Note sull'utilizzo di Verilog per la prima parte del corso di Architettura degli Elaboratori

M. Danelutto

Corso di Laurea in Informatica – Universitá di Pisa

A.A.2010–2011

Lo scopo di queste note é quello di mettere a disposizione il minimo indispensabile per poter utilizzare Verilog per la simulazione del funzionamento dei componenti utilizzati nel corso di Architetture degli elaboratori presso il Corso di laurea in Informatica dell'Universitá di Pisa. Gli strumenti discussi, e soprattutto gli esempi presentati in queste note, dovrebbero permettere un approccio guidato all'uso del linguaggio Verilog per la simulazione di componenti architetturali relativamente semplici. Asintoticamente, dovrebbero permettere la modellazione dell'intera architettura D-RISC introdotta nel libro di testo del corso.

Queste note *non* rappresentano un corso di Verilog, tuttavia. Il linguaggio é introdotto solo sommariamente nella sezione 2. Si rimanda alla consistente letteratura sull'argomento quando si senta la necessitá di approfondire un argomento, di aver un chiarimento su un concetto o comunque di sperimentare caratteristiche non utilizzate per gli esempi trattati in queste note e di conseguenza non spiegate o spiegate in maniera poco approfondita.

Indice

Elenco delle figure

Capitolo 1

Installazione

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 Verilog version 0.9.2` e `GTKWave Analyzer v3.3.2` sotto Mac OS X 10.6 e/o Ubuntu 10.04.

1.1 Icarus Verilog

I sorgenti di Icarus Verilog si trovano sul sito FTP `ftp://ftp.icarus.com/pub/eda/verilog/v0.9`. Per il download consiglio di procedere da terminale. Di seguito il dump della sessione che permette di scaricare i sorgenti da ricompilare.

```
Marco-Daneluttos-MacBook-Pro:Componenti marcodanelutto$  ftp ftp://ftp.icarus.com
Connected to icarus.icarus.com.
220 "Welcome to icarus.com FTP service."
331 Please specify the password.
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
200 Switching to Binary mode.
ftp> cd pub
250 Directory successfully changed.
ftp> cd eda
250 Directory successfully changed.
ftp> cd verilog
250 Directory successfully changed.
ftp> cd v0.9
250 Directory successfully changed.
ftp> ls
229 Entering Extended Passive Mode (|||56755|)
150 Here comes the directory listing.
drwxr-sr-x    2 1045     49            104 Jul 15  2009 OpenSolaris
drwxr-sr-x    2 1045     49            144 Dec 30 16:55 SuSE-10.3
-rw-r--r--    1 1045     100       1117412 Mar 20  2009 verilog-0.9.1-0.src.rpm
-rw-r--r--    1 1045     100       1113403 Mar 20  2009 verilog-0.9.1.tar.gz
-rw-r--r--    1 1045     49            192 Mar 20  2009 verilog-0.9.1.txt
-rw-r--r--    1 1045     100       1131073 Dec 30 16:53 verilog-0.9.2-0.src.rpm
-rw-r--r--    1 1045     49        1127067 Dec 30 16:54 verilog-0.9.2.tar.gz
```

```
226 Directory send OK.
ftp> binary
200 Switching to Binary mode.
ftp> get verilog-0.9.2.tar.gz
local: verilog-0.9.2.tar.gz remote: verilog-0.9.2.tar.gz
229 Entering Extended Passive Mode (|||46439|)
150 Opening BINARY mode data connection for verilog-0.9.2.tar.gz (1127067 bytes).
100\% |***********************************|  1100 KiB   97.25 KiB/s    00:00 ETA
226 File send OK.
1127067 bytes received in 00:11 (95.21 KiB/s)
ftp> quit
221 Goodbye.
Marco-Daneluttos-MacBook-Pro:Componenti marcodanelutto$
```

La compilazione e l'installazione del tool avvengono con la classica sequenza di comandi[1]:

```
./configure --prefix=<directory-di-installazione>
make
make check
make install
```

Se non si indica la directory di installazione (utilizzando quindi un semplice comando `./configure`) la directory di installazione é `/usr/local` che richiede di solito i diritti da super utente per effettuarvi modifiche quali quelle richieste nel processo di installazione.

Al termine del processo di installazione, ricordatevi di fare in modo di avere la `bin` dove é avvenuta l'installazione nel `PATH` dei comandi. Sotto `bash` potete utilizzare il comando:

```
export PATH=$PATH:/usr/local/bin
```

se la directory di installazione é quella di default. Sotto `csh` o `tcsh` invece dovreste utilizzare i comandi:

```
set PATH $PATH:/usr/local/bin; export $PATH
```

(Ricordatevi anche per rendere permanenti questi path occorre installare questi comandi nei file di configurazione della vostra shell: `.profile` per la `bash` o `.cshrc` per le altre due shell)

Se tutto é andato a buon fine dovreste riuscire ad invocare da prompt della shell i comandi `iverilog` (compilatore) `vvp` (simulatore) e `dinotrace` (visualizzatore).

```
Marco-Daneluttos-MacBook-Pro:verilog-0.9.2 marcodanelutto$ iverilog
iverilog: no source files.

Usage: iverilog [-ESvV] [-B base] [-c cmdfile|-f cmdfile]
                [-g1995|-g2001|-g2005] [-g<feature>]
                [-D macro[=defn]] [-I includedir] [-M depfile] [-m module]
                [-N file] [-o filename] [-p flag=value]
                [-s topmodule] [-t target] [-T min|typ|max]
                [-W class] [-y dir] [-Y suf] source_file(s)

See the man page for details.
Marco-Daneluttos-MacBook-Pro:verilog-0.9.2 marcodanelutto$ vvp
vvp: no input file.
Marco-Daneluttos-MacBook-Pro:verilog-0.9.2 marcodanelutto$ dinotrace
Marco-Daneluttos-MacBook-Pro:verilog-0.9.2 marcodanelutto$
```

[1] da dare dopo aver scompattato i sorgenti ed aver eseguito un cambio di directory nella root dei sorgenti

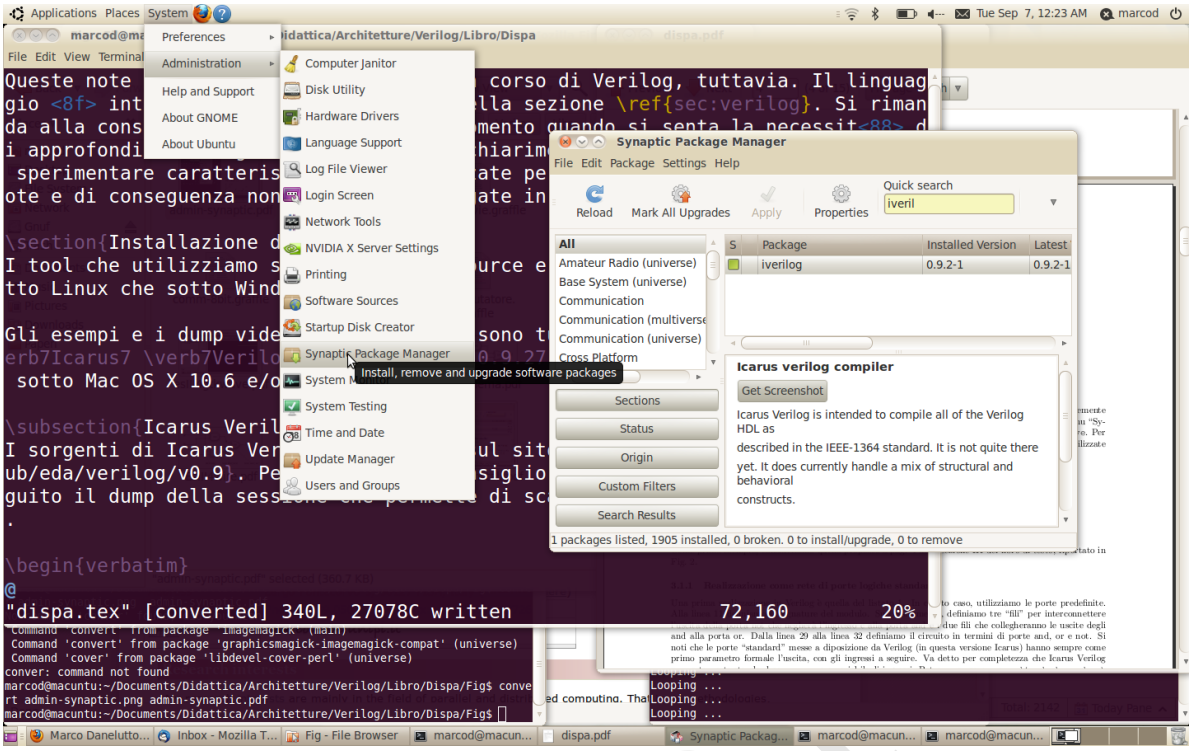

Figura 1.1: Installazione con Ubuntu e Synaptic Package Manager. Si selezione il menu System scegliendo poi la voce Admin (in alto a sinistra. La finestra con il package iverilog é visibile sulla destra.

1.2 GtkWave

I sorgenti di GtkWave si trovano sotto Sourceforge all'indirizzo `http://gtkwave.sourceforge.net/`.

Come nel caso precendente, la compilazione e installazione avvengono con i soliti comandi `./configure`, `make` e `make install`. Tuttavia, il tool ha molte dipendenze relative a pacchetti grafici e non. Consiglio chiaramente di utilizzare un pacchetto giá pronto (rpm, deb, etc.) quando disponibile e di effettuare l'installazione del visualizzatore utilizzando i tool di installazione propri della versione di sistema operativo che state utilizzando: `yum` o `rpm` per Linux, per esempio, `mac ports` per Mac Os X, etc.

Dal momento che l'installazione del `gtkwave` é di fatto opzionale (Icarus ha il proprio visualizzatore), non ci dilunghiamo ulteriormente sulle modalitá (e peculiaritá) di installazione di questo pacchetto.

1.3 Dinotrace

Il visualizzatore `dinotrace` é incluso di default in alcune delle distribuzioni di `iverilog` (ad esempio in quella per Mac OS X). Qualora non lo fosse, si puó scaricare da web (`http://www.veripool.org/projects/dinotrace/wiki/Installing`) e installare con la solita procedura 1) `./configure`, 2) `make`, 3) `make install`. Il pacchetto richiede la presenza di Motif per X11. Qualora non fosse installata di default si puó procedere all'installazione di `LessTif` *prima* di procedere con l'installazione di `dinotrace`. Dal momento che sono richiesti i file header per Motif per la compilazione di `dinotrace`, occorre procedere con l'installazione dei pacchetti `dev` che normalmente contengono anche gli header file (il pacchetto `LessTif2` non li contiene).

1.4 Installazione sotto Linux Ubuntu

Se si utilizza un sistema Linux Ubuntu, icarus e GTKwave possono essere installati molto piú semplicemente utilizzando il Synaptic Package Manager (Fig. 1.1). Selezionando il Synaptic Package Manager dal menu "System" → "Administration", si puó selezionare e installare sia il package `iverlog` che il package `gtkwave`. Per completare l'installazione, prima selezionate i package ("mark for installation") e successivamente utilizzate il bottone "Apply" che lancerá l'installazione vera e propria.

1.5 Installazione sotto Windows

Per installare `iverilog` sotto Windows, si puó utilizzare il pacchetto installatore reperibile al sito `http://bleyer.org/icarus/`. L'installazione del pacchetto avviene secondo la normale procedura sei `setup\` windows. Al termine dell'installazione, sia il compilatore che il simulatore ed il visualizzatore `gtkwave` vengono installati sotto la directory bin del path utilizzato per l'installazione. **Attenzione**: come specificato nei messaggi di installazione, il path utilizzato per l'installazione **non** deve contenere spazi!

L'utilizzo di `iverilog`, `vvp` e `gtkwave` avviene da prompt della riga di comando come nel caso di Mac OS X o Linux. Aprite una shell DOS dal menu `programmi -> Accessori -> Prompt Comandi`. Se al prompt, dando uno dei tre comandi non ottenete che un messaggio di errore significa che non avete settato il `PATH` correttamnete.

Capitolo 2

Verilog

Esistono diversi testi, manuali e presentazioni che trattano di Verilog. In bibliografia, riportiamo sostanzialmente due cose:

- il libro di testo [1], che é a mio avviso un buon testo di architetture degli elaboratori e che contiene un intero capitolo sui linguaggi per la descrizione/modellazione dell'hardware. In particolare il capitolo tratta Verilog e VHDL con un buon numero di esempi, tutti presentati sia nella versione Verilog che nella versione VHDL. Il libro presenta una semplice implementazione di un processore MIPS e ne da' la versione Verilog e VHDL. Vale la pena dargli un'occhiata;

- un certo numero di PDF reperibili via WEB, che contengono introduzioni [2, 3], tutorial [4, 6] e quick reference [5]. Sono da ritenere materiale da consultazione, anche se l'introduzione contenuta in [2] da' un'idea abbastanza ragionevole delle possibilitá offerte dal linguaggio.

Qualora l'esperimento dell'A.A. 2010–2011 (introduzione del Verilog come "mezzo" per la realizzazione di semplici esercizi sulla prima parte del corso di Architettura degli Elaboratori) avesse successo e si decidesse di reiterarlo nei successivi anni accademici, provvederemo a scrivere un capitolo "Verilog" completo.

Al momento, rimandiamo il lettore ai riferimenti presentati in bibliografia, e in particolare al libro degli Harris [1].

Capitolo 3

Reti combinatorie

3.1 Esempio di pagina III.4

Come primo esempio, consideriamo quello proposto a pag. 4 della sezione III del libro di testo, riportato in Fig. 3.1.

3.1.1 Realizzazione come rete di porte logiche standard

Una prima realizzazione in Verilog é quella del listato 3.1. In questo caso, utilizziamo le porte predefinite. Alla linea 11 definiamo la signature del modulo. Successivamente, definiamo tre "fili" per interconnettere l'uscita della porta not che negherá l'ingresso c alla porta and e i due fili che collegheranno le uscite degli and alla porta or. Dalla linea 29 alla linea 32 definiamo il circuito in termini di porte and, or e not. Si noti che le porte "standard" messe a diposizione da Verilog (in questa versione Icarus) hanno sempre come primo parametro formale l'uscita, con gli ingressi a seguire. Va detto per completezza che Icarus Verilog supporta porte standard con un numero variabile di ingressi. Potremmo usare una `and(and_abc,a,b,c)` per instanziare un'and a tre ingressi, per esempio.

Listing 3.1: EsIII4 (utilizzando rete di porte logiche)

```
1 //
2 // Questa    la rete dell'esempio a pagina III.4 del libro.
3 //
4 // definisco il modulo
5 // il primo parametro    quello di output, per convenzione
6 // gli altri rappresentano gli input
7 // (non obbligatorio, ma la convenzione e' quella utilizzata anche per le
8 //  porte predefinite (come la and, or, not, etc.) )
9 //
10
```

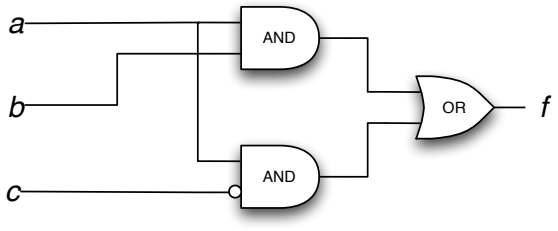

Figura 3.1: Esempio di rete combinatoria (dalla pagina III.4 del libro di testo)

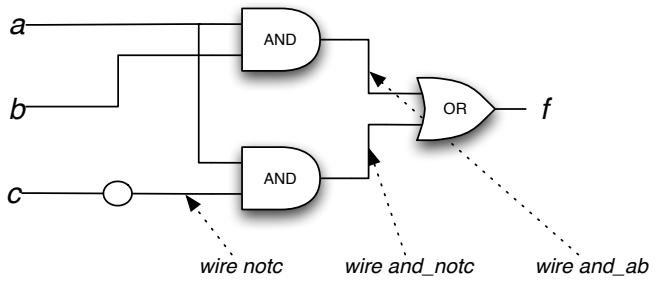

Figura 3.2: Rete EsIII4 con evidenziati i collegamenti fra le porte dichiarati nel Listing 3.1

```
11 module esIII4(output wire f,
12   input wire a, input wire b, input wire c);
13
14    // definizione dei "fili" che interconnettono le porte
15    // all'interno della rete. I "fili" che interfacciano la rete
16    // con il mondo esterno (input e output) sono quelli definiti
17    // nella lista di parametri del modulo
18
19    wire and_ab;
20    wire notc;
21    wire and_anotc;
22
23    //
24    // definisco la struttura della rete, utilizzando le porte predefinite
25    // e i fili che realizzano le interconnessioni
26    // Le porte predefinite hanno sempre il primo parametro che rappresenta
27    // l'output e gli altri che rappresentano l'input
28    //
29    and(and_ab, a, b);
30    not(notc, c);
31    and(and_anotc, a, notc);
32    or(f,and_ab,and_anotc);
33
34 // fine del modulo
35 endmodule
```

Per testare il funzionamento della rete combinatoria modellata dal modulo Verilog esIII4, prepariamo un programma di test (Listing 3.2). Per prima cosa, all'esterno della definizione del modulo, definiamo la scala dei tempi da utilizzare (nanosecondi, divisi in picosecondi, linea 4. Attenzione, il carattere che precede la `timescale` *deve* essere il backquote `, che su alcune tastiere (per esempio quella del Mac) richiede una sequenza particolare di tasti per essere immesso (per esempio sul Mac occorre un option-9 seguita da uno spazio).

Successivamente, definiamo i registri che modelleranno i nostri ingressi. Non essendo specificata la dimensione dei registri, essi verranno compilati come registri da 1 bit (dichiarazioni alle linee 12–14). Se avessimo voluto dichiarare un registro da 2 bit, per esempio, avremmo dovuto utilizzare il codice `reg [1:0]a;` oppure `reg [0:1]a;`. Nel primo caso il registro sarebbe stato definito in modo che il bit piú significativo si possa accedere con `a[1]` mentre nel secondo caso l'"ordinamento" dei bit sarebbe stato tale da avere il bit piú significativo in `a[0]` (come avviene normalmente negli esempi del libro di testo).

Dopo aver definito i registri che ci servono per gli ingressi, definiamo il filo che porterá l'uscita della rete combinatoria (linea 19) e successivamente instanziamo la rete stessa (linea 28). L'istanziazione della rete avviene nominando il nome del modulo seguito (dopo uno spazio) dal nome dell'istanza e dai parametri formali. Il nome del modulo potrá essere utilizzato per accedere alle variabili interne, per esempio, o nella visualizzazione delle tracce di output per distinguere tracce con lo stesso nome.

Dopo aver definito la struttura della rete di test (input, output, rete da testare), in una clausola `initial` definiamo dove memorizzare i risultati della simulazione (linee 36–39: definizione del nome del file che conterrá la traccia dei cambiamenti delle variabili del nostro programma nel tempo (linea 37) e definizione di quali variabili monitorare (linea 39, in questo caso tutte, non avendo specificato il nome del modulo da monitorare)), i valori iniziali degli ingressi (linee 42–44) e la variazione degli stessi ingressi nel tempo (linee 49–56). La gratella seguita da un numero indica il ritardo (nell'unitá di tempo definita dalla `timescale`) dopo il quale lo statement successivo verrá eseguito. Nel nostro caso, a 10 nsec dall'assegnamento iniziale delle variabili metteremo a ad 1, poi dopo altri 10 nsec metteremo a a 0 e b a 1, dopo altri 10 nsec a a 1 e c a 1, etc.

Lo statement (comando di sistema) alla riga 60 dice di terminare la simulazione, dopo 10 nsec dall'ultimo comando.

Listing 3.2: Programma di test per EsIII4

```
1 //
2 // definizione di comodo per stabilire il range temporale
3 //
4 `timescale 1ns / 1ps
5
6 // nome del modulo; senza parametri perch    quello di test
7 module provaEsIII4();
8
9   //
10  // registri che conterranno i valori di input alla rete da testare
11  //
12  reg a;
13  reg b;
14  reg c;
15
16  //
17  // filo per l'output della rete da testare (potrebbe anche essere reg)
18  //
19  wire f;
20
21  //
22  // istanziazione della rete (non    una chiamata di procedura!)
23  // i parametri attuali vengono collegati come specificato mediante
24  // i parametri formali nella definizione del modulo esIII4
25  // prima delle parentesi tonde dei parametri troviamo il nome del modulo
26  // istanziato (serve per distinguere fra istanziazioni diverse)
27  //
28  esIII4 esEsIII4(f,a,b,c);
29
30  //
31  // statement da eseguire all'inizio della simulazione
32  //
33  initial
34  begin
35
36    // file di ouput per la visualizzazione del comportamento
37    $dumpfile("provaEsIII4.vcd");
38    // esegui il dump dei cambiamenti di tutte le variabili in gioco
39    $dumpvars;
40
41    // assegnamento iniziale dei registri di input alla rete
42    a = 0;
43    b = 0;
44    c = 0;
45
46    // simulazione di cambi della configurazione in ingresso
47    //
48
49    // dopo 10 nsec cambia valore di a
50    #10 a = 1;
```

```
51      // dopo altri 10 nsec cambia sia a che b
52      #10 a = 0;
53          b = 1;
54      #10 a = 1;
55          c = 1;
56      #10 b = 0;
57
58      // attendi altri 10 nsec e termina la simulazione
59      // (producendo il .vcd con il dump dei cambiamenti delle variabili)
60      #10 $finish;
61
62    // fine del (blocco di) comando da eseguire all'inizio della simulazione
63    end
64
65  // fine del modulo di test
66  endmodule
```

Per "vedere" il risultato della nostra rete combinatoria a seguito degli input codificati nel programma Verilog, eseguiamo i seguenti 3 passi:

i) compiliamo il programma utilizzando il compilatore `iverilog` con il comando

```
iverilog provaEsIII4.vl EsIII4.vl
```

(attenzione: vanno nominati come file di ingressi sia quello che contiene il modulo di test che quello che contiene la definizione del modulo `esIII4`. Questo comando, non avendo specificato mediante un flag `-o xxx` il nome del file dell'eseguibile, lascerá l'eseguibile in `a.out`, come avviene di solito con i compilatori C/C++.

ii) eseguiamo la simulazione facendo girare il programma compilato con l'interprete, mediante il comando `vvp a.out`

iii) lanciamo l'interfaccia grafica che ci permette di visualizzare il file `.vcd` che contiene la traccia dei valori delle variabili nella simulazione appena eseguita. Qui possiamo usare due visualizzatori: `dinotrace`, che é contenuto nella distribuzione (sorgente) di Icarus Verilog, o `gtkwave`, che é un po' piú carino e sofisticato, ma che richiede un'installazione a parte. In Fig. 3.3 vediamo come si presenta la traccia visualizzata con `dinotrace`, mentre in Fig. 3.4 vediamo come la stessa traccia viene rappresentata mediante `gtkwave`. Si noti come con `dinotrace` sono evidenziati i nomi delle variabili relativamente al modulo cui esse appartengono, mentre `gtkwave` riporta un frame per la navigazione dei moduli (in alto a sinistra) ma non mette il nome del modulo accanto al nome della variabile.

Listing 3.3: EsIII4 (utilizzando il behavioural mode di Verilog)

```
1  //
2  // Questa    la rete dell'esempio a pagina III.4 del libro.
3  //
4  // definisco il modulo
5  // il primo parametro    quello di output, per convenzione
6  // gli altri rappresentano gli input
7  // (non obbligatorio, ma la convenzione e' quella utilizzata anche per le
8  //  porte predefinite (come la and, or, not, etc.) )
9  //
10
11 module esIII4(output wire f,
12   input wire a, input wire b, input wire c);
13
14   assign f = (a & b) | ( a & (~c));
15
16 endmodule
```

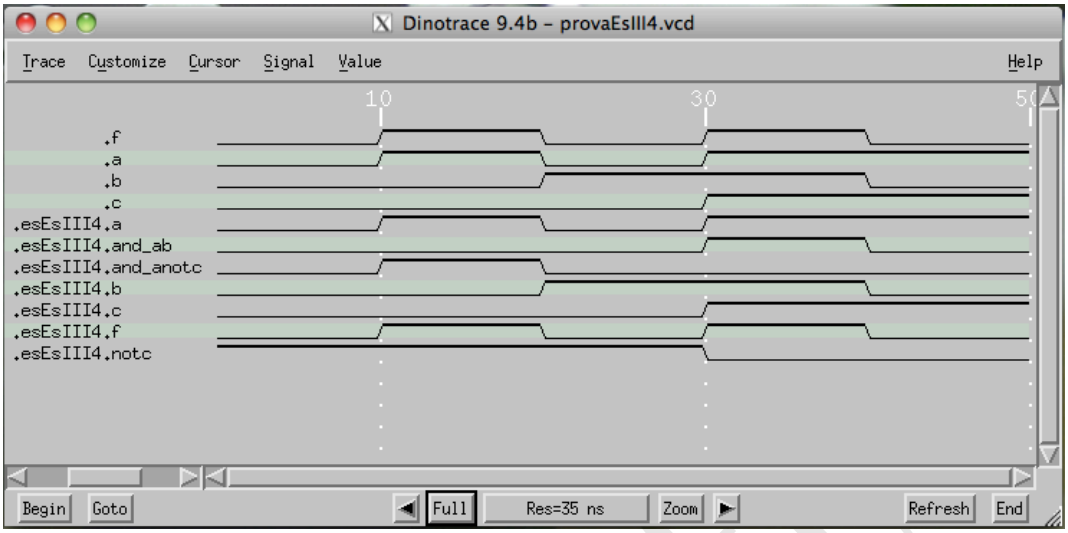

Figura 3.3: Traccia EsIII4 (con dinotrace)

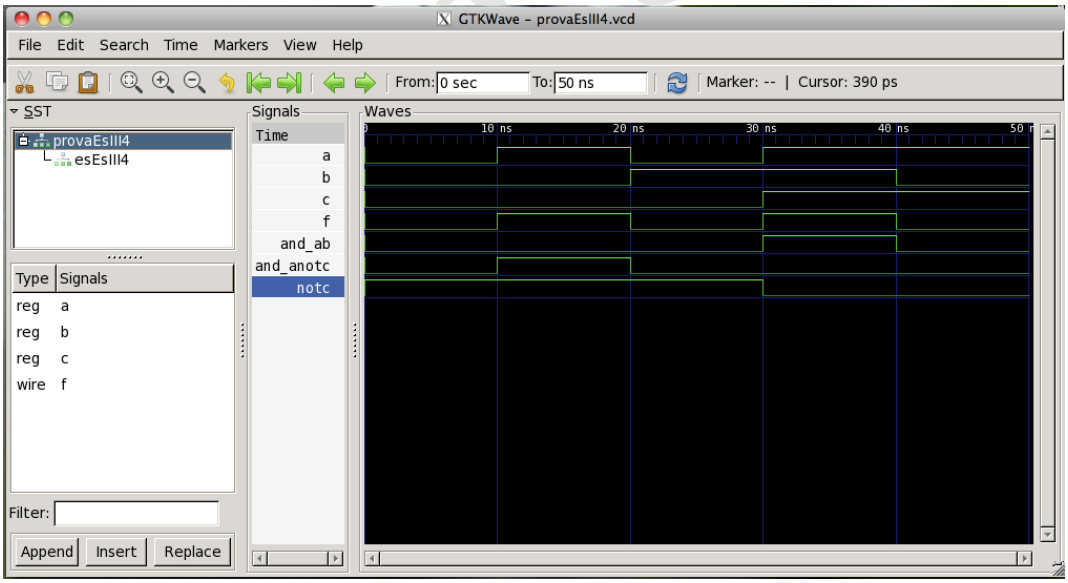

Figura 3.4: Traccia EsIII4 (con gtkwave)

3.1.2 Realizzazione mediate espressioni dell'algebra di Boole

Un secondo modo di utilizzare Verilog per la definizione di moduli che rappresentano reti combinatorie é quello messo a disposizione dal modo "behavioural" di Verilog. Anziché fornire la rete di porte logiche che implementa la rete logica, come appena visto, possiamo fornire direttamente l'espressione booleana calcolata dalla rete logica stessa. Il listato 3.3 dá la definizione del modulo `esIII4` secondo questo secondo modo di utilizzare Verilog. In questo caso, alla linea 14 definiamo un *assegnamento continuo* che determina il valore del parametro in output in termini di un'espressione della logica booleana che ha come parametri i parametri in input del modulo. L'and é rappresentato dalla e commerciale (&), l'or dalla barra verticale (|) e il not dalla tilde (∼).

Questo stesso modulo puó essere testato utilizzando il programma del Listato 3.2 senza modifiche, dato che é definito con lo stesso nome utilizzato nel Listato 3.1. Ovviamente, compiliamo con il comando `iverilog provaEsIII4.vl EsIII4_behavioural.vl` per utilizzare questa definizione invece di quella che utilizza la rete di porte logiche. L'analisi della traccia risultante con `dinotrace` o `gtkwave` evidenzierá lo stesso comportamento del caso in cui avevamo utilizzato il modulo definito come rete di porte logiche.

Listing 3.4: EsIII4 (utilizzando tabelle di veritá)

```
1  //
2  // Questa    la rete dell'esempio a pagina III.4 del libro.
3  //
4  // definisco il modulo
5  // il primo parametro    quello di output, per convenzione
6  // gli altri rappresentano gli input
7  // (non obbligatorio, ma la convenzione e' quella utilizzata anche per le
8  //  porte predefinite (come la and, or, not, etc.) )
9  //
10
11 primitive esIII4(output f,
12   input a, input b, input c);
13
14   table
15     0 0 0 : 0;
16     0 0 1 : 0;
17     0 1 0 : 0;
18     0 1 1 : 0;
19     1 0 0 : 1; // deriva dal secondo and
20     1 0 1 : 0;
21     1 1 0 : 1; // deriva dal primo and
22     1 1 1 : 1; // sia il primo che il secondo and sono a 1
23   endtable
24
25 // fine del modulo
26 endprimitive
```

3.1.3 Realizzazione mediante tabelle di veritá

L'ultimo modo di utilizzo di Verilog per modellare reti combinatorie é quello che fa uso di tabelle di veritá. In questo caso (vedi Listato 3.4) anziché definire un `module` definiamo una `primitive`, a riflettere il fatto che questa modalitá é per lo piú indicata per il trattamento degli oggetti "primitivi" delle nostre costruzioni.

Mentre nel caso della definizione di un modulo l'ordine dei parametri non ha importanza, nel caso della definizione di una rete combinatoria mediante tabella di veritá i parametri devono essere *rigorosamete* indicati mettendo per prima *l'unico* parametro un uscita, che deve essere da 1 bit, e poi tutti i parametri in ingresso, anch'essi da 1 bit.

Successivamente (linee 14–23) possiamo definire la tabella di veritá elencando tante righe quante sono le righe della tabella. Ognuna delle righe dovrá contenere i valori delle variabili di input (separati da spazi) seguiti da un ":", seguito dal valore dell'uscita corrispondente a quella configurazione di input.

Di nuovo, utilizzando questa definizione per la rete `esIII4` e compilandola insieme al modulo di test otterremo gli stessi risultati ottenuti usando il modulo definito come rete di porte standard o secondo il modo behavioural di Verilog.

Verilog permette di definire primitive mediante tabelle di veritá con valori non specificati, esattamente come facciamo noi nel corso. Un bit non specificato in ingresso viene denotato mediante un punto interrogativo.

Nel nostro caso, vediamo dalla tabella della veritá che quando i primi ingressi sono `0 0` o `1 1` il valore del terzo ingresso non é influente sul valore dell'uscita. Possiamo quindi specificare la nostra primitica come nel listato 3.5.

Listing 3.5: EsIII4 (utilizzando tabelle di veritá con valori non specificati)

```
//
// Questa    la rete dell'esempio a pagina III.4 del libro.
//
// definisco il modulo
// il primo parametro    quello di output, per convenzione
// gli altri rappresentano gli input
// (non obbligatorio, ma la convenzione e' quella utilizzata anche per le
//  porte predefinite (come la and, or, not, etc.) )
//

primitive esIII4(output f,
  input a, input b, input c);

  table
    0 0 ? : 0;
    0 1 ? : 0;
    1 0 0 : 1; // deriva dal secondo and
    1 0 1 : 0;
    1 1 ? : 1; // sia il primo che il secondo and sono a 1
  endtable

// fine del modulo
endprimitive
```

3.2 Componenti base

In questa sezione accenniamo alla possibile definizione dei componenti base utilizzati nel libro di testo (commutatori, selezionatori, alu, unitá di memoria e registri).

La definizione di questi componenti permetterá di evidenziare alcune caratteristiche di Verilog utili per la costruzione "per composizione" di moduli complessi a partire da moduli piú semplici.

3.2.1 Commutatore

Cominciamo con la definizione di un commutatore a due vie, con ingressi ed uscite ad 1 bit. Il nostro commutatore avrá dunque tre ingressi (i due ingressi e il segnale di controllo alpha) ed una uscita e potrá essere definito come nel listato 3.6.

Listing 3.6: Commutatore a due ingressi da 1 bit ciascuno (modo behavioural)

```
module commutatore(output z, input x, input y, input alpha);

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

endmodule
```

Se volessimo definire il commutatore in modo da trattare ingressi da piú bit, potremmo sfruttare il meccanismo dei *parametri* di Verilog. Considerate il listato **??**. In questo caso, abbiamo definito un `parameter`

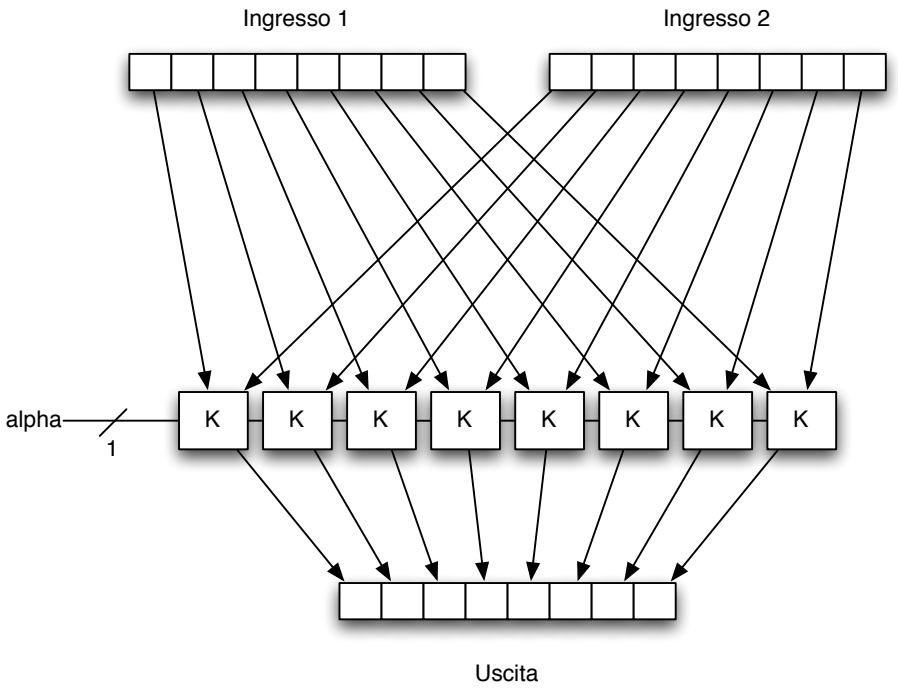

Figura 3.5: Commutatore da due ingressi (8bit) realizzato con 8 commutatori da 1bit

alla riga 3 che si chiama `N` e vale `32`. In realtá, quando istanzieremo questo modulo, potremo variare il valore assegnato ad N utilizzando una sintassi che subito dopo il tipo del modulo istanziato indica fra parentesi tonde precedute dalla grattella il valore dei paremetri del modulo stesso, *nell'ordine in cui sono definiti all'interno del modulo*. Ad esempio, potremmo ottenere un'istanza di `commutatore_nbit` di nome `k12` utilizzando il codice

```
commutatore_nbit #(16) k12(z1,x1,x2,alpha[1]);
```

che istanzierá un commutatore di valori a 16 bit anziché a 32 bit.

Listing 3.7: Commutatore a due ingressi da N bit ciascuno (modo behavioural)

```
1 module commutatore_nbit(z, x, y, alpha);
2
3   parameter N = 32;
4
5   output [N-1:0]z;
6   input [N-1:0]x;
7   input [N-1:0]y;
8   input alpha;
9
10  assign z = ((~alpha) ? x : y);
11
12 endmodule
```

Verilog mette a disposione un comando per generare un certo numero di istanze di un singolo modulo, il comando `generate`. Utilizzando tale comando, e assumendo di aver definito il commutatore a due vie per ingressi da 1 bit come nel listato 3.6, possiamo definire un commutatore a N bit come giustapposizione di N commutatori per ingressi da 1 bit. Il listato 3.8 illustra questa tecnica e la Fig. 3.5 mostra la realizzazione con la stessa tecnica di un commutatore per ingressi da 8 bit a partire da commutatori con ingressi da 1 bit.

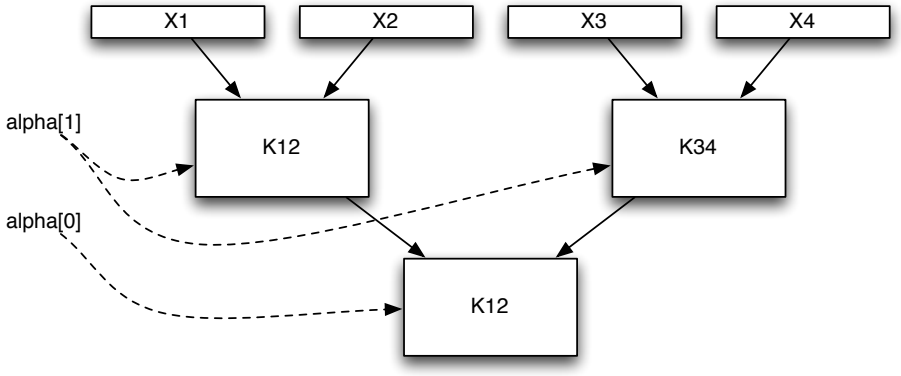

Figura 3.6: Commutatore da 4 ingressi realizzato con 3 commutatori da 2 ingressi

Listing 3.8: Commutatore a due ingressi da N bit ciascuno definito come giustapposizione di N commutatori per valori da 1 bit

```
1  module commutatore_nbit_generative(z,x,y,alpha);
2
3    parameter N = 32;
4
5    output [N-1:0]z;
6    input  [N-1:0]x;
7    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       commutatore t(z[i],x[i],y[i],alpha);
16     end
17   endgenerate
18
19 endmodule
```

In questo caso dichiariamo i parametri in ingresso e in uscita come valori da N bit (linee 5–8). Successivamente dichiariamo una "variabile generativa" che servirá da indice per la generazione delle istanze dei commutatori che commutano ingressi da un singolo bit (linea 10) e la utilizziamo per "generare" N instanze del modulo `commutatore` (linee 12–17).

Utilizzando questo nuovo modulo `commutatore_nbit_generative` possiamo ora (finalmente) definire il commutatore a 4 vie (cioé quello che sceglie uno fra i quattro ingressi disponibili, a seconda del valore di una variabile di controllo da 2 bit. Questo commutatore puó essere definito[1] come composizione di tre commutatori a 2 vie: uno sceglie fra il primo e il secondo ingresso utilizzando come variabile di controllo l'ultimo bit di alpha, un altro sceglie fra il terzo ed il quarto ingresso, utilizzando la stessa variabile di controllo. Infine, un terzo commutatore sceglie fra l'uscita del primo e l'uscita del secondo commutatore, utilizzando come variabile di controllo il bit piú significativo di alpha (vedi listato 3.9 e Fig. 3.6).

Listing 3.9: Commutatore a quattro ingressi da N bit ciascuno definito come composizione di 3 commutatori a due ingressi

```
1  module commutatore_4vie_nbit(z,x1,x2,x3,x4,alpha);
```

[1] anche se non in modo molto efficiente, visto che in questo caso arriviamo a quattro livello di logica, in luogo dei 2 che deriverebbero dalla definizione "diretta"

```
2
3   parameter N = 32;
4
5   output [N-1:0]z;
6   input  [N-1:0]x1;
7   input  [N-1:0]x2;
8   input  [N-1:0]x3;
9   input  [N-1:0]x4;
10  input  [0:1]alpha;
11
12  wire [N-1:0]z1;
13  wire [N-1:0]z2;
14
15  commutatore_nbit_generative #(N) k12(z1,x1,x2,alpha[1]);
16  commutatore_nbit_generative #(N) k34(z2,x3,x4,alpha[1]);
17  commutatore_nbit_generative #(N) k  (z, z1,z2,alpha[0]);
18
19 endmodule
```

3.3 Ritardi di stabilizzazione

Negli esempi visto fino ad ora, si puó notare come la stabilizzazione delle uscite delle porte logiche sia istantanea. Questo di fatto non é. Il simulatore Verilog di Icarus non fa di suo assunzioni sul ritardo delle porte. Per modellare il ritardo di $1t_p$ che si assunto nel libro di testo per le porte con al piú 8 ingressi, utilizziamo la possibilitá offerta da Verilog di introdurre una temporizzazione esplicita negli statement. Ad esempio, con lo statement

```
assign #1 z = a & b;
```

si puó asserire che l'assegnamento continuo del valore di $a\ AND\ b$ richiede 1 unitá di tempo (1 nsec, con la `timescale` di default utilizzata nei nostri esempi). Possiamo definire dunque definire tutte le nostre porte logiche elementari utilizzando un codice come quello del listato 3.10 che implementa un AND con un ritardo di 1 unitá di tempo.

Listing 3.10: Porta AND con ritardo di una unitá di tempo

```
1 module and_tp(output z, input x, input y);
2
3   assign #1 z = x & y;
4
5 endmodule
```

Utilizzando queste porte, in luogo di quelle standard, possiamo definire i nostri componenti in modo che il ritardo di stabilizzazione sia modellato correttamente. Il listato 3.11 mostra il codice di un confrontatore realizzato con le porte che modellano il ritardo. Il listato 3.12 mostra un semplice modulo di test e la Fig. 3.7 fa vedere la traccia che si ottiene.

Listing 3.11: Confrontatore con ritardi modellati secondo le convenzioni del libro di testo

```
1 //
2 // confrontatore : 1 se i dati sono diversi, 0 se sono uguali
3 // somma di prodotti : (not(x) and y) or (x and not(y))
4 //
5 module confrontatore(z,x,y);
6
7   output wire z;
8   input wire x;
9   input wire y;
10
11  wire termine1;
```

```
12   wire termine2;
13   wire notx;
14   wire noty;
15
16   not(notx,x);
17   not(noty,y);
18
19   and_tp and1(termine1,notx,y);
20   and_tp and2(termine2,x,noty);
21
22   or_tp or1(z,termine1,termine2);
23
24 endmodule
```

Listing 3.12: Modulo di test del confrontatore

```
1  `timescale 1ns / 1ps
2
3  module prova_confrontatore();
4
5    //
6    // registri che conterranno i valori di input alla rete da testare
7    //
8    reg a;
9    reg b;
10
11   //
12   // filo per l'output della rete da testare (potrebbe anche essere reg)
13   //
14   wire f;
15
16   //
17   // istanziazione della rete (non    una chiamata di procedura!)
18   // i parametri attuali vengono collegati come specificato mediante
19   // i parametri formali nella definizione del modulo esIII4
20   // prima delle parentesi tonde dei parametri troviamo il nome del modulo
21   // istanziato (serve per distinguere fra istanziazioni diverse)
22   //
23   confrontatore c1(f,a,b);
24
25   //
26   // statement da eseguire all'inizio della simulazione
27   //
28   initial
29   begin
30
31     // file di ouput per la visualizzazione del comportamento
32     $dumpfile("prova_confrontatore.vcd");
33     // esegui il dump dei cambiamenti di tutte le variabili in gioco
34     $dumpvars;
35
36     // assegnamento iniziale dei registri di input alla rete
37     a = 0;
38     b = 0;
39
40     // simulazione di cambi della configurazione in ingresso
41     //
42     #10 a = 1;
43     #10 b = 1;
44     #10 a = 0;
45     #10 b = 0;
46
47     // attendi altri 10 nsec e termina la simulazione
48     // (producendo il .vcd con il dump dei cambiamenti delle variabili)
49     #10 $finish;
```

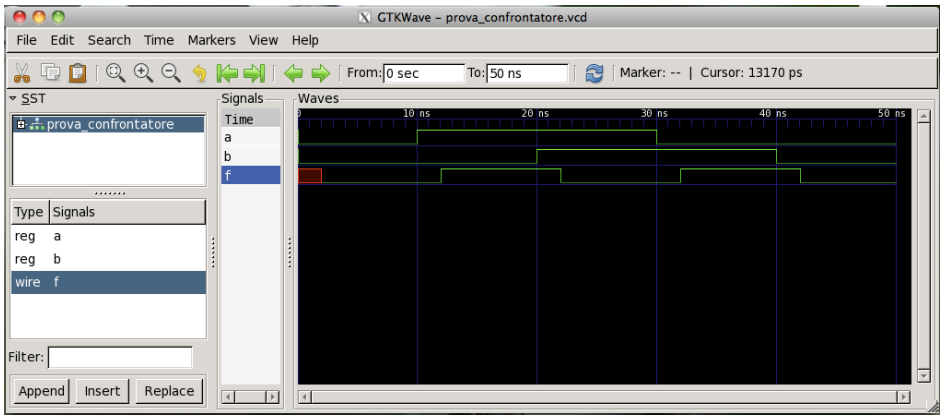

Figura 3.7: Visualizzazione del comportamento del confrontatore con ritardi del listato 3.12

```
50
51   // fine del (blocco di) comando da eseguire all'inizio della simulazione
52   end
53
54 // fine del modulo di test
55 endmodule
```

Come si nota, l'uscita del confrontatore assume il valore atteso dopo 2nsec dalla stabilizzazione dei valori in ingresso. Quando si voglia modellare un componente elementare utilizzando il modo behavioural di Verilog, si puó inserire direttamente un ritardo corrispondente al numero dei livelli di logica del componente nello statement di assegnamento continuo che definisce l'uscita. Il listato 3.13 mostra come possiamo definire un confrontatore in modalitá behavioural con un ritardo di 2 unitá di tempo. E' chiaramente responsabiltá del programmatore (nostra), fare in modo che l'ammontare del ritardo corrisponda alla profonditá dell'espressione booleana utilizzata per implementare l'uscita.

Listing 3.13: Confrontatore (behavioural mode) con ritardi di stabiizzazione

```
1 //
2 // confrontatore : 1 se i dati sono diversi, 0 se sono uguali
3 // somma di prodotti : (not(x) and y) or (x and not(y))
4 //
5 module confrontatore(z,x,y);
6
7   output wire z;
8   input wire x;
9   input wire y;
10
11   assign #2 z = ((~x) & y) | (x & (~y));
12
13 endmodule
```

3.4 Selezionatore

Il listato 3.14 riporta il modulo che implementa un selezionatore da 1 bit espresso utilizzando il modo behavioural. Il selezionatore ha un ingresso da 1 bit, `x` e un ingresso di controllo `alpha` ancora da 1 bit. L'ingresso di controllo determina quale delle due uscite vale quanto l'ingresso `x`. L'altra uscita é posta (convezionalmente) a 0. Per questo motivo, si usano due statement di assegnamento continuo, uno per la prima uscita `z[0]` e l'altro per la seconda uscita `z[1]`.

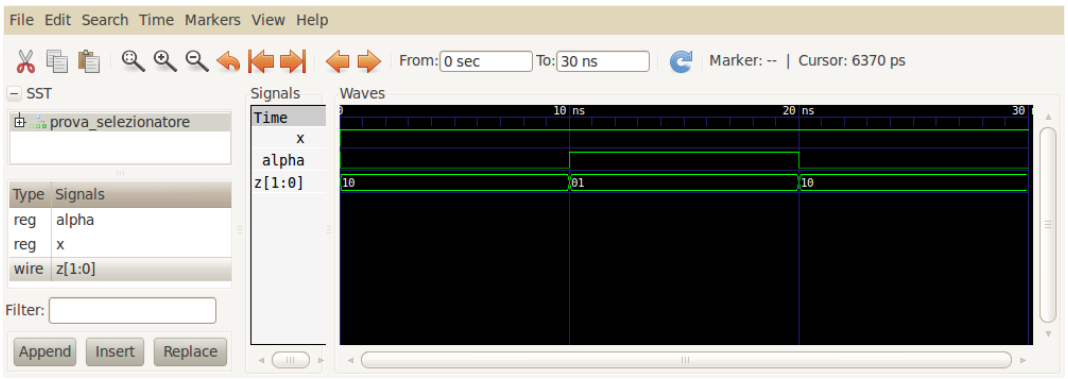

Figura 3.8: Comportamento del modulo selezionatore da 1 bit del Listato 3.14

Listing 3.14: Selezionatore da 1 bit

```
1 module selezionatore(output [1:0]z,
2    input x, input alpha);
3
4   assign z[1] = (~alpha) & x;
5   assign z[0] = (alpha) & x;
6
7 endmodule
```

Il comportamento di questo modulo selezionatore é evidenziato nella Fig. 3.10, ottenuta utilizzando `gtkwave`.

Volendo utilizzare ingressi e uscite da piú bit, sempre in un selezionatore a due vie, si possono utilizzare due variabile per modellare l'uscita del modulo, dimensionandole mediante un parametro del modulo stesso, come illustrato nel Listato 3.15.

Listing 3.15: Selezionatore a due vie con ingresso da N bit

```
1 module selezionatoreN(z0, z1, x, alpha);
2
3    parameter N = 4;
4
5    output [(N-1):0]z0;
6    output [(N-1):0]z1;
7    input  [(N-1):0] x;
8    input  alpha;
9
10   assign z0 = (alpha==0 ? x : 0);
11   assign z1 = (alpha==1 ? x : 0);
12
13 endmodule
```

Il parametro `N` in questo caso puó essere instanziato all'atto dell'utilizzo del modulo, come illustrato nel Listato 3.16 alla riga 10.

Listing 3.16: Listato di prova per il selezionatore a due vie con ingresso da N bit

```
1 //
2 // definizione di comodo per stabilire il range temporale
3 //
4 `timescale 1ns / 1ps
5
6
7 // nome del modulo; senza parametri perch     quello di test
8 module prova_selezionatore();
```

```
9
10   parameter N = 4;
11   //
12   // registri che conterranno i valori di input alla rete da testare
13   //
14   reg [N-1:0]x;
15   reg alpha;
16
17   wire [N-1:0]z0;
18   wire [N-1:0]z1;
19
20   selezionatoreN #(N) S(z0,z1,x,alpha);
21
22   //
23   // statement da eseguire all'inizio della simulazione
24   //
25   initial
26   begin
27
28     // file di ouput per la visualizzazione del comportamento
29     $dumpfile("prova_selezionatore.vcd");
30     // esegui il dump dei cambiamenti di tutte le variabili in gioco
31     $dumpvars;
32
33     // assegnamento iniziale dei registri di input alla rete
34     x = 15;
35     alpha = 0;
36
37     // simulazione di cambi della configurazione in ingresso
38     //
39
40     #10 alpha = 1;
41     #10 alpha = 0;
42
43     // attendi altri 10 nsec e termina la simulazione
44     // (producendo il .vcd con il dump dei cambiamenti delle variabili)
45     #10 $finish;
46
47   // fine del (blocco di) comando da eseguire all'inizio della simulazione
48   end
49
50 // fine del modulo di test
51 endmodule
```

L'output del listato di prova é riportato in Fig. 3.9.

3.5 ALU

Vediamo ora come pu essere modellata una semplice ALU. L'ALU che utilizziamo come esempio implementa 4 operazioni su interi, addizione, sottrazione, shift destro e sinistro. In aggiunta al risultato, l'ALU genera due bit "flag", il bit zero, che vale 1 in caso di risultato pari a zero, e il bit segno, che vale 1 in caso di risultato negativo e 0 in caso di risultato positivo. La lunghezza degli ingressi é trattata in modo parametrico. Il valore di default del parametro é 32 (bit). Il parametro puó essere cambiato in modo semplice all'atto dell'istanziazione del modulo avendo cura di interporre fra il nome del modulo e la lista dei parametri il valore del parametro "attuale" racchiuso fra parentesi e preceduto dalla gratella, come del resto gi visto in altri casi.

Alla riga 13, si definiscono tre registri per contenere i valori che saranno associati alle uscite. Lo statement `always` della riga 17 provvede ad assegnare il valore corretto ai registri, a seconda dei parametri di ingresso del modulo.

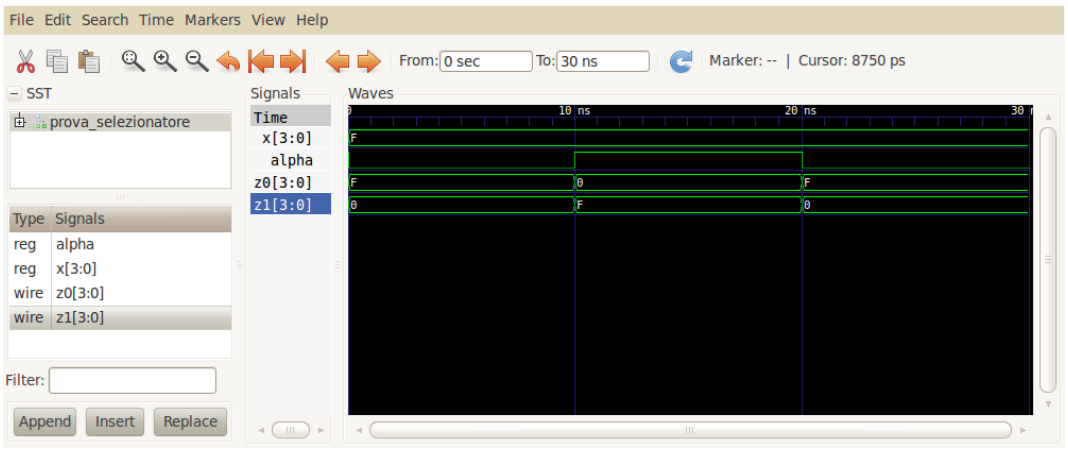

Figura 3.9: Comportamento del modulo selezionatore da N bit del Listato 3.15

La riga 29 definisce il ritardo convenzionale della ALU. Gli statement di assegnamento alle righe 31–33 realizzano l'assegnamento "continuo" dei valori dei registri alle uscite.

Il codice Verilog per l'ALU é riportato nel Listato 3.17.

Listing 3.17: Semplice ALU a quattro operazioni

```
1  module alu(risultato,zero,segno,ingresso1,ingresso2,alpha);
2
3    parameter N = 32;   // larghezza della parola
4
5  `define RITARDO 10
6
7    output [N-1:0]risultato;
8    output zero;
9    output segno;
10
11   input  [N-1:0]ingresso1;
12   input  [N-1:0]ingresso2;
13   input  [1:0]alpha;    // 4 operazioni possibili
14
15   reg [N-1:0] temp;     // temporanei per i calcoli
16   reg bits;
17   reg bitz;
18
19   assign  zero      = bitz;
20   assign  segno     = bits;
21   assign  risultato = temp;
22
23   // aggiornamento temporanei
24   always @(alpha or ingresso1 or ingresso2)
25   begin
26    case(alpha) // assegnamento bloccante, in modo da poter lavorare sui ris
27      0 : temp = ingresso1+ingresso2;  // somma
28      1 : temp = ingresso1-ingresso2;  // sottrazione
29      2 : temp = ingresso1>>ingresso2; // shift destro (divisione per 2)
30      3 : temp = ingresso1<<ingresso2;   // shift sinistro (molt per 2)
31    endcase
32    // assegnamento *non* bloccante, *dopo* l'aggiornamento del temp
33    bits <= (temp[N-1]);
34    bitz <= (temp==0 ? 1 : 0);
35   end
36
37 endmodule
```

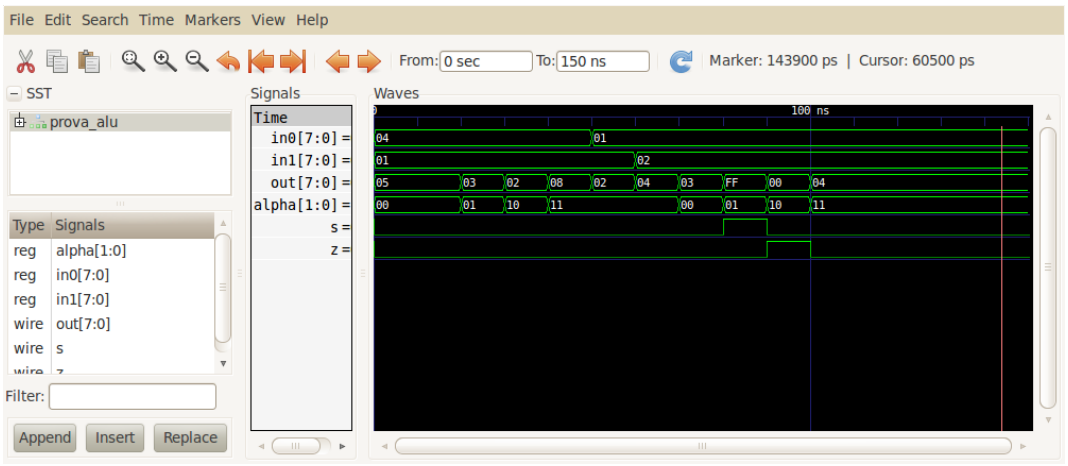

Figura 3.10: Comportamento del modulo ALU del Listato 3.17

3.6 Registri & Memorie

Vediamo adesso come modellare utilizzando il Verilog i registri impulsati e le memorie.

Clock Per questi due tipi di componenti dobbiamo considerare un ingresso aggiuntivo, il segnale di clock. Questo ingresso puó essere modellato con gli strumenti che abbiamo utilizzato fino a questo punto per i segnali "a livelli", ovvero semplicemente specificando in ingresso al modulo una linea da un bit. A livello di programma di test, utilizzeremo una tecnica particolare per modellare il segnale di clock: uno statement `always` dentro al quale assegneremo al clock il suo valore complementare, realizzando l'assegnamento con uno statement bloccante e un ritardo pari alla lunghezza del ciclo di clock stesso. Il modo piú elementare di realizzare un segnale di clock é riportato nel Listato 3.18. Variando i ritardi introdotti prima dei due statement di assegnamento alle righe 10 e 12 si puó ottenere un segnale con un periodo fissato, e che vale 0 per la maggior parte del tempo.

Listing 3.18: Semplice codice per la realizzazione di un segnale di clock

```
1  `timescale 1ns / 1ps
2
3  module prova_clock();
4
5    reg clock;
6
7    always
8    begin
9      if(clock==0)
10       #4 clock = 1;
11     else
12       #1 clock = 0;
13   end
14
15
16   initial
17   begin
18     $dumpfile("gen_clock.vlc");
19     $dumpvars;
20
21     #50 $finish;
22
23   end
24 endmodule
```

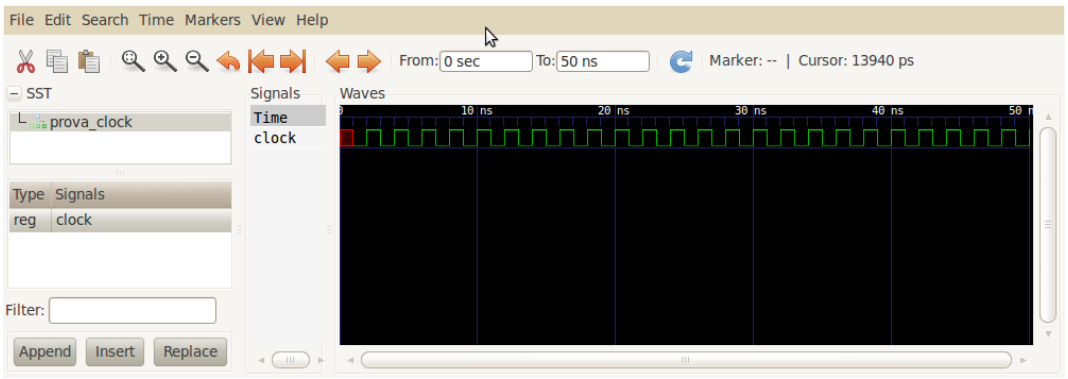

Figura 3.11: Simulazione di un segnale di clock (stesso tempo per le fasi 0 e 1)

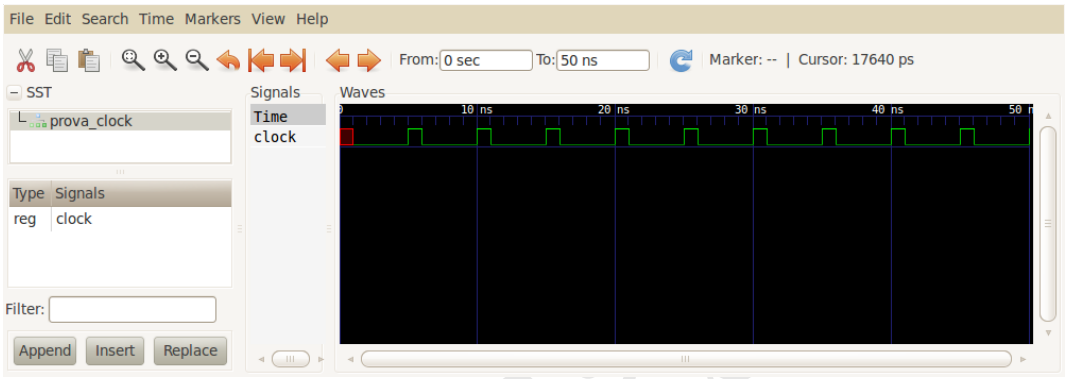

Figura 3.12: Simulazione di un segnale di clock (fase 0 4 volte pi lunga della fase 1)

La Fig. 3.11 fa vedere il comportamento ottenuto simulando l'esecuzione del Listato 3.18, mentr la Fig. 3.12 mostra il comportamento ottenuto imponendo un ritardo di 4 cicli alla riga 10.

3.6.1 Registro

Un registro impulsato a `N` bit puó essere definito come riportato nel Listato 3.19. Come possiamo vedere dalla Fig. 3.13, il registro viene correttamente inizializzato a 0 e scritto solo in corrispondenza del fronte di discesa del segnale di clock, se contemporaneamente il beta vale 1.

Lo statement `initial` alla riga 20 provvede all'inizializzazione del registro a 0. In assenza di tale inizializzazione, la lettura del registro produrrebbe valori `x` (il valore scelto in Verilog per rappresentare lo stato "indefinito" di un dispositivo).

Lo statement `always` alla 25 é quello che effettivamente implementa la scrittura del registro. Sul fronte di discesa[2] del segnale di clock (cioé quando il segnale di clock scende dal valore 1 al valore 0) e se e solo se il beta vale 1, andiamo a memorizzare nel registro il valore dell'ingresso.

Il modulo é parametrico rispetto al numero di bit utilizzati per il registro (`N`). Il valore del parametro puó essere modificato in fase di istanziazione del modulo `registro`, come al solito.

Listing 3.19: Implementazione di un registro da N bit

```
1 // registro da N bit
```

[2]questa é una scelta. Avremmo potuto utilizzare anche il fronte di salita, indicando un posedgde invece del negedge.

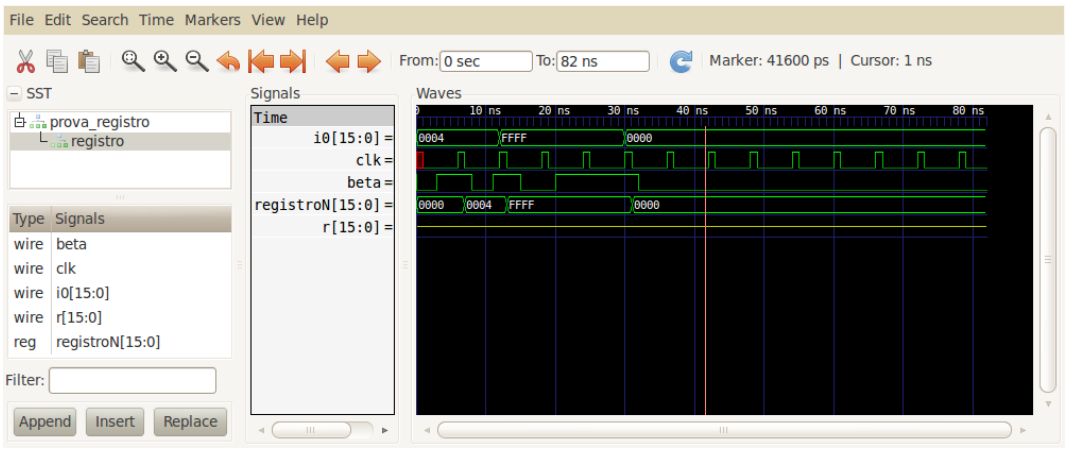

Figura 3.13: Comportamento del registro da N bit definito nel Listato 3.19.

```
2  // beta     il controllo di scrittura
3  // i0       il segnale in ingresso
4  // clk      il clock
5  //
6  // semantica standard: scrive i0 se clk alto e beta, uscita sempre uguale
7  // al contenuto del registro
8  //
9  module registro(r,clk,beta,i0);
10
11   parameter N = 32;
12
13   output [N-1:0]r;
14   input clk;
15   input beta;
16   input [N-1:0]i0;
17
18   reg [N-1:0]registroN;
19
20   initial
21   begin
22     registroN = 0;
23   end
24
25   always @ (negedge clk)
26   begin
27   if(beta==1)
28     registroN = i0;
29   end
30
31   assign r = registroN;
32 endmodule
```

3.6.2 Memorie

Passiamo ora a vedere una possibile modellizzazione di un modulo di memoria con Verilog.

Consideriamo una memoria di cui si possano definire mediante opportuni parametri la capacitá (primo parametro che rappresenta il numero di bit N dell'indirizzo e, di conseguenza, il numero di celle (2^N) della memoria) e la lunghezza della parola (secondo parametro, M). Consideriamo ovviamente un ingresso `clock` (il segnale di clock) ed un ingresso `beta` (abilitazione alla scrittura). La semantica del modulo é quella usuale:

- con `beta=0`, in ogni momento l'uscita della memoria pari al contenuto della sua cella indirizzata dall'ingresso indirizzo come risultante/scritto all'ultimo "battimento" del clock.
- se `beta=1`, il contenuto della cella di indirizzo `addr` viene sostituito con la parola presente sull'ingresso della memoria.

Una possibile implementazione del modulo di memoria é riportata nel Listato 3.20.

Listing 3.20: Modulo di memoria

```
1  module memoria # (parameter N=6, M=32) (
2    output [M-1:0] data_out,
3    input clock,
4    input beta,
5    input [N-1:0] address,
6    input [M-1:0] data_in
7  );
8
9    reg [M-1:0]mem_bank[2**N-1:0];
10   reg [N:0]i;
11
12   initial
13   begin
14     for(i=0; i<2**N; i=i+1)
15       mem_bank[i]=0;
16   end
17
18   always @ (posedge clock)
19   begin
20     if(beta)
21       mem_bank[address] = data_in;
22   end
23
24   assign
25     data_out = mem_bank[address];
26
27 endmodule
```

Alla riga 9 definiamo la memoria come vettore da 2^N posizioni di parole da M bit. Lo statement `initial` della riga 12 provvede all'inizializzazione del modulo di memoria, in modo che tutte le locazioni contengano inizialmente il valore 0 Lo statement `always` della linea 18 fá si che il valore presente sull'ingresso venga scritto nella locazione individuata dall'indirizzo in ingresso sul fronte di salita del segnale di clock. Infine, l'assegnamento continuo alla linea 24 provvede a copiare in uscita il contenuto della cella di memoria individuata dallo stesso indirizzo in ingresso. I paramtri di default definiscono una memoria di 64 posizioni ($N = 6$, numero delle locazioni di memoria $= 2^6$) e la lunghezza della parola di memoria ($N = 32$).

La Figura 3.14 riporta un esempio del comportamento del modulo di memoria del Listato 3.20 ($N = 4$) e $M = 32$.

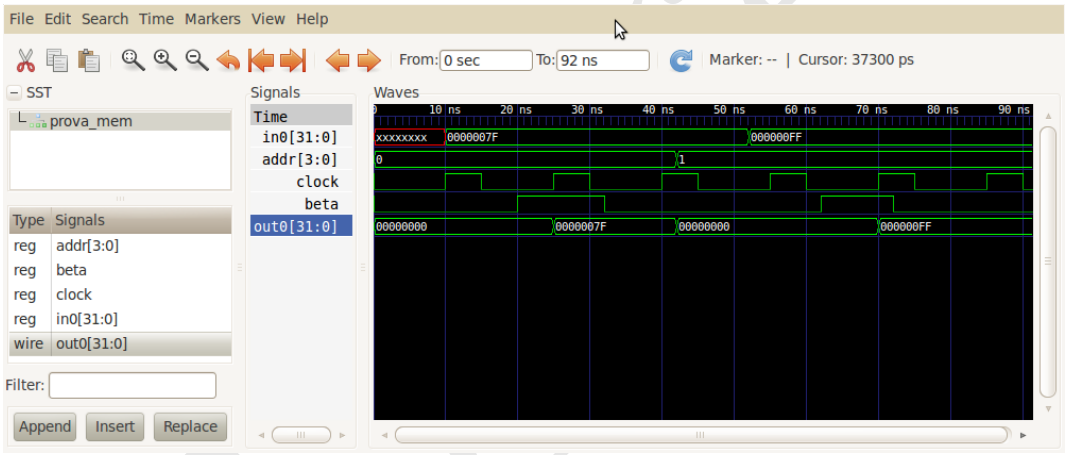

Figura 3.14: Comportamento della memoria definita nel Listato 3.20.

Capitolo 4

Automi e Reti sequenziali

In questo capitolo vediamo Verilog permette di modellare automi a stati finiti e reti sequenziali. Ci avvaliamo di un esempio trattato nel libro di testo (esempio a pagina 12 del Capitolo III). Tramite questo esempio vediamo come si puó moellare utilizzando Verilog sia una rete sequenziale (LLC) che un automa di Mealy o di Moore.

4.1 Automa di Mealy

Cominciamo considerando l'implementazione diretta di un automa di Mealy. Consideriamo l'automa riportato in Fig. 4.1. Il listato del modulo che implementa l'automa é quello riportato nel Listato 4.1.

Il modulo ha due ingressi e una singola uscita. Gli ingressi sono l'ingresso vero e proprio e il segnale di clock necessario per impulsare il registro di stato. L'uscita rappresenta l'uscita dell'automa. Ingressi e uscite sono da un singolo bit nel nostro caso. Lo stato interno é modellato mediante un registro da 1 bit, non visibile all'esterno.

Si noti come l'uscita é definita come un registro (anzich come un wire, vedi linea 3 del listato) in modo da poterne permettere l'assegnamento.

Listing 4.1: Implementazione dell'automa di Mealy di Fig. 4.1

```
1 module EsIII12(z,clock,x);
2
3   output reg z;   // uscita
4   input wire x;   // ingresso
5   input wire clock;   // clock
6
7   reg stato;  // 1 bit di default
8
9 `define S0  0   // codifica stati
10 `define S1  1
```

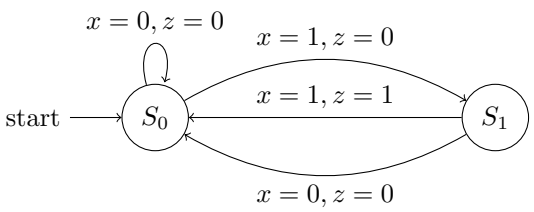

Figura 4.1: Automa di Mealy (EsIII)

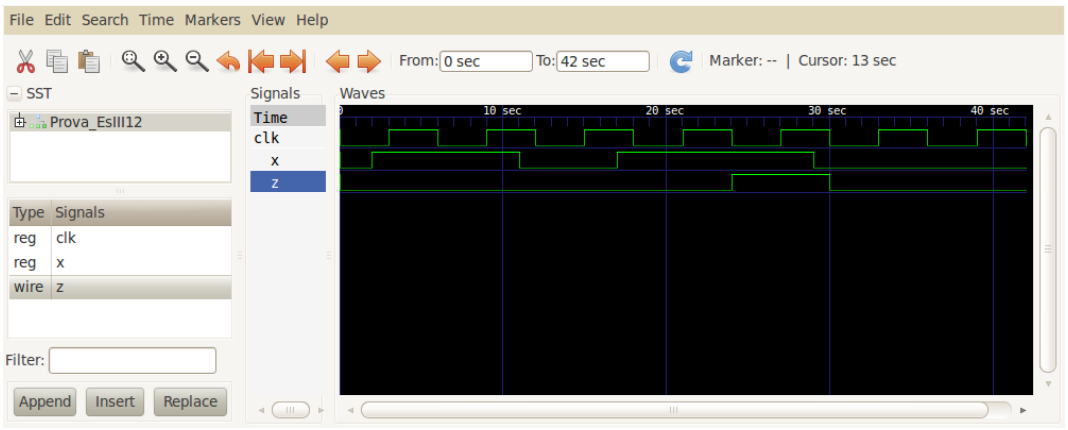

Figura 4.2: Funzionamento dell'automa di Mealy del Listato 4.1

```
11
12   initial
13   begin
14     stato = `S0;    // si inizia in S0
15   end
16
17   always @(negedge clock)
18   begin
19   case(stato)
20     `S0: begin
21   stato <= (x==0 ? `S0 : `S1);
22   z     <= 0;
23       end
24     `S1: begin
25   stato <= `S0;
26         z     <= (x==0 ? 0 : 1);
27   end
28   endcase
29   end
30
31 endmodule
```

Alle linee 9 e 10 si definiscono due costanti per esprimere simbolicamente i due stati del nostro automa. L'inizializzazione dell'automa avviene con lo statement `initial` della linea 12, che provvede ad inizializzare lo stato a `S0`.

Il funzionamento dell'automa é definito mediante lo statement `always` della riga 17 e dal corrispondente blocco begin/end delle linee 18–33. Ogni qualvolta il fronte del clock in discesa, `sS0` o `S1` (case di linea 19), assegnamo di conseguenza il nuovo valore dello stato interno e l'uscita. L'operazione avviene con un assegnamento non bloccante, tipico delle situazioni modellate nelle reti sequenziali.

La Fig. 4.2 riporta il funzionamento del modulo che modella l'automa di Mealy della Fig. 4.1.

4.2 Automa di Moore

Consideriamo l'automa di Moore equivalente all'automa di Mealy della Fig. 4.1. Tale automa é riportato nella Fig. 4.3.

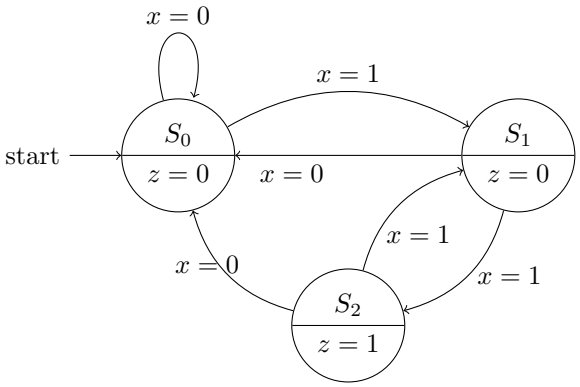

Figura 4.3: Automa di Moore (EsIII)

Listing 4.2: Implementazione dell'automa di Moore di Fig. 4.3

```
1  module EsIII12(z,clock,x);
2
3    output reg z;   // uscita
4    input wire x;   // ingresso
5    input wire clock;   // clock
6
7    reg [0:1]stato;   // stato interno
8
9  `define S0  2'b00   // codifica stati
10 `define S1  2'b01
11 `define S2  2'b11
12
13   initial
14   begin
15     stato = `S0;
16     z = 0;
17   end
18
19   always @(negedge clock)
20   begin
21   case(stato)
22     `S0: stato = (x==0 ? `S0 : `S1);
23     `S1: stato = (x==0 ? `S0 : `S2);
24     `S2: stato = (x==0 ? `S0 : `S1);
25      default: stato = `S0;
26   endcase
27   // l'uscita dipende esclusivamente dallo stato interno
28   z <= (stato==`S2 ? 1 : 0);
29   end
30
31 endmodule
```

L'implementazione secondo il modello behavioural di Verilog avviene in maniera del tutto analoga a quanto visto per l'automa di Mealy. Le poche differenze degne di nota sono:

- abbiamo bisogno di un registro da 2 bit per la rappresentazione dello stato (linea 7 e definizione delle costanti di stato alle linee 9–11)

- siccome servono tre stati, definiamo un caso "default" nel case per modellare situazioni di errore. In questo caso ci riportiamo nello stato iniziale (linea 25).

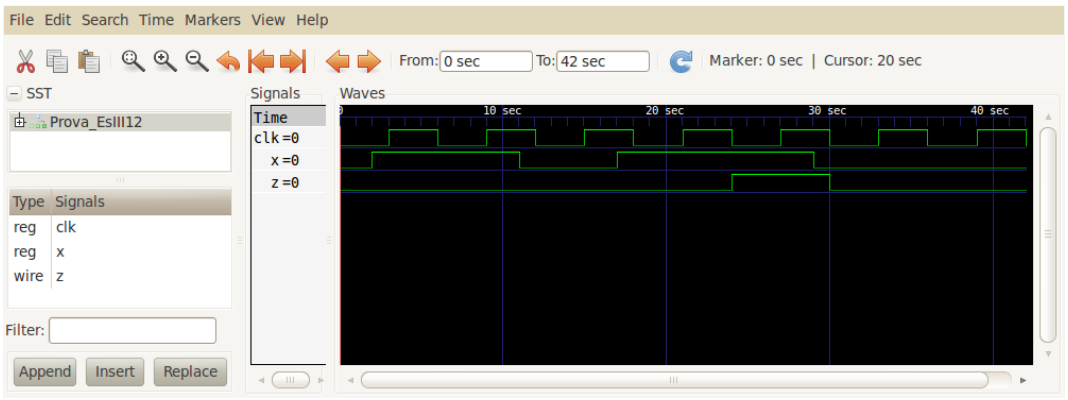

Figura 4.4: Funzionamento dell'automa di Moore del Listato 4.2

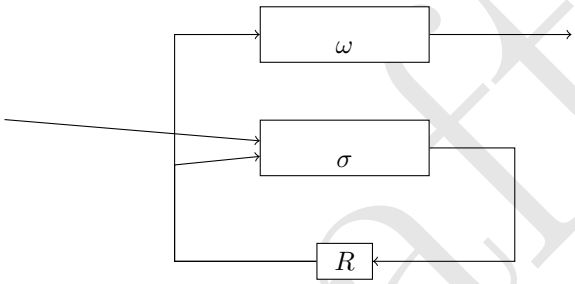

Figura 4.5: Rete sequenziale di Moore

- il calcolo del nuovo stato interno *precede* (assegnamento sincrono) il calcolo dell'uscita (assegnamenti sincroni alle linee 22–24 e assegnamento asincrono alla 28).

Utilizzando lo stesso modulo di test utilizzato per testare il modulo che implementava il corrispondente automa di Mealy, otteniamo l'uscita di Fig. 4.4.

Si puó notare come effettivamente l'output sia lo stesso di quello riportato in Fig. 4.2 "sfalsato" di un ciclo di clock, com'é giusto aspettarsi quando si utilizza un automa di Moore invece che di Mealy.

4.3 Reti sequenziali LLC

In questa sezione mostriamo come si possa modellare in Verilog una rete seqeunziale mediante reti combinatorie ω/σ e registro di stato. Mostriamo cioé come si possa modellare una rete sequenziale LLC (Level input, Level output, Clocked).

Consideriamo a tale scopo il solo automa di Moore della Fig. 4.3. La Fig. 4.5 riporta il modello corrispondente di rete sequenziale.

Per come abbiamo scelto la codifica degli stati (vedi Listato 4.2), la funzione ω dará come risultato sempre il primo bit (quello piú significativo) del registro di stato. Possiamo dunque immaginare di implementare la rete combinatoria ω in Verilog come illustrato dal Listato 4.3. Come descritto nel testo (pag. III.49) la rete σ pu essere implementata come illustrato nel Listato 4.4, dal momento che abbiamo:

$$\begin{aligned} stato[1] &= x\ \overline{stato[1]}\ stato[0] \\ stato[0] &= x\ \overline{stato[1]}\ \overline{stato[0]} + x\ stato[1] \end{aligned}$$

Listing 4.3: Implementazione della rete ω relativa alla rete sequenziale che implementa l'automa di Moore della Fig. 4.3

```
module omega(z,s);

  output wire z;
  input  wire [1:0]s;

  assign
    z = s[1];

endmodule
```

Listing 4.4: Implementazione della rete σ relativa alla rete sequenziale che implementa l'automa di Moore della Fig. 4.3

```
module sigma(z,x,s);

  output wire [1:0]z;
  input  wire x;
  input  wire [1:0]s;

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

endmodule
```

La rete sequenziale nella sua interezza utilizza le due reti σ e ω e un registro di stato da due bit, come illustrato nel Listato 4.5.

Listing 4.5: Implementazione della rete sequenziale che implementa l'automa di Moore della Fig. 4.3

```
module moore(z,x,clk);

  output reg       z;
  input  wire      x;
  input  wire      clk;
  // stato interno
  reg [1:0]stato;

  // codifica stati interni
`define S0 2'b00
`define S1 2'b01
`define S2 2'b10

  // layout
  wire  [1:0]nuovostato;
  wire       uscita;

  sigma f_sigma(nuovostato,x,stato);
  omega f_omega(uscita,stato);

  initial
  begin
    stato = `S0;
    z = 0;
  end

  always@(negedge clk)
  begin
    stato <= nuovostato;
    z     <= uscita;
  end
```

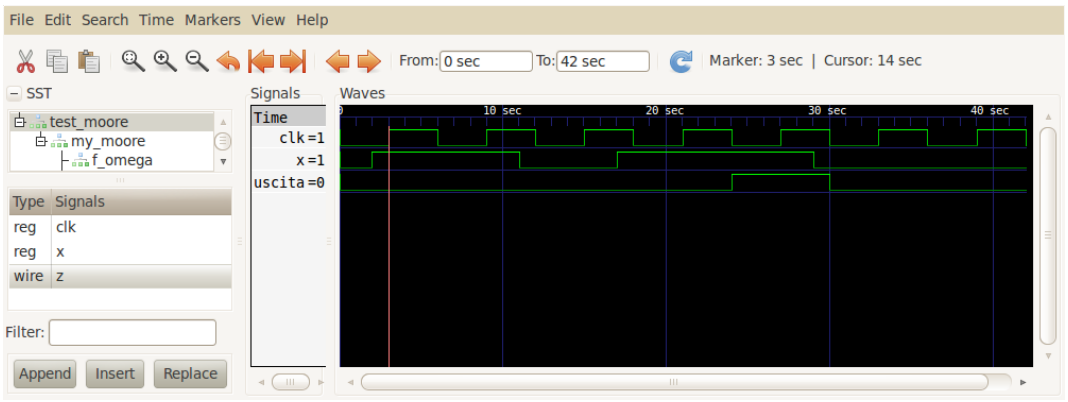

Figura 4.6: Funzionamento della rete sequenziale di Moore che implementa l'automa della Fig. 4.3

```
32
33 endmodule
```

Come si si puó constatare vedendo l'output riportato in Fig. 4.6, il risultato é lo stesso ottenuto mediante l'automa implementato secondo il modello behavioural.

Capitolo 5

Reti sequenziali realizzate con componenti standard

In questo capitolo, vediamo come possiamo modellare reti sequenziali realizzate come combinazione di componenti standard.

5.1 Rete A: pag. III.57 del libro di testo

Consideriamo la realizzazione della "Rete A" presentata a pagina III.57 del libro di testo e di cui riportiamo la struttura in Fig. 5.1.

Utilizziamo i componenti giá descritti nelle precedenti sezioni di questa dispensa. In questo caso, peró facciamo in modo che tutti i moduli utilizzati realizzino i ritardi tipici considerati nel libro di testo. In particolare:

- per la realizzazione del registro A utilizziamo un modulo simile al modulo "registro" del Listato 3.19
- per la realizzazione dei commutatori K_1 e K_2 utilizziamo un modulo simile al modulo "commutatore_nbit" del Listato 3.7, modificato in modo da introdurre un ritardo da $2t_p$ (considereremo che $1t_p = 1\ nsec$, da ora in avanti) (vedi Listato 5.1).
- per la ALU, utilizziamo un componente simile alla ALU a 4 operazioni "alu" del Listato 3.17. La nostra variante avrá solo due operazioni (di somma e sottrazione) e un solo flag (segno) e introdurrá un ritardo di $10t_p$ (vedi Listato 5.2).

Listing 5.1: Commutatore a due ingressi da N bit ciascuno con ritardo di $2t_p$

```
1  module commutatore_nbit(z, x, y, alpha);
2
3    parameter N = 32;
4
5    output [N-1:0]z;
6    input [N-1:0]x;
7    input [N-1:0]y;
8    input alpha;
9
10   assign #2 z = ((~alpha) ? x : y);
11
12 endmodule
```

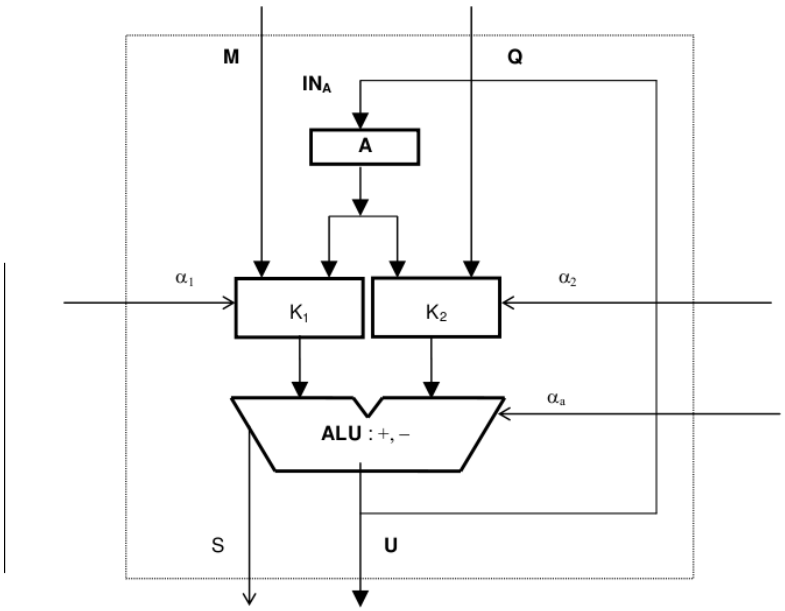

Figura 5.1: “Rete A”

Listing 5.2: ALU con due operazioni con ritardo di $10t_p$

```
1 module alu(risultato,segno,ingresso1,ingresso2,alpha);
2
3   parameter N = 32;   // larghezza della parola
4
5   output reg [N-1:0]risultato;
6   output reg        segno;
7
8   input  [N-1:0]ingresso1;
9   input  [N-1:0]ingresso2;
10  input         alpha;    // 2 operazioni possibili
11
12  always @(alpha or ingresso1 or ingresso2)
13  begin
14   case(alpha)
15     0 : #10 risultato = ingresso1+ingresso2; // somma
16     1 : #10 risultato = ingresso1-ingresso2; // sottrazione
17   endcase
18   segno = risultato[N-1];
19  end
20
21 endmodule
```

Il modulo che realizza la rete sequenziale “ReteA” utilizzando i componenti standard appena definiti é riportato nel Listato 5.3.

Listing 5.3: Modulo che implementa la “ReteA” di pag. III.57 del libro di testo

```
1 `timescale 1ns / 1ps
2
3 module retea();
4
5    parameter N = 32;
```

```
6
7     // ingressi
8     reg    [N-1:0] M;
9     reg    [N-1:0] Q;
10    // ingressi di controllo
11    reg            alpha1;
12    reg            alpha2;
13    reg            alpha3;
14    reg            betaA; // sempre a 1
15    // collegamenti fra moduli standard
16    wire   [N-1:0] uscitaK1;
17    wire   [N-1:0] uscitaK2;
18    wire   [N-1:0] uscitaRegA;
19    // uscite
20    wire           S;
21    wire   [N-1:0] U;
22    // clock
23    reg            clock;
24
25    // layout della rete sequenziale
26    registro         #(N) registroA(uscitaRegA,clock,betaA,U);
27    commutatore_nbit #(N) commK1(uscitaK1,M,uscitaRegA,alpha1);
28    commutatore_nbit #(N) commK2(uscitaK2,uscitaRegA,Q,alpha2);
29    alu              #(N) alu(U,S,uscitaK1,uscitaK2,alpha3);
30
31   // generazione del clock
32   always
33   begin
34     if(clock)
35   #1 clock = ~clock;
36     else
37   #14 clock = ~clock;
38   end
39
40   // modulo per il test
41   initial
42   begin
43     $dumpfile("dump.vcd");
44     $dumpvars;
45
46     betaA = 1'b1;
47     clock = 1'b0;
48
49     M = 15;
50     Q =  5;
51     alpha1 = 0;
52     alpha2 = 1;
53     alpha3 = 1;
54
55     #20 alpha1 = 1;
56     #20 alpha2 = 1;
57     #17 alpha3 = 0;
58         Q = -6;
59
60     #30 $finish;
61
62   end
63
64 endmodule
```

Gli ingressi (dati e controllo) sono definiti alle linee 7–14. Abbiamo definito anche un ingresso di controllo betaA, per controllare le scritture nel registro A. Questo ingresso di controllo non é presente in Figura 5.1 dal momento che stiamo assumendo che la scrittura nel registro avvenga ad ogni ciclo clock, e quindi betaA é sempre 1 nel nostro listato. Le linee 16–18 definiscono i "fili" che collegano le varie parti del sistema, mentre

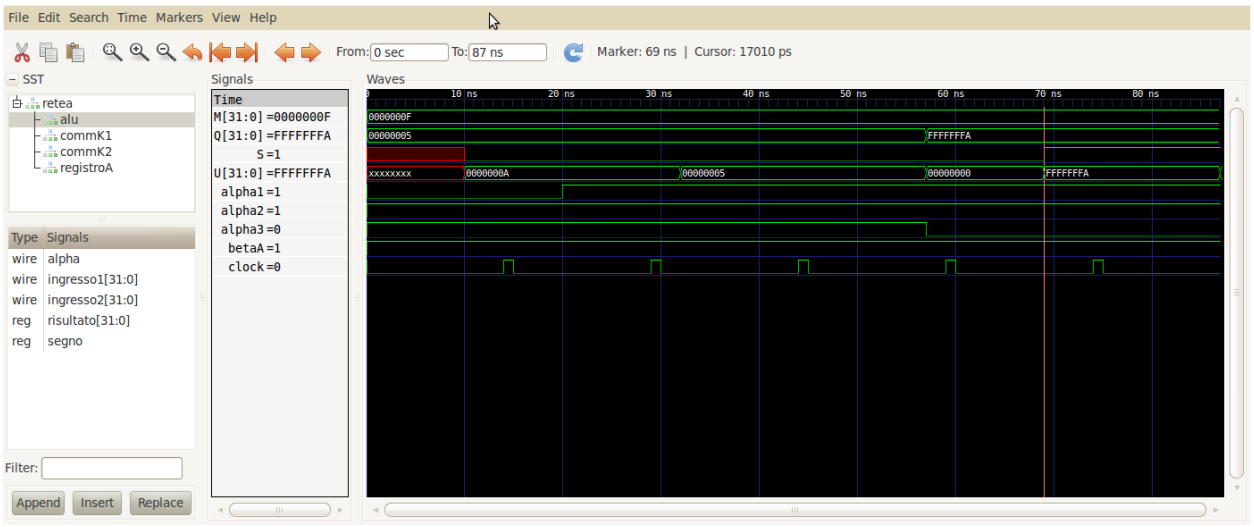

Figura 5.2: Output della simulazione della “Rete A”

le righe 20–21 definiscono le uscite vere e proprie della rete sequenziale. Da notare che l'uscita U viene anche utilizzata come ingresso del registro A.

Le linee 25–29 definiscono la struttura della rete sequenziale. Vengono instanziati quattro moduli (il registro, i due commutatori e la ALU) e si utilizzano ingressi, uscite e fili di collegamento per realizzare una rete esattamente identica a quella rappresentata in Fig. 5.1.

Le linee 31–38 definiscono il segnale di clock. In questo caso abbiamo stabilito che il ciclo di clock sia di $15nsec$, 14 per la stabilizzazione delle reti combinatorie e 1 per il δ.

Infine, le line 40–62 definiscono il modulo di test. In questo caso facciamo vedere solo un paio di operazioni. Inizialmente, eseguiamo una sottrazione fra l'ingresso M e l'ingresso Q (K1 ha un ingresso di controllo pari a 0 (selezione ingresso sinistro), K2 lo ha a 1 (selezione ingresso destro), la ALU ha anch'essa un ingresso di controllo a 1 (sottrazione)).

Nella Fig. 5.2 che fa vedere la simulazione dell'esecuzione mediante GTKWave, possiamo notare come questa prima operazione, notiamo come dopo $10nsec$ l'uscita U si stabilizza al valore 10. In realtá la stabilizzazione dovrebbe avvenire dopo $12nsec$, visto che i commutatori introducono un ritardo di $2t_p$ e la ALU ne introduce uno di $10t_p$. Questo non avviene perché siamo in fase di inizializzazione del simulatore e il trigger che fa partire la valutazione della ALU é in realtá l'inizializzazione dell'alpha1 a 1, anziché l'arrivo dei valori dal uscitaK1 e uscitaK2.

Successivamente, variamo alpha1 in modo da passare alla ALU come primo ingresso l'uscita di K1 anziché l'ingresso M. A questo punto, dopo $12nsec$ abbiamo correttamente la stabilizzazione del nuovo valore dell'uscita, che verrá successivamente scritto nel registro A al prossimo ciclo di clock (ai $44nsec$ nel grafico).

Dunque calcoliamo ancora un sottrazione (alpha3=1) fra il registro A e Q (alphaK1 = 1 e alphaK2 = 1), che porta al risultato (corretto) 0 dopo la stabilizzazione della rete (a $57nsec$ nel grafico).

Lasciamo come esercizio per il lettore la possibilit di variare i parametri del modulo di test in modo da realizzare altre sequenze di operazioni.

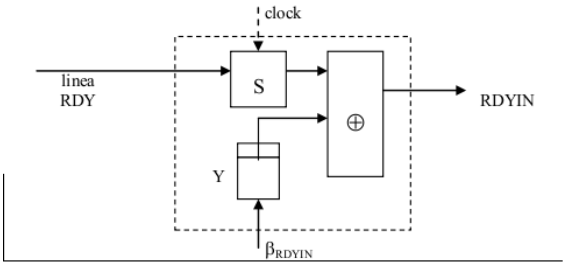

Figura 5.3: Sincronizzatore RDY

5.2 Indicatore di interfaccia d'ingresso (RDY)

Consideriamo adesso il sincronizzatore RDY come descritto a pagina IV.130 del libro di testo e riportato in Fig. 5.3. I componenti standard registro, comparatore e contatore modulo 2 sono definiti nei listati 3.19, 5.4 e 3.13.

Listing 5.4: Comparatore (0 in uscita per ingressi identici e 1 per ingressi diversi)

```
1 module comparatore(input wire i1, input wire i2, output wire o);
2
3   assign o = ~ ((i1 & i2) | ( (~i1) & (~i2) ));
4
5 endmodule
```

Listing 5.5: Sincronizzatore RDY

```
1 module rdy(rdyin,rdy,clock,beta);
2
3   output rdyin;   // segnale da testare
4   input  rdy;     // ingresso rdy da altra unita
5   input  clock;   // segnale di clock
6   input  beta;    // segnale di reset del RDYIN
7
8   wire   uscitaContatore;
9   wire   uscitaRegistro;
10
11  contatoreModulo2 Y(beta,clock,uscitaContatore);
12  comparatore oplus(uscitaContatore,uscitaRegistro,rdyin);
13  registro #(1) S(uscitaRegistro,clock,1'b1,rdy);
14
15 endmodule
```

Disponendo di questi componenti possiamo definire il sincronizzatore RDY utilizzando il codice del Listato 5.5. Come si vede dal codice, non si fa altro che definire i "wire" per collegare il registro di ingresso ed il contatore modulo 2 al confrontatore e utilizzare questi wire, insieme agli ingressi e alla uscita, per realizzare lo schema di Fig. 5.3. Da notare che il registro di ingresso é definito come registro da 1 bit (parametro `#(1)` passato durante l'instanziazione, e che il beta di detto registro é sempre a 1 (parametro `1'b` passato al posto del beta nei parametri attuali dell'instanziazione del modulo registro.

Il semplice programma di test riportato nel Listato 5.6 (e i cui risultati sono evidenziati in Fig. 5.4 mostra il corretto funzionamento del sincronizzatore.

Listing 5.6: Test del sincronizzatore RDY

```
1 `timescale 1ns / 1ps
```

```
2
3 module test();
4
5    // ingressi
6    reg   lineaRdy;
7    reg   betaRdyIn;
8    // uscite
9    wire  rdyIn;
10   // clock
11   reg   clock;
12
13   // layout della rete sequenziale
14   rdy rdy(rdyIn,lineaRdy,clock,betaRdyIn);
15
16  // generazione del clock
17  always
18  begin
19    if(clock)
20  #1 clock = ~clock;
21    else
22  #4 clock = ~clock;
23  end
24
25  // modulo per il test
26  initial
27  begin
28    $dumpfile("dump.vcd");
29    $dumpvars;
30
31    clock = 0;
32
33    lineaRdy = 0;
34    betaRdyIn = 0;
35
36    #6 lineaRdy = 1;
37
38    #10 betaRdyIn = 1;
39    #7 betaRdyIn = 0;
40
41    #4 lineaRdy = 0;
42    #5 betaRdyIn = 1;
43    #5 betaRdyIn = 0;
44
45    #30 $finish;
46
47  end
48
49 endmodule
```

Il segnale in ingresso passa da 0 a 1 al tempo $6nsec$. Al fronte di discesa del ciclo di clock (al tempo $10nsec$) il segnale di uscita del sincronizzatore (il bit RDY_{in} che testiamo nel microcodice, di fatto) registra l'avvenuta segnalazione del ready passando a 1. A $16nsec$ mettiamo a 1 l'ingresso di controllo del contatore modulo 2 (sarebbe un "reset RDY_{in}" nel microcodice). Sul fronte di discesa del prossimo segnale di clock, l'uscita del sincronizzatore viene posta a 1 anche se il segnale di ingresso rimane a 1 come da specifica del sincronizzatore. A $27nsec$ il segnale di ingresso cambia, e al successivo ciclo di clock il segnale di uscita ritorna alto. Il segnale di reset inviato in immediata successione fa sí che il segnale in uscita RDY_{in} venga immediatamente riposto a 0 al successivo ciclo di clock.

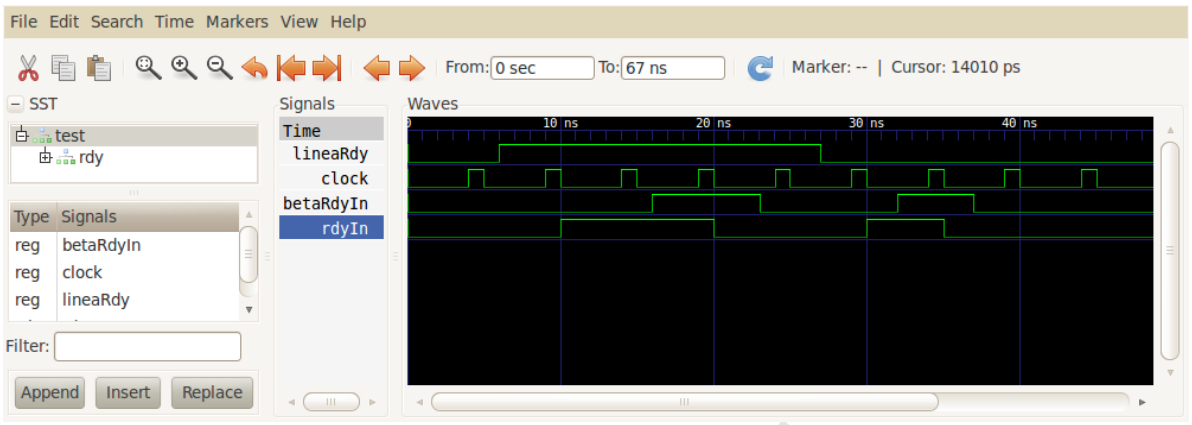

Figura 5.4: Test dell'indicatore di interfaccia di ingresso (RDY)

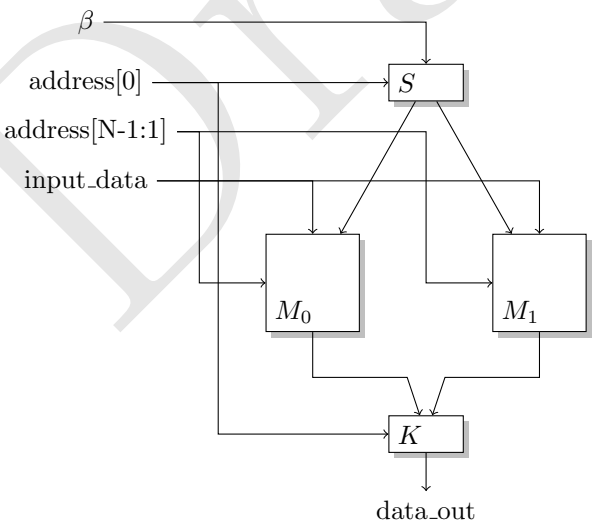

Figura 5.5: Memoria modulare interallacciata (due moduli da 1K parole gestiti come una memoria da 2K parole)

5.3 Indicatore di interfaccia di uscita (ACK)

Gli indicatori di interfaccia di uscita sono semplici contatori modulo 2 con un solo ingresso (β). Ogni volta che l'ingresso vale 1 in corrispondenza del ciclo di clock, effettuano un incremento modulo 2, ovvero passano da 0 a 1 o da 1 a 0.

Un indicatore di interfaccia di uscita si puó quindi modellare come illustrato nel Listato 5.8. La Fig. **??** mostra il comportamento di un indicatore di interfaccia di uscita (il modulo di test é quello del Listato **??**.

Listing 5.7: Indicatore di interfaccia di uscita

```
1  module ack(ackout, clock, beta);
2
3    output reg ackout;
4    input      clock;
5    input      beta;
6
7    initial
8    begin
9      ackout = 0;
10   end
11
12   always @(negedge clock)
13   begin
14     if(beta==1)
15       ackout = ~ ackout;
16   end
17
18 endmodule
```

Listing 5.8: Indicatore di interfaccia di uscita

```
1  `timescale 1ns / 1ps
2
3  module test();
4
5     // ingressi
6     reg   beta;
7     // uscite
8     wire  ackout;
9     // clock
10    reg   clock;
11
12    // layout della rete sequenziale
13    ack ack(ackout,clock,beta);
14
15   // generazione del clock
16   always
17   begin
18     if(clock)
19   #1 clock = ~clock;
20     else
21   #4 clock = ~clock;
22   end
23
24   // modulo per il test
25   initial
26   begin
27     $dumpfile("dump.vcd");
28     $dumpvars;
29
30     clock = 0;
31
32     beta = 0;
33
```

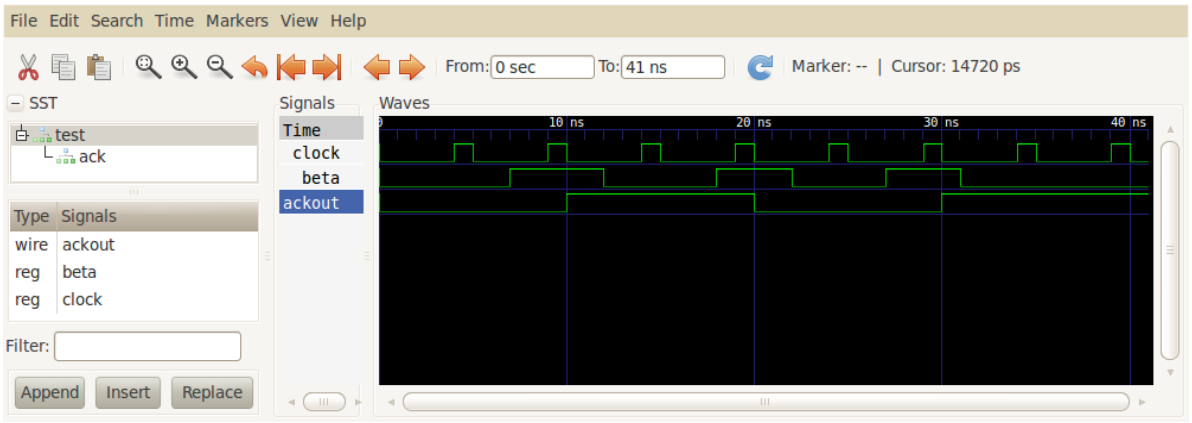

Figura 5.6: Test dell'indicatore di interfaccia di uscita

```
34      #7 beta = 1;
35      #5 beta = 0;
36      #6 beta = 1;
37      #4 beta = 0;
38      #5 beta = 1;
39      #4 beta = 0;
40
41
42      #10 $finish;
43
44    end
45
46  endmodule
```

5.4 Memoria modulare

Supponiamo di voler modellare mediante Verilog una memoria modulare interallacciata di 2K parole costruita utilizzando 2 moduli da 1K parola l'uno. Lo schema della memoria modulare sará quello della Fig. 5.5.

Il selettore S provvede a passare il β che abilita la scrittura al primo o al secondo modulo, a seconda del bit meno significativo dell'indirizzo passato alla memoria modulare. Questo bit serve anche da segnale di controllo (α_k) per il commutatore K. Vista la semplicitá del selettore, lo abbiamo sostituito con due piccole reti combinatorie che calcolano $\beta_{M0} = \sim$ `address[0]` & β e $\beta_{M1} =$ `address[0]` & β. Il risultato é il Listato 5.9.

Listing 5.9: Memoria modulare interallacciata

```
1  module memModInter(data_out, clock, write_enable, address, data_in);
2
3    parameter N=11;
4    parameter M=32;
5
6    input          clock;
7    input          write_enable;
8    input  [N-1:0] address;
9    input  [M-1:0] data_in;
10
11   output [M-1:0] data_out;
12
```

```
13   wire           modSel0;
14   wire           wr0;
15   wire           wr1;
16   wire   [M-1:0] data_out_0;
17   wire   [M-1:0] data_out_1;
18
19   // wr0 -> segnale di abilitazione alla scrittura nel primo modulo
20   not(modSel0,address[0]);
21   and(wr0,modSel0,write_enable);
22   // wr1 -> segnale di abilitazione alla scrittura nel secondo modulo
23   and(wr1,address[0],write_enable);
24
25   // definizione dei due moduli
26   ram #(10,M) modulo0(data_out_0,clock,wr0,address[N-1:1],data_in);
27   ram #(10,M) modulo1(data_out_1,clock,wr1,address[N-1:1],data_in);
28   // commutatore che comanda la lettura
29   k  #(M) k1(data_out,data_out_0,data_out_1,address[0]);
30
31 endmodule
```

Il modulo "ram" di memoria e il commutatore sono quelli dei Listati 3.20 e 5.1, rispettivamente. I fili definiti alle linee 13–17 servono per portare i segnali necessari al funzionamento delle porte utilizzate (linee 19–23) per sostiuire il selettore per la gestione del β di abilitazione alla scrittura e per collegare le uscite dei due moduli di memoria agli ingressi del commutatore che genera il valore "letto" in uscita. Si noti che non sono stati utilizzati ritardi in questi listati, per semplicitá.

Il Listato 5.10 riporta il modulo di test per questa memoria modulare, e la Fig. 5.7 mostra i risultati ottenuti facendo girare questo modulo.

Listing 5.10: Modulo di test per la memoria modulare interallacciata

```
1 `timescale 1ns / 1ps
2
3 module test();
4
5    // ingressi
6    reg   [10:0] addr;
7    reg   [31:0] dato_in;
8    reg          WrEn;
9    // uscite
10   wire  [31:0] dato_out;
11   // clock
12   reg   clock;
13
14   // layout della rete sequenziale
15   memModInter mm(dato_out,clock,WrEn,addr,dato_in);
16
17   // generazione del clock
18   always
19   begin
20     if(clock)
21   #1 clock = ~clock;
22     else
23   #9 clock = ~clock;
24   end
25
26   // modulo per il test
27   initial
28   begin
29     $dumpfile("dump.vcd");
30     $dumpvars;
31
32     clock = 0;
33
34     dato_in = 7;
```

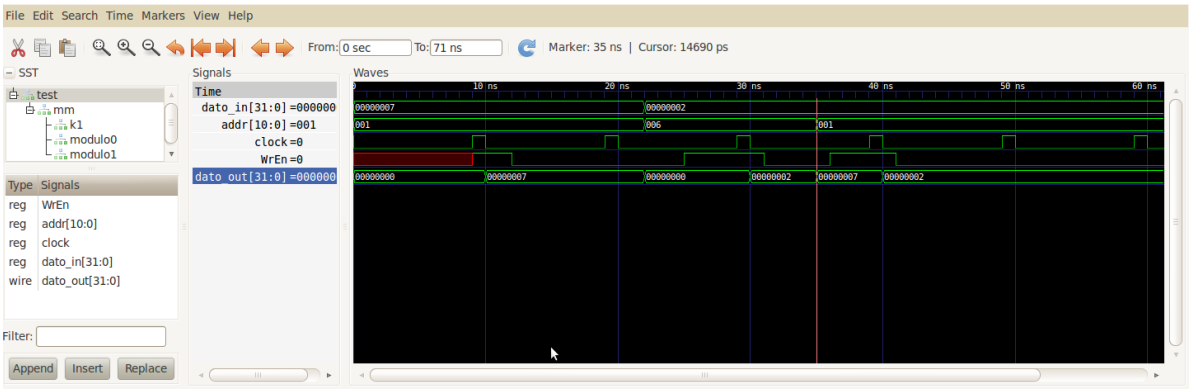

Figura 5.7: Test della memoria modulare

```
35      addr = 1;
36      #9 WrEn = 1;
37      #3 WrEn = 0;
38
39
40      #10 dato_in = 2;
41      addr = 6;
42      #3 WrEn = 1;
43      #6 WrEn = 0;
44
45      #4 addr = 1;
46      #1 WrEn = 1;
47      #5 WrEn = 0;
48
49      #30 $finish;
50
51    end
52
53 endmodule
```

5.5 Memorietta associativa

Modelliamo utilizzando Verilog una memorietta associativa di P sole posizioni. Le chiavi avranno lunghezza C e la parola sará di W bit.

Ci servono un certo numero (P) di confrontatori, capaci di eseguire il confronto fra la chiave fornita in ingresso e le chiavi memorizzate nelle P posizioni della memorietta associativa. Nel Listato 5.4 abbiamo visto come si modella un comparatore da 1 bit. Possiamo utilizzare questo modulo per realizzare un comparatore da N bit. A tale scopo, utilizziamo una tecnica simile a quella utilizzata per il commutatore da N bit (Listato 3.8). Inoltre, utilizziamo la possibilitá fornita da Verilog di implementare "reduce" di parole mediante operatori booleani. Una reduce é un'operazione su un vettore di dati $x_1, \ldots, x_n$ che, dato un operatore $\oplus$ calcola $x_1 \oplus x_2 \oplus \ldots \oplus x_n$.

I Listati 5.11 e 5.12 mostrano l'implementazione del confrontatore da un bit (come primitiva, fornendone la tabella della veritá) e del confrontatore da N bit (metodo generative + reduce).

Listing 5.11: Confrontatore da 1 bit

```
1 primitive confrontatore(output z,
2     input x, input y);
```

```
3   table
4     0 0 : 0;
5     0 1 : 1;
6     1 0 : 1;
7     1 1 : 0;
8   endtable
9
10 endprimitive
```

Listing 5.12: Confrontatore da N bit

```
1 module comparatore(r,x,y);
2
3   parameter N = 32;
4
5   input  [N-1:0]x;
6   input  [N-1:0]y;
7   output        r;
8
9   wire   [N-1:0]z;
10
11  genvar i;
12
13  generate
14    for(i=0; i<N; i=i+1)
15    begin
16      confrontatore t(z[i],x[i],y[i]);
17    end
18  endgenerate
19
20  assign r = | z;
21
22 endmodule
```

A questo punto possiamo cominciare a modellare la parte della memoria associativa che confronta le chiavi con la chiave data e genera il segnale di fault e l'eventuale indirizzo individuato dalla chiave nella memoria dei dati. Il modulo conterrá la memoria delle chiavi, i confrontatori e dovrá mettere a disposizione come "operazioni esterne" la possibilitá di ricercare una chiave o di memorizzarla in una certa posizione.

La realizzazione di questo modulo é presentata nel Listato 5.13.

Listing 5.13: Modulo gestione chiavi

```
1 module unifault(clock,op,ind,chiave,fault,indout);
2
3   parameter P = 2;
4   parameter C = 16;
5   parameter W = 32;
6
7   input                clock;    // segnale di clock
8   input                op;       // op=1 scrivi chiave a ind, =1 leggi chiave
9   input        [P-1:0] ind;      // indirizzo per scrivere chiave
10  input        [C-1:0] chiave;   // chiave da scrivere
11  output               fault;    // segnale di chiave non presente
12  output reg   [P-1:0] indout;   // indirizzo della chiave (se trovata)
13
14
15  reg    [C-1:0]    chiavi[2**P-1:0];   // memoria delle chiavi
16  wire   [2**P-1:0] confronti;    // uscite dei confrontatori
17
18  // blocco dei confrontatori: metodo generativo
19  genvar i;
20  generate
21    for(i=0; i<2**P; i=i+1)
22      comparatore #(C) c(confronti[i],chiavi[i],chiave);
```

```
23   endgenerate
24
25   // generazione del segnale di fault
26   assign
27     fault = & confronti;
28
29   // generazione dell'indirizzo della chiave
30   reg    [P-1:0]    indtemp;
31   reg    [2**P-1:0] indl;
32   always @(negedge clock)
33   begin
34     case (op)
35     1'b0: // read
36       begin
37         indtemp = 0;
38         indl = confronti;
39         while(indl[0]==1)
40         begin
41           indtemp = indtemp + 1;
42           indl = indl >> 1;
43         end
44         indout <= indtemp;
45       end
46     1'b1: // write
47       begin
48         chiavi[ind] = chiave;
49       end
50     endcase
51   end
52
53 endmodule
```

Il modulo é parametrico rispetto al numero di posizioni della memoria associativa (sono 2^P, usando P come numero di bit dell'indirizzo per la memoria che contiene le chiavi), alla lunghezza in bit delle chiavi (C) e alla lunghezza della parola utilizzata per le informazioni (W).

La memoria delle chiavi (dichiarata alla linea 15) viene utilizzata come un insieme di registri indipendenti nello statement `generate` delle righe 20–23, in modo da instanziare un comparatore da C bit per posizione. L'uscita dei comparatori, opportunamente "ridotta" mediante un operatore AND definisce il valore del segnale di fault (assign alle linee 26–27).

Le due "operazioni esterne" del modulo sono definite nel blocco `always` delle linee 32–51. Il primo ramo del case (etichetta `1'b0`) calcola l'indirizzo contando in quale posizione si trova il bit a 0 nel vettore dei risultati dei confrontatori fra chiave in ingresso e chiavi memorizzate. Il secondo ramo del case (etichetta `1'b1`) memorizza semplicemente la chiave presentata in ingresso nella posizione indicata dall'indirizzo (anch'esso) in ingresso, e serve quindi per l'inizializzazione dei valori delle chiavi.

Il corretto funzionamento del modulo é stato testato con il programma di test del Listato 5.14, il cui output é riportato in Listato 5.15 (terminale) e in Fig. 5.8 (GtkWave).

Listing 5.14: Test del modulo gestione chiavi

```
1 `timescale 1ns / 1ps
2
3 module prova_unifault();
4
5   reg clock;
6
7   parameter P = 2;  // bit di indirizzo della mem associativa
8   parameter C = 16;   // bit delle chiavi
9   parameter W = 32;   // bit delle parole
10
11  reg   [P-1:0] indin;  // indirizzo in ingresso
12  reg           op;   // operazione da fare
13  reg   [C-1:0] chiave; // chiave da utilizzare
```

```
14
15   wire          fault;   // segnale di fault
16   wire  [P-1:0] indout;  // indirizzo del dato in mem
17
18   unifault #(P,C,W) uf(clock, op, indin, chiave, fault, indout);
19
20   always
21   begin
22     if(clock==1)
23       #1 clock = ~clock;
24     else
25       #5 clock = ~clock;
26   end
27
28   initial
29   begin
30     clock = 0;
31
32     $dumpfile("dump.vcd");
33     $dumpvars;
34
35     $monitor("op %b ind %b ch %b -> fault %b ind %b \n",
36        op, indin, chiave, fault, indout);
37
38     op = 1;   // scrittura
39     chiave = 13;
40     indin = 0;
41
42     #7
43     chiave = 7;
44     indin = 1;
45
46     #6
47     chiave = 17;
48     indin = 2;
49
50     #6
51     chiave = 3;
52     indin = 3;
53
54     // iniziano le letture
55     #6
56     op = 0;
57     chiave = 7;
58
59     #6
60     chiave = 8;
61
62     #6
63     chiave = 17;
64
65     #10 $finish;
66   end
67
68 endmodule
```

Listing 5.15: Risultato del test del modulo gestione chiavi

```
1 VCD info: dumpfile dump.vcd opened for output.
2 op 1 ind 00 ch 0000000000001101 -> fault 0 ind xx
3
4 op 1 ind 01 ch 0000000000000111 -> fault x ind xx
5
6 op 1 ind 01 ch 0000000000000111 -> fault 0 ind xx
7
```

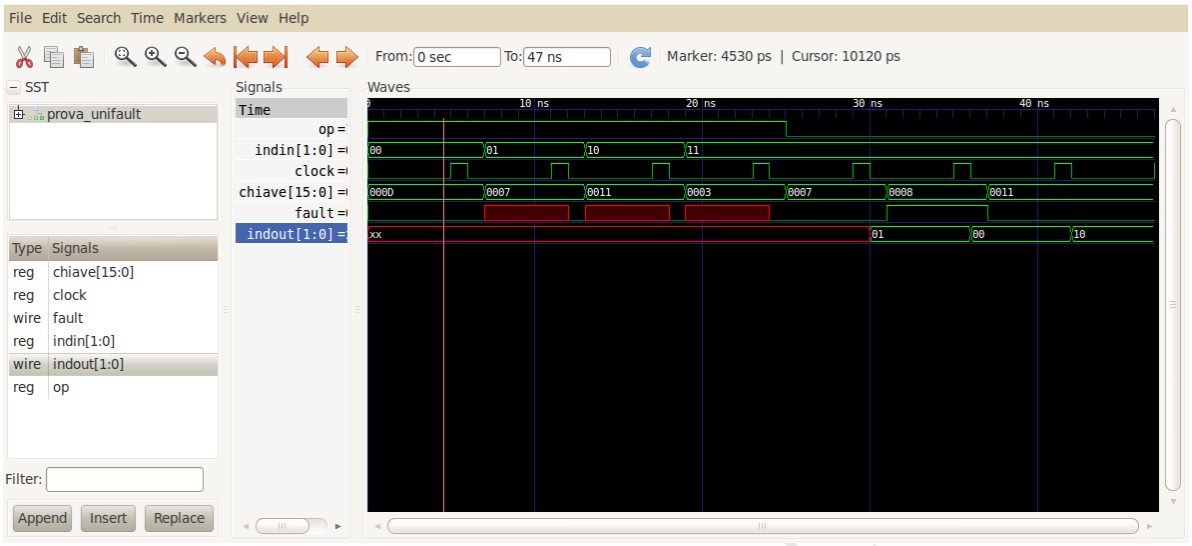

Figura 5.8: Test del modulo gestione chiavi

```
 8 op 1 ind 10 ch 0000000000010001 -> fault x ind xx
 9
10 op 1 ind 10 ch 0000000000010001 -> fault 0 ind xx
11
12 op 1 ind 11 ch 0000000000000011 -> fault x ind xx
13
14 op 1 ind 11 ch 0000000000000011 -> fault 0 ind xx
15
16 op 0 ind 11 ch 0000000000000111 -> fault 0 ind xx
17
18 op 0 ind 11 ch 0000000000000111 -> fault 0 ind 01
19
20 op 0 ind 11 ch 0000000000001000 -> fault 1 ind 01
21
22 op 0 ind 11 ch 0000000000001000 -> fault 1 ind 00
23
24 op 0 ind 11 ch 0000000000010001 -> fault 0 ind 00
25
26 op 0 ind 11 ch 0000000000010001 -> fault 0 ind 10
```

Con questo modulo a disposizione, possiamo fare due cose:

1. scrivere un ulteriore modulo che implementi la parte relativa alla memorizzazione dei valori associati alle chiavi e quindi integrare in un modulo “memoriaAssociativa” una istanza di entrambi i moduli, opportunamente configurati (mediante i parametri) e collegati (mediante wire di dimensioni opportune), oppure

2. estendere leggermente il modulo appena presentato nel Listato 5.15 per includere un modulo di memoria che possa contenere direttamente anche i valori asociati alle chiavi. Tale memoria verrebbe utilizzata in fase di inserimento di una chiave per memorizzare anche il valore del dato corrispondente e in fase di lettura per trovare il valore del dato da restituire in caso di hit.

Scegliamo la seconda strada (e lasciamo l’altra per esercizio). Il modulo modificato é riportato nel Listato 5.16. Come si vede, sono stati aggiunti due parametri al modulo: il parametro “valore”, in ingresso, per passare il valore da memorizzare insieme ad una chiave in fase di inizializzazione/scrittura (vedi operazione

relativa alle linee 50–54) e il parametro “valout”, significativo solo in caso di assenza di fault e per operazioni di lettura (si veda l’implementazione dell’operazione di lettura alle linee 37–49: a differenza di quanto accadeva nella realizzazione del modulo del Listato 5.13, l’indirizzo “indtemp” viene utilizzato non solo per passarlo in uscita come risultato della lettura ma anche per accedere al valore associato alla chiave (linea 48), successivamente passato come risultato). Nel Listato 5.17 riportiamo il relativo modulo di test e nel Listato 5.18 riportiamo l’output a terminale.

Listing 5.16: Modulo memorietta associativa (completo)

```
1  module assoc(clock,op,ind,val,chiave,fault,indout,valout);
2
3    parameter P = 2;
4    parameter C = 16;
5    parameter W = 32;
6
7    input                clock;    // segnale di clock
8    input                op;       // op=1 scrivi chiave a ind, =1 leggi chiave
9    input        [P-1:0] ind;      // indirizzo per scrivere chiave
10   input        [W-1:0] val;      // valore da associare alla chiave
11   input        [C-1:0] chiave;   // chiave da scrivere
12   output               fault;    // segnale di chiave non presente
13   output reg   [P-1:0] indout;   // indirizzo della chiave (se trovata)
14   output reg   [W-1:0] valout;   // valore associato alla chiave cercata
15
16
17   reg    [C-1:0]    chiavi[2**P-1:0];   // memoria delle chiavi
18   reg    [W-1:0]    valori[2**P-1:0];   // memoria dei valori
19   wire   [2**P-1:0] confronti;    // uscite dei confrontatori
20
21   // blocco dei confrontatori: metodo generativo
22   genvar i;
23   generate
24     for(i=0; i<2**P; i=i+1)
25       comparatore #(C) c(confronti[i],chiavi[i],chiave);
26   endgenerate
27
28   // generazione del segnale di fault
29   assign
30     fault = & confronti;
31
32   // generazione dell'indirizzo della chiave
33   reg    [P-1:0]    indtemp;
34   reg    [2**P-1:0] indl;
35   always @(negedge clock)
36   begin
37     case (op)
38     1'b0: // read
39       begin
40         indtemp = 0;
41         indl = confronti;
42         while(indl[0]==1)
43         begin
44           indtemp = indtemp + 1;
45           indl = indl >> 1;
46         end
47         indout <= indtemp;
48         valout <= valori[indtemp];
49       end
50     1'b1: // write
51       begin
52         chiavi[ind] = chiave;
53         valori[ind] = val;
54       end
55     endcase
56   end
```

```
57
58 endmodule
```

Listing 5.17: Modulo di test per la memorietta associativa

```
1  `timescale 1ns / 1ps
2
3  module prova_assoc();
4
5    reg clock;
6
7    parameter P = 2;  // bit di indirizzo della mem associativa
8    parameter C = 16;   // bit delle chiavi
9    parameter W = 32;   // bit delle parole
10
11   reg   [P-1:0] indin;  // indirizzo in ingresso
12   reg           op;   // operazione da fare
13   reg   [C-1:0] chiave; // chiave da utilizzare
14   reg   [W-1:0] valore; // valore da memorizzare
15
16   wire          fault;   // segnale di fault
17   wire  [P-1:0] indout;  // indirizzo del dato in mem
18   wire  [W-1:0] valout;  // valore letto
19
20   assoc #(P,C,W) uf(clock, op, indin, valore, chiave, fault, indout, valout);
21
22   always
23   begin
24     if(clock==1)
25       #1 clock = ~clock;
26     else
27       #5 clock = ~clock;
28   end
29
30   initial
31   begin
32     clock = 0;
33
34     $dumpfile("dump.vcd");
35     $dumpvars;
36
37     $monitor("op %d ind %d ch %d val %d -> fault %d ind %d val %d \n",
38        op, indin, chiave, valore,fault, indout,valout);
39
40     op = 1;   // scrittura
41     chiave = 13;
42     indin = 0;
43     valore = 4;
44
45     #7
46     chiave = 7;
47     indin = 1;
48     valore = 3;
49
50     #6
51     chiave = 17;
52     indin = 2;
53     valore = 2;
54
55     #6
56     chiave = 3;
57     indin = 3;
58     valore = 1;
59
60     // iniziano le letture
```

```
61      #6
62      op = 0;
63      chiave = 7;
64
65      #6
66      chiave = 8;
67
68      #6
69      chiave = 17;
70
71      #10 $finish;
72    end
73
74  endmodule
```

Listing 5.18: Risultato del test del modulo memorietta associativa

```
1  VCD info: dumpfile dump.vcd opened for output.
2  op 1 ind 0 ch    13 val          4 -> fault 0 ind x val          x
3
4  op 1 ind 1 ch     7 val          3 -> fault x ind x val          x
5
6  op 1 ind 1 ch     7 val          3 -> fault 0 ind x val          x
7
8  op 1 ind 2 ch    17 val          2 -> fault x ind x val          x
9
10 op 1 ind 2 ch    17 val          2 -> fault 0 ind x val          x
11
12 op 1 ind 3 ch     3 val          1 -> fault x ind x val          x
13
14 op 1 ind 3 ch     3 val          1 -> fault 0 ind x val          x
15
16 op 0 ind 3 ch     7 val          1 -> fault 0 ind x val          x
17
18 op 0 ind 3 ch     7 val          1 -> fault 0 ind 1 val          3
19
20 op 0 ind 3 ch     8 val          1 -> fault 1 ind 1 val          3
21
22 op 0 ind 3 ch     8 val          1 -> fault 1 ind 0 val          4
23
24 op 0 ind 3 ch    17 val          1 -> fault 0 ind 0 val          4
25
26 op 0 ind 3 ch    17 val          1 -> fault 0 ind 2 val          2
```

Bibliografia

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

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

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

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

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

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