

Lezione del 5 Ottobre 2021

Introduzione

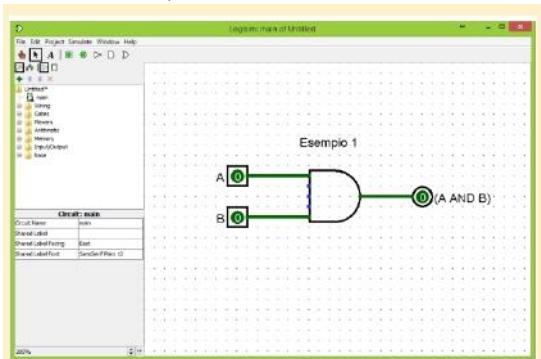
Di cosa si occupa questo insegnamento?

Per la definizione di "architettura":

- arte e tecnica del progettare, disegnare, realizzare edifici o altre grandi opere basandosi su principi estetici e ingegneristici (la grande opera che costruiamo si chiama elaboratore);
- (in informatica) struttura e componenti di un sistema di elaborazione.

Obiettivi in ordine di complessità crescente:

- comprendere i principi fondamentali alla base della realizzazione di un elaboratore digitale;
- studiare come l'informazione viene rappresentata dentro un elaboratore e secondo quali principi essa può essere manipolata;
- analizzare i componenti fondamentali che permettono di far eseguire all'hardware operazioni elementari;
- combinare tali componenti per realizzare elaborazioni sempre più complesse fino ad ottenere il componente centrale di un elaboratore, la CPU.

Il programma che seguiremo	
Teoria	Laboratorio
<ul style="list-style-type: none">• Introduzione• Rappresentazione e elaborazione logica dell'informazione: codifiche binarie per numeri naturali, interi e reali, funzioni logiche e algebra di Boole• Rappresentazione ed elaborazione fisica dell'informazione: porte logiche e tabelle di verità• Sintesi di funzione logiche con circuiti combinatori, realizzazione dell'unità aritmetico logica (ALU)• Memorizzazione dell'informazione: elementi di logica sequenziale (bistabili, latch, sincronizzazione tramite clock)• Sintesi di circuiti	<ul style="list-style-type: none">• Codifica binaria• Logica combinatoria (forma combinatoria, cammino critico)• Logica combinatoria avanzata (moltiplicazione, ALU)• Logica sequenziale (memorie, macchine a stati finiti)  <p><i>logisim</i></p>

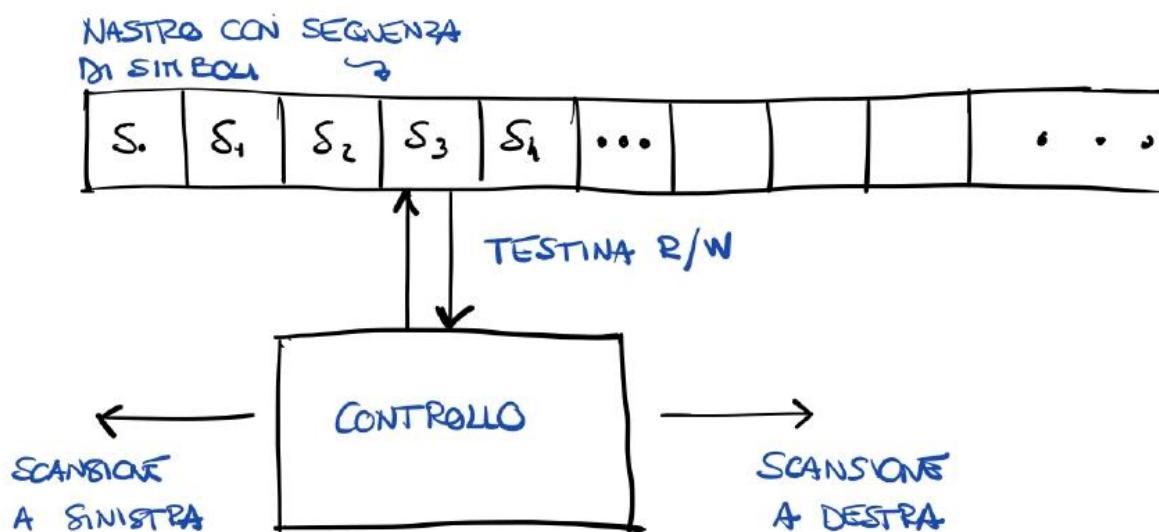
- | | |
|---|--|
| sequenziali, macchine a stati finiti
<ul style="list-style-type: none"> • Progetto di una CPU singolo-ciclo e cenni al caso multi-ciclo | |
|---|--|

L'elaboratore

Un elaboratore è una macchina capace di rappresentare, memorizzare e manipolare delle informazioni dati in input per produrne altre in output.

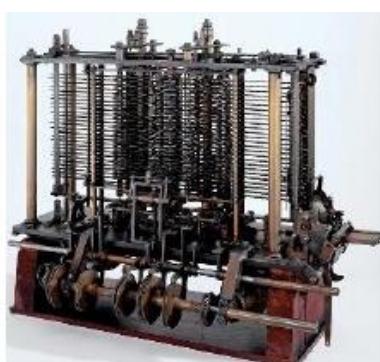
Esempio: una macchina che prende in input 10 numeri interi, li ordina e li salva in memoria.

Il modello matematico dell'elaboratore più noto è la macchina di Turing (1936), elaboratore teorico capace di svolgere qualsiasi algoritmo.



Breve storia dell'elaboratore

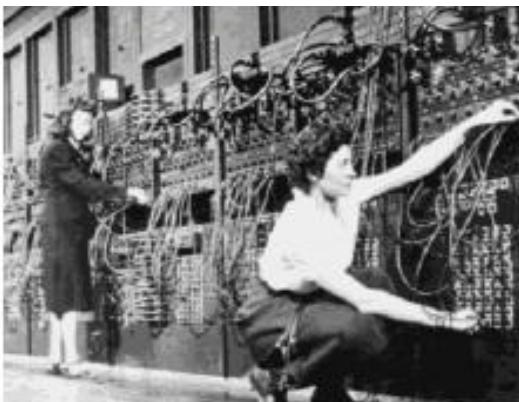
Abbiamo un'era pre-elettronica, composta da calcolatori meccanici e elettromeccanici che va dal 1623 al 1945.



L'Analytical Engine
progettato da Charles
Babbage (1837)

Con il successivo avvento dei calcolatori elettronici con salti generazionali associati ad importanti innovazioni tecnologiche:

- Prima generazione (1946-1955): utilizzo di valvole, diodi e triodi, prestazioni migliorate di 1000 volte attraverso ENIAC, con programmi realizzati cambiando il cabaggio della macchina manualmente.



ENIAC (1946), primo computer general purpose, Università della Pennsylvania

- Seconda generazione (1952-1963): introduzione dell'elettronica a stato solido (semiconduttori) e delle memorie ferromagnetiche, primo super-calcolatore CDC, invenzione del Fortran, primo linguaggio ad alto livello.



IBM 7090 (1959) usa transistor anziché valvole

- Terza generazione (1963-1971): avvento dei circuiti integrati, IBM 360 prima famiglia di calcolatori.



IBM 360 (1964), la prima famiglia di calcolatori

- Quarta generazione (1971-1977): miniaturizzazione in larga scala (VLSI), introduzione del microprocessore, memorie a semiconduttori, Apple II (1977).



Apple II (1977), primo home computer su larga scala

- Quinta generazione (1978-2003): avvento dei personal computer, PC come workstation.

```
C:\MSX\B1P> Directory of C:\MSX
.
.
.
11-22-2013 1392-09-01 2:45p <DIR> COMMAND
11-22-2013 1392-09-01 2:45p <DIR> DOS
11-23-2013 1392-09-02 2:27p <DIR> COMMAND
11-23-2013 1392-09-02 2:27p <DIR> DOS
11-23-2013 1392-09-02 2:27p <DIR> COMMAND
11-23-2013 1392-09-02 2:28p <DIR> CURSORS
11-29-2013 1392-09-02 2:28p <DIR> DLLS
11-03-2013 1395-09-06 2:28p <DIR> FONTS
11-23-2013 1392-09-02 2:29p <DIR> ICONS
11-23-2013 1392-09-02 2:29p <DIR> INT
11-23-2013 1392-09-02 2:29p <DIR> PROGRAMS
11-23-2013 1392-09-02 2:28p <DIR> SYSTEM
11-23-2013 1392-09-02 2:28p <DIR> TEMP
0 file(s)          0 bytes
13 dir(s)        54,396,916 bytes free
546,112 bytes free memory

C:\MSX>ver
MSXb Version 0.22 Beta Release
C:\MSX>
```

MS DOS (1982)

- Futuro: computer quantistici e computer molecolari.

Legge di Moore

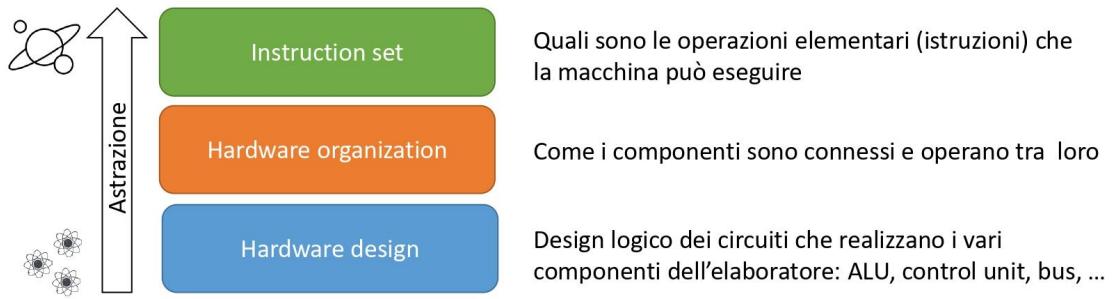
La storia dell'elaboratore presenta alcune caratteristiche generali:

- Aumento della velocità;
- Miniaturizzazione;
- Diminuzione dei costi;
- Efficienza energetica;
- Differenziazione.

L'evoluzione di alcuni di questi principi fu intuita già nel 1965 da Gordon Moore, infatti secondo la sua legge, la complessità dei circuiti integrati raddoppia ogni 18 mesi, descrivendo così lo sforzo ingegneristico degli ultimi 50 anni. Attenzione! l'evoluzione dei circuiti integrati sta per volgere al termine.

Architettura di un elaboratore

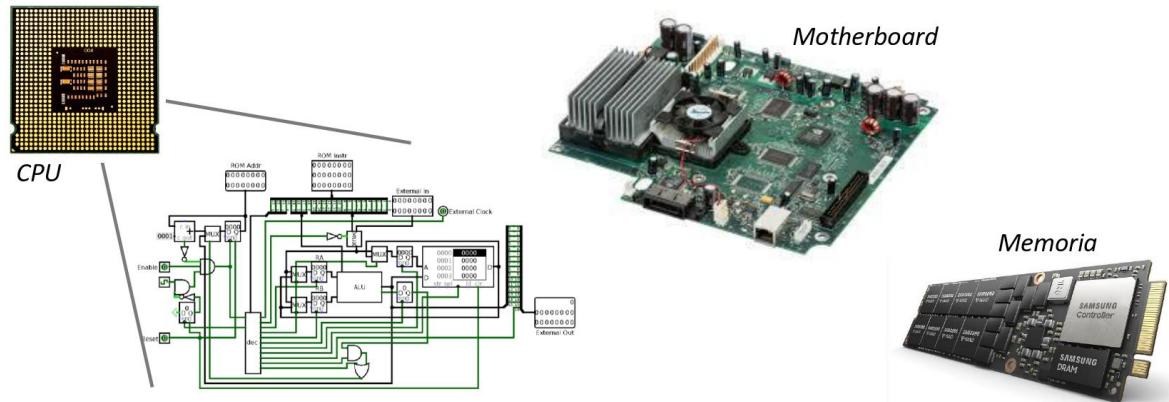
L'architettura di un elaboratore è una descrizione di come è fatta una macchina in grado di svolgere elaborazione automatica dell'informazione.



Il vantaggio dell'astrazione risiede nel fatto che ogni livello si avvale di elementi definiti nel livello sottostante trascurando come questi siano fatti al loro interno. In questo corso seguiremo un approccio bottom-up.

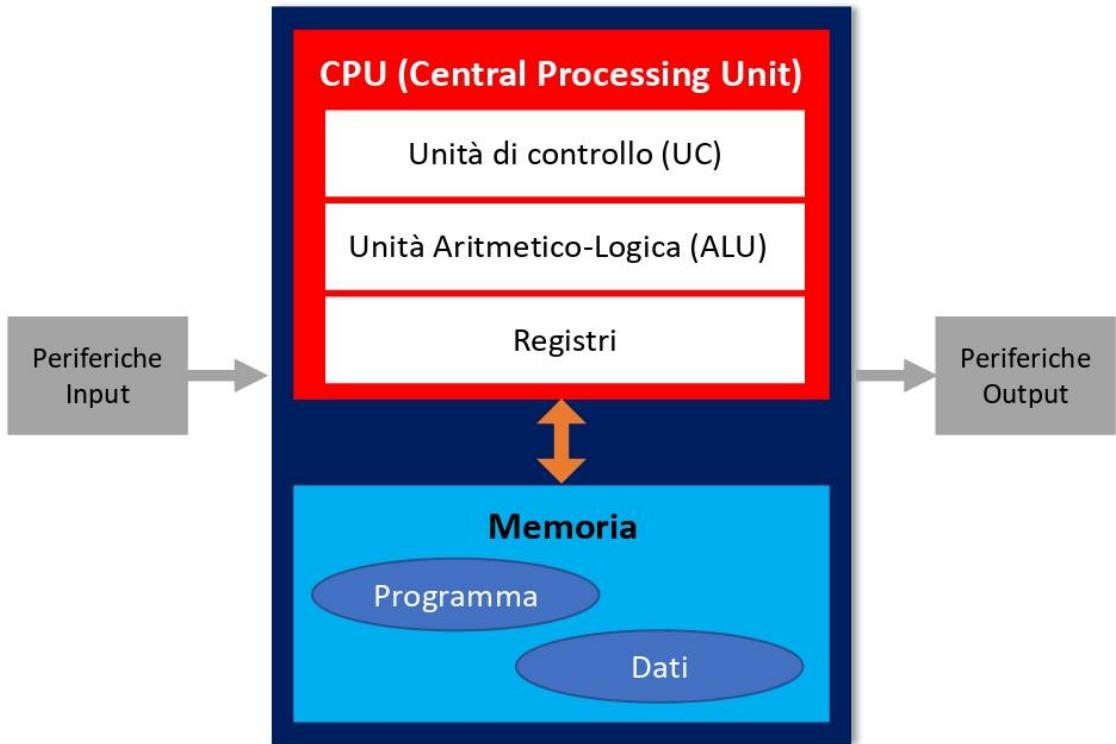
Hardware

Esso è l'insieme dei componenti fisici che compongono un elaboratore, ciascuno adibito ad una particolare funzione.



Organizzazione

L'organizzazione della maggior parte degli elaboratori oggi segue un modello chiamato Architettura di Von Neumann



Adesso vedremo una panoramica generale che verrà poi espansa durante Architettura degli elaboratori II.

La CPU (o central processing unit) ha il compito di eseguire delle istruzioni in sequenza (ad esempio somme, moltiplicazioni, test, accessi in memoria ecc...).

Per fare questo si avvale di:

- UC (unità di controllo), supervisore dell'esecuzione;
- ALU (unità aritmetico-logica), esecuzione di calcoli logici o aritmetici;
- Registri, memoria interna che fa da "banco da lavoro".

La memoria ha il compito di immagazzinare:

- Il programma (detto anche "testo"), cioè la sequenza di istruzioni che deve eseguire la CPU;
- I dati, cioè i valori su cui le istruzioni lavorano o rappresentano i loro risultati.

La CPU può accedervi in lettura o in scrittura attraverso un canale di comunicazione detto "bus".

E' strutturata in parole, ognuna di esse possiede un indirizzo. Per leggere o scrivere un dato in memoria è necessario specificare il proprio indirizzo.

Un elaboratore, per comunicare con il mondo esterno, per chiedere un input o mostrare un output ha bisogno di:

- Periferiche di input, le quali permettono di acquisire dati (tastiera, mouse, dischi);

- Periferiche di output, le quali ci permettono di visualizzare il dato (display, dischi, stampanti).

I limiti di questo modello sono di tipo “Bottleneck”, cioè collo di bottiglia, perchè durante questi anni secondo la legge di Moore, vi è stata una grande evoluzione delle CPU, mentre le memorie sono diventate solo più affidabili e dense, quindi non permettendo alla CPU di operare in modo più ottimale. La soluzione è pensare ad una architettura più avanzata (che vedremo più avanti).

Lezione del 7 Ottobre 2021

Turno B di laboratorio, SIGMA in via Celoria 18.

Ciclo di esecuzione

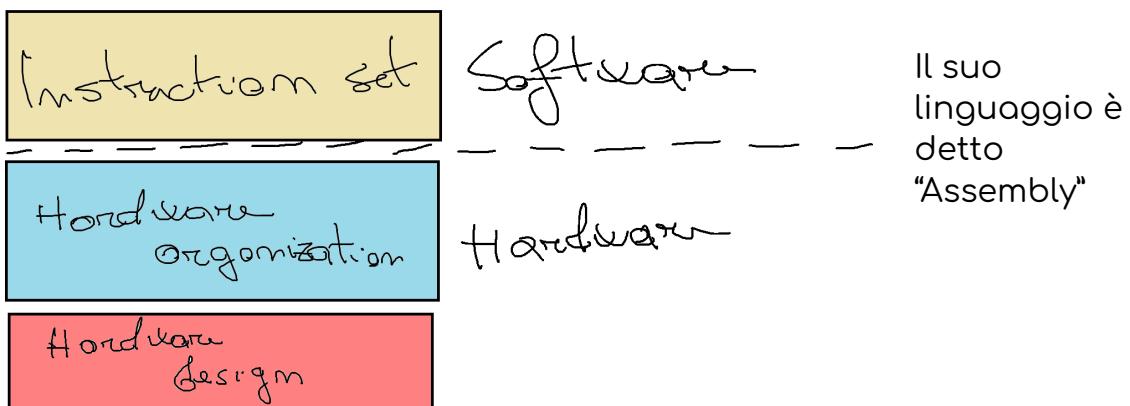
I componenti dell'architettura di Von Neumann svolgono ciascuno, in modo complementare, una particolare funzione. Interagendo e coordinandosi tra loro, questi componenti collaborano all'esecuzione di un programma istruzione dopo istruzione. L'esecuzione di ogni istruzione è divisa in 5 fasi ed il suo insieme è chiamato “Ciclo di esecuzione”.

- 1) Fetch: L'istruzione viene prelevata dalla memoria e caricato in un registro speciale (“instruction register”);
- 2) Decode: La UC (Unità di Controllo) riconosce di che istruzione si tratta e predisponde la ALU (Unità Aritmetico-Logica) per la sua esecuzione;
- 3) Execute: La ALU svolge le operazioni aritmetico-logiche necessarie all'esecuzione dell'istruzione;
- 4) Memory: Viene effettuato un accesso alla memoria, il quale può essere di scrittura o lettura, se l'istruzione lo prevede (lw e sw);
- 5) Write Back: dove vengono aggiornati i registri.

Queste 5 fasi corrispondono a 5 zone fisiche e specifiche della CPU. Può succedere che alcune di queste istruzioni passino per queste zone predestinate alle 5 fasi, aspettando senza compiere alcuna azione.

Instruction Set

Esso è l'insieme delle istruzioni che la macchina può eseguire. Sono espresse in linguaggio macchina, cioè in bit (per ogni istruzione 32 bit) che la UC è in grado di riconoscere. Conoscendo l'instruction set di una macchina possiamo scrivere programmi senza dover considerare i dettagli fisici dell'hardware (quindi è possibile eseguire l'azione di astrazione sul sistema).



Esso viene anche chiamato "Instruction Set Architecture" (ISA) e a volte soltanto "Architecture". Macchine diverse possono avere la stessa ISA (svolgono quindi gli stessi programmi) ma hardware diversi (quindi questo porta a costi e prestazioni diverse).

Esistono due tipi principali di ISA:

- ISA RISC (Reduced Instruction Set Computer), con istruzioni semplici e regolari con poco utilizzo di hardware;
- ISA CISC (Complex Instruction Set Computer), con istruzioni complesse che svolgono più azioni.

Le architetture CISC si dividono a loro volta in Intel x86 (o Intel 64), PowerPC, ARM (negli smartphone e nelle raspberry pi), RISC V e MIPS (Multiprocessor without Interlocked Pipeline Stages).

Elaboratore digitale

L'elaboratore deve essere in grado di rappresentare l'informazione attraverso una grandezza fisica: la tensione elettrica.



Questo grafico rappresenta un fenomeno fisico nel tempo legato al voltaggio. I valori che il voltaggio può assumere sono analoghi all'informazione, che poi viene trasformata in fenomeno fisico.



Successivamente venne usato un altro metodo, dividendo il risultato del fenomeno in range, nella quale le cifre possono rappresentare un valore alto o un valore basso. Questa è detta elaborazione digitale, poiché viene trasformata in informazione numerica.

18

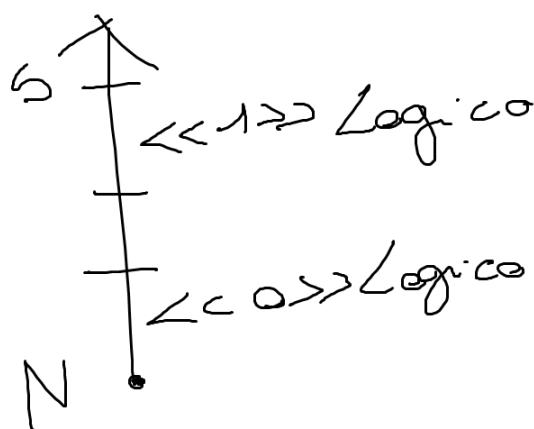
Analogico

Un segnale con voltaggio che vale
18

Digitale

Un segnale per rappresentare l'"1"
ed un altro per rappresentare l'"0"

I computer moderni rappresentano i valori con una serie di 0 e di 1, che cambiano valore sulle macchine attraverso apparecchi chiamati "transistor".



Sistemi di numerazione

I sistemi di numerazione sono composti da base e notazione.

Nel nostro caso nel sistema di numerazione binaria la nostra base è 2, cioè il numero delle cifre che possono essere rappresentate, mentre in base alla posizione n il numero prende il valore di $2^n \cdot 0$ o $2^n \cdot 1$.

Lezione dell'11 Ottobre 2021

Il valore di una stringa di n cifre in base n

$$\sum_{i=0}^{n-1} val(di)B^i$$

Codifica dei naturali \aleph , da base B a base 10

Prendiamo per esempio il numero:

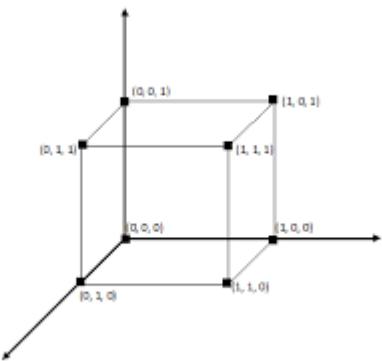
$$(111011)_2 = \sum_0^5 di \cdot 2^i = 1 \cdot 2^0 + 1 \cdot 2^1 + 0 + 1 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 = (59)_{10}$$

Per passare da un numero in base 10 ad un numero in base B bisogna applicare l'algoritmo iterativo delle divisioni:

- 1) Dividere N per b attraverso la divisione intera con resto;
- 2) Il resto diventa la prima cifra meno significativa del risultato;
- 3) Se il quoziente era 0, abbiamo finito;
- 4) Se il quoziente non era 0, esso diventerà il nuovo N e ripartiremo dallo step 1.

Rappresentazione grafica dei numeri binari

Un numero n-bit può essere inserito in uno spazio n-dimensionale:



I collegamenti fatti tra due numeri li cambiano di 1 bit

Due valori n1 e n2, il numero di posizioni diverse tra il primo numero ed il secondo è detta "distanza di Hamming".

Codice di Grey

Supponiamo di avere n=3 bit

Quanti numeri naturali possiamo scrivere in base 2 con 3 bit? $B^n = 8$, da 0 a 7.

Numeri decimali	Numeri binari	Distanze di Hamming	Codice di Gray
0	000		000
1	001	1	001
2	010	2	011
3	011	1	010
4	100	3	110
5	101	1	111
6	110	2	101
7	111	1	100

Mentre le distanze di Hamming sono 1213121, attraverso il codice di Gray sono unitarie.

Interi

Essi hanno una caratteristica in più: il segno. Vi sono infatti due modi diversi di rappresentarli:

- 1) Modulo e segno (facile per noi ma non conveniente per i calcolatori);
- 2) Complemento a 2 (più complicato per noi ma facile da capire per il calcolatore).

MDS

Esso riserva un bit per indicare il segno mentre i restanti indicano i valori ("unsigned" per i naturali).

Pro:

- Molto intuitivo per un essere umano.

Contro:

- Lo 0 ha due codifiche, quindi questo metodo pecca di ridondanza (-0, +0);
- Le operazioni risultano laboriose, infatti quando viene eseguita la somma algebrica controllando se i segni siano uguali per poi sommarli/sottrarli se i segni sono uguali/diversi.

Laboratorio del 12 Ottobre 2021

Professor Massimo W. Rivolta, turno B
massimo.rivolta@unimi.it

Esame: progetto individuale, il quale viene caricato e presentato al docente. Il voto di laboratorio vale $\frac{1}{3}$ del voto totale.

Nel corso tratteremo due macroargomenti:

- Rappresentazione numerica nel linguaggio macchina;
- Digital logic e elettronica in un simulatore computerizzato.

Perché si crede che il calcolatore sia bravo a fare i calcoli? Perchè il calcolatore è stato creato dall'uomo proprio per svolgere quella funzione.

I numeri sono un'invenzione umana, quindi come fa una macchina a capire il concetto di numero? Tramite dei segnali, essi sono di natura elettrica, quindi derivati da un fenomeno fisico.

Nell'architettura digitale tutta l'informazione viene codificata in bit, cioè l'unità minima di informazione per l'elaboratore (questa è una definizione che diamo noi), mentre per il calcolatore, a livello fisico, lo 0 e l'1 vengono differenziati tra corrente bassa e corrente alta.

La macchina è quindi composta da tanti interruttori i quali assumono un determinato significato a seconda del segnale della tensione.

Per associare un numero a questi interruttori bisogna prima riflettere su come rappresentare un numero in generale.

L'esempio lampante è la notazione posizionale, attraverso la quale ogni posizione del numero ha un peso diverso.

Quindi data una sequenza di numeri, essi andranno moltiplicati per la loro base elevata alla posizione di quest'ultimo:

$$\sum_{i=0}^{n-1} c_i B^i = c_0 B^0 + c_1 B^1 + c_2 B^2 \dots c_n B^n$$

Questa relazione può essere generalizzata cambiando la base in una base generica.

Con $B = 10$ abbiamo 10 cifre possibili, con B cifre abbiamo numero minimo 0 e numero massimo $B-1$.

In Base 2 (quella che usiamo nei calcolatori), avremo come B minimo 0 e come B massimo 1.

Per indicare la base di un numero di solito viene messo come pedice dopo il numero stesso.

Esempio:

$(127)_{10}$ oppure

$(1010)_2 = (2^3 + 2^1)_{10} = (10)_{10}$ tecnica per trasformare un numero in qualsiasi base in un numero in base 10

$(3AC)_{16} = (3 * 16^2 + 10 * 16^1 + 12 * 16^0)_{10} = (16 * 16 * 3 + 160 + 12)_{10} = (940)_{10}$

Metodo chiamato "polinomiale"

Invece per il procedimento contrario, ovvero convertire un numero N in base 10 in base B utilizziamo questo algoritmo:

- 1) Dividiamo N per B (attraverso la divisione intera con resto);
- 2) Il resto della divisione diventa la prima cifra meno significativa del numero;
- 3) Se il quoziente è 0 abbiamo finito;
- 4) Se il quoziente non è 0, poniamo come N il quoziente e ripartiamo dallo step 1.

Esempio 1:

$$(13)_{10} = (?)_2$$

13/2=6	resto 1	Il numero in base 2 sarà uguale a 1101
6/2=3	resto 0	
3/2=1	resto 1	
1/2=0	resto 1	

Esempio 2:

$$(4021)_{10} = (?)_{16}$$

4021/16=251	resto 5	Il numero in base 16 sarà uguale a FB5
251/16=15	resto 11	
15/16=0	resto 15	

Avendo come problema una conversione da base 2 a base 16, sapendo che $16=2^4$, procediamo attraverso il raggruppamento e le “lookup table”.

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

Interi con numero finito di bit

Le architetture degli elaboratori lavorano con un numero finito di bit, data la loro natura fisica. Dati n bit, i numeri positivi rappresentabili occuperanno un intervallo di $[0, 2^n-1]$.

Complemento a 2

In complemento a 2 (C2) i numeri positivi vengono rappresentati allo stesso modo, mentre i numeri negativi possono essere rappresentati in due modi:

- $2^n - |N|$;
 - Rappresentare $|N|$ in modulo standard complementare a 1
tutti i bit aggiungere 1.

Come fa a capire la macchina se il numero inserito sia positivo o negativo? Dal range di rappresentazione che dati n bit è uguale a $[-2^{n-1}, 2^{n-1}-1]$.

Esercizi

- Convertire da base 10 a base 8: 112; 23; 89; 254
 - Convertire da base 10 a base 2: 45; 64; 321; 76
 - Convertire da base 2 a base 10: 101100; 11101
 - Determinare la base per cui è esatta la seguente operazione: $\sqrt{232}=14$
 - Eseguire in ca2: 44+12; 36-11; 48+59; 16-9

Lezione del 14 Ottobre 2021

Complemento a 2

- Supponiamo di avere a disposizione n bit e un N numero intero
 - Se N è un numero positivo o nullo lo codifico come naturale, e il bit più a sinistra viene messo a 0 (come in Modulo e segno)
 - Se N è negativo lo codifico come $2^n - |N|$ su n bit, cioè il complementare del numero su 2^n

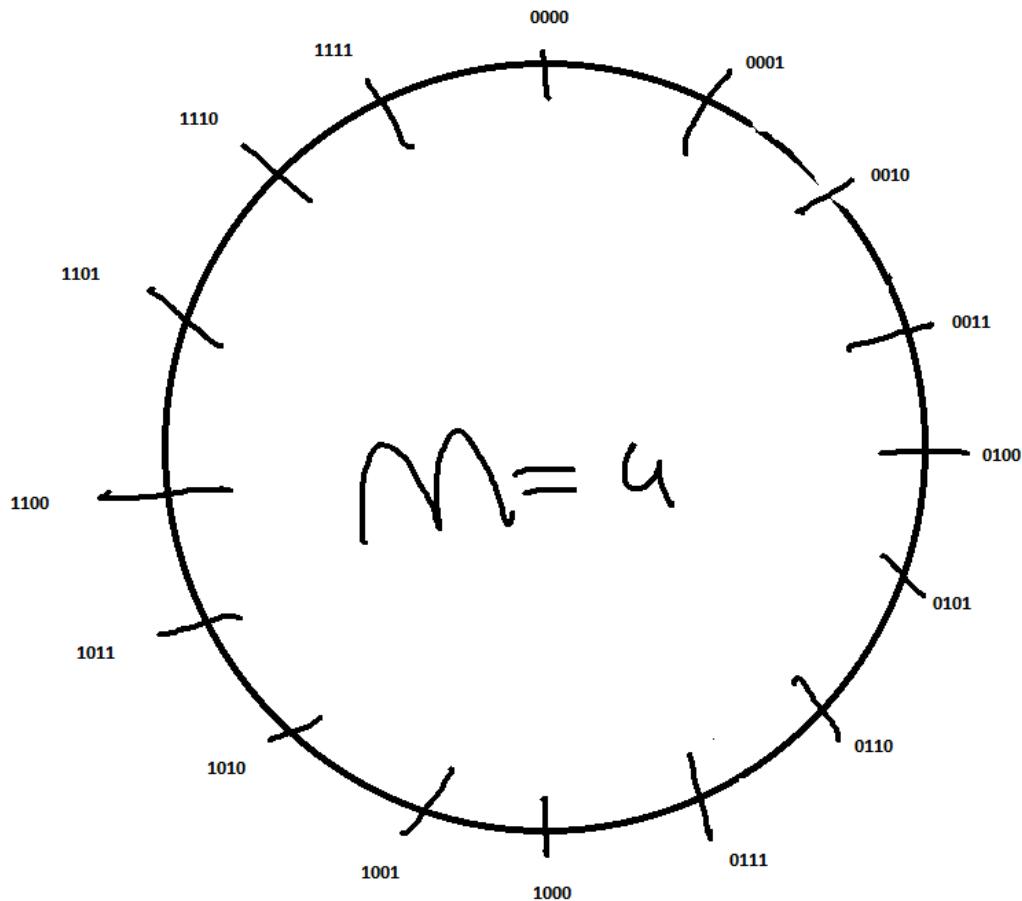
Supponiamo di avere n=4 bit:

$N=5 \rightarrow$ converto 5 su 3 su $n-1$ bit (3) e aggiungo uno 0 a sinistra $\rightarrow 0101$, dove il primo zero indica un numero positivo o nullo, mentre 101 vale 5 sia in complemento a 2, sia in MDS

$N = -5 \rightarrow$ converto $2^4 - |-5| = 11$ su 4 bit $\rightarrow 1011$, cioè 11 nei naturali mentre 5 nel complemento a 2

$N=8 \rightarrow$ converto 8 su 3 bit \rightarrow overflow, 4 bit per codificarlo non bastano
ce ne vogliono almeno 5

$N = -1 \rightarrow 2^4 - |-1| = 15$ su 4 bit $\rightarrow 1111$
 $N = -8 \rightarrow 2^4 - |-8| = 8$ su 4 bit $\rightarrow 1000$
 $N = -11 \rightarrow 2^4 - |-11| = 5$ su 4 bit $\rightarrow 0101$ $5 = -11???$



Per i positivi aggiungendo lo 0 a sinistra l' N più grande è 2^{n-1} mentre nei negativi l' N più grande è -2^{n-1} , ed iniziano tutti con 0.

Ogni numero che sta fuori da $[-2^{n-1}, 2^{n-1}-1]$ non può essere rappresentato in complemento a 2. Lo 0 in complemento a 2 ha una sola codifica.

Se ho $-N$:

- 1) Verifico se posso rappresentarlo;
- 2) Converto N in binario;
- 3) Faccio il complemento a 1 ($0 \rightarrow 1$, $1 \rightarrow 0$);
- 4) Sommo 1 in binario

Per esempio

-6 su 3 bit:

- 1) $-2^2 = -4$ non posso rappresentarlo.

-6 su 4 bit:

- 1) $-2^3 = -8$ posso rappresentarlo;
- 2) 6 in binario = 0110;

- 3) Complemento a 1 = 1001;
 4) Aggiungo 1 = 1010.

Da complemento a 2 alla base 10

$$(111011)_{C2} = - (\text{perché c'è } 1 \text{ a sinistro}) b_{n-1} 2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i =$$

$$-2^4 + 2^3 + 2^1 + 2^0 = -16 + 8 + 2 + 1 = (-5)_{10}$$

Il bit di segno può essere moltiplicato senza modificare il valore

$4 - S$ su 4 bit

$$\begin{array}{r} 0100 \\ 1011 \\ \hline 1111 = (-1) \text{ giusto} \end{array}$$

su 4 bit

$$\begin{array}{r} 1111 \\ 0111 \\ \hline 10110 = (6) \end{array}$$

Vascotto \rightarrow giusto

La somma dei valori in complemento a 2 deve sempre ricadere nell'intervallo $[-2^{n-1}, 2^{n-1}-1]$.

Abbiamo un overflow quando:

- 1) I numeri sono positivi e la somma è codificata come negativa;
- 2) Gli ultimi due riporti sono diversi.

Codifica dei numeri reali

Ad esempio $0.45, \pi, 2,5354355356436, 6,26 \cdot 10^{-34}$

Abbiamo due differenze principali tra i numeri interi e quelli reali:

- I numeri reali non vengono usati per contare ma per "misurare";
- Essi non sono enumerabili.

Non possiamo rappresentarli a precisione infinita ma solo approssimarli a numeri razionali a precisione finita, se lo sviluppo decimale è infinito, esso è periodico.

Come estendiamo la notazione posizionale per rappresentare un numero razionale?

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
-------	-------	-------	-------	-------	-------	-------	-------	---	----------	----------	----------	----------

$$\sum_{i=0}^n b_i \cdot 2^i + \sum_{i=-n}^{-1} b_i \cdot 2^i$$

Per andare da base 10 a base 2 usiamo un altro algoritmo.

Dato ($I.F$, ParteIntera, ParteFrazionaria)₁₀ da convertire in base due:

- 1) Si converte I in binario;
- 2) Si moltiplica F per 2;
- 3) La parte intera del risultato diventa la prima cifra significativa da calcolare;
- 4) Torniamo a 2 considerando la parte frazionaria del risultato di F per 2.

Questo algoritmo potenzialmente va all'infinito, ma può terminare quando:

- La parte frazionaria diventa 0;
- Abbiamo finito i bit a disposizione per la parte frazionaria.

$$\begin{aligned}
 & .4375 \cdot 2 = .875 \text{ Parte int. } 0 \\
 & .875 \cdot 2 = 1.75 \text{ Parte int. } 1 \\
 & .75 \cdot 2 = 1.5 \text{ Parte int. } 1 \\
 & .5 \cdot 2 = 1.0 \text{ Parte int. } 1 \\
 & \text{Si termina}
 \end{aligned}$$

.0111

I numeri con la virgola possono essere rappresentati in vari modi, ad esempio utilizzando la rappresentazione della virgola fissa.

Assegno n_I per i bit della parte intera e per la parte frazionaria ci restano $n - n_I = n_F$, dato che questa distinzione è perenne sappiamo sempre dove va inserita la virgola.

Lezione del 18 Ottobre 2021

Rappresentazione dei numeri reali in \mathbb{R}

Troncamento e arrotondamento

Avendo a disposizione un numero finito di bit, se dovessimo generare un numero con più cifre di quelle a nostra disposizione, ci vediamo costretti a troncare i numeri una volta finiti i suddetti bit.

Il troncamento è un arrotondamento sempre verso lo zero (difetto per i positivi, eccesso per i negativi).

Un altro metodo è l'arrotondamento, dove tronco sempre le cifre in overflow, scegliendo di avvicinarmi o allontanarmi dallo 0 per minimizzare l'errore di arrotondamento.

Esempio:

101.1101101 in 6 bit

Le ultime 4 cifre vanno scartate, inoltre la prima di queste 4 cifre ci fa capire se dobbiamo arrotondare per difetto o per eccesso.

Quindi 101.1101101 diventa 101.111

Per rappresentare i numeri in \mathbb{R} non basta avere la notazione, ma anche quanti bit dare alla parte intera e quanti dare alla parte frazionaria

$$n = n_I + n_F$$

Questa cosa può essere fatta attraverso la rappresentazione in virgola fissa, e attraverso la rappresentazione in virgola mobile.

Virgola fissa

- Assegno un numero di bit alla parte intera ed alla parte frazionaria rimangono $n - n_I$ bit. Il numero di bit assegnati alle due parti non possono essere poi cambiati.

La quantità più vicina allo zero rappresentabile è 2^{-n_F} , esso è il contributo più piccolo del numero, ma anche la "precisione" del numero.



I limiti di questo metodo risiedono nel fatto che l'intervallo di numeri rappresentabili è meno ampio, ma l'errore rimane costante.

Virgola mobile

In questa rappresentazione la virgola non ha una posizione predefinita. Per esempio, 11.011 e -0.0001111, riescono a coesistere in un elaboratore a virgola mobile, ma in uno a virgola fissa no.

In questo modello la posizione della virgola deve essere esplicitata consumando memoria.

Questa rappresentazione è divisa in 3 campi:

- 1 bit per il segno (0 positivo e 1 negativo come in MDS);
- n_e bit per indicare un numero intero con segno chiamato "esponente";
- n_m bit che codificano un numero frazionario detto "mantissa".

segno (s)	esponente (e)	mantissa (m)
-----------	---------------	--------------

Con un determinato numero di bit ottenuto facendo la somma di 1 (bit per il segno) + n_e + n_m .

Il numero sarà quindi uguale $a = (-1)^s \cdot m \cdot 2^e$

L'esponente fa muovere la virgola attraverso la notazione "scientifica". Quindi i numeri precedentemente citati verranno rappresentati come:

11.011			-0.0001111		
s	e	m	s	e	m
0	1	1.1011	1	-4	1.111

La virgola nella mantissa viene messa dopo il primo 1, ovvero dopo la prima cifra significativa, e questo è esplicitato nella forma "normalizzata" della mantissa, nella quale di solito 1. viene omesso ed è quindi implicito.

Esempio:

Non normalizzati $\Rightarrow 1101 \cdot 2^{-1}$ e $10.11 \cdot 2^1$

Normalizzati $\Rightarrow 1.101 \cdot 2^2$ e $1.011 \cdot 2^2$

E' più facile dire chi è più grande/più piccolo dei due.

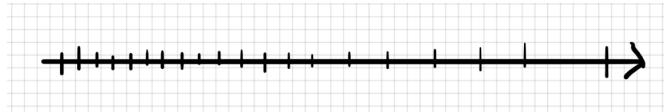
Overflow: più grande dello spazio massimo

Underflow: più piccolo dello spazio minimo

Virgola fissa



Virgola variabile



A numeri piccoli corrispondono errori piccoli, mentre a numeri grandi corrispondono errori grandi.

Standard IEEE 754

L'implementazione della rappresentazione standardizzata in virgola mobile negli elaboratori è regolata da "IEEE-SA standard number 754 for floating-point arithmetics". Essa viene espressa in due formati:

- Formato a precisione singola (float) su 32 bit, quella che andremo ad analizzare;
- Formato a precisione doppia (double) su 64 bit.

In precisione singola un numero in virgola mobile ha: n=1 bit per il segno, n=8 bit per l'esponente e n=23 bit per la mantissa.

s	e	m
---	---	---

Con questo formato possiamo rappresentare:

- Numeri normalizzati;
- Numeri denormalizzati, chiamati anche non normalizzati oppure sub-normalizzati;
- Codici speciali, ad esempio $\pm \infty$ e 0^+ , 0^- .

Perchè un numero sia normalizzato $0 < E < 255$ con gli estremi esclusi. E è il valore dell'esponente al quale è stato precedentemente sommato 127, quindi $e = E - 127$, (esponente in eccesso 127).

Infine il numero sarà uguale $a \Rightarrow (-1)^s \cdot 1.m \cdot 2^e$

L'esponente più piccolo sarà E=1, quindi e=-126

L'esponente più grande sarà E=254, quindi e = 127

Laboratorio del 19 Ottobre 2021

In un calcolatore abbiamo una sequenza limitata di interruttori, quindi di bit (0 e 1). Vediamo la codifica di numeri con la virgola.

Numeri frazionari

Con metodo posizionale un numero 0.F viene rappresentato con la formula:

$$\sum_{i=-m}^{-1} c_i \cdot B^i$$

Esempio: $(0.587)_{10} = 5 \cdot 10^{-1} + 8 \cdot 10^{-2} + 7 \cdot 10^{-3}$

Trasformazione di un numero frazionario in base 10 in base 2

- 1) La parte intera I viene trasformata in binario con il metodo che conosciamo già;
- 2) La parte frazionaria .F viene moltiplicata per 2;
- 3) La parte intera del risultato diventa la cifra più significativa;
- 4) Torniamo allo step 2 considerando come .F la parte frazionaria del risultato.

Questo algoritmo termina quando:

- La parte frazionaria del risultato diventa uguale a 0, e quindi ripetendo questo algoritmo avremo una serie infinita di 0;
- Finiamo il numero di bit predisposti per rappresentare il numero.

Handwritten notes showing the conversion of 0.587 from base 10 to base 2 using the multiplication by 2 method:

$$0.587 \cdot 2 = 1,174 \quad \text{parte intera } 1$$
$$0.174 \cdot 2 = 0.348 \quad \text{parte intera } 0$$
$$0.348 \cdot 2 = 0.696 \quad \text{parte intera } 0$$
$$0.696 \cdot 2 = 1.392 \quad \text{parte intera } 1$$
$$0.392 \cdot 2 = 0.784 \quad \text{parte intera } 0$$

. . . - - -

Abbiamo due metodi per rappresentare i numeri frazionari in binario: rappresentazione a virgola fissa e rappresentazione a virgola mobile.

Rappresentazione a virgola fissa

In questa rappresentazione abbiamo un numero prefissato di bit per la parte intera (n_i) così che alla parte decimale rimangano $n_{TOT}-n_i$ bit.

La virgola cade in una e una sola rappresentazione, perché in questa rappresentazione è implicita.

Rappresentazione in virgola mobile

La formula generale di rappresentazione dei numeri in virgola mobile è la seguente:

$$(N)_B = (-1)^s \cdot m \cdot B^e$$

In questa formula:

- s è il segno (0 o 1);
- m è la mantissa, dove la parte frazionaria di N si chiama p ed è anche detta precisione della mantissa, con 1. implicito;

- e è l'esponente di B.

Il minimo della mantissa è 1 mentre il massimo è < 2

Prendendo due numeri generali in VirgolaFissa ed in VirgolaMobile:
 VF iii.fff VM i.fff*10^{ee}

	V min	V max
VF	000.000	999.999 $\approx 10^3$
VM	0.000*10 ⁰⁰	9.999*10 ⁹⁹

	VF	VM
Risoluzione o precisione	000.001	1.000*10 ^{ee}
	Fissa!	Dipende dall'esponente!

Avendo una certa variabile x, e la sua risolutezza r(x), l'errore assoluto sarà uguale a:

$$e_A(x) = x - r(x)$$

Mentre l'errore relativo sarà uguale a:

$$e_r(x) = \frac{x-r(x)}{x} = \frac{e_A(x)}{x}$$

Possiamo quindi inserire questi valori in una tabella

	VF	VM
e _A	<10 ³	<10 ^{-3*10^{ee}}
	Errore assoluto fisso	Dipende dall'esponente
e _r	$[\frac{10^{-3}}{10^3}, \frac{10^{-3}}{0}]$	$\frac{\text{MAX } e_A(x)}{\min(x)} = \frac{10^{-3} \cdot 10^{ee}}{10^{ee}} = 10^{-3}$
	Errore relativo variabile	Errore relativo fisso

Standard IEEE 754

Esso definisce lo standard a precisione singola che utilizza 32 bit per rappresentare i numeri con la virgola, con:

- 1 bit per il segno s;
- 8 bit per l'esponente E;
- 23 bit per la mantissa.

E non è uguale al nostro precedente e, infatti E è rappresentato in eccesso 127, quindi $E = e + 127$, ed infine $e = E - 127$. Il valore di e quindi varia da -126 a 127.

I valori dovrebbero essere da 0 a 255, però il valore iniziale e quello finale vengono utilizzati per rappresentare codici speciali.

La mantissa varia da [1,2) senza rappresentare il 1. nei 23 bit

Esempio 1:

$$(+17.375)_{10} \xrightarrow{1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1}$$

$$-s = 0 \text{ pos}$$

$$-(17)_{10} = (10001)_2$$

$$-(0.375)_{10} = (0.011)_2$$

$$-(17.375)_{10} = (10001.011)_2$$

Forma normalizzata

$$1.0001011 \cdot (10)_2$$

$$e = 9$$

$$E = 131 = 10000011$$

Quindi

$$s = 0 \quad m = 0001011$$

$$l = 10000011$$

Esempio 2:

$$(-0.8)_{10}$$

$$S = 1$$

$$(0.8)_{10} = (0.\overline{1100})_2$$

$$(1.\overline{1001})_2 \cdot (\overset{-1}{10})_2$$

$$e = -1 \quad E = 126$$

$$E = (0111110)_2$$

Quindi:

$$S = 1$$

$$e = 0111110$$

$$m = 10011001100\dots$$

Se:

- $0 < E < 255$ (quindi $e \in [-126, 127]$) allora abbiamo un numero normalizzato;
- $m=0, E=0$ abbiamo ± 0 , dipende dal segno;
- $m=0, E=255$ abbiamo $\pm \infty$, dipende dal segno;
- $m \neq 0, E=255$ abbiamo Nan (Not a number);
- $m \neq 0, E=0$ abbiamo un numero subnormalizzato.

Se prendiamo il numero normalizzato più piccolo, ovvero quello con segno 0, esponente -126 e mantissa uguale a $1.00000\dots$, il nostro numero rientrerà nell'ordine del 10^{-38} .

Se invece all'esponente inseriamo -127, il compilatore lo trasformerà in -126 e inserirà nella parte intera della mantissa, al posto di 1., 0., facendo così arrivare la precisione del numero nell'ordine del 10^{-45} .

Esercizio 1

Da B_{10} a B_2

$$(33 \cdot 1001)_{10} = (100001 \cdot F)_2$$

$F \cdot 2 = 2002$ parte intera 0

$$0 \cdot 2002 \cdot 2 = 0 \cdot 4004 \quad 0$$

$$0 \cdot 4004 \cdot 2 = 0 \cdot 8008 \quad 0$$

$$0 \cdot 8008 \cdot 2 = 1 \cdot 6016 \quad 1$$

$$0 \cdot 6016 \cdot 2 = 1 \cdot 2032 \quad 1$$

$$0 \cdot 2032 \cdot 2 = 0 \cdot 4064 \quad 0$$

$$(33 \cdot 1001)_{10} = 100001.000110\dots$$

Esercizio 2

Da B_2 a B_{10}

$$(110001.100110)_2$$

$$= (2^5 + 2^4 + 1 + 2^1 + \frac{1}{16} + \frac{1}{32} + \frac{1}{128})_{10}$$

$$= 49.6015625$$

Esercizio 3

- 24.511 a bit parte F

$$\left[-2^{\frac{m-1}{2}}, 2^{\frac{m-1}{2}} \right]$$

$$2^{-4} - 8 - 16 - 32 \quad \begin{matrix} S = m-1 \\ m=6 \end{matrix}$$

$$24 \text{ in } B_2 = 011000 \Rightarrow 100111 + 1 \\ \Rightarrow 101000$$

$$N = (101000.F)_2 = (101000,1000)$$

$$\begin{array}{rcl} .511 \cdot 2 = 1.022 & \wedge & \\ 0.022 \cdot 2 = 0.044 & 0 & \\ 0.044 \cdot 2 = 0.088 & 0 & \\ 0.088 \cdot 2 = 0.176 & 0 & \end{array}$$

Esercizio 4

$$S = 1$$

$$24.511 = 11000.1000$$

forma normale rigata

$$1.10001 \cdot (10)^4_2$$

$$l = 4 + 12 = 131$$

$$m = 1000100 \dots$$

$$l = 10000011$$

Lezione del 21 Ottobre 2021

Numeri de-normalizzati

Se $e=0$ e $m \neq 0$ stiamo codificando un numero sub-normalizzato dove:

- s è sempre il segno;
- m è la parte frazionaria (diversa da 0) con parte intera "0.";
- e viene scartato e si assume che sia -126.

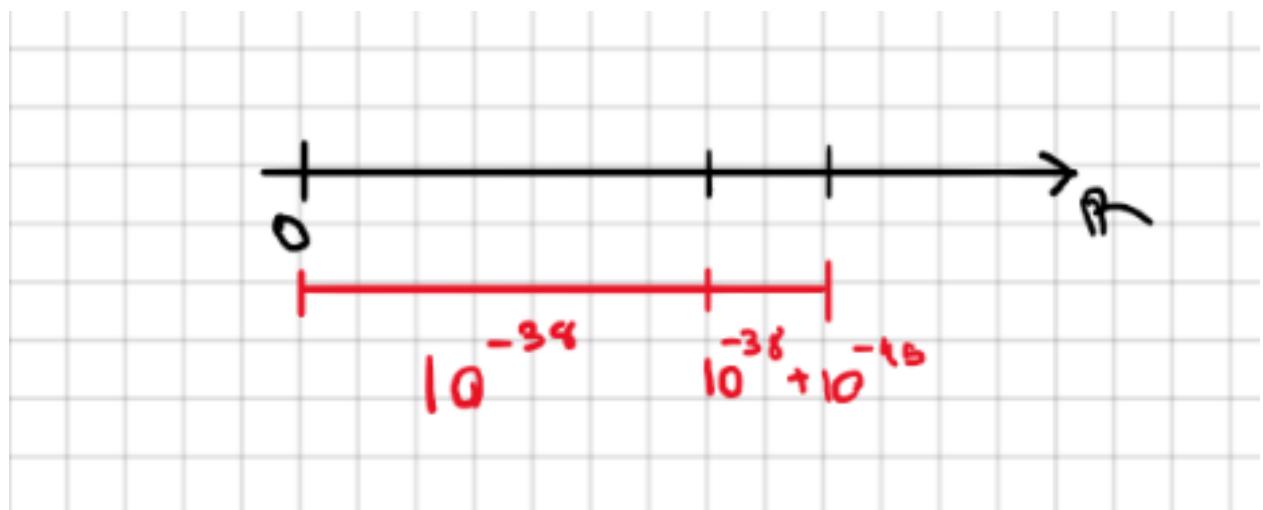
Il numero rappresentato questa volta è:

$$(-1)^s \times 0.m \times 2^{-126}$$

Il numero normalizzato più vicino allo 0 è $\Rightarrow 2^{-126} \approx 1,17 \times 10^{-38}$

Il numero successivo a questo numero è

$$\Rightarrow 2^{-126} + 2^{-149} \approx 1,17 \times 10^{-38} + 1.4 \times 10^{-45}$$



Invece, il numero denormalizzato più vicino allo 0 è

$$\Rightarrow 2^{-23} \times 2^{-126} \approx 1,4 \times 10^{-45}$$

Codici speciali

e	m	IEEE754
$0 < e < 255$	qualsiasi	numero normalizzato
$e = 0$	$m \neq 0$	numero subnormalizzato
$e = 0$	$m = 0$	± 0 (dipende dal segno)
$e = 255$	$m = 0$	$\pm \infty$ (dipende dal segno)
$e = 255$	$m \neq 0$	NaN (Not a number)

Considerazioni

Perché l'esponente è in eccesso 127?

Questo rende più facile il confronto fra due numeri:

- se i segni sono diversi, il numero con segno 0 è il più grande;
- se i segni sono uguali, l'esponente domina sulla mantissa, quindi quello con l'esponente più grande è il maggiore.

Funzioni e parti logiche elementari

Come si elabora un circuito in grado di rappresentare ed elaborare le informazioni?

Un circuito digitale può essere pensato da tanti piccoli elementi, capaci di svolgere piccole operazioni di elaborazione logica elementare.

Connettendo tra loro tali elementi, saremo poi in grado di svolgere elaborazioni complesse.

Le elaborazioni logiche sono definite nell'algebra di Boole

Algebra di Boole

Un'algebra è un insieme di simboli, valori e regole per svolgere operazioni su di essi. L'algebra di Boole comprende simboli e valori binari su cui possiamo svolgere operazioni logiche.

Le variabili possono essere TRUE (1) o FALSE (0). I valori di queste variabili vengono influenzati e cambiati attraverso tre operazioni logiche: NOT, AND e OR.

Combinando le variabili e gli operatori otteniamo le funzioni logiche:

- l'insieme delle variabili è uguale a $B = \{0, 1\}$;
- variabile $a \in B$;
- funzioni della forma $f^n: B \rightarrow B$.

NOT negazione booleana

Essa si indica con \bar{a} e la sua tabella di verità è:

a	\bar{a}
0	1
1	0

AND congiunzione booleana

Si indica con ab (oppure $a \wedge b$), se entrambe valgono 1, and vale 1. Viene anche chiamato prodotto logico, anche se sarebbe più corretto dire minimo logico. La sua tabella di verità è:

a	b	ab
0	0	0

0	1	0
1	0	0
1	1	1

OR disgiunzione logica

Si indica con $a+b$ (oppure con $a \vee b$), se almeno una delle due vale 1, l'or vale 1. Viene chiamato somma logica, anche se sarebbe più corretto pensarlo come massimo logico. La sua tabella di verità è:

a	b	$a+b$
0	0	0
0	1	1
1	0	1
1	1	1

Precedenza tra operatori

- NOT precedenza su AND e OR;
- AND precedenza su OR.

Esempio

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

Prima il NOT, poi l'AND e infine l'OR.

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

Principio di dualità

Avendo una funzione, la sua duale si ottiene:

- Scambiando gli AND e gli OR;
- Scambiando gli 0 e gli 1.

Sempre per il principio di dualità, se una funzione è valida allora anche la sua duale è valida.

$$a + \bar{a} = 1 \text{ la sua duale } a\bar{a} = 0$$

$$(a\bar{b}) + 1 = 1 \text{ la sua duale } (a + \bar{b})0 = 0$$

Entrambe vere con duali vere.

Proprietà degli operatori

	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	$(ab)c = a(bc)$	$(a + b) + c = a + (b + c)$
	AND rispetto ad OR	OR rispetto ad AND
Distributiva	$a(b + c) = ab + ac$	$a + bc = (a + b)(a + c)$
Assorbimento 1	$a(a + b) = a$	$a + ab = a$
Assorbimento 2	$a(\bar{a} + b) = ab$	$a + \bar{a}b = a + b$
De Morgan	$\overline{ab} = \bar{a} + \bar{b}$	$\overline{a + b} = \bar{a}\bar{b}$

Le proprietà evidenziate sono i postulati

Dimostrazione delle altre proprietà

Ipotesi: Dualità e Postulati

Tesi: Idempotenza $aa = a$

Passaggio	Ottenuto grazie a....
a	
$= a + 0$	Identità ($0 + a = a$)
$= a + a\bar{a}$	Inverso ($a\bar{a} = 0$)
$= (a + a)(a + \bar{a})$	Distributiva ($a + bc = (a + b)(a + c)$)
$= (a + a)1$	Inverso ($a + \bar{a} = 1$)
$= aa + 0 = aa$	Duale

Ipotesi: Dualità, Postulati, Idempotenza

Tesi: Elemento nullo $1 + a = 1$

Passaggio	Ottenuto grazie a....
$1 + a$	

$= a + \bar{a} + a$	Inverso ($a + \bar{a} = 1$)
$= a + \bar{a}$	Idempotenza ($a + a = a$)
$= 1$	Inverso ($a + \bar{a} = 1$)

Ipotesi: Dualità, Postulati, Idempotenza, Elemento nullo

Tesi: Assorbimento 1 $a(a + b) = a$

Passaggio	Ottenuto grazie a...
$a(a + b)$	
$= aa + ab$	Distributiva ($a(b + c) = ab + ac$)
$= a + ab$	Idempotenza ($aa = a$)
$= a1 + ab$	Identità ($1a = a$)
$= a(1 + b)$	Distributiva ($a(b + c) = ab + ac$)
$= 1a$	Elemento nullo ($1 + a = 1$)
$= a$	Identità ($1a = a$)

Dimostrare le altre

Lezione del 25 Ottobre 2021

Esercizio

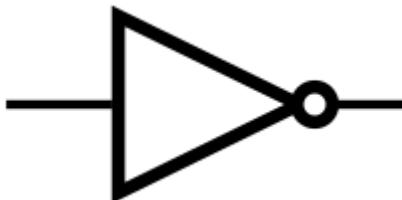
Semplificare: $(\overline{\overline{ab}} + \bar{c}) \overline{\overline{ab}} + \overline{\overline{abc}}$

Passaggio	Ottenuto grazie a ...
$(\overline{\overline{ab}} + \bar{c}) \overline{\overline{ab}} + \overline{\overline{abc}}$	
$= \overline{\overline{abc}} + \overline{\overline{abc}}$	Assorbimento 2 ($a(\bar{a} + b) = ab$)
$= \overline{\overline{ab}}(\bar{c} + c)$	Distributiva ($a(b + c) = ab + ac$)
$= \overline{\overline{ab}}$	Elemento inverso
$= a + b$	De Morgan ($\overline{\overline{ab}} = \bar{a} + b$)

Porte logiche

Esse sono delle controparti hardware delle espressioni dell'algebra di Boole. Quindi questi strumenti ci permettono di trasformare la tensione e la combinazione di essi in operazioni logiche capaci di acquisire segnali in input e restituire un output coerente. Queste porte logiche si indicano graficamente con i seguenti simboli:

- NOT



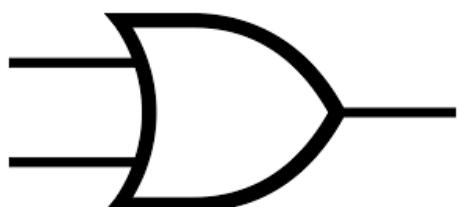
Prende in input in a e restituisce il valore di \bar{a} con il complemento a 2.

- AND



Prende in input a e b e restituisce ab .

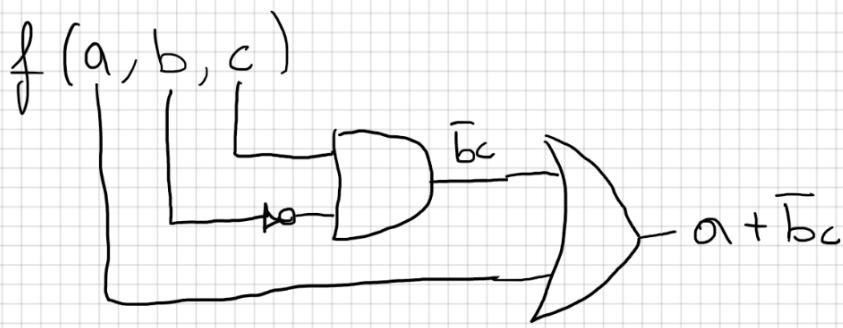
- OR



Prende in input a e b e restituisce $a + b$.

Noi possiamo soltanto controllare gli input e osservare gli output.
Attraverso queste porte logiche possiamo rappresentare espressioni booleane complesse.

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



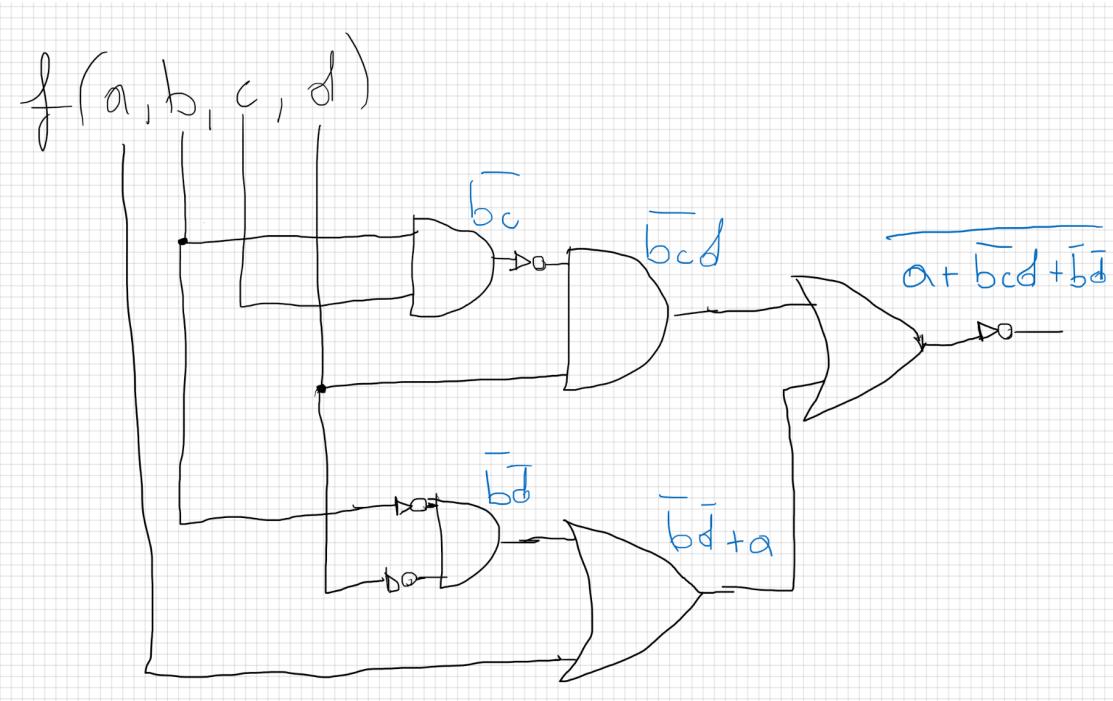
Gli input devono essere allineati.

Questo è il nostro primo circuito combinatorio, il quale combina gli input per ottenere gli output.

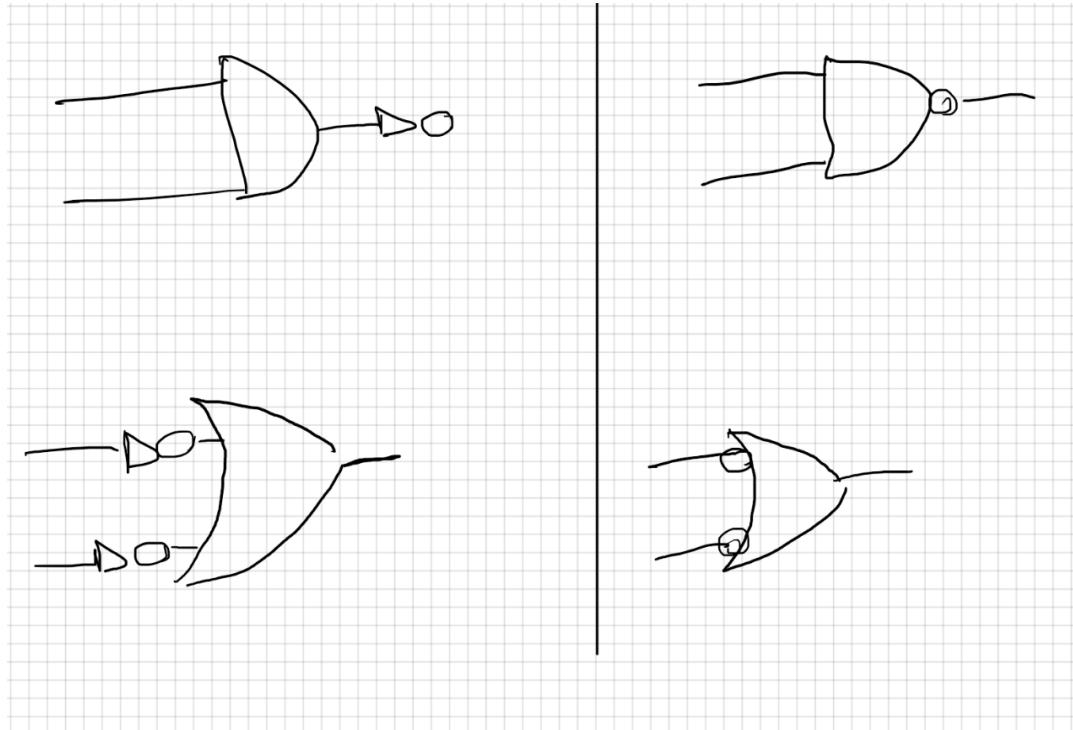
L'elaborazione procede dagli input agli output.

L'uscita dipende solo dagli input, infatti a input uguali corrispondono output uguali.

Esercizio: $f(a, b, c, d) = \overline{a + \overline{bcd} + \overline{bd}}$



Notazioni alternative con il NOT



Rappresentazione di una funzione logica

In generale, in una funzione f definita su algebra Booleana, abbiamo tre modi di rappresentarla:

- Espressione Booleana;
- Circuito combinatorio;
- La tabella di verità.

I primi due metodi sono metodi compatti, mentre la tabella di verità ha una riga per ogni possibile combinazione di valori in input e che specifica, su ogni riga, il valore della funzione nella configurazione corrispondente. Tutto ciò è possibile perché su base 2 non abbiamo problemi di ambiguità.

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

Espressione booleana: $a + \bar{b}c$

Circuito combinatorio:

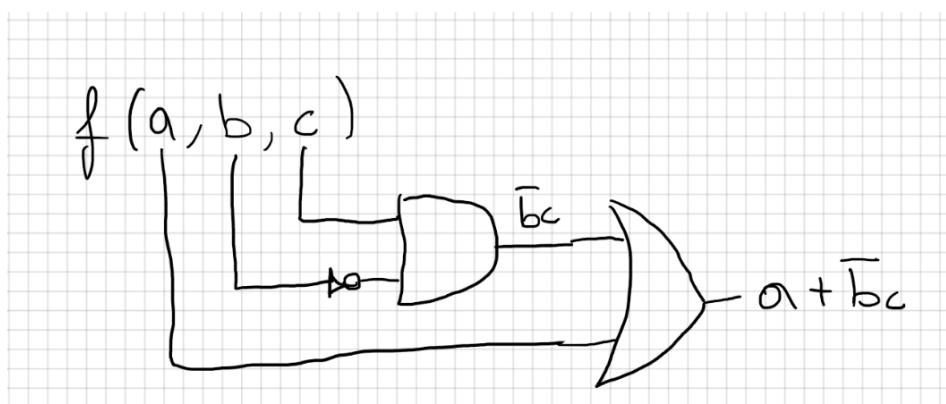


Tabella di verità:

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

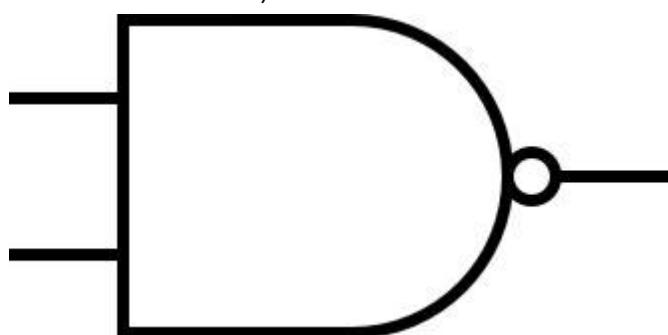
Data una funzione $f: B^n \rightarrow B$, la tabella di verità costituisce un metodo esaustivo di rappresentazione, con 2^n righe e $n + 1$ colonne.

Operatori composti

NOT, AND e OR sono gli operatori logici elementari. Gli operatori composti sono più complessi ma più comodi da utilizzare poiché attraverso di essi possiamo ricavare gli operatori elementari.

NAND

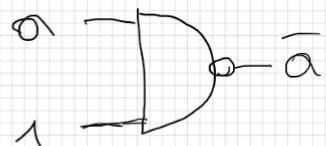
Definito anche come Not AND, esso è un and negato, quindi l'opposto di AND. NAND è una porta elementare la quale può essere utilizzata per ricostruire NOT, AND e OR.



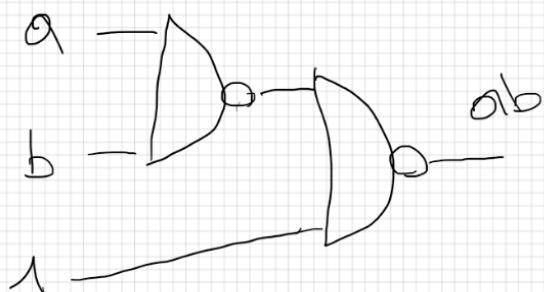
a	b	\bar{ab}
0	0	1
0	1	1
1	0	1

1	1	0
---	---	---

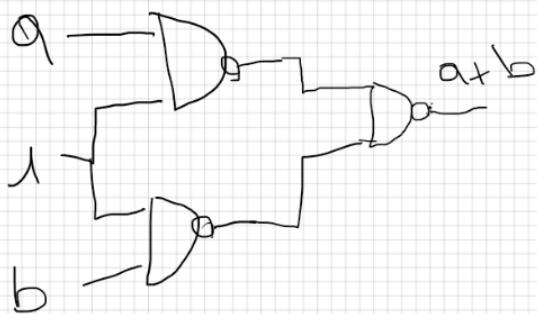
NOT



AND



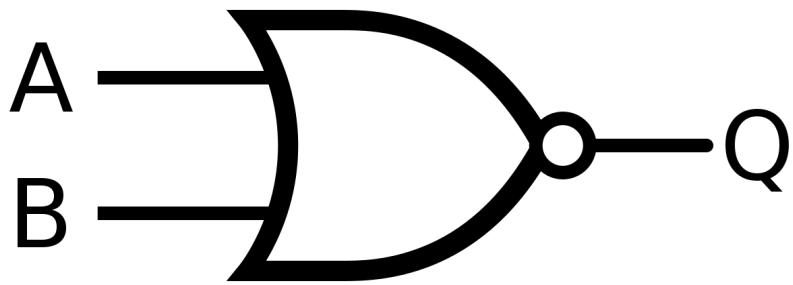
OR



NOR

OR negato all'uscita, vale 1 solo quando entrambi gli input sono uguali a 0.

Anche NOR è un componente universale.



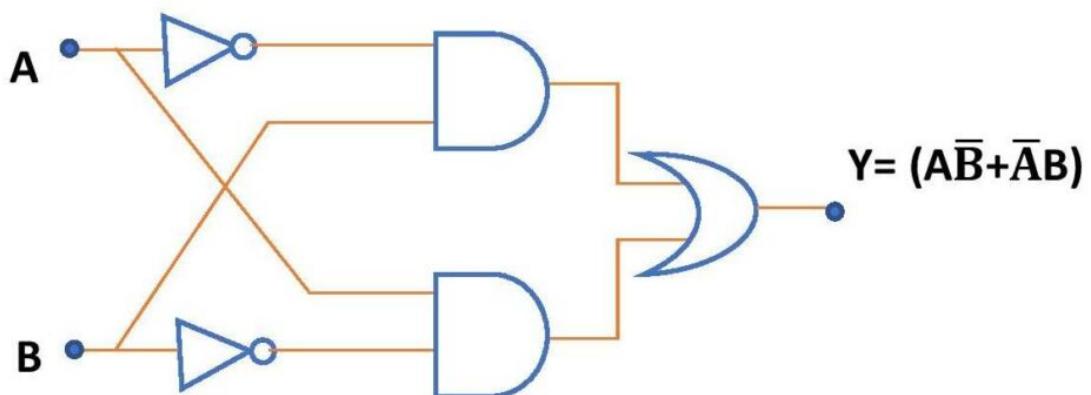
a	b	$\overline{a + b}$
0	0	1
0	1	0
1	0	0
1	1	0

XOR

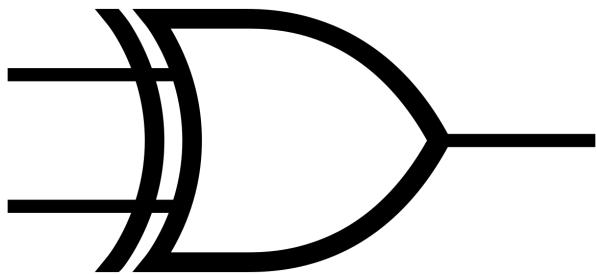
Anche definito come exclusive OR, quindi è un OR esclusivo con operatore Booleano \oplus .

Vale 1 solo quando uno ed uno solo degli input è 1.

Si esprime con $a\bar{b} + \bar{a}b$



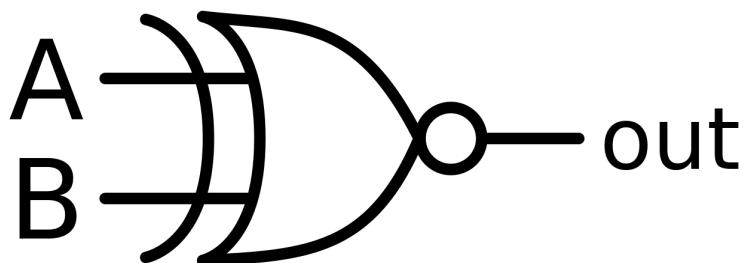
Il simbolo di XOR è



XOR ha tre interpretazioni principali:

- Funzione di diversità, vale solo con i bit diversi;
- Somma di a e b, con complemento quando vi è un 1+1;
- Terza implementazione che andremo poi ad introdurre in futuro.

Per la funzione di uguaglianza dobbiamo negare lo XOR, creando lo XNOR.



Analisi e sintesi di un circuito: forma canonica

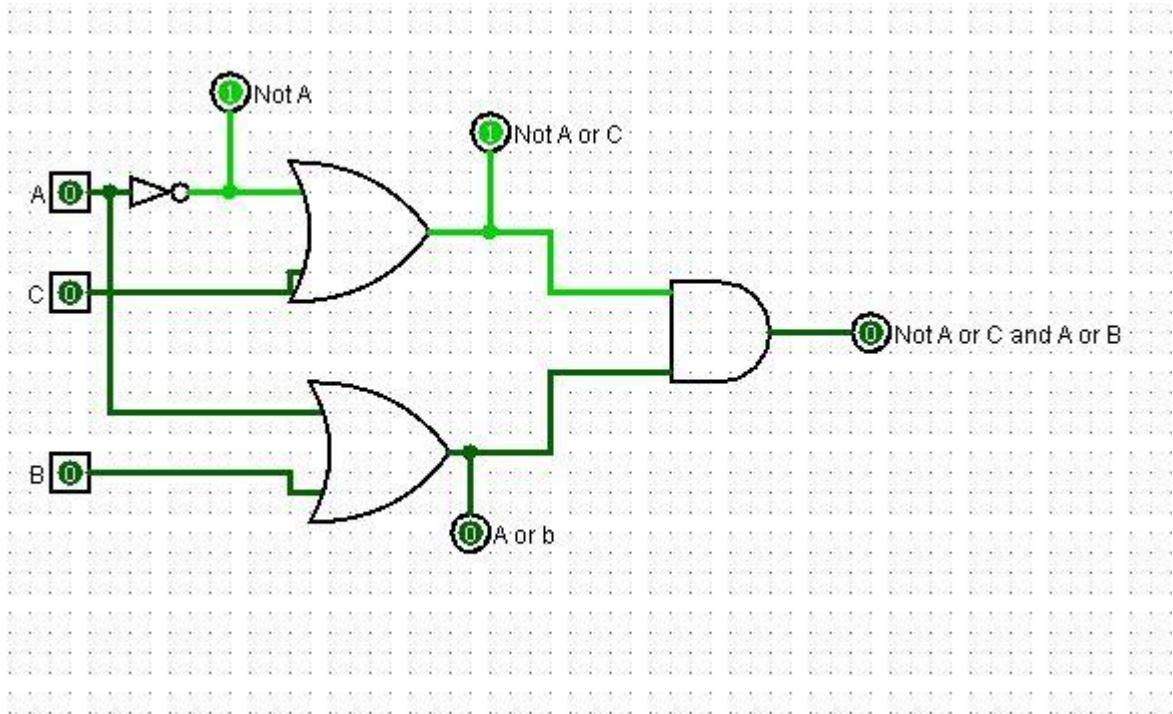
- Analisi, attraverso la quale a partire dalla tabella di un circuito si costruisce la tabella di verità;
- Sintesi, attraverso la quale attraverso la tabella o l'espressione booleana costruiamo il circuito che lo implementa.

Prima forma canonica

Partendo dalla tabella di verità indichiamo tutti i punti in cui il valore è 1. Attraverso l'ipotesi del mondo chiuso sappiamo che tutti i punti non indicati hanno valore 0. A quel punto basterà sommare le casistiche in cui la funzione è uguale a 1.

Laboratorio del 26 Ottobre 2021

Esercizi su logisim
Esercizio 1



A	B	C	Not A	Not A or C	A or B	Not A or C and A or B
0	0	0	1	1	0	0
0	0	1	1	1	0	0
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	0	0	1	0
1	0	1	0	1	1	1
1	1	0	0	0	1	0
1	1	1	0	1	1	1

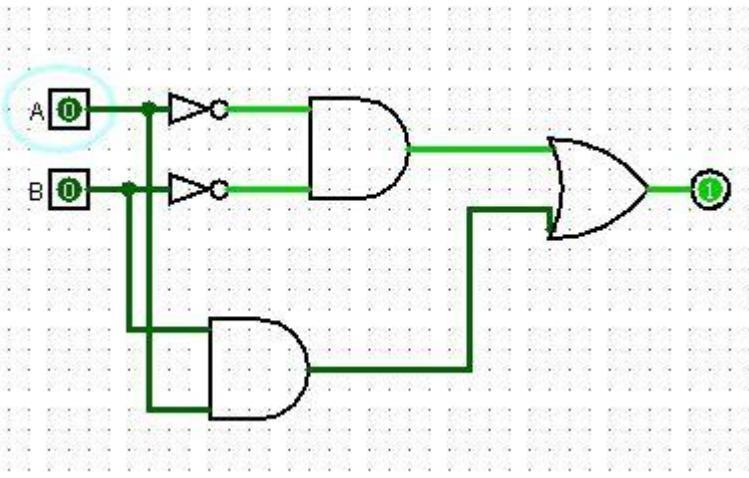
Esercizio 2

Derivare XNOR senza usare operatori composti

$$a \text{xnor } b = \overline{ab} + ab$$

a	b	\overline{ab}	ab	$\overline{ab} + ab$
0	0	1	0	1
0	1	0	0	0
1	0	0	0	0

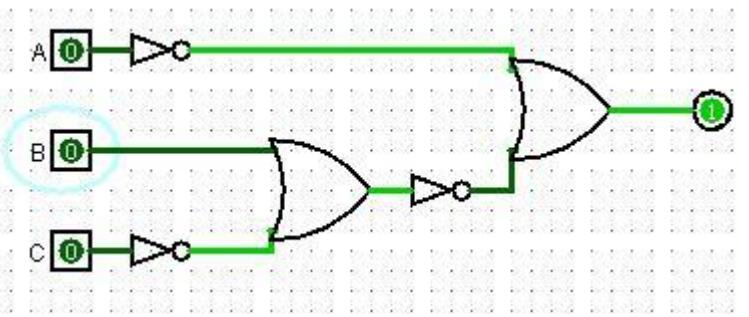
1	1	0	1	1
---	---	---	---	---



Esercizio 3

$$\bar{a} + (\overline{b + \bar{c}})$$

a	b	c	\bar{c}	\bar{a}	$b + \bar{c}$	$\overline{b + \bar{c}}$	$\bar{a} + (\overline{b + \bar{c}})$
0	0	0	1	1	1	0	1
0	0	1	0	1	0	1	1
0	1	0	1	1	1	0	1
0	1	1	0	1	1	0	1
1	0	0	1	0	1	0	0
1	0	1	0	0	0	1	1
1	1	0	1	0	1	0	0
1	1	1	0	0	1	0	0



Esercizio 4

Verificare che E_1 e E_2 siano uguali

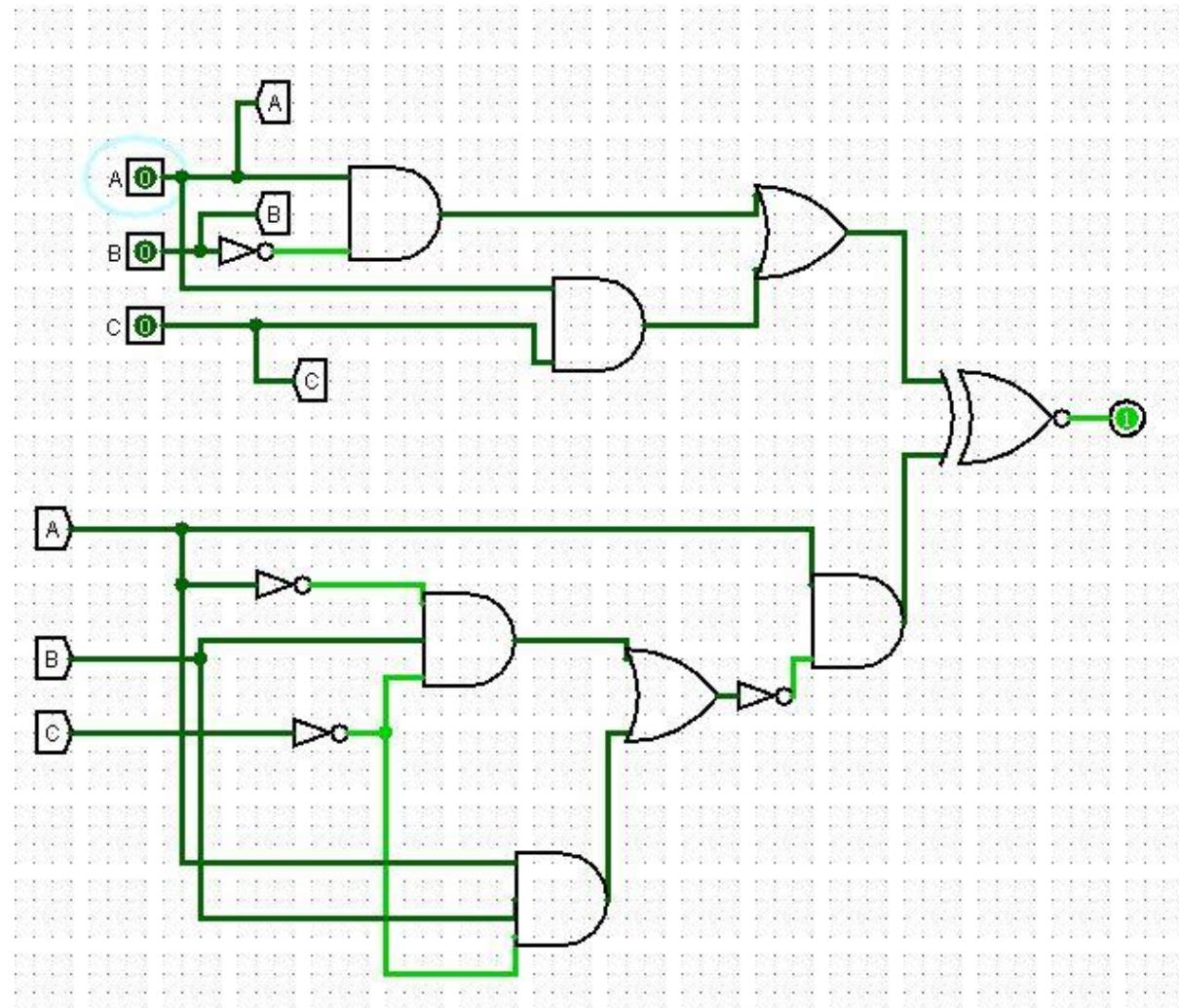
$$E_1 = \overline{\overline{abc} + abc}a$$

$$E_2 = \overline{ba} + ac$$

Semplificare E_1

$$(\overline{abc} + abc)a \Rightarrow (\overline{bc}(\overline{a} + a))a \Rightarrow \overline{bc}a \Rightarrow (\overline{b} + c)a \Rightarrow \overline{ba} + ac$$

identica a E_2



Esercizio 5

Trasformare la seguente espressione E_1 utilizzando solo l'operatore NAND, per poi confrontarlo con l'espressione E_2 .

$$E_1 = (a \text{ NOR } b)(c + \overline{b})$$

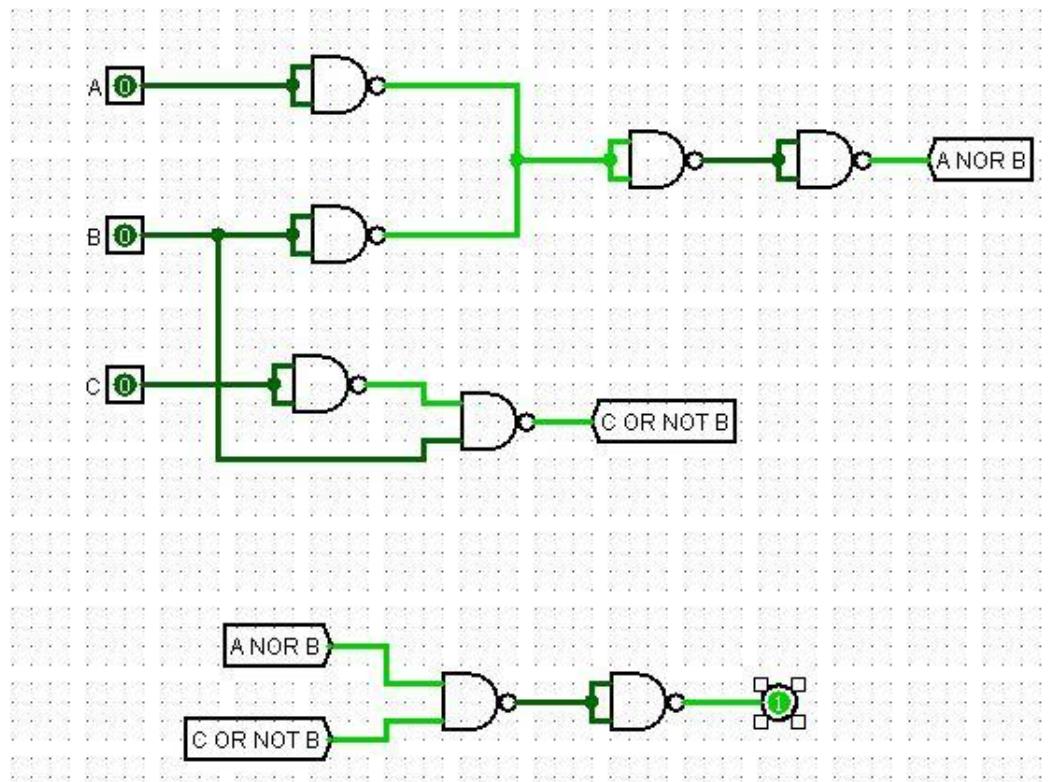
$$E_2 = \overline{ab}$$

$$\overline{b} = \text{NAND } b$$

$$a \text{ NOR } b = \text{NAND}[(\text{NAND } a) \text{ NAND } (\text{NAND } b)]$$

$$c + \overline{b} = (\text{NAND } c) \text{ NAND } b$$

$$(a \text{ NOR } b)(c + \overline{b}) = \text{NAND}\{\{\text{NAND}[(\text{NAND } a) \text{ NAND } (\text{NAND } b)]\} \text{ NAND } [(\text{NAND } c) \text{ NAND } b]\}$$



$$E_1 = (a \text{ NOR } b)(c + \bar{b})$$

$$E_2 = \overline{ab}$$

$$E_1 = \overline{(a + b)}(c + \bar{b})$$

$$= (\overline{ab})(c + \bar{b})$$

$$= \overline{abc} + \overline{abb}$$

$$= \overline{abc} + \overline{ab}$$

$$= (\overline{ab})(c + 1)$$

$$= \overline{ab}$$

Lezione del 28 Ottobre 2021

Prima forma canonica

Indichiamo la funzione partendo dalla sua tabella di verità e indichiamo tutti i punti in cui essa è uguale a 1. Per l'ipotesi del mondo chiuso gli altri punti avranno per forza valore 0 (descrizione esaustiva all'interno dell'algebra di Boole).

Per ricavare la funzione logica dobbiamo prendere tutti i punti in cui la funzione è 1, unendo le variabili con gli AND e negandole per farle diventare tutte 1 usando i NOT, per poi essere uniti attraverso gli OR.

Ciascun termine dove la funzione equivale a 1 viene chiamato "Mintermine", il nome deriva dal fatto che la funzione AND corrisponde al minimo delle variabili.

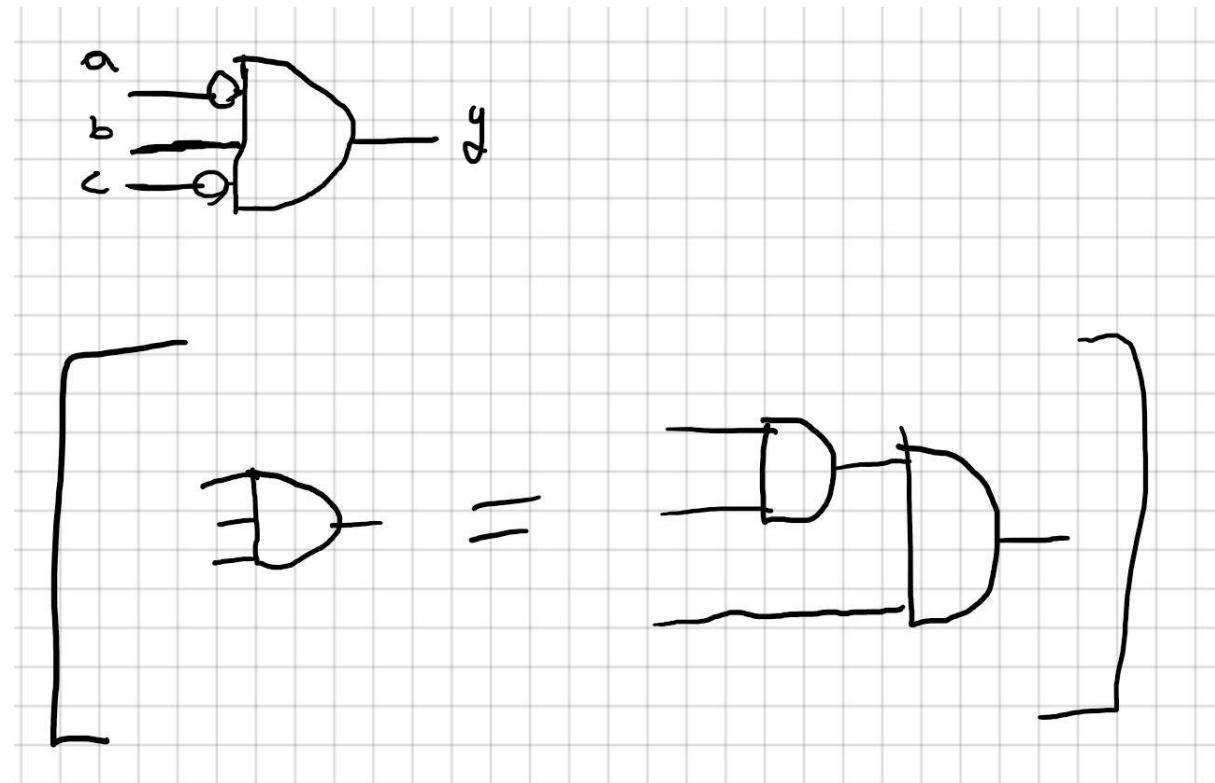
Vi sono tanti mintermini quanti 1 nella funzione.

Per creare il mintermine con porta logica AND che dia 1 come output, le variabili in ingresso che valgono 0 vengono negate con i NOT.

Se chiamiamo m_i , l' i -esimo mintermine, la funzione sarà la sommatoria tra tutti i mintermini (grande OR tra tutti gli AND).

La prima forma canonica viene anche chiamata anche "somma tra prodotti".

Se ad esempio dobbiamo esprimere $\bar{a}\bar{b}\bar{c}$:



Per porte a n ingressi abbiamo $n-1$ porte logiche a 2 uscite, con la prima variabile che entra in tutte le porte mentre l'ultima variabile entra solo nell'ultima porta logica.

Nella prima forma canonica distinguiamo 2 stadi, lo stadio AND con i mintermini e lo stadio con l'OR.

Per ogni mintermine abbiamo una porta logica AND con numero di ingressi dipendente dal numero di variabile, e abbiamo un ingresso nella porta OR.

La prima forma canonica a volte risulta essere più dispendiosa, anche se è più comoda per i computer poiché divisa in stadi.

Se nei mintermini non sono presenti tutte le variabili essi prenderanno il nome di "Implicanti" ed esse sintetizzano i mintermini.

Seconda forma canonica

Se posso descrivere una funzione partendo dai casi in cui essa è vera, posso farlo anche partendo dai casi in cui essa è falsa (e quindi uguale a 0).

Facendo così abbiamo i punti dove la funzione è falsa, combinando i termini con le variabili negate (portandole sempre da 0 a 1) otteniamo la somma dei punti in cui la funzione è negata, quindi

$$\bar{y} \Rightarrow y = \overline{\text{AND}} + \text{AND} + \text{AND} + \text{AND}.....$$

$$y = (\bar{a} + \bar{b} + \bar{c})(\bar{a} + \bar{b} + \bar{c})(\bar{a} + \bar{b} + \bar{c})(\bar{a} + \bar{b} + \bar{c})....$$

Questo passaggio avviene grazie alla legge di de Morgan, e le variabili vengono ri-negate dopo la prima negazione (se valevano 0 vengono negate 2 volte, se valevano 1 vengono negate 1 volta).

In questo caso gli OR vengono detti "Maxtermini".

Abbiamo tanti maxtermini quanti 0 della funzione, e la seconda forma canonica prende quindi il nome di "prodotto di somme".

Anche nella seconda forma canonica sono presenti due stadi: stadio OR e stadio AND.

Valutazione dei costi e delle prestazioni

La valutazione di costi e prestazione è limitata principalmente da due fenomeni fisici:

- "Propagation delay", tempo Δt nel quale il cambio di input si passa in tutte le porte logiche restituendo un output stabile;
- "Fan-Out limitato", cioè il numero di collegamenti in uscita subisce limitazioni, ed influisce anche sulla velocità dell'output.

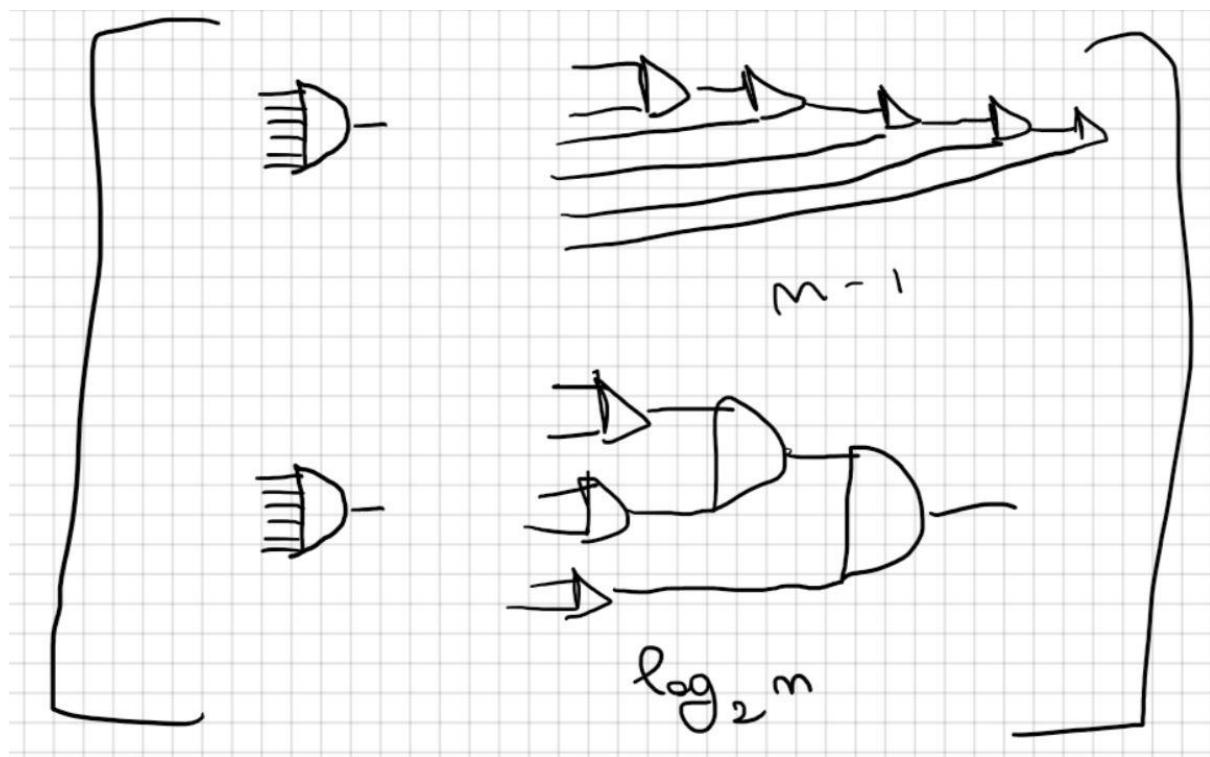
Cammino critico

Il propagation delay può essere stimato attraverso lo studio del cammino critico, cioè il percorso del segnale massimo che comporterà un rallentamento della stabilizzazione del segnale in uscita una volta cambiato l'input di Δt per ogni porta.

Il ritardo della stabilizzazione dell'output dipende dal cammino critico massimo.

Il cammino critico non è altro che la somma degli stadi del circuito.

Inoltre abbiamo due forme delle porte ad ingresso multiplo viste precedentemente che differiscono per numero di stadi:



Funzione di parità

Prende in input una serie di n bit (0 o 1) e aggiunge un bit in uscita il quale varrà 0 se il numero di 1 nella serie è pari, mentre varrà 1 se il numero di 1 nella serie è dispari (rendendo poi pari il numero di 1 nella serie).

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
1	0	0	1
0	1	1	0
1	1	0	0
1	0	1	0
1	1	1	1

Quindi con la prima forma canonica notiamo che vale uno nei 4 casi:
 $\overline{abc} + \overline{ab\bar{c}} + \overline{a\bar{b}c} + abc$

Se noi abbiamo un numero I da analizzare, possiamo dividerlo in due ed analizzarlo come I_1 e I_2 :

I_1	I_2	I	
0	0	0	pari + pari = pari
0	1	1	pari + dispari = dispari
1	0	1	dispari + pari = dispari
1	1	0	dispari + dispari = pari

La tabella di verità somiglia a quella della funzione XOR o exclusive OR.

Laboratorio del 2 Novembre 2021

Forme canoniche e cammino critico

- Prima forma canonica (SOP: Sum Of Products)

$$\sum_{i=1}^Q m_i, \quad Q \leq 2^n$$

m_i mintermine

Q numero di mintermini

- Seconda forma canonica (POS: Product Of Sums)

$$\prod_{i=1}^W M_i, \quad W \leq 2^n$$

M_i maxtermine

W numero di maxtermini

- Cammino critico: massimo numero di porte (escluso l'inverter) da attraversare da un qualsiasi ingresso ad una qualsiasi uscita.

Esercizio 2

Sia data la seguente espressione logica:

$$X = a(a + \bar{b})(b + c) + \bar{b}d$$

- 1) Si derivi la tabella di verità

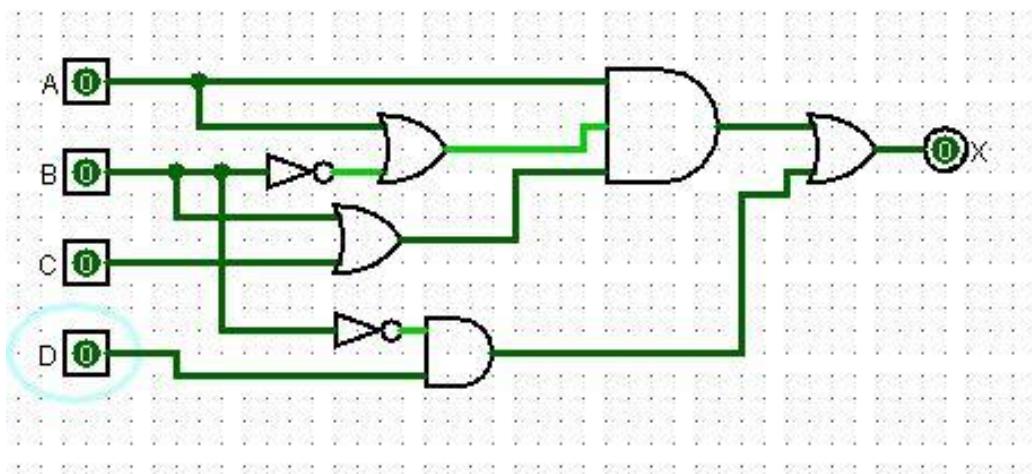
a	b	c	d	X
0	0	0	0	0

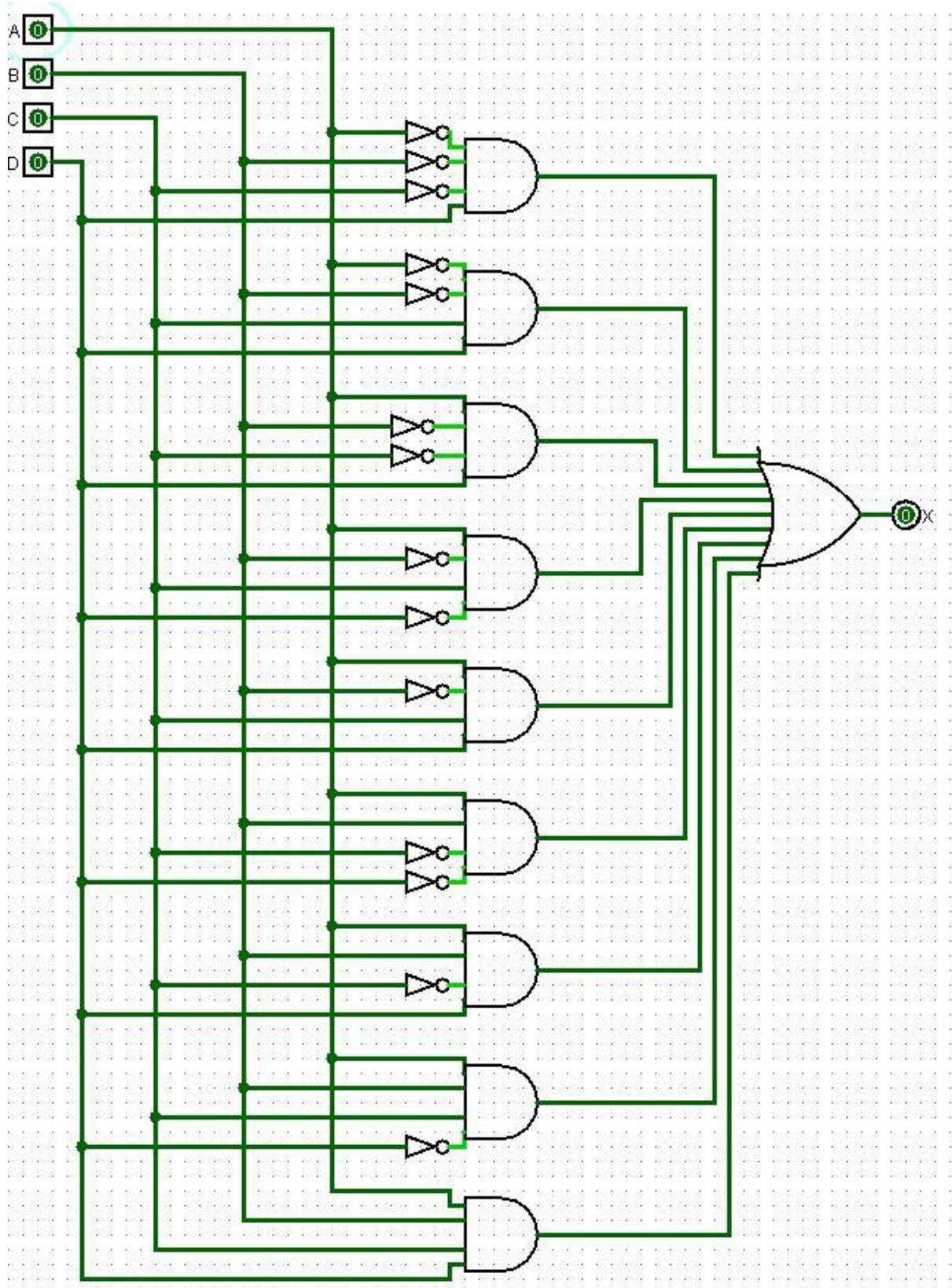
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

2) Si derivi la SOP

$$SOP = \overline{abcd} + \overline{ab}cd + a\overline{bc}d + \overline{abc}\overline{d} + ab\overline{cd} + a\overline{bcd} + ab\overline{c}d + abc\overline{d}$$

3) Si implementi su logisim il circuito associato alla formula originale ed il circuito associato alla SOP e li si confrontino

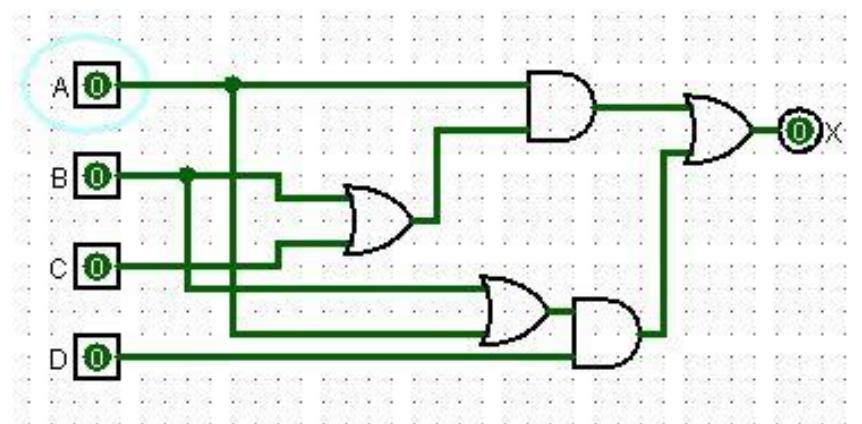




- 4) Si proceda poi alla semplificazione algebrica della SOP, si implementi il circuito corrispondente e lo si confronti con gli altri due circuiti implementati

$$SOP = \overline{abcd} + \overline{ab}cd + a\overline{bc}d + \overline{abc}\overline{d} + a\overline{bcd} + ab\overline{cd} + a\overline{bc}\overline{d} + abc\overline{d} + abcd$$

$$\begin{aligned}
&= (a + \bar{a}) \bar{b} \bar{c} d + (a + \bar{a}) \bar{b} c \bar{d} + a c \bar{d} (b + \bar{b}) + a b \bar{c} (d + \bar{d}) + a b c d \\
&= \bar{b} \bar{c} d + \bar{b} c \bar{d} + a c \bar{d} + a b \bar{c} + a b c d \\
&= \bar{b} d (c + \bar{c}) + a c \bar{d} + a b (\bar{c} + c d) \\
&= \bar{b} d + a c \bar{d} + a b (\bar{c} + c d) \\
&= \bar{b} d + a c \bar{d} + a b (\bar{c} + d) \\
&= \bar{b} d + a c \bar{d} + a b \bar{c} + a b d \\
&= d (\bar{b} + a b) + a c \bar{d} + a b \bar{c} \\
&= d (\bar{b} + a) + a c \bar{d} + a b \bar{c} \\
&= d \bar{b} + a d + a c \bar{d} + a b \bar{c} \\
&= a (d + c \bar{d}) + \bar{b} d + a b \bar{c} \\
&= a d + a c + \bar{b} d + a b \bar{c} \\
&= a (c + b \bar{c}) + a d + \bar{b} d \\
&= a c + a b + a d + \bar{b} d \\
&= a (c + b) + d (a + \bar{b})
\end{aligned}$$



Mintermini adiacenti

- Si parte dall'espressione in forma canonica SOP
- Due mintermini sono detti adiacenti se differiscono su una sola variabile
- Un'espressione con mintermini adiacenti può essere minimizzata nel seguente modo: $\bar{b} \bar{c} d + \bar{b} c \bar{d} = \bar{b} d$
- Ritroviamo l'esigenza di trovare un metodo che ci permetta di individuare velocemente i mintermini adiacenti

Mappe di Karnaugh

Prima di creare delle vere e proprie mappe di Karnaugh dobbiamo mettere in ordine la nostra serie di input in modo che differisca sempre di uno, per poi andare a convertirlo in mappe bidimensionali.

Mappa normale:

a	b	c	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Mappa di Karnaugh

$bc \setminus a$	0	1
00	0	0
01	1	0
11	0	0
10	1	1

Esse sono tabelle a due dimensioni che rappresentano la tavola di verità in un modo diverso.

Hanno 2^n quadrati dove n è il numero di variabili.
Ogni quadrato contiene il valore di un mintermine.

Lezione del 4 Novembre 2021

Funzione di maggioranza

La funzione vale 1 se la maggioranza degli input equivale a 1, altrimenti vale 0. Prendiamo ad esempio una funzione di maggioranza su 3 bit:

a	b	c	y
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

La funzione secondo la prima forma canonica sarà quindi composta da: $\bar{a}\bar{b}c + \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + ab\bar{c}$

Attraverso le semplificazioni arriviamo a $y = ab + ac + bc$

Semplificazione dei circuiti

Per ottenere una semplificazione dei circuiti solitamente lavoriamo sull'espressione booleana.

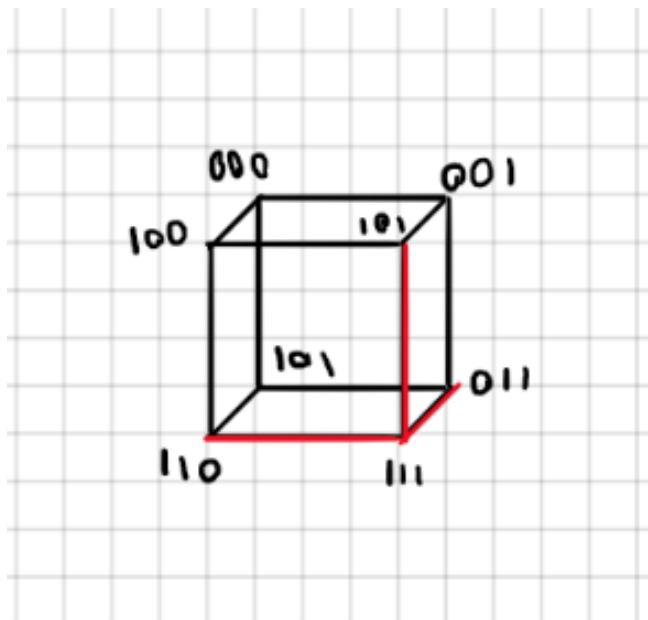
Tuttavia esiste un altro metodo, cioè utilizzando le mappe di Karnaugh, metodo meccanico e grafico basato sull'identificazione degli implicanti.

Mappa di Karnaugh

Essa garantisce sempre l'espressione minima.

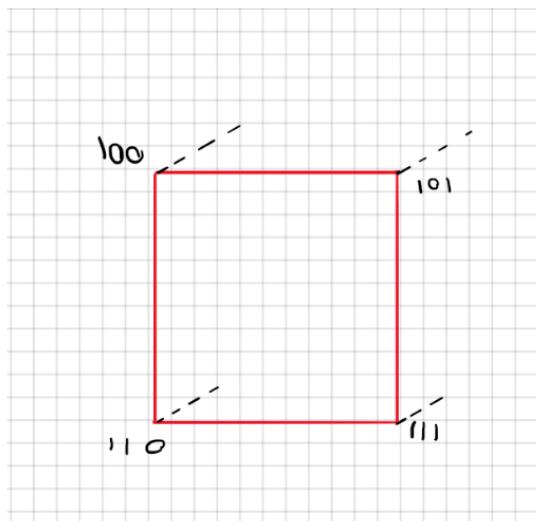
In una funzione di n variabili ogni combinazione di input è rappresentata da una stringa di n caratteri.

Nel sistema binario abbiamo la codifica di un sistema grafico per rappresentare le stringhe binarie su n dimensioni.



Ciascun vertice corrisponde ad una configurazione di input, mentre ad ogni lato corrisponde una variabile che cambia in quella configurazione.

Due mintermini in questa figura sono i vertici in cui la funzione vale 1, prendendo ad esempio $a\bar{b}\bar{c} + a\bar{b}c$ esso diventerà $a\bar{b}(\bar{c} + c)$ ed infine $a\bar{b}$. Dati due mintermini sullo stesso lato possiamo ottenere una semplificazione calcolando un implicante (il numero di implicanti dipende dal numero di lati con due mintermini adiacenti).



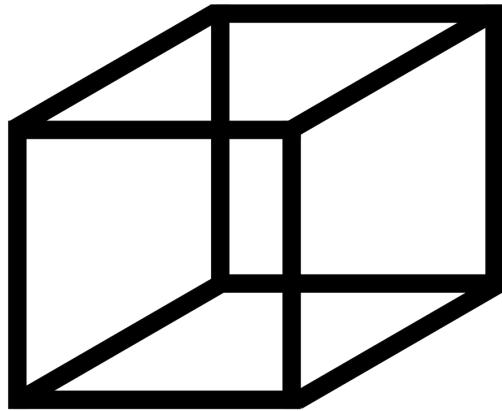
In questo caso abbiamo 4 mintermini connessi:

$$a\bar{b}\bar{c} + a\bar{b}c + a\bar{b}\bar{c} + abc = a(\bar{b}\bar{c} + \bar{b}c + b\bar{c} + bc) = a(\bar{b}(\bar{c} + c) + b(\bar{c} + c)) = a(\bar{b} + b) = a$$

Dalla rappresentazione grafica si possono ottenere mappe piene a due dimensioni:



$b \setminus a$	0	1
0		
1		

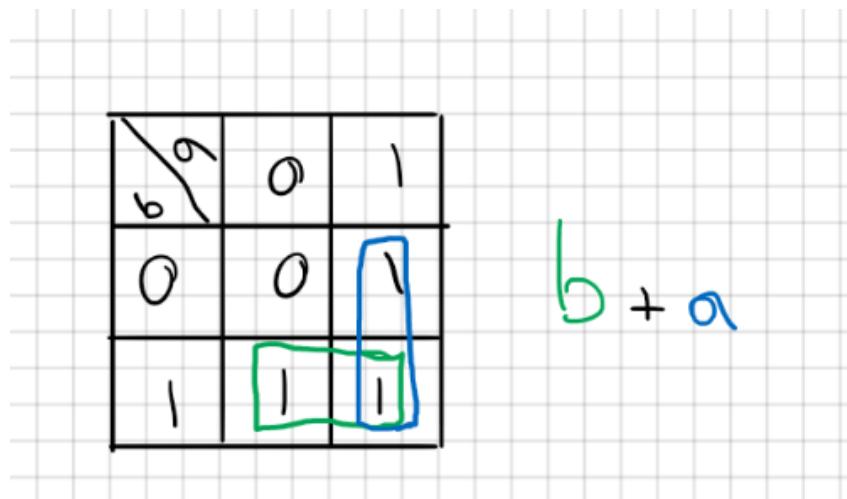


$c \setminus ab$	00	01	11	10
0				
1				

Lungo ogni direzione della tabella seguiamo la codifica di Gray (cambiando di 1 bit per volta).

Riempiamo quindi la tabella con i valori della funzione.

La semplificazione avrà luogo dove si formeranno dei rettangoli che racchiudono tutti gli 1, massimali e con area uguale ad una potenza di 2.



La rappresentazione piana è ciclica (wraparound world).

Blocchi funzionali

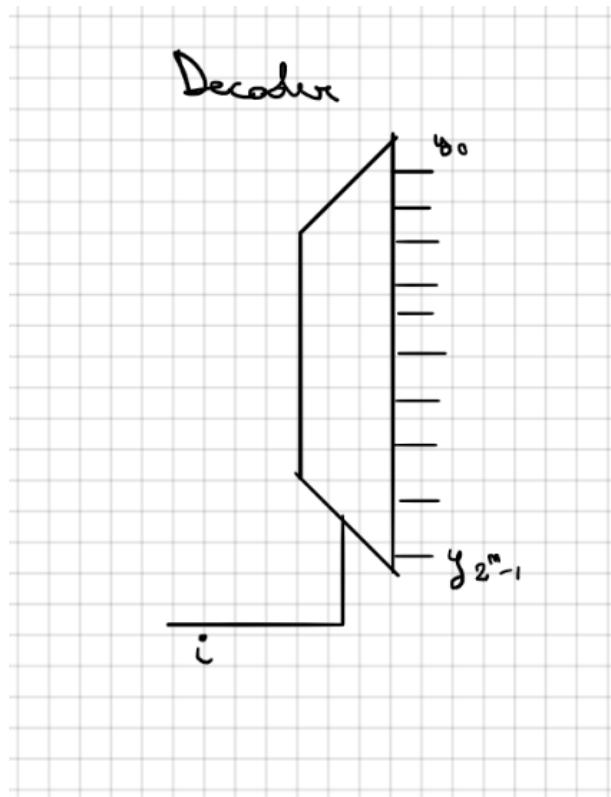
Essi sono sotto-circuiti naturali che svolgono funzioni molto comuni. Servono per modulizzare i sottocicli e costruire elementi di una libreria.

Decoder

Ha n ingressi e 2^n uscite. Gli n bit di ingresso li possiamo interpretare come un numero compreso tra $[0, 2^n - 1]$.

Il numero di configurazioni dei segnali in input equivale proprio al numero di uscite, ovvero 2^n .

Se il valore in ingresso sarà uguale a i allora l' i -esima porta di uscita varrà 1 mentre le altre 0.



Interpretazione

Conversione di un valore nel suo “one-hot-coding”, cioè valore di uscita singolo.

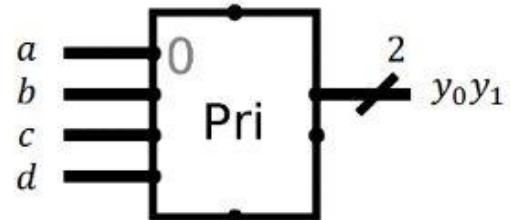
Com'è fatto un decoder:

Lezione dell'8 Novembre 2021

Encoder

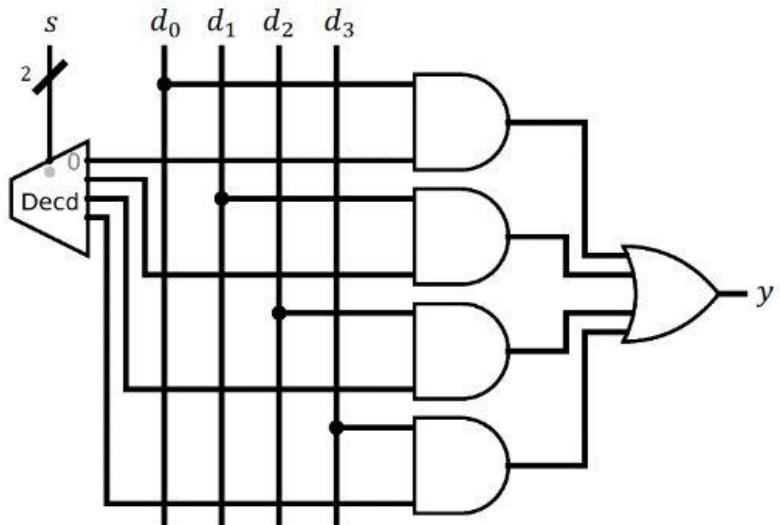
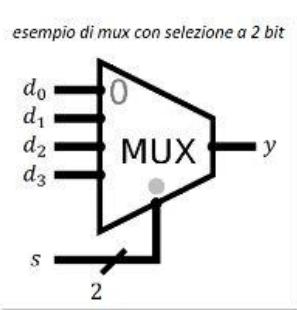
Circuito con 2^n ingressi ed n uscite. Tra i 2^n input solo uno di essi deve essere 1, in questo caso l'output corrisponderà al valore binario dell'input, se l'input ha più di un 1 allora verrà definito un "don't care".

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



Multiplier

Anche chiamato MUX, esso è un circuito con $2^n + n$ ingressi e un bit di uscita, i 2^n ingressi vengono messi in un AND con gli output di un decoder di n valori, in modo che uno solo di essi valga uno e lasci uscire uno solo dei 2^n bit in output.

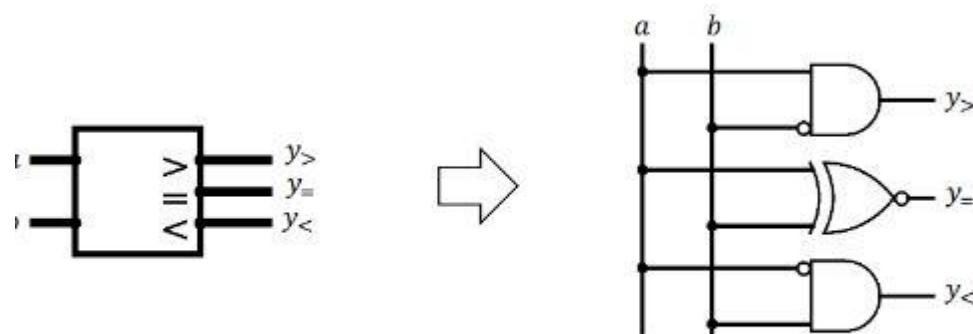


Comparatore

Circuito che riceve in ingresso due numeri binari su n bit e da in uscita uno dei tre possibili output:

- maggioranza stretta;
- uguaglianza;
- minoranza stretta.

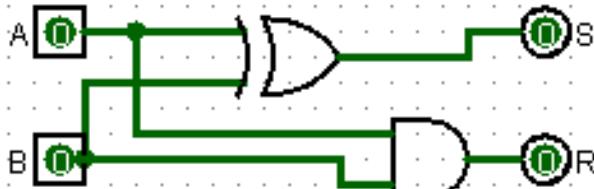
Per un comparatore con numeri su più di 2 bit bisogna combinare più operatori da 2 bit, che comparano le varie parti del numero binario.



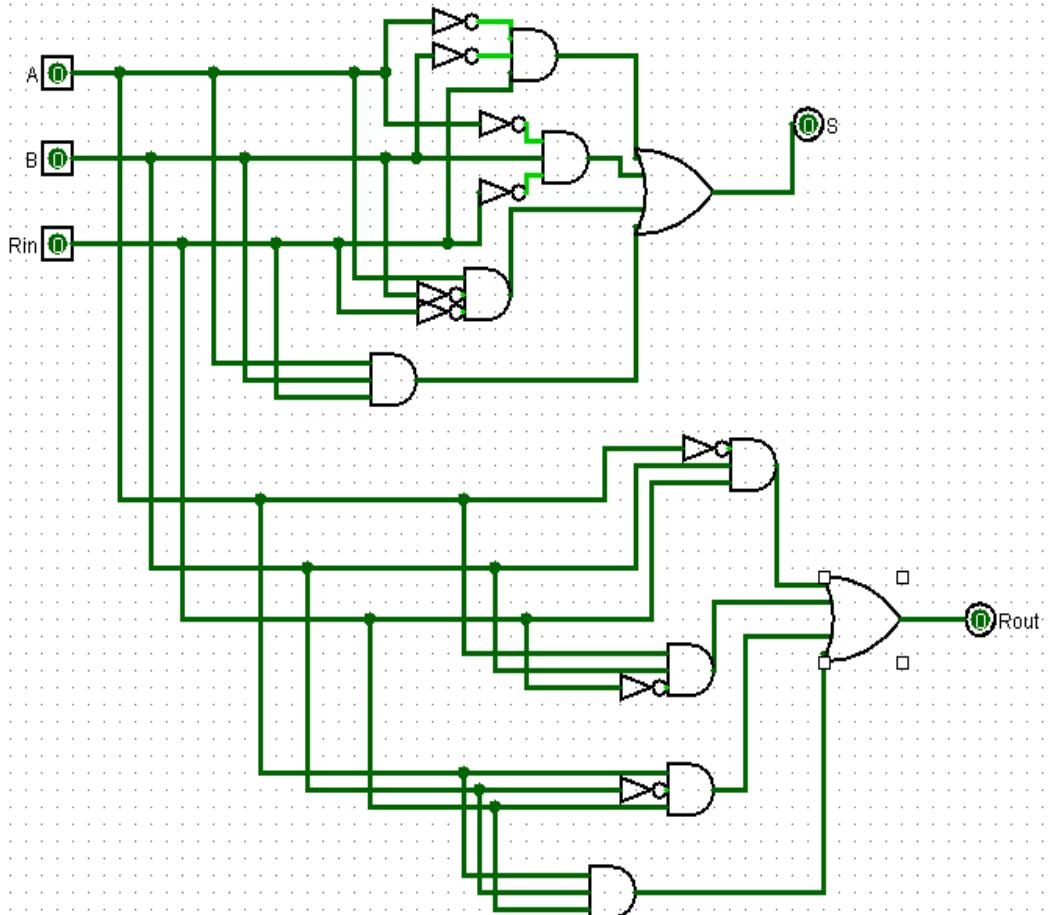
Circuiti aritmetici

Essi sono circuiti che ci permettono di svolgere operazioni aritmetiche:

- Half Adder, cioè la somma di numeri binari a 1 bit, dato in uscita il risultato e l'eventuale riporto;



- Full Adder, cioè la somma di due numeri binari ad un bit, che consente di mettere in ingresso un riporto;



- Sommatoria a Propagazione di riporto su n bit, cioè una sommatoria di riporti formato da una catena che inizia con un Half Adder e continua con Full Adders, collegati da riporti in uscita che vanno in entrata in al Full Adder successivo.

Problema: i blocchi non operano in parallelo, quindi l' $n+1$ -esimo blocco deve aspettare il riporto dell' n -esimo blocco.

- Anticipazione di riporto, cioè il "carry lookahead", attraverso il quale creo un circuito ad hoc che calcoli i riporti immediatamente per far operare i sommatori in parallelo senza aspettarsi l'uno con l'altro.

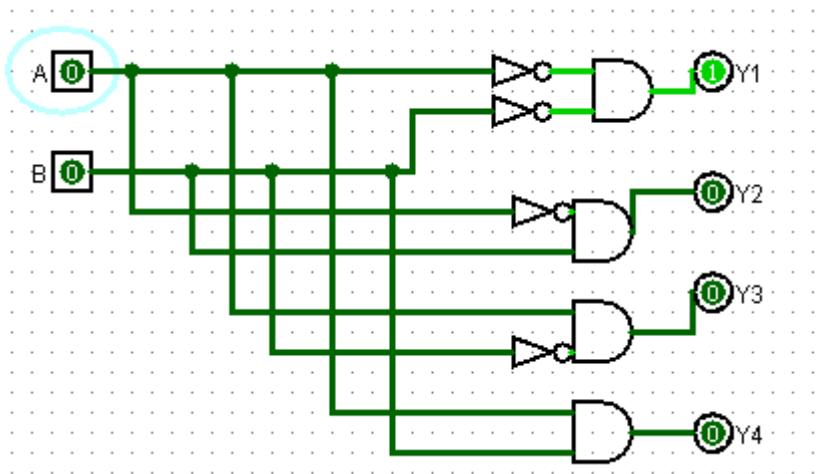
Laboratorio del 9 Novembre 2021

Esercizio 1

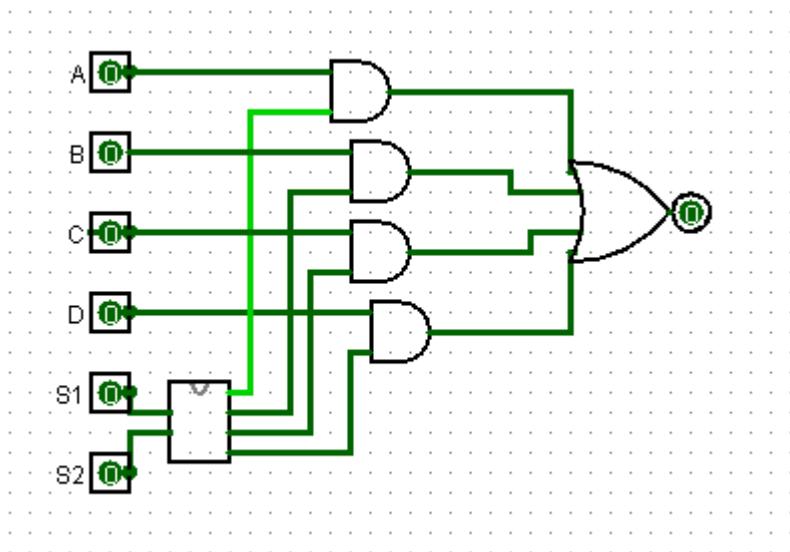
- Si progetti e si implementi su Logisim il circuito a 2 bit;
Suggerimento: il decodificatore riceve in ingresso una sequenza di 2 bit e attiva in uscita una delle 4 linee, in particolare quella identificata dalla sequenza di bit in ingresso

a	b	y_0	y_1	y_2	y_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$$y_0 = \overline{ab} \quad y_1 = \overline{a}\overline{b} \quad y_2 = a\overline{b} \quad y_3 = ab$$



- Si utilizzi il decodificatore così creato per implementare in Logisim un multiplexer a 4 vie; Suggerimento: il multiplexer seleziona una delle quattro linee in ingresso e la lascia passare in uscita

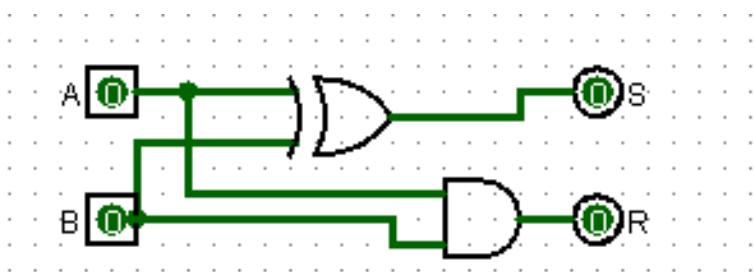


Esercizio 2

- Si scriva la tabella di verità per un addizionatore ad 1 bit senza riporto in ingresso

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

$$s = \bar{a}\bar{b} + \bar{a}\bar{b} \quad r = ab$$

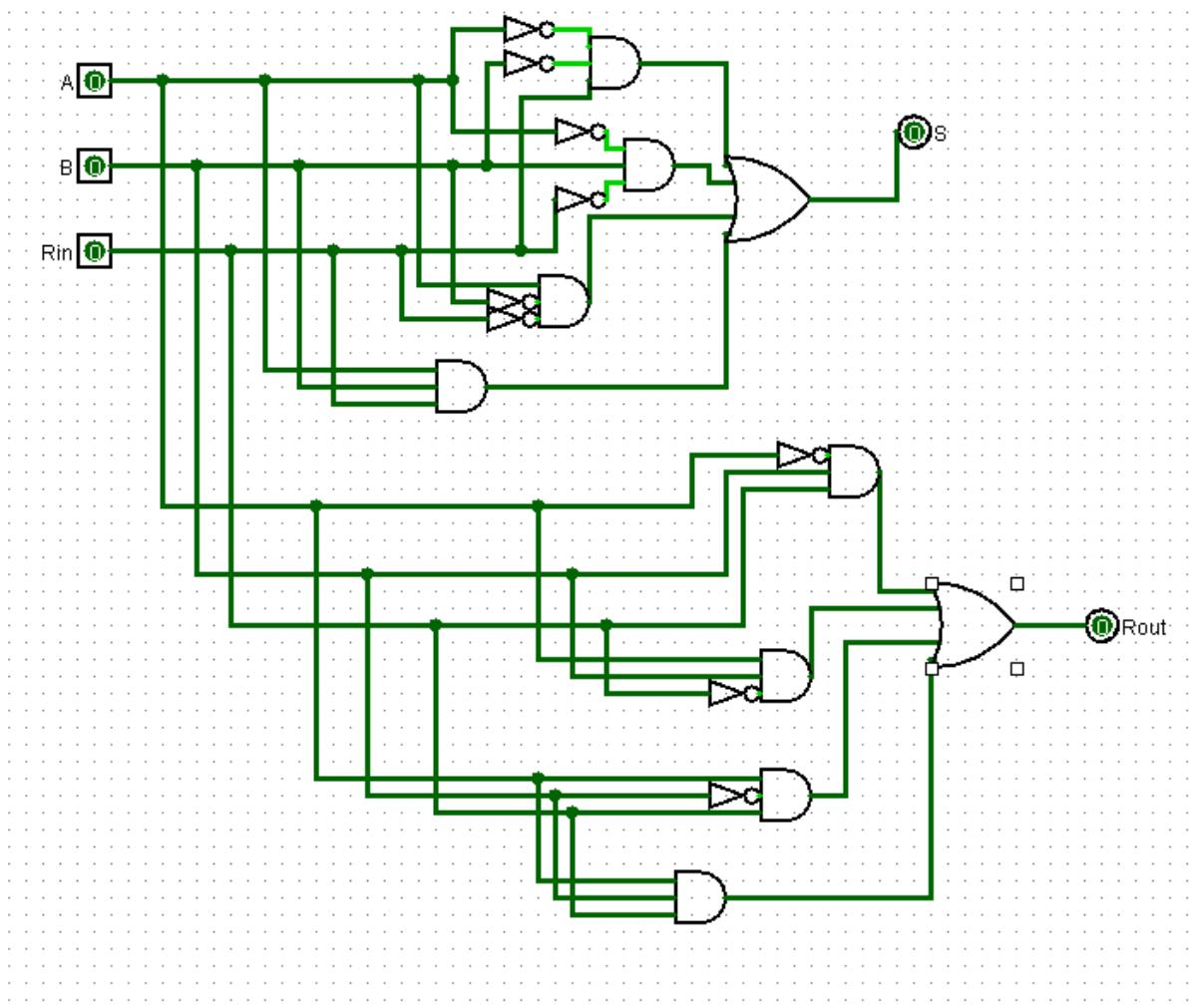


Esercizio 3

- Si scriva la tabella di verità per un addizionatore ad 1 bit con riporto in ingresso (Full Adder)
- Se ne dia un'implementazione su Logisim basata su SOP e si solvi il circuito
- Si fornisca poi una versione semplificata utilizzando il circuito di Half Adder

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

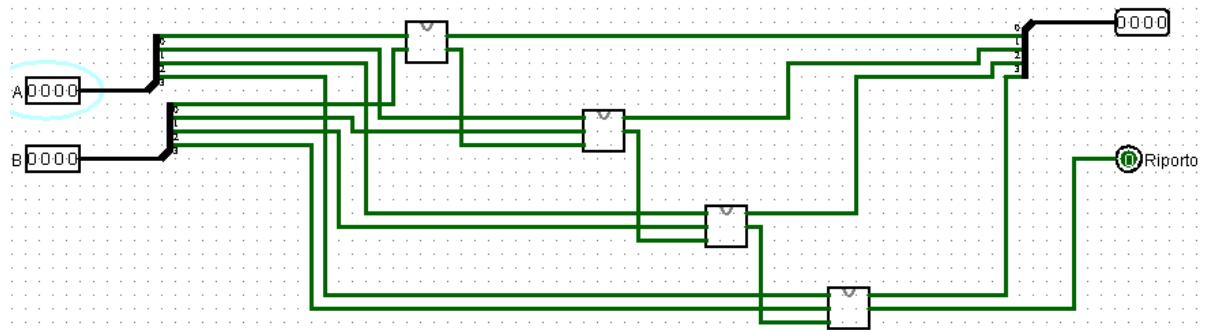
$$s = \overline{abr}_{in} + \overline{abr}_{in} + abr_{in} + abr_{in}r_{out} = \overline{abr}_{in} + abr_{in} + abr_{in} + abr_{in}$$



Esercizio 4

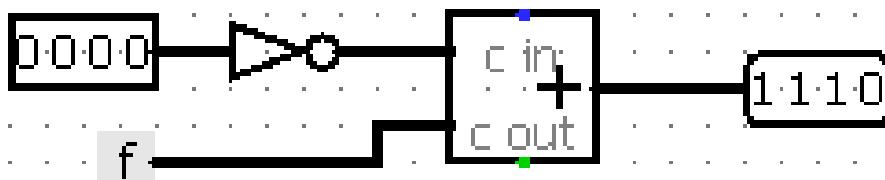
- Si utilizzino il circuito Half Adder precedentemente sviluppato e il Full Adder per implementare l'azione su un determinato numero n di bit

L'Half Adder ha come cammino critico 1, mentre il full adder ha come cammino critico 2, mentre il cammino critico totale è di 7 [1+2(n-1)].



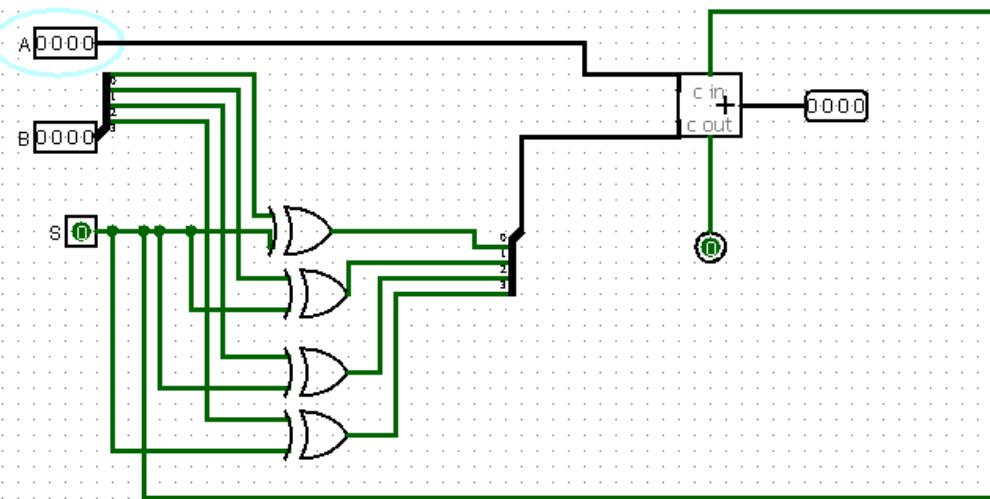
Esercizio 5

- Si realizzi un circuito che dato un numero X, attraverso il complemento a 2 restituisca -X



Esercizio 6

Si realizzi un circuito che operi la somma e la differenza di due numeri a e b in 4 bit utilizzando un bit di selezione s

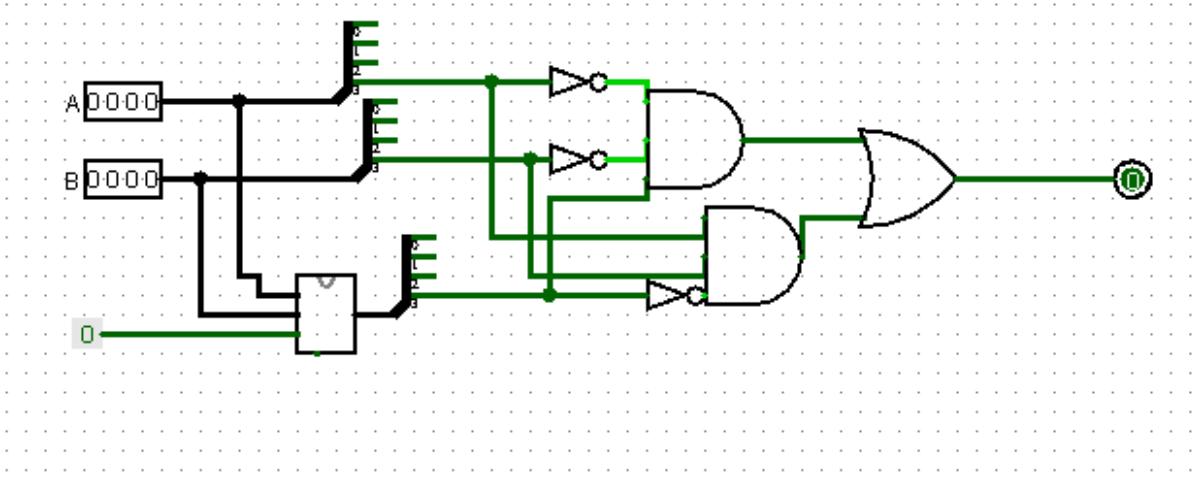


Esercizio 7

- Si modifichi il programma precedente in modo da poter controllare quando si verifica un overflow

a	b	s	o
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

$$o = \overline{abs} + abs$$

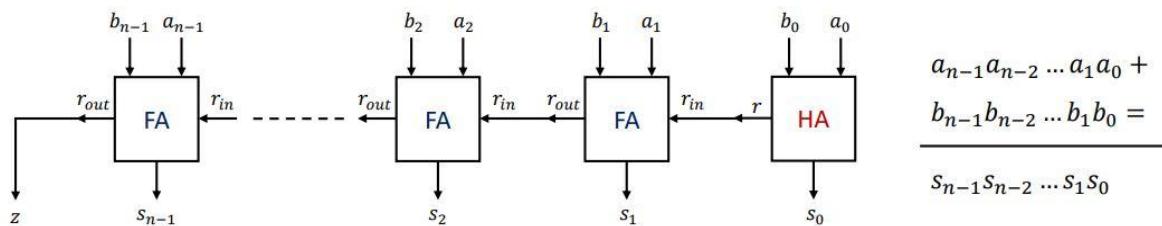


Lezione dell'11 Novembre 2021

Sommatore a propagazione di riporto su n bit

Per poter sommare numeri $a = a_{n-1}a_{n-2}\dots a_1a_0$ e $b = b_{n-1}b_{n-2}\dots b_1b_0$ su n bit posso collegare in serie i sommatori su 1 bit.

L'Half Adder somma i due bit meno significativi, calcolando il bit meno significativo della somma e propaga il riporto verso il Full Adder adiacente che somma i bit successivi.



L'ultimo riporto z se sommiamo due numeri naturali indica l'overflow, se sommiamo due numeri in complemento a 2 va ignorato.

Calcolo del cammino critico:

La propagazione del riporto implica che l'ultimo blocco debba aspettare gli altri, quindi i blocchi non lavorano in parallelo

$$1 + 3 + 3 + 3 + \dots = 1 + 3(n - 1) \approx 3n$$

Anticipazione di riporto (carry-lookahead)

Il sommatore a propagazione di riporto ha un cammino critico elevato, quindi per attraversare tutto il percorso viene rallentata l'esecuzione.

Idea: invece di calcolare i riporti provo a scrivere un'espressione diretta in modo da anticipare il suo calcolo con un sottocircuito opposto.

- Ricordiamo che l' i -esimo Full Adder ha come espressione
- $r_{i+1} = a_i b_i + r_{i-1} (a_{i-1} b_{i-1})$
- Rinomino i termini dell'espressione, $a_i b_i$ è il termine di generazione G_i e in ogni Full Adder è subito pronto mentre $(a_i + b_i)$ è il termine di propagazione P_i il quale deve aspettare la propagazione di r_i

- Per ogni riporto derivo la sua espressione, in generale abbiamo

$$r_n = G_{n-1} + P_{n-1}G_{n-2} + P_{n-1}P_{n-2}G_{n-3} + \dots + P_{n-1}P_{n-2}\dots P_0 r_0$$

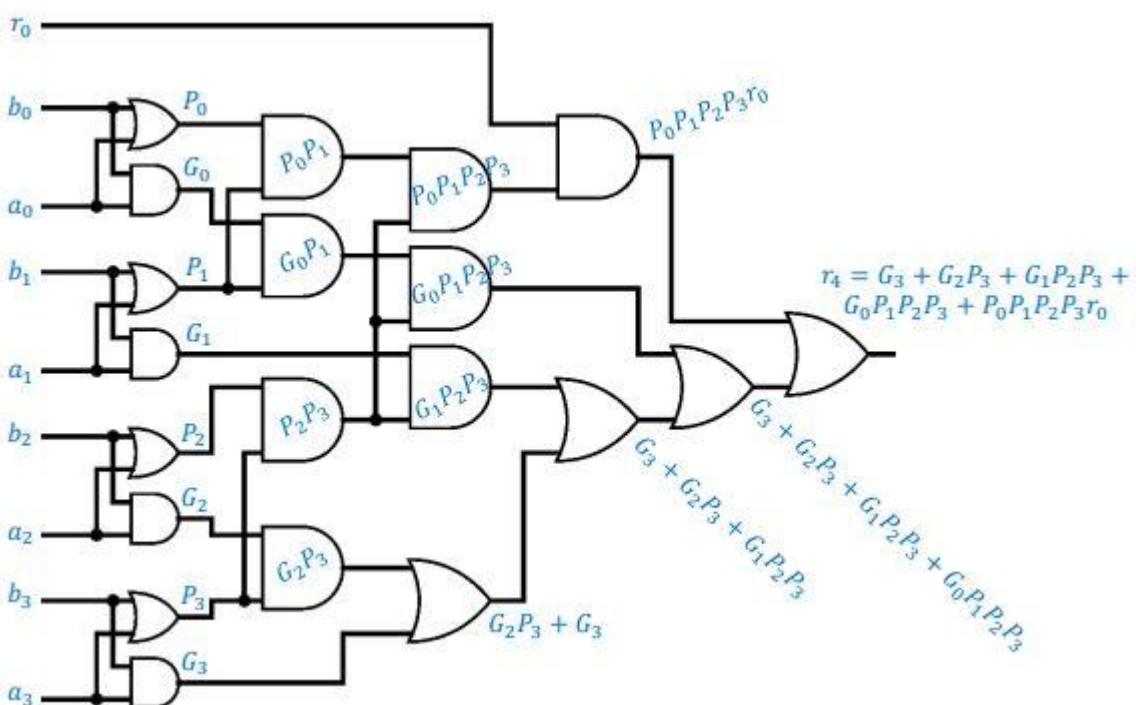
Anticipazione di riporto

Consideriamo un sommatore a propagazione di riporto con $n = 4$, se lo implementiamo con lo schema precedente, e quindi solo con Full Adder, otteniamo un cammino critico pari a 12.

Se calcoliamo ogni riporto con un circuito dedicato che implementa la sua espressione diretta otteniamo che:

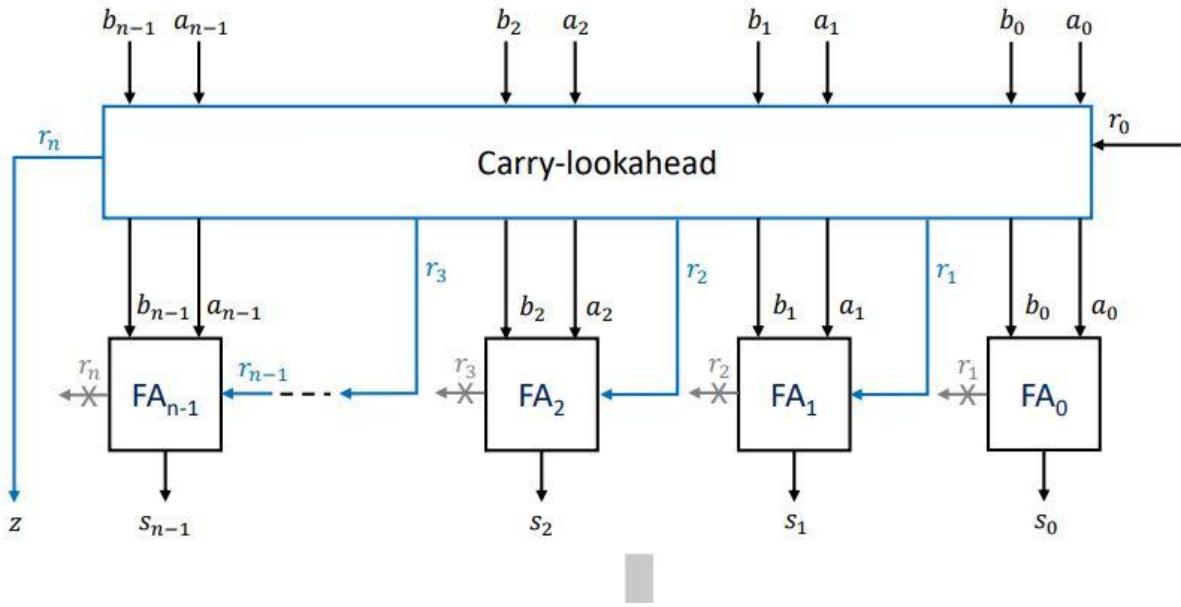
- $r_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0r_0$

Il cammino critico di questo percorso è 6, se calcoliamo il riporto di questa rete anticipiamo il suo arrivo all'uscita e abbassiamo il cammino critico di tutto il circuito.



Quindi infine come cambia la struttura del sommatore?

- Abbiamo spezzato la catena di propagazione di riporti;
- Ora ciascun Full Adder può lavorare in parallelo grazie all'unità di look-ahead;
- Il cammino critico è sempre dato dall'ultimo riporto, ma questa volta è più corto.



Sottrazione

Il modulo per l'addizione che abbiamo costruito può essere esteso per gestire anche le sottrazioni.

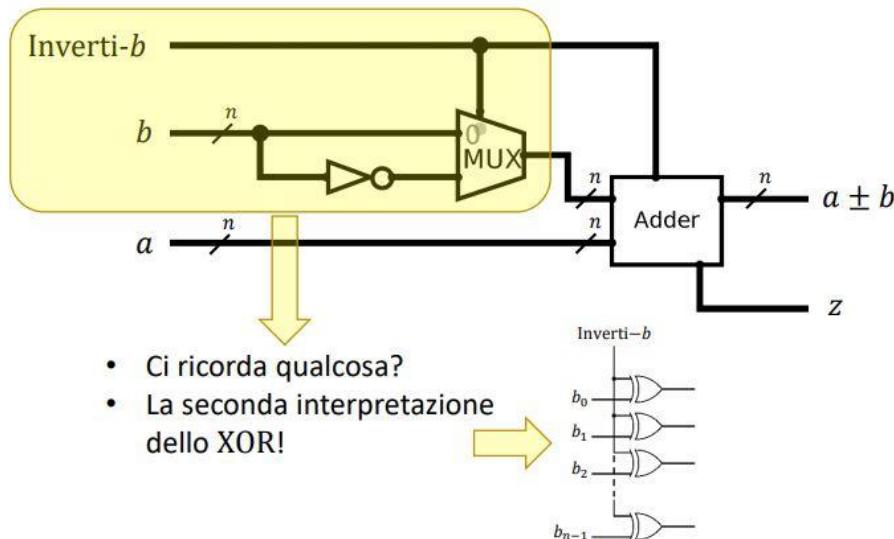
Metodo, la sottrazione $a - b$ si interpreta e quindi si esegue come una somma binaria tra a e il complemento a 2 di b .

Fare il complemento a 2 di b significa complementare a 1 e aggiungere 1. Quindi abbiamo un bit chiamato "inverti b ", il quale se viene posto a 1 produce 2 effetti:

- L'Adder riceve il complemento a 1 di b ;
- L'Adder riceve un riporto di ingresso di 1.

In questo caso quindi l'Adder riceve: $a + \bar{b} + 1 = a + (\bar{b} + 1) = a - b$

Se "inverti b " è posto a 1, z va scartato, altrimenti indica un overflow.



Estensione del segno e shift

Due operazioni comuni che ci possono tornare utili

Estensione del 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 il Modulo e segno fino a raggiungere il numero totale di bit desiderati. Nel caso in cui gli n bit iniziali rappresentino un naturale o un intero in Complemento a 2, 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 ($\gg k$) o a sinistra ($\ll k$). Se i bit rappresentano un numero naturale e non vengono cancellati 1, $\ll k$ equivale ad una moltiplicazione per 2^k .



Moltiplicatore

La moltiplicazione binaria si organizza in due fasi:

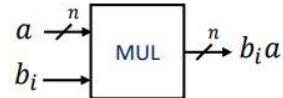
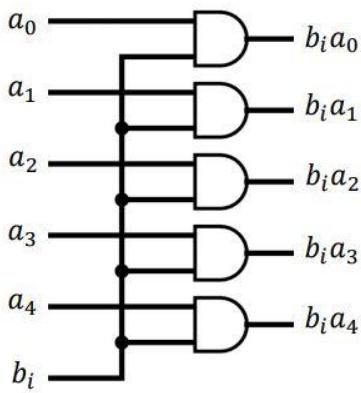
- 1) Calcolo dei prodotti parziali, AND a coppie di bit in posizione shiftata;
- 2) Somma bit a bit (considerando anche i riporti) dei prodotti parziali, usando dei Full Adder.

$ \begin{array}{r} 10111 \times a \\ 101 = b \\ \hline b_0 \times a \rightarrow 111 \\ b_1 \times a \rightarrow 00000 \\ b_2 \times a \rightarrow 10111 \\ \hline 1110011 \quad a \times b \end{array} $	<p style="text-align: center;">Prodotti parziali</p> $ \begin{array}{ccccccc} 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} $	<p style="text-align: center;">Somma</p>
---	--	--

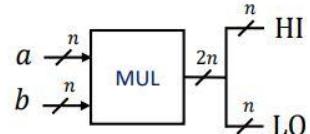
In generale il prodotto di 2 numeri su n bit, può dare un risultato su $2n$ bit, di norma si suddivide il risultato in due numeri separati con HI si indicano gli n bit della parte alta (indicano anche un overflow), mentre con LO gli n bit della parte bassa.

Moltiplicatore $1 \times n$

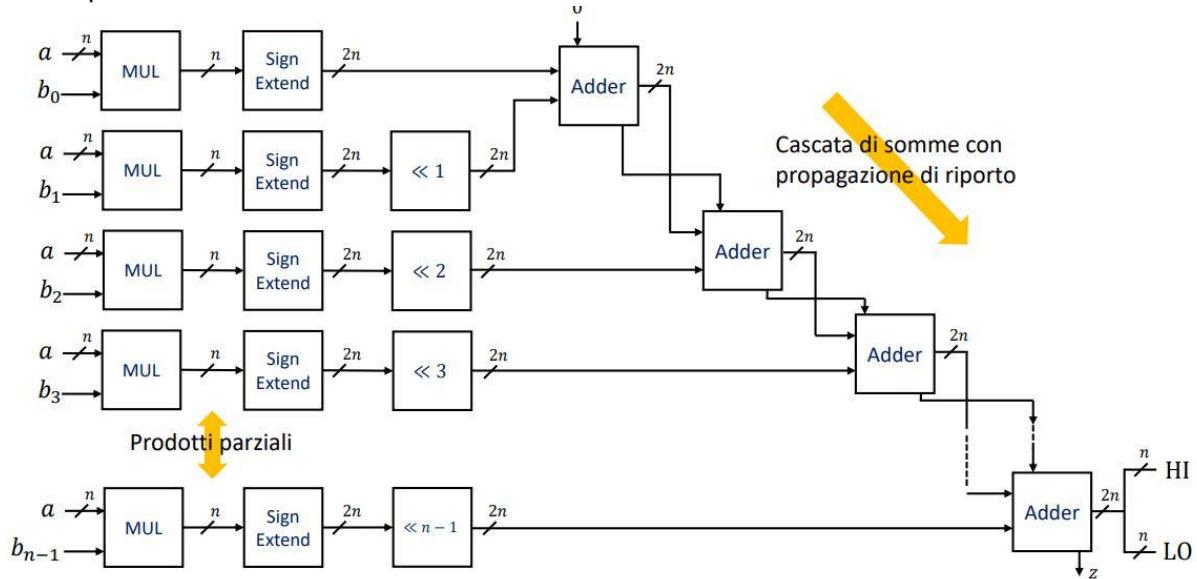
Consideriamo un moltiplicatore che prende in input un singolo bit e un operando su n bit, ci serve per calcolare ciascuna riga dei prodotti parziali (prendiamo come esempio $n = 5$).



Usando questo componente
posiamo costruire un
moltiplicatore $n \times n$



Moltiplicazione $n \times n$

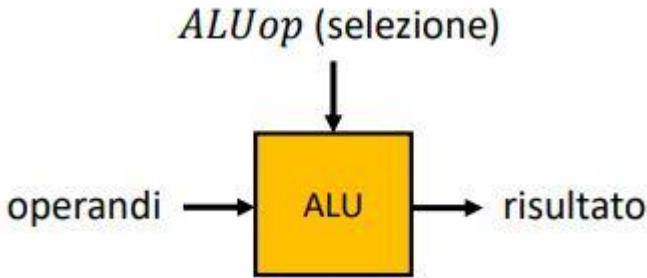


Lezione del 15 Novembre 2021

ALU: Unità Aritmetico-Logica

In una CPU, la ALU è la centrale hardware che svolge le operazioni di calcolo, queste possono essere di tipo aritmetico (ad esempio somme, sottrazioni, moltiplicazioni ecc...) oppure di tipo logico (ad esempio AND e OR). A livello astratto la ALU è un circuito combinatorio multifunzionale definito così:

- Input: gli operandi e un codice selezione, chiamato ALUop, che identifica la funzione f (l'operazione richiesta alla ALU);
- Output: il risultato dell'applicazione di f agli operandi.



Logica di progettazione:

- Modulare: una ALU da n bit (per operando 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.

ALU

La ALU è un componente centrale nella Architettura di Von Neumann, è l'elemento responsabile della fase di **execute**. Ogni operazione supportata è eseguita in hardware da un sotto-circuito combinatorio dedicato.

Le CPU moderne possono contenere diverse ALU per svolgere più operazioni in parallelo, ad esempio nelle GPU (CPU progettate specificatamente per calcoli finalizzati alla grafico).

Le ALU possono essere molto sofisticate, in questo caso vedremo la ALU MIPS, una versione base che supporta queste operazioni tra due operandi a e b :

- AND (logico);
- OR (logico);
- Somma (aritmetica);
- Sottrazione (aritmetica);
- Comparazione (logica);
- Test di uguaglianza allo zero (logico).

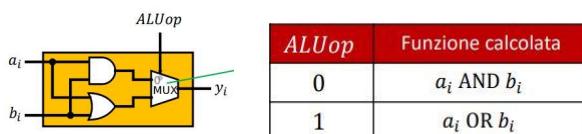
AND e OR

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 .

La selezione sarà implementata con un Multiplexer (in questo caso un MUX a 1 bit).

Parallelismo: la ALU calcola sempre sia AND che OR, ma solo una delle due viene posta in uscita attraverso la selezione.

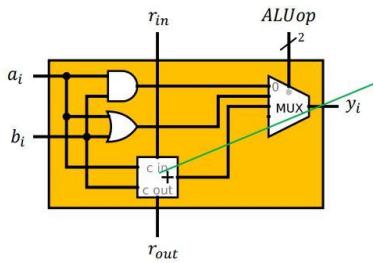
Aluop è, in questo caso, il singolo bit di selezione: se vale 0 in uscita avremo $a_i b_i$, se vale 1 avremo $a_i + b_i$.



Somma (ADD)

Estendiamo la ALU appena realizzata in modo che supporti la somma, sempre su 1 bit. Aggiungiamo un Full Adder, la ALU ora guadagna un nuovo input r_{in} e un nuovo output r_{out} . La ALU ora richiede una di tre diverse operazioni, quindi il MUX deve essere a 2 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

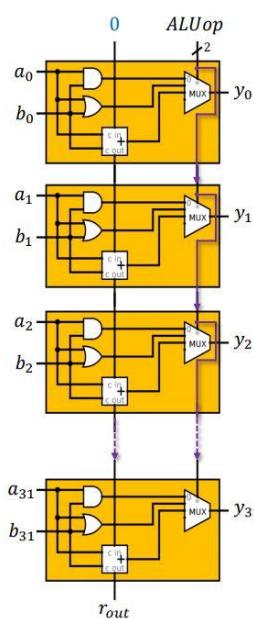
AND, OR e ADD su 32 bit

Collego 32 ALU da 1 bit utilizzando lo schema della propagazione dei riporti.

Tutte le ALU lavorano in parallelo.

Il primo riporto è sempre settato a 0.

ALUop è sempre su 2 bit ed è lo stesso in tutte le ALU.



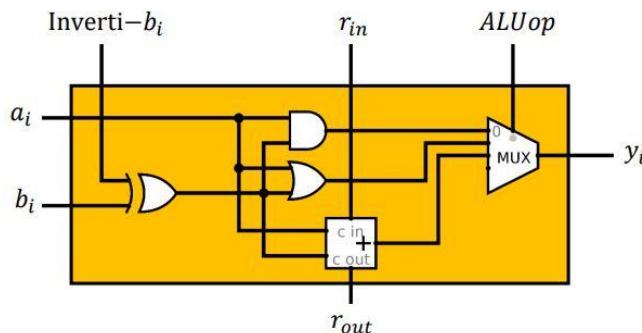
Stesse operazioni di prima ma ora sono su $n = 32$ bit.

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 ", se questo segnale viene posto a 1 l'operando b_i viene sostituito con \bar{b}_i (il suo complemento a 1).

Posso fare la sottrazione settando "Inverti b " e r_{in} entrambi a 1.



AND, OR, ADD e SUB su 32 bit

Collegiamo le 32 alu da 1 bit usando sempre lo stesso schema di propagazione dei riporti.

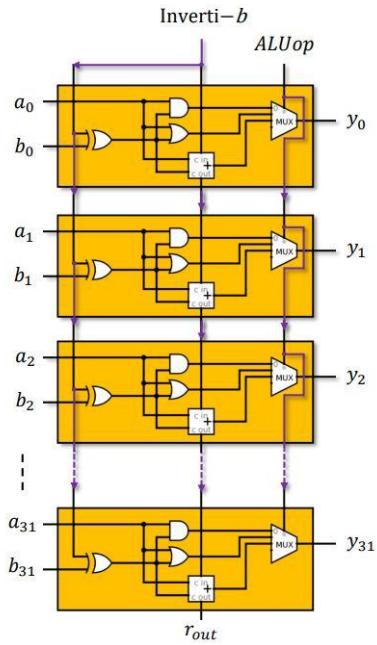
ALUop: esattamente come prima (2 bit, lo stesso per tutte le ALU da 1 bit).

"Inverti b " è lo stesso in tutte le ALU ed è anche collegato al primo r_{in} .

Settando "Inverti b " a 1 ottengo simultaneamente:

- 1) Tutti i bit di b vengono invertiti quindi dentro la ALU b viene subito trasformato in \bar{b} ;
- 2) Il primo riporto in ingresso è 1 quindi i sommatori svolgono $1 + a + \bar{b}$ che in complemento a 2 equivale a $a - b$.

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



Comparazione (SLT)

Attraverso la sottrazione possiamo ottenere anche il confronto di uguaglianza tra a e b : basta fare $a - b$ e controllare se il risultato è 0. Spoiler alert! Nella struttura finale della ALU aggiungeremo un'uscita di 1 bit detta "bit di zero", che vale 1 ogni qualvolta il risultato calcolato dalla ALU è nullo.

Per fare i test di disuguaglianza aggiungiamo alla nostra ALU l'operazione logica SLT (set less than):

$$\text{SLT}(a, b) = \begin{cases} 1, & a < b \\ 0, & a \geq b \end{cases}$$

Come si implementa all'interno della ALU?

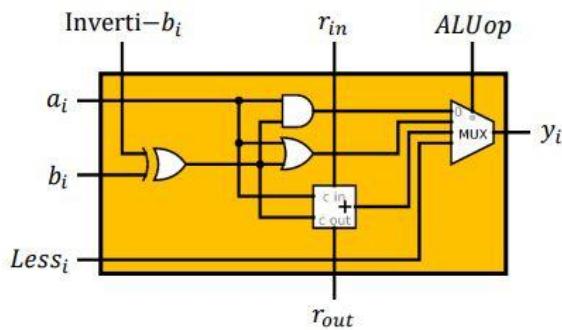
- 1) Calcoliamo $a - b$;
- 2) Se il risultato è < 0 allora in uscita mando $000\dots1$;
- 3) Altrimenti mando $000\dots0$;

Per verificare che il risultato della differenza sia negativo basta controllare il bit di segno in uscita dal Full Adder nella ALU in posizione $n - 1$ (l'ultima ALU) e cioè il bit di segno del risultato. I bit dalla posizione 1 a $n - 1$ dell'uscita sono sempre 0.

Quindi:

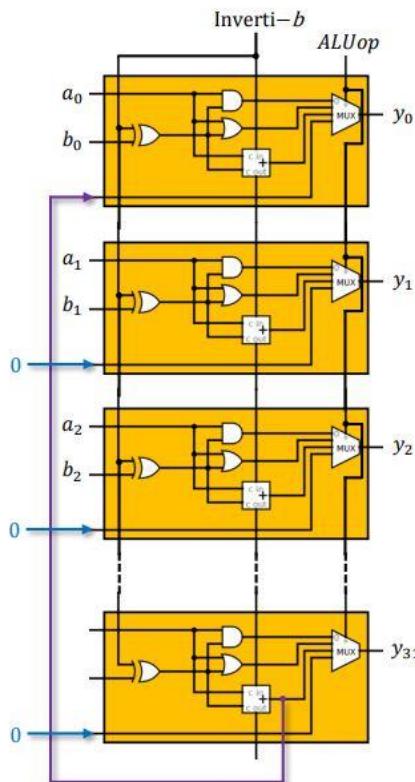
- 1) Setto le uscite y_1, y_2, \dots, y_{n-1} a 0;
- 2) Calcolo $a - b$;
- 3) Setto l'uscita y_0 a s_{n-1} (bit di somma in posizione $n - 1$).

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



Per fare la comparazione su n bit quindi:

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



Overflow

Come riconoscere la presenza di overflow?

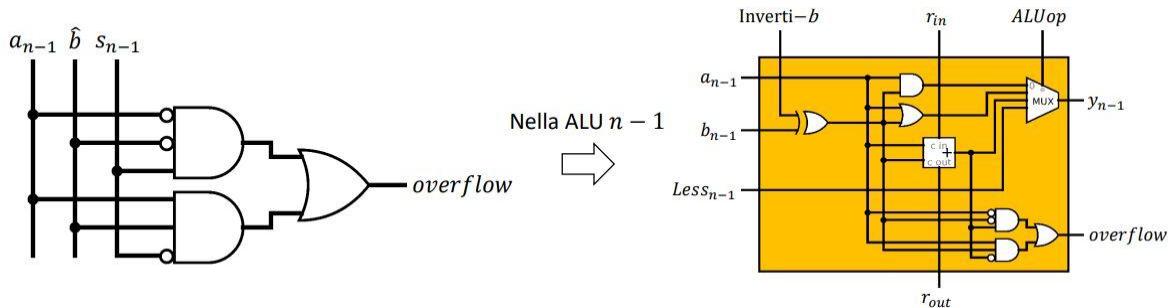
Richiamo: in complemento a 2 l'overflow si identifica in 2 modi:

- 1) Gli ultimi due riporti sono diversi ;
- 2) Sommando 2 numeri positivi ottengo un negativo oppure sommando 2 negativi ottengo un positivo.

Possiamo implementare il secondo metodo con un semplice circuito combinatorio che prende in input i bit di segno di a e b e del risultato della somma s : questi bit sono tutti nella ALU $n - 1$.

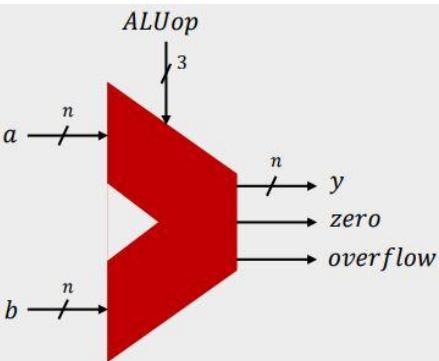
Attenzione! Nella ALU $n - 1$ il bit di segno di b non è b_{n-1} ma

$$b_{n-1} \text{ XOR } \text{"Inverti } b\text{"} = \hat{b}$$



Struttura finale

(Il bit di zero è stato creato con un NOR tra tutti i bit del risultato di y)



ALUop	Funzione calcolata
000	$a \text{ AND } b$ (bitwise)
001	$a \text{ OR } b$ (bitwise)
010	$\text{ADD}(a, b)$
110	$\text{SUB}(a, b)$
111	SLT

Per svolgere le operazioni della ALU con i numeri in Floating point abbiamo bisogno di una ALU-FP.

Laboratorio del 16 Novembre 2021

Esercizio 1

- 1) Scrivere la SOP e la POS delle seguenti tabelle di verità;
- 2) Implementare su logisim il circuito relativo alla SOP della prima tabella e della POS della seconda tabella e calcolare il cammino critico tra A e Y e tra C e Y per entrambi i circuiti;
- 3) Implementare i due circuiti utilizzando solo porte NAND per la tabella di sinistra e solo porte NOR per quella di destra. Calcolare poi il cammino critico tra A e Y e tra C e Y per entrambi i circuiti;
- 4) Costruire le mappe di Karnaugh delle due tabelle di verità e implementare i circuiti su logisim. Calcolare il cammino critico tra A e Y e tra C e Y per entrambi i circuiti.

a	b	c	y	a	b	c	y
0	0	0	1	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	0	1	1	1
1	0	0	1	1	0	0	0
1	0	1	1	1	0	1	1
1	1	0	0	1	1	0	1
1	1	1	0	1	1	1	1

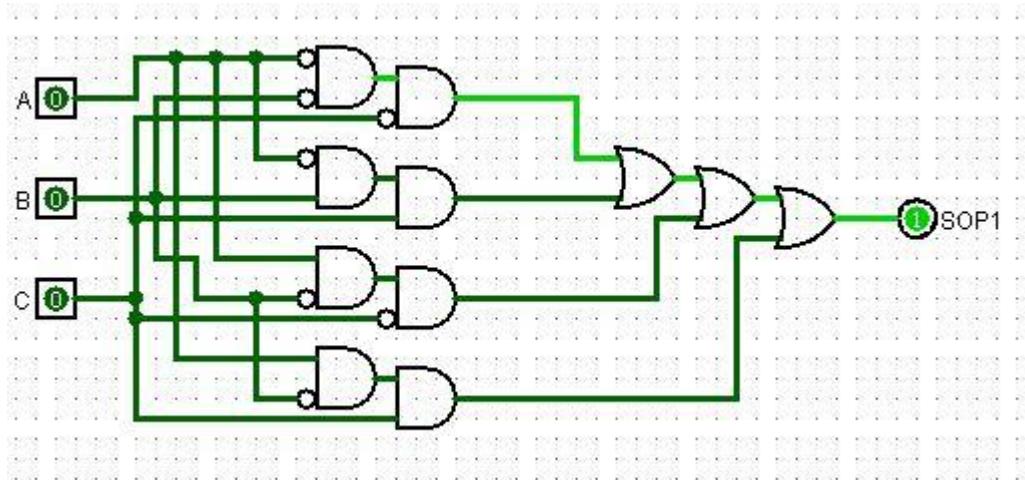
$$SOP_1 = \overline{abc} + \overline{ab}c + a\overline{bc} + ab\overline{c}$$

$$POS_1 = (a + b + \overline{c})(a + \overline{b} + c)(\overline{a} + \overline{b} + c)(\overline{a} + \overline{b} + \overline{c})$$

$$SOP_2 = \overline{a}bc + a\overline{b}c + ab\overline{c} + abc$$

$$POS_2 = (a + b + c)(a + b + \overline{c})(a + \overline{b} + c)(\overline{a} + b + c)$$

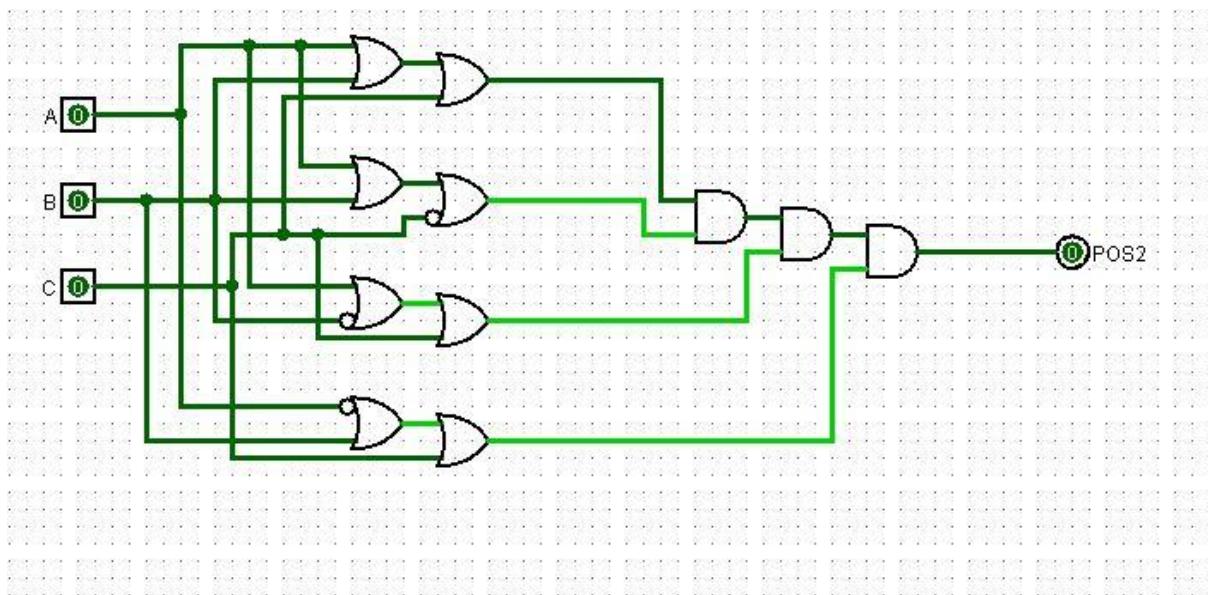
$$SOP_1:$$



Cammino critico tra A e Y: 5

Cammino critico tra C e Y: 4

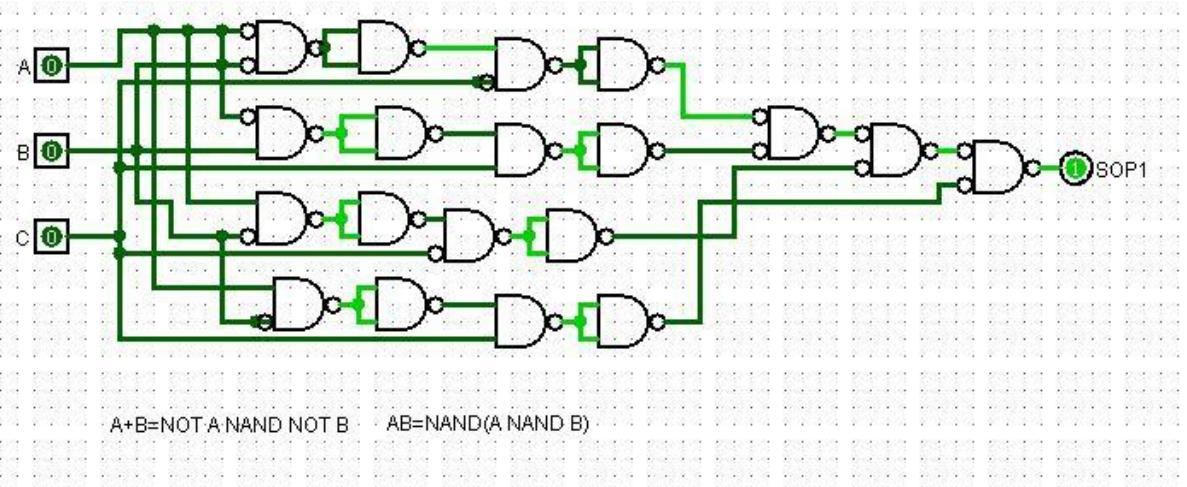
POS_1 :



Cammino critico tra A e Y: 5

Cammino critico tra C e Y: 4

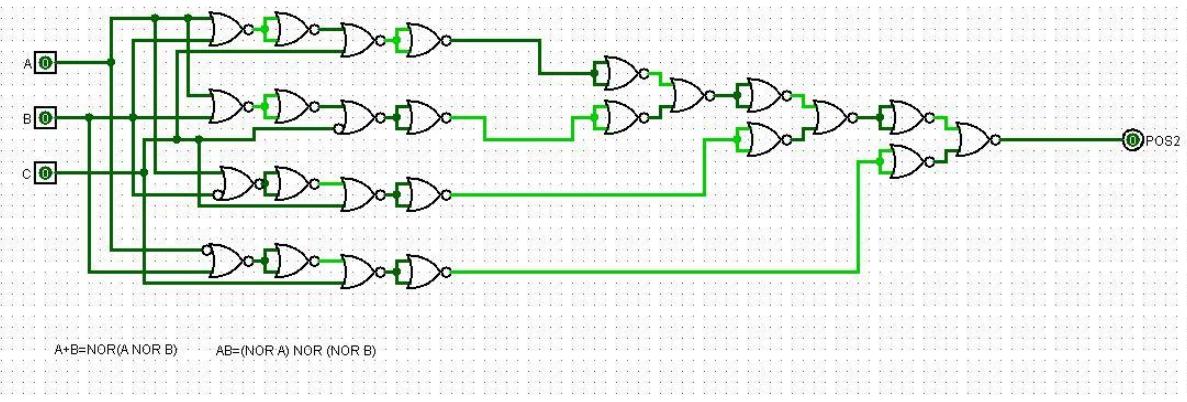
Prima SOP con solo NAND



Cammino critico tra A e Y: 7

Cammino critico tra C e Y: 5

Seconda POS con solo NOR



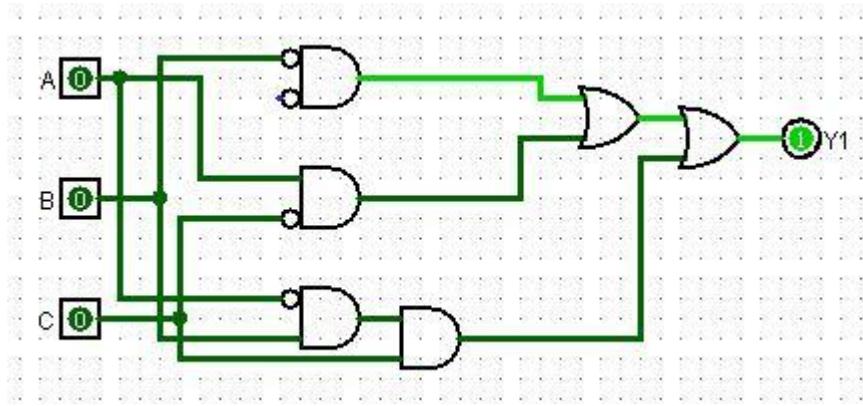
Cammino critico tra A e Y: 10

Cammino critico tra C e Y: 8

Mappe di Karnaugh:

$a \backslash bc$	00	01	11	10		$a \backslash bc$	00	01	11	10
0	1	0	1	0		0	0	0	1	0
1	1	1	0	0		1	0	1	1	1

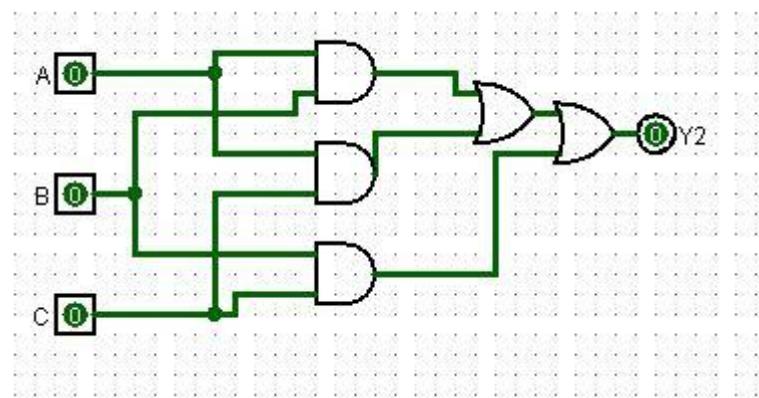
Formula semplificata della prima tabella: $\bar{bc} + a\bar{b} + \bar{a}bc$



Cammino critico tra A e Y: 3

Cammino critico tra C e Y: 3

Formula semplificata della seconda tabella: $ac + bc + ab$



Cammino critico tra A e Y: 3

Cammino critico tra C e Y: 3

Esercizio 2

- 1) Costruire la mappa di Karnaugh della seguente tabella di verità;
- 2) Implementare su Logisim il circuito corrispondente e determinare le combinazioni di input-output associate al cammino critico minore e maggiore.

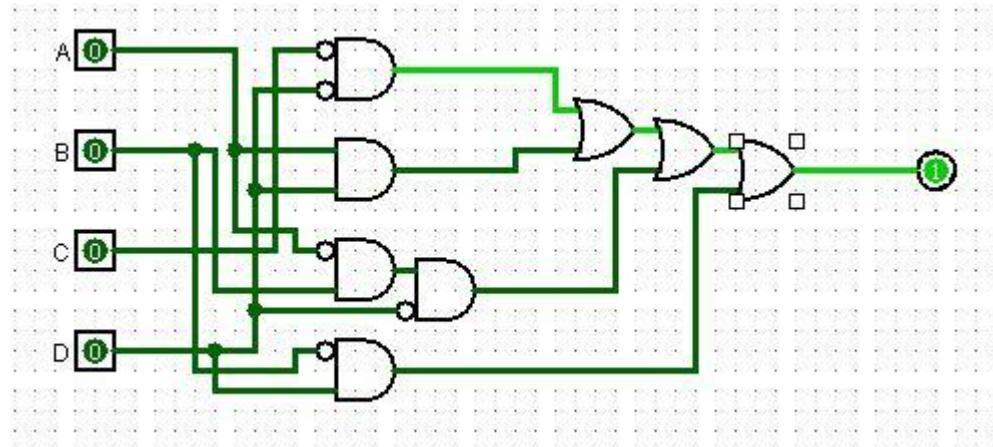
a	b	c	d	y
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0

0	1	1	0	1
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Mappa di Karnaugh

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

Formula semplificata: $\overline{cd} + ad + \overline{abd} + \overline{bd}$

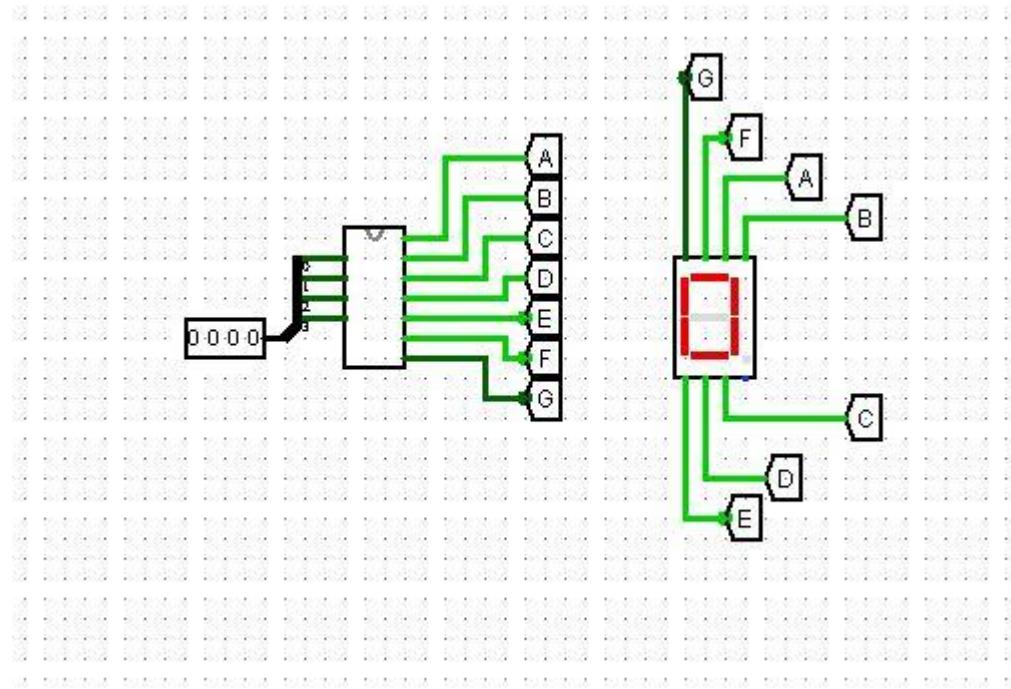


Input del cammino critico maggiore (4): \overline{cd}

Input del cammino critico minore (2): \overline{bd}

Esercizio 3

- 1) Imparare ad usare il display a 7 segmenti;
- 2) Creare un chip che dato un input a 4 bit (positivo) permetta di pilotare il display e di visualizzare il numero corretto da 0 a 9;
- 3) Imparare ad usare l'hex display;
- 4) Fare un programma che prenda in input due numeri a 4 bit e ne mostri la somma a display.



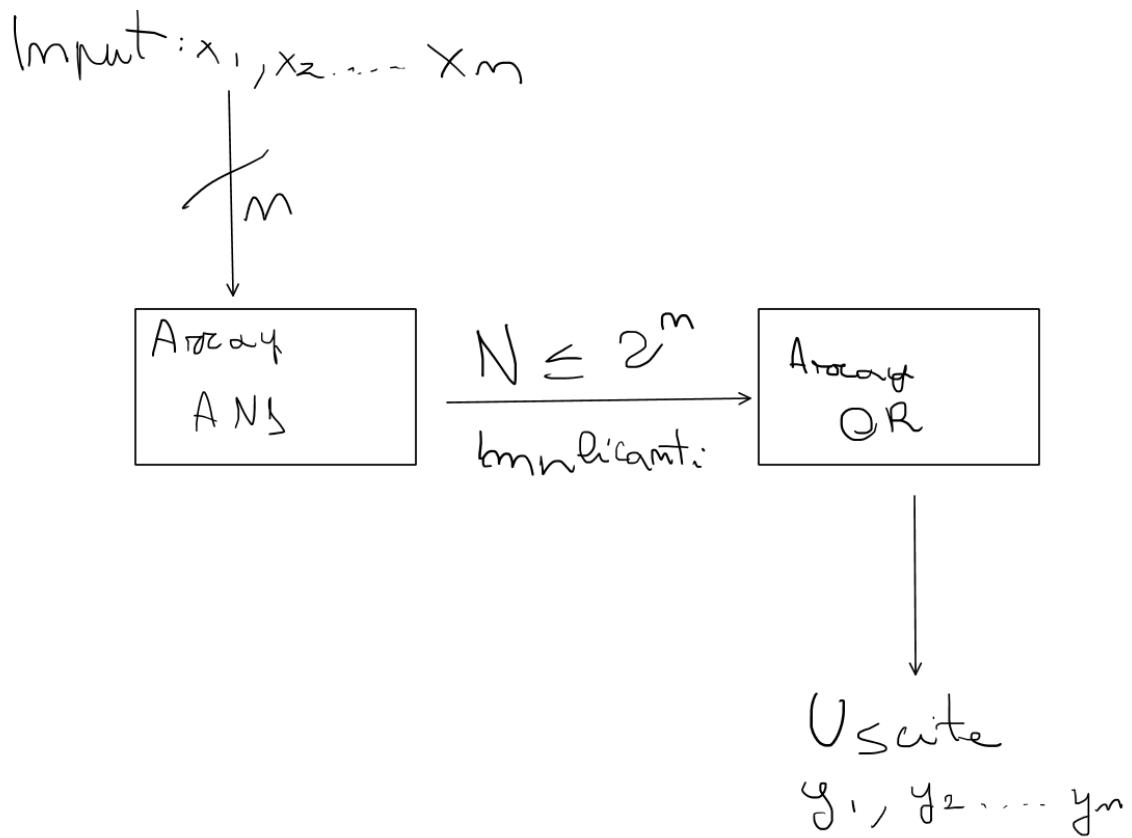
Lezione del 18 Novembre 2021

Programmable logic Arrays (PLA)

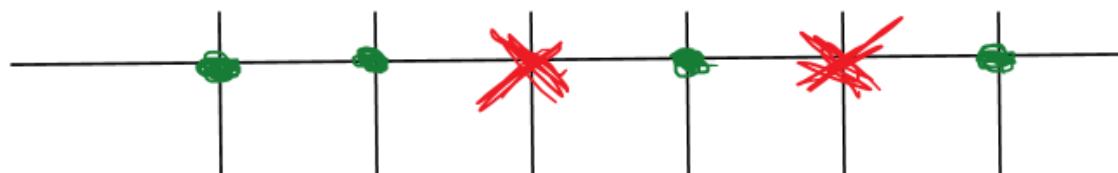
Sono una serie di dispositivi logici programmabili i quali vengono utilizzati per l'implementazione dei circuiti.

Esso è un circuito grezzo su cui tramite un processo automatizzato possiamo forgiare la nostra funzione.

Partiamo con una serie di input: $x_1, x_2 \dots x_n$



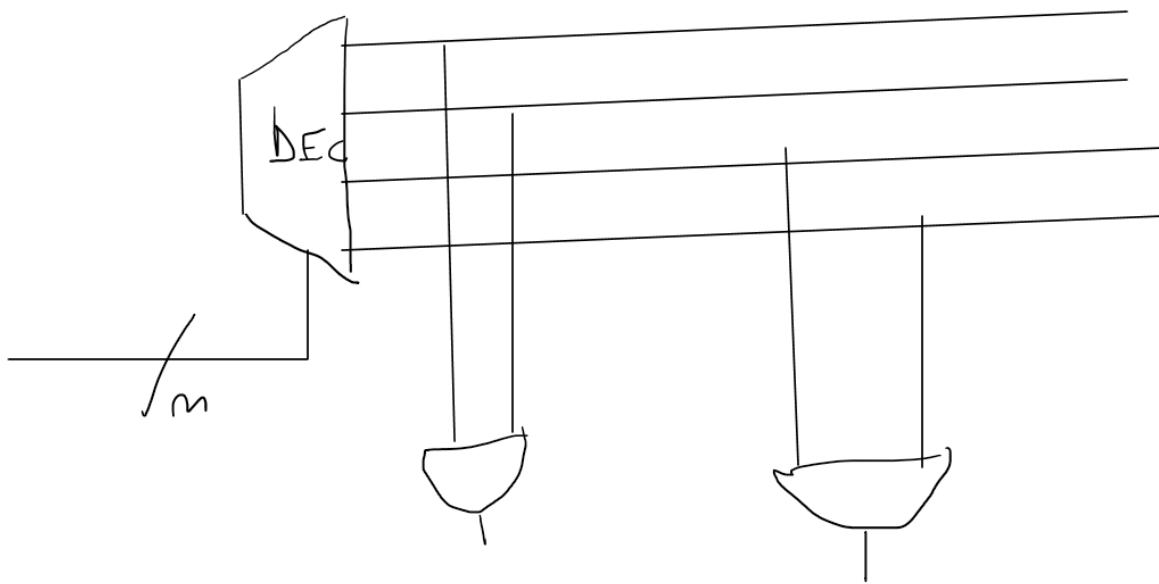
Possiamo scegliere se collegare o scollegare gli input agli implicanti, e la PLA lo fa in automatico a partire dalla tabella di verità. Questo processo avviene tramite fusibile (o antifusibile) bruciando il collegamento con un'elevata tensione.



ROM

Read Only Memory

E' come la PLA, ma "hardwired", cioè l'input è visto come l'indirizzo di una cella, e quindi passa all'interno di un decoder.



PLA e ROM

PLA

Possono essere opportunamente dimensionati e il numero di implicanti può essere limitato.

Se però si hanno $N \leq 2^n$ implicanti non si può rappresentare una qualsiasi funzione logica.

Potrebbe essere necessaria una semplificazione
ROM

Se si hanno n input si hanno sempre 2^n celle di memoria.

Si ha a disposizione ogni mintermine quindi si può implementare una qualsiasi funzione logica.

Lo svantaggio delle ROM è la crescita esponenziale della loro dimensione con n .

Circuiti Sequenziali

In un circuito combinatorio, se inseriamo un input $x^{(t)}$ al tempo t otteniamo un output al tempo $t + \Delta t$, il quale dipende solo dagli input iniziali. Infatti con lo stesso input otteniamo sempre lo stesso output. Invece in un circuito sequenziale viene considerato l'insieme degli input $x^{(t-2)}, x^{(t-1)}, x^{(t)}$ abbiamo un output che può dipendere da tutta o una parte della sequenza di input, non solo da quello iniziale. Quindi questo circuito deve essere in grado di ricordare.

I circuiti sequenziali hanno una memoria interna indicata con il termine "stato".

Lo stato codifica ciò che il circuito ricorda del passato, mentre il circuito combinatorio è un circuito sequenziale che non ricorda. Molto spesso in questi circuiti l'uscita corrisponde allo stato.

Un circuito combinatorio può essere espresso con $y = f(x)$ mentre in un circuito sequenziale abbiamo $\langle y, s_{next} \rangle = f(x, s)$

Dove:

- y = Output ;
- s_{next} = Stato successivo ;
- x = input ;
- s = Stato corrente .

Oltre all'output il circuito sequenziale modifica s stato corrente in s_{next} stato successivo all'esecuzione.

Per ricordare qualcosa ci devono essere almeno ≥ 2 stati diversi. Se il circuito avesse solo uno stato sarebbe un circuito combinatorio.

Dal punto di vista concettuale com'è fatto un circuito sequenziale?

Com'è fatta una memoria?

Nei combinatori:

- L'uscita dipende solo dagli input;
- Il circuito procede in un verso solo.

Nei sequenziali:

- Differenza funzionale;
- Il flusso di elaborazione può anche procedere in senso opposto, con un processo chiamato "retroazione".

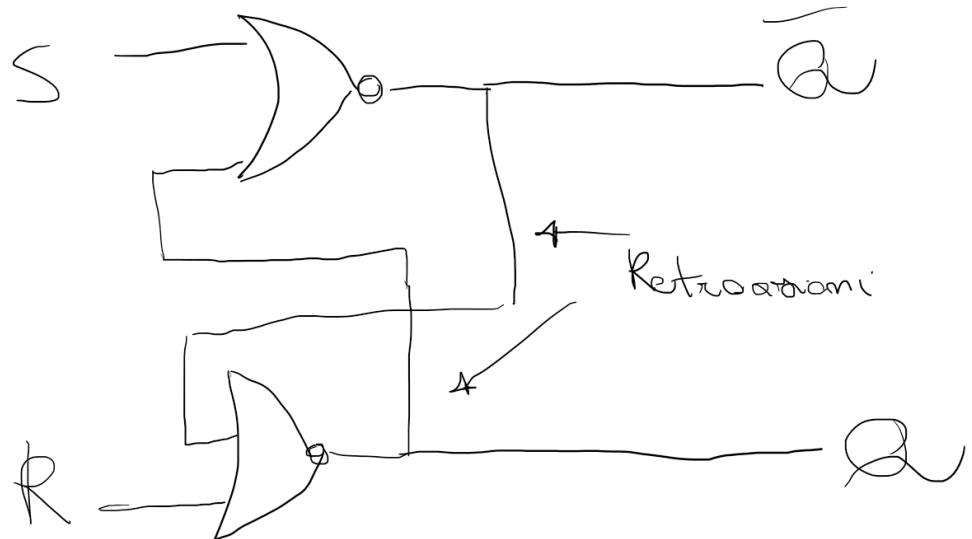
Il bistabile

Partiamo dalle caratteristiche

- Almeno 2 stati
- Retroazione

Il circuito sequenziale che sta alla base di tutti è il Bistabile SR (Set-Reset).

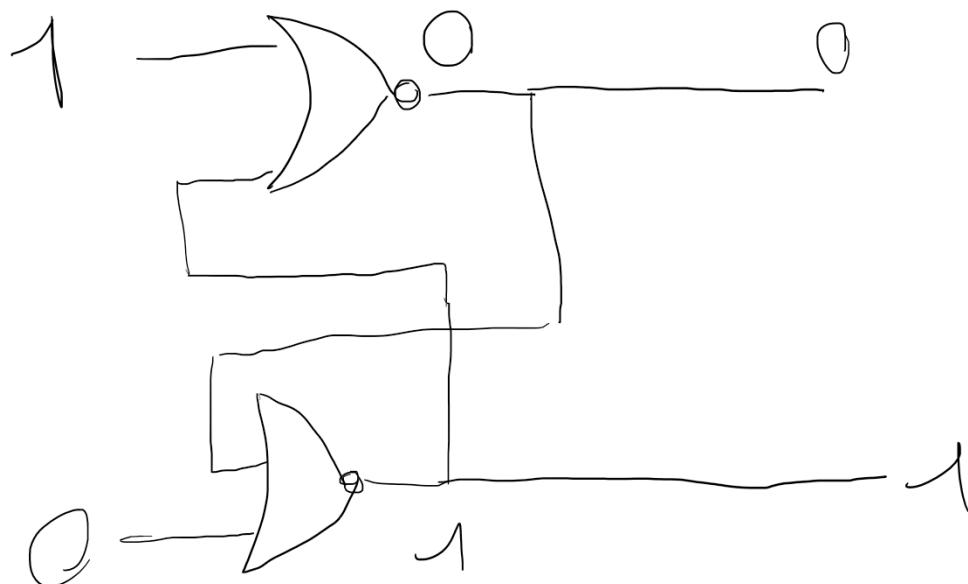
Si ottiene retrozionando due porte NOR (la quale vale 1 solo quando entrambi gli input sono uguali a 0).



Prima memoria, circuito in grado di ricordare un bit.

$S = \text{Set}$ $R = \text{Reset}$ $Q = \text{Stato}$ $\bar{Q} = \text{Stato negato}$

Facendo diventare $S = 1$



Se rимetto a $S = 0$, le uscite rimangono uguali. E' come se Q e \bar{Q} fossero s e s_{next} .

Se mettiamo $R = 1$, avremo come $Q = 0$ e come $\bar{Q} = 1$. Anche riportando $R = 0$ gli stati rimarranno invariati.

Se $R, S = 1$ allora Q, \bar{Q} varranno entrambi 0. Rimettendoli entrambi a 0 sarà impossibile farlo nello stesso istante, quindi ci sarà un Set o un Reset.

Lo stato di arrivo è imprevedibile quindi non si mettono mai S e R uguali a 1 insieme.

Questo circuito si chiama bistabile perché abbiamo due uscite stabili con stessi input (stato di Latch).

Per questo tipo di circuiti esistono tabelle di verità? Si ma in esse vengono inseriti gli input e lo stato.

	00	01	10	11
0	0	0	1	NO
1	1	0	1	NO

Oltre a questo esiste anche la tabella delle transizioni, pressoché simile

S	R	Q	Q_{next}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	NO
1	1	1	NO

Lezione del 22 Novembre 2021

Nella lezione precedente abbiamo visto il funzionamento del Bistabile Set-Reset (SR), il quale è alla base dei circuiti sequenziali. Riprendendo dall'ultima volta, usando l'interpretazione combinatoria della transizione da stato corrente a stato prossimo posso sintetizzare la funzione stato prossimo $Q_{next} = T(s, r, Q)$.

Dato in input (s, 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.

s	r	Q	Q_{next}
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	x → 1
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	x → 1

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

Utilizzo della logica sequenziale

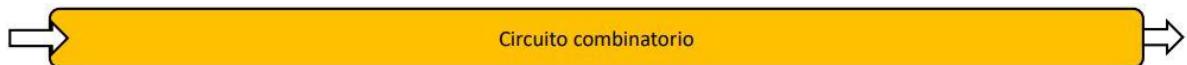
Prima di sviluppare circuiti sequenziali più complessi a partire dal nostro latch SR, chiediamoci a cosa servono in generale questi circuiti.

- Primo utilizzo: conservare risultati di alcune elaborazioni, ad esempio risultati intermedi prodotti dalla ALU, sono di fatto una memoria.
- Secondo utilizzo: affrontare il problema del cammino critico in circuiti combinatori molto complessi.

Approccio: segmentare un circuito combinatorio in diversi sotto-circuiti e attuare una sincronizzazione tra i vari sotto-circuiti.

Architetture sincrone

Un circuito combinatorio complesso presenta un cammino critico elevato, prima che le uscite siano stabili deve passare molto tempo.



Idea: segmentare il circuito combinatorio in circuiti più semplici che, collegati in serie, siano equivalenti al circuito originale.

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 i quali memorizzano il risultato proveniente da sinistra e lo rendono disponibile a destra come catena di montaggio.

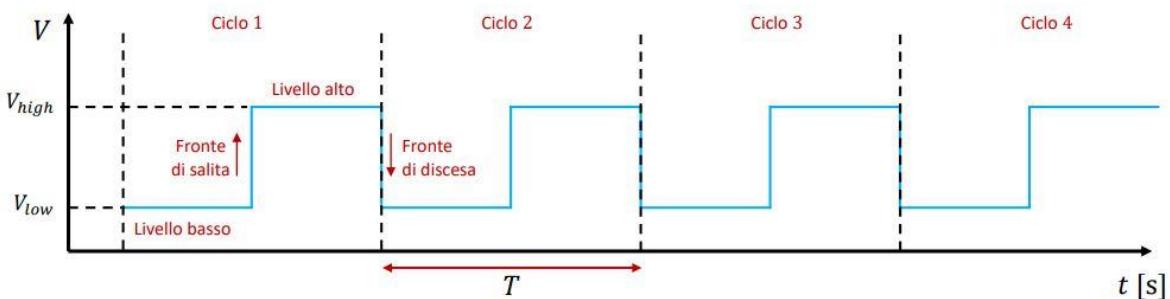
Per poter svolgere questa "staffetta" i vari circuiti devono coordinarsi o sincronizzarsi tra di loro. Per questo abbiamo un "segnale di clock", il quale dirige i passaggi sincronizzando le varie fasi.

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, esso sincronizza l'elaborazione dicendo a ciascun reparto:

- Quando apre il cancello di sinistra e ricevere un dato dal circuito combinatorio;
- Quando apre quello di destra per passare il dato al circuito combinatorio successivo.

Il segnale di Clock



Il periodo T (in secondi) 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.

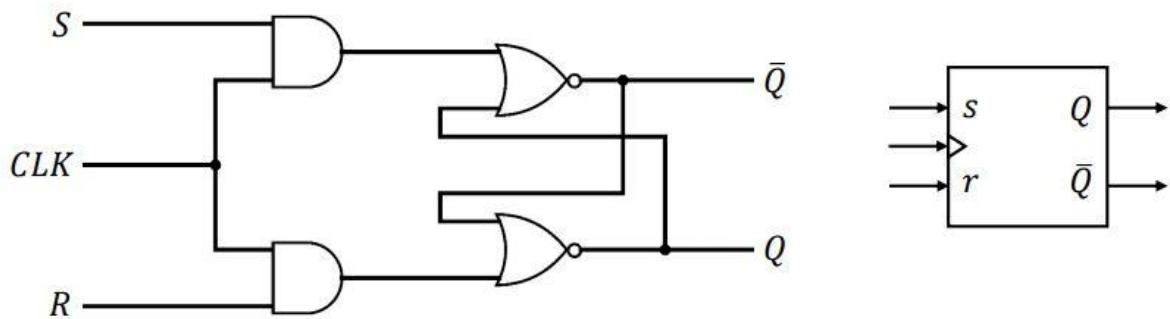
La frequenza di clock (in Hertz, cicli al secondo) è l'inverso del periodo $f = \frac{1}{T}$. I livelli alto e basso del clock corrispondono a valori alti V_{high} e bassi V_{low} di tensione che rappresentano l'1 e lo 0 logico. In una architettura sincrona i circuiti devono obbedire al segnale di clock.

Abbiamo quindi due tipi di architetture:

- Architetture sensibili ai livelli:
 - gli stati cambiano quanto il livello di clock è alto (o basso) ;
 - quando il clock è basso (o alto) non avvengono variazioni di stato, i segnali restano stabili e si propagano nella logica combinatoria ;
 - essendo i segnali stabili, la logica combinatoria può svolgere le sue elaborazioni senza problemi.
- Architetture sensibili ai fronti:
 - più restrittivo, le variazioni di stato possono avvenire solo sui fronti, cioè nell'attimo in cui il clock passa da 1 a 0 o viceversa.

Bistabile Set-Reset (SR) sincrono

Progettiamo una variabile 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.

Sintesi della funzione finale $T(s, r, q, CLK)$

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

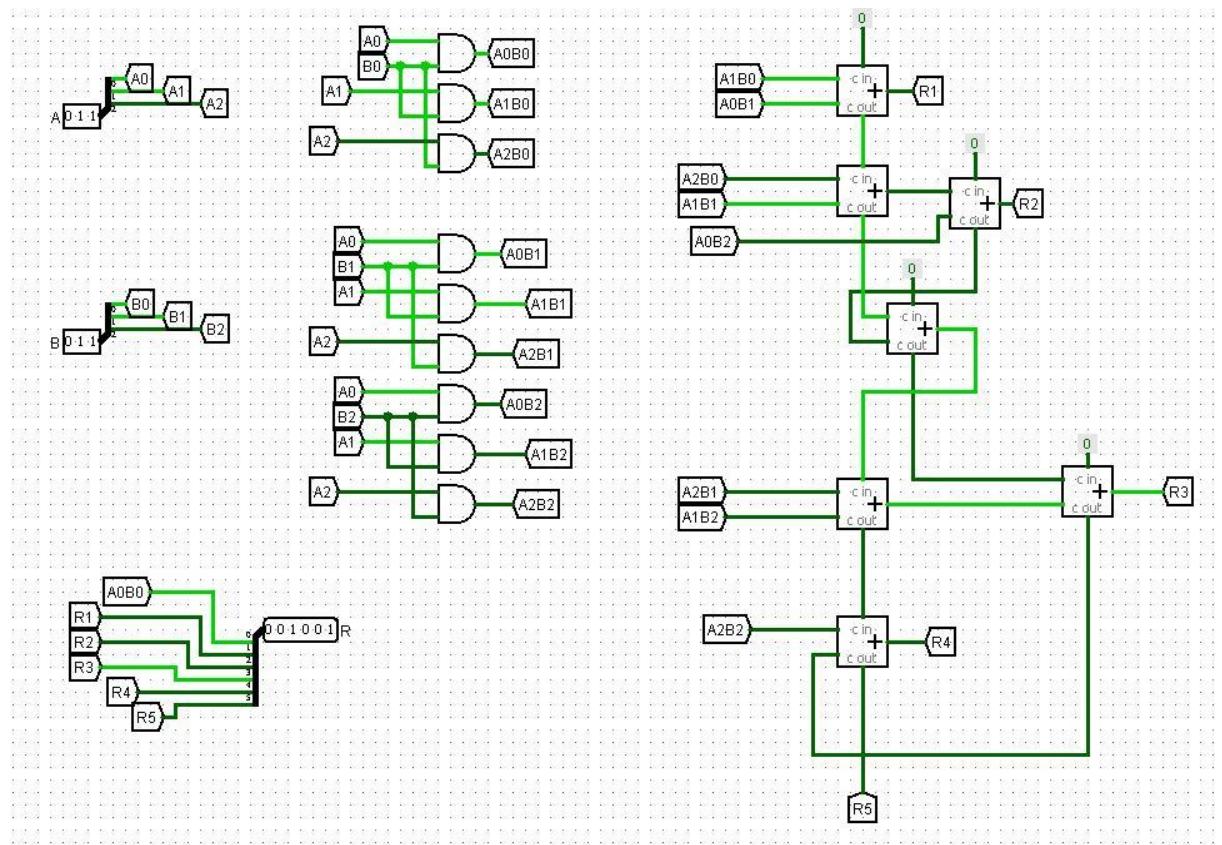
$T(s, r, q, CLK) = \overline{CLK}Q + CLKs + CLKQ\bar{r}$
 $= \overline{CLK}Q + CLK(s + \bar{r}Q)$

Laboratorio del 23 Novembre 2021

Moltiplicazione e ALU

Esercizio 1

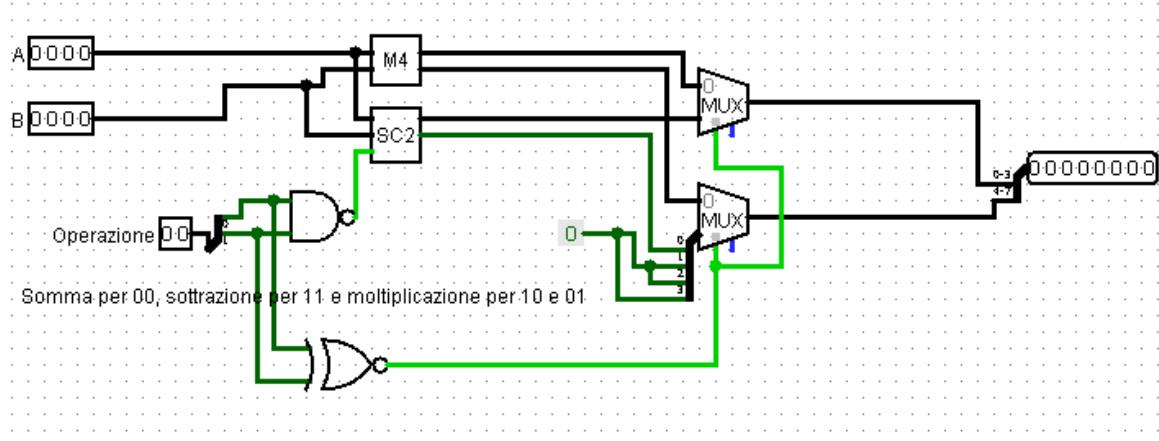
- Si progetti un moltiplicatore HW tra due numeri a 3 bit
Dividiamo il problema dividendo gli input in $a_0 a_1 a_2$ e $b_0 b_1 b_2$



Possiamo importare altri pacchetti di circuiti su Logisim su Project -> Load Library -> Logisim Library

Esercizio 2

- Realizzare una mini-alu in grado di svolgere somma, sottrazione e prodotto a 4 bit. (L'operazione viene selezionata dall'utente)
- Per la realizzazione si utilizzino i circuiti in circuits.zip
- Suggerimento: si utilizzino dei moduli multiplexer/demultiplexer per la selezione dell'operazione e per la selezione del risultato in uscita.

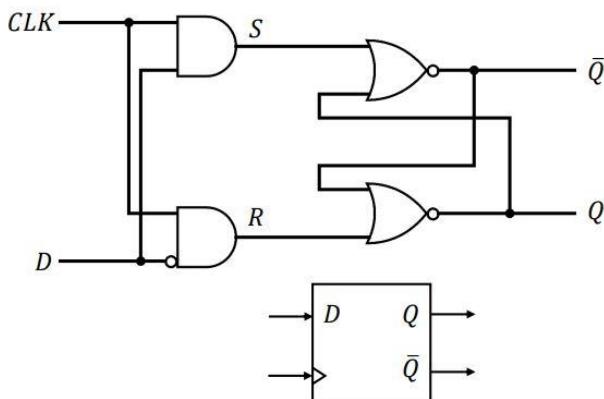


Lezione del 25 Novembre 2021

Latch D

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, quindi 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.



Sintesi dello stato prossimo $T(D, Q, CLK)$

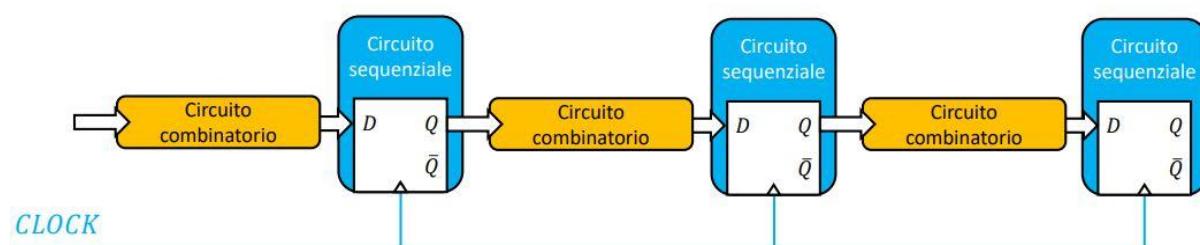
		Input D	
		0	1
Stato Q Clock CLK	00	0	0
	01	1	1
	11	0	1
	10	0	1

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

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

Architettura sincrona con Latch D

Possiamo usare il Latch D nell'architettura sincrona che abbiamo introdotto?



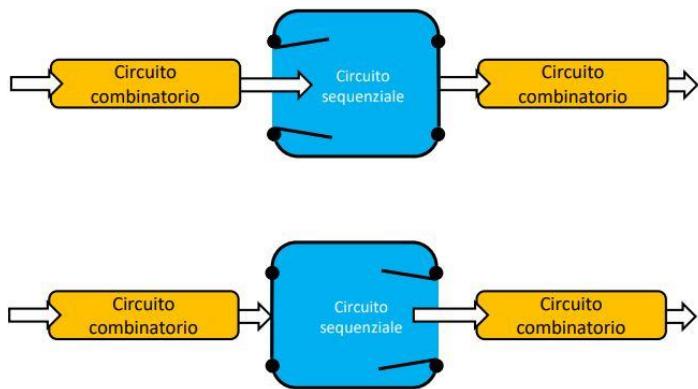
La sensibilità sul livello crea il problema della trasparenza. Quando il clock vale 0 tutti i latch mantengono lo stato corrente, niente cambia, le uscite sono stabili, mentre quando il clock vale 1 tutti i latch sono sensibili a cambiamenti di stati.

Tutti i cancelli sono aperti! Perdiamo la separazione tra stadi e il segnale può attraversare tutto il circuito da sinistra a destra, riottenendo così il problema del cammino critico.

Idea: costruire dei circuiti sequenziali con cancello doppio.

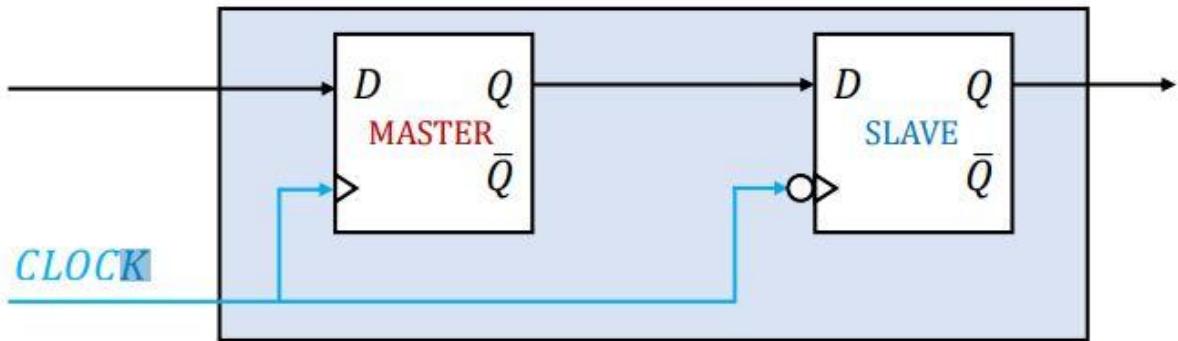
1. Primo step (clock alto):
 - 1.1. Il cancello di sinistra si apre ;
 - 1.2. Il dato proveniente dal primo circuito combinatorio entra nel circuito sequenziale e viene memorizzato;
 - 1.3. Il cancello di destra è chiuso! La separazione tra stadi tiene.
2. Secondo step (clock basso):
 - 2.1. Il cancello di destra si apre ;
 - 2.2. Il dato che era stato memorizzato precedentemente viene mandato in uscita, il secondo circuito combinatorio lo riceve in input ;
 - 2.3. 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.



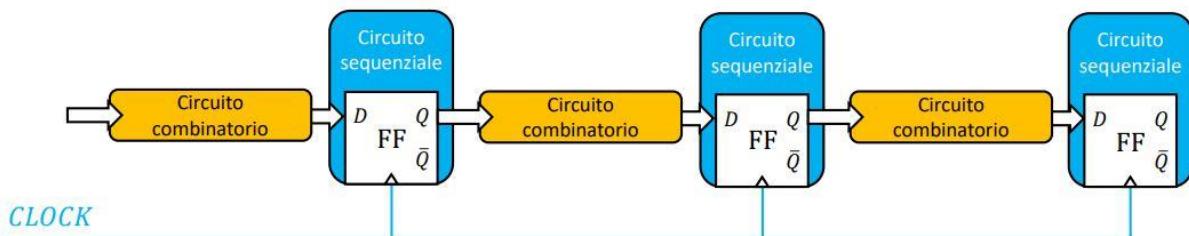
→ $CLOCK = 1$: FLIP

- ◆ master aperto, il dato in input viene scritto nello stato del primo latch D ;
- ◆ slave isolato, continua a mettere in uscita il suo stato corrente (ex stato del latch D master).

→ $CLOCK = 0$: 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.

Il Flip Flop è un circuito sensibile al fronte, dove l'uscita commuta in modo istantaneo sul fronte di discesa del clock.



Lezione del 29 Novembre 2021

Uno dei limiti dei precedenti circuiti è che lo stato è rappresentato da un singolo bit.

Vediamo un utilizzo interessante di questi circuiti per generare macchine a stati finiti (FSM).

Macchine a stati finiti

Finite state machine, con il modello matematico che descrive l'elaborazione e formalizza un'operazione (più debole della macchina di Turing).

Si assume che l'elaborazione sia fatta da una macchina che:

- 1) Riceve degli input (numero finito di input) ;

- 2) Essa possiede un proprio stato interno (numero di stati finito) ;
- 3) Emette informazioni in uscita (numero finito di output) .

La macchina procede per passi in modo iterativo:

- 1) Calcola le uscite come funzione dell'input e dello stato corrente ;
- 2) Calcola un nuovo stato da inserire come input nello stato corrente ;
- 3) Fa assumere un nuovo valore allo stato interno del circuito .

Questo metodo funziona bene con sistemi che richiedono poca memoria, ad esempio con un contapassi, un semaforo o una vending machine.

Questo modello non funziona quando serve TUTTA la sequenza di input precedenti.

Cosa ci ricorda?

I circuiti sequenziali studiati in precedenza.

Come definire una FSM con numero di stati superiori a 2.

Possiamo mettere più bistabili.

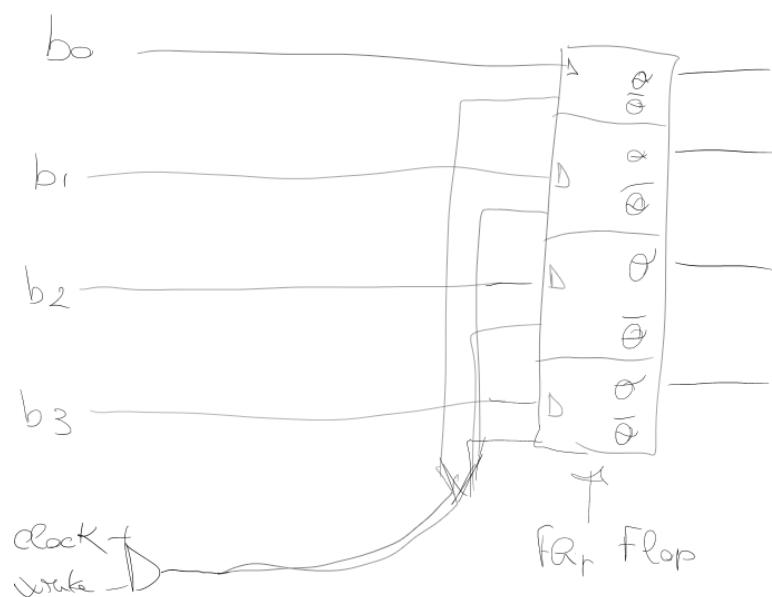
Dobbiamo risolvere 2 problemi:

- 1) Ci serve più di un bit di memoria ;
- 2) Dobbiamo formalizzare le transizioni .

Registri

Memoria su più bit? Usando n Flip Flop abbiamo un registro a n bit

Esempio di registro a 4 bit



Scegliamo attraverso write se far entrare le variabili nei Flip Flop

Nuovo schema



Dal punto di vista matematico?

Come si definisce una FSM

$X = \{x_1, x_2 \dots x_m\}$ insieme degli stati

$x^i \in X$ stato iniziale della macchina

$I = \{i_1, i_2 \dots i_w\}$ insieme degli input

$O = \{o_1, o_2 \dots o_k\}$ insieme degli output

Funzione $T: X \cdot I \rightarrow X$

Funzione delle transizioni o degli stati prossimi

Se consideriamo $g()$ funzione di uscita se:

- 1) $g: X \rightarrow O$ avremo una Macchina di Moore, dove l'uscita dipende solo dallo stato;
- 2) $g: X \cdot I \rightarrow O$ avremo una Macchina di Mealy, dove l'uscita dipende dallo stato e dall'input.

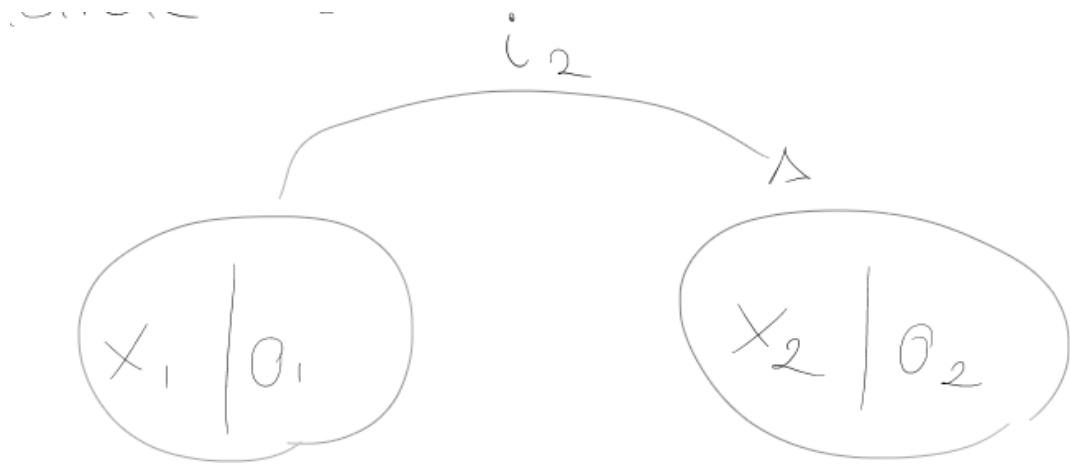
Grafo delle transizioni di stato

Possiamo rappresentare le transizioni della FSM attraverso un grafo orientato.

Esso è composto da nodi e connessioni, abbiamo un nodo per ogni stato della macchina.

Le transizioni si modellano come archi.





Cambia lo stato al momento dell'inserimento della variabile, cambiando l'output (Macchina di Moore)

Esempio:

- 4 pulsanti: V, R, B, Clear ;
- Led Rosso serratura chiusa ;
- Led Verde serratura aperta ;
- La sequenza V, R, B apre la cassaforte ;
- Il tasto clear se la cassaforte è chiusa resetta la pulsantiera mentre se la cassaforte è aperta la chiude.

Primo step: definiamo il modello matematico

Insieme degli stati $X = \{C_0, C_1, C_2, A\}$

A stato in cui la cassaforte è aperta

C_0, C_1, C_2 stati di riconoscimento degli input, allo stato C_i significa che sono stati riconosciuti i input

Stato iniziale C_0

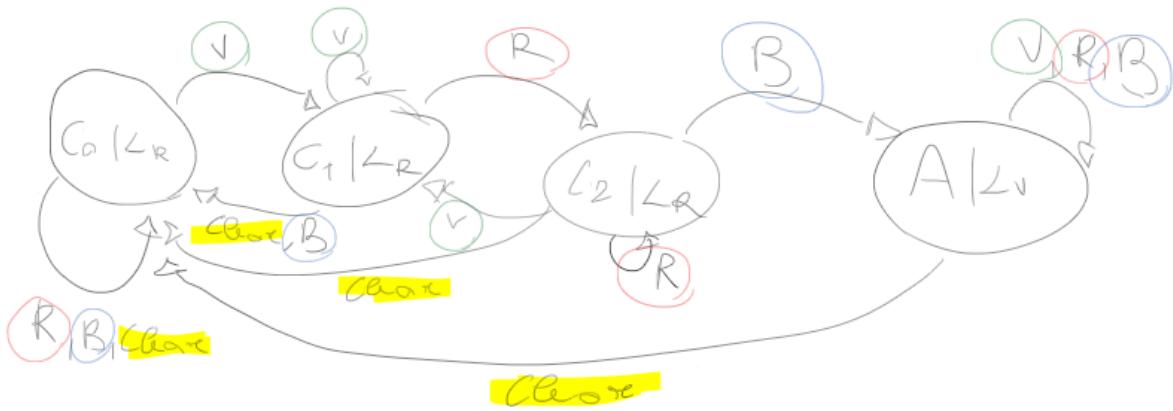
Insieme degli input $I = \{V, R, B, Clear\}$

Insieme degli output $O = \{L_r, L_v\}$

Funzione di uscita $g(C_i) = L_r \quad \forall i \in \{0, 1, 2\} \quad g(A) = L_v$

Per gli output ci basta un bit $L_r = 0$ e $L_v = 1$

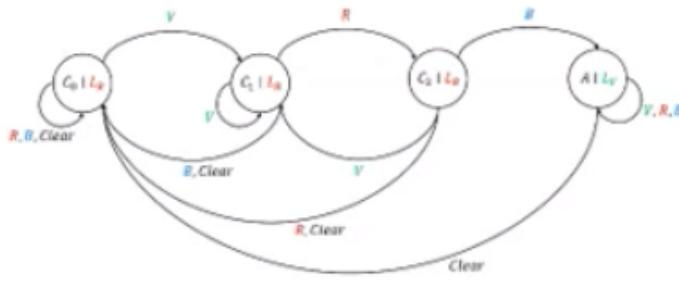
Gli input vanno convertiti in 2 bit $V = 00$ $R = 01$ $B = 10$ $Clear = 11$



	V	R	B	$Clear$	Uscita
C_0	C_1	C_0	C_0	C_0	L_r
C_1	C_1	C_2	C_0	C_0	L_r
C_2	C_1	C_2	A	C_0	L_r
A	A	A	A	C_0	L_v

Com'è fatto il circuito?

- Insieme degli stati $X = \{C_0, C_1, C_2, A\}$
 - Stato C_i serratura chiusa e ho riconosciuto i primi i simboli della combinazione
 - Stato A , serratura aperta
- stato iniziale della macchina $C_0 \in X$
- insieme degli input $I = \{V, R, B, Clear\}$
- insieme degli output $O = \{L_R, L_V\}$
 - L_R led rosso
 - L_V verde
- funzione di uscita $g(C_i) = L_R \forall i \in \{0,1,2\}, g(A) = L_V$



	V	R	B	Clear	Uscita
C_0	C_1	C_0	C_0	C_0	L_R
C_1	C_1	C_2	C_0	C_0	L_R
C_2	C_1	C_0	A	C_0	L_R
A	A	A	A	C_0	L_V

Input i	Rappresentazione binaria		Uscita	Rappresentazione binaria		
	q_1	q_0		i_1	i_0	y_0
V	0	0		0	0	
R	0	1		0	1	
B	1	0		1	0	
Clear	1	1		1	1	

Tabella di verità

q_0	q_1	i_0	i_1	q_0^{next}	q_1^{next}	y
0	0	0	0	0	1	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	1	0
0	1	0	1	1	0	0

0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	1	0
1	0	0	1	0	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	0
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	0	0	0

Laboratorio del 30 Novembre 2021

Memorie

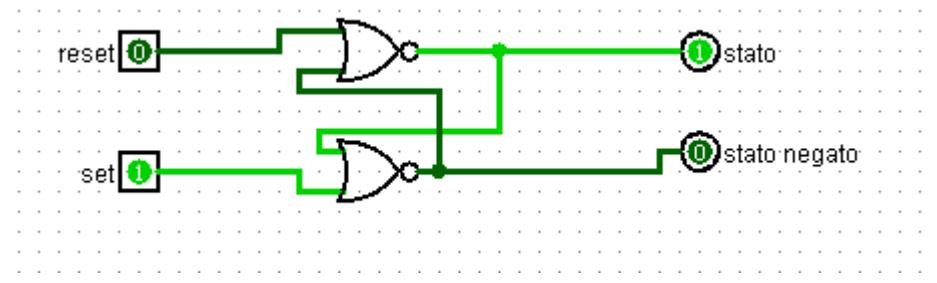
Servono nei circuiti sequenziali per ricordare gli input.

In programmazione per salvare i valori vanno assegnate a delle variabili (anche se esse poi possono cambiare).

Il circuito con memoria avrà un output che dipende dall'input corrente e precedente.

Esercizio 1

Bistabile asincrono SR con due NOR



Circuito in grado di ricordare l'input nello stato.

Mettendo entrambi a 1 e poi entrambi a 0, lo stato e lo stato negato dipenderanno dalla variabile passata per ultima a 0.

Questo circuito è sincrono o asincrono?

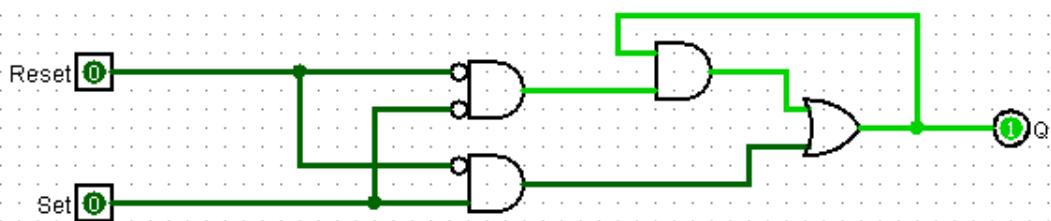
E' asincrono dato che non abbiamo un clock che ci dice quando eseguire il set ed il reset.

Tabella delle transizioni di stato:

S	R	Q(stato prec)	Q*(stato succ)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

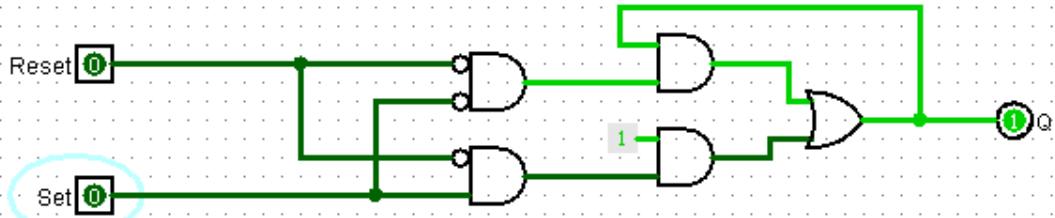
$$Q_{next} = \overline{SR}Q_{prec} + SR$$

Circuito derivante dalla SOP



Dopo averlo provato avviene un errore di oscillation apparente, dopo aver messo set su 1, ma nel momento in cui mettiamo R e S a 0 il circuito fa rimanere costante l'output a 1, anche se deriva da un OR con due linee a 0.

La soluzione è aggiungere un ritardo temporale sul set

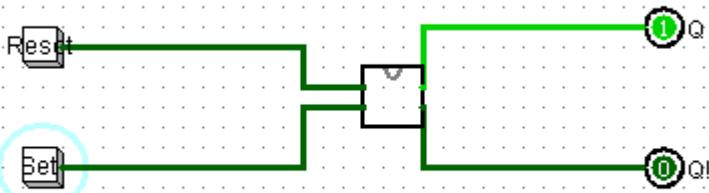


Se consideriamo le X come 1 avremo

$$Q_{next} = S + \bar{R}Q_{prec}$$

Esercizio 1

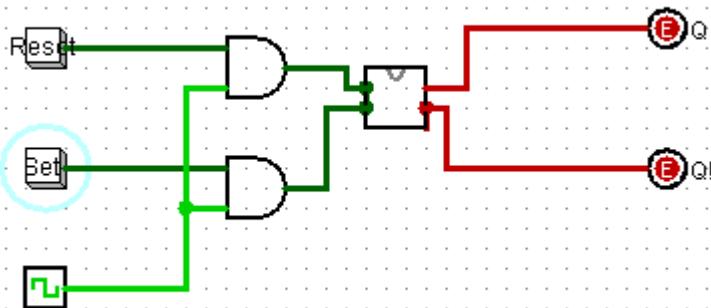
Realizzare il circuito precedente con bottoni e tunnel



Così non avremo mai la combinazione proibita dove S e R sono 1

Esercizio 2

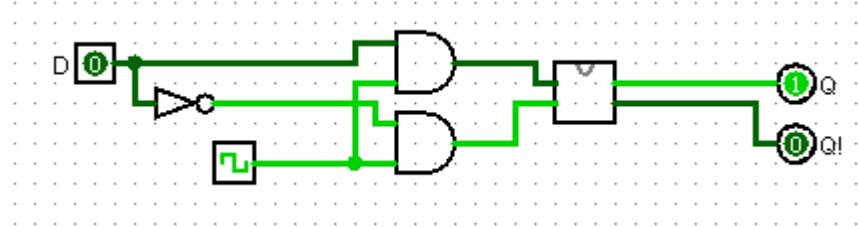
- Si aggiunga un clock (frequenza 0.5 Hz) e due porte AND al circuito realizzato nell'esercizio 1 per ottenere un latch sincrono SR
- Si osservi come cambia la risposta del circuito con diverse frequenze di clock



La memoria viene cambiata in sincrono con il clock

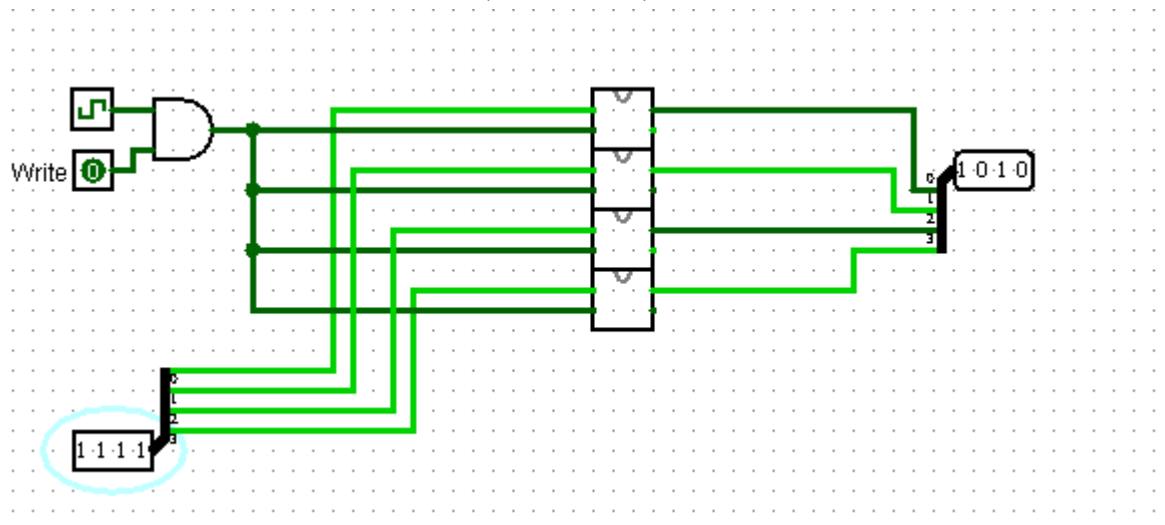
Esercizio 3

- Utilizzando il bistabile sincrono creiamo un latch sincrono D



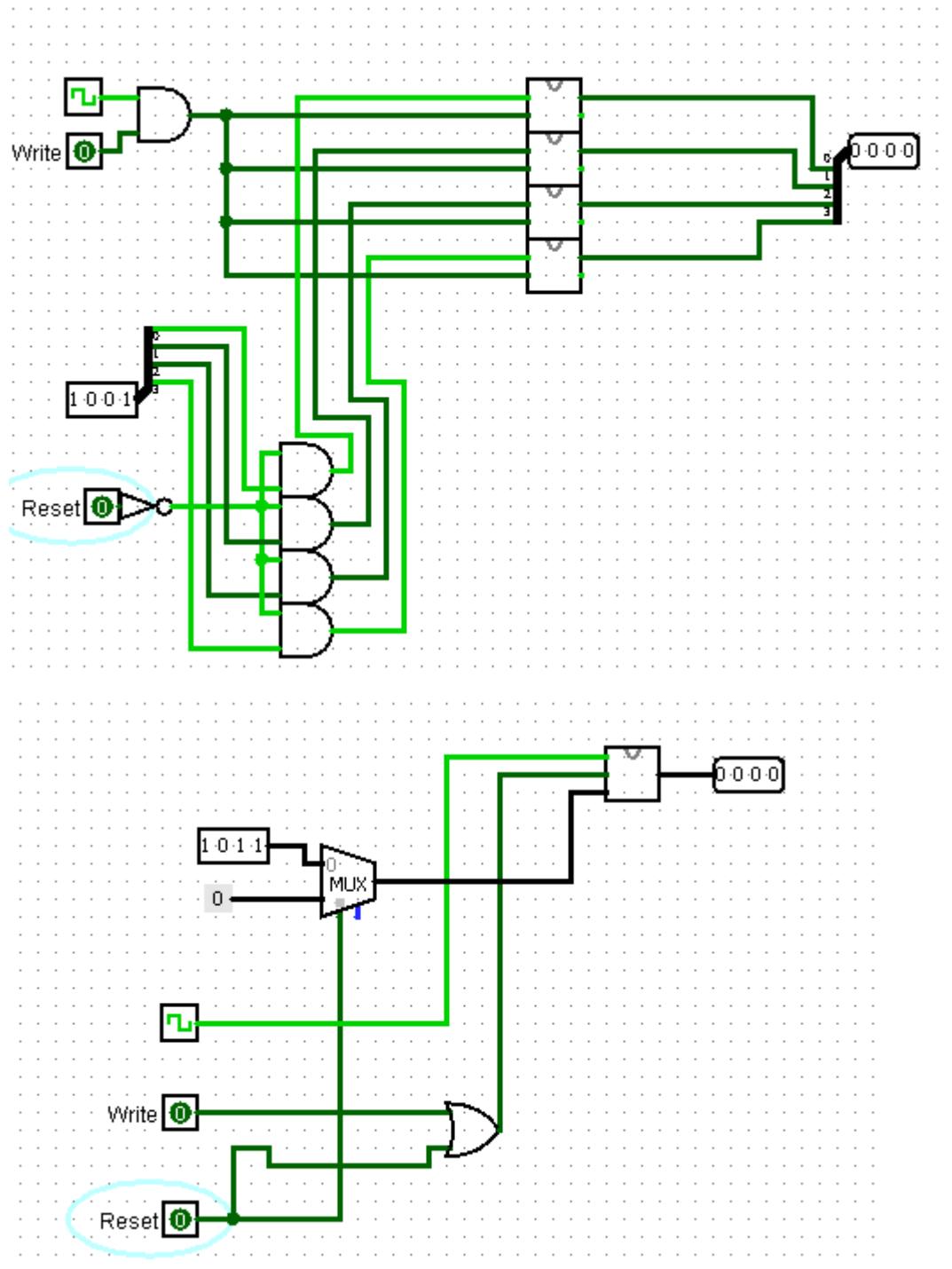
Esercizio 4

- Si utilizzino 4 latch D sincroni per realizzare un banco di memoria a 4 bit
- Tale circuito sarà caratterizzato da:
 - 4 bit in ingresso
 - 1 ingresso per clock
 - 1 ingresso per il bit di write, il quale abilita o inibisce la scrittura nel banco di memoria quando è posto a 1



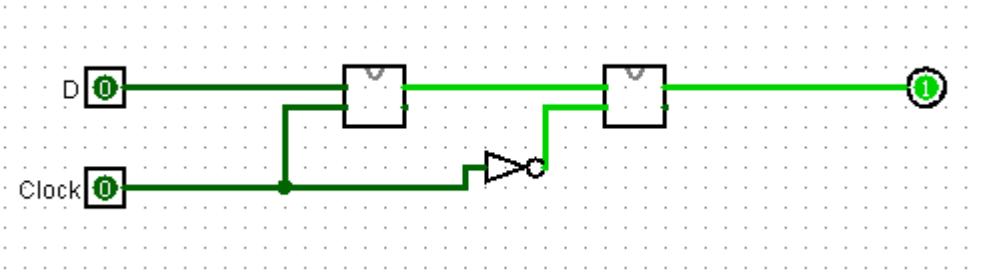
Esercizio 5

- Implementare un bit di input per il reset della memoria

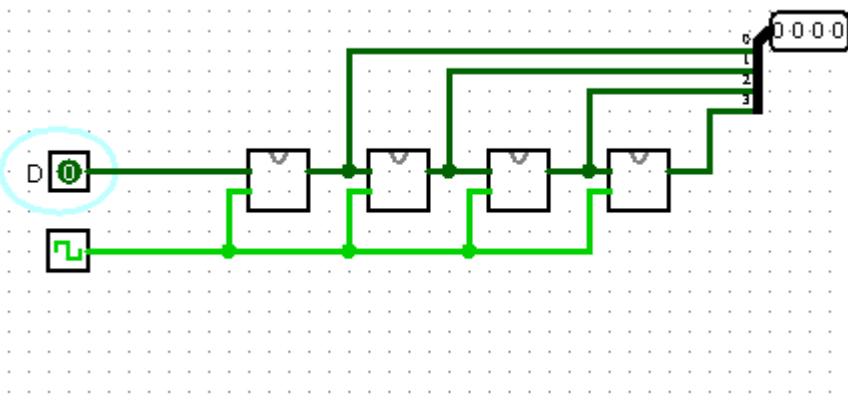


Esercizio 6

- Si realizzi un Flip Flop partendo da due Latch D



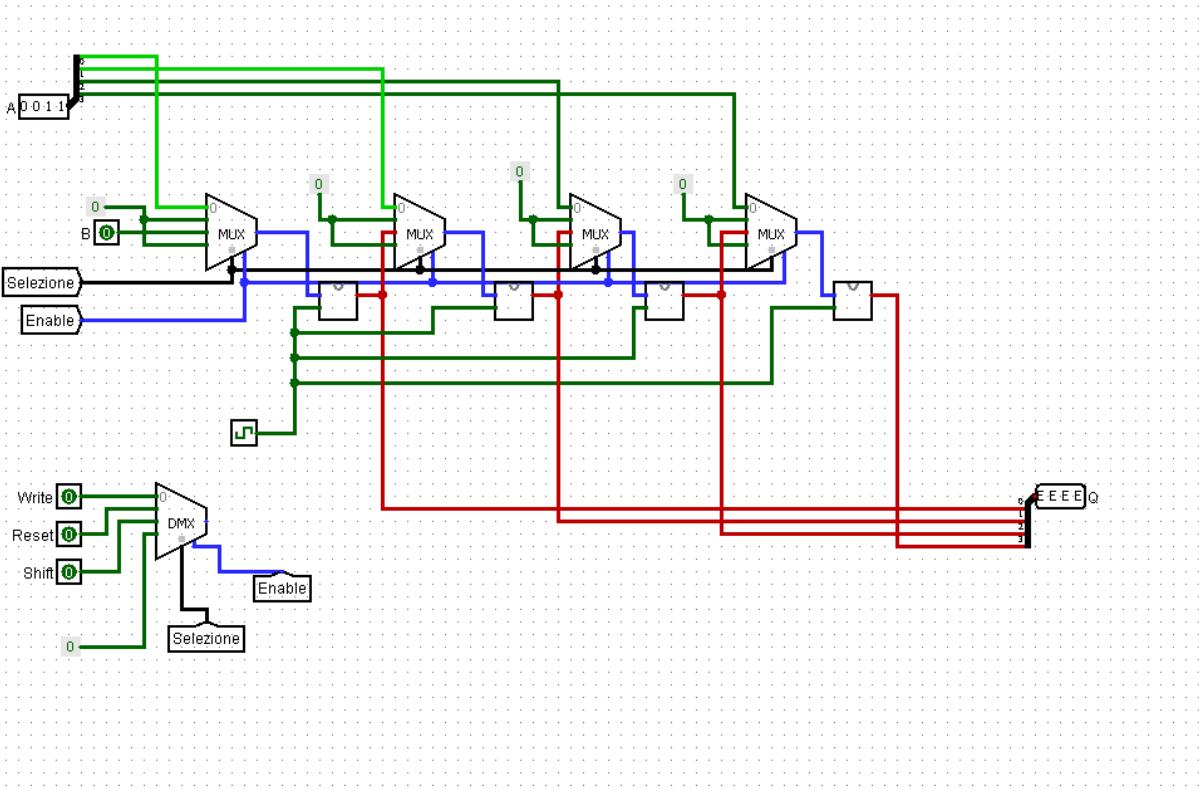
- Si realizzi quindi uno shift register a 4 bit



Esercizio 7

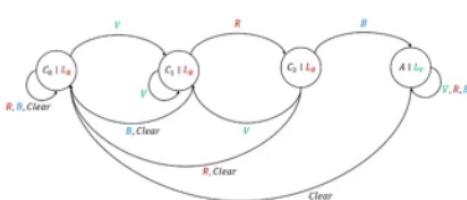
Si realizzi un banco di memoria a 4 bit che permetta:

- Scrittura di 4 bit
- Reset dei 4 bit
- Shift della parola verso destra attraverso l'inserimento di un bit a sinistra



Lezione del 2 Dicembre 2021

- Insieme degli stati $X = \{C_0, C_1, C_2, A\}$
 - Stato C_i serratura chiusa e ho riconosciuto i primi i simboli della combinazione
 - Stato A , serratura aperta
- stato iniziale della macchina $C_0 \in X$
- insieme degli input $I = \{V, R, B, Clear\}$
- insieme degli output $O = \{L_R, L_V\}$
 - L_R led rosso
 - L_V verde
- funzione di uscita $g(C_i) = L_R \forall i \in \{0,1,2\}, g(A) = L_V$



	V	R	B	Clear	Uscita
C0	C1	C0	C0	C0	L_R
C1	C1	C2	C0	C0	L_R
C2	C1	C0	A	C0	L_R
A	A	A	A	C0	L_V

Stato Q	Rappresentazione binaria	
	q_1	q_0
C_0	0	0
C_1	0	1
C_2	1	0
A	1	1

Input i	Rappresentazione binaria	
	i_1	i_0
V	0	0
R	0	1
B	1	0
Clear	1	1

Uscita	Rappresentazione binaria	
	y_1	y_0
L_R	0	
L_V		1

q_1	q_0	i_1	i_0	q_1^{next}	q_0^{next}	y_0
0	0	0	0	0	1	0
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	1	0
1	0	0	1	0	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	0
1	1	0	0	1	1	1
1	1	0	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	0	0	1

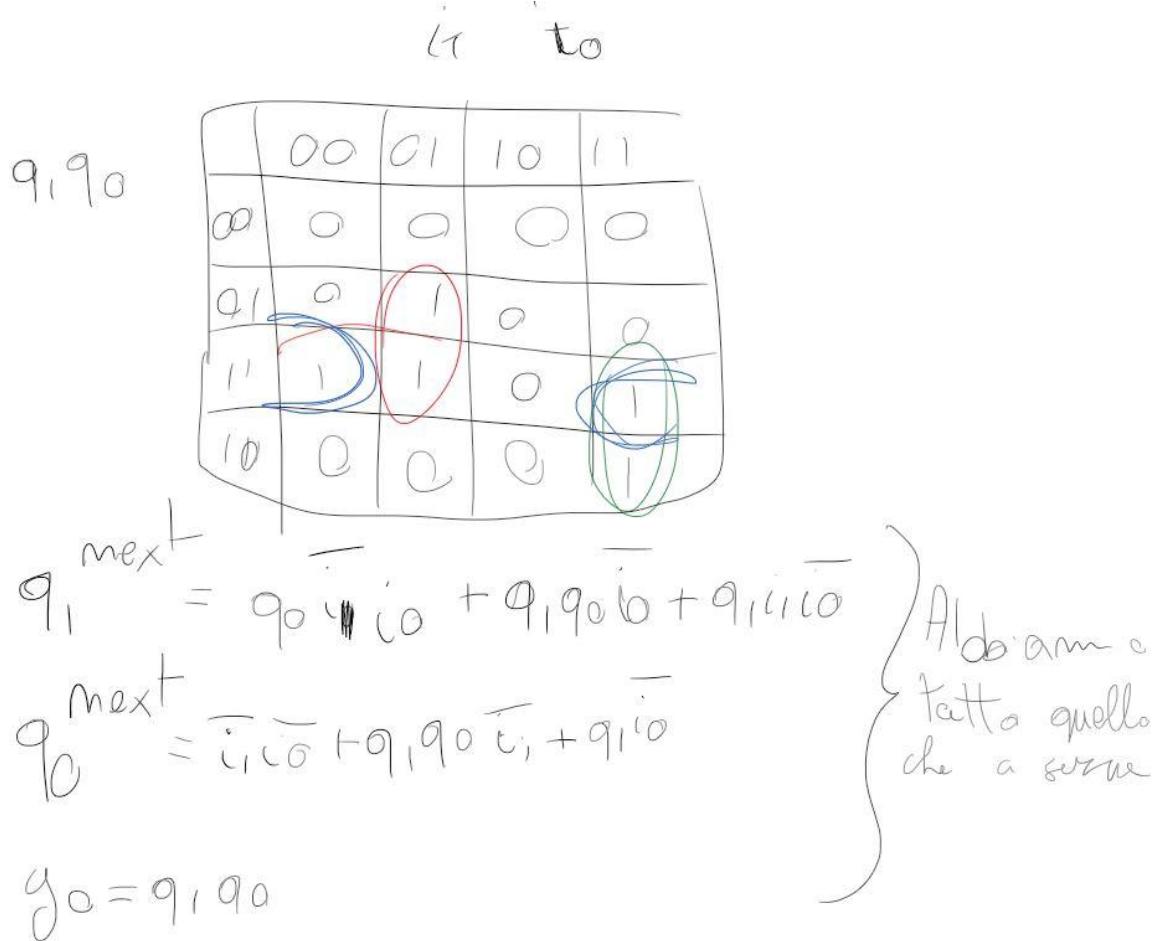
Costruendo la SOP della tabella di verità abbiamo

$$q_0^{next} = \overline{q_1} \overline{q_0} \overline{i_1} \overline{i_0} + \overline{q_1} q_0 \overline{i_1} \overline{i_0} + q_0 \overline{q_1} \overline{i_1} \overline{i_0} + q_1 \overline{q_0} i_1 \overline{i_0} + q_1 q_0 \overline{i_1} \overline{i_0} + q_1 q_0 \overline{i_1} i_0 + q_1 q_0 i_1 \overline{i_0}$$

$$q_1^{next} = \overline{q_1} q_0 \overline{i_1} i_0 + q_1 \overline{q_0} i_1 \overline{i_0} + q_0 q_1 \overline{i_1} \overline{i_0} + q_1 q_0 \overline{i_0} i_1 + q_1 q_0 \overline{i_0} i_1$$

$$y_0 = q_0 q_1$$

Abbiamo sintetizzato le espressioni booleane. Possiamo semplificare usando le mappe di Karnaugh, con una tabella per ogni funzione logica:

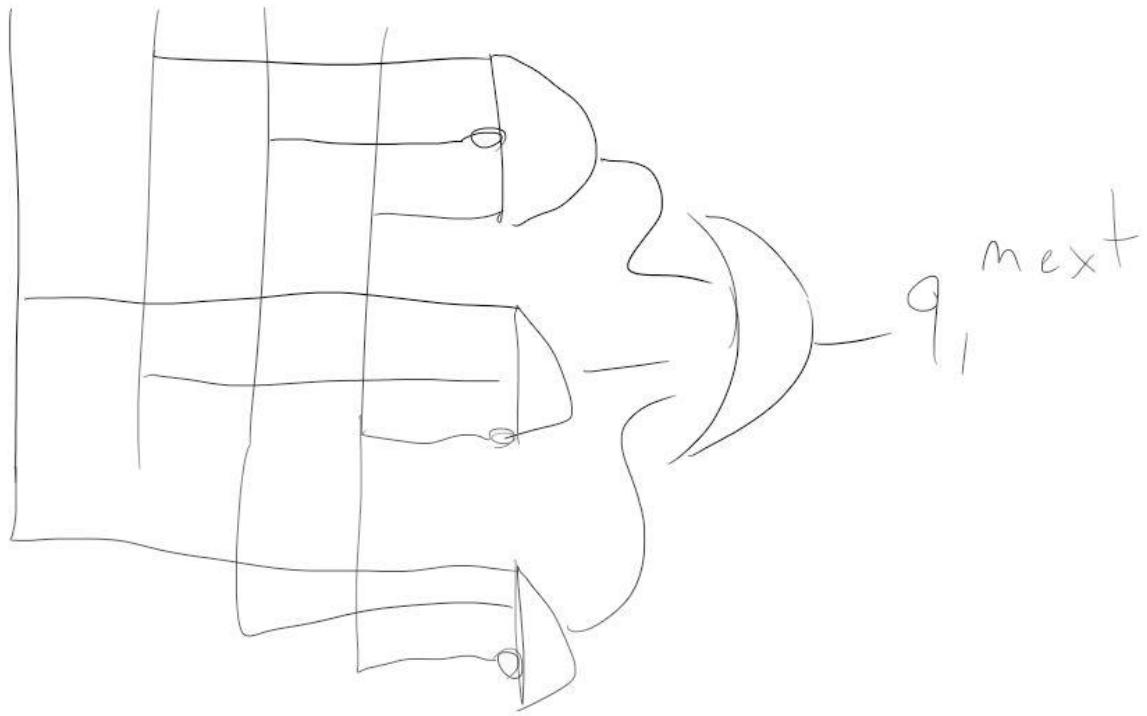


$$q_1^{next} = q_0 \overline{i_1} \overline{i_0} + q_1 q_0 \overline{i_1} + q_1 \overline{i_1} \overline{i_0}$$

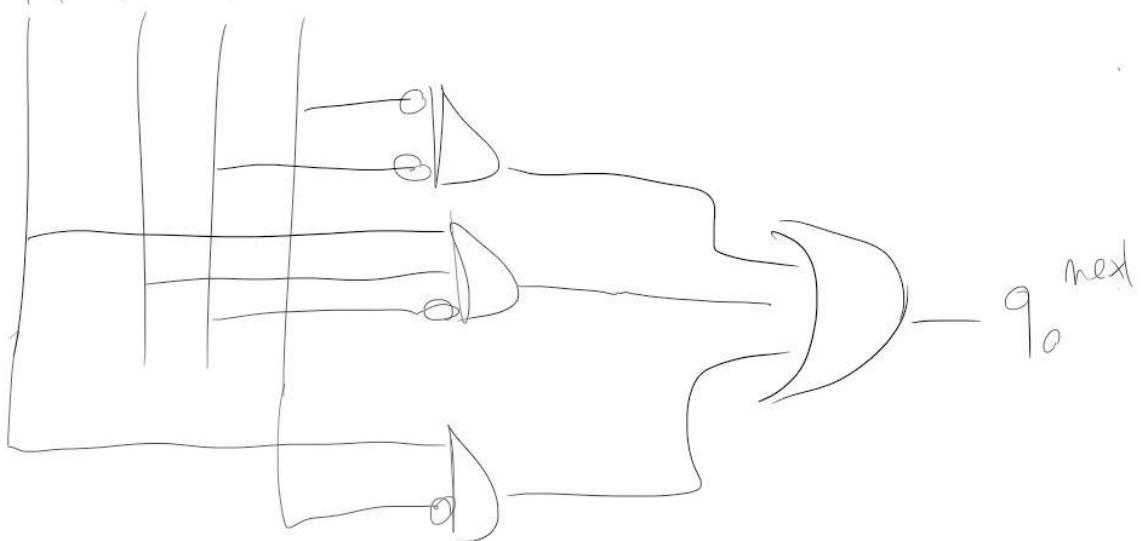
$$q_0^{next} = \overline{i_1} \overline{i_0} + q_1 q_0 \overline{i_1} + q_1 \overline{i_0}$$

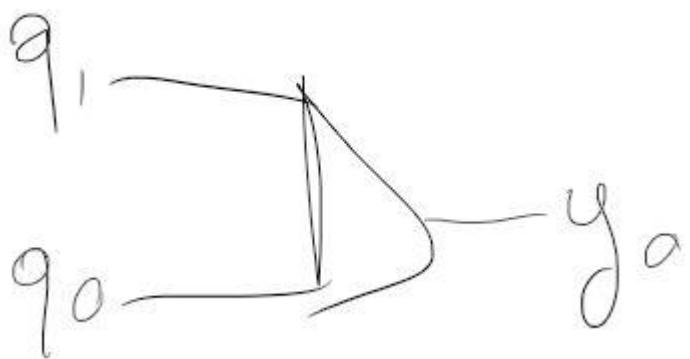
$$y_0 = q_1 q_0$$

$q_1 \ q_0 \ i_1 \ i_0$

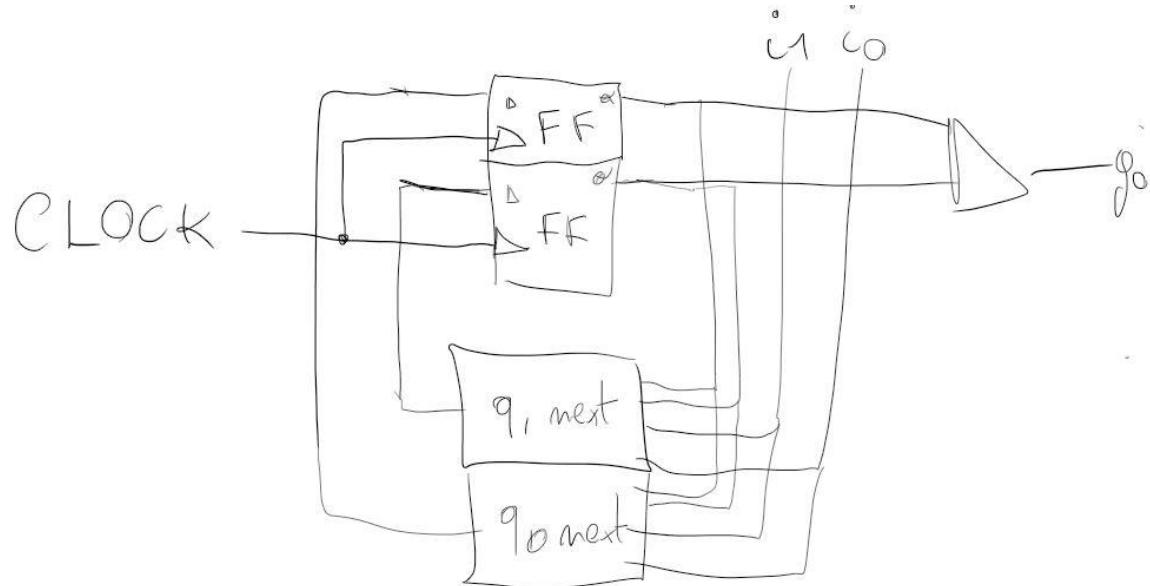


$q_1 \ q_0 \ i_1 \ i_0$



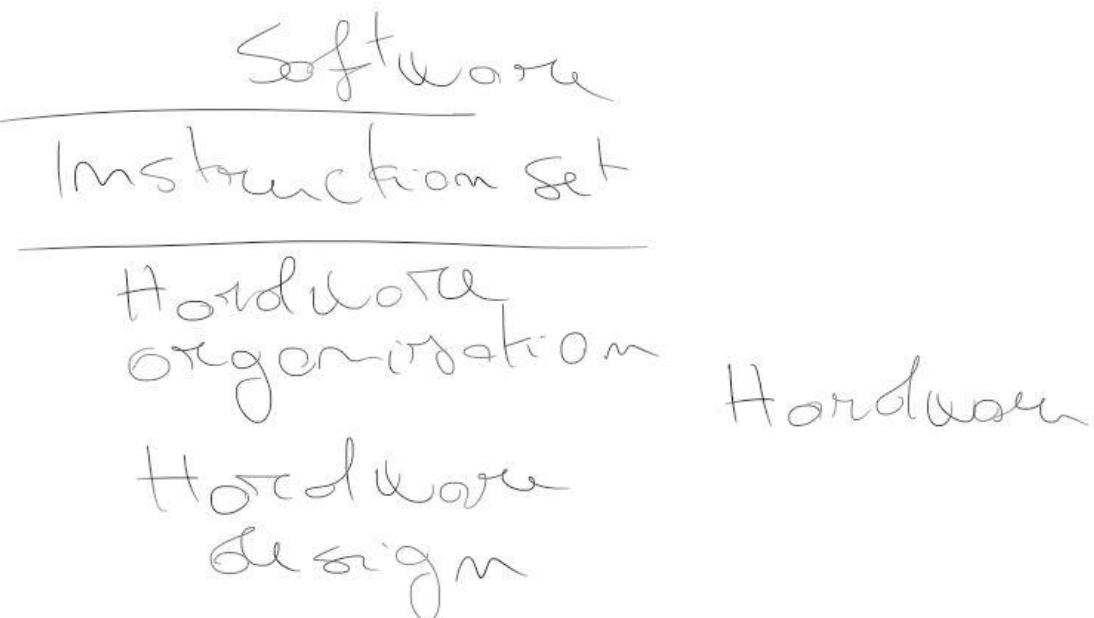


Adesso li inscatoliamo ottenendo



Nei Flip Flop viene memorizzato il valore dello stato corrente e ne viene restituita una copia. I blocchetto di q_0^{next} e di q_1^{next} copiano una transizione di stato.

CPU a singolo ciclo



- Obiettivo: costruire una CPU in grado di eseguire le istruzioni dell' ISA MIPS
- CPU a singolo ciclo: un'istruzione per ciclo di clock

Cos'è un'istruzione?

Un'istruzione è una stringa binaria tale che la CPU possa decodificarla ed eseguire l'elaborazione che essa rappresenta. Il linguaggio binario con cui sono espresse le istruzioni si chiama linguaggio macchina. Tante istruzioni formano il linguaggio macchina stesso.

Macro elementi della CPU

Fetch -> Decode -> Execute -> Memory -> Write Back

- Fetch: prendo l'informazione
- Decode: la decodiflico
- Execute: eseguo l'istruzione
- Memory: scrivo su memoria
- Write Back: scrivo sui registri

Abbiamo altri due macro-elementi fondamentali della CPU:

- Data path, cioè il percorso che le informazioni seguono nella CPU, attraversando i sotto-componenti (quindi insieme dei percorsi che possono fare le istruzioni);
- Logica di controllo, gestita dall'unità di controllo, la quale manovra il data path e gli "scambi" e controlla i sottocomponenti della CPU (Come un Multiplexer).

Istruzioni della CPU

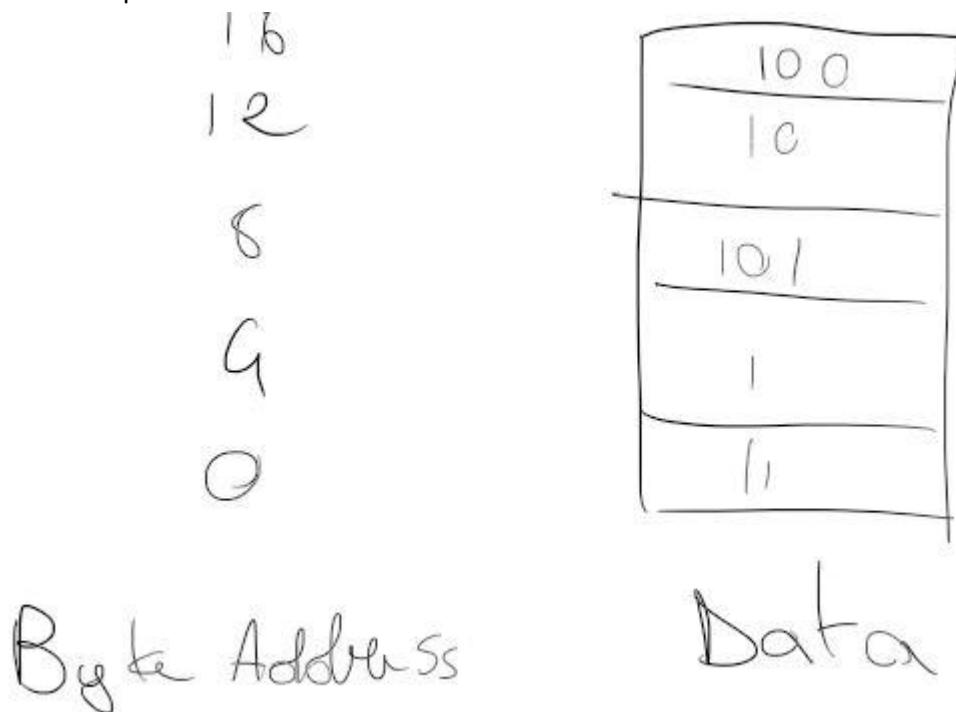
- 1) Istruzioni Aritmetico-Logiche ;
- 2) Accesso alla memoria dati ;
- 3) Controllo di flusso .

Ogni istruzione viene codificata su 32 bit divisi in campi. In MIPS abbiamo 3 formati:

- 1) Formato R (register) per le istruzioni aritmetico-logiche che operano sui registri ;
- 2) Formato I (immediate) per le istruzioni aritmetico-logiche che operano su valori immediati (costanti) per le istruzioni di salto combinato ;
- 3) Formato J (jump) per le istruzioni di salto non condizionato .

CPU: La Memoria

La memoria è come un array unidimensionale dove ogni elemento si chiama parola di memoria.



Se ogni parola è di n bit (ampiezza) la memoria può contenere 2^n parole (altezza). Ad esempio con $n = 32$ si hanno 2^{32} parole, quindi circa 4 GB. La parola è l'unità base di trasferimento da e verso la memoria in byte (8 bit).

Ogni parola è associata ad un indirizzo multiplo di 4 (32 bit \rightarrow 4 byte)

Lezione del 13 Dicembre 2021 (Appunti)

Big Endian contro Little Endian

Ricordiamo la memoria come un array di byte, con i propri indirizzi che si riferiscono a parole di 4 byte, dove l'indirizzo stesso è quello di uno dei 4 byte.

In Big Endian (MIPS) nell'indirizzo viene scritto il byte più significativo mentre in Little Endian il contrario.

Ingredienti della CPU: Memoria

La memoria centrale sarà un modulo a cui possiamo accedere.

Abbiamo due sezioni di memoria:

- 1) Memoria istruzioni ;
- 2) Memoria dati.

La prima viene usata solo in lettura.

L'indirizzo della prossima istruzione sta all'interno di un registro chiamato Program Counter (PC).

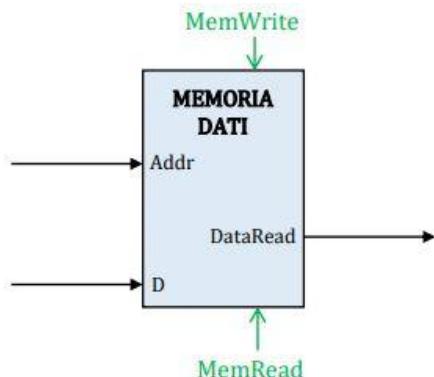
La memoria dati viene usata sia in lettura, sia in scrittura, ma non nello stesso ciclo di clock.

Input:

- Addr = indirizzo della parola in memoria ;
- MemRead = segnale di controllo (1 bit) dove con 1 legge mentre con 0 è a riposo ;
- MemWrite = segnale di controllo (1 bit) dove con 1 scrive mentre con 0 è a riposo ;
- D = il dato (o parola) da trasferire .

Output:

- Dataread = il dato (o parola) recuperato dalla memoria .



Random Access Memory (RAM) : tempo di accesso in memoria fisso, indipendente dall'indirizzo.

MemRead e MemWrite in ogni ciclo di clock solo una di esse sarà uguale a 1 (00, 01, 10).

Il Register File

E' il banco di lavoro; una memoria interna molto piccola costituita da unità dette registri. La CPU conserva in questa memoria i dati usati più frequentemente.

Molte istruzioni operano su valori contenuti sui registri, anziché indicare il valore si indica quindi l'indirizzo del registro.

Le architetture che lavorano con i registri vengono chiamate load/store, poiché vengono caricati i dati sui registri (load) per poi essere caricate in memoria (store).

In MIPS un registro ha la stessa dimensione di una parola di memoria (32 bit).

Il Register File del MIPS contiene 32 registri e per indicarne l'indirizzo abbiamo bisogno di 5 bit.

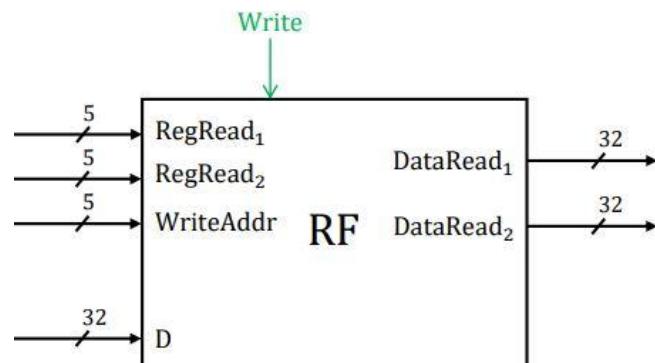
Il Register File può essere usato sia in lettura che in scrittura (anche nello stesso ciclo di clock). Proprietà : si possono leggere due registri contemporaneamente.

Input:

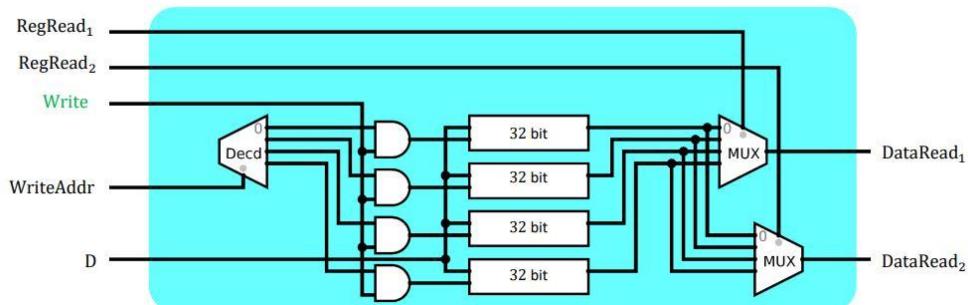
- Write = segnale di controllo (1 bit) che vale 1 se vogliamo scrivere in un registro ;
- RegRead₁ e RegRead₂ = sono due indirizzi da 5 bit da andare a leggere ;
- WriteAddr = indirizzo a 5 bit del registro in cui scrivere (se in modalità scrittura) ;
- D = dato (32 bit) da scrivere nel registro indirizzato da WriteAddr .

Output:

- DataRead₁ e DataRead₂ = i valori (32 bit) contenuti nei registri indirizzati da RegRead₁ e RegRead₂ .

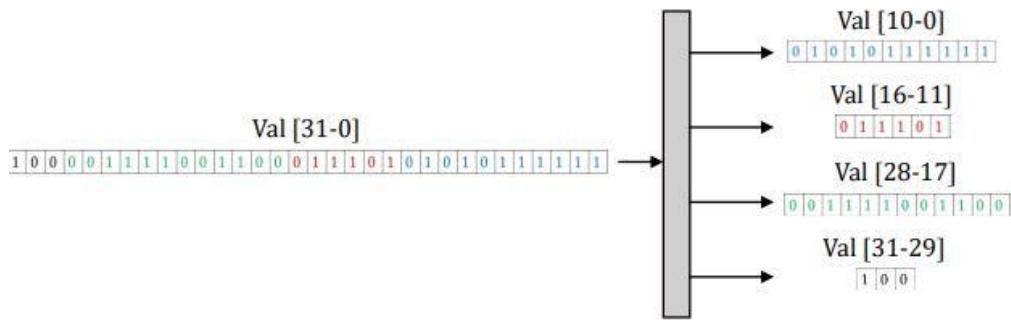


Esempio di Register File semplificato

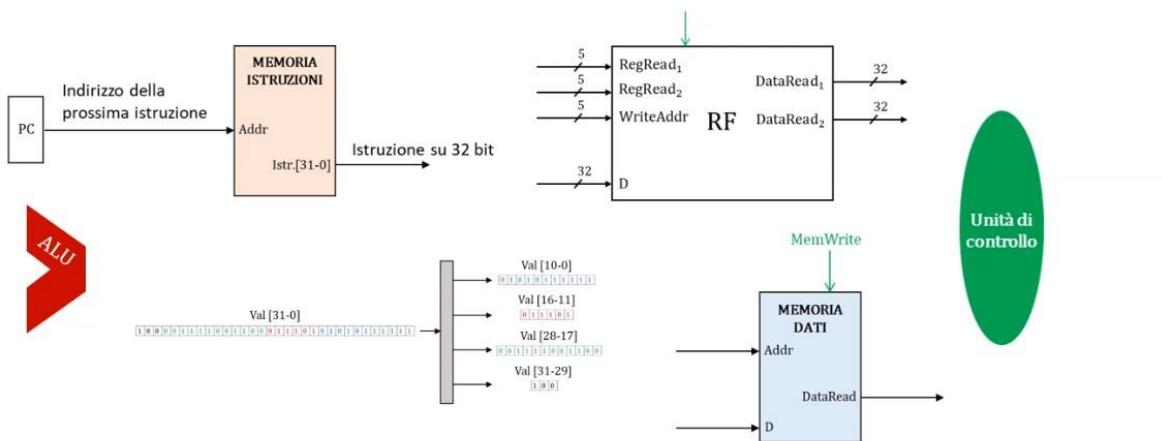


Un componente utile: Splitter

Esso ci permette di separare un segnale da più bit in sottogruppi di bit. Lo usiamo principalmente per estrarre i campi dai 32 bit.



Abbiamo tutti gli elementi, manca solo l'unità di controllo che decide i bit di controllo.



Istruzioni con formato R

Il formato R è per quelle istruzioni che lavorano su valori sei registri (quindi usano esclusivamente i registri):

- add rd, rs, rt = carica in rd la somma tra rs e rt ;
- sub rd, rs, rt = carica in rd la differenza tra rs e rt ;
- and rd, rs, rt = carica in rd AND tra i bit di rs e rt ;
- or rd, rs, rt = carica in rd OR tra i bit di rs e rt;
- slt rd, rs, rt = carica nel registro rd il valore 1 se il valore in rs è strettamente minore del valore in rt, altrimenti carica in rd 0.

Ciascuna delle seguenti istruzioni viene scritta così:

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

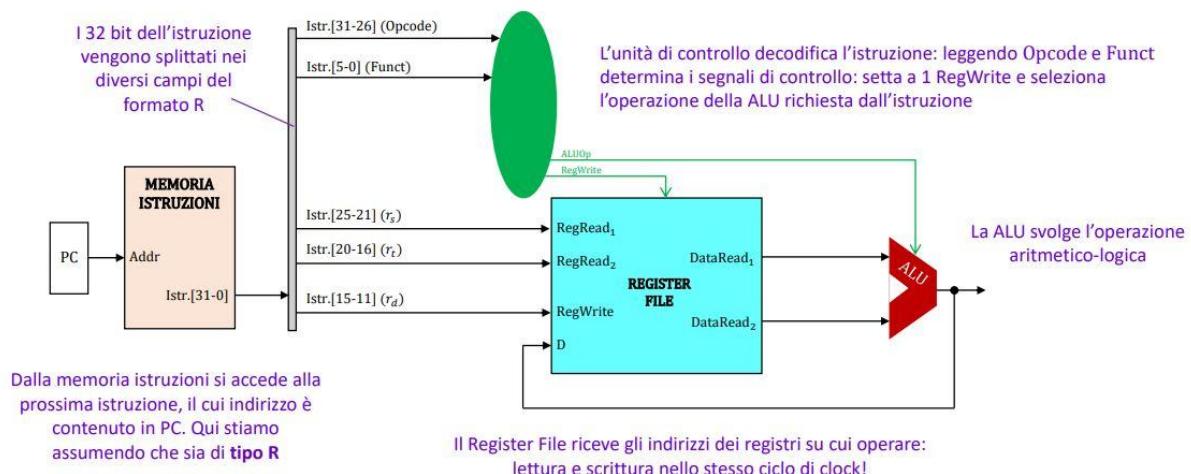
- OPCODE = codice identificativo dell'operazione, del tipo di operazione, infatti per le istruzioni di tipo R abbiamo il codice "000000" ;

- rs = source register, 5 bit di numero di registro, il primo indirizzo dal quale andare a prendere le informazioni ;
- rt = next source register, 5 bit di numero di registro, il secondo indirizzo dal quale andare a prendere le informazioni ;
- rd = destination register, 5 bit di numero di registro, registro nel quale scrivere il risultato ;
- shamt = shift amount, per le istruzioni di shift (per adesso non lo vedremo) ;
- funct = function, indica quale operazione va eseguita.

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

Data path

Costruzione del data path per le istruzioni di tipo R



Istruzioni con formato I

Istruzioni che utilizzano sia costanti, sia variabili. Questo operando costante (immediato) occupa 16 bit

Le istruzioni sono:

- **lw rt, Offset(rs)** = loadword, carica in **rt** la parola di memoria con indirizzo **rs** + il valore di **Offset** ;
- **sw rt, Offset(rs)** = storeword, trasferisce il contenuto di **rt** nella parola di memoria con indirizzo **rs** + valore di **Offset** ;

- beq rs, rt Offset = branch if equal, se il contenuto del registro rs è uguale a quello di rt è uguale a quello di rt salta all'istruzione PC+valore di Offset, altrimenti procede con PC+4.

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

L'OPCODE di loadword è 100011

Il registro di destinazione è ora rt.

Per calcolare la somma di servizio useremo la ALU.

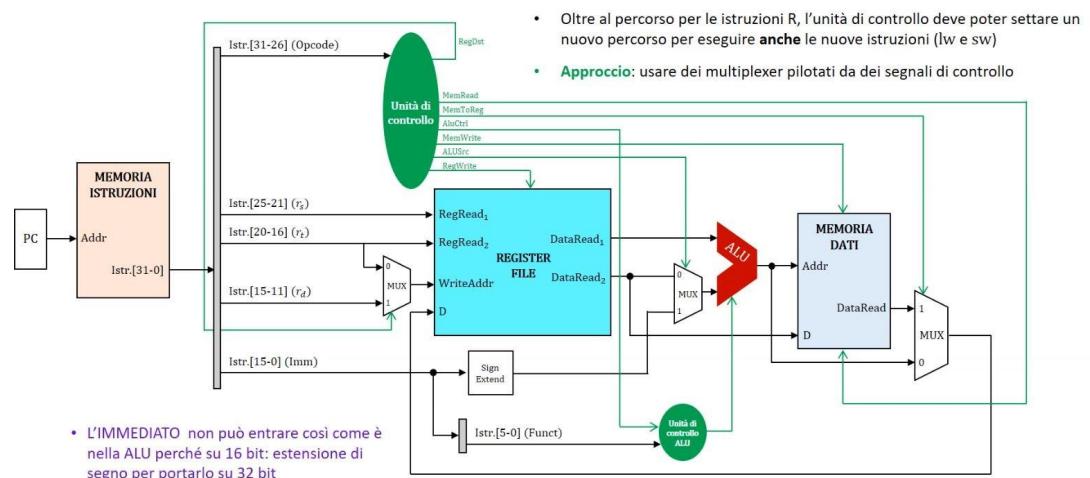
!Laboratorio del 14 Dicembre 2021

Lezione del 16 Dicembre 2021

Data path per istruzioni in formato I

Dobbiamo consentire all'unità di controllo di settare i percorsi sul data path. Oltre alle istruzioni R, dobbiamo percorrere le nuove istruzioni I (sw e lw). L'approccio che utilizziamo è attraverso dei multiplexer.

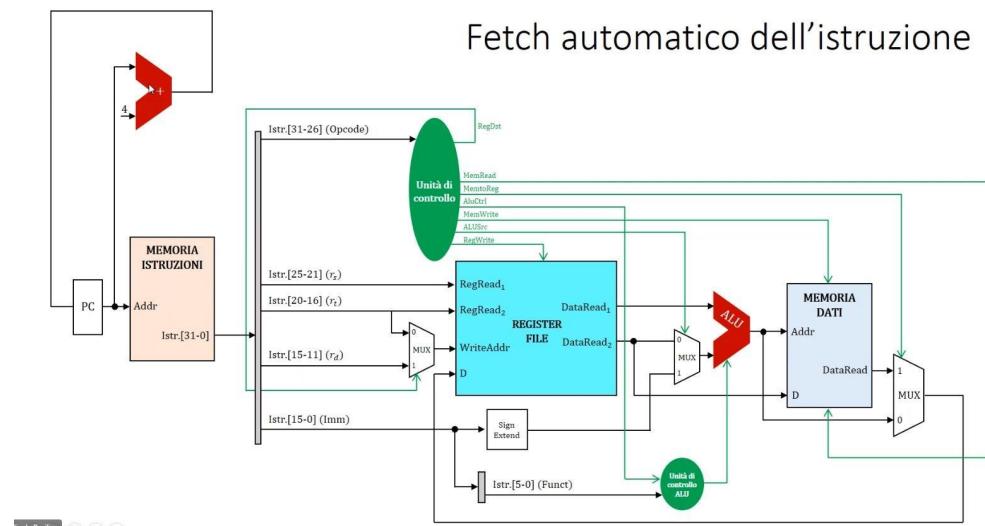
Le istruzioni di tipo R passano per gli 0 dei multiplexer, mentre se passano a 1 abbiamo un'istruzione lw di tipo I (tranne per il primo MUX). Per la sw collegiamo DataRead₂ a D della memoria dati (quindi per la scrittura). ALUsource sarà 0 per le istruzioni R mentre 1 per l'immediato. MemWrite per scrivere in memoria dati mentre MemToReg decide quale valore verrà scritto nei registri. RegDest decide la destinazione nella scrittura. MemRead decide se dobbiamo leggere dalla memoria.



Possiamo dotare la nostra CPU di una modifica per far eseguire la prossima istruzione modificando ogni volta il PC aggiungendogli 4.

Problema del Fetch automatico

Per farlo si estende il data path per far sì che ad ogni ciclo avremo $PC = PC + 4$. Questo è il normale flusso di esecuzione, il quale può essere alterato con le istruzioni di salto. Adesso doteremo la nostra CPU la funzionalità di proseguire con $PC + 4$.



Abbiamo aggiunto un sommatore per fare $PC + 4$, aggiungendo un sommatore non lo facciamo tramite la ALU stessa per riservarla alle istruzioni. Ad ogni ciclo di clock il PC viene sommato a 4.

Alterare il flusso.

I salti provocano un'alterazione del flusso. Quindi quello che ci chiediamo è che aspetto hanno i salti all'interno del data path?
Avere qualcosa di diverso da $PC + 4$.

Questo procedimento si può fare in due modi:

- 1) Salto condizionato, dipendente da una condizione vera a runtime;
- 2) Salto non condizionato, il quale avviene sempre .

In entrambi i casi il datapath deve poter verificare la condizione e deve poter verificare la condizione e poter cambiare il flusso.

Salto condizionato

Branch on equal, beq rs, rt Offset

OPCODE = 000100

Se rs e rt sono diversi abbiamo $PC + 4$, mentre se sono uguali bisogna fare un salto.

Usiamo la ALU attraverso il bit di zero (quando il risultato della sottrazione è uguale a 0), per poi far indicare il salto dall'Offset.

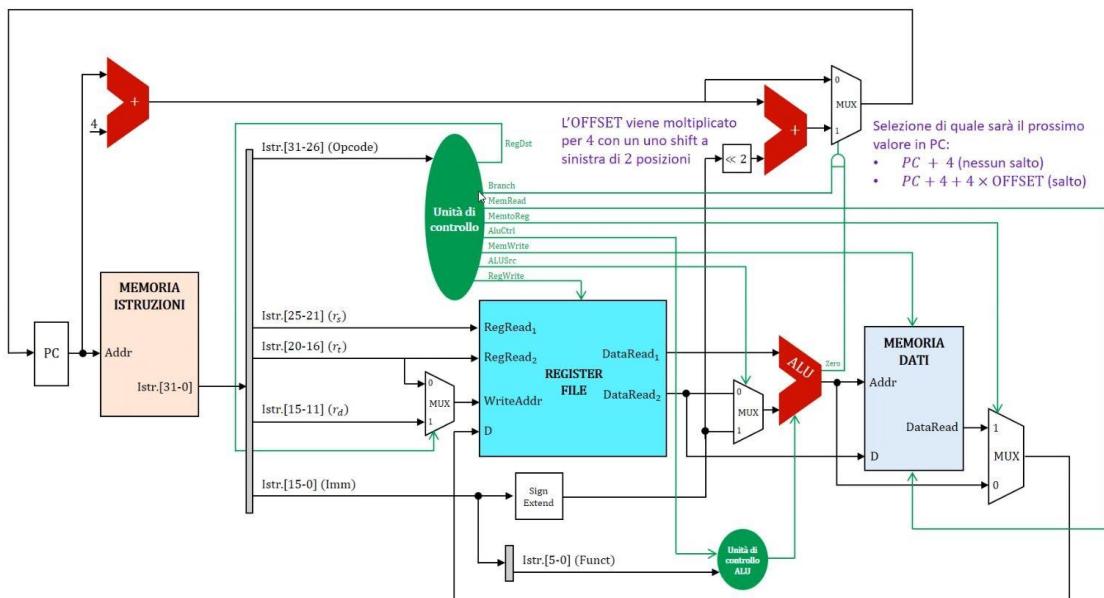
Modalità di indirizzamento (PC relative) interpretato come un intero in complemento a 2 come un Offset di numero di istruzioni (non di byte). Quindi infine il PC sarà uguale a $(PC+4)+4 \times \text{Offset}$.

Con questo avremo:

- Salto all'indietro più ampio = $(PC+4)+4 \times (-2^{15})$;
- Salto in avanti più ampio = $(PC+4)+4 \times (2^{15}-1)$.

La moltiplicazione per 4 avviene con uno shift di 2 posizioni \ll .

Costruzione del data path



Istruzioni con formato J

OPCODE	PSEUDO-INDIRIZZO
6 bit	26 bit

j Pseudo-Indirizzo, scrive nel PC un nuovo indirizzo ottenuto applicando la modalità di indirizzamento pseudo-diretta sui 26 bit del campo Pseudo-Indirizzo.

OPCODE 000010

Modalità di indirizzamento:

- Aggiungere due zero a sinistra per poi eseguire lo shift a sinistra di due posizioni \ll ;
- Aggiungere a sinistra i 4 bit più significativi del PC .

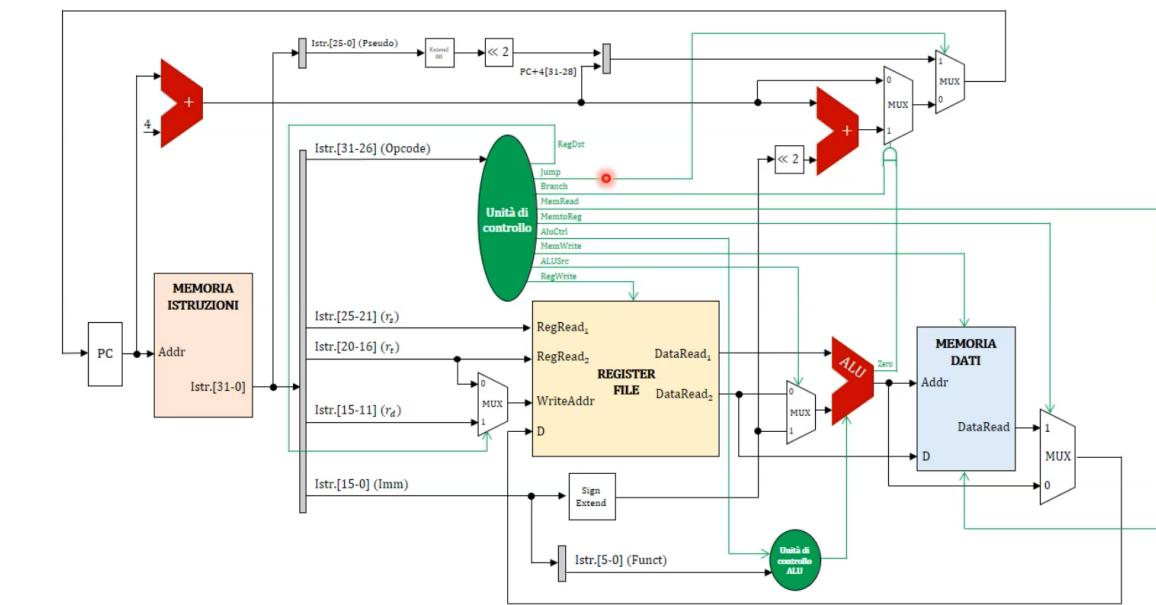
Il risultato è un indirizzo su 32 bit da sovrascrivere nel PC.

Lezione del 20 Dicembre 2021

Nel caso dovessimo fare un salto molto ampio con le istruzioni di tipo J? Concateniamo più istruzioni J, e nel caso finissimo lo spazio nel Pc (limite hardware) abbiamo delle componenti software (sistema operativo) che ci permettono di avere uno pseudo spazio infinito.

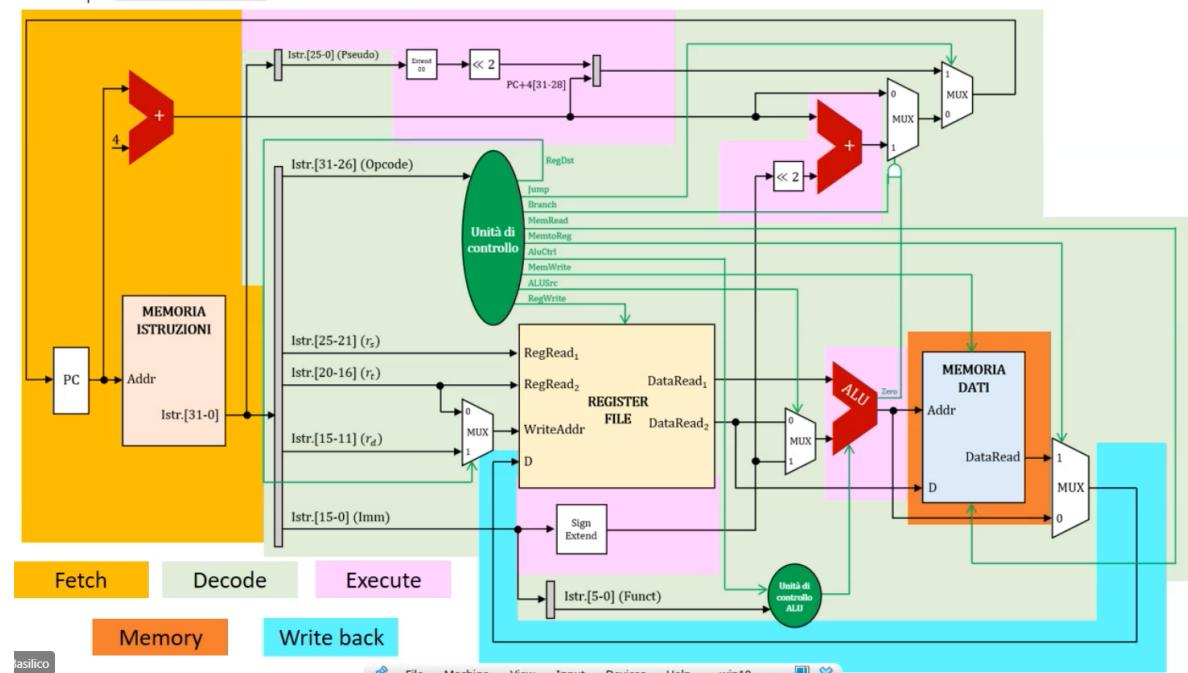
Data path per le istruzioni di tipo J

Istruzioni con formato J



Fasi della CPU illustrate e evidenziate

Le cinque fasi di una istruzione



Come sono fatte le unità di controllo e l'unità di controllo ALU?

Prendiamo per esempio le istruzioni J, con OPCODE 000010. In questo caso l'UC deve far sì che a questo OPCODE corrisponda il segnale di Jump=1, altrimenti Jump=0. Il controllo di questi datapath viene eseguito in maniera combinatoria.

Costruzione dell'UC

Riepilogo dei segnali

- 1) RegWrite = Attiva la scrittura nel Register File, il dato D viene scritto nel registro WriteAddr ;
- 2) MemRead = Attiva la lettura in memoria ;
- 3) MemWrite = Attiva la scrittura in memoria;
- 4) ALUSrc = Quando è 0 il secondo operando della ALU viene dal Register File con uscita DataRead₂ mentre quando è uguale a 1 il secondo operando della ALU è l'estensione su 32 bit dell'Immediato (bit 15-0) ;
- 5) MemToReg = Quando è uguale a 0 il dato D da scrivere nel Register File proviene dall'uscita dalla ALU mentre quando è uguale a 1 il dato D da scrivere nel Register File proviene dal DataRead in memoria ;
- 6) RegDst = Quando è uguale a 0 l'indirizzo del registro di destinazione è estratto dal campo rt (bit 20-16) mentre quando è 1 L'indirizzo del registro di destinazione è estratto dal campo rd dell'istruzione (bit 15-11) ;
- 7) Branch AND Zero = Quando è uguale a 0 L'indirizzo da scrivere in PC è PC+4 (a meno che Jump non sia posto a 1) mentre quando è 1 L'indirizzo da scrivere nel PC è PC+4 + Immediato su 32 bit e moltiplicato per 4 (bit 15-0) ;
- 8) Jump = Quando è uguale a 0 L'indirizzo da scrivere in PC è PC+4 (a meno che Branch AND Zero non sia posto a 1) mentre quando è 1 l'indirizzo da scrivere in PC è ottenuto aggiungendo due zeri a sinistra, shiftando a sinistra e aggiungendo i 4 bit più significativi del PC precedente .

Unità di controllo della ALU

ALUCtrl = comunica con l'UC della ALU

ALUOp = decide l'operazione della ALU

ALUCtrl	ALUOp
Tipo R	Operazione indicata da Funct
Accesso a memoria (lw e sw)	La ALU fa una somma
Branch On Equal	La ALU fa una differenza

Iniziamo scegliendo una codifica per ALUOp

OPCODE	ALUCtrl
000000 (Tipo R)	10 Aritmetico-Logica
100011 (lw)	00 Accesso a memoria
101011 (sw)	00 Accesso a memoria
000100 (beq)	01 Branch

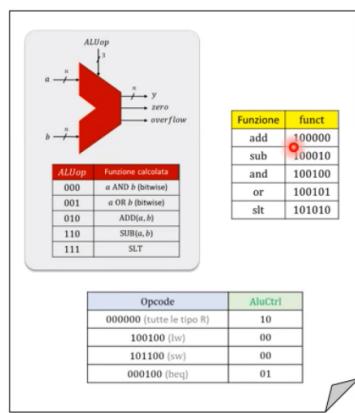
Convenzione scelta arbitrariamente

Con 6 bit di input abbiamo una tabella con 2^6 righe. Ma in tabella abbiamo 5 righe, infatti in MIPS abbiamo un OPCODE di 6 bit, ma dato che la nostra CPU è semplificata noi abbiamo:

- 5 istruzioni di tipo R ;
- 2 istruzioni di accesso a memoria (lw e sw) ;
- 2 istruzioni di salto (beq e istruzioni di tipo J) .

Quindi assumiamo che OPCODE diversi non esistano.

Unità di controllo ALU



lw	0	0	funct						ALUOp ₂	ALUOp ₁	ALUOp ₀
			f ₅	f ₄	f ₃	f ₂	f ₁	f ₀			
sw	0	0	x	x	x	x	x	x	0	1	0
beq	0	1	x	x	x	x	x	x	1	1	0
add	1	0	1	0	0	0	0	0	0	1	0
sub	1	0	1	0	0	0	1	0	1	1	0
and	1	0	1	0	0	1	0	0	0	0	0
or	1	0	1	0	0	1	0	1	0	0	1
slt	1	0	1	0	1	0	1	0	1	1	1

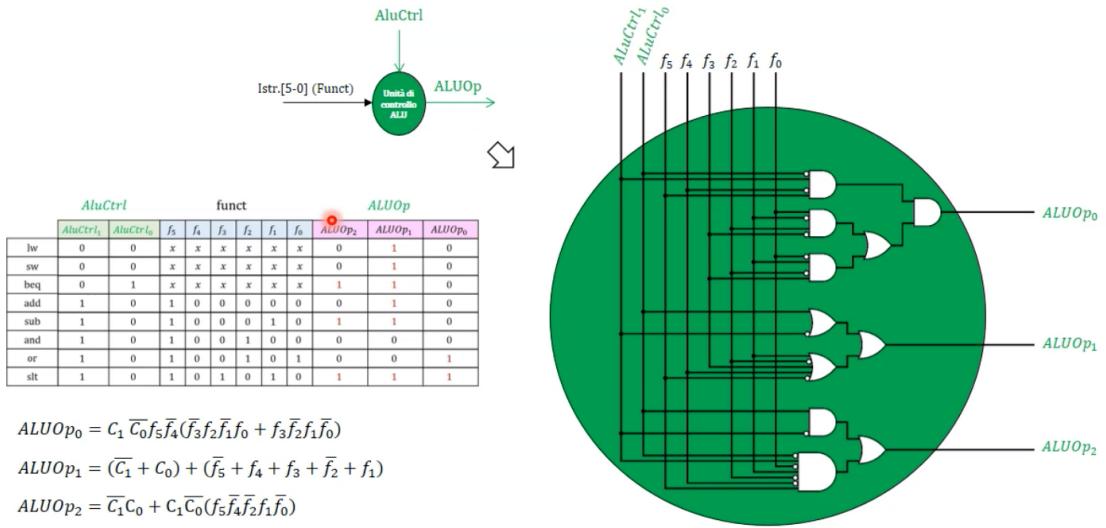
$$AluCtrl_i \rightarrow C_i$$

$$ALUOp_0 = C_1 \bar{C}_0 f_5 \bar{f}_4 (\bar{f}_3 f_2 \bar{f}_1 f_0 + f_3 \bar{f}_2 f_1 \bar{f}_0)$$

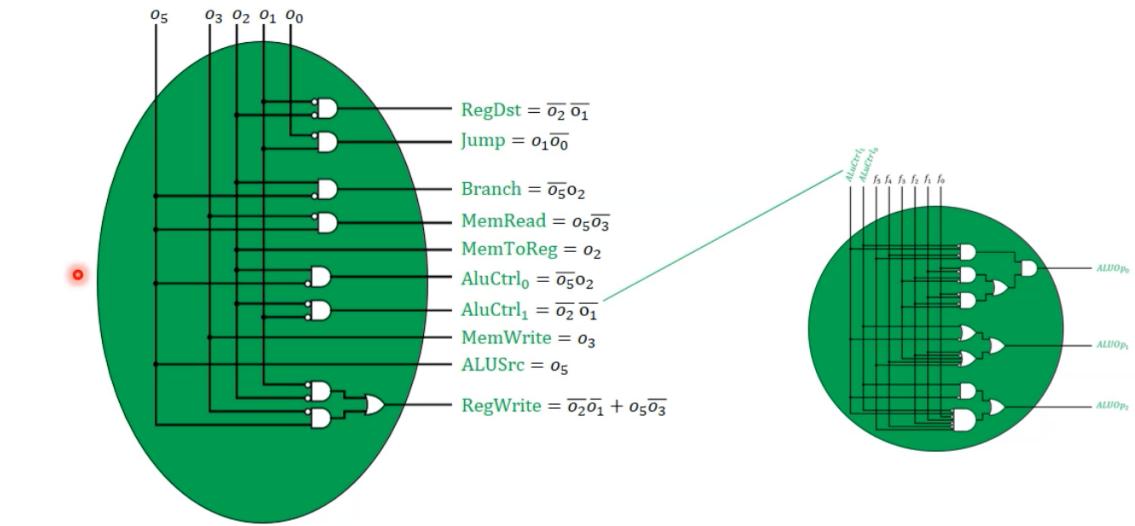
$$ALUOp_1 = (\bar{C}_1 + C_0) + (\bar{f}_5 + f_4 + f_3 + \bar{f}_2 + f_1)$$

$$ALUOp_2 = \bar{C}_1 C_0 + C_1 \bar{C}_0 (f_5 \bar{f}_4 \bar{f}_2 f_1 \bar{f}_0)$$

Unità di controllo ALU



Unità di controllo finale



!Laboratorio del 21 Dicembre 2021