Verilog

per il corso di AESO

Marco Danelutto
A.A. 2022–23

Indice

Capitolo 1

Verilog

Questo capitolo intende dare una breve e sommaria descrizione del sottoinsieme di Verilog necessario per la realizzazione dei progetti di Architettura degli Elaboratori. Non vuole essere una trattazione completa del linguaggio e, in particolare, non tratta tutti gli aspetti legati all'utilizzo del linguaggio come strumento per la sintesi su FPGA.

Il sottoinsieme del linguaggio viene descritto in maniera informale e facendo abbondante uso di esempi. Per una introduzione più completa del linguaggio si rimanda alla letteratura riportata in bibliografia.

> Il linguaggio Verilog, come tutti gli altri linguaggi "RTL" (Register Transfer Languages), è in linguaggio per la descrizione dell'hardware. In quanto tale permette di definire componenti che calcolano funzioni, con o senza stato, che possono successivamente essere utilizzati come moduli di altri componenti. I linguaggi RTL possono essere utilizzati per la *simulazione* o per la *sintesi* di circuiti. Nel primo caso, il programma che descrive un certo circuito (componente) viene utilizzato per simularne il comportamento e per controllare dunque che calcoli ciò per cui era stato progettato. Nel secondo caso, il programma viene utilizzato per generare le specifiche da utilizzare per realizzare un circuito che implementi fisicamente quello descritto dal programma. Le specifiche posson consistere nello schema di realizzazione di un integrato VLSI o nel file di configurazione di una FPGA.

Nelle sezioni che seguono, introduciamo prima i tipi di dati trattati in Verilog, poi i moduli ed i comandi ed infine discutiamo brevemente l'utilizzo del linguaggio Verilog per la realizzazione di esercizi e progetti quali quelli assegnati nell'ambito del corso di Architettura degli Elaboratori.

1.1 Dati

In Verilog si possono utilizzare diversi tipi di dati: costanti (literal), wire, registri, vettori e interi (variabili generiche). Nel seguito descriviamo sommariamente tutte queste diverse categorie.

1.1.1 Costanti (literal)

I numeri si possono rappresentare specificando la base ed il numero di cifre, utilizzando la notazione

<n>'<b>xxxx

dove <n> in decimale rappresenta il numero di bit, <b> è un singolo carattere che rappresenta la base (`d` per decimale, `b` per binario, `o` per ottale e `h` per esadecimale). Quindi per rappresentare 9 in binario su 4 bit si utilizzerà la costante `4'b1001`, per indicare 127 in esadecimale si utilizzerà `8'hff`, per indicare 16 in ottale si utilizzerà `6'o20`.

1.1.2 Wire

I wire sono "fili", ovvero servono per realizzare i collegamenti utilizzati per connettere componenti implementati come `module` Verilog. Un wire si può dichiarare mediante la parola chiave `wire`:

```
wire x;
wire y,z;
wire [0:7]a;
wire [3:0]b;
```

Le prime due dichiarazioni introducono tre fili di nome `x` `y` e `z`, ognuno dei quali realizza un collegamento da 1 bit.

Le ultime due dichiarazioni introducono due gruppi di fili:

- uno da 7 bit (`a`) e
- uno da 4 bit (`b`)

Gli indici fra parentesi quadre permettono di identificare quanti sono i fili (d a 0 a 7, quindi 8 e da 3 a 0, quindi 4) e come si identificano i singoli bit:

- il bit 0 è il più significativo e il bit 7 il meno significativo nel primo caso ($[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$)
- il bit 3 è il più significativo e il bit 0 è il meno significativo nel secondo caso ($[b_3, b_2, b_1, b_0]$).

Si possono riferire singoli wire di un gruppo utilizzando le parentesi quadre:

- `a[0] è il bit più significativo del wire da 8 bit di nome \verba1`
- `b[1:0]` sono i due bit meno significativi del wire `b` (b_1, b_0)

1.1.3 Registri

I registri si possono dichiarare utilizzando la parola chiave `reg` e convenzioni come quelle utilizzate per i wire per definirne la dimensione:

```
reg uno, due;
reg [0:7]unbyte;
reg [31:0]unaword;
```

`uno` e `due` sono due registri da un bit. `unbyte` è un registro da 8 bit (`unbyte[0]` è il bit più significativo). `unaword` è un registro da 32 bit (`unaword[31]` è il bit più significativo).

1.1.4 Array

Si possono definire array utilizzando sempre le parentesi quadre, poste *dopo* l'identificatore nella dichiarazione:

```
reg v[16];
reg [7:0]t[256];
```

`v` è un vettore di registri da 1 bit da 16 posizioni, `t` è un vettore di 256 registri da 8 bit ciascuno. Per assegnare un vlaore alla posizione $i-$esima del vettore `t` (l-value) o per leggerne il valore (r-value) utilizziamo la sintassi con l'indice fra parentesi quadre, come in C:

```
t[i] = ...;          ... = ... t[i] ... ;
```

1.1.5 Integers

Le variabili generiche vengono introdotte con la parola chiave `integer`. Sono implicitamente di tipo `reg` e sono interpretate come interi con segno (positivi e negativi)[1].

[1] i registri sono invece considerati sempre unsigned

1.2 Operatori

In Verilog si possono utilizzare molti degli operatori solitamente disponibili nei normali linguaggi di programmazione:

- Operatori aritmetici: `+ - * / %` (modulo)
- Operatori relazionali: `< > <= >= == !=`
- Operatori bit a bit: `~ & | ^` (bitwise not, and, or e xor, rispettivamente)
- Operatori logici: `! && ||` (not, or e and)
- Operatori di shift: `<< >>` (shift a sinistra e a destra)
- Operatore di concatenazione `{ , }` (concatena nell'ordine. `{a,b,c}` resituisce la concatenazione dei bit di a b e c, nell'ordine)
- Operatore di replicazione: `{n{item}}` (ripete n volte item come in una concatenazione. `{2{a}}` equivale a `{a,a}`.)
- Operatore condizionale: `( ? : )` (`(x<=y ? 1'b1 : 1'b0)` restituisce il bit 1 se e è minore o uguale a y, altrimenti restituisce il bit 0)

La precedenza fra gli operatori è definita come nella tabella seguente (precedenza decrescente dall'alto al basso):

Operatore	**Nome**
`[ ]`	selettore di bit o di parti
`( )`	parentesi
`! ~`	NOT logico e bit a bit
`& | ~& ~| ^ ~^`	operatori "reduce" (and, or, nand, nor, xor, nxor)
`+ -`	segno unario
`{ }`	concatenazione (`{2'B01,2'B10}=4'B0110`)
`{{ }}`	replicazione (`{2{2'B01}}=4'B0101`)
`* / %`	moltiplicazione, divisione, modulo
`+ -`	addizione, sottrazione
`<< >>`	shift destro e sinistro (`X<<2` moltiplica X per 4)
`< <= > >=`	confronti, registri e wire interpretati come numeriinteri positivi
`== !=`	uguaglianza/disuguaglianza logica
`&`	and bit a bit di una parola
`^ ~^`	xor nxor bit a bit
`|`	or bit a bit
`&&`	and logico (0 è false, il resto è true)
`||`	or logico
`?:`	condizionale (`X==Y ? 1'B1 : 1'B0`)

1.3 Componenti (moduli)

I moduli in Verilog rappresentano l'astrazione di una componente. Un modulo può rappresentare una funzione (rete logica) o una funzione con stato (rete sequenziale). I moduli possono essere ricorsivamente definiti come composizione di altri moduli.

In Verilog si possono definire componenti attraverso il costrutto `module`. Un modulo è molto simile ad una dichiarazione di procedura:

- ha un nome,

- una lista di parametri formali (di ingresso o di uscita[2])

- un corpo, che definisce come i parametri di uscita vengono calcolati a partire dai parametri in ingresso.

Tuttavia, i moduli non vengono "chiamati" bensì *instanziati* utilizzando opportuni parametri attuali.

La sintassi per definire un modulo è la seguente:

```
module <nome>(<lista parametri formali>);
  <corpo>
endmodule
```

I parametri formali possono essere dichiarati come `input` o `output`, sia nella testa della dichiarazione che fra la testa e il corpo (come nel vecchio C).

Supponiamo di avere dichiarato i moduli:

```
module Uno(output z, input x, input y);
  ...
endmodule

module Due(z, x, y);
  input x,y;
  output z;

  ...
endmodule
```

Il primo modulo usa la dichiarazione del tipo dei parametri direttamente nelle parentesi tonde. Il secondo dichiara i nomi dei parametri nelle parentesi e poi il tipo (in ingresso o in uscita) è dichiarato dopo l'intestazione. I parametri in uscita possono essere di tipo `wire` (questo è il titpo di default, non occorre specificarlo) oppure register. In questo secondo caso occorre specificare la parola chiave `reg` nella dichiarazione, sia essa fra le parentesi o successiva alla intestazione del modulo.

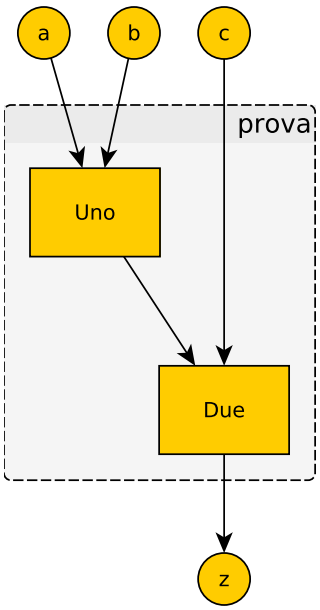
Possiamo utilizzare i nostri due moduli così definiti all'interno di in un altro modulo, instanziandoli. Per esempio:

```
module prova(zz, a,b,c);
  output zz;
  input a,b,c;

  wire da_uno_a_due_x;

  Uno istanza_di_uno(da_uno_a_due_x,a,b);
  Due istanza_di_due(zz, da_uno_a_due_x, c);

endmodule
```

In questo caso, vengono create due istanze (uno per ciascun tipo di modulo) e si usa un filo per collegare l'uscita dell'istanza del modulo di tipo Uno al primo ingresso del modulo di tipo Due. Gli altri ingressi (i due del primo modulo e l'altro ingresso del secondo) arrivano dagli ingressi del nostro modulo di tipo prova.

Sostanzialmente realizziamo uno schema tipo:

[2] si possono definire anche parametri in ingresso e uscita ma non servono per gli scopi del nostro corso



Una discussione più ampia sulla dichiarazione e sull'utilizzo dei moduli è ripor tata in Sez. ??.

1.3.1 "Parametri" dei moduli

I moduli possono avere "parametri" definiti per default e che possono essere variati in fase di istanziazione. I parametri corrispondono alle `#define` del linguaggio C, con alcune piccole ma sostanziali differenze. I parametri si dichiarano subito dopo la testa del modulo come

```
parameter <nome> = <valore>;
```

I parametri possono essere utilizzati nel modulo, sia per definire i parametri formali dichiarati nella testa del modulo che nel corpo. Ad esempio, possiamo definire un modulo con parametri di input e output di dimensione parametrica che modella un commutatore (due ingressi da N bit, un ingresso di controllo da 1 bit, una uscita da N bit) specificando una cosa tipo:

```
module commutatore_nbit(z, x, y, alpha);

  parameter N = 32;

  output [N-1:0]z;
  input [N-1:0]x;
  input [N-1:0]y;
  input alpha;

  assign
    z = ((~alpha) ? x : y);

endmodule
```

I parametri possono essere ridefiniti in fase di istanziazione del modulo, semplicemente anteponendo il nuovo valore del parametro preceduto da un # e fra parentesi tonde al nome dell'istanza del modulo, in fase di instaziazione. Dunque potremmo istanziare il modulo `comm`

```
  commutatore_nbit #(16) mio_commutatore(...);
```

Questa riga di codice crea un'istanza di un commutatore a due ingressi da 16 bit, anche se la dichiarazione del modulo `commutatore_nbit` definisce il parametro $N = 32$.

In caso si usino più parametri, i loro valori possono essere specificati in fase di istanziazione, nell'ordine in cui sono stati dichiarati nel modulo, fra le parentesi tonde precedute dalla gratella.

1.3.2 Moduli per il test di componenti

Mediante i moduli possiamo definire componenti e "moduli di prova" (*testbench*) ovvero moduli che servono per testare componenti o assemblaggi di componenti.

Un modulo di prova è un modulo *senza parametri formali*. All'interno del modulo di prova possiamo utilizzare alcune istruzioni (vedi sez. 1.5) che serviranno a guidare la simulazione dei componenti e a registrarne gli effetti.

La tipica struttura di un modulo di prova è la seguente:

```
module <nome>();

 // dichiarazioni di wire per ognuno degli output del componente testato
 ...
 // dichiarazioni di register per ognuno degli input del componente testato
 ...

 // istanziazione del componente
 ...
 // programma di prova : assegna valori agli input
 // (valori diversi in  tempi diversi)
 ..
endmodule
```

1.4 Blocchi e comandi

All'interno di un modulo si possono usare sostanzialmente diversi tipi di comandi, nonchè blocchi di comandi delimitati da un `begin end`. In un modulo si possono anche utilizzare, al livello più esterno, blocchi di comandi delimitati da `begin end` e qualificati dalle keyword:

- **initial**
  i comandi del blocco vengono eseguiti solo alla partenza della simulazione. Ad esempio, qualora nella dichiarazione di un modulo compaia il blocco di comandi:

  ```
  intial
    begin
      r0 = 0;
      r1 = 1;
    end;
  ```

  i comandi fra il `begin` e l'`end` vengono eseguiti all'atto dell'istanziazione del modulo e inizializzano una volta per tutte il registro `r0` a 0 e il registro `r1` a 1.

- **always**
  i comandi del blocco vengono eseguiti continuamente, come se il blocco fosse il corpo di un `(while(true))`. Ad esempio, un blocco tipo:

  ```
  always
    begin
      #1 clock = ~clock;
    end
  ```

  in un modulo dove abbiamo anche specificato

  ```
  reg clock;

  initial
    begin
      clock = 0;
    end
  ```

  farà sì che il valore del registro `clock` oscilli fra 0 e 1, mantenendo lo stato 0 (1) per una unità di tempo. Qualora volessimo mantenere alto il livello del clock per una unità di tempo e basso per 17 unità di tempo (per esempio) potremmo utilizzare un blocco always.

```
always
  begin
    #17 clock = 1;
    #1 clock = 0;
  end
```

Alla parola chiave `always` si possono associare dei modificatori. Dopo la always si può introdurre una `@` seguita da una lista di variabili fra parentesi tonde (detta *sensitivity list*), col significato: esegui il blocco always ogni volta che una delle variabili della lista cambia valore:

```
always @ (x or y)
  begin
    ...
  end
```

esegue il blocco ogni volta che cambia il valore di x o quello di y. Dalla versione 2001 del verilog al posto dell'`or` si può utilizzare la virgola:

```
always @(x, y)
```

ha la stessa semantica della notazione precedente con gli `or`. Possiamo anche utilizzare la `@` per introdurre una cosa tipo

```
  always @ (negedge x)
```

oppure

```
  always @ (posedge x)
```

che significano, rispettivamente, esegui il blocco che segue ogni qualvolta x passa da 1 a 0 o da 0 a 1.

Comandi

Dentro ad un blocco si possono utilizzare diversi tipi di comandi:

- assegnamento
  esistono diversi tipi di assegnamento: bloccante e non bloccante. L'assegnamento bloccante (simbolo =) termina prima che venga eseguito il prossimo statement (magari di assegnamento). Dunque

```
x = 1;
y = x;
```

assegna ad x il valore 1 e **successivamente** assegna ad y il valore di x, quindi 1. Nell'assegnamento non bloccante (simbolo <=) gli assegnamenti avvengono tutti allo stesso istante, ovvero la lettura delle variabili delle parti destre e il calcolo delle espressioni da assegnare avvengono contemporaneamente. Dunque

```
x <= y;
y <= x;
```

realizza uno scambio fra i valori di x e y. Esiste anche un terzo tipo di assegnamento, l'assegnamento *continuo* (simbolo `assign <parte-sn> = <expr-ds>`) la cui semantic a invece è: *assegna in continuazione il risultato della parte destra alla parte sinistra. Se varia un valore utilizzato nella parte destra, rivalutala e riassegnala alla parte sinistra*. Dunque in questo caso

```
assign x = y + z;
```

assegna a x il valore della somma di y e z. Ogni volta che y o z variano, la loro somma viene nuovamente assegnata a x. In tutti i casi, si possono (solo ai fini della simulazione) introdurre dei ritardi, in unità di tempo, mediante la sintassi `#<ritardo>` utilizzata prima dell'assegnamento o durante l'assegnamento.

```
  #10 x = y + z;
```

aspetta 10 unità di tempo e quindi assegna la somma di y e z a x.

```
  x = #10 y + z;
```

calcola y+z subito ma effettua l'assegnamento ad x della somma solo dopo 10 unità di tempo.

Ai fini del nostro corso, non specificheremo come indicare l'unità di misura per il tempo e assumeremo che una unità corrisponda ad un t_p secondo la terminologia del libro di testo.

Si noti infine che il left hand side di un assegnamento *deve* essere un registro (non si possono fare assegnamenti a wire!), ma che l'assegnamento continuo può essere utilizzato per pilotare i `wire`.

- cicli
  si possono eseguire comandi in un ciclo utilizzando il for e il while, con sintassi praticamente identica a quella del C:
  - ciclo for:

```
      for(i=0; i<=N; i=i+2)
        begin
        end

```

    con limitazioni sul tipo di incremento (solo i=i +/- valore) e sul tipo di test (solo < <= > >=)
  - ciclo while:

```
      while(cond)
        begin
        end

```

    con limitazione sul fatto che il corpo deve contenere una temporizzazione;

  da notare che il `for` è sintetizzabile, ovvero può essere utilizzato nella definizione di un componente come mezzo per includere nel componente stesso un numero definito di altri componenti, mentre il `while` non è sintetizzabile, ma può essere utilizzato nei *testbench*.

- condizionali:

```
    if(cond)
      <ramo-then>
    else
      <ramo-else>

```

  con il ramo else facoltativo

- scelta multipla:

```
    case(espressione)
      valore1: begin ... end
      valore2: begin ... end
      ...
      default: begin ... end
    endcase

```

  In questo caso il caso `default` è facoltativo.

Un caso particolare di comando che si può utilizzare per la definizione di un modulo è il blocco `generate`. Un blocco generate puà essere utilizzato per generare un certo numero di istanze di componenti. L'esempio che segue fa vedere come possiamo utilizzare il `generate` per instanziare una serie di multiplexer da due ingressi di un singolo bit per realizzare un multiplexer da due ingressi da N bit ciascuno.

```
module commutatore_nbit_generative(z,x,y,alpha);

  parameter N = 32;

  output [N-1:0]z;
  input  [N-1:0]x;
  input  [N-1:0]y;
```

```
8   input  alpha;
9
10  genvar i;
11
12  generate
13    for(i=0; i<N; i=i+1)
14    begin
15      mux2 t(z[i],x[i],y[i],alpha);
16    end
17  endgenerate
18
19 endmodule
```

Alla linea 10 dichiariamo la variabile da utilizzare per la generazione (e per il relativo ciclo for). Alla riga 12 iniziamo il blocco generative. Il for all'interno del blocco crea N istanze del modulo `mux2`. La scelta dei parametri attuali delle istanze in base al valore della variabile `genvar i` permette di realizzare i collegamenti come desiderato: l'istanza i del multiplexer riceve in ingresso i bit $i-$esimi di `x` e `y`, il segnale di controllo `alpha` e produce in uscita il bit $i-$esimo di `z`.

L'implementazione di un modulo secondo il modello "behavioural" è quella che si ottiene utilizzando i comando behavioural di tipo assegnamento, `if-then-else` e `case` , in blocchi `always`, `generate` o `assign`.

Si noti che un ciclo "while(true)" di fatto corrisponde a livello di modulo ad un:

```
1  always @(*)
2    begin
3      ...
4    end
```

1.5 Direttive

I comandi che possiamo utilizzare per la simulazione sono tutte direttive che iniziano col segno dollaro `$`. Le direttive permettono di salvare la traccia di esecuzione di una simulazione, di terminare la simulazione stessa o di stampare valore delle variabili utilizzate sul terminale. Fra i comandi che possiamo utilizzare per controllare la simulazione, citiamo:

- `$dumpfile(‘‘nomefile’’);`
  permette di eseguire un dump di tutte le variabili nel programmma, in modo da poter analizzare il risultato della simulazione con un programma tipo gtkwave successivamente

- `$dumpvars;`
  permette di fare un dump di tutti i valori delle variabili del modulo nel tempo all'interno del file specificato con la `$dumpfile`. Si possono passare parametri alla `$dumpvars`. In particolare, `$dumpvars(k,top)` eseguirà il dump di tutte le variabili del modulo top e dei moduli annidati fino a a k livelli (se $k = 0$ di tutti i moduli).

- `$time;`
  restituisce il valore del tempo corrente (in unità di tempo)

- `$display(formato, lista variabili);`
  mostra il contenuto delle variabili nella lista, secondo il formato (opzionale). La stringa di formato (simile a quella della printf del C) utilizza `%d`, `%b`, `%t` e `%h` per visualizzare valor i in decimale, binario, di tempo e esadecimale, rispettivamente. Dunque `$display(‘‘X vale %b’’, x)` fa vedere il valore di `x` in binario.

- `$monitor(formato,lista variabili);`
  funziona come la display, ma esegue la stampa ogni volta che le variabili cambiano valore

- `$finish;`
  termina la simulazione. Di solito il testbench (modulo senza parametri che istanzia un componente e lo testa) conclude il blo cco `initial` che contiene il main della simulazione con uno statement `#NN $finish;` ovvero attende un certo intervallo di tempo e poi termina.

Ora che sappiamo anche quali sono le direttive utilizzabili in un modulo, possiamo vedere un esempio completo di modulo di test. Cosideriamo l'esempio di clock descritto nella sez. 1.4. Proviamo a testarne il funzionamento con un modulo di prova:

```
module clock();   // definiamo un modulo che testa un segnale clock

reg clock;        // registro da 1 bit, che utilizziamo per
                  // contenere il valore del clock nel tempo

initial           // alla partenza del circuito
begin
  clock = 0;      // il valore del clock e' 0
end

always            // while (true)  ... cioe' per sempre
begin
  #10 clock = 1;  // aspetta 10 unita' di tempo poi metti clock a 1
  #1  clock = 0;  // poi aspettane 1 e metti clock a 0
end               // e ricomincia il while(true)

initial           // questo e' il main del programma di test
begin             // segnala quando cambia clock
  $monitor("%t %d",$time,clock);

  #30             // aspetta 30 unita' di tempo
  $finish;        // e termina la simulazione
end
endmodule
%end{verbatim}
```

Se abbiamo creato il programma in un file "clock.vl" e compiliamo il programma con un comando:

```
iverilog clock.vl -o clock
```

successivamente possiamo eseguire il modulo di test lanciando da shell:

```
./clock
```

ottenendo l'output:

```
marcod@sony-duo-11:~/Verilog/Libro/Dispa$ iverilog clock.vl -o clock
marcod@sony-duo-11:~/Verilog/Libro/Dispa$ ./clock
                   0 0
                  10 1
                  11 0
                  21 1
                  22 0
marcod@sony-duo-11:~/Verilog/Libro/Dispa$
```

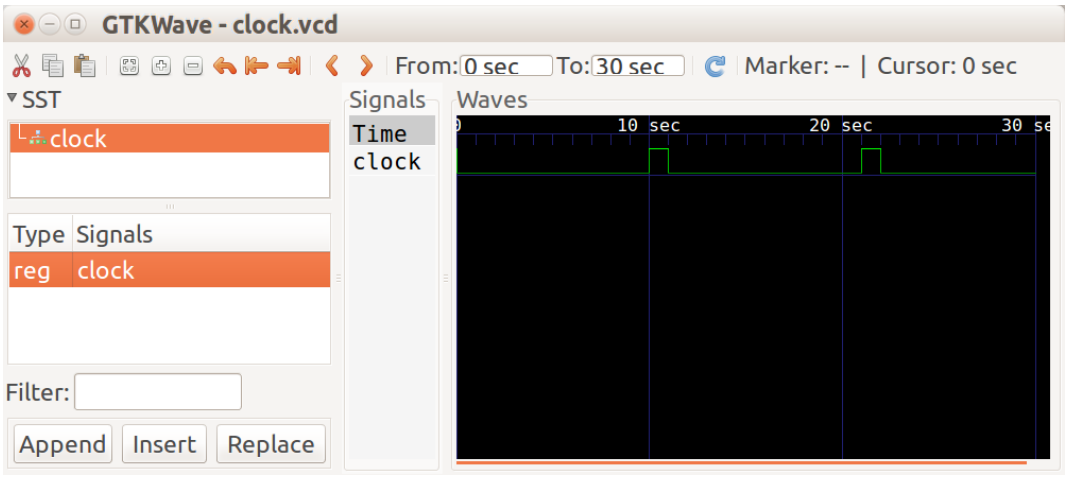
Alternativamente, possiamo visualizzare l'andamento della simulazione con `gtkwave`. Introduciamo due direttive

```
$dumpfile("clock.vcd");
$dumpvars;
```

subito dopo la `$monitor`, compiliamo e eseguiamo come fatto prima e lanciamo `gtkwave` passandogli come parametro il nome del file utilizzato nella `dumpfile`:

```
gtkwave clock.vcd
```

Si aprirà una finestra sulla quale possiamo selezionare in alto a sinistra (blocco "SST") il nome del modulo, prendere la variabile clock che comparirà in basso a sinistra nella lista della variabili del modulo ("Type Signals") e portarla nella lista in alto al centro (Colonna "Signals"), e vederne quindi l'andamento nel tempo:


GTKWave - clock.vcd
From: 0 sec
To: 30 sec
Marker: -- | Cursor: 0 sec
SST
clock
Type
Signals
reg
clock
Filter:
Append
Insert
Replace
Signals
Time
clock
Waves
10 sec
20 sec
30 se

Capitolo 2

Reti combinatorie

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

2.1 Componenti descritti con tabelle di verità

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

- L'intestazione del modulo richiede la parola chiave `primitive`, il nome del modulo e la lista che contiene il segnale in uscita (a sinistra, normalmente. Il segnale è per forza da un bit solo) e i segnali in ingresso.
- Il modulo termina con una `endprimitive`
- ciascuno dei segnali in ingresso è definito utilizzando la parola chiave `input` seguita dal nome del segnale. L'unico parmetro di uscita è definito da un identificatore preceduto dalla parola chiave `output`
- Il corpo del modulo è costituito da una tabella di verità compresa fra le keyword `table endtable`.
- Ciascuna riga della tabella di verità deve contenere:
  - tanti valori (ciascuno $\in \{0, 1, ?\}$) quanti sono gli ingressi, rappresentati nell'ordine
  - seguiti da un carattere "`:`"
  - seguito da uno 0 o un 1 che rappresenta il valore dell'uscita nel caso gli ingressi siano quelli specificati a sinistra dei `:`
  - ad esempio, una riga
    ```
    0 1 0 :  1
    ```
    a fronte dei parametri formali
    ```
    (output z, input x, input y, input w)
    ```
    indica che `z` varrà `1` quando `x = 0`, `y = 1` e `w = 0`.
- il valore `?` indica un non specificato.
- **Attenzione:** tutte le combinazioni di ingresso devono essere definite nella tabella di verità. Non è possibile abbreviare una `table` omettendo le righe con uscita `0` come invece facciamo quando disegnamo le taelle di verità con carta e penna, per semplicità.

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

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

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

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

endprimitive
```

2.1.2 Esempio: multiplexer con due ingressi da 1 bit

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

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

endprimitive
```

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

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

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

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

2.2 Componenti descritti con espressioni booleane

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

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

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

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

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

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

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

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

  assign

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

  assign

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

endmodule
```

In questo esempio osservate due cose:

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

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

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

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

endprimitive
```

possiamo osservare che il riporto è 1 quando:

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

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

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

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

2.3 Componeneti descritti in modo strutturale

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

Per definire un componente come rete di sottocomponenti occorre:

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

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

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

  wire ...;

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

endmodule
```

2.3.1 Esempio: sommatore di due bit

Utilizziamo come componenti i due moduli `primitive` definiti nella sezione 2.1.1 e 2.2, ovvero i moduli `fa_somma` e `fa_riporto`.

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

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

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

endmodule
```

2.4 Esempio: sommatore di due numeri da 2 bit

Supponendo di avere a disposizione un modulo `fulladder` come quello definito nella sezione 2.3.1, possiamo ottenere un `fulladder2` che opera su numeri a due bit collegando il riporto in uscita di un `fulladder` che somma i due bit meno significativi dei due ingressi e il riporto iniziale (presumibilmente `0`) ad un `fulladder` che somma i due bit più significativi dei due ingressi al riporto dell'altro `fulladder`, generando il bit più significativo del risultato **e** il bit di riporto finale (vedi schema in Fig. 2.1).

Tenendo presente che segnali da più di un bit si possono dichiarare facendo precedere l'identificatore da una coppia di interi fra parentesi quadre, separati da `:`, che rappresentano il valore del primo e dell'ultimo indice da utilizzare per accedere i singoli bit (vedi sezione 2.5), il modulo può essere definito come segue:

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

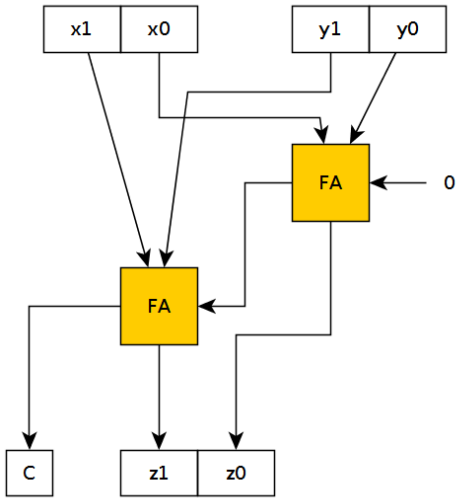


Figura 2.1: Addizionatore di due numeri da 2 bit costruito utilizzando due addizionatori di numeri da 1 bit con riporto

```
module add2(output riporto, output [1:0]somma,
            input ripin, input [1:0]x1, input [1:0]x2);

   wire     rips;

   fulladder fa0(rips, somma[0], ripin, x1[0], x2[0]);
   fulladder fa1(riporto, somma[1], rips, x1[1], x2[1]);

endmodule
```

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

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

2.5 Variabili e costanti

2.5.1 Variabili da più di un bit

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

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

Qualora volessimo utilizzare più bit (in un registro o in un filo) dobbiamo far precedere l'indentificatore da una coppia di parentesi quadre che al loro interno contengono un indice che rappresenta il bit più significativo e un indice che rappresenta il bit meno significativo, separati dal simbolo `:` (vedi Fig. 2.2). La possibilità di utilizzare un

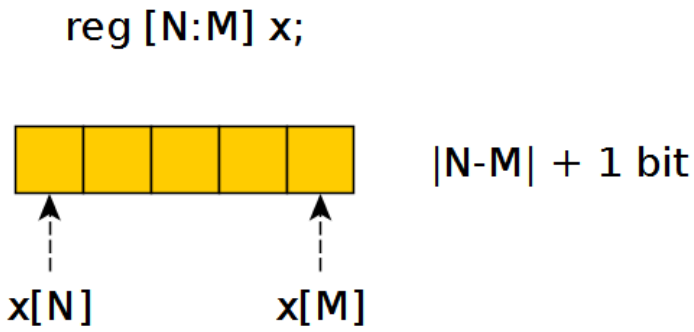


Figura 2.2: Dichiarazione di un registro da più bit

formalismo che permette di indicare range di indice diversi offre possibilità che si adattano a qualunque esigenza di rappresentazione degli indici:

- `reg [7:0] b1;`
  dichiara un registro da 8 bit (valore assoluto di (7-0) più 1). Il bit più significativo è `b1[7]` e quello meno significativo è `b1[0]`. In questo caso l'indice indica il peso del bit: se `b1[i]==1` allora il suo peso è 2^i altrimenti il suo peso è 0.

- `reg [0:7] b;2`
  dichiara un registro da 8 bit (valore assoluto di (7-0) più 1). Il bit più significativo è `b1[0]` e quello meno significativo è `b1[7]`.

- `reg [1:8] b3;`
  dichiara un registro da 8 bit (valore assoluto di (7-0) più 1). Il bit più significativo è `b1[1]` e quello meno significativo è `b1[8]`. L'indice indica la posizione del bit, secondo l'ordinamento "naturale" da destra a sinistra, partendo da 1.

2.5.2 Costanti

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

- `4'b0111` indica la costante 7_{10} rappresentata su 4 bit in binario. Quindi il "4" sta per 4 bit, la "b" per binario e la stringa "0111" rappresenta il valore

- `4'd7` indica la stessa costante, espressa in **d**ecimale

- `4'o07` indica la stessa costante, espressa in base **o**ttale

- `8'xff` indica il numero 255, espresso in esadecimale (he**x**adecimal)

2.5.3 Giustapposizione

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

2.5.4 Campi di un valore da n bit

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

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

2.6 Moduli parametrici

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

2.6.1 Multiplexer da due ingresso con numero di bit parametrico

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

```
module mux(output [N-1:0] z,
           input ctrl, input [N-1:0] x1, input [N-1:0] x2);

  parameter N = 8;

  assign
    z = (ctrl == 0 ? x1 : x2);

endmodule
```

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

```
  reg [15:0] x1;
  reg [15:0] x2;
  reg ctrl;
  wire [15:0] z;

  mux #(16) mux16(z,ctrl,x1,x2);
```

2.6.2 Ritardi

Nel testbench (programma di prova di un modulo) abbiamo utilizzato la sintassi `#3 x=0;` che significa "attendi 3 unità di tempo e poi assegna 0 a x". Possiamo denotare l'attesa di 5 unità di tempo inserendo il comando `#5;`. Possiamo anche definire un ritardo anche nelle `assign` di un modulo (facciamo precedere all'identificatore cui si assegna il valore un termine `#n` che indichi il ritardo fra la valutazione della parte destra e l'assegnamento del valore risultante alla parte sinistra dell'espressione, per modellare in modo esplicito i ritardi[4]. Noi intenderemo un `#1` come un singolo Δt. Volendo quindi associare un ritardo ai nostri moduli, potremo anche vedere che succede a livello temporale nella comparsa dei risultati dopo il cambiamento degli ingressi dei nostri moduli sotto test. Per esempio, potremmo modificare il modulo `somma` della sezione 2.2.1 come segue:

[3] in forma più compatta `b & 8'x0f`

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

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

  assign
    // ritardo di due delta t : uno per il livello AND
    // e uno per il livello  OR
    #2 z = (~riportoiniziale & ~x & y) |
           (~riportoiniziale & x & ~y) |
           (riportoiniziale & ~x & ~y) |
           (riportoiniziale & x & y);

  assign

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

endmodule
```

In questo modo potremo vedere come le uscite del sommatore avvengano dopo 2 unità di tempo dalla stabilizzazione degli ingressi. Senza modificare il programma che crea un addizionatore di numeri da due bit a partire da questo fulladder da 1 bit (vedi sezione 2.4), potremmo anche vedere che il risultato finale si ha dopo $4\Delta t$ per via del collegamento in cascata dei due fulladder (ciascuno con ritardo da $2\Delta t$).

Capitolo 3

Reti Sequenziali

Queste note ricapitolano quello che serve per programmare e testare (funzionalmente) componenti logici che siano reti sequenziali sincrone utilizzando Verilog (non System Verilog!). Queste note danno per scontato quanto descritto nelle note sull'implementazione di reti combinatorie in Verilog.

3.1 Modello strutturale

Un primo modo di realizzare le reti sequenziali è quello di programmarle come reti formate dai tre componenti: le due reti combinatorie per il calcolo delle uscite e del prossimo stato interno e il componente registro di stato. Le componenti per il calcolo delle uscite e del prossimo stato interno sono reti combinatorie e possono essere programmate come visto nelle note "Reti combinarorie in Verilog", ovvero utilizzanto una o più moduli di tipo `primitive` o un singolo modulo di tipo `module`. Per la realizzazione del componente registro utilizziamo un `module` programmato in modo behavioural che rispetta la semantica del registro come vista a lezione:

```
1  module registro(output [N-1:0]z, input [N-1:0]x, input en, input clk);
2
3     // si puo' definire la lunghezza del registro come parametro del modulo
4     parameter N = 8;
5
6     // questo e' il dispositivo fisico che contiene lo stato
7     reg [N-1:0] s;
8
9     // inizializzazione (visto che non abbiamo il reset)
10    initial
11      s = 0;
12
13    // funzione di transizione dello stato interno:
14    // quando il clock va alto, in presenza di enable
15    // memorizza il valore che trovi in ingresso
16    always @(posedge clk)
17      begin
18   if(en==1)
19     s = x;
20      end
21
22    // il valore dell'uscita e' sempre il valore del registro
23    assign z = s;
24
25 endmodule // reg
```

Il modulo è parametrico. Per default definisce un registro da 8 bit (definizione del `parameter N` alla riga 4). All'interno definiamo un `reg` Verilog che conterrà lo stato del nostro registro (riga 7). Il blocco `initial` serve ad inizializzare il registro a 0 (righe 10–11). Il blocco `always @{posedge clk}` che va dalla riga 16 alla 20 controlla la scrittura nel registro: ad ogni fronte di salita del segnale di clock (`posedge`) se è a 1 il segnale di abilitazione in scrittura (`en`) si memorizza nel registro quanto presente in quel momento sull'ingresso. L'`assign` della riga 23 fa sì che l'uscita del registro corrisponda sempre al suo contenuto, indipendentemente dal valore del segnale `en`.

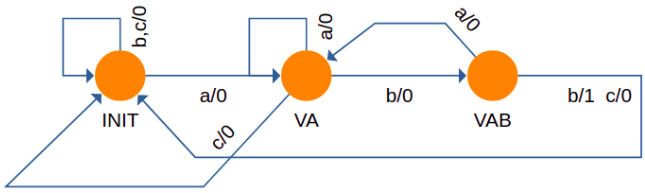
Utilizzando questo modulo, la generica struttura di un modulo che implementa una rete sequenziale sarà una cosa tipo quella che segue:

```
1  module ReteSeq(output [...]uscita, input [...]ingresso, input clock);
2
3    // dichiarazione dei wire che permettono di collegare l'uscita
4    // del registro di stato alla rete che calcola le uscite e a
5    // quella che calcola lo stato interno successivo e che permettono
6    // di collegare la rete che calcola lo stato interno successivo
7    // all'ingresso del registro di stato
8
9    wire uscitaregistro;
10   wire ingressoregistro;
11
12   // dichiarazione del modulo registro per lo stato interno
13   registro #(...) statointerno(uscitaregistro, ingressregistro, ...);
14
15   // dichiarazione del modulo che calcola il prossimo stato interno
16   prossimostato next(ingressoregistro, uscitaregistro, ingresso):
17
18   // dichiarazione del modulo che calcola l'uscita
19   // se la rete fosse di mealy avremmo
20   uscita z(uscita, ingresso, uscitaregistro);
21   // se fosse di moore sarebbe
22   // uscita z(uscita, uscitaregistro);
23
24 endmodule
```

Da notare alla linea 13 la riscrittura del parametro N del modulo registro (`#(...)`) che permette di definire un registro con l'esatto numero di bit richiesti.

3.1.1 Esempio: riconoscitore di sequenze "abb"

Consideriamo l'automa di Mealy in figura, che riconosce le sequenze "abb" all'interno di sequenze di caratteri appartenenti all'insieme `{a,b,c}`.



Lo stato iniziale ì `INIT`, `VA` è lo stato in cui ci ricordiamo di aver visto una a e `VAB` quello in cui ci ricordiamo di aver visto una a seguita da una `b`. Immaginiamo di codificare i tre stati `{INIT,VA,VAB}` come segue: `INIT=00`, `VA=01` e `VAB = 11`. E analogamente utilizziamo `00,01,11` per codificare rispettivamente `a,b,c`.

Con queste convenzioni, la tabella di verità per la funzione delle uscite sarà (s1 ed s0 rappresentano il bit più significativo dello stato e quello meno significativo, rispettivamente. x1 ed x0 sono i bit che rappresentano l'ingresso corrente):

```
 s1 s0 x1 x0 | z
-------------+----
 0  0  -  -  | 0
 0  1  -  -  | 0
 1  1  0  1  | 1
 1  1  1  -  | 0
 1  1  0  0  | 0
 1  0  -  -  | 0
```

Pertanto la funzione che calcola l'uscita potrà essere scritta come:

$$z = s_1 s_2 \overline{x_1} x_0$$

e quindi realizzata mediante il modulo Verilog:

```
module z(output zeta, input [1:0]s, input [1:0]x);

   assign zeta = s[1]&s[0]&(~x[1])&x[0];

endmodule // z
```

Per il calcolo dello stato interno abbiamo un tabella di verità diversa:

s1	s0	x1	x0	s1's0'
0	0	0	0	0 1
0	0	-	1	0 0
0	1	0	0	0 1
0	1	1	1	0 0
0	1	0	1	1 1
1	1	0	0	0 1
1	1	-	1	0 0

Possiamo utilizzare un modulo con due assign, una che controlla il primo bit dell'uscita (s1') e una che controlla il secondo (s0')[1]:

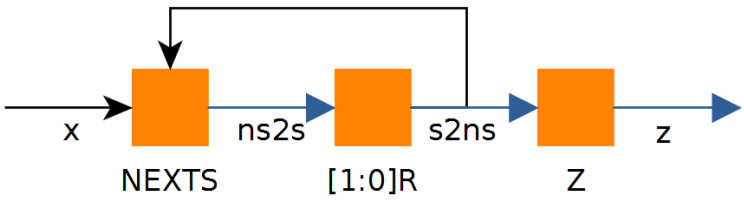
```
module nexts(output [1:0]s1,  input [1:0]s, input [1:0]x);

   assign
     s1[1] = (~s[1]) & s[0] & (~x[1]) & x[0];

   assign
     s1[0] = ((~x[1])&(~x[0])) | ((~s[1]) & s[0] & (~x[1]));


endmodule // z
```

Con questi due moduli e il modulo registro precedentemente descritto, possiamo rappresentare la rete sequenziale di Mealy che implementa l'automa secondo il modello strutturale come segue:

```
module fsm_me(output y, input [1:0]x, input clock);

   // wire necessari a connettere i componenti
   // uscita della rete combinatoria che calcola il nuovo stato
   wire [1:0] ns2s;
   // uscita del registro di stato
   wire [1:0] s2ns;

   // il registro di stato (enable sempre a 1: ogni ciclo di clock scrive)
   registro #(2) stato(s2ns,ns2s,1'b1,clock);

   // rete combinatoria che calcola il valore del prossimo stato a partire
   // dallo stato corrente e dagli ingressi (RETE DI MEALY)
   nexts prossimostato(ns2s,s2ns,x);
   // rete combinatoria che calcola l'uscita a partire
   // dallo stato corrente e dagli ingressi (RETE DI MEALY)
   z zeta(y, s2ns, x);

endmodule
```

Lo schema implementato è esattamente quello della figura che segue:

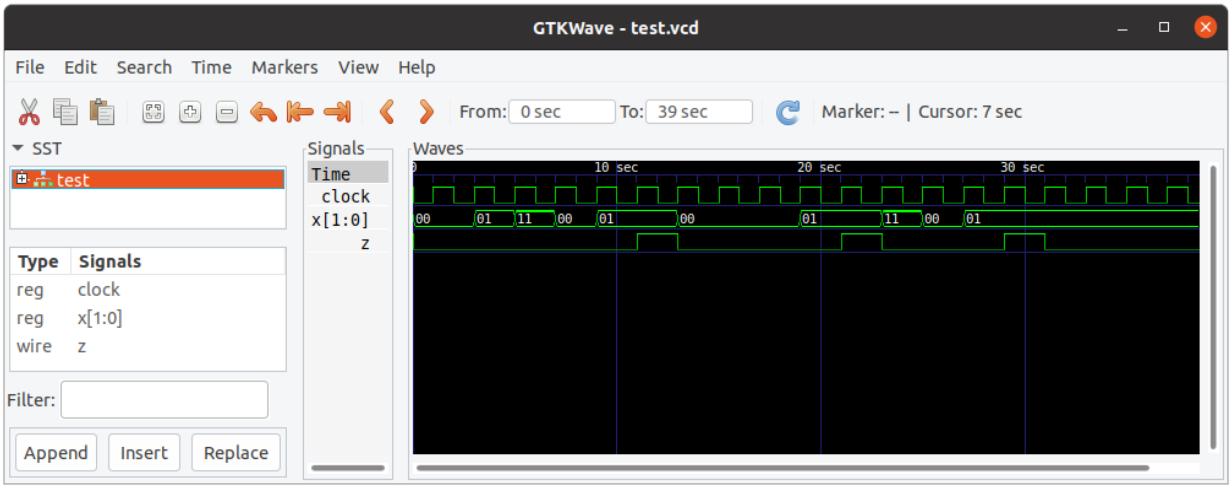
[1] l'espressione booleana per s0' è in realtà ottenuta studiando la relativa mappa di Karnaugh; diversamente avremmo dovuto avere un or di 4 termini, visto che nella colonna s0' abbiamo 4 "1"



Il comportamento della rete può essere testato utilizzando il testbench che segue:

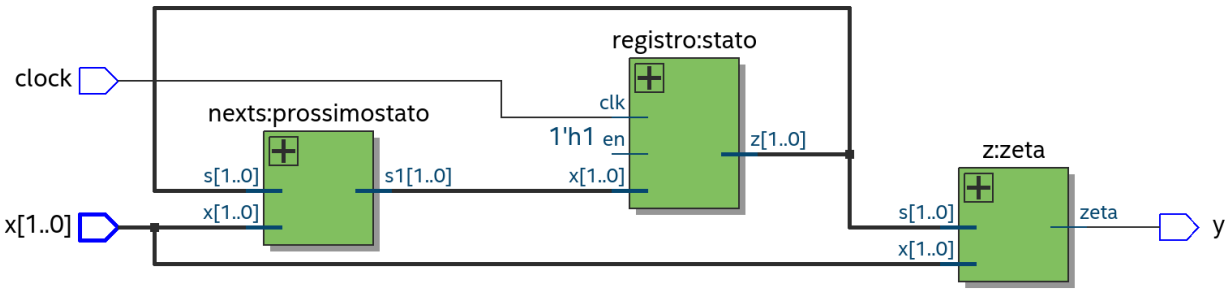
```
module test();

   // ingressi
   reg [1:0] x;
   // clock;
   reg       clock;
   // uscita
   wire      z;


   // modulo sotto test
   fsm_me mealy(z, x, clock);

   // generazione del segnale di clock
   always
     begin
  #1 clock = ~clock;
     end

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

  clock = 0;
  x = 0;  // x = A

  // x = B
  #3 x = 1;
  // x = C;
  // #2 x = 3;
  #2 x = 2'b11;
  // x = A
  #2 x = 0;
  #2 x = 1;
  #2 x = 1;


  // sequenza a a a b b c a b b
  #2 x = 0;
  #2 x = 0;
  #2 x = 0;
  #2 x = 1;
  #2 x = 1;
  #2 x = 3;
  #2 x = 0;
  #2 x = 1;
  #2 x = 1;

  #10 $finish;
     end
endmodule // test
```

Il risultato dell'esecuzione del testbench visualizzato con gtkwave fa vedere che effettivamente vengono riconosciute tutte e sole le sequenze abb:

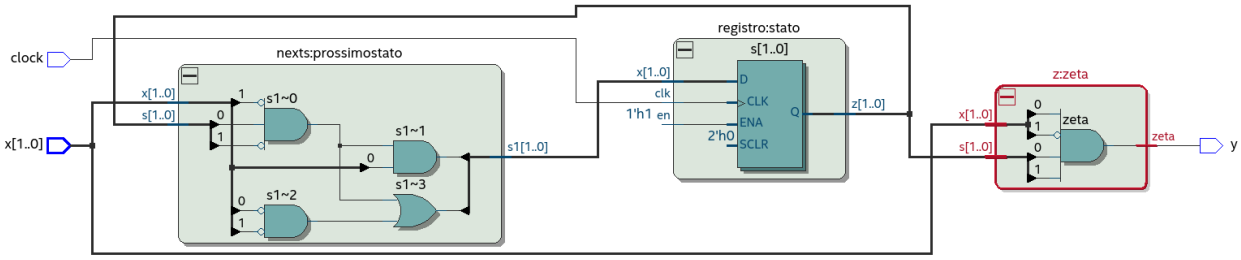


al sesto ciclo clock alto, dopo aver "visto" una a (00) e due b (01 per due clock alti) e più avanti sempre nelle stesse condizioni.

La compilazione con Quartus genera il circuito riportato in figura:



in cui si riconoscono chiaramente la logica per il calcolo del nuovo stato interno (a sinistra), qualla per il calcolo dell'uscita (and a destra) e infine il registro (hw) utilizzato per mantenere lo stato interno (al centro). "Aprendo" i vari blocchi si vede come i blocchi di logica combinatoria corrispondano alle porte utilizzate nel codice e come il `registro` sia implementato con blocchi registro hardware.



3.2 Verilog behavioural

Il linguaggio Verilog può essere utilizzato per programmare componenti in modo molto simile a come si programmano funzioni e procedure in un linguaggio imperativo. In particolare, un `module` può essere realizzato a partire da:

- dichiarazioni di parmetri (`parameter <nome>=<valore>`);

- dichiarazioni di `reg` (stato interno) e `wire` (collegamenti);
- istanze di altri moduli (di tipo `primitive` o `module`) utilizzanod le variabile `reg` o `wire` dichiarate precedentemente e/o le variabili che fanno parte della lista dei parametri formali del modulo come parametri attuali dell'istanza del modulo;
- blocchi di tipo `initial`, i cui comandi vengono eseguiti all'attivazione ("accensione") del modulo stesso;
- blocchi di tipo `always`, i cui comandi vengono eseguiti ogni volta che valgono le condizioni di attivazione dell'`always` e, in particolare:
  - per comandi `always begin ... end`, i comandi fra begin ed end sono eseguiti in continuazione. Questo tipo di blocco `always` richiede che all'interno sia presente almeno uno statement con un ritardo (ovvero con un'espressione `#...`);
  - per comandi `always @( ... lista variabili ...) begin end`, i comandi sono eseguiti ogni volta che una dele variabili nella lista cambia valore. In questo caso la lista delle variabili può contenere, separati dalla virgola o nomi di variabili o espressioni contenenti le parole chiave `posedge` o `negedge` seguite da una variabile. In questo caso l'esecuzione dei comandi nel blocco scatta non già quando varia il valore della variabile ma solo quando tale valore passa da 0 a 1 (`posedge`) o da 1 a 0 (`negedge`);
- comandi di tipo `assign`, per assegnare in modo continuo il valore di un'espressione a una variabile.

Esitono molte altre possibilità, che noi non consideriamo in questa sede, essendo queste più che sufficienti per lo scopo del corso.

I comandi che possiamo utilizzare all'interno dei blocchi appena descritti hanno una sintassi stile C e comprendono:

- **assegnamenti** che possono essere sincroni (segno =) o asincroni (segno <=). Quelli sincroni avengono in modo bloccante. La sequenza di comandi

  ```
  a = 0;
  b = a & c;
  ```

  assegna 0 a b indipendentemente dal valore di c, perchè *prima* viene eseguito l'assegnamento di a e *poi* quello di b. La sequenza di comandi:

  ```
  a <= 0;
  b <= a & c;
  ```

  esegue i due assegnamenti in parallelo e alla fine a vale 0 e b vale il risultato dell'and fra il valore che aveva a prima dei comandi e il valore di c.

- **condizionali** nella forma `if(...) <comando>;`
  oppure `if(...) <comando> else <comando>;`

- **scelta condizionale** nella forma

  ```
  case(<variabile>)
  <valore>: <comando>
  <valore>: <comando>
  ...
  default: <comando>
  endcase
  ```

  (dove il `default` può essere omesso)

Non esite un comando for come lo intendiamo in C. Esiste un for cosiddetto "generativo" che permette di dichiarare una serie di componenti. L'indice del for va dichiarato precedentemente come `genvar` e successivamente si può utilizzare un blocco

```
generate
  for( .... )
    begin
      ...
    end
endgenerate
```

dove all'interno del for si possono dichiarare *istanze di moduli* utilizzando nei parametri attuali delle istanze l'indice generato dal for. Per esempio, il codice che segue istanzia una serie di moduli che prendono ognuno un bit di una variabile x e generano ognuno un bit di una variabile y:

```
genvar i;
...
generate
 for(i=0; i<N; i=i+1)
   begin
     modulo m(y[i], x[i]);
   end
endgenerate
```

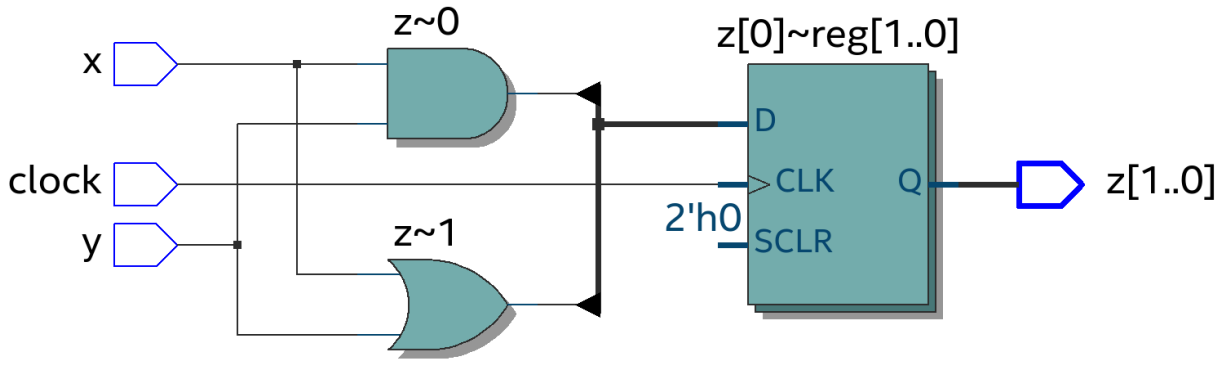
In conclusione, con il modo behavioural di programmare i componenti `module` di Verilog possiamo definire in modo programmatico:

- cosa fare per inizializzare il modulo (statement o blocco `initial`);
- cosa fare quando variano i valori in ingresso (statement o blocchi `always` e `assign`).

Nei comandi di assegnamento valgono regole diverse a seconda di come sia stata dichiarata la variabile di cui vogliamo cambiare il valore. All'interno dei blocchi behavioural `initial` e `always` si possono assegnare valori (con assegnamento sincrono o asincrono) alle sole variabili definite come `reg`. In uno statement `assign` si possono assegnare valori alle sole variabili dichiarate come `wire`. Le variabili che compaiono in `output` nella lista dei parametri formali del modulo devono essere intese come variabili di tipo `wire`. Si può comunque cambiarne il tipo utilizzando fra la parola chiave `output` e l'identificatore della variabile la parola chiave `reg`. I tool di sintesi saranno in grado di evincere il tipo corretto per la variabile di uscita. Ad esempio, un modulo tipo:

```
module m(output reg [1:0]z, input x, input y, input clock);

  always @(posedge clock)
  begin
    z[1] <= x & y;
    z[0] <= x | y;
  end

endmodule
```

genererà un registro per contenere l'uscita z, di fatto definendo un componente di logica sequenziale. La sintesi del `module` con Quartus genera infatti il seguente circuito:



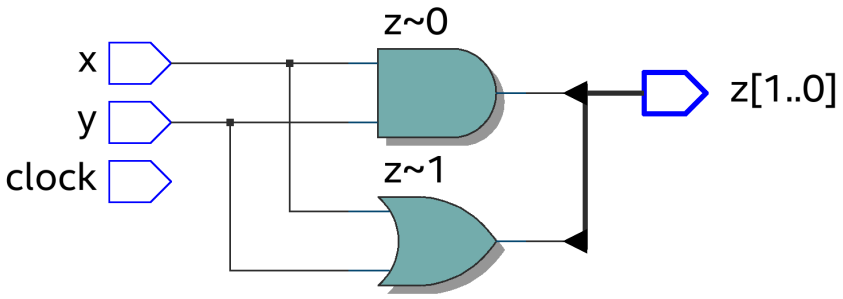
Un modulo tipo:

```
module m(output reg [1:0]z, input x, input y, input clock);

  always @(*)
  begin
  z[1] <= x & y;
  z[0] <= x | y;
  end

endmodule
```

oppure un modulo tipo:

```
module m(output [1:0]z, input x, input y);

  assign
     z[1] <= x & y;
  assign
     z[0] <= x | y;

endmodule
```

generano invece entrambi un circuito senza registri (cioè della logica combinatoria):



a dimostrazione del fatto che la sintesi è determinata, oltre che dalla dichiarazione del `reg` anche da suo utilizzo.

3.3 Modello behavioural

Il modello "behavioural" permette di programmare la rete sequenziale utilizzando codice Verilog behavioural invece che di realizzarla come collegamento di moduli di registro e di logica combinatoria esistenti. In questo caso quello che si fa è programmare un modulo `module`:

- utilizzando un `reg` per mantenere lo stato interno,
- utilizzando un altro `reg` per rappresentare il prossimo stato interno,
- utilizzando un blocco `initial` per inizializzare lo stato interno,
- utilizzando un blocco `always @(ingressi, stato)` per calcolare il prossimo stato interno,
- utilizzando un blocco `always @(posedge clock)` per aggiornare il registro di stato con il valore calcolato come prossimo stato interno,
- utilizzando una o più `assign` per generare le uscite.

La struttura del modulo dovrebbe essere quindi qualcosa del tipo:

```
module ReteSeq(output [...]uscita, input [...]ingresso, input clock);

  reg [...] stato, prossimostato;

  intial
    stato <= ...;
```

```
7
8   always @(posedge clock)
9     stato <= prossimostato;
10
11  always @(ingresso, stato)
12    begin
13    // calcolo di prossimostato
14    end
15
16  assign
17    uscita = ... ;
18
19 endmodule
```

Si noti che gli assegnamenti a stato nel blocco `initial` e a prossimostato nel blocco `always` sono assegnamenti asincroni (<= invece che semplicemente =). Questo fa sì che avvengano in parallelo. Se così non fosse, potremmo avere un comportamento non corretto con conseguenze che possono impattare anche la correttezza del calcolo dell'uscita.

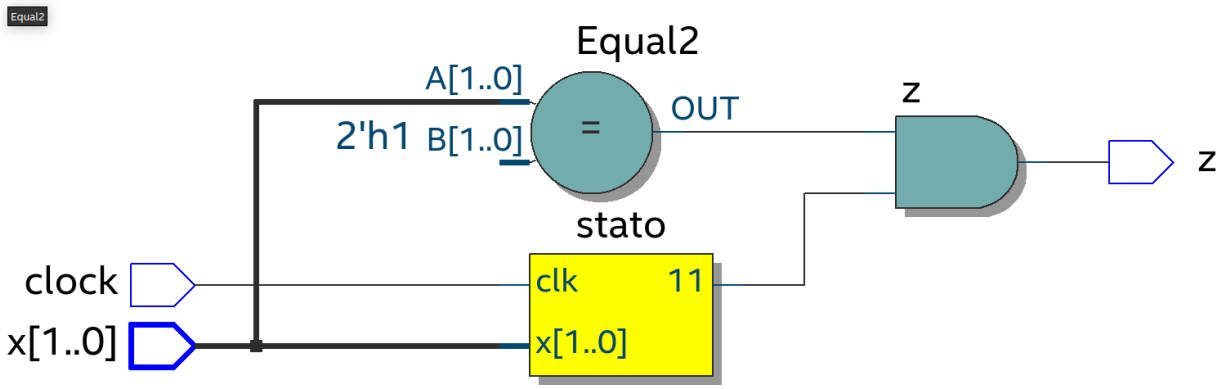
3.3.1 Riconoscitore di sequenze abb

Secondo i principi appena descritti, il riconoscitore di sequenze (rete di Mealy) descritto nella sezione 3.1.1 può essere implementato come segue:

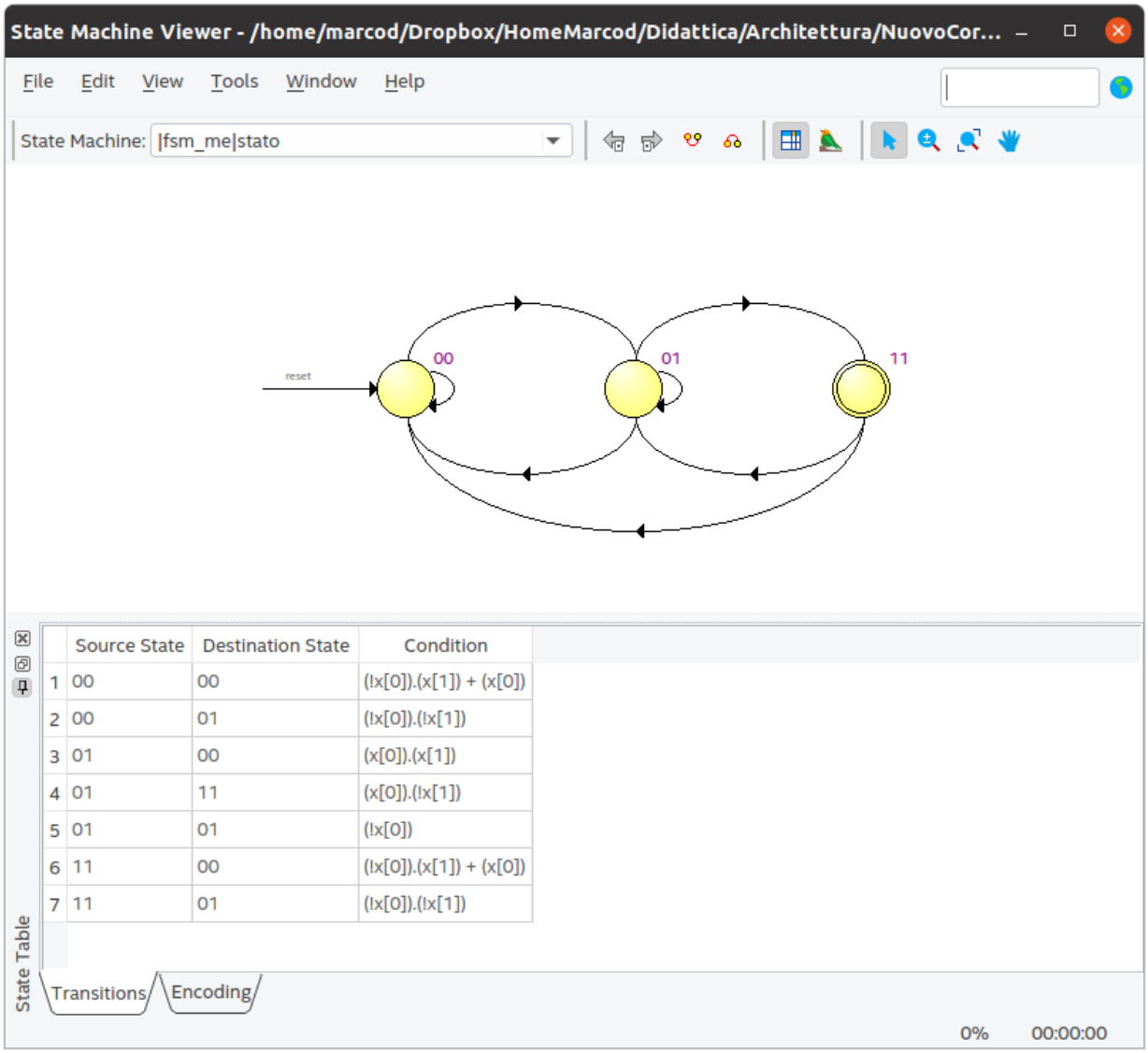
```
1  `define INIT 2'b00
2  `define VA   2'b01
3  `define VAB  2'b11
4
5  `define A 2'b00
6  `define B 2'b01
7  `define C 2'b11
8
9  module fsm_me(output z, input [1:0] x, input clock);
10
11    reg [1:0] stato;
12    reg [1:0] nuovostato;
13
14    initial
15      stato = `INIT;
16
17    always @(posedge clock)
18      begin
19   stato <= nuovostato;
20      end
21
22    always @(x, stato)
23      begin
24   case(stato)
25     `INIT :
26       begin
27          nuovostato <= (x == `A ? `VA : `INIT);
28       end
29     `VA :
30       begin
31          nuovostato <= (x == `B ? `VAB : (x == `C ? `INIT : `VA));
32       end
33     `VAB :
34       begin
35          nuovostato <= (x == `A ? `VA : `INIT);
36             end
37   endcase // case (stato)
38      end
39
40    assign
41      z = (((stato == `VAB) && (x == `B)) ? 1 : 0);
42
43 endmodule // fsm_be
```

Si noti che abbiamo utilizzato costanti simboliche sia per la rappresentazione dei diversti stati che per la rappresentazione dei diversi caratteri.

Questa volta, Quartus sintetizza una rete con componenti nettamente diversi:



ed in particolare il blocco in basso al centro è inteso essere un automa. Aprendone la descrizione come automa vediamo questo:



	Source State	Destination State	Condition
1	00	00	(!x[0]).(x[1]) + (x[0])
2	00	01	(!x[0]).(!x[1])
3	01	00	(x[0]).(x[1])
4	01	11	(x[0]).(!x[1])
5	01	01	(!x[0])
6	11	00	(!x[0]).(x[1]) + (x[0])
7	11	01	(!x[0]).(!x[1])

che è esattamente (e correttamente) l'automa da cui eravamo partiti, derivato però dal codice Verilog del listato.

i

Capitolo 4

Materiale sul libro di testo

In questa sezione mettiamo in evidenza cosa si può utilizzare del libro di testo [2] e cosa invece differisce sostanzialmente, essendo peculiare del System Verilog e non incluso nel Verilog standard.

Parte	**Note**
Reti combinatorie: sez 4.1, 4.2 e 4.3 (esclusa 4.2.8)	Questa parte può essere utilizzata quasi così com'è. L'unica vera differenza fra il System Verilog e il Verilog è l'utilizzo della parola chiave `logic` al posto delle due parole chiave `reg` e `wire` del Verilog. `logic` indica una variabile e il suo tipo (registro o wire) è derivato dall'uso che se ne fa. `iverilog` accetta la dichiarazione `logic` ma apparentemente la utilizza come se fosse una dichiarazione di registro. La Sez 4.2.8 non va considerata, visto che on abbiamo trattato i valori `x` e `z` Gli esercizi con le `tristate` non li abbiamo discussi.
Registri, costrutti behavioural e reti sequenziali: Sez. 4.4, 4.5, 4.6	In questa parte il libro utilizza pesantemente i nuovo costrutti `always_comb` e `always_ff` che non fanno parte del Verilog. La maggior parte degli esempi non compila con `iverilog`. Caso per caso andrà considerato del codice equivalente che però rispetti la sintassi Verilog.
Sez. 4.7	Sostanzialmente da non considerare. I tipi di dato da utilizzare sono descritti nella dispensa
Sez. 4.8	Verilog utilizza la sintassi leggermente diversa presentata nelle dispense.
Sez. 4.9	La struttura dei testbench è la stessa del Verilog, ma alcuni costrutti sono introdotti nel System Verilog. Da utilizzare la sintassi delle dispense.

Capitolo 5

Sintesi

Verilog può essere utilizzato per derivare la progettazione di un componente fisico che implementi il circuito descritto dal programma. Questo processo viene di solito indicato con il termine *sintesi*. Esistono tutta una serie di strumenti che permettono di utilizzare Verilog (o altri linguaggi RTL) per la sintesi di circuiti, che possono essere divisi in due classi fondamentali:

- gli strumenti per la programmazione di FPGA, e
- gli strumenti per la progettazione VLSI.

5.1 Programmazione di FPGA

Gli strumenti di programmazione delle FPGA generano, a partire dal sorgente Verilog, una configurazione per una specifica FPGA (Field Programmable Gate Array), ovvero per un dispositivo che contiene da decine di migliaia a centinaia di milioni di *celle*, organizzate in griglie regolari, ciascuna delle quali può essere configurata in tre diversi modi:

- per calcolare una funzione di un piccolo numero di bit (normalmente da 3 a 7) che calcola un singolo bit,
- per implementare un singolo bit di memoria (latch o flip flop),
- per eseguire il routing di qualcosa calcolato dalle celle adiacenti verse altre celle adiacenti.

Ogni modello di FPGA dispone di celle diverse sia per numero che per disposizione e per capacità di calcolo. FPGA "piccole", possono avere qualche migliaio di celle. Quelle più grosse arrivano a decine di milioni di celle. Inoltre, le FPGA hanno di norma al loro interno colonne di celle "dedicate" ovvero "macro" celle (notevolmente più grosse delle celle normali) che possono eseguire semplici operazioni aritmetiche (normalmente in virgola mobile) tipo *multiply-and-add*[1] oppure che implementano piccoli moduli di memoria di qualche Kbyte. La presenza di queste macro celle permette di realizzare circuiti più complessi, a patto di riuscire e implementare nel resto delle celle la logica che permette di alimentare i moduli di calcolo o di calcolare i valori da scrivere nei moduli di memoria.

Il risultato dell'esecuzione del processo di sintesi di un circuiti con gli strumenti di programmazione delle FPGA sarà dunque un file di configurazione (una specie di file di boot della FPGA) che, quando caricato dal componente FPGA la programma in modo che le diverse celle e macro celle si comportino come necessario per implementare il circuito da cui siamo partiti. Parte integrante del file di configurazione (normalmente chiamato *bitstream*) è la definizione di quali piedini del chip FPGA vengano utilizzati per i segnali di ingresso, di uscita e per importare segnali particolari, tipo il segnale di clock.

I maggiori costruttori di FPGA mettono a disposizione strumenti diversi per la programmazione dei loro chip. Altera ha Quartus, Xilinx ha Vivado e in entrambi i casi esiste una versione che non necessita di licenza e che può essere utilizzata per progetti di limitate dimensioni e che producono file di configurazione per le FPGA di gamma bassa (quelle più piccole ed economiche). E' da tenere in conto che il processo di generazione del file

[1] tipica operazione per calcolo scientifico: prende due operandi, ne calcola il prodotto e lo somma a un registro che, partendo da 0, accumula la somma di tutti i prodotti calcolati fino a quel momento

di configurazione comprende fasi che di per sè sono molto complesse (gli algoritmi sono nella classe NP) che richiedono l'applicazione di diversi tipi di euristiche. Ad esempio, come mappare l'insieme di celle necessarie sull'insieme di celle disponibili è un problema di graph placement dimostrabilmente non polinomiale nel caso in cui si cerchi la soluzione ottimale. Questa situazione ha due tipi di conseguenze che vale la pena di citare:

- il tempo di esecuzione del processo di sintesi è notevole. Semplici circuiti possono richiedere uno o più minuti di calcolo per produrre il file di configurazione su un processore stato dell'arte per laptop/server. Circuiti più complessi richiedono ore di calcolo;

- come sempre quando si usano euristiche, un eventuale ottimizzazione "a mano" può produrre risultati migliori sia in termini di occupazione delle celle sulla FPGA (usarne meno) che in termini di performance (ottenere una configurazione che implementa il circuito in modo più veloce).

5.2 Programmazione di VLSI

[2] Gli strumenti di progettazione di circuiti VLSI (Very Large Scale Integration) permettono di progettare tutte le operazioni necessarie a costruire un chip che implementi il circuito descritto dal programma Verilog. Partendo dal programma Verilog si genera quindi una *netlist* dei componenti necessari (porte logiche, blocchi predefiniti (e.g. ALU o memorie), etc.) e dei relativi collegamenti. Quindi si passa alla progettazione del layout sul chip (disposizione dei componenti sulla superficie di silicio disponibile), alla generazione delle maschere che servono per le fasi di produzione del chip stesso. Il risultato finale è la produzione di un chip che implementa il circuito. Questo processo, differentemente da quello per la programmazione delle FPGA è una cosa i cui tempi si misurano in giorni o mesi. Ciascuna delle fasi è soggetta a procedure di verifica dei risultati ottenuti che possono richiedere molto più tempo della fase stessa di produzione di quei risultati. L'intero processo è estremamente costoso e solo i grandi produttori si possono permettere di intraprenderlo.

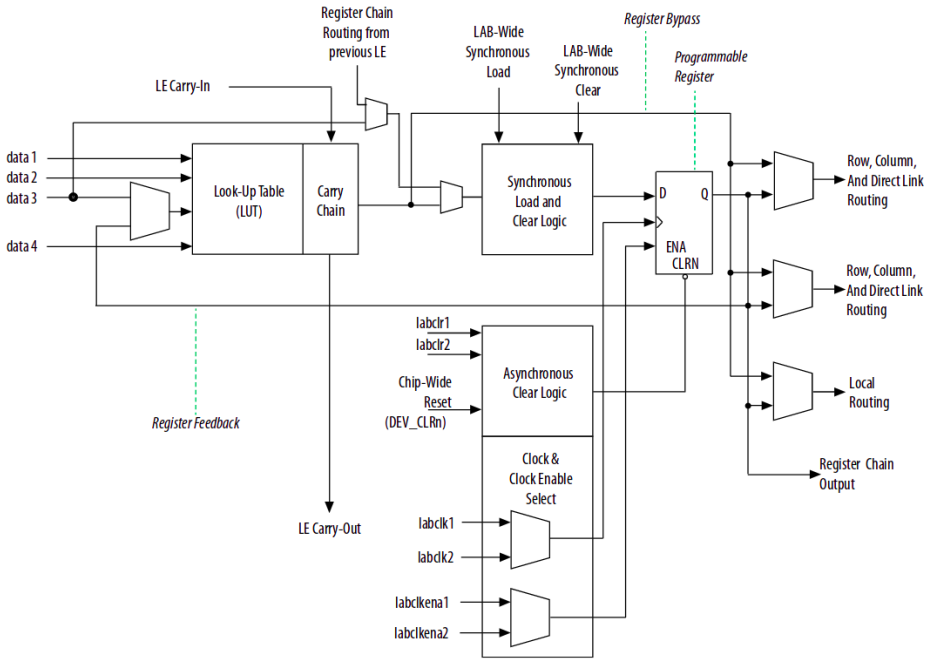
5.3 Esempio di sintesi con Quartus Lite Intel

Facciamo vedere a solo scopo illustrativo come posso ottenere un l'implementazione di un circuito descritto in Verilog su una FPGA. Illustriamo solo i passi necessari per vedere come il circuito verrà implementato e non diciamo nulla di come in effetti produrre il file di configurazione, che richiede azioni più specifiche e che dipendono anche dal tipo di dispositivo FGPA considerato.

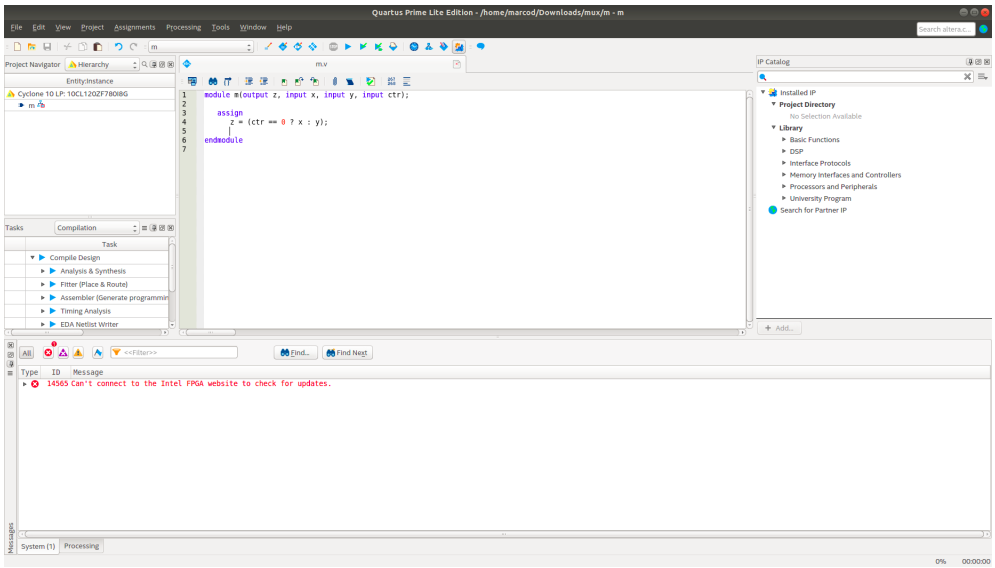
Supponiamo di voler implementare un semplice circuito mutiplexer, che accetta 2 ingressi da 1 bit e produce una uscita di 1 bit. L'uscita riporta il segnale presente su uno dei due ingressi scelto a seconda della configurazione di un ulteriore ingresso di controllo.

Il primo passo è quello di aprire il software di sviluppo e creare un progetto. Nella fase di creazione del progetto viene richiesto quale tipo di linguaggio si utilizza (Verilog o VHDL, per esempio) e quale FPGA vogliamo utilizzare. In questo esempio abbiamo scelto di utilizzare una Cyclone 10 con 119088 celle, 576 macro celle moltiplicatore e ca. 4M bit in blocchi di memoria. A titolo informativo, il singolo "logic element" (ovvero cella) di questa FPGA è fatto così:

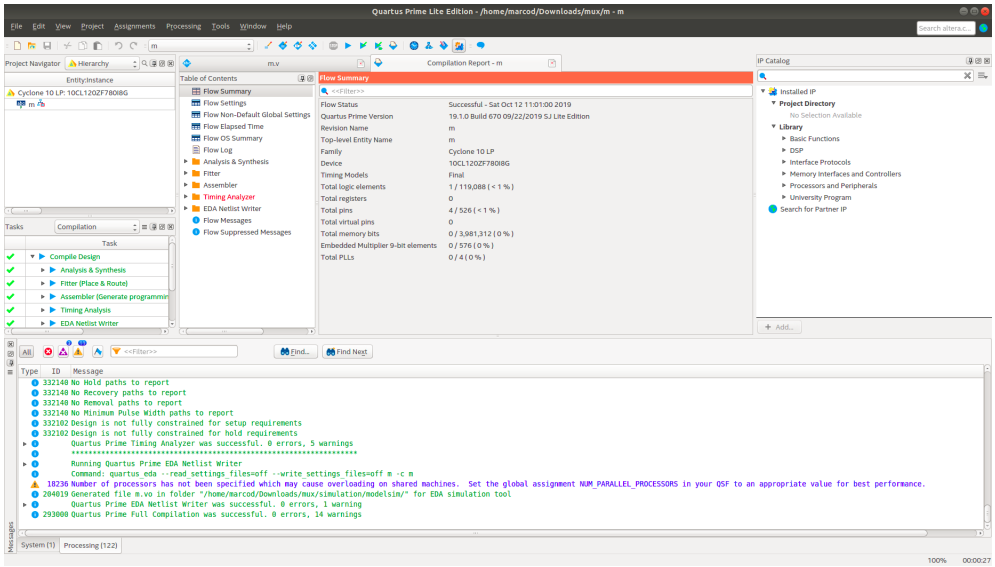
[2] La trattazione dell'argomento qui è assolutamente superficiale e ha il solo scopo di dare un'idea generale delle differenze del processo rispetto alla sintesi per FPGA.



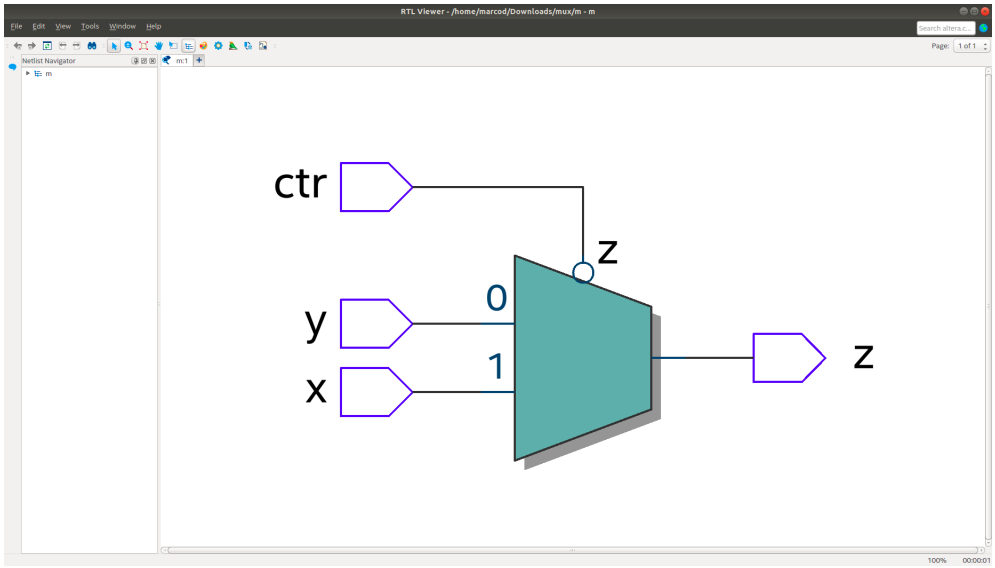
La schermata che segue riporta la configurazione iniziale, con il file Verilog del multiplexer (modulo "m").



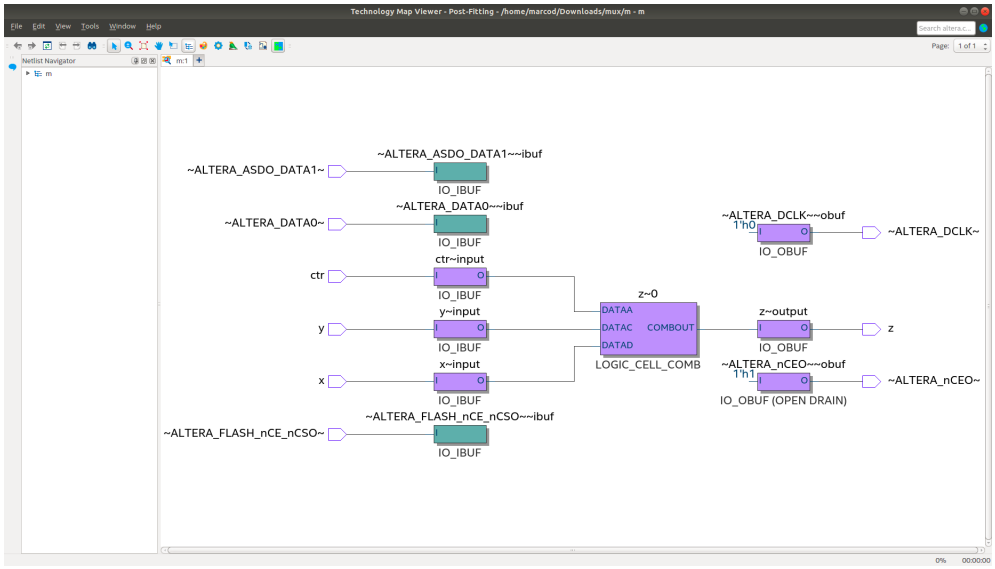
Selezionando dal menu "Processing" l'entry "Start compilation", dopo ca un minuto su un laptop con i5 dual core otteniamo quanto mostrato nella prossima snapshot:



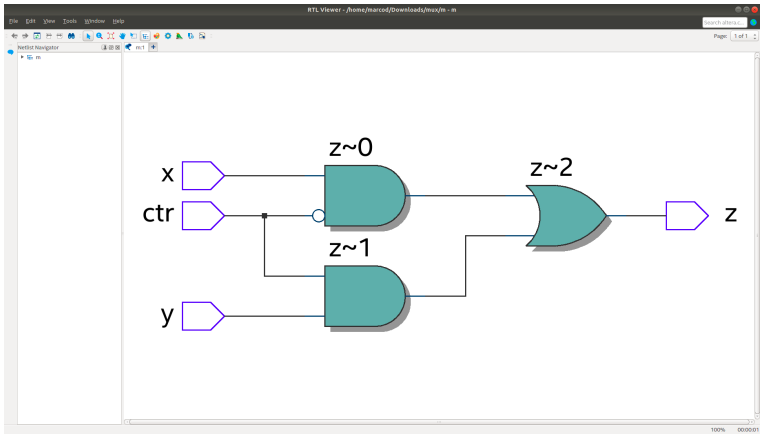
Adesso possiamo vedere cosa è stato generato come prodotto della sintesi, andando a scegliere dal menu "Tools" entry "Netlist" subentry "RTL":



I tool di sintesi hanno riconosciuto il multiplexer e lo hanno implementato come tale (1 cella che esegue il routing a seconda dell'ingresso di controllo). Se chiediamo invece che la vista RTL quella delle celle otteniamo però la seguente cosa:



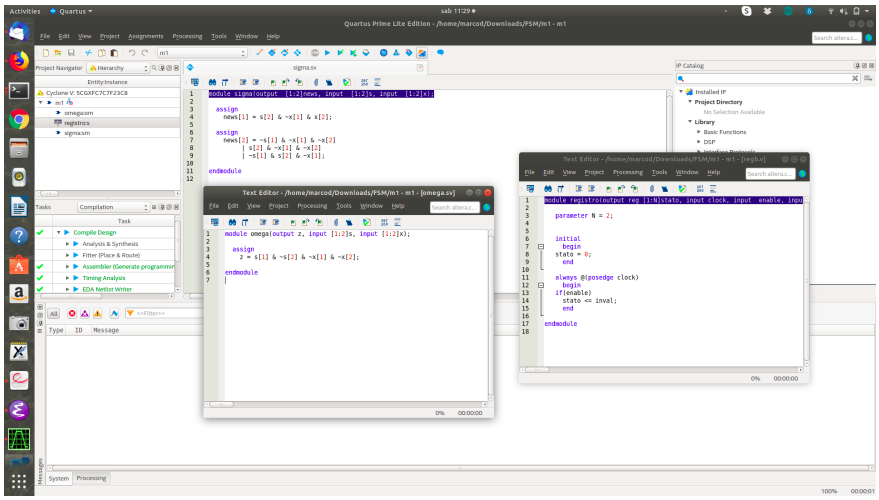
dove si vede chiaramente come il multiplexer sia stato ottenuto programmando una cella come una cella che implementa una rete logica da tre ingressi e un'uscita (la `LOGIC_COMB_CELL` al centro) al netto dei vari buffer utilizzati per pilotare i segnali. Se però il multiplexer lo descriviamo come espressione booleana (sostituiamo la assign del codice precedente con una che esegue `assign z = !ctr&x | ctr&y;`), otteniamo la netlist RTL:



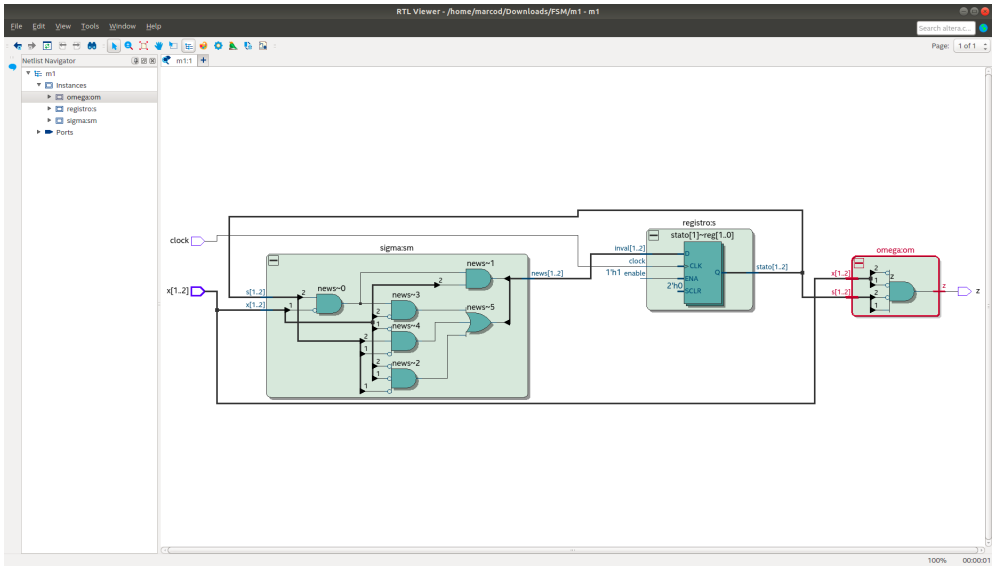
che corrisponde allo stesso tipo di utilizzo delle celle FPGA della figura con la `LOGIC_COMB_CELL` vista poco fa.

5.3.1 Sintesi di una rete sequenziale (modello strutturale)

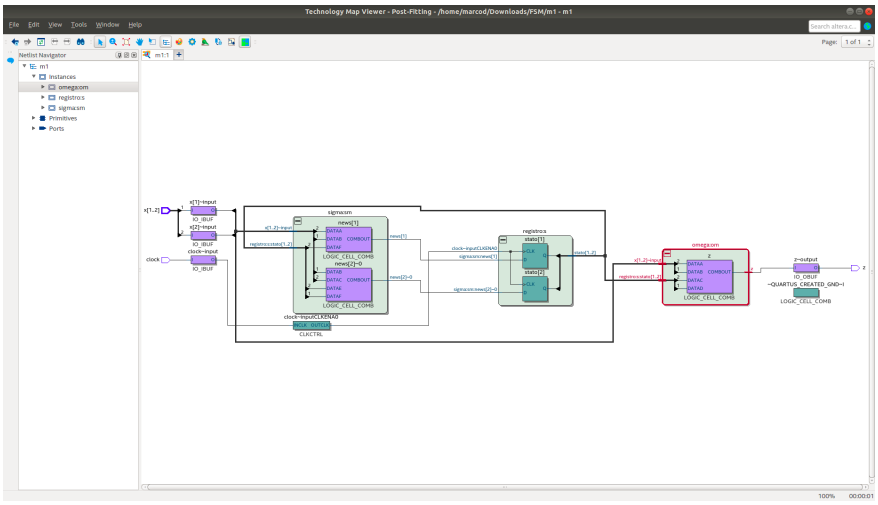
Se consideriamo qualcosa di più complesso, come per esempio la rete sequenziale che riconosce le stringhe "abba" sull'alfabeto `{a,b,c}` il processo di sintesi fornisce qualcosa di più significativo. In questo caso nel progetto includiamo i file per sigma, omega, e registro:



avviando la compilazione otteniamo:



cui corrisponde la configurazione di celle della FPGA:



Se invece di utilizzare il codice relativo al modello strutturale della rete sequenziale utilizzassimo il codice behavioural:

Text Editor - /home/marcod/Downloads/FSM/m1 - m1 - [m1.sv]

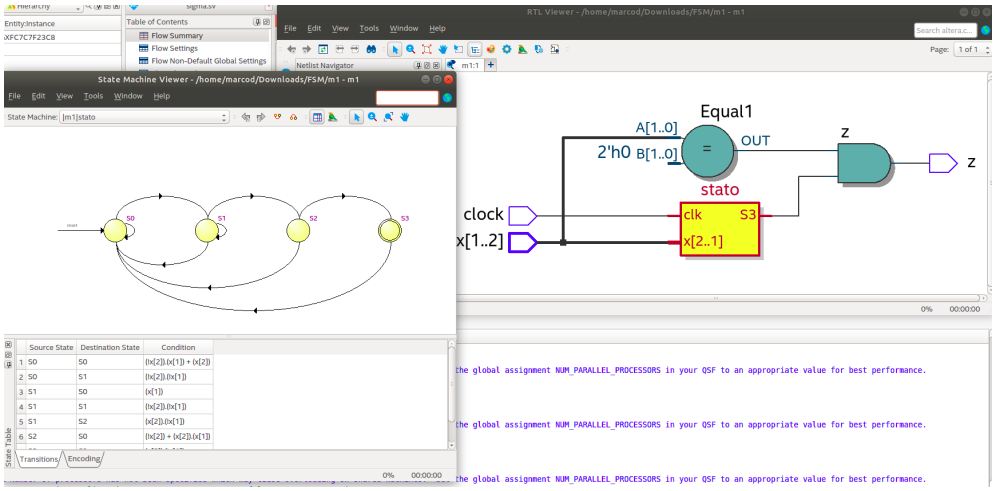
File Edit View Project Processing Tools Window Help

Search altera.c...

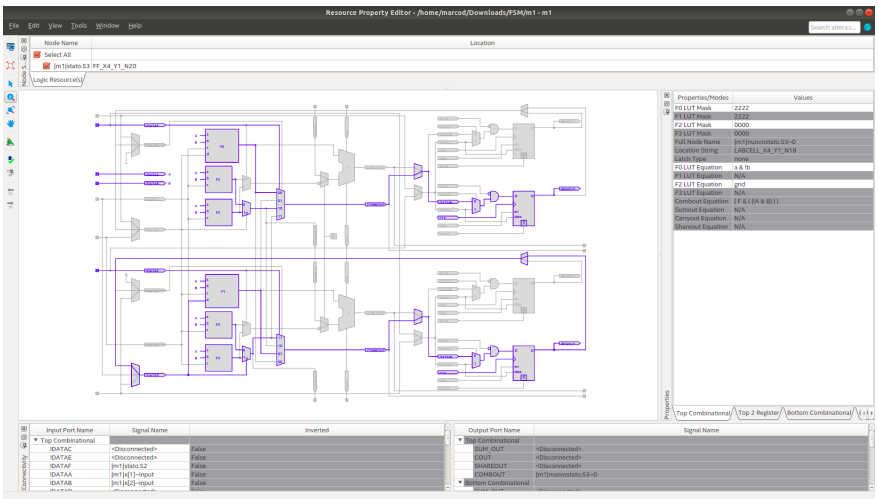
```
1  module m1(output z, input [1:2]x, input clock);
2
3     reg [1:2] stato;
4     reg [1:2] nuovostato;
5
6     parameter S0=2'b00;
7     parameter S1=2'b01;
8     parameter S2=2'b11;
9     parameter S3=2'b10;
10    parameter A=2'b00;
11    parameter B=2'b01;
12    parameter C=2'b11;
13
14    initial
15      stato = S0;
16
17    always @(posedge clock)
18       stato <= nuovostato;
19
20    always @(*)
21      begin
22    case(stato)
23      S0: nuovostato = (x == A ? S1 : S0);
24      S1: nuovostato = (x == B ? S2 : (x == A ? S1 : S0));
25      S2: nuovostato = (x == B ? S3 : (x == B ? S1 : S0));
26      S3: nuovostato = S0;
27    endcase // case (stato)
28      end
29
30    assign
31      z = ((stato == S3 && x == A) ? 1 : 0);
32
33 endmodule
34
```

100% 00:00:01

allora il risultato della sintesi sarebbe nettamente diverso:



Vedete che la sintesi ha prodotto un minimo di rete combinatoria per il calcolo dell'uscita e una macchina a stati finiti (la parte gialla, di cui la vista ad automa (finestra a sinistra) si ottiene cliccandoci sopra due volte. Questo corrisponde (vista chip planner) alla mappatura della figura che segue sulle celle della FPGA. Facciamo solo notare che ci sono delle ALU grigie (la parte grigia è quella che non viene utilizzata) che rappresentano la colonna di macro celle di cui abbiamo parlato nella sezione 5.1.



Capitolo 6

Installazione tool e manuali online

I tool che utilizziamo sono tutti Open Source e possono essere utilizzati sia sotto Linux che sotto Windows e Mac OS X.

Gli esempi e i dump video di queste note sono tutti stati testati utilizzando `Icarus iverilog` e `GTKWave`. Per Linux sono disponibili pacchetti che si possono istallare con un comando

```
apt install iverilog gtkwave
```

eseguito con i diritti di superutente. Windows e MAC OS/X necessitano di procedure di installazione leggermente diverse, comunque documentate sul sito dei tool:

- `http://iverilog.icarus.com/` per `iverilog`
- `http://gtkwave.sourceforge.net/` per `gtkwave`

6.1 Installazione Linux (UBUNTU)

Per installare iverilog:

- sudo apt install iverilog

Per installare gtkwave:

- sudo apt install gtkwave

6.2 Utilizzazione Linux

6.2.1 Compilazione

Un modulo Verilog `mod.v` e il relativo programma di test `test.v` possono essere compilati con il comando:

```
iverilog mod.v test.v -o eseguibile
```

In generale, dopo il nome del compilatore si debbono mettere i nomi di tutti i moduli necessari ad implementare l'unità, incluso il programma di test. Qualora mancasse un modulo, si ottiene un messaggio di errore tipo:

```
1 marcod@marcod-ThinkPad-E480:~/Didattica/Architettura/NuovoCorso/Esercizi/FSM/ABBA$ iverilog
      test-m1.v  m1.v regb.v sigma.v
2 m1.v:8: error: Unknown module type: omega
3 2 error(s) during elaboration.
4 *** These modules were missing:
5         omega referenced 1 times.
6 ***
7 marcod@marcod-ThinkPad-E480:~/Didattica/Architettura/NuovoCorso/Esercizi/FSM/ABBA$
```

E' evidente che ci siamo dimenticati del modulo che implementa omega.

6.2.2 Esecuzione della simulazione

Qualora si siano indicati tutti i file necessari, viene creato un eseguibile che, nel caso non ne sia stato specificato il nome utilizzando un'opzione `-o <nomefileeseguibile>`, sarà il file `a.out`. Il file eseguibile in realtà contiene codice da eseguire mediante l'interprete `vvp`. Le prime righe del file saranno qualcosa tipo:

```
marcod@marcod-ThinkPad-E480:~/Didattica/Architettura/NuovoCorso/Esercizi/FSM/ABBA$ head a.
    out
#! /usr/bin/vvp
:ivl_version "10.1 (stable)";
:ivl_delay_selection "TYPICAL";
:vpi_time_precision + 0;
:vpi_module "system";
:vpi_module "vhdl_sys";
:vpi_module "v2005_math";
:vpi_module "va_math";
S_0x5625aa43da70 .scope module, "testm1" "testm1" 2 1;
 .timescale 0 0;
marcod@marcod-ThinkPad-E480:~/Didattica/Architettura/NuovoCorso/Esercizi/FSM/ABBA$
```

Come si vede, la prima riga dice che il programma va eseguito utilizzando `/usr/bin/vvp`. Dunque il file può essere eseguito semplicemente dando il comando:

```
./a.out
```

oppure richiamando direttamente l'interprete con il comando:

```
vvp a.out
```

Se il programma di test genera le tracce di esecuzione con un `$dumpfile("file.vcd")`, i risultati possono essere analizzati con il comando

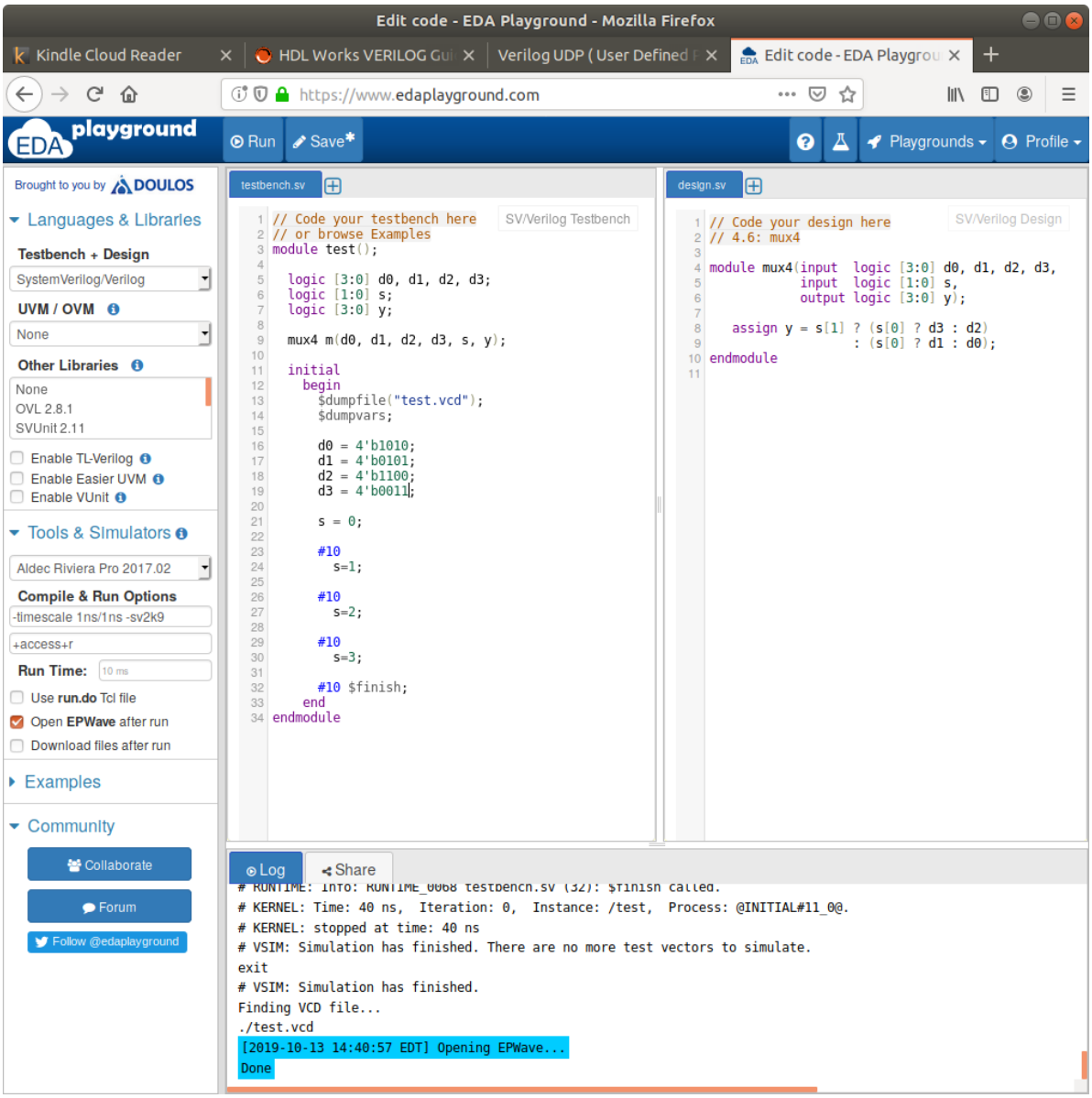
- `gtkwave file.vcd`

6.3 Strumenti online

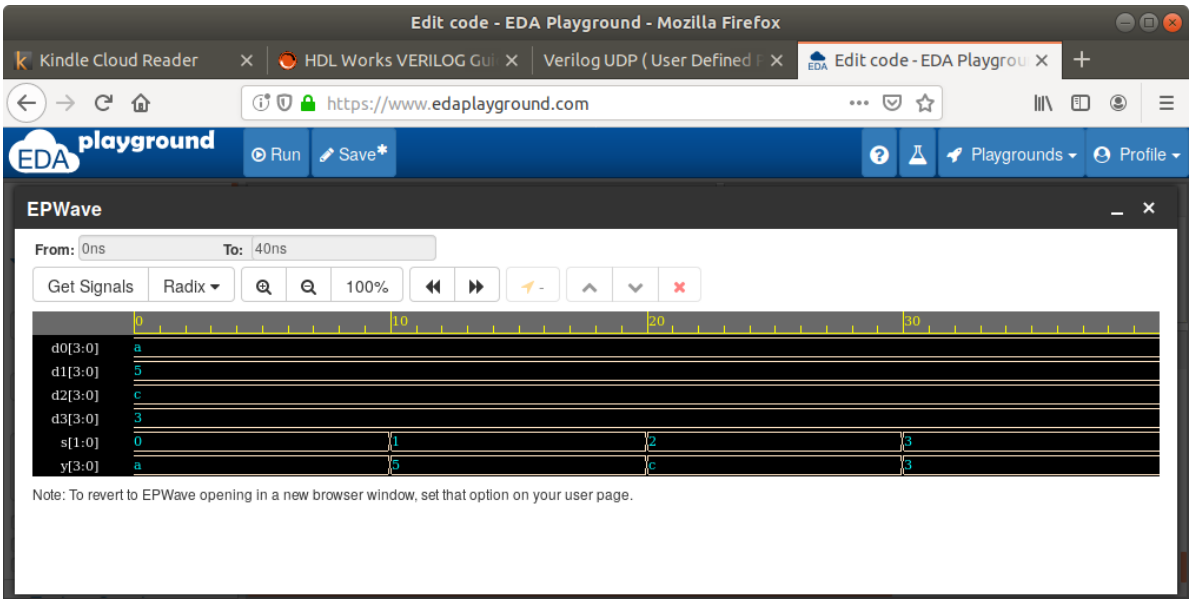
Infine, per sperimentare semplici esercizi Verilog o System Verilog, si può utilizzare il sito online

```
https://www.edaplayground.com/
```

Per esempio, potremmo simulare il comportamento del codice di `mux4` come descritto nel libro di testo (System Verilog) utilizzando il codice in figura:



Se selezioniamo “Open EPwave after run” sulla sinistra e, per esempio, il compilatore simulatore della Aldec Riviera, otteniamo una cosa tipo:



che ci permette di verificare il comportamento del modulo sotto test.

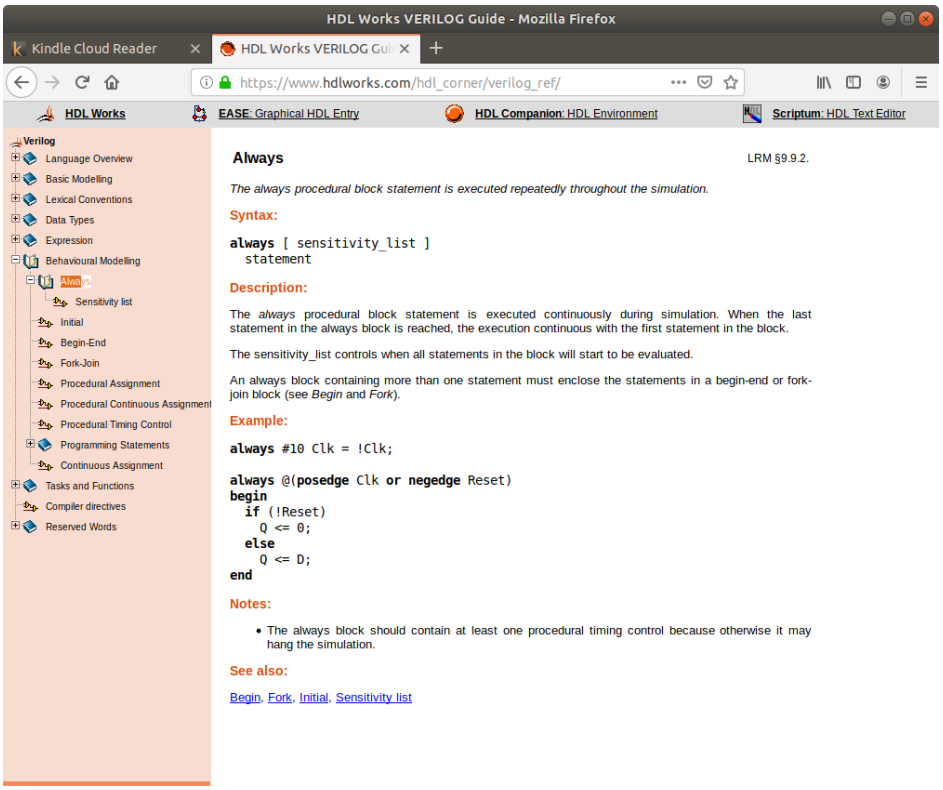
6.4 Materiale di consultazione

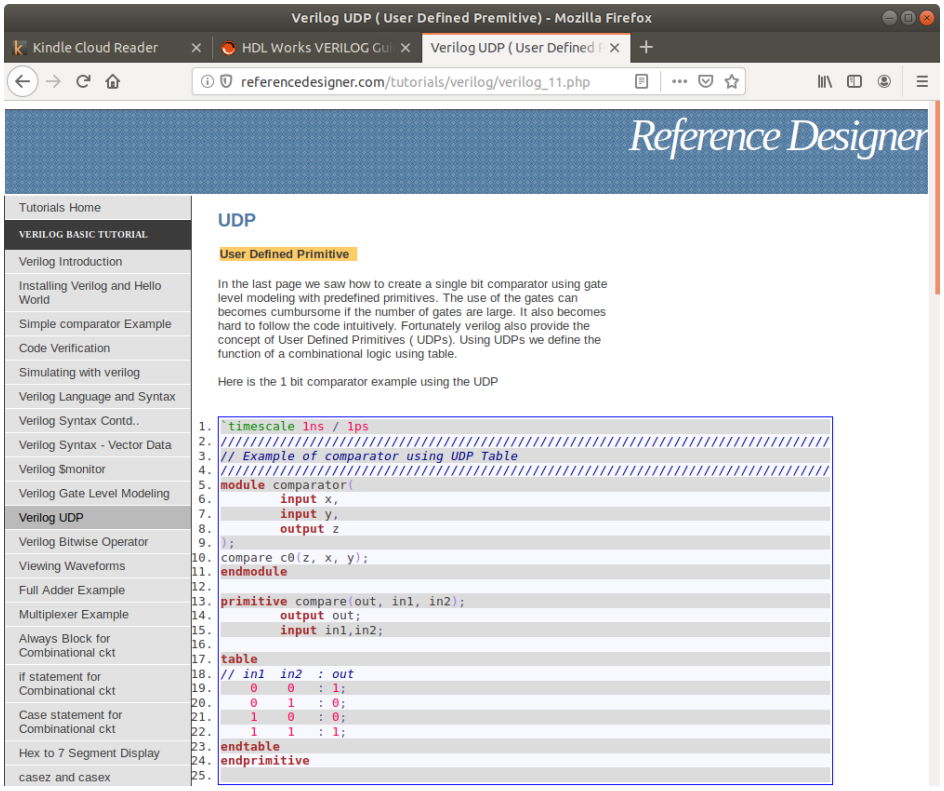
Un buon sito che descrive correttamente e in maniera concisa la sintassi del Verilog è quello che si trova all'indirizzo:

`https://www.hdlworks.com/hdl_corner/verilog_ref/`

Potrebbe essere utile come materiale di consultazione anche il sito web al link:

`http://referencedesigner.com/tutorials/verilog/verilog_01.php`




Verilog UDP (User Defined Premitive) - Mozilla Firefox
Kindle Cloud Reader
HDL Works VERILOG Gui
Verilog UDP (User Defined F
referencedesigner.com/tutorials/verilog/verilog_11.php
Reference Designer
Tutorials Home
VERILOG BASIC TUTORIAL
Verilog Introduction
Installing Verilog and Hello World
Simple comparator Example
Code Verification
Simulating with verilog
Verilog Language and Syntax
Verilog Syntax Contd..
Verilog Syntax - Vector Data
Verilog $monitor
Verilog Gate Level Modeling
Verilog UDP
Verilog Bitwise Operator
Viewing Waveforms
Full Adder Example
Multiplexer Example
Always Block for Combinational ckt
if statement for Combinational ckt
Case statement for Combinational ckt
Hex to 7 Segment Display
casez and casex
UDP
User Defined Primitive
In the last page we saw how to create a single bit comparator using gate level modeling with predefined primitives. The use of the gates can becomes cumbursome if the number of gates are large. It also becomes hard to follow the code intuitively. Fortunately verilog also provide the concept of User Defined Primitives (UDPs). Using UDPs we define the function of a combinational logic using table.
Here is the 1 bit comparator example using the UDP
1. `timescale 1ns / 1ps
2. //
3. // Example of comparator using UDP Table
4. //
5. module comparator(
6. input x,
7. input y,
8. output z
9.);
10. compare c0(z, x, y);
11. endmodule
12.
13. primitive compare(out, in1, in2);
14. output out;
15. input in1,in2;
16.
17. table
18. // in1 in2 : out
19. 0 0 : 1;
20. 0 1 : 0;
21. 1 0 : 0;
22. 1 1 : 1;
23. endtable
24. endprimitive
25.

Bibliografia

[1] D. M. Harris & S. L. Harris, Digital Design and Computer Architecture, Morgan Kaufmann, 2007, Capitolo 4 "Hardware Description Languages".

[2] D. M. Harris & S. L. Harris, Digital Design and Computer Architecture: ARM edition, Morgan Kaufmann, 2017.

[3] Peter M. Nyasulu, Introduction to Verilog, 2001, http://www.csd.uoc.gr/~hy220/2008f/lectures/verilog-notes/VerilogIntroduction.pdf

[4] Doulos, The Verilog Golden Reference Guide, 1996, www.fpga.com.cn/hdl/training/verilog\%20reference\%20guide.pdf

[5] Deepak Kumar Tala, Verilog Tutorial, 2003, www.ece.ucsb.edu/courses/ECE152/.../VerilogTutorial.pdf

[6] E. Madhavan, Quick reference for Verilog HDL, 1995, www.stanford.edu/class/ee183/handouts.../VerilogQuickRef.pdf

[7] S. A. Edardw, *The Verilog language*, slides Columbia University, 2002, http://www.cs.columbia.edu/~sedwards/classes/2002/w4995-02/verilog.9up.pdf