### 8.1 Attributes - Signal Attributes that return a value

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'EVENT</td>
<td>True if an event occurred during the current delta, else false</td>
</tr>
<tr>
<td>S'ACTIVE</td>
<td>True if a transaction occurred during the current delta, else false</td>
</tr>
<tr>
<td>S'LAST_EVENT</td>
<td>Time elapsed since the previous event on S</td>
</tr>
<tr>
<td>S'LAST_VALUE</td>
<td>Value of S before the previous event on S</td>
</tr>
<tr>
<td>S'LAST_ACTIVE</td>
<td>Time elapsed since previous transaction on S</td>
</tr>
</tbody>
</table>

A event — true if a _______________ has just occurred

A active — true if A has _________________, even if A does not change
8.1 Attributes - Signal

Attribute Evaluation Example

Signal Attributes

A <= B  --  B changes at time T

¥ Event
— occurs on a signal every time it is ___________

¥ Transaction
— occurs on a signal every time it is ___________

¥ Example:

<table>
<thead>
<tr>
<th>A event</th>
<th>B event</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T + 1d</td>
<td></td>
</tr>
</tbody>
</table>

8.1 Attributes - Signal Attributes

that create a signal

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Creates</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'DELAYED [(time)]*</td>
<td>signal same as S delayed by specified time</td>
</tr>
<tr>
<td>S'全域 [(time)]*</td>
<td>Boolean signal that is true if S had no events for the specified time</td>
</tr>
<tr>
<td>S'QUIET [(time)]*</td>
<td>Boolean signal that is true if S had no transactions for the specified time</td>
</tr>
<tr>
<td>S'TRANSACTION</td>
<td>signal of type BIT that changes for every transaction on S</td>
</tr>
</tbody>
</table>

* Delta is used if no time is specified
8.1 Attributes - Attribute Test

VHDL and Waveforms

entity attr_ex is
  port (B,C : in bit);
end attr_ex;

architecture test of attr_ex is
  signal A, C_delayed5, A_trans : bit;
  signal A_stable5, A_quiet5 : boolean;
begin
  A <= B and C;
  C_delayed5 <= C'delayed(5 ns);
  A_trans <= A'transaction;
  A_stable5 <= A'stable(5 ns);
  A_quiet5 <= A'quiet(5 ns);
end test;

check: process
begin
  wait until rising_edge(Clk);
  assert (D'stable(setup_time))
    report("Setup time violation")
    severity error;
  wait for hold_time;
  assert (D'stable(hold_time))
    report("Hold time violation")
    severity error;
end process check;
### Chapter 8

#### CPE/EE 422/522

**Electrical and Computer Engineering**

#### Assert Statement

```plaintext
assert boolean-expression
report string-expression
severity severity-level
```

- If boolean expression is ______
  display the string expression on the monitor
- Severity levels: __________, __________, __________, __________

---

#### 8.1 Attributes - Array Attributes

**Type** ROM is array (0 to 15, 7 downto 0) of bit;
**Signal** ROM1 : ROM;

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Returns</th>
<th>Examples</th>
</tr>
</thead>
</table>
| A'LEFT(N)       | left bound of Nth index range| ROM1\_LEFT(1) = 0  
                  |                              | ROM1\_LEFT(2) = 7  |
| A'RIGHT(N)      | right bound of Nth index range | ROM1\_RIGHT(1) = 15  
                  |                              | ROM1\_RIGHT(2) = 0  |
| A'HIGH(N)       | largest bound of Nth index range | ROM1\_HIGH(1) = 15       
                  |                              | ROM1\_HIGH(2) = 7    |
| A'LOW(N)        | smallest bound of Nth index range | ROM1\_LOW(1) = 0       
                  |                              | ROM1\_LOW(2) = 0    |
| A'LENGTH(N)     | size of Nth index range     | A can be either an __________ or an __________.  
                  |                              | A\_LENGTH(1) = 16  
                  |                              | A\_LENGTH(2) = 8    |
| A'REVERSE_RANGE(N) | Nth index range reversed | Array attributes work  
                  |                              | with __________, __________,  
                  |                              | and __________.  
                  |                              | ROM1\_REVERSE\_RANGE(1) = 15  
                  |                              | downto 0  
                  |                              | ROM1\_REVERSE\_RANGE(2) = 0 to 7  

---

Electrical and Computer Engineering

Spring 2004 Slide #7
8.1 Attributes - Procedure for Adding Vectors without Using Attributes

-- This procedure adds two n-bit bit_vectors and a carry and
-- returns an n-bit sum and a carry. Add1 and Add2 are assumed
-- to be of the same length and dimensioned n-1 downto 0.

procedure Addvec

(Add1, Add2: in bit_vector;
  Cin: in bit;
  signal Sum: out bit_vector;
  signal Cout: out bit;
  n: in positive) is

variable C: bit;

begin
  C := Cin;
  for i in 0 to n-1 loop
    Sum(i) <= Add1(i) xor Add2(i) xor C;
  end loop;
end Addvec;

Note: Add1 and Add2 vectors must be dimensioned as N-1 downto 0.

Use ___________ to write more general procedure that places no restrictions on the range of vectors other than the __________ must be the same.

8.1 Attributes - Procedure for Adding Vectors with Using Attributes

-- This procedure adds two bit_vectors and a carry and returns a sum
-- and a carry. Both bit_vectors should be of the same length.

procedure Addvec2

(Add1, Add2: in bit_vector;
  Cin: in bit;
  signal Sum: out bit_vector;
  signal Cout: out bit) is

variable C: bit := Cin;
alias n1: bit_vector(Add1'length-1 downto 0) is Add1;
alias n2: bit_vector(Add2'length-1 downto 0) is Add2;
alias S : bit_vector(Sum'length-1 downto 0) is Sum;

begin
  assert ((n1'length = n2'length) and (n1'length = S'length))
  report "Vector lengths must be equal!"
  severity error;
  for i in s'reverse_range loop
    S(i) <= n1(i) xor n2(i) xor C;
  end loop;
end Addvec2;
8.2 Transport and Inertial Delays

Reject is equivalent to a combination of inertial and transport delay:

\[ Z_1 \leq X \text{ after } 4 \text{ ns}; \]
\[ Z_2 \leq X \text{ after } 10 \text{ ns}; \]
\[ Z_3 \leq \text{ transport } Z_2 \text{ after } 6 \text{ ns}; \]

8.3 Operator Overloading

- Operators +, - operate on integers
- Write procedures for bit vector addition/subtraction
  - addvec, subvec
- Operator overloading allows using + operator to implicitly call an appropriate addition function
- How does it work?
  - When compiler encounters a function declaration in which the function name is an operator enclosed in double quotes, the compiler treats the function as an operator overloading (+)
  - When a + operator is encountered, the compiler automatically checks the types of operands and calls appropriate functions
8.3 Operator Overloading - VHDL Package

```vhdl
package bit_overload is
  function "+" (Add1, Add2: bit_vector) return bit_vector;
  function "+" (Add1: bit_vector, Add2: integer) return bit_vector;
end bit_overload;

library BTLIB;
use BTLIB.bit_pack.all;

package body bit_overload is
  -- This function returns a bit_vector sum of two bit_vector operands.
  function "+" (Add1, Add2: bit_vector) return bit_vector is
    variable sum: bit_vector(Add1'length downto 0);
    variable c: bit := '0';    -- no carry in
    alias a1: bit_vector(Add1'length downto 0) is Add1;
    alias a2: bit_vector(Add2'length downto 0) is Add2;
    begin
      for i in reverse range sum'range loop
        sum(i) := a1(i) xor a2(i) xor c;
        c := (a1(i) and a2(i)) or (a1(i) and c) or (a2(i) and c);
      end loop;
      return sum;
    end "+";

  -- This function returns a bit_vector sum of a bit_vector and an integer
  -- using the previous function after the integer is converted.
  function "+" (Add1: bit_vector, Add2: integer) return bit_vector is
    begin
      return (bitvec_add (Add1, int2vec (Add2, Add1'length)));
    end "+";
end bit_overload;
```

Overloading can also be applied to procedures and functions — procedures have the same name — type of the actual parameters in the procedure call determines which version of the procedure is called.
8.4 Multivalued Logic and Signal Resolution

- Bit (0, 1)
- Tristate buffers and buses => high impedance state Z
- Unknown state X
  - e.g., a gate is driven by Z, output is unknown
  - a signal is simultaneously driven by 0 and 1

Resolution function to determine the actual value of f since it is driven from two different sources
8.3 Multivalued Logic and Signal Resolution

¥ VHDL signals may either be ___________ or ___________

¥ Resolved signals have an associated ________________

¥ Bit type is unresolved —
  — there is no resolution function
  — if you drive a bit signal to two different values
    in two concurrent statements, the compiler will ________________

```vhdl
signal R : X01Z := 'Z'; ...
R <= transport '0' after 2 ns, 'Z' after 6 ns;
R <= transport '1' after 4 ns;
R <= transport '1' after 8 ns, '0' after 10 ns;
```

Resolution Function

<table>
<thead>
<tr>
<th>Time</th>
<th>$s(0)$</th>
<th>$s(1)$</th>
<th>$s(2)$</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>'Z'</td>
<td>'Z'</td>
<td>'Z'</td>
<td>'Z'</td>
</tr>
<tr>
<td>2</td>
<td>'0'</td>
<td>'Z'</td>
<td>'Z'</td>
<td>'Z'</td>
</tr>
<tr>
<td>4</td>
<td>'0'</td>
<td>'1'</td>
<td>'Z'</td>
<td>'X'</td>
</tr>
<tr>
<td>6</td>
<td>'Z'</td>
<td>'1'</td>
<td>'Z'</td>
<td>'1'</td>
</tr>
<tr>
<td>8</td>
<td>'Z'</td>
<td>'1'</td>
<td>'1'</td>
<td>'1'</td>
</tr>
<tr>
<td>10</td>
<td>'Z'</td>
<td>'1'</td>
<td>'0'</td>
<td>'X'</td>
</tr>
</tbody>
</table>
8.3 Multivalued Logic and Signal Resolution - Resolution Function VHDL

```
package fourpack is
  type u_x01z is ('X','0','1','Z'); -- u_x01z is unresolved
  type u_x01z_vector is array (natural range <>) of u_x01z;
  function resolved (scu_x01z_vector) return u_x01z;
end fourpack;

package body fourpack is
  type x01z_table is array (u_x01z_vector) of u_x01z;
  constant resolution_table : x01z_table := (
    'X','X','X','X',
    'X','X','X','X',
    'X','X','X','X',
    'X','X','X','X');
  function resolved (scu_x01z_vector) return u_x01z is
    variable result : u_x01z := 'Z';
    begin
      if (x'length = 1) then
        return result;
      else
        for i in x'range loop
          result := resolution_table(result(i));
        end loop;
        return result;
      end if;
    end resolved;
end fourpack;
```

8.5 IEEE-1164 Standard Logic

<table>
<thead>
<tr>
<th>¥ 9-valued logic system</th>
</tr>
</thead>
<tbody>
<tr>
<td>U — Uninitialized</td>
</tr>
<tr>
<td>X — Forcing Unknown</td>
</tr>
<tr>
<td>0 — Forcing 0</td>
</tr>
<tr>
<td>1 — Forcing 1</td>
</tr>
<tr>
<td>Z — High impedance</td>
</tr>
<tr>
<td>W — Weak unknown</td>
</tr>
<tr>
<td>L — Weak 0</td>
</tr>
<tr>
<td>H — Weak 1</td>
</tr>
<tr>
<td>- — Don't care</td>
</tr>
</tbody>
</table>

If forcing and weak signal are tied together, the forcing signal dominates.

Useful in modeling the internal operation of certain types of ICs.

In this course we use a subset of the IEEE values: X10Z
8.5 IEEE-1164 Standard Logic -
Resolution Function

CONSTANT resolution_table : stdlogic_table := {
   -- | U X O 1 Z W L H |
   -- ---------------------------------------- -- |
   ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- U |
   ( 'U', 'X', 'X', 'X', 'O', 'O', 'O', 'O' ), -- X |
   ( 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O' ), -- O |
   ( 'U', 'U', 'U', 'U', 'X', 'X', 'X', 'X' ), -- H |
   ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- X |}

8.5 IEEE-1164 Standard Logic -
AND Definition

CONSTANT and_table : stdlogic_table := {
   -- | U X O 1 Z W L H |
   -- ---------------------------------------- -- |
   ( 'U', 'U', 'O', 'O', 'U', 'U', 'U', 'U' ), -- U |
   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- X |
   ( 'O', 'O', 'O', 'O', 'O', 'O', 'O', 'O' ), -- O |
   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- H |
   ( 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- X |}
### 8.5 IEEE-1164 Standard Logic - AND Function VHDL

```vhdl
function "and" ( l : std_ulogic; r : std_ulogic ) return UX01 is
    begin
        return (and_table(l, r));
    end "and";

function "and" ( l,r : std_logic_vector ) return std_logic_vector is
    alias lv : std_logic_vector ( 1 to 'LENGTH ) is l;
    alias rv : std_logic_vector ( 1 to r'LENGTH ) is r;
    variable result : std_logic_vector ( 1 to 'LENGTH );
    begin
        if ( 'LENGTH /= r'LENGTH ) then
            assert FALSE
                report "arguments of overloaded 'and' operator are not of the same length" severity FAILURE;
        else
            for i in result'RANGE loop
                result(i) := and_table(lv(i), rv(i));
            end loop;
        end if;
        return result;
    end "and";
```

### 8.6 Generics

- Used to specify ___________ for a _____________ in such a way that the _____________ values must be specified when the _____________ is instantiated

- Example: rise/fall time modeling

```vhdl
entity NAND2 is
genetic ( Trise, Tfall: time; load: natural );
port (a,b : in bit; c : out bit);
end NAND2;
architecture behavior of NAND2 is
signal nand_value : bit;
begin
    nand_value <= a and b;
    c <= nand_value after ( Trise + 3 ns * load ) when nand_value = '1'
        else nand_value after ( Tfall + 2 ns * load );
end behavior;
```
8.6 Generics - Values are given in Component Instantiations

entity NAND2_test is
port (in1, in2, in3, in4 : in bit;
     out1, out2 : out bit);
end NAND2_test;
architecture behavior of NAND2_test is
component NAND2 is
    generic (Trise : time := 3 ns; Tfall : time := 2 ns;
             load : natural := 1);
    port (a, b : in bit;
          c : out bit);
    end component;
begin
U1: NAND2 generic map (2 ns, 1 ns, 2) port map (in1, in2, out1);
U2: NAND2 port map (in3, in4, out2);
end behavior;

8.7 Generate Statements

¥ Provides an easy way of instantiating components when we have an ___________ array of identical components
¥ Example: 4-bit ripple carry adder
8.7 Generate Statements - Example: 4-bit Adder without Generate Statements

entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit; -- Inputs
        S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
end Adder4;

architecture Structure of Adder4 is
component FullAdder
  port (X, Y, Cin: in bit; -- Inputs
        Cout, Sum: out bit); -- Outputs
end component;
signal C: bit_vector(3 downto 1);
begin -- instantiate four copies of the FullAdder
  FA0: FullAdder port map (A(0), B(0), Ci, C(1), S(0));
  FA1: FullAdder port map (A(1), B(1), C(1), C(2), S(1));
  FA2: FullAdder port map (A(2), B(2), C(2), C(3), S(2));
  FA3: FullAdder port map (A(3), B(3), C(3), Co, S(3));
end Structure;

8.7 Generate Statements - Example: 4-bit Adder with Generate Statements

entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); Ci: in bit; -- Inputs
        S: out bit_vector(3 downto 0); Co: out bit); -- Outputs
end Adder4;

architecture Structure of Adder4 is
component FullAdder
  port (X, Y, Cin: in bit; -- Inputs
        Cout, Sum: out bit); -- Outputs
end component;
signal C: bit_vector(4 downto 0);
begin
  C(0) <= Ci;
  -- generate four copies of the FullAdder
  FullAdd4: for i in 0 to 3 generate
    begin
      FAx: FullAdder port map (A(i), B(i), C(i), C(i+1), S(i));
    end generate FullAdd4;
  end generate FullAdd4;
end Structure;
8.8 Synthesis of VHDL Code

- Synthesizer
  - take a VHDL model as an input
  - synthesize the logic: output is a structural gate level implementation
- Synthesizers accept a subset of VHDL as input
- Efficient implementation?

Context

```vhdl
A <= B and C;
wait until clk'event and clk = '1';
A <= B and C;
```

Implies CM for A

Implies a register or flip-flop

---

8.8 Synthesis of VHDL Code (cont'd)

- Always use constrained integers
  - if not specified, the synthesizer may infer 32-bit register
- When integer range is specified, most synthesizers will implement integer addition and subtraction using binary adders with appropriate number of bits
  - integer range 0 to 7 gives 3 unsigned bits
  - Integer range -8 to 7 gives 4 signed bits
- General rule: when a signal is assigned a value, it will hold that value until it is assigned new value
### 8.8 Synthesis of VHDL Code -
#### Unintentional Latch Creation

**What if a = 3?**

The previous value of b should be held in the latch, so G should be 0 when a = 3.

```vhdl
entity latch_example is
  port(a: in integer range 0 to 3;
       b: out bit);
end latch_example;

architecture test1 of latch_example is
begin
  process(a)
  begin
    case a is
      when 0 => b <= '1';
      when 1 => b <= '0';
      when 2 => b <= '1';
      when others => b <= null;
    end case;
  end process;
end test1;
```

### 8.8 Synthesis of VHDL Code -
#### How are Unspecified Cases Handled?

**What if A /= 1?**

Retain the previous value for NextState?

Synthesizer might interpret this to mean that NextState is unknown!

```vhdl
if A = '1' then NextState <= 3;
end if;
```

```vhdl
if A = '1' then NextState <= 3;
else  NextState <= 2;
end if;
```
8.8 Synthesis of VHDL Code - Case Statement

```vhdl
entity case_example is
  port(a: in integer range 0 to 3;
       b: out integer range 0 to 3);
end case_example;
architecture test1 of case_example is
begin
  process(a)
  begin
    case a is
      when 0 => b <= 1;
      when 1 => b <= 3;
      when 2 => b <= 0;
      when 3 => b <= 1;
    end case;
  end process;
end test1;
```

8.8 Synthesis of VHDL Code - Case Statement: Pre- and Post- Optimization
8.8 Synthesis of VHDL Code - If Statement

entity if_example is
  port(A, B: in bit;
   C, D, E: in bit_vector(2 downto 0);
   Z: out bit_vector(2 downto 0));
end if_example;

architecture test1 of if_example is
begin
  process(A, B)
  begin
    if A = '1' then Z <= C;
    elsif B = '1' then Z <= D;
    else Z <= E;
  end if;
end process;

end test1;

Synthesized code before optimization

8.8 Synthesis of VHDL Code - Standard VHDL Synthesis Package

Every VHDL synthesis tool provides its own package of functions for operations commonly used in hardware models.

IEEE is developing a standard synthesis package, which includes functions for arithmetic operations on bit_vectors and std_logic vectors:

- numeric_bit package defines operations on bit_vectors
  - type unsigned is array (natural range<>) of bit;
  - type signed is array (natural range<>) of bit;
- package include overloaded versions of arithmetic, relational, logical, and shifting operations, and conversion functions
- numeric_std package defines similar operations on std_logic vectors
### 8.8 Synthesis of VHDL Code - Numeric_bit, Numeric_std

- **Overloaded operators**
  - **Unary:** abs, -
  - **Arithmetic:** +, -, *, /, rem, mod
  - **Relational:** >, <, >=, <=, =, /=
  - **Logical:** not, and, or, nand, nor, xor, xnor
  - **Shifting:** shift_left, shift_right, rotate_left, rotate_right, sll, srl, rol, ror

---

If the left and right signed operands are of different lengths, the shortest operand will be sign-extended before performing an arithmetic operation. For unsigned operands, the shortest operand will be extended by filling in 0's on the left. Examples:

- **signed:** "01101" + "1011" becomes "01101" + "1011" = "01080"
- **unsigned:** "01101" + "1011" becomes "01101" + "01011" = "11000"

When addition is performed on unsigned or signed operands, the final carry is discarded and overflow is ignored. If a carry is needed, an extra bit can be added to one of the operands. Examples:
8.8 Synthesis of VHDL Code - 
Numeric_bit, Numeric_std (cont’d)

constant A: unsigned(3 downto 0) := "1101";
constant B: signed(3 downto 0) := "1011";
variable Sumu: unsigned(4 downto 0);
variable Sums: signed(4 downto 0);
variable Overflow: boolean

Sumu := '0' & A + unsigned("0101");
-- result is "10010" (sum = 2, carry = 1)
Sums := B(3) & B + signed("1101");
-- result is "11000" (sum = -8, carry = 1)
Overflow := Sums(4) /= Sums(3) -- Overflow is false

In the above example, the notation unsigned("0101") is a type qualification which assigns the type unsigned to the bit vector "0101".

8.9 Synthesis Examples

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity examples is
  port (signal clock: in bit;
        signal A, B: in signed(3 downto 0);
        signal ge: out boolean;
        signal acc: inout signed(3 downto 0) := "0000");
  signal count: inout unsigned(3 downto 0) := "0000");
end examples;

architecture x1 of examples is
begin
  ge <= (A >= B); -- 4-bit comparator
  process
  begin
    wait until clock'event and clock = '1';
    acc <= acc + B; -- 4-bit register and 4-bit adder
    count <= count + 1; -- 4-bit counter
  end process;
end;
8.9 Synthesis Examples - Mealy BCD to BCD + 3 Converter

```vhdl
entity SM1_2 is
  port(X, CLK: in bit; Z: out bit);
end SM1_2;
architecture Table of SM1_2 is
  signal State, Nextstate: s_type;
  constant S0: s_type := 0;
  constant S1: s_type := 1;
  constant S2: s_type := 2;
  constant S3: s_type := 3;
  constant S4: s_type := 4;
  constant S5: s_type := 5;
  constant S6: s_type := 6;
begin
  process(State, X) begin -- Combinational Network
    case State is
      when S0 =>
        if X='0' then Z<='1'; Nextstate<=S1;
        else Z<='0'; Nextstate<=S2; end if;
      when S1 =>
        if X='0' then Z<='1'; Nextstate<=S3;
        else Z<='0'; Nextstate<=S4; end if;
      when S2 =>
        if X='0' then Z<='0'; Nextstate<=S4;
        else Z<='1'; Nextstate<=S4; end if;
      when others => null;
    end case;
  end process;
  process(CLK) begin -- State Register
    if CLK='1' and CLK'event then
      State <= Nextstate;
    end if;
  end process;
  end Table;
end SM1_2;
```
8.9 Synthesis Examples -
Mealy BCD to BCD + 3 Converter

3 FF, 13 gates

8.10 Files and TEXTIO

- File input/output in VHDL
- Used in _____________
  —___________ of test data
  —__________ for test results
- VHDL provides a standard TEXTIO package
  —read/write lines of text
8.10 Files and TEXTIO -
File Declarations

File Declaration

\[ \text{file file-name: file-type [open mode] is "file-pathname";} \]

Example:

\[ \text{file test_data: text open read_mode is "c:\test1\test.dat"} \]

\( \Rightarrow \) declares a file named test_data of type text which is opened in the read mode. The physical location of the file is in the test1 directory on the c: drive.

Modes for Opening a File

- **read_mode**: file elements can be read using a read procedure
- **write_mode**: new empty file is created; elements can be written using a write procedure
- **append_mode**: allows writing to an existing file

8.10 Files and TEXTIO -
Standard TEXTIO Package

- Contains __________ and __________ for working with _____ composed of lines of text

\( \Rightarrow \) Defines a file type named text:

\[ \text{type text is file of string;} \]

- Contains procedures for __________ of text from a file of type text and for __________ of text to a file
8.10 Files and TEXTIO -  
Reading from a text file

¥ ______________reads a line of text and places it in a buffer with an associated pointer
¥ The pointer to the buffer must be of type line, which is declared in the textio package as:
   – type line is access string;
¥ When a variable of type line is declared, it creates a pointer to a string
¥ Code
   variable buff: line;
   ...
   readline (test_data, buff);
   — reads a line of text from test_data and places it in a buffer which is pointed to by buff

8.10 Files and TEXTIO -  
Extracting Data from the Line Buffer

¥ To extract data from the line buffer, call a ______ procedure one or more times
¥ For example, if bv4 is a bit_vector of length four ____________, the call
   read(buff, bv4)
   — extracts a 4-bit vector from the buffer, sets bv4 equal to this vector, and adjusts the pointer buff to point to the next character in the buffer. Another call to read will then extract the next data object from the line buffer.
8.10 Files and TEXTIO - Extracting Data from the Line Buffer

¥ TEXTIO provides overloaded read procedures to read data of types ____,
______, ____
______, ____, ____, ____, ____, and ____ from buffer

¥ Read forms
  read(pointer, value)
  read(pointer, value, good)
— good is _______ that returns TRUE if the read is
______ and FALSE if it is not
— type and size of value determines which of the read
  procedures is called
— character, strings, and bit_vectors within files of type text are
  not delimited by quotes

8.10 Files and TEXTIO - Writing to TEXTIO files

¥ Call one or more ______ procedures to write data
to a line buffer and then call ___________ to write the line to a file
  variable buffw : line;
  variable int1 : integer;
  variable bv8 : bit_vector(7 downto 0);
  ...
  write(buffw, int1, right, 6); -- right just., 6 ch. wide
  write(buffw, bv8, right, 10);
  writeln(buffw, output_file);

¥ Write parameters:
  1) buffer pointer of type line,
  2) a value of any acceptable type,
  3) justification (left or right), and
  4) field width (number of characters)
8.10 Files and TEXTIO - An Example

¥ Procedure to read data from a file and store the data in a memory array
¥ Format of the data in the file
  - address N comments
  byte1 byte2 ... byteN comments
  ¥ address — 4 hex digits
  ¥ N — indicates the number of bytes of code
  ¥ bytei — 2 hex digits
  ¥ each byte is separated by one space
  ¥ the last byte must be followed by a space
  ¥ anything following the last state will not be read and will be treated as a comment

8.10 Files and TEXTIO - An Example (cont'd)

¥ Code sequence: an example
  - 12AC 7 (7 hex bytes follow)
    AE 03 B6 91 C7 00 0C (LDX imm, LDA dir, STA ext)
    005B 2 (2 bytes follow)
    01 FC
¥ TEXTIO does not include read procedure for hex numbers
  — we will read each hex value as a string of characters and then convert the string to an integer
¥ How to implement conversion?
  ¥ table lookup — constant named lookup is an array of integers indexed by characters in the range 0 to F
  ¥ this range includes the 23 ASCII characters:
    0, 1, ... 9, :, ;, <, =, >, ?, @, A, ... F
  ¥ corresponding values:
    0, 1, ... 9, -1, -1, -1, -1, 10, 11, 12, 13, 14, 15
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all; -- CONV_STD_LOGIC_VECTOR(int, size)
use std.textio.all;

entity testfill is
end testfill;

architecture fillmem of testfill is

type RAMtype is array (0 to 8191) of std_logic_vector(7 downto 0);
signal mem: RAMtype := (others=>(others=> '0'));

procedure fill_memory(signal mem: inout RAMtype) is

type HexTable is array(character range <>) of integer;
-- valid hex chars: 0, 1, ... A, B, C, D, E, F (upper-case only)
constant lookup : HexTable('0' to 'F') :=
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -1, -1, -1, -1,
 -1, -1, -1, -1, 10, 11, 12, 13, 14, 15);
file infile: text open read_mode is "mem1.txt"; -- open file for reading
-- file infile: text is in "mem1.txt"; -- VHDL '87 version
variable buff: line;
variable addr_s: string(4 downto 1);
variable data_s : string(3 downto 1); -- data_s(1) has a space
variable addr1, byte_cnt: integer; variable data: integer range 255 downto 0;

begin
while (not endfile(infile)) loop
readline (infile, buff); -- read addr hexnum
read(buff, addr_s); -- read number of bytes to read
addr1 := lookup(addr_s(4))*4096 + lookup(addr_s(3))*256 + lookup(addr_s(2))*16 + lookup(addr_s(1));
readline (infile, buff);
for i in 1 to byte_cnt loop
read (buff, data_s); -- read 2 digit hex data and a space
data := lookup(data_s(3))*16 + lookup(data_s(2));
mem(addr1) <= CONV_STD_LOGIC_VECTOR(data, 8);
addr1 := addr1 + 1;
end loop;
end loop;
end fill_memory;

begin
  testbench: process
  begin
    fill_memory(mem); -- insert code that uses memory data
  end process;
end testmem;

end fillmem;