

# Documentazione Verilog per Esercitazioni di Reti Logiche A.A. 2024/25

Raffaele Zippo

7 ottobre 2025

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Operatori</b>	<b>4</b>
2.1	Valori letterali ( <i>literal values</i> )	4
2.1.1	Estensione e troncamento	4
2.2	Operatori aritmetici	4
2.3	Operatori logici e <i>bitwise</i>	4
2.3.1	<i>Reduction operators</i>	5
2.4	Operatore di selezione [...]	5
2.5	Operatore di concatenazione {...}	6
2.5.1	Operatore di replicazione N{...}	6
2.6	Operazioni comuni	6
2.6.1	Estensione di segno	6
2.6.2	Shift a destra e sinistra	6
<b>3</b>	<b>Sintassi per reti combinatorie</b>	<b>7</b>
3.1	module	7
3.1.1	input e output	7
3.2	wire	7
3.3	Usare un module in un altro module	8
3.4	Tabelle di verità	8
3.5	Multiplexer	9
3.6	Reti parametrizzate	9
<b>4</b>	<b>Sintassi per reti sincronizzate</b>	<b>11</b>
4.1	Istanziamento	11
4.2	Collegamento a wire	11
4.3	Struttura generale di un blocco always	12
4.4	Comportamento al reset	12
4.5	Aggiornamento al fronte positivo del clock	12
4.6	Limitazioni della simulazione: temporizzazione, non-trasparenza e operatori di assegnamento	13
<b>5</b>	<b>Simulazione ed uso di GTKWave</b>	<b>14</b>
5.1	Compilazione e simulazione	14
5.1.1	Testbench con `timescale	15
5.2	Waveform e debugging	15
5.2.1	Zoom, ordinamento, formattazione	16
5.2.2	Non specificati e alta impedenza	16
5.2.3	Pulsante <i>Reload</i>	16
5.2.4	Linea di errore	16
<b>6</b>	<b>Essere efficienti con VS Code</b>	<b>18</b>
6.1	Le basi elementari	18
6.2	Le basi un po' meno elementari	18
6.3	Editing multi-caret	19

\*

# Capitolo 1

## Introduzione

Questa documentazione è organizzata per fornire riferimenti rapidi per ciascun contesto d'uso del Verilog. Nel far questo, prendiamo in considerazione il fatto che in Verilog la stessa sintassi può avere usi diversi in contesti diversi: per esempio, si parlerà in modo diverso di `reg` per testbench simulative rispetto a come se ne parla per reti sincronizzate.

Le definizioni “vere” di queste sintassi sono più astratte di quanto presentato qui, proprio per accomodare usi diversi. Un esempio di documentazione più completa ma non orientata agli usi di questo corso si trova su [www.chipverif.com](http://www.chipverif.com).

## Capitolo 2

# Operatori

### 2.1 Valori letterali ( *literal values* )

In ogni linguaggio, i *literal values* sono quelle parti del codice che rappresentano valori costanti. Per ovvi motivi, in Verilog questi sono principalmente stringhe di bit.

La definizione (completa) di un valore letterale è data da

1. dimensione in bit
1. formato di rappresentazione
1. valore

Per esempio, `4'b0100` indica un valore di 4 bit, espressi in notazione *binaria*, il cui valore in binario è `0100`. Le altre notazioni che useremo sono `d` per decimale ( `4'd7` corrisponde al binario `0111` ) e `h` per esadecimale ( `8'had` corrisponde al binario `10101101` ).

#### 2.1.1 Estensione e troncamento

Verilog automaticamente estende e tronca i letterali la cui parte valore è sopra o sottospecificata rispetto al numero di bit. Per esempio, `4'b0` viene automaticamente esteso a `4'b0000`, mentre `6'had` viene automaticamente troncato a `6'b101101`.

### 2.2 Operatori aritmetici

Il Verilog supporta molti degli operatori comuni, che possiamo usare in espressioni combinatorie: `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`, `==`.

Prestare attenzione, però, ai dimensionamenti in bit degli operandi e a come Verilog li estende per eseguire le operazioni.

### 2.3 Operatori logici e *bitwise*

Verilog supporta i classici operatori logici `&&`, `||` e `!`. Questi lavorano su valori booleani ( `0` è `false`, diverso da zero è `true` ), e producono un solo bit come risultato.

Operatore logico	Tipo di operazione
<code>&amp;&amp;</code>	and
<code>  </code>	or
<code>!</code>	not

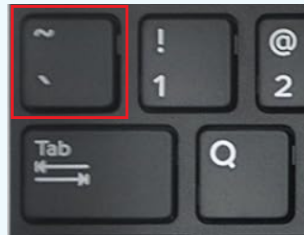
Questi vanno distinti dagli operatori *bitwise* (in italiano *bit a bit* ), che lavorano invece per un bit alla volta (e per bit corrispondenti) producendo un risultato delle stesse dimensioni degli operandi.

Operatore bitwise	Tipo di operazione
$\&$	and
$\sim\&$	nand
$ $	or
$\sim $	nor
$\wedge$	xor
$\sim\wedge$	xnor
$\sim$	not

Per indicare porte logiche, utilizzare gli operatori bitwise.

#### Come scrivere la tilde ~

Nel layout di tastiera QWERTY internazionale, la tilde ha un tasto dedicato, a sinistra dell'1.



Nel layout di tastiera QWERTY italiano, invece, la tilde non è presente. Ci sono 3 opzioni:

3. passare al layout QWERTY internazionale
3. imparare scorciatoie alternative, che dipendono dal sistema operativo
3. usare scripting come AutoHotkey per personalizzare il layout

L'opzione 1 richiede di imparare un layout diverso, ma è consigliabile per tutti gli usi di programmazione dato che risolve altri problemi come il backtick ` e rende più semplici da scrivere caratteri come [ ] e ;. [Qui](#) le istruzioni per cambiare layout su Windows.

L'opzione 2 varia da sistema a sistema. Su Windows, la combinazione di tasti è `alt + 126`, facendo attenzione a digitare il numero usando il tastierino numerico e non la riga dei numeri.

L'opzione 3 non è utilizzabile all'esame. Per uso personale, vedere [qui](#).

### 2.3.1 Reduction operators

I *reduction operators* applicano un'operazione tra tutti i bit di un elemento di più bit, producendo un risultato su un solo bit. Sia per esempio  $x$  di valore `4'b0100`, allora la sua riduzione `and x`, equivalente a `x[3] & x[2] & x[1] & x[0]`, varrà `1'b0`; mentre la sua riduzione `or, | x`, varrà `1'b1`. Le riduzioni possono rendere alcune espressioni combinatorie più semplici da scrivere.

Operatore	Tipo di riduzione
$\&$	and
$\sim\&$	nand
$ $	or
$\sim $	nor
$\wedge$	xor
$\sim\wedge$	xnor

## 2.4 Operatore di selezione [...]

Quando si dichiara un elemento, come un *wire*, si utilizza la notazione `[N:0]` per indicare l'elemento ha  $N+1$  bit, indicizzati da 0 a  $N$ . Per esempio, per dichiarare un filo da 8 bit, scriveremo

```
wire [7:0] x;
```

Possiamo poi utilizzare l'operatore per selezionare uno o più bit di un tale componente. Per esempio, possiamo scrivere `x[2]`, che seleziona il bit di posizione 2 (*bit-select*), e `x[6:3]`, che seleziona i quattro bit dalla posizione 6 alla posizione 3 (*part-select*).

## 2.5 Operatore di concatenazione {...}

L'operatore di concatenazione viene utilizzato per combinare due o più espressioni, vettori, o bit in un'unica entità.

```
input [3:0] a, b;
wire [7:0] ab;
assign ab = a, b;
```

L'operatore può anche essere usato a sinistra di un assegnamento.

```
input [7:0] x;
wire [3:0] xh, xl;
assign xh, xl = x;
```

### Maneggiare fili non ha nessun costo

Questo operatore corrisponde, circuitalmente, al semplice raggruppare o separare dei fili. Non è un'operazione combinatoria, e per questo non consuma tempo. È per questo che negli esempi sopra gli assign non hanno alcun ritardo #T.

### 2.5.1 Operatore di replicazione N{...}

L'operatore di replicazione semplifica il tipico caso d'uso di ripetere un bit o un gruppo di bit  $N$  volte. Si può utilizzare solo all'interno di un concatenamento che sia a *destra* di un assegnamento e con  $N$  costante. È equivalente a scrivere  $N$  volte ciò che si vuole ripetere.

```
input [3:0] x;
wire [15:0] x_repeated_4_times;
assign x_repeated_4_times = 4x; // equivalente a x, x, x, x
```

Il suo uso più comune è l'estensione di segno di interi, mostrato più avanti.

## 2.6 Operazioni comuni

### 2.6.1 Estensione di segno

Quando si estende un numero su più bit bisogna considerare se il numero è un naturale o un intero. Per estendere un naturale, basta aggiungere degli zeri.

```
wire [7:0] x_8;
wire [11:0] x_12;
assign x_12 = 4'h0, x_8;
```

Per estendere un intero, dobbiamo invece replicare il bit più significativo.

```
wire [7:0] x_8;
wire [11:0] x_12;
assign x_12 = 4x_8[7], x_8;
```

### 2.6.2 Shift a destra e sinistra

Per fare shift a destra e sinistra ci basta utilizzare gli operatori di selezione e concatenamento. Lo shift a sinistra è lo stesso per numeri naturali e interi, posto che non ci sia overflow.

```
input [7:0] x;
wire [7:0] x_mul_4;
assign x_mul_4 = x[5:0], 2'b0;
```

Lo shift a destra richiede invece di considerare il segno, se stiamo lavorando con interi.

```
input [7:0] x; // rappresenta un numero naturale
wire [7:0] x_div_4;
assign x_div_4 = 2'b0, x[7:2];
```

```
input [7:0] x; // rappresenta un numero intero
wire [7:0] x_div_4;
assign x_div_4 = 2x[7], x[7:2];
```

## Capitolo 3

# Sintassi per reti combinatorie

Una rete combinatoria si esprime come un `module` composto solo da `wire`, espressioni combinatorie e componenti che sono a loro volta reti combinatorie.

### 3.1 module

Il blocco `module ... endmodule` definisce un *tipo* di componente, che può poi essere istanziato in altri componenti. La dichiarazione di un `module` include il suo nome e la lista delle sue porte.

```
module nome_rete ( porta1, porta2, ... );  
    ...  
endmodule
```

#### 3.1.1 input e output

Per ciascuna porta di un `module`, dichiariamo se è di `input` o `output`, e di quanti bit è composta. Se non specificata, la dimensione default è 1. La dichiarazione di porte con le stesse caratteristiche si può fare nella stessa riga.

Le porte `input` sono dei `wire` il cui valore va assegnato *al di fuori* di questa rete.

Le porte `output` sono dei `wire` il cui valore va assegnato *all'interno* di questa rete.

```
module nome_rete ( porta1, porta2, porta3, porta4 );  
    input [3:0] porta1, porta2;  
    output [3:0] porta3;  
    output porta4;  
    ...  
endmodule
```

#### **inout**

Non usiamo porte `inout` nelle reti combinatorie.

### 3.2 wire

Un `wire` è un filo che trasporta un valore logico. Se non specificata, la dimensione default è 1. La dichiarazione di `wire` con le stesse caratteristiche si può fare nella stessa riga.

```
wire [3:0] w1, w2;  
wire w3, w4, w5;
```

Con uno statement `assign` possiamo associare al `wire` una *espressione combinatoria*: il `wire` assumerà continuamente il valore dell'espressione, rispondendo ai cambiamenti dei suoi operandi. Lo statement `assign` può includere un fattore di ritardo, `#T`, per indicare che il valore del filo segue il valore dell'espressione con ritardo di `T` unità.

```
assign #1 w5 = w3 & w4;
```

Un `wire` può essere associato a una porta di un `module`, come mostrato nella sezione successiva.

### 3.3 Usare un module in un altro module

Una volta definito un module, possiamo istanziare componenti di questo *tipo* in un altro module.

```
nome_module nome_istanza (
    .porta1(...), .porta2(...), ...
);
```

Questo corrisponde, circuitualmente, al prendere un componente fisico di tipo `nome_module`, chiamato `nome_istanza` per distinguerlo dagli altri, e posizionarlo nella nostra rete collegandone i vari piedini con altri elementi.

All'interno degli statement `.porta(...)` specifichiamo quale porta, espressione o wire del module corrente va collegato alla porta del module istanziato.

Insieme agli statement `assign` e l'uso di `wire`, questo ci permette di comporre reti combinatorie su diversi livelli di complessità e con poca duplicazione del codice.

Come esempio, costruiamo un `and` a 1 ingresso e lo usiamo per comporre un `and` a 3 ingressi.

```
module and(a, b, z);
    input a, b;
    output z;

    assign #1 z = a & b;
endmodule

module and2(a, b, c, z);
    input a, b, c;
    output z;

    wire z1;
    and a1(
        .a(a), .b(b),
        .z(z1)
    );

    and a2(
        .a(c), .b(z1),
        .z(z)
    );
endmodule
```

### 3.4 Tabelle di verità

Talvolta il modo più immediato per esprimere una rete combinatoria è tramite la sua tabella di verità. È anche noto che data una tabella di verità possiamo ottenere una sintesi della rete combinatoria, utilizzando metodi come le mappe di Karnaugh.

In Verilog, il modo più immediato di esprimere una tabella di verità è utilizzando una catena di operatori ternari.

```
module and(x, y, z);
    input x, y;
    output z;
    assign #1 z =
        (x,y == 2'b00) ? 1'b0 :
        (x,y == 2'b01) ? 1'b0 :
        (x,y == 2'b10) ? 1'b0 :
        /*x,y == 2'b11*/ 1'b1;
```

Un'alternativa è l'uso di `function` e `casex`.

```
module and(x, y, z);
    input x, y;
    output z;
    assign #1 z = tabella_verita(a, b);

    function tabella_verita;
        input [1:0] ab;
        casex(ab)
            2'b00: tabella_verita = 1'b0;
            2'b01: tabella_verita = 1'b0;
            2'b10: tabella_verita = 1'b0;
            2'b11: tabella_verita = 1'b1;
        endcase
    endfunction
endmodule
```

Per indicare tabelle di verità con più di un bit in uscita si scrive, per esempio, `function [1:0] tabella_verita;`. Nel `casex` si può utilizzare anche un caso default, scrivendo come ultimo caso `default: tabella_verita = ...;`.



**Attenzione all'uso delle function**

Le *function* sono blocchi di *codice da eseguire*, parti del *behavioral modelling* di Verilog. Il simulatore ne svolge i passaggi come un programma, senza consumare tempo e senza alcun corrispettivo hardware previsto. È per questo, per esempio, che dobbiamo specificare noi il tempo consumato nello statement *assign*. L'uso mostrato qui delle *function* è l'unico ammesso per una *sintesi* di reti combinatorie. In presenza di ogni altra elaborazione algoritmica, di cui non sia evidente il corrispettivo hardware, sarà invece considerata una *descrizione* di rete combinatoria.

## 3.5 Multiplexer

I multiplexer sono da considerarsi noti e sintetizzabili, e si possono esprimere con uno o più operatori ternari ?.

**Operatore ternario**

La sintassi è della forma *cond ? v\_t : v\_f*, dove *cond* è un predicato (espressione *true* o *false*) mentre *v\_t* e *v\_f* sono espressioni dello stesso tipo.

L'espressione ha valore *v\_t* se il predicato *cond* è *true*, *v\_f* altrimenti.

Per un multiplexer con selettore a 1 bit, basterà un solo ?.

```
input sel;
assign #1 multiplexer = sel ? x0 : x1;
```

Per un selettore a più bit si dovranno usare in serie per gestire più casi

```
input [1:0] sel;
assign #1 multiplexer =
    (sel == 2'b00) ? x0 :
    (sel == 2'b01) ? x1 :
    (sel == 2'b10) ? x2 :
    /*sel == 2'b11*/ x3 ;
```

**Differenza tra multiplexer e tabella di verità**

La sintassi qui mostrata sembra identica a quella mostrata poco prima per le tabelle di verità. Sono quindi la stessa cosa? **No**.

Dato uno specifico ingresso, una rete combinatoria avrà come uscita sempre il valore corrispondente nella tabella di verità, che è specifico e costante (a meno di *non specificati*). Per un multiplexer, invece, l'uscita è il valore di uno degli ingressi, che è libero di mutare. Le realizzazioni circuitali di questi componenti sono completamente diverse.

Per la sintassi Verilog, invece, la differenza è da poco (prendere un *right hand side* da una variabile o da un letterale). Di nuovo, è importante stare attenti a *cosa si sta facendo* quando si scrive codice Verilog.

## 3.6 Reti parametrizzate

In un *module* si possono definire parametri per generalizzare la rete. In particolare, questo è utilizzato in *reti\_standard.v* per fornire reti il cui dimensionamento va specificato da chi le utilizza.

Per esempio, vediamo come è definita una rete di somma a N bit.

```
module add(
    x, y, c_in,
    s, c_out, ow
);
    parameter N = 2;

    input [N-1:0] x, y;
    input c_in;

    output [N-1:0] s;
    output c_out, ow;

    assign #1 c_out, s = x + y + c_in;
    assign #1 ow = (x[N-1] == y[N-1]) && (x[N-1] != s[N-1]);
endmodule
```

Con *N = 2* viene impostato il valore di default del parametro. Quando istanziamo la rete altrove, possiamo modificare questo parametro, per esempio per ottenere un sommatore a 8 bit.

```
add #( .N(8) ) a (
    ...
);
```

Un module può avere più di un parametro, che possono essere impostati indipendentemente.

```
nome_modulo #( .nome_parametro1(v1), .nome_parametro2(v2)... ) nome_istanza (
    ...
);
```

#### Immutabilità dei parametri

I parametri determinano la quantità di hardware, che non può essere cambiata mentre la rete è in uso. I valori associati devono essere costanti.

#### Parametrizzazione e sintesi di reti combinatorie

La parametrizzazione è facilmente applicabile a *descrizioni* di reti combinatorie dove si usano espressioni combinatorie che il simulatore è facilmente in grado di adattare a diverse quantità di bit.

È molto più complicato applicarla a *sintesi* di reti combinatorie, dato che non si possono istanziare componenti in modo parametrico, per esempio  $N$  full adder da 1 bit per sintetizzare un full adder a  $N$  bit.

## Capitolo 4

# Sintassi per reti sincronizzate

Una rete sincronizzata si esprime come un `module` contenente registri, che sono espressi con `reg` il cui valore è inizializzato in risposta a `reset_` ed aggiornato in risposta a fronti positivi del `clock`.

Gran parte della sintassi già vista per le reti combinatorie rimane valida anche qui, e dunque non la ripetiamo. Ci focalizziamo invece su come esprimere registri usando `reg`.

### 4.1 Istanziamento

Un registro si istanzia con statement simili a quelli per `wire` :

```
reg [3:0] R1, R2;  
reg R3, R4, R5;
```

#### Nomi in maiuscole e minuscolo

Verilog è *case sensitive*, cioè distingue come diversi nomi che differiscono solo per la capitalizzazione, come `out` e `OUT`.

Nel corso, utilizziamo questa feature per distinguere a colpo d'occhio `reg` e `wire`, utilizzando lettere maiuscole per i primi e minuscole per i secondi. Questo è particolarmente utile quando si hanno registri a sostegno di un `wire`, tipicamente un'uscita della rete o l'ingresso di un `module` interno.

Seguire questa convenzione non è obbligatorio, ma fortemente consigliato per evitare ambiguità ed errori che ne conseguono.

### 4.2 Collegamento a wire

Un `reg` si può utilizzare come “fonte di valore” per un `wire`. Questo equivale circuitalmente a collegare il `wire` all'uscita del `reg`.

```
output out;  
reg OUT;  
assign out = OUT;
```

In questo caso, `out` seguirà sempre e in modo continuo il valore di `OUT`, propagandolo a ciò a cui viene collegato a sua volta. In questo caso non introduciamo nessun ritardo `#T` nell' `assign` perché si tratta di un semplice collegamento senza logica combinatoria aggiunta.

Allo stesso modo, si può collegare un `reg` all'ingresso di una rete.

```
reg [3:0] X, Y;  
add #( .N(4) ) a(  
    .x(X), .y(Y), .c_in(1'b0),  
    ...  
);
```

Non ha invece alcun senso cercare di fare il contrario, ossia collegare direttamente un `wire` all'ingresso di un `reg`. Anche se questo ha senso circuitalmente, Verilog richiede di esprimere questo all'interno di un blocco `always` per indicare anche *quando* aggiornare il valore del `reg`.

### 4.3 Struttura generale di un blocco always

Il valore di un reg si aggiorna all'interno di blocchi always. La sintassi generale di questi blocchi è la seguente

```
always @( event ) [if( cond )] [ #T ] begin
    [multiple statements]
end
```

Il funzionamento è il seguente: ogni volta che accade event, se cond è vero e dopo tempo T, vengono eseguiti gli statement indicati. Se lo statement è uno solo, si possono anche omettere begin e end.

Per Verilog, qui come *statement* si possono usare tutte le sintassi procedurali che si desiderano, incluse quelle discusse per le testbench che permettono di scrivere un classico programma “stile C”. Per noi, *no*. Useremo questi blocchi in dei modi specifici per indicare

3. come si comportano i registri al reset,
3. come si comportano i registri al fronte positivo del clock.

### 4.4 Comportamento al reset

Per indicare il comportamento al reset useremo statement del tipo

```
always @(reset_ == 0) begin
    R1 = 0;
end
```

Il funzionamento è facilmente intuibile: finché reset\_ è a 0, il reg è impostato al valore indicato. Il blocco begin ... end può contenere l'inizializzazione di più registri. Tipicamente, raggrupperemo tutte le inizializzazioni in una *descrizione*, mentre le terremo separate in una *sintesi*.

Un registro può non essere inizializzato: in tal caso, il suo valore sarà *non specificato*, in Verilog x. Ricordiamo che questo significa che il registro ha un qualche valore misurabile, ma non è possibile determinare logicamente a priori e in modo univoco quale sarà.

In un blocco reset è *indifferente* l'uso di = o <= per gli assegnamenti (vedere sezione più avanti).

#### Valore assegnato al reset

Per la sintassi Verilog, a destra dell'assegnamento si potrebbe utilizzare qualunque espressione, sia questa costante (per esempio, il letterale 1'b0 o un parameter ) o variabile (per esempio, il wire w ).

Se pensiamo però all'equivalente circuitale, hanno senso solo valori costanti. Infatti, impostare un valore al reset equivale a collegare opportunamente i piedini preset\_ e preclear\_ del registro.

### 4.5 Aggiornamento al fronte positivo del clock

Per indicare il comportamento al fronte positivo del clock useremo statement del tipo

```
always @(posedge clock) if(reset_ == 1) #3 begin
    OUT <= ~OUT;
end
```

Il funzionamento è il seguente: ad ogni fronte positivo del clock, se reset\_ è a 1 e dopo 3 unità di tempo, il registro viene aggiornato con il valore indicato. Differentemente dal reset, qui si può utilizzare qualunque logica combinatoria per il calcolo del nuovo valore del registro.

L'unità di tempo (impostato a 3 in questo corso solo per convenzione, così come il periodo del clock a 10 unità) rappresenta il tempo di propagazione  $T_{propagation}$  del registro, ossia il tempo che passa dal fronte del clock prima che il registro mostri in uscita il nuovo valore.

Tutti gli assegnamenti in questi blocchi devono usare l'operatore <=, e non =. Come spiegato nella sezione più avanti, questo è necessario perché i registri simulati siano non-trasparenti.

Tipicamente usiamo registri *multifunzionali*, ossia che operano in maniera diversa in base allo stato della rete.

In una *descrizione*, questo si fa usando un singolo registro di stato STAR e indicando il comportamento dei vari registri multifunzionali al variare di STAR. Questo ci fa vedere in generale come si comporta l'intera rete al variare di STAR. In questa notazione, è lecito omettere un registro in un dato stato, implicando che quel registro *conserva* il valore precedentemente assegnato.

```
localparam S0 = 0, S1 = 1;
always @(posedge clock) if(reset_ == 1) #3 begin
    casex(STAR)
        S0: begin
```

```

        A <= ~B;
        B <= A;
        STAR <= (A == 1'b0) ? S1 : S0;
    end
    S1: begin
        A <= B;
        B <= ~A;
        STAR <= (B == 1'b1) ? S1 : S0;
    end
endcase
end

```

In una *sintesi*, invece, si sintetizza ciascun registro individualmente come un multiplexer guidato da una serie di *variabili di comando*. Il multiplexer ha come ingressi *tutti* i risultati combinatori che il registro utilizza, e in base allo stato (da cui vengono generate le variabili di comando) solo uno di questi è utilizzato per aggiornare il registro al fronte positivo del clock. Questo è rappresentato in Verilog utilizzando le variabili di comando per discriminare il *case*, e indicando un comportamento combinatorio per ciascun valore di queste variabili. In questa notazione, non è lecito omettere le operazioni di conservazione, mentre è lecito utilizzare non specificati per indicare comportamenti assegnati a più ingressi del multiplexer. Nell'esempio sotto, con `2'b1X` si indica che a entrambi gli ingressi `10` e `11` del multiplexer è collegato il valore `DAV_`.

```

always @(posedge clock) if(reset_ == 1) #3 begin
    casex(b1, b0)
        2'b00: DAV_ <= 0;
        2'b01: DAV_ <= 1;
        2'b1X: DAV_ <= DAV_;
    endcase
end

```

## 4.6 Limitazioni della simulazione: temporizzazione, non-trasparenza e operatori di assegnamento

Ci sono alcune differenze tra i registri, intesi come componenti elettronici, e i *reg* descritti in Verilog così come abbiamo visto. Queste differenze non sono d'interesse *se non si fanno errori*. In caso di errori, si potrebbero osservare comportamenti altrimenti inspiegabili, ed è per questo che è utile conoscere queste differenze per poter risalire alla fonte del problema.

I registri hanno caratteristiche di temporizzazione sia prima che dopo il fronte positivo del clock: ciascun ingresso va impostato almeno  $T_{setup}$  prima del fronte positivo, mantenuto fino ad almeno  $T_{hold}$  dopo, e il valore in ingresso è rispecchiato in uscita solo dopo  $T_{propagation}$ .

Date le semplici strutture sintattiche che utilizziamo, la simulazione non è così accurata e non considera  $T_{setup}$  e  $T_{hold}$ . In particolare, il simulatore campiona i valori in ingresso non *prima* del fronte positivo, ma direttamente quando aggiorna il valore dei registri, ossia *dopo*  $T_{propagation}$  dal fronte positivo del clock.

In altre parole: tutti i campionamenti e gli aggiornamenti dei registri sono fatti allo stesso tempo di simulazione, ossia  $T_{propagation}$  dopo il fronte positivo del clock.

Questo porterebbe a violare la non-trasparenza dei registri, se non fosse per l'operatore di assegnamento `<=`, detto *non-blocking assignment*. Questo operatore si comporta in questo modo: tutti gli assegnamenti `<=` contemporanei (ossia allo stesso tempo di simulazione) non hanno effetto l'uno sull'altro perché campionano il *right hand side* all'inizio del time-step e aggiornano il *left hand side* alla fine del time-step.

Questo simula correttamente la non-trasparenza dei registri, ma solo se *tutti* usano `<=`. Gli assegnamenti con `=`, detti *blocking assignment*, sono invece eseguiti completamente e nell'ordine in cui li incontra il simulatore (si assuma che quest'ordine sia del tutto casuale).

Al tempo di reset questo ci è indifferente, perché sono (circuitalmente) leciti solo assegnamenti con valori costanti e non si possono quindi creare anelli per cui è di interesse la non-trasparenza.

## Capitolo 5

# Simulazione ed uso di GTKWave

Documentiamo qui il software da utilizzare per il testing e debugging delle reti prodotte, ossia iverilog, vvp e GTKWave. A differenza dell'ambiente per Assembler, questi sono facilmente reperibili per ogni piattaforma, o compilabili dal sorgente. In sede d'esame si utilizzano da un normale terminale Windows, senza utilizzare macchine virtuali. [Qui](#) si trovano installer per Windows.

Negli esercizi di esame vengono forniti i file necessari a compilare simulazioni per testare la propria rete. Questi sono tipicamente i file `testbench.v` e `reti_standard.v`. Il primo contiene una serie di test che verificano il corretto comportamento della rete prodotta rispetto alle specifiche richieste. Il secondo contiene invece delle reti combinatorie che si potranno assumere note e sintetizzabili, da usare per la sintesi di rete combinatoria.

Non tutti gli esercizi hanno una parte di sintesi di rete combinatoria, e quindi il file `reti_standard.v`. Inoltre, ciascun esercizio ha il *proprio* file `reti_standard.v`, che sarà diverso da quelli allegati ad altri esercizi.

### 5.1 Compilazione e simulazione

Sia `descrizione.v` il sorgente contenente la descrizione della rete sincronizzata da noi prodotto, e che vogliamo testare.

Si compila la simulazione con il comando da terminale `iverilog`. Il comando richiede come argomenti i file da compilare assieme. Di default, il binario prodotto si chiamerà `a.out`, mentre con l'opzione `-o nome` è possibile impostarne uno a scelta. Per esempio:

```
iverilog -o desc testbench.v reti_standard.v descrizione.v
```

Il file prodotto non è eseguibile da solo, ma va lanciato usando `vvp`. Per esempio:

```
vvp desc
```

Questo lancerà la simulazione. In un test di successo, vedremo le seguenti stampe:

```
VCD info: dumpfile waveform.vcd opened for output.  
$finish called at [un numero]
```

La prima stampa ci informa che il file `waveform.vcd` sta venendo popolato, la seconda ci informa del tempo di simulazione al quale questa è terminata con il comando `$finish`. Alcune versioni di `vvp` non stampano quest'ultima di default - non è un problema.

Le testbench degli esercizi d'esame stampano a video quando incontrano un errore: un test fallito avrà quindi delle righe in più in mezzo a quelle presentate qui. Per esempio, `Timeout - waiting for signal failed` indica che la simulazione si era bloccata in attesa di un evento che non è mai accaduto, come un segnale di handshake.

#### Le testbench non sono mai complete

Se la simulazione non stampa errori, questo indica solo che la testbench *non ne ha trovato alcuno*. Non implica, invece, che *non ci siano* errori. Questo sia perché è impossibile scrivere una testbench davvero esaustiva per tutti i possibili percorsi di esecuzione, ma anche perché è facile scrivere Verilog che *sembra* funzionare bene ma che in realtà usa costrutti che rendono la rete irrealizzabile in hardware.

È sempre responsabilità dello studente assicurarsi che non ci siano errori. In fase di autocorrezione, anche se la testbench non trova nessun errore, è sempre possibile (anzi, dovuto) assicurarsi della correttezza del compito e fare correzioni se necessarie.

### 5.1.1 Testbench con `timescale

Con la sintassi ``timescale` è possibile controllare l'unità di misura default e la granularità della simulazione. Per esempio, un file `testbench.v` che comincia come segue imposta l'unità di misura a 1s (il solito) e la granularità di simulazione a 1ms, permettendo di osservare cambiamenti più veloci di un secondo.

```
`timescale 1s/ms

module testbench();
...
```

Questa sintassi è utilizzata in alcuni testi d'esame, per esempio se sono previste RC particolarmente veloci. Per maggiori dettagli, vedere [qui](#).

Se la sintassi ``timescale` è utilizzata, è **obbligatorio** compilare la simulazione ponendo il file `testbench.v` come primo file del comando, ossia `iverilog -o desc testbench.v ....`

In caso contrario, il compilatore stamperà il seguente warning:

```
warning: Found both default and `timescale based delays.
```

## 5.2 Waveform e debugging

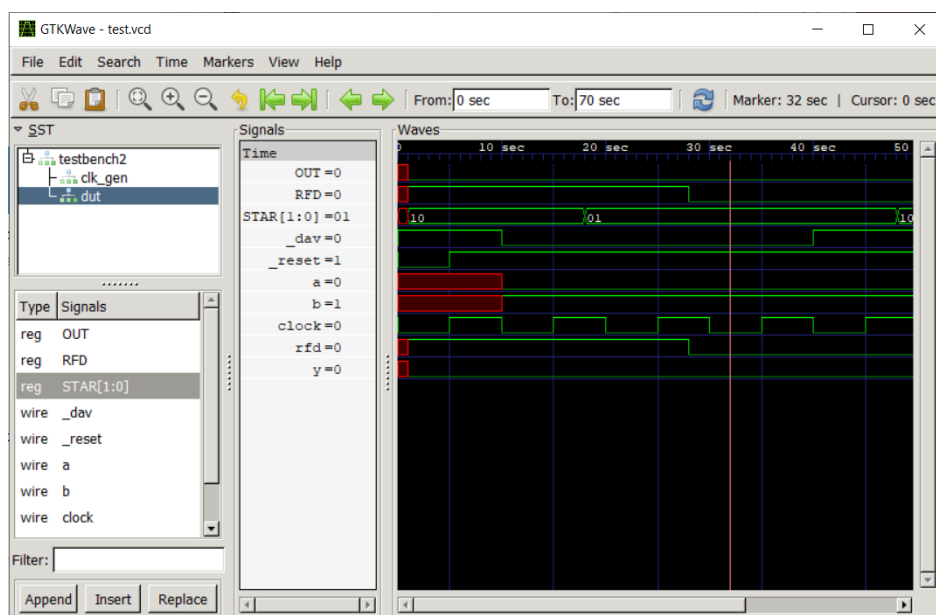
La simulazione genera un file `waveform.vcd` contenente l'evoluzione di tutti i fili e registri nella simulazione. Questo file è prodotto grazie alle seguenti righe, incluse in tutte le testbench:

```
initial begin
    $dumpfile("waveform.vcd");
    $dumpvars;
    ...
```

Con questo file possiamo studiare l'evoluzione della rete e trovare eventuali errori. Per analizzarlo, usiamo GTK-Wave, richiamabile da terminale con

```
gtkwave waveform.vcd
```

Si dovrebbe aprire quindi una finestra dal quale possiamo analizzare l'evoluzione della rete.



Il programma mostra sulla sinistra le varie componenti nella simulazione e, se li selezioniamo, i fili e registri che li compongono. Ci interesserà in particolare `dut` ( *device under test* ), che sarà proprio il componente da noi realizzato. Selezionando poi i vari `wire` e `reg` che compaiono sotto, e cliccando "Append", compariranno nella schermata a destra, dove possiamo vedere l'evoluzione nel tempo.

### 5.2.1 Zoom, ordinamento, formattazione

Lo zoom della timeline a destra è regolabile, usando la rotellina del mouse o le lenti d'ingrandimento in alto a sinistra. Cliccando in punti specifici della timeline spostiamo il cursore, cioè la linea rossa verticale. Possiamo quindi leggere nella colonna centrale il valore di ciascun segnale all'istante dove si trova il cursore.

I segnali nella schermata principale sono ordinabili, per esempio è in genere utile spostare `clock` e `STAR` in alto. Di default, sono formattati come segnali binari, se composti da un bit, o in notazione esadecimale, se da più bit. Cliccando col destro su un segnale è possibile cambiare la formattazione in diversi modi, incluso decimale.

### 5.2.2 Non specificati e alta impedenza


Prestare particolare attenzione ai valori non specificati ( `x` ) e alta impedenza ( `z` ), che sono spesso sintomi di errori, per esempio per un filo di input non collegato.

Nella waveform, i valori non specificati sono evidenziati con un'area rossa, mentre i fili in alta impedenza sono evidenziati con una linea orizzontale gialla posta a metà altezza tra 0 e 1.

### 5.2.3 Pulsante Reload

Il comando `gtkwave waveform.vcd` blocca il terminale da cui viene lanciato, rendendo impossibile mandare altri comandi finché non viene chiuso. È quindi frequente vedere studenti chiudere e riaprire GTKWave ogni volta che c'è bisogno di risimulare la rete.

Questo approccio è però inefficiente, dato che si dovrà ogni volta riselectare i fili, riformattarne i valori, ritrovare il punto d'errore che si stava studiando.

Il pulsante *Reload*, indicato con l'icona , permette di ricaricare il file `waveform.vcd` senza chiudere e riaprire il programma, e mantenendo tutte le selezioni fatte.

È per questo una buona idea utilizzare una delle seguenti strategie:

3. usare due terminali, uno dedicato a `iverilog` e `vvp`, l'altro a `gtkwave`;
3. lanciare il comando `gtkwave` in background. Nell'ambiente Windows all'esame, questo si può fare aggiungendo un `&` in fondo: `gtkwave waveform.vcd &`.

In entrambi i casi, otteniamo di poter rieseguire la simulazione mentre GTKWave è aperto, e poter quindi sfruttare il pulsante *Reload*.

#### Se l'operatore `&` non funziona

In alcune installazioni di Powershell l'operatore `&` non funziona. L'operatore è un semplice alias per `Start-Job`, e si può ovviare al problema usando questo comando per esteso:

```
Start-Job gtwave waveform.vcd
```

L'operatore è documentato [qui](#).

### 5.2.4 Linea di errore

Nelle testbench d'esame è (di solito) presente anche una *linea di errore* che permette di identificare subito i punti in cui la testbench ha trovato un errore. Questo è particolarmente utile per scorrere lunghe simulazioni.

Queste linee sono realizzate nella testbench con una variabile `reg error` inizializzata a 0 e un blocco `always` che risponde ad ogni variazione di `error` per rimetterla a 0 dopo una breve attesa. Questa attesa breve ma non nulla fa sì che basti assegnare 1 ad `error` per ottenere un'impulso sulla linea, facilmente visibile.

In GTKWave, possiamo trovare il segnale `error` tra i `wire` e `reg` del modulo `testbench` ( *non* in `dut` ). Mostrando questo segnale, possiamo riconoscere i punti di errore come impulsi, come nell'esempio seguente.





## Capitolo 6

# Essere efficienti con VS Code

VS Code è l'editor disponibile in sede d'esame e mostrato a lezione. Come ogni strumento di lavoro, è una buona idea imparare ad usarlo bene per essere più rapidi ed efficaci. Questo si traduce, in genere, nel prendere l'abitudine di usare meno il mouse e più la tastiera, usando le dovute scorciatoie e combinazioni di tasti.

In questa documentazione ci focalizziamo sulle combinazioni per Windows, che sono quelle che troverete all'esame. Evidenzierò con una ☆ le combinazioni più importanti e probabilmente meno note.

### Salvare i file

Fra le cause dei vari errori per cui riceviamo richieste d'aiuto, una delle più frequenti è che i file modificati non sono stati salvati. Un file modificato ma non salvato è indicato da un pallino nero nella tab in alto, e le modifiche non saranno visibili a altri programmi come gcc e iverilog.

Si consiglia di salvare spesso e abitualmente, usando `ctrl + s`.

## 6.1 Le basi elementari


Quando si scrive in un editor, il testo finisce dove sta il cursore (in inglese *caret*). È la barra verticale che indica dove stiamo scrivendo. Si può spostare usando le frecce, non solo destra e sinistra ma anche su e giù. Usando font monospace, infatti, il testo è una matrice di celle delle stesse dimensioni, ed è facile prevedere dove andrà il caret anche mentre ci si sposta tra le righe.

Vediamo quindi le combinazioni più comuni.

	Tasti	Cosa fa
	Tenere premuto shift	Seleziona il testo seguendo il movimento del cursore.
	<code>ctrl + c</code>	Copia il testo selezionato.
	<code>ctrl + v</code>	Incolla il testo selezionato.
	<code>ctrl + x</code>	Taglia (cioè copia e cancella) il testo selezionato.
	<code>ctrl + f</code>	Cerca all'interno del file.
	<code>ctrl + h</code>	Cerca e sostituisce all'interno del file.
☆	<code>ctrl + s</code>	Salva il file corrente.
	<code>ctrl + shift + p</code>	Apri la <i>Command Palette</i> di VS Code.

## 6.2 Le basi un po' meno elementari

Si può spostare il cursore in modo ben più rapido che un carattere alla volta.

	Tasti	Cosa fa
☆	<code>ctrl + freccia sx o dx</code>	Sposta il cursore di un <i>token</i> (in genere una parola, ma dipende dal contesto).
	<code>home</code> (inizio in italiano, più spesso  )	Sposta il cursore all'inizio della riga.
	<code>end</code> (fine in italiano)	Sposta il cursore alla fine della riga.
	<code>ctrl + shift + f</code>	Cerca all'interno della cartella/progetto/...
	<code>ctrl + shift + h</code>	Cerca e sostituisce all'interno della cartella/progetto/...
	<code>alt + freccia su/giù</code>	Sposta la riga corrente (o le righe selezionate) verso l'alto/basso.
☆	<code>ctrl + alt + freccia su/giù</code>	Copia la riga corrente (o le righe selezionate) verso l'alto/basso.

## 6.3 Editing multi-caret

Normalmente c'è un cursore, e ogni modifica fatta viene applicata dov'è quel *singolo* cursore.

Negli esempi che seguono, userò `|` per indicare un cursore, e coppie di `_` come delimitatori del testo selezionato.

```
Contenu|to dell'editor
```

Premendo A

```
ContenuA|to dell'editor
```

L'idea del multi-caret è di avere più di un cursore, per modificare più punti del testo allo stesso tempo. Questo è utile se abbiamo più punti del testo con uno stesso *pattern*.

	Tasti	Cosa fa
☆	ctrl + d	Aggiunge un cursore alla fine della prossima occorrenza del testo selezionato.
	esc	Ritorno alla modalità con singolo cursore.

Vediamo un esempio.

```
Prima |riga dell'editor
Seconda riga dell'editor
Terza riga dell'editor
```

Si comincia selezionando del testo.

```
Prima _riga_| dell'editor
Seconda riga dell'editor
Terza riga dell'editor
```

Usiamo ora `ctrl + d` per mettere un nuovo caret dopo la prossima occorrenza di "riga".

```
Prima _riga_| dell'editor
Seconda _riga_| dell'editor
Terza riga dell'editor
```

Abbiamo ora due caret e se facciamo una modifica verrà fatta in tutti e due i punti. Premendo per esempio `e`, andremo a sovrascrivere la parola "riga" in entrambi i punti.

```
Prima e| dell'editor
Seconda e| dell'editor
Terza riga dell'editor
```

Entrambi i cursori seguiranno indipendentemente anche gli altri comandi: movimento per caratteri, movimento per token, selezione, copia e incolla.

Per sfruttare questo, conviene scrivere codice secondo pattern in modo da facilitare questo tipo di modifiche. Per esempio, è utile avere cose che vorremmo poi modificare contemporaneamente su righe diverse, in modo da sfruttare `home` e `end` in modalità multi-cursore.

Vedremo in particolare come la sintesi di reti sincronizzate diventa molto più semplice se si sfrutta appieno l'editor.