VHDL
Basic I/O

EL 310

Erkay Savaş

Sabancı University

Basic I/O and Its Applications

- Objects of `file` type
  - it is a special type that serve as an interface between the VHDL programs and the host environment.
- Motivation
  - how file objects are created, read, written and used within VHDL simulations
- To get it right
  - File input and output cannot be synthesized
  - I/O operations do not refer to I/O pins of FPGA chips

Test Benches with File Object

- Test bench
  - VHDL programs for testing VHDL models
- A typical test bench
  - reads test inputs from a file
  - applies them to the VHDL model under test
  - and records model outputs for analysis

Basic I/O Operations

- Type of a file object
  - depends what sort of data is stored in them
  - can be anything: integer, string, real number, std_logic_vector, etc.
- Three types of basic operations
  - Declaration of a file and its type
  - opening and closing a file of a specified type
  - reading and writing a file.

File Declarations

- Syntax
  - **type** `TEXT` **is file of** `string; -- ASCII`
  - **type** `INTF` **is file of** `integer;`
  - The first types of files contain ASCII characters, that form human readable text.
  - The second can store sequence of integers that are stored in binary form.
  - `string` and `integer` are predefined types of the language and can be found in the package STANDARD.
  - **file** `integer_file: INTF;`
  - **file** `input_file: TEXT;`

File Declarations

- TEXT is also a predefined type in the package TEXTIO.
  - we will learn more about it.
- The files `integer_file` and `input_file` can be thought of pointers to files contains sequences of integers and characters, respectively.
  - they are also being called as file handles.

Opening and Closing Files

- After declaration, files must be opened prior to use
- After use the files must be closed
- We have procedures for opening and closing files:
- **procedure** FILE_OPEN(**file** file_handle: FILE_TYPE;
  File_Name: **in** STRING;
  Open_Kind: **in** FILE_OPEN_KIND:=READ_MODE);
- **procedure** FILE_OPEN(File_Status: **out** FILE_OPEN_STATUS;
  file file_handle: FILE_TYPE;
  File_Name: **in** STRING;
  Open_Kind: **in** FILE_OPEN_KIND:=READ_MODE);
- **procedure** FILE_CLOSE(**file** file_handle: FILE_TYPE);

Opening and Closing Files

- `file file_handle: FILE_TYPE;` -- pointer to the file
- `File_Name: in STRING` -- name of the file
- `Open_Kind: in FILE_OPEN_KIND:=READ_MODE` -- in which mode the file is to be opened.
- The opening modes for a file:
  - READ_MODE -- default mode
  - WRITE_MODE
  - APPEND_MODE
- `File_Status: out FILE_OPEN_STATUS` -- the result of the procedure FILE_OPEN. Four values:
  - OPEN_OK
  - STATUS_ERROR
  - NAME_ERROR
  - MODE_ERROR

Example

```
-- declare a file type in the architecture declarative region
  type IntegerFileType is file of integer;
  process is
    file data_in: IntegerFileType; -- declare the file handle
    variable fstatus: file_open_status; -- declare file
                  -- status variable of file_open_status type
  -- other declarations
begin
  file_open(fstatus, datain, “myfile.txt”, read_mode);
  --
  -- body of process; reading and writing files and
  -- performing computations
  --
end process;
-- termination implicitly causes a call to FILE_CLOSE
```

Example: Implicit File Open

```
-- declare a file type in the architecture declarative region
  type IntegerFileType is file of integer;
  process is
    -- implicitly open a file in file declaration
    file data_in: IntegerFileType open read_mode is
    “my_file.txt”;
     -- other declarations
  begin
    --
    -- body of process; reading and writing files and
    -- performing computations
    --
  end process;
  -- termination implicitly causes a call to FILE_CLOSE
```

Reading & Writing Files

- The standard VHDL subprograms
  - **procedure** READ(**file** file_handle: FILE_TYPE; value: **out** type);
  - **procedure** WRITE(**file** file_handle: FILE_TYPE; value: **in** type);
  - **function** ENDFILE(**file** file_handle: FILE_TYPE) **return** Boolean;
- The read/write functions can be used after the file is opened.
- The language supports reading and writing from files of the predefined types of the language.
- For other types, you need to write your own I/O procedures built on these basic procedures.

VHDL 1987 I/O

- VHDL 1993
  - **file** infile: text **open** read_mode **is** “inputdata.txt”;
  - **file** outfile: text **open** write_mode **is** “outputdata.txt”;
- VHDL 1987
  - **file** infile: text **in is** “inputdata.txt”;
  - **file** outfile: text **out is** “outputdata.txt”;
  - **procedure** READ(**file** file_handle: FILE_TYPE; value: **out** type);
  - **procedure** WRITE(**file** file_handle: FILE_TYPE; value: **in** type);
  - **function** ENDFILE(**file** file_handle: FILE_TYPE) **return** Boolean;
  - No explicit FILE_OPEN and FILE_CLOSE

Example

```
entity io93 is
end entity io93;
architecture beh of io93 is
begin
  process is
    type IntegerFileType is file of integer;
    file data_out: IntegerFileType;
    variable fstatus: FILE_OPEN_STATUS;
    variable count: integer:= 0;
  begin
    file_open(fstatus, data_out, “myfile.txt”, write_mode);
    for i in 1 to 8 loop
      write(data_out, count);
      count := count + 2;
    end loop;
    wait;
  end process;
end architecture beh;
```

The Package TEXTIO

- A standard package supported by all VHDL simulators.
- It provides a standard set of file types, data types, and I/O functions.
- TEXTIO is in the library STD;
- The library STD does not have to be explicitly declared.
- However, the packages must be declared in order to use the package content
  - **use** `STD.textio`.**all**;

The Package TEXTIO

- Standard file type: TEXT
  - package provides the procedures for reading and writing the predefined types of the language such as bit, integer, and character.
  - See Appendix F in the textbook.
- Several lines
  - `-- A LINE is a pointer to a string`
    `type LINE is access STRING;`
  - `-- A file of variable-length ASCII records`
    `type TEXT is file of STRING;`
  - `procedure READLINE(file F: TEXT; L: out LINE);`
  - `procedure READ(L: inout LINE; value: out bit);`
  - `procedure WRITELINE(file F: TEXT; L: inout LINE);`
  - `procedure WRITE(L: inout LINE; value: out bit);`

TEXTIO Mechanism

- `LINE` serve as a buffer area for reading and writing
  - `read()` and `write()` procedures access and operate on this buffer
  - They are overloaded and defined for `bit`, `bit_vector`, and `string`.
  - `readline()` and `writeline()` procedures move the contents of this buffer to and from files.
- Access types are similar to pointers in Pascal and C languages.
- There are two special file handles called `input` and `output` that are defined in the package TEXTIO.

TEXTIO: `input` *and* `output`

- Two special predefined file handles
- `input` and `output`
- they are mapped to the `std_input` and `std_output` of the host environment that is the console window of the VHDL simulator.
  - **file** INPUT: TEXT **open** READ_MODE **is** "STD_INPUT";
  - **file** OUTPUT: TEXT **open** WRITE_MODE **is** "STD_OUTPUT";

Example: TEXTIO

```
use STD.textio.all;
entity formatted_io is
end entity;
architecture beh of formatted_io is
begin
process is
  file outfile: text;
  variable f_status: FILE_OPEN_STATUS;
  variable count: integer := 5;
  variable value: bit_vector(3 downto 0) := x”6”;
  variable buf: LINE; -- buffer between the program and file
begin
  file_open(fstatus, outfile, “myfile.txt”, write_mode);
  ...
end process;
end architecture beh;
```

Example: TEXTIO

```
...
process is
begin
  file_open(f_status, outfile, “myfile.txt”, write_mode);
  L1: write(buf, string'("This is an example of formatted IO"));
  L2: writeline(outfile, buf);
  L3: write(buf, string'("The first parameter is="));
  L4: write(buf, count);
  L5: write(buf, ' ');
  L6: write(buf, string'("The second parameter is="));
  L7: write(buf, value);
  L8: writeline(outfile, buf);
  L9: write(buf, string'(“... and so on"));
  L10: writeline(outfile, buf);
  L11: file_close(outfile);
  wait;
end process;
end architecture beh;
```

ModelSim Note

- A common error when calling
  - `WRITE (L, "hello");` or
  - `WRITE (L, "010111");`
  - will cause the following error
  - `ERROR: Subprogram "WRITE" is ambiguous.`
- In the TextIO package, the `WRITE` procedure is overloaded for the types `STRING` and `BIT_VECTOR`.
  - **procedure** `WRITE(L:` **inout** `LINE; VALUE:` **in** `BIT_VECTOR; JUSTIFIED:` **in** `SIDE:= RIGHT; FIELD:` **in** `WIDTH := 0);`
  - **procedure** `WRITE(L:` **inout** `LINE; VALUE:` **in** `STRING; JUSTIFIED:` **in** `SIDE:= RIGHT; FIELD: in WIDTH := 0);`

ModelSim Note

- `"hello"` could be interpreted as a string or a bit
- Use the following syntax:

```
WRITE (L, string'("hello"));
WRITE (L, bit_vector'(" 010111 "));
```

Output of The Program

- buf of `LINE` type can be accessed by only `write` and `read` procedures.
- The file "myfile.txt" will contain

```
This is an example of formatted IO
The first parameter is=5 The second parameter is=0110
... and so on
```

- If you write "STD_OUTPUT" instead of "myfile.txt" then the output will be written to simulator console (ModelSim's Main Window)
- Note that "STD_OUTPUT" must be in capital letters.

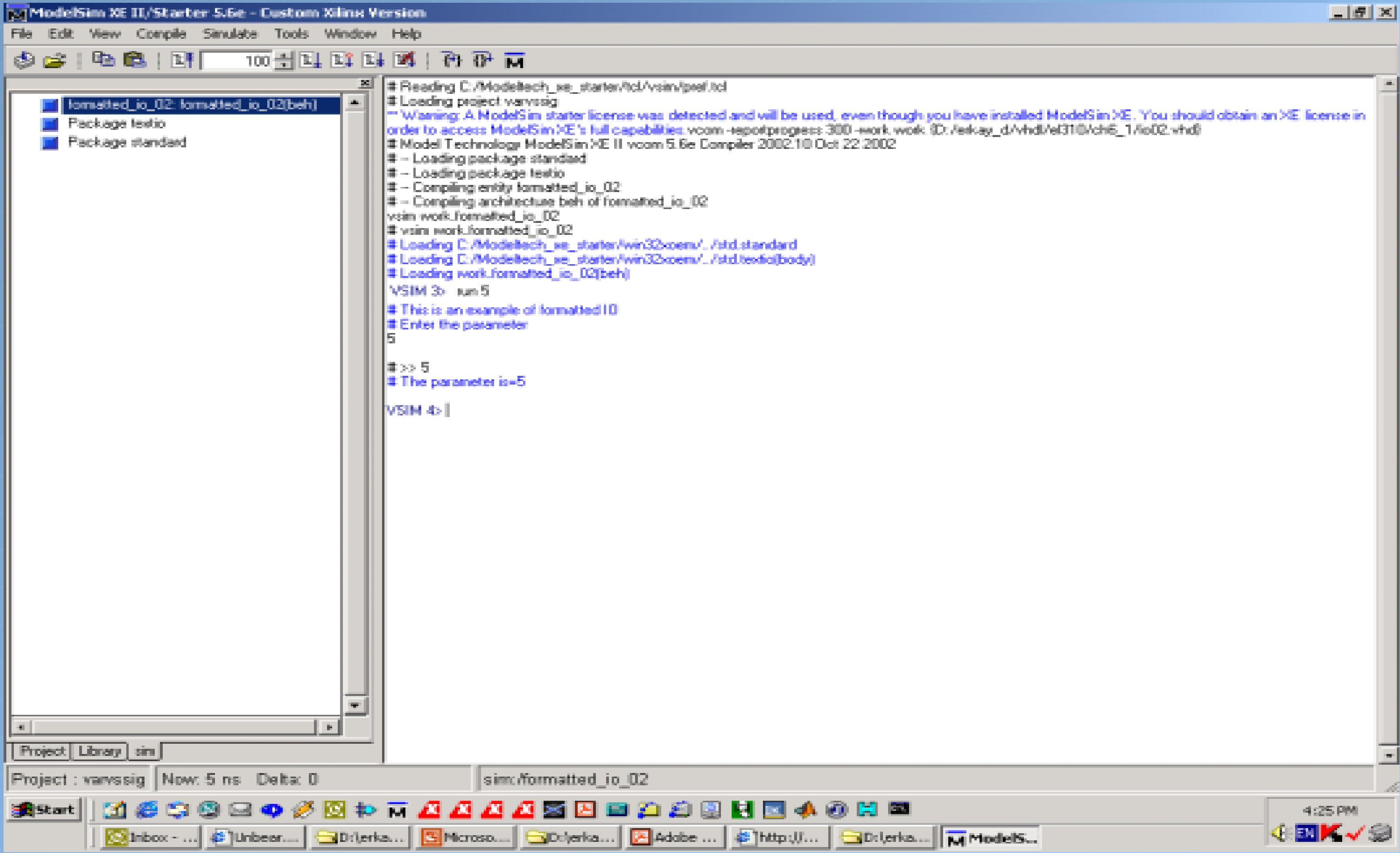
Yet Another Example

```
use STD.textio.all;
entity formatted_io_02 is
end entity;
architecture beh of formatted_io_02 is
begin
  process is
    file infile, outfile: text; -- two files
    variable f_status: FILE_OPEN_STATUS;
    variable buf_in, buf_out: LINE; -- buffers between the
                                    -- program and file
    variable count: integer;
  begin
    file_open(f_status, infile, "STD_INPUT", read_mode);
    file_open(f_status, outfile, "STD_OUTPUT", write_mode);
    ...
end process;
end architecture beh;
```

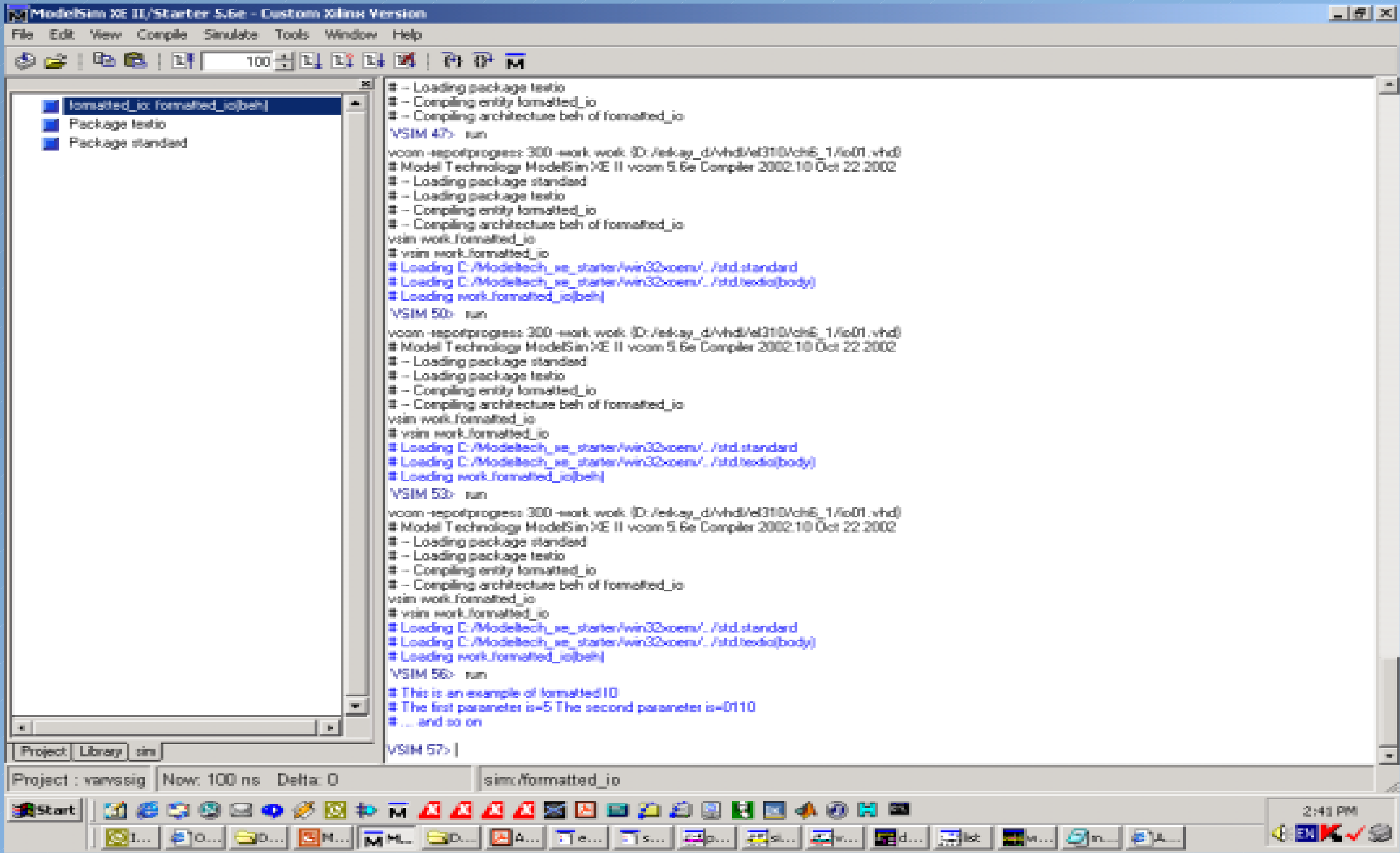
Yet Another Example

```
...
process is
  ...
  begin
    L1: write(buf_out, string'("This is an example of
                                          formatted IO"));
    L2: writeline(outfile, buf_out);
    L3: write(buf_out, string'("Enter the parameter"));
    L4: writeline(outfile, buf_out);
    L5: readline(infile, buf_in);
    L6: read(buf_in, count);
    L7: write(buf_out, string'("The parameter is="));
    L8: write(buf_out, count);
    L9: writeline(outfile, buf_out);
    L10: file_close(outfile);
    wait;
  end process;
end architecture beh;
```

Yet Another Example



Use of STD_OUTPUT



STD_INPUT and STD_OUTPUT

```
use STD.textio.all;
entity formatted_io_02 is
end entity;
architecture beh of formatted_io_02 is
begin
  process is
    variable count: integer;
    variable buf_in, buf_out: LINE; -- buffers between the
                                    -- program and file
  begin
    ...
end process;
end architecture beh;
```

STD_INPUT and STD_OUTPUT

```
  variable count: integer;
  variable buf_out, buf_in: LINE;
  begin
    L1: write(buf_out, string'("This is an example of
                                   formatted IO"));
    L2: writeline(output, buf_out);
    L3: write(buf_out, string'("Enter the parameter"));
    L4: writeline(output, buf_out);
    L5: readline(input, buf_in);
    L6: read(buf_in, count);
    L7: write(buf_out, string'("The parameter is="));
    L8: write(buf_out, count);
    L9: writeline(output, buf_out);
    wait;
  end process;
end architecture beh;
```

Dealing with `std_logic_vector`

```
library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;

package classio is
  procedure read_v1d(file f: text; v: out std_logic_vector);
  procedure write_v1d(file f: text; v: in std_logic_vector);
end package classio;

package body classio is
...
end package body classio;
```

Aim: to create a package that contains procedures to handle reading and writing of `std_logic_vector` type objects in a human-readable format.

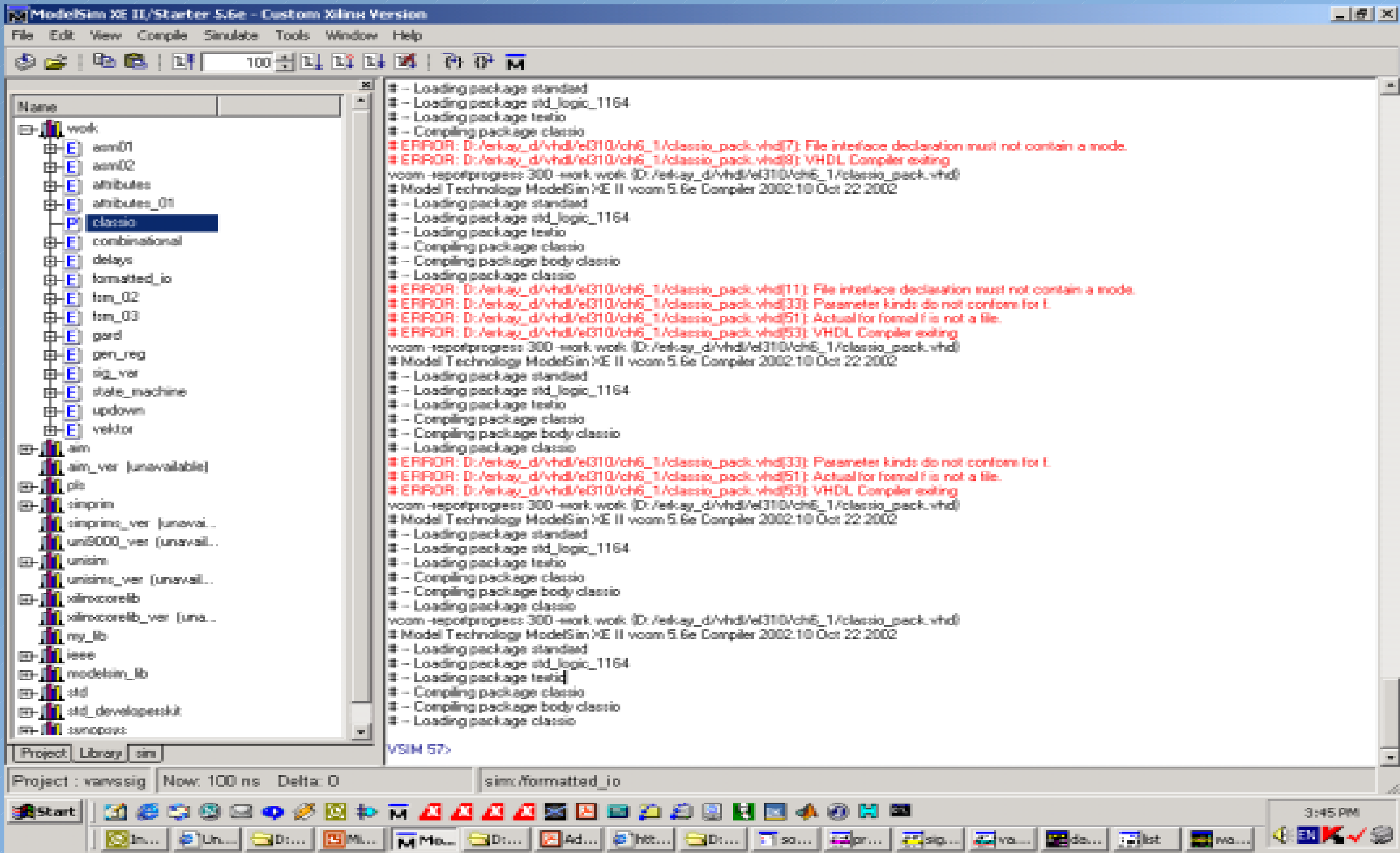
Classio Package

```
package body classio is
  procedure read_v1d(file f: text; v: out std_logic_vector) is
    variable buf: line; variable c: character;
  begin
    readline(f, buf); -- complete line is read into buffer buf
    for i in v'range loop
      read(buf, c);
      case c is
        when 'X' => v(i) := 'X';
        when 'U' => v(i) := 'U';
        when 'Z' => v(i) := 'Z';
        when '0' => v(i) := '0';
        when '1' => v(i) := '1';
        when '-' => v(i) := '-';
        when 'W' => v(i) := 'W';
        when 'H' => v(i) := 'H';
        when 'L' => v(i) := 'L';
        when others => v(i) := '0';
      end case;
    end loop;
  end procedure read_v1d;
...
end package body classio;
```

Classio Package

```
package body classio is
...
  procedure write_v1d(file f: text; v: in std_logic_vector) is
    variable buf: line; variable c: character;
  begin
    for i in v'range loop
      case v(i) is
        when 'X' => write(buf, 'X');
        when 'U' => write(buf, 'U');
        when 'Z' => write(buf, 'Z');
        when '0' => write(buf, character'('0'));
        when '1' => write(buf, character'('1'));
        when '-' => write(buf, '-');
        when 'W' => write(buf, 'W');
        when 'H' => write(buf, 'H');
        when 'L' => write(buf, 'L');
        when others => write(buf, character'('0'));
      end case;
    end loop;
    writeline(f, buf);
  end procedure write_v1d;
end package body classio;
```

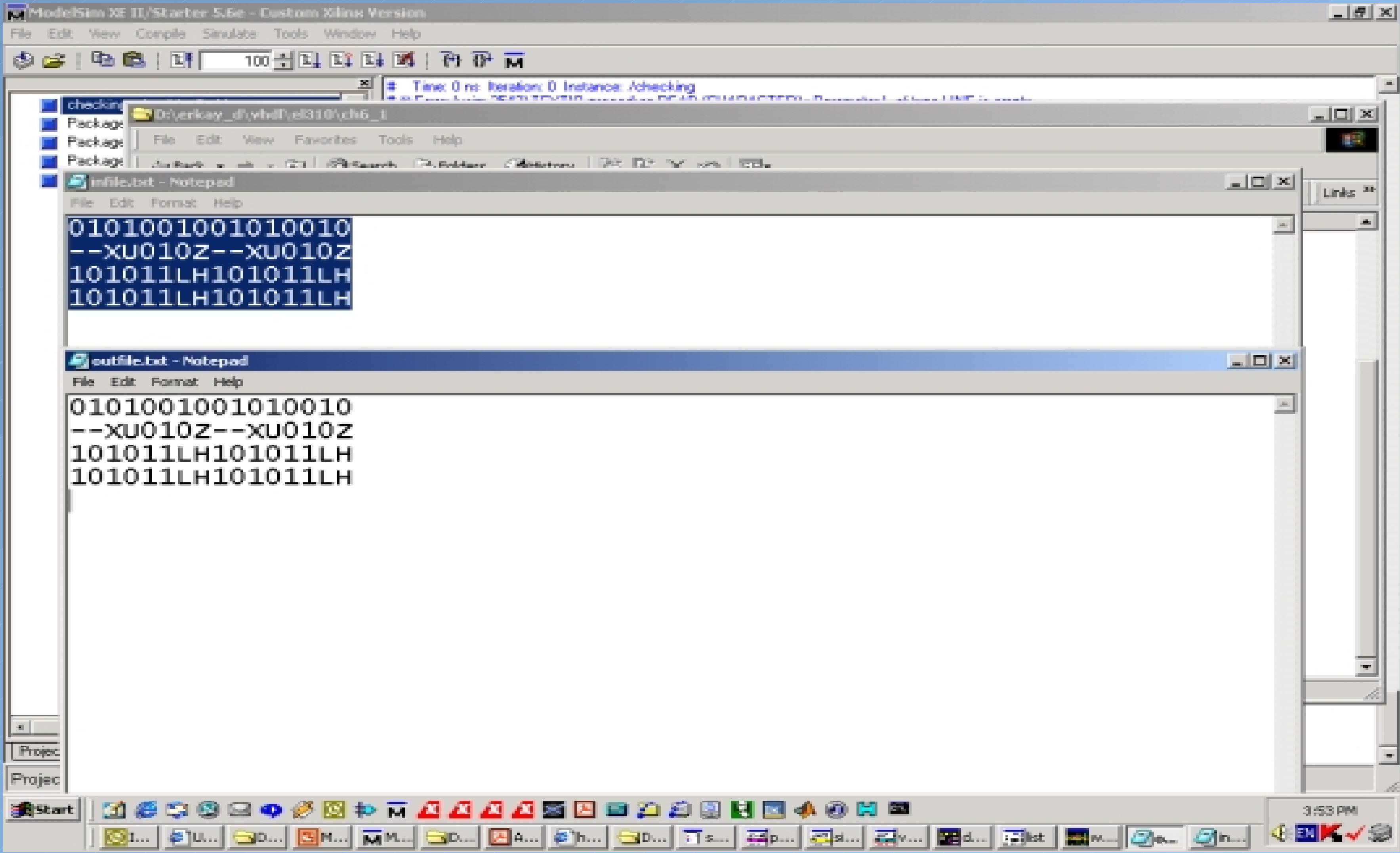
Classio Package



Using the Classio Package

```
library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;
use WORK.classio.all; -- the package classio has been compiled in
                      -- the working directory
entity checking is
end entity;
architecture beh of checking is
begin
  process is
    file infile: TEXT open read_mode is “infile.txt”;
    file outfile: TEXT open write_mode is “outfile.txt”;
    variable check: std_logic_vector(15 downto 0) := x”0008”;
  begin
    while not (endfile(infile)) loop
      read_v1d(infile, check);
      write_v1d(outfile, check);
    end loop;
    file_close(outfile);
    wait;
  end process;
end architecture beh;
```

The Output of the Example



Reading File Names From Console

```
library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;

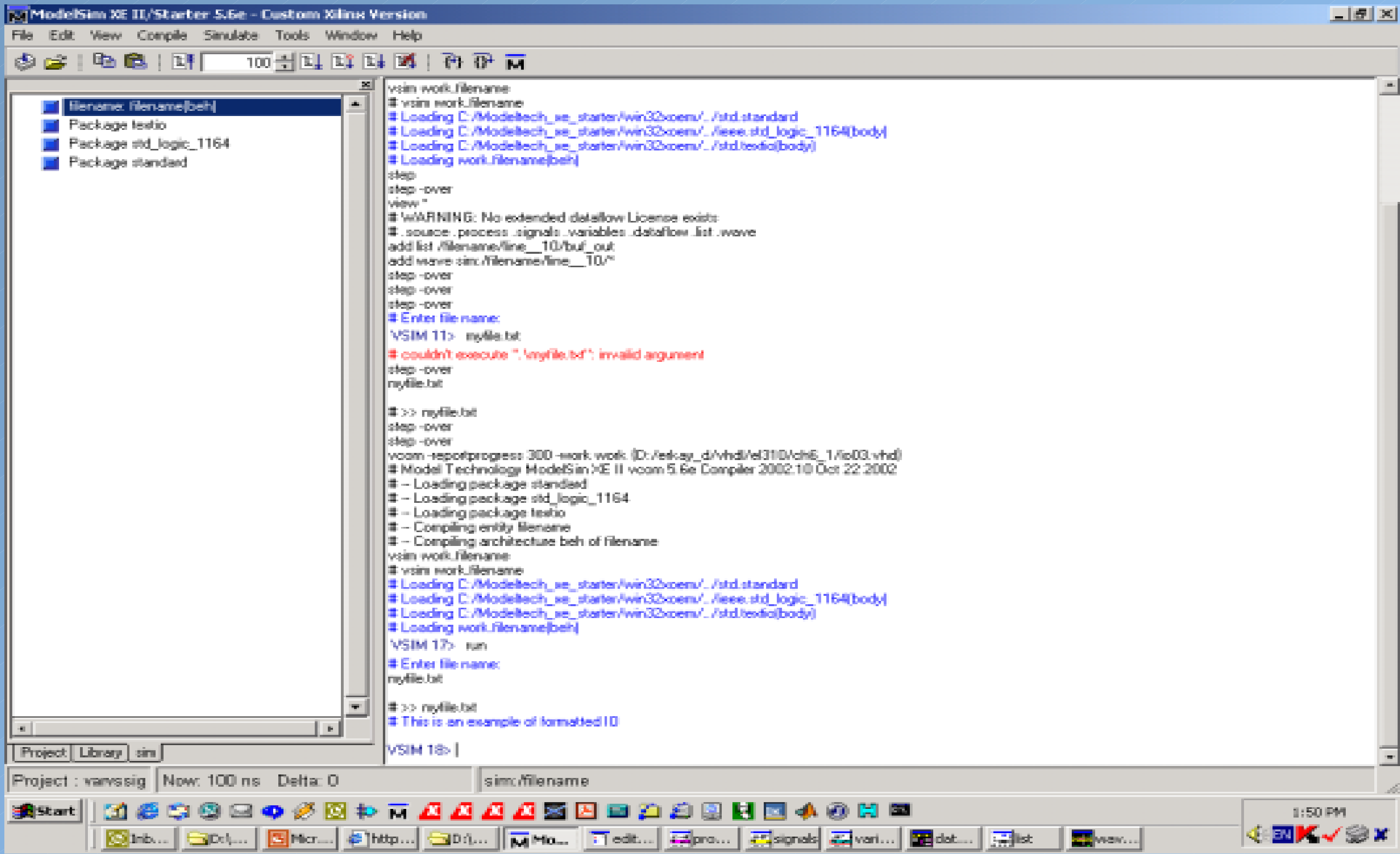
entity filename is
end entity;

architecture beh of filename is
begin
  process is
    file thefile: text;
    variable buf_out, buf_in: LINE;
    variable fname: string(1 to 10);
    variable f_status: FILE_OPEN_STATUS;
  begin ...
```

Reading File Names From Console

```
...
begin
    write(buf_out, string'("Enter file name:"));
    writeline(output, buf_out);
    readline(input, buf_in);
    read(buf_in, fname);
    file_open(f_status, thefile, fname, read_mode);
    readline(thefile, buf_in);
    writeline(output, buf_in);
    wait;
  end process;
end architecture beh;
```

Reading File Names From Console



Testbenches

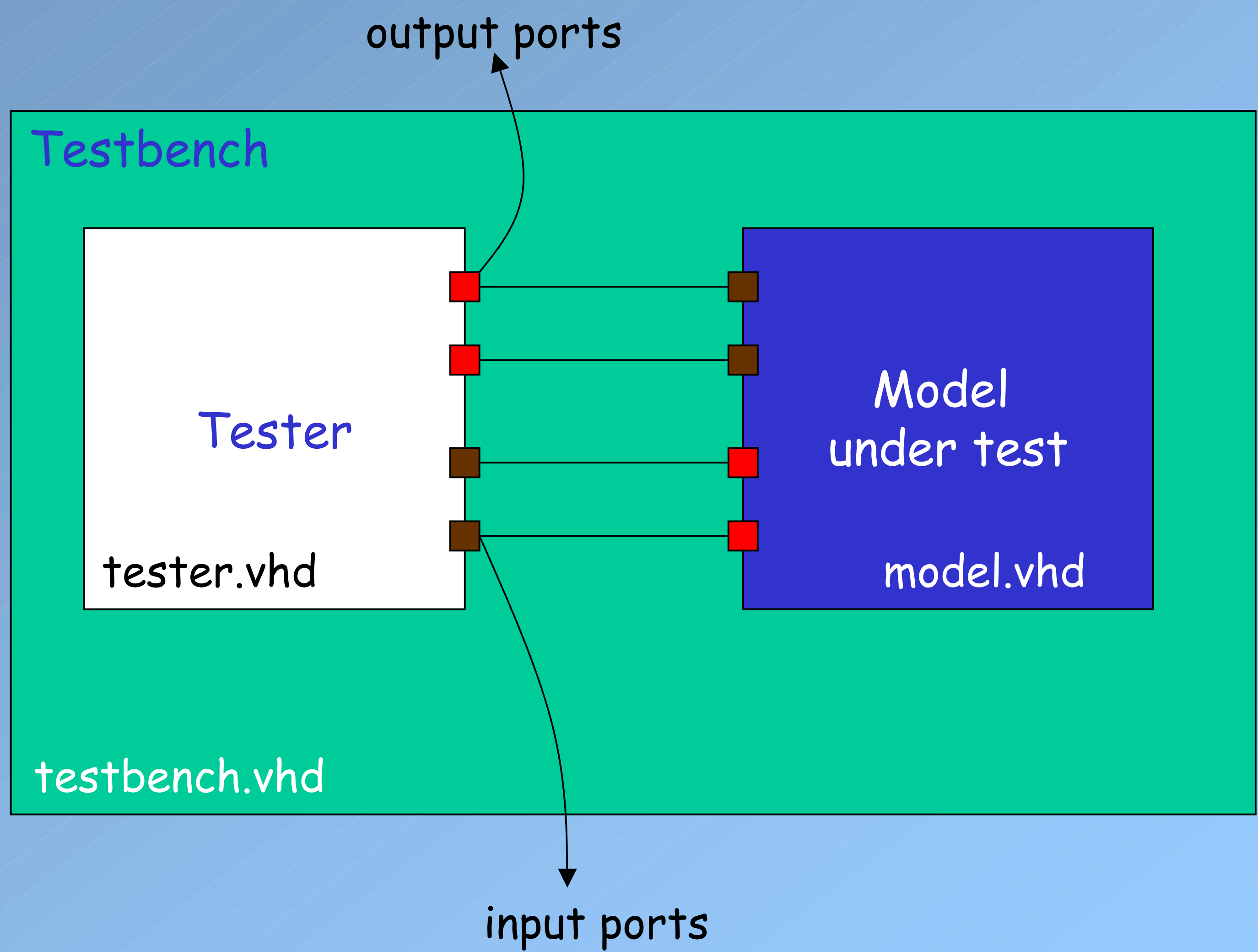
- Basic testing strategy:
  - place the unit in a test frame
  - apply it a set of inputs whose corresponding output values are known
  - observe the outputs of the unit and compare them against the given output values
- Test what?
  1. the model is operating as designed
  2. the design itself is correct.

– Testing during simulation will save a lot of manufacturing efforts.

Testbench Model in VHDL

- Many simulators provide some tools to construct testbenches.
- The VHDL itself is very expressive to construct such testbenches
  - Along with the component under test, the units that are generating the input stimulus and comparing the outputs can also be written in VHDL.
- VHDL Modules in a testbench
  - model.vhd
  - tester.vhd
  - testbench.vhd

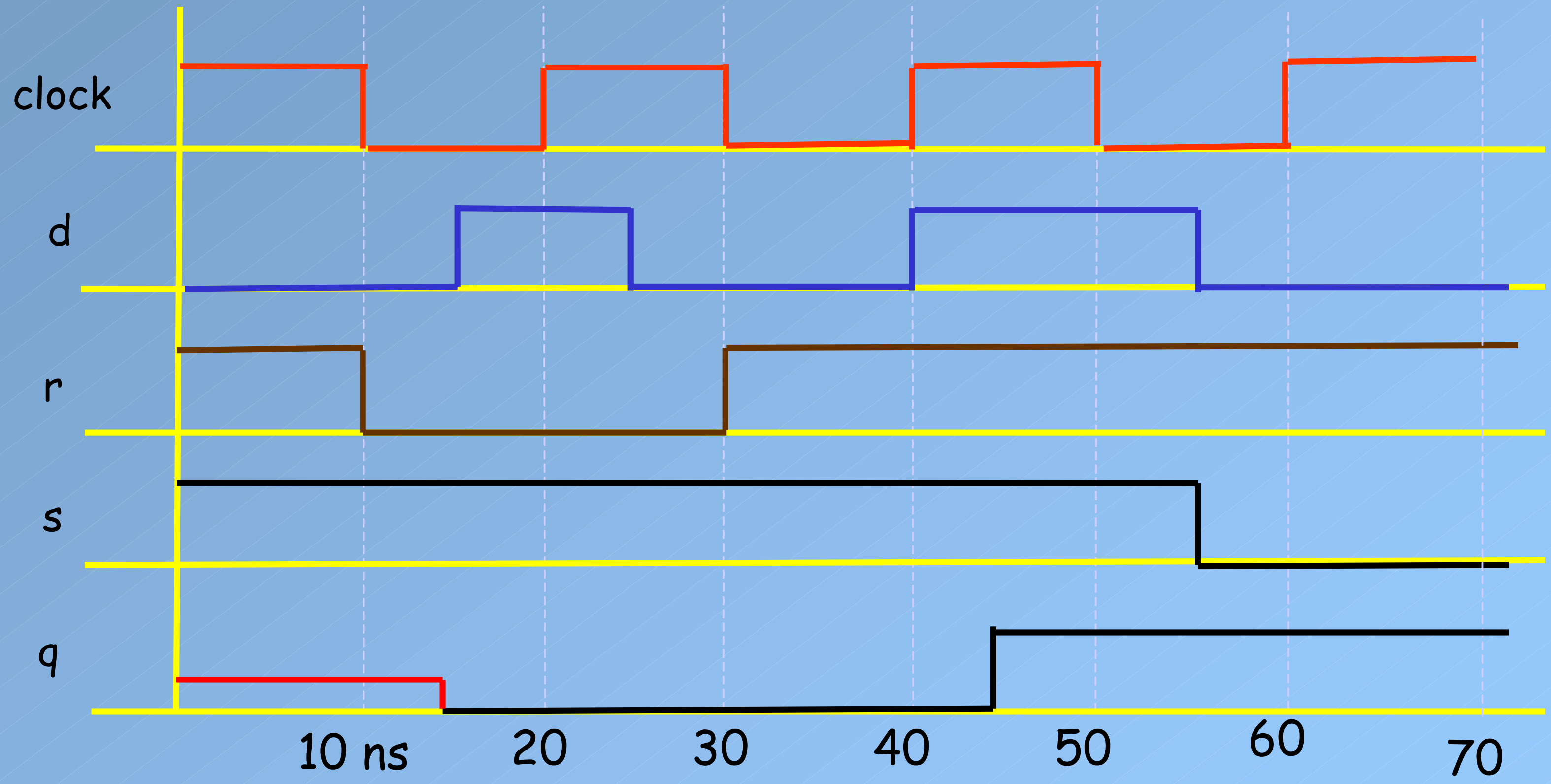
Testbench Model in VHDL



Example: Testing a D Flip-Flop

```
library IEEE;
use IEEE.std_logic_1164.all;
entity asynch_dff is
  port (r, s, d, clock: in std_logic;
        q, qbar: out std_logic);
end entity asynch_dff;
architecture beh of asynch_dff is
begin
  output: process (r, s, clock) is
  begin
    if (r = ‘0’) then
      q <= ‘0’ after 5ns; qbar <= ‘1’ after 5 ns;
    elsif(s = ‘0’) then
      q <= ‘1’ after 5ns; qbar <= ‘0’ after 5 ns;
    elsif(rising_edge(clock)) then
      q <= d after 5ns; qbar <= not d after 5 ns;
    end if;
  end process output;
end architecture beh;
```

Sample Test Pattern on D Flip-Flop



A Simple Tester

```
library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;
use WORK.classio.all;

entity asynch_dff_tester is
  port(r, s, d, clock: out std_logic;
       q, qbar: in std_logic);
end entity asynch_dff_tester;

architecture beh of asynch_dff_tester is
begin
  clock_process: process is
  begin
    clock <= '1', '0' after 10 ns;
    wait for 20 ns;
  end process clock_process;
  ...
```

A Simple Tester

```
...
architecture beh of asynch_dff_tester is
...
  io_process: process is
    file infile: TEXT open read_mode is “infile.txt”;
    file outfile: TEXT open write_mode is “outfile.txt”;
    variable buf: line;
    variable msg: string(1 to 20):=“This vector failed!”;
    variable check: std_logic_vector(4 downto 0);
  begin
  ...
  end process io_process;
end architecture beh;
```

A Simple Tester

```
  begin
    while not (endfile(infile)) loop
      read_v1d(infile, check);
      r <= check(4);
      s <= check(3);
      d <= check(2);
      wait for 20 ns;

      if(q /= check(1) or qbar /= check(0)) then
        write(buf, msg);
        writeline(outfile, buf);
        write_v1d(outfile, check);
      end if;
    end loop;
    file_close(outfile);
    wait;
  end process io_process;
end architecture beh;
```

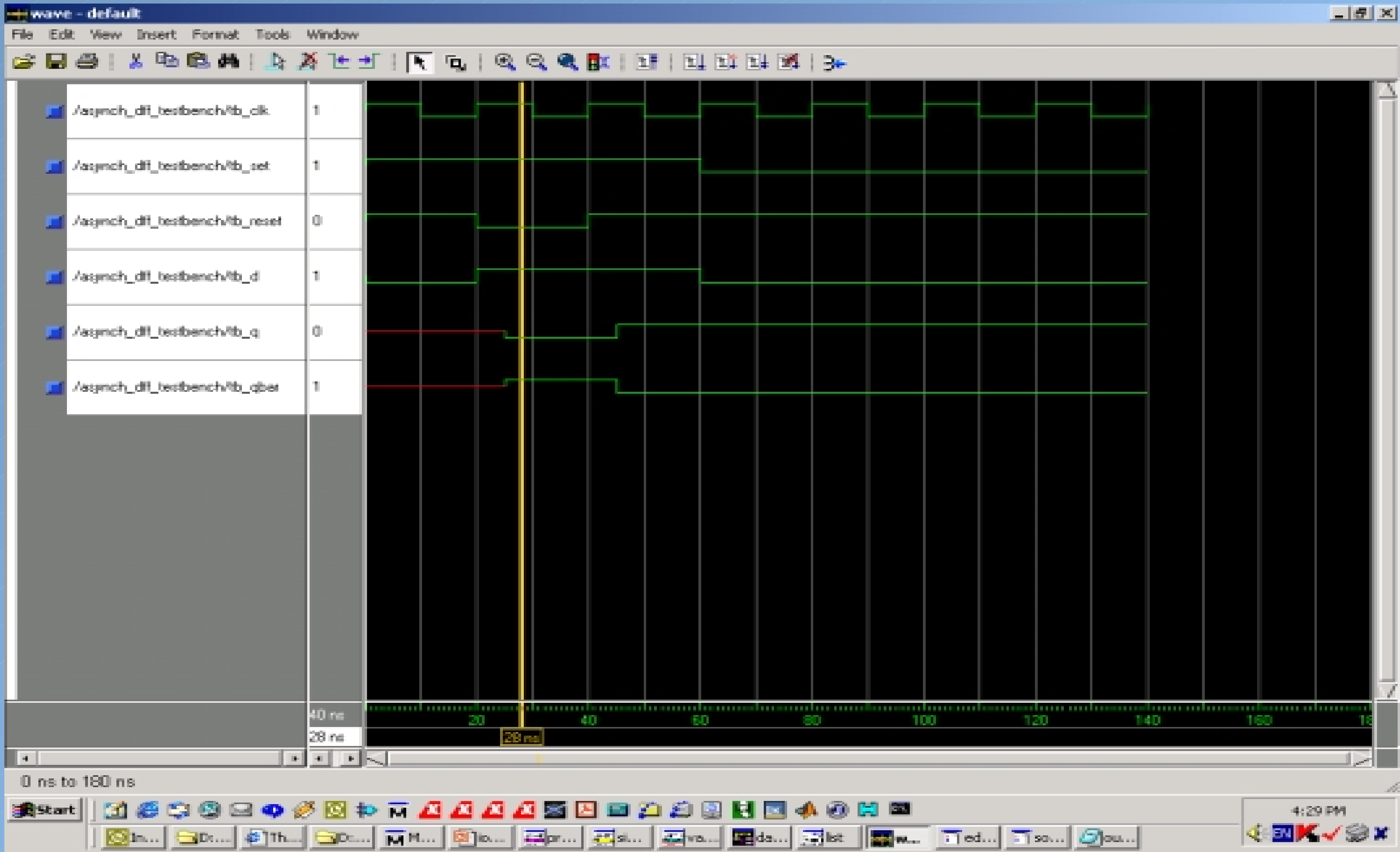
VHDL Model for Testbench

```
use WORK.classio.all;
entity asynch_dff_testbench is end entity;
architecture beh of asynch_dff_testbench is
  component asynch_dff is
    port (r, s, d, clock: in std_logic; q, qbar: out std_logic);
  end component asynch_dff;
  component asynch_dff_tester is
    port(r, s, d, clock: out std_logic; q, qbar: in std_logic);
  end component asynch_dff_tester;
  for T1: asynch_dff_tester use entity WORK.asynch_dff_tester;
  for M1: asynch_dff use entity WORK.asynch_dff(beh);
  signal tb_set, tb_reset, tb_d, tb_q, tb_qbar, tb_clk: std_logic;
begin
  T1: asynch_dff_tester port map (r=>tb_reset, s=>tb_set,
                  d=>tb_d, clock=>tb_clk, q=>tb_q, qbar=>tb_qbar);
  M1: asynch_dff port map(r=>tb_reset, s=>tb_set, d=>tb_d,
                  clock=>tb_clk, q=>tb_q, qbar=>tb_qbar);
end architecture beh;
```

Test Vectors

- infile.txt contains
  - ```
    11001
    01101
    11110
    10010
    10011
    ```
  - `reset set d q qbar`
- Testbench will write into outfile.txt
  - ```
    This vector failed!
    11001
    This vector failed!
    10011
    ```

Simulation Results for Testbench

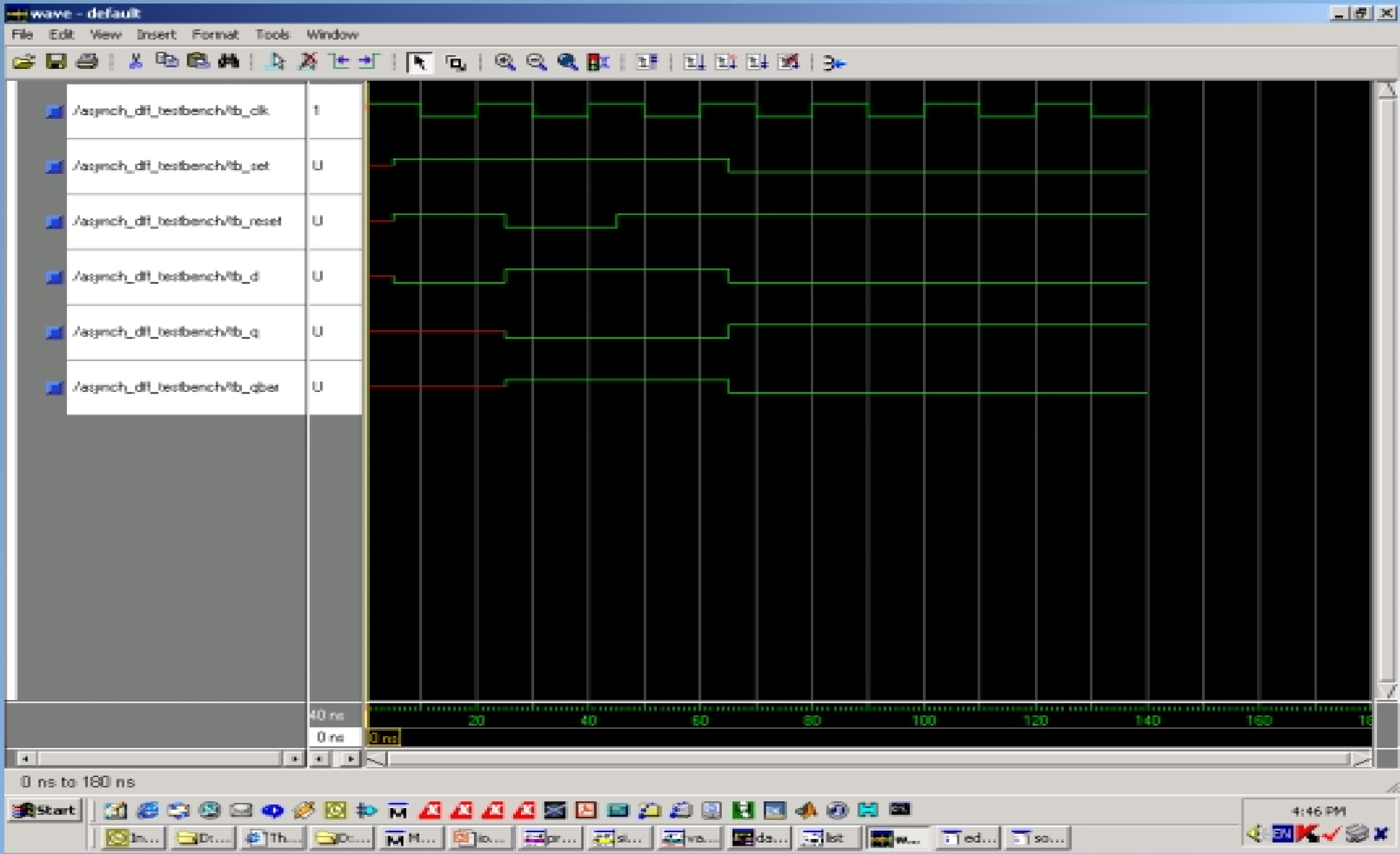


A Better Tester

```
begin
  wait for 5 ns;
  while not (endfile(infile)) loop
      read_v1d(infile, check);
      r <= check(4);
      s <= check(3);
      d <= check(2);
      wait for 20 ns;

      if(q /= check(1) or qbar /= check(0)) then
        write(buf, msg);
        writeline(outfile, buf);
        write_v1d(outfile, check);
      end if;
   end loop;
   file_close(outfile);
   wait;
  end process io_process;
end architecture beh
```

A Better Tester

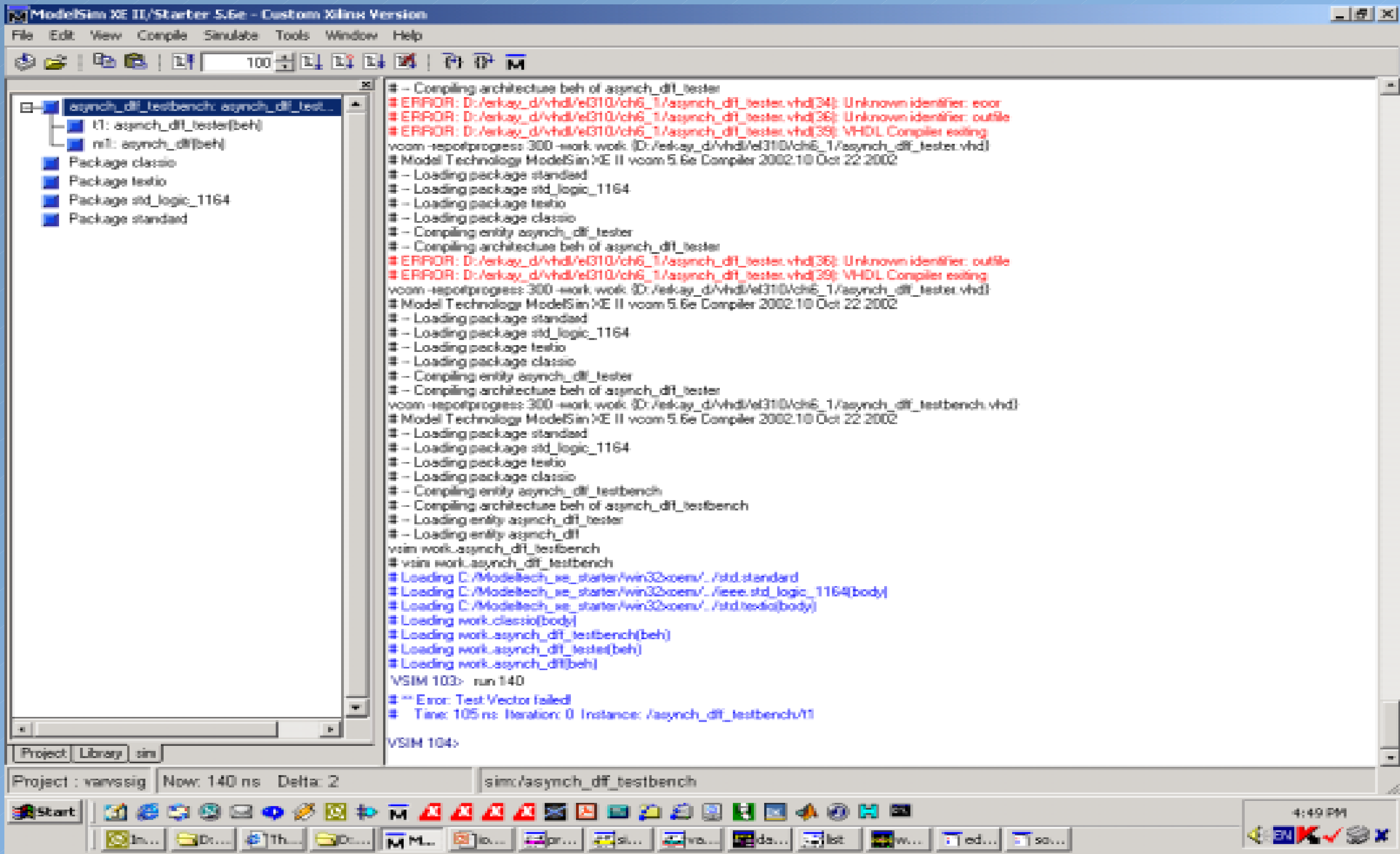


Tester with ASSERT Statement

```
io_process: process is
  file infile: TEXT open read_mode is "infile.txt“;
  variable check: std_logic_vector(4 downto 0);
begin
  wait for 5 ns;
  while not (endfile(infile)) loop
    read_v1d(infile, check);
    r <= check(4);
    s <= check(3);
    d <= check(2);
    wait for 20 ns;

    assert Q = check(1) and Qbar = check(0)
    report "Test Vector failed!“
    severity ERROR;
  end loop;
  wait;
end process io_process;
```

Simulator Console



Testbench Overview

- A testbench is a structural model with two components:
  - model under test
  - tester
- Components in tester module
  - processes to generate waveforms
  - VHDL statements to read test vectors from input files and apply them to the model under test
  - VHDL statements to record the outputs that are produced by the model under test in response to the test vectors

Testbench Template

```
library Lib1;
library Lib2;
use Lib1.package_name.all;
use Lib2.package_name.all;
entity test_bench_name is end entity;
architecture arch_name of test_bench_name is
  component tester_name is
    port (input signals: in type; output signals: out type);
  end component tester_name;
  component model_name is
    port (input signals: in type; output signals: out type);
  end component model_name;
  signal internal signals : type:=initialization;
begin
  T1: tester_name port map(port=>signal1, ...);
  M1: model_name port map(port=>signal1, ...);
end arch_name;
```

Other Ways to Create Testbenches

- Tabular Approach:
  - testbench for 3-bit counter

```
library ieee;
use ieee.std_logic_package.all;

entity counter is
  generic (n: integer);
  port(clock, reset: in std_logic;
       count: inout signed(n-1 downto 0));
end entity;
```

Tabular Approach

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity counter is
  generic (n: integer);
  port(clock, reset: in std_logic;
       count: inout signed(n-1 downto 0));
end entity;

architecture beh of counter is
begin
  counter: process(clock, reset)
  begin
    if reset = '1' then
      count <= (others => '0');
    elsif(rising_edge(clock)) then
      count <= count + 1;
    end if;
  end process;
end beh;
```

Tester for the Counter 1/5

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity counter_tester is
end entity;

architecture beh of counter_tester is
  component counter
    generic(n:integer:=6);
    port(clock, reset: in std_logic;
         count: inout signed(n-1 downto 0));
  end component;
  signal clock, reset: std_logic;
  signal count: signed(2 downto 0);
  type test_vector is record
    clock: std_logic;
    reset: std_logic;
    count: signed(2 downto 0);
  end record;
...
```

Tester for the Counter 2/5

```
type test_vector_array is array(natural range <>) of test_vector;

constant test_vectors: test_vector_array := (
  -- reset the counter
  (clock => '0', reset => '1', count => "000"),
  (clock => '1', reset => '1', count => "000"),
  (clock => '0', reset => '0', count => "000"),
  -- clock the counter several times
  (clock => '1', reset => '0', count => "001"),
  (clock => '0', reset => '0', count => "001"),
  (clock => '1', reset => '0', count => "010"),
  (clock => '0', reset => '0', count => "010"),
  (clock => '1', reset => '0', count => "011"),
  (clock => '0', reset => '0', count => "011"),
  (clock => '1', reset => '0', count => "100"),
  (clock => '0', reset => '0', count => "100"),
  (clock => '1', reset => '0', count => "101"),
  (clock => '0', reset => '0', count => "101"),
```

Tester for the Counter 3/5

```
 (clock => '1', reset => '0', count => "110"),
(clock => '0', reset => '0', count => "110"),
(clock => '1', reset => '0', count => "111"),
(clock => '0', reset => '0', count => "111"),
(clock => '1', reset => '0', count => "000"),
(clock => '0', reset => '0', count => "000"),
(clock => '1', reset => '0', count => "001"),
(clock => '0', reset => '0', count => "001"),
(clock => '1', reset => '0', count => "010"),
-- reset the counter
(clock => '0', reset => '1', count => "000"),
(clock => '1', reset => '1', count => "000"),
(clock => '0', reset => '0', count => "000"),
-- clock the counter several times
(clock => '1', reset => '0', count => "001"),
(clock => '0', reset => '0', count => "001"),
(clock => '1', reset => '0', count => "010"),
(clock => '0', reset => '0', count => "010")
);
```

Tester for the Counter 4/5

```
begin
-- instantiate counter under test
CUT: counter generic map(3)
             port map(clock=>clock, reset=>reset, count=>count);
-- apply test vectors and check results
verify: process is
  variable vector: test_vector;
  variable errors: Boolean:=false;
begin
  for i in test_vectors'range loop
    -- get vector i
    vector := test_vectors(i);

    -- schedule vector i
    clock <= vector.clock;
    reset <= vector.reset;

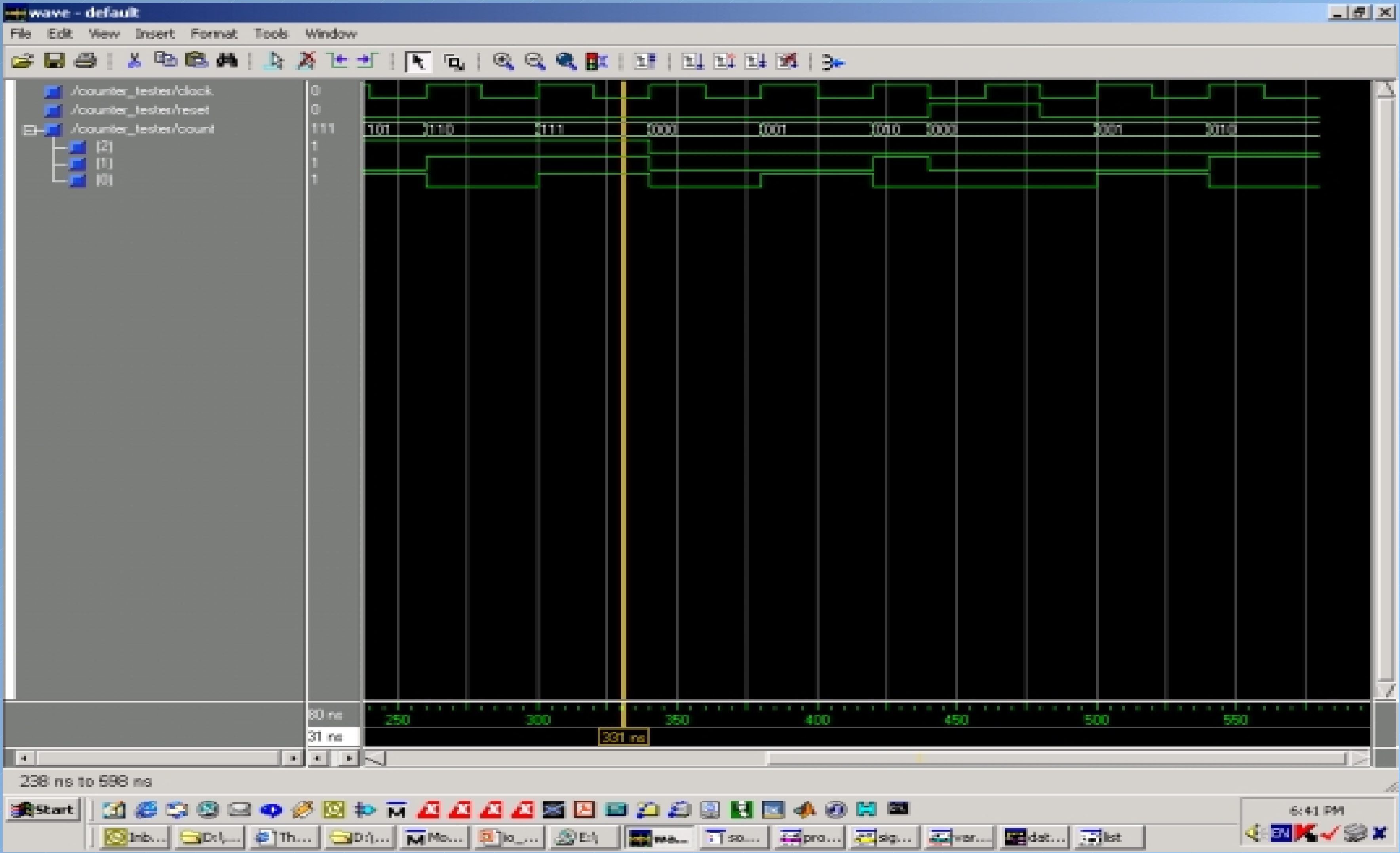
    -- wait for circuit to settle
    wait for 20 ns;
```

Tester for the Counter 5/5

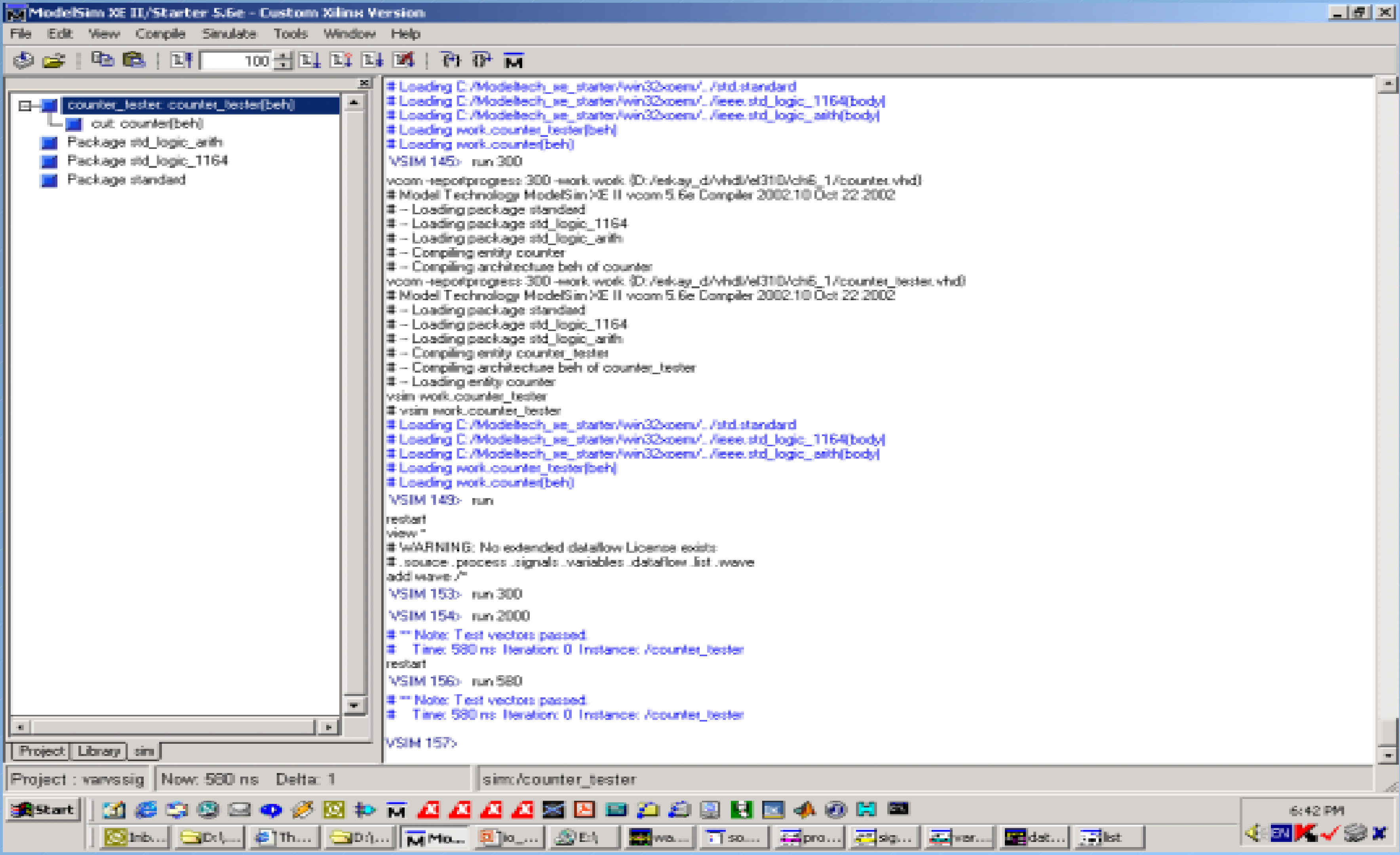
```
      -- check output vectors
      if count /= vector.count then
        assert false
          report "counter mulfunctioning";
          errors := true;
      end if;
    end loop;

    -- assert reports on false
    assert not errors
      report "Test vectors failed."
      severity note;
    assert errors
      report "Test vectors passed."
      severity note;
    wait;
  end process;
end beh;
```

Simulation Results



Simulator Console



Record Data Types

- An object of a record type is composed of elements of the same or different types.
  - It is analogous to `struct` type in C.
- Example:
  - ```
    type pin_type is range 0 to 10;
    type MODULE is
      record
        size           : integer range 20 to 200;
        critical_delay : Time;
        no_inputs      : pin_type;
        no_outputs     : pin_type;
      end record;
    ```
  - ```
    variable nand_comp:module;
    nand_component:=(50, 20 ns, 3, 2);
    ```

Record Data Types

- Examples:
  - **variable** nand_comp: module;
    signal nand_generic: module;
  - nand_generic <= nand_comp;
  - nand_comp.no_inputs := 2;

Array Data Types

- An object of an array type consists of elements of the same type.
- Examples:
  - **type** ADDRESS_WORD **is** **array**(0 **to** 63) **of** bit;
  - **type** DATA_WORD **is** **array**(7 **downto** 0) **of** std_ulogic;
  - **type** ROM **is** **array**(0 **to** 127) **of** data_word;
  - **type** DECODE_MATRIX **is** **array**(positive **range** 15 **downto** 1, natural **range** 0 **to** 3) **of** std_ulogic;
  - **subtype** natural **is** integer **range** 0 **to** integer'high;
  - **subtype** positive **is** integer **range** 1 **to** integer'high;

Array Data Types

- Examples
  - **variable** ROM_ADDR: ROM;
  - **signal** ADDRES_BUS: ADDRESS_WORD;
  - **constant** DECODER: DECODE_MATRIX; -- deferred
  - **variable** DECODE_VALUE: DECODE_MATRIX;

  - ROM_ADDR(5) := “01000101”;
  - DECODE_VALUE := DECODER;
  - ADDRESS_BUS(8 to 15) <= x”ff”;
- Unconstrained Arrays:
  - We specify the number of elements when we declare objects of unconstrained array type.
  - Subtype declaration may also specify the dimension.
  - **type** STACK_TYPE **is array** (integer range<>) **of** ADDRESS_WORD;

Array Data Types

- **type** STACK_TYPE **is array** (integer range<>) **of** ADDRESS_WORD;
- **subtype** stack **is** stack_type(0 **to** 63);
- **type** op_type **is** (add, sub, mul, div);
- **type** timing **is array** (op_type **range** <>, op_type **range** <>) **of** time;

- **variable** FAST_STK: STACK_TYPE(-127 **to** 127);
- **constant** ALU_TIMING: TIMING:=

```
  -- ADD, SUB, MUL
  ((10 ns, 20 ns, 45 ns),    -- ADD
   (20 ns, 15 ns, 40 ns),    -- SUB
   (45 ns, 40 ns, 30 ns));   -- MUL
```

Array Data Types

- STRING and BIT_VECTOR are two predefined one-dimensional unconstrained array types.
- Examples:
  - **variable** message: string(1 **to** 17):= “Hello, VHDL World”;
  - **signal** rx_bus: bit_vector(0 **to** 5) := o”37”;
  - **constant** add_code: bit_vector := (‘0’, ‘1’, ‘1’, ‘1’, ‘0’);
- No unconstrained array of an unconstrained array
  - **type** memory **is array** (natural **range** <>) **of** std_ulogic_vector;
  - **type** reg_file **is array** (natural **range** <>) **of** bit_vector(0 **to** 7);

String Literals

- One-dimensional array of characters is called string literal.
  - "This is a test"
  - "Spike is detected"
  - "State ""ready"" entered!"
- Two types of objects can be assigned with string literals
  - STRING and BIT_VECTOR.
  - Example:

```
variable ERROR_MESSAGE: string(1 to 19);
error_message := "Fatal Error: ABORT!"
variable bus_value: bit_vector(1 to 3);
BUS_VALUE := "1101";
```

String Literals

- The type of a string literal can also be explicitly stated by <u>qualified expression</u>.
  - `WRITE(L, BIT_VECTOR'("1110001"));`
  - `CHARINT'('1', N); -- ambiguous`
  - `CHARINT'(character'('1'), N);`
- Bit String Literals
  - `X"FF0";`
  - `B"00_0011_1101";`
  - `O"327";`
  - `X"FF_F0_AB";`

String Literals: Assigning

- There are different ways to assign values to an array object
  - **variable** op_codes: bit_vector(1 **to** 5);
  - op_codes := "01001";
  - op_codes := ('0', '1', '0', '0', '1');
  - op_codes := (2 => '1', 5 => '1', **others** => '0');
  - opcodes := (**others** => '0');

Type Conversion

- VHDL allows for very restricted type casting:
  - `sum := INTEGER(polywidth * 1.5);`
- Type conversion is allowed between closely related types
  - Between integer and real
  - between array types that have the same dimensions whose index types are closely related and element types are the same
  - Any other type conversion is done through user-defined functions.

Type Conversion Examples

- Examples:
  - **type** signed **is** **array** (natural **range** <>) **of** bit;
  - **type** bit_vector **is** **array**(natural **range** <>) **of** bit;
  - **signal** FCR: signed(0 **to** 7);
  - **signal** EMA: bit_vector(0 **to** 7);
  - FCR <= EMA; -- illegal
  - FCR <= signed(EMA);
  - **signal** real_sig: real;
  - **signal** int_a, int_b: integer;
  - real_sig <= real(int_a)/real(int_b);

Access Types

- Pointers to dynamically allocated object of some other type:
  - ```
    type pin_type is range 0 to 10;
    type MODULE is
      record
        size           : integer range 20 to 200;
        critical_delay : Time;
        no_inputs      : pin_type;
        no_outputs     : pin_type;
      end record;
    ```
  - `type PTR is access MODULE;`
- When they do not point to an object, they have value of **null**.
  - `variable  mod1ptr, mod2ptr: ptr; -- default value is null;`
  - `mod1ptr := new MODULE;`
    ```

Access Types

- The objects can be created dynamically
  - **variable** mod1ptr, mod2ptr: ptr; -- default value is **null**;
  - mod1ptr := **new** MODULE;
  - mod1ptr := **new** MODULE’(25, 10 ns, 4, 9);
- Access & misc.
  - mod1ptr.size, mod1ptr.critical_dly, etc.
  - deallocate(mod1ptr);
  - mod1ptr := mod2ptr;
  - **type** bitvec_ptr **is access** bit_vector;
  - **variable** bitvec1: bitvec_ptr := **new** bit_vector(“1001”);
  - mod2ptr := **new** MODULE’(critical_dly=>10 ns, no_inputs=>2, no_outputs=>3, size=>100);

Operators

- Logical operators
  - **not**
  - associative: **and, or, xor, xnor,**
  - non-associative: **nand, nor**
  - Example: `A` **nand** `B` **nand** `C; -- illegal`
- Relational
  - `=, /=, <, <=, >, >=`
  - `bit_vector('0', '1', '1') < bit_vector(('1', '0', '1')` is true
  - **type** `MVL` **is** `('U', '0', '1', 'Z')`
    `MVL'('U') < MVL'('Z')` is true
  - `"VHDL" < "VHDL92"` is true

Operators

- Shift Operators
  - `sll, srl, sla, sra, rol, ror`
  - Examples:
  - “1001010” sll 2 is “0101000”
  - “1001010” srl 3 is “0001001”
  - “1001010” sla 2 is “0101000”
  - “1001010” sra 3 is “1111001”
  - “1001010” rol 2 is “0101010”
  - “1001010” ror 3 is “0101001”
  - “1001010” ror -5 is “0101010”
  - “1001010” rol -4 is “0101001”

Operators

- Arithmetic
  - +, -, &
  - '0' & '1' is "01"
  - 'C' & 'A' & 'T' is "CAT
  - *, /, mod, rem
  - A **rem** B = A – (A/B)*B
  - A **mod** B = A – B*N -- for some integer N
  - abs, **

Aliases

- An alias declares an alternate names for all or part of a named item.
  - It provides a convenient shorthand for items that have long names
  - Example:

```
signal S: bit_vector (31 downto 0);
```

31 30 22 0

sign	exponent	mantissa

31 26 23 18 0

opcode	reg	base	offset

Aliases

- Examples:
  - **signal** S: bit_vector (31 **downto** 0);
  - **alias** sign: bit **is** S(31);
  - **alias** exponent: bit_vector(7 **downto** 0) **is** S(30 **downto** 23);
  - **alias** mantissa: bit_vector(22 **downto** 0) **is** S(22 **downto** 0);
  - **alias** opcode: bit_vector(0 **to** 4) **is** S(31 **downto** 27);
  - **alias** reg: bit_vector(2 **downto** 0) **is** S(26 **downto** 24);
  - **alias** base: bit_vector(4 **downto** 0) **is** S(23 **downto** 19);
  - **alias** offset: bit_vector(18 **downto** 0) **is** S(18 **downto** 0);