Basi del linguaggio Verilog

▼ Creatori originali: @Giuseppe Calvello, @Gianbattista Busonera

Introduzione

Verilog è uno dei tre principali linguaggi utilizzati per modellare l'hardware. Essendo basato su C, risulta essere più semplice da comprendere e apprendere di altri linguaggi, ma risulta meno tipizzato.

Inoltre, è un linguaggio intrinsecamente parallelo per modellare il parallelismo intrinseco dell'hardware.

Con Verilog si possono modellare sistemi a diversi livelli di astrazione:

1. modellazione a livello di porte logiche:
    a. livello più basso, che utilizza porte elementari Verilog;
    b. tutte le istruzioni sono simultanee;
    c. l'ordine delle istruzioni NON è importante.
2. modellazione strutturale:
    a. descrive la struttura di un circuito con i moduli, a diversi livelli;
    b. tutte le istruzioni sono simultanee;
    c. l'ordine delle istruzioni NON è importante.
3. modellazione flusso di dati (dataflow) o register transfer level (RTL):
    a. descrive il flusso di dati tra ingressi e uscite, utilizzando istruzioni di assegnazione simultanea;
    b. l'assegnazione è continua;
    c. tutte le istruzioni sono simultanee;
    d. l'ordine delle istruzioni NON è importante.
4. modellazione comportamentale:
    a. descrive ciò che fa il circuito utilizzando blocchi procedurali d'istruzioni;
    b. le assegnazioni sono procedurali, e quindi le istruzioni, vengono eseguite in sequenza;
    c. l'ordine delle istruzioni è importante.

⚠️ Nello stesso progetto possono essere utilizzati modelli di diverso tipo.

Utilizzo

Verilog può essere utilizzato per:

- Simulazione;
- Sintesi;
- Verifica formale.

Input e Output

In Verilog esistono diversi tipi di dichiarazioni delle “porte” di ingresso/uscita in un modulo:

- Input: porta di ingresso;
- Output: porta di uscita;
  - Signed: va esplicitamente dichiarato! È, però, necessario solo nelle moltiplicazioni;
  - Unsigned: scelta di default.
- Inout: porta di ingresso e uscita.

⚠️ Di base, Verilog fa le operazioni in CA2 e, quindi, le operazioni di somma e sottrazione non subiscono variazioni in caso di “signed” o “unsigned”, ma solo quelle di moltiplicazione.

Tipi di dato

Verilog ha due tipi principali di dati: `net` e `variabile`.

net

I dati di tipo `net` permettono connessioni tra le parti di un progetto, come fossero fili.

Un esempio di dato `net` è il tipo `wire` , il quale:

- permette di trasferire un valore tra i componenti;
- può essere pilotato da un’assegnazione continua;
- non permette un’assegnazione procedurale;
- non può memorizzare un valore;

- non può memorizzare uno stato, e deve quindi essere costantemente pilotato, altrimenti il valore viene perso;

```
// definizione di un tipo di dato wire
wire <nome_wire>
```

Esistono altri tipi di dato `net`, quali `input` .

variabile

I dati di tipo `variabile` permettono di memorizzare i valori dei dati.

Un esempio di dato `variabile` è il tipo `reg` , il quale:

- può memorizzare il valore di un'assegnazione procedurale;
- ricorda l'ultimo valore assegnato;
- non può essere pilotato da un'assegnazione continua;
- non corrisponde necessariamente a un registro nell'hardware.

```
// definizione di un tipo di dato reg
reg <nome_reg>
```

Esistono altri tipi di dato `variabile`, quali `intero` e `genvar`.

> ⚠️ Qualsiasi tipo di dato può essere utilizzato come ingresso (RHS) in un'espressione, indipendentemente dal suo genere.

Tipi di dato aggregato

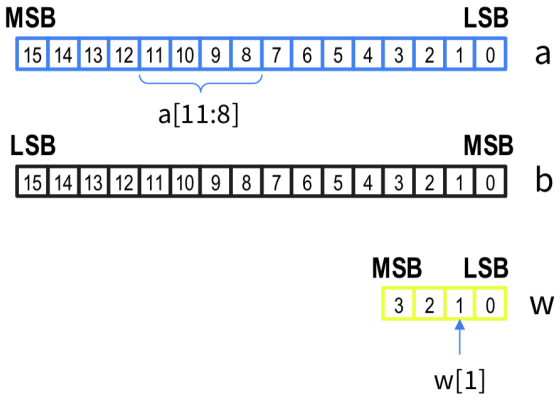
Vettori di bit

Sia che si utilizzi un tipo di dato "net" o un tipo di dato "variabile", possiamo specificare il parallelismo di una data connessione (net) o di una data variabile (reg) tramite:

Un vettore di bit, o bus, è una dichiarazione multi-bit che utilizza un singolo nome. Esso viene specificato come un intervallo `[msb:lsb]` .

Prendono anche il nome di “vectors” (o packed arrays).

Esistono anche gli “unpacked arrays”, utilizzati come classici array, non approfonditi.



Esempio di un vettore di bit

▼ Esempi - Vettori di bit

Definiamo alcuni esempi di definizione e di accesso ai vettori di bit:

```
input [15:0] a; // a è un vettore d'ingresso a 16 bit → tipo di dato net
                // serve infatti per poter collegare l'ingresso di un
                // componente con l'uscita di un altro

reg [0:15] b;   // il bit 0 è il MSB; il vettore b è di tipo reg
wire [3:0] w;   // il bit 3 è il MSB; il vettore w è di tipo wire
```

```
w[1];        // selezione del secondo bit del vettore w
//^ si prende il bit numero 1

a[11:8];     // selezione dell'intervallo [11:8] di 4 bit
//^ si prendono i bit 11, 10, 9, 8
```

Si noti come l'intervallo di selezione deve essere coerente con la dichiarazione del vettore.

Matrici in Verilog

Le matrici possono esser definite specificando un intervallo di indirizzi, è necessario fornire gli indici superiore ed inferiore , dopo il nome dell’identificatore.

Può essere formata da scalari (0 o 1) o vettori di bit (matrice di vettori). Possono avere qualsiasi numero di dimensione.

⚠️ La sintassi è la seguente:

```
<tipo> [intervallo_colonne] nome_matrice [intervallo_righe];
```

▼ Esempi - Matrici

```
// per comprendere bene da dove si parte per la creazione di matrici,
// conviene creare inizialmente una matrice monodimensionale (cioè, un vettore)
reg a[15:0] // la matrice monodimensionale (è un vettore)
// abbiamo quindi 16 elementi da (implicitamente) 1 bit, ed è simile a scrivere:
// reg [0:0] a [15:0]
// otteniamo così 16 "righe" fatte da reg di 1 bit ciascuna, cioè da scalari.

wire [7:0] b[15:0] // b matrice di 16 elementi di tipo wire ognuno con
                   // dimensione (parallelismo) 8 bit

wire [7:0] c[3:0][3:0] // c è matrice 4×4, 16 elementi di dimensione 8 bit
```

Spesso una matrice di registri rappresenta una memoria.

```
reg [7:0] mem [1023:0] // mem è una matrice di 1024 elementi, ciascuno da 8 bit

mem[i]          // indica un byte specifico della memoria (indirizzo i)
mem[i][j]       // indica un bit specifico di un byte specifico
                // (indirizzo i,j)
                // posso dichiarare anche matrici
```

Valori logici in Verilog

Verilog utilizza quattro valori di base, evidenziati nella tabella.

Simbolo	Significato	Descrizione
`0`	Basso logico (Low)	Rappresenta il livello logico basso, come in digitale “spento”.
`1`	Alto logico (High)	Rappresenta il livello logico alto, come in digitale “acceso”
`x` o `X`	Indefinito (Unknown)	Valore sconosciuto o conflitto (es. due inverter che pilotano valori diversi). E’ utile in simulazione per rilevare errori o stati non inizializzati.

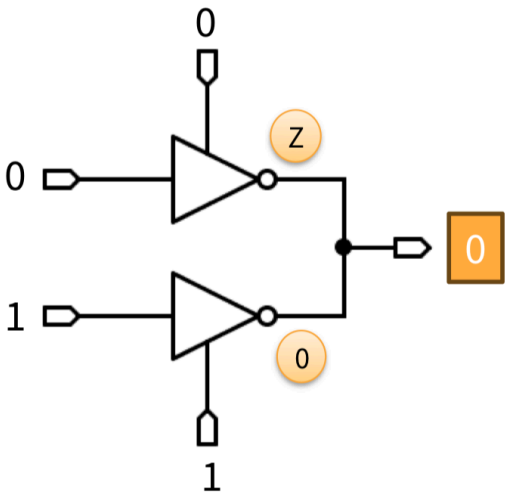
Simbolo	Significato	Descrizione
`z` o `Z`	Alta impedenza (High Impedance)	Stato di disconnessione del segnale (es. quando una linea è “libera”). Viene usato per linee condivise, ad esempio nei bus tri-state, dove solo un dispositivo alla volta guida la linea.

Esempio delle combinazioni logiche

Nel caso di un valore `z` come in figura in uscita da un inverter di tipo tri-state non abilitato in combinazione a un altro inverter di tipo tri-state (abilitato) produce sul nodo d’uscita un valore corretto (cioè 0).

Cosa succede nel caso di due uscite collegate insieme? Otteniamo la seguente tabella:

	0	1	X
0	0	X	X
1	X	1	X
X	X	X	X
Z	0	1	X



Esempio di una delle combinazioni, utilizzando delle porte tri-state

Costanti in Verilog

Le costanti sono valori letterali usati per rappresentare numeri (binari, decimali, esadecimali, ecc.) nel codice. Sono utili per assegnare valori a segnali o registri e per descrivere maschere di bit, pattern, indirizzi, ecc.

Sintassi delle costanti in Verilog

Definiamo la sintassi generale delle costanti:

```
<numero_bit>'<base><valore>
```

- `numero_bit` indica la dimensione in bit della costante dichiarata;
  - se omesso, la dimensione è 32 bit.
- `'` separatore, sempre necessario;
- `base` indica la base numerica in cui è espresso il valore della costante;

Simbolo	Base
`b`	binaria
`o`	ottale
`d`	decimale
`h`	esadecimale (hex)

    - se la base non è specificata, si assume sia in base decimale!
- `valore` è il numero nella base specificata;
- è possibile utilizzare il carattere `_` per migliorare la leggibilità di gruppi di bit.

```
8'b10111011     // senza underscore
8'b1011_1011    // con underscore, più leggibile

// entrambi rappresentano esattamente lo stesso valore, nel secondo caso
// sono stati divisi i bit in gruppi di 4 per migliorarne la leggibilità
```

▼ Esempi - Costanti

```
// binario su 8 bit
8'b1011_1011

// esadecimale su 32 bit (se la dimensione della costante
// non è specificata, la costante è composta da 32 bit)
'hA3F0

// ottale su 16 bit
16'o56377

// decimale su 32 bit
32'd999 // uguale a 999
```

Macro in Verilog

Le macro in Verilog sono direttive definite dal preprocessore, simili a quelle di C e C++, e servono a definire costanti, frammenti di codice riutilizzabili o condizionali di compilazione.

Vengono interpretate prima della compilazione vera e propria.

Definizione di una macro

La direttiva `` `define `` serve a dichiarare una macro, ovvero un'etichetta che verrà sostituita dal testo associato in fase di pre-processamento.

> ⚠️ La virgoletta (`` ` ``, backtick) non è un apostrofo, ma su Windows è ottenuto tramite `ALT+96` .

```
`define NOME_MACRO valore_macro // in C è #define NOME_MACRO valore
```

- `NOME_MACRO` definisce il nome della macro, solitamente a caratteri maiuscoli;
- `valore_macro` è una qualsiasi sequenza di caratteri, e molto spesso una costante numerica.

Ovunque venga utilizzato il valore `'NOME_MACRO`, esso verrà sostituito dal valore associato `valore_macro`.

▼ Esempio - Definizione di una macro

```
// associa al nome XSIZE il valore decimale 96
'define XSIZE 8'd96

wire ['XSIZE-1:0] xpos; // 'XSIZE prende il valore di XSIZE, cioè 96
```

For, if, case

Tali sintassi ricordano estremamente la sintassi del C, sostituituendo "{" con `begin` e "}" con `end` :

- per il for:

```
genvar i, j; // si utilizza per ciclare
for(i = 0; i < N; i++)
   istruzione1;

for(j = 0; j < N; j++) begin
   istruzione1;
   ...
   istruzioneK;
end
```

- per gli if:

```
output [1:0] risultato;
input a, b;
```

```
if(sel == 2'b00) // se il selettore è uguale a 00 in base 2 (su due bit)
   risultato = a+b;
else if (sel == 2'b01) begin
   risultato = a-b;
end
else if (sel == 2'b10)
   risultato = a&b;
else
   risultato = a|b;
```

- per i case:

```
case (sel)
   2'b00: begin
      risultato = a + b;
   end
   2'b01: risultato = a - b;
   2'b10: risultato = a & b;
   default: risultato = 16'bX; // mette tutto in unknown ... come don't care
endcase
```

Si noti come non è necessario utilizzare `begin` ed `end` nel caso di una sola istruzione.

Tipi di modellazione

Modellazione strutturale

Descrive la struttura gerarchica del circuito tramite istanze di moduli (che possono a loro volta essere RTL, gate-level, etc.), è simile a disegnare uno schematico, ma con una descrizione testuale.

L'elemento base è `module .... endmodule` per ogni blocco, le cui istanze si collegano con net (tipicamente `wire`).

```
module <nome> ([lista delle porte]);
   // contenuti del modulo
endmodule

// si noti come un modulo può omettere la
// lista delle porte
module <nome>;
```

```
endmodule
```

Tutte le connessioni sono simultanee, e quindi l'ordine delle istanze non conta.

Il loro uso tipico è:

- integrare blocchi IP (Intellectual Property block), ovvero blocchi di logica già progettati, testati ed ottimizzati, che è possibile comprare, licenziare o riutilizzare all'interno di un progetto invece che implementarlo da zero;
- costruire sistemi gerarchici (per esempio, CPU + periferiche).

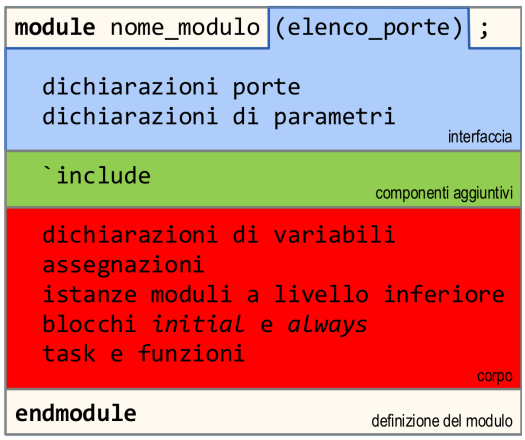
Nella pratica moderna, il 90% del codice è scritto in RTL o dataflow, per poi lasciare al sintetizzatore la generazione gate-level finale. Si usa, invece, la modellazione strutturale per mettere insieme i vari moduli.

Definizione di un modulo

Un modulo è un componente riutilizzabile. In Verilog, ogni blocco di circuito è racchiuso tra le parole chiave `module` ed `endmodule` .

All'interno del modulo si dichiarano:

- elenco degli ingressi e delle uscite (le uscite vanno inserite prima per convenzione);
- le caratteristiche delle porte (ingressi e uscite, eventualmente signed) e il loro parallelismo;
- eventuale dichiarazione di parametri (costanti di cui possiamo modificare il valore), che ci consentono di rendere parametrico il nostro circuito;
- tramite la parola chiave `` `include `` possiamo inserire nel nostro modulo altri componenti definiti in precedenza;
- segue la parte più corposa del nostro modulo, che analizzeremo man mano nella trattazione.



Visualizzazione della struttura interna di un modulo

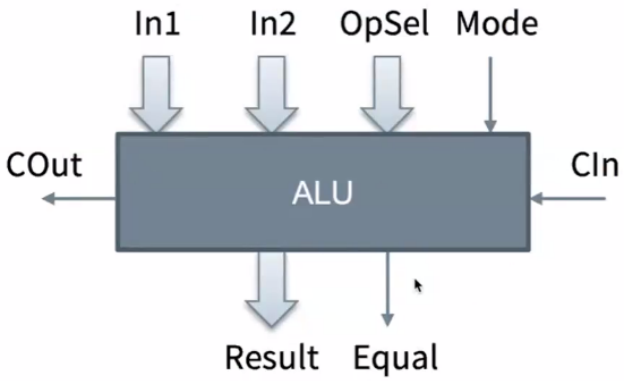
```
module  nome_modulo
        #(parametri opzionali)   // facoltativo: parametri generici
        (elenco_porte);          // porte fra parentesi tonde
    /* dichiarazioni */
    // 1. dichiarazione delle porte:
    //    input, output (signed) , inout(+ eventuale larghezza bus)
```

```
    // 2. dichiarazione di segnali locali (wire, reg, logic, ecc.)
    // 3. descrizione della logica con:
    //    - primitive di porta          (gate-level)
    //    - assign continui             (data-flow / RTL)
    //    - blocchi always/initial      (behavioral / RTL)
endmodule
```

▼ Esempio - Interfaccia del modulo ALU

Si progetti, tramite Verilog, l'interfaccia del modulo ALU in figura.

Ipotizziamo che questa ALU possa fare solo operazioni di somma e sottrazione e che gli operandi `In1` e `In2` siano su 3 bit.



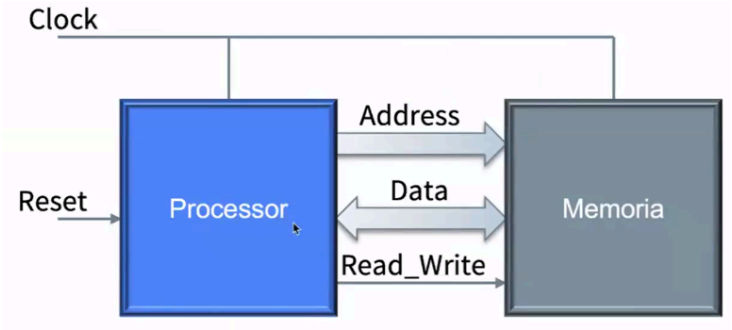
Visualizzazione di un modulo ALU

```
module ALU (Result, COut, Equal,        // prima le uscite, per convenzione
                    In1, In2, OpSel, CIn, Mode); // seguite dagli ingressi.
                                                 // si dichiarano esplicitamente il
                                                 // tipo delle "porte" (ingressi/uscite)
                                                 // e il parallelismo

output [4:0] Result; // porta d'uscita Result su 4 bit
                                 // in caso di somme di numeri a 3 bit potremmo
                                 // ottenere un numero su 4 bit

output COut;      // carry out
output Equal;     // se Equal = 1 ⇒ In1 = In2
input [2:0] In1;  // primo operando
input [2:0] In2;  // secondo operando
input [2:0] OpSel; // ipotizziamo ci siano 8 operazioni possibili
input CIn;
input Mode;       // modalità aritmetica (1) o logica (se 0)
// FINE INTERFACCIA
...
endmodule
```

▼ Esempio - Interfaccia di un processore

Si progetti, tramite Verilog, l'interfaccia del modulo Processore in figura.

Tale processore ha la capacità di leggere o scrivere in memoria, ed è comandato da un clock.



Visualizzazione di un modulo Processore

```
module Processor (Read_Write, Data, Clock, Reset, Address);

output Read_Write;
output [19:0] Address;
inout [15:0] Data; // è sia un input che un output!
                   // Il processore può leggere o scrivere dati in/da memoria
input Clock;
input Reset;
// FINE INTERFACCIA
...
endmodule
```

⚠ Normalmente, le porte di tipo `inout` dovrebbero essere pilotate con porte tri-state, in modo da evitare la creazione di conflitti.

Modellazione a livello di porta logica (GATE-LEVEL)

La modellazione a livello di porta logica rappresenta il circuito come un'interconnessione esplicita di porte elementari (AND, OR, XOR, NAND, flip-flop, ecc.). Ogni istanza di porta è contemporanea, e si ha quindi del parallelismo.

Usi tipici includono:

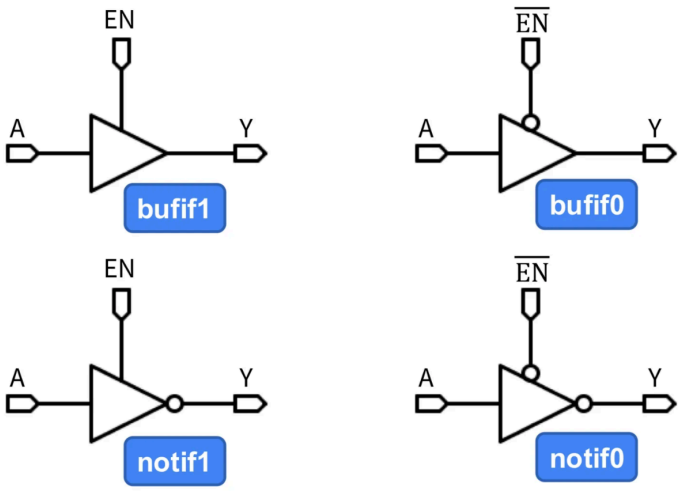
- checks post-sintesi;
- netlist generata dai tool;
- esercizi per piccole logiche.

Simula esattamente la rete fisica, e può includere ritardi di propagazione.

Porte logiche elementari in Verilog

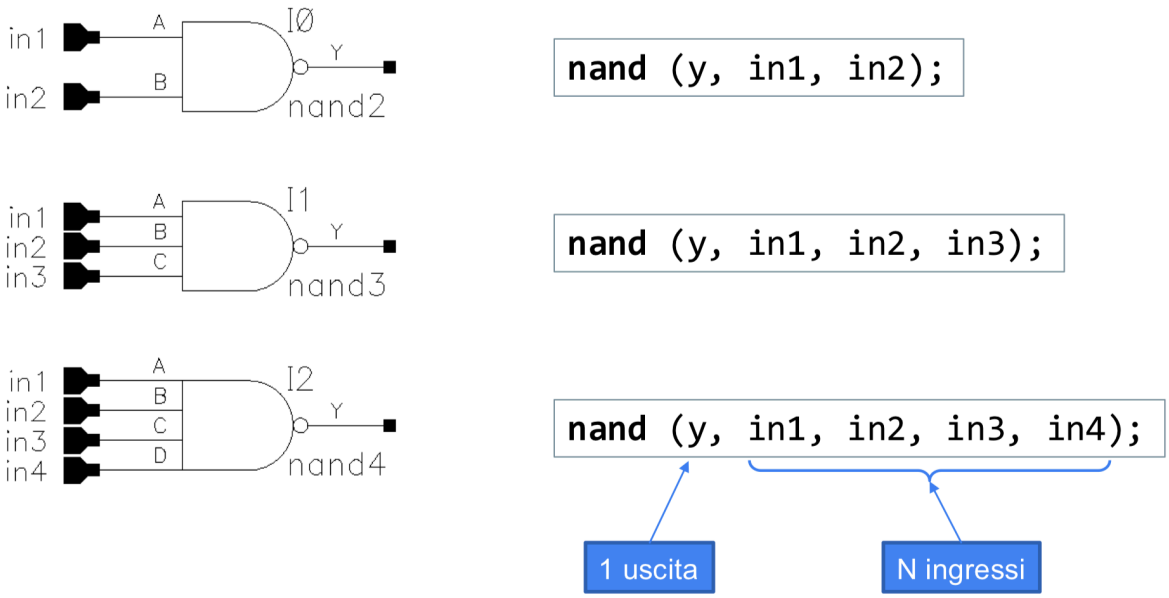
Le porte logiche di base sono:

- `and` ;
- `or` ;
- `not` ;
- `buf` ;
- `nand` ;
- `nor` ;
- `xor` ;
- `xnor` ;
- porte logiche tri-state.
  - `bufif1` , `bufif0` ;
  - `notif1` , `notif0` .



Rappresentazione di porte logiche tri-state

I pin delle porte logiche elementari sopra citate sono espandibili, come mostrato nella figura.



Espansione dei pin delle porte logiche

Valori di uscita delle porte logiche elementari

AND	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

OR	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

XOR	0	1	X	Z
0	0	1	X	X
1	1	0	X	X
X	X	X	X	X
Z	X	X	X	X

NAND	0	1	X	Z
0	1	1	1	1
1	1	0	X	X
X	1	X	X	X
Z	1	X	X	X

NOR	0	1	X	Z
0	1	0	X	X
1	0	0	0	0
X	X	0	X	X
Z	X	0	X	X

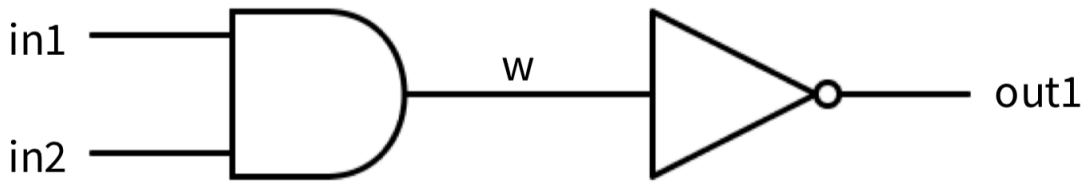
XNOR	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

BUF	
0	0
1	1
X	X
Z	X

NOT	
0	1
1	0
X	X
Z	X

Tabelle che rappresentano i valori di uscita delle porte logiche elementari

▼ Esempio - Modellazione a livello di porta logica



Circuito di riferimento da simulare: una porta NAND

```
module my_nand (out1, in1, in2);
   // inizio interfaccia
   output out1;
   input in1, in2; // input inseriti nella stessa definizione,
                   // in quanto sono entrambi da 1 bit
   // fine interfaccia

   // visto che devo collegare la porta AND e la porta NOT, mi serve un wire "w"
   wire w;
   and(w, in1, in2); // l'uscita della porta AND è il wire "w"
   not(out1, w);     // l'uscita della porta not è out1, l'ingresso è w
endmodule
```

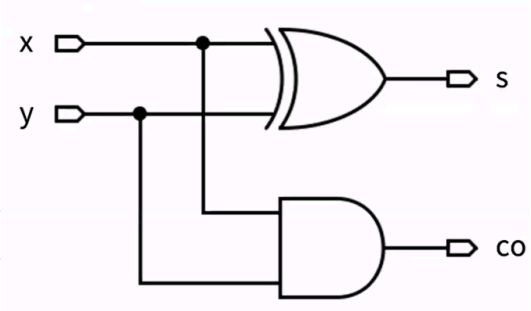
▼ Esempio - Semisommatore con porte logiche

Si realizzi l'Half Adder in figura, tale che

`s = x+y` e `co` sia il Carry Out.

In tal senso, occorre ricordare che vogliamo fare in modo che `s` sia:

0+0 = 0	1+0 = 1	0+1 = 1	1+1



Visualizzazione del modulo Half Adder

Questo si può ottenere facendo lo XOR tra x e y, mentre il Carry Out è ottenibile dall'AND di x e y.

```
module hadd(
   s, co, // uscite
   x, y   // ingressi
);
   input x, y;
   output s, co;
   // FINE INTERFACCIA

  xor(s, x, y);     // somma
  and(carry, x, y); // riporto
endmodule
```

Ritardi di porte logiche

I ritardi in Verilog sono preceduti da un cancelletto (`#`). Tipicamente, un ritardo associato a una porta logica è dichiarato come segue:

$$\#(\text{ritardo di propagazione low} \rightarrow \text{high, ritardo di propagazione h}\rightarrow\text{l}) = \#(t_{\mathrm{p}}^{\mathrm{L}\rightarrow\mathrm{H}}, t_{\mathrm{p}}^{\mathrm{H}\rightarrow\mathrm{L}})$$

Si può essere ancora più specifici, fornendo il range dei ritardi minimi, tipici e massimi. Sui valori di $t_{\mathrm{p}}^{\mathrm{L}\rightarrow\mathrm{H}}$ o di $t_{\mathrm{p}}^{\mathrm{H}\rightarrow\mathrm{L}}$ possiamo definire i ritardi `minimi:tipici:massimi`.

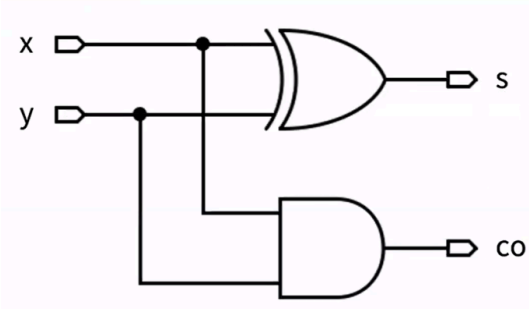
È inoltre indispensabile specificare l'unità di misura di tali ritardi tramite la direttiva:

```
`timescale <unità_di_tempo>/<precisione_temporale>
// solitamente si specifica ad inizio file
```

⚠️ Si noti che tali ritardi sono utili solo nella simulazione comportamentale del circuito. Non hanno, infatti, alcun significato nella sintesi e nella realizzazione effettiva del circuito.

▼ Esempio - Semisommatore con porte logiche, con ritardi

Si realizzi un Half Adder del tipo in figura, tale che `s = x+y` e `co` sia il Carry Out.



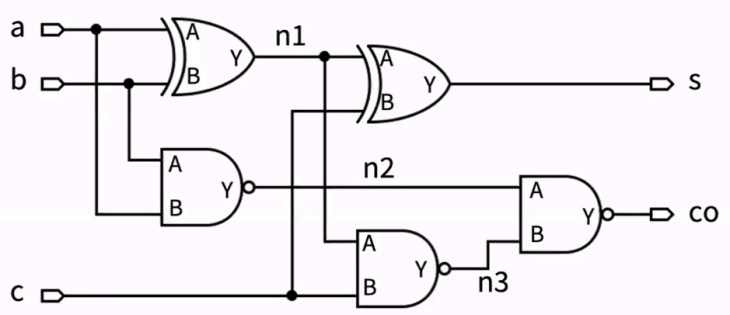
Si sa che:

- $t^{\mathrm{L}\to\mathrm{H}}_{\mathrm{p_{XOR}},\mathrm{min}} = 2\ \mathrm{ns}$;
- $t^{\mathrm{L}\to\mathrm{H}}_{\mathrm{p_{XOR}},\mathrm{max}} = 4\ \mathrm{ns}$;
- $t^{\mathrm{H}\to\mathrm{L}}_{\mathrm{p_{XOR}}} = 5\ \mathrm{ns}$;
- $t^{\mathrm{L}\to\mathrm{H}}_{\mathrm{p_{AND}}} = t^{\mathrm{H}\to\mathrm{L}}_{\mathrm{p_{AND}}} = 3.6\ \mathrm{ns}$.

```
// dico che l'unita di tempo è 1 ns e che la precisione temporale
// è pari a 100 ps = 0.1 ns
`timescale 1ns/100ps
...
module hadd(
   s, co, // uscite
   x, y   // ingressi
);
   input x,y;
   output s, co;
   // FINE INTERFACCIA

  xor # // definizione del ritardo
        (2:3:4,     // definizione del tempo minimo:tipico:medio del tp l→h XOR
          5)        // definizione del tempo tipico del tp h→l XOR
         (s, x, y);  // somma
   // per intero, andrebbe scritto:
   //     xor #(2:3:4, 5) (s,x,y);

  and #(3.567) // definizione del ritardo della porta AND
  // NOTA BENE: visto che la precisione è di 100 ps = 0.1 ns,
  // 3.567 ns viene arrotondato a 3.6 ns
           (co, x, y);
   // and #(3.6) (co,x,y);
endmodule
```

▼ Esempio - Utilizzo di sottomoduli

L’obiettivo è realizzare un Full Adder a 3 ingressi tramite porte logiche elementari.

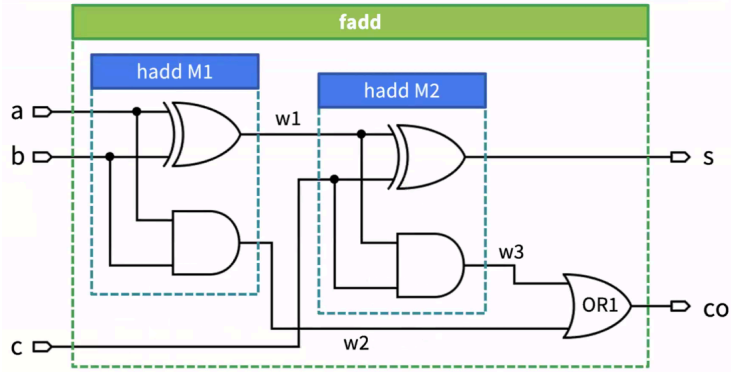


Schema circuitale

```
module fadd(s, co, a,b,c); // come di consuetudine: prima gli output, poi gli input
   output co, s;
   input a,b,c;
   // FINE INTERFACCIA

   wire n1, n2, n3;
   // FINE NET

   // l'ordine dei comandi sotto è stato volutamente "sparso" per rimarcare che,
   // con questa metodologia di programmazione, l'ordine delle istruzioni
   // NON è importante
   nand(n3,n1,c);
   xor(n1,a,b);
   nand(n2,a,b);
   xor(s,n1,c);
   nand(co,n2,n3);
endmodule
```

E’ anche vero, però, che possiamo realizzare un Full Adder come interconnessione di Half Adder!

Quindi, avendo prima creato il modulo hadd, possiamo istanziarlo due volte per creare lo schema in figura.

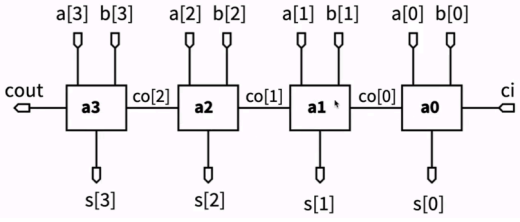


Schema Full Adder definito come interconnessione di Half Adder

```
module fadd(co,s,a,b,c);
   output co,s;
   input a,b,c;

   wire w1, w2, w3;
   // si istanzia il modulo hadd dandogli un nome!
   // hadd (nome modulo) M1 (nome istanza) (componenti)
   hadd M1 (w1, w2, a,b):  // la somma va su w1, il carry su w2
   hadd M2 (s, w3, w1, c); // la somma va su s, il carry su w3
   // N.B. serve avere un modulo hadd in un altro file!
   or OR1 (co, w3, w2);
endmodule
```

▼ Esempio - Sommatore a quattro bit con sottomoduli

- Implementazione gerarchica del modulo add4
  - Quattro moduli fadd
  - Ogni fadd consiste di
    - Due moduli hadd
    - Una porta logica elementare OR




```
1  module add4 (s, cout, ci, a, b);
2
3      input  [3:0] a, b;
4      input        ci;
5      output [3:0] s;
6      output       cout;
7
8      wire   [2:0] co;
9
10     fadd a0 (co[0], s[0], a[0], b[0], ci);
11     fadd a1 (co[1], s[1], a[1], b[1], co[0]);
12     fadd a2 (co[2], s[2], a[2], b[2], co[1]);
13     fadd a3 (cout, s[3], a[3], b[3], co[2]);
14
15 endmodule
```

Definizione del sommatore a quattro bit con sottomoduli

Modellazione RTL

La modellazione RTL consente di compattare la sintassi e “allontanarci” dalle porte logiche grazie a un meccanismo chiamato “assegnazione continua”.

Tale meccanismo prevede di assegnare operazioni complesse di alto livello in una sola riga di codice, per poi lasciare al sintetizzatore logico l’utilizzo delle porte logiche necessarie.

A tale scopo, si possono implementare:

- Sommatori;
- Comparatori;
- Multiplexer;
- Moltiplicatori paralleli.

Operatore Assign

Ogni qualvolta in cui un operando presente nell’assegnazione di tipo `assign` cambia valore, al contempo viene “ricalcolata” la variabile che è stata dichiarata come `assign`.

```
assign #ritardo <nome_rete> = <espressione>;
// il ritardo (opzionale) prende l'unità di misura dal `timescale a inizio file
```

La destinazione di un’assegnazione dovrebbe sempre essere un wire o una porta di uscita.

▼ Esempi - Operatore assign

```
assign out = a&b|c; // out = a AND b OR c
// significa che out cambierà "istantaneamente" ogni volta che a, b o c variano
// tutto questo è più comodo di utilizzare effettivamente porte AND e OR

assign eq = (a+b == c) // ci si chiede se la somma tra a e b sia uguale a c
// se a+b == c, eq = 1; diversamente 0.

wire #10 inv = ~in; // la variabile inv cambierà dopo 10 timescale
                                // da quando "in" varia
// inv = NOT(in) con ritardo.
// !!! questa sintassi è equivalente a:
//     wire inv; assign #10 inv = ~in
// posso comunque definire e assegnare un wire!

wire [7:0] c = a+b; // anche questa è un'assegnazione (assign) continua (implicita)
```

⚠️ Occhio a evitare loop del tipo `assign a = b+a;` , poiché tale tipo di assegnazione non è sintetizzabile.

Operatori Verilog utili in assegnazioni continue

Operatori aritmetici	
`a + b`	Somma
`a - b`	Differenza
`-a`	Cambio segno
`a * b`	Moltiplicazione
`a / b`	Divisione
`a % b`	Resto

Operatori bit a bit	
`~a`	NOT bit a bit
`a & b`	AND bit a bit
`a | b`	OR bit a bit
`a ^ b`	XOR bit a bit
`a ~^ b`	XNOR bit a bit
`a ^~ b`	XNOR bit a bit

Operatori di spostamento	
`a << n`	Shift logico a sinistra
`a >> n`	Shift logico a destra
`a <<< n`	Shift aritmetico a sinistra
`a >>> n`	Shift aritmetico a destra
`{a, b}`	Concatenazione

Operatori relazionali	
`a == b`	Uguale
`a != b`	Diverso
`a < b`	Minore
`a > b`	Maggiore
`a <= b`	Minore o uguale
`a >= b`	Maggiore o uguale

Operatori di riduzione	
`&a`	AND tutti i bit
`|a`	OR tutti i bit
`^a`	XOR tutti i bit
`~&a`	NAND tutti i bit
`~|a`	NOR tutti i bit
`~^a`	XNOR tutti i bit

Operatori logici	
`!a`	Negazione logica
`a && b`	AND logico
`a || b`	OR logico
`sel?a:b`	Condizionale ternario

Operatori Verilog supportati nelle assegnazioni continue

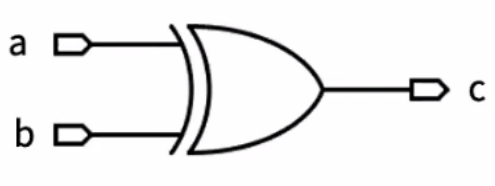
⚠️ La tilde (`~`) è ottenuta su Windows con `ALT+126` .

▼ Esempi - Operatori Verilog

```
assign c = a+b; // c sarà (in automatico???) di N+1 bit se a e b sono su N bit

assign c = a>b; // c starà su 1 bit; mi restituisce un valore logico

assign c = a||b; // a e b sono entrambi su 1 bit, così come c e si fa l'OR logico
// se a o b fossero stati su N bit, sarebbero stati gestiti come 0 solo se
// fossero 0, altrimenti a e b sarebbero stati ridotti a 1 logico

assign c = a&b; // a,b,c su N bit, in quanto si fa l'AND bit a bit

assign c = &a; // AND di tutti i bit di a.
```

▼ Esempio - XOR con ritardo (RTL)

Si realizzi la porta logica XOR, come in figura.

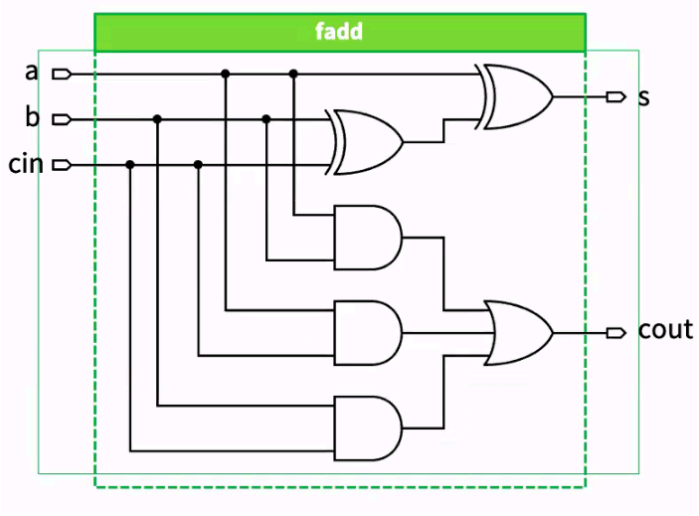


Porta XOR

```
module my_xor(c, a, b); // al solito, prima uscita, poi ingressi
   output c;
   input a,b;

   assign #2 c = a ^ b; // a XOR (bit a bit) b con ritardo di 2 timescale
endmodule
// SI RICORDI CHE TALE RITARDO SI USA SOLO IN SIMULAZIONE (NON IN SINTESI)!
```

▼ Esempio - Sommatore completo

Si implementi un Full Adder generico, come in figura.



Schema di un Full Adder

```
module fadd (cout, s, a, b, cin)
   output cout, s;
   input a,b,cin;

   assign s = (b ^ cin) ^ a;

   assign cout = (a&b) | (a&cin) | (b&cin);
endmodule
```

▼ Esempio - Sommatore a 4 bit

Si implementi un sommatore tale che gli operandi siano su 4 bit.

```
module fh4(s, cout, a,b,cin);
   output [3:0] s;  // somma su 4 bit
   output cout;     // carry out su un bit
   input [3:0] a,b; // operandi di ingresso su 4 bit
   input cin;       // carry in di ingresso su un bit

   // Si potrebbe fare, come prima utilizzo di XOR e (AND + OR) oppure,
   // come in questo caso, sfruttare l'operatore di concatenazione
   assign {cout,s} = a+b+cin; // a+b+cin genera un output a 5 bit di cui l'MSB
                                          // sarà il carryout
```

Realizzazione di moduli parametrici

Il modulo “generico”, definito come parametrico, ha come obiettivo quello di farci fornire, dall’utente, un parametro che possa servirci per scopi quali:

- la dichiarazione di un vettore con le corrette dimensioni;
- l’istanziazione di un modulo che prevede “N” (con N passato come parametro) iterazioni.

La sintassi per l’utilizzo di un modulo parametrico è la seguente:

```
<nome_modulo> #(elenco_parametri) <nome_istanza> (elenco_porte);
```

La sintassi per la dichiarazione di un parametro è, invece, la seguente:

```
parameter <nome_parametro>; // se non si vuole dare un valore di default al parametro
parameter <nome_parametro> = <valore_default>; // se si vuole dare un valore di default
```

▼ Esempio - Sommatore a N bit parametrico

Vogliamo realizzare un sommatore parametrico tale che il numero N di bit degli operandi sia passato dall’utente.

```
module #(parameter N = 2) adder (cout, s, a, b, cin);
        // specifichiamo un parametro "N", di default uguale a 2
   input [N-1:0] a,b; // specifico che gli operandi sono su N bit
   input cin;
   output [N-1:0] s; // specifico che la somma (senza cout) sta su N bit
   output cout;
```

```
    assign {cout, s} = a+b+cin;
endmodule
```

Vediamo ora un altro modo (meno leggibile) di implementare questo codice.

```
module adder (cout, s, a, b, cin);
    parameter N = 2; // specifichiamo un parametro "N", di default uguale a 2
    input [N-1:0] a,b; // specifico che gli operandi sono su N bit
    input cin;
    output [N-1:0] s; // specifico che la somma (senza cout) sta su N bit
    output cout;

    assign {cout, s} = a+b+cin;
endmodule
```

Vediamo ora un po' di utilizzi del modulo parametrico `adder`.

```
// in un altro file!
adder #(8) adder_8 (co, s, a,b,ci)       // sommatore a 8 bit
adder adder_2 (co, s, a, b, ci)          // sommatore a 2 bit (uso il default)
adder #(12) adder_12 (co, s, a, b, ci)   // sommatore a 12 bit
adder #(.N(8)) adder_8 (co, s, a, b, ci) // specifico che mi riferisco a N
```

⚠️ Come faccio a specificare il ritardo di un modulo da me creato?

Nel caso di un sommatore a 8 bit con 4 ns di ritardo NON si può fare una chiamata di questo tipo:

```
`timescale 1ns/100ps
...
adder #(8) #4 adder_8 (co, s, a,b,ci) // sommatore a 8 bit
```

I ritardi in Verilog non possono essere specificati su moduli dichiarati da noi in maniera diretta (come facevamo con le porte elementari), possiamo rendere il ritardo parametrico, come segue:

```
module adder (cout, s, a, b, cin);
   parameter N = 2; // specifichiamo un parametro "N", di default uguale a 2
   parameter delay = 0;
   input [N-1:0] a,b; // specifico che gli operandi sono su N bit
   input cin;
   output [N-1:0] s; // specifico che la somma (senza cout) sta su N bit
   output cout;

   assign #(delay) {cout, s} = a+b+cin;
   // si noti che, nel caso di ritardo costante, avremmo potuto inserire
   // assign #4 {cout, s} ....; Poichè stiamo dando un parametro, però
   // occorre metterlo tra parentesi per aiutare il parser.
endmodule
```

Successivamente, si può effettuare una chiamata come segue:

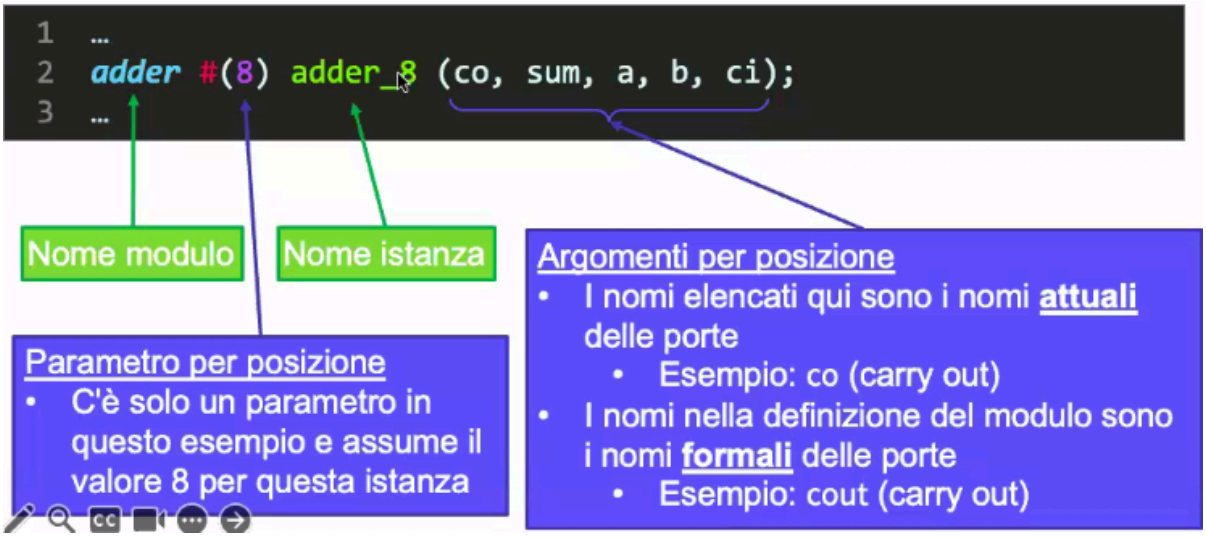
```
adder #(8, 4) adder_8_4 (co, s, a, b, cin); // che funziona correttamente
adder #(.N(8), .delay(4)) adder_8_4 (co, s, a, b, cin); // più leggibile
adder #(.delay(4), .N(8)) adder_8_4 (co, s, a, b, cin); // più leggibile
```

Si presti attenzione che tali informazioni non vengono dal docente, ma sono ricavate dalla rete e, pertanto, l'assegnazione di un ritardo ad assign #(delay) potrebbe non funzionare correttamente con tutti i compilatori!

Passaggio di argomenti e parametri

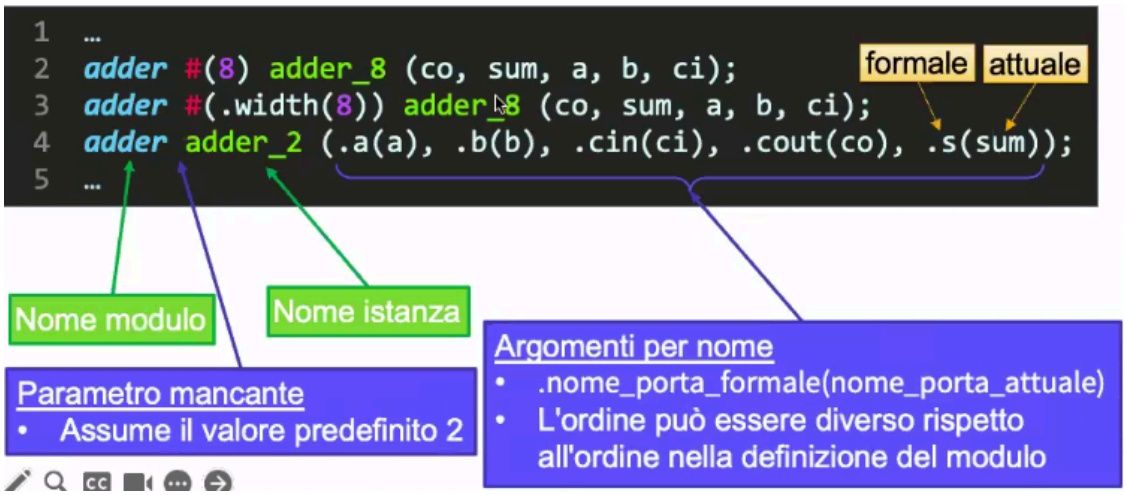
Quando chiamiamo un modulo possiamo scegliere di passare i parametri "attuali" al modulo, il quale contiene i parametri "formali" per posizione (come in C) o per nome (in maniera tale da rendere l'ordine ininfluente e rendere il codice più leggibile).

Data la seguente funzione, un esempio di passaggio di parametri per posizione è il seguente:



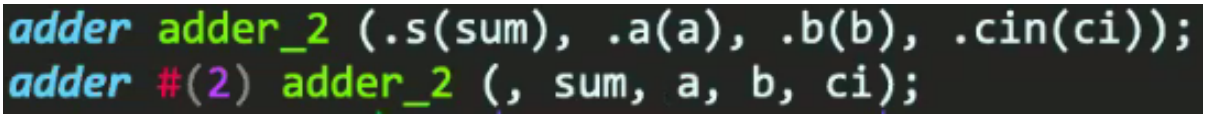
Modulo parametrico implementato in precedenza, in cui tutti gli argomenti e i parametri sono passati per posizione

Un esempio di passaggio di parametri per nome è il seguente:



Modulo parametrico implementato in precedenza, dove si fa utilizzo in riga 3 di argomenti e parametri per nome. Si noti che width, da noi, è stato chiamato N

Si possono, inoltre, lasciare delle porte non collegate come segue:

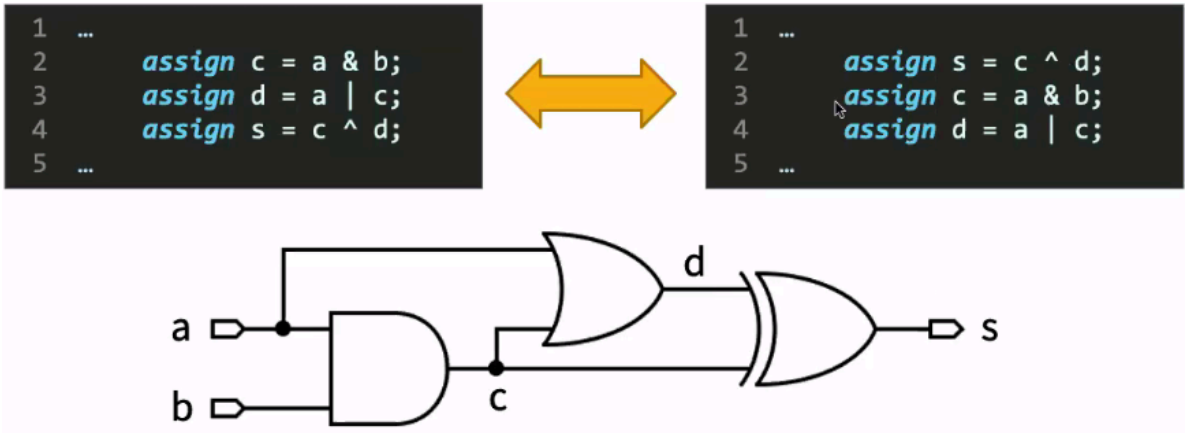


Se un'uscita non ci serve, possiamo decidere di non collegarla a niente. Ciò non si può fare con gli ingressi!

Ordine delle istruzioni

L'ordine delle istruzioni, come detto nell'introduzione, non è importante in RTL, poiché:

- tutte le istruzioni sono eseguite in parallelo;
  1. le espressioni a destra dell'uguale vengono prima calcolate tutte insieme;
  2. vengono assegnate tutte insieme ai wire a sinistra dell'uguale;
- sono sempre equivalenti a uno schema a blocchi.



Equivalenza tra codici anche con ordine delle istruzioni differente!

Blocco generate con for e if

Come detto in precedenza, si può parametrizzare un modulo in maniera tale che si iteri N volte.

In questo caso, il blocco `generate` ci consente di effettuare cicli e blocchi condizionali solo per strutture "ripetitive" (si tratta di una esecuzione durante la fase di compilazione, non in runtime), per le quali dovremmo usare tante righe di codice per istanziare uno stesso comando, come nel seguente esempio:

```
parameter N; // generica variabile parametrica che descrive il numero delle iterazioni
genvar i; // variabile utile nel ciclo for

generate
   for (i = 0; i < N; i++) begin
      if(i == 0)
         istruzione1;
      else if (i == N-1)
         istruzione2;
      else begin;
         istruzione3;
         istruzione4;
      end
   end
endgenerate
```

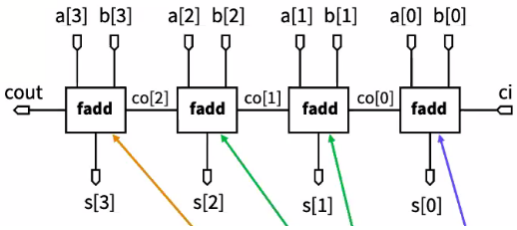
```
// Si noti come:
// - con una istruzione si comporta come in C, ovvero non
//   sono necessarie le graffe;
// - con più istruzioni sotto un blocco condizionale o un for
//   sono necessarie le “graffe”, definite da begin ed end.
```

⚠️ Nei blocchi `generate` non è possibile dare nomi ai moduli istanziati in maniera “semplice”, ma avrebbe comunque poco senso farlo (vedi esempio).

Si noti che non è un loop iterativo che assicura di realizzare istruzioni in sequenza, ma è come se servisse solo a replicare il codice.

▼ Esempio - Sommatore completo su N bit con generate

Si parta dal sommatore in figura a 4 bit, per poi estenderlo a N bit.



In blu la prima istanza, in verde le intermedie, in arancione l’ultima istanza

```
module adder (s, cout, ci, a, b);
   parameter N = 4; // di default realizziamo un sommatore a 4 bit, come in figura
   input [N-1:0] a,b;
   input ci;
   output [N-1:0] s;
   output cout;

   wire [N-2:0] co; //si utilizza un vettore che contenga i vari wire
   // sono in totale N-1 bit in quanto l'N-esimo è cout
   genvar i; // variabile intera utile per l'iterazione

   generate
      for (i = 0; i < N; i++) begin
         if(i == 0)
            fadd(co[i], s[i], a[i], b[i], ci); // ISTANZA BLU
         else if (i == N-1)
```

```
            fadd(co[i], s[i], a[i], b[i], co[i-1]); // ISTANZA ARANCIONE
        else
            fadd(cout, s[i], a[i], b[i], co[i-1]); // ISTANZA VERDE
    end
  endgenerate
endmodule

// Si noti che nelle chiamate al modulo “fadd” non si
// specifica il nome dell’istanza.
//
// Nel caso del generate, ciò non si può fare perchè
// si andrebbe, negli intermedi, a istanziare moduli
// con lo stesso nome.
```

Modellazione comportamentale

La modellazione comportamentale è ottenuta tramite blocchi `always`, i quali definiscono una sorta di processi, ovvero funzioni le cui istruzioni sono eseguite in sequenza, ma nello stesso istante.

Per fare un paragone, un blocco `always` è una specie di chiamata ricorsiva che viene effettuata ogni qual volta i parametri “sensibili” (che devono essere `reg` o `wire`) del blocco `always` vengono modificati al di fuori del blocco `always` stesso.

Con tale paradigma si possono modellare sia circuiti combinatori, sia sequenziali.

Ecco la sintassi di un blocco `always`:

```
always @(<elenco_sensibilità>) begin
    ...
end

// Quando uno o più segnali presenti nell’elenco di sensibilità
// cambia valore, il blocco always viene eseguito.
```

⚠️ Tutte le uscite di un blocco `@always` devono essere di tipo `reg`:

- le assegnazioni usano solo “=” o “<=“;
- mantengono memoria dell’ultima assegnazione, al contrario dei `wire` (che, in uscita, sono frutto di assegnazioni “continue”).

Il tipo `reg` assomiglia a un registro, ma non ne rappresenta necessariamente uno.

Dentro i blocchi `always` possono essere utilizzati i costrutti `if` e `case`, grazie al fatto che le operazioni vengono eseguite in ordine.

Always con circuiti combinatori

La caratteristica dei circuiti combinatori è tale per cui l'uscita dipende, in ogni istante, dal valore degli ingressi e, pertanto, il blocco `always` sarà del tipo:

```
always @(ingresso1, ingresso2, ..., ingressoN) begin
   ...
end

// L'elenco di sensibilità è dato dagli ingressi presenti nel circuito combinatorio.
```

Per essere sicuri di valutare tutti gli ingressi in una logica combinatoria si può anche utilizzare tale sintassi:

```
always @(*) begin // Viene eseguito sulla variazione di:
   out1 = a+b+c+d; // sulla variazione di uno tra a,b,c,d
   out2 = f*e+3; // sulla variazione di uno tra f ed e.
end

// Il blocco always viene eseguito ogni qual volta che un
// secondo membro di un'assegnazione cambia valore.
```

▼ Esempio - Porta AND

Si realizzi la porta AND in figura, tramite il blocco `always`.

Porta AND

```
reg y; // uscita sempre di tipo reg
wire a,b; // ingressi possono essere wire o reg.
always @(a or b) begin
   y = a&b;
end
```

Tale codice è identico a scrivere il seguente codice:

```
reg y; // uscita sempre di tipo reg, mantiene memoria dell'ultima assegnazione
wire a,b; // ingressi possono essere wire o reg.
always @(*)
   y = a&b; // con una sola riga di codice posso evitare di usare begin e end
```

Esso è ancora uguale a:

```
wire y, a,b; // il wire presuppone un'assegnazione continua che esiste sempre!
assign y = a & b; // quì possiamo usare un wire come uscita!

// Reg si potrebbe usare come secondo membro di un'operazione
// di assign ma non come primo membro
```

Always con circuiti sequenziali

Nel caso di circuiti sequenziali è necessario far sì che il blocco `always` sia attivato solo durante il fronte di salita/discesa del clock tramite le parole chiave:

- `posedge`: fronte di salita;
- `negedge`: fronte di discesa.

```
always @(posedge clock) // esempio 1
always @(negedge clock or reset) // esempio 2, FF attivo su fronte di discesa
// con reset asincrono

// Il clock è un input, non una parola chiave (wire o input che sia)
```

▼ Esempio - Porta AND + FF

Quando arriva il fronte di salita del clock, si vuole che un flip flop memorizzi l'AND tra due ingressi a e b.

```
reg a,b,ff;
always @(posedge clock) // ogni fronte di salita del clock voglio che il ff memorizzi
   ff = a & b;
```

Assegnazioni bloccanti

⚠️ È importante ricordare che, all'interno del corso, utilizzeremo solo l'assegnazione bloccante.

Le assegnazioni bloccanti hanno un effetto immediato (come in C):

```
always @(a or b) begin // 1
   a = b;              // 2
   b = a;              // 3
end                   // 4

// Qualora a o b (o entrambi) abbiano una variazione di valore,
// il blocco always NON verrà eseguito.
```

1. supponiamo che, durante la prima esecuzione del blocco `always`, si abbia `a = 0` e `b = 1`, come in tabella;

	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice
a	0	1	1	1
b	1	1	1	1

2. alla seconda riga di codice otteniamo `a = b` e, quindi, `a` passa da 0 a 1, mentre `b` mantiene il suo valore precedente (pari a 1);
3. alla terza riga di codice, `a` mantiene il suo valore precedente (1) e si esegue `b = a`, ma `a = 1`, quindi `b = a = 1`;
4. si raggiunge la fine del blocco `always`: `a` e `b` mantengono i loro valori precedenti (1 e 1).

> ⚠️ Ci si potrebbe chiedere come mai non si rientri nel blocco `always`, dato che `a` ha cambiato valore.
>
> Questo non avviene perché il blocco `always` viene rieseguito se e solo se `a` o `b`, o addirittura entrambi subiscono variazioni, ma esse devono avvenire al di fuori del blocco `always` stesso.

Visto che le variazioni ai parametri di sensibilità (`a` e `b`) sono solo interne al blocco `always`, quest'ultimo non viene nuovamente eseguito!

Assegnazioni non bloccanti

Le assegnazioni non bloccanti avvengono solo al termine dell'esecuzione del blocco `always` e, in tal senso, comporterebbero una nuova esecuzione del blocco `always` nel caso di variazione.

```
always @(a or b) begin // 1
   a <= b;             // 2 assegna a = b alla fine del blocco always
   b <= a;             // 3 assegna b = a alla fine del blocco always
end                    // 4
```

```
// qualora a o b (o entrambi) abbiano una variazione di
// valore, il blocco always verrà eseguito.
```

1. supponiamo che, durante la prima esecuzione del blocco `always`, si abbia `a = 0` e `b = 1`, come in tabella;

1° esecuzione	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice	Valore futuro
a	0	0	0	1	1
b	1	1	1	0	0

2. viene eseguita l'istruzione `a <= b` e, quindi, il valore futuro di `a` sarà pari al valore corrente di `b` (1) solo alla fine del blocco `always`;
3. viene eseguita l'istruzione `b <= a` e, quindi, il valore futuro di `b` sarà pari al valore corrente di `a` (0) solo alla fine del blocco `always`;
4. raggiunta la fine del blocco `always` vengono assegnati i valori futuri di `a` e `b`.

Poiché `a` e `b` sono variate al di fuori del blocco `always` e sono nella lista di sensibilità, il blocco si riattiva e viene nuovamente eseguito.

2° esecuzione	1° riga di codice	2° riga di codice	3° riga di codice	4° riga di codice	Valore futuro
a	1	1	1	0	0
b	0	0	0	1	1

Poiché, nuovamente, `a` e `b` sono variate e si trovano nella lista di sensibilità, il blocco `always` viene nuovamente eseguito (indefinitivamente, come in figura).

	1ª esecuzione				2ª esecuzione				3ª esecuzione				4ª esecuzione				5ª esecuzione				6ª esecuzione				7ª esecuzione				8ª esecuzione				9ª esecuzione				
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	Futuro
a	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	
b	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	0	

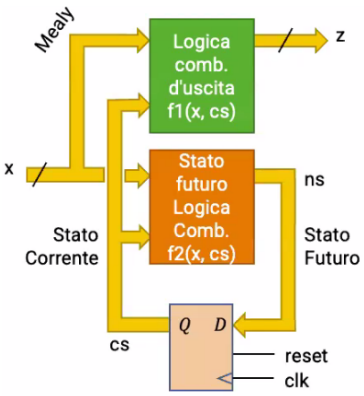
Tipicamente questo comportamento non è desiderato, anche se somiglia a un clock

Esempio - Realizzazione di un circuito sequenziale Mealy (FSM)

Come noto, una macchina di Mealy è tale per cui il valore dell'uscita dipende sia dal valore corrente degli ingressi, sia dallo stato corrente.

Si vuole, quindi, realizzare una macchina di Mealy come in figura.

Si noti che il FF-D è attivo sul fronte di salita.



Macchina di Mealy

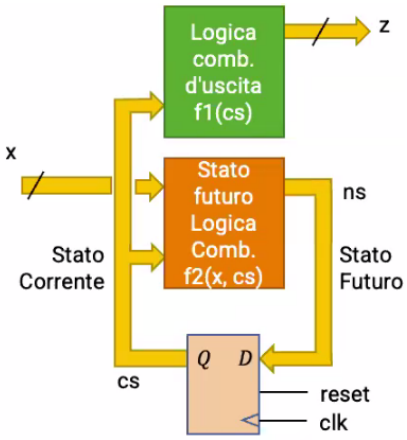
```
module MealyFSM (z, clk, reset, x);
   output reg z;
   input clk, reset, x;
   // si poteva anche fare output z; reg z;
   reg ns; // next state (sono come wire ma necessariamente devono essere dichiarati reg)
   reg cs; // current state (come wire ma serve dichiararlo come reg poichè output).

   /// LE REG SONO NECESSARIE PER USARLE COME "uscita" dei blocchi always

   // GESTIAMO LA VARIAZIONE DEGLI STATI (dipendente  da clock e reset)
   always @(posedge clock // attivo sul fronte di salita del clock
                or
                posedge reset) // reset asincrono: se reset passa da 0→1 allora si entra
   begin: stateReg // etichetta utile per documentazione, nome del blocco always
      if (reset)
         cs = 0;
      else
         cs = ns; // current state = next state... aggiorno lo stato corrente
   end
   always @(x or cs) begin: outputFunction
      // la lista di sensibilità è data dall'ingresso e dallo stato corrente.
      z = f1(x,cs); // l'uscita dipende da ingresso e stato corrente.
   end

   always @(x or cs) begin: nextStateTransition
      ns = f2(x, cs); // il next state dipende da ingresso e stato corrente
endmodule
```

Esempio - Realizzazione di un circuito sequenziale Moore (FSM)

Come noto, una macchina di Moore è tale per cui il valore dell'uscita dipende solo dallo stato corrente.

Si vuole, quindi, realizzare una macchina di Moore come in figura.

Si noti che il FF-D è attivo sul fronte di salita e che il reset è sincrono (cioè deve arrivare insieme al clock).



Macchina di Moore

```
module MealyFSM (z, clk, reset, x);
   output z;
   input clk, reset, x;
   reg ns; // next state (sono come wire ma necessariamente devono essere dichiarati reg)
   reg cs; // current state (come wire ma serve dichiararlo come reg poichè output).
   reg z;
   /// LE REG SONO NECESSARIE PER USARLE COME "uscita" dei blocchi always

   // GESTIAMO LA VARIAZIONE DEGLI STATI (dipendente  da clock)
   always @(posedge clock) // vogliamo un reset sincrono!
   begin: stateReg
      if (reset) // Il reset deve arrivare durante la transizione del clock!
         cs = 0;
      else begin
         cs = ns;
      end
   end
   always @(cs) begin: outputFunction
      // la lista di sensibilità è data dal solo stato corrente.
      z = f1(cs); // l'uscita dipende da ingresso e stato corrente.
   end

   always @(x or cs) begin: nextStateTransition
      ns = f2(x, cs); // il next state dipende da ingresso e stato corrente
endmodule
```