

1 Lezione del 15-10-24

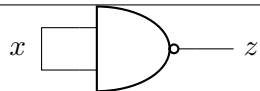
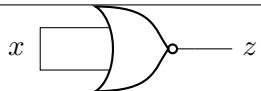
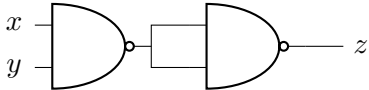
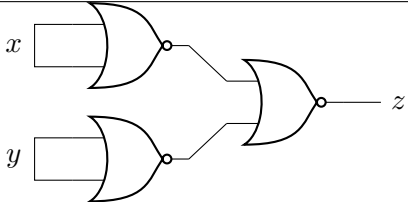
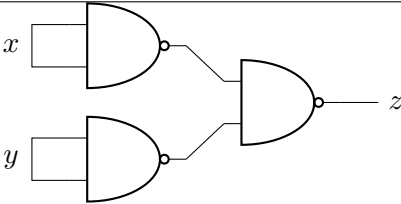
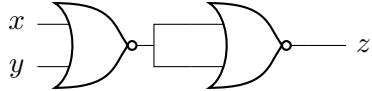
1.1 Porte logiche universali

Si dice che NAND e NOR sono **porte logiche universali**. Si possono realizzare AND, OR e NOT usando solo porte NAND o solo porte NOR.

Algebricamente, questo significa fare:

Porta	Realizzazione NAND	Realizzazione NOR
NOT	$x = x \cdot x \Rightarrow \bar{x} = \overline{x \cdot x}$	$x = x + x \Rightarrow \bar{x} = \overline{x + x}$
AND	$x \cdot y = \overline{(\overline{x \cdot y})}$	$x \cdot y = \overline{\bar{x} + \bar{y}}$ (de Morgan)
OR	$x + y = \overline{\bar{x} \cdot \bar{y}}$ (de Morgan)	$x + y = \overline{(\overline{x + y})}$

cioè collegare porte logiche fisiche nelle seguenti configurazioni:

Porta	Realizzazione NAND	Realizzazione NOR
NOT		
AND		
OR		

Potremmo sembrare che, se si poteva realizzare qualsiasi rete combinatoria con AND, OR e NOT su 2 livelli di logica, usando solo NAND o NOR dovremmo accontentarci di 4 livelli di logica (AND e OR richiedono di per sé una rete a 2 livelli di logica).

In verità, le porte NAND e NOR permettono di creare circuiti logici con gli stessi livelli di logica delle porte AND, OR e NOT.

1.1.1 Sintesi a porte NAND

Vediamo quindi il seguente algoritmo per la sintesi di un circuito con sole porte NAND:

Algoritmo 1 sintesi a porte NAND

Input: un circuito in forma SP

Output: una sintesi a porte NAND

Si sostituisce la porta OR con il suo equivalente a NAND

Si sostituisce ciascun AND con il suo equivalente a NAND

Si eliminano le coppie di NOT interne a cascata

Ignoriamo i NOT sull'ingresso, in quanto abbiamo visto sono effettivamente gratuiti. Abbiamo quindi che, rimuovendo le coppie NOT interni (creati da coppie di NAND con gli stessi input) ritorniamo in una forma a 2 livelli di logica.

Dal punto di vista algebrico si ha:

$$z = P_1 + P_2 + \dots + P_k = \overline{\overline{P_1 + P_2 + \dots + P_k}} = \overline{\overline{P_1} \cdot \overline{P_2} \cdot \dots \cdot \overline{P_k}}$$

dove il complemento superiore è l'ultima porta NAND (quella che sostituisce l'OR), e i singoli P_i complementati sono singole porte NAND (P_i è un prodotto, quindi $\overline{P_i}$ è una porta NAND).

1.1.2 Sintesi a porte NOR

Vediamo poi l'algoritmo per la sintesi di un circuito con sole porte NOR:

Algoritmo 2 sintesi a porte NOR

Input: un circuito in forma PS

Output: una sintesi a porte NOR

Si sostituisce la porta AND con il suo equivalente a NOR

Si sostituisce ciascun OR con il suo equivalente a NOR

Si eliminano le coppie di NOT interne a cascata

Anche qui ignoriamo i NOT sull'ingresso, per gli stessi motivi di prima, e rimuovendo le coppie NOT interni (creati da coppie di NAND con gli stessi input) ritorniamo nuovamente in una forma a 2 livelli di logica.

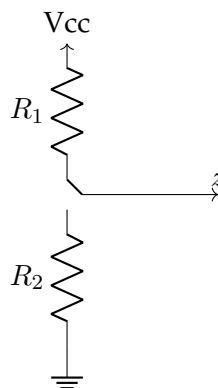
Dal punto di vista algebrico si ha:

$$z = S_1 \cdot S_2 \cdot \dots \cdot S_k = \overline{\overline{S_1 \cdot S_2 \cdot \dots \cdot S_k}} = \overline{\overline{S_1} + \overline{S_2} + \dots + \overline{S_k}}$$

dove il complemento superiore è l'ultima porta NOR (quella che sostituisce l'AND), e i singoli S_i complementati sono singole porte NOR (S_i è una somma, quindi $\overline{S_i}$ è una porta NOR).

1.2 Porte tri-state

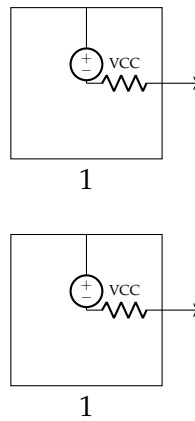
Fa comodo poter connettere insieme le uscite delle reti usando bus condivisi, cioè linee di ingresso-uscita. Abbiamo che l'uscita di una rete, dal punto di vista di una rete, corrisponde a un interruttore fra il Vcc (1 logico) o la massa (0 logico), cioè:



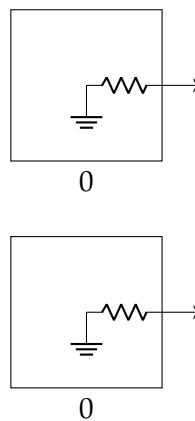
dove R_1 e R_0 sono incognite.

Quando vado a collegare più uscite sulla stessa linea possono crearsi più situazioni:

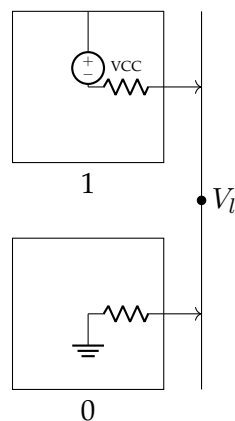
- **1 logici:** se ho due 1 logici, cioè due generatori di potenziale a V_{cc} , connessi sulla stessa linea, ho che la tensione sulla linea è sempre V_{cc} , quindi tutto ok:



- **0 logici:** allo stesso tempo, se ho due 0 logici, quindi due collegamenti a massa, sulla linea si avrà tensione nulla:



- **0 e 1 logici:** se collego un 1 logico e uno 0 logico alla stessa linea, ottengo effettivamente un partitore di tensione:

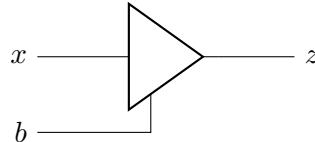


da cui ricavo:

$$V_l = \frac{R_0}{R_1 + R_0}$$

Notiamo soprattutto che se R_0 e R_1 sono molto piccoli, otteniamo correnti I molto grandi, che significa componenti bruciati.

Per risolvere il problema dato da 0 e 1 logici connessi sulla stessa linea, usiamo specifici apparecchi detti **porte tri-state**, che sono capaci di disconnettere fisicamente un'uscita da una linea condivisa. Si rappresentano come:



dove x è l'ingresso, z l'uscita, e b l'enabler. A $b = 1$ la porta si comporta come un'elemento neutro, mentre a $b = 0$ offre un'alta impedenza, effettivamente scollegando l'uscita. La tabella di verità corrispondente sarà:

b	x	z
1	0	0
1	1	1
0	-	Hi-Z

Notiamo che il valore Hi-Z (alta impedenza) non è un valore logico: ciò che esce da una porta in stato Hi-Z viene interpretato come un filo staccato dal resto della rete. Ogni porta logica gestisce poi questa situazione secondo le sue specifiche di realizzazione, restando comunque attaccata sia a V_{cc} che a massa, e quindi non in uno stato HiZ.

In Verilog, possiamo modellizzare una porta tristate come:

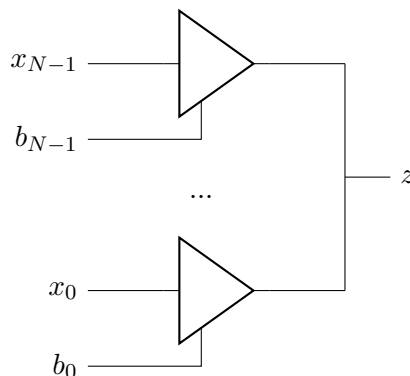
```

1 // una porta tri-state con @x variabile di ingresso, @z variabile di
2 // uscita e @b variabile di controllo attiva alta
3 module tristate(x, b, z);
4     input x, b;
5     output z;
6
7     assign z = b ? x : 'BZ;
8 endmodule

```

1.2.1 Multiplexer decodificati

Un componente realizzato attraverso le porte tri-state è il multiplexer decodificato:



Questo componente offre alta indipendenza a tutte le variabili x_i in ingresso tranne una, quella all'indice j , selezionata attraverso una variabile di comando b_j .

In Verilog, possiamo realizzare un multiplexer decodificato sfruttando la definizione già data di decodificatore. Vediamo quindi multiplexer decodificati da 2 a 1:

```

1 // un multiplexer decodificato da 2 a 1 che prende @x1_x0 come
2 // ingresso, @b0 come comando e @z0 come uscita
3 module b2to1_decoded_muxer(x1_x0, b0, z0);
4     inout [1:0] x1_x0;
5     input b0;
6     output z0;
7
8     wire [1:0] s1_s0;
9
10    b1to2_decoder b1to2(b0, s1_s0);
11
12    assign z0 = s1_s0[1] & x1_x0[1] | s1_s0[0] & x1_x0[0];
13    assign x1_x0[1] = s1_s0[1] ? x1_x0[1] : 1'BZ;
14    assign x1_x0[0] = s1_s0[0] ? x1_x0[0] : 1'BZ;
15 endmodule

```

da 4 a 1:

```

1 // un multiplexer decodificato da 4 a 1 che prende @x3_x0 come
2 // ingresso, @b1_b0 come comando e @z0 come uscita
3 module b4to1_decoded_muxer(x3_x0, b1_b0, z0);
4     input [3:0] x3_x0;
5     input [1:0] b1_b0;
6     output z0;
7
8     wire [3:0] s3_s0;
9
10    b2to4_decoder b2to4(b1_b0, s3_s0);
11
12    assign z0 = s3_s0[3] & x3_x0[3] | s3_s0[2] & x3_x0[2]
13              | s3_s0[1] & x3_x0[1] | s3_s0[0] & x3_x0[0];
14    assign x3_x0[3] = s3_s0[3] ? x3_x0[3] : 1'BZ;
15    assign x3_x0[2] = s3_s0[2] ? x3_x0[2] : 1'BZ;
16    assign x3_x0[1] = s3_s0[1] ? x3_x0[1] : 1'BZ;
17    assign x3_x0[0] = s3_s0[0] ? x3_x0[0] : 1'BZ;
18 endmodule

```

da 8 a 1:

```

1 // un multiplexer decodificato da 8 a 1 che prende @x7_x0 come
2 // ingresso, @b2_b0 come comando e @z0 come uscita
3 module b8to1_muxer_d(x7_x0, b2_b0, z0);
4     input [7:0] x7_x0;
5     input [2:0] b2_b0;
6     output z0;
7
8     wire [7:0] s7_s0;
9
10    b3to8_decoder b3to8(b2_b0, s7_s0);
11
12    assign z0 = s7_s0[7] & x7_x0[7] | s7_s0[6] & x7_x0[6]
13              | s7_s0[5] & x7_x0[5] | s7_s0[4] & x7_x0[4]
14              | s7_s0[3] & x7_x0[3] | s7_s0[2] & x7_x0[2]
15              | s7_s0[1] & x7_x0[1] | s7_s0[0] & x7_x0[0];
16    assign x7_x0[7] = s7_s0[7] ? x7_x0[7] : 1'BZ;
17    assign x7_x0[6] = s7_s0[6] ? x7_x0[6] : 1'BZ;
18    assign x7_x0[5] = s7_s0[5] ? x7_x0[5] : 1'BZ;

```

```

19 assign x7_x0[4] = s7_s0[4] ? x7_x0[4] : 1'BZ;
20 assign x7_x0[3] = s7_s0[3] ? x7_x0[3] : 1'BZ;
21 assign x7_x0[2] = s7_s0[2] ? x7_x0[2] : 1'BZ;
22 assign x7_x0[1] = s7_s0[1] ? x7_x0[1] : 1'BZ;
23 assign x7_x0[0] = s7_s0[0] ? x7_x0[0] : 1'BZ;
24 endmodule

```

e infine da 16 a 1:

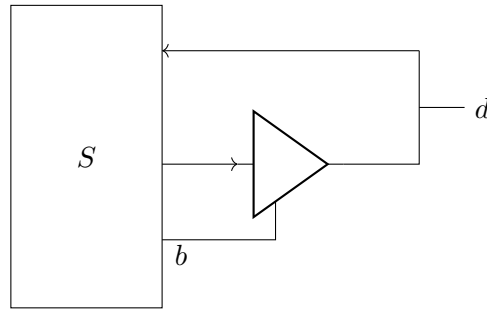
```

1 // un multiplexer decodificato da 16 a 1 che prende @x15_x0 come
2 // ingresso, @b3_b0 come comando e @z0 come uscita
3 module b16to1_muxer_d(x15_x0, b3_b0, z0);
4     input [15:0] x15_x0;
5     input [3:0] b3_b0;
6     output z0;
7
8     wire [15:0] s15_s0;
9
10    b4to16_decoder b3to8(b3_b0, s15_s0);
11
12    assign z0 = s15_s0[15] & x15_x0[15] | s15_s0[14] & x15_x0[14]
13               | s15_s0[13] & x15_x0[13] | s15_s0[12] & x15_x0[12]
14               | s15_s0[11] & x15_x0[11] | s15_s0[10] & x15_x0[10]
15               | s15_s0[9] & x15_x0[9] | s15_s0[8] & x15_x0[8]
16               | s15_s0[7] & x15_x0[7] | s15_s0[6] & x15_x0[6]
17               | s15_s0[5] & x15_x0[5] | s15_s0[4] & x15_x0[4]
18               | s15_s0[3] & x15_x0[3] | s15_s0[2] & x15_x0[2]
19               | s15_s0[1] & x15_x0[1] | s15_s0[0] & x15_x0[0];
20 assign x15_x0[15] = s15_s0[15] ? x15_x0[15] : 1'BZ;
21 assign x15_x0[14] = s15_s0[14] ? x15_x0[14] : 1'BZ;
22 assign x15_x0[13] = s15_s0[13] ? x15_x0[13] : 1'BZ;
23 assign x15_x0[12] = s15_s0[12] ? x15_x0[12] : 1'BZ;
24 assign x15_x0[11] = s15_s0[11] ? x15_x0[11] : 1'BZ;
25 assign x15_x0[10] = s15_s0[10] ? x15_x0[10] : 1'BZ;
26 assign x15_x0[9] = s15_s0[9] ? x15_x0[0] : 1'BZ;
27 assign x15_x0[8] = s15_s0[8] ? x15_x0[0] : 1'BZ;
28 assign x15_x0[7] = s15_s0[7] ? x15_x0[1] : 1'BZ;
29 assign x15_x0[6] = s15_s0[6] ? x15_x0[1] : 1'BZ;
30 assign x15_x0[5] = s15_s0[5] ? x15_x0[1] : 1'BZ;
31 assign x15_x0[4] = s15_s0[4] ? x15_x0[0] : 1'BZ;
32 assign x15_x0[3] = s15_s0[3] ? x15_x0[0] : 1'BZ;
33 assign x15_x0[2] = s15_s0[2] ? x15_x0[1] : 1'BZ;
34 assign x15_x0[1] = s15_s0[1] ? x15_x0[0] : 1'BZ;
35 assign x15_x0[0] = s15_s0[0] ? x15_x0[0] : 1'BZ;
36 endmodule

```

1.2.2 Linea di ingresso/uscita

Si usano le porte tri-state per permettere a componenti di comunicare su linee di ingresso/uscita, ad esempio con la memoria. In questo caso, si biforca la linea, ammettendo la linea in entrata così com'è, e mettendo una porta tri-state nella linea in uscita.



Così, quando il componente S vuole comunicare con l'esterno, imposta l'enabler b a 1 e mette sulla linea d ciò che vuole comunicare. Altrimenti tiene b a 0 e ascolta ciò che arriva su d .

Se il componente S comunica con un altro componente T , questi dovranno impostare alternativamente i loro enabler b_S e b_T a 1 e 0, scambiandosi messaggi sulla linea d .

Una struttura di questo tipo, detta anche "forchetta", può essere realizzata in Verilog come segue:

```

1 // una forchetta sulla variabile bidirezionale @x controllata da @b,
2 // su cui il modulo scrive quanto rileva nel filo @z
3 module tristate(x);
4     inout x;
5     wire z;
6     wire b;
7
8     assign x = b ? z : 1'BZ;
9 endmodule

```

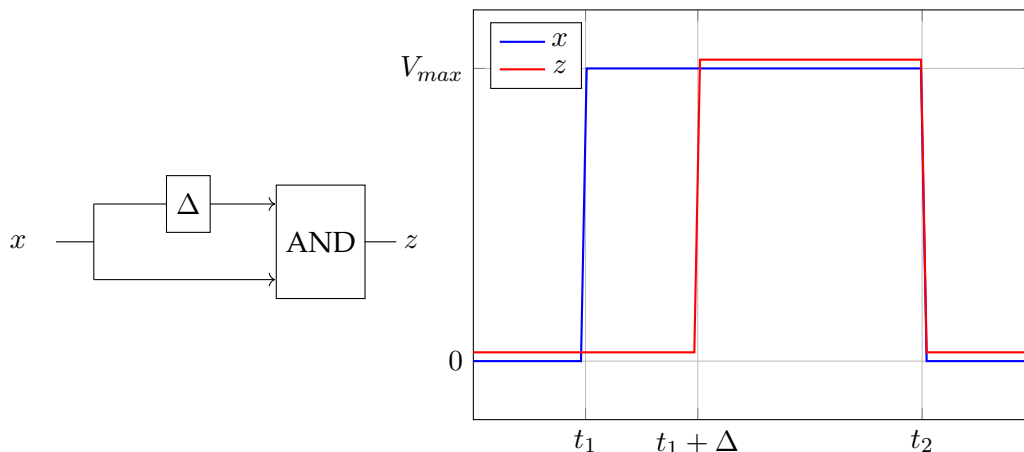
1.3 Circuiti di ritardo e formatori di impulso

A volte bisogna trattare di **segnali**. In questo caso si usano gli elementi neutri Δ (realizzati spesso con numeri pari di invertitori), che sappiamo porre un **ritardo simmetrico** agli ingressi, dove simmetrico significa identico sulle transizioni $0 \rightarrow 1$ e $1 \rightarrow 0$.

Potrebbe essere utile avere circuiti con ritardi **asimmetrici**, cioè variabili sulle transizioni $0 \rightarrow 1$ e $1 \rightarrow 0$. Indichiamo questi componenti come Δ^+ .

1.3.1 Circuito di ritardo sul fronte di salita

Collegando un neutro Δ assieme al segnale stesso ad una porta AND, si ottiene un'andamento del tipo:



Nello specifico, transizionando da $1 \rightarrow 0$, si ha che il primo ingresso che va a 0 porta a 0 l'uscita. C'è un ritardo piccolo da parte della porta AND. Quando invece si transiziona da $0 \rightarrow 1$, si ha che il secondo ingresso che va a 1 (quello che passa da Δ) porta a 1 l'uscita. C'è un ritardo grande da parte del Δ e della porta AND.

La sintesi in Verilog di un circuito di questo tipo è la seguente:

```

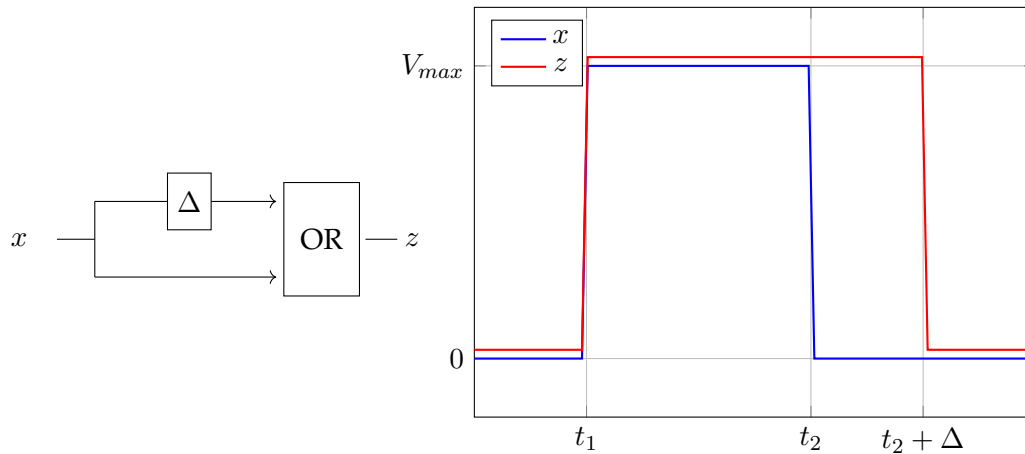
1 // un generatore di ritardo sul fronte di salita con variabile di
2 // ingresso @x e variabile di uscita ritardata @z
3 module posedge_delay_buffer(x, z);
4     input x;
5     output reg z;
6
7     always @(posedge x) begin
8         #5 z = x;
9     end
10    always @(negedge x) begin
11        z = x;
12    end
13 endmodule
14
15 // implementazione a porte logiche
16 module posedge_delay_buffer_p(x, z);
17     input x;
18     output z;
19
20     reg y;
21     initial y = 0;
22
23     always @(posedge x or negedge x) begin
24         #5 y = x;
25     end
26
27     assign z = y & x;
28 endmodule

```

Si nota che questa descrizione (e le seguenti) sono state realizzate in due modi: il primo prevede l'uso di blocchi **always** con statement bloccanti o non bloccanti, mentre il secondo usa la "sintesi a porte logiche" presentata sopra. Entrambe le descrizioni, però, hanno scopo dimostrativo in quanto gli statement di ritardo (**#5**) hanno valenza solo durante la simulazione della rete e non in fase di sintesi.

1.3.2 Circuito di ritardo sul fronte di discesa

Allo stesso modo, collegando un neutro Δ assieme al segnale stesso ad una porta OR, si ottiene un'andamento del tipo:



Nello specifico, transizionando da $0 \rightarrow 1$, si ha che il primo ingresso che va a 1 porta a 1 l'uscita. C'è un ritardo piccolo da parte della porta OR. Quando invece si transiziona da $1 \rightarrow 0$, si ha che il secondo ingresso che va a 0 (quello che passa da Δ) porta a 0 l'uscita. C'è un ritardo grande da parte del Δ e della porta OR.

La sintesi in Verilog di un circuito di questo tipo è la seguente:

```

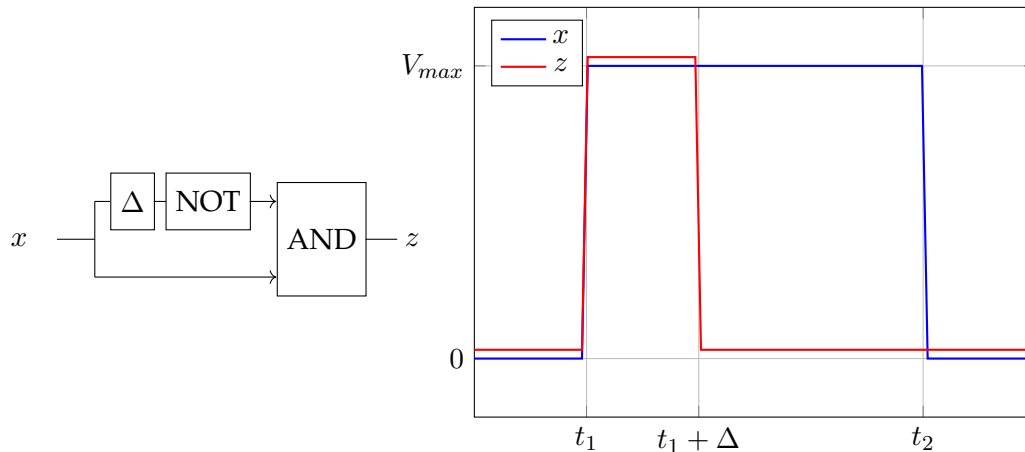
1 // un generatore di ritardo sul fronte di discesa con variabile di
2 // ingresso @x e variabile di uscita ritardata @z
3 module negedge_delay_buffer(x, z);
4     input x;
5     output reg z;
6
7     always @(posedge x) begin
8         z <= x;
9     end
10    always @(negedge x) begin
11        #5 z <= x;
12    end
13 endmodule
14
15 // implementazione a porte logiche
16 module negedge_delay_buffer_p(x, z);
17     input x;
18     output z;
19
20     reg y;
21     initial y = 0;
22
23     always @(posedge x or negedge x) begin
24         #5 y = x;
25     end
26
27     assign z = y | x;
28 endmodule

```

1.3.3 Formatore di impulso sul fronte di salita

I formatori di impulso sono reti combinatorie che generano in uscita un **impulso** di durata nota. Si indicano con P^+ .

Si crea un formatore di impulso sul fronte di salita collegando la negazione di un Δ e il segnale stesso ad una porta AND, cioè:



Nello specifico, transizionando da $0 \rightarrow 1$, si ha che il segnale va a 1, attivando la AND (l'ingresso dalla NOT era già attivo). Dopo il ritardo Δ , NOT torna a 0, e quindi l'uscita della AND va a 0. Si ha quindi un'impulso di durata del ritardo Δ . Transizionando da $1 \rightarrow 0$, invece, si ha che il segnale "ancora" istantaneamente l'uscita della AND a zero, ergo non si hanno altri artefatti.

La sintesi in Verilog di un circuito di questo tipo è la seguente:

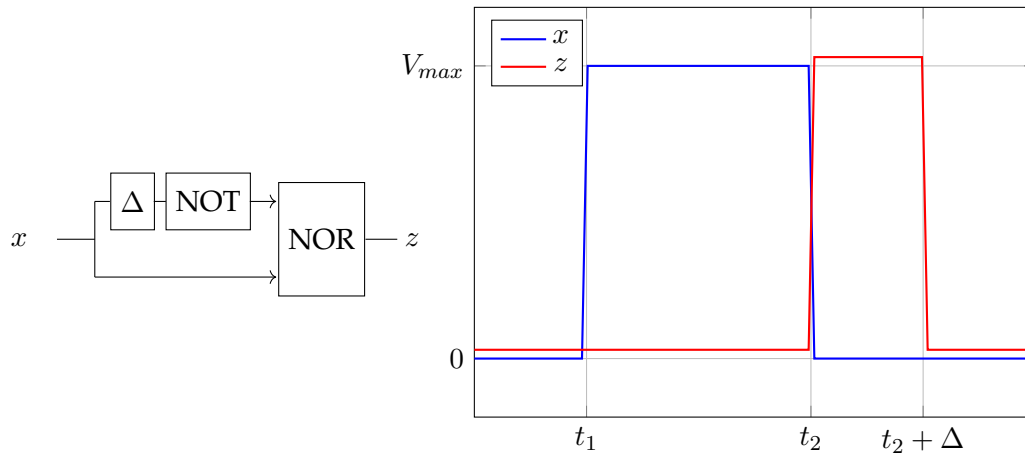
```

1 // generatore di impulso sul fronte di salita con variabile di
2 // ingresso @x e variabile d'impulso @z
3 module posedge_impulse_generator(x, z);
4     input x;
5     output reg z;
6
7     initial z = 0;
8
9     always @(posedge x) begin
10         z = 1;
11         #5;
12         z = 0;
13     end
14 endmodule
15
16 // implementazione a porte logiche
17 module posedge_impulse_generator_p(x, z);
18     input x;
19     output z;
20
21     reg y;
22     initial y = 0;
23
24     always @(posedge x or negedge x) begin
25         #5 y = x;
26     end
27
28     assign z = ~y & x;
29 endmodule

```

1.3.4 Formatore di impulso sul fronte di discesa

Si crea un formatore di impulso sul fronte di discesa collegando la negazione di un Δ e il segnale stesso ad una porta NOR, cioè:



Nello specifico, transizionando da $0 \rightarrow 1$, si ha che il segnale va a 1, ergo la NOR resta a 0 (l'ingresso dalla NOT era già attivo, e due ingressi attivi sono sempre 0 della NOR). Dopo il ritardo Δ , l'uscita della NOT torna a 0, così che quando si stacca il segnale, per una durata Δ entrambe le linee in entrata alla NOR vanno a 0, e quindi questa va a 1. Dopo il ritardo Δ , NOT torna a 0, e quindi l'uscita della AND va a 0. Si ha quindi, ancora una volta, un'impulso di durata del ritardo Δ .

La sintesi in Verilog di un circuito di questo tipo è la seguente:

```

1 // generatore di impulso sul fronte di salita con variabile di
2 // ingresso @x e variabile d'impulso @z
3 module negedge_impulse_generator_x(x, z);
4     input x;
5     output reg z;
6
7     always @(negedge x) begin
8         z = 1;
9         #5;
10        z = 0;
11    end
12 endmodule
13
14 // implementazione a porte logiche
15 module negedge_impulse_generator(x, z);
16     input x;
17     output z;
18
19     reg y;
20     initial y = 0;
21
22     always @(posedge x or negedge x) begin
23         #5 y = x;
24     end
25
26     assign z = ~(~y | x);
27 endmodule

```