**Laboratory 5: Half Adder and Full Adder**

**5.1 Objectives**

The objectives of this laboratory are:

* To become familiar with the Xilinx Foundation Series Tools for the design of logic circuits.
* To understand and use Verilog HDL for the design of simple combinational logic circuits.
* To implement simple combinational logic circuits using the Nexys2 FPGA prototyping board.

A hardware description language (HDL) is a method to describe hardware using software. An HDL representation of any hardware block is a software file, which adheres to a specific syntactical format. We will also use a tool called Xilinx Integrated Software Environment (Xilinx ISE) which will help us to convert Verilog codes to fully functional designs on the Xilinx series of Field Programmable Gate Arrays (FPGAs). In this lab, you will design a half adder and a full adder using Verilog, on a Nexys2 board (from Digilent), which contains a Spartan 3E FPGA (from Xilinx). You will also use the I/Os on the Nexys2 board (Figure 5.2) to read in input bits and display the output.

**5.2 Verilog**

Throughout the semester, you will build increasingly complex designs using Verilog, a hardware description language (HDL) widely used to model digital systems. The language supports the design, verification, and implementation of digital circuits at various levels of abstraction. The language differs from a conventional programming language such as C in that the execution of *statements* is not strictly sequential.

A Verilog design can consist of a hierarchy of *modules*. Modules are defined with a set of input, output, and bidirectional ports. Internally, a module contains a list of wires and registers. *Concurrent* and *Sequential* statements define the behavior of the module by defining the relationships between the ports, wires, and registers. *Sequential* statements are placed inside a *begin/end* block and executed in sequential order within the block. But all C*oncurrent* statements and all *begin/end* blocks in the design are executed in parallel. This is the key difference between Verilog and standard programming languages. A module can also contain one or more instances of another module to define hierarchy.

Only a subset of statements in the language is *synthesizable*. If the modules in a design contain only *synthesizable* statements, software tools like Xilinx ISE can be used to *synthesize* the design into a gate-level *netlist* that describes the basic components and connections to be implemented in hardware. The *synthesized* netlist may then be transformed into a *bit-stream* for any programmable logic devices like FPGAs. Note that this enables a significant improvement in designer productivity: a designer writes hardware behavior in synthesizable Verilog and the ISE (or similar) tool realizes this circuit on a hardware platform such as an FPGA. Verilog designs can be written in two forms:

1. **Structural Verilog:** This is a Verilog coding style in which an exact gate-level netlist is used to describe explicit connections between various components, which are explicitly declared (instantiated) in the Verilog code. Structural Verilog is described below, as this lab uses structural Verilog.
2. **Behavioral Verilog:** In this format, Verilog code is written to describe the function of the hardware, without making explicit references to connections and components. A logic synthesis tool is required in this case to convert this Verilog code into gate-level netlists. Usually, a combined coding style is used where part of the hardware is described in structural format and part of the hardware is described in behavioral format according to convenience.

**5.2.1 Structural Verilog Basics**

For this lab, you will use structural Verilog, a limited subset of Verilog that allows you to describe circuits in terms of wires, gates and modules. Specifically, you will be allowed to:

1. Instantiate simple modules.

2. Use the wire construct.

3. Instantiate the primitive gates **and**, **or**, **not**, **xor** (using any number of inputs):

Additionally, although they are not strictly Structural Verilog constructs, you will be using Verilog generate and parameter statements. With regards to using any source to learn Verilog, remember that you are only allowed to instantiate primitive gates, modules, and wires in this lab.

**Wires**

Wires in structural Verilog are analogous to wires in a circuit you build by hand: they are used to transmit values between inputs and outputs. Wires should be declared before they are used:

wire a;

wire b, c;  // declare multiple wires using commas

The wires above are scalar (i.e. represent 1 bit). They can also be vectors:

wire [7:0]   d;  // 8-bit wire declaration

wire [31:0]  e;  // 32-bit wire declaration

Wires can be assigned to other wires, concatenated, and indexed:

wire [31:0] f;

assign f = {d,e[23:0]}; // concatenate d with lower 24 bits of e

In the line above, the brackets [] are used to index a 24-bit range of e and the braces {} concatenate comma-separated wires.

**Gates (Structural Primitives)**

In this lab, you may use the following primitives:  and, or, xor, not, nand, nor, xnor. In general, the syntax is:

operator (output, input1, input2);

For example, the following Verilog statement implements the Boolean equation F = a + b:

wire a, b, F;

/\* … some code that assigns values to a and b \*/

or (F, a, b);

Complex logic functions can be implemented using intermediate wires between these primitive gates.

**Modules**

Modules provide a means of abstraction and encapsulation for your design. They consist of a port declaration and Verilog code to implement the desired functionality. For example, consider a module that computes y = (a + b)(c + d):

module example\_module(a, b, c, d, y);

    // Port and wire declarations:

    input wire a, b, c, d;

    output wire y;

    wire a\_or\_b, c\_or\_d;

    // Logic:

    or   (a\_or\_b, a, b);

    or   (c\_or\_d, c, d);

    and  (y, a\_or\_b, c\_or\_d);

endmodule

There are a few things to note from this example:

1. The ports must be declared as input or output wires, but can be thought of as wires within the module.
2. Wires declared within a module (such as a\_or\_b) are limited in scope to that module.
3. Modules should be created in a Verilog file (.v) where the filename matches the module name (so the above example should be located in example\_module.v).

Then, after creating a module, you can instantiate it in other modules:

example\_module unique\_name(

    .a(a), .b(b), .c(c), .d(d), .y(result));

(Assuming a, b, c, d, and result are valid wires in the module that this instantiation occurs in, and unique\_name is globally unique.)

The syntax .<input/output>(<wire>) is used to explicitly hook up wires to the correct input/outputs of a module. You can also write

example\_module unique\_name(a, b, c, d, result); // correct order

which, while perfectly valid, is not recommended since it is possible to mix up the order of the wires. The first form is also easier to read.

example\_module unique\_name(result, a, b, c, d); // wrong order!

**5.3 Field Programmable Gate Arrays (FPGA)**

A field-programmable gate array is a semiconductor device containing programmable logic components called "logic blocks", and programmable interconnects. Logic blocks can be programmed to perform the function of basic logic gates such as AND, and XOR, or more complex combinational functions such as decoders or mathematical functions. In most FPGAs, the logic blocks also include memory elements like flip-flops. A hierarchy of programmable interconnects allows logic blocks to be interconnected as needed by the system designer, somewhat like a one-chip programmable breadboard. Logic blocks and interconnects can be programmed in the field by the customer or designer (after the FPGA is manufactured) to implement any logical function as and when required hence the name *field programmable logic arrays*.

Realizing a circuit design on an FPGA board consists of three steps, which are performed using a software tool like Xilinx ISE, a tool from Xilinx which integrates various stages of the FPGA design cycle into one software tool:

1. **Synthesis:** This is the process of converting a Verilog description into a primitive gate-level netlist. The final product of the design partitioning phase is a netlist file, a text file that contains a list of all the instances of primitive components in the translated circuit and a description of how they are connected.
2. **Implementation:** 
   1. **Translation:** The translate step takes all of the netlists and design constraints information and outputs a Xilinx NGD (native generic database) file.
   2. **Mapping:** The mapping step maps the above NGD file to the technology-specific components on the FPGA and generates an NCD (native circuit description) file. This is necessary because different FPGAs have different architectures, resources, and components. Among other tasks, it is responsible for the process of transforming the primitive gates and flip-flops in the netlist into LUTs (lookup tables) and other primitive FPGA elements. For example, if you described a circuit composed of many gates, but ultimately of 6 inputs and 1 output, the circuit will be mapped down to a single 6-LUT. Likewise, if you described a flip-flop it will be mapped to a specific type of flip-flop that actually exists on the FPGA.
   3. **Placement:** This step places the mapped components in a manner that minimizes wiring, delay etc. Placement takes a mapped design and determines the specific location of each component in the design on the FPGA.
   4. **Routing:** This step configures the programmable interconnects (wires) so as to wire the components in the design. Because the number of possible paths for a given signal is very large, and there are many signals, this is typically the most time-consuming part.
3. **Programming the FPGA Device:** In this step, the placed and routed design is converted to a bit-stream using the Xilinx ISE tool. The bit-stream generated by the tool (as a .bit file) is loaded on to the FPGA. This bit-stream file programs the logic and interconnects of the FPGA in such a way that the design gets implemented.

Figure 5.1 illustrates the design flow described above.



Figure 5.1. Xilinx Design Flow

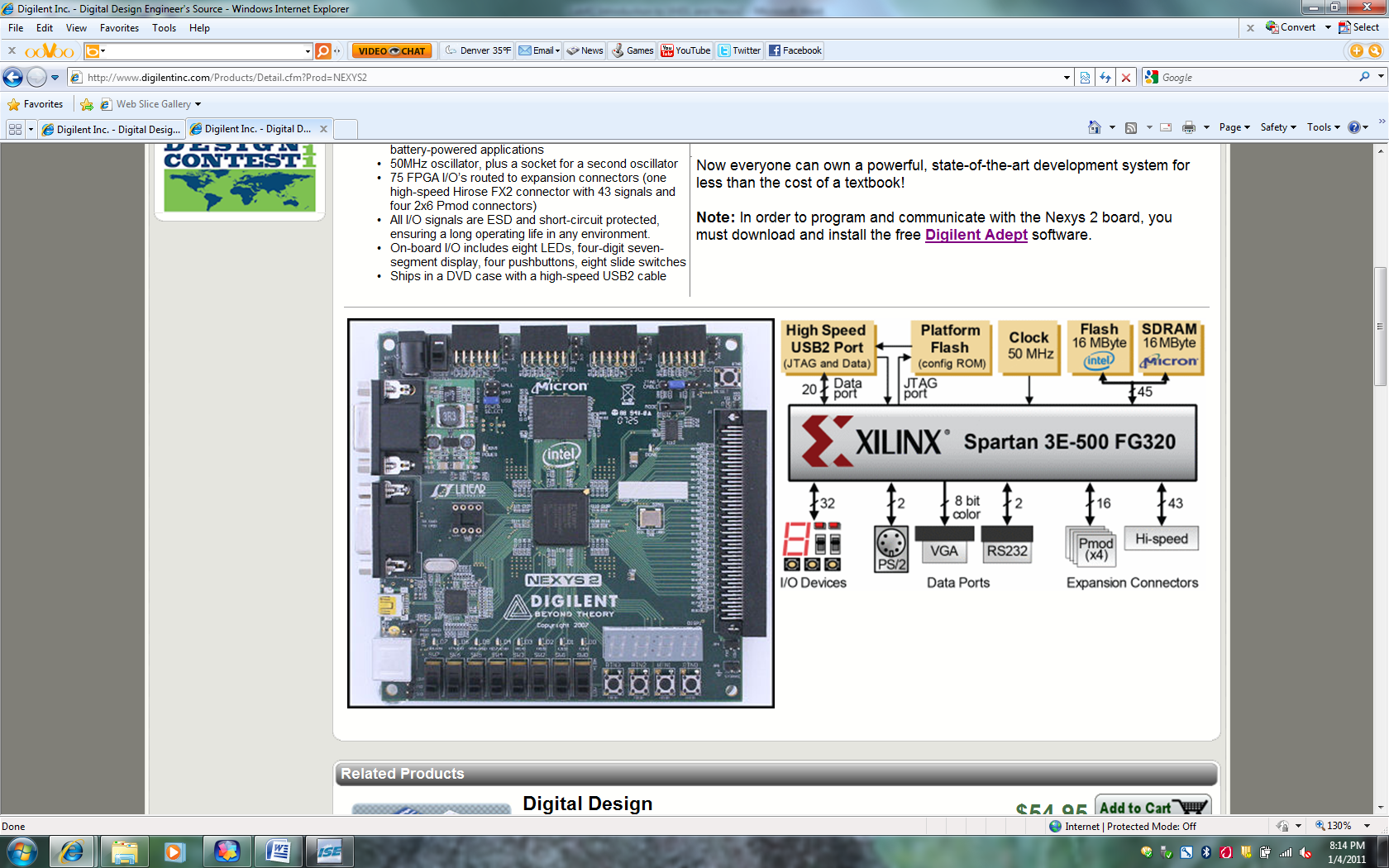
**5.4 Digilent Nexys2 Development Board**

You will build several digital circuits this semester, ranging in complexity from a simple half adder to a simple digital calculator. Digilent's Nexys2 board is the vehicle you will use to implement these circuits. The Nexys2 board is a powerful digital system design platform built around the Xilinx Spartan 3E series of FPGAs. The board has the facility to program the FPGA using a USB connection to your PC. The board provides programmable interfaces to a global reset, four push buttons, a rotational knob, four on/off switches, eight LEDs, clock, memories and the 7-segment displays, as shown in Figure 5.2.A list of the key features and their location on the board is listed below:

PS/2 mouse/keyboard port

On-Board 50 MHz Oscillator

CLK\_50MHz: (B8)

******

USB Connector

Expansion connectors

FPGA IC

Reset Button

Done LED

Power In

Two RS-232 serial ports

Power Switch

Power On indicator LED

Push-Button Switches

Seven Segment Display

8 Toggle Switches [sw(7:0)]

8 LEDs [ld(7:0)]

Figure 5.2: Components on a Nexys2 board [1]



Figure 5.3: Nexys2 inputs and outputs [1]

**Inputs: Slide Switches and Pushbuttons**

The Nexys2 board includes several input devices, output devices, and data ports, allowing many designs to be implemented without the need for any other components. Four pushbuttons and eight slide switches are provided for circuit inputs. Pushbutton inputs are normally low, and they are driven high only when the pushbutton is pressed. Slide switches generate constant high or low inputs depending on their position. Pushbutton and slide switch inputs use a series resistor for protection against short circuits (a short circuit would occur if an FPGA pin assigned to a pushbutton or slide switch was inadvertently defined as an output).

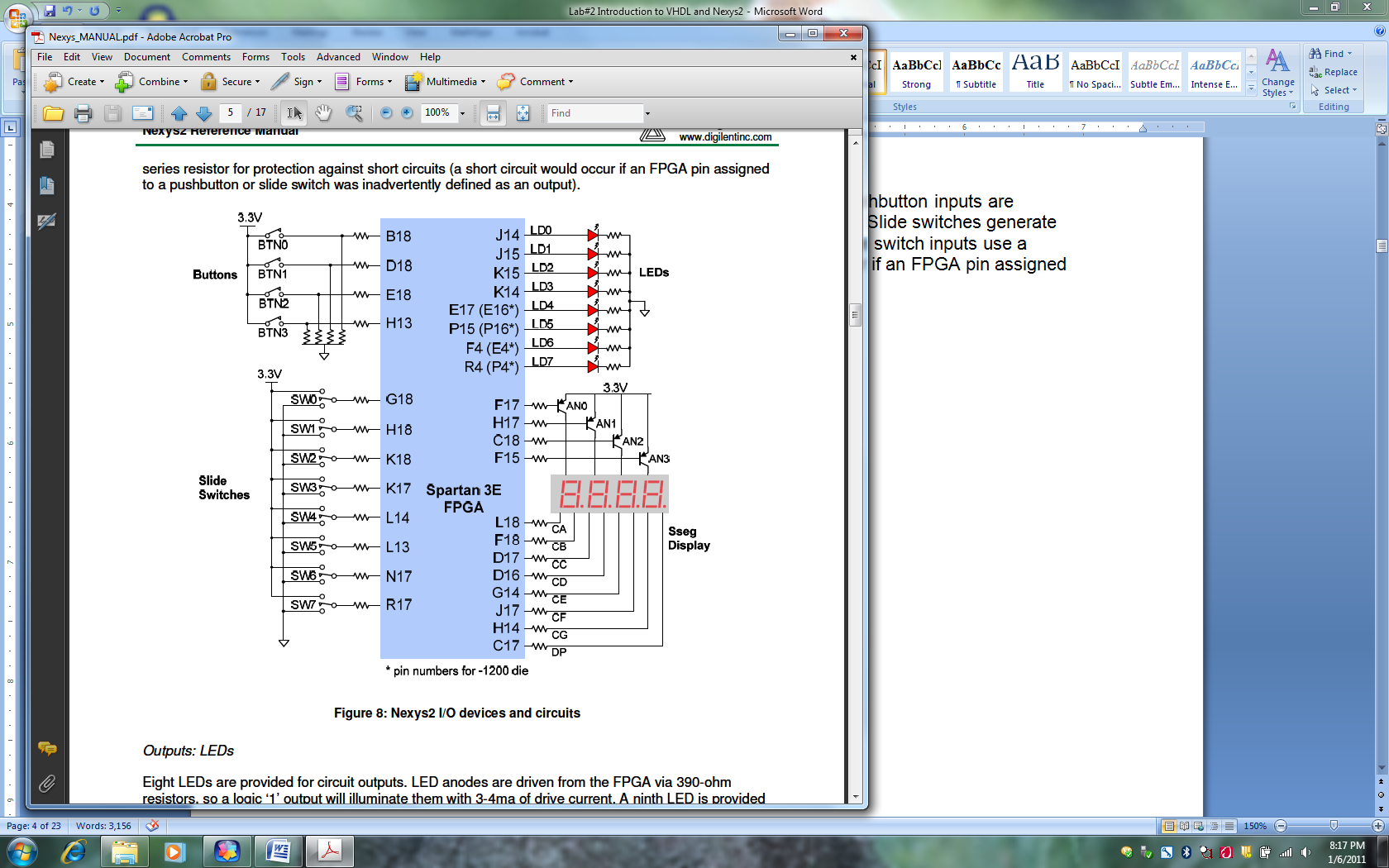


Figure 5.3: Nexys2 I/O devices and circuits [1]

Please refer the reference manual for any additional information:

Digilent Nexys2 Board Reference Manual <http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf>

**5.5 Pre-Lab**

Please make sure to complete the prelab before you attend your lab section. The lab will be long and frustrating if you do not do the prelab ahead of time. In this lab, you will use Verilog to implement a half adder and a full adder on the Xilinx Spartan 3E FPGA. The Verilog used to describe the adder will be in structural format, containing the exact gate-level netlist of the design.

**5.5.1 Pre-Lab Procedure**

**Part 1: Tutorials**

1. Go over Xilinx Tutorial 1.
2. Go over Xilinx Tutorial 2.

**Part 2: Half Adder**

1. Specification. Design a 1-bit Half Adder circuit with two inputs a and b, and two outputs s and c\_out. All signals should use positive logic.
2. Define Inputs and Outputs. Draw a block diagram of the circuit showing the inputs and outputs for the circuit.
3. Draw a truth-table that shows both outputs for the two inputs.
4. The Half Adder circuit given in Figure 3-1 (on page 40 of the Ciletti book) uses a 2-input EXOR gate and a 2-input AND gate.
5. Create Verilog code. Using the **Create** **New Project** Wizard, specify (create) the working directory for this lab (you could name the directory lab5) and name this project *FullAdder* (or something similar). Create a Verilog HDL File and write the Verilog code that implements the minimal expressions. Save the file with the name *HalfAdder.v* (or similar as appropriate). Submit a copy of your code in your prelab report. Leave the project open.
6. Functional Simulation. Perform a functional simulation of the circuit to verify that it is working correctly.
7. Create Symbol. Create a symbol for the half adder to use in the graphical editor. This creates a symbol file that is a Graphic File and can be viewed and edited by opening it.

**Part 3: Full Adder**

1. Specification. Design a 1-bit Full Adder circuit with three inputs a, b, and c\_in, and two outputs s and c\_out. All signals should use positive logic. A full adder can be built from two half adders and a 2-input OR gate.
2. Define Inputs and Outputs. Draw a block diagram of the circuit showing the inputs and outputs for the circuit.
3. In your open project create a new source (Project > New Source) and create a Verilog module for the Full Adder that uses two instantiations of the existing Half Adder. You may follow the format in Figure 3-3 (on page 43) of the Ciletti book with one exception. Xilinx requires you to use the I/O designation for a nested module that is described in Figure 3-7 (on page 115) of the Ciletti book. So, for M1 in the example of Figure 3-3:

Add\_half\_0\_delay M1 (.sum(w1), .c\_out(w2), .a(a), .b(b));

1. Save the file with the name *FullAdder.v* (or similar as appropriate). Submit a copy of your code in your prelab report.
2. Functional Simulation. Perform a functional simulation of the circuit. Paste the results in your prelab report.
3. Create Symbol. Create a symbol for the full adder to use in the graphical editor.
4. Bring your Verilog code in a flash drive.

**5.5.2 Pre-lab Questions**

1. Write a Verilog declaration of the following nets: (a) a 32-bit net having name data\_bus and type wire; (b) scalar nets clock, set, and reset.
2. Write a Verilog declaration of the following registers: (a) a 32-bit register having name operand\_I and type reg; (b) an integer having name K; (c) a 32 × 64 two-dimensional array of 16-bit words having name Pixel\_Color.
3. What characters does Verilog allow in an identifier?
4. Answer T (true) or F (false):
   1. Verilog is case sensitive.
   2. Built-in primitives describe only combinational logic.
   3. Verilog does not have user-defined data types.
   4. The right arithmetic shift operator is denoted by >>.
   5. The output ports of a module must be listed first in the list of ports.
   6. The type of the input port of a module must be a net.
   7. Declaring a module within another module creates a design hierarchy.
   8. All UDPs (user-defined primitives) have scalar outputs.
   9. The truth table of a UDP may have multiple edge transitions in a row if the shorthand notation (??) is used.

**5.5.3 Pre-Lab Report**

In your prelab report, include the following:

1. Truth Table, circuit schematic, Verilog program, and simulation resultsfor the **Half Adder**, with all possible values of inputsaandb.
2. Truth Table, circuit schematic, Verilog program, and simulation results for the **Full Adder**, with all possible values of inputsa,b,andc\_in.

Incorrect or incomplete designs and Verilog programs will not receive full credit. **If you have any problems with Verilog syntax and other pre-lab related issues, please resolve them before coming to the lab. Your TA may not be able to help you with these issues during the lab session.**

**5.6 In-Lab Procedure**

Set your full adder Verilog file as the top module of your design and implement the complete design (*synthesize, map,* and *Place & Route*) using the Xilinx ISE tools. Program the FPGA using the bit-stream file which is generated in the process. For checkoff, you will show the TA the following:

1. Show the Half Adder and the Full Adder working on the board.
2. Demonstrate at least one addition operation from the patterns you have used in the test bench. Show that your simulation results above match the observed waveforms on the DLA.
   1. **Post-Lab Report**

Along with your post-lab report, submit the observed waveforms (in separate sheets) for the Half Adder and the Full Adder.

**References**

[1] Digilent Nexys2 Board Reference Manual: <http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf>