Half Adder, Full Adder and Counters

ENEE 245: Digital Circuits and Systems Laboratory

Lab 5

5.1 Objectives

The objectives of this laboratory are:

- To become familiar with the Xilinx Vivado 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 Basys3 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 Vivado 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 Basys3 board (from Digilent), which contains an Artix7 FPGA (from Xilinx). You will also use the I/Os on the Basys3 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 VIVADO 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 VIVADO (or similar) tool realizes this circuit on a hardware platform such as an FPGA. Verilog designs can be written in two forms:

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.

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!
```

5.2.2 Behavioral (Procedural) Verilog Basics

For this lab, you will also use procedural Verilog, to design counters, which allows `if` statements, `for` loops, and `case` statements. All procedural Verilog programs begin with the **`always`** statement:

```
always @(argument)
```

The `@(argument)` is the sensitivity list of the signal. It determines when the block of code will be evaluated. The argument can be **`posedge`** (for positive edge) or **`negedge`** (for negative edge) of a signal, or just the signal name:

```
always @(posedge clock)
always @(S)
```

In the first statement, the procedure is executed every positive edge of the clock (acting like a positive edge triggered flip-flop) and in the second case whenever S changes (allowing combinational like logic). The **or** and **and** operators can be used to combine signals together, as shown below:

```
always @(posedge clock or posedge clr)
always @(S and Q)
```

Parametric Statements

We can use the Verilog parameter statement to design a generic counter with clock as input and output an arbitrary size counter, as shown below.

Note that the sensitivity list of the **always** statement contains the phrase **posedge** `clk` **or posedge** `clr`. This means that the **if** statement within the *always* block will execute whenever either `clr` or `clk` goes high. If `clr` goes high then the output `q[N-1:0]` will go to zero. On the other hand, if `clr` = 0 and `clk` goes high, then the output `q[N-1:0]` will be incremented by 1. Note that this counter counts from 0 to F and then wraps around to 0.

The following Verilog program can be used to generate this counter.

```
// Example 5a: N-bit counter
  module counter
    #(parameter N = 4)
   (input wire clr ,
    input wire clk ,
    output reg [N-1:0] q
   );

// N-bit counter
  always @(posedge clk or posedge clr)
    begin
      if (clr == 1)
        q <= 0;
      else
        q <= q + 1;
    end

  endmodule
```

Note the use of the **`parameter`** statement that defines the counter size `N` to have a default value of 4. This value can be overridden when the counter is instantiated as shown below for an 8-bit counter cnt8.

```
counter #(.N(8))
  cnt8 (.clr(clr),
        .clk(clk),
        .q(q)
  );
```

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 Vivado, 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:

a. **Translation:** The translate step takes all of the netlists and design constraints information and outputs a Xilinx NGD (native generic database) file.

b. **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.

c. **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.

d. **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 Vivado 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.

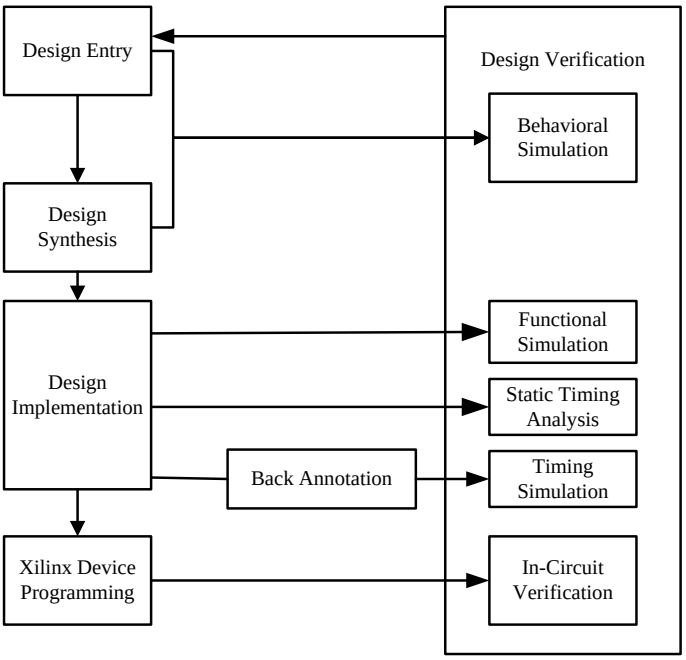


Figure 5.1. Xilinx Design Flow

5.4 Digilent Basys3 Development Board

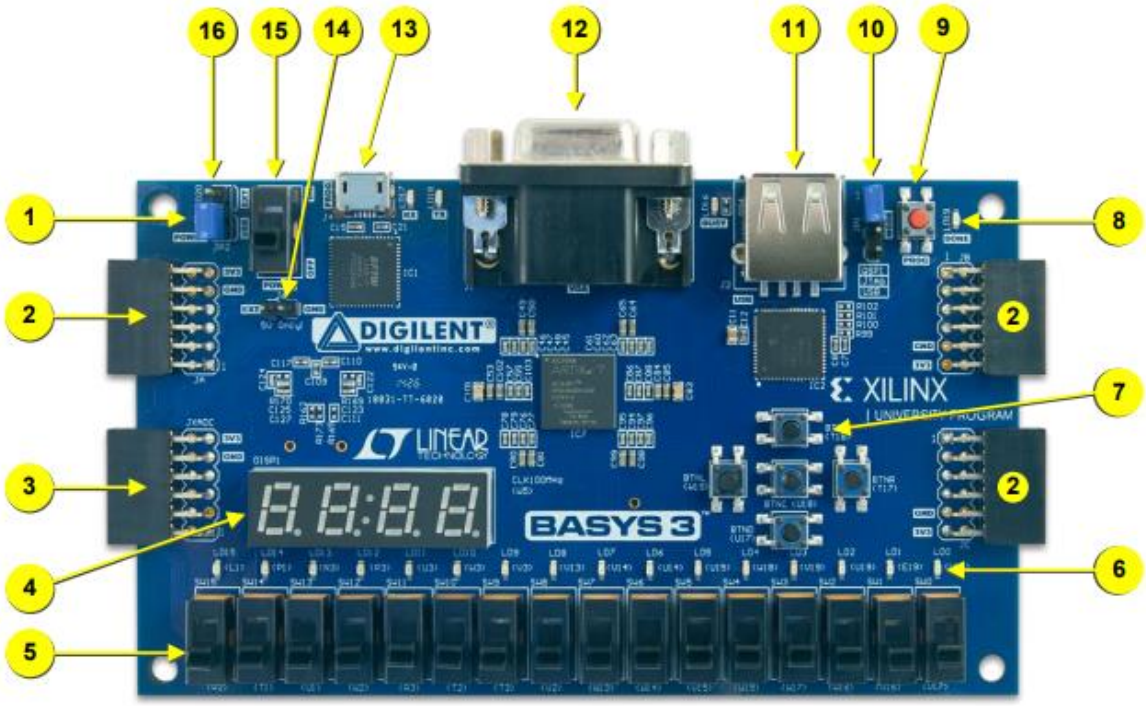
You will build several digital circuits this semester, ranging in complexity from a simple half adder to a simple digital calculator. Digilent's Basys3 board is the vehicle you will use to implement these circuits. The Basys3 board is a powerful digital system design platform built around the Xilinx Artix 7 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, five push buttons, sixteen on/off switches, sixteen LEDs, clock, memories and four 7-segment displays, as shown in Figure 5.3.

Inputs: Slide Switches and Pushbuttons

The Basys3 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).

A list of the key features and their location on the board is listed below:

Figure 5.2: Basys3 inputs and outputs



Callout	Component Description	Callout	Component Description
1	Power good LED	9	FPGA configuration reset button
2	Pmod ports	10	Programming mode jumper
3	Analog signal Pmod port (XADC)	11	USB host connector
4	4 Digit 7-segment display	12	VGA connector
5	Slide Switches (16)	13	Shared UART/JTAG USB port
6	LEDs (16)	14	External power connector
7	Pushbuttons (5)	15	Power switch
8	FPGA programming done LED	16	Power select jumper

Figure 5.3: Components on a Basys3 board

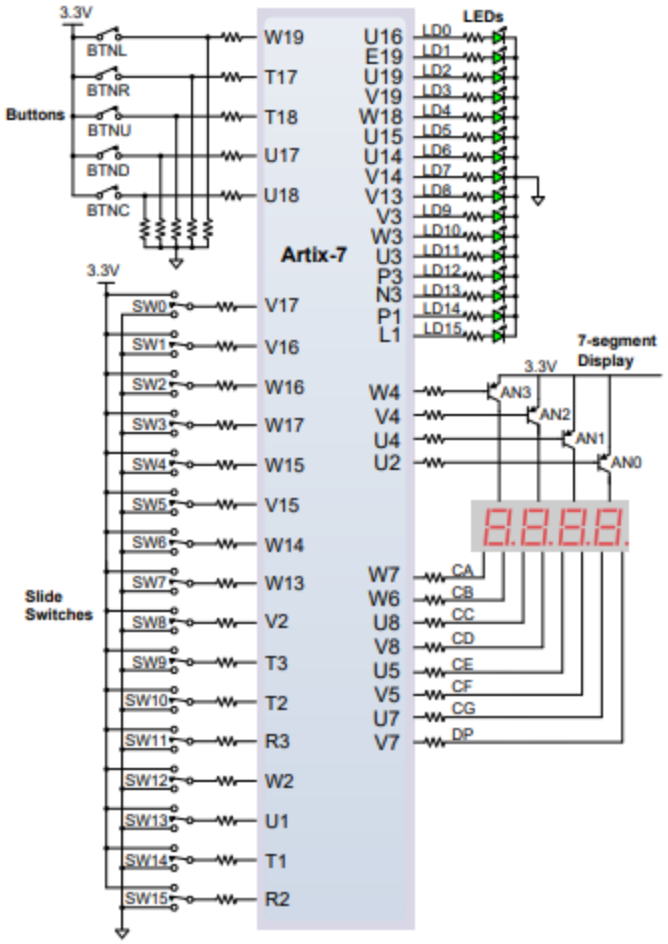


Figure 5.4: Basys3 I/O devices and circuits

In addition to the basic I/O, there are 4 expandable Pmods which can be used to directly input or output signals.

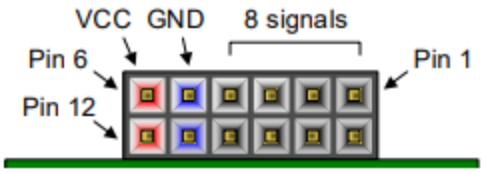


Figure 20. Pmod ports; front view as loaded on PCB.

Pmod JA	Pmod JB	Pmod JC	Pmod XDAC
JA1: J1	JB1: A14	JC1: K17	JXADC1: J3
JA2: L2	JB2: A16	JC2: M18	JXADC2: L3
JA3: J2	JB3: B15	JC3: N17	JXADC3: M2
JA4: G2	JB4: B16	JC4: P18	JXADC4: N2
JA7: H1	JB7: A15	JC7: L17	JXADC7: K3
JA8: K2	JB8: A17	JC8: M19	JXADC8: M3
JA9: H2	JB9: C15	JC9: P17	JXADC9: M1
JA10: G3	JB10: C16	JC10: R18	JXADC10: N1

Table 6. Basys 3 Pmod pin assignment.

Figure 5.5 Basys3 Pmod connectors

Please refer the reference manual for any additional information; it can be found on ELMS. Additionally a generic user constraints file (XDC) for the Basys3 can be found; it specifies all the pin locations available.

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 Artix 7 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. Review Xilinx Tutorial 1.
2. Review Xilinx Tutorial 2.
3. Review Xilinx Tutorial YouTube videos on the last lecture slides

Part 2: Half Adder

4. Design a 1-bit Half Adder circuit with two inputs `a` and `b`, and two outputs `s` and `c_out`.
5. Draw a block diagram of the circuit showing the inputs and outputs for the circuit.
6. Draw a truth-table that shows both outputs for the two inputs.
7. 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 (screenshot of all the code is preferred). Leave the project open.
8. Perform a functional simulation of the circuit to verify that it is working correctly.

Part 3: Full Adder

9. Design a 1-bit Full Adder circuit with three inputs `a`, `b`, and `c_in`, and two outputs `s` and `c_out`.
10. Draw a block diagram of the circuit showing the inputs and outputs for the circuit.
11. 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 and an OR gate.
12. Save the file with the name *FullAdder.v* (or similar as appropriate). Submit a copy of your code in your prelab report (screenshot of all the code is preferred).
13. Perform a functional simulation of the circuit to verify that it is working correctly.

Part 4: Counter

14. Design a generic N-bit Counter with clock and clear inputs `clk` and `clr`, and an N-bit output `q`. All signals should use positive logic.
15. Draw a block diagram of the counter showing the inputs and outputs for the circuit.
16. In your open project create a new source (Project > New Source) and create a Verilog module for the Counter.
17. Save the file with the name *Counter.v* (or similar as appropriate). Submit a copy of your code in your prelab report (screenshot of all the code is preferred).
18. Perform a functional simulation of the circuit to verify that it is working correctly.

Part 5: Top Level Modules

19. Create a new module which combines the 2-bit counter and half adder. Use the output of the counter as the input to the half adder. Simulate the circuit. Paste the results in your prelab report.
20. Create a new module which combines the 3-bit counter and full adder. Use the output of the counter as the input to the full adder. Simulate the circuit. Paste the results in your prelab report.
21. Bring your Verilog code in a flash drive.

5.5.2 Pre-lab Questions

1. Write a Verilog declaration of the following wires: (a) a 32-bit wire having name data_bus and type wire; (b) scalar wires 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):
    a. Verilog is case sensitive.
    b. Built-in primitives describe only combinational logic.
    c. The right arithmetic shift operator is denoted by >>.
    d. The output ports of a module must be listed first in the list of ports.
    e. The type of the input port of a module must be a wire.
    f. Instantiating a module within another module creates a design hierarchy.
    g. Wires can be assigned to in always blocks.
    h. Outputs of modules can be a register.
    i. Assign should always be used inside always blocks.
    j. Statements are executed sequentially.
    k. Begin and end are equivalent to { and } in C.

5.5.3 Pre-Lab Report

In your prelab report, include the following:

1. Half adder truth table
2. Half adder block diagram
3. Half adder Verilog program
4. Full adder truth table
5. Full adder block diagram
6. Full adder Verilog program
7. N-bit counter block diagram
8. N-bit counter code (screenshot preferred)
9. Half adder and 2-bit counter program (screenshot preferred)
10. Half adder and 2-bit counter simulation output
11. Full adder and 3-bit counter program (screenshot preferred)
12. Full adder and 3-bit counter simulation output

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

1. Set the half adder with the 2-bit counter as the top module of your design.

   The module should look similar to the following:

```
3  module ha_count(
4      input clk,
5      input clr,
6      output sum,
7      output c_out,
8      output [1:0] ab
9     );
10
11     wire [1:0] q;
12
13     assign ab = q;      //allows us to see inputs
14
15     half_adder h1(.a(q[1]), .b(q[0]), .c_out(c_out), .s(sum));
16     counter #(2) c1(.clk(clk), .clr(clr), .q(q));
17
18 endmodule
```

Note that we are using the counter to create the inputs to the half adder and at the top level outputting both the output of the half adder and the inputs to the half adder.

Right click on the module name in the Sources pane and select Set as Top.

Run synthesis. After it is completed, open the synthesized design. Select the I/O Port pane. If it is not shown, in the menus, select Window -> I/O Ports. Assign clk to W5 which is the pin connected to the internal 100 MHz clock, clr to one of the push buttons, and the outputs to one of the Pmod ports.

Synthesize the module. Implement the module. Generate Programming File. Use the hardware manager to program your FPGA.

View Sum, Cout, A and B on the DLA. Make sure you are grounding the DLA with one of the ground pins provided on the Pmod port. Save a copy for your lab report.

2. Set the full adder with the 3-bit counter as the top module of your design. Repeat the implementation and data gathering from the first part for the full adder.

5.7 Post-Lab Report

In your post-lab report, include the observed waveforms for the Half Adder and the Full Adder and the pin assignments (e.g. Clk was assigned to the internal clock, clr was assigned to a push button). In addition answer the following questions:

1. What sampling rate should be used on the DLA to capture the signals from the adders?
2. How was hierarchical design used in the final module?
3. Use the RTL schematic viewer to see the synthesized hardware. Explain why it is correct.
4. What is the difference between structural and behavioral Verilog? Which did you use for which module?