### Table 8-1  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>

### Table 8-2  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'STABLE [(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
Figure 8-1 Examples of Signal Attributes

**VHDL Code for Attribute Test**

```vhdl
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;
```

**Waveforms for Attribute Test**

![Waveforms](image_url)
### Table 8-3 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>
<tbody>
<tr>
<td>A'LEFT(N)</td>
<td>left bound of Nth index range</td>
<td>ROM1'LEFT(1) = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROM1'LEFT(2) = 7</td>
</tr>
<tr>
<td>A'RIGHT(N)</td>
<td>right bound of Nth index range</td>
<td>ROM1'RIGHT(1) = 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROM1'RIGHT(2) = 0</td>
</tr>
<tr>
<td>A'HIGH(N)</td>
<td>largest bound of Nth index range</td>
<td>ROM1'HIGH(1) = 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROM1'HIGH(2) = 7</td>
</tr>
<tr>
<td>A'LOW(N)</td>
<td>smallest bound of Nth index range</td>
<td>ROM1'LOW(1) = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROM1'LOW(2) = 0</td>
</tr>
<tr>
<td>A'RANGE(N)</td>
<td>Nth index range</td>
<td>ROM1'RANGE(1) = 0 <strong>to</strong> 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROM1'RANGE(2) = 7 <strong>downto</strong> 0</td>
</tr>
<tr>
<td>A'REVERSE_RANGE(N)</td>
<td>Nth index range reversed</td>
<td>ROM1'REVERSE_RANGE(1) = 15<strong>downto</strong> 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROM1'REVERSE_RANGE(2) = 0 <strong>to</strong> 7</td>
</tr>
<tr>
<td>A'LENGTH(N)</td>
<td>size of Nth index range</td>
<td>ROM1'LENGTH(1) = 16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ROM1'LENGTH(2) = 8</td>
</tr>
</tbody>
</table>
Figure 8-2  Procedure for Adding Bit-Vectors

-- 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;
    C := (n1(i) and n2(i)) or (n1(i) and C) or (n2(i) and C);
  end loop;
  Cout <= C;
end Addvec2;
Figure 8-3 Transport and Inertial Delays

Z1 <= transport X after 10 ns; -- transport delay
Z2 <= X after 10 ns; -- inertial delay
Z3 <= reject 4 ns X after 10 ns; -- delay with specified rejection pulse width
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 BITLIB;
use BITLIB.bit_pack.all;

package body bit_overload is
    -- This function returns a bit_vector sum of two bit_vector operands.
    -- The add is performed bit by bit with an internal carry
    function "+" (Add1, Add2: bit_vector) return bit_vector is
        variable sum: bit_vector(Add1'length-1 downto 0);
        variable c: bit := '0'; -- no carry in
        alias n1: bit_vector(Add1'length-1 downto 0) is Add1;
        alias n2: bit_vector(Add2'length-1 downto 0) is Add2;
        begin
            for i in sum.reverse_range loop
                sum(i) := n1(i) xor n2(i) xor c;
                c := (n1(i) and n2(i)) or (n1(i) and c) or (n2(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 (Add1 + int2vec(Add2, Add1'length));
    end "+";
end bit_overload;
use WORK.fourpack.all;
entity t_buff_exmpl is
    port (a,b,c,d : in X01Z; -- signals are
        f: out X01Z); -- 4 valued
end t_buff_exmpl;

architecture t_buff_conc of t_buff_exmpl is
begin
    f <= a when b = '1' else 'Z';
    f <= c when d = '1' else 'Z';
end t_buff_conc;

architecture t_buff_bhv of t_buff_exmpl is
begin
    buff1: process (a,b)
    begin
        if (b='1') then f<=a;
        else
            f<='Z';    --"drive" the output high Z when not enabled
        end if;
    end process buff1;

    buff2: process (c,d)
    begin
        if (d='1') then f<=c;
        else
            f<='Z';    --"drive" the output high Z when not enabled
        end if;
    end process buff2;
end t_buff_bhv;
Figure 8-7  Resolution of Signal Drivers

<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>'0'</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>
Figure 8-8  Resolution Function for X01Z Logic

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 resolve4 (s:u_x01z_vector) return u_x01z;
  subtype x01z is resolve4 u_x01z;
  -- x01z is a resolved subtype which uses the resolution function resolve4
  type x01z_vector is array (natural range <>) of x01z;
end fourpack;

package body fourpack is
  type x01z_table is array (u_x01z,u_x01z) of u_x01z;
  constant resolution_table : x01z_table := (
    ('X','X','X','X'),
    ('X','0','X','0'),
    ('X','X','1','1'),
    ('X','0','1','Z'));
  function resolve4 (s:u_x01z_vector) return u_x01z is
    variable result : u_x01z := 'Z';
    begin
      if (s'length = 1) then
        return s(s'low);
      else
        for i in s'range loop
          result := resolution_table(result,s(i));
        end loop;
      end if;
      return result;
    end resolve4;
end fourpack;
Table 8-4. Resolution Function Table for IEEE 9-Valued Logic

CONSTANT resolution_table : stdlogic_table := (  
  -- | U X 0 1 Z W L H -  
  -- |--------------------------  
  ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U  
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X  
  ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0  
  ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1  
  ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z  
  ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W  
  ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L  
  ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H  
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- | -  
);

IEEE-1164 Standard Logic

'U' Uninitialized             'X' Forcing Unknown             '0' Forcing 0
'1' Forcing 1                 'Z' High Impedance               'W' Weak Unknown
'L' Weak 0                    'H' Weak 1                       '–' Don't care
Table 8-5  And Table for IEEE 9-valued Logic

CONSTANT and_table : stdlogic_table := (  
-- | U  X  0  1  Z  W  L  H  - |
-- |--------------------------------|
  ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
  ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
  ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
  ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
  ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - | );

Table 8-5  And Table for IEEE 9-valued Logic

CONSTANT and_table : stdlogic_table := (  
<table>
<thead>
<tr>
<th>U  X  0  1  Z  W  L  H  -</th>
</tr>
</thead>
</table>
  ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
  ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | 0 |
  ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | 1 |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | Z |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | W |
  ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ), -- | L |
  ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- | H |
  ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - | );
Figure 8-9 And Function for std_logic_vectors

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 l'LENGTH ) is l;
  alias rv : std_logic_vector ( 1 to r'LENGTH ) is r;
  variable result : std_logic_vector ( 1 to l'LENGTH );
begin
  if ( l'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";
Figure 8-10 Function to Determine a PLA Output

```vhdl
type PLAmtrx is array (integer range <>, integer range <>) of std_logic;
function PLAout (PLA: PLAmtrx; Input: std_logic_vector)
  return std_logic_vector is
  alias In1: std_logic_vector(Input'length-1 downto 0) is Input;
  variable match: std_logic;  variable PLAcol, step: integer;
  variable PLArow: std_logic_vector(PLA'length(2)-1 downto 0);
  variable PLAinp: std_logic_vector(Input'length-1 downto 0);
  variable Output:
    std_logic_vector((PLA'length(2)-Input'length-1) downto 0);
begin
  Output := (others=>'0');    -- Initialize output to all zeros
  if PLA'left(2) > PLA'right(2) then step := -1; else step := 1; end if;
  LP1: for row in PLA'range loop
    match := '1';    -- Assume match for now
    PLAcol := PLA'left(2);
    LP2: for col in PLArow'range loop
      PLArow(col) := PLA(row,PLAcol);  PLAcol := PLAcol + step;
    end loop LP2;
    PLAinp := PLArow(PLArow'high downto PLArow'high-Input'length+1);
    LP3: for col in In1'range loop
      if IN1(col) /= PLAinp(col) and PLAinp(col) /= 'X' then
        match := '0';  exit;    -- mismatched row
      end if;
    end loop LP3;
  end loop LP1;
  if (match = '1') then Output := Output or PLArow(Output'range); end if;
end loop PLAout
end
```
Figure 8-11  Test of PLAout Function

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

library mvllib;
use mvllib.mvl_pack.all;

entity Platest is
    port (ABC: in std_logic_vector(2 downto 0);
         F: out std_logic_vector(3 downto 0));
end Platest;

architecture PLA1 of PLAtest is
constant PLA3_2: PLAmtrx (0 to 4, 6 downto 0) :=
    ("00X1010", "1X01100", "X1X0101", "X100010", "1X10001");
bEGIN
    F<= PLAout (PLA3_2, ABC);
end PLA1;
```
Figure 8-12  Rise/Fall Time Modeling Using Generic Statement

entity NAND2 is
  generic (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 nand b;
  c <= nand_value after (Trise + 3 ns * load) when nand_value = '1'
  else nand_value after (Tfall + 2 ns * load);
end behavior;

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;
Figure 8-13  Adder4 Using Generate Statement

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;
  Co <= C(4);
end Structure;
**Figure 8-14  Example of Unintentional Latch Creation**

```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 => null;
    end case;
  end process;
end test1;
```

(a) VHDL code that infers a latch  
(b) Synthesized Network
Figure 8-15(a & b) Synthesis of a Case Statement

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;
Figure 8-15(b & c) Synthesized Case Statement Before & After Optimization
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 = '0' then Z <= D;
        else Z <= E;
        end if;
    end process;
end test1;
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" + "11011" = "01000"
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:

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".
Figure 8-17  VHDL Code Example for Synthesis

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;
**Figure 8-18(a) VHDL Code for Synthesis of State Machine**

```vhdl
entity SM1_2 is
    port(X, CLK: in bit; Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
    subtype s_type is integer range 0 to 7;
    signal State, Nextstate: s_type;
    constant S0: s_type := 0; -- state assignment
    constant S1: s_type := 4;
    constant S2: s_type := 5;
    constant S3: s_type := 7;
    constant S4: s_type := 6;
    constant S5: s_type := 3;
    constant s6: s_type := 2;
begin
    process(State,X) -- Combinational Network
    begin
        Z <= '0'; Nextstate <= S0; -- added to avoid latch
    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 =>
        Z<='0'; Nextstate<=S0; -- added to avoid latch
    end case;
end process;
end Table;
```

23
Figure 8-18(b) VHDL Code for Synthesis of State Machine

```
when S3 =>
  if X='0' then Z<='0'; Nextstate<=S5;
  else Z<='1'; Nextstate<=S5; end if;
when S4 =>
  if X='0' then Z<='1'; Nextstate<=S5;
  else Z<='0'; Nextstate<=S6; end if;
when S5 =>
  if X='0' then Z<='0'; Nextstate<=S0;
  else Z<='1'; Nextstate<=S0; end if;
when S6 =>
  if X='0' then Z<='1'; Nextstate<=S0; end if;
when others => null;
end case;
end process;

process(CLK) -- State Register
begin
  if CLK='1' and CLK'event then -- rising edge of clock
    State <= Nextstate;
  end if;
end process;
end Table;
```
Figure 8-19  State Machine Synthesized From Figure 8-18
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity FMUL is
  port (CLK, St: in std_logic;
   F1,E1,F2,E2: in UNSIGNED(3 downto 0);
   F: out UNSIGNED(6 downto 0);
   V, done: out std_logic);
end FMUL;

architecture FMULB of FMUL is

signal A, B, C: UNSIGNED(3 downto 0); -- fraction registers
signal X, Y: UNSIGNED(4 downto 0); -- exponent registers
signal Load, Adx, SM8, RSF, LSF: std_logic;
signal AdSh, Sh, Cm, Mdone: std_logic;
signal PS1, NS1: integer range 0 to 3; -- present and next state
signal State, Nextstate: integer range 0 to 4; -- multiplier control state

begin
main_control: process(PS1,St,Mdone,X,A,B)
begin
  Load <= '0'; Adx <= '0'; NS1 <= 0; -- clear control signals
  SM8 <= '0'; RSF <= '0'; LSF <= '0'; V <= '0'; F <= "0000000";
  done <= '0';
  case PS1 is
  when 0 => F <= "0000000"; -- clear outputs
    done<='0'; V <='0';
    if St = '1' then Load <= '1'; NS1 <= 1; end if;
  when 1 => Adx <= '1'; NS1 <= 2;
  end case;
end process main_control;

end FMULB;
Figure 8-20(b) Revised VHDL Code for Floating-Point Multiplier

```vhdl
when 2 =>
  if Mdone = '1' then -- wait for multiply
    if A = 0 then SM8 <= '1'; -- zero fraction
    elsif A = 4 and B = 0 then RSF <= '1'; -- shift AB right
    elsif A(2) = A(1) then LSF <= '1'; -- test for unnormalized/ shift AB left
    end if;
  NS1 <= 3;
  end if;
when 3 => -- test for exp overflow
  if X(4) /= X(3) then V <= '1'; else V <= '0'; end if;
  done <= '1';
  F <= A(2 downto 0) & B; -- output fraction
  if ST = '0' then NS1 <= 0; end if;
end case;
end process main_control;

mul2c: process(State,Adx,B) -- 2's complement multiply
begin
  AdSh <= '0'; Sh <= '0'; Cm <= '0'; Mdone <= '0'; -- clear control signals
  Nextstate <= 0;
  case State is
    when 0=> -- start multiply
      if Adx='1' then
        if B(0) = '1' then AdSh <= '1'; else Sh <= '1'; end if;
        Nextstate <= 1;
      end if;
    when 1 | 2 => -- add/shift state
      if B(0) = '1' then AdSh <= '1'; else Sh <= '1'; end if;
      Nextstate <= State + 1;
  end case;
end process;
```

Figure 8-20(c) Revised VHDL Code for Floating-Point Multiplier

when 3 =>
  if B(0) = '1' then Cm <= '1'; AdSh <= '1'; else Sh <= '1'; end if;
  Nextstate <= 4;
when 4 => Mdone <= '1'; Nextstate <= 0;
end case;
end process mul2c;

update: process  -- update registers
variable addout: UNSIGNED(3 downto 0);
begin
  wait until (CLK = '1' and CLK'event); PS1 <= NS1; State <= Nextstate;
  if Cm = '0' then addout := A + C; else addout := A - C; end if;  -- add 2's comp. of C
  if Load = '1' then X <= E1(3)&E1; Y <= E2(3)&E2;
    A <= "0000"; B <= F1; C <= F2; end if;
  if ADX = '1' then X <= X + Y; end if;
  if SM8 = '1' then X <= "11000"; end if;
  if RSF = '1' then A <= '0'&A(3 downto 1); B <= A(0)&B(3 downto 1);
    X <= X + 1; end if;  -- increment X
  if LSF = '1' then
    A <= A(2 downto 0)&B(3); B <= B(2 downto 0)&'0';
    X <= X + 31; end if;  -- decrement X
  if AdSh = '1' then
    A <= (C(3) xor Cm) & addout(3 downto 1); -- load shifted adder
    B <= addout(0) & B(3 downto 1); end if;  -- output into A & B
  if Sh = '1' then
    A <= A(3) & A(3 downto 1);  -- right shift A & B
    B <= A(0) & B(3 downto 1);  -- with sign extend
  end if;
end process update;
end FMULB;
8.10 Files and TEXTIO

File Declaration

`file file-name:  file-type [open mode] is "file-pathname";`

Example:

`file test_data:  text open read_mode is "c:\test1\test.dat"

 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

File Types

A file can contain only one type of object as specified by the file type.

Example:

`type bv_file is file of bit_vector;

 defines bv_file to be a file type which can only contain bit_vectors`

Endfile Function

`endfile(file_name)

 returns TRUE if the file pointer is at the end of the file`
Standard TEXTIO Package

- contains declarations and procedures for working with files composed of lines of text.
- defines a file type named text:

  ```
  type text is file of string;
  ```

- contains procedures for reading lines of text from a file of type text and for writing lines of text to a file.

Procedure readline 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. The 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.

To extract data from the line buffer, call a read 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.
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, 10, 11, 12, 13, 14, 15);
  file infile: text open read_mode is "mem1.txt";
  -- 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;
Figure 8-21(b) VHDL Code to Fill a Memory Array from a File

begin
  while (not endfile(infile)) loop
    readline (infile, buff);
    read (buff, addr_s); -- read addr hexnum
    read(buff, byte_cnt); -- 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 fillmem;
package TEXTIO is
  -- Type Definitions for Text I/O
  type LINE is access STRING;  -- a LINE is a pointer to a STRING value
  type TEXT is file of STRING;  -- a file of variable-length ASCII records
  type SIDE is (RIGHT, LEFT);  -- for justifying output data w/in fields
  subtype WIDTH is NATURAL;    -- for specifying widths of output fields

-- Standard Text Files
  file  INPUT: TEXT open read_mode is "STD_INPUT";
  file  OUTPUT: TEXT open write_mode is "STD_OUTPUT";

-- Input Routines for Standard Types
  procedure READLINE (file F: TEXT; L: out LINE);
  procedure READ (L: inout LINE; VALUE: out BIT; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out BIT);
  procedure READ (L: inout LINE; VALUE: out BIT_VECTOR; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out BIT_VECTOR);
  procedure READ (L: inout LINE; VALUE: out BOOLEAN; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out CHAR; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out CHARACTER);
  procedure READ (L: inout LINE; VALUE: out INTEGER; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out INTEGER);
  procedure READ (L: inout LINE; VALUE: out REAL; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out REAL);
  procedure READ (L: inout LINE; VALUE: out STRING; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out STRING);
  procedure READ (L: inout LINE; VALUE: out TIME; GOOD: out BOOLEAN);
  procedure READ (L: inout LINE; VALUE: out TIME);
Appendix C  TEXTIO Package

-- Output Routines for Standard Types

procedure WRITELINE (file F: TEXT; L: inout LINE);
procedure WRITE (L: inout LINE; VALUE: in BIT;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in BIT_VECTOR;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in BOOLEAN;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in CHARACTER;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in INTEGER;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in REAL;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
    DIGITS: in NATURAL:= 0);
procedure WRITE (L: inout LINE; VALUE: in STRING;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0);
procedure WRITE (L: inout LINE; VALUE: in TIME;
    JUSTIFIED: in SIDE:= RIGHT; FIELD: in WIDTH := 0;
    UNIT: in TIME:= ns);

end package TEXTIO;