MTest: Framework for writing unit tests in Magma

Rishab Mehra and Parimarjan Negi

March 24, 2017

1 Introduction

Unit testing has become one of the cornerstones of agile software development in recent years. It plays a crucial role in maintaining code bases and making incremental changes without breaking any existing functionality. But there are some subtle differences in unit testing of hardware code as compared to unit testing software. One of the main issues with unit testing on the FPGA is that we need to communicate the inputs and outputs to the Icestick. This raises other interesting questions - like the control that should be provided to the other user over setting various variables on the Icestick. Our goal was to design a prototype system that makes hardware unit testing appear as straightforward as writing a software unit test.

2 Design Goals

2.1 Simplicity

Throughout the design process, wherever possible, we wanted to keep the usage and implementation details as simple as possible. From the users perspective, this is essential, as people do not spend as much time writing tests - so it is crucial that the testing framework is very easy to use. From the point of view of implementation - we felt that there are many components of the design which could potentially be changed (for instance, the communication protocol) - so we tried to keep a simple design that would make it easy to change individual components of the framework. A few such points we focused on were:

- We wanted to have a simple, but easily extendable interface.
- A unit-test should be in one self contained file. All the details of uploading the circuit, and communication, should be hidden from the user.
- Should be easy to specify input-output values - in binary or hex, or just stream input-output combinations from python.

2.2 API and Abstraction

In software unit-testing frameworks, users typically provide the input arguments - and are able to assert some conditions on the output of the function. For hardware, this might not be enough control. A couple of different use cases are:

1. The user wants to send an array of 4 bits to an adder, eg. sum = add2(array1, array2). This is similar to the software testing model we described.

2. The user might want to wire just one bit, eg. dff.CE to some bit

3. The output the user wants to test might not be explicitly returned from the function - for example, testing the value of counter.O or mux.O

Of course it is possible to adjust any circuit such that each of these cases is reduced to just setting the input - output of a module in which the test circuit is defined. But it is clearly not ideal for us to expect the user to change his circuits in order to write test cases.

Another design we considered was to let the user provide us a dictionary of variable names to value pairs. This should work in principle - but we felt that it would require support for many different cases (for instance, (1) and (2) above - which have different ways of setting values).

In the end, we settled on an abstraction in which we provide the user with n 8-bit arrays as receivers, and an 8-bit array output that they control. We were limited in the size of the input/output arrays to 8 bits because of the UART protocol. This requires them to do additional work - ie. wire up the receivers and output as they want. But it also provides them more control. For instance, this abstraction could be used to test smaller components of a large circuit that you are writing.

We also believe, for testing magma circuits, it is a more natural way for them to be able to wire bits as they want directly. This abstraction also makes it easier from the perspective of the testing library - as now we only need to ensure the given values are correctly placed on the input bytes, and dont have to worry about different kind of wiring situations.

2.3 Testing Speed

Compiling and uploading circuits to icestick is quite slow even for relatively simple circuits - so it was crucial that we can run multiple tests (input-output combinations) for every circuit we upload. (See Implementation for the details).

Besides that, we have not focused on speed optimizations - it might be possible to run the tests much faster if we can control the clock / or enable bits on the Icestick (see Design Issues). For now, we have at least a 0.1 seconds sleep for every cycle - and possibly more - depending on how long the circuit takes to finish the computation.

3 Design Issues

3.1 Sequential Circuits

The design, as we have described so far, works quite well for combinatorial circuits - or those which do not depend on state. For instance, the state of counters, is constantly changing - so ideally, we need to be able to disable these while we are setting up the inputs, and only enable them after the inputs have been set. Then in theory, we should be able to just reset all these Magma primitives before the next test cycle.

Here, we considered two similar designs, but were not able to complete their implementation. These are described in section 3.1 and 3.2 below.

3.2 Freeze Bit

Conceptually, here we would hook up the enable bits of all the Magma primitives to a bit on an extra receiver that MTest controls. Then, we could enable/disable the circuits quite easily. We had tested this by manually wiring the enable bit of a counter in such a way - and it worked reliably.

But, for implementing this in a general setting, we needed an automated way to wire the enable bits of all the primitives. One way we considered was by making a pass over the function to be tested - making appropriate changes as neccessary. But the main drawback of this approach seemed that it did not seem very clean to us - particularly since we will also need to check other imported functions etc. This drawback led us to think about alternative designs, as the one described below.

Another option would be to use python metaprogramming so that these changes are made in a clean way by adding a @test keyword before the function. But we did not have enough time to explore this approach further.

3.3 Custom Clock

The design we were in the process of implementing works as follows:

1. The main.CLKIN throughout the given circuit is replaced with a 8 bit counter_clk. Initially, the counter_clk is disabled.

2. The user provides a third input of 8 bits in the input file, which represents the number of clock cycles from when the input is received to when the output is supposed to be sent back.

3. The enable of the counter_clk is based on this input argument. So as long as the value of this argument is greater than 0, the counter_clk will remain enabled, and the program will continue execution.

4. Once we send the number of cycles to a receiver clk gets enabled. Then at every cycle we subtract 1 from the received cycles until it they become 0. Thus, the clk is enabled for the numbers of cycles we sent over the receiver.

Issues Here, we describe a few issues we ran into while implementing this.

1. We implemented this protocol manually (without integrating it into the MTest framework) and it is unreliable as of now. Sometimes the subtraction continues for an extra cycle causing all 8 bits to not be 0 when checked, and giving a random output.

2. We need to fix the above bug and then proceed to automating it.

3. We are unsure if complex counter dependent circuits will be reset completely with every input using this approach. We will have to set the reset bits of every primitive to 1 when a new input/cycle sequence is received. This will require handling of a lot of edge cases.

3.4 Single UART Pin

Our design was based on modifying an echo module which uses the single UART pin. I think this could have been made simpler if we had actually also used the RTR and DTR bits available on the Icestick (as was done in the MDB project by Duc).

3.5 Shift Registers

Ideally, we felt that we should use some form of a shift register to read in the data supplied by a single UART receiver (8 bits at a time) into a much longer array. In principle, we needed a PIPO register - where the input and output arrays are of different sizes (so the input would be of length 8, because of the UART receiver, and the output could be of length of 24). Since this functionality wasn't provided by Magma, we tried to hack a similar chain by connecting a PISO-SIPO. Conceptually, the idea was that the PISO register would take in the UART receiver, and output 1 bit at a time, which in turn could be fed into a SIPO register with much longer output. This required additional synchronization so the single bit output from the PISO register would stop after 8 bits. But despite getting some results, this produced very unreliable performance, and we weren't able to find the exact issue. So instead, we designed a slightly different (simpler) protocol as described in the Implementation section. Our protocol was more expensive in terms of the space being used on the Icestick (since we declare multiple receiver circuits) but this was a tradeoff as it did make the implementation cleaner.

4 Implementation

4.1 MTest

Example
The testing interface is provided by the MTest class. A simple testing example is provided below, without all the boilerplate magma imports.

```
from add import Add
from MTest.mtest import *

icestick = IceStick()
main = icestick.main()

test = MTest(main, num_inputs
   =3, num_outputs=1)

sum_temp, cout_temp = Add(test.
   receivers[0], test.receivers
   [1], 0)

sum, cout = Add(sum_temp, test.
   receivers[2], cout_temp)

test.set_output_byte(sum)

test.end_circuit()

# Testing phase:
test.test_input_file('testing/
   test_add3.txt')
```

We initialize an MTest object with the number of input receivers, and send it the main Magma object of icestick. The n receiver arrays will be provided in an array - which can be used as desired. For the output, you have to tell MTest which values (of 8 bits or less) need to be wired to the output.
Test case files are given in the format Input1,Input2, — Output, or you could call an alternate function (test_inputs) so you can just or you can just stream single input-output pairs from a python function.

Clone Files
As you see in the example above, after specifying the connections for the receivers, you need to call test.end_circuit. This signals the end of the Magma phase to MTest, and we create a temporary file with all the contents so far, make the neccessary updates to the Makefile etc, and upload it to Magma then.
So far, we have not made any further changes to the temporary file, but in the future, we could make further modifications (for instance, if we wanted to wire the enable bits of all the primitives to a freeze bit).

4.2 UART protocol

For UART, we used an existing Verilog library, which works reliably. [https://github.com/cyrozap/iCEstick-UART-Demo]

For the receiver, it starts reading when the edge falls (start bit) and then it checks the value three times over the baud rate and takes the average, solving the problem of having to implement the UART protocol with a single pin.

4.3 Receiver Protocol

We solved the problem of shift registers (described in Section 3.5) by using a different protocol we designed. We made a Counter of size log2(num-inputs) + N. We use the +N so we have extra cycles to stabilize the output. Whenever we receive an input over the UART receiver, we decode this Counter, store the value in an input byte and, enable the counter for one cycle (to increment it by one). We do this for each of the inputs received so we get the values for every input saved uniquely (theyll be stored in a byte according to the value decoded from the Counter).

To get the next set of inputs (when one test case has finished), we simply increment the counter by sending it 0s over the transmitter. Also, each time the transmitter sends a signal, we echo the output-byte back. Thus, we use these cycles of sending 0s to check when the output-byte stabilizes and show it to the user as the output received from the hardware. We stabilize when a minimum of 2 same values have been received from the hardware. This makes sure that we are able to handle circuits, which stabilize over several cycles. The tradeoff, however, is that the testing takes longer.

5 Evaluation

We tested our test unit on multiple magma/mantle primitives and their slight modifications. The circuits we tested were:

1. **Adder2:** Adding two 8 bit numbers
2. **Adder3:** Adding three 8 bit numbers
3. **Subtractor:** Subtracting two 8 bit numbers
4. **And8** Taking the AND of two 8 bit inputs
5. **NAnd3** Taking the NAnd of three one bit inputs
6. **Mux2** Selecting one out of two input bits based on a third input bit.

These files can all be found in our github repo.

6 Future Work

- Generalize to n-outputs. This should be pretty straightforward to do with a system similar to the n-input receivers we had used.
- Solve the problem of non-combinatorial circuits. We described the approaches we have attempted in sections 3.2 and 3.3 - but these still need more work.
- Use a single receiver object instead of multiple receivers. With the current implementation, we waste additional space on the icestick - so this would not be scalable to large inputs.
- Add support for writing multiple test circuits in the same file, for example, testing an adder and a subtractor.

- Our method of checking the output byte for stabilization worked well for the circuits we tested, but it might be useful to have a more robust protocol for this. For instance, if we could freeze the circuit (as described in point 2 above), then we could ask the user to specify the number of required cycles - and send the output back only then.