CPE 626
Advanced VLSI Design
Lecture 4: VHDL Recapitulation (Part 2)

Aleksandar Milenkovic

http://www.ece.uah.edu/~milenka
http://www.ece.uah.edu/~milenka/cpe626-04F/
milenka@ece.uah.edu

Assistant Professor
Electrical and Computer Engineering Dept.
University of Alabama in Huntsville

Outline

- *Introduction to VHDL*
- *Modeling of Combinational Networks*
- *Modeling of FFs*
- *Delays*
- *Modeling of FSMs*
- *Wait Statements*
- VHDL Data Types
- VHDL Operators
- Functions, Procedures, Packages

Variables

- What are they for:
  Local storage in processes,
  procedures, and functions
- Declaring variables

```
variable list_of_variable_names : type_name
[ := initial value ];
```

Variables must be declared within the process in which they are used and are local to the process

Note: exception to this is SHARED variables

Signals

- Signals must be declared outside a process
- Declaration form

```
signal list_of_signal_names : type_name
[ := initial value ];
```

- Declared in an architecture can be used anywhere within that architecture

Constants

- Declaration form

```
constant constant_name : type_name := constant_value;

constant delay1 : time := 5 ns;
```

- Constants declared at the start of an architecture can be used anywhere within that architecture
- Constants declared within a process are local to that process

Variables vs. Signals

- Variable assignment statements
  - expression is evaluated and the variable is instantaneously updated (no delay, not even delta delay)

```
variable_name := expression;
```

- Signal assignment statement

```
signal_name <= expression [after delay];
```

  – expression is evaluated and the signal is scheduled to change after delay; if no delay is specified the signal is scheduled to be updated after a delta delay

Variables vs. Signals (cont'd)

Process Using Variables

```
entity dummy is
end dummy;

architecture var of dummy is
    signal trigger, sum: integer:=0;
begin
    process
    variable var1: integer:=1;
    variable var2: integer:=2;
    variable var3: integer:=3;
    begin
        wait on trigger;
        var1 := var2 + var3;
        var2 := var1;
        var3 := var2;
        sum <= var1 + var2 + var3;
    end process;
end var;
```

Sum = ?

Process Using Signals

```
entity dummy is
end dummy;

architecture sig of dummy is
    signal trigger, sum: integer:=0;
    signal sig1: integer:=1;
    signal sig2: integer:=2;
    signal sig3: integer:=3;
begin
    process
    begin
      wait on trigger;
      sig1 <= sig2 + sig3;
      sig2 <= sig1;
      sig3 <= sig2;
      sum <= sig1 + sig2 + sig3;
    end process;
end sig;
```

Sum = ?

Predefined VHDL Types

- Variables, signals, and constants can have any one of the predefined VHDL types or they can have a user-defined type
- Predefined Types
  - bit – {'0', '1'}
  - boolean – {TRUE, FALSE}
  - integer – [-2^{31} - 1.. 2^{31} – 1}
  - real – floating point number in range –1.0E38 to +1.0E38
  - character – legal VHDL characters including lower-uppercase letters, digits, special characters, ...
  - time – an integer with units fs, ps, ns, us, ms, sec, min, or hr

User Defined Type

- Common user-defined type is *enumerated*

```
type state_type is (S0, S1, S2, S3, S4, S5);
signal state : state_type := S1;
```

- If no initialization, the default initialization is the leftmost element in the enumeration list (S0 in this example)

- VHDL is strongly typed language =>
signals and variables of different types cannot be mixed in the same assignment statement,
and no automatic type conversion is performed

Arrays

- Example

```
type SHORT_WORD is array (15 downto 0) of bit;
signal DATA_WORD : SHORT_WORD;
variable ALT_WORD : SHORT_WORD := "0101010101010101";
constant ONE_WORD : SHORT_WORD := (others => '1');
```

- ALT_WORD(0) – rightmost bit
- ALT_WORD(5 downto 0) – low order 6 bits

- General form

```
type arrayTypeName is array index_range of element_type;
signal arrayName : arrayTypeName [:=InitialValues];
```

Arrays (cont'd)

- Multidimensional arrays

```
type matrix4x3 is array (1 to 4, 1 to 3) of integer;
variable matrixA: matrix4x3 :=
((1,2,3), (4,5,6), (7,8,9), (10,11,12));
```

- matrixA(3, 2) = ?

- Unconstrained array type

```
type intvec is array (natural range<>) of integer;

type matrix is array (natural range<>,natural range<>)
of integer;
```

- range must be specified when the array object is declared

```
signal intvec5 : intvec(1 to 5) := (3,2,6,8,1);
```

Sequential Machine Model Using State Table

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

architecture Table of SM1_2 is
   type StateTable is array (integer range <>, bit range <>) of integer;
   type OutTable is array (integer range <>, bit range <>) of bit;
   signal State, NextState: integer := 0;
   constant ST: StateTable (0 to 6, '0' to '1') :=
      ((1,2), (3,4), (4,4), (5,5), (5,6), (0,0), (0,0));
   constant OT: OutTable (0 to 6, '0' to '1') :=
      (('1','0'), ('1','0'), ('0','1'), ('0','1'), ('1','0'), ('0','1'), ('1','0'));
begin                                   -- concurrent statements
   NextState <= ST(State,X);            -- read next state from state table
   Z <= OT(State, X);                   -- read output from output table
   process(CLK)
   begin
      if CLK = '1' then                 -- rising edge of CLK
         State <= NextState;
      end if;
   end process;
end Table;
```

	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

Predefined Unconstrained Array Types

- Bit_vector, string

```
type bit_vector is array (natural range <>) of bit;
type string is array (positive range <>) of character;

 constant string1: string(1 to 29) := "This string is 29 characters."
constant A : bit_vector(0 to 5) := "10101";
-- ('1', '0', '1', '0', '1');
```

- Subtypes
  - include a subset of the values specified by the type

```
subtype SHORT_WORD is : bit_vector(15 to 0);
```

- POSITIVE, NATURAL – predefined subtypes of type integer

VHDL Operators

- Binary logical operators: and or nand nor xor xnor
- Relational: = /= < <= > >=
- Shift: sll srl sla sra rol ror
- Adding: + - & (concatenation)
- Unary sign: + -
- Multiplying: * / mod rem
- Miscellaneous: not abs **

- Class 7 has the highest precedence (applied first), followed by class 6, then class 5, etc

Example of VHDL Operators

In the following expression, A, B, C, and D are bit_vectors:

```
    (A & not B or C ror 2 and D) = "110010"
```

The operators would be applied in the order:

```
    not, &, ror, or, and, =
```

If A = "110", B = "111", C = "011000", and D = "111011", the computation would proceed as follows:

```
    not B = "000"  (bit-by-bit complement)
    A & not B = "110000"   (concatenation)
    C ror 2 = "000110"   (rotate right 2 places)
    (A & not B) or (C ror 2) = "110110   (bit-by-bit or)
    (A & not B or C ror 2) and D = "110010"  (bit-by-bit and)
    [(A & not B or C ror 2 and D) = "110010"] = TRUE
      (the parentheses force the equality test to be done last and the result is TRUE)
```

Example of Shift Operators (cont'd)

The shift operators can be applied to any bit_vector or boolean_vector. In the following examples, A is a bit_vector equal to "10010101":

```
A sll 2  is  "01010100" (shift left logical, filled with '0')
A srl 3 is  "00010010" (shift right logical, filled with '0')
A sla 3 is  "10101111" (shift left arithmetic, filled with right bit)
A sra 2is  "11100101" (shift right arithmetic, filled with left bit)
A rol 3 is  "10101100" (rotate left)
A ror 5 is  "10101100" (rotate right)
```

VHDL Functions

- Functions execute a sequential algorithm and return a single value to calling program

```
function rotate_right (reg: bit_vector)
        return bit_vector is
begin
        return reg ror 1;
end rotate_right;
```

- A = "10010101"

```
B <= rotate_right(A);
```

- General form

```
function function-name (formal-parameter-list)
        return return-type is
        [declarations]
begin
        sequential statements -- must include return return-value;
end function-name;
```

For Loops

General form of a for loop:

```
    [loop-label:] for loop-index in range loop
          sequential statements
    end loop [loop-label];
```

Exit statement has the form:

```
    exit;                   -- or
    exit when condition;
```

For Loop Example:

```
-- compare two 8-character strings and return TRUE if equal
function comp_string(string1, string2: string(1 to 8))
      return boolean is

variable B: boolean;
begin
      loopex: for j in 1 to 8 loop
            B := string1(j) = string2(j);
            exit when B=FALSE;
      end loop loopex;
      return B;
end comp_string;
```

Add Function

```
-- This function adds 2 4-bit vectors and a carry.
-- It returns a 5-bit sum

function add4 (A,B: bit_vector(3 downto 0); carry: bit)
   return bit_vector is

variable cout: bit;
variable cin: bit := carry;
variable Sum: bit_vector(4 downto 0):="00000";
begin
loop1: for i in 0 to 3 loop
   cout := (A(i) and B(i)) or (A(i) and cin) or (B(i) and cin);
   Sum(i) := A(i) xor B(i) xor cin;
   cin := cout;
end loop loop1;
Sum(4):= cout;
return Sum;
end add4;
```

Example function call:

```
Sum1 <= add4(A1, B1, cin);
```

VHDL Procedures

- Facilitate decomposition of VHDL code into modules
- Procedures can return any number of values using output parameters

```
procedure procedure_name (formal-parameter-list) is
 [declarations]
 begin
      Sequential-statements
 end procedure_name;

 procedure_name (actual-parameter-list);
```

Procedure for Adding Bit_vectors

```
-- 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;
      C := (Add1(i) and Add2(i)) or (Add1(i) and C) or (Add2(i) and C);
   end loop;
   Cout <= C;
end Addvec;
```

Example procedure call:

```
Addvec(A1, B1, Cin, Sum1, Cout, 4);
```

Parameters for Subprogram Calls

		Actual Parameter	
Mode	Class	Procedure Call	Function Call
in[1]	constant[2]	expression	expression
	signal	signal	signal
	variable	variable	n/a
out/inout	signal	signal	n/a
	variable[3]	variable	n/a

[1] default mode for functions [2] default for in mode [3] default for out/inout mode

Packages and Libraries

- Provide a convenient way of referencing frequently used functions and components
- Package declaration

```
package package-name is
   package declarations
end [package][package-name];
```

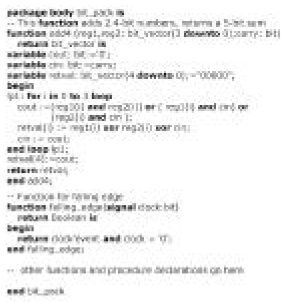
- Package body [optional]

```
package body package-name is
   package body declarations
end [package body][package name];
```

Library BITLIB – bit_pack package

```
package bit_pack is
  function add4 (reg1,reg2: bit_vector(3 downto 0);carry: bit)
     return bit_vector;
  function falling_edge(signal clock:bit)
     return Boolean ;
  function rising_edge(signal clock:bit)
     return Boolean ;
  function vec2int(vec1: bit_vector)
     return integer;
  function int2vec(int1,NBits: integer)
     return bit_vector;
  procedure Addvec
     (Add1,Add2: in bit_vector;
        Cin: in bit;
        signal Sum: out bit_vector;
        signal Cout: out bit;
        n: in natural);

  component jkff
     generic(DELAY:time := 10 ns);
     port(SN, RN, J,K,CLK: in bit; Q, QN: inout bit);
  end component;

  component dff
     generic(DELAY:time := 10 ns);
     port (D, CLK: in bit; Q: out bit; QN: out bit := '1');
  end component;

  component and2
     generic(DELAY:time := 10 ns);
     port(A1, A2: in bit; Z: out bit);
  end component;

  component and3
     generic(DELAY:time := 10 ns);
     port(A1, A2, A3: in bit; Z: out bit);
  end component;

  component and4
     generic(DELAY:time := 10 ns);
     port(A1, A2, A3, A4: in bit; Z: out bit);
  end component;

  component or2
     generic(DELAY:time := 10 ns);
     port(A1, A2: in bit; Z: out bit);
  end component;

  component or3
     generic(DELAY:time := 10 ns);
     port(A1, A2, A3: in bit; Z: out bit);
  end component;
--
  (other component declarations go here)
--

end bit_pack;
```

Library BITLIB – bit_pack package



```
package body bit_pack is
-- This function adds 2 4-bit numbers, returns a 5-bit sum
function add4 (reg1,reg2: bit_vector(3 downto 0);carry: bit)
   return bit_vector is
variable cout: bit:='0';
variable cin: bit:=carry;
variable retval: bit_vector(4 downto 0):="00000";
begin
lp1: for i in 0 to 3 loop
   cout := (reg1(i) and reg2(i)) or ( reg1(i) and cin) or
              (reg2(i) and cin );
   retval(i) := reg1(i) xor reg2(i) xor cin;
   cin := cout;
end loop lp1;
retval(4):=cout;
return retval;
end add4;

-- Function for falling edge
function falling_edge(signal clock:bit)
   return Boolean is
begin
   return clock'event and clock = '0';
end falling_edge;

-- other functions and procedure declarations go here

end bit_pack
```

CPE 626: Advanced VLSI Design VHDL Recap (Part II)

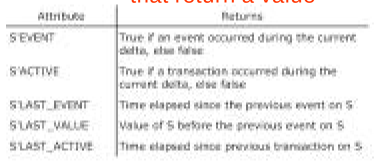
Department of Electrical and
Computer Engineering
University of Alabama in Huntsville

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

Signal Attributes

Attributes associated with signals that return a value



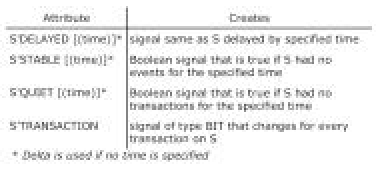
Attribute	Returns
S'EVENT	True if an event occurred during the current delta, else false
S'ACTIVE	True if a transaction occurred during the current delta, else false
S'LAST_EVENT	Time elapsed since the previous event on S
S'LAST_VALUE	Value of S before the previous event on S
S'LAST_ACTIVE	Time elapsed since previous transaction on S

A'event – true if a change in S has just occurred

A'active – true if A has just been reevaluated, even if A does not change

Review: Signal Attributes (cont'd)

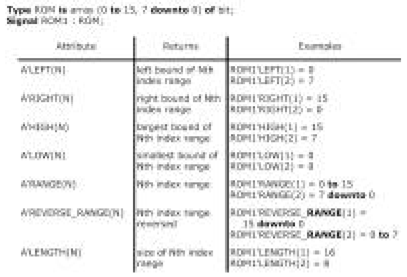
Attributes that create a signal



Attribute	Creates
S'DELAYED [(time)]*	signal same as S delayed by specified time
S'STABLE [(time)]*	Boolean signal that is true if S had no events for the specified time
S'QUIET [(time)]*	Boolean signal that is true if S had no transactions for the specified time
S'TRANSACTION	signal of type BIT that changes for every transaction on S

* *Delta is used if no time is specified*

Array Attributes



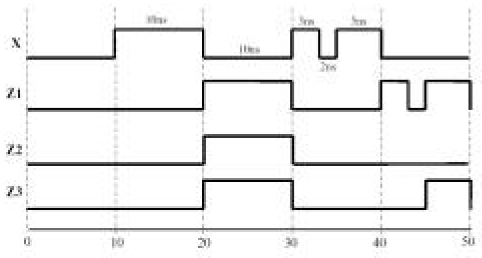
Type ROM **is** array (0 **to** 15, 7 **downto** 0) **of** bit;
Signal ROM1 : ROM;

Attribute	Returns	Examples
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'RANGE(N)	Nth index range	ROM1'RANGE(1) = 0 **to** 15 ROM1'RANGE(2) = 7 **downto** 0
A'REVERSE_RANGE(N)	Nth index range reversed	ROM1'REVERSE_**RANGE**(1) = 15 **downto** 0 ROM1'REVERSE_**RANGE**(2) = 0 to 7
A'LENGTH(N)	size of Nth index range	ROM1'LENGTH(1) = 16 ROM1'LENGTH(2) = 8

A can be either an array name or an array type.

Array attributes work with signals, variables, and constants.

Transport and Inertial Delay



```
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
```

Review: Operator Overloading

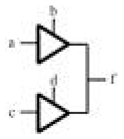
- Operators +, - operate on integers
- Write procedures for bit vector addition/subtraction
  - addvec, subvec
- Operator overloading allows using + operator to implicitly call an appropriate addition function
- How does it work?
  - When compiler encounters a function declaration in which the function 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

VHDL Package with Overloaded Operators

```
-- 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(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;
```

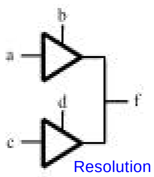
Multivalued Logic

- 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'



Tristate Buffers

```
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;
```



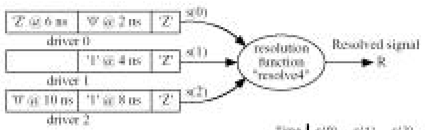
Resolution function to determine the actual value of f since it is driven from two different sources

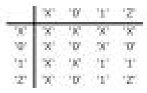
Signal Resolution

- 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

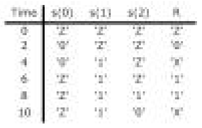
Signal Resolution (cont'd)

```
signal R : X01Z := 'Z'; ...
R <= transport '0' after 2 ns, 'Z' after 6
  ns;
```



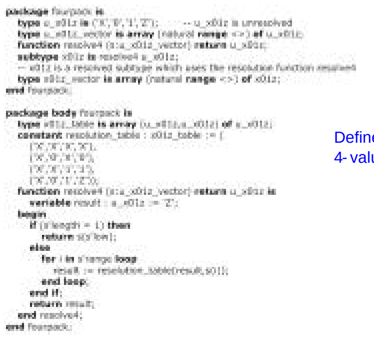


	'X'	'0'	'1'	'Z'
'X'	'X'	'X'	'X'	'X'
'0'	'X'	'0'	'X'	'0'
'1'	'X'	'X'	'1'	'1'
'Z'	'X'	'0'	'1'	'Z'



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'

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?

AND and OR Functions Using X01Z

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

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

IEEE 1164 Standard Logic

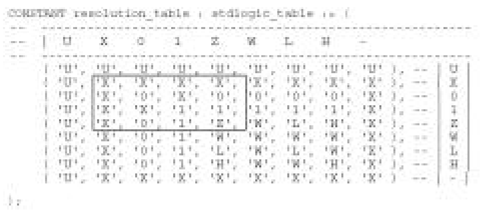
- 9-valued logic system
  - 'U' – Uninitialized
  - 'X' – Forcing Unknown
  - '0' – Forcing 0
  - '1' – Forcing 1
  - 'Z' – High impedance
  - 'W' – Weak unknown
  - 'L' – Weak 0
  - 'H' – Weak 1
  - '-' – Don't care

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

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

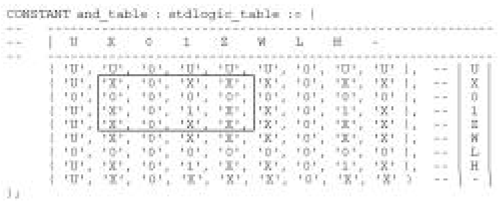
In this course we use a subset of the IEEE values: X10Z

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

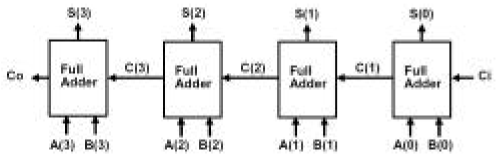
```
entity NAND2 is
  generic (Trise, Tfall: time; load: natural);
  port (a,b : in bit;  c: out bit);
end NAND2;

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;
```

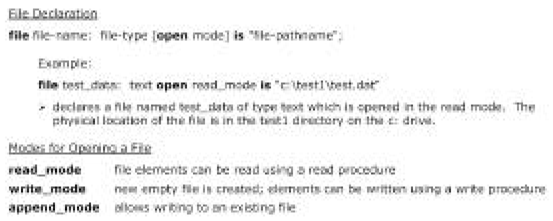
4-bit Adder using Generate

```
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;
```

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



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 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

- Readline 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 ar e 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)

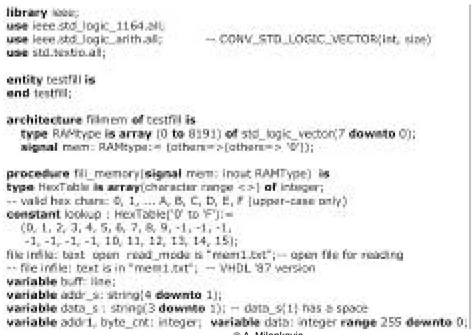
An Example

- 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)

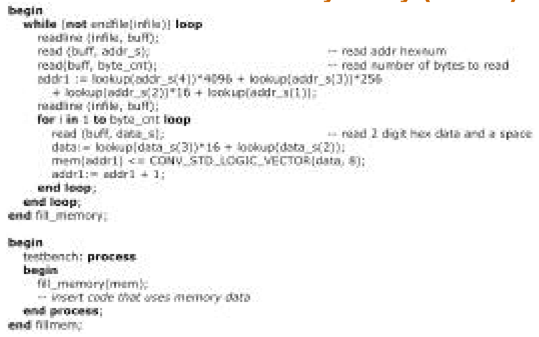
- Code sequence: an example
  ```
  12AC 7 (7 hex bytes follow)
  AE 03 B6 91 C7 00 0C (LDX imm, LDA dir, STA ext)
  005B 2 (2 bytes follow)
  01 FC_
  ```
- TEXTIO does not include read procedure for hex numbers
  - we will read each hex value as a string of characters and then convert the string to 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;
```

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;
```

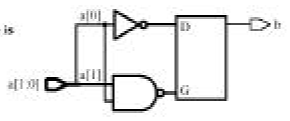
Implies CM for A Implies a register or flip-flop

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.

If Statements

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

What if A /= 1?

Retain the previous value for NextState?

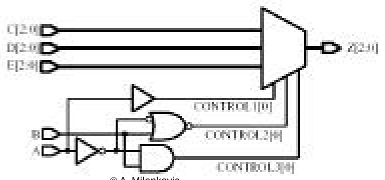
Synthesizer might interpret this to mean that NextState is unknown!

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

Synthesis of an If Statement

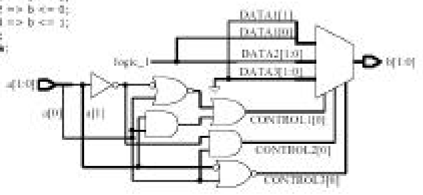
```
entity if_example is
   port(A,B: in bit;
      C,D,E: in bit_vector(2 downto 0);
      Z: out bit_vector(2 downto 0));
end if_example;

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

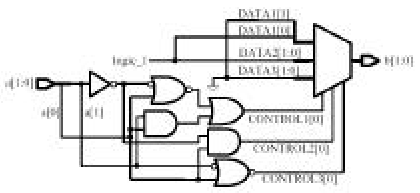


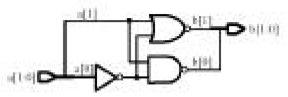
Synthesis of a Case Statement

```
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





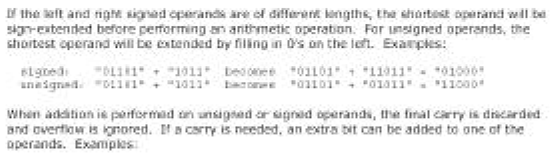
Standard VHDL Synthesis Package

- 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

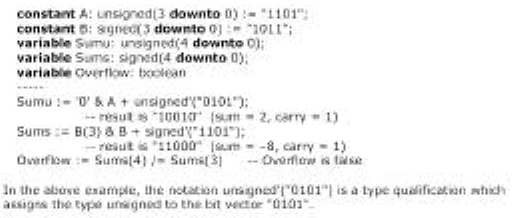
Numeric_bit, Numeric_std (cont'd)


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"
```

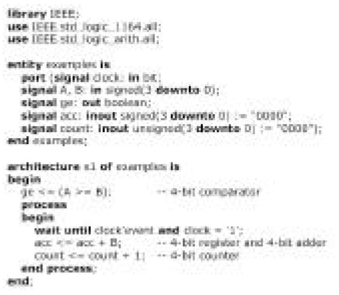
When addition is performed on unsigned or signed operands, the final carry is discarded and overflow is ignored. If a carry is needed, an extra bit can be added to one of the operands. Examples:

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
```

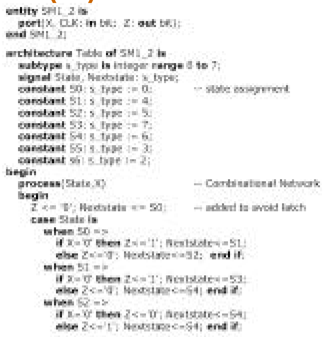
In the above example, the notation unsigned'("0101") is a type qualification which assigns the type unsigned to the bit vector "0101".

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;
```

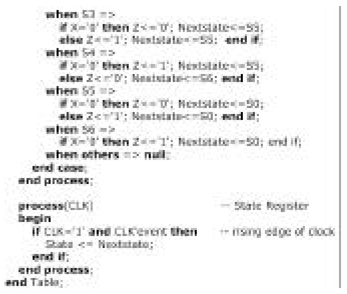
Synthesis Examples (2a)

- 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;
```

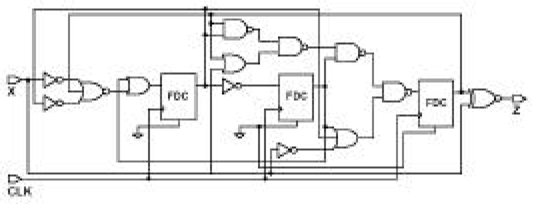
Synthesis Examples (2b)

- 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

Writing Test Benches

- MUX 16 to 1
  - 16 data inputs
  - 4 selection inputs

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
entity SELECTOR is
   port(
         A: in std_logic_vector(15 downto 0);
         SEL: in std_logic_vector(3 downto 0);
         Y: out std_logic);
end SELECTOR;


architecture RTL of SELECTOR is
begin
   Y <= A(conv_integer(SEL));
end RTL;
```

Assert Statement

- Checks to see if a certain condition is true, and if not causes an error message to be displayed

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

- Four possible severity levels
  - NOTE
  - WARNING
  - ERROR
  - FAILURE
- Action taken for a severity level depends on the simulator

Writing Test Benches

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity TBSELECTOR is
end TBSELECTOR;

architecture BEH of TBSELECTOR is
   component SELECTOR
   port(
         A: in std_logic_vector(15 downto 0);
         SEL: in std_logic_vector(3 downto 0);
         Y: out std_logic);
   end component;
   signal TA : std_logic_vector(15 downto 0);
   signal TSEL : std_logic_vector(3 downto 0);
   signal TY, Y : std_logic;
   constant PERIOD : time := 50 ns;
   constant STROBE : time := 45 ns;
```

Writing Test Benches

```
begin
P0: process
   variable cnt : std_logic_vector(4 downto 0);
   begin
    for j in 0 to 31 loop
       cnt := conv_std_logic_vector(j, 5);
       TSEL <= cnt(3 downto 0);
       Y <= cnt(4);
       A <= (A'range => not cnt(4));
       A(conv_integer(cnt(3 downto 0))) <= cnt(4);
       wait for PERIOD;
    end loop;
    wait;
   end process;
```

Writing Test Benches

```
begin
check: process
   variable err_cnt : integer := 0;
   begin
     wait for STROBE;
     for j in 0 to 31 loop
          assert FALSE report “comparing” severity NOTE;
          if (Y /= TY) then
               assert FALSE report “not compared” severity WARNING;
               err_cnt := err_cnt + 1;
          end if;
          wait for PERIOD;
     end loop;
     assert (err_cnt = 0) report “test failed” severity ERROR;
     assert (err_cnt /= 0) report “test passed” severity NOTE;
     wait;
   end process;
   sel1: SELECTOR port map (A => TA, SEL = TSEL, Y => TY);
end BEH;
```

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