Note esercitazioni Verilog

Raffaele Zippo

10 novembre 2022

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 Cosa serve sapere

Quanto è qui contenuto non è, come *teoria*, parte del programma d'esame. È tuttavia molto utile, in *pratica*, per (esercitarsi a) svolgerne le prove.

1.2 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 computer. 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, tramite un diagramma di temporizzazione.
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/`

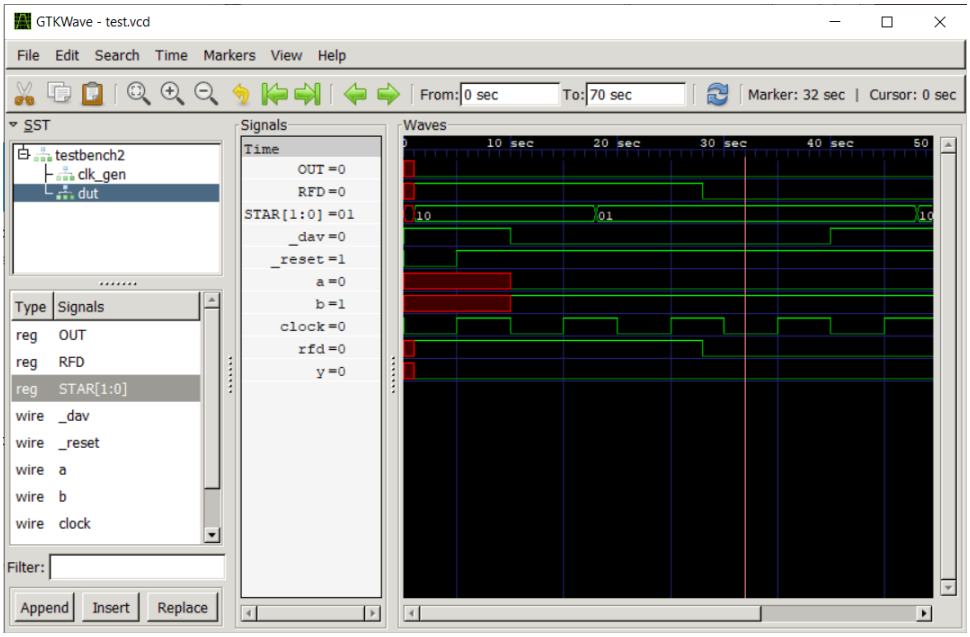


Figura 2.1: GTKWave

WaveTrace

Come alternativa a GTKWave, si segnala l'estensione WaveTrace[3]. Questa puo' risultare più immediata da usare in casi semplici, dato che si integra con l'editor di testo. Tuttavia, le feature sono più limitate di GTKWave e, nella licenza free, è limitato a 8 segnali.

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 useremo Visual Studio Code[4] con un'apposita estensione per evidenziare le keyword del Verilog[5].

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

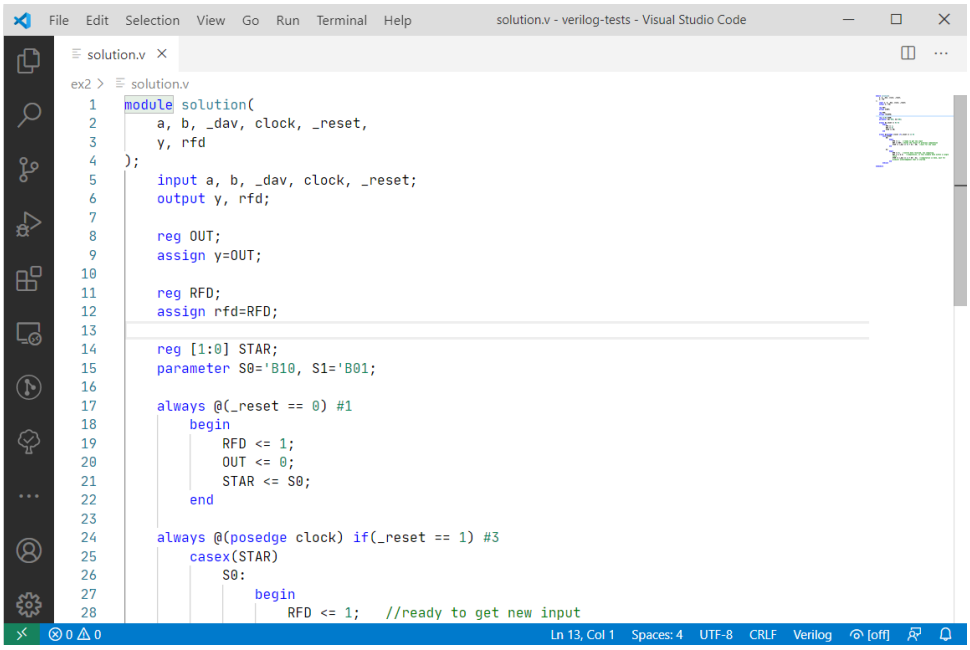


Figura 2.2: VS Code

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[6].
  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.

[6] Come segnalato da molti studenti, nella versione *v11-20210204* l'interfaccia di gtkwave è molto lenta. Si consiglia quindi la versione precedente, *v11-20201123*

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 `testbench.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 istantaneamente. 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 istantanei, 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.

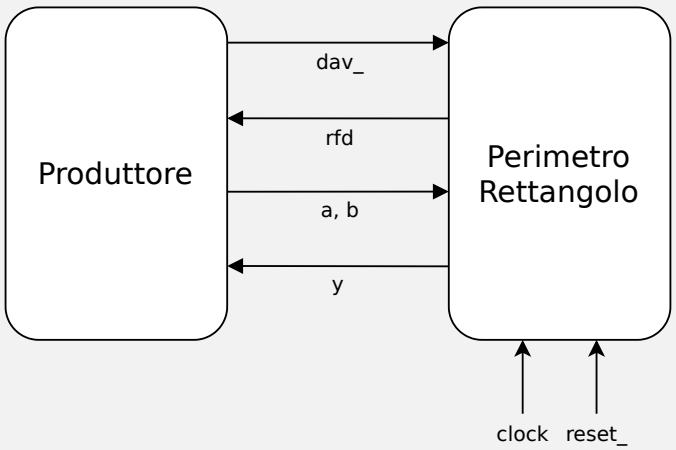
Una volta definiti i moduli e il testbench, è possibile avviare una simulazione via terminale:

```
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 avere un approccio *a*sincrono, 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.

In questo esempio seguiremo per semplicità 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("waveform.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 waveform.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("waveform.vcd");
        $dumpvars;
```

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

```
C:\cartella\con\il\codice> gtkwave waveform.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.

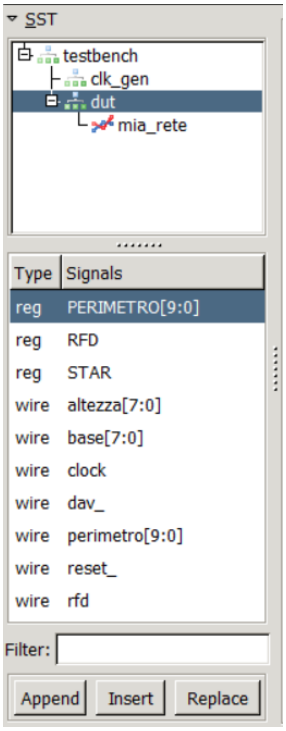


Figura 4.1: Vista dei moduli della simulazione

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.

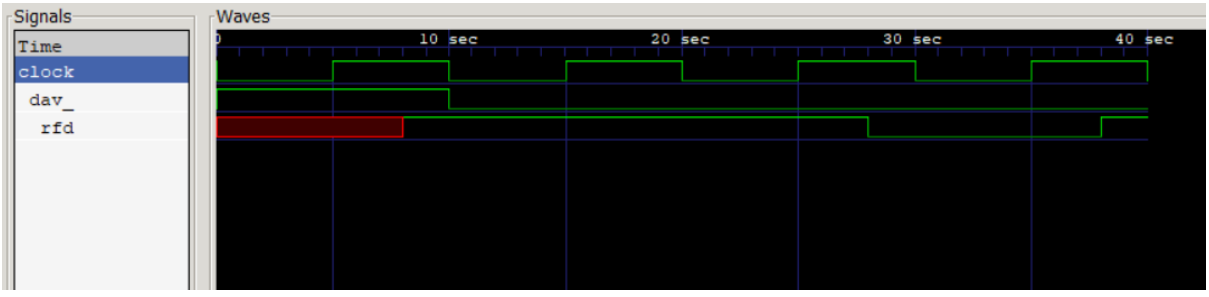


Figura 4.2: Vista delle waveform di interesse

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.

5 Sintassi Verilog

Questo capitolo è da intendersi come integrazione, non sostituzione, a quanto presentato nel libro di testo.
Il linguaggio Verilog ha lo scopo di supportare un'estesa famiglia di casi d'uso, e pertanto è estesa e piena di eccezioni la sua sintassi. In una trattazione didattica, c'è bisogno di limitarsi alle cose pertinenti a quanto si studia: in questo caso svolgere esercizi di descrizione e sintesi di reti abbastanza semplici, e relativa verifica via simulazione.
In ogni caso, tenere sempre a mente che Verilog è un linguaggio per *descrivere hardware*: ciò che si scrive ha senso solo se si ha in mente l'hardware che si sta descrivendo.

5.1 Dichiarazione di `module`

Con l'eccezione dei testbench, i `module` che vedremo corrispondono ad un *tipo* di rete logica: definisce quali sono gli input e gli output (fili), il comportamento (relazione tra input e output nel tempo).

La sintassi che useremo è del tipo

```
module MioModulo (
    input1, input2, ... , inputN,
    output1, output2, ... , outputN
)
    input input1;
    input [1:0] input2, input3;
    ...
    output output1;
    output [1:0] output2, output3;
    ...
endmodule
```

Nella dichiarazione, tra parentesi tonde, specifichiamo solo numero e nomi delle porte del modulo; la caratterizzazione di queste porte, cioè input/output e la dimensione in bit, viene invece data nel corpo del modulo.

Un'altra parte importante di un `module` sono i suoi `parameters`. Questo è un meccanismo per generalizzare i `module` e renderli adattabili a diversi contesti.

```
module AndParametrico (
    a, b,
    c
)
    parameter N = 2;    // numero bit
    parameter T = 1;    // tempo di attraversamento

    input [N-1:0] a, b;
    input [N-1:0] c;

    assign #T c = a & b;
endmodule
```

5.2 Corpo di un `module`

Il corpo di un `module` contiene i componenti e le relazioni logiche che costituiscono il comportamento del `module`. I componenti sono di tre tipi:

- `reg`
- `wire`
- Istanze di altri `module`

5.2.1 `reg`

La definizione base di `reg` in Verilog è "qualcosa che mantiene un valore". Non significa che è una variabile (non è un linguaggio di programmazione) ma piuttosto è una definizione abbastanza vaga da accomodare i diversi usi che se ne possono fare in diversi contesti.
Useremo `reg` in due modi:

- Nelle descrizioni/sintesi, come un *registro*
- Nelle testbench, come fonti di input per i moduli testati

5.2.2 `wire`

Un `wire` è invece un concetto più semplice: è qualcosa che collega, non mantiene un valore di per sé. Questo può portare confusione se si pensa alla sintassi di `assign`:

```
input a, b;
wire w;
assign w = a & b;
```

Quando si scrive questo non si deve pensare che al `wire w` si assegna un valore, come se fosse una variabile, ma piuttosto che gli input `a, b` e il filo `w` sono collegati tramite una porta logica AND.

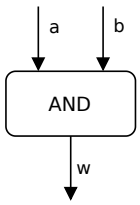


Figura 5.1: Schema corrispondente alla `assign` descritta.

La `assign` si usa quindi per specificare della logica tra diversi elementi, e si noti in particolare che il rhs[1] deve essere una *espressione combinatoria*.
Attenzione al livello di dettaglio di queste espressioni: è lecito, in Verilog, scrivere `a & b` così come `a * b`, ma mentre la prima specifica una porta logica semplice, e la consideriamo quindi *sintesi*, la seconda no. La moltiplicazione è infatti, come visto, un'operazione che va implementata con della appropriata logica che, in base al livello di dettaglio desiderato, può non aver senso omettere.
Definiremo, quando necessario, quali operatori sono ammessi.

[1] *right hand side*, la parte a destra dell'uguale. Opposto di *lhs*

5.2.3 Istanze di `module`

Come detto prima, un `module` è un tipo, una specifica (con vari livelli di dettaglio) di com'è fatta e si comporta una certa classe di reti logiche. Lo scopo principale di ciò è riutilizzarlo in altri in altri `module`. La sintassi che useremo, un po' più complessa di quanto presentato nel libro di testo, è del tipo:

```
module MioModulo(a, b, c)
    parameter N = 2;
    ...
endmodule

module AltroModulo(...)
    wire X, Y, Z;

    MioModulo #( .N(4) ) instanza_modulo (
        .a(X), .b(Y), .c(Z)
    );
    ...
endmodule
```

Ci sono due principali aggiunte.
La prima è l'uso di assegnamento per nome, ossia l'assegnare esplicitamente il filo `X` alla porta `a` del modulo e così via. Questo è in alternativa al classico assegnamento per posizione. È una funzionalità strettamente di linguaggio, che ci evita preoccupazioni ed errori nello scrivere il codice.

La seconda aggiunta è la sintassi per specificare i parametri, `#( ... )`.
Questa viene usata per *sovrascrivere* un parametro, un parametro non specificato viene lasciato al valore di default specificato nel modulo. Ci sarà utile per utilizzare moduli (che forniremo) definiti genericamente su `N` bit in diversi contesti, specificando di volta in volta quanto è `N`.

5.3 Differenza tra `function` e `module`

Una `function` in Verilog è molto più simile alle funzioni di linguaggi di programmazione, che ai `module` visti prima. Infatti lo scopo della sintassi è definire un *algoritmo*, non dell'*hardware*, che dati degli ingressi calcola e restituisce un output.

Mentre i `module` vengono *istanziati*, cioè si dice quanti ce ne sono, con quali parametri, e come sono connesse le varie porte; le `function` vengono *chiamate* come parte di una espressione. Inoltre, mentre i `module` sono componenti *fisiche* con tempi di attraversamento debitamente simulati, le `function` non hanno un corrispettivo fisico, e non consumano tempo. Il simulatore ne calcola semplicemente il risultato eseguendo il codice contenuto.

Ne consegue che le `function` vanno generalmente evitate (al di fuori di testbench) dato che il nostro scopo è descrivere hardware e non algoritmi.

Evidenziamo un caso in particolare in cui l'uso di `function` può essere conveniente, e cioè quando si vuole descrivere una rete combinatoria tramite la sua tabella di verità, piuttosto che la sua composizione. Infatti tramite una `function` si può utilizzare il blocco `casex` per associare semplicemente ingressi ad uscite.

```
module ReteCombinatoria(
    x1, x2,
    y1, y2
)
    input x1, x2;
    output y1, y2;
```

```
    assign #1 {y1, y2} = TabellaVerita(x1, x2);

    function [1:0] TabellaVerita
        inputx1, x2;
        casex({x1, x2})
            // [ingresso o default]: TabellaVerita = [uscita];
            2'b00: TabellaVerita = 2'b11;
            2'b01: TabellaVerita = 2'b00;
            2'b1?: TabellaVerita = 2'b01;
        endcase
    endfunction
endmodule
```

6 Scrivere una testbench

Premettiamo che, allo scopo dell'esame, ne' saper scrivere ne' saper leggere una testbench sono valutati.
Questo è uno strumento in più:

- Saper scrivere una testbench vi permette di esercitarvi meglio, potendo testare le reti che scrivete
- Saper leggere una testbench vi permette, in sede d'esame, di interpretare meglio i messaggi d'errore e correggere i difetti della propria soluzione[1]

Nel presentare la sintassi della testbench, affrontiamo prima le eventuali fonti di confusione che ne possono derivare.
Una testbench è comunque una descrizione di hardware, perché questo è quello che Verilog fa: i componenti utilizzati (reg, wire, module) potranno essere corrisposti a elementi hardware. La principale differenza però è che non pensiamo a descrivere un hardware reale, e cioè come si possa realizzare, ma solo ad un hardware che sia simulabile.
Questo ci permette, occasione che noi cogliamo, di descrivere semplicemente comportamenti che sono semplici dal punto di vista simulativo ma privi di alcun senso dal punto di vista realizzativo. È importante ricordarsi della distinzione, perché usare certi costrutti in una *descrizione* o *sintesi* è un errore grave.

Detto ciò, una *testbench* è un modulo il cui scopo è descrivere l'intero sistema, costituito principalmente dai dispositivi che intendiamo testare (*dut*, device under test), pilotarne gli input e monitorarne gli output.

Prima caratteristica è quella di essere priva di input/output. Ciò avrebbe poco senso in un vero sistema: il minimo necessario, per qualunque oggetto, è un pulsante di accensione. In una simulazione, questo è dato dall'atto stesso di avviare l'eseguibile.

```
module testbench()
    ...
endmodule
```

Dopodiché, vengono definiti i componenti che costituiscono il sistema. Questo include:

- I `reg`, che sono qui utilizzati solo come generatori *input* per i dispositivi del sistema
- I `wire`, che sono al solito utilizzati come collegamento. È in teoria possibile utilizzare anche qui `assign` complesse (e.g. `a * b`), ma è un approccio generalmente sconsigliato: chi testa la testbench?
- I vari `module`. Questi includono sia i *dut* che eventuali reti necessarie a pilotarle o interpretarne/verificarne i risultati. In particolare, per reti sincronizzate, dovremmo fornire un segnale di clock.

Questa parte è sintatticamente identica a quanto già visto.

[1] Avete anche la possibilità di fare modifiche alla testbench, se si pensa che aiuti il debug. Queste modifiche rimangono personali, non vengono consegnate e non cambiano i test di autocorrezione. Può non essere una buona idea però, per questioni di tempo.

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

    MioModulo dut (
        .a(a), .b(b),
        .c(c)
    )
endmodule
```

La parte più caratteristica della testbench è la descrizione dei comportamenti: avrà infatti una struttura simile a uno script, che descrive la sequenza di input da dare al sistema nel tempo e utilizza funzioni dell'ambiente di simulazione per fornire diagrammi di temporizzazione e messaggi di successo/errore.

La tipica struttura che utilizzeremo è la seguente:

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

        a = 0; b = 0;   // Applica input
        #10;            // Attesa per corretto pilotaggio delle reti combinatorie
        if(y !== 0)     // Controllo risultato
            $display("0 0 -> 0 failed.");
        else
            $display("0 0 -> 0 success.");

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

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

        a = 1; b = 1;
        #10;
        if(y !== 1)
            $display("1 1 -> 1 failed.");
        else
            $display("1 1 -> 1 success.");
    end
```

Lo statement `initial` introduce un blocco (che chiameremo impropriamente *script*, per semplicità) da eseguire al tempo 0 della simulazione.
Gli statement `$dumpfile` e `$dumpvars` creano un file di output da cui si può ricostruire il diagramma di temporizzazione usando GTKWave.
Il resto è una sequenza di assegnazioni dei `reg`, ossia gli input del sistema testato, attesa per i dovuti tempi di propagazione, verifica degli output. Gli statement `$display` sono utilizzati per stampare su terminale, che può quindi essere usato per capire rapidamente l'andamento del test.

Questa struttura è facilmente adattabile per testare altre reti combinatorie, dove la corrispondenza input-output è invariante nel tempo. Per testare reti sequenziali dovremo utilizzare una struttura più complessa, che testi *sequenze* di input.

6.1 Attendere eventi asincronamente

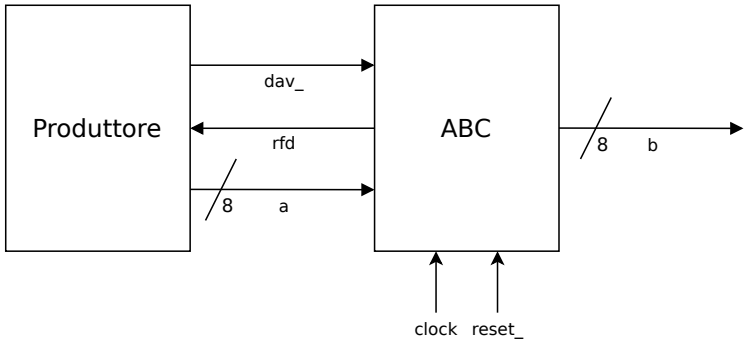


Figura 6.1: Schema del sistema

Si consideri il sistema in fig. 6.1. La rete `ABC` deve sostenere handshake con il `Produttore`, elaborando (non ci interessa come) il byte `a` per ottenere il byte `b`. Il nuovo valore di `b` è emesso dopo che l'handshake col `Produttore` è stato chiuso, ma non è data alcuna specifica sui tempi di risposta.

Il metodo visto finora (section 4.1) non ci basta a testare opportunamente questa rete. Dobbiamo verificare che gli handshake siano seguiti correttamente, indipendentemente dai tempi che intercorre tra gli eventi. Sarebbe quindi opportuno *aspettare* che gli output cambino, piuttosto che controllare ad istanti scelti arbitrariamente.

Utilizzeremo il costrutto `@()`, già visto insieme alla keyword `always`. Usata singolarmente in uno script, questo costrutto ci permette di *attendere* che avvenga una transizione.

```
for (i = 2; i < 48 ; i++ ) begin
    a = i;
    #(2*clk.HALF_PERIOD);
    dav_ = 0;
    @(negedge rfd);
    #(clk.HALF_PERIOD);
    a = ~i;
    #(2*clk.HALF_PERIOD);
    dav_ = 1;
    @(posedge rfd);
    @(b);
    if(b !== ~i)
        $display("Error!")
end
```

In questo esempio, la testbench, partendo dal caso in cui `dut` abbia `rfd` a `1`, ciclicamente:

- assegna al registro `a` il prossimo valore[2]
- mette `dav_` a `0`
- *attende* che `rfd` passi a `0`
- cambia (in modo più o meno casuale) `a`, così da forzare errori se `dut` ne legge il valore. Poi mette `dav_` a `1`

[2] Per casi più complessi, si può utilizzare una `function` che usi `i` come input per generare casi di test

- *attende* che `rfd` passi a `1`, tornando alla situazione iniziale
- *attende* che `b` cambi valore, e lo confronta quindi con il valore atteso[3]

Utilizziamo il costrutto in due modi:

- per vettori di più bit, per attendere generici cambiamenti: `@(b)`
- per singoli bit per attendere specifici *edge*: `@(posedge rfd)`, `@(negedge rfd)`

Si noti che non si ha alcuna transizione, e quindi non si sbloccano attese, se si fanno assegnazioni dove il valore rimane lo stesso. Ne consegue che vanno evitati casi di test successivi che hanno lo stesso valore atteso di output, o avere come primo caso di test uno il cui valore atteso sia `0`, dato che è un valore di inizializzazione abbastanza di moda.

Rimane un quesito aperto: se *attendiamo* che `dut` faccia una certa operazione, ma `dut` incorrettamente non la fa, come rileviamo l'errore? Per rispondere ci serve prima un altro costrutto, discusso nella successiva sezione.

6.2 Testare più comportamenti in parallelo

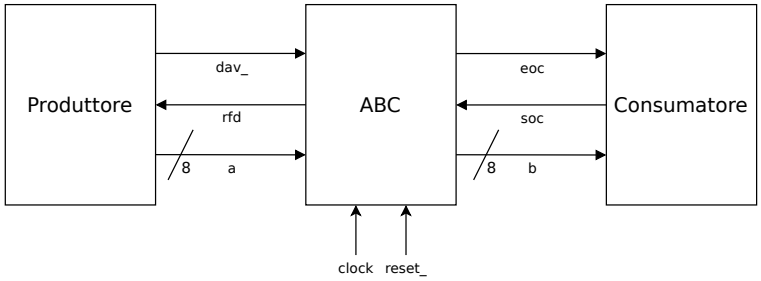


Figura 6.2: Schema del sistema

Si consideri il sistema in fig. 6.2. La rete `ABC` deve sostenere handshake sia con il `Produttore` che con il `Consumatore`, elaborando (non ci interessa come) il byte `a` per ottenere il byte `b`. Non è data alcuna specifica sui tempi di risposta o sull'ordine delle operazioni.

Anche in questo caso, i metodi visti finora non ci sono sufficienti. Dato che non ci è dato sapere in che ordine saranno compiuti gli handshake, diverse implementazioni che aprono/chiudono gli handshake in ordine diverso sarebbero comunque valide, e la testbench non dovrebbe aggiungere restrizioni oltre a quanto dato dalle specifiche.
Ma gli script che abbiamo visto fanno *una cosa alla volta*. Quel che ci serve è un modo di definire script da eseguire *contemporaneamente, in parallelo*.

Per questo, utilizzeremo il costrutto `fork ... join`.

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

    //fase di reset

    fork
        begin : producer
            reg [7:0] i;
            for (i = 2; i < 48 ; i++ ) begin
```

[3] Come per l'input: si può utilizzare una `function` per i casi più complessi. Quando si scrive test è bene descrivere l'output atteso tramite algoritmi, così che sia più chiaro e verificabile.

```
                //handshake Produttore, fornitura degli input come f(i)
            end
        end
        begin : consumer
            reg [7:0] i;
            for (i = 2; i < 48 ; i++ ) begin
                //handshake Consumatore, controllo degli output con valore atteso g(i)
            end
        end
    join

    $finish;
end
```

Il contenuto di `fork ... join` si comporta, rispetto allo script che lo circonda, come un normale costrutto visto finora, sia per esempio `for`. Con questo s'intende che lo script *initial* parte da `$dumpfile`, esegue le varie istruzioni nell'ordine in cui sono ed una alla volta, poi esegue il contenuto di `fork` e *solo quando ogni sua operazione è terminata* prosegue oltre, in questo caso con `$finish`.

A differenza di un `for`, però, questo costrutto esegue più operazioni contemporaneamente. I blocchi `begin...end` contenuti in `fork...join` sono appunto i blocchi di istruzioni da eseguire *parallelamente ed indipendentemente*: avremo quindi che le varie attese per segnali di handshake sono fatte indipendentemente, con ciascuna attesa che blocca solo l'esecuzione del proprio blocco del fork e non degli altri.

Ciascun blocco termina quando ha eseguito tutte le proprie istruzioni. In questo caso, questo è quando il rispettivo `for` ha terminato le proprie iterazioni. Il `fork...join` termina quando tutti i suoi blocchi `begin...end` sono terminati.

Notiamo inoltre che a questi blocchi sono stati dati *nomi*: `producer` e `consumer`. Questi vengono utilizzati per distinguere i due contesti di esecuzione, che ci aiuta per

- Poter definire `reg` utilizzati solo in quel contesto di esecuzione, che possono quindi avere lo stesso nome. In questo caso, sono i due `i`.
- Questi nomi sono poi utilizzati da GTKWave, cosa che ci è utile nel debugging

6.3 Rilevare errori nelle attese asincrone

Dopo aver introdotto il `fork...join`, possiamo rispondere alla domanda precedente: come posso rilevare l'errore dato dal fatto che l'evento che sto aspettando non avverrà mai?

Possiamo in realtà rispondere solo ad una versione ridotta di questa domanda: come rilevare che l'evento non è avvenuto *entro un ragionevole tempo*?

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

    //the following structure is used to wait for expected signals, and fail if too much time passes
    fork : f
        begin
            #100000;
            $display("Timeout - waiting for signal failed");
            disable f;
        end
```

```
        //actual tests start here
        begin
            //reset phase

            fork
                begin: producer
                    // producer-side behavior
                end

                begin: consumer
                    // consumer-side behavior
                end
            join

            #10;
            disable f;
        end
    join
    $finish;
end
```

Il codice sopra introduce un nome (`f`) per il `fork` più esterno, che è poi utilizzato in `disable f`. Questa istruzione è come un `break` per un `for`: fa terminare il `fork` e tutti i suoi blocchi ancora in esecuzione.
I due blocchi esterni si comportano quindi, rispettivamente, nel seguente modo:

1. Se è passato tanto tempo (stabilito arbitrariamente con `#100000`), stampa un errore e termina.
2. Se si è giunti al termine dello script, cioè non c'è stata nessuna attesa infinita, termina[4].

Si può notare che questa struttura ha due `fork` uno dentro l'altro: questo non pone nessun problema.

6.4 Tracce di errore

Uno strumento che può risultare utile è introdurre dei registri da un bit il cui unico scopo è, dalla traccia in GTKWave, evidenziare il punto in cui un dato errore ha avuto luogo[5]. Questo è facile da fare con i costrutti già noti:

```
reg error;

initial begin
    $dumpfile("waveform.vcd");
    $dumpvars;
    error = 0;
    ...
    if(...) begin
        error = 1;
        $display("Error!")
    end
end

always @(posedge error) begin
    #1;
    error = 0;
end
```

[4] L'attesa `#10` è solo per migliorare la leggibilità delle tracce durante debugging
[5] È possibile ottenere effetti simili usando costrutti specifici come `event`, che però hanno sintassi più particolari.

Se visualizziamo in GTKWave la traccia del registro `error` vedremo, nei punti in cui si sono rilevati degli errori, dei brevi picchi da a 0 a 1. Questi saranno facilmente visibili anche a basso zoom su lunghe tracce.