

INCONTRO INFORMATIVO SUI LINGUAGGI DI DESCRIZIONE DELL'HARDWARE (HDL)

RELATORI:

Giuliano Cardinali

Emanuel Conti

A CURA DI:

<https://www.pidrsm.sm>

<https://www.facebook.com/professionistiperinnovazionedigitale>

PROFESSIONISTI PER L'INNOVAZIONE DIGITALE

— ASSOCIAZIONE —





PARTE PRIMA

RETI LOGICHE

CENNI STORICI

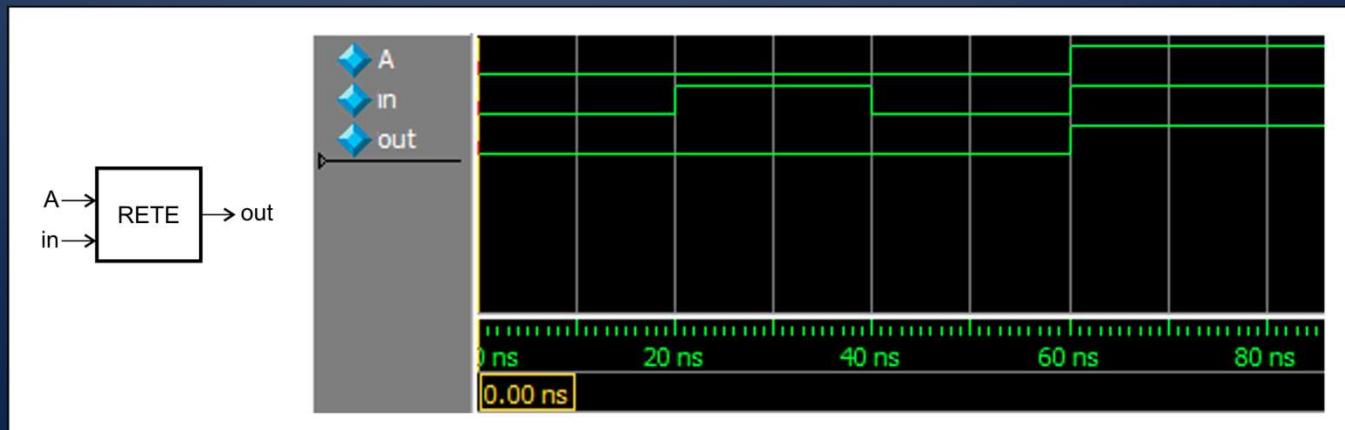
- Sistema binario: Leibniz (1705).
- Algebra booleana: Boole (1847).
- Algebra booleana applicata ai circuiti elettrici: grazie a Pierce (1886), al lavoro di alcuni filosofi e matematici dell'ottocento e a seguito degli sviluppi tecnologici dell'inizio del XX secolo, si è giunti alla realizzazione dei primi circuiti logici digitali nel 1924 grazie a Walther Bothe.
- Claude Shannon, il padre della Teoria dell'Informazione, nel 1938 ha dimostrato che la logica a due valori di Boole e il lavoro di Pierce erano applicabili ai circuiti elettrici (relè) ed elettronici (tubi a vuoto) contribuendo fortemente a creare circuiti digitali descritti mediante la «Teoria delle Reti Logiche».

LE RETI LOGICHE

- Le Reti Logiche elaborano informazioni. Sono alla base di molti dispositivi elettronici e sono il cuore dei moderni computer digitali.
- L'algebra di Boole è usata per descriverne struttura e comportamento.
- Normalmente i valori trattati sono simboli che rappresentano vero e falso (per convenzione 1 e 0). Le variabili adottano i valori di cui sopra e sono pertanto definite binarie.

SEGNALI NEL CONTESTO DELLE RETI LOGICHE

- Nei circuiti digitali elettronici è utile considerare le variabili binarie come segnali: si tratta di tensioni o correnti che evolvono nel tempo.
- Quando rappresentati idealmente, riportano discontinuità nei passaggi fra i due valori.



SEGNALI REALI E RETI LOGICHE

- Nella realtà delle cose, la funzione che rappresenta l'evoluzione dei valori nel tempo non ammette soluzione di continuità.
- I segnali possono essere degradati da fattori o fenomeni fisici interni o esterni alla rete logica stessa, che possono comprometterne l'integrità.

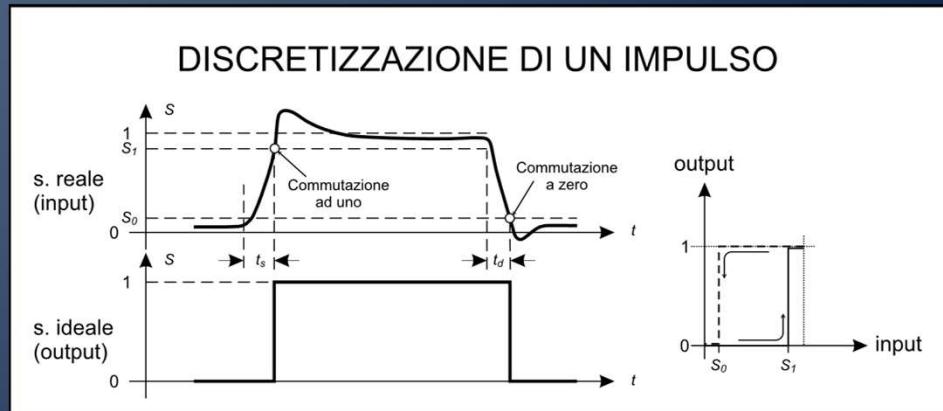


INTEGRITÀ DEI SEGNALI: POSSONO DEGRADARE A CAUSA DI...

- Attenuazione per dissipazione (termica) di energia nei dispositivi e nei collegamenti elettrici.
- Overshoot/Uundershoot, cioè possono assumere valori che "oltrepassano" i confini assoluti ammissibili a causa degli effetti del secondo ordine dovuti alle componenti reattive distribuite nel circuito (induttanze e capacità parassite).
- In regime di parametri distribuiti, ritardi nella propagazione per motivi fisici ineludibili (tempi di propagazione non trascurabili nelle linee)
- Riflessioni fra le due estremità di una linea di trasmissione dovute a disadattamenti di impedenza fra sorgente del segnale, carico e linea stessa.
- Rumore elettrico di diversa natura (rumore termico intrinseco quale l'effetto Johnson) o indotto (da rumore da accoppiamento induttivo/capacitivo o campi elettromagnetici esterni)

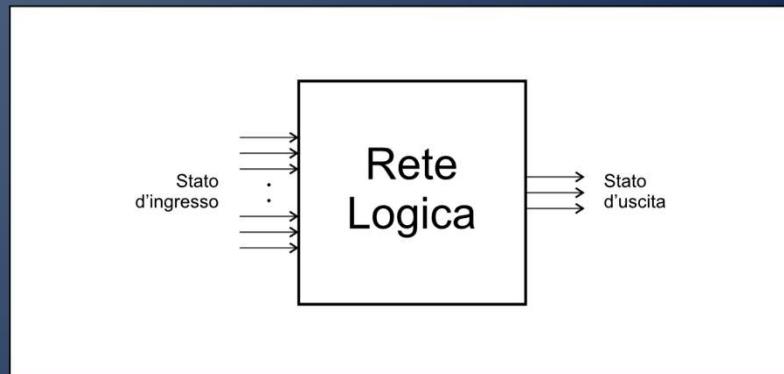
INTERPRETAZIONE DEI SEGNALI NELLE RETI LOGICHE

- Per mitigare gli errori di interpretazione dei segnali in variabili binarie, si usa definire due soglie (livelli convenzionali) S_0 e S_1 . Quando il segnale di input sorpassa la soglia S_1 si avrà il valore 1 in uscita. Perché l'uscita possa tornare a 0, l'ingresso deve scendere sotto la soglia S_0 . L'isteresi definita dalle soglie e dallo stato di uscita, permette di ottenere una uscita priva di incertezze (spike) in presenza di rumore o disturbi di piccola entità.
- La discretizzazione operata dalle soglie su segnali perturbati o degradati, genera comunque fenomeni di jitter (tremolii nel tempo ai cambi di valore). Le soglie fanno anche sì che un impulso risulti leggermente distorto nella sua durata e ritardato nel tempo. Nella pratica delle cose, la discretizzazione di un segnale non produce mai uscite ideali come in figura.



LE RETI LOGICHE COME BLACKBOX

- Le reti logiche sono assimilabili a «scatole nere» (blackbox) che possono avere stato di ingresso e stato d'uscita.
- Il raggruppamento di più segnali (variabili binarie) correlati è assimilabile ad un «vettore» che rappresenta un cosiddetto «stato»; uno stato può, a sua volta, rappresentare singolarmente più valori ed avere quindi una corrispondenza con «oggetti» conosciuti, come ad esempio numeri o caratteri. Una singola variabile binaria può essa stessa essere considerata uno stato.
- Gli stati osservabili sono usati per studiarle e quindi, quando possibile, per descriverne il funzionamento. Possono avere anche stati interni non osservabili benché necessari al loro funzionamento.



IL PROGETTO LOGICO (1)

- Il progetto logico mira a realizzare dispositivi digitali che elaborano segnali in entrata e producono segnali in uscita. Il processo di definizione della struttura di una rete logica partendo dalle specifiche è detto «Sintesi». L'operazione opposta è detta «Analisi».
- Esistono diverse metodologie per la Sintesi: dai metodi «manuali» ai procedimenti automatici. Attualmente la sintesi manuale è eseguita molto raramente.
- Per progettare una rete logica bisogna tradurre le specifiche verbali in specifiche formali, usando diagrammi di segnali ideali, tabelle della verità, equazioni booleane per quanto riguarda le cosiddette «reti combinatorie». Per le reti dette «sequenziali» serviranno generalmente tabelle di flusso, diagrammi degli stati o i cosiddetti ASM chart.

IL PROGETTO LOGICO (2)

- Le specifiche determineranno una possibile architettura per il sistema da progettare.
- Con le specifiche in mano si possono scegliere le tecnologie da adottare in base all'architettura e ad esigenze squisitamente tecniche o commerciali.
- Le tecnologie possono influenzare il tipo di strumenti di sviluppo necessari; a loro volta, a seconda della disponibilità e del costo, possono ridefinire tecnologie e architettura.

LE RETI LOGICHE: OPERATORI FONDAMENTALI (1)

- Adoperando i tre operatori fondamentali (negazione, intersezione ed unione) per «combinare» le variabili binarie, è possibile comporre tutte le espressioni desiderate.
- Nel contesto delle reti logiche le espressioni rappresentano sempre dei predicati. I predicati, a loro volta, possono essere definiti come espressioni che hanno come risultato solo vero o falso. Fa eccezione solo lo scenario di «simulazione» o di analisi, dove la logica viene estesa a tre valori (logica trivale), per cui un predicato potrebbe non fornire una risposta certa a monte di informazioni indefinite.

SIMBOLI DEI CONNETTIVI LOGICI					
AND	 &		\wedge	\cdot	&
OR	 ≥ 1		\vee	$+$	
NOT	\circ		\neg	$-$!

LE RETI LOGICHE: OPERATORI FONDAMENTALI (2)

Una funzione $y = f(x_1, \dots, x_n)$ è una legge $\{0,1\}^n \rightarrow \{0,1\}$ che mappa y per ogni combinazione di x_1, \dots, x_n .

Le funzioni possono essere costruite per enumerazione su tabelle.

x	$f_1(x)$	$x_1 x_2$	$f_2(x_1, x_2)$	$x_1 x_2$	$f_3(x_1, x_2)$
0	1	0 0	0	0 0	0
1	0	0 1 1 0 1 1	1 1 1	0 1 1 0 1 1	0 0 1

Tali funzioni sono descrivibili anche tramite equazioni booleane con l'ausilio dei tre operatori fondamentali relativi a negazione (complementazione o NOT, simboli $\bar{}$ o \neg), somma (OR, simbolo $+$ o $|$) e prodotto (AND, simbolo \cdot o $\&$):

$$f_1(x) = \bar{x},$$

$$f_2(x_1, x_2) = x_1 + x_2,$$

$$f_3(x_1, x_2) = x_1 \cdot x_2.$$

CENNI SULLE PROPRIETÀ ALGEBRICHE (1)

Qualsiasi funzione ammette un'espressione algebrica costruita con tali operatori e con eventuali parentesi.

Gli operatori $\bar{\cdot}$, $+ e \cdot$ godono di proprietà completamente definite dalle funzioni f_1 , f_2 e f_3 viste prima.

Il principio di dualità dell'algebra di Boole permette di formulare le proprietà a coppie semplicemente scambiando gli operatori $+ e \cdot$ e i simboli 0 e 1.

$$P_1) \quad x + 0 = x$$

$$P'_1) \quad x \cdot 1 = x$$

$$P_2) \quad x + 1 = 1$$

$$P'_2) \quad x \cdot 0 = 0$$

$$P_3) \quad x + x = x$$

$$P'_3) \quad x \cdot x = x$$

$$P_4) \quad x + \bar{x} = 1$$

$$P'_4) \quad x \cdot \bar{x} = 0$$

$$P_5) \quad x_1 + x_2 = x_2 + x_1$$

$$P'_5) \quad x_1 \cdot x_2 = x_2 \cdot x_1$$

$$P_6) \quad (x_1 + x_2) + x_3 = x_1 + (x_2 + x_3)$$

$$P'_6) \quad (x_1 \cdot x_2) \cdot x_3 = x_1 \cdot (x_2 \cdot x_3)$$

$$P_7) \quad (x_1 \cdot x_2) + (x_1 \cdot x_3) = x_1 \cdot (x_2 + x_3)$$

$$P'_7) \quad (x_1 + x_2) \cdot (x_1 + x_3) = x_1 + (x_2 \cdot x_3)$$

Infine possiamo definire la proprietà "autoduale" in questo modo:

$$P_8) \quad \bar{\bar{x}} = x$$

CENNI SULLE PROPRIETÀ ALGEBRICHE (2)

Si noti come le proprietà P_6 e P'_6 (proprietà associativa), dal momento che indicano come l'ordine di una serie di $+$ e di \cdot sia inessenziale, possano estendere la definizione di tali operatori ad un numero arbitrario di variabili. Anche le proprietà P_2 e P'_2 giustificano l'esistenza di operatori a n variabili. Ad esempio, la proprietà P_2 può rappresentare una funzione $x_1 + \dots + x_n$ che vale 0 solo se tutte le variabili x_i valgono 0. Dualmente parlando, la proprietà P'_2 invece descrive una funzione $x_1 \cdot \dots \cdot x_n$ che vale 1 solo se tutte le variabili x_i possiedono valore 1.

Notare anche che la proprietà P'_7 (proprietà distributiva) non è valida nell'algebra normale.

Altre proprietà utili alla semplificazione algebrica (e non solo) sono:

$$P_9) \quad x_1 \cdot x_2 + x_1 \cdot \bar{x}_2 = x_1$$

$$P'_9) \quad (x_1 + x_2) \cdot (x_1 + \bar{x}_2) = x_1$$

$$P_{10}) \quad x_1 + x_1 \cdot x_2 = x_1$$

$$P'_{10}) \quad x_1 \cdot (x_1 + x_2) = x_1$$

Infine citiamo il Teorema di De Morgan nelle sue due forme:

$$P_{11}) \quad \overline{x_1 + x_2 + \dots + x_n} = \bar{x}_1 \cdot \bar{x}_2 \cdot \dots \cdot \bar{x}_n$$

$$P'_{11}) \quad \overline{x_1 \cdot x_2 \cdot \dots \cdot x_n} = \bar{x}_1 + \bar{x}_2 + \dots + \bar{x}_n$$

ENUMERAZIONE DI FUNZIONI (1)

Esistono $2^{(2^n)}$ funzioni possibili che si possono ottenere dalla combinazione di n variabili binarie. Per $n=2$ sono le seguenti:

2^n	$\left[\begin{array}{c cc} a & b \\ \hline 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{array} \right]^n$	$f_0 = \text{constante } 0$
	0	$f_1 = a \wedge b$
	1	$f_2 = p_2$
	0	$f_3 = a$
	1	$f_4 = p_1$
	0	$f_5 = b$
	1	$f_6 = a \oplus b$
	1	$f_7 = a \vee b$
	0	$f_8 = \neg(a \vee b)$
	0	$f_9 = a \equiv b$
	1	$f_{10} = \bar{b}$
	0	$f_{11} = s_1$
	1	$f_{12} = \bar{a}$
	1	$f_{13} = a \rightarrow b$
	1	$f_{14} = \neg(a \wedge b)$
	1	$f_{15} = \text{constante } 1$

ENUMERAZIONE DI FUNZIONI (2)

Esistono $2^{(2^n)}$ funzioni possibili che si possono ottenere dalla combinazione di n variabili binarie. Per $n=2$ sono le seguenti:

$f_0 = constante\ 0$	$f_1 = a \wedge b$	$f_2 = p_2$	$f_3 = a$	$f_4 = p_1$	$f_5 = b$	$f_6 = a \oplus b$	$f_7 = a \vee b$	$f_8 = \neg(a \vee b)$	$f_9 = a \equiv b$	$f_{10} = \bar{b}$	$f_{11} = s_1$	$f_{12} = \bar{d}$	$f_{13} = a \rightarrow b$	$f_{14} = \neg(a \wedge b)$	$f_{15} = constante\ 1$
a	b														
0	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1
0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0
1	0	0	0	1	1	0	0	1	1	0	0	0	1	0	1
1	1	0	1	0	1	0	1	0	1	0	1	0	0	1	0

ENUMERAZIONE DI FUNZIONI (3)

Esistono $2^{(2^n)}$ funzioni possibili che si possono ottenere dalla combinazione di n variabili binarie. Per $n=2$ sono le seguenti:

ENUMERAZIONE DI FUNZIONI (4)

Esistono $2^{(2^n)}$ funzioni possibili che si possono ottenere dalla combinazione di n variabili binarie. Per $n=2$ sono le seguenti:

ENUMERAZIONE DI FUNZIONI (5)

Esistono $2^{(2^n)}$ funzioni possibili che si possono ottenere dalla combinazione di n variabili binarie. Per $n=2$ sono le seguenti:

		$a \quad b$		$f_0 = \text{costante } 0$															
				$f_1 = a \wedge b$	$f_2 = p_2$	$f_3 = a$	$f_4 = p_1$	$f_5 = b$	$f_6 = a \oplus b$	$f_7 = a \vee b$	$f_8 = \neg(a \vee b)$	$f_9 = a \equiv b$	$f_{10} = \bar{b}$	$f_{11} = s_1$	$f_{12} = \bar{a}$	$f_{13} = a \rightarrow b$	$f_{14} = \neg(a \wedge b)$	$f_{15} = \text{costante } 1$	
		a	b	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 1	0 0	1 0	0 1	0 0	1 1	0 1	1 0	1 1
2^n		0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	1	1	
		0	1	0	0	0	0	0	1	1	1	1	1	0	0	0	1	1	
		1	0	0	0	1	1	0	0	1	1	0	0	0	1	1	0	1	
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	

IMPLICAZIONE

$$f_{13} = a \rightarrow b$$

ENUMERAZIONE DI FUNZIONI (6)

Esistono $2^{(2^n)}$ funzioni possibili che si possono ottenere dalla combinazione di n variabili binarie. Per $n=2$ sono le seguenti:

STRUTTURA DELLE RETI LOGICHE

- **Reti Combinatorie**

Assimilabili a funzioni «pure», lo stato di uscita dipende unicamente dai valori dello stato d'ingresso (funzione biunivoca o suriettiva).

- **Reti Sequenziali**

Le uscite dipendono dalla «storia» degli stati d'ingresso fino all'istante di osservazione considerato. Sono definite in termini di reti combinatorie provviste di feedback su se stesse o con l'aggiunta di elementi di memoria.

METODI MANUALI PER LA SINTESI DI RETI COMBINATORIE (1)

- Con le definizioni formali sul funzionamento della rete specificate nelle tabelle di verità o con espressioni, si possono operare trasformazioni atte a produrre strutture idonee alla realizzazione pratica: vale a dire che questo processo prevede il loro riarrangiamento, ad esempio per adattarle alla tecnologia da utilizzare.
- La riorganizzazione delle specifiche implica anche la ricerca del minor costo in termini di dimensione della rete, compatibilmente con le prestazioni richieste. Qui si tenta la minimizzazione della rete tenendo però sempre conto delle prestazioni desiderate.

METODI MANUALI PER LA SINTESI DI RETI COMBINATORIE (2)

- La progettazione manuale di una rete logica implica l'uso di manipolazione algebrica diretta delle equazioni, l'utilizzo di tecniche «grafiche» su tabelle di verità opportunamente riarrangiate, quali le mappe di Veitch-Karnaugh o un mix di queste ed altre tecniche.
- Una rete logica relativamente complessa può essere costituita da diverse sottoreti che possono essere trattate separatamente.
- È possibile considerare l'utilizzo del «minimizzatore logico» Espresso (o di altri strumenti simili) come un ausilio alla progettazione manuale.

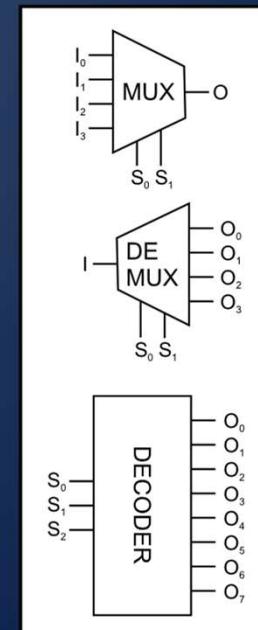
SINTESI E MINIMIZZAZIONE DI RETI COMBINATORIE

- Definizione della rete con tabelle di verità o equazioni booleane (o equivalente)
- Eventuale minimizzazione:
 - Minimizzazione manuale di reti combinatorie
 - Con algebra di Boole (manipolazione algebrica, tediosa ed incline ad errori).
 - Con Mappe di Veitch-Karnaugh (limitate a sei variabili).
 - Con Mappe di Reed-Müller (con utilizzo dell'operatore XOR).
 - Minimizzazione automatica (con computer)
 - Con il metodo tabellare Quine-McCluskey (proibitivo con alto numero di variabili).
 - Con il minimizzatore euristico *Espresso* o strumenti simili.

DALLE EQUAZIONI AI CIRCUITI

Le espressioni sono manipolate al fine di sfruttare blocchi logici predefiniti, quali porte AND, OR, NOT, le porte dotate di intrinseca completezza funzionale qual NAND, NOR, XOR o funzioni più complesse come DECODER, MUX, DEMUX, HALF-ADDER, FULL-ADDER, e altre ancora. Suddette funzioni sono incorporate in chip a bassa densità (SSI) e conosciute come «logica discreta».

INPUT →	0	1		
NOT	1	0		
AND	0	0	0	1
OR	0	1	1	1
NAND	1	1	1	0
NOR	1	0	0	0
XOR	0	1	1	0
XNOR	1	0	0	1



CATEGORIE DI LOGICHE COMBINATORIE

- **Rete logica combinatoria in forma canonica (o normale)**

Prevede due forme:

- Forma normale disgiuntiva (somma di prodotti, o SP)
- Forma normale congiuntiva (prodotti di somme, o PS)

NB: per «*prodotto*» si intende la congiunzione (AND) e per «*somma*» la disgiunzione (OR)

Una rete in forma canonica è caratterizzata da due livelli di logica (i NOT non si contano): si intende dire che un segnale in ingresso attraversa due operatori prima di raggiungere l'uscita.

- **Rete logica combinatoria multilivello**

Si tratta di una rete che vede almeno un segnale di ingresso attraversare più di due operatori prima di raggiungere l'uscita. È una forma generalmente più compatta della precedente, ma solitamente esibisce tempi di propagazione maggiori. Con questa categoria di reti è anche possibile realizzare circuiti basati su un solo tipo di operatore, purché questi sia dotato di completezza funzionale (vale a dire che l'operatore prevede la negazione di almeno una variabile); esempi di tali operatori sono: NAND, NOR e XOR.

RETI LOGICHE COMBINATORIE (FORME CANONICHE)

- Sono reti costituite da due livelli di logica.
- Definiscono la funzione unendo le intersezioni di più segnali o negazione di essi (somme di prodotti o somme di minterm) o intersecando le unioni di più segnali o negazione di essi (prodotti di somme o prodotti di maxterm).
- I Prodotti P_n (minterm) identificano le combinazioni di ingresso (i termini T_n) quando la funzione vale 1. Le Somme S_n (maxterm) identificano i termini per i quali la funzione vale 0.

$f(a, b, c)$

	a	b	c	z	P_0	P_3	P_6	P_7	S_1	S_2	S_4	S_5
T_0	0	0	0	1	1	0	0	0	1	1	1	1
T_1	0	0	1	0	0	0	0	0	0	1	1	1
T_2	0	1	0	0	0	0	0	0	1	0	1	1
T_3	0	1	1	1	0	1	0	0	1	1	1	1
T_4	1	0	0	0	0	0	0	0	1	1	0	1
T_5	1	0	1	0	0	0	0	0	1	1	1	0
T_6	1	1	0	1	0	0	0	1	0	1	1	1
T_7	1	1	1	1	0	0	0	1	0	1	1	1

ESEMPIO DI PROGETTO

- La funzione z descritta dalla tabella a lato, può essere scritta sotto forma di espressione booleana usando una delle due forme canoniche. In questo esempio è stata usata la forma SP (Somma di Prodotti).
- Dalla descrizione tabellare di 'z' si selezionano i termini dove la funzione vale 1 (i soli minterm); nel predicato $z = f(a, b, c)$ si riportano i termini relativi ai prodotti P_0, P_3, P_6 e P_7 avendo cura di negare gli argomenti (a, b oppure c) che per un dato prodotto P_n riportano zero.
- Applicando le proprietà dell'algebra viste prima, è possibile ottenere una forma ridotta oppure alternativa dell'espressione originale (si noti che l'ultima versione della funzione z nel riquadro in basso a destra non è in forma canonica, cioè vedremo poi che rappresenta una logica con più di due livelli).

	a	b	c	z	P_0	P_3	P_6	P_7	S_1	S_2	S_4	S_5
T_0	0	0	0	1	1	0	0	0	1	1	1	1
T_1	0	0	1	0	0	0	0	0	0	1	1	1
T_2	0	1	0	0	0	0	0	0	1	0	1	1
T_3	0	1	1	1	0	1	0	0	1	1	1	1
T_4	1	0	0	0	0	0	0	0	1	1	0	1
T_5	1	0	1	0	0	0	0	0	1	1	1	0
T_6	1	1	0	1	0	0	0	1	0	1	1	1
T_7	1	1	1	1	0	0	0	1	1	1	1	1

Selezione prodotti

$$z = \sum(P_0, P_3, P_6, P_7)$$

$$z = \bar{a} \cdot \bar{b} \cdot \bar{c} + \bar{a} \cdot b \cdot c + a \cdot b \cdot \bar{c} + a \cdot b \cdot c$$

Minimizzazione algebrica

$$z = \bar{a} \cdot \bar{b} \cdot \bar{c} + \bar{a} \cdot b \cdot c + a \cdot b \cdot \bar{c} + a \cdot b \cdot c$$

$$z = \bar{a} \cdot \bar{b} \cdot \bar{c} + b \cdot c + b \cdot \bar{c} + a \cdot b + a \cdot b$$

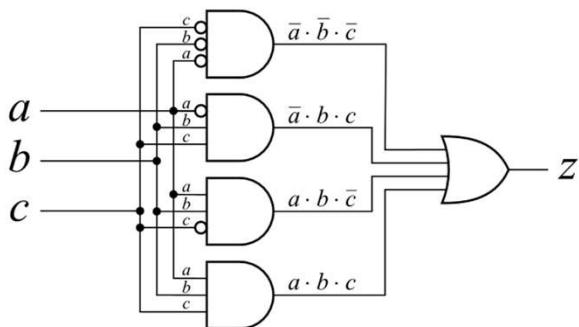
$$z = \bar{a} \cdot \bar{b} \cdot \bar{c} + a \cdot b + b \cdot c$$

$$z = \bar{a} \cdot \bar{b} \cdot \bar{c} + b \cdot (a + c)$$

IMPLEMENTAZIONE CON PORTE LOGICHE

- È possibile realizzare una versione non minimizzata e in forma canonica SP della funzione z usando soltanto porte logiche corrispondenti agli operatori di negazione, intersezione ed unione.
- Si noti che alcuni termini che vanno alle porte AND tutte a tre ingressi, attraversano un operatore di negazione (costituito da un cerchietto vuoto).
- Le forme canoniche sono ideali per realizzare circuiti logici affidabili e si adattano naturalmente alle strutture dei dispositivi logici programmabili più semplici, ammettendo una mappatura tecnologica diretta.

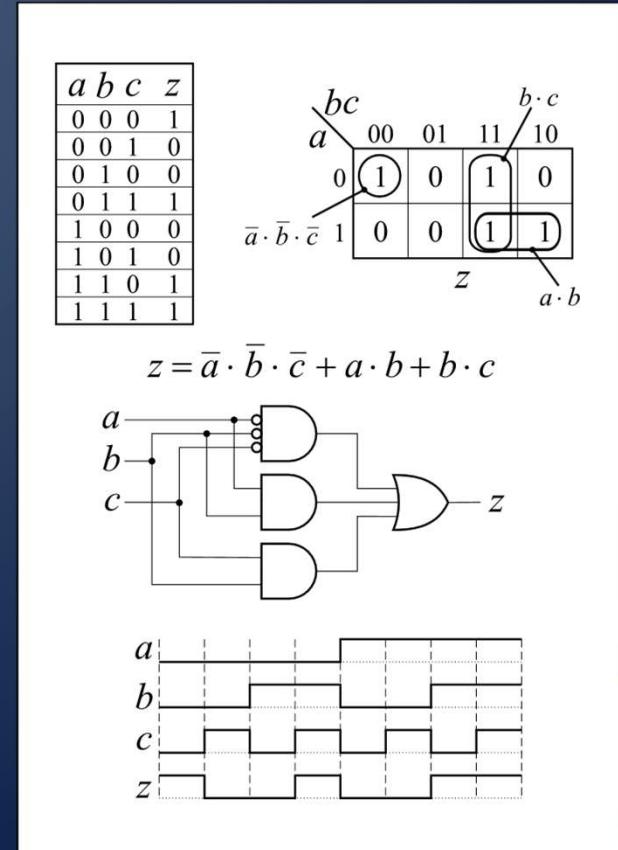
Funzione 'z' in forma canonica SP
non minimizzata realizzata con porte logiche



$$z = \bar{a} \cdot \bar{b} \cdot \bar{c} + \bar{a} \cdot b \cdot c + a \cdot b \cdot \bar{c} + a \cdot b \cdot c$$

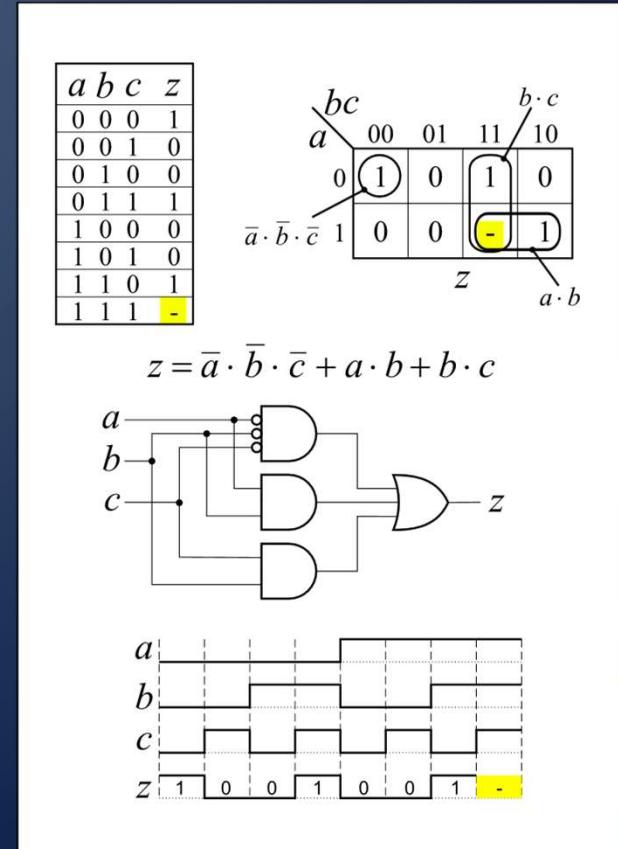
MINIMIZZAZIONE CON METODO GRAFICO (1)

- Oltre alla manipolazione algebrica vista prima, un sistema alternativo per minimizzare una funzione prevede l'uso delle Mappe di Veitch-Karnaugh, conosciute principalmente come Mappe di Karnaugh o K-Map. Sfruttano le proprietà P_9 o P'_9 , viste in precedenza, mettendo in evidenza la cosiddetta adiacenza logica di due o più *minterm* (SP) o di due o più *maxterm* (PS).
- Nel caso delle forme SP, nelle K-Map si cerca di raggruppare tutti i *minterm* che geometricamente possono rappresentare dei quadrati o dei rettangoli i cui lati hanno come altezza o larghezza un numero di caselle equivalente ad una potenza di due. Nel caso delle forme PS si raggruppano i maxterm allo stesso modo. Il raggruppamento di termini in aree regolari con le caratteristiche sopra citate prende il nome di «copertura». Più ampia è una copertura e minore sarà il numero di variabili coinvolte e quindi richiederà un operatore più «piccolo».



MINIMIZZAZIONE CON METODO GRAFICO (2)

- In alcuni casi le tabelle che rappresentano le funzioni possono essere non completamente specificate. Vale a dire che per alcuni termini il risultato è ininfluente e può essere considerato sia zero che uno a seconda della convenienza. Nell'esempio a lato il termine T_7 è considerato come *minterm* (uno) in quanto permette di avere due coperture sovrapposte più ampie della singola casella. Vale a dire che per quelle coperture sono coinvolte solo due variabili invece di tre, permettendo di avere una funzione più «piccola» e quindi un circuito più semplice.
- Le K-Map sono utili per minimizzare funzioni fino a sei variabili. Oltre le sei variabili l'adiacenza logica è difficilmente percepibile, per cui non è più conveniente usarle.



MINIMIZZAZIONE CON IL SOFTWARE ESPRESSO (1)

- Il minimizzatore logico Espresso può essere considerato un ausilio alla sintesi manuale. È anche stato usato come minimizzatore all'interno di software CAE. È nato all'Università di Berkeley in California come ausilio alla minimizzazioni di reti combinatorie con molte variabili d'ingresso.
- Espresso è un minimizzatore euristico che basa il processo di minimizzazione su alcune regole empiriche oltre a proprietà dell'algebra di Boole. Il risultato potrebbe essere sub-ottimale, comunque in grado di produrre in breve tempo risultati di eccellente qualità anche in caso di reti con un importante numero di variabili in ingresso.
- Il minimizzatore logico Espresso è largamente diffuso in rete e può essere trovato sia in codice sorgente che come *build* per diverse piattaforme (<https://github.com/Gigantua/Espresso>).

MINIMIZZAZIONE CON IL SOFTWARE ESPRESSO (2)

- Minimizzazione della funzione z.

File z_sp.pla

```
.i 3
.o 1
.p 4

.ilb a b c
.ob z

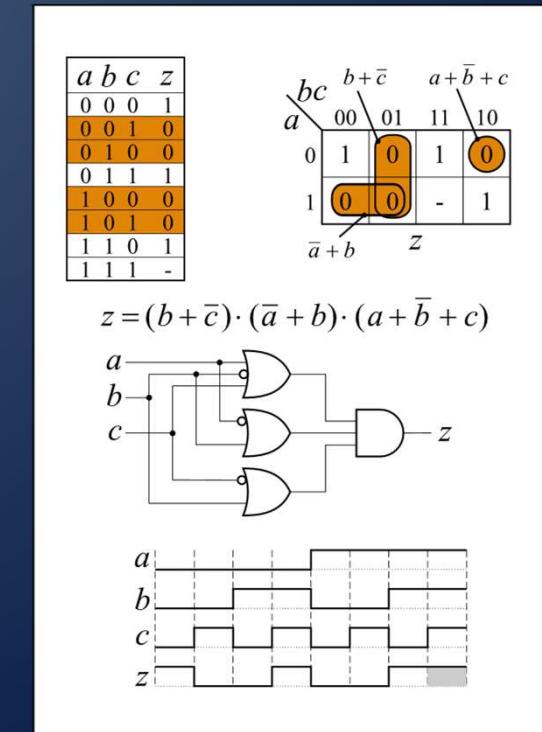
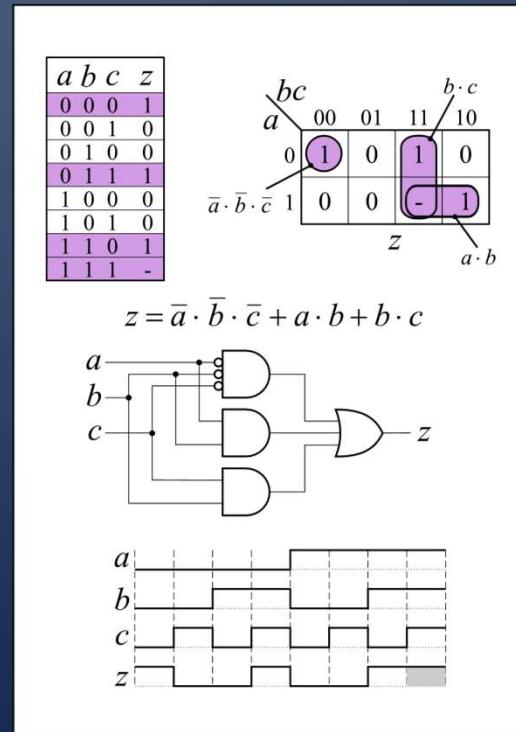
000 1
011 1
110 1
111 -
.e
```

File z_ps.pla

```
.i 3
.o 1
.p 5
.type r

.ilb a b c
.ob z

001 0
010 0
100 0
101 0
111 -
.e
```



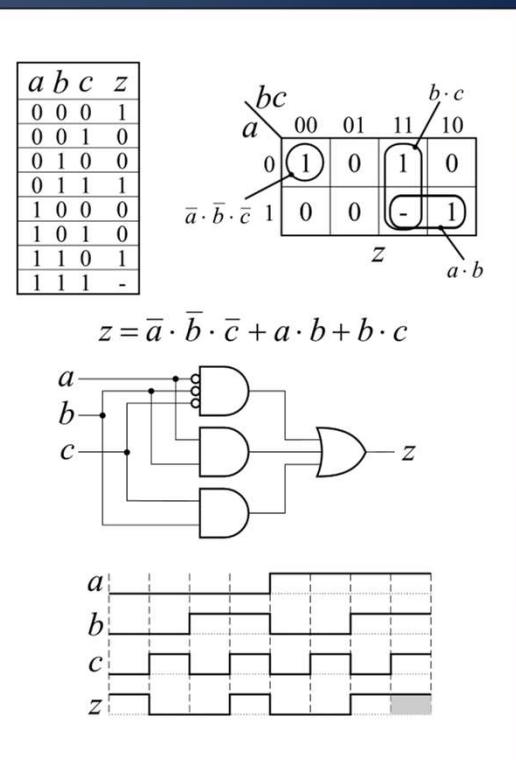
MINIMIZZAZIONE CON IL SOFTWARE ESPRESSO (3)

- Minimizzazione della funzione z definita sui minterm.

Output del minimizzatore per z_sp.pla

```
.i 3  
.o 1  
.p 4  
  
.ilb a b c  
.ob z  
  
000 1  
011 1  
110 1  
111 -  
.e
```

```
C:\Espresso>espresso z_sp.pla  
.i 3  
.o 1  
.ilb a b c  
.ob z  
.p 3  
000 1  
11- 1  
-11 1  
.e  
  
C:\Espresso>espresso -o eqntott z_sp.pla  
z = (!a & !b & !c) | (a & b) | (b & c);  
  
C:\Espresso>
```



MINIMIZZAZIONE CON IL SOFTWARE ESPRESSO (4)

- Minimizzazione della funzione z definita sui maxterm.

Output del minimizzatore per z_ps pla

```
.i 3
.o 1
.p 5
.type r
.phase 0

.ilb a b c
.ob z

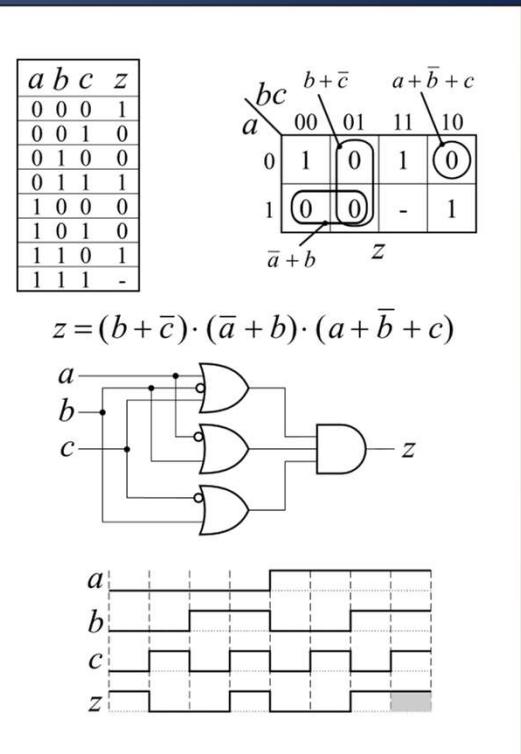
001 0
010 0
100 0
101 0
111 -
```



```
C:\Espresso>espresso z_ps pla
.i 3
.o 1
.ilb a b c
.ob z
#.phase 0
.p 3
010 1
10- 1
-01 1
.e

C:\Espresso>espresso -o eqntott z_ps pla
z = (!a & b & !c) | (a & !b) | (!b & c);

C:\Espresso>
```



PERCHÉ ESPRESSO?

```
# 7seg.pla
.i 4
.o 7
.p 16
.ilb i3 i2 i1 i0
.ob a b c d e f g
0000 1111110
0001 0110000
0010 1101101
0011 1111001
0100 0110011
0101 1011011
0110 1011111
0111 1110000
1000 1111111
1001 1110011
1010 1110111
1011 0011111
1100 1001110
1101 0111101
1110 1001111
1111 1000111
.e
```

```
C:>espresso 7seg.pla
.i 4
.o 7
.ilb i3 i2 i1 i0
.ob a b c d e f g
.p 14
1011 0001100
100- 1000000
001- 0001001
-000 0001010
0-11 1110000
0100 0110011
-00- 0110000
1101 0111101
0101 1011011
-0-0 1100100
11-0 1001110
111- 1000111
10-- 0010011
0110 1011111
.e
```

```
C:>espresso -o eqntott 7seg.pla
a = (i3 & !i2 & !i1) | (!i3 & i1 & i0) | (!i3 & i2 &! i1 & i0) | (!i2 & !i0)
| (i3 & i2 & !i0 ) | (i3 & i2 & i1) | (!i3 & i2 & i1 & !i0);

b = (!i3 & i1 & i0) | (!i3 & i2 & !i1 & !i0) | (!i2 & !i1)
| (i3 & i2 & !i1 & i0) | (!i2 & !i0);

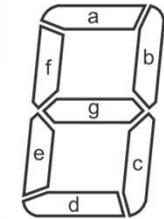
c = (!i3 & i1 & i0) | (!i3 & i2 & !i1 & !i0) | (!i2 & !i1)
| (i3 & i2 & !i1 & i0) | (!i3 & i2 & !i1 & i0) | (i3 &! i2)
| (!i3 & i2 & i1 & !i0);

d = (i3 & !i2 &i1 & i0) | (!i3 & !i2 & i1) | (!i2 & !i1 & !i0)
| (i3 & i2 & !i1 & i0) | (!i3 & i2 & !i1 & i0 ) | (i3&i2&!i0)
| (!i3 & i2 & i1 & !i0);

e = (i3 & !i2 & i1 & i0) | (i3 & i2 & !i1 & i0) | (!i2 & !i0)
| (i3 & i2 & !i0) | (i3 & i2 & i1) | (!i3 & i2 & i1 & !i0);

f = (!i2 & !i1 & !i0) | (!i3 & i2 & !i1 & !i0) | (!i3 & i2 & !i1 & i0)
| (i3 & i2 & !i0) | (i3 & i2 & i1) | (i3 & !i2) | (!i3 & i2 & i1 & !i0);

g = (!i3 & !i2 & i1) | (!i3 & i2 & !i1 & !i0) | (i3 & i2 & !i1 & i0)
| (!i3 & i2 & !i1 & i0) | (i3 & i2 & i1) | (i3 & !i2)
| (!i3 & i2 & i1 & !i0);
```



0 1 2 3 4 5 6 7 8 9 A b C d E F

GLI ASIC (1)

- ASIC è l'acronimo di Application Specific Integrated Circuit.
- Nell'ambito delle logiche digitali, si tratta di dispositivi progettati appositamente per ottimizzare, in termini di velocità ed efficienza, uno specifico compito di calcolo o elaborazione.
- Questi dispositivi sono storicamente sviluppati dalle aziende di semiconduttori su commessa del cliente o dal cliente stesso con un supporto molto importante da parte del produttore.
- La progettazione di un ASIC solitamente è un processo lungo e costoso.

GLI ASIC (2)

- Gli ASIC riducono al minimo il costo del prodotto finale e sono quindi giustificati solo da volumi di produzione molto importanti.
- Quando l'ASIC implementa funzioni abbastanza generiche da poter essere utilizzato su più prodotti o da più clienti, si parla di ASSP (Application Specific Standard Product/Part). Essendo utilizzati su più progetti, gli ASSP contribuiscono a diminuire i "costi ingegneristici non ricorrenti" (NRE cost) nel ciclo di sviluppo.
- L'impossibilità di ricorrere ogni volta alla progettazione di soluzioni completamente personalizzate (ASIC) o di funzioni standard specifiche (ASSP), ha portato le aziende di semiconduttori a sviluppare le cosiddette Logiche Programmabili.

LE LOGICHE PROGRAMMABILI (PLD)

- Le Logiche Programmabili (PLD, Programmable Logic Device) sono dispositivi la cui architettura interna è predefinita dal costruttore.
- Le PLD offrono il vantaggio, rispetto agli ASIC, di vedere enormemente ridotti i costi e i tempi necessari per lo sviluppo e di non richiedere quantitativi minimi importanti in termine di parti da ordinare al produttore.
- Le tecnologia delle PLD sono cambiate di pari passo con quelle della produzione di altre tipologie di chip, seguendone anche l'evoluzione in termini di densità di integrazione.
- Agli albori della tecnologia, esistevano PLD con scala d'integrazione SSI (Small scale Integration) che implementavano funzionalità abbastanza ridotte. Poi sono state adottate densità sempre maggiori, come la MSI (Medium Scale Integration), la LSI (Large Scale Integration) e poi VLSI (Very Large Scale Integration). Quest'ultima è usata nelle Logiche Programmabili moderne. Le PLD moderne più grandi programmabili dall'utente, possono facilmente ospitare al loro interno oltre 100 milioni di funzioni logiche semplici (dette porte o gates).

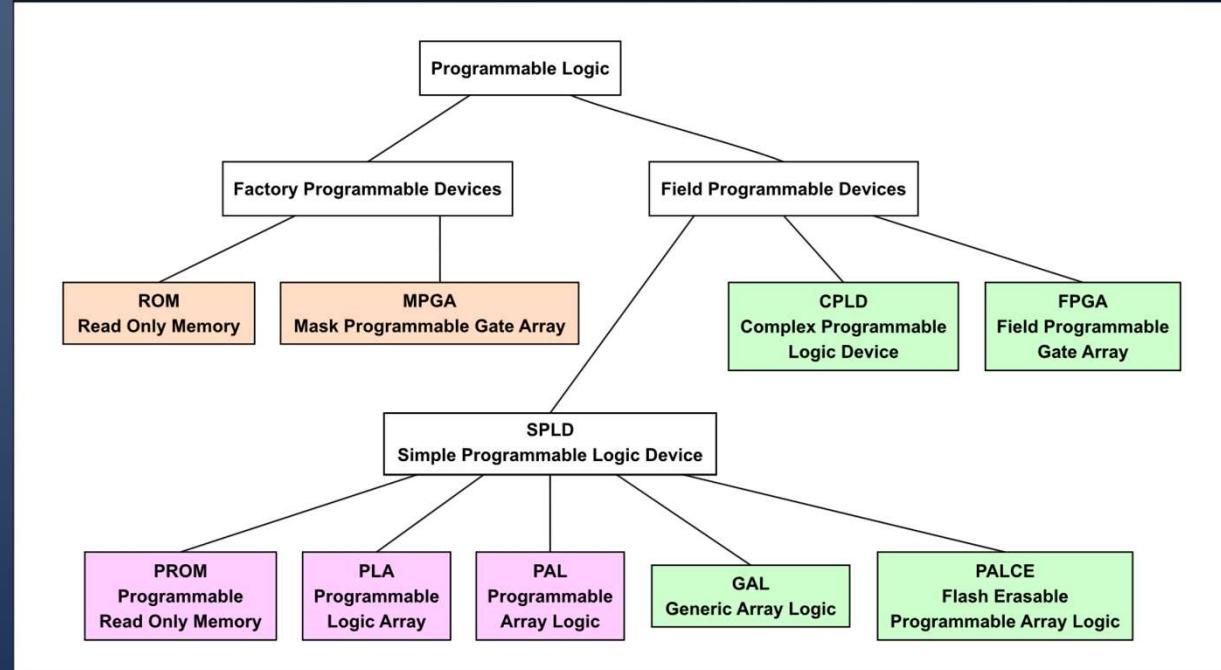
TASSONOMIA DELLE LOGICHE PROGRAMMABILI (1)

Le logiche programmabili si possono suddividere in:

- Le logiche programmabili «sul campo» (gli FPD, Field Programmable Device), cioè programmabili in linea di produzione solo la prima volta o programmabili/aggiornabili quando sono già montati sul circuito stampato (PCB).
- Le logiche pre-programmate dal costruttore (i Factory Programmable Device).

In generale si tratta di «Gate Array», cioè dispositivi con molte funzioni logiche alle quali mancano solo i collegamenti (le metallizzazioni). Sono logiche da considerare quando i volumi di produzione richiesti sono abbastanza importanti ma non tali da giustificare il costo dello sviluppo di un vero e proprio ASIC.

In fase di prototipazione il cliente utilizza un FPD, poi richiede al costruttore dispositivi equivalenti pre-programmati su loro specifica.



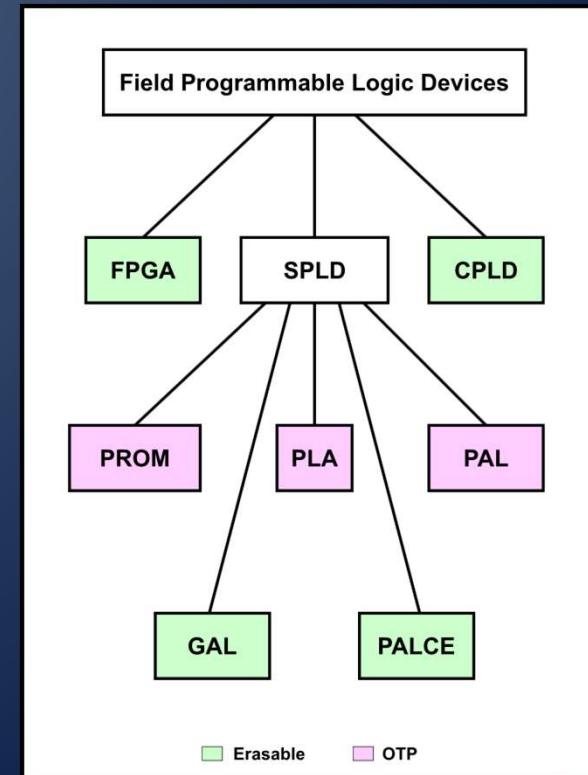
FIELD PROGRAMMABLE LOGIC (1)

SPLD: Simple Programmable Logic Devices

A questo gruppo i dispositivi programmabili un'unica volta (OTP, One Time Programmable) e sono le PROM (Programmable Read Only memory), le PLA (Programmable Logica Array) e le PAL (Programmable Array Logic). Inoltre esistono versioni più moderne delle PAL che sono riprogrammabili: le più note sono GAL e PALCE.

CPLD

Cioè Complex Programmable Logic Devices. Sono dispositivi che ospitano diversi macro-blocchi interconnessi che implementano la forma normale disgiuntiva (forma canonica SP) simili a PAL cancellabili.

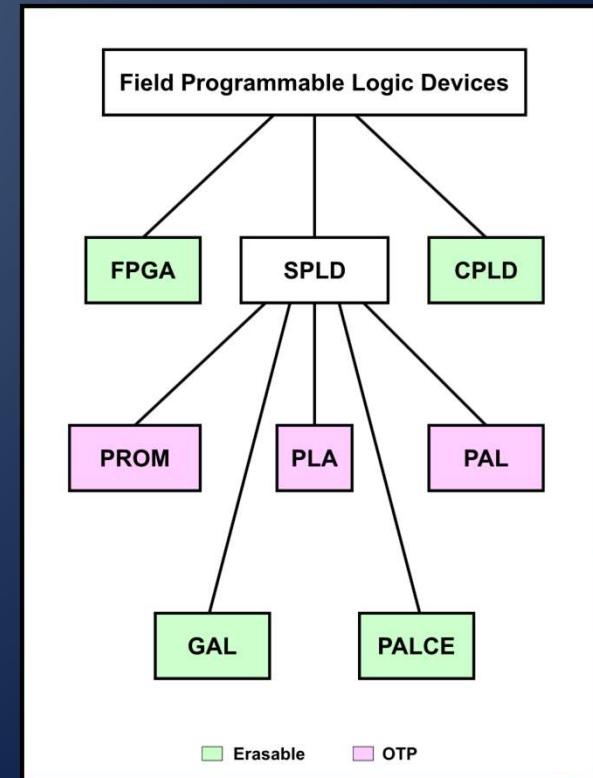


FIELD PROGRAMMABLE LOGIC (2)

FPGA: Field Programmable Gate Array

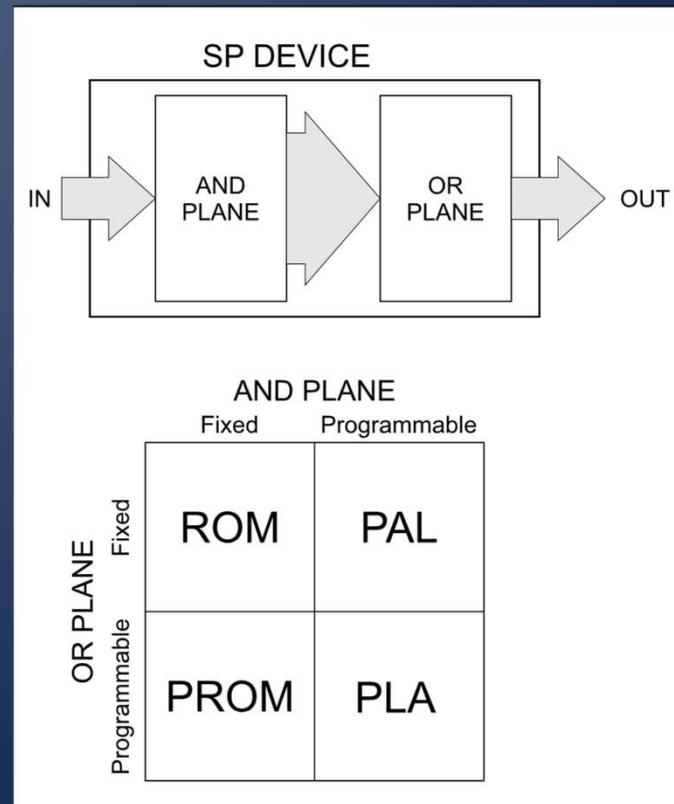
Come il nome stesso suggerisce, si tratta di Gate Array programmabili. Ne esistono di tre tipi:

- Antifuse: le connessioni fra metalizzazioni sono effettuate usando un minuscolo segmento di silicio amorfo (non cristallino) che normalmente esibisce una resistenza altissima ($> 10^6 \Omega$). Appena si applicano tensioni e correnti appropriate, il silicio amorfo si converte in silicio policristallino conduttivo.
- RAM statica: una volta caricata un configurazione, i bit della RAM pilotano alcuni transistor che fungono da interruttori che collegano le metalizzazioni e per configurare le LUT (LookUp Tables).
- FLASH/EPROM: alla stregua delle versioni con RAM statica, i bit della memoria non volatile si usano per attivare le connessioni fra le metalizzazioni e per configurare le LUT.



DISPOSITIVI CLASSICI CHE IMPLAMENTANO LE FORME CANONICHE SP (SSI)

- ROM: Read Only Memory
Hanno un pattern di output prestabilito dal costruttore.
- PROM: Programmable Read Only Memory
Solo il piano di OR è programmabile.
- PAL: Programmable Array Logic
Il piano di AND è programmabile e il piano di OR è prestabilito dal costruttore. Implementano solo le forme SP (somme di prodotti o somme di minterm).
- PLA: Programmable Logic Array
Entrambi i piani (AND e OR) sono programmabili.



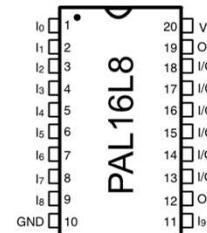
ESEMPIO CON PAL16L8 (SSI)

Le PAL (Programmable Array Logic) sono dispositivi logici programmabili ormai obsoleti che furono inventati da John Birkner e introdotti da MMI nel lontano 1978.

Il dispositivo PAL16L8 è un dispositivo per logica combinatoria (o sequenziale asincrono). Storicamente realizzato in tecnologia bipolare (con transistor BJT), fu un dispositivo di enorme successo, utilizzato dai primi costruttori di mini-computer (DEC, Data General, etc.) per ottimizzare i loro sistemi.

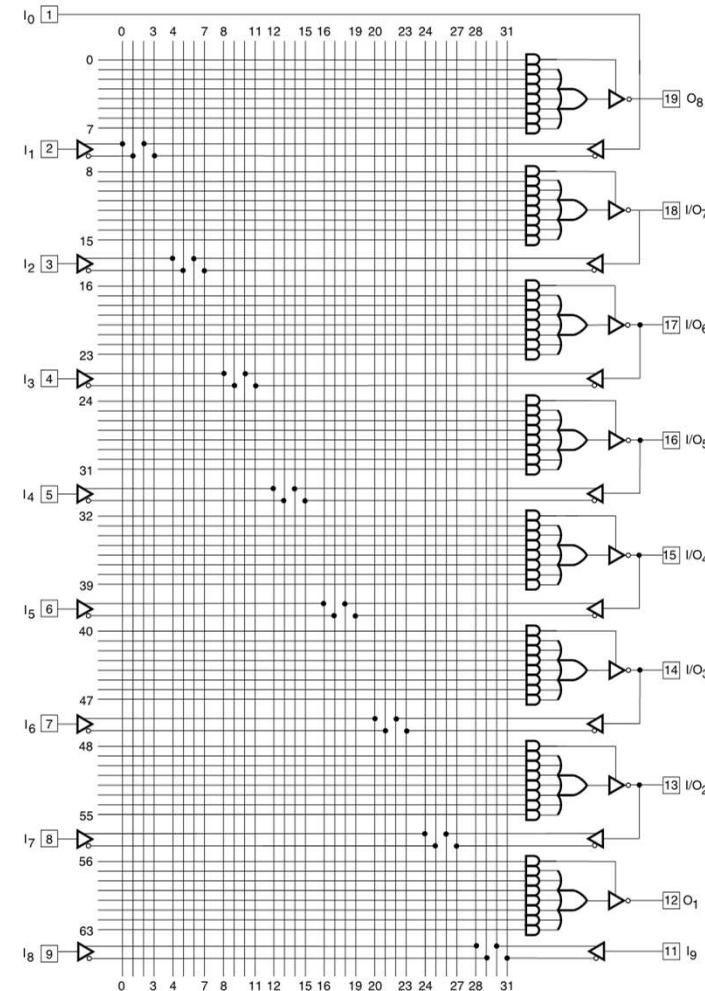
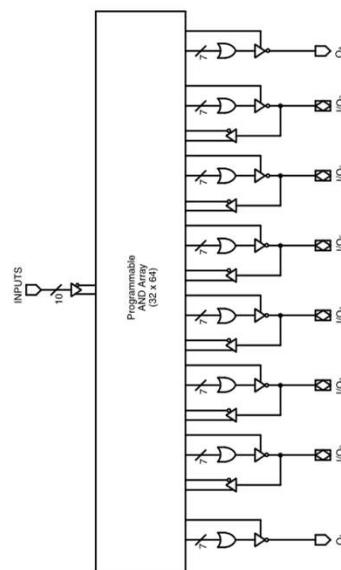


ESEMPIO DI LOGICA PROGRAMMABILE PAL

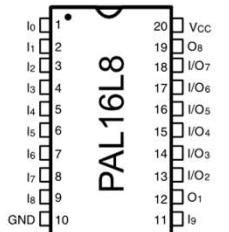


PIN DESIGNATIONS

V _{CC}	= Supply Voltage
GND	= Ground
I	= Input
I/O	= Input/Output
O	= Output

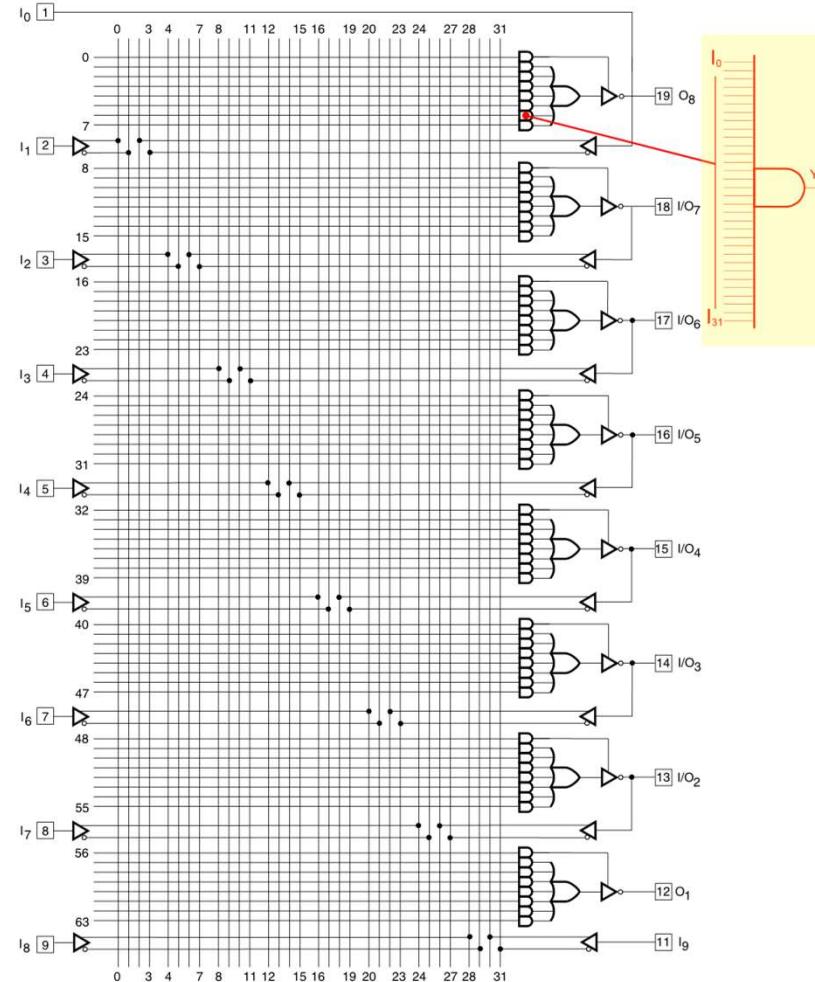
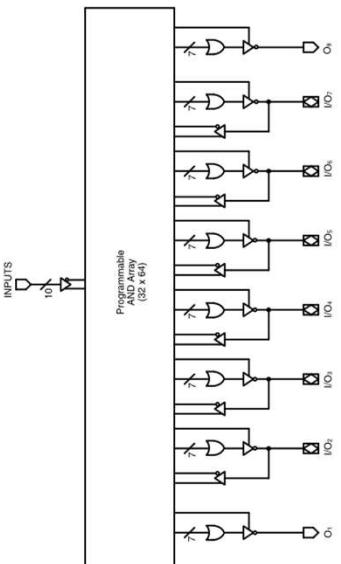


ESEMPIO DI LOGICA PROGRAMMABILE PAL



PIN DESIGNATIONS

V _{CC}	=	Supply Voltage
GND	=	Ground
I	=	Input
I/O	=	Input/Output
O	=	Output



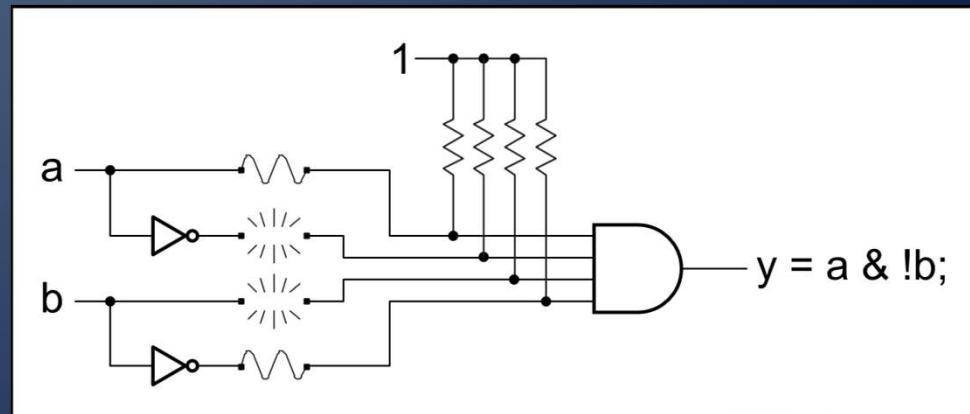
PROGRAMMAZIONE DELLE SPLD OTP

- Nei dispositivi quali PAL, PLA e PROM la programmazione è effettuata interrompendo dei minuscoli fusibili.

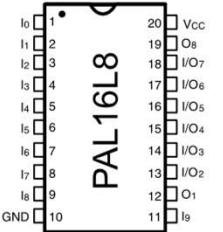
Nello specifico dei dispositivi PAL, in serie ad ogni segnale ed al suo negato sul percorso che raggiunge la porta AND, ci sono dei fusibili. Inizialmente i fusibili sono tutti integri.

Quando un fusibile, a seguito della programmazione viene interrotto, il segnale non può più raggiungere la porta AND. Perciò, l'ingresso è collegato ad 1 tramite una resistenza detta di pull-up.

Di fatto, ciò vuole dire che l'uscita dell'AND dipenderà esclusivamente dagli ingressi in cui fusibili sono rimasti integri. Nell'esempio, si può notare che solo a e \bar{b} possono costituire il termine y .

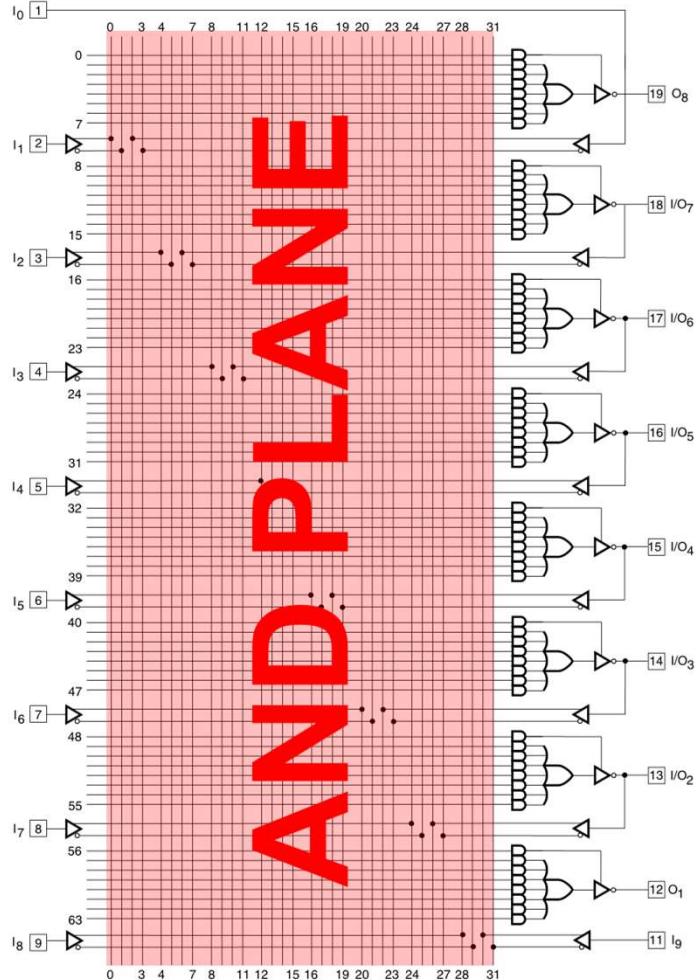
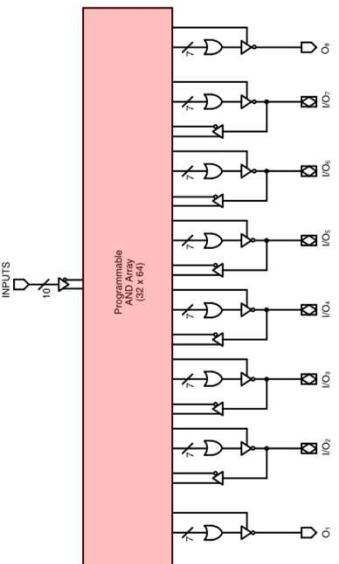


ESEMPIO DI LOGICA PROGRAMMABILE PAL



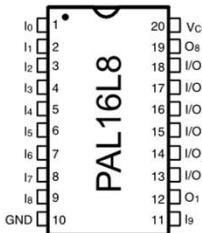
PIN DESIGNATIONS

V_{CC} = Supply Voltage
GND = Ground
I = Input
I/O = Input/Output
O = Output



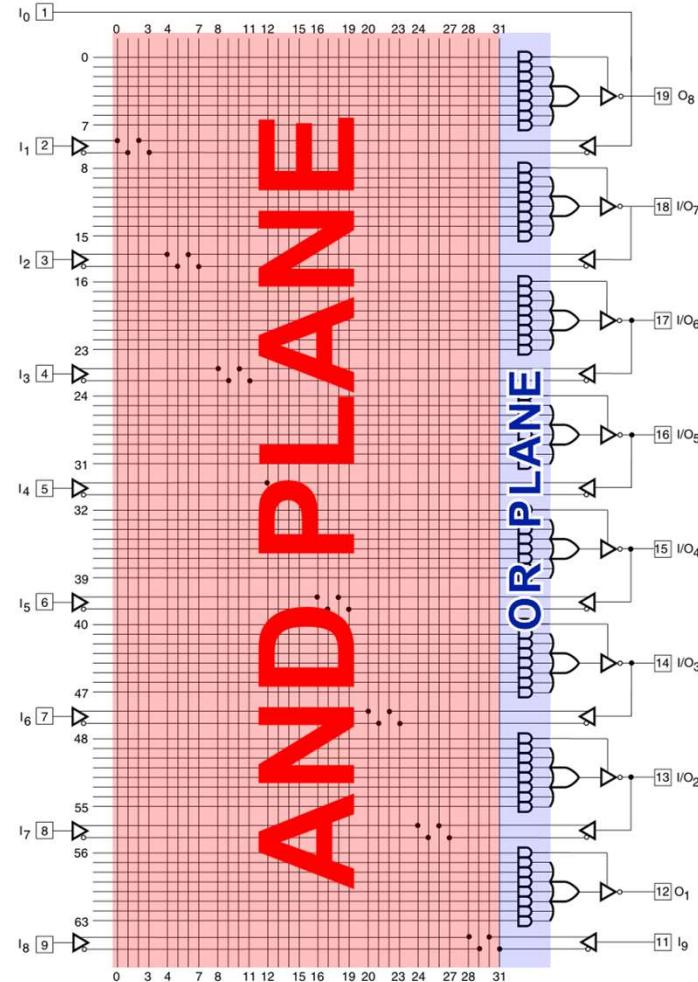
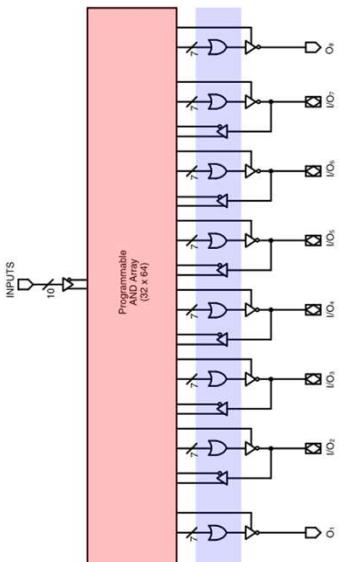
AND PLANE

ESEMPIO DI LOGICA PROGRAMMABILE PAL

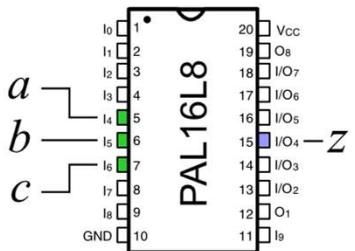


PIN DESIGNATIONS

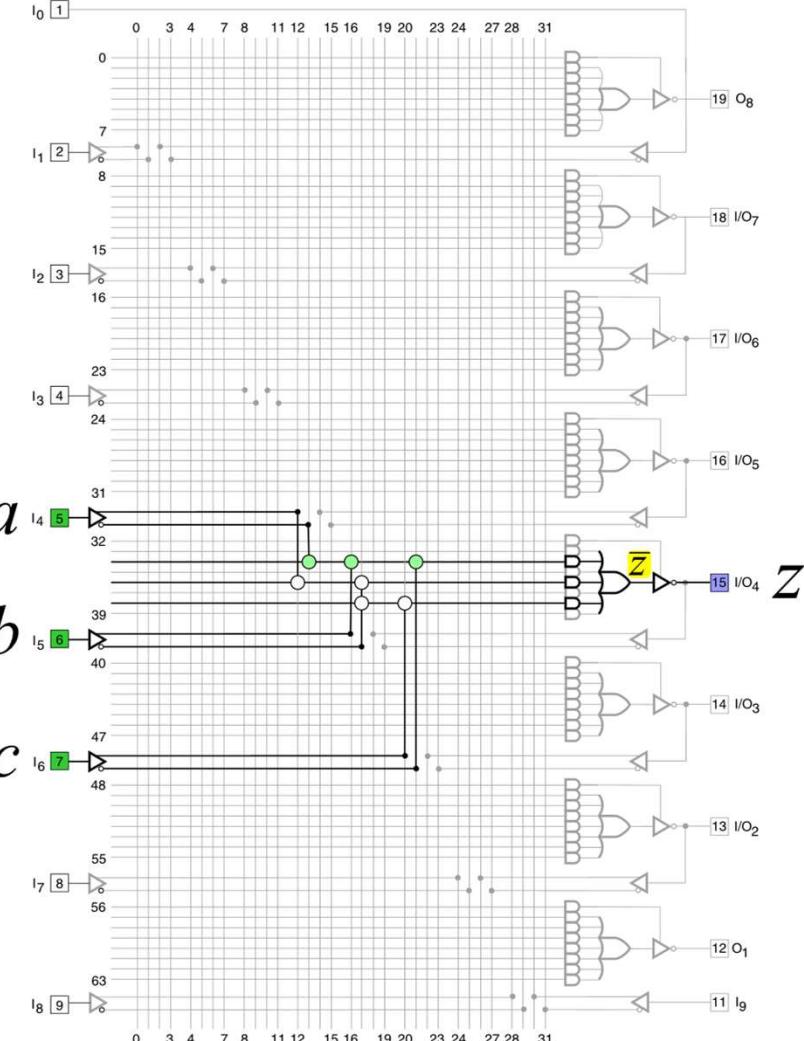
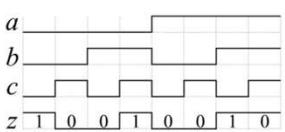
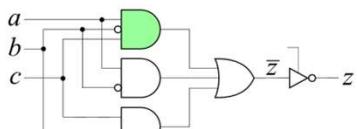
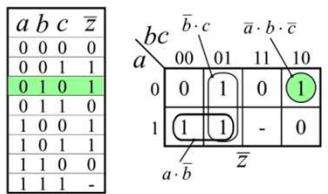
V_{CC} = Supply Voltage
GND = Ground
I = Input
I/O = Input/Output
O = Output



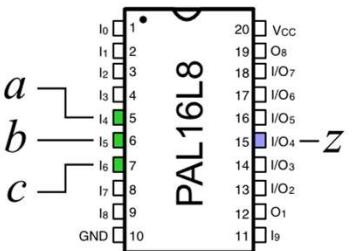
ESEMPIO DI LOGICA PROGRAMMABILE PAL



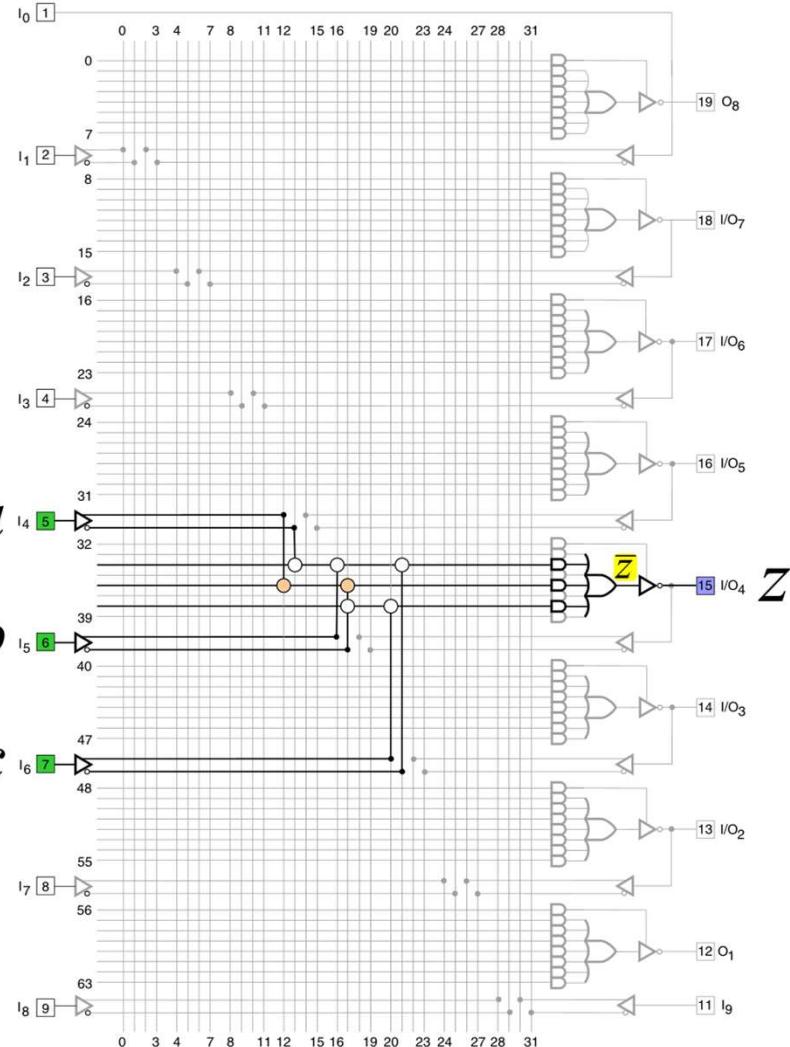
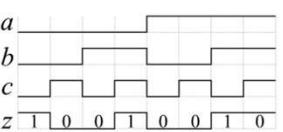
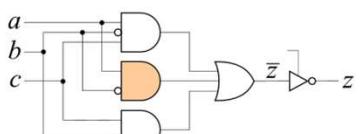
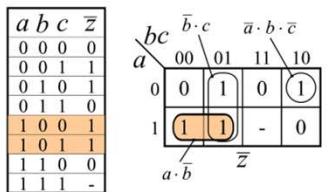
$$\bar{z} = \overline{a} \cdot b \cdot \overline{c} + a \cdot \overline{b} + \overline{b} \cdot c$$



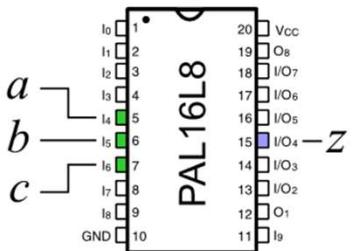
ESEMPIO DI LOGICA PROGRAMMABILE PAL



$$\bar{z} = \bar{a} \cdot b \cdot \bar{c} + a \cdot \bar{b} + \bar{b} \cdot c$$



ESEMPIO DI LOGICA PROGRAMMABILE PAL



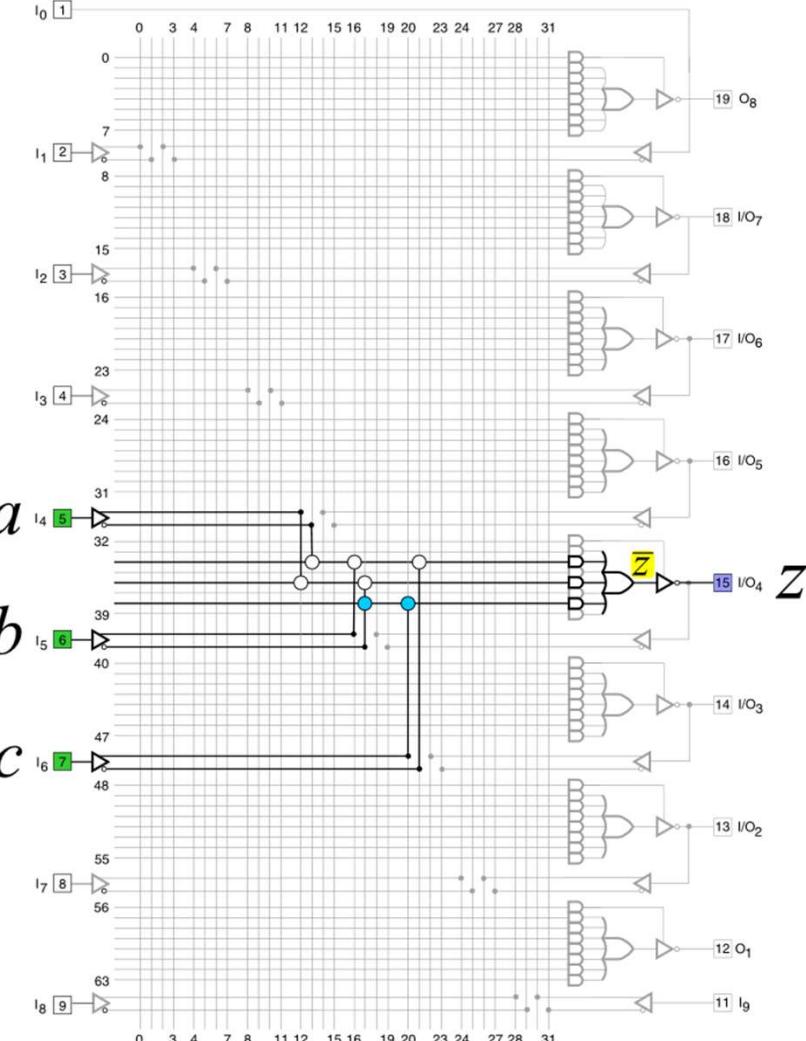
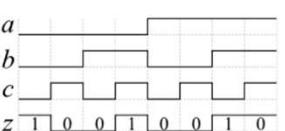
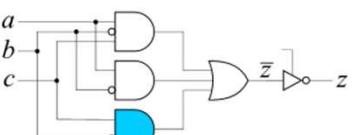
$$\bar{z} = \bar{a} \cdot b \cdot \bar{c} + a \cdot \bar{b} + \bar{b} \cdot c$$

a	b	c	\bar{z}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	-

Truth table for \bar{z} based on inputs a , b , c :

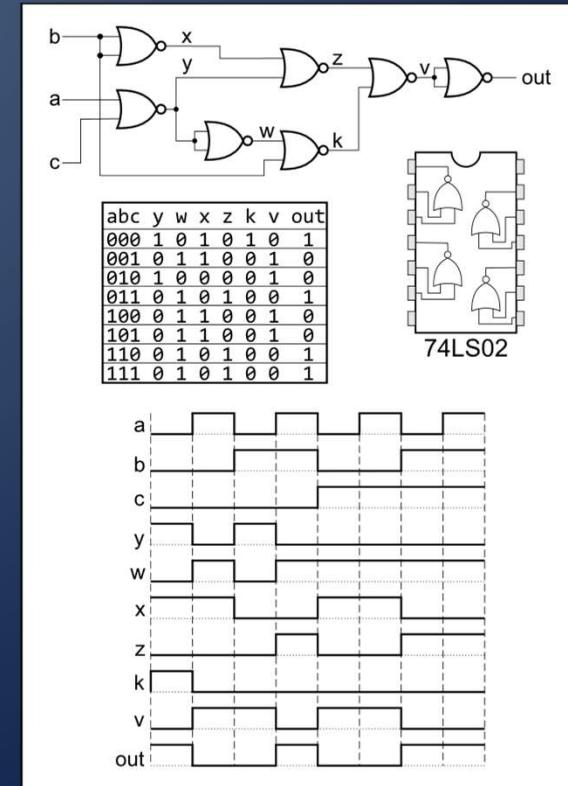
Inputs: a , b , c | Output: \bar{z}

a	b	c	$\bar{b} \cdot c$	$\bar{a} \cdot b \cdot \bar{c}$	$a \cdot \bar{b}$	\bar{z}
0	0	0	0	0	0	0
0	0	1	1	0	0	1
0	1	0	0	0	0	0
0	1	1	1	0	0	1
1	0	0	0	1	-	0
1	0	1	0	0	1	1
1	1	0	0	0	0	0
1	1	1	-	-	-	-



RETI LOGICHE COMBINATORIE MULTILIVELLO

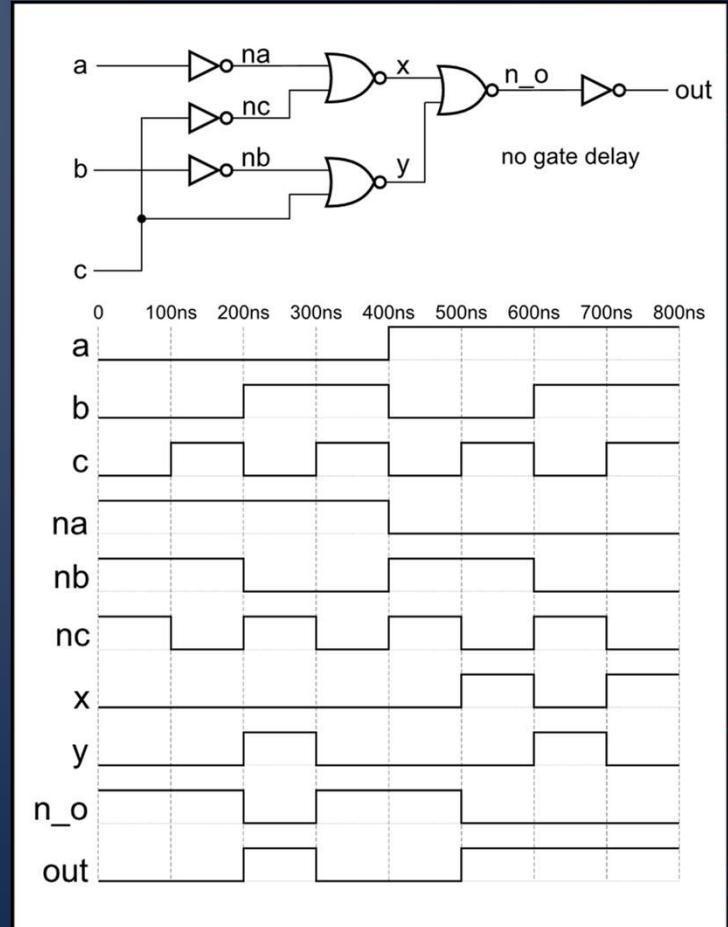
- Possono sfruttare semplici porte logiche o forme canoniche collegate in cascata. Il tempo di stabilizzazione delle uscite al variare degli ingressi può essere maggiore ed esibire meno coerenza.
- Le reti multilivello sono tipiche dei dispositivi programmabili moderni (CPLD e FPGA); collegano macro-celle/LUT (Look Up Table, blocchi logici predefiniti) fra loro per realizzare strutture logiche anche molto complesse.



ALEE (HAZARDS)

A lato si può vedere una rete multilivello che produce l'uscita *out* per gli ingressi *a*, *b* e *c*. Siccome la rete è realizzata con un solo tipo di componente, per ottenere la funzione *out* è necessario mettere in cascata più porte rispetto ad una rete in forma canonica, ottenendo di fatto una rete multilivello.

Dal momento che i componenti non esibiscono ritardi di propagazione, il comportamenti di questa rete è ideale. Vale a dire che non ci sono «incertezze» nella evoluzione dei segnali nel tempo.

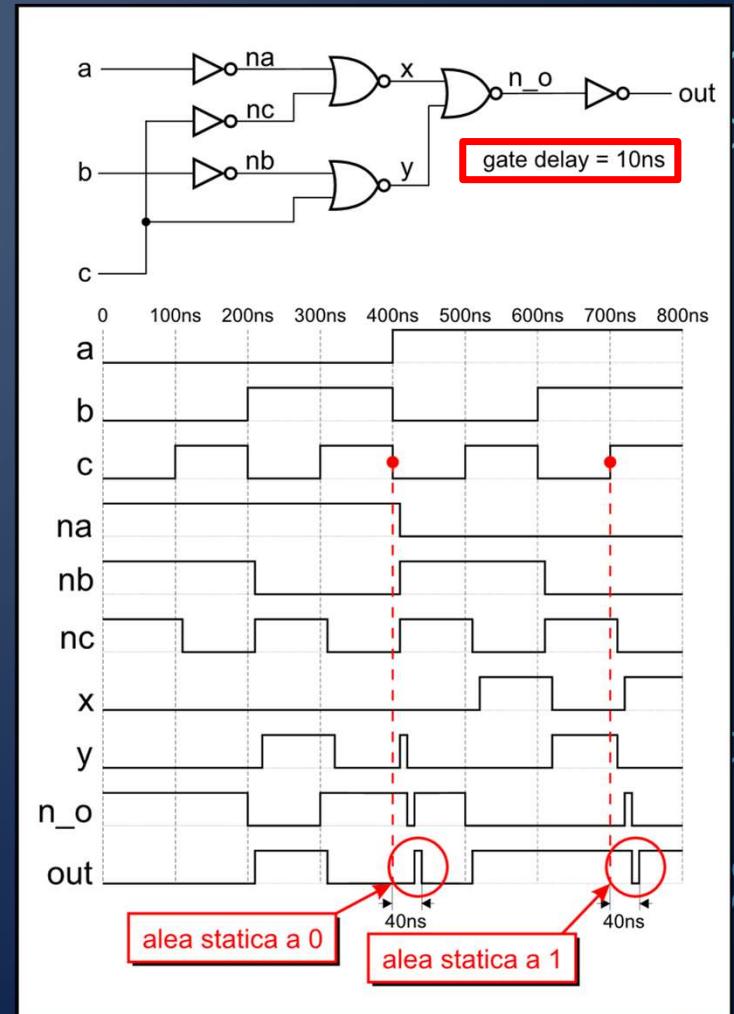


ALEE STATICHE (STATIC HAZARDS)

Quando i tempi di propagazione delle porte o dei percorsi fisici (collegamenti elettrici) non è trascurabile, il circuito che produce *out* può esibire incertezze prima di assestarsi su un valore nominale.

Nella figura a lato si possono notare degli impulsi molto brevi che prendono il nome di alee. Si tratta di transitori che perturbano il segnale per un brevissimo periodo. Bisogna tenerne conto in fase di progettazione, in quanto possono causare malfunzionamenti nelle reti logiche. Le alee statiche non sono appannaggio delle sole reti multilivello. Anche le forme canoniche con componenti non ideali o percorsi «lenti» possono esibire incertezze.

Nell'esempio pratico a lato, sono state usate (anche per gli inverter) le porte NOR di due SN74LS02, che esibiscono un ritardo tipico di 10 ns: si noti che i ritardi di propagazione non sono proprio trascurabili rispetto alla velocità dei segnali applicati (fronti ogni 100 ns).

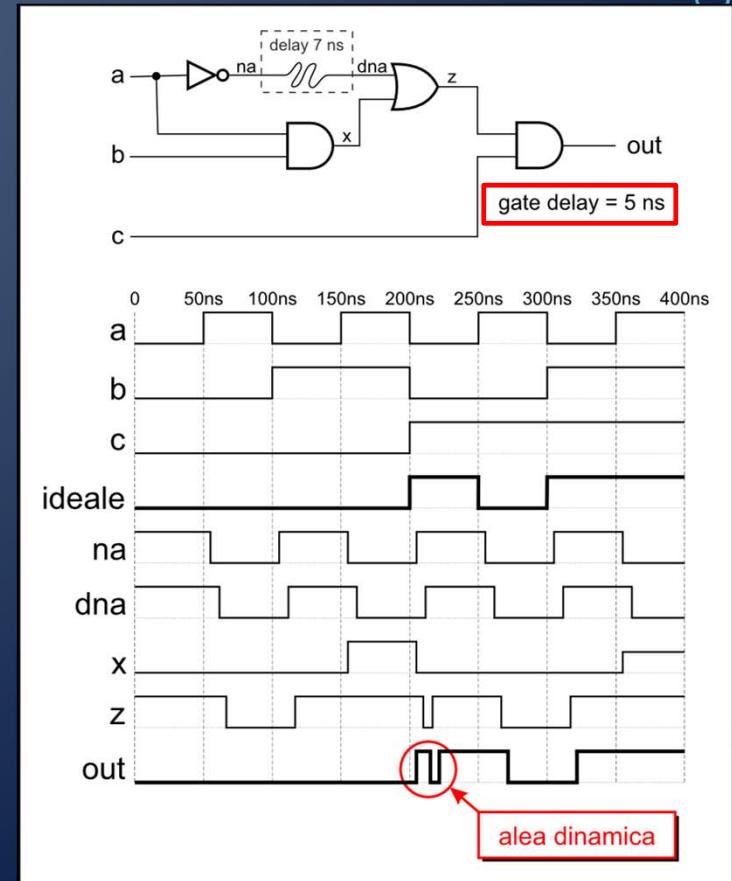


ALEE DINAMICHE (DYNAMIC HAZARDS)

Il circuito a lato esibisce un secondo tipo di alea che produce un'uscita che si porta inizialmente sul valore previsto (da 0 a 1), ma non rimane stabile: infatti segue immediatamente un impulso col valore opposto (zero), per poi assestarsi finalmente sul valore desiderato (uno).

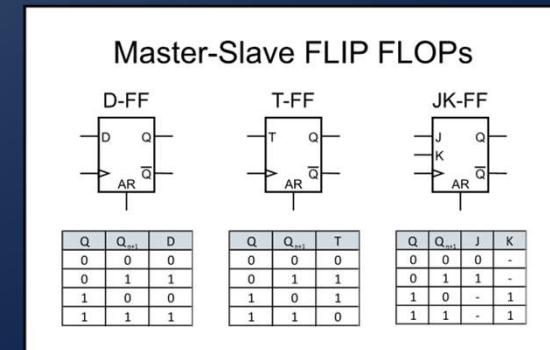
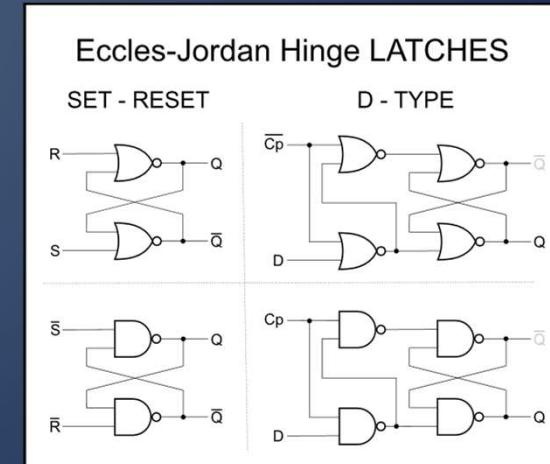
Questo comportamento circuitale è definito come «alea dinamica». Le alee dinamiche compaiono quando, a seguito della variazione di uno o più segnali di ingresso, il segnale in uscita ha una transizione al valore opposto, solo che questa non si stabilizza repentinamente ma può oscillare una o più volte.

La presenza di alee dinamiche comporta notevoli problemi nella progettazione e realizzazione delle cosiddette «reti sequenziali asincrone» con più di due stati. Questo genere di reti non saranno trattate in questa presentazione, anche se vedremo un esempio di una semplicissima rete sequenziale asincrona a due stati più avanti.



GLI ELEMENTI DI MEMORIA

- Alcune configurazioni circuitali costituiscono gli elementi di memoria. Tali elementi sono il cuore delle reti sequenziali.
- Gli elementi di memoria comunemente usati possono essere sensibili ai «livelli» (riquadro in alto) o ai «fronti» (riquadro in basso).
- Gli elementi di memoria sensibili ai «fronti» sono spesso usati con un segnale con cadenza specifica che determina il ritmo di elaborazione (clock). Sono principalmente di tre tipi, D, T e JK e il loro funzionamento è descritto dalle cosiddette «tabelle di eccitazione».
- Sono sempre dotati di un feedback interno a uno o più livelli. Si possono studiare come reti combinatorie una volta interrotti i feedback e considerati come normali ingressi.
- Le reti combinatorie viste in precedenza, «alimentano» le variabili di ingresso di questi elementi di memoria per costruire sistemi di reti sequenziali più grandi e più complessi.



LOGICA TERNARIA O TRIVALENTE (3VL)

- La logica trivalente è una estensione della logica sentenziale binaria che abbiamo visto finora.
- È utile nell'analisi e nella simulazione di circuiti digitali binari con *feedback*, cioè con le uscite della rete che tornano in ingresso.
- I valori rappresentati sono tre: 0, 1 e X.
- Una funzione in logica trivalente $z = f(x_1, \dots, x_n)$ è una legge $\{0,1,X\}^n \rightarrow \{0,1,X\}$ che mappa z per ogni combinazione di x_1, \dots, x_n .
- Compare come rappresentazione di «*indefinito*» nei simulatori.

OPERATORI NOT, OR E AND IN LOGICA TRIVALENT

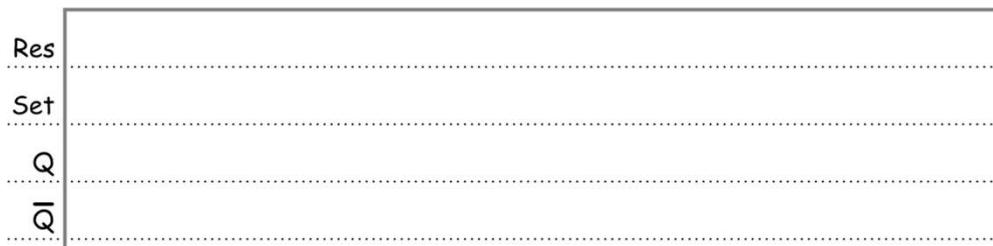
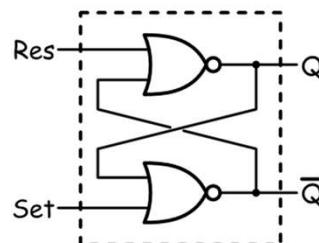
x	NOT		OR		AND	
	$f_1(x)$	$x_1 x_2$	$f_2(x_1, x_2)$	$x_1 x_2$	$f_3(x_1, x_2)$	
0	1	0 0	0	0 0	0	
1	0	0 1	1	0 1	0	
X	X	0 X	X	0 X	0	
		1 0	1	1 0	0	
		1 1	1	1 1	1	
		1 X	1	1 X	X	
		X 0	X	X 0	0	
		X 1	1	X 1	X	
		X X	X	X X	X	

SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

La logica trivale è utile per studiare il comportamento di reti logiche asincrone.

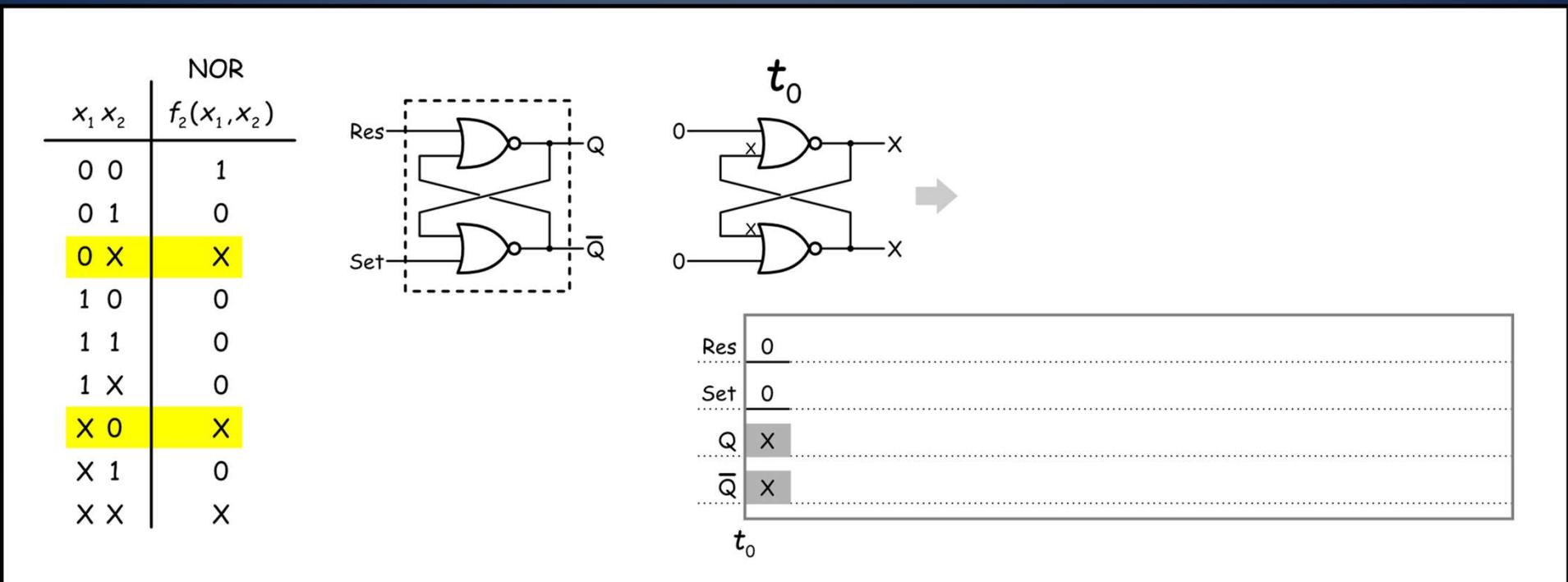
Come esempio vediamo lo studio di un *hinge latch Set-Reset* fatto con porte NOR.

NOR	
$x_1 x_2$	$f_2(x_1, x_2)$
0 0	1
0 1	0
0 X	X
1 0	0
1 1	0
1 X	0
X 0	X
X 1	0
XX	X



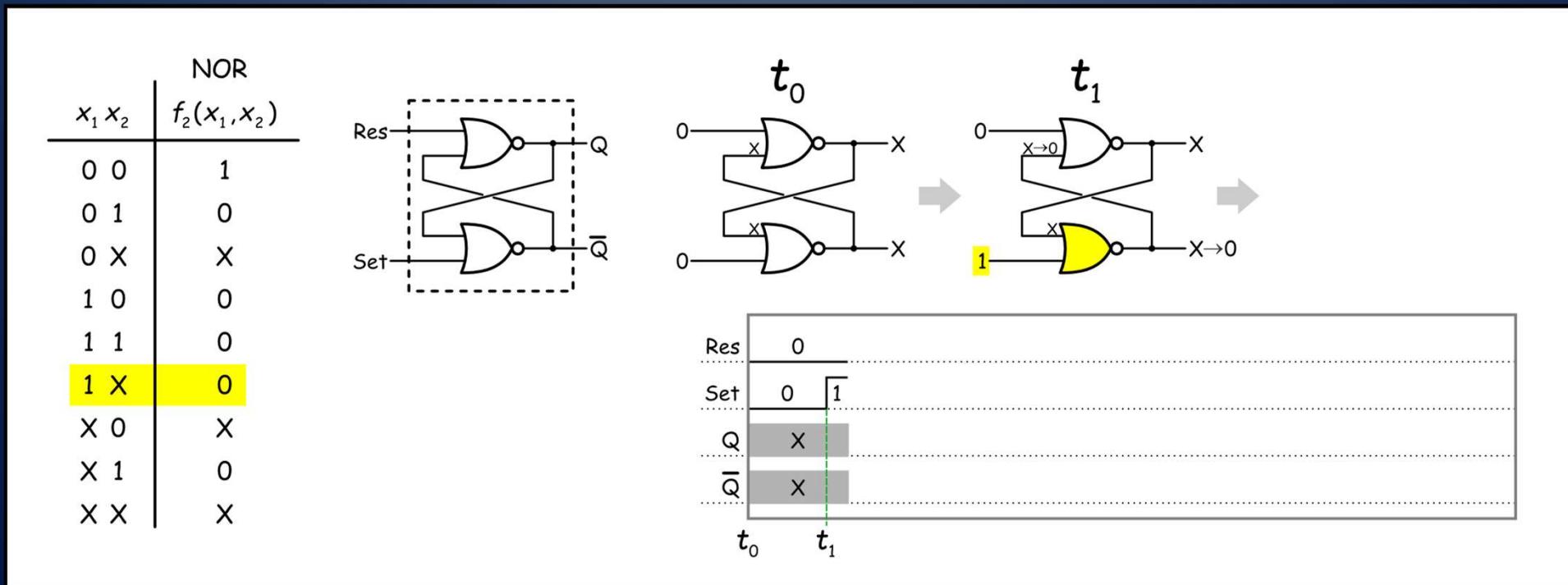
SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

Inizialmente (istante t_0) lo stato d'uscita delle porte NOR è sconosciuto in quanto gli ingressi a 0 (zero) ammettono valori X (indefiniti) come uscite. In pratica, non è possibile determinare con certezza lo stato del Latch S-R. Si può affermare però che l'uscita Q e \bar{Q} siano una la negazione dell'altra in condizione di stabilità.



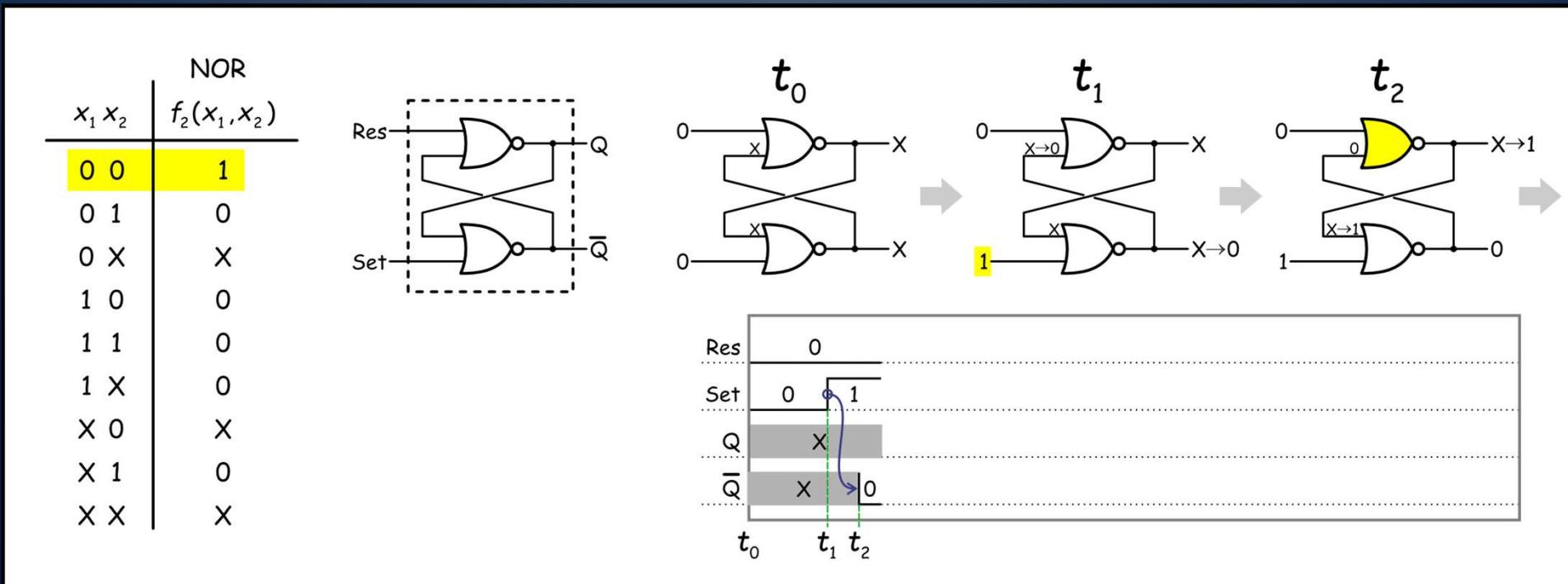
SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

Appena l'ingresso di Set è portato ad 1 (istante t_1), la porta ad esso collegato «reagisce» preparandosi a produrre uno 0 indipendentemente dal fatto che il secondo ingresso veda X.



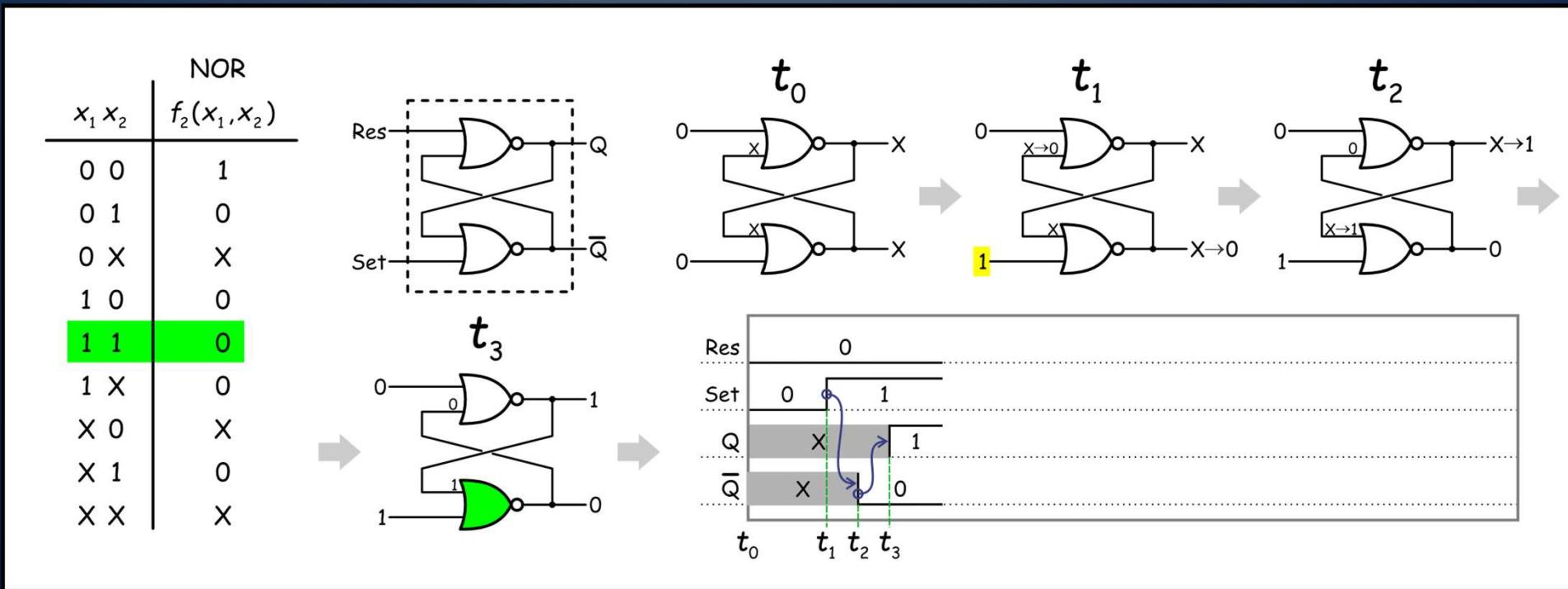
SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

All'istante t_2 , la porta NOR collegata all'uscita \bar{Q} genera stabilmente il valore zero. Comunque, la variazione dell'uscita innesca l'altra porta NOR che, vedendo al configuraione 00 al proprio ingresso si prepara a generare il valore 1 per l'uscita Q .



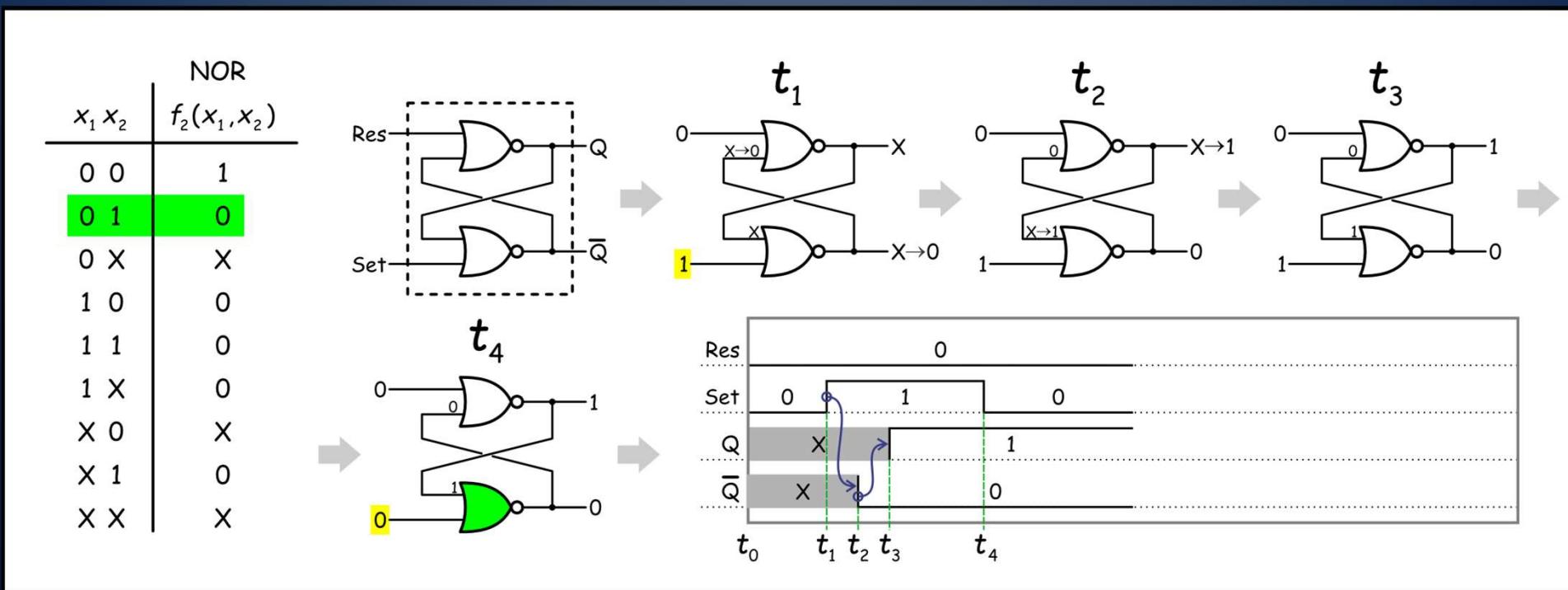
SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

L'uscita \bar{Q} conferma il valore 0 (istante t_3) rafforzato dalla configurazione 11 all'ingresso del NOR che la riguarda.



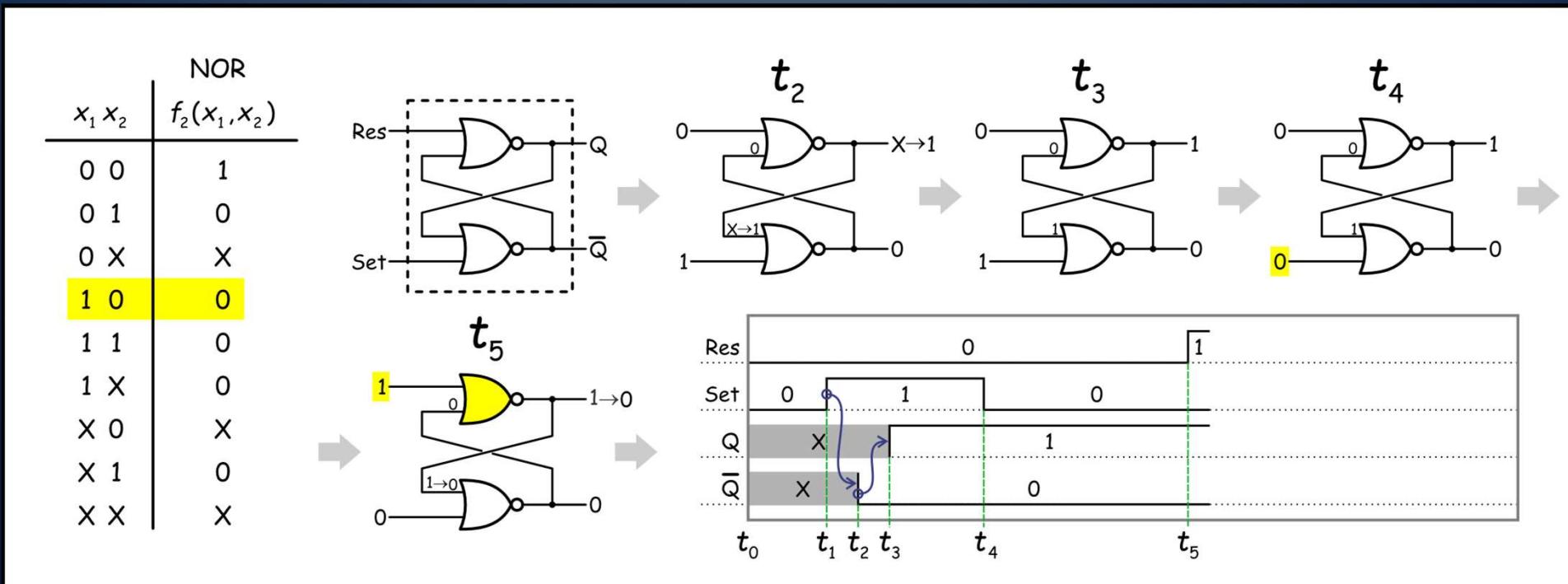
SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

De-asserendo il comando Set (ponendolo nuovamente a zero all'istante t_4), si nota che il latch, come previsto, mantiene il valore precedentemente imposto.



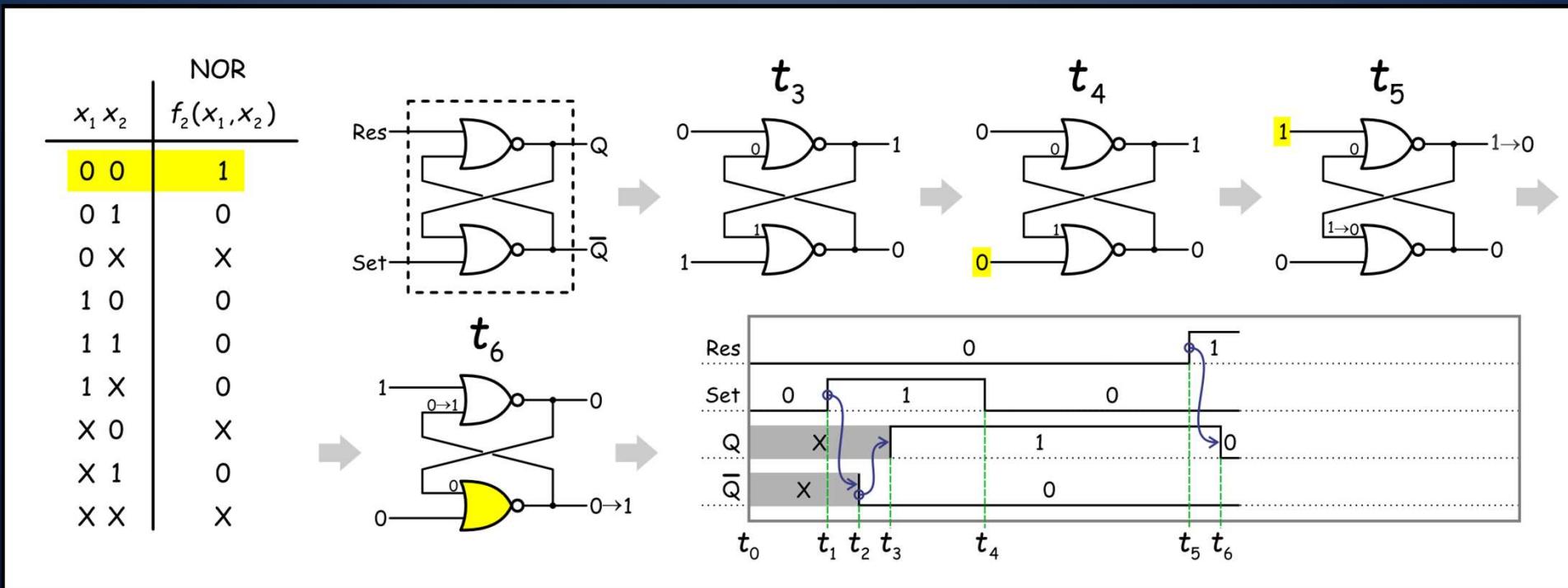
SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

All'istante t_5 si porta ad 1 l'ingresso Res (Reset) del latch. La porta collegata all'uscita Q , subito innesca un cambiamento in uscita dato dalla configurazione 10 presente sugli ingressi.



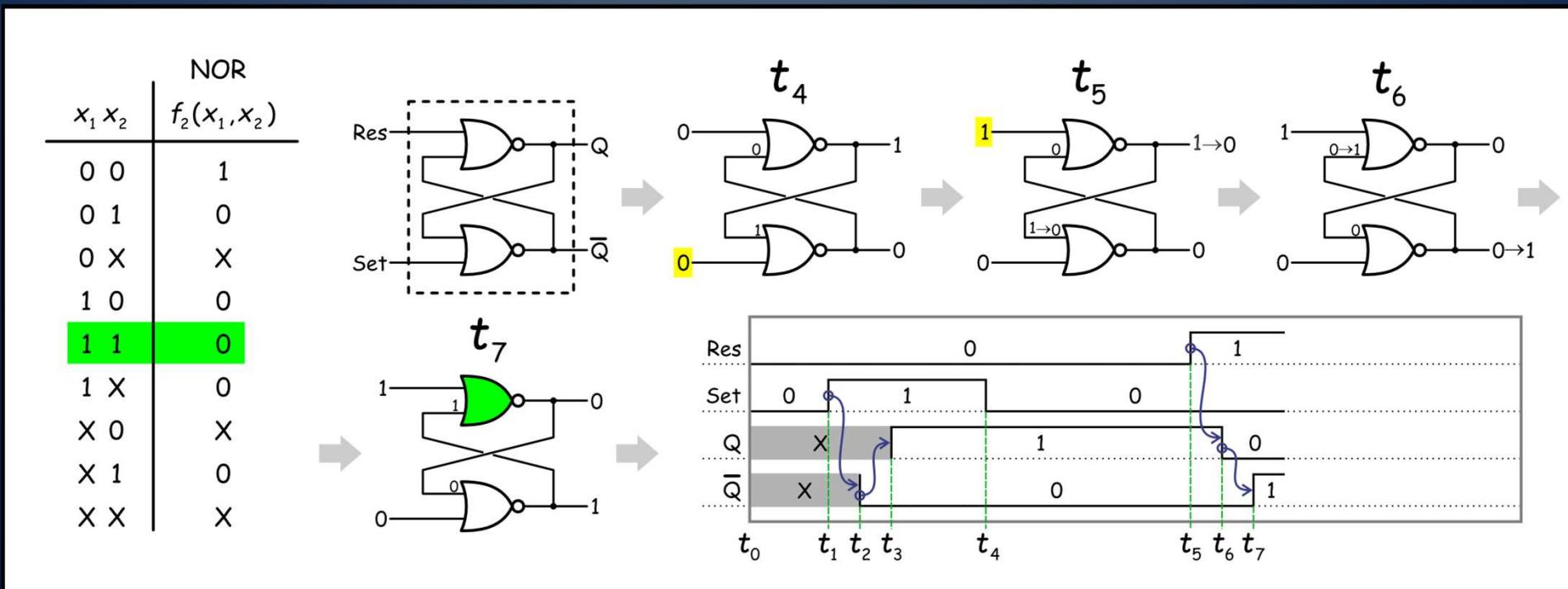
SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

All'istante t_6 l'uscita Q del latch si porta immediatamente a zero. Questo evento comporta l'innesto del prossimo cambiamento ad 1 dell'uscita relativa a \bar{Q} .



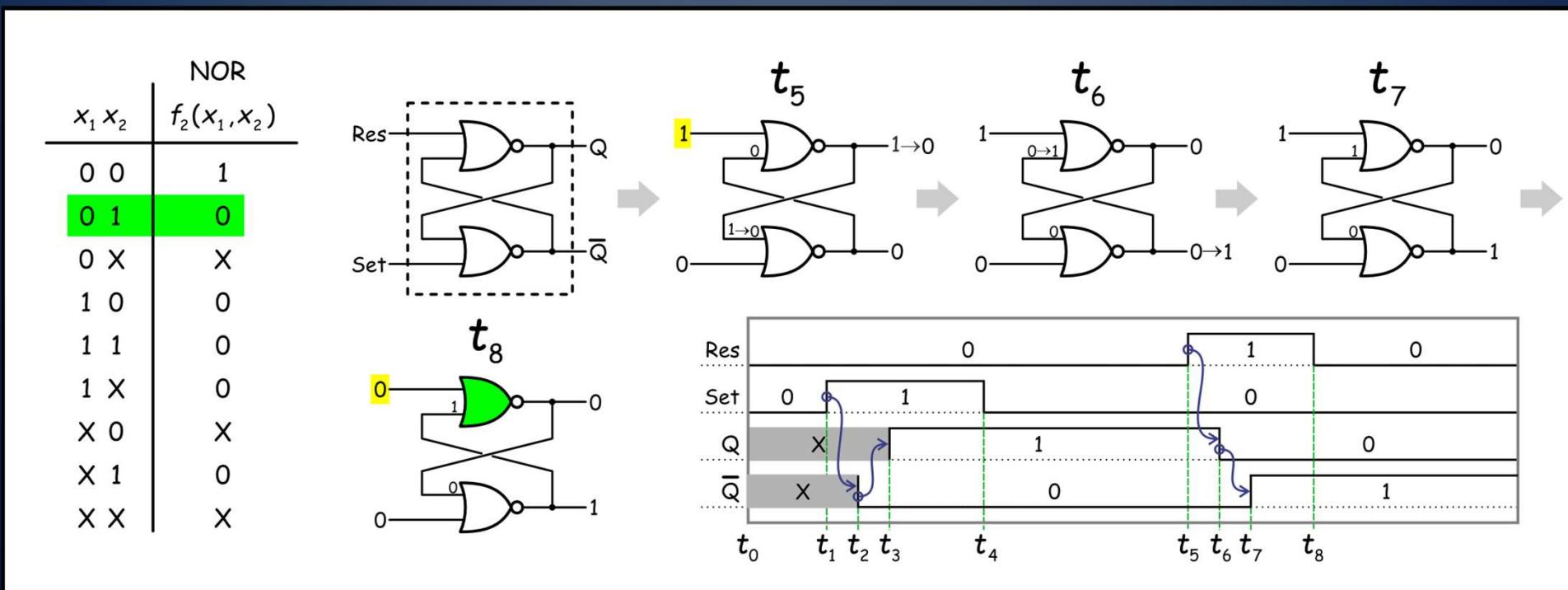
SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

Come previsto, poco dopo l'uscita \bar{Q} si stabilizza ad 1 all'istante t_7 .

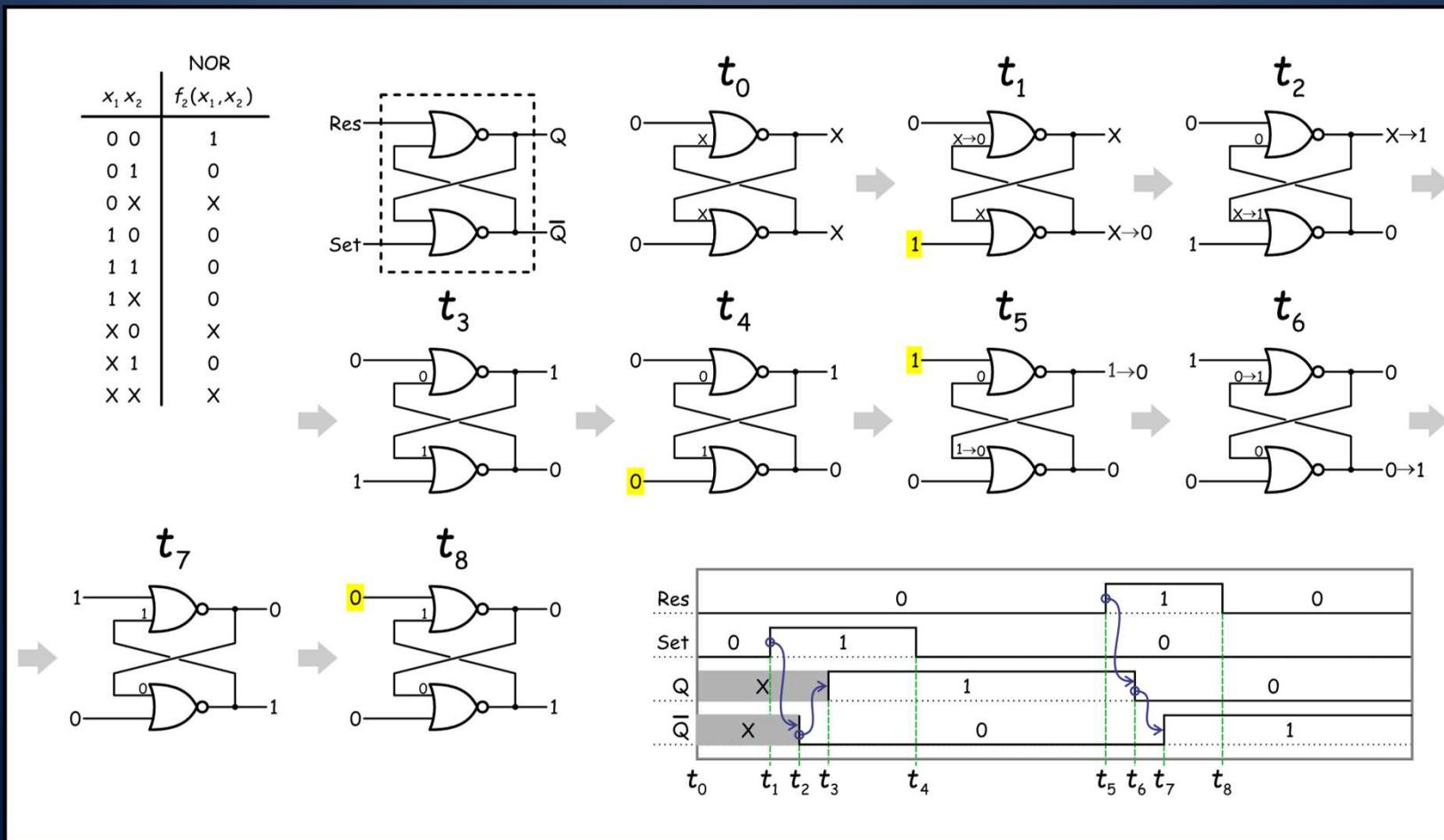


SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE

Togliendo il comando Res, la configurazione 01 all'ingresso del NOR dell'uscita Q conferma il valore zero. Il latch ora ricorderà lo stato fino al prossimo Set.



SIMULAZIONE LATCH S-R CON LOGICA TRIVALENTE



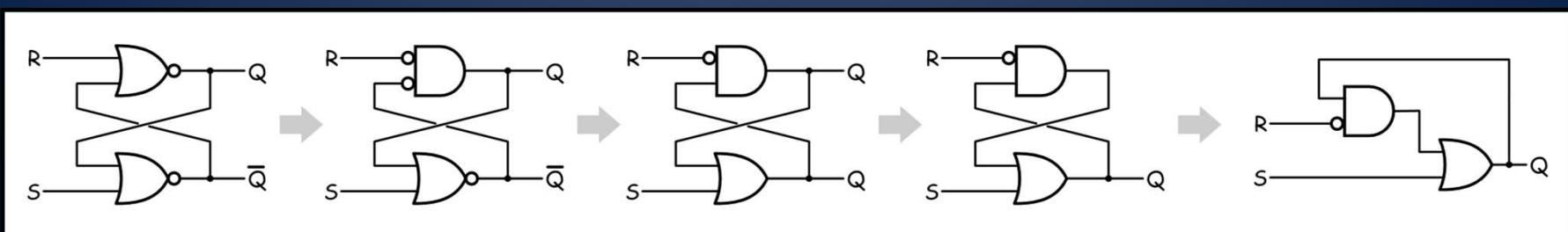
RETI LOGICHE SEQUENZIALI (1)

La simulazione del Latch S-R vista prima, suggerisce l'esistenza di reti logiche in grado di «ricordare» la fase di elaborazione, normalmente conosciuta anche come stato interno.

Sempre osservando l'esperimento precedente, si può dedurre il fatto che riportare i segnali d'uscita in ingresso possa essere la chiave per dare ad una rete logica la capacità di possedere memoria.

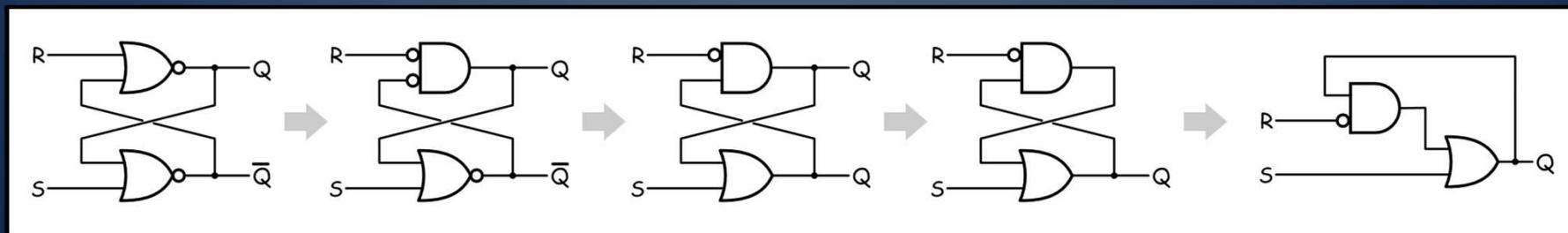
Con una semplice manipolazione possiamo vedere che il Latch S-R basato su NOR che abbiamo visto in precedenza può essere definito anche come una rete combinatoria in forma canonica SP che legge la propria uscita (consideriamo solo l'uscita Q).

Si noti che la forma al centro nel disegno sottostante suggerisce che, a seconda di quale delle due uscite (Q) si vuole usare come feedback, è possibile optare indifferentemente per una delle due forme canoniche (SP se si sceglie l'OR o PS se si sceglie l'AND).

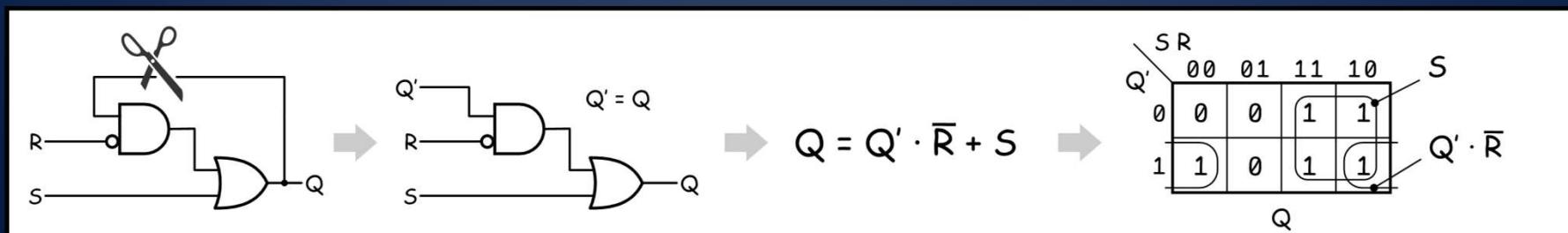


RETI LOGICHE SEQUENZIALI (2)

L'ultima forma a destra evidenzia il fatto che l'uscita Q è riportata in ingresso.



Ponendo la relazione $Q' = Q$, è possibile aprire l'anello ed ottenere una rete combinatoria pura dalla quale si ricava l'espressione booleana $Q = Q' \cdot \bar{R} + S$.



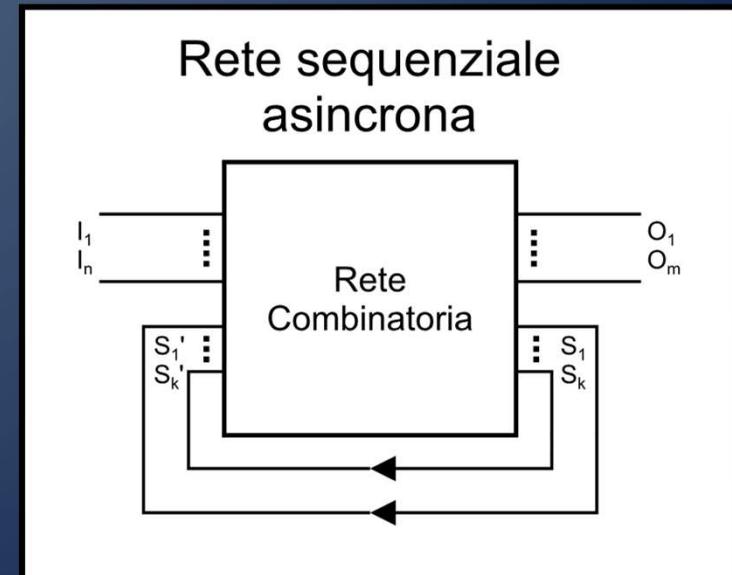
MODELLO RETI LOGICHE SEQUENZIALI ASINCRONE

Nella forma canonica SP del Latch S-R, possiamo dire che l'uscita Q rappresenta anche lo stato (o meglio, è la variabile di stato) della rete sequenziale.

La categoria delle reti sequenziali che riportano direttamente la variabile di stato in ingresso è detta «asincrona».

La variazione delle uscite si ripercuote immediatamente sugli ingressi; se la variabile di stato è costituita da più espressioni (cioè è un vettore di più variabili binarie correlate), questa categoria di reti logiche diventa complicata da gestire.

A causa delle alee (soprattutto dinamiche) menzionate precedentemente, ci possono essere codifiche errate della variabile di stato, con conseguente errore o instabilità del sistema. Le tecniche per la mitigazione di tali problemi sono ben oltre lo scopo di questa presentazione.



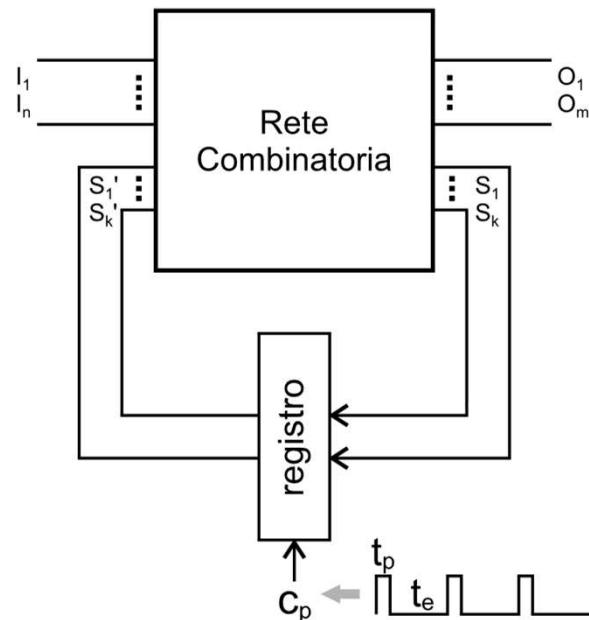
LE RETI SEQUENZIALI SINCRONE (1)

Per cercare di risolvere il problema e le criticità tipiche delle reti sequenziali asincrone, si può provare a mettere un «registro a controllo impulsivo» sugli anelli. Un registro a controllo impulsivo può essere costituito da una schiera di Latch di tipo D (già visti in precedenza) con gli ingressi di abilitazione C_p collegati in comune. Applicando un impulso agli ingressi di abilitazione dei latch è possibile richiedere al registro di «tracciare», memorizzandolo continuamente, lo stato interno della rete sequenziale. Quando l'impulso non è presente il registro presenta l'ultima configurazione memorizzata. Così facendo si evita di riportare continuamente lo stato d'uscita sugli ingressi. Di fatto si ottiene una elaborazione «sincronizzata» con la cadenza degli impulsi.

Questo modello sincrono comunque non è privo di problemi: ci si deve assicurare che il tempo t_e sia abbastanza lungo da garantire un assestamento della rete combinatoria; allo stesso tempo bisogna che l'impulso t_p sia lungo a sufficienza perché il registro possa memorizzare lo stato, ma nello stesso tempo deve essere sufficientemente breve tanto da non creare un anello asincrono (un collegamento continuo).

La configurazione qui descritta, che utilizza i cosiddetti «Transparent Latch», è difficile da gestire in termini di affidabilità della rete.

Rete sequenziale sincrona
a controllo impulsivo

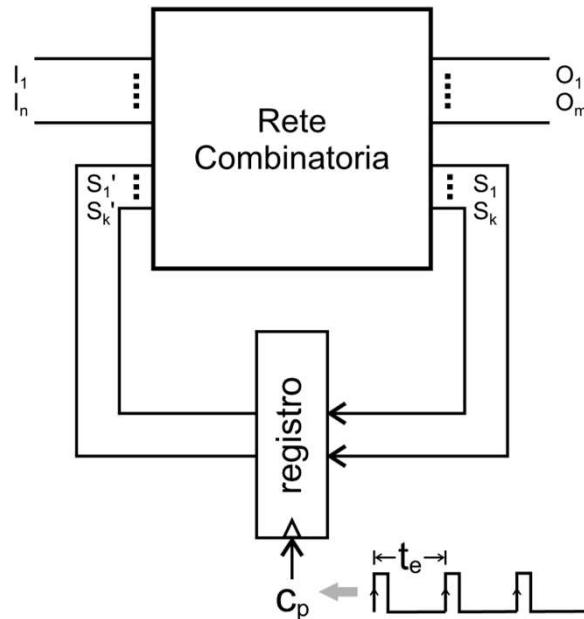


LE RETI SEQUENZIALI SINCRONE (2)

Per arrivare ad un modello di rete sincrona affidabile bisogna costruire i registri di stato facendo uso di Flip-Flop. Usando questo tipo di registri viene a meno il fabbisogno di controllare la larghezza dell'impulso di cadenza (detto anche «Clock»). Il registro fatto coi Flip-Flop memorizza le informazioni basandosi sui «cambiamenti» del segnale C_p e non in base al livello. In caso di fronte di salita, cioè da 0 ad 1 parliamo di «fronte positivo» (posedge). Se si considera il passaggio da 1 a 0 parliamo di «fronte negativo» (negedge). Un Flip-Flop reagisce a solo uno dei due fronti in base alla sua natura.

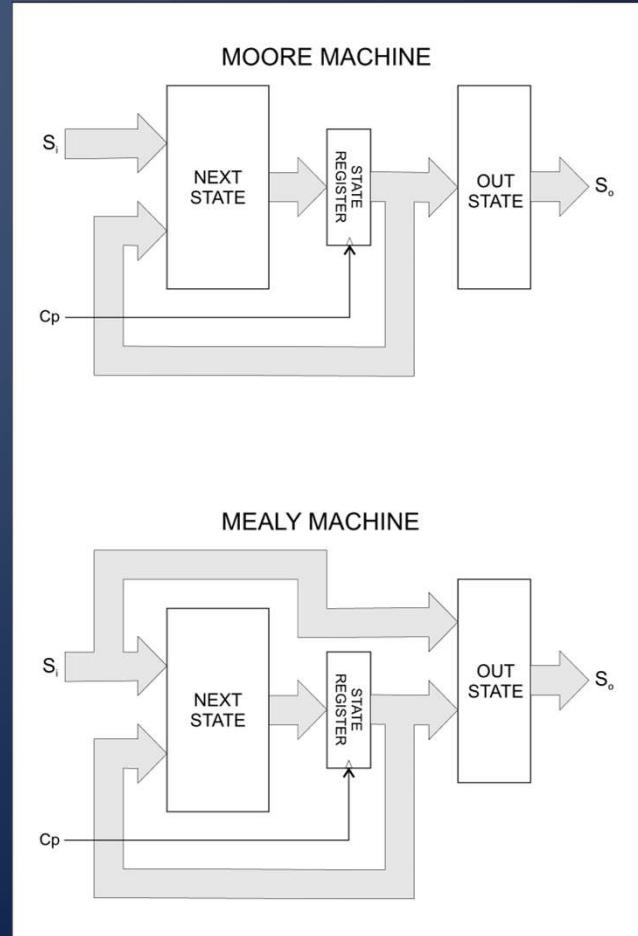
I Flip-Flop per questo tipo di applicazioni detti sono detti Master-Slave, in quanto sono costituiti da due latch in cascata che lavorano su livelli opposti del valore di C_p . Se il primo latch memorizza col livello 0, «tracciando» continuamente l'ingresso, il secondo latch presenta il valore memorizzato in uscita. Appena il segnale C_p passa da 0 ad 1, il primo latch smette di tracciare mantenendo l'ultimo valore mentre il secondo latch «traccia» il valore stabile del primo. Un «triangolino» all'ingresso C_p del registro denota la sensibilità al fronte positivo. Se è presente anche un cerchietto vuol dire che il registro è sensibile al fronte negativo.

Rete sequenziale sincrona con registri sensibili al fronte

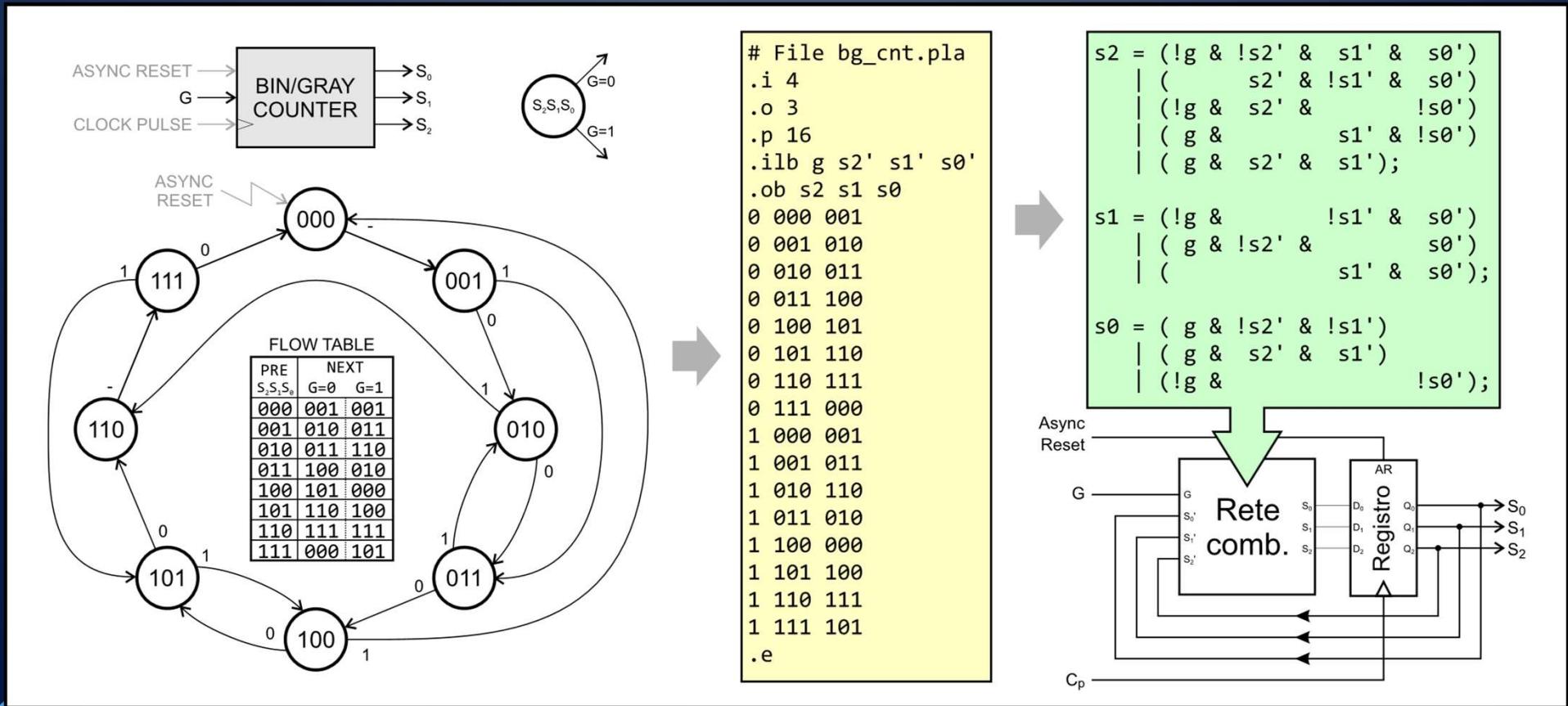


LE RETI SEQUENZIALI SINCRONE (3)

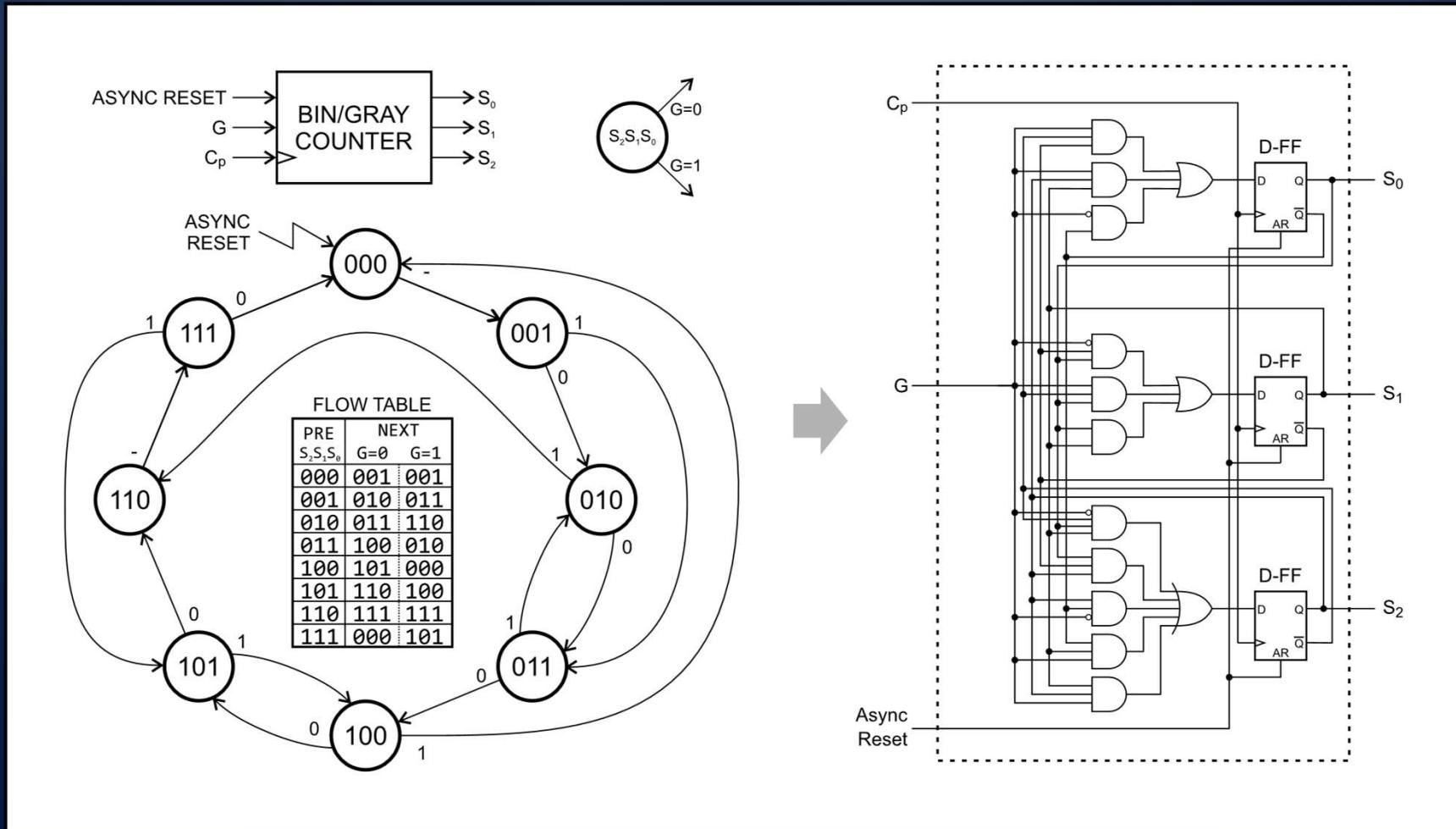
- Utilizzano una cadenza (clock) per l'elaborazione.
- Necessitano di un registro (insieme di flip-flop) per memorizzare lo stato interno.
- Usano una rete combinatoria per calcolare il prossimo stato.
- Possono avere una rete combinatoria per produrre lo stato d'uscita.
- Sono dette Macchine di Moore se l'uscita dipende solo dallo stato interno; sono chiamate Macchine di Mealy quelle il cui stato d'uscita dipende sia dallo stato interno che dallo stato d'ingresso.



ESEMPI DI RETE SEQUENZIALE SINCRONA (1)



ESEMPI DI RETE SEQUENZIALE SINCRONA (2)



AUMENTO DELLA COMPLESSITÀ CIRCUITALE

- Per ridurre dimensione, consumo e costo e permettere strutture logiche più complesse e veloci, l'industria dei semiconduttori ha introdotto componenti sempre più complessi e nuove tecnologie hardware e relativi sistemi software di sviluppo.
 - i. Le cosiddette «logiche programmabili semplici» (SPLD) quali PROM, PAL, PLA, GAL, PALCE in sostituzione di parti di reti fatte con componenti discreti (porte logiche e piccole funzioni SSI).
 - ii. Introduzione dei primi rudimentali strumenti software HDL per gestire la programmazione degli SPLD con PALASM e successivamente ABEL, ISP e CUPL.
 - iii. Apparizione sul mercato delle prime «logiche programmabili complesse» (CPLD).
 - iv. Comparsa del primo dispositivo Gate Array programmabile (FPGA), ovvero XC2064 di Xilinx inc., introdotto nel 1985. Il tool per programmare questo dispositivo si chiamava XACT (Xilinx Advanced CAD Technology), la cui sintassi era basata su ABEL (Advanced Boolean Expression Language) di Data I/O Corporation.

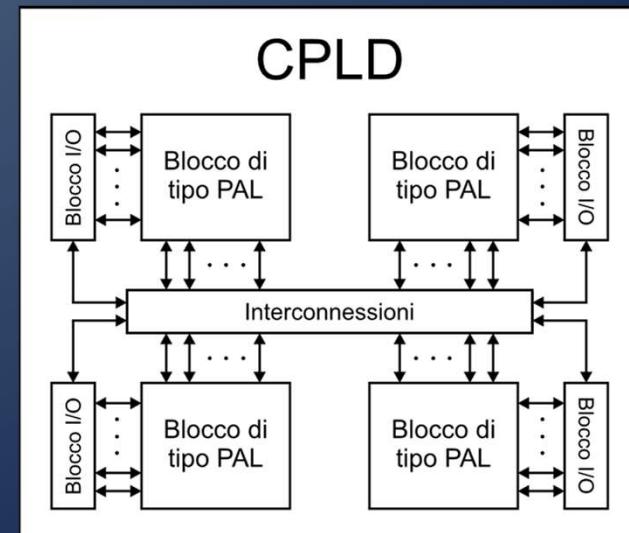
CPLD

Le CPLD possono essere considerate un'evoluzione delle PAL e sono costituite da più strutture PAL messe assieme, note come «macrocelle».

Le macrocelle sono collegate tramite un'interconnessione programmabile, denominata anche GIM (global interconnection matrix). Riconfigurando la GIM è possibile realizzare i più disparati circuiti logici interconnettendo le macrocelle fra loro.

Le CPLD comunicano con il mondo esterno utilizzando gli I/O digitali. Nell'esempio a lato si può vedere che questi ultimi sono divisi in gruppi intimamente ed efficientemente collegati alla propria macrocella. Comunque una macrocella può raggiungere gli I/O di altre macrocelle attraverso la matrice di interconnessione programmabile (GIM), seppur meno efficientemente.

Le CPLD hanno una complessità sufficiente da rendere necessari strumenti CAE per la loro programmazione.



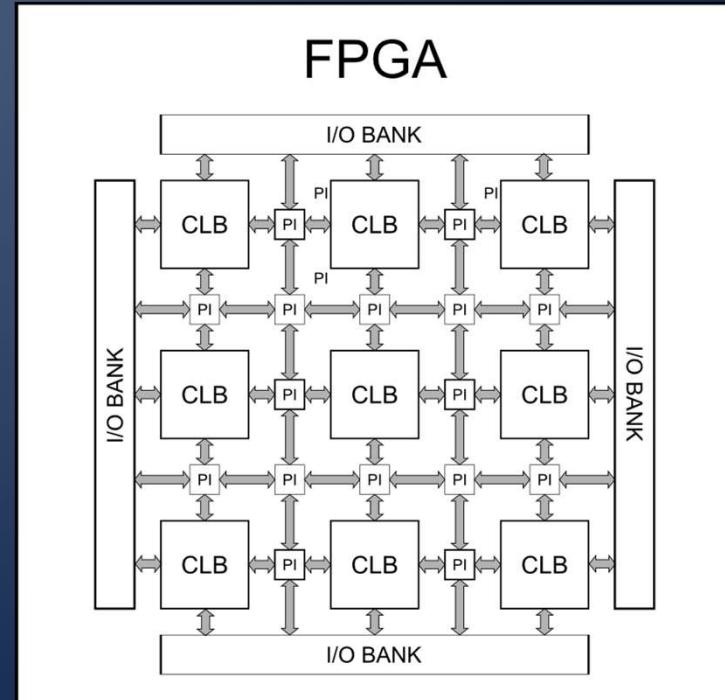
FPGA

Le FPGA sono a tutti gli effetti dei Gate Array programmabili. Si tratta dei dispositivi programmabili dall'utente che più si avvicinano agli ASIC. Le FPGA sono organizzate in blocchi logici (CLB, Configurable Logic Block), un po' come per le CPLD, ma questi sono più semplici, in numero maggiore e possono essere interconnessi fra loro con le interconnessioni programmabili (PI). Queste ultime collegano gli I/O i quali normalmente sono divisi in banchi.

In generale, le CLB sono costituite da un numero relativamente esiguo di elementi. Una tipica CLB è costituita da una LUT (Look Up Table) a 4-5 ingressi, un Full-Adder e un flip-flop di tipo D. Spesso è presente anche un multiplexer all'ingresso del Flip-Flop. Non esiste uno schema standard per le LUT che sono sempre diverse a seconda del costruttore.

Le FPGA moderne hanno anche molti macroblocchi predefiniti (Hard Block) dedicati a funzioni complesse, e sono specificamente ottimizzati in termini di velocità e spazio occupato sul chip. Ci sono dispositivi con blocchi di memoria, moltiplicatori paralleli, intere funzioni DSP, processori e molto altro. Le FPGA implementano anche il supporto per la generazione dei clock (PLL) e dei ritardi programmabili. Alcune sono tanto ricche di risorse da poter implementare interi sistemi al loro interno (SoC, System on Chip).

La programmazione di una FPGA moderna è affrontabile solo con strumenti software sofisticati che prevedono la mappatura più efficiente delle celle e un oculato instradamento dei segnali fra le celle e verso gli I/O.



GESTIONE DELLA COMPLESSITÀ CIRCUITALE

- Sintesi di complessità inaffrontabile manualmente o coi primitivi HDL relativamente all'utilizzo di moderne FPGA o PLD complesse.
- Simulazione come supporto primario alla progettazione di reti logiche per anticipare la verifica del funzionamento ben prima della realizzazione.
- Necessità di sintetizzare automaticamente la rete logica o parti di essa.
- Comparsa dei primi strumenti sofisticati di Automazione Elettronica del Progetto (EDA).

PARTE SECONDA

HARDWARE DESCRIPTION LANGUAGES

IL PROGETTO AUTOMATIZZATO

- Gli HDL (Hardware Description Language) sono linguaggi di modellazione, simulazione ed eventuale verifica formale dell'hardware.
- Se opportunamente scritto, un modello definito con HDL può ammettere la sintesi automatica.
- I linguaggi principalmente utilizzati sono Verilog (e SystemVerilog), VHDL e più raramente SystemC.
- Permettono la descrizione del modello a diversi livelli di astrazione (comportamentale, strutturale RTL, a livello di gate o di switch/transistor).
- Ammettono la descrizione di vettori di test per stimolare i modelli da eseguire su appositi simulatori. Possono supportare funzioni stocastiche (code e distribuzioni di probabilità casuali) per supportare la verifica delle prestazioni.
- Permettono eventualmente la verifica formale del modello (SystemVerilog).

EDA – ELECTRONIC DESIGN AUTOMATION (1)

- Insieme di strumenti, nello specifico dei sistemi digitali, per la progettazione di ASIC o per lo sviluppo su CPLD e FPGA.
- Ambiti degli EDA per lo sviluppo di sistemi digitali:
 1. Design Entry: definizione del progetto della rete logica (schemi elettrici e linguaggi HDL).
 2. Verifica della rete logica (pre-sintesi) a mezzo esecuzione del codice HDL su simulatore.
 3. Sintesi della rete logica dal codice HDL generando netlist (strutture di componenti/funzioni e nodi come interconnessioni) specifiche per l'architettura dell'hardware di destinazione.
 4. Simulazione della rete logica a livello strutturale (pre-layout): i blocchi della tecnologia sono considerati come componenti ideali.

EDA – ELECTRONIC DESIGN AUTOMATION (2)

- Ambiti degli EDA per lo sviluppo di sistemi digitali (continua)
 5. Place & Route, cioè mappatura della rete nei vari blocchi predefiniti della tecnologia scelta e instradamento dei segnali.
 6. Calcolo dei tempi necessari ai segnali per attraversare le interconnessioni ed i blocchi logici (tempi di propagazione). Generazione di una ulteriore netlist provvista di elementi atti a generare ritardi equivalenti ai tempi di propagazione sopra calcolati (back-annotation). Lo scopo della back-annotation è quello di avere simulazioni accurate post-sintesi che tengano conto anche delle condizioni fisiche operative (quali temperatura ambiente stimata e tensione/i di alimentazione).
 7. Programmazione del dispositivo (con bitstream) o generazione dei layout (maschere) per la produzione.

PARTE SECONDA

IL LINGUAGGIO VERILOG®

BREVE STORIA DEL LINGUAGGIO VERILOG

- Verilog HDL è stato inventato da Phil Moorby e Prabhu Goel intorno al 1984 come linguaggio di modellazione hardware proprietario di Gateway Design Automation Inc.
- Stabilizzato come specifiche nel 1990 e reso aperto da Cadence Design System nel 1991 sempre da Candence che organizzò il consorzio Open Verilog International (OVI).
- Successivamente è stato presentato all'IEEE ed è diventato lo standard IEEE 1364-1995, comunemente indicato come Verilog-95.
- Per coprire alcune carenze riscontrate del Verilog-95 alcune estensioni furono ripresentate all'IEEE e diventarono lo standard IEEE 1364-2001, noto come Verilog-2001.
- Verilog-2005 (IEEE Standard 1364-2005) è stato pubblicato con piccole correzioni e modifiche.
- Sempre nel 2005 è stato pubblicato SystemVerilog, un superset di Verilog-2005, con molte nuove funzionalità e capacità per facilitare la verifica della progettazione.
- A partire dal 2009, gli standard SystemVerilog e Verilog sono stati fusi in SystemVerilog 2009 (IEEE Standard 1800-2009).

VERILOG E LINGUAGGI PER COMPUTER

- Differenza dai linguaggi di programmazione per computer.
 - **Scopo:** Verilog serve a descrivere un circuito o un intero sistema digitale; un linguaggio di programmazione per computer serve a produrre codice eseguibile.
 - **Costrutti:** Verilog descrive le proprietà di fili, registri, porte logiche, etc., mentre un linguaggio di programmazione descrive variabili, funzioni e strutture per il controllo di flusso.
 - **Esecuzione:** Verilog descrive ma non esegue direttamente: può essere utilizzato per generare configurazioni o maschere per circuiti fisici; un linguaggio per computer descrive codice che, una volta compilato, è eseguito su una CPU.

INTRODUZIONE AL LINGUAGGIO VERILOG

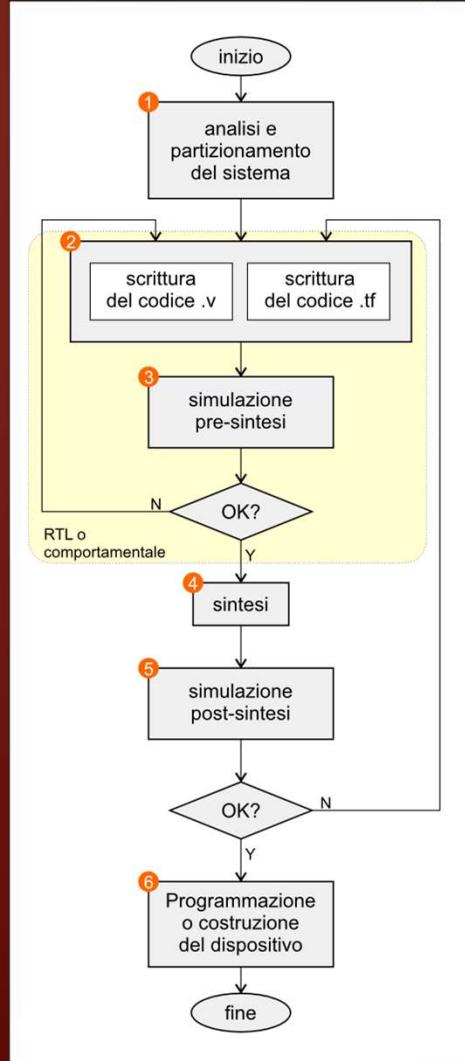
- Un progetto Verilog è costituito da almeno un file di testo; solitamente si usa .v come estensione (.sv per SystemVerilog).
- Verilog è «case sensitive»: var_X e var_x sono due oggetti diversi.
- Il costrutto principale in Verilog è il modulo (elemento lessicale 'module').
- Un file .v può contenere uno o più moduli e altri comandi, quali le cosiddette direttive e task (che iniziano con \$).
- Alcuni file possono avere l'estensione .tf (test fixtures): sono file Verilog (come i .v) che contengono i vettori (da usare col simulatore) di stimolo e verifica.

LIVELLI DI ASTRAZIONE

- Verilog supporta la modellazione comportamentale astratta, quindi può essere utilizzato per modellare la funzionalità di una rete logica ad un alto livello di astrazione. Ciò è utile per verificare la fase di analisi e partizionamento del sistema (proof of concept).
- Verilog supporta il livello RTL (Register Transfer Level), che è utilizzato per la progettazione dettagliata dei circuiti digitali. Se il codice è scritto in modo appropriato, i tool di sintesi trasformano le descrizioni RTL fino al livello di gate/blocchi logici/switch.
- Verilog supporta le descrizioni a livello di gate e switch, utilizzate per la verifica di progetti digitali, inclusa la simulazione logica a bassissimo livello, analisi temporale statica e dinamica, analisi di testabilità e classificazione dei guasti.
- I vettori di test, che stimolano la rete e contengono i risultati attesi per confrontarli con l'uscita della rete stessa sono scritti anch'essi in Verilog, solitamente in forma non sintetizzabile.
- Al livello più alto, Verilog contiene funzioni stocastiche (code e distribuzioni di probabilità casuali) per supportare le test-fixtures per la modellazione delle prestazioni.

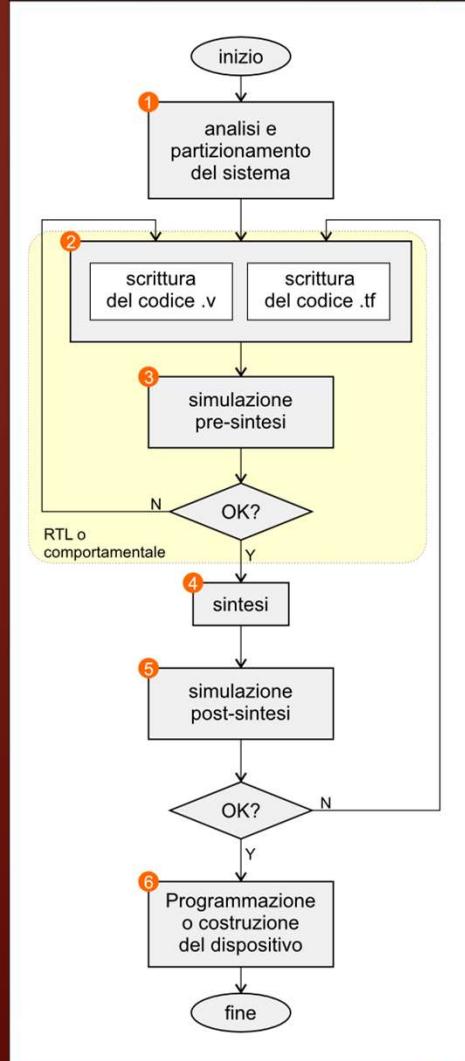
FLUSSO DI SVILUPPO ASIC CON HDL (1)

1. Analisi e partizionamento del problema.
2. Scrittura o editing dei blocchi di codice (moduli) in stile comportamentale o RTL e relativi vettori di test.
3. Simulazione (pre sintesi) del codice: in caso di risultati inaspettati si torna al passo 2.
4. Sintesi in blocchi specifici della tecnologia scelta.



FLUSSO DI SVILUPPO ASIC CON HDL (2)

5. Simulazione (post-sintesi) del codice: in caso di errore si torna al passo 2.
6. Place & route. Mapping delle «risorse» e interconnessioni.
7. La simulazione post-place & route è da considerarsi un passo facoltativo e non molto utile da farsi se non in circostanze particolari. Invece della simulazione si utilizza la tecnica basata sui cosiddetti «constraint» applicati ai percorsi dei segnali critici.
8. Programmazione del dispositivo o realizzazione circuitale.

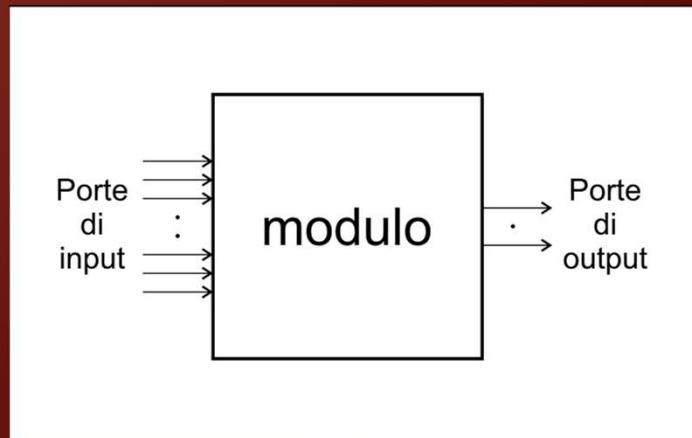


TUTORIAL INTRODUTTIVO (1)

I Moduli:

- Sono blocchi di codice che implementano funzionalità.
- Comunicano con l'esterno tramite porte (input/output), che sono assimilabili a connessioni elettriche (wire).
- Contengono costrutti di codice che ne descrivono la struttura interna.
- Sono delimitati dalle parole chiave *module* e *endmodule*.

```
module nome( [elenco porte] );
    // Contenuto del modulo
endmodule
```



TUTORIAL INTRODUTTIVO (2)

I Moduli:

- Possono essere privi di un elenco delle porte.
- Le keyword *input* e *output* si usano per definire le rispettive porte di ingresso ed uscita.
- All'interno del modulo ci sono i costrutti che definiscono il comportamento. In questo riquadro vediamo un costrutto che rappresenta un assegnamento combinatorio «continuo».

```
module nome;  
    // Contenuto del modulo  
endmodule
```

```
module fn_Z( input a, input b, input c, output z );  
    // Contenuto del modulo  
endmodule
```

```
module device_under_test( input a, input b, input c, output z );  
    assign z = ~a & ~b & ~c | a & b | b & c;  
endmodule
```

TUTORIAL INTRODUTTIVO (3)

I Moduli:

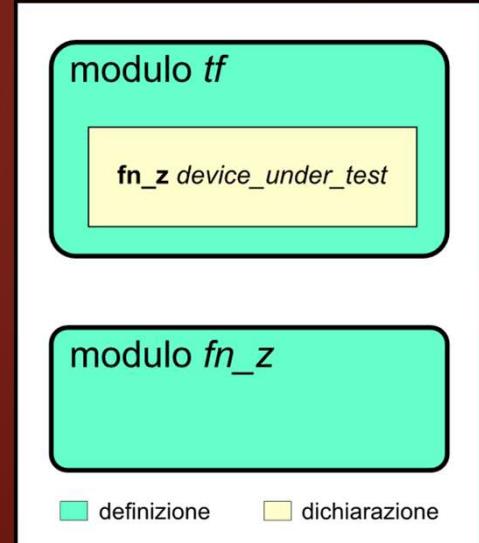
- Per verificare la correttezza di quanto abbiamo scritto finora, useremo uno strumento open source chiamato Icarus Verilog (iverilog). Icarus Verilog è un'implementazione per il linguaggio di descrizione dell'hardware Verilog IEEE-1364.
- Icarus Verilog è un compilatore e simulatore, che legge un file sorgente Verilog e produce un file di uscita. Quest'ultimo, nell'ambito di alcune piattaforme, può anche essere un eseguibile puro. Se la compilazione del modulo non produce errori siamo di fronte a codice almeno sintatticamente corretto.

```
$ iverilog assign.v
$ ll
drwxr-xr-x  2 user user 4096 Apr 23 12:43 .
drwxr-x--- 29 user user 4096 Apr 23 11:56 ..
-rwxr-xr-x  1 user user 2367 Apr 23 12:43 a.out*
-rw-r--r--  1 user user   109 Apr 23 12:32 assign.v
$ ./a.out
$ _
```

TUTORIAL INTRODUTTIVO (4/1)

I Moduli:

- I sistemi digitali sono strutturati gerarchicamente: vale a dire che sono costituiti da blocchi interconnessi fra loro aventi funzioni più o meno complesse. Alcuni blocchi contengono blocchi che a loro volta possono contenere altri blocchi e così via. Verilog, tramite i moduli, può modellare la struttura gerarchica di tali sistemi.
- I moduli comunque non possono essere definiti all'interno di altri moduli. Anche se questo sembra contraddirsi il punto precedente, va ricordato che – alla stregua dei linguaggi di programmazione – anche Verilog distingue fra dichiarazioni e definizioni.



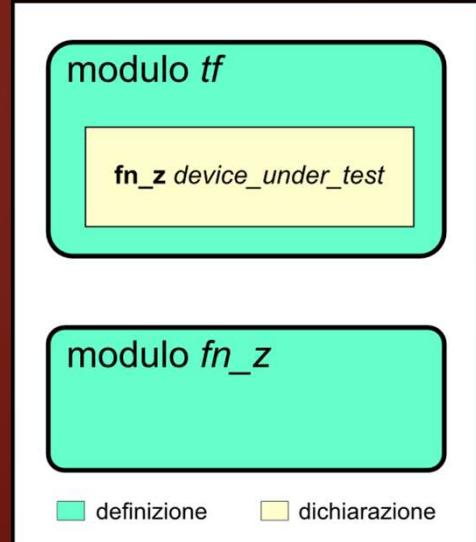
```
module tf;
  fn_z device_under_test(...);
endmodule

module fn_z(...);
  // Contenuto del modulo
endmodule
```

TUTORIAL INTRODUTTIVO (4/2)

I Moduli:

- I moduli possono altresì contenere «istanze» (cioè dichiarazioni) di altri moduli, ma non possono contenere direttamente o indirettamente istanze di se stessi.
- Un modulo può anche dichiarare multiple istanze di altri moduli.
- All'apice della gerarchia avremo un unico modulo che li contiene tutti (direttamente o indirettamente) ed è identificato come «top» module (*tf* nell'esempio a fianco).



```
module tf;
    fn_z device_under_test(...);
endmodule

module fn_z(...);
    // Contenuto del modulo
endmodule
```

TUTORIAL INTRODUTTIVO (5/1)

I Moduli:

- Alcuni moduli (test bench) possono contenere gli stimoli per la simulazione (test fixtures). Gli stimoli sono di solito raggruppati in opportuni costrutti.

```
// File assign_sim.v
`timescale 10ns/100ps

module tf;
    reg a, b, c;
    wire z;

initial begin
    $monitor( a, b, c,, z );
    $dumpvars( 0, tf );

    a = 0; b = 0; c = 0; #10
    a = 0; b = 0; c = 1; #10
    a = 0; b = 1; c = 0; #10
    a = 0; b = 1; c = 1; #10
    a = 1; b = 0; c = 0; #10
    a = 1; b = 0; c = 1; #10
    a = 1; b = 1; c = 0; #10
    a = 1; b = 1; c = 1; #10

    $finish;
end

fn_z device_under_test( a, b, c, z );
endmodule

module fn_z( input a, b, c, output z );
    assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (5/2)

I Moduli:

- Alcuni moduli (test bench) possono contenere gli stimoli per la simulazione (test fixtures). Gli stimoli sono di solito raggruppati in opportuni costrutti.
- Nel nostro esempio, per definire una sequenza temporale di applicazione degli stimoli si usa la keyword *initial*. Durante la simulazione, quanto controllato da *initial* è eseguito solo una volta. Per poter incorporare più di un comando dentro *initial*, dobbiamo ricorrere ad un altro costrutto che rappresenta il cosiddetto blocco.

```
// File assign_sim.v
`timescale 10ns/100ps

module tf;
    reg a, b, c;
    wire z;

    initial begin
        $monitor( a, b, c,, z );
        $dumpvars( 0, tf );

        a = 0; b = 0; c = 0; #10
        a = 0; b = 0; c = 1; #10
        a = 0; b = 1; c = 0; #10
        a = 0; b = 1; c = 1; #10
        a = 1; b = 0; c = 0; #10
        a = 1; b = 0; c = 1; #10
        a = 1; b = 1; c = 0; #10
        a = 1; b = 1; c = 1; #10

        $finish;
    end

    fn_z device_under_test( a, b, c, z );
endmodule

module fn_z( input a, b, c, output z );
    assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (5/3)

I Moduli:

- Alcuni moduli (test bench) possono contenere gli stimoli per la simulazione (test fixtures). Gli stimoli sono di solito raggruppati in opportuni costrutti.
- Nel nostro esempio, per definire una sequenza temporale di applicazione degli stimoli si usa la keyword *initial*. Durante la simulazione, quanto controllato da *initial* è eseguito solo una volta. Per poter incorporare più di un comando dentro *initial*, dobbiamo ricorrere ad un altro costrutto che rappresenta il cosiddetto blocco.
- Il blocco è sintatticamente definito dalla coppia di keyword *begin...end*. All'interno di un blocco, che è sempre controllato da un altro costrutto (in questo caso *initial*), le dichiarazioni sono considerate essere un tutt'uno. Nel caso di *initial* i comandi del blocco (ad esclusione delle direttive che iniziano con \$) sono eseguiti sequenzialmente. I blocchi *initial* non sono sintetizzabili. Per il momento i dettagli dei costrutti per la simulazione saranno ignorati.

```
// File assign_sim.v
`timescale 10ns/100ps

module tf;
    reg a, b, c;
    wire z;

    initial begin
        $monitor( a, b, c,, z );
        $dumpvars( 0, tf );

        a = 0; b = 0; c = 0; #10
        a = 0; b = 0; c = 1; #10
        a = 0; b = 1; c = 0; #10
        a = 0; b = 1; c = 1; #10
        a = 1; b = 0; c = 0; #10
        a = 1; b = 0; c = 1; #10
        a = 1; b = 1; c = 0; #10
        a = 1; b = 1; c = 1; #10

        $finish;
    end

    fn_z device_under_test( a, b, c, z );
endmodule

module fn_z( input a, b, c, output z );
    assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (6/1)

I Moduli:

- All'interno dei moduli di simulazione esistono le istanze dei moduli da simulare (costituiscono generalmente una gerarchia). Nell'esempio a fianco, vediamo che il modulo *fn_z* definisce una istanza (*device_under_test*) che riceve gli stimoli generati dal modulo ospitante *tf*.

```
// File assign_sim.v
`timescale 10ns/100ps

module tf;
    reg a, b, c;
    wire z;

    initial begin
        $monitor( a, b, c,, z );
        $dumpvars( 0, tf );
        a = 0; b = 0; c = 0; #10
        a = 0; b = 0; c = 1; #10
        a = 0; b = 1; c = 0; #10
        a = 0; b = 1; c = 1; #10
        a = 1; b = 0; c = 0; #10
        a = 1; b = 0; c = 1; #10
        a = 1; b = 1; c = 0; #10
        a = 1; b = 1; c = 1; #10
    end

    fn_z deviceUnderTest( a, b, c, z );
endmodule

module fn_z( input a, b, c, output z );
    assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (6/2)

I Moduli:

- All'interno dei moduli di simulazione esistono le istanze dei moduli da simulare (costituiscono generalmente una gerarchia). Nell'esempio a fianco, vediamo che il modulo *fn_z* definisce una istanza (*device_under_test*) che riceve gli stimoli generati dal modulo ospitante *tf*.
- Esistono altri elementi fondamentali che abbiamo già incontrato. Oltre a *reg* vediamo la keyword *wire* evidenziata a lato. Entrambi gli elementi permettono l'interconnessione fra i moduli.
- I *wire* sono assimilabili a collegamenti elettrici reali. Possono essere letti o assegnati. Nessun valore viene memorizzato al loro interno. Devono essere pilotati da una istruzione *assign* o da una porta di un modulo.
- I *reg* rappresentano gli elementi di memoria (registri) già visti precedentemente. Mantengono il loro valore finché non gliene viene fornito uno successivo. Contrariamente al loro nome, i *reg* non corrispondono necessariamente a registri fisici. Possono essere sintetizzati come Flip-Flop, latch o circuiti combinatori.

```
// File assign_sim.v
`timescale 10ns/100ps

module tf;
    reg a, b, c;
    wire z;

    initial begin
        $monitor( a, b, c,, z );
        $dumpvars( 0, tf );
        a = 0; b = 0; c = 0; #10
        a = 0; b = 0; c = 1; #10
        a = 0; b = 1; c = 0; #10
        a = 0; b = 1; c = 1; #10
        a = 1; b = 0; c = 0; #10
        a = 1; b = 0; c = 1; #10
        a = 1; b = 1; c = 0; #10
        a = 1; b = 1; c = 1; #10
    end

    $finish;
end

fn_z deviceUnderTest( a, b, c, z );
endmodule

module fn_z( input a, b, c, output z );
    assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (7/1)

I Moduli:

- Gli stimoli veri e propri sono costituiti da sequenze di valori assegnati ai **registri**.

```
// File assign_sim.v
`timescale 10ns/100ps
```

```
module tf;
    reg a, b, c;
    wire z;
```

```
initial begin
    $monitor( a, b, c,, z );
    $dumpvars( 0, tf );
```

```
a = 0; b = 0; c = 0; #10
a = 0; b = 0; c = 1; #10
a = 0; b = 1; c = 0; #10
a = 0; b = 1; c = 1; #10
a = 1; b = 0; c = 0; #10
a = 1; b = 0; c = 1; #10
a = 1; b = 1; c = 0; #10
a = 1; b = 1; c = 1; #10
```

```
$finish;
end
```

```
fn_z deviceUnderTest( a, b, c, z );
endmodule
```

```
module fn_z( input a, b, c, output z );
    assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

a	b	c	z
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



TUTORIAL INTRODUTTIVO (7/2)

I Moduli:

- Gli stimoli veri e propri sono costituiti da sequenze di valori assegnati ai registri.
- I "test bench" di Verilog dovrebbero includere una direttiva ``timescale`. Il primo valore definisce l'unità di tempo predefinita e il secondo la risoluzione della simulazione. In questo esempio, l'unità di tempo predefinita è 10 ns e la risoluzione della simulazione è 100 ps.

```
// File assign_sim.v
`timescale 10ns/100ps

module tf;
    reg a, b, c;
    wire z;

    initial begin
        $monitor( a, b, c,, z );
        $dumpvars( 0, tf );
        a = 0; b = 0; c = 0; #10
        a = 0; b = 0; c = 1; #10
        a = 0; b = 1; c = 0; #10
        a = 0; b = 1; c = 1; #10
        a = 1; b = 0; c = 0; #10
        a = 1; b = 0; c = 1; #10
        a = 1; b = 1; c = 0; #10
        a = 1; b = 1; c = 1; #10

        $finish;
    end

    fn_z device_under_test( a, b, c, z );
endmodule

module fn_z( input a, b, c, output z );
    assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (7/3)

I Moduli:

- Gli stimoli veri e propri sono costituiti da sequenze di valori assegnati ai registri.
- I "test bench" di Verilog dovrebbero includere una direttiva `timescale. Il primo valore definisce l'unità di tempo predefinita e il secondo la risoluzione della simulazione. In questo esempio, l'unità di tempo predefinita è 10 ns e la risoluzione della simulazione è 100 ps.
- Dopo ogni assegnamento ai registri *a*, *b*, e *c*, è presente l'operatore #. Nel nostro caso, # interrompe l'elaborazione per periodo di tempo calcolato moltiplicando l'unità di tempo definita dalla direttiva *timescale* per il valore che segue l'operatore stesso. Un ritardo di #10 risulterà in un passo di 100 ns.

```
// File assign_sim.v
`timescale 10ns/100ps

module tf;
    reg a, b, c;;
    wire z;

initial begin
    $monitor( a, b, c,, z );
    $dumpvars( 0, tf );

    a = 0; b = 0; c = 0; #10
    a = 0; b = 0; c = 1; #10
    a = 0; b = 1; c = 0; #10
    a = 0; b = 1; c = 1; #10
    a = 1; b = 0; c = 0; #10
    a = 1; b = 0; c = 1; #10
    a = 1; b = 1; c = 0; #10
    a = 1; b = 1; c = 1; #10

    $finish;
end

fn_z device_under_test( a, b, c, z );
endmodule

module fn_z( input a, b, c, output z );
    assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (8/1)

- Verilog dispone di net e variabili di tipo scalare e vettoriale.
 - Una dichiarazione senza specificare un cosiddetto «intervallo» è considerata larga un bit ed è uno scalare.

```
// File assign_sim_b.v
`timescale 10ns/100ps

module tf;
  reg [2:0] in;
  wire out;

initial begin
  $monitor( in,, in[2], in[1], in[0],, out );
  $dumpvars( 0, tf );

  in = 3'b000; #10
  in = 3'b001; #10
  in = 3'b010; #10
  in = 3'b011; #10
  in = 3'b100; #10
  in = 3'b101; #10
  in = 3'b110; #10
  in = 3'b111; #10

  $finish;
end

fn_z device_under_test( in[2], in[1], in[0], out );
endmodule

module fn_z( input a, b, c, output z );
  assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (8/2)

- Verilog dispone di net e variabili di tipo scalare e vettoriale.
 - Una dichiarazione senza specificare un cosiddetto «intervallo» è considerata larga un bit ed è uno scalare.
 - Se in una dichiarazione viene specificato un intervallo, avremo un'entità a più bit nota anche come «vettore». La sintassi per definire un intervallo (e quindi un vettore) è la seguente:
[MSB bit index : LSB bit index]

```
// File assign_sim_b.v
`timescale 10ns/100ps

module tf;
reg [2:0] in;
wire out;

initial begin
$monitor( in,, in[2], in[1], in[0],, out );
$dumpvars( 0, tf );

in = 3'b000; #10
in = 3'b001; #10
in = 3'b010; #10
in = 3'b011; #10
in = 3'b100; #10
in = 3'b101; #10
in = 3'b110; #10
in = 3'b111; #10

$finish;
end

fn_z device_under_test( in[2], in[1], in[0], out );
endmodule

module fn_z( input a, b, c, output z );
assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (8/3)

- Verilog dispone di net e variabili di tipo scalare e vettoriale.
 - Una dichiarazione senza specificare un cosiddetto «intervallo» è considerata larga un bit ed è uno scalare.
 - Se in una dichiarazione viene specificato un intervallo, avremo un'entità a più bit nota anche come «vettore». La sintassi per definire un intervallo (e quindi un vettore) è la seguente:
[MSB bit index : LSB bit index]
- In verilog per specificare i valori si usa la seguente sintassi:
[width][radix][literal]
 - width*: dichiara l'ampiezza del valore in bit;
 - literal*: rappresenta il valore vero e proprio
 - radix*: specifica la natura del valore espresso da *literal* e può essere: *d* (decimale), *h* (esadecimale), *o* (ottale), *b* (binario).

```
// File assign_sim_b.v
`timescale 10ns/100ps

module tf;
  reg [2:0] in;
  wire out;

initial begin
  $monitor( in,, in[2], in[1], in[0],, out );
  $dumpvars( 0, tf );

  in = 3'b000; #10
  in = 3'b001; #10
  in = 3'b010; #10
  in = 3'b011; #10
  in = 3'b100; #10
  in = 3'b101; #10
  in = 3'b110; #10
  in = 3'b111; #10

  $finish;
end

fn_z device_under_test( in[2], in[1], in[0], out );
endmodule

module fn_z( input a, b, c, output z );
  assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (9/1)

- Indicizzazione di vettori

Per selezionare uno o più bit appartenenti ad un vettore si usa l'operatore `[]`, ed ha la seguente sintassi:

[msb-index:lsb-index] o [bit-index]

```
// File assign_sim_b.v
`timescale 10ns/100ps

module tf;
  reg [2:0] in;
  wire out;

  initial begin
    $monitor( in,, in[2], in[1], in[0],, out );
    $dumpvars( 0, tf );

    in = 3'b000; #10
    in = 3'b001; #10
    in = 3'b010; #10
    in = 3'b011; #10
    in = 3'b100; #10
    in = 3'b101; #10
    in = 3'b110; #10
    in = 3'b111; #10

    $finish;
  end

  fn_z device_under_test( in[2], in[1], in[0], out );
endmodule

module fn_z( input a, b, c, output z );
  assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (9/2)

- Indicizzazione di vettori

Per selezionare uno o più bit appartenenti ad un vettore si usa l'operatore `[]`, ed ha l' seguente sintassi:

`[msb-index:lsb-index]` o `[bit-index]`

- Il modulo può avere i parametri che descrivono l'interfaccia con la quale comunica esternamente. Ai parametri sono associati i nomi necessari all'utilizzo all'interno del modulo.

```
// File assign_sim_b.v
`timescale 10ns/100ps

module tf;
  reg [2:0] in;
  wire out;

  initial begin
    $monitor( in,, in[2], in[1], in[0],, out );
    $dumpvars( 0, tf );
    in = 3'b000; #10
    in = 3'b001; #10
    in = 3'b010; #10
    in = 3'b011; #10
    in = 3'b100; #10
    in = 3'b101; #10
    in = 3'b110; #10
    in = 3'b111; #10
  end

  $finish;
end

fn_z device_under_test( in[2], in[1], in[0], out );
endmodule

module fn_z( input a, b, c, output z );
  assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (9/3)

- Indicizzazione di vettori

Per selezionare uno o più bit appartenenti ad un vettore si usa l'operatore `[]`, ed ha l' seguente sintassi:

`[msb-index:lsb-index]` o `[bit-index]`

- Il modulo può avere i parametri che descrivono l'interfaccia con la quale comunica esternamente. Ai parametri sono associati i nomi necessari all'utilizzo all'interno del modulo.
- È necessario dichiarare la **direzione** dei segnali per tutti i parametri del modulo.

```
// File assign_sim_b.v
`timescale 10ns/100ps

module tf;
  reg [2:0] in;
  wire out;

  initial begin
    $monitor( in,, in[2], in[1], in[0],, out );
    $dumpvars( 0, tf );
    in = 3'b000; #10
    in = 3'b001; #10
    in = 3'b010; #10
    in = 3'b011; #10
    in = 3'b100; #10
    in = 3'b101; #10
    in = 3'b110; #10
    in = 3'b111; #10
  end

  $finish;
end

fn_z device_under_test( in[2], in[1], in[0], out );
endmodule

module fn_z( input a, b, c, output z );
  assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

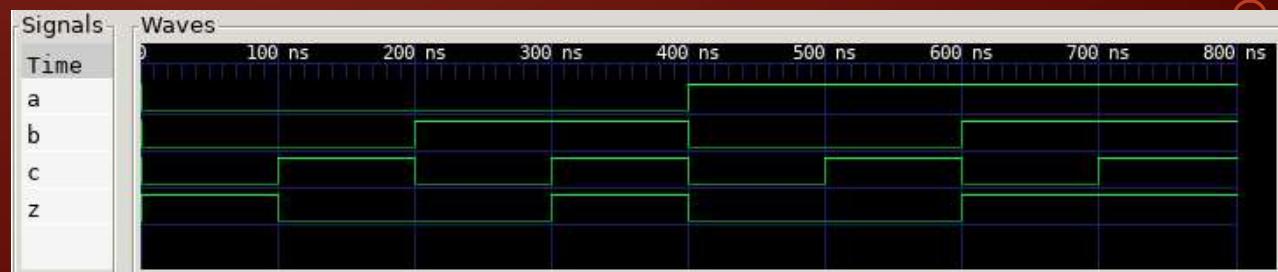
TUTORIAL INTRODUTTIVO (10)

I Moduli:

- Con gli stimoli forniti nel modulo *tf* all'istanza *device_under_test* di tipo *fn_z*, si esegue ancora Icarus Verilog.
- Lanciando il comando *vvp* (il motore run-time del simulatore) e passandogli *assign_sim.out* come parametro, viene prodotto il file *dump.vcd* e scritto il risultato della simulazione sullo standard output. Si può vedere che l'ultima colonna corrisponde a quanto previsto dalla tabella di verità della funzione z.
- Infine il file *dump.vcd* passato al software *gtkwave*, permette di visualizzare un diagramma con i segnali *a*, *b*, *c* e *z* che evolvono nel tempo.

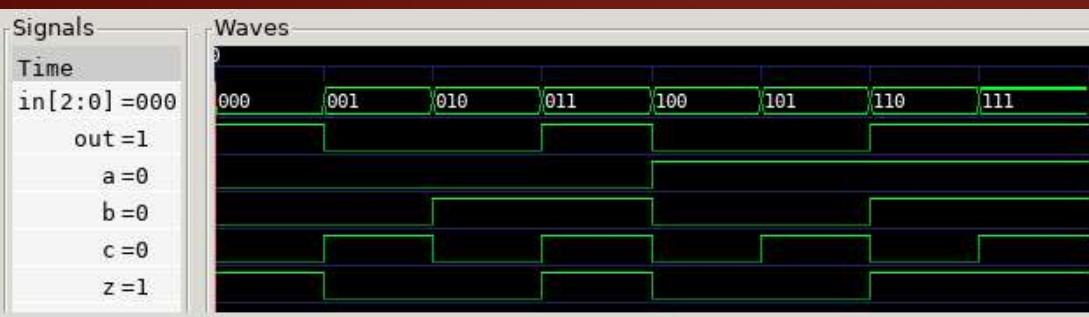
<i>a</i>	<i>b</i>	<i>c</i>	<i>z</i>
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

```
$ iverilog assign_sim.v -o assign_sim.out
$ vvp assign_sim.out
VCD info: dumpfile dump.vcd opened for output.
000 1
001 0
010 0
011 1
100 0
101 0
110 1
111 1
$ gtkwave dump.vcd &
$ _
```



TUTORIAL INTRODUTTIVO (11)

- Come abbiamo visto, i tipi di dati si usano per caratterizzare gli elementi di memoria (reg) e le «net», che in Verilog ricordiamo sono assimilabili ai wire. È possibile raggruppare ordinatamente più wire o reg per rappresentare un unico tipo di dato.
- Gli agglomerati di wire possono essere considerati alla stregua dei cosiddetti «Bus» come sono classicamente rappresentati nel contesto dei sistemi digitali.
- Gli elementi di memoria sono assimilabili alle variabili nei linguaggi di programmazione; un singolo Flip-Flop (reg) è un esempio di variabile che memorizza un bit.
Un agglomerato di reg rappresenta un tipico registro di n -bit.



```
// File assign_sim_b.v
`timescale 10ns/100ps

module tf;
  reg [2:0] in;
  wire out;

  initial begin
    $monitor( in,, in[2], in[1], in[0],, out );
    $dumpvars( 0, tf );
    in = 3'b000; #10
    in = 3'b001; #10
    in = 3'b010; #10
    in = 3'b011; #10
    in = 3'b100; #10
    in = 3'b101; #10
    in = 3'b110; #10
    in = 3'b111; #10

    $finish;
  end

  fn_z device_under_test( in[2], in[1], in[0], out );
endmodule

module fn_z( input a, b, c, output z );
  assign z = ~a & ~b & ~c | a & b | b & c;
endmodule
```

TUTORIAL INTRODUTTIVO (12)

Gli operatori

Verilog possiede diversi operatori: a lato vediamo una tabella riassuntiva dove possiamo trovare anche gli operatori logici (bitwise o bit-a-bit) usati finora. Sotto la tabella delle precedenze.

Operatore	Precedenza
+ - ! ~ (unari)	alta
**	
* / %	
+ - (binari)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& ~&	
^ ~^ ^~	
~	
&&	
? :	bassa

Operatore	Operazione	Descrizione
{ , }	Concatenazione	Unisce i bit di due o più espressioni separate da virgole
+	Addizione	Somma due operandi
-	Sottrazione	Differenza tra due operandi
-	Meno unario	Cambia il segno dell'operando
*	Moltiplicazione	Moltiplica due operandi
/	Divisione	Divide due operandi
%	Modulo	Calcola il resto
**	Potenza	Elevazione a potenza
>	Maggiore di	Vero se maggiore
>=	Maggiore o uguale di	Vero se maggiore o uguale
<	Minore di	Vero se minore
<=	Minore o uguale di	Vero se minore o uguale
!	Negazione logica	Complemento logico unario
&&	AND logico	AND di due valori logici
	OR logico	OR di due valori logici
==	Uguaglianza logica	Confronta due valori per l'uguaglianza
!=	Disuguaglianza logica	Confronta due valori per la disuguaglianza
==>	Uguaglianza (case)	Confronta due valori per l'uguaglianza (case)
!=>	Disuguaglianza (case)	Confronta due valori per la disuguaglianza (case)
~	Negazione bitwise	Completa ogni bit dell'operando
&	AND bit a bit	Produce l'AND bit a bit di due operandi
	OR bit a bit	Produce l'OR inclusivo bit a bit di due operandi
^	XOR bit a bit	Produce lo XOR esclusivo bit a bit di due operandi
^~ or ^~	Equivalenza	Produce lo XNOR esclusivo bit a bit di due operandi
&	Riduzione unaria AND	Produce l'AND a singolo bit di tutti i bit dell'operando
~&	Riduzione unaria NAND	Produce il NAND a singolo bit di tutti i bit dell'operando
	Riduzione unaria OR	Produce l'OR a singolo bit di tutti i bit dell'operando
~	Riduzione unaria NOR	Produce il NOR a singolo bit di tutti i bit dell'operando
^	Riduzione unaria XOR	Produce lo XOR a singolo bit di tutti i bit dell'operando
^~ or ^~	Riduzione unaria XNOR	Produce lo XOR a singolo bit di tutti i bit dell'operando
<<	Scorrimento a sinistra	Sposta l'operando sinistra a sinistra (logicamente) del numero di posizioni di bit specificato dall'operando destro
>>	Scorrimento a destra	Sposta l'operando sinistro a destra (logicamente) del numero di posizioni di bit specificato dall'operando destro
<<<	Scorrimento aritmetico a sinistra	Sposta l'operando sinistro a sinistra (aritmeticamente) del numero di posizioni di bit specificato dall'operando destro
>>>	Scorrimento aritmetico a destra	Sposta l'operando sinistro a destra (aritmeticamente) del numero di posizioni di bit specificato dall'operando destro
? :	Condizionale	Assegna uno dei due valori in base all'espressione

TUTORIAL INTRODUTTIVO (13/1)

Blocco Always

- Verilog consente una logica più complessa del semplice `assign` grazie all'uso dei **blocchi *always***. La logica combinatoria può essere scritta utilizzando `always @(*)`. Il valore all'interno delle parentesi è chiamata «*sensitivity list*». L'uso di un * indica al tool di calcolare automaticamente suddetta «*sensitivity list*».

```
module fn_z( input a, b, c, output /*reg*/ z );
  reg z; // necessario con always
  // assign z = ~a & ~b & ~c | a & b | b & c;
  // always @(a,b,c)
  always @(*)
    begin
      z = ~a & ~b & ~c | a & b | b & c;
    end
  endmodule
```

TUTORIAL INTRODUTTIVO (13/2)

Blocco Always

- Verilog consente una logica più complessa del semplice `assign` grazie all'uso dei blocchi `always`. La logica combinatoria può essere scritta utilizzando `always @(*)`. Il valore all'interno delle parentesi è chiamata «*sensitivity list*». L'uso di un * indica al tool di calcolare automaticamente suddetta «*sensitivity list*».
- Se un segnale è controllato all'interno di un blocco `always`, deve essere definito come `reg`. Viceversa, come abbiamo visto, se è utilizzato un assegnamento continuo (`assign`), può essere definito come `wire` il quale rappresenta sempre una connessione, mentre un `reg` è equivalente ad un `wire` se è utilizzato all'interno di un costrutto `always @(*)`.

```
module fn_z( input a, b, c, output /*reg*/ z );
  reg z; // necessario con always
  // assign z = ~a & ~b & ~c | a & b | b & c;
  // always @(a,b,c)
  always @(*)
    begin
      z = ~a & ~b & ~c | a & b | b & c;
    end
  endmodule
```

TUTORIAL INTRODUTTIVO (14)

Il blocco *always*

- Un **reg** rappresenta un registro se è usato all'interno di un **blocco *always*** con logica sequenziale sincrona, la quale è generata specificando la lista dei segnali nella **«sensitivity list»** del blocco *always*.

```
module counter( input Clk, input G );
  reg [2:0] S = 3'd0; // WARNING: non sintetizzabile

  always @( posedge Clk ) begin
    case( S )
      3'b000: S <= !G ? 3'b001 : 3'b001;
      3'b001: S <= !G ? 3'b010 : 3'b011;
      3'b010: S <= !G ? 3'b011 : 3'b110;
      3'b011: S <= !G ? 3'b100 : 3'b010;
      3'b100: S <= !G ? 3'b101 : 3'b000;
      3'b101: S <= !G ? 3'b110 : 3'b100;
      3'b110: S <= !G ? 3'b111 : 3'b111;
      3'b111: S <= !G ? 3'b000 : 3'b101;
    endcase
  end
endmodule
```

TUTORIAL INTRODUTTIVO (14/2)

Il blocco *always*

- Un *reg* rappresenta un registro se è usato all'interno di un blocco *always* con logica sequenziale sincrona, la quale è generata specificando la lista dei segnali nella «*sensitivity list*» del blocco *always*.
- Inoltre, con le keyword *posedge* e *negedge* è possibile indicare su quale fronte (variazione del segnale) viene attivato il blocco *always*; *posedge* $0 \rightarrow 1$, *negedge* $1 \rightarrow 0$.

```
module counter( input Clk, input G );
  reg [2:0] S = 3'd0; // WARNING: non sintetizzabile

  always @(posedge Clk) begin
    case( S )
      3'b000: S <= !G ? 3'b001 : 3'b001;
      3'b001: S <= !G ? 3'b010 : 3'b011;
      3'b010: S <= !G ? 3'b011 : 3'b110;
      3'b011: S <= !G ? 3'b100 : 3'b010;
      3'b100: S <= !G ? 3'b101 : 3'b000;
      3'b101: S <= !G ? 3'b110 : 3'b100;
      3'b110: S <= !G ? 3'b111 : 3'b111;
      3'b111: S <= !G ? 3'b000 : 3'b101;
    endcase
  end
endmodule
```

TUTORIAL INTRODUTTIVO (14/3)

Il blocco *always*

- Un *reg* rappresenta un registro se è usato all'interno di un blocco *always* con logica sequenziale sincrona, la quale è generata specificando la lista dei segnali nella «*sensitivity list*» del blocco *always*.
- Inoltre, con le keyword *posedge* e *negedge* è possibile indicare su quale fronte (variazione del segnale) viene attivato il blocco *always*; *posedge* $0 \rightarrow 1$, *negedge* $1 \rightarrow 0$.
- All'interno del blocco *always*, è contenuto uno statement *case* che serve a sintetizzare la funzione di prossimo stato, ripresa dall'esempio del contatore Gray/Binario visto in precedenza. Com'è noto, la funzione che calcola il prossimo stato è una struttura combinatoria e si può notare che in questo caso non sono state utilizzate delle equazioni per definirla. I blocchi *always* possono rendere il codice più leggibile e immediato da comprendere.

```
module counter( input Clk, input G );
  reg [2:0] S = 3'd0; // WARNING: non sintetizzabile

  always @( posedge Clk ) begin
    case( S )
      3'b000: S <= !G ? 3'b001 : 3'b001;
      3'b001: S <= !G ? 3'b010 : 3'b011;
      3'b010: S <= !G ? 3'b011 : 3'b110;
      3'b011: S <= !G ? 3'b100 : 3'b010;
      3'b100: S <= !G ? 3'b101 : 3'b000;
      3'b101: S <= !G ? 3'b110 : 3'b100;
      3'b110: S <= !G ? 3'b111 : 3'b111;
      3'b111: S <= !G ? 3'b000 : 3'b101;
    endcase
  end
endmodule
```

FLOW TABLE		
PRE	NEXT	
S ₂ S ₁ S ₀	G=0	G=1
000	001	001
001	010	011
010	011	110
011	100	010
100	101	000
101	110	100
110	111	111
111	000	101

TUTORIAL INTRODUTTIVO (14/4)

Il blocco *always*

- Un *reg* rappresenta un registro se è usato all'interno di un blocco *always* con logica sequenziale sincrona, la quale è generata specificando la lista dei segnali nella «*sensitivity list*» del blocco *always*.
- Inoltre, con le keyword *posedge* e *negedge* è possibile indicare su quale fronte (variazione del segnale) viene attivato il blocco *always*; *posedge* $0 \rightarrow 1$, *negedge* $1 \rightarrow 0$.
- All'interno del blocco *always*, è contenuto uno statement *case* che serve a sintetizzare la funzione di prossimo stato, ripreso dall'esempio del contatore Gray/Binario visto in precedenza. Com'è noto, la funzione che calcola il prossimo stato è una struttura combinatoria e si può notare che in questo caso non sono state utilizzate delle equazioni per definirla. I blocchi *always* possono rendere il codice più leggibile e immediato da comprendere.
- Verilog ha due operatori di assegnazione: bloccante ($=$) e non bloccante (\leq). L'operatore bloccante ($=$) si usa per la logica combinatoria e quello non bloccante per la logica sequenziale sincrona.

```
module counter( input Clk, input G );
  reg [2:0] S = 3'd0; // WARNING: non sintetizzabile

  always @(* posedge Clk ) begin
    case( S )
      3'b000: S <= !G ? 3'b001 : 3'b001;
      3'b001: S <= !G ? 3'b010 : 3'b011;
      3'b010: S <= !G ? 3'b011 : 3'b110;
      3'b011: S <= !G ? 3'b100 : 3'b010;
      3'b100: S <= !G ? 3'b101 : 3'b000;
      3'b101: S <= !G ? 3'b110 : 3'b100;
      3'b110: S <= !G ? 3'b111 : 3'b111;
      3'b111: S <= !G ? 3'b000 : 3'b101;
    endcase
  end
endmodule
```

TUTORIAL INTRODUTTIVO (15/1)

I moduli “Test Bench”

- In aggiunta al modulo del contatore vero e proprio, serve un modulo aggiuntivo (**clock**) che genera la cadenza (**clk_gen**). Questo modulo, così com’è scritto non è sintetizzabile, ma questo non è un problema in quanto serve solo per generare uno dei due vettori di test (**Cp**) per **counter_1**.

```
`timescale 10ns/100ps

module tf;
  reg Mode; // 0=Bin 1=Gray
  wire Cp;

  initial begin
    $dumpvars( 0, tf );
    Mode = 0; #102
    Mode = 1; #108
    $finish;
  end

  clock clk_gen( .Out( Cp ) );
  counter counter_1( .Clk( Cp ), .G( Mode ) );
endmodule

module clock( output reg Out );
  // WARNING: modulo non sintetizzabile
  initial
    Out = 1;

  always
    begin
      #5
      Out = ~Out;
    end
  endmodule
```

TUTORIAL INTRODUTTIVO (15/2)

I moduli “Test Bench”

- In aggiunta al modulo del contatore vero e proprio, serve un modulo aggiuntivo (clock) che genera la cadenza (clk_gen). Questo modulo, così com'è scritto non è sintetizzabile, ma questo non è un problema in quanto serve solo per generare uno dei due vettori di test (Cp) per counter_1.
- I due moduli dichiarati dentro *tf*, cioè clk_gen e counter_1, sono interconnessi fra loro tramite la net Cp (wire).

```
'timescale 10ns/100ps

module tf;
    reg Mode; // 0=Bin 1=Gray
    wire Cp;

    initial begin
        $dumpvars( 0, tf );
        Mode = 0; #102
        Mode = 1; #108
        $finish;
    end

    clock clk_gen(.Out( Cp ) );
    counter counter_1(.Clk( Cp ), .G( Mode ) );
endmodule

module clock( output reg Out );
    // WARNING: modulo non sintetizzabile
    initial
        Out = 1;

    always
        begin
            #5
            Out = ~Out;
        end
endmodule
```

TUTORIAL INTRODUTTIVO (15/3)

I moduli “Test Bench”

- In aggiunta al modulo del contatore vero e proprio, serve un modulo aggiuntivo (clock) che genera la cadenza (clk_gen). Questo modulo, così com'è scritto non è sintetizzabile, ma questo non è un problema in quanto serve solo per generare uno dei due vettori di test (Cp) per counter_1.
- I due moduli dichiarati dentro tf, cioè clk_gen e counter_1, sono interconnessi fra loro tramite la net Cp (wire).
- Nell'esempio ci sono due costrutti che iniziano con \$. Il primo (**\$dumpvars**) istruisce il simulatore a considerare la futura osservazione di tutte le net del modulo principale (il top module tf).

Il secondo costrutto è **\$finish**. Questa «task di sistema» dice al simulatore di interrompersi e terminare.

```
`timescale 10ns/100ps

module tf;
  reg Mode; // 0=Bin 1=Gray
  wire Cp;

  initial begin
    $dumpvars( 0, tf );
    Mode = 0; #102
    Mode = 1; #108
    $finish;
  end

  clock clk_gen( .Out( Cp ) );
  counter counter_1( .Clk( Cp ), .G( Mode ) );
endmodule

module clock( output reg Out );
  // WARNING: modulo non sintetizzabile
  initial
    Out = 1;

  always
    begin
      #5
      Out = ~Out;
    end
endmodule
```

TUTORIAL INTRODUTTIVO (16/1)

- Per rendere sintetizzabile il contatore, bisogna aggiungere un meccanismo per azzerare lo stato dall'esterno. A tal scopo si può aggiungere un *input* (net *Rst*) all'elenco degli input/output del modulo.

```
module counter( input Clk, input Rst, input G );
  reg [2:0] S;

  always @( posedge Clk or posedge Rst )
    begin
      if ( Rst )
        begin
          S <= 3'b000;
        end
      else
        begin
          case ( S )
            3'b000: S <= !G ? 3'b001 : 3'b001;
            3'b001: S <= !G ? 3'b010 : 3'b011;
            3'b010: S <= !G ? 3'b011 : 3'b110;
            3'b011: S <= !G ? 3'b100 : 3'b010;
            3'b100: S <= !G ? 3'b101 : 3'b000;
            3'b101: S <= !G ? 3'b110 : 3'b100;
            3'b110: S <= !G ? 3'b111 : 3'b111;
            3'b111: S <= !G ? 3'b000 : 3'b101;
          endcase
        end
    end
endmodule
```

TUTORIAL INTRODUTTIVO (16/2)

- Per rendere sintetizzabile il contatore, bisogna aggiungere un meccanismo per azzerare lo stato dall'esterno. A tal scopo si può aggiungere un *input* (net *Rst*) all'elenco degli input/output del modulo.
- La net *Rst* è quindi aggiunta anche alla *sensitivity list* dello statement *always* e ogni sua variazione (*posedge*) verrà quindi presa in considerazione.

```
module counter( input Clk, input Rst, input G );
  reg [2:0] S;

  always @(* posedge Clk or posedge Rst)
    begin
      if ( Rst )
        begin
          S <= 3'b000;
        end
      else
        begin
          case ( S )
            3'b000: S <= !G ? 3'b001 : 3'b001;
            3'b001: S <= !G ? 3'b010 : 3'b011;
            3'b010: S <= !G ? 3'b011 : 3'b110;
            3'b011: S <= !G ? 3'b100 : 3'b010;
            3'b100: S <= !G ? 3'b101 : 3'b000;
            3'b101: S <= !G ? 3'b110 : 3'b100;
            3'b110: S <= !G ? 3'b111 : 3'b111;
            3'b111: S <= !G ? 3'b000 : 3'b101;
          endcase
        end
    end
endmodule
```

TUTORIAL INTRODUTTIVO (16/3)

- Per rendere sintetizzabile il contatore, bisogna aggiungere un meccanismo per azzerare lo stato dall'esterno. A tal scopo si può aggiungere un *input* (net *Rst*) all'elenco degli input/output del modulo.
- La net *Rst* è quindi aggiunta anche alla *sensitivity list* dello statement *always* e ogni sua variazione (*posedge*) verrà quindi presa in considerazione.
- Se al momento dell'attivazione del blocco *always* il valore di *Rst* è uguale a 1, siamo di fronte ad una richiesta di azzeramento del registro che conserva lo stato.

```
module counter( input Clk, input Rst, input G );
  reg [2:0] S;

  always @(* posedge Clk or posedge Rst )
  begin
    if ( Rst )
      begin
        S <= 3'b000;
      end
    else
      begin
        case ( S )
          3'b000: S <= !G ? 3'b001 : 3'b001;
          3'b001: S <= !G ? 3'b010 : 3'b011;
          3'b010: S <= !G ? 3'b011 : 3'b110;
          3'b011: S <= !G ? 3'b100 : 3'b010;
          3'b100: S <= !G ? 3'b101 : 3'b000;
          3'b101: S <= !G ? 3'b110 : 3'b100;
          3'b110: S <= !G ? 3'b111 : 3'b111;
          3'b111: S <= !G ? 3'b000 : 3'b101;
        endcase
      end
  end
endmodule
```

TUTORIAL INTRODUTTIVO (16/4)

- Per rendere sintetizzabile il contatore, bisogna aggiungere un meccanismo per azzerare lo stato dall'esterno. A tal scopo si può aggiungere un *input* (net *Rst*) all'elenco degli input/output del modulo.
- La net *Rst* è quindi aggiunta anche alla *sensitivity list* dello statement *always* e ogni sua variazione (*posedge*) verrà quindi presa in considerazione.
- Se al momento dell'attivazione del blocco *always* il valore di *Rst* è uguale a 1, siamo di fronte ad una richiesta di azzeramento del registro che conserva lo stato.
- Se non è stato *Rst* ad attivare il blocco *always*, allora è stato sicuramente l'ingresso *Clk*. Il registro sarà perciò aggiornato con la regola vista in precedenza contenuta nello statement *case*.

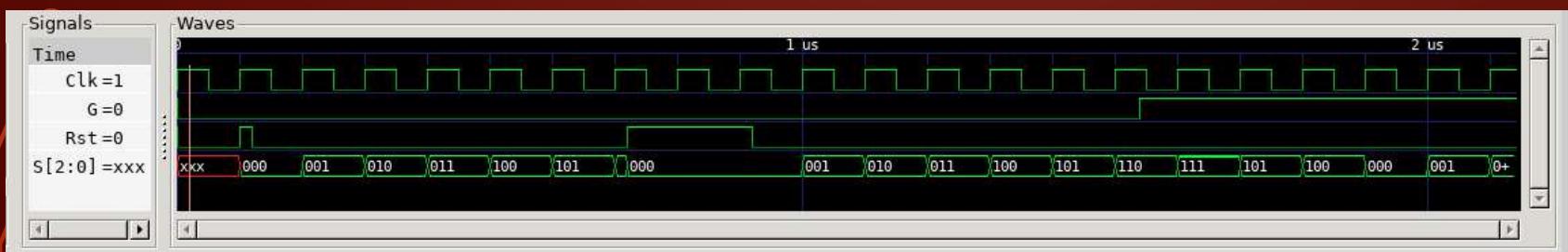
```
module counter( input Clk, input Rst, input G );
  reg [2:0] S;

  always @(* posedge Clk or posedge Rst )
    begin
      if ( Rst )
        begin
          S <= 3'b000;
        end
      else
        begin
          case ( S )
            3'b000: S <= !G ? 3'b001 : 3'b001;
            3'b001: S <= !G ? 3'b010 : 3'b011;
            3'b010: S <= !G ? 3'b011 : 3'b110;
            3'b011: S <= !G ? 3'b100 : 3'b010;
            3'b100: S <= !G ? 3'b101 : 3'b000;
            3'b101: S <= !G ? 3'b110 : 3'b100;
            3'b110: S <= !G ? 3'b111 : 3'b111;
            3'b111: S <= !G ? 3'b000 : 3'b101;
          endcase
        end
    end
endmodule
```

TUTORIAL INTRODUTTIVO (17)

Una volta aggiornate le test fixture per supportare il nuovo ingresso di Reset, è possibile eseguire la simulazione per verificare il corretto funzionamento del contatore. Come si può vedere nei diagrammi sottostanti, l'impulso Rst azzerà inizialmente il registro che partiva con un valore indefinito (xxx). Quando il Rst è rilasciato, i successivi impulsi di clock (Clk) fanno avanzare il conteggio. Un livello di Rst a 1 per un tempo abbastanza lungo da superare il periodo di qualche ciclo di clock, dimostra che l'ingresso Rst è prioritario, alla stregua di un reset asincrono applicato direttamente ai registri.

In ultimo, si può vedere che una variazione da zero ad uno dell'ingresso G, modifica il criterio di conteggio passando da Binario a codice Gray.



```
timescale 10ns/100ps

module tf;
  reg Mode;
  wire Cp;
  reg Reset;

initial begin
  $dumpvars( 0, tf );
  Mode = 0; #154
  Mode = 1; #60
  $finish;
end

initial begin
  Reset = 0; #10
  Reset = 1; #2
  Reset = 0; #60
  Reset = 1; #20
  Reset = 0;
end

clock Clk( .Out( Cp ) );
counter counter_1 (
  .Clk( Cp ),
  .Rst( Reset ),
  .G( Mode )
);
endmodule
```

Q&A

- Relatori:

Emanuel Conti (e.conti@embedded.sm)

Giuliano Cardinali (giulicard@gmail.com)

RIFERIMENTI BIBLIOGRAFICI

- Reti logiche e calcolatori
F. Luccio, L. Pagli – ISBN: 978-8833954870
- FPGAs: Instant Access
Clive Maxfield - ISBN: 978-0750689748
- Bebop to the Boolean Boogie: An Unconventional Guide to Electronics
Clive Maxfield - ISBN: 978-1856175074
- Logic Circuit Design: Selected Topics and Methods, Second Ed.
Shimon P. Vingron - ISBN: 978-3031406720
- The Verilog® Hardware Description Language, 5th Ed.
Donald E. Thomas, Philip R. Moorby - ISBN: 978-1402070896
- Digital design : with an introduction to the verilog hdl
M. Morris Mano, Michael D. Ciletti.—5th ed. - ISBN-13: 978-0132774208
- Digital Design Principles and Practices
John F. Wakerly - ISBN: 978-0134460093