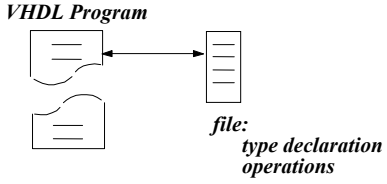
Basic Input and Output

File Objects

- VHDL objects
  - signals
  - variables
  - constants
  - *Files*
- The file type permits us to declare and use file objects



File Declarations

- Files can be distinguished by the type of information stored
  type text is file **of string**;
  type IntegerFileType **is file of integer**;

- File declarations VHDL 1987
  - **file** infile: text **is in** “inputdata.txt”;
  - **file** outfile: text **is out** “outputdata.txt”;

- File declarations VHDL 1993
  - **file** infile: text **open** read_mode **is** “inputdata.txt”;
  - **file** outfile: text **open** write_mode **is** “outputdata.txt”;

Binary File I/O (VHDL 1993)

```
entity io93 is -- this entity is empty
end entity io93;
architecture behavioral of io93 is
begin
process is
type IntegerFileType is file of integer; --
file declarations
file dataout :IntegerFileType;
variable count : integer:= 0;
variable fstatus: FILE_OPEN_STATUS;
begin
file_open(fstatus, dataout,"myfile.txt",
write_mode); -- open the file
for j in 1 to 8 loop
write(dataout,count); -- some random
values to write to the file
count := count+2;
end loop;
wait; -- an artificial way to stop the process
end process;
end architecture behavioral;
```

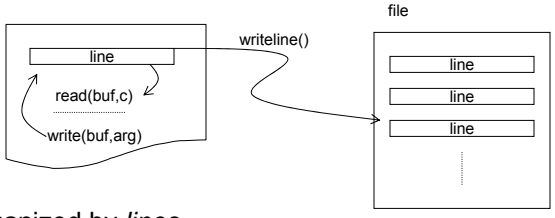
- VHDL provides read(f,value), write(f, value) and endfile(f)
- VHDL 93 also provides File_Open() and File_Close()
- Explicit vs. implicit file open operations

Binary File I/O (VHDL 1987)

```
--
-- test of binary file I/O
--
entity io87_write_test is
end io87_write_test;
architecture behavioral of io87_write_test is
begin
process
type IntegerFileType is file of integer;
file dataout :IntegerFileType is out
"output.txt";
variable check :integer :=0;
begin
for count in 1 to 10 loop
check := check +1;
write(dataout, check);
end loop;
wait;
end process;
end behavioral;
```

- VHDL 1987 provides read(f,value), write(f, value) and endfile(f)
- Implicit file open operations via file declarations

The TEXTIO Package



- A file is organized by *lines*
- read() and write() procedures operate on line data structures
- readline() and writeline() procedures transfer data from-to files
- Text based I/O
- All procedures encapsulated in the TEXTIO package in the library STD
  - Procedures for reading and writing the pre-defined types from lines
  - Pre-defined access to *std_input* and *std_output*
  - Overloaded procedure names

Example: Use of the TEXTIO Package

```
use STD.Textio.all;
entity formatted_io is -- this entity is empty
end formatted_io;
architecture behavioral of formatted_io is
begin
process is
file outfile :text; -- declare the file to be a text file
variable fstatus :File_open_status;
variable count: integer := 5;
variable value : bit_vector(3 downto 0):= X"6";
variable buf: line; -- buffer to file
begin
file_open(fstatus, outfile,"myfile.txt",
write_mode); -- open the file for writing
L1: write(buf, "This is an example of
formatted I/O");
L2: writeline(outfile, buf); -- write buffer to
file
L3: write(buf, "The First Parameter is =");
L4: write(buf, count);
L5: write(buf, ' ');
L6: write(buf, "The Second Parameter is = ");
L7: write(buf, value);
L8: writeline(outfile, buf);
L9: write(buf, "...and so on");
L10: writeline(outfile, buf);
L11: file_close(outfile); -- flush the buffer to
the file
wait;
end process;
end architecture behavioral;
```

Result → This is an example of formatted IO
The First Parameter is = 5 The Second Parameter is = 0110
...and so on

Extending TEXTIO for Other Datatypes

- Hide the ASCII format of TEXTIO from the user
- Create type conversion procedures for reading and writing desired datatypes, e.g., std_logic_vector
- Encapsulate procedures in a package
- Install package in a library and make its contents visible via the **use** clause

Example: Type Conversion

```
procedure write_v1d (variable f: out
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 'L' => write(buf, 'L');
when 'H' => write(buf, 'H');
when others => write(buf, character'('0'));
end case;
end loop;
writeline (f, buf);
end procedure write_v1d;
```

- Text based type conversion for user defined types
- Note: writing values vs. ASCII codes

Example: Type Conversion

```
procedure read_v1d (variable f:in text;
    v : out std_logic_vector) is
variable buf: line;
variable c : character;

begin
readline(f, 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 'L' => v (i) := 'L';
 when 'H' => v (i) := 'H';
 when others => v (i) := '0';
end case;
end loop;
end procedure read_v1d
```

- read() is a symmetric process

Useful Code Blocks (from Bhasker95)

- Formatting the output

```
write (buf, “This is the header”);
writeline (outfile,buf);
write (buf, “Clk =”);
write (buf, clk);
write (buf, “, N1 =”);
write (buf, N1);
```

- Text output will appear as follows

```
This is the header
Clk = 0, N1 = 01001011
```


Useful Code Blocks (Bhaskar95)

- Reading formatted input lines

```
# this file is parsed to separate comments
 0001 65 00Z111Z0
 0101 43 0110X001
```

bit vector integer std_logic_vector

- The code block to read such files may be

```
while not (endfile(vectors) loop
readline(vectors, buf);
if buf(1) = ‘#’ then
   continue;
end if;
read(buf, N1);
read (buf, N2);
read (buf, std_str);
```

convert to std_logic_vector

Useful Code Blocks: Filenames

```
process is
variable buf : line;
variable fname : string(1 to 10);
begin
--
-- prompt and read filename from standard input
--
write(output, "Enter Filename: ");
readline(input,buf);
read(buf, fname);
--
-- process code
--
end process;
```

- Assuming "input" is mapped to simulator console
  - Generally "input" and "output" are mapped to standard input and standard output respectively

Useful Code Blocks: Testing Models

```
library IEEE;

use IEEE.std_logic_1164.all;
use STD.textio.all;
use Work.classio.all;
-- the package classio has been compiled into the working directory

entity checking is
end checking; -- the entity is an empty entity

architecture behavioral of checking is
begin

-- use file I/O to read test vectors and write test results

end architecture behavioral;
```

my I/O library

Testing process

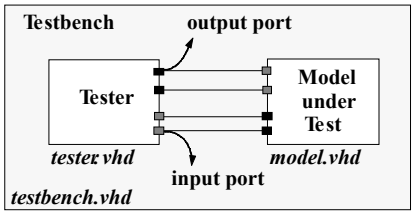
Useful Code Blocks: Testing Models (cont.)

```
process is
-- use implicit file open
--
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
-- copy the input file contents to the output file
while not (endfile (infile)) loop
read_v1d (infile, check);
--
--
write_v1d (outfile, check);
end loop;
file_close(outfile); -- flush buffers to output file
wait; -- artificial wait for this example
end process;
end architecture behavioral;
```

Can have a model here to test

Example: Usually will not have this in your models

Testbenches



- Testbenches are transportable
- General approach: apply stimulus vectors and measure and record response vectors
- Application of predicates establish correct operation of the model under test

Example

```
library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;
use WORK.classio.all; -- declare the I/O package
entity srtester is -- this is the module generating the tests
port (R, S, D, Clk : out std_logic;
Q, Qbar : in std_logic);
end entity srtester;
architecture behavioral of srtester is
begin
clk_process: process -- generates the clock waveform with
begin -- period of 20 ns
Clk<= '1', '0' after 10 ns, '1' after 20 ns, '0' after 30 ns;
wait for 40 ns;
end process clk_process;
```

•Tester module to generate periodic signals and apply test vectors

Example (cont.)

```
Example (cont.)
io_process: process                    -- this process performs the test
file infile : TEXT is in "infile.txt"; -- functions
file outfile : TEXT is out "outfile.txt";
variable buf : line;
variable msg : string(1 to 19) := "This vector failed!";
variable check : std_logic_vector (4 downto 0);
begin
while not (endfile (infile)) loop -- loop through all test vectors in
read_v1d (infile, check);         -- the file
-- make assignments here
wait for 20 ns;                   -- wait for outputs to be available after applying
if (Q /= check (1) or (Qbar /= check(0))) then -- error check
write (buf, msg);
writeline (outfile, buf);
write_v1d (outfile, check);
end if;
end loop;
wait; -- this wait statement is important to allow the simulation to halt!
end process io_process;
end architectural behavioral;
```

Structuring Testers

```
library IEEE;
use IEEE.std_logic_1164.all;
use WORK.classio.all; -- declare the I/O package
entity srbench is
end srbench;
architecture behavioral of srbench is
--
-- include component declarations here
--
-- configuration specification
--
for T1:srtester use entity WORK.srtester (behavioral);
for M1: asynch_dff use entity WORK.asynch_dff (behavioral);
signal s_r, s_s, s_d, s_q, s_qb, s_clk : std_logic;
begin
T1: srtester port map (R=>s_r, S=>s_s, D=>s_d, Q=>s_q, Qbar=>s_qb, Clk =>
s_clk);
M1: asynch_dff port map (R=>s_r, S=>s_s, D=>s_d, Q=>s_q, Qbar=>s_qb, Clk
=> s_clk);
end behavioral;
```


Stimulus Generation

- Stimulus vectors as well as reference vectors for checking
- Stimulus source
- “on the fly” generation
  - Local constant arrays
  - File I/O
- Clock and reset generation
  - Generally kept separate from stimulus vectors
  - Procedural stimulus

Stimulus Generation: Example (Smith96)

```
process
begin
databus <= (others => '0');
for N in 0 to 65536 loop
databus <= to_unsigned(N,16) xor
shift_right(to_unsigned(N,16),1);
for M in 1 to 7 loop
wait until rising_edge(clock);
end loop;
wait until falling_edge(Clock);
end loop;
--
-- rest of the the test program
--
end process;
```

- Test generation vs. File I/O: how many vectors would be need?

Stimulus Generation: Example (Smith96)

```
while not endfile(vectors) loop
readline(vectors, vectorline); -- file format is 1011011
if (vectorline(1) = '#' then
next;
end if;
read(vectorline, datavar);
read((vectorline, A); -- A, B, and C are two bit vectors
read((vectorline, B); -- of type std_logic
read((vectorline, C);
--
--signal assignments
Indata <= to_stdlogic(datavar);
A_in <= unsigned(to_stdlogicvector(A)); -- A_in, B_in and C_in are of
B_in <= unsigned(to_stdlogicvector(B)); -- unsigned vectors
C_in <= unsigned(to_stdlogicvector(C));
wait for ClockPeriod;
end loop;
```

Validation

- Compare reference vectors with response vectors and record errors in external files
- In addition to failed tests record simulation time
- May record additional simulation state

The "ASSERT" Statement

```
assert Q = check(1) and Qbar = check(0)
report "Test Vector Failed"
severity error;
```

Example of Simulator Console Output

```
Selected Top-Level: srbench (behavioral)
: ERROR : Test Vector Failed
: Time: 20 ns, Iteration: 0, Instance: /T1.
: ERROR : Test Vector Failed
: Time: 100 ns, Iteration: 0, Instance: /T1.
```

- Designer can report errors at predefined levels: NOTE, WARNING, ERROR and FAILURE (enumerated type)
- Report argument is a character string written to simulation output
- Actions are simulator specific
- Concurrent vs. sequential assertion statements
- TEXTIO may be faster than ASSERT if we are not stopping the simulation

Example: (Bhaskar 95)

```
architecture check_times of DFF is
constant hold_time: time:=5 ns;
constant setup_time : time:= 2 ns;
begin
process
variable lastevent: time;
begin
if d'event then
assert NOW = 0 ns or (NOW - lastevent) >=
hold_time
report "Hold time too short"
severity FAILURE;
lastevent := NOW;
end if;
-- check setup time
-- D flip flop behavioral model
end process;
end architecture check_times
```

- Report statements may be used in isolation

Summary

- Basic input/output
  - ASCII I/O and the TEXTIO package
  - binary I/O
  - VHDL 87 vs. VHDL 93
- Testbenches
- The ASSERT statement