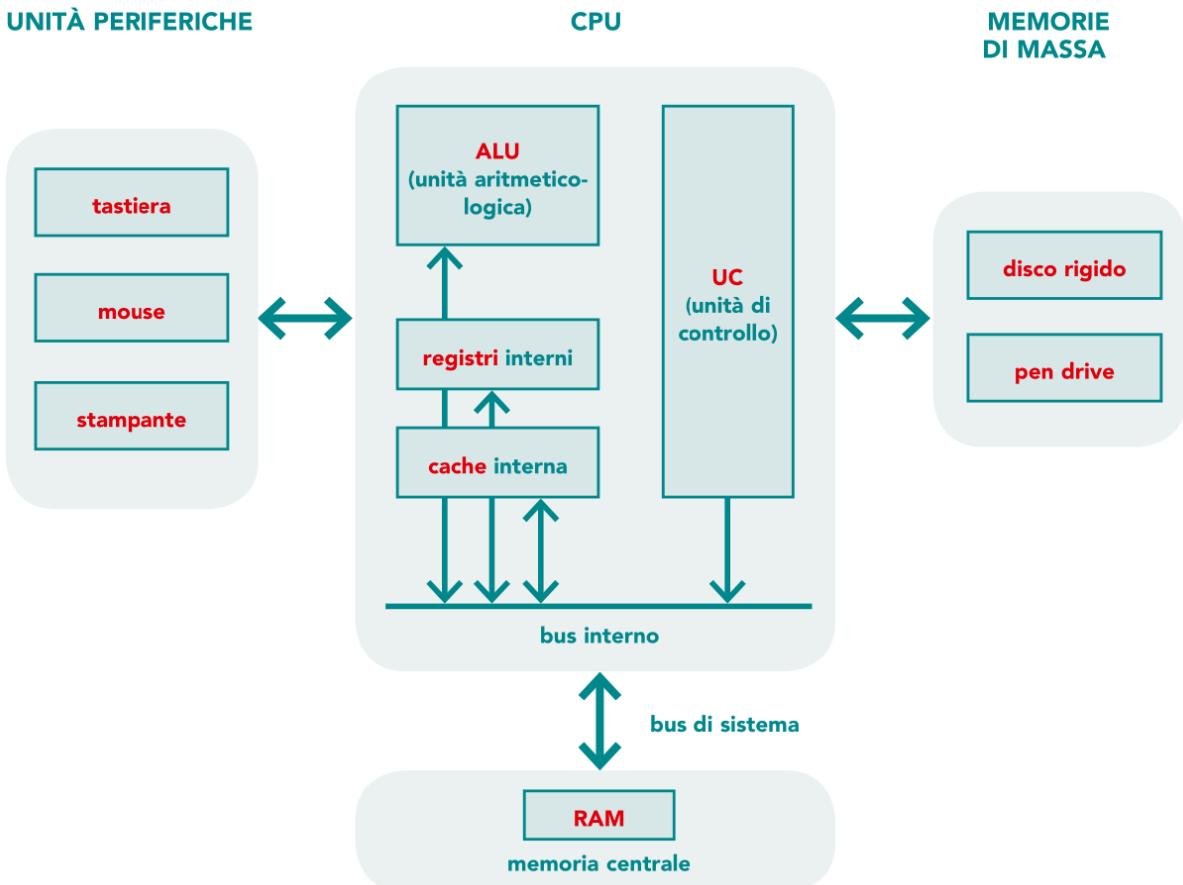
Queste fasi prendono il nome di ciclo generale del processore.

#### 1.1.2 L'unità logico-aritmetico

L'**ALU** si occupa di realizzare le operazioni logiche ed aritmetiche richieste per eseguire un'istruzione. Si avvale di due registri in ingresso e di un registro in uscita nel quale viene inserito il risultato dell'operazione.

È in grado di svolgere solamente operazioni **semplici**, quelle più complesse sono implementate mediante programmi.

## 1.2 La memoria

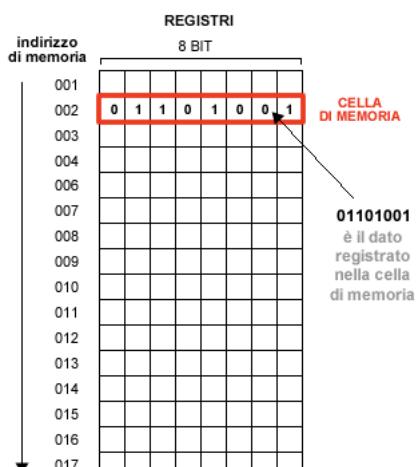


La memoria centrale ospita i programmi in esecuzione insieme ai loro dati.

I dati sono registrati nella memoria (qualunque essa sia) sotto forma di bit. Il **bit**, la più piccola unità di informazione memorizzabile, è memorizzato su un elemento fisico **bistabile**, un dispositivo elettronico che può assumere solamente due stati possibili in base alla tensione registrata. Tale componente rappresenta una singola **cella di memoria**, la quale ci consente di gestire  $2^1$  informazioni (binarie), due stati possibili.

La memoria si compone di un insieme  $n$  di celle, la quali consentono quindi di gestire  $2^n$  informazioni diverse. Un insieme di  $m$  celle di memoria prende il nome di **registro**... la memoria è quindi organizzata in registri di uguale dimensione, ognuno identificato mediante un indirizzo.

Se  $m$  è il numero di registri di memoria, il numero di bit che compongono l'indirizzo è dato da:  $n = \log_2 m$ .



L'indirizzo consente al processore di individuare precisamente ogni registro, leggere o scrivere un dato al suo interno.

La dimensione totale della memoria centrale è data dal numero dei registri presenti e si misura in **byte** e relativi multipli e sottomultipli (si dice che è "**byte addressed**").

Un processore a 32 bit (come il MIPS) permette di indirizzare  $2^{32}$  byte i quali equivalgono a circa 4 Gigabyte di RAM. La dimensione degli indirizzi dipende direttamente dalla dimensione della memoria centrale.

Possiamo distinguere diverse tipologie di memorie centrali:

- **RAM** (Random Access Memory): sono costituite da chip di **DRAM** (Dynamic Random Access Memory) o **SRAM** (Static Random Access Memory), si tratta di memorie ad accesso casuale (per accedere a dati posti in registri differenti si impiega lo stesso tempo) di tipo volatile (i dati vengono persi quando cessa l'alimentazione). Si utilizzano le DRAM in quanto il costo delle SRAM è nettamente superiore (vengono impiegate per la realizzazione di memorie cache);
- **ROM** ( Read Only Memory ): è una memoria di sola lettura. Non è una memoria volatile ma **permanente**. Il processore non può scrivere o modificare i dati al suo interno ma soltanto leggerli. In genere, la memoria ROM è usata all'accensione del computer per eseguire le prime operazioni di bootstrap.
- **Cache**: memoria volatile molto veloce e costosa, realizzata con tecnologia SRAM. Solitamente è presente nei processori in dimensioni molto ridotte.

## 2. Il linguaggio macchina e l'assembly

Il **linguaggio macchina** determina le istruzioni che la CPU è in grado di comprendere ed eseguire. L'insieme delle operazioni eseguibili dal processore prende il nome di **Instruction Set Architecture** (ISA).

Tale linguaggio è strettamente dipendente dall'hardware, per questo motivo esistono famiglie diverse di processori incompatibili fra loro dal momento che il loro ISA è diverso. Ne sono un esempio le famiglie 80x86, ARM, MIPS, RISC V.

All'inizio dell'era tecnologica, la tendenza era quella di realizzare architetture hardware con un set di istruzioni molto vasto e complesso in modo tale da poter eseguire direttamente un numero più ampio di operazioni. Tali processori prendono il nome di processori **CISC** (Complex Instruction Set Computing).

Negli anni '80 si afferma una nuova filosofia che mira a costruire processori più efficienti mantenendo un set di istruzioni ridotto e semplice senza implementare, tramite istruzioni dirette, le operazioni più complesse (le quali sono comunque fruibili tramite software). Nacquero così i processori **RISC** (Reduced Instruction Set Computing).

I processori CISC mettono a disposizione del programmatore istruzioni atomiche più complesse, simili a quelle riscontrabili nel linguaggio di alto livello. I processori RISC invece tendono ad avere operazioni semplici e veloci, con grande abbondanza di registri per memorizzare i risultati intermedi. Al giorno d'oggi questa distinzione non è più così marcata dal momento che la maggior parte dei processori consumer sono CRISP, cioè un mix fra i due.

Nel **linguaggio macchina** le istruzioni sono codificate come stringhe di bit che il processore può interpretare ed eseguire direttamente. Per semplificare la programmazione è stato inventato il linguaggio assembly il quale consente al programmatore di ignorare la struttura binaria delle istruzioni.

Il **linguaggio assembly** è un linguaggio a basso livello di tipo simbolico. È formato da una serie di mnemonici (o keyword) ognuno dei quali corrisponde a un'istruzione (opcode) codificata in linguaggio macchina. Sostanzialmente ogni codice operativo del linguaggio macchina viene sostituito, nell'assembly, da una sequenza di caratteri che lo rappresenta in forma mnemonica. per esempio, il codice operativo per la somma potrebbe essere trascritto come **ADD** quello per il salto come **JMP**.

Anche il linguaggio assembly, come il linguaggio macchina, è **legato all'hardware** per cui un programma scritto nel linguaggio assembly di un processore non sarà compatibile con un processore di un'architettura diversa. Ad esempio, il linguaggio assembly scritto per un processore Intel x86 non funzionerà su un processore ARM, perché l'architettura e le istruzioni di questi due processori sono diverse.

Generalmente la sintassi di un'istruzione assembly si compone di 4 elementi:

label	Mnemonico	Operandi	Commento
for:	add,	\$t1,\$t2,\$t3	#addizione \$t1=\$t2+\$t3

- **label**: sequenza di caratteri alfanumerici utilizzati per l'implementazione dei costrutti;
- **Mnemonico**: codice operativo o pseudocodice
- **operandi**: operandi, separati da virgole, sui quali applicare l'operazione specificata dal mnemonico;
- **commento**: viene anticipato da # ed è ignorato dall'assemblatore.

### 3. Architettura MIPS

**MIPS** è stata una delle prime aziende a costruire una delle prime architetture RISC commerciali.

Il processore MIPS (*Microprocessor without Interlocking Pipe Stages*) si basa su:

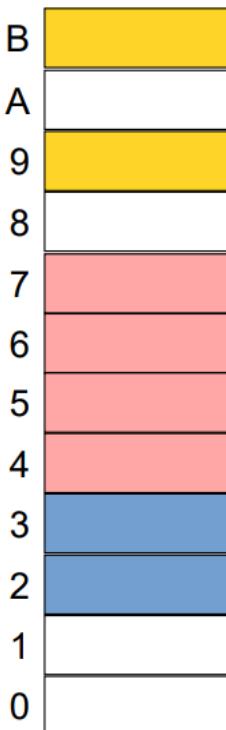
- Architettura di tipo Load/Store con istruzioni registro-registro a 3 operandi;
- Istruzioni da 32 bit fruibili in 3 formati: R (register), I (immediate), J (jump);
- 32 registri generali da 32 bit
- Immediati a 16 bit rappresentati in complementi a 2

#### 3.1 I registri del MIPS

Registri			
Nome	Numero	Utilizzo	Chiamante deve preservare?
\$zero	\$0	costante 0	N/A
\$at	\$1	Assembly Temporary	No
\$v0-\$v1	\$2-\$3	Values for function returns and expression evaluation	No
\$a0-\$a3	\$4-\$7	Argomenti delle funzioni	No
\$t0-\$t7	\$8-\$15	Registri Temporanei	No
\$s0-\$s7	\$16-\$23	Registri Salvati	Si
\$t8-\$t9	\$24-\$25	Registri Temporanei	No
\$k0-\$k1	\$26-\$27	Riservati al kernel del S.O.	No
\$gp	\$28	Global Pointer	Si
\$sp	\$29	Stack Pointer	Si
\$fp	\$30	Frame Pointer	Si
\$ra	\$31	Return Address	N/A

- \$zero : contiene il valore 0, è utilizzato per semplificare e velocizzare i confronti;
- \$at (o *Assembler Temporary*) è riservato dall'assemblatore per permettere l'utilizzo delle pseudoistruzioni. Non è indirizzabile.
- \$v0 – \$v1 (*Registri Valore*) sono utilizzati per restituire valori dalle funzioni;
- \$a0 – \$a3 (*Registri Argomento*) sono utilizzati per passare argomenti alle funzioni;
- \$t0 – \$t9 (*registri temporanei*) sono utilizzati nelle funzioni. Il contenuto non viene preservato al termine della funzione in quanto utile solo all'esecuzione della stessa.
- \$s0 – \$s7 (*Registri Salvati*) sono utilizzati nel main program ed è quindi necessario preservarne il contenuto;
- \$k0 – \$k1 sono registri riservati al kernel del sistema operativo.
- \$sp (Stack Pointer) contiene l'indirizzo della cima dello stack ed è usato per operazioni di *Push* e *Pop* rispettivamente per inserire o estrarre elementi dallo Stack.
- \$ra (Return Address) contiene l'indirizzo di rientro da chiamata a sottoprogramma. È impiegato nell'istruzione **jal** (*Jump And Link*) per memorizzare l'indirizzo dell'istruzione successiva (PC + 4) e permettere di riprendere la regolare esecuzione del programma chiamante tramite **jr \$ra** (*Jump to Register*).
- \$hi e \$lo sono utilizzati per accedere al risultato delle operazioni di moltiplicazione e divisione
- \$pc (Program Counter) contiene l'indirizzo della prossima istruzione da eseguire.

#### 3.3 La gestione della memoria



Il processore supporta **dati numerici** in complemento a 2 di diversa dimensione:

- **Byte** (8 bit)
- **Halfword** (16 bit)
- **Word** (32 bit)

Si tratta di un processore a 32 bit con **indirizzamento al byte** (ogni indirizzo identifica un byte) per cui in memoria, un **byte** occuperà 1 registro, una **halfword** occuperà 2 registri consecutivi, una **word** 4 registri da 1 byte.

I dati devono essere **allineati** in memoria. Considerando un indirizzo  $A$  di  $s$  byte, deve valere:

$$A \% s = 0$$

I dati devono essere quindi allocati ad **indirizzi multipli** di  $s$ .

Per un **byte** quindi  $s = 1$ , per una **halfword**  $s = 2$  mentre per una **word**  $s = 4$ . Quindi, una word deve essere allocata ad indirizzi multipli di 4, una halfword ad indirizzi multipli di 2 mentre un byte ad indirizzi multipli di 1.

Il MIPS è un processore che sfrutta l'ordinamento (endianness) in memoria **big-endian**, questo significa che il bit più significativo (MSB) viene allocato nel registro che presenta l'indirizzo minore (quello più in alto). Il formato little endian prevede invece l'allocazione del MSB all'indirizzo maggiore della word che si sta considerando. Il big endian è stato scelto come standard.

#### ESEMPIO

Data la word:  $0x1A2B3C4D$  può essere memorizzata in 2 maniere differenti:

##### BIG ENDIAN:

Data	Address
1A	000000
2B	000001
3C	000010
4D	000011

##### LITTLE ENDIAN

Data	Address
4D	000000
3C	000001
2B	000010
1A	000011

## 3.4 Il formato delle istruzioni

L'architettura MIPS prevede Istruzioni da 32 bit fruibili in **3 formati**: R (register), I (immediate), J (jump).

#### FORMATO R

codop	rs	rt	rd	shamt	funz
6 bit	5 bit	5 bit	5 bit	5 bit	6bit

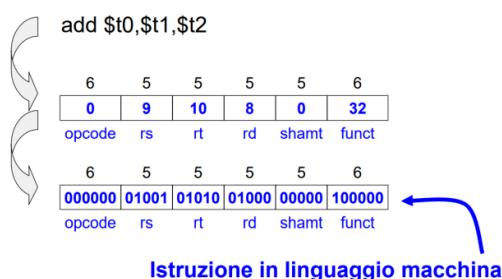
- **codop**: specifica parzialmente il codice operativo dell'operazione da effettuare, è chiamato solitamente **codice operativo** o opcode;
- **rs (Source Register)**: registro contenente il primo operando;
- **rt (Target Register)**: registro contenente il secondo operando;

- **rd (Destination Register)**: registro di destinazione, riceve il risultato dell'operazione;
- **shamt** (shift amount): numero di posizioni di scorrimento. Viene utilizzato con le istruzioni di shift;
- **funz**: specifica la variante dell'operazione base definita dal codice operativo. Questo campo è detto "codice funzione".

#### ESEMPIO - istruzione add

Consideriamo l'istruzione `add $t0, $t1, $t2`. L'istruzione fa utilizzo del formato R e i vari campi sono definiti nel seguente modo:

codop	rs	rt	rd	shamt	funz
0	9 (\$t1)	10 (\$t2)	8 (\$t0)	0	32



Un problema nasce quando un'istruzione richiede campi di **dimensioni maggiori** rispetto a quelle specificate sopra. Per esempio, l'istruzione load word (`LW`), richiede di specificare due registri e una costante. Se si utilizzasse il formato R per un'istruzione di spostamento dati come "load word", la costante sarebbe allocata in un registro di 5 bit per cui non potrebbe superare il valore di  $2^5 = 32$ .

Un campo da 5 bit risulterebbe troppo **piccolo**... per questo motivo esistono diversi formati delle istruzioni, ognuno dei quali si adatta meglio ad una determinata circostanza.

L'istruzione `LW` per esempio, utilizza il formato I (dal momento che, per la sintassi dell'istruzione, sono richiesti 2 registri e un immediato).

#### FORMATO I

codop	rs	rt	costante o indirizzo
6 bit	5 bit	5 bit	16 bit

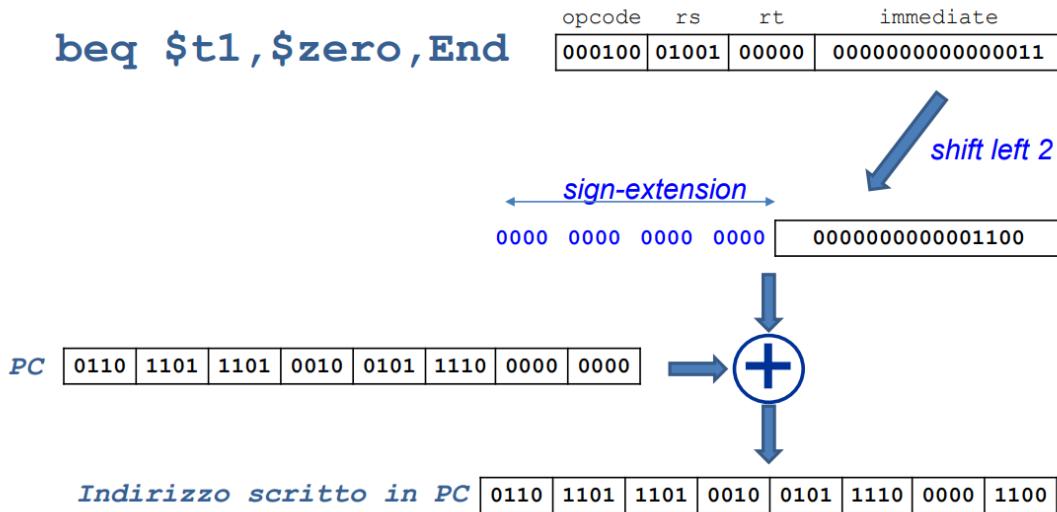
- **codop**: specifica parzialmente il codice operativo dell'operazione da effettuare, è chiamato solitamente **codice operativo** o opcode;
- **rs (Source Register)**: registro contenente l'unico operando;
- **rt (Target Register)**: registro che conterrà il risultato dell'operazione
- **immediate**: può essere un immediato reale (con SE) oppure un offset (con SE) per specificare un indirizzo

Fanno uso del formato I tutte le istruzioni che operano con immediati come `lw`, `sw`... anche le istruzioni di **branch** usano il formato I: `beq`, `bne`...

In questo modo l'istruzione `LW` (così come le altre) può trasferire dalla memoria centrale, qualsiasi parola in un intervallo di  $\pm 2^{15} = 32768$ . Per "bypassare" questo limite si può sfruttare l'indirizzamento PC-relative andando quindi a specificare uno spiazzamento rispetto all'indirizzo corrente, e non l'indirizzo assoluto dell'istruzione al quale si vuole saltare.

Nel caso di un'istruzione di branch quindi, il campo immediate conterrà lo spiazzamento rispetto all'indirizzo attualmente contenuto nel program counter.

## Formazione dell'indirizzo



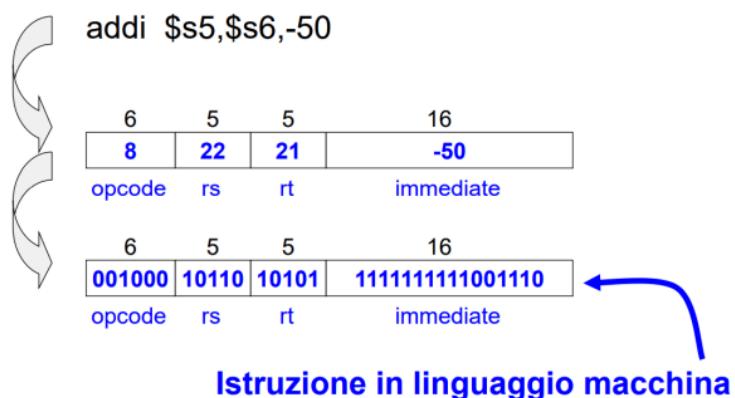
Gli indirizzi dei salti sono allineati alla word questo significa che, all'esecuzione di un'istruzione di branch:  $PC = (PC + 4) + (\text{immediate} \cdot 4)$ . L'immediato viene quindi moltiplicato per 4:

I due formati possono essere distinti grazie ai valori presenti nel primo campo: a ciascun formato è assegnato un insieme di valori del campo "op", in modo tale che l'hardware sappia esattamente se considerare l'istruzione nel formato I oppure R.

### ESEMPIO - istruzione addi

Consideriamo l'istruzione `addi $t0, $t1, -50`

codop	rs	rt	costante o indirizzo
8	9 (\$t1)	10 (\$t2)	-50

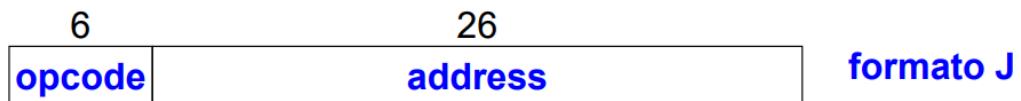
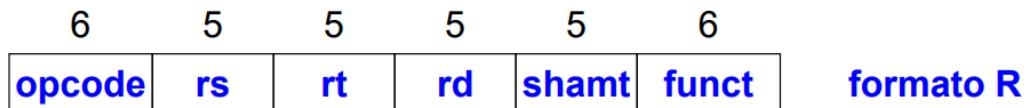


### **FORMATO J**

codop	address
6 bit	26 bit

Viene utilizzato per i salti, j, jal... In questo caso non si hanno limiti sulla destinazione raggiungibile dal salto. Il problema riguarda la dimensione degli indirizzi (32 bit) dal momento che devono entrare in un campo da 26 bit.

Dal momento che, anche in questo caso l'allineamento è alla word (ultimi 2 bit pari a 0)



## **3.5 Le modalità di indirizzamento**

Con **modalità di indirizzamento** si intende la modalità con cui il processore consente alle istruzioni di accedere ai relativi operandi durante il ciclo di fetch-execute. Le modalità di indirizzamento specificano come ottenere l'operando (o gli operandi) su cui una determinata istruzione deve agire.

- immediato `sit $t0, $s1, 100`
- registro `siti $t0, $s1, 100`
- base+offset `lw $t0, 4($t1)`
- pc-relative `beq $t0, $s1, label`
- pseudodiretto `j label`

## **4. Le istruzioni dell'assembly MIPS**

### **4.1 Le istruzioni aritmetiche**

Instruction	Example	Meaning	Comments
<b>add</b>	add \$1,\$2,\$3	\$1=\$2+\$3	
<b>subtract</b>	sub \$1,\$2,\$3	\$1=\$2-\$3	
<b>add immediate</b>	addi \$1,\$2,100	\$1=\$2+100	"Immediate" means a constant number
<b>add unsigned</b>	addu \$1,\$2,\$3	\$1=\$2+\$3	Values are treated as unsigned integers, not two's complement integers
<b>subtract unsigned</b>	subu \$1,\$2,\$3	\$1=\$2-\$3	Values are treated as unsigned integers, not two's complement integers
<b>add immediate unsigned</b>	addiu \$1,\$2,100	\$1=\$2+100	Values are treated as unsigned integers, not two's complement integers
<b>Multiply (without overflow)</b>	mul \$1,\$2,\$3	\$1=\$2*\$3	Result is only 32 bits!
<b>Multiply</b>	mult \$2,\$3	\$hi,\$low=\$2*\$3	Upper 32 bits stored in special register <b>hi</b> Lower 32 bits stored in special register <b>lo</b>
<b>Divide</b>	div \$2,\$3	\$hi,\$low=\$2/\$3	Remainder stored in special register <b>hi</b> Quotient stored in special register <b>lo</b>

Le operazioni aritmetiche si svolgono esclusivamente con gli operandi presenti nei registri del processore... non è possibile quindi effettuare operazioni con operandi allocati nella memoria centrale.

Tutti gli **immediati** in MIPS hanno dimensione pari a 16 bit. Se la rappresentazione di un determinato numero (immediato) richiede meno bit ( $< 16$ ), si effettua il **sign extension**, replicando il bit di segno sui bit superiori: la rappresentazione di un numero in complementi alla base, su  $N$  bit, può essere facilmente estesa su un registro di ampiezza maggiore, replicando il bit di segno sui bit superiori.

Le seguenti sono **pseudoistruzioni** ovvero istruzioni riconducibili a serie successive di espressioni base.

## Pseudoistruzioni aritmetiche

Pseudoistr.	Significato	Esempio
abs rd,rs	$rd = \text{ABS}(rs)$	abs \$t1,\$t2
div rd,rs,src	$rd = rs \div src$	div \$t1,\$t2,100
divu rd,rs,src	$rd = rs \div src$	divu \$t1,\$t2,\$t3
mul rd,rs,src	$rd = rs \times src$	mul \$t1,\$t2,100
mulo rd,rs,src	$rd = rs \times src$	mulo \$t1,\$t2,\$t3
mulou rd,rs,src	$rd = rs \times src$	mulou t\$1,\$t2,\$t3
		unsigned
		eccezione possibile
rem rd,rs,src	$rd = rs \bmod src$	rem \$t1,\$t2,100
remu rd,rs,src	$rd = rs \bmod src$	remu \$t1,\$t2,100
		signed
		unsigned

ESEMPIO: ampiezza del registro da 8 a 16 bit

Prendiamo come esempio il numero decimale 245, rappresentabile su 8 bit come 11110101.

$$245_{(10)} = 11110101_{(2)}$$

Il numero occupa solamente 8 bit ma il MIPS prevede che tutti gli immediati siano a 16 bit. Si effettua quindi il sign extension:

$$245_{(10)} = 11110101_{(2)} = 1111111111110101_{(2)}$$



Con le operazioni aritmetiche in complementi a 2 è possibile che si presenti un overflow. L'overflow si verifica quando i bit a disposizione non sono sufficienti a rappresentare il risultato di un'operazione. Con i numeri unsigned l'overflow viene ignorato, con i numeri signed (complementi a 2) l'overflow viene generato mediante **un'eccezione**, detta anche **interrupt**. Sostanzialmente si tratta di una chiamata non preventivata ad una procedura, l'indirizzo dell'istruzione che ha causato l'overflow viene salvato nel registro EPC (exception program counter) e l'esecuzione salta a un indirizzo predefinito a partire dal quale è contenuta la procedura giusta per gestire l'eccezione.

## 4.2 Le istruzioni logiche

Le istruzioni logiche consentono sostanzialmente di operare e manipolare gruppi di bit o singoli bit all'interno delle word. Così come le operazioni aritmetiche, anche le operazioni logiche si effettua solamente tra registri del processore.

- `and`
- `or`
- `xor`
- `nor`
- `andi`
- `ori`
- `xori`

**AND**

$A$	$B$	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

**OR**

$A$	$B$	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

**XOR**

$A$	$B$	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

**NOR**

$A$	$B$	$A \downarrow B$
0	0	1
0	1	0
1	0	0
1	1	0

### ESEMPIO

Consideriamo 2 registri `$s1` e `$s2` contenenti i seguenti valori:

## Source Registers

\$s1	1111	1111	1111	1111	0000	0000	0000	0000
\$s2	0100	0110	1010	0001	1111	0000	1011	0111

### Assembly Code

and \$s3, \$s1, \$s2  
 or \$s4, \$s1, \$s2  
 xor \$s5, \$s1, \$s2  
 nor \$s6, \$s1, \$s2

### Result

\$s3	0100	0110	1010	0001	0000	0000	0000	0000
\$s4	1111	1111	1111	1111	1111	0000	1011	0111
\$s5	1011	1001	0101	1110	1111	0000	1011	0111
\$s6	0000	0000	0000	0000	0000	1111	0100	1000

## 4.3 Le istruzioni di scorrimento

- `sll`       $\$t1 = \$t2 \ll 10$       shift logico a sinistra con immediato
- `sllv`      $\$t1 = \$t2 \ll \$t3$       shift logico a sinistra con registro
- `srl`       $\$t1 = \$t2 \gg 10$       shift logico a destra con immediato
- `srlv`      $\$t1 = \$t2 \gg \$t3$       shift logico a sinistra con registro
- `sra`       $\$t1 = \$t2 \ll 10$       shift aritmetico a destra
- `sraw`

Le istruzioni di scorrimento possono essere di tipo logico o aritmetico e consentono sostanzialmente di spostare verso destra o sinistra  $n$  bit. La differenza tra lo shift logico e quello aritmetico sta nella **gestione del bit di segno**. Da notare che non esiste un apposito mnemonico per lo shift aritmetico a sinistra in quanto è equivalente a quello logico... in questo caso non occorre valutare il bit di segno, si procede direttamente riempiendo con degli 0.

Lo **shift a destra** invece è differente in quanto, procedendo con lo zero extension (riempiendo quindi i bit a sinistra con degli 0) si potrebbe cambiare il segno del numero rappresentato (in quanto rappresentati in complementi a due).

In tal caso lo shift aritmetico tiene in considerazione il bit di segno e procede con **sign-extension** in base ad esso.

Lo shift logico invece, non tenendo conto del bit di segno, effettua unicamente **zero-extension**.

Source Values	
\$s1	1111 0011 0000 0000 0000 0010 1010 1000
shamt (shift amount)	00100
Assembly Code	
sll \$t0, \$s1, 4	\$t0 0011 0000 0000 0000 0010 1010 1000 0000
srl \$s2, \$s1, 4	\$s2 0000 1111 0011 0000 0000 0000 0010 1010
sra \$s3, \$s1, 4	\$s3 1111 1111 0011 0000 0000 0000 0010 1010
Result	

Le istruzioni di shift possono essere utilizzate per realizzare operazioni di moltiplicazione o divisione per  $2^n$ :

Uno scorrimento a destra di  $n$  bit, può essere visto come una divisione per  $2^n$ , uno scorrimento a sinistra può essere visto come una moltiplicazione per  $2^n$ .

## Istruzioni di scorrimento aritmetiche

Uno scorrimento a destra di n bit, può essere visto come una divisione per  $2^n$ :

Es.:

$01010110_2 \rightarrow 86_{10}$  con uno scorrimento a destra di 2 bit si ottiene

$00010101_2 \rightarrow 21_{10}$  vengono inseriti due 0 a sinistra  
↑

**Che cosa succede se si considerano numeri signed ?**

Es.:

$10010110_2 \rightarrow -106_{10}$  con uno scorrimento a destra di 2 bit si ottiene

$00100101_2 \rightarrow +37_{10}$  **errato !**

$11100101_2 \rightarrow -27_{10}$  vengono inseriti due 1 a sinistra **corretto**

↑ **sign extension**

### 4.4 Le istruzioni di movimento dati

Il MIPS prevede alcune particolari istruzioni che consentono di spostare dati tra i registri del processore e la memoria centrale. Questo avviene in quanto i registri del processore non sono sufficienti.

Si possono spostare dati di dimensione pari a 1, 2 o 4 byte, tuttavia i trasferimenti sono sempre da 32 bit. Per i registri speciali HI e LO si utilizzano istruzioni particolari.

Il MIPS prevede due gruppi di istruzioni di movimento dati:

*load      memoria → registro*  
*store     registro → memoria*

La sintassi generale delle istruzioni di movimento è la seguente:

istruzione <registro su cui caricare o da caricare> <offset>(<registro contenente l'indirizzo>)

#### 4.4.1 Istruzioni load

<b>Istruzione</b>	<b>Significato</b>
<b>lb \$t1, address</b>	Mem[address](8 bit) -> \$t1      esteso con segno
<b>lbu \$t1, address</b>	Mem[address](8 bit) -> \$t1      esteso con 0
<b>lh \$t1, address</b>	Mem[address](16 bit) -> \$t1      esteso con segno
<b>lhu \$t1, address</b>	Mem[address](16 bit) -> \$t1      esteso con 0
<b>lw \$t1, address</b>	Mem[address](32 bit) -> \$t1
<b>lui \$t1, imm</b>	$imm \times 2^{16} \rightarrow \$t1$ Carica una costante da 16 bit nella parte alta del registro, azzerando la parte bassa

Per caricare dati nei registri del processore occorre conoscere (deve essere salvato in un registro), **l'indirizzo** del dato da caricare, situato attualmente in memoria centrale.

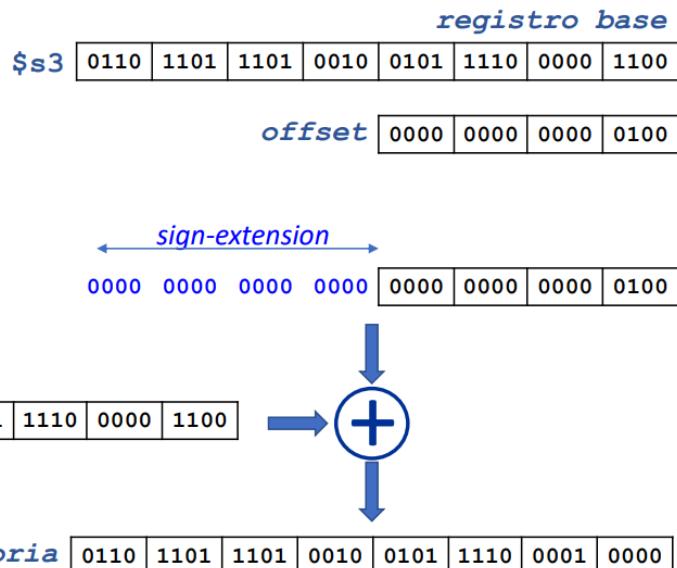
L'indirizzo si costruisce in fase di esecuzione dell'istruzione sommando al registro base lo spiazzamento.

## Formazione dell'indirizzo

**lw \$t0, 4(\$s3)**

L'indirizzo al quale accedere in memoria è costruito in fase di esecuzione dell'istruzione, sommando il contenuto del registro base allo spiazzamento sign-extended

Quanti bit sono utilizzati?



### ESEMPIO

**lw \$t0, 4(\$s3)**

In questo esempio si cerca di caricare in \$t0 il contenuto presente all'indirizzo contenuto in \$s3 incrementato di 4. Prima di caricare il valore in \$t0 viene quindi calcolato l'indirizzo corretto sommando all'indirizzo di base (contenuto in \$s3), lo spiazzamento di 4.

## Esempi di istruzioni di load

Il registro \$s3 "punta" a questo indirizzo in memoria

	Indirizzo								registro
\$s3	0110	1101	1101	0010	0101	1110	0000	1100	
	0110	1101	1101	0010	0101	1110	0000	1101	
	0110	1101	1101	0010	0101	1110	0000	1110	1100 0101
	0110	1101	1101	0010	0101	1110	0000	1111	0110 1001
	0110	1101	1101	0010	0101	1110	0001	0000	1010 0101
	0110	1101	1101	0010	0101	1110	0001	0001	1101 0011
	0110	1101	1101	0010	0101	1110	0001	0010	0010 1100
	0110	1101	1101	0010	0101	1110	0001	0011	1111 0100

lb \$t0,4(\$s3)	\$t0	1111	1111	1111	1111	1111	1111	1010	0101
lbu \$t1,4(\$s3)	\$t1	0000	0000	0000	0000	0000	0000	1010	0101
lh \$t2,4(\$s3)	\$t2	1111	1111	1111	1111	1010	0101	1101	0011
lhu \$t3,4(\$s3)	\$t3	0000	0000	0000	0000	1010	0101	1101	0011
lw \$t4,4(\$s3)	\$t0	1010	0101	1101	0011	0010	1100	1111	0100

Una volta individuato l'indirizzo corretto si procede con il caricamento in \$t0. Il registro \$s3 è effettivamente un puntatore a un registro presente in memoria centrale in quanto contiene il suo indirizzo.

Le istruzioni di **store** consentono invece di trasportare dati dai registri del processore alla memoria centrale.

Istruzione	Significato
sb \$t1,address	\$t1(0:7) → Mem[address]
sh \$t1,address	\$t1(0:15) → Mem[address]
sw \$t1,address	\$t1(0:31) → Mem[address]

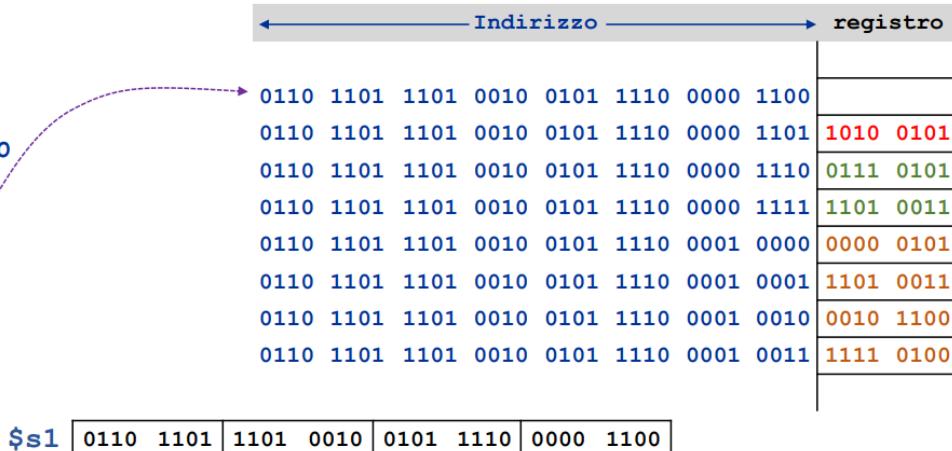
### Esempio - store

sw \$t0,4(\$s3)

Il valore contenuto in \$t0 viene caricato nel registro della memoria centrale il cui indirizzo di base è presente in \$s3. Analogamente all'istruzione "load", inizialmente si calcola l'indirizzo corretto effettuando la somma dell'indirizzo di base (contenuto in \$s3) e lo spiazzamento (in questo caso di 4).

## Esempi di istruzioni di store

Il registro \$s1 "punta" a questo indirizzo in memoria



<b>sb</b> <b>\$t1,1(\$s1)</b>	<b>\$t1</b> 0101 1101   0101 0000   1010 0011   <b>1010 0101</b>
<b>sh</b> <b>\$t2,2(\$s1)</b>	<b>\$t2</b> 1101 0001   0010 1011   <b>0111 0101</b>   1101 0011
<b>sw</b> <b>\$t3,4(\$s1)</b>	<b>\$t3</b> <b>0000 0101</b>   1101 0011   0010 1100   <b>1111 0100</b>

Una particolare istruzione è **lui** (load upper immediate), la quale consente di caricare una costante (in MIPS le costanti sono da 16bit) nella parte alta di un registro, azzerando la parte inferiore.

Tale istruzione (accoppiata solitamente con l'istruzione **ori**) torna utile per caricare in un registro una costante da 32 bit:

Caricamento di una costante da 16 bit

<b>addiu \$t5,\$zero,0x3D1C</b>	<b>\$t5</b> 0000 0000   0000 0000   0011 1101   0001 1100
<b>addi \$t6,\$zero,-1</b>	<b>\$t6</b> 1111 1111   1111 1111   1111 1111   1111 1111

**sign extension**

Caricamento in \$t0 di una costante da 32 bit (0xA27B51C5)

<b>lui \$t0,0xA27B</b>	<b>\$t0</b> 1010 0010   0111 1011   0000 0000   0000 0000
<b>ori \$t0,\$t0,0x51C5</b>	<b>\$t0</b> 1010 0010   0111 1011   0101 0001   1100 0101

Si utilizza l'istruzione **lui** per caricare nella parte alta del registro i primi 16-bit, a partire dal MSB, dell'immediato *A27B* espresso in base esadecimale (per questo il prefisso *0x*).

$$A27B_{(16)} = 1010001001111011_{(2)}$$

L'istruzione **lui** azzerava automaticamente i restanti 16 bit presenti nella parte bassa del registro.

Nella parte bassa del registro si vuole caricare l'immediato

$$51C5_{(16)} = 101000111000101_{(2)}$$

Per fare ciò si utilizza l'istruzione logica `ori` "confrontando" il contenuto di \$t0 con 51C5:

\$t0 (prima)	1010 0010	0111 1011	0000 0000	0000 0000
0x51C5	0000 0000	0000 0000	0101 0001	1100 0101
\$t0 (dopo)	1010 0010	0111 1011	0101 0001	1100 0101

$A$	$B$	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

Effettuando l'or tra A27B (già contenuto in \$t0) e 51C5 si può riempire la parte bassa del registro in quanto la somma logica (or) restituisce 0 solamente quando entrambi i bit sono 0.

## Pseudoistruzioni di movimento dati

<i>Pseudoistr.</i>	<i>Significato</i>	<i>Esempio</i>
<code>move rd,rs</code>	$rd = rs$	<code>move \$s1,\$t2</code>
<code>li rd,imm</code>	$rd = imm$	<code>li \$t0,0x2A34</code>
<code>ld rd,address</code>	$rd = \text{Mem}[address](32 \text{ bit})$ $rd+1 = \text{Mem}[address+4](32 \text{ bit})$	
<code>sd rd,address</code>	$\text{Mem}[address] = rd \quad (32 \text{ bit})$ $\text{Mem}[address+4] = rd+1 \quad (32 \text{ bit})$	

## 4.5 Le istruzioni di controllo

### 4.5.1 Istruzioni di confronto

<code>slt \$t1,\$t2,\$t3</code>	<i>if (\$t2&lt;\$t3) \$t1=1; else \$t1=0</i>	<i>Confronto tra registri (con segno)</i>
<code>slti \$t1,\$t2,100</code>	<i>if (\$t2&lt;100)\$t1=1; else \$t1=0</i>	<i>Cfr. registro-costante (con segno)</i>
<code>sltu \$t1,\$t2,\$t3</code>	<i>if (\$t2&lt;\$t3) t\$t1=1; else \$t1=0</i>	<i>Cfr. tra registri (senza segno)</i>
<code>sltiu \$t1,\$t2,100</code>	<i>if (\$t2&lt;100) \$t1=1; else \$t1=0</i>	<i>Cfr. registro-costante (senza segno)</i>

Effettuano il confronto tra due operandi. Se la condizione è verificata viene posto nel primo registro 1, se la condizione non è verificata viene posto nel registro 0.

#### ESEMPIO - istruzioni di confronto

<code>\$t0</code>	<table border="1"><tr><td>1010</td><td>0101</td><td>1101</td><td>0011</td><td>0010</td><td>1100</td><td>1111</td><td>0100</td></tr></table>	1010	0101	1101	0011	0010	1100	1111	0100
1010	0101	1101	0011	0010	1100	1111	0100		
<code>\$t1</code>	<table border="1"><tr><td>0110</td><td>1001</td><td>0101</td><td>1100</td><td>0010</td><td>1100</td><td>1111</td><td>0100</td></tr></table>	0110	1001	0101	1100	0010	1100	1111	0100
0110	1001	0101	1100	0010	1100	1111	0100		
	<code>0xED17</code> <table border="1"><tr><td>1110</td><td>1101</td><td>0001</td><td>0111</td></tr></table>	1110	1101	0001	0111				
1110	1101	0001	0111						

<code>slt \$t2,\$t0,\$t1</code>	<code>\$t2</code> <table border="1"><tr><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0001</td></tr></table>	0000	0000	0000	0000	0000	0000	0000	0001
0000	0000	0000	0000	0000	0000	0000	0001		
<code>sltu \$t3,\$t0,\$t1</code>	<code>\$t3</code> <table border="1"><tr><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td></tr></table>	0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000		
<code>slti \$t4,\$t0,0xED17</code>	<code>\$t4</code> <table border="1"><tr><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0001</td></tr></table>	0000	0000	0000	0000	0000	0000	0000	0001
0000	0000	0000	0000	0000	0000	0000	0001		
<code>sltiu \$t5,\$t0,0xED17</code>	<code>\$t5</code> <table border="1"><tr><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td><td>0000</td></tr></table>	0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	0000	0000	0000	0000	0000	0000		

Nel primo esempio `.slt $t2,$t0,$t1` si effettua il confronto  $$t0 < $t1$  fra i contenuti dei due registri e si pone in  $$t2$  0 in quanto il contenuto di  $$t0$  è maggiore di quello presente in  $$t1$ .

## Pseudoistruzioni

**seq** rdest,rsource1,source2  
**sge** rdest,rsource1,source2  
**sgeu** rdest,rsource1,source2  
**sgt** rdest,rsource1,source2  
**sgtu** rdest,rsource1,source2  
**sle** rdest,rsource1,source2  
**sleu** rdest,rsource1,source2

## Condizione verificata

rsource1 == source2  
rsource1 >= source2 (signed)  
rsource1 >= source2 (unsigned)  
rsource1 > source2 (signed)  
rsource1 > source2 (unsigned)  
rsource1 <= source2 (signed)  
rsource1 <= source2 (unsigned)

### Nota:

rdest e rsource1 sono registri, mentre source2 può essere un registro o un immediato a 16 bit.  
Non c'è sign-extension, ma solo zero-extension

## 4.5.2 Istruzioni per il controllo di flusso

Consentono di **alterare il flusso di esecuzione** delle istruzioni del programma, trasferendo il controllo a un'istruzione diversa. Sono essenziali per la realizzazione dei costrutti.

La logica richiama molto quella dell'istruzione "go to".

Distinguiamo 2 diverse tipologie di istruzioni per il controllo del flusso:

Istruzioni di branch (salto condizionato)

## Istruzioni di branch

<b>beq \$t1,\$t2,label</b>	<i>if (\$t1 == \$t2)</i> <i>branch label</i>	<i>Test di uguaglianza</i>
<b>bne \$t1,\$t2,label</b>	<i>if (\$t1!= \$t2)</i> <i>branch label</i>	<i>Test di disuguaglianza</i>
<b>bgez \$t1,label</b>	<i>if (\$t1&gt;= \$zero)</i> <i>branch label</i>	<i>Verifica se il valore in \$t1 è non negativo</i>
<b>bgtz \$t1,label</b>	<i>if (\$t1&gt; \$zero)</i> <i>branch label</i>	<i>Verifica se il valore in \$t1 è positivo</i>
<b>blez \$t1,label</b>	<i>if (\$t1&lt;= \$zero)</i> <i>branch label</i>	<i>Verifica se il valore in \$t1 è non positivo</i>
<b>bltz \$t1,label</b>	<i>if (\$t1&lt; \$zero)</i> <i>branch label</i>	<i>Verifica se il valore in \$t1 è negativo</i>

## Pseudoistruzioni di branch

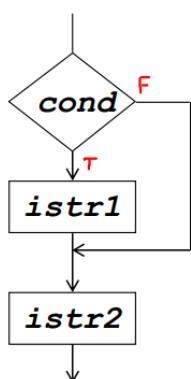
<b>b label</b>	salto incondizionato
<b>beqz rs,label</b>	$rs == 0$
<b>bge rs1,rs2,label</b>	$rs1 \geq rs2$
<b>bgeu rs1,rs2,label</b>	$unsigned(rs1) \geq unsigned(rs2)$
<b>bgt rs1,rs2,label</b>	$rs1 > rs2$
<b>bgtu rs1,rs2,label</b>	$unsigned(rs1) > unsigned(rs2)$
<b>ble rs1,rs2,label</b>	$rs1 \leq rs2$
<b>bleu rs1,rs2,label</b>	$unsigned(rs1) \leq unsigned(rs2)$
<b>blt rs1,rs2,label</b>	$rs1 < rs2$
<b>bltu rs1,rs2,label</b>	$unsigned(rs1) < unsigned(rs2)$
<b>bnez rs,label</b>	$rs != 0$

Istruzioni di jump (salto incondizionato)

Istruzione	Significato
<b>j label</b>	jump label <i>Salta all'istruzione all'indirizzo label (indirizzo da 26 bit)</i>
<b>jr \$t0</b>	jump (\$t0) <i>Salta all'istruzione all'indirizzo contenuto nel registro \$t0 (indirizzo da 32 bit)</i>
<i>Uso particolare di jr</i>	
<b>jr \$ra</b>	jump (\$ra) <i>Per i ritorni da procedura (indirizzo da 32 bit)</i>

## 4.6 Pattern

### 4.6.1 Costrutto if-then



```

if (cond)
  istr1;
istr2;
  
```

Esempio:

```

if (a>max)
  max=a;
  
```

```

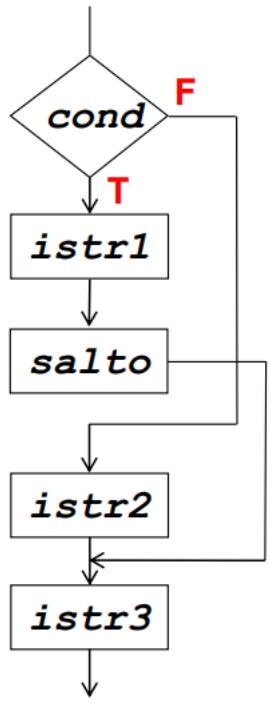
lw $t0, a
lw $t1, max
ble $t0, $t1, next
then: sw $t0, max
next:
  
```

La logica è invertita:

```

if (a<=max) goto next
  
```

### 4.6.2 Costrutto if-then-else



```

if (cond)
    istr1;
else
    istr2;
istr3

```

```

li $t0, 10
li $t1, 25
ble $t0, $t1, else

```

Esempio:

```

if (a>b)
    max=a;
else
    max=b;
c=max*2;

```

```

then: move $t2, $t0
      j next
else: move $t2,$t1
next: sll $t3, $t2,1

```

La logica è invertita:

```

if (a<=b) goto else
then: max=a;
      goto next
else: max=b
next: c=max*2

```

#### 4.6.3 Costrutto while

```

while (a>b)
    {blocco}

```

La logica è invertita:

```

while: if (a<=b) goto next
      {blocco}
      goto while
next:

```

```

lw $t0, a
lw $t1, b
while: ble $t0, $t1, next
# {blocco}
j while

```

Analogamente avviene per il costrutto for il quale si riconduce a un costrutto while.

### 4.7 Le direttive e le strutture dati

Le direttive servono a far svolgere particolari azioni all'assemblatore.

#### 4.7.1 Definizione di segmenti di memoria

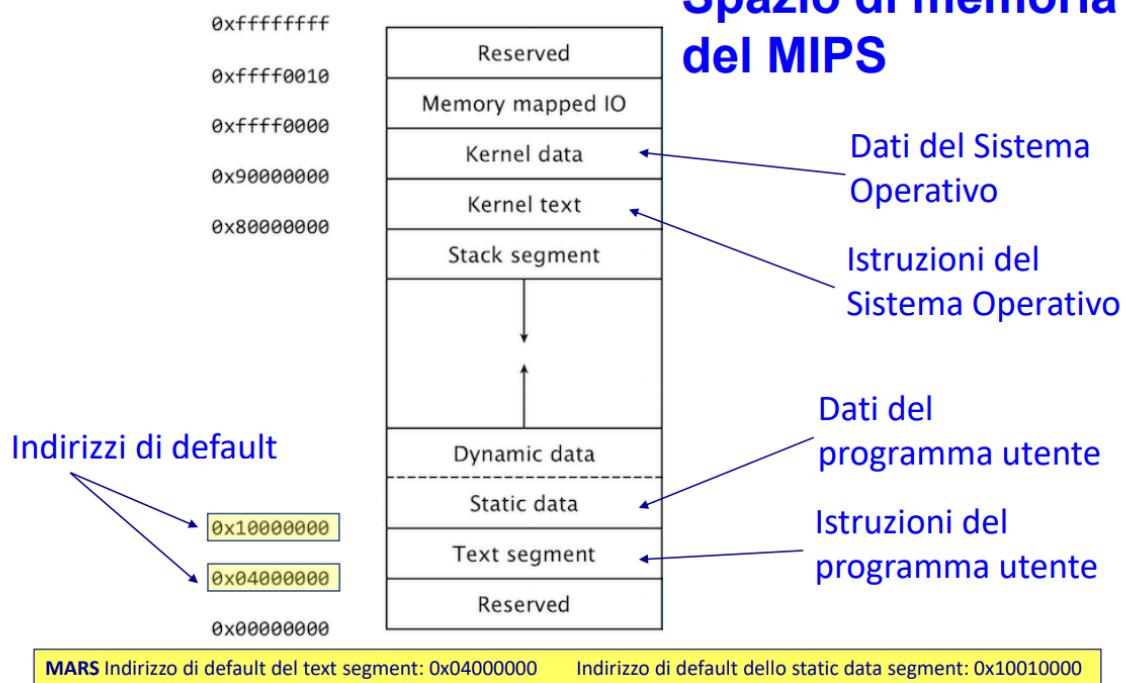
`.data <indirizzo>` Consente di memorizzare gli elementi successivi alla direttiva nel segmento dati.

`.text <indirizzo>` Consente di memorizzare gli elementi successivi alla direttiva nel segmento testo.

`.kdata <indirizzo>` Consente di memorizzare gli elementi successivi alla direttiva nel segmento dati del sistema operativo.

`.ktext <indirizzo>` Consente di memorizzare gli elementi successivi alla direttiva nel segmento testo del sistema operativo.

## Spazio di memoria del MIPS



### 4.7.2 Inizializzazione dell'area dati

`.byte b1, b2, ..., bn` Memorizza  $n$  valori da un byte in byte consecutivi della memoria.

`.half h1, h2, ..., hn` Memorizza  $n$  valori da un half word in byte consecutivi della memoria;

`.word w1, w2, ..., wn` Memorizza  $n$  valori da una word in byte consecutivi della memoria

`.ascii string` Memorizza i caratteri della stringa string in byte consecutivi della memoria. Ad ogni carattere corrisponde un byte;

`.asciiz string` Memorizza i caratteri della stringa string in byte consecutivi della memoria aggiungendo un valore 0 in coda. Ad ogni carattere corrisponde un byte.

#### ESEMPI

- `.byte 20` riserva un byte inizializzato a 20
- `.byte -1, 0xFF` riserva due byte inizializzati a 0xFF
- `.byte 'A', '0'` riserva due byte inizializzati a 65 e 48 (fa riferimento ai relativi caratteri ASCII)
- `.half 20` riserva un halfword inizializzato a 20
- `.half -1, 5` riserva due halfword inizializzati a 0xFFFF e 5
- `.word 20` riserva una word inizializzata a 20
- `.word -1` riserva una word inizializzata a 0xFFFFFFFF
- `.ascii "pippo"` riserva 5 byte inizializzati con i caratteri della stringa
- `.asciiz "pippo"` riserva 5 byte inizializzati con i caratteri della stringa e aggiunge uno 0 in coda

Per far riferimento ai singoli dati memorizzati si utilizza la pseudoistruzione `la $t0, <addr>` che carica nel registro \$t0 l'**indirizzo** contenuto nel secondo registro.

#### ESEMPIO

registro base							
\$s3	0110	1101	1101	0010	0101	1110	0000 1100
offset	0000	0000	0000	0100			
\$t0	0110	1101	1101	0010	0101	1110	0001 0000
<b>la \$t0, 4(\$s3)</b>							
\$a0	0001	0000	0000	0001	0000	0000	1000
<b>la \$a0, nome</b>							

#### 4.7.3 Allineamento dati

.align n Allinea il dato successivo a blocchi di  $2^n$  byte

ESEMPIO

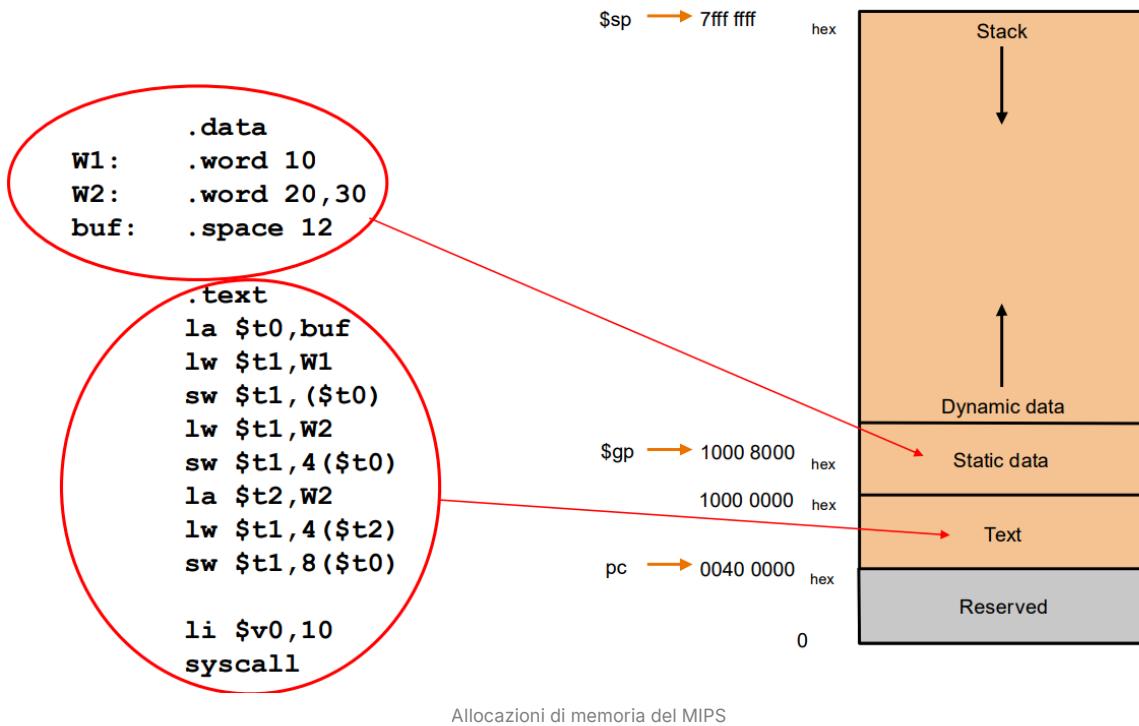
- .align 2 allinea alla word i dati successivi;
- .align 0 elimina l'allineamento automatico

	Indirizzo	registro	
	0001 0000 0000 0001 0000 0000 0000 0000 0000	0111 0100	't'
	0001 0000 0000 0001 0000 0000 0000 0001	0110 1001	'i'
	0001 0000 0000 0001 0000 0000 0000 0010	0111 0100	't'
	0001 0000 0000 0001 0000 0000 0000 0011	0110 1111	'o'
.data	0001 0000 0000 0001 0000 0000 0000 0100	0000 0000	'\0'
nome: .asciiz "tito"	0001 0000 0000 0001 0000 0000 0000 0101		
.align 2	0001 0000 0000 0001 0000 0000 0000 0110		
num: .word 20	0001 0000 0000 0001 0000 0000 0000 0111		
	0001 0000 0000 0001 0000 0000 0000 1000	0000 0000	
	0001 0000 0000 0001 0000 0000 0000 1001	0000 0000	
	0001 0000 0000 0001 0000 0000 0000 1010	0000 0000	
	0001 0000 0000 0001 0000 0000 0000 1011	0001 1110	
	0001 0000 0000 0001 0000 0000 0000 1100		
	0001 0000 0000 0001 0000 0000 0000 1101		

La direttiva .space n invece viene utilizzata per allocare n byte **non inizializzati** a partire dall'indirizzo corrente, nel segmento .data.

ESEMPIO

	Indirizzo	registro	
	0001 0000 0000 0001 0000 0000 0000 0000	0010 1010	
	0001 0000 0000 0001 0000 0000 0000 0001	1011 1001	0x2AB5 half
	0001 0000 0000 0001 0000 0000 0000 0010	0000 0000	byte
	0001 0000 0000 0001 0000 0000 0000 0011	0001 1110	byte
	0001 0000 0000 0001 0000 0000 0000 0100	0000 1010	byte
	0001 0000 0000 0001 0000 0000 0000 0101	1111 1111	byte
	0001 0000 0000 0001 0000 0000 0000 0110	0010 1010	byte
	0001 0000 0000 0001 0000 0000 0000 0111	1011 1001	byte
	0001 0000 0000 0001 0000 0000 0000 1000	0111 0100	't'
	0001 0000 0000 0001 0000 0000 0000 1001	0110 1001	'i'
	0001 0000 0000 0001 0000 0000 0000 1010	0111 0100	't'
	0001 0000 0000 0001 0000 0000 0000 1011	0110 1111	'o'
	0001 0000 0000 0001 0000 0000 0000 1100	0000 0000	'\0'
	0001 0000 0000 0001 0000 0000 0000 1101		byte
<b>.data</b>			
<b>value:</b> .half 0x2AB5			
<b>res:</b> .space 6			
<b>nome:</b> .asciiz "tito"			
<b>Etichetta</b>	<b>Indirizzo</b>		
<b>value</b>	0001 0000 0000 0001 0000 0000 0000 0000		
<b>res</b>	0001 0000 0000 0001 0000 0000 0000 0010		
<b>nome</b>	0001 0000 0000 0001 0000 0000 0000 1000		



Le **syscall** (chiamate al sistema operativo) sono particolari direttive che consentono di invocare funzionalità del sistema operativo. La maggior parte di esse servono per operazioni input/output.

Generalmente la chiamata avviene nel seguente modo:

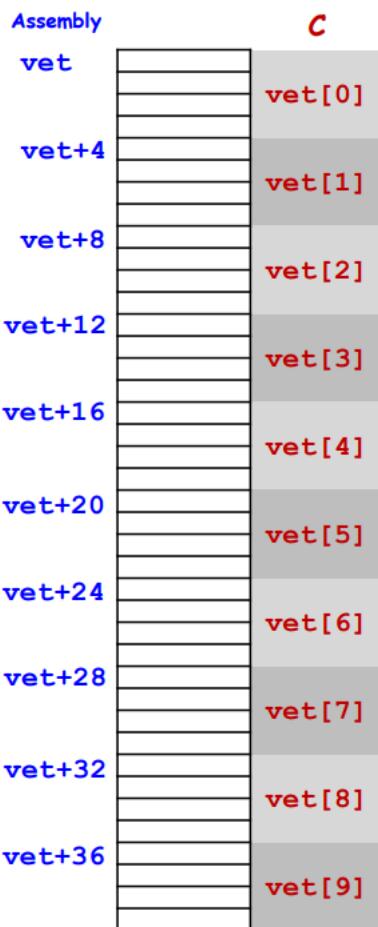
- Si carica il codice operativo della system call nel registro \$v0
- Si caricano gli argomenti nei registri \$a0, \$a1 \$a2
- Si esegue l'istruzione **syscall**
- Si copiano i risultati in registri d'appoggio per evitare che vengano persi

Funzione di Servizio	Codice di chiamata di sistema	Argomenti	Risultato
print_int	1	\$a0 = intero	
print_float	2	\$f12 = virgola mobile, singola pr.	
print_double	3	\$f12 = virgola mobile, doppia pr.	
print_string	4	\$a0 = stringa	
read_int	5		Intero (in \$v0)
read_float	6		Virgola mobile, singola pr. (in \$v0)
read_double	7		Virgola mobile, doppia pr. (in \$v0)
read_string	8	\$a0 = buffer, \$a1 = lunghezza	
sbrk	9	\$a0 = quantità	Indirizzo (in \$v0)
exit	10		
print_char	11	\$a0 = carattere	
read_char	12		Carattere (in \$v0)
open	13	\$a0 = nome file (stringa), \$a1 = flags, \$a2 = modalità	Descrittore del file (in \$a0)
read	14	\$a0 = descrittore del file, \$a1 = buffer, \$a2 = lunghezza	Num. caratteri letti (in \$a0)
write	15	\$a0 = descrittore file, \$a1 = buffer, \$a2 = lunghezza	Num. caratteri scritti (in \$a0)
close	16	\$a0 = descrittore del file	
exit2	17	\$a0 = risultato	

## 4.8 Le strutture dati

In assembly le strutture dati devono essere gestite manualmente... prendendo per esempio in considerazione un vettore, bisogna stabilire il tipo, il nome e il numero di byte che occupa ogni singolo elemento.

ESEMPIO - caricamento di un array



```
int vet [10];
int main(){
    int i;
    for (i=0;i<10;i++)
        vett[i]=2+i;
}
```

```
.data
vet: .space 40

.text
la $t0, vet
li $t1, 10
li $t2, 0
for: bge $t2, $t1, fine
      addi $t3, $t2, 2
      sw $t3, ($t0)
      addiu $t0, $t0, 4
      addiu $t2, $t2, 1
      j for
fine: li $v0, 10
      syscall
```

### ESEMPIO: somma elementi di un array

```
int vet []={7,5,-2,4};
int num=4;
int main(){
    int i,n,s,x;
    s=0;
    num=n;
    for (i=0;i<n;i++){
        x=vet[i];
        s=s+x;
    }
    printf ("%d",x);
}
```

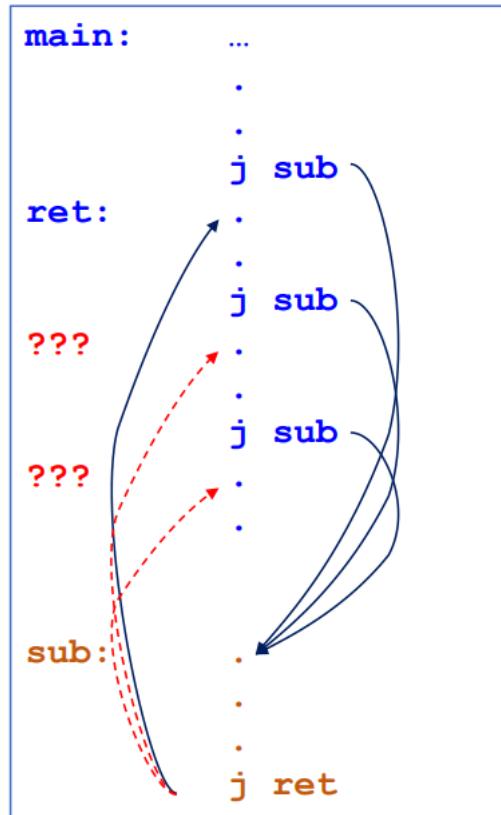
```
.data
vet: .word 7,5,-2,4
num: .word 4

.text
la $t0, vet
lw $t1, num
li $t2, 0
li $t3, 0
for: bge $t2, $t1, fine
      lw $t4, ($t0)
      addi $t3, $t3, $t4
      addiu $t0, $t0, 4
      addiu $t2, $t2, 1
      j for
fine: move $a0, $t3
      li $v0, 1
      syscall
      li $v0, 10
      syscall
```

## 4.9 I sottoprogrammi

In assembly non esiste un supporto diretto per l'implementazione dei **sottoprogrammi**... bisogna gestire manualmente il passaggio dei parametri e il controllo.

In primis non é possibile utilizzare le classiche istruzioni di salto incondizionato dal momento che verrebbe meno il principio di riusabilità del sottoprogramma.



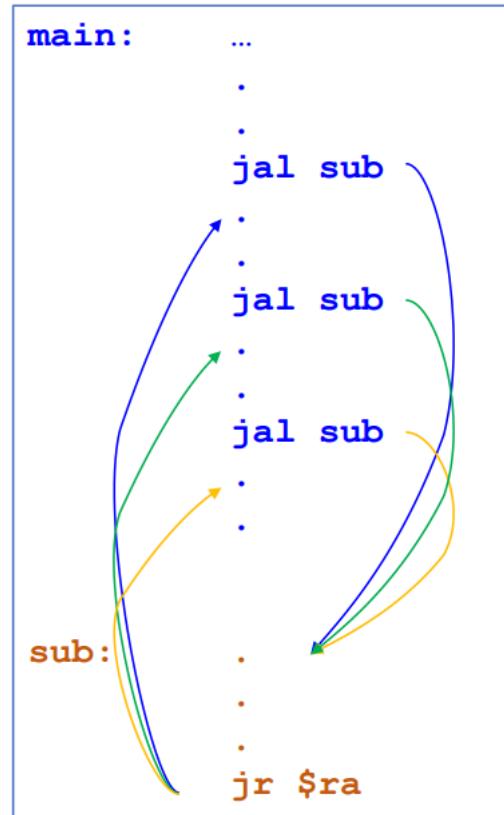
Se il chiamante invocasse il sottoprogramma in piú punti, sfruttando l'istruzione `j <label>` (jump to label), il controllo tornerebbe sempre allo stesso punto, il punto in cui é presente l'etichetta.

Per ovviare a questo problema, oltre ad effettuare il salto, bisogna aggiornare là indirizzo di ritorno. A tal proposito si utilizza l'istruzione `jal` (jump and link) che salva, nel registro `$ra` l'indirizzo dell'istruzione successiva al salto.

Più precisamente il registro \$ra conterrà il valore:  $\$ra = \$pc + 4$ , l'indirizzo di ritorno della prima istruzione da eseguire quando il controllo ritorna al chiamante. Il registro \$pc contiene invece l'indirizzo del sottoprogramma "sub".

Alla fine del sottoprogramma dovrà quindi essere inserita l'istruzione `jr $ra` (jump on register) per effettuare il salto all'indirizzo di ritorno. Il controllo in questo modo torna al chiamante.

Per quanto riguarda la gestione dei dati invece, il MIPS prevede una serie di convenzioni dette ***"calling conventions"***.



Il passaggio dei parametri generalmente sfrutta i **registri** interni:

- I registri **\$a0-\$a3** (arguments) devono contenere i parametri effettivi, quelli che il chiamante dovrà passare al sottoprogramma. Sono modificabili dal sottoprogramma;
  - I registri **\$v0-\$v1** (values) conterranno invece i parametri formali, quelli che il sottoprogramma restituirà al chiamante. Sono modificabili dal sottoprogramma;

Dal momento che i registri \$a, \$v e \$t sono modificabili dal sottoprogramma, se il chiamante ha qualcosa di importante in questi registri, è tenuto a salvarlo altrove prima di invocare il sottoprogramma in quanto quest'ultimo potrebbe sovrascrivere i dati presenti.

I registri \$zero, \$s0-\$s7 e \$sp invece non sono modificabili dal sottoprogramma.

## Istruzioni per la chiamata di sottoprogramma

Sono particolari istruzioni per il controllo di flusso. Prima del salto, l'indirizzo dell'istruzione successiva viene salvato nel registro \$ra (\$31).

**jal label**      \$ra = PC + 4;      *salto incondizionato*  
                  *jump label*

**jalr \$t1**      \$ra = PC + 4;      *salto incondizionato*  
                  *jump (\$t1)*

## ESEMPI

```
.text
max3:
    move $v0, $a0
    bge $v0, $a1, skip
    move $v0, $a1
skip:
    bge $v0, $a2, ret
    move $v0, $a2
ret:   jr $ra
```

```
# leggint -- mostra un messaggio e legge un intero
#
# Parametri di ingresso:
#     nessuno
#
# Valori in uscita:
#     $v0 -- valore letto
.text
leggint:
    la    $a0,message      # stampa messaggio di prompt
    li    $v0,4
    syscall

    li    $v0,5      # legge un intero da input
    syscall          # e lo memorizza in $v0

    jr    $ra          # return

.data
message:
    .asciiz "Fornisci un valore intero: "

# printint -- stampa un messaggio ed un intero forniti in input
#
# Parametri di ingresso:
#     $a0 -- messaggio di output
#     $a1 -- valore da stampare
#
# Valori in uscita:
#     nessuno
.text
printint:
    # stampa messaggio di output il cui indirizzo è già presente in $a0
    li    $v0,4
    syscall

    move  $a0, $a1          # stampa il valore intero passato
    li    $v0,1
    syscall

    la $a0, CRLF           # stampa un CRLF
    li $v0,4
    syscall

    jr    $ra          # return

.data
CRLF:   .asciiz "\n"
```

La principale limitazione di questo metodo riguarda il numero di registri disponibili:

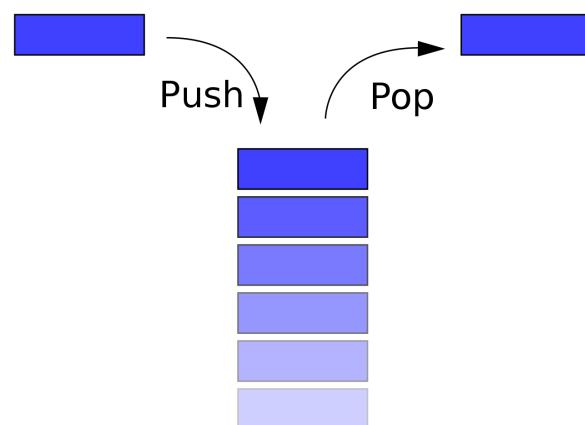
- il chiamante può infatti passare al sottoprogramma solamente **4 parametri**,
- la procedura non può far tornare al chiamante più di **2 parametri**

Inoltre occorre mantenere invariati alcuni registri che il chiamato potrebbe modificare facendo saltare tutto il meccanismo e trovare un sistema per allocare strutture dati in memoria. Il sistema basato sulle calling conventions non permette neanche di effettuare chiamate innestate.

Per risolvere questi problemi si utilizza lo **Stack**.

#### 4.5.1 Lo Stack

Lo **Stack** (pila) è un'area di memoria per l'allocazione di dati dinamici. Si tratta di una struttura dati del tipo **LIFO** (Last in First out)... i dati vengono messi gli uni sugli altri (tipo una pila) per cui, l'ultimo dato ad essere allocato sarà il primo ad essere servito.

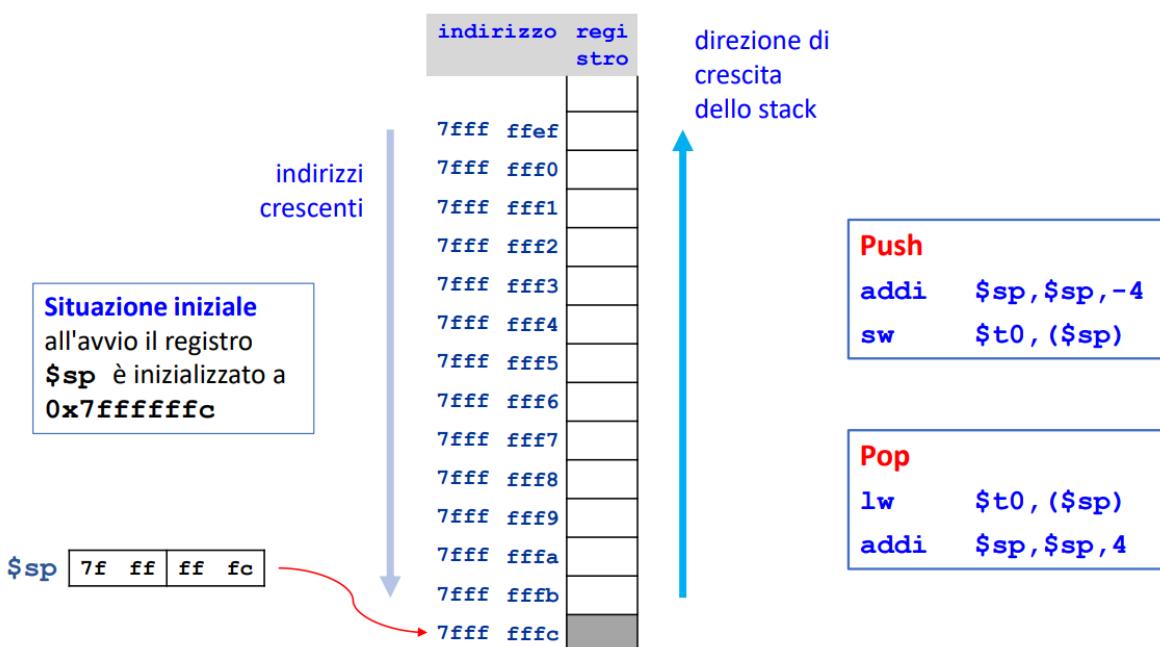


I dati vengono inseriti sullo stack con l'operazione di `push` vengono rimossi con l'operazione di `pop`.

Il registro `$sp` (stack pointer o puntatore allo stack) contiene l'indirizzo del top dello stack e viene aggiornato ad ogni operazione di `push` o `pop`. In particolar modo, per effettuare un'operazione di `push`, `$sp` viene **DECREMENTATO**, per effettuare `pop`, `$sp` viene **INCREMENTATO** (la dimensione dello stack quindi si riduce).

Questo avviene perché lo stack si espande per valori crescenti di `$sp`.

## Lo stack nel MIPS



ESEMPIO: Salvataggio di 2 registri sullo stack

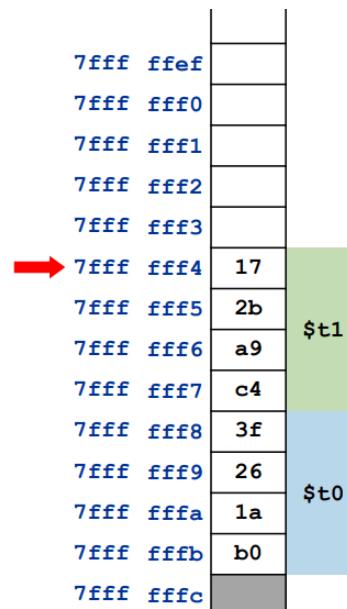
```

# push di $t0 e $t1
addi $sp,$sp,-4      addi $sp,$sp,-8
sw $t0,($sp)          sw $t0,4($sp)
addi $sp,$sp,-4      sw $t1,($sp)
sw $t1,($sp)

.
.
.

# pop di $t1 e $t0
lw $t1,($sp)          lw $t0,4($sp)
addi $sp,$sp,4         lw $t1,($sp)
lw $t0,($sp)           addi $sp,$sp,8
addi $sp,$sp,4

```



Lo stack può quindi essere impiegato come strumento alternativo per il passaggio di parametri. Il chiamante ha il compito di allocare sullo stack i parametri ingresso e predisporre lo spazio per quelli in uscita. Il sottoprogramma accede ai parametri in ingresso e al termine salva sullo stack i parametri in uscita. Al ritorno dal sottoprogramma, il chiamante preleva dallo stack i parametri in uscita e dealloca lo spazio precedentemente allocato.

Chiamante e chiamato devono concordare su:

- numeri di parametri
- dimensione dei parametri
- posizione sullo stack

## 5. Reti logiche e algebra booleana

### 5.1 I sistemi hardware

Un sistema hardware è un sistema realizzato mediante interconnessioni di componenti elettronici analogici o digitali: gli ingressi e le uscite di un sistema digitale assumono valori finiti e discreti. Nei **sistemi digitali** gli ingressi e le uscite vengono fisicamente rappresentate mediante valori discreti di tensione, in particolar modo, i sistemi digitali più semplici, considerano solamente due valori di tensione rappresentati convenzionalmente con 0 e 1. Il vantaggio di questi sistemi è la rigorosa base matematica su cui si fondano che deriva dall'**algebra booleana**.

Le stringhe di bit quindi, vengono rappresentate fisicamente mediante **tensioni elettriche** a basso voltaggio. Si suppone per esempio che un alto voltaggio equivale al bit 1 mentre un basso voltaggio equivale al bit 0.

Idealmente, la tensione nel tempo, disegna un grafico geometrico, **privò di sbavature...** nella realtà l'onda non è perfettamente geometrica per cui, per far sì che questo meccanismo funzioni, è necessario che il segnale venga interpretato come 0 o 1 anche se il livello di tensione non è precisamente quello di riferimento.



I circuiti di commutazione (o **reti logiche**) consentono di compiere delle elaborazioni su tali informazioni (sui bit) e costruire quindi, a partire da determinate costruzioni di bit, altre configurazioni che rispecchiano la codifica prefissata. Sostanzialmente sono proprio le reti logiche che consentono (per esempio) lo svolgimento delle principali operazioni matematiche ad un calcolatore.

Di questi sistemi digitali possiamo distinguerne due tipologie: **sistemi asincroni** e **sistemi sincroni**. Nei **sistemi sincroni** tutte le operazioni sono coordinate da un clock centrale, ciò significa che le uscite degli elementi cambiano valore solamente in istanti ben precisi di tempo. In un **sistema asincrono** invece le uscite possono variare in modo arbitrario nel tempo. Al giorno d'oggi la maggior parte dei sistemi hardware è sincrona.

Le reti logiche in cui le uscite dipendono esclusivamente e istantaneamente dai valori che gli ingressi assumono prendono il nome di **reti combinatorie**. Le reti in cui i valori di uscita dipendono dalla sequenza temporale degli ingressi sono dette **reti sequenziali**.

I sistemi hardware possono essere descritti sotto 3 diversi aspetti (livelli):

- **livello comportamentale:** in che modo funziona il sistema? Qual è il suo compito?
- **livello logico:** definisce l'interconnessione degli elementi logici di base. Sostanzialmente avviene la "traduzione" in circuito logico del sistema;
- **livello circuitale:** i componenti base sono rimpiazzati dai dispositivi elettronici che implementano i componenti logici

## 5.2 Algebra booleana: le funzioni elementari

L'algebra Booleana prende il nome dal matematico inglese John Boole che formalizzò un sistema algebrico per trattare in modo sistematico la logica. Una restrizione dell'algebra booleana al sistema binario è la cosiddetta **algebra di commutazione** introdotta da Claude Shannon a metà del ventesimo secolo. Con il termine "algebra booleana" si fa in realtà spesso riferimento all'algebra di commutazione.

Nell'algebra booleana sono ammesse **3 operazioni elementari**:

AND	OR	NOT
$0 * 0 = 0$	$0 + 0 = 0$	$!0 = 1$
$0 * 1 = 0$	$0 + 1 = 1$	$!1 = 0$
$1 * 0 = 0$	$1 + 0 = 1$	
$1 * 1 = 1$	$1 + 1 = 1$	

Ogni variabile può infatti essere definita come **funzione di altre variabili**:  $w = f(x, y, z)$ . Ciò ci consente di identificare le **funzioni logiche elementari**:

- $z = x * y$  funzione AND
- $z = x + y$  funzione OR

- $z = !x$  funzione NOT

Le funzioni elementari and, or e not costituiscono un insieme **funzionalmente completo** in quanto consentono di realizzare qualunque circuito.

E godono delle seguenti proprietà:

• Comutativa:	$a+b=b+a$	$a*b=b*a$
• Associativa:	$(a+b)+c=a+(b+c)$	$(a*b)*c=a*(b*c)$
• Idempotenza:	$(a+a)=a$	$(a*a)=a$
• Assorbimento:	$a+(a*b)=a$	$a*(a+b)=a$
• Distributiva:	$a*(b+c)=a*b+a*c$	$a+(b*c)=(a+b)*(a+c)$
• Min e max:	$a*0=0$	$a+1=1$
• Elem.to neutro:	$a+0=a$	$a*1=a$
• Complemento:	$a*(!a)=0$	$a+(!a)=1$
• De Morgan:	$!(a+b)=!a!*!b$	$!(a*b)=!a+!b$

Dal momento che l'insieme su cui si basa l'algebra booleana è finito, in 2 variabili è possibile definire:

x	y	$f(x,y)$															
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	0	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1


  
**AND**      **XOR**      **OR**      **EQU**

Ogni variabile x e y può assumere solamente 2 combinazioni possibili: 0 e 1. Per cui in totale sono possibili  $4^2 = 16$  combinazioni.

## 5.4 Minimizzazione

L'algebra di Boole consente di esprimere in forma algebrica le funzioni dei circuiti fornendo dei metodi per l'analisi e la sintesi. Le tabelle di verità vengono sostanzialmente convertite in espressioni algebriche.

- Una variabile affermata o negata prende il nome di **letterale**,
- Un prodotto o somma di letterali è detto **termine**,
- si definisce **mintermine** il prodotto di letterali di tutte le variabili di una funzione
- si definisce **maxtermine** la somma dei letterali di tutte le variabili di una funzione

### ESEMPIO: espressione booleana

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

$$f = !x!yz + !xy!z + x!y!z + x!yz + xy!z + xyz$$

$$f = (x + y + z)(x + !y + !z)$$

Le due espressioni sono equivalenti, nel primo caso si considera la somma dei mintermini (**SP**), nel secondo caso il prodotto di maxtermini (**PS**).

Come si evince dall'esempio sovrastante esistono 2 forme canoniche in cui è possibile esprimere le funzioni booleane:

- Somma di mintermini (o somma di termini prodotto) e si indica con SP
- Prodotto di maxtermini (o prodotto di termini somma) e si indica con PS.

Tali formule sono conseguenza del teorema di espansione di Boole.

### Teorema di espansione di Shannon

Considerata  $f$  una funzione booleana in  $n$  variabili definita in  $B^n$  si ha, per tutti gli  $(x_1, x_2, \dots, x_n)$ :

$$f(x_1, x_2, \dots, x_n) = x'_1 \cdot f(0, x_2, \dots, x_n) + x_1 \cdot f(1, x_2, \dots, x_n)$$

Ugualmente in forma PS vale:

$$f(x_1, x_2, \dots, x_n) = (x'_1 + f(1, x_2, \dots, x_n)) \cdot (x_1 + f(0, x_2, \dots, x_n))$$

Il teorema ci consente di espandere una generica funzione a  $n$  varibili nelle due forme:

$$f(x_1, x_2, \dots, x_n) =$$

$$\begin{aligned} & f(0, \dots, 0, 0) \cdot x'_1 \dots x'_{n-1} x'_n \\ & + f(0, \dots, 0, 1) \cdot x'_1 \dots x'_{n-1} x_n \\ & + f(0, \dots, 1, 0) \cdot x'_1 \dots x_{n-1} x'_n \\ & \quad \dots \\ & f(1, \dots, 1, 1) \cdot x_1 \dots x_{n-1} x_n \end{aligned}$$

Questa forma è detta prima forma canonica (o egualmente forma SP) e i termini

$$f(0, \dots, 0, 0) \quad f(1, \dots, 0, 0) \quad \dots \quad f(1, \dots, 1, 1)$$

sono detti discriminanti.

### Esempio - Minimizzazione con teorema di espansione

Supponiamo di voler espandere la seguente funzione applicando iterativamente il teorema di espansione:

Si voglia, per esempio, ricavare l'espressione in prima forma canonica della funzione di tre variabili  $f(x, y, z) = x + y'z$ . Applicando iterativamente il teorema di espansione di Shannon si ottiene:

$$\begin{aligned} f(x, y, z) &= x'f(0, y, z) + xf(1, y, z) \\ &= x'[y'f(0, 0, z) + yf(0, 1, z)] + x[y'f(1, 0, z) + yf(1, 1, z)] \\ &= x'y'z'f(0, 0, 0) + x'y'zf(0, 0, 1) + x'yz'f(0, 1, 0) + \\ &\quad + x'yzf(0, 1, 1) + xy'z'f(1, 0, 0) + xy'zf(1, 0, 1) + \\ &\quad + xyz'f(1, 1, 0) + xyzf(1, 1, 1). \end{aligned}$$

Calcolando i valori dei discriminanti si ottiene:

$$\begin{array}{llll} f(0, 0, 0) = 0 & f(0, 0, 1) = 1 & f(0, 1, 0) = 0 & f(0, 1, 1) = 0 \\ f(1, 0, 0) = 1 & f(1, 0, 1) = 1 & f(1, 1, 0) = 1 & f(1, 1, 1) = 1 \end{array}$$

e infine, sostituendo questi valori nell'espressione espansa, si ha:

$$f(x, y, z) = xy'z' + xyz' + xy'z + xyz + x'y'z$$

Il teorema può essere sfruttato anche per tradurre una tabella di verità nella relativa forma algebrica:

x	y	f
0	0	1
0	1	0
1	0	1
1	1	0

$$f(x, y) = f(0, 0) \cdot x'y' + f(0, 1) \cdot x'y + f(1, 0) \cdot xy' +$$

Dalla tabella di verità si deducono le uguaglianze:

$$f(0, 0) = 1 \quad f(0, 1) = 0 \quad f(1, 0) = 1 \quad f(1, 1) = 0$$

Effettuando la sostituzione nella formula precedente si ottiene:

$$f(x, y) = x'y' + xy'$$

Nel caso in cui comparisse una funzione non completamente specificata si procede assegnando al relativo discriminante una variabile del tipo  $p_i$ .

Dal momento che una funzione booleana può essere rappresentata da un numero infinito di espressioni booleane conviene scegliere l'espressione minima, quella corrispondente (successivamente) al circuito a costo minore.

I metodi per la **minimizzazione** delle funzioni booleane si basano sulle proprietà dell'algebra di Boole, in particolar modo le seguenti entità:

$$ab + a'b = a$$

$$(a + b) * (a + !b) = a$$

Due mintermini che differiscono per un solo letterale generano un **consenso**. I letterali che differiscono possono essere semplificati.

### Minimizzazione funzione booleana con proprietà dell'algebra

Minimizziamo la funzione:

$$f = !x!yz + !xy!z + x!y!z + x!yz + xy!z + xyz$$

$$\begin{aligned}
 & !x!yz + !xy!z + !yx + xy!z + xyz = \\
 & !x!yz + !xy!z + !yx + xy = \\
 & !xlyz + !xy!z + !yx + xy = \\
 & !y(!xz + x) + y(!x!z + x) = \\
 & !y(x + z) + y(z + x) = \\
 & !yx + !yz + y!z + yx = \\
 & f = x + !yz + y!z
 \end{aligned}$$

Evidenziati in **rosso** i consensi, in **blu** le varie messe in evidenza necessarie.

#### 5.4 Minimizzazione con le mappe di Karnaugh

Una funzione booleana può anche essere rappresentata graficamente mediante le **mappe di Karnaugh**. Si presenta come una tabella in cui ogni riga e colonna è etichettata con numeri binari i quali identificano il mintermine associato a ciascuna cella. Ogni cella quindi rappresenta un mintermine della funzione.

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

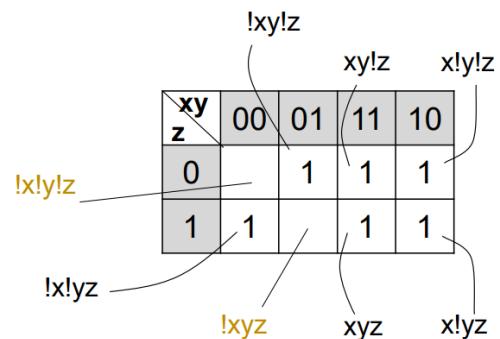
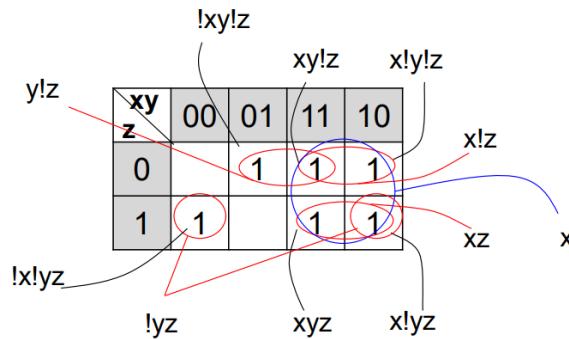


figura 10

Si riempiono solamente le celle per cui il valore della funzione è **pari a 1**, le altre celle sono lasciate vuote.

La scelta di utilizzare dei numeri binari crescenti come indici di righe e colonne consente di disporre i mintermini che generano consenso su **celle adiacenti** (rispetto le righe o le colonne).

In sostanza le mappe di Karnaugh semplificano la minimizzazione delle espressioni booleane mettendo in evidenza i vari consensi.



Mappa di Karnaugh della funzione precedente

Un minterme si dice **implicante** quando la funzione restituisce 1. Considerando la tabella di sopra,  $\neg x \neg y z$  (corrispondente alla riga 001) è un implicante di  $f$  in quanto la funzione restituisce 1.

Si definisce **implicante primo** invece, un implicante che non è incluso in altri implicanti con un numero minore di letterali.

Considerando la mappa superiore,  $x$ ,  $\neg y z$ ,  $y \neg z$  sono implicanti primi.

#### ESEMPIO - Definizione di implicanti primi

- $\neg a \neg b$  e  $\neg a d$  sono implicanti primi perché non sono inclusi in implicanti con un numero minore di letterali
- $\neg a b d$  è un implicante non primo perché è incluso nell'implicante  $\neg a d$

	$ab$	00	01	11	10
$cd$					
00	1	0	0	0	
01	1	1	0	0	
11	1	1	0	1	
10	1	0	0	1	

Si definisce **implicante primo essenziale** un minterme di una funzione incluso in uno solo degli implicanti primi.

#### ESEMPIO - Definizione di implicanti primi essenziali

Se un mintermine di una funzione è incluso in uno solo degli implicanti primi, quell'implicante è definito **essenziale**.

$\neg ab$  è un implicante primo essenziale perché include il mintermine  $\neg ab \neg c d$  che non è incluso in nessun altro implicante primo.

Ci sono altri implicanti primi essenziali?

	ab	00	01	11	10
cd					
00	0	1	0	0	
01	1	1	0	1	
11	0	1	1	1	
10	0	1	1	0	

Esistono alcune funzioni in cui non è specificato il valore che la funzione deve assumere per certe configurazioni di ingressi. In questi casi si parla di **funzioni non completamente specificate**.

z	xy	00	01	11	10
0	1	1	x	1	
1	1	x		x	

I mintermini non specificati si indicano con una "x" e sono detti "**don't care**" (o condizione di indifferenza).

Queste situazioni possono verificarsi principalmente per due motivi:

- non ha importanza quale valore assume la funzione
- la natura del problema garantisce che una data combinazione delle variabili in ingresso non si possa mai presentare

## Porte Logiche

Le **porte logiche** sono i componenti elettronici di base, dei circuiti digitali che hanno in ingresso uno o più segnali e producono un unico segnale di uscita. Consentono sostanzialmente di simulare le operazioni logiche mediante dei controlli sui segnali elettrici.

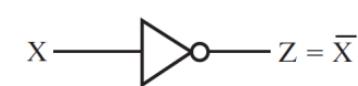
Le porte elementari:



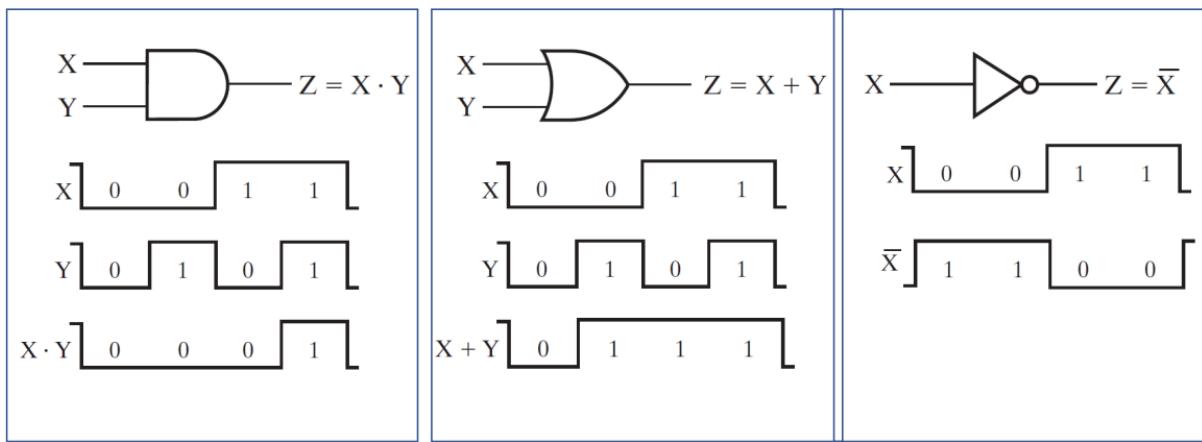
Porta AND



Porta OR



Porta NOT

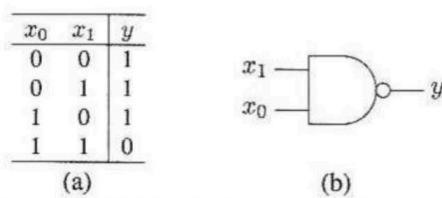


Le porte logiche rappresentano quindi le principali funzioni logiche, questo significa che **ogni funzione booleana può essere rappresentata mediante un circuito logico.**

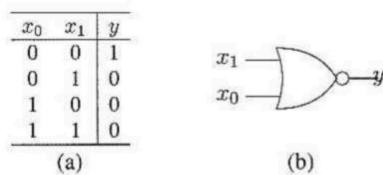
L'insieme di queste 3 porte è detto funzionalmente completo in quanto è possibile realizzare qualunque funzione logica.

Analogamente altri 2 insiemi funzionalmente completi sono rappresentati dagli operatori NAND e NOR a sé stanti. È possibile realizzare qualunque funzione logica utilizzando esclusivamente porte NAND o NOR.

La porta **NAND** mette sul segnale di uscita il complemento del prodotto logico dei segnali di ingresso ciò effettua la negazione dell'uscita di una porta AND.



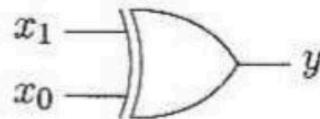
La porta **NOR** invece mette sul segnale di uscita il complemento della somma logica dei segnali in ingresso, sostanzialmente effettua la negazione dell'uscita di una porta OR.



La porta **XOR** (or esclusivo) realizza le funzionalità di disegualanza dei segnali in ingresso: l'uscita assume valore 1 se uno degli ingressi ha valore 1:

$x_0$	$x_1$	$y$
0	0	0
0	1	1
1	0	1
1	1	0

(a)



(b)

La porta **XNOR** ha funzionalità opposta rispetto alla porta XOR... mette quindi in uscita 1 se i due ingressi sono uguali.

$x_0$	$x_1$	$y$
0	0	1
0	1	0
1	0	0
1	1	1

(a)



(b)

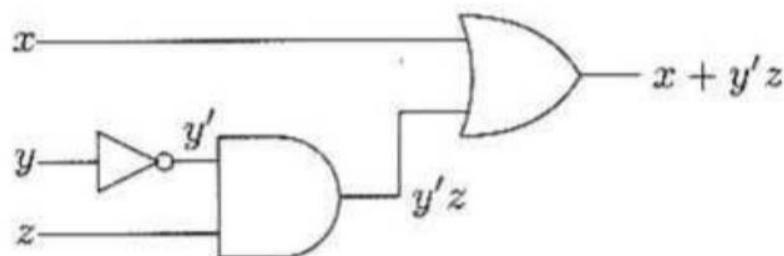
Le porte logiche possono essere combinate fra loro per realizzare dei circuiti logici i quali rappresentano delle funzioni logiche più complesse.

### Esempio - circuito logico

Considerata la funzione booleana

$$f = x + y'z$$

è possibile disegnare la relativa rete logica:

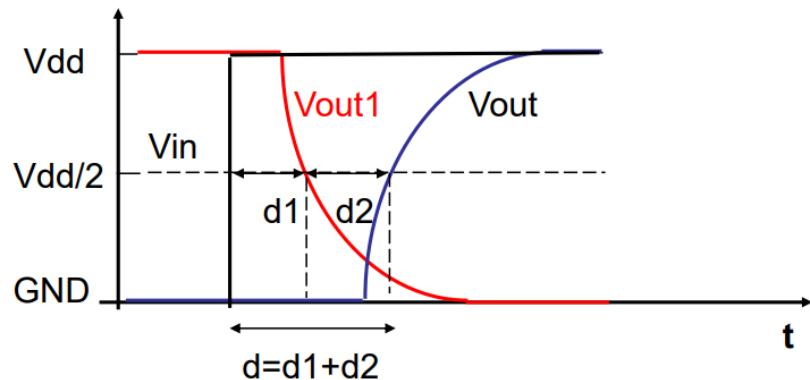
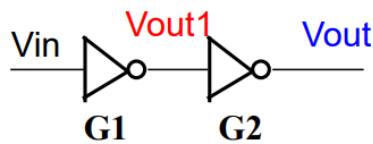
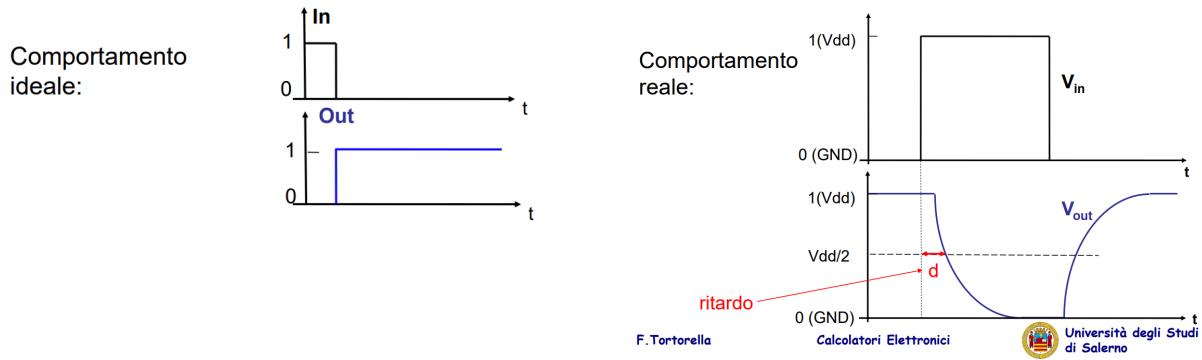


### Il ritardo di propagazione

Come già visto, un problema riguarda la **trasmissione non istantanea** dei segnali elettrici all'interno del calcolatore: è necessario del tempo affinché la tensione passi dal valore quota HV a LV o viceversa. Ciò significa è necessario un po di tempo prima che un bit passi da 0 a 1 o viceversa... è presente un leggero **ritardo**. Questo ritardo è legato alla realtà fisica dei componenti che formano le porte logiche (nella maggior parte di casi transistor).

Versione Ideale

Versione Reale



Con due porte in cascata il ritardo di propagazione è uguale alla **somma dei ritardi**. Un circuito composto da  $n$  porte presenta un ritardo pari alla somma dei ritardi di tutte le porte presenti.

Il percorso a ritardo maggiore che caratterizza l'intero circuito è detto **percorso critico (critical path)**.

Il ritardo di una rete logica si stabilisce quindi "contando" il numero di porte logiche presenti sul percorso critico, trascurando i negatori in quanto possono essere accorpati alle porte successive o precedenti.

## Le reti combinatorie

Una rete combinatoria è un circuito logico digitale formato da  **$n$  ingressi** ( $x_1, x_2, \dots, x_n$ ) ed  **$m$  uscite** ( $y_1, y_2, \dots, y_m$ ). Gli  $n$  ingressi possono assumere valori binari, ad ogni combinazione degli ingressi corrisponde una sola combinazione delle uscite.

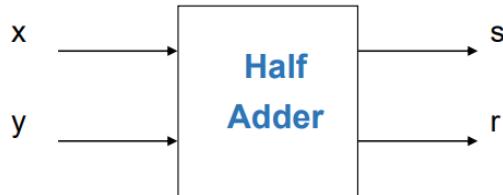
Dal punto di vista logico ogni uscita può essere definita come una funzione booleana degli ingressi:  $y_i = y_i(x_1, x_2, \dots, x_n)$ . Ad ogni istante il valore delle uscite dipende esclusivamente dal valore assunto dagli ingressi nello stesso istante. Questo significa che le uscite non dipendono dal tempo in quanto cambiano istantaneamente in base ai valori assunti dagli ingressi... si dice quindi che le uscite sono funzione degli ingressi.

Tale definizione si applica correttamente solo a reti ideali, poiché implica che non ci siano ritardi fra una modifica di un valore di ingresso e la corrispondente modifica dei valori di uscita. A causa dei ritardi differenti nell' "attraversamento" di porte logiche da parte dei segnali, si possono infatti verificare malfunzionamenti dei valori delle uscite al variare degli ingressi. Questi malfunzionamenti prendono il nome di alee.

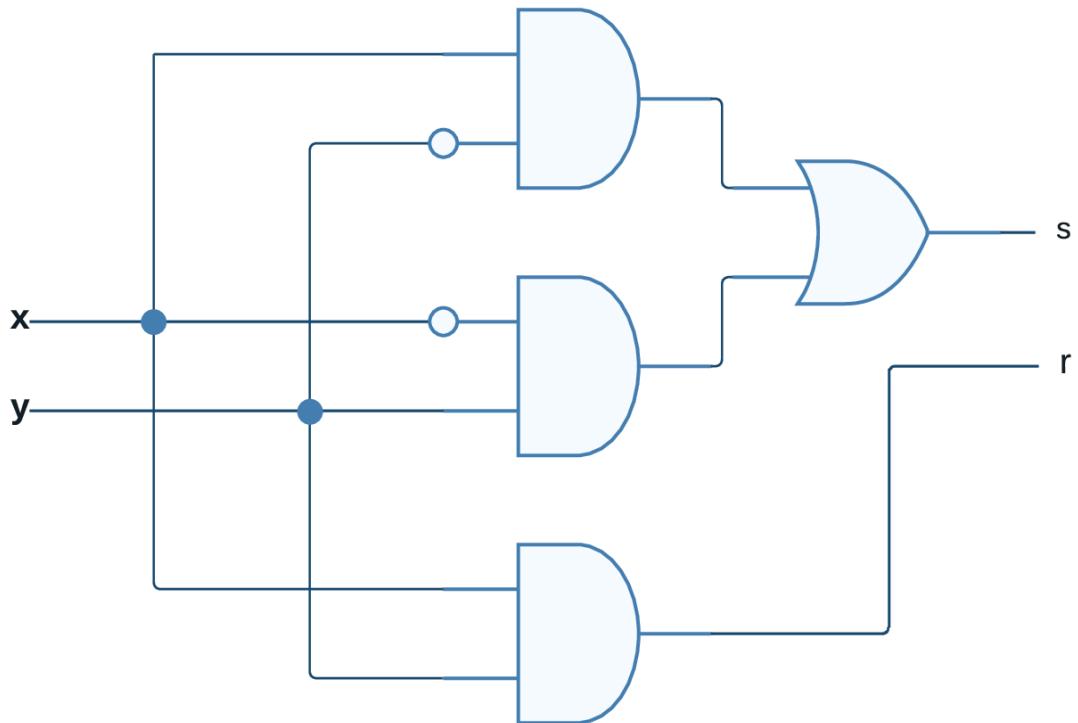
## ESEMPIO: 1-bit Half Adder

Un esempio è il **semiaddizionatore** (half adder), un circuito logico che effettua la somma di due cifre binarie. In ingresso si hanno le due cifre, sulla linea "x" e "y"... in uscita, rispettivamente sulla linea "s" verrà presentata la somma, sulla linea "r" il riporto se presente (si genera riporto solo se i 2 addendi sono pari a 1).

Il circuito prevede 2 ingressi, le cifre da sommare e 2 uscite, la somma e il riporto.



x	y	s	r
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



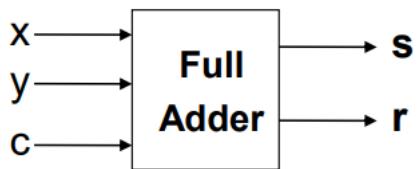
Schema logico Half-Adder

L'half adder presenta alcune limitazioni... in primis il fatto di non gestire il riporto.

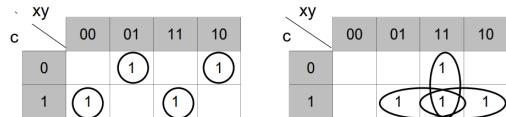
### 1-bit Full Adder

L'addizionatore completo, detto per l'appunto "**full-adder**", tiene conto dei riporti... si hanno pertanto 3 linee in ingresso: i due bit da sommare, x e y, il carry in ingresso (c). In uscita si hanno invece la somma (S) e il resto o carry uscente (r).

x	y	c	s
0	0	0	0
0	0	1	1

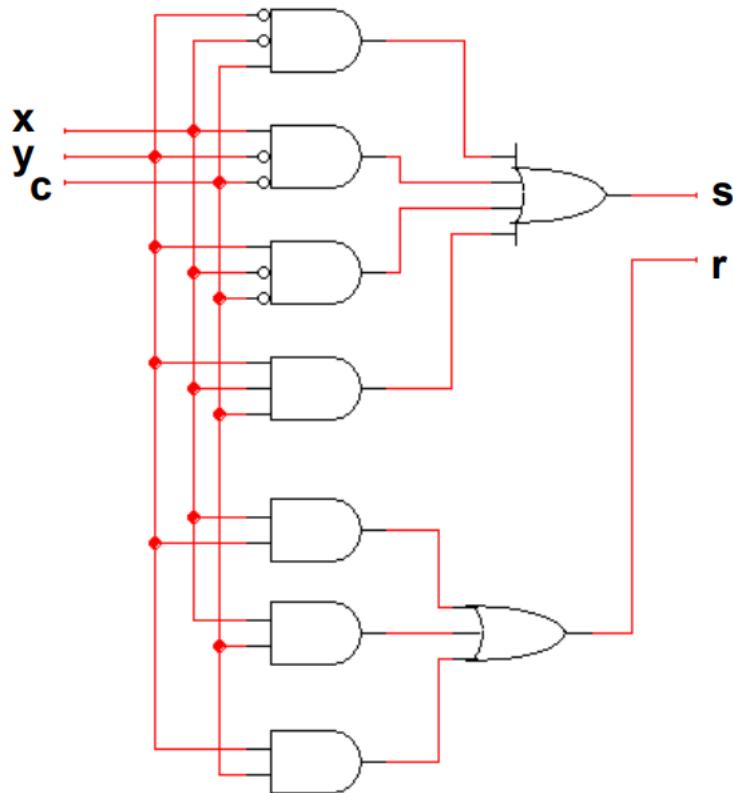


x	y	c	s
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



$$s = \overline{x}\overline{y}c + \overline{x}y\overline{c} + xy\overline{c} + x\overline{y}c$$

$$r = xy + yc + xc$$

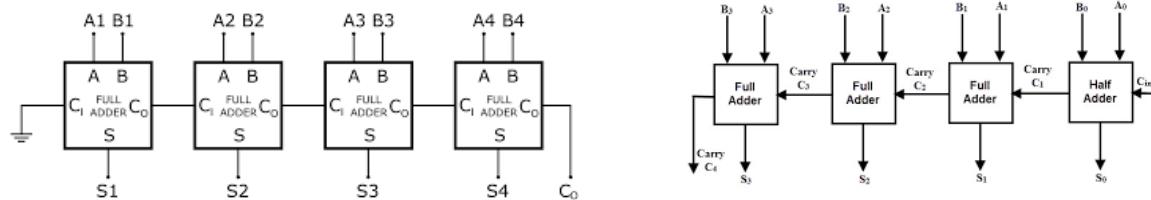


Il full-adder realizzato in questo modo consente di realizzare la somma tra numeri binari formati da **1 solo bit**, chiaramente tenendo conto dei carry.

#### 4-bit full-adder

Per realizzare un addizionatore ad  $n$  bit occorre collegare in serie  $n$  full adder (dove il primo FA presenta carry entrante pari a zero  $c = 0$ ) o in alternativa è possibile rimpiazzare il primo FA con un HA.

Questa rete combinatoria prende il nome di "sommatore a propagazione di riporto" o **ripple-carry adder**.

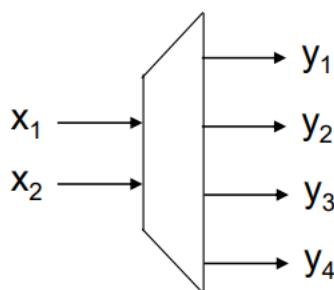


Questa architettura è semplice e compatta ma non particolarmente veloce in quanto, per effettuare la somma dei bit in i-esima posizione bisogna conoscere il **carry entrante**. Dal momento che ogni full-adder è realizzato mediante una rete su due livelli, il riporto (indicando con T il ritardo generico di una singola porta), è pronto dopo un tempo pari a  $2T$ . Dal momento che, nell'esempio è trattato un addizionatore a 4 bit, il ritardo complessivo è di  $4 \cdot 2T = 8T$ , e in generale per un sommatore ad  $n$  bit:  $2nT$ .

Quindi il punto critico nei sommatori ripple-carry sta proprio nella propagazione del riporto. Potendo disporre in anticipo di tutti i riporti, le somme potrebbero essere calcolate in parallelo.

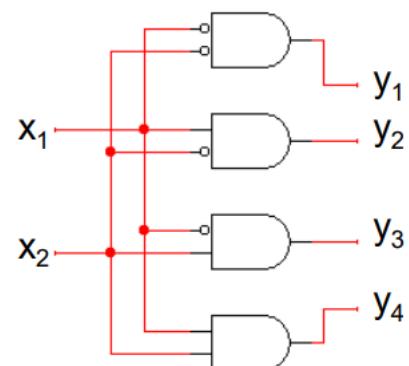
### Decoder (o decodificatore)

Il **decodificatore** è un circuito che presenta  $n$  ingressi e  $2^n$  uscite e permette sostanzialmente di attivare una sola uscita selezionata da un numero binario in ingresso. Questo significa che sarà presente una sola uscita pari a 1.



$x_1$	$x_2$	$y_1$	$y_2$	$y_3$	$y_4$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

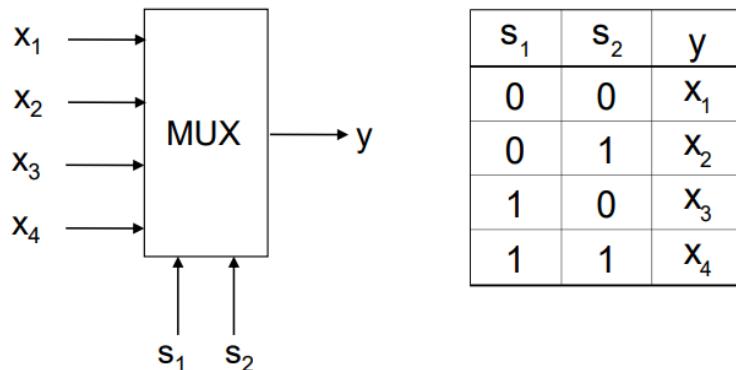
$$\begin{aligned}y_1 &= !x_1!x_2 \\y_2 &= !x_1x_2 \\y_3 &= x_1!x_2 \\y_4 &= x_1x_2\end{aligned}$$



### Multiplexer (o selettore)

Il **multiplexer** (spesso abbreviato **MUX**) è un circuito combinatorio che manda un primo ingresso da distribuire ad una uscita selezionata da un numero binario posto su un gruppo di ingressi di selezione. Uno dei segnali di selezione "s" consente di portare sull'uscita "y" uno dei segnali di ingresso  $x_i$ .

Gli ingressi sono divisi in "ingressi dati" e sono pari ad  $n$  e ingressi selezione (pari a  $\log_2(n)$ ). L'uscita è unica e uguale ad uno degli ingressi dati, scelto sulla base degli ingressi selezione.



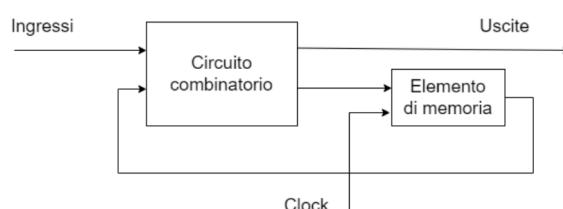
## Le reti sequenziali

Come abbiamo visto la principale caratteristica delle reti a logica combinatoria è il fatto che le uscite dipendono istantaneamente dagli ingressi. Nei sistemi più complessi è tuttavia necessario che i valori di uscita dipendano dalla **sequenza temporale degli ingressi**... è quindi necessario mantenere in memoria il comportamento passato del sistema. Tali reti prendono il nome di **reti sequenziali**.

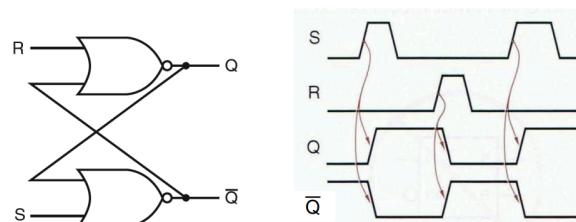
Gli elementi in grado di conservare informazioni sono detti **bistabili** in quanto caratterizzati da due stati stabili, 0 e 1. Il passaggio da uno stato all'altro dipende da un evento esterno in assenza del quale il valore memorizzato viene preservato.

Se l'evento scatenante è il clock si parla di elementi di memoria sincroni, in altro caso si parla di elementi di memoria asincroni.

Una rete sequenziale sincrona utilizza un segnale di sincronizzazione (clock) che definisce gli istanti in cui l'elemento di memoria **aggiorna** il suo contenuto. Una rete sequenziale può essere schematizzata secondo il seguente modello:



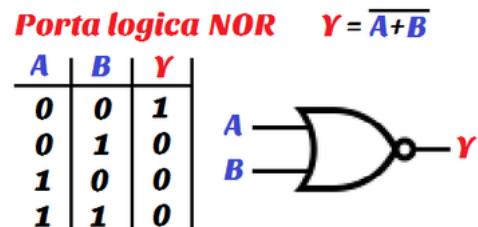
## Latch RS



I segnali S e R prendono il nome di **Set** e **Reset**: Un 1 su Set porta Q ad 1, un 1 su Reset porta Q a 0.

Il circuito è formato da 2 porte NOR:

S	R	Q	Q'
1	0	1	0
0	0	1	0 (after $S = 1, R = 0$ )
0	1	0	1
0	0	0	1 (after $S = 0, R = 1$ )
1	1	0	0 (forbidden)



(b) Function table

Per  $S = R = 0$  lo stato del bistabile rimane invariato:

le porte NOR non dipendono dagli ingressi ma dalle uscite Q e Q'. Tali porte si comportano da invertitori: se per esempio  $Q = 0 \rightarrow Q' = 1$  e viceversa. Tale condizione causa uno stato di mantenimento delle uscite ovvero di memorizzazione dei valori logici precedentemente assunti.

Per  $S = 0, R = 1 \implies Q = 0, Q' = 1$  il bistabile si porta nello stato di **reset**:

La prima porta NOR ricevendo in ingresso un 1, commuta la sua uscita; l'altra porta, a sua volta, avendo in ingresso S=0 e Q=0 assume in uscita il livello logico Q=1. Lo stato di ingresso S=0 ed R=1 determina una situazione di azzeramento dell'uscita Q.

Per  $S = 1, R = 0 \implies Q = 1, Q' = 0$  il bistabile si porta nello stato di **set**:

L'uscita della seconda porta, che riceve in ingresso un 1, diventa o rimane Q=0, di conseguenza l'uscita di P1, avendo entrambi gli ingressi a 0 commuta a livello logico 1. La condizione S=1 ed R=0 che ha come ingresso attivo S (set=imposta) mentre R è inattivo, provoca l'impostazione dello stato logico 1 sull'uscita Q.

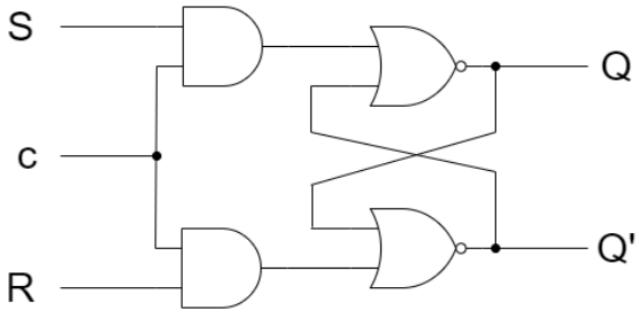
Per  $S = R = 1$

comporta una situazione di ambiguità ed è inutilizzabile in quanto entrambe le uscite sarebbero forzate a 0.

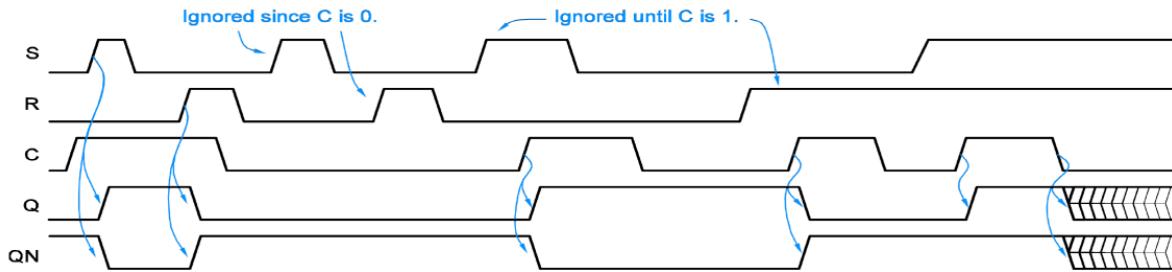
La durata minima del segnale d'ingresso deve essere maggiore o uguale a  $2d$  dove d è il ritardo di porta. Occorre quindi che il segnale in ingresso non cambi per il tempo necessario affinché la rete raggiunga uno stato stabile. Inoltre non dovrebbero essere consentite le transizioni in cui variano entrambi gli ingressi.

### Latch RS sincrono (gated o con abilitazione)

In questo caso è presente il segnale di **clock** che determina quando lo stato del latch può essere modificato. Il segnale cambia solamente quando il clock è 1.



c	S	R	Q	Q'
0	X	X	Q	Q'
1	0	0	Q	Q'
1	1	0	1	0
1	0	1	0	1
1	1	1	?	?



Per  $c = 1, S = 1, R = 0$  il bistabile si porta nello stato di set;

Per  $c = 1, S = 0, R = 1$  il bistabile si porta nello stato di reset

Per  $c = 1, S = 0, R = 0$  lo stato resta invariato

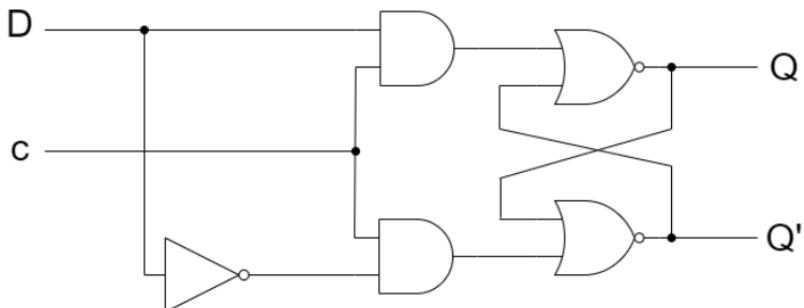
Per  $c = 0, S = x, R = y$  lo stato resta invariato a prescindere dai valori assunti da S e R.

Lo stato del bistabile è quindi preservato durante tutto il periodo di tempo nel quale  $c = 0$ . Diventa sensibile ad ogni variazione degli ingressi quando  $c = 1$ .

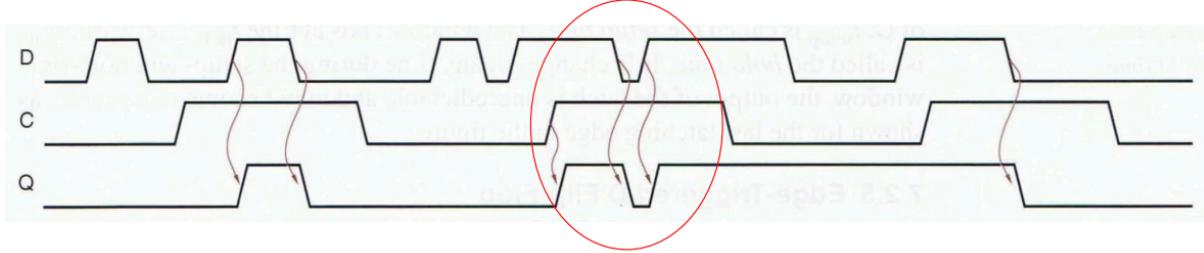
### Latch D sincrono

In questo caso il Latch presenta un solo ingresso oltre al clock. In questa versione si elimina lo stato indefinito  $R = S = 1$ .

La porta NOT fa sì che  $D = S = R'$ .



c	D	Q	Q'
0	X	Q	Q'
1	0	0	1
1	1	1	0



Il problema principale del latch sta nel fatto che, se la rete combinatoria è particolarmente veloce, il contenuto dell'elemento di memoria può essere modificato più volte all'interno di un solo ciclo di clock, rendendo indefinito il comportamento della rete. I flip-flop sono degli elementi bistabili sensibili alla transizione tra livelli di clock, in questo modo l'ingresso è memorizzato in un preciso istante definito dalla transizione del clock da un livello all'altro.

### flip-flop

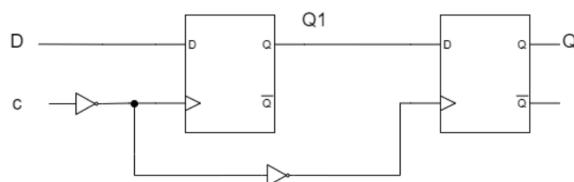
Un **flip-flop** è un elemento che modifica la sua uscita solamente quando il segnale di abilitazione cambia. Esistono due modelli principale: edge-triggered e master-slave.

Nel primo caso l'uscita è modificata durante la variazione detta fronte (**edge**) del clock. L'uscita si può modificare sul fronte di salita o discesa. È importante che l'ingresso D rimanga stabile intorno al fronte del segnale C, per un piccolo intervallo di tempo, altrimenti l'uscita non è affidabile. Ciò significa che, per produrre risultati validi, è necessario un periodo di clock sufficientemente lungo, maggiore rispetto al ritardo legato al percorso critico della rete combinatoria.

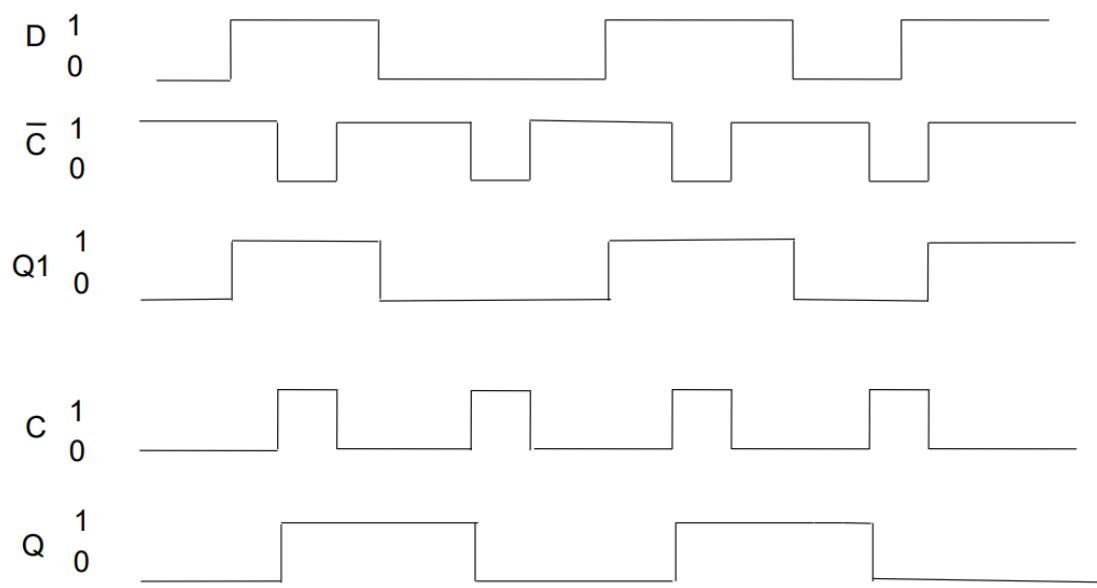
Il **flip-flop master-slave** invece prevede che il valore dell'ingresso venga memorizzato su uno dei fronti e successivamente portato in uscita in corrispondenza dell'altro fronte.

Dato che un singolo flip-flop è in grado di memorizzare un singolo bit, più flip flop vengono organizzati in registri per memorizzare più dati. Ogni flip-flop è sincronizzato da un clock comune.

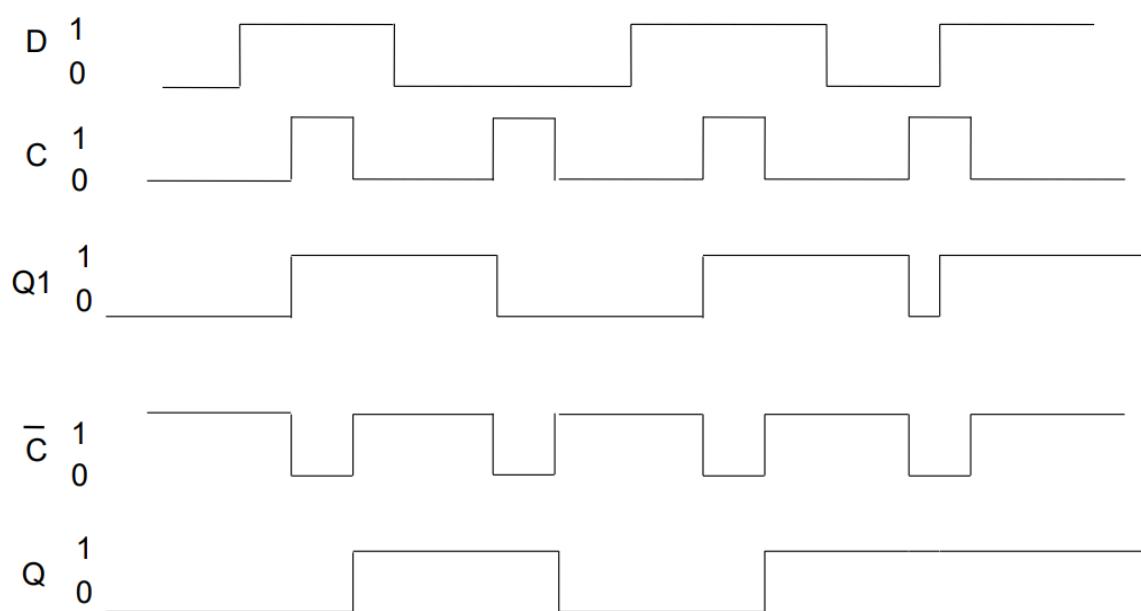
### flip-flop D edge triggered



flip-flop sensibile sul fronte di salita del clock (**flip-flop D raising edge triggered**)



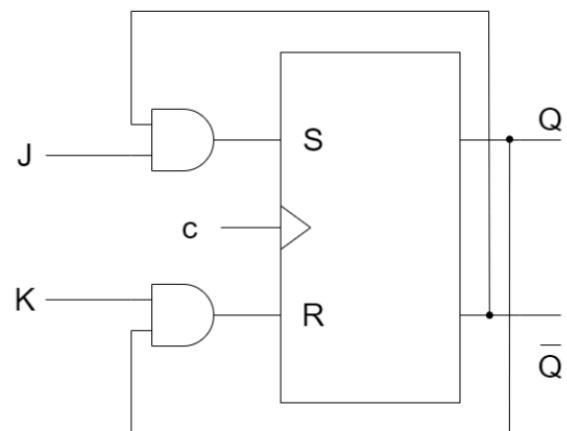
Analogamente il flip-flop D sensibile sul fronte di discesa (**flip-flop D falling edge triggered**)



### Flip-flop JK

J	K	$Q^+$	$\bar{Q}^+$
0	0	Q	$\bar{Q}$
0	1	1	0
1	0	0	1
1	1	$\bar{Q}$	Q

Diversamente dal FF RS, questa volta anche la configurazione di ingresso  $J=1, K=1$  è consentita e l'effetto è quello di invertire le uscite correnti.



## 6. Datapath a ciclo singolo

Con Datapath si intendo un insieme di unità di calcolo necessarie per l'esecuzione delle istruzioni da parte della CPU.