

Emiddio Ingenito - Architettura Degli Elaboratori I

Elaboratore ed elaborazione:

Macchina di Turing:

Architettura di un'elaboratore

Modello di Von Neumann:

Ciclo esecuzione codice:

MIPS

Codifica dell'informazione:

Esempio:

Sistema di numerazione:

Numeri naturali:

Codice di Grey:

Codifica degli interi Z

Numeri reali:

Estensione della notazione posizionale, per la rappresentazione di una parte decimale.

Da base 10 a base 2.

Ripartizione dei bit tra parte intera e frazionaria:

Standard IEEE 754

Numeri normalizzati (queste regole valgono solo per i normalizzati)

Numeri de-normalizzati

Circuiti digitali:

Algebra di Boole:

Operazioni logiche:

Precedenza degli operatori:

Principio di dualità:

Proprietà degli operatori logici:

Dimostrazione di alcune proprietà

Applicazione delle proprietà:

Porte logiche:

Circuiti combinatori di porte logiche (Da espressione/funzione matematica, a circuito hardware):

Operatori composti:

NAND:

NOR:

XOR:

XNOR:

Sintesi e analisi di un circuito:

Funzione di parità:

Funzione di maggioranza:

Mappe di Karnaugh

Mappa di Karnaugh con configurazioni a 4 valori in input:

Rappresentazione piana ciclica:

Blocchi funzionali:

Decoder:

Encoder:

Multiplexer (MUX):

Circuito associato (costituito utilizzando un decoder):

Comparatore:

Half adder:

Full adder:

Sommatore su N bit (Sommatore a "propagazione di riporto"):

Sommatore ad anticipazione di riporto (carry lookahead in inglese):

Sottrazione:

Estensione di segno e shift:

Moltiplicatore:

ALU:

AND e OR:

ADD:

SUB:

Comparazione (Set Less Than):

Overflow:

ALU per operazioni floating point:

PLA:

ROM (Read Only Memory):

Dai circuiti combinatori a quelli sequenziali:

Bistabile SR (Latch):

Circuiti sequenziali sincroni:

Bistabile SR Sincrono (Latch sincrono):

Latch D:

Flip Flop:

Macchine a stati finiti (FSM, Finite State Machines):

Approccio Firmware:

Approccio sequenziale applicato alla moltiplicazione:

Approccio sequenziale applicato alla divisione:

Divisione binaria:

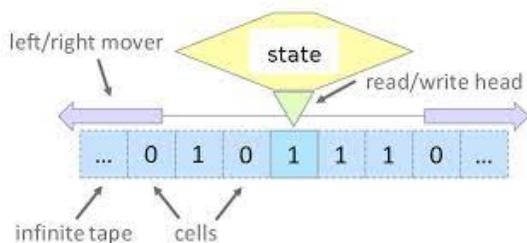
Algoritmo:

Elaboratore ed elaborazione:

Elaboratore → Macchina in grado di **rappresentare, memorizzare e manipolare**, delle informazioni date in input, per produrne altre in output.

L'Operazione di **elaborazione**, è strettamente connessa con l'hardware (elaboratore) che la esegue.

Macchina di Turing:



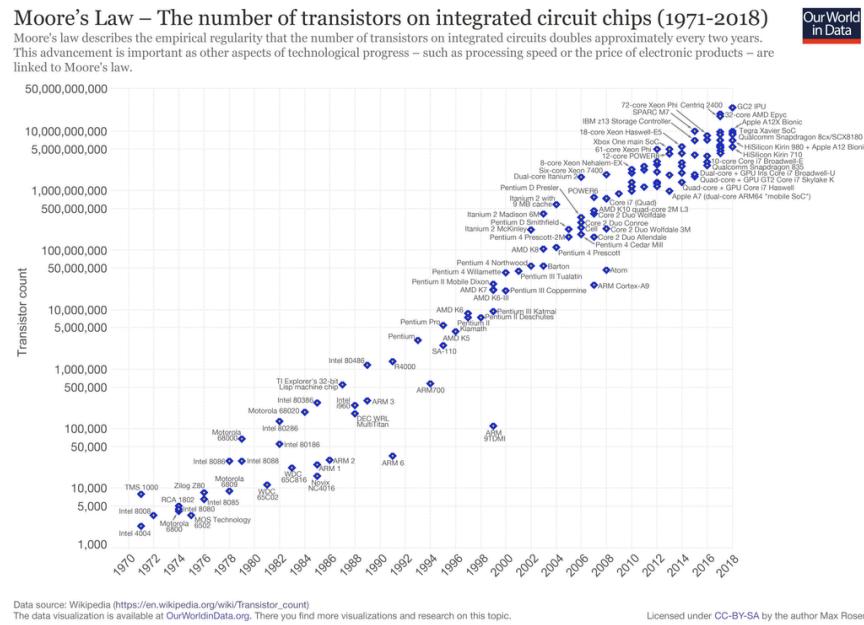
Una semplice rappresentazione teorica di elaboratore, è la **"Macchina di Turing"**, in grado di eseguire qualsiasi algoritmo.

Tale modello permette di capire se un'algoritmo è computabile o no, in base a se è possibile eseguirla sulla "Macchina di Turing" o meno.

La Storia dell'elaboratore, ha mostrato nel corso del tempo:

- Aumento velocità
- Miniaturizzazione
- Diminuzione costi
- Efficienza energetica
- Differenziazione e specializzazione

La legge di "Moore" (Moore's law), esprime tali miglioramenti in ordine esponenziale.



"La complessità di un microcircuito, misurata ad esempio tramite il numero di transistor per chip, raddoppia ogni 18/24 mesi."

Come si nota nell'angolo in alto a destra, quest'era sta giungendo al termine, cioè tale grafico sta assumendo più le sembianze di una sigmoide (esponenziale che col tempo si appiattisce) che di un'esponenziale.

Architettura di un'elaboratore



L'Architettura di un'elaboratore, è la descrizione di com'esso è fatto, e comprende:

- **Hardware Design** → Disegno logico dei circuiti che realizzano i vari componenti dell'elaboratore (ALU, Control Unit, Bus...)
- **Hardware Organization** → Schema di connessione dei vari componenti, e descrizione di come operano tra loro.
- **Instruction set architecture (ISA)** → Quali sono le istruzioni che l'elaboratore può eseguire, sequenza di istruzioni in linguaggio macchina, cioè quelle che la UC è in grado di decodificare come istruzioni.
 - **ISA RISC (Reduced ISA)** - prevede sole istruzioni semplici e regolari, che richiedono poca capacità computazionale per essere eseguite.

Vantaggi: Più facili da progettare, con prestazioni ed efficienza energetica migliori (essendo le singole operazioni più semplici, consigliate per applicazioni mobile ed embedded, scaldano meno), più facile per il compilatore convertire codice di alto livello in linguaggio macchina.

Svantaggi: Aumenta tuttavia la quantità di codice da produrre (Esempio: scrivere uno stesso codice in Python o in Assembly), inoltre introduce accessi alla memoria più frequenti.

Alcune architetture **RISC: PowerPC, ARM, MIPS**

- **ISA CISC (Complex ISA)** - sono previste nel linguaggio macchina, anche istruzioni più complesse.

Vantaggi: Codice più compatto, set di istruzioni più ricco, meno accessi a memoria.

Svantaggi: Richiesta architettura più complessa su cui eseguire CISC, difficoltà per il compilatore.

Alcune architetture **CISC: Intel X86/X64, AMD 64.**

Esempio: passare da un linguaggio di programmazione di basso livello, ad uno di alto livello.

L'architettura di un elaboratore, introduce quindi livelli di astrazione sempre più elevati, per cui l'utente esterno può utilizzare la macchina, senza conoscerne

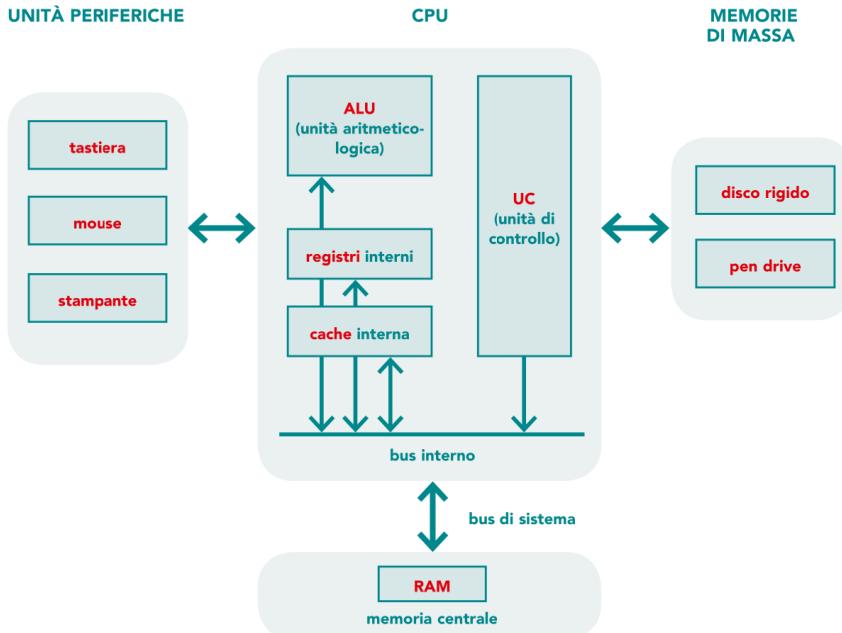
l'architettura.

Modello di Von Neumann:

Schema concettuale che permette di descrivere un elaboratore generico.

Ogni elaboratore deve avere i seguenti componenti, indipendentemente dal modo in cui ogni produttore li implementa:

- CPU - Unità di elaborazione col compito di eseguire istruzioni in sequenza (operazioni matematiche, logiche, accesso a memoria...)
 - UC - Supervisore dell'esecuzione
 - ALU - Esecutore di calcoli logici o aritmetici
 - Registri - piccole locazioni di memoria estremamente performanti.
- Memoria - a cui la CPU accede in lettura o scrittura, attraverso una struttura di connessione detta "bus" composta da piste elettriche ricavate sulla piastra madre, suddivisa in blocchi di bit dette "word", a 16,32 o 64 bit.
- Periferiche - Dispositivi fisici esterni, direttamente connessi alla motherboard, che permettono interazioni esterne.
 - Periferiche di I/O
 - Periferiche "di massa" (HDD/SSD/Flash Drive), prevedono fenomeni fisici di consumo, oltre a velocità di risposta estremamente ridotte rispetto alle memorie alimentate.



Questo modello, introduce un importante limite: il "bottleneck".

I componenti di questo modello infatti, non hanno seguito negli anni lo stesso range di incremento di performance (Esempio, la CPU abbiamo detto che ha seguito la legge di Moore, ma la memoria RAM d'altro canto, è diventata più grande e più veloce, ma non così tanto da star dietro alla CPU!)

La CPU, dovendosi quindi costantemente interfacciare con la RAM, rimarrà bloccata a lavorare ai ritmi più lenti della RAM.

Da ciò deriva che il 99% del tempo, la CPU non fa che aspettare i tempi di risposta più lenti della RAM.

Ciclo esecuzione codice:

- Fetch - La CPU, controllando il valore del "Program Counter", si posiziona nell'indirizzo di memoria da esso indicato, legge una porzione variabile di codice in base all'architettura del computer (32/64, legge quindi 4 o 8 Byte), e la carica nell'instruction register.
- Decode - La UC elabora l'istruzione, riconosce le operazioni da eseguire, predisponde l'ALU e i registri per l'esecuzione, caricando i dati richiesti dalla

memoria principale, incrementa "Program Counter"

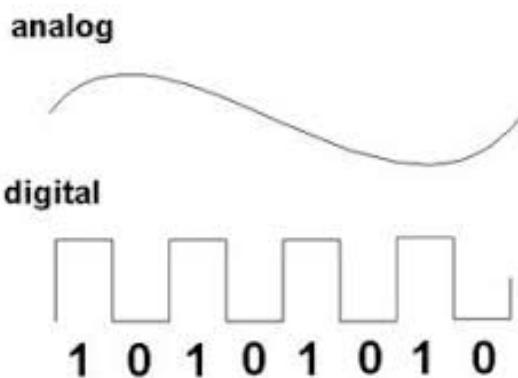
- Execute - Esecuzione dell'istruzione, produzione di un output.
- Memory - Se l'istruzione lo prevede, viene effettuato un accesso a memoria. (Salvataggio dell'output in RAM)
- Write back - Accesso ai registri. (Salvataggio dell'output in registri)

All'accensione del computer, inizia la fase di Fetch dalla posizione 0, in cui è localizzato il firmware.

MIPS

L'Architettura di riferimento per il corso di architettura, è MIPS (Multiprocessor without Interlocked Pipeline Stages), che implementa ISA di tipo RISC.

Codifica dell'informazione:



Per svolgere la propria funzione, un elaboratore **digitale** deve poter rappresentare l'informazione attraverso una grandezza fisica definita, cioè **tensione elettrica**.

Ogni informazione, può quindi essere espressa con un valore di tensione.

Le informazioni fisiche ed elettriche sono in "Analogia" (segnale di tipo analogico), se al variare del segnale, corrisponde una pari variazione del fenomeno fisico.

Il segnale invece di tipo "digitale", non rappresenta l'intensità di un segnale, ma il suo valore in cifre.

Esempio:

Ipotizzando di voler rappresentare il valore "18".

Approccio Analogico	Approccio Digitale
Il Valore del segnale in volt, rappresenta il valore 18 (o 18V, o si sceglie una scala, stabilendo però sempre una corrispondenza diretta)	Sono prodotti più segnali, uno per ogni cifra, che rappresentano le cifre "1" e "8".

Sistema di numerazione:

Come si costruisce la corrispondenza tra un valore (diciotto) e la sua rappresentazione digitale? Come si possono fare operazioni tra numeri rappresentati in quel modo?

Le risposte stanno nella teoria dei sistemi di numerazione

Un sistema di numerazione è composto di due parti fondamentali:

- **Base** - Insieme di simboli che possiamo usare per rappresentare un numero.

L'Insieme $B_{10} = \{0,1,2,3,4,5,6,7,8,9\}$, è l'insieme base del sistema di numerazione decimale.

L'Insieme $B_{16} = \{0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F\}$, è l'insieme base del sistema di numerazione esadecimale.

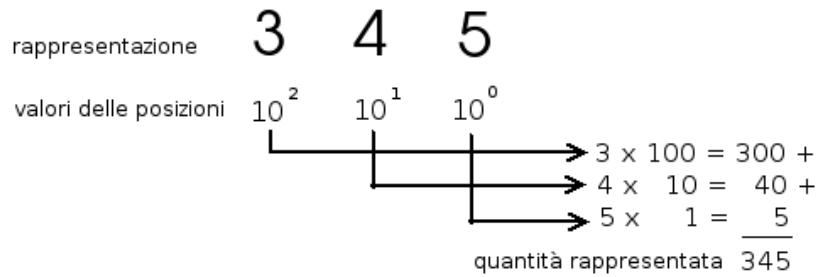
Con n cifre in base B , posso rappresentare B^n stringhe.

"Quanti numeri diversi posso scrivere con 4 cifre in base 16?" 16^4

"Quante cifre della base B mi servono per scrivere almeno P stringhe diverse?" $\log(b,p)$

- **Notazione** - Regole con cui passo da una sequenza di simboli, ad un valore.

La notazione principale è la **posizionale**, secondo la quale si fa la **somma dei valori associati ai simboli**, pesati in base alla **base** utilizzata e la loro **posizione**.



Numeri naturali:

I numeri naturali si scrivono in notazione posizionale pura (come nell'esempio precedente)

- Da base B a base **10**: calcolo della somma pesata

$$(111011)_2 = \sum_{i=0}^5 b_i 2^i = 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 1 \times 2^5 = (59)_{10}$$

- Da base **10** a base B: algoritmo iterativo delle divisioni (Divisioni Successive)



Da base 2 a base 16 e vice versa

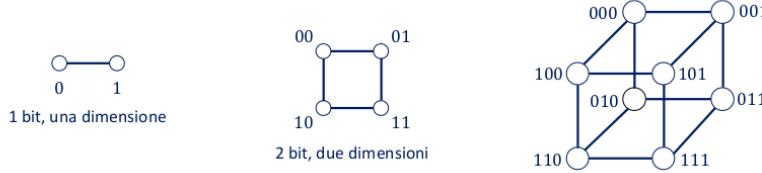
- È un caso particolare della conversione tra due basi B_1 e B_2 dove una è una potenza dell'altra, cioè $B_2 = B_1^m$ per un qualche m intero positivo
- In questo caso $B_1 = 2$, $B_2 = 16$ e quindi $m = 4$
- La cifra in posizione i di un numero in base B^m corrisponde all' i -esimo gruppo di m cifre del numero in base B
- Nel nostro caso abbiamo 16 possibili gruppi diversi di 4 cifre binarie, possiamo costruire una tabella che mette in corrispondenza ogni **cifra esadecimale** con una stringa di 4 bit

0	0000	4	0100	8	1000	C	1100
1	0001	5	0101	9	1001	D	1101
2	0010	6	0110	A	1010	E	1110
3	0011	7	0111	B	1011	F	1111

- Per le conversioni è sufficiente ispezionare la tabella e trovare le corrispondenze (*nota: non dovrebbe servire impararla a memoria!*)
- Esercizio:** convertire $(A\ 8\ F\ B)_{16}$ in base 2
Soluzione: Ispezionando, simbolo per simbolo, la tabella ottengo $(1010\ 1000\ 1111\ 1011)_2$

Codice di Grey:

- Un numero binario su n bit può essere interpretato come un punto in uno spazio n -dimensionale

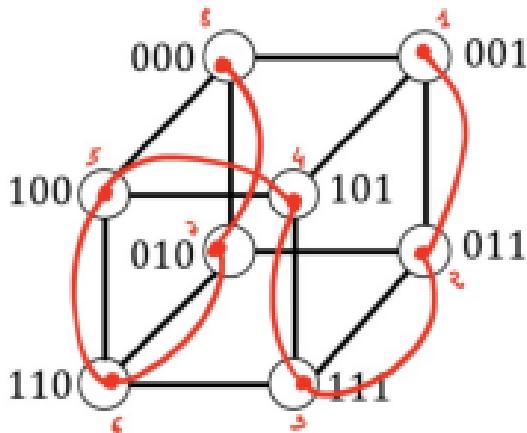


- In tutti i casi, numeri adiacenti (collegati) differiscono solo di un bit
- Dati n_1 e n_2 il numero di posizioni in cui un bit ha valore diverso tra un numero e l'altro si chiama **distanza di Hamming**
- Misura la distanza tra due codifiche, la differenza tra due rappresentazioni di due numeri che è diversa, in generale, dalla differenza tra i due valori rappresentati
- Esercizio:** quanto è la distanza di Hamming tra 10111010 e 10010111?
- Risposta: se evidenzio i bit diversi ho 10111010 e 10010111 quindi la distanza di Hamming è pari a 4 (numero di bit che devo complementare per trasformare un numero nell'altro)

Adottando una codifica posizionale, e supponendo di avere 3 bit binari (B^{t^n} numeri), le "distanze di Hamming" tra numeri successivi, saranno 1,2,1,3,1,2,1

Nel codice di Grey le distanze sono sempre pari a 1, per questo è detto "codice a distanza unitaria".

I Codici di questo tipo, sono individuati con un percorso hamiltoniano sulla rappresentazione grafica (che percorre tutti i punti senza rivisitarli)



Codifica degli interi Z

Gli interi introducono il segno, che dobbiamo rappresentare.

Ci sono due metodi per distinguere tra numeri positivi e negativi:

- Modulo e segno

Riservo il primo bit a sinistra (MSD) per la rappresentazione del segno (0 +, 1 -), i restanti ($n-1$) per il modulo.

Introduce ridondanza della duplice rappresentazione dello 0 (con segno + e -)

Esempio: il numero 10011, se interpretato come naturale vale 19, se invece lo interpretiamo come un intero in notazione modulo e segno vale -3

- Complemento a 2

- Se N è positivo o nullo, lo codifico su $n-1$ come numero naturale e pongo MSD a 0, Il Numero più grande è $2^{n-1} - 1$, tutti i positivi iniziano con 0, per convenzione anche lo zero ha segno positivo
- Se N è negativo, lo codifico come il naturale $2^n - |N|$ su n bit, Il Numero più piccolo è -2^{n-1} , tutti i negativi iniziano con 1

- **Esercizi:** suppongo di avere $n = 4$ bit
- $N = 5 \rightarrow$ converto 5 su 3 bit (metodo delle divisioni iterative) e aggiungo uno 0 a sinistra $\rightarrow 0101$
- $N = -5 \rightarrow$ converto $2^4 - 5 = 16 - 5 = 11$ su 4 bit $\rightarrow 1011$
- $N = 8 \rightarrow$ converto 8 su 3 bit \rightarrow overflow!
I 4 bit non bastano ne servono almeno 5
- $N = -1 \rightarrow$ converto $2^4 - 1 = 15$ su 4 bit $\rightarrow 1111$
- $N = -8 \rightarrow$ converto $2^4 - 8 = 8$ su 4 bit $\rightarrow 1000$
- $N = -11 \rightarrow$ converto $2^4 - 11 = 5$ su 4 bit $\rightarrow 0101$
ma quindi è 5 o -11? Attenzione all'intervallo rappresentabilità!



Chi sta fuori dall'intervallo $[-2^{n-1} ; 2^{n-1} - 1]$ (Lo zero, per convenzione positivo) non può essere rappresentato su n bit in complemento a 2

- Metodo alternativo per convertire $-N$ in C2 su n bit:
 1. Verifico la rappresentabilità su n bit (va sempre fatto)
 2. Converto N in binario
 3. Faccio il complemento a 1 (inverte tutti i bit)
 4. Sommo 1 in binario (regole dei naturali)
- **Esercizio:** convertire -6 su 3 bit
 1. Il numero più piccolo su 3 bit è $-2^2 = -4$, non si può fare!
- **Esercizio:** convertire -6 su 4 bit
 1. Il numero più piccolo su 4 bit è $-2^3 = -8$, ok!
 2. 6 in binario su 4 bit è 0110
 3. Complemento a 1: 1001
 4. Sommo 1:

$$\begin{array}{r}
 1 \\
 0\ 0\ 1 \\
 + \\
 0\ 0\ 0\ 1 \\
 \hline
 1\ 0\ 1\ 0
 \end{array}
 \leftarrow \text{risultato}$$
- Per convertire da C2 a base 10 basta usare la regola posizionale **dando al bit più significativo un peso negativo**

$$\begin{aligned}
 (11101)_C2 &= -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i \\
 &= -1 \times 2^5 + 1 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 \\
 &= (-5)_{10}
 \end{aligned}$$
- Il bit più significativo è un bit di segno a tutti gli effetti: può essere esteso senza modificare il valore assoluto: $(111111111111111111011)_C2$ è sempre $(-5)_{10}$
- Altra buona proprietà: le somme algebriche si fanno con lo stesso procedimento dei naturali ignorando l'ultimo riporto

$$\begin{array}{r}
 0\ 1\ 0\ 0\ + \\
 (4)_{10} = (0100), (-5)_{10} = (1011), \quad \longrightarrow \quad 0\ 1\ 0\ 1\ 1 = \\
 \hline
 1\ 1\ 1\ 1
 \end{array}$$
- **Esercizio:** eseguire $7 - 1$ (su 4 bit)

$$\begin{array}{r}
 1\ 1\ 1\ 1 \\
 0\ 1\ 1\ 1 + \\
 (7)_{10} = (0111)_C2, (-1)_{10} = (1111)_C2, \quad \longrightarrow \quad 1\ 1\ 1\ 1 = \\
 \hline
 (1) 0\ 1\ 1\ 0
 \end{array}$$

- Il problema dell'overflow si ripropone anche per le somme in C2
- Il risultato di una somma di due numeri in C2 su n bit potrebbe cadere fuori dall'intervallo di rappresentabilità $[-2^{n-1}, 2^{n-1}-1]$

$$\begin{array}{r}
 111 \\
 00110010 + (80)_{10} \quad \text{Mi aspetto } (130)_{10} \text{ ma} \\
 01010000 = (50)_{10} \quad \text{il risultato in C2} \\
 \hline
 10000010
 \end{array}
 \quad \text{Il numero più grande su 8 bit è 127!}$$

$$\begin{array}{r}
 1 \\
 10000000 + (-128)_{10} \quad \text{Mi aspetto } (-129)_{10} \\
 11111111 = (-1)_{10} \quad \text{ma il risultato in C2} \\
 \hline
 (1) 01111111
 \end{array}
 \quad \text{Il numero più piccolo su 8 bit è -128!}$$

- Può succedere solo quando si sommano numeri dello stesso segno
- Si riconosce facilmente:
 1. Sommo due numeri **positivi** (bit di segno 0) e ho un risultato **negativo** (bit di segno 1)
 2. Sommo due numeri **negativi** (bit di segno 1) e ho un risultato **positivo** (bit di segno 0)
- Alternativa: controllare gli ultimi due riporti generati, se sono diversi c'è stato overflow

Numeri reali:

- Non li usiamo per contare, ma per **misurare**, numeri **piccolissimi** o molto **grandi**.
- A differenza dei Naturali, non si possono contare, perchè **tra due reali ci sono infiniti reali**.

I Reali sono quindi non rappresentabili su **n** bit, dati i loro infiniti decimali, ma possiamo solo approssimarli con dei numeri razionali a precisione finita.

Consideriamo quindi l'insieme Q (Frazioni).

Estensione della notazione posizionale, per la rappresentazione di una parte decimale.

Introduco nella notazione posizionale, la virgola, simbolo che divide le posizioni di indici positivi e quelle di indici negativi.

$$(1101,110)_2$$

3 2 1 0 -1 -2 -3
 $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3}$
 $8 + 4 + 0 + 1 + 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 0$



Si assume per semplicità, che la parte intera sia sempre rappresentata in puro binario, e non in Complemento a 2.

Da base 10 a base 2.

Dato un numero (**I.F**) in base **dieci**, dove **I** indica la parte intera, e **F** la parte frazionaria:

- Converto la parte intera **I** in un numero **binario naturale**
- Moltiplico **.F** per **due**, e la parte intera ottenuta, diventa la prima cifra più significativa che resta da calcolare
- **Ripetere punto precedente**, applicato al risultato della moltiplicazione precedente, esclusa la parte intera.

Esempio: Convertire in binario il numero **(4.4375)** in base 10.

Conversione parte intera: 4 in Binario → 100

Conversione parte frazionaria:

$$0.4375 * 2 = \mathbf{0.875} \text{ Parte intera } \rightarrow \mathbf{0}$$

$$0.875 * 2 = \mathbf{1.75} \quad \text{Parte intera } \rightarrow \mathbf{1}$$

$$0.75 * 2 = \mathbf{1.5} \quad \text{Parte intera } \rightarrow \mathbf{1}$$

$$0.5 * 2 = \mathbf{1} \quad \text{Parte intera } \rightarrow \mathbf{1}$$

0

Per ottenere la rappresentazione binaria della parte decimale, **raggruppo le parti intere guardando dall'alto verso il basso**:

| (4.4375) in base 10 → 100.0111 in base 2.



Attenzione! Non sempre la parte frazionaria rappresentata in binario prevede **cifre finite**. In tale caso, si procede ad un'approssimazione per **arrotondamento o troncamento**

- **Arrotondamento:** scarto le cifre come nel troncamento, ma scelgo se arrotondare per eccesso o per difetto cercando di minimizzare l'errore di approssimazione
 - **Esercizio** arrotondare $(101.110\textcolor{red}{1}1010100)_2$: per eccesso $101.110 + 000.001 = 101.111$
 - **Esercizio** arrotondare $(0.00101\textcolor{red}{0}011010100)_2$: per difetto 0.00101 (come troncamento)
-

Ripartizione dei bit tra parte intera e frazionaria:

- **Rappresentazione in virgola fissa**

Dati N bit, assegno sempre una porzione fissa di bit per la parte intera, e una per la parte frazionaria, e mantengo questa ripartizione per ogni rappresentazione numerica.

Essendo la ripartizione sempre fissa, la virgola può essere omessa, perché so sempre dov'è posizionata.

Esempio: Calcolatore con **architettura a 5 bit**, con ripartizione a 2 bit per la parte intera, e 3 bit per la parte frazionaria.

Se volessi rappresentare il numero 7.1, non potrei, perchè la stretta assegnazione di 2 bit per la parte intera, non mi permette di rappresentare il numero 7.

Domanda: quale è il numero massimo rappresentabile?

Risposta: $2nI - 1 + \sim 1 \cong 2nI$ (2 elevato al numero di bit della parte intera meno uno, più circa uno (parte frazionaria)
nell'esempio sopra è $11.111_2 = 3.875_{10}$, cioè quasi 4 ($\cong 4$)

Domanda: quale è il numero più vicino allo 0 rappresentabile?

Risposta: 2^{-nF}

2^{-nF} è il contributo più piccolo possibile dato da un bit nella nostra codifica, è anche la differenza di valore tra due rappresentazioni numeriche successive (due tacche): viene anche chiamato **precisione**

Più è alta la precisione, più il valore della precisione è PICCOLO.

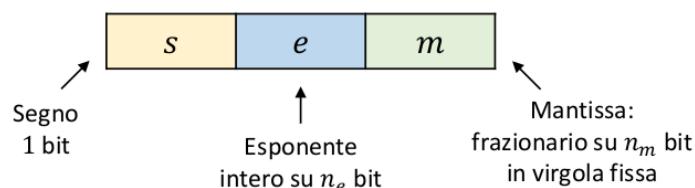
In virgola fissa, tale precisione è uguale sia per la parte intera che per quella decimale, pertanto lo stesso errore che potrebbe essere tollerabile negli interi, potrebbe non esserlo nella parte frazionaria.

2^(-nF)



Più cifre dedichiamo alla parte frazionaria, più piccolo è il valore di precisione e più fitte sono le tacche: otteniamo una approssimazione migliore!

- Rappresentazione a **virgola mobile**:



$$\text{In totale } 1 + n_e + n_m = n$$

Ora i miei N bit, vanno divisi in 3 gruppi:

- Bit di segno (0 - Positivo, 1 - Negativo)
- Esponente
- Mantissa in forma normalizzata, cioè con parte intera a una sola cifra.

In base 2 significa che la mantissa è sempre fatta così 1.XXXXXX

L'1 è quindi implicito

Rende più semplici alcune operazioni, come i confronti:

Domanda: chi è il maggiore tra 1101×2^{-1} e 10.11×2^1 (non normalizzati)

Domanda riformulata: chi è il maggiore tra 1.101×2^2 e 1.011×2^2 (normalizzati)

Nel secondo caso è più facile rispondere! Basta confrontare gli esponenti (ordini di grandezza) e, se sono uguali, si confrontano le mantisse

Il Numero sarà quindi rappresentato secondo notazione scientifica.

Il Valore del numero è dato da: $-1^s \times m \times 2^e$

Dove -1^s sta a indicare l'elevamento a 0 o 1, per esponente positivo o negativo ($-1^0 = 1$ o $-1^1 = -1$, traduzione matematica della rappresentazione del segno con valori binari 0 e 1)

In base al segno dell'esponente, posso spostare la virgola della mantissa a sinistra o destra, e quindi posso rappresentare sia numeri molto piccoli che molto grandi.

Esempi:



L' esponente, mi indica nel primo caso di spostare la virgola a destra, quindi il numero rappresentato sarà 00011.011

Nel secondo esempio, il numero rappresentato è 0.0001111

Standard IEEE 754

L'implementazione della rappresentazione in virgola mobile dentro i calcolatori moderni è regolata da uno standard: l'IEEE-SA n. 754 for floating point arithmetic.

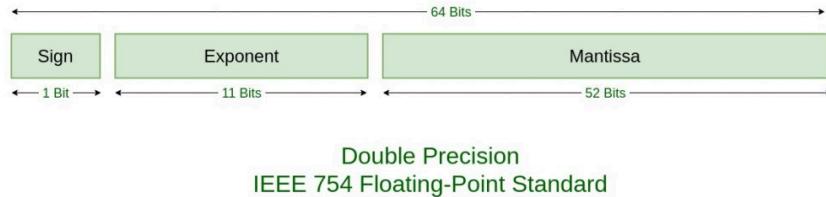
- Formato a precisione singola (detto float) su 32 bit
- Formato a precisione doppia (detto double) su 64 bit

Regola: in precisione singola un numero in virgola mobile è fatto così: 1 bit per il segno, E (esponente) su 8 bit, m (mantissa) su 23 bit.

Con questi 32 bit si può rappresentare:

- Numeri in virgola mobile normalizzati
- Numeri in virgola mobile non normalizzati (detti sub-normalizzati o de-normalizzati)
- Codici speciali

Su 64 bit invece:



In entrambe le rappresentazioni, l'esponente può assumere due valori "speciali", cioè tutti 0, per la rappresentazione dello zero o dei numeri de-normalizzati, e tutti 1, per la rappresentazione dell'infinito o del valore NaN (NaN è un valore di avvertimento, che viene restituito come risultato di operazioni invalide, come la divisione per zero, o la radice quadrata di un negativo).

Numeri normalizzati (queste regole valgono solo per i normalizzati)

Se $0 < E < 255$ (binario naturale) allora i 32 bit stanno codificando un numero normalizzato, quindi implica che i 3 campi vadano interpretati in questo modo:

- s è il segno del numero (0 per dire positivo, 1 per dire negativo)
- m è la parte frazionaria della mantissa assunta in forma normalizzata: (1.m) dove 1. è隐含的 (implicito)
- E è il valore dell'esponente a cui è stato sommato 127, si dice in eccesso 127, quindi il vero esponente è $e = E - 127$
- Il valore rappresentato è $(-1)^s \times 1.m \times 2^e$

Esercizio: rappresentare il valore 3.25 in formato IEEE 754 a precisione singola

1. Converto in binario la parte intera 11 e la parte frazionaria .01
2. Normalizzo $11.01 = 1.101 \times 2^4$
3. $s = 0$, $E = 1+127 = 128 = (10000000)$, $m = 10100000000000000000000000000000$ (Il Primo 1 è implicito)

"Perchè il valore dell'esponente è in eccesso 127?"

Il motivo per cui il campo esponente in IEEE 754 è in eccesso di 127 è una scelta di progettazione che ha diverse vantaggi pratici.

Nel formato a virgola mobile IEEE 754, i numeri sono rappresentati come una combinazione di mantissa, esponente e segno. Il campo dell'esponente è un numero intero senza segno rappresentato su un certo numero di bit.

La rappresentazione in eccesso (o bias) è stata introdotta per semplificare le operazioni aritmetiche e facilitare la comparazione tra i numeri in virgola mobile. Invece di rappresentare l'esponente come un numero con segno, si utilizza un valore **bias** (in eccesso) costante, in questo caso 127 per il formato a singola precisione (32 bit) e 1023 per il formato a doppia precisione (64 bit).

L'utilizzo di un bias consente di rappresentare sia numeri positivi che negativi senza la necessità di trattare separatamente i casi di segno. Ad esempio, se il campo esponente fosse a 8 bit senza bias, avrebbe potuto rappresentare valori da -127 a +128. Tuttavia, con un bias di 127, il campo esponente può rappresentare valori da -127 a +128 senza la necessità di gestire separatamente i casi di segno. (0;255)

Quindi, il valore reale dell'esponente è ottenuto sottraendo il bias dal valore memorizzato nel campo esponente. Questo approccio semplifica l'implementazione dell'aritmetica in virgola mobile e migliora l'efficienza delle operazioni di confronto.

Numeri de-normalizzati

Se $E = 0$ e $m \neq 0$ allora i 32 bit stanno codificando un numero sub-normalizzato (cioè un numero compreso tra 0 e il più piccolo normalizzato rappresentabile). Di conseguenza i 3 campi si interpretano in questo modo:

- s è il segno del numero (0 positivo, 1 negativo)
- m è la parte frazionaria del numero la cui parte intera è assunta essere 0: (0.m), dove 0. è implicito

- E viene scartato e si assume $e = -126$ (**ATTENZIONE:** non -127) (127 è il minimo per i normalizzati)
- Il valore rappresentato è $(-1)^s \times 0.m \times 2^{-126}$

Esercizio: qual è il numero normalizzato più vicino allo 0?

Risposta: $1.000 \dots \times 2^{-126} = 2^{-126}$

Il successivo è $1.00 \dots 001 \times 2^{-126}$

Il numero de-normalizzato più vicino allo 0?

Risposta: $0.000 \dots 1 \times 2^{-126}$ circa 1.4×10^{-45}

Il successivo è $0.000\dots 10 \times 2^{-126}$

Valore di E	Valore di m	Cosa rappresentano i 32 bit in IEEE 754?
$0 < E < 255$	qualsiasi	Numero normalizzato
$E = 0$	$m \neq 0$	Numero de-normalizzato
$E = 0$	$m = 0$	± 0 (a seconda di s)
$E = 255$	$m = 0$	\pm Infinito (a seconda di s)
$E = 255$	$m \neq 0$	NaN (Not a Number)

Esistono due codifiche per lo 0, ma con i reali potrebbe non essere ridondante

NaN è un simbolo che indica il risultato di un'operazione non permessa come $10/0$

Perché sommare 127 al valore dell'esponente?

- Riduce la complessità dell'hardware che deve manipolare questi numeri, facilitando alcune operazioni frequenti
- Esempio: confronto due numeri in IEEE 754, chi è il maggiore?
 - Se i segni sono diversi è immediato (il maggiore è quello con $s = 0$ perché positivo)
 - Se i segni sono uguali basta confrontare i restanti bit come si faceva con i naturali. L'esponente, in posizione più alta, domina i bit della mantissa e il segno non va gestito perché non è codificato esplicitamente (eccesso 127)

Circuiti digitali:

Un Circuito digitale, è un composto di tanti elaboratori logici elementari (porte logiche).

Connettendo tra loro le porte logiche, ottengo circuiti in grado di svolgere elaborazioni sempre più complesse.

Le elaborazioni logiche, non sono altro che operazioni matematiche, ma definite da una particolare algebra, che opera sui valori binari: l'algebra di Boole.

Algebra di Boole:

Un'Algebra, è composta da un insieme di simboli, valori e regole per svolgere operazioni su di essi (operatori).

L'Algebra di Boole, opera su simboli e valori binari, (True o False, $B = \{0,1\}$).

In essa non svolgo operazioni come la somma, sottrazione, moltiplicazione e divisione, ma eseguo operazioni di tipo binario, dette "operazioni logiche" (NOT, AND, OR).

Una variabile (a_n) è detta binaria se appartiene al dominio $B = \{0,1\}$.

Combinando valori e simboli binari con gli operatori logici, definisco funzioni logiche (esattamente come nell'algebra a cui siamo abituati), del tipo $f(a_1, a_2, \dots, a_n) \in B$

Esempio di funzione logica su tre variabili:

$$f(a, b, c) = a + \bar{b}c = (a + ((\bar{b})c))$$

Operazioni logiche:

- **NOT - Negazione** logica di una espressione booleana, si indica come "a negato" (\bar{a}), richiede un solo operando.

a	\bar{a}
0	1
1	0

- **AND - Congiunzione** (o prodotto, perchè $a*b=ab$) di due variabili booleane a,b, si indica con (ab).
- Il Risultato è il "**minimo**" delle variabili passate
(Esempio: a=0, b=1, il più piccolo è a=0, quindi il risultato sarà 0)

a	b	ab
0	0	0
0	1	0
1	0	0
1	1	1

- **OR - Disgiunzione** di due variabili booleani ($a+b$)

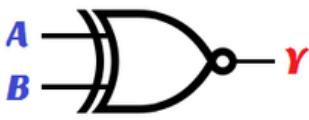
"1 se almeno uno dei due è 1"

Il Risultato è il "**massimo**" delle variabili passate (ES: a=0, b=1, il più grande è b=1, quindi il risultato sarà 1)

- **XNOR** - Restituisce 1 solo quando A e B sono uguali (0 e 0, o 1 e 1).

Porta logica XNOR $Y = \overline{A \oplus B}$

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



Precedenza degli operatori:

Così come in algebra naturale so che le moltiplicazioni e le divisioni hanno precedenza su addizioni e sottrazioni, anche nell'algebra di Boole sono definite delle **"regole di precedenza"**.

La Precedenza Booleana è NOT → AND → OR.

Esempio: $a+c(b$ negato), prima nego b, poi svolgo $c*b$ (AND) poi $a+cb$ (OR).

$$f(a, b, c) = a + \bar{b}c$$

$$\text{Esempio: } f(1,0,0) = 1 + 1*0 = 1 + 0 = 1$$

Principio di dualità:

Data un'espressione booleana, posso ottenere la sua duale trasformando:

- Tutti gli AND con gli OR, e viceversa.
- Tutti gli 0 con gli 1 e viceversa.

Esempio: $1*0 + 0 = 0 \rightarrow$ Duale $\rightarrow 0+1 * 1 = 1$ (Entrambe le espressioni hanno stesso risultato, o entrambe = 1 o entrambe = 0)

Proprietà degli operatori logici:

Proprietà	AND	OR
Identità	$1a = a$	$0 + a = a$
Elemento nullo	$0a = 0$	$1 + a = 1$
Idempotenza	$aa = a$	$a + a = a$
Inverso	$a\bar{a} = 0$	$a + \bar{a} = 1$
Commutativa	$ab = ba$	$a + b = b + a$
Associativa	$(ab)c = a(bc)$	$(a + b) + c = a + (b + c)$

Proprietà	di AND rispetto ad OR	di OR rispetto ad AND
Distributiva	$a(b + c) = ab + ac$	$a + bc = (a + b)(a + c)$
Assorbimento I	$a(a + b) = a$	$a + ab = a$
Assorbimento II	$a(\bar{a} + b) = ab$	$a + \bar{a}b = a + b$
De Morgan	$\overline{ab} = \bar{a} + \bar{b}$	$\overline{a + b} = \bar{a}\bar{b}$

Le proprietà in grassetto nell'immagini, si dicono "postulati", cioè si assumono vere, non si possono dimostrare. (Sono come gli assiomi, si dicono vere e basta)

Tutte le altre proprietà, si dimostrano proprio perchè le 4 in grassetto si assumono vere.

Le stesse proprietà per l'and e per l'or, sono uno il duale dell'altra.

Esempio: $1a=a \rightarrow$ Duale = $0+a = a$



I Valori delle variabili non si cambiano! Si sostituiscono solo gli 0 con 1, non si negano i valori delle variabili (a).

La Proprietà distributiva di AND rispetto ad OR (Del prodotto rispetto alla somma logica), ha sintassi identica a quella che già conosciamo

Dimostrazione di alcune proprietà

Data l'ipotesi che valga la dualità e valgano i postulati (Identità, Inverso, Commutativa e Distributiva).

Dimostrazione dell'idempotenza, cioè $a*a... = a$:

Partendo da a , $a = a + 0$, per proprietà d'identità

Da $a + 0 = a+a(a$ negato), per proprietà dell'inverso. ($0 = a(a$ negato)

Da $a+a(a$ negato) = $(a + a)(a + a$ Negato), per proprietà distributiva $(a + bc) = (a + b)(a + c)$

Da $(a + a)(a + a$ Negato) $\rightarrow (a + a)*1$, per proprietà dell'inverso $(a + a$ Negato)=1

Da $(a+a)*1 \rightarrow (a+a)$, per proprietà d'identità $(1*a)=a$

Da $(a+a) \rightarrow a*a$, per proprietà d'identità. $(1*a)=a$

Quindi partendo da "a", siamo arrivati a "a*a", dimostrato che $a*a=a$.

Dimostrazione dell'assorbimento e dell'elemento nullo:

Ipotesi: Dualità, Postulati, Idempotenza
Tesi: Elemento nullo, $1 + a = 1$

Ipotesi: Dualità, Postulati, Idempotenza, El. nullo
Tesi: Assorbimento I, $a(a + b) = a$

Passaggio	Ottenuto grazie a...	Passaggio	Ottenuto grazie a...
$1 + a$		$a(a + b)$	
$= a + \bar{a} + a$	Inverso ($a + \bar{a} = 1$)	$= aa + ab$	Distributiva
$= a + \bar{a}$	Idempotenza ($a + a = a$)	$= a + ab$	Idempotenza ($aa = a$)
$= 1$	Inverso ($a + \bar{a} = 1$)	$= a1 + ab$	Identità ($1a = a$)
Dimostrata proprietà dell'elemento nullo!		$= a(1 + b)$	Distributiva
		$= a$	El. nullo ($1 + a = 1$)
Dimostrata proprietà dell'assorbimento !!			

Applicazione delle proprietà:

Applicazione delle proprietà

- Esercizio: semplificare la seguente espressione logica: $\overline{a} + \overline{c} + \overline{c} + b(\overline{c} + c\overline{b})$

Passaggio	Ottenuto grazie a...
$\overline{a} + \overline{c} + \overline{c} + b(\overline{c} + c\overline{b})$	
$= \overline{a}\overline{c} + \overline{c} + b(\overline{c} + c\overline{b})$	De Morgan
$= \overline{c} + b(\overline{c} + c\overline{b})$	Assorbimento I ($a + ab = a$)
$= \overline{c} + bc(1 + \overline{b})$	Distributiva
$= \overline{c} + b\overline{c}$	El. nullo e identità
$= \overline{c}$	Assorbimento I

Applicazione delle proprietà

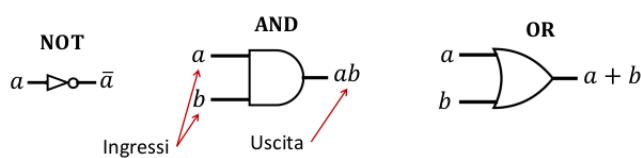
- Esercizio: semplificare la seguente espressione logica: $(\overline{\overline{ab}} + \overline{c})\overline{ab} + \overline{a}\overline{bc}$

Passaggio	Ottenuto grazie a...
$(\overline{\overline{ab}} + \overline{c})\overline{ab} + \overline{a}\overline{bc}$	
$= \overline{ab}\overline{c} + \overline{ab}\overline{c}$	Assorbimento II $a(\overline{a} + b) = ab$
$= \overline{ab}(\overline{c} + c)$	Distributiva
$= \overline{ab}$	El. inverso
$= \overline{a} + \overline{b}$	De Morgan
$= a + \overline{b}$	

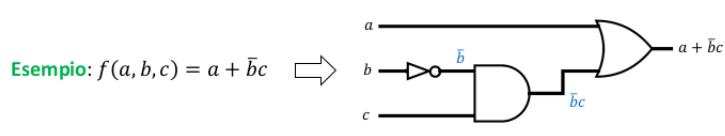
Porte logiche:

Le Porte logiche, sono i componenti hardware associati agli operatori logici, dei circuiti elementari che commutano i segnali di tensione esattamente come fanno gli operatori logici nell'algebra di Boole.

Si indicano graficamente come:

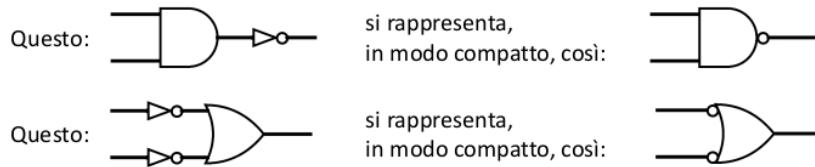


Circuiti combinatori di porte logiche (Da espressione/funzione matematica, a circuito hardware):



Questo circuito, non mantiene memoria dei valori in ingresso, quindi dato un input, ottengo sempre lo stesso output.

- Notazione grafica compatta: quando il NOT (l'inverter) si trova su un ingresso o su una uscita di una porta logica può essere denotato con un **pallino** su quell'ingresso o uscita
- Ad esempio:



Operatori composti:

NAND:

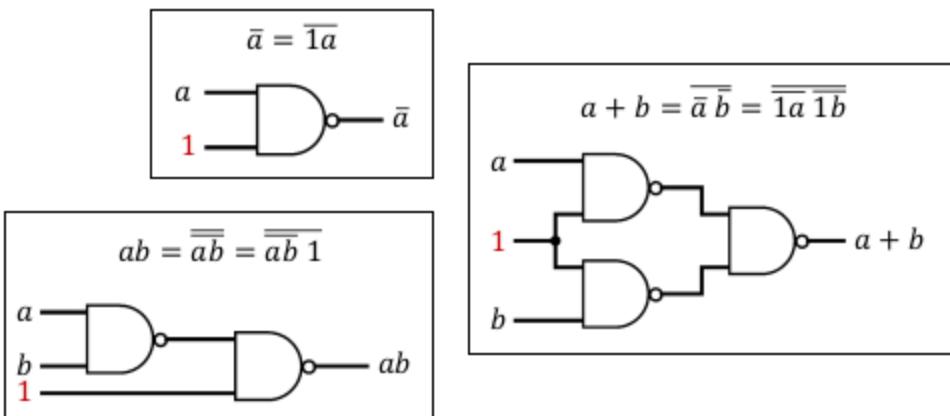
"Not AND", vale 0 solo quando entrambi gli input valgono 1.



a	b	\bar{ab}
0	0	1
0	1	1
1	0	1
1	1	0

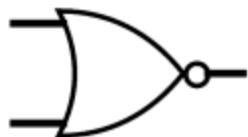


Tramite questa porta, posso implementare anche AND, OR e NOT, pertanto è detta "porta universale".



NOR:

"Not OR", vale 1 solo quando entrambi solo a 0.



a	b	$\bar{a} + \bar{b}$
0	0	1
0	1	0
1	0	0
1	1	0



Anche questa è una porta universale, cioè permette l'implementazione di AND OR e NOT. (Dimostrazione identica a NAND, principio di dualità)

$$\bar{a} = \overline{0 + a}$$

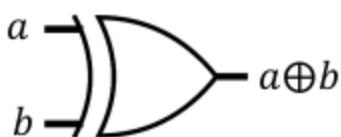
$$ab = \overline{\bar{a} + \bar{b}} = \overline{\overline{0 + a} + \overline{0 + b}}$$

$$a + b = \overline{\overline{a} + \overline{b}} = \overline{0 + \overline{a} + \overline{b}}$$

XOR:

Vale 1 solo quando uno solo degli input vale 1.

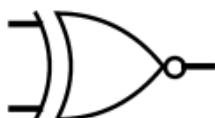
a	b	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0



XNOR:

XOR negato, vale 1 solo quando gli input sono uguali (00 o 11)

a	b	$\overline{a \oplus b}$
0	0	1
0	1	0
1	0	0
1	1	1



Sintesi e analisi di un circuito:

- Analisi di un circuito → Costruire la tabella di verità di un circuito dato.
- Sintesi di un circuito → Costruire il circuito (e quindi ricavare la funzione) partendo dalla tabella di verità.

Si può fare in due modi:

- Prima forma canonica (SOP, Sum of products): dalla tabella di verità, estraggo i "mintermini", cioè i punti in cui la funzione vale 1.

Segno le configurazioni di bit in cui la funzione vale 1, metto in AND, e nego i termini che in quella configurazione valgono 0.

Se nella configurazione di input corrispondente la variabile vale 0, nel mintermini comparirà negata, altrimenti in forma naturale

Le varie configurazioni vanno messe in OR tra loro.

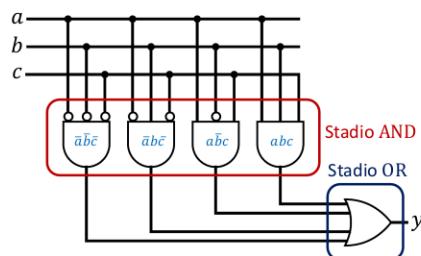
a	b	c	y	y vale 1 se e soltanto se:	y vale 1 se e soltanto se (dualità):
0	0	0	1	$a = 0, b = 0 \text{ e } c = 0$	$\bar{a} = 1, \bar{b} = 1 \text{ e } \bar{c} = 1$
0	0	1	0	oppure	oppure
0	1	0	1	$a = 0, b = 1 \text{ e } c = 0$	$\bar{a} = 1, b = 1 \text{ e } \bar{c} = 1$
0	1	1	0	oppure	oppure
1	0	0	0		
1	0	1	1	$a = 1, b = 0 \text{ e } c = 1$	$a = 1, \bar{b} = 1 \text{ e } c = 1$
1	1	0	0	oppure	oppure
1	1	1	1	$a = 1, b = 1 \text{ e } c = 1$	$a = 1, b = 1 \text{ e } c = 1$
\Rightarrow				$y = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}\bar{c} + a\bar{b}c + abc$	$\Rightarrow ab\bar{c}$

Ricavo poi il circuito:

Prima forma canonica

- Sintesi del circuito: schema a due stadi
 - Stadio AND:** una porta AND per ogni mintermine
 - Stadio OR:** un OR tra tutte le uscite dello stadio AND
- Anche la sintesi del circuito, come la scrittura dell'espressione Booleana, è un procedimento meccanico: lo stesso per ogni funzione logica
- E la forma più compatta?
Probabilmente no! Semplificando l'espressione potremmo sintetizzare un circuito più semplice

$$y = \bar{a}\bar{b}\bar{c} + \bar{a}\bar{b}\bar{c} + a\bar{b}c + abc$$



- Seconda forma canonica (POS, Product of sum): dalla tabella di verità, estraggo i "maxtermini", cioè le configurazioni di input in cui la funzione vale 0.

Segno le varie configurazioni, mettendole in OR tra loro.

Se un bit vale 1, compare in forma negata.

Le varie configurazioni vanno messe in AND tra loro (in ogni passaggio, è esattamente il contrario della prima forma canonica).

Seconda forma canonica

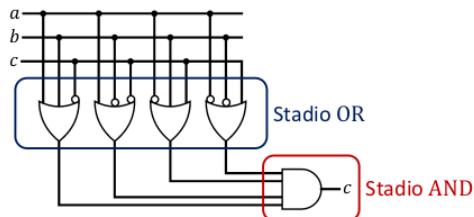
$$\begin{array}{|c|c|c|c|} \hline a & b & c & y \\ \hline 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 1 \\ \hline 0 & 1 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ \hline 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 1 & 1 \\ \hline \end{array} \Rightarrow y = (a + b + \bar{c})(a + \bar{b} + \bar{c})(\bar{a} + b + c)(\bar{a} + \bar{b} + c)$$

Maxtermini:

- Definiti come OR tra tutte le variabili (input) della funzione, ogni variabile compare una volta sola nella sua forma naturale o negata
- Ne abbiamo uno per ogni configurazione di input in cui la funzione vale 1
- Se nella configurazione di input corrispondente la variabile vale 1, nel maxtermine comparirà negata, altrimenti in forma naturale
- Chiamando M_i l' i -esimo maxtermine, una funzione y può essere sempre espressa come l'AND tra tutti i suoi n maxtermini: $y = \prod_{i=1}^n M_i$

Sintesi del circuito

- Ogni maxtermine corrisponde ad un OR a più ingressi (tanti quante sono le variabili)
- Valgono le stesse considerazioni che abbiamo per la SOP



Estrarre gli 1 è la SOP, estraggo mintermini, gli 0 vengono negati.

Estrarre gli 0 è la POS, estraggo MAXTERMINI, gli 1 vengono negati.

Funzione di parità:

Restituisce 1 se il numero di 1 in una configurazione di N bit in input, è dispari.

$p(l_1)$	$p(l_2)$	$p(l)$
0	0	0
0	1	1
1	0	1
1	1	0

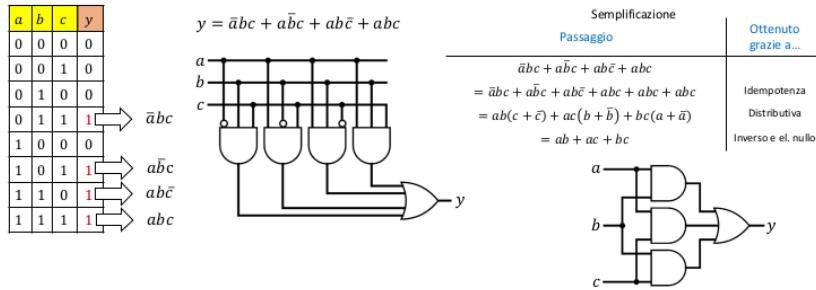
$pari + pari = pari$
 $pari + dispari = dispari$
 $dispari + pari = dispari$
 $dispari + dispari = pari$

$p(l) = p(l_1) \oplus p(l_2)$ Definizione ricorsiva basata su XOR
 Su 3 bit:

Funzione di maggioranza:

Restituisce 1 solo se il numero di 1 in una configurazione di N bit in input, è maggiore degli 0.

Se il numero di 0 e 1 sono uguali, restituisce 0.



Mappe di Karnaugh

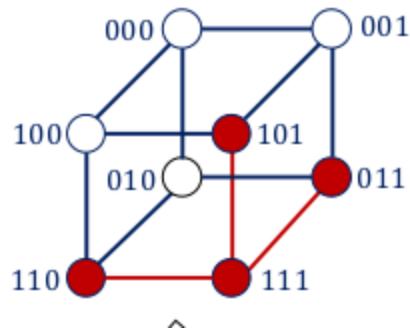
E' un metodo meccanico e grafico, che garantisce di trovare l'espressione minima di un'espressione booleana (e quindi, del circuito ad essa associato).

In un funzione con n variabili, ogni configurazione di input, è una stringa di n bit. (abc nella tabella in esempio).

<i>a</i>	<i>b</i>	<i>c</i>	<i>y</i>
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Posso trasformare tale tabella, in una rappresentazione grafica a n dimensioni:

- 1 Valore in input → Linea
- 2 Valori in input → Quadrato
- 3 Valori in input → Cubo.
- 4 Valori in input → Toro

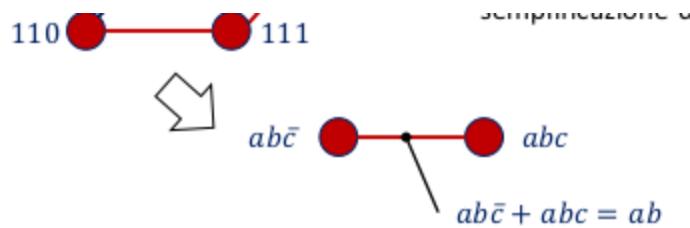


I Punti in questo cubo, sono posizionati in modo da avere distanza massima tra vertici adiacenti pari a 1 (cambia un solo bit tra i vertici adiacenti).

I Punti evidenziati in rosso, sono i “mintermini” della funzione, cioè le configurazioni di input in cui la funzione vale 1.

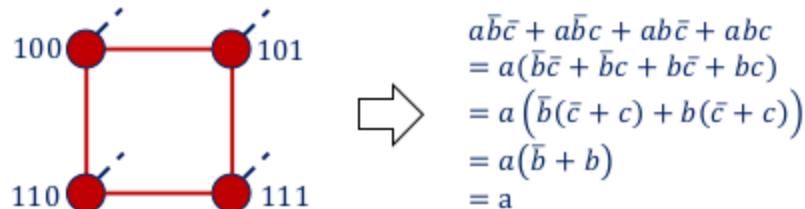
Se considero due punti adiacenti, noto che il lato che li unisce, permette un'opportunità di semplificazione.

Esempio: il lato che unisce 110 e 111, mi accorgo che la funzione vale sempre 1, indipendentemente dal valore di C (terzo bit), pertanto la funzione può essere riscritta come AB.



Lo stesso principio di semplificazione si applica anche a gruppi di vertici connessi:

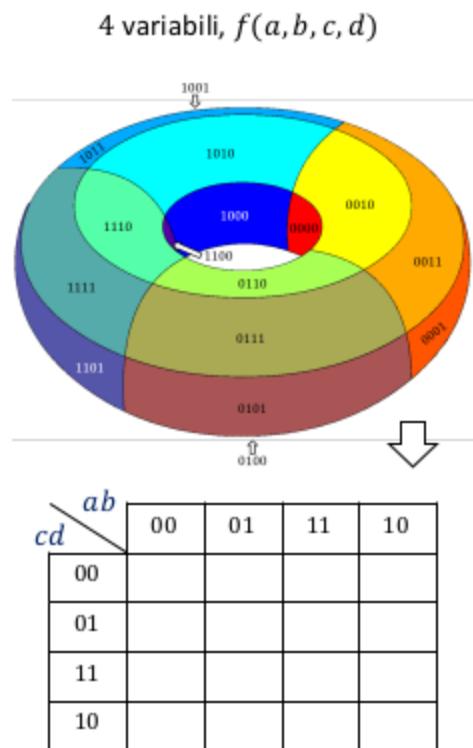
Esempio:



Notando che di tutti i 4 valori, l'unico che rimane costante è 'a', cioè la funzione vale 1 indipendentemente dai valori di b e c, basta che 'a' sia 1, pertanto i 4 vertici possono essere semplificati come 'a'.

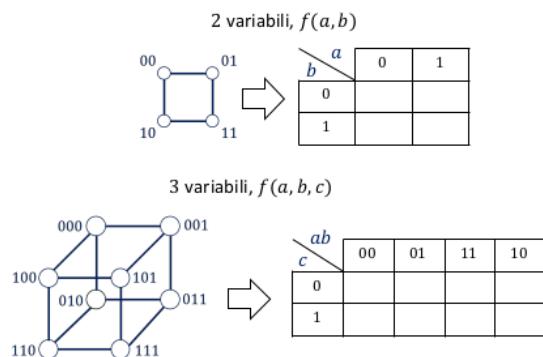
Arrivo alla stessa soluzione, anche sommando i valori dei 4 vertici e semplificando, ma il metodo grafico risulta più semplice, anche se applicabile ad un numero massimo di configurazioni a 4 bit (e già stiamo cercando di vedere una quarta dimensione, che non esiste).

Mappa di Karnaugh con configurazioni a 4 valori in input:



Come possiamo notare, lavorare su una figura del genere, risulta più complesso rispetto a lavorare su un quadrato o un cubo.

Conviene passare ad una rappresentazione tabellare:



Notiamo che le "label" delle colonne (e delle righe, se presenti) sono posizionate secondo la codifica di gray, cioè tra due colonne adiacenti cambia 1 solo bit.

Scrivo poi nelle caselle corrispondenti agli input, i valori associati, e procedo ad identificare "i rettangoli di 1"

- Posso formare rettangoli sovrapposti

- Ogni rettangolo deve essere il più grande possibile
- L'area del rettangolo deve essere una potenza di 2.

The figure shows three Karnaugh maps with arrows indicating the simplification steps:

- First Map:** A 2x2 Karnaugh map with variables a (horizontal) and b (vertical). It has four cells: (0,0)=0, (0,1)=1, (1,0)=1, (1,1)=1. A green rectangle covers cells (0,1) and (1,1). A blue rectangle covers cell (0,0). An arrow points from the map to the expression $f(a,b) = b + a$.
- Second Map:** A 2x4 Karnaugh map with variables a (horizontal) and c (vertical). It has four columns labeled 00, 01, 11, 10. A green rectangle covers cells (0,1) and (1,1). A blue rectangle covers cell (0,0). A yellow rectangle covers cells (1,0) and (1,1). An arrow points from the map to the expression $f(a,b,c) = a$.
- Third Map:** A 4x4 Karnaugh map with variables a (horizontal) and d (vertical). It has four columns labeled 00, 01, 11, 10 and four rows labeled 00, 01, 11, 10. A green rectangle covers cell (0,1). A blue rectangle covers cell (1,0). A yellow rectangle covers cell (1,1). A red rectangle covers cell (0,0). An arrow points from the map to the expression $f(a,b,c,d) = cd + \bar{a}b + a\bar{b}d$.

Per ogni rettangolo, ne ricavo l'implicante.

Nella prima tabella:

- Rettangolo verde → Basta che A sia 1, non mi interessa il valore di B
- Rettangolo Rosso → Basta che B sia 1.

Metto in "OR" tutti gli implicanti, e l'espressione minima della prima tabella è $f(a,b)$
 $= b + a$.

Seconda tabella:

- Rettangolo Verde → Basta che A sia 1, possono cambiare sia B che C, e la funzione vale sempre 1.

Espressione minima → $f(a,b,c) = a$.

Terza tabella:

- Rettangolo Rosso → Basta che CD sia 11, poi del resto non mi interessa (CD)
- Rettangolo Verde → Basta che AB sia 01 (AB)
- Rettangolo Giallo → Basta che AB sia 10, e D sia 1. (ABD)

Metto in "OR" tutti gli implicanti, dove un valore dev'essere necessariamente 0, lo nego.

Espressione minima $\rightarrow f(a, b, c, d) = cd + \bar{a}b + a\bar{b}d$

Rappresentazione piana ciclica:

		<i>ab</i>	00	01	11	10
		<i>c</i>	0	1	1	1
<i>a</i>	<i>b</i>	0	1	0	1	1
1	0	1	0	1	1	1

$f(a, b) = \bar{b} + a$

		<i>ab</i>	00	01	11	10
		<i>cd</i>	1	1	0	0
<i>a</i>	<i>b</i>	0	1	0	0	
0	0	1	1	0	0	0
0	1	0	1	0	0	0
1	1	0	1	1	0	0
1	0	1	1	0	1	0

$f(a, b) = \bar{a}\bar{d} + \bar{a}b + bcd + \bar{b}cd$

Blocchi funzionali:

“Moduli” già implementati, che svolgono elaborazioni molto comuni, che utilizzo come fossero “black box”, senza interessarmi della loro implementazione interna.

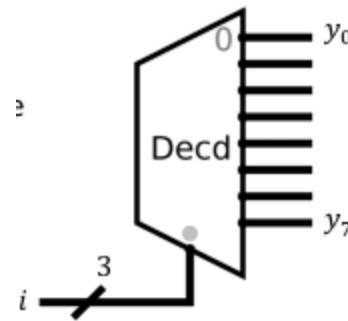
Decoder:

Circuito con n ingressi e 2^n uscite.

Il Numero di valori possibili, è pari al numero di uscite (ES: 3 bit in ingresso, posso avere 2^3 valori, 8 configurazioni, 000,001,010,011,100,101,110,111).

Accende l’i-esima uscita, in base al valore i in ingresso:

esempio di decoder a 3 bit



$i=3$ sul filo di input, indica che quel cavo in input, all'interno è composto da 3 cavi intrecciati e inguainati, avrò quindi configurazioni a 3 bit.

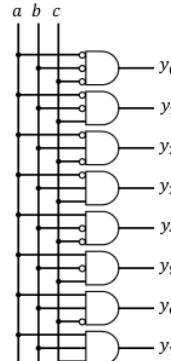
Se sul cavo di input, arriva l'input $101 = 5$, la quinta uscita y_5 sarà attivata, e tutte le altre a zero.

Traduce quindi un valore nel suo codice "one-hot", cioè da una sequenza di bit in ingresso, mette a 1 un solo bit (accende una sola uscita).

- Come è fatto al suo interno?

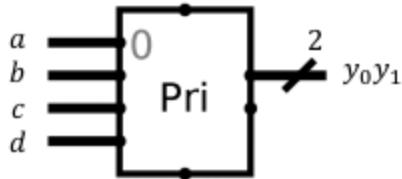
Tabella di verità

a	b	c	y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



- Ogni uscita è una funzione logica con un solo mintermine
- La stessa struttura scala con l'aumentare dei bit in ingresso

Encoder:



Fa il contrario del decoder, cioè trasforma un codice one-hot, in un valore binario corrispondente.

2^n input, in cui solo un valore può essere 1, n uscite.

"Pri", nel disegno sta per "priority", è un'altro modo di chiamare il componente encoder.

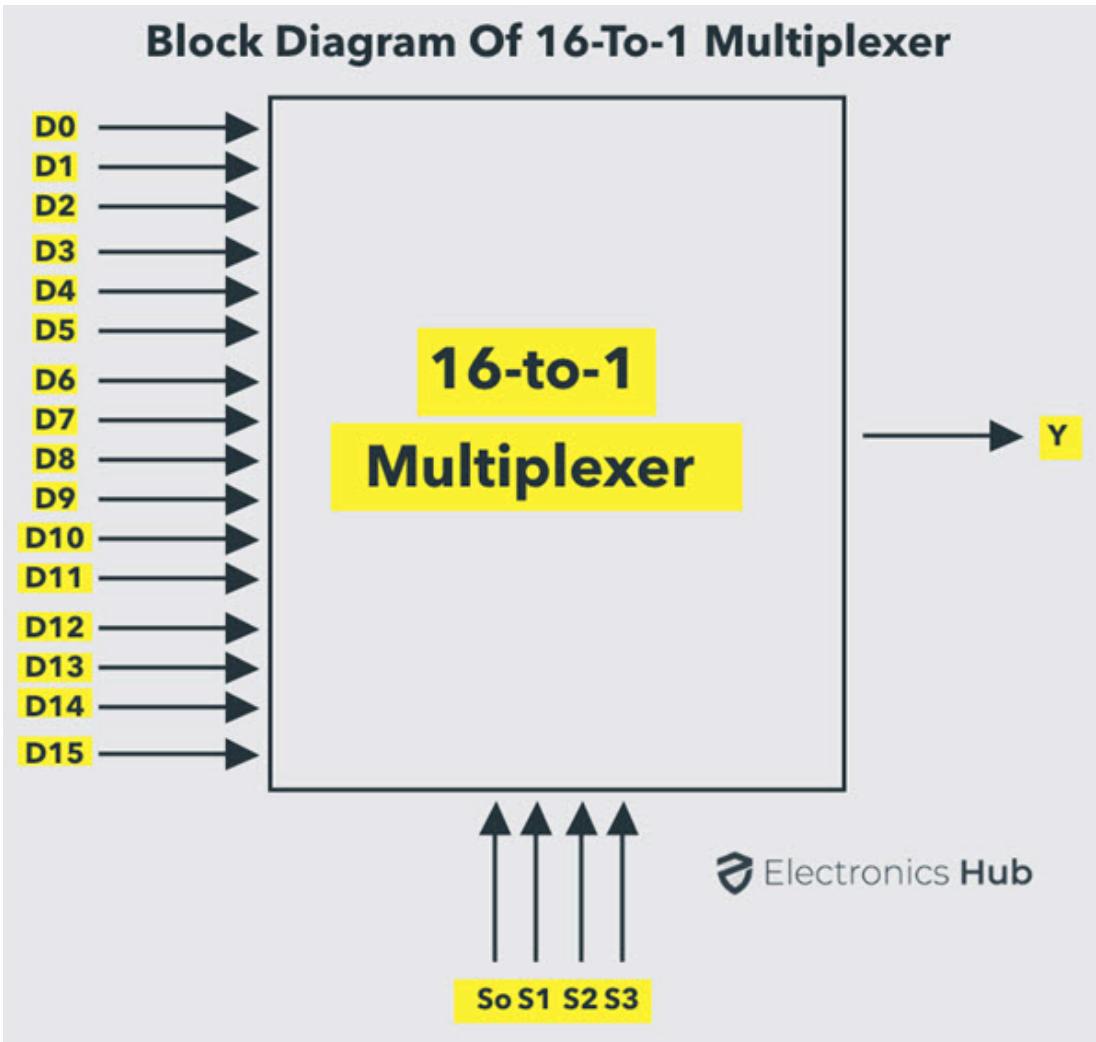
Per definizione, gli input possono avere solo un bit a 1, quindi nella tabella di verità, nel caso di configurazioni non valide (in cui non ci sono 1, o ce ne sono più di uno), scrivo delle 'x'.

a	b	c	d	y ₀	y ₁
0	0	0	0	x	x
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	x	x
0	1	0	0	1	0
0	1	0	1	x	x
0	1	1	0	x	x
0	1	1	1	x	x
1	0	0	0	1	1
1	0	0	1	x	x
1	0	1	0	x	x
1	0	1	1	x	x
1	1	0	0	x	x
1	1	0	1	x	x
1	1	1	0	x	x
1	1	1	1	x	x

Multiplexer (MUX):

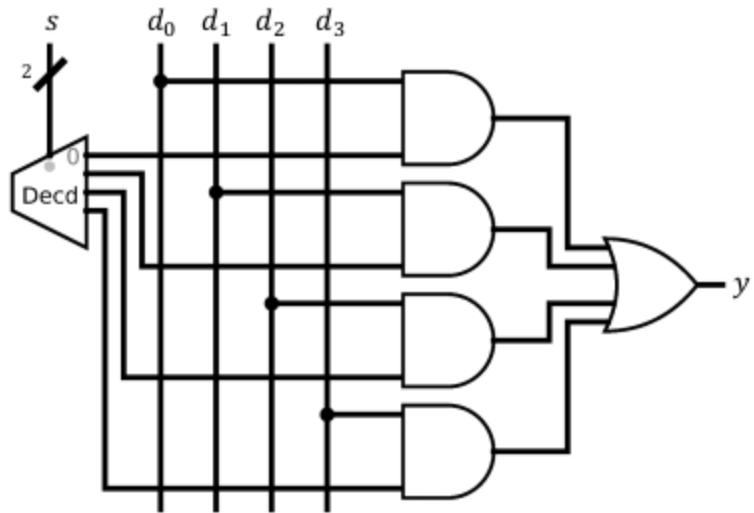
Circuito a $(2^n) + n$ input, e 1 bit in uscita.

Date 2^n linee dati in entrata, e un altro valore a n bit che indica il "numero" di una linea (codice di selezione), il "MUX" restituisce in uscita il valore della linea (N-esima) indicata dagli n bit.



Esempio: se come N bit in entrata ho "0010", il multiplexer restituirà il valore della linea D2.

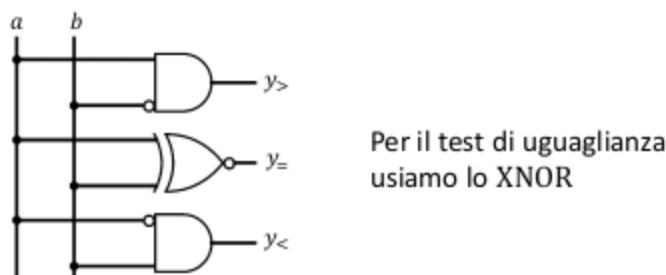
Circuito associato (costituito utilizzando un decoder):



Comparatore:

Riceve in ingresso due numeri binari su N bit, e calcola tre uscite corrispondenti a tre test di confronto:

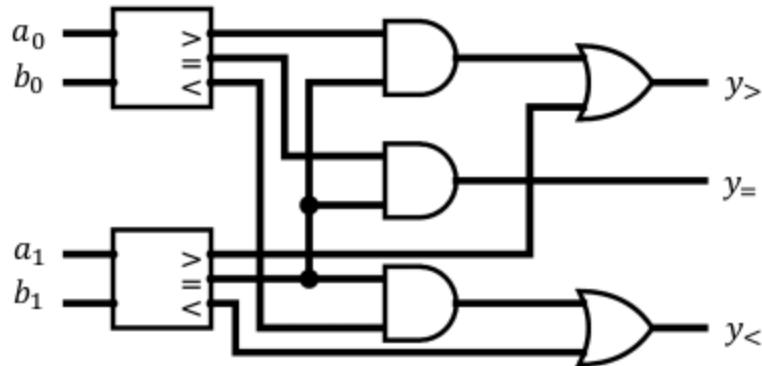
- Test di uguaglianza ($a == b$)
- Maggioranza Stretta ($a > b$)
- Minoranza stretta ($a < b$)



Comparatore a 1 bit

Con un comparatore a 1 bit, posso costruire comparatori a 2 bit, e a sua volta coi comparatori a 2 bit posso costruire comparatori a 4 bit ecc...

In generale, la costruzione di un comparatore a N bit prevede l'uso ricorsivo di comparatori a N/2.



(Ogni uscita di uno, va prima in and con l'uguale dell'altro, poi con la stessa uscita dell'altro)

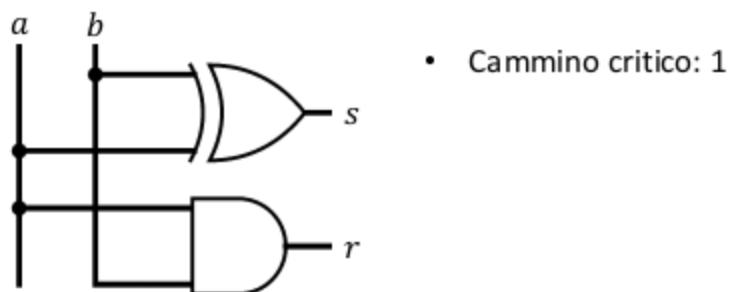
Half adder:

Esegue la somma binaria di numeri a 1 bit.

Prevede due bit di output, uno di somma e uno di riporto (la somma binaria prevede quasi sempre riporto)

Implementato con XOR e AND.

a	b	s	r
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



$$(a+b = s \text{ con riporto } r)$$

Si chiama Half Adder, perchè nella somma non prevede la considerazione del riporto.

Cioè non prevede il caso 1+1 ma con riporto di 1 (che farebbe 1 riporto 1)

Notiamo che la somma prevede 1 solo quando i bit sono diversi (XOR) e il riporto vale 1 solo quando ab sono pari a 1 (AND)

Full adder:

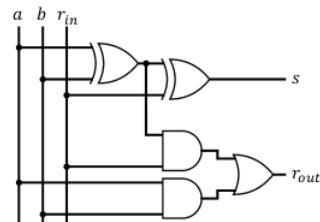
Il Full adder prevede di descrivere tutte le regole della somma binaria, cioè anche il caso 1+1 riporto 1.

Prevede quindi 3 input, a, b, e un eventuale bit di riporto.

a	b	r_{in}	s	r_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

$$\begin{aligned}
 s &= \bar{a}b\bar{r}_{in} + a\bar{b}\bar{r}_{in} + \bar{a}\bar{b}r_{in} + abr_{in} \\
 &= \bar{r}_{in}(\bar{a}b + a\bar{b}) + r_{in}(\bar{a}\bar{b} + ab) \\
 &= \bar{r}_{in}(a \oplus b) + r_{in}(\bar{a} \oplus \bar{b}) \\
 &= r_{in} \oplus (a \oplus b) = r_{in} \oplus a \oplus b \\
 r_{out} &= ab\bar{r}_{in} + \bar{a}br_{in} + a\bar{b}r_{in} + abr_{in} \\
 &= ab(\bar{r}_{in} + r_{in}) + r_{in}(\bar{a}b + a\bar{b}) \\
 &= ab + r_{in}(a \oplus b) = ab + r_{in}(a + b)
 \end{aligned}$$

Implementiamo questa Ma questa si dimostra
 questa che è equivalente!
 grazie al termine ab



- Cammino critico: 3

La tabella di verità con riporto input = 0, è identica a quella dell'half adder.

Noto che s è equivalente a due XOR. e il riporto è equivalente ad (A and B) OR Rinput and (A XOR B).

Sommatore su N bit (Sommatore a "propagazione di riporto"):

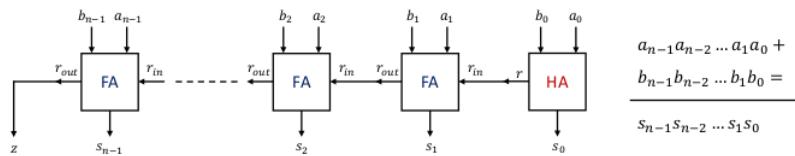
Pensando alla somma di due numeri in binario, la somma degli LSB di due numeri, non prevede considerazione di un riporto, perchè non c'è nulla a destra che può

averlo generato, dopo invece la somma degli LSB, la somma degli altri bit alla sinistra, deve prevedere la considerazione del riporto.

L'implementazione del sommatore a N bit, prevede quindi l'utilizzo di un half adder per la somma degli LSB, e tanti Full Adder concatenati.

L'ultima cifra sarà il resto dell'i-esima operazione (che è ignorato, non fa parte del risultato, ma può essere utile come indicatore di overflow in caso di somma in CA2).

Cammino critico: circa $3N$



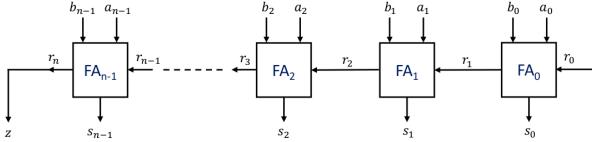
Sommatore ad anticipazione di riporto (carry lookahead in inglese):

Nel sommatore a propagazione di rapporto, ogni full adder abbia bisogno di "aspettare" il riporto dal full adder di prima (propagazione di riporto), introduce una latenza non trascurabile, visto che ogni full adder lavora in serie (e non in parallelo, perchè l'esecuzione di un full adder dipende dal quella del FA precedente)

Bisogna quindi cercare di ridurre il cammino critico del circuito precedente, semplificandone il funzionamento, migliorando le performance (aumentando la difficoltà dell'implementazione però).

L'idea parte dall'idea che il punto critico del circuito precedente, è "aspettare il riporto".

Parto a semplificare un circuito di soli Full Adder, per semplificare le cose.



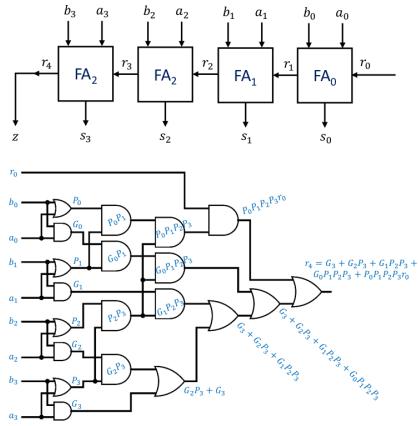
- Ricordiamo che nell' i -esimo Full Adder (FA) $r_{i+1} = a_i b_i + r_i(a_i + b_i)$, usiamo l'espressione semplificata (senza lo XOR)
- Rinomino i termini dentro l'espressione:
 - $a_i b_i$ è il termine di generazione: G_i ; in ogni FA è «subito pronto», non deve aspettare!
 - $(a_i + b_i)$ è il termine di propagazione: P_i ; deve aspettare la propagazione di r_i
- Per ogni riporto derivo la sua espressione, inizio con r_1 e vado avanti fino a r_n
- $r_1 = a_0 b_0 + r_0(a_0 + b_0) = G_0 + r_0 P_0$
- $r_2 = a_1 b_1 + r_1(a_1 + b_1) = G_1 + r_1 P_1 = G_1 + (G_0 + r_0 P_0)P_1 = G_1 + P_1 G_0 + P_1 P_0 r_0$
- $r_3 = a_2 b_2 + r_2(a_2 + b_2) = G_2 + r_2 P_2 = G_2 + (G_1 + r_1 P_1)P_2 = G_2 + (G_1 + (G_0 + r_0 P_0)P_1)P_2 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 r_0$
- ...
- In generale : $r_n = G_{n-1} + P_{n-1} G_{n-2} + P_{n-1} P_{n-2} G_{n-3} + \dots + P_{n-1} P_{n-2} \dots P_0 r_0$

Notiamo che ogni resto, è funzione dei 3 input del full adder precedente, quindi posso scrivere i vari riporti come funzione dei 3 input precedenti, in cui il resto sarà a sua volta input dei 3 precedenti ecc..

Notiamo inoltre che r_0 vale 0, quindi può essere omesso.

Per ogni resto quindi, posso ottenere un'espressione che è si lunghissima, ma prevede solo gli input di ogni full adder, che sono disponibili fin da subito!

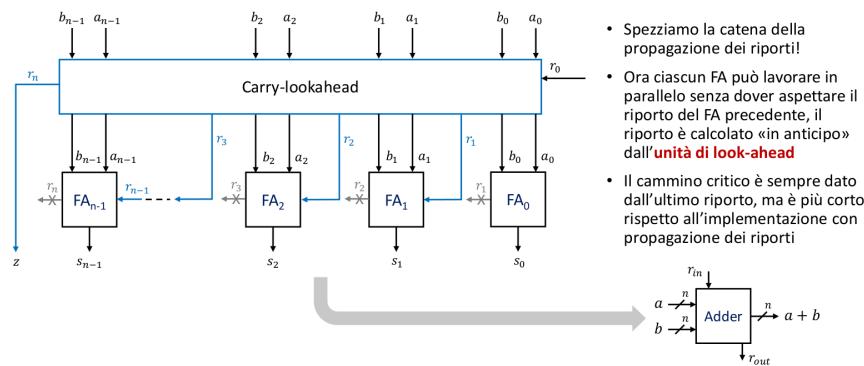
- Consideriamo un sommatore a propagazione di riporto con $n = 4$, se lo implementiamo con lo schema precedente (solo FA) otteniamo un cammino critico pari a $3n = 12$, che corrisponde al numero di porte da attraversare per propagare i riporti fino a r_4
- Se calcoliamo ogni riporto con un circuito dedicato che implementa la sua espressione diretta otteniamo che :
$$r_4 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 r_0$$
- Il cammino critico di questo circuito è 6: se calcoliamo il riporto con questa rete anticipiamo il suo arrivo all'uscita (dimezziamo il numero di porte attraversate) e abbassiamo il cammino critico di tutto il circuito!



Il cammino critico di quel circuito enorme, che è il circuito per il calcolo dell'ultimo riporto (quello più oneroso) è 6, che è comunque meno di $3N$, che è il cammino critico che prevedeva il sommatore a propagazione per ottenere l'ultimo riporto.

Se per ogni riporto calcolo il circuito associato, e semplifico tutti i circuiti che calcolano i riporti per ogni singolo full adder in un singolo componente, ottengo:

- Come cambia la struttura del sommatore?



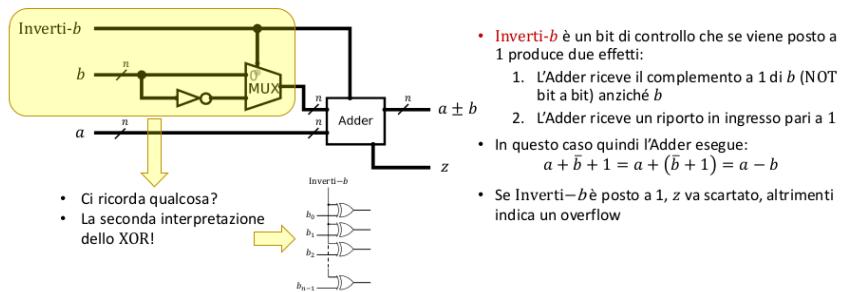
L'elaborazione avviene in parallelo ora, tra i Full Adder, e il componente "Carry-lookahead".

Se incatolo tutto questo circuito in un solo componente, ottengo un "Adder", cioè un sommatore su N bit, con 3 input e 2 uscite.

Sottrazione:

Sottrazione

- Il modulo per l'addizione che abbiamo costruito può essere esteso per gestire anche le sottrazioni
- Metodo:** la sottrazione $a - b$ si interpreta, e quindi si esegue, come una somma binaria tra a e il complemento a 2 di b (la somma si esegue sempre con le stesse regole dei naturali, eccezione fatta per l'uso dell'ultimo riporto)
- Fare il complemento a 2 di b significa complementare a 1 e sommare 1



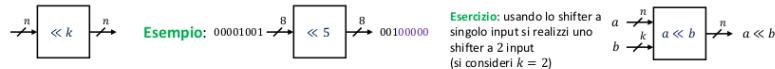
Estensione di segno e shift:

Estensione di segno e shift

- Due operazioni comuni che ci possono tornare utili
- Estensione di segno:** ho un segnale su n bit che voglio dare in input ad un circuito che riceve segnali su $n + m$ bit
- Estendere il segno vuol dire replicare a sinistra l'MSD fino a raggiungere il numero totale di bit desiderati; nel caso in cui gli n bit iniziali rappresentino un naturale o un intero in C2, questa operazione non altera il valore rappresentato



- Shift:** trascinare gli n bit verso sinistra o destra di k posizioni: sinistra $\ll k$, destra $\gg k$;
- Facendo lo shift, k cifre scompaiono e compaiono k nuovi 0 a destra ($\ll k$) o a sinistra ($\gg k$); se i bit rappresentano un numero naturale e non vengono cancellati 1, $\ll k$ equivale ad una moltiplicazione per 2^k



Moltiplicatore:

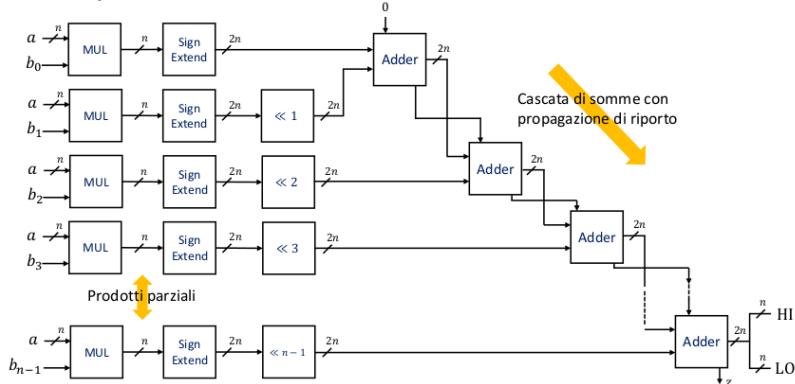
Moltiplicatore

- La moltiplicazione binaria si organizza in due fasi
 - Calcolo dei prodotti parziali: AND a coppie di bit in posizione shiftata
 - Somma bit a bit (considerando anche i riporti) dei prodotti parziali, useremo dei Full Adder

$$\begin{array}{r}
 10111 \times a \\
 101 = b
 \end{array}
 \quad \text{Prodotti parziali}
 \quad \begin{array}{l}
 b_0 \times a \rightarrow 1011 \\
 b_1 \times a \rightarrow 0000 \\
 b_2 \times a \rightarrow 1011 \\
 \hline
 1110011 \quad a \times b
 \end{array}
 \quad \begin{array}{l}
 b_0a_4 \ b_0a_3 \ b_0a_2 \ b_0a_1 \ b_0a_0 \\
 b_1a_4 \ b_1a_3 \ b_1a_2 \ b_1a_1 \ b_1a_0 \\
 b_2a_4 \ b_2a_3 \ b_2a_2 \ b_2a_1 \ b_2a_0
 \end{array}
 \quad \downarrow \text{Somma}$$

- In generale il prodotto di 2 numeri su n bit, può dare un risultato su $2n$ bit, di norma si suddivide il risultato in due numeri separati: con HI si indicano gli n bit della parte alta (indicano anche se c'è un overflow), con LO gli n della parte bassa

Moltiplicatore $n \times n$

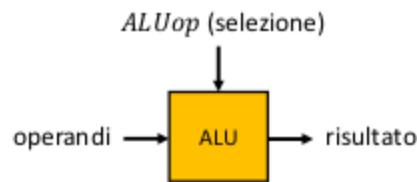


ALU:

Componente hardware incaricato di svolgere operazioni di tipo aritmetico (ad esempio, somme, sottrazioni, etc. ..) oppure di tipo logico (ad esempio AND e OR)

Nel ciclo di esecuzione di un programma, è il componente incaricato della fase di execute.

Possiamo vederla come un circuito combinatorio multifunzione che riceve in input degli operandi e un codice di selezione "ALUop", che indica l'operazione richiesta alla ALU, e restituisce il risultato dell'operazione richiesta, applicata agli operandi.



Logica di progettazione:

- Modulare: una ALU da n bit (per operandi e risultato) si progetta componendo ALU da 1 bit (moduli)
- Parallelia: dati gli operandi, la ALU calcola internamente tutte le funzioni di cui è capace, la selezione (multiplexer) ne porrà in uscita una sola

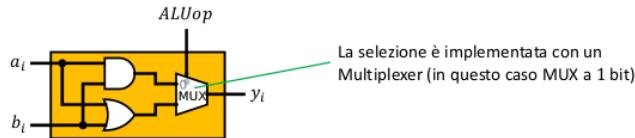
La ALU che implementeremo, supporterà le seguenti operazioni:

- AND (logica)
- OR (logica)
- Somma (aritmetica)
- Sottrazione (aritmetica)

- Comparazione (logica)
 - Test di uguaglianza allo zero (logica)
-

AND e OR:

- Iniziamo col progettare una ALU elementare ad 1 bit in grado di svolgere le operazioni logiche di AND e OR tra due operandi di 1 bit che indichiamo con a_i e b_i

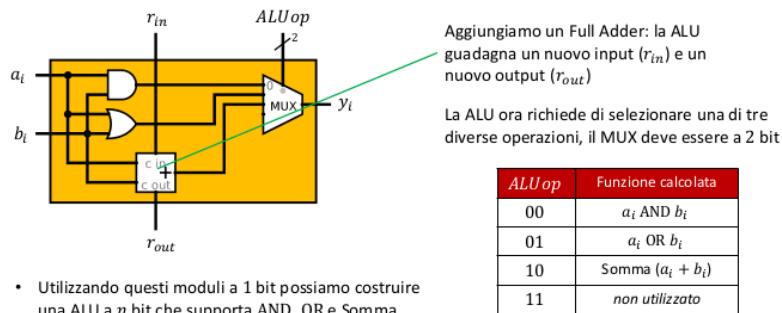


- Parallelismo: la ALU calcola sempre sia AND che OR, ma solo una delle due viene posta in uscita attraverso la selezione
- $ALUop$ è, in questo caso, il singolo bit di selezione:
se vale 0 in uscita avremo $a_i b_i$, se vale 1 avremo $a_i + b_i$

$ALUop$	Funzione calcolata
0	$a_i \text{ AND } b_i$
1	$a_i \text{ OR } b_i$

ADD:

- Estendiamo la ALU appena realizzata in modo che supporti anche la somma, sempre su 1 bit



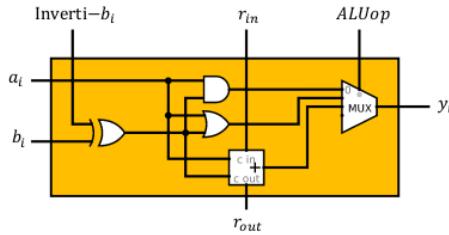
- Utilizzando questi moduli a 1 bit possiamo costruire una ALU a n bit che supporta AND, OR e Somma

$ALUop$	Funzione calcolata
00	$a_i \text{ AND } b_i$
01	$a_i \text{ OR } b_i$
10	Somma ($a_i + b_i$)
11	non utilizzato

SUB:

Sottrazione (SUB)

- Per supportare la sottrazione devo aggiungere la possibilità, in ogni full adder, di complementare a 1 l'operando b e aggiungere 1 alla somma
- Abbiamo già visto come fare! Aggiungo un nuovo bit di controllo: Inverti- b_l , se questo segnale viene posto a 1 l'operando b_l viene sostituito con \bar{b}_l (il suo complemento a 1)
- Posso fare la sottrazione settando Inverti- b_l e r_{in} entrambi a 1

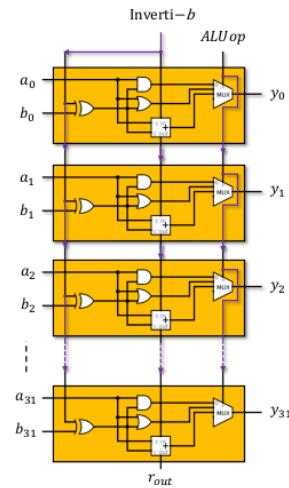


(Per complementare a 1, uso lo XOR per invertire il bit B, e lo stesso segnale invertiB va dentro il primo riporto)

AND, OR, ADD e SUB su 32 bit

- Collegiamo le 32 ALU da 1 bit usando sempre lo schema di propagazione dei riporti
- ALUop: esattamente come prima (2 bit, lo stesso per tutte le ALU da 1 bit)
- Inverti- b : è lo stesso in tutte le ALU ed è anche collegato al primo r_{in}
- Settando Inverti- b a 1 ottengo simultaneamente che:
- Tutti i bit di b vengono invertiti quindi dentro la ALU b viene subito trasformato in \bar{b}
- Il primo riporto in ingresso è 1 quindi i sommatori svolgono $1 + a + \bar{b}$ che in C2 significa $a - b$

Inverti- b	ALUop	Funzione calcolata
0	00	$a \text{ AND } b$ (bitwise)
0	01	$a \text{ OR } b$ (bitwise)
0	10	Somma ($a + b$)
1	10	Sottrazione ($a - b$)
*	11	non utilizzato



Comparazione (Set Less Than):

Alla ALU di prima, aggiungo un nuovo input "LESS", che verrà passato all'uscita quando su ALUop pongo il valore 11.

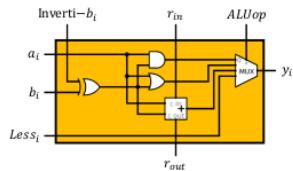
Questo nuovo input parte dal risultato dell'ultimo sommatore nell'ultima ALU, e va direttamente nel MUX della prima ALU.

In ogni ALU a parte la prima poi, metto uno 0 che va direttamente nel MUX.

Quindi quando ALUop è settato a 11, la configurazione o sarà 000....0 ($a \geq b$) oppure 000....1 ($a < b$)

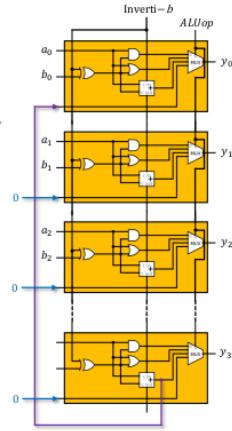
Comparazione (SLT)

- Estendo il modulo ALU da 1 bit aggiungendo un nuovo input $Less_i$
- Questo bit viene passato sull'uscita quando il selettore $ALUop$ vale **11**, cioè la configurazione di selezione che fino ad ora era *non utilizzata* e che ora **assegniamo a SLT**



Per fare la comparazione:

- Setto $Less_1, Less_2, \dots, Less_{n-1}$ a **0**
- Setto $Inverti-b$ a **1**, così i sommatori svolgono $a - b$
- Setto $Less_0 = s_{n-1}$ (il bit di segno del risultato di $a - b$)
- Setto $ALUop$ a **11**



(Per fare la comparazione, ogni alu deve eseguire la sottrazione bitwise, quindi invertiB va posto a 1).

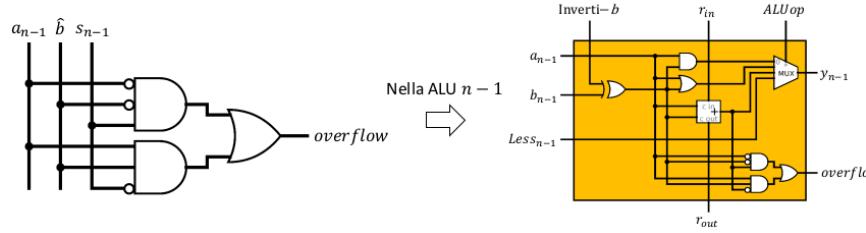
Overflow:

La presenza di un overflow, si identifica in due modi:

- Gli ultimi due riporti sono diversi
- Sommando due numeri positivi ottengo un negativo, o sommando due negativi ottengo un positivo.

Per implementare il controllo dell'overflow, introduco un circuito che tramite il secondo metodo, controlla i due bit di segno degli operandi, e li confronta con il bit di segno del risultato.

- Possiamo implementare il secondo metodo con un semplice circuito combinatorio che prende in input i bit di segno di a, b e del risultato della somma s : questi bit stanno tutti nella ALU $n - 1$!
- ATTENZIONE!** Nella ALU $n - 1$ il bit di segno di b non è b_{n-1} ma $b_{n-1} \oplus \text{Inverti-}b = \hat{b}$

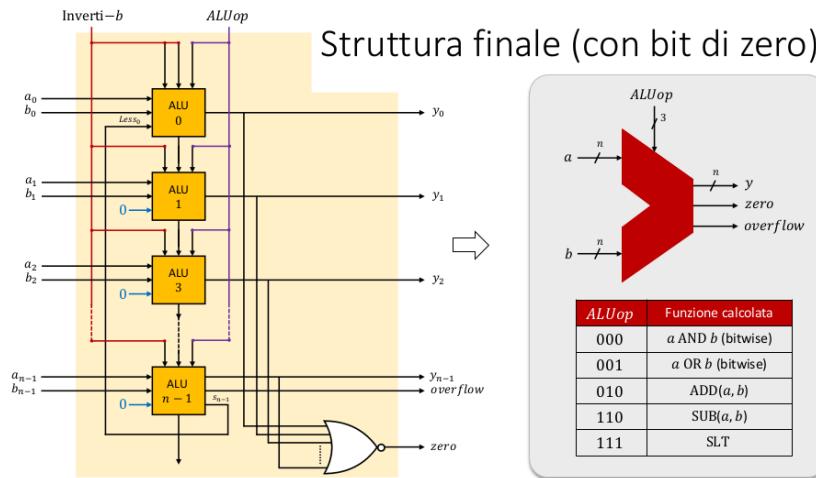


L'ultima ALU quindi, avrà struttura come a destra.

Gli input sono A, B complementato a 1 (preso dopo lo XOR), e il risultato del sommatore.

La struttura dell'ALU, che opera su più bit in serie, ha struttura come sotto indicata, e "impacchettata" in un componente che si disegna con il simbolo a destra.

In base al valore di "ALUop", che può avere solo i 5 valori indicati, eseguo un'operazione diversa sui due numeri a N bit in ingresso.



Per il controllo dello zero, metto in NOR tutte le uscite delle singole ALU.

ALU per operazioni floating point:

La ALU implementata precedentemente, è in grado di eseguire operazioni solo su valori interi.

Dati due numeri in formato IEE754, ho bisogno di un circuito più complesso per computarli.

Questo circuito si chiama ALU-FP, specifica per eseguire operazioni in virgola mobile.

Per capirne il funzionamento (non ne vedremo l'implementazione)

- Come si svolge $99.5 + 0.85$?
 - $99.5 + 0.85$ normalizzando diventa: $9.95 \times 10^1 + 8.5 \times 10^{-1}$
 - Allineo gli esponenti de-normalizzando (temporaneamente) il numero con esponente più basso: $9.95 \times 10^1 + 0.085 \times 10^1$
 - Sommo le mantisse: $9.95 + 0.085 = 10.035$
 - Normalizzo il risultato: 1.0035×10^2
-

PLA:

Dispositivi logici programmabili, utilizzati per l'implementazione hardware dei circuiti combinatori.

Di per se, non fa nulla, è un componente "grezzo", ha bisogno di essere programmato appositamente per svolgere una funzione logica.

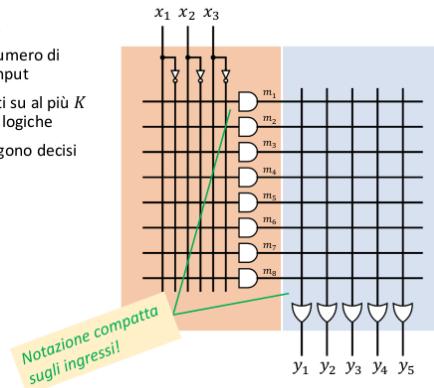
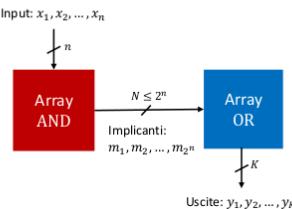
Una volta programmato, mantiene il suo funzionamento, non può essere riprogrammato.

Una PLA, al suo interno, rappresenta le operazioni logiche in prima forma canonica (SOP), quindi tramite AND e OR.

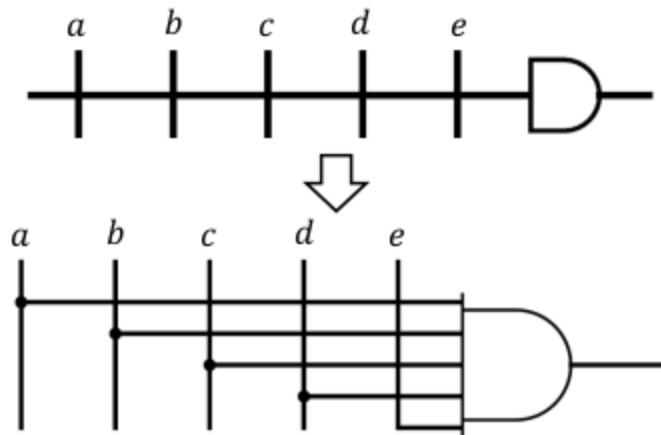
Il Layout della PLA si basa infatti su due stadi (array), AND e OR.

L'array AND, viene programmato per calcolare N implicanti (ES: nella funz: ABC+DFH, ABC è un implicante), che è al più due alla x dove x è il numero di input (pensare a righe mappa di karnaugh, 2 alla numero di input righe, quindi al massimo 2 alla n uscite a 1).

- Il layout a due stadi si basa su due array: AND e OR
- L'array AND viene programmato per calcolare un numero di implicant N che è al più 2^n dove n è il numero di input
- L'array OR consente di sommare gruppi di implicant su al più K uscite, quindi possiamo codificare al più K funzioni logiche
- n , N e K sono parametri dimensionali del PLA, vengono decisi durante la fabbricazione del dispositivo



Nello schema a destra, gli ingressi sono indicati in notazione compatta, con un filo che collega tutti gli ingressi.

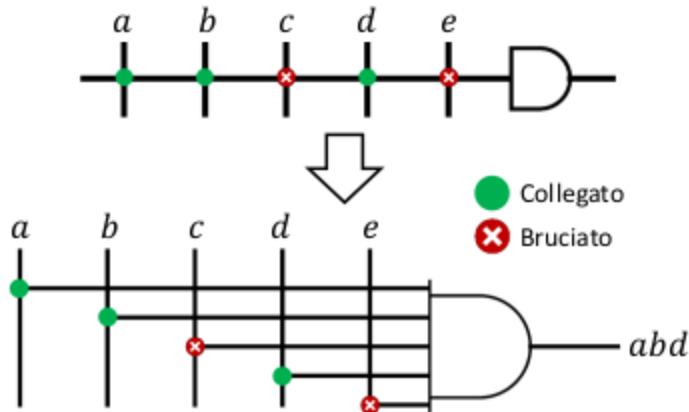


Siccome le "variabili" negli implicant possono apparire in forma naturale o negata, notiamo che per ogni input X, il valore è "sdoppiato", in X e X_{negato} .

Per programmare un PLA, scollego (brucio i fusibili associati) e lascio collegati, gli ingressi negli AND e negli OR, in base a ciò che voglio calcolare.

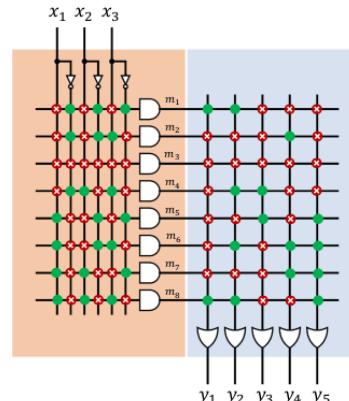
Esempio: Per calcolare il solo AND tra ABD, lasciando C ed E scollegati:

- Bruciando alcuni collegamenti e lasciandone altri collegati (viene di solito specificato tramite una matrice)
- Tecnologia possibile: un fusibile (o antifusibile) su ogni collegamento, applicando una tensione abbastanza alta «brucio» il fusibile e quindi blocca il collegamento



- **Esempio** di implementazione

x_1	x_2	x_3	y_1	y_2	y_3	y_4	y_5
0	0	0	1	1	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	0	0	0
0	1	1	0	1	1	0	0
1	0	0	0	0	1	0	1
1	0	1	0	1	0	1	1
1	1	0	0	0	0	1	1
1	1	1	1	1	0	0	1

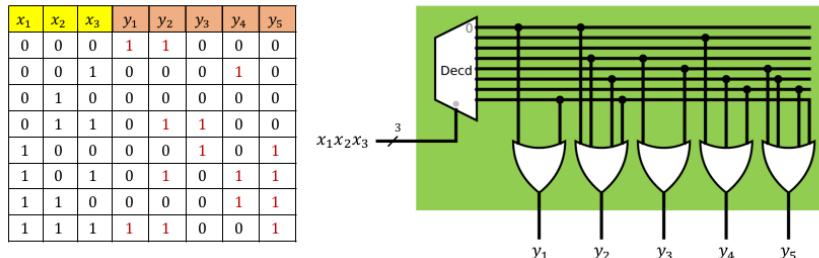


Nota: il mintermine M_3 , è inutile calcolarlo, perchè vale 0, quindi non saldo nessun input perchè sarebbe inutile calcolarlo.

ROM (Read Only Memory):

L'Input (X_1, X_2, X_3), è visto come l'indirizzo di memoria a cui sono memorizzati gli Y bit delle uscite.

Nella tabella quindi, ho per ogni riga, gli indirizzi delle celle di memoria (su 3 bit, infatti la ROM implementata ha 8 celle di memoria), e i valori associati ad ogni cella.



Il Decoder accende la linea indicata dall'input.

ES: 111 in ingresso, accendo la linea 7.

E tutte le uscite che hanno la linea 7 in OR, si accendono.

ES: Se ho in input 011, si accenderà la linea 3, e la linea 3 è in OR a y_2 e y_3 , che si accenderanno.

L'Array AND è hardwired, genera sempre tutti i 2^n mintermini possibili.

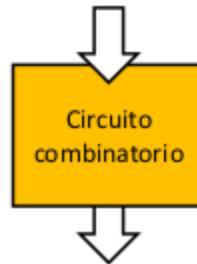
L'Array OR è programmabile.

Dai circuiti combinatori a quelli sequenziali:

Nei circuiti combinatori, dati X input, al tempo T avrò il solo output dell'ultimo input.

Esempio: Telecomando della macchina, se schiaccio apri chiudi chiudi apri apri chiudi apri, al tempo T, avrò la sola macchina aperta, quindi l'ultimo tasto che ho premuto, non mi interessa degli input di mezzo.

$x^{(t)}$ Input al tempo t



Output al tempo t
(in realtà $t + C \times \Delta_p$)
Dipende solo da $x^{(t)}$

Nei circuiti sequenziali invece, l'output al tempo T , può dipendere da tutta o parte della sequenza degli input.

Esempio: telecomando della TV quando cambio canale, per andare al canale 150, premo prima l'1, poi il 5 e poi lo 0, ma al tempo T l'output non è solo 0, ma è dato dal totale della sequenza degli input.

Per fare ciò, il circuito sequenziale, a differenza di quelli combinatori, hanno bisogno di memoria, quindi di ricordare gli input precedenti.

I Circuiti sequenziali si dicono quindi "a stato", cioè il ricordo del passato da parte del circuito, cioè la sequenza di input che ha visto fino ad ora.

I Circuiti combinatori possono essere visti come circuiti sequenziali a un solo stato.

Sulla carta possiamo scrivere che il funzionamento di un circuito combinatorio, produce in output una funzione $y \leftarrow f(x)$.

Dato un input x posso determinare l'uscita y e a parità di input l'uscita è sempre la stessa

In un circuito sequenziale invece $y, s_{next} \leftarrow f(x, s)$, cioè dati input e stato corrente:

- L'uscita y dipende sia dall'input x che dallo stato corrente s : a fronte di uno stesso input

posso ottenere uscite diverse se lo stato corrente del circuito è diverso

- Oltre a calcolare un'uscita, il circuito effettua una transizione di stato passando dallo stato

corrente s allo stato successivo s_{next} (in certi casi può succedere che $s_{next} = s$)

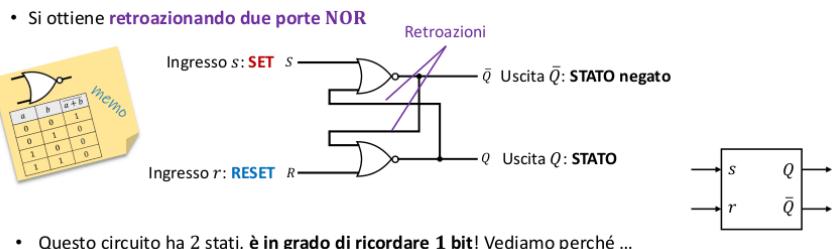
Per poter ricordare qualcosa, un circuito deve avere ≥ 2 stati possibili, altrimenti se avesse uno stato solo, sarebbe un circuito combinatorio.

Un circuito sequenziale, deve avere inoltre "retroazione", cioè poter tornare ad uno stato precedente.

Bistabile SR (Latch):

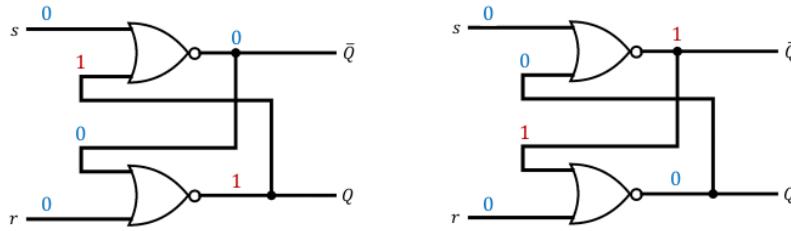
Il Primo circuito sequenziale è il bistabile, che si implementa applicando la retroazione a due porte NOR.

Dati due input Set e Reset, ho due output, Stato e Stato Negato (i due output saranno sempre uno il negato dell'altro, mai 00 e mai 11).



Una copia dei segnali di uscita Q e Q negato, ritorna indietro, e diventa input delle due porte NOR.

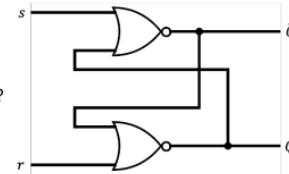
Il Bit di uscita (che poi ritorna in input), è lo stato.



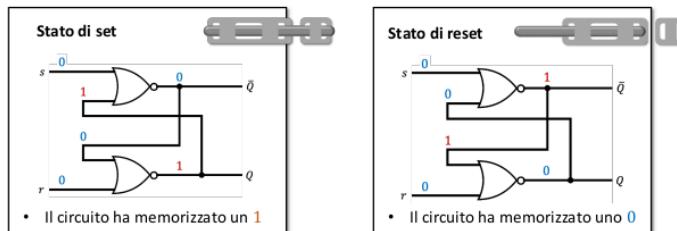
- Pongo a **1** il segnale di set s
 - L'uscita Q passa a **1** (\bar{Q} passa a zero **0**)
 - Riparto s a **0**: l'uscita Q resta **stabile** a **1**!
 - Come se ci fosse un **1** «impresso» nella struttura intrinseca del circuito: è lo **stato**: il circuito sta **ricordando** l'evento di set avvenuto nel passato
-
- Pongo a **1** il segnale di reset r
 - L'uscita Q passa a **0** (\bar{Q} passa a **1**)
 - Riparto r a **0**: l'uscita Q resta **stabile** a **0**!
 - Come se ci fosse uno **0** «impresso» nella struttura intrinseca del circuito: è lo **stato**: il circuito sta **ricordando** l'evento di reset avvenuto nel passato

Bistabile Set-Reset (SR)

- E se poniamo a **1** entrambi i segnali s e r contemporaneamente?
- Fintanto che $s = r = 1$, entrambe le uscite valgono a **0**
- $Q = \bar{Q} = 0$ perdiamo la semantica dei nomi delle uscite!
- Ripetendo lo stesso ragionamento di prima, cosa succede quando torno a $s = r = 0$?
- I due segnali non commutano contemporaneamente, c'è sempre un piccolo Δt tra i due:
 - Se $s \rightarrow 0$ prima che $r \rightarrow 0$, ho un reset ($Q \rightarrow 0$, $\bar{Q} \rightarrow 1$)
 - Se $r \rightarrow 0$ prima che $s \rightarrow 0$, ho un set ($Q \rightarrow 1$, $\bar{Q} \rightarrow 0$)**Lo stato di arrivo è imprevedibile!**
- $s = r = 1$ si può fare ma perdiamo l'interpretazione di stato e affidiamo al caso parte del comportamento del circuito. **Quindi eviteremo di farlo!**



Bistabile Set-Reset (SR)



- Il circuito si chiama **bistabile** perché riesce a mantenere stabilmente uno tra due diversi stati (set e reset), viene anche chiamato **latch** (chiavistello)
- Notiamo nelle due figure: stesso circuito, stessi input, **ma uscite diverse**! Sarebbe impossibile con un circuito combinatorio (senza retroazioni)
- Abbiamo costruito una memoria a 1 bit, il bit memorizzato è visibile in uscita (Q)

Bistabile Set-Reset (SR)

- Un circuito sequenziale ammette una tabella di verità?
- La tabella di verità non è più sufficiente, dobbiamo fornire un formalismo più generale: tabella delle transizioni

	Input <i>sr</i>				
	00	01	10	11	
Stato <i>Q</i>	0	0	0	1	non usato
	1	1	0	1	non usato

<i>s</i>	<i>r</i>	<i>Q</i>	<i>Q_{next}</i>
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	<i>x</i>
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	<i>x</i>

- In alternativa possiamo considerare in modo esplicito il tempo e vedere lo stato prossimo come una funzione logica di input e stato corrente
- Possiamo in questo modo comporre una tabella delle transizioni fatta come una tabella di verità
- **Attenzione!** *Q* e *Q_{next}* sono lo stesso segnale ma in tempi diversi (corrente, successivo)

Bistabile Set-Reset (SR)

- Usando questa interpretazione combinatoria della transizione da stato corrente a stato prossimo, posso sintetizzare la funzione stato prossimo $Q_{next} = T(s, r, Q)$
- Dato input (*s* e *r*) e stato corrente (*Q*) calcola lo stato prossimo (*Q_{next}*)
- Come se *Q* e *Q_{next}* fossero segnali diversi, anche se sappiamo che non lo sono

<i>s</i>	<i>r</i>	<i>Q</i>	<i>Q_{next}</i>
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	<i>x</i> → 1
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	<i>x</i> → 1

$$\begin{aligned}
 T(s, r, Q) &= s\bar{r}\bar{Q} + sr\bar{Q} + \bar{s}\bar{r}Q + s\bar{r}Q + srQ \\
 &= s\bar{r}\bar{Q} + sr\bar{Q} + \bar{s}\bar{r}Q + s\bar{r}Q + s\bar{r}Q + srQ \\
 &= s(\bar{r}\bar{Q} + r\bar{Q} + \bar{r}Q + rQ) + \bar{r}Q(s + \bar{s}) \\
 &= s + \bar{r}Q
 \end{aligned}$$

Circuiti sequenziali sincroni:

Oltre a memorizzare i risultati delle elaborazioni, i circuiti sequenziali, possono essere utilizzati per segmentare l'elaborazione di circuiti combinatori molto complessi.

Un Circuito combinatorio complesso infatti, richiede un cammino critico elevato prima di produrre un'uscita stabile.

Se memorizzo i risultati intermedi in dei circuiti sequenziali, e sincronizzo i passaggi, ottengo un circuito equivalente.

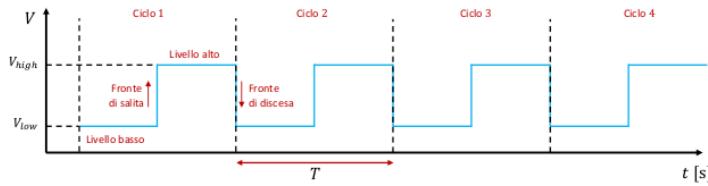
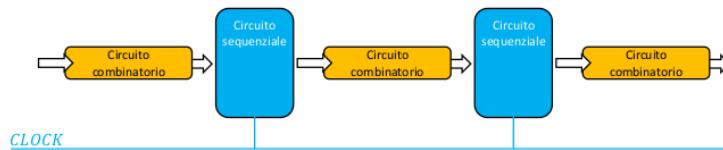
- Eseguire l'elaborazione per passi, un sotto-circuito dopo l'altro in sequenza: quando un sotto circuito ha completato (le sue uscite sono stabili), il successivo legge in input le uscite del precedente e procede
- Come si ottiene questa elaborazione «a staffetta»? **Salvando** i risultati intermedi e **sincronizzando** i passaggi



- Reparti di stoccaggio/smistamento: memorizzano il risultato proveniente da sinistra e lo rendono disponibile a destra come in una catena di montaggio

La sincronizzazione, è applicata grazie ad un segnale detto "di clock".

- I circuiti sequenziali possono essere immaginati come dei reparti di stoccaggio e smistamento con due cancelli automatici, uno a destra e uno a sinistra
- Il clock è un segnale in grado di aprire e chiudere i cancelli
- Sincronizza l'elaborazione dicendo a ciascun reparto
 - quando aprire il cancello di sinistra e ricevere un dato dal circuito combinatorio
 - quando aprire quello di destra per passare il dato al circuito combinatorio successivo



- Il periodo T (in secondi s) misura la lunghezza temporale di un ciclo di clock, va dimensionato in modo che la logica combinatoria abbia il tempo necessario per commutare in modo stabile le uscite
- La frequenza di clock (in hertz Hz, cicli al secondo) è l'inverso del periodo $f = \frac{1}{T}$
- I livelli alto e basso del clock corrispondono a valori alti (V_{high}) e bassi (V_{low}) di tensione del segnale che rappresentano l'**1** e lo **0** logico

In un'architettura sincrona, i circuiti sono "sensibili" al segnale di clock, cioè cambiano stato solo quando si verifica una certa condizione sul clock.

- Architetture sensibili ai livelli:

Il circuito cambia stato quando il livello del clock è alto o basso.

Quando il clock è alto o basso (opposto), non avvengono variazioni di stato, ma i segnali restano stabili, e l'elaborazione continua senza problemi.

- Architetture sensibili ai fronti:

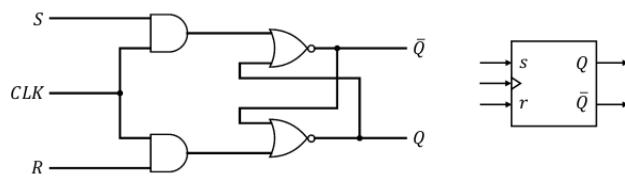
Le variazioni di stato avvengono all'attimo in cui il clock passa da 0 a 1 o viceversa.

Bistabile SR Sincrono (Latch sincrono):

Una variante del Bistabile che "obbedisce" al clock, può essere realizzata mettendo il clock in and con Set e Reset.

Bistabile Set-Reset (SR) sincrono

- Progettiamo una variante del bistabile SR che «obbedisce» al clock (CLK)



- Sensibile al livello
 - Se $CLK = 0$ si forza lo stato di riposo, lo stato non può cambiare, il cancello è chiuso
 - Se $CLK = 1$ allora si può fare un set o un reset per cambiare lo stato, il cancello è aperto

		Input sr			
		00	01	11	10
Stato Q Clock CLK	00	0	0	0	0
	01	1	1	1	1
	11	1	0	1	1
	10	0	0	1	1

$T(s, r, q, CLK) = \overline{CLK}Q + CLKs + CLKQ\bar{r}$
 $= CLKQ + CLK(s + \bar{r}Q)$

Gli input sono ordinate secondo codifica di gray (00,01,11,10).

Notare che gli implicanti (rettangoli della mappa di Karnaugh), non sono i massimi, ma permettono comunque di scrivere la funzione, facendo comparire in essa la funzione del bistabile "normale".

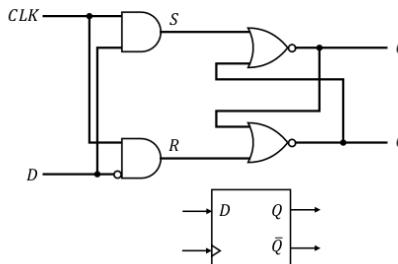
Latch D:

Tra le condizioni di input, la combinazione 11 è invalida, nel senso che se faccio set e reset dello stato.

Posso forzare il mio circuito ad accettare solo combinazioni di S e R opposte (01,10), con la variazione del circuito della "Latch D".

Cioè portando uno stesso segnale a S e R, mettendo uno dei due in "not".

- Introduciamo una semplice miglioria nel bistabile SR sincrono
- Nel bistabile abbiamo due input s e r , ma sappiamo che ne usiamo sempre uno per volta
- Possiamo raggrupparli in un unico input D



- $D = 1$ corrisponde a SET
- $D = 0$ corrisponde a RESET
- D sta per «dato», quando il clock è alto D viene scritto dentro il bistabile
- Anche in questo caso abbiamo sensibilità sul livello: per tutto il tempo in cui il clock è alto lo stato può cambiare

Nella figurina del circuito (il quadrato con D Q e Q negato), il segnale sotto a D, rappresentato con la parentesi angolare, è il clock.

		Input D	
		0	1
Clock CLK	00	0	0
	01	1	1
	11	0	1
	10	0	1

$T(D, q, \text{CLK}) = \overline{\text{CLK}}q + \text{CLK}D$

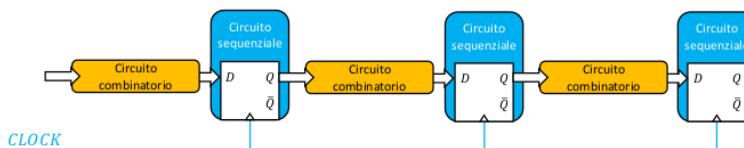
- Se il clock è basso si conserva lo status quo, Q
- Se il clock è alto D viene scritto nel bistabile



Attenzione! Questa variazione non ammette la combinazione 00 ne 11.

Flip Flop:

- Possiamo utilizzare il latch D nell'architettura sincrona che abbiamo introdotto?

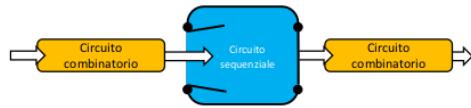


- La sensibilità sul livello crea il problema della **trasparenza**
- Quando il clock vale 0 tutti i latch mantengono lo stato corrente, niente cambia, le uscite sono stabili
- Quando il clock vale 1 tutti i latch sono sensibili a cambiamenti di stati
- Tutti i cancelli sono aperti! Perdiamo la separazione tra stadi e il segnale può attraversare tutto il circuito da sinistra a destra: riotteniamo il problema del cammino critico

Idea: costruire dei circuiti sequenziali con cancello doppio

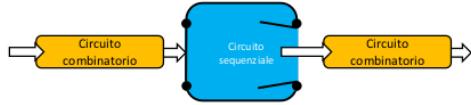
Primo step (clock alto):

- il cancello di sinistra si apre
- il dato proveniente dal primo circuito combinatorio entra nel circuito sequenziale e viene memorizzato
- Il cancello di destra è chiuso! La separazione tra stadi tiene



Secondo step (clock basso):

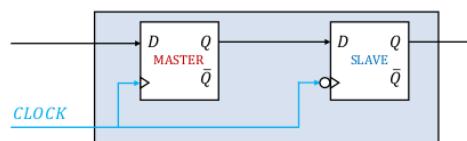
- il cancello di destra si apre
- Il dato che era stato memorizzato precedentemente viene mandato in uscita, il secondo circuito combinatorio lo riceve in input
- Il cancello di sinistra è chiuso! Il circuito precedente completa la sua elaborazione stabilizzandosi



Il circuito sequenziale che implementa questa idea si chiama **Flip Flop** e combina 2 latch D

Flip Flop

- Due latch D collegati in serie in una configurazione master-slave, dove il latch slave riceve il clock negato



• **CLOCK → 1: FLIP**

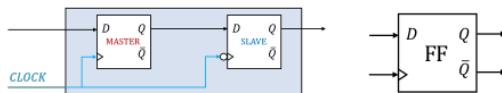
- **master aperto**, il dato in input viene scritto
- **slave isolato**, continua a mettere in uscita il suo stato corrente (spoiler: è il dato che era contenuto nel master prima del FLIP)

• **CLOCK → 0: FLOP**

- **master isolato**, mette in uscita il suo stato corrente (il dato memorizzato durante il FLIP)
- **slave aperto**, in un tempo quasi istantaneo lo stato del master viene copiato nello slave che inizia subito a porlo stabilmente in uscita

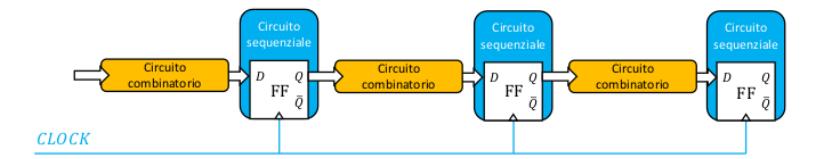
Flip Flop

- Il Flip-Flop è un circuito sensibile ai fronti



- L'uscita commuta in modo istantaneo sul fronte di discesa del clock e resta stabile fino al prossimo fronte di discesa quindi per tutto il ciclo di clock

- L'architettura sincrona procede per step, uno per ogni ciclo di clock



Macchine a stati finiti (FSM, Finite State Machines):

La Macchina a stato finito, è un modello matematico dell'elaborazione.

Si assume che l'elaborazione, sia definita come "le operazioni che una macchina compie", cioè:

- Riceve informazioni in input (finiti)
- Ha uno stato interno (in numero finito)
- Emette delle informazioni in uscita.

Se consideriamo queste tre operazioni, il flip flop e anche il bistabile, sono delle macchine a stati finiti (Ricevono due input, hanno uno stato (0 o 1), ed emettono in uscita un valore, cioè la copia dello stato interno).

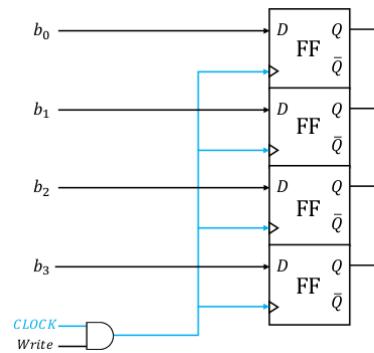
Ma se gli stati assumibili dalla macchina fossero più di due? Nel caso di Flip Flop e Bistabile, lo stato può essere sempre 0 o 1, ma se avessi più combinazioni?

Per avere un circuito sequenziali con più di due stati, ho bisogno di:

- Più memoria, cioè per N stati ho bisogno di log in base 2 di N bit (per 8 stati, ho bisogno di $2^3=3$ bit).
- Ci serve un formalismo più complesso per descrivere la dinamica del circuito.

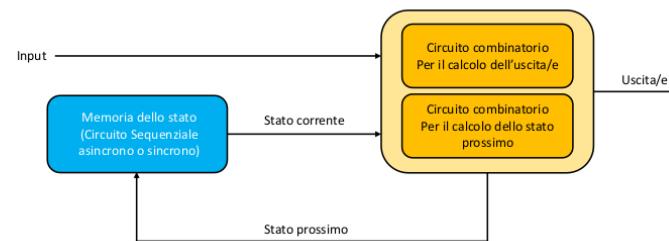
Per risolvere il problema della memoria, metto in serie N flip flop, uno per ogni bit di memoria, per creare un Registro a N bit.

- Come si costruisce una memoria su più bit?
- Usando n Flip Flop, costruisco una registro da n bit
- **Esempio:** registro su 4 bit con segnale di write (opzionale)



Il Segnale di Write, permette di isolare l'azione di scrittura sul registro.

- Adottiamo uno schema più generale per un circuito sequenziale



- Esempio: il bistabile SR può essere implementato come un caso particolare di questo schema in cui:
 - Input: s, r
 - Circuito per il calcolo delle uscite: lo stato Q (filo) e lo stato negato \bar{Q} (NOT)
 - Circuito per lo stato prossimo: $s + \bar{r}Q$
 - Memoria: 1 bit

Dal punto di vista matematico, una FSM prevede:

- Un insieme finito degli stati ($X = \{x_1, x_2, x_3, \dots\}$)
- Uno stato iniziale x_0 appartenente a X .
- Un insieme degli input che la macchina può ricevere ($I = \{i_1, i_2, i_3, \dots\}$)
- Un insieme degli output che la macchina può generare ($O = \{o_1, o_2, o_3, \dots\}$)
- Funzione di transizione allo stato prossimo $T(x, i)$
- Funzione di produzione dell'uscita $g(x)$ o $g(x, i)$, in base a se l'uscita dipende dal solo stato corrente (Macchina di Moore) o anche dagli input (Macchina di Mealy)

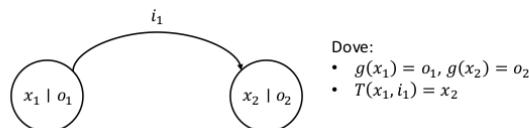
Nel bistabile per esempio, in cui l'uscita è una copia dello stato corrente, la funzione di uscita è $g(x)$.

Una FSM può anche essere rappresentata mediante un grafo:

- Ogni nodo rappresenta uno stato x della FSM e l'uscita associata $g(x)$

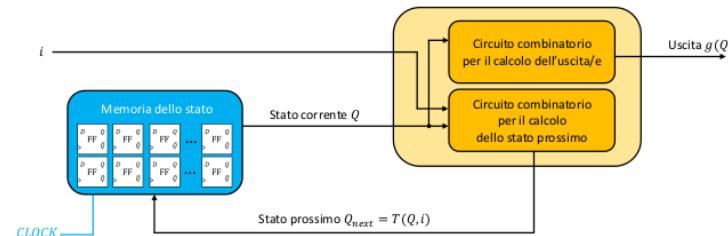


- Ogni arco rappresenta una transizione tra stati



Architettura del circuito di una FSM

- Cominciamo dalla prima domanda: **come è fatto il circuito che implementa la FSM?**
- Useremo lo schema introdotto in precedenza, che ora possiamo descrivere in modo più specifico:



- Cosa ci serve per costruire un circuito di questo tipo?
 1. Sapere da quanti Flip Flop è costituita la memoria dello stato
 2. Sapere come è fatto il circuito per il calcolo dell'uscita
 3. Sapere come è fatto il circuito per il calcolo dello stato prossimo

Approccio Firmware:

Per costruire un circuito moltiplicatore per esempio, possiamo procedere secondo due approcci:

- Combinatorio → L'Operazione è eseguita tutta insieme, in un solo ciclo di clock.
- Sequentiale → Descrive una macchina a stati finiti, la moltiplicazione suddivisa in operazioni elementari, eseguite in sequenza una per ogni cicli di clock.

La sequenza di esecuzione, corrisponde all'**algoritmo** dell'operazione.

L'Approccio combinatorio, "costa di più", ma performa meglio.

Quello sequenziale invece, è meno performante ma meno costoso e più flessibile.



Firmware = Software non mutabile, una volta scritto in memoria non è riscrivibile (o se lo è, richiede particolari procedure per essere modificato).

Approccio sequenziale applicato alla moltiplicazione:

Realizzazione di una FSM che segue un approccio firmware, cioè può eseguire una sola operazione, la moltiplicazione, non è general purpose.

La Moltiplicazione, può essere pensata come una "somma ripetuta", cioè se dico $5*3$, è l'equivalente di dire $5+5+5$.

$$\begin{array}{r}
 n = 5 \\
 \begin{array}{r}
 a \longrightarrow 10111 \times \\
 b \longrightarrow 00101 =
 \end{array} \\
 \hline
 & 111 \\
 k = 0 & \longrightarrow 10111 \\
 k = 1 & \longrightarrow 00000 \\
 k = 2 & \longrightarrow 10111 \\
 k = 3 & \longrightarrow 00000 \\
 k = 4 & \longrightarrow 00000 \\
 \hline
 p = a \times b & 001110011
 \end{array}$$

L'Algoritmo della moltiplicazione prevede:

- Un Moltiplicando, espresso su N bit. (A)
- Moltiplicatore a N bit (B)
- Prodotto, su 2N bit (Contiene ad ogni iterazione, la somma dei prodotti parziali. All'ultima iterazione conterrà il prodotto A*B).
- K - Contatore iterazioni (Come se pensassimo alla moltiplicazione come fosse un "for" che somma)

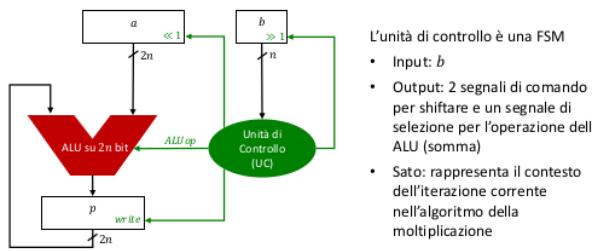
Step per eseguire l'operazione:

- Inizializzazione

Assegno 0 a "p" e "k".

Estendo il segno del moltiplicando, su 2N bit, esattamente come la variabile P, così che possa sommare il moltiplicando a "p".

- Controllo il bit più a destra del moltiplicatore, se esso è 1, aggiungo il moltiplicando "A" al risultato "P".
- Shift a sinistra il moltiplicando, lo preparo ad un eventuale prossima somma. (A)
- Shift a destra il moltiplicatore. (B)
- Aumento il valore di K.
- Ripeto dal punto 2, l'algoritmo termina quando con gli shift ho esaurito i bit del moltiplicatore (B)



Per l'implementazione del circuito, utilizzo dei registri a N e 2N bit.

Questi registri sono particolari registri detti "programmabili", cioè possono eseguire una specifica operazione sul dato in esso memorizzato.

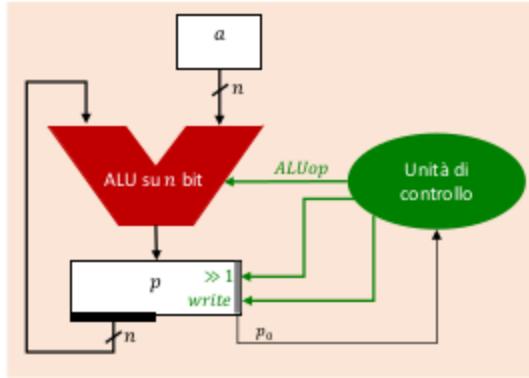
In Questa implementazione, i registri sono programmati in modo da poter eseguire operazioni di shift a sinistra per il registro che contiene il moltiplicando, e shift a destra per il registro che contiene il moltiplicatore.

L'ALU su 2N bit, è incaricata di eseguire l'operazione " $a+p$ ", per B volte.

L'Unità di controllo, è una FSM che coordina l'esecuzione dell'algoritmo, può comandare le operazioni di shift su A e B, e comanda la ALU, che esegue una diversa operazione in base al valore del segnale ALUop (valori diversi per addizione, sottrazione, moltiplicazione ecc...)

Ad ogni ciclo di clock, la UC guarda il bit B0, e in base al suo valore, sceglie se aggiornare il valore di P (sommare, mettendo il segnale ALUop al codice corrispondente alla moltiplicazione, indicando alla ALU di eseguire la somma, e Write a 1, aggiornando il valore di P) oppure no, effettua poi lo shift di A e B.

Nell'implementazione, può essere realizzata una versione "ottimizzata", introducendo una ALU a N bit, al posto di 2N, facendo shiftare P rispetto ad A, ed eliminando il registro che contiene B, precaricando nella parte bassa del registro P (n bit meno significativi), il moltiplicatore.



Approccio sequenziale applicato alla divisione:

Dato a il dividendo, e b il divisore:

$$\begin{array}{r}
 \overbrace{7 \ 4}^{\text{red}} \overbrace{8 \ 9}^{\text{green}} \mid 3 \ 2 \\
 -6 \ 4 \\
 \hline
 1 \ 0 \ 8 \\
 -9 \ 6 \\
 \hline
 1 \ 2 \ 9 \\
 -1 \ 2 \ 8 \\
 \hline
 1
 \end{array}$$

- Considero, da sinistra a destra, il minimo numero di cifre del dividendo che formano un numero $r^a \geq b$
- Determino q, pari a quante volte b sta in r^a , e scrivo q nella prossima cifra del quoziente

- Ci sono altre cifre del dividendo da considerare?
 1. Si - Calcolo $r^{\wedge} - (b \cdot q)$, gli affianco a destra la prossima cifra del dividendo, il valore ottenuto è il resto parziale che assegno a r^{\wedge} , riparto dal punto 2.
 2. No - ho terminato l'operazione, $r^{\wedge} - (b \cdot q)$ è il resto finale r .
-

Divisione binaria:

$\begin{array}{r} 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{array}$		$\begin{array}{r} 1 \\ 1 \\ 1 \end{array}$
$\begin{array}{r} 1 \\ 1 \\ 1 \end{array}$		
$\begin{array}{r} 1 \\ 0 \\ 0 \\ 1 \end{array}$		$\begin{array}{r} 1 \\ 1 \end{array}$
$\begin{array}{r} 1 \\ 1 \\ 1 \end{array}$		
$\begin{array}{r} 1 \\ 0 \end{array}$		

Per ogni iterazione, la cifra da aggiungere al quoziente finale, è 0 o 1.

Per capire se essa è 0 o 1, basta capire se il divisore sta nella parte del dividendo considerata, e per fare ciò, basta sottrarre i due numeri e vedere se il segno è negativo (non è contenuto) o è positivo (divisore contenuto nella porzione considerata).

Algoritmo:

Dato 10110011 / 1101, dove N=6 (numero di bit del dividendo):

- Assegno alla variabile r^{\wedge} , il dividendo esteso su $2n$ bit. ($r^{\wedge} = 000000\ 1011011$)
- Allineo il divisore b a sinistra della cifra più significativa di a in r^{\wedge} , esteso su $2n$ bit.
($b = 001101\ 000000$), l'ultima cifra significativa di b inizia alla posizione dopo la prima cifra significativa di r^{\wedge}

$r^\wedge = 000001011011$

$b = 011010000000$

$\hat{r} \leftarrow 000\ 000\ 101\ 001$
 $b \leftarrow 001\ 101\ 000\ 000$
 $q \leftarrow 000\ 000$ (opzionale)

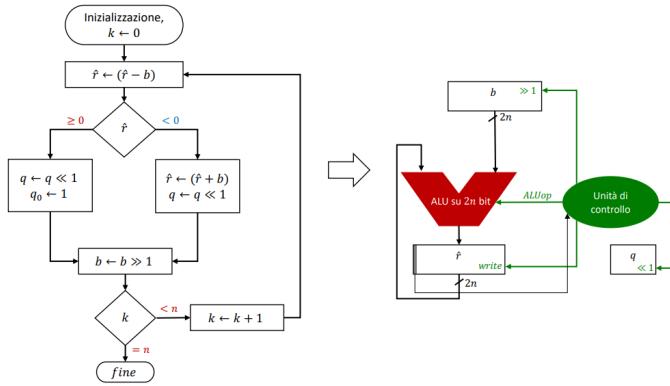
Inizializzazione

- Inizializzo il registro q per contenere il quoziente, su N bit. ($q = 000000$)
- Le cifre del quoziente “entrano” da destra verso sinistra (le cifre fanno “shiftare” q a sinistra)
- Svolgo $N+1$ iterazioni nel seguente modo:

$k = 0$	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6 (n)$
$r^\wedge = 000\ 000\ 101\ 001 -$ $b = 001\ 101\ 000\ 000$	$000\ 000\ 101\ 001 -$ $000\ 110\ 100\ 000$	$000\ 000\ 101\ 001 -$ $000\ 011\ 010\ 000$	$000\ 000\ 101\ 001 -$ $000\ 001\ 101\ 000$	$000\ 000\ 101\ 001 -$ $000\ 000\ 110\ 100$	$000\ 000\ 101\ 001 -$ $000\ 000\ 011\ 010$	$000\ 000\ 001\ 111 -$ $000\ 000\ 001\ 101$
$(-b)_{C_2^2} = 110\ 011\ 000\ 000 =$ $\hat{r} - b = 110\ 011\ 101\ 001$	$000\ 000\ 101\ 001 +$ $111\ 001\ 100\ 000 =$ $111\ 011\ 101\ 001$	$000\ 000\ 101\ 001 +$ $111\ 001\ 100\ 000 =$ $111\ 101\ 011\ 001$	$000\ 000\ 101\ 001 +$ $111\ 100\ 110\ 000 =$ $111\ 111\ 011\ 001$	$000\ 000\ 101\ 001 +$ $111\ 110\ 011\ 000 =$ $111\ 111\ 101\ 001$	$000\ 000\ 101\ 001 +$ $111\ 111\ 100\ 110 =$ $111\ 111\ 110\ 011$	$000\ 000\ 001\ 111 +$ $111\ 111\ 110\ 011 =$ $000\ 000\ 000\ 010$
Risultato < 0 (sempre) $q \leftarrow q \ll 1$ $b \leftarrow b \gg 1$ $q = 000\ 000$	Risultato < 0 $q \leftarrow q \ll 1$ $b \leftarrow b \gg 1$ $q = 000\ 000$	Risultato < 0 $q \leftarrow q \ll 1$ $b \leftarrow b \gg 1$ $q = 000\ 000$	Risultato < 0 $q \leftarrow q \ll 1$ $b \leftarrow b \gg 1$ $q = 000\ 000$	Risultato < 0 $q \leftarrow q \ll 1$ $b \leftarrow b \gg 1$ $q = 000\ 000$	Risultato ≥ 0 $\hat{r} \leftarrow (\hat{r} - b)$ $q \leftarrow q \ll 1, q_0 = 1$ $b \leftarrow b \gg 1$ $q = 000\ 001$	Risultato ≥ 0 $\hat{r} \leftarrow (\hat{r} - b)$ $q \leftarrow q \ll 1, q_0 = 1$ $b \leftarrow b \gg 1$ $q = 000\ 011$

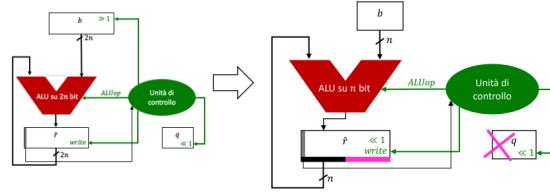
A ogni iterazione, faccio entrare uno 0 o un 1 a destra di q, shifto b a destra, e se b entra in r^\wedge , sottraggo b a r^\wedge

La Sottrazione tra r^\wedge e b, viene effettuata come somma di r^\wedge e il negativo di b (-b in complemento a 2)



Ottimizzazione

- Ad ogni step shiftiamo verso destra il divisore b e lo sommiamo con il resto parziale \hat{r}
- Dentro b non scriviamo mai nuovi bit
- Quindi possiamo ottenere la stessa operazione facendo restare b fisso su n bit e shiftando il resto parziale verso sinistra ad ogni iterazione
- La somma verrà svolta tra b (che ora è su n bit) e gli n bit più alti di \hat{r} : la ALU diventa a n bit!



Ulteriore ottimizzazione

- Ogni volta che shiftiamo \hat{r} a sinistra aggiungiamo uno zero a destra che non viene più usato
- Gli n bit a destra di \hat{r} possono essere usati per scrivere il quoziente!

Anche qui un'ottimizzazione sarebbe utilizzare b su n bit, e un alu su n bit.

Il Registro di $2n$ bit per r^{\wedge} , viene utilizzato nella parte bassa per memorizzare il quoziente.

CPU a singolo ciclo:

La CPU che costruiremo, sarà a singolo ciclo, cioè eseguirà una singola operazione per ciclo di clock.

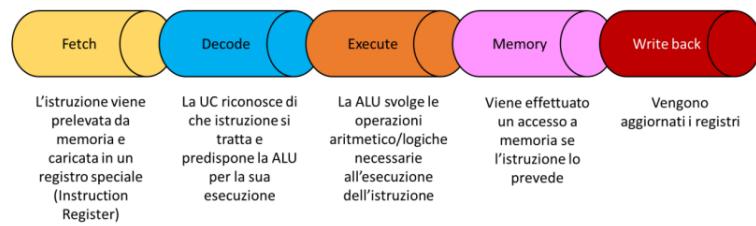
Costruiremo una CPU general purpose, in grado di eseguire le istruzioni descritte dall'Instruction Set Architecture MIPS.

L'ISA è l'inizio del software, l'insieme delle istruzioni in linguaggio macchina, supportate da quella CPU.

Ma cos'è un'istruzione? Un'istruzione è una stringa binaria che codifica le informazioni necessarie alla CPU per eseguire un'operazione.

La CPU decodifica tale stringa, ed esegue l'operazione che essa rappresenta.

Un codice sorgente compilato per un particolare ISA, esegue su tutti gli elaboratori che implementano tale ISA (cioè che abbia stessa ISA, o che l'ISA dell'elaboratore implementi le istruzioni del linguaggio macchina), che sia AMD, Intel ecc...



Nella fase Memory si scrive in RAM, in Write Back si scrive sul banco dei registri.

Il Circuito della CPU è composto da due componenti principali:

- **Data Path**: è il percorso che le informazioni seguono all'interno della CPU attraversando i suoi sotto-componenti, può variare a seconda dell'istruzione
- **Logica di Controllo** (Unità di controllo)
 - Manovra gli «scambi» del data path in modo che le informazioni seguano un percorso diverso a seconda dell'istruzione
 - Controlla i sotto-componenti della CPU a seconda dell'istruzione (ad es. decidendo i segnali di controllo della ALU)

E Dovrà eseguire le istruzioni:

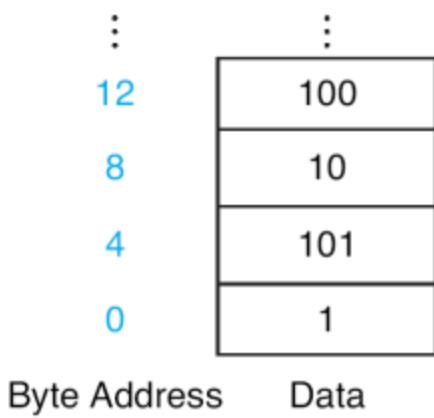
- Aritmetico Logiche (a carico della ALU) - ADD, SUB, AND, OR, NOT...
- Accesso alla memoria (load e store, lettura e scrittura della memoria)
- Controllo di flusso - IF, WHILE.. salti condizionati e incondizionati, cioè deviazioni nella lettura delle istruzioni.

Nel linguaggio macchina MIPS, ogni istruzione è codificata su 32 bit.

Ogni istruzione può essere composta secondo 3 formati:

- R (Register) - per le istruzioni aritmetico logiche che operano su valori in registri.
 - I (Immediate) - istruzioni aritmetico logiche che operano sulle costanti (valori immediati) e per la istruzioni di salto condizionato.
 - J (Jump) - salto non condizionato.
-

La memoria:



La Memoria può essere immaginata come un array unidimensionale di "Word" a n bit.

La Dimensione massima della memoria, sarà 2^n (Motivo per cui nelle architetture a 32bit ho massimo 4GB di ram utilizzabili, $2^{32} = 4\text{GB}$).

La lettura e la scrittura della memoria, avviene GENERALMENTE per numero di word.

Ad ogni word è associato un indirizzo di memoria (anche detti metadati sui dati, cioè l'indirizzo associato ad un dato, a una cella di memoria), sempre su n bit. (Word da 32 bit, avrò metadati, o indirizzi, sempre su 32 bit).

In MIPS, ogni Word è a 32 bit, 4 byte. ecco perchè gli indirizzi vanno di 4 in 4, perchè ogni word occupa 4 byte.

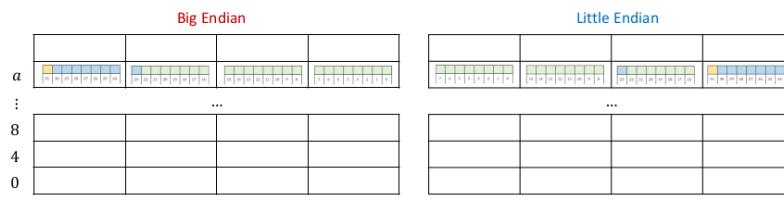
Si dice che gli indirizzi sono "allineati alla memoria", cioè sono multipli della dimensione in byte degli indirizzi stessi. (di 4 in 4, gli indirizzi sono a 32 bit).

Ogni word, è composta da 4 byte, quindi la word indicata dal byte address 0, occuperà i byte 0,1,2,3.

Al momento della lettura/scrittura della memoria, devo scegliere come memorizzare i dati.

Scandisco i byte da sinistra verso destra ma li memorizzo secondo due convenzioni:

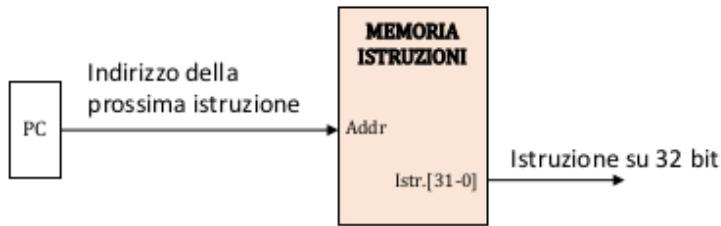
- Big Endian - i byte letti sono memorizzati da sinistra verso destra (Prima memorizzo parte più significativa, poi parte meno significativa)
- Little endian - i byte letti sono memorizzati da destra verso sinistra



A Livello implementativo, la memoria è un modulo, a cui accedo tramite indirizzi, per operazioni di lettura e scrittura.

E' divisa in due sezioni:

- Memoria istruzioni: Ogni word (4 o 8 bytes) contiene un istruzione per la CPU, la consideriamo utilizzata in sola lettura. Il programma è precedentemente scritto in essa dal sistema operativo. L'indirizzo della cella da leggere nella memoria istruzioni, cioè il "segnalibro" di dove sono arrivato a leggere, sta in un apposito registro detto "program counter".



- Memoria dati: Ogni word è un dato, un valore. E' utilizzata sia in lettura che in scrittura, su di essa potrei ricavare un dato per un'operazione, ma anche voler scrivere il risultato di un'operazione. Ad ogni ciclo di clock, posso eseguire una sola operazione, o di lettura o di scrittura.

Prevede 4 segnali di input:

- Addr - indirizzo della word da leggere/su cui scrivere
- Data - la word che voglio scrivere in memoria
- MemRead - Segnale che indica se voglio leggere in memoria (1 - leggo, 0 - non leggo)
- MemWrite - Indica se voglio scrivere (1 - scrivo, 0 non scrivo).

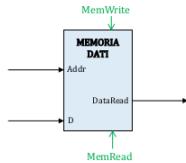
Se voglio leggere, "Data" sarà non utilizzato, se invece voglio scrivere, ho bisogno di specificare sia Addr che Data.

MemRead e MemWrite prevederanno combinazioni 00 (non faccio nulla), MemRead a 0 e MemWrite a 1 (Scrivo), MemRead a 1 e MemWrite a 0 (Leggo).

La combinazione con MemRead e MemWrite a 1 non è valida, è un errore.

Può prevedere un output "DataRead", cioè il dato letto (non prevede output se la memoria è utilizzata in modalità di scrittura).

- Input:
 - **Addr**: indirizzo della parola di memoria (32 bit)
 - **MemRead**: segnale di controllo (1 bit), 1 per lettura, 0 a riposo
 - **MemWrite**: segnale di controllo (1 bit), 1 per scrittura, 0 a riposo
 - **D**: il dato (parola) da trasferire in memoria se siamo in modalità scrittura (32 bit)
- Output
 - **DataRead**: il dato (parola) recuperato dalla memoria se siamo in modalità lettura (32 bit)



- **Random Access Memory (RAM)**: tempo di accesso ad una parola di memoria è fisso, indipendentemente dall'indirizzo a cui si trova una parola
- **MemRead e MemWrite**: in un ciclo di clock solo uno dei due verrà posto ad 1

Register file:

Altro tipo di memoria, è il "Register File", cioè una memoria RW (può essere utilizzata sia in lettura che scrittura) ad accesso molto rapido, costituito da un insieme di registri, in cui la CPU mantiene una copia dei dati più utilizzati in RAM, così che essi non siano costantemente presi dalla RAM, migliorando le prestazioni.

La CPU, prima dell'esecuzione, cerca di prevedere quali dati sono più utilizzati, da un'analisi del codice, e cerca già di predisporre il register file, copiando dati dalla RAM.

La CPU lavora quindi, per il 99% del tempo, con operazioni di load e store, cioè lettura e scrittura su e dal banco registri.

Anche il banco registri è una memoria indicizzata.

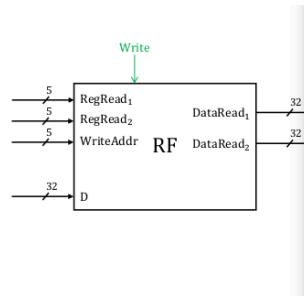
In MIPS, il banco registri prevede 32 registri, ognuno da 32 bit, ognuno con associato un indirizzo a 32 bit.

Per indicizzare i 32 bit, bastano $2^5 = 5$ bit.

In uno stesso ciclo di clock, il Register File può essere letto e scritto contemporaneamente.

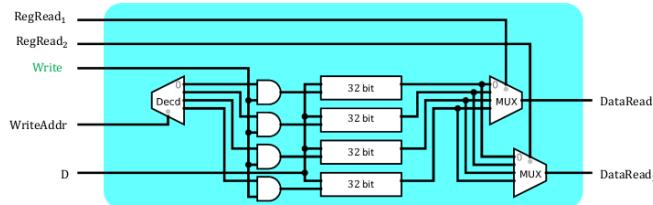
In un ciclo di clock, il Register File viene SEMPRE LETTO, ma POTREBBE anche essere scritto.

- Proprietà interessante: si possono leggere due registri contemporaneamente (due uscite di lettura)
- Input
 - **Write**: segnale di controllo (1 bit), 1 per scrittura
 - **RegRead₁** e **RegRead₂**: indirizzi (5 bit) dei due registri da leggere
 - **WriteAddr**: indirizzo (5 bit) del registro in cui scrivere (se in modalità scrittura)
 - **D**: dato (32 bit) da scrivere nel registro indirizzato da **WriteAddr** (se in modalità scrittura)
- Output
 - **DataRead₁** e **DataRead₂**: i valori (32 bit) contenuti nei registri indirizzati da **RegRead₁** e **RegRead₂** (se in modalità lettura)



- Write - viene posto a 1 se il register file deve essere ANCHE scritto (register file è sempre letto).
- RegRead (DataRead) 1 e 2 - Indirizzi dei registri da leggere (posso leggere due registri in uno stesso ciclo di clock), in Output su DataRead 1 e 2, avrò i dati corrispondenti agli indirizzi indicati in RegRead 1 e 2.
- WriteAddr - Indirizzo del registro su cui scrivere (in MIPS, 5 bit)
- D - Dato da scrivere nel registro di indirizzo indicato da WriteAddr.

- Esempio di register file con 4 registri a 32 bit
- In questo esempio, avendo solo 4 registri, bastano 2 bit per indirizzare un registro



I Due multiplexer a destra, hanno compito di "controller della memoria", restituiscono il valore del registro associato all'indirizzo di RegRead 1 e 2.

La logica responsabile della scrittura, è a carico di un decoder, che ha un'uscita per ogni registro.

Nel Decoder, entra l'indirizzo del registro su cui scrivere, la sua uscita è in AND col segnale Write (la scrittura avviene solo se su WriteAddr c'è qualcosa, e Write è a 1).

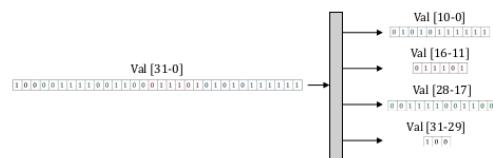
Il Dato "D" si presenta invece direttamente ad ogni ingresso.

Ogni registro, è equivalente ad un Flip Flop, ecco perchè posso leggere e scrivere contemporaneamente.

Splitter:

Permette di dividere un segnale in più sottogruppi.

- **Esempio:**



Nell'esempio, divide un registro a 32 bit in 4 word.

Quali sono le istruzioni che la nostra CPU dovrà supportare?

Le istruzioni dell'ISA MIPS che la nostra CPU supporterà sono:

Ogni istruzione può essere composta secondo 3 formati:

- R (Register) - per le istruzioni aritmetico logiche che operano su valori in registri.
- I (Immediate) - istruzioni aritmetico logiche che operano sulle costanti (valori immediati) e per le istruzioni di salto condizionato.
- J (Jump) - salto non condizionato.

Operazioni "R":

- **add** r_d, r_s, r_t : carica nel registro r_d la somma dei valori contenuti nei registri r_s e r_t
- **sub** r_d, r_s, r_t : carica nel registro r_d la differenza tra valori contenuti nei registri r_s e r_t
- **and** r_d, r_s, r_t : carica nel registro r_d l'AND bit a bit tra valori contenuti nei registri r_s e r_t
- **or** r_d, r_s, r_t : carica nel registro r_d l'OR bit a bit tra valori contenuti nei registri r_s e r_t
- **slt** r_d, r_s, r_t : carica nel registro r_d il valore 1 se il valore contenuto nel registro r_s è strettamente minore di quello contenuto nel registro r_t

La ALU che abbiamo precedentemente implementato, è già in grado di eseguire queste operazioni aritmetico-logiche! Ma non è in grado di ricavare i valori dati gli indirizzi dei registri indicati.

Le istruzioni in formato R, sono rappresentate con 32 bit nel seguente formato:

OPCODE	r_s	r_t	r_d	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

- OPCODE - codice identificativo dell'operazione da eseguire. Per tutte le istruzioni di tipo R, è sempre pari a 000000.

La CPU quando riceve l'istruzione, la prima cosa che fa è leggere il campo OPCODE, quando legge 000000, capisce il formato dell'istruzione, e saprà che nei successivi 5 bit ci sono l'indirizzo del registro sorgente, i successivi 5 bit ancora sono ancora di un'altro registro sorgente, e i prossimi ancora sono l'indirizzo del registro destinazione.

- shamt - diminutivo di "Shift Amount", non utilizzato per le nostre operazioni.
- funct - 6 bit, indicano la funzione aritmetico - logica da eseguire (add, sub...)

Ci sono 2^6 operazioni, ma noi per semplicità consideriamo solo le 5 operazioni in tabella.

All'esame non ci sarà bisogno di sapere la tabella OPCODE, te la darò lui

Funzione	funct
add	100000
sub	100010
and	100100
or	100101
slt	101010

Esempio di istruzione con formato R

Istr: 0 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 1 0 0 1 0 0

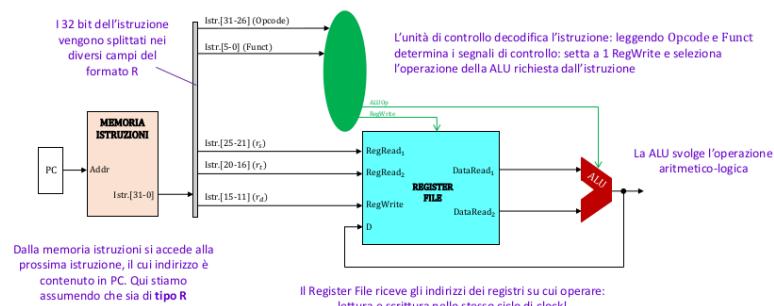
(in formato esadecimale 0x01902824)

OPCODE	r_s	r_t	r_d	shamt	funct
000000	01100	10000	00101	00000	100100
000000	12	16	5	00000	and

and r_5, r_{12}, r_{16} : carica nel registro r_5 l'AND bit a bit tra i valori nei registri r_{12} e r_{16}

Data Path di un'operazione di tipo R:

(Assumo che sia di tipo R, quindi lo splitter divide già l'istruzione nei campi giusti, e che i dati siano già stati precaricati dalla RAM ai registri.)



- Dal Program Counter, accedo alla memoria istruzioni e ricavo la prossima istruzione da eseguire (assumo sia di tipo R)

- Tramite lo splitter, "divido" l'istruzione a 32 bit, nei 6 campi dell'operazione R (in slide sono 5 perchè shamt non è considerato).
- Ricavo i dati dai registri con indirizzo indicati da Rs e Rt: Questi campi finiscono nelle linee di input di RegRead 1 e 2 di un Register File.
- Le due linee di output del Register File, arrivano alla ALU, che svolge l'operazione aritmetico logica. L'Output della ALU, torna all'input D del Register File, per essere scritto in Rd (Registro Destinazione).

Come faccio a indicare all'ALU quale operazione aritmetico logica eseguire?

Come faccio a indicare al Register File quando scrivere (Write a 1)?

Introduco una unità di controllo (verde) che riceve in input l'OPCODE, e il campo FUNCT, e comanda l'ALU in base al valore di FUNCT.

Operazioni "I":

Prevedono valori di registri + valori "costanti":

- Il formato I è usato per le istruzioni che utilizzano sia valori che sono in qualche registro sia **costanti**: valori immediati su 16 bit specificati direttamente dentro l'istruzione stessa!
- Le istruzioni di questo formato che vedremo sono:
 - **lw r_t , OFFSET(r_s)**: carica nel registro r_t la parola di memoria il cui indirizzo è pari a $r_s + \text{OFFSET}$
 - **sw r_t , OFFSET(r_s)**: trasferisci il contenuto del registro r_t nella parola di memoria all'indirizzo $r_s + \text{OFFSET}$
 - **beq r_s, r_t, OFFSET** : se il contenuto del registro r_s è uguale a quello del registro r_t , salta all'istruzione che sta all'indirizzo PC + OFFSET, altrimenti prosegui normalmente con la prossima istruzione (PC + 4)

OPCODE	r_s	r_t	IMMEDIATO
6 bit	5 bit	5 bit	16 bit

- Questo formato include anche istruzioni aritmetico-logiche come addi, andi, etc.. Queste sono varianti delle istruzioni originali dove uno degli operandi è un immediato (quindi caricato direttamente dall'istruzione e non da un registro)
- Ci sono anche varianti delle istruzioni per accesso a memoria (lb, sb) e dei branch (bne, beqz, ...)

L'Immediato, non può essere a 32 bit ovviamente, altrimenti occuperebbe tutti i 32 bit di cui è composta l'istruzione.

OFFSET è uno scostamento, quando in programmazione faccio `array[index]`, è l'equivalente di fare base address + offset (base address è il nome dell'array, offset è index)

Load Word e Store Word fanno la stessa cosa, Load Word trasferisce da RAM a Registri, Store Word trasferisce da Registri a RAM.

Quindi il Load e Store (Leggo e scrivo), si intendono "dalla RAM".

Nel formato R e I, OPCODE, Rs, Rt, sono nella stessa posizione, solo che gli ulteriori 16 bit al posto di essere FUNCT e SHAMT, sono per il valore immediato indicato nell'istruzione.

Esempio di istruzione lw con formato I

Istr: 1 0 0 0 1 1 0 1 1 1 0 0 0 1 0 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0

(in formato esadecimale 0x8DC500F0)

OPCODE	r_s	r_t	OFFSET
100011	01110	00101	0000000011110000
lw	14	5	240

lw r_5 , 240(r_{14}): carica nel registro r_5 la parola di memoria il cui indirizzo si ottiene comando 240 al valore contenuto nel registro r_{14}

Attenzione!

- Il contenuto del registro r_{14} deve essere un indirizzo valido, altrimenti si genera un errore
- Il valore dell'offset deve essere un multiplo di 4 altrimenti l'indirizzo risultante non sarà allineato e si genera un errore

Attenzione! in Rs deve esserci un indirizzo valido, e anche l'offset deve essere correttamente allineato, quindi un multiplo di 4 (essendo ogni word in RAM a 32 bit), altrimenti la lettura fallisce.

Esempio di istruzione sw con formato I

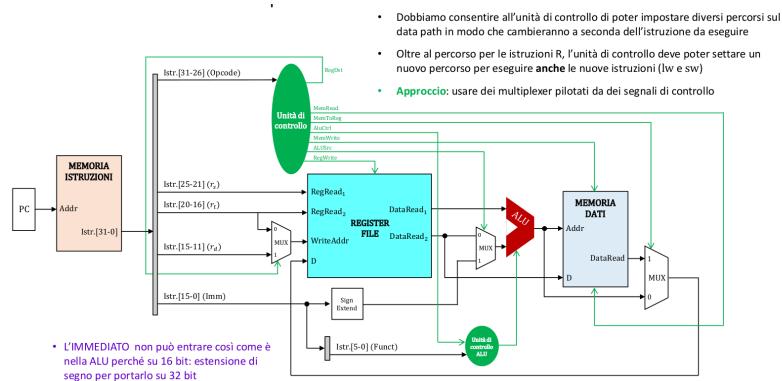
Istr: 1 0 1 0 1 1 0 1 1 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0

(in formato esadecimale 0xAFCD0C04)

OPCODE	r_s	r_t	OFFSET
101011	11110	01101	0000110000000100
sw	30	13	3076

sw r_{13} , 3076(r_{30}): trasferisci il valore contenuto nel registro r_{13} nella parola di memoria il cui indirizzo si ottiene comando 3076 al valore contenuto nel registro r_{30}

Data Path generico, per istruzioni di tipo R e I:



Notiamo che OPCODE, RS e RT sono sempre utilizzati, poi in base a se l'istruzione è R o I, utilizzerò i 16 bit rimanenti come RD shamt e func o come IMM (Notiamo infatti come i bit si sovrappongono, [15-11] e [15-0])

In WriteAddr, devo sapere cosa mettere: se l'operazione è di tipo R, in WriteAddr metto RD, se ho invece una Store Word (Leggo da Registro e metto in RAM, non c'è scrittura del register file) metto 0, se ho una Load Word invece, devo mettere RT (Leggo da RAM e scrivo in Registro)

Metto un Multiplexer che effettua questa scelta

Nell'ALU, devo sapere se il secondo operando è un immadiato o il valore di un'altro registro. Metto un'altro MUX.

L'Immediato passa in un componente che ne estende la lunghezza a 32 bit, perchè la ALU non può operare con esso espresso su 16 bit.

Dopo la fase di execute (lavoro della ALU), c'è la fase di Memory, scrittura in RAM, quindi l'output della ALU va direttamente in memoria dati.

Se ho un'operazione R, la memoria non deve essere interpellata, il risultato della ALU va direttamente nello 0 del MUX, in ADDR della memoria dati non ci va proprio.

Torna indietro al register file, per essere scritto nel registro indicato da Rd.

Se ho un'operazione di tipo I:

Se ho una Load Word, dalla memoria devo LEGGERE, quindi in ADDR, va il risultato dell'ALU, e il DataRead della memoria dati torna indietro al register file per la scrittura.

Se ho una Store Word, in memoria devo SCRIVERE, quindi in ADDR va il risultato dell'ALU (Indirizzo dove devo scrivere, risultato di base address+offret), e il D, mi va il dato da scrivere, che viene da DataRead2 del RegisterFile.

Segnali di lettura/scrittura: hanno un effetto solo quando posti a 1

RegWrite	Attiva la scrittura nel Register File: il dato D viene scritto nel registro indirizzato da WriteAddr
MemRead	Attiva la lettura in memoria: sull'uscita DataRead viene posto il valore della parola di memoria che ha indirizzo Addr
MemWrite	Attiva la scrittura in memoria: il dato D viene scritto nella parola di memoria che ha indirizzo Addr

Segnali di selezione

	Se posto a 0	Se posto a 1
ALUSrc	Il secondo operando della ALU viene dal Register File: uscita DataRead ₂	Il secondo operando della ALU è l'estensione su 32 bit del campo IMMEDIATO dell'istruzione (bit 15-0)
MemToReg	Il dato D da scrivere nel Register File proviene dall'uscita della ALU	Il dato D da scrivere nel Register File dall'uscita di lettura DataRead della memoria
RegDst	L'indirizzo del registro destinazione è estratto dal campo r_d dell'istruzione (bit 20-16)	L'indirizzo del registro destinazione è estratto dal campo r_d dell'istruzione (bit 15-11)
Branch AND Zero	L'indirizzo da scrivere in PC è $PC + 4$ (a meno che jump non sia posto a 1)	L'indirizzo da scrivere in PC è $PC + 4$ a cui si somma il valore del campo IMMEDIATO dell'istruzione (bit 15-0) esteso su 32 bit e moltiplicato per 4
Jump	L'indirizzo da scrivere in PC è $PC + 4$ (a meno che Branch AND Zero non sia posto a 1)	L'indirizzo da scrivere in PC ottenuto aggiungendo al valore del campo PSEUDO-INDIRIZZO dell'istruzione (bit 0-15) due zeri a destra e i 4 bit più significativi di PC a sinistra

- **AluCtrl**: comunica all'unità di controllo della ALU quale tipo di istruzione la CPU sta eseguendo, ci interessano solo le istruzioni aritmetico-logiche, gli accessi a memoria e la branch; con jump la ALU non è necessaria

- **ALUOp**: configura la ALU perché esegua una delle operazioni supportate

AluCtrl	ALUOp
Istruzione Aritmetico-Logica (tipo R)	La ALU fa l'operazione indicata da funct
Accesso a memoria (lw o sw)	La ALU fa una somma
Branch on equal	La ALU fa una differenza

- Iniziamo scegliendo una codifica per AluCtrl il cui valore dipende da Opcode

Opcode	AluCtrl
000000 (tutte le tipo R)	10
100011 (lw)	00
101011 (sw)	00
000100 (breq)	01

AluCtrl = 10 → Aritmetico-logica

AluCtrl = 00 → Accesso a memoria

AluCtrl = 01 → Branch