
Advanced Testing using VHDL

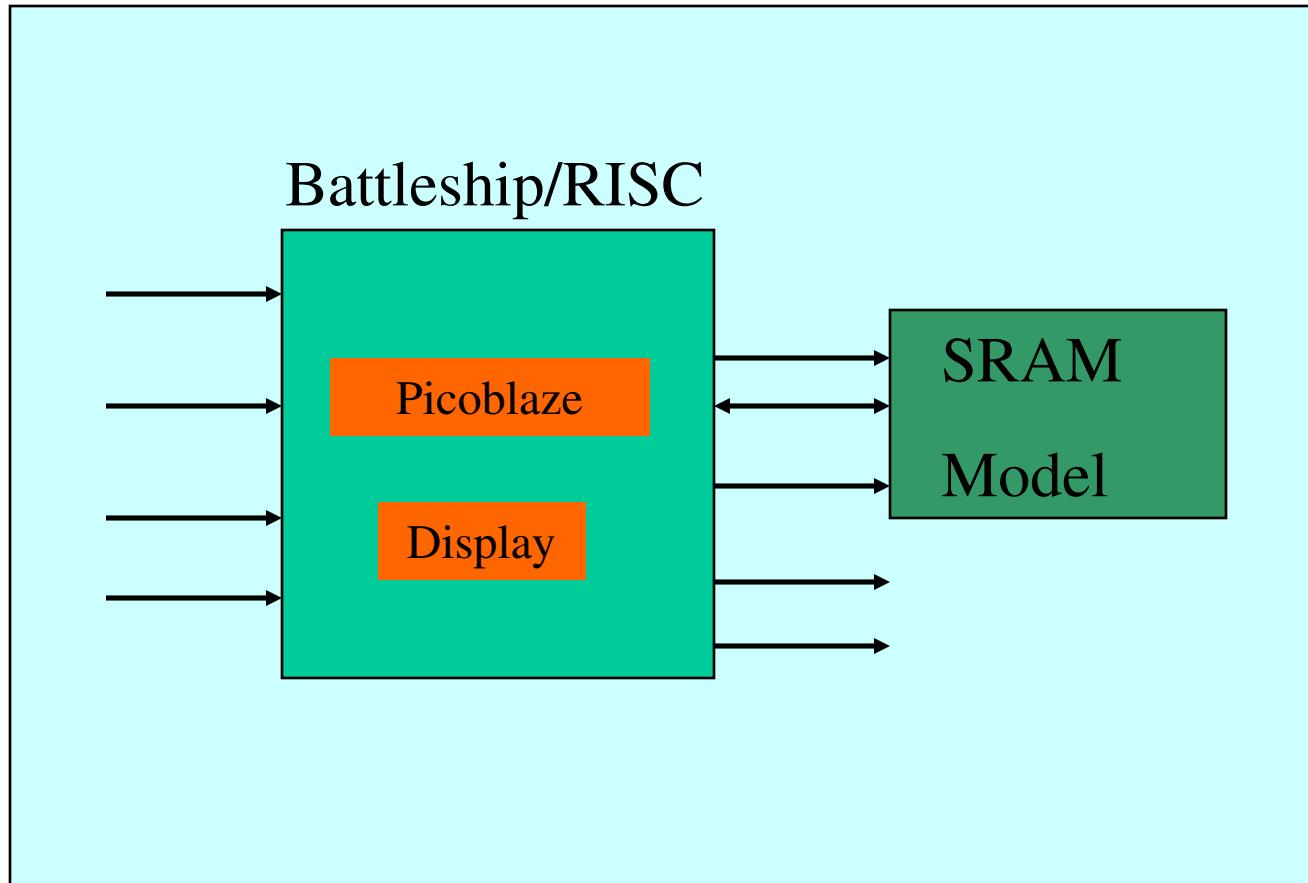
Module 9

Overview

- SRAM Model
- Attributes
- Loop Statements
- Test Bench examples using
 - TEXTIO
 - Conversion functions
 - Reading file containing test vectors
 - Writing test results to file

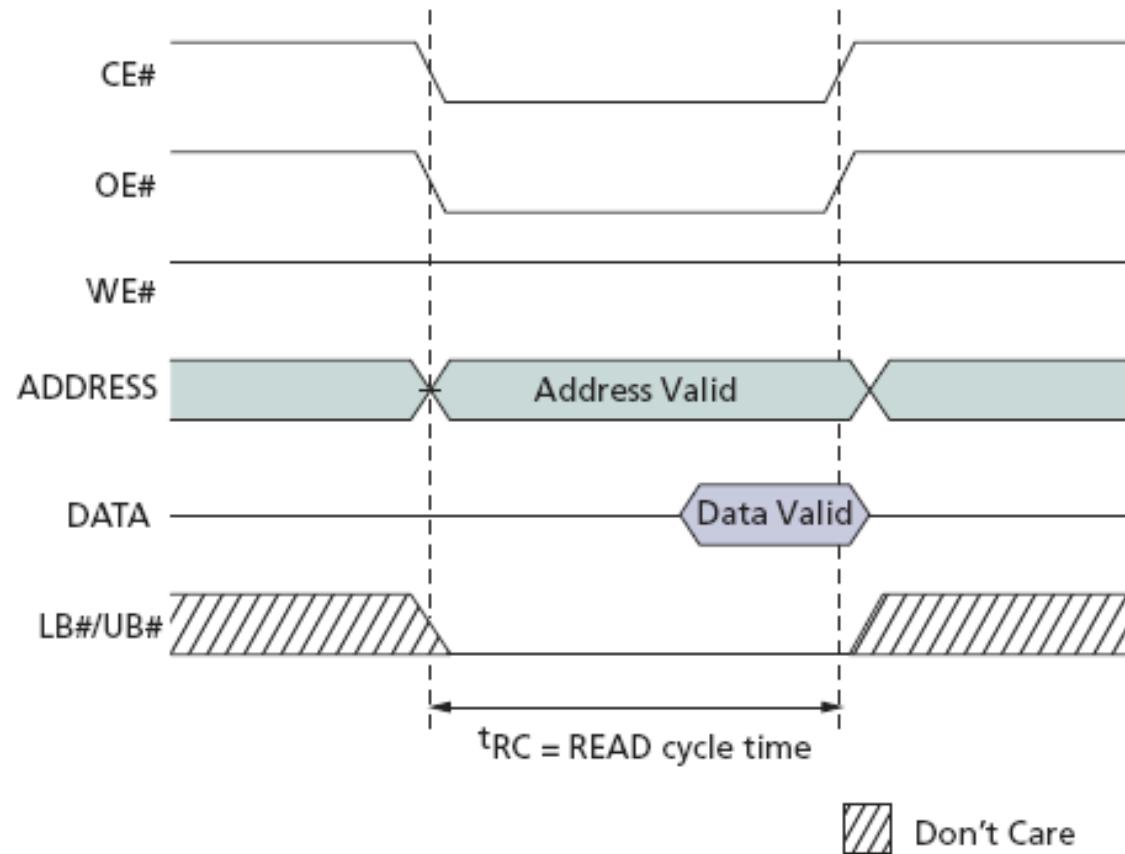
Adding the SRAM model

- New testbench

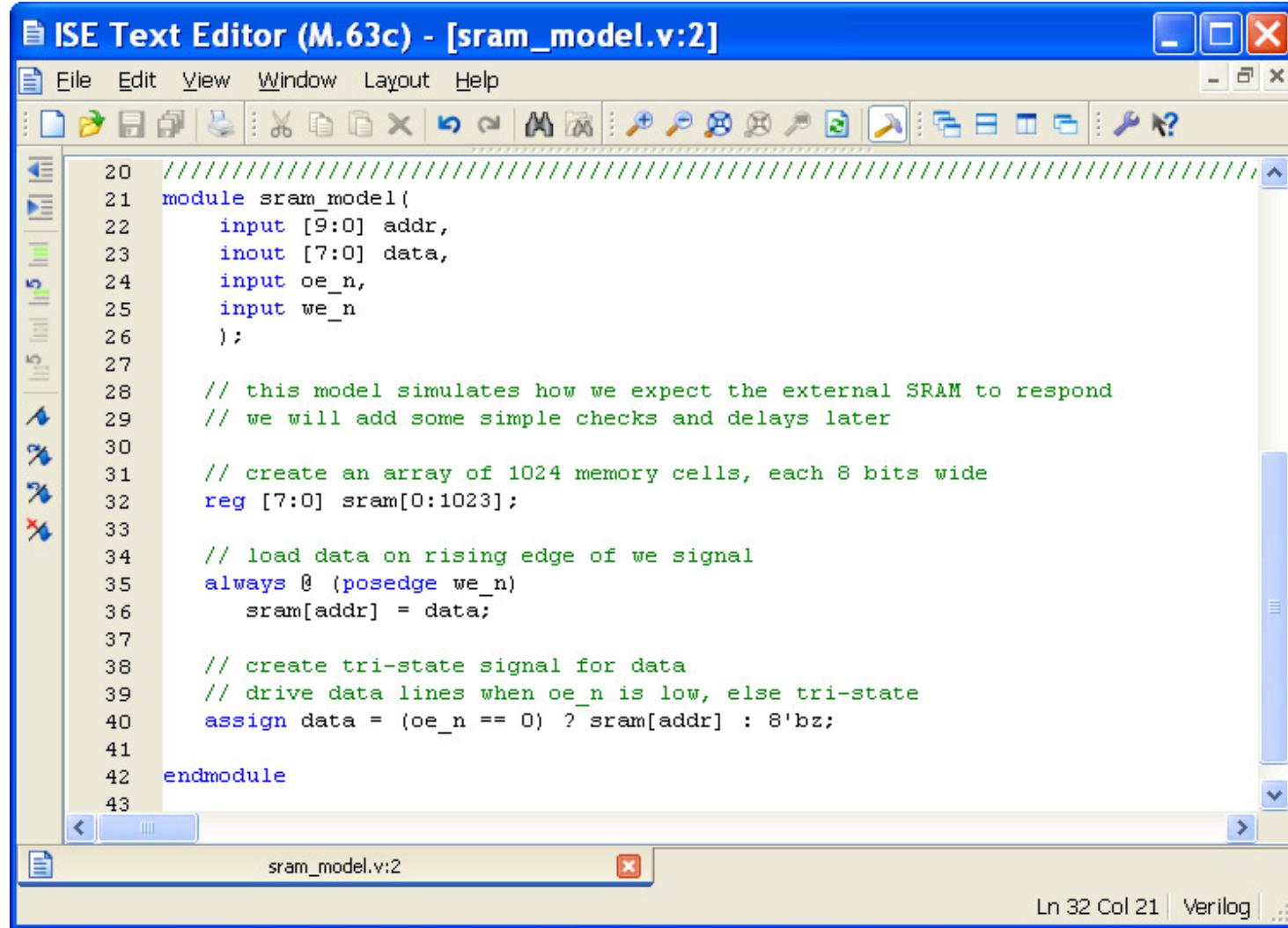


SRAM - Simplified Read Operation

READ Operation (ADV# LOW)



SRAM Model - Verilog

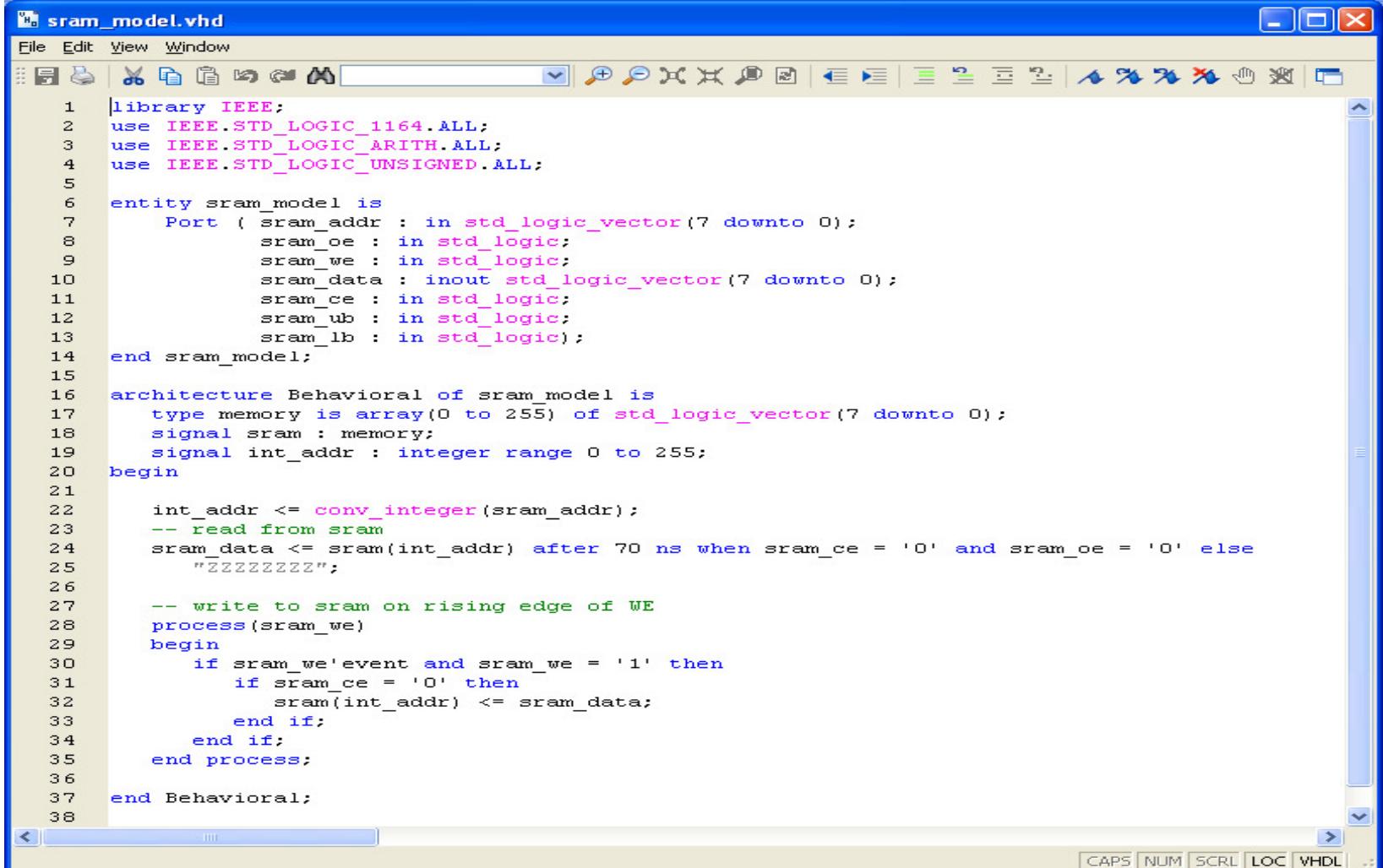


The screenshot shows the ISE Text Editor interface with the title bar "ISE Text Editor (M.63c) - [sram_model.v:2]". The menu bar includes File, Edit, View, Window, Layout, and Help. The toolbar contains various icons for file operations like Open, Save, Find, and Print. The main editor area displays the following Verilog code:

```
20 //////////////////////////////////////////////////////////////////
21 module sram_model(
22     input [9:0] addr,
23     inout [7:0] data,
24     input oe_n,
25     input we_n
26 );
27
28 // this model simulates how we expect the external SRAM to respond
29 // we will add some simple checks and delays later
30
31 // create an array of 1024 memory cells, each 8 bits wide
32 reg [7:0] sram[0:1023];
33
34 // load data on rising edge of we signal
35 always @ (posedge we_n)
36     sram[addr] = data;
37
38 // create tri-state signal for data
39 // drive data lines when oe_n is low, else tri-state
40 assign data = (oe_n == 0) ? sram[addr] : 8'bzz;
41
42 endmodule
43
```

The status bar at the bottom right shows "Ln 32 Col 21 | Verilog".

SRAM Model - VHDL



The screenshot shows a VHDL editor window titled "sram_model.vhd". The code implements a SRAM model with the following structure:

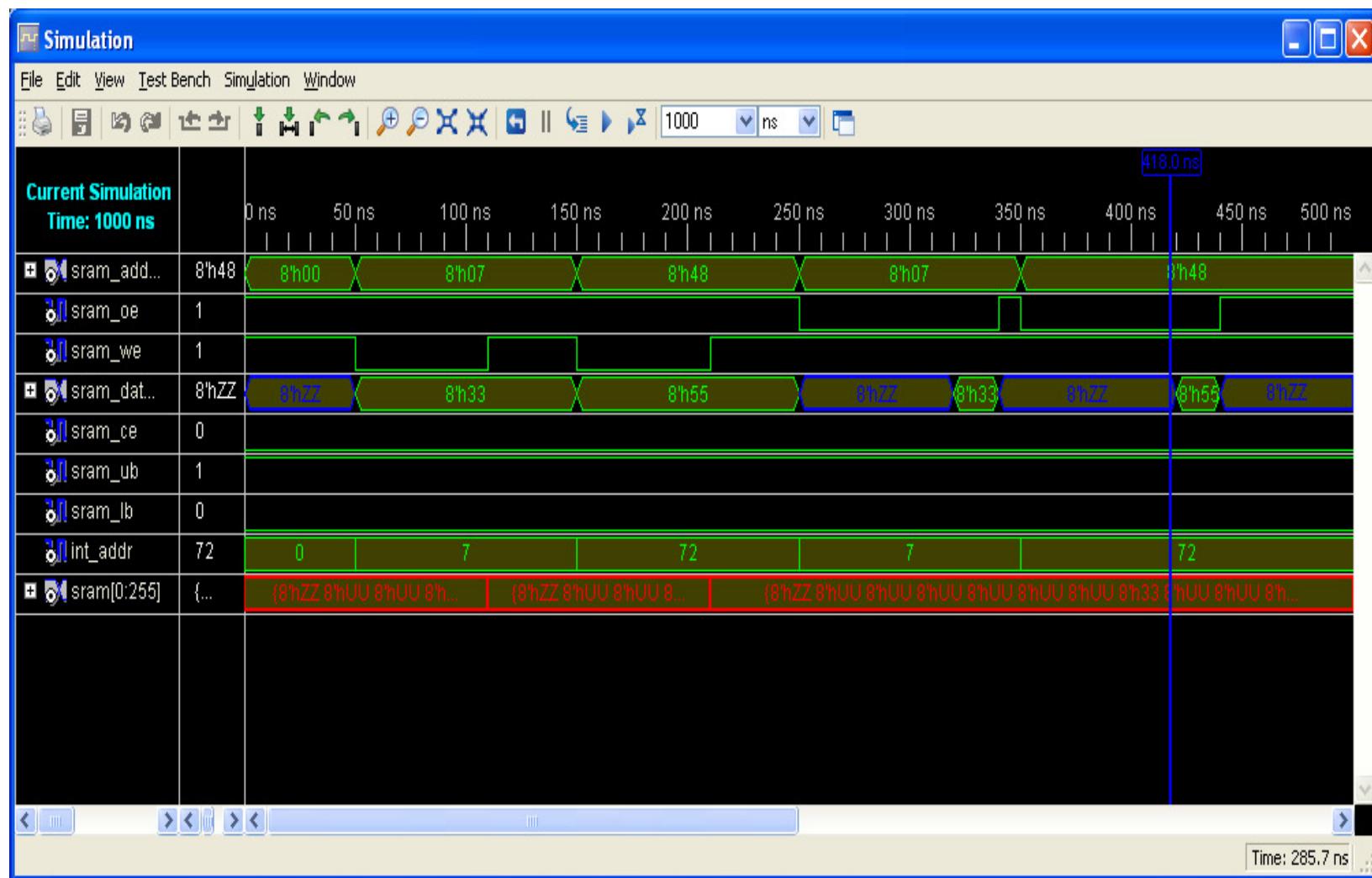
- Library and Use:** The code uses IEEE standard logic packages.
- Entity:** The entity is named "sram_model" with a single port containing six inputs: sram_addr, sram_oe, sram_we, sram_data, sram_ce, and sram_lb.
- Architecture:** The architecture is named "Behavioral". It defines a memory array from index 0 to 255, initializes it to all zeros ("ZZZZZZZZ"), and handles reads and writes based on control signals.
- Read Operation:** A process reads the memory at address int_addr when sram_ce = '0' and sram_oe = '0'. If either signal is '1', the result is "ZZZZZZZZ".
- Write Operation:** A process writes to memory at address int_addr if sram_we = '1'. It checks if sram_ce = '0' and updates the memory with sram_data.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sram_model is
    Port ( sram_addr : in std_logic_vector(7 downto 0);
           sram_oe : in std_logic;
           sram_we : in std_logic;
           sram_data : inout std_logic_vector(7 downto 0);
           sram_ce : in std_logic;
           sram_ub : in std_logic;
           sram_lb : in std_logic);
end sram_model;

architecture Behavioral of sram_model is
    type memory is array(0 to 255) of std_logic_vector(7 downto 0);
    signal sram : memory;
    signal int_addr : integer range 0 to 255;
begin
    int_addr <= conv_integer(sram_addr);
    -- read from sram
    sram_data <= sram(int_addr) after 70 ns when sram_ce = '0' and sram_oe = '0' else
        "ZZZZZZZZ";
    -- write to sram on rising edge of WE
    process(sram_we)
    begin
        if sram_we'event and sram_we = '1' then
            if sram_ce = '0' then
                sram(int_addr) <= sram_data;
            end if;
        end if;
    end process;
end Behavioral;
```

SRAM Model Read and Write



VHDL Attributes

- Signals can have attributes associated with them
- Example predefined signal attributes are
 - function
 - value
 - type
 - range
- Also possible to have user-defined attributes

Function Attributes

- Predefined functions
- Returns information about the behavior of signals
 - $s'EVENT$
 - returns true if an event occurred
 - change in value
 - $s'ACTIVE$
 - returns true if signal s is active
 - new value assigned to signal s (may be same value)
 - $s'LAST_EVENT$
 - returns elapsed time since last event
 - $s'LAST_ACTIVE$
 - $s'LAST_VALUE$
 - returns value of s before the last event

Function Attribute example

- Check for setup time violation on d input changing

```
CONSTANT setup_time : TIME := 12 ns; -- TIME is predefined

PROCESS(clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        ASSERT(d'LAST_EVENT >= setup_time)
            REPORT "setup violation"
            SEVERITY error;
    END IF;
```

- Assert statement checks that the input d has not had an event during the setup time.
- If time returned is less than setup time - assertion will fail

‘now’ function

- NOW
 - predefined function that returns simulation time

- ```
IF d'EVENT THEN
 lasteventonD := NOW;

IF clk'EVENT AND clk = '1' THEN
 ASSERT(now - lasteventonD) >= setup_time
 REPORT "setup violation"
 SEVERITY error
```

# RANGE attributes

---

- Only for constrained array types
- Returns range of specified type
- Two range attributes
  - `a'RANGE`
  - `a'REVERSE RANGE`

```
TYPE address_bus IS ARRAY (63 DOWNTO 0) OF std_logic;

SIGNAL cpu_address : address_bus; -- declare array

FOR j IN cpu_address'RANGE LOOP -- 63 DOWNTO 0
FOR j IN cpu_address'REVERSE_RANGE LOOP -- 0 TO 63
```

# VALUE and ARRAY Attributes

---

- These return the bounds of a type
- Four predefined attributes
  - $a'LEFT$ 
    - returns left bound of type
  - $a'RIGHT$ 
    - returns right bound of type
  - $a'HIGH$ 
    - returns upper bound of type
  - $a'LOW$ 
    - returns lower bound of type
- Also
  - $a'LENGTH$ 
    - returns total length of the array

# Value and Array Attribute examples

---

```
TYPE address_bus IS ARRAY (63 DOWNTO 0) OF std_logic;
 SIGNAL cpu_address : address_bus; -- declare array
BEGIN

 PROCESS
 VARIABLE i, j, k, l, m : INTEGER;
 BEGIN
 i := cpu_address'LEFT; -- 63
 j := cpu_address'RIGHT; -- 0
 k := cpu_address'HIGH; -- 63
 l := cpu_address'LOW; -- 0
 m := cpu_address'LENGTH; -- 64
 END PROCESS;
```

# Enumerated type example

---

```
TYPE days IS (mon, tues, wed, thurs, fri, sat, sun);
SUBTYPE weekend IS days RANGE sat TO sun;
SIGNAL day1, day2, day3, day4, day5 : days;
SIGNAL count : INTEGER;

BEGIN

PROCESS
BEGIN

 day1 <= days'LEFT; -- mon
 day2 <= days'RIGHT; -- sun
 day3 <= days'HIGH; -- sun
 day4 <= weekend'LOW; -- sat

 count <= days'POS(tues); -- 1
 day5 <= days'VAL(3)); -- thurs

END PROCESS;
```

# LOOP Statements

---

- Used to iterate through a set of sequential statements
- Three types

```
FOR identifier IN range LOOP
```

```
END LOOP;
```

```
WHILE boolean_expression LOOP
```

```
END LOOP;
```

```
LOOP
```

```
 EXIT WHEN condition_test
```

```
END LOOP;
```

# FOR LOOP

---

- general syntax

```
FOR identifier IN range LOOP
```

```
END LOOP;
```

- example:

```
factorial := 1;
FOR number IN 2 TO n LOOP
 factorial := factorial * number;
END LOOP;
```

- number = 2, 3, 4...
- no explicit declaration for loop identifier required
- also DOWNTO

# WHILE Loop

---

- general syntax

```
WHILE boolean_expression
 ...
END LOOP;
```

- example:

```
j := 0;
sum := 10;
wh_loop: WHILE j < 20 LOOP
 sum := sum * 2
 j := j + 3;
END LOOP wh_loop;
```

- Note: optional label for loop statement

# LOOP

---

- No iteration scheme
  - statements in body executed repeatedly until loop exits
- General syntax

```
LOOP
 EXIT WHEN boolean_expression; -- optional EXIT statement
END LOOP;
```

- Example:

```
j := 0;
sum := 1;
12: LOOP
 sum := sum * 10
 j := j + 17;
 EXIT WHEN sum > 100;
END LOOP 12;
```

- If exit statement not present loop executes indefinitely

# LOOP cont'd

---

- EXIT statement

- only used inside a loop

- General syntax

```
EXIT [loop_label] [WHEN condition]
 EXIT WHEN boolean_expression; -- optional EXIT statement
END LOOP;
```

- Example (alternative to previous example):

```
IF sum > 100 THEN
 EXIT;
END IF;
```

# LOOP cont'd

---

- NEXT statement
  - used to exit a loop for the current iteration of a loop
  - continue to the next iteration
- General syntax

```
NEXT [loop_label] [WHEN condition]
```

# LOOP example

---

```
PROCESS
BEGIN
11: LOOP
 -- statements
 --
12: LOOP
 -- statements
 --
 test_num := test_num + 1;
 EXIT 12 WHEN test_num = 25;
 IF solenoid_1 = '1' THEN
 drive_b := '0';
 NEXT 11; -- go to top of loop1
 END IF;
 IF trigger = '0' THEN
 -- statements
 END LOOP 12;
 EXIT 11 WHEN sim_tests = 200;
END LOOP 11;
```

# Test Bench example

---

- example test bench:

```
-- Test Bench to exercise and verify correctness of DECODE entity
ENTITY tb2_decode IS
END tb2_decode;

ARCHITECTURE test_bench OF tb2_decode IS
 TYPE input_array IS ARRAY (0 TO 3) OF std_logic_vector(1 DOWNTO 0);
 CONSTANT input_vectors: input_array :=
 ("00", "01", "10", "11");
 -- input_vectors array contains test vectors for input
 SIGNAL in1 : STD_LOGIC_VECTOR (1 DOWNTO 0);
 SIGNAL out1 : STD_LOGIC_VECTOR (3 DOWNTO 0);

COMPONENT decode
 PORT (
 sel : IN std_logic_vector(1 downto 0);
 y : OUT std_logic_vector(3 downto 0));
END COMPONENT;
```

# Test Bench example (cont'd)

---

```
BEGIN
 decode_1: decode PORT MAP(sel => in1, y => out1);

 apply_inputs: PROCESS
 BEGIN
 FOR j IN input_vectors'RANGE LOOP
 in1 <= input_vectors(j);
 WAIT FOR 50 ns;
 END LOOP;
 END PROCESS apply_inputs;

 test_outputs: PROCESS
 BEGIN
 WAIT UNTIL (in1 = "01");
 WAIT FOR 25 ns;
 ASSERT (out1 = "0110")
 REPORT "Output not equal to 0110"
 SEVERITY ERROR;
 END PROCESS test_outputs;
 END test_bench;
```

---

# Assert Statements

---

- During testing
  - Usually want to display information about signals and variables
- Assert statement is rather limited
  - Report clause only allows a single string
  - no built-in provision for formatting data

```
ASSERT FALSE
 REPORT "first line" & CR & "second line";
```

# Test Bench example

---

- Adapted from Pellerin and Taylor (pages 242-246)
  - demonstrates use of
    - textio to read in a test vector file
    - complex strings in Assert Statements
    - string to vector and vector to string conversion functions
  - uses a decoder component for test

# VHDL for Synthesis

The image shows two windows from the Xilinx ISE development environment. The left window is the 'decoder.vhd \* - ISE Text Editor' containing VHDL code for a decoder. The right window is the 'Xilinx ECS - [decoder.ngr]' schematic editor showing a logic diagram of the decoder.

**VHDL Code (decoder.vhd):**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity decoder is
 Port (sel : in std_logic_vector(2 downto 0);
 y : out std_logic_vector(7 downto 0));
end decoder;

architecture Behavioral of decoder is
begin
 y <= "00000001" when sel = "000" else
 "00000010" when sel = "001" else
 "00000100" when sel = "010" else
 "00001000" when sel = "011" else
 "00010000" when sel = "100" else
 "00100000" when sel = "101" else
 "01000000" when sel = "110" else
 "10000000";
end Behavioral;
```

**Schematic Diagram:**

The schematic diagram shows a 'Decoder' block with a single input 'sel(2:0)' and an output 'y(7:0)'. The output is labeled 'A<2:0>O<7:0>'.

## test\_vec.txt File

---

- Format is
  - 3 bits for SEL input
  - 8 bits for expected output

0000000001

0010000010

01000000100

01100001000

10000010000

10100110000

11001000000

11110000000

# Test Bench VHDL file

---

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE std.textio.ALL; -- use text I/O features of standard library

ENTITY test_bench IS
END test_bench;

ARCHITECTURE tb1 OF test_bench IS
COMPONENT decoder -- component to be tested
 PORT(sel : IN std_logic_vector(2 DOWNTO 0);
 y : OUT std_logic_vector(7 DOWNTO 0));
END COMPONENT;

-- FOR all: decoder USE ENTITY work.decoder; -- configuration
```

# Test Bench VHDL file (cont'd)

---

```
-- function to convert string of character to vector
FUNCTION str2vec(str : string) RETURN std_logic_vector IS
 VARIABLE vtmp: std_logic_vector(str'RANGE);
BEGIN
 FOR i IN str'RANGE LOOP
 IF str(i) = '1' THEN
 vtmp(i) := '1';
 ELSIF str(i) = '0' THEN
 vtmp(i) := '0';
 ELSE
 vtmp(i) := 'X';
 END IF;
 END LOOP;
 RETURN vtmp;
END str2vec;
```

# Test Bench VHDL file (cont'd)

---

```
-- function to convert vector to string
-- for use in assert statements
FUNCTION vec2str(vec : std_logic_vector) RETURN string IS
 VARIABLE stmp : string(vec'LEFT+1 DOWNTO 1);
BEGIN
 FOR i IN vec'REVERSE_RANGE LOOP
 IF vec(i) = '1' THEN
 stmp(i+1) := '1';
 ELSIF vec(i) = '0' THEN
 stmp(i+1) := '0';
 ELSE
 stmp(i+1) := 'X';
 END IF;
 END LOOP;
 RETURN stmp;
END vec2str;
```

# Test Bench VHDL file (cont'd)

---

```
SIGNAL clk : std_logic := '0';

-- create internal signals to connect to test component
SIGNAL sel : std_logic_vector(2 DOWNTO 0);
SIGNAL y : std_logic_vector(7 DOWNTO 0);

BEGIN

-- instantiate decoder test component
u1: decoder PORT MAP(sel => sel, y => y);

clk <= NOT clk AFTER 50 ns; -- create clock for timing
```

# Test Bench VHDL file (cont'd)

---

```
PROCESS
 -- declare and open file (1987 style)
 FILE vector_file: text IS in "test.vec";
 VARIABLE file_line : line; -- text line buffer
 VARIABLE str_stimulus_in: string(11 DOWNTO 1);
 VARIABLE stimulus_in : std_logic_vector(10 DOWNTO 0);
 VARIABLE y_expected : std_logic_vector(7 DOWNTO 0);

BEGIN
 -- loop through lines in test file
 WHILE NOT endfile(vector_file) LOOP

 -- read one complete line into file_line
 readline(vector_file, file_line);

 -- extract the first field from file_line
 read(file_line, str_stimulus_in);

 -- convert string to vector
 stimulus_in := str2vec(str_stimulus_in);
```

# Test Bench VHDL file (cont'd)

---

```
WAIT UNTIL clk = '1'; -- rising edge of clock
-- now get sel input and apply to decoder
sel <= stimulus_in(10 DOWNTO 8); -- top 3 bits for input;

WAIT UNTIL clk = '0'; -- falling edge
-- now check if decoder outputs are correct
y_expected := stimulus_in(7 DOWNTO 0); -- expected output

-- compare decoder output with expected value
IF y /= y_expected THEN
 ASSERT false
 REPORT "decoder failure" & CR &
 "Expected y to be " & vec2str(y_expected) &
 " but its value was " & vec2str(y)
 SEVERITY ERROR;
END IF;
END LOOP;

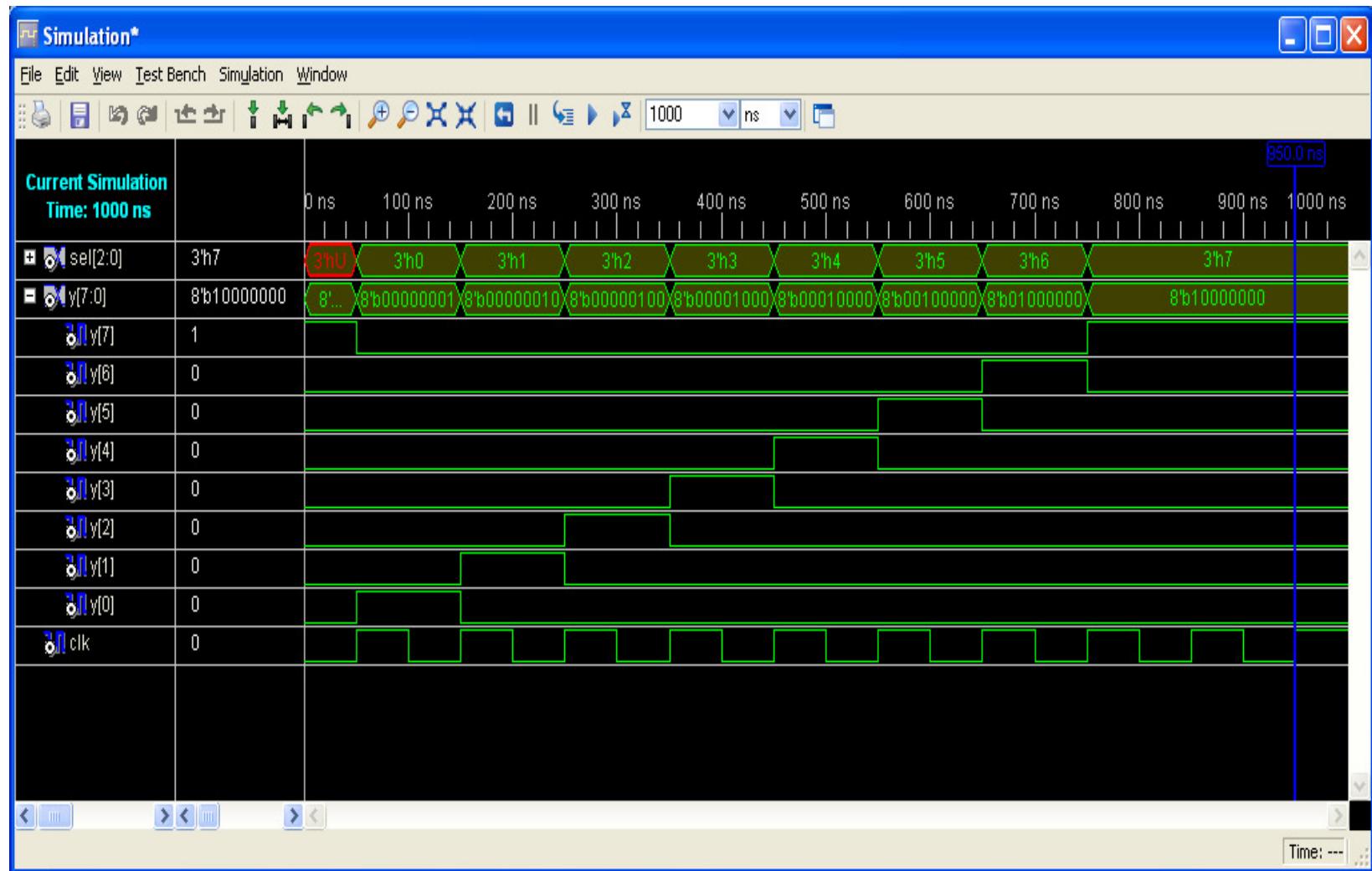
WAIT; -- suspend the simulation
END PROCESS;
END tb1;
```

# VHDL Test Bench - Results

The screenshot shows the Xilinx ISE Design Suite 10.1 interface with the following details:

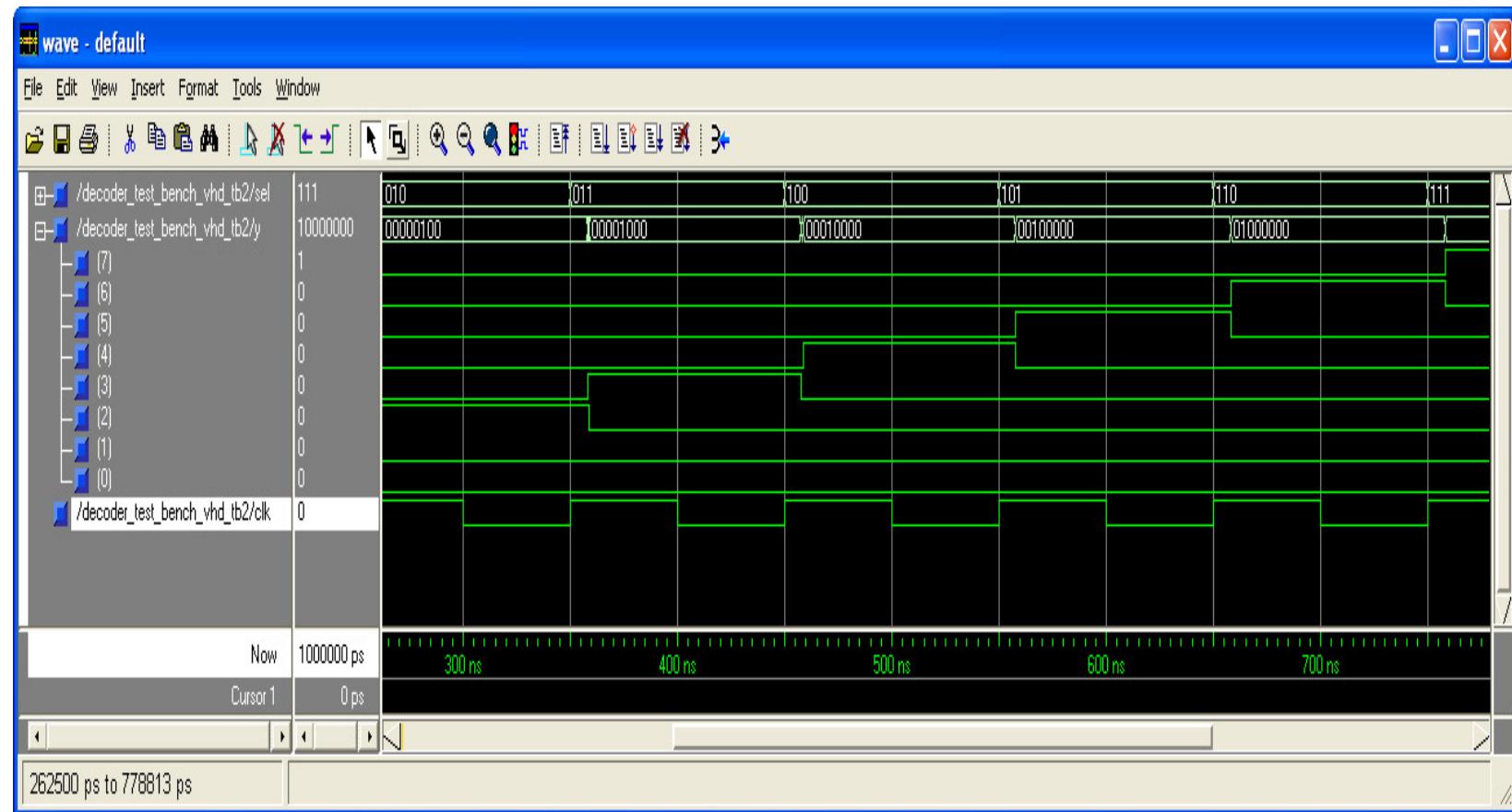
- File Menu:** File, Edit, View, Project, Source, Process, Window, Help.
- Sources Panel:** Shows files for Behavioral Simulation: test\_bench\_decoder, test\_vec.txt, xc3s200-4n256, and decoder\_test\_bench\_vhd\_tb - behavior (I). The test\_bench\_decoder file is selected.
- Processes Panel:** Shows options for adding sources, running behavioral checks, and simulating behavioral models. The "Behavioral Check Syntax" option is highlighted.
- Code Editor:** Displays the VHDL code for the test bench. The code reads test vectors from a file, applies them to a decoder, and compares the output against expected values. It includes assertions and error reporting.
- Transcript Window:** Shows the simulation log, indicating a failure at 600.000 ns where the expected output was 00110000 but the actual value was 00100000.
- Bottom Navigation:** Includes tabs for Console, Errors, Warnings, Tcl Shell, Find in Files, Sim Console - decoder\_test\_bench\_vhd\_tb, and CAPS/NUM/SCRL/Ln1 Col1/VHDL buttons.

# Test Bench Waveform

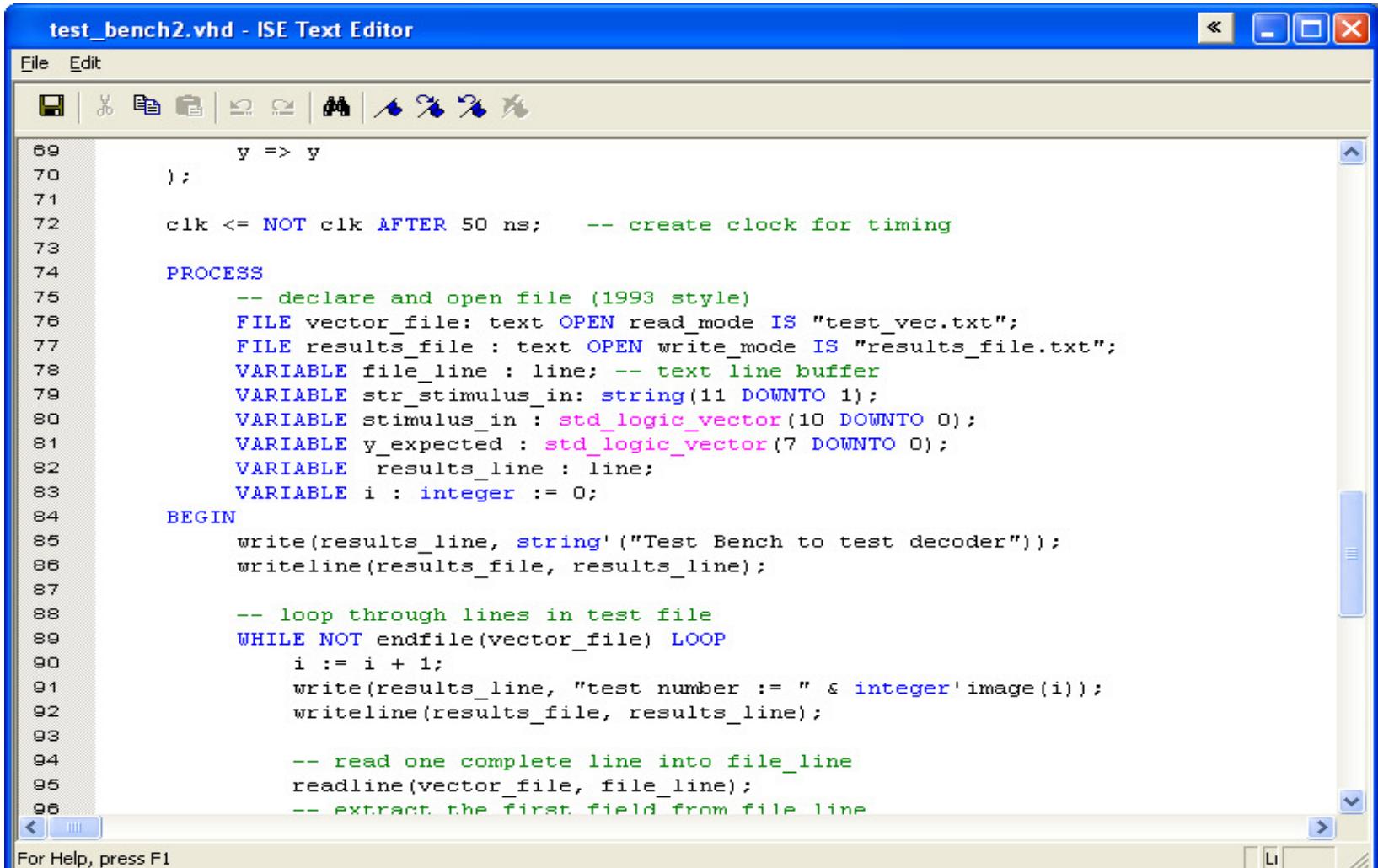


# Simulation of post-place and route

---



# Writing Test Results to File

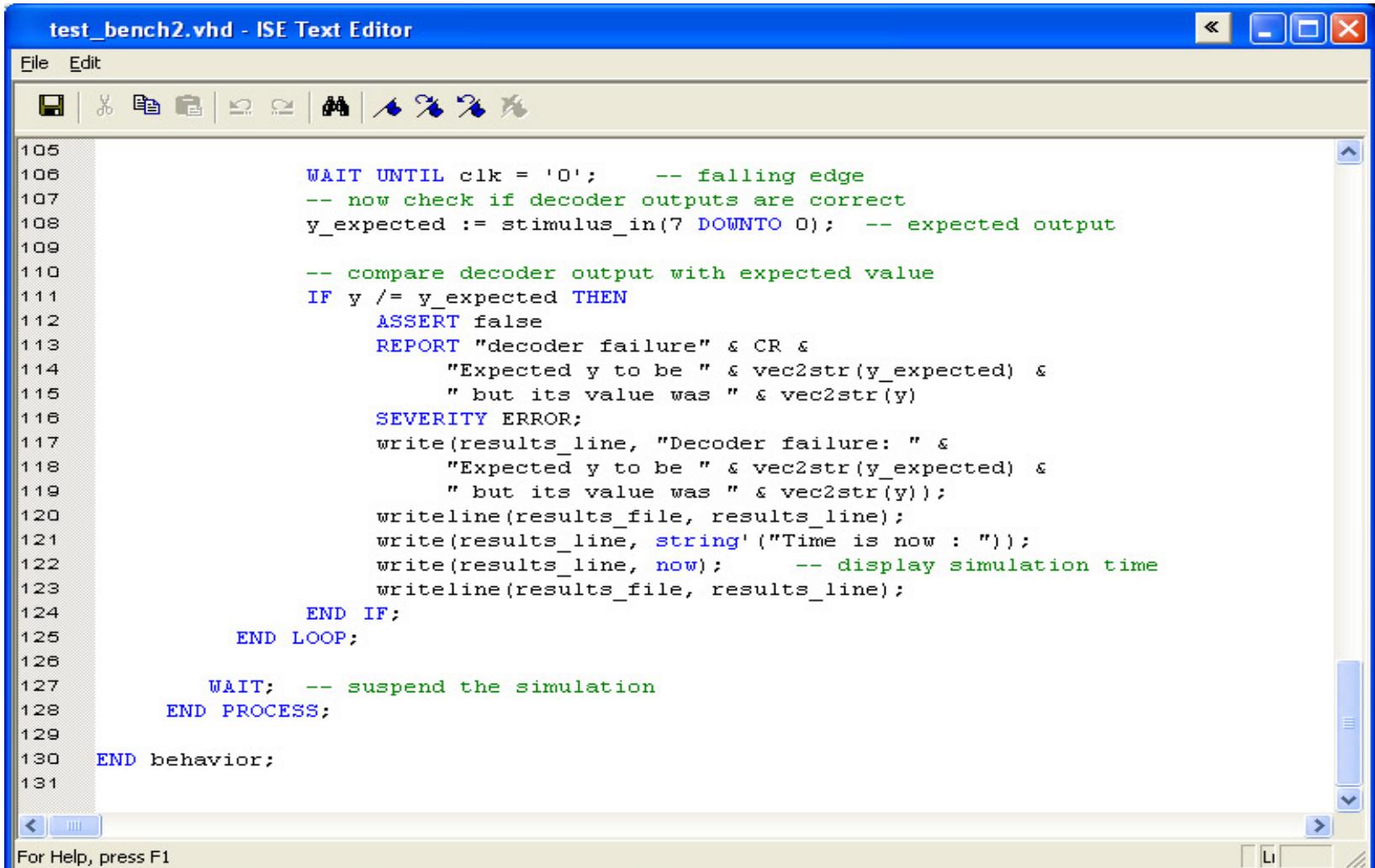


The screenshot shows the ISE Text Editor window with the title "test\_bench2.vhd - ISE Text Editor". The code is written in VHDL and performs the following tasks:

- Line 69: `y => y`
- Line 70: A comment block starting with `---`.
- Line 71: A clock assignment: `clk <= NOT clk AFTER 50 ns;`
- Line 72: A comment: `-- create clock for timing`
- Line 73: A blank line.
- Line 74: `PROCESS` keyword.
- Line 75: A comment: `-- declare and open file (1993 style)`
- Line 76: `FILE vector_file: text OPEN read_mode IS "test_vec.txt";`
- Line 77: `FILE results_file : text OPEN write_mode IS "results_file.txt";`
- Line 78: `VARIABLE file_line : line; -- text line buffer`
- Line 79: `VARIABLE str_stimulus_in: string(11 DOWNTO 1);`
- Line 80: `VARIABLE stimulus_in : std_logic_vector(10 DOWNTO 0);`
- Line 81: `VARIABLE y_expected : std_logic_vector(7 DOWNTO 0);`
- Line 82: `VARIABLE results_line : line;`
- Line 83: `VARIABLE i : integer := 0;`
- Line 84: `BEGIN` keyword.
- Line 85: `write(results_line, string'("Test Bench to test decoder"));`
- Line 86: `writeline(results_file, results_line);`
- Line 87: A blank line.
- Line 88: `-- loop through lines in test file`
- Line 89: `WHILE NOT endfile(vector_file) LOOP`
- Line 90: `i := i + 1;`
- Line 91: `write(results_line, "test number := " & integer'image(i));`
- Line 92: `writeline(results_file, results_line);`
- Line 93: A blank line.
- Line 94: `-- read one complete line into file_line`
- Line 95: `readline(vector_file, file_line);`
- Line 96: `-- extract the first field from file_line`

At the bottom of the editor window, there is a status bar with the text "For Help, press F1".

# Writing Test Results to File (cont'd)



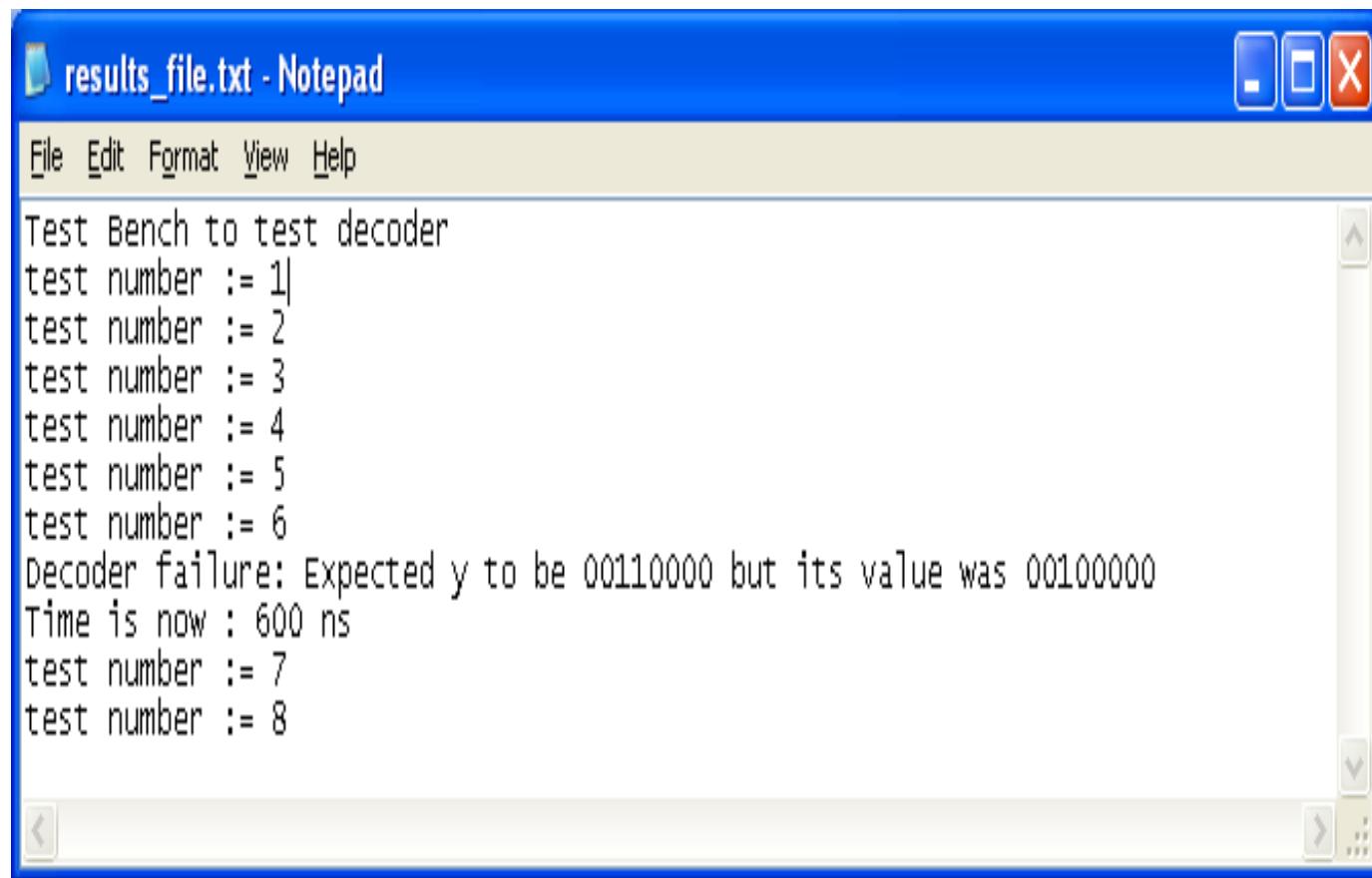
The screenshot shows the ISE Text Editor window with the title "test\_bench2.vhd - ISE Text Editor". The code in the editor is as follows:

```
105 WAIT UNTIL clk = '0'; -- falling edge
106 -- now check if decoder outputs are correct
107 y_expected := stimulus_in(7 DOWNTO 0); -- expected output
108
109 -- compare decoder output with expected value
110 IF y /= y_expected THEN
111 ASSERT false
112 REPORT "decoder failure" & CR &
113 "Expected y to be " & vec2str(y_expected) &
114 " but its value was " & vec2str(y)
115 SEVERITY ERROR;
116 write(results_line, "Decoder failure: " &
117 "Expected y to be " & vec2str(y_expected) &
118 " but its value was " & vec2str(y));
119 writeline(results_file, results_line);
120 write(results_line, string'("Time is now : "));
121 write(results_line, now); -- display simulation time
122 writeline(results_file, results_line);
123
124 END IF;
125 END LOOP;
126
127 WAIT; -- suspend the simulation
128 END PROCESS;
129
130 END behavior;
131
```

The code implements a test bench for a decoder. It waits for a falling edge of the clock signal. Then, it compares the decoder's output vector (y) with the expected output vector (y\_expected). If they differ, it asserts an error, prints a report with the expected and actual values, and writes this information to a results file. It also displays the current simulation time. Finally, it suspends the simulation.

# results\_file

---



A screenshot of a Windows Notepad window titled "results\_file.txt - Notepad". The window contains the following text:

```
Test Bench to test decoder
test number := 1
test number := 2
test number := 3
test number := 4
test number := 5
test number := 6
Decoder failure: Expected y to be 00110000 but its value was 00100000
Time is now : 600 ns
test number := 7
test number := 8
```

# Assert Statement Limitations

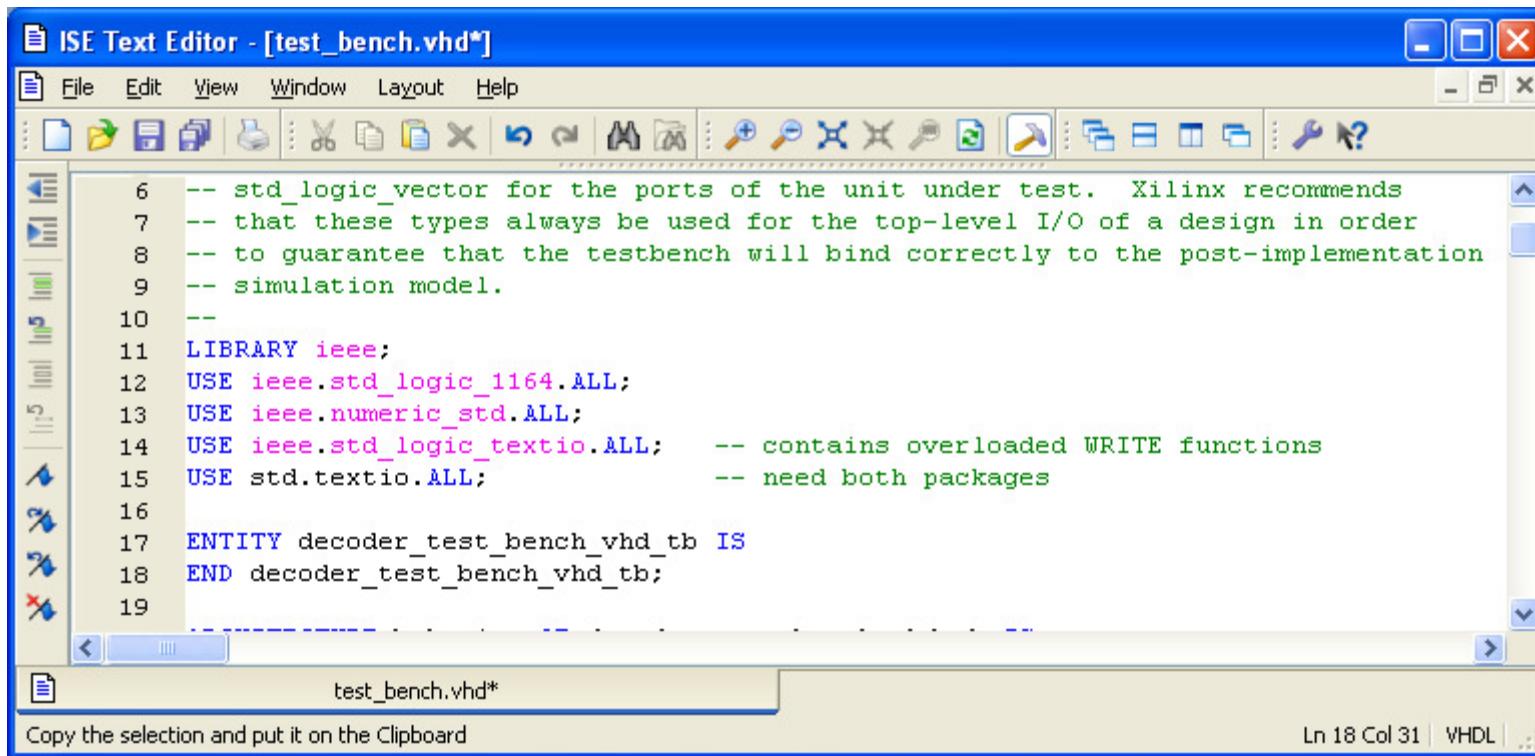
---

- Assert statements are limited to strings only
  - need conversion functions for displaying signals
- Use TEXTIO instead of Assert Statements to print messages
  - supports most data types
  - allows writing to multiple files

# Improving output messages

---

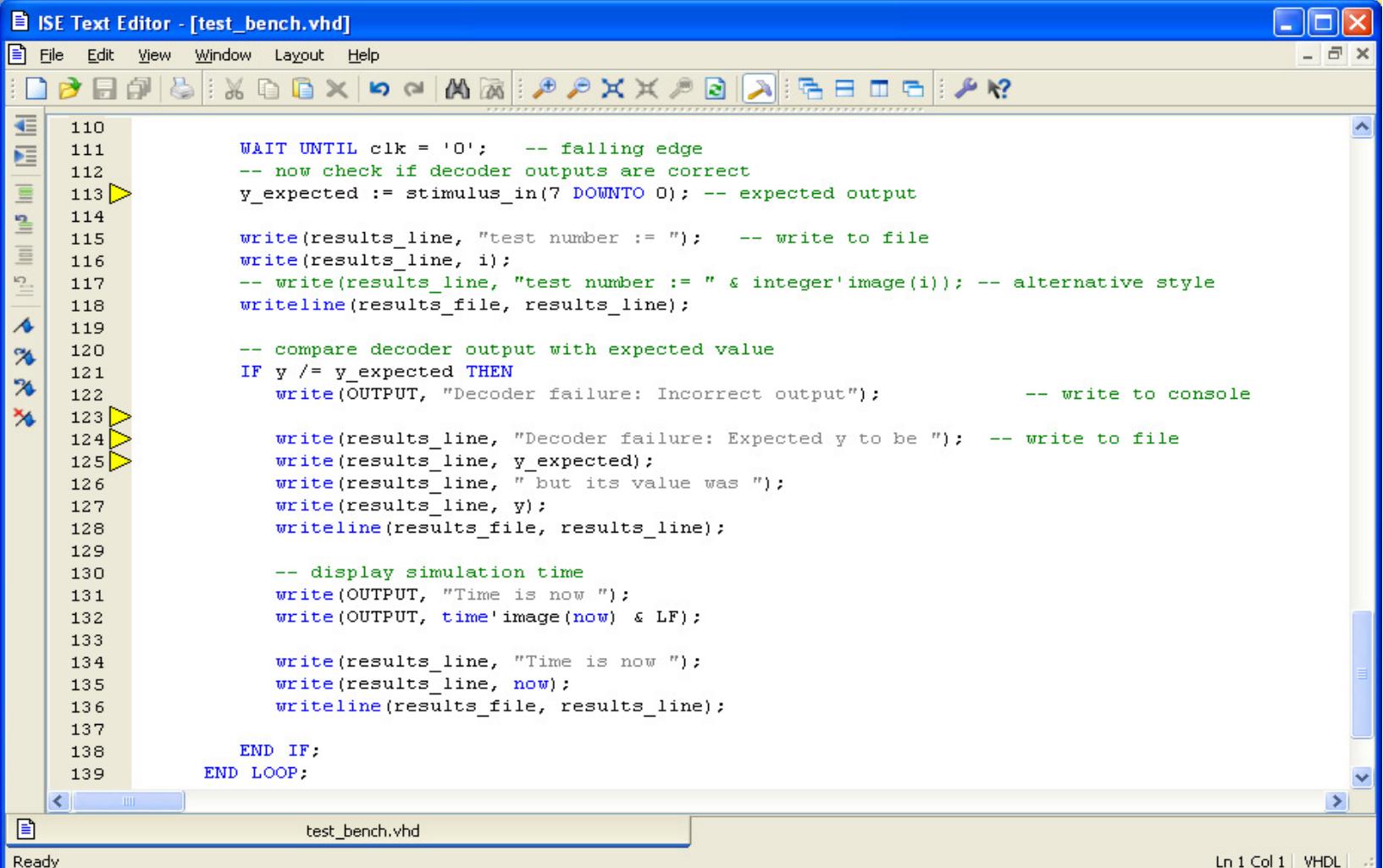
- Use ieee.std\_logic\_textio.ALL package
- Removes need for conversion functions



The screenshot shows the ISE Text Editor interface with a VHDL file named "test\_bench.vhd\*". The code uses the IEEE std\_logic\_textio package instead of conversion functions.

```
6 -- std_logic_vector for the ports of the unit under test. Xilinx recommends
7 -- that these types always be used for the top-level I/O of a design in order
8 -- to guarantee that the testbench will bind correctly to the post-implementation
9 -- simulation model.
10 --
11 LIBRARY ieee;
12 USE ieee.std_logic_1164.ALL;
13 USE ieee.numeric_std.ALL;
14 USE ieee.std_logic_textio.ALL; -- contains overloaded WRITE functions
15 USE std.textio.ALL; -- need both packages
16
17 ENTITY decoder_test_bench_vhd_tb IS
18 END decoder_test_bench_vhd_tb;
19
```

# Modified write statements



The screenshot shows the ISE Text Editor interface with the title bar "ISE Text Editor - [test\_bench.vhd]". The menu bar includes File, Edit, View, Window, Layout, and Help. The toolbar contains various icons for file operations like Open, Save, Find, and Print. The main code editor area displays the following VHDL code:

```
110 WAIT UNTIL clk = '0'; -- falling edge
111 -- now check if decoder outputs are correct
112 y_expected := stimulus_in(7 DOWNTO 0); -- expected output
113
114 write(results_line, "test number := "); -- write to file
115 write(results_line, i);
116 -- write(results_line, "test number := " & integer'image(i)); -- alternative style
117 writeline(results_file, results_line);
118
119 -- compare decoder output with expected value
120 IF y /= y_expected THEN
121 write(OUTPUT, "Decoder failure: Incorrect output"); -- write to console
122
123 write(results_line, "Decoder failure: Expected y to be ");
124 write(results_line, y_expected);
125 write(results_line, " but its value was ");
126 write(results_line, y);
127 writeline(results_file, results_line);
128
129 -- display simulation time
130 write(OUTPUT, "Time is now ");
131 write(OUTPUT, time'image(now) & LF);
132
133
134 write(results_line, "Time is now ");
135 write(results_line, now);
136 writeline(results_file, results_line);
137
138 END IF;
139 END LOOP;
```

The code implements a test bench loop. It waits for a falling edge of the `clk` signal. Then, it checks if the decoder outputs are correct by comparing them with the expected output from `stimulus_in`. If there is a mismatch, it writes an error message to the console and the results file, along with the expected and actual values. It also displays the current simulation time. Finally, it writes the current time and the value of `now` to the results file.