Architettura degli elaboratori

UniVR - Dipartimento di Informatica

Fabio Irimie

1° Semestre 2023/2024

Indice

1 Introduzione

L'informatica è nata per la risoluzione di problemi di calcolo, in particolare quelli di calcolo numerico. Per questo motivo i primi computer erano macchine che eseguivano operazioni aritmetiche. Per risolvere questi problemi si usano degli algoritmi che sono una sequenza di istruzioni semplici che portano poi a risolvere problemi di complessità variabile. Anche gli algoritmi hanno una complessità che deve essere adeguata alla risoluzione del problema.

1.1 Hardware

Un algoritmo deve essere trasformato in un processo di calcolo automatico, quindi deve essere implementato tramite hardware. Ci sono due tipi di hardware:

- **Embedded** che è un hardware dedicato ad un singolo compito. Ad esempio il microonde.
- **General purpose** non si sa l'utilizzo finale, quindi ha funzionalità generali ampliate dal software installato. L'hardware general purpose è programmabile attraverso il software. Un esempio è il PC.

In base al tipo di hardware l'algoritmo viene implementato in diversi modi:

- **Algoritmo** $\rightarrow$ **Software**: Tramite un linguaggio di programmazione
- **Algoritmo** $\rightarrow$ **Hardware embedded**: Tramite linguaggi di basso livello come C, Assembly o il sistema operativo.
- **Algoritmo** $\rightarrow$ **Hardware**: Tramite sintesi logica

1.2 Campionamento dei dati

Ogni cosa nel mondo è rappresentabile da funzioni continue nel tempo $f(t)$, ma con risorse finite è impossibile rappresentare infiniti dati, bisogna quindi campionarli.

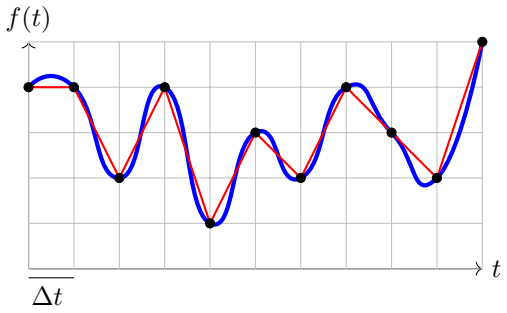


Figura 1: Funzione casuale continua nel tempo

Per campionare la funzione nella figura 1 bisogna scegliere un intervallo di tempo Δt e prendere un valore della funzione ogni Δt. In questo caso le linee verticali rappresentano il **campionamento**, mentre quelle orizzontali reppresentano la **discretizzazione o quantizzazione**. La linea rossa è una spezzata approssimata della funzione continua, infatti per il teorema di Shannon:

Teorema 1 *Deciso il grado di errore da voler compiere, esistono una precisa frequenza di campionamento e un intervallo di discretizzazione che garantiscono quell'errore.*

Il sistema di calcolo è ora diventato digitale, cioè elabora i segnali numerici in ingresso per produrre segnali numerici in uscita.

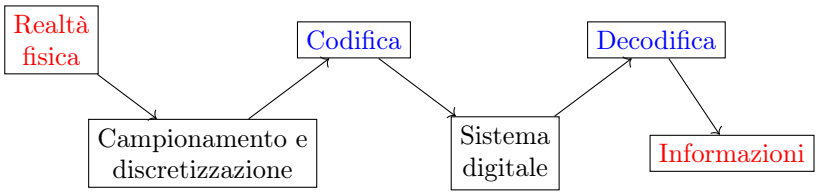


Figura 2: Dalla realtà fisica al sistema digitale

2 Sistemi di codifica

Ogni sistema digitale lavora in base binaria, quindi entrano N bit ed escono M bit. I bit in uscita devono essere codificati per realizzare delle informazioni. Ci sono 2 tipi di informazioni:

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

2.1 Codifica di informazioni non numeriche

Ogni informazione deve avere un codice univoco in modo che il sistema digitale non possa sbagliare a decodificarla. Date M informazioni si ricavano $n = log_2(M)$ codici disponibili per rappresentarle.

Esempio 2.1
Con $M = 7$ informazioni:

- $n = log_2(7) \approx 3$ *bit*
- $2^3 = 8$ *codici disponibili*

2.2 Numeri interi assoluti

I numeri interi assoluti rappresentano solo i valori da 0 a $2^n - 1$, dove n è il numero di bit disponibile.

La codifica da base decimale a base binaria prende il nome di **codifica a modulo**

Esempio 2.2
Si deve convertire il numero 57_{10} *in base binaria*

$$n = log_2(57) = 6 \; bit \; (minimi)$$

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

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 si moltiplica per 1.*
   - *Se la potenza* 2^i *è maggiore del numero, allora si moltiplica per 0.*
2. *Le sottrazioni continuano fino a quando si giunge a 0.*

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

$$57 = 111001$$

2.3 Numeri interi relativi

La codifica più ovvia per i numeri interi relativi è la codifica a **modulo + segno**. Tuttavia rappresenta varie problematiche, per cui si preferisce usare la codifica in **complemento a 2**.

2.3.1 Codifica a modulo + segno

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

Il segno si rappresenta con un bit, 0 per il positivo e 1 per il negativo. Il bit più significativo è il bit del segno, mentre i bit meno significativi rappresentano il modulo.

1 bit: segno ±	7 bit: modulo

Figura 3: Bit dedicati alla codifica a modulo + segno

Considerando l'esempio 2.2 si hanno le seguenti rappresentazioni:

$$+57_{10} = \mathbf{0}|111001_2$$
$$-57_{10} = \mathbf{1}|111001_2$$

Sorge però un problema quando si vuole rappresentare il valore 0_{10}, che in binario risulterebbe:

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

Inoltre le somme che passano dal positivo al negativo e viceversa risultano errate.

2.3.2 Codifica in complemento a 2

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

La codifica in complemento a 2 rimuove tutti i problemi della codifica in modulo + segno. Questa codifica infatti rende le somme molto più semplici. La somma facile infatti è l'obiettivo di questa codifica e parte dell'idea di trovare la codifica di -1, pertanto si cerca di formulare $-1 + 1 = 0$.

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

Tabella 1: Obiettivo della codifica in complemento a 2

Se si considera il numero di bit $n = 4$, allora l'intervallo di valori è $-2^3 \leq N \leq 2^3 - 1$:

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

Tabella 2: Codifica in complemento a 2 con $n = 4$ bit

I valori nel complemento a 2 ciclano, quindi se si somma 1 a 7 si ottiene -8.

Esempio 2.3
Sottrazione con il complemento a 2: $43 - 17 = 25$

$$n = 7 \textit{ bit}$$

1. *Per prima cosa si prende il valore assoluto del numero negativo* 17_{10} *e si converte in binario.*

$$17_{10} = 0010001_2$$

2. *Si inverte il numero trovato.*

$$!(0010001_2) = 1101110_2 = -18_{10}$$

3. *Si somma 1 al numero trovato.*

$$\begin{array}{r} 1101110 + \\ 0000001 = \\ \hline 1101111 \end{array}$$
$$1101111_2 = -17_{10}$$

Tabella 3: Somma di 1 al numero invertito

4. *Si somma il numero trovato al numero positivo.*

$$\begin{array}{r} 0010001 + \\ 1101111 = \\ \hline \mathbf{1}0011010 \end{array}$$

Tabella 4: Somma del numero positivo con il numero negativo

5. *Il risultato ottenuto è:*

$$\mathbf{1}0011010$$

Si osserva che c'è un bit in più rispetto a quelli disponibili (quello in grassetto), vuol dire che risulta in overflow[a]*, quindi si scarta il bit più significativo e si ottiene:*

$$0011010_2 = 26_{10}$$

che è il risultato corretto.

[a]Indica il "traboccamento", cioè se viene superato il limite massimo l'overfflow è un errore, non perchè sia sbagliata la somma, ma perchè il risultato non è codificabile con il numero di bit disponibili

Estensione del numero con il complemento a 2

- Se un numero è **positivo** va esteso con gli **0**

$+57_{10}+$ $+7_{10} =$	$0111001_2 +$ $\mathbf{0000}111_2 =$
$+64_{10}$	1000010_2

Tabella 5: Estensione di un numero positivo

- Se un numero è **negativo** va esteso con gli **1**

$+57_{10}+$	$0111001_2 +$
$-7_{10} =$	$\mathbf{1111}111_2 =$
$+50_{10}$	10110010_2

Tabella 6: Estensione di un numero negativo

3 Numeri razionali

I numeri razionali sono composti da una parte intera e una parte frazionaria. Si possono codificare in 2 modi:

- **Virgola fissa**(fixed point): viene usata maggiormente nei sistemi embedded quando si sa a priori il numero più grande e la precisione che si vuole ottenere
- **Virgola mobile**(floating point): viene usata maggiormente nei sistemi general purpose.

3.1 Codifica in virgola fissa

Esempio 3.1
Si hanno a disposizione 8 bit: 4 per la parte intera e 4 per la parte frazionaria. Vogliamo decodificare il numero 0110.1011_2*:*

$$\overbrace{\overset{2^3}{0}\ \overset{2^2}{1}\ \overset{2^1}{1}\ \overset{2^0}{0}}^{+6}.\overbrace{\overset{2^{-1}}{1}\ \overset{2^{-2}}{0}\ \overset{2^{-3}}{1}\ \overset{2^{-4}}{1}}^{\frac{1}{2}+\frac{1}{8}+\frac{1}{16}}$$

$$+6+\frac{1}{2}+\frac{1}{8}+\frac{1}{16}=6+\frac{11}{16}=\frac{107}{16}=6.6875$$

Se si vuole codificare un numero da decimale a binario bisogna tenere in considerazione che non è certo che il numero sia razionale anche in base 2, quindi bisogna approssimare per rappresentarlo.

Esempio 3.2
Prendiamo in considerazione $+4+\frac{3}{5}$*, in questo caso bisogna andare "a tentoni" e trovare la rappresentazione binaria che approssima con il minor errore possibile.*

$$4_{10}=0100_2$$

$$0.1001=\frac{9}{10}\Delta\frac{3}{80}$$

$$0.0111 = \frac{7}{16}\Delta - \frac{4}{80}$$
$$0.0110 = \frac{3}{8}\Delta\frac{9}{40}$$
$$\underline{0.1010 = \frac{5}{8}\Delta - \frac{1}{40}}$$

Δ *rappresenta l'errore, quindi la rappresentazione più vicina è* 0100.1010_2. *Però non è stato rappresentato* $\frac{3}{5}$, *ma* $\frac{1}{2} + \frac{1}{16} = \frac{9}{16}$.

Questo metodo è pesante perchè bisogna controllare più alternative.

3.1.1 Errore percentuale

Bisogna decidere se calcolarlo rispetto alla parte intera o a quella frazionaria. Nel seguente esempio viene calcolato l'errore percentuale rispetto alla parte frazionaria dell'esempio 3.2.

Esempio 3.3

$$\frac{1}{40} : \frac{3}{5} = \frac{1}{40} * \frac{5}{3} = \frac{1}{24} \approx 0.052\%$$

Il massimo errore che si può fare è l'overflow.

3.2 Codifica in virgola mobile

Gli standard della virgola mobile sono: IEEE 754. Questo standard è stato rivisto molte volte e ora viene usato da tutte le codifiche per i numeri in virgola mobile.
Il numero viene separato in 3 parti:

- **S**: Segno
- **e**: Esponente
- **M**: Mantissa

La struttura del numero è quindi:

$$N = \pm \cdot M^{\pm e}$$

Questo permette di dividere il numero in modo da poter scegliere quanti bit dedicare alla mantissa e quanti all'esponente. Si riscontrano però i seguenti problemi:

- Bisogna scegliere la base in cui fare la codifica → base 2
- Bisogna scegliere la divisione di bit tra *segno*, *mantissa* e *esponente* → 1 S, 23 M, 8 e
- La rappresentazione deve essere univoca → $1. \ldots_2$
- Bisogna trovare un modo per rappresentare gli errori

Se la mantissa e la base sono in base 2 la moltiplicazione e la divisione sono agevolate tramite l'utilizzo dello *shift*.

- $0110 \cdot 2 = 1100$ è uno shift a sinistra in binario.

$$\begin{array}{c} 0110 \\ \downarrow\downarrow\downarrow \\ 1100 \end{array}$$

Figura 4: Shift a sinistra in binario

- $1010/2 = 0101$ è uno shift a destra in binario.

$$\begin{array}{c} 1010 \\ \downarrow\downarrow\downarrow \\ 0101 \end{array}$$

Figura 5: Shift a destra in binario

3.2.1 Divisione di bit tra segno, mantissa ed esponente

Un numero è rappresentabile in 2 modi:

- Singola precisione 32 bit $\rightarrow$ float
- Doppia precisione 64 bit $\rightarrow$ double

Prendiamo in considerazione 32 bit, ora dobbiamo decidere quanti bit dedicare alla mantissa e all'esponente.

$$2^{\pm e}$$

$$\begin{aligned} |e| = 4bit &= 2^{+7} \\ 5bit &= 2^{+15} \\ 6bit &= 2^{+31} \\ 7bit &= 2^{+63} \\ 8bit &= 2^{+127} \end{aligned}$$

L'impatto dei bit sull'esponente è doppiamente esponenziale, quindi cresce tantissimo.

- **8 bit** all'esponente, quindi l'esponente può assumere valori da -127 a $+127$.
- **23 bit** alla mantissa, quindi la mantissa può assumere valori da 0 a $2^{23} - 1$
- **1 bit** al segno.

1 bit: segno ±	8 bit: esponente	23 bit: mantissa

Figura 6: Bit dedicati alla codifica in virgola mobile

Per la rappresentazione univoca la mantissa si codifica in virgola fissa. Cioè si parte da una mantissa con un **punto fisso** e dividendo o moltiplicando (shift) si può spostare la virgola per arrivare alla forma **1.00000...** e questa forma è la rappresentazione univoca.

Questa operazioe si chiama **normalizzazione** e visto che la rappresentazione è sempre la stessa l'*1.* non viene rappresentato, quindi viene inserito nella mantissa solo tutto ciò che viene dopo.

$$\begin{gathered}
11111111\ \pm\infty \\
11111110\ +127 \\
\dots \\
00000000\ \pm 0 \\
\dots \\
00000001\ -126 \\
00000000\ -127
\end{gathered}$$

Figura 7: Range dell'esponente

Si è deciso di codificare l'esponente in **Eccesso 127**. Quindi per rappresentare lo zero si usa come esponente il minore numero possibile: $1 \cdot 2^{-127} = 0$. Per codificare i numeri si somma 127 al numero desiderato e visto che i numeri possibili ora vanno da -127 a +127 se codifichiamo il risultato in modulo avremo dei numeri da 0 a 256.

Esempio 3.4
Si vuole decodificare il seguente numero:

$$1\ 01110111\ 0110...0$$

$$M = -(1 + \frac{1}{4} + \frac{1}{8}) * 2^e = -(\frac{11}{8}) * 2^e$$

$$e = (1 + 2 + 4 + 16 + 32 + 64) - 127 = 119 - 127 = -8$$

$$N = -\frac{11}{8} * 2^{-8}$$

Esempio 3.5
Codifica $+(4+\frac{1}{2}+\frac{1}{16})*2^{+34}$

1. *Sappiamo già che il numero è positivo quindi:*

$$S = 0$$

2. *Calcoliamo la mantissa:*

$$4+\frac{1}{2}+\frac{1}{16} = \underbrace{100}_{4_{10}}.\underbrace{10010\ldots0}_{\frac{1}{2}+\frac{1}{16}}$$

3. *La mantissa va normalizzata moltiplicando per 4:*

$$100.10010\ldots0 * 2^{+2} = 1.0010010\ldots0$$

$$M = 0010010\ldots0$$

4. *Calcoliamo l'esponente:*

$$e = 34+2 = 36$$

Si aggiunge 2 perchè abbiamo fatto lo shift di 2 bit.

$$e = 36+127 = 163$$

$$163_{10} = 10100011_2$$

5. *Il numero in virgola mobile è:*

$$0\ 10100011\ 0010010\ldots0$$

- $0\ 00000000\ 0...0 = +0$
- $1\ 00000000\ 0...0 = -0$

Quando l'esponente è tutto 1 e la mantissa tutta 0 allora equivale a $\pm\infty$ in base al primo bit. Se invece la mantissa è diversa da 0 con esponente tutti 1 allora rappresenta un errore NaN.

4 Modelli

Per un progetto bisogna creare un **modello** che rappresenti il sistema. Boole ha cercato di rappresentare tutte le algebre. Lo ha fatto attraverso una quintupla: $< B^n, \cdot, +, \{0,1\} >$

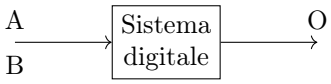


Figura 8: Modello di un sistema digitale

- B^n è l'insieme di valori
- $\{0, 1\}$ è l'alfabeto (sistema binario)
- "$\cdot$" e "$+$" sono 2 operatori

Bool garantisce che si può creare qualsiasi funzione utilizzando soltanto i 2 operatori:

$$f(B^n) \to B^m$$

Esempio 4.1
Si vuole creare un modello con 2 bit in entrata e 1 in uscita:

$$n = 2\ m = 1$$

$O = 1 \leftrightarrow A = B$
$f(B^2) \to B$

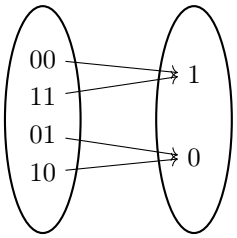


Figura 9: Modello di un sistema digitale

Per mappare i valori in ingresso con quelli di uscita si usa una ***tabella di verità****:*

A B	O
0 0	1
0 1	0
1 0	0
1 1	1

Tabella 7: Tabella di verità

Chiamiamo mintermine un punto dello spazio booleano in ingresso in cui la funzione vale 1. Il maxtermine è il contrario. L'insieme di mintermini $\{{m_0}^a, m_3\}$ *si chiama* ***ON-SET*** *L'insieme dei maxtermini* $\{m_1, m_2\}$

*si chiama **OFF-SET**. Basta uno dei due insiemi (ON-SET, OFF-SET) per definire la funzione.*

$$m_3 = A \cdot B$$

Dire che m_3 è il prodotto delle due variabili è un modo corretto per rappresentarlo.

$$m_0 = \bar{A} \cdot \bar{B}$$

Per rappresentare il mintermine basta fare il prodotto delle variabili se valgono 1 o delle variabili negate se valgono 0.

Per rappresentare la funzione si può usare la somma dei mintermini:

$$O = m_0 + m_3 = \bar{A} \cdot \bar{B} + A \cdot B = 0$$

Questa rappresentazione viene detta: Espressione in somma di prodotti

Teorema 2 *Dato un ON-SET c'è sempre una sola espressione in somma di prodotti che lo rappresenti.*

[a] m_n: n è il valore in modulo del relativo numero binario, m sta per modulo. $m_3 = 11_2$

4.1 Tabelle di verità

4.1.1 Operatore prodotto

A	B	O
0	0	0
0	1	0
1	0	0
1	1	1

Tabella 8: Tabella di verità dell'AND

4.1.2 Operatore somma

A	B	O
0	0	0
0	1	1
1	0	1
1	1	1

Tabella 9: Tabella di verità dell'OR

4.1.3 Operatore negazione

A	O
0	1
1	0

Tabella 10: Tabella di verità del NOT

5 Transistor

È un "comando di accensione" che permette di accendere o spegnere un circuito.

5.1 Transistor CMOS

5.1.1 Transistor N

Mette in collegamento 2 punti:

- Se la corrente è 0V allora non c'è collegamento
- Se la corrente è 3V allora c'è collegamento

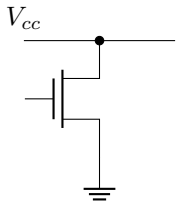


Figura 10: Transistor N

5.1.2 Transistor P

Mette in collegamento 2 punti:

- Se la corrente è 0V allora c'è collegamento
- Se la corrente è 3V allora non c'è collegamento

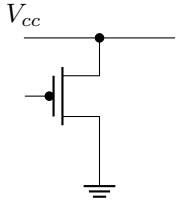


Figura 11: Transistor P

5.1.3 Circuito di negazione (NOT)

Si realizza con un transistor P e uno N in serie.

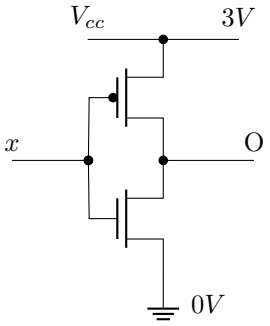


Figura 12: Circuito di negazione

La tabella della verità è:

x	O
0V	3V
3V	0V

Tabella 11: Tabella di verità del circuito

Se assegnamo ad ogni valore un numero binario:

x	O
0	1
1	0

Tabella 12: Tabella di verità del circuito in binario

Si può notare che è la funzione di negazione rappresentata con la seguente porta logica:

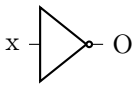


Figura 13: Porta logica NOT

5.1.4 Circuito del prodotto (AND)

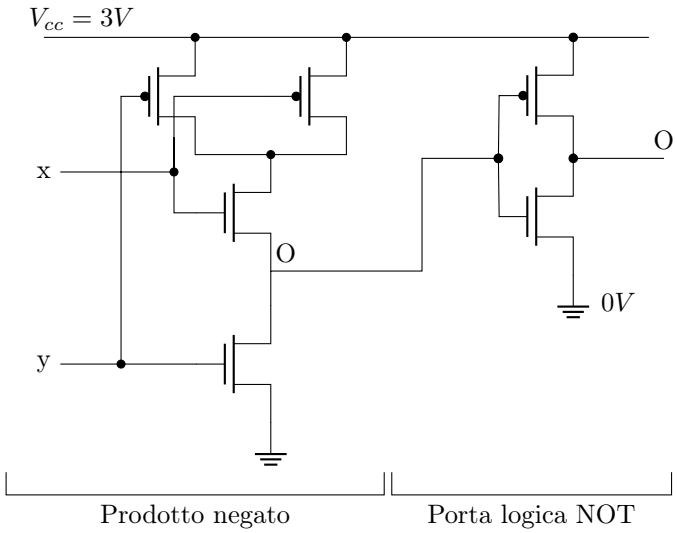


Figura 14: Circuito del prodotto

Il prodotto negato più il NOT è uguale ad un AND:

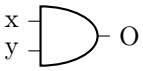


Figura 15: Porta logica AND

La tabella della verità è:

x y	O
0 0	0
0 1	0
1 0	0
1 1	1

Tabella 13: Tabella di verità del circuito

5.1.5 Circuito della somma (OR)

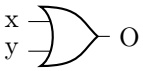


Figura 16: Porta logica OR

La tabella della verità è:

x y	O
0 0	0
0 1	1
1 0	1
1 1	1

Tabella 14: Tabella di verità della somma

6 Espressione in somma di prodotti

Il seguente circuito è un esempio di espressione in somma di prodotti dell'esempio 4.1:

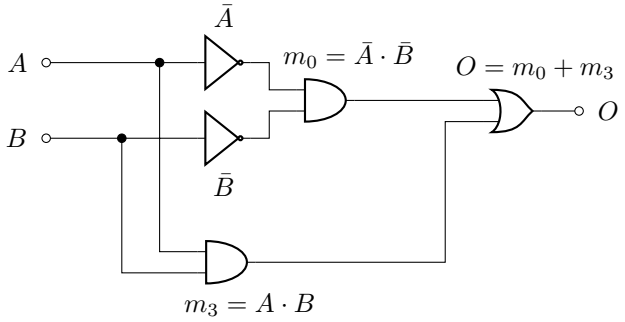


Figura 17: Circuito dell'espressione in somma di prodotti

I circuiti devono spesso tenere conto di alcune specifiche da ottimizzare:

- **Area**: minor numero di porte logiche
- **Latency**: più porte logiche si attraversano più sarà il ritardo
- **Power**: più porte logiche si attraversano più sarà il consumo
- **Safety**: più porte logiche si attraversano più sarà la probabilità di errore

Prendiamo in considerazione la funzione $f(B^3)^1 \to B$:

[1] Il numero di funzioni booleane possibili è $2^{2^3} = 256$ e il valore cresce esponenzialmente con l'aumento dei bit

X Y Z	O
0 0 0	0
0 0 1	1
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	0
1 1 1	1

Tabella 15: Tabella di verità della funzione

ON-SET= $\{m_1, m_3, m_5, m_7\}$
La funzione rappresentata con un'espressione in somma di prodotti è:

$$O = m_1 + m_3 + m_5 + m_7 = \bar{X}\bar{Y}Z + \bar{X}YZ + X\bar{Y}Z + XYZ$$

Proviamo a stimare le dimensioni di questo circuito. Si utilizza il concetto di **letterale** che è una coppia chiave-valore. La funzione O è composta da 12 letterali e questo numero è in relazione con il numero di transistor nel senso che se una funzione ha più letterali di un altra si può già sapere che avrà bisogno di un minor numero di transistor.

6.1 Tecniche di ottimizzazione

La regola principale dell'ottimizzazione è l'**assorbimento**: Preso un prodottp P moltiplicato ad un letterale a e la somma di questo prodotto, ma con il letterale negato $\bar{a}$ allora il risultato è $P \cdot (a + \bar{a})$ dove $(a + \bar{a})$ fa sempre 1, quindi rimane P.

$$aP + \bar{a}P = P \cdot (a + \bar{a}) = P$$

$$\underbrace{2 \cdot (|P| + 1)}_{\text{Cardinalità prima dell'assorbimento}} \quad \Rightarrow \quad \underbrace{|P|}_{\text{Cardinalità dopo l'assorbimento}}$$

Quindi se prendiamo come riferimento la funzione O si può applicare la regola dell'assorbimento per ridurre il numero di letterali:

$$\begin{array}{c} \bar{X}\bar{Y}Z + \bar{X}YZ + X\bar{Y}Z + XYZ \\ \searrow \quad \downarrow \quad \searrow \quad \downarrow \\ \bar{X}Z(\bar{Y} + Y) + XZ(\bar{Y} + Y) \end{array}$$

E riapplicando la regola si arriva al minimo:

$$\begin{array}{c} \bar{X}Z + XZ \\ \searrow \quad \downarrow \\ Z(\bar{X} + X) \end{array}$$

$$Z$$

6.2 Terminologia

Ogni mintermine è un prodotto (o implicante), ma dopo aver applicato la regola di assorbimento non è più un mintermine, ma soltanto prodotto (o implicante).

$$\bar{X}\bar{Y}Z \rightarrow \bar{X}Z$$

La Y non c'è più nel risultato dell'assorbimento, ciò vuol dire che non ci interessa il suo valore perchè non varia il risultato. Si può scrivere sia 11 che $1-1$
Quindi ad esempio:
$Z = --1 = 4$ mintermini: $\{001, 011, 101, 111\}$

Definizione 6.1
Implicante primo *è un implicante non contenuto in nessun altro implicante*

Definizione 6.2
La ***distanza di Hamming*** *è il numero di bit che differenziano 2 codici.*

$01\boldsymbol{10} \rightarrow 01\boldsymbol{01}$ *distanza di Hamming = 2*
$01\boldsymbol{0} \rightarrow 01\boldsymbol{1}$ *distanza di Hamming = 1*

7 Assorbimento svolto graficamente

Prendendo come riferimento la funzione $f(B^3)^1 \rightarrow B$ definita precedentemente (che chiameremo O) si può guardare la funzione come se fosse sul piano cartesiano con centro in un punto qualsiasi. Ogni punto adiacente al centro è un punto con distanza di Hamming = 1.

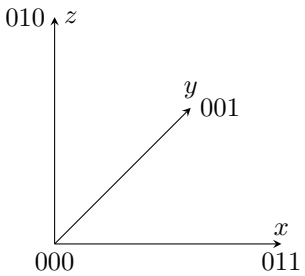


Figura 18: Funzione rappresentata su un piano cartesiano

L'assorbimento può essere fatto soltanto tra gli ON-SET con distanza di Hamming = 1. Per effettuare l'assorbimento ci si posiziona nel punto di un mintermine e si "guarda" in tutte le direzioni per eventuali altri mintermini con cui fare il prodotto.

Nella seguente figura i vertici rossi rappresentano gli OFF-SET e i vertici blu rappresentano gli ON-SET.

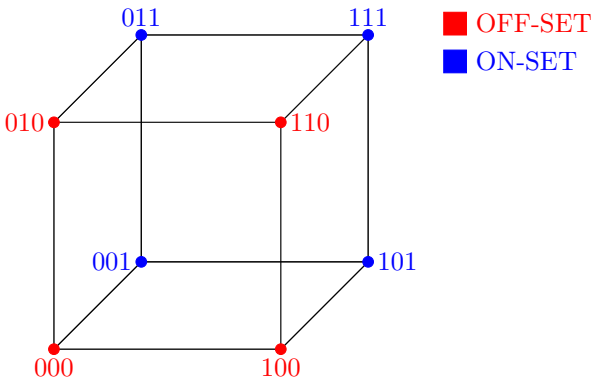


Figura 19: Funzione rappresentata su un cubo

Si trascura la faccia del cubo con l'OFF-SET per rendere la rappresentazione più semplice. Prendendo coppie di vertici dell'ON-SET sullo stesso lato del cubo si può fare il prodotto tra i 2 mintermini:

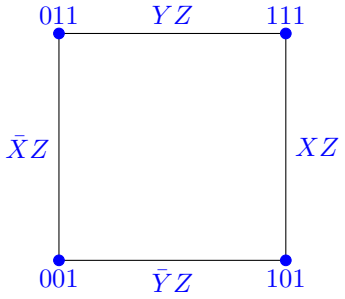


Figura 20: Prima semplificazione

Ora si può fare l'assorbimento anche tra i prodotti ottenuti dall'assorbimento:

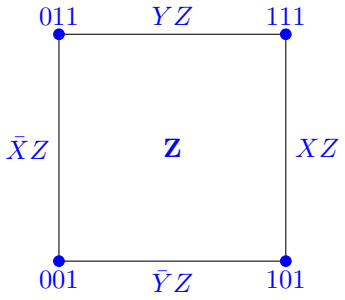


Figura 21: Seconda semplificazione

Si arriva quindi a dire che Z è un **implicante primo** perchè non c'è nessun altro implicante che lo contiene.

Definizioni utili 7.1
Quando si parla di implicante si può anche dire ***sottocubo*** *e l'implicante primo può essere chiamato anche* ***sottocubo di dimensione massima****.*

Implicante = Sottocubo
Implicante primo = Sottocubo di dimensione massima

Esistono condizioni favorevoli (come la funzione O) in cui un implicante primo contiene tutti i mintermini della funzione.
Ci sono più tipi di implicanti primi:

- **Essenziali**: includono almeno un mintermine che non è coperto da nessun altro implicante primo (fanno parte della soluzione finale).
- **Non essenziali**: Implicanti primi che coprono mintermini coperti anche da altri implicanti. Si identificano con l'**algoritmo di copertura**

7.1 Mappe di Karnaugh

Karnaugh ha creato una mappa che permette di rappresentare su un piano tutte le variabili booleane (nel caso della funzione O si mettono i valori del cubo nella tabella) in modo da poter fare l'assorbimento in modo più semplice. I valori posti sopra le celle sono messi in modo che siano a distanza di Hamming = 1. Nella seguente mappa di Karnaugh si possono vedere i valori della funzione O:

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

Tabella 16: Mappa di Karnaugh della funzione O

In questa mappa si può vedere che Z è un implicante primo (o sottocubo di dimensione massima). Le mappe di Karnaugh sono come una sfera, quindi se si va oltre il bordo si torna dall'altra parte.

Un altro esempio di mappa di Karnaugh è il seguente:

Esempio 7.1
Prendiamo in considerazione una funzione casuale a 4 variabili:

VZ \ XY	00	01	11	10
00	1	1	1	1
01	0	0	1	0
11	0	0	1	1
10	1	1	1	1

Tabella 17: Mappa di Karnaugh di una funzione casuale

Si può verificare che ci sono 3 implicanti primi essenziali:

- $\bar{V}$*: essenziale*
- XY*: essenziale perchè copre* 1101
- XZ*: essenziale perchè copre* 1011

$$O = \bar{V} + XY + XZ \quad (5\ letterali)$$

Per capire quali sono gli implicanti primi bisogna raggruppare gli 1 in rettangoli più grandi possibile, ma sempre di grandezza 2^n *(2, 4, 8, 16, ...). Per ciascun raggruppamento bisogna trovare le variabili che non cambiano il loro valore. Per il raggruppamento rosso:*

- X *cambia valore, passando da* 0 *in* 0000 *e* 0100 *a* 1 *in* 1100 *e* 1000*, quindi deve essere esclusa.*
- Y *cambia valore, passando da* 0 *in* 0000 *e* 1000 *a* 1 *in* 0100 *e* 1100*, quindi deve essere esclusa.*
- Z *cambia valore, passando da* 0 *in* 0000 *a* 1 *in* 0010*, quindi deve essere esclusa.*
- $\bar{V}$ *mantiene lo stesso stato in tutto il gruppo, quindi deve essere inclusa nel prodotto risultante*

Lo stesso ragionamento viene applicato per tutti i gruppi, fino ad arrivare al risultato finale.

Le mappe di Karnaugh sono utili soltanto se le variabili sono meno di 5, altrimenti bisogna usare più mappe.

8 Metodo di Quine-McCluskey

Questo metodo ha 2 versioni:

- Funzioni completamente specificate
- Funzioni parzialmente specificate

Si divide in 2 fasi:

1. Si espande il più possibile il problema per cercare il massimo grado di minimizzazione. (ad esempio trattando un *don't care* come 1 per permettere ulteriori ottimizzazioni)
2. Bisogna capire quali servono veramente.

8.1 Esempio con funzione completamente specificata

Esempio 8.1
Prendiamo una funzione completamente specificata

$$O = f(x, y, z, w) = \{m_1, m_4, m_5, m_6, m_7, m_9, m_{11}, m_{14}, m_{15}\}$$

m	$x\ y\ z\ w$
1	0 0 0 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
9	1 0 0 1
11	1 0 1 1
14	1 1 1 0
15	1 1 1 1

Tabella 18: Tabella dei mintermini

36 Letterali.
L'unico caso in cui due stringhe sono a distanza di Hamming $= 1$ *è quando il numero di* 1 *differisce di uno.*
Il metodo di Quine-McCluskey riordina le m *in base al numero di* 1 *che contengono, questo è il primo passo:*

m	x y z w	
1	0 0 0 1	$\surd$
4	0 1 0 0	$\surd$
5	0 1 0 1	$\surd$
6	0 1 1 0	$\surd$
9	1 0 0 1	$\surd$
7	0 1 1 1	$\surd$
11	1 0 1 1	$\surd$
14	1 1 1 0	$\surd$
15	1 1 1 1	$\surd$

Tabella 19: Tabella riordinata

Si individuano i gruppi che sono a distanza di Hamming 1. Nel prossimo passo confrontiamo i gruppi con 1 *bit* $= 1$ *e con* 2 *bit* $= 1$*, se sono a distanza di Hamming 1 allora si mette* don't care *nel bit che cambia. Nella prima colonna c'è la coppia di* m *che viene confrontata.*

m	$x\ y\ z\ w$	
1, 5	0 - 0 1	A
1, 9	- 0 0 1	B
4, 5	0 1 0 -	$\surd$
4, 6	0 1 - 0	$\surd$
5, 7	0 1 - 1	$\surd$
6, 7	1 0 1 -	$\surd$
6, 14	- 1 1 0	$\surd$
9, 11	1 0 - 1	C
7, 15	- 1 1 1	$\surd$
11, 15	1 - 1 1	D
14, 15	1 1 1 -	$\surd$

Tabella 20: Prima semplificazione

Tutti i mintermini della tabella 19 sono coperti da un implicante della tabella 20.

Ora si può semplificare anche la tabella 20 se i don't care sono nella stessa variabile:

m	$x\ y\ z\ w$	
4, 5, 6, 7	0 1 - -	E
6, 7, 14, 15	- 1 1 -	F

Tabella 21: Seconda semplificazione

I valori senza $\surd$ *sono implicanti primi perchè non sono coperti da nessun altro implicante della tabella 21. Anche i 2 valori nella tabella 21 sono*

implicanti primi.
Implicanti primi: A, B, C, D, E, F

$$A = 0 - 01 = \bar{X}\bar{Z}W$$

$$B = -001 = \bar{Y}\bar{Z}W$$

$$C = 10 - 1 = X\bar{Y}W$$

$$D = 1 - 11 = XZW$$

$$E = 01 - - = \bar{X}Y$$

$$F = -11 - = YZ$$

16 Letterali.
Ad ogni passo del metodo di Quine-McCluskey diminuisce il numero di letterali.

Ora bisogna trovare gli implicanti primi essenziali

m	A	B	C	D	E	F
1	1	1				
4					**1**	
5	1				1	
6					1	1
7					1	1
9		1	1			
11			1	1		
14						**1**
15				1		1

Tabella 22: Tabella con implicanti primi

E ed F sono essenziali perchè coprono m_4 *e* m_{14}*. Inoltre E ed F coprono anche* m_5, m_6, m_7 *e* m_6, m_7, m_{14}, m_{15}*.*

Tenendo in mente le m coperte, la tabella diventa:

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

Tabella 23: Tabella senza implicanti essenziali

Euristica *È un metodo per trovare la soluzione corretta, ma non garantisce che sia ottima.*

*Se si cancella ad esempio m_1, m_9 perchè coperti da B, allora B domina A e C domina D per la regola di **dominanza per colonne**. Prendendo la colonna con più elementi ho più probabilità di trovare la soluzione ottima.*

$$O = E + F + B + C = \bar{X}Y + YZ + \bar{Y}\bar{Z}W + X\bar{Y}W$$

10 Letterali.
*La **pseudo-essenzialità** è l'essenzialità dopo aver già fatto un'ottimizzazione.*

Esempio 8.2
Dominanza per righe

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

Tabella 24: Tabella con implicanti primi

- *β dominato da ϵ*
- *δ dominato da γ*

Dobbiamo prendere β e δ perchè cancellando A e D si cancellano anche γ ed ϵ perchè sono dominati. La tabella diventa:

	A	B	C	D
α	1		1	
β	1			1
δ		**1**	1	

Tabella 25: Tabella senza implicanti essenziali

Si fa finta che B e D siano essenziali (essenzialmente scelti a caso).

8.2 Esempio con funzione parzialmente specificata

Potrebbe uscire nel risultato un *don't care*, ad esempio per condizioni di ingresso non utilizzate.

Prendiamo in considerazione una funzione booleana $f(B^n) = B$ parzialmente specificata che viene descritta tramite 3 insiemi:

- **ON-SET**: insieme delle configurazioni per cui vale 1
- **DC-SET**: insieme delle configurazioni per le quali la funzione non è specificata

- **OFF-SET**: insieme delle configurazioni per cui vale 0

L'intersezione fra i 3 insiemi deve essere vuota, mentre l'unione è l'insieme di tutte le configurazioni possibili.

Per conoscere tutti e 3 gli insiemi basta conoscerne 2 di essi.

Esempio 8.3
Funzione parzialmente specificata

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

Tabella 26: Tabella della funzione parzialmente specificata

$$ON - SET = \{m_4, m_{10}, m_{11}, m_{13}, m_{14}, m_{15}\}$$
$$DC - SET = \{m_3, m_5, m_6, m_7\}$$

Il primo passo è quello di ampliare il problema, quindi si considerano i don't care come 1, e poi rioridnare i mintermini in base al numero di 1.

m	x y z v	
4	0 1 0 0	√
3	0 0 1 1	√
5	0 1 0 1	√
6	0 1 1 0	√
10	1 0 1 0	√
7	0 1 1 1	√
11	1 0 1 1	√
13	1 1 0 1	√
14	1 1 1 0	√
15	1 1 1 1	√

Tabella 27: Tabella riordinata

Il secondo passo è quello di tentare la semplificazione:

m	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 -	√

Tabella 28: Prima semplificazione

Ora si applica di nuovo la semplificazione:

m	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

Tabella 29: Seconda semplificazione

Visto che non si possono fare ulteriori semplificazioni A, B, C, D, E sono tutti implicanti primi.

Ora si cerca di capire quali sono gli implicanti primi essenziali considerando però soltanto l'ON-SET:

m	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

Tabella 30: Tabella con implicanti primi

Si cancellano le righe A, C, E perchè sono implicanti primi essenziali, quindi si coprono anche tutte le righe delle colonne A, C, E che contengono 1. Si arriva quindi a coprire tutta la tabella e il risultato finale è:

$$O = A + C + E$$

9 Circuiti Combinatori

- **PROM**: Programmable Read Only Memory (ROM)
- **PLA**: Programmable Logic Array, attivano diverse porte logiche

I circuiti a 2 livelli sono composti da 2 livelli di porte logiche:

1. Porte AND
2. Porte OR

9.0.1 PLA (Programmable Logic Array)

Esempio 9.1
Un esempio di PLA:

$$O_1 = \bar{X}Y + YV + XZ$$

$$O_2 = \bar{X}\bar{Y} + YV + X\bar{Z}$$

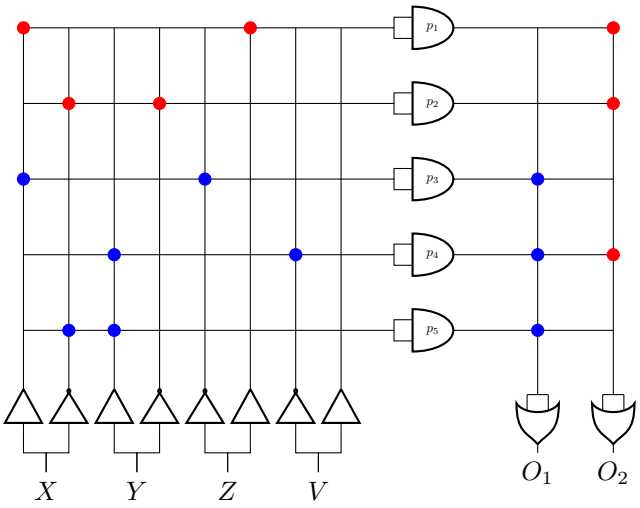


Figura 22: PLA

Prende in input coppie di variabili e le assegna ad un array di AND e l'output di questi AND viene assegnato ad un array di OR.

9.0.2 CPLD (Complex Programmable Logic Device)

I CPLD attivano diversi PLA:

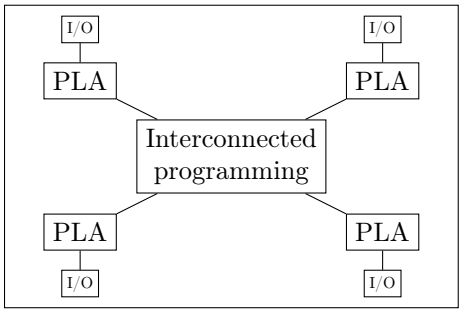


Figura 23: CPLD

9.0.3 FPGA (Field Programmable Gate Array)

I FPGA attivano diversi CPLD:

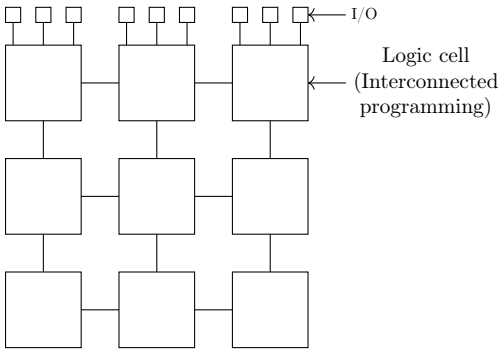


Figura 24: FPGA

9.0.4 SoC (System on Chip)

I SoC attivano diversi CPLD

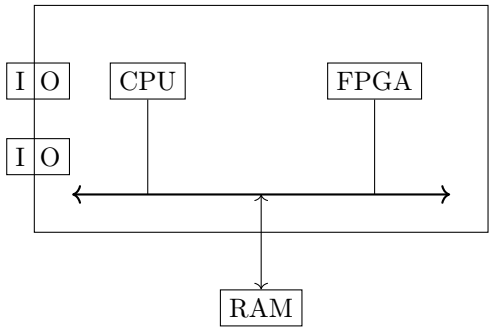


Figura 25: SoC

10 Laboratorio

10.1 Modellazione in SIS

È il successore di *Espresso* e permette di fare la sintesi di circuiti, cioè si genera passo dopo passo il layout per il silicio.

Il modello di codice di SIS è il seguente:

```
.model <model-male> // nome della funzione
.inputs <input-list> // elenco degli input
.outputs <output-list> // elenco degli output

.names // On-set/Off-set per ogni input
<command>
...
<command>

.end // il file deve essere
```

Esempio 10.1
Prendiamo in considerazoine la tabella di verità dell' implicazione logica:

a b	a $\Longrightarrow$ b
0 0	1
0 1	1
1 0	0
1 1	1

Tabella 31: Tabella di verità dell'implicazione logica

che si può scrivere anche nel seguente modo:

a b	a $\Longrightarrow$ b
0 -	1
1 0	0
1 1	1

Tabella 32: Tabella di verità ridotta

```
.model IMPLIES
.inputs I1 I2
.outputs O

.names I1 I2 O
// si definisce l'on-set o l'off-set
0- 1
11 1
.end
```

Esiste il comando **.exdc** che permette di specificare il **don't care set** di una funzione booleana.
La sintassi è la seguente

```
.exdc
.names [lista delle variabili]
[lista delle configurazioni, mettendo 1 come output] // l'1 come
output non vuol dire che si sta forzando il don't care a 1, ma
serve solo per il parser
```

Lista di comandi utili:

- **read_blif**: carica il modello sis;
- **simulate [valori in bit, separati da spazi]**: esegue un passo di simulazione del circuito;
- **help**: mostra i comandi disponibili;

- **help [comando]**: mostra la descrizione del comando;
- **print_stats**: fornisce informazioni sul circuito, quali numero di input (pi), output (po), elementi di memoria (latches), letterali (lits(sop)), numero di nodi (nodes);
- **quit**: esce da sis;
- **write_blif [nome file]**: salva il modello sis in un file;
- **write_blif**: stampa a video il file blif del circuito caricato in memoria senza dover lasciare l'ambiente SIS.
- **write_eqn**: stampa a video l'equazione del circuito caricato, l'espressione è scritta in somma di prodotti dove il simbolo "!" indica la negazione del letterale che lo segue.
- **full_simplofy**: ottimizza il circuito caricato, ma sarebbe meglio usare *write_eqn* prima di usare questo comando.

10.2 Ottimizzazione

Ottimizzare significa ridurre l'**area** o il **ritardo**.

- L'**area** di un circuito digitale è misurabile come il numero di porte logiche a 2 ingressi necessarie per la realizzazione del circuito
- Il **ritardo** di un circuito digitale è misurabile come il massimo numero di porte logiche che un segnale applicato agli ingressi deve attraversare per raggiungere l'uscita.
- L'**ottimizzazione** di un circuito digitale consiste nella trasformazione del circuito in uno *funzionalmente equivalente* avente area e/o ritardo minimi. Lo scopo è quello di ottenere circuiti piccoli e veloci.

Bisogna anche tenere conto del consumo energetico e della temperatura.

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

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

Per **circuiti a una uscita** esiste un metodo esatto (Mc Cluskey) che permette di trovare gli implicanti primi essenziali ed esiste sia un metodo esatto che approsimato anche per ottenere la funzione di copertura ottima.

Per **circuiti a più uscite** esiste solo un metodo *approssimato* per trovare la copertura ottima che si basa sul metodo esatto di identificazione degli implicanti primi essenziali di ogni singola uscita.

10.3 Modelli gerarchici

Un **componente gerarchico** è un componente del sistema che contiene al proprio interno dei sottocomponenti, ossia delle **istanze di altri componenti**. (Un sommatore di n bit può essere costruito da n sommatori).

Un componente si può includere in un modello *blif* con le seguenti istruzioni:

```
.subckt nomeComp paramFormale=paramAttuale // Crea l'istanza di un
  componente
.search nomeFileComp.blif // Include il file blif di un componente
```

Ad esempio, per includere un sommatore a 4 bit:

```
// sommatore.blif
.model SOMMATORE
.inputs A B CIN
.outputs O COUT

.names A B K
10 1
01 1

.names K CIN O
10 1
01 1

.names A B CIN COUT
11- 1
1-1 1
-11 1
.end
```

Si vuole includere il sommatore per creare un sommatore a 2 bit:

```
.model SOMMATORE2
.inputs A1 A0 B1 B0 CIN
.outputs O1 O0 COUT

.subsckt SOMMATORE A=A0 B=B0 CIN=CIN O=O0 COUT=C0
.subsckt SOMMATORE A=A1 B=B1 CIN=C0 O=O1 COUT=COUT
.search sommatore.blif
```

Ci sono dei componenti che possono essere utili, ad esempio:

- Generazione di un bit costante a 0:

```
.model zero1
.outputs uscita
.names uscita
.end
```

- Generazione di un bit a costante 1:

```
.model uno1
.outputs uscita
.names uscita
1
.end
```

10.4 Ottimizzazione approssimata multilivello

Consente al progettista di bilanciare area e ritardo di di un circuito con maggior grado di libertà rispetto alla minimizzazione a 2 livelli.

Tuttavia, non esistono tecniche esatte efficienti che portino alla realizzazione di configurazioni ottime usando la minimizzazione multi-livello; pertanto si ricorre a tecniche euristiche che garantiscono buone soluzioni in tempi di calcolo ragionevoli.

Alcuni comandi utili sono le seguentu:

- **sweep**: eliminazione dei nodi con un'unica linea di ingresso e di nodi con valore costante
- **eliminate**: eliminazione di un nodo interno alla rete. Si consideri che il nodo N rappresenti la funzione $y = (a + b) * c$, l'eliminazione di N prevede la sostituzione della variabile y in tutti i nodi che la utilizzano con l'espressione booleana $(a + b) * c$.
- **resub**: sostituzione di un nodo interno con un'insieme di nodi la cui funzionalità sia equivalente a quella del nodo sostituito. L'operazione viene effettuata per diminuire la complessità di un nodo
- **extract**: estrazione di una sottoespressione comune a più nodi che viene rappresentata con un nuovo nodo
- **simplify**: riduzione della complessità di ogni singolo nodo con algoritmo di Quine-McCluskey

10.5 Mapping tecnologico

La realizzazione di un circuito può usare due tecnologie fondamentali:

- **ASIC**: utilizzando le porte logiche di base messe a disposizione da chi si occupa di fondere il semiconduttore

– **FPGA**: utilizzando le porte logiche di base disponibili all'interno della board FPGA a disposizione

In entrambi i casi, le porte logiche di base sono contenute in una **libreria tecnologica** , la quale specifica anche le caratteristiche di **area e ritardo** delle porte logiche contenuta nella libreria.

I comandi principali sono:

- **read_library libreria**: carica la libreria tecnologica di nome "libreria". Le librerie sono specificate nel formato **genlib** (estensione .genlib, ad esempio synch.genlib e mcnc.genlib)
- **print_library**: visualizza informazioni inerenti alla libreria caricata
- **map**: esegue l'operazione di mapping. Le opzioni principali del comando sono:
  - **-m 0**: permette di ottenere un circuito ottimizzato rispetto all'area.
  - **-n 1**: permette di ottenere un circuito ottimizzato rispetto al ritardo.
  - **-s**: visualizza informazioni relative ad area e ritardo dopo il mapping:
    - **total gate area**: fornisce il valore dell'area come numero di celle standard della libreria tecnologica
    - **maximum arrival time**: indica il ritardo
- **write_blif -n**: mostra la rappresentazione del circuito associata alle porte della libreria
- **print_delay**: stampa informazioni relative al ritardo del circuito
- **reduce_depth**: riduce la lunghezza dei cammnini critici

10.6 Sintesi a N livelli

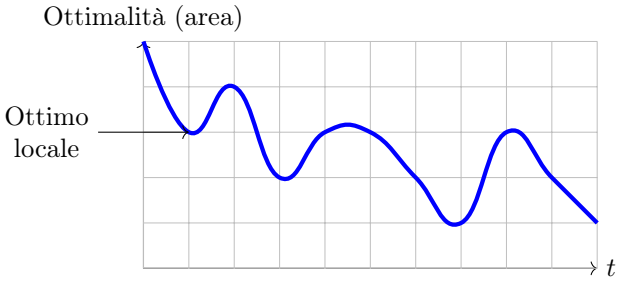


Figura 26: Ottimalità nel tempo

Definizioni utili 10.1
Il cammino critico è il percorso più lungo che il segnale deve effettuare e indica il ritardo.

In un circuito a 2 livelli riducendo l'area si riduce anche il ritardo.

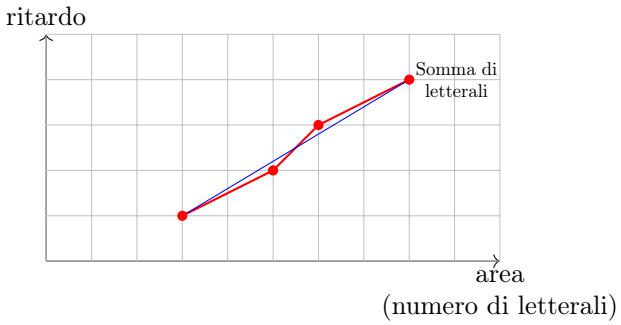


Figura 27: Ottimalità di circuiti a 2 livelli

In circuiti a N livelli il grafico del ritardo e dell'area è un'iperbole:

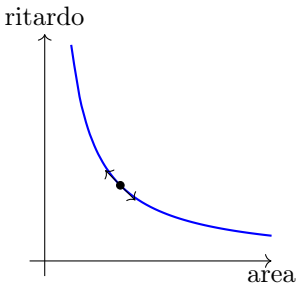


Figura 28: Curva di Pareto

Quindi bisogna trovare un buon compromesso tra area e ritardo, perchè diminuendo troppo l'area aumenta anche il ritardo.

10.6.1 Network

DAG Direct Acyclic Graph. Non permettono la creazione di cicli (Acyclic).

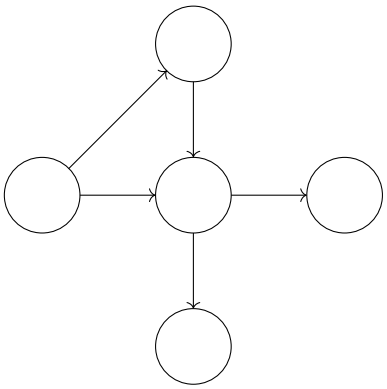


Figura 29: Esempio di DAG

Gli input sono dei nodi, e ogni nodo di ingresso avrà soltanto archi uscenti. I nodi di uscita avranno solo archi entranti. Il numero di letterali è dato dalla somma dei letterali di tutte le funzioni.

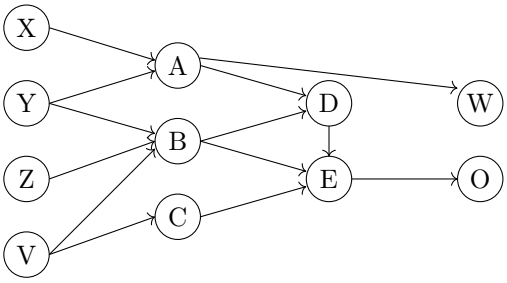


Figura 30: Esempio di network

Il numero di nodi può essere messo in relazione con il ritardo (più nodi ci sono più nodi devono attraversare i segnali, quindi più ritardo).

- Area: numero di letterali;
- Delay: numero di nodi;

10.6.2 Algoritmi

- **simplify**(Quine-McCluskey):trova la minimizzazione massima per ogni nodo del network
- **full_simplify**: è l'algoritmo più pesante per semplificare il circuito e utilizza BDB (Binady Decision Diagram) come struttura dati (struttura a grafo).

10.7 Script

In sis c'è una lista di **script** che permettono di minimizzare il circuito, ad esempio:

- **rugged**: permette di minimizzare il circuito;
- **sweep**: ripulisce il circuito dai nodi inutili
- **eliminate**: elimina un nodo dal circuito condensandolo nei nodi che lo circondano

10.7.1 full_simplify

Prendiamo come esempio il seguente network:

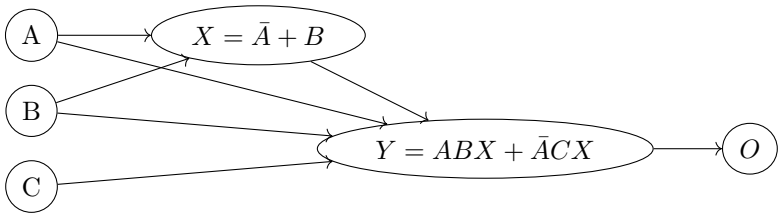


Figura 31: Esempio di network

Con un semplice **simplify** sul nodo X il circuito resta uguale. Si può notare dalla mappa di Karnaugh del nodo X che tutti i mintermini sono coperti e sono essenziali

AB / CX	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	1	1	1	0
10	0	0	0	0

Tabella 33: Mappa di Karnaugh del nodo X

Il **full_simplify**, invece calcola i **controllability don't care set**[2], che sarebbero le configurazioni di ingresso che non si possono presentare. ad esempio: ABX= (0,0,0), (0,1,0), (1,0,1), (1,1,0).
Se prendiamo in considerazione la tabella di verità del nodo X si può ottenere il controllability don't care set:

[2]Gli **observability don't care** sono valori di ingresso che non servono per calcolare l'uscita

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

Tabella 34: Tabella di verità del nodo X

A	B	C	X
0	0	-	0
0	1	-	0
1	0	-	1
1	1	-	0

Tabella 35: Controllability don't care set

Se si inseriscono questi don't care nella mappa di Karnaugh si osserva che sono disponibili altri raggruppamenti:

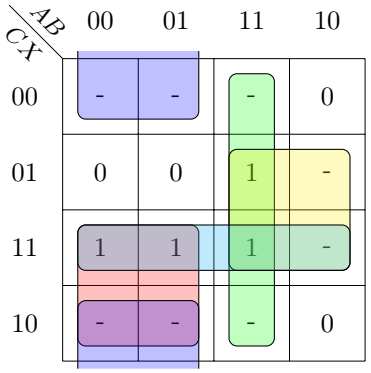
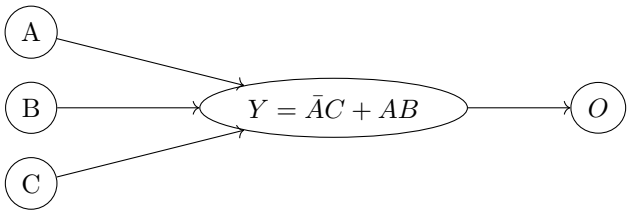


Tabella 36: Mappa di Karnaugh del nodo X con i don't care

Il risultato finale sarà:

$$Y = \bar{A}C + AB$$

Usando questi don´t care si può semplificare il circuito e si nota che il nodo X non è più necessario, quindi si può eliminare (sweep) e il network finale sarà il seguente:



Per poter realizzare il circuito minimizzato a 2 livelli bisogna fare il technology mapping che sarebbe l'ultimo passaggio per realizzare il network in qualcosa di reale.

- **FPGA**: $f(B^5) \to B$
- **ASIC**: libreria di porte logiche (composte da più porte elementari e date dal produttore)

L'algoritmo più famoso di technology mapping si chiama **tree mapping** e dopo l'applicazione di questo algoritmo si avrà una misurazione per area, ritardo e potenza.

10.8 Modellazione in Verilog

Verilog è un linguaggio di modellazione più simile ad un linguaggio di programmazione ed è divisa nei nei seguenti livelli di astrazione (di più alto livello):

- **Behavioural**: il comportamento del sistema hardware viene descritto specificando le azioni da comprendere, in maniera quasi algoritmica
- **Strutturale**: il comportamento del sistema hardware veiene descritto come aggregazione e interconnesione di componenti, i quali a loro volta specificheranno delle funzionalità in uno dei tre stili di modellazione
- **Gate-level**: il comportamento del sistema hardware viene descritto come aggregazione di porte logiche di base, quali porte AND, OR, NOT, ecc.

10.8.1 Tipi di dato

I segnali di Verilog possono essere di 4 tipi, corrispondenti a stati fisici di segnali hardware:

- **0**: zero logico, oppure una condizione falsa
- **1**: uno logico, oppure una condizione vera
- **X**: valore sconosciuto, oppure don't care
- **Z**: alta impedenza del segnale

È possibile esprimere valori numerici a più bit utilizzando i letterali numerici, la cui struttura di base è: $[D]'[B][Valore]$

- **D**: dimensione del segnale (numero di bit) espresso come numero in base 10

- **B**: base, può essere una delle seguenti:
  - **d**: base 10
  - **b**: base 2
  - **o**: base 8
  - **h**: base 16

Un esempio di letterale numerico è: $32'd17 = (17)_{10}$
oppure: $32'hABCD = (43981)_{10}$

Gli elementi di memorizzazione e le connessioni tra componenti vengono rappresentati mediante 2 costruttori principali:

- **wire**: rappresenta la connessione tra elementi, il cui valore è guidato solamente dall'uscita di una rete logica (oppure, mediante *assegnamenti continui*)
- **reg**: modella il comportamento di un elemento di memorizzazione

Per specificare **modelli a livello comportamentale** è possibile utilizzare variabili, simili a quelle dei linguaggi di programmazione, ad esempio:

```
integer i; // variabile intera a 32 bit.
real r;    // variabile reale (floating point)
time t;    // variabile temporale
```

10.8.2 Moduli e interfacce

Il costrutto **module** è l'unità fondamentale di un modello Verilog.

Un modulo rappresenta un componente del sistema, caratterizzato da **un'interfaccia**, ossia la definizione delle sue **porte di ingresso** e **porte di uscita** ad esempio:

```
module complemento(input [3:0] in, output [3:0] out);
begin
  ...
endmodule
```

Il [3:0] indica che è da 4 bit e indica che il bit più grande è a sinistra, cioè little endian (il più piccolo bit a destra), il corrispondente è big endian (il più grande bit a destra).

Un **blocco sequenziale** è il costrutto principale per la modellazione del comportamento di un sistema elettronico. È descritto da un insieme di istruzioni comprese tra le parole *begin* e *end*. Esistono 2 tipi di blocchi sequenziali:

- **always**: blocco che viene eseguito ogniqualvolta si verifica un evento (l'output cambia quando cambiano gli input)
- **initial**: blocco che descrive il comportamento del sistema a partire dall'istante iniziale della sua esecuzione

La sintassi dell'always è la seguente:

```
always @(evento)
begin:
  ...
end
```

La sintassi dell'initial è la seguente:

```
initial begin: nome
  ...
end
```

10.8.3 Costrutti condizionali e cicli

Il linguaggio Verilog mette a disposizione vari costrutti condizionali e iterativi:

- **if**:

```
if(condizione) begin
  [istruzioni sequenziali]
end
else if(condizione) begin
  [istruzioni sequenziali]
end else
  [istruzioni sequenziali]
end
```

- **case**:

```
case(espressione)
  valore_1: begin
    [blocco di istruzioni sequenziali]
  end
  valore_2: begin
    [blocco di istruzioni sequenziali]
  end
    ...
  default: begin
    [blocco di istruzioni sequenziali]
  end
endcase
```

- **while**:

```
  while(condizione) begin
    [istruzioni sequenziali]
  end
```

- **for**:

```
  for(inizializzazione; condizione; incremento) begin
    [istruzioni sequenziali]
  end
```

- **repeat**:

```
  repeat(nunmero) @ (evento) begin
    [istruzioni sequenziali]
  end
```

10.8.4 Assegnamenti

Si assegnano dei segnali che sono tipicamente oggetti di tipo **wire**. Vengono continuamente assegnati mediante il costrutto:

```
  assign segnale = [ritardo] valore o espressione;
```

Per assegnare i registri esistono 2 tipi di assegnamenti:

- **Assegnamenti non bloccanti**: vengono fatti in sequenza (come in C)

```
registro <= valore o espressione;
```

- **Assegnamenti bloccanti**: vengono fatti in parallelo

```
registro = valore o espressione;
```

10.8.5 Esempi

Esempio 10.2 (Calcolo del complemento a 2)

```
module complemento(input [3:0] in, output [3:0] out);
  integer i;
  reg [3:0] negato;

  always @(in)
  begin
    for(i = 0; i < 4; i = i + 1) begin
      negato[i] <= !in[i];
    end

  end
  assign out = negato + 1;
endmodule
```

Esempio 10.3 (Sommatore a 2 bit)

```
module somma2bit(
  input [3:0] a,
  input [3:0] b,
  output reg [3:0] out);

  always @(a or b) begin
    out = a + b;
  end
endmodule
```

10.8.6 Testbench

Il **testbench** è un modulo che genera gli ingressi e legge le uscite del modello che si sta progettando. Dunque, il suo ruolo è quello di stimolare l'esecuzione del modello in esame. Esistono 2 alternative principali:

- **Module**: ad-hoch che istanzia il componente in esame
- **script**: di comandi che generano gli ingressi del componente in esame.

I comandi utili alla creazione di un testbench sono:

- **$dumpfile(nomefile.vcd)**: crea un file di tracce (waveform) in cui salvare l'andamento della simulazione
- **$display(argomenti)**: stampa a video la stringa creata specificando gli argomenti (sintassi simile a C)

Un esempio di testbench è il seguente:

Esempio 10.4

```
'timescale 1ns / 1ps

module tb();
  reg [3:0] in;
  wire [3:0] out;
  integer i;
  complemento c(.in(in), .out(out));

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
    for(i = -7; i <= 7; i = i + 1) begin
      in <= i;
      #2;
      $display("in: %d -> out: %d",
        $signed(in), $signed(out));
      end
    end
  endtask
endmodule
```

10.8.7 Simulazione di Verilog

Si può utilizzare **EDAPlayground** che è un insieme di strumenti online.

10.8.8 Modellazione a Gate Level

Verilog permette di specificare modelli più vicini a quella che sarà l'implementazione finale del circuito. Il linguaggio mette a disposizione una libreria di

porte logiche standard che implementano le funzioni booleane corrispondenti. Le porte nella libreria sono: AND, OR, XOR, NAND, NOR, XNOR e NOT.

Ogni porta logica richiede di specificare i segnali di uscita seguiti da quelli di ingresso. Quindi ad esempio, $O = A \vee B$ sarà implementata specificando:

```
or (O, A, B);
```

La rappresentazione a gate level può essere espressa anche mediante assegnamenti continui, conoscendo l'operatore bit-wise corrispondente ad ogni porta.

Esempio 10.5 (Sommatore a 1 bit gate level)

```
module sommatore( input A, B, CIN, output O, COUT);
  wire V, W, Z;

  xor(V, A, B);
  xor(O, V, CIN);
  and(W, A, B);
  and(Z, V, CIN);
  or(COUT, W, Z);
endmodule
```

10.9 Circuiti sequenziali e macchine a stati finiti

Vedere il capitolo 12 e 13.

10.9.1 Circuiti sequenziali

I circuiti sequenziali possono essere:

- **Sinctoni** se i valori delle uscite assumono significato in corrispondenza di un evento su un segnale di sincronismo (solitamente detto clock).
- **Asincroni** se i valori delle uscite cambiano al variare degli ingressi senza tenere conto di un segnale di sincronismo.

L'uscita di un circuito sequenziale dipende dai valori in ingresso anche negli istanti passati, occorre quindi una sorta di memoria di tale storia passata. Questa memoria è rappresentata dallo **stato**.

L'elettronica digitale mette a disposizione due tipi di componenti detti **latch** e **flip-flop** in grado di memorizzare il valore di un bit (stato binario). Occorre quindi codificare tutti gli stati che può assumere il circuito mediante dei numeri binari e poi utilizzare dei latch o flip-flop per memorizzare tali numeri.

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

10.9.2 Circuiti sequenziali in SIS

È necessario definire la tabella delle transizioni ed eventualmente definire manualmente la codifica degli stati (esiste un comando che permette di eseguire la codifica in modo automatico).

1. La tabella delle transizioni viene scritta tra:

```
.start_kiss
.i 2 // Input
.o 2 // Output
.s 2 // Numero degli stati
.r NAME // Stato di reset
.p 5 // Numero di transizioni
.end_kiss
```

   Letransizioni devono essere specificate come un insieme di righe che riportano in ordine: valore degli ingressi, stato presente, stato prossimo, valore delle uscite.

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

   - il numero di segnali di input
   - il numero di segnali di output
   - il numero di transizioni
   - il numero di stati
   - lo stato di reset

2. Dopo la tabella delle transizioni (dopo *.end_kiss*) possono essere riportate le istruzioni necessarie per definire la codifica degli stati qualora non si decida di farla definire a SIS in modo automatico. La keyword da utilizzare per definire la codifica è .code seguita dal nome dello stato e dalla sua codifica binaria.

Un esempio di implementazione è il seguente:

Esempio 10.6
Implementazione di un semaforo con priorità

```
.model SEMAFORO
.inputs TRAFFICONS TRAFFICOEO
.outputs LUCENS LUCEEO
.start_kiss
.i 2
.o 2
.s 2
.r VERDENS
.p 5
```

```
00 VERDENS VERDENS 10
01 VERDENS VERDEEO 01
1- VERDENS VERDENS 10
0- VERDEEO VERDEEO 01
1- VERDEEO VERDENS 10
.end_kiss
.code VERDENS 0
.code VERDEEO 1
.end
```

10.9.3 Minimizzazione ed assegnamento degli stati

3. 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* se non è stata fatta la codifica binaria degli stati.

4. Generazione logica 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*, in questo caso non c'è bisogno del comando *state_minimize stamina*.
   - altrimenti occorre assegnare automaticamente gli stati con il comando *state_assign jedi* (che genera anche le funzioni δ e λ). Attenzione che prima occorre minimizzare gli stati e poi farne l'assegnazione.
   - Eseguire la minimizzazione delle funzioni δ e λ, ad esempio lanciando lo script *script.rugged* come visto per la minimizzazione dei circuiti combinatori.

10.9.4 Modellare FSM in Verilog

Dopo aver dichiarato due segnali di stato: uno che contiene lo stato corrente, e uno che contiene lo stato prossimo. Ad ogni ciclo di clock, lo stato corrente viene aggiornato con lo stato prossimo.

Bisogna avere 2 blocchi sequenziali:

- **Blocco sincrono**: che aggiorna ad ogni ciclo di clock lo stato utilizzando lo stato prossimo. Eventualmente, il blocco sequenziale può utilizzare un costrutto condizionale per gestire la fase di reset.
- **Blocco asincrono**: sensibile a tutti i segnali di ingresso della funzione δ, che calcola lo stato prossimo.

```
module semaforo(
  input rst, clk, trafficons, trafficoeo,
  output reg lucens, luceeo);
    reg stato = 1'b0,
        stato_prossimo = 1'b0;
    always @(posedge clk) begin : SEQ
      if(rst) stato = 1'b0;
      else stato = stato_prossimo;
    end
    always @(stato, trafficons, trafficoeo) begin : COMB_OUT
      case(stato)
        1'b0:
          if(~trafficons && trafficoeo) begin
            lucens = 1'b0; luceeo = 1'b1;
            stato_prossimo = 1'b1;
            end else begin
            lucens = 1'b1; luceeo = 1'b0;
            stato_prossimo = 1'b0;
        end
        1'b1:
          if(~trafficons) begin
            lucens = 1'b0; luceeo = 1'b1;
            stato_prossimo = 1'b1;
          end else begin
            lucens = 1'b1; luceeo = 1'b0;
            stato_prossimo = 1'b0;
          end
      endcase
    end
endmodule
```

10.9.5 Testbench per circuiti sequenziali

- È necessario stimolare il segnale di clock. Si utilizza un costrutto always temporizzato che inverte il clock.
- Si consiglia di utilizzare la funzione $\$finish$ al termine del blocco sequenziale *alwats begin*

Esempio 10.7
Esempio di testbench per il semaforo:

```
module tb_semaforo();

  reg rst, clk, trafficons, trafficoeo;
  wire lucens, luceeo;

  semaforo sem(rst, clk, trafficons, trafficoeo, lucens,
luceeo);

  always #10 clk = ~clk;

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
    clk = 1'b0;
    ... # Assegnamenti sugli ingressi.
    $finish;
  end
endmodule
```

10.10 Progettare FSMD in SIS

Per includere un componente all'interno di un modello *.blif* si usano le seguenti istruzioni:

```
.subckt nomecomponente parametroformale=parametroattuale ...
.search nomefilecomponente.blif
```

10.10.1 Esempi

Esempio 10.8
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*
  - ***re***: *rising edge*
  - ***ah***: *active high*
  - ***al***: *active low*
  - ***as***: *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. Ciò significa che il registro utilizza il clock generale del circuito in cui è inserito (a ogni simulate viene generato un colpo di clock).*
- ***init-val***: *è lo stato iniziale del latch, che può essere 0,1,2,3. "2" indica "don't care" e "3" indica "unknown" o non specificato.*

Un esempio di registro è il seguente:

```
.model REGISTRO
.inputs A
.outputs O
.latch A O re NIL 0
.end
```

Utilizzando il registro a 1 bit si può creare un registro a N bit:

```
.model REGISTRO4
.inputs A3 A2 A1 A0
.outputs O3 O2 O1 O0
.subckt REGISTRO A=A3 O=O3
.subckt REGISTRO A=A2 O=O2
.subckt REGISTRO A=A1 O=O1
.subckt REGISTRO A=A0 O=O0
.search registro.blif
.end
```

Esempio 10.9
Sommatore a 1 bit:

```
.model SOMMATORE
.inputs A B CIN
.outputs O COUT
.names A B K
10 1
01 1
.names K CIN O
10 1
01 1
.names A B CIN COUT
11- 1
1-1 1
-11 1
.end
```

Un sommatore a 2 bit si può creare utilizzando il sommatore a 1 bit:

```
.model SOMMATORE2
.inputs A1 A0 B1 B0 CIN
.outputs O1 O0 COUT
.subckt SOMMATORE A=A0 B=B0 CIN=CIN O=O0 COUT=C0
.subckt SOMMATORE A=A1 B=B1 CIN=C0 O=O1 COUT=COUT
.search sommatore.blif
.end
```

Esempio 10.10
Multiplexer a 4 ingressi 1 bit ciascuno:

```
.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
```

Le prime due colonne sono i segnali di controllo. Se $S1 = 0$ *e* $S0 = 0$ *allora viene selezionato* $i0$ *e così via.*
Multiplexer a 4 ingressi 2 bit ciascuno:

```
.model MUX2
.inputs X1 X0 a1 a0 b1 b0 c1 c0 d1 d0
.outputs o1 o0
.subckt MUX1 S1=X1 S0=X0 i3=a1 i2=b1 i1=c1 i0=d1 out=o1
.subckt MUX1 S1=X1 S0=X0 i3=a0 i2=b0 i1=c0 i0=d0
out=o0
.search mux1.blif
.end
```

Esempio 10.11
Multiplexer a 2 ingressi 3 bit ciascuno:

```
.model MUX3
.inputs A2 A1 A0 B2 B1 B0 S
.outputs O2 O1 O0
.names S A2 B2 O2
11- 1
0-1 1
.names S A1 B1 O1
11- 1
0-1 1
.names S A0 B0 O0
11- 1
0-1 1
.end
```

Esempio 10.12
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
```

Esempio 10.13
Comparatore a 4 bit:

```
.model UGUALE4
.inputs A3 A2 A1 A0 B3 B2 B1 B0
.outputs O
.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 O
1111 1
.search xnor.blif
.end
```

Gli xnor danno in output 1 *se i bit sono uguali. Se tutti gli xnor* X *sono* 1 *allora* $O = 1$ *altrimenti* $O = 0$.

Esempio 10.14
"Maggiore" a 6 bit (restituisce 1 se il numero rappresentato dai primi 6 bit è maggiore del numero rappresentato dai successivi 6 bit):

```
.model 6_gt
.inputs A5 A4 A3 A2 A1 A0 B5 B4 B3 B2 B1 B0
.outputs AgtB
.subckt xor A=A5 B=B5 X=X5
.subckt xor A=A4 B=B4 X=X4
.subckt xor A=A3 B=B3 X=X3
.subckt xor A=A2 B=B2 X=X2
.subckt xor A=A1 B=B1 X=X1
.subckt xor A=A0 B=B0 X=X0
.names A5 A4 A3 A2 A1 A0 X5 X4 X3 X2 X1 X0 AgtB
1-----1----- 1
-1----01---- 1
--1---001--- 1
---1--0001-- 1
----1-00001- 1
-----1000001 1
.search xor.blif
.end
```

10.10.2 Passi di progettazione

- Modellare la FSM del Controllore mediante la tabella delle transizioni e assegnare la codifica agli stati con $state\_assign\ jedi$
- Salvare il Controllore ottenuto dopo la codifica degli stati in un file diverso da quello originale usando il comando $write\_blif\ nomefile$
- Modellare il Datapath come una interconnessione di componenti funzionali, quali sommatori, moltiplicatori, multiplexer, registri, ecc
- Inglobare il Controllore e il Datapath in un unico file blif come se fossero due componenti, collegando opportunamente i segnali di comunicazione. E' importante specificare con la direttiva *.search* il file del Controllore ottenuto dopo la codifica degli stati e non quello scritto a mano.

10.11 Progettare FSMD in Verilog

Si divide un file in 3 processi:

- Blocco sequenziale sincrono che emula il comportamento dei registri del modello di Huffman
- Blocco sequenziale asincrono, sensibile ai cambiamenti di stato, dei segnali di ingresso della FSM, che implementa la macchina a stati. Il processo è solitamente implementato con costrutti *case* ed if

- Blocco sequenziale sincrono che implementa le operazioni nel datapath. Il flusso di controllo del blocco sequenziale è solitamente regolato dai valori dei segnali di controllo generati dalla FSM (ed eventualmente, dai segnali d'ingresso nel datapath)

Esempio 10.15

```
module SemaforoTemporizzato(
  input clk, trafficons, trafficoeo,
  output reg lucens, luceeo);
  reg [1:0] stato = 2'b00;
  reg [1:0] stato_prossimo = 2'b00;
  reg inizio = 1'b0, fine = 1'b0;
  reg [2:0] registro = 3'b000;

  // Blocchi sincroni
  always @(clk) begin : UPDATE
    stato = stato_prossimo;
  end

  always @(posedge clk) begin : DATAPATH
    if(inizio) begin
      registro = 3'b000;
      fine = 1'b0;
    end else begin
      if(registro < 3'b111) begin
        registro = registro + 1'b1;
        fine = 1'b0;
      end else begin
      fine = 1'b1;
      end
    end
  end

  // Blocco asincrono
  always @(stato, trafficons, trafficoeo, fine)
  begin : FSM
    case(stato)
    2'b00:
      if(~trafficons && trafficoeo) begin
        lucens = 1'b1; luceeo = 1'b0;
        inizio = 1'b1;
        stato_prossimo = 2'b10;
      end else begin
        lucens = 1'b1; luceeo = 1'b0;
        stato_prossimo = 2'b00;
      end
    2'b11:
      begin inizio = 1'b0;
      if(fine) begin
        lucens = 1'b1; luceeo = 1'b0;
        stato_prossimo = 2'b00;
      end else begin
        lucens = 1'b0; luceeo = 1'b1;
        stato_prossimo = 2'b11;
      end end
    endcase
  end
endmodule
```

10.12 Testbench per Verilog e SIS

- Il modello Verilog (più astratto) deve guidare lo sviluppo della sua implementazione di SIS.
- Il comportamento del modello Verilog (più semplice da verificare), può essere utilizzato per verificare il comportamento del modello SIS
- A parità di ingressi, i modelli Verilog e SIS devono produrre la stessa sequenza di uscite

Dunque, si modifica il testbench Verilog per produrre i file necessari alla verifica del modello SIS. Per creare file in Verilog:

- Dichiarazione di una variabile intera fd che identifichi il file (i.e., file descriptor)
- $fd = \$fopen(\text{"}nome\ file\text{"}, \text{"}w\text{"})$: apertura del file in scrittura
- $\$fclose(fd)$: chiusura del file
- $\$fdisplay(fd, \text{"}stringa\ da\ stampare\text{"})$: funzione per la stampa

Esempio 10.16
Un esempio di file testbench che generi anche un testbench per SIS è il seguente.
Si creano due file: uno script testbench.sis che farà si che SIS esegua il test; un file output$_v$erilog.txt che salva i valori di output ad ogni ciclo di clock.

```
module tb_semaforo();
// File descriptors.
integer tbf, outf;
...

initial begin
...
  tbf = $fopen("testbench.script", "w");
  outf = $fopen("output_verilog.txt", "w");

  // funzione $fdisplay per stampare sul file tbf
  $fdisplay(tbf,"read_blif semaforo_temporizzato.blif");

  // Gli ingressi vengono scritti nel file testbench.script,
  // mentre le uscite vanno salvate nel
  // file output_verilog.txt
  clk = 1'b0;
  trafficons = 1'b0;
  trafficoeo = 1'b0;
  $fdisplay(tbf, "simulate %b %b", trafficons, trafficoeo);
  #20
  $fdisplay(outf, "Outputs: %b %b", lucens, luceeo);
  trafficons = 1'b0;
  trafficoeo = 1'b1;
  $fdisplay(tbf, "simulate %b %b", trafficons, trafficoeo);
  #20
  $fdisplay(outf, "Outputs: %b %b", lucens, luceeo);
  ...
  #20
  $fdisplay(outf, "Outputs: %b %b", lucens, luceeo);
  $fdisplay(tbf, "quit");
  // E'fondamentale chiudere i file con $fclose.
  $fclose(tbf);
  $fclose(outf);
  $finish;
end
endmodule
```

10.12.1 Utilizzo del file in SIS

Utilizzare il file testbench.script per stimolare l'esecuzione del sistema in Sis, filtrando le uscite (filtrate) nel file *output$_s$is.txt*:

```
bash> sis -f testbench.script -x | grep Outputs: > output_sis.txt
```

Verificare le differenze tra l'output del modello verilog e l'output del modello sis:

```
bash> diff output_sis.txt output_verilog.sis
```

Se il comando non stampa nulla, i due output sono equivalenti.

11 Hardware design su FPGA con HDL

HDL è un linguaggio di modellazione che permette di progettare circuiti su hardware.

11.1 EDA (Electronic Design Automation)

La legge di Moore dice che ogni due anni il numero di transistor nei circuiti raddoppia.

Con gli anni aumenta anche l'astrazione, che però comporta anche perdita di dettagli.

Con verilog simuliamo il circuito (test bench) creando così un golden model, che sarà il modello di riferimento per realizzare il circuito con SIS. Nel caso in cui il modello fatto con sis si comporta come il bengh test, allora esso è corretto soltanto rispetto a quel bench test.

Si potrebbe scrivere un modello usando un linguaggio di programmazione comune, il problema è che i linguaggi di programmazinoe sono sequenziali e quindi eseguono le istruzioni in sequenza. Nei circuiti ogni porta logica continua a lavorare sui segnali che ha in ingresso, quindi non è sequenziale e per rappresentare l'hardware serve un linguaggio parallelo (i linguaggi di programmazione non sono fatti per questo). Per questo motivo c'è una netta differenza tra il tempo di simulazione e il tempo simulato (tempo reale). Un HDL che useremo è Verilog in cui tutto concorre (quindi in parallelo) e non ci sono istruzioni sequenziali.

12 Circuiti sequenziali

È un circuito i cui valori di uscita non dipendono soltanto dai valori di ingresso correnti, ma anche da quelli precedenti. Questa la differenza dispetto a una rete combinatoria, che invece non tiene traccia della sequenza passata di input.

$$O_t = f(I_1, I_2, ..., I_k)$$

Definizioni utili 12.1
I circuiti combinatori sono un sottoinsieme dei circuiti sequenziali, per la precisione sono circuiti sequenziali con solo 1 stato.

Il sistema digitale **memorizza** gli ingressi fino a quel momento, cioè memorizza lo **stato** del sistema. Un esempio di circuito sequenziale è il seguente:

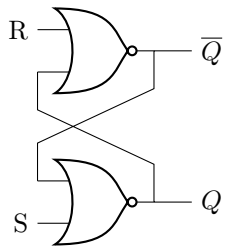


Figura 32: Latch SR

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

Tabella 37: Tabella di verità del Latch SR

Se S e R valgono 0, allora il circuito mantiene lo stato precedente.

C'è bisogno di un "metronomo" che faccia scorrere il tempo, per i circuiti si utilizza un **clock** che è un segnale periodico che oscilla tra 0 e 1 e scandisce il tempo. Deve essere presente in tutti i sistemi digitali sequenziali.

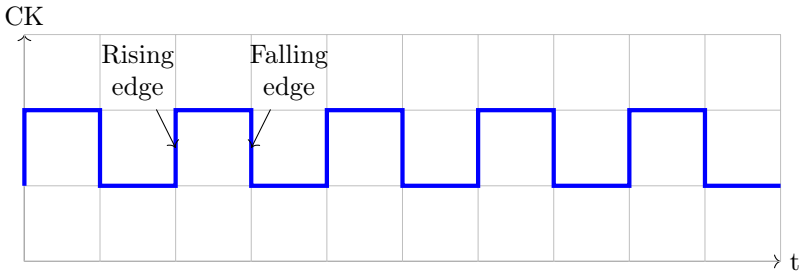


Figura 33: Clock

Di seguito c'è il circuito di un D-Latch che è un latch con un clock.

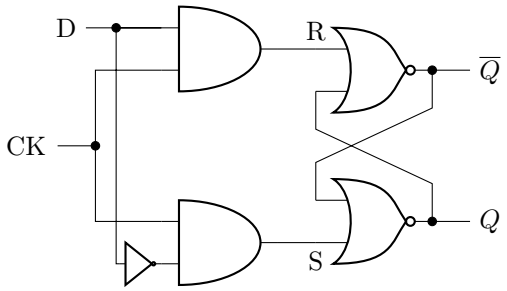


Figura 34: D Latch

D CK	Q_{t+1}
0 1	0
1 1	1
0 0	Q_t
1 0	Q_t

Si può scrivere anche:

D CK	Q_{t+1}
D 1	D
- 0	Q_t

Cioè, quando il clock è 1 il valore di Q è uguale a quello di D, mentre quando il clock è 0 il valore di Q è uguale a quello di Q_t.

Il grafico dei segnali è il seguente:

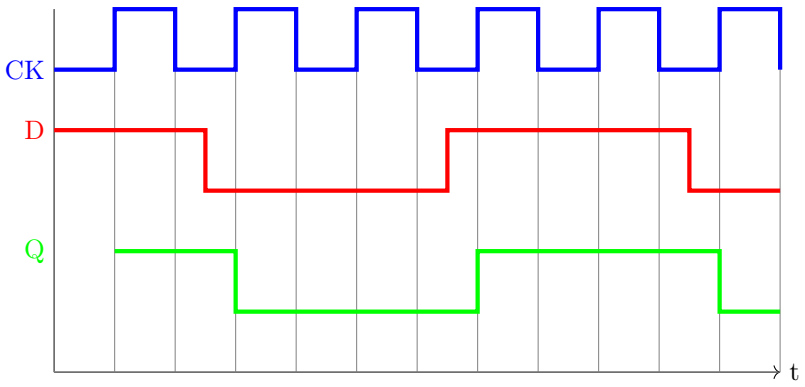


Figura 35: Grafico dei segnali del D Latch

Il circuito sincronizza il segnale di ingresso con quello del clock.

12.1 Astrazione

Per rendere più facile la realizzazione dei circuiti si astraggono le porte in componenti più complessi.

Alcuni esempi sono i seguenti:

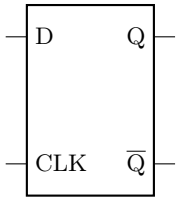


Figura 36: Simbolo del latch

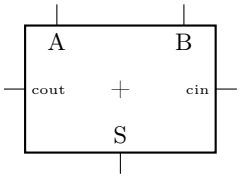


Figura 37: Simbolo dell'adder

Unendo 8 sommatori si ottiene un sommatore a 8 bit.
Oppure unendo N Latch si ottiene un registro a N bit:

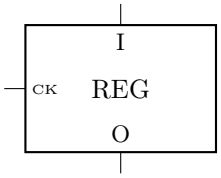


Figura 38: Simbolo del registro

Un altro esempio è la seguente tabella:

A B S	O
0 - 0	0
1 - 0	1
- 0 1	0
- 1 1	1

che assegna all'uscita il valore di A se S è 0, altrimenti assegna il valore di B. Questo circuito si chiama **multiplexer** (MUX) e si rappresenta col seguente circuito:

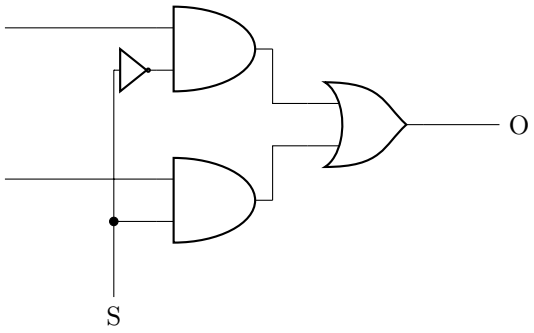


Figura 39: Circuito del multiplexer

Il simbolo del multiplexer è il seguente:

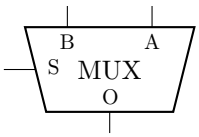


Figura 40: Simbolo del multiplexer

Esempio 12.1
Realizziamo con i precedenti componenti un contatore a modulo 1024.

Lo chiamiamo "count" e quando il reset $= 1$ *allora count* $= 0$*. Per contare 1024 numeri bisogna memorizzare 10 bit, quindi prendiamo un registro a 10 bit.*

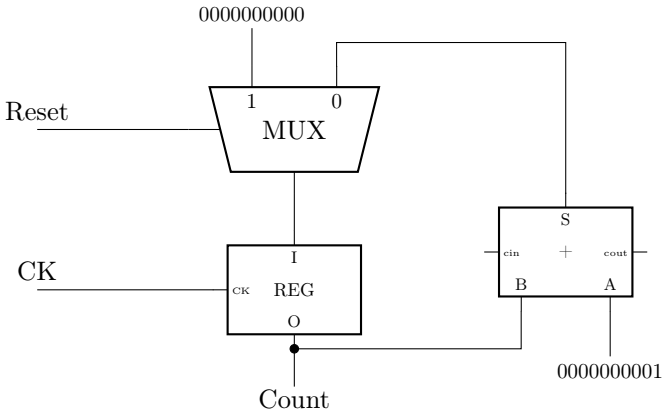


Figura 41: Contatore a modulo 1024

Il segnale del circuito è il seguente:

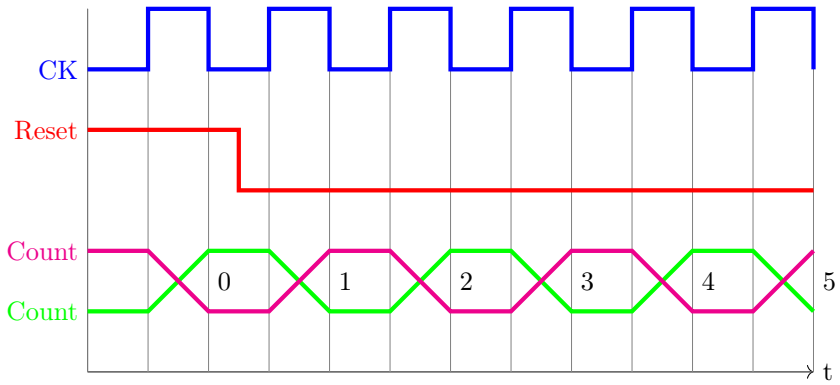


Figura 42: Grafico dei segnali del contatore

Non si può sapere l'uscita del circuito perchè ha un ritardo (dato dal cammino critico) che non permette al segnali di stabilizzarsi in tempo. Per questo motivo questo circuito non rappresenta un contatore, ma essenzialmente è un generatore casuale di numeri. Bisogna quindi creare un altro elemento di memoria.

Un altro elemento di memoria è il flip-flop D (oppure Latch Master-Slave):

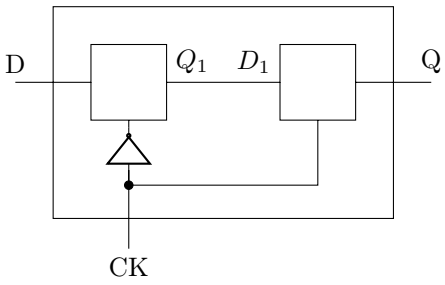


Figura 43: Flip-flop D

Il campionamento degli ingressi avviene sul fronte di salita del clock.

13 Macchine a stati finiti (FSM)

Per descrivere un qualsiasi comportamento sequenziale si usano come **modello** le FSM (Final State Machines). Tutte le macchine sensate avranno delle sequenze finite e conterranno l'evoluzione da stato presente a stato prossimo in base alla configurazione di ingresso e genereranno una configurazione di uscita. Esistono diversi tipi di SFM, ma in questo corso ci concentriamo solo sulle macchine **sincrone** e **deterministiche**.

- **Sincrone**: evolvono a ogni ciclo di clock (temporizzate)

- **Deterministiche**: dato un ingresso si ottiene sempre la stessa uscita

Una macchina a stati è una sestupla:

$$M =< S, I, O, \delta, \lambda, s >$$

- **S**: insieme, finito e non vuoto, degli stati
- **I**: alfabeto in ingresso, n bit in ingresso. $|I| = 2^n$
- **O**: alfabeto di uscita, m bit in uscita $|O| = 2^m$
- δ: funzione di stato prossimo, $\delta : S \times I \to S$
- λ: funzione di uscita, $\lambda : S \times I \to O$. Esistono più tipi di macchine in base alla funzione λ
  - **FSM di Mealy**: Genera l'uscita in base allo stato corrente e l'input corrente
    $$\lambda : S \times I \to O$$
  - **FMS di Moore**: Genera l'uscita in base allo stato corrente
    $$\lambda : S \to O$$

  I due tipi di macchine dal punto di vista dell'espressività sono equivalenti, quindi si può passare da una all'altra.
- **s**: stato iniziale (può non esserci), $s \in S$

13.1 Rappresentazione delle FSM

13.1.1 State Transition Table (STT)

Per ogni coppia S, I indica lo stato prossimo e l'uscita.

- Nelle **colonne**: simboli di ingresso
- Nelle **righe**: stato corrente
- Nelle **celle**:
  - **Mealy**: stato prossimo e uscita
  - **Moore**: stato prossimo

Esempio 13.1 (Mealy)

$$M =< \{A, B, C\}, \{0, 1\}, \{0, 1\}, \delta, \lambda >$$

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

Tabella 38: State Transition Table di una FSM di Mealy

Esempio 13.2 (Moore)

$$M =< \{A, B, C\}, \{0, 1\}, \{0, 1\}, \delta, \lambda >$$

La funzione lambda viene scritta in una colonna separata: z

	0	1	z
A	B	C	0
B	A	C	1
C	B	A	1

Tabella 39: State Transition Table di una FSM di Moore

13.1.2 State Transition graph (STG)

Un grafo è un costrutto matematico preciso formato da una coppia $G =< V, E >$ dove:

- V è un insieme di nodi (vertex)
- E è un insieme di archi orientati (edges)

Un esempio di grafo è il seguente:

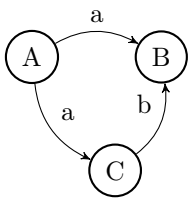


Figura 44: Esempio di STG

Esempio 13.3 (Mealy)

$$M =< \{A, B, C\}, \{0, 1\}, \{0, 1\}, \delta, \lambda >$$

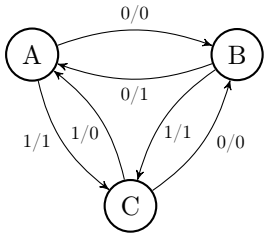


Figura 45: Esempio di STG con Mealy

Esempio 13.4 (Moore)

$$M =< \{A, B, C\}, \{0, 1\}, \{0, 1\}, \delta, \lambda >$$

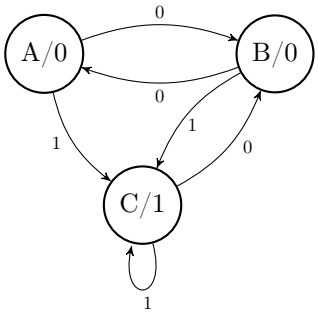


Figura 46: Esempio di STG con Moore

Esercizio 13.1
Progettare una macchina con 1 bit di ingresso x e 1 bit di uscita y. $y = 1$ se su x è stato letto un numero pari di 0, seguiti da un numero dispari di 1. y torna a 0 quando il numero di 1 diventa pari, oppure arriva uno 0.

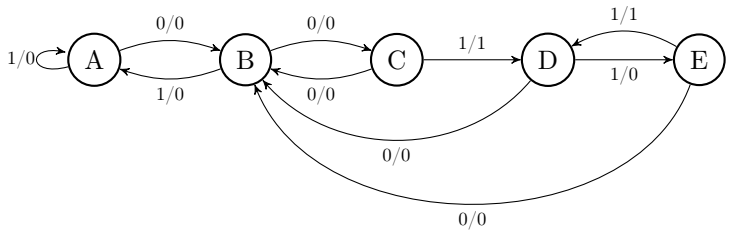


Figura 47: Esempio di STG con Moore

- ***A**: aspetto il primo 0 della sequenza*
- ***B**: ho letto una sequenza dispari di 0*
- ***C**: ho letto una sequenza di pari di 0*
- ***D**: ho letto una sequenza pari di 0 e una sequenza dispari di 1*
- ***E**: ho letto una sequenza pari di 0 e una sequenza pari di 1*

	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

13.2 Modello di Huffman

I circuiti sequenziali si realizzano fisicamente con il **modello di Huffman**. Questo modello divide il circuito in 2 parti:

- **Parte combinatoria**
- **Parte di registri**

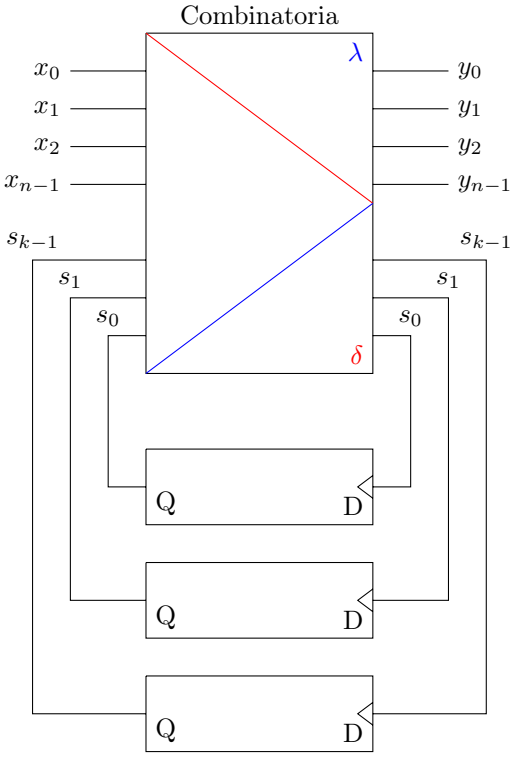


Figura 48: Modello di Huffman

13.3 Codifica degli stati

Prendiamo in considerazione la macchina a stati della figura 47.

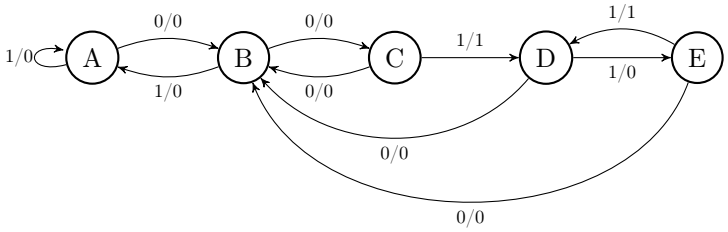


Figura 49: Esempio di STG con Moore

	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

I registri dovranno contenere la **codifica degli stati**. Come prima cosa bisogna capire quanti bit servono per rappresentare gli stati.

$$\sigma = |S|$$

Mi servono k bit dove:

$$k = log_2(\sigma) = log_2(5) = 3$$

Le seguenti **variabili di stato** si possono codificare con 3 bit:

- **A** = 000
- **B** = 001
- **C** = 010
- **D** = 011
- **E** = 100

Un metodo più "furbo" per codificare le variabili è assegnare ad ogni stato un codice che contenga soltanto un bit a 1. Questo metodo è chiamato **codifica one-hot**.

- **A** = 00001
- **B** = 00010
- **C** = 00100
- **D** = 01000
- **E** = 10000

Ora si mette insisme la parte combinatoria con la parte sequenziale. Si riscrive la tabella degli stati utilizzando la codifica a 3 bit:

$y_2\ y_1\ y_0$	$x = 0$	$x = 1$
0 0 0	001/0	000/0
0 0 1	010/0	000/0
0 1 0	001/0	011/1
0 1 1	001/0	100/0
1 0 0	001/0	011/1

La funzione δ è la seguente

$$\delta(y_2', y_1', y_0') = \delta(y_2 \wedge y_1 \wedge y_0 \wedge x)$$

E la tabella di verità della funzione è la seguente:

y_2 y_1 y_0 x	y_2' y_1' y_0'
0 0 0 0	0 0 1
0 0 0 1	0 0 0
0 0 1 0	0 1 0
0 0 1 1	0 0 0
0 1 0 0	0 0 1
0 1 0 1	0 1 1
0 1 1 0	0 0 1
0 1 1 1	1 0 0
1 0 0 0	0 0 1
1 0 0 1	0 1 1
1 0 1 0	- - -
1 0 1 1	- - -
1 1 0 0	- - -
1 1 0 1	- - -
1 1 1 0	- - -
1 1 1 1	- - -

La funzione λ è la seguente:

$$z = \lambda(y_2, y_1, y_0, x)$$

E la tabella di verità della funzione è la seguente:

y_2 y_1 y_0 x	z
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	0
0 1 0 0	0
0 1 0 1	1
0 1 1 0	0
0 1 1 1	0
1 0 0 0	0
1 0 0 1	1
1 0 1 0	-
1 0 1 1	-
1 1 0 0	-
1 1 0 1	-
1 1 1 0	-
1 1 1 1	-

Ora si può disegnare il modello di Huffman della funzione

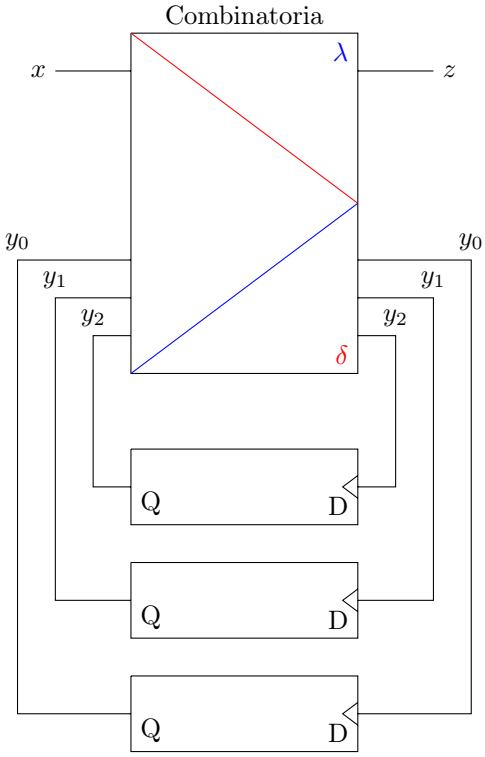


Figura 50: Modello di Huffman

14 Equivalenza tra macchina a stati

Le FSM si dividono in 2 tipi:

- **Completamente specificate**: per ogni coppia stato/ingresso è specificata la coppia stato prossimo/uscita
- **Parzialmente specificate**: per alcune coppie stato/ingresso non viene specificato lo stato prossimo **oppure** (anche entrambi) l'uscita

Prendiamo in considerazione 2 macchine:

$$M_1 =< S_1, I_1, O_1, \delta_1, \lambda_1 >$$

$$M_2 =< S_2, I_2, O_2, \delta_2, \lambda_2 >$$

Tali che $I_1 = I_2$ e $O_1 = O_2$, M_1 e M_2 sono equivalenti se e solo se per ogni stato $s_1^a \in S_1$ esiste uno stato $s_2^b \in S_2$ tale che ponendo la macchina M_1 nello stato s_1^a e la macchina M_2 nello stato s_2^b e applicando una qualsiasi sequenza di ingresso J_a le due sequenze di uscita prodotte sono identiche.

Definizione 14.1

$$M =< S, I, O, \delta, \lambda > \quad \textit{completamente specificata}$$

*$s_i, s_k \in S$ sono **equivalenti** (o indistiguibili) $s_i \sim s_k$ se e solo se per ogni sequenza di ingresso:*

$$I_\alpha \dots i_n$$

$$\lambda(s_i, I_\alpha) = \lambda(s_k, I_\alpha)$$

Non si può trovare un algoritmo che verifichi l'uguaglianza perchè ci sono infiniti test da verificare.

La relazione $\sim$ è una relazione di equivalenza ed è:

- **Riflessiva**: $s_i \sim s_i$
- **Simmetrica**: $s_i \sim s_k \rightarrow s_k \sim s_i$
- **Transitiva**: $s_i \sim s_k \wedge s_k \sim s_j \rightarrow s_i \sim s_j$

14.1 Macchina minima

Una macchina M si dice **minima** se in S non esiste una coppia di stati $s_i, s_k \in S$ tale che $s_i \sim s_k$. **Per ogni macchina esiste una e una sola macchina minima** (a meno di rinominazione degli stati).

15 Algoritmo di Paull-Unger

Bisogna cambiare la definizione di equivalenza per rimuovere l'infinità di test.

Definizione 15.1
*$s_i \sim s_k$ se e solo se per ogni $i \in I$ si verificano **entrambe** le seguenti condizioni:*

1. $\lambda(s_i, i) = \lambda(s_j, i)$
2. $\delta(s_i, i) \sim \delta(s_j, i)$ *(definito ricorsivamente)*

s	x=0	x=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

Adesso si crea una tabella triangolare in cui si fanno i confronti tra gli stati.

B	(B,C)			
C	×	×		
D	(A,E)	(B,C) (A,E)	×	
E	×	×	∼	×
	A	B	C	D

Tabella 40: Tabella di confronto

Nel passo successivo si controllano le coppie di stati con i confronti già fatti, se valgono rimangono nella tabella, altrimenti si eliminano.

B	×			
C	×	×		
D	×	×	×	
E	×	×	∼	×
	A	B	C	D

Tabella 41: Tabella di confronto finale

Nel passo successivo si creano nuovi stati a partire dalla tabella.

$$\alpha = \{A\}, \;\; \beta = \{B\}$$

$$\gamma = \{C, E\}, \;\; \delta = \{D\}$$

Si ricrea poi la tabella con i nuovi stati e sarà la macchina minima.

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

15.0.1 Esempi

Esempio 15.1
Prendiamo in considerazione la seguente tabella:

S	0	1
A	G/00	C/01
B	G/00	D/01
C	D/10	A/11
D	C/10	B/11
E	G/00	F/01
F	F/10	E/11
G	A/01	F/11

Creiamo la tabella dei confronti:

B	(C,D)					
C	×	×				
D	×	×	(A,B)			
E	(C,F)	(D,F)	×	×		
F	×	×	(D,F) (A,E)	(C,F) (B,E)	×	
G	×	×	×	×	×	×
	A	B	C	D	E	F

Tabella 42: Tabella di confronto finale

Osserviamo che il confronto (B,A) vale solo se vale il confronto (C,D) che anch'esso vale solo se vale (B,A), cioè:

$$A \sim B \ \ se \ \ C \sim D \qquad C \sim D \rightarrow A \sim B$$

$$C \sim D \ \ se \ \ A \sim B \qquad A \sim B \rightarrow C \sim D$$

*Utilizziamo quindi dei grafi (**cricca**[a]) per risolvere il problema visto che non abbiamo cicli che validano tutti gli stati.*

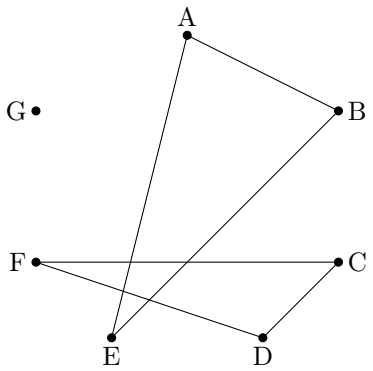


Figura 51: Grafo delle cricche

Dal grafo deriviamo le cricche:

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

$$\beta = \{C, D, F\}$$

$$\gamma = \{G\}$$

*Questi sottoinsiemi sono chiamati **Classi di equivalenza**.*
Nel prossimo passo si crea la tabella delle classi di equivalenza:

	0	1
α	$\gamma/00$	$\beta/01$
β	$\beta/10$	$\alpha/11$
γ	$\alpha/01$	$\beta/11$

[a]Un insieme di nodi connessi tra loro. Una **Cricca massima** è una cricca che non è contenuta in un altra cricca

15.1 Compatibilità

$s_i \vee s_k$ se e solo se per ogni $i_i \in I$

- ovunque siano ambedue specificati $\delta(s_i, i_i) \vee \delta(s_k, i_i)$
- ovunque siano ambedue specificati $\lambda(s_i, i_i) = \lambda(s_k, i_i)$

Dopo aver utilizzato l'algoritmo di Paull-Unger si può creare la **tabella di compatibilità**

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

Tabella 43: Esempio di macchina a stati

La tabella dei confronti è la seguente:

B	(E,D)			
C	$\sim$	(D,E)		
D	$\times$	$\times$	(E,A) (C,A)	
E	(A,B)	(D,A)	(E,A) (C,B)	(A,B)
	A	B	C	D

Tabella 44: Tabella di confronto

La tabella finale è la seguente:

B	(E,D)			
C	$\sim$	(D,E)		
D	$\times$	$\times$	(E,A) (C,A)	
E	(A,B)	$\times$	(E,A) (C,B)	(A,B)
	A	B	C	D

Tabella 45: Tabella di confronto finale

Bisogna fare il grafo:

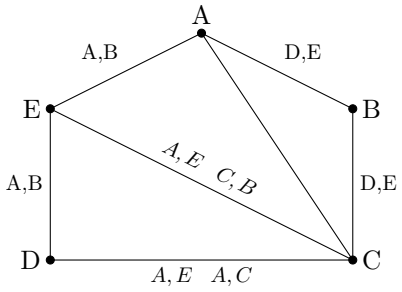


Figura 52: Grafo delle cricche

Infine bisogna trovare le classi massime di compatibilità (**cricche massime**), ma possono esistere solo se esistono le condizioni definite sugli archi.

$$\alpha = \{A, B, C\} \text{ condizione D,E}$$

$$\beta = \{A, C, E\} \text{ condizione } A, B \;\; A, E \;\; C, B$$

$$\gamma = \{C, D, E\} \text{ condizione } A, E \;\; A, B \;\; A, C \;\; C, B$$

Bisogna prenderne una (a caso), assumerla compatibile e ricavare le altre. In questo caso scegliamo α e vengono cancellati tutti i collegamenti del grafo che coinvolgono α.

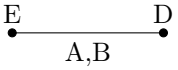


Figura 53: Grafo delle cricche

Dall'ultimo grafo otteniamo di nuovo le cricche massime:

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

Gli stati alla fine saranno 2, cioè quello che abbiamo scelto all'inizio (α) e la cricca massima dell'ultimo grafo:

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

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

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

16 Componenti del datapath

Il **Datapath** è un progetto di un'unità di elaborazione mediante l'aggregazione di componenti elementari. Esistono 2 tipi di componenti:

- **Memorizzazione** (registri):
- **Elaborazione** (unità funzionali):

16.1 Registri

16.1.1 Registro parallelo/parallelo

Consente di memorizzare un valore e di restituirlo in uscita.

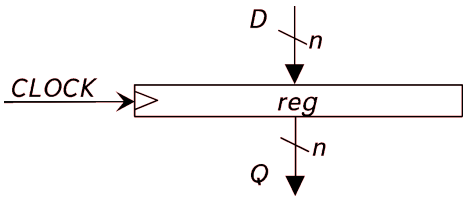


Figura 54: Registro parallelo/parallelo

```
module RegistroParalleloParallelo #( parameter N = 8)(
  input [N-1:0] D,
  input clock,
  output [N-1:0] Q);
  reg [N-1:0] dato = 8'b00000000;
  assign Q = dato;
  always @(posedge clock) begin
    dato = D;
  end
endmodule
```

16.1.2 Registro seriale/seriale

Memorizza i valori come sliding window e restituisce in output il valore più vecchio.

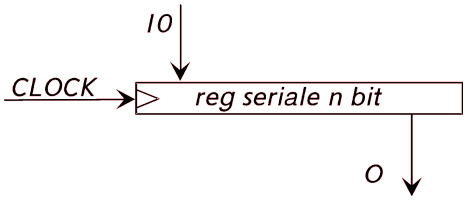


Figura 55: Registro seriale/seriale

```
module RegistroSerialeSeriale (input IO, input clock,
  output O);
  parameter N = 8; reg [N-1:0] dato = 8'b00000000;
  assign O = dato[0];
  always @(posedge clock) begin
  dato = {IO, dato[N-1:1]};
    end
  endmodule
```

16.1.3 Registro parallelo/seriale

È un misto tra i 2 precedenti, ma con un valore che decide in che modo si comporta.

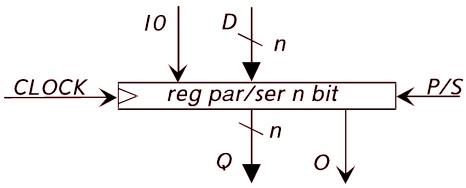


Figura 56: Registro parallelo/seriale

```
module RegistroParalleloSeriale #(parameter N=8)(
  input PS, input IO, input [N-1:0] D, input clock,
  output [N-1:0] Q, output O);
  reg [N-1:0] dato = 8'b00000000;
  assign Q = dato;
  assign O = dato[0];
  always @(posedge clock) begin
    if(PS) dato = D;
    else dato = {IO, dato[N-1:1]};
  end
endmodule
```

16.2 Unità funzionali

16.2.1 Multiplexer

Arrivano vari segnali e decido quale tenere.

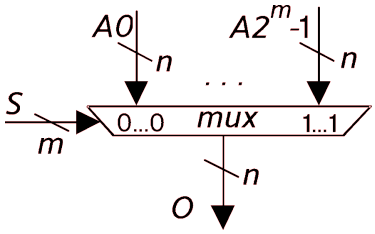


Figura 57: Multiplexer

```
module Multiplexer #(parameter N=8)(
  input [1:0] S, input [N-1:0] A3, input [N-1:0] A2,
  input [N-1:0] A1, input [N-1:0] A0,
  output reg [N-1:0] O);
  always @(A3, A2, A1, A0, S) begin
    case(S)
      2'b00: O = A0;
      2'b01: O = A1;
      2'b10: O = A2;
      2'b11: O = A3;
      default: O = 8'b00000000;
    endcase
  end
endmodule
```

16.2.2 Demultiplexer

Arriva un segnale e decido dove mandarlo.

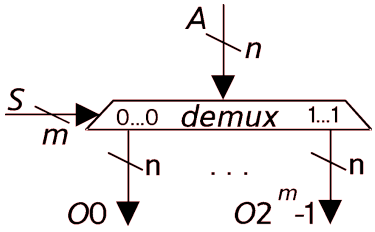


Figura 58: Demultiplexer

```
module Demultiplexer #(parameter N = 8) (
  input [1:0] S, input [N-1:0] A,
  output reg [N-1:0] O3, output reg [N-1:0] O2,
  output reg [N-1:0] O1, output reg [N-1:0] O0);

  always @(S,A) begin
    case(S)
      2'b00: begin
         O0 = A; O1 = 0; O2 = 0; O3 = 0;
       end
      2'b01: begin
         O0 = 0; O1 = A; O2 = 0; O3 = 0;
       end
      2'b10: begin
         O0 = 0; O1 = 0; O2 = A; O3 = 0;
       end
      2'b11: begin
         O0 = 0; O1 = 0; O2 = 0; O3 = A;
       end
    endcase
  end
endmodule
```

16.2.3 Decoder

In uscita si ha un segnale che indica quale ingresso è attivo.

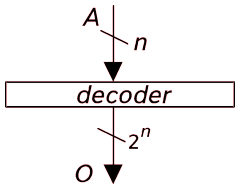


Figura 59: Decoder

```
module Decoder #(parameter N = 8)(
  input [N-1:0] A, output reg [(2**N)-1:0] O);
  integer i;
  always @(A)
  begin
    for(i = 0; i < (2**N); i = i + 1) begin
      if(i == A) O[i] = 1'b1;
      else O[i] = 1'b0;
    end
  end
endmodule
```

16.2.4 Shifter

Si inserisce un valore e si decide di quanto si vuole shiftare.

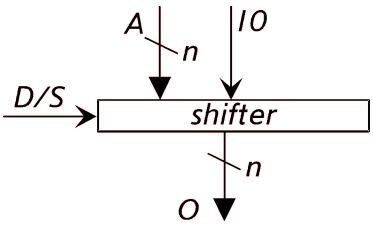


Figura 60: Shifter

```
module Shifter #(parameter N = 8)
  (input DS, input [N-1:0] A, input I0,
  output reg [N-1:0] O);
  always @(A, I0) begin
  if(DS) O = {I0, A[N-1:1]};
  else O = {A[N-2:0], I0};
  end
endmodule
```

16.3 Unità aritmetiche

16.3.1 Sommatore

Fa la somma tra 2 numeri in complemento a 2.

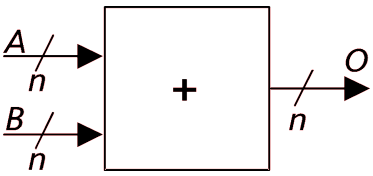


Figura 61: Sommatore

```
module Sommatore #(parameter N = 8)(
  input [N-1:0] A, input [N-1:0] B,
  output [N-1:0] O);
  assign O = A + B;
endmodule
```

16.3.2 Moltiplicatore

Moltiplica 2 numeri in complemento a 2.

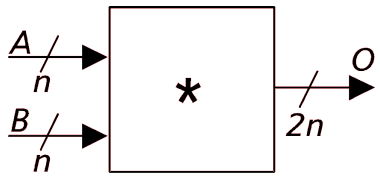


Figura 62: Moltiplicatore

```
module Moltiplicatore #(parameter N = 8)(
  input [N-1:0] A, input [N-1:0] B,
  output [(N**2)-1:0] O);
  assign O = A * B;
endmodule
```

16.4 Unità logiche

16.4.1 And

Fa l'and tra 2 numeri nella stessa posizione, ad esempio: $A_1 \wedge B_1 \ldots A_k \wedge B_k$.

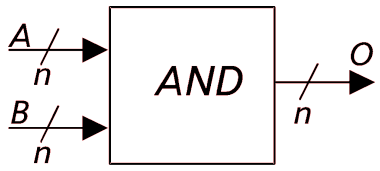


Figura 63: And

```
module And #(parameter N = 8)(
  input [N-1:0] A, input [N-1:0] B,
  output reg [N-1:0] O);
    integer i;
    always @(A, B) begin
      for(i = 0; i < N; i = i + 1) begin
        O[i] = A[i] & B[i];
      end
    end
endmodule
```

16.4.2 Not

Inverte i valori di un numero.

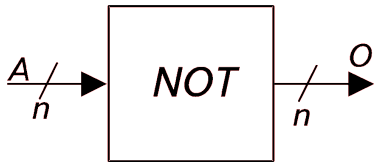


Figura 64: Not

```
module Not #(parameter N = 8)(
  input [N-1:0] A
  output reg [N-1:0] O);
    integer i;
    always @(A) begin
      for(i = 0; i < N; i = i + 1) begin
        O[i] = ~A[i];
      end
    end
endmodule
```

16.4.3 Altri operatori logici

- Or
- Nand
- Nor
- Xor
- XNor

16.5 Operatori di controllo

16.5.1 Maggiore

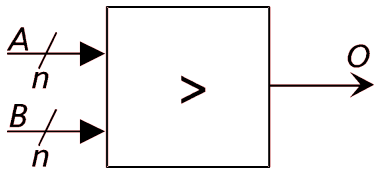


Figura 65: Maggiore

```
module Maggiore #(parameter N = 8)(
 input [N-1:0] A, input [N-1:0] B,
 output reg O);

   always @(A, B) begin
     if( A > B ) O = 1'b1;
     else O = 1'b0;
   end
endmodule
```

16.6 Esempio

Si vuole costruire un ALU da 8 bit (in complemento a 2) con accumulatore (registro che salva l'ultimo risultato calcolato) che esegue 4 operazioni:

- Somma
- Sottrazione
- Maggiore/Minore

Manca il componente del sottrattore, ma si può creare facilmente facendo la somma di un numero negativo. Si implementa quindi un componente che inverte un numero:

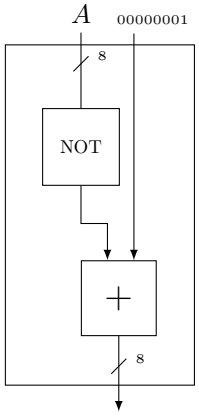
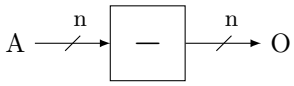
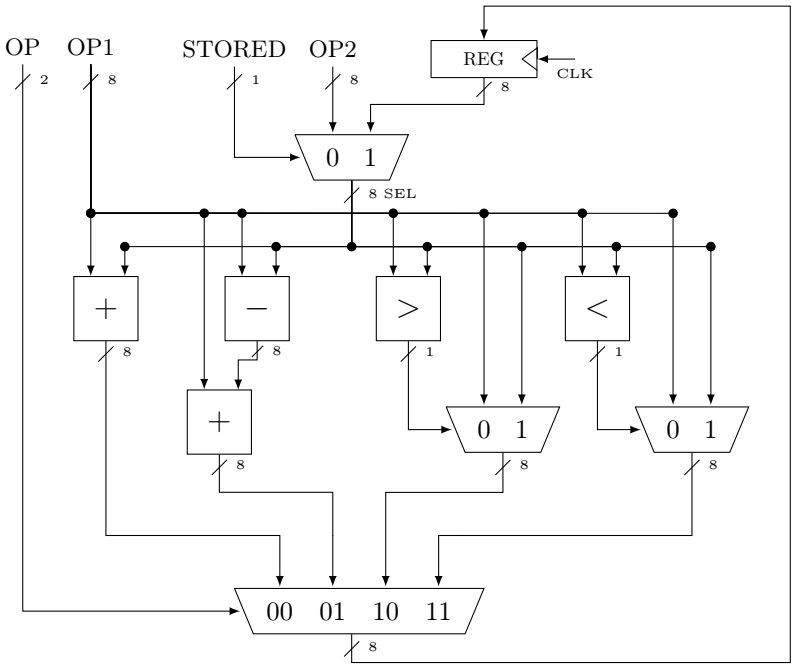


Figura 66: Circuito invertitore

Questo circuito diventerà quindi un nuovo componente:



Un segnale di input "STORED" mi dice se usare "OP2" o un accumulatore



Il codice del circuito è il seguente:

```
module Opposto #(parameter N = 8)(
  input [N-1:0] operando,
  output [N-1:0] risultato);
    assign risultato = -operando;
endmodule
```

```
module ALU #(parameter N = 8)(
  input clock,
  input [N-1:0] op1,
  input [N-1:0] op2,
  input stored,
  input [1:0] oper,
  output [N-1:0] O);

    wire [N-1:0] acc, sel, t1, t2, t3, t4, t5, out;
    wire s1, s2;

    RegistroParalleloParallelo registro(out, clock, acc);
    Multiplexer2 mux1(stored, acc, op2, sel);
    Multiplexer2 mux2(s1, op1, sel, t3);
    Multiplexer2 mux3(s2, op1, sel, t4);
    Multiplexer mux4(oper, t4, t3, t2, t1, out);
    Sommatore sum1(op1, sel, t1);
    Opposto opp1(sel, t5);
    Sommatore sum2(op1, t5, t2);
    Maggiore mag(op1, sel, s1);
    Minore min(op1, sel, s2);

    assign O = out;
endmodule
```

Di seguito il codice implementato con il metodo Behavioural:

```
module ALU_behav #(parameter N = 8)(
  input clock,
  input [N-1:0] op1,
  input [N-1:0] op2,
  input stored,
  input [1:0] oper,
  output [N-1:0] O);

    reg [N-1:0] acc, sel, out;

    always @(stored, acc, op2) begin
      if(stored) sel = acc;
      else sel = op2;
    end

    always @(posedge clock) begin
      if(clock) acc = out;
    end

    assign O = out;

    always @(op1, sel, oper) begin
      case(oper)
      2'b00 :
        out = op1 + sel;
      2'b01 :
        out = op1 - sel;
      2'b10 :
        if(op1 > sel) out = op1;
        else out = sel;
      2'b11 :
        if(op1 < sel) out = op1;
        else out = sel;
      endcase
    end
endmodule
```

17 Modello FSMD (FSM con Datapath o Controllore/Datapath)

È composto da 2 parti:

- **Controllo**: Finite State Machine
- **Elaborazione**: Datapath

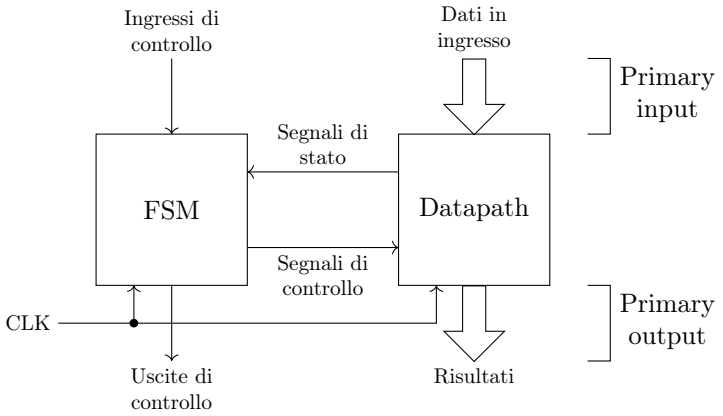


Figura 67: Modello FSMD

Per progettare il modello precedente servono diversi passaggi:

1. Definire l'insieme di operazioni necessarie (identificare le unità funzionali necessarie)
2. Identificare le operazioni che le unità funzionali devono svolgere
3. Identificare le necessità di memorizzare i dati (ad esempio i risultati intermedi)
4. Progetto il datapath di ogni operazione
5. Identificare i segnali di controllo
6. Progettare il controllore (FSM) che genera i segnali di controllo

Questi passi non danno sempre una decisione univoca, infatti si possono prendere più decisioni che permettono di bilanciare il modello per avere una parte preponderante di macchina a stati, oppure una parte preponderante di datapath.

17.1 Esempio

Prendiamo in considerazione l'esempio del semaforo:

Per evitare incidenti, il circuito di controllo deve garantire che le luci sulle strade NS e EO siano sempre accese in opposizione. Il circuito assegna priorità alla strada NS e commuta dal verde al rosso su NS solo se TRAFFICONS=0 e TRAFFICOEO=1, in caso contrario mantiene il verde su NS. In assenza di traffico sia su NS che su EO il semaforo non modifica la configurazione raggiunta. Non appena giunge traffico su NS, indipendentemente da cosa succede su EO, il semaforo assegna la luce verde a NS e la luce rossa a EO.

Gli input sono:

- **TRAFFICONS[1]**: 1 indica traffico su nord-sud
- **TRAFFICOEO[1]**: 1 indica traffico su est-ovest

Le uscite sono:

- **LUCENS[1]**: 1 indica luce verde su nord-sud
- **LUCENEO[1]**: 1 indica luce verde su est-ovest

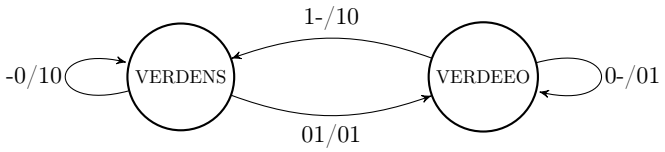


Figura 68: FSM del semaforo

Aggiungiamo la funzionalità che il controllore non commuta mai da verde a rosso senza aver prima atteso 1024 cicli di clock.

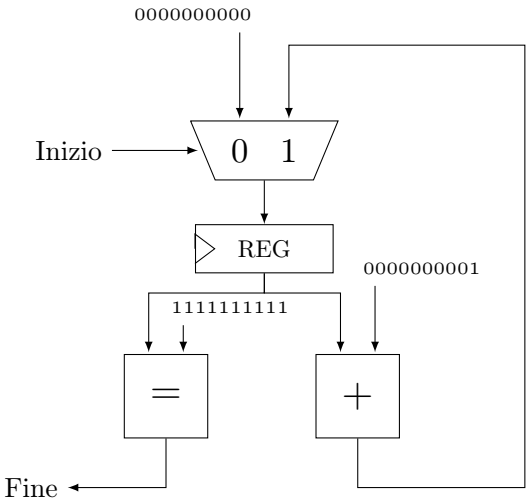


Figura 69: Datapath del contatore 1024

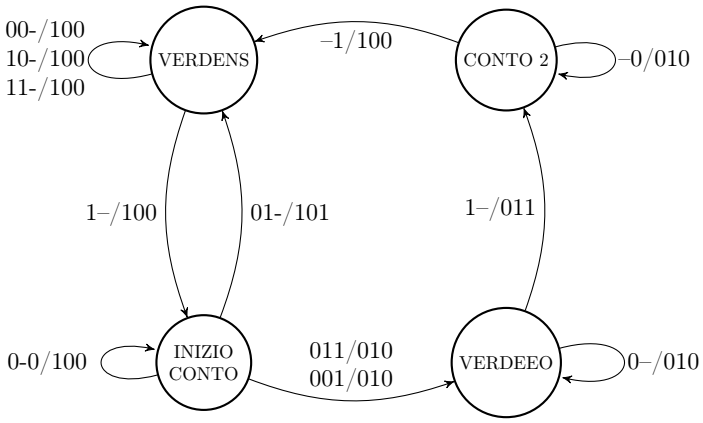


Figura 70: FSM del semaforo

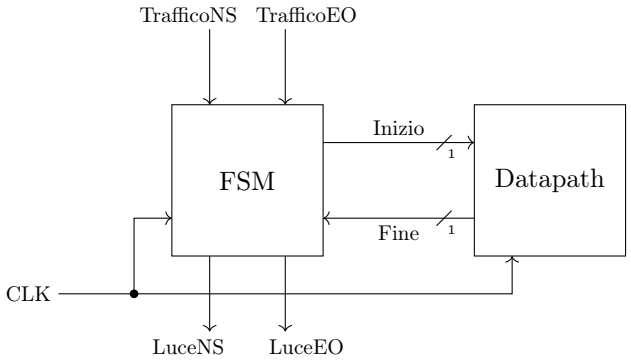


Figura 71: Modello FSMD del semaforo

L'implementazione in Verilog è la seguente:

```
module SemaforoTemporizzato(
  input rst, clk, trafficons, trafficoeo,
  output reg lucens, luceeo);

  reg [1:0] stato = 2'b00;
  reg [1:0] stato_prossimo = 2'b00;
  reg inizio = 1'b0, fine = 1'b0;
  reg [3:0] registro = 3'b000;

  always @(clk) begin : UPDATE
    if(rst) stato = 2'b00;
    else stato = stato_prossimo;
  end

  always @(clk) begin : DATAPATH
    if(inizio) begin
      registro = 3'b000;
      fine = 1'b0;
    end else begin

    if(registro < 3'b111) begin
      registro = registro + 1'b1;
      fine = 1'b0;
    end
    else begin
      fine = 1'b1;
    end
  end
  ...
```

```
  ...
  always @(stato, trafficons, trafficoeo, fine) begin : FSM
    case(stato)
    2'b00:
      if(~trafficons && trafficoeo) begin
        lucens = 1'b1; luceeo = 1'b0; inizio = 1'b1;
        stato_prossimo = 2'b10;
      end else begin
        lucens = 1'b1; luceeo = 1'b0;
        stato_prossimo = 2'b00;
      end
    2'b01:
      if(~trafficons) begin
        lucens = 1'b0; luceeo = 1'b1;
        stato_prossimo = 2'b01;
      end else begin
        lucens = 1'b0; luceeo = 1'b1; inizio = 1'b1;
        stato_prossimo = 2'b11;
      end
    2'b10:
      begin inizio = 1'b0;
      if(trafficons) begin
        lucens = 1'b1; luceeo = 1'b0;
        stato_prossimo = 2'b00;
      end
      else if(fine) begin
        lucens = 1'b0; luceeo = 1'b1;
        stato_prossimo = 2'b01;
      end else begin
        lucens = 1'b1; luceeo = 1'b0;
        stato_prossimo = 2'b10;
      end end
    2'b11:
      begin inizio = 1'b0;
      if(fine) begin
        lucens = 1'b1; luceeo = 1'b0;
        stato_prossimo = 2'b00;
      end else begin
        lucens = 1'b0; luceeo = 1'b1;
        stato_prossimo = 2'b11;
      end end
    endcase
  end
end
```

18 High Level Synthesis

Verrà fatto vedere in C come si può scrivere un codice che verrà poi sintetizzato in VHDL. Implementiamo l'algoritmo per calcolare il massimo comune divisore:

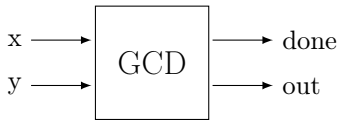


Figura 72: Circuito per il calcolo del massimo comune divisore

Bisogna indicare al circuito quando il risultato è giusto, quindi si aggiunge un segnale di controllo per indicare che il risultato è pronto e la chiamo **done**.

Il codice in C è il seguente:

```c
int gcd(int x, int y) {
  int temp;
  while(x > 0) {
    if(y <= x) { // Si mette y <= x per poter riutilizzare lo stesso
componente del > ma negato
      temp = x;
      x = y;
      y = temp;
    }
    x = x - y;
  }
  return y;
}
```

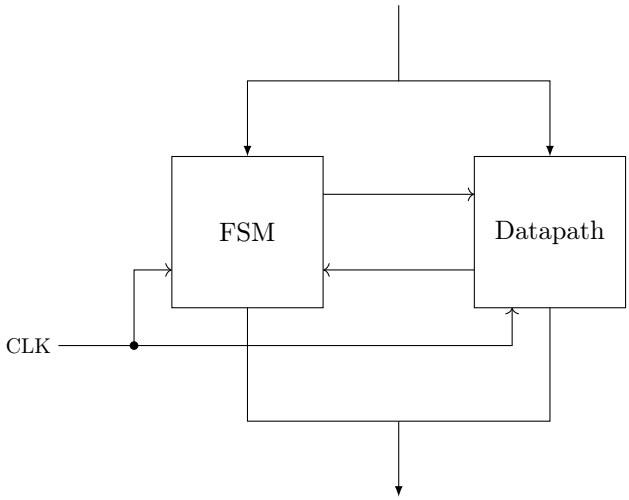


Figura 73: Modello del progetto a cui si vuole arrivare

- **Scheduling**: si decide in che cicli di clock effettuare le operazioni

- **Allocation**: si decide dove allocare le operazioni

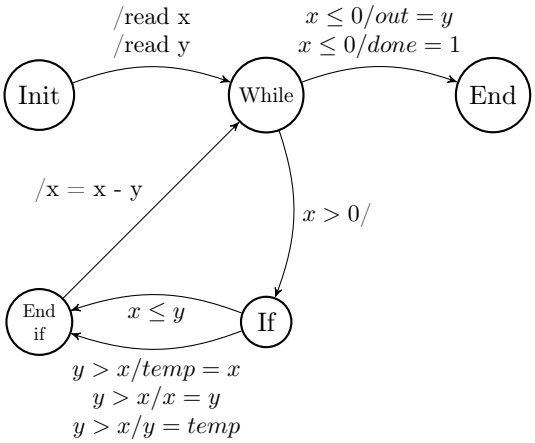


Figura 74: FSM annotata

Ad esempio con 8 cicli di clock si produce il risultato se $x = 8$ e $y = 4$. Per capire quali sono i registri necessari dalla FSM si controlla quali sono le variabili che vengono assegnate in un ciclo di clock e vengono utilizzate in un ciclo di clock diverso, in quel caso si utilizza un registro. In questo caso si utilizzano 2 registri, uno per x e uno per y, $temp$ è soltanto un filo perchè viene usato soltanto in un cilclo di clock.

Per implementare il circuito controllore si utilizzano le condizioni definite nel diagramma di stato. Si aggiunge un segnale di controllo per ogni condizione:

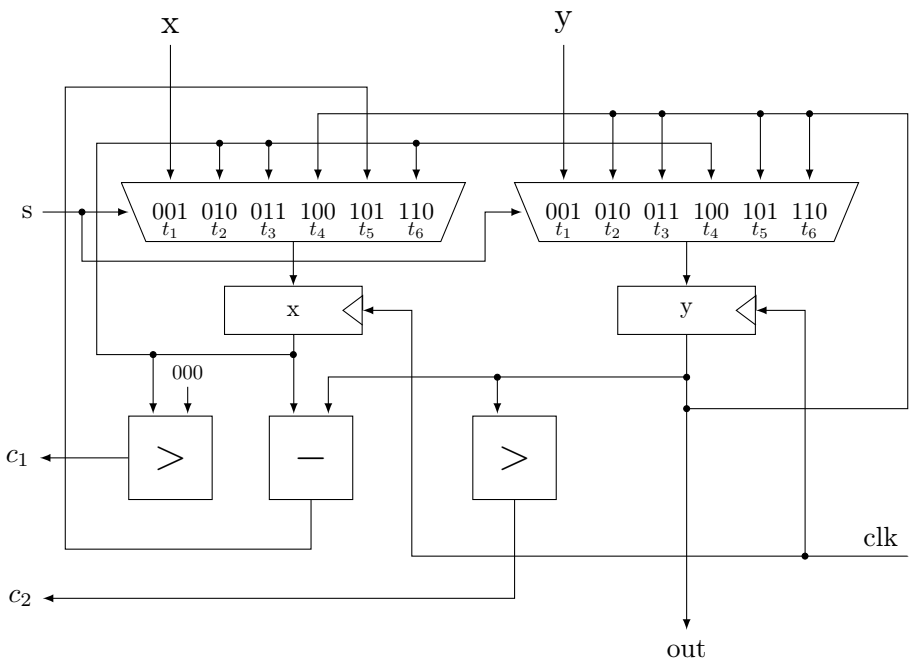


Figura 75: Datapath del massimo comune divisore

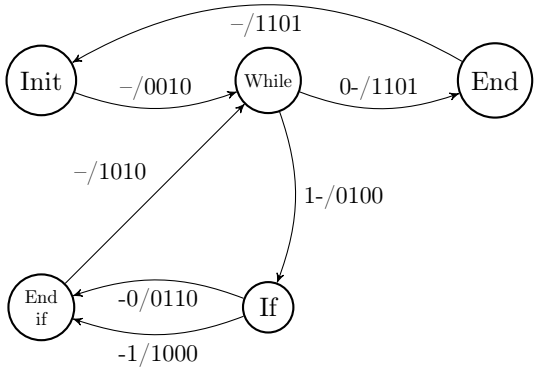


Figura 76: FSM con segnali di controllo

Unendo la FSM con il datapath si ottiene il circuito finale:

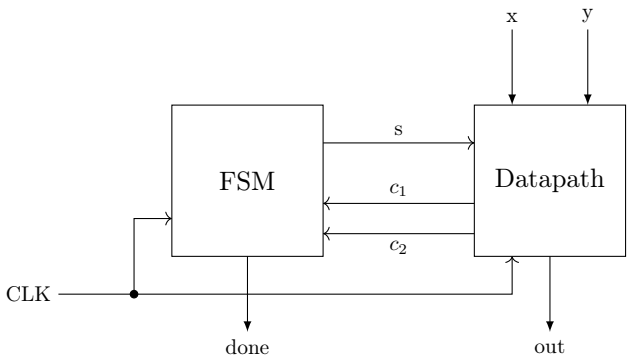


Figura 77: FSMD Finale

19 Architettura di un elaboratore

Si vuole costruire un dispositivo che sia configurabile e quindi possa eseguire diversi programmi.

19.1 Componenti base

- Sommatore
- Moltiplicatore
- Invertitore
- Selettore del numero maggiore

Esempio 19.1
Alcuni esempi di operazioni che si possono fare con i componenti base:

A B	SelOp1 SelOp2	SelOpr	LoarA LoadB	SelOut	Out
Dati	Operandi	Operazione	Memorizzazione	Risultato	
$A = B + C$					
B C	00 01	00	– –	10	$B + C$
$C = A \times 2$					
A 2	00 01	01	10 –	11	$0 \dots 0$
$D = C + B$					
- B	10 01	00	– –	10	$D = A \cdot 2 + B$
$E = A - B$ RegistroA ← -B					
B -	00 –	10	10 –	11	$0 \dots 0$
$if(E > A)\ O = E;$ RegistroB ← A					
A 0	00 01	00	00 10	11	$0 \dots 0$
- -	11 10	00	10 00	11	$0 \dots 0$
- -	10 11	11	– –	10	O

Si vuole costruire un dispositivo che sia abbastanza configurabile da poter rappresentare tutti gli algoritmi possibili. Se una macchina è in grado di rappresentare tutti gli algoritmi possibili allora è detta **Turing Complete**.

Questo approccio permette di programmare in binario, che risulta scomodo e complicato, quindi sono stati creati linguaggi di livello sempre più alto per semplificare la programmazione. Il passo più alto del binario è l'**Assembly**, seguito dai linguaggi di programmazione come il **C**.