CPE 626
Advanced VLSI Design
Lecture 3: VHDL Recapitulation

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

Intro to VHDL
- Technology trends
  - 1 billion transistor chip running at 20 GHz in 2007
- Need for Hardware Description Languages
- Systems become more complex
- Design at the gate and flip-flop level becomes very tedious and time consuming
- HDLs allow
  - Design and debugging at a higher level before conversion to the gate and flip-flop level
  - Tools for synthesis do the conversion
- VHDL, Verilog
- VHDL – VHSIC Hardware Description Language

VHDL Description of Combinational Networks

Entity-Architecture Pair

Full Adder Example

entity FullAdder is
  port (X, Y, Cin: in bit; -- Inputs
  Cout, Sum: out bit; -- Outputs
end FullAdder;

architecture Equations of FullAdder is
begin
  -- Concurrent Assignments
  Sum <= X xor Y xor Cin after 10 ns;
  Cout <= (X and Y) or (X and Cin) or (Y and Cin) after 10 ns;
end Equations;
Whenever one of the signals in the sensitivity list changes, the sequential statements are executed in sequence one time.
JK Flip-Flop Model

```vhdl
entity JFF is
    port (S1, R, N, J, K, CLK, Q, Q'out): in bit;
    Q, new Q, Q': out bit ("1"); -- see Note 1
end JFF;
architecture JKFF of JFF is
begin
process (S1, R, N, CLK)
begin
if R = '1' then Q <= '1' after 10 ns; -- R=0 will clear the FF
elsif S1 = '1' then Q <= '1' after 10 ns; -- S1=0 will set the FF
elsif CLK = '1' and CLK event then
    Q <= Q and not Q or (not Q and Q) after 10 ns; -- see Note 2
    Q'out <= Q; -- see Note 3
end if;
end process;
end JKFF;
```

Concurrent Statements vs. Process

Simulation Results

<table>
<thead>
<tr>
<th>time</th>
<th>delta</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>+0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>+0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>+1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>+2</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>+3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Using Nested IFs

```vhdl
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 RTL1 of SELECTOR is
begin
p0 : process (A, SEL)
begin
if SEL = "0000" then Y <= A(0);
elsif SEL = "0001" then Y <= A(1);
elsif SEL = "0010" then Y <= A(2);
elsif SEL = "0011" then Y <= A(3);
elsif SEL = "0100" then Y <= A(4);
elsif SEL = "0101" then Y <= A(5);
elsif SEL = "0110" then Y <= A(6);
elsif SEL = "0111" then Y <= A(7);
elsif SEL = "1000" then Y <= A(8);
elsif SEL = "1001" then Y <= A(9);
elsif SEL = "1010" then Y <= A(10);
elsif SEL = "1011" then Y <= A(11);
elsif SEL = "1100" then Y <= A(12);
elsif SEL = "1101" then Y <= A(13);
elsif SEL = "1110" then Y <= A(14);
else Y <= A(15);
end if;
end process;
end RTL1;
```

VHDL Models for a MUX

```vhdl
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 RTL3 of SELECTOR is
begin
with SEL select
    Y <= A(0)  when "0000",
         A(1)  when "0001",
         A(2)  when "0010",
         A(3)  when "0011",
         A(4)  when "0100",
         A(5)  when "0101",
         A(6)  when "0110",
         A(7)  when "0111",
         A(8)  when "1000",
         A(9)  when "1001",
         A(10) when "1010",
         A(11) when "1011",
         A(12) when "1100",
         A(13) when "1101",
         A(14) when "1110",
         A(15) when others;
end with SEL;
end RTL3;
```

MUX Models (1)

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.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 RTL1 of SELECTOR is
begin
    Y <= A(0)  when SEL = "0000",
         A(1)  when SEL = "0001",
         A(2)  when SEL = "0010",
         A(3)  when SEL = "0011",
         A(4)  when SEL = "0100",
         A(5)  when SEL = "0101",
         A(6)  when SEL = "0110",
         A(7)  when SEL = "0111",
         A(8)  when SEL = "1000",
         A(9)  when SEL = "1001",
         A(10) when SEL = "1010",
         A(11) when SEL = "1011",
         A(12) when SEL = "1100",
         A(13) when SEL = "1101",
         A(14) when SEL = "1110",
         A(15) when others;
end process;
end RTL1;
```

MUX Models (2)
MUX Models (3)

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 RTL2 of SELECTOR is
begin
  p1 : process (A, SEL)
  begin
    case SEL is
      when "0000" => Y <= A(0);
      when "0001" => Y <= A(1);
      when "0010" => Y <= A(2);
      when "0011" => Y <= A(3);
      when "0100" => Y <= A(4);
      when "0101" => Y <= A(5);
      when "0110" => Y <= A(6);
      when "0111" => Y <= A(7);
      when "1000" => Y <= A(8);
      when "1001" => Y <= A(9);
      when "1010" => Y <= A(10);
      when "1011" => Y <= A(11);
      when "1100" => Y <= A(12);
      when "1101" => Y <= A(13);
      when "1110" => Y <= A(14);
      when others => Y <= A(15);
    end case;
  end process;
end RTL2;

MUX Models (4)

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 RTL4 of SELECTOR is
begin
  Y <= A(conv_integer(SEL));
end RTL4;

Compilation and Simulation of VHDL Code

- Compiler (Analyzer) - checks the VHDL source code
- does it conform with VHDL syntax and semantic rules
- are references to libraries correct
- Intermediate form used by a simulator or by a synthesizer
- Elaboration

Timing Model

VHDL uses the following simulation cycle to model the stimulus and response nature of digital hardware

1. Start Simulation
2. Update Signals
3. Execute Processes
4. End Simulation

Delay Types

- All VHDL signal assignment statements prescribe an amount of time that must transpire before the signal assumes its new value
- This prescribed delay can be in one of three forms:
  - Transport - prescribes propagation delay only
  - Inertial - prescribes propagation delay and minimum input pulse width
  - Delta - the default if no delay time is explicitly specified

Transport Delay

- Transport delay must be explicitly specified
- i.e. keyword "TRANSPORT" must be used
- Signal will assume its new value after specified delay
Inertial Delay

- Provides for specification propagation delay and input pulse width, i.e. ‘inertia’ of output:

  \[ \text{output} \leftarrow \text{REJECT time_expression} \text{ INERTIAL waveform} \]

- Inertial delay is default and REJECT is optional:

  \[ \text{output} \leftarrow \text{NOT Input AFTER 10 ns; } \]
  \[ \text{propagation delay and minimum pulse width are } 10\text{ns} \]

![Inertial Delay Diagram](image)

Inertial Delay (cont.)

- Example of gate with ‘inertia’ smaller than propagation delay

  - e.g. Inverter with propagation delay of 10ns which suppresses pulses shorter than 5ns

  \[ \text{output} \leftarrow \text{REJECT 5ns INERTIAL NOT Input AFTER 10ns;} \]

![Inertial Delay Diagram](image)

- Note: the REJECT feature is new to VHDL 1076-1993

Delta Delay

- Default signal assignment propagation delay if no delay is explicitly prescribed

  - VHDL signal assignments do not take place immediately

  - Delta is an infinitesimal VHDL time unit so that all signal assignments can result in signals assuming their values at a future time

  - E.g.

    \[ \text{output} \leftarrow \text{NOT Input; } \]
    \[ \text{output assumes new value in one delta cycle} \]

- Supports a model of concurrent VHDL process execution

- Order in which processes are executed by simulator does not affect simulation output

![Delta Delay Diagram](image)

Simulation Example

- Supporting a model of concurrent VHDL process execution

- Order in which processes are executed by simulator does not affect simulation output

![Simulation Example](image)

Modeling a Sequential Machine

Mealy Machine for 8421 BCD to 8421 BCD + 3 bit serial converter

How to model this in VHDL?
Behavioral VHDL Model

Two processes:
• the first represents the combinational network;
• the second represents the state register

Simulation of the VHDL Model

Simulation command file:
wave CLK X State NextState Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600

Waveforms:

Dataflow VHDL Model

The following is a description of the sequential machine of
Figure 1.2.14 terms of its next state equations.
The following state assignment was used:
S0→S0; S1→S2; S2→S3; S3→S4; S4→S0

entity shm_2 is
port(clk, in: std_logic;
Q; out: std_logic_vector(3 downto 0);
end shm_2;
architecture Equations of shm_2 is
begin
process(clk)
begin
if clk’event and clk =’1’ then
Q(0) after 20 ns;
end if;
end process;
end Equations;

Simulation of the Structural Model

Simulation command file:
wave CLK X Q1 Q2 Q3 Q Z
force CLK 0 0, 1 100 -repeat 200
force X 0 0, 1 350, 0 550, 1 750, 0 950, 1 1350
run 1600

Waveforms:

Structural Model

library BITLIB;
use BITLIB.bit_pack.all;

entity shm_2 is
port(clk: in std_logic);
end shm_2;
architecture Structure of shm_2 is
begin
-- Other processes...

Wait Statements

• an alternative to a sensitivity list
• Note: a process cannot have both wait statement(s)
  and a sensitivity list
• Generic form of a process with wait statement(s)

How wait statements work?
• Execute seq. statement until
  a wait statement is encountered.
• Wait until the specified condition is satisfied.
• Then execute the next
  set of sequential statements until
  the next wait statement is encountered.
• When the end of the process is reached
  start over again at the beginning.
Forms of Wait Statements

- **Wait on**
  - wait on sensitivity-list;
  - wait until boolean-expression;

- **Wait for**
  - waits until the time specified by the time expression has elapsed

- **Wait until**
  - the boolean expression is evaluated whenever one of the signals in the expression changes, and the process continues execution when the expression evaluates to TRUE

\[ \text{wait for } 0 \text{ ns;} \]

Using Wait Statements (1)

Using Wait Statements (2)

Problem #1

- Using the labels, list the order in which the following signal assignments are evaluated if \( \text{in2} \) changes from a '0' to a '1'. Assume \( \text{in1} \) has been a '1' and \( \text{in2} \) has been a '0' for a long time, and then at time \( t \) \( \text{in2} \) changes from a '0' to a '1'.

\[
\begin{align*}
\text{entity not_another_prob is} & \\
\text{port} & (\text{in1, in2: in bit}; \\
\text{a: out bit);} & \\
\end{align*}
\]

\[
\begin{align*}
\text{architecture oh_behave of not_another_prob is} & \\
\text{begin} & \\
\text{L1: } & \text{d } <\text{= not(in1);} \\
\text{L2: } & \text{e } <\text{= not(in2);} \\
\text{L3: } & \text{f } <\text{= (d and in2);} \\
\text{L4: } & \text{g } <\text{= (c and in1);} \\
\text{L5: } & \text{h } <\text{= g or f;} \\
\text{end oh_behave;} & \\
\end{align*}
\]

Problem #2

- Under what conditions do the two assignments below result in the same behavior? Different behavior? Draw waveforms to support your answers.

\[
\begin{align*}
\text{out } <\text{= reject 5 ns inertial (not a) after 20 ns;} & \\
\text{out } <\text{= transport (not a) after 20 ns;} & \\
\end{align*}
\]

Variables

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

\[
\begin{align*}
\text{variable list_of_variable_names : type_name} & [ := \text{initial value} ]; & \\
\end{align*}
\]

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
  \[
  \text{signal list_of_signal_names : type} \text{[ := initial value ];}
  \]
  - Declared in an architecture can be used anywhere within that architecture

Constants
- Declaration form
  \[
  \text{constant constant_name : type} \text{[ := constant_value ];}
  \]
  \[
  \text{constant delay1 : time := 5 ns;}
  \]
  - 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)
  \[
  \text{variable_name := expression;}
  \]
  - Signal assignment statement
  \[
  \text{signal_name <= expression [after delay];}
  \]
  - 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

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}.. 2^{31} – 1]
  - 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
  \[
  \text{type state_type is (SO, S1, S2, S3, S4, S5);}
  \]
  \[
  \text{signal state : state_type := S1;}
  \]
  - If no initialization, the default initialization is the leftmost element in the enumeration list (SO 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;
  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**
    ```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) = 7

- **Unconstrained array type**
  ```vhdl
  type intvec is array (natural range <>) of integer;
  ```

  - range must be specified when the array object is declared
  ```vhdl
  signal intvec5 : intvec(1 to 5) := (3,2,6,8,1);
  ```

**Sequential Machine Model Using State Table**

```vhdl
entity SM1_2 is
  port (X: in bit);
end SM1_2;

architecture SM1_2 of SM1 is
  component STATE_TABLE
  port (X, Y: in bit; Z: out bit);
end component;

begin
  SM1_2 : STATE_TABLE
  port map (X, X, Z);
end SM1_2;
```

**Predefined Unconstrained Array Types**

- **Bit_vector, string**

  ```vhdl
  constant A : bit_vector(0 to 5) := "10101";
  ```

  - Include a subset of the values specified by the type

- **Subtypes**

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

  - 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 or B) and (C and D) = 11010111100000011101000000100000

The operators would be applied in the order:

not, & xor, or, and, -

If A = 1100, B = 1111, C = 01100100, and D = 1110111, the computation would proceed as follows:

not B = 0001 (bit-by-bit complement)
A & not B = 110000 (conjunction)
C xor D = 10010010 (exclusive right 4 places)
(A is not D) or (not C and D) = 11101111 (bit-by-bit or)
A is not D or C xor D = 1111010 = TRUE (one parenthesis forces the equality 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 shr 2 is “00101010” (shift left logical, filled with ‘0’)
- A shift 3 is “00010100” (shift right logical, filled with ‘0’)
- A sls 3 is “10010111” (shift left arithmetic, filled with right bit)
- A sar 2 is “11100101” (shift right arithmetic, filled with left bit)
- A rot 3 is “10101100” (rotate left)
- A rev 5 is “10101100” (rotate right)

VHDL Functions

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

```vhdl
function rotate_right (reg bit vector) return bit vector is
begin
    return reg rot 1;
end rotate_right;
```

- 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:

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

Exit statement has the form:

```
exit;
exit when condition;
```

For Loop Example:

```vhdl
variable B: boolean;
begin
    for i in 1 to 8 loop
        B := string[i] = string2[i];
        exit when B = FALSE;
    end loop;
end comp_string;
```

VHDL Procedures

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

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

Procedure for Adding Bit_vectors

- This procedure adds two n-bit vectors and returns an n-bit sum and a carry.

```vhdl
procedure add (A, B: bit vector; C: out bit) return sum and carry is
variable Sum, C: bit;
begin
    for i in 0 to n-1 loop
        Sum(i) := A(i) + B(i) + C;
        C := (A(i) and B(i)) or (A(i) and C) or (B(i) and C);
    end loop;
end procedure add;
```

Example procedure call:

```
Sum1 <= add(A, B, C);
```

Add Function

```vhdl
function add (A, B: bit vector; C: out bit) return sum and carry is
variable Sum, C: bit;
begin
    for i in 0 to n-1 loop
        Sum(i) := A(i) + B(i) + C;
        C := (A(i) and B(i)) or (A(i) and C) or (B(i) and C);
    end loop;
end function add;
```

Example function call:

```
Sum2 <= add(A, B, C);
```
### 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>in</td>
<td>constant</td>
<td>expression</td>
<td>expression</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>signal</td>
<td>signal</td>
</tr>
<tr>
<td>out/in</td>
<td>variable</td>
<td>variable</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>signal</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></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
    ```vhdl
    package package-name is
    end package [package-name];
    ```
  - Package body (optional)
    ```vhdl
    package body package-name is
    end package body [package-name];
    ```

### Library BITLIB – bit_pack package
```
package bit_pack is
  function byte_to_tristate (b: in std_logic_vector) return std_logic_vector;
  function tri_state_to_byte (b: in std_logic_vector) return std_logic_vector;
  function tri_state_to_powerdown (b: in std_logic_vector) return std_logic_vector;
  function byte_to_powerdown (b: in std_logic_vector) return std_logic_vector;
end package;
```

### 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>S’EVENT</td>
<td>True if an event occurred during the current delta, else false</td>
</tr>
<tr>
<td>S’ACTIVE</td>
<td>True if a transition occurred during the current delta, else false</td>
</tr>
<tr>
<td>S’LAST_EVENT</td>
<td>Time elapsed since the previous event on S</td>
</tr>
<tr>
<td>S’LAST_VALUE</td>
<td>Value of S before the previous event on S</td>
</tr>
<tr>
<td>S’LAST_ACTIVE</td>
<td>Time elapsed since previous transition on S</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

A can be either an array name or an array type.
Array attributes work with signals, variables, and constants.

Transport and Inertial Delay

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 '+' (A,B : bit_vector) return bit_vector
  function '++' (A : bit_vector; B : integer) return bit_vector
  end bit_overload.

  library ieee;
  use ieee.std_logic_1164.all;
  package body bit_overload is
  function '+' (A,B : bit_vector) return bit_vector is
    begin
      return A + B;
    end;
  function '++' (A : bit_vector; B : integer) return bit_vector is
    begin
      case B is
        when 0 => return A;
        when 1 => return (A & "1'");
        when others => return (A & std_logic_vector(B) & "1'");
      end case;
    end;
  end 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

- Bit (0, 1)
- Tristate buffers and buses => high impedance state ‘Z’
- Unknown state ‘X’
  - e.g., a gate is driven by ‘Z’, output is unknown
  - a signal is simultaneously driven by ‘0’ and ‘1’

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

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)

- 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

Resolution Function for X01Z

- Define AND and OR for 4-valued inputs?

AND and OR Functions Using X01Z

- 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

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

AND Table for IEEE 9-valued

AND Function for std_logic_vectors

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
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 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(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;

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:
  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, and string 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 `writeln` 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
  ```
- Code sequence: an example
  ```
  12AC 7 (7 hex bytes follow) 
  AE 03 b6 01 01 00 SC (DX imm, LDA dir, STA ext) 
  0018 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', ..., 'F', '0', '1', ..., 'F'.
  - Corresponding values:
    0, 1, ..., 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, 10, 11, 12, 13, 14, 15
Advanced VLSI Design

VHDL Code to Fill Memory Array

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use std.textio;

entity.memoryfill is
architecture fsm of memoryfill is
  type RAMType is array (0 to 999) of std_logic_vector(7 downto 0);
signal mem: RAMType := (others => "0");
procedure fill_memory(signal mem: in RAMType type); is
  type RAMType is array (0 to 999) of integer;
  -- valid hex chars: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (upper-case only)
  constant Setup: RAMType := (0,1,2,3,4,5,6,7,8,9,-1,-1,-1,-1,-1,-1,-1);
  procedure Buff_read moda: is
    file tmp: text open read mode is "mem.txt";
    var buf: string(4 downto 1) := "0"
    variable addr: integer range 0 to 999;
    variable data: string(16 downto 1) := "0"
    variable addr, byte_cnt integer, variable data (addr, byte_cnt): 32 bits downto 0;
begin
  for i in 0 to 999 loop
    addr := i;
    wait until clk'event and clk = '1';
    A <= B and C;
    -- wait until clk'event and clk = '1';
    execute(char.read.data); -- open file for reading
    -- file.read: text is in "mem.txt";
    -- VHDL 97 version
    variable addr: integer;
    variable data: string(16 downto 1);
    variable data (addr, byte_cnt): 32 bits downto 0;
    variable addr, byte_cnt: integer;
begin
  end loop;
end fill_memory;
```  

VHDL Code to Fill Memory Array (cont’d)

```vhdl
begin
  wait until clk'event and clk = '1';
  A <= B and C;
  execute(char.read.data); -- open file for reading
  -- file.read: text is in "mem.txt";
  -- VHDL 97 version
  variable addr: integer;
  variable data: string(16 downto 1);
  variable data (addr, byte_cnt): 32 bits downto 0;
  variable addr, byte_cnt: integer;
begin
  for i in 0 to 999 loop
    addr := i;
    wait until clk'event and clk = '1';
    A <= B and C;
    -- wait until clk'event and clk = '1';
    execute(char.read.data); -- open file for reading
    -- file.read: text is in "mem.txt";
    -- VHDL 97 version
    variable addr: integer;
    variable data: string(16 downto 1);
    variable data (addr, byte_cnt): 32 bits downto 0;
    variable addr, byte_cnt: integer;
begin
  end loop;
end fill_memory;
```  

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

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use std.textio;

entity exemples is
  port B: in bit;
case A is
  when 0 => b <= '0';
  when '1' => b <= not B;
  when '1' => b <= not B;
end case;
end exemple;
```  

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

```
begin
  if A = ‘1’ then NextState <= 3;
  else NextState <= 2;
end if;
```  

Unintentional Latch Creation

```
entity latch example is
  port b: in bit;
end latch example;
architecture test of latch example is
begin
  process()
  begin
    case a is
      when 0 => b <= '0';
      when '1' => b <= not B;
      when '1' => b <= not B;
    end case;
  end process;
end test;
```  

If Statements

```
if A = ‘1’ then NextState <= 3;
else NextState <= 2;
```  

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use std.textio;

entity latch example is
  port b: in bit;
end latch example;
architecture test of latch example is
begin
  process()
  begin
    case a is
      when 0 => b <= '0';
      when '1' => b <= not B;
      when '1' => b <= not B;
    end case;
  end process;
end test;
```  

```
if A = ‘1’ then NextState <= 3;
else NextState <= 2;
```
Advanced VLSI Design

Synthesis of an If Statement

entity e is
    port(p: in std_logic_vector; cnt: out std_logic);
end e;

architecture test of e is begin
    if (p = "0") then cnt := '1';
    else cnt := '0';
end if;
end test;

Synthesized code before optimization

Advanced VLSI Design

Synthesis of a Case Statement

entity case_example is
    port(id: in std_logic_vector; out: out std_logic_vector);
end case_example;

architecture test of case_example is begin
    case id is
    when '0' => out := '1';
    when '1' => out := '0';
    when '2' => out := '1';
    when '3' => out := '0';
    end case;
end test;

Advanced VLSI Design

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

Advanced VLSI Design

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

Advanced VLSI Design

Numeric_bit, Numeric_std (cont’d)

If the left and right operand are of different lengths, the shorter operand will be sign-extended before performing an arithmetic operation. For unsigned operands, the shorter operand will be extended by filling in 0’s on the left. Examples:

signed:  "0011" + "0010" becomes  "0011" + "0010" signed:  "00111" + "01011" becomes  "00111" + "01011" + "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.
Numeric_bit, Numeric_std (cont’d)

constant A: unsigned(3 downto 0) := "1101";
constant B: signed(3 downto 0) := "0111";
variable Suma: unsigned(5 downto 0);
variable Sums: signed(4 downto 0);
variable Overflow: boolean

Suma := 'F' & A + unsigned("0101");
result in "11011" (sum = A, carry = 1)
Sums := B(3) & A + signed("1101");
result in "11000" (sum = B, carry = 1)
Overflow := Sums(4) /= Suma(0) -- Overflow is false

In the above example, the notation unsigned("0101") is a type qualification which
assumes 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 ex1 is
port(clk: in std_logic;
         datain, add, carry_in: in std_logic;
         sum, carry_out: out std_logic);
end ex1;
architecture main of ex1 is
begin
    process(clk)
    begin
        if clk'event and clk = '1' then
            if add = '0' and carry_in = '0' then
                sum <= datain;
                carry_out <= '0';
            else
                sum <= datain + add + carry_in;
                carry_out <= '1';
            end if;
        end if;
    end process;
end main;

Synthesis Examples (2a)

Mealy machine:
BCD to
BCD+3 Converter

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.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 entity;
architecture RTL of SELECTOR is
begin
    y <= a(conv_integer(sel));
end process;

Writing Test Benches

MUX 16 to 1

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.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 entity;
architecture RTL of SELECTOR is
begin
    y <= a(conv_integer(sel));
end process;

end testbench;

MUX 16 to 1

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.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 entity;
architecture RTL of SELECTOR is
begin
    y <= a(conv_integer(sel));
end process;

end testbench;
Assert Statement
- Checks to see if a certain condition is true, and if not causes an error message to be displayed
  
  ```vhdl
  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

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
entity TBSELECTOR is
end TBSELECTOR;
architecture BEH of TBSELECTOR is
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;
end BEH;
```

```vhdl
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;
end check;
```

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

Other 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