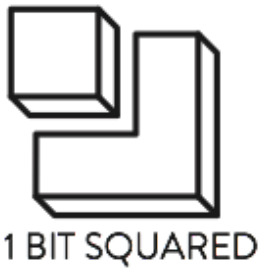


Teardown Portland 2019

iCEBreaker Workshop

https://github.com/icebreaker-fpga/icebreaker-workshop

Original idea by Clifford Wolf
clifford@symbioticeda.com

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

iCEBreaker Hardware sponsored by
Lattice Semiconductor
Latticesemi.com

Inspired and adapted from WTFPGA workshop by
Joe FitzPatrick
joefitz@securinghardware.com

This page intentionally no longer blank. ;)

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.

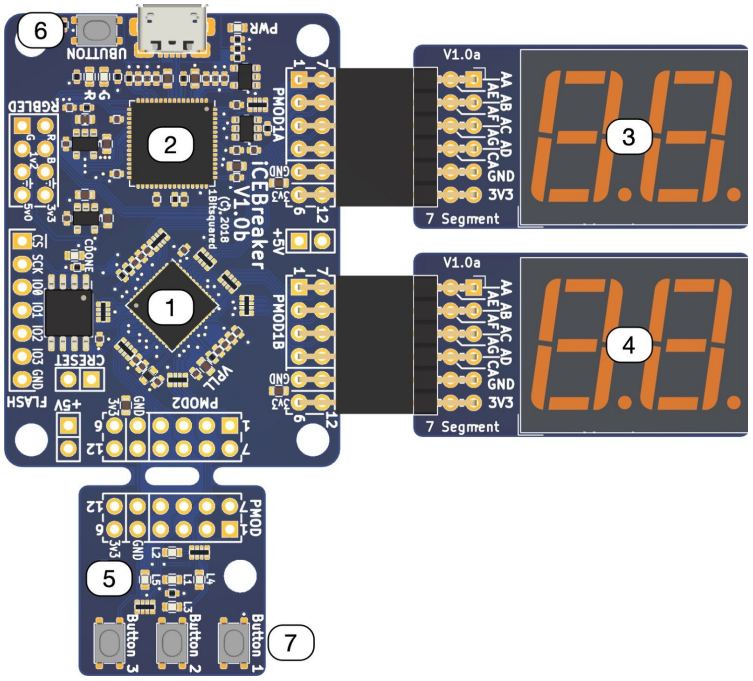
What We Won't Learn

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. **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.
2. **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.
3. **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. 7-segment display Pmod 1A **P1A1-4, P1A7-10**
4. 7-segment display Pmod 1B **P1B1-4, P1B7-10**
5. LEDs **LED1-5**
6. User (negative logic) button **BTN_N**
7. More Buttons **BTN1, BTN2, BTN3**

https://icebreaker-fpga.com/

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

If you are provided with a laptop for the workshop you can skip this page and move on to the fun stuff. If you are using your own laptop for the workshop, you will need a Linux or Mac OS computer. We do not recommend using windows or linux for windows for the workshop.

For the software stack installation instructions enter the sft subdirectory and open the README.md file.

```
cd ../sft
vi README.md
```

You can also find the installation readme online at:
https://github.com/icebreaker-fpga/icebreaker-workshop/tree/master/sft

If you are having any more trouble, or you feel like the build script included in the workshop might be out of date, make sure to check out the official summon-fpga-tools repository at:
https://github.com/esden/summon-fpga-tools

After you are done installing the toolchain you can, enter the `stopwatch` subdirectory and you are ready to launch into the workshop! :)

```
cd ../stopwatch
```

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 iCEBreaker workshop directory
    by typing **cd icebreaker-workshop/stopwatch**

    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 **stopwatch.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 and drive the 7-segment displays.

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 called?

    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** push button 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 and Combinatorial Logic

Let's start with some minor changes to our Verilog, then configure our board. We will be changing the way the three buttons (**BTN1-3**) on the "head" Pmod of the iCEBreaker are affecting the five LED (**LED1-5**):

1. Change the assignments for **LED1..LED5** so that
    a. **LED1** is on when buttons 1 and 2 are pressed.
    b. **LED2** is on when buttons 1 and 3 are pressed.
    c. **LED3** is on when buttons 2 and 3 are pressed.
    d. **LED4** is on when the "user button" (**BTN_N**) is pressed. (Note that this button is inverted, i.e. the value of **BTN_N** is zero when the button is pressed.)
    e. **LED5** is on when any of the four buttons is pressed.
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 the early versions of the WTFPGA workshop V1 and V2, that this workshop is the descendant of, 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 about what the software is doing and what tools are used to accomplish those steps. It is a 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/)

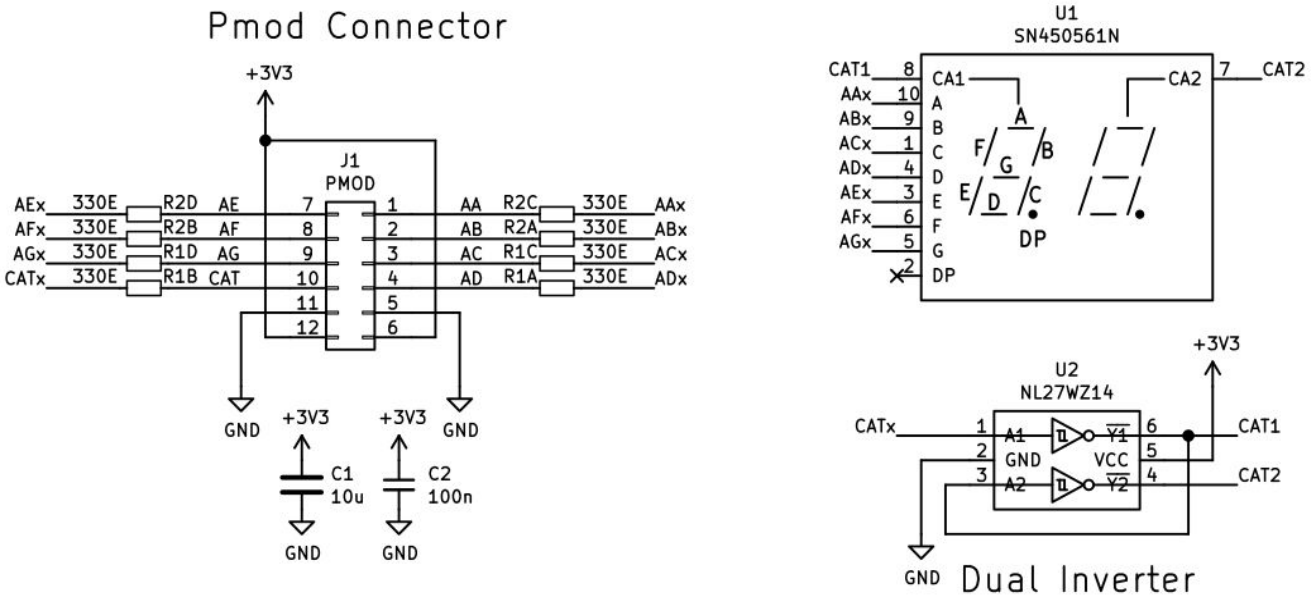
d. Hopefully everything will go as planned. If you have issues, look in the console for possible build errors. If you have trouble, ask for help!

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?

Add missing 7-segment digits

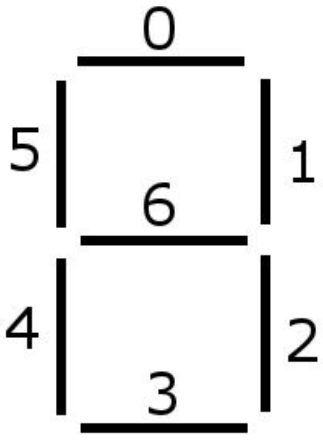
The 7-segment display Pmod modules that are attached to your iCEBreaker have the following schematic.



As you can see in the verilog code the pins of the Pmod A and B get assigned a vectors of wires called **seven_segment_top** and **seven_segment_bot** to map the Pmod pin names to a bit vector. This makes handling and passing of the segments to modules easier.

These vectors are being fed through the module **seven_seg_ctrl** that switches very quickly between the two digits to create the impression that both segments are illuminated at the same time. That module in turn uses the **seven_seg_hex** module. The module **seven_seg_hex** is converting a 4-bit binary number into a 7-bit seven segment control vector

The mapping of the segment vector bits to the segments looks like this:



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

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

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

2. We are missing the definitions for digits **3** and **8**. Go ahead and add them based on the above diagram.

3. Now **make prog** your design. Does it work?

Switch to decimal counting

The module **bcd16_increment** reads a 16-bit BCD (Binary Coded Decimal) number (a decimal digit in each nibble) and increments it by one. Replace the line **assign display_value_inc = display_value + 1;** with an instance of **bcd16_increment** so that the stop watch counts in decimal instead of hexadecimal.

(See the instances of **seven_seg_ctrl** for how to instantiate a module.)

Add RESET button

Add an if-statement to the **always @(posedge CLK)** block in the top module that will reset **display_value** to zero when the "user button" (**BTN_N**) is pressed.

Add START/STOP buttons

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 passing parameters to a function than actual variables. Registers allow you to capture and store data for repeated or later use.

Add a (1 bit) **running** register (initialized to zero), and change the code that increments **display_value** to only apply the increment when running is **1**.

Add if-statements to the **always @(posedge CLK)** block in the top module that will set **running** to **1** when **BTN3** is pressed, and reset running to **0** when **BTN1** is pressed. Now these two buttons function as START and STOP buttons for the stop watch.

Also change the RESET functionality so that **running** is reset to **0** when the "user button" (**BTN_N**) is pressed.

Add lap time measurement

Finally let's also add lap time measurement: pressing the center button on the board should display the current time for two seconds while we keep counting in the background.

For this, we need to add a 16 bit register **lap_value** and an 8 bit register **lap_timeout**.

lap_timeout should be decremented in every **clkdiv_pulse** cycle until it reaches zero. The seven segment display should show the value of **lap_value** instead of **display_value** when **lap_timeout** has a nonzero value.

Pressing the center button (**BTN2**) should set **lap_timeout** to **200** and copy the value from **display_value** to **lap_value**.

Note: The syntax **x = a ? b : c** can be used to assign value **b** to **x** when **a** is nonzero, and value **c** otherwise.

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 stopwatch, like switching to displaying minutes and seconds when the seconds counter rolls over.
- Modify the design to flash the LEDs.
- Modify the design to PWM fade the LEDs.
- Add display dimming.

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 available for pre-order 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:

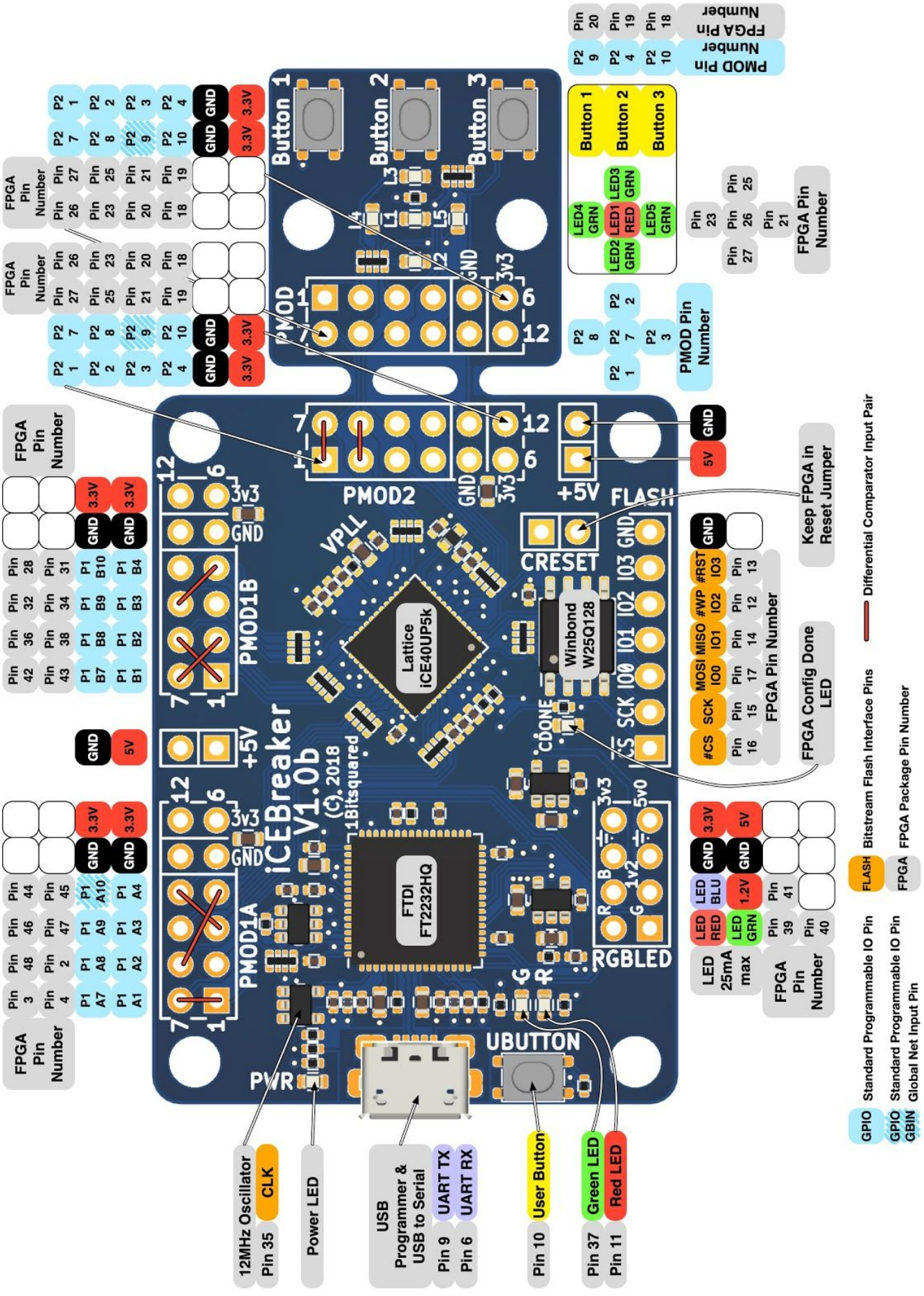
https://www.crowdsupply.com/1bitsquared/icebreaker-fpga

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
PMOD
Button 1
Button 2
Button 3
FTDI FT2232HQ
Lattice iCE40UP5k
Winbond W25Q128
RGBLED
UBUTTON
PWR
FLASH
CRESET
CDONE
VPLL
FPGA Pin Number
PMOD Pin Number
FPGA Config Done LED
Keep FPGA in Reset Jumper
LED 25mA max
12MHz Oscillator
Power LED
USB Programmer & USB to Serial
User Button
Green LED
Red LED
UART TX
UART RX
Standard Programmable IO Pin
Standard Programmable IO Pin
Global Net Input Pin
Bitstream Flash Interface Pins
FPGA Package Pin Number
Differential Comparator Input Pair