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

\[ A \leq B \implies B \text{ changes at time } T \]

- **Event**
  - occurs on a signal every time it is __________
- **Transaction**
  - occurs on a signal every time it is __________
- **Example:**

<table>
<thead>
<tr>
<th></th>
<th>A’event</th>
<th>B’event</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T + 1d</td>
<td></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'(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
8.1 Attributes - Attribute Test

**VHDL and Waveforms**

```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'transient;
  A_stable5 <= A'stable(5 ns);
  A_quiet5 <= A'quiet(5 ns);
end test;
```

8.1 Attributes - Using Attributes for Error Checking

```vhdl
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;
```
### Assert Statement

- **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

<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>ROMI/LEFT(1) = 0 ROMI/LEFT(2) = 7</td>
</tr>
<tr>
<td>A/RIGHT(N)</td>
<td>right bound of Nth index range</td>
<td>ROMI/RIGHT(1) = 15 ROMI/RIGHT(2) = 0</td>
</tr>
<tr>
<td>A/HIGH(N)</td>
<td>largest bound of Nth index range</td>
<td>ROMI/HIGH(1) = 15 ROMI/HIGH(2) = 7</td>
</tr>
<tr>
<td>A/LOW(N)</td>
<td>smallest bound of Nth index range</td>
<td>ROMI/LOW(1) = 0 ROMI/LOW(2) = 0</td>
</tr>
<tr>
<td>A/RANGE(N)</td>
<td>Nth index range</td>
<td>ROMI/RANGE(1) = 0 to 15 ROMI/RANGE(2) = 7 downto 0</td>
</tr>
<tr>
<td>A/REVERSE_RANGE(N)</td>
<td>Nth index range reversed</td>
<td>ROMI/REVERSE_RANGE(1) = .15 downto 0 ROMI/REVERSE_RANGE(2) = 0 to 7</td>
</tr>
<tr>
<td>A/LENGTH(N)</td>
<td>size of Nth index range</td>
<td>ROMI/LENGTH(1) = 16 ROMI/LENGTH(2) = 8</td>
</tr>
</tbody>
</table>
Chapter 8

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;
 C_in: in bit;
 signal Sum: out bit_vector;
 signal Cout: out bit;
 n: in positive) is
 variable C: bit;

begin
 C := C_in;
 for i in 0 to n-1 loop
 Sum(i) <= Add1(i) xor Add2(i) xor C;
 C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add1(i) and C);
 end loop;
 Cout <= C;
 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.

Chapter 8

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;
 C_in: in bit;
 signal Sum: out bit_vector;
 signal Cout: out bit) is
 variable C: bit := C_in;
 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 = 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;

Electrical and Computer Engineering
### 8.2 Transport and Inertial Delays

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

\[
\begin{align*}
Z_m & \leq X \text{ after } 4 \text{ ns} ; \\
Z_3 & \leq \text{transport } Z_m \text{ after } 6 \text{ ns} ; \\
\end{align*}
\]

### 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

```
-- This package provides two overloaded functions for the plus operator
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(length=Add1.length downto 0);
    -- no carry in
    alias n1: bit_vector(length=Add1.length downto 0) is Add1;
    alias n2: bit_vector(length=Add2.length 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;
```

8.3 Operator Overloading - Overloading Procedures and Functions

- A, B, C – bit vectors
- A <= B + C + 3 ?
- A <= 3 + B + C ?

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

use WORK.fourvalued.all;
entity t_buff_exmpl is
  port (a,b,c,d : in K912); -- signals are
  f : out K912); -- a 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
  buf1: 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 buf1;
  buf2: 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 buf2;
end t_buff_bhv;
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 ____________________

8.3 Multivalued Logic and Signal Resolution - Resolution Function

```
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;
```

<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>'1'</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>

Diagram: [Diagram of signal resolution function]
8.3 Multivalued Logic and Signal Resolution - Resolution Function VHDL

```vhdl
package Hughpack is
  type u_x01z is ('U','0','1','Z'); -- u_x01z is unresolved
  type u_x01z_vector is array (natural range <>) of u_x01z;
  function resolve4 (lcsu_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 package;

package body Hughpack is
  type x01z_table is array (u_x01z,u_x01z) 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 resolve4 (lcsu_x01z_vector) return u_x01z is
  variable result : u_x01z := 'Z';
  begin
    if (length = 1) then
      return s(low); end if;
  loop
    for i in range loop
      result := resolution_table(result,i(i));
    end loop;
  return result;
end resolve4;
end package;
```

8.5 IEEE-1164 Standard Logic

- 9-valued logic system
  - '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

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 | 0 | 1 | Z | W | L | H |
---|---|---|---|---|---|---|---|---|
{'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U'}, -- U |
{'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X'}, -- X |
{'U', 'X', '0', 'X', '0', 'X', '0', 'X'}, -- 0 |
{'U', 'X', '1', 'X', '1', 'X', '1', 'X'}, -- 1 |
{'U', 'X', '0', '0', '0', 'X', '0', '0'}, -- Z |
{'U', 'X', '0', '1', '1', 'X', '1', '1'}, -- W |
{'U', 'X', '0', 'L', 'L', 'X', 'L', 'L'}, -- L |
{'U', 'X', '0', 'H', 'H', 'X', 'H', 'H'}, -- H |
{'U', 'X', '0', 'X', 'X', 'X', 'X', 'X'}, -- - |
```

### 8.5 IEEE-1164 Standard Logic - AND Definition

**CONSTANT and_table : stdlogic_table :=**

```
-- | U | X | 0 | 1 | Z | W | L | H |
---|---|---|---|---|---|---|---|---|
{'U', 'U', '0', 'U', 'U', 'U', 'U', 'U'}, -- U |
{'U', 'X', '0', 'X', '0', 'X', '0', 'X'}, -- X |
{'U', 'X', '0', '0', '0', 'X', '0', '0'}, -- 0 |
{'U', 'X', '0', '1', '1', 'X', '1', '1'}, -- 1 |
{'U', 'X', '0', 'L', 'L', 'X', 'L', 'L'}, -- Z |
{'U', 'X', '0', 'H', 'H', 'X', 'H', 'H'}, -- W |
{'U', 'X', '0', 'X', 'X', 'X', 'X', 'X'}, -- H |
{'U', 'X', '0', 'X', 'X', 'X', 'X', 'X'}, -- - |
```

8.5 IEEE-1164 Standard Logic - AND Function VHDL

```vhdl
function "and" ( l : std_logic; r : std_logic ) 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 '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
  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 and b;
  < <= 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

```vhdl
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);
beg
  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

```vhdl
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);
beg
  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;
```

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
  
  \[ A \leq B \text{ and } C; \]
  
  wait until clk'event and clk = '1';
  
  \[ A \leq B \text{ and } C; \]

  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 $a = 3$.

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

if $A = '1'$ then $\text{NextState} \leq 3$
end if;

What if $A \neq 1$?
Retain the previous value for $\text{NextState}$?
Synthesizer might interpret this to mean that $\text{NextState}$ is unknown!

if $A = '1'$ then $\text{NextState} \leq 3$
else $\text{NextState} \leq 2$
end if;
8.8 Synthesis of VHDL Code - 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.

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

- 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

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 := 4;
  constant S2 : s_type := 5;
  constant S3 : s_type := 7;
  constant S4 : s_type := 6;
  constant S5 : s_type := 0;
  constant S6 : s_type := 2;
begin
  process(State, X)
  begin
    Z <= '0'; Nextstate <= S0; -- added to avoid latch
  end process;
  process(CLK)
  begin
    if CLK='1' and CLK'event then
      State <= Nextstate;
    end if;
  end process;
begin
  begin
    process(State, X)
    begin
      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<= '1'; Nextstate<=S4; else Z<= '1'; Nextstate<=S5; end if;
      when others => null;
    end case;
  end process;
  begin
    process(CLK)
    begin
      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
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

Standard TEXTIO Package
• Contains __________ and __________ for working with ____ composed of lines of text
• Defines a file type named 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

- **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.
- Code
  ```
  variable buff: line;
  ...
  readline (test_data, 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
  - bytes1 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, -1, -1, -1, 10, 11, 12, 13, 14, 15
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use std.textio.all;

entity testfill is
end testfill;

architecture RAMType is
  type RAMType is array (0 to 8191) of std_logic_vector(7 downto 0);
  signal rem: RAMType := (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, -1, -1, -1, -1);

file infl: text open read_mode is "mem1.txt"; -- open file for reading
  file infl: text in file "mem1.txt" is text "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(infl)) loop
    readline (infl, buff);
    read (buff, addr_s); -- read hex number
    read (buff, byte_cnt); -- read number of bytes to read
    addr1 := lookup(addr_s(3))*16 + lookup(addr_s(2));
    addr1 := addr1 + 1;
    for i in 1 to byte_cnt loop
      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 testbench;
end testfill;