LAB GUIDE

A: Design Procedure for digital circuits

Here is a short explanation of the design process as it applies to digital logic design for FPGAs. During this short explanation, you can refer to Figure A.1 as needed. When designing a digital logic design, the first step is to clearly define what needs to be done including all of the inputs and outputs, any timing considerations, and all the functions that need to be addressed. This falls under the 'Design High-Level Preparation' step. This is taken care of for you in this lab. The next step is the 'Design Entry' step. This involves actually entering the design into a tool of some sorts. At this point there is no connection to a physical device, meaning that you have not defined what inputs go to what pins of a real device, and in many cases, not even the device is defined at this point.

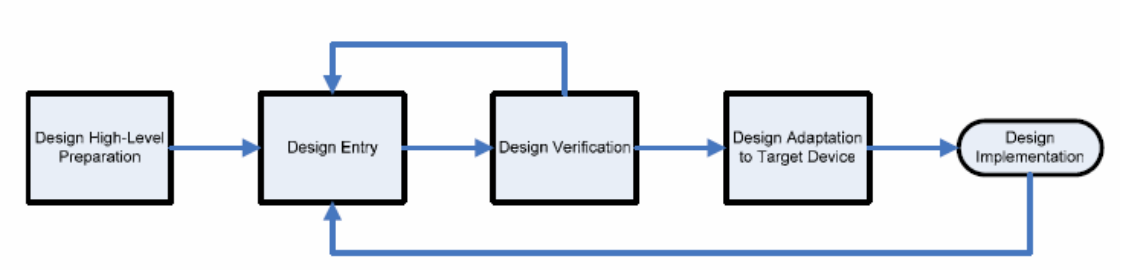

Figure A.1: Sample Design Flow for a digital System

The next step is normally simulation of the design in software called 'Design Verification.' For very simple designs, this is not always necessary but is a good idea. For this section we will be skipping this step. 'Design Verification' can lead to changes in the design hence the connection to the 'Design Entry' block.

After the design is entered and tested, it is adapted to the target device. In this step we decide which pins of the physical device connect to which parts of the internal logic design, the timing requirements for many of the pins, and often the selection of the physical device itself. For the sections in this manual, the device has been chosen for you. However you still need to decide on the pin connections and required timings yourself.

The final step is to download the synthesized design to the physical device and try it out. If needed, changes can then be made to the design to reflect errors or incomplete design specifications.

B: Using the Xilinx Software to Simulate your Logic

This is a step-by-step procedure to use the Xilinx ISE 7.1i software for simulating your designs. We will be using an adder circuit as an example.

Start the Xilinx software. (Start, All Programs, Xilinx ISE 7.1i, Project Navigator)

Notes:

1. You can stop your Xilinx session at any time. Save all source files that are open, and exit the software. When you exit, the project file is automatically saved with the most recent changes you made. When you restart the software, you will see the content of your project with the last saved changes.

2. You can access Help at any time during a session. Press **F1** to view the help for the specific tool or function your are currently using. Design flow-based help is called ISE Help and is accessible from the **Help** menu in Project Navigator. This help package contains information about creating and maintaining your complete design flow in ISE.

1. Create a New Project.

A **project** in ISE is a collection of all files necessary to create and download a design to the selected device. The project we will be creating will be targeted to use our FPGA and allow us to draw a schematic as the main way of entering the design. Here are the steps for creating a new project:

1. Select **File, New Project**.
2. In the **New Project Wizard** dialog box, type the desired location in the **Project Location** field, or browse to the directory under which you want to create your new project directory using the browse button next to the **Project Location** field. You will need to save your project on your network drive, Z: or some removable media such as a USB drive. Do not save your files on the local machine.
3. Enter "Lab3" in the **Project Name** field. When you enter "Lab3" in the **Project Name** field, a Lab3 subdirectory is automatically created in the directory path in the **Project Location** field. For example, for the directory path Z:\COE203, entering the Project Name "Lab3" modifies the path as Z:\COE203\Lab3.
4. Use the pull-down arrow to select **Schematic** from the **Top-Level Module Type** field. Click in the field to access the pull-down list. (NOTE: you can apply the fundamentals learned from this tutorial to either an HDL or schematic design containing both schematic and/or HDL sources.)
5. Click **Next**.
6. In the **New Project Wizard Device and Design Flow** dialog box, use the pull-down arrow to select the **Value** for each **Property Name**. Click in the field to access the pull-down list. Make sure the values are as follows:

    - Device Family: **Spartan3**
    - Device: **xc3s200**
    - Package: **ft256**

- Speed Grade: **-4**
- Top-Level Module Type: **Schematic**
- Synthesis Tool: **XST**
- Simulator: **ModelSim**
- Generated Simulation Language: **Verilog**.

When the table is complete, your project properties should look like the following:

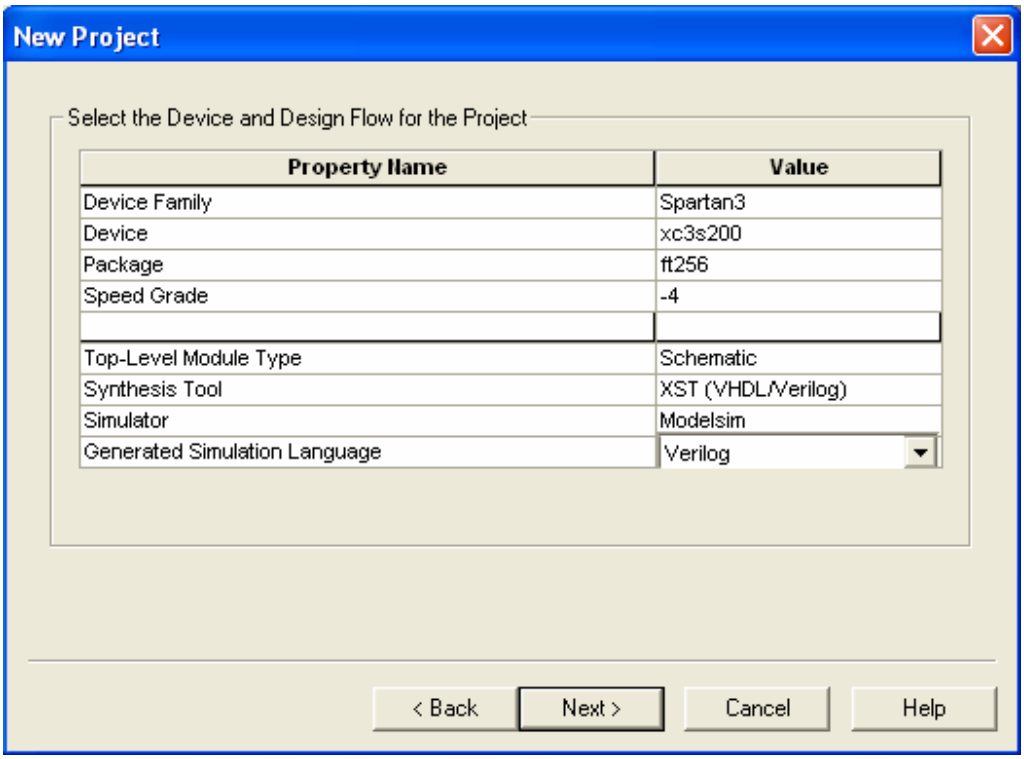

Figure B.1: Project Properties

7. Click **Next**
8. In the **Create a New Source** dialog box, click the **New Source** button. Select **Schematic** from the box on the left, and type in a file name for your project such as "lab3". Click **Next**. Click **Finish**. Click **Next**
9. In the **Add Existing Sources** dialog box, click **Next**.
10. In the **New Project Information** dialog box, click **Finish**.

ISE creates and displays the new project in the **Sources in Project** window, and opens the lab3.sch file in the Xilinx tool for creating and editing schematic diagrams, Engineering Capture System (ECS).

2. Schematic Design Entry

This section demonstrates how to create a schematic that contains logic gates. It describes how to wire them together, add net names to the wires, and add I/O markers to show where signals enter or exit the schematic. You may refer to Figure B.2 in this section as a reminder of what you need to build.

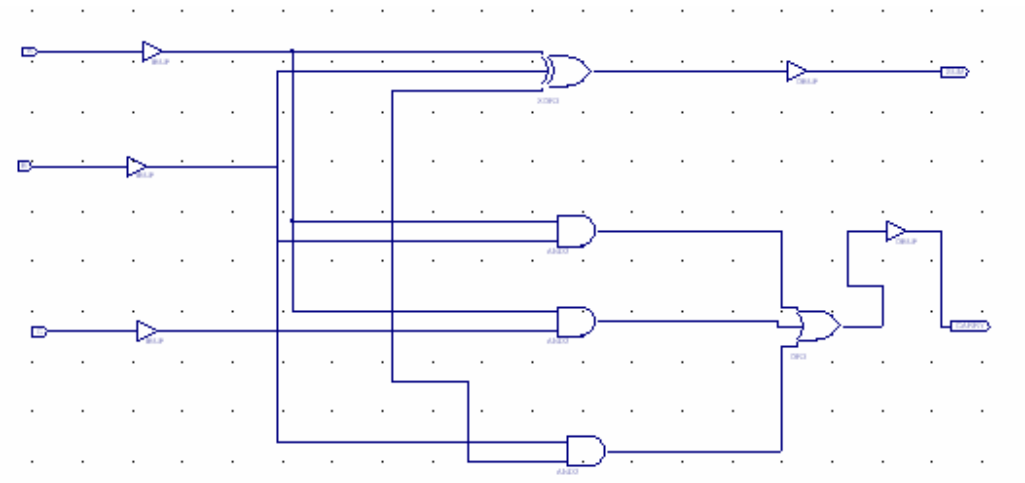

Figure B.2: The circuit to be designed

1. Create a Top-Level Schematic.

   ECS is already launched and a blank sheet is open in an ECS schematic window. If ECS is not launched, then double click on the schematic in the Sources in Project window. In ECS, you will create a schematic diagram from scratch.

2. Add a 2 input AND gate.
   - Select **Add, Symbol** or click the **Add Symbol** icon in the Tools toolbar (looks like a gate with a resistor below t).
   - Select **Logic** from the list of **Categories**.
   - Select and2 from the list of **Symbols**.
   - Place one AND gate on the schematic. Click the left mouse button to place the gate on the schematic where the cursor sits.
   - Press Esc to exit **Add Symbol** mode and restore your select tool.

3. Add any other gates (xor3, or3) you might need to make your schematic.
4. If you like, adjust your view using the **Zoom** option (**View, Zoom, In**) and the scroll bars in ECS.
5. Now we need to wire the schematic. When wiring the schematic symbols, some wires interconnect the modules and others are extended and left hanging for I/O.

- To activate the drawing tool, select **Add, Wire** or select the **Add Wire** icon from the Tools toolbar (looks like a pencil drawing a wire).
- To add a hanging wire or to extend the wire:
    - Click and hold the mouse button at the vertex of a pin or simply click on the pin. When your mouse is over a pin, a box will appear.
    - Drag the mouse to extend the wire to the desired length or click at the point you wish to connect to.
    - Release the mouse button at the location you want the wire to terminate or double click.
- When finished wiring, press **Esc** to exit **Add Wire** mode.

6. Add I/O buffers

   IO buffers are used by the schematic capture tool to understand which internal signals are connected to the pins of the physical device that will be used. You have three input wires and one output wire. You will need an input buffer for each input and an output buffer for the output.

    - Select **Add** -> **Symbol** or click the **Add Symbol** icon from the **Tools** toolbar.
    - Select **IO** from the menu
    - Select **ibuf** and place 3 input buffers on the left side of your schematic
    - Select **obuf** and place 1 output buffer on the right side of your schematic
    - Add a hanging wire to the left side of each input buffer and the right side of each output buffer
    - Wire the inputs and outputs into your logic

7. Add Net Names to Wires:

   After wiring the schematic symbols, you are ready to add net names to the wires. Net names should only be added to nets that will be directly connected to I/O pins. (That is, before input buffers and after output buffers.) Note: net names are only for you to keep track of nodes easily and will not affect the behavior of your design. Adding net names to the IO will make it easier for you to assign IO pins later.

    - Select **Add, Net Name** or click the **Add Net Name** icon from the Tools toolbar.
    - To create and place a net name for each hanging wire:
        - Type the net name in the text box in the **Options** tab, located to the left of the screen.
            - Note: leave the default options as **Name the branch** and **Keep the name**.

- Place the cursor, which now displays the net name, at the end of the hanging wire.
- Click the left mouse button.

- Name A B, and C as inputs and Z as the output.

8. Add I/O Markers

   IO markers are needed by the design tool to synthesize the design. They give a logical connection for the synthesis tool to understand that the internal signal will be passed outside either the chip or schematic. It is very important that the correct type of IO marker be used. Putting an input IO marker on an output buffer will cause an error.

   - Select **Add, I/O Marker** or click the **Add I/O Marker** icon from the Tools toolbar.
   - Add input markers to the A, B, C inputs:
     a) Select the **Add an input marker** radio button on the **Options** tab.
     b) Place the cursor, which now displays the input graphic, at the end of the input wire.
     c) Click the left mouse button to add the marker. (Note: You can also label all of your inputs or outputs at once by drawing a box around the nodes you wish to label.)

   The input graphic is added to the end of the wire, around the net name.

   - Add an output marker to the Z output:
     a) Select the **Add an output marker** radio button on the **Options** tab.
     b) Place the cursor, which now displays the input graphic, at the end of the output wire.
     c) Click the left mouse button to add the marker. The output graphic is added to the end of the wire, around the net name.

Your schematic is complete. Save the schematic diagram using **File, Save**. Then exit ECS.

3. Behavioral Simulation

ISE provides an integrated flow with the ModelTech ModelSim simulator that allows simulations to be run from the Xilinx Project Navigator graphical user interface (GUI).

In this section, we will introduce the concept of test bench and show how to verify the function of our circuit by behavioral simulation.

What is a test bench?

A test bench supplies stimuli to the design, observes the outputs of the design, and compares the observed outputs with the expected values. If any mismatch happens, the test bench issues certain messages signifying that there are errors in the design. Figure B.3 shows the concept of test bench.

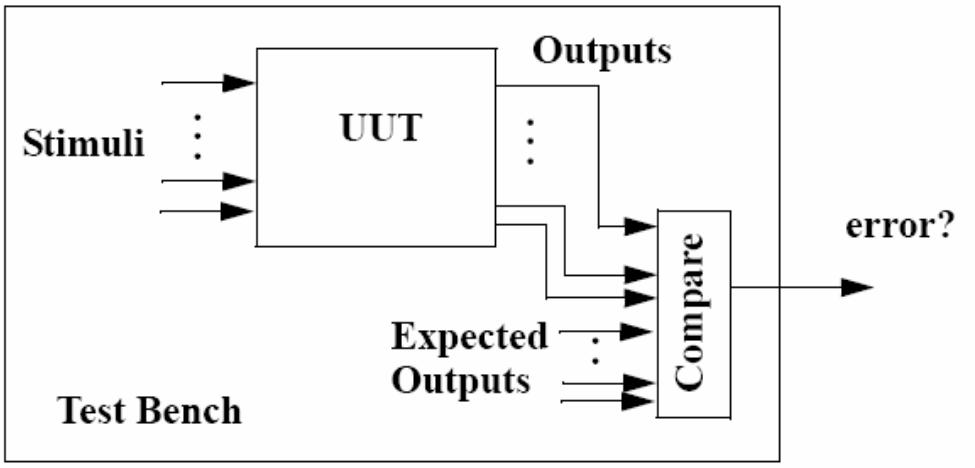

Figure B.3: A conceptual diagram of the test bench

1. In your **Project Navigator** window, click on your schematic file ***lab3 (lab3.sch)*** to make it active. Now select **Project → New Source**. In the window that opens up select the option **Test Bench Waveform.** Specify a name for the waveform in the **File Name** field and click on **Next.** In the following window click on **Next** and then finally click **Finish.**

2. In the **Initialize Timing** window (Figure B.4), select the option **Combinatorial Design**. In the input boxes after **Check outputs and Assign Inputs**, enter the value 25, **Initial Length of Test Bench** to 1000 ns and set the time scale to **ns** as

shown below and click **OK**. This will open the **HDL Bencher** window (Figure B.5).

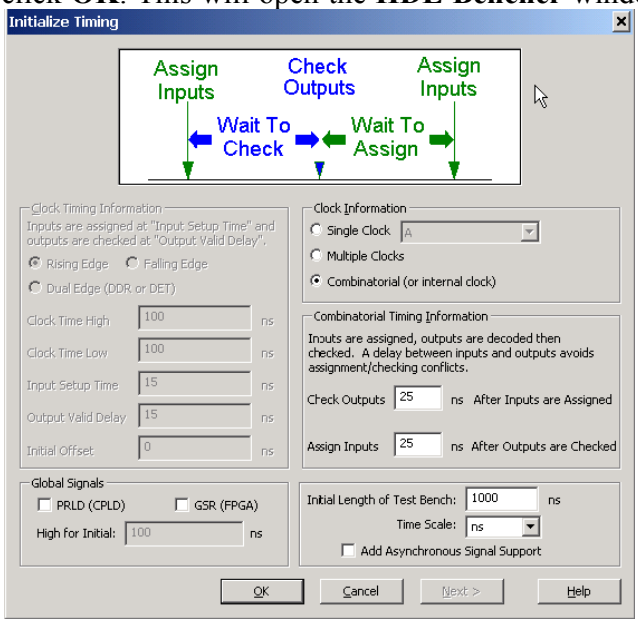

Figure B.4: Initialize Timing Window

3. Click **OK** and you'll see the waveform window of the test bench. The three input signals are marked cyan while the two output signals are marked yellow. By directly clicking on the waveform you can change the values of the signal. Just play around a little to get familiar with it. Now specify the waveforms of the three inputs as shown in Figure B.5. Notice that the waveforms of *A*, *B* and C cover all possible 8 combinations. For each of the eight combinations, draw the expected outputs on the waveforms of *CARRY* and *SUM*. Recall that the outputs are supposed to be 25ns later than the inputs as we specified in the previous window. Save the waveforms after you're done.

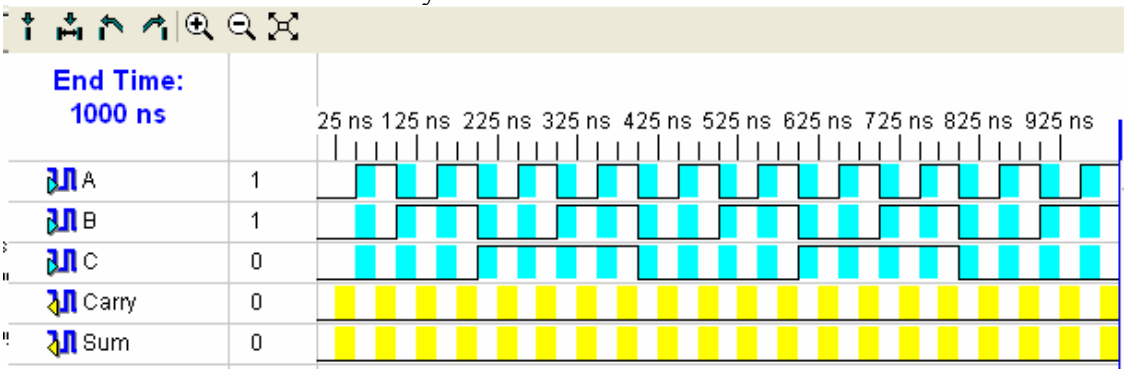

Figure B.5: Test Bench Waveforms

4. Now go back to the ***Project Navigator*** windows, and make sure that you have the test bench waveform you just created selected in ***Sources in Project*** window. In

the ***Process View*** window, double click ***Simulate Behavioral Model***. This will open up ModelSim simulation windows and run the test bench simulation. If ModelSim fails to start, you need to go back to check the license.

5. Right-click on the waveforms and select ***Zoom Range***. Choose ***Start*** as 1 ns and ***End*** as 1000 ns.

6. Check the waveforms to see whether there are any errors. In particular, pay attention to signal ***tx_error*** in the ***Objects*** window. ***tx_error*** counts how many errors are detected in the simulation. In this case, ***tx_error*** is 0 meaning everything looks fine.

4. Assigning Package Pins

We will use the switches and LEDs that are already on the Spartan-3 Board to implement the design.

1. Choose input and output pins on your Digital Logic Board. See appendix A or for the pinout diagram. Refer to Appendix B to find which pins on the FPGA are connected to the switches and LEDs on board. Pick two LED pins (output) and three switch pins (inputs).
2. Click on **Assign Package Pins** under **User Constraints.** This will launch the Pin-out Area Constraint Editor (PACE).
3. In PACE, select the **Package View Tab** to open the Package Pins window. This window shows the graphical representation of the device package.
4. The **Design Object List** window shows all the IOs in the design.
5. In the Design Object List window, click and drag the input and output signals to the specific locations which you have assigned in Step 1. A sample assignment is shown in Figure B.6.

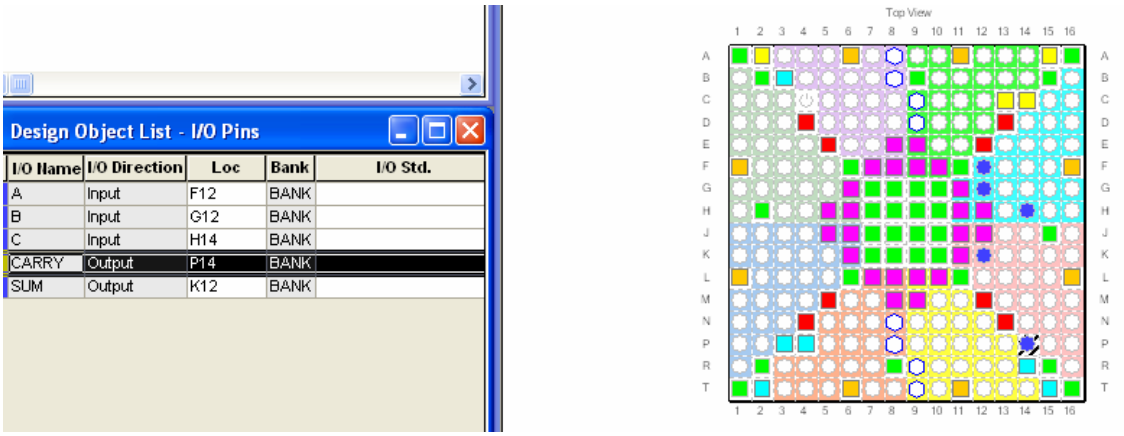

Figure B.6: Assigning pins in PACE

6. Once the pins are locked down, select **File → Save**. The changes made in PACE are saved in the *lab3.ucf* file in your current working directory.
7. Exit PACE.

5. Design Implementation

Design implementation covers running the Implement Design process in **Project Navigator**.

Note: For more information about implementing a design, see ISE Help. Select **Help, ISE Help Contents**, expand either the FPGA or CPLD hierarchy in the left pane and expand the **Implementing a Design** hierarchy.

1. Run Implement Design

   First, run all processes (Synthesis through Place & Route) associated with the design. To do so, run Implement Design on the schematic file:

   - Select lab3.sch in the **Sources in Project** window.
   - Double-click **Implement Design** in the **Processes for Source lab3** window. This runs all processes. Be patient – this takes a while!

A check in the **Processes for Source** window denotes a process that was run successfully. An exclamation indicates that the process was run and that there is a warning for the process. More information about warnings can be obtained in the **Transcript** window.

6. Timing Analysis

To see the timing report, go to ***Implement Design->Place&Route->Generate Post-Place&Route StaticTiming->Text-based Post-Place&Route Static Timing Report***. The timing report will be shown in the right window similar to Figure B.7. From the timing report, we see that the critical path (i.e. worst case delay) is from C to Carry with a delay of 7.851ns.

```
18
19  INFO:Timing:2698 - No timing constraints found, doing default enumerat
20  INFO:Timing:2752 - To get complete path coverage, use the unconstraine
21     option. All paths that are not constrained will be reported in the
22     unconstrained paths section(s) of the report.
23
24
25  Data Sheet report:
26  -----------------
27  All values displayed in nanoseconds (ns)
28
29  Pad to Pad
30  ---------------+---------------+---------+
31  Source Pad     |Destination Pad|  Delay  |
32  ---------------+---------------+---------+
33  A              |Carry          |    7.817|
34  A              |Sum            |    7.802|
35  B              |Carry          |    7.596|
36  B              |Sum            |    7.575|
37  C              |Carry          |    7.851|
38  C              |Sum            |    7.745|
39  ---------------+---------------+---------+
40
41  Analysis completed Wed Feb 22 15:15:45 2006
42  --------------------------------------------------------------------------
```

Figure B.7: Timing Report of the circuit

7. Timing (post-place-and-route simulation)

Timing simulation uses the block and routing delay information from a routed design to give a more accurate assessment of the behavior of the circuit under worst-case conditions. For this reason, timing simulation is performed after the design has been placed and routed.

In order to simulate the design, a test bench is needed to provide stimulus to the design. You should use the same test bench that was used to perform the behavioral simulation.

To set the simulation process properties:
1. In the Sources in Project window, select the test bench file.
2. In the Processes for Source window, click the + next to ModelSim Simulator to expand the process hierarchy.

To start the timing simulation, double-click **Simulate Post-Place and Route Verilog Model** in the Processes for Source window. ISE will run NetGen to create the timing simulation model. ISE will then call ModelSim and create the working directory, compile the source files, load the design, and run the simulation for the time specified.

To view signals during the simulation, you must add them to the Wave window. ISE automatically adds all the top-level ports to the Wave window. Additional signals are displayed in the Signal window based on the selected structure in the Structure window.
There are two basic methods for adding signals to the Simulator Wave window.
• Drag and drop from the Signal/Object window.
• Highlight signals in the Signal/Object window and then select **Add → Wave → Selected Signals**.

Right-click on the waveforms and select ***Zoom Range***. Choose ***Start*** as 1 ns and ***End*** as 1000 ns. Zoom in on the waveforms and find the exact delay on the longest path.

8. Programming the Digilent Board

1. Turn on your Digilent Board and make sure that the cable is properly connected.
2. Double-click **Generate Programming File** to create a bitstream of this design.
3. The BitGen program creates the design_name.bit bitstream file (in this design, the lab3.bit file). The bitstream file contains the actual configuration data.
4. Double-click on **Configure Device.** This lunches the iMPACT software.
5. Select **Boundary Scan Mode** in the Configure Devices dialog box.
6. Click **Next**.
7. Select **Automatically connect to cable and identify Boundary-Scan chain** in the Boundary Scan Mode Selection dialog box.
8. Click **Finish**.

The configuration file is used to program the device. When the software prompts you to select a configuration file for the device XC3S200:

1. Select the BIT file from your project working directory.
2. Click **Open**.
3. If the software gives the option of assigning a new configuration file, Click **Cancel.**
4. Right-click on the XC3S200 device
5. Select **Program** from the right-click menu.
6. Uncheck **Verify**. Click **OK.**

When the Program operation completes, a large blue message appears showing that programming was successful.

Your design has now been programmed. The board should now be working and should allow you to use the switches for providing the inputs to the adder. Two LEDs should show the CARRY and SUM bits.

C: Verilog HDL Basics Guide

1. Introduction

Verilog HDL is a **Hardware Description Language (HDL)**. A Hardware Description Language is a language used to describe a digital system, for example, a computer or a component of a computer; for example a microprocessor, and adder, or simply a flip flop.

Just like with schematic based hardware design, Verilog allows us to describe a digital logic system based on a structure of wires, gates, and registers using a systematic language. This language is unlike most other programming languages, where they read like steps in a recipe. Instead, Verilog is written so that most components respond in parallel, simultaneously.

Using an HDL like Verilog, one may describe a digital system at several levels of detail. For example, we may describe the layout of the wires, resistors and transistors on an **Integrated Circuit (IC)** chip, i.e. the **switch level**. Or, it might describe the logical gates, flip flops and their interconnecting wires in what is called a **net-list** – this is circuit description at the **gate level**. An even higher level describes the registers and the transfers of information, via buses and wires, between registers. This is called the **Register Transfer Level (RTL)**.

2. Why Use Verilog HDL?

Digital systems are highly complex. At their most detailed level, they may consist of millions of elements, i.e. transistors or logic gates. Therefore, for large digital systems, gate-level design is very difficult to achieve in a short time. For many decades, logic schematics served as the primary method for design entry, but not any more. Today, hardware complexity has grown to such a degree that a schematic with logic gates is almost useless as it shows only a web of connectivity and not the functionality of design. Since the 1970s, Computer engineers and electrical engineers have moved toward hardware description languages (HDLs). The most prominent modern HDLs in industry are Verilog and VHDL. Verilog is the top HDL used by over 10,000 designers at such hardware vendors as Sun Microsystems, Apple Computer and Motorola.

The Verilog language provides the digital designer with a means of describing a digital system at a wide range of levels of detail, and, at the same time, provides access to computer-aided design tools to aid in the design process at these levels.

Our goal in this course is not to create VLSI chips, or even to completely master the language, such that we can describe the functionality of virtually *any* digital system, but instead to obtain a basic understanding of the HDL based digital logic design process.

3. The Verilog Language

There is no attempt in this handout to describe the complete Verilog language. It describes only the portions of the language needed to allow students to explore the architectural aspects of computers. In fact, this handout covers only a small fraction of the language. For the complete description of the Verilog HDL, consult the references at the end of the handout.

3.1 Modules, and Structural Design in Verilog

The Verilog language describes a digital system as a set of **module**s. Each of these **module**s has an interface to other **module**s to describe how they are interconnected. In this way, **module**s can be used to describe an abstract, high-level net-list. The modules may run concurrently, but usually we have one top level module which specifies a closed system containing both test data and hardware models. The top level **module** invokes 'instances' of other **module**s. An 'instance' of a **module** is essentially an invoked copy of any module. In the same way that in schematic design multiple copies of a hardware block may be invoked, it is possible to 'instantiate' multiple copies of the same **module.** In order to differentiate between multiple 'instantiations' of a **module**, each instantiation is assigned a unique name.

Modules can represent pieces of hardware ranging from simple gates to complete systems, e. g., a microprocessor. Modules can either be specified behaviorally or structurally (or a combination of the two). A **behavioral specification** defines the behavior of a digital system (module) using traditional programming language constructs, e.g. 'if' and assignment statements. A **structural specification** expresses the behavior of a digital system (module) as a hierarchical interconnection of sub modules. At the bottom of the hierarchy the components must be primitives or specified behaviorally. Verilog primitives include gates, like AND, OR, NOT, NAND and NOR, as well as pass transistors (switches).

A structural specification of digital logic in Verilog is very similar to its schematic based design. Each object in a schematic design can be represented in Verilog as a module. If an object in a schematic is built up of multiple other schematic objects, each of those objects can in turn be represented by a Verilog module, which is instantiated within the original module. The semantic structure of a module is given in Figure C.1:

```
module <module name> (<port list>);
        <declares>
        <module items>
endmodule
```

Figure C.1: Parametric definition of a Module

The **<module name>** is an identifier that uniquely names the module. The **<port list>** is a list of input, inout and output ports which are used to connect to other modules. Both of

these fields are similar to the names and I/Os of objects in a schematic design. The **<declares>** section specifies data objects as registers, memories and wires as well as procedural constructs such as **function**s and **task**s, that will be used to implement the functionality of the module. The **<module items>** may be **`initial`** constructs, **`always`** constructs, continuous assignments or instances of other, lower level modules.

The semantics of the **module** construct in Verilog is very different from subroutines, procedures and functions in other languages. A module is never called! A module is instantiated at the start of the program and stays around for the life of the program, just as if it were a schematic object connected to other schematic objects. A Verilog module instantiation is used to model a hardware circuit where we assume no one unsolders or changes the wiring. Each time a module is instantiated, we give its instantiation a name. For example, **NAND1** and **NAND2** are the names of instantiations of our **NAND** gate in Figure C.2.

```
// Model of a Nand gate
module NAND(in1, in2, out);

  input in1, in2;
  output out;

  assign out = ~(in1 & in2);          // continuous assign statement
                                      // for details of operators used,
                                      // please see Section 3.2
endmodule
```

Figure C.2: Description of a NAND gate

The ports **in1**, **in2** and **out** are 'names' of wires, much like wire names in schematic designs. The Continuous Assignment **assign** continuously watches for changes to variables in its right hand side and whenever that happens, the right hand side is re-evaluated and the result immediately propagated to the left hand side (**out**). Continuous Assignment statement is used to model combinatorial circuits. A structural specification of a module **AND** obtained by connecting the output of one **NAND** to both inputs of another one is shown in Figure C.3.

```
// Structural model of AND gate from two NANDS
module AND(in1, in2, out);

  input in1, in2;
  output out;
  wire w1;

  NAND NAND1(in1, in2, w1);    // two instantiations of the module NAND
  NAND NAND2(w1, w1, out);
endmodule
```

Figure C.3: AND gate constructed from two NAND gates

This module has two instances of the **NAND** module called **NAND1** and **NAND2** connected together by an internal wire **w1**. The general form to invoke an instance of a module is:

```
<module name> <parameter list> <instance name> (<port list>);
```

where **<parameter list>** are optional values of parameters passed to the instance. An example parameter passed would be the delay for a gate. As a final example, here is the implementation of a half adder circuit using the NAND module described above:

```
module HalfAdd(X, Y, C, S)

  input X, Y;
  output C, S;
  wire S1, S2, S3;

  NAND NANDA (X, Y, S3);
  NAND NANDB (X, S3, S1);
  NAND NANDC (S3, Y, S2);
  NAND NANDD (S1, S2, S);
  assign C = S3;
endmodule
```

Figure C.4: Module for a half-adder

Below is the schematic representation of the above 'HalfAdd' module, showing the result of the instantiations of the NAND modules:

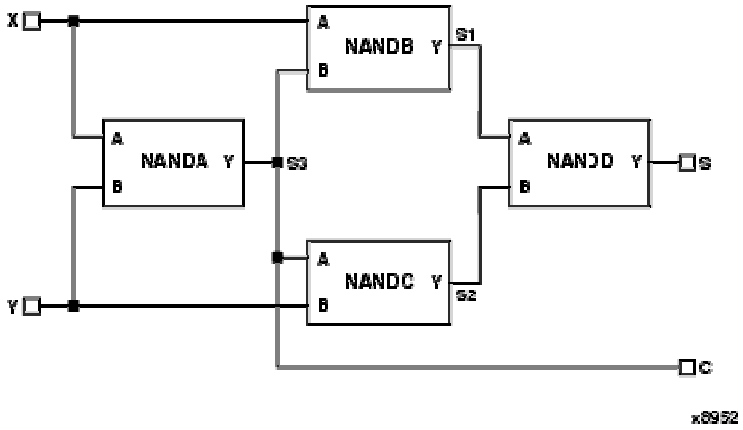

Figure C.5: A Schematic representation of the half-adder module

3.1.1 Verilog Primitive Modules

Figure C.4 above describes the implementation of a NAND gate using continuous assignment statements, while Figure C.5 describes the functionality of an AND gate, implemented by instantiating two of the NAND gates described before. This may give the impression that the functionality of even the basic logic gates needs to be described as modules. However, Verilog provides built in modules for most of the basic gate functionalities, that need not be defined or described, and can simply be instantiated directly, wherever needed. However, the defined terminal order for the outputs and inputs of these primitive modules must be respected. A list of some of the most commonly required primitive modules is given below:

Gate Type		Terminal Order
And Or Xor	Nand Nor xnor	(1_output, 1-or-more_inputs)
Buf	Not	(1-or-more_outputs, 1_input)

Thus it is possible to instantiate a 3 input **and** gate, or a 4 input **xnor** gate, without having to declare and define a module for them. As an example, Figure C.6 below describes the functionality of the module 'HalfAdd', using these primitive modules only.

```
module HalfAdd(X, Y, C, S)

  input X, Y;
  output C, S;
  //wire S1, S2, S3;

  and AND_1 (C, X, Y);
  xor XOR_1 (S, X, Y);
endmodule
```

Figure C.6: Timing Report of the circuit

In the above example, two primitive modules are instantiated, an **and** gate instantiation named **AND_1** to implement the Carry output, and a **xor** gate instantiation named **XOR_1** to implement the Sum output of the Half adder.

3.2 Verilog Basics

3.2.1 Primitive Operators

In this section, we introduce the basic Verilog primitive operations that will be needed throughout the later COE-203 lab sessions. Although there are several different types of operators, we are chiefly concerned with the following types:

- Bitwise Logical operators,
- Arithmetic and shift operators,

For a more comprehensive treatment of Verilog operators, please refer to [1].

3.2.1.1 Bitwise Operators

Bitwise operators operate on the bits of the operand or operands. For example, the result of A & B is the AND of each corresponding bit of A with B. Operating on an unknown (**x**) bit results in the expected value. For example, the AND of an **x** with a FALSE (0) is an **x**. The OR of an **x** with a TRUE (1) is a TRUE (1).

Operator	**Name**	**Comments**
~	Bitwise negation	
&	Bitwise AND	
|	Bitwise OR	
^	Bitwise XOR	
~&	Bitwise NAND	
~|	Bitwise NOR	
~^ or ^~	Equivalence Bitwise NOT XOR	

the above bitwise logical operations are equally valid for multi-bit variables (e.g. two 8-bit busses) as well as single bit variables.

3.2.1.2 Arithmetic and Shift Operators

Arithmetic and Shift operators perform the specified function on their operand(s).

Operator	**Name**	**Comments**
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	Divide by zero produces '*x*', i.e. don't care.
%	Modulus	
<<	Left Shift	e.g. Y = A >> 3, where Y and A are 8-bit variables
>>	Right Shift	

The arithmetic primitives operate on two variables, which can be *wire*, *reg* or *integer* type – the former two always being treated as unsigned.

3.2.2 Verilog Data Types

3.2.2.1 Physical data types

Since the purpose of Verilog HDL is to model digital hardware, the primary data types are for modeling registers (**reg**) and wires (**wire**). The **reg** variables store the last value that was procedurally assigned to them whereas the **wire** variables represent physical connections between structural entities such as gates. A **wire** does not store a value. A

wire variable is really only a label on a wire. The **reg** and **wire** data objects may have the following possible values:

- Logical zero (**0**)
- Logical onezero (**1**)
- Unknown logical value (**x**)
- High Impedance of Tristate Gate (**z**)

The size of a register or wire may be specified in the variable declaration. For example, the declarations

```
reg [0:7] A, B;
wire [0:3] Dataout;
reg [7:0] C;
```

specify registers A and B to be 8-bit wide with the most significant bit the zeroth bit, whereas the most significant bit of register C is bit seven. The wire 'Dataout' is 4 bits wide.

The bits in a register or wire can be referenced by the notation **[<start-bit>:<end-bit>]**. For example, in the second procedural assignment statement

```
begin
      A = 8'b01011010;
      B = {A[0:3] | A[4:7], 4'b0000};
end
```

B is set to the first four bits of A bitwise or-ed with the last four bits of A and then concatenated with 0000. B now holds a value of 11110000. The **{}** brackets means the bits of the two or more arguments separated by commas are concatenated together.

3.2.2.2 Abstract Data types

In addition to modeling hardware, there are other uses for variables in a hardware model. For example, the designer might want to use an **integer** variable to count the number of times an event occurs. For the convenience of the designer, Verilog HDL has several data types which do not have a corresponding hardware realization. These data types include **integer** and **real**, which behave pretty much as in other languages, e. g., C or Java. Be warned that a **reg** variable is unsigned and that an **integer** variable is a signed 32-bit integer. This has important consequences when performing subtraction.

3.3 Procedural Statements and RTL Level Design

Up until now, we were studying what is considered Structural Design using Verilog: interconnecting components using module instantiations, and describing combinational logic using 'continuous assignment statements'. However, as designs grow in complexity, it becomes evident that a schematic based design entry tool would be much

more effective at describing digital logic instead of using structural Verilog. So why use Verilog in the first place?

The beauty of Verilog lies in the fact that it can be used to describe designs at higher levels of 'abstraction' than the gate-level, or even the structural level. This allows us to describe a lot of logic using only a few lines of code.

Most commercially available synthesis tools (Such as the one you are using for your Xilinx FPGA) expect to be given a design description in RTL form. RTL is an acronym for *register transfer level*. This implies that your Verilog code describes how data is transformed as it is passed from register to register. The transforming of the data is performed by the combinational logic that exists between the registers. Don't worry! RTL code also applies to pure combinational logic - you don't have to use registers. To show you what is meant by RTL code, let's consider the simple example of the half-adder module that was described structurally in Figures C.4 and C.6. An implementation of the *'HalfAdd'* module using bitwise operators to implement the Boolean logic, instead of instantiating multiple NAND modules is shown in Figure C.7 below:

```
module HalfAdd(X, Y, C, S)

  input X, Y;
  output C, S;
    //wire S1, S2, S3; (we don't need these wires anymore!)

  assign C = (X & Y);
  assign S = (X ^ Y);
endmodule
```

Figure C.7: Half-adder module using bit-wise operators

As we can see, using the Boolean logic operators, we have written continuous assignment statements that describe the *'HalfAdd'* module in RTL form. In fact even module instances are examples of synthesizable RTL statements – thus structural Verilog is a valid subset of RTL level design description.

However, one of the reasons to use synthesis technology is to be able to describe the design at a higher level of abstraction than using a collection of module instances or low-level binary operators in a continuous assignment. We would like to be able to describe ***what*** the design does and leave the consideration of ***how*** the design is implemented up to the synthesis tool. This is a first step (and a pretty big conceptual one) on the road to high-level design. We are going to use a feature of the Verilog language that allows us to specify the functionality of a design (the *'what*') that can be interpreted by a synthesis tool.

3.3.1 The *'always'* Block

'always' blocks are 'procedural' blocks that contain sequential statements. This means that their contents can be written pretty much like writing procedures in any software programming languate like Java or C, except that the contents of an *'always'* block are

always available for execution (remember, we are implementing hardware, not software, so once built, the hardware will *always* be there, waiting to be used).

```
always @(<sensitivity-list>)
begin
  // 'procedural'statements
end
```

Figure C.8: Basic format of an *always* block

This means that the statements inside an *'always'* block are executed not only up until the closing 'end' statement, but can be executed again. This means that a way of controlling execution through an always block is required. In describing synthesizable designs, a '*sensitivity list*' is often used to control execution. The basic format of an *'always'* block is given in Figure C.8.

The sensitivity list consists of one or more signals. When at least one of these signals changes, the always block executes through to the end keyword as before. Except that now, the sensitivity list *prevents* the always block from executing again until another change occurs on a signal in the sensitivity list.

The statements inside the always block describe the functionality of the design (or a part of it). Let's consider another version of the *HalfAdd* module as given in Figure C.9:

```
module HalfAdd(X, Y, C, S)

  input X, Y;
  output C, S;
  wire S1, S2, S3;

  always @(<sensitivity-list>)
  begin
    C <= (X & Y);
    S <= (X ^ Y);
  end
endmodule
```

Figure C.9: Half-adder module using procedural assignment

Instead of a continuous assignment, we now have a *procedural* assignment to describe the functionality of the H*alfAdd* module. Notice that the sensitivity list isn't valid Verilog code. We need to create a meaningful sensitivity list. How do we decide when to execute the always block? Perhaps a better question is what do we need to do in order to have C and S change value? Answer: C and S can only change when at least one of X or Y changes. After all, these are the two inputs to the half adder. This is our sensitivity list:

```
module HalfAdd(X, Y, C, S)

  input X, Y;
  output C, S;
  wire S1, S2, S3;

  always @(X or Y)
  begin
    C <= (X & Y);
    S <= (X ^ Y);
  end
endmodule
```

Figure C.10: Half-adder module using *always* block

In Figure C.10, we have simply replaced the continuous assignments of the *'HalfAdd'* module with equivalent procedural assignments, using the *'always'* statement. However, the RTL design methodology is much more capable than that, as we will see in the next subsection.
The sensitivity list of an always statement can also include the special keywords *'posedge'* and *'negedge'* which when associated with a signal in the sensitivity list, make the logic within the *always* block edge sensitive to that signal, as opposed to the regular level sensitivity. These keywords, combined with the *always* statement, are very useful in implementing synchronous, or clock driven circuits.

3.4 Control Constructs in Verilog – *'if'* and *'case'* Statements

Verilog HDL has a rich collection of control statements which can used in the procedural sections of code, i. e., within an **initial** or **always** block. Most of them will be familiar to the programmer of traditional programming languages like C. The main difference is instead of C's { } brackets, Verilog HDL uses **begin** and **end**. In Verilog, the { } brackets are used for concatenation of bit strings.

3.4.1 The *'if'* Statement

The *'if'* statement in Verilog is a procedural statement that conditionally executes other procedural statements depending upon the value of some condition. An *'if'* statement may optionally contain an *'else'* part, executed if the condition is false. Although the else part is optional, for the time being, we will code up *'if'* statements with a corresponding else rather than simple if statements. In order to have more than one sequential statement executed in an *'if'* statement, multiple statements are bracketed together using the *'begin...end'* keywords. An example of the usage of the *'if'* statement is given in Figure C.11 below.

```
 reg f, g;      // a new reg variable, g
                // variables f and g could be any other kind of output,
                // like the input ports of an instantiated module, or
                // simply the output ports of the current module, etc.

  always @(sel or a or b)
  begin
    if (sel == 1)
      begin
        f = a;
        g = ~a;
      end
    else
      begin
        f = b;
        g = a & b;
      end
  end
```

Figure C.11: An example of an *if* statement

'if' statements are synthesized by generating a multiplexer for each variable assigned within the *'if'* statement. The select input on each mux is driven by logic determined by the *'if'* condition, and the data inputs are determined by the expressions on the right hand sides of the assignments.

3.4.2 The *'case'* Statement

The *'case'* statement represents a multi-way branch, unlike the *'if'* statement, which is only a two-way branch. The basic structure of the *'case'* statement is given in Figure C.12

```
case (<expression>)
  <value1>: <statement>
  <value2>: <statement>
      .
      .
      .
  default: <statement>
endcase
```

Figure C.12: Structure of a *case* statement

Unlike the *'case'* statement in C, the first **<value>** that matches the value of the **<expression>** is selected and the associated statement is executed then control is transferred to after the *'endcase'*, i.e. no *'break'* statements are needed as in C.

As an example of the use of *'case'* statements, consider the implementation of a basic arithmetic unit, given in the figure below (Truth Table is in figure C.13a, while Verilog Code is in Figure C.13b):

Logic Unit				
0	0	0	Y <= A & B	Bitwise AND of A and B
0	0	1	Y <= A | B	Bitwise OR of A and B
0	1	0	Y <= A ^ B	Bitwise XOR of A and B
0	1	1	Y <= ~A	Bitwise NOT of A (1's complement)
1	0	0	Y <= A ~& B	Bitwise NAND of A and B
1	0	1	Y <= A ~| B	Bitwise NOR of A and B
1	1	0	Y <= 0	Transfer 0
1	1	1	Y <= 0	Transfer 0

Figure C.13a: Truth table of an Arithmetic Unit

```
module Logic (A, B, Y, Sel)

  input  [7:0] A, B;  // 8-bit input busses
  input  [2:0] Sel;
  output [7:0] Y;

  reg    [7:0] Y;     // the output is declared to be of type 'reg'


  always @(A or B or Sel)
  begin
    //------------
    // Logic Unit
    //------------
    case (Sel[2:0])
      2'b 000 : Y = A & B;     // A and B
      2'b 001 : Y = A | B;     // A or B
      2'b 010 : Y = A ^ B;     // A xor B
      2'b 011 : Y = ~A;        // 1's complement of A
      2'b 100 : Y = ~(A & B);  // A nand B
      2'b 101 : Y = ~(A | B);  // A nor B
      default : Y = 0;         // note that the last two bit
                               // combinations are not included, they
                               // are taken care of by the 'default'
                               // statement.
    endcase
  end
endmodule
```

Figure C.13b: Verilog Code for the truth table in C.13a

In Verilog, a branch for every case choice value is not needed, so the **default** clause is always optional. However if the default clause is omitted a latch will always be inferred. The reason for this is that when the **default** statement is missing, the Verilog compiler assumes that not all possible combinations of the output have been specified. Thus in order to prevent unexpected output values from being generated, the compiler generates a

latch to hold the output value during transitions of the input from one know input combination to the next.

This happens even if the **case** statement already has an output signal explicitly assigned in what is thought to be all branches covering all case choice values. The reason for this is that although all case conditions may be thought of as being covered, every possible combination of these values: X (don't care), 1, 0, Z (High Impedance), is almost always not covered for all **case** choice values.

Thus in order to prevent the unintentional inference of a latch in your circuit, introducing unnecessary delay, it is recommended that a **default** value always be specified for a given output.

3.4.3 Differences between *'if'* and *'case'* statements

The *'if'* statement generally produces priority-encoded logic and the *'case'* statement generally creates balanced logic. An *'if'* statement can contain a set of different expressions while a *'case'* statement is evaluated against a common controlling expression. In general, the *'case'* statement is used for complex decoding while the *'if'* statement is used for speed critical paths.

3.5 Modeling Finite State Machines

Designers of digital circuits are invariably faced with needing to design circuits that perform specific sequences of operations, for example, controllers used to control the operation of other circuits. Finite State Machines (FSMs) have proven to be a very efficient means of modeling the sequencer circuits. By modeling FSMs in hardware description languages such as Verilog, designers can concentrate on modeling the desired functionality of the FSM, without being overly concerned with circuit implementation details – this is left to the synthesis tools.

A FSM is any circuit specifically designed to sequence through specific patterns of states in a predetermined sequential manner, and which conforms to the structure shown in Figure C.14 below.

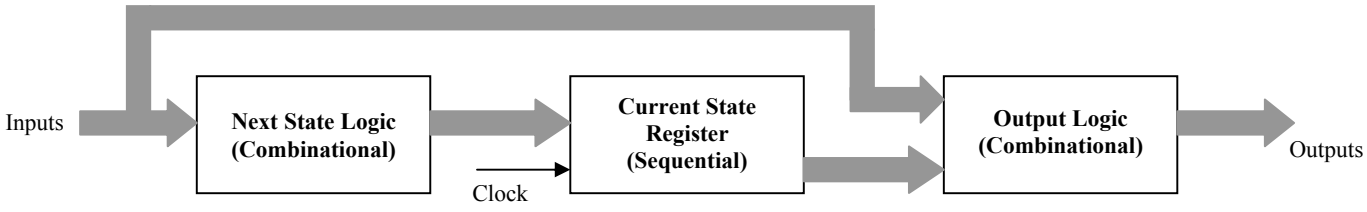

Figure C.14: Block diagram of a sequential circuit

1. Current State Register: Set of Flip-flops used to hold the current state of the FSM. Its value represents the current state in the particular sequence of operations being performed. When operating, it is clocked from a free running clock source.
2. Next State Logic: Combinational Logic used to Generate the Next state in the sequence, The next state output is a function of the state machine's inputs and its current state.
3. Output Logic: Combinatorial logic is used to generate required output signals. For Mealy machines, the state machine outputs depend on the Current state value (i.e. the state register outputs) as well as the state machine inputs. For Moore machines, on the other hand, the FSM outputs are only a function of the Current state.

3.5.1 Modeling FSMs in Verilog

3.5.1.1 Coding Styles and Issues

There are several ways of modeling the same state machine using an HDL like Verilog. Yet even a subtle code change can cause a model to behave differently than expected. The HDL code may be structured into three separate parts representing the three parts of a state machine as depicted in Figure C.14. Alternatively, different combinations of blocks can be combined in the model. Either way, the coding style is independent of the state machine being designed.

For the purpose of this lab, and even in general, it is preferable to model each of the three parts of the state machine separately, in order to keep things simple, easy to understand and debug. This approach is illustrated in the example given in Figure C.15a and C.15b below.

3.5.1.2 Resets and Fail-safe Behavior

There is no way of predicting the initial value of the state register flop-flops when implemented in an IC and 'powered up'. It could become permanently stuck in an un-coded state. In order to ensure that our state machine is always in a known state, or can be put into an initial state, it is necessary to include a *reset* mechanism into the design of the state machine. Resets can be either synchronous or asynchronous:

- Synchronous Reset: in this approach, the reset signal provides an overriding input to the Next-state combinational logic, which when asserted, causes the state machine to enter a known initial state, at the next clock pulse.
- Asynchronous reset: this ensures that the state machine can always be initialized to a known valid state, before the first active clock transition and normal operation commences. The asynchronous resent signal is independent of the clock and can be asserted any time.

Synchronous reset signal can be modeled simply by including its functionality into the combinational *Next State Logic* block. Asynchronous resets, are modeled in Verilog by

using an '*if*' statement inside the '*always*' statement that models the Current State Register of the state machine. The asynchronous reset signal is included in the sensitivity list of this always block with either a '*posedge*' or a '*negedge*' keyword.

For a descriptive illustration of how to model a FSM in Verilog, consider the example state diagram provided in Figure C.15a below, and its Verilog implementation in Figure C.15b:

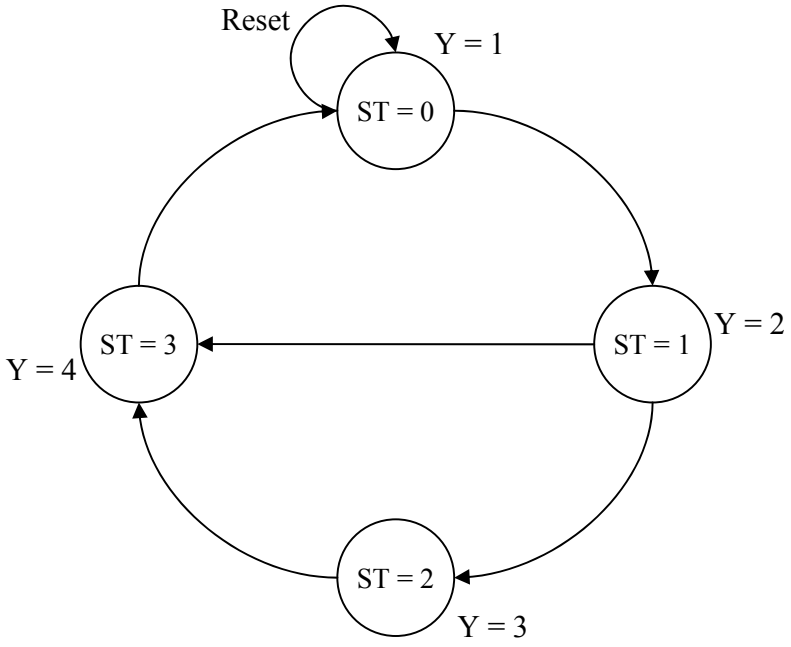

Figure C.15a: An example of a state diagram

```
module Example_FSM (Clock, Reset, Control, Y)
    input         Clock, Reset, Control;
    output  [2:0] Y;
    reg     [2:0] Y;
    reg     [1:0] CurrentState, NextState;


    always @ (Control or CurrentState)    // the next state logic is
        begin:NEXT_STATE_LOGIC            // implemented in this always
            case(CurrentState)            // block.
                2'b 00:
                  begin
                      NextState = 2'b 01;
                  end
                2'b 01:
                  begin
                      if(Control)
                          NextState = 2'b 10;
                      else
                          NextState = 2'b 11;
                  end
                2'b 10:
                  begin
                      NextState = 2'b 11;
                  end
                2'b 11:
                  begin
                      NextState = 2'b 00;
                  end
                default: NextState = 2'b 00; // default clause helps avoid
            endcase                          // inference of unnecessary
        end                                  // latches at the output of
                                             // the block

    always @ (posedge Clock or posedge Reset)      // this always block
        begin:CURRENT_STATE_REGS_AND_ASYNCH_RESET  // implements the
            if(Reset)                              // next state regs
          CurrentState = 0;                        // and asynchronous
            else                                   // reset
                CurrentState = NextState;
        end

    always @ (CurrentState)         // this always block implements the
        begin:OUTPUT_LOGIC          // output logic signal, using case
            case(CurrentState)      // statements
                0: Y = 1;
                1: Y = 2;
                2: Y = 3;
                3: Y = 4;
                default: Y = 1;
            endcase
        end

endmodule
```

Figure C.15b: Verilog code for describing the state diagram of Figure C.15a

Bibliography

[1] Verilog Cookbook.

D: Switch Debouncing

A switch is a mechanical device and as such is much slower than an electronic circuit. When a switch is opened or closed the mechanical contacts do not break or make a connection instantaneously, but can ``bounce" between open and closed, thus making several transitions. If you were to use a mechanical switch to increment a counter (to count, say, people going through a turnstile), a single closure of the switch could increment the counter many times. The bouncing typically continues for about 10 ms.

One of the ways of dealing with the bouncing problem is to use an S'R' Latch as shown in Figure D.1.

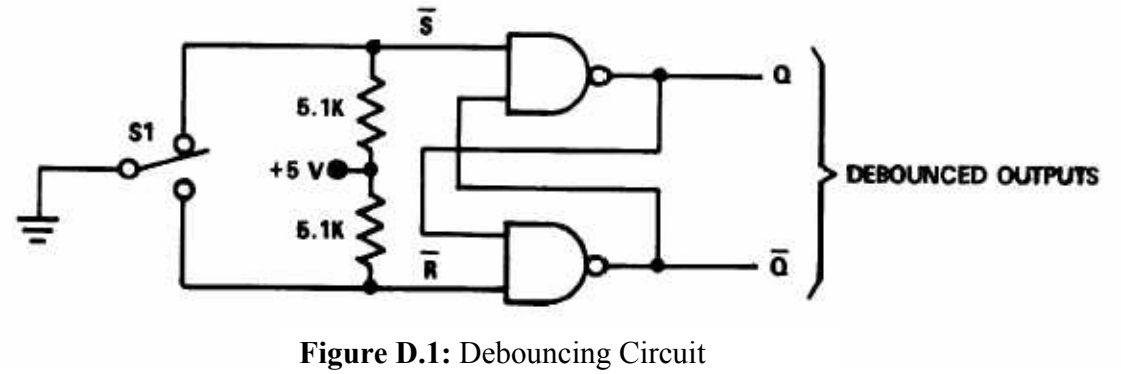

Figure D.1: Debouncing Circuit

When the switch is in the bottom position, the *R'* input on the latch is 0 and $Q = 0$. When the switch is thrown to the top position, the *S'* input on the latch becomes 0, which sets Q to 1. If the switch bounces away from the top position, the inputs to the latch become $R' = S' = 1$ and the value $Q = 1$ is stored by the latch. When the switch is thrown to the bottom position, Q changes to 0 and this value is stored in the latch if the switch bounces.

E: Policies and Procedures

Lab Report Guidelines:
You are responsible for documenting your work and your report must include (at a minimum) the following:

1. Cover sheet (showing your name, ID and Title of the experiment)
2. Introduction - what you did
3. Description of the design - how you did it (You can show the Boolean Equations and write down the steps which you took to reach those equations. List any design aids (such as Logic Works) and how you used them.
4. Implementation of design – which pin numbers you assigned to the inputs and outputs
5. Features of final result – the final design is working properly or not. If not, where do you think the errors lie.
6. Problems Faced
7. Conclusion - Was it a good/bad design/ implementation? **Why?** What would you do differently next time? Any comments on the lab itself are most appreciated.
8. Post-Lab Questions (if any)
9. Include the following in the appendix

    - Schematics – print out the schematic diagram from your xilinx software
    - Simulations - show functional / timing simulation using ModelSim
    - Performance metrics –
        1. Total number of 4-input LUTs used,
        2. Maximum Clock Frequency
        3. Minimum input arrival time before clock
        4. Maximum output required time after clock
        5. Maximum combinational path delay

    .Lab reports **must** be typed and must include enough information to recreate your design. Submit it to the instructor **before** the next lab session. Attach the grading sheet with your lab report at the time of submission.