Variables

- What are they for:
  - Local storage in processes, procedures, and functions
- Declaring variables
  ```vhdl
  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
  ```vhdl
  signal list_of_signal_names : type_name
  [ := initial value ];
  ```

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

Variables vs. Signals

- Variable assignment statements
  ```vhdl
  expression is evaluated and the variable is instantaneously updated (no delay, not even delta delay)
  variable_name := expression;
  ```

- Signal assignment statement
  ```vhdl
  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

Constants

- Declaration form
  ```vhdl
  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 (cont’d)

Process Using Variables

```vhdl
entity dummy is
  architecture seq of dummy is
    signal Sum : integer := 0;
  end sequence;
end dummy;
```

Process Using Signals

```vhdl
entity dummy is
  architecture seq of dummy is
    signal Sum : integer := 0;
  end sequence;
end dummy;
```

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` – `{INT_MIN .. INT_MAX}`
  - `real` – floating point number in range `-1.0E38 to +1.0E38`
  - `character` – legal VHDL characters including lower- and 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`**
  ```vhdl
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**

```vhdl
type SHORT_WORD is array (15 downto 0) of bit;
signal DATA_WORD : SHORT_WORD;
signal 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**
  ```vhdl
type arrayTypeName is array index_range of element_type;
signal arrayName : arrayTypeName [:=InitialValues ];
```

Arrays (cont’d)

**Multidimensional arrays**

```vhdl
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**
  ```vhdl
type intvec is array (natural range<>) of integer;
signal intvec5 : intvec(1 to 5) := (3,2,6,8,1);
```

Sequential Machine Model Using State Table

**Example**

```vhdl
entity STM is
  port (X, clocks, reset);
  end STM;
architecture Table of STM is
  type TState is array (natural range<>) of TData;
type TData is record
    -- state variables
    State: TState;
    -- output data
    PS, X, E: signal integer;
  end record;
  constant (T(Data)) of (0 to 7) :=
    ((1,1,0), (0,1,0,0), (1,0,1,1), (1,0,0,1,1), (1,1,1,0,1,1), (0,1,1,0,1,0), (1,0,0,1,0,0),
     (1,1,1,1,1,0,1));
begin
  process(CLK)
  begin
    if CLK = '1' then
      State := next_state; -- new state from state table
      PS := next_output;   -- new output from output table
    end if;
  end process;
end Table;
```
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**
  - constant A : bit vector(0 to 5) := "101011";
  - constant B : string(0 to 29) := "This string is 29 characters."
  - constant C : character := '1';
  - constant D : character := '0';

- **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:

\( A \) is not \( B \) and \( C \) or \( D \) = "110010".

The operators would be applied in the order:

not, \( \& \), or, \( \& \), and, -

If \( A = '110' \), \( B = '111' \), \( C = '01000' \), and \( D = '110111' \), the computation would proceed as follows:

not \( B \) = "000" (bit-by-bit complement);
\( A \) \& \( not B \) = "10000" (concatenation);
\( C \) \& \( not B \) = "000" (rotated right, 1 position);
\( A \) \& \( C \) or \( D \) = "110111" (bit-by-bit or);
\( A \) \& \( not B \) \& \( C \) or \( D \) = "100010" (bit-by-bit and);
\( A \) \& \( C \) or \( D \) = "100101" (bit-by-bit and);
\( A \) \& \( B \) or \( C \) and \( D \) = "100101" (true).

(Not all parentheses are shown for clarity. 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".

- **sll**: \( 00101010 \) (shift left logical, filled with 0)
- **srl**: \( 10101010 \) (shift right logical, filled with 0)
- **sla**: \( 10101010 \) (shift left arithmetic, filled with right bit)
- **sra**: \( 10101010 \) (shift right arithmetic, filled with left bit)
- **rol**: \( 01010101 \) (rotate left)
- **ror**: \( 10101010 \) (rotate right)

VHDL Functions

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

  ```vhdl
  function rotate, right (type bit, vector) return bit, vector is
  begin
  return rot, right;
end function rotate, right;

  A := "10010101";
  B := rotate, right(A);
  ```

- **General form**

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

For Loops

**General form of a for loop**

```vhdl
[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**

```vhdl
for i in 1 to 10 loop
  b := string(1); string(2);
  exit when B=FALSE;
end loop;
```

```vhdl
end for string;
```
Add Function

```vhdle
-- This function adds 2 4-bit vectors and a carry.
-- It returns a 5-bit sum
function add(AB, bit vector(3 downto 0); carry: bit)
return bit vector is
variable sum: bit; carry:
variable bit vector(4 downto 0):‘00000;
begins
loop for i in 0 to 3 loop
sum(i) := (AB(i) and R(i)) or (AB(i) and CN) or (R(i) and CN);
sum(i) := sum(i) xor R(i) xor CN;
end loop;
return sum;
end add;
```

Example function call:
```vhdle
Sum1 := add(A, B, Cin);
```

VHDL Procedures

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

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

Procedure for Adding Bit vectors

```vhdle
procedure Addvec
(A, B, C, Cin, Sum, Cout : bit);
```

Parameters for Subprogram Calls

<table>
<thead>
<tr>
<th>Mode</th>
<th>Class</th>
<th>Procedure Call</th>
<th>Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>in1</td>
<td>constant</td>
<td>expression</td>
<td>expression</td>
</tr>
<tr>
<td>out/in</td>
<td>signal</td>
<td>signal</td>
<td>signal</td>
</tr>
<tr>
<td>variable</td>
<td>variable</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

1. default mode for functions
2. default for in mode
3. default for out/in mode

Packages and Libraries

- Provide a convenient way of referencing frequently used functions and components
- Package declaration
  ```vhdle
  package package-name is
  [package declarations]
  end package;[
- Package body [optional]
  ```vhdle
  package body package-name is
  [package body declarations]
  end package body;[

Library BITLIB – bit_pack package

```vhdle
package bit_pack is
function add (req ?[1:0] bit vector(3 downto 0); carry: bit)
returns (bit vector);[...
```
Library BITLIB – bit_pack package

- The function adds 2 4-bit values, returns a 5-bit sum.
- The function adds 2 4-bit values, returns a 6-bit sum.
- Example:
  ```vhdl
  library bit_lib;
  use bit_lib.bit_pack;
  
  signal a, b, c : bit_vector(7 downto 0);  -- 2 4-bit vectors
  signal s : bit_vector(11 downto 0);         -- 5-bit sum
  signal s' : bit_vector(12 downto 0);       -- 6-bit sum
  
  procedure add (i : in boolean; j, k : in bit_vector(7 downto 0);)
  begin
    s := a + b;  -- Add two 4-bit vectors
    s' := a + b + c;  -- Add two 4-bit vectors plus a third
  end procedure;
  
  begin
    add(true, a, b);  
    add(true, a, b, c);  
  end;
  ```

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

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

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

Array Attributes

Attributes that create a signal

- Attributes that create a signal

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Attributes</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>S'EVENT</td>
<td>Signal</td>
<td>S'EVENT =&gt; 5</td>
</tr>
<tr>
<td>S'ACTIVE</td>
<td>Signal</td>
<td>S'ACTIVE =&gt; 5</td>
</tr>
<tr>
<td>S'LAST_EVENT</td>
<td>Time</td>
<td>S'LAST_EVENT =&gt; 5</td>
</tr>
<tr>
<td>S'LAST_VALUE</td>
<td>Value</td>
<td>S'LAST_VALUE =&gt; 5</td>
</tr>
<tr>
<td>S'LAST_ACTIVE</td>
<td>Time</td>
<td>S'LAST_ACTIVE =&gt; 5</td>
</tr>
</tbody>
</table>

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

Array attributes work with signals, variables, and constants.
Review: Operator Overloading

- Operators +, - operate on integers
- Write procedures for bit vector addition/subtraction
- as vec, 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

```vhdl
library ieee;
package bit_vector_arithmetic is
  type bit_vector is array (natural range <>) of bit;
  function + (bv1, bv2 : bit_vector) return bit_vector;
end package;
```

```vhdl
package body bit_vector_arithmetic is
  function + (bv1, bv2 : bit_vector) return bit_vector is
    result : bit_vector := (others => '0');
    carry  : bit := '0';
    begin
      for i in result'range loop
        result(i) := bv1(i) xor bv2(i);
        if bv1(i) = bv2(i) then
          result(i) := '0';
        else
          result(i) := '1';
        end if;
      end loop;
      for i in 0..1 loop
        if result(i) = '1' then
          carry := '1';
        end if;
      end loop;
      result := addresult(result, carry);
    end function;
end package body;
```

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

```vhdl
architecture truth_table of buf is
  function f : bit_vector := (others => '0');
  begin
    process (a, b, c)
      variable x : bit_vector := (others => '0');
      variable y : bit_vector := (others => '0');
      variable z : bit_vector := (others => '0');
      begin
        if a = '1' then
          x := '1';
        else
          x := '0';
        end if;
        if b = '1' then
          y := '1';
        else
          y := '0';
        end if;
        if c = '1' then
          z := '1';
        else
          z := '0';
        end if;
      end process;
      f := x or y or z;
    begin
      process (f)
        variable b : bit_vector := (others => '0');
        begin
          if f = '1' then
            b := '1';
          else
            b := '0';
          end if;
        end process;
    end begin;
end architecture;
```

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;
R <= transport '1' after 4 ns;
R <= transport '1' after 8 ns, '0' after 10 ns;

Resolution Function for X01Z

```
package Per500 is
  type x01z_vector is array (natural range <>) of x01z;
  function resolve (x01z_vector return x01z) return x01z;
end Per500;
```

Resolution Function for IEEE 9-valued

```
package IEEE1164 is
  type x01z_vector is array (natural range <>) of x01z;
  constant resolution_table : x01z_table := ( ...
end IEEE1164;
```

AND and OR Functions Using X01Z

<table>
<thead>
<tr>
<th>AND</th>
<th>X</th>
<th>Y</th>
<th>T</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>'X'</td>
<td>'X'</td>
<td>0</td>
<td>'X'</td>
<td>'Z'</td>
</tr>
<tr>
<td>'X'</td>
<td>'0'</td>
<td>0</td>
<td>'X'</td>
<td>'X'</td>
</tr>
<tr>
<td>'0'</td>
<td>'X'</td>
<td>'0'</td>
<td>'X'</td>
<td>'X'</td>
</tr>
<tr>
<td>'X'</td>
<td>'X'</td>
<td>'1'</td>
<td>'X'</td>
<td>'1'</td>
</tr>
<tr>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>X</th>
<th>Y</th>
<th>T</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>'X'</td>
<td>'X'</td>
<td>0</td>
<td>'X'</td>
<td>'Z'</td>
</tr>
<tr>
<td>'X'</td>
<td>'0'</td>
<td>'X'</td>
<td>'X'</td>
<td>'Z'</td>
</tr>
<tr>
<td>'0'</td>
<td>'X'</td>
<td>0</td>
<td>'X'</td>
<td>'X'</td>
</tr>
<tr>
<td>'X'</td>
<td>'X'</td>
<td>'1'</td>
<td>'X'</td>
<td>'1'</td>
</tr>
<tr>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
<td>'X'</td>
</tr>
<tr>
<td>'Z'</td>
<td>'1'</td>
<td>'0'</td>
<td>'X'</td>
<td>'AND'</td>
</tr>
</tbody>
</table>

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

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

In this course we use a subset of the IEEE values: X10Z
AND Function for std_logic_vectors

```vhdl
function "and" (l : std_logic; r : std_logic) return std_logic is
begin
  return (l and r);
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

```vhdl
entity NAND2 is
  generic (THIGH: time; TLOW: time; load: natural);
  port (A, B: in bit; Y: out bit);
end NAND2;
architecture behavior of NAND2 is
  signal Y_int: bit;
begin
  Y_int <= not (A and B);
  Y <= Y_int after (THIGH + TLOW);  
end behavior;
```

Rise/Fall Time Modeling Using Generics

```vhdl
entity RCA is
  generic (time: natural);
  port (A: in bit; B: in bit; S: out bit);
end RCA;
architecture Generate of RCA is
begin
  process
    variable c: bit;  
  begin
    c := 0;
    for i in 0 to 4 loop
      c := (not c) and (A(i) or B(i));
      S(i) <= c after (2 * time);
    end loop;
  end process;
end Generate;
```

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

```vhdl
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); C: in bit; S: out bit_vector(3 downto 0); C0: out bit);
architecture Structure of Adder4 is
begin
  FA0: FullAdder port map (A(0), B(0), C, S(0), C0);
  FA1: FullAdder port map (A(1), B(1), C, S(1), C0);
  FA2: FullAdder port map (A(2), B(2), C, S(2), C0);
  FA3: FullAdder port map (A(3), B(3), C, S(3), C0);
end Structure;
```

4-bit Adder using Generate

```vhdl
entity Adder4 is
  port (A, B: in bit_vector(3 downto 0); C: in bit; S: out bit_vector(3 downto 0); C0: out bit);
architecture Structure of Adder4 is
begin
  signal C0: bit;  
  generate
    for i in 0 to 3 generate
      FA: FullAdder port map (A(i), B(i), C, C0, S(i), C0);
    end generate;
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

Standard TEXTIO Package
- Contains declarations and procedures for working with files composed of lines of text
- Defines a file type named text:
  ```vhdl
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:
  ```vhdl
type line is access string;
```
- When a variable of type line is declared, it creates a pointer to a string
- Code
  ```vhdl
  variable buff: line;
  ...
  readline (test_data, 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
  ```vhdl
  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.
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

An Example (cont’d)

- Code sequence: an example
  
  ```
  12AC 7 (7 hex bytes follow)
  AE 03 B6 01 C7 00 0C 02 (2 bytes follow)
  ```

- 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, 10, 11, 12, 13, 14, 15

VHDL Code to Fill Memory Array

- Code sequence: an example
  
  ```
  12AC 7 (7 hex bytes follow)
  AE 03 B6 01 C7 00 0C 02 (2 bytes follow)
  ```

- 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, 10, 11, 12, 13, 14, 15

VHDL Code to Fill Memory Array (cont’d)

- Code sequence: an example
  
  ```
  12AC 7 (7 hex bytes follow)
  AE 03 B6 01 C7 00 0C 02 (2 bytes follow)
  ```

- 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, 10, 11, 12, 13, 14, 15

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;
  ```

- 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

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

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

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

Synthesis of an If Statement

```vhdl
entity if_example is
  port(a, b: in std_logic);
  c: out integer range 0 to 3);
end if_example;
architecture test of if_example is
begin
  if a = '1' then c <= 3;
  else c <= 2;
end if;
```

Synthesis of a Case Statement

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

Case Statement: Before and After Optimization

Synthesized code before optimization

- `if a = '1' then c <= 3; else c <= 2; end if;`
- `case a is
  when 0 => c <= 0;
  when 1 => c <= 1;
  when 2 => c <= 2;
  when 3 => c <= 3;
  end case;
end test;`
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;
- package include overloaded versions of arithmetic, relational, logical, and shifting operations, and conversion functions.

numeric_std package defines similar operations on std_logic_vectors.

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, slt, srl, rol, ror

Numeric_bit, Numeric_std (cont'd)

If the left and right operand lengths are different, the shortest operand will be sign-extended before performing an arithmetic operation. For unsigned numbers, the shortest operand will be extended by filling a 0 on the left. Examples:

```
signed: "00110" + "00110" = "01100"
signed: "00110" - "00110" = "00000"
```

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:

```
```

Synthesis Examples (1)

```
```

Synthesis Examples (2a)

Mealy machine: BCD to BCD-3 Converter

```
```

Synthesis Examples (2b)

```
```

Synthesis Examples (3a)

```
```

Synthesis Examples (3b)

```
```

Synthesis Examples (4a)

```
```

Synthesis Examples (4b)

```
```

Synthesis Examples (5a)

```
```

Synthesis Examples (5b)

```
```
**Synthesis Examples (2b)**

Mealy machine:

BCD to BCD+3 Converter

```verbatim
when S2 =>
  if x = '0' then Y <= '0';
  else '1';
end if;
end when;
```

```
when S1 =>
  if x = '0' then Y <= '0';
  else '1';
end if;
end when;
```

```
when S0 =>
  if x = '0' then Y <= '0';
  else '1';
end if;
end when;
```

```
when S3 =>
  if x = '0' then Y <= '0';
  else '1';
end if;
end when;
```

```
when S2 =>
  if x = '0' then Y <= '0';
  else '1';
end if;
end when;
```

```
when S1 =>
  if x = '0' then Y <= '0';
  else '1';
end if;
end when;
```

```
when S0 =>
  if x = '0' then Y <= '0';
  else '1';
end if;
end when;
```

```
when S3 =>
  if x = '0' then Y <= '0';
  else '1';
end if;
end when;
```

```
when others => null;
end case;
```

## MUX 16 to 1

- 16 data inputs
- 4 selection inputs

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

entity TBSELECTOR is
end TBSELECTOR;
```

```verbatim
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;

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;
```

## Assert Statement

- Checks to see if a certain condition is true, and if not causes an error message to be displayed
- Four possible severity levels
  - NOTE
  - WARNING
  - ERROR
  - FAILURE

**NOTE**

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

**WARNING**

**ERROR**

**FAILURE**

Action taken for a severity level depends on the simulator.

**ERROR**

```verbatim
assert boolean-expression
report string-expression
severity ERROR;
```

**FAIL**

```verbatim
assert boolean-expression
report string-expression
severity FAILURE;
```

## Writing Test Benches

```verbatim
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);
ed 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