Modellazione RTL

▼ Creatore originale: @Gianbattista Busonera

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 (implicit
```

⚠️ 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.

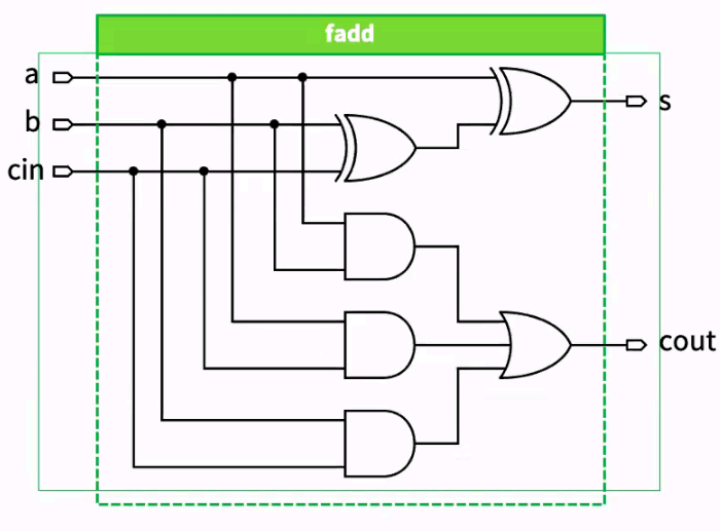
Porta XOR

```
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 SINTE
```

▼ 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 parame
parameter <nome_parametro> = <valore_default>; // se si vuole dare un valore di d
```

▼ 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:

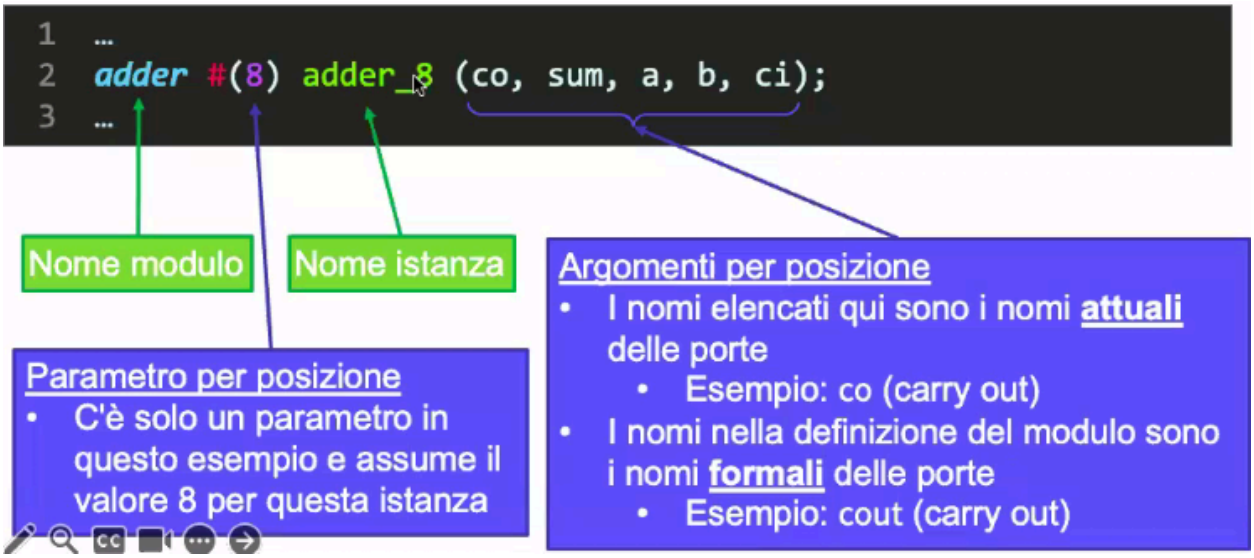
```
adder #(8, 4) adder_8_4 (co, s, a, b, cin); // che funziona correttamente
adder #(.N(8), .delay(4)) adder_8_4 (co, s, a, b, cin); // più leggibile
adder #(.delay(4), .N(8)) adder_8_4 (co, s, a, b, cin); // più leggibile
```

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

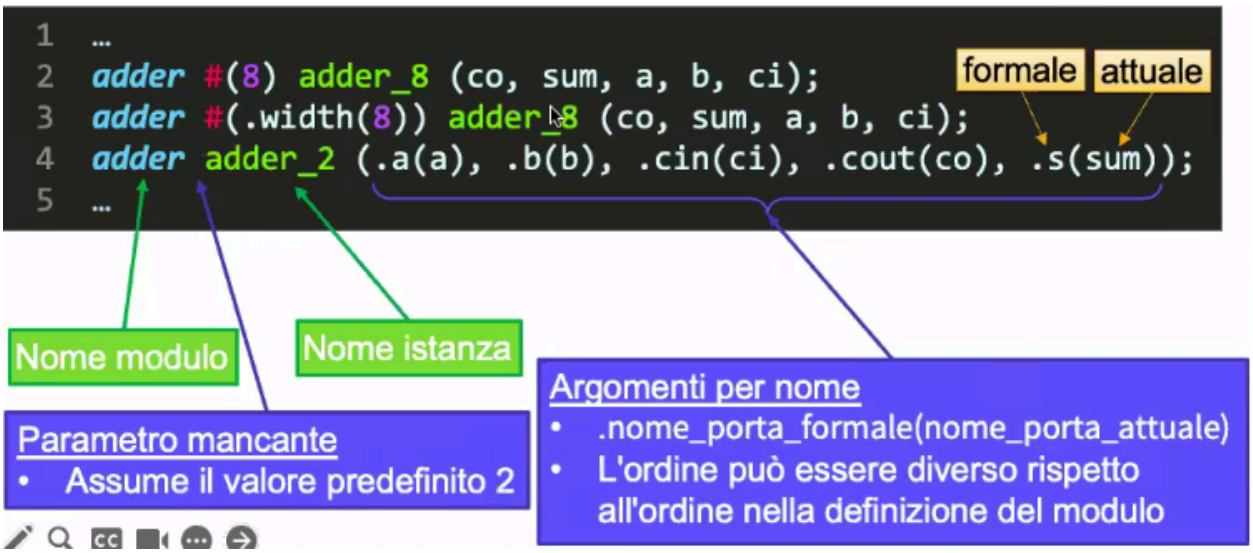
Quando chiamiamo un modulo possiamo scegliere di passare i parametri “attuali” al modulo, il quale contiene i parametri “formali” per posizione (come in C) o per nome (in maniera tale da rendere l’ordine ininfluente e rendere il codice più leggibile).

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:

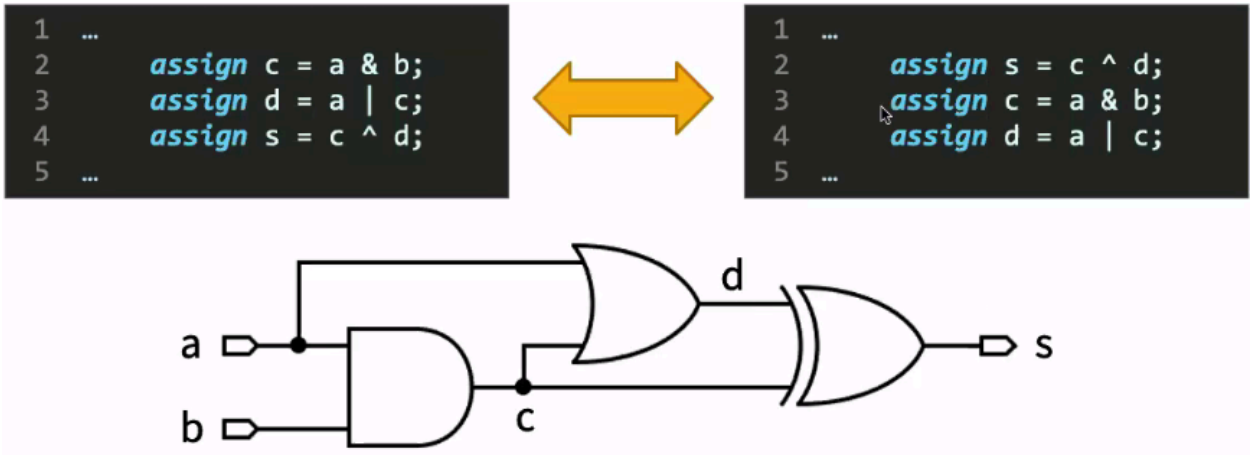
```
adder adder_2 (.s(sum), .a(a), .b(b), .cin(ci));
adder #(2) adder_2 (, sum, a, b, ci);
```

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 iterazior
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
```

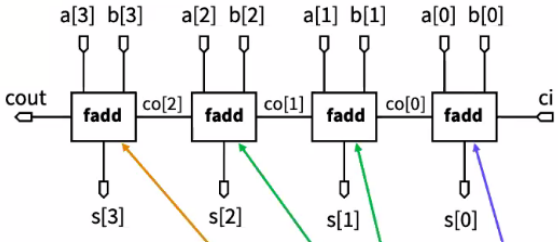
```
// Si noti come:
// - con una istruzione si comporta come in C, ovvero non
//   sono necessarie le graffe;
// - con più istruzioni sotto un blocco condizionale o un for
//   sono necessarie le “graffe”, definite da begin ed end.
```

⚠ 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.
```