The development of these VHDL tutorial slides has been funded by Microsoft Corporation as part of the Microsoft Online Learning Initiatives in the Department of Electrical and Computer Engineering at the University of Waterloo.

Questions or comments about these VHDL tutorial slides should be directed to the author:

William D. Bishop

wdbishop@uwaterloo.ca
VHDL Tutorial Outline

- VHDL Overview
  - Introduction to VHDL
  - History of VHDL
  - Important Terminology
  - The Standardization of VHDL

- VHDL Fundamentals
  - Libraries and Packages
  - Entities, Architectures, and Configurations
  - Signals and Data Types
  - Operators
  - Processes
VHDL Tutorial Outline (cont.)

- VHDL Examples
  - Combinational circuits
  - Sequential circuits
  - Type conversion
VHDL Overview
**Introduction to VHDL**

- VHDL is a language that is used to describe the behavior of digital circuit designs

- VHDL designs can be simulated and translated into a form suitable for hardware implementation

- Hierarchical use of VHDL designs permits the rapid creation of complex digital circuit designs
History of VHDL

- VHDL was developed by the VHSIC (Very High Speed Integrated Circuit) Program in the late 1970s and early 1980s
  - The VHSIC program was funded by the U.S. Department of Defense
  - Existing tools were inadequate for complex hardware designs
- The evolution of VHDL has included the following milestones:
  - In 1981, VHDL was first proposed as a hardware description language
  - In 1986, VHDL was proposed as an IEEE standard
  - In 1987, the first VHDL standard (IEEE-1076-1987) was adopted
  - In 1993, a revised VHDL standard (IEEE-1076-1993) was adopted
  - In 2002, the current VHDL standard (IEEE-1076-2002) was adopted
- VHDL is now used extensively by industry and academia for the purpose of simulating and synthesizing digital circuit designs
Important Terminology

• *Simulation* is the prediction of the behavior of a design
  - VHDL provides many features suitable for the simulation of digital circuit designs
  - *Functional simulation* approximates the behavior of a hardware design by assuming that all outputs change at the same time
  - *Timing simulation* predicts the exact behavior of a hardware design

• *Synthesis* is the translation of a design into a netlist file that describes the structure of a hardware design
  - VHDL was not designed for the purpose of synthesis
  - Not all VHDL statements are synthesizable
Important Terminology (cont.)

- **Field-Programmable Gate Arrays (FPGAs)** are programmable logic devices that permit the rapid prototyping of a digital circuit design
  - *Configuring* a device allows the FPGA to implement virtually any digital circuit design
  - VHDL designs may be created for the purpose of generating a bitstream file to configure a device

- **Application-Specific Integrated Circuits (ASICs)** are custom integrated circuits designed to implement a specific application
  - VHDL designs may be created for the purpose of generating the detailed layout files necessary to fabricate an ASIC
Comments on VHDL Synthesis

- Perhaps Brown and Vranesic summed up the hazards of VHDL synthesis best in their text when they wrote:
  - The tendency for the novice is to write code that resembles a computer program, containing many variables and loops. It is difficult to determine what logic circuit the CAD tools will produce when synthesizing such code.

- Brown and Vranesic suggest the following:
  - A good general guideline is to assume that if the designer cannot readily determine what logic circuit is described by the VHDL code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to describe.
The Standardization of VHDL

- **IEEE 1076-1987**

- **IEEE 1076 INTERPRETATIONS-1991**

- **IEEE 1076-1993**

- **IEEE 1076-2000**

- **IEEE 1076-2002**
  - Standard VHDL Language Reference Manual [0-7381-3247-0]
Extensions to VHDL

- **IEEE 1076.1-1999**
  - IEEE Standard VHDL Analog and Mixed-Signal Extensions [0-7381-1640-8]

- **IEEE 1076.2-1996**
  - IEEE Standard VHDL Mathematical Packages [1-55937-894-8]

- **IEEE 1076.3-1997**

- **IEEE 1076.4-1995**
  - IEEE Standard VITAL ASIC Modeling Specification [1-55937-691-0]

- **IEEE 1076.5-xxxx**
  - IEEE Standard VHDL Utilities Packages [Not Standardized]

- **IEEE 1076.6-1999**
  - IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis [0-7381-1819-2]
Related IEEE Standards

- **IEEE 1164-1993**
  - Standard Multivalue Logic System for VHDL Model Interoperability [1-55937-299-0]

- **IEEE 1364-1995**
  - IEEE Standard Description Language Based on the Verilog™ Hardware Description Language [1-55937-727-5]
VHDL References

- These tutorial slides provide an overview of the essential features of VHDL

- For more information on VHDL, refer to the following references:
  
  
  
VHDL Fundamentals
Naming Conventions

• For the purpose of this tutorial, the following naming conventions will be used:
  - All VHDL keywords are shown in uppercase
  - All identifiers are shown in lowercase
  - The color highlighting used by Altera Quartus II has been used to enhance the readability of the VHDL code fragments

• In general, you should consult the style guide for your tools
  - Most (if not all tools) provide a VHDL coding style guide with style recommendations
  - Most companies implement a VHDL coding style to improve the readability of hardware designs
Libraries and Packages

- Libraries provide a set of packages, components, and functions that simplify the task of designing hardware.

- Packages provide a collection of related data types and subprograms.

- The following is an example of the use of the ieee library and its std_logic_1164 package:

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
```
Entities, Architectures, and Configurations

- The structure of a VHDL design resembles the structure of a modern, object-oriented software design.

- All VHDL designs provide an external interface and an internal implementation.

- A VHDL design consists of entities, architectures, and configurations.
Entities

• An entity is a specification of the design’s external interface

• Entity declarations specify the following:
  1. The name of the entity
  2. A set of generic declarations specifying instance-specific parameters
  3. A set of port declarations defining the inputs and outputs of the hardware design

• Generic declarations and port declarations are optional
Entity Declarations

- Entity declarations are specified as follows:

```vhdl
ENTITY entity_name IS
    GENERIC(
        generic_1_name : generic_1_type;
        generic_2_name : generic_2_type;
        generic_n_name : generic_n_type
    );
    PORT(
        port_1_name : port_1_dir port_1_type;
        port_2_name : port_2_dir port_2_type;
        port_n_name : port_n_dir port_n_type
    );
END entity_name;
```
Example Entity Declaration

- The following is an example of an entity declaration for an AND gate:

```vhdl
ENTITY andgate IS
PORT ( a : IN std_logic;
      b : IN std_logic;
      c : OUT std_logic );
END andgate;
```

**NOTE:**

In the PORT declaration, the semi-colon is used as a separator.
Ports

- Port name choices:
  - Consist of letters, digits, and/or underscores
  - Always begin with a letter
  - Case insensitive

- Port direction choices:
  IN       Input port
  OUT      Output port
  INOUT    Bidirectional port
  BUFFER   Buffered output port

NOTE:
A buffer is an output that can be “read” by the architecture of the entity.
Ports (cont.)

- IEEE standard 1164-1993 defines a package which provides a set of data types that are useful for logic synthesis
  - The external pins of a synthesizable design must use data types specified in the `std_logic_1164` package
  - IEEE recommends the use of the following data types to represent signals in a synthesizable system:

```vhdl
std_logic
std_logic_vector(<max> DOWNTO <min>)
```
Architectures

• An architecture is a specification of the design’s internal implementation

• Multiple architectures can be created for a particular entity

• For example, you might wish to create several architectures for a particular entity with each architecture optimized with respect to a design goal:
  - Performance
  - Area
  - Power Consumption
  - Ease of Simulation
Architecture Declarations

- Architecture declarations are specified as follows:

```
ARCHITECTURE architecture_name OF entity_name IS
BEGIN
    -- Insert VHDL statements to assign outputs to
    -- each of the output signals defined in the
    -- entity declaration.
END architecture_name;
```
Example Architecture Declaration

• The following is an example of an architecture declaration for an AND gate:

```vhdl
ARCHITECTURE synthesis1 OF andgate IS
BEGIN
  c <= a AND b;
END synthesis1;
```

**NOTE:**

The keyword **AND** denotes the use of an AND gate.
Configurations

- A configuration is a specification of the mapping between an architecture and a particular instance of an entity
- By default, a configuration exists for each entity
- The default configuration maps the most recently compiled architecture to the entity
- Configurations are most often used to specify alternative architectures for hardware designs
Signals

- Signals represent wires and storage elements
- Signals may only be defined inside architectures
- Signals are associated with a data type
- Signals have attributes
- VHDL is a strongly-typed language:
  - Explicit type conversion is supported
  - Implicit type conversion is not supported
Signal Representations

- Binary number representations are sufficient for software programming languages

<table>
<thead>
<tr>
<th>Binary</th>
<th>Forcing 1</th>
<th>‘1’</th>
<th>Forcing 0</th>
<th>‘0’</th>
</tr>
</thead>
</table>

- Physical wires cannot be modelled accurately using a binary number representation

- Additional values are necessary to accurately represent the state of a wire
Multi-Valued Logic Representations

- MVL (Multi-Valued Logic) representations provide the additional values necessary to represent high-impedance and unknown signals

- Two popular multi-valued signal representations are defined by \texttt{ieee.std_logic_1164}:
  - MVL - 4
  - MVL - 9
MVL - 4

- MVL - 4 adds 2 values (‘X’ and ‘Z’) to model the state of signals more accurately:

<table>
<thead>
<tr>
<th>MVL - 4</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Forcing 1</td>
<td>‘1’</td>
<td>Forcing Unknown</td>
<td>‘X’</td>
</tr>
<tr>
<td>Forcing 0</td>
<td>‘0’</td>
<td>High Impedance</td>
<td>‘Z’</td>
</tr>
</tbody>
</table>

- Wires can be driven with multiple values

- MVL-4 is rarely used since it still does not provide enough states to model signals accurately
MVL - 9

- MVL - 9 adds 5 more values to model the state of signals very accurately:

<table>
<thead>
<tr>
<th>MVL - 9</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Uninitialized</td>
<td>‘U’</td>
<td>Weak 1</td>
</tr>
<tr>
<td>Don’t Care</td>
<td>‘~’</td>
<td>Weak 0</td>
</tr>
<tr>
<td>Forcing 1</td>
<td>‘1’</td>
<td>Weak Unknown</td>
</tr>
<tr>
<td>Forcing 0</td>
<td>‘0’</td>
<td>High Impedance</td>
</tr>
<tr>
<td>Forcing Unknown</td>
<td>‘X’</td>
<td></td>
</tr>
</tbody>
</table>

- Signals can be driven with multiple values

- Signals can be resolved when conflicting values have been driven
More on MVL-9

- Four standardized types use MVL-9:

  **Unresolved Types**
  - `std_ulogic`
  - `std_ulogic_vector( <max> DOWNTO <min>)`

  **Resolved Types**
  - `std_logic`
  - `std_logic_vector( <max> DOWNTO <min> )`

- Resolved types use resolution functions to determine the value on a signal when conflicting values are driven on the signal at the same time
Built-In Data Types

• VHDL supports a rich set of built-in data types as well as user-defined data types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>Binary, Unresolved</td>
</tr>
<tr>
<td>BIT VECTOR</td>
<td>Binary, Unresolved, Array</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Binary, Unresolved, Array</td>
</tr>
<tr>
<td>REAL</td>
<td>Floating Point</td>
</tr>
</tbody>
</table>

• Built-in data types work well for simulation but not so well for synthesis

• Built-in data types are suitable for use inside an architecture but should not be used for external pins
Synthesis Vs. Simulation

- All synthesizable designs can be simulated
- Not all simulation designs can be synthesized
- Consider the following VHDL code:

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY simple_buffer IS
    PORT ( din : IN std_logic;
           dout : OUT std_logic );
END simple_buffer;

ARCHITECTURE behaviourall OF simple_buffer IS
BEGIN
    dout <= din AFTER 10 ns;
END behaviourall;
```
Synthesis Vs. Simulation (cont.)

• The input din is assigned to dout after 10 ns
  – Can this represent a real-world system? YES
  – Can this be implemented in a device? PERHAPS
  – Can this be implemented in all devices? NO

• This architecture can be simulated but not synthesized

• Some VHDL design entry tools only permit the use of synthesizable keywords

• Most tools understand a synthesizable subset of VHDL93
Logical Operators

- VHDL supports the following logical operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>( \cdot )</td>
</tr>
<tr>
<td>NAND</td>
<td>( \cdot )</td>
</tr>
<tr>
<td>OR</td>
<td>( \lor )</td>
</tr>
<tr>
<td>NOR</td>
<td>( \lor )</td>
</tr>
<tr>
<td>XOR</td>
<td>( \oplus )</td>
</tr>
<tr>
<td>XNOR</td>
<td>( \oplus )</td>
</tr>
</tbody>
</table>

- VHDL also supports the overloading of existing operators and the creation of new operators using functions.
Other Operators

- VHDL supports the following relational operators:
  - =  (Equal)
  - /= (Not Equal)
  - < (Less Than)
  - > (Greater Than)

- VHDL supports the following mathematical operators:
  - +  (Addition)
  - - (Subtraction)
  - * (Multiplication)
  - / (Division)
Assignment Statements

SIGNAL a, b, c : std_logic;
SIGNAL avec, bvec, cvec : std_logic_vector(7 DOWNTO 0);

-- Concurrent Signal Assignment Statements
-- NOTE: Both a and avec are produced concurrently
a <= b AND c;
avec <= bvec OR cvec;

-- Alternatively, signals may be assigned constants
a <= '0';
b <= '1';
c <= 'Z';
avec <= "00111010"; -- Assigns 0x3A to avec
bvec <= X"3A"; -- Assigns 0x3A to bvec
cvec <= X"3" & X"A"; -- Assigns 0x3A to cvec
Assignment Statements (cont.)

SIGNAL a, b, c, d : std_logic;
SIGNAL avec : std_logic_vector(1 DOWNTO 0);
SIGNAL bvec : std_logic_vector(2 DOWNTO 0);

-- Conditional Assignment Statement
-- NOTE: This implements a tree structure of logic gates
a <= '0' WHEN avec = "00" ELSE
     b WHEN avec = "11" ELSE
     c WHEN d = '1' ELSE
     '1';

-- Selected Signal Assignment Statement
-- NOTE: The selection values must be constants
bvec <= d & avec;
WITH bvec SELECT
a <= '0' WHEN "000",
    b WHEN "011",
    c WHEN "1--",   -- Some tools won’t synthesize ‘--’ properly
    '1' WHEN OTHERS;
Assignment Statements (cont.)

```vhdl
SIGNAL a : std_logic;
SIGNAL avec, bvec : std_logic_vector(7 DOWNTO 0);

-- Selected Signal Assignment Statement
-- NOTE: Selected signal assignments also work
-- with vectors
WITH a SELECT
  avec <= "01010101" WHEN '1',
  bvec WHEN OTHERS;
```
Process Statements

- VHDL supports processes
- Processes encapsulate a portion of a design
- Processes have a sensitivity list that specifies signals and ports that cause changes in the outputs of the process
  - Sensitivity lists can be used to preserve the state of a hardware system
- For example, an edge-triggered flip-flop circuit is sensitive to a particular clock edge
  - The output of the edge-triggered flip-flop changes if and only if a particular clock edge is received
  - Otherwise, the previous output remains asserted
Process Statements

- The keywords used for conditional assignments and selected assignments differ from those used within a process:

<table>
<thead>
<tr>
<th>Outside Processes</th>
<th>Inside Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHEN..ELSE</td>
<td>IF..ELSIF..ELSE..END IF</td>
</tr>
<tr>
<td>WITH..SELECT..WHEN</td>
<td>CASE..WHEN..END CASE</td>
</tr>
</tbody>
</table>

- A selected assignment outside a process is functionally equivalent to a case statement within a process.

- Processes can be used for combinational logic but most often, processes encapsulate sequential logic.
SIGNAL reset, clock, d, q : std_logic;

PROCESS (reset, clock)
-- reset and clock are in the sensitivity list to
-- indicate that they are important inputs to the process
BEGIN
-- IF keyword is only valid in a process
IF (reset = '0') THEN
  q <= 0;
ELSIF (clock'
  EVENT AND
  clock = '1') THEN
  q <= d;
END IF;
END PROCESS;

NOTE:
This implements a D flip-flop with an asynchronous active-low reset signal.
SIGNAL a, b, c, d : std_logic;

PROCESS (a, b, d)
-- a, b, and d are in the sensitivity list to indicate that
-- the outputs of the process are sensitive to changes in them
BEGIN
  -- CASE keyword is only valid in a process
  CASE d IS
    WHEN '0' =>
      c <= a AND b;
    WHEN OTHERS =>
      c <= '1';
  END CASE;
END PROCESS;

NOTE:

This implements a combinational circuit.
D Flip-Flop Example

- Using a process and the EVENT attribute of a signal, it is possible to specify a D flip-flop.
- The EVENT attribute can be used to check for the rising edge of a clock signal.
- The block diagram of a D Flip-Flop is shown below:
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY dffe IS
    PORT( rst, clk, ena, d : IN std_logic;
            q : OUT std_logic);
END dffe;

ARCHITECTURE synthesis1 OF dffe IS
BEGIN
    PROCESS (rst, clk)
    BEGIN
        IF (rst = '1') THEN
            q <= '0';
        ELSIF (clk'EVENT) AND (clk = '1') THEN
            IF (ena = '1') THEN
                q <= d;
            END IF;
        END IF;
    END PROCESS;
END synthesis1;
Complex Sequential Circuits

- Complex circuits may be constructed using FSMs (Finite State Machines)
- FSMs are easily specified using processes and the CASE statement
- For those interested, an example of a FSM specified in VHDL is provided on the next two slides
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY vending IS
  PORT(
    reset : IN std_logic;
    clock : IN std_logic;
    buttons : IN std_logic_vector(1 DOWNTO 0);
    lights : OUT std_logic_vector(1 DOWNTO 0)
  );
END vending;
ARCHITECTURE synthesis1 OF vending IS
  TYPE statetype IS (Idle, Opt1, Opt2, Error);
  SIGNAL currentstate, nextstate : statetype;
BEGIN
  fsm1: PROCESS( buttons, currentstate )
  BEGIN
    CASE currentstate IS
      WHEN Idle =>
        lights <= "00";
      WHEN buttons IS
        WHEN "00" =>
          nextstate <= Idle;
        WHEN "01" =>
          nextstate <= Opt1;
        WHEN "10" =>
          nextstate <= Opt2;
        WHEN OTHERS =>
          nextstate <= Error;
      END CASE;
    WHEN Opt1 =>
      lights <= "01";
      IF buttons /= "01" THEN
        nextstate <= Idle;
      END IF;
  END CASE;
END fsm1;
END;
WHEN Opt2 =>
    lights <= "10";
    IF buttons /= "10" THEN
        nextstate <= Idle;
    END IF;
WHEN Error =>
    lights <= "11";
    IF buttons = "00" THEN
        nextstate <= Idle;
    END IF;
END CASE;
END PROCESS;

fsm2: PROCESS( reset, clock )
BEGIN
    IF (reset = '0') THEN
        currentstate <= Idle;
    ELSIF (clock'EVENT) AND (clock = '1') THEN
        currentstate <= nextstate;
    END IF;
END PROCESS;
END synthesis1;
VHDL Examples

Audio commentary is not yet available for this portion of the VHDL Tutorial. Feel free to advance the slides manually using the on-screen controls.

As you browse these slides, it is HIGHLY recommended that you attempt to solve each question prior to advancing to the solution.
Comment on VHDL Examples

- The following slides present examples of synthesizable VHDL code
- Ideally, you should complete each question before viewing the solution
  - The solutions presented can be simulated and synthesized using any of the VHDL synthesis tools available at the University of Waterloo
Combinational Example 1A

- Design a VHDL entity named `andnand` to specify the interface of the following circuit:

![Circuit Diagram]

- Use `std_logic` for the port signal types of all input and output pins
Combinational Example 1A - Solution

- The VHDL description of the `andnand` entity should resemble the following:

```vhdl
ENTITY andnand IS
  PORT ( a : IN std_logic;
         b : IN std_logic;
         q : OUT std_logic;
         qbar : OUT std_logic );
END andnand;
```
Combinational Circuit Example 1B

- Design a VHDL architecture to specify the internal implementation of \texttt{and} and \texttt{nand}

- Name the architecture \texttt{synthesis1}
Combinational Example 1B - Solution

- The VHDL description of the andnand architecture should resemble the following:

```
ARCHITECTURE synthesis1 OF andnand IS
BEGIN
  q    <= a AND b;
  qbar<= a NAND b;
END synthesis1;
```
Combinational Example 2A

- Design a VHDL entity named `quadmux` to specify the interface of the Quad 2-Input MUX shown below:
Combinational Example 2A - Solution

- The VHDL description of the `quadmux` entity should resemble the following:

```vhdl
ENTITY quadmux IS
    PORT ( a : IN std_logic_vector(3 DOWNTO 0);
           b : IN std_logic_vector(3 DOWNTO 0);
           sel : IN std_logic;
           c : OUT std_logic_vector(3 DOWNTO 0) );
END quadmux;
```
Combinational Example 2B

• Design a VHDL architecture to specify the internal implementation of `quadmux`

![Quad 2-Input MUX Diagram]

• Recall that a Quad 2-Input MUX implements the following truth table:

<table>
<thead>
<tr>
<th>sel</th>
<th>c[3..0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a[3..0]</td>
</tr>
<tr>
<td>1</td>
<td>b[3..0]</td>
</tr>
</tbody>
</table>

• Name the architecture `synthesis1`
Combinational Example 2B - Solution

• The VHDL description of the quadmux architecture should resemble the following:

```vhd
ARCHITECTURE synthesis1 OF quadmux IS
BEGIN
  WITH sel SELECT
  c <= a WHEN '0',
       b WHEN OTHERS;
END synthesis1;
```
Combinational Example 3

- Design a synthesizable VHDL specification of a Seven Segment Display Controller

- The Seven Segment Display Controller is shown in the system below:
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY seven_seg IS
    PORT( dataIn : IN std_logic_vector(3 DOWNTO 0);
          segments : OUT std_logic_vector(7 DOWNTO 0) );
END seven_seg;

ARCHITECTURE synthesis1 OF seven_seg IS
BEGIN
    WITH dataIn SELECT
        segments <= "10000001" WHEN "0000", -- 0
                  "11001111" WHEN "0001", -- 1
                  "10010010" WHEN "0010", -- 2
                  "10000110" WHEN "0011", -- 3
                  "11001100" WHEN "0100", -- 4
                  "10100100" WHEN "0101", -- 5
                  "10100000" WHEN "0110", -- 6
                  "10001111" WHEN "0111", -- 7
                  "10000000" WHEN "1000", -- 8
                  "10000100" WHEN "1001", -- 9
                  "11111111" WHEN OTHERS;
END synthesis1;
Sequential Example 1

- Design a synthesizable VHDL specification of a 8-bit register with an enable signal and an asynchronous reset signal.

- The block diagram of the 8-bit register is shown below:
Sequential Example 1 - Solution

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY dregister IS
  PORT( rst, clk, ena : IN std_logic;
        d : IN std_logic_vector(7 DOWNTO 0);
        q : OUT std_logic_vector(7 DOWNTO 0) );
END dregister;

ARCHITECTURE synthesis1 OF dregister IS
BEGIN
  PROCESS (rst, clk)
  BEGIN
    IF (rst = '1') THEN
      q <= X"00";
    ELSIF (clk'EVENT) AND (clk = '1') THEN
      IF (ena = '1') THEN
        q <= d;
      END IF;
    END IF;
  END PROCESS;
END synthesis1;
Sequential Example 2

- Design a synthesizable VHDL implementation of a 32-bit counter with an enable signal and an asynchronous reset signal

- The block diagram of the 32-bit counter is shown below:
Sequential Example 2 - Solution

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY counter IS
  PORT(
    reset : IN std_logic;
    clock : IN std_logic;
    enable : IN std_logic;
    value : OUT std_logic_vector(31 DOWNTO 0)
  );
END counter;
ARCHITECTURE synthesis1 OF counter IS

SIGNAL count : unsigned(31 DOWNTO 0); -- The unsigned type is used
   -- so that unsigned arithmetic
   -- will be synthesized

BEGIN
   PROCESS (reset, clock)
   BEGIN
      IF (reset = '1') THEN
         count <= X"00000000";
      ELSIF (clock'EVENT) AND (clock = '1') THEN
         IF (enable = '1') THEN
            count <= count + 1;
         END IF;
      END IF;
   END PROCESS;

   value <= std_logic_vector(count); -- Here, the count value is
   -- converted to std_logic_vector
   -- using a conversion function

END synthesis1;
Acknowledgements

- These VHDL tutorial slides have been adapted from the notes created for ECE 324 by the following people:
  
  Wayne M. Loucks  Rob B. Gorbet  
  Carol C. W. Hulls  William D. Bishop  
  Roger Sanderson  

- In addition, examples and comments have been extracted from the notes for SE 141, ECE 223, and ECE 427 produced by the following people:
  
  Mark Aagaard  Andrew Kennings