

MS Project Report

Joseph Luciano

Introduction

The *cocotb* library provides a Python interface to an RTL simulator (Icarus, GHDL, VCS, ModelSim, etc.) to drive and monitor signals. The library offers many features to assist in the test bench construction and verification process, much like the features used in the Universal Verification Methodology (UVM) for System Verilog. According to the docs, “*cocotb is a Coroutine based COsimulation TestBench environment for verifying VHDL and systemVerilog RTL using Python.*” Alternatively, one could say that *cocotb* is an RTL simulator plugin with a Python library for writing logic to verify your designs. *cocotb* offers an alternative to traditional hardware verification involving UVM/OVM, or any test bench written in VHDL or Verilog code.

Python is an interpreted, object-oriented, high-level programming language that has an enormous community that constantly generates new libraries and capabilities that can be used to enhance a traditional test bench. Python can also manipulate data in ways a hardware language simply cannot (although System Verilog offers some functionality here). For example, the Python library *Matplotlib* can be imported to help visualize data collected during simulation. Alternatively, signals can be packetized and manipulated with *Scapy* rather than writing all the packetization code in a UVM/OVM setting. For more advanced techniques, a machine learning library such as *TensorFlow* could be imported and used to train a model on signals observed in the simulator.

Python is already an extremely popular programming language that is growing in popularity throughout the technology industry. As of early 2020, 44.1% of developers use Python (even at companies that perform hardware verification such as Intel, AMD, Nvidia, etc). However, Computer Engineering students at Binghamton are only exposed to MATLAB, C, and VHDL/Verilog with little experience in a Shell terminal. *cocotb* offers students an opportunity to gain experience with one of the most popular programming languages while learning hardware verification techniques. *cocotb* could grow into a widely adopted tool across the verification industry given the growing community and rapid development from a number of contributors.

This paper will first explain how to setup and prepare *cocotb* for use in a Linux environment. Next, the Capabilities section will discuss what is possible with *cocotb* while showing code snippets to demonstrate the concepts in action. A full adder will be tested before simulating an ARM processor to produce insightful charts. Advanced techniques for large and complex designs will also be discussed before making final remarks.

Motivation

This past summer I got experience working with an extremely large codebase. Within it were at least a dozen different programming languages all interacting with and utilizing each other to

perform impressive tasks. I found this incredibly insightful, and it inspired me to begin exploring new programming languages and building programs on my own. I believe experience with *cocotb* could offer a similar insight to students by observing their Python code drive signals into the simulation of their VHDL or Verilog code. I see using *cocotb* as a great opportunity to demonstrate fundamental hardware and software development concepts.

Installation and Setup

cocotb has an installation and quick start guide: <https://docs.cocotb.org/en/stable/install.html>

I installed *cocotb* on a Linux Debian virtual machine using VirtualBox. This was after unsuccessfully attempting to set it up on my Windows Desktop. The installation and setup can be accomplished by running the following two commands in a Linux terminal:

```
- sudo apt-get install make gcc g++ python3 python3-dev python3-pip
- pip install cocotb
```

Since I was interested in testing VHDL code, I needed to download a VHDL simulator and waveform viewer; I chose GHDL and GTKWave. I installed GHDL and GTKWave with the following command:

```
• sudo apt-get install ghdl gtkwave
```

Once *cocotb* was installed, I downloaded the GitHub files which can be stored anywhere on the Linux machine. The repository contains a lot of helpful examples to begin understanding the best practices to follow when writing new code. To run an example, first navigate within the repository's directory (ie. 'cocotb-1.3.2' or 'cocotb-master') and go to '/examples/dff/tests/'. Second, run the `make` command in the collocated terminal to run a simulation and test of a flip flop module. This will not run if you do not already have the default simulator (Icarus Verilog) installed. To run the VHDL example code with the GHDL simulator, run the following command:

```
• make TOPLEVEL_LANG=vhdl SIM=ghdl
```

When making a new Makefile, the default language and simulator can be specified. Later examples will show how this can be done. The *cocotb* repository and more can be found at: <https://github.com/cocotb/cocotb>

Before writing my own code, I installed PyCharm for my IDE. PyCharm integrates a wide range of tools to assist you when writing your code. However, it will primarily help with writing Python code rather than the Makefile or VHDL/Verilog. You can find the installation guide here: <https://www.jetbrains.com/help/pycharm/installation-guide.html#standalone>. Alternatively, you could use Visual Studio Code which offers a similar experience: <https://code.visualstudio.com/docs/setup/linux>

To get a rudimentary education on *cocotb*, I watched a series of videos by *cocotb* contributors. I found this content very insightful, and some presentations offer a great introduction to general verification topics. These videos do a great job providing examples that highlight the key

features that any new *cocotb* user would need to know. Additional *cocotb* resources can be found at: <https://github.com/cocotb/cocotb/wiki/Further-Resources>.

Capabilities

A *cocotb* test can be as basic as assigning a value to a signal, waiting for the signal to propagate, and observing the result. In more advanced applications, *cocotb* can generate hundreds of tests with a ‘TestFactory’ and verify the results with a ‘Scoreboard’. This section will first cover some of the basic *cocotb* capabilities such as making assignments to the design under test (dut), raising test failures, and logging relevant information. Second, a walkthrough of coroutines, third party libraries, and a class-based test bench will be given. Lastly, advanced capabilities such as TestFactory and Scoreboard will be presented.

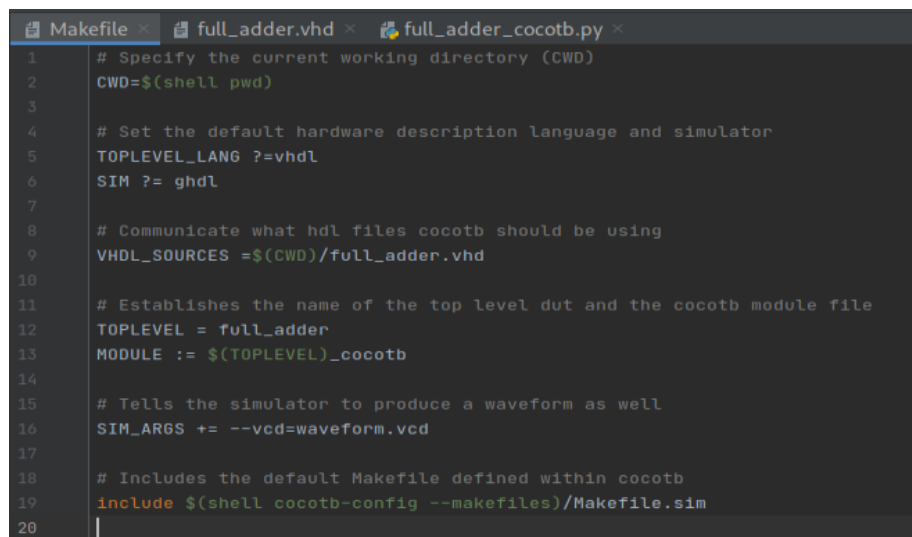
Basic - Testing a Full Adder with *cocotb*

The first example will show how a *cocotb* testbench can be created and run. A full adder module design will be tested to demonstrate basic Python and *cocotb* features as well as the results of the simulation and *cocotb* tests. Please refer to **Appendix A – Full Adder Code** for the final code.

Full Adder File Structure and Default Terminal Output

Once *cocotb* is installed, a test can be run with as little as 3 files:

1. Makefile – sets the configuration of the *cocotb* tests such as the language and simulator
2. module.vhd – the hardware description file(s) of the dut
3. module_cocotb.py – the ‘*cocotb* file’ that directs the simulator and produces results

A screenshot of a code editor showing a Makefile for a Full Adder testbench. The editor has three tabs: 'Makefile', 'full_adder.vhd', and 'full_adder_cocotb.py'. The Makefile content is as follows:

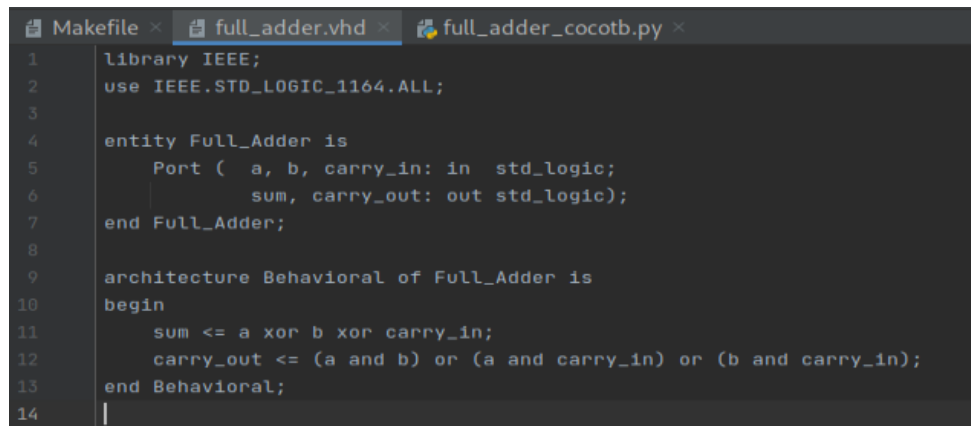
```
1 # Specify the current working directory (CWD)
2 CWD=$(shell pwd)
3
4 # Set the default hardware description language and simulator
5 TOPLEVEL_LANG ?=vhdl
6 SIM ?= ghdl
7
8 # Communicate what hdl files cocotb should be using
9 VHDL_SOURCES =$(CWD)/full_adder.vhd
10
11 # Establishes the name of the top level dut and the cocotb module file
12 TOPLEVEL = full_adder
13 MODULE := $(TOPLEVEL)_cocotb
14
15 # Tells the simulator to produce a waveform as well
16 SIM_ARGS += --vcd=waveform.vcd
17
18 # Includes the default Makefile defined within cocotb
19 include $(shell cocotb-config --makefiles)/Makefile.sim
20
```

Figure 1: Full Adder Makefile

Figure 1: Full Adder Makefile above displays the bare minimum that should be included in a Makefile (the waveform is not required but is usually generated in all test runs). Additional Makefile logic can be performed such as selecting the simulator or source files based on the chosen language. Other information can be communicated with *cocotb* to generate the desired

outputs or simulate under the proper conditions. For example, additional arguments can be passed to the simulator to overrun the default settings. For the purposes of this demonstration no more is required than:

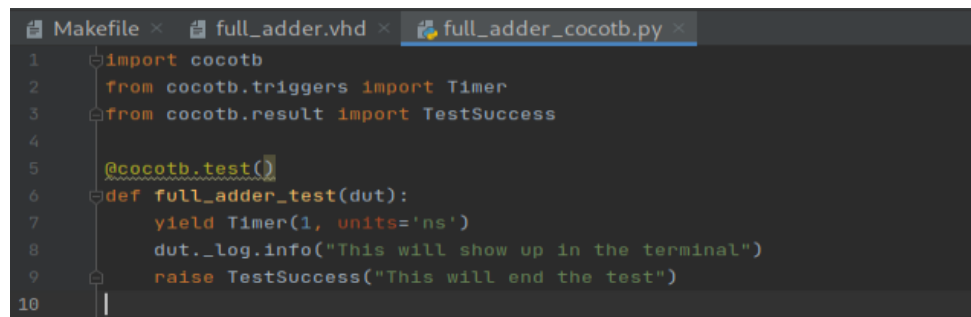
- Specifying the hardware description language (HDL) and compatible simulator
- Pointing to the HDL source files
- Naming the top-level entity and the module (TOPLEVEL and the source file must match)
- Including the default simulation Makefile provided with *cocotb*



```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity Full_Adder is
5     Port ( a, b, carry_in: in std_logic;
6           sum, carry_out: out std_logic);
7 end Full_Adder;
8
9 architecture Behavioral of Full_Adder is
10 begin
11     sum <= a xor b xor carry_in;
12     carry_out <= (a and b) or (a and carry_in) or (b and carry_in);
13 end Behavioral;
14
```

Figure 2: Full Adder Source File

Figure 2: Full Adder Source File above displays a basic full adder module written in VHDL.



```
1 import cocotb
2 from cocotb.triggers import Timer
3 from cocotb.result import TestSuccess
4
5 @cocotb.test()
6 def full_adder_test(dut):
7     yield Timer(1, units='ns')
8     dut._log.info("This will show up in the terminal")
9     raise TestSuccess("This will end the test")
10
```

Figure 3: Full Adder cocotb Test

Figure 3: Full Adder cocotb Test above demonstrates both features of Python and *cocotb*. The Python features include:

- Importing packages, modules, and submodules from the *cocotb* libraries
- Defining a function with a decorator (the `@cocotb.test()` decorator is exclusive to *cocotb*)
- Utilizing the `yield` and `raise` statements

Line 1 imports the *cocotb* package which contains default modules that enable the file to be recognized by *cocotb* when running the Makefile. Lines 2 and 3 import the ‘Timer’ and ‘TestSuccess’ functions from *cocotb*’s ‘triggers’ and ‘result’ submodules. Triggers are tasks that control when the execution of the current coroutine will continue. An ‘await’ or ‘yield’ statement

will stop execution of the current coroutine, and the completion of the trigger (such as a timer or clock edge) will resume execution. Results end the simulation and can return debug information in the case of an error. For more on tasks and coroutines in Python, please reference:

<https://docs.python.org/3/library/asyncio-task.html>.

Line 5 uses a test decorator to signify the function definition on line 6 as a test. In Python, decorators are functions that can be used to add functionality to other functions. For example, a function that returns a string could be decorated with a function that takes that string and capitalizes the first letter before returning the result. *cocotb* uses decorators to identify the tests, coroutines, and other functions that are tied to the simulation.

Line 6 defines the ‘full_adder_test’ function which takes the ‘dut’ parameter. Since this function is decorated as a *cocotb* test, a ‘dut’ object of the top-level module will be passed into it. In this case, the full adder component is the top-level module. The ‘dut’ object allows each submodule and signal of the top-level component to be accessed in addition to other functionality such as logging. Line 8 accesses the dut objects Logger with ‘_log’. The string within ‘info’ will be displayed in the terminal debug when the simulation is run.

Lines 7 and 9 utilize Python’s ‘yield’ and ‘raise’ keywords. A yield keyword suspends operation until the code to the right of it finishes running. However, a yield statement can be used for much more than what is shown in this example. *cocotb* now recommends using the ‘await’ keyword rather than the ‘yield’ keyword. In this case, the *cocotb* timer waits until 1 nanosecond passes in the simulation which delays the Python code so that signals may propagate within the simulator before performing another read or write. The raise keyword is used to raise exceptions; *cocotb* implements its own set of exceptions in the ‘results’ submodule to accommodate full testbench functionality. ‘TestFailure’ is another function from the results submodule that stops the simulation and returns debug information.

```

joe@debian: ~/MS_Project/CocotB/Full_Adder_Test
joe@debian:~/MS_Project/CocotB/Full_Adder_Test$ make
make results.xml
make[1]: Entering directory '/home/joe/MS_Project/CocotB/Full_Adder_Test'
./usr/bin/ghdl -i --workdir=sim_build --workwork /home/joe/MS_Project/CocotB/Full_Adder_Test/Full_Adder.vhd && \
./usr/bin/ghdl -m --workdir=sim_build --workwork full_adder
MODULE=full_adder_cocotb TESTCASE= TOPLEVEL=full_adder TOPLEVEL_LANG=vhdl \
./usr/bin/ghdl -r --workdir=sim_build --workwork full_adder --vpi=/home/joe/.local/lib/python3.7/site-packages/cocotb/libs/libcocotbvpi_ghdl.so --vcd=waveform.vcd
loading VPI module '/home/joe/.local/lib/python3.7/site-packages/cocotb/libs/libcocotbvpi_ghdl.so'
--ns INFO cocotb.gpi ..nbed/gpi_embed.cpp:174 in set_program_name_in_venv Did not detect Python virtual environment. Using system-wide Python interpreter
--ns INFO cocotb.gpi ../gpi/GpiCommon.cpp:105 in gpi_print_registered_impl VPI registered
VPI module loaded!
--ns WARNING cocotb.gpi ../b/vpi/VpiCbHdl.cpp:493 in run_callback Unable to get argv and argc from simulator:
--ns INFO cocotb.gpi ../nbed/gpi_embed.cpp:244 in embed_sim_init Python interpreter initialized and cocotb loaded!
0.00ns WARNING cocotb.gpi VpiImpl.cpp:180 in get_simulator_product Could not obtain info about the simulator
0.00ns INFO cocotb ..init...py:202 in _initialise_testbench Running on UNKNOWN version UNKNOWN
0.00ns INFO cocotb ..init...py:209 in _initialise_testbench Running tests with cocotb v1.4.0 from /home/joe/.local/lib/python3.7/site-packages/cocotb
0.00ns INFO cocotb ..init...py:229 in _initialise_testbench Seeding Python random module with 1620402997
0.00ns INFO cocotb.regression regression.py:127 in __init__ Found test full_adder_cocotb.Full_Adder_Test
0.00ns INFO cocotb.regression regression.py:463 in _start_test Starting test: 'full_adder_test'
0.00ns INFO ..st.full_adder_test,0x7f1067634198 decorators.py:256 in _advance Description: None
This will show up in the terminal
Test Passed: full_adder_test
Passed 1 tests (0 skipped)
** TEST PASSES/FAIL SIM TIME(NS) REAL TIME(S) RATIO(NS/S) **
** full_adder_cocotb.Full_Adder_Test PASSES 1.00 0.00 1773.49 **
*****
** ERRORS : 0 **
*****
** SIM TIME : 1.00 NS **
** REAL TIME : 0.01 S **
** SIM / REAL TIME : 147.96 NS/S **
*****
1.00ns INFO cocotb.full_adder full_adder_cocotb.py:8 in full_adder_test
1.00ns INFO cocotb.regression regression.py:351 in _score_test
1.00ns INFO cocotb.regression regression.py:479 in _log_test_summary
1.00ns INFO cocotb.regression regression.py:548 in _log_test_summary
1.00ns INFO cocotb.regression regression.py:565 in _log_sim_summary
1.00ns INFO cocotb.regression regression.py:255 in tear_down
make[1]: Leaving directory '/home/joe/MS_Project/CocotB/Full_Adder_Test'
joe@debian:~/MS_Project/CocotB/Full_Adder_Test$

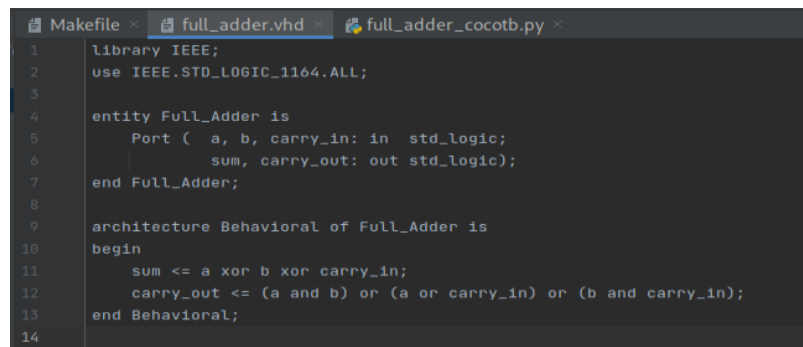
```

Figure 4: Full Adder Test Output

Figure 4: Full Adder Test Output above displays the output when running the code shown in the 3 previous figures. All the content is controlled by *cocotb* except for the line with ‘This will show up in the terminal’, which was manually logged in the Python test that was written above in Figure 3: Full Adder *cocotb* Test. With more tests and debug information, the terminal window can quickly grow and obscure relevant information.

Full Adder Bug Testing

The previous example does not drive any signals or monitor the outputs. Therefore, not verifying the operation of the VHDL component. However, before testing the component, there should be an error to find. Figure 5: Full Adder Source File with Error below shows a modification to line 12 that will lead to errors.



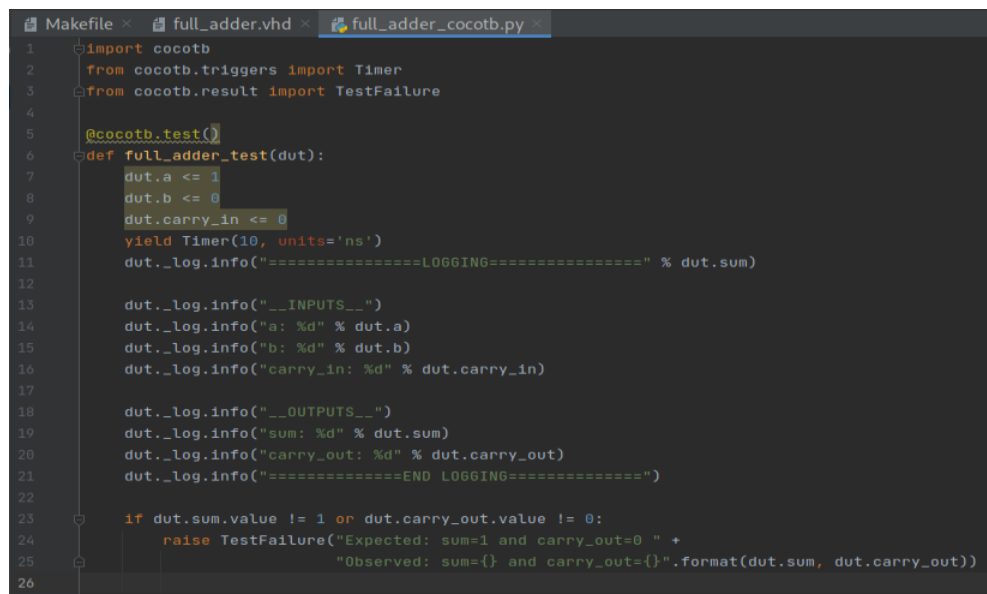
```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity Full_Adder is
5      Port ( a, b, carry_in: in std_logic;
6            sum, carry_out: out std_logic);
7  end Full_Adder;
8
9  architecture Behavioral of Full_Adder is
10 begin
11     sum <= a xor b xor carry_in;
12     carry_out <= (a and b) or (a or carry_in) or (b and carry_in);
13 end Behavioral;
14

```

Figure 5: Full Adder Source File with Error

A single bit full adder has 3 inputs (a, b, carry_in), each with 2 possible values (0, 1) yielding 8 possible combinations. A testbench could contain 8 different tests, each driving the unique set of signals to the DUT. Due to the small scope of the full adder, it is possible to comprehensively test the component in this way. However, a more complex module would require other verification techniques to ensure the design meets the specifications.



```

1  import cocotb
2  from cocotb.triggers import Timer
3  from cocotb.result import TestFailure
4
5  @cocotb.test()
6  def full_adder_test(dut):
7      dut.a <= 1
8      dut.b <= 0
9      dut.carry_in <= 0
10     yield Timer(10, units='ns')
11     dut._log.info("=====LOGGING===== " % dut.sum)
12
13     dut._log.info("__INPUTS__")
14     dut._log.info("a: %d" % dut.a)
15     dut._log.info("b: %d" % dut.b)
16     dut._log.info("carry_in: %d" % dut.carry_in)
17
18     dut._log.info("__OUTPUTS__")
19     dut._log.info("sum: %d" % dut.sum)
20     dut._log.info("carry_out: %d" % dut.carry_out)
21     dut._log.info("=====END LOGGING=====")
22
23     if dut.sum.value != 1 or dut.carry_out.value != 0:
24         raise TestFailure("Expected: sum=1 and carry_out=0 " +
25                           "Observed: sum={} and carry_out={}".format(dut.sum, dut.carry_out))
26

```

Figure 6: Full Adder *cocotb* Test with Error

Figure 6: Full Adder cocotb Test with Error, only shows the 1 of 8 tests that would expose the error. Note that only 'TestFailure' is imported on line 3. Signal a is assigned '1' while signals b and carry_in are assigned a '0'. The '<=' operator converts the number to the right of it to a binary string before assigning it to the signal. Signal <= 5 is equivalent to Signal = BinaryValue("5"). The '<=' operator would not be necessary in this case since only 1-bit signals are assigned, but it is best practice to avoid conversion errors that the compiler may miss. Lastly, every signal is logged before the test completes; thorough logging is not the best practice, but done in this example so that every piece of information can be seen from the terminal. Lastly, the if statement on line 23 requires that the 'value' attribute of the 'dut.sum' and 'dut.carry_out' be used to compare the signals. 'dut' and its internal signals are all class objects of:

```
- <class 'cocotb.handle.ModifiableObject'>
```

You can log the values of this object without calling the value attribute, but the value attribute is necessary to compare the signal to an integer. If the signal values are not as expected, a 'TestFailure' is raised, and debug info is printed. The empty brackets on line 25 are populated by the comma separated values in the 'format' method, a feature of Python.

```
Running on UNKNOWN version UNKNOWN
Running tests with cocotb v1.4.0 from /home/joe/.local/lib/python3.7/site-packages/cocotb
Seeding Python random module with 1620413047
Found test full_adder_cocotb.full_adder_test
Running test 1211 full_adder_test
Starting test: "full_adder_test"
Description: None
=====LOGGING=====
__INPUTS__
a: 1
b: 0
carry_in: 0
__OUTPUTS__
sum: 1
carry_out: 1
=====END LOGGING=====
ated, and in future will return 'ModifiableObject._path'. To get a string representation of the value,

Test Failed: full_adder_test (result was TestFailure)
Traceback (most recent call last):
  File "/home/joe/MS_Project/CocoTB/Full_Adder_Test/full_adder_cocotb.py", line 25, in full_adder_test
    "Observed: sum={} and carry_out={}".format(dut.sum, dut.carry_out))
cocotb.result.TestFailure: Expected: sum=1 and carry_out=0 Observed: sum=1 and carry_out=1
Failed 1 out of 1 tests (0 skipped)
=====
** TEST                                     PASS/FAIL SIM TIME(NS) REAL TIME(S) RATIO(NS/S) **
** full_adder_cocotb.Full_adder_test      FAIL          10,00          0,00    3344,74 **
=====
** ERRORS : 1 **
** SIM TIME : 10,00 NS **
** REAL TIME : 0,01 S **
** SIM / REAL TIME : 1028,04 NS/S **
=====
Shutting down...
```

Figure 7: Full Adder Test Output with Error

Figure 7: Full Adder Test Output with Error displays the terminal output from running this test. With only input signal 'a' having a value of '1', the 'sum' signal should be '1' and the 'carry_out' signal should be '0'. The if statement caught this error and displayed the debug information correctly. The error stems from changing '(a and carry_in)' on line 12 of Figure 2: Full Adder Source File to '(a or carry_in)' on line 12 of Figure 5: Full Adder Source File with Error. Changing the source code back to the correct version results in the output seen below in Figure 8: Full Adder Test Output without Error.


```

Running on UNKNOWN version UNKNOWN
Running tests with cocotb v1.4.0 from /home/joe/.local/lib/python3.7/site-packages/cocotb
Seeding Python random module with 1620413349
Found test_full_adder_cocotb.full_adder_test
Running test: full_adder_test
Starting test: "full_adder_test"
Description: None
=====LOGGING=====
__INPUTS__
a: 1
b: 0
carry_in: 0
__OUTPUTS__
sum: 1
carry_out: 0
=====END LOGGING=====
Test Passed: full_adder_test
Passed 1 tests (0 skipped)
*****
** TEST                                     PASS/FAIL SIM TIME(NS) REAL TIME(S)  RATIO(NS/S) **
** full_adder_cocotb.full_adder_test      PASS           10,00          0,00      3961,38 **
*****

*****
**                                ERRORS : 0                                **
*****
**                                SIM TIME : 10,00 NS                        **
**                                REAL TIME : 0,00 S                          **
**                                SIM / REAL TIME : 2536,78 NS/S                **
*****

Shutting down...

```

Figure 8: Full Adder Test Output without Error

Intermediate – Creating an ARM Test Bench

Python permits data manipulation and object-oriented programming (OOP) techniques to process information and structure your code. OOP enables the programmer to create reusable and understandable code by defining classes that can contain scoped variables and methods (functions that are inside a class). *cocotb* typically takes advantage of OOP by creating a testbench class for the things that need to be done in each test. The testbench class can initialize a set of conditions that would normally have to be done with several lines of code in each test. The testbench class also encapsulates methods that will have direct access to the class variables; replacing the need to pass in arguments to a function.

cocotb can be used to drive simple and complex signals into the simulation while monitoring the output. This section will demonstrate more *cocotb* features while creating a test bench class for the dut. The example dut is an ARM processor with a full test program uploaded to instruction memory. The test program will run, and signal data will be extracted by accessing the submodule hierarchy within the dut. The test bench will drive the appropriate signals to track the number of writes to each register and a similar process will count the number of uses for each instruction bit before plotting the data with a Python library called *Matplotlib*.

Please refer to **Appendix B – ARM Processor Code** for the code to run this example. The Makefile, program.txt, and arm_cocotb.py files should be in the top-level directory with all of the VHDL files in a subdirectory called 'hdl'. The naming conventions must match the contents of the Makefile; any modifications to the file names or locations needs to be reflected in the Makefile.

The architecture of the ARM processor can be seen below in Figure 9: ARM Processor Diagram.

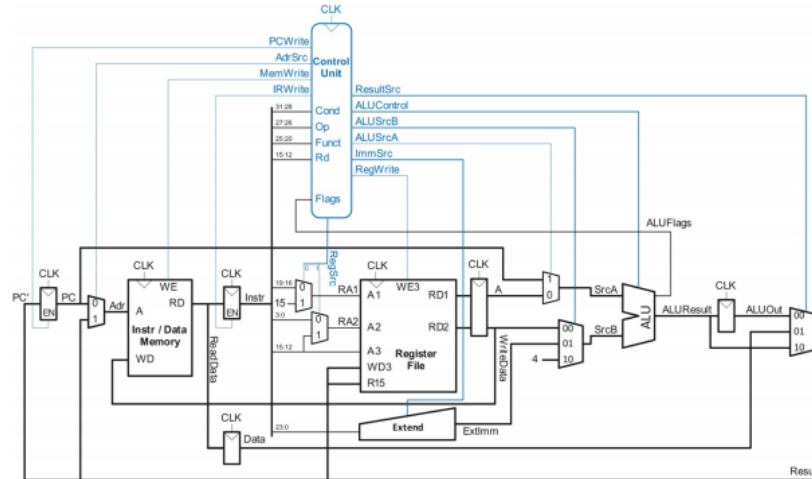


Figure 9: ARM Processor Diagram

Note that the ARM processor has ‘Instr / Data Memory’ as well as a ‘Register File’. The following example will count each time a new value is written to the ‘Register File’ (regfile for short) and count the times each instruction bit is used by accessing ‘Instr / Data Memory’.

First, the necessary libraries and functions need to be imported. In Figure 10: ARM cocotb Class below, lines 1 to 4 import Clock and RisingEdge definitions since the ARM processor operates on a clock signal. Matplotlib’s pyplot will be used to make 2 figures at the end of the test run.

```

arm_cocotb.py | ARM.vhd | Datapath.vhd | Memory.vhd | Register_File.vhd
1  import cocotb
2      from cocotb.clock import Clock
3      from cocotb.triggers import RisingEdge
4  import matplotlib.pyplot as plt
5
6
7  class ArmTestbench(object):
8
9      def __init__(self, dut, debug=False):
10         self.mem_bit_counter = [0] * 32      # Allocate 32 numbers for each instruction bit
11         self.regfile_writes = [0] * 8        # Allocate 8 numbers for each reg in regfile
12
13         self.dut = dut
14         self.mem = dut.inst_datapath.instr_data_memory
15         self.regfile = dut.inst_datapath.i_register_file
16
17         cocotb.fork(Clock(signal=dut.clk, period=10, units='ns').start(start_high=False))
18         self.dut.en_arm <= 1
19
20     @cocotb.coroutine
21     def log_regfile_usage(self):
22         while True:
23             yield RisingEdge(self.regfile.we3)
24             self.regfile_writes[self.regfile.a3.value.integer] += 1
25
26     @cocotb.coroutine
27     def log_instr_bit_usage(self, num_of_instructions):
28         for instr_index in range(num_of_instructions):
29             self.mem.a2 <= instr_index
30             yield RisingEdge(self.dut.clk)
31             self.dut._log.info(instr_index)
32             for bit_index in range(32):
33                 if str(self.mem.rd2.value[bit_index]) != 'U':
34                     self.mem_bit_counter[bit_index] += self.mem.rd2.value[bit_index]

```

Figure 10: ARM cocotb Class

Figure 10: ARM cocotb Class above displays the class structure used to collect the necessary data from the dut. Line 7 declares an `ArmTestBench` class that will have the dut object passed into it. Lines 9 to 18 define the test bench's initialization process: arrays for the data are allocated, pointers to the memory units are created, and the dut clock and enable signal are set. Lines 14 and 15 access submodules of the dut through the names given to them in the VHDL source files. The top-level ARM module instantiates a Datapath component as 'inst_datapath'. The Datapath component instantiates the data memory and register file as 'instr_data_memory' and 'i_register_file', respectively. These names are not the names of the VHDL source files, but the assigned names which can be seen below in Figure 11: GTKWave Hierarchy.

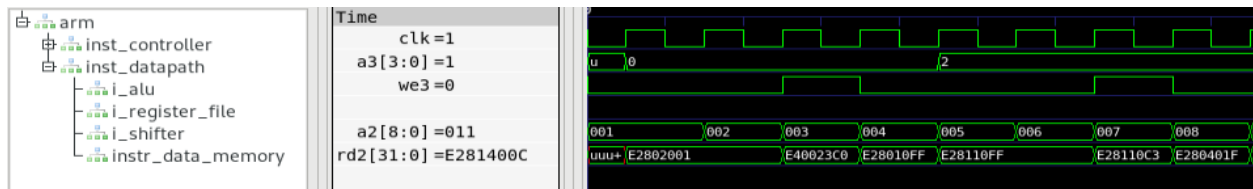


Figure 11: GTKWave Hierarchy

Line 17, of Figure 10: ARM cocotb Class, forks a coroutine called Clock that assigns an oscillating clock signal to the dut's clk signal. The 'fork' method causes the process to run concurrently with the simulation and the current test script. If the coroutine should stop the execution of the Python code (not run concurrently), the coroutine definition should be prefixed with the 'async' keyword. The 'Clock' coroutine runs in tangent with everything else (not asynchronynous). 'self' is a Python convention that points to the object encapsulating that piece of code.

Lines 20 to 34, of Figure 10: ARM cocotb Class, define two coroutines that can capture the desired data during simulation. All functions or methods that depend on the simulation time should be prefixed with the `@cocotb.coroutine` decorator. Coroutines are called from within another coroutine, or a function prefixed with `@cocotb.test()`. Line 21 defines the `log_regfile_usage` method of the `ArmTestBench` class; the method waits for the rising edge of the regfile's write enable signal before recording the current register. The moment 'we3' goes high, the method increments the element that is indexed by the current write address 'a3'. If 'a3' equals 0110 when 'we3' goes high, then the sixth element of the `regfile_writes[]` array will be incremented. The 'a3' signal is converted to an integer type because arrays must be indexed with an integer.

Line 27 defines the `log_instr_bit_usage` method that accepts a `num_of_instructions` argument. The argument controls the total number of instructions used when computing the times each instruction bit was used. Line 28 creates a for loop with `instr_index` that goes from 0 to `num_of_instructions - 1`. The 'a2' address port is assigned `instr_index` before waiting until the next rising edge of the clock. The current instruction number is printed to the terminal. Line 32 begins a for loop that goes through each of the 32 bits in each instruction. Line 33 ensures that no 'U' values are counted as instructions. `self.mem.rd2.value[]` is an array

of binary value types so it can be indexed to isolate each bit by itself. Casting the binary value to a string allows the comparison to 'U'. *cocotb* supports 'X' and 'Z' values as well. As mentioned before, *self* provides direct access to the variables stored within the class. Line 34 then adds `self.mem.rd2.value[bit_index]` to the respective element of `self.mem_bit_counter[]`. Each bit of the read port value is either 1 or 0 which can be added to the counting array element without the need for an if statement.

The testbench initiates the proper conditions and contains the functions needed to extract the data points of interest. Figure 12: ARM *cocotb* Test below displays the *cocotb* test that instantiates the testbench and calls its methods by forking them as a separate process before plotting the extracted data.

```

37  @cocotb.test()
38  def basic_test(dut):
39      tb = ArmTestbench(dut)
40      num_of_tracked_instructions = 80
41
42      cocotb.fork(tb.log_regfile_usage())
43      instr_tracker = cocotb.fork(tb.log_instr_bit_usage(num_of_tracked_instructions))
44      yield instr_tracker.join()
45
46      plt.bar(range(8), tb.regfile_writes)
47      plt.title('Number of Times each Register in the Regfile was Written to')
48      plt.xlabel('Register Number')
49      plt.ylabel('Number of Register Writes')
50      plt.show()
51
52      plt.bar(range(32), tb.mem_bit_counter)
53      plt.title('Bit Usage Breakdown of First {} Instructions'.format(num_of_tracked_instructions))
54      plt.xlabel('Instruction Bit (0 is MSB)')
55      plt.ylabel('Number of Bit Uses')
56      plt.show()
57

```

Figure 12: ARM *cocotb* Test

Line 39 creates a pointer, 'tb', to an object of the 'ArmTestbench' class before initializing a variable for the number of instructions to run. Line 42 forks the 'log_regfile_usage' method to run independently of the Python script. Line 43 creates a reference, 'instr_tracker' to the forked 'log_instr_bit_usage' coroutine. Line 44 waits for the 'instr_tracker' coroutine to finish executing before continuing by yielding on the 'join' method. Since 'log_instr_bit_usage' terminates, the forked coroutine will eventually end. But since 'log_regfile_usage' uses an infinite loop, joining on that process would produce an error. Once the coroutines have been executed, lines 46 to 56 use the pyplot library to plot the data. The syntax is very similar to making plots in MATLAB. The results can be seen below in Figure 13: Number of Regfile Writes and Figure 14: Usage of Each Instruction Bit. When the 'make' command is ran, the figures will pop up in separate windows.

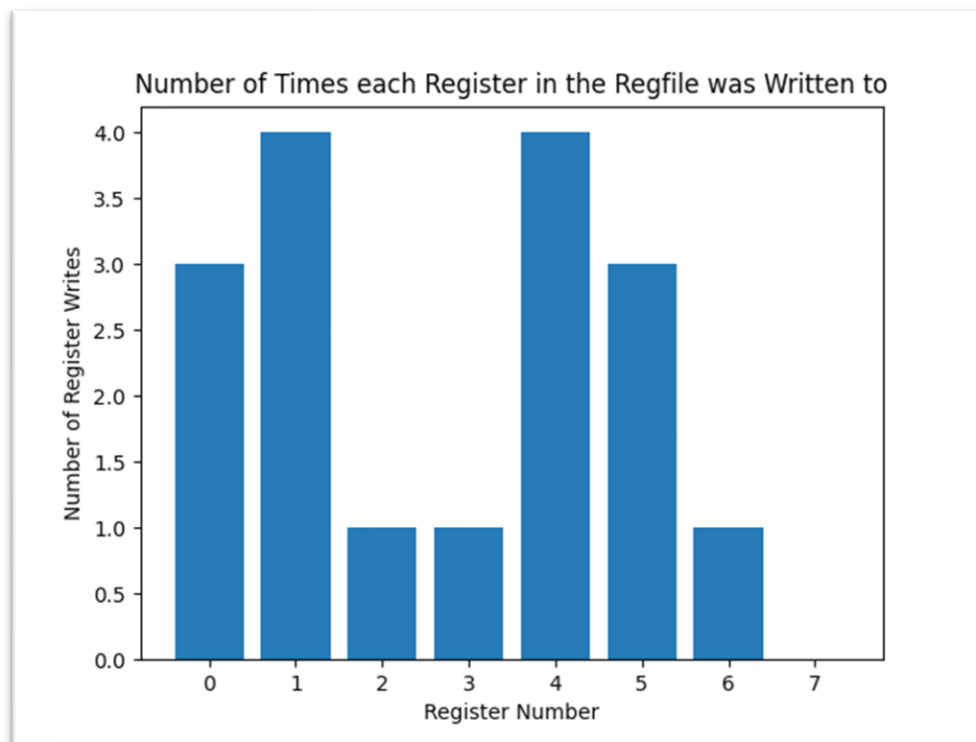


Figure 13: Number of Regfile Writes

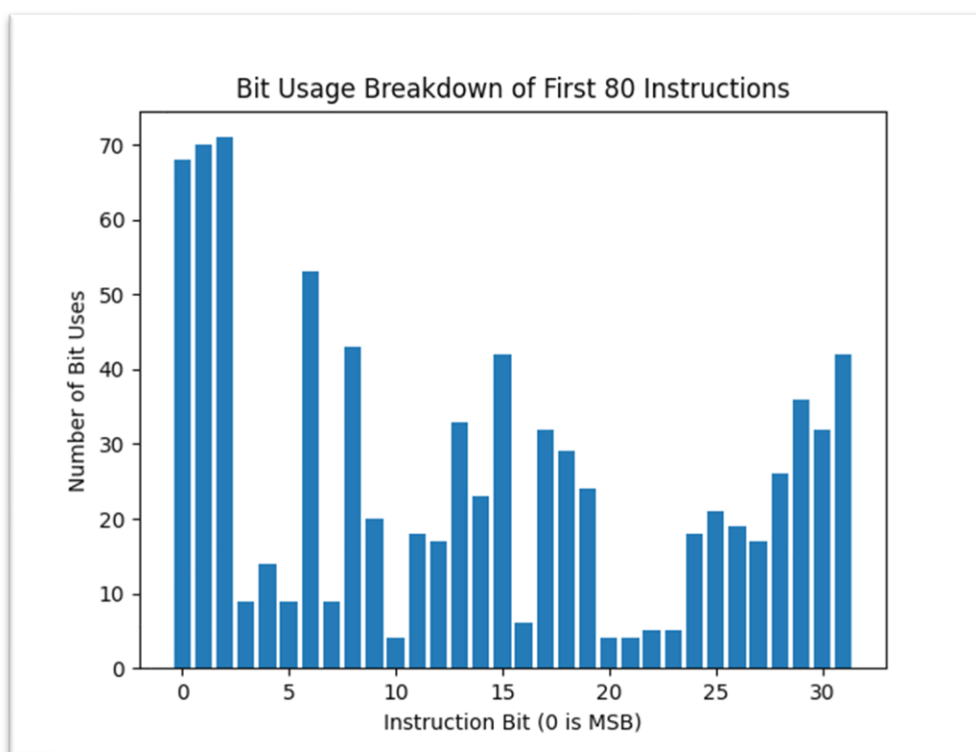
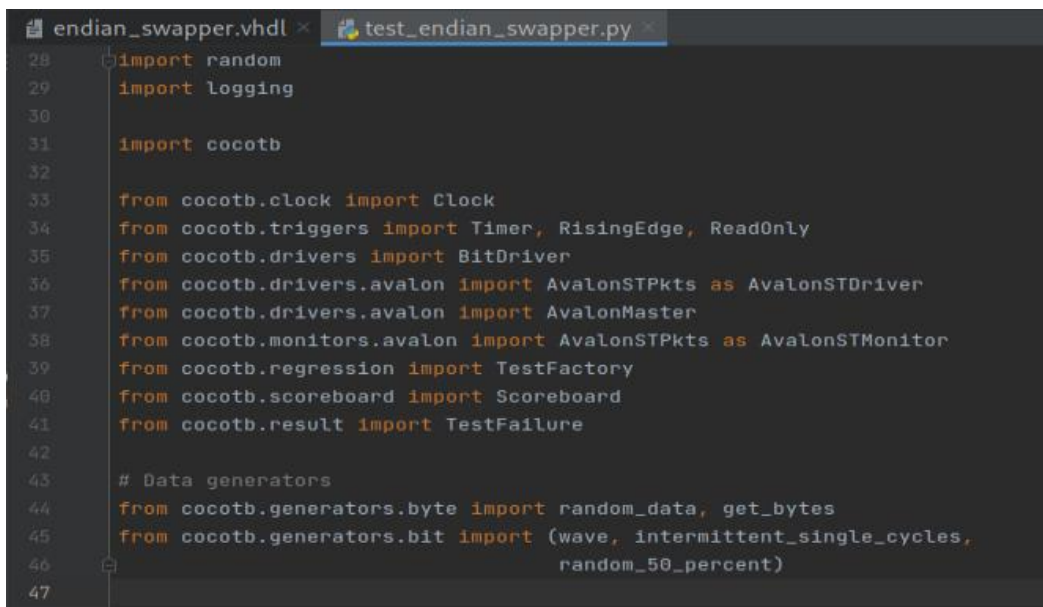


Figure 14: Usage of Each Instruction Bit

Advanced – Endian Swapper with TestFactory and Scoreboard

This section will review the ‘endian_swapper’ example from *cocotb* to highlight some advanced techniques to assist in hardware verification. This example is meant to present advanced techniques available in *cocotb*. This will not be an in-depth review but rather an overview with basic explanations of each section. A GitHub repository containing the code for this example can be found at: <https://github.com/jlucian3/endian-swapper-code>

An endian swapper transforms a little-endian packet into a big-endian packet, and vice versa. To generate the large amounts of data to test such a module, many libraries are imported below in Figure 15: Endian Swapper Libraries. Bit drivers and the Avalon interface offer a way to format the necessary data and supply it to the endian swapper. ‘TestFactory’ receives a series of parameters to generate several *cocotb* tests that will run during simulation. ‘Scoreboard’ compares the expected output to the actual output and keeps track of errors.



```
28 import random
29 import logging
30
31 import cocotb
32
33 from cocotb.clock import Clock
34 from cocotb.triggers import Timer, RisingEdge, ReadOnly
35 from cocotb.drivers import BitDriver
36 from cocotb.drivers.avalon import AvalonSTPkts as AvalonSTDriver
37 from cocotb.drivers.avalon import AvalonMaster
38 from cocotb.monitors.avalon import AvalonSTPkts as AvalonSTMonitor
39 from cocotb.regression import TestFactory
40 from cocotb.scoreboard import Scoreboard
41 from cocotb.result import TestFailure
42
43 # Data generators
44 from cocotb.generators.byte import random_data, get_bytes
45 from cocotb.generators.bit import (wave, intermittent_single_cycles,
46                                   random_50_percent)
47
```

Figure 15: Endian Swapper Libraries

The *cocotb* contributors have developed many different modules that can greatly reduce the work required to begin testing designs. Industry standard interfaces will have much more support than custom interfaces. UVM/OVM also offers libraries that assist in developing test benches with drivers, monitors, scoreboards, and more. However, a significant amount of UVM/OVM content lies within private companies that is not publicly available.

```

endian_swapper.vhdl x test_endian_swapper.py x
67 class EndianSwapperTB(object):
68
69     def __init__(self, dut, debug=False):
70         self.dut = dut
71         self.stream_in = AvalonSTDriver(dut, "stream_in", dut.clk)
72         self.backpressure = BitDriver(self.dut.stream_out_ready, self.dut.clk)
73         self.stream_out = AvalonSTMonitor(dut, "stream_out", dut.clk,
74                                           config={'firstSymbolInHighOrderBits':
75                                                 True})
76
77         self.csr = AvalonMaster(dut, "csr", dut.clk)
78
79         cocotb.fork(stream_out_config_setter(dut, self.stream_out,
80                                             self.stream_in))
81
82         # Create a scoreboard on the stream_out bus
83         self.pkts_sent = 0
84         self.expected_output = []
85         self.scoreboard = Scoreboard(dut)
86         self.scoreboard.add_interface(self.stream_out, self.expected_output)
87
88         # Reconstruct the input transactions from the pins
89         # and send them to our 'model'
90         self.stream_in_recovered = AvalonSTMonitor(dut, "stream_in", dut.clk,
91                                                    callback=self.model)
92
93         # Set verbosity on our various interfaces
94         level = logging.DEBUG if debug else logging.WARNING
95         self.stream_in.log.setLevel(level)
96         self.stream_in_recovered.log.setLevel(level)
97

```

Figure 16: Endian Swapper Testbench

When testing a hardware design, a ‘Driver’ provides the input data that drives the dut functionality. Backpressure refers to another driver that sends a large amount of input data to observe the dut perform under high stress. A ‘Monitor’ extracts the relevant output data from the dut. A ‘Scoreboard’ compares the expected output (separately generated) to the output taken from a ‘Monitor’ to ‘score’ the design. Figure 16: Endian Swapper Testbench above shows how the previously discussed components can be initialized and connected to other components. Note that the testbench also creates a reset coroutine seen below in Figure 17: Endian Swapper Reset.

```

103 @cocotb.coroutine
104 def reset(self, duration=10):
105     self.dut._log.debug("Resetting DUT")
106     self.dut.reset_n <= 0
107     self.stream_in.bus.valid <= 0
108     yield Timer(duration, units='ns')
109     yield RisingEdge(self.dut.clk)
110     self.dut.reset_n <= 1
111     self.dut._log.debug("Out of reset")
112

```

Figure 17: Endian Swapper Reset

```

114 @cocotb.coroutine
115 def run_test(dut, data_in=None, config_coroutine=None, idle_inserter=None,
116             backpressure_inserter=None):
117
118     cocotb.fork(Clock(dut.clk, 5, units='ns').start())
119     tb = EndianSwapperTB(dut)
120
121     yield tb.reset()
122     dut.stream_out_ready <= 1
123
124     # Start off any optional coroutines
125     if config_coroutine is not None:
126         cocotb.fork(config_coroutine(tb.csr))
127     if idle_inserter is not None:
128         tb.stream_in.set_valid_generator(idle_inserter())
129     if backpressure_inserter is not None:
130         tb.backpressure.start(backpressure_inserter())
131
132     # Send in the packets
133     for transaction in data_in():
134         yield tb.stream_in.send(transaction)
135
136     # Wait at least 2 cycles where output ready is low before ending the test
137     for i in range(2):
138         yield RisingEdge(dut.clk)
139         while not dut.stream_out_ready.value:
140             yield RisingEdge(dut.clk)
141
142     pkt_count = yield tb.csr.read(1)
143
144     if pkt_count.integer != tb.pkts_sent:
145         raise TestFailure("DUT recorded %d packets but tb counted %d" % (
146             pkt_count.integer, tb.pkts_sent))
147     else:
148         dut._log.info("DUT correctly counted %d packets" % pkt_count.integer)
149
150     raise tb.scoreboard.result
151

```

Figure 18: Endian Swapper run_test Coroutine

The `run_test()` coroutine, outlined above in Figure 18: Endian Swapper run_test Coroutine, receives configuration arguments that dictate the configuration, idle, and backpressure. Additionally, the transactions are sent into the dut and basic errors are checked with relevant debug info. The scoreboard automatically begins from the testbench initialization, requiring the test to only return the results on line 150.

A `run_test()` function is decorated as a coroutine because it should not directly be called during simulation. The function is passed into a test factory upon initialization, as seen below in Figure 19: Endian Swapper TestFactory Initialization. Once the ‘TestFactory’ settings have been established, the tests can be generated (32 in this case). Each test varies in time and the success depends on the results from the Scoreboard.


```

166     factory = TestFactory(run_test)
167     factory.add_option("data_in",
168                       [random_packet_sizes])
169     factory.add_option("config_coroutine",
170                       [None, randomly_switch_config])
171     factory.add_option("idle_inserter",
172                       [None, wave, intermittent_single_cycles, random_50_percent])
173     factory.add_option("backpressure_inserter",
174                       [None, wave, intermittent_single_cycles, random_50_percent])
175     factory.generate_tests()
176

```

Figure 19: Endian Swapper TestFactory Initialization

Utilizing a prebuilt TestFactory and Scoreboard requires an understanding of their functionality but removes a significant amount of code. For a complex piece of hardware that transfers packets across an interface, the libraries used in this example can be very helpful. *Cocotb* currently supports ad9361, amba, Avalon, opb, and xgmii drivers, a collection of industry standard interfaces for both analog and digital applications. Please refer to the [cocotb documentation](#) to learn more.

Final Remarks

The hardware verification industry primarily uses System Verilog along with UVM techniques to validate their designs. As an alternative, *cocotb* offers many features to verify a broad set of applications, especially in basic applications that do not deal with highly complex modules. Designs can be thoroughly tested while providing relevant debug information by manipulating the HDL simulation and processing the observed data. The developers of *cocotb* are quickly catching up to offer many of the features offered by other hardware verification tools.

The Python ecosystem greatly enhances the usefulness of simulation collateral. Beyond performing traditional verification, *cocotb* enables greater data manipulation, visualization, and processing than System Verilog. Plotting simulation data can reveal concerning trends that would otherwise require extensive work when done with OVM/UVM. Although not demonstrated here, a machine learning model could be integrated to generate tests that expose errors that would otherwise go unnoticed. The scope of such a project would be very large but can be more easily managed since all the code would be written in a common language. The Python language further extends the capabilities of *cocotb* to surpass traditional hardware verification tools.

Additional Examples

The *cocotb* developers have been recently updating their examples as of May 2021. Most of the examples have been simplified in recent updates. The ‘mixed_signal’ and ‘matrix_multiplier’ examples highlight some interesting techniques and styles for writing tests. You can find the most recent examples at: <https://github.com/cocotb/cocotb/tree/master/examples>

Conclusion

cocotb integrates a powerful programming language and the ecosystem of libraries around it with reliable simulation tools to effectively verify new hardware designs in ways that cannot be achieved with traditional methods. The Universal Verification Methodology (UVM) brings automation tools to System Verilog; the current standard for most companies that perform validation. However, System Verilog and the UVM tools associated with it can be very verbose and difficult to learn. *cocotb* offers some of the same features, such as TestFactory and Scoreboard, but with a much simpler interface and a wide range of open-source tools. Using Python as the core language also ties *cocotb*'s growth to the Python community and all its progress as well. *cocotb* continues to grow the community and add important features that developers want as industry leaders begin to explore its use cases. *cocotb* may never fully replace traditional verification tools, but it offers a powerful alternative that allows new Computer Engineers to begin building test benches with little experience; something that cannot always be said for System Verilog or VHDL.

References

1. Liu, S. (2021, April 29). *Most used languages among software developers globally 2020*. Statista. <https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/>.
2. Real Python. (2021, May 8). *Python import: Advanced Techniques and Tips*. Real Python. <https://realpython.com/python-import/#:~:text=In%20Python%2C%20you%20use%20the,for%20structuring%20your%20code%20effectively>.
3. *Decorators in Python*. DataCamp Community. (n.d.). <https://www.datacamp.com/community/tutorials/decorators-python>.
4. *When to use yield instead of return in Python?* GeeksforGeeks. (2019, December 6). <https://www.geeksforgeeks.org/use-yield-keyword-instead-return-keyword-python/>.
5. *Triggers*. Triggers - cocotb 1.6.0.dev0+gec99a877.d20210503 documentation. (n.d.). <https://docs.cocotb.org/en/stable/triggers.html>.
6. Python raise Keyword. (n.d.). https://www.w3schools.com/python/ref_keyword_raise.asp.

Appendix A – Full Adder Code

The following 3 files should be in the same directory. Navigate to this directory within a terminal and run the make command.

Makefile

```
# Specify the current working directory (CWD)
CWD=$(shell pwd)

# Set the default hardware description language and simulator
TOPLEVEL_LANG ?=vhdl
SIM ?= ghdl

# Communicate what hdl files cocotb should be using
VHDL_SOURCES =$(CWD)/full_adder.vhd

# Establishes the name of the top level dut and the cocotb module file
TOPLEVEL = full_adder
MODULE := $(TOPLEVEL)_cocotb

# Tells the simulator to produce a waveform as well
SIM_ARGS += --vcd=waveform.vcd

# Includes the default Makefile defined within cocotb
include $(shell cocotb-config --makefiles)/Makefile.sim
```

full_adder.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Full_Adder is
    Port ( a, b, carry_in: in std_logic;
           sum, carry_out: out std_logic);
end Full_Adder;

architecture Behavioral of Full_Adder is
begin
    sum <= a xor b xor carry_in;
    carry_out <= (a and b) or (a and carry_in) or (b and carry_in);
end Behavioral;
```

full_adder_cocotb.py

```
import cocotb
from cocotb.triggers import Timer
from cocotb.result import TestFailure

@cocotb.test()
def full_adder_test(dut):
    dut.a <= 1
    dut.b <= 0
    dut.carry_in <= 0
    yield Timer(10, units='ns')
    dut._log.info("=====LOGGING===== " % dut.sum)

    dut._log.info("__INPUTS__")
    dut._log.info("a: %d" % dut.a)
    dut._log.info("b: %d" % dut.b)
    dut._log.info("carry_in: %d" % dut.carry_in)

    dut._log.info("__OUTPUTS__")
    dut._log.info("sum: %d" % dut.sum)
    dut._log.info("carry_out: %d" % dut.carry_out)
    dut._log.info("=====END LOGGING=====")

    if dut.sum.value != 1 or dut.carry_out.value != 0:
        raise TestFailure("Expected: sum=1 and carry_out=0 " +
                          "Observed: sum={} and carry_out={}".format(dut.sum, dut.carry_out))
```

Appendix B – ARM Processor Code

The ‘Makefile’, ‘arm_cocotb.py’, and ‘program.txt’ should be located in the top level directory. The following .vhd files should be located in a subdirectory named ‘hdl’.

Makefile

```
CWD=$(shell pwd)

TOPLEVEL_LANG ?=vhdl
SIM ?= ghdl

VHDL_SOURCES =$(CWD)/hdl/*.vhd

TOPLEVEL = arm
MODULE := $(TOPLEVEL)_cocotb
COCOTB_HDL_TIMEPRECISION=1us

CUSTOM_SIM_DEPS=$(CWD)/Makefile

ifeq ($(SIM),questa)
    SIM_ARGS=-t 1ps
endif

ifeq ($(SIM),$(filter $(SIM),ius xcelium))
    SIM_ARGS += -v93
endif

SIM_ARGS += --vcd=waveform.vcd

include $(shell cocotb-config --makefiles)/Makefile.sim

# list all required Python files here
sim: $(MODULE).py
```

arm_cocotb.py

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import RisingEdge
import matplotlib.pyplot as plt

class ArmTestbench(object):
```

```

def __init__(self, dut, debug=False):
    self.mem_bit_counter = [0] * 32      # Allocate 32 numbers for each instruction bit
    self.regfile_writes = [0] * 8       # Allocate 8 numbers for each reg in regfile

    self.dut = dut
    self.mem = dut.inst_datapath.instr_data_memory
    self.regfile = dut.inst_datapath.i_register_file

    cocotb.fork(Clock(signal=dut.clk, period=10, units='ns').start(start_high=False))
    self.dut.en_arm <= 1

@cocotb.coroutine
def log_regfile_usage(self):
    while True:
        yield RisingEdge(self.regfile.we3)
        self.regfile_writes[self.regfile.a3.value.integer] += 1

@cocotb.coroutine
def log_instr_bit_usage(self, num_of_instructions):
    for instr_index in range(num_of_instructions):
        self.mem.a2 <= instr_index
        yield RisingEdge(self.dut.clk)
        self.dut._log.info(instr_index)
        for bit_index in range(32):
            if str(self.mem.rd2.value[bit_index]) != 'U':
                self.mem_bit_counter[bit_index] += self.mem.rd2.value[bit_index]

@cocotb.test()
def basic_test(dut):
    tb = ArmTestbench(dut)
    num_of_tracked_instructions = 80

    cocotb.fork(tb.log_regfile_usage())
    instr_tracker = cocotb.fork(tb.log_instr_bit_usage(num_of_tracked_instructions))
    yield instr_tracker.join()

    plt.bar(range(8), tb.regfile_writes)
    plt.title('Number of Times each Register in the Regfile was Written to')
    plt.xlabel('Register Number')
    plt.ylabel('Number of Register Writes')
    plt.show()

    plt.bar(range(32), tb.mem_bit_counter)

```

```
plt.title('Bit Usage Breakdown of First {} Instructions'.format(num_of_tracked_instructions))
plt.xlabel('Instruction Bit (0 is MSB)')
plt.ylabel('Number of Bit Uses')
plt.show()
```

hdl – Subdirectory

ALU_2018.vhd

```
-----
-- Company:      Binghamton University
-- Engineer:     Carl Betcher
--
-- Create Date:   17:43:33 11/16/2016
-- Design Name:   ARM Processor ALU
-- Module Name:   ALU - Behavioral
-- Project Name:   ARM Processor (Multi-Cycle)
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ALU is
    Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
          B : in  STD_LOGIC_VECTOR (31 downto 0);
          shifter_co : STD_LOGIC; -- added for shifting
          ALUSrcB : in STD_LOGIC_VECTOR (2 downto 0); -- added for shifting
          ALUControl : in STD_LOGIC_VECTOR (2 downto 0);
          Result : out STD_LOGIC_VECTOR (31 downto 0);
          ALUFlags : out STD_LOGIC_VECTOR (3 downto 0));
end ALU;

architecture Behavioral of ALU is

    signal Asig : unsigned (31 downto 0);
```



```

signal Bsig : unsigned (31 downto 0);
signal notBsig : unsigned (31 downto 0);
signal Sum : unsigned (32 downto 0);
signal SumMux : unsigned (31 downto 0);
signal ResultSig : unsigned (31 downto 0);
signal CarryIn : unsigned (32 downto 0);

signal N, Z, C, V : std_logic;

-- signal AdderA, AdderB : unsigned (32 downto 0);

begin

    -- A and B inputs of ALU are unsigned values
    Asig <= unsigned(A);
    Bsig <= unsigned(B);
    notBsig <= not Bsig;

    -- Selects B for addition, not B for subtraction
    SumMux <= Bsig when ALUControl(0) = '0' else notBsig;

    -- 33-bit sum of Asig and SumMux and CarryIn
    CarryIn <= (0 => ALUControl(0), others => '0');
    Sum <= resize(Asig,33) + resize(SumMux,33) + CarryIn;

    -- Selects operation of the ALU
    with ALUControl select
    ResultSig <= Sum(31 downto 0) when "000" | "001", -- ADD, SUB, CMP
                Asig and Bsig when "010",           -- AND
                Asig or Bsig when "011",            -- ORR
                Asig xor Bsig when "110",           -- EOR
                SumMux when "100" | "101",          -- MOV, MVN
                Asig and notBsig when "111",        -- BIC
                Sum(31 downto 0) when others;

    -- 32-bit output of the ALU
    Result <= std_logic_vector(ResultSig(31 downto 0));

    -- Determine the values of the flags
    N <= ResultSig(31);
    Z <= '1' when ResultSig = 0 else '0';
    C <= (Sum(32) and not ALUControl(1) and not ALUControl(2)) -- Carry for ADD/SUB
        or (shifter_co and ALUControl(2) and ALUSrcB(2) -- Carry for Shifter
            and not ALUSrcB(1) and not ALUSrcB(0)); --
    V <= (ALUControl(0) xnor (A(31) xor B(31)))and

```

```

        (A(31) xor Sum(31)) and not ALUControl(1);

    -- Output the flags
    ALUFlags <= N & Z & C & V;

end Behavioral;

```

ARM.vhd

```

-----
-- Company:          Binghamton University
-- Engineer(s):      Carl Betcher
--
-- Create Date:      11/18/2017
-- Design Name:      ARM Processor
-- Module Name:      ARM - Behavioral
-- Project Name:     ARM Multi-Cycle Processor
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ARM is -- multi-cycle ARM processor
    generic ( addr_size : positive := 9 );
    port(clk, reset, en_ARM:   in  STD_LOGIC;
          SWITCH              : in  STD_LOGIC_VECTOR(7 downto 0);
          PCOut               : out STD_LOGIC_VECTOR(7 downto 0);
          InstrOut            : out STD_LOGIC_VECTOR(27 downto 20);
          ReadDataOut         : out STD_LOGIC_VECTOR(7 downto 0)
    );
end ARM;

architecture Behavioral of ARM is

    COMPONENT controller
    PORT(

```

```

    clk : IN std_logic;
    reset : IN std_logic;
    en_ARM : IN std_logic;
    cond : IN std_logic_vector(3 downto 0);
    ALUFlags : IN std_logic_vector(3 downto 0);
    Op : IN std_logic_vector(1 downto 0);
    Funct : IN std_logic_vector(5 downto 0);
    Rd : IN std_logic_vector(3 downto 0);
    Instr: IN STD_LOGIC_VECTOR (11 downto 4);
    PCWrite : OUT std_logic;
    RegWrite : OUT std_logic;
    MemWrite : OUT std_logic;
    IRWrite : OUT std_logic;
    AdrSrc : OUT std_logic;
    ALUSrcA : OUT std_logic;
    decode_state : OUT std_logic;
    ResultSrc : OUT std_logic_vector(1 downto 0);
    ALUSrcB : OUT std_logic_vector(2 downto 0);
    ImmSrc : OUT std_logic_vector(1 downto 0);
    RegSrc : OUT std_logic_vector(1 downto 0);
    ALUControl : OUT std_logic_vector(2 downto 0)
);

```

```
END COMPONENT;
```

```
COMPONENT datapath
```

```
GENERIC ( addr_size : positive := 9 );
```

```
PORT(
```

```

    clk : IN std_logic;
    reset : IN std_logic;
    en_ARM : IN std_logic;
    decode_state : IN std_logic;
    SWITCH : in STD_LOGIC_VECTOR(7 downto 0);
    RegSrc : IN std_logic_vector(1 downto 0);
    ImmSrc : IN std_logic_vector(1 downto 0);
    ResultSrc : IN std_logic_vector(1 downto 0);
    ALUSrcB : IN std_logic_vector(2 downto 0);
    ALUControl : IN std_logic_vector(2 downto 0);
    MemWrite : IN std_logic;
    IRWrite : IN std_logic;
    RegWrite : IN std_logic;
    ALUSrcA : IN std_logic;
    PCWrite : IN std_logic;
    AdrSrc : IN std_logic;
    ALUFlags : OUT std_logic_vector(3 downto 0);

```

```
PCOut : out STD_LOGIC_VECTOR(7 downto 0);
```

```
InstrOut : out STD_LOGIC_VECTOR(31 downto 4);
ReadDataOut : out STD_LOGIC_VECTOR(7 downto 0)
);
END COMPONENT;
```

```
-- Signals needed to make connections between the datapath and controller
```

```
signal ALUFlags : std_logic_vector(3 downto 0);
signal RegSrc : std_logic_vector(1 downto 0);
signal RegWrite : std_logic;
signal ImmSrc : std_logic_vector(1 downto 0);
signal ALUSrcA : std_logic;
signal ALUSrcB : std_logic_vector(2 downto 0);
signal ALUControl : std_logic_vector(2 downto 0);
signal ResultSrc : std_logic_vector(1 downto 0);
signal MemWriteSig : std_logic;
signal PCWrite : std_logic;
signal AdrSrc : std_logic;
signal IRWrite : std_logic;
signal Instr : std_logic_vector(31 downto 4);
```

```
signal decode_state : std_logic;
```

```
begin
```

```
InstrOut <= Instr(27 downto 20);
```

```
-- Instantiate the Controller for the ARM Processor
```

```
Inst_controller: controller PORT MAP(
    clk => clk,
    reset => reset,
    en_ARM => en_ARM,
    cond => Instr(31 downto 28),
    ALUFlags => ALUFlags,
    Op => Instr(27 downto 26),
    Funct => Instr(25 downto 20),
    Rd => Instr(15 downto 12),
    Instr => Instr(11 downto 4),
    PCWrite => PCWrite,
    RegWrite => RegWrite,
    MemWrite => MemWriteSig,
    IRWrite => IRWrite,
    AdrSrc => AdrSrc,
    decode_state => decode_state,
    ALUSrcA => ALUSrcA,
    ResultSrc => ResultSrc,
```

```

        ALUSrcB => ALUSrcB,
        ImmSrc => ImmSrc,
        RegSrc => RegSrc,
        ALUControl => ALUControl
    );

    -- Instantiate the Datapath for the ARM Processor
    Inst_datapath: datapath
    GENERIC MAP (addr_size)
    PORT MAP(
        clk => clk,
        reset => reset,
        en_ARM => en_ARM,
        decode_state => decode_state,
        SWITCH => SWITCH,
        RegSrc => RegSrc,
        ImmSrc => ImmSrc,
        ResultSrc => ResultSrc,
        ALUSrcB => ALUSrcB,
        ALUControl => ALUControl,
        MemWrite => MemWriteSig,
        IRWrite => IRWrite,
        RegWrite => RegWrite,
        ALUSrcA => ALUSrcA,
        PCWrite => PCWrite,
        AdrSrc => AdrSrc,
        ALUFlags => ALUFlags,
        PCOut => PCOut,
        InstrOut => Instr,
        ReadDataOut => ReadDataOut
    );

end Behavioral;

```

Cond_Logic.vhd

```

-----
-- Company:          Binghamton University
-- Engineer:         Carl Betcher
--
-- Create Date:      22:32:43 11/16/2016
-- Design Name:     ARM Processor Decoder
-- Module Name:      Cond_Logic - Behavioral
-- Project Name:     ARM Processor (Single Cycle)
-- Target Devices:

```

```

-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Cond_Logic is
    Port ( clk, reset : in  STD_LOGIC;
          Cond, ALUFlags : in  STD_LOGIC_VECTOR (3 downto 0);
          FlagW : in  STD_LOGIC_VECTOR (2 downto 0); -- expanded FlagW to 3 bits
          PCS, NextPC, RegW, MemW : in  STD_LOGIC;
          NoWrite : in std_logic; -- added for CMP instruction
          PCWrite, RegWrite, MemWrite : out STD_LOGIC);
end Cond_Logic;

architecture Behavioral of Cond_Logic is

    signal FlagWrite : std_logic_vector(2 downto 0); -- expanded FlagWrite to 3 bits
    signal Flags : std_logic_vector(3 downto 0);
    signal CondEx, CondExReg : std_logic;
    signal N, Z, C, V : STD_LOGIC;

begin

    -- Conditional Logic checks if instruction should execute
    -- (if not, force PCWrite, RegWrite, and MemWrite to '0')
    -- and possibly updates the Status Register (Flags(3:0))

    -- Register the ALU Flags
    -- FlagW(2) = Enable setting of N and Z flags
    -- FlagW(1) = Enable setting of C flag
    -- FlagW(0) = Enable setting of V flag
    FlagWrite <= FlagW AND (CondEx & CondEx & CondEx);

    process(clk) -- Status Register for N and Z Flags
    begin
        if rising_edge(clk) then
            if reset = '1' then

```

```

        Flags(3 downto 2) <= (others => '0');
    elsif FlagWrite(2) = '1' then
        Flags(3 downto 2) <= ALUFlags(3 downto 2);
    end if;
end if;
end process;

process(clk) -- Status Register for C Flag
begin
    if rising_edge(clk) then
        if reset = '1' then
            Flags(1) <= '0';
        elsif FlagWrite(1) = '1' then
            Flags(1) <= ALUFlags(1);
        end if;
    end if;
end process;

process(clk) -- Status Register for V Flag
begin
    if rising_edge(clk) then
        if reset = '1' then
            Flags(0) <= '0';
        elsif FlagWrite(0) = '1' then
            Flags(0) <= ALUFlags(0);
        end if;
    end if;
end process;

-- Condition Checking Logic
N <= Flags(3); Z <= Flags(2); C <= Flags(1); V <= Flags(0);
process(Cond, N, Z, C, V)
begin
    case Cond is
        when "0000" => CondEx <= Z; -- EQ
        when "0001" => CondEx <= not Z; -- NE
        when "0010" => CondEx <= C; -- CS/HS
        when "0011" => CondEx <= not C; -- CC/LO
        when "0100" => CondEx <= N; -- MI
        when "0101" => CondEx <= not N; -- PL
        when "0110" => CondEx <= V; -- VS
        when "0111" => CondEx <= not V; -- VC
        when "1000" => CondEx <= not Z and C; -- HI
        when "1001" => CondEx <= Z or not C; -- LS
        when "1010" => CondEx <= not (N xor V); -- GE
    end case;
end process;

```



```

        when "1011" => CondEx <= N xor V;                -- LT
        when "1100" => CondEx <= not Z and not(N xor V);-- GT
        when "1101" => CondEx <= Z or (N xor V);         -- LE
        when "1110" => CondEx <= '1';                  -- AL (or none)
        when others => CondEx <= '0';

    end case;
end process;

-- Register the CondEx signal
process(clk)
begin
    if rising_edge(clk) then
        CondExReg <= CondEx;
    end if;
end process;

-- Only allow the architectural state of the ARM processor to change
-- if the instruction is to be executed
PCWrite <= (PCS and CondExReg) or NextPC;
RegWrite <= RegW and CondExReg
                                and not NoWrite; -- added for CMP instruction
                                                -- to not write back to the RF

MemWrite <= MemW and CondExReg;

-- Still works with this code that does not use the CondExReg register above
-- PCWrite <= (PCS and CondEx) or NextPC;
-- RegWrite <= RegW and CondEx
--                                and not NoWrite; -- added for CMP instruction
--                                -- to not write back to the RF
-- MemWrite <= MemW and CondEx;

end Behavioral;

```

Controller.vhd

```

-----
-- Company:      Binghamton University
-- Engineer(s):   Carl Betcher
--
-- Create Date:   11/18/2017
-- Design Name:    ARM Processor Controller
-- Module Name:    Controller - Behavioral
-- Project Name:   ARM Multi-Cycle Processor
-- Target Devices:
-- Tool versions:

```

```

-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity controller is -- single cycle control decoder
    port(clk, reset, en_ARM: in STD_LOGIC;
        cond      : in  STD_LOGIC_VECTOR(3 downto 0);
        ALUFlags: in  STD_LOGIC_VECTOR(3 downto 0);
        Op       : in  STD_LOGIC_VECTOR(1 downto 0);
        Funct    : in  STD_LOGIC_VECTOR(5 downto 0);
        Rd       : in  STD_LOGIC_VECTOR(3 downto 0);
        Instr    : in  STD_LOGIC_VECTOR(11 downto 4);
        PCWrite, RegWrite, MemWrite, IRWrite, AdrSrc,
            ALUSrcA, decode_state : out STD_LOGIC;
        ResultSrc, ImmSrc, RegSrc : out STD_LOGIC_VECTOR(1 downto 0);
        ALUControl, ALUSrcB      : out STD_LOGIC_VECTOR(2 downto 0)
    );
end;

architecture Behavioral of Controller is

    COMPONENT Decoder
        PORT ( Op      : in  STD_LOGIC_VECTOR (1 downto 0);
            Funct : in  STD_LOGIC_VECTOR (5 downto 0);
            Rd     : in  STD_LOGIC_VECTOR (3 downto 0);
            Instr  : in  STD_LOGIC_VECTOR (11 downto 4);
            PCS, NextPC, RegW, MemW, decode_state : out  STD_LOGIC;
            NoWrite : out std_logic; -- added for CMP instruction
            IRWrite, AdrSrc, ALUSrcA : out  STD_LOGIC;
            ResultSrc, ImmSrc : out  STD_LOGIC_VECTOR (1 downto 0);
            FlagW : out  STD_LOGIC_VECTOR (2 downto 0); -- expanded FlagW to 3 bits
            RegSrc : out  STD_LOGIC_VECTOR (1 downto 0);
            ALUControl, ALUSrcB : out  STD_LOGIC_VECTOR (2 downto 0);
            clk, reset, en_ARM : in STD_LOGIC
        );
    END COMPONENT;

    COMPONENT Cond_Logic

```

```

PORT ( clk, reset : in  STD_LOGIC;
      Cond, ALUFlags : in  STD_LOGIC_VECTOR (3 downto 0);
      FlagW : in  STD_LOGIC_VECTOR (2 downto 0); -- expanded FlagW to 3 bits
      PCS, NextPC, RegW, MemW : in  STD_LOGIC;
      NoWrite : in  std_logic; -- added for CMP instruction
      PCWrite, RegWrite, MemWrite : out  STD_LOGIC
    );
END COMPONENT;

```

```

signal FlagW : std_logic_vector(2 downto 0); -- expanded FlagW to 3 bits
signal NextPC : std_logic;
signal PCS : std_logic;
signal RegW : std_logic;
signal MemW : std_logic;
signal NoWrite : std_logic; -- added for CMP instruction

```

```

begin

```

```

-- Instantiate the Decoder Function of the Controller

```

```

i_Decoder: Decoder PORT MAP(
  Op => Op,
  Funct => Funct,
  Rd => Rd,
  Instr => Instr,
  PCS => PCS,
  NextPC => NextPC,
  RegW => RegW,
  MemW => MemW,
  NoWrite => NoWrite,
  IRWrite => IRWrite,
  AdrSrc => AdrSrc,
  ALUSrcA => ALUSrcA,
  ResultSrc => ResultSrc,
  ALUSrcB => ALUSrcB,
  ImmSrc => ImmSrc,
  FlagW => FlagW,
  RegSrc => RegSrc,
  ALUControl => ALUControl,
  clk => clk,
  reset => reset,
  en_ARM => en_ARM,
  decode_state => decode_state
);

```

```

-- Instantiate the Conditional Logic Function of the Controller

```

```

i_Cond_Logic: Cond_Logic PORT MAP(
    clk =>      clk,
    reset =>    reset,
    Cond =>      Cond,
    ALUFlags => ALUFlags,
    FlagW =>    FlagW,
    PCS =>      PCS,
    NextPC =>   NextPC,
    RegW =>      RegW,
    MemW =>      MemW,
    NoWrite =>  NoWrite,
    PCWrite =>  PCWrite,
    RegWrite => RegWrite,
    MemWrite => MemWrite
);

end Behavioral;

```

debounce.vhd

```

-----
-- Company:      Binghamton University
-- Engineer:     Carl Betcher
--
-- Create Date:   09/24/2014
-- Design Name:
-- Module Name:   debounce - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:    A debounce circuit that waits a DELAY before accepting
--                the change in level of the input signal, thus filtering out
--                any instabilities of the signal level caused by switch bounce
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity debounce is

```

```

    Generic ( DELAY : integer := 640000 -- DELAY = 20 mS / clk_period
              );
Port ( clk : in  STD_LOGIC;
      sig_in : in  STD_LOGIC;
      sig_out : out STD_LOGIC
      );
end debounce;

architecture Behavioral of debounce is

    type state_type is (S1, S2, S3, S4, S5);
    signal state, next_state : state_type;

    signal timer : unsigned(21 downto 0); -- delays up to 4,194,303
    signal ld_timer : std_logic;
    signal en_timer : std_logic;
    signal timer_eq_0 : std_logic;

begin

    process(clk)
    begin
        if rising_edge(clk) then
            if ld_timer = '1' then
                timer <= to_unsigned(DELAY,timer'length);
            elsif en_timer = '1' then
                timer <= timer - 1;
            else
                timer <= timer;
            end if;
        end if;
    end process;

    process(timer)
    begin
        if timer = 0 then
            timer_eq_0 <= '1';
        else
            timer_eq_0 <= '0';
        end if;
    end process;

    process(clk)
    begin
        if rising_edge(clk) then

```

```

        state <= next_state;
    end if;
end process;

process(state,sig_in,timer_eq_0)
begin
    ld_timer <= '0';
    en_timer <= '0';
    sig_out <= '0';
    case(state) is
        when S1 =>
            ld_timer <= '1';
            if sig_in = '1' then next_state <= S2; else next_state <= S1; end if;
        when S2 =>
            en_timer <= '1';
            if sig_in = '0' then next_state <= S1;
            elsif timer_eq_0 = '1' then next_state <= S3; else next_state <= S2;    end if;
        when S3 =>
            sig_out <= '1';
            next_state <= S4;
        when S4 =>
            ld_timer <= '1';
            if sig_in = '0' then next_state <= S5; else next_state <= S4; end if;
        when S5 =>
            en_timer <= '1';
            if sig_in = '1' then next_state <= S4;
            elsif timer_eq_0 = '1' then next_state <= S1; else next_state <= S5;    end if;
    end case;

    end process;

end Behavioral;

```

Decoder.vhd

```

-----
-- Company:      Binghamton University
-- Engineer(s):   Carl Betcher
--
-- Create Date:   11/18/2017
-- Design Name:   ARM Processor Decoder
-- Module Name:    Decoder - Behavioral
-- Project Name:   ARM Multi-Cycle Processor
-- Target Devices:
-- Tool versions:

```

```

-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Decoder is
    Port ( Op      : in  STD_LOGIC_VECTOR (1 downto 0);
          Funct    : in  STD_LOGIC_VECTOR (5 downto 0);
          Rd       : in  STD_LOGIC_VECTOR (3 downto 0);
          Instr    : in  STD_LOGIC_VECTOR (11 downto 4);
          PCS, NextPC, RegW, MemW, decode_state : out STD_LOGIC;
          NoWrite  : out std_logic; -- added for CMP instruction
          IRWrite, AdrSrc, ALUSrcA : out  STD_LOGIC;
          ResultSrc, ImmSrc : out  STD_LOGIC_VECTOR (1 downto 0);
          FlagW : out  STD_LOGIC_VECTOR (2 downto 0); -- expanded FlagW to 3 bits
          RegSrc : out  STD_LOGIC_VECTOR (1 downto 0);
          ALUSrcB, ALUControl : out  STD_LOGIC_VECTOR (2 downto 0);
          clk, reset, en_ARM : in STD_LOGIC
    );
end Decoder;

architecture Behavioral of Decoder is

    alias cmd : std_logic_vector(3 downto 0)
        is Funct(4 downto 1); -- DP Instruction Command
        -- ADD: cmd="0100"    CMP="1010"
        -- SUB: cmd="0010"    EOR="0001"
        -- AND: cmd="0000"    MOV="1101"
        -- ORR: cmd="1100"    MVN="1111"

    alias I  : std_logic is Funct(5); -- I-bit = '0' --> Src2 is a register
        --      = '1' --> Src2 is an immediate

    alias S  : std_logic is Funct(0); -- S-bit = '1' --> set condition flags

    alias L  : std_logic is Funct(0); -- L-bit = '0' --> mem op is STR
        --      = '1' --> mem op is LDR

    signal ALUDecOp : std_logic_vector(5 downto 0);
    signal FlagW_Dec : std_logic_vector(6 downto 0);
    signal MOV_Dec : std_logic;

```



```

signal RegWsig : std_logic;
signal Branch : std_logic;
signal ALUOp : std_logic;

type state_type is (S0_Fetch, S1_Decode, S2_MemAdr, S3_MemRead, S4_MemWB,
                    S5_MemWrite, S6_ExecuterR, S7_ExecuteI, S8_ALUWB, S9_Branch);
signal state : state_type := S0_Fetch;
signal next_state : state_type;

begin

    -- Output RegW
    RegW <= RegWsig;

    -- PC LOGIC
    -- PCS = 1 if PC is written by an instruction or branch (B)
    PCS <= '1' when (Rd = x"F" and RegWsig = '1') or Branch = '1' else '0';

    -- MAIN DECODER FSM
    -- State Register
    process (clk)
    begin
        if rising_edge(clk) then
            if reset = '1' then
                state <= S0_Fetch;
            else
                state <= next_state;
            end if;
        end if;
    end process;

    -- Next State and Output Logic
    process (state, Op, I, L, en_ARM)
    begin
        -- default outputs
        NextPC <= '0';
        RegWsig <= '0';
        MemW <= '0';
        IRWrite <= '0';
        AdrSrc <= '0';
        ALUSrcA <= '0';
        ResultSrc <= "00";
        ALUSrcB <= "000";
        ALUOp <= '0';
    end process;

```

```

Branch <= '0';
decode_state <= '0';
-- default next_state is current state
next_state <= state;

case state is
  when S0_Fetch =>
    NextPC <= '1';
    IRWrite <= '1';
    ALUSrcA <= '1';
    ResultSrc <= "10";
    ALUSrcB <= "010";
    next_state <= S1_Decode;
  when S1_Decode =>
    ALUSrcA <= '1';
    ResultSrc <= "10";
    ALUSrcB <= "010";
    decode_state <= '1';
    -- FSM will stop in Decode state when en_ARM = '0'
    if en_ARM = '1' then
      if Op = "01" then
        next_state <= S2_MemAdr;
      elsif Op = "00" and I = '0' then
        next_state <= S6_ExecuteR;
      elsif Op = "00" and I = '1' then
        next_state <= S7_ExecuteI;
      else -- Op = "10"
        next_state <= S9_Branch;
      end if;
    end if;
  when S2_MemAdr =>
    ALUSrcB <= "001";
    if L = '1' then
      next_state <= S3_MemRead;
    else
      next_state <= S5_MemWrite;
    end if;
  when S3_MemRead =>
    AdrSrc <= '1';
    next_state <= S4_MemWB;
  when S4_MemWB =>
    RegWsig <= '1';
    ResultSrc <= "01";
    next_state <= S0_Fetch;
  when S5_MemWrite =>

```

```

        MemW <= '1';
        AdrSrc <= '1';
        next_state <= S0_Fetch;
    when S6_ExecuteR =>
        ALUSrcB <= "100";
        ALUOp <= '1';
        next_state <= S8_ALUWB;
    when S7_ExecuteI =>
--        ALUSrcB <= "001";
        ALUSrcB <= "101"; -- Added Rotated Immediate
        ALUOp <= '1';
        next_state <= S8_ALUWB;
    when S8_ALUWB =>
        RegWsig <= '1';
        next_state <= S0_Fetch;
    when S9_Branch =>
        ResultSrc <= "10";
        ALUSrcB <= "001";
        Branch <= '1';
        next_state <= S0_Fetch;
    end case;
end process;

-- ALU DECODER

-- ALUControl sets the operation to be performed by ALU
ALUDecOp <= ALUOp & cmd & S;
with ALUDecOp select
ALUControl <=  "000" when "101000" | "101001", -- ADD
                "001" when "100100" | "100101"  -- SUB
                | "110101", -- CMP (added instruction)
                "010" when "100000" | "100001", -- AND
                "011" when "111000" | "111001", -- ORR
                "110" when "100010" | "100011", -- EOR (added instruction)
                "100" when "111010" | "111011", -- MOV (added instruction)
                "101" when "111110" | "111111", -- MVN (added instruction)
                "111" when "111100" | "111101", -- BIC (added F2018)
                "000" when others; -- Not DP

-- FlagW: Flag Write Signal
-- Asserted when ALUFlags should be saved
-- FlagW(2) = '1' --> save NZ flags (ALUFlags(3:2))
-- FlagW(1) = '1' --> save C flag (ALUFlags(1))
-- FlagW(0) = '1' --> save V flag (ALUFlags(0))
MOV_Dec <= '1' when Instr(11 downto 4) = "00000000" OR Funct(5) = '0' else '0';

```

```

FlagW_Dec <= ALUOp & cmd & S & MOV_Dec;
with FlagW_Dec select
FlagW <=  "111" when "1010010" | "1010011", -- ADD
          "111" when "1001010" | "1001011", -- SUB
          "110" when "1000010" | "1000011", -- AND
          "110" when "1110010" | "1110011", -- ORR
          "111" when "1101010" | "1101011", -- CMP (added instruction)
          "110" when "1000110" | "1000111", -- EOR (added instruction)
          "110" when          "1110111", -- MOV (added instruction)
          "110" when "1111110" | "1111111", -- MVN (added instruction)
          "110" when "1111010" | "1111011", -- BIC (added F2018)
          "000" when others;    -- Not DP or DP with S=0

-- NoWrite added for CMP instruction to inhibit writing back the
-- result of the subtraction.
-- For the CMP instruction, S is always '1'
-- Delay by one clk to line it up with WReg signal
process (clk)
begin
    if rising_edge(clk) then
        if ALUDecOp = "110101" then NoWrite <= '1'; else NoWrite <= '0'; end if;
    end if;
end process;

-- Instruction Decoder
-- Outputs ImmSrc and RegSrc
RegSrc(0) <= '1' when Op = "10" else '0';
RegSrc(1) <= '1' when Op = "01" else '0';
ImmSrc <= Op;

end Behavioral;

```

Memory.vhd

```

-----
-- Company:      Binghamton University
-- Engineer(s):  Carl Betcher
--
-- Create Date:   11/18/2017
-- Design Name:   ARM Processor Memory
-- Module Name:    Memory - Behavioral
-- Project Name:   ARM Multi-Cycle Processor
-- Target Devices:
-- Tool versions:
-- Description:    Dual-Port Memory with Synchronous Reads

```

```

--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
library std;
use std.textio.all;

entity Memory_DP is

    Generic ( data_width : positive := 32;
              addr_width : positive := 9);

    Port ( clk : in STD_LOGIC;
          WE2 : in STD_LOGIC;
          EN1 : in STD_LOGIC;
          EN2 : in STD_LOGIC;
          A1  : in STD_LOGIC_VECTOR (addr_width-1 downto 0);
          A2  : in STD_LOGIC_VECTOR (addr_width-1 downto 0);
          WD  : in STD_LOGIC_VECTOR (data_width-1 downto 0);
          RD1 : out STD_LOGIC_VECTOR (data_width-1 downto 0);
          RD2 : out STD_LOGIC_VECTOR (data_width-1 downto 0));

end Memory_DP;

architecture Behavioral of Memory_DP is

    -- Declare type for the memory
    type MEM_type is array(0 to 2**addr_width-1)
        of bit_vector(data_width-1 downto 0);

    -- Declare function for reading a file and returning
    -- a data array of the initial memory contents with the program
    impure function init_MEM (file_name : in string)
        return MEM_type is

        FILE MEM_file      : text is in file_name;
        variable MEM_word : line;
        variable MEM      : MEM_type;


```

```

    variable I      : natural;

begin
    -- Loop for reading each line in the file
    -- until end of file is reached
    -- Then, fill in remaining memory with zeros
    I := 0;
    while not endfile(MEM_file) loop
        readline (MEM_file, MEM_word);
        read (MEM_word, MEM(I));
        I := I + 1;
    end loop;
--
-- The following for loop fills in the remaining memory with zeros. But, it generates the following warning
--which I don't understand.
--WARNING:HDLCompiler:746 - "C:\Users\Carl\Documents\EECE 351 - DSD I\LABS - PAPILIO DUO\PROJECT-ARMsc\
--Project Testing\ARM_MultiCycle\source_code\Memory.vhd" Line 69: Range is empty (null range)
--However, with or without this code, the rest of memory past the instructions is filled with zeros.
--It's probably because the type is bit_vector which can only be 0 or 1. So, I'm going to leave it out.
--
-- for J in I to MEM_type'left loop
--     MEM(J) := (others => '0');
-- end loop;
    return MEM;
end function;

-- OLD CODE - required filling in program.txt with zeros beyond the program to fill the entire memory.
-- -- Declare type for the memory
-- type MEM_type is array(0 to 2**addr_width-1)
--     of bit_vector(data_width-1 downto 0);
--
-- -- Declare function for reading a file and returning
-- -- a data array of the initial memory contents with the program
-- impure function init_MEM (file_name : in string)
--     return MEM_type is
--
--     FILE MEM_file      : text is in file_name;
--     variable MEM_word : line;
--     variable MEM      : MEM_type;
--
-- begin
--     -- Loop for reading each line in the file
--     for I in MEM_type'range loop
--         readline (MEM_file, MEM_word);
--         read (MEM_word, MEM(I));
--     end loop;

```

```

--      end loop;
--      return MEM;
--  end function;

-- Declare a signal for the memory array read from the file
signal MEM : MEM_type := init_MEM("program.txt");

begin

-- Memory Port 1
process (clk)
begin
    if rising_edge(clk) then
        if EN1 = '1' then
            RD1 <= to_stdlogicvector(MEM(to_integer(unsigned(A1)))); -- Synchronous Read
        end if;
    end if;
end process;

-- Memory Port 2
process (clk)
begin
    if rising_edge(clk) then
        if WE2 = '1' then
            MEM(to_integer(unsigned(A2))) <= to_bitvector(WD); -- Synchronous Write
        elsif EN2 = '1' then
            RD2 <= to_stdlogicvector(MEM(to_integer(unsigned(A2)))); -- Synchronous Read
        end if;
    end if;
end process;

end Behavioral;

```

Register_File.vhd

```

-----
-- Company: Binghamton University
-- Engineer: Joseph Luciano
--
-- Create Date:    11:23:48 09/23/2019
-- Design Name:
-- Module Name:    RegFile_ARM - Behavioral
-- Project Name:
-- Target Devices:

```

```

-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Register_File is
    generic (addr_size : natural := 4;
             data_size : natural := 32);

    Port ( clk : in  STD_LOGIC;
          A1 : in  STD_LOGIC_VECTOR (addr_size-1 downto 0);
          A2 : in  STD_LOGIC_VECTOR (addr_size-1 downto 0);
          A3 : in  STD_LOGIC_VECTOR (addr_size-1 downto 0);
          WD3 : in  STD_LOGIC_VECTOR (data_size-1 downto 0);
          R15 : in  STD_LOGIC_VECTOR (data_size-1 downto 0);
          WE3 : in  STD_LOGIC;
          RD1 : out STD_LOGIC_VECTOR (data_size-1 downto 0);
          RD2 : out STD_LOGIC_VECTOR (data_size-1 downto 0));
end Register_File;

architecture Behavioral of Register_File is
    type RegFile_type is array (0 to 2**(addr_size)) of std_logic_vector(data_size-1 downto 0);
    signal RegFile : RegFile_type := (others => (others => '0'));

begin

    process(clk)
        begin

```



```

        if rising_edge(clk) then
            if (WE3 = '1') then
                RegFile(to_integer(unsigned(A3))) <= WD3;
            end if;
        end if;
    end process;

    RD1 <= R15 when A1 = "1111" else
        RegFile(to_integer(unsigned(A1)));
    RD2 <= R15 when A2 = "1111" else
        RegFile(to_integer(unsigned(A2)));

end Behavioral;

```

Shifter.vhd

```

-----
-- Company:
-- Engineer:
--
-- Create Date:    09:52:43 11/20/2017
-- Design Name:
-- Module Name:    Shifter - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if using
-- arithmetic functions with Signed or Unsigned values
--use IEEE.NUMERIC_STD.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
--library UNISIM;

```

```

--use UNISIM.VComponents.all;

entity Shifter is
    Port ( shifter_in : in  STD_LOGIC_VECTOR (31 downto 0);
          sh_type : in  STD_LOGIC_VECTOR (1 downto 0);
          shamt5 : in  STD_LOGIC_VECTOR (4 downto 0);
          shifter_out : out STD_LOGIC_VECTOR (31 downto 0);
          shifter_co : out STD_LOGIC);
end Shifter;

architecture Behavioral of Shifter is

    signal shift1 : std_logic_vector(31 downto 0);
    signal shift2 : std_logic_vector(31 downto 0);
    signal shift4 : std_logic_vector(31 downto 0);
    signal shift8 : std_logic_vector(31 downto 0);
    signal shift16 : std_logic_vector(31 downto 0);
    signal LSLcarry : std_logic;

begin

    -- Shift by 0 or 16
    process (shamt5(4), sh_type, shifter_in)
    begin
        if shamt5(4) = '1' then
            case sh_type is
                when "00" => shift16(31 downto 16) <= shifter_in(15 downto 0);           -- LSL
                                shift16(15 downto 0) <= (others => '0');
                when "01" => shift16(31 downto 16) <= (others => '0');           -- LSR
                                shift16(15 downto 0) <= shifter_in(31 downto 16);
                when "10" => shift16(31 downto 16) <= (others => shifter_in(31));   -- ASR
                                shift16(15 downto 0) <= shifter_in(31 downto 16);
                when others => shift16(31 downto 16) <= shifter_in(15 downto 0);   -- ROR
                                shift16(15 downto 0) <= shifter_in(31 downto 16);
            end case;
        else
            shift16 <= shifter_in;
        end if;
    end process;

    -- Shift by 0 or 8
    process (shamt5(3), sh_type, shift16)
    begin
        if shamt5(3) = '1' then
            case sh_type is

```

```

        when "00" => shift8(31 downto 8) <= shift16(23 downto 0);      -- LSL
                           shift8( 7 downto 0) <= (others => '0');
        when "01" => shift8(31 downto 24) <= (others => '0');          -- LSR
                           shift8(23 downto 0) <= shift16(31 downto 8);
        when "10" => shift8(31 downto 24) <= (others => shift16(31));  -- ASR
                           shift8(23 downto 0) <= shift16(31 downto 8);
        when others => shift8(31 downto 24) <= shift16( 7 downto 0);    -- ROR
                           shift8(23 downto 0) <= shift16(31 downto 8);

        end case;
    else
        shift8 <= shift16;
    end if;
end process;

-- Shift by 0 or 4
process (shamt5(2), sh_type, shift8)
begin
    if shamt5(2) = '1' then
        case sh_type is
            when "00" => shift4(31 downto 4) <= shift8(27 downto 0);    -- LSL
                           shift4( 3 downto 0) <= (others => '0');
            when "01" => shift4(31 downto 28) <= (others => '0');        -- LSR
                           shift4(27 downto 0) <= shift8(31 downto 4);
            when "10" => shift4(31 downto 28) <= (others => shift8(31));  -- ASR
                           shift4(27 downto 0) <= shift8(31 downto 4);
            when others => shift4(31 downto 28) <= shift8( 3 downto 0);    -- ROR
                           shift4(27 downto 0) <= shift8(31 downto 4);

            end case;
        else
            shift4 <= shift8;
        end if;
    end process;

-- Shift by 0 or 2
process (shamt5(1), sh_type, shift4)
begin
    if shamt5(1) = '1' then
        case sh_type is
            when "00" => shift2(31 downto 2) <= shift4(29 downto 0);    -- LSL
                           shift2( 1 downto 0) <= (others => '0');
            when "01" => shift2(31 downto 30) <= (others => '0');        -- LSR
                           shift2(29 downto 0) <= shift4(31 downto 2);
            when "10" => shift2(31 downto 30) <= (others => shift4(31));  -- ASR
                           shift2(29 downto 0) <= shift4(31 downto 2);
            when others => shift2(31 downto 30) <= shift4( 1 downto 0);    -- ROR
        end case;
    end if;
end process;

```

```

shift2(29 downto 0) <= shift4(31 downto 2);

end case;
else
    shift2 <= shift4;
end if;
end process;

-- Shift by 0 or 1
process (shamt5(0), sh_type, shift2)
begin
    if shamt5(0) = '1' then
        case sh_type is
            when "00" => shift1(31 downto 1) <= shift2(30 downto 0);           -- LSL
                                shift1(0) <= '0';
            when "01" => shift1(31) <= '0';                                   -- LSR
                                shift1(30 downto 0) <= shift2(31 downto 1);
            when "10" => shift1(31) <= shift2(31);                           -- ASR
                                shift1(30 downto 0) <= shift2(31 downto 1);
            when others => shift1(31) <= shift2(0);                           -- ROR
                                shift1(30 downto 0) <= shift2(31 downto 1);

        end case;
    else
        shift1 <= shift2;
    end if;
end process;

process (shamt5, shift1, shifter_in)
begin
    if shamt5 = "00000" then
        shifter_out <= shifter_in;
    else
        shifter_out <= shift1;
    end if;
end process;

-- ShiFer Carry Out
with shamt5 select
LSLcarry <= '0'           when "00000",
                    shifter_in(31) when "00001",
                    shifter_in(30) when "00010",
                    shifter_in(29) when "00011",
                    shifter_in(28) when "00100",
                    shifter_in(27) when "00101",
                    shifter_in(26) when "00110",
                    shifter_in(25) when "00111",

```

```

        shifter_in(24) when "01000",
        shifter_in(23) when "01001",
        shifter_in(22) when "01010",
        shifter_in(21) when "01011",
        shifter_in(20) when "01100",
        shifter_in(19) when "01101",
        shifter_in(18) when "01110",
        shifter_in(17) when "01111",
        shifter_in(16) when "10000",
        shifter_in(15) when "10001",
        shifter_in(14) when "10010",
        shifter_in(13) when "10011",
        shifter_in(12) when "10100",
        shifter_in(11) when "10101",
        shifter_in(10) when "10110",
        shifter_in(9)  when "10111",
        shifter_in(8)  when "11000",
        shifter_in(7)  when "11001",
        shifter_in(6)  when "11010",
        shifter_in(5)  when "11011",
        shifter_in(4)  when "11100",
        shifter_in(3)  when "11101",
        shifter_in(2)  when "11110",
        shifter_in(1)  when others; --"11111";

    shifter_co <= LSLcarry when sh_type = "00" else '0';

end Behavioral;

```

TopLevel_Multi_Cycle.vhd

```

-----
-- Company:      Binghamton University
-- Engineer(s):   Carl Betcher
--
-- Create Date:   11/18/2017
-- Design Name:    ARM Processor Top Level
-- Module Name:    TopLevel - Behavioral
-- Project Name:   ARM Multi-Cycle Processor
-- Target Devices:
-- Tool versions:
-- Description:
--

```

```

-- Dependencies:
--
-- Revisions:      12/6/2017  Modification to stretch the reset signal to ensure
--                               that the instruction register gets loaded with the
--                               first instruction from memory with the single-port
--                               memory
--
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--use ieee.numeric_std.all;

entity TopLevel is
    Generic ( DELAY : integer := 640000 -- DELAY = 20 mS / clk_period
              );
              -- for Simulation, DELAY = 3
    Port ( Clk : in STD_LOGIC;
           DIR_RIGHT : in STD_LOGIC;
           DIR_LEFT : in STD_LOGIC;
           DIR_DOWN : in STD_LOGIC;
           DIR_UP : in STD_LOGIC;
           SWITCH : in STD_LOGIC_VECTOR (7 downto 0);
           LED : out STD_LOGIC_VECTOR (7 downto 0);
           Seg7_SEG : out STD_LOGIC_VECTOR (6 downto 0);
           Seg7_DP : out STD_LOGIC;
           Seg7_AN : out STD_LOGIC_VECTOR (4 downto 0)
           );
end TopLevel;

architecture Behavioral of TopLevel is

    COMPONENT ARM_Control
    GENERIC ( DELAY : integer := 640000 -- DELAY = 20 mS / clk_period
              );
              -- for Simulation, DELAY = 3
    PORT(
        clk : IN std_logic;
        stop_reset : IN std_logic;
        run : IN std_logic;
        step : IN std_logic;
        stop_at_BP : IN std_logic;
        PC_eq_SWITCH : in STD_LOGIC;
        reset_out : out STD_LOGIC;
        en_ARM : OUT std_logic
    );

```

```
END COMPONENT;
```

```
COMPONENT HEXon7segDisp
```

```
PORT(
```

```
    hex_data_in0 : in  STD_LOGIC_VECTOR (3 downto 0);
    hex_data_in1 : in  STD_LOGIC_VECTOR (3 downto 0);
    hex_data_in2 : in  STD_LOGIC_VECTOR (3 downto 0);
    hex_data_in3 : in  STD_LOGIC_VECTOR (3 downto 0);
    dp_in : IN std_logic_vector(2 downto 0);
    clk : IN std_logic;
    seg_out : OUT std_logic_vector(6 downto 0);
    an_out : out  STD_LOGIC_VECTOR (3 downto 0);
    dp_out : OUT std_logic
);
```

```
END COMPONENT;
```

```
COMPONENT ARM
```

```
GENERIC ( addr_size : positive := 9 );
```

```
PORT(
```

```
    clk : IN std_logic;
    reset : IN std_logic;
    en_ARM : IN std_logic;
    SWITCH : IN std_logic_vector(7 downto 0);
    PCOut : OUT std_logic_vector(7 downto 0);
    InstrOut : OUT std_logic_vector(27 downto 20);
    ReadDataOut : OUT std_logic_vector(7 downto 0)
);
```

```
END COMPONENT;
```

```
-- Constant defining memory address range
```

```
constant addr_width : positive := 9;
```

```
-- Signal for Hex Display Controller input
```

```
signal HexDisp : std_logic_vector(15 downto 0) := x"0000";
```

```
-- Signals for displaying Funct on LEDs, and PC and Memory RD on 7-seg display
```

```
signal PC : std_logic_vector(7 downto 0);
```

```
signal IR_Funct : std_logic_vector(27 downto 20);
```

```
signal MEM_RD : std_logic_vector(7 downto 0);
```

```
-- Signals needed to connect ARM_Control module
```

```
signal stop_reset, run, stop_at_bp, step : std_logic := '0';
```

```
signal en_ARM, reset : std_logic := '0';
```

```
signal PC_eq_SWITCH : std_logic := '0';
```

```

begin

    -- Momentary switches used to control the ARM processor
    stop_reset <= DIR_LEFT;
    run <= DIR_UP;
    step <= DIR_RIGHT;
    stop_at_bp <= DIR_DOWN;

    -- Module to control the ARM processor using button inputs
    i_ARM_Control: ARM_Control
    GENERIC MAP(Delay => Delay)
    PORT MAP(
        clk => Clk,
        stop_reset => stop_reset,
        run => run,
        step => step,
        stop_at_BP => stop_at_bp,
        PC_eq_SWITCH => PC_eq_SWITCH,
        reset_out => reset,
        en_ARM => en_ARM
    );

    -- When program is stopped (en_ARM = '0'), the program counter (PC) is
    -- displayed on the left two characters of the 7-segment display
    -- and the data memory address, A(7 downto 0) is SWITCH and the
    -- data in addressed memory location appearing on ReadData(7 downto 0),
    -- is displayed on the right two characters of the 7-segment display
    HexDisp <= PC & MEM_RD when en_ARM = '0' else x"0000";

    -- Instantiate Hex to 7-segment controller module
    HEXon7segDisp1: HEXon7segDisp PORT MAP(
        hex_data_in0 => HexDisp(15 downto 12),
        hex_data_in1 => HexDisp(11 downto 8),
        hex_data_in2 => HexDisp(7 downto 4),
        hex_data_in3 => HexDisp(3 downto 0),
        dp_in => "000", -- no decimal point
        seg_out => Seg7_SEG,
        an_out => Seg7_AN(3 downto 0),
        dp_out => Seg7_DP,
        clk => Clk
    );

    Seg7_AN(4) <= '1'; -- Anode 4 is always "off"

    -- Instantiate the ARM processor

```



```

i_ARM: ARM
GENERIC MAP ( addr_width )
PORT MAP(
    clk => Clk,
    reset => reset,
    en_ARM => en_ARM,
    SWITCH => SWITCH,
    PCOut => PC,
    InstrOut => IR_Funct,
    ReadDataOut => MEM_RD
);

-- Comparator to determine if PC equals value on Switches
PC_eq_SWITCH <= '1' when PC = SWITCH else '0';

-- Instr (27 downto 20) displayed on LEDs
-- LED(7 downto 0) <= Reverse(IR_Funct);  -- use with Papilio One
LED(7 downto 0) <= IR_Funct;

end Behavioral;

```

program.txt

```

11100000010011110000000000001111
1110001010000000001000000000001
11100100000000000010001111000000
11100010100000000010000111111111
11100010100000010001000011111111
11100010100000010001000011111111
11100010100000010001000011000011
111000101000000010000000011111
1110001010000000101000000100001
11100000100001010101000000000100
1110001001010101010000000001000000
0000001110000010001000000000010
00000100000000010010000000000000
11100000010011110011000000001111
1110000001001111010000000001111
11100000010011110101000000001111
1110001010000001010000000001100
1110001010000001011000000100011
1110001010000100010000000000100
11100100000001000100000000000000
1110000001010100000000000000110
00111010111111111111111111111011
1110000001001111010000000001111
1110001010000001010000000001100
1110001010000100010000000000100
11100100000101000011000000000000
1110000001010011000000000000100
0001001010000101010100000000001
1110000001010100000000000000110
10111010111111111111111111111001
11100010010101010000000000000000
0000001110000010001000000000100
11100101100000010010000000000000

```

111010100000000000000000000000
11100010000000100010000011111011
11100000010011110011000000001111
11100000010011110101000000001111
11100010100000110100000011111111
11100010100001010101000000000001
11100000100001000100000000000100
1110001001010101010000000000010111
00011010111111111111111111111011
11100000100101000100000000000100
0111101000000000000000000000011
11100011100000100010000000001000
11100101100000010010000000000000
11101010000000000000000000000000
11100010000000100010000011110111
11100010100001010101000000000001
11100000100101000100000000000100
0101101000000000000000000000011
11100011100000100010000000010000
11100101100000010010000000000000
11101010000000000000000000000000
11100010000000100010000011101111
11100010100001010101000000000001
11100010100001010101000000000001
11100010010101010000000000100001
00001010000000000000000000000101
11100000100101000100000000000100
00011010111111111111111111111010
11100011100000100010000001000000
11100101100000010010000000000000
11101010000000000000000000000000
11100010000000100010000010111111
11100000010011111001000000001111
11100010100010010011000001010101
11100010100010010100000010101010
11100010100010010101000011000011
1110001010001001011000000111100
11100010100010010111000011111111
111000000000011100000000000101
1110001001011000000000001000001
0001101000000000000000000001011
11100000000101101000000000000101
000110100000000000000000000001001
11100000000001001000000000000111
11100000010110000000000000000100
00011010000000000000000000000110
11100001100000111000000000000100
11100000010110000000000000000111
0001101000000000000000000000011
1110001010000000000000000000011
11100000010011110010000000001111
1110001010000000000000000000011
11100001010100000000000000000001
00111010000000000000000000000000
111010100000000000000000000001001
11100011010100000000000000000000
00001010000000000000000000000000
11101010000000000000000000000110
1110001101010000000000011111111
10111010000000000000000000000000

1110101000000000000000000001
111000101000001000100000000001
1110101000000000000000000000
11100000100111100100000000111
11100100000000100100000000100
111000001001111001100000001111
11100010100001101000000111111
11100000100111101010000000111
11100001101000001010000000100
111000110101010100000000111111
0000101000000000000000000000
11101010000000000000000000101
111000111010000010100010100101
11100011010101010000000010100101
0000101000000000000000000000
111010100000000000000000001001
111000001001111011100000001111
1110001010000111100000001100011
111000001001111100100000001111
1110000110110000100100000001000
00001010000000000000000000100
01001010000000000000000000011
111000111000001000100000000010
1110101000000000000000000000
1110001000000010001000001111101
11100100000000100100000000100
111000001001111010100000001111
1110001001000101010100001111111
111000100100010101010000000001
111000011110000011000000000101
111000110101011000000001111111
0000101000000000000000000000
11101010000000000000000000100
11100000100111101100000000111
1110001010000111100000001100011
111000001001111100100000001111
111000111111000011000011111111
00001010000000000000000000111
01011010000000000000000000110
111000010101010100000000000110
0000101000000000000000000000
11101010000000000000000000001
111000111000001000100000000100
1110101000000000000000000000
111000100000010001000011111011
11100100000000100100000000100
11100000100111100100000001111
11100010100010010011000001010101
11100010100010010100000010101010
1110001010001001010100001111111
111000101000100101100000000000
1110000111000101010100000000011
111000010101010101000010101010
00011010000000000000000000111
010010100000000000000000000110
111000010101010100000000000110
0000101000000000000000000000
11101010000000000000000000011
1110001110000010001000000001000
1110101000000000000000000000
1110001000000010001000001110111
11100100000000100100000000100
1110001110100000101100000001111
11100011101000001100000010101010
111000000101011110100000001100
1110001101011101000000010100101
0000101000000000000000000000

[illegible]