File Types

※ Special type that contains sequential data.
※ Useful for writing test benches.
※ I/O functions are in the TEXTIO package within the STD library (also see pack1076).
※ In VHDL'87, files are opened when declared, and they cannot be closed.
※ VHDL'93 allows files to be opened and closed as needed.

I/O Functions in TEXTIO for VHDL'87

type TEXT is file of string;
file infile:text is in "input.txt";
file outfile:text is out "output.txt";
procedure READ(file f:TEXT; value:out type);
procedure WRITE(file f:TEXT; value:in type);
procedure ENDFILE(file f:TEXT)
return boolean;

I/O Functions in TEXTIO for VHDL'93

file infile:text open read_mode is
input.txt”;
file outfile:text open write_mode is
“output.txt”;
READ, WRITE, and ENDFILE the same.
procedure FILE_OPEN(file f:TEXT,
    External_Name in STRING;
    Open_Kind:in FILE_OPEN_KIND:=READ_MODE);
procedure CLOSE_FILE(file f:TEXT);
Classio Package

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

package classio is

procedure read_vld(variable f in text;
    v out std_logic_vector);

procedure write_vld(variable f out text;
    v in std_logic_vector);

package body classio is

procedure read_vld(
variable f in text;
    v out std_logic_vector);

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 others => v(i):='0';
    end case;
end loop;
end;

end;

Classio Package (write_vld)

procedure write_vld(
variable f out text;
    v in std_logic_vector);

variable buf:line;
variable c:character;
begin
for i in v'range loop
    case v(i) is
        when 'X' => write(buf,'X');
        ...
    end case;
end loop;
write(buf,character('0'));
end;

Using the Classio Package

library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;
use work.classio.all;
entity checking is
end checking;

architecture behavioral of checking is
begin
process
file infile:TEXT is in "infile.txt";
file outfile:TEXT is out "outfile.txt";
variable check:std_logic_vector(15 downto 0);
    :=to_stdlogicvector(x"0008");
begin
wait for 10 ns;
while not(endfile(infile)) loop
    read_vl(infile,check);
    write_vl(outfile,check);
end loop;
end process;
end behavioral;
The Package TEXTIO

* Provides standard set of file types, data types, and input/output functions (see Appendix D).

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
entity asynch_dff is
  port(R,S,D,Clk:in std_logic; Q,Qbar:out std_logic);
end asynch_dff;
architecture behavioral of asynch_dff is
begin
  output:process(R,S,Clk) begin
    if (R='0') then
      Q <= '0' after 5 ns;
      Qbar <= '1' after 5 ns;
    elsif (S='0') then
      Q <= '1' after 5 ns;
      Qbar <= '0' after 5 ns;
    elsif (Clk='event and Clk='1') then
      Q <= D after 5 ns;
      Qbar <= not D after 5 ns;
    end if;
  end process output;
end behavioral;
```

Testbenches in VHDL

* Apply stimulus to the input ports.
* Reads the outputs of the module being tested.

Tester for Asynch DFF

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use STD.textio.all;
use WORK.classio.all;
entity srtester is
  port(R,S,D,Clk:out std_logic;
       Q,Qbar:in std_logic);
end srtester;
architecture behavioral of srtester is
begin
  clk_process:process begin
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    wait for 10 ns;
  end process clk_process;
end behavioral;
```
Tester for Asynch DFF (cont)

io_process: process
file infile: TEXT is in "infile.txt";
file outfile: TEXT is in "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
    read_vld(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);
      write_vld(outfile, check);
    end if;
  end loop;
end process io_process;
end behavioral;

Structural Description of the Testbench

library IEEE;
use IEEE.std_logic_1164.all;
use WORK.classio.all;
entity srbench is
end srbench;
architecture behavioral of srbench is
component asynch_dff
  port(R,S,D,Clk: std_logic; Q,Qbar:out std_logic);
end component;
component srtester
  port(R,S,D,Clk: std_logic; Q,Qbar:in std_logic);
end component;
for M1: asynch_dff use entity WORK.asynch_dff(behavioral);
signal r,s,d,q,qb,clk:std_logic;
begin
  T1: srtester port map(R=>r,S=>s,D=>d,Q=>q,Qbar=>qb,Clk=>clk);
  M1: asynch_dff port map(R=>r,S=>s,D=>d,Q=>q,Qbar=>qb,Clk=>clk);
end behavioral;

Testbench for Asynch_dff: Infile.txt

11001
01101
11110
10010
10011 -- illegal case

ASSERT Statement

* Previous example recorded errors or failures by writing the test vector to the file “outfile.txt”.
* Alternatively, we could use an assert statement:

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

* Severity levels: NOTE, WARNING, ERROR, and FAILRE.
Data Objects

※ Three basic types:
※ signals,
※ variables, and
※ constants.
※ Each has specific data type (ex. bit or integer) and a unique set of possible values.

Scalar Types

Table 1: VHDL’s Built in Scalar Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Typical Values</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>‘1’, ‘0’</td>
<td>Enumerated type.</td>
</tr>
<tr>
<td>Boolean</td>
<td>True, False</td>
<td>Result of comparison.</td>
</tr>
<tr>
<td>Integer</td>
<td>-2,-1,0,1,2,3,4,...</td>
<td>Minimum 32 bit precision.</td>
</tr>
<tr>
<td>Natural</td>
<td>0,1,2,...</td>
<td>Non-negative numbers</td>
</tr>
<tr>
<td>Positive</td>
<td>1,2,3,...</td>
<td>All positive numbers</td>
</tr>
<tr>
<td>Real</td>
<td>1.0, -1.0E5</td>
<td>Minimum range +-1E38</td>
</tr>
<tr>
<td>Severity_level</td>
<td>note, warning, error, and failure.</td>
<td>Enumerated type.</td>
</tr>
<tr>
<td>Time</td>
<td>1 ua, 7 ns, 100 ps</td>
<td>A physical type.</td>
</tr>
</tbody>
</table>
Enumerated Types

* Enumerated types can be used to describe high-level design concepts using symbolic values.

architecture FSM of PARITY is
    type states is (Even,Odd);
    signal current_state:states;
begin
    strans:process(Rst,Clk)
    begin . . .
        case current_state is
        when Even=> . . .
        when Odd=> . . .
        end case
    end process;
end FSM;

Composite Types

Table 1: VHDL’s Built in Composite Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Typical Values</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit_vector</td>
<td>'001001011&quot;, &quot;10&quot;</td>
<td>Array of bit.</td>
</tr>
<tr>
<td>String</td>
<td>“Simulation failed&quot;</td>
<td>Array of characters.</td>
</tr>
<tr>
<td>records</td>
<td>Any collection of values</td>
<td>User defined composite.</td>
</tr>
</tbody>
</table>

Time and Other Physical Types

* Distinguished by having units of measure.

type time is range -2147483647 to 2147483647
    units
        fs;
        ps = 1000 fs;
        ns = 1000 ps;
        us = 1000 ns;
        ms = 1000 us;
        sec = 1000 ms;
        min = 60 sec;
        hr = 60 min;
end units;

Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>And</td>
<td>Any bit or boolean</td>
<td>Same type</td>
</tr>
<tr>
<td>or</td>
<td>Or</td>
<td>Any bit or boolean</td>
<td>Same type</td>
</tr>
<tr>
<td>nand</td>
<td>Not and</td>
<td>Any bit or boolean</td>
<td>Same type</td>
</tr>
<tr>
<td>nor</td>
<td>Not or</td>
<td>Any bit or boolean</td>
<td>Same type</td>
</tr>
<tr>
<td>xor</td>
<td>Exclusive or</td>
<td>Any bit or boolean</td>
<td>Same type</td>
</tr>
<tr>
<td>nxor</td>
<td>Exclusive nor</td>
<td>Any bit or boolean</td>
<td>Same type</td>
</tr>
</tbody>
</table>
## Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equality</td>
<td>Any type</td>
<td>Boolean</td>
</tr>
<tr>
<td>/=</td>
<td>Inequality</td>
<td>Any type</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;</td>
<td>Ordering</td>
<td>Any scalar or discrete array type</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Adding Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Any numeric type</td>
<td>Same type</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Any numeric type</td>
<td>Same type</td>
</tr>
<tr>
<td>&amp;</td>
<td>Concat</td>
<td>Any array or element</td>
<td>Same type</td>
</tr>
</tbody>
</table>

## Multiplying Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Left: integer or real Right: same type</td>
<td>Same type</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Left: physical type Right: integer or real</td>
<td>Same as left</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Left: integer or real Right: physical type</td>
<td>Same as right</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Any integer or real</td>
<td>Same type</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Left: physical type Right: integer or real</td>
<td>Same as left</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Left: physical type Right: same type</td>
<td>Integer</td>
</tr>
</tbody>
</table>

## Miscellaneous Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>Modulus</td>
<td>Any integer type</td>
<td>Same type</td>
</tr>
<tr>
<td>rem</td>
<td>Remainder</td>
<td>Any integer type</td>
<td>Same type</td>
</tr>
<tr>
<td>+</td>
<td>Identity</td>
<td>Any numeric type</td>
<td>Same type</td>
</tr>
<tr>
<td>-</td>
<td>Negation</td>
<td>Any numeric type</td>
<td>Same type</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td>Left: integer type Right: integer type</td>
<td>Same as left</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
<td>Left: real type Right: integer type</td>
<td>Same as left</td>
</tr>
<tr>
<td>abs</td>
<td>Absolute value</td>
<td>Any numeric type</td>
<td>Same type</td>
</tr>
<tr>
<td>not</td>
<td>Logical negation</td>
<td>Any bit or boolean</td>
<td>Same type</td>
</tr>
</tbody>
</table>
### Shift Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand types</th>
<th>Result type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll</td>
<td>Shift left logical</td>
<td>Left: any one-dim array whose element type is bit or boolean</td>
<td>Same as left</td>
</tr>
<tr>
<td>srl</td>
<td>Shift right logical</td>
<td>(same as above)</td>
<td>Same as left</td>
</tr>
<tr>
<td>sla</td>
<td>Shift left arith.</td>
<td>(same as above)</td>
<td>Same as left</td>
</tr>
<tr>
<td>sra</td>
<td>Shift right arith.</td>
<td>(same as above)</td>
<td>Same as left</td>
</tr>
<tr>
<td>rol</td>
<td>Rotate left logical</td>
<td>(same as above)</td>
<td>Same as left</td>
</tr>
<tr>
<td>ror</td>
<td>Rotate right logical</td>
<td>(same as above)</td>
<td>Same as left</td>
</tr>
</tbody>
</table>

### What We Have Learned So Far

- Files, file types, and file declarations.
- Basic operations for reading and writing files.
- Creating procedures for r/w different data types.
- Construction and operation of testbenches.
- Miscellaneous information about identifiers, data types, and operators.
- NEXT WEEK: Start project group meetings.