Lab 3: Simulation and Testing

University of California, Berkeley
Department of Electrical Engineering and Computer Sciences
EECS150 Components and Design Techniques for Digital Systems
Ronald Fearing, Austin Buchan, Stephen Twigg
Due October 3rd, 2013

Table of Contents

0 Introduction 1
1 Prelab 2
2 Lab Procedure 2
2.1 Relevance to Your Final Project 2
2.2 Functional Specification 3
2.3 Lab Resources 4
2.4 Testing the Design 4
2.4.1 Verilog Testbench 4
2.4.2 Test Vector Testbench 5
2.4.3 Writing Test Vectors 5
2.5 Using Modelsim 6
2.6 Viewing Waveforms 7
3 Checkoff 10

0 Introduction

In this lab, you will learn how to simulate your modules and test them in software before pushing them to the board. In the previous labs, you had to push your code through the entire tool chain and impact the bit stream onto the FPGA before you could verify that your design worked.

This is feasible for simple designs that can quickly be synthesized and quickly verified on the board, but this approach does not scale. Here we will cover how to simulate a hardware design and write test benches, both of which are essential in the verification process of large and complex systems.

We will also be introducing the Finite Impulse-Response (FIR) filter, which will be a versatile component of the project SIFT tracking implementation. We will examine a modular design by chaining together Multiply ACumulate (MAC) units that in practice will synthesize to specialized DSP logic in an FPGA.

1 Prelab

It is highly suggested that you complete the Verilog for this lab before you begin. Additionally, answer the following questions about FIR filters after reading the functional specification in 2.2:

1. Why would we want a register in between the multiplication and addition stage of the DSP48E?

2. If x[n] and h_k are of bit width w, how wide must each of R0-R4 be to prevent overflow in an N-tap FIR?

3. How many valid outputs will be produced if 100 consecutive valid inputs are sent through a 7-tap FIR filter?

4. What would the coefficients of a 4-tap moving average filter be (i.e. y[n] is the average of x[n],x[n-1],x[n-2],x[n-3])? What about y[n] = 2*x[n] - x[n-1] + x[n-3] (use the convention below for coefficient index)?

5. Complete the following RTL timing chart by entering the contents of each register of a two-stage FIR (assume the coefficient registers have already been loaded).

Clock	MAC0 (coeff = h_0)				MAC1 (coeff = h_1)			
	R0	R1	R2	R3	R0	R1	R2	R3
0	x[0]	-	-	-	-	-	-	-
1	x[1]	x[0]	-	-	-	-	-	-
2	x[2]	x[1]	x[0] h_0	-	x[0]	-	-	-
3								
4								
5								
6								
7								

6. If x[n] is at the input of an N-tap FIR chain at clock cycle 0, when is y[n] available at the output?

2 Lab Procedure

In this lab, you will be writing an FIR filter that you will be using later on for the next lab and your project. You will also be learning techniques for simulating and verifying your design which are critical aspects of the development flow and for your project.

2.1 Relevance to Your Final Project

FIR filters essentially compute a filtered output y[n] as a convolution of a signal vector x[n] and a weight vector h_k:

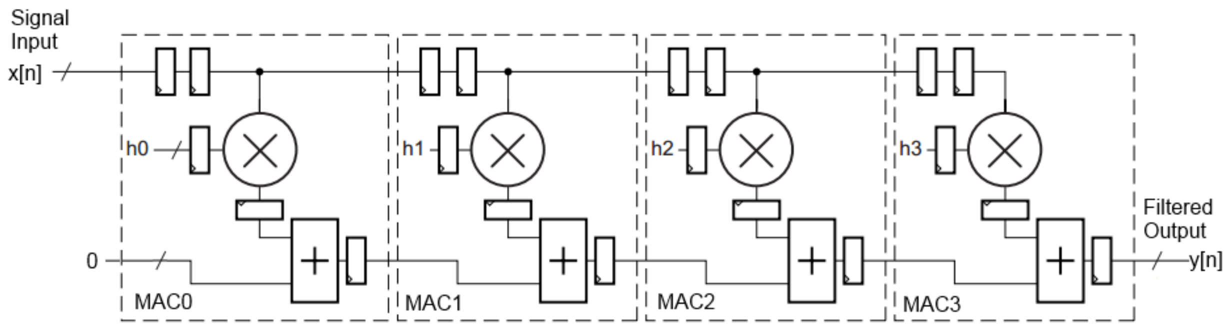
$$y[n] = \sum_{k=0}^{N-1} x[n-k]h_k$$

N is the number of coefficients in the filter, also known as "taps." By choosing the appropriate weights for h_k, the filter can behave in several useful modes such as an edge detector, Gaussian blur, or moving average filter. We will only be considering integer math in this lab, but fixed-point fractional representations can easily be implemented by appropriately bit-shifting the coefficients and filter output.

We will be particularly interested in the Gaussian and edge detection modes in the final project, since several of these need to be computed repeatedly on windowed regions of the input image. The implementation we look at below is a streaming pipelined approach that is optimized for maximal throughput using the DSP48E units available in the Virtex-5 FPGA. A detailed discussion of the structure in the DSP48E is available in the Xilinx XtremeDSP Design Considerations manual. Before continuing, take a look at the Slice diagram on page 16 of this manual to familiarize yourself with the internals of a DSP48E.

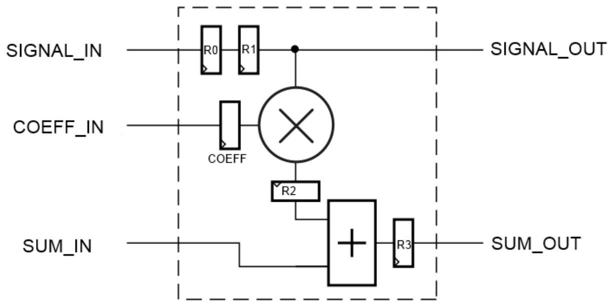
2.2 Functional Specification

To fully leverage the DSP hardware available on our FPGAs, we will be using a Systolic FIR architecture. This means that the filter is implemented as a chain of identical processing units connected such that the output of one stage is the input of the next. Using generate statements and parameters, we can easily code this structure in Verilog to use an arbitrary number of taps. The following diagrams[1] show a 4-tap Systolic FIR, with detail on the register labeling of a MAC unit. Note the register labels in the MAC detail, you can assume that labeling for the prelab question.



The RTL specification for a MAC unit is:

```
If(RESET==1)
      R[0-3]←0, COEFF←0;
Else if (LOAD==1)
      COEFF←COEFF_IN;
Else
      R1←SIGNAL_IN, R1←R0,
      R2←COEFF*R1,
      R4 ← R2 + SUM_IN;
```



[1] Slightly modified from Xilinx's DSP: Designing for Optimal Results

In addition to the filter functionality, your design will include a ValidIn and ValidOut signal. ValidIn is asserted when valid input is available at the FIR signal input (i.e. a value of x[n] is about to be clocked in). ValidOut should be asserted when the output is valid (i.e. y[n] is at the output). This means that all of the values propagated through the FIR pipeline were valid when used to compute the current output. This behavior follows the specification for the Valid handshake in this reference, so If you have questions about the details refer to that.

For this lab you can assume that all input signal streams will have ValidIn continuously asserted, and that the values arrive without gaps. You must implement the logic to determine when ValidOut should be asserted. Ways you might think about implementing this include a state machine with a counter, or a 1-bit shift register that mimics the data path length of the input values.

2.3 Lab Resources

To retrieve the lab resources, run the wget command and decompress the tar file. Again the extraction command is:

```
% wget http://inst.eecs.berkeley.edu/~cs150/fa13/lab3/lab3.tar.gz
% tar –xvzf lab3.tar.gz
```

Make sure that you have both a `/src` and `/sim` folder in the `/lab3` directory. Notice that there is no Makefile on the top level as we will not be synthesizing our design to the board, only verifying it works in simulation. Optionally, you can copy the Makefile from previous labs to run the MAC or FIR designs through synthesis, and verify that they use the number of DSP48E units as you expect.

2.4 Testing the Design

Before writing any of our modules, we will first write the tests so that once you've written the modules you'll be able to test them immediately. Another reason why you should write your tests first is that if you need to change your module design, you can always run it against your test to see if it still works. You should also understand the expected functionality of these modules before writing any code or tests.

There are a few approaches you can use to test your design. For this lab, you will only be testing two modules, so you will resort to unit testing. For your project, you will be expected to write unit tests for any modules that you design and implement and write integration tests.

2.4.1 Verilog Testbench

One way of testing Verilog code is with test bench files. The skeleton of a test bench file has been provided for you in MACTestbench.v. We have already declared all of the signal names, but you should fill out the parameterized widths of the signals based on your answers in the prelab to prevent overflow. There are several important parts of this file to note:

1. `` `timescale 1ns / 1ps `` - This specifies the reference time unit and the time precision. This means that every delay in the test bench is 1ns long and the simulation should be accurate up to 1ps.

2. Clock generation is done with the code below.

    a. The `initial` block to set the clock to 0 at the beginning of the simulation. You *must* start the clock at 0, otherwise you will be trying to change inputs at the same time the clocks changes and it will cause strange behavior.

b. You must use an `always` block without a trigger list to cause the `Clock` to change by itself

```
parameter Halfcycle = 5; //half period is 5ns
localparam Cycle = 2*Halfcycle;
reg Clock;
// Clock Signal generation:
initial Clock = 0;
always #(Halfcycle) Clock = ~Clock;
```

3. `task checkOutput;` - this task encapsulates some Verilog that you would otherwise have to copy paste over and over. Note that it is **not** the same thing as a function (as Verilog also has functions).

For the MACTestbench unit, you will need to implement the body of the checkOutput task. Remember that the behavior of the checkOutput task is not going to be synthesized to hardware, so you can treat the variables changes as more of a scripted behavior rather than worrying about the semantics of hardware updates. You should be able to accomplish assigning the sequences of signals using only blocking assignments and delays. Make sure that the task waits long enough before executing the if statements that compare the output to what is expected.

Once implemented, you should be able to call checkOutput and it will exercise the MAC unit with the specified inputs, and checking the desired outputs. We've included a few very simple test cases at the end of MACTestbench that should pass when you have implemented checkOutput (and the MAC unit). Add some test cases of your own, including negative numbers for the signal, coefficient, and sum in.

After implementing checkOutput and including a few of your own test cases, you should use it to test a range of values. This can be done with a for loop, and calculating the test cases based on the loop index variable. Write code that exhaustively tests values of signal, coeff = [-8..8]. Optionally, you can look up the documentation on the Verilog `$random` command, which will generate a random bit vector that you can use as inputs to a test case.

2.4.2 Test Vector Testbench

An alternative way of testing is to use a test vector, which is a series of bit arrays that map to the inputs and outputs of your module. The inputs can be all applied at once if you are testing a combinational logic block, or applied over time for a sequential logic block (e.g. an FSM or pipeline).

We have provided a full implementation of FIRTestbench since the details of file input and ValidOut signal detection can be quite tricky. This test bench can be considered the specification standard that your FIR should match when complete. It uses files `coeff.input`, `signal.input`, and `filt.input` to describe respectively the filter coefficients, signal input, and expected filtered output.

In FIRTestbench you should look carefully at the use of `$readmemb` to read in a vector of values to a Verilog signal. Also pay attention to which signals are declared as `wire`, `reg`, and `integer`.

2.4.3 **Writing Test Vectors**

Additionally, you will also have to generate actual test vectors to use in your test bench. A test vector can either be generated in Verilog (as we saw in the MACTestbench), or using a scripting language. Since we have already written a Verilog test bench for the MAC unit, we will tackle automatically generating test vectors.

We've provided a test vector generator written in Python, which is a popular language used for scripting. We used this generator to generate the test vectors provided to you. If you're curious, you can read the next paragraph and poke around in the file. If not, feel free to skip ahead to the next section.

By default, the script generates 4 random coefficients and 128 random signal inputs, both of width 16. The output signal is calculated for values of 2*width, but the input range is restricted to prevent overflow. If you change these parameters for your hardware design, make sure you change the test generation script parameters as well. For convenience, all of the signal values are also output to a data.csv file, which can be easily viewed in graph form using a spreadsheet program. There are helper functions that can generate sinusoidal input signals if you want to play around with that. This will be especially useful if you want to impose some sort of structure on the input values, such as having the value of x[n] correspond to the index to help with debugging.

The script `GenerateFIRTest.py` is located in `sim/tests`. From that directory, run it so that it generates test vector files in the `/sim/tests/` folder:

```
% ./GenerateFIRTest.py
```

If you modify the Python script, run the generator again to make new test vectors. This will overwrite the file, so don't do this if you are still debugging a set of test vectors. If you want to save generated files for later, rename them so they aren't overwritten, and name then back to the original input filenames when you want to use them.

2.5 Using Modelsim

Once you've written your test benches as well as implemented the Verilog modules, you can now simulate your design. In this class, you will be using `ModelSim`, a popular hardware simulation and debugging environment. The staff has wrapped up the functionality that you will need from `ModelSim` in a `Makefile`. To simulate your design, you must first compile it and fix any syntax errors that arise:

```
% cd ~/lab3/sim
% make compile
```

Once you have your design compiling, you need to run some test cases. The build system looks inside the `tests` directory for test cases to run. Each test case is a `.do` file, which is a script in Tcl, a scripting language used by a variety of CAD tools. For the most part you don't need to worry about the details of Tcl; you will just be using it to issue commands directly to `ModelSim`. The following is the Tcl script that runs MACTestbench.

```
set MODULE MACTestbench
start $MODULE
add wave $MODULE/*
add wave $MODULE/dut/*
run 100us
```

The first line sets the value of the variable `MODULE` to `MACTestbench`. Its value is referenced through the rest of the script as `$MODULE`. The `start` command tells `ModelSim` which Verilog module it should simulate.

The `add` command is interesting. By default, `ModelSim` doesn't collect any waveform information from the simulation. '`*`' is a shortcut for "anything", so these commands tell `ModelSim` to record the signals for all the signals in the test bench as well as the signals in `dut`. Once you start building designs with more complexity, you may want to look at the signals inside a given submodule. To add these signals, simply edit the `.do` file by adding a new "add wave <target>" command; for example, if DUT1 and DUT2 contain a module called my_submodule:

```
add wave $MODULE/DUT1/my_submodule/*
add wave $MODULE/DUT2/my_submodule/*
```

Finally, the `run` command actually runs the simulation. It takes an amount of time as an argument, in this case `100us` (100 microseconds). Other units (ns, ms, s) are possible. The simulation will run for this amount of time. In most cases this will serve as a timeout because your test benches should cause the simulation to exit (using the `$finish()` system call) when they are done.

Let's try running the simulation. To run all of the cases in the tests directory:

```
% make
```

This will first recompile your design if needed, then run the simulation. Other commands that may be useful are:

•. `make clean`: Sometimes you can accidentally cancel a simulation or otherwise cause make to believe that your simulation results are up to date even if they aren't. If you're in doubt, run this command before running `make`.

•. `make results/<testcasename>.transcript`: When you have multiple test benches in your project and you only want to run one of them.

You should see the output of simulation printed to your terminal. It will also be written to `results/<testcasename>.transcript`. You should see the one of the following lines in the output:

```
# FAIL: Incorrect result for opcode 000000, funct: 100011:
# A: 0xdbfa08fd, B: 0x318c32a8, DUTout: 0xaa6dd655, REFout:        0x559229ab
```

Or:

```
# ALL TESTS PASSED!
```

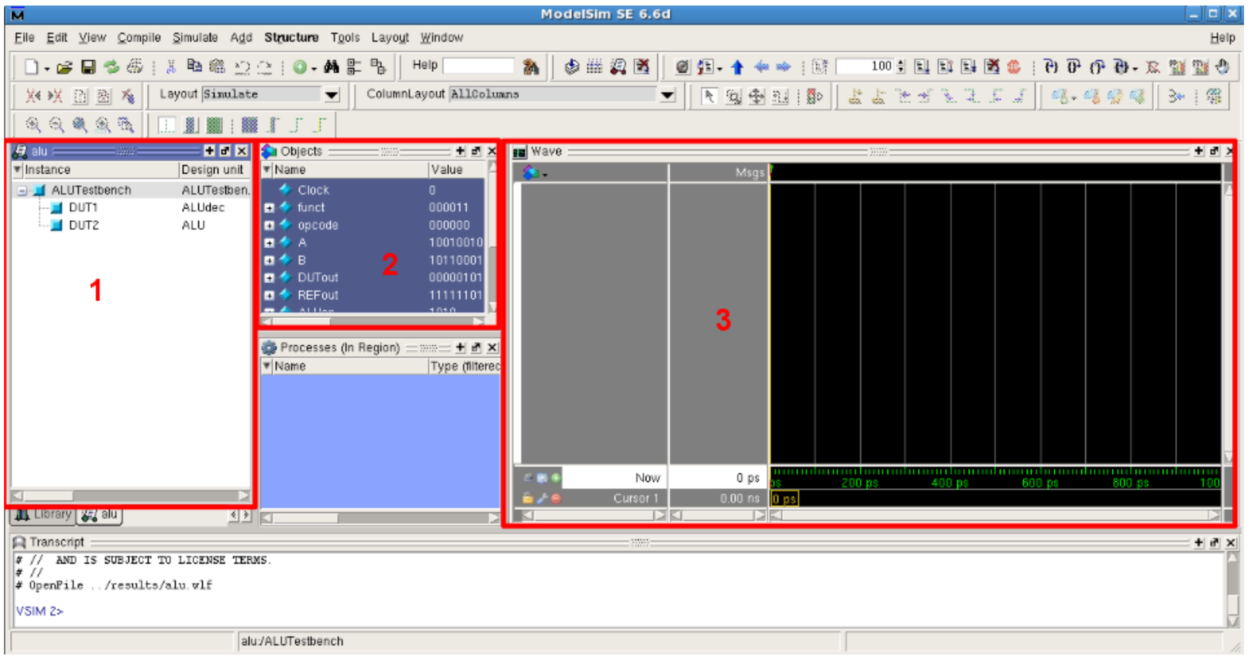
2.6 Viewing Waveforms

After simulation completes you can view the waveforms for signals that you added in your test case script. The waveform database is stored in `.wlf` files inside the `results` directory. To view them use the viewwave script included in the sim directory.

```
% ./viewwave results/mac.wlf
```

This will pop open a `ModelSim` window that shows you a hierarchical view of the signals that your simulation captured.

Note: ModelSim is your FRIEND! Throughout the course of the project, ModelSim will be your primary tool for debugging your designs. It is very important that you spend the time to understand how to run tests and use ModelSim to view the results.



```
# //  AND IS SUBJECT TO LICENSE TERMS.
# //
# OpenFile ../results/alu.vlf

VSIM 2>
```

The above is a screenshot of `ModelSim` when you first open it. The boxed screens are:

1. List of the modules involved in the test bench. You can select one of these to have its signals show up in the object window.
2. Object window - this lists all the wires and regs in your module. You can add signals to the waveform view by selecting them, right-clicking, and doing `Add > To Wave > Selected Signals`.
3. Waveform viewer - The signals that you add from the object window show up here. You can navigate the waves by searching for specific values, or going forward or backward one transition at a time.

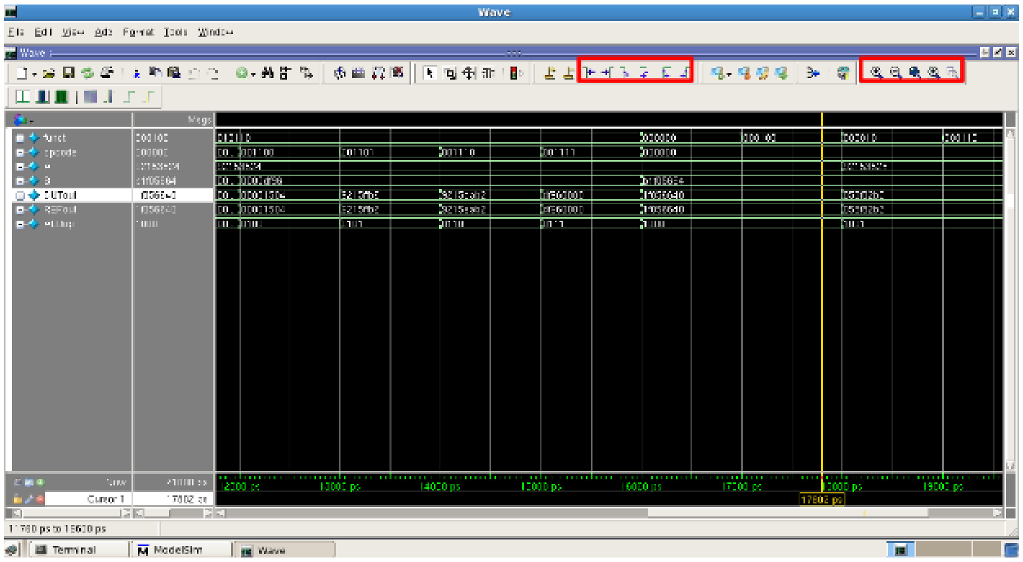
As an example of how to use the waveform viewer, suppose you get the following output when you run a hypothetical ALUTestbench:

```
...

# PASS: opcode 000000, funct 000110
#       A: 0x92153525, B: 0xb1f05664, DUTout: 0x058f82b3,
      REFout: 0x058f82b3
# FAIL: Incorrect result for opcode 000000, funct: 000011:
#       A: 0x92153525, B: 0xb1f05664, DUTout: 0x058f82b3,
      REFout: 0xfd8f82b3
```

The `$display()` statement actually already tells you everything you need to know to fix your bug, but you'll find that this is not always the case. For example, if you have an FSM and you need to look at multiple time steps, the waveform viewer presents the data in a much neater format. If your design had more than one clock domain, it would also be nearly impossible to tell what was going on with only `$display()` statements. Besides, you want to get some practice using ModelSim anyhow.

You add all the signals from `ALUTestbench` to the waveform viewer and you see the following window:



The two highlighted boxes contain the tools for navigation and zoom. You can hover over the icons to find out more about what each of them do. You can find the location (time) in the waveform viewer where the test bench failed by:

1. Selecting `DUTout`

2. Clicking `Edit > Wave Signal Search > Search for Signal Value > 0x058f82b3`

Now you can examine all the other signal values at this time. You notice that `REFout` has a value of `0xfd8f82b3`. From the `opcode` and the `funct`, you know that this is supposed to be `SRA` instruction, and it looks like your ALU performed a SRL instead. However, you wrote

```
Out = B >>> A[4:0];
```

That looks like it should work, but it doesn't! It turns out you need to tell Verilog to treat B as a signed number for SRA to work as you wish. You change the line to say:

```
Out = $signed(B) >>> A[4:0];
```

After making this change, you run the tests again and cross your fingers. Hopefully, you will see the line: `# ALL TESTS PASSED!` If not, you will need to debug your module until all test from the test vector file and the hardcoded test cases pass.

ModelSim has quite a few features that may be useful in certain instances; far too many to detail here. A more in-depth tutorial is provided on the class website here. If you need to do something with ModelSim, and you feel like that functionality should exist already, Google it. Or ask a TA. But try Google first. If you discover something useful, share your findings!

3 Checkoff

Congratulations! You've written and thoroughly tested a key component in a DSP pipeline and should now be well-versed in testing Verilog modules. Please answer the following questions to be checked off by a TA.

1. In FIRTestbench, the inputs to the FIR were generated randomly. When would it be preferable to perform an exhaustive test rather than a random test?

2. What bugs, if any, did your test bench help you catch?

3. For one of your bugs, come up with a short assembly program that would have failed had you not caught the bug.

Also be prepared to show your working FIR and MAC test bench files to your TA and explain your hardcoded cases.

You should be able to show that the tests for the test vectors generated by the python script and your hardcoded test vectors for the MAC unit both work.