SECURING HARDWARE.COM

Teardown: Portland 2018

# WTFPGA?

https://github.com/securelyfitz/WTFpga

Sponsored by

DIGILENT®
A National Instruments Company

Developed & Presented by
Joe FitzPatrick
joefitz@securinghardware.com

# Introduction

Welcome to the workshop! This is a hands-on crash-course in Verilog and FPGAs in general. It is self-guided and self-paced. Instructors and assistants are here to answer questions, not drone on with text-laden slides.

While microcontrollers run code, FPGAs let you define wires that connect things together, as well as logic that continuously combines and manipulates the values carried by those wires. Verilog is a hardware description language that lets you define how the FPGA should work.

Because of this, FPGAs are well suited to timing-precise tasks. If you need to repeatedly process a consistent amount of data with minimal delay, an FPGA would be a good choice. As your processing becomes more complicated, or your data becomes more variable, microcontrollers can become a better solution.

The objective of this workshop is to do something cool with FPGAs in only two hours. In order to introduce such a huge topic in such a short time, LOTS of details will be glossed over. Two hours from now you're likely to have more questions about FPGAs than when you started - but at least you'll know the important questions to ask if you choose to learn more.

## What We Won’t Learn

In order to introduce Verilog and FPGAs in such a short time, we’re going to skip over several things that will be important when you build your own FPGA-based designs, but are not necessary to kickstart your tinkering:

1. **Toolchain Setup**: All the software is ready to go on the laptop. Plan for several hours to download and install tools on your own. There are walkthroughs available from Xilinx, Digilent, and others.

2. **Synchronous Logic:** We’re dealing entirely with human (AKA slow) input and output today. Running at maximum performance requires synchronizing all of the logic using a common clock, and optimizing the logic to fit that enforced timing.

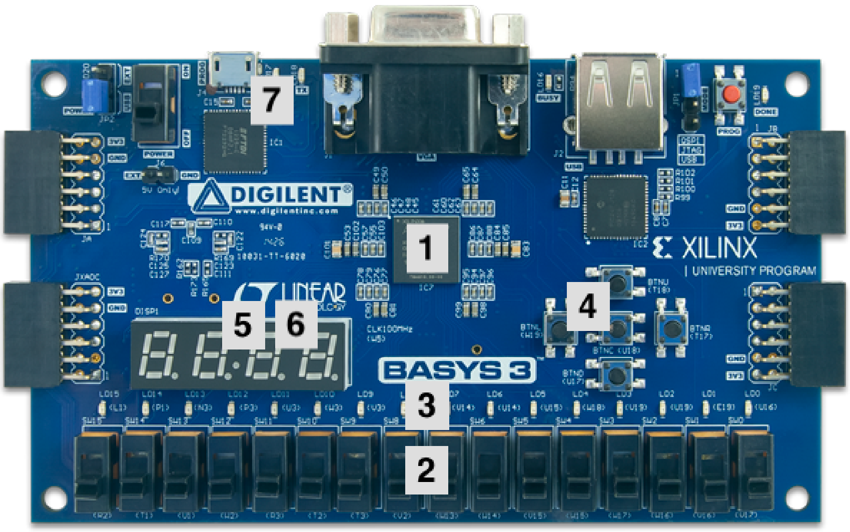
3. **IP Cores**: FPGA vendors pre-build or automatically generate code to let you easily interface your FPGA to interfaces like DRAM, network, or PCIe. We’ll stick to LEDs and switches today.

4. **Simulation**: Didn’t work right the first time? Simulation lets you look at all the signals in your design without having to use hardware or potentially expensive observation equipment.

5. **Testbenches**: For effective simulation, you need to write even more Verilog code to stimulate the inputs to your system.

## Meet the Hardware

This is a Digilent BASYS3 FPGA demonstration/learning board. It's great for learning because it has so many user-accessible inputs and outputs.



| | | |
|---|---|---|
| 1. | Xilinx XC7A35T FPGA | |
| 2. | DIP switches: | **sw[15:0]** |
| 3. | LEDs: | **led[15:0]** |
| 4. | Pushbuttons: | **btnU, D, L, R, and C** |
| 5. | 7-segment displays: | **seg[6:0]** |
| 6. | 7-segment selector: | **an[3:0]** |
| 7. | USB port for power and programming | |

https://digilentinc.com/basys-3

1. Take a look at the board, and identify each of the components listed above. There are  few other components not listed that we won’t use in this workshop.
2. Power on your laptop and connect the power cable from a USB port to the power connector on the device. The FPGA will automatically load a demo configuration from the onboard serial flash. Play around:
    a. What do the DIP switches do?

       ______________________________________

    b. What do the pushbuttons do?

       ______________________________________

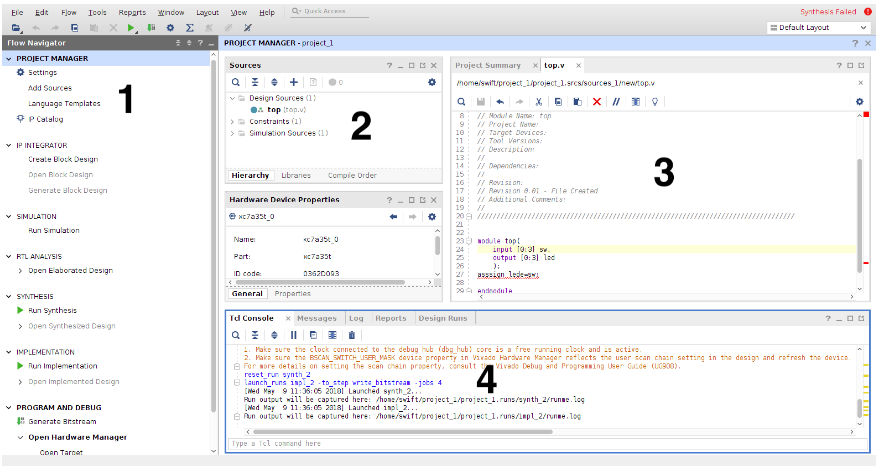
    c. What happens when you power cycle the board?

       ______________________________________

# Configuring an FPGA

Now that we are familiar with the hardware as-is, let's walk through the process of using the Xilinx tools to make sure we can configure the board.

1. Power on your laptop and open the project:
    a. Double click on the icon labeled Vivado on the desktop. (it will take a moment...)
    b. Choose **WTFPGA** under **Recent Projects** to open up the project we'll start with.



```
// Module Name: top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////


module top(
    input [0:3] sw,
    output [0:3] led
    );
asssign lede=sw;

endmodule
```

```
1. Make sure the clock connected to the debug hub (dbg_hub) core is a free running clock and is active.
2. Make sure the BSCAN_SWITCH_USER_MASK device property in Vivado Hardware Manager reflects the user scan chain setting in the design and refresh the device.
For more details on setting the scan chain property, consult the Vivado Debug and Programming User Guide (UG908).
reset_run synth_2
launch_runs impl_2 -to_step write_bitstream -jobs 4
[Wed May  9 11:36:05 2018] Launched synth_2...
Run output will be captured here: /home/swift/project_1/project_1.runs/synth_2/runme.log
[Wed May  9 11:36:05 2018] Launched impl_2...
Run output will be captured here: /home/swift/project_1/project_1.runs/impl_2/runme.log
```

1. **Flow Navigator:** Control stages of the design workflow
2. **Sources:** View your design's hierarchy, components, and files
3. **File viewer/editor:** view reports and edit source files
4. **Console:** find status, and error information

2. All we will do for now is load and run this design to our FPGA board:

    a. If it's not already connected, plug your FPGA board into your laptop with a USB cable.

    b. In the Flow Navigator on the left, choose PROGRAM AND DEBUG -> Open Hardware Manager -> Open Target -> Auto Connect

    c. At the top of the screen, there is a green bar. Click on Program device.

    d. The default options should be correct. Click Program.

    e. Confirm that you were able to replace the default configuration with a new one by checking to see if the behaviour has changed.

    f. Test the buttons and switches. Does this new configuration do anything useful?

       ______________________________________

    g. Toggle the power switch on the FPGA board. Does your default configuration get restored?

       ______________________________________

# Reading Verilog

Now that we know how to use the tools to configure our device, let's start by examining some simple verilog code. Programming languages give you different ways of storing and passing data between blocks of code. Hardware Description Languages (HDLs) allow you to write code that defines how things are connected

1. In the **Flow Navigator** on the left, click on **Project Manager** to leave programming mode and return to our project files
2. Double click on **WTFpga (WTFpga.v)** in the Sources view in the upper left. It will open the file for editing:

    a. Our **module** definition comes first, and defines all the inputs and outputs to our system. Can you locate them on your board?

    b. Next are **wire** definitions. Wires are used to directly connect inputs to one or more outputs.

    c. Next are parallel **assign** statements. All of these assignments are always happening, concurrently.

    d. Next are **always** blocks. These are blocks of statements that happen sequentially, and are triggered by the **sensitivity list** contained in the following **@( )**.

    e. Finally we can instantiate modules. There are a few already instantiated but not really used for anything yet.

3. Now let’s try and map our board’s functionality to the Verilog that makes it happen.
    a. What happened when you pressed buttons?

       ________________________________________

    b. Can you find the pushbuttons in the module definition? What are they named?

       ________________________________________

    c. Can you find an assignment that uses each of the pushbuttons? What are they assigned to?

       ________________________________________

    d. Can you follow the assignments to an output?

       ________________________________________

    e. Do you notice anything interesting about the order of the **assign** statements?

       ________________________________________

You should be able to trace the **btnU** and **btnD** pushbutton inputs, through a pair of wires, to a pair of LED outputs. Note that these aren’t sequential commands. All of these things happen at once. It doesn’t actually matter what order the assign statements occur.

## Making Assignments

Let’s start with some minor changes to our Verilog, then configure our board. Conveniently, there are 16 switches and 16 LEDs on this board. Let’s change the assignment from one pushbutton (**btnD and btnU**) to the switches (**sw[15:0]**). Also, we don’t need to explicitly create a wire to connect things, we can do it directly:

1. First, we make modifications to the file
    a. Remove the **wire1 and wire2** definitions from the file.
    b. Remove all the **assign** statements from the file
    c. Replace the previous assignment for **led** with:
    ```
    assign led[15:0]=sw[15:0];
    ```
    d. Save your file with the icon just above the text window.
2. Next, we need to create a new configuration for our device. Brace yourself - this will take a LONG time - almost 8 minutes! Let’s get it started first, then review what’s happening.
    a. In the **Flow Navigator** on the left, choose **PROGRAM AND DEBUG -> Generate Bitstream** near the bottom.
    b. It should inform you **Synthesis is Out-of-date**. Choose yes, we need to rebuild everything.

3. It will take up to **8 minutes** if everything works right. If you have a simple typo in your file, it should tell you **within a minute**. While we wait, let's understand what's going on:

    a. First, the software will **Synthesize** the design - turn the Verilog code into basic logical operations, optimizing it in the process. This takes **2 to 3 minutes**.

    b. Next, the tools will **Implement** the design. This takes the optimized list of registers and assignments, and **places** them to the logical blocks available on the specific FPGA device we have configured, then **routes** the connections between them. This takes **3 to 4 minutes**.

    c. When that completes, the fully laid out implementation needs to be packaged into a format for programming to the device. There are a number of options, but we will use a .bit bitstream file for programming over JTAG. This takes **about 1 minute**.

    d. Hopefully everything will go as planned. If you have issues, look in the console for possible build errors. If you have trouble, ask for help!

4. Finally, the .bit file needs to be sent to the device over USB. When this happens, the demo configuration will be cleared and the new design will take its place:

    a. In the **Flow Navigator**, choose **PROGRAM AND DEBUG -> Open Hardware Manager -> Program Device -> xc7a35t_0**.

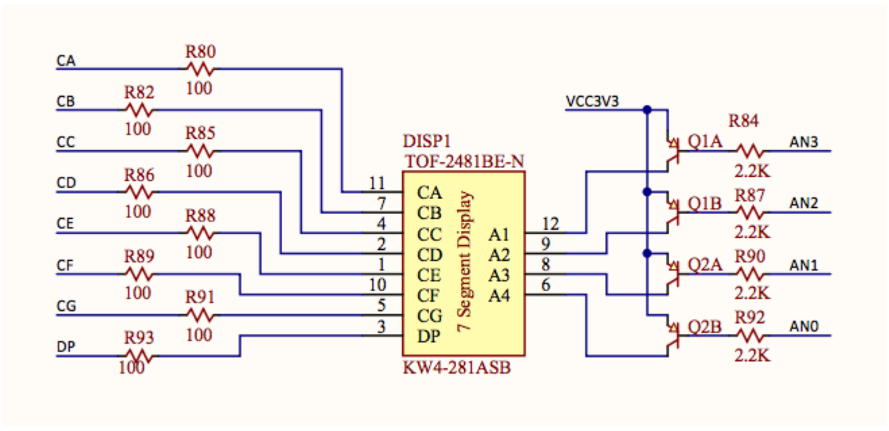
    b. Click **Program**.

    c. Test your system. Did it do what you expected?

    ______________________________________

# Combinational Logic

Simple assignments demonstrate the parallel nature of FPGAs, but combinational logic makes it much more useful. We’re going to write a small module (like a procedure) that will convert the binary value shown on the LEDs into a hex digit on the 7-segment display.

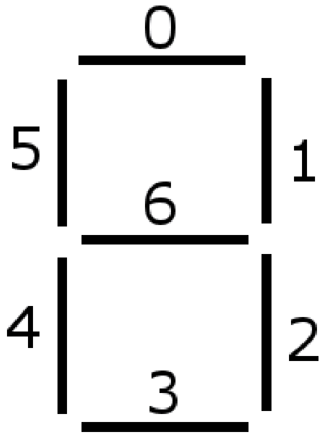


There are 7 **`seg[]`** output wires that control which lines are on or off, and correspond to **CA** to **CG** on the left. There are 4 **`an[]`** output wires that choose which digit(s) are on, and correspond to **AN0** to **AN3** on the right.

To display 4 different characters, we need to cycle through them fast enough so that they persist.

All of the code to generate the clock (**`clkdiv.v`**) and multiplex the display (**`seven_seg_mux.v`**) is already written and included in the project, giving direct access to each of the displays as **disp0**, **disp1**, **disp2**, and **disp3**.

1. First, let’s connect the stubbed out nibble_to_seven_seg module into our design:
   a. Find **nibble_to_seven_seg** in **WTFpga.v**. The module is instantiated, but not connected to anything.
   b. Connect the low 4 bits of our switches to the nibblein field: **.nibblein(sw[3:0]),**
   c. Connect the 8 bits of output to the seven segment display: **.segout(disp0)**
2. Next, let’s duplicate it so we can see a full hex byte:
   a. Cut-and-paste a new **nibble_to_seven_seg**
   b. Rename the new one from **nibble0** to **nibble1**
   c. Connect the next 4 bits of the switch: **sw[7:4]**
   d. Connect the display via **disp1**.
   e. Repeat for **disp2** and **disp3**.
3. Now, let’s implement **nibble_to_seven_seg**:
   a. Open **nibble0** from the project browser.
   b. We’re going to use a case statement, which works as in most programming languages. See the example in the comments of the code.
   c. For each case, we need the expected value of **nibblein**, and the right value of **segout**, an array of bits that represent the segments of the display as shown on the next page.



d. When a bit is ‘0’, it is ON and illuminated, when it’s ‘1’ it is OFF. Since that’s not very straightforward, we can use the **~** operator for bitwise inversion so that ‘1’ means an LED is illuminated.

e. For example, hex ‘1’ looks like **~7’b0000110**. We can express this as:
**4’h1: segout=~7’b0000110;**
which roughly translates to:

| | |
|---|---|
| **4’** | when our 4 bits |
| **h1:** | equal hex 0x01 |
| **segout=** | assign a value to segout |
| **~7’b** | of seven inverted bits |
| **0000110;** | leds 1 and 2 illuminated |

f. Figure out what you need to set for each of the hex values using the diagram.

4. **Generate Bitstream**, then **Program Device**. Does it work?

# Registers

While there is so much more to combinational logic than we actually touched on, let's move on to a new concept - Registers. Assignments are excellent at connecting blocks together, but they're more similar to passed parameters to a function than actual variables. Registers allow you to capture and store data for repeated or later use.

We will use a register to store the value we set on the DIP switches for later use

1. Open WTFpga.v
2. First, let's add a register to store our value.
    a. Find the area for wire and reg definitions
    b. Add a register for **storedValue**:
       **reg [15:0] storedValue;**
3. Find the **always @** block
    a. **always @** is the header for a synchronous block of code. Like software, commands get executed in order inside the always block
    b. The list in parenthesis following **@** is the **Sensitivity List**. Whenever one of the signals in the sensitivity list changes, the block is run. It is similar to binding a callback function or mapping an interrupt in software terms
    c. Let's put **posedge btnU** in the sensitivity list. This means each positive edge - every time

**btnU** goes from low to high - we will execute this block:

**always @ (posedge btnU)**

d. Now, assign the value of the switches to storedValue inside the always block:

**storedValue<=sw;**

e. Note that assignments are different inside the always block - we are setting a register, not assigning a wire anymore!

4. We need to display this value somehow. Let's use a different button to show the stored value, and use the let's use the ternary operator (**a?b:c**, meaning "if **a** then **b** else **c"**) to decide whether to display the current or stored value:

   a. In the wire section, let's create a new **wire**, dispval, to represent the value we want to see on the display: **wire [15:0] dispValue;**

   b. In the assignment section, we'll use the ternary operator to select the right value of dispval:

      **assign dispValue = a?b:c;**

   c. Replace the test value, '**a**', with what we want to use to toggle the displayed value, **btnC**.

   d. Replace the true result, '**b**', with what we want to see when it's pressed, **storedValue**.

   e. Replace the false result, '**c**', with what we want to see when it's released, **sw**.

f. Adjust the nibblein values of your nibble0..3 modules to use **dispValue** instead of **sw**.

5. Generate your bitstream. While it works, consider:

    a. What do you press to store your switch values?

    ______________________________

    b. What do you press to display your stored value?

    ______________________________

    c. What happens if you press both at the same time?

    ______________________________

6. Program your device, and test it out:

    a. Do the display characters change any time you switch the DIPs?

    ______________________________

    b. Can you store a value and display it?

    ______________________________

    c. Does it work how you expected?

    ______________________________

# Calculating

Lets combine everything we've done so far with a little bit of glue and lipstick to make it into a basic 16-bit calculator. To do this we need to perform some math and display the result.

1. First let's get addition working:
    a. Add a wire for the result:

       **wire [15:0] sum;**

    b. Assign the result a value:
       **assign sum = sw + storedValue;**

2. Display the result on the display when **btnR** is pressed. The easiest way is to use a nested ternary operator:

   **assign dispValue = d?e:(a?b:c)**;

    a. Your new test is represented by '**d**', which in this case is **bntR**.
    b. Your new true condition is represented by '**e**' and is what will be displayed when your test is true, which in this case is **sum**.
    c. Your prior assignment is represented by **(a?b:c)** - keep what you already had in its place.

3. Generate your new bitstream. While it works, consider:

    a. How else could we have implemented this instead of a ternary operator?

    ______________________________________

    b. How would using an if-else tree complicate the design?

    ______________________________________

    Remember that **wires** are always connected, always concurrent assignments!

    c. How would using a case statement complicate the design? What would we switch based on?

    ______________________________________

    d. How could we format the ternary operator a bit more legibly?

    ______________________________________

    Note that we can add whitespace and break over multiple lines.

    e. What if it builds but doesn't do what we expect? What could we do to debug our design?

    ______________________________________

4. Eventually you'll be ready to program your device and test it.

    a. Does it work as you expected?

    ______________________________________

    b. What happens if you press multiple buttons?

    ______________________________________

    c. What if you toggle switches while pressing buttons?

    ______________________________________

5. Let's add subtraction. We only need to change two lines to have the device subtract when **bntL** is pressed:

    a. Define the wire to name your result

    b. Assign your wire the difference of your switches and stored word.

    c. Make an even more nested ternary operator. Consider formatting and adding line breaks to make it more legible.

    d. Generate and program your device. Did it work?

    ______________________________________

# Exploring More

You’ve now completed a basic calculator, that uses most of the core concepts used to design nearly all silicon devices in use today. If time permits, here are a few additional things you can explore or try with this board:

- What happens in overflow and carry situations? Can you make something different happen?
- Can you display something other than numbers?
- Examine the WTFpga.xdc file. This contains all the mappings of the FPGA’s pins to the names you use in your code.
- Add additional math functions to your calculator. How about bitwise operators like AND, OR, and XOR? To do this, you will need to uncomment lines corresponding to additional button inputs in WTFpga.xdc
- Examine clkdiv.v by clicking on displayClockGen in the project browser. This is a clock divider that divides the 50mhz reference clock into lower speed clocks used internally. What does it do? How does it seem to work?
- Modify clkdiv.v to speed up or slow down the clock. What happens?
- Modify the design to flash the LEDs.
- Modify the design to PWM fade the LEDs.
- Examine the seven_seg_mux.v file. What does it do? how does it seem to work? Can you dim the display?

# Exploring on Your Own

What's next? Here are some recommendations for doing some FPGA and Verilog work on your own:

PicoEVB, TinyFPGA, and Beaglewire are three current FPGA-based CrowdSupply projects. They have a variety of purposes and toolchains, so check out each of them:

PicoEVB: https://www.crowdsupply.com/rhs-research/picoevb
TinyFPGA: https://www.crowdsupply.com/tinyfpga/tinyfpga-bx
BeagleWire:
https://www.crowdsupply.com/qwerty-embedded-design/beaglewire

Digilent and Numato build FPGA boards for education and prototyping. These have more features built in but are priced accordingly.

Digilent: http://www.digilentinc.com/
Numato: http://numato.com/

Lattice semiconductor also has some very low-end, but also low-cost FPGA development boards, as low as $20. There is also a lightweight, open source toolchain available:

Icestorm: http://www.clifford.at/icestorm/

Hamsterworks has a wiki full of many things to do with FPGAs including multiple multi-part FPGA courses and projects: http://hamsterworks.co.nz/mediawiki/index.php/Main_Page

Asic-world has a verilog guide along with examples and references: http://www.asic-world.com/verilog/veritut.html