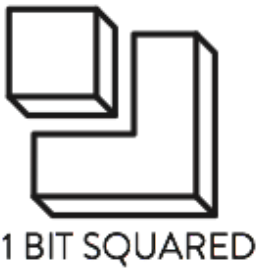


35C3: Leipzig 2018

WTFPGA?

https://github.com/esden/WTFpga

Developed by Joe FitzPatrick
joefitz@securinghardware.com

FOSSified and Presented by
Piotr Esden-Tempski
piotr@1bitsquared.com

Clifford Wolf
clifford@symbioticeda.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 or massively-parallel tasks. If you need to repeatedly process a consistent amount of data with minimal delay, an FPGA would be a good choice. Signal and graphics processing problems, often done with GPUs if power and cost are no object, are often easy to parallelize and FPGAs allow you to widen your pipeline until you run out of resources. 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. **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.

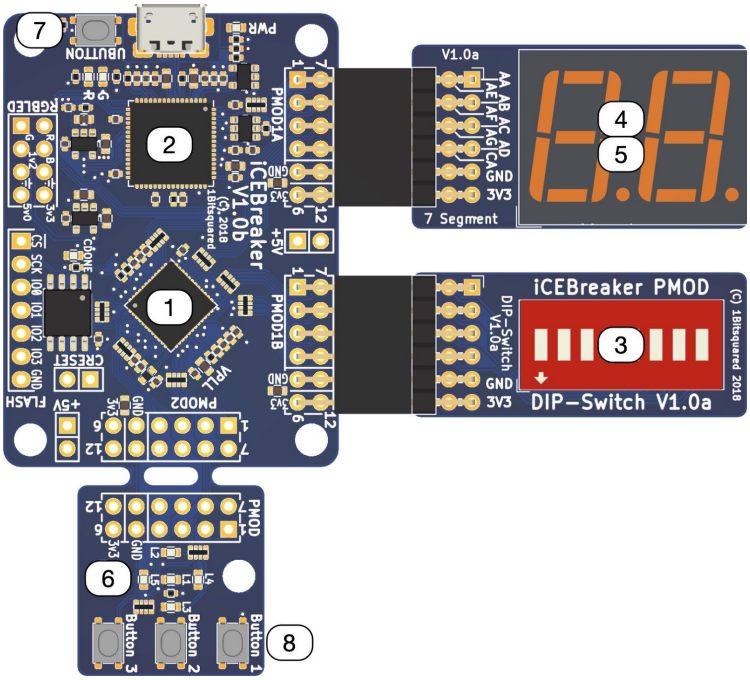
2. **IP Cores**: FPGA vendors pre-build or automatically generate code to let you easily interface your FPGA to interfaces like RAM, network, or PCIe. We'll stick to LEDs and switches today.

3. **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.

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

Meet the Hardware

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



1. iCE40UP5K FPGA
2. USB Programmer & USB to UART adapter
3. DIP Switches **sw[7:0]**
4. 7-segment display segments **seg[6:0]**
5. 7-segment cathode select **ca**
6. LEDs **led[4:0]**
7. User (negative logic) button **BTN_N**
8. More Buttons **BTN1, BTN2, BTN3**

http://bit.ly/csicebreaker

1. Take a look at the board, and identify each of the components listed above. There are a few components not listed that we won't use in this workshop.

2. Power on your laptop and connect the USB cable from a USB port to the connector on the iCEBreaker board. The FPGA will automatically load a demo configuration from the onboard serial flash. Play around:

    a. What do the pushbuttons do?

    b. Do you see the blink patterns on the LEDs?

Setup the Software

You will need a Linux or Mac OS computer for this workshop. We will install the Yosys, nextpnr and icestorm tools. Before you can install them you will need a few additional dependencies.

On Debian or Ubuntu you need the following dependencies:

```
sudo apt install build-essential clang bison \
      flex libreadline-dev gawk tcl-dev libffi-dev \
      git mercurial graphviz xdot pkg-config \
      python python3 libftdi-dev qt5-default \
      python3-dev libboost-dev git
```

On Mac OS you will need homebrew and the following dependencies:

```
brew install cmake python boost boost-python3 qt5 git
```

Now that you have all the dependencies you can clone our workshop repository:

```
git clone https://github.com/esden/WTFpga.git
cd WTFpga
```

In the workshop repository you have to run the `summon-fpga-tools.sh` script and set your `PATH` so it can find the tools you just installed:

```
./summon-fpga-tools.sh
export PATH=~/sft/bin:$PATH
```

That is it now you are ready enter the wtfpga subdirectory and you are ready to launch into the workshop! :)

```
cd wtfpga
```

Configuring an FPGA

Now that we are familiar with the hardware as-is, let's walk through the process of synthesizing an FPGA design of your own and uploading it to the iCEBreaker. We will be using the amazing open source tools called icestorm, nextpnr and Yosys. We have configured and set up everything for you to get you going quickly.

1. Power on your laptop and open the terminal:

    a. Open the command line terminal by clicking on the terminal icon in the task bar at the bottom of the laptop screen.

    b. Enter the WTFPGA workshop directory by typing **cd wtfpga**

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

    d. Run the build and upload process by executing make in the directory by typing: **make prog**

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

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

Reading Verilog

Now that we know how to use the tools to configure our FPGA, 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. Open wtfpga.v (in the repository we cloned previously) in the text editor of your choosing.

    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.

2. 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 **BTN1** and **BTN3** 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. Now we will start using the DIP switches on one of the PMOD modules. We will be connecting them to the 5 LEDs found on the break-off section of the iCEBreaker. Let's change the assignment from one pushbutton (**BTN1 and BTN3**) to the switches (**sw[4:0]**). 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[4:0]=sw[4:0];
    ```
    d. Save the file.
2. Next, we need to create a new configuration for our FPGA. Brace yourself - it will be really quick! ;-)
    a. In the command line terminal that we opened earlier, type **make prog** or use the up arrow on your keyboard to recall the previous command, then press **enter**.

3. You should see some text scroll by and the new design should be uploaded and running within a few seconds. If we were using proprietary tools (Vivado or Quartus) as we did in V1 and V2 of this workshop the synthesis would take **~8 minutes** depending on the computer used. We used these 8 minutes to talk about the synthesis process itself. Even though we don't have to wait that long, let's talk what the software is doing and what tools are used to accomplish those steps. It is bit different from a software compiler.

    a. First, the software will **synthesize** the design - turn the Verilog code into basic logical blocks, optimizing it in the process using yosys. (http://www.clifford.at/yosys/)

    b. Next, the tools will **implement** the design. This takes the optimized list of registers and assignments, and **places** them into the logical blocks available on the specific FPGA we have configured, then **routes** the connections between them using the tool called nextpnr. (https://github.com/YosysHQ/nextpnr)

    c. When that completes, the fully laid out implementation needs to be packaged into a format for programming the FPGA. There are a number of options, but we will use a .bit bitstream file for programming over **SPI** using **icepack** from the **icestorm** tool collection. (http://www.clifford.at/icestorm/)

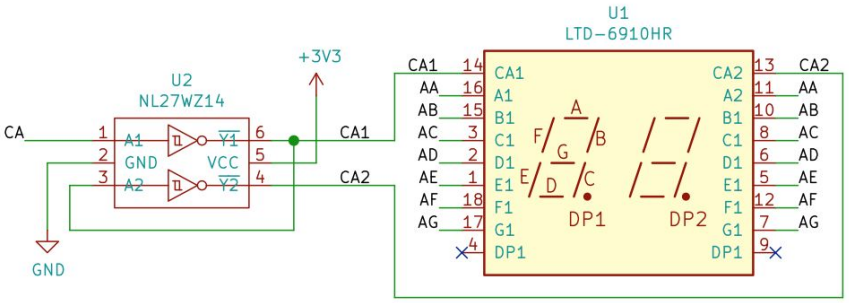
d. Hopefully everything will go as planned. If you have issues, look in the console for possible build errors. If you have trouble, ask for help!

e. Finally, the .bin file needs to be sent to the FPGA over USB. When this happens, the demo configuration will be cleared and the new design will take its place using the **iceprog** tool.

f. 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 LEDs are on or off, and correspond to **AA** to **AG** on the left and right of the display. There is one **`ca`** output wire that uses two inverting buffers to select which digit is on, and corresponds to **CA** on the left.

To display 2 different characters, we need to cycle between them fast enough so that they persist in the eye.

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** and **disp1**.

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 to **sw[7:4]**.

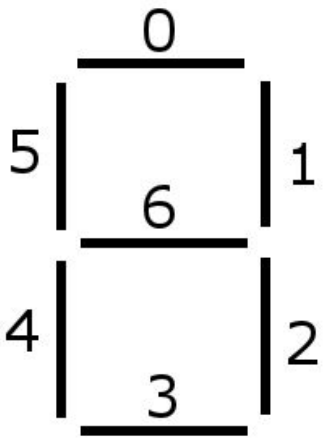
    d. Connect the display via **disp1**.

3. Now, let's implement **nibble_to_seven_seg**:

    a. Open **seven_seg_hex.v** file in the your text editor.

    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. Now **make prog** your design. 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** source file in your text editor.
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 [7:0] 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 **negedge BTN_N** in the sensitivity list. This means each negative edge - every time **BTN_N** goes from high to low - we will execute this block:

always @ (negedge BTN_N)

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. We can 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**, dispValue, to represent the value we want to see on the display: **wire [7:0] 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, **BTN2**.

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..1 modules to use **dispValue** instead of **sw**.

5. Generate your bitstream and upload, after that consider the following questions:

   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?

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

   e. Can you store a value and display it?

   f. Does it work how you expected?

Calculating

Let's combine everything we've done so far with a little bit of glue and lipstick to make it into a basic 8-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 [7:0] sum;

    b. Assign the result a value:

    assign sum = sw + storedValue;

2. Display the result on the display when **BTN1** 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 **BTN1**.

    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. Before you synthesize and upload the resulting design consider the following questions:

    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. Now go ahead and synthesize your design and upload:

    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 implement subtraction. We only need to change two lines to have the device subtract when **BTN3** 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 **icebreaker.pcf** 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 come up with different schemes of accessing them. Maybe combinations of buttons?
- Examine **clkdiv.v** in your text editor. This is a clock divider that divides the 12MHz 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?

Thank You and Final Notes

What's next? I hope you liked and enjoyed this really speedy dive into FPGA development. As you might know by now the iCEBreaker is currently crowdfunding on CrowdSupply. If you want to help us out, make sure to take photos of your iCEBreaker and share them on your social media, linking to our campaign:

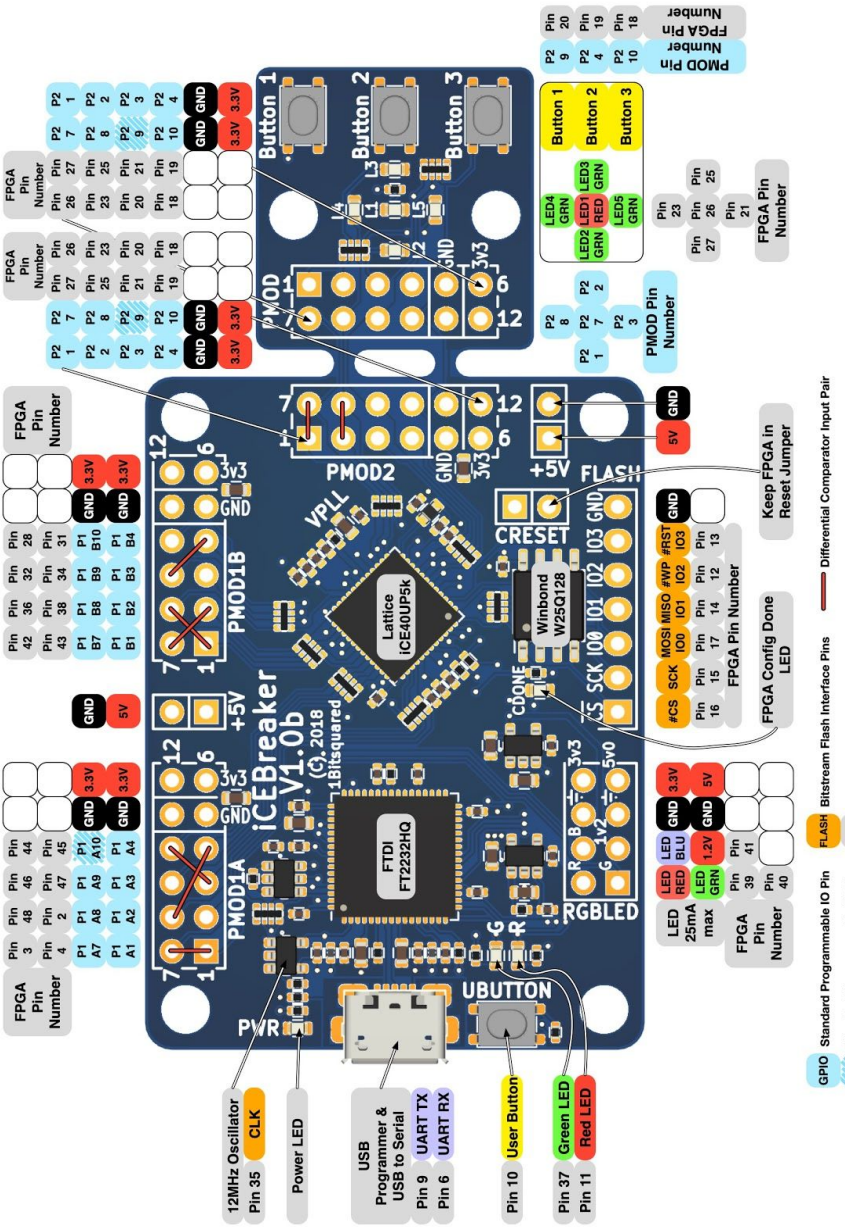
https://www.crowdsupply.com/1bitsquared/icebreaker-fpga

If you continue playing with your iCEBreaker make sure to share with us what you built, and ask questions. You can find us on Gitter, in the iCEBreaker channel:

https://gitter.im/icebreaker-fpga/Lobby

If you want some further reading and more examples, check out the iCEBreaker examples repositories. If you come up with additional examples make sure to send us a pull request! :D

https://github.com/icebreaker-fpga


iCEBreaker V1.0b
(C) 2018 1BitSquared
PMOD1A
PMOD1B
PMOD2
FTDI FT2232HQ
Lattice iCE40UP5k
Winbond W25Q128
RGBLED
UBUTTON
PWR
CRESET
FLASH
VPLL
Button 1
Button 2
Button 3
12MHz Oscillator
Pin 35
CLK
Power LED
USB Programmer & USB to Serial
Pin 9
UART TX
Pin 6
UART RX
Pin 10
User Button
Pin 37
Green LED
Pin 11
Red LED
FPGA Pin Number
PMOD Pin Number
FPGA Pin Number
FPGA Config Done LED
Keep FPGA in Reset Jumper
FPGA Pin Number
LED 25mA max
FPGA Pin Number
GPIO Standard Programmable IO Pin
GPIO Standard Programmable IO Pin
GBIN Global Net Input Pin
FLASH Bitstream Flash Interface Pins
FPGA FPGA Package Pin Number
Differential Comparator Input Pair