VERILOG PRIMER

Primer, Introduction and Examples

Table of contents:

Preface	2
Conventions Used	2
UNIX Primer	3
Basic UNIX Instructions	3
About UNIX	3
Verilog Primer	4
UNIX preparation	4
Starting Verilog	4
Entering the description	5
Compiling the description	6
Elaborating the description	6
Simulating the testbench	8
Displaying internal signals	9
The Structure of Verilog Models	10
Verilog Coding Examples	12
The most simple multiplier	12
Multiplier circuit (RTL)	13
Sigma-Delta A/D-Converter	16
Small examples	21

For students designing and testing VLSI integrated circuits at the VLSI laboratory of the Dept. of Electron Devices (V2-324) using the CADENCE Verilog simulator environment on Sun workstations under the UNIX Operating System.

Budapest, the 10. 12. 2002.

Peter Gärtner.

Ez a segédanyag megtalálható: \\FERMI\gaertner\Verilog\vprimer.doc

Preface

This manual is primarily intended for students designing and testing VLSI integrated circuits or parts thereof at the VLSI laboratory of the DED (V2-324) using the CADENCE Verilog simulator environment on Sun workstations under the UNIX Operating System.

For doing this work, first of all, you have to acquire from the system manager a personal user account in the Sun Network with UID and password.

This manual consists of four main parts:

- Primer for UNIX, for persons who have not yet worked with UNIX. It provides the minimum necessary knowledge to have some orientation in the operating system and to start Verilog.
- Primer for Verilog, to start the tool and learn the simplest steps for entering the circuit description and doing the simulation.
- A short introduction to the structure of Verilog models.
- Two full examples of circuits/systems descriptions and a collection of small examples.

In this primer the words *description*, *model* and *module* will be used as synonyms for Verilog code units.

Eventually it should be mentioned, too, what this manual does not comprise: circuit theory and a detailed description of the Verilog language.

Experience with Windows on PCs is of advantage. In spite of running under UNIX the window system of CADENCE shows much similarity with Windows.

Conventions Used

There are several conventions used in this manual. The mouse of the Sun machines has three buttons. In the following there is some terminology explained which will be used in relation to mouse operations.

click left	press and release the left mouse button (quickly)
click middle	press and release the middle mouse button (quickly)
click right	press and release the right mouse button (quickly)
drag left	press and hold the left mouse button while moving the mouse
drag middle	press and hold the middle mouse button while moving the mouse
drag right	press and hold the right mouse button while moving the mouse

If more than one CADENCE window is open then the relevant window will be specified by adding *WWW:* for the window WWW.

If a double target *xxx->yyy* is specified with clicking, that may happen to be two separate clicks at *xxx* and *yyy* or a drag from *xxx* to *yyy*, depending upon how the popup menu for yyy comes up.

<...>	press the key on the keyboard that corresponds to what is inside the brackets (either a character or a special key like CR (carriage return or enter), ESC (escape), SHIFT, CTRL, ALT.
type **something**	you should type (verbatim) whatever is printed boldfaced.

UNIX Primer

Basic UNIX Instructions

(Unix instructions have to be typed in a command ('shell') window. All instructions have to be terminated with <CR>!)

Command		Description
ls	list:	lists elements of a directory by their names
ls –l	list long:	detailed listing of a directory: access right, owner, length, date, name
ls –a	list all:	list including the hidden files too (beginning with '.')
ls –al	list all long:	detailed long listing of all files
ls –lt		long listing ordered by the time of generation
mkdir *dirname*		make directory named *dirname*
rmdir *dirname*		remove (delete) directory *dirname* (only if the directory is empty)
rm *filename*		remove (delete) the file *filename*
rm -r *dirname*		delete the directory *dirname* with all its contents (hierarchical! USE IT WITH CAUTION!!)
du	disk usage	lists the complete hierarchy downwards with size (1 kByte blocks)
cd *subdir*		change directory to *subdir*
cd		change directory to the home directory of the user
textedit *filename*		opens the file *filename* for editing (new file if *filename* does not exist)

About UNIX

After logging in you are at the highest level of your user account. This is your *Home Directory*, which can be referred to by the tilde '~' character. UNIX comes up with an *xterm window* which is mainly for the messages of the operating system and does not have a scroll bar. Left click at the left button in the upper right corner so the window becomes an icon in the bottom bar. Then with a left click a menu pops up. Left click **Shells->Cmdtool**. A command shell will be opened with a vertical scroll bar. This window can be your workhorse as long as you are working direct with UNIX.

The directory where you are can be represented by the dot '**.**', the preceding higher level directory by two dots '**..**'.

Typing **ls -al** you will find among others the file *.cshrc* which contains settings for the operating system. (If it does not yet exist you may open a new one with the editor.) The following three lines show examples for your own usage:

alias lth 'ls -lt | head' If you type **lth** then UNIX will produce a time-ordered list of the ten most recent files - an alias which can be favourably used for checking the recent changes in the directory.

alias ed 'textedit \!*&' Instead of *textedit xxx* you can simply type **ed xxx** and the editor will start with the file *xxx*. The ampersand '&' will make the editor start as a stand-alone process so that your window remains free for other work.

Any change in *.cshrc* will be effective only after your next logging-in.

HINT: If you copy *~gaertner/.cshrc* to your home directory then you will have these and several other features in your account:

cp ~gaertner/.cshrc . <CR>

When already copied, you can add other aliases for your personal usage, too.

Verilog Primer

The objective of this primer is to teach a quick and easy start into the CADENCE system without going into details. Going through this primer some simple circuit model will be simulated making the following steps:

- entering the description
- compiling the description
- elaborating the description
- simulating the testbench.

The description of these basic steps is completed by additional explanations on how to have internal signals displayed on the screen.

UNIX preparation

Create a directory for your Verilog activities on UNIX level, for instance *myveri*:

mkdir myveri<CR>

Then change the directory to it:

cd myveri<CR>

Here create a new directory for your Verilog source files, the best name for it would be *source*:

mkdir source<CR>

The working environment of Verilog is now prepared. If you want to, you may start the text editor by typing **textedit Filename.v** and start writing the Verilog source code of your model. But you can do that inside Verilog, too.

Starting Verilog

Middle click at an empty place of the screen. The **Eng. Tools** popup window opens. Make a left click at **Simulators->Verilog/VHDL**. A new UNIX shell comes up and asks for the Verilog home directory. Type the name of the recently created directory, e.g. **myveri<CR>**. The main window of Verilog *NCLaunch* appears (Fig. 1.). On the top of it you find a menu bar. Click at **Edit->Preferences**. The *Preferences* dialog box appears (Fig. 2.). In the first line check the entry *Editor Command*. If it does not read *textedit %F* then replace it with this command. Thereby you specify the regular text editor of the UNIX operating system. Enter it by clicking at the **OK** button.

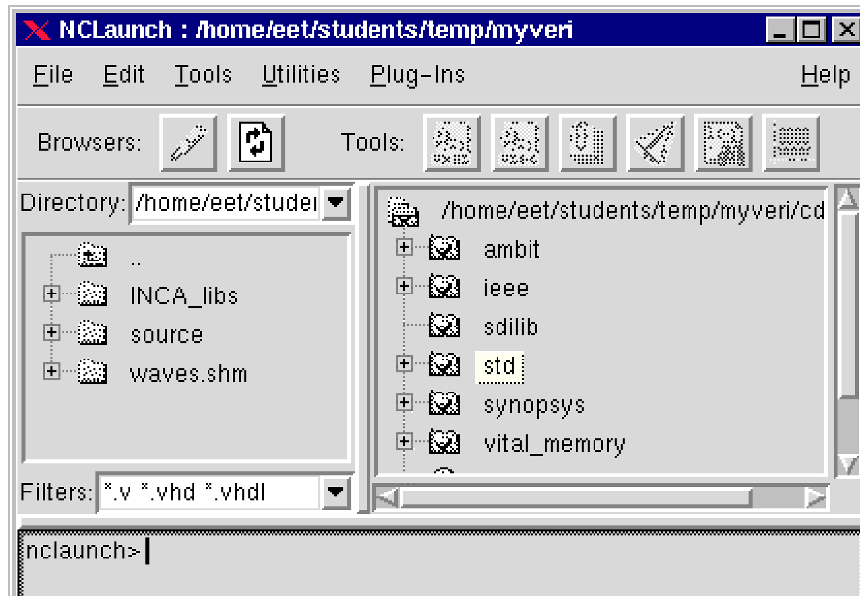

Fig. 1. Verilog main window

Before *NCLaunch* appears CADENCE may ask you the question in another dialog box if you want to use the three-pass procedure or the direct (one-pass) procedure. Choose the three-pass procedure because it provides you with better error checks and easier correction possibilities.

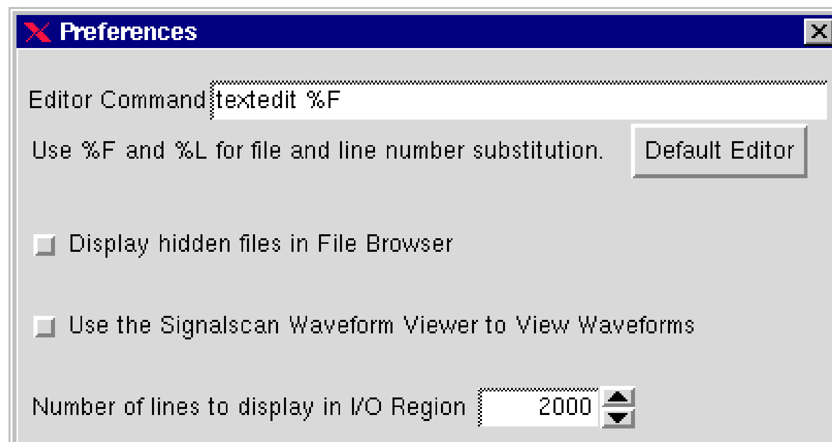

Fig. 2. Specifying the text editor of Unix

Entering the description

Click at **File->Edit New File**. The *Edit a New File* dialog box opens (Fig. 3.). Under its title

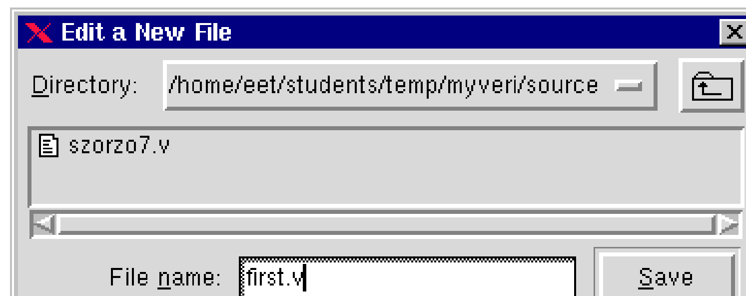

Fig. 3. Specifying the file to be edited

bar it shows the actual directory of the editor, most likely the recently specified *myveri*. Underneath you can see the content of this directory. The only content is probably the subdirectory *source* that you have recently created. Make a left double-click at it. The specified directory will change to *myveri/source* and underneath the content is empty.

Now enter into the box *File name* the name of your first Verilog project, for instance **first.v** and click at the **Save** button. The *Edit a New File* box disappears and the UNIX text editor opens with *first.v* in the title bar. The system is ready to accept your first Verilog project. Enter some simple demo-project, such as *rsln* and *test_rsln* from the pages 10/11. Make a left click at **File->Save** when you have finished the entry. For the time being, the editor is no more necessary, it is left to you to close or to iconify it. (If iconified, you can easier access it if you have to correct some error in the description.)

The aim is to simulate the module described in the file *first.v*. To do so the description has to be processed in two steps, *compilation* and *elaboration*. At this point it is worth while mentioning that compilation is analogous to that of computer programs. Each module is taken one by one and translated into an internal format (such as object files in computers). Elaboration does some kind of linking the modules with each other to form a simulatable unit.

Compiling the description

By a left click select the file *first.v* in the browser pane on the left part of the *NCLaunch* window. The file name will be highlighted. Left click **Tools->Verilog Compiler**. The *Compile Verilog* window opens (Fig. 4.). Check wether the entry in the file box is correct –

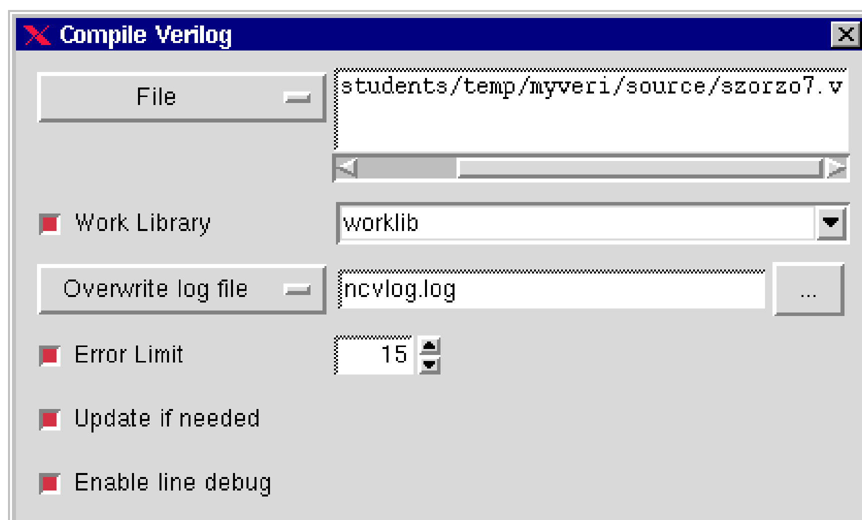

Fig. 4. Compiler window

it should contain the selected file (..../myveri/source/first.v). The checkbox *Work Library* is checked (brown) and the name defaults to *worklib*. Left click at the **OK** button. The compiler starts and writes some message into the command line pane at the bottom of the *NCLaunch* window. If there are errors in your source file *first.v* then you have to correct them in the editor window and, after saving the corrected version, repeat the compilation.

Meanwhile the module browser pane on the right part of the *NCLaunch* window displays the module tree, similar to the file tree on the left. It contains several library entries such as *ieee* and *std*. The last one is your working library *worklib* marked with a reddish-yellow hat. The sub-entries of *worklib* are your new modules which you described in *first.v*.

Elaborating the description

In the main window the + signs in the little box left of the sub-entry indicate that they, too, contain subentries. Click at them and the leafs of the module tree appear, having the simple name *module*. Select by a left click the module on the highest hierarchical level, which is that of the testbench (Fig. 5.).

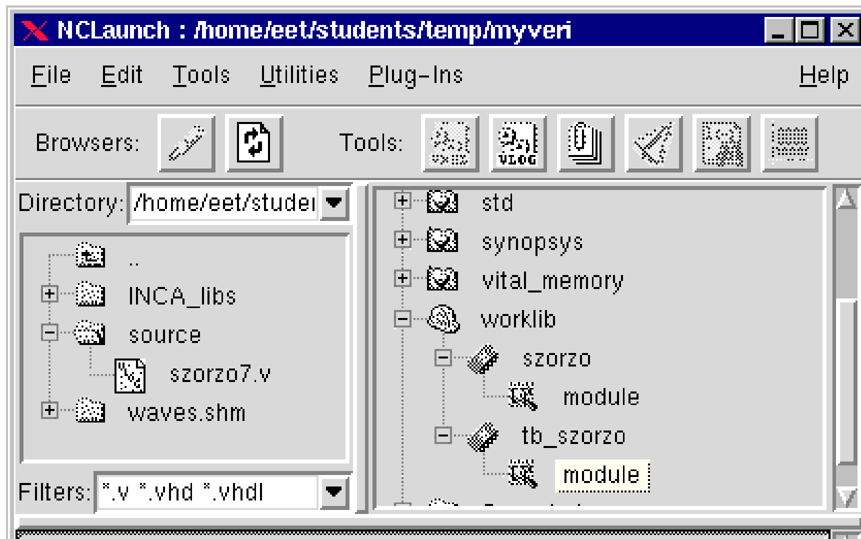

Fig. 5. Selecting the module to be elaborated

Notice that only after you have selected a module, the icon of the elaborator (third from left among the tools in the icon bar, showing sheets of paper and a clip) will be enabled. Now it would be time to click at it, starting the elaborator. However, at the very first elaboration, the procedure is different.

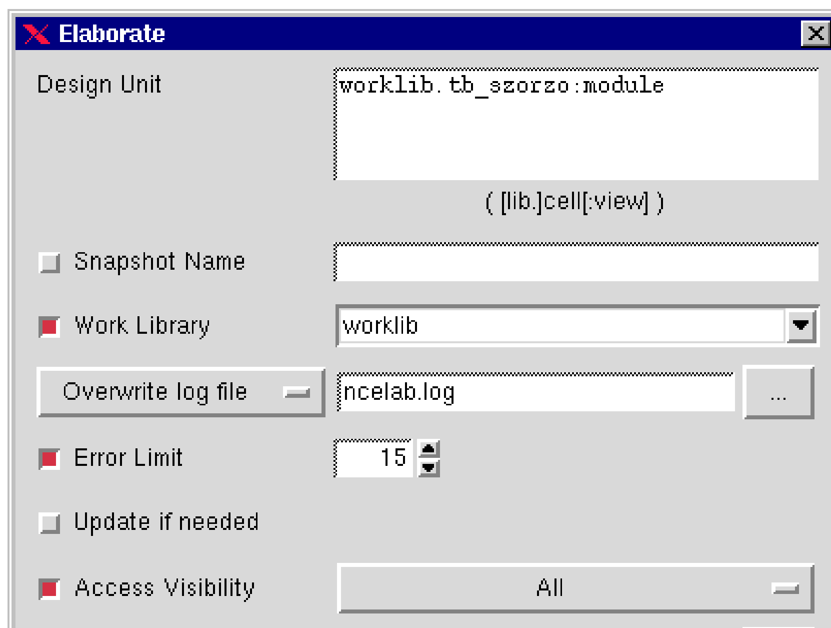

Fig. 6. The window of the elaborator

You have to make a left click at the menu item **Tools->Elaborator** which invokes the dialog box *Elaborate* (Fig. 6.). Check the followings: The *Design Unit* field should contain the name of your module. The checkbox *Work Library* should be checked (brown) and the field contain *worklib*. *Access Visibility* should also be checked and set to *All*. Correct the settings if necessary and click at the **OK** button. The elaborator starts and, after a while, some message appears in the command line pane. Check if there are errors reported.

After successful elaboration you can open the *Snapshots* folder in the module pane and you will find there your testbench module prepared for simulation (Fig. 7.).

Fig. 7. The elaborated module

Simulating the testbench

Select the snapshot of the testbench module by a left click and then click at the icon of the simulator, fourth from left among the tools (or **Tools->Simulator** in the menu bar). The *Cadence NC Verilog* simulator window opens (Fig. 8.).

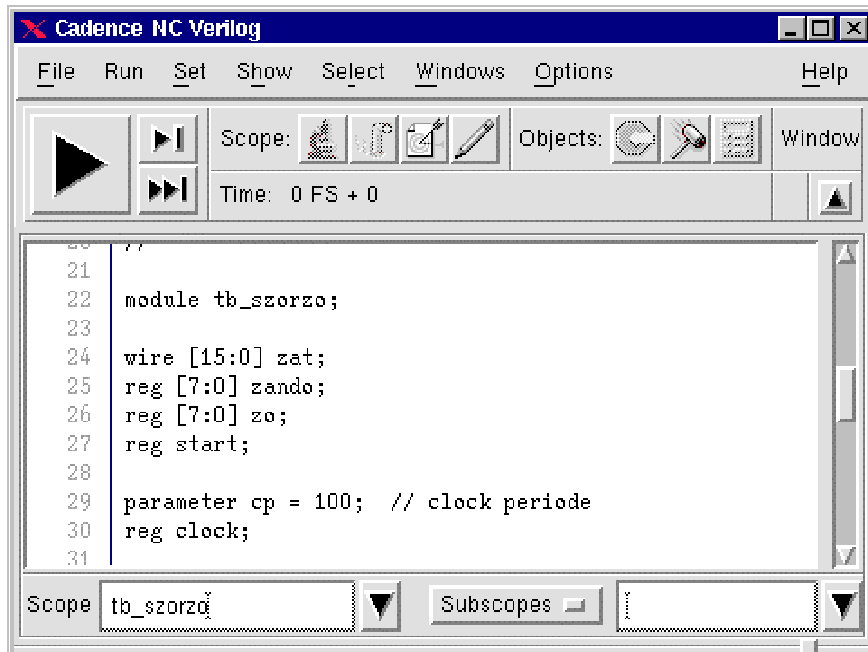

Fig. 8. The simulation window

The upper pane shows the source code of the actual scope, as a default the highest level, that is the testbench itself. Above the pane you can read the simulation time which is, before the start, zero. Later on, if you make several runs, you will have to left click **Cadence NC Verilog:File->Reset Simulation** before each new run.

Left click **Select->Signals** in the menu bar. In the source code all the signals appear highlighted (Fig. 9.). Now left click at **Windows->Waveform**. It takes some time (about 2 to 3 minutes!) till the window **SimVision: Waveform** opens (Fig. 10.). The waveform pane is still empty but the list of signals should be displayed in the left pane.

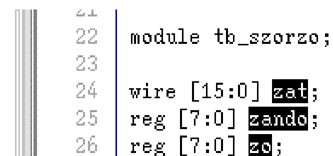

Fig. 9. Highlighted names

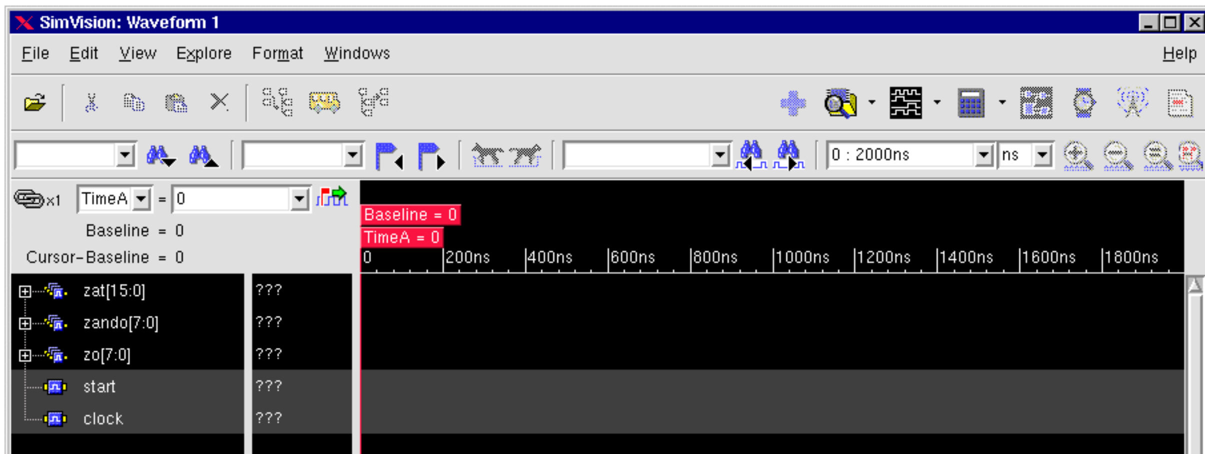

Fig. 10. The waveform window

Now the simulation can be started by a left click on the big black triangle on the left side of the icon bar of the window *Cadence NC Verilog*. The simulation completes very quickly and the waveforms appear in the great pane of the waveform window (Fig. 11.). Above the upper

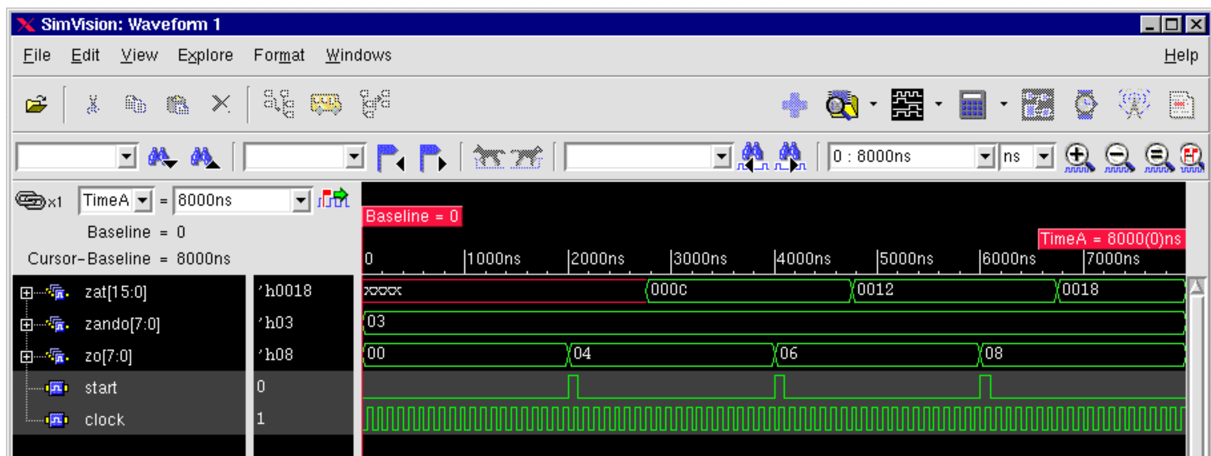

Fig. 11. The result of the simulation

right corner you can find zooming buttons. At the upper left corner there are the small red flags of the cursors. They can be dragged by the left mouse button. In the narrow pane between the signal list and the waveform the signal values can be read, at the simulation time indicated by the cursor *TimeA*.

Displaying internal signals

So far only the signals of the highest level have been displayed. It is possible to go down in the hierarchy and display internal signals as well. Between the two large panes of the window *Cadence NC Verilog* there is the field for selecting the scope of the display. By clicking right of the *Subscopes* box at the small button with the black triangle a dropdown list appears showing the internal modules of the simulated system (Fig. 12.). Selecting one of them the actual source code text appears in the source code pane.

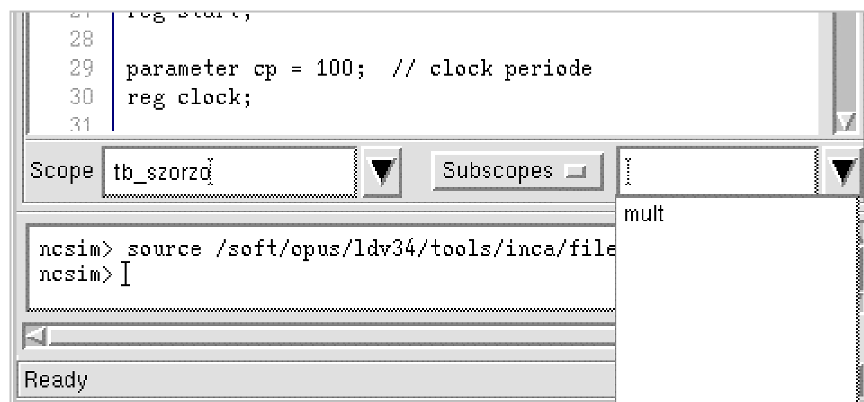

Fig. 12. Down in the hierarchy

If you need all the signals of the internal module then you can select them by means of the menu bar, with a left click at **Select->Signals**. Usually only some of them are needed. In such a case you can left click at them in the source code one by one. From the second one on, however, you have to keep the control key **<CTRL>** depressed. The selected signals will be highlighted. Having selected the signals you can transfer them into the waveform window by left clicking at the waveform button in the icon bar of *SimVision: Waveform 1* (sixth one from right).

The Structure of Verilog Models

In the followings a simplified description of Verilog models is presented. It will help you to build your first Verilog descriptions. The basic unit of Verilog descriptions is the module. It is delimited with the keywords *module* and *endmodule*. It exchanges information with the rest of the world via input and output (or bidirectional inout) ports. The ports have to be given in the port list, and the elements of the list have to be declared if they are input, output or inout. If a port is not only a single signal but a bus or vector then its width has to be declared as well. Accordingly, the frame of the module surrounding the body should look like this:

```
module name(p1, p2, p3, ... pn);
  input p1, p2;
  input [msb1 : lsb1] p3;
  output p4, p5;
  output [msb2 : lsb2] p6;
  ...
  ...
  Body of the module
  ...
  ...
endmodule
```

The body of the module has to do the processing of the input signals to form the outputs. Outputs have to be driven by elements capable of driving. Such elements are:

- registers (abstraction of a flipflop),
- logic gates (generic gate functions),
- continuous assignments (abstraction of a combinational logic function),
- other modules instantiated in the body of the module and having an output which is driven by a driving element inside.

Registers have to be declared. They receive their values assigned in procedural assignments inside procedures such as *initial* and *always*. Their identifiers can be used similarly as wires driving inputs of other elements and the outputs of the module. This construct is illustrated by the following RS-latch description, consisting of a cross-coupled nand pair: (Fig. 13.)

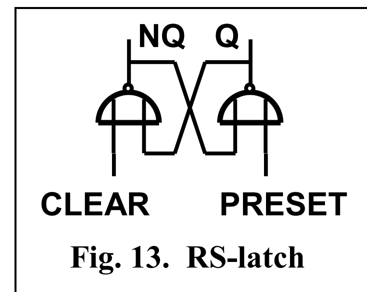

Fig. 13. RS-latch

```
module rsl1(q, qn, preset, clear);
  output q, qn;
  input preset, clear;
  reg q, qn;

  always @(preset or qn)
      #1 q = !(qn && preset);

  always @(clear or q)
      #1 qn= !(q  && clear);

endmodule
```

The outputs of the nand gates *q* and *qn* are declared as registers. Their functions form an *always* procedure each, reacting to any change at their inputs. Both have unit delay.

The same RS-latch can be modelled by *built-in generic gates* of Verilog. In this case wires have to be declared as connecting elements between the parts of the model. The wires cannot drive by themselves but they are driven by the outputs of the gates. The model of the same function is:

```
module rsl2(q, qn, preset, clear);
  output q, qn;
  input preset, clear;
  wire q, qn;

  nand #1        // declare two nand gates with unit delay
    g1(q, qn, preset),
    g2(qn, q, clear);

endmodule
```

In this description of the latch the declaration of the wires *q* and *qn* could be omitted because, by using their names in the context, the compiler recognizes their role as wire and automatically declares them (*implicite wire declaration*).

A third possible description of an RS-latch can be done by means of *continuous assignments*. Here again wires perform the connections and, again, they are in the context *implicitly declared*:

```
module rsl3(q, qn, preset, clear);
  output q, qn;
  input preset, clear;
//  wire q, qn;           Not necessary because of implicite wire declaration

  assign #1 q = !(qn && preset);
  assign #1 qn= !(q  && clear);

endmodule
```

If the model of a circuit (or a function) has been constructed then the next step is verifying it by simulation. For this purpose a testbench is needed, which contains an instance of the model and provides the stimuli.

The testbench is also a module but a special one which does not have in- and outputs. Instead, the test-bench forms the external world for the model to be tested. The driving signals have to be generated here as well as the outputs of the model have to be received and, if necessary, processed. (The necessity may arise if the output signals are in their, in the model generated, "natural" form not directly evaluatable. For instance, often it is easier to eveluate the output of a model after a serial/parallel conversion.) Also, the test-bench provides the load for the outputs of the model - simulating its working environment.

Here follows a testbench for the RS-latch. When making the instance you can choose which model you put into the testbench and simulate, rsl1, rsl2 or rsl3:

```
module rslx_test;
  wire q, qn;         // declare two wires to receive outputs
  reg preset, clear;  // declare two input variables
  parameter d = 10;   // used as the waveform time step

// create an instance of the RS-latch

  rsl1 latch(q, qn, preset, clear);
```

```
// stimulus description - assigns values to inputs

  initial             // runs only once
    begin
         preset = 0; clear = 1;
      #d preset = 1;
      #d clear = 0;
      #d clear = 1;
    end

endmodule
```

Verilog Coding Examples

The following examples illustrate the construction of the description of circuits and functions by means of the Verilog HDL. They are furnished with ample comments for the sake of readability and understandability. For the same purpose indentation has been thoroughly applied. They should help you build your own modules. Two full (simulatable) descriptions are given with test-bench. The multiplier shows two versions of the same problem. The first version is the simplest possible behavioural description of an 8 by 8 bit multiplier. It just multiplies the two numbers producing the 16-bit result. The second version was modelled with practical realizability (and possible synthesis) in mind, and, in addition, the realized (not really but possibly synthesized!) circuit is presented (Figs. 14. and 15.).

The second example is an analog/digital converter (ADC). It is a mixed-signal system in that its input is an analog signal which is connected to an analog comparator. The reference input of the comparator gets feed-back from the output of the comparator integrated by an RC integrator circuit. These parts of the system are described only on the behavioural level. The rest of the system is fully digital and is modelled again with synthesis in mind. Here, too, the circuit realization is also given (Fig. 16.).

These full examples are followed by several small illustrations of different Verilog constructs. Lines of dots serve as delimiters between them.

`// The most simple multiplier (behavioural description)`

```
// using a continuous assignment and zero delay

`timescale 100ns/1ns

module szorzo(szorzat, szorzando, szorzo);
output [15:0] szorzat;
input [7:0] szorzando;
input [7:0] szorzo;

wire [15:0] szorzat;

assign szorzat = szorzando * szorzo;

endmodule                // end of the multiplier module


module tb_szorzo;        // Testbench of the multiplier

wire [15:0] product;
```

```
reg [7:0] multiplicand;
reg [7:0] multiplier;

szorzo mult(product, multiplicand, multiplier);

initial
  begin
    multiplicand = 8'h3;
    multiplier = 8'h0;
    #10 multiplier = 8'h4;
    #10 multiplier = 8'h6;
    #10 multiplier = 8'h8;
    #10 $finish;
  end

endmodule

//....................................................
```

// Multiplier circuit (detailed RTL description)

```
// Described on RTL-level with accumulator register
// and conditional expression which is meant for
// directing synthesization to creating optimal structure.
// The first module describes the data path, controlled
// by an instance of the controller which is described
// in a separate module

`timescale 1ns/1ns           // time unit = 1 nanosec

module szorzo(szorzat, szorzando, szorzo, start, clock);
output [15:0] szorzat;       // product
input [7:0] szorzando;       // multiplicand
input [7:0] szorzo;          // multiplier
input start, clock;

reg [15:0] szorzat;
wire [3:0]q;
wire clear;
reg [15:0] akku;             // accumulator

ctrl ct(q, clear, start, clock);  // controller instantiated

always
  begin : mul                // The process is given the name mul
   @(posedge clock)
      if (clear == 1'b1) akku = 16'h0000;
      if ((q<8) && (q>=0))
        akku = szorzo[q]
                   ? {akku[14:0], 1'b0} + {8'h00, szorzando[7:0]}
                   : {akku[14:0], 1'b0};
      if (q == 4'h0) szorzat = akku;
  end

endmodule
```

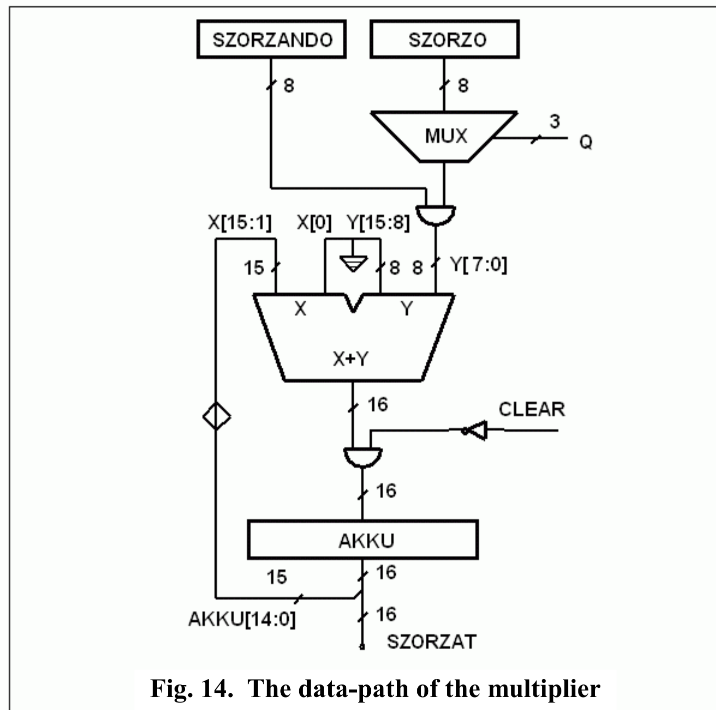

Fig. 14. The data-path of the multiplier

```
// Testbench of the multiplier

module tb_szorzo;

parameter cp = 100;          // clock period = 100 nsec

wire [15:0] zat;             // product
reg [7:0] zando;             // multiplicand
reg [7:0] zo;                // multiplier
reg start, clock;

szorzo mult(zat, zando, zo, start, clock);  // multiplier instance

initial                      // stimuli
  begin
    clock = 0;
    start = 0;
    zando = 8'h3;
    zo    = 8'h0;
    #2000 zo    = 8'h4;
          start = 1;
    #cp   start = 0;
    #1900 zo    = 8'h6;
          start = 1;
    #cp   start = 0;
    #1900 zo    = 8'h8;
          start = 1;
    #cp   start = 0;
    #1900 $finish;
  end

always #(cp/2) clock = ~clock;     // clock generator

endmodule

//...................................................
// ctrl.v  --  controller for the multiplier
//             activated by a pulse at the input start

module ctrl(q, clear, start, clock);
```

```
output [3:0] q;
output clear;
input start, clock;

reg [3:0] q;
reg clear;

initial
  begin
    clear=0;
    q = 4'h9;
  end

always @(posedge clock)
  begin
      if (start==1 && q == 4'h9)
          begin
            clear = 1'b1;
            q = q - 4'h1;
          end
      else if (q == 4'h8)
          begin
            clear = 1'b0;
            q = q - 4'h1;
          end
      else if (q < 4'h8)
                  q = q - 4'h1;
      if (q == 4'hf) q = 4'h9;
  end

endmodule
```

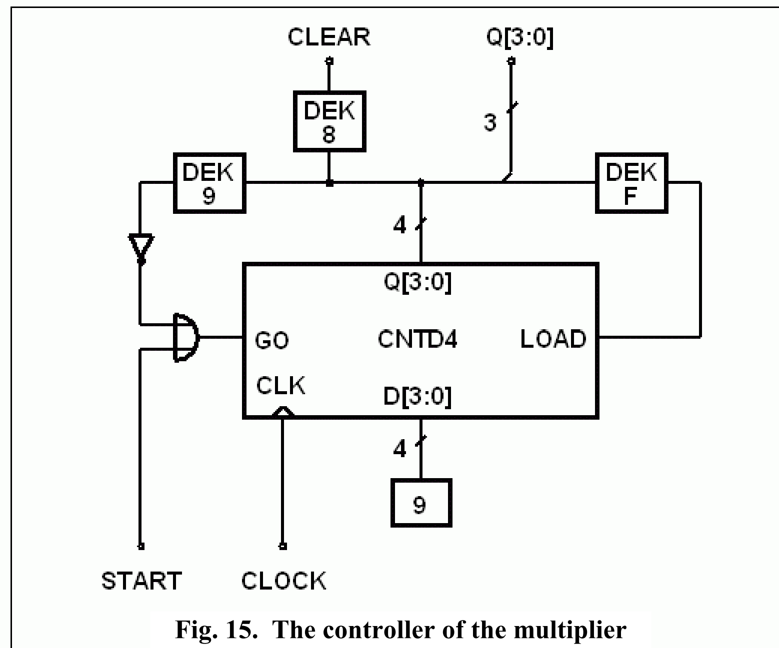

Fig. 15. The controller of the multiplier

```
// Sigma-Delta A/D-Converter
// ==========================

// Test-bench for Sigma-delta

`timescale 1ns/1ps

// Prescribing actual parameter values for parametrizable
// models (annotation)
```

```
module annotate;

parameter WW=5;    // A/D conversion with WW=5 bit resolution

  defparam tb_sigdel.sd.WW = WW;
  defparam tb_sigdel.sd.tim.WW = WW;
  defparam tb_sigdel.sd.cnt.WW = WW;
  defparam tb_sigdel.WW = WW;
  defparam tb_sigdel.CP = 100;      // Clock periode is 100 nsec

// Integrator time-constant:

  defparam tb_sigdel.sd.v2p.rci.RC = 2000;

endmodule          // end of the annotation

module tb_sigdel;  // begin of the test-bench

parameter WW=8, CP=1000;  // Parameters superseded by annotation

reg [15:0] uin;           // Quasi-analog input voltage
reg clock;
wire [(WW-1):0] q;        // Converted result
wire rdy;                 // Conversion done

sigmadelta sd(q, rdy, uin, clock);   // instance of the converter

always #(CP/2) clock = ~clock;       // clock generator

initial     // specifying (quasi-)analog input voltages
  begin
    clock = 0;
    uin = 0;
    #250    uin = 16'h7000;
    #5000   uin = 16'hffff;
    #15000  uin = 16'h9000;
    #5000   uin = 16'h0;
    #15000  $finish;
  end

endmodule








//.............................................................

// Sigma/Delta A/D-Converter

module sigmadelta(qq, rdy, uin, clock);
output qq;
output rdy;
input [15:0] uin;
input clock;

parameter WW=8;

wire [(WW-1):0] q;
wire rdy, ss;

// the output register qq captures and holds the converted
// value till the next conversion is done
```

```
reg [(WW-1):0] qq;
initial qq=0;
always @(posedge clock)
    if (rdy) qq = q;

volt2puls v2p(ss, uin, clock);   // voltage/pulse converter
counter cnt(q, ss, rdy, clock);  // ones´s counter
timer tim( , rdy, clock);        // time basis, output q is not used

endmodule

//............................................................

// Voltage-to-pulses converter, converts analog voltage to a
// stream of pulses

module volt2puls(q, uin, clock);
output q;
input uin;
input clock;

// Quasi-analog voltages:

wire [15:0] uin;          // input
wire [15:0] ufb;          // feed-back
wire [15:0] ubuf;         // buffer of comparator output

wire q;           // original comparator output, digital

// It just contains 3 sub-modules

komp cmp(q,uin,ufb,clock);     // Comparator
output_buffer bf(ubuf, q);     // Buffer generating analog output
rcint rci(ufb, ubuf, clock);   // Feed-back RC integrator

endmodule




//............................................................

// Clocked analog comparator with quasi-analog integer input

module komp(q,ux,uref, clock);
output q;
input [15:0] ux;
input [15:0] uref;
input clock;

reg q;

initial q=0;

always @(posedge clock);
     q = uref<ux;

endmodule

//............................................................

// Output buffer, producing quasi-analog output voltage.
// Input: logic 0 and 1; Output: 16-bit quasi-analog integer
//                                values 16'h0000 and 16'hffff

module output_buffer(qa,din);
```

```
input din;       // digital input
output qa;       // analog output

reg [15:0] qa;

always @din
     qa = din ? 16'hffff : 16'h0000 ;

endmodule

//..............................................................

// Integrating RC feedback circuit for sigma-delta A/D
// conversion, with quasi-analogous 16-bit integer variables.
// The voltage range 0...5V is mapped from 'h0000 to 'hffff
// Realizes the equation duout = (uin - uout)*dt/RC
// Signs are separated because regs store unsigned numbers

module rcint(uout, uin, clock);
output uout;
input [15:0] uin;
input clock;

parameter RC = 3000;    // integrator time constant
              // for correct operation RC >= 10*clock-period

reg [15:0] uout;
reg [15:0] delu1;
reg [15:0] delu2;

time delt, tprev;

always
  begin
    @(posedge clock)
        delt = $time - tprev;   // dt
        tprev = $time;
        if (uin<uout)           // Increasing output
          begin
            delu1 = uout-uin;
            delu2 = delu1*delt/RC;
            uout = uout - delu2;
          end
        else                    // Decreasing output
          begin
            delu1 = uin-uout;
            delu2 = delu1*delt/RC;
            uout = uout + delu2;
          end
  end

initial
  begin
    uout = 0;
    tprev = 0;
  end

endmodule

//..............................................................

// WW bits wide counter with synchronous clear
// for counting the ones in the input stream

module counter(q,go,clear,clock);
output q;
```

```
input go, clear,clock;

parameter WW=8;

reg [(WW-1):0] q;

initial q=0;

always
  begin
    @(posedge clock)
      if (go) q = q + 1'b1;
      if (clear) q = {WW{1'b0}};
  end

endmodule
```

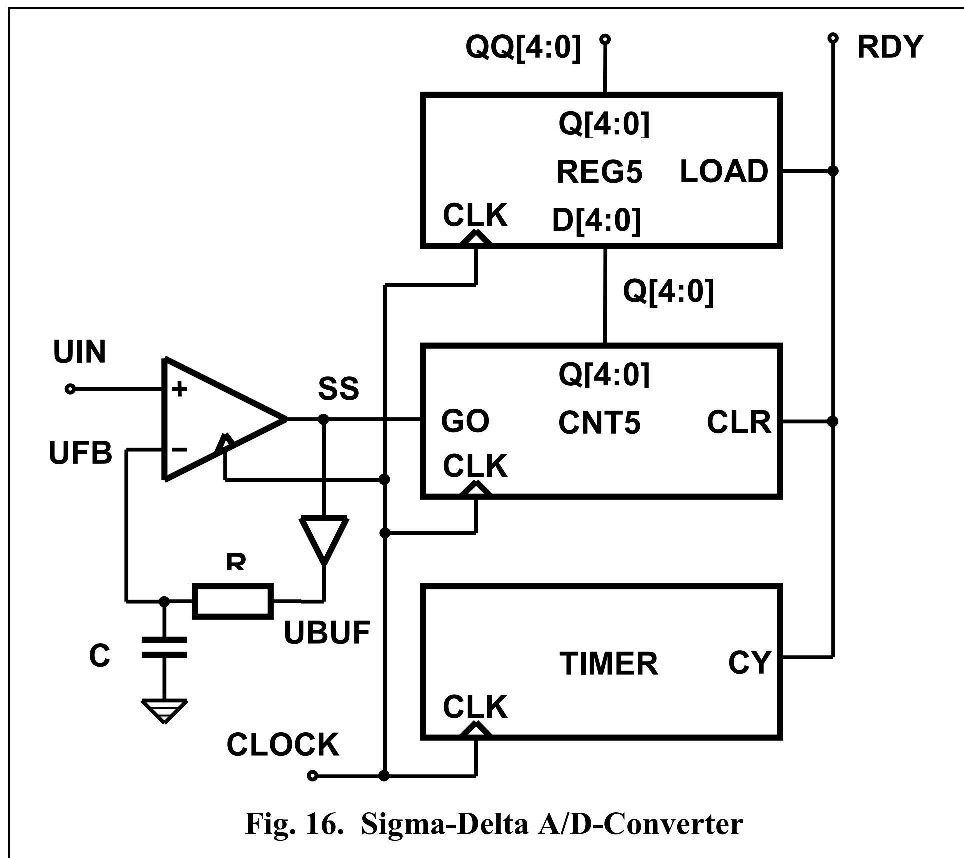

Fig. 16. Sigma-Delta A/D-Converter

```
// Timer: provides the time basis for the A/D conversion
// WW bits wide counter, carry=1 when the value is all-ones

module timer(q,cy,clock);
output q, cy;
input clock;

parameter WW=8;

reg [(WW-1):0] q;
reg cy;

initial
  begin
    q=0; cy=0;
  end
```

```
always
  begin
    @(posedge clock)
      q = q + 1'b1;
      if (q == {WW{1'b1}})
        begin
          cy = 1'b1;
          @(posedge clock)
            q = {WW{1'b0}};
            cy = 1'b0;
        end
  end

endmodule
//.............................................................
```

// Small examples of different Verilog constructs

```
// 1. Continuous assignment and concatenation,
//    applied in a four-bit adder with carry

wire carry_out, carry_in;
wire [3:0] sum_out, ina, inb;
assign
  {carry_out, sum_out} = ina + inb + carry_in;

//........................................................
// 2. 4x16 --> 1x16 data-multiplexor module using the
//                  conditional operator

module select_bus(busout, bus0, bus1, bus2, bus3, enable, s);
  parameter n = 16;
  parameter Z = 16´bz;
  output [1:n] busout;
  input [1:n] bus0, bus1, bus2, bus3;
  input enable;
  input [1:2] s;
  tri [1:n] data;
  tri [1:n] busout = enable ? data : Z;
  assign
    data = (s==0) ? bus0 : Z,
    data = (s==1) ? bus1 : Z,
    data = (s==2) ? bus2 : Z,
    data = (s==3) ? bus3 : Z;
endmodule

//........................................................
// 3. IF - ELSE statement

if (index > 0)
  begin
    if (rega > regb)
      result = rega;
  end                       // because of begin/end pair
else                        // else belongs to the first if
  result = regb;

//........................................................
// 4. CASE statement used in a four-bit decimal decoder

reg [3:0] rega;
reg [0:9] result;

case (rega)
```

```
   4´d0: result = 10´b0111111111;
   4´d1: result = 10´b1011111111;
   4´d2: result = 10´b1101111111;
   4´d3: result = 10´b1110111111;
   4´d4: result = 10´b1111011111;
   4´d5: result = 10´b1111101111;
   4´d6: result = 10´b1111110111;
   4´d7: result = 10´b1111111011;
   4´d8: result = 10´b1111111101;
   4´d9: result = 10´b1111111110;
   default result = ´bx;
endcase

//.....................................................
// 5. REPEAT  --  applied in a simple multiplier

parameter size = 8, longsize = 16;
reg [size:1] opa, opb;      // multiplicand, multiplier
reg [longsize:1] result;
begin : mult
  reg [longsize:1] shift_opa, shift_opb;
  shift_opa = opa;
  shift_opb = opb;
  result = 0;

repeat (size)
  begin
    if (shift_opb[1]) result = result + shift_opa;
    shift_opa = shift_opa << 1;
    shift_opb = shift_opb >> 1;
  end
end

//.....................................................
// 6. WHILE  --  this block counts the logical ones
//               contained in the bits of rega

begin : count1s        // a name is given to the block
  reg [7:0] tempreg;
  count = 0;
  tempreg = rega;
  while (tempreg)
    begin
      if (tempreg[0]) count = count +1;
      tempreg = tempreg >> 1;
    end
end

//.....................................................
// 7. FOR  --  the multiplier of the example REPEAT,
//             using the construct ´for´

parameter size = 8, longsize = 16;
reg [size:1] opa, opb;
reg [longsize:1] result;
begin : mult
  integer bindex;
  result = 0;
  for (bindex = 1; bindex <= size; bindex = bindex + 1)
    if (opb[bindex])
      result = result + (opa << (bindex - 1));
end
```

```
//.........................................................
// 8. Delay control

parameter d = 25, e = 120;
reg [0:7] rega, regb, regc;

#d rega = regb;
#((d+e)/2) rega = regb;
#regc rega = regb;

//.........................................................
// 9. Event control

@r rega = regb;              // activated by any changes
                             // of value in the register r

@(posedge clk) rega = regb;  // controlled by the rising and
@(negedge clk) rega = regb;  // falling edge of the clock

//.........................................................
// 10. Named event

event end_wave;    // Event declaration

// activation in a block

parameter d 50;
reg [7:0] r;
begin              // waveform controlled by
  #d r = 'h35;     // sequential delay
  #d r = 'hE2;
  #d r = 'h00;
  #d r = 'hF7;
  #d -> end_wave;  // generate the event
end

// using the event somewhere else in the program

@end_wave rega = regb;

//.........................................................
// 11. Level-sensitive event control (wait)

begin
  wait (enable) #10 a = b;
  #10 c=d;
end

// When the control comes to the block:
//   - if enable=1 then after 10nsec a=b
//   - if enable=0 then waits for enable=1
//     and then after 10nsec a=b

//.........................................................
// 12. TASK  --  Example: programmable monoflop

task mflop;
  output pulse;
  input clock;
  input [31:0] ticks;    // pulse duration in clock ticks
  pulse = 'b1;           // pulse starts
```

```
  begin
    repeat (tics)
      @(posedge clock);  // wait for rising edge of clock
    pulse = 'b0;         // pulse finished
  end
endtask

// Activation somewhere else in the program

reg open;
reg [31:0] length;

mflop(open, length);

//.......................................................
// 13. FUNCTION  --  Computing the factorial

function [31:0] factorial;
  input [3:0] n;
  reg [3:0] index;
  begin
    factorial = n ? 1 : 0;
    for(index = 2; index <= n; index = index + 1)
      factorial = factorial + 1;
  end
endfunction

// Activation somewhere else in the program

result = apha * factorial(dd);
```