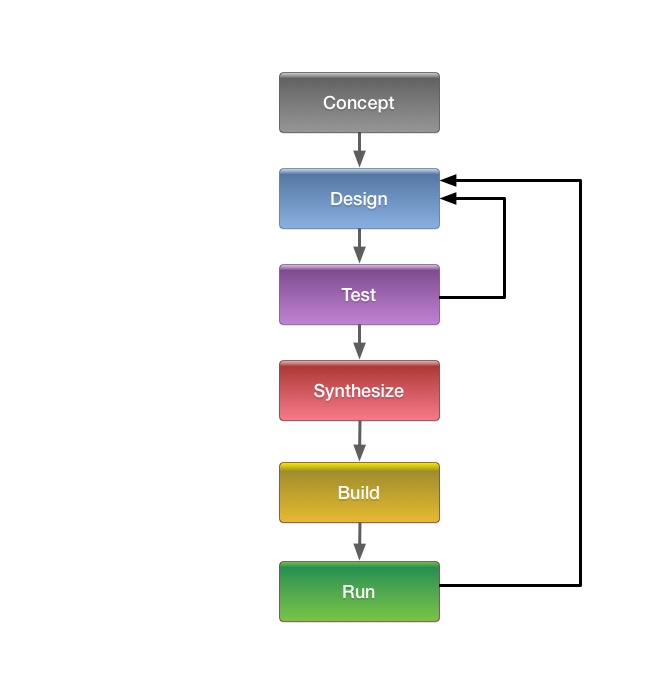
Step 2

Fair warning. You are about to enter a battle zone. This is not easy! It is fraught with peril and can be frustrating. Things worth doing are rarely easy. Some boards are easier to work with than others. Step 2 is where you sort out your particular development environment and document the steps required for someone else to duplicate it.

At all stages, we are going to share our work and help each other out. Radio functions are not mysteries. They are well-studied and can be mastered. Persistence reveals results!

# Learn FPGA Design Flow with a Frequency Divider Circuit

## Adapted from Chapter 2 of “Make: FPGAs”



Above is a diagram of how things like FPGA design almost always happen.

We begin with a **concept**.

We **design** our implementation.

We **test** the implementation, use what we learn to adjust the design, and then test the result. We may do this many times!

Once we’re satisfied with the result, we **synthesize**, **build**, and **run**.

When we run the completed design, we **test it again**.

It is highly likely that we may find that we need to make additional adjustments to the design, and we have to go back to the **design-and-test** loop. This is normal!

## Hardware Hacking Hello World

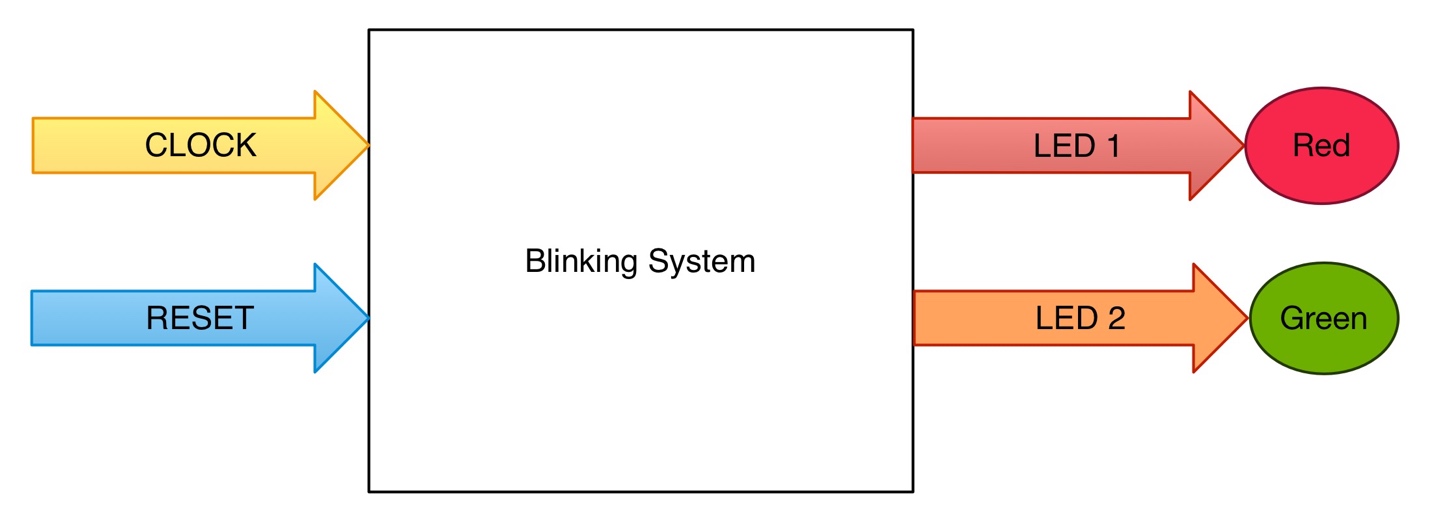
You may already be familiar with the Hello World concept, where the phrase “Hello World” is printed out on a screen or readout. In the software world, “Hello World” is often the first real test that the system can be programmed and is working.

On the hardware side, when you want to test that the system can be programmed and is working, the iconic “Hello World” is blinking an LED.

## Extending Hello World

For this step, we’re going to extend the basic “Hello World” of blinking an LED. We are going to combine blinking an LED with dividing down a fast signal to get a slower one. We will also have a signal that overrides the basic behavior.

Here’s a block diagram of what we’re doing.



Our inputs are system clock (CLOCK) and reset (RESET). When documenting work like this in drawings, inputs generally come in from the left and flow to the right. Our outputs are signals that blink two LEDs (LED 1 and LED 2). Outputs also flow from left to right. Blocks are used to abstract functions or collections of functions. High level block diagrams are the simplest and most abstracted form of documentation. Each block can be replaced with a more detailed, or lower level, block diagram.

We are going to make the LEDs blink at different rates. The output blinking rates we are going to produce with the Blinking System are much lower than the system clock. If we used the system clock directly to blink LEDs, the LEDs would blink at a rate much faster than what we could physically see. We have to divide down the system clock rate to get LED blink rates that can be seen with human eyes. Being able to control the LED behavior with a reset signal is an important concept. Signals from user interface and sensors need to be successfully incorporated into the system design all along the process. We’re beginning with a very simple reset signal, but this control signal concept will be extended to numerous signals for a working radio.

## Select Your FPGA Board

In order to do this step with physical hardware, **you will have to choose some sort of evaluation board or development module or testbed. You need an FPGA that can be programmed.**

For Phase 4 Ground, we are going to be using a variety of SDRs, and even make our own!   
  
Do you have an SDR with an FPGA that Vivado can talk to? Great! If you need one, then this is your chance to go get one!

*You can tell whoever you need to that Michelle said it was ok!*

Vivado talks to mid-scale and up Xilinx parts like Ultrascale, Virtex-7, Kintex-7, Artix-7, and Zynq-7000 series. If you have a board with any of these parts, then Vivado is what you need. It’s what you installed in Step 1.

If you want to use smaller Xilinx parts, like the Spartan-3, Spartan-6, Virtex-4, Virtex-5, and Virtex-6 families, then you will need the (discontinued) Xilinx ISE. Xilinx ISE is in “sustaining” mode, which means no new versions will be released, but you can still install it and use it.

We strongly recommend Vivado. It’s the current version of the toolchain from Xilinx. The chips supported by Vivado are what we’re going to be dealing with.

Different boards may need different versions of Vivado. That’s ok. Install whatever is called out for your board. Don’t fight it, just install it.

For example, let’s look at the Red Pitaya.

http://pavel-demin.github.io/red-pitaya-notes/led-blinker/

These are a set of notes to get the Red Pitaya cooperating with Vivado, and also blinking an LED. Do you have a Red Pitaya? Then the link above is a great start!

If you’re using something like a USRP x310, then according to the Ettus website, Xilinx Vivado 2015.2 Design Suite is what you’ll need. Don’t bang your head against a wall. If something blows up, back off and double-check.

Getting things set up for development can be hard. Tribal lore, unclear directions, things that change out from under you – all of this and more is considered to be part of the embedded development landscape.

This is not an excuse. Difficult or badly designed environments should not be normal or put up with without complaint. However, dealing with the innards of an FPGA is not the same as firing up a word processor and printing off a document. With great power comes great responsibility and almost always a steep learning curve. A good attitude (and sympathetic co-conspirators) is irreplaceable!

## Get an LED Blinking

This section will be updated as people document their recipes.

## What Did We Accomplish?

We chose a board.

We got it working with Vivado.

We blinked an LED.

We implemented an LED **Blinking System**.

* We learned how to divide down the system clock to get useful human-rate signals.
* We added a control signal.

## Next Step: Learn about Concurrency with a Digital Clock0

# Blink two Leds

by Carl Wall, VE3APY

The iCEstick is an iCE40 evaluation dongle by Lattice semiconductor. It is a USB stick which has a FTDI FT2232HL usb to serial chip, a 32Mbit SPI flash memory and a iCE40HX1K FPGA chip with some leds and a PMOD connector. To program iCEstick the open source IceStorm project software will be used. Since the IceStorm software only uses verilog that is what will be use for this example. The link to IceStorm build information is at

<http://www.clifford.at/icestorm/>

The manual for the iCEstick can be downloaded near bottom of web page.

<http://www.latticesemi.com/icestick>

Two of the more common FPGA Hardware Description Languages are VHDL and Verilog. You can have lots of flame wars over which is best. The bottom line if you are doing a lot of FPGA programming, the company that is paying you will normally decided which to use. Here is a little video which gives a quick overview.

<https://www.youtube.com/watch?v=frBnuKeshoM>

I heard one comment that VHDL tries to prevent you from shooting yourself in the foot, but Verilog will let you do it because you know what you are doing. Since I have a book in one hand and coding with the other, I have no opinions yet. ;-)

The second request from Michelle was to blink two leds and and also have a reset to the counter for the leds. This will require that we count down the crystal frequency for the leds. Since google is our friend we will do a google search to find some examples. The two that I used are

<http://badprog.com/electronics-verilog-blinking-a-led>

<http://simplefpga.blogspot.ca/2012/06/code-to-make-led-blink.html>

I also used this code example for an example of a shell script file for the building of a project for the iCEstick using the IceStorm software.

<https://github.com/wd5gnr/icestick>

Two files will be needed, the verilog.v code file and an icestick.pcf file which is what pins are connected to the outside world and what we would like to call them.

Lets start with the verilog code.

**blink2.v**

/\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* blink2.v \*/

/\* Nov 23, 2016 \*/

/\* VE3APY \*/

/\* Carl Wall \*/

/\* \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* module \*/

module blinking\_two\_leds (

input clock\_12, // 12 MHz clock on board

input PMOD1, // J2 pin 1

output LED1, // Red led

output LED5 // Green led

);

/\* reg \*/

reg [32:0] counter;

/\* assign \*/

assign LED1 = counter[24]; // Red led

assign LED5 = counter[25]; // Green led will blink half rate of red one.

assign reset = PMOD1;

/\* always \*/

always @ (posedge clock\_12 or posedge reset) begin

if (reset)

counter <= 0;

else

counter <= counter + 1;

end

endmodule

It is always good form to start with a header block which gives the name of the program, the date and who to call when it does not work. These days the type of license should also be in there, for anyone who might want to reuse your code.

The next thing in this program is to tell Verilog what names will be attached to inputs and outputs. In our case we have two inputs, the 24 Mhz clock and a signal from a switch to do a reset. We have two outputs LED1 which is a red LED and LED5 which is a green LED.

The next block of code we let Verilog know we will be using a counter which is 33 bits long by use of the **reg [32:0]**. Remember that we are counting from zero so there is 33 bits not 32. We then put what we want to call this counter, OK I was not original with the name. In a larger program it is good to give some thought to your names.

The next part is being used to change the names of some of the signals, I have assigned two bits from the counter chain to LED1 and LED5, and assigned **PMOD1** signal to the label **reset** which will allow me to use reset later which is a name people will know.

The **always** is were the meat of how things will work is described. On the positive level of the reset signal the counter will be loaded with zero. Otherwise the counter will be incremented by one on the positive edge of the clock\_12 pulse.

So how did I decided where to tap the signals to drive the LEDS. Since I am math challenged I looked at the examples and picked a tap which I thought would be in the ball park. Compiled the code, and downloaded it. The flashing was pretty slow, so I reduced the tap number on the counter chain, until it looked right. This approach is OK with a small program like this, but a larger program which might have a much longer compile time this approach could be very slow.

Then I checked that the reset switch did what I thought it would do and was finished. I also noticed that some of the leds which I did not used were on very dim, so the floating pins I did not used was causing that. So if I was sending this off to production I would add code to make those unused led pins low.

**blink2.pcf**

# blink2.pcf

#

#Clock

set\_io clk\_12 21 # 12 MHz clock

# LEDs

set\_io LED1 99 # red

set\_io LED5 95 # green

#PMOD

# Note: pin 5 and 11 are ground, pins 6 and 12 are VCC

set\_io PMOD1 78 #PI01\_02 Reset

blink2.pcf is the constraints file. When the compiler/router is figuring out how to build your circuit in the FPGA it will use this file to limit where it can put the input/output connections from the FPGA design to the pins on the FPGA chip. The file is a simple list of the names you have called things and the chip pin you are expecting them on. Some versions of software do not like you naming things and not using them in your FPGA, so some constraints files can have a lot of unused names commented out. It is nice to have a common constraints file for the board that you are using, saves some trouble shooting later. A more complicated FPGA chip would also have more options in the constraints file for the various pins.

We now have a Verilog file and a constraints file, so lets build the project. We are going to start by doing it on the command line and then using a shell script to do the same thing. I am going to assume that you followed the IceStorm build instructions and have the tool chain installed.

We will open a terminal window and **cd** to the folder where we have the Verilog and constraints file.

yosys -p "synth\_ice40 -blif blink2.blif" blink2.v

This command will convert the verilog file to a RTL file in the blif format. Check the web page for yosys for more details. This is black magic to me.

arachne-pnr -d 1k -p blink2.pcf blink2.blif -o blink2.asc

This command takes the blif file and the constraints file and figures out how to build the design in the FPGA chip. It produces an ascii file of the result. Or in the words of the developer. Arachne-pnr implements the place and route step of the hardware compilation process for FPGAs. It accepts as input a technology-mapped netlist in BLIF format.

icepack blink2.asc blink2.bin

Takes the ascii file and converts it into a binary file which will be loaded into the SPI flash on the iCEstick.

iceprog blink2.bin

This command takes the binary file and downloads it to the Flash on the iCEstick. Since this is over USB some trouble shooting is sometimes required. If the command fails but works if you do use, sudo iceprog blink2.bin then you have a permission problem when the iCEstick got mounted. Put the following information

ACTION=="add", ATTR{idVendor}=="0403", ATTR{idProduct}=="6010", MODE:="666"

into the file named 53-lattice-ftdi.rules

and put that into /etc/udev/rules.d/

Now that we can built the project at the command line, when you are building it a lot this can lead to a lot of typing so here is a shell script which is put in the same folder as the verilog and constraints files to automate the procedure.

**build.sh**

#!/bin/bash

# Build with open source tools

if [ -z "$1" ]

then

echo Usage: build.sh main\_name [other .v files]

echo Example: ./build.sh demo library.v

exit 1

fi

set -e # exit if any tool errors

MAIN=$1

shift

echo Using yosys to synthesize design

yosys -p "synth\_ice40 -blif $MAIN.blif" $MAIN.v $@

echo Place and route with arachne-pnr

arachne-pnr -d 1k -p blink2.pcf $MAIN.blif -o $MAIN.txt

echo Converting ASCII output to bitstream

icepack $MAIN.txt $MAIN.bin

echo Sending bitstream to device

iceprog $MAIN.bin

./built.sh blink2 is the command we would now used to build the project

I would like to thank Clifford Wolf for developing IceStorm and the other tools which lead to the Open Source development tools for the Ice40 FPGAs.

**Appendix**

Good overview of the steps of using the IceStorm software.

<https://www.youtube.com/watch?v=1CNVsxoLI60>

Talk by Clifford Wolf at the 32C3 about the IceStorm project

<https://www.youtube.com/watch?v=9rYiGDDUIzg>

Links to the various software parts of IceStorm

<http://www.clifford.at/icestorm/>

<http://www.clifford.at/yosys/>

<https://github.com/cseed/arachne-pnr>

Books which I thumbed through while trying to code blink2.v

Verilog by Example A concise introduction for fpga design

by Blaine C. Readler

Programming FPGAs Getting started with Verilog

by Simon Monk

Make: FPGA

by David Romano