E190AX Advanced Digital Design
Lab 1: RISC-V Software, Simulation, and Testing

The goal of this lab is to learn to:

- assemble and disassemble programs with the RISC-V GCC compiler
- simulate RISC-V programs with the riscvOVPsimPlus virtual platform simulator
- simulate and debug the riscv-wally SystemVerilog with Verilator and Questa
- write your own test cases

Collaboration Policy

You may work with others on the tutorial parts of this lab, so long as you make sure you understand how to do all the steps yourself and make sure everyone you are working with understands how to do the steps. You need to debug the Verilog yourself and write your own test cases, but you may talk with others about solving computer/software problems and about general strategies to approach the lab.

Getting Started

If you don't already have a Slack account, go to slack.com and register for one. Send your username to Prof. Harris to be added to hmc_e190ax.

If you don't already have a github account, go to github.com and register for one. Send your username to the class Git czar to be added to the davidharrishmc/riscv-wally repository.

You should have received an email with your tera.eng.hmc.edu login and password. Review the Hitchhikers Guide to Tera tutorial. Install x2go or your preferred X client and log into Tera. If you aren't already a skilled Linux user, this is the time to get comfortable. Open a terminal and practice getting around without relying on a GUI.

If you aren't already familiar with Git, review Prof. Stine's Git tutorial. Follow the directions on the last slide to clone the davidharrishmc/riscv-wally repository into your home directory. Poke around and get a sense of what is there.

If you haven't already done so from the end of the git tutorial, go to your home directory and clone the repository now:

```
cd ~
git clone https://github.com/davidharrishmc/riscv-wally
```

Assembling and Disassembling Programs with GCC Toolchain

The GCC RISC-V cross-compiler is installed on Tera. We will use it to compile RISC-V assembly language programs into Executable and Linkable Format (ELF), a binary machine language format that can run on a RISC-V processor or simulator.

We'll be using the Imperas test suite of assembly language programs to check that instructions execute correctly. Make a directory for Lab 1 and copy over the RV64I ADD test program:

```
mkdir ~/lab1
cd ~/lab1
cp ~/riscv-wally/imperas-riscv-tests/riscv-test-suite/rv64i/src/I-ADD-01.S .
```

Open Visual Studio Code by typing `code` or using the Applications * Accessories * Visual Studio Code menu. Open I-ADD-01.S and look at the first part. You'll see it loads two randomish 64-bit numbers into registers x16 and x31, adds them into x1, and stores the result into memory at a label "test_1_res." The RVTEST_IO_ASSERT_GPR_EQ macro in the Imperas suite doesn't actually check anything, but is helpful for us as humans to know the answer should be 0xF07C7631C0061DB8. The program does the same for many different values and registers and stores all the results to memory to create a "signature" that should only be correct if all of the instructions in the program work correctly.

Assemble the program using the long and horrible command (suggest cutting and pasting it in)

```
riscv64-unknown-elf-gcc -nostdlib -nostartfiles -march=rv64g I-ADD-01.S
-I../riscv-wally/imperas-riscv-tests/riscv-test-env
-I../riscv-wally/imperas-riscv-tests/riscv-test-env/p
-I../riscv-wally/imperas-riscv-tests/riscv-target/riscvOVPsimPlus
-T../riscv-wally/imperas-riscv-tests/riscv-test-env/p/link.ld -o I-ADD-01.elf
```

The -march flag indicates a machine architecture of RV64G (RV64IMAFD). Other flags tell the assembler not to include standard libraries or startup files, and to look in certain search paths for Include files and for the linker file specifying where the code and data segments go in memory.

Then disassemble the assembly language into machine language using:

```
riscv64-unknown-elf-objdump -d I-ADD-01.elf > I-ADD-01.elf.objdump
```

Look at the machine language .objdump file. (You can use View * Editor Layout * Two Columns and drag one of the files to the other column to see them side by side). Each line indicates the address, the machine language code, and the disassembled assembly code. You'll see some of the included setup code at the beginning that was not visible in the .S file. The simulator begins at address 0x80000000 on reset. The program initializes some of the Control and Status registers (CSRs) including the trap handler vector (MTVEC), loads the start of the testcode (0x80000100) into the Machine Exception Program Counter (MEPC), and performs a MRET to jump to the MEPC. At 0x80000100, the program from the .S file begins. It loads t1 with 0x80003000, which is where the signature is stored. The li pseudoinstruction translates into a series of adds and shifts to form the 64-bit immediate. At 0x80000148, the add takes place (note that registers x1, x31, and x16 are also known as ra, t6, and a6 and are reported as such by the disassembler). Then the sd writes the result to the address given by t1 to save the signature. Similar code repeats for many more test cases.

Compiling and Disassembling Programs with GCC Toolchain

You can also compile C programs using the same flow.

```
cp ~/riscv-wally/imperas-riscv-tests/riscv-ovpsim-plus/examples/fibonacci/fibonacci.c .
riscv64-unknown-elf-gcc -march=rv64g fibonacci.c -o fibonacci.elf
```

To disassemble an ELF into RISC-V assembly and machine language:

```
riscv64-unknown-elf-objdump -d fibonacci.elf > fibonacci.elf.objdump
```

Open fibonacci.c and fibonacci.elf.objdump in Visual Studio Code. Compare the fib function at lines 23-25 of the C program to the assembly language at lines 68-102. Look for how the assembly language saves and restores registers, compares the input to 1, makes recursive function calls, and returns the result in a0. Also notice how the compiler does some dumb and unnecessary things with moving registers around.

See if turning on compiler optimization helps. Recompile with the -O1 flag:

```
riscv64-unknown-elf-gcc -march=rv64g -O1 fibonacci.c -o fibonacci1.elf
```

and then disassemble again. The assembly language code should look much cleaner.

If a bit of optimization is good, how about going all the way with -O3 instead? Now you'll find fib bloated to 230 lines! The compiler unrolled the code to reduce the overhead of recursive function calls at the expense of a much larger program.

Simulating RISC-V Programs with Imperas riscvOVPsimPlus

Imperas makes Virtual Platform simulators for various computer architectures. The simulators use a Just-in-time code-morphing to translate the simulated architecture into machine language on the real host computer so the simulation runs remarkably fast.

Simulate the addition program using the command:

```
riscvOVPsimPlus.exe --trace --traceregs --variant RV64GC --program I-ADD-01.elf > trace
```

Open the trace file in Visual Studio Code and compare against the disassembled program. You'll see a series of lines including the program counter, machine language instruction, and disassembled instruction, followed by a line showing the change to the destination register. Follow the trace through the first 122 lines as the system follows the reset vector, configures some CSRs (details aren't interesting), and starts executing the testcode at line 84.

Try simulating fibonacci.elf in the same way. You can turn off the tracing to speed it up.

```
riscvOVPsimPlus.exe --variant RV64GC --program fibonacci.elf > trace
```

The Imperas suite also comes with the CoreMark program, a benchmark commonly used for ARM and RISC-V processors despite its known limitations (see Hennessy & Patterson for details). It contains kernels of matrix multiply, linked list, and FSM code that emphasize arithmetic, memory, and branching performance. Run CoreMark in simulation using:

```
cd ~/riscv-wally/imperas-riscv-tests/riscv-ovpsim-plus/examples/CoreMark
./RUN_RV32_CoreMark.sh
```

You'll see it runs 1.6 billion instructions in about 2.2 seconds of host computer time, or 750 million instructions per second, remarkable for an instruction simulator. You should see it report CoreMark 1.0 : 43.07746.

Extra credit for the first to track down how this relates to CoreMarks/MHz (43/100 = 0.43, which seems unrealistically slow).

Building the Imperas Test Suite

The Imperas test suite has assembly language test programs for each instruction in each variant of the architecture in

```
imperas-riscv-tests/riscv-test-suite
```

Poke around the directories to see what is there. You don't need to be concerned with b (bit manipulation), k (crypto), or v (vector) extensions, which we are not implementing and which are not compiled by default. Look at the assembly language files in the src directories and the signature reference_output files in the references directories.

It would be extremely tedious to individually compile and test each test. Imperas provides a hierarchical set of Makefiles to automate the process. It is good to learn about Makefiles, but these are unusually complex and are likely only interesting if you are already an experienced `make` user.

Build the entire test suite with

```
cd ~/riscv-wally/imperas-riscv-tests
make
```

Note: if you've opened an additional terminal window, you may need to run the setup script before it can find the paths:

```
source /cad/scripts/setups/S21/riscv-setup
```

The make script assembles every .S file to make an .elf file in the `work` directory, and runs objdump to produce a disassembled version. It runs riscvOVPsimPlus on every elf file, saves the signature in a `.signature.output` file, compares it against the `.reference.output` file, and reports any discrepancies that would be cause by a messed up assembly language file. HMC has modified the make script to also run `exe2memfile.pl` to convert each .elf to a hexadecimal memory file suitable for Verilog to load with $readmemh. Make is smart and looks at creation dates, so if you run it again, it will only recompile test files that you changed.

If you ever think the build could have been corrupted, you can run a fresh build by removing all of the work directory and then reinvoking make

```
make allclean
make
```

One of the learning objectives of this course is to become comfortable automating tasks wherever appropriate. At first it will take you longer to automate than to do the task manually, but as you do repetitive tasks over the semester, you'll be glad you've automated them, and you'll also get faster at developing the automation. Each of you come from different backgrounds and are familiar with different tools and languages;

you can use whatever is natural for you, but also take the time to learn from others and share your techniques with your classmates.

Simulating riscv-wally SystemVerilog with Verilator & Questa

This class starts with riscv-wally, a pipelined processor model in SystemVerilog. It is a parameterized extension of the pipelined processor from Digital Design & Computer Architecture RISC-V edition that supports all of the RV32/64 instructions, the C compressed instructions, and privileged operations (CSRs, exceptions, and interrupts). It passes all of the Imperas tests, although test coverage is woefully thin on the privileged portions.

Look in `~/riscv-wally/wally-pipelined/src`. Poke around the code and figure out how it is structured and what the testbench does. You'll see that it loads all of the Imperas tests from the `.elf.memfile` files, runs them, and checks that the results in memory match the `.signature_output`.

We will simulate with Questa, the industrial-strength version of ModelSim from Mentor Graphics. The Questa warnings are not terrific, so we will use another open-source tool called Verilator with a good "lint" feature to first catch all the bugs that can be found at compile time.

Run Verilator lint with:

```
cd ~/riscv-wally/wally-pipelined
verilator --lint-only --top-module wallypipelinedsoc
-Iconfig/rv64ic src/*/*.sv
```

Alternatively, this is automated in a one-line script to save you typing:

```
./lint-wally
```

You should get no warnings because the code provided is clean.

Now you can simulate with Questa either at the command line or with the GUI and waveform window. The command line version doesn't save all the internal signals so it is about 10x faster and is preferred if you just want to verify that the code runs correctly.

Run the command line with

```
cd regression
./sim-wally-batch
```

You should see it run all of the tests and print SUCCESS! at the end, or report any failures (there should be none). Look in the script to see how it invokes a .do file, and look at the .do file to see how it compiles and runs the code automatically. Also notice how the script provides the rv64ic configuration. Look at the wally-config.vh file in that directory specifying XLEN, MISA, and other characteristics of the configurable processor. You could run on a different configuration by specifying the file.

Next look at `sim-wally`, which invokes the GUI, and look at the corresponding `wally-pipelined.do` file, which uses the `+acc` flag to keep accessibility of all internal signal for debugging, and the add wave commands to add all of the signals, with the most interesting ones at the top. It defaults to config/rv64ic. Run the script with the command

```
./sim-wally
```

and watch the first test program (rv64i/I-ADD-01) run. You should see the program counter start at 0x8000000, and then the initialization stuff runs. Scroll until the program counter reaches 0x80000100 and compare against the objdump file to watch the simulation, load registers x31 and x16, add into x1, and store the result into memory at 0x80003000. Review the waveforms until you understand what each of the stages of the pipeline is doing up through the first `sd` instruction at 0x8000014c. You'll need to understand the waveforms well enough to be able to debug discrepancies in the next part, and you may need to review Chapter 7 of DDCA if you haven't looked at pipelined processors in a while.

Note that different tests store their results to different locations in memory. The testbench lists the starting address offset for each test and automatically checks appropriately. The rv32/64i test cases start the body of the program at 0x80000100, but the compressed instructions start at 0x800000E4 because the startup code is a little shorter. You don't need to do anything different to run the simulation, but if you are checking results by hand, it's useful to be aware of these addresses.

If sim-wally fails and you want to make a fix to the Verilog and rerun without waiting for Modelsim to close and reopen, you can type into the Modelsim transcript window command line:

```
do wally-pipelined.do
```

Finally, we have a regression script that runs all of the configurations (initially rv32ic and rv64ic). Invoke with

```
./regression-wally.py
```

It should print success or failure of each test, and an overall success or failure. Running this entire regression is time-consuming, so you may not want to do so each time you check in code. However, the regression czar should set up a script to run it nightly and let team members know if they broke something with a checkin.

Debugging riscv-wally SystemVerilog

The riscv-wally repository has a branch called lab1buggy with four bugs for your debugging pleasure. In this section, you should develop a systematic way to find bugs in the Verilog without having to strain your brain very hard.

Switch to the buggy branch by typing

```
git checkout lab1buggy
git pull
```

If you were a git power user, you could diff lab1buggy and the main branch and see the bugs I put in, but that would miss the point of the lab. Instead, let's get some practice digging in and debugging. However, if one day in the future the processor was working and you made some changes and it stops working, it's well worth using git to examine the changes you just made to find the cause.

Now run `lint-wally`. You should see a syntax error in datapath.sv. Use the information in the error to find and fix the bug. Run `lint-wally` again and check that the bug is gone.

Next we'll walk through debugging one of the substantive bugs. Run sim-wally. The simulation will hang. Choose Simulate * Break to stop the endless loop. Yikes, this is going to be a hard one. Go to the beginning of the simulation and zoom out until you can see the program counter. You'll see it start out normally, then get into an endless loop through 8000003E 40 44 48 at about 200 ns. It's immediately suspicious that the program counter should go to 3E, which is not a multiple of 4.

A general strategy is to find the first error we can identify, then trace signals back until we can find where the inputs are good and the output is bad, and we've localized the

problem. This requires understanding the design well enough to be able to predict what the signals should be, but takes no special genius to stare at the code and have the bug pop out. It's useful to track these things in your engineering notebook because it's easy to get lost about what you were investigating and what you expect.

Let's look at the test bench and test program to figure out what it should be doing. According to the test bench at line 80, the first test is rv64i/I-ADD-01. Open ~/riscv-wally/imperas-riscv-tests/work/rv32i/I-ADD-01.elf.objdump. Sure enough, there is no code at 3E. Let's trace back and see how the PC got there.

In Visual Studio Code, choose File * Open Folder and open the wally-pipelined/src folder. All of the .sv files in the design should appear in the left column. According to the SystemVerilog, PCF is an output of the datapath. Which module produced it? Since the modules are instantiated with .* notation, we can't just search in the module and find out. But pclogic sounds promising, and looking in there, we see PCF is an output, coming from the pcreg flop, whose input is PCNextF.

Use the sim pane in Modelsim to find the relevant signals. Look under testbench/dut/hart/dp/pclogic and drag PCNextF to just under PCF for easy reference. We see at time 190 that it becomes 3E, which is bad. According to the SystemVerilog, PCNextF comes from UnalignedPCNextF, which comes from the pcmux controlled by {PrivilegedChangePCM, PCSrcE}. These waveforms are {1,0}, which tells the mux to select the PrivilegedNextPCM input, which is also 3E, which can't be right.

It's easiest to trace this one if you have spent enough time with the code to understand where signals are coming from. Otherwise, use Edit * Find in Files… and enter PrivilegedNextPCM, which shows up as an output of trap.sv. Double-click on that search result, and we see at line 67 an always_comb block describing a mux to choose PrivilegedNextPCM. Adding more waveforms from trap, we see PrivilegedNextPCM is still 3E here (it could be different in different parts of the hierarchy if we somehow misconnected signals). It doesn't seem like any of the conditions of the if statements are true, and MTVEC_REGW is 3E, explaining PrivilegedNextPCM.

Now, why is MTVEC bad? Looking at the objump, we see mtvec was written at address 58 with csrw mtvec, t0, and t0 got a value from addi t0, t0, 16 at 54. t0 is register x5 (you can look this up in Appendix B). Let's scroll back and see if something is wrong with t0. We see the PC was 54 back at time 111. One pipeline stage later, we see the ADDI has reached the Decode stage (time 121). Scroll forward to fin the ADDI in the Writeback stage at 152. Just to be paranoid, we see the PC is still 54 here, as we expected. RegWriteW is 1, RdW is 5, and ResultW is 3e, explaining writing 3E to t0.

Why did ADDI produce the wrong answer. Look back to the Execute stage at 131. SrcAE is 4F and SrcBE is 10 and the result is 3e. This doesn't seem right that 4F + 10 = 3E. Let's look closer at the ALU.

Look at the ALU Verilog to see how it should operate. Drag some signals into the waveforms. Drag over a and b, which agree with SrcAE and SrcBE. Look at alucontrol, which is 00. According to line 77, that should be for addition or subtraction, and result should get sum. Look at sum, which is 3e again. Still bad. Look at the code again. Sum comes from presum, which depends on condinvb. presum is 3e, still bad. condinvb is FFFF...FFEF. This seems suspicious for an addition. So we've isolated the problem to line 41, in which the inputs are alucontrol = 0 and b = 10, and the output is condinv = FF..FFEF rather than 10. Looks like we are inverting when we shouldn't.

Switch the ~b and b in line 41. Let's see check that fixes the problem. Type do wally-pipelined.do in the Transcript window to rerun the sim. Another infinite loop. I think I'm going to hate this lab. Break the sim again.

Now scroll to around 200. Phew, no 3E anymore. In fact, we see the PC looping 4c 50 54 now. Looks like we fixed the condinv bug. Scroll back and we see the problem started somewhere around 100 ns. Now you're on your own to fix this third bug. Record what it was.

Once you've fixed it, there's a fourth bug still bug lurking. You'll find there is no more endless loop and most test programs pass, but 7 have errors. Most of the test cases on ADDW, SLLW, and a few others are failing. You could try scrolling through 1.5 million ns of simulation to find the first failure, but it's easier to go into the testbench and copy the rv64i/I-ADDW-01 from line 83 the beginning of the tests64i (line 80) so it runs right away. According to the testbench, the 0th test is bad (oh good, easy to locate), and simulation is producing f07c7631c0061db8, while the expected signature is ffffffffc0061db8. Looks like something went wrong with the upper bits. Rerun the sim, compare against the I-ADDW-01 objdump to know what should be happening, and you be able to locate this bad result early in the simulation. If you're in doubt of what I-ADDW-01 should be doing as it runs, you can use riscvOVPsimPlus to generate a trace of expected register results, and compare that to the ResultW signals when RegWrite = 1 (but it's also good to get used to being able to predict mentally). Squash the last bug and record what you did to fix it.

When you are all done, don't check back in your corrected code, or your classmates will miss out on the fun of finding the bugs themselves. Instead, switch back to the main branch.

```
git checkout main
git pull
```

Writing new Test Cases

The imperas-riscv-tests are not particularly thorough. In particular, they don't test the range of operand inputs very well. We would like to add more design validation test cases, yet not so many that simulation becomes painfully slow. A good set of test vectors includes directed and random tests.

For example, consideADD. The vectors should include "corner cases" that are most likely to do something unusual. Good 64-bit corner cases include the smallest and largest numbers, numbers right near the two's complement rollover from negative to positive, and a random positive and random negative number.

{0, 1, 2, FF, 624B3E976C52DD14, 7FFFFFFFFFFFFFFE, 7FFFFFFFFFFFFFFF, 8000000000000000, 8000000000000001, C365DDEB9173AB42, FFFFFFFFFFFFFFFE, FFFFFFFFFFFFFFFF}

For ADD, let's create a set of directed vectors consisting of the cross-product of all 12 corner cases for each input, or 144 vectors. Let's also generate 100 completely random vectors. For each vector, let's place the two inputs in a random pair of registers, subject to constraints:

- don't use the same register for both inputs!
- don't use x0, which is hardwired to 0
- don't use x6, which points to a place in memory to store the results.

Look at the wally-pipelined/testgen/testgen-ADD-SUB.py script. It generates these test vectors, including the expected values, for both ADD and SUB. It then writes a .S assembly language program containing the tests, and the .reference.output file with the expected result. It does this for both XLEN=32 and 64, and places the .S and .reference.output files in the appropriate imperas-riscv-tests/riscv-test-suite/rv32i or rv64i directories. Study the script to understand how it works and how you would modify it to add other instructions and corners.

Run the script and look at the files it produces. Edit the Makefrag file in the rv32i and rv64i directories to add these files (WALLY-ADD and WALLY-SUB) to the set being tested. Run make to build the new tests, and check that the .reference.output matches simulation (make should print PASS for each test).

If tests don't pass, look in work/rv32i or 64i for WALLY-ADD.diff that shows the discrepancies between the signature and reference values. Your script probably generated incorrect expectations. You can then look at the .S file to find the corresponding bad test case.

Add the WALLY-ADD and WALLY-SUB test cases to the SystemVerilog testbench. Take care to check the address where the results begin (look at the begin_signature address in the .objdump). Simulate with Modelsim and confirm they all work.

Your new Test Cases

Sign up on the Lab 1 Test Vector spreadsheet for which group of instructions you want to test (and based on the unit you are on). It's ok to for a few people to do this as pair programming, so long as every group gets covered. The instructions high on the list are most like the ADD example and are best if you're not too experienced writing scripts, though some will involve some fun bit manipulations in Python to produce the expected results. The instructions lower on the list will require more original coding and are best if you're experienced with software.

If you are interested in building the M (multiply/divide) hardware, pick these instructions; they aren't yet implemented, so you'll only be able to test that they run on riscvOVPsimPlus, not on the actual hardware yet. Later in the project, you'll use your vectors.

If you are interested in the hardware/software interface, consider signing up for the CSR instructions and taking ownership of a comprehensive privileged test suite this semester.

Edit Section 7.1.1 of the Wally Architecture Specification and write your test plan in the appropriate subsection. Think first about what you think should be included and write that down. Then browse through the Imperas test cases for your group of instructions and make sure your cases are more comprehensive and haven't overlooked anything they are testing. Aim for hundreds of test cases unless you have a compelling reason for more (e.g. floating point, or eventually the full privileged system).

Write your test vector generator. Place it in the wally-pipelined/testgen directory. It should put tests and expected values in the imperas-riscv-tests/riscv-test-suite/rv32i and rv64i. Modify the Makefrag scripts to include your tests. Run make and fix any problems. Besure the Check says OK, not IGNORE, and that there are no errors reported and that the log files in the work directory look clean.

Unless you are doing the M instructions, add your instructions to the SystemVerilog testbench. Remember to check where begin_signature is located in your objdump file; it is typically 0x3000 or 0x4000 past the start of the program depending on the length of the code, and needs to be manually entered in the testbench. Run with ModelSim and confirm that the processor still works. Fix the bugs in your test vectors. Extra credit if you can find a failure in the Verilog!

Check in your test vector generator, new tests and expected values, Makefrag, and the updated testbench.sv to the main branch of the git repository. Remember to do a git pull on the repository first to make sure it is up to date and minimize conflicts, especially on testbench.sv and Makefrag because everyone will be changing it. Check that it looks good (around the list of new test vectors) and still runs after you've checked it in.

What to Turn In

[4] What were the third and fourth bugs in the lab1buggy branch?

[2] Edit the appropriate section of 7.1.1 of the Architecture Specification with your test plan. Extra credit for hard test plans that show deep thought.

[2] Did all of your tests pass "OK" when you run Make? Paste in the appropriate part of the transcript as proof.

[2] Did all of your tests run successfully in ModelSim? Paste in the appropriate part of the transcript as proof.

[Extra Credit] Were you able to detect and fix any bugs with your test vectors?

[1] How much time did you spend on this lab? The amount will not count toward your grade, but you will get a point for reporting it.