

# Architettura degli Elaboratori

Laurea in Informatica - UniTo - Ordinamento 2023 (2.0)

## Sommario

---

<b>Tipi di calcolatori e loro caratteristiche</b>	<b>4</b>
Legge di Moore	4
<b>Che cosa c'è dietro un programma</b>	<b>4</b>
Componenti di un calcolatore	5
Gerarchia delle memorie	5
<b>Prestazioni</b>	<b>6</b>
Misurare le prestazioni	6
Prestazioni della CPU	6
Misura delle prestazioni associate alle istruzioni	7
<b>Il linguaggio dei calcolatori</b>	<b>7</b>
Operazioni svolte dall'hardware	7
Operandi dell'hardware	8
Operandi allocati in memoria	9
Operandi immediati o costanti	10
Rappresentazione delle istruzioni nel calcolatore	10
Campi delle istruzioni nel RISC-V	11
Operazioni logiche	12
Istruzioni per prendere decisioni	14
<b>Procedure</b>	<b>16</b>
Le fasi di una invocazione di procedura	16
Utilizzo di più registri	17
Allocazione dello spazio nello stack	18
Che cosa succederebbe se si volessero passare più di otto parametri a una procedura?	18
Indirizzamento nei salti	18
<b>Tradurre e avviare un programma</b>	<b>20</b>
Linguaggi Assemblativi	20
Le macro	20
Compilatore	21
Assemblatore	21
Problema delle Forward Reference	21
1° Passo: Creazione della Symbol Table	21
2° Passo: Traduzione del Codice	21
Vantaggi dell'Assemblatore a Due Passi	21
Linker	22
Compiti del Linker	22
Processo di Linking - Esempio (TODO)	22
Diagramma del Processo	22

ELF - File Oggetto nei Sistemi UNIX	23
Loader	23
Binding e Rilocazione Dinamica	23
Momenti per Effettuare il Collegamento	24
Collegamento Statico vs Dinamico	24
Collegamento Statico	24
Collegamento Dinamico	24
Conclusione	24
<b>Circuiti logici digitali di base</b>	<b>25</b>
Livello della Logica Digitale	25
Circuiti Digitali	25
Porte Logiche	25
Circuiti Combinatori	25
Decoder	25
Multiplexer	25
Circuiti Numerici	26
Addizionatori	26
ALU (Arithmetic Logic Unit)	26
ALU a 1 Bit	26
ALU a 64 Bit	26
ALU a 1 bit (sub) - (nor)	26
ALU per slt	27
ALU controllo overflow	27
ALU per beq	27
Clock	27
Controllo delle Operazioni dell'ALU	27
<b>Circuiti sequenziali</b>	<b>28</b>
Dispositivo che memorizza 1 bit? Latch di tipo SR!	28
Latch di tipo SR sincronizzato	28
Latch di tipo D sincronizzato	29
Il problema della “trasparenza”	29
Diagramma temporale del D-latch	29
Flip-flop di tipo D	30
Flip-flop di tipo D sul fronte di discesa	30
Tutti i tipi di Flip-Flop	30
Buffer (non) invertente   Tri-state	30
Registri	31
Blocco di registry (register file)	31
Organizzazione della memoria	31
Chip di memoria	32
Tipi di memoria: RAM	32
Tipi di memoria: ROM	32
Macchine a Stati Finiti	32
Analisi di Reti Sequenziali Sincrone	33

Tabelle di Stato	33
Circuiti di Mealy e di Moore	33
Diagramma di Stato	33
Sintesi di Reti Sequenziali Sincrone	33
<b>Il processore</b>	<b>34</b>
Convenzioni del progetto logico	34
Metodologia di temporizzazione	35
Realizzazione di un'unità di elaborazione	35
Istruzioni in formato R (add, sub, and, or)	36
Istruzioni di caricamento di un registro (load) e di trasferimento alla memoria (store)	36
Istruzioni di salto condizionato (beq)	37
Unità di Controllo	38
Controllo operazione della ALU	38
Codop	39
Riassunto	40
<b>Gerarchia delle memorie</b>	<b>41</b>
Memoria cache	42
Allocazione	42
Come si fa a sapere se un dato è presente nella cache? Se presente, come facciamo a trovarlo?	
42	
Come facciamo a sapere se la parola richiesta si trova nella cache oppure no?	43
Prestazioni di una cache	43
Cache associativa	43
Cache set-associativa	43
Gestione della miss	44
<b>Gestione della scrittura</b>	<b>44</b>
<b>I Bus</b>	<b>45</b>
Connessioni di una CPU	46
Larghezza del bus	47
Bus clocking	47
Bus sincroni:	47
Bus asincroni:	48
Cosa succede se più di un dispositivo richiede l'utilizzo del bus contemporaneamente?	
48	
Altri tipi di cicli di bus	49
Trasferimento a blocchi	49
Read-Modify-Write (test-and-set)	49
Interrupt	49
<b>I/O</b>	<b>50</b>
Tipi di Istruzioni a Livello ISA	50
Comunicazione tra CPU e Moduli di I/O	50
Registri nei Controllori	50
I/O Programmato con Busy Waiting	50
Interrupt	50
Esempio	51
Caratteristiche degli Interrupt	51

Priorità degli Interrupt	51
Trap ed Eccezioni	52
DMA (Direct Memory Access)	52
Eccezioni nel Processore RISC-V	52
Gestione delle Eccezioni	52
Interrupt (Interruzione)	52
Errore	52
Environment Call (ecall)	52
Environment Break (ebreak)	53
Gestione delle Eccezioni	53
Routine di Gestione	53
Salvataggio dello Stato	53
Registri Eccezioni nel RISC-V	54
<b>Riconoscimenti</b>	<b>55</b>

## Tipi di calcolatori e loro caratteristiche

Nonostante calcolatori molto diversi tra loro condividano la stessa tecnologia hardware, nella maggior parte dei casi le soluzioni utilizzate non sono identiche. Infatti queste applicazioni sono caratterizzate da requisiti di progetto differenti che implicano un diverso utilizzo dell'hardware.

A grandi linee, i calcolatori possono essere raggruppati in tre classi ben distinte.

I **personal computer** rappresentano il tipo di calcolatore più conosciuto; essi offrono buone prestazioni a un singolo utente mantenendo il costo limitato; inoltre vengono spesso utilizzati per eseguire software scritto da terze parti.

I **server** sono la forma moderna di quelli che un tempo erano calcolatori di dimensioni decisamente maggiori e, di norma, ad essi si accede solo attraverso la rete. Essi sono orientati all'elaborazione di carichi di lavoro di grosse dimensioni. I server sono realizzati con le stesse tecnologie di un PC, ma offrono una maggiore potenza di calcolo, una maggiore velocità di input/output e una maggiore capacità della memoria.

I **calcolatori embedded** (cioè dedicati) sono i più numerosi e coprono un ampio spettro di applicazioni e prestazioni. I sistemi di calcolo di questo tipo sono progettati per eseguire una singola applicazione o un insieme di applicazioni correlate tra loro; queste applicazioni sono di norma integrate con l'hardware e si presentano all'utente come un sistema monolitico.

Le applicazioni di tipo embedded richiedono spesso prestazioni limitate con vincoli stringenti sul costo e sulla potenza assorbita dal dispositivo.

## Legge di Moore

Una delle costanti dei calcolatori è la rapida evoluzione, descritta principalmente dalla **legge di Moore**, la quale stabilisce che le risorse messe a disposizione dai circuiti integrati vengano duplicate ogni 18-24 mesi.

## Che cosa c'è dietro un programma

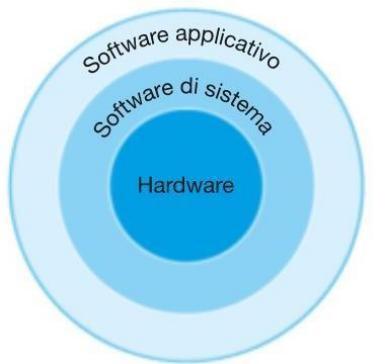
Il calcolatore può eseguire solo istruzioni di basso livello estremamente semplici. Passare da un'applicazione complessa alle semplici istruzioni comprensibili al calcolatore è un processo che coinvolge diversi strati di software, organizzati principalmente in maniera gerarchica.

Nel cerchio più esterno compaiono le applicazioni, mentre i diversi componenti del **software di sistema** sono posizionati nel cerchio intermedio tra l'hardware e le applicazioni software.

Il software di sistema ha diversi componenti, ma sono due quelli essenziali per tutti i calcolatori moderni: il sistema operativo e il compilatore.

Il **sistema operativo** permette di interfacciare i programmi utente con l'**hardware** del calcolatore, fornendo un gran numero di servizi e funzioni di supervisione.

I **compilatori** eseguono la traduzione di un programma scritto in linguaggio ad alto livello in istruzioni eseguibili dall'hardware, il che è una funzione complessa.



## Componenti di un calcolatore

Il **processore (CPU)** è la parte attiva del calcolatore, quella che esegue fedelmente le istruzioni di un programma: è in grado di effettuare somme tra numeri, fare test su essi, inviare segnali per attivare dispositivi di I/O e così via.

Il processore comprende due componenti principali: l'**unità di controllo** e l'**unità di elaborazione dati (datapath)**. L'unità di controllo invia i comandi all'unità di elaborazione dati, alla memoria e ai dispositivi di I/O secondo le istruzioni del programma.

Il datapath provvede a eseguire le operazioni aritmetico-logiche sui dati.

La **memoria** è il luogo dove vengono tenuti i programmi in esecuzione assieme ai loro dati.

**DRAM** e **RAM** sono memorie **ad accesso casuale** e l'accesso richiede lo stesso tempo indipendentemente dalla particolare area di memoria a cui si accede.

La **memoria cache** consiste in una memoria piccola ma veloce che funge da tampone nei confronti della DRAM che invece è più grande e più lenta. La cache è costruita usando una tecnologia di memorie di tipo **SRAM**, ovvero **statica ad accesso casuale**.

L'**ISA (Instruction Set Architecture)** è l'**interfaccia astratta tra hardware e il livello più basso del software del calcolatore**. Comprende tutte le informazioni necessarie per scrivere un programma in linguaggio macchina funzionante in modo corretto, comprese istruzioni, registri, gli accessi alla memoria, I/O etc.

## Gerarchia delle memorie

Una **memoria volatile** è in grado di mantenere i dati solamente se è alimentata. Un esempio di memoria volatile è la DRAM.

Una **memoria non volatile**, invece, conserva i dati anche quando viene a mancare l'alimentazione; viene utilizzata per conservare i dati tra un'esecuzione e l'altra.

La **memoria principale**, detta anche **memoria primaria**, viene utilizzata per contenere i programmi durante la loro esecuzione.

La **memoria di massa**, detta anche **memoria secondaria**, è una memoria non volatile utilizzata per conservare i programmi e i dati tra un'esecuzione e l'altra.

## Prestazioni

- Tempo di risposta:** Detto anche **tempo di esecuzione**, è il tempo totale richiesto da un calcolatore per completare un task; esso comprende gli accessi a disco, gli accessi a memoria, le attività di I/O, il tempo richiesto dal sistema operativo, il tempo di esecuzione della CPU etc.
- Throughput:** Detto anche **larghezza di banda**, rappresenta il numero di programmi completati per unità di tempo.

Per massimizzare le prestazioni vogliamo minimizzare il tempo di esecuzione richiesto da un dato task. Possiamo quindi mettere in relazione le prestazioni con il tempo di esecuzione; per un generico calcolatore X varrà la relazione:

$$\text{Prestazioni}_X = \frac{1}{\text{Tempo di esecuzione}_X}$$

## Misurare le prestazioni

Il calcolatore che esegue un certo lavoro nel tempo minore è il più veloce.

I calcolatori lavorano spesso in condivisione e può accadere che il processore stia lavorando su più programmi contemporaneamente; per questo motivo si distingue tra **tempo assoluto** di esecuzione di un programma e tempo durante il quale il processore ha **effettivamente** lavorato su quel programma.

Il **tempo di esecuzione della CPU** è il tempo effettivamente speso dalla CPU nella computazione richiesta dal programma e non comprende il tempo speso per le operazioni di I/O o nell'esecuzione di altri programmi.

Il tempo di CPU può essere ulteriormente suddiviso in **tempo di CPU utente** (tempo effettivamente speso dalla CPU nella computazione richiesta da un programma) e **tempo di CPU di sistema** (tempo speso dalla CPU per eseguire le funzioni del sistema operativo richieste per l'esecuzione di un programma).

Quasi tutti i calcolatori sono costruiti utilizzando un segnale che sincronizza le varie funzioni implementate nell'hardware; questo segnale è periodico nel tempo e i relativi intervalli di tempo sono i **cicli di clock**. Il **periodo di clock** è il tempo necessario per completare un intero ciclo di clock; la **frequenza di clock** è il suo inverso.

## Prestazioni della CPU

$$\text{Tempo di CPU relativo a un programma} = \frac{\text{Cicli di clock della CPU relativi al programma}}{\text{Periodo di clock}}$$

$$\text{Tempo di CPU relativo a un programma} = \frac{\text{Cicli di clock della CPU relativi a un programma}}{\text{Frequenza di clock}}$$

## Misura delle prestazioni associate alle istruzioni

Il numero di cicli di clock necessari per l'esecuzione di un programma si può scrivere come:

$$\text{Cicli di clock della CPU} = \frac{\text{Numero di istruzioni del programma}}{\text{Numero medio di cicli di clock per istruzione}}$$

**Cicli di clock per istruzione (CPI):** Numero medio di cicli di clock che le diverse istruzioni richiedono per essere completate.

Dato che istruzioni diverse possono richiedere un tempo di esecuzione differente in funzione del compito che svolgono, il CPI è una quantità utile a confrontare due calcolatori diversi che condividono la stessa architettura dell'insieme di istruzioni.

Possiamo quindi scrivere questa equazione fondamentale in funzione del numero di istruzioni, del CPI e del periodo di clock:

$$\text{Tempo di CPU} = \text{Numero di istruzioni} \times \text{CPI} \times \text{Periodo di clock}$$

$$\text{Tempo di CPU} = \frac{\text{Numero di istruzioni} \times \text{CPI}}{\text{Frequenza di clock}}$$

---

## Il linguaggio dei calcolatori

### Operazioni svolte dall'hardware

Qualsiasi calcolatore deve saper eseguire le operazioni aritmetiche.

A differenza di altri linguaggi di programmazione, in un linguaggio Assembler ciascuna linea può contenere al **massimo** una istruzione.

Il numero di operandi per una operazione come la somma è pari a tre: i due numeri da sommare e il riferimento alla locazione in cui memorizzare il risultato.

C/C++

```
add a, b, c // la somma di b e c è posta in a
```

Il fatto di richiedere che tutte le istruzioni abbiano esattamente **tre** operandi è conforme alla filosofia di mantenere **l'hardware** semplice.

**Il primo principio per la progettazione dell'hardware è: la semplicità favorisce la regolarità.**

# Operandi dell'hardware

Gli operandi delle istruzioni aritmetiche del RISC-V devono obbedire ad alcune restrizioni: devono essere scelti tra un numero limitato di locazioni particolari, chiamate *registri*.

## Operandi RISC-V

Nome	Esempio	Commenti
32 registri	x0-x31	Accesso veloce ai dati. Nel RISC-V gli operandi devono essere contenuti nei registri per potere eseguire delle operazioni. Il registro x0 contiene sempre il valore 0
$2^{61}$ parole di memoria	Memoria[0], Memoria[8], ... Memoria[18 446 744 073 709 551 608]	Alla memoria si accede solamente attraverso istruzioni di trasferimento dati. Il RISC-V utilizza l'indirizzamento al byte, perciò due variabili ampie due parole (double word) hanno indirizzi in memoria a distanza 8. La memoria consente di memorizzare strutture dati, vettori, o il contenuto dei registri

I registri rappresentano sia le **primitive** utilizzate nella progettazione dell'hardware sia gli elementi visibili al programmatore.

La dimensione dei registri nell'architettura RISC-V è di **64 bit**; gruppi di 64 bit prendono il nome di **doubleword**, a cui si accede naturalmente in un calcolatore.

Una **word** è invece il numero di bit a cui si accede più naturalmente in un calcolatore ed è costituita da 32 bit.

Una delle differenze più importanti fra le variabili utilizzate nei linguaggi di programmazione e i registri è il numero limitato di questi ultimi; infatti sono esattamente **32** nei calcolatori RISC-V.

La ragione di questa limitazione si trova nel secondo principio fondamentale per la progettazione dell'hardware: **minori sono le dimensioni, maggiore è la velocità**.

Un numero molto elevato di registri potrebbe **aumentare** la durata del ciclo di clock semplicemente perché i segnali elettrici impiegherebbero un tempo maggiore a compiere il percorso assegnato.

x0	zero
x1	Return address (ra)
x2	Stack pointer (sp)
x3	Global pointer (gp)
x4	Thread pointer (tp)
x8	Frame pointer (fp)
x10-x17	Registri usati per il passaggio di parametri nelle procedure e valori di ritorno
x5-x7 , x28-x31	Registri temporanei, non salvati in caso di chiamata
x8-x9, x18-x27	Registri da salvare: il contenuto va preservato se utilizzati dalla procedura chiamata

## Operandi allocati in memoria

Benché il processore possa contenere un numero limitato di dati nei registri, la memoria può contenere miliardi di dati. Di conseguenza le strutture dati (vettori e strutture) vengono allocate in memoria.

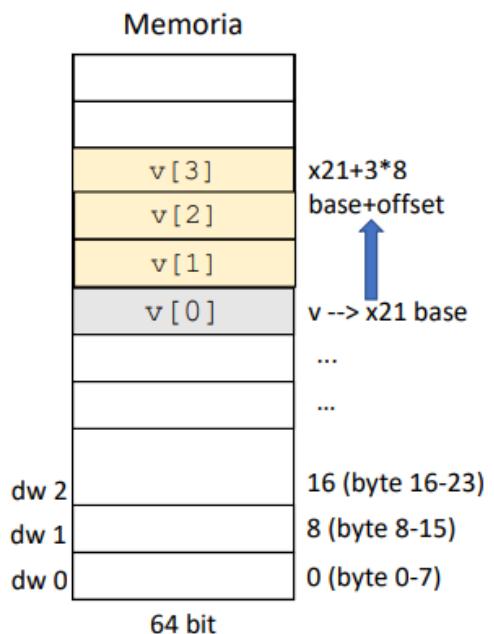
Nel RISC-V le istruzioni aritmetiche richiedono che gli operandi siano memorizzati nei registri; l'assembler RISC-V deve quindi contenere delle **istruzioni di trasferimento dati** che trasferiscono dati fra la memoria e i registri.

Per accedere a una word o double word in memoria, l'istruzione deve fornire l'**indirizzo** di memoria corrispondente.

La memoria può essere vista come un grande vettore monodimensionale, con l'indirizzo che funge da indice e parte a contare da zero.

Dato che il RISC-V utilizza l'indirizzamento della memoria al singolo byte e ogni doubleword contiene 8 byte, gli indirizzi sono multipli di 8.

La figura sopra quindi deve essere corretta tenendo conto di questo **vincolo di allineamento**.



L'istruzione che sposta un dato dalla memoria a un registro è la **load** (carica).

**Esempio:** Sia A un vettore di doubleword contenuto in memoria, voglio trasferire il contenuto di A[8] in un registro:

```
ld x9, 64(x22) // il valore A[8] viene caricato nel  
// registro x9
```

L'indirizzo dell'elemento A[8] è dato dalla somma dell'indirizzo base (x22) e il numero che permette di selezionare l'elemento 8, ovvero  $8*8 = 64$ .

x22 è detto *registro base* e la costante si chiama *offset*.

L'istruzione che invece sposta un dato da un registro alla memoria è la *store*.

Esempio: Voglio memorizzare in A[12] (che sta in memoria) la somma tra l'elemento A[8] e la variabile contenuta in x21.

```
ld x9, 64(x22)      // il valore A[8] viene caricato nel
                      // registro x9
add x9, x21, x9    // il registro x9 assume il valore della
                      // somma
sd x9, 96(x22)     // memorizza la somma in A[12]
```

L'indirizzo d dell'elemento A[12] è dato dalla somma dell'indirizzo base (x22) e il numero che permette di selezionare l'elemento 12, ovvero  $8 \times 12 = 96$ .

Come per la load, x22 è detto *registro base* e la costante si chiama *offset*, esistono anche:

- lh, lhu (half-word)
- lw, lwu (word)
- ld, ldu (double-word)
- lb, lbu (byte)

## Operandi immediati o costanti

Spesso i programmi utilizzano all'interno di una operazione una costante, per esempio per incrementare l'indice di un contatore in modo da puntare all'elemento successivo di un vettore.

In più della metà delle operazioni aritmetiche RISC-V uno degli operandi è una costante.

La versione dell'operazione di somma in cui un operando è una costante è chiamata addi (*add immediate*).

```
addi x22, x22, 4          // x22 = x22 + 4
```

Le operazioni su costanti sono molto frequenti. Inserendo le costanti all'interno delle istruzioni aritmetiche, le operazioni risultano molto più veloci e richiedono meno energia rispetto al caso in cui le costanti siano caricate in memoria.

La costante zero ha il ruolo di semplificare l'insieme delle istruzioni, consentendo di realizzare delle utili varianti.

Per esempio, si può **negare** il contenuto di un registro utilizzando l'operazione sub con zero come primo operando. Per questo motivo, i RISC-V dedicano il registro x0 a contenere il valore prefissato di zero.

## Rappresentazione delle istruzioni nel calcolatore

Anche le istruzioni nel calcolatore sono memorizzate come una sequenza di segnali elettrici e vengono rappresentate con stringhe di bit.

L'istruzione viene scomposta in campi di numeri binari secondo quello che viene chiamato **formato dell'istruzione**.

**L'istruzione RISC-V richiede esattamente 32 bit, una word.**

Ricordando che **linguaggio macchina** è la rappresentazione binaria utilizzata per la comunicazione all'interno dei calcolatori, una sequenza di istruzioni in linguaggio macchina viene definita **codice macchina**.

Per evitare la lettura e la scrittura di lunghe stringhe in binario, si ricorre all'uso della numerazione **esadecimale** (numeri in base 16), che può essere convertita facilmente in binario.

## Campi delle istruzioni nel RISC-V

Ai diversi campi delle istruzioni viene associato un nome (Registro di tipo R):

funz7	rs2	rs1	funz3	rd	codop
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit

Con il seguente significato:

*codop*: **codice operativo** che specifica operazione e formato dell'istruzione stessa;

*rd*: registro destinazione: riceve il risultato dell'operazione;

*funz3*: un codice operativo aggiuntivo;

*rs1*: registro contenente il primo **operando** sorgente;

*rs2*: registro contenente il secondo operando sorgente;

*funz7*: un codice operativo aggiuntivo.

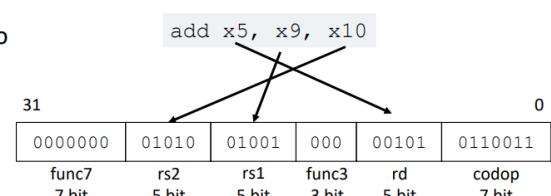
Può nascere un problema quando un'istruzione richiede campi di dimensioni maggiori rispetto a quelle sopra specificate, come nel caso della load, che richiede di specificare due registri e una costante: se la costante venisse inserita in un campo da 5 bit, non potrebbe superare il valore di  $2^5 - 1$ , cioè 31.

Nasce un conflitto fra il desiderio di mantenere la stessa lunghezza per tutte le istruzioni e quello di avere un unico formato. Introduciamo quindi il terzo principio fondamentale della progettazione dell'hardware: **un buon progetto richiede buoni compromessi**.

Nel RISC-V si è deciso di mantenere uguale la lunghezza di tutte le istruzioni, predisponendo formati diversi per tipi di istruzioni diverse.

Il formato descritto sopra è chiamato di *tipo R* (R sta per registro).

Esempio



Un altro tipo di formato è detto *tipo I* (I sta per immediato) e viene utilizzato dalle operazioni in cui un operando è una costante:

immediato	rs1	funz3	rd	codop
12 bit	5 bit	3 bit	5 bit	7 bit

### Esempi

ld x9, 176(x22)				
31	immediato 12 bit	rs1 5 bit	funz3 3 bit	0
000010110000	10110	011	01001	0000011

addi x7, x5, -6				
31	immediato 12 bit	rs1 5 bit	funz3 3 bit	0
111111111010	00101	000	00111	0010011

Questa soluzione consente di mantenere i campi *rs1* e *rs2* nella stessa posizione in tutti i formati di istruzioni; inoltre i campi *codop* e *func3* hanno la **stessa dimensione** e si trovano nella **stessa posizione** in tutte le istruzioni.

Si possono distinguere i due formati in base al codice operativo: a ciascun formato è assegnato un insieme di valori del campo *codop*, in modo tale che l'hardware sappia esattamente come deve trattare i rimanenti bit dell'istruzione.

Formato di tipo S:

31	0
Immediato [11:5]	codop
7 bit	7 bit

rs2	rs1	funz3	Immediato[4:0]
5 bit	5 bit	3 bit	5 bit

## Operazioni logiche

Sono operazioni utili a operare su gruppi di bit o su singoli bit di una word.

Operazioni logiche	Istruzioni RISC-V
Shift a sinistra	sll, slli
Shift a destra	srl, srli
Shift a destra aritmetico	sra, srai
AND bit a bit	and, andi
OR bit a bit	or, ori
XOR bit a bit	xor, xori
NOT bit a bit	xori

La prima tipologia di queste operazioni è lo **shift** (scorrimento) e consiste nello spostare tutti i bit di una word a sinistra o a destra, riempiendo i bit vuoti con degli zeri.

Esempio supponiamo che il registro x19 contenga:

```
000000000 000000000 000000000 000000000 000000000 000000000 000000000 00001001due = 9dec
```

ed eseguiamo l'istruzione di shift a sinistra di 4; il numero ottenuto sarà:

```
000000000 000000000 000000000 000000000 000000000 000000000 000000000 10010000due = 144dec
```

Questa operazione è eseguibile tramite l'istruzione **slli** (*shift left logical immediate*):  
**slli x11, x19, 4 // reg x11 = x19 << 4 bit**

funz6	immediato	rs1	funz3	rd	codop
0	4	19	1	11	19

L'operazione duale a quella riportata sopra è la **srl** (*shift right logical immediate*).

Queste operazioni di shift utilizzano il formato **di tipo I**.

Dato che non serve far scorrere i bit di un registro formato da 64 bit per più di 63 posizioni, solamente i 6 bit meno significativi del campo immediato vengono effettivamente utilizzati.

I rimanenti 6 bit vengono utilizzati come un campo aggiuntivo di codice operativo.

L'operazione di shift logico fornisce una ulteriore funzionalità: lo scorrimento di un numero a sinistra di  $i$  cifre produce lo stesso risultato di una moltiplicazione per  $2^i$ .

Allo stesso modo, vediamo un terzo tipo di operazione di shift: lo *scorrimento a destra aritmetico srai*, simile alla **srl** tranne che invece di riempire con degli zeri, i bit che si liberano vengono riempiti copiando il bit del segno.

Il RISC-V fornisce anche una variante per ognuna di queste operazioni che, anzi che utilizzare un immediato, prendono il numero con cui fare lo scorrimento da un registro: **sll**, **srl** e **sra**.

Un'altra operazione logica, che permette di **isolare** i campi di una word, è l'**AND**.

L'operazione di AND è un'operazione logica bit a bit su due operandi, che restituisce 1 se entrambi gli operandi sono uguali a 1, 0 altrimenti.

**Esempio** supponiamo che il registro x11 contenga:

```
00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000due
```

e il registro x10 contenga:

```
00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000due
```

il valore contenuto nel registro x9 sarebbe:

```
00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000due
```

Questa operazione si scrive con l'istruzione:

```
and x9, x10, x11 // x9 = x10 & x11
```

L'operazione di AND bit a bit può essere usata per **forzare a 0** i bit di una word fornendo in input all'AND una parola contenente zeri in quelle posizioni (questa parola viene chiamata *maschera*).

**Esempio:**

x6	00100100 00010101 00001011 10100110	Sorgente
and x5, x6, x7	x7 00000100 00000110 00000010 00010010	Maschera
	x5 00000100 00000100 00000010 00000010	Risultato

L'**OR** è un'operazione bit a bit su due operandi, che restituisce 1 se *almeno uno dei due operandi* è uguale a 1.

Esempio supponiamo di avere gli stessi registri visti sopra:

il risultato dell'operazione or x9, x10, x11 // x9 = x10 | x11

sarà:

00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000<sup>Q76</sup>

L'operazione di OR bit a bit può essere usata per **forzare a 1** i bit di una word fornendo la maschera adatta.

Esempio:

or x5, x6, x7	x6 00100100 00010101 00001011 10100110	Sorgente
	x7 00000100 00000110 01000010 00010010	Maschera
	x5 00100100 00010111 01001011 10110110	Risultato

L'ultima operazione logica è la **negazione**: l'operazione di **NOT** prende un operando e inverte i bit (tutti gli 1 diventano 0 e viceversa).

Per mantenere il formato **dell'operazione a tre operandi**, nel RISC-V abbiamo l'operazione **XOR** (OR esclusivo) invece della negazione semplice.

Dato che XOR produce 1 quando i valori dei due operandi sono diversi, si può ottenere NOT tramite xor di un numero con 111...111.

## Istruzioni per prendere decisioni

Il calcolatore è capace di prendere decisioni: in base ai dati in ingresso e ai valori calcolati durante l'elaborazione, possono essere eseguite istruzioni diverse.

Il RISC-V è in grado di implementare un processo decisionale simile al costrutto *if* tramite istruzioni di **salto condizionato** (*conditional branches*):

beq rs1, rs2, L1

e significa: vai all'istruzione etichettata L1 se il valore contenuto in rs1 corrisponde a quello contenuto in rs2;

beq = *branch if equal* (salta se uguale).

Analogamente, abbiamo l'istruzione bne, che significa *branch if not equal* (salta se non uguale):

bne rs1, rs2, L1

e significa: vai all'istruzione etichettata L1 se il valore contenuto in rs1 *non* corrisponde a quello contenuto in rs2.

### Esempio di costrutto if:

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

Linguaggio C

$i \rightarrow x19$   
 $g \rightarrow x20$   
 $h \rightarrow x21$   
 $j \rightarrow x23$

La scelta di test per not equal è più conveniente in questo caso

```
bne x22,x23,ELSE
add x19,x20,x21
beq x0,x0,ENDIF
```

ELSE: sub x19,x20,x21  
ENDIF:

RISC-V assembler

N.B.

L'assembler evita al compilatore o al programmatore il compito di calcolare gli indirizzi dei salti.

L'insieme dei confronti possibili prevede, oltre l'uguaglianza e disuguaglianza viste sopra, anche:  
 $<$ ,  $\leq$ ,  $>$ ,  $\geq$ .

La comparazione di stringhe di bit deve prevedere sia i numeri dotati di segno sia quelli senza. Questi controlli possono essere effettuati tramite:

- salta se minore: blt (*branch if less than*)
- salta se minore o uguale: ble (*branch if less than or equal*)
- salta se maggiore: bgt (*branch if greater than*)
- salta se maggiore o uguale: bge (*branch if greater than or equal*)
- salta se minore di (senza segno): bltu
- salta se maggiore o uguale di (senza segno): bgeu

Le stesse istruzioni assembler possono essere utilizzate anche per l'implementazione di cicli.

### Esempio di ciclo for:

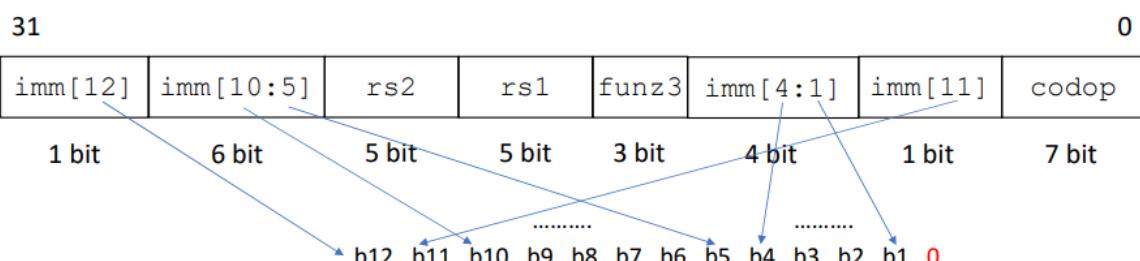
```
for (i=0;i<100;i++)
{
    ...
}
```

$i \rightarrow x19$

```
FOR: add x19,x0,x0
      addi x20,x0,100
      bge x19,x20,ENDFOR
      ...
      addi x19,x19,1
      beq x0,x0,FOR
ENDFOR:
```

I salti **condizionati** utilizzano il formato di tipo **SB**, questo formato può rappresentare indirizzi di salto da -4096 a 4094, in multipli di due.

**ATTENZIONE:** I bit del campo immediato considerando il bit meno significativo (b0) impostato sempre a zero fa sì che abbiamo 13 bit e non più 12 (-2048 a +2046)



Il salto, ovviamente, implica l'utilizzo del PC.

PC = PC + Immediato

In questo caso sono 13 bit di immediato, quindi è possibile arrivare a -4096.

Le istruzioni su RISC-V sono 32Bit, però questa limitazione del solo bit meno significativo, ti permette di saltare anche a 16 bit, noi però lavoriamo con (b1 = 0 e b0 = 0)

## Procedure

Una **procedura** è un sottoprogramma utilizzato in modo da rendere l'intero programma più comprensibile e permettere il riutilizzo del codice.

Le procedure consentono ai programmatori di concentrarsi su una parte del problema alla volta; l'interfaccia fra la procedura e il resto del programma e dei dati è costituita dai *parametri*, i quali permettono di passare dei valori alla procedura e di restituire i risultati al programma chiamante.

Per eseguire una procedura, un programma deve eseguire questi sei passi:

Il Chiamante deve:

1. mettere i **parametri** in un luogo accessibile alla procedura;
2. trasferire il controllo ad essa;

Il Chiamato deve:

3. acquisire le risorse necessarie per la sua esecuzione;
4. eseguire il compito richiesto;
5. mettere il risultato in un luogo accessibile al programma chiamante;
6. restituire il controllo al punto di origine, dato che la stessa procedura può essere chiamata in diversi punti del programma.

Il software RISC-V per le chiamate a procedura utilizza i registri secondo queste convenzioni:

- **x10-x17 (a0-a7)**: registri argomento per il passaggio dei parametri o la restituzione dei valori calcolati;
- **x1 (ra)**: registro contenente l'indirizzo di ritorno per tornare al punto di origine.

L'istruzione per passare alla procedura è **jal** (**jump and link**), la quale esegue un salto all'indirizzo della procedura e contemporaneamente salva nel registro **ra** l'indirizzo dell'istruzione successiva, detto **indirizzo di ritorno**:

**jal ra, EtichettaProcedura**

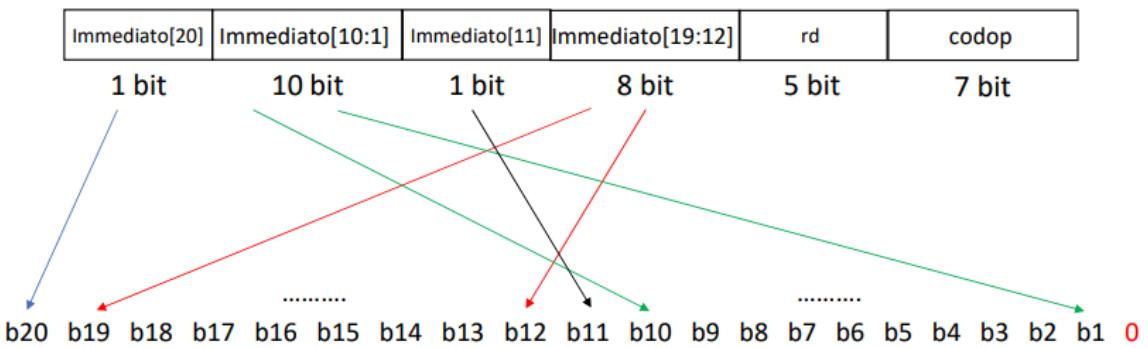
L'indirizzo di ritorno è necessario, perché la stessa procedura può essere chiamata da diversi punti del programma.

### Le fasi di una invocazione di procedura

1. Pre-Chiamata
  - a. Preparazione dei dati nei registri a0-a7
  - b. Eventuale salvataggio di dati
2. Invocazione della procedura (jal)
3. Prologo del chiamato
  - a. Eventuale salvataggio nello stack di registri
4. Corpo della procedura
5. Epilogo lato chiamato
  - a. Salvare i dati in a0-a1
  - b. I registri devono essere ripristinati
6. Ritorno al chiamante
  - a. ret
7. Post-chiamata

Viene introdotto un nuovo tipo: J

31



## Utilizzo di più registri

Supponiamo che il compilatore abbia bisogno, all'interno della procedura, di un numero maggiore di registri rispetto agli 8 elencati sopra.

Il contenuto dei registri utilizzati dal chiamante deve essere ripristinato con il valore *precedente* alla chiamata. Per questo motivo è necessario copiare il contenuto dei registri in memoria, più precisamente nello **stack**, cioè una coda di tipo **LIFO** (*last-in-first-out*).

Lo stack ha bisogno di un puntatore all'indirizzo dell'ultimo dato introdotto per indicare il punto in cui la procedura successiva può salvare il contenuto dei registri e da dove poi possa recuperarli per ripristinarli.

Nel RISC-V il puntatore allo stack (**stack pointer**) è il registro x2 (sp).

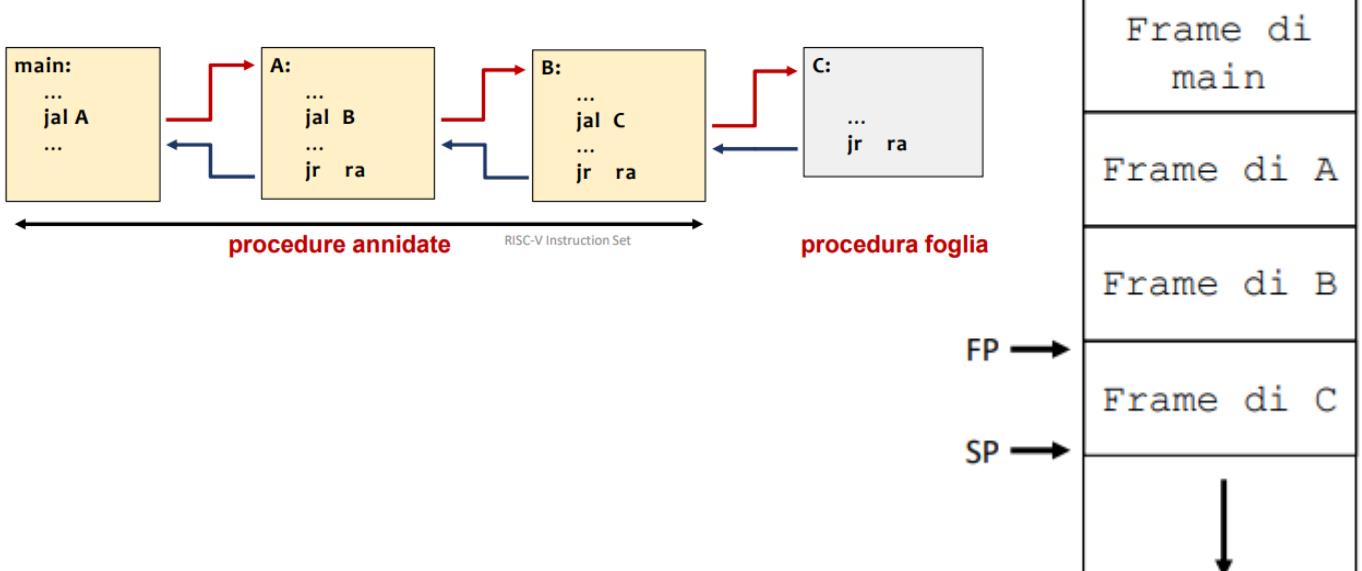
Lo stack pointer viene incrementato o decrementato di una **doubleword** ogni volta che si toglie (**pop**) o si inserisce (**push**) il contenuto di un registro.

Lo stack "cresce" a partire da indirizzi di memoria alti verso indirizzi di memoria bassi: quando si inseriscono contenuti nello stack, il valore dello stack pointer diminuisce; quando i dati vengono estratti, lo stack pointer aumenta, riducendo la dimensione dello stack.

Per evitare di salvare e ripristinare registri il cui valore non verrà mai utilizzato, si suddividono 19 dei registri in due gruppi:

- x5-x7 (t0-t2) e x28-x31 (t3-t6): registri temporanei, che non sono salvati in caso di chiamata di una procedura;
- x8-x9 (fp-s1) e x18-x27 (s2-s11): registri da salvare il cui contenuto deve essere preservato in caso di chiamata a procedura.

**Lo stack pointer contiene l'indirizzo dell'ultima cella di memoria occupata nello stack**



## Allocazione dello spazio nello stack

Lo stack può essere utilizzato anche per memorizzare le variabili locali della procedura che non trovano spazio nei registri. Il segmento dello stack che contiene i registri salvati da una procedura e le variabili locali prende il nome di **record di attivazione**, o **frame della procedura**.

Allocazione dello stack prima, durante e dopo la chiamata della procedura.



Il **frame pointer** (fp) è il valore che individua la posizione dei registri salvati e delle variabili locali di una data procedura.

Lo stack pointer dovrebbe cambiare durante l'esecuzione di una procedura; in questo caso il riferimento alle variabili locali in memoria potrebbe assumere offset diversi a seconda della loro posizione nella procedura. L'utilizzo del frame pointer è vantaggioso proprio perché fa sì che i riferimenti alle variabili in stack di una procedura mantengano lo stesso offset.

Che cosa succederebbe se si volessero passare più di otto parametri a una procedura?

Per convenzione i parametri aggiuntivi vengono messi nello stack al di sopra dell'indirizzo puntato dal frame pointer: la procedura si aspetterà i primi otto parametri nei registri appositi e i restanti nell'area di stack, indirizzabili attraverso fp.

## Indirizzamento nei salti

Le istruzioni di salto condizionato RISC-V utilizzano il formato di *tipo-SB*.

Questo formato può rappresentare indirizzi di salto da -4096 a 4094 **in multipli di 2**: è possibile saltare solo a indirizzi pari.

Il formato di tipo SB consiste in 7 bit di codice operativo, 3 bit di codice funzione, due registri operandi su 5 bit e un campo immediato di indirizzo. Quest'ultimo è implementato con una codifica insolita, che semplifica l'elaborazione da parte della CPU ma complica l'assembler.

L'istruzione di salto **incondizionato jump-and-link (jal)** è l'unica che utilizza il formato di *tipo J*, che consiste in un codice operativo di 7 bit, un registro operando di destinazione su 5 bit e un indirizzo **immediato su 20 bit**. L'indirizzo dell'istruzione successiva viene scritto nel campo rd.

Come per il formato di tipo SB, l'operando che contiene l'indirizzo in questo formato utilizza una codifica insolita e non può codificare gli **indirizzi dispari**.

Se gli indirizzi del programma trovassero posto in questo campo a 20 bit, risulterebbe che nessun programma potrebbe avere una dimensione superiore a  $2^{20}$ , troppo **piccola** per essere utilizzata nelle applicazioni reali. Una valida alternativa consiste nello specificare un registro il cui contenuto deve essere sommato all'indirizzo del salto; l'istruzione di salto dovrebbe quindi effettuare il seguente calcolo:

Program counter = Registro + Spiazzamento del salto

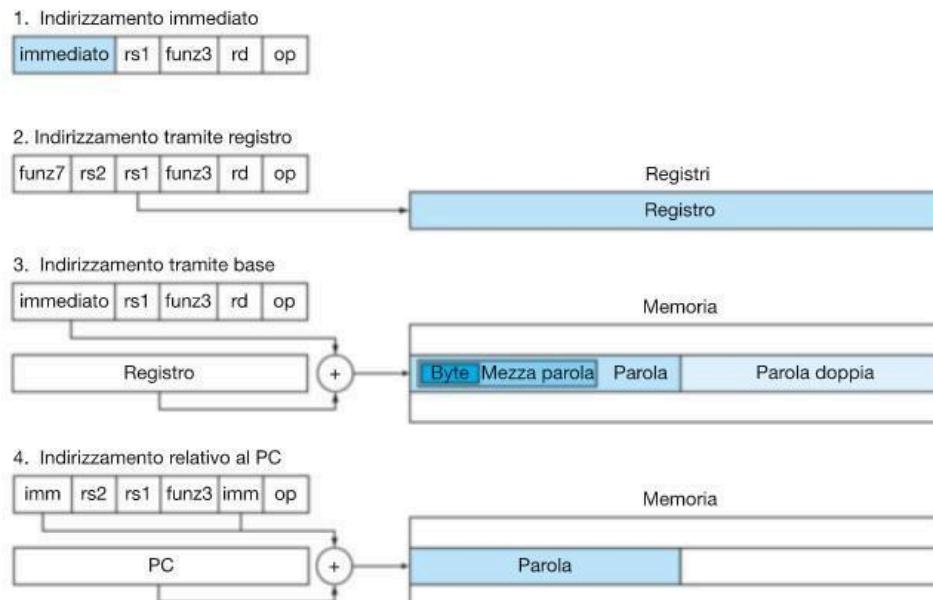
Questa somma consentirebbe al programma di indirizzare  $2^{64}$  posizioni pur continuando a utilizzare i salti condizionati, risolvendo il problema della dimensione dell'indirizzo di salto.

Il metodo di indirizzamento utilizzato si chiama **indirizzamento relativo al program counter** (*PC-relative addressing*). Grazie a questo metodo, il RISC-V consente di effettuare salti molto lunghi a uno qualsiasi tra  $2^{32}$  indirizzi utilizzando una sequenza di due istruzioni: lui scrive i bit da 12 a 31 in un registro temporaneo e *jalr* somma i 12 bit meno significativi all'indirizzo ottenuto con la somma.

Dato che le istruzioni RISC-V sono ampie 4 byte, le istruzioni di salto sono state progettate per ampliare il loro spazio di indirizzamento definendo l'indirizzo relativo al PC in termini di numero di *word* tra l'istruzione corrente di salto e l'istruzione di destinazione del salto, invece che in termini di numero di byte. Tuttavia gli architetti RISC-V hanno voluto supportare la possibilità che le istruzioni siano ampie 2 byte, per cui lo spiazzamento viene definito nelle istruzioni di salto in termini di *half word* che intercorrono tra l'indirizzo dell'istruzione corrente e quello di destinazione del salto.

Quindi il campo di indirizzi di 20 bit nell'istruzione *jal* può codificare una distanza di  $\pm 2^{19}$  half word a partire dal valore attuale del PC.

Analogamente, anche il campo di 12 bit delle istruzioni di salto condizionato è espresso anch'esso in termini di half word; questo vuol dire che rappresenta un indirizzo di 13 bit in termini di byte.



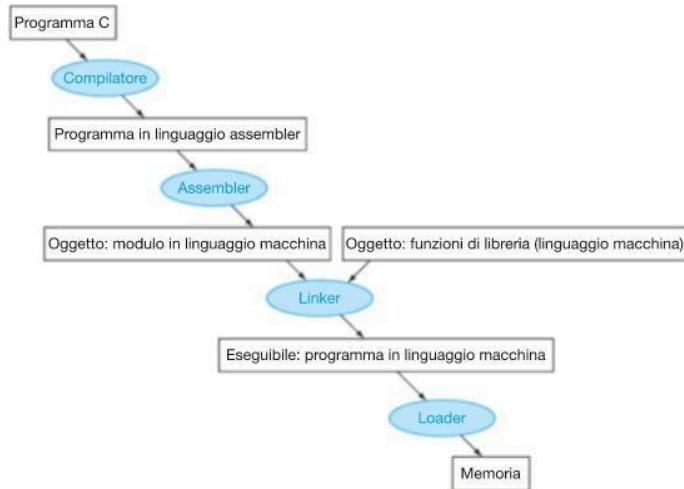
**Figura 2.17 Illustrazione delle quattro modalità di indirizzamento del RISC-V.** Gli operandi sono evidenziati in blu. L'operando della modalità 3 si trova in memoria, mentre quello della modalità 2 si trova in un registro. Si noti che le varianti delle istruzioni di load e store possono accedere al byte, alla mezza parola, alla parola o alla parola doppia. Nella modalità 1, l'operando è contenuto nell'istruzione stessa. La modalità 4 viene utilizzata per indirizzare le istruzioni in memoria, aggiungendo un indirizzo ampio al PC. Si noti che un'operazione può utilizzare diverse modalità di indirizzamento: la somma, per esempio, può avere sia un operando immediato (*addi*) sia tutti gli operandi nei registri (*add*).

Nome (dimensione del campo)	Campi						Commenti
	7 bit	5 bit	5 bit	3 bit	5 bit	7 bit	
Tipo R	funz7	rs2	rs1	funz3	rd	codop	Istruzioni aritmetiche
Tipo I	Immediato[11:0]		rs1	funz3	rd	codop	Istruzioni di caricamento dalla memoria e aritmetica con costanti
Tipo S	immed[11:5]	rs2	rs1	funz3	immed[4:0]	codop	Istruzioni di trasferimento alla memoria (store)
Tipo SB	immed[12, 10:5]	rs2	rs1	funz3	immed[4:1,11]	codop	Istruzioni di salto condizionato
Tipo UJ	immediato[20, 10:1, 11, 19:12]				rd	codop	Istruzioni di salto incondizionato
Tipo U	immediato[31:12]				rd	codop	Formato caricamento stringhe di bit più significativi

## Tradurre e avviare un programma

Un programma scritto in linguaggio ad alto livello viene prima di tutto compilato in linguaggio assembler e poi assemblato per ottenere un modulo oggetto in linguaggio macchina. Il linker unisce uno o più moduli tra loro e con le procedure contenute nelle librerie, e risolve tutti i riferimenti incrociati. Il loader, infine, carica il codice macchina nella opportuna area di memoria, in modo che possa essere eseguito dal processore.

Per accelerare il processo, alcuni passi possono essere saltati o eseguiti in parallelo.



## Linguaggi Assemblativi

Quando si parla di Linguaggio Assemblativo si intende un linguaggio le cui istruzioni sono ottenute dalle istruzioni ISA sostituendo i codici binari con codici mnemonici, il linguaggio assemblativo è quindi molto vicino al linguaggio macchina: c'è sostanzialmente una corrispondenza uno-uno tra le istruzioni ISA e le istruzioni del linguaggio assemblativo.

In realtà, il linguaggio assemblativo rende più facile la vita del programmatore, permettendo l'uso di:

- Etichette (Nomi di variabili, indirizzi)
- Primitive per allocazione in memoria di variabili
- Costanti (#Define)
- Definizioni di **macro**

Per passare dal programma scritto in linguaggio **assemblativo** al programma eseguibile in linguaggio macchina (ISA) si utilizza un programma traduttore detto **assemblatore** (assembler) che traduce i codici mnemonici nei codici numerici corrispondenti alle istruzioni ISA.

Tra i vantaggi del Linguaggio assemblativo abbiamo anche la possibilità di usare le Pseudo-istruzioni, per esempio in RISC-V:

- mv rd rs
- beqz rs, offset
- jal offset
- etc.

```
# swap macro
.macro swap reg1, reg2, reg3 ← parametri
    add \reg3, \reg2, zero
    add \reg2, \reg1, zero
    add \reg1, \reg3, zero
.endm
```

### Le macro

Una definizione di macro è un modo per assegnare un nome ad una sequenza di istruzioni.

Per la definizione di una macro occorre:

1. Un **header** della macro che indica il nome della macro da definire
2. Il **testo** che comprende il corpo della macro
3. Una **"Assembly Directive"** che indica la fine della definizione

## Compilatore

Trasforma il programma C (o qualsiasi altro di alto livello) in un *programma in linguaggio assembler*, cioè in una forma simbolica di ciò che il calcolatore è in grado di comprendere.

Il programma Java è eseguito da un interprete (Java Virtual Machine)

La JVM può invocare il compilatore Just In Time (JIT), che compila i metodi del linguaggio Java nel linguaggio macchina del calcolatore sul quale è in esecuzione.

## Assemblatore

L'assemblatore **traduce** un programma scritto in linguaggio assemblativo nel corrispettivo programma in linguaggio macchina eseguibile.

L'assemblatore legge tutte le istruzioni del programma assemblativo, ne traduce in linguaggio macchina i codici operativi, i dati e le label, controllandone la correttezza sintattica, e restituisce in output il file "oggetto".

Livello implementativo tramite compilazione e **non** interpretazione

### Problema delle Forward Reference

Nel linguaggio assembly, può accadere che un'istruzione faccia riferimento a un'etichetta (o simbolo) che non è ancora stata definita nel punto in cui l'istruzione appare. Questo è noto come "**forward reference**".

Per esempio, una porzione di codice potrebbe chiamare una funzione o fare un salto a un'etichetta che sarà definita solo più avanti nel codice. Il problema delle forward reference è che l'assemblatore deve conoscere l'indirizzo di queste etichette per generare il codice macchina corretto.

### 1° Passo: Creazione della Symbol Table

Nel primo passo, l'assemblatore scorre l'intero programma per individuare tutti i nomi (etichette) che compaiono come riferimento simbolico di dati o di istruzioni. Durante questo passo, l'assemblatore non traduce ancora le istruzioni in codice macchina, ma si limita a raccogliere informazioni sulle etichette e i loro indirizzi.

- **Individuazione delle etichette:** L'assemblatore esamina il codice sorgente e registra tutte le etichette utilizzate nel programma.
- **Creazione della Symbol Table:** L'assemblatore costruisce una tabella dei simboli (Symbol Table) che associa ogni etichetta alla sua posizione relativa all'interno del programma. Questa tabella include informazioni come il nome dell'etichetta e il suo indirizzo (spesso indicato come offset relativo all'inizio del programma).
- Traduce tutte le **pseudo-istruzioni**. (Da verificare, discordante tra slide)

### 2° Passo: Traduzione del Codice

Nel secondo passo, l'assemblatore utilizza la Symbol Table costruita nel primo passo per tradurre i codici mnemonici delle istruzioni, degli operandi e delle etichette in codice macchina.

- **Consultazione della Symbol Table:** Durante la traduzione, l'assemblatore consulta la Symbol Table per risolvere tutti i riferimenti simbolici. Se un'istruzione fa riferimento a un'etichetta, l'assemblatore utilizza l'indirizzo memorizzato nella Symbol Table per generare il codice macchina corretto.
- **Traduzione dei codici mnemonici:** L'assemblatore converte i mnemonici delle istruzioni (come **MOV**, **ADD**, **JMP**) nei corrispondenti codici operativi in linguaggio macchina.
- **Traduzione degli operandi:** Gli operandi delle istruzioni vengono tradotti nei loro valori binari. Se un operando è un'etichetta, il suo valore viene risolto utilizzando la Symbol Table.

## Vantaggi dell'Assemblatore a Due Passi

- **Risoluzione delle forward reference:** Poiché l'assemblatore ha già raccolto tutte le informazioni sulle etichette nel primo passo, può risolvere qualsiasi forward reference durante il secondo passo.
- **Maggiore flessibilità:** Gli sviluppatori possono scrivere il codice senza preoccuparsi dell'ordine delle definizioni delle etichette, rendendo il processo di codifica più naturale e fluido.
- **Affidabilità:** La presenza di due passaggi distinti permette di separare la fase di analisi (costruzione della Symbol Table) dalla fase di traduzione, riducendo la possibilità di errori.

In sintesi, l'assemblatore a due passi è un metodo robusto per gestire la traduzione del codice assembly in codice macchina, risolvendo efficacemente il problema delle forward reference tramite una fase preliminare di analisi e costruzione della Symbol Table, seguita dalla traduzione vera e propria del codice.

Inoltre, gli assemblatori accettano numeri espressi in basi diverse (oltre al binario, abbiamo visto che è valida la notazione decimale, quella ottale e quella esadecimale), che poi verranno convertite in sequenza di bit.

## Linker

Per evitare che la modifica anche di una sola linea di codice richieda di ricompilare e riassemblare l'intero programma, con un conseguente spreco di risorse computazionali, è necessario un sistema che permetta di compilare e assemblare ciascuna procedura indipendentemente dalle altre. In tal modo, ciascuna modifica di una linea di codice rende necessario ricompilare e riassemblare *solo* la procedura a cui la linea di codice appartiene. Per fare ciò è necessario un nuovo programma di sistema, il **link editor** o **linker**; esso prende *tutti* i programmi (le procedure) in codice macchina che sono stati assemblati indipendentemente e li **unisce**.

Il motivo per cui il linker è particolarmente utile è che risulta molto più veloce correggere il codice piuttosto che ricompilarlo e riassemblarlo di nuovo.

### Compiti del Linker

Il linker esegue una serie di operazioni per combinare i moduli oggetto:

1. **Costruzione della Tabella dei Moduli:** Il linker costruisce una tabella che contiene tutti i moduli oggetto e le loro lunghezze.
  - a. Le loro lunghezze alla fine verranno sommate ed usate per la lunghezza finale dell'eseguibile
2. **Assegnazione degli Indirizzi:** Il linker assegna un indirizzo di inizio a ogni modulo oggetto. Questo permette di creare un unico spazio di indirizzamento continuo.
3. **Aggiunta delle Relocation Constants:** Il linker individua tutte le istruzioni che accedono alla memoria e aggiunge a ciascun indirizzo una "relocation constant", che corrisponde all'indirizzo di partenza del suo modulo. Questo processo è noto come "relocation".

**Aggiornamento dei Riferimenti ai Moduli:** Il linker trova tutte le istruzioni che fanno riferimento ad altri moduli e le aggiorna con l'indirizzo corretto.

## Processo di Linking - Esempio (TODO)

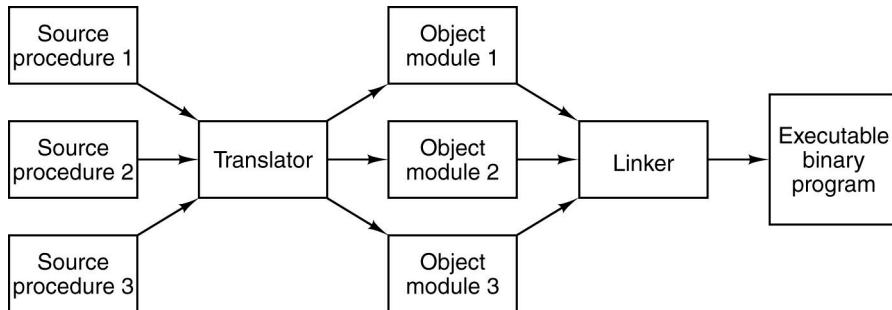
Per comprendere meglio il processo di linking, consideriamo un esempio pratico:

- **Eseguire il Link di Due File Oggetto:** Supponiamo di avere due file oggetto generati da due moduli distinti.
- **Trovare gli Indirizzi Aggiornati:** Dopo il linking, il linker aggiorna gli indirizzi delle prime istruzioni del file eseguibile completo.
- **Procedure A e B:** Supponiamo di avere le procedure A e B fornite da due moduli distinti.
- **Indirizzi delle Parole Doppie X e Y:** Supponiamo che queste parole doppie siano variabili o dati definiti nei moduli.

## Diagramma del Processo

Il diagramma allegato illustra il processo di linking e loading:

1. **Source Procedures:** Le procedure sorgente vengono tradotte dal traduttore (assemblatore o compilatore) in moduli oggetto.
2. **Object Modules:** I moduli oggetto risultanti hanno spazi di indirizzamento separati.
3. **Linker:** Il linker fonde questi moduli in un unico modulo eseguibile, assegnando indirizzi e aggiornando i riferimenti.
4. **Executable Binary Program:** Il risultato finale è un programma binario eseguibile, pronto per essere caricato in memoria e eseguito.



## ELF - File Oggetto nei Sistemi UNIX

Il formato ELF (Executable and Linkable Format) è utilizzato nei sistemi UNIX per i file oggetto. Questi file contengono diverse sezioni che svolgono ruoli specifici nel processo di linking e loading:

1. **Object File Header:** Descrive la dimensione e la posizione degli altri segmenti del file oggetto.
2. **Text Segment:** Contiene il codice in linguaggio macchina.
3. **Static Data Segment:** Contiene i dati allocati per tutta la durata del programma, sia statici che dinamici.
4. **Relocation Information:** Identifica le istruzioni e i dati che dipendono da indirizzi assoluti quando il programma viene caricato in memoria.
5. **Symbol Table:** Contiene le etichette di cui non è stata trovata una definizione, come quelle per i moduli esterni.
6. **Debugging Information:** Informazioni per il debugger, che permettono di associare le istruzioni in linguaggio macchina al codice sorgente.

## Loader

Una volta che il linker ha creato l'eseguibile, questo viene memorizzato su un supporto di memoria secondaria. Quando è il momento di eseguirlo, il sistema operativo lo carica in memoria centrale e avvia l'esecuzione. Il loader, un programma del sistema operativo, esegue le seguenti operazioni:

1. **Leggere l'Intestazione:** Determina la dimensione del programma e dei dati.
2. **Riservare spazio in Memoria:** Riserva sufficiente spazio in memoria per contenere il programma e i dati.
3. **Copiare Programma e Dati:** Copia il programma e i dati nello spazio riservato in memoria.
4. **Copiare Parametri nello Stack:** Se presenti, copia i parametri passati al main nello stack.
5. **Inizializzare Registri e Stack Pointer:** Inizializza tutti i registri necessari e lo stack pointer.
6. **Avviare l'Esecuzione:** Salta a una procedura che copia i parametri dallo stack ai registri e invoca il main.

## **Binding e Rilocazione Dinamica**

Il binding è il processo di collegamento tra nomi simbolici e indirizzi fisici. La rilocazione dinamica si riferisce al processo di aggiornamento degli indirizzi di memoria quando un programma viene spostato in memoria. Le informazioni di rilocazione possono essere scartate dopo il linking, rendendo problematico lo spostamento dei programmi in memoria.

Momenti per Effettuare il Collegamento

Il collegamento può essere effettuato in diversi momenti:

1. **Al Momento della Scrittura del Programma**
2. **Al Momento della Traduzione del Programma**
3. **Al Momento del Linking (prima del loading)**
4. **Al Momento del Loading**
5. **Al Momento dell'Esecuzione (uso di un registro di base)**

## **Collegamento Statico vs Dinamico**

Collegamento Statico

- **Incorporazione delle Librerie:** Le funzioni di libreria diventano parte del codice eseguibile.
- **Versioni delle Librerie:** Se viene rilasciata una nuova versione, un programma che carica staticamente le librerie continua a utilizzare la vecchia versione.
- **Dimensione del Programma:** La libreria può essere molto più grande del programma, rendendo i file binari eccessivamente grandi.

Collegamento Dinamico

- **DLL (Dynamically Linked Libraries):** Le funzioni di libreria non vengono collegate e caricate fino all'inizio dell'esecuzione del programma.
- **Collegamento Lazy:** Ogni procedura viene caricata solo dopo la sua prima chiamata, riducendo il tempo di avvio e l'occupazione di memoria.
  - Ogni funzione non locale inizia chiamando una procedura fasulla (dummy), posizionata dopo il codice del programma.
  - La procedura fasulla contiene un'istruzione di salto indiretto specifica per ogni funzione non locale.
  - Alla prima chiamata di una funzione di libreria, il programma esegue il salto alla procedura fasulla corrispondente.
  - La procedura fasulla esegue un salto a un codice che inserisce un identificatore della funzione desiderata in un registro e poi chiama il linker-loader dinamico.
  - Il linker-loader dinamico trova la funzione, la mappa in memoria e aggiorna l'indirizzo nella procedura fasulla per puntare direttamente alla funzione.
  - Da quel momento, la funzione è disponibile e il programma può riprendere dal punto in cui era stato interrotto per la chiamata iniziale.

## **Conclusione**

Il formato ELF e il processo di linking e loading, compresi il loader e le tecniche di binding e rilocazione dinamica, sono fondamentali per l'esecuzione efficiente dei programmi nei sistemi UNIX. La scelta tra collegamento statico e dinamico influisce sulle prestazioni, la dimensione del programma e la gestione delle librerie.

# Circuiti logici digitali di base

## Livello della Logica Digitale

La logica digitale si occupa dei circuiti digitali di base come porte logiche, registri e memoria.

Questi componenti sono alla base del funzionamento dei computer, che operano utilizzando solo due valori logici: 0 (segnale tra 0 e 1 volt) e 1 (segnale tra 2 e 5 volt). I circuiti digitali trasformano segnali binari di ingresso in segnali binari di uscita.

## Circuiti Digitali

I circuiti digitali sono dispositivi che utilizzano solo due valori logici: 0 e 1. Un circuito digitale trasforma segnali di ingresso ( $x, x_2, \dots, x_n$ ) in segnali di uscita ( $z_1, z_2, \dots, z_m$ ).

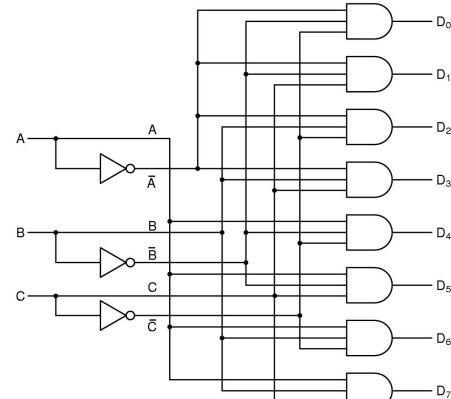
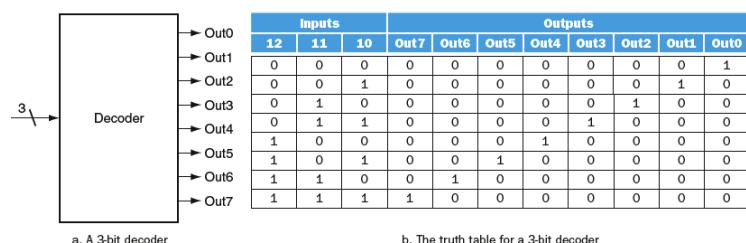
## Porte Logiche

Le porte logiche sono gli elementi primitivi dei circuiti digitali e calcolano funzioni logiche sui segnali binari. Sono utilizzate in circuiti combinatori e sequenziali. I circuiti combinatori producono un'uscita che è funzione esclusiva degli ingressi, mentre i circuiti sequenziali producono un'uscita che dipende anche dallo stato precedente.

## Circuiti Combinatori

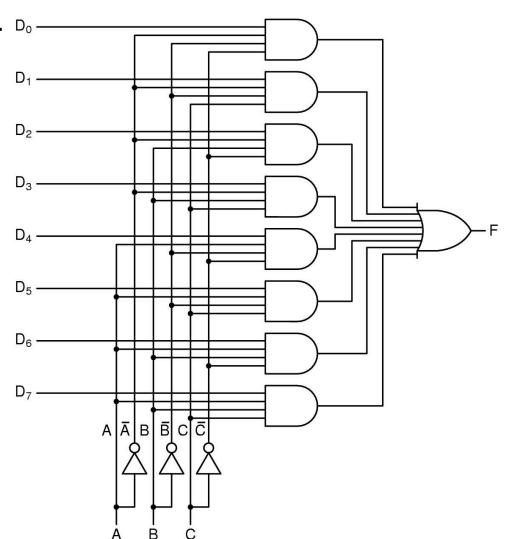
### Decoder

Un decoder prende un numero di  $n$  bit come ingresso e seleziona una delle  $2^n$  linee di uscita. Questo può essere utilizzato per attivare una certa componente o un banco di memoria. Ecco un esempio di un decoder con 3 ingressi (A, B, C).



### Multiplexer

Un multiplexer seleziona uno tra diversi ingressi per inviarlo all'uscita basandosi su linee di controllo. In generale possiamo avere  $2^n$  ingressi, 1 uscita e  $n$  ingressi di controllo.

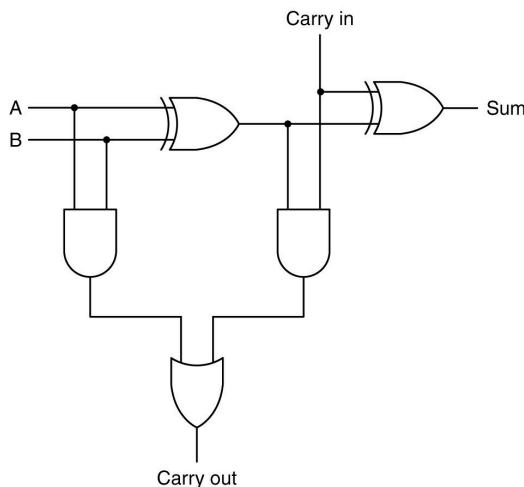


## Circuiti Numerici

### Addizionatori

Gli addizionatori ricevono due bit da sommare e un bit di riporto in ingresso, producendo un bit di risultato e un bit di riporto in uscita. Un esempio di addizionatore con gate XOR è il seguente:

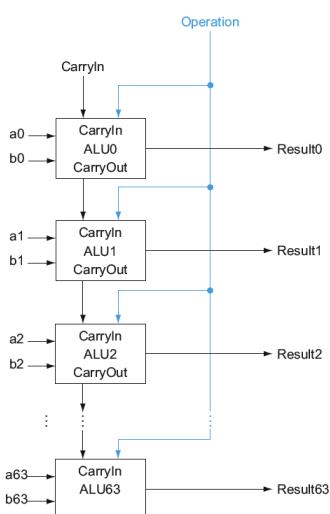
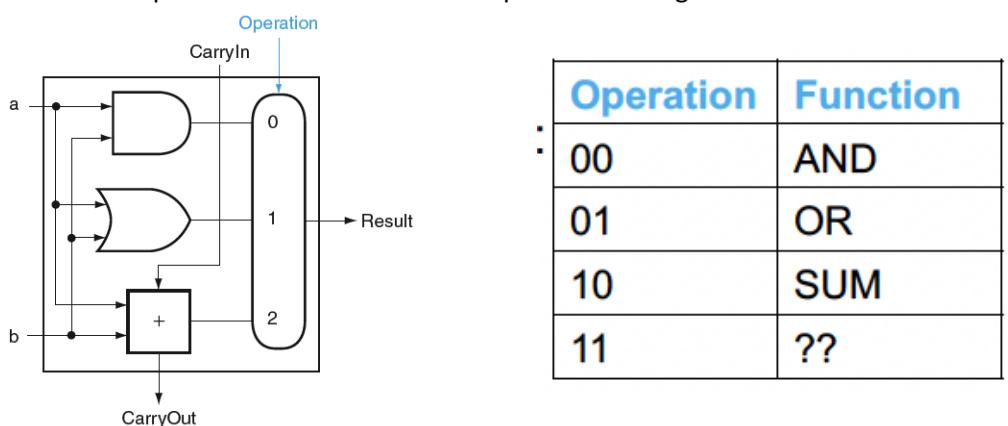
A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



## ALU (Arithmetic Logic Unit)

### ALU a 1 Bit

Una ALU ad 1 bit può calcolare diverse operazioni come AND, OR, e somma. Un ingresso di controllo seleziona l'operazione desiderata. La ALU può anche eseguire sottrazioni utilizzando il complemento a 2.



### ALU a 64 Bit

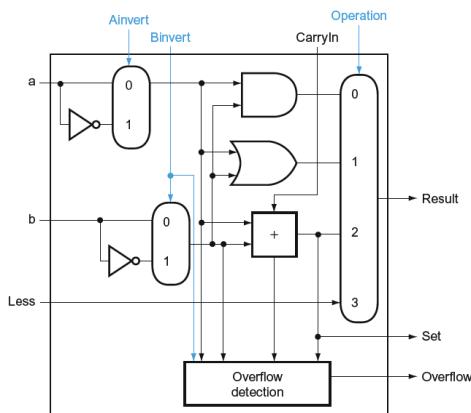
Una ALU a 64 bit esegue operazioni su operandi di 64 bit. Ogni ALU a 1 bit esegue l'operazione sulla coppia di bit degli operandi nella stessa posizione. Per sommare operandi di 64 bit, il CarryOut di ogni bit viene passato come CarryIn al bit successivo.

### ALU a 1 bit (sub) - (nor)

Dati **a** e **b** è in grado di calcolare le funzioni precedenti con **b** o con **!b**. Un ingresso di controllo aggiuntivo (**Binvert**) seleziona l'operazione desiderata. Serve ad esempio per le sottrazioni (**a-b**)

*Con le sottrazioni il CarryIn viene impostato ad 1 per il bit meno significativo.  
(Complemento a due → NOT B + 1)*

Un ingresso di controllo aggiuntivo (**Ainvert**) seleziona l'operazione desiderata.



### ALU per slt

L'istruzione Set on Less Than (slt) restituisce 1 se  $rs1 < rs2$  e 0 altrimenti, il valore di tutti i bit in uscita tranne quello meno significativo deve essere 0. Il valore dei bit meno significativi dipende dal confronto

$$(a - b) < 0 \Rightarrow a < b$$

Quindi basta vedere il bit più significativo

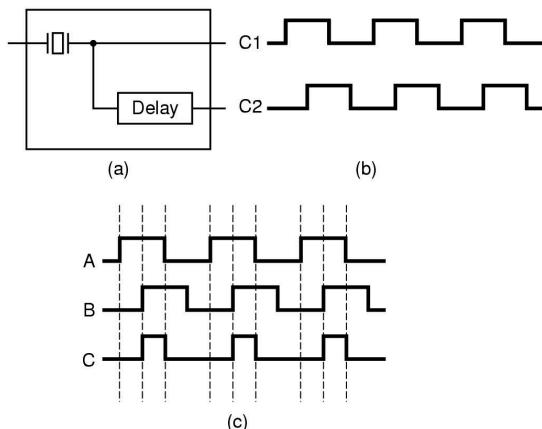
### ALU controllo overflow

Nella somma o differenza di due interi con segno in complemento a due, abbiamo overflow se i due operandi hanno lo stesso segno ma il risultato ha segno opposto (CarryIn XOR CarryOut)

### ALU per beq

L'istruzione Branch if Equal (beq) realizza un salto se due registri sono uguali, quindi eseguendo  $(a-b)$  si controlla se il risultato è zero.  $(a-b) == 0 \rightarrow a == b$  (zero vale 1 quando  $a == b$ )

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



### Clock

Il clock è un circuito che emette una serie di impulsi con una specifica **larghezza** e **frequenza**. Il tempo di ciclo di clock è l'intervallo tra i fronti corrispondenti di due impulsi consecutivi. La frequenza di clock si calcola come l'inverso del tempo di ciclo.

$$1\text{ms (millisecondo)} = 1 * 10^{-3} \text{s}$$

$$1 \text{ KHz (KiloHertz)} = 1 * 10^{-3} \text{ Hertz}$$

La frequenza specifica il numero di periodi di lock per unità di tempo (secondo), si misurano in Hertz

### Controllo delle Operazioni dell'ALU

Il controllo delle operazioni dell'ALU può essere gestito combinando ingressi per selezionare la funzione desiderata. Ad esempio, i bit per gli ingressi Anegate, Bnegate, e Operation determinano l'operazione che l'ALU deve eseguire (AND, OR, somma, sottrazione, ecc.).

## Circuiti sequenziali

I circuiti sequenziali sono invece in grado di calcolare funzioni che dipendono anche da uno stato interno, quindi riescono a memorizzare informazioni all'interno.

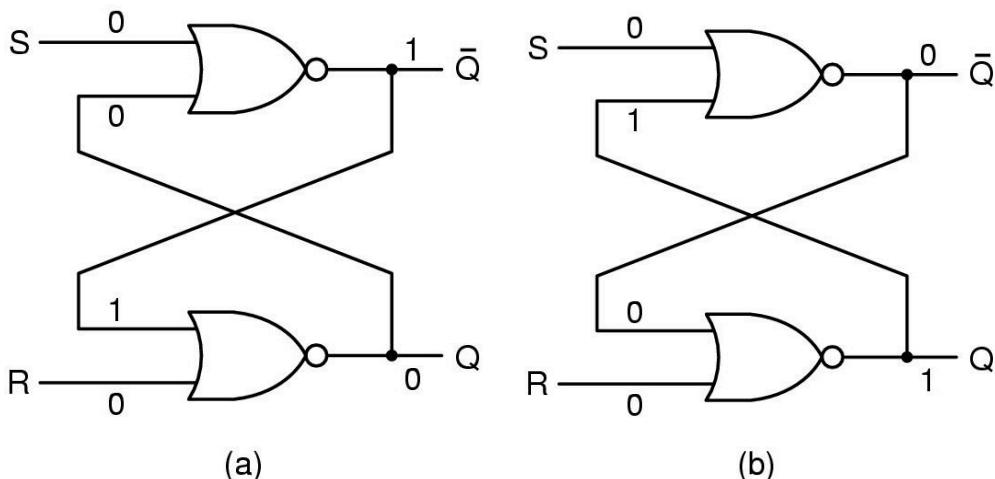
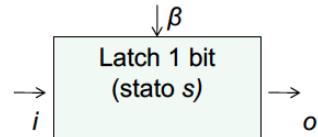
## Dispositivo che memorizza 1 bit? Latch di tipo SR!

due ingressi  $i$  e  $\beta$  ed un'uscita  $o$ .

mantiene uno stato interno  $s$

se  $\beta=1$  (store),  $o \leftarrow s \leftarrow i$

se  $\beta=0$  (hold),  $o \leftarrow s$



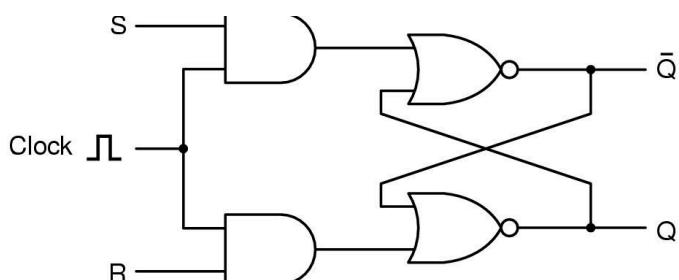
Implementazione di un dispositivo di memoria ad 1 bit

- Hold:  $R = S = 0$  (due stati **stabili**)
- Set (Store 1):  $S = 1$  e  $R = 0$  porta il latch allo stato 1
- Reset (Store 0):  $S = 0$  e  $R = 1$  porta il latch allo stato 0
- $S = R = 1$  lo stato non è stabile:
  - $Q$  imprevedibile
  - Possibile oscillazione
  - Si potrebbe innescare un oscillazione

I segnali **S** e **R** non possono essere contemporaneamente uguali a 1 per poter memorizzare un valore corretto.

Ma (**S**, **R**) sono di solito calcolati da un circuito **combinatorio**, ci mettono un pò di tempo a stabilizzarsi, quindi bisogna evitare che in questi possibili stati intermedi il latch venga scritto, la soluzione per evitare una possibile oscillazione è quella di usare un **clock**.

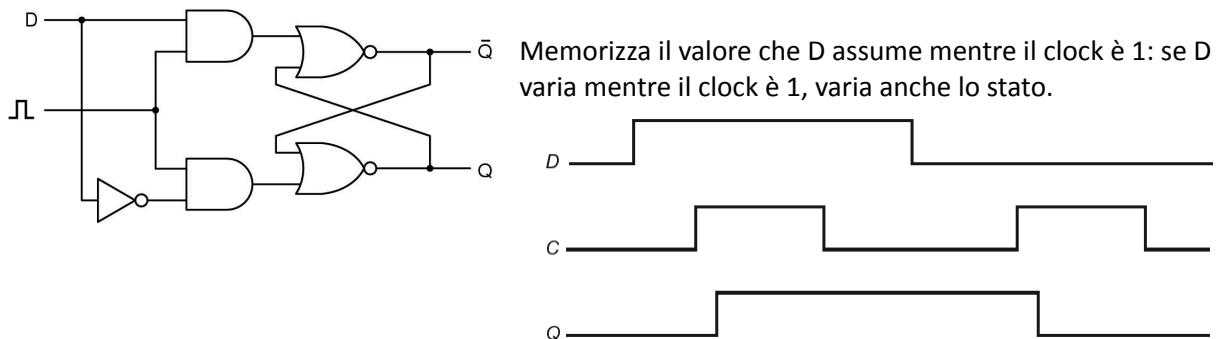
## Latch di tipo SR sincronizzato



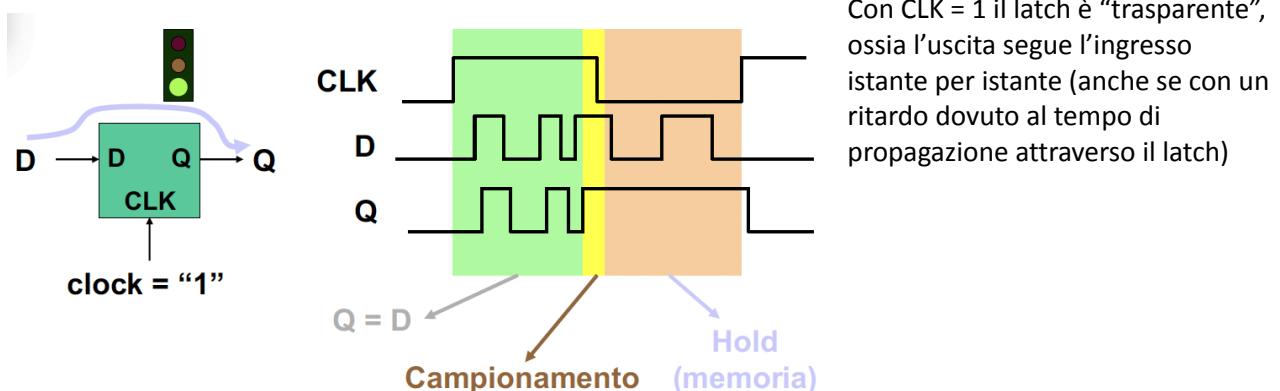
Un clock garantisce che il latch cambi stato solo in certi momenti specifici, le porte AND abilitano gli input S e R solo quando il clock è 1

## Latch di tipo D sincronizzato

Memorizza il valore che D assume mentre il clock è 1: se D varia mentre il clock è 1, varia anche lo stato  
Il latch D evita l'ambiguità dello stato S = 1, R = 1.



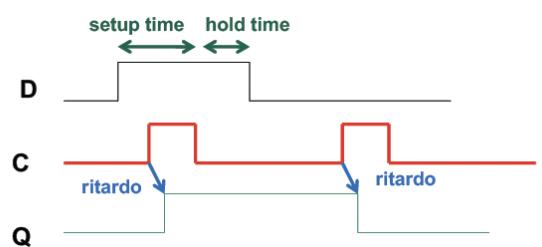
## Il problema della “trasparenza”



## Diagramma temporale del D-latch

Il segnale D, ottenuto solitamente come output di un circuito combinatorio.

- Deve essere già stabile quando C diventa alto
- Deve rimanere stabile per tutta la durata del livello alto di C (**setup time**)
- Deve infine rimanere stabile per un altro periodo di tempo per evitare problemi (**hold time**)

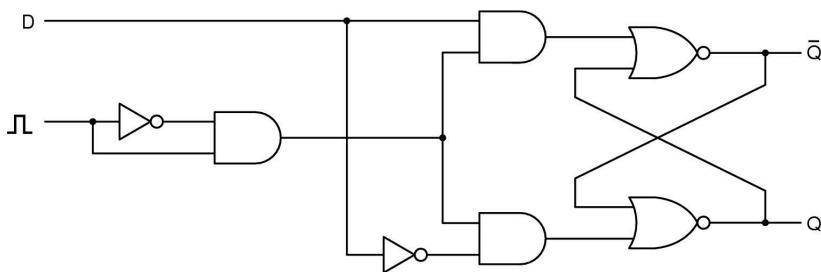


Il D-latch ha un’altro problema se il clock rimanesse alto per molto tempo, allora un eventuale segnale di ritorno “sporco”, proveniente dal circuito combinatorio potrebbe essere memorizzato nel latch.

Si possono progettare componenti di **memoria**, in cui la memorizzazione può avvenire in diversi istanti rispetto al segnale a gradino di clock:

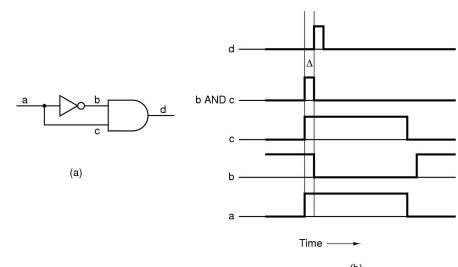
- Metodologia con **commutazione a livello**: la commutazione avviene per tutto il livello del clock (alto o basso). Il D-latch è di questo tipo
- Metodologia con **commutazione sul fronte**: la commutazione **avviene** solo nel fronte di salita o sul fronte di discesa del clock, così che la memorizzazione avvenga istantaneamente, evitando la possibilità di più memorizzazione nello stesso intervallo di clock.

## Flip-flop di tipo D

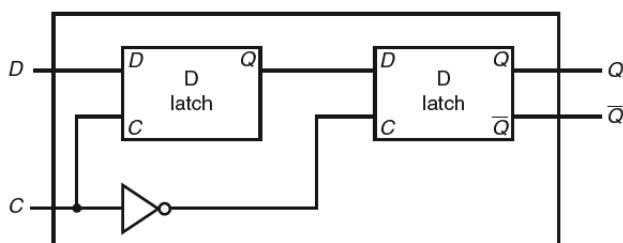


Un latch è azionato da livello (commutazione a livello).  
Un flip-flop è azionato dal fronte (commutazione sul fronte)

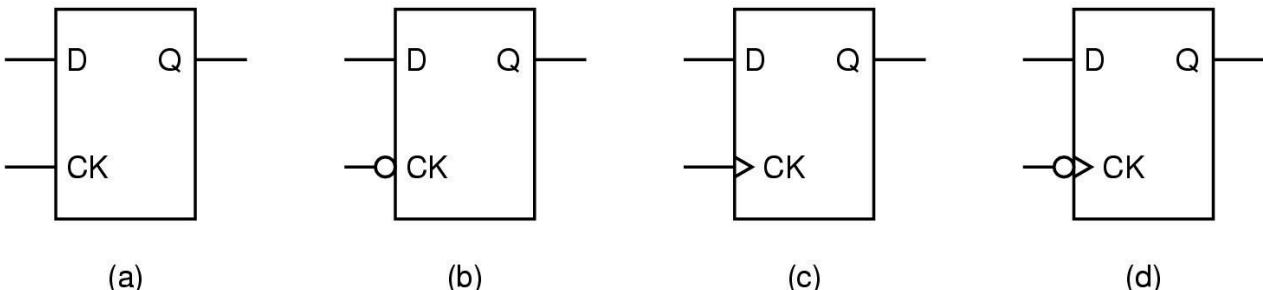
L'invertitore crea un piccolo ritardo alla propagazione del segnale **a** verso **b**  
Il latch D verrà attivato ad un ritardo fisso dopo il fronte di salita del clock  
(Attraversamento dell'AND)



## Flip-flop di tipo D sul fronte di discesa

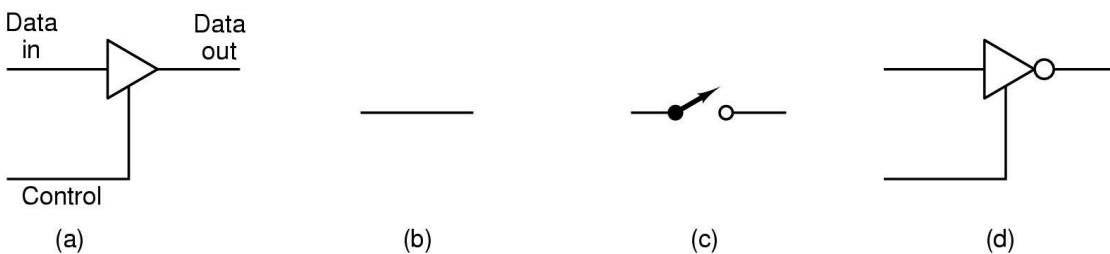


## Tutti i tipi di Flip-Flop



- A. latch di tipo D attivato con livello 1 del clock
- B. latch di tipo D attivato con livello 0 del clock
- C. flip-flop di tipo D attivato sul fronte di salita del clock
- D. flip-flop di tipo D attivato sul fronte di discesa del clock

## Buffer (non) invertente | Tri-state



- A. Il buffer (non) invertente si comporta come un filo quando l'ingresso control è alto (caso b)
- B. Il buffer disconnette DataIn e DataOut quando control è basso (caso c)
- C. Buffer invertente (caso d)

# Registri

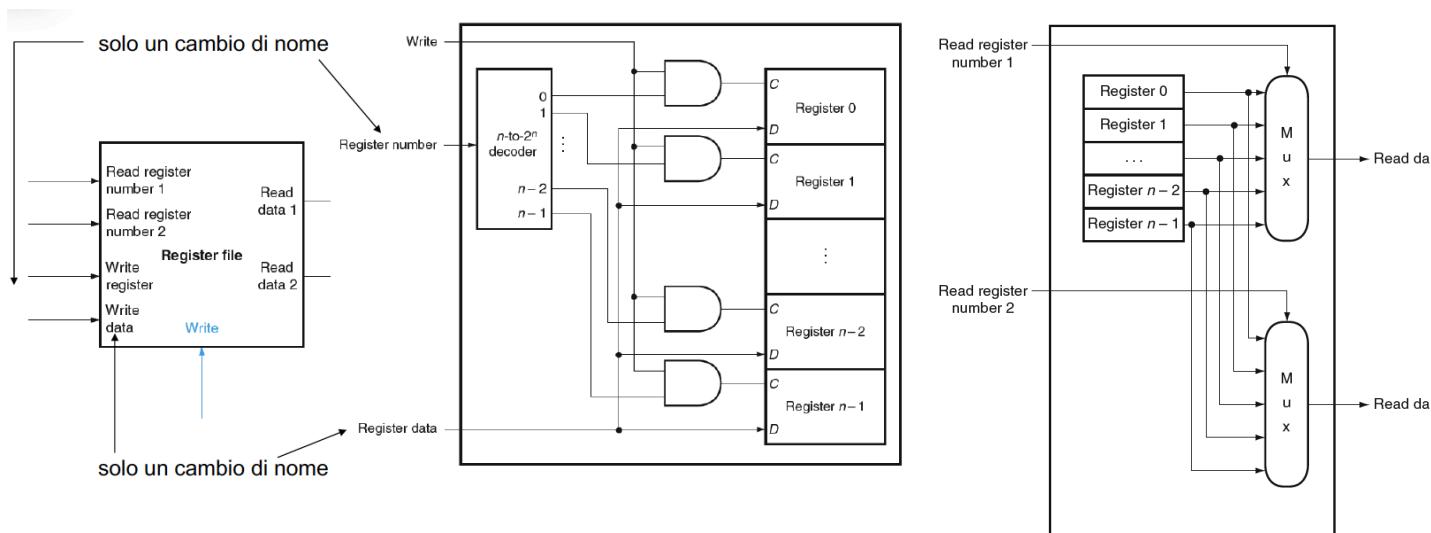
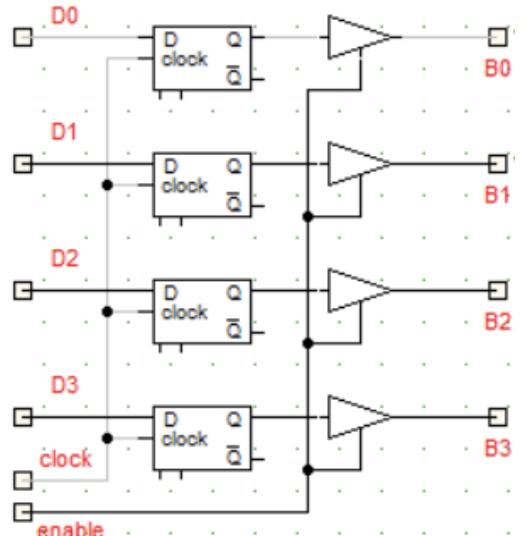
L'insieme di flip-flop D raggruppati insieme per formare un **registro**. Lo stesso clock è in ingresso a tutti i flip-flop del registro, l'ingresso enable permette (dis)connettere il registro dal bus di output tramite buffer non invertenti

## Blocco di registry (register file)

Insieme di registri leggibili e scrivibili

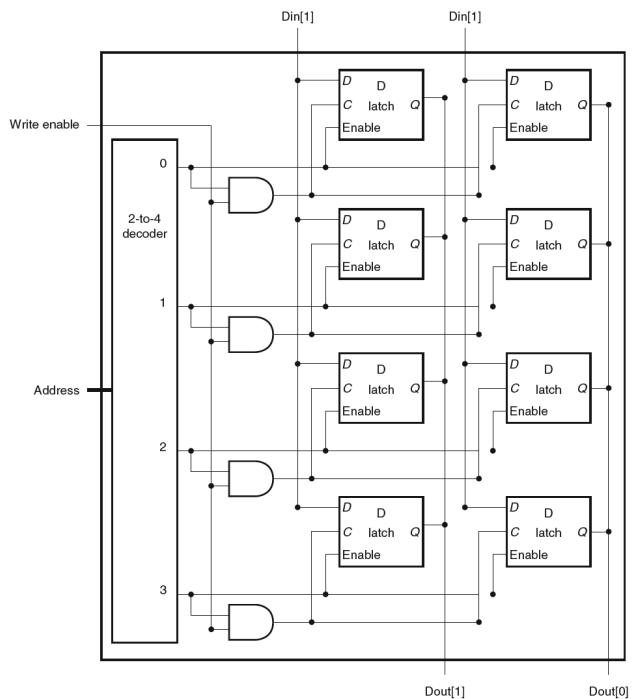
Si deve fornire in ingresso il numero del registro sul quale compiere l'operazione.

Si implementa con un multiplexer per ogni porta di lettura, con un decoder per la porta di scrittura ed un insieme di registri basati sui flip-flop di tipo D.



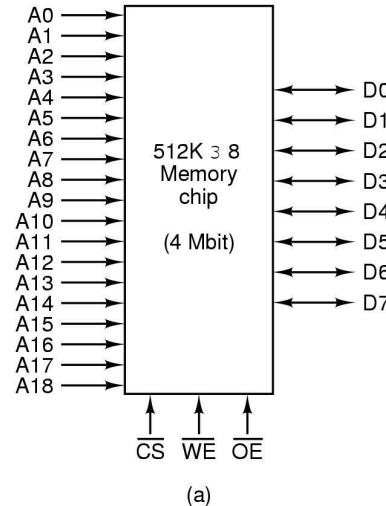
# Organizzazione della memoria

- Memoria 4 x 2
  - 5 linee di ingresso:
    - 2 per i dati in input (Din)
    - 2 per l'indirizzo (Address)
    - 1 per il bit di controllo (Write enable)
  - 2 linee di uscita
    - 2 (Dout)
- Memoria 4 x 3
  - 8 linee di ingresso:
    - 3 per i dati in input
    - 2 per l'indirizzo
    - 3 per i bit di controllo:
      - CS per Chip Select
      - RD per distinguere Read e Write
      - OE per abilitare l'output
  - 3 per output

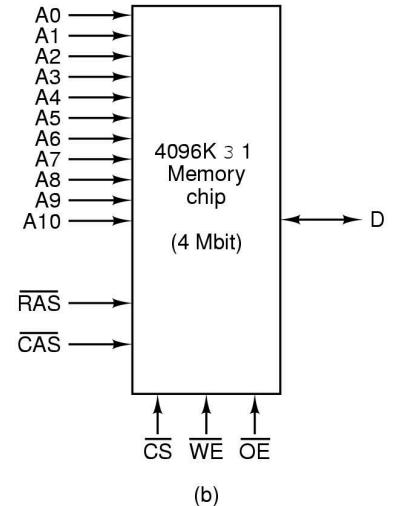


## Chip di memoria

- n linee di indirizzo corrispondono a  $2^n$  righe di flip flop
- b linee di output corrispondono a b colonne di flip-flop
- I segnali possono essere attivati quando il livello è alto o basso
- Matrici  $2^n * b$ :



(a)



(b)

## Tipi di memoria: RAM

RAM: random access memory

- SRAM: RAM statiche (flip-flop tipo D), estremamente veloci, utilizzate per realizzare la cache
- DRAM: RAM dinamiche (transistor con condensatore), vanno rinfrescati, offrono grandi capacità ma più lente
  - DRAM FPM (Fast Page Mode): organizzare in matrici
  - DRAM EDO (Extended Data Output): con semplice pipeline per l'output
  - SDRAM (Synchronous DRAM): usata inizialmente su cache e memorie centrali
  - SDRAM-DDR (Double Data Rate)

## Tipi di memoria: ROM

ROM: Read-Only Memory, utilizzati per dati che non devono essere modificati

- PROM (Programmable ROM)
- EPROM (Erasable PROM): Tramite luce ultravioletta
- EEPROM (Electrically Erasable PROM): memoria cancellabile per mezzi di impulsi elettrici (più lente)
- FLASH (EEPROM cancellabile a blocchi)

## Macchine a Stati Finiti

Dispositivo automatico in grado di interagire con l'ambiente esterno, a fronte di uno stimolo in ingresso (input), esibisce un comportamento in uscita (output) che dipende anche da informazioni memorizzate in elementi interni (stati).

Ci limitiamo a macchine con memoria finita (Le macchine di **Turing** hanno memoria illimitata).

Una macchina a stati finiti (FSM), è vista come una scatola che possiamo descrivere mostrando cosa succede ad ogni passo.

Legge un simbolo in ingresso (che appartiene ad un insieme finito A), produce un simbolo in uscita (che appartiene ad un insieme finito B), cambia il proprio stato interno.

Le Macchine a stati finiti sono implementati da reti logiche sequenziali, esse memorizzano lo stato nel latch attraverso un clock.

Il clock determina il ritmo dei calcoli e delle relative operazioni di memorizzazione: il circuito sequenziale viene detto **sincrono**.

## Analisi di Reti Sequenziali Sincrone

L'analisi delle reti sequenziali sincrone si concentra sull'esame del comportamento di circuiti logici che dipendono sia dallo stato corrente che dagli ingressi esterni. In una rete sequenziale sincrona, il cambiamento di **stato** è regolato da un segnale di **clock**. L'analisi comprende la determinazione delle tabelle di transizione di stato e delle uscite, basate sulle equazioni di stato derivate dal circuito.

## Tabelle di Stato

Le tabelle di stato rappresentano una descrizione tabellare dei cambiamenti di stato e delle uscite di una rete sequenziale in risposta agli ingressi. Ogni riga della tabella elenca lo stato corrente, l'ingresso, il prossimo stato e l'uscita. Queste tabelle sono cruciali per comprendere il comportamento dinamico dei circuiti **sequenziali** e per progettare il corretto funzionamento del sistema.

## Circuiti di Mealy e di Moore

I circuiti sequenziali si dividono principalmente in due categorie: Mealy e Moore.

- **Circuito di Mealy:** In un circuito di Mealy, le uscite dipendono dagli **ingressi** correnti e dallo **stato** corrente. Questo comporta che le uscite possono cambiare subito dopo un cambiamento negli ingressi, senza aspettare il prossimo ciclo di clock.
- **Circuito di Moore:** In un circuito di Moore, le uscite dipendono solo dallo **stato** corrente. Ciò significa che le uscite cambiano solo alla transizione di stato, sincronizzate con il segnale di clock. Questa caratteristica rende i circuiti di Moore generalmente più **semplici** da analizzare e prevedere rispetto ai circuiti di Mealy.

## Diagramma di Stato

Un diagramma di stato è una rappresentazione grafica della transizione degli stati di una rete sequenziale. I nodi del diagramma rappresentano gli stati, mentre le frecce tra i nodi rappresentano le transizioni di stato, etichettate con le condizioni che causano tali transizioni. I diagrammi di stato sono utili per visualizzare e progettare il comportamento del sistema, facilitando l'individuazione di eventuali errori di progettazione.

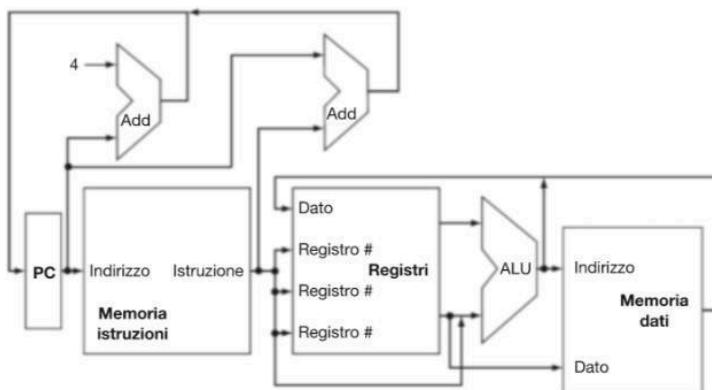
## Sintesi di Reti Sequenziali Sincrone

La sintesi di reti sequenziali sincrone è il processo di progettazione di un circuito sequenziale a partire da una specifica di comportamento **desiderato**. La sintesi coinvolge la definizione degli stati, la creazione delle tabelle di stato e l'implementazione delle equazioni logiche necessarie. Questo processo può includere la minimizzazione degli stati per ridurre la complessità del circuito e l'ottimizzazione delle equazioni logiche per migliorare l'efficienza.

## Il processore

Esamineremo un'implementazione che comprende le seguenti istruzioni di base del RISC-V:

- le istruzioni di riferimento alla memoria `ld` e `sd`;
- le istruzioni aritmetico-logiche `add`, `sub`, `and` e `or`;
- le istruzioni di salto condizionato dal risultato di un test di uguaglianza, `beq`.
- Per ogni istruzione questi sono i passi:
  1. Prendi l'istruzione seguente dalla memoria (fase di `fetch`);
  2. Cambia il PC per indicare l'istruzione seguente
  3. Determina il tipo dell'istruzione appena letta (fase di `decode`);
    - a. L'unità di controllo è la parte effettiva che decodifica le istruzioni
  4. Se l'istruzione usa una parola in memoria, determina dove si trova
  5. Metti la parola, se necessario, in un registro della CPU
  6. Esegui l'istruzione (fase di `execute`);
    - a. Eseguire un calcolo effettivo
    - b. Elaborare il contenuto per determinare l'indirizzo in memoria (load o store)
    - c. Eseguire un confronto (jump)
  7. Torna al punto 1.



Questo è uno schema ad alto livello di astrazione dell'implementazione di un RISC-V.

## Convenzioni del progetto logico

Gli elementi funzionali che costituiscono l'unità di elaborazione del RISC-V sono costituiti da due diverse classi di elementi logici: elementi che operano sui dati, detti **combinatori**, ed elementi che contengono lo stato, detti **sequenziali**.

La ALU raffigurata sopra è un esempio di elemento **combinatorio**, ovvero che in ogni istante i suoi output dipendono solo dagli input ricevuti nello stesso istante.

Gli elementi sequenziali contengono lo *stato* e, per questo, hanno al loro interno elementi di memoria.

La memoria **istruzioni**, la memoria **dati** e i **registri** della figura sopra sono esempi di elementi di stato (**sequenziali**). Un elemento di stato possiede almeno due ingressi e un'uscita. Gli ingressi richiesti sono il valore da scrivere nell'elemento e il *clock*, che determina quando scrivere. L'uscita di un elemento di stato è il valore contenuto al suo interno, scritto in un ciclo di *clock* precedente.

# Metodologia di temporizzazione

La **metodologia di temporizzazione** definisce quando i segnali possono essere scritti e quando possono essere letti. È importante temporizzare le operazioni di lettura e scrittura perché, se un segnale venisse letto e contemporaneamente scritto, il valore letto potrebbe non corrispondere a quello atteso.

Utilizziamo una metodologia di **temporizzazione sensibile ai fronti** (*edge-triggered*): essa garantisce che il valore memorizzato all'interno di un elemento sequenziale venga aggiornato solamente in corrispondenza di un fronte del segnale di clock.

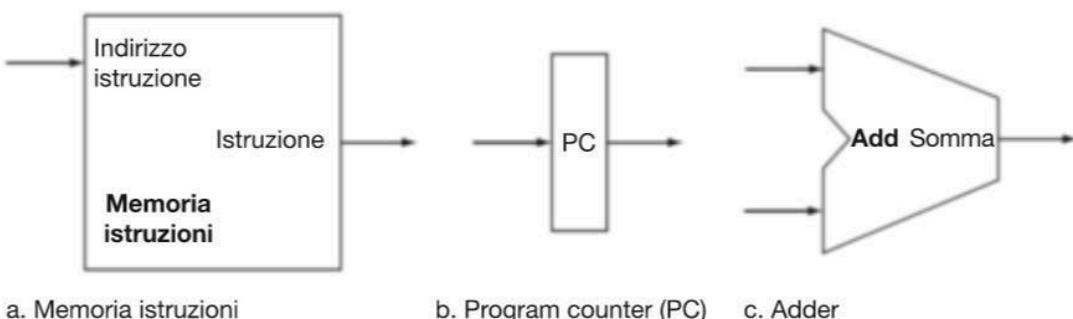
Poiché solo gli elementi sequenziali memorizzano i dati, un qualsiasi circuito combinatorio deve ricevere l'input da un insieme di elementi di stato. Gli ingressi sono i valori che erano stati scritti in un ciclo di clock precedente, mentre gli output del circuito combinatorio sono i valori che potranno essere utilizzati in un ciclo di clock successivo.

La metodologia sensibile ai fronti permette di leggere il contenuto di un registro, inviare il valore attraverso uno o più blocchi di logica combinatoria e scrivere lo stesso registro nello stesso ciclo di clock.

In questo modo non si rischia di innescare una retroazione all'interno dello stesso ciclo di clock, e il circuito funziona correttamente.

Quasi tutti gli elementi di stato e combinatori dell'architettura RISC-V a 64 bit hanno ingressi e uscite di ampiezza pari a 64 bit, essendo l'ampiezza della maggior parte dei dati elaborati dal processore.

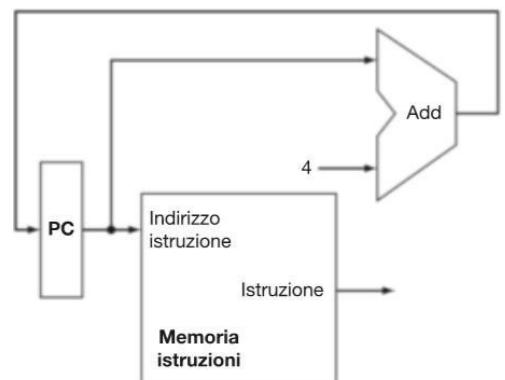
# Realizzazione di un'unità di elaborazione



I primi elementi dell'**unità di elaborazione** (*datapath elements*) sono:

- un'**unità di memoria** in cui salvare le istruzioni del programma e che sia in grado di fornire in uscita l'istruzione associata all'indirizzo dato in ingresso;
- Il **program counter**, che è un registro utilizzato per memorizzare l'indirizzo dell'istruzione corrente;
- Un sommatore per **incrementare** il PC (di 4 byte nel caso più semplice) e ottenere l'indirizzo dell'istruzione successiva (può essere costruito a partire dalla ALU).

Si possono combinare i tre elementi per formare una unità di elaborazione che prelevi le istruzioni e incrementi il PC per ottenere l'indirizzo dell'istruzione successiva del programma.



## Istruzioni in formato R (add, sub, and, or)

Tutte le istruzioni di questo tipo **leggono** due registri, eseguono un'operazione con la ALU sul contenuto di questi due registri e, alla fine, scrivono il risultato in un registro.

I registri universali a 32 bit del processore sono raccolti nel **register file**, un insieme di registri in cui ciascuno di essi può essere letto o scritto specificando il numero ad esso associato all'interno dell'insieme. Il register file contiene lo stato dei registri del calcolatore. Avremo inoltre bisogno di una ALU per operare sui valori letti.

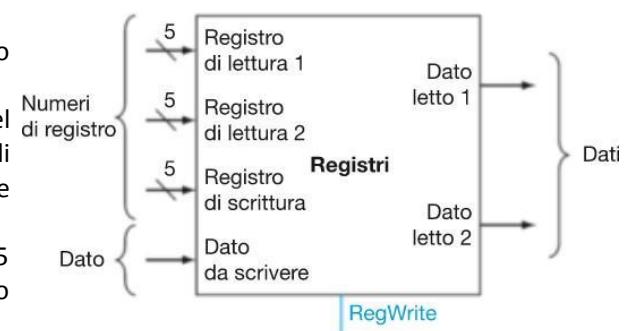
Dato che le istruzioni di tipo R hanno tre registri come **operandi**, per ciascuna istruzione dovremo leggere due dati di una word ciascuno dal register file e poi scrivere il risultato, sempre nel register file.

Per scrivere un dato di una word serviranno due ingressi:

il primo deve specificare il *numero del registro di scrittura*, il secondo deve fornire il *dato* da scrivere.

Il register file fornisce in qualsiasi momento in uscita il contenuto del registro letto; la scrittura, invece, viene controllata da un segnale di controllo esplicito, "RegWrite". Quindi serviranno complessivamente quattro ingressi e due uscite.

Gli ingressi che specificano il numero dei registri hanno ampiezza di 5 bit in modo da specificare 32 registri, mentre i bus dei dati in ingresso e in uscita sono di 64 bit ciascuno.



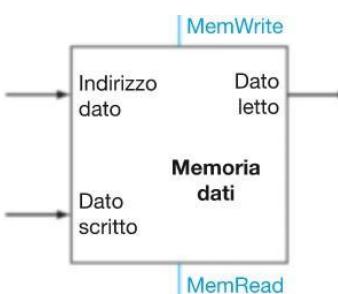
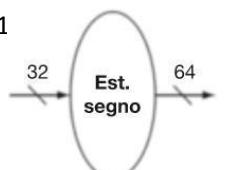
Qui è mostrata una ALU che riceve due input di 64 bit e produce un risultato su 64 bit; inoltre produce un segnale a 1 bit che vale 1 se il risultato dell'operazione è 0.

## Istruzioni di caricamento di un registro (load) e di trasferimento alla memoria (store)

Queste istruzioni hanno la forma generale **ld x1, offset(x2)** e **sd x1, offset(x2)** e calcolano un indirizzo di memoria sommando il contenuto del registro base (**x2**) al campo offset di 12 bit.

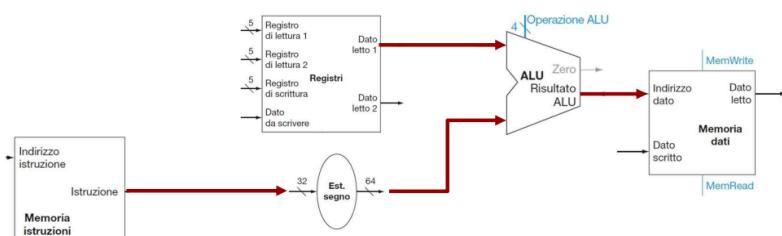
Se l'istruzione è una store il dato da memorizzare deve essere letto dal register file (dove risiede in **x1**), mentre se è una load il valore letto dalla memoria deve essere scritto all'interno del register file nel registro specificato (**x1**). Per eseguire queste istruzioni occorrono sia il register file che la ALU.

Inoltre saranno necessarie anche una unità per l'**estensione del segno** del campo offset (da 1 un'unità di memoria dati in cui scrivere o da cui leggere il dato).



La memoria dati viene scritta dalle istruzioni di store, e quindi sarà dotata dei segnali di controllo sia di lettura sia di scrittura; riceverà in ingresso, inoltre, l'indirizzo e il dato che deve essere scritto.

(b) Tipo I      immediate[11:0]      rs1      funz3      rd      codop



## Istruzioni di salto condizionato (beq)

La beq ha tre operandi: due registri il cui contenuto viene confrontato per determinare se è uguale, e un offset di -4096 a 4094 (SB Immediato) utilizzato per calcolare l'**indirizzo di destinazione del salto** (sommendo il campo offset dell'istruzione, dopo averlo esteso a 32 bit con segno, al PC).

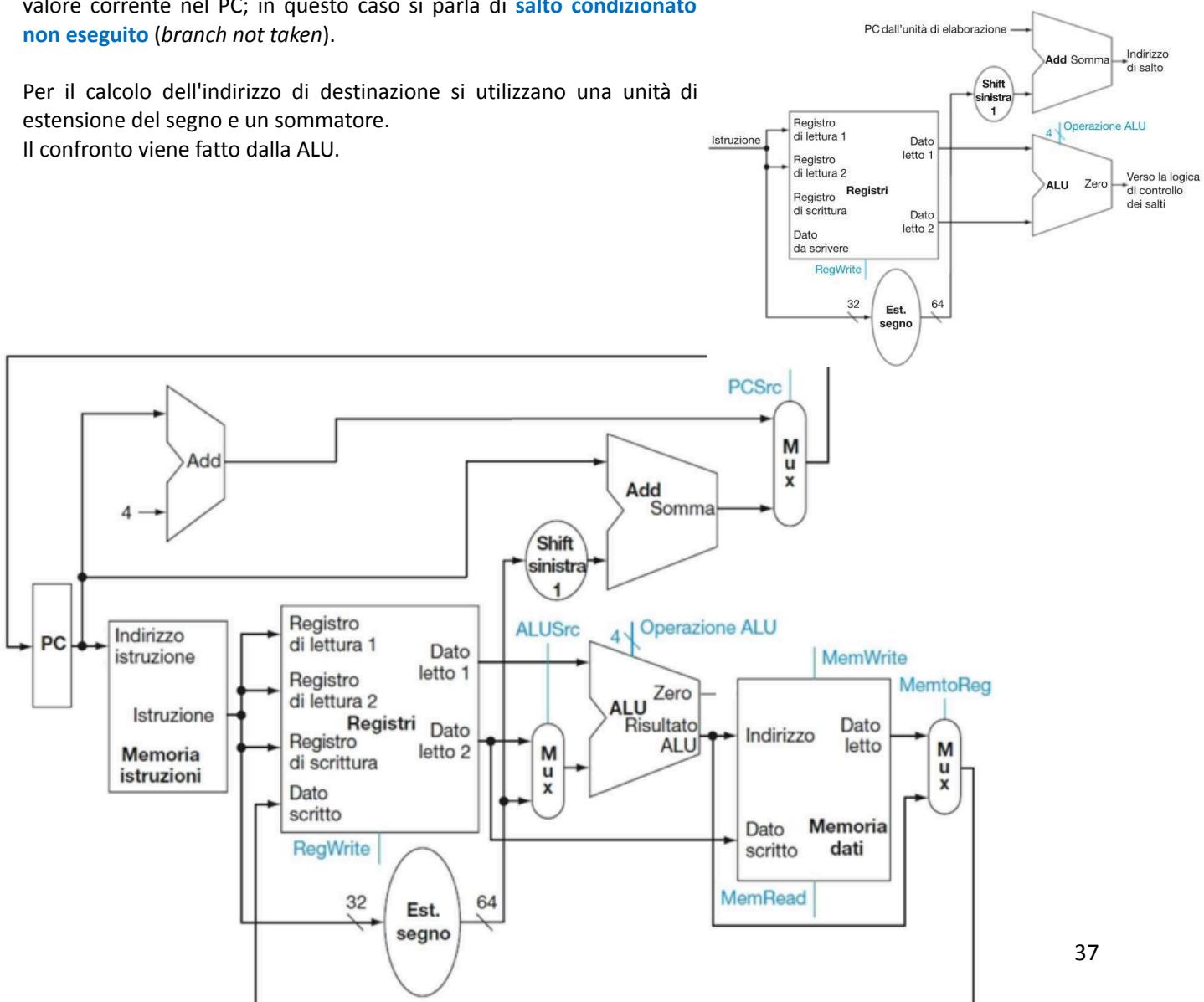
- L'indirizzo di base per il calcolo dell'indirizzo di salto è quello dell'istruzione di salto stessa.
  - Il campo offset viene spostato di 1 bit a sinistra, in modo tale che l'offset non codifichi lo spiazzamento in numero di **byte** ma in numero di half word. Tale spostamento aumenta lo spazio di indirizzamento dell'offset di un fattore 2 rispetto alla codifica dello spiazzamento in byte.
1. indirizzamento **immediato** (immediate addressing), in cui l'operando è una costante contenuta nell'istruzione stessa;
  2. indirizzamento tramite **registro** (register addressing), in cui l'operando è un registro;
  3. indirizzamento tramite **base e spiazzamento** (base and displacement addressing), in cui l'operando è in una locazione di memoria individuata dalla somma del contenuto di un registro e di una costante contenuta nell'istruzione stessa;
  4. indirizzamento relativo al program **counter** (PC-relative addressing), in cui l'indirizzo di salto è la somma del contenuto del program counter e di una costante contenuta nell'istruzione stessa.

Per gestire questa ultima complicazione è necessario far scorrere il campo offset di 1 bit a sinistra.

Oltre a calcolare l'indirizzo di destinazione del salto, bisogna determinare se l'istruzione da eseguire dopo sia quella nella posizione di memoria successiva oppure quella contenuta all'indirizzo di destinazione del salto. Quando la **codifica** del salto è vera (i due operandi sono uguali) l'indirizzo di destinazione del salto diventa il nuovo valore del PC e si parla di **salto condizionale eseguito** (*branch taken*); se il contenuto degli operandi è diverso, il valore del PC viene incrementato di 4 e diventa il valore corrente nel PC; in questo caso si parla di **salto condizionato non eseguito** (*branch not taken*).

Per il calcolo dell'indirizzo di destinazione si utilizzano una unità di estensione del segno e un sommatore.

Il confronto viene fatto dalla ALU.



## Unità di Controllo

Fino ad ora abbiamo completato un'unità di elaborazione dati elementare, manca l'**unità di controllo**, essa dovrà accettare dei valori in ingresso e generare:

- un segnale di scrittura per ciascun elemento di stato
- un segnale di selezione per ciascun multiplexer
- i segnali di controllo per l'ALU

### Controllo operazione della ALU

Per il controllo della ALU, ci serve:

- 1 bit per l'ingresso Ainvert
- 1 bit per l'ingresso Binvert
- 2 bit per gli ingressi Operation

Per le istruzioni **load** e **store** la ALU deve eseguire una somma per calcolare l'indirizzo in memoria.

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

Per le istruzioni di **Tipo R**, la ALU deve eseguire delle operazioni (AND, OR, somma o sottrazione) in funzione del valore dei 7 bit del campo funz7 e dei 3 bit del campo funz3.

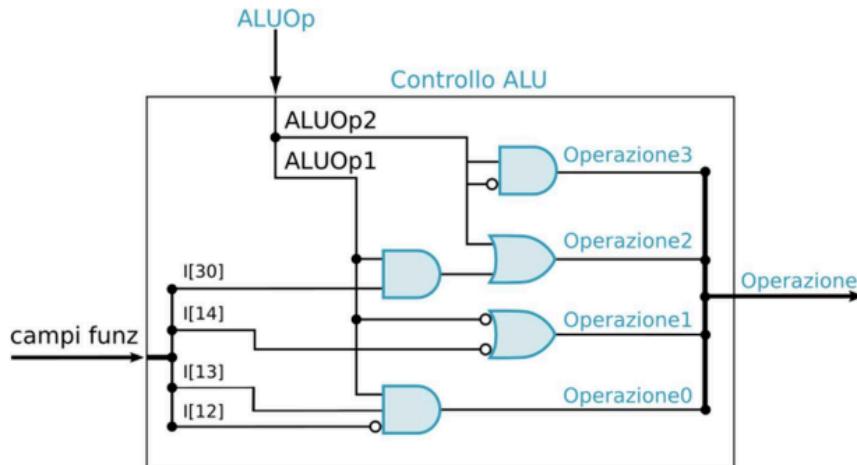
Per l'istruzione di **salto condizionato**, la ALU deve eseguire una sottrazione tra i due operandi e controllare se il risultato è zero.

Quindi, i 4 bit di controllo della ALU possono essere generati da una piccola unità di controllo che riceve in ingresso i campi funz7 e funz3 dell'istruzione e un campo di controllo su 2 bit, chiamato **ALUOp**.

- **ALUOp** = 00 → somma per le istruzioni load e store
- **ALUOp** = 01 → sottrazione per le beq
- **ALUOp** = 10 → l'operazione viene determinata dal contenuto dei campi funz7 e funz3.

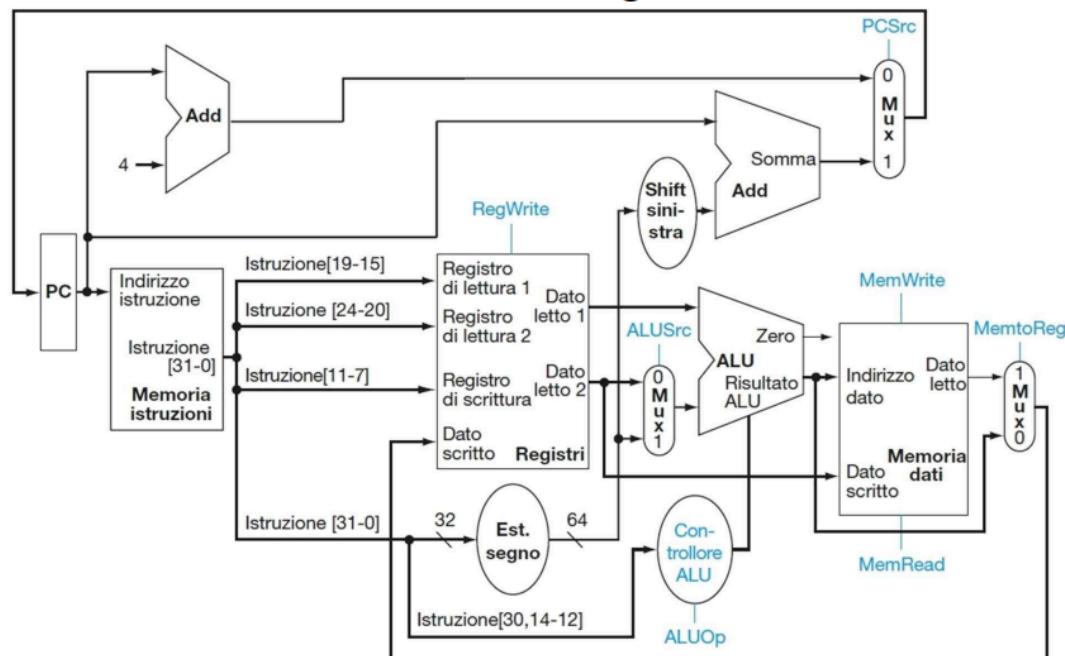
Livelli multipli di decodifica: l'unità di controllo principale imposta i bit di **ALUOp**, poi utilizzati come ingressi dell'unità di controllo della ALU, che genera i segnali effettivi della ALU.

Id / sd	ALUOp		Campo funz7							Campo funz3			Operazione
	ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
beq	X	1	X	X	X	X	X	X	X	X	X	X	0110
add	1	X	0	0	0	0	0	0	0	0	0	0	0010
sub	1	X	0	1	0	0	0	0	0	0	0	0	0110
and	1	X	0	0	0	0	0	0	0	1	1	1	0000
or	1	X	0	0	0	0	0	0	0	1	1	0	0001



## Codop

Codice operativo (codop): il campo che denota il tipo di operazione e il formato di un'istruzione.  
Il codop è sempre contenuto nei bit 6:0, a seconda del codop, il campo funz3 (12:14) e funz7 (31:25) servono come campi di estensione del codice operativo.



Nome del segnale	Effetto quando non asserito	Effetto quando asserito
RegWrite	Nullo	Il dato viene scritto nel register file nel registro individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall'estensione del segno dei 12 bit del campo immediato dell'istruzione
PCSrc	Nel PC viene scritta l'uscita del sommatore che calcola il valore di $PC + 4$	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto
MemRead	Nullo	Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea "Dato letto"
MemWrite	Nullo	Il contenuto della memoria nella posizione puntata dall'indirizzo viene sostituito con il dato presente sulla linea "Dato scritto"
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati

Tutti gli elementi di stato ricevono il clock come ingresso隐式. Il clock controlla le operazioni di scrittura.

L'unità di controllo imposta tutti i segnali tranne **PCSrc**, basandosi esclusivamente sul codice operativo dell'istruzione stessa, esso viene asserito se l'istruzione è una branch if equal, ma anche se l'uscita Zero della ALU, utilizzata per il confronto di uguaglianza, è vera.

## Riassunto

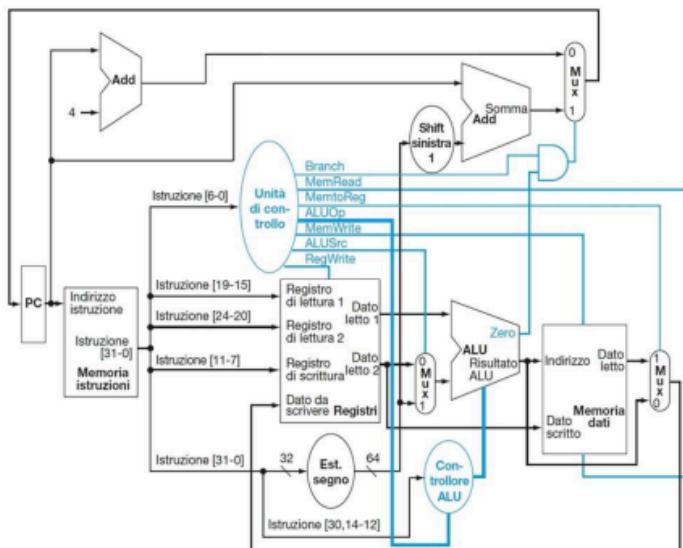
## Ingresso:

- 7 bit dell'istruzione (codop)

**Uscite:**

- Due segnali a 1 bit utilizzati per controllare i multiplexer (**ALUSrc** e **MemtoReg**)
  - Tre segnali a 1 bit per controllare lettura e scrittura del register file e della memoria dati (**RegWrite**, **MemRead**, **MemWrite**)
  - Un segnale a 1 bit utilizzato come segnali di controllo per i salti condizionati (**Branch**)
  - Un segnale di controllo a 2 bit per la ALU (**ALUOp**)

Istruzione	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
Tipo R	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1



# Gerarchia delle memorie

## Principio di località

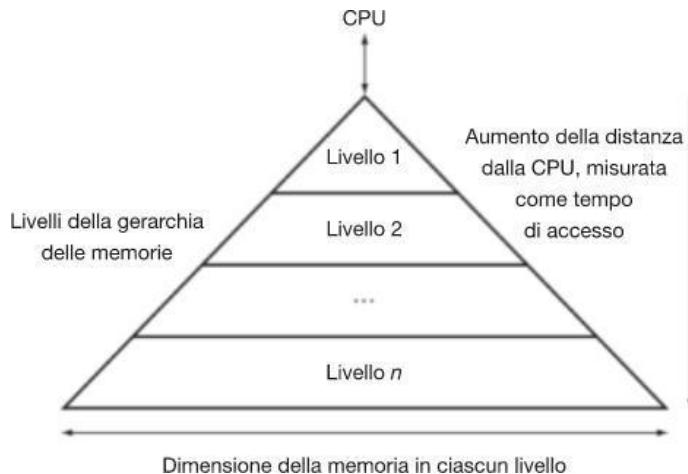
Sta alla base del comportamento dei programmi in un calcolatore: afferma che, in un certo istante di tempo, accede soltanto a una porzione relativamente piccola del suo spazio di indirizzamento.

Esistono due diversi tipi di località:

- Località temporale: quando si fa riferimento a un elemento, c'è la tendenza a fare riferimento allo stesso elemento dopo poco tempo;
- Località 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 a esso.

Il principio di località viene sfruttato strutturando la memoria di un calcolatore in forma gerarchica.

La **gerarchia delle memorie** 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 (di solito più piccole). La memoria più veloce è posta vicino al processore e quella più lenta, meno costosa, è posizionata più lontano.



Anche i dati sono organizzati in modo **gerarchico**: un livello più vicino al processore contiene in generale un sottoinsieme dei dati memorizzati in ognuno dei livelli di memoria.

Una gerarchia delle memorie può essere composta da più livelli, ma i dati vengono di volta in volta trasferiti solo tra due livelli vicini.

La più piccola quantità di informazione che può essere presente o assente in questa gerarchia su due livelli è denominata **blocco** o **linea**.

Se il dato richiesto dal processore è contenuto in uno dei blocchi del livello superiore, si dice che la richiesta ha avuto successo e si indica con **hit**. Se il dato non viene trovato nel livello superiore, si dice che la richiesta fallisce e si indica con **miss**. In questo secondo caso, per trovare il blocco contenente il dato richiesto, occorre accedere al livello inferiore della gerarchia.

La frequenza degli hit si chiama **hit rate** e viene utilizzata come indice delle prestazioni della gerarchia delle memorie; la frequenza delle miss, **miss rate**, è pari a (a - hit rate).

La velocità degli accessi è importante sia in caso di successo sia in caso di fallimento, quindi si definisce il **tempo di hit** come tempo di accesso al livello superiore della gerarchia, e comprende anche il tempo necessario a stabilire se c'è successo o fallimento.

La **penalità di miss** è il tempo necessario a sostituire un blocco del livello superiore con un nuovo blocco, caricato dal livello inferiore, e a trasferire i dati al processore.

## Tecnologia SRAM

In una memoria SRAM il dato viene memorizzato per tutto il tempo in cui l'alimentazione è attiva.

Le SRAM sono semplicemente dei circuiti integrati organizzati come vettori di memoria che hanno solo una porta d'accesso che può fornire sia la lettura che la scrittura. Hanno uno stesso tempo di accesso per tutti i dati, anche se i tempi di accesso in lettura e scrittura possono essere diversi.

Le SRAM non hanno bisogno di essere rinfrescate (*refresh*), per cui il loro accesso è **molto vicino** al periodo di clock.

## Tecnologia DRAM

In una memoria RAM *dinamica* (DRAM) il dato viene memorizzato come carica in un condensatore e un solo transistor è sufficiente per leggere il dato o per sovrascriverlo.

Le DRAM sono molto più dense e il costo per bit è inferiore a quello delle SRAM.

Dato che nelle DRAM l'informazione viene memorizzata in un condensatore, non rimane indefinitamente e occorre rinfrescare periodicamente (viene letto il contenuto e riscritto, in due cicli *cock*).

## Memorie flash

Le memorie flash sono un tipo di memoria a sola lettura, cancellabile elettricamente e programmabile (EEPROM). A differenza delle DRAM, i bit delle memorie flash si deteriorano dopo un certo numero di scritture. Per questo, la maggior parte dei dispositivi che utilizzano memorie flash contiene un controllore che distribuisce le scritture consentite rimappando i blocchi di memoria che sono stati scritti più spesso sui blocchi che sono stati scritti meno di frequente (tecnica chiamata *livellamento dell'usura*, o *wear leveling*). Questa tecnica fa diminuire le prestazioni teoriche ma è necessaria. D'altro canto, la presenza di un controllore fa sì che vengano identificate le celle di memoria difettose, migliorando le prestazioni.

## Memorie a disco

Un disco magnetico è formato da un gruppo di dischi, detti *piatti*, che ruotano solidali a una velocità compresa tra i 5400 e i 1500 giri al minuto. Ciascun piatto è ricoperto da materiale **magnetico registrabile**. Per scrivere e leggere dati, poco al di sopra delle superfici di ogni disco, è posizionato un *braccio mobile* contenente una piccola bobina elettromagnetica (*testina di lettura/scrittura*). La superficie di ogni disco è divisa in cerchi concentrici chiamati **tracce**, ognuna delle quali è a sua volta suddivisa in **settori**, che sono l'unità di memorizzazione delle informazioni.

## Memoria cache

Il termine cache viene usato per indicare sistemi di memoria gestiti in modo tale da ottenere i massimi benefici dalla località degli accessi. Quasi tutti i calcolatori prodotti oggi contengono memorie cache.

### Allocazione

La memoria principale consiste di  $2^n$  byte indirizzabili, la memoria può essere considerata anche organizzata in **blocchi di k byte** ( $M = 2^n/k$  blocchi).

La cache consiste di  $C$  linee di  $k$  byte (blocchi)  $C \ll M$

Come si fa a sapere se un dato è presente nella cache? Se presente, come facciamo a trovarlo?

Se ogni parola può essere scritta in una sola posizione della cache, allora sappiamo dove trovarla, ammesso che la parola sia effettivamente presente. Il modo più semplice per associare una sola locazione della cache ad ogni parola della memoria consiste nel definire una corrispondenza tra l'*indirizzo in memoria* della parola e la locazione nella cache.

Questa organizzazione è detta **a mappatura diretta** (*direct mapped cache*).

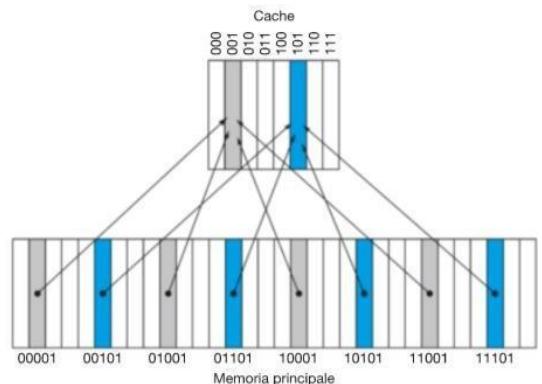
Quasi tutte le cache di questo tipo utilizzano la seguente operazione per trovare il blocco che corrisponde a un dato indirizzo della memoria principale:

(*indirizzo del blocco*) modulo (*numero di blocchi nella cache*)

Come facciamo a sapere se la parola richiesta si trova nella cache oppure no?

Alla cache viene aggiunto un insieme di bit che costituiscono il campo **tag** (etichetta).

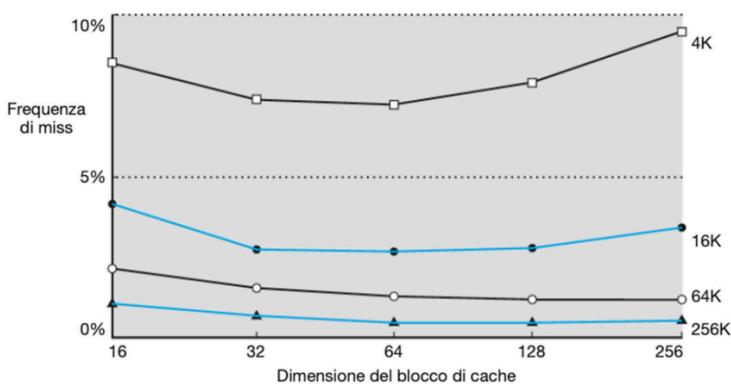
I tag contengono le informazioni necessarie a verificare se una parola della cache corrisponda o meno alla parola cercata. Un tag contiene solo la parte superiore dell'indirizzo della parola nella memoria principale, in particolare i bit che non vengono utilizzati come indice per individuare il blocco all'interno della cache.



È necessario anche disporre di un metodo per sapere quando un blocco della cache non contiene informazioni valide. Per esempio, quando un processore viene avviato, la cache è vuota e i numeri contenuti nei campi tag non hanno alcun significato. Per alcune locazioni questa condizione può persistere anche dopo aver eseguito molte istruzioni. Per sapere quando il tag associato a queste locazioni deve essere ignorato, aggiungiamo un **bit di validità** (valid bit) che, se non è impostato a 1, fa sì che la richiesta di lettura non possa avere successo.

### Prestazioni di una cache

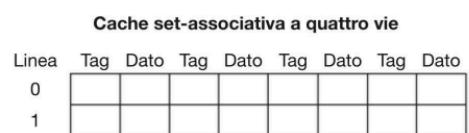
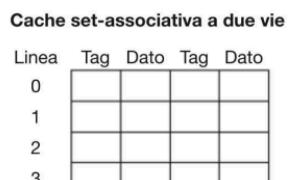
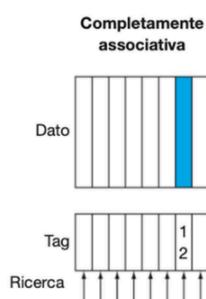
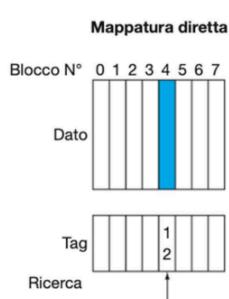
Aumentare la dimensione del blocco non è sempre una buona idea



### Cache associativa

Un blocco della memoria principale può essere scritto in una qualsiasi locazione della cache.

La ricerca va effettuata su tutti gli elementi della cache, svolta in parallelo attraverso un comparatore (aumento dei costi) ed essa è adatta per un numero ristretto di blocchi.



## Cache set-associativa

### Struttura della Cache Set-Associativa

- **Cache Set-Associativa a n Vie:** In questa configurazione, la cache è divisa in più linee (o set), ognuna delle quali contiene n blocchi. Ciò significa che un blocco di memoria principale può essere caricato in una delle n posizioni (vie) all'interno di una linea specifica della cache.
- **Mappatura del Blocco:** La mappatura del blocco nella cache set-associativa è determinata dal calcolo:  $(\text{Numero del blocco}) \% (\text{Numero delle Linee della Cache})$ . Una volta determinata la linea, il blocco può essere memorizzato in una qualsiasi delle n vie di quella linea.

### Vantaggi della Cache Set-Associativa

1. **Riduzione dei Conflitti:** Rispetto alla mappatura diretta, dove ogni blocco di memoria principale può essere mappato solo in una posizione specifica della cache, la cache set-associativa permette una maggiore flessibilità. Ciò riduce i conflitti di mappatura, dove due blocchi che competono per la stessa posizione nella cache possono causare una sostituzione frequente di dati.
2. **Ricerca Efficiente:** Pur non essendo completamente associativa, dove ogni blocco può essere in qualsiasi posizione della cache, la cache set-associativa richiede solo la ricerca all'interno di una linea specifica. Questo bilancia il tempo di ricerca con l'efficienza di utilizzo dello spazio.
3. **Compromesso Ottimale:** La cache set-associativa offre un buon compromesso tra la semplicità della mappatura diretta e la flessibilità della mappatura completamente associativa. Migliora le prestazioni riducendo i miss causati dai conflitti di mappatura.

### Esempio di Posizionamento nella Cache

Consideriamo una cache con otto blocchi in totale e supponiamo che abbiamo una cache set-associativa a 2 vie con 4 linee (ogni linea ha 2 blocchi). Supponiamo di dover mappare il blocco 12 della memoria principale.

1. **Determinazione della Linea:**  
Linea=12 mod 4=0  
Il blocco 12 sarà mappato nella linea 0.
2. **Posizionamento nella Linea:** Il blocco 12 può essere posizionato in uno qualsiasi dei 2 blocchi della linea 0

### Gestione della miss

La gestione della miss richiede un'unità di controllo separata che collabora con il processore. Questa si occupa dell'accesso alla memoria principale e del "rifornimento" della cache.

A ogni miss della cache possiamo mettere in stallo l'intero processore, essenzialmente bloccando il contenuto dei registri temporanei e di quelli visibili al programmatore, per tutto il tempo necessario a caricare i dati dalla memoria principale.

Se l'accesso ad un'istruzione si traduce in una miss, il contenuto del registro istruzioni non sarà più valido. Per caricare l'istruzione corretta nella cache dobbiamo poter dire al livello inferiore della gerarchia delle memorie di eseguire un'operazione di lettura. I passi sono:

1. Viene inviato il valore corretto del PC alla memoria;
2. Una volta costruito l'indirizzo corretto, si può chiedere alla memoria principale di effettuare la lettura dell'istruzione corrispondente e attendere che termini la lettura;
3. Scrivere la word proveniente dalla memoria nella posizione opportuna del blocco della cache, aggiornare il campo tag corrispondente scrivendovi i bit più significativi dell'indirizzo (presi dalla ALU) e impostare il bit di validità a 1;

4. Far ripartire l'esecuzione dell'istruzione dall'inizio, ripetendo la fase di fetch (che comporterà una hit).

Le operazioni di controllo svolte da una cache in lettura dei dati sono essenzialmente identiche

## Gestione della scrittura

Supponiamo che un'operazione di store scriva il dato solamente nella cache dei dati, senza modificare la memoria principale; al termine della scrittura, la memoria principale avrebbe un contenuto diverso da quello della cache. In questo caso si dice che la memoria e la cache sono *incoerenti*.

Per conservare la coerenza tra memoria e cache, si utilizza il **write-through**: si scrive sempre il dato in entrambe le memorie.

L'altro elemento fondamentale della scrittura è la gestione delle miss in scrittura. Occorre caricare dalla memoria principale le word appartenenti al blocco interessato. Dopo aver caricato il blocco e averlo scritto in cache, possiamo sovrascrivere la word del blocco che aveva causato la miss; questa word viene anche scritta nella memoria principale utilizzando il suo indirizzo completo.

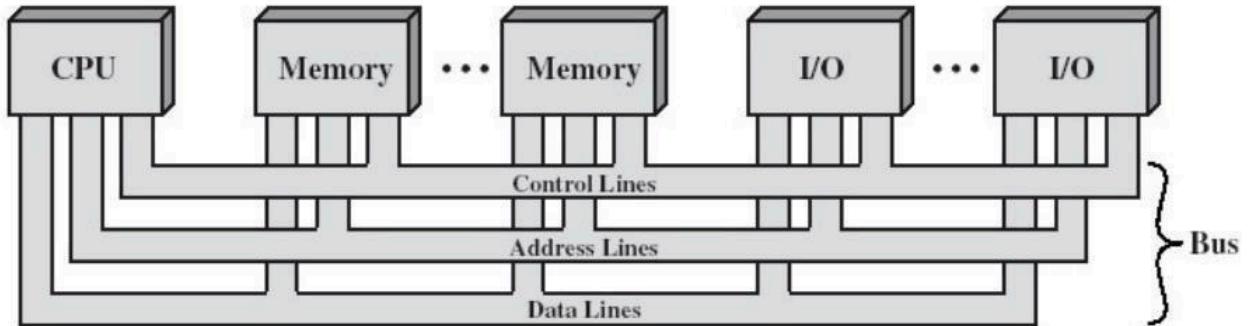
Il meccanismo descritto non offre buone prestazioni.

Una possibile soluzione consiste nell'usare una memoria tampone, detta **buffer di scrittura**. Il buffer di scrittura memorizza i dati in attesa che essi vengano scritti in memoria: dopo aver salvato il dato nella cache e nel buffer di scrittura, il processore può proseguire l'esecuzione.

Una volta completata la scrittura di un dato nella memoria principale, il corrispondente spazio nel buffer viene liberato. Se questo è pieno e il processore deve eseguire un'operazione di scrittura, questo viene messo in stallo finché non si libera spazio nel buffer.

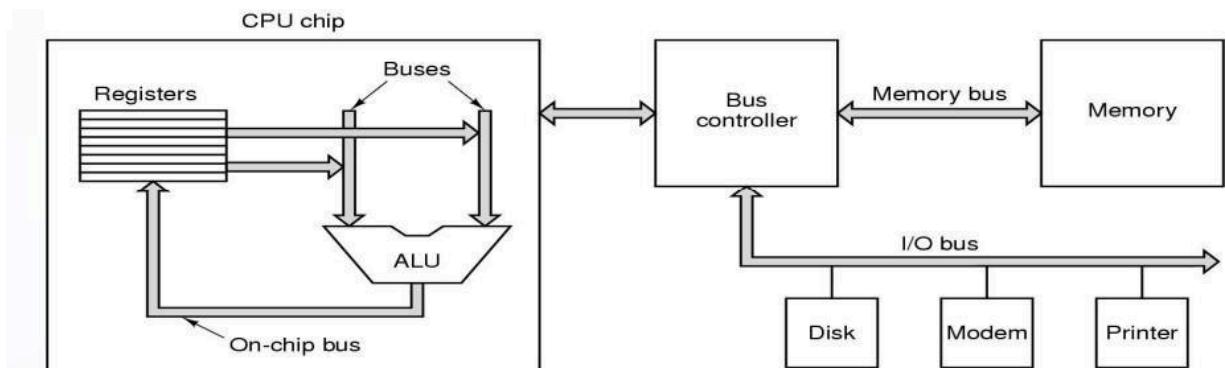
Lo schema alternativo al write-through è chiamato **write-back**: quando si verifica una scrittura, il dato viene scritto solamente nel blocco corrispondente della cache e il blocco modificato viene salvato nel livello inferiore della gerarchia solo quando deve essere rimpiazzato. Quest'ultimo schema può migliorare le prestazioni ma è più complesso da implementare.

## I Bus



Il bus è l'insieme di linee **elettriche** che collegano i moduli di un elaboratore attraverso il trasporto di bit. Ci sono vari tipi di bus, con dimensioni fisiche molto diverse:

- **Bus di sistema** - interconnette la CPU, le schede di I/O e la memoria;
- **Bus interni al chip** - mettono in comunicazione i moduli della CPU;
- **Bus SCSI** - connette le periferiche (può avere una estensione di qualche metro);
- **Portante fisico di una rete ETHERNET** - connette calcolatori.



Un bus è costituito da un fascio di **collegamenti** elettrici. In generale viene rappresentato mediante una freccia larga ad indicare che le linee in esso contenute hanno funzionalità distinte (controllo, indirizzo, dati). Affinché i moduli connessi dal bus siano in grado di comunicare, è necessario che essi interagiscano con il bus secondo un insieme di regole ben definite, detto **protocollo del bus**.

Quando ci sono collegamenti i bit non partono tutti nello stesso istante (fisicamente), mentre dal punto di vista logico devono partire ed arrivare insieme.

Le linee del bus possono essere:

- **Linee di dati (data bus)** - il numero di linee, che corrisponde alla **larghezza** del bus (nel RISC-V la larghezza è di 64 bit), determina il numero di bit che possono essere trasmessi alla volta (ha un impatto sulle prestazioni del sistema);
- **Linee di indirizzo** - permettono di individuare la sorgente/destinazione dei dati trasmessi sul data bus;
- **Linee di controllo** - controllano l'accesso e l'utilizzo delle linee di dati e di indirizzo (definiscono cosa i circuiti devono fare attraverso funzioni di controllo).

- Il bus indirizzi **rappresenta** tutte le connessioni tra CPU e memoria principale.
- Il bus di controllo è la **connessione** fisica tra la CPU e altri dispositivi con il computer. Trasporta le informazioni di controllo tra la CPU e altri dispositivi all'interno del computer.
- Il bus dati è una via di **comunicazione** attraverso la quale i dati possono viaggiare tra la CPU, la memoria e le periferiche del computer.

## Connessioni di una CPU

I processori **non** sono tutti uguali quindi anche il numero di bit trasportato dai bus è differente per ogni CPU.

### Indirizzo:

Il numero di bit del bus **indirizzi** può determinare la quantità di memoria che la CPU può indirizzare verso la memoria principale.

Ad  $n$  piedini corrispondono  $2^n$  locazioni di memoria indirizzabili (i valori tipici sono  $n = 16, 20, 32, 64$ ).

### Dati:

Il numero di fili determina la velocità di trasferimento dei dati. Ogni filo/linea trasmette un singolo bit alla volta.

Quindi  $n$  piedini permettono di leggere/scrivere una parola di  $n$  bit con una sola operazione (i valori tipici sono  $n = 8, 16, 32, 36, 64$ ).

### Controllo:

Le linee di controllo regolano il flusso e la scansione dei dati verso e dal chip. Una linea del bus viene utilizzata per indicare se la CPU sta attualmente leggendo o scrivendo nella memoria principale.

Categorie principali:

- Controllo del bus;
- Interrupt;
- Arbitraggio del bus;
- Varie

I dispositivi hanno connessioni diverse a seconda della loro natura.

I dispositivi collegati ad un bus si dividono in:

- **Attivi (master)** - possono decidere di iniziare un trasferimento, in genere sono collegati al bus per mezzo di un particolare chip, detto **bus driver**.
- **Passivi (slave)** - rimangono in attesa di richieste, in genere sono collegati per mezzo di un chip detto **bus receiver**.

Ci sono dispositivi che si comportano sia come master che come slave (ad esempio la CPU) e sono collegati attraverso un chip combinato, detto **bus transceiver**.

I problemi principali nella progettazione di un bus riguardano la larghezza (numero di linee), l'arbitraggio (come scegliere tra due **dispositivi** che vogliono diventare contemporaneamente arbitri dello stesso bus) e il funzionamento (come avviene il trasferimento dei bit).

## Larghezza del bus

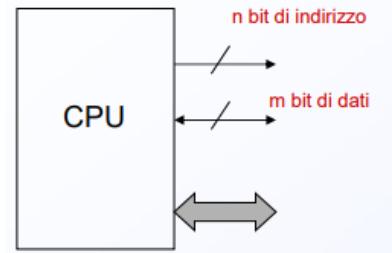
- Il numero delle linee utilizzate per trasferire gli **indirizzi** determinano la massima quantità della memoria **indirizzabile**.
- Il numero delle linee utilizzate per il trasferimento di **dati** determina la quantità di informazioni che è possibile trasferire con una singola **operazione**.

È possibile indirizzare  $2^n$  celle e trasferire  $m$  bit alla volta. Più un bus è **largo**, più la sua **capacità aumenta**.

Bus larghi sono più costosi di quelli stretti, ma offrono una *banda* più larga e quindi maggiore velocità di trasferimento.

Una maggiore velocità in generale si ottiene aumentando la larghezza del bus (più bit/trasferimento) o **diminuendo** il ciclo di bus (più trasferimenti/secondo).

Per ovviare ai problemi dati dai bus molto larghi, talvolta si opta per un **multiplexed bus**: le linee utilizzate per il trasferimento dei dati e degli indirizzi sono le stesse; prima si trasmettono gli indirizzi e poi i dati. Ovviamente questa soluzione è più lenta.



## Bus clocking

I bus si possono dividere in **due** categorie ben distinte.

Bus sincroni:

Hanno una linea pilotata da un oscillatore con una determinata **frequenza**; tutte le attività del bus richiedono un numero intero di questi cicli.

La durata delle fasi è **nota** ad **entrambi** i partecipanti (master e slave) e l'unica incognita è l'inizio della **comunicazione**.

Nella specifica di temporizzazione occorre tener conto di alcuni parametri temporali:

$T_{AD}$ : intervallo di tempo tra il fronte di salita del clock e l'istante in cui sono valide le linee degli indirizzi (max)

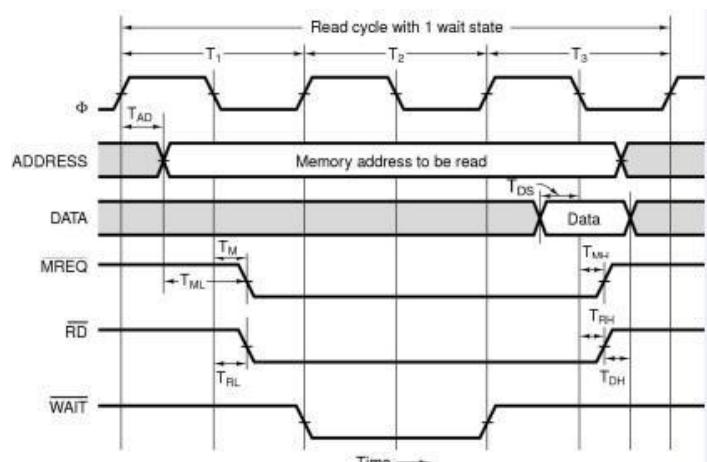
$T_{ML}$ : intervallo di tempo dall'istante in cui sono valide le linee degli indirizzi prima dell'effettiva segnalazione di MREQ (min)

$T_M/T_{RL}$ : intervallo di tempo tra il fronte di discesa del clock e il fronte di discesa di MREQ/RD (max)

$T_{DS}$ : tempo di setup per le linee dati prima del fronte di discesa del clock (min)

$T_{DH}$ : hold time tra il fronte di salita di MREQ e la rimozione da parte dello slave dei dati (min)

$T_{MH}/T_{RH}$ : ritardo di MREQ/RD dal fronte di discesa del clock (max)

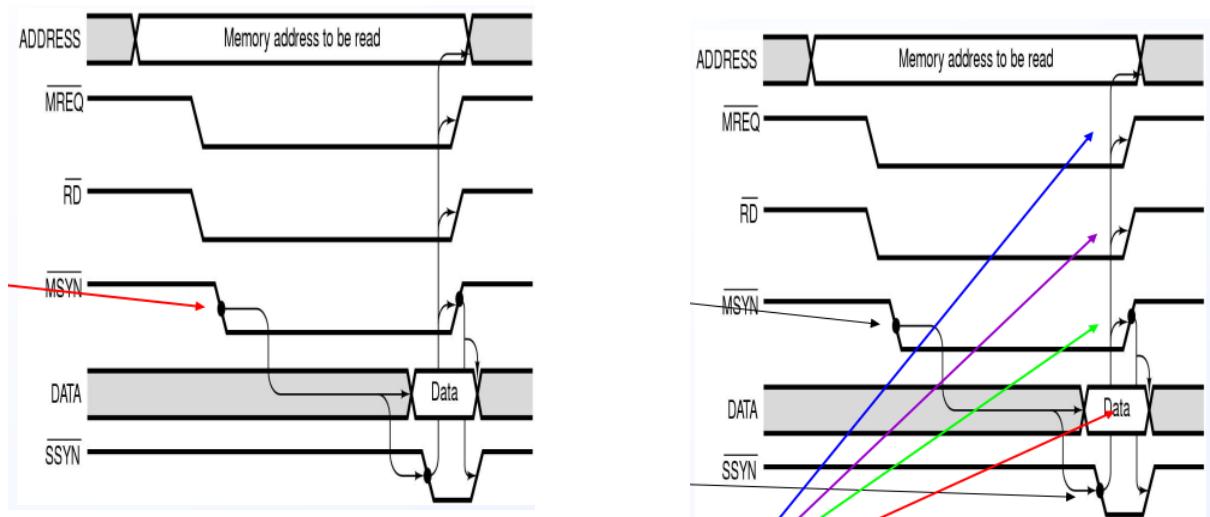


Bus asincroni:

Non hanno un clock principale; i cicli del bus possono essere *della lunghezza necessaria* e non devono essere uguali per tutti i dispositivi.

Il bus asincrono deve adattarsi al dispositivo più lento. Master e slave hanno dei segnali per sincronizzarsi.

- Attivati i segnali di accesso alla memoria e lettura, il bus asincrono attiva il segnale **Master SYNchronization**;
- Lo slave esegue il suo lavoro all'attivazione del segnale e quindi terminato lo attiva il segnale di **Slave SYNchronization**;
- I dati vengono memorizzati dal master che nega i segnali di accesso alla memoria, lettura e Master SYNchronization;
- Lo slave, dopo aver visto la negazione del segnale di master synchronization, nega a sua volta il segnale di Slave SYNchronization. → **Full HandShake**

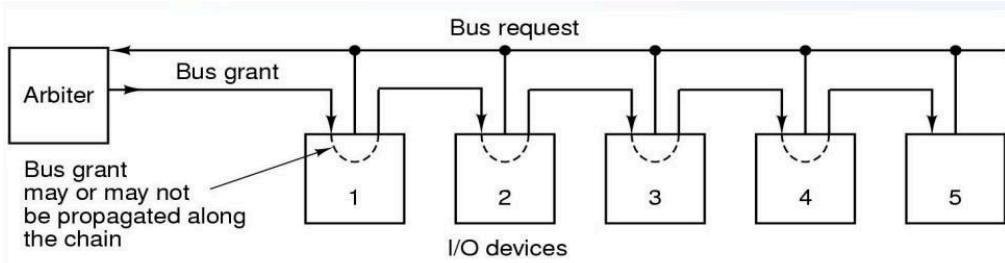


	Vantaggi	Svantaggi
Bus sincrono	<ul style="list-style-type: none"> <li>• Realizzazione slave semplice</li> <li>• Se la durata di un'operazione è fissa non occorre una linea di wait</li> </ul>	<ul style="list-style-type: none"> <li>• La durata di un'operazione di comunicazione non deve necessariamente avere una durata pari ad un numero intero di cicli</li> </ul>
Bus asincrono	<ul style="list-style-type: none"> <li>• Flessibilità; <ul style="list-style-type: none"> <li>◦ la durata di una operazione è determinata unicamente dalla velocità della coppia master/slave</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• Per completare una operazione di comunicazione sono sempre necessarie 4 azioni</li> <li>• Occorre inserire negli slave i circuiti necessari a rispondere opportunamente al protocollo</li> </ul>

*Cosa succede se più di un dispositivo richiede l'utilizzo del bus contemporaneamente?*

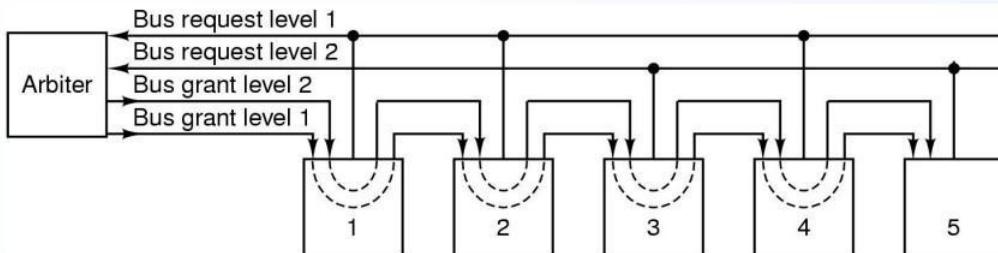
È necessario un meccanismo di **arbitraggio del bus** per evitare ambiguità delle informazioni immesse sul bus stesso. Sono possibili due strade: arbitraggio **centralizzato** oppure arbitraggio **decentralizzato**.

Il dispositivo che intende controllare il bus segnala un bus request all'arbitro di bus; il controllo viene eventualmente concesso tramite un **bus grant**. A valle del bus grant, il dispositivo (master) può iniziare la transazione.



Nel caso di **arbitraggio centralizzato**, c'è un arbitro che, quando "vede" una richiesta, attiva la linea di *grant* del bus.

Nel meccanismo del **Daisy chaining** il *grant* viene trasmesso lungo la linea bus grant finchè un dispositivo non accetta l'assegnamento; vince il dispositivo più vicino.

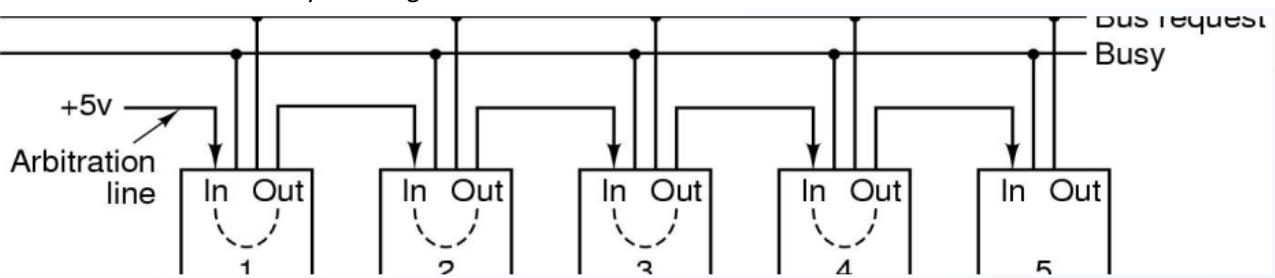


Possono esserci più livelli di priorità: l'assegnamento segue lo stesso meccanismo del daisy chaining, ma i dispositivi con priorità più alta hanno la precedenza nell'assegnamento del grant da parte dell'arbitro.

L'arbitraggio **decentralizzato** è più fault tolerant (nel caso di quello **centralizzato**, se non funziona l'arbitro, non funziona più niente).

Ci sono tante linee quanti sono i **dispositivi** connessi, e ognuno di questi osserva le linee prima di effettuare la richiesta (soluzione poco flessibile ma più economica).

Vediamo una soluzione con tre linee: *bus request*, *busy* e *linea di arbitraggio*. Quest'ultima descritta è più economica e veloce del daisy chaining centralizzato.



## Altri tipi di cicli di bus

### Trasferimento a blocchi

In un sistema di comunicazione master-slave, il **master** avvia il processo inviando allo slave il numero di parole richieste tramite una linea dati. Successivamente, il master attiva un segnale **BLOCK** su una linea dedicata per indicare l'inizio della **trasmissione**. Lo slave risponde trasmettendo una **parola per ogni ciclo** fino a quando tutte le parole richieste sono state inviate. Questa procedura è comune nelle operazioni di lettura dalla **cache**. Una volta ricevuto il numero totale di parole richieste, il master resetta il segnale **BLOCK** per terminare la comunicazione.

### Read-Modify-Write (test-and-set)

Si usa nel caso multiprocessore: più CPU collegate alla stessa memoria, questo ciclo impedisce ad altre CPU di usare il bus.

### Interrupt

Ciclo di bus per il segnale di interrupt. Segnale di **interrupt** viene inviato dalle periferiche di I/O quando hanno terminato un lavoro. Se più periferiche hanno terminato contemporaneamente vengono regolamentato da un arbitro (Controller Interrupt)

# I/O

## Tipi di Istruzioni a Livello ISA

L'architettura ISA (Instruction Set Architecture) supporta diversi tipi di istruzioni per la gestione dell'I/O, tra cui:

- **I/O programmato con busy waiting:** la CPU verifica ciclicamente lo stato dei dispositivi I/O, attendendo che siano pronti per eseguire operazioni. Questo metodo è semplice ma inefficiente, poiché la CPU rimane in attesa, consumando cicli di clock.
- **I/O controllato dall'interrupt:** i dispositivi I/O inviano segnali di interrupt alla CPU quando sono pronti per trasferire dati. Questo libera la CPU dal busy waiting, migliorando l'efficienza complessiva.
- **I/O in DMA (Direct Memory Access):** un controller DMA gestisce direttamente il trasferimento di dati tra memoria e dispositivi I/O, riducendo il carico sulla CPU.

## Comunicazione tra CPU e Moduli di I/O

La CPU comunica con i dispositivi I/O tramite **controllori** che trasformano i comandi della CPU in segnali elettrici per le periferiche e viceversa. Ogni controllore dispone di **registri** identificati da indirizzi che possono essere:

- **Memory mapped I/O:** gli indirizzi dei registri dei controllori fanno parte dell'insieme degli indirizzi di memoria.
- **Isolated I/O:** gli indirizzi sono dedicati esclusivamente ai registri dei controllori, separati dall'insieme degli indirizzi di memoria.

## Registri nei Controllori

I controllori dispongono di tre tipi di registri:

- **Dati:** contengono i dati di ingresso e uscita, ad esempio, i dati da stampare per una stampante.
- **Comandi:** ricevono comandi dalla CPU, come i comandi di stampa.
- **Stato:** forniscono informazioni sullo stato del dispositivo alla CPU, come l'avanzamento dei lavori di una stampante.

## I/O Programmato con Busy Waiting

In questo metodo, la CPU controlla continuamente lo stato del dispositivo I/O ispezionando il bit di stato del controllore, aspettando che segnali di essere pronto. Questo approccio è inefficiente poiché la CPU rimane in attesa, consumando risorse.

- **Lettura da tastiera:** il controllore della tastiera impone un bit di stato quando un tasto viene premuto, e la CPU continua a leggere questo bit finché non trova un valore di 1.
  - Il controllore della tastiera oltre ad impostare il bit Ready = 1, impone anche il codice ASCII del tasto cliccato nel registro buffer
  - La CPU legge il buffer e impone il bit Ready = 0
- **Scrittura su display:** la CPU attende che il display sia pronto (bit Ready = 1) prima di inviare un carattere al buffer del display.

## Interrupt

Gli interrupt permettono alla CPU di liberarsi dal **busy waiting**, segnalando quando un dispositivo ha completato il suo lavoro.

- **Interrupt hardware:** gli interrupt hardware interrompono il flusso di controllo del programma in corso, trasferendo il controllo a un gestore di interrupt che esegue le azioni appropriate. Dopo la gestione dell'interrupt, il controllo ritorna al programma interrotto.
- **Gestione degli interrupt:** include il salvataggio dei registri di stato, la lettura delle informazioni dal buffer del dispositivo, la gestione di eventuali **errori** e il ripristino dello stato del programma interrotto.

## Esempio

Consideriamo un esempio di scrittura sullo schermo di una riga di "count" caratteri puntata da "ptr", gestita con interruzione a livello del singolo carattere, dove il controller del dispositivo può visualizzare solo un carattere alla volta.

### Azioni Hardware (HW)

1. **Attivazione Interrupt:** Il controller attiva una linea di interrupt del bus di sistema per segnalare alla CPU che è pronto a gestire il carattere.
2. **Acknowledge della CPU:** Quando la CPU è pronta a gestire l'interruzione, invia un segnale di ack (acknowledge) sul bus di sistema.
3. **Invio del Vettore di Interrupt:** Il controller invia un intero sulle linee dati, chiamato vettore di interrupt.
4. **Prelievo del Vettore di Interrupt:** La CPU preleva il vettore di interrupt dal bus e lo salva.
5. **Salvataggio dello Stato:** La CPU salva il Program Counter (PC) e altri registri di stato sullo stack.
6. **Indice di Memoria:** Il vettore di interrupt viene usato come indice per trovare l'indirizzo di memoria del gestore dell'interruzione.
7. **Modifica del PC:** La CPU modifica il PC per puntare al gestore dell'interruzione e può evitare che altre interruzioni interrompano la gestione corrente.

### Azioni Software (SW)

1. **Salvataggio dei Registri:** Il codice ISA di gestione delle interruzioni salva tutti i registri necessari per preservare lo stato del programma interrotto.
2. **Lettura dal Buffer del Dispositivo:** Si leggono le informazioni necessarie dal buffer del dispositivo.
3. **Gestione degli Errori:** Se si verifica un errore di I/O, viene gestito opportunamente.
4. **Aggiornamento delle Variabili:** Le variabili "ptr" e "count" vengono aggiornate. Se ci sono altri caratteri da visualizzare, il carattere puntato da "ptr" viene copiato nel buffer del controller del dispositivo.
5. **Acknowledge di Fine Trattamento:** Si inviano eventuali ack di fine trattamento dell'interruzione.
6. **Ripristino dei Registri:** I registri salvati vengono ripristinati.
7. **Return from Interrupt (RETI):** Viene eseguita un'istruzione apposita di "return from interrupt" che ripristina la modalità e lo stato della CPU.

## Caratteristiche degli Interrupt

- **Asincroni:** Gli interrupt sono asincroni rispetto al programma in esecuzione, cioè possono verificarsi in qualsiasi momento.
- **Trasparenza:** Gli interrupt devono essere gestiti in modo trasparente, assicurando che lo stato dell'esecuzione dopo la gestione dell'interruzione ritorni esattamente come era prima dell'interruzione stesso.

## Priorità degli Interrupt

Quando più dispositivi generano interrupt simultaneamente, si può definire una priorità tra i dispositivi.

Il gestore delle interruzioni disabilita le interruzioni.

Ogni dispositivo può avere un livello di priorità e può generare interruzioni mascherabili o non mascherabili. Se la CPU non supporta più livelli di priorità, si può utilizzare un chip dedicato per la gestione degli interrupt.

## Trap ed Eccezioni

Le trap o eccezioni sono trasferimenti del flusso di controllo causati da condizioni eccezionali o da istruzioni che richiedono servizi di sistema operativo.

- **Gestione delle trap:** il gestore delle trap esegue azioni appropriate in risposta a condizioni come overflow o richieste di servizi di sistema operativo, modificando il flusso di controllo a una locazione prefissata.
  - Gestione dell'overflow senza trap:
    - l'hw setta un bit in un apposito registro e il programmatore ISA se lo desidera, dopo un'istruzione che può causare overflow, testa il registro e salta ad apposita procedura
  - Gestione con trap:
    - la condizione di overflow (rilevata da hw) genera una trap, che modifica il flusso di controllo ad una locazione prefissata

## DMA (Direct Memory Access)

Il DMA consente il trasferimento di dati tra memoria e dispositivi I/O senza il coinvolgimento diretto della CPU. Questo è gestito da un controller DMA, che si contende l'uso del bus con la CPU.

## Eccezioni nel Processore RISC-V

### Gestione delle Eccezioni

Le eccezioni devono essere gestite in modo da non alterare lo stato delle applicazioni. Il gestore delle eccezioni salva i **registri necessari**, esegue il codice in **modalità protetta** (kernel mode) e disabilita altre **eccezioni** durante la gestione di quella corrente..

Il programma in esecuzione deve essere sospeso e poi riattivato nel punto in cui si è verificata l'eccezione

### Interrupt (Interruzione)

- **Descrizione:** Eccezione causata da eventi esterni al programma in esecuzione.
- **Esempi:** Pressione di un tasto, movimento del mouse, ecc.
- **Caratteristiche:**
  - **Asincrona:** Gli interrupt sono asincroni rispetto all'esecuzione del programma, ovvero possono verificarsi in qualsiasi momento senza relazione diretta con le istruzioni del programma.
  - **Gestione:** Vengono gestiti tra le istruzioni del programma, interrompendo temporaneamente il flusso di esecuzione per gestire l'evento esterno.

### Errore

- **Descrizione:** Eccezione causata da eventi interni al programma.
- **Esempi:** Condizioni eccezionali come overflow, divisione per zero, ecc.
- **Caratteristiche:**
  - **Sincrona:** Gli errori sono sincroni rispetto all'esecuzione del programma, avvenendo in diretta conseguenza delle istruzioni eseguite.
  - **Gestione:** Devono essere gestiti immediatamente al verificarsi dell'errore per correggere o terminare l'esecuzione del programma.

## Environment Call (ecall)

- **Descrizione:** Eccezione sincrona causata da una richiesta di un servizio di sistema.
- **Esempi:** Richiesta di stampa di un messaggio, lettura di un intero, ecc.
- **Caratteristiche:**
  - **Gestione:** Il sistema operativo interviene per fornire il servizio richiesto, eseguendo operazioni come I/O o gestione delle risorse di sistema.

## Environment Break (ebreak)

- **Descrizione:** Eccezione sincrona utilizzata per scopi diagnostici o di debugging.

## Gestione delle Eccezioni

- **Metodi di Implementazione:**
  - **Salto Diretto:** Il flusso di controllo salta direttamente a un indirizzo specifico dove si trova la routine di gestione. Questo metodo è più veloce perché non necessita di prelevare l'indirizzo della routine di gestione.
    - **RISC-V:** Utilizza il registro speciale STVEC per memorizzare l'indirizzo base. La modalità di salto diretto è indicata dai due bit meno significativi di STVEC (STVEC.MODE) impostati a "00".
  - **Vettore di Interruzione:** Utilizza una tabella di indirizzi delle routine di gestione per ogni causa di eccezione.
    - **RISC-V e Altri Processori:** Il PC viene impostato a MEM [base + cause\*4]. La modalità a vettore di interruzione è indicata dai bit STVEC.MODE impostati a "01".

## Salvataggio dello Stato

- **Metodi di Salvataggio:**
  - **Stack (Push):** Salvataggio dei registri sullo stack.
    - **Processori:** Vax, 68k, 80x86 salvano l'intero set di registri.
    - **RISC-V, MIPS:** Salvano solo i registri necessari.
  - **Registri Ausiliari (Shadow Registers):** Utilizzati da M88k, ARM per evitare il sovraccarico di salvataggio sullo stack.
  - **Registri Speciali:** Utilizzati per memorizzare specifiche informazioni di stato.
    - **RISC-V, MIPS:** Registri come EPC (Program Counter), CAUSE, STATUS, TVAL (o BadVaddr) vengono utilizzati per gestire le eccezioni e ripristinare lo stato.

## Registri Eccezioni nel RISC-V

- SEPC
  - Indirizzo dell'istruzione colpevole
- SSTATUS
  - i bit di abilitazione globale degli interrupt
- SCAUSE
  - i bit 63 e [3:0] codificano le possibili sorgenti di eccezione
  - • 0 – Instruction address misaligned
  - • 2 – Illegal instruction
  - • 3 – Breakpoint • 4 – Load address misaligned • 5 – Load address fault • 6 – Store address misaligned • 7 – Store address fault • 8 – Environment call from U-mode • 9 – Environment call from S-mode • C – Instruction page fault
- STVAL
- SIP
- SIE

- STVEC
  - indirizzo base della lista dei «vettori di interrupt»
- SSCRATCH

## Riconoscimenti

Questi appunti sono stati originariamente creati e resi pubblici da **Elena Derosas**. Il contributo di Elena è stato fondamentale per la realizzazione di questo materiale, e la sua generosità nel condividere il suo lavoro è immensamente apprezzata. Puoi trovare la versione originale degli appunti al seguente link: [Appunti originali di Elena Derosas.](#)

La seconda versione di questi appunti è stata curata da **Paolo Dionesalvi**, che ha apportato modifiche e integrazioni per allineare il contenuto con il programma del nuovo ordinamento. Ogni sforzo è stato fatto per mantenere l'integrità delle informazioni originali, aggiungendo al contempo ulteriori dettagli e chiarimenti per migliorare la comprensione degli argomenti trattati.