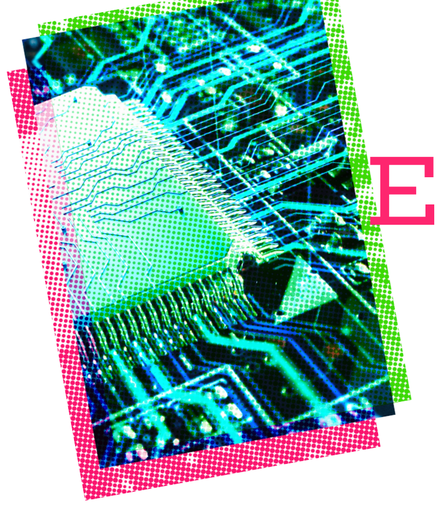

Embedded
FreeBSD

Fabric — Baby Steps

BY CHRISTOPHER R. BOWMAN

In previous columns, we took a basic look at the Zynq chip and mentioned it's fabric. Since then, we haven't really mentioned it much. But, in the last column, we got bhyve running a CentOS image, and so now it's time to look at our first fabric circuit. This column will be all circuit and no FreeBSD, but with the next couple columns, we will start to look at systems that combine the two.

In addition to the Zynq-7000 SoC Technical Reference Manual which documents the Zynq chip that forms the bulk of the Arty Z7-20 functionality, Digilent's Arty Z7 Reference Manual contains a wealth of valuable information on how the Zynq chip is wired and connected on the board. If we look at section 12, we can see that the board has 4 LEDs that attach directly to the ZYNQ chip (R14, P14, N16, M14). If we can set these pins to a logic 1 or high-voltage level, then the pins will source current that will flow through the LEDs and then through the current-limiting resistor and to ground causing the LEDs to turn on.

Since we're just getting started, let's try to build the smallest and simplest circuit to turn on these pins. The simplest circuit would be to statically tie these pins high in the fabric. Great, how do we do this? We're going to have to introduce Verilog to do this.

Over the last decade, two languages have really come to dominate chip design: VHDL and Verilog.

Much like in the early days of programming, where programs were written in machine code and then assembler and eventually in high-level languages like C, ICs were originally hand drawn or laid out with mylar tape. As circuits became larger and CAD programs were developed, circuits were designed using programs — EDA (Electronic Design Automation). Schematic capture programs allowed one to design circuits by connecting devices — originally transistors, and later gates — graphically using a GUI. Eventually, languages were created which allowed the description of circuits textually. I used an early language called SFL for two of my first 3 chips back in the 1.2-micron days. But over the last decade, two languages have really come to dominate chip design: VHDL and Verilog. These languages are much like Ada and C in many conceptual ways. Ada and VHDL are very verbose and have strong type checking. They were both designated as the preferred language for US DOD work, and while both are still present, they're about as popular in their respective fields today. Verilog on the other hand, like C, has become the most popular language in circuit design. It's not as strongly typed as VHDL is, just as C isn't

as strongly typed as Ada. In any case, these days, if you want to do digital (not analog) circuit design of anything more than a handful of gates, you're using VHDL or Verilog, with most new designs use Verilog.

Circuit designs in Verilog get turned into circuits using a synthesis tool just as we use compilers to turn C into a running program. There is a bit more to it. For instance, in addition to turning the Verilog into a set of connected gates implementing the design, when designing a computer chip, you also need to run a tool to place the gates and route the interconnect wires. Fortunately for us, all that functionality is present in the Xilinx/AMD tool called Vivado. Not only does Xilinx/AMD offer Vivado, but it's a free download, and you can get a license to unlock quite a bit of the functionality of the Zynq chips free of charge. If you want to do anything more than write programs for your Arty Z7 board, you'll need to download and install this software on a Linux or windows machine. We covered setting up a bhyve instance running Linux in the last column so that you can run Vivado on a Linux VM under bhyve on your FreeBSD machine.

Circuit designs in Verilog get turned into circuits using a synthesis tool just as we use compilers to turn C into a running program.

Now that we're oriented, let's turn towards our first and simplest circuit to turn on the LEDs on our board. First, we will need a Verilog description of our circuit. I can't possibly teach you Verilog in one small column, so I will just present the design and try to describe what it's doing.

```
module top(
    output [3:0]led
);

wire [3:0]led;

assign led = {1'b1, 1'b0, 1'b1, 1'b0};

endmodule
```

First, designs are captured in modules and this one has a 4-bit bus of wires coming out of it called **led**. We are driving this bus with a set of 4 concatenated constants. Those constants are 1-bit signals, half of which are logic "hi" or 1, and half are logic "low" or 0. I picked an alternating set of values so that I can tell which way the LEDs are wired: msb to lsb or lsb to msb. This way, I don't have to work it out from the markings on the board and the documentation.

The next thing we need to do is provide a constraints file. Constraints files have two major functions: they convey timing information, and in the FPGA world, they also convey some placement information. For our first experiment, we need to tell the tool what pins on the chip are connected to the wires of the **led** bus, and we also need to tell the tool how to setup the IO pins we wish to use. Digilent, quite helpfully, has a master XDC file that has this information for this board and many others in a github repo. Unfortunately, they don't include a copyright notice in the file or readme and so I can't include it in my project. The few

relevant lines are provided here, and if you use the make file included in my static_leds repo for this article, it will download the file from github and uncomment the relevant lines. You'll need GNU make and wget installed on your system.

```
set_property -dict { PACKAGE_PIN R14  IOSTANDARD LVCMOS33 } \
     [get_ports { led[0] }]; #IO_L6N_T0_VREF_34 Sch=LED0
set_property -dict { PACKAGE_PIN P14  IOSTANDARD LVCMOS33 } \
     [get_ports { led[1] }]; #IO_L6P_T0_34 Sch=LED1
set_property -dict { PACKAGE_PIN N16  IOSTANDARD LVCMOS33 } \
     [get_ports { led[2] }]; #IO_L21N_T3_DQS_AD14N_35 Sch=LED2
set_property -dict { PACKAGE_PIN M14  IOSTANDARD LVCMOS33 } \
     [get_ports { led[3] }]; #IO_L23P_T3_35 Sch=LED3
```

Finally, we need to run Vivado to convert this into a BIT file which is a representation of the circuit we've designed. Sadly, it's not as easy as just passing these two files to Vivado. Circuit design is a complex process with many options. Vivado, like many EDA (Electronic Design Automation) tools, is a TCL-based tool that needs a script to operate. In my repo, I've created the simplest **GNUMakefile** I can to automatically download the XDC file, patch it, and run Vivado with a TCL script. I encourage you to look through it, but if you just want to get on with it, update the path variables and then a simple **make** under Linux should do everything and leave you with a file **implementation/static.bit** which is the circuit file we need to load into the Zynq chip.

So, I have my circuit file, now what? U-boot contains an FPGA bitstream loader, so we will start with that. Copy the **FPGA.bit** file to the MSDOS partition of your SD card, insert the card into the board, and push the reset button. At the U-boot prompt, interrupt the boot process and run the following commands:

```
Zynq> fatload mmc 0 0x4000000 static.bit
4045663 bytes read in 249 ms (15.5 MiB/s)
Zynq> fpga loadb 0 0x4000000 4045663
```

The first command loads the **static.bit** file from the FAT partition on the SD card into memory. The second tells U-boot to program the FPGA with the file contents now in memory at **0x4000000**. At this point, you should see two of the four LEDs on the board turn bright red. Congratulations! You've built and loaded your first FPGA design!

We could stop here, but let's do two more things before we call it quits. Let's make our LEDs blink instead of just turning on, and let's see how we can load the FPGA from under FreeBSD. That will start to lay the ground work for cool things.

To make the LEDs blink, we need to change our circuit by changing the Verilog. There is a new repo for this new circuit, but I'll summarize the changes here. First our new Verilog:

```
module top(
    input clk,
    output [3:0] led
);

localparam cycles_per_second = 125000000;

reg [3:0]leds;
reg [31:0]counter;
```

```
always @ (posedge clk)
begin
  if (counter == 0) begin
    leds <= leds + 1;
    counter <= cycles_per_second;
  end else counter <= counter - 1;
end

assign led = leds;

endmodule
```

We've now added a clock input that will drive a 31-bit counter, and we've made that and a 4-bit counter. The 31-bit counter loads with the value 123,000,000 and counts down to zero. When it hits 0, the 4-bit **leds** counter increments. We choose 125,000,000 because, looking at the Arty reference manual section 11, we see the ethernet phy supplies a 125MHz clock on pin H16.

Next, we need to tell Vivado about the clock on pin H16:

```
set_property -dict { PACKAGE_PIN H16 \
     IOSTANDARD LVCMOS33 } \
     [get_ports { clk }]; #IO_L13P_T2_MRCC_35 Sch=SYSCLK
create_clock -add -name sys_clk_pin -period 8.00 \
     -waveform {0 4} [get_ports { clk }];#set
```

Again, a simple make should build a file **implementation/blinky.bit** which can be transferred to the SD card and loaded into the FPGA as above.

Now you should the see the LEDs blink counting in binary.

Ok, before we wrap it up for this column, let's talk about one last thing. Let's see how we can program the FGPA from inside FreeBSD. Turns out this is simple. There is a **/dev/devcfg** device that was originally intended for you to simply **cat** a bit file to this device, but I don't think the work on it was quite completed. There is a simple C program **xbin2bit** which has its own **git repo**. if you have root permissions (by default **/dev/devcfg** is owned by root) you can simply run and pass it your bit file:

```
# xbin2bit blinky.bit
```

Did you run that and see your FreeBSD system halt, but the LEDs keep blinking? Yep, turns out our Verilog designs need to be a little more sophisticated so that the processor doesn't stop. We'll explore this more in the next column.

If you've got questions, comments, feedback, or flames on any of this I'd love to hear from you. You can contact me at articles@ChrisBowman.com.

CHRISTOPHER R. BOWMAN first used BSD back in 1989 on a VAX 11/785 while working 2 floors below ground level at the Johns Hopkins University Applied Physics Laboratory. He later used FreeBSD in the mid 90's to design his first 2 Micron CMOS chip at the University of Maryland. He's been a FreeBSD user ever since and is interested in hardware design and the software that drives it. He has worked in the semiconductor design automation industry for the last 20 years.