CPE/EE 422/522
Advanced Logic Design
L14

Electrical and Computer Engineering
University of Alabama in Huntsville

Additional Topics in VHDL

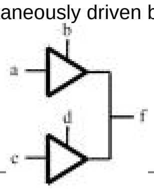
- Attributes
- Transport and Inertial Delays
- Operator Overloading
- Multivalued Logic and Signal Resolution
- IEEE 1164 Standard Logic
- Generics
- Generate Statements
- Synthesis of VHDL Code
- Synthesis Examples
- Files and Text IO

Review: 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

Review: Multivalued Logic

- 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'

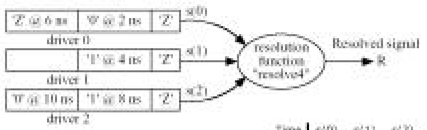


Review: Signal Resolution

- VHDL signals may either be resolved or unresolved
- Resolved signals have an associated resolution function
- 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 generate an error

Review: Signal Resolution (cont’d)

```
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;
```



	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'X'	'X'
'0'	'X'	'0'	'X'	'0'
'1'	'X'	'X'	'1'	'1'
'Z'	'X'	'0'	'1'	'Z'

Time	s(0)	s(1)	s(2)	R
0	'Z'	'Z'	'Z'	'Z'
2	'0'	'Z'	'Z'	'0'
4	'0'	'1'	'Z'	'X'
6	'Z'	'1'	'Z'	'1'
8	'Z'	'1'	'1'	'1'
10	'Z'	'1'	'0'	'X'

Review: Resolution Function for X01Z

```
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;
```

Define AND and OR for 4- valued inputs?

IEEE 1164 Standard Logic

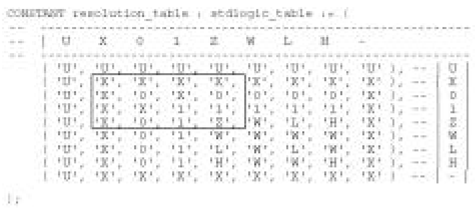
- 9-valued logic system
  - ‘U’ – Uninitialized
  - ‘X’ – Forcing Unknown
  - ‘0’ – Forcing 0
  - ‘1’ – Forcing 1
  - ‘Z’ – High impedance
  - ‘W’ – Weak unknown
  - ‘L’ – Weak 0
  - ‘H’ – Weak 1
  - ‘-’ – Don’t care

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

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

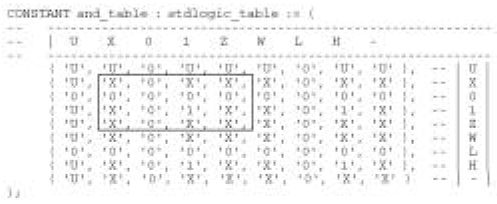
In this course we use a subset of the IEEE values: X10Z

Resolution Function for IEEE 9-valued



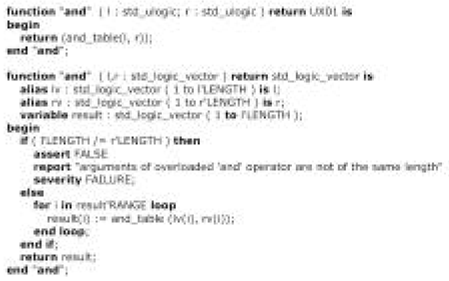
```
CONSTANT resolution_table : stdlogic_table := (
--  ---------------------------------------------------------
--  |  U    X    0    1    Z    W    L    H    -
--  ---------------------------------------------------------
    ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
    ( 'U', 'X',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

AND Table for IEEE 9-valued



```
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' )   -- | - |
);
```

AND Function for std_logic_vectors

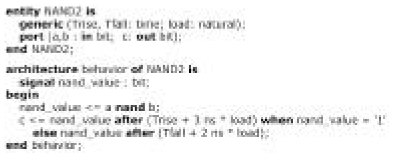


```
function "and" ( l : std_ulogic; r : std_ulogic ) return UX01 is
begin
   return (and_table(l, r));
end "and";

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";
```

Generics

- Used to specify parameters for a component in such a way that the parameter values must be specified when the component is instantiated
- Example: rise/fall time modeling



```
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;
```

Rise/Fall Time Modeling Using Generics

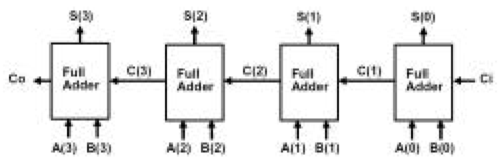
```
entity NAND2 is
   generic (Trise, Tfall: time; load: natural);
   port (a,b : in bit;  c: out bit);
end NAND2;

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;
```

Generate Statements

- Provides an easy way of instantiating components when we have an iterative array of identical components
- Example: 4-bit RCA



4-bit Adder

```
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;
```

4-bit Adder using Generate

```
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;
```

Synthesis of VHDL Code

- Synthesizer
  - take a VHDL code as an input
  - synthesize the logic: output may be a logic schematic with an associated wirelist
- Synthesizers accept a subset of VHDL as input
- Efficient implementation?
- Context

```
                  ...
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

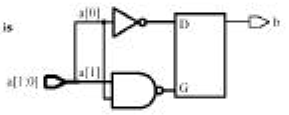
Synthesis of VHDL Code (cont'd)

- When use integers specify the range
  - 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
- General rule: when a signal is assigned a value, it will hold that value until it is assigned new value

Unintentional Latch Creation

```
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;
```



What if a = 3?

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

To eliminate latch => replace the word null with b <= 0;

If Statements

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

What if A /= 1?

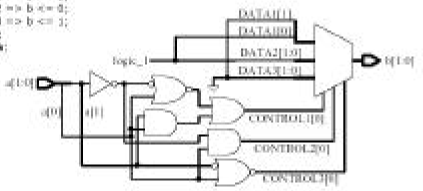
Retain the previous value for NextState?

Synthesizer might interpret this to mean that NextState is unknown!

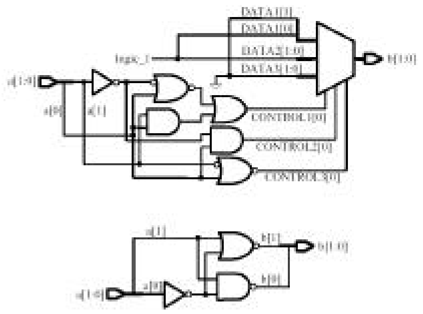
```
if A = '1' then NextState <= 3;
else  NextState <= 2;
end if;
```

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;
```



Case Statement: Before and After Optimization

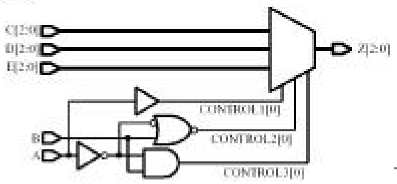


Synthesis of an If Statement

```
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 = '0' then Z <= D;
      else Z <= E;
      end if;
   end process;
end test1;
```

Synthesized code before optimization



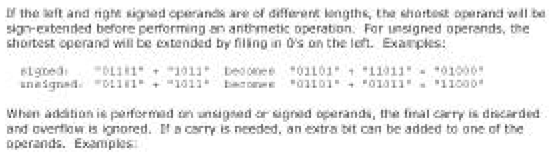
Standard VHDL Synthesis Package

- 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

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

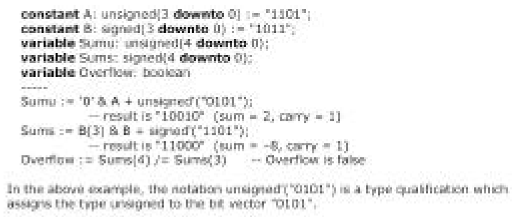
Numeric_bit, Numeric_std (cont'd)


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"
```

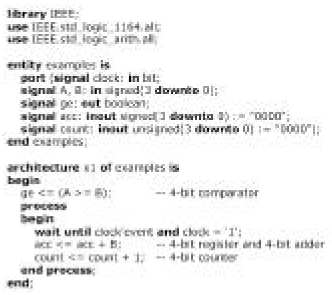
When addition is performed on unsigned or signed operands, the final carry is discarded and overflow is ignored. If a carry is needed, an extra bit can be added to one of the operands. Examples:

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".

Synthesis Examples (1)

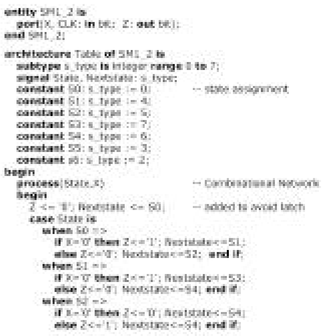

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

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;
```

Synthesis Examples (2a)

- Mealy machine: BCD to BCD+3 Converter

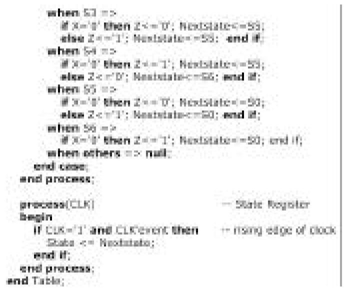


```
entity SM1_2 is
   port(X, CLK: in bit; Z: out bit);
end SM1_2;

architecture Table of SM1_2 is
   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;
```

Synthesis Examples (2b)

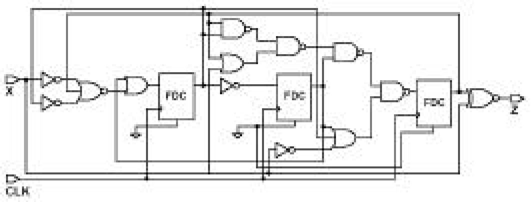
- Mealy machine: BCD to BCD+3 Converter



```
       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;
```

Synthesis Examples (2c)

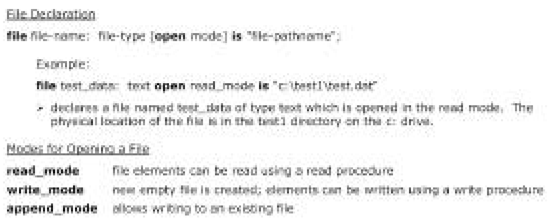


3 FF, 13 gates

Files

- File input/output in VHDL
- Used in test benches
  - Source of test data
  - Storage for test results
- VHDL provides a standard TEXTIO package
  - read/write lines of text

Files



<u>File Declaration</u>

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.

<u>Modes for Opening a File</u>

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 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

Reading TEXTIO file

- <u>Readline</u> reads a line of text and places it in a buffer with an associated pointer
- 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

Extracting Data from the Line Buffer

- 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.

Extracting Data from the Line Buffer (cont'd)

- TEXTIO provides overloaded *read* procedures to read data of types bit, bit_vector, boolean, character, integer, real, string, and time from buffer
- Read forms

```
read(pointer, value)
read(pointer, value, good)
```

  – good is boolean that returns TRUE if the read is successful 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

Writing to TEXTIO files

- Call one or more write procedures to write data to a line buffer and then call writeline 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)

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

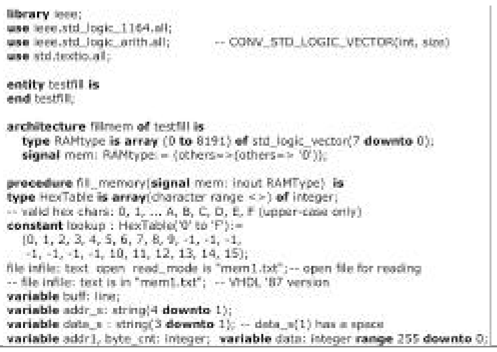
An Example (cont'd)

- 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

VHDL Code to Fill Memory Array



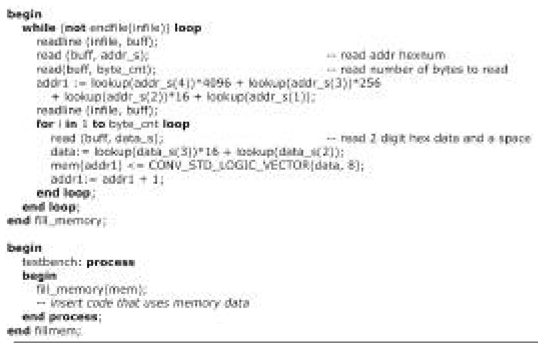
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;          -- CONV_STD_LOGIC_VECTOR(int, size)
use std.textio.all;

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";-- 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;
```

VHDL Code to Fill Memory Array (cont'd)



```
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;
```

Things to Remember

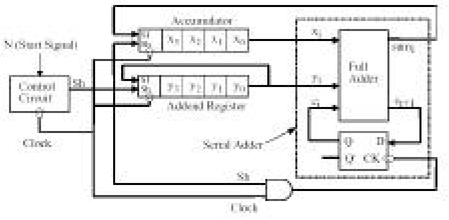
- Attributes associated to signals
  - allow checking for setup, hold times, and other timing specifications
- Attributes associated to arrays
  - allow us to write procedures that do not depend on the manner in which arrays are indexed
- Inertial and transport delays
  - allow modeling of different delay types that occur in real systems
- Operator overloading
  - allow us to extend the definition of VHDL operators so that they can be used with different types of operands

Things to Remember (cont'd)

- Multivalued logic and the associated resolution functions
  - allow us to model tri -state buses, and systems where a signal is driven by more than one source
- Generics
  - allow us to specify parameter values for a component when the component is instantiated
- Generate statements
  - efficient way to describe systems with iterative structure
- TEXTIO
  - convenient way for file input/output

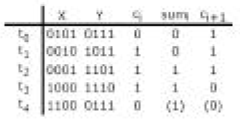
Networks for Arithmetic Operations

Case Study: Serial Adder with Accumulator

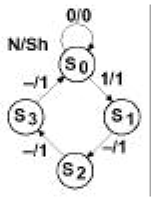


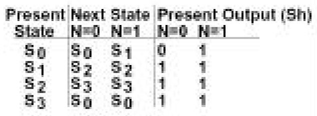
Networks for Arithmetic Operations

Serial Adder with Accumulator



	X	Y	c_i	sum_i	c_{i+1}
t_0	0101	0111	0	0	1
t_1	0010	1011	1	0	1
t_2	0001	1101	1	1	1
t_3	1000	1110	1	1	0
t_4	1100	0111	0	(1)	(0)





Present State	Next State		Present Output (Sh)	
	N=0	N=1	N=0	N=1
S_0	S_0	S_1	0	1
S_1	S_2	S_2	1	1
S_2	S_3	S_3	1	1
S_3	S_0	S_0	1	1

State Graphs for Control Networks

- Use variable names instead of 0s and 1s
  - E.g., XiXj/ZpZq
    - if Xi and Xj inputs are 1, the outputs Zp and Zq are 1 (all other outputs are 0s)
  - E.g., X = X1X2X3X4, Z = Z1Z2Z3Z4
    - X1X4'/Z2Z3 == 1 - - 0 / 0 1 1 0

Constraints on Input Labels

- Assume: I – input expression => we traverse the arc when I=1

1. If I_i and I_j are any pair of input labels on arcs exiting state S_k, then $I_iI_j = 0$ if $i \neq j$.

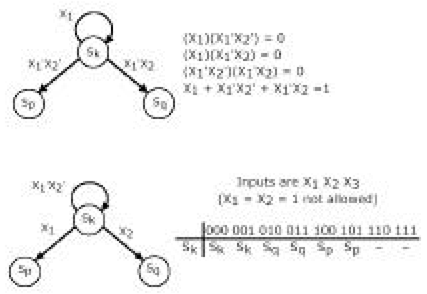
Assures that at most one input label can be 1 at any given time

2. If n arcs exit state S_k and the n arcs have input labels $I_1, I_2, \ldots, I_n$, respectively, then $I_1 + I_2 + \ldots + I_n = 1$.

Assures that at least one input label will be 1 at any given time

1 + 2: Exactly one label will be 1 =>
the next state will be uniquely defined for every input combination

Constraints on Input Labels (cont'd)



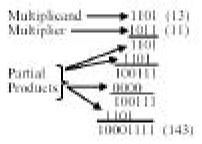
$(x_1)(x_1'x_2') = 0$
$(x_1)(x_1'x_2) = 0$
$(x_1'x_2')(x_1'x_2) = 0$
$x_1 + x_1'x_2' + x_1'x_2 = 1$

Inputs are $x_1\ x_2\ x_3$
($x_1 = x_2 = 1$ not allowed)

	000	001	010	011	100	101	110	111
S_k	S_k	S_k	S_q	S_q	S_p	S_p	–	–

Networks for Arithmetic Operations

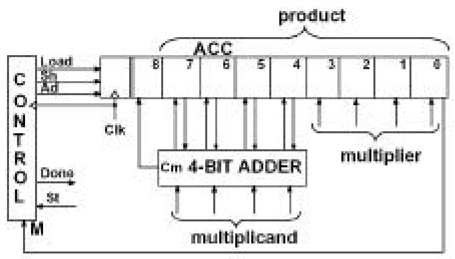
Case Study: Serial Parallel Multiplier



```
Multiplicand ──→ 1101  (13)
Multiplier   ──→ 1011  (11)
                 1101
                1101
               100111
               0000
              100111
             1101
            10001111  (143)
```

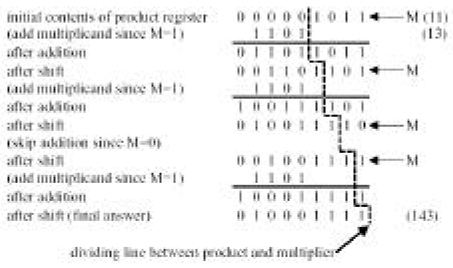
Note: we use unsigned binary numbers

Block Diagram of a Binary Multiplier



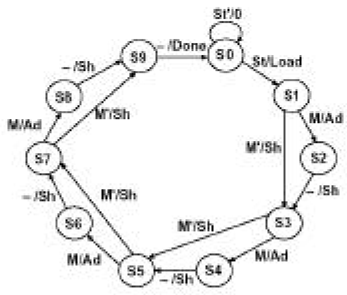
Ad – add signal // adder outputs are stored into the ACC
Sh – shift signal // shift all 9 bits to right
Ld – load signal // load multiplier into the 4 lower bits of the ACC and clear the upper 5 bits

Multiplication Example

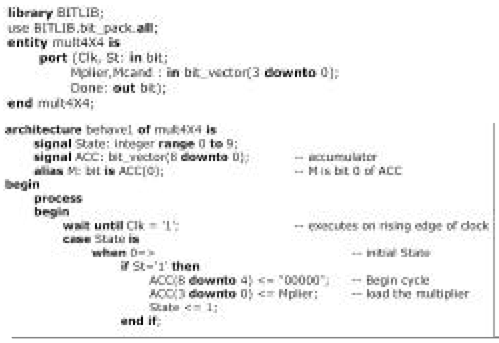


```
initial contents of product register   0 0 0 0 0|1 0 1 1 ←── M (11)
(add multiplicand since M=1)             1 1 0 1               (13)
after addition                         0 1 1 0 1|1 0 1 1
after shift                            0 0 1 1 0 1|1 0 1 ←── M
(add multiplicand since M=1)             1 1 0 1
after addition                         1 0 0 1 1 1|1 0 1
after shift                            0 1 0 0 1 1 1|1 0 ←── M
(skip addition since M=0)
after shift                            0 0 1 0 0 1 1 1|1 ←── M
(add multiplicand since M=1)             1 1 0 1
after addition                         1 0 0 0 1 1 1 1|1
after shift (final answer)             0 1 0 0 0 1 1 1 1      (143)
```

State Graph for Binary Multiplier



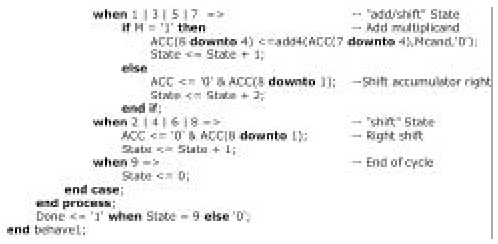
Behavioral VHDL Model



```
library BITLIB;
use BITLIB.bit_pack.all;
entity mult4X4 is
     port (Clk, St: in bit;
          Mplier,Mcand : in bit_vector(3 downto 0);
          Done: out bit);
end mult4X4;

architecture behave1 of mult4X4 is
     signal State: integer range 0 to 9;
     signal ACC: bit_vector(8 downto 0);            -- accumulator
     alias M: bit is ACC(0);                        -- M is bit 0 of ACC
begin
     process
     begin
          wait until Clk = '1';                     -- executes on rising edge of clock
          case State is
               when 0=>                                        -- initial State
                    if St='1' then
                         ACC(8 downto 4) <= "00000";          -- Begin cycle
                         ACC(3 downto 0) <= Mplier;           -- load the multiplier
                         State <= 1;
                    end if;
```

Behavioral VHDL Model (cont'd)



```
               when 1 | 3 | 5 | 7  =>                          -- "add/shift" State
                    if M = '1' then                            -- Add multiplicand
                         ACC(8 downto 4) <=add4(ACC(7 downto 4),Mcand,'0');
                         State <= State + 1;
                    else
                         ACC <= '0' & ACC(8 downto 1);         --Shift accumulator right
                         State <= State + 2;
                    end if;
               when 2 | 4 | 6 | 8 =>                           -- "shift" State
                    ACC <= '0' & ACC(8 downto 1);              -- Right shift
                    State <= State + 1;
               when 9 =>                                       -- End of cycle
                    State <= 0;
          end case;
     end process;
     Done <= '1' when State = 9 else '0';
end behave1;
```