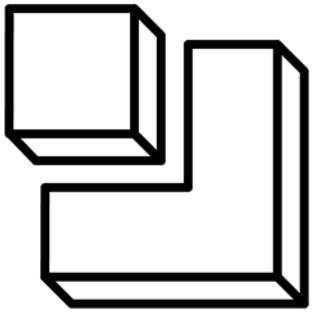
iCEBreaker FPGA Stopwatch Workshop

Piotr Esden-Tempski, Sylvain Munaut, Mike Walters and friends

Version 2.0, 27 Dec 2019



1 BIT SQUARED

Table of Contents

Overview

Course Objectives

This lab manual is a workbook that accompanies the iCEBreaker FPGA stopwatch workshop.

Our objective is for attendees to get as much hands-on time with hardware as possible during our classes. Lectures serve as introductions to topics, but the walk-throughs and challenges in this manual will lead you to deeper understanding and higher retention of important information. We recommend recording as much as possible in this manual, as it may become a resource for your future hardware hacking adventures.

This training is based on the SecuringHardware.com model developed by Joe FitzPatrick. If you enjoy this workshop make sure to check out the list of workshops SecuringHardware.com offer. This workshop was developed by 1BitSquared for the iCEBreaker FPGA development board. We also develop other workshops. You can find the WTFpga and iCEBreaker workshops on GitHub at https://github.com/icebreaker-fpga.

Chapter 1. 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, micro controllers 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.

If you have any questions or run into trouble let any of the helpers know, we are here to help. If you are working on this outside of an organized workshop join the 1BitSquared 1BitSquared discord [https://1bitsquared.com/pages/chat] and ask away in the **icebreaker** and/or **fpga** channels.

1.1. What We Won't Learn

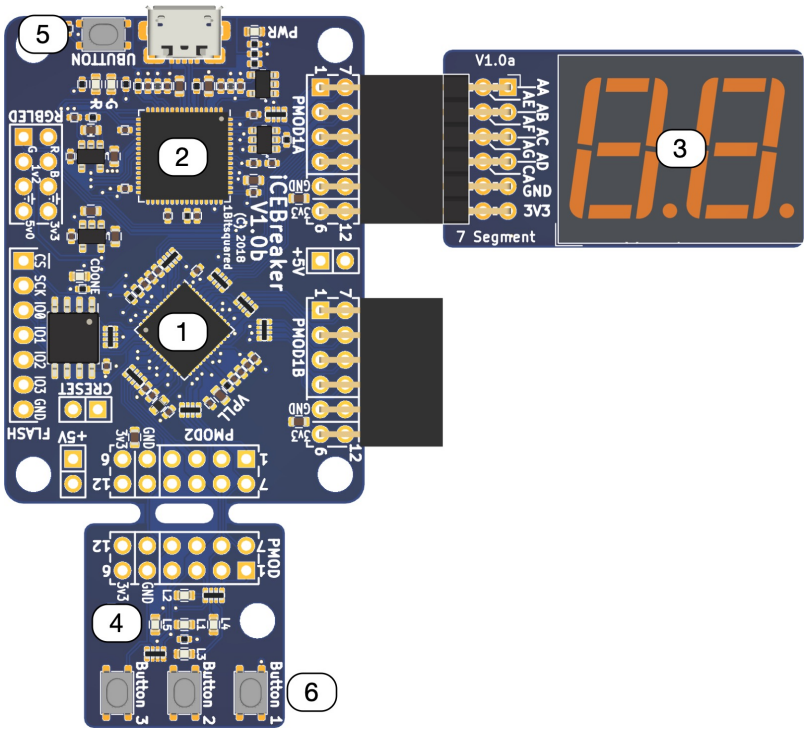
In order to introduce Verilog and FPGAs in such a short time, we're going to skip over several things that will be important when you build your own FPGA-based designs, but are not necessary to kickstart your tinkering:

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.

1.2. Hardware Overview

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 & UART adapter
3. 7-segment display Pmod 1A `P1A1-4, P1A7-10`
4. LEDs `LED1-5`
5. User (negative logic) button `BTN_N`
6. 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. Connect the USB cable from your laptop 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?

Chapter 2. Setting Up

2.1. Get the toolchain

If you did not already you will need to install the FPGA toolchain first.

Follow the instructions for the fomu-toolchain [https://github.com/im-tomu/fomu-toolchain].

If you are having trouble with the setup due to internet connectivity the helpers might have a USB stick you can use to copy the toolchain and the workshop materials to your laptop. You will need to unzip the toolchain archive and add the `bin` directory to your `PATH` environment variable.

2.2. Toolchain Troubleshooting

I am running iceprog and the programmer is not being detected

Linux

- Check if the device is being detected by the kernel with 'lsusb' it will either show up as a Future Electronics device or the name of the programmer vendor.
- If the device is being detected by the kernel you might not have permissions to access the device. If you run `sudo iceprog ⋯` and the device is decected you can give yourself permissions by creating a udev file at: `/etc/udev/rules.d/53-icebreaker-ftdi.rules` and adding the following line in that file:

```
ATTRS{idVendor}=="0403", ATTRS{idProduct}=="6010", MODE="0660", GROUP="plugdev",
TAG+="uaccess"
```

After adding that file you need to at least replug the programmer or even reload the udev rules.

Windows

You will need the zadig drivers [https://zadig.akeo.ie], choose icebreaker interface 0 and then choose winusb. If the icebreaker interface does not show up, check that "List all devices" is ticked under "Options".

2.3. Get the workshop repository

You probably already have it on your drive as you are reading this, but for good measure here is how you obtain the workshop git repository:

```
git clone https://github.com/icebreaker-fpga/icebreaker-workshop.git
cd icebreaker-workshop/stopwatch
```


Chapter 3. Your first digital design

Now that everything is set up let's dive into it and build our first digital design.

3.1. 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.

1. Open the command line terminal:
    a. Enter the iCEBreaker stopwatch workshop directory directory by typing `cd icebreaker_workshop/stopwatch`
    b. If it's not already connected, plug your iCEBreaker board into your laptop with a USB cable.
    c. Run the build and upload process by executing make in the directory by typing: `make prog`
    d. Confirm that you were able to replace the default configuration with a new one by checking to see if the behavior has changed.
    e. Test the buttons and switches. Does this new configuration do anything useful?

3.2. 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 is one already instantiated that drives the 7-segment display.
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.

3.3. 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 simultaneously.
   b. `LED2` is on when buttons 1 and 3 are pressed simultaneously.
   c. `LED3` is on when buttons 2 and 3 are pressed simultaneously.
   d. `LED4` is on when the button `BTN_N` is pressed. (Note that this button is inverted, i.e. the value of `BTN_N` is zero when 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 about 8 minutes depending on the computer used. We instead 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 .bin bitstream

file for programming over SPI interface 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!

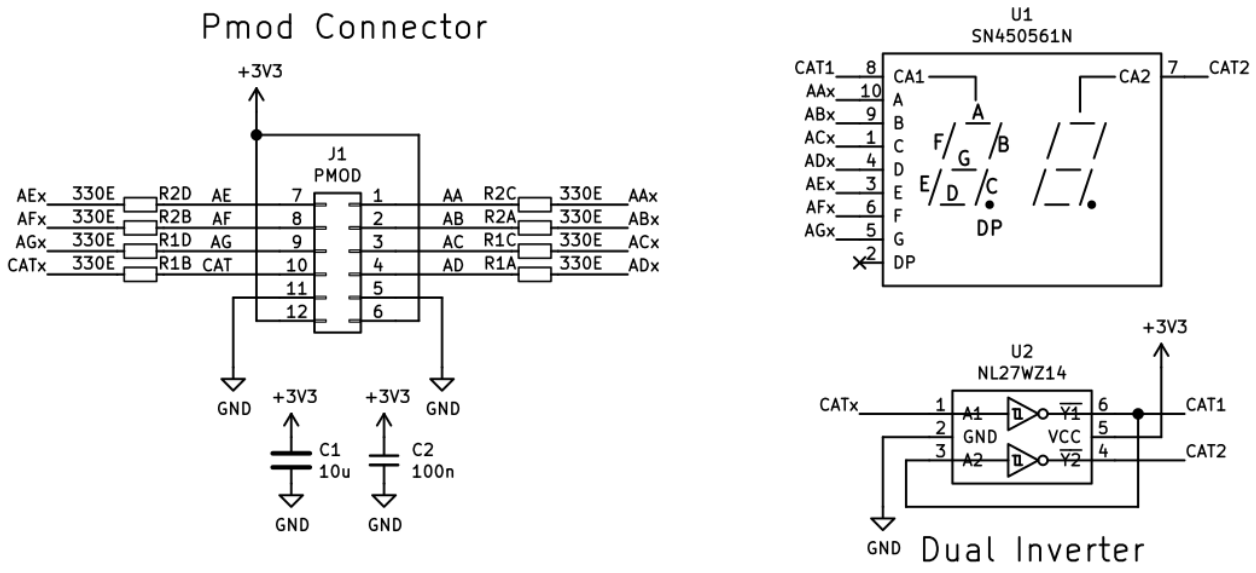
e. Finally, the .bin file needs to be sent to the FPGA over USB. When this happens, the demo configuration will be cleared and the new design will take its place using the **iceprog** tool.

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

3.4. Add missing 7-segment digits

If you have not done that already, this is the time to solder on the missing Pmod connector closest to the USB connector (marked Pmod1A) on your iCEBreaker. After you do that you should plug in the 7-segment display Pmod.

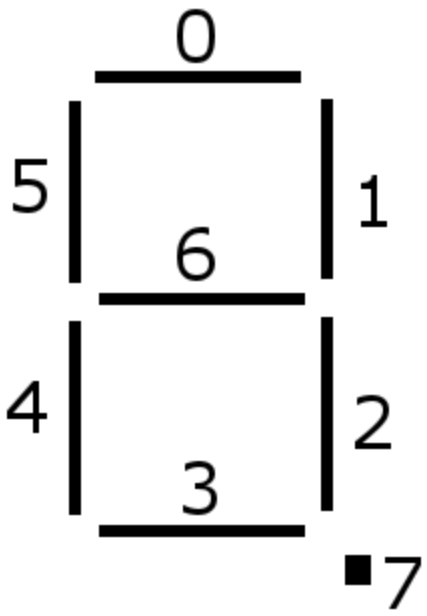
The 7-segment display Pmod module that is attached to your iCEBreaker has the following schematic.



As you can see in the Verilog code the pins of the iCEBreaker Pmod get assigned a vector of wires called `seven_segment`.

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 dout
`7'b`	of seven bit
`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?

3.5. Switch to decimal counting

The module `bcd8_increment` reads a 8-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 `bcd8_increment` so that the stopwatch counts in decimal instead of hexadecimal.

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

3.6. 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.

3.7. 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` so that it only applies 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` and `display_value` is reset to `0` when the "user button" `BTN_N` is pressed.

3.8. Add lap time measurement

Finally let's also add lap time measurement: pressing the center button `BTN2` 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 8-bit register `lap_value` and an 5-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 `20` 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.

3.9. Exploring More

You've now completed a basic stop watch! While simple, this design demonstrates most of the core concepts needed to design almost all silicon devices in use today. If time permits, here are a few

additional things you can explore or try with this board:

- Can you change the design so that the digits flash when `lap_value` is being displayed?
- Can you make BTN3 toggle between START/STOP on each press?
- 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 stop watch, 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.

Chapter 4. Final notes

I hope you had fun with this workshop. There is so much more to learn about digital design and what you can do with iCEBreaker itself as well as FPGA development.

If you want to continue hacking with the iCEBreaker we have a few of them here with us if you want to get them. Just find Esden. :D

Definitely post on twitter and tag @esden if you do something cool with your iCEBreaker.

You can find more information about the iCEBreaker at 1BitSquared:

https://1bitsquared.com/products/icebreaker

If you continue playing with your iCEBreaker make sure to share with us what you built, and ask questions. You can find us on the 1BitSquared Discord server:

https://1bitsquared.com/pages/chat

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/

See you around! :D