Reti combinatorie in Verilog

Marco Danelutto

September 29, 2020

Abstract

Queste note sono un super riassunto di quello che serve per programmare e testare (funzionalmente) componenti logici che siano reti combinatorie utilizzando Verlog (non System Verilog!).

1 Componenti descritti con tabelle di verità

Verilog permette di definire un componente utilizzando tabelle di verità, utilizzando un modulo di tipo `primitive`.

- L'intestazione del modulo richiede la parola chiave `primitive`, il nome del modulo e la lista che contiene il segnale in uscita (a sinistra, normalmente. Il segnale è per forza da un bit solo) e i segnali in ingresso.
- Il modulo termina con una `endprimitive`
- ciascuno dei segnali in ingresso è definito utilizzando la parola chiave `input` seguita dal nome del segnale. L'unico parmetro di uscita è definito da un identificatore preceduto dalla parola chiave `output`
- Il corpo del modulo è costituito da una tabella di verità compresa fra le keyword `table endtable`.
- Ciascuna riga della tabella di verità deve contenere:
  - tanti valori (ciascuno $\in \{0, 1, ?\}$) quanti sono gli ingressi, rappresentati nell'ordine
  - seguiti da un carattere "`:`"
  - seguito da uno 0 o un 1 che rappresenta il valore dell'uscita nel caso gli ingressi siano quelli specificati a sinistra dei `:`
  - ad esempio, una riga

    ```
    0 1 0 :  1
    ```

    a fronte dei parametri formali

    ```
    (output z, input x, input y, input w)
    ```

    indica che `z` varrà `1` quando `x = 0`, `y = 1` e `w = 0`.

- il valore `?` indica un non specificato.
- tutte le combinazioni di ingresso devono essere definite nella tabella di verità. Non è possibile abbreviare una `table` omettendo le righe con uscita `0` come invece facciamo quando disegnamo le taelle di verità con carta e penna, per semplicità.

1.1 Esempio: calcolo della somma di due bit e di un riporto iniziale

La somma di due bit e un bit di riporto fa `0` se tutti i bit sono `0` o se solo 2 dei tre bit sono `1`, fa `1` se esattamente uno dei tre ingressi è `1` oppure se lo sono tutti e tre. Quindi il modulo può essere scritto come segue:

```
primitive fa_somma(output s, input r, input x1, input x2);

   table
      0 0 0 : 0 ;
      0 0 1 : 1 ;
      0 1 0 : 1 ;
      0 1 1 : 0 ;
      1 0 0 : 1 ;
      1 0 1 : 0 ;
      1 1 0 : 0 ;
      1 1 1 : 1 ;
   endtable

endprimitive
```

1.2 Esempio: multiplexer con due ingressi da 1 bit

In questo caso possiamo scrivere la tabella di verità in modo compatto utilizzando valori di ingresso "don't care" (non specificati) rappresentandoli con il simbolo `?`.

```
primitive mux2x1(output z, input c, input x1, input x2);
   table
      0 0 ? : 0 ;
      0 1 ? : 1 ;
      1 ? 0 : 0 ;
      1 ? 1 : 1 ;
   endtable

endprimitive
```

Il fatto che occorra specificare un'uscita per tutte le combinazioni degli ingressi impedisce di tralasciare le righe con uscita pari a `0`. Il modulo che segue compila ma non funziona, in quanto per le uscite non speicificate anzichè `0` si osserverà un'uscita col valore speciale `x` (non specificato).

```
// QUESTO NON FUNZIONA (INIZIO)
primitive mux2x1(output z, input c, input x1, input x2);

```

```
   table
      0 1 ? : 1 ;
      1 ? 1 : 1 ;
   endtable

endprimitive
// QUESTO NON FUNZIONA (FINE)
```

2 Componenti descritti con espressioni booleane

Per definire un componente utilizzando expressioni dell'algebra booleana, utilizziamo moduli introdotti dalla keyword `module` invece che `primitive`. Nel corpo del modulo utilizzeremo il comando di "assegnamento continuo" `assign` che, seguito da uno statement di assegnamento, assegna il valore del risultato dell'espressione a destra dell'uguale alla variabile a sinistra dell'uguale *ogni volta che cambia uno dei valori di ingressi coinvolti nell'espressione a destra dell'uguale*. Per esempio `assign x = y;` assegna il valore di `y` a `x` ogni volta che `y` cambia. Si possono utilizzare operatori che rappresentano le operazioni booleane AND, OR e NOT:

Operatore	**Rappresentazione verilog**
and (bitwise)	&
or (bitwise)	|
not	~
and (logical)	&&
or (logical)	||
not	!

(da notare che per valori da 1 bit gli operatori bitwise sono equivalenti a quelli logici)

Differentemente dai moduli `primitive` in un `module` possiamo definire un numero arbitrario di uscite. Per ognuna delle uscite dovremmo utilizzare uno statement `assign` separato.

Per convenzione[1], le uscite sono elencate per prime nella lista dei parametri del modulo `module` (esattamente come avviene per l'unica uscita di un modulo `primitive`).

Infine, nel calcolo dell'espressione da assegnare che si trova a destra del simbolo `=` possiamo utilizzare l'espressione condizionale `( cond ? then : else)` come in C/C++.

2.1 Esempio: calcolo della somma di due bit e di un riporto iniziale

[1] non è necessario, ma conviene seguire la convenzione nel nostro codice

```
module somma(output riporto, output z,
             input riportoiniziale, input x, input y);

  assign

    z = (~riportoiniziale & ~x & y) |
        (~riportoiniziale & x & ~y) |
        (riportoiniziale & ~x & ~y) |
        (riportoiniziale & x & y);

  assign

    riporto = (~riportoiniziale & x & y) |
              (riportoiniziale & ~x & y) |
              (riportoiniziale & x) ;

endmodule
```

In questo esempio osservate due cose:

- il modulo definisce due bit di uscita, non uno solo, e
- nel calcolo del riporto abbiamo considerato con il terzo termine in OR sia la penultima che l'ultima riga della tabella di verità corrispondente, visto che le righe differiscono per il solo input `y` e dunque è come se avessimo raccolto un $(\overline{y} + y)$

Inoltre, avremmo potuto codificare in modo più compatto la parte del riporto. Guardando la tabella di verità del riporto:

```
primitive fa_riporto(output s, input r, input x1, input x2);

   table
      0 0 0 : 0 ;
      0 0 1 : 0 ;
      0 1 0 : 0 ;
      0 1 1 : 1 ;
      1 0 0 : 0 ;
      1 0 1 : 1 ;
      1 1 0 : 1 ;
      1 1 1 : 1 ;
   endtable

endprimitive
```

possiamo osservare che il riporto è `1` quando:

- il riporto iniziale è `0` e entrambi gli ingressi sono `1`, oppure
- il riporto iniziale è `1` e almeno uno degli ingressi è `1`.

questo si può tradurren nel `module somma` descritto poche righe sopra a questa sostituendo la seconda `assign` con questo codice:

```
assign riporto = (riportoiniziale == 0 ? (x & y) : (x | y);
```

3 Componeneti descritti in modo strutturale

Il terzo e ultimo tipo di componenti che consideriamo sono quelli descritti in modo "strutturale" ovvero come reti di altri componenti predefiniti come `primitive`, `module` o a loro volta in modo strutturale.

Per definire un componente come rete di sottocomponenti occorre:

- definire in modulo `module` con un proprio nome e la sua lista di segnali in uscita e in ingresso;
- definire tanti `wire` quanti sono i collegamenti necessari fra un segnale in uscita da uno dei moduli componenti e un segnale in ingresso di un altro modulo componente
- dichiarare un'istanza di modulo per ognuno dei moduli componenti. Le istanze di modulo si dichiarano utilizzando come tipo il nome (identificatore) utilizzato nella definizione del modulo[2], un identificatore che da' il nome all'istanza e una lista di parametri attuali, che verranno collegati nell'ordine ai parametri formali del modulo.
- utilizzare i nomi dei segnali in ingresso e uscita (parametri formali) e degli eventuali `wire` così definiti come parametri attuali delle istanze dei moduli, rispettando la semantica dei collegamenti che vogliamo implementare.

La struttura di un modulo così fatto sarà dunque qualcosa tipo:

```
module NOME(....);

  wire ...;

  NOMETIPOMODULO ID1(...);
  ...
  NOMETIPOMODULO IDK(...);

endmodule
```

3.1 Esempio: sommatore di due bit

Utilizziamo come componenti i due moduli `primitive` definiti nella sezione 1.1 e 2, ovvero i moduli `fa_somma` e `fa_riporto`.

Entrambi i moduli prendono in ingresso gli stessi parametri in ingresso del modulo che calcola la somma, ma uno calcola il bit di risultato e l'altro calcola il bit di riporto.

```
module fulladder(output riporto, output risultato,
                 input riportoiniziale, input x1, input x2);

   fa_riporto m1(riporto, riportoiniziale, x1, x2);
   fa_somma   m2(risultato, riportoiniziale, x1, x2);
```

[2] l'identificatore che è stato utilizzato fra la parola chiave `primitive` o `module` e la lista dei parametri formali

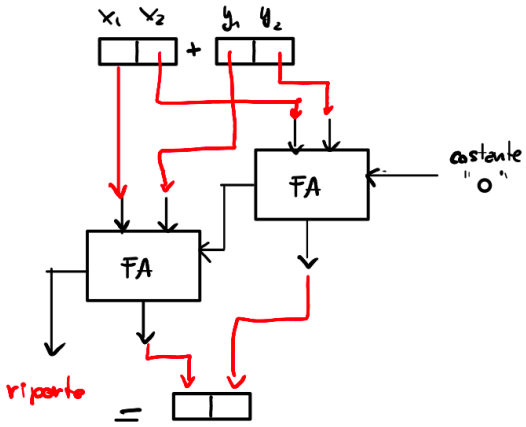

Figure 1: Addizionatore di due numeri da 2 bit costruito utilizzando due addizionatori di numeri da 1 bit con riporto

```
6
7 endmodule
```

4 Esempio: sommatore di due numeri da 2 bit

Supponendo di avere a disposizione un modulo `fulladder` come quello definito nella sezione 3.1, possiamo ottenere un `fulladder2` che opera su numeri a due bit collegando il riporto in uscita di un `fulladder` che somma i due bit meno significativi dei due ingressi e il riporto iniziale (presumibilmente `0`) ad un `fulladder` che somma i due bit più significativi dei due ingressi al riporto dell'altro `fulladder`, generando il bit più significativo del risultato `e` il bit di riporto finale (vedi schema in Fig. 1).

Tenendo presente che segnali da più di un bit si possono dichiarare facendo precedere l'identificatore da una coppia di interi fra parentesi quadre, separati da `:`, che rappresentano il valore del primo e dell'ultimo indice da utilizzare per accedere i singoli bit (vedi sezione 5), il modulo può essere definito come segue:

```
1 module add2(output riporto, output [1:0]somma,
2             input ripin, input [1:0]x1, input [1:0]x2);
3
4   wire      rips;
5
6   fulladder fa0(rips, somma[0], ripin, x1[0], x2[0]);
7   fulladder fa1(riporto, somma[1], rips, x1[1], x2[1]);
8
9 endmodule
```

Si noti il `rips` che serve unicamente a collegare il riporto in uscita dal primo modulo al riporto in entrata al secondo modulo. Il nome del wire compare quindi

- al posto del parametro attuale che corrisponde al formale riporto oin uscita del primo modulo, e
- al posto del parametro attuale che corrsiponde al formale riporto in ingresso del secondo modulo.

5 Variabili e costanti

5.1 Variabili da più di un bit

In Verilog possiamo dichiarare variabili che rappresentano registri (stato interno) mediante la parola chiave `reg`, e fili (collegamenti fra moduli) mediante la parola chiave `wire`. Ai registri possono essere assegnati valori nel programma di test, mentre dei `wire` ha senso solo leggere che valore portano o utilizzarli per collegamenti. Qualsiasi dichiarazione di una variabile, sia `reg` che `wire` ottenuta indicando solo l'identificatore da utilizzare definisce valori da 1 bit:

- `wire x;`
  definisce un filo capace di trasportare un singlo bit,
- `reg y;`
  definisce un registro capace di memorizzare un singolo bit.

Qualora volessimo utilizzare più bit (in un registro o in un filo) dobbiamo far precedere l'indentificatore da una coppia di parentesi quadre che al loro interno contengono un indice che rappresenta il bit più significativo e un indice che rappresenta il bit meno significativo, separati dal simbolo `:` (vedi Fig. 2). La possibilità di utilizzare un formalismo che permette di indicare range di indice diversi offre possibilità che si adattano a qualunque esigenza di rappresentazione degli indici:

- `reg [7:0] b1;`
  dichiara un registro da 8 bit (valore assoluto di (7-0) più 1). Il bit più significativo è `b1[7]` e quello meno significativo è `b1[0]`. In questo caso l'indice indica il peso del bit: se `b1[i]==1` allora il suo peso è 2^i altrimenti il suo peso è `0`.
- `reg [0:7] b;2`
  dichiara un registro da 8 bit (valore assoluto di (7-0) più 1). Il bit più significativo è `b1[0]` e quello meno significativo è `b1[7]`.
- `reg [1:8] b3;`
  dichiara un registro da 8 bit (valore assoluto di (7-0) più 1). Il bit più significativo è `b1[1]` e quello meno significativo è `b1[8]`. L'indice indica la posizione del bit, secondo l'ordinamento "naturale" da destra a sinistra, partendo da 1.

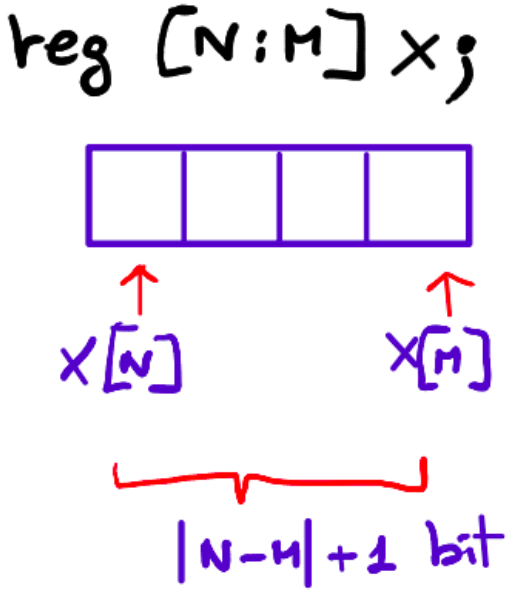

Figure 2: Dichiarazione di un registro da più bit

5.2 Costanti

In Verilog, le costanti possono essere espresse indicando quanti bit occupano, la base e un numero espresso in quella base:

- `4'b0111` indica la costante 7_{10} rappresentata su 4 bit in binario. Quindi il "4" sta per 4 bit, la "b" per binario e la stringa "0111" rappresenta il valore
- `4'd7` indica la stessa costante, espressa in **d**ecimale
- `4'o07` indica la stessa costante, espressa in base **o**ttale
- `8'xff` indica il numero 255, espresso in esadecimale (he**x**adecimal)

5.3 Giustapposizione

Due valori da un certo numero di bit (per esempio n ed m bit) possono essere utilizzati al posto di numeri da $n + m$ bit indicandoli, nell'ordine voluto, fra parentesi graffe. L'espressione Verilog `{2'b01,4'd4}` indica il numero formato dalla giustapposizione dei valori binari `01` e `0100` quindi il valore `6'b010100`.

5.4 Campi di un valore da n bit

Possiamo estrarre una configurazione di bit adiacenti da un valore di n utilizzando fra parentesi quadre, dopo l'identificatore, l'indice di partenza e quello di arrivo del campo da considerare:

- se `b` fosse dichiarato come `reg [7:0] b;` (un byte), allora per prendere il nibble meno significativo potrei utilizzare `b[3:0]` e per quello più significativo `b[7:4]`
- per mascherare la parte bassa del valore `b` potrei utilizzare l'espressione `{b[7:4],4'b0000}` oltre che, naturalmente, la classica espressione che utilizza una operazione di tipo AND con una costante "maschera" ovvero `b & 8'b00001111`[3].

6 Moduli parametrici

A volte è utile definire moduli parametrici. Verilog permette di definire identificatori come parmetri con un certo valore all'interno di un modulo con la sintassi `parameter ID = value;`. L'identificatore può essere usato ovunque nel modulo (incluso nella lista dei parametri) per denotare il valore `value`. Il valore del parametro può essere cambiato in fase di istanziazione facendo precedere all'identificatore dell'istanza una coppie di parentesi tonde precedute dal cancelletto che contengono la lista (ordinata) dei valori da assegnare ai parametri del modulo. Se c'è un unico parametro, allora le parentesi racchideranno un unico valore.

6.1 Multiplexer da due ingresso con numero di bit parametrico

Un multiplexer che sceglie fra due ingressi, ciascuno di N bit può essere programmato come segue:

```
1  module mux(output [N-1:0] z,
2             input ctrl, input [N-1:0] x1, input [N-1:0] x2);
3
4    parameter N = 8;
5
6    assign
7      z = (ctrl == 0 ? x1 : x2);
8
9  endmodule
```

Qualora volessimo utilizzare un multiplexer che scegli fra valori da 16 bit invece che da 8 bit, dovremmo istanziarlo come segue:

```
1    reg [15:0] x1;
2    reg [15:0] x2;
3    reg ctrl;
```

[3] in forma più compatta `b & 8'x0f`

```
4   wire [15:0] z;
5
6   mux #(16) mux16(z,ctrl,x1,x2);
```

6.2 Ritardi

Nel testbench (programma di prova di un modulo) abbiamo utilizzato la sintassi `#3 x=0;` che significa "attendi 3 unità di tempo e poi assegna 0 a x". Possiamo denotare l'attesa di 5 unità di tempo inserendo il comando `#5;`. Possiamo anche definire un ritardo anche nelle `assign` di un modulo (facciamo precedere all'identificatore cui si assegna il valore un termine `#n` che indichi il ritardo fra la valutazione della parte destra e l'assegnamento del valore risultante alla parte sinistra dell'espressione, per modellare in modo esplicito i ritardi[4]. Noi intenderemo un `#1` come un singolo Δt. Volendo quindi associare un ritardo ai nostri moduli, potremo anche vedere che succede a livello temporale nella comparsa dei risultati dopo il cambiamento degli ingressi dei nostri moduli sotto test. Per esempio, potremmo modificare il modulo `somma` della sezione 2.1 come segue:

```
1  module somma(output riporto, output z,
2               input riportoiniziale, input x, input y);
3
4    assign
5      // ritardo di due delta t : uno per il livello AND
6      // e uno per il livello  OR
7      #2 z = (~riportoiniziale & ~x & y) |
8             (~riportoiniziale & x & ~y) |
9             (riportoiniziale & ~x & ~y) |
10            (riportoiniziale & x & y);
11
12   assign
13
14     #2 riporto = (~riportoiniziale & x & y) |
15                  (riportoiniziale & ~x & y) |
16                  (riportoiniziale & x) ;
17
18 endmodule
```

In questo modo potremo vedere come le uscite del sommatore avvengano dopo 2 unità di tempo dalla stabilizzazione degli ingressi. Senza modificare il programma che crea un addizionatore di numeri da due bit a partire da questo fulladder da 1 bit (vedi sezione 4), potremmo anche vedere che il risultato finale si ha dopo $4\Delta t$ per via del collegamento in cascata dei due fulladder (ciascuno con ritardo da $2\Delta t$).

[4] questo vale solo per la simulazione di un circuito, non per la sintesi