

Architettura degli Elaboratori
uniVR - Dipartimento di Informatica
– I Semestre –

Amos Lo Verde

29 luglio 2023

Indice

Prefezione	1
1 Introduzione	3
1.1 Algoritmi di risoluzione	3
1.2 Sistema digitale	4
2 Sistemi di codifica	7
2.1 Numeri interi assoluti	9
2.2 Numeri interi relativi	10
2.2.1 Codifica a modulo + segno	10
2.2.2 Codifica complemento a 2	11
2.3 Numeri razionali	13
2.3.1 Codifica in virgola fissa	13
2.3.2 Codifica in virgola mobile	14
3 Algebra di commutazione	19
3.1 Identità dell'algebra booleana	20
3.1.1 Teoremi fondamentali sulle identità	20
3.2 Porte logiche	21
3.2.1 Transistor di tipo N	23
3.2.2 Transistor di tipo P	23
3.2.3 Porte logiche elementari costruite per composizione	23
3.3 Rappresentazione di funzioni booleane	26
3.3.1 1 ^a forma canonica (somma di prodotti)	29
3.3.2 2 ^a forma canonica (prodotto di somme)	29
3.4 Tecniche di ottimizzazioni	30
3.4.1 Concetti preliminari per l'ottimizzazione	31
3.4.2 Mappa di Karnaugh	33
3.5 Metodo di Quine-McCluskey	35
3.5.1 Funzioni completamente specificate	35
3.5.2 Funzioni parzialmente specificate	40
3.5.3 Metodo per ipotesi	44

4 Reti combinatorie	45
4.1 Sintesi a 2 livelli	45
4.1.1 Circuiti PLA, FPGA, ASIC	46
4.1.2 Problemi e vantaggi della sintesi a 2 livelli	48
4.2 Sintesi a N livelli	50
4.2.1 Modello Network	52
4.2.2 Technology Mapping	55
5 Da reti combinatorie a circuiti sequenziali	61
5.1 Tempo	61
5.2 Memoria	62
5.2.1 Registro	63
5.3 Macchina a stati finiti (FSM)	66
5.3.1 Rappresentazioni della FSM	66
5.3.2 Modello di Huffman	68
5.3.3 Flusso di sintesi di FSM	68
5.4 Minimizzazione degli stati	69
5.4.1 Minimizzazione di macchine parzialmente specificate	72
5.5 Codifica degli stati	75
5.5.1 Grafo delle adiacenze	76
6 DATAPATH	81
6.1 Unità funzionali	81
6.2 Sintesi ad alto livello	84
6.3 Modello FSMD	85
6.4 FMSD da un algoritmo	91
7 Laboratorio SIS	97
7.1 Descrizione del primo componente	97
7.1.1 Descrizione funzioni booleane in SIS	99
7.2 Ottimizzazione esatta dei circuiti combinatori	101
7.2.1 Minimizzazione dei mintermini	102
7.2.2 Gestione del Don't Care Set	102
7.2.3 Minimizzazione di circuiti multi-livello	104
7.3 Technology Mapping su SIS	105
7.4 Circuiti Sequenziali e Macchine a Stati Finiti	106
7.4.1 Circuiti sequenziali in SIS	106
7.5 DATAPATH	109
7.5.1 Unità funzionali del DATAPATH	109
7.5.2 Modellazione dei componenti	110
7.5.3 Modellazione di una FSMD in SIS	113

Prefazione

Questa dispensa è stata scritta mettendo assieme le nozioni prese dalle: lezioni seguite in presenza (A.A. 2020/2021), videolezioni caricate sulla piattaforma Moodle e dal testo di riferimento originario consigliato per il corso.

L'utilizzo di questo materiale non deve sostituire la frequentazione del corso, ma solo supportarlo a fine didattico.

È possibile condividere gratuitamente questa dispensa. Inoltre si ricorda che tutti i materiali sono sempre disponibili al canale Telegram <https://t.me/univrinfo>.

Crediti

Professori del corso (A.A. 2020/2021): [Franco Fummi](#), [Luca Geretti](#), [Davide Quaglia](#), [Stefano Centomo](#)

Testo di riferimento (A.A 2020/2021): [*Progettazione digitale \[seconda edizione\]* \(Franco Fummi, Mariagiovanna Sami, Cristina Silvano, Michele Lora\)](#)

Testo di riferimento aggiornato: [*Progettazione digitale \[terza edizione\]* \(Franco Fummi, Michele Lora, Mariagiovanna Sami, Cristina Silvano\)](#)

Introduzione

1.1 Algoritmi di risoluzione

Nel XX secolo Alan Turing dimostra che non tutti i problemi sono risolvibili in maniera automatica, definendo così a livello teorico le **categorie di problemi**. Alcuni problemi di differenti categorie sono risolvibili mediante un **algoritmo di risoluzione**, ossia una serie di passi elementari eseguiti da un esecutore automatico. Un algoritmo di risoluzione può essere realizzato in due modi:

- **Sintesi logica**: a partire da un algoritmo si produce direttamente una realizzazione fisica HW (*hardware*), quindi elettronica, dell'algoritmo stesso.
- **Programmazione**: a partire da un algoritmo si produce un SW (*software*) tramite linguaggi di alto livello.

Sintesi logica

Tradurre un algoritmo in HW è un processo lungo, complesso, molto efficiente, dispendioso, limitato a un singolo uso specifico.

Per rendere più flessibile l'HW si aggiunge il SW così da renderlo programmabile. Nascono i **sistemi embedded** che possiedono molto HW e una piccola parte SW; alcuni esempi di sistemi *embedded* sono: forno a microonde, ABS auto.

I sistemi opposti ai sistemi *embedded*, cioè con maggior SW e una piccola parte di HW, sono chiamati **sistemi general purpose**. Quando viene creato un prodotto *general purpose* il produttore non sa l'uso che ne verrà fatto, per esempio l' utilizzo specifico di un PC dipende dal SW che determina se lo scopo della macchina è riprodurre musica, prendere appunti, oppure creare presentazioni.

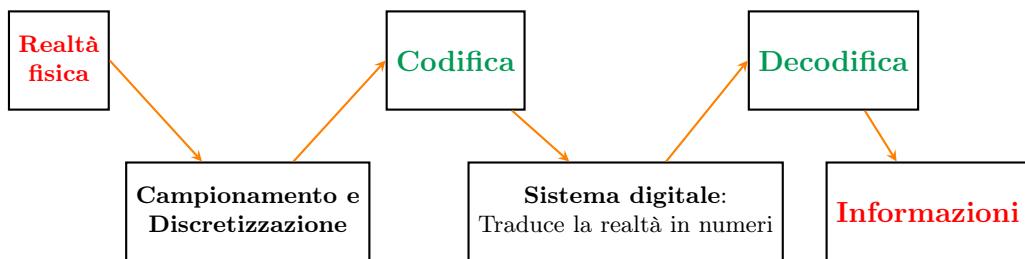
Assembly è il linguaggio di programmazione più vicino all'HW (di basso livello), che permette di programmare direttamente tutte le funzionalità dell'HW. Ogni istruzione corrisponde a una funzionalità che l'HW mette a disposizione.

Programmazione

Questa modalità permette di lavorare sui *software* presenti all'interno di sistemi *general purpose*, dove è presente una maggiore concentrazione SW.

Basti pensare alle applicazioni di un sistema operativo che vengono realizzate tramite **linguaggi di alto livello**. Tali linguaggi permettono di tradurre un algoritmo in SW e rispetto alla **sintesi logica** è un processo più semplice: con maggiori livelli d'astrazione, meno dispendioso, ma maggiore difficoltà nella gestione specifica delle risorse.

1.2 Sistema digitale



Definizione 1.2.1: Sistema digitale

Manipola in modo numerico i segnali numerici di ingresso per produrre segnali di uscita anch'essi numerici.

Realtà fisica

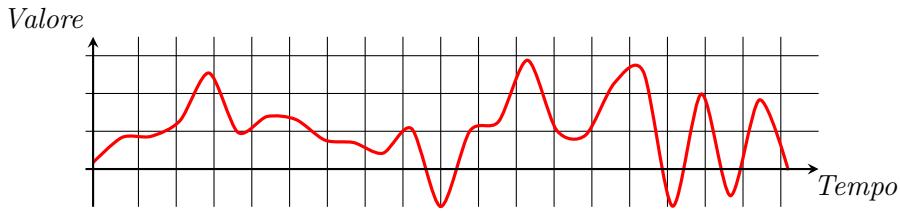
La realtà non è formata da un insieme di numeri e per essere manipolata dal **sistema digitale** bisogna prima convertirla e codificarla.

Per trasformare un numero di campioni infinito (realtà continua) in un numero di campioni finito, è necessario **campionare** la realtà a intervalli di tempo. Tuttavia ciò non basta, perché ogni campione è ancora un numero reale che richiede infinite cifre, allora bisogna rappresentare i campioni con una determinata precisione finita detta **discretizzazione**. I campioni vengono dunque approssimati ai campioni discreti più vicini: la funzione della curva si trasforma in una **spezzata poligonale**.

La **spezzata poligonale** rappresenta la funzione reale con delle approssimazioni. Il vantaggio è che permette di avere in un certo intervallo di tempo un numero finito di campioni, ognuno dei quali è descritto con dei numeri che hanno un numero finito di cifre. I campioni diventano da **reali a razionali**.

Teorema 1.2.1: Shannon

Deciso il grado d'errore da voler compiere, esistono una precisa frequenza di campionamento e un intervallo di discretizzazione che garantiscono quell'errore.



In questo esempio grafico viene riportata una generica curva del suono:

- le linee verticali rappresentano il campionamento;
- le linee orizzontali rappresentano la discretizzazione.

Codifica

Ogni campione deve essere assegnato a un codice numerico affinché il **sistema digitale** possa manipolarlo.

Osservazione

Il sistema digitale non può operare direttamente sulla realtà; ha bisogno di lavorare con dei dati rappresentati da numeri finiti. Per convertire i dati in codici numerici bisogna scegliere una base numerica tramite la quale il sistema digitale deve lavorare. Per questioni pratiche e tecnologiche negli anni '40 fu scelta la base 2 poiché sono disponibili solo due cifre: 1 (*true*) e 0 (*false*).

In conclusione il sistema digitale riceve in ingresso dei ***bit*** (*Binary digIT*) e produce in uscita altri ***bit***, che tramite un **sistema di decodifica** diventeranno informazioni intellegibili.

Sistemi di codifica

Una volta ottenute le informazioni dal mondo reale, tramite il campionamento e la discretizzazione, occorre codificarle per essere manipolate nel **sistema digitale**.

Ogni sistema digitale lavora in base binaria, per cui entrano N bit ed escono M bit. Questi bit in uscita devono essere codificati per realizzare delle informazioni. Esistono due tipologie di informazioni:

- **Informazioni intellegibili:** sono già chiare agli esseri umani, come un testo scritto.
- **Informazioni non intellegibili:** hanno bisogno di macchine per essere riprodotte, come le casse per l'audio.

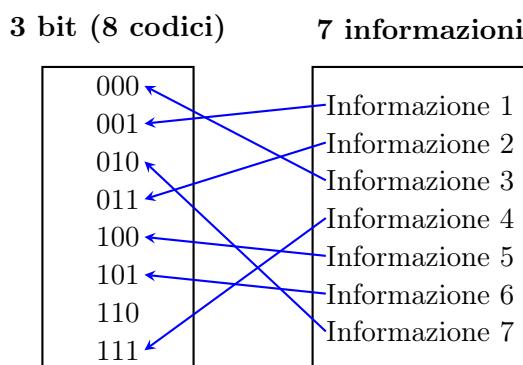
Codifica dell'informazione non numerica

Ogni **informazione** deve avere un **codice univoco** in modo tale che il sistema digitale non sbagli a manipolare le informazioni. In generale date M informazioni si ricavano $n = \lceil \log_2(M) \rceil$ codici disponibili per codificare tali informazioni.

Esempio

Vengono fornite $M = 7$ informazioni, di conseguenza si ricavano:

- $n = \lceil \log_2(7) \rceil = 3$ bit.
- $2^3 = 8$ codici disponibili per la codifica.



A ogni informazione viene assegnato un codice qualsiasi grazie alla libertà di codifica. Per le **informazioni standard**, che tutti i sistemi digitali manipolano, sono state definite le seguenti codifiche:

- **Testo:** la codifica *standard* del testo risale agli anni '70 e si basa sull'**ASCII**: nasce al fine di codificare i linguaggi che derivano dal latino o che usano l'alfabeto latino. L'ASCII attuale è a 8 bit, cioè 1B, per cui si hanno a disposizione 256 codici di codifica: dallo 0 al 255. I *bit* a destra del codice sono detti **meno significativi**, mentre più si va a sinistra e più diventano significativi: il primo *bit* a sinistra è detto **bit più significativo**.

1 bit	7 bit
-------	-------

- I primi 7 bit definiscono 128 codici (da 0 a 127), che nella codifica ASCII sono uguali in tutte le lingue: sono le lettere di base dell'alfabeto.
- A ogni simbolo corrisponde un numero, per esempio: se si preme sulla tastiera il carattere Q, allora si chiude l'interruttore elettrico che il circuito riconosce come carattere Q e gli associa il suo valore numerico.
- Il *bit* restante (quello più significativo) definisce i codici dal 128 fino al 255, i quali cambiano a seconda della lingua.

- **Numeri:**

- Insieme \mathbb{N} (interi assoluti); vedi Sezione 2.1
- Insieme \mathbb{Z} (interi relativi); vedi Sezione 2.2
- Insieme \mathbb{Q} (razionali relativi); vedi Sezione 2.3: hanno un'elevata precisione in grado di colmare la mancanza dei reali. Si suddividono in:
 - * virgola fissa;
 - * virgola mobile.
- Insieme \mathbb{R} : non è rappresentabile, poiché richiederebbe infiniti valori e di conseguenza infinite risorse.

- **Multimediale** (alcuni esempi):

- Audio:
 - * CD
 - * MP3
- Video:
 - * MPEG4
 - * MOV

2.1 Numeri interi assoluti

Intervallo : $0 \leq N \leq 2^n - 1$

I **numeri interi assoluti** rappresentano solo valori interi dallo 0 fino a $2^n - 1$, dove n è il numero di *bit* disponibili per la codifica.

Esempio

Si deve convertire il numero 57_{10} in base binaria.

$$n = \lceil \log_2(57) \rceil = 6 \text{ bit (minimi)}$$

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1 = 63 \text{ (codici massimi)}$$

Si eseguono i seguenti passaggi:

1. Si sottraggono le potenze di 2 partendo da $n - 1$.
 - Se la potenza 2^i è minore o uguale del numero, allora la si moltiplica per 1.
 - Se la potenza 2^i è maggiore del numero, allora la si moltiplica per 0.
2. Le sottrazioni continuano fino a quando si giunge a 0.

$$57_{10} - 1 \cdot 2^5 = 25_{10} - 1 \cdot 2^4 = 9_{10} - 1 \cdot 2^3 = 1_{10} - 0 \cdot 2^2 = 1_{10} - 0 \cdot 2^1 = 1_{10} - 1 \cdot 2^0 = 0.$$

Il risultato finale corrisponde alle cifre evidenziate in azzurro ottenute in sequenza, ossia 111001_2 .

Esempio

Si deve convertire il numero 14_{10} in base binaria:

$$n = \lceil \log_2(14) \rceil = 4 \text{ bit (minimi)}$$

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1 = 15 \text{ (codici massimi)}$$

$$14_{10} - 1 \cdot 2^3 = 6_{10} - 1 \cdot 2^2 = 2_{10} - 1 \cdot 2^1 = 0_{10} - 0 \cdot 2^0 = 0.$$

Il risultato finale corrisponde alle cifre evidenziate in azzurro ottenute in sequenza, ossia 1110_2 .

Questa procedura di codifica da *base decimale* a *base binaria* prende il nome di **codifica a modulo**.

Osservazione

Si esegue una somma tra due numeri decimali con a fianco la medesima somma in base binaria (con 6 bit a disposizione):

$$\begin{array}{r} 57_{10} + \\ 14_{10} = \\ \hline 71_{10} = \end{array} \quad \begin{array}{r} 111001_2 + \\ 001110_2 = \\ \hline 1000111_2 = \end{array}$$

Nell'addizione in base binaria vengono aggiunti davanti al 1110_2 due zero, evidenziati in grassetto, che non alterano l'addizione (accade in ogni base).

Nel risultato dell'addizione binaria l'1 in grassetto (nella posizione più significativa) segnala l'uso del 7° bit, risultando in *overflow*^a dato che il calcolatore nell'esempio lavora su 6 bit, quella somma risulterebbe $000111_2 = 111_2$, perciò errata.

^aIndica il “traboccamento”, cioè se viene superato il limite massimo l'*overflow* è un errore, non perché sia sbagliata la somma, ma perché il risultato non è codificabile con il numero di bit disponibili.

2.2 Numeri interi relativi

La codifica più ovvia a cui pensare per gli **interi relativi** è la **codifica a modulo + segno**. Tuttavia rappresenta varie problematiche per cui è meglio utilizzare la **codifica complemento a 2**.

2.2.1 Codifica a modulo + segno

$$\text{Intervallo: } -2^{n-1} + 1 \leq N \leq 2^{n-1} - 1$$

Il segno viene rappresentato con 1 bit: lo 0 per il più (+) e l'1 per il meno (-).

Il **bit più significativo** rappresenta il segno, mentre i **bit meno significativi** rappresentano il modulo.

1 bit: segno \pm	7 bit: modulo
-----------------------	---------------

Considerando gli esempi visti in precedenza, si hanno le seguenti rappresentazioni:

$$\begin{array}{ll} +57_{10} = 0|111001_2 & +14_{10} = 0|1110_2 \\ -57_{10} = 1|111001_2 & -14_{10} = 1|1110_2 \end{array}$$

Il problema sorge quando si vuole rappresentare il valore 0_{10} , che in binario risulterebbe:

$$+0_{10} = 0|000000_2 \quad -0_{10} = 1|000000_2$$

Le somme che passano dal positivo al negativo e dal negativo al positivo risultano errate.

Questa codifica per le operazioni elementari ha bisogno di tre circuiti (esempio con la sottrazione):

1. deve capire quale sia il modulo più grande;
2. successivamente eseguire la sottrazione;
3. infine riassegnare il segno del modulo più grande.

Diventa troppo dispendioso seguire questo procedimento per le operazioni elementari, non è poi pensabile per le operazioni composte che richiederebbero ancora più risorse. L'obiettivo diventa quello di trovare un modo per effettuare la somma tra due numeri senza dover subire troppi passaggi dispendiosi (complemento a 2).

2.2.2 Codifica complemento a 2

$$\text{Intervallo : } -2^{n-1} \leq N \leq 2^{n-1} - 1$$

Per codificare i numeri interi relativi la codifica modulo + segno non serve. L'obiettivo è trovare una codifica semplice per la somma. L'idea parte con il voler trovare la codifica relativa al -1 , pertanto si tenta di formulare $-1 + 1 = 0$:

Obiettivo	Risultato
$????_2 +$ $0001_2 =$	$1111_2 +$ $0001_2 =$
$0000_2 =$	$0000_2 =$

Si considera $n = 4$ bit, perciò si ha l'intervallo di valori $-2^3 \leq N \leq 2^3 - 1$ (Tabella 2.2.2):

$0_{10} = 0000_2$	$-1_{10} = 1111_2$
$1_{10} = 0001_2$	$-2_{10} = 1110_2$
$2_{10} = 0010_2$	$-3_{10} = 1101_2$
$3_{10} = 0011_2$	$-4_{10} = 1100_2$
$4_{10} = 0100_2$	$-5_{10} = 1011_2$
$5_{10} = 0101_2$	$-6_{10} = 1010_2$
$6_{10} = 0110_2$	$-7_{10} = 1001_2$
$7_{10} = 0111_2$	$-8_{10} = 1000_2$

La verifica dell'*overflow* non si fa guardando il segno finale, ma se supera il limite di esistenza, per esempio: su 4 bit la somma $3 + 5 = 8$ in binario risulta negativa¹

Il **complemento a 2** perciò opera negando il valore positivo e sommando +1:

$$-57_{10} = !(+57_{10}) + 1_{10} = \underbrace{1000110_2}_{-58_{10}} + 0000001_2 = 1000111_2$$

Sottrazione con il complemento a 2

- Se un numero è negativo vanno estesi con gli 1.
- Se un numero è positivo vanno estesi con gli 0.

$$\begin{array}{r} +57_{10} + \\ -7_{10} = \\ \hline +50_{10} = \end{array} \quad \left| \begin{array}{l} 0111001_2 + \\ \mathbf{1111001}_2 = \\ \hline \end{array} \right.$$

Osservazione

Il -7 nel complemento a 2 a 4 bit, come elencato nella Tabella 2.2.2, è 1001_2 , mentre nell'ultimo esempio (dove si lavora a 7 bit) il -7 deve essere esteso ai bit più significativi (evidenziati in grassetto). Se questa procedura di estensione fosse omessa i numeri negativi verrebbero interpretati come positivi.

Differenze di codifiche da base 2 a base 10

Modulo	$1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 =$ $= 64 + 32 + 16 + 8 + 1 = 121_{10}$
Modulo + segno	$1 111001_2 \rightarrow -(1*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0) =$ $= -(1 + 8 + 16 + 32) = -57_{10}$
Complemento a 2	Si applica la regola della negazione + 1: $-(!(1111001) + 1) = -(0000110 + 1) = -(111) = -7_{10}$

¹L'*overflow* nel complemento a 2 si verifica quando una somma di due positivi risulta negativa e quando la somma di due negativi risulta positiva.

2.3 Numeri razionali

Sono composti da una parte intera e una parte frazionaria. Esistono due codifiche:

- **Codifica in virgola fissa** (*fixed point*): presente nella maggior parte dei **sistemi embedded**; ottimale quando è noto il numero più grande e la precisione che si vuole ottenere.
- **Codifica in virgola mobile** (*floating point*): presente nella maggior parte dei **(sistemi purpose)**; ottimale all'uso generale sconosciuto dell'utente.

La codifica in virgola mobile è molto più complessa della codifica in virgola fissa.

2.3.1 Codifica in virgola fissa

Esempio

Si hanno a disposizione 8 bit: 4 bit per la parte intera e 4 bit per la parte frazionaria. Successivamente si considera il seguente numero binario 0111.1011:

$$0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 . 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = 7 + \frac{11}{16} = \frac{123}{16}$$

Gli esponenti per le cifre della parte intera partono da 0 e incrementano verso sinistra, mentre per le cifre della parte decimale partono da -1 e decrementano verso destra.

È di vitale importanza sapere che il cambio di base non preservi la razionalità del numero:

- potrebbe diventare reale e di conseguenza richiedere una rappresentazione infinita;
- è probabile che si debba fare un'approssimazione, sia che diventi reale o che rimanga razionale: nasce il problema di capire se arrotondare per eccesso o difetto.

Mantenendo gli 8 bit e la medesima suddivisione di 4 bit tra parte intera e frazionaria, si vuole sommare $5 + \frac{4}{5}$. Il 5_{10} diventa semplicemente 0101_2 , mentre per la parte frazionaria esistono due casi:

1. Il numero da base 10 a base 2 preserva la razionalità ed è rappresentabile così come viene ottenuto dalla conversione nello spazio a disposizione.
2. Il numero non ha mantenuto la razionalità o non è rappresentabile totalmente nello spazio per la parte frazionaria, perciò bisogna avvicinarsi il più possibile.

Il $\frac{4}{5}$ corrisponde a 0.8_{10} , ma in base binaria non viene preservata la razionalità, perché:

$$0.8 \cdot 2 = \textcolor{red}{1.6} \rightarrow \mathbf{1}$$

$$0.6 \cdot 2 = 1.2 \rightarrow \mathbf{1}$$

$$0.2 \cdot 2 = 0.4 \rightarrow \mathbf{0}$$

$$0.4 \cdot 2 = 0.8 \rightarrow \mathbf{0}$$

$$0.8 \cdot 2 = \textcolor{red}{1.6} \rightarrow \mathbf{1}$$

⋮

Il prodotto non potrà mai arrivare perfettamente a 1. Dunque la sequenza 1100_2 si ripete periodicamente: $0.8_{10} = 0.110011001100\cdots = 0.\overline{1100}$.

Lo spazio a disposizione è di 4 bit e pertanto vengono trattenuti solo i primi 0.1100_2 , i quali però non corrispondono in maniera precisa a 0.8_{10} , bensì a 0.75_{10} .

È possibile fare di meglio grazie alla **codifica per eccesso**, cioè prendere in considerazione gli 1 finché non si supera la soglia negativa:

- dato che $\frac{4}{5} \geq 2^{-1}$ allora: $\frac{4}{5} - (\mathbf{1} \cdot 2^{-1}) = \frac{4}{5} - \frac{1}{2} = \frac{3}{10}$;
- dato che $\frac{3}{10} \geq 2^{-2}$ allora: $\frac{3}{10} - (\mathbf{1} \cdot 2^{-2}) = \frac{3}{10} - \frac{1}{4} = \frac{1}{20}$;
- dato che $\frac{1}{20} \leq 2^{-3}$ allora: $\frac{1}{20} - (\mathbf{0} \cdot 2^{-3}) = \frac{1}{20}$;
- essendo l'ultimo bit a disposizione allora: $\frac{1}{20} - (\mathbf{1} \cdot 2^{-4}) = \frac{1}{20} - \frac{1}{16} = -\frac{1}{80}$.

Con questo metodo la parte frazionaria su 4 bit risulta 0.1101_2 , riuscendo a essere migliore perché genera un errore minore rispetto a 0.1100_2 :

- Errore di 0.1101_2 : $\Delta = \left| -\frac{1}{80} \right| = \frac{1}{80}$.
- Errore di 0.1100_2 : $\Delta = \frac{1}{20}$.

2.3.2 Codifica in virgola mobile

Quando non si sa quanti *bit* dedicare alla parte intera e alla parte frazionaria, si fa uso della **codifica in virgola mobile**².

- Singola precisione 32 bit → *float*.
- Doppia precisione 64 bit → *double*.

Ciò è servito affinché tutti i calcolatori potessero eseguire gli stessi calcoli in virgola mobile: solo a fine anni '90 ogni calcolatore aveva adottato questo *standard*, prima non era possibile farlo per via di circuiti che non tutti potevano permettersi.

²La standardizzazione di questa codifica risale a fine anni '80 e fa riferimento allo *standard IEEE754*.

La struttura del numero in virgola mobile è del tipo: $N = \pm M \cdot B^{\pm e}$. Dove nel dettaglio le sottoparti corrispondono a:

- **M**: è la mantissa;
- **B**: è la base 2;
- **e**: è l'esponente.

Formati *standard* IEE 754-1985

	Precisione			
	Singola	Singola estesa	Doppia	Doppia estesa
Bit mantissa	23	≥ 31	52	≥ 63
e_{\max}	127	≥ 1023	1023	≥ 16383
e_{\min}	-126	≤ -1022	-1022	≤ -16382
Bias	127	/	1023	/
Larghezza esponente	8	≥ 11	11	≥ 15
Larghezza formato	32	≥ 43	64	≥ 79

La base 2 è stata scelta per via delle moltiplicazioni e divisioni tramite lo *shift*:

- **shift a sinistra**: si moltiplica per 2 a ogni spostamento.

$$0110.0100 = 6.25_{10} \rightarrow 1100.1000 = 12.5_{10}$$

- **shift a destra**: si divide per 2 a ogni spostamento.

$$0110.0100 = 6.25_{10} \rightarrow 0011.0010 = 3.125_{10}$$

Emergono tre problematiche:

1. **Normalizzazione**: un numero si può rappresentare in più modi.

Esempio

Il numero 16 può avere le seguenti rappresentazioni tutte equivalenti:

$$N = 16 = 4 * 2^2 = 2 * 2^3 = 8 * 2^1 = 32 * 2^{-1} = \dots$$

Esiste una **pluralità di codici** che rappresentano lo stesso numero, di conseguenza serve trovare una sola e unica forma; se si dovesse controllare ogni volta il *test d'uguaglianza* non si finirebbe più.

2. **Bilanciamento dei bit:** si deve suddividere lo spazio tra mantissa ed esponente:

- **Mantissa:** indica la precisione del numero.

24 bit : 1 bit segno + 23 bit modulo

- **Esponente:** indica la grandezza minima/massima del numero (8 bit).

3. **Codifica della mantissa ed esponente:**

- **Mantissa:** viene codificata in **virgola fissa** ed è divisa in due parti:
 - Parte intera: è codificata in **modulo + segno**, perché lo *shift* risulta esatto per la normalizzazione (lo *shift* viene effettuato solo sul modulo e non sul segno).
 - Parte frazionaria: viene codificata in **virgola fissa**.
- **Esponente:** lo si codifica in **eccesso 127**, la quale è stata creata solo per questo specifico uso: avere lo 0_{10} codificato come una serie di zeri.

1 bit: segno \pm	8 bit: Esponente	23 bit: Mantissa
-----------------------	---------------------	---------------------

Esempio

Si mostra la rappresentazione dello 0_{10} tramite virgola mobile a singola precisione (*float*, 32 bit):

$$0_{10} \rightarrow 0 | 00000000 | 0000000000\dots0$$

In totale sono 32 zeri:

- 23 zeri per la **mantissa**;
- 8 zeri per l'**esponente**;
- 1 zero per il **segno**.

Per ovviare la normalizzazione, questa codifica tiene conto che deve sempre riportare “1 punto qualcosa”. Se il numero binario fosse per esempio 101.0010_2 , allora si effettuerebbero una serie di *shift* verso destra fino a renderlo “1 punto qualcosa” e trattenere solo i *bit* dopo il punto:

$$101.0010_2 \rightarrow 1.010010_2$$

come è possibile vedere, vengono conservati solo i *bit* dopo il punto, mentre il *bit* prima del punto non viene codificato. Viceversa con il numero 0.0001_2 la manipolazione effettuerebbe più *shift* verso sinistra fino a renderlo 1.0000_2 .

Quando avvengono *X shift* a destra si deve poi sommare *X* all'esponente, mentre quando avvengono *Y shift* a sinistra si deve poi sottrarre *Y* all'esponente.

Pertanto con questo *standard* la serie dei 32 zeri descritti prima diventa: $1.000000\dots0_2$; deve per forza essere “1 punto qualcosa”, dove quell’1 non viene codificato e rimane solo la serie di zeri nella mantissa. In questo modo la serie di tutti zeri è definita uguale a 1.

Bisogna trovare un numero tale per cui $1 \cdot 2^x = 0$. Il più piccolo numero ottenibile deriva dall'utilizzo di -127 come esponente, perciò $1 \cdot 2^{-127}$ rappresenta un numero estremamente piccolo, allora per convenzione si dice che: l'intera serie di soli zeri rappresenta lo 0, perché in realtà rappresentano il numero più piccolo rappresentabile, ossia $1 \cdot 2^{-127}$. Per ottenere questa convenzione si pone $000000\dots0 = -127$ e si somma a ogni esponente $+127$ (vedi gli esempi sottostanti).

Esempio

Si deve codificare il numero $N = (5 + \frac{3}{4}) \cdot 2^{-20}$:

- **Mantissa:** [parte intera] . [parte frazionaria] $\rightarrow 101.1100\dots0$.

Occorre effettuare la normalizzazione e pertanto eseguire *shift* verso destra, tenendo conto delle *X* posizioni che andranno sommate all'esponente.

$$101.1100\dots0 \rightarrow 1.\mathbf{011100\dots0} \quad (X = 2)$$

- **Esponente:** si somma $+127$ all'esponente attuale, ossia $-20 + 127 = +107$. Dopotiché bisogna sommare o sottrarre le *X* posizioni effettuate dallo *shift*; in questo caso sono stati fatti 2 *shift* verso destra dividendo ogni volta la mantissa, perciò si somma $+2$: $107 + 2 = 109$.

L'esponente si ricava convertendo il 109_{10} in base binaria: 01101101_2

Il risultato finale in virgola mobile corrisponde a: $N = 0|01101101|011100\dots0$.

Esempio

Si deve decodificare il numero $N = 1|01001010|110000\dots0$:

- **Mantissa:** poiché è nota $110000\dots0$, allora secondo la normalizzazione era $1.110000\dots0$. Si converte in binario questo valore: $2^0 + 2^{-1} + 2^{-2} = (1 + \frac{1}{2} + \frac{1}{4})$. Inoltre davanti si mette il segno: in questo caso negativo dato che il *bit* iniziale è posto a 1.

- **Esponente:** si converte in decimale la serie 01001010_2 e si sottrae 127: $(2 + 8 + 64) - 127 = -53$.

Il risultato finale corrisponde a: $N = -\frac{7}{4} \cdot 2^{-53}$.

Codici speciali in virgola mobile

$+0_{10}$	\rightarrow	0 00000000 00000000...0
-0_{10}	\rightarrow	1 00000000 00000000...0
$+\infty$	\rightarrow	0 11111111 00000000...0
$-\infty$	\rightarrow	1 11111111 00000000...0
NaN	\rightarrow	0 11111111 00001001...0
NaN	\rightarrow	1 11111111 01000100...0

I valori $\pm\infty$ e **NaN** (*Not a Number*) si differenziano soltanto per i *bit* contenuti nella mantissa: nei primi due casi ci sono solo zeri.

Algebra di commutazione

L'elaborazione digitale delle informazioni può essere vista come una **manipolazione di bit**, ossia una trasformazione delle parole di un codice in altre parole valide dello stesso codice. Assumendo che la parola convertita in *bit* sia un numero finito, allora ne consegue che lo siano anche gli insiemi I (*input*) e O (*output*).



Per descrivere dei dispositivi digitali è necessario possedere un modello che permetta di rappresentare insiemi di valori binari e funzioni che li mettano in relazione: il modello utilizzato è l'**algebra di Boole**¹.

Nel dettaglio l'**algebra di commutazione**, tra le possibili algebre booleane, è quella che risulta più utile in questo contesto. Gli elementi contenuti all'interno di questa algebra sono i valori **0** (*falso*) e **1** (*vero*), mentre le operazioni a due valori disponibili sono:

- **AND**: vale 1 solo se applicata a due valori uguali a 1, altrimenti vale 0.
- **OR**: vale 0 solo se applicata a due valori uguali a 0, altrimenti vale 1.

Inoltre dalla presenza di due soli valori (0 e 1), è direttamente derivabile l'operazione **NOT**: vale 1 se applicata al valore 0 e vale 0 se applicata al valore 1.

Le tre operazioni possono essere viste come funzioni elementari:

- $B \times B$: per le operazioni **AND** e **OR**.
- $B \rightarrow B$: per l'operazione **NOT**.

La funzione $B \times B$ (ovvero B^2) è l'insieme di 4 elementi composto da tutte le coppie di valori di B: 00, 01, 10, 11. Su questi valori è possibile costruire le tabelle di verità applicando le tre operazioni appena discusse.

¹George Boole ridusse la logica in una semplice algebra, incorporando quindi la logica nella matematica.

Coppie	AND	OR	NOT
00	0	0	$0 \rightarrow 1$
01	0	1	$1 \rightarrow 0$
10	0	1	
11	1	1	

Il problema da affrontare è quello di modellare un dispositivo digitale mediante una relazione tra i valori binari dell'insieme di ingresso I e i valori binari dell'insieme di uscita O . Questi insiemi vengono descritti dagli spazi B^n e B^m , quali vengono messi in relazione mediante m funzioni di commutazione $f(B^n) = B$. A ogni funzione corrisponde una tabella di verità e infinite espressioni realizzabili mediante circuiti elettronici.

Simboli algebrici

Le tre operazioni logiche vengono rappresentate secondo i propri simboli algebrici:

- **AND** (\cdot): $z = \text{AND}(x, y) = x \cdot y$.
- **OR** (+): $z = \text{OR}(x, y) = x + y$.
- **NOT** (\bar{x}): $y = \text{NOT}(x) = \bar{x}$.

3.1 Identità dell'algebra booleana

Gli **operatori elementari** dell'algebra booleana sono caratterizzati da una serie di identità. Per tutte vale il principio di dualità tra l'operatore AND e OR.

Nella seguente tabella sono riportate le proprietà e i termini x, y e z indicano un qualsiasi valore tra 1 e 0:

	AND	OR
Identità	$1 \cdot x = x$	$0 + x = x$
Elemento nullo	$0 \cdot x = 0$	$1 + x = 1$
Idempotenza	$x \cdot x = x$	$x + x = x$
Inverso	$x \cdot \bar{x} = 0$	$x + \bar{x} = 1$
Commutativa	$x \cdot y = y \cdot x$	$x + y = y + x$
Associativa	$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	$(x + y) + z = x + (y + z)$
Assorbimento	$x \cdot (x + y) = x$	$x + (x \cdot y) = x$
Distributiva	$x \cdot (y + z) = (x \cdot y) + (x \cdot z)$	$x + (y \cdot z) = (x + y) \cdot (x + z)$

3.1.1 Teoremi fondamentali sulle identità

Per le proprietà descritte in precedenza esistono 2 teoremi fondamentali.

Teorema 3.1.1: De Morgan

Data la dualità tra le operazioni AND e OR accade che:

- La negazione dell'AND tra due variabili corrisponde all'OR delle variabili negate: $\overline{x \cdot y} = \overline{x} + \overline{y}$.
- La negazione dell'OR tra due variabili corrisponde all'AND delle variabili negate: $\overline{x + y} = \overline{x} \cdot \overline{y}$.

Teorema 3.1.2: Shannon

Data una funzione booleana $f(B^n) = B$ descritta, in base alle variabili d'ingresso, come $f(x_1, x_2, \dots, x_n)$ è sempre vero che:

$$f(x_1, x_2, \dots, x_n) = x_1 \cdot f(1, x_2, \dots, x_n) + \overline{x_1} \cdot f(0, x_2, \dots, x_n)$$

oppure dualmente:

$$f(x_1, x_2, \dots, x_n) = (x_1 + f(0, x_2, \dots, x_n)) \cdot (\overline{x_1} + f(1, x_2, \dots, x_n))$$

3.2 Porte logiche

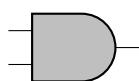
Per costruire un circuito digitale, avente lo stesso comportamento di una funzione booleana, è necessario trovare un corrispondente elettronico per le variabili e operatori.

Una variabile booleana è rappresentata da un conduttore, la cui differenza di potenziale permette di attribuire i valori **0** e **1**: in molti circuiti i potenziali sono rispettivamente 0 V e 3 V. La misurazione della differenza di potenziale di un conduttore consente di attribuire il valore logico 0 o 1 alla variabile corrispondente a quel conduttore.

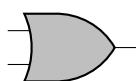
Gli operatori booleani trasformano i valori delle variabili booleane, di conseguenza devono manipolare le differenze di potenziale sulle linee di ingresso in modo tale da operare secondo le tabelle di verità, dove ai valori logici 0 e 1 corrispondono le differenze di potenziale.

Questi operatori sono rappresentati in **porte logiche**:

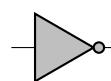
AND



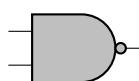
OR



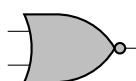
NOT



NAND

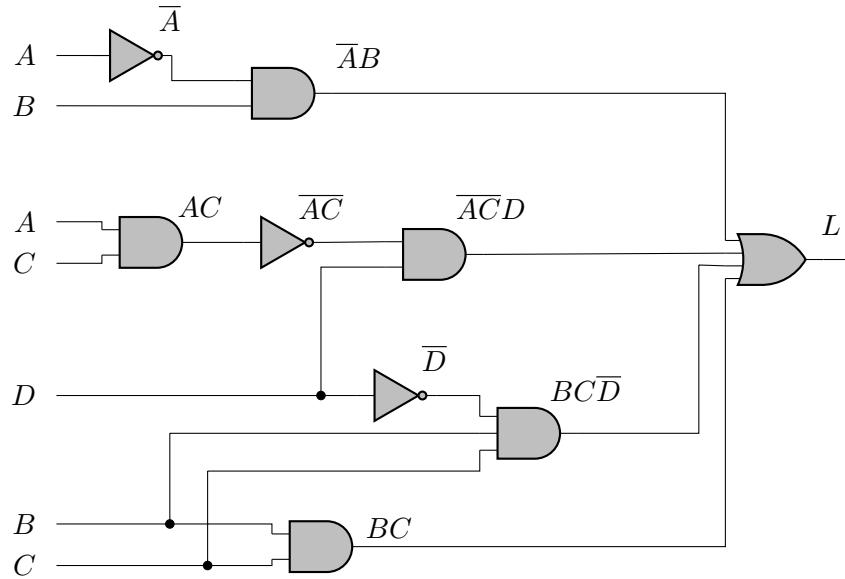


NOR



Esempio

Viene fornita la seguente somma di prodotti: $L = \overline{AB} + \overline{ACD} + BC\overline{D} + BC$.



Il pallino pieno ● indica una divisione del segnale.

Il pallino vuoto ○ indica la negazione (o l'amplificazione).

Osservazione

Si possono creare porte logiche a N ingressi, ma più ingressi vengono aggiunti e più il circuito diventa costoso a livello energetico. Non varia nulla costruire porte logiche a cascata anziché a N ingressi.

Per comprendere come una porta logica possa rappresentare un operatore booleano è necessaria la rappresentazione tramite **circuiti elettronici elementari** che la compongono, ovvero i **transistor**.

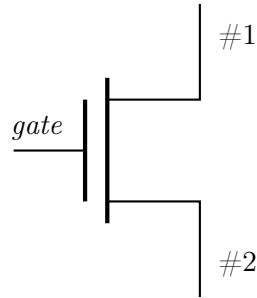
Un *transistor* possiede il comportamento di un interruttore comandato capace, in base alla presenza o assenza di tensione sull'ingresso di comando, di consentire o evitare il passaggio di corrente facendo quindi misurare alla sua uscita una differenza di potenziale uguale a 0 V oppure 3 V.

Si prenda in considerazione la tecnologia **CMOS** (*Complementary Metal Oxide Semiconductor*): è possibile modellare due tipologie di *transistor* detti NMOS e PMOS.

3.2.1 Transistor di tipo N

La presenza di una tensione di 3 V (valore logico 1) all'ingresso di comando (detto pure *gate*) mette in collegamento i terminali **#1** e **#2**. Viceversa l'assenza di tensione sull'ingresso di comando, ossia 0 V (valore logico 0), interrompe il collegamento tra i due terminali.

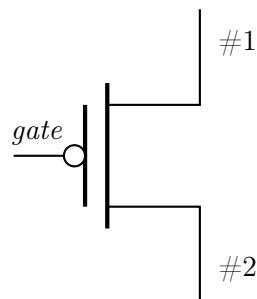
- 3 V all'ingresso (1) → collega i terminali.
- 0 V all'ingresso (0) → scollega i terminali.



3.2.2 Transistor di tipo P

L'assenza di tensione (sono presenti 0 V) all'ingresso di comando mette in collegamento i terminali **#1** e **#2**. Invece la presenza di una tensione pari a 3 V sull'ingresso di comando interrompe il collegamento tra i due terminali.

- 3 V all'ingresso (1) → scollega i terminali.
- 0 V all'ingresso (0) → collega i terminali.



3.2.3 Porte logiche elementari costruite per composizione

Operatore logico NOT

L'operatore logico NOT è ottenuto dal collegamento di un *transistor* P e N.

Se sulla linea *x* si trova una differenza di potenziale pari a 0 V (valore logico 0), allora il *transistor* P conduce mentre quello N interrompe il flusso di corrente. La linea di uscita

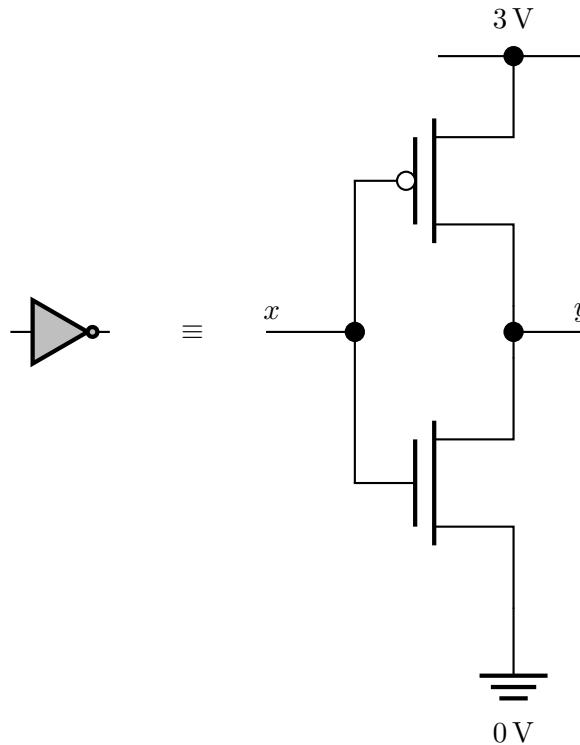
y viene collegata all'alimentazione e si misura una differenza di potenziale pari a 3 V (valore logico 1). Al contrario se la linea x viene portata al valore di 3 V (valore logico 1), allora la linea y viene collegata a terra con 0 V (valore logico 0).

Operatore logico NOT

x	y
0	1
1	0

Porta logica NOT

x	y
0 V	3 V
3 V	0 V



Porta logica NAND (NOT AND)

La porta logica NAND viene realizzata negando l'operatore logico AND. Nella parte superiore è presente un **parallelo PMOS**, mentre nella parte inferiore è presente una **serie di NMOS**.

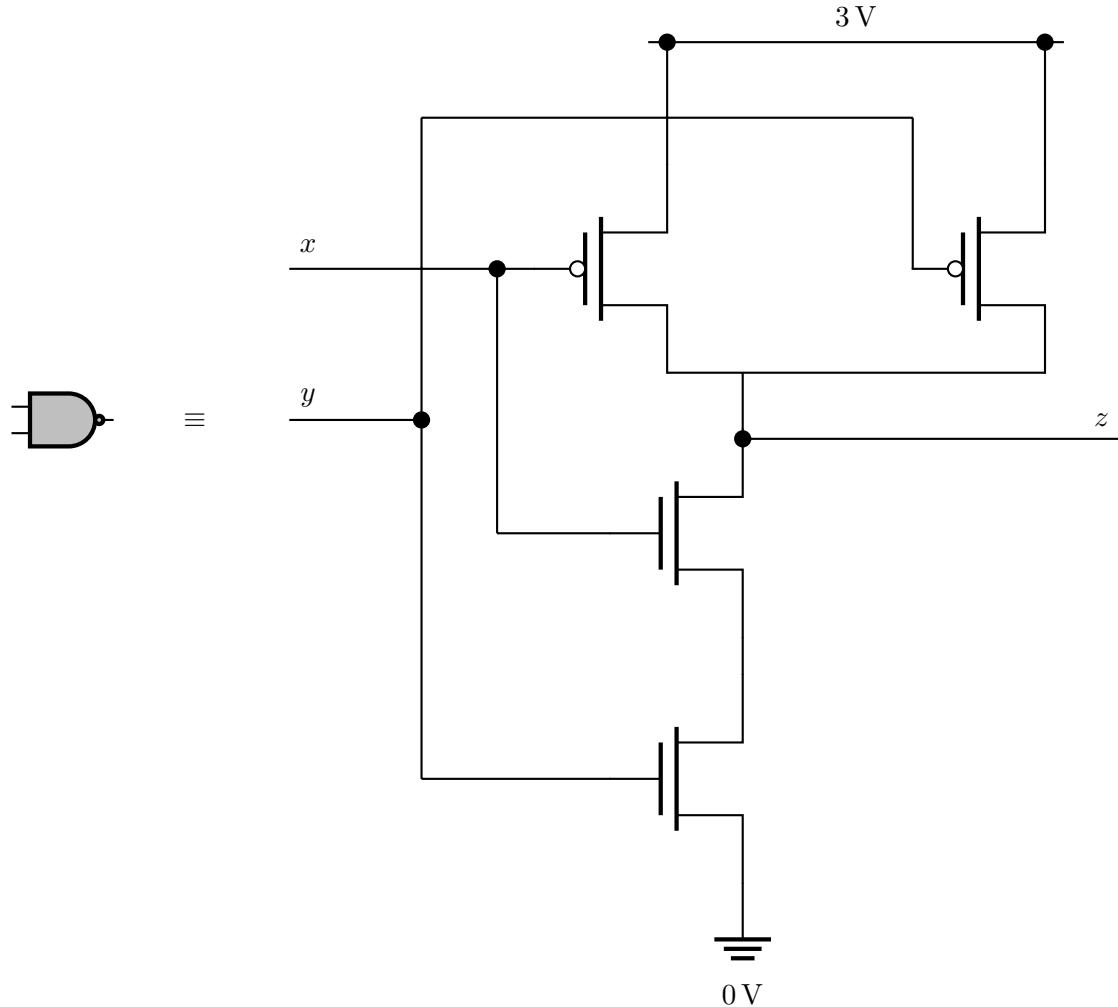
La tabella di verità corrisponde esattamente all'opposto di quella dell'AND:

Operatore logico NAND

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

Porta logica NAND

x	y	z
0 V	0 V	3 V
0 V	3 V	3 V
3 V	0 V	3 V
3 V	3 V	0 V



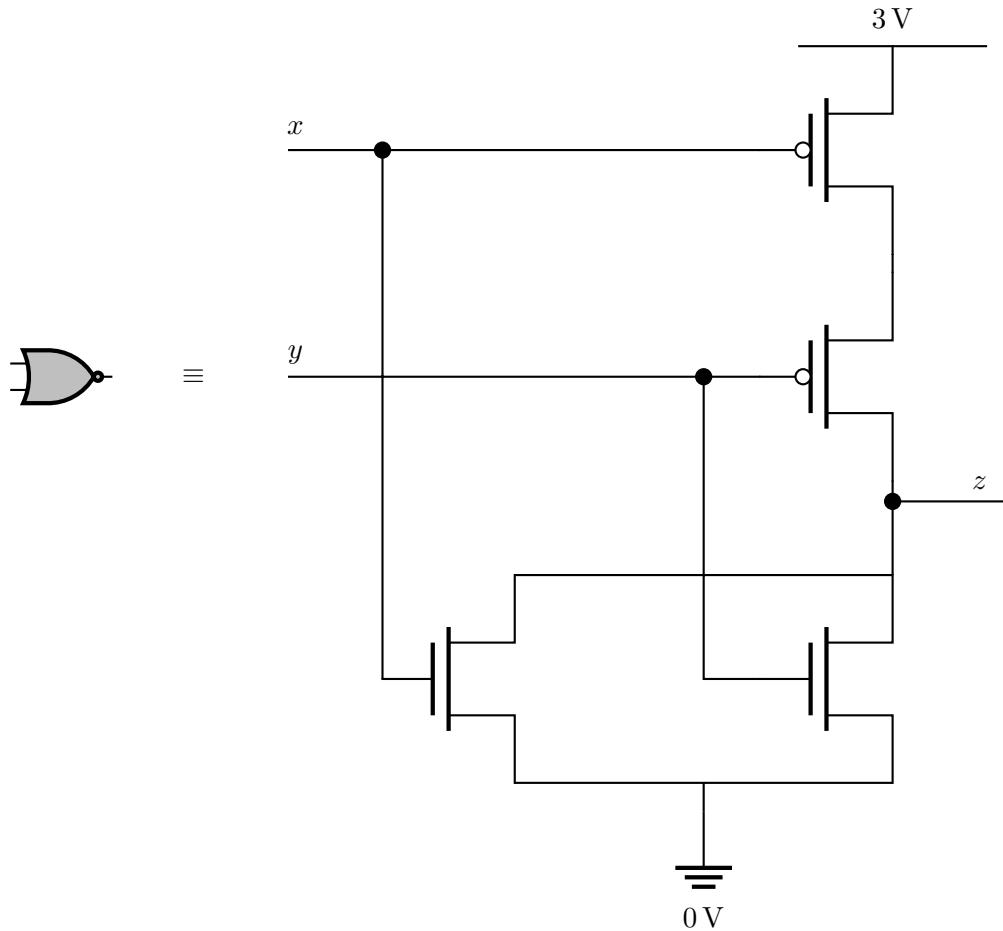
Porta logica NOR (NOT OR)

La porta logica NOR viene realizzata negando l'operatore logico OR. È complementare alla porta NAND, pertanto nella parte superiore è presente un **parallelo NMOS**, mentre nella parte inferiore è presente una **serie di PMOS**.

La tabella di verità corrisponde esattamente all'opposto di quella dell'OR:

Operatore logico NOR		
x	y	z
0	0	1
0	1	0
1	0	0
1	1	0

Porta logica NOR		
x	y	z
0 V	0 V	3 V
0 V	3 V	0 V
3 V	0 V	0 V
3 V	3 V	0 V



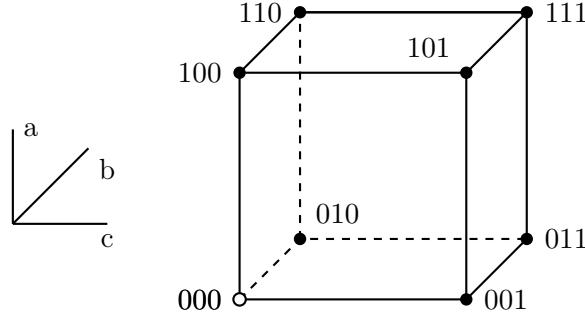
3.3 Rappresentazione di funzioni booleane

Per tradurre un'operazione booleana in un'espressione, ossia in un circuito, bisogna definire due **forme canoniche**².

Si consideri la funzione $f(B^n) = B$ nello spazio a n dimensioni, dove i punti corrispondono ai 2^n valori dell'insieme di ingresso. I punti vengono disposti sui vertici dell' n -cubo in modo che due punti adiacenti differiscano di un solo bit, nota come **distanza di Hamming 1**.

Si riporta la rappresentazione cubica della funzione $\text{OR}(a, b, c)$ descritta come la funzione $f(B^3) = B$. I vertici del cubo sono indicati con un pallino pieno se in corrispondenza di quel valore di ingresso la funzione vale 1 e con un pallino vuoto se la funzione vale 0.

²La forma canonica di una espressione booleana è un'espressione logica contenente tutte le variabili booleane in forma vera o negata, in forma di prodotti fondamentali o somme fondamentali di essi. Viene ricavata dalla tabella della verità.



- **Letterale:** indica una coppia **variabile-valore**. A ogni variabile sono associati due letterali, per esempio alla variabile a sono associate le coppie $(a, 0)$ e $(a, 1)$. Per brevità si possono riferire con le forme ridotte:

Naturale: $a \rightarrow (a, 1)$

Negata: $\bar{a} \rightarrow (a, 0)$

Le variabili di ingresso sono letterali; una funzione può essere usata come un letterale. Riferendosi alla funzione cubica rappresentata a, b e c sono tre letterali.

a	b	c	OR		a	b	c	OR
0	0	0	0		\bar{a}	\bar{b}	\bar{c}	M_0
0	0	1	1		\bar{a}	\bar{b}	c	m_1
0	1	0	1		\bar{a}	b	\bar{c}	m_2
0	1	1	1	⇒	\bar{a}	b	c	m_3
1	0	0	1		a	\bar{b}	\bar{c}	m_4
1	0	1	1		a	\bar{b}	c	m_5
1	1	0	1		a	b	\bar{c}	m_6
1	1	1	1		a	b	c	m_7

- **Implicante:** indica un prodotto dei letterali $P = x_i^a \dots x_k^a$ (con l'insieme di indici $i \dots k$ compreso nell'insieme di indici $i \dots n$, mentre a indica che la variabile può comparire sia in forma naturale sia in forma negata) di una funzione $f(x_1, \dots, x_n)$ tale che ogni volta sia $P = 1$ si ha $f = 1$. Riferendosi alla funzione cubica rappresentata ab è un implicante.
- **Mintermine:** sono tutte le variabili di ingresso che compaiono nell'implicante della funzione data. Tutti i vertici dell' n -cubo indicati con un pallino pieno sono mintermini.
- **On-set:** indica l'insieme dei mintermini della funzione $f(x_1, \dots, x_n)$. Nell'esempio del cubo, l'insieme è formato da:

$$\text{On-set} = \{m_1, m_2, m_3, m_4, m_5, m_6, m_7\}$$

- **Maxtermine:** indica un punto nello spazio booleano B^n di una funzione $f(x_1, \dots, x_n)$ tale per cui la funzione calcolata in quel punto è 0. Nell'esempio della funzione cubica, l'unico maxtermine presente è il punto $\bar{a}\bar{b}\bar{c}$.
- **Off-set:** indica i punti dello spazio booleano che non appartengono all'*on-set*, cioè è l'insieme dei maxtermini della funzione. Nell'esempio del cubo, l'insieme è formato da:

$$\text{Off-set} = \{M_0\}$$

- **Sottocubo di soli 1:** un implicante corrisponde a questo sottocubo, cioè a un insieme di 2^k configurazioni di ingresso a distanza di Hamming unitaria, a ognuna delle quali è associato un 1 della funzione.
Nella funzione cubica i quattro pallini peni della faccia superiore del cubo sono rappresentati dal sottocubo a .
- **Implicante primo:** un implicante è detto primo se non esiste alcun altro implicante di dimensioni maggiori che lo contenga interamente.

Osservazione

Un implicante di dimensioni maggiori corrisponde a un prodotto di un numero minore di letterali.

Nell'esempio del cubo il lato superiore sinistro è descritto dall'implicante $a\bar{c}$ che non è primo, perché fa parte della faccia superiore del cubo che rappresenta l'implicante a . Al contrario l'implicante a non è contenuto in nessun altro implicante, di conseguenza a è un implicante primo.

- **Implicante primo essenziale:** un implicante primo è detto essenziale se esiste almeno un 1 incluso dell'implicante stesso che non è incluso in alcun altro implicante della funzione data.
Nell'esempio del cubo gli implicanti primi a, b e c sono essenziali, poiché ciascuno copre almeno un mintermine non coperto da altri implicanti primi.
- **Copertura:** è un insieme di implicanti che coprono tutti i mintermini della funzione stessa. Nella funzione d'esempio l'insieme di implicanti primi $\{a, b, c\}$ è una copertura; anche l'insieme $\{a, b, \bar{a}c, ac\}$ è una copertura.“

Costo della funzione *OR*:

$$z = \bar{a}\bar{b}c + \bar{a}b\bar{c} + \bar{a}bc + a\bar{b}\bar{c} + a\bar{b}c + ab\bar{c} + abc = 3 \cdot 7 = 21 \text{ letterali}$$

3.3.1 1^a forma canonica (somma di prodotti)

Data una funzione booleana $f(x_1, \dots, x_n)$, sia l'*On-set* della funzione composto da k (con $k \leq 2^n$) mintermini $\{m_1, m_2, \dots, m_k\}$, allora vengono sommati tutti i mintermini:

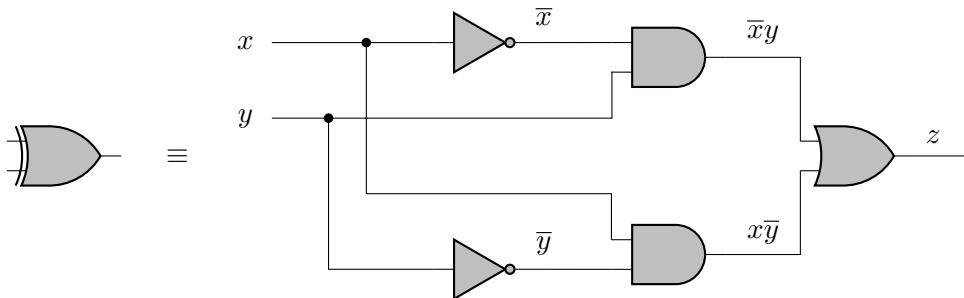
$$f(x_1, \dots, x_n) = m_1 + m_2 + \dots + m_k$$

- I k mintermini sono realizzati mediante porte AND a n ingressi.
- Le uscite delle n porte AND sono collegate a una porta OR di k ingressi.

Esempio

Viene rappresentata la funzione $f(x, y) = \text{XOR}(x, y)$ e z ne rappresenta il costo:

$$z = \bar{x}y + x\bar{y}$$



La porta AND superiore realizza il mintermine $m_1 = \bar{x}y$, mentre la porta inferiore il mintermine $m_2 = x\bar{y}$.

3.3.2 2^a forma canonica (prodotto di somme)

Questa seconda forma è duale della prima e si basa sulla composizione dei maxtermini. Data una funzione booleana $f(x_1, \dots, x_n)$, sia l'*Off-set* della funzione composto da k (con $k \leq 2^n$) maxtermini $\{M_1, M_2, \dots, M_k\}$, allora si moltiplicano tutti i maxtermini:

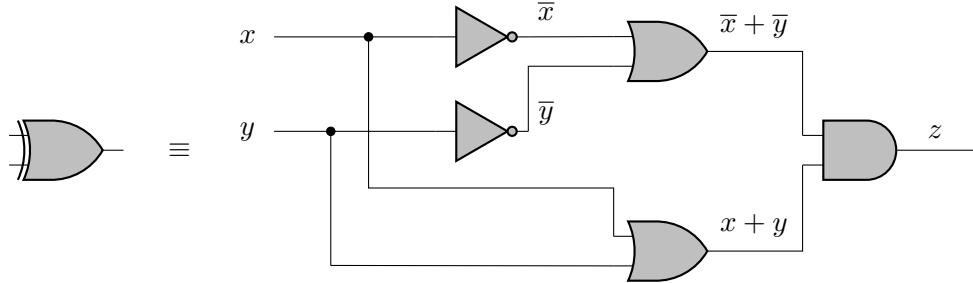
$$f(x_1, \dots, x_n) = M_1 M_2 \dots M_k$$

- I k maxtermini sono realizzati mediante porte OR a n ingressi.
- Le uscite delle n porte OR sono collegate a una porta AND di k ingressi.

Esempio

Viene rappresentata la funzione $f(x, y) = \text{XOR}(x, y)$ e z ne rappresenta il costo:

$$z = (\bar{x} + \bar{y})(x + y)$$



La porta OR superiore realizza il maxtermine $M_3 = \bar{x} + \bar{y}$, mentre la porta inferiore $M_0 = x + y$.

Considerazioni sulle forme canoniche

I circuiti in somma di prodotti e prodotto di somme sono **circuiti a due livelli**. Questa nomenclatura identifica il numero di livelli di porte logiche che i segnali devono attraversare per propagarsi dagli ingressi all'uscita.

Osservazione

In questo calcolo non vengono considerate eventuali porte NOT necessarie a ottenere la forma negata delle variabili in ingresso.

3.4 Tecniche di ottimizzazioni

Dopo aver rappresentato una funzione booleana e trovato il costo di tale funzione, è necessario ottimizzare la rappresentazione tramite delle tecniche di semplificazione. Si considera la seguente funzione a tre variabili in ingresso, con le seguenti informazioni:

- **On-set:** $\{m_0, m_1, m_4, m_5, m_7\}$;
- **Off-set:** $\{M_2, M_3, M_6\}$;
- **Costo:** $m_0 + m_1 + m_4 + m_5 + m_7 = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + x\bar{y}\bar{z} + x\bar{y}z + xyz = 3 \cdot 5 = 15$ letterali.

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

La funzione f possiede un determinato costo ricavato dalla somma di mintermini, ma è possibile applicare la **legge di assorbimento** tra i mintermini; si applica questa legge ai mintermini $\bar{x}\bar{y}z + \bar{x}\bar{y}z$. L'obiettivo è annullare il letterale che risulta diverso in entrambi i mintermini e in questo caso è il letterale z , perché: $\bar{x}\bar{y}z + \bar{x}\bar{y}z = \bar{x}\bar{y} \cdot (\bar{z} + z) = \bar{x}\bar{y}$. Si prosegue con questo procedimento anche per gli altri mintermini:

- Tra m_4 e m_5 : $\bar{x}\bar{y}z + \bar{x}\bar{y}z = \bar{x}\bar{y} \cdot (\bar{z} + z) = \bar{x}\bar{y}$.
- Tra m_5 e m_7 : $\bar{x}\bar{y}z + xyz = xz \cdot (\bar{y} + y) = xz$.

La forma finale temporanea della funzione vale 6 letterali: $\bar{x}\bar{y} + x\bar{y} + xz$.

È possibile applicare la legge di assorbimento per lo stesso mintermine più di una volta come nel caso di m_5 .

Legge di assorbimento: $b \cdot B + \bar{b} \cdot B = B \cdot (b + \bar{b}) = B$

Data la nuova forma del costo, è permesso proseguire ancora con tale legge e ripetere questo procedimento finché la forma non risulta essere più semplificabile: $\bar{x}\bar{y} + x\bar{y} = \bar{y}$. Quindi il risultato finale è: $\bar{y} + xz = 3$ letterali.

▷ Come si può essere sicuri che questo risultato sia il migliore?

Serve un algoritmo per l'ottimizzazione assoluta, altrimenti non si può garantire che sia la forma finale migliore.

3.4.1 Concetti preliminari per l'ottimizzazione

Per arrivare alla costruzione dell'algoritmo bisogna:

- Applicare la legge di assorbimento per arrivare all'insieme di prodotti più piccolo.
- Provare che la formula finale copra tutti i mintermini. Nel risultato finale dell'esempio precedente, il primo prodotto \bar{y} è nella specifica forma³ -0-. Quindi \bar{y} copre 4 mintermini (000, 001, 100, 101), mentre il secondo prodotto xz è nella forma: 1-1. Di conseguenza quest'ultimo copre soltanto 2 mintermini (101, 111).

³I trattini segnalano la presenza di **don't care**, cioè la possibilità che quel punto possa essere 1 oppure 0.

- Utilizzare la legge di assorbimento finché non si giunge a soli **implicanti primi**, ovvero implicanti che non sono contenuti in nessun altro implicante.

Non sono implicanti primi

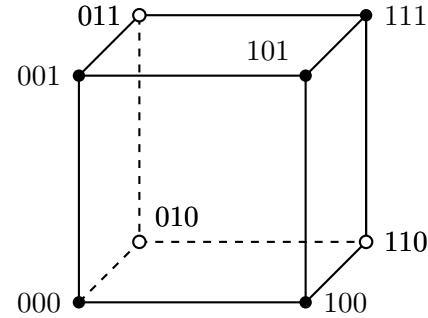
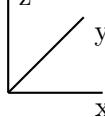
$$\boxed{\overline{xy} + x\bar{y}} + xz$$

Sono implicanti primi

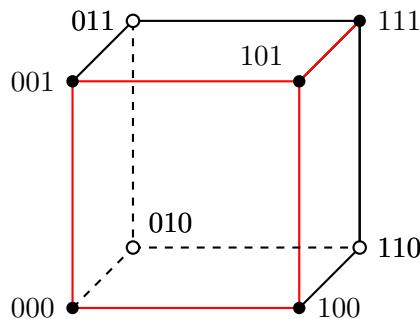
- Infine non tutti gli implicanti primi potrebbero servire, occorre individuare gli **implicanti primi essenziali**, ossia implicanti primi che coprono almeno un minitermine che non è coperto da nessun altro implicante primo.

Si consideri la funzione f vista in precedenza:

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



La legge di assorbimento si può applicare solo a mintermini adiacenti a distanza di Hamming⁴ di 1 bit. Non è possibile applicare la legge in diagonale, ma solo seguendo i lati.



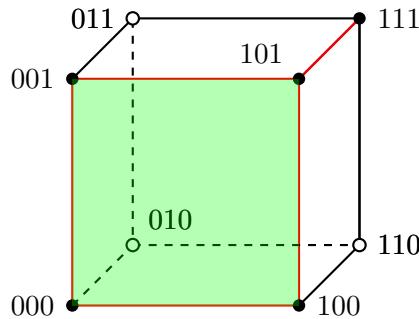
⁴Numero di *bit* per cui due codici differiscono, per esempio: 000 e 001 differiscono di 1 bit, 001 e 100 differiscono di 2 bit.

I lati evidenziati in rosso rispettano la **distanza di Hamming 1**, perciò è possibile applicare la legge di assorbimento tra i mintermini adiacenti risultando:

$$\begin{aligned} 000 + 001 &\rightarrow \overline{x}yz + \overline{x}y\bar{z} = \overline{xy} \\ 000 + 100 &\rightarrow \overline{x}yz + x\bar{y}\bar{z} = \overline{yz} \\ 001 + 101 &\rightarrow \overline{x}y\bar{z} + x\bar{y}z = \bar{y}z \\ 100 + 101 &\rightarrow x\bar{y}\bar{z} + x\bar{y}z = x\bar{y} \\ 101 + 111 &\rightarrow x\bar{y}z + xyz = xz \end{aligned}$$

Avendo una faccia (contrassegnata in verde) di 4 implicanti, si può applicare l'assorbimento tra di loro semplificando ulteriormente:

$$\overline{xy} + \overline{yz} + \overline{yz} + x\bar{y} + xz = \overline{y} + xz$$



È stato applicato l'assorbimento ai 4 mintermini evidenziati poiché presenti nella stessa faccia.

3.4.2 Mappa di Karnaugh

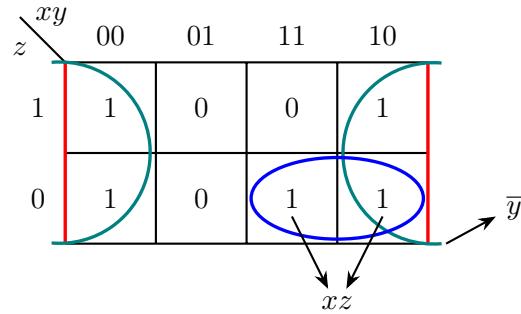
Definizione 3.4.1: Mappa di Karnaugh

È un metodo visivo che permette di trovare sottocubi rappresentando su un piano lo spazio geometrico.

Le celle rappresentano i vertici del cubo. Quando vengono distribuiti i *bit* sulla mappa, devono essere a distanza di Hamming 1:

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

La mappa vista su un piano può essere pensata come una sfera, dove il lato sinistro si congiunge con quello destro (evidenziati in rosso), in questo modo è possibile ricavare i **sottocubi massimi**:



In questo caso ci sono due implicanti primi, i quali sono anche implicanti essenziali, tuttavia potrebbe capitare una funzione dove si trovano più implicanti primi e bisogna ricercare quelli essenziali.

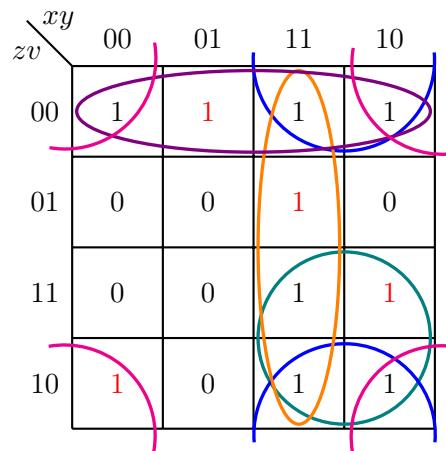
Osservazione

Non è detto che tutti gli implicanti primi siano essenziali.

Per trovare gli implicanti essenziali si cercano gli 1 coperti da un solo implicante.

Esempio

Data una funzione f , i **sottocubi massimi** che si possono formare sono cerchiati con colori differenti, mentre gli 1 in rosso definiscono gli implicanti primi essenziali.



Il costo della funzione è pari alla somma degli implicanti primi essenziali che coprono gli 1 coperti da un solo implicante: $f = \bar{z}\bar{v} + \bar{y}\bar{v} + xy + xz$.

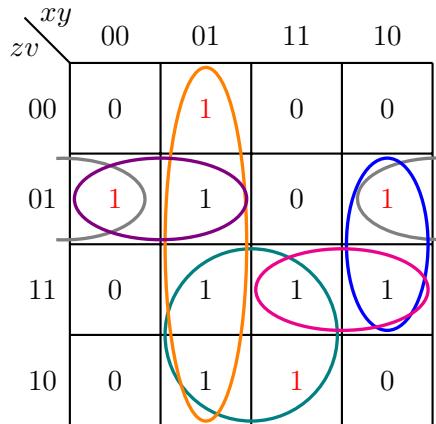
Esempio

Si considera la seguente funzione $f(B^4) \rightarrow B$ con la relativa tabella di verità:

x	y	z	v	\parallel	o
0	0	0	0		0
0	0	0	1		1
0	0	1	0		0
0	0	1	1		0
0	1	0	0		1
0	1	0	1		1
0	1	1	0		1
0	1	1	1		1

x	y	z	v	\parallel	o
1	0	0	0		0
1	0	0	1		1
1	0	1	0		0
1	0	1	1		1
1	1	0	0		0
1	1	0	1		0
1	1	1	0		1
1	1	1	1		1

Di seguito è fornita la mappa di Karnaugh con gli implicanti primi cerchiati e gli uni degli implicanti primi essenziali evidenziati in rosso:



Il costo finale della funzione risulta: $o = \bar{x}y + yz + \bar{x}\bar{z}v + x\bar{y}v$.

3.5 Metodo di Quine-McCluskey

3.5.1 Funzioni completamente specificate

Il metodo visivo con le **mappe di Karnaugh** può essere comodo per funzioni booleane con pochi ingressi, ma quando iniziano a essere tanti (già a partire da 5 o più) diventa difficile individuare gli implicanti primi essenziali. Ecco che il **metodo di Quine-McCluskey** trova una copertura ottima attraverso un algoritmo.

Per illustrare il funzionamento dell'intero processo, si tiene in considerazione la funzione booleana vista nell'ultimo esempio delle mappe di Karnaugh. Si riporta per praticità la tabella di verità relativa a questa funzione e il suo *On-set*:

$$\text{On-set} = \{m_1, m_4, m_5, m_6, m_7, m_8, m_9, m_{11}, m_{14}, m_{15}\}$$

x	y	z	v	o		x	y	z	v	o
0	0	0	0	0		1	0	0	0	0
0	0	0	1	1		1	0	0	1	1
0	0	1	0	0		1	0	1	0	0
0	0	1	1	0		1	0	1	1	1
0	1	0	0	1		1	1	0	0	0
0	1	0	1	1		1	1	0	1	0
0	1	1	0	1		1	1	1	0	1
0	1	1	1	1		1	1	1	1	1

Questo algoritmo opera seguendo due fasi:

1. Individuazione degli implicanti primi.

- Si riordinano i mintermini della tabella secondo il numero di 1 contenuti nella corrispondente configurazione di ingresso, ottenendo così dei gruppi di mintermini con lo stesso numero di 1.

m_i	x	y	z	v
m_1	0	0	0	1
m_4	0	1	0	0
m_5	0	1	0	1
m_6	0	1	1	0
m_9	1	0	0	1
m_7	0	1	1	1
m_{11}	1	0	1	1
m_{14}	1	1	1	0
m_{15}	1	1	1	1

Le linee orizzontali delineano i gruppi delle configurazioni i aventi lo stesso numero di 1.

- Ogni configurazione i in un gruppo viene confrontata con tutte le configurazioni nel gruppo immediatamente successivo, poiché sono le uniche a essere potenzialmente a distanza di Hamming 1. Quando si trova una configurazione j adiacente alla i , ciò indica la presenza di una somma di due mintermini, del tipo: $Aa + A\bar{a} = A$; dove A è il prodotto di tutti i letterali identici nelle due configurazioni, mentre a è l'unico letterale che nelle due configurazioni compare in forma diversa.

Quindi si costruisce una seconda tabella in cui viene inserita la configurazione A , con un trattino che identifica il *don't care* al posto della variabile a . Entrambe le configurazioni i e j nella tabella di partenza vengono marcate, in quanto non corrispondono a implicanti primi, perché è stato identificato un sottocubo di dimensioni maggiori che li include. Questo procedimento si ripete per tutti i possibili confronti che si possono svolgere all'interno della tabella tra gruppi di configurazioni.

m_i	x	y	z	v		$m_{\{i,j\}}$	x	y	z	v
m_1	0	0	0	1	✓	1,5	0	-	0	1
m_4	0	1	0	0	✓	1,9	-	0	0	1
m_5	0	1	0	1	✓	4,5	0	1	0	-
m_6	0	1	1	0	✓	4,6	0	1	-	0
m_9	1	0	0	1	✓	5,7	0	1	-	1
m_7	0	1	1	1	✓	6,7	0	1	1	-
m_{11}	1	0	1	1	✓	6,14	-	1	1	0
m_{14}	1	1	1	0	✓	9,11	1	0	-	1
m_{15}	1	1	1	1	✓	7,15	-	1	1	1
						11,15	1	-	1	1
						14,15	1	1	1	-

- Una volta ottenuta la nuova tabella, si effettua lo stesso procedimento tenendo in considerazione anche i *don't care*, la cui posizione deve combaciare con quella di altri *don't care* per fare avvenire l'assorbimento. Nel momento in cui una configurazione i non trova una corrispondenza con un'altra configurazione j , allora si marca la configurazione i con una lettera maiuscola la quale sta a indicare che i è sicuramente un implicante primo non contenuto in un sotto-cubo di dimensioni maggiori. Ovviamente, anche se raro, questa marcatura può avvenire anche nella tabella iniziale del punto precedente.

$m_{\{i,j\}}$	x	y	z	v		$m_{\{i,j\}}$	x	y	z	v
1,5	0	-	0	1	A	(4,5), (6,7)	0	1	-	-
1,9	-	0	0	1	B	(6,7), (14,15)	-	1	1	-
4,5	0	1	0	-	✓					
4,6	0	1	-	0	✓					
5,7	0	1	-	1	✓					
6,7	0	1	1	-	✓					
6,14	-	1	1	0	✓					
9,11	1	0	-	1	C	(4,5), (6,7)	0	1	-	-
7,15	-	1	1	1	✓	(6,7), (14,15)	-	1	1	-
11,15	1	-	1	1	D					
14,15	1	1	1	-	✓					

↓

$m_{\{i,j\}}$	x	y	z	v	
(4,5), (6,7)	0	1	-	-	E
(6,7), (14,15)	-	1	1	-	F

Osservazione

Potrebbe capitare che due coppie di configurazioni i_1, j_1 e i_2, j_2 producano lo stesso risultato, allora in quel caso si mantiene solo una delle due coppie. Per esempio le coppie $(4,5), (6,7)$ producono lo stesso risultato di $(4,5), (5,7)$, di conseguenza si mantiene soltanto una delle due.

Non essendoci più implicanti primi da confrontare, si mantengono quelli marcati dalle lettere: $A = \bar{x}\bar{z}v$, $B = \bar{y}\bar{z}v$, $C = x\bar{y}v$, $D = xzv$, $E = \bar{x}y$, $F = yz$.

2. Ricerca della copertura ottima.

- Si crea una nuova tabella dove vengono disposti in colonna tutti i mintermini e in riga tutti gli implicanti primi marcati in precedenza, mentre all'interno vengono disposti gli 1 dove l'implicante include/copre il mintermine:

m_i	A	B	C	D	E	F
m_1	1	1				
m_4						1
m_5	1					1
m_6					1	1
m_7					1	1
m_9		1	1			
m_{11}			1	1		
m_{14}						1
m_{15}				1	1	

- In questa tabella si ricercano gli **implicanti primi essenziali** nelle righe che contengono un solo 1 (quelli cerchiati).

m_i	A	B	C	D	E	F
m_1	1	1				
m_4						①
m_5	1					1
m_6					1	1
m_7					1	1
m_9		1	1			
m_{11}			1	1		
m_{14}						①
m_{15}				1	1	

Si semplifica la tabella eliminando le colonne e le righe appartenenti ai mintermini che hanno l'1 nella medesima colonna degli 1 cerchiati:

m_i	A	B	C	D	E	F
m_1	1	1				
m_4					①	
m_5	1					
m_6			1	1		
m_7			1	1		
m_9		1	1			
m_{11}			1	1		
m_{14}					①	
m_{15}				1	1	

\Rightarrow

m_i	A	B	C	D
m_1	1	1		
m_9		1	1	
m_{11}			1	1

- A questo punto non è più possibile trovare l'**essenzialità**. Si passa a ricercare la **dominanza**, ossia le colonne che riescono a coprire più 1 rispetto altre colonne. La colonna B domina la colonna A e la C domina D , pertanto si eliminano tutte le colonne dominate:

m_i	A	B	C	D		m_i	B	C
m_1	1	1				m_1	①	
m_9		1	1			m_9	1	1
m_{11}			1	1		m_{11}		①

Rimangono le colonne B e C che vengono dette **pseudo-essenziali**. La colonna B è pseudo-essenziale per m_1 , mentre C è pseudo-essenziale per m_{11} . Cancellando anche queste ultime due colonne, non rimangono più semplificazioni da eseguire.

Infine il risultato finale è: $f = B + C + E + F$.

Il risultato ottenuto è un risultato ottimo, ma non è detto che sia l'unico.

Caso particolare

Data una qualsiasi funzione che si riconduce nella tabella sottostante, non è detto che ci sia sempre l'essenzialità; non c'è nemmeno la dominanza di colonne. Allora si può procedere secondo la **dominanza per riga**: γ e ε dominano su α , β e δ .

m_i	A	B	C	D
α	1		1	
β	1			1
γ		1	1	1
δ		1	1	
ε	1	1		1

Le righe dominanti γ e ε vengono eliminate. A questo punto si ricercano le colonne dominanti e si eliminano quelle dominate: B e D .

m_i	A	C
α	1	1
β	①	
δ		①

Da quest'ultima tabella ottenuta A risulta essenziale per β e C è essenziale per δ , di conseguenza il costo risulta $f = A + C$.

3.5.2 Funzioni parzialmente specificate

Esistono funzioni descritte secondo **condizioni di indifferenza** (*don't care*), corrispondenti a:

- configurazioni in ingresso che non si presenteranno mai e per le quali qualunque valore d'uscita è ammisible;
- configurazioni di uscita non utilizzate per le quali qualunque valore è ammisible.

Questa tipologia di **funzioni parzialmente specificate** viene descritta mediante:

- ***On-set***: insieme delle configurazioni in ingresso per le quali la funzione vale 1;
- ***Don't care-set (DC-set)***: insieme delle configurazioni in ingresso per le quali la funzione non è specificata e può assumere indifferentemente sia il valore 0 che il valore 1.

Osservazione

L'unione dell'*On-set*, *DC-set* e *Off-set* rappresenta l'insieme di tutte le configurazioni in ingresso di B^n . Di conseguenza una funzione parzialmente specificata può essere descritta attraverso l'*On-set* e *Off-set*, oppure con una delle altre possibili combinazioni di due dei tre insiemi.

Si considera la seguente funzione $f(x, y, z, v)$:

x	y	z	o	
0	0	0	1	
0	0	1	1	$On-set = \{m_0, m_1, m_4, m_5, m_7\}$
0	1	0	-	
0	1	1	0	$Off-set = \{M_3, M_6\}$
1	0	0	1	
1	0	1	1	$DC-set = \{m_2\}$
1	1	0	0	
1	1	1	1	

In questa funzione il punto m_2 non appartiene né all'*On-set* né all'*Off-set*, perché in quel punto è indifferente generare uno 0 o un 1. Ciò è dovuto magari al fatto che per esempio il valore generato non viene mai usato dal sistema.

Allora si inserisce il simbolo di *don't care* e risulta un vantaggio per l'algoritmo di minimizzazione, così da poter scegliere quando valere 0 o 1.

Il *DC-set* è l'insieme che si riferisce ai punti che indicano un *don't care*.

È compito dell'algoritmo di Quine-McCluskey completare i *don't care*:

- Nella prima fase, quando si ricercano gli implicanti primi, tutto il *DC-set* diventa parte dell'*On-set*. Gli implicanti primi che verranno costruiti saranno nel caso peggiore gli stessi che si avrebbero non sfruttando il *DC-set*, oppure nel caso migliore saranno (grazie ai nuovi punti) di più o più grandi: costeranno meno e disporranno più alternative.

Può capitare che alcuni degli implicanti primi trovati in questo modo coprano solo il *DC-set*, ciò significa che una zona (rappresentabile nello spazio) è composta solo da *DC-set* che trasformati in 1 permettono la costruzione di un implicante primo. Tuttavia quell'implicante primo che copre solo punti del DC-SET non ha utilità, quindi è già eliminabile.

- Nella seconda fase, dove l'obiettivo è quello trovare il minimo numero di implicanti, il *DC-set* viene considerato come *Off-set*. Ovvero nella tabella di copertura si inseriscono soltanto i punti dell'*On-set* e non del *DC-set*.

Così facendo si sfruttano i *don't care* il meglio possibile.

I *don't care* presenti a destra, come uscite della funzione, indicano che la funzione è parzialmente specificata. Viceversa se i *don't care* si trovano a sinistra, nell'elenco dei punti della funzione, allora risulta essere solo una semplificazione di scrittura e non una funzione parzialmente specificata.

Esempio

Si considera una funzione booleana $f(x, y, v, z)$ e la sua tabella di verità:

x	y	z	v	o	x	y	z	v	o
0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	1	0
0	0	1	0	0	1	0	1	0	1
0	0	1	1	-	1	0	1	1	1
0	1	0	0	1	1	1	0	0	0
0	1	0	1	-	1	1	0	1	1
0	1	1	0	-	1	1	1	0	1
0	1	1	1	-	1	1	1	1	1

Da questa tabella di verità si derivano i vari insiemi:

$$\text{On-set} = \{m_4, m_{10}, m_{11}, m_{13}, m_{14}, m_{15}\}$$

$$\text{Off-set} = \{M_0, M_1, M_2, M_8, M_9, M_{12}\}$$

$$\text{DC-set} = \{m_3, m_5, m_6, m_7\}$$

1. Individuazione degli implicanti primi.

- La prima fase avviene come nelle funzioni completamente specificate, con la sola differenza di considerare oltre l'*On-set* anche il *DC-set*. Quindi si ottiene la seguente tabella ordinata per gruppi di configurazioni:

m_i	x	y	z	v
m_4	0	1	0	0
m_3	0	0	1	1
m_5	0	1	0	1
m_6	0	1	1	0
m_{10}	1	0	1	0
m_7	0	1	1	1
m_{11}	1	0	1	1
m_{13}	1	1	0	1
m_{14}	1	1	1	0
m_{15}	1	1	1	1

- Si svolgono i confronti tra le configurazioni i di un gruppo con le configurazioni j del gruppo successivo. Ovviamente le configurazioni dei gruppi confrontati devono essere a distanza di Hamming 1, altrimenti non si effettuano a prescindere i confronti.

m_i	x	y	z	v	$m_{\{i,j\}}$	x	y	z	v	
m_4	0	1	0	0	✓	4,5	0	1	0	-
m_3	0	0	1	1	✓	4,6	0	1	-	0
m_5	0	1	0	1	✓	3,7	0	-	1	1
m_6	0	1	1	0	✓	3,11	-	0	1	1
m_{10}	1	0	1	0	✓	5,7	0	1	-	1
m_7	0	1	1	1	✓	5,13	-	1	0	1
m_{11}	1	0	1	1	✓	6,7	0	1	1	-
m_{13}	1	1	0	1	✓	6,14	-	1	1	0
m_{14}	1	1	1	0	✓	10,11	1	0	1	-
m_{15}	1	1	1	1	✓	10,14	1	-	1	0
						7,15	-	1	1	1
						11,15	1	-	1	1
						13,15	1	1	-	1
						14,15	1	1	1	-

- Si esegue ancora una volta lo stesso procedimento, ricordando che le configurazioni i e j combacianti devono avere anche i *don't care* nella stessa posizione. Se così non dovesse essere, allora non sono coppie valide; 1-01 e 10-1 non lo sono, mentre le coppie -101 e -001 lo sono.

$m_{\{i,j\}}$	x	y	z	v	
4,5	0	1	0	-	✓
4,6	0	1	-	0	✓
3,7	0	-	1	1	✓
3,11	-	0	1	1	✓
5,7	0	1	-	1	✓
5,13	-	1	0	1	✓
6,7	0	1	1	-	✓
6,14	-	1	1	0	✓
10,11	1	0	1	-	✓
10,14	1	-	1	0	✓
7,15	-	1	1	1	✓
11,15	1	-	1	1	✓
13,15	1	1	-	1	✓
14,15	1	1	1	-	✓

$m_{\{i,j\}}$	x	y	z	v
(4,5), (6,7)	0	1	-	-
(3,7), (11,15)	-	-	1	1
(5,7), (13,15)	-	1	-	1
(6,7), (14,15)	-	1	1	-
(10,11), (14,15)	1	-	1	-

- Infine poiché non si hanno più confronti disponibili, allora si marcano con delle lettere le configurazioni

$m_{\{i,j\}}$	x	y	z	v	
(4,5), (6,7)	0	1	-	-	A
(3,7), (11,15)	-	-	1	1	B
(5,7), (13,15)	-	1	-	1	C
(6,7), (14,15)	-	1	1	-	D
(10,11), (14,15)	1	-	1	-	E

2. Ricerca della copertura ottima.

- Si crea una nuova tabella dove vengono disposti in colonna tutti i mintermini appartenenti solo all'*On-set* e in riga tutti gli implicanti primi marcati in precedenza, mentre all'interno vengono disposti gli 1 dove l'implicante include/copre il mintermine:

m_i	A	B	C	D	E
m_4	1				
m_{10}					1
m_{11}		1			1
m_{13}			1		
m_{14}				1	1
m_{15}	1	1	1	1	

- In questa tabella si ricercano gli implicanti primi essenziali nelle righe che contengono un solo 1 (quelli cerchiati).

m_i	A	B	C	D	E
m_4	(1)				
m_{10}					(1)
m_{11}		1			1
m_{13}			(1)		
m_{14}				1	1
m_{15}		1	1	1	1

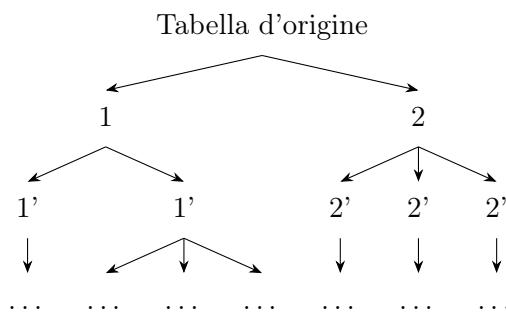
Si semplifica la tabella eliminando le colonne e le righe appartenenti ai mintermini che hanno l'1 nella medesima colonna degli 1 cerchiati:

m_i	A	B	C	D	E
m_4	1				
m_{10}					1
m_{11}		1			1
m_{13}			1		
m_{14}				1	1
m_{15}		1	1	1	1

- A questo punto la tabella si è completamente semplificata, risultando così: $f = A + C + E$.

3.5.3 Metodo per ipotesi

Se si dovesse arrivare a un punto in cui non è possibile trovare implicanti primi essenziali, pseudo-essenziali e nemmeno dominanze di righe/colonne, allora c'è un ulteriore metodo di proseguimento definito come **metodo per ipotesi**. In questo metodo viene applicato l'**algoritmo di “Branch and Bound”**: si effettuano delle ipotesi di proseguimento e si scelgono mano a mano le migliori.



Il risultato ottenuto è un risultato buono, ma non è detto che sia un risultato ottimo. Inoltre potrebbe essere difficile completare ogni ipotesi quando le ipotesi sono troppe da valutare: diventa computazionalmente inefficiente.

Reti combinatorie

Una funzione booleana $f(B^n) \rightarrow B$ può essere esaminata tramite:

- Sintesi a 2 livelli → permette di trovare un **algoritmo esatto** (eccetto quando la copertura minima è di complessità esponenziale, in questo caso la soluzione è approssimata):

Al risultato finale di minimizzazione si ottiene una serie di **porte AND** che calcolano i prodotti, poi il risultato dei prodotti all'interno di una **porta OR** che ne fa la somma.

Questa struttura di circuito rimane teorica, perché avrebbe una forma impossibile da realizzare. Inoltre le **porte AND** sarebbero troppo grandi all'aumentare dei letterali.

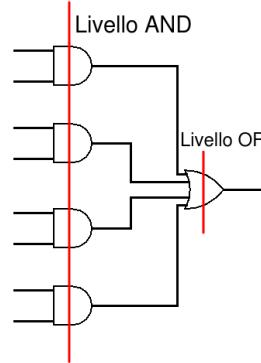
- Sintesi a n livelli → permette di trovare solo **algoritmi approssimati** (non esatti): I risultati con la **sintesi a n livelli** in alcuni casi, sebbene non dia mai risultati esatti ma solo approssimati, risulta migliore della **sintesi a 2 livelli**.

4.1 Sintesi a 2 livelli

La **sintesi a 2 livelli** non è utilizzabile perché restituisce risultati non direttamente utilizzabili.

I 2 livelli sono intesi come:

- **Livello dell'AND**: che sono tutte le porte AND in serie del circuito.
- **Livello dell'OR**: una porta OR la quale somma i risultati ottenuti dagli AND.



Questo tipo di circuiti richiede la costruzione di porte AND/OR troppo grandi. Di conseguenza costruire porte AND/OR a N ingressi richiederebbe una crescita esponenziale e non lineare.

4.1.1 Circuiti PLA, FPGA, ASIC

-Circuito PLA:

Ideati nella seconda metà degli anni '60 e prodotti verso la fine degli anni '70, i **circuiti PLA** (*programmable logic array*), sono dei circuiti creati appositamente per realizzare funzioni booleane minimizzate a **2 livelli**.

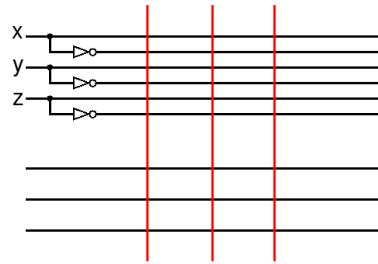
Struttura **circuito PLA**:

- Le linee orizzontali sono gli ingressi e sono limitati a un certo numero massimo che pongono il vincolo di utilizzo.
- Successivamente ogni ingresso è anche negato, creando delle linee orizzontali sotto quelle di ingresso.
- Le linee verticali sono le **linee di prodotto**, e permettono di realizzare prodotti (hanno la funzionalità di AND).
Inoltre queste **linee di prodotto** sono anche degli ingressi aggiuntivi, **ingressi di programmazione**, che vengono usati nella fase di programmazione.
- A fine linea degli ingressi esistono altre linee orizzontali che sono le **linee di somma** (hanno la funzionalità di un OR).

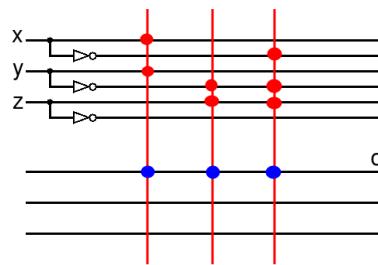
Esempio:

Bisogna realizzare la funzione $O = xy + \bar{y}z + \bar{x}\bar{y}z$.

Per creare il prodotto tra xy bisogna collegare la **linea di prodotto** con gli ingressi x e y , per fare ciò bisogna fondere la linea di prodotto 1 con i due ingressi applicando una corrente di maggiore intensità che per effetto Joule fonde quei punti (il contatto creato è permanente). Per creare gli altri prodotti si procede nella medesima maniera.



Infine i prodotti vengono sommati nella **linea di somma** permettendo di rappresentare la funzione booleana.



I **circuiti PLA** hanno due difetti:

1. Sono legate al concetto di 2 livelli, e per realizzare circuiti più grandi con una serie di funzionalità specifiche serve allentare il vincolo dei 2 livelli.
2. La PLA non è riprogrammabile: una volta programmata fa ciò per cui è stata programmata e non altro.

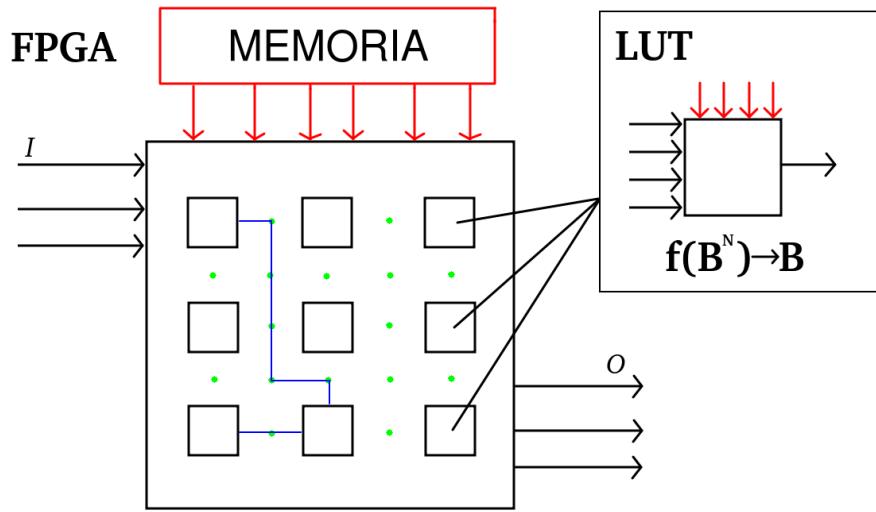
Questo problema ha fatto sì che si passasse verso le **FPGA** (*Field Programmable Gate Array*) che è riprogrammabile ogni volta che serve.

-Circuito FPGA:

Il **circuito FPGA** è composto da:

- All'interno è presente una serie di blocchetti chiamati **LUT** (*Look Up Table*). Nelle **FPGA** moderne sono presenti decine/centinaia di migliaia di **LUT**.
- I **LUT** sono dei circuiti composti da un numero N limitato di bit in ingresso e un bit d'uscita, in questo modo è capace di realizzare una qualsiasi funzione fra tutte le funzioni booleane $f(B^N) \rightarrow B$: prendendo una serie di **ingressi di programmazione** e settando i valori di ingresso a un valore predefinito.
- Da sinistra entrano gli **ingressi principali** (I).
- Da destra escono le **uscite principali** (O).

- Dall'alto entrano degli **ingressi di programmazione** provenienti da una memoria (in genere **memoria flash**) che permette di memorizzare tutte le configurazioni che programmano tutta la **FPGA**: quindi per programmare una **FPGA** bisogna mettere nella sua memoria una sequenza di zeri e uni che programmano tutte le **LUT** interne.
- La memoria consente anche di programmare gli **SWITCH** presenti in ogni punto della **FPGA** e che permettono di decidere la direzione delle uscite delle **LUT**.



L'**FPGA** ha il difetto di essere sovradimensionato rispetto ciò che realmente serve: non si usa totalmente le risorse presenti all'interno.

Quando il numero di circuiti da realizzare è superiore in media ai centomila pezzi allora si realizza un **ASIC** (*Application Specific Intregation Circuit*).

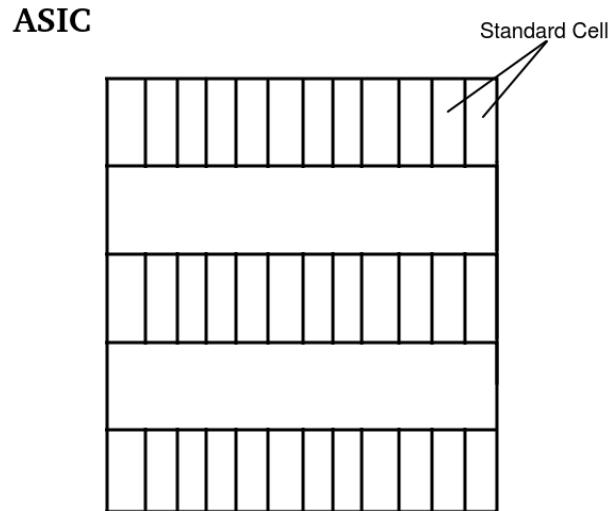
-Circuito ASIC:

L'**ASIC** è un circuito che viene realizzato disegnando tutte le porte logiche (trovate alla fine della minimizzazione) transistor per transistor.

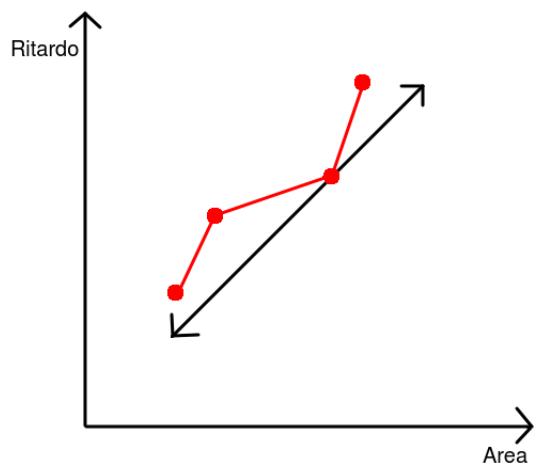
Generalmente è diviso in fasce che permettono di realizzare una serie di porte logiche chiamate **Standard Cell**, ossia le unità di base con cui realizzare un circuito. Quindi per realizzare un **ASIC** si trasforma una funzionalità in un insieme di **Standard Cell**.

4.1.2 Problemi e vantaggi della sintesi a 2 livelli

Una funzione $f(B^n) \rightarrow B$ riesce a essere minimizzata a 2 livelli trovando l'espressione minima con Q&MC, ma la sua realizzazione diventa un problema perché le porte ottenute sono ideali e non realizzabili (come porte AND da 500 ingressi, o porte OR da 2000 ingressi).



L'unico buon vantaggio della sintesi a 2 livelli è che il rapporto tra Area (dei letterali) e Ritardo (cammino critico¹) è direttamente proporzionale, quindi: diminuendo l'area attraverso la minimizzazione della funzione automaticamente si minimizza anche il ritardo.



Una porta AND e un porta OR hanno un certo ritardo T che varia a seconda di come si costruiscono le porte.

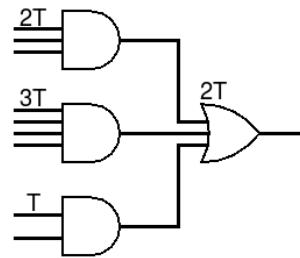
Per esempio si prende una realizzazione a 2 livelli dove ci sono tre porte AND: la prima con 3 ingressi, la seconda con 4 ingressi e la terza con 2 ingressi. Successivamente è

¹Il cammino critico è il percorso più lungo (quindi attraversa il maggior numero di porte) che va da un ingresso all'uscita

presente una singola porta OR a 3 ingressi.

Il ritardo della prima e seconda porta AND non è solo T, perché le porte AND con più ingressi possono essere viste come porte AND a 2 ingressi in cascata, pertanto: la prima porta AND a 3 ingressi equivale a 2 porte AND con 2 ingressi e costerebbe 2T, mentre la seconda porta AND a 4 ingressi viene vista anche come la cascata di 3 porte AND a 2 ingressi con un costo di 3T.

Il cammino critico del circuito descritto è il massimo del costo delle porte AND, 3T, più il costo della porta OR, 2T, per un totale di 5T.



Tramite la regola di assorbimento vengono eliminati degli ingressi, e a volte pure delle intere porte, ecco allora che il rapporto tra area e ritardo è direttamente proporzionale.

4.2 Sintesi a N livelli

La **sintesi a N livelli** rinuncia al vincolo di avere una serie di porte AND e una porta OR, ma assume il vincolo di avere una forma generica e non più precisa (come la sintesi a 2 livelli).

Il primo svantaggio a cui si incorre è che non esistono algoritmi esatti: tutti gli algoritmi per eseguire la minimizzazione a N livelli sono approssimati²

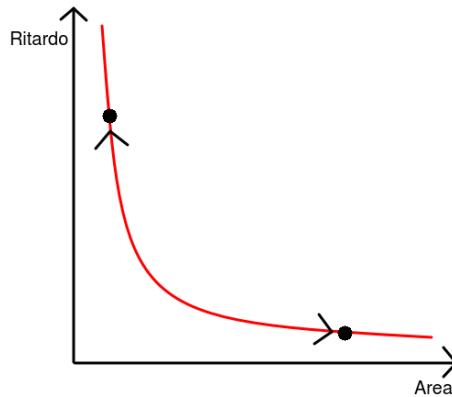
Cambia anche la relazione tra area e ritardo: **diventa inversamente proporzionale**: se si diminuisce l'area si aumenta il ritardo, al contrario se si diminuisce il ritardo aumenta l'area.

Servirà quindi trovare il giusto equilibrio tra area e ritardo per il circuito da dover realizzare.

Questo rapporto inversamente proporzionale risulta essere un problema, perché effettuare una minimizzazione significherebbe aumentare il ritardo. Allora per prima cosa bisogna cercare di ottimizzare l'area e successivamente il ritardo.

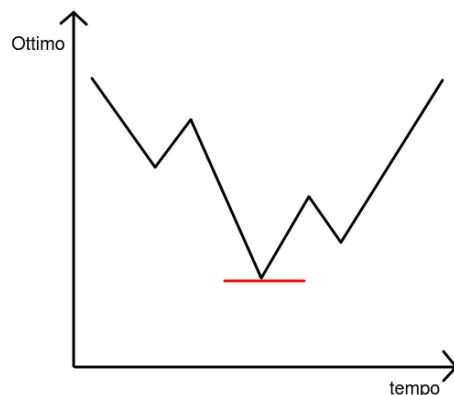
Un secondo problema è trovare la soluzione migliore. Si può ricercare una **soluzione**

²Per "approssimato" si intende che l'algoritmo trova una soluzione, ma non è detto che sia quella minima.



ottima locale dedicando un certo tempo al problema, continuando ad applicare gli algoritmi di ottimizzazione per poi distruggere la soluzione e riapplicarli, fino a quando il tempo dedicato non scade.

Trovato il miglior ottimo locale nell'arco di tempo dedicato non dà la certezza che sia il miglior ottimo assoluto, ma non si può nemmeno aspettare all'infinito. Ecco perché per la ricerca dell'ottimo locale si va a tentativi.



Per trovare il livello da utilizzare (andando sempre a tentativi) si ha bisogno di due algoritmi:

1. **Algoritmi di ottimizzazione** che permettono di scendere.
2. **Algoritmi di ristrutturazione**.

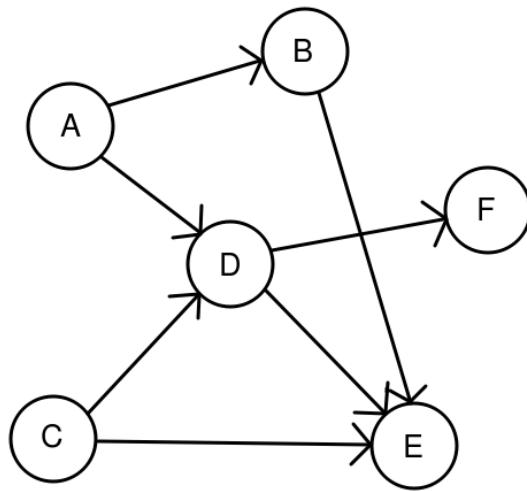
Questo processo termina quando scade il tempo dedicato alla ricerca del livello ottimale. Inoltre per poter lavorare con questi due algoritmi non si utilizza più il **modello booleano**, dato che esprime solo la funzione e non la struttura del circuito.

4.2.1 Modello Network

Al posto del **modello booleano** subentra il **modello network**, che al contrario del primo è in grado di esprimere la topologia del circuito.

Questo modello si basa su un grafo (che contiene nodi e archi) di tipo **DAG** (*Direct Acyclic Graph*)^{3]}:

- **NODI:** corrispondono ai cerchi del grafo, inoltre ogni nodo produce un segnale rappresentato da un nome. Ogni nodo può rappresentare una funzione di $f(B^n) \rightarrow B$, e quel valore può essere direzionato in altri nodi.
- **ARCHI:** corrispondono alle linee che direzionano un nodo all'altro.



Proprietà sui **DAG**:

1. Ci sono alcuni nodi che presentano solo archi in uscita, come il nodo A e C dell'esempio: rappresentano l'input del circuito.
2. Alcuni nodi possono presentare solo archi in entrata, come il nodo F ed E dell'esempio: rappresentano l'output del circuito.

³"Direct" perché gli archi sono unidirezionali; "Acyclic" perché non esistono cicli nel grafo, ovvero partire da un nodo e poter tornare su esso

Su questo **modello network** vengono applicati gli algoritmi di ottimizzazione e ristrutturazione:

- **Alg. ottimizzazione:** lavorano sui singoli nodi, lasciando inalterata la struttura della rete.
- **Alg. ristrutturazione:** lavora modificando la rete, lasciando inalterati i nodi presenti.

-Algoritmi di ottimizzazione:

Con questi algoritmi il risultato visibile sarà una diminuzione dei letterali, mentre i nodi rimangono quelli che sono.

Sono a disposizione 2 algoritmi:

1. **Simplify:** applicazione di Q&MC a tutti i nodi: vengono minimizzati dal punto di vista della sintesi a 2 livelli.
2. **Full_Simplify:** evoluzione di Q&MC: realizzata tramite una struttura dati chiamata **BDD** (*Binary Decision Diagram*): rappresenta funzioni booleane che nella maggioranza dei casi ha una dimensione in nodi non esponenziale con il numero di variabili.

Permette di semplificare ogni singolo nodo rispetto quelli vicini, calcolando come quelli vicini influenzano quel singolo nodo, in particolare:

- **CDC** (*Controllability Don't Care*): valori di ingresso di un nodo che non si presentano a quel nodo.
- **ODC**: (*Osservability Don't Care*): valori di ingresso di un nodo che producono un'uscita che non serve al risultato finale.

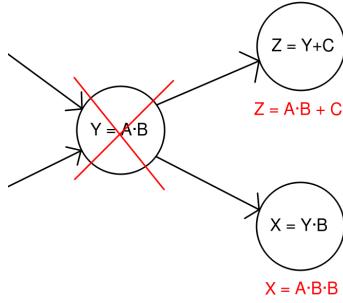
Entrambi questi due insiemi di don't care vengono eliminati.

-Algoritmi di ristrutturazione:

Con questi algoritmi il risultato visibile sarà una modifica dei nodi, con la possibilità di aumentare i letterali.

- **Sweep:** controlla all'interno della network tutti i nodi e verifica se l'uscita di un nodo è almeno usata da un altro nodo (se quindi trova un'utilità): elimina i nodi inutili.
- **Eliminate:** prende un nodo e lo sostituisce all'interno di chi lo usa, in questo modo diminuisce i nodi ma aumenta i letterali.

Se questo comando viene lanciato in modo incontrollato ha il rischio di riempire la memoria. Ecco un esempio di **Eliminate**:



- **Resub:** aumenta il numero dei nodi scomponendo un nodo in nodi più piccoli.
- **FX:** svolge lo stesso lavoro della **Resub**, ma solo quando una parte è comune a due nodi.

-Script in SIS:

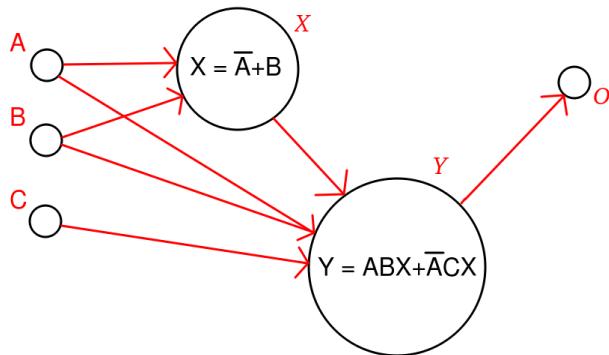
I comandi elencati in precedenza vengono utilizzati tramite gli **script**, cioè delle sequenze di comandi.

In particolare il migliore **script** creato è lo **script.rugged**: una volta lanciato trova una soluzione, se rilanciato nuovamente trova un ulteriore soluzione che può essere migliore o peggiore della precedente, e così via fino a quando non finisce il tempo dedicato alla ricerca della soluzione migliore.

Ma come si trova il giusto equilibrio tra ritardo o area?

Se non esistono vincoli precisi si valuta il criterio di guadagno tra nodi e letterali, altrimenti la concentrazione è sul numero di nodi imposto dal vincolo.

L'ultimo comando dello **script.rugged** è la **Full_Simplify**. Esempio di **Full_Simplify**:



Il comando **eliminate** in questa rete non riesce a minimizzare i nodi poiché già minimi; mentre la **Full_Simplify** parte dal nodo Y (quello più vicino all'uscita) e risulta che:

- L'insieme **OCD** del nodo Y è vuoto perché è collegato direttamente all'uscita, quindi tutti i suoi valori di ingresso ABX e $\bar{A}CX$ producono un risultato sull'uscita.
- Per l'insieme **CDC** il discorso cambia: il comando va a costruire questo insieme andando indietro nei nodi che producono gli ingressi per Y e analizzandoli al loro interno: $X = \bar{A} + B$

A	B	X
0	0	1
0	1	1
1	0	0
1	1	1

I valori che non possono essere prodotti dalla tabella appena descritta sono:

A	B	X
0	0	0
0	1	0
1	0	1
1	1	0

Questi valori non rappresentabili raffigurano l'insieme di valori **CDC** del nodo Y ottenuti negando la funzione di X.

La **Full_Simplify** fa tutto ciò negando il **BDD** di X, costruisce tutto ciò che non può produrre X e compone tra di loro, eseguendo varie operazioni a seconda di com'è fatta la rete, i **BDD** dei vari nodi precedenti per arrivare a produrre ciò che agli ingressi di Y non può arrivare (il **CDC**).

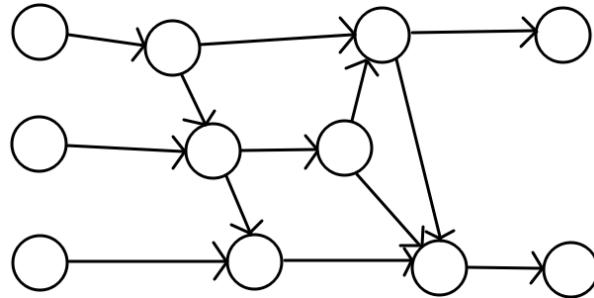
4.2.2 Technology Mapping

Il **Technology Mapping** è il processo per cui si trasformano le **porte logiche ideali** in **porte logiche reali**.

Queste **porte logiche reali** sono porte logiche di una **libreria tecnologica** fornite dal produttore di silicio, oppure sono tutte le **LUT** di una **FPGA**.

Quello che bisogna fare è trovare il minor numero di porte tecnologiche della libreria che realizzino il circuito.

Viene dato un grafo **DAG** di partenza che rappresenta una rete e tanti grafi che rappresentano le porte di libreria. L'obiettivo è quello di trovare il **Technology Mapping** ideale, ovvero il problema chiamato **problema delle clique**: utilizzare il minor numero di grafi della libreria che coprono i nodi del grafo di partenza.



La complessità del **problema delle clique** è esponenziale, dunque non fornisce una soluzione esatta.

È necessario utilizzare metodi alternativi per risolvere i problemi NP:

1. Usare **algoritmi greedy**, ossia scegliere le soluzioni che si ritengono più opportune.
2. Usare l'algoritmo **Branch and Bound** provando ogni combinazione.
3. Ancora meglio è ridurre un problema NP completo a un problema non più NP completo. Questa trasformazione non fa sparire nel nulla la complessità, ma fa guadagnare in alcuni aspetti e ne peggiora altri.

Sebbene tutti e tre i metodi siano validi quello con il quale si approccerà è il terzo.

Il grafo DAG viene scomposto in un insieme di **alberi**⁴ passando così a una **foresta di alberi** e aumentando il numero di nodi (è in questo modo che si conserva la complessità: aumentando il numero di nodi).

La trasformazione da grafi DAG in insieme di alberi avviene anche per la libreria tecnologica.

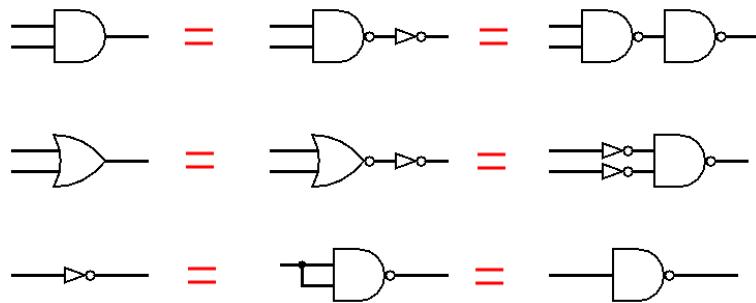
Cambia il problema che non è più il **problema delle clique**, ma diventa trovare il minor numero di alberi di libreria che coprono l'albero dato, questa operazione si chiama **Tree Mapping**.

Tutto ciò ha fatto guadagnare in meglio che il problema non sia più NP completo, ma allo stesso tempo ha peggiorato l'ottimalità iniziale.

Per eseguire l'algoritmo di **Tree Mapping** si deve garantire che i nodi del circuito di partenza siano uguali ai nodi del circuito delle porte logiche in libreria.

Una qualsiasi funzione booleana può essere descritta mediante porte AND, OR e NOT. Grazie al **Teorema di De Morgan** è possibile trasformare le porte AND, OR e NOT rispettivamente in NAND seguito da NOT, NOR seguito da NOT ,NAND con medesimo valore di ingresso:

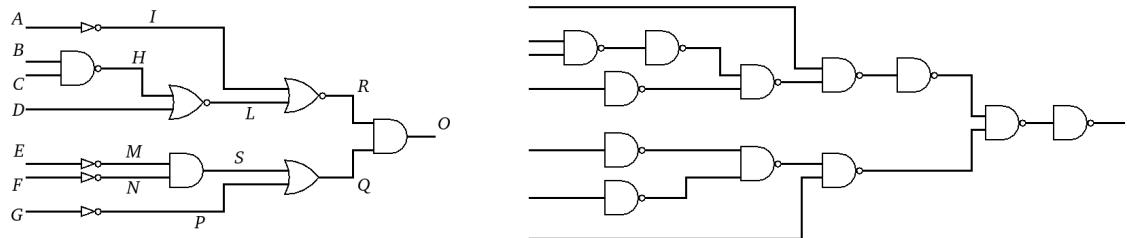
⁴Un **albero** è un grafo con alcune caratteristiche: ogni nodo dell'albero ha un solo predecessore.



Quindi tutte le porte logiche possono essere riscritte con un NAND (operatore universale), perciò i nodi delle librerie saranno ricondotti ad alberi composti da NAND.

Esempio:

A sinistra viene dato un circuito qualsiasi, mentre a destra viene a trasformato con i NAND.



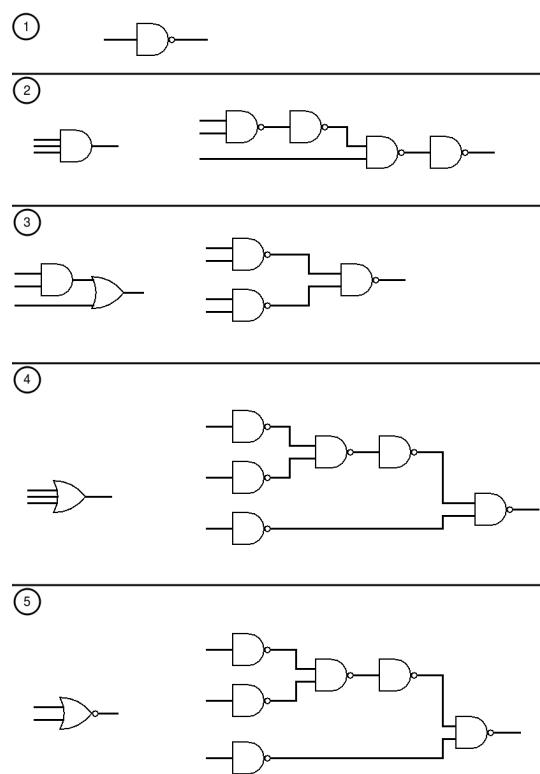
Viene fornita poi anche una libreria di porte con la quale coprire il circuito trasformato precedenza in NAND.

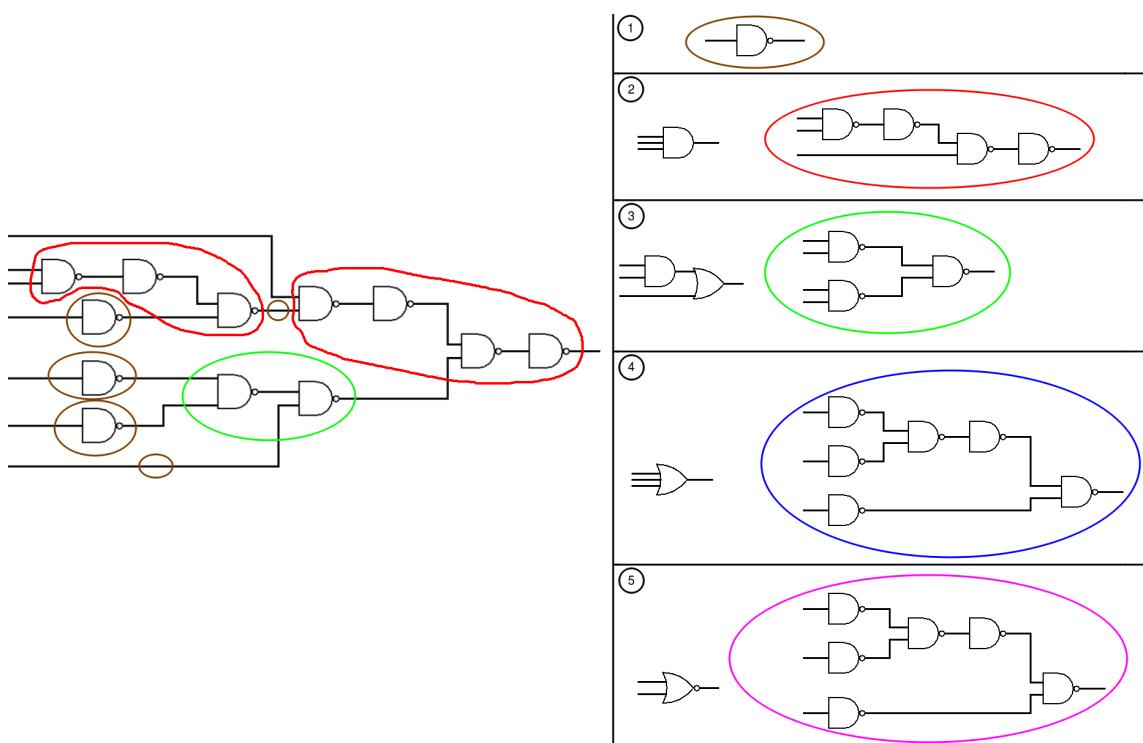
A sinistra ci sono le porte e a destra il loro equivalente in NAND:

Si parte dall'uscita andando a ritroso per coprire il maggior numero di porte.

A destra c'è la libreria e a sinistra la copertura:

Più è ampia la libreria tecnologica e meno sono le perdite durante la fase di mapping.





Da reti combinatorie a circuiti sequenziali

In precedenza si è visto come poter rappresentare un sistema digitale attraverso il modello $f(B^n) \rightarrow B^m$.

In modo sottinteso c'è sempre stata un'ipotesi, cioè quella che in ogni istante t si possono calcolare le uscite: $O_t = f(I_t)$. Ma i sistemi digitali non funzionano con solo degli ingressi in entrata in un certo istante t , serve anche lo **stato di sistema**.

Lo **stato di sistema** rappresenta in modo compresso tutti gli ingressi avvenuti nel tempo: $O_t = f(I_t, S_t)$. Queste informazioni devono essere memorizzate, perché quando arrivano nuovi ingressi diventa possibile decidere, a seconda dei vecchi ingressi, quale uscita dare.

Per rappresentare lo **stato** è necessario un nuovo modello chiamato **modello FSM** (*Finite State Machine*) e non più quello booleano che rappresenta solo gli ingressi.

$O_t = f(I_t)$ è un **circuito combinatorio** (tutti quelli visti fin'ora).

$O_t = f(I_t, S_t)$ è un **circuito sequenziale**, che compone ogni sistema digitale esistente.

-Concetti necessari per il circuito sequenziale:

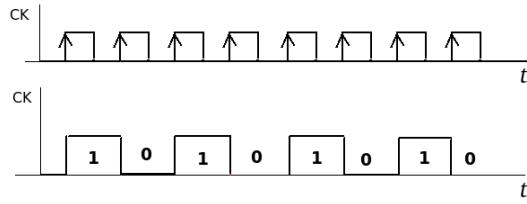
Per i **circuiti sequenziali** sono necessari due concetti fondamentali:

- Il tempo: quindi come scandire il tempo.
- La memoria: per la memorizzazione dello stato.

5.1 Tempo

Il tempo è un segnale chiamato **clock** presente in ogni **circuito sequenziale**:

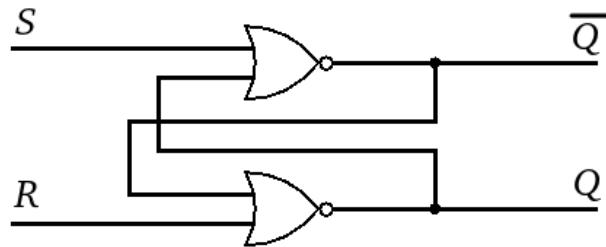
Il **segnale di clock** dal punto di vista elettronico è fatto da un quarzo, che ha una sua frequenza interna di oscillazione (a forma di sinusoide), al quale attorno ha un sistema di rilevazione che restituisce un segnale elettrico.



Questo segnale elettrico viene compattato (squadrato) e segnerà il tempo: il tempo t significa il **ciclo di clock** t che è stato scandito.

5.2 Memoria

La memoria è definita come questo circuito:



Viene creato un ciclo (cosa assolutamente proibita nei circuiti combinatori) dove l'uscita \bar{Q} va in ingresso del NOR sotto e l'uscita Q va in ingresso del NOR sopra.

S	R	Q_{t+1}	\bar{Q}_{t+1}
1	0	1	0
0	1	0	1
0	0	Q_t	\bar{Q}_t

è stabile
 è stabile
 necessita dei valori precedenti

Nel caso quindi che sia in S e R l'ingresso sia 0 allora il risultato richiede quello ricevuto in precedenza. Un circuito così non può essere combinatorio, perché dovrebbe essere sempre deterministico e alle stesse entrare restituire le stesse uscite.

Si può vedere lo stesso esempio nel tempo t che scorre:

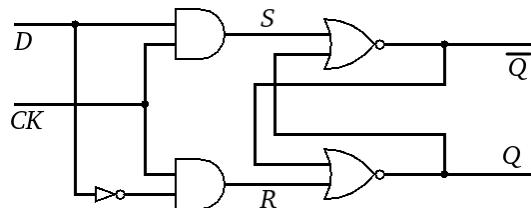
Come si può notare da due stesse entrate, 0 e 0, ci sono due uscite Q differenti: il circuito mantiene il valore precedente attraverso lo stato.

Questo circuito è dunque **sequenziale** e viene chiamato **LATCH**: circuito di base capace di memorizzare un bit.

	<i>S</i>	<i>R</i>	<i>Q</i>
	1	0	1
	0	0	1
	0	1	0
<i>t</i>	0	0	0

In questo circuito, dato così com'è, bisogna evitare che arrivino in entrata i valori 1 e 1, altrimenti si rischierebbe un'oscillazione instabile.

Allora la soluzione a questo problema è trasformare il **LATCH** in un **D-LATCH**, in questo modo avendo un nuovo segnale D e il Clock (CK) si è sicuri che successivamente non ci potrà mai essere 1 e 1 in S e R:



D	CK	S	R	Q_{t+1}	\bar{Q}_{t+1}
0	1	0	1	0	1
1	1	1	0	1	0
-	0	0	0	Q_t	\bar{Q}_t

Il **D-LATCH** è l'unità base utilizzata per memorizzare un bit.

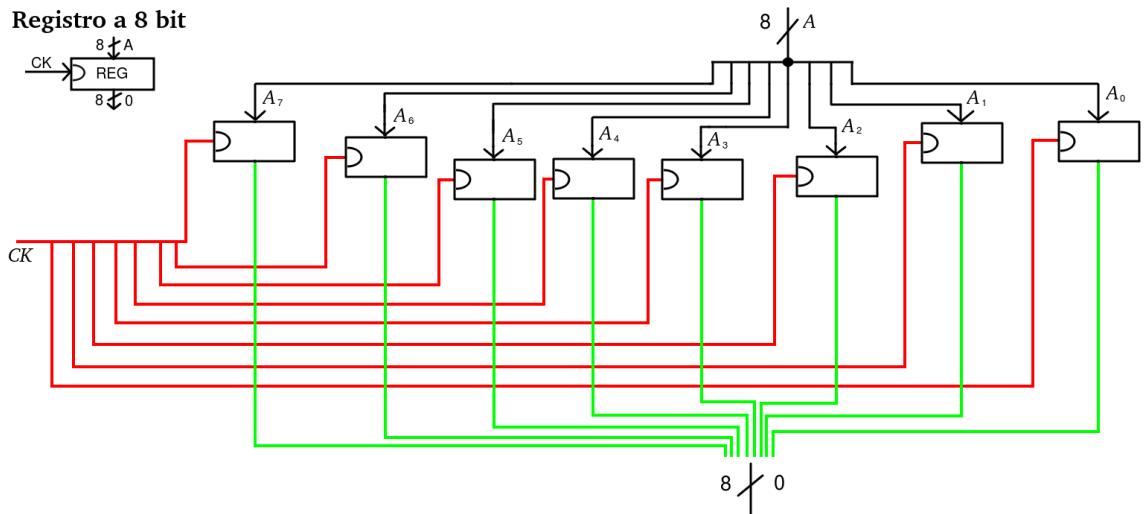
5.2.1 Registro

Prendendo più **D-LATCH** si forma un registro, quindi una memoria, in grado di memorizzare più bit.

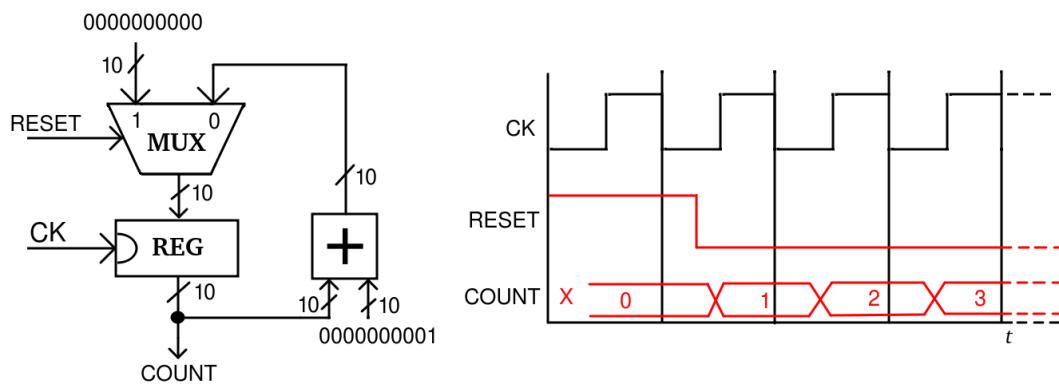
Esempio di un registro $8 \times A^1$:

Questo registro permette di memorizzare uno stato che può assumere 256 valori diversi.

¹La simbologia $N \times A$ indica che il filo A trasporta N bit, vengono poi chiamati A_0, \dots, A_N .



Esempio di contatore modulo 1024 a ogni clock:
Ogni volta che si arriva a 1024 il conteggio va avanti di 1, il contatore deve contare su un segnale chiamato **count**, inoltre deve esserci un altro segnale di **reset** che quando vale 1 allora **count** diventa 0 altrimenti il conteggio va avanti.



A destra è raffigurato il diagramma temporale:

- **CK**: sono i periodi di Clock.
- **RESET**: sono gli ingressi, parte da 1 e poi cambia in 0 e rimane costante.
- **COUNT**: sono le uscite, quando vale 0 il **CK** il **COUNT** vale un valore ignoto X , ma quando il **CK** sale a 1 il **RESET** vale 1: il Multiplexer fa passare il primo ingresso che sono tutti zeri e vengono caricati nel registro.
Nuovamente il **CK** va a 0 e viene mantenuto il valore precedente, il **RESET** scende a 0.

Successivamente il **CK** sale a 1 e il registro memorizza l'1, il Clock torna a 0 e si mantiene l'1 precedente, poi il Clock risale a 1 e il registro memorizza la somma del valore precedente +1 e va a 2, ecc.

Ma tutto ciò non può funzionare a causa del ciclo che si crea all'interno del circuito combinatorio, poiché per tutto il tempo in cui il Clock vale 1 i **D-LATCH** diventano "trasparenti" e gli ingressi vanno sull'uscita.

Dunque questo circuito parte da 0, ma alla fine del primo ciclo di Clock avrà raggiunto un valore arbitrario che rimane stabile per tutto il tempo in cui il Clock è uguale a 0 per poi essere instabile quando il Clock è uguale a 1.

Il problema in questo circuito è il **D-LATCH** perché fa passare l'ingresso sull'uscita per tutto il tempo in cui il Clock vale 1.

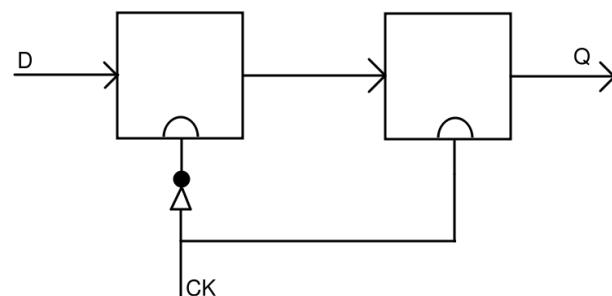
Bisogna spezzare il ciclo del circuito combinatorio, e passare da registri basati su **D-LATCH** a registri basati su **FLIP-FLOP**.

-Circuito FLIP-FLOP:

Il **FLIP-FLOP** è una coppia di **D-LATCH** messi uno dietro l'altro.

In questo modo quando il Clock vale 0 il primo **D-LATCH** fa passare il valore D, mentre il secondo **D-LATCH** blocca il valore e mostra quello precedente.

Quando poi il Clock vale 1 il primo **D-LATCH** blocca il valore D, mentre il secondo **D-LATCH** fa uscire il precedente valore lasciato in attesa.

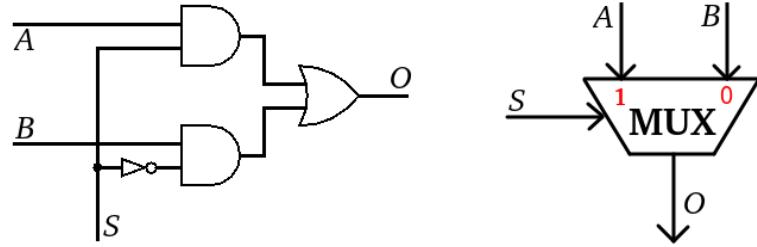


Usando dei **FLIP-FLOP** si spezza il ciclo dell'esempio precedente.

-Circuito Multiplexer:

Il **circuito Multiplexer** (circuito selettore) è un **circuito combinatorio**:

Il suo funzionamento consiste che quando i segnali di selezione S vale 1 l'ingresso A va sull'uscita O, mentre quando S vale 0 l'ingresso B va sull'uscita O.



5.3 Macchina a stati finiti (FSM)

Una macchina a stati finiti è un'estensione del circuito combinatorio in cui si individua una quintupla di elementi:

$$\text{FSM} = \langle S, I, O, \delta, \lambda \rangle$$

- **S** : Insieme degli stati: la cardinalità deve essere $|S| \geq 1$
Se $|S| = 1$ allora è un circuito combinatorio.
- **I** : Insieme dei simboli in ingresso: la cardinalità deve essere $|I| \geq 0$ (2^n ingressi)
Se $|I| = 0$ allora è una **macchina autonoma** che si evolve dai suoi stati.
- **O** : Insieme dei simboli d'uscita: la cardinalità deve essere $|O| \geq 1$ (2^m ingressi)
- δ : Funzione dello stato prossimo: $\delta : S_{attuale} \times I \rightarrow S_{prossimo}$
- λ : Funzione d'uscita, che può essere di due tipi a seconda della macchina.

-Tipologie di FSM:

Esistono due tipologie di macchine a stati finiti:

- **Mealy**: l'uscita è funzione dello stato di ingresso: $\lambda : S \times I \rightarrow O$
- **Moore**: l'uscita è in funzione dello stato: $\lambda : S \rightarrow O$

A volte viene impostato uno stato iniziale S_0 inserito come specifica.

5.3.1 Rappresentazioni della FSM

Per rappresentare una **FSM** si hanno due modi:

1. **STG**: *State Transition Graph*
2. **STT**: *State Transition Table*

Queste due rappresentazioni sono equivalenti, forniscono la stessa identica informazione completa della macchina.

Data una specifica in linguaggio naturale bisogna arrivare a una **STG** o una **STT**. Possono esistere più **STG** o **STT** che soddisfano la specifica.

Esempio di **STT con Mealy**:

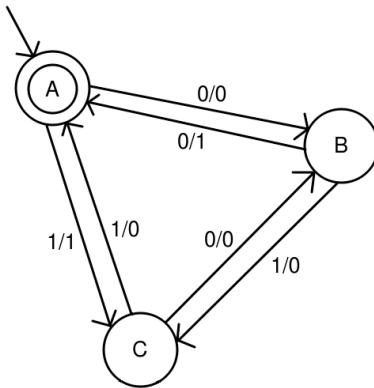
	0	1
• A	B/0	C/1
B	A/1	C/0
C	B/0	A/0

Prende gli stati (A, B, C) e simboli di ingresso.

Il formato è S/O (Stato/Uscita).

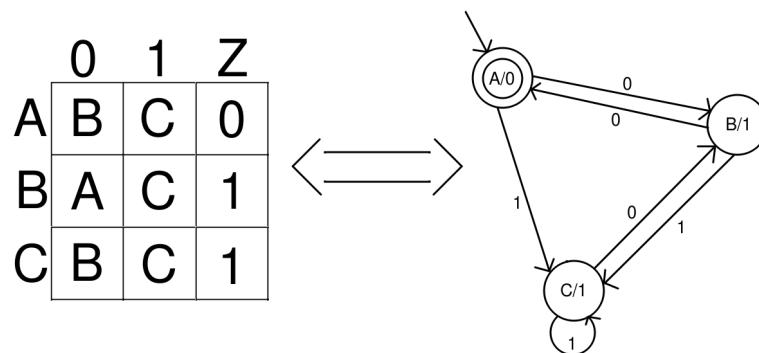
Esempio di **STG con Mealy**:

Prendendo lo stesso esempio fatto con la **STT** ecco la rappresentazione in grafo:



Lo stato iniziale della **STT** ha il pallino •, nella **STG** ha un doppio cerchio oppure una freccia d'inizio.

Esempio (differente dal precedente) di **STG e STT con Moore**:

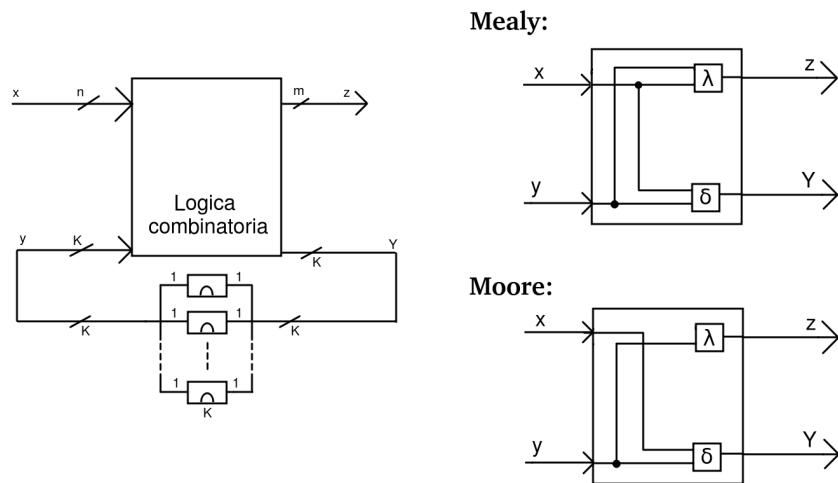


Cambia la tabella perché l'uscita è funzione dello stato, quindi non si può più mettere insieme agli ingressi, ha bisogno di una colonna a parte (colonna Z). Nelle singole cellette ci sarà solo lo stato prossimo senza l'uscita.

Per quanto riguarda i grafi nei nodi è presente lo stato con l'uscita relativa, mentre negli archi è presente solo l'entrata che specifica che in che stato andare.

5.3.2 Modello di Huffman

Per rappresentare a livello fisico **Mealy** e **Moore** si usa il **Modello di Huffman**: modello strutturale che prevede la composizione della FSM in un blocco di logica combinatoria.



5.3.3 Flusso di sintesi di FSM

1. Interpretazione della specifica in linguaggio naturale (richiesta umana di progettazione).
2. Rappresentazione come **STG**.
3. Conversione della **STG** in **STT**.
4. Minimizzazione degli stati.
5. Codifica degli stati.
6. Scelta degli elementi di memoria. *
7. Minimizzazione combinatoria. *
8. Copertura su hardware. *

* Questi punti non verranno trattati.

5.4 Minimizzazione degli stati

La **minimizzazione degli stati** consiste nel diminuire al minimo il numero degli stati iniziali, ottenuti dalla **STG** (trasformata poi in **STT**), in modo che il comportamento della **FSM** sia identico a quello di partenza.

Per fare ciò è necessario sapere quando due stati sono **equivalenti** (o **indistinguibili**):

Definizione equivalenza:

Due stati S_i e S_k sono *equivalenti* (o *indistinguibili*) $S_i \sim S_k$ se e solo se si verifica che per una qualunque sequenza di ingresso $I_\alpha (= i_1, \dots, i_k)$, le sequenze d'uscita poste alla macchina sono identiche.

Questa relazione di **equivalenza** supporta 3 proprietà:

1. **Riflessiva:** $S_i \sim S_i$
2. **Simmetrica:** $S_i \sim S_j \rightarrow S_j \sim S_i$
3. **Transitiva:** $S_i \sim S_j \wedge S_j \sim S_k \rightarrow S_i \sim S_k$

Da queste proprietà viene introdotta una partizione dell'insieme degli stati: **partizione di equivalenza**.

Si indica mediante il π , e sarà l'insieme di un certo numero di blocchi:

$\pi = \{B_1, \dots, B_r\}$, dove B_i è chiamato **blocco** e corrisponde a $\{S_j\}$ (stati fra loro equivalenti).

Essendo una **partizione** allora l'unione dei **B** dia **S**:

$$\cup_i B_i = S \text{ (insieme complessivo degli stati)}$$

Invece con l'intersezione fra un blocco e un altro deve essere nullo:

$$\forall_{i,j} B_i \cap B_j \neq \emptyset$$

Devono quindi essere insiemi disgiunti, se non lo fossero, dunque esiste l'intersezione tra blocchi, per la proprietà transitiva sarebbero tutti lo stesso blocco.

Con la minimizzazione si passa dalla cardinalità di **S** a quella di π : $|S| \rightarrow |\pi|$.

La **partizione di equivalenza** risulta unica, pertanto esiste una sola soluzione di minimizzazione.

-Algoritmo di Paull-Unger:

L'**algoritmo di Paull-Unger** ha l'obiettivo di riconoscere se due stati sono equivalenti. In particolare è un algoritmo ricorsivo che definisce le condizioni sotto le quali due stati sono equivalenti.

Definizione:

$S_i \sim S_j$ se e solo se $\forall i_a \in I$ si verificano:

1. $\lambda(S_i, i_a) = \lambda(S_j, i_a)$: se hanno la stessa uscita a parità di ingresso.
2. $\delta(S_i, i_a) \sim \delta(S_j, i_a)$: equivalenza dello stato prossimo.

Questo algoritmo termina in 2 casi:

1. $S_i \not\sim S_j$, se:

- Le uscite sono diverse per almeno un ingresso.
- $\exists i_b \in I$ tale che $\delta(S_i, i_b) \not\sim \delta(S_j, i_b)$ (gli stati prossimi non sono equivalenti).

2. $S_i \sim S_j$ se $\forall i_a \in I$ è $\delta(S_i, i_a) \sim \delta(S_j, i_a)$

Caso particolare in cui è soddisfatta la relazione di equivalenza:

$\delta(S_i, i_a) = S_i$ e $\delta(S_i, i_a) = S_j$, oppure $\delta(S_i, i_a) = \delta_i$ e $\delta(S_j, i_a) = \delta_j$.

-Metodo con tabella delle implicazioni:

Dato l'insieme di stati $S = \{A, B, C, D, E\}$ si mettono in colonna tutti gli stati a partire dal secondo (in questo esempio B) e in riga tutti gli stati dal primo fino al penultimo (ultimo escluso):

B			
C			
D			
E			

	A	B	C	D
--	---	---	---	---

Copertura tabella:

- X: coppia non equivalente.
- \sim : coppia equivalente.
- S_1, S_2 : coppie degli stati prossimi ancora da verificare (vincoli).

Esempio FMS Mealy:

	1	0	B	C
A	B/0	A/0	X	X
B	C/0	A/0		B,C
C	B/0	D/1	A,E	A,E
D	B/0	E/0	X	X
E	B/0	D/1	X	X

	A	B	C	D
			\sim	X

Si prendono in considerazione prima le uscite e poi gli stati prossimi, se entrambi si equivalgono allora viene inserito \sim altrimenti **X**.

I confronti sono fatti tra i membri della tabella a sinistra: **A** con **B**, **A** con **C**, **A** con **D**, **A** con **E**, **B** con **C**, **B** con **D**, **C** con **D**, **C** con **E**, **D** con **E**:

- **A** con **B**: Le uscite sono 0 da entrambi, quindi si controllano gli stati prossimi di cui **B** con **C** diventa un **vincolo** (perciò bisogna controllare se **B** e **C** sono equivalenti) mentre **A** con **A** sono equivalenti per la proprietà riflessiva.
- **A** con **C**: Le uscite non sono uguali, quindi si assegna da subito la **X** di non equivalenza.
- **A** con **D**: Le uscite sono 0 da entrambi, quindi si controllano gli stati prossimi di cui **B** con **B** per proprietà riflessiva sono equivalenti mentre **A** con **E** diventa un **vincolo**.
- **A** con **E**: Le uscite non sono uguali, quindi si assegna da subito la **X** di non equivalenza.
- **B** con **C**: Le uscite non sono uguali, quindi si assegna da subito la **X** di non equivalenza.
- **B** con **D**: Le uscite sono 0 da entrambi, quindi si controllano gli stati prossimi di cui sia **B** con **C** e sia **A** con **E** diventano **vincoli**.
- **B** con **E**: Le uscite non sono uguali, quindi si assegna da subito la **X** di non equivalenza.
- **C** con **D**: Le uscite non sono uguali, quindi si assegna da subito la **X** di non equivalenza.
- **C** con **E**: Le uscite sono uguali da entrambe le parti, quindi si controllano gli stati prossimi e per la proprietà riflessiva sia **B** con **B** che **D** con **D** sono equivalenti, perciò si assegna il simbolo di equivalenza \sim .
- **D** con **E**: Le uscite non sono uguali, quindi si assegna da subito la **X** di non equivalenza.

Il passo successivo è quello di controllare i **vincoli** inseriti nelle celle:

se la cella del vincolo è equivalente allora per proprietà transitiva si inserisce \sim , se la cella del vincolo non è equivalente per proprietà transitiva allora si inserisce la **X**.

I **vincoli** presenti indicano le celle **B,C** e **A,E** che non sono equivalenti, di conseguenza si assegna **X** alle celle lasciate in sospeso:

B	X			
C	X	X		
D	X	X	X	
E	X	X	\sim	X
A	B	C	D	

Una volta completata la **tavola delle implicazioni** si creano i blocchi di stati che si equivalgono:

	0	1
A	B/0	A/0
B	C/0	A/0
C	B/0	D/1
D	B/0	E/0
E	B/0	D/1

Formati i nuovi blocchi condensando insieme gli eventuali stati che si equivalgono si può creare la nuova STT:

	1	0
α	$\beta/0$	$\alpha/0$
β	$\gamma/0$	$\alpha/0$
γ	$\beta/0$	$\delta/1$
δ	$\beta/0$	$\gamma/0$

Un caso particolare è la **circularità di vincolo** dove i vincoli riportano ad altri vincoli legandoli tra loro si può considerare quelle celle equivalenti \sim :

$S_i \sim S_j$ dipende da $S_k \sim S_m$ che dipende da... $S_i \sim S_j$.

5.4.1 Minimizzazione di macchine parzialmente specificate

Con le macchine parzialmente specificate derivano delle conseguenze:

- Esistono soluzioni multiple, cioè esistono diverse assegnazioni dei **dc** (don't care) che producono diverse equivalenze.
- Nasce il concetto di **compatibilità**: esiste un'assegnazione **dc** tali che gli stati sono equivalenti:

Due stati S_j e S_k sono compatibili ($S_j \nu S_k$) se $\forall i_i \in I$:

1. Ovunque siano entrambi specificati, vale $\lambda(S_k, i_i) = \lambda(S_j, i_i)$: uguaglianza d'uscita.
2. Ovunque siano entrambi specificati, vale $\delta(S_k, i_i) \nu \delta(S_j, i_i)$: equivalenza degli stati prossimi.

Altra definizione di compatibilità:

Esiste un'assegnazione **dc** tali che gli stati sono equivalenti.

Inoltre nel caso generale non vale la proprietà transitiva con la **compatibilità**, quindi in caso di **circularità dei vincoli** non si possono considerare compatibili.

Dunque l'**algoritmo di Paull-Unger** va esteso poiché si giunge a una tabella delle implicazioni con vincoli che non si possono soddisfare tramite proprietà transitiva.

Esempio:

	1	0		
A	E/0	A/0	D,E	
B	D/0	B/0	ν	D,E
C	E/-	C/-		
D	A/1	A/1	X	X
E	A/-	D/-	A,B	X
			A,C	A,E
			B,C	A,B
			A	B
			C	D

Si costruisce la tabella delle implicazione nel medesimo modo visto per le macchine specificate, con la sola differenza della **compatibilità** definita da ν :

- Il confronto tra A e C per gli ingressi non viene effettuato perché è presente il **dc**, quindi si passa agli stati prossimi dove per proprietà riflessiva E ed E sono equivalenti, mentre A e C sono gli stati che si stanno valutando quindi non serve inserirlo come vincolo.
- Il confronto tra A ed E, B ed E, C ed E, infine D ed E, producono in un caso non equivalenza mentre negli altri presentano vari vincoli.

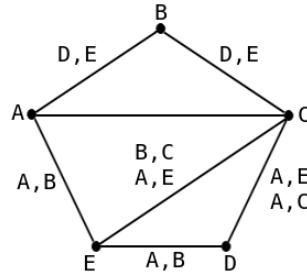
La compatibilità **A,C** non preclude, come con l'equivalenza, l'eliminazione del vincolo presente nella cella **C,D** perché si potrebbe decidere che quella compatibilità non si trasformi in una equivalenza.

Perché se si rendesse **A,C** equivalente potrebbe darsi che se ne perdano altre di equivalenze in altre celle, quindi: quando c'è una compatibilità non sempre è la miglior scelta renderlo equivalente.

Pertanto dalla tabella delle implicazioni si ottiene un insieme di potenziali equivalenze, questo insieme viene chiamato **classi di compatibilità**.

La **classe di compatibilità** è un insieme di stati compatibili e i loro vincoli. Ciò che interessa sono le **classi massime**: sottografi completi massimi del **grafo di compatibilità**.

Il **grafo di compatibilità** permette di individuare le classi massime; grafo delle tabelle precedenti:



Da questo grafo si trovano 3 **classi di compatibilità massima**:

1. $\{A, B, C\}$ è una classe di compatibilità massima con **D,E**.
2. $\{C, D, E\}$ è " " con **A,B/A,E/B,C/A,C**.
3. $\{A, C, E\}$ è " " con **A,B/A,E/B,C**.

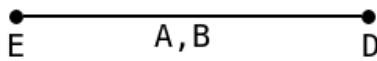
Da questa serie di classi si deve decidere chi sarà equivalente e chi no. Per scegliere si usa il **metodo greedy**:

1. Si sceglie una classe di compatibilità.
2. Vengono rimossi dal grafo tutti i nodi della classe scelta e i lati rimasti (chiamati "orfani"). Si controlla se i vincoli sono soddisfatti dalla scelta:
 - Nel caso non lo siano si torna al punto 1 ripristinando il grafo.
 - Nel caso lo siano si marca la classe come un nuovo stato.
3. Se il grafo è vuoto termina l'algoritmo, altrimenti si torna al punto 1 senza ripristinare il grafo.

-Esempio:

Punto 1: viene scelta (come esempio) la classe $\{A, B, C\}$:

Punto 2: si rimuove la classe scelta insieme a tutti i lati "orfani" e rimane **ED**



Si verificano se i vincoli presenti sono soddisfacibili dalla scelta iniziale.

A,B è soddisfacibile? Sì, perché inizialmente è stata scelta la classe $\{A, B, C\}$ in cui è contenuto. Di conseguenza, dato che i vincoli sono soddisfatti, si marca la classe $\{A, B, C\} = \alpha$.

Punto 3: Dato che il grafo non è vuoto si torna al **punto 1**; si aggiornano le classi di compatibilità che sono rimaste: $\{C, D, E\}$ e $\{A, C, E\}$ (mentre $\{A, B, C\}$ è stata già scelta, quindi non disponibile).

Da quelle rimaste si tolgono le lettere della classe soddisfatta in precedenza e diventano rispettivamente: $\{D, E\}$ e $\{E\}$. Tra le due $\{E\}$ non è una classe massima e si elimina.

Ripetizione punto 1: rimane soltanto $\{D, E\}$ che viene scelta.

Ripetizione punto 2: si rimuove la classe scelta insieme a eventuali lati "orfani" e rimane un grafo vuoto. La classe $\{D, E\}$ viene marcata $\{D, E\} = \beta$

Ripetizione punto 3: dato che il grafo è vuoto l'algoritmo termina. α e β sono i nuovi stati.

5.5 Codifica degli stati

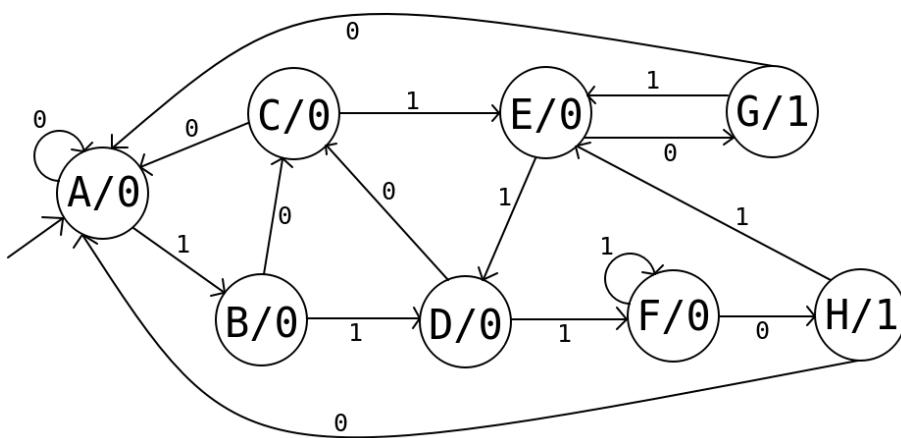
La **codifica degli stati** avviene dopo la minimizzazione della **STT** del problema dato.

Esercizio d'esempio:

Viene dato in ingresso **X** e un'uscita **Z**, dove **Z** vale 1 se e solo se la sequenza d'ingresso è stata fra quelle definite da **1-10**.

Esempio sequenza: data 010110101110101 si trovano le seguenti sequenze utili 0101 1010 111010 1, e viene restituito **Z** per ogni valore: 000000010001010 (mostra 1 appena trova la sequenza valida e 0 fino a quando non l'ha trovata).

-**STG** della richiesta:



-Conversione della **STG** in **STT**:

	$x = 0$	$x = 1$	z
A	A	B	0
B	C	D	0
C	A	E	0
D	C	F	0
E	G	D	0
F	H	F	0
G	A	E	1
H	A	E	1

Una volta conclusa la conversione da **STG** in **STT** si effettua la minimizzazione degli stati (visto nel precedente paragrafo), ma che ora non verrà effettuata e si passa direttamente alla **codifica degli stati**.

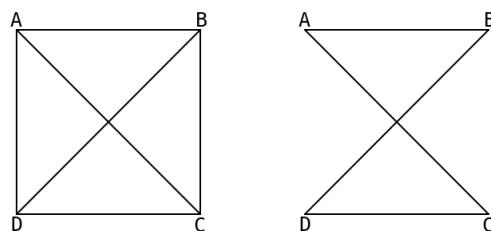
Le codifiche possibili sono $\frac{2^k!}{(2^k - |S|)!}$, e risultano troppe combinazioni nel caso si vogliano provare tutte quante. Si utilizza una **euristica** per trovare delle possibili codifiche, si basa su 3 criteri²

1. È opportuno che due stati S_i e S_j che, a parità di ingressi, hanno stessi stati prossimi, allora abbiano **codifiche adiacenti**: si minimizza δ .
2. È opportuno che due stati S_i e S_j che, a parità di ingressi, hanno stessa uscita, allora abbiano stesse **codifiche adiacenti**: si minimizza λ .
Per **Moore** non ha senso questo criterio.
3. È opportuno che due stati S_j e S_k tali che valgano $S_j = \delta(S_i, i_\alpha)$ e $S_k = \delta(S_i, i_\beta)$, con i_α e i_β **ingressi adiacenti**, abbiano **codifiche adiacenti**.

5.5.1 Grafo delle adiacenze

Questa tipologia di grafi prevede che si disegni una figura regolare con un certo numero di nodi dipendente dal numero di stati.

Esempio con 4 stati:



Nel primo caso (sfortunato) ogni adiacenza vale uguale e non è possibile sapere quali

²Per stabilire quando due stati dovrebbero essere adiacenti (a distanza di **Hamming**).

adiacenze si devono rispettare, mentre nell'esempio a fianco bisogna rispettare **AB**, **AC**, **BD**, **CD**, prendendo tutte le codifiche possibili a distanza di Hamming unitaria e ponendo l'assegnazione.

La codifica nel secondo caso è:

AB	00	A
AC	01	B
BD	11	D
CD	10	C

La codifica del primo esempio invece è più complessa e richiede il concetto dei **pesi dei segmenti**: cioè ogni suggerimento aumenta il peso di un segmento³.

Ogni volta che si ripete un segmento suggerito il peso aumenta. Esempio:

	0	1
A	A/1	B/0
B	C/1	D/0
C	A/0	B/1
D	C/0	D/1

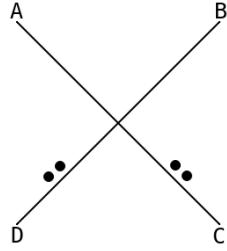
Viene applicato il **primo criterio di adiacenza** valutando: A con B, A con C, A con D, B con C, B con D e C con D:

- A con B a parità di ingressi non hanno gli stessi stati prossimi.
- A con C a parità di ingressi hanno gli stessi stati prossimi, quindi si suggerisce l'adiacenza **AC** con **peso= 2**.
- A con D a parità di ingressi non hanno gli stessi stati prossimi.
- B con C a parità di ingressi non hanno gli stessi stati prossimi.
- B con D a parità di ingressi hanno gli stessi stati prossimi, quindi si suggerisce l'adiacenza **BD** con **peso= 2**.
- C con D a parità di ingressi non hanno gli stessi stati prossimi.

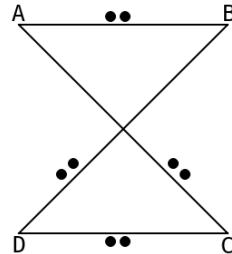
Successivamente si applica il **secondo criterio di adiacenza**:

- A con B a parità di ingressi hanno le stesse uscite, quindi si suggerisce l'adiacenza **AB** con **peso= 2**.
- A con C a parità di ingressi non hanno le stesse uscite.
- A con D a parità di ingressi non hanno le stesse uscite.
- B con C a parità di ingressi non hanno le stesse uscite.

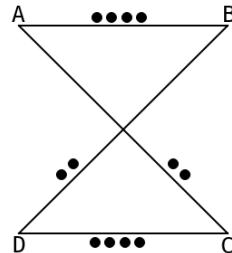
³I pesi vengono rappresentati da un pallino, se c'è un solo suggerimento è presente solo il segmento senza pallini, mentre se non ci sono suggerimenti non c'è il segmento



- B con D a parità di ingressi non hanno le stesse uscite.
- C con D a parità di ingressi hanno le stesse uscite, quindi si suggerisce l'adiacenza **CD** con **peso= 2**.



Infine si applica il **terzo criterio di adiacenza** fissandosi su una riga e suggerire adiacenza guardando agli stati prossimi per ingressi adiacenti.



Il peso finale delle adiacenze presenti è: $AB = 4$, $CD = 4$, $AC = 2$, $BD = 2$.

Tra queste adiacenze si prediligono quelle con maggiore peso al fine di poter minimizzare al meglio δ e λ .

Bisogna posizionare le adiacenze (partendo da quelle con peso maggiore) in modo che siano a distanza di Hamming, quindi:

00	A
01	B
11	
10	

AB è stato posizionato in 00 per A e 01 per B, si poteva anche scegliere 01 per A e 00 o 11 per B, basta che siano adiacenti⁴ non si può posizionare la A in 00 e B in 11, non rispetterebbero la distanza di Hamming.

Dopodiché si posiziona **CD** che ha peso maggiore di **AC** e **BD**:

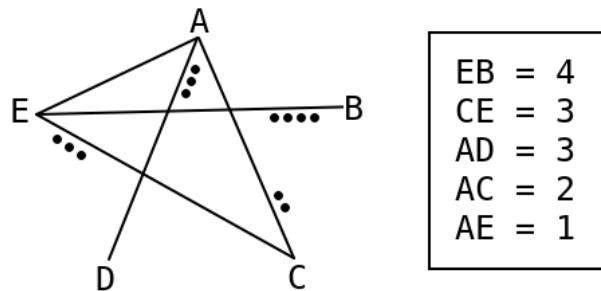
00	A
01	B
11	C
10	D

CD è stato posizionato rispettando la distanza di Hamming, ma con questa disposizione non si riesce a formare **BD** e nemmeno **AC**, quindi va cambiata la disposizione di **CD**:

00	A
01	B
11	D
10	C

Con questa nuova disposizione è stato ruotato **CD** rispettando comunque la distanza di Hamming, inoltre si possono formare le coppie **BD** e **AC** dato che rispettano la distanza.

Esempio con grafo delle adiacenze già noto:



In questo caso non si riescono a soddisfare tutte le adiacenze perché la **E** appare 3 volte e non si riesce a posizionare in modo tale che esistano le adiacenze suggerite. Quindi bisogna sacrificare le adiacenze con peso minore, e in questo caso si sacrifica **AE** che presenta un solo suggerimento:

⁴Sono indifferenti le rotazioni e gli specchiamenti.

000	E
001	B
011	
010	
110	
111	D
101	A
100	C

Con questa disposizione si soddisfano solo: **EB**, **CE**, **AD**, **AC**; si sacrifica **AE** ed è il meglio che si può ottenere.

DATAPATH

	0	1
A	E/0	A/0
B	D/0	B/0
C	E/-	C/-
D	A/1	A/1
E	A/-	A/-

$$\alpha = \{A, B, C\}$$

$$\beta = \{D, E\}$$

	0	1
α	$\beta/0$	$\alpha/0$
β	$\alpha/1$	$\alpha/1$

Se il circuito dipende da elaborazioni di dati allora non è più semplice adottare il modello per descrivere il circuito.

Preso come esempio il circuito di memoria RAM: l'approccio **FSM** comporterebbe tanti stati con medesimo comportamento (prelevano i dati dalle locazioni), ergo diventa pesante usufruire di questo modello.

Preso come altro esempio l'addizione con accumulo: in maniera analoga si creano troppo stati all'aumentare della memoria.

In linea generale non risulta ottimale l'approccio **FSM** quando si gestiscono dei dati; al contrario questo modello è molto adatto ai circuiti di controllo, dove si hanno diversi cammini di esecuzione in funzione degli ingressi.

È diverso il concetto per cui stesso cammino indipendente dagli ingressi.

Perciò l'approccio preferito per il cammino dei dati è quello **DATAPATH**. Si basa su un metodo strutturale di progettazione, mentre quello **FSM** è comportamentale:

- **Strutturale**: identifica gli elementi di computazione e la connessione tra di loro.
- **Comportamentale**: identifica l'evoluzione del sistema.

6.1 Unità funzionali

Per l'utilizzo del **DATAPATH** si introduce una libreria di componenti composta da 4 elementi:

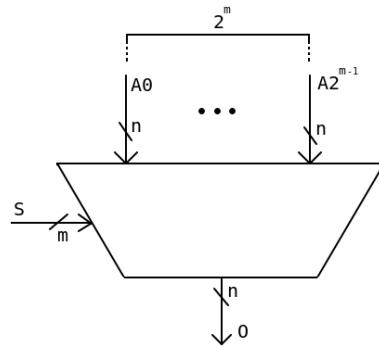
1. **Selezione**;

2. Aritmetiche;
3. Logiche;
4. Confronto.

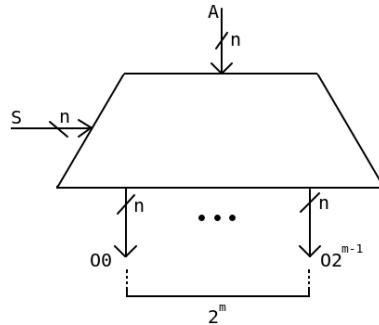
-Unità di selezione:

Queste unità si suddividono in:

- **Multiplexer:** seleziona 1 su 2^m ingressi e lo dà su **O**.



- **Demultiplexer:** seleziona 1 su 2^m uscite su cui mandare **A**.

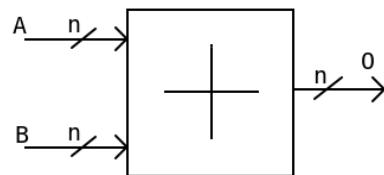
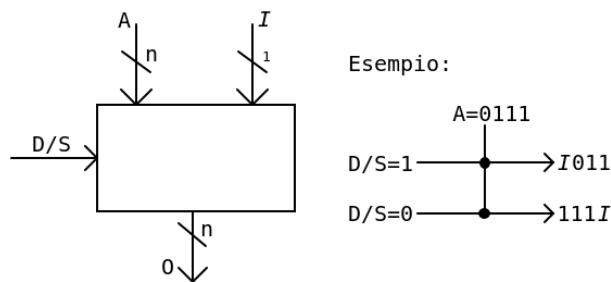
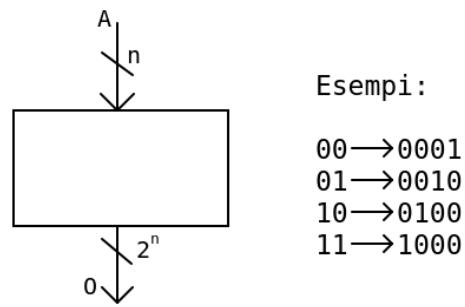


- **Decoder:** abilita il segnale su 2^n .
- **Shifter:** scala a D/S = destra/sinistra (1/0) e inserisce il bit di ingresso **I**.

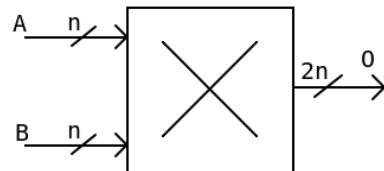
-Unità aritmetiche:

Queste unità si suddividono in:

- **Sommatore:**
La sua generalizzazione ha il **cin** e **cout**.



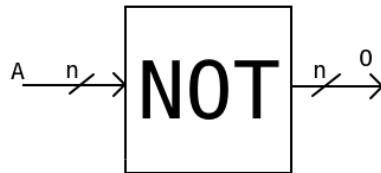
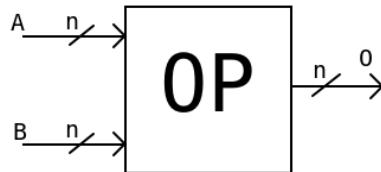
- **Moltiplicatore:**



-Unità logiche:

Queste unità si suddividono in:

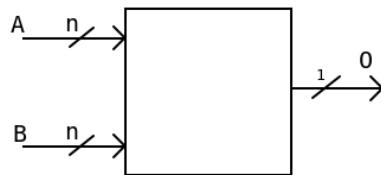
- **AND, OR, NAND, NOR, XOR, XNOR:** operazione bit a bit.
- **NOT:** caso particolare delle unità logiche.



-Unità di confronto:

Queste unità si suddividono in:

- $>, <, =, \neq, \leq, \geq$: o è vero o è falso con 1/0 in O.



6.2 Sintesi ad alto livello

La **sintesi ad alto livello** è una teoria che permette di ragionare in termini di blocchi.

Esempio **ALU** (*Arithmetic Logic Unit*):

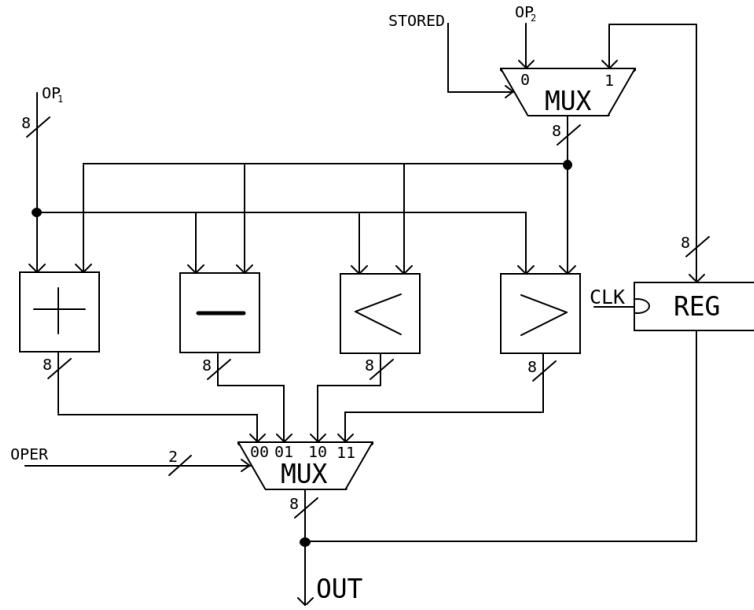
È a 8 bit, fa 4 operazioni: somma, sottrazione, minore e maggiore (in questi ultimi due restituisce il valore minore e maggiore).

Le somme vengono rappresentate in **complemento a 2** e le operazioni vengono svolte in un ciclo di clock.

Elementi a disposizione:

CLK = clock , op₁[8] / op₂[8] = input , OUT[8] = output , OPER[2] = operazioni da svolgere , stored = segnale (se vale 1 l'operazione viene eseguita tra op₁ e il contenuto del registro che memorizza l'ultimo risultato, se vale 0 viene eseguita tra op₁ e op₂).

Circuito **ALU**:



Vengono eseguite tutte le operazioni e successivamente tramite il **multiplexer** viene scelta quella da compiere. Il valore uscito si memorizza in un **registro** a 8 bit con ingresso il **clock**, altrimenti sarebbe diventato un circuito combinatorio.

Se il segnale di **stored** è 0 dal **multiplexer** passa **OP₂**, al contrario passa il valore memorizzato nel **registro**.

I blocchi della sottrazione, minore e maggiore non appartengono alla libreria del **DATAPATH**, quindi devono essere costruiti.

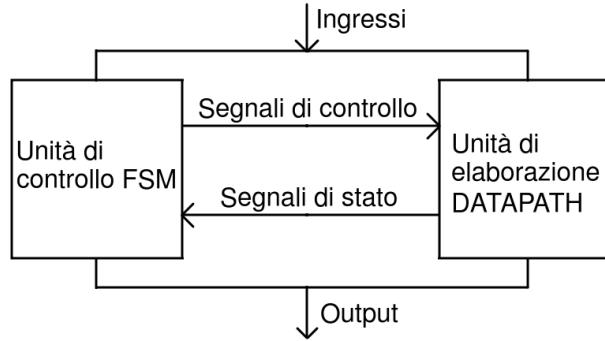
6.3 Modello FSMD

Il **modello FSMD** corrisponde all'interazione di una componente **FSM** e una **DATAPATH** usate insieme per realizzare una specifica:

Data una specifica si deve stabilire l'**interfaccia** tra FSM e DATAPATH, ovvero identificare i **segnali di controllo** (partono da FSM e arrivano al DATAPATH) e i **segnali di stato** (partono da DATAPATH e arrivano alla FSM) che si scambiano.

Esempio: Semaforo con priorità:

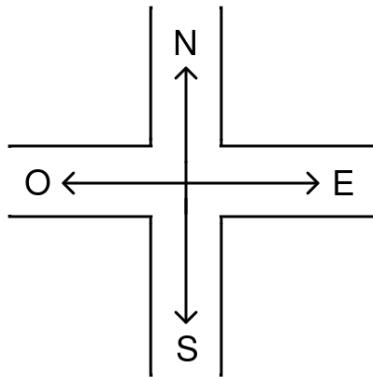
1. Esempio base con solo FSM.
2. Estensione I che richiede il DATAPATH.



3. Estensione **II** che richiede il DATAPATH, ma per cui ci sono più interfacce (quindi implementazioni).

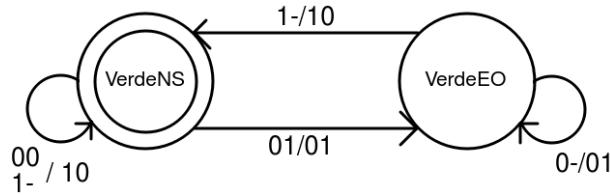
1) Esempio base:

- Diretrice **NS** ed **EO**.
- Ci sono sensori di traffico.
- I semafori danno solo verde o rosso (mai verde da entrambe le parti).
- Priorità a **NS**: se c'è traffico su **NS** a prescindere gli viene dato verde (anche se c'è traffico pesante su **EO**).



$$\begin{aligned} \text{Input} &\left\{ \begin{array}{l} \text{TrafficoNS[1]: se 1 allora traffico su NS} \\ \text{TrafficoEO[1]: se 1 allora traffico su EO} \end{array} \right. \\ \text{Output} &\left\{ \begin{array}{l} \text{LuceNS[1]: se 1 allora verde su NS} \\ \text{LuceEO[1]: se 1 allora verde su EO} \end{array} \right. \end{aligned}$$

Codifica: *trafficoNS traffico EO / LuceNS Luce EO*
Spiegazione delle direzioni:



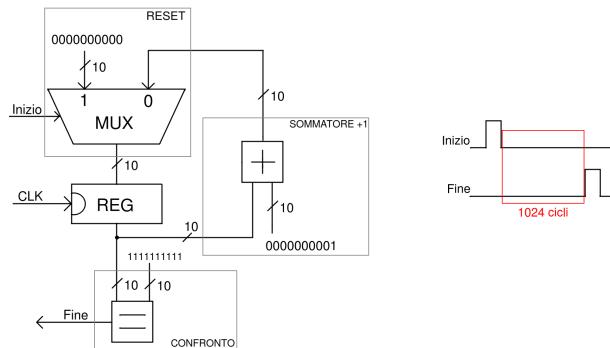
- **Da VerdeNS a VerdeEO (01/01):** Non c'è traffico su NS, ma solo su EO; quindi verde a EO e rosso a NS.
- **Da VerdeNS a VerdeNS ($\frac{00}{1-} / 10$):** Rimane in NS se non c'è traffico da entrambe le parti (00) oppure se c'è traffico su NS (1-) per priorità; quindi verde a NS e rosso a EO.
- **Da VerdeEO a VerdeNS (1-/10):** Non c'è traffico su EO, ma solo su NS, oppure se c'è traffico da entrambi (priorità a NS); quindi verde a NS e rosso a EO.
- **Da VerdeEO a VerdeEO (0-/01):** Rimane in EO se non c'è traffico da entrambe le parti, oppure se c'è traffico solo su EO; quindi verde a EO e rosso a NS.

2) Estensione I:

Il controllore non commuta un semaforo senza aver atteso 1024 cicli di clock. Questo serve a dare tempo al sistema, come se esistesse un giallo nel semaforo: nel caso base il semaforo oscillerebbe tra verde e rosso (a seconda delle condizioni) sull'istante che il traffico cambia.

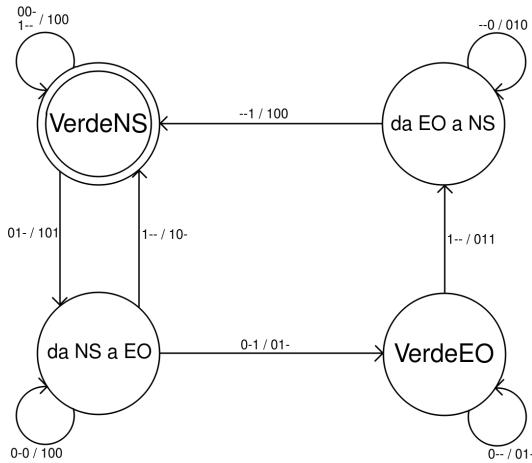
La FSM dice al DATAPATH di iniziare il conteggio di 1024 cicli, nel momento in cui si creano le condizioni per una commutazione la FSM aspetta dal DATAPATH il completamento del conteggio prima di commutare:

1. Si realizza prima il DATAPATH che conta:



2. Si realizza (o si estende il caso base) la FSM che gestisce la commutazione; la produzione di "inizio" e la cattura di "fine".

Codifica: TrafficoNS TrafficoEO Fine / LuceNS LuceEO Inizio



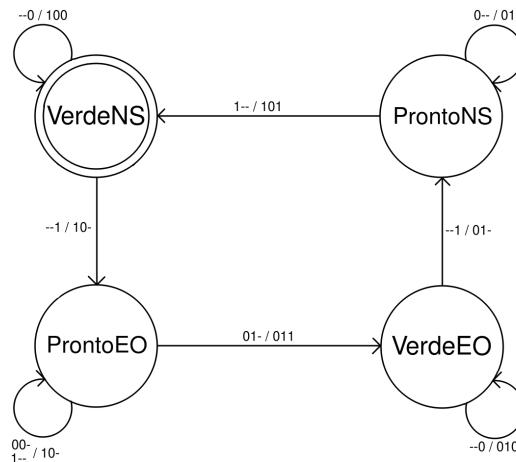
- **Da VerdeNS a "da NS a EO" (01-/101):** Non c'è traffico in NS e c'è in EO, mentre il segnale fine non ha importanza che sia 0 o 1; quindi verde a NS e rosso a EO con segnale di inizio 1 (perché deve iniziare il conteggio di 1024).
- **Da VerdeNS a VerdeNS ($\frac{00-1}{1--}$ /100):** Rimane se non c'è traffico in nessuna parte (00) oppure se c'è in NS dando la priorità (1-), mentre il segnale di fine può essere 0 o 1; quindi verde a NS e rosso a EO con segnale di inizio 0.
- **Da "da NS a EO" a VerdeNS (1 --/10-):** Se si presenta traffico in NS, indipendentemente da tutto, viene cancellato il conteggio e si torna a NS; quindi verde a NS e rosso a EO con segnale di inizio 0 o 1.
- **Da "da NS a EO" a VerdeEO (0-1/01-):** Non c'è traffico su NS, può esserci o no traffico su EO, ma si è ottenuto 1 come segnale di fine per il conteggio; quindi verde a EO e rosso a NS con segnale di inizio 0 o 1.
- **Da "da NS a EO" a "da NS a EO" (0-0/100):** Si rimane nel caso non ci sia traffico in NS, può esserci o no traffico su EO, il segnale di fine rimane a 0; quindi verde a NS e rosso a EO con segnale di inizio 0 (per fare scendere l'inizio dato come 1).
- **Da VerdeEO a "da EO a NS" (1 --/011):** Basta che ci sia traffico in NS (per priorità) e 0 o 1 in EO, mentre il segnale di fine può essere 0 o 1; quindi verde a EO e rosso a NS con segnale di inizio 1.
- **Da VerdeEO a VerdeEO (0 --/01-):** Rimane nel caso in NS non ci sia traffico e che ci sia o no in EO, mentre la fine non importa che sia 0 o 1; quindi è verde a EO e rosso a NS con segnale di fine 0.

- **Da "da EO a NS" a VerdeNS (- - 1/100):** Non importa che ci sia traffico in NS e EO, con segnale di fine 1; quindi verde a NS e rosso a EO con segnale inizio 0.
- **Da "da EO a NS" a "da EO a NS" (- - 0/010):** Non importa che ci sia traffico in NS e EO, con segnale fine 0; quindi verde a EO e rosso a NS con segnale inizio 0.

3) Estensione II:

In questa seconda estensione del caso base bisogna attendere **almeno** 1024 cicli tra una commutazione all'altra:

- **Caso a)** si mantiene il DATAPATH della prima estensione e si riadatta la FSM.
Codifica: *TrafficoNS TrafficoEO Fine / LuceNS LuceEO Inizio*



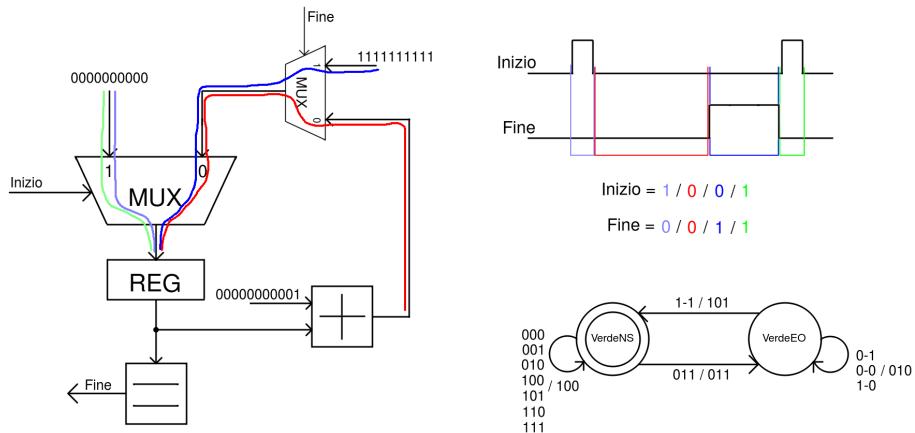
- **Da VerdeNS a ProntoEO (- - 1/10-):** Basta soltanto che venga dato il segnale di fine 1, perché come minimo comunque bisogna attendere 1024 cicli, perciò è lo stato **ProntoEO** che effettua la commutazione in caso ci siano le condizioni valide per passare a **VerdeEO**; quindi verde a NS e rosso a EO, mentre il segnale di inizio può essere 0 o 1.
- **Da VerdeNS a VerdeNS (- - 0/100):** Non importa se c'è traffico in NS ed EO perché deve prima aspettare 1024 cicli di clock, mentre il segnale di fine è 0; quindi verde a NS e rosso a EO, con segnale di inizio 0.
- **Da ProntoEO a VerdeEO (01-/011):** Non c'è traffico in NS e c'è in EO, con segnale di fine 0 o 1; quindi verde a EO e rosso a NS, con segnale di inizio 1 per iniziare a contare.
- **Da ProntoEO a ProntoEO (0 0 - /10-):** Rimane nel caso non ci sia traffico in NS e EO (00-), oppure che ci sia traffico in NS perché ha priorità

(1 - -), mentre il segnale di fine 0 o 1; quindi verde a NS e rosso a EO, con segnale di inizio 0 o 1.

- **Da VerdeEO a ProntoNS (- - 1/01-)**: Analogico ragionamento del passaggio tra VerdeNS e ProntoEO, l'unica differenza è il verde a EO e rosso a NS.
- **Da VerdeEO a VerdeEO (- - 0/010)**: Analogico ragionamento dell'arco che va da VerdeNS a VerdeNS, nemmeno la priorità di NS non vale in questo caso; quindi verde a EO e rosso a NS, con segnale di inizio 0.
- **Da ProntoNS a VerdeNS (1 - -/101)**: C'è traffico in NS e per priorità non importa se c'è in EO, mentre il segnale di fine 0 o 1; quindi verde a NS e rosso a EO, con segnale di inizio 1 che parte con il conteggio.
- **Da ProntoNS a ProntoNS (0 - -/01-)**: Rimane nel caso non ci sia traffico in NS, mentre il segnale di fine è 0 o 1; quindi rosso a NS e verde a EO, con segnale di inizio 0 o 1.

Quindi in questa versione i due stati **ProntoEO** e **ProntoNS** servono solo per contare il tempo minimo di 1024 cicli di clock indipendentemente da tutto ciò che accade.

- **Caso b)** si costruisce un DATAPATH in cui "fine" rimane alto da 1024 in poi.



-Casi DATAPATH:

- **Se inizio=1, fine=0**: Con inizio 1 e fine pari a 0 il registro si blocca a 0.
- **Se inizio=0, fine=0**: Con inizio 0 si sta contando e con fine pari a 0 non si è ancora arrivati a conteggio.
- **Se inizio=0, fine=1**: Con inizio 0 e fine pari a 1 viene fornito 1023 al registro.
- **Se inizio=1, fine=1**: Con inizio 1 e fine pari a 1 il registro si blocca a 0.

-FSM:

- **Da VerdeNS a VerdeEO (011/011)**: Non c’è traffico in NS e c’è in EO, mentre il segnale di fine è 1 perché ha atteso almeno 1024 cicli; quindi verde a EO e rosso a NS, con segnale di inizio 1 così da iniziare il conteggio.
- **Da VerdeNS a VerdeNS (*/100)**: Tutti i casi per cui non c’è il passaggio da VerdeNS a VerdeEO; quindi verde a NS e rosso a EO, con segnale di inizio 0 così da contare i cicli.
- **Da VerdeEO a VerdeNS (1-1/101)**: C’è traffico in NS a prescindere da EO, mentre il segnale di fine è 1 perché sono passati almeno 1024 cicli; quindi verde a NS e rosso a EO, con segnale di inizio 1.
- **Da VerdeEO a VerdeEO ($\begin{smallmatrix} 0 & - & - \\ 1 & - & 0 \end{smallmatrix} / 010$)**: Si rimane quando non c’è traffico in NS (0 - -), oppure quando c’è traffico in NS ma è ancora 0 il segnale di fine; quindi verde a EO e rosso a NS, con segnale di inizio 0 perché deve contare (altrimenti rimane in loop).

*: 000, 001, 010, 100, 101, 110, 111.

6.4 FMSD da un algoritmo

Le precedenti **FMSD** erano applicate al **linguaggio naturale**, quindi che la specifica fosse interpretata in modo da definire l’interfaccia tra i due componenti: FSM e DATA-PATH.

Nel caso di una **FMSD** da un algoritmo la procedura è univoca.

Per esempio viene preso in considerazione l’algoritmo per il massimo comune divisore:

```
int x, y;
while(true) {
    while(!ready);
    x = x_i;
    y = y_i;
    while(x != y) {
        if(x < y)
            y = y - x;
        else
            x = x - y;
    }
    d_o = x;
}
```

In questo algoritmo (scritto in like-C) sono presenti 3 segnali di ingresso: **ready**, **x_i**, **y_i**; e uno di uscita: **d_o**.

Quando arriva un segnale dall’esterno all’interno di **ready** che avvisa che gli altri due ingressi sono pronti, allora si procede con i calcoli.

Esempio dell'algoritmo con dei valori:

$x_i = 30$, $y_i = 75$

Si entra nel loop:

$x \neq y$ (sì) $\rightarrow x < y$ (sì) $\rightarrow y = 75 - 30 = 45$
 $x \neq y$ (sì) $\rightarrow x < y$ (sì) $\rightarrow y = 75 - 30 = 15$
 $x \neq y$ (sì) $\rightarrow x < y$ (no) $\rightarrow x = 30 - 15 = 15$
 $x \neq y$ (no) $\rightarrow d_o = 15$

Il passo successivo è quello di arrivare da questo algoritmo a una macchina **FSMD**. Prima di fare ciò bisogna analizzare l'algoritmo scritto e si possono notare delle componenti di sottrazione e confronto che richiedono il **DATAPATH**. Però ci vuole anche del controllo che richiede la **FSM**.

La procedura per costruire DATAPATH e FSM consiste in:

1. Individuare i blocchi di esecuzione per costruire una **EFSM** (*Extended Finite State Machine*).

```

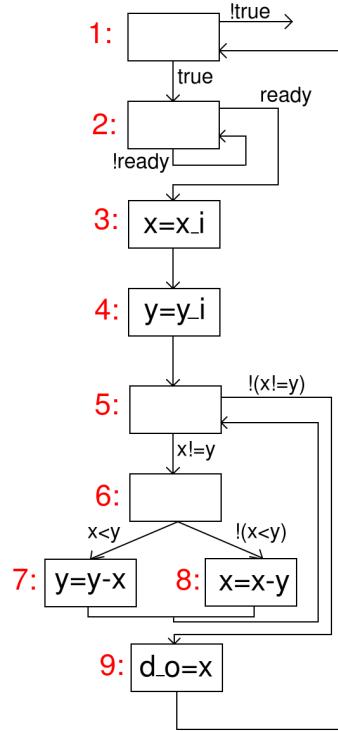
int x, y;
1:while(true) {
    2:while(!ready);
    3:x = x_i;
    4:y = y_i;
    5:while(x != y) {
        6:if(x < y)
            7:y = y - x;
        else
            8:x = x - y;
    }
    9:d_o = x;
}

```

È una **FSM** con informazioni pertinenti al **DATAPATH** e descrive completamente l'algoritmo.

2. Si definiscono i 3 tipi di blocchi trovati:

- **Assegnazione**: definito da un rettangolo con una singola uscita di flusso.
- **Ciclo while**: definito da due uscite di flusso: una che prosegue e la seconda ritorna al blocco.
- **Condizione if/else**: definito da due uscite di flusso: entrambe proseguono per due strade differenti per poi ricongiungersi in un unico flusso.

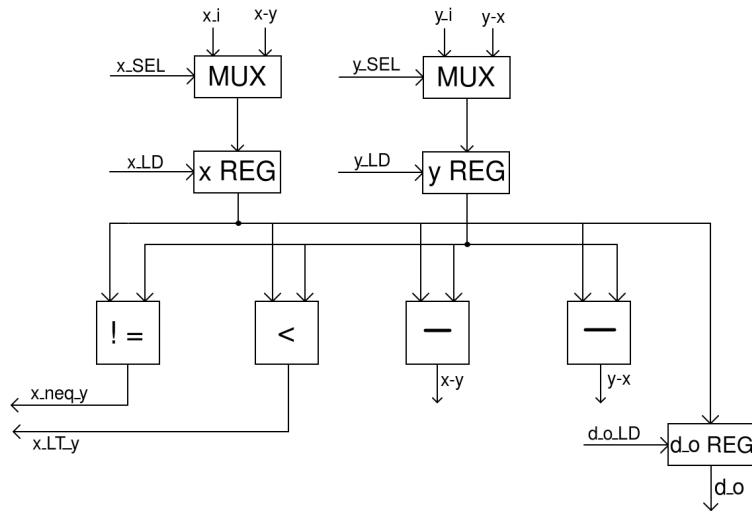


3. Dalla **EFSM** (rappresentata dai blocchi) si costruisce il **DATAPATH** secondo questi criteri:

- Ogni operazione di confronto logico/aritmetico individua un blocco operativo nel **DATAPATH**.
- In presenza di più sorgenti per una variabile, si introduce un **multiplexer** di opportuna dimensione.
In questo caso x ha sorgente x_i (cioè $x-y$), mentre y ha sorgente y_i (cioè $y-x$).
- Si introduce un registro per ogni variabile interna e per ogni uscita che non è assegnata in ogni stato. I registri sono registri con un segnale di **enable**: se $E = 1$ allora il valore viene aggiornato sul fronte di clock, altrimenti no.
 x, y, d_o richiedono registri.

4. Si inizia a introdurre il **DATAPATH**. Gli approcci possibili sono:

- Il **top-down**: si identificano i macro-blocchi, si collegano e poi si definisce l'interno dei macro-blocchi.
- Il **bottom-up**: si parte dagli elementi di libreria e vengono collegati a costruire dei macro-blocchi.



Si utilizza un mixto di entrambi per la costruzione del DATAPATH:

$$\begin{aligned}
 \text{Input (interno) da FSM} & \left\{ \begin{array}{l} x_LD \\ y_LD \\ x_SEL \\ y_SEL \\ d_o_LD \end{array} \right. & \text{Output (interno) da FSM} & \left\{ \begin{array}{l} x_NEQ_y \\ y_NEQ_x \end{array} \right. \\
 \text{Input (esterno) della FSMD} & \left\{ \begin{array}{l} x_i \\ y_i \\ ready \end{array} \right. & \text{Output (esterno) della FSMD} & \{ d_o
 \end{aligned}$$

Questa è l'**interfaccia** fra i due componenti.

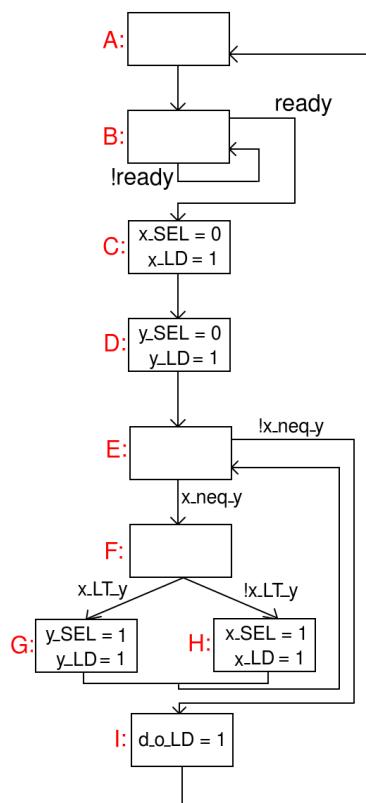
5. Avendo l'**interfaccia** si può stabilire la FSM, quindi convertire la **EFSM** in **FSM**.

Ausili:

Evitare di definire i valori dei segnali di input/output della FSM quando non sono rilevanti in un dato stato:

- Se non si scrive il valore di un segnale di **enable** (o **load**) allora vale 0.
- Se non si scrive il valore di un segnale di selezione, allora vale *don't care*.
- Se non si scrive il valore di un input, allora vale *don't care*.

Conversione (i numeri sono stati cambiati in lettere):



Laboratorio SIS

SIS è un software che implementa in parte l'algoritmo di **Q&MC**. Il suo scopo è quello di descrivere **circuiti digitali**¹

7.1 Descrizione del primo componente

Come già dichiarato in precedenza **SIS** descrive **circuiti digitali** attraverso una funzione booleana.

a	b	a→b
0	0	1
0	1	1
1	0	0
1	1	1

Per esempio viene data la funzione booleana dell'implicazione: In base
se alla funzione data è possibile definire tramite la **tabella di verità** i valori d'uscita, in questo modo si definisce il seguente codice:

```
.model implicazione
.inputs a b
.outputs c
.names a b c
00 1
01 1
11 1
.end
```

Spiegazione comandi:

- **.model "nome del componente"**: riporta il nome del componente da descrivere: nell'esempio precedente veniva descritta l'**implicazione**.
- **.inputs "porte"**: vengono definite le porte di input: nell'esempio precedente con l'implicazione si hanno due valori in ingresso (**a** e **b**) nella tabella di verità.

¹Per visualizzare i comandi presenti su SIS utilizzare il comando **help**. Per visualizzare un comando nello specifico inserire **help nome_comando**.

- **.outputs "porte"**: vengono definite le porte di output: nell'esempio precedente con l'implicazione si ha un valore d'uscita (definita **c**) presente nella colonna destra della tabella di verità.
- **.names "porte di input" "porta di output"**: si definisce la funzionalità dell'oggetto descritto: la relazione tra le porte di input e di output. **Attenzione** che nel caso ci siano più di una porta di output devono essere descritti più **.names** per ogni porta.
Inoltre il comando **.names** può essere utilizzato anche per definire nodi intermedi che non sono necessariamente porte di output.
- **Definizione del comportamento**: dopo l'introduzione del comando **.names** si descrive il comportamento della funzione riportando solo o l'insieme di **ON-SET** od **OFF-SET**: in questo caso è stato riportato l'**ON-SET**.
In questo modo **SIS** andrà a costruire le porte logiche indicate per mostrare il risultato durante la simulazione, e automaticamente costruirà le rimanenti per esclusione dei valori non inseriti (quindi l'insieme non descritto).
- **.end**: conclude la descrizione dell'oggetto.

Una volta conclusa la descrizione dell'oggetto il file va salvato come **implicazione.blif** (in genere inserire il nome usato per **.model**).

Sul terminale si lanciano i comandi i comandi:

SIS READ_BLIF implicazione.blif SIMULATE 0 0 SIMULATE 1 0 ...	<i>(Avvia il programma SIS)</i> <i>(Carica il file .blif)</i> <i>(Simula gli ingressi a = 0 e b = 0 dando risultato c = 1)</i> <i>(Simula gli ingressi a = 1 e b = 0 dando risultato c = 0)</i> ...
--	---

Per visualizzare su schermo il contenuto del file **.blif** inserire il comando:

write_blif.

Se si vuole visualizzare il contenuto di un altro file **.blif** va specificato subito dopo il nome.

Inoltre è possibile con **print_stats** visualizzare le statistiche del componente creato, cioè:

- Quanto consuma in termini di porte: **pi** (*primary input*) e **po** (*primary output*).
- Quanta area occupa: **nodes**.
- Quant letterali occupa la descrizione: **list(sop)**. Se si definisce la funzione in modo differente si possono diminuire o aumentare il numero di letterali.
Nell'esempio precedente se veniva inserito nella descrizione solo l'**OFF-SET** (quindi: 10 0) al posto di 6 letterali sarebbero stati 2.

-Esercizio sommatore binario:

Descrivere in formato **.blif** un sommatore binario ad un bit, simularne il comportamento con **SIS** ed analizzare le statistiche del circuito con il comando **print_stats**.

Codice:

```
.model sommatore
.inputs a b cin
.outputs s cout
.names a b cin s
000 0
011 0
101 0
110 0
.names a b cin cout
000 0
001 0
010 0
100 0
.end
```

a	b	cin	s	cout
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

Spiegazione:

-In input vengono presi **a**, **b** e **cin** dove **a** e **b** vengono sommati tra loro insieme a **cin** (il riporto), in output vengono mostrati rispettivamente il risultato finale dell'addizione in **s** e in **cout** un eventuale riporto.

-Nel codice vengono descritti i comportamenti per i due output **s** e **cout**.

7.1.1 Descrizione funzioni booleane in SIS

-Funzione AND:

Codice:

```
.model and
.inputs a b
.outputs out
11 1
.end
```

a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

-Funzione OR:

a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

Codice:

```
.model or
.inputs a b
.outputs out
00 0
.end
```

-Funzione NOT:

a	out
0	1
1	0

Codice:

```
.model not
.inputs a
.outputs out
0 1
.end
```

-Funzione NAND:

a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

Codice:

```
.model nand
.inputs a b
.outputs out
11 0
.end
```

-Funzione NOR:

a	b	out
0	0	1
0	1	0
1	0	0
1	1	0

Codice:

```
.model nor
.inputs a b
.outputs out
00 1
.end
```

-Funzione XOR:

a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Codice:

```
.model xor
.inputs a b
.outputs out
00 0
11 0
.end
```

-Funzione XNOR:

a	b	out
0	0	1
0	1	0
1	0	0
1	1	1

Codice:

```
.model xnor
.inputs a b
.outputs out
01 0
10 0
.end
```

7.2 Ottimizzazione esatta dei circuiti combinatori

L'**ottimizzazione** di un circuito digitale consiste nel trasformare del circuito in uno funzionalmente equivalente avente **area e/o ritardi** minimi.

Lo scopo dell'ottimizzazione è quello di ottenere circuiti piccoli e veloci.

La minimizzazione di circuiti a 2 livelli avviene in 3 passi:

1. Si identificano tutti gli implicanti primi essenziali.
2. Si identifica un insieme minimo di implicanti che coprano tutti i mintermini non coperti dagli implicanti primi essenziali.
3. La funzione di copertura ottima è data dalla somma degli implicanti trovati ai punti 1 e 2.

-Circuiti a una uscita:

Esiste un metodo esatto per trovare gli implicanti primi essenziali.

Esiste un metodo esatto o approssimato (dipende dal circuito) per ottenere la funzione di copertura ottima.

-Circuiti a più uscite:

Esiste un metodo approssimato per trovare la copertura ottima che si basa sul metodo esatto di identificazione degli implicanti primi essenziali di ogni singola uscita.

7.2.1 Minimizzazione dei mintermini

Prima di iniziare la minimizzazione dei mintermini è possibile visualizzare la funzione descritta in termini matematici per le uscite tramite il comando **write_eqn**, mentre con **print_stats** si presentano altre informazioni sulla funzione.

I comandi disponibili per la minimizzazione sono (descritti nel Capitolo 4.2.1):

- **simplify**: Applicazione di Q&MC a tutti i nodi.
- **full_simplify**: evoluzione di Q&MC.

7.2.2 Gestione del Don't Care Set

Una gestione intelligente del **DC set** può rendere il circuito più efficiente dopo aver effettuato la minimizzazione tramite appositi comandi.

Esempio di esercizio:

Ricevuta in input una sequenza composta da 5 bit, vengono associate le lettere dell'alfabeto, dalla A(=0) alla Z(=20), e in output viene mostrato 1 solo se la sequenza inserita corrisponde a una vocale.

La tabella di verità non risulterà completamente specificata dal momento che con 5 cifre binarie si possono rappresentare i numeri dallo 0 al 31.

Quindi bisogna valutare come gestire il **DC set** (che corrisponde ai valori fuori dall'alfabeto, dal 21 al 31 compresi).

-DC set appartenente all'off-set:

```
.model vocali_offset
.inputs i4 i3 i2 i1 i0
.outputs o

.names i4 i3 i2 i1 i0 o
00000 1
00100 1
01000 1
01100 1
10010 1

.end
```

Prima della minimizzazione possiede **25 letterali**.

Dopo la minimizzazione possiede **8 letterali**.

-DC set appartenente all'on-set:

```
.model vocali_onset
.inputs i4 i3 i2 i1 i0
.outputs o

.names i4 i3 i2 i1 i0 o
00000 1
00100 1
01000 1
01100 1
10010 1
10101 1
10110 1
10111 1
11000 1
11001 1
11010 1
11011 1
11100 1
11101 1
11110 1
11111 1

.end
```

Prima della minimizzazione possiede **80 letterali**.
Dopo la minimizzazione possiede **11 letterali**.

-DC set specificato a SIS:

```
.model vocali_dcset
.inputs i4 i3 i2 i1 i0
.outputs o

.names i4 i3 i2 i1 i0 o
00000 1
00100 1
01000 1
01100 1
10010 1

.exdc
.names i4 i3 i2 i1 i0 o
```

```

10101 1
10110 1
10111 1
11000 1
11001 1
11010 1
11011 1
11100 1
11101 1
11110 1
11111 1
.end

```

Prima della minimizzazione possiede **25 letterali**.

Dopo la minimizzazione possiede **6 letterali**.

Tutte e tre le soluzioni funzionano correttamente, ma la differente gestione del **DC set** porta a dei miglioramenti dopo la minimizzazione.

Il **DC set** è stato specificato a **SIS** inserendo prima del comando **.names** il comando: **.exdc**.

Con questo comando si descrive a livello sintattico il **DC set** come se fosse parte dell'onset.

7.2.3 Minimizzazione di circuiti multi-livello

La minimizzazione nei **circuiti multi-livello** consente di bilanciare area e ritardo dando maggior libertà rispetto alla **minimizzazione a 2 livelli**.

Purtroppo non esistono tecniche che consentono di ottenere la migliore ottimizzazione in assoluto, ma solo una buona ottimizzazione: si ricorre a tecniche euristiche.

Per riuscire a minimizzare a multi-livello **SIS** mette a disposizione vari comandi che distruggono e ricostruiscono la funzione booleana (per poi applicare **full_simplify**):

- **sweep**: elimina i nodi con un'unica linea di ingresso e con valori costanti.
- **eliminate**: elimina un nodo all'interno della rete tale che: un nodo N ha funzione $y = (a + b) \cdot c$ e la sua eliminazione prevede che il contenuto $(a + b) \cdot c$ sia sostituito al posto di y all'interno di altri nodi che lo contenevano.
- **resub**: sostituisce un nodo interno con un insieme di nodi la cui funzionalità sia equivalente a quella del nodo sostituito. In questo modo si diminuisce la complessità di un nodo.

- **fx**: simile alla **resub**, ma lavora solo quando una parte è comune a due nodi.
- **extract**: estrae una sottoespressione comune a più nodi che viene rappresentata con un nuovo nodo.
- **simplify**: riduce la complessità di ogni singolo nodo tramite l'algoritmo di Q&MC.

-Uso dello script.rugged:

Per effettuare la minimizzazione a multi-livello si possono utilizzare anche degli script che contengono una serie di comandi che distruggono e ricostruiscono la funzione booleana. Lo si può lanciare più volte tramite il comando **source script.rugged**, ma non è detto che a ogni avvio il risultato migliori, si va per tentativi e a volte potrebbe anche peggiorare dopo essere migliorato.

7.3 Technology Mapping su SIS

Una volta descritto il circuito tramite **SIS**, e dopo le minimizzazioni, è possibile effettuare il **Technology Mapping** mediante vari comandi:

- **read_library "nome-libreria"**: Carica la libreria tecnologica indicata. Le librerie sono specificate nel formato **.genlib**, per esempio **synch.genlib** e **mcnc.genlib**.
- **print_library**: Visualizza informazioni inerenti alla libreria caricata.
- **map**: Esegue l'operazione di *mapping*:
 - **map -m [0,1]**: con il valore 0 permette di ottenere un circuito minimizzato rispetto all'area, mentre con il valore 1 minimizza rispetto al ritardo.
 - **map -n 1**: permette di ottenere un circuito minimizzato (meglio di **map -m 1**) rispetto al ritardo.
 - **map -s**: permette di visualizzare alcune informazioni relative ad area e ritardo dopo il mapping.
- **write_blif -n**: Mostra la rappresentazione del circuito associata alle porte della libreria.
- **print_delay**: Fornisce informazioni relative al ritardo del circuito.
- **reduce_depth**: Riduce la lunghezza dei cammini critici.

Dopo aver avviato il mapping del circuito se si lancia il comando **write_blif** si notano nodi intermedi.

7.4 Circuiti Sequenziali e Macchine a Stati Finiti

Un **circuito sequenziale** può essere modellato utilizzando una **macchina a stati finiti** (FSM) definita mediante una 6-upla:

$M = (S, I, O, \delta, \lambda, s)$, dove:

- S : Insieme degli stati
- I : Insieme degli ingressi
- O : Insieme delle uscite
- δ : Funzione di stato prossimo che a una coppia (ingresso e stato attuale) associa lo stato prossimo.
- λ : Funzione d'uscita che associa una valore per l'uscita allo stato attuale (FMS di Moore) oppure a una coppia (FSM di Mealy).
- s : Stato di reset (non sempre è definito).

Una **FSM** si rappresenta con **STG** (grafo delle transizioni) o con una **STT** (tabella delle transizioni); entrambi i metodi si equivalgono.

I circuiti sequenziali possono essere:

- **Sincroni**: I valori delle uscite assumono significato in corrispondenza di un evento su un segnale detto *clock*.
- **Asincroni**: I valori delle uscite cambiano al variare degli ingressi senza tener conto del segnale di *clock*.

L'uscita di un **circuito sequenziale** dipende dai valori in ingresso anche negli istanti passati, occorre dunque una memoria rappresentata dallo stato.

L'elettronica mette a disposizione due componenti, detti **latch** e **flip-flop**, in grado di memorizzare il valore di un bit.

È necessario codificare tutti gli stati che può assumere il circuito mediante dei numeri binari e poi utilizzare i **latch** o **flip-flop** per memorizzare i valori.

Una **FSM** con N stati necessita di $\log_2 N$ componenti di memoria elementari.

7.4.1 Circuiti sequenziali in SIS

Per modellare un circuito sequenziale in **SIS** serve definire la tabella delle transizioni ed eventualmente definire manualmente la codifica degli stati.

La tabella delle transizioni deve essere descritta all'interno della sezione delimitata dalle

keyword **.start_kiss** e **.end_kiss**.

Le transizioni vengono specificate come un insieme di righe che riportano in quest ordine: valore degli ingressi, stato attuale, stato prossimo, valore delle uscite.

La tabella delle transizioni deve essere preceduta da 5 righe che specificano:

- Il numero di segnali in input.
- Il numero di segnali di output.
- Il numero di transizioni.
- Il numero di stati.
- Lo stato di reset.

Dopo **.end_kiss** possono essere riportate le istruzioni necessarie per definire la codifica degli stati nel caso non si decida di farla definire a **SIS** in modo automatico.

La keyword per definire la codifica è **.code** seguita dal nome dello stato e dalla sua codifica binaria.

Dopo aver modellato il circuito è possibile procedere con la minimizzazione degli stati. Una volta caricato il file **.blif** la minimizzazione degli stati si esegue con il comando **state_minimize stamina**.

La generazione logica delle funzioni δ e λ :

- Se il file contiene già la codifica binaria degli stati (**.code**) allora occorre generare le funzioni δ e λ con il comando **stg_to_network**.
- Altrimenti occorre assegnare automaticamente gli stati con **state_assign jedi**, ma prima occorre avere minimizzato gli stati e poi fare l'assegnazione.

Dopodiché la minimizzazione del δ e λ può essere compiuta dallo **script.rugged** come per la minimizzazione dei circuiti combinatori.

Esempio:

Questo circuito sequenziale è in grado di riconoscere la sequenza di ingresso **00 01 11** (ingresso IN). Il circuito inizia ad analizzare i valori dell'ingresso IN quando START è 1. Quando viene riconosciuta la sequenza OUT mostra 1, inoltre OUT rimane a 1 fino a quando gli ingressi assumeranno il valore 10; quando avviene ciò il circuito viene posto in attesa che il segnale START passi da 0 a 1.

Codifica: *START IN / OUT*.

```
.model automa
.inputs START IN1 IN0
.outputs OUT
```

```

.start_kiss
.i 3 #Numero di segnali di ingresso
.o 1 #Numero di segnali di uscita
.s 5 #Numero di stati
.p 15 #Numero di transizioni
.r ATT #Stato di reset

#Tabella delle transizioni
#(ingressi, stato attuale, stato prossimo, uscita)
0-- ATT ATT 0
1-- ATT NUL 0
--1 NUL NUL 0
-00 NUL 00 0
-00 00 00 0
-1- 00 NUL 0
-01 00 01 0
-00 01 00 0
-10 01 NUL 0
-01 01 NUL 0
-11 01 11 1
--1 11 11 1
--1 11 11 1
-0- 11 11 1
-10 11 ATT 0
.end_kiss

#Codifica degli stati (e' opzionale).
#Può essere calcolata automaticamente
#tramite il comando state_assign jedi
.code ATT 000
.code NUL 001
.code 00 010
.code 01 011
.code 11 100

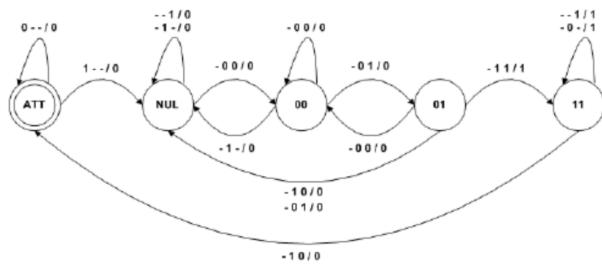
.end

```

Lo stato **ATT** è quindi lo stato di "attesa" fino a quando il valore START non diventa 1, successivamente passa allo stato di partenza del riconoscimento **NUL**. Quando il circuito riconosce la sequenza torna in **ATT** (attesa di START).

Per capire meglio ecco la STG del circuito:

Per simularla tramite **SIS** si utilizza il comando **simulate** come visto in precedenza (inserendo i valori di input), esempio: **simulate 0 1 1.**



7.5 DATAPATH

Un circuito digitale è composto dall'unione del **controllore** (FSM) e **DATAPATH**.

7.5.1 Unità funzionali del DATAPATH

Per includere un componente all'interno di un modello **blif** si utilizzano i seguenti comandi:

- **.subckt nomeComponente parametroFormale=parametroAttuale**

Il **parametroFormale** è il simbolo con il quale viene individuata la porta di input di quel componente.

Il **parametroAttuale** è il componente al quale si vuole collegare quello del **parametroFormale**.

- **.search nomeFileComponente.blif**

In questo modo viene viene collegato all'interno del componente.

Esempio:

Sommatore a 1 bit:

```

.model SOMMATORE
.inputs A B CIN
.outputs OUT COUT

#K e' un nodo intermedio
.names A B K
10 1
01 1

.names K CIN OUT
10 1
01 1

.names A B CIN COUT
11 - 1

```

```
1 -1 1
-11 1
```

```
.end
```

Sommatore a 2 bit:

```
.model SOMMATORE2
.inputs A1 A0 B1 B0 CIN
.outputs OUT1 OUT0 COUT

.subckt SOMMATORE A=A0 B=B0 CIN=CIN OUT=OUT0 COUT=CO
.subckt SOMMATORE A=A1 B=B1 CIN=CO OUT=OUT1 COUT=COUT
.search sommatore.blif

.end
```

7.5.2 Modellazione dei componenti

-**Registro a 1 bit:**

```
.latch <input> <output> <type> <control> <init-val>
```

- <**input**>: è l'ingresso del latch.
- <**output**>: è l'uscita del latch.
- <**type**>: può essere **fe** (*falling edge*) fronte di discesa, **re** (*rising edge*) fronte di salita, **ah** (*active high*) momento in cui è stabile nel valore alto, **al** (*active low*) momento in cui è stabile nel valore basso, **as** (*active asynchronous*).
- <**control**>: è il segnale di **clock** per il latch. Può essere un clock del modello, l'uscita di una qualsiasi funzione del modello, o la parola "*NIL*" per nessun clock interno.
Quindi il registro utilizza il clock generale del circuito in cui è inserito.
- <**init-val**>: è lo stato iniziale del latch, può essere 0, 1, 2, 3.
Il **2** indica il *don't care* e il **3** indica "*unknown*" (o non specificato).

Esempio:

```
.model REGISTRO
.inputs A
.outputs OUT

#L'input e' la A
#L'output e' OUT
```

```
# Il tipo e' "re", quindi e' sul fronte di salita
#"NIL" non e' collegato a niente, dunque diventa asincrono
#0 e' il valore iniziale del registro
.latch A OUT re NIL 0

.end
```

-Registro a 4 bit:

Per creare un registro più grande si può semplicemente istanziare quello a 1 bit visto in precedenza, tramite i comandi **.subckt** e **.search**.

Quindi diventerà:

```
.model REGISTR04
.inputs A3 A2 A1 A0
.outputs O3 O2 O1 O0

.subckt REGISTR0 A=A3 O=O3
.subckt REGISTR0 A=A2 O=O2
.subckt REGISTR0 A=A1 O=O1
.subckt REGISTR0 A=A0 O=O0
.search registro.blif

.end
```

-Multiplexer a 4 ingressi a 1 bit:

```
.model MUX1
.inputs S1 S0 i3 i2 i1 i0
.outputs OUT

.names S1 S0 i3 i2 i1 i0 OUT
001--- 1
01-1-- 1
10--1- 1
11---1 1

.end
```

-Multiplexer a 4 ingressi a 2 bit:

```
.model MUX2
.inputs X1 X0 a1 a0 b1 b0 c1 c0 d1 d0
.outputs OUT1 OUT2

.subckt MUX1 S1=X1 S0=X0 i3=a1 i2=b1 i1=c1 i0=d1 OUT=OUT1
.subckt MUX1 S1=X1 S0=X0 i3=a0 i2=b0 i1=c0 i0=d0 OUT=OUT0
.search mux1.blif

.end
```

-Multiplexer a 2 ingressi a 3 bit:

```
.model MUX3
.inputs A2 A1 A0 B2 B1 B0 S
.outputs OUT2 OUT1 OUT0

.names S A2 B2 OUT2
11- 1
0-1 1

.names S A1 B1 OUT1
11- 1
0-1 1

.names S A0 B0 OUT0
11- 1
0-1 1

.end
```

-Demultiplexer a 1 bit e 4 uscite:

```
.model DEMUX
.inputs S1 S0 IN
.outputs X Y Z W

.names S1 S0 IN X
001 1

.names S1 S0 IN Y
011 1
```

```
.names S1 S0 IN Z
101 1

.names S1 S0 IN W
111 1

.end
```

-Comparatore a 4 bit:

```
.model COMPARATORE4
.inputs A3 A2 A1 A0 B3 B2 B1 B0
.outputs OUT

.subckt xnor A=A3 B=B3 X=X3
.subckt xnor A=A2 B=B2 X=X2
.subckt xnor A=A1 B=B1 X=X1
.subckt xnor A=A0 B=B0 X=X0

.names X3 X2 X1 X0 OUT

.search xnor.blif

.end
```

7.5.3 Modellazione di una FSMD in SIS

1. Modellare la FMS del **Controllore** mediante la tabella delle transizioni e assegnare la codifica degli stati con **state_assign jedi**.
2. Salvare il **Controllore** ottenuto dopo la codifica degli stati in un file diverso da quello originale usando il comando **write_blif nomefile**.
3. Modellare il **DATAPATH** come una interconnessione di componenti funzionali come i sommatori, moltiplicatori, multiplexer, registri, ecc.
4. Inglobare il **Controllore** e il **DATAPATH** in un unico file **blif** come se fossero due componenti, collegando in maniera opportuna i segnali di comunicazione. È importante specificare con la direttiva **.search** il file del **Controllore** ottenuto dopo la codifica degli stati.