Note esercitazioni Verilog

Raffaele Zippo

2 novembre 2020

Indice

1 Introduzione

Scopo di queste note è introdurre allo sviluppo, testing e debugging di reti logiche progettate in Verilog tramite un ambiente di sviluppo adatto all'uso didattico.
Presenteremo il software utilizzato, e come lo si usa per sviluppare (e correggere) semplici esempi.
In generale, questo documento è un *work in progress*.

1.1 Il linguaggio Verilog

Il Verilog è un *hardware description language* (HDL), ossia un linguaggio utilizzato per modellare e progettare sistemi elettronici digitali. Le caratteristiche e potenzialità del linguaggio si sono evolute in base alle necessità di progettazione di questi sistemi, che includono diversi livelli di astrazione, strumenti di test e debug, supporto alla sintetizzazione diretta su FPGA.
Allo stesso tempo, quindi, le varie forme di sintassi utilizzabili dipendono dal contesto e sono, in larga parte, lasciate al controllo dell'ingegnere.

In questo corso ci limitiamo a distinguere tre tipi di usi:

- scrittura di *sintesi* di reti logiche, ossia una descrizione univoca di come la rete logica è da implementare in hardware
- scrittura di *descrizioni*, ossia una descrizione del comportamento della rete logica. Questa è sintetizzabile in hardware, ma in modo non univoco: diversi approcci portano a diversi risultati.
- scrittura di *testbench*, cioè moduli che descrivono comportamenti non sintetizzabili in hardware il cui scopo è testare altri moduli in un ambiente simulativo. Per esempio, avremmo accesso a stampa su terminale e lettura da file, concetti privi di senso per una rete logica.

La flessibilità del linguaggio può portare confusione, è per questo importante tenere a mente che la discriminante è sempre lo scopo e uso del codice che si scrive.
Ciascuna delle forme sopra descritte ha il proprio scopo. Una *sintesi* è il prodotto ultimo del processo di progettazione, dato che descrive come andrà realizzata la rete desiderata in hardware. Una *descrizione* è un prodotto intermedio più facile da interpretare, modificare, correggere. Questo perché si focalizza sul comportamento ed evoluzione della rete, omettendo dettagli quali le connessioni tra porte logiche che realizzano tale comportamento, che possono essere discussi in fasi successive della progettazione. Una *testbench* ci permette di testare, verificare e correggere modelli in un'ambiente simulativo senza lasciare il calcolatore. Questo è in sostituzione a dispendiose, sia in risorse che tempo, prove su hardware.

2 L'ambiente di sviluppo

Gli ambienti di sviluppo HDL sono solitamente pacchetti software molto complessi, che mirano a supportare tutto il processo di sviluppo di nuovo hardware - dal semplice mockup e simulazione, all'analisi dei costi, alla sintetizzazione e prova su hardware FPGA.
Per i nostri scopi, ci basterà un ambiente minimo che si limita alla simulazione.

2.1 Icarus Verilog

Icarus Verilog[1] è una suite di programmi da linea di comando per la simulazione e sintesi di codice Verilog. Noi utilizzeremo, di questi:

- `iverilog`: si comporta come il compilatore `gcc`, compilando un eseguibile che ci permette di avviare la simulazione dell'hardware descritto. La sintassi tipica che useremo è:

  ```
  iverilog -o nome_output modulo_1.v ... modulo_n.v
  ```

- `vvp`: esegue la simulazione, a partire dal file prodotto da `iverilog`. La sintassi tipica che useremo è:

  ```
  vvp nome_output
  ```

2.2 GTKWave

Come vedremo, le stampe a terminale sono il modo più immediato per indicare l'andamento di una simulazione. Tuttavia, per verificare e correggere un modulo, è spesso più utile poterne studiare la completa evoluzione nel tempo.
Da una simulazione è possibile ottenere un file .vcd (Value Change Dump), che è possibile aprire con dei software appositi chiamati *waveform viewers*. Il waveform viewer che useremo è GTKWave[2].

[1] `http://iverilog.icarus.com/`
[2] `http://gtkwave.sourceforge.net/`

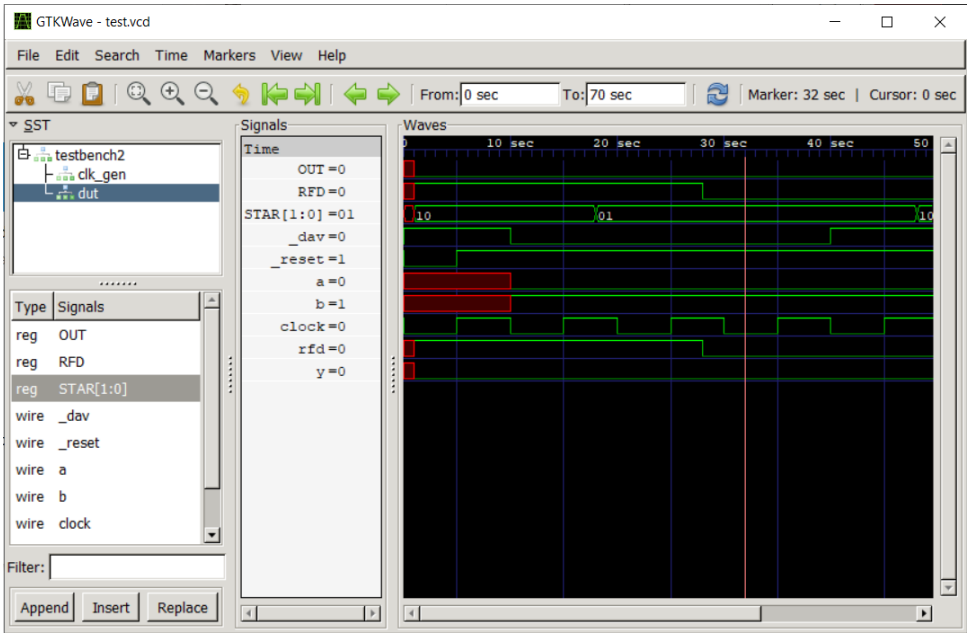


Figura 2.1: GTKWave

2.3 Visual Studio Code

Come per ogni linguaggio, i file di codice sono semplici file testuali modificabili con qualunque editor. La scelta dell'editor dipende dal supporto fornito, dalla comodità d'uso ma anche dall'abitudine e gusto personale.

In queste esercitazioni userò Visual Studio Code[3] con un'apposita estensione per evidenziare le keyword del Verilog[4].

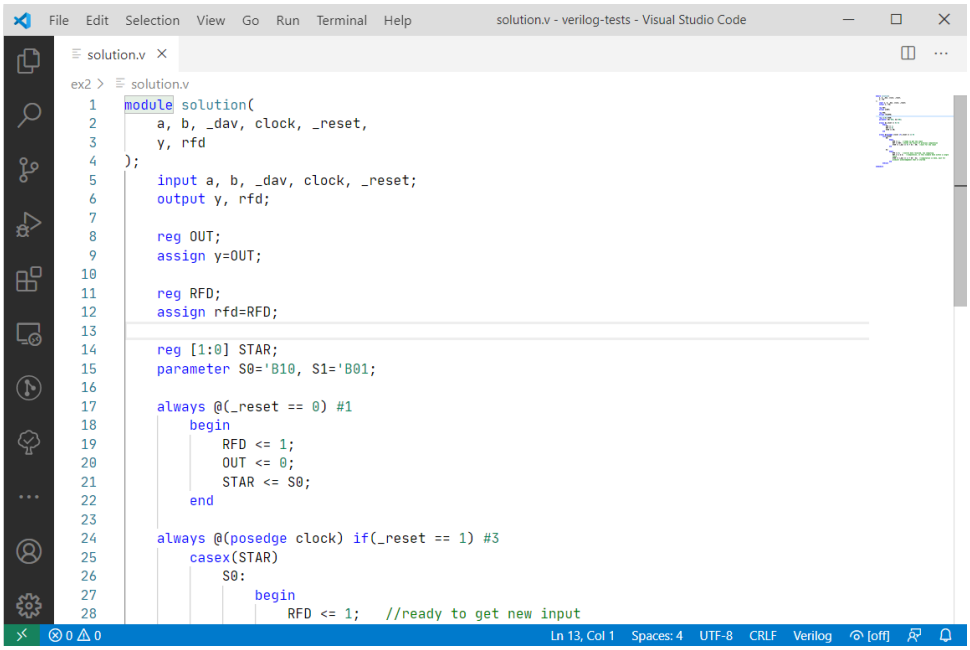


Figura 2.2: VS Code

[3] https://code.visualstudio.com/
[4] https://marketplace.visualstudio.com/items?itemName=mshr-h.VerilogHDL

2.4 Dove trovare il software

Tutto il software sopra indicato è gratis e pubblicamente disponibile, anche come codice sorgente. Per come ottenere ciascuno per il proprio sistema operativo, fare riferimento alle pagine web di ciascun progetto.

Per i sistemi da me testati:

- Windows 10: alla pagina `https://bleyer.org/icarus/` si trovano installer contenenti sia Icarus Verilog che GTKWave.
  Fare attenzione al percorso di installazione, che non deve contenere spazi, e che l'opzione "Add executable folder(s) to the user PATH" sia selezionata.
- Ubuntu 20.04: nei repository ufficiali apt sono presenti i pacchetti `iverilog` e `gtkwave`.

```
sudo apt install iverilog gtkwave
```

Per verificare la corretta installazione, da terminale si possono usare i comandi

```
iverilog -V
vvp -V
gtkwave --version
```

Il cui output dovrebbe contenere, per ciascuno, il nome del programma, la versione installata e la licenza con cui è distribuita.

3 Esempio con rete combinatoria

Per provare l'ambiente simulativo, partiamo dal caso più semplice di una rete combinatoria. Supponiamo di avere una rete definita come segue nel file `rete.v`, facendo finta, per l'esercizio, che sia abbastanza complessa da meritare una verifica via simulazione.

```
module rete_combinatoria(a, b, y);
    input a, b;
    output y;

    assign #1 y = a | b;
endmodule
```

Una *testbench* è del codice scritto per pilotare e testare dei moduli in un ambiente simulativo. Lo scopo è quindi quello di creare l'ambiente minimo (e con il minore sforzo) per poter verificare che i moduli sotto test si comportino correttamente.
Il codice di una testbench non è quindi necessariamente sintetizzabile, non essendo destinata ad hardware reale, e può quindi usare costrutti che, dal punto di vista hardware, non avrebbero senso.
Definiamo la nostra testbench nel file `testbenc.v` come segue:

```
module testbench();
    reg a, b;
    wire y;

    // instantiate device under test
    rete_combinatoria dut(.a(a), .b(b), .y(y));

    // apply inputs one at a time
    initial begin
        a = 0; b = 0; #10; // apply input, wait
        if(y !== 0) $display("0 0 failed."); // check result

        a = 0; b = 1; #10;
        if(y !== 0) $display("0 1 failed.");

        a = 1; b = 0; #10;
        if(y !== 0) $display("1 0 failed.");

        a = 1; b = 1; #10;
        if(y !== 1) $display("1 1 failed.");
    end
endmodule
```

Questa testbench non fa altro che verificare, uno dopo l'altro, tutti i possibili input della rete. Com'è prevedibile, questa operazione diventa infattibile con l'aumentare del numero degli input, sia per il numero di casi da considerare che per il tempo necessario a testarli tutti.
Parte della costruzione di buoni test è riuscire a trovare i casi più significativi, in modo da cogliere i possibili errori senza coprire l'intera tabella di verità.

Ciò diventa ancor più vero quando si trattano reti con memoria, dove non si dovrà più testare la risposta al singolo input ma la risposta ad una *sequenza* di input.

Riguardo i costrutti utilizzati, evidenziamo:

- L'uso dello statement `initial`: questo indica qualcosa da eseguire al tempo 0, concetto che ha senso solo in una simulazione.
- L'uso di assignment bloccanti ai registri (`=`): *In questo contesto* non ci interessa pensare ai registri come dispositivi elettronici che mutano "in parallelo", come facciamo nel codice sintetizzabile, ma piuttosto come semplici variabili a cui vengono assegnati valori instantaneamente. Questo rende più facile scrivere i casi di test, come faremmo in un linguaggio di programmazione come il C.
- L'uso di attese (`#N`) tra assegnazione degli input e check: Anche se nell'ambiente di simulazione possiamo assumere gli assegnamenti instantanei, ciò non significa che lo siano all'interno della rete sotto test. Questo è ancor più vero in reti sincronizzate, dove dovremmo attendere multipli del periodo di clock.
  Si noti quindi come cambia il significato dello stesso costrutto usato *in contesti diversi*: in una descrizione l'attesa modella un fenomeno fisico, in una testbench si usa per temporizzare i vari step del test.
- L'uso di chiamate come `$display`: Le parole chiave comincianti col `$` identificano funzioni dell'ambiente di simulazione. In particolare `$display` permette di stampare a terminale.

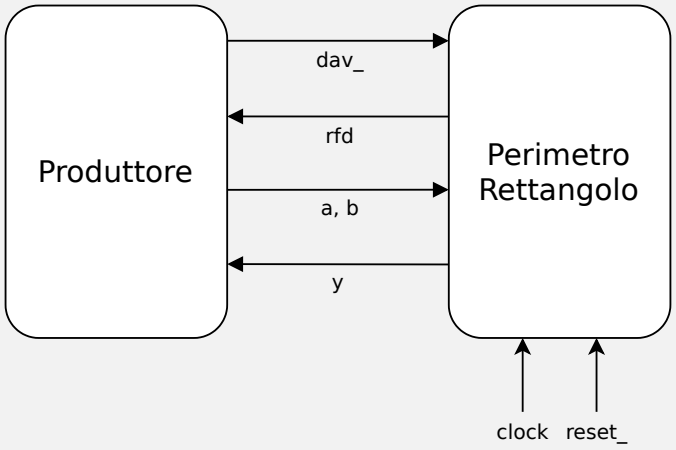
Una volta definiti i moduli e il testbench, è possibile avviare una simulazione via terminale:

```
C:\cartella\con\il\codice> iverilog -o rc .\testbench.v .\rete.v
C:\cartella\con\il\codice> vvp rc
0 1 failed.
1 0 failed.
```

Dall'output della simulazione, qualcosa non va. Lasciamo al lettore trovare cosa.

4 Esempio rete sincronizzata

Per questo esempio, consideriamo la seguente specifica:



Descrivere il circuito `PerimetroRettangolo` che, ciclicamente, preleva due byte dal Produttore, `base` e `altezza`, li elabora ed invia il risultato in uscita `perimetro`. Il prelievo viene fatto tramite handshake.
L'elaborazione viene fatta tramite una funzione `mia_rete(base,altezza)`, che interpreta i byte ricevuti come numeri naturali costituenti la base e l'altezza di un rettangolo e restituisce il perimetro del rettangolo.

4.1 Testbench

Dal punto di vista di testing e simulazione, abbiamo diversi aspetti da considerare:

- Vanno forniti segnali di reset e clock
- Il protocollo di handshake è seguito correttamente
- Il risultato della computazione è corretto

Clock

Il segnale di clock è solitamente prodotto da specifici circuiti oscillatori che sfruttano proprietà di materiali come il quarzo.
Di ciò non ci interessa però in questo contesto: ci basta un modulo simulativo che produca l'output desiderato all'interno della nostra testbench.
Definiamo quindi il modulo `clock_generator.v` come

```
module clock_generator(clock);
    output clock;
```

```
    parameter CLOCK_HALF_PERIOD = 5;

    reg CLOCK;
    assign clock = CLOCK;

    initial CLOCK <= 0;
    always #CLOCK_HALF_PERIOD CLOCK <= ~CLOCK;
endmodule
```

Notare che la definizione del parametro `CLOCK_HALF_PERIOD` ci permette di accedere a questo valore anche da altri moduli, in particolare dalla testbench.

Handshake e temporizzazione

Per verificare il protocollo di handshake possiamo utilizzare una sequenza di input e check come fatto nell'esempio precedente. C'è da chiedersi però come temporizzare le attese (operatore #) tra questi, e se in generale sia sensato farlo.
Infatti, i protocolli di handshake sono utilizzati tra due reti, che possono avere caratteristiche molto diverse, perché ciascuna *non* faccia assunzioni sui tempi di computazione dell'altra. Sarebbe quindi più accurato introdurre una rete *a*sincrona che controlli la validità del protocollo, senza assunzioni sul tempo che passa tra un evento e l'altro.
D'altra parte, dato che stiamo testando una rete da noi progettata, queste assunzioni hanno un senso se viste come parte dei requisiti: per esempio se è requisito che la rete sia in grado di rispondere con un risultato entro N periodi di clock.

Qui seguiremo quest'ultimo pensiero, ossia richiederemo un tempo massimo di risposta dalla rete `PerimetroRettangolo`.

Definiamo quindi la nostra testbench nel file `testbench.v` come segue:

```
module testbench();
    reg [7:0] base, altezza;
    wire [9:0] perimetro;

    reg dav_, reset_;
    wire rfd, clock;

    // instantiate clock generator
    clock_generator clk_gen(.clock(clock));

    // instantiate device under test
    PerimetroRettangolo dut(
        .base(base), .altezza(altezza), .dav_(dav_), .clock(clock), .reset_(reset_), //inputs
        .perimetro(perimetro), .rfd(rfd) //outputs
    );

    initial begin
        $dumpfile("test.vcd");
        $dumpvars;

        //reset phase
        reset_ = 0; dav_ = 1; #(clk_gen.CLOCK_HALF_PERIOD);
        reset_ = 1; #(clk_gen.CLOCK_HALF_PERIOD);

        if(rfd != 1)
            begin
                $display("rfd is 0 after reset");
                $finish;
```

```
            end

        // apply input, wait
        base = 'D20; altezza = 'D30; dav_ = 0;
        #(6*clk_gen.CLOCK_HALF_PERIOD);

        if(rfd != 0)
            begin
                $display("either (a) data not ACKed or (b) did not wait for computation ACK");
                $finish;
            end

        dav_ = 1;
        #(6*clk_gen.CLOCK_HALF_PERIOD); // time given to complete computation

        if(rfd != 1)
            begin
                $display("did not complete computation in the given time");
                $finish;
            end

        if(perimetro != 'D100)
            begin
                $display("computation result is wrong");
                $finish;
            end

        //if control reaches here
        $display("test passed");
        $finish;
    end
endmodule
```

4.2 Debugging con GTKWave

Consideriamo una soluzione così posta:

```
module PerimetroRettangolo(
    base, altezza, perimetro, dav_, rfd, clock, reset_
);
    input         clock,reset_;
    input         dav_;
    output        rfd;
    input   [7:0] base, altezza;
    output  [9:0] perimetro;

    reg           RFD;
    reg     [9:0] PERIMETRO;

    reg STAR;
    parameter S0=0,S1=1;

    assign   rfd=RFD;
    assign   perimetro=PERIMETRO;

    function[9:0] mia_rete;
        input [7:0] base, altezza;
```

```
        mia_rete = {({1'B0,base}+{1'B0,altezza}),1'B0};
    endfunction

    always @(reset_==0)
        begin
            STAR=S0;
        end

    always @(posedge clock) if (reset_==1) #3
        casex(STAR)
            S0: begin
                    RFD <= 1;
                    STAR <= (dav_=='B0) ? S1 : S0;
                end

            S1: begin
                    RFD <= 0;
                    PERIMETRO <= mia_rete(base,altezza);
                    STAR <= S0;
                end
        endcase

endmodule
```

Eseguendo la simulazione, l'output è il seguente:

```
VCD info: dumpfile test.vcd opened for output.
either (a) data not ACKed or (b) did not wait for computation ACK
```

La prima riga è una stampa del simulatore, di cui discuteremo fra poco. La seconda è invece un messaggio di errore da parte della testbench che abbiamo scritto. Da questo è facile risalire all'errore nella soluzione ma, di nuovo, supponiamo che ciò non sia vero a scopo di esempio.
In questa testbench abbiamo introdotto due nuove funzioni di sistema:

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

Questi comandi fanno sì che la simulazione generi un file di log (`test.vcd`) con la quale possiamo vedere l'evoluzione della rete nel tempo. Per farlo, chiamiamo da terminale

```
C:\cartella\con\il\codice> gtkwave test.vcd
```

Sulla sinistra (fig. 4.1) vedremo la gerarchia dei moduli all'interno della simulazione e, selezionandoli, i `wire` e `reg` all'interno di ciascuno.

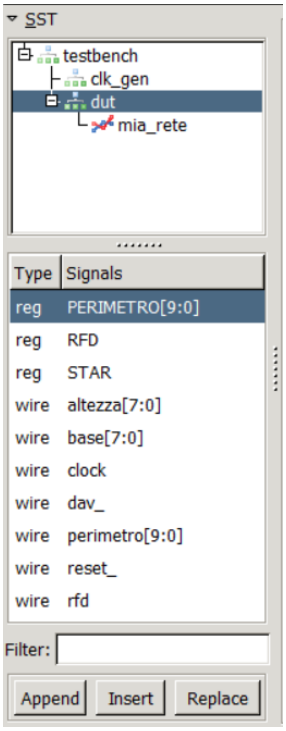


Figura 4.1: Vista dei moduli della simulazione

Dato il messaggio di errore, sospettiamo che il problema sia nell'handshake. Selezioniamo quindi i fili relativi (`dav_`, `rfd`, `clock`) e clickiamo Append per aggiungerli nella vista a destra.
Inizialmente, la vista proposta avrà dei marker verticali ad intervalli regolari scelti automaticamente dal programma. È invece più utile per noi avere questi marker ai posedge del clock, dato che la nostra rete si evolve rispondendo a questi. Per ottenere questo risultato:

- Selezionare il wire `clock` nella sezione Signals
- Nella barra dei menu, Search->Pattern search 1. Si aprirà una finestra
- Dove si legge il valore default "Don't Care" selezioniamo "Rising Edge"
- Clickare Mark e poi Exit per chiedere la finestra

La vista a questo punto dovrebbe essere come in fig. 4.2.

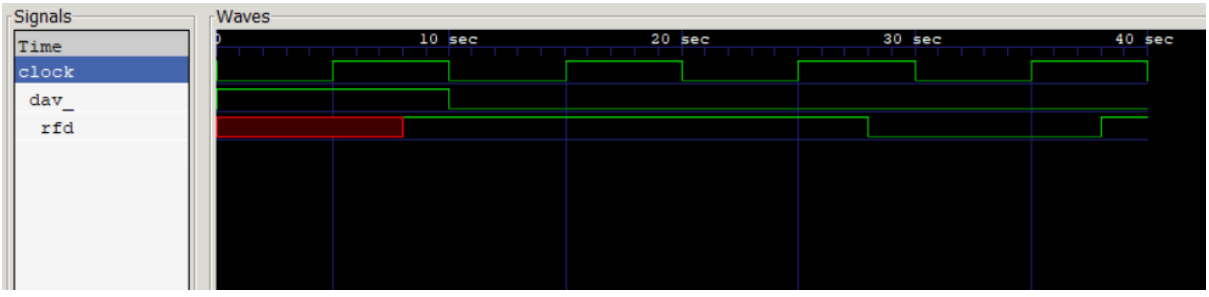


Figura 4.2: Vista delle waveform di interesse

Osservando queste, sarà ancor più facile identificare l'errore, che di nuovo lasciamo al lettore.
Questo approccio è in generale migliore per il *debugging*, ossia trovare la causa di un errore quando sappiamo già che c'è. L'uso di `if` e `$display`, invece, è più adatto per il *testing*, cioè automatizzare la verifica delle condizioni di errore più comuni e stabilire una prima confidenza che l'implementazione sia corretta.