WTFpga?

Developed by Joe FitzPatrick
FOSSified by Piotr Esden-Tempski and Clifford Wolf
Presented by Josh Johnson

September 9, 2019

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. Josh is here to answer questions, not drone on with text-laden slides like usual.

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.

2 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:

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

3 Meet the Hardware

We will be using the not creatively named iCE40-feather board for this workshop. It contains an iCE40UP5K FPGA, USB programmer and USB to UART converter, LiPo battery charge control, and plenty of LEDs in an Adafruit Feather compatible form factor. To extend its capabilities for this workshop, we will be attaching a FeatherWing which has a dual seven segment display, eight DIP switches, and a slide switch.

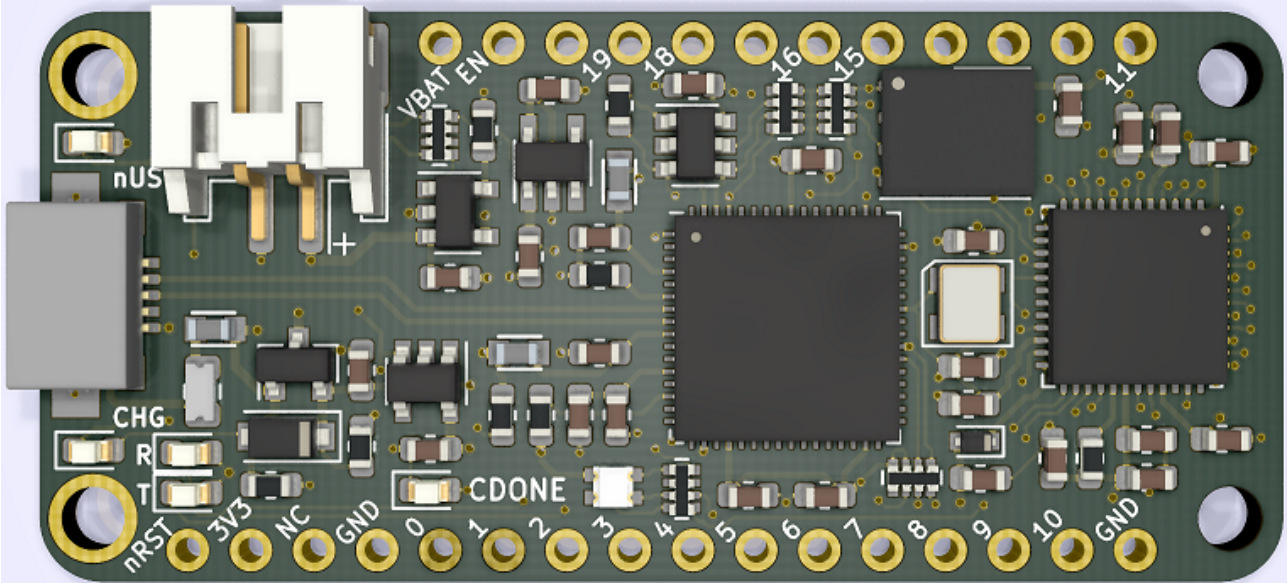

Figure 1: The iCE40-feather FPGA board being utilised.

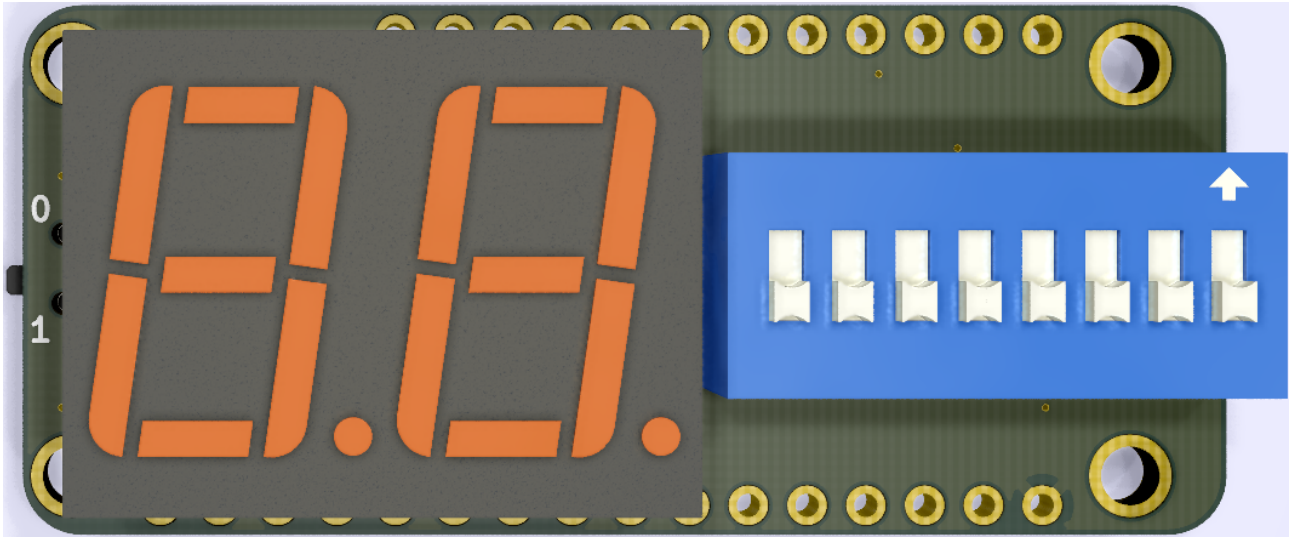

Figure 2: The seven segment and DIP switch FeatherWing.

4 Getting to Blinky

Due to time constraints, it is suggested that you install the toolchain and get a blinking LED on your iCE40-feather before attending the workshop. If you have not already installed the required yosys, nextpnr, and IceStorm tools, follow the instructions in `install.md`. If you do not have the FPGA dev board, read the instructions in `README.md` or get in contact with Josh.

With the toolchain installed and hardware in hand, it is now time to blink some LEDs! If you have not already done so, clone the repository and open the blink directory.

git clone https://github.com/joshajohnson/WTFpga
cd WTFpga/blink

Now its time to walk through the process of synthesizing an FPGA design of your own and uploading it to the FPGA. We will be using the amazing open source tools called IceStorm, nextpnr and yosys.

- Ensure that the FPGA is connected via a micro USB cable to your computer, and that the cable is not missing data pins.
- Open the command line in the **WTFpga/blink** folder.
- Build and upload the gateware by typing **make prog** into the terminal.
- Ensure that the led marked nUSR is blinking. If not it's time to begin troubleshooting!
  - NOTE: Windows users may need to update drivers used with the FTDI IC used for USB to Serial communications. Check the install notes in `install.md`.
- Once you get the LED blinking, open the file `blink.v` and have a look around. See if you can change the frequency of the blinking LED by altering the code.
- After changing `blink.v` and saving it, run **make prog** again and confirm the frequency changes.

If you have made it this far, congratulations! You have successfully programmed your FPGA, and can now attend the WTFpga workshop knowing that you'll hit the ground running. If you are having issues programming the board, get in contact with Josh and he will lend a hand troubleshooting.

5 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.

- **cd** to the directory **wtfpga-lab** and open **wtfpga.v** in the text editor of your choosing.
- Our module definition comes first, and defines all the inputs and outputs to our system. Can you locate them on your board?
- Next are wire definitions. Wires are used to directly connect inputs to one or more outputs.
- Next are parallel assign statements, all of these assignments are always happening, concurrently.
- In some designs, we would find always blocks. These are blocks of statements that happen sequentially, and are triggered by the sensitivity list contained in the following @(). We will run into them later in the lab.
- Finally we can instantiate modules. There are a few already instantiated but not really used for anything yet.

Now let's try and map our board's functionality to the Verilog that makes it happen.

- From the terminal we have open in the wtfpga-lab directory, type **make prog** and press enter to load the wtfpga project onto the FPGA.
- What happenes when you move the DIP switches?
- Can you find the DIP switches in the module definition? What are they named?
- Can you find the assignments which use each of the switches? What are they assigned to?
- Can you follow the assignments to an output?
- Do you notice anything interesting about the order of the assign statements?

You should be able to trace the DIP switches inputs, through a number of wires, to seven segment outputs. You should also note that the seven segment display is inverted, the reasoning of which will be explained later.

Note that these aren't sequential commands. All of these things happen at once. It dosen't actually matter what order the assign statements occur.

6 Making Assignments

Let's start with some minor changes to our Verilog.

- Remove all of the **assign seg[x] = sw[x];** assignments from the file
- Replace the previous assignments with **assign seg[6:0] = ~sw[6:0];**

Next, we need to create a new configuration for our FPGA. Brace yourself - it will be really quick!

- In the command line terminal that we opened earlier, type **make prog**, and press enter.

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 was done in V1 and V2 of this workshop, the synthesis would take approximately 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 a bit different from a software compiler.

- 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/)
- 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)
- 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/)
- 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!
- Finally, the .bin file needs to be sent to the FPGA over USB. When this happens, the existing configuration will be cleared and the new design will take its place using the iceprog tool.

Check what happens when you move the DIP switches. Does it function as you expect? What has changed from last time?

- Before continuing, remove the **assign anode = 2'b11;** and **assign seg[6:0] = ~sw[6:0];** statements from your design, and comment in the **displayMux** module.

7 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 DIP switches into a hex digit on the 7-segment display.

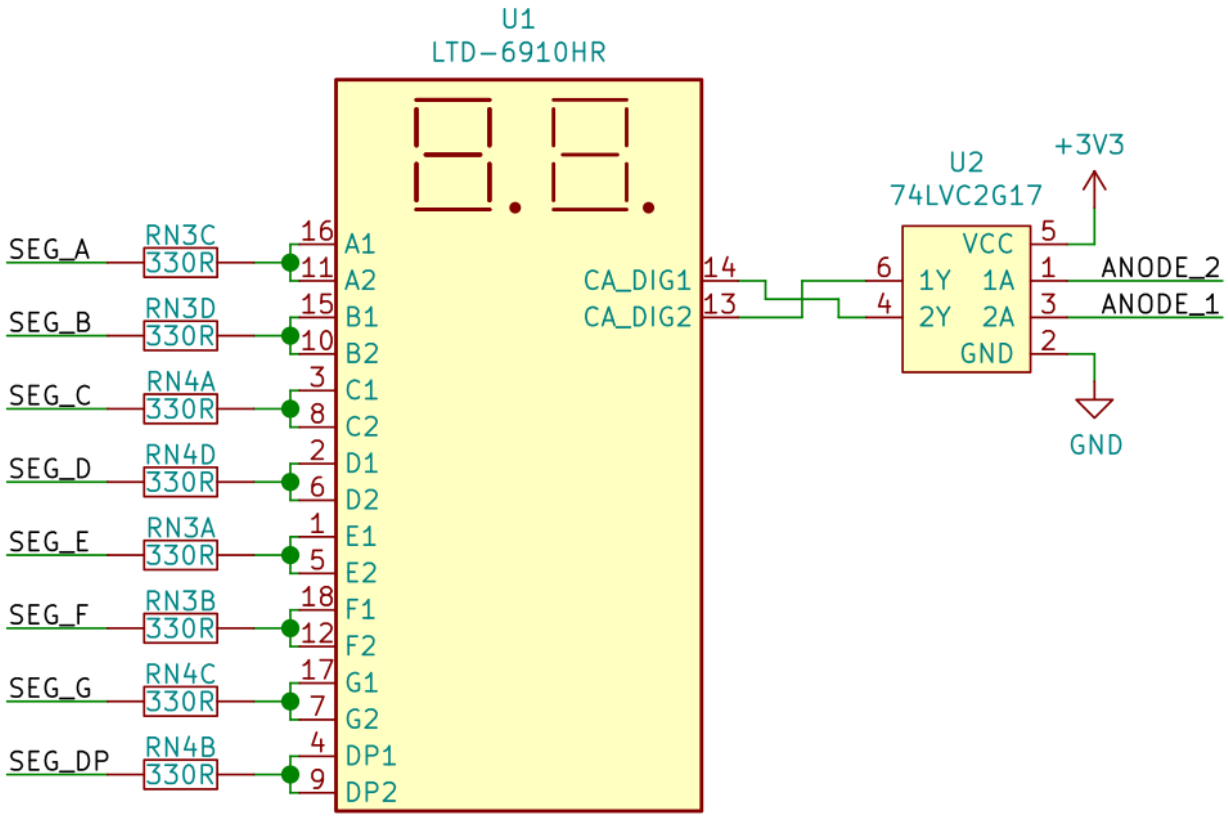

Figure 3: Schematic for the seven segment display.

There are 7 **seg** output wires that control which LEDs are on or off, and correspond from **A** to **G** on the display. There are two anode wires connected through buffers to select which digit is on, and corresponds to **CA_DIG1** and **CA_DIG2** on the right. To display two 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 (clockDiv.v) and multiplex the display (dispMux.v) is already written and included in the project, giving direct access to each of the displays as disp0 and disp1.

- First, let's connect the stubbed-out **nibbleDecode** module into our design.
  - Find **nibbleDecode** in **wtfpga.v**. The module is instanticated, but not connected to anything.
  - Connect the low 4 bits of our switches to the nibblein field: **.nibblein(sw[3:0]),**
  - Connect the 7 bits of output to the right most seven segement display: **.seg(disp1)**
- Next, let's duplicate it so we can see a full hex byte.
  - Copy and paste a new **nibbleDecode** instantiation.

  - Rename it from **nibbleDecodeLSD** to **nibbleDecodeMSD** as this will display the most significant digit (left most display).
  - Connect the upper 4 bits of the switches **sw[7:4]** to **nibblein**, and rename the output display to **disp0**.

- We now need to implement logic which converts our binary value into a character on the seven segement display.
  - Open **nibbleDecode.v** in your text editor.
  - We're going to use a case statement to convert a binary nibble into a series of bits which represent the seven segment display segements which need to be illuminated.
  - For each case, we need the expected value of **nibbleIn**, and the correct value of **seg**, an array of bits which represent the segments on the display shown below.

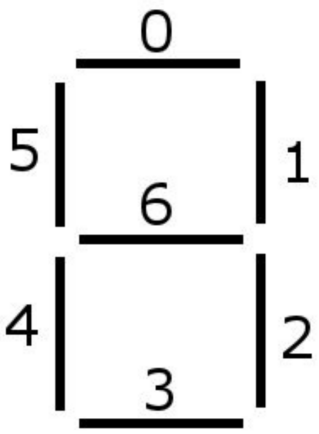

Figure 4: Mapping of seven segment segments to position. Typically labeled A through G.

  - Due to the display using a common anode, when a bit is '0', the corresponding segment is ON, and when a bit is a '1' the segment is OFF. Since that is confusing, we can use the $\sim$ operator for bitwise inversion such that '1' means the segment is illuminated.
  - For example, hex '1' looks like **$\sim$7'b0000110**. We can express this as **4'h1: seg = $\sim$7'b0000110**, which roughly translates to:

4'	when our 4 bits
h1:	equal hex 0x01
seg =	assign a value to seg
$\sim$7'b	of seven inverted bits
0000110	leds 1 and 2 illuminated

  - Figure out what you need to set for the remaining hex values using the diagram.

- Due to how included files are handled in this lab, you will have to **make clean** before running **make prog** after editing a file other than **wtfpga.v**.

- **make clean** then **make prog** your design. Does it work?

8 Registers

While there is so much more to combinatorial 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 pass paramaters than actual variables. Registers allow you to capture and store data for repeated or later use.

Whilst in previous editions of this workshop a calculator was implemented to demonstrate registers, due to the lack of buttons we will use registers to allow us to gracefully display decimal values on the seven segment display. Whilst this isn't the best demonstration of register use, it should hopefully give you an idea of how they can be utilised.

- Open **wtfpga.v** in your text editor.
- Un-comment the instantiation of **displaySelect**..
- We will use **displaySelect** to take the binary value of the DIP switches **sw**, along with the state of the **switch** on the left of the seven segment display, to output either decimal or hexadecimal values through **nibbleMS** and **nibbleLS** to display on the seven segment display.
- Instead of connecting the switches directly to the decoders, we need to route them through the **displaySelect** module.
- In **wtfpga.v** define two **wires**, **nibbleMS** and **nibbleLS** of width **[3:0]** to act as intermediary signals.
- In the **nibbleDecode** modules, replace the **sw[3:0]** and **sw[7:4]** wires with **nibbleLS** and **nibbleMS**.

Now we have an instantiation for **displaySelect**, we need to add some logic for it to be useful.

- Open **displaySelect.v** in your text editor, and comment in all of the code.
- Find the **always @** block.
  - **always @** is the header for a synchronous block of code.
  - 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.
- Lets put **posedge clk** in the sensitivity list. This means each positive edge - every time the **clk** goes from low to high - we will execute this block: **always @(posedge clk)**
- Now, we need to choose if we will display hexadecimal or decimal values depending on the state of **switch**. Add **switch** into the brackets of the if - else block.
- When in hex mode, all we need to do is route the DIP switch (**sw**) wires out of the module and back to the decoder as before. This can be achieved by adding **nibbleMS <= sw[7:4];**, along with the corresponding line for the least significant nibble in the if block. Note that assignments are different inside the always block - we are setting a register, not assigning a wire anymore!

  - You may be wondering what that strange <= sign is, and why is isn't a = like you typically see in most programming languages?
  - <= is a **non blocking** assignment, and means that in an always block, every line will be executed in parallel.
  - On the other hand, = is a **blocking** assignment, and ensures that the following line of code will only be executed after it's previous line has been executed.
  - We will cover this in more detail in next month's meetup.

- Build this design with **make clean** then **make prog** and ensure it works so far. The display should function as normal when the switch is in the **1** position, but should be static when in the **0** position.

To implement the decimal decoding, we need to determine what digit needs to be shown on each display. In software, a straightforward implementation would involve taking the number modulo 10 to find the digit in the ones position, and then subtracting that from the total number to find the value in the tens column. However, division and modulo operations are computationally expensive, and for reasons we will not cover over should not be implemented on an FPGA though using the / and % operators.

Instead, we will use if else statements to determine the number in the tens column, and then subtract this from the total value to find the number to display in the ones position.

- Uncomment the lines below "determine value to display in most significant display". It compares the number from the DIP switch to 90, 80, etc, and sets **nibbleMS** which is then displayed in the tens position. Complete the rest of the statements.

- Complete the calculation of **nibbleLS** through inserting the correct values into the incomplete calculation. Hint: implement the sentence above these dot points.

- **make clean** then **make prog** to upload the updated gateware to the FPGA. Move the slide switch up into the **0** position and confirm that it decodes to decimal correctly. What happens when you count past 99? How can we solve this?

One method of resolving this would to not allow the displayed number to go above 99. Another would be to only show the two lowest digits on the display. I'll show you how to implement the former.

- First, below the module definition, instantiate a register of width 8 called dispNum which will hold the value displayed. You can do this, and set it's initial state to 0 with:
  reg [7:0] dispNum = 0;

- Now, directly between the **end else begin** and **if (dispNum $\geq$ 7'd90) begin** lines, we need to add an if statement to do the following:
  if (sw $\leq$ 99) begin
  set **dispNum** equal to **sw**
  end else begin
  set **dispNum** to **99**
  end

- Now change the **sw** variable to **dispNum** for all occurrences below this line.

- Run your code and confirm that it limits the display to 99. Now, by adding an additional if statement and some more logic, configure the display to only show the lowest two digits on the display when it runs out of space.

9 Exploring More

You've now completed the seven segment decoder, which uses most of the core concepts used to design nearly all silicon devices in use today. If time permits, there are a few additional things you can explore or try with this board.

- Examine the **feather.pcf** file. This contains all of the mappings of the FPGA's pins to the names you use in your code.
- 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.
- Modify **dispDecode.v** along with **wtfpga.v** to allow more than just hexadecimal numbers to be displayed.
- Examine the **displayMux.v** file. What does it do? How does it seem to work? Can you dim the display?
- Turn the display into a countdown timer, wherein you set a number on the display, and then move the slide switch to begin the countdown. For even more of a challenge, once the timer has reached 0, flash the display.
- Have a chat to Josh about the design and assembly of the iCE40-feather and how FPGA implementation differs from microcontrollers. For those interested, the design files are available at `github.com/joshajohnson/iCE40-feather`, and additionally a PDF schematic is available in the **WTFpga/docs** folder.

10 Thank You and Further Resources

I hoped you enjoyed this rather short dive into FPGA development. What's next? Over the next one or two (not sure yet!) meetups we will dive further into the world of FPGAs, and cover some of the topics we skipped, along with some of the questions I'm sure you have.

If you have any comments, questions, or feedback, please either have a chat in person of get in contact through one of the following channels:
Email: `josh@joshajohnson.com`
Twitter: `@_joshajohnson`
BSidesCBR Slack: `josh`

If you are looking for something to keep you entertained until next month, below are some resources which may be useful.

Hamsterworks has a wiki full of many things to do with FPGAs including 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`

nandland has a bunch of FPGA related content, along with a YouTube channel. `https://www.nandland.com/`

The repo for the iCE40-feather contains more documentation regarding the board, along with a few example projects. I haven't had much time to design projects for it, so if you build something using the board please open a pull request with your design! `https://github.com/joshajohnson/iCE40-feather`

Thanks once again for coming along, and I hope to see everyone here next month for even more FPGA fun!