

Lez. 4 – Modellazione in Verilog

Laboratorio di Architettura degli Elaboratori

Michele Lora

23 Novembre 2023

- **Comportamentale**: il comportamento del sistema hardware viene descritto specificando le azioni da comprendere, in maniera quasi algoritmica.
- **Strutturale**: il comportamento del sistema hardware viene descritto come aggregazione e interconnessione di componenti, i quali a loro volta specificheranno delle funzionalità in uno dei tre stili di modellazione.
- **Gate-level**: il comportamento del sistema hardware viene descritto come aggregazione di porte logiche di base, quali porte **and**, **or**, **xor**, **nand**, etc.

Tipi di dato

I segnali in Verilog possono avere solo i quattro valori corrispondenti a stati fisici di segnali hardware:

- '0': zero logico, oppure una condizione false;
- '1': uno logico, oppure una condizione vera;
- 'X': valore sconosciuto, oppure don't care;
- 'Z': alta impedenza del segnale.

È possibile esprimere valori numerici a più bit utilizzando i letterali numerici, la cui struttura base è: $D'B\text{valore}$

- D: dimensione del segnale (numero di bit) espresso come numero in base 10.
- B: base, può essere una delle seguenti: d: base 10, b: base 2, o: base 8, h: base 16 (esadecimale).
- Esempi: $32'd17 = (17)_{10}$, $32'hABCD = (43981)_{10}$

Gli elementi di memorizzazione e le connessioni tra componenti vengono rappresentati mediante due costrutti principali:

- **wire**: rappresenta la connessione tra elementi, il cui valore è guidato solamente dall'uscita di una rete logica (oppure, mediante **assegnamenti continui**).
- **reg**: modella il comportamento di un elemento di memorizzazione.

Per specificare **modelli a livello comportamentale** è possibile utilizzare variabili, simili a quelle dei linguaggi di programmazione:

```
integer i; // variabile intera a 32 bit.  
real r;    // variabile Reale (floating point).  
time t;    // variabile di tipo tempo.
```

Le variabili non possono essere dichiarate dentro a blocchi sequenziali, ma solo nel preambolo del modulo.

Il costrutto `module` è l'unità fondamentale di un modello Verilog.

Un modulo rappresenta un componente del sistema, caratterizzato da un'**interfaccia**, ossia la definizione delle sue **porte di ingresso** e **porte di uscita**.

Esempio

```
module complemento(input [3:0] in, output [3:0] out);  
begin  
  ...  
endmodule
```

Blocco sequenziale: costruito principale per la modellazione del comportamento di un sistema elettronico. È descritto da un insieme di istruzioni comprese tra le parole `begin` and `end`. Esistono due tipi di blocchi sequenziali:

- blocco `always`: blocco che viene eseguito ogniqualvolta si verifica un evento.
- blocco `initial`: blocco che descrive il comportamento del sistema a partire dall'istante iniziale della sua esecuzione.

Sintassi `always`

```
always @( evento )  
begin : nome  
    ...  
end
```

Sintassi `initial`

```
initial begin : nome  
    ...  
end
```

Il linguaggio Verilog mette a disposizione vari costrutti condizionali ed iterativi:

- costrutto condizionale `if`;
- costrutto condizionale `case`;
- costrutto di ciclo `while`;
- costrutto di ciclo `for`;
- blocco sequenziale `repeat`.

Costrutto condizionale **if**

```
if(condizione) begin
    [istruzioni sequenziali]
end
else if(condizione) begin
    [istruzioni sequenziali]
end else
    [istruzioni sequenziali]
end
```


Costrutto condizionale **case**

```
case(espressione)
  valore_1: begin
    [blocco di istruzioni sequenziali]
  end
  valore_2: begin
    [blocco di istruzioni sequenziali]
  end
  ...
  default: begin
    [blocco di istruzioni sequenziali]
  end
endcase
```

Costrutti iterativi, cicli **while**, **for** e **repeat**

Sintassi del ciclo **while**

```
while(condizione) begin
    [istruzioni sequenziali]
end
```

Sintassi del ciclo **for**

```
for(inizializzazione; condizione; update) begin
    [istruzioni sequenziali]
end
```

Sintassi del blocco **repeat**

```
repeat(numero) @ (evento) begin
    [istruzioni sequenziali]
end
```

Assegnamento continuo dei segnali

Segnali: tipicamente oggetti di tipo `wire`. Vengono continuamente assegnati mediante il costrutto:

```
assign segnale = [ritardo] valore o espressione;
```

Assegnamento dei registri

Due tipi diversi di assegnamento ai registri:

- **assegnamenti non bloccanti:**

```
registro <= valore o espressione;
```

- **assegnamenti bloccanti:**

```
registro = valore o espressione;
```

Solitamente, entrambi utilizzati dentro a blocchi sequenziali.

Esempio 1: Calcolo del complemento a 2

```
module complemento(input [3:0] in, output [3:0] out);

    integer i;
    reg [3:0] negato;

    always @(in)
    begin
        for(i = 0; i < 4; i = i + 1) begin
            negato[i] <= !in[i];
        end
    end

    assign out = negato + 1;
endmodule
```

Esempio 2: Sommatore a 4 bit

```
module somma4bit(  
    input [3:0] a,  
    input [3:0] b,  
    output reg [3:0] out);  
  
    always @(a or b) begin  
        out = a + b;  
    end  
endmodule
```

Stimolare il modello: testbench

Il **testbench** è il modulo che genera gli ingressi e legge le uscite del modello che si sta progettando. Dunque, il suo ruolo è quello di stimolare l'esecuzione del modello in esame.

Due alternative principali:

- **module** ad-hoc che istanzia il componente in esame;
- **script** di comandi che generano gli ingresso del componente in esame.

In questo corso, ci limitiamo a utilizzare la prima soluzione.

Comandi utili alla creazione di un testbench:

- `$dumpfile(nomefile.vcd)`: crea un file di tracce (waveform) in cui salvare l'andamento della simulazione;
- `$display(argomenti)`: stampa a video la stringa creata specificando gli argomenti (sintassi C-like).

Esempio: testbench per il complemento a 2.

```
'timescale 1ns / 1ps

module tb();
    reg [3:0] in;
    wire [3:0] out;

    integer i;

    complemento c(.in(in), .out(out));

    ...
endmodule
```

Esempio: testbench per il complemento a 2.

```
...  
    initial begin  
        $dumpfile("dump.vcd");  
        $dumpvars(1);  
        for(i = -7; i <= 7; i = i + 1) begin  
            in <= i;  
            #2;  
            $display("in: %d -> out: %d",  
                    $signed(in), $signed(out));  
        end  
    end  
endtask  
endmodule
```


Durante il corso utilizzeremo **EDAPlayground** un insieme di strumenti online: <https://www.edaplayground.com/>

EDAPlayground mette a disposizione un editor di testo, diversi simulatori (sia commerciali che open-source) e tool di sintesi. È necessario registrarsi, se la registrazione viene effettuata con l'indirizzo email **istituzionale** (e.g., @studenti.univr.it) permette di utilizzare i simulatori commerciali.

In laboratorio useremo **Mentor Questa**, un tool commerciale. Dunque, è consigliato di effettuare la registrazione a EDAPlayground utilizzando la propria email istituzionale. In alternativa, è possibile usare Icarus Verilog (strumento libero).

Configurazione di EDAPlayground

The screenshot displays the EDAPlayground web interface. The top navigation bar includes the EDA logo, 'playground' text, and buttons for 'Run', 'Save', and 'Copy'. A 'KnowHow WEBINARS' banner promotes free December webinars on SDC, Debugging SystemC & FPGA acceleration, with a 'REGISTER NOW' button. The left sidebar contains a 'Languages & Libraries' section with a 'Testbench + Design' dropdown set to 'SystemVerilog/Verilog'. Below this, 'UVM / OVM' is set to 'None', and 'Other Libraries' includes 'None', 'OVL 2.8.1', and 'SVUnit 2.11'. Checkboxes for 'Enable TL-Verilog', 'Enable Easier UVM', and 'Enable VUnit' are present. The 'Tools & Simulators' section shows 'Mentor Questa 2021.3' selected. The 'Compile Options' section has '-timescale 1ns/1ns'. The 'Run Options' section has '-voptargs=+acc=npr'. The 'Run Time' is set to '10 ms'. Checkboxes for 'Use run.do Tcl file', 'Use run.bash shell script', 'Open EPWave after run' (checked), 'Show output file after run', and 'Download files after run' are visible. The main area shows two code editors: 'SV/Verilog Testbench' and 'SV/Verilog Design'. The Testbench code defines a module 'tb' with a register 'in', a wire 'out', and a task 'display'. The Design code defines a module 'complemento' that takes 'in' and 'out' as inputs and produces 'negato' as output. The bottom panel shows a 'Log' section with a list of simulation steps and a timestamped message: '[2023-11-21 20:34:29 UTC] Opening EPWave... Done'.

EDA playground

Run Save Copy

KnowHow WEBINARS Attend FREE December webinars on SDC, Debugging SystemC & FPGA acceleration REGISTER NOW

Playgrounds Profile

Languages & Libraries

Testbench + Design

SystemVerilog/Verilog

UVM / OVM

None

Other Libraries

None

OVL 2.8.1

SVUnit 2.11

☐ Enable TL-Verilog

☐ Enable Easier UVM

☐ Enable VUnit

Tools & Simulators

Mentor Questa 2021.3

Compile Options

-timescale 1ns/1ns

Run Options

-voptargs=+acc=npr

Run Time: 10 ms

☐ Use run.do Tcl file

☐ Use run.bash shell script

☒ Open EPWave after run

☐ Show output file after run

☐ Download files after run

SV/Verilog Testbench

```
1 // Code your testbench here
2 // or browse Examples
3 timescale 1ns / 1ps
4
5 module tb();
6   reg [3:0] in;
7   wire [3:0] out;
8
9   integer i;
10
11   complemento c(.in(in), .out(out));
12
13   initial begin
14     $dumpfile("dump.vcd");
15     $dumpvars(1);
16     for(i = -7; i <= 7; i = i + 1) begin
17       in <= i;
18       #2;
19       display;
20     end
21   end
22
23   task display;
```

SV/Verilog Design

```
1 // Code your design here
2
3 module complemento(
4   input [3:0] in,
5   output [3:0] out
6 );
7
8   integer i;
9   reg [3:0] negato;
10
11   always @(in)
12   begin
13     for(i = 0; i < 4; i = i + 1) begin
14       negato[i] <= !in[i];
15     end
16   end
17
18   assign out = negato + 1;
19 endmodule
20
```

Log

Share

```
in: 0 -> out: 0
in: 1 -> out: -1
in: 2 -> out: -2
in: 3 -> out: -3
in: 4 -> out: -4
in: 5 -> out: -5
in: 6 -> out: -6
in: 7 -> out: -7
Finding VCD file...
./dump.vcd
[2023-11-21 20:34:29 UTC] Opening EPWave...
Done
```

Modellazione a Gate Level

Verilog permette di specificare modelli pi' vicini a quella che sar  l'implementazione finale del circuito. Il linguaggio mette a disposizione una libreria di porte logiche standard che implementano le funzioni Booleane corrispondenti. Le porte nella libreria sono: `and`, `or`, `xor`, `nand`, `nor`, `xnor`, e `not`.

Ogni porta logica richiede di specificare i segnali di uscita seguiti da quelli di ingresso. Quindi, ad esempio, $O = A \vee B$ sar  implementata specificando:

```
or(O, A, B);
```

gate e continuous assignments: la rappresentazione a gate level pu  essere espressa anche mediante assegnamenti continui, conoscendo l'operatore bit-wise corrispondente ad ogni porta:

<https://www.asic-world.com/verilog/operators1.html>

Esempio: sommatore ad 1 bit - gate level

```
module sommatore( input A, B, CIN, output O, COUT);  
  
    wire V, W, Z;  
  
    xor(V, A, B);  
    xor(O, V, CIN);  
    and(W, A, B);  
    and(Z, V, CIN);  
    or(COUT, W, Z);  
endmodule
```

Scrivere il testbench per il sommatore a quattro bit (esempio 2), descritto con lo stile comportamentale. Utilizzare i costrutti iterativi (`for` oppure `while`) per testare tutte le possibili combinazioni di ingresso.

Esercizio 2

Scrivere un componente che riceve in ingresso due numeri a 4 bit ed un segnale di controllo. Se il segnale di controllo vale '0' il componente effettuerà la somma dei due numeri in ingresso; altrimenti, eseguirà la sottrazione. Utilizzare lo stile comportamentale per descrivere il circuito.

Riscrivere il sommatore ad 1 bit descritto a gate level utilizzando gli assegnamenti continui.