An introduction to Migen

Version: April 9 2013

Sébastien Bourdeauducq

This tutorial gives a first introduction to FPGA design using Migen. It assumes some knowledge of Unix commands, Python programming language and logic design.

1 Software setup

1.1 Third-party tools

This tutorial requires a Linux machine with Python 3, Git and Xilinx ISE. Note that Migen cannot be used with Python 2, but most Linux distributions allow you to easily install both Python 2 and Python 3 on the same machine.

1.2 Migen and Mibuild

We simply obtain the sources from the Git repositories and set the `PYTHONPATH` environment variable so that Python searches those directories when importing modules. Alternatively, Migen and Mibuild can be permanently installed on your system by running their respective `setuptools` script (`python3 setup.py install`)

```
$ git clone git://github.com/milkymist/migen.git
$ git clone git://github.com/milkymist/mibuild.git
$ export PYTHONPATH=`pwd`/migen:`pwd`/mibuild
```

1.3 Simulator

This step can be skipped if you do not intend to use the simulator.

Migen relies on an external Verilog simulator to simulate your designs. It is known to work with Icarus Verilog.

Make sure that your installed version of Icarus Verilog is recent enough to include commit b85e7efca86757c4a752bbba5de2127fe9df0a13 (from April 2, 2012). The bug that this commit fixes makes the Migen simulator completely dysfunctional.

To communicate with Icarus Verilog, the Migen simulator uses a UNIX domain socket and a custom protocol which is handled by a VPI plug-in (written in C) on the Icarus side.

To build and install this plug-in, run the following commands from the `vpi` directory in the Migen source tree:

```
$ make [INCDIRS=-I/usr/...]
$ make install [INSTDIR=/usr/...]
```

The variable `INCDIRS` (default: empty) can be used to give a list of paths where to search for the include files. This is useful considering that different Linux distributions put the `vpi_user.h` file (shipped with Icarus Verilog) in various locations.

The variable `INSTDIR` (default: `/usr/lib/ivl`) specifies where the `migensim.vpi` file is to be installed.

A VCD file viewer such as GTKWave should also be installed.

2 First steps

A central component of Migen is the FHDL layer. It allows you to create and manipulate logic designs in Python and convert them to synthesizable Verilog.

Run a Python interpreter and import the Migen FHDL declarations:

```
$ python3
>>> from migen.fhdl.structure import *
```

The basic building block of a FHDL design is the `Signal` object. It serves the same purpose as `signal` in VHDL and `wire` or `reg` in Verilog.

We create two such signals, having a width of 1 bit each:

```
>>> a = Signal(1)
>>> b = Signal(1)
```

The width of 1 is the default, so one can also simply write Signal().

We would now like to represent Boolean equations between signals, for example the `OR` of these two signals we just created. Migen provides the `_Operator` object for this purpose, but since using it directly results in a very cluttered syntax, it also redefines the basic Python operations on signals so that such `_Operator` objects can be created in a much lighter way:

```
>>> a | b
<migen.fhdl.structure._Operator object at 0x965e14c>
```

Let's examine the contents of our newly-created object:

```
>>> tmp = a | b
>>> tmp
<migen.fhdl.structure._Operator object at 0x965e86c>
>>> tmp.op
'|'
>>> tmp.operands
[<Signal a at 0xb6f7ae2c>, <Signal b at 0x965e82c>]
```

As you can see, the object contains the information to represent our `OR` gate. `_Operator` objects can be of course combined to form expression trees (of arbitrary complexity):

```
>>> c = Signal()
>>> tmp = a | (b & c)
>>> tmp
<migen.fhdl.structure._Operator object at 0x965eeac>
>>> tmp.op
'|'
>>> tmp.operands
[<Signal a at 0xb6f7ae2c>,
  <migen.fhdl.structure._Operator object at 0x965e56c>]
>>> tmp.operands[1].op
'&'
>>> tmp.operands[1].operands
[<Signal b at 0x965e82c>, <Signal c at 0xb6f6c0ec>]
```

We now have a means of representing Boolean equations involving signals. We would now like to assign such expressions to other signals. FHDL provides the `_Assign` object for this purpose, as well as a technique to create it easily using the `eq` method of `Signal` objects:

```
>>> x = Signal()
>>> tmp = x.eq(a | b)
>>> tmp
<migen.fhdl.structure._Assign object at 0xa2371ec>
>>> tmp.l # left hand-side of assignment
<Signal x at 0xa23756c>
>>> tmp.r # right hand-side of assignment
<migen.fhdl.structure._Operator object at 0xa23790c>
```

In a typical FPGA design, an assignment can be triggered by two types of events:

1. whenever an input changes *(combinatorial assignment)*
2. at the edge of the clock signal *(synchronous assignment)*

Defining when an assignment takes place is done by using the in-place addition operation (+=) on special properties of a `Module` object. The `comb` special property makes the assignment combinatorial, while the `sync` property synchronizes it with the default (aka system) clock.

Migen supports designs with multiple clock domains, but they are beyond the scope of this tutorial.

We can now fully model a pure (combinatorial) `OR` gate between signals `a`, `b` and `x`:

```
>>> from migen.fhdl.module import Module
>>> my_or = Module()
>>> my_or.comb += x.eq(a | b)
```

Modules are convertible to Verilog. Note the `ios` option of the `convert` function, that specifies which signals (in our case, all of them) should be exported as inputs/outputs of the Verilog module. Without that option, signals would stay inside the Verilog module (try it).

```
>>> from migen.fhdl import verilog
>>> print(verilog.convert(my_or, ios={a, b, x}))
/* Machine-generated using Migen */
module top(
        input a,
        input b,
        output x
);


// synthesis translate off
reg dummy_s;
initial dummy_s <= 1'd0;
// synthesis translate on
assign x = (a | b);

endmodule
```

3 A LED blinker

3.1 Design

We are now ready for a slightly more complicated design. It consists of a decrementing counter, which, when it reaches 0, toggles a one-bit signal (which will blink a LED) and reloads from another signal (that controls the period of the toggling).

Create a file `ledblinker.py` containing the following:

```
from migen.fhdl.structure import *
from migen.fhdl.module import Module
from migen.fhdl import verilog

class Blinker(Module):
        def __init__(self, led, maxperiod):
                counter = Signal(max=maxperiod+1)
                period = Signal(max=maxperiod+1)
                self.comb += period.eq(maxperiod)
                self.sync += If(counter == 0,
                                led.eq(~led),
                                counter.eq(period)
                        ).Else(
                                counter.eq(counter - 1)
                        )

led = Signal()
my_blinker = Blinker(led, 30000000)
print(verilog.convert(my_blinker, ios={led}))
```

Take note of the following points:

- we are inheriting from the `Module` class. This is the preferred way of creating a component in Migen, rather than instantiating `Module` directly and externally using its special properties as we did before. In a more complex design, a module can integrate other modules by using the `submodules` special property.
- another way to create multi-bit signals is to use the `max` parameter of the constructor, which specifies the exclusive upper bound that the value of the signal can reach. There is a similar `min` parameter. Both `min` and/or `max` can be negative, in which case Migen will use two's complement arithemetic to represent negative values.
- we use the `If` object, which represents conditional statements (which have the same sense as in Verilog or VHDL). Another Python syntax trick is used here for `Else`, which is actually a method of the `If` object that modifies the latter when called and inserts the statement list for the "false" part of the conditional.

Run this script and examine the generated Verilog source.

As an exercise, you can add a control signal that toggles between a high and a slow blinking frequency. Conditional statements can also be used in combinatorial lists (as in Verilog or VHDL). When using += on a special Module property, the statement(s) you specify are added to the existing statements in that module.

3.2 Simulation

For the purposes of the simulation, set the period signal to a lower value, e.g. 5. Add the following to the script:

```
from migen.sim.generic import Simulator, TopLevel

...

sim = Simulator(my_blinker, TopLevel("ledblinker.vcd"))
sim.run(200)
```

You can remove the import of the `migen.fhdl.verilog` module and the printing of the Verilog source. Running the script now produces a VCD file which you can open with GTKWave:

```
$ gtkwave ledblinker.vcd
```

Notice that a clock and reset signal have been added automatically.

Since observing waveforms manually is a tedious and error-prone process, Migen lets you read and write simulated signals from Python. All the libraries and features that Python offers can be used, which enables you to create powerful test benches. In this tutorial, we will simply print out at which clock cycles the transitions on `led` occur.

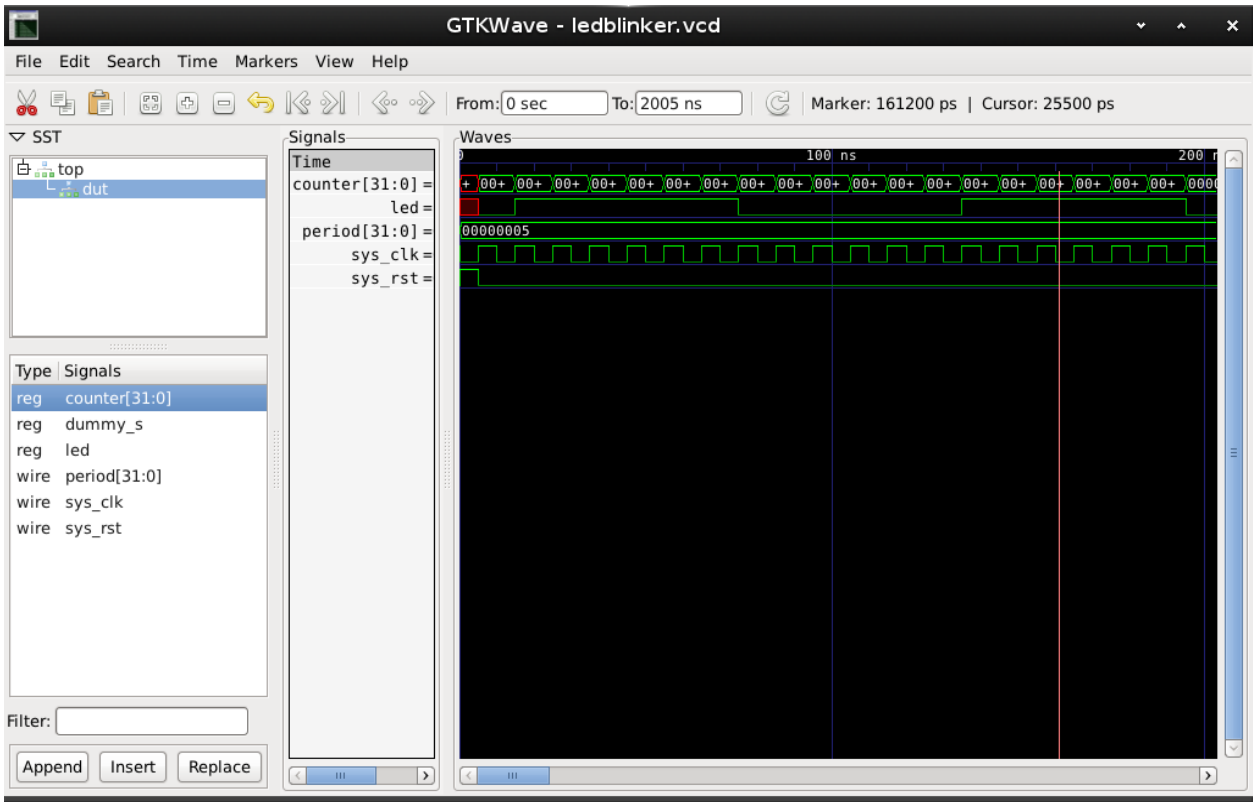

Fig. 1: LED blinker signals in GTKWave.

Add the following code to the script:

```
...
class Blinker(Module):
        def __init__(self, led, maxperiod):
                self.v_led_old = 0
                ...

        def do_simulation(self, s):
                v_led = s.rd(led) # read the current signal value
                if v_led != self.v_led_old:
                        print("{old}->{new} cycle={cycle}".format(
                                old=self.v_led_old, new=v_led,
                                cycle=s.cycle_counter))
                        self.v_led_old = v_led
```

We are using another `Module` feature, the `do_simulation` method. At each simulated clock cycle, the Migen simulator runs this function in each `Module`, passing them an object that lets them manipulate the values of the signals.

Run the simulator again. It should run the `do_simulation` method and produce the following output:

```
0->1 cycle=1
1->0 cycle=7
0->1 cycle=13
1->0 cycle=19
0->1 cycle=25
1->0 cycle=31
0->1 cycle=37
1->0 cycle=43
...
```

As an exercise, you can add code that verifies that the transitions are happening on the intended edges (i.e. make the test bench self-checking).

3.3 Hardware implementation

Revert any code changes you have done for simulation purposes and go back to the original design (you can still leave out the explicit conversion to Verilog).

Hardware implementation on supported boards is greatly simplified by using the Mibuild library. This tutorial assumes you have a Milkymist One, a Papilio Pro or a RHINO board. Import the corresponding module with one of the following lines:

```
from mibuild.platforms import m1 as board
from mibuild.platforms import papilio_pro as board
from mibuild.platforms import rhino as board
```

See the “platforms” folder in the Mibuild sources for a complete list of supported boards. You can also add your own and submit it for inclusion into Mibuild. Altera boards are also supported.

Next, create a platform object and obtain the `led` signal from Mibuild instead of creating it yourself:

```
plat = board.Platform()
led = plat.request("user_led")
my_blinker = Blinker(led, 30000000)
```

You are now ready to build the bitstream! Mibuild will add a default clock, which has a frequency of 50MHz on the Milkymist One, 32MHz on the Papilio Pro and 100MHz on the RHINO.

```
plat.build_cmdline(my_blinker)
```

Run the script with those modifications, and after a too long compilation process, you will obtain a `top.bit` bitstream file in a newly created folder `build`.

Mibuild assumes the Xilinx tools are installed in `/opt/Xilinx`. If they are not, run your script as follows: `python3 ledblinker.py --ise-path /path_to_ise`

Load the bitstream file using the appropriate programming tool for your board, and you should see the LED blinking. The Milkymist One and Papilio Pro boards can be programmed with UrJ-TAG:

```
$ jtag
jtag> cable milkymist # for Milkymist One
jtag> cable Flyswatter # for Papilio Pro
jtag> detect
jtag> pld load build/top.bit
```

4 To go further

This tutorial has given you an overview of the base component of Migen, FHDL. It enables developers to use the full power of Python to generate and verify the logic of their designs.

Migen comes with many other components and features that use FHDL to build system-on-chip interconnect, dataflow systems, configuration and status register banks, finite state machines, and more. Read the Migen manual available at `http://milkymist.org/3/migen.html` for a more complete description. A heavy user of Migen is the milkymist-ng system-on-chip, whose source code you can obtain from `http://milkymist.org/3/mng.html`.

Direct questions and feedback about Migen or this tutorial to the devel@lists.milkymist.org mailing list or the IRC channel #milkymist on Freenode.