First Steps with Migen

Nina Engelhardt

December 7, 2014

Helpful things to be familiar with

- Boolean Logic
- Binary representation of numbers
- Python syntax (including classes)

Setup

Make sure you can ssh into the lab machine. We have installed the synthesis tools and the tools for programming the FPGA on this machine, so you will need to copy your code there for build.

If you are using windows and can't figure out how to copy files via ssh, a USB key will also work for transferring files.

Choose an editor with syntax highlighting for python.

1 Your first design: a XOR gate

For the first step, we will create an extremely simple design. It consists of a single XOR.

In your favourite editor, create a new file. Because we will all be compiling on the same machine, name it `first_combinatorial_<your_name_here>.py`.

Import the basic migen tools:

```
from migen.fhdl.std import *
```

The basic unit of hardware design is a module. In migen, `Module` is a class that you can subclass and instantiate. Let's create a new class describing our type of module that performs the XOR operation. Our XOR will need operands. Add two signals to serve as inputs: `a` and `b`. We'll also need to send the output value somewhere: let's call it `o`.

```
class MyXOR(Module):
    def __init__(self, a, b, o):
```

All three arguments will be of type `Signal`. A `Signal` is 1 bit wide by default. `Signal(n)` will create a `Signal` n bits wide.

Now we can proceed to add the actual logic. For now, we only have combinatorial logic. All combinatorial statements should be added to the pseudo-attribute `comb` of a `Module`:

```
        self.comb += o.eq(a ^ b)
```

That's it! We've defined a module that performs the XOR operation. Save your file.

But wait... it's not doing anything!

Defining things is all good and well, but we're not doing mathematics here, this is engineering. We want to actually *make* something. So we need to add instructions to tell migen to create a new instance of this MyXOR module.

First we need to tell it which board we want to use. At the top of the file, add the following imports:

```
from mibuild.generic_platform import *
from mibuild.platforms import papilio_pro
```

Then scroll down again to the bottom. We will be working with the Papilio Pro today. Instantiate the `papilio_pro` platform:

```
if __name__ == "__main__":
    platform = papilio_pro.Platform()
```

Sadly, the board we are using does not have buttons, so we will use some of the free GPIO pins. To add the buttons, we need the following lines:

```
    _button_io = [
        ("btn0", 0, Pins("C:2"), IOStandard("LVTTL"),
   Misc("PULLDOWN")),
        ("btn1", 0, Pins("C:10"), IOStandard("LVTTL"),
   Misc("PULLDOWN"))
    ]
    platform.add_extension(_button_io)
```

We want to use buttons as inputs and show the output value on the LED. Now that it knows about all of these, ask the platform nicely to give them to you:

```
    btn0 = platform.request("btn0")
    btn1 = platform.request("btn1")
    led = platform.request("user_led")
```

You will receive signals connected to the right place on the board. Now, instantiate your XOR module and give it these IO signals:

```
    myxor = MyXOR(btn0, btn1, led)
```

Finally, tell it to build the design. Let's also give it a unique name so that you will be able to tell the produced files apart from the other participants':

```
    platform.build_cmdline(myxor, build_name="
   first_combinatorial_<your_name_here>")
```

Now save the file again. Then execute it on the lab computer:

```
python3 first_combinatorial_<your_name_here>.py
```

If all goes right, you should obtain a folder named `build` containing a file called `first_combinatorial_<your_name_here>.bit`. To program the board, please come up to the lab computer and we will walk you through testing your first design live!

2 Second design: synchronous statements

The second thing we need to be able to do for the FM transmitter is use synchronous statements. The board is equipped with a clock, a signal that switches from 0 to 1 and back to 0 at 32 MHz. A synchronous assignment, instead of being executed immediately, will only take effect at the rising edge of the clock, i.e. when it goes from 0 to 1.

Let's use this to make our LED blink.

Create a second file, name it `second_synchronous_<your_name_here>.py`, and copy the include statements from before:

```
from migen.fhdl.std import *
from mibuild.generic_platform import *
from mibuild.platforms import papilio_pro
```

Define a new Module to contain your design:

```
class LEDBlinker(Module):
```

This time we only need one signal connection to the outside, namely the LED:

```
    def __init__(self, led):
```

The easiest idea would be to simply switch the value of the led signal every time the clock changes. But that would make our LED blink at 32MHz, which is way too fast to see! Instead, we want it to blink about 1 times a second, i.e. 1Hz. So we need to count to ≈ 32 million and only then change the value.

The nearest power of two is $33554432 = 2^{25}$. So let's define a 25 bit Signal:

```
        count = Signal(25)
```

Then, increment `count` by 1 every cycle:

```
        self.sync += count.eq(count+1)
```

There is no overflow handling, so `count` will simply wrap around to 0 when it

reaches 2^{25}.

We want the light to be on about half of the time it takes to count there and off the other half. We could compare the value of `count` to 2^{24} and turn the light on if it is greater, off if it is smaller. But we can spare ourselves some work by looking at one property of the binary representation of `count`: all numbers between 0 and 2^{24}-1 will have the most significant bit be 0, whereas the MSB will be 1 for the numbers $2^{24} - 2^{25}$-1. So we can simply display the MSB on the LED:

```
        self.comb += led.eq(count[24])
```

and voila!

As before, insert code to instantiate the platform, request the LED, instantiate your LEDBlinker, and build the whole design:

```
if __name__ == "__main__":
    platform = papilio_pro.Platform()
    led = platform.request("user_led")
    blinker = LEDBlinker(led)
    platform.build_cmdline(blinker, build_name="
   second_synchronous_<your_name_here>")
```

Run this script on the lab computer and try it out on the board.