Architettura Degli Elaboratori

Indice:

Lezione 1

26 settembre 2025

Il linguaggio dei computer è formato esclusivamente dai valori 0 e 1, perché i computer non comprendono direttamente i linguaggi di programmazione. I bit rappresentano questi valori binari, mentre un insieme di 8 bit costituisce un byte.

I computer utilizzano il sistema binario perché hanno solo due stati fisici: acceso (1) e spento (0). In una sequenza di bit, il numero più a sinistra è chiamato bit più significativo (MSB o MSD), mentre il numero più a destra è il bit meno significativo (LSB o LSD).

Quando si lavora con basi numeriche diverse da quella decimale, è necessario indicare la base del numero per evitare ambiguità.

Per convertire un numero decimale frazionario (cioè con la virgola) in binario, bisogna decidere in anticipo la precisione desiderata, cioè quante cifre binarie si vogliono considerare dopo la virgola. La procedura consiste nel moltiplicare il numero frazionario per 2:

- Se il numero prima della virgola è 1, lo si mette da parte come cifra binaria;
- Si azzera la parte intera e si moltiplica nuovamente la frazione per 2, ripetendo il processo fino a raggiungere la precisione desiderata.

Esempio:

$0{,}87 \times 2 = 1{,}74 \quad m_1 = 1$ (parte intera)
$0{,}74 \times 2 = 1{,}48 \quad m_2 = 1$
$0{,}48 \times 2 = 0{,}96 \quad m_3 = 0$
$0{,}96 \times 2 = 1{,}92 \quad m_4 = 1$
$0{,}92 \times 2 = 1{,}84 \quad m_5 = 1$
$0{,}84 \times 2 = 1{,}68 \quad m_6 = 1$
$0{,}68 \times 2 = 1{,}36 \quad m_7 = 1$
$m = (0{,}1101111)_2$

Per convertire un numero decimale in binario si usano due algoritmi diversi a seconda del tipo di numero:

- Se il numero è intero, si utilizza l'algoritmo del resto;
- Se il numero ha una parte frazionaria, si utilizza l'algoritmo della moltiplicazione.

I bit rappresentano le singole cifre di un numero binario.

I numeri ottali utilizzano le cifre da 0 a 7, mentre i numeri esadecimali utilizzano le cifre da 0 a 9, seguite dalle lettere A, B, C, D, E, F per rappresentare i valori da 10 a 15.

Per convertire un numero binario in ottale, si divide il numero binario in gruppi di 3 bit, partendo dalla virgola (per la parte frazionaria) o dall'unità (per la parte intera). Ogni gruppo di 3 bit viene poi trasformato in una cifra ottale corrispondente.

Esempio: $(1100111000{,}10111)_2$
1. $(001\ 100\ 111\ 000,\ 101\ 110)_2$
2. $(\ 1\quad 4\quad 7\quad 0,\quad 5\quad 6\ )_8$

Per convertire un numero binario in esadecimale, si divide il numero binario in gruppi di 4 bit, partendo dalla virgola (per la parte frazionaria) o dall'unità (per la parte intera). Ogni gruppo di 4 bit viene quindi trasformato in una cifra esadecimale, che può andare da 0 a 9 oppure da A a F, a seconda del valore del gruppo.

Esempio: $(1100111000{,}10111)_2$
1. $(0011\ 0011\ 1000,\ 1011\ 1000)_2$
2. $(\ 3\quad 3\quad 8\ ,\quad B\quad 8\ )_{16}$

Per convertire un numero ottale o esadecimale in binario, basta scomporre ogni cifra nel suo corrispondente valore binario.
Nel sistema ottale ogni cifra (da 0 a 7) viene rappresentata con 3 bit, mentre nel sistema esadecimale ogni cifra (da 0 a F) viene rappresentata con 4 bit.
Ad esempio, la cifra ottale 7 corrisponde a 111 in binario, mentre la cifra esadecimale A corrisponde a 1010 in binario.

Il numero più grande che posso rappresentare in binario dipende dal numero di bit che utilizzo. Infatti, il valore massimo si ottiene quando tutti i bit valgono 1.

Per esempio, con 4 bit il numero più grande che posso ottenere è 1111, che in decimale corrisponde a 15.

L'overflow si verifica quando il risultato di un'operazione non può essere rappresentato con i bit a disposizione.
Questo succede, per esempio, quando la somma di due numeri genera un risultato troppo grande per essere contenuto nel numero di bit usati.
Se ho 4 bit e il risultato ne richiede 5, allora si parla di overflow.
Un esempio pratico è la somma 1111 (cioè 15) più 0001 (cioè 1): il risultato è 1 0000, che occupa 5 bit, quindi si verifica overflow.

Quando invece il sottraendo è maggiore del minuendo, la differenza è negativa.
In questi casi, nel sistema binario si usa la rappresentazione in complemento a 2 per poter gestire correttamente i numeri negativi.

Per capire quanti bit servono a rappresentare un certo numero di valori, si utilizza il logaritmo in base 2.
In pratica, il numero di bit necessario si ottiene calcolando $\log_2$ del numero di valori da rappresentare, e poi arrotondando per eccesso se il risultato non è intero.
Ad esempio, se voglio rappresentare 10 numeri diversi (da 0 a 9), faccio $\log_2(10)$, che è circa 3,32.
Questo significa che mi servono 4 bit, perché con 3 bit posso rappresentare solo 8 valori (da 0 a 7), mentre con 4 bit posso arrivare a 16 valori (da 0 a 15).

Se invece conosco già quanti bit ho a disposizione e voglio sapere quanti numeri posso rappresentare, basta fare 2 elevato al numero di bit.
Ad esempio, con 5 bit posso rappresentare 2^5, cioè 32 valori, che vanno da 0 a 31

Lezione 2

3 ottobre 2025

Nella rappresentazione in complemento a 2, il bit più significativo (cioè il primo da sinistra) indica il segno del numero:

- se il bit più significativo è 0, il numero è positivo;
- se è 1, il numero è negativo.

Per trasformare un numero decimale negativo in binario con il complemento a 2, si parte dal suo valore assoluto, cioè dal numero positivo corrispondente, e si procede così:

1. Si scrive il numero in binario su un certo numero di bit (ad esempio 4, 8, ecc.).
2. Si invertono tutti i bit (cioè gli 0 diventano 1 e gli 1 diventano 0): questo passaggio si chiama complemento a 1.
3. Si aggiunge 1 al risultato ottenuto: questo è il complemento a 2 vero e proprio.

Ad esempio, se voglio rappresentare -3 su 4 bit, faccio:

$$2^4-3=16-3=13$$

e 13 in binario (su 4 bit) è 1101, che rappresenta proprio -3 in complemento a 2.

Un metodo pratico per fare più velocemente il complemento a 2 "a mano" è questo: partendo da destra, si lascia tutto invariato fino al primo 1 incluso, e poi si invertono tutti i bit rimanenti a

sinistra.
Questo trucco serve per evitare di dover ribaltare e sommare ogni volta.

Esempio:

$$(0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1100)_2 = 12_{10}$$
$$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 0100)_2 = -12_{10}$$

Infine, quando ci si trova davanti a una sequenza di 1 in complemento a 2, per semplificare i calcoli conviene iniziare a contare dal primo 1 partendo da destra, in modo da capire più facilmente il valore del numero rappresentato.

Esempio:

$$(1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1001)_2$$
$$= (-2^5 + 2^3 + 2^0)_{10} = (-32 + 8 + 1)_{10} = (-23)_{10}$$

Nel complemento a 2, le operazioni di somma e sottrazione si eseguono esattamente come per i numeri senza segno (cioè i numeri binari normali).
La differenza sta solo nell'interpretazione del risultato: il bit più significativo indica il segno del numero.

L'underflow si verifica quando il risultato di un'operazione produce un numero più piccolo (cioè più negativo) di quello che si può rappresentare con il numero di bit a disposizione.
Alcuni libri e persone, per semplicità, chiamano "overflow" anche questo tipo di errore, anche se tecnicamente si tratta di underflow.

Quando si somma un numero positivo con uno negativo, non può verificarsi overflow, perché il risultato rimane sempre all'interno dell'intervallo rappresentabile: infatti la somma di un valore positivo e di uno negativo tende sempre verso lo zero.

Nel complemento a 2, se durante una somma si genera un riporto finale oltre i bit disponibili, questo viene semplicemente ignorato.
Questo perché il riporto in eccesso non influisce sul risultato corretto nella rappresentazione in complemento a 2.

L'overflow invece può verificarsi solo quando si sommano due numeri dello stesso segno, cioè due positivi o due negativi.
Per capire se si è verificato overflow, si osserva il segno del risultato:

- Se sommo due numeri positivi e ottengo un numero negativo, significa che c'è stato overflow.
- Allo stesso modo, se sommo due numeri negativi e il risultato è positivo, anche in questo caso si è verificato overflow.

Esempio:

```
  1
  0111 +      Overflow perché segni diversi!
  0100 =
 ------
  1011        (7+4)10 = 1110 (overflow perché>7)

 1
  1000 +      Overflow perché segni diversi!
  1000 =
 ------
 X0000        (-8-8)10 = -1610(overflow perché<8)
```

Nel complemento a 2, per verificare se si è verificato un overflow, si osservano i bit più significativi (cioè quelli di segno) dei due numeri che stiamo sommando e del risultato.
Se i bit più significativi dei due addendi sono uguali tra loro e uguali anche a quello del risultato, allora non si è verificato overflow.

Esempio:

```
  0010 +      Corretto perché segni uguali!
  0001 =
 ------
  0011        (2+1)10 = 310

 11
  1110 +      Corretto perché segni uguali!
  1100 =
 ------
 X1010        (-2-4)10 = -610
```

In caso contrario, se i due addendi hanno lo stesso segno ma il risultato ha segno diverso, significa che c'è stato overflow.

Quando lavoro con i numeri in virgola mobile e sposto la virgola in un numero binario, devo sempre tenere conto di quante posizioni l'ho spostata.
Ogni spostamento della virgola, infatti, corrisponde a una potenza di 2, perché nel sistema binario ogni cifra rappresenta una potenza di 2.

In pratica:

- Se sposto la virgola verso sinistra, sto rendendo il numero più piccolo, quindi devo moltiplicare per 2 elevato al numero di posizioni in cui ho spostato la virgola (l'esponente diventa positivo).
- Se sposto la virgola verso destra, sto rendendo il numero più grande, quindi devo moltiplicare per 2 elevato al numero di posizioni, ma con esponente negativo.

Esempio:

$$-10110{,}101 = -1{,}0110101 \times 2^{4}$$
$$+0{,}001101 = +1{,}101 \times 2^{-3}$$

Con i numeri reali (cioè quelli con la virgola) non si può usare il complemento a 2 per rappresentarli.
Il complemento a 2, infatti, è un metodo pensato solo per i numeri interi, perché funziona

basandosi su un numero fissato di bit e su un meccanismo che permette di ottenere automaticamente il valore negativo di un numero intero.

Quando però entrano in gioco i numeri reali (come 3.75, −2.5, 0.125) il complemento a 2 non è più adatto, perché:

1. I numeri reali hanno una parte frazionaria, che richiede un modo diverso di essere memorizzata in binario.
2. Abbiamo bisogno di rappresentare valori molto grandi e molto piccoli, quindi serve un sistema più flessibile.
3. Il complemento a 2 non permette di gestire bene precisione, segno ed esponente.

Per questo motivo gli architettori dei computer usano uno standard apposito, chiamato IEEE 754, dove ogni numero reale è diviso in tre parti:

– Segno
– Esponente (in “excess bias”)
– Mantissa (o frazione)

Questo tipo di rappresentazione permette di trattare numeri reali in modo preciso ed efficiente, cosa che il complemento a 2 non può fare.

La virgola mobile si può rappresentare con la formula:

$$(-1)^S \times (1,M)_2 \times 2^y$$

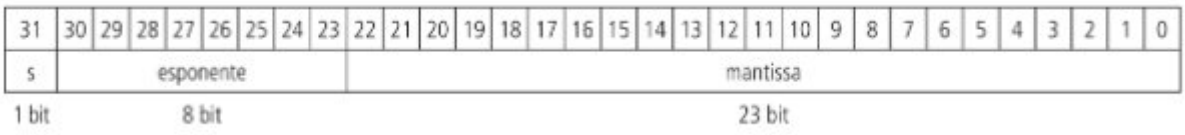
All’interno della CPU, i numeri reali possono essere rappresentati con due diversi livelli di precisione: la singola precisione (float), che utilizza 32 bit, e la doppia precisione (double), che utilizza 64 bit.

Nel formato a singola precisione, i 32 bit sono suddivisi in tre parti:

– 1 bit è dedicato al segno del numero (0 per i positivi, 1 per i negativi);
– 8 bit rappresentano l’esponente, che serve a determinare l’ordine di grandezza del numero;
– 23 bit rappresentano la mantissa, cioè la parte frazionaria del numero, quella che viene dopo la virgola.

In questo modo, il computer riesce a rappresentare un’ampia gamma di numeri reali, sia molto grandi che molto piccoli, con una certa precisione, che nel caso del tipo float è limitata ai 32 bit disponibili.

Esempio:



Nella rappresentazione in doppia precisione, i primi 20 bit vengono utilizzati per rappresentare la parte frazionaria del numero (quella dopo la virgola), mentre 11 bit servono per memorizzare l’esponente. L’ultimo bit, infine, indica il segno del numero, cioè se è positivo o negativo.

Esempio:

Esempio:

- Rappresentare in virgola mobile -37,75
- Conversione in binario di 37,75:
  100101,11
- Forma normalizzata: $-1{,}0010111 \times 2^5$
- M = 0010111
- y = 101 (5)
- S=1

Lo standard IEEE 754 è il metodo usato dai computer per rappresentare i numeri in virgola mobile, cioè quei numeri che possono avere sia una parte intera sia una parte decimale (come 3.75, −1.2, 0.125…).

Per memorizzare questi numeri, lo standard divide il valore in tre parti:

- bit di segno (0 per positivo, 1 per negativo)
- esponente
- mantissa (o frazione)

Poiché l'esponente può essere positivo o negativo, lo standard non utilizza il complemento a 2, ma una codifica polarizzata ("bias"), che permette di evitare numeri negativi all'interno dei bit dell'esponente.

Nella rappresentazione a 32 bit, l'esponente è codificato usando un bias di 127. Questo significa che:

- per avere sempre un valore non negativo nei bit dell'esponente,
- il computer aggiunge 127 all'esponente reale.

Esempio:

$$(-1)^S \times (1,M) \times 2^{E-127}$$

$$y = E-127 \text{ ovvero } E = 127+y$$

Nella rappresentazione a 64 bit, lo standard usa un bias più grande: 1023.

Esempio:

$$(-1)^S \times (1,M) \times 2^{E-1023}$$

$$y = E-1023 \text{ ovvero } E = 1023+y$$

Quando devo trasformare in decimale la parte frazionaria di un numero binario (cioè la parte dopo la virgola), devo utilizzare potenze negative di 2, perché ogni posizione dopo la virgola rappresenta una frazione sempre più piccola.

Lezione 3

10 ottobre 2025

Il calcolatore ha una CPU (Central Processing Unit), è formato da due parti principali, comunica con la memoria e dispone di un ingresso e un'uscita per scambiare dati con l'esterno. Ha anche una memoria, che contiene due tipi fondamentali di informazioni:
– i programmi che verranno eseguiti dalla CPU;
– i dati necessari per far funzionare correttamente i programmi.

La CPU è composta da due sezioni:
– la Control Unit (Unità di Controllo), che coordina e controlla tutte le operazioni del processore. Interpreta le istruzioni e decide cosa deve fare la CPU in ogni momento, inviando i comandi ai vari componenti come l'ALU e i registri;
– il Datapath, cioè la parte dove vengono svolte concretamente le operazioni. Comprende l'ALU (Arithmetic Logic Unit), i registri e tutti i circuiti necessari per eseguire i calcoli e spostare i dati durante l'esecuzione del programma.

Gli algoritmi servono a descrivere, passo dopo passo, come risolvere un determinato problema. Un algoritmo può essere scritto anche in linguaggio umano, non per forza in linguaggio macchina. Per far comprendere il funzionamento dell'algoritmo all'esecutore (cioè al computer) bisogna però scriverlo in un linguaggio di programmazione.
Sia l'algoritmo che il linguaggio di programmazione sono considerati strumenti ad alto livello, quindi comprensibili da una persona.

Il sistema operativo è un programma diverso dalle applicazioni comuni, perché agisce come controllore del sistema: gestisce le risorse del computer (CPU, memoria e dispositivi di input/output) e decide come e quando assegnarle ai vari programmi che sono in esecuzione.

Nella parte più bassa del calcolatore troviamo i transistor, che sono i componenti fondamentali dell'hardware. Da essi vengono realizzate le porte logiche, cioè i circuiti base che permettono di eseguire operazioni logiche e aritmetiche.

Per progettare un hardware si seguono diversi passaggi:

1. definire cosa deve fare il circuito (la sua funzione);
2. descrivere il comportamento tramite l'algebra booleana, che permette di esprimere le operazioni logiche;
3. costruire il circuito logico corrispondente e cercare di ottimizzarlo riducendo la complessità e il numero di porte;
4. effettuare la mappatura tecnologica, cioè tradurre l'algebra booleana in componenti fisici (porte logiche, transistor, ecc.);

5. verificare che il circuito funzioni correttamente.

Le porte logiche sono strumenti sia matematici che circuitali, che servono a descrivere e comprendere il funzionamento dei circuiti digitali. Nella logica binaria esistono solo due valori, 0 e 1, e abbiamo tre operatori principali: AND, OR e NOT.

La logica binaria è molto simile all'aritmetica binaria: l'operazione AND assomiglia alla moltiplicazione, mentre l'OR assomiglia alla somma. Per questo il simbolo dell'AND è spesso la moltiplicazione, e quello dell'OR è l'addizione.

NOT

X	$Z = \overline{X}$
0	1
1	0

L'AND funziona così:

AND

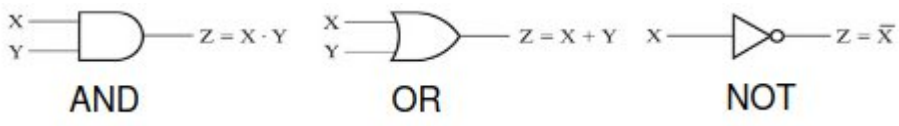
X	Y	$Z = X \cdot Y$
0	0	0
0	1	0
1	0	0
1	1	1

L'OR funziona così:

OR

X	Y	$Z = X + Y$
0	0	0
0	1	1
1	0	1
1	1	1

Le porte logiche hanno i seguenti simboli:



Lo XOR funziona così:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Utilizziamo le identità dell'algebra booleana per ottenere il circuito logico migliore, cioè un circuito più semplice, più veloce nell'eseguire i calcoli e che richiede un numero minore possibile di porte logiche.
L'idea è che, semplificando l'espressione booleana, si riduce anche la complessità del circuito che verrà realizzato.

Per misurare il costo di un circuito si considerano due tipi di costo fondamentali:

- Il numero di termini, cioè il numero di porte AND presenti nell'espressione (ogni termine AND rappresenta un pezzo del circuito).
- Il numero di letterali, cioè quante variabili, o variabili negate, compaiono nell'espressione.

Un letterale è semplicemente una variabile (come A, B, C…) oppure la variabile negata (A', B', C'…).
Più letterali ci sono, più il circuito risulta complesso da realizzare.

Esempio di letterale:

F = XYZ, i letterali sono X, Y, Z

Esempio di termini:

F = XYZ + XZ, i termini sono XYZ e XZ

Lezione 4

17 ottobre 2025

Una funzione booleana può essere rappresentata tramite una tabella di verità, dove a ogni combinazione possibile di input corrisponde un unico valore di output (vero o falso, cioè 1 o 0).
Per esprimere matematicamente una funzione booleana si utilizzano delle strutture standard chiamate forme canoniche, che permettono di scrivere la funzione in modo ordinato e univoco.

Le principali forme canoniche sono:
– Somma di prodotti (forma canonica disgiuntiva)
– Prodotto di somme (forma canonica congiuntiva)

Queste forme sono state scelte perché rendono più semplice la semplificazione delle funzioni logiche e permettono di ottenere circuiti molto più piccoli ed efficienti.

Esistono due tipi di forme canoniche:
– SOP (Sum Of Products) = somma di prodotti di letterali
– POS (Product Of Sums) = prodotto di somme di letterali

Il termine mintermini si riferisce alla forma SOP, mentre i maxtermini sono utilizzati per la forma POS.

Un mintermine è un prodotto (AND) che include tutte le n variabili della funzione, ognuna presente come letterale, cioè nella forma diretta o negata.
Un maxtermine è una somma (OR) che include tutte le n variabili della funzione, sempre come letterali diretti o negati.

Nella forma SOP (Sum of Products), le variabili che valgono 0 nella combinazione vengono negate, mentre quelle che valgono 1 restano nella forma diretta.
Nella forma POS (Product of Sums) accade l'opposto: le variabili che valgono 0 non si negano, mentre quelle che valgono 1 vengono negate.

Per ottenere la forma POS, si prendono dalla tabella di verità tutte le combinazioni in cui f = 0.
Per ottenere la forma SOP, si prendono tutte le combinazioni in cui f = 1.

Con 3 variabili, si devono raggruppare gli 1 adiacenti che hanno in comune alcune variabili

Con 4 variabili, si raggruppano gli 1 in blocchi di dimensione pari a potenze di 2 (1, 2, 4, 8, …), cercando di evitare 1 isolati.

Nelle mappe di Karnaugh è possibile usare l'effetto "Pac-Man", cioè considerare gli estremi della mappa come se fossero adiacenti: si può "uscire" da un lato e "rientrare" dall'altro per formare gruppi più grandi e semplificare meglio la funzione.

Lezione 5

24 ottobre 2025

Un implicante è un gruppo di celle con valore 1 in una K-map (mappa di Karnaugh) che sono adiacenti tra loro. Quando raggruppi questi 1 adiacenti, ottieni un implicante che semplifica l'espressione booleana

Un implicante primo è un implicante che non può essere ulteriormente semplificato. In altre parole, un implicante primo è il gruppo più grande possibile di celle con valore 1 che non può essere esteso senza perdere la sua capacità di coprire i mintermini della funzione booleana.

Un implicante primo essenziale è un implicante primo che copre almeno una cella che non può essere coperta da nessun altro implicante. In altre parole, un implicante primo essenziale è un implicante che contiene un mintermine unico (cioè una cella 1 che non può essere coperta da altri gruppi).

Gli implicanti con don't care (DC) sono le celle X che trovi nelle K-map. Quando hai tre 1 adiacenti e una X adiacente, puoi trasformare quella X in 1 per formare un gruppo di 4 celle, ma non puoi mai fare un gruppo composto solo da X. Le X (don't care) ti permettono di semplificare i gruppi, ma un implicante deve sempre coprire almeno una cella con 1. Le celle X possono essere trattate come 1 per estendere i gruppi, ma non possono mai essere l'unico elemento di un implicante.

Lezione 6

31 ottobre 2025

Il decoder è un circuito combinatorio che esegue una decodifica, cioè trasforma un codice binario compatto in una versione meno compatta e più estesa.
In pratica, a partire da un numero binario in ingresso, il decoder attiva una sola uscita, precisamente quella corrispondente al valore dell'ingresso.

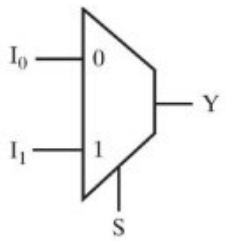
Esempio:
Se l'ingresso è 011 (che equivale al numero 3 in decimale), il decoder attiva solo l'uscita 3, mentre tutte le altre rimangono a 0.
In sintesi: parte da un codice compatto e lo espande.

L'encoder è l'opposto del decoder. Serve a rappresentare l'informazione in modo più compatto: prende più linee di ingresso e le codifica in un numero binario più corto.
In altre parole: parte da molti segnali e li trasforma in un codice compatto, cioè fa l'operazione inversa della decodifica.

Il multiplexer (o MUX) è un circuito combinatorio che seleziona quale ingresso deve essere mandato in uscita, in base al valore di un segnale di selezione (S).
È paragonabile a un'istruzione if/else in un linguaggio di programmazione.

Esempio:

- Se S = 0, in uscita passa l'ingresso I0.
- Se S = 1, in uscita passa l'ingresso I1.



In generale, un multiplexer con n bit di selezione può sceglieretra 2^n ingressi. In pratica, il numero di bit di selezione determina quante linee di ingresso il multiplexer può gestire: più bit di selezione ci sono, maggiore è il numero di ingressi selezionabili.

Un half adder è un circuito combinatorio che permette di sommare due bit. Il sommatore logico, sia nella versione half adder che full adder, ha due uscite principali: Sum e Carry.

- Sum rappresenta il risultato della somma dei bit di ingresso, ed è calcolato tramite l'operatore XOR:

$$S = \overline{X}Y + X\overline{Y} = (X \oplus Y)$$

(vale 1 solo se uno dei due ingressi è 1, ma non entrambi)

- Carry rappresenta il riporto della somma (cioè il "resto" che va alla posizione successiva) ed è calcolato con un AND:

$$C = XY$$

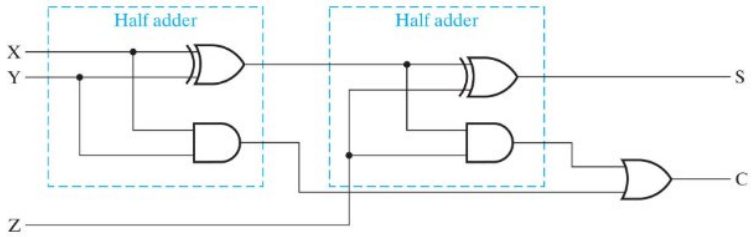
(vale 1 solo se entrambi gli ingressi sono 1)

Quando nella somma dei bit compare un riporto, diventa necessario sommare tre bit contemporaneamente: i due bit di ingresso e il riporto proveniente dalla somma precedente. Per gestire questa situazione si utilizza un circuito combinatorio chiamato full adder.

Il full adder può essere concettualmente rappresentato come due half adder collegati in cascata. In pratica, il primo half adder somma due dei bit di ingresso producendo una somma parziale e un riporto; il secondo half adder somma questa somma parziale con il terzo bit (ad esempio il riporto precedente), generando così il risultato finale della somma e un nuovo riporto.

In questo modo, i full adder possono essere concatenati per sommare sequenze di bit più lunghe, gestendo correttamente tutti i riporti intermedi.

Esempio:



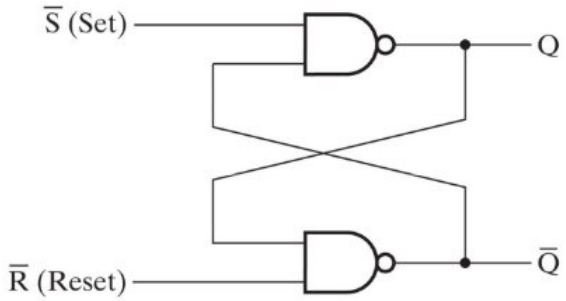
I circuiti sequenziali sono circuiti logici particolari in cui una parte dell'uscita viene riportata come ingresso. Questa caratteristica permette al circuito di ricordare informazioni sullo stato precedente, ed è per questo che si dice che i circuiti sequenziali abbiano memoria. Il valore memorizzato in un dato momento è chiamato stato del circuito.

I circuiti sequenziali si dividono principalmente in due categorie:

1. Sincroni:
    - Evolvono nel tempo discreto, scandito da un segnale di clock comune.
    - Il clock coordina tutte le operazioni, rendendo il comportamento del circuito prevedibile e ordinato.
1. Asincroni:
    - Non utilizzano un clock; la loro evoluzione dipende direttamente dagli ingressi e avviene in tempo continuo.
    - Sono generalmente più veloci, ma anche più complessi da progettare, poiché sono più sensibili ai ritardi dei segnali.

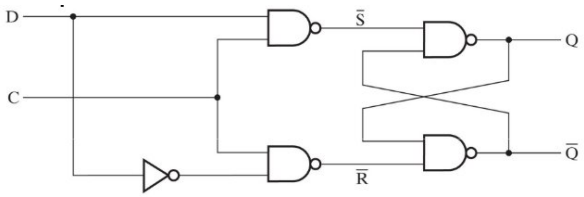
Un latch è un blocco di memoria elementare usato per memorizzare un singolo bit. In un latch è possibile sia scrivere che leggere il valore memorizzato.

Il latch Set-Reset (SR latch) è un tipo di latch di base, costruito collegando in modo incrociato due porte NAND o NOR. Serve per impostare (Set) o azzerare (Reset) il bit memorizzato, permettendo così di controllare il valore della memoria.



Nel latch SR, esiste una combinazione vietata, che si verifica quando S = 1 e R = 1. In questa condizione entrambe le uscite, Q e $\overline{Q}$, diventano 0, violando la regola fondamentale secondo cui $\overline{Q}$ deve essere sempre il complemento di Q. In pratica, quando S e R sono entrambi uguali a 1, il circuito entra in uno stato instabile e non è possibile prevedere quale valore manterrà dopo.

Per ovviare a questo problema, esiste il latch D (detto anche data latch o delay latch), che è una versione semplificata del latch SR. Il latch D serve per memorizzare un singolo bit (0 o 1) in modo stabile, evitando le combinazioni vietate e rendendo più semplice il controllo della memoria.



Il latch D ha un solo ingresso, chiamato D, e un segnale di controllo chiamato Enable (o talvolta Clock). Questo tipo di latch risolve il problema delle combinazioni non valide presente nel latch SR.

Il funzionamento del latch D dipende dal segnale Enable:

- Quando Enable = 1, il latch legge il valore presente sull'ingresso D e lo porta in uscita (Q); in pratica, l'uscita segue l'ingresso.
- Quando Enable = 0, il latch mantiene il valore precedente in memoria, indipendentemente dall'ingresso D.

In questo modo, il latch D permette di memorizzare un singolo bit in maniera stabile e controllata, evitando condizioni instabili.

Lezione 8

14 novembre 2025

In assembly MIPS, per confrontare due registri si utilizzano comandi specifici:

- beq (branch if equal) serve per verificare se due registri sono uguali; se lo sono, il salto viene eseguito.
- bne (branch if not equal) serve per verificare se due registri non sono uguali; in questo caso il salto viene eseguito solo se i registri differiscono.
- j (jump) esegue un salto incondizionato, indipendentemente dai valori dei registri.

I comandi beq e bne corrispondono al concetto di go to presente nei linguaggi ad alto livello. In assembly, infatti, non esistono cicli o istruzioni di iterazione come in altri linguaggi: tutto viene gestito tramite salti condizionati e incondizionati.

Durante l'esame, ogni riga di codice va commentata con il simbolo #, indicando il significato dell'istruzione.

I registri MIPS hanno alcune convenzioni binarie:

- Da $t0 a $t7 iniziano con 01 in binario.
- Da $s0 a $s7 iniziano con 10 in binario.
- Da $t8 a $t9 iniziano con 11 in binario.

Le istruzioni MIPS si dividono in tre formati principali: R, I e J:

- Le istruzioni di formato R sono aritmetiche, logiche e di scorrimento; non contengono un indirizzo.
- Le istruzioni di formato I includono oltre alle istruzioni R anche salti condizionati e trasferimento dati; contengono un indirizzo o un immediato.
- Le istruzioni di formato J servono solo per salti incondizionati e contengono solo un indirizzo.

Nel formato R, i campi principali sono:

- op: identifica la classe dell'istruzione.
- rs: registro sorgente contenente il primo operando.
- rt: registro sorgente contenente il secondo operando.
- rd: registro destinazione dove viene salvato il risultato.
- shamt: campo usato per le operazioni di scorrimento.
- funct: indica la variante specifica dell'operazione.

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Esempio:

	6 bit	5 bit	5 bit	5 bit	5 bit	6 bit
	op	**rs**	**rt**	**rd**	**shamt**	**func**
`add $s1,$s2,$s3`	000000	10010	10011	10001	00000	100000

Il formato R lo posso convertire in esadecimale

Esempio:

```
add $s1,$s2,$s3
0000 0010 0101 0011 1000 1000 0010 0000
 0    2    5    3    8    8    2    0
```

In esadecimale: 0x02538820

Nel formato I ho i seguenti campi:

- Op che mi serve per identificare il tipo di istruzione
- Rs che è il registro base
- Rt che è il registro di destinazione
- Indirizzo che può essere l'offset o una costate o l'etichetta di un salto

op	rs	rt	indirizzo/costante
6 bit	5 bit	5 bit	16 bit

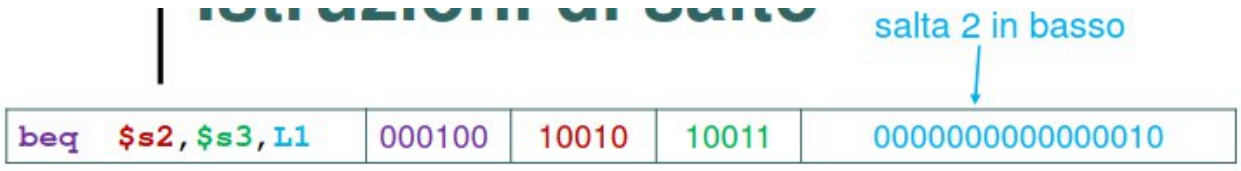
Esempio:

	6 bit	5 bit	5 bit	16 bit
	op	**rs**	**rt**	**Indirizzo/costante**
`addi $s2,$s3,-29`	001000	10011	10010	1111111111100011

Nel campo indirizzo delle istruzioni MIPS, non vengono memorizzati solo valori numerici o offset, ma anche le istruzioni di salto. In pratica, questo campo può contenere un indirizzo assoluto, un offset relativo o un'etichetta che indica la destinazione di un salto condizionato o incondizionato.

Questo permette alle istruzioni di salto di utilizzare il campo indirizzo per determinare dove dirigere il flusso del programma, rendendo possibile sia il salto a brevi distanze (tramite offset) sia a indirizzi più lontani (tramite etichette o registri).

Esempio:



Per determinare a quale indirizzo assoluto salta un'istruzione MIPS, bisogna moltiplicare il valore presente nel campo indirizzo per 4. Questo avviene perché gli indirizzi delle istruzioni in MIPS sono allineati a 4 byte (ogni istruzione occupa 4 byte di memoria), quindi il campo indirizzo memorizza in pratica il numero di parole, non di byte.

In questo modo, moltiplicando per 4 otteniamo l'indirizzo in byte, corrispondente alla destinazione effettiva del salto.

Esempio:

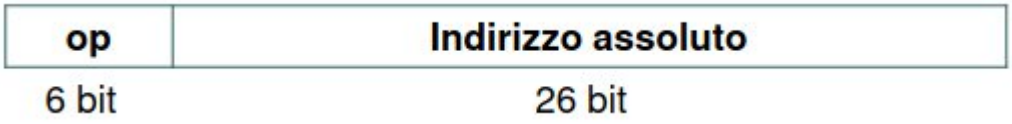
L1×4= 40016 + 2×4= 40024

Nel formato J delle istruzioni MIPS sono presenti solo due campi principali:

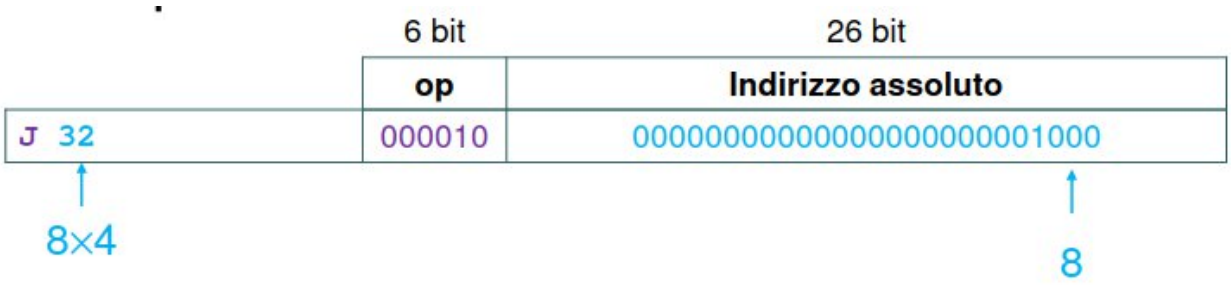
- op: serve a identificare il tipo di istruzione, indicando che si tratta di un salto incondizionato.
- indirizzo: contiene l'indirizzo assoluto della destinazione del salto, espresso come indirizzo di parola.

Poiché ogni istruzione MIPS occupa 4 byte, per ottenere l'indirizzo effettivo in memoria bisogna moltiplicare il valore presente nel campo indirizzo per 4.

Il formato J è quindi utilizzato esclusivamente per i salti incondizionati e permette di raggiungere qualsiasi istruzione tramite l'indirizzo calcolato.



Esempio:



Quando è necessario saltare a indirizzi superiori a 2^28 byte, non è possibile usare le normali istruzioni di salto del formato J, perché il campo indirizzo in queste istruzioni è troppo piccolo per rappresentare valori così grandi. In questi casi si utilizza l'istruzione jr rs (jump register).

Con jr rs, l'indirizzo di destinazione del salto non è codificato direttamente nell'istruzione, ma è contenuto nel registro rs. Questo permette di saltare a qualsiasi indirizzo di memoria accessibile, superando il limite imposto dal formato J.