Introduction to Verilog HDL

Ben Abdallah Abderazek
National University of Electro-communications, Tokyo,
Graduate School of information Systems
May 2004

What you will understand after having this lecture ?

- After having this lecture you will be able to:
  - Understand Design Steps with Verilog-HDL
  - Understand main programming technique with Verilog HDL
  - Understand tools for writing and simulating a given design (module(s)).

Choice of Hardware Description Languages

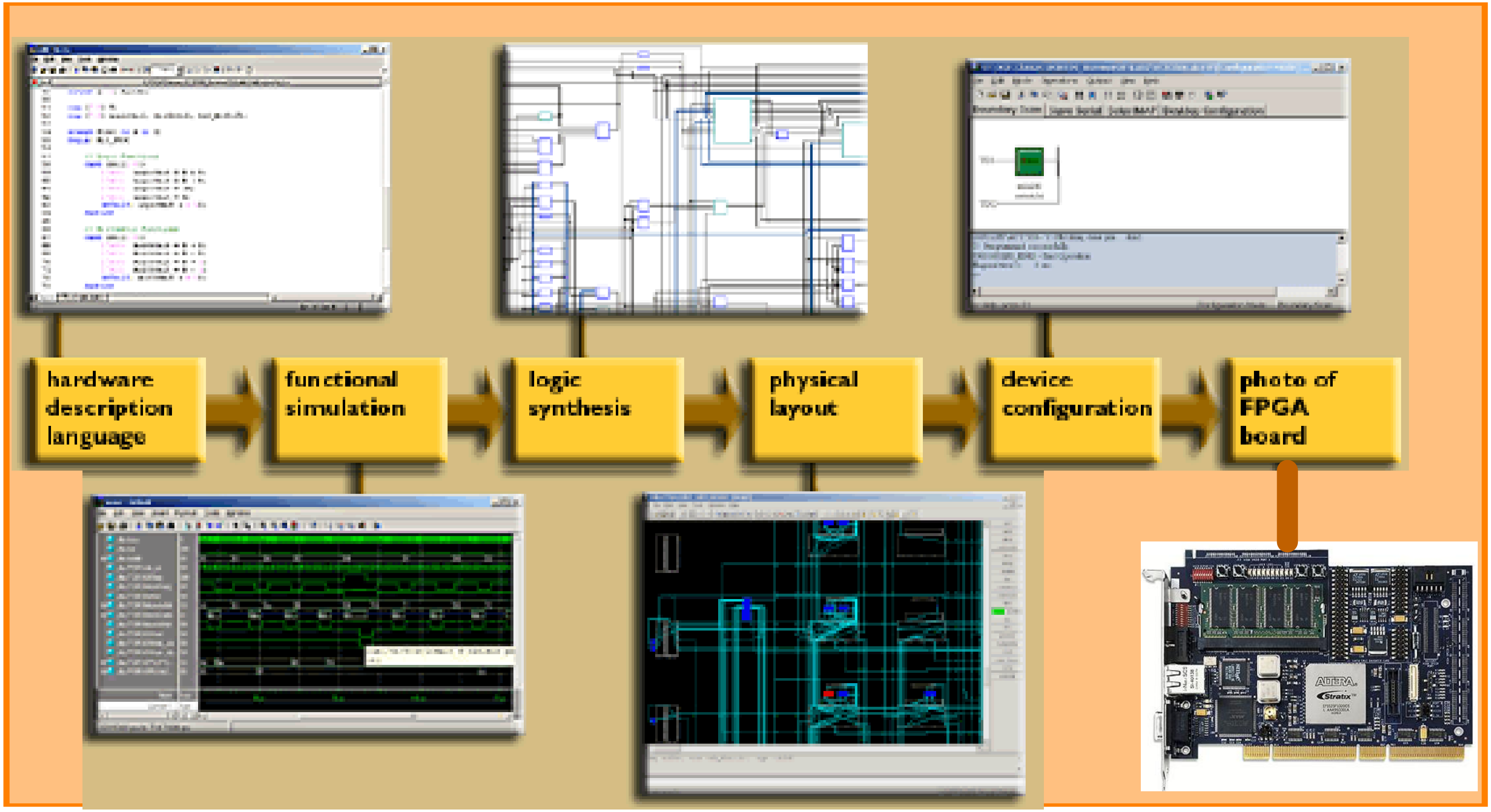
- There are a fair number of HDLs, but two are by far most prevalent in use:
- Verilog-HDL, the Verilog Hardware Description Language, not to be confused with Verilog-XL, a logic simulator program sold by Cadence.
- VHDL, or VHSIC Hardware Description Language and VHSIC is Very High Speed Integrated Circuit.
- Reality: Probably need to know both languages
  - Impossible to say which is better – matter of taste!!

In this lecture, I will be using only Verilog-HDL.

Why Verilog?

- **Why use an HDL？**
  - Describe complex designs (millions of gates)
  - Input to synthesis tools (synthesizable subset)
  - Design exploration with simulation
- **Why not use a general purpose language ？**
  - Support for structure and instantiation (objects?)
  - Support for describing bit-level behavior
  - Support for timing
- **Verilog vs. VHDL**
  - Verilog is relatively simple and close to C
  - VHDL is complex and close to Ada
  - Verilog has 60% of the world digital design market Verilog modeling range From gates to processor level

Design Process in Verilog-HDL

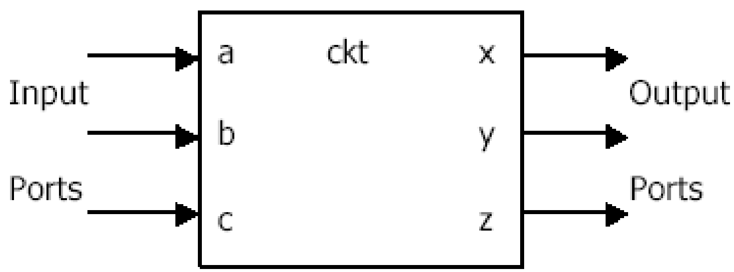


Design Process in Verilog-HDL

- Understand problem and generate block diagram of solution
- Code block diagram in verilog
- Synthesize verilog
- Create verification script to test design
- Run static timing tool to make sure timing is met
- Design is mapped, placed, routed, and *.bit file is created and download to FPGA

Modeling Structure: Modules

- The module is the basic building block in Verilog
- Modules can be interconnected to describe the structure of your digital system
- Modules start with keyword **module** and end with keyword **endmodule**

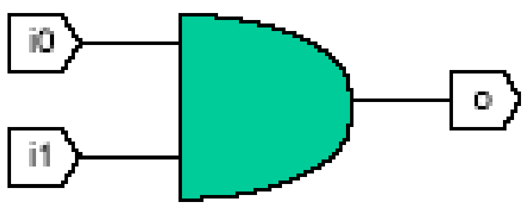


Port List

```
module ckt(x,y,z,a,b,c);
// Output port definitions
output x;
output y;
output z;
// Input port definitions
input a;
input b;
input c;
        .
   module body
        .
endmodule
```

Modeling Structure: Ports

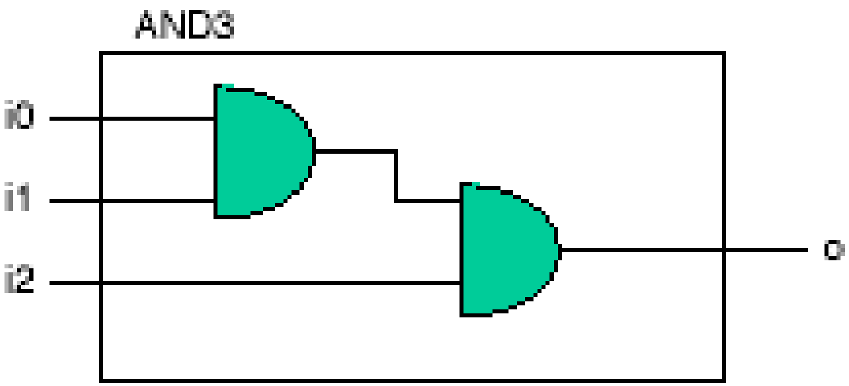
- Module Ports
  - Similar to pins on a chip
  - Provide a way to communicate with outside world
  - Ports can be input, output or inout




```
Module AND (i0, i1, o);
    input  i0, i1;
    output O;

          .
          .
          .

endmodule
```

Modeling Structure: instances

- **Module instances**
  - Verilog models consist of a hierarchy of module *instances*
  - In C++ speak: modules are classes and instances are objects




```
Module AND3 (i0, i1, i2, o);
    input  i0, i1, i2  ;
    output O;

    wire temp

    AND a0 (i0, i1, temp);
    AND a1 (i2, temp, O);
endmodule
```

Data Values

- For our logic design purposes, we'll consider Verilog to have four different bit values:

- ✓ 0, logic zero.
- ✓ 1, logic one.
- ✓ z, high impedance.
- ✓ x, unknown.

Data Values

- When specifying constants, whether they be single bit or multi-bit, you should use an explicit syntax to avoid confusion:

- 4’d14 // 4-bit value, specified in decimal
- 4’he // 4-bit value, specified in hex
- 4’b1110 // 4-bit value, specified in binary
- 4’b10xz // 4-bit value, with x and z, in binary

- The general syntax is:

– {bit width}’{base}{value}

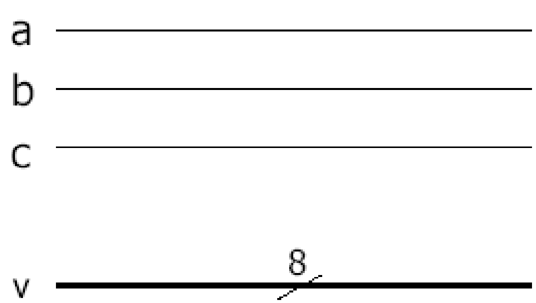
Data Type

- There are two main data types in Verilog. These data types may be single bit or multi-bit.
  - Wires
    - Wires are physical connections between devices and are "continuously assigned".
    - Nets do not "remember", or store, information -This behaves much like an electrical wire...
  - Registers
    - Regs are "procedurally assigned" values and "remember", or store, information until the next value assignment is made.
    - Register type is denoted by reg

Data Type Declaration

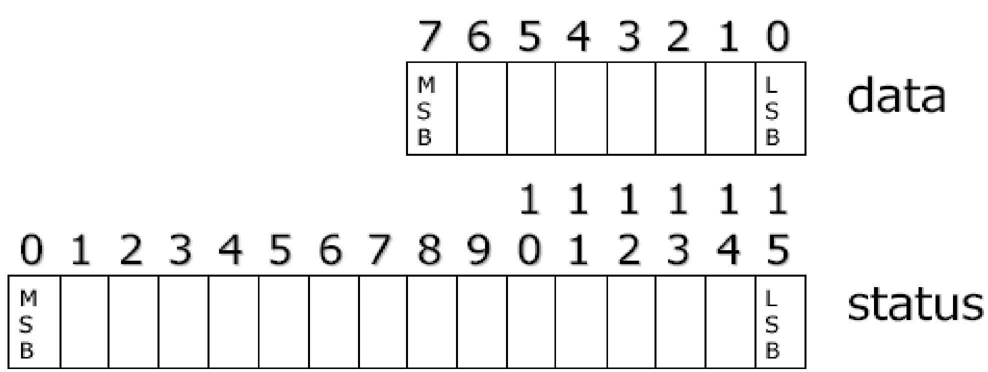
➢Wire (wire) Definition

```
wire a, b, c;      // Define 1-bit nets a, b, and c.

wire [7:0]v;       // Define 8-bit wire vector
```



➢Register (reg) Definition

```
reg [7:0] data;         // 8-bits wide, LSB0

reg [0:15] status;      // 16-bits wide, MSB0
```



Variable Declaration

- **constants**

Un-Sized (32-bit)

127	$= 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111\ 1111_2$

Sized (As specified)

4'b1010	$= 1010_2$
8'd255	$= 1111\ 1111_2$
16'hbeef	$= 1011\ 1110\ 1110\ 1111_2$

Example Module

```
// qcc:: queue computation circuit
module qcc (
          // output ports
          lqh_out,
          qh_out,
          qt_out,
          qh_plus_operand,
          flags_out,
          // input ports from previous
          // qcc or initial pointers values
          lqh_in,
          qh_in,
          qt_in,
          //from the decoder unit
          delta_lqh,    //
          delta_qh,     // number of operands consumed
          delta_qt,     // number of produced results ( 0 or 1)
          operand,      //
          flags_in      //
          );
     //
     output  [7:0]  lqh_out;
     output  [7:0]  qh_out;
     output  [7:0]  qt_out;
     output  [7:0]  qh_plus_operand;
     output  [7:0]  flags_out;
     //
     input  [7:0]   lqh_in;       // declare live queue head value
     input  [7:0]   qh_in;        // declare queue head value
     input  [7:0]   qt_in;        // declare queue head value
     input  [7:0]   delta_lqh;    //  declare delata_lqh
     input  [7:0]   delta_qh;     // consumed data number
     input  [7:0]   delta_qt;     // produced data number
     input  [7:0]   operand;      // declare operand
     input  [7:0]   flags_in;     //
```

Verilog Operator

Arithmetic

+	(addition)
-	(subtraction)
*	(multiplication)
/	(division)
%	(modulus)

Example:

```
parameter n = 4;
reg[3:0] a, c, f, g, count;
f = a + c;
g = c - n;
 count = (count +1)%16;          //Can count 0 thru 15.
```

Relational

<	(less than)
<=	(less than or equal to)
>	(greater than)
>=	(greater than or equal to)
==	(equal to)
!=	(not equal to)

Example:

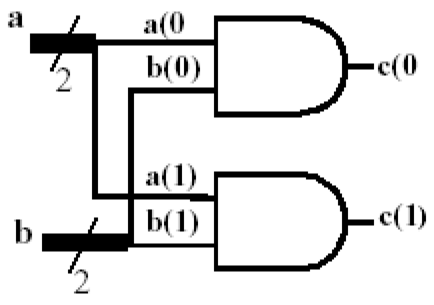
```
if (x = = y)    e = 1;
else            e = 0;

// Compare in 2's compliment;  a>b
  reg [3:0] a,b;
  if (a[3]= = b[3])   a[2:0] > b[2:0];
  else                b[3];
```

Bitwise

~	(bitwise NOT)
&	(bitwise AND)
|	(bitwise OR)
^	(bitwise XOR)
~^ or ^~	(bitwise XNOR)

Example:

```
module and2 (a, b, c);
  input [1:0] a, b;
  output [1:0] c;
  assign c = a & b;
endmodule
```



Verilog Operator

Logical

```
!    (logical NOT)
&&   (logical AND)
||   (logical OR)
```

Example:

```
wire[7:0] x, y, z;          // x, y and z are multibit variables.
reg a;
. . .
if ((x == y) && (z)) a = 1; // a = 1 if x equals y, and z is nonzero.
  else  a = !x;             // a =0 if x is anything but zero.
```

Shift

```
<<  (shift left)
>>  (shift right)
```

Example:

```
assign c = a << 2;    /* c = a shifted left 2 bits;
                         vacant positions are filled with 0's */
```

Concatenation

```
{ }(concatenation)
```

Example:

```
wire [1:0] a, b;   wire [2:0] x;     wire [3;0] y, Z;
assign x = {1'b0, a};  // x[2]=0,  x[1]=a[1],  x[0]=a[0]
assign y = {a, b};  /* y[3]=a[1],  y[2]=a[0],  y[1]=b[1],
y[0]=b[0] */

assign {cout, y} = x + Z;  // Concatenation of a result
```

Lexical Conventions

- Close to the programming language C++.
- Comments are designated by // to the end of a line or by /* to */ across several lines.
- Keywords, e. g., module, are reserved and in all lower case letters.
- case sensitive, meaning upper and lower case letters are different.

Port and Data Types

- An input port can be driven from outside the module by a wire or a reg, but inside the module it can only drive a wire (implicit wire).

- An output port can be driven from inside the module by a wire or a reg, but outside the module it can only drive a wire (implicit wire).

- An inout port, on both sides of a module, may be driven by a wire, and drive a wire.

Data type declaration syntax and examples

```
// qcc:: queue computation circuit
module qcc (
        // output ports
        lqh_out,
        qh_out,
        qt_out,
        qh_plus_operand,
        flags_out,
        // input ports from previous
        // qcc or initial pointers values
        lqh_in,
        qh_in,
        qt_in,
        //from the decoder unit
        delta_lqh,   //
        delta_qh,    // number of operands consumed
        delta_qt,    // number of produced results ( 0 or 1)
        operand,     //
        flags_in     //
        );
    //
    output  [7:0]  lqh_out;
    output  [7:0]  qh_out;
    output  [7:0]  qt_out;
    output  [7:0]  qh_plus_operand;
    output  [7:0]  flags_out;
    //
    input  [7:0]  lqh_in;        // declare live queue head value
    input  [7:0]  qh_in;         // declare queue head value
    input  [7:0]  qt_in;         // declare queue head value
    input  [7:0]  delta_lqh;     //  declare delata_lqh
    input  [7:0]  delta_qh;      // consumed data number
    input  [7:0]  delta_qt;      // produced data number
    input  [7:0]  operand;       // declare operand
    input  [7:0]  flags_in;      //
```

treat these as a wire, or you can add an explicit "reg portname;" declaration and then treat it as a reg data type

Treat these as if they were wires here

Continuous Assignment

- Continuous assignments are made with the assign statement:

assign LHS = RHS;

<u>Rules:</u>

- The left hand side, LHS, must be a wire.
- The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.

Continuous Assignment

- **Example 1**

```
module two_input_xor (in1, in2, out);
input in1, in2;                      // use these as a wire
output out;                          // use this as a wire
assign out = in1 ^ in2;
endmodule
```

- **Example 2**

```
module two_input_xor (in1, in2, out);
input in1, in2;
output out;
wire product1, product2;
assign product1 = in1 & !in2;        // could have done all in
assign product2 = !in1 & in2;        // assignment of out with
assign out = product1 | product2;    // bigger expression
endmodule
```

Procedural Constructs

- Two Procedural Constructs
  - initial Statement
  - always Statement
- initial Statement : Executes only once
- always Statement : Executes in a loop

Syntax examples:

```
initial
begin
     // These procedural assignments are executed
     // one time at the beginning of the simulation.
end

always @(sensitivity list)
begin
     // These procedural assignments are executed
     // whenever the events in the sensitivity list
     // occur.
end
```

Sensitivity list:

- always @(a or b)   // any changes in a or b
- always @(posedge a) // a transitions from 0 to 1
- always @(negedge a) // a transitions from 1 to 0
- always @(a or b or negedge c or posedge d)

Assignment rules:

- The left hand side, LHS, must be a reg.
- The right hand side, RHS, may be a wire, a reg, a constant, or expressions with operators using one or more wires, regs, and constants.

Procedural Constructs

- Combinational logic using operators:

```
module two_input_xor (in1, in2, out);
input  in1, in2;                        // use these as wires
output out;                             // use this as a wire
reg    out;

always @(in1 or in2)          // Note that all input terms
begin                         // are in sensitivity list!
     out = in1 ^ in2;         // Or equivalent expression...
end

// I could have simply used:
// always @(in1 or in2) out = in1 ^ in2;

endmodule
```

Procedural Constructs

- Combinational logic using if-else:

```
module two_input_xor (in1, in2, out);
input  in1, in2;                       // use these as wires
output out;                            // use this as a wire
reg    out;

always @(in1 or in2)         // Note that all input terms
begin                        // are in sensitivity list!
     if (in1 == in2) out = 1'b0;
     else out = 1'b1;
end

endmodule
```

Procedural Constructs

- Combinational logic using case:

```
module two_input_xor (in1, in2, out);
input  in1, in2;                     // use these as wires
output out;                          // use this as a wire
reg    out;

always @(in1 or in2)        // Note that all input terms
begin                       // are in sensitivity list!
     case ({in2, in1})      // Concatenated 2-bit selector
     2'b01:  out = 1'b1;
     2'b10:  out = 1'b1;
     default: out = 1'b0;
     endcase
end
endmodule
```

Delay Control

You can add control the timing of assignments in procedural blocks in several ways:

- Simple delays.
  - #10;
  - #10 a = b;
- Edge triggered timing control.
  - @(a or b);
  - @(a or b) c = d;
  - @(posedge clk);
  - @(negedge clk) a = b;

Delay Control (cont.)

- Delay can be introduced
  - Example: assign #2 sum = a ^ b;
  - "#2" indicates 2 time-units
  - No delay specified : 0 (default)
- Associate time-unit with physical time
  - `timescale time-unit/time-precision
  - Example: `timescale 1ns/100 ps
- Timescale

  `timescale 1ns/100ps
  - 1 Time unit = 1 ns
  - Time precision is 100ps (0.1 ns)
  - 10.512ns is interpreted as 10.5ns

Delay Control (cont.)

- Example:

```
`timescale 1ns/100ps
module HalfAdder (A, B, Sum, Carry);
  input A, B;
  output Sum, Carry;
  assign #3 Sum = A ^ B;
  assign #6 Carry = A & B;
endmodule
```

System Tasks

- The $ sign denotes Verilog system tasks, there are a large number of these, most useful being:
  - $display("The value of a is %b", a);
    - Used in procedural blocks for text output.
    - The %b is the value format (binary, in this case…)
  - $finish;
    - Used to finish the simulation.
    - Use when your stimulus and response testing is done.
  - $stop;
    - Similar to $finish, but doesn't exit simulation.

Event Control

- Event Control
  - Edge Triggered Event Control
  - Level Triggered Event Control
- Edge triggered Event Control

```
@ (posedge CLK) //Positive Edge of CLK
   Curr_State = Next_state;
```

@ negedge	@ posedge
$1 \rightarrow x$	$0 \rightarrow x$
$1 \rightarrow z$	$0 \rightarrow z$
$1 \rightarrow 0$	$0 \rightarrow 1$
$x \rightarrow 0$	$x \rightarrow 1$
$z \rightarrow 0$	$z \rightarrow 1$

- Level Triggered Event Control

```
@ (A or B) //change in values of A or B
   Out = A & B;
```

Loop Statement

- Loop Statement
  - Repeat
  - While
  - For
- Repeat Loop
  - Example

```
repeat (count)
    sum = sum + 6;
```

  - If condition is a x or z is treated as 0

Loop Statement (cont.)

While Loop

- Example:

```
while (Count < 10) begin
  sum = sum + 5;
  Count = Count +1;
end
```

- If condition is a x or z it is treated as 0

For Loop

- Example:

```
for (Count = 0; Count < 10; Count = Count + 1) begin
  sum = sum + 5;
end
```

Conditional statement

- if Statement
- Format:

if (condition)

procedural_statement

else if (condition)

procedural_statement

- Example

```
if (Clk)
    Q = 0;
else
    Q = D;
```

Conditional Statement (cont.)

- Case Statement

Example 1:

```
case (X)
  2'b00: Y = A + B;
  2'b01: Y = A – B;
  2'b10: Y = A / B;
endcase
```

Example 2:

```
case (3'b101 << 2)
  3'b100: A = B + C;
  4'b0100: A = B – C;
  5'b10100: A = B / C; //This statement is executed
endcase
```

Memories

- An array of registers

```
reg [ msb : lsb ] memory1 [ upper : lower ];
```

<u>Example</u>

```
reg [3:0] mem [0:63]   // an array of  64 4-bit registers

reg mem [4:0]; // an array of  5 1-bit register
```

Compiler Directives

‘include – used to include another file

- Example

‘include “./pqp_fetch.v”

`define – (Similar to #define in C) used to define global parameter

Example:

```
`define BUS_WIDTH 16
reg [ `BUS_WIDTH - 1 : 0 ] System_Bus;
```

`undef – Removes the previously defined directive

Example:

```
`define BUS_WIDTH 16
  …
  reg [ `BUS_WIDTH - 1 : 0 ] System_Bus;
  …
`undef BUS_WIDTH
```

Suggested Coding Style

- Write one module per file, and name the file the same as the module. Break larger designs into modules on meaningful boundaries.
- Always use formal port mapping of sub-modules.
- Use parameters for commonly used constants.
- Be careful to create correct sensitivity lists.

Suggested Coding Style

- Don't ever just sit down and "code". Think about what hardware you want to build, how to describe it, and how you should test it.
- You are not writing a computer program, you are describing hardware… ***Verilog is not C!***
- Only you know what is in your head. If you need help from others, you need to be able to explain your design -- either verbally, or by detailed comments in your code.

PART II

Tools you need

&

Design Example

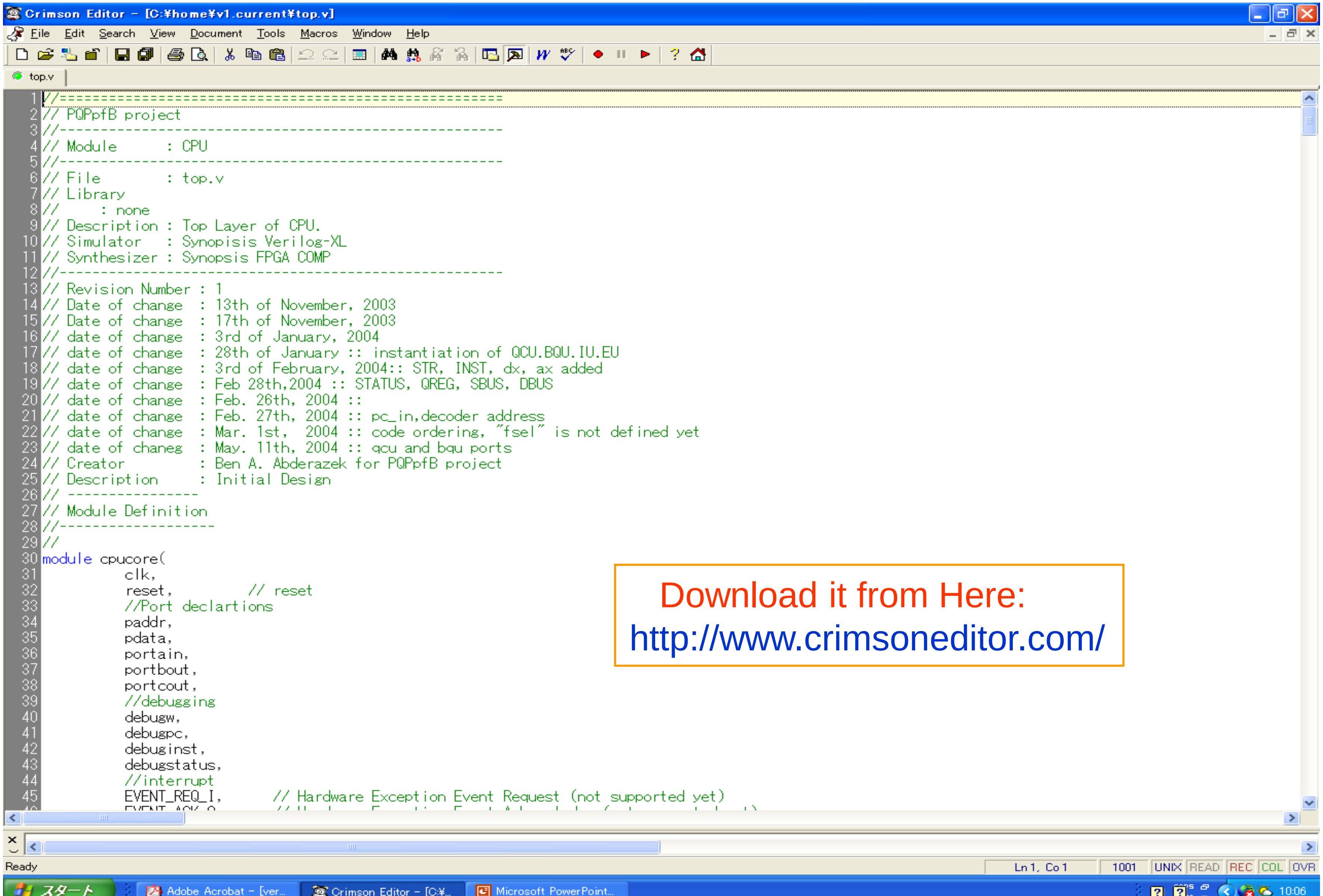
Tools

- You need two things

1. **Editor**
   - Crimson Editor 3.51 Release (Freeware) (for Windows)
   - Emacx (For UNIX)
2. **Simulators**
   - **Verilog-XL**: This is the most standard simulator in the market, as this is the sign off simulator.
   - **NCVerilog**This simulator is good when it comes to gate level simulations.
   - **Fc2** FPGA compiler for synthesis (net list generation)
   - **Simvision** for wave form viewing

What Editor you may use for your Verilog Code ?

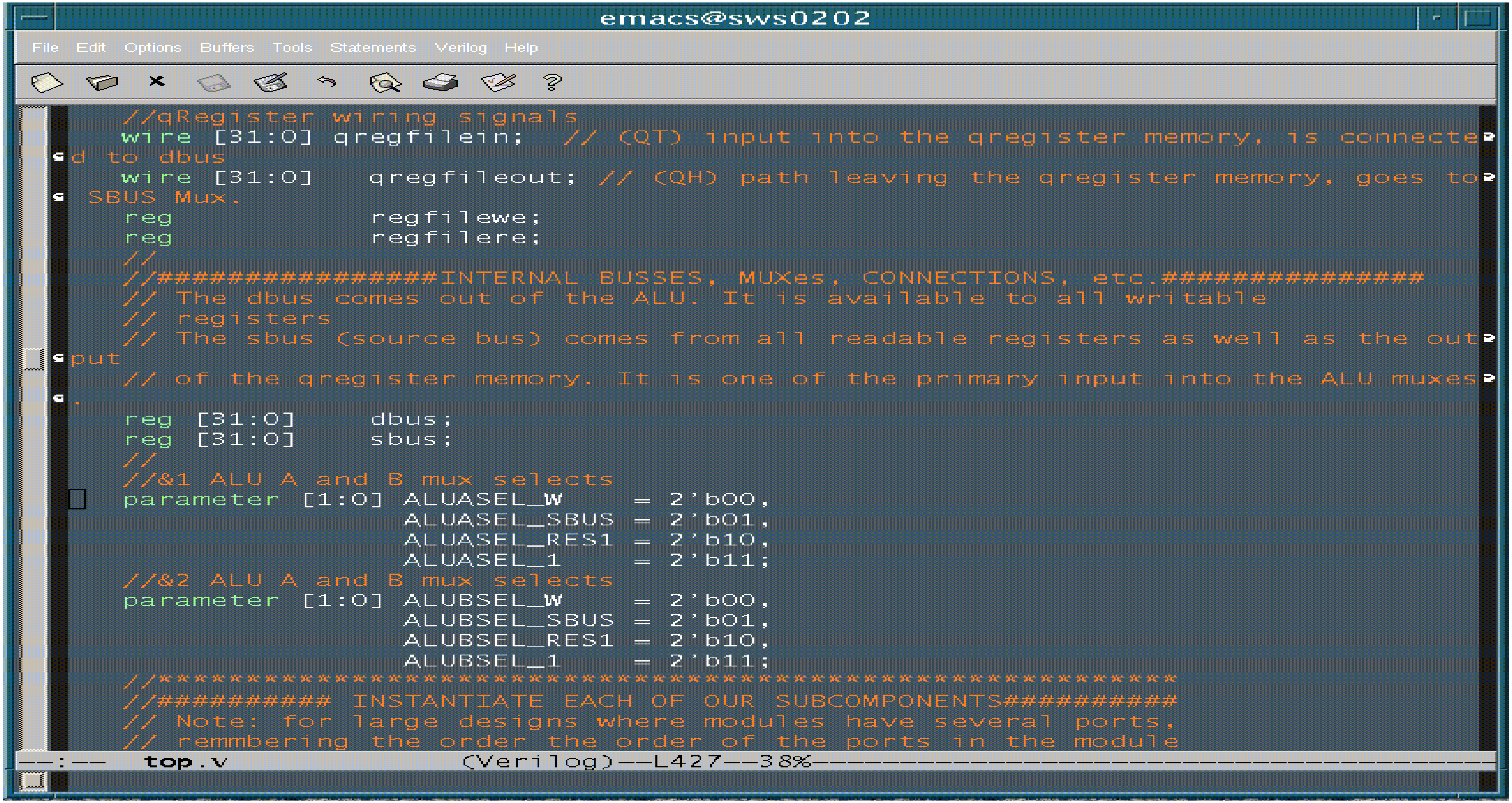
Crimson Editor (for windows OS)



Download it from Here:
http://www.crimsoneditor.com/

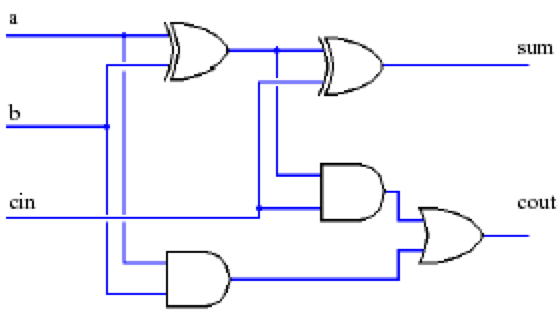
What Editor you can use for your Verilog Code ?

Emacs (for UNIX OS)



From your UNIX WS
at the commend prompt type:
mule top.v &

Example of one bit Full Adder



Behavior model

```
/****************************************
              One Bit Full Adder
****************************************/

module fadder(a,b,cin,sum,cout);

/*----------------------------------
  IO Signal Declaration
  ----------------------------------*/
  input   a;
  input   b;
  input   cin;
  output  sum;
  output  cout;

/*----------------------------------
  IO Signal type(reg/net) Specification
  ----------------------------------*/
  wire a;
  wire b;
  wire cin;
  wire sum;
  wire cout;

/*----------------------------------
  Internal Signal declaration
  ----------------------------------*/
  wire tempSum;

/*----------------------------------
  Logic Design
  ----------------------------------*/
  // Data Flow Style
  assign tempSum = a ^ b;
  assign sum  = tempSum ^ cin;
  assign cout = (tempSum & cin) | (a & b);

endmodule
```

Test bench for fader to output signal variation on the screen

```
module testFadder();
//-----------------------------------
//  Port-connection signal declaration
//-----------------------------------
  reg  t_a;
  reg  t_b;
  reg  t_cin;
  wire t_sum;
  wire t_cout;
//-----------------------------------
  Design module instantiation
//-----------------------------------
  fadder u_fadder( .a    (t_a),
                   .b    (t_b),
                   .cin  (t_cin),
                   .sum  (t_sum),
                   .cout (t_cout)
                 );
//-----------------------------------
  Stimulus generation
//-----------------------------------
  initial begin
    //Initialize the input signals
    t_a    = 1'b0;t_b    = 1'b0;t_cin  = 1'b0;
    //Assign the pattern to the logic
    #1 t_a   = 1'b0; t_b   = 1'b0; t_cin = 1'b1;

    #1 t_a   = 1'b0; t_b   = 1'b1; t_cin = 1'b0;
    #1 t_a   = 1'b0; t_b   = 1'b1; t_cin = 1'b1;
    #1 t_a   = 1'b1; t_b   = 1'b0; t_cin = 1'b0;
    #1 t_a   = 1'b1; t_b   = 1'b0; t_cin = 1'b1;
    #1 t_a   = 1'b1; t_b   = 1'b1; t_cin = 1'b0;
    #1 t_a   = 1'b1; t_b   = 1'b1; t_cin = 1'b1;
    // Provide some mechanism to finish the simulation
    // If $stop is used, the simulation is stopped temporarily.
    // If $finish is used the simulator just exits.
   #2 $finish;
  end
/*-----------------------------
  Display/Monitor your signals
  -----------------------------*/
  initial begin
     $monitor($time,,,"a = %b, b = %b, cin = %b, cout = %b, sum = %b", t_a, t_b, t_cin, t_cout, t_sum);
  end
endmodule
```

Where to FIND and how to RUN the Verilog XL Simulator ?

To use the simulator you should:

(a) First add the following line to your .tcshrc file

```
### Synopsys configuration ###
setenv SYNOPSYS              /cad/synopsys03/design_compiler
setenv SNPSLMD_LICENSE_FILE    $SYNOPSYS/admin/license/key  #after
2000.05
setenv LD_LIBRARY_PATH
${LD_LIBRARY_PATH}:${CADENCE}/tools.sun4v/lib
set path = ( $path $CADENCE/tools.sun4v/bin ¥
             $CADENCE/tools/bin ¥
             $SYNOPSYS/sparc64/syn/bin ¥
             $SYNOPSYS/../fpga_compiler2/bin)
setenv MANPATH
${MANPATH}:$CADENCE/tools.sun4v/man/man1:$CADENCE/share/man/man1:/$CAD
ENCE/share/man/man5:$SYNOPSYS/doc/syn/man
limit coredumpsize 0
setenv CVSROOT $HOME/CVS_DB
```

(b) To use the simulator remote login to one of the machines:

1. nws0300
2. sws0202

To run the Verilog-XL simulator from your UNIX Workstation type:

verilog fadder.v testfadder.v

Terminal

Window Edit Options He

```
RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
Technical Data and Computer Software clause at DFARS 252.227-7013 or
subparagraphs (c)(1) and (2) of Commercial Computer Software -- Restricted
Rights at 48 CFR 52.227-19, as applicable.

                Cadence Design Systems, Inc.
                555 River Oaks Parkway
                San Jose, California  95134

For technical assistance please contact the Cadence Response Center at
1-877-CDS-4911 or send email to support@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkv@cadence.com

Compiling source file "fadder.v"
Compiling source file "testfadder.v"
Highest level modules:
testFadder

                   0  a = 0, b = 0, cin = 0, cout = 0, sum = 0
                   1  a = 0, b = 0, cin = 1, cout = 0, sum = 1
                   2  a = 0, b = 1, cin = 0, cout = 0, sum = 1
                   3  a = 0, b = 1, cin = 1, cout = 1, sum = 0
                   4  a = 1, b = 0, cin = 0, cout = 0, sum = 1
                   5  a = 1, b = 0, cin = 1, cout = 1, sum = 0
                   6  a = 1, b = 1, cin = 0, cout = 1, sum = 0
                   7  a = 1, b = 1, cin = 1, cout = 1, sum = 1
L68 "testfadder.v": $finish at simulation time 9
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.0 secs in simulation
End of Tool:    VERILOG-XL      04.10.001-p   May 18, 2004  14:20:17
nws0300:/home/ben/tm3/teach>
```

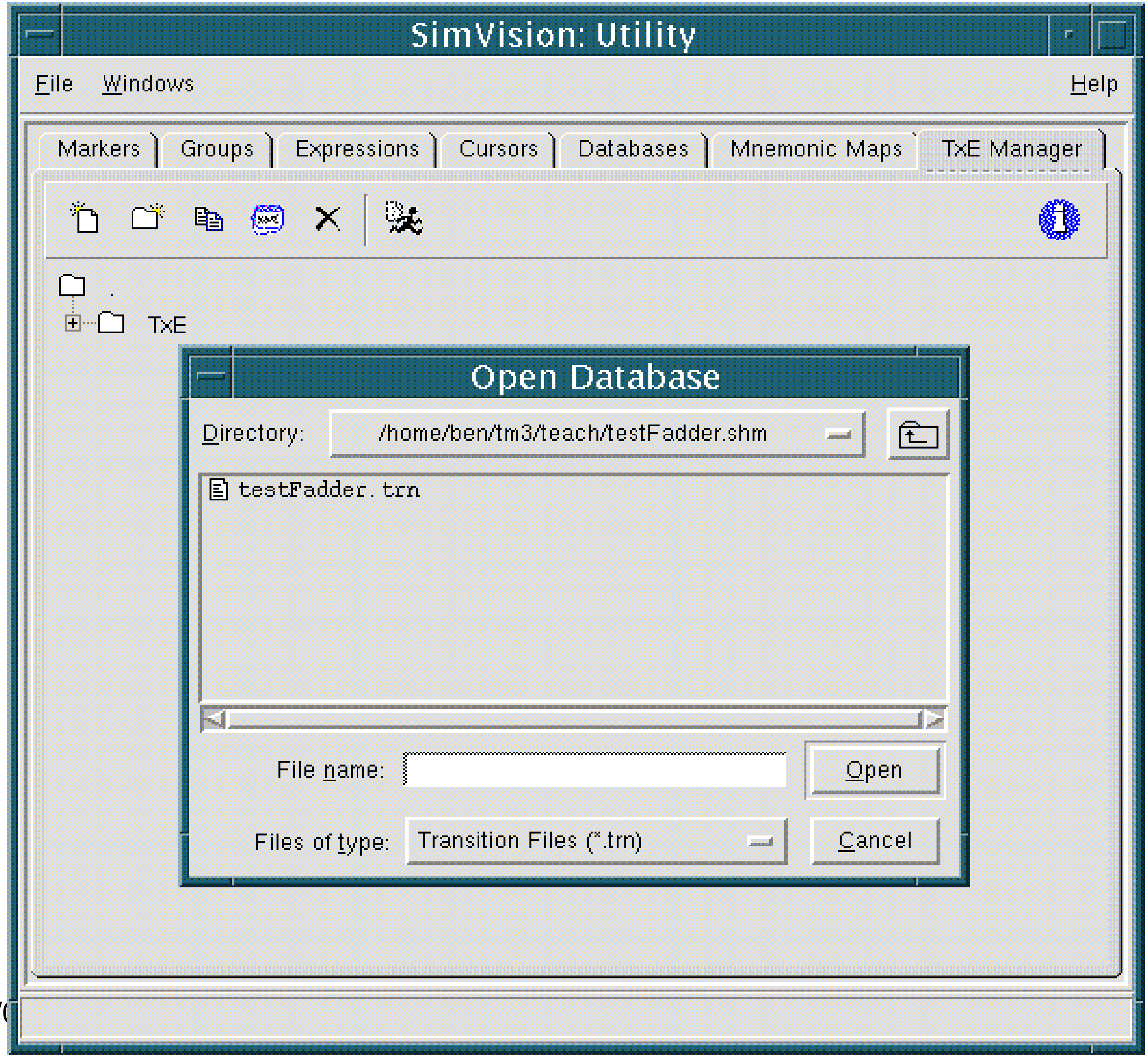
Test bench for fader for use with Simvision Wave viewer

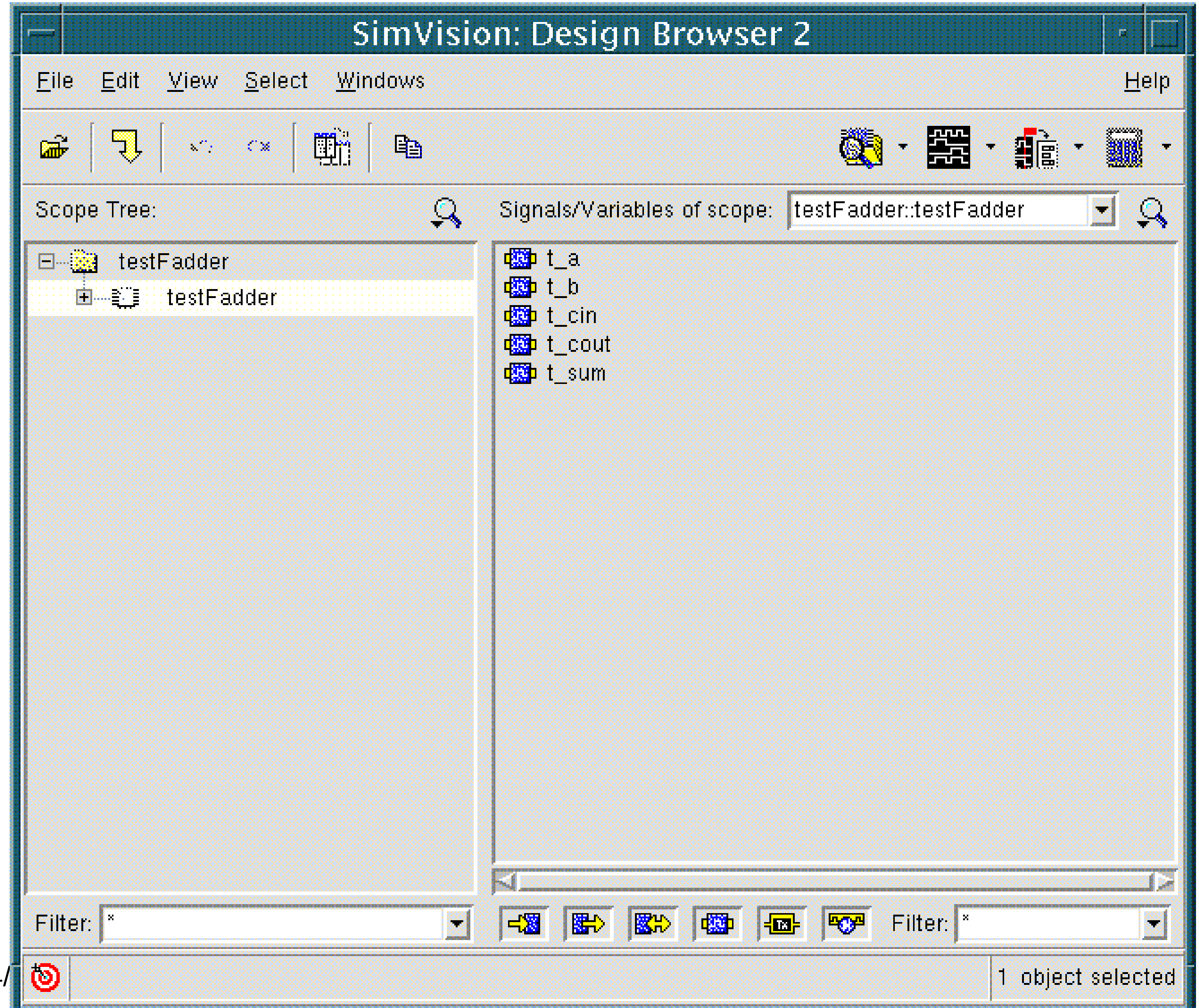
```
//-----------------------------------
  Design module instantiation
//-----------------------------------
  fadder u_fadder( .a    (t_a),
                   .b    (t_b),
                   .cin  (t_cin),
                   .sum  (t_sum),
                   .cout (t_cout)
                 );
//-----------------------------------
  Stimulus generation
//-----------------------------------
  initial begin
    //Initialize the input signals
    t_a    = 1'b0;t_b    = 1'b0;t_cin  = 1'b0;
    //Assign the pattern to the logic
    #1 t_a   = 1'b0; t_b   = 1'b0; t_cin = 1'b1;

    #1 t_a   = 1'b0; t_b   = 1'b1; t_cin = 1'b0;
    #1 t_a   = 1'b0; t_b   = 1'b1; t_cin = 1'b1;
    #1 t_a   = 1'b1; t_b   = 1'b0; t_cin = 1'b0;
    #1 t_a   = 1'b1; t_b   = 1'b0; t_cin = 1'b1;
    #1 t_a   = 1'b1; t_b   = 1'b1; t_cin = 1'b0;
    #1 t_a   = 1'b1; t_b   = 1'b1; t_cin = 1'b1;
    // Provide some mechanism to finish the simulation
    // If $stop is used, the simulation is stopped temporarily.
    // If $finish is used the simulator just exits.
   #2 $finish;
  end
/*-----------------------------
  Display/Monitor your signals
  -----------------------------*/
/*-----------------------------------------------------
  Dump the signal transition data for wavefrom viewing
  -----------------------------------------------------*/
  initial begin
    $shm_open ("testFadder.shm") ;
    $shm_probe("AS") ;
    $shm_close(0);
  end
endmodule
```

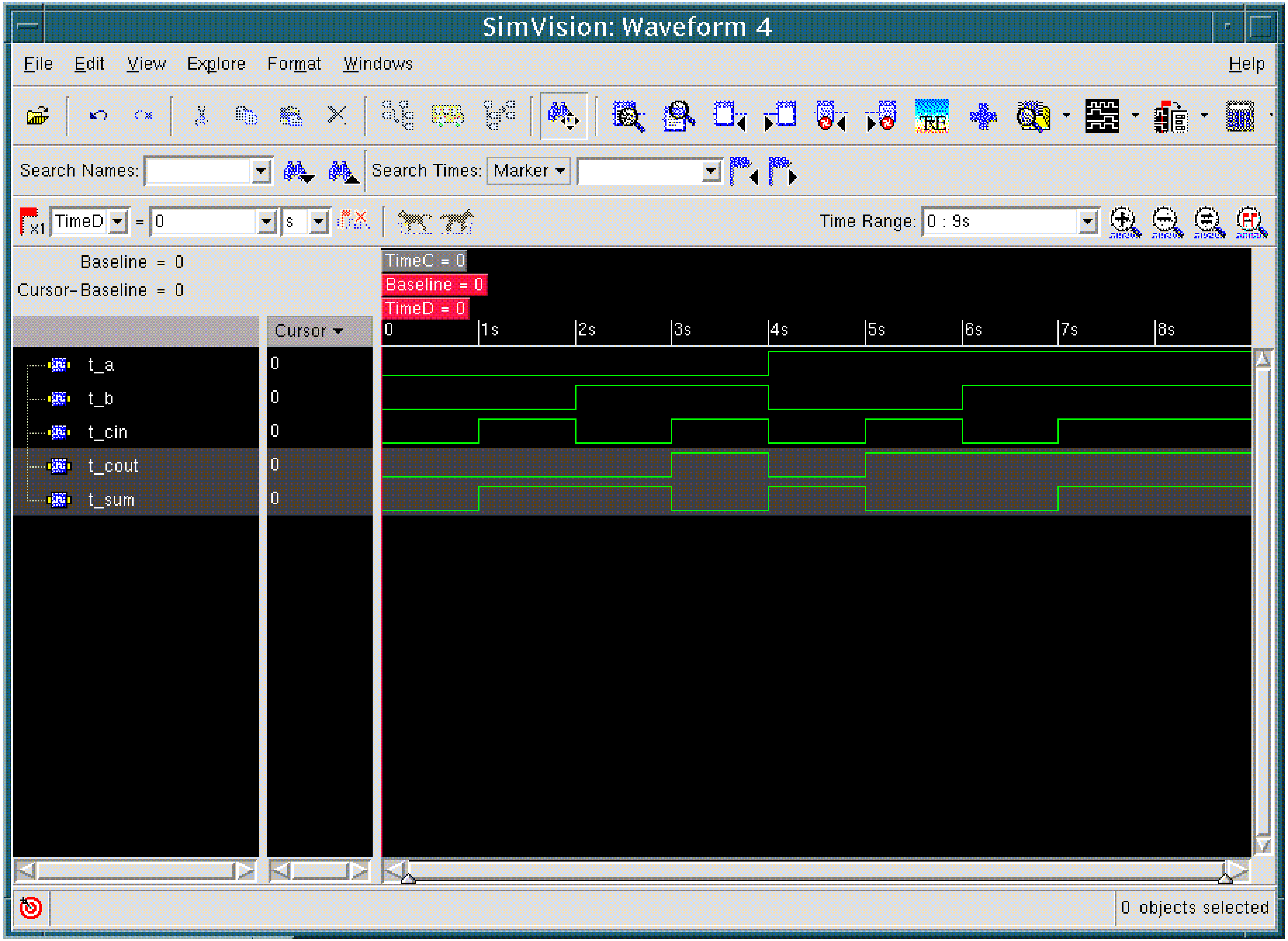
Terminal

Window Edit Options Help

```
THIS SOFTWARE AND ON-LINE DOCUMENTATION CONTAIN CONFIDENTIAL INFORMATION
AND TRADE SECRETS OF CADENCE DESIGN SYSTEMS, INC.  USE, DISCLOSURE, OR
REPRODUCTION IS PROHIBITED WITHOUT THE PRIOR EXPRESS WRITTEN PERMISSION OF
CADENCE DESIGN SYSTEMS, INC.
RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to
restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in
Technical Data and Computer Software clause at DFARS 252.227-7013 or
subparagraphs (c)(1) and (2) of Commercial Computer Software -- Restricted
Rights at 48 CFR 52.227-19, as applicable.

                Cadence Design Systems, Inc.
                555 River Oaks Parkway
                San Jose, California  95134

For technical assistance please contact the Cadence Response Center at
1-877-CDS-4911 or send email to support@cadence.com

For more information on Cadence's Verilog-XL product line send email to
talkv@cadence.com

Compiling source file "fadder.v"
Compiling source file "testfadder2.v"
Highest level modules:
testFadder

SST2 Database Write API -- DWAPI Version 04.10-p002 -- 10/15/2002
Copyright 1997-2002 Cadence Design Systems, Inc.

L68 "testfadder2.v": $finish at simulation time 9
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.1 secs in simulation
End of Tool:    VERILOG-XL      04.10.001-p    May 18, 2004  14:23:17
nws0300:/home/ben/tm3/teach>
```


SimVision: Utility
File
Windows
Help
Markers
Groups
Expressions
Cursors
Databases
Mnemonic Maps
TxE Manager
TxE
Open Database
Directory:
/home/ben/tm3/teach/testFadder.shm
testFadder.trn
File name:
Open
Files of type:
Transition Files (*.trn)
Cancel


SimVision: Design Browser 2
File
Edit
View
Select
Windows
Help
Scope Tree:
Signals/Variables of scope:
testFadder::testFadder
testFadder
testFadder
t_a
t_b
t_cin
t_cout
t_sum
Filter:
*
Filter:
*
1 object selected


SimVision: Waveform 4
File Edit View Explore Format Windows Help
Search Names:
Search Times: Marker
TimeD = 0 s
Time Range: 0 : 9s
Baseline = 0
Cursor-Baseline = 0
TimeC = 0
Baseline = 0
TimeD = 0
Cursor
0 1s 2s 3s 4s 5s 6s 7s 8s
t_a 0
t_b 0
t_cin 0
t_cout 0
t_sum 0
0 objects selected