Learning Objectives

1. Digital Systems Design
   - To design and implement of several binary arithmetic circuits

2. Xilinx Design Methodology and VHDL
   - To gain more experience with Xilinx Design Methodology
   - To learn the basics of VHDL Structural Modeling and its use in basic circuit implementation

Introduction and Overview

In this experiment, you will design and implement two common digital circuits: a 4-bit Binary Adder and a 4-bit Comparator. Implementing these designs builds upon your knowledge of VHDL and the Xilinx ISE development environment. As an added bonus, the 4-bit comparator is used in the final experiment to implement a Digital Alarm System.

Good digital design practices involve the ability to segment a larger system into smaller pieces that can be designed and tested independently before integrating them into a larger system. This process can be repeated at multiple levels of a digital circuit design and is often referred to as a hierarchical design approach. A VHDL mechanism known as structural modeling directly supports the hierarchical digital design in that it allows for the segmenting of large designs and their subsequent integration into a larger circuit. In this lab, you use these modular design techniques to implement a 4-bit Ripple Carry Adder (RCA). The design of an RCA is best done by designing a single bit module and cascading this module four times to form the 4-bit RCA. A 4-bit adder is designed in this experiment but the same technique can be used to design a RCA of any bit length.

The adder circuit is one of many arithmetic circuits of interest. Other arithmetic circuits include subtractors, multipliers, dividers, and comparators. Circuits that require these functions are typically microcontrollers, microprocessors, and microwave ovens. Complex processors typically contain all of these circuits while simpler machines may only have addition and subtraction circuits, leaving it up to a programmer to implement a multiplication algorithm using only addition operators. Comparators are generic devices that determine the relationship between two n-bit inputs, e.g., if input A is less than input B. Depending on how they are designed, they can generate outputs for one or more of the following: EQUAL, LESS_THAN, GREATER_THAN, or some combination of the three.

The simple comparator you will design compares two 4-bit numbers and indicates if they are exactly equal (or not). With respect to a Digital Alarm System like the one in our final experiment, such a comparator can be used to determine if an access code is equal to a preset key. The comparator takes a 4-bit input, compares it to a hardwired key value, and asserts an output signal if the input value matches the key value.

Comparators can be implemented by many different methods, using a number of different algorithms. The logic circuit approach you’re probably most familiar with is to use exclusive-NOR gates for the comparator design. However, the comparator is a good example of a device that could be implemented using a variety of approaches in VHDL.
Structural Modeling Using VHDL

There are generally three approaches to writing VHDL code: dataflow modeling, behavioral modeling, and structural modeling. Your VHDL designs in previous experiments were either dataflow or behavioral models. The use of structural modeling facilitates the design and development of complex digital circuits in that the re-use of previously designed VHDL modules is nicely supported by this method. The VHDL modular design approach also directly supports the hierarchical design paradigm which is essential to the design and understanding of complex digital systems.

The design of complex digital circuits using VHDL should closely resemble the structure of complex computer programs. Many of the techniques and practices used to construct large and well-structured computer programs written in higher-level languages should also be applied when using VHDL to describe digital circuits. This common structure we are referring to is the ever so popular modular approach to source coding. The term “structural modeling” is the terminology that VHDL uses for the modular design approach.

The benefits of modular design to VHDL are similar to the benefits that modular design or object oriented design provide for higher-level computer languages. Modular designs make complex systems easier to design and easier to understand by packing low-level functionality into modules. These modules can easily be reused in other designs, thus saving the designer time by removing the need to “reinvent (and retest) the wheel.” The resultant design at its highest level is easier to understand due to the compartmentalization of the circuit’s functions. Thus, at the highest level, the design is seen as an interconnection of different (possibly complex) functions; without the complexity of how each function is implemented obscuring or confusing this viewpoint.

VHDL and Computer Programming Languages: Exploiting the Similarities

The main tool for modularity in higher-level languages such as “C” is the function. In other less useful computer languages, such modularity is accomplished through the use of methods, procedures, and subroutines. The approach used in “C” is to:

1) Name the function interface you plan on writing (the function declaration),
2) Code what the function does (the function body),
3) Let the program know the function exists and is available to be called (the proto-type), and
4) Call the function from the main portion of the code.

The approach used in VHDL is similar:

1) Name the module you plan to describe (the entity),
2) Describe what the module does (the architecture),
3) Let the program know the module exists and can be used (component declaration), and
4) Use the module in your code (component instantiation, or mapping).

The similarities between these two approaches are listed in Table 1.

<table>
<thead>
<tr>
<th>“C” programming language</th>
<th>VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Describe function interface</td>
<td>the entity</td>
</tr>
<tr>
<td>Describe what the function does (coding)</td>
<td>the architecture</td>
</tr>
<tr>
<td>Provide a function prototype to main program</td>
<td>component declaration</td>
</tr>
<tr>
<td>Call the function from main program</td>
<td>component instantiation (mapping)</td>
</tr>
</tbody>
</table>

Table 1: Similarities between modules in "C" and VHDL.
EXAMPLE

Design the circuit shown below using structural modeling. Specifically, design individual modules for the AND and OR devices and use them in the final circuit using structural modeling.

Solution: The solution presented here implements the discrete gate version of the circuit shown on the right in the example. This solution is primarily a demonstration of VHDL structural modeling, and does not represent the most efficient method to represent this circuit using VHDL. The approach of this solution is to model each of the discrete gates as individual “systems”. The interface requirements of VHDL structural modeling are the same regardless of whether the circuit elements are simple gates or complex digital subsystems.

The circuit shown in Figure 1 contains some extra information that is used in the VHDL structural implementation. First, the dashed line represents the boundary of the overall VHDL entity i.e., signals that cross this boundary appear in the entity declaration for this implementation. Second, each of the internal signals (signals that do not cross the dashed entity boundary) is assigned a unique name. Assigning names for the internal signals is similar to the method presented in Experiment 6.

Figure 1: Discrete gate implementation of example circuit.

The first part of the solution is to provide entity and architecture implementations for the individual gates (the design modules) shown in Figure 1. We need to provide at least one definition of an OR gate and another for the AND gate. We only need to provide one definition of the AND gate despite the fact that two AND gates are shown in the diagram. The modular VHDL approach allows us to reuse circuit definitions and we will take advantage of this feature. The gate definitions are shown in Figure 2.
Figure 2: Entity and Architecture definitions for discrete gates.

The module implementations shown in Figure 2 present no new VHDL details and are thus massively boring. The new information is contained in how the circuit elements listed in Figure 2 are used as components in a larger, higher-level circuit. The procedure for implementing a structural VHDL design is summarized in the following steps. These steps assume that the entity declarations for the interior modules already exist.

1. Generate the entity declaration for the higher-level circuit
2. Declare the design units (modules) used in the higher-level design
3. Declare required internal signals
4. Instantiate and Map the design units

Step One in a structural implementation is identical to the standard approach we’ve used for implementing other VHDL circuits: describe the entity. The entity declaration is derived directly from the dashed box in the example and is shown in Figure 3. Once again, the entity declaration describes the interface to the circuit being described.

```
entity my_example is
  Port ( A,B,C : in std_logic;
         F : out std_logic);
end my_example;
```

Figure 3: Entity declaration for example circuit.

Step Two is to declare the design units that are used in the circuit. In VHDL lingo, declaration refers to the act of making a particular design unit available for use in a particular design module. Note that the act of declaring a design unit, by definition, transforms your circuit into a hierarchical design. The declaration of a design unit makes the unit available for placement into the design hierarchy. As with a function prototype in a high-level programming language, the component declaration provides all the information that the higher-level module needs to know about how to interface with a design unit that is described in detail in some other module – namely: what are its inputs and outputs? (How many? What sizes? What types?)
For our design, we need to declare two separate design units: the OR gate and the AND gate. Note that despite the fact that there are two AND gates in the design, you only need one AND gate declaration.

There are two factors involved in declaring a design unit: 1) how to do it, and, 2) where to place it. A component declaration can be viewed as a modification of the associated entity declaration. The difference is that the word entity is replaced with the word component at the beginning of the declaration, and the word component must also follow the word end to terminate the declaration. The best way to do this is by cutting, pasting, and modifying the original entity declaration. The resulting component declaration is placed in the architecture declaration after the architecture line and before the begin line. The two component declarations and their associated entity declarations are shown in Table 2. Figure 4 shows the component declarations as they appear in working VHDL code.

In order for the VHDL compiler to properly associate the component declaration for a design unit in a higher-level module with its detailed description in some other module, your component declaration must use exactly the same name and specify exactly the same port signals (same names, same types, and in the same order) as the entity in the lower-level module that provides the design details.

<table>
<thead>
<tr>
<th>Entity Declarations</th>
<th>Component Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>entity my_or is</td>
<td>component my_or</td>
</tr>
<tr>
<td>Port ( A,B : in std_logic; F : out std_logic);</td>
<td>Port ( A,B : in std_logic; F : out std_logic);</td>
</tr>
<tr>
<td>end my_or;</td>
<td>end component;</td>
</tr>
<tr>
<td>entity my_and is</td>
<td>component my_and</td>
</tr>
<tr>
<td>Port ( A,B : in std_logic; F : out std_logic);</td>
<td>Port ( A,B : in std_logic; F : out std_logic);</td>
</tr>
<tr>
<td>end my_and;</td>
<td>end component;</td>
</tr>
</tbody>
</table>

Table 2: A comparison of entity and component declarations.

Step Three is to declare internal signals used by your design. The required internal signals for this design are the signals that are not intersected by the dashed line shown in Figure 1. In other words, these are signals that the world outside of the entity is not aware of. These two signals are similar to local variables used in higher-level programming languages in that they must be declared before they are used in the design. These signals effectively provide an interface between the various design units that are instantiated in the final design. For this design, two signals are required and used as the outputs of the AND gates and inputs to the OR gate. Internal signal declarations such as these appear along with the component declarations in the architecture declaration after the architecture line and before the begin line. Note that the declaration of intermediate signals is similar to the signal declaration contained in the entity body. The only difference is that the intermediate signal declaration does not contain the mode specifier (in, out, etc.). The internal signal declarations are included as part of the final solution shown in Figure 4.

Step Four is to create individual instances of the required modules and map these instances of the various components in the architecture body. Cleverly enough, we refer to this process of adding and connecting the unique instances of the design units: the component instantiation process. In the terminology presented here, declaration refers to the component definition that appears before the begin line in the architecture, while instantiation refers to the creation of individual instances (or copies) of a component; and these are placed after the begin line.

The process of signal mapping provides the interface definitions for the individual components in the design. This mapping step associates external connections from each of the components to signals defined in the next level upwards in the design hierarchy. Each of the input / output signals associated with individual components “maps” to either an internal or external signal in the higher-level design module. In this way, an individual component instance gets connected to other components and ports in the higher-level design module.
Each of the individual mapping statements (each component instantiation) includes a unique name for the particular instance of the component (a label), as well as the name of the original entity that defines that component (which should match a name provided in a component declaration earlier in the same higher-level module architecture). The actual signal mapping information follows the VHDL key words of: **port map**. All of this information appears in the final solution shown in Figure 4.

```vhdl
entity my_example is
  Port ( A,B,C : in std_logic;
          F : out std_logic);
end my_example;

architecture ckt3 of my_example is

  -- component declaration ---------------
  component my_or
    Port ( A,B : in std_logic;
           F : out std_logic);
  end component;

  -- component declaration ---------------
  component my_and
    Port ( A,B : in std_logic;
           F : out std_logic);
  end component;

  -- intermediate signal declaration
  signal sig1, sig2 : std_logic;

begin
  -- component instantiation -----------
  x1: my_and port map (A => A,
                       B => B,
                       F => sig1);

  x2: my_and port map (A => A,
                       B => C,
                       F => sig2);

  X3: my_or port map (A => sig1,
                      B => sig2,
                      F => F);
end ckt3;
```

Figure 4: VHDL code for the top of the design hierarchy for the example circuit.

**Procedure**

**Procedure 1: Implementation of a Half Adder**

1. Design a Half Adder. The Half Adder provides the sum and carry out of the addition of two 1-bit inputs. Place the associated truth table, the minimized expressions for the sum and carry output and black-box diagram in your lab report.

2. Implement the Half Adder circuit in VHDL. Verify proper operation of the circuit using ModelSim. Include the VHDL code and an annotated simulation output in your lab report.
Procedure 2: Implementation of a Full Adder

1. Design a Full Adder. The Full Adder provides the sum and carry out of the addition of three 1-bit inputs. Place the associated truth table, the minimized expressions for the sum and carry output, and black-box diagram in your lab report.

2. Implement the Full Adder circuit in VHDL. Create your Full Adder VHDL module as a New Source in the same Xilinx ISE project that contains your half adder design. Verify proper operation of the circuit using ModelSim. Include the VHDL code in your report and an annotated simulation output in your lab report.

Procedure 3: 4-bit Ripple Carry Adder

1. Using the Half Adder and Full Adder that you designed in the previous sections, design and implement the 4-bit Ripple Carry Adder (RCA) shown in Figure 7. Use VHDL structural modeling within the same Xilinx ISE project for your implementation, using both the Half Adder and Full Adder VHDL modules designed in the two previous procedures as components in the RCA design. Include the VHDL code and a block diagram for the RCA in your lab report. Make sure the names of your component modules, input/output signals and intermediate signals in your VHDL code match the names in your block diagram.

2. Perform a “Behavioral Simulation” of your 4-bit Ripple Carry Adder using ModelSim. Include a test case in your test bench waveforms that will result in the worst case (longest) delays from the RCA inputs to final settled values for the device outputs (see Problem #4). Include the annotated output of the Behavioral simulation in your lab report.

3. Implement your RCA design on the development board using the pin assignments in Table 3.
4. Perform a “Post-Route Simulation” of your Ripple Carry Adder using ModelSim and the same test bench waveforms used for the Behavioral Simulation. Measure the value of the worst case RCA output delay shown in your simulation results and record this value in your lab report. Provide a print out of the simulation results showing this worst case delay (zoomed-in view), with the beginning and end of this delay time clearly marked. Use the following steps to run the Post-Route Simulation:

a) In the Sources for: drop-down menu, select Post-Route Simulation. Select your Testbench File in the Sources window.

b) Select the Processes tab below the Processes window. Expand the Modelsim Simulator heading. Right-click on the Simulate Post-Place & Route Model heading that appears.

c) Select the Simulation Properties Category in the Process Properties window that appears.

d) If needed, change the Value next to the Property Name “Simulation Run Time” to match the value and units you used for your Testbench Waveforms.

e) Click on OK to close the Properties.

f) Launch the ModelSim XE simulator by Double-clicking on the Simulate Post-Place & Route Model heading in the Processes window.

5. Repeat the “Behavioral Simulation” to measure the value that it determines for the same worst case output delay. Record the Behavioral Simulation value of this worst case delay in your lab report. Explain any differences in the delay values seen in the two simulations.

6. Demonstrate the final design to your lab instructor before you commence the celebration.

![Block diagram for a 4-bit Ripple Carry Adder.](image)

**Figure 7: Block diagram for a 4-bit Ripple Carry Adder.**

<table>
<thead>
<tr>
<th>Board</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>Nexys &amp;</td>
<td>a&lt;sub&gt;1&lt;/sub&gt;-a&lt;sub&gt;0&lt;/sub&gt;: SW7-SW4</td>
<td>b&lt;sub&gt;3&lt;/sub&gt;-b&lt;sub&gt;0&lt;/sub&gt;: SW3-SW0</td>
</tr>
<tr>
<td>Nexys-2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3: Pin assignment for the RCA.**
Procedure 4: The 4-bit Comparator

1. Include in your lab report the truth table and minimized output equation for a 2-bit comparator (a circuit that compares two 2-bit numbers). The comparator should have a single 1-bit output signal, EQUAL, that is asserted when the two 2-bit input numbers are exactly equal.

2. Extend the 2-bit comparator to a 4-bit comparator by extrapolating the result from the previous step. The 4-bit comparator has two 4-bit inputs and one output, EQUAL, which is asserted if the two 4-bit inputs are equivalent. Include the equations describing the 4-bit comparator in your lab report. Enter your design in a new Xilinx ISE project using VHDL and include the source code in your lab report.

3. Simulate your 4-bit comparator using ModelSim and include the waveform in your report.

4. Implement the comparator on the development board using the pin assignments shown in Table 4 and demonstrate the working circuit to your instructor. Be sure to save a copy of your comparator for future use in Experiment 9.

<table>
<thead>
<tr>
<th>Board</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nexys &amp; Nexys-2</td>
<td>a2-a0: SW7-SW4</td>
<td>b2-b0: SW3-SW0</td>
</tr>
</tbody>
</table>

Table 4: Pin assignment for the comparator.

Questions

1. Determine the propagation delay (in number of gates) from each input to each output for the half adder. Base your answer on the logic from the reduced equations.

2. Determine the propagation delay (in number of gates) from each input to each output for the full adder. Base your answer on the logic from the reduced equations.

3. How many rows would there be in the truth table for a 4-bit binary adder? How many input and output variables are there for a 4-bit adder? Would it be feasible to design a 32-bit adder using a truth table and Boolean logic expressions? Why or why not?

4. How many logic gates are needed to build the 4-bit ripple carry adder? What is the worst case propagation delay path through the 4-bit ripple carry adder? What 4-bit input values for A and B cause the worst case delay?

5. What is the worst cast delay (in number of gates) for an 8-bit ripple carry adder? Write a formula that relates the worst case number of gate delays to the number of bits in the ripple carry adder.

6. How would you modify your 4-bit ripple carry adder to make an adder/subtractor?
7. There is a particular shortcoming in the design of the ripple carry adder implemented in this experiment (in addition to the long propagation delays) that prevents it from being reused itself as a module for constructing RCAs with higher bit-widths. What is that shortcoming?

8. The comparator in this experiment could either be designed using a behavioral, dataflow, or structural model. Identify which VHDL model(s) you used for your 4-bit comparator design, and explain what features distinguish it as that particular model type.

9. Speculate on how the comparator might fit into the design of a Digital Alarm System that is so often mentioned. Fully explain your answer.