Basys 3 FPGA Manual

Elizabethtown College
Clay Buxton

v1.0 – Clay Buxton 5/12/19
v0.2 – Clay Buxton 4/26/19
v0.1 – Clay Buxton 3/25/19

This manual is a work in progress. Sections with no prefix are finished and complete. Sections marked with a [WIP] mean they are currently work in progress, these may not be finished. Segments marked [NS] mean they have not been started and have yet to be started.

This manual in no way, shape, or form should be taken as straight truth. I've been learning as well as writing this manual, it is prone to mistakes, and many beginner errors. If other resources conflict with what this manual says, take the information told in other sources.

Table of Contents:

Resources:

Digilents Basys 3 Github Repository https://github.com/Digilent/Basys3
This repository holds the constraints file for the Basys 3 as well as a few helpful example projects. It is an excellent resource for looking at some examples in Verilog for the board.

Clays EGR333 Github Repository https://github.com/clbx/EGR333
This repository holds all the code shown in this manual and some simple projects to help learn Verilog

MIT Intro to Verilog http://web.mit.edu/6.111/www/f2016/handouts/L03_4.pdf
These slides are a good, brief, and complete look into the basics of Verilog. Very Useful

1 Introduction to FPGA’s

This section does not hold any information on how to program or operate the boards. Operable information starts in section 2

FPGA’s or Field Programmable Gate Arrays are development boards that house a chip that can be programmed to perform as a set of logic blocks functionally.

This manual is strictly for the Basys 3 housing the Artix 7 chip. Vivado is used to program this chip, and any reference to programming should be assumed to be done through Vivado unless otherwise specified.

FPGA vs. Breadboarding vs. Logisim

- FPGA’s allow for a wide range of features that designing a circuit manually on a breadboard and simulating one like in Logisim do not provide.

- Using Vivado, we can streamline the design, testing, and implementation of a circuit. Unlike with breadboarding it must all be done manually and Logisim where it is challenging to implement.

- FPGA’s allow for quick re-use. An FPGA has the ability to be reprogrammed in a matter of minutes. Logisim also provides this, however when hand-building circuits this becomes time-consuming

- Designing a circuit in Vivado allows for the use of HDL and other programming languages. This gives the ability to rapidly develop designs

- Voltage, current, and power are irrelevant when planning for an FPGA; all this is taken care of behind the scenes.

Designing for an FPGA allows for much higher density designs. An FPGA is able to support more circuitry than 100 breadboards and models in Logisim face the technical limitations of the software along with a lack of an efficient way to manage many designs.

FPGAs are made from a single component, a logic cell. These logic cells can be reconfigured and re-assigned as needed by the program being written to them. Logic cells contain three main parts, a look-up table or LUT, a D flip-flop, and a multiplexer. A LUT can function any logic function based on the number of pins they are allocated. They can be assigned to any logic input and output and are not limited to gates or simple components. This design allows LUTs to be used together to implement any logic and are used to implement logic on an FPGA.

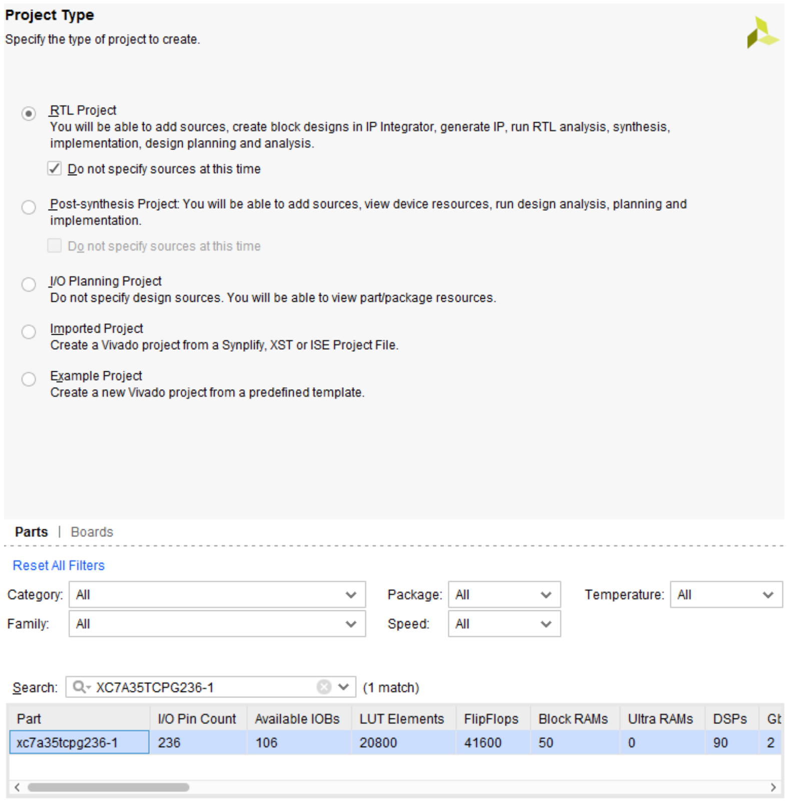
The Basys 3 boards are programming using the Vivado Software Suite. Vivado is the replacement for the old Xilinx ISE design suite from 2014 onwards. Vivado does not support any older chips, and Xilinx ISE does not support any newer chips. Unlike ISE which relied primarily on gate design, Vivado utilizes mostly the use of HDL and supports designs built with high-level languages like C and C++.

2 Setting Up Your First Project and Basic Vivado UI

Vivado is the design suite used to write programs onto the FPGAs. At first, it can be a very intimidating program. In the first few sections of this manual, the FPGAs are programmed using a **Hardware Descriptive Language (HDL)**. Hardware Descriptive languages are different from most high-level languages like C and Java. Instead of writing code traditionally, HDL's describe hardware components. This manual uses Verilog for all examples and tutorials in the first few sections. However, this manual only covers simple Verilog (enough to get a simple circuit up and running on the FPGAs) in depth topics are not included as there are many better resources to use for that.

To start Vivado find the shortcut on the desktop or find it in the start menu. Make sure you are launching Vivado and not Vivado HLS. The shortcut should be something along the lines of **Vivado 20XX.Y** where XX is the year and Y the version number

Once Vivado has loaded you should see a Project Screen with three sections **Quick Start**, **Tasks**, and **Learning Center**, along with the recent projects window. Click **Create Project** to start with a new project.



Project Type

Specify the type of project to create.

[✓ RTL Project] You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

[✓ Do not specify sources at this time]

Post-synthesis Project: You will be able to add sources, view device resources, run design analysis, planning and implementation.

Do not specify sources at this time

I/O Planning Project
Do not specify design sources. You will be able to view part/package resources.

Imported Project
Create a Vivado project from a Synplify, XST or ISE Project File.

Example Project
Create a new Vivado project from a predefined template.

Parts | Boards

Reset All Filters

Category: All
Package: All
Temperature: All
Family: All
Speed: All

Search: XC7A35TCPG236-1 (1 match)

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Gt
xc7a35tcpg236-1	236	106	20800	41600	50	0	90	2

[**1**] The first window presents you with the name and location of the project. Where you save this doesn't matter, put it wherever is most convenient for you. I did not notice any reduced performance by having it in my public/private folders.

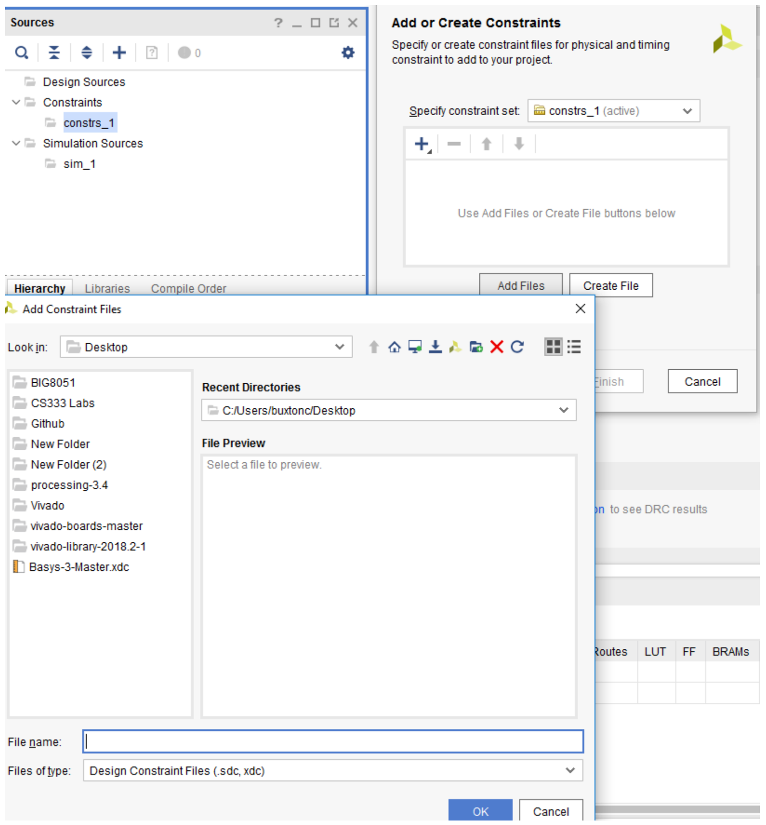
[2] The second window asks you to specify the **Project Type.** Select **RTL Project** and be sure that "**Do not specify sources at this time"** is **checked**. The box should be because you do not currently have any sources to add and want to open a blank project. RTL project is the standard project in Vivado that targets the FPGAs.

[**3**] The third window asks you to specify the board that we are using. The part number for the Basys 3 FPGAs is **XC7A35TCPG236-1**. Select it and click next, then finish.

You should now see the main screen for the Vivado Development Environment. There are three main sections here. You can see **Flow Navigator** to the left, **Project Manager** in the middle, and the **Console and other outputs** on the bottom of the Project Manager.

In the Project manager, there are three smaller boxes. The **Sources**, **Project Summary**, and **Properties**. You can ignore the properties and summary boxes for now

In the **Sources** box, you'll see three folders, **Design Sources**, **Constraints**, and **Simulation Sources**. Our first focus is going to be on the Constraints. A **Constraint** is a file that tells Vivado what hardware to use and where on the board it is. The constraint files have a .xdc file extension. This file is provided to you, though you may write your own. It is called "**Basys-3-Master.xdc**" or something along those lines. Click the "**plus**" icon at the top of the sources window. The next section describes how to configure the design constraints for the FPGA.



The **Design Sources** folder is where you put the source code for the project. That process is covered in a later step.

Now connect the MicroUSB cable to the FPGA and your Computer and turn the "on" switch on the FPGA. The FPGA should power on with the example program running.

Summary

To set up a project, follow the steps:

1. Set the location of the project and give it a name
2. Select **RTL Project**, make check **Do not specify design sources**
3. Select the **XC7A35TCPG236-1**
4. Import the **Basys-3-Master.xdc** design constraint

3 Inputs, Outputs and configuring

Design Constraints

IMPORTANT
Before you're done editing your constraints file make sure these things are done:

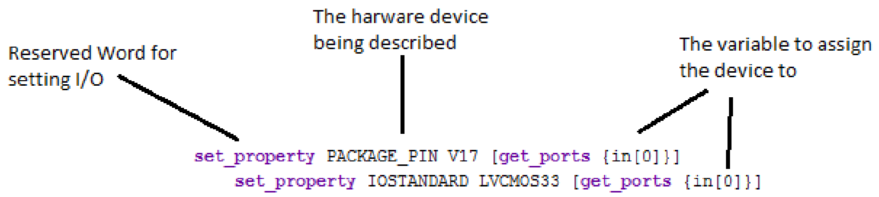
[**1**] All the variables that are assigned in the constraints are used in your code

[**2**] You don't use a variable in the code that is undefined in the constraints

Otherwise your program will not compile!

The Basys 3 Constraints file can be found in the Digilent Basys 3 Github Repository

To configure what inputs and outputs you are using in your project, and assign the hardware to a variable in software, you need to edit the constraints file. When you open the file **Constraints/constrs_1/ Basys-3-Master.xdc** it will open long file with many lines starting with `set_property.` Every two lines of this file describes one input or output on the FPGA. Below is a diagram which describes what each of the parts of the lines do.


Reserved Word for setting I/O
The harware device being described
The variable to assign the device to

```
set_property PACKAGE_PIN V17 [get_ports {in[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {in[0]}]
```

`set_property` Is a reserved word that tells Vivado that we are setting the property of a hardware device.

`PACKAGE_PIN` Describes which device on the FPGA we are accessing. Here it is referencing V17 which is the first switch. Sometimes the pin on the board can be hard to identify it is generally the number in parentheses printed on the board near the device you want to use.

`get_ports` Assigns the hardware device to a specific variable. Here it is putting the switch to the first position of the "in" array.

For example, if we wanted to set the first LED on the board to the variable `led` in code. We would put the following lines in the design constraints file

```
set_property PACKAGE_PIN V16 [get_ports {led}]
    set_property IOSTANDARD LVCMOS33 [get_ports {led}]
```

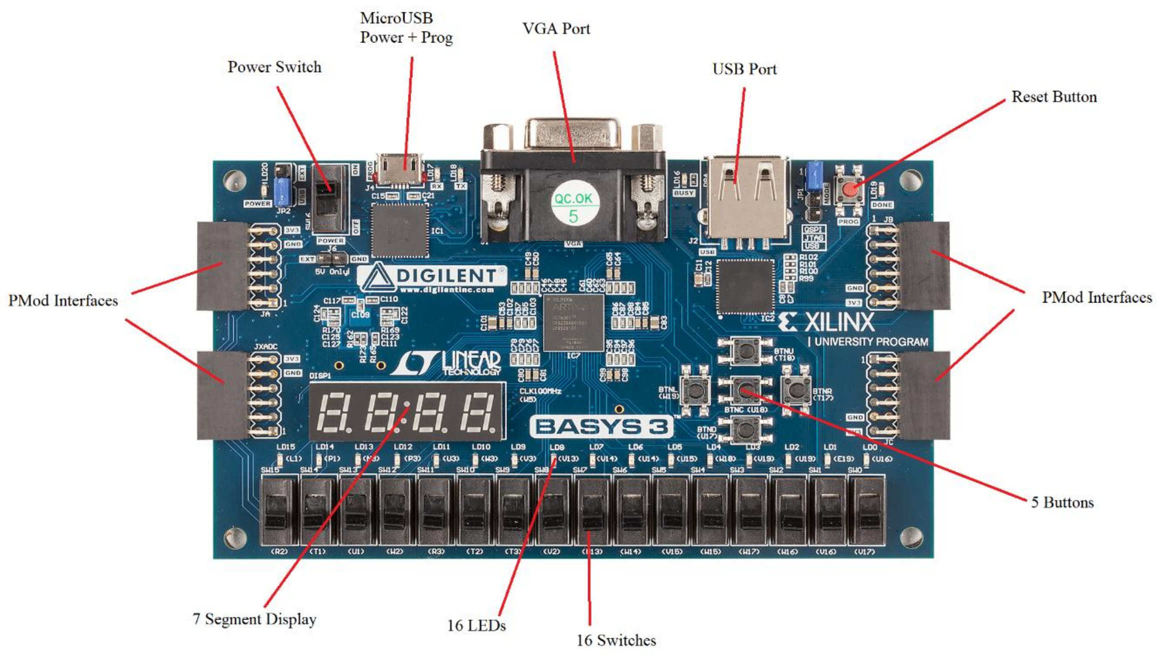
The `IOSTANDARD` portion of the line tells Vivado what protocol is used to talk to the hardware though you don't need to worry about this, it won't change.

To correctly setup which devices you are using, comment and uncomment the lines of the things you need to use. Make sure to keep unused items commented and uncomment all used items. If you don't use a device described in the constraints, your project will not build. Similarly, if you try and use commented device, it will not work, and you'll get similar results.

There are 12 sections in this file each correlates to a different type of input or output on the board.

The main ones that are covered in the early sections are the **switches**, **LEDs**, **clock**, and **buttons.**

Some of the inputs are assigned directly; others require different ways of communicating with the hardware. Things like the switches and LEDs are directly referenced, but some of the more complicated devices cannot. (ex: `led[1] = true` will turn on a light) however most of the other devices have specific ways to communicate with the board; these are covered in their respective sections.



List of Inputs/Outputs:

Clock: The first section is for the internal clock that Verilog projects can use. It has an extra line that allows for setting clock parameters.

Switches: This section is for the 16 flip switches along the bottom of the board. You should use these in an array. These switches are referenced directly.

LEDs: The 16 Small LEDs above the switches along the bottom of the board. It is recommended to use these in an array. There are other referenceable LEDs on the board, but they are used for other functions, and you should stick to the 16 along the bottom. These can be referenced directly.

7 Segment Display: A 4 digit seven segment display.

Buttons: There are five referencable buttons in the middle of the board. The buttons are **not** debounced and can be referenced directly

PMod Headers: These are the 4, 12 pin connectors off the sides of the board. They can be used to communicate with external things. These are covered extensively in another section

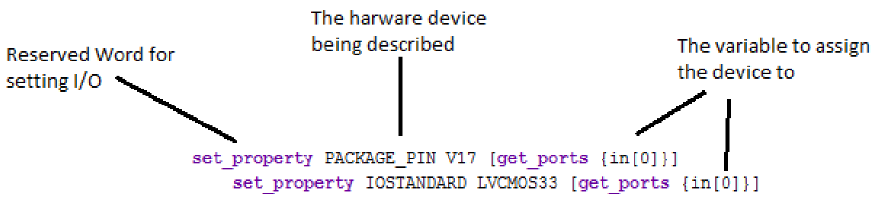
VGA Connector: VGA Connector to allow to display graphics. This is not covered in the current version of the manual

USB Connections: Way to communicate over USB. This is not covered in the current version of the manual

Quad SPI Flash: Flash memory. This is not covered in the current version of the manual

Summary

The constraints file holds the information of what is being used and what isn't being used for the project. In each set of lines, you can set what hardware is being used and assign it to certain variables by referencing this chart.


Reserved Word for setting I/O
The harware device being described
The variable to assign the device to

```
set_property PACKAGE_PIN V17 [get_ports {in[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {in[0]}]
```

4 Basic Verilog (4 Bit Adder)

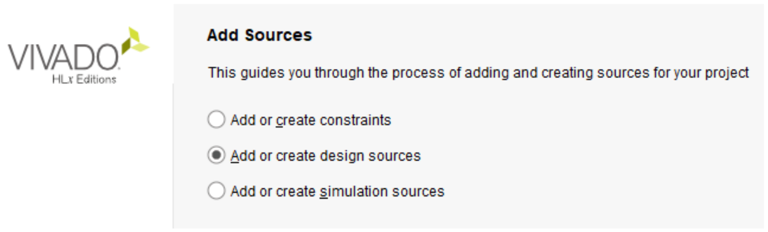
As previously mentioned Verilog is different than a high-level language like C or Java. Verilog describes hardware like wires, gates and other hardware components. Things like `wire`, `xor`, and `reg` are reserved words instead of `int`, `double`, and `char`. They describe physical components that make up a circuit. Verilog still has things like logic and loops, which are used in the circuits.

The absolute best thing to do when starting to program in Verilog is to remember you're not writing a program, you're writing about a circuit.

By the end of this section, you'll have a full 4-bit adder in Verilog and program it to the FPGA in the next.

The first thing we want to do is create our source files. Since we are going to create a 4-bit full adder, we are going to make 3 circuits; a half-adder, a full-adder, and then our assembled 4-bit full adder. Each file is a module, and a module is essentially a sub-circuit from Logisim. A circuit that is defined in one place and can be used elsewhere. To assemble our full adder, we need 3 files.

To create source files click the "**plus**" icon in the **Sources** panel. Select **Add or Create Design Sources** and next.

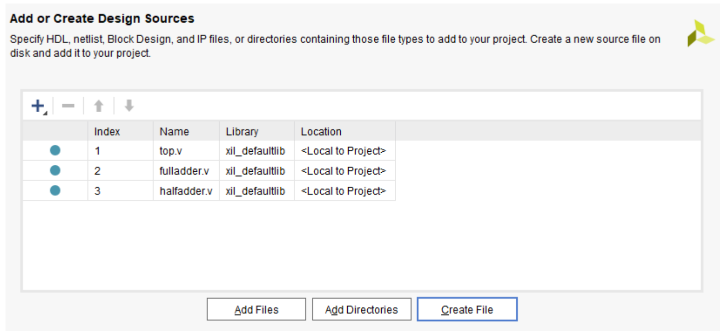

Add Sources

This guides you through the process of adding and creating sources for your project

- ○ Add or create constraints
- ● Add or create design sources
- ○ Add or create simulation sources

In the next panel, it shows all of the files that are being added to the project. Since we don't have any files to import, we want to create the files. Click **Create File**, and call the first file **top**. The top is similar to main for an HLL.

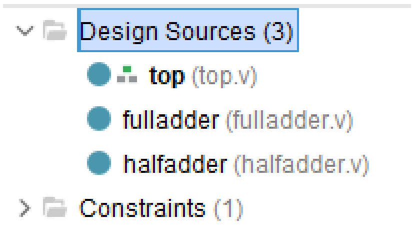
Make sure the file type is **Verilog,** and the file location is **<Local to Project>.** Then create **fulladder** and **halfadder** with the same settings.


Add or Create Design Sources

Specify HDL, netlist, Block Design, and IP files, or directories containing those file types to add to your project. Create a new source file on disk and add it to your project.

	Index	Name	Library	Location
●	1	top.v	xil_defaultlib	<Local to Project>
●	2	fulladder.v	xil_defaultlib	<Local to Project>
●	3	halfadder.v	xil_defaultlib	<Local to Project>

Once you click finish, a new window will appear that allows you to declare what the inputs and outputs are of each of the modules. All this does is autogenerate the code inputs and outputs, you usually can fill this out, but for this manual, we're going to do it manually. Leave it blank and hit **OK** and **Yes** on the next window.


Design Sources (3)
- top (top.v)
- fulladder (fulladder.v)
- halfadder (halfadder.v)

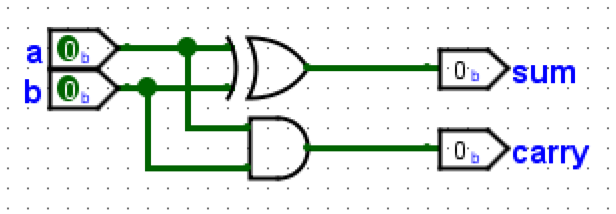
Constraints (1)

You should now see the 3 files in the Sources folder in the sources pane. You'll notice that one of them is bolded, this is what Vivado has currently selected as the **top** if "top,v" is not currently selected, right click on it and select **set as top**

Now that we have our source files laid out, we can begin to write our code.

Half Adder

At this point, you should know how to design a half adder in something like Logisim. It takes in 2 inputs, “adds” them using an XOR and AND gate and the outputs them on an output and carry outputs.



When opening the file, you’ll see some code already generated.

```
`timescale 1ns / 1ps
```

Sets the timescale for the project, leave this as it is, it’s not important right now

The large commented block is to put in information about the file into it. It is important to always fill this out this for every file, no matter how mundane it might feel. There is an example given in the appendix.

```
halfadder.v

module halfadder(

    );
endmodule
```

Shown to the left is the base of the module. As you can see it looks very similar to a class in an HLL. It works similarly. Inside the parentheses, we’ll define what the inputs and outputs of the circuit are

Let’s write some code!

```
halfadder.v

module halfadder(
    input a, b,
    output sum, cout
    );

endmodule
```

Inside the parentheses, we put **a**, **b**, **sum**, and **cout**. You also assign inputs and outputs here as shown.

Order here matters, when you call the circuit in another design, you’ll give the variables in the order described here

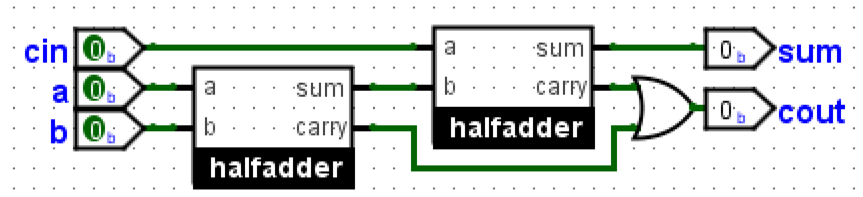
Using gates in Verilog is very similar to calling a function in another language. There are functions for XOR, AND, OR, XOR, etc. To use these you give them arguments just like you would a function in Java or C. The first parameter is the output of the gate, and then the following are the input. You can give it as many inputs as you please.

```
halfadder.v

module halfadder(
    input a, b,
    output sum, cout
    );

    xor(sum,a,b);
    and(cout,a,b);
endmodule
```

Here we describe an AND gate and an XOR gate. The AND gate takes in **a** and **b** and outputs to **cout**. The XOR gate also takes in **a** and **b** but outputs to **sum**, just like in the Logisim circuit above

To the left is a completed half adder. Now we can move on to the next part, the full adder.

Full Adder



As you know a full adder uses two half adders to add 3, 1-bit numbers and output the sum

fulladder.v

```
module fulladder(
    input a,b,cin,
    output sum,cout
    );

endmodule
```

Go to your **fulladder.v** to start work on the full adder module.

Like we did for the half adder, assign your inputs and outputs.

Using another module is very similar to calling a gate. Make sure to give the module the inputs and outputs in the same order you defined them in code

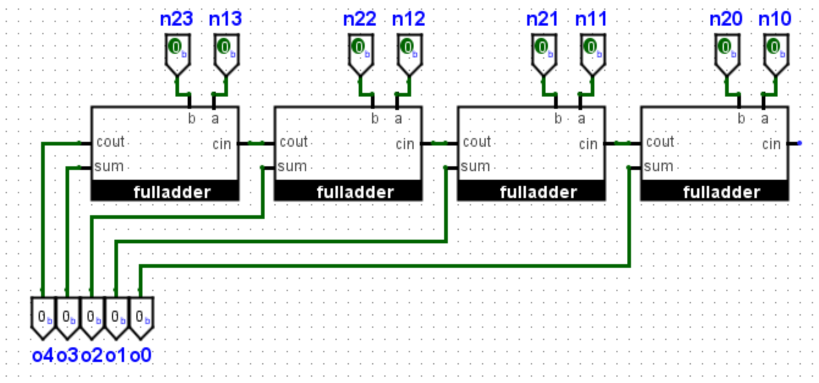
fulladder.v

```
module fulladder(
    input a,b,cin,
    output sum,cout
    );

    wire sum2sum, cout0, cout1;

    halfadder(a,b,sum2sum,cout0);
    halfadder(sum2sum,cin,sum,cout1);
    or(cout,cout0,cout1);

endmodule
```

Here we had to add 3 wires as well; wires connect different parts of your circuit. We didn't need them before because inputs and outputs are typed as a wire, but now since we have wires in the internals of our circuits, we need to define them to connect the half-adders.

The 4 Bit Adder

Now that we have all of the components for our 4-bit adder it's time to assemble it. We just put 4 full adders in a row and connect them. In the logisim design, I had to shrink down the output and input names. n10 refers to num1[0] in the code, n20 refers to num2[0], n11 refers to num1[1], and so on.



top.v

```
module top(
    input [3:0] num1,
    input [3:0] num2,
    output [4:0] out,
    );



endmodule
```

You can see here that some of the inputs look a little bit different than before. The `[3:0]` in front of the variables mean they are **busses**. Busses are wires that can handle more than one bit at a time and are similar to arrays from an HLL. A 3:0 bus holds 4 bits, from bit 0 to bit 3. Busses do not have a width limit (within reason).

top.v

```
module top(
    input [3:0] num1,
    input [3:0] num2,
    output [4:0] out,
    );

    wire cout0,cout1,cout2;


endmodule
```

We'll need to create some wires so we can connect our full adders.

top.v

```
module top(
    input [3:0] num1,
    input [3:0] num2,
    output [4:0] out
    );

    wire cout0,cout1,cout2;

    fulladder(1'b0,num1[0],num2[0],out[0],cout0);
    fulladder(cout0,num1[1],num2[1],out[1],cout1);
    fulladder(cout1,num1[2],num2[2],out[2],cout2);
    fulladder(cout2,num1[3],num2[3],out[3],out[4]);


endmodule
```

Now we can add in our four full adders to the circuit. There's two new things in this snippet too `num1[0]` a bus reference and `1'b0` a literal.

Busses are referenced like arrays in an HLL. Just give the name and index as you would regularly

Constants are defined in either binary, octal, decimal or hexadecimal. The very first number defines how many bits the number is. In this case, we just wanted a one bit zero to fill our first full adders carry in. The `'b` tells it that the number we're giving it is in binary, you can give it `'o`, `'d`, or `'h` for octal, decimal, and hexadecimal respectively

Our full adder is now complete! In the next section, we'll program the board.

5 Programming the FPGA

Armed with a 4-bit adder and a general idea of how constraints work, we can now edit our constraints file to fir the needs of our program. In the constraints, we need to set inputs and outputs for the top file.
We have a 2, 4-bit inputs and then a 5-bit output. Using switches for the inputs seems to make the most sense so we'll set our first four switches to **num1[]** and the next four switches to **num2[]**

Basys-3-Master.xdc

```
## Switches
set_property PACKAGE_PIN V17 [get_ports {num1[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {num1[0]}]
set_property PACKAGE_PIN V16 [get_ports {num1[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {num1[1]}]
set_property PACKAGE_PIN W16 [get_ports {num1[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {num1[2]}]
set_property PACKAGE_PIN W17 [get_ports {num1[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {num1[3]}]
```

Open your **Basys-3-Master.xdc** and assign the first four switches to num1[0] through num1[3]

Basys-3-Master.xdc

```
set_property PACKAGE_PIN W15 [get_ports {num2[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {num2[0]}]
set_property PACKAGE_PIN V15 [get_ports {num2[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {num2[1]}]
set_property PACKAGE_PIN W14 [get_ports {num2[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {num2[2]}]
set_property PACKAGE_PIN W13 [get_ports {num2[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {num2[3]}]
```

Then do the same for the next four switches.

Basys-3-Master.xdc

```
## LEDs
set_property PACKAGE_PIN U16 [get_ports {out[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {out[0]}]
set_property PACKAGE_PIN E19 [get_ports {out[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {out[1]}]
set_property PACKAGE_PIN U19 [get_ports {out[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {out[2]}]
set_property PACKAGE_PIN V19 [get_ports {out[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {out[3]}]
set_property PACKAGE_PIN W18 [get_ports {out[4]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {out[4]}]
```

And then set the outputs

Make sure unused lines are commented out; otherwise your project will not build!

Now it’s time to put the project onto the board. Make sure your FPGA it plugged in and powered on, you should see the demo program running. In the **Project Manager** window towards the bottom you’ll see a button called **Generate Bitstream** this runs all the necessary steps to compile your project.

There are three steps from Verilog to board.

The first is **Synthesis**. The Synthesis synthesizes all of your code into a gate level design. Your Verilog code turns into gates as it would on a physical circuit

The second is **Implementation.** During implementation, Vivado optimizes the gate design to run faster and use less power. If you were to take a look at the circuit after optimization, it would look unrecognizable since the circuit has been re-arranged to make it run better on the board.

The last step is to **Generate Bitstream;** this puts your gate design into a compiled binary file that the FPGA knows how to read. At this point, the design is no longer readable by anything except the board itself.

You can run all of these steps individually but starting any of the later ones will start the earlier ones if changes were made.

At this point run **Generate Bitstream**, be patient as this can take a few minutes. You can see what step it is currently on in the top right of the window

Once this is finished Click **Open Hardware Manager** to open the hardware manager.

In the green ribbon at the top of the window, you’ll see a message saying **No hardware target is open**. Click **Open Target** and select **Auto Connect**, make sure the FPGA is on.

How the FPGA is connected and in the same ribbon you should see a **Program Device** button, click this. Press **Program** on the next window

The FPGA is now running your 4 Bit Adder!

6 Verilog Snippets

This section isn't a tutorial as much as snippets and facts about Verilog I've learned going along. These are my observations and could be incorrect.

Using always(): In many tutorials, you'll see `always(*) begin end` from my experience Vivado/the FPGA's do not like this. The item inside the parentheses in an always statement is when the loop is triggered. If you did `always(button) begin end` It would enter that whenever a button was high. However, Vivado will give you an error if you use anything besides a clock. So to my knowledge clk is the only thing that is valid in there without Vivado getting upset.

I don't believe this is right though since many tutorials use inputs other than clocks in their code. I'm not sure why Vivado doesn't like this.

Difference between = and <= :

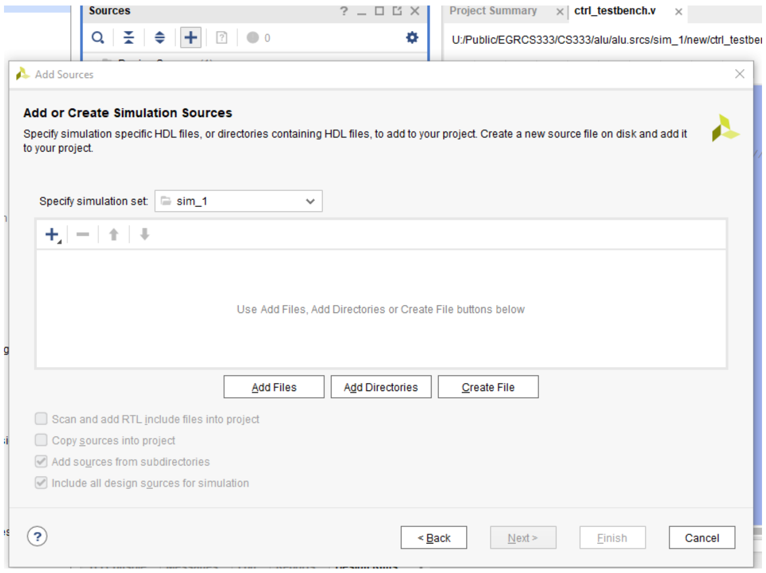
= is a blocking statement. In an always block, the line of code will be executed after the previous line has executed.

<= is non-blocking in nature. That means that in an always block, every line will be executed in parallel.
From Stack Overflow: [https://stackoverflow.com/questions/35435420/what-is-the-difference-between-and-in-verilog]

7 Test Benches & Timing Diagrams

Simulations show the output of a design without having to implement it to the board. These programs are called test benches and describe a set of steps that the simulator will execute on the design.

First, you must create a simulation source. Click the + icon in the Sources menu and select Simulation Source. Then create a new file.



Add or Create Simulation Sources

Specify simulation specific HDL files, or directories containing HDL files, to add to your project. Create a new source file on disk and add it to your project.

Specify simulation set: sim_1

Use Add Files, Add Directories or Create File buttons below

- [ ] Scan and add RTL include files into project
- [ ] Copy sources into project
- [x] Add sources from subdirectories
- [x] Include all design sources for simulation

```
    reg[5:0] A;
    reg[5:0] B;
    reg[3:0] C;
    wire[6:0] led;
    reg clk;
```

Create registers for all the inputs and wires for all the outputs

```
    initial begin
        clk = 0;
        forever #5 clk=~clk;
    end
```

Create a new initial block, this sets the clk to 0 at first and every 5 nanoseconds the clock will flip. This is then set for the entire testbench

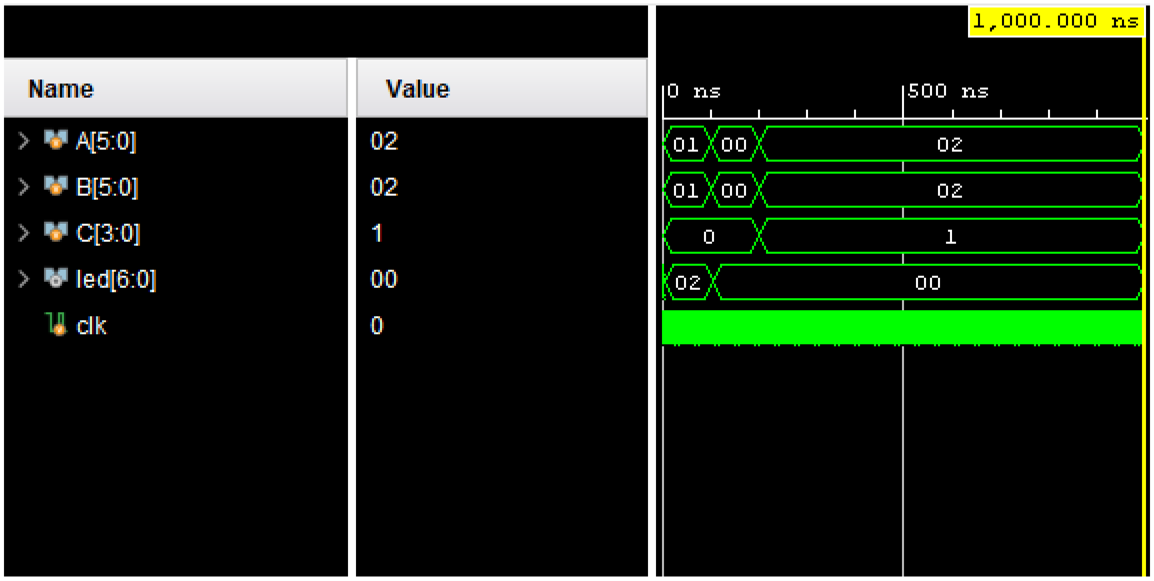
```
ctrl dut (
        .clk(clk),
        .A(A),
        .B(B),
        .C(C),
        .led(led)
    );
```

After that set up your circuit. Here ctrl is the circuit being tested

```
    initial begin
        A = 6'b000001;
        B = 6'b000001;
        C = 4'b0000;
        #100;
        A = 6'b000000;
        B = 6'b000000;
        C = 4'b0000;
        #100;
        A = 6'b000010;
        B = 6'b000010;
        C = 4'b0001;
    end
```

Then you describe your test bench. Here the inputs are set to different values and then are set to wait 100 nanoseconds between changing again. This allows us to see the changes happening in the test bench

Now that the test bench is created, it's time to run the simulation, under the Simulation tab of the Project manager, click on Run Simulation and select Run Behavioral Simulation. After its finished click the Zoom to Fit button to put the test bench in view and you'll have the test bench



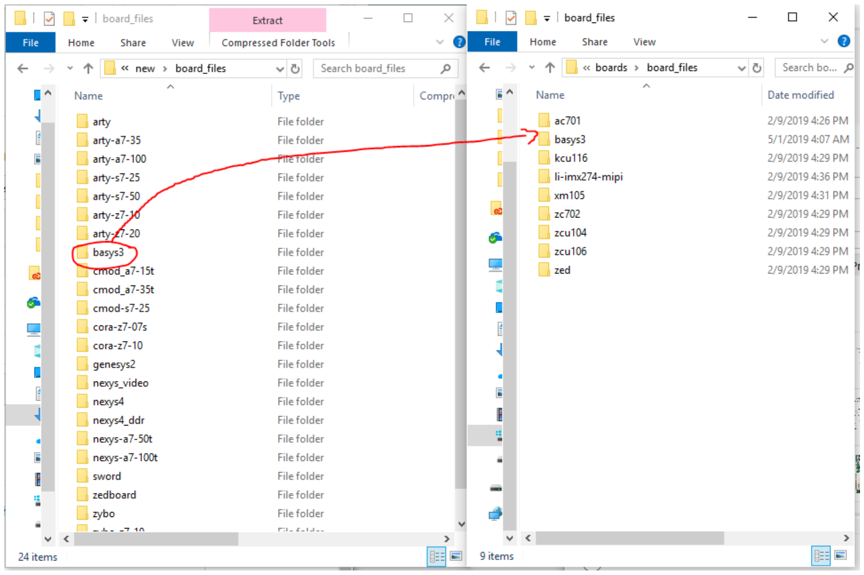
8 Using IP Cores

This section focuses on how to implement a basic design using IP cores in Vivado. IP Cores stand for **Intellectual Property** core. These allow for quickly adding already designed designs to a circuit. Note that this is very different from the previous chapters of the manual. We will no longer be programming in Verilog but creating a circuit by dragging and dropping components and allowing Vivado to do a lot of the work for us. Once we have designed our circuit, we'll write code in the Vivado SDK in which compilation is targeted to run on our designed device

This first part will walk you through how to create a **Microblaze** design in Vivado. Microblaze is a microprocessor core defined in software. It's a 32bit RISC based processor with many configurable variables. The Microblaze is a processor which we can design around and then write to the FPGA.

Before starting you need to obtain the Basys 3 board files from the Digilent Github repository here [https://github.com/Digilent/vivado-boards]. Download or clone the repository and browse to the board files of your Vivado install. Generally this is at C:\Xilinx\Vivado\20XX.X\data\boards\boardfiles. From the repository in /new/board_files/ copy the Basys3 folder to your Vivado install directory.

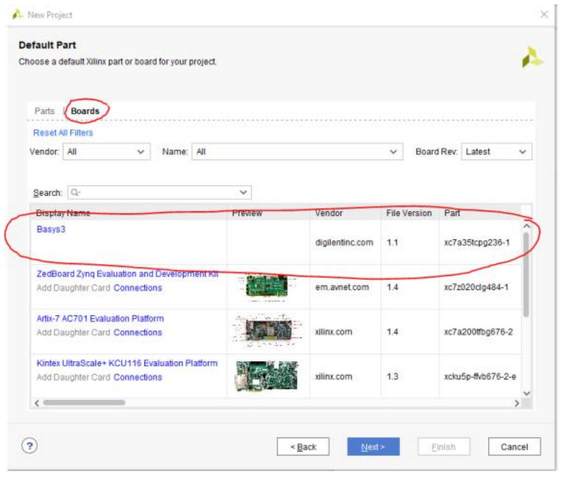
This allows you to use the Basys 3 board in the Vivado IP Designer. If you had Vivado running during this process. You need to restart Vivado if it was running during that process



Name	Type
arty	File folder
arty-a7-35	File folder
arty-a7-100	File folder
arty-s7-25	File folder
arty-s7-50	File folder
arty-z7-10	File folder
arty-z7-20	File folder
basys3	File folder
cmod_a7-15t	File folder
cmod_a7-35t	File folder
cmod-s7-25	File folder
cora-z7-07s	File folder
cora-z7-10	File folder
genesys2	File folder
nexys_video	File folder
nexys4	File folder
nexys4_ddr	File folder
nexys-a7-50t	File folder
nexys-a7-100t	File folder
sword	File folder
zedboard	File folder
zybo	File folder

Name	Date modified
ac701	2/9/2019 4:26 PM
basys3	5/1/2019 4:07 AM
kcu116	2/9/2019 4:29 PM
li-imx274-mipi	2/9/2019 4:36 PM
xm105	2/9/2019 4:31 PM
zc702	2/9/2019 4:29 PM
zcu104	2/9/2019 4:29 PM
zcu106	2/9/2019 4:29 PM
zed	2/9/2019 4:29 PM

At this point create a Vivado RTL project like you usually would. Be sure not to define any sources.
At the stage where you would traditionally choose the part being used. Select the **Boards** tab and select the Basys 3



Default Part
Choose a default Xilinx part or board for your project.

Vendor: All Name: All Board Rev: Latest
Search:

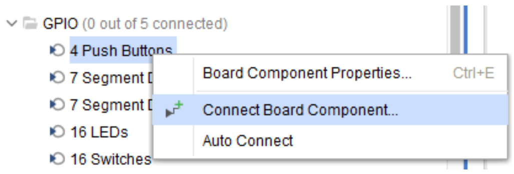
Display Name	Preview	Vendor	File Version	Part
Basys3		digilentinc.com	1.1	xc7a35tcpg236-1
ZedBoard Zynq Evaluation and Development Kit Add Daughter Card Connections		em.avnet.com	1.4	xc7z020clg484-1
Artix-7 AC701 Evaluation Platform Add Daughter Card Connections		xilinx.com	1.4	xc7a200tfbg676-2
Kintex UltraScale+ KCU116 Evaluation Platform Add Daughter Card Connections		xilinx.com	1.3	xcku5p-ffvb676-2-e

Complete the project creation as usual

This is where the design process deviates from the traditional Verilog design. Instead of creating sources, open the IP Integrator tab and select **Create Block Design.** Give the design a name and click OK.

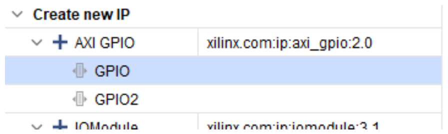
Be sure in the top left window the following tabs are present: **Sources, Design, Signals, Board.** If Board is missing, go back, you currently have the part still selected and need to choose the Basys 3 board.
Now to begin the design

Open the board tab and find **GPIO** in the secions. Right click on **4 Push Buttons** and click **Connect Board Component…** This inserts the push buttons on the board into our IP Design.

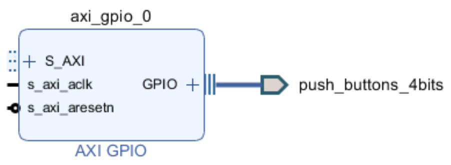

GPIO (0 out of 5 connected)
4 Push Buttons
7 Segment D
7 Segment D
16 LEDs
16 Switches

Board Component Properties... Ctrl+E
Connect Board Component...
Auto Connect

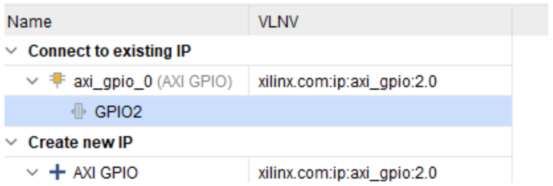
In the dialog window that pops up select **GPIO** under the **AXI GPIO** section. Click OK


Name	VLNV
Create new IP	
AXI GPIO	xilinx.com:ip:axi_gpio:2.0
GPIO	
GPIO2	
IOModule	xilinx.com:ip:iomodule:3.1

You should now see the GPIO block in the Diagram window with the push buttons connected



Now connect the **16 LEDs** and select **GPIO2** under the **Connect to existing IP** tab. Click OK

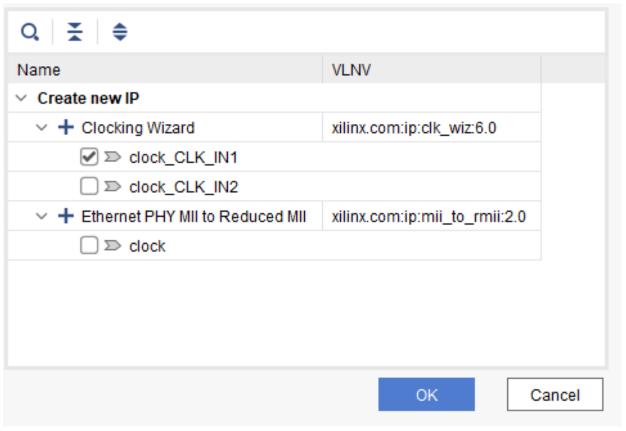

Name	VLNV
Connect to existing IP	
axi_gpio_0 (AXI GPIO)	xilinx.com:ip:axi_gpio:2.0
GPIO2	
Create new IP	
AXI GPIO	xilinx.com:ip:axi_gpio:2.0

The GPIO block should now have a GPIO2 port and the LEDs at the end of it

Now we need to add the clock and the microprocessor itself.

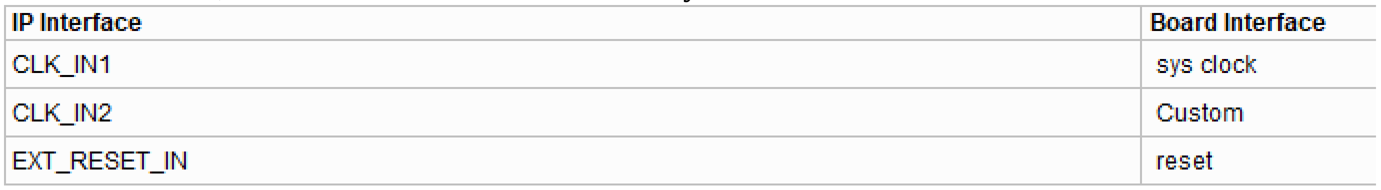
In the boards tab add the **System Clock.**

In the first window make sure only **clock_CLK_IN1** is ticked


Name	VLNV
Create new IP	
Clocking Wizard	xilinx.com:ip:clk_wiz:6.0
✓ clock_CLK_IN1	
☐ clock_CLK_IN2	
Ethernet PHY MII to Reduced MII	xilinx.com:ip:mii_to_rmii:2.0
☐ clock	

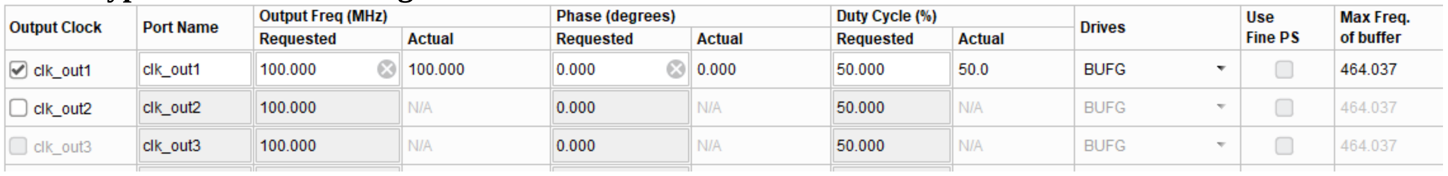
A new **Clocking Wizard** block will appear in your design. Double click on the block to open the customization window for the clock

In the **Board** tab, make sure the **CLK_IN1** is set to **sys clock** and **EXT_RESET_IN** is set to **reset**

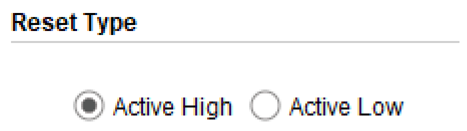


IP Interface	Board Interface
CLK_IN1	sys clock
CLK_IN2	Custom
EXT_RESET_IN	reset

In the **Output Clocks** tab make sure that **clk_out1** both **checked** and set at **100Mhz** and at the bottom of the tab **Reset Type** is set to **Active High**



Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drives	Use Fine PS	Max Freq. of buffer
		Requested	Actual	Requested	Actual	Requested	Actual			
✓ clk_out1	clk_out1	100.000	100.000	0.000	0.000	50.000	50.0	BUFG	☐	464.037
☐ clk_out2	clk_out2	100.000	N/A	0.000	N/A	50.000	N/A	BUFG	☐	464.037
☐ clk_out3	clk_out3	100.000	N/A	0.000	N/A	50.000	N/A	BUFG	☐	464.037



Reset Type

[✓ Active High] Active Low

At the top of the Diagram window there is a green bar click **Run Connection Automation** this does a bit of automatic configuring.

Now add the USB UART block from the Board window. This will allow for debugging over USB.

Now add the Microblaze microprocessor. To do this, right click on an empty spot of the diagram and click on **Add IP** in the window that pops up search for **Microblaze.** Another green ribbon will appear, click on **Run Block Automation** this will put all the blocks in the right place.

During this configuration make sure the follow values are as follows:
Preset: None

Local Memory: 32KB

Local Memory ECC: None

Cache Configuration: None

Debug Module: Debug Only

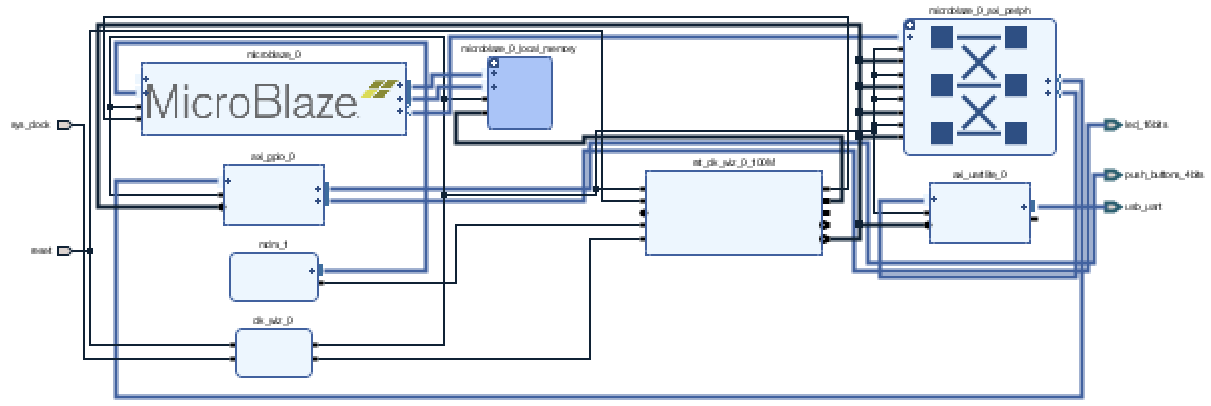
Peripheral AXI Port: Enabled

Interrupt Controller: Unchecked

Clock Connection: /clk_wiz_0/clk_out1

Another green ribbon will appear, click **Run Connection Automation** then click OK

Hit **F6** to validate the design. If Validation is not successful, go back and restart the steps with a new design.



The final design should look something like this

The final thing to do before generating a bitstream is to generate an HDL wrapper to run the design on the board.

Switch to the **Sources** tab, right click on your design and select **Generate HDL Wrapper**. Tick **Let Vivado manage wrapper and auto-update** then click OK. This will generate your top file. Now click **Generate Bitstream**.

Now we need to export the design to the Vivado SDK. In the **File** menu select **Export** and then **Export Hardware.** Make sure **Include Bitstream** is ticked and keep the export location as the local project. Click OK

In the File Menu click **Launch SDK** make sure the locations are set to local project

The Vivado SDK is a software development environment in which you can develop code that is targeted at your design made in the IP integrator. The SDK can take a few minutes to load, please be patient. Once the SDK is loaded it will probably look very familiar, this is because the SDK is based in Eclipse. Everything language independent that you can do in eclipse you can do here.

Create a new Application Project in **File -> New -> Application Project**

Give your Application a name and make sure **C** is selected as the target language. Click next and choose **Empty Application** as the template. Click Finish.

Right click on the **src** folder and create a new source file. Call this file **main.c** add the

main.c

```c
#include "xgpio.h"

//send data over UART
#include "xil_printf.h"

//information about AXI peripherals
#include "xparameters.h"

int main()
{
	XGpio gpio;
	u32 btn, led;

	XGpio_Initialize(&gpio, 0);

	XGpio_SetDataDirection(&gpio, 2, 0x00000000);
	XGpio_SetDataDirection(&gpio, 1, 0xFFFFFFFF);

	while (1)
	{
		btn = XGpio_DiscreteRead(&gpio, 1);

		if (btn != 0) // turn all LEDs on when any button is pressed
			led = 0xFFFFFFFF;
		else
			led = 0x00000000;

		XGpio_DiscreteWrite(&gpio, 2, led);

		xil_printf("\rbutton state: %08x", btn);
	}
}
```

following code to main.c

Now click on **Xilinx** in the toolbar and select **Program FPGA**

Keep all the settings the same and click **Program**

This will then load the program onto the FPGA, but not run it. Right click on your project folder and select **Run As -> Launch on Hardware (System Debugger)**

You will now need to obtain a serial console application. Terra Term [https://osdn.net/projects/ttssh2/downloads/70691/teraterm-4.102.exe/] is recommended

Set TerraTerm to **serial** and select the COM port the FPGA is connected to. In the serial port setup make sure to set the following settings

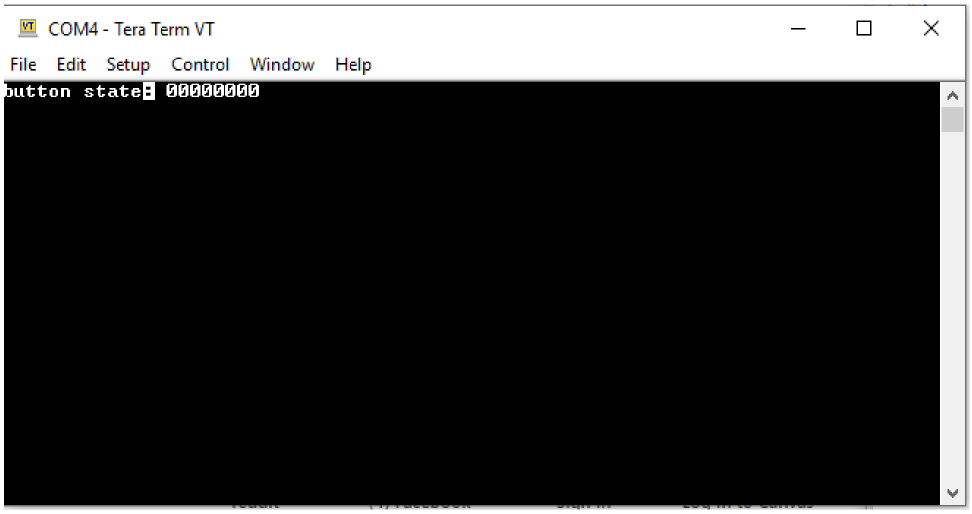
Speed: 9600

Data: 8-Bit

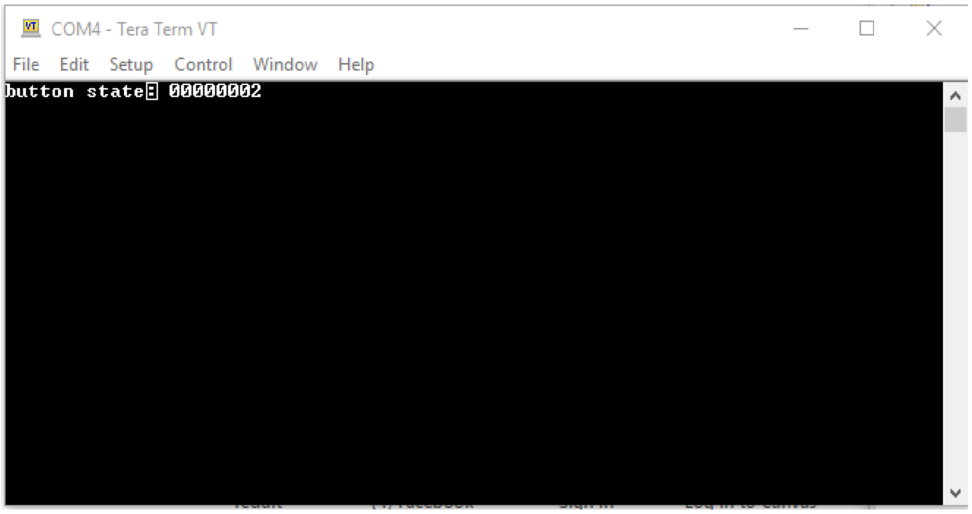
Parity: None

Stop Bits: 1 bit

You should now be able to see the message in the terminal and see the hex representation of the buttons in the terminal



```
button state: 00000000
```



```
button state: 00000002
```

9 Using Pmod (Compass Example)

PMod is an open standard developed by Digilent to use with FPGAs or Microcontrollers. As of the writing of this manual we have the **CMPS2** a compass module, the **DHB1** a motor controller, and **GPS** which does exactly what you think it does. In the coming sections we will cover how to use all of these modules. Each of these also uses their own connection method, so parts of these may be able to be used with other Pmod devices.

To use these, insert them into the PMod ports on the sides of the FPGA

Before programming anything we need to get the Digilent IP Library from their repository here [https://github.com/Digilent/vivado-library/releases?_ga=2.10682993.418939316.1557554693-1032257056.1556697100]

Download the latest release even if you are using a new version of Vivado than what is shown. For example while writing this manual I am using 2018.3 while the newest version of their library is 2018.2. Extract it somewhere where you can find it later

Open the project you made in the last section.

We need to add the Digilent Library you just downloaded, under **Project Manager** open **Settings**

Click the arrow next to **IP** and select **Repository**

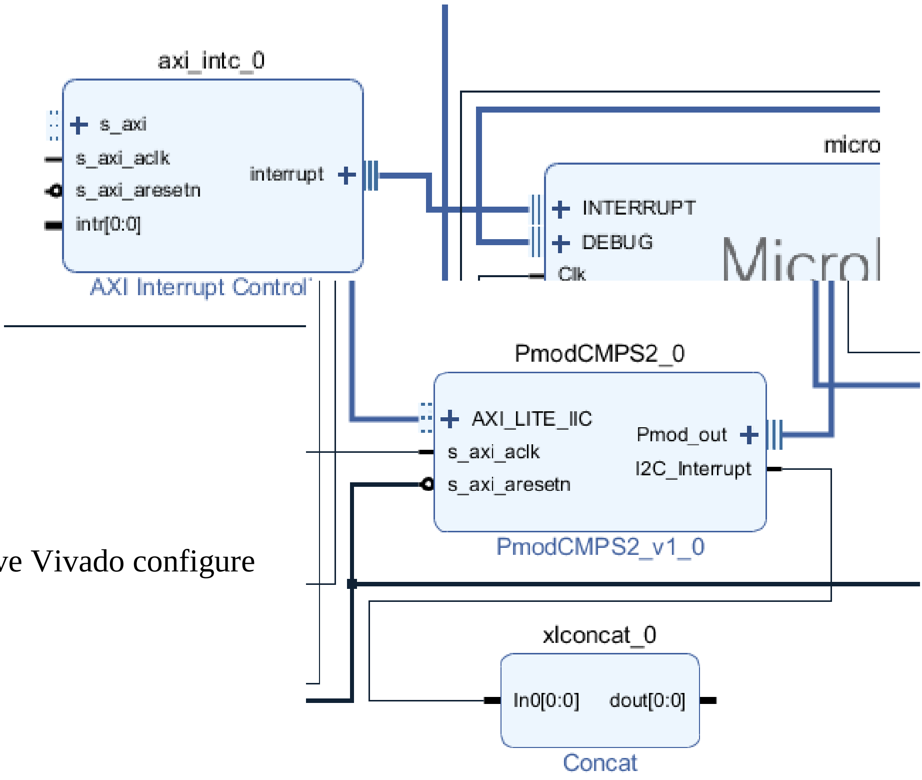
Click the **+** and add the extracted repository. A window should come up telling you that IP cores have been added. Click **Apply** and **OK**

In the **Board** tab scroll down to the Pmod Connectors. Connect it to your design and select the Pmod component you want to use. In this first part we'll be using the compass module **CMPS2.** In this example I have CMPS2 connected to Pmod port JA.

Run **Connection Automation** make sure your Pmod device is checked.

At this point 2 blocks may need to be added another **clock** or a Microblaze **interrupt**. The CMPS2 does not need a clock, but it does need the interrupt. The clock will not be covered in this manual since none of our components require a separate clock.

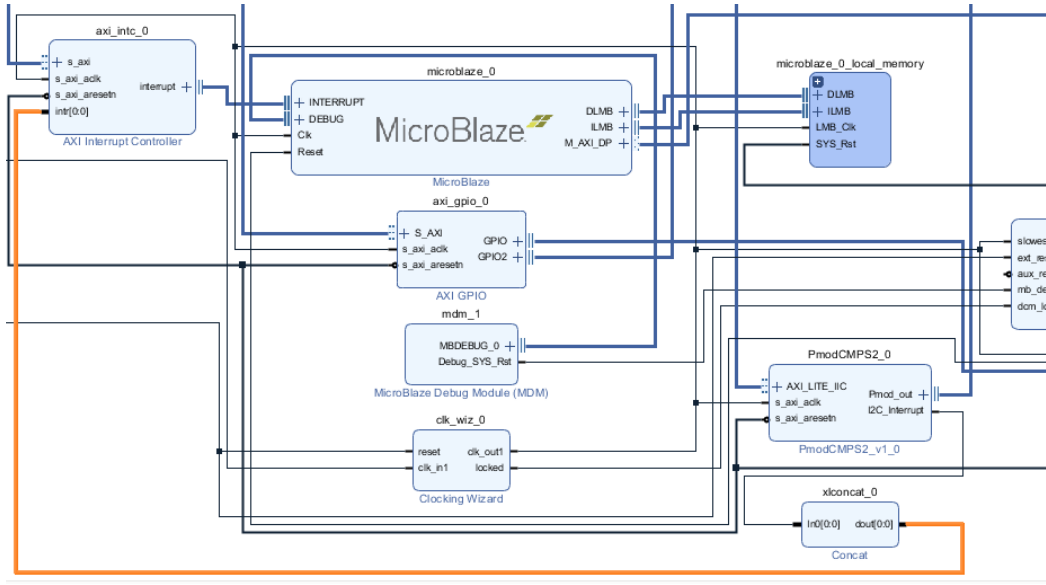
We now need to add an **AXI Interrupt Controller.** Add the AXI Interrupt Controller to your design and manually connect the interrupt port on that block to the one of the Microblaze processor.



Now click **Run Connection Automation** to have Vivado configure everything else.

Now add a **Concat** IP Core. Reconfigure it to only have the number of Pmod devices you are using (right now we are only using 1). Now connect the Interrupt port on your Pmod Device to the In port of the concat block.

Now connect the concat output port to the **intr** port of the AXI Interrupt Controller.



Click the Regenerate Layout Button this organizes the layout of the design a bit

And then validate the design. If validation is not successful restart these steps, you missed a step or didn't configure a component correctly.

Return to the sources window and re-generate the HDL wrapper and generate a bitstream. Export and open the SDK like you did in the previous section.

Create a new Empty Project, make sure C is the target language

Under the platform folder at the top, expand the folder and the driver folder. In here you can find the Pmod interface and an example

Move the example source file you want into the src folder of your project.

Write the program to the FPGA and run as you did in the previous step.

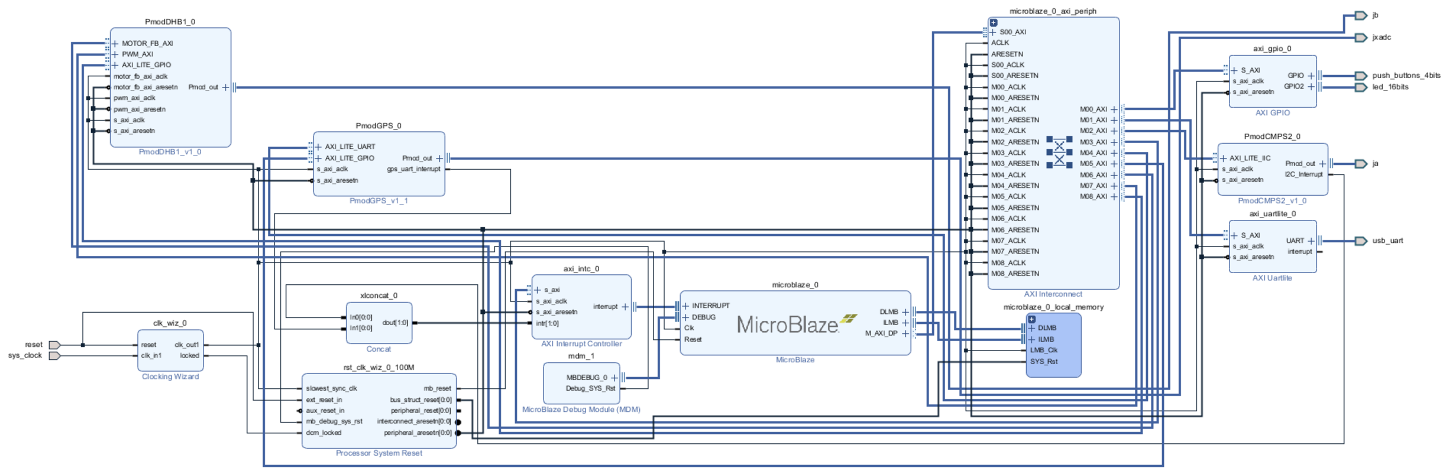
To show information over the Serial Terminal, change any instances of **printf** to **xil_printf** this tells the FPGA to write to the serial terminal.

If all works you should receive compass info (or info from your FPGA perfectly)

You can either continue working off the same design as in previous sections or create a new one. I will be building off the ones in previous design so that the finished design will have motor control, GPS and compass in it.

To implement the other devices to your design, add new Pmod blocks to the design. Be sure to increase the concat if it needs an Interrupt and connect that and make a new clock if your Pmod requires it. Run the connection wizard after you are done

The GPS, Motor, and Compass should look something like this



Be sure to generate the HDL Wrapper and then export the hardware to the SDK. Using the examples given you can then implement the other 2 Pmod devices.

Appendix

This appendix holds information useful to the manual that doesn't belong in the middle of a section

Example Verilog Header Comment

```
//////////////////////////////////////////////////////////////////////////
// Company: Elizabethtown College
// Engineer: Clay Buxton
//
// Create Date: 02/09/2019 08:02:42 PM
// Design Name: 4 Bit Adder
// Module Name: top
// Project Name: 4 Bit Adder
// Target Devices: Basys 3
// Tool Versions: Vivado 2018.3
// Description: Adds 4-bit numbers
//
// Dependencies: fulladder.v
//
// Revision: 0.02 - Project Finished
// Revision 0.01 - File Created
// Additional Comments: None
//
//////////////////////////////////////////////////////////////////////////
```