Overview

The goal of this assignment is to get you familiar with design using Chisel and some of the VLSI CAD tools, both of which will be used throughout the course. This lab should also introduce you to SHA3 a cryptographic algorithm you will be implementing and optimizing over the semester. Specifically, during this lab you will implement a basic version of SHA3, test it using three backends, FIRRTL, Verilator, and VCS, and finally gain an understanding for the algorithm trade offs in SHA3.

Deliverables

This lab is due Saturday, January 28 at 11:59PM. The deliverables for this lab are:

- (a) your working Chisel RTL checked into your private git repository at Github
- (b) a set of tests for both individual modules and the complete design (also checked into your repository)
- (c) build results and reports generated by Chisel C++ and VCS checked into your git repo (results and reports only! No binaries!)
- (d) written answers to the questions given at the end of this document checked into your git repository as writeup/report.pdf or writeup/report.txt

You are encouraged to discuss your design with others in the class, but you must write your own code and turn in your own work.

VLSI Toolflow Introduction

You will use Chisel to generate a FIRRTL intermediate representation. This intermediate representation is either used to run on a FIRRTL interpreter or used to generate Verilog. The FIRRTL interpreter serves as a low startup overhead to test your design. It has the slowest performance per cycle and is not testing the Verilog that you will be running through the VLSI tools.

To test longer running simulations, we will use a Verilator backend. The Verilator backend takes the Verilog output, cross compiles it to C++, and then compiles it to native instructions. It has significantly faster performance per cycle but has the disadvantage of two translations and therefore has a lower level of trust.

Lastly, we have available to us a Synopsys VCS backend. The VCS Synopsys backend takes the Verilog and compiles it down directly. It has worse performance per cycle than Verilator but better
performance per cycle than the FIRRTL interpreter. VCS Synopsys only does one translation of the Verilog and is a commercial tool providing better support, hopefully.

Using the tests you write, you can verify the functionality of your RTL without the use of any CAD tools. Typically, you will run small unit tests first on the FIRRTL interpreter. When you are satisfied that it works for small tests, you will then verify the Verilog output using Verilator and Synopsys VCS. Both tools are capable of producing a more detailed debugging aid, a *vcd* file. This extra detail comes at a slow down in simulation and is a less productive but sometimes necessary method. Another CAD tool Discovery Visualization Environment (*dve*) can read and display a waveform view of the circuits operation, typically from a *vpd* or *vcd* file.
Getting Started

You can follow along through with this lab by typing in the commands marked with a '%' symbol at the shell prompt. To cut and paste commands from this lab into your bash shell (and make sure bash ignores the '%' character) just use an alias to "undefine" the '%' character like this:

```bash
% alias %=""
```

Note: OS X Preview may not copy newlines correctly. If you have problems, try using Adobe Reader.

All of the CS250 laboratory assignments should be completed on one of the EECS instructional machines allocated for the class. Please follow the setup instructions on the course website before attempting this lab. In particular, you will need to source a setup script before you can run the CAD tools. This script specifies the location of each tool and sets up necessary environment variables.

You will be using Git to manage your CS250 laboratory assignments. Please see the Git tutorial posted on the course website for more information about how to use Git. Each student will have a private git repository hosted on github.com. If you don’t already have a Github account, you will need to create one. Once you have an account, you must post your Github account name and CS250 class account username on Piazza. Once you do this, your TA will create a private repository for you and you will be able to access the lab materials.

The lab materials we provide will be hosted in a template repository. You will clone this template repository to a directory on the machine you’re working on. Afterwards, you will set your remote repository to point at your private repository. This will create a local repository that is linked to two different remote repositories (one which is managed by the staff and is read only, while the other is your private repository). If any updates are made to the template repository, you should be able to easily merge the changes into your local repository.

As the CAD tools generate a lot of data and your class account home directories have only a small disk quota (not to mention access speed issues with network mounted filesystems), we will need to use the local disk of the machine to store the outputs of the CAD tools. By default, the permissions on a directory that you create in /scratch will be set to that its contents are only readable by your class account. You will use git to backup your design files to a server hosted by github. Assuming your username is cs250-ab (change this to your own class account username), you can create a local working git directory on one of the EECS instructional machines using the following commands:

```bash
% cd /scratch
% mkdir cs250-ab
% cd cs250-ab
% git init
% git remote add template git@github.com:ucberkeley-cs250/lab-templates.git
% git remote add origin git@github.com:ucberkeley-cs250/cs250-ab.git
```

You will need to setup ssh keys and export them to github in order to access the repositories. To do this, follow the instructions at https://help.github.com/articles/generating-an-ssh-key/.

To do this the lab you will make use of some infrastructure that we have provided. The infrastructure includes Makefiles and scripts needed to complete the lab. The following commands fetch these
files from the template repository, and then copy them into your private repository. To simplify the
rest of the lab we will also define a `$LABROOT` environment variable which contains the absolute
path to the project’s top-level root directory.

```bash
% cd /scratch/cs250-ab
% git pull template master
remote: Counting objects: 191, done.
remote: Compressing objects: 100% (136/136), done.
remote: Total 191 (delta 41), reused 188 (delta 41)
Receiving objects: 100% (191/191), 185.87 KiB | 293 KiB/s, done.
Resolving deltas: 100% (41/41), done.
From https://github.com/ucberkeley-cs250/lab-templates
* branch master -> FETCH_HEAD

% git pull origin master #if remote ref master cannot be found, don’t worry
#proceed to next command

...% git push origin master
...
% git submodule init
% git submodule update
% cd lab1
% LABROOT=$PWD
```

The `git submodule` commands are used to clone external git repositories that the project depends
on. In this lab, no submodules will be needed.

The two remote repositories are named template and origin. origin points to your private repository
and template points to the read-only staff account. If the provided lab files are ever updated, a
simple `git pull template master` should merge in these changes with your own local versions of
the files.

Please commit and run `git push origin master` frequently. /scratch is only intended as tem-
porary storage and is not backed up. /scratch lives on a local drive, so if you ever decide to
work on a different machine, you can push/pull your design files to/from Github to move your
design files from one machine’s local drive to the other’s. Follow the instructions below to move
files between machines (assuming all the files of interest have already been committed to your local
repository). This procedure will not move your build directories (you will need to rerun synthesis
or place-and-route to regenerate the files on the new machine), so only switch machines if there is
a good reason to do so.

(on machine A)

```
% git push origin master
```

(on machine B)

```
% git pull origin master
```

The resulting `$LABROOT` directory contains the following subdirectories: src contains your source
Chisel; build contains the generated files for simulating both the C++ code with the emulator
and the verilog code with vcs. The src directory contains the Scala modules and tests you will
be using in this lab assignment. Figure 1 shows each directory that you have been provided and
includes comments about what they do.
lab1/

Makefile build tools that handles elaboration and testing commands
.gitignore Tells Git to ignore generated-src/, and other dynamically generated files
src/main/Sha3 Main directory for sha3 project
    scala/ .scala Chisel code your implementation lives here
    c/ .c Reference implementation in C
src/test/Sha3/scala *.scala Your Chisel test code lives here
test-run-dir/ Generated code for simulation
    verilator/ Verilator generated files for verilog simulation
    vcs/ VCS generated files for verilog simulation
    firrtl/ FIRRTL generated code for interpretation
project/ sbt project specifications
build.sbt sbt project specifications
build/ Reports and generated Verilog for VLSI tools generated here
chisel/ sbt generated files
target/ sbt generated files

Notation:
blue means that these files generated dynamically, and are not stored in the repository

Figure 1: Directory organization for lab1/
Introduction to SHA3

Secure hashing algorithms represent a class of hashing functions that provide four attributes: ease of hash computation, inability to generate the message from the hash (one-way property), inability to change the message and not the hash (weakly collision free property), and inability to find two messages with the same hash (strongly collision free property). The National Institute of Standards and Technology (NIST) recently held a competition for a new algorithm to be added to its set of Secure Hashing Algorithms (SHA). In 2012 the winner was determined to be the Keccak hashing function and a rough specification for SHA3 was established. The algorithm operates on variable length messages with a sponge function, and thus alternates between absorbing chunks of the message into a set of state bits and permuting the state. The absorbing is a simple bitwise XOR while the permutation is a more complex function composed of several operations, χ, θ, ρ, π, ι, that all perform various bitwise operations, including rotations, parity calculations, XORs, etc. The Keccak hashing function is parameterized for different sizes of state and message chunks but for this lab we will only support the Keccak-256 variant with 1600 bits of state and 1088 bit message chunks. In addition, for this lab we will ignore the variable length portion to avoid one of the most complicated parts of Keccak the padding. Our interface, which is discussed further below, assume a single chunk of message is ready to be absorbed and hashed. You can see a block diagram of what your resulting design should look like in Figure 2.

The SHA3 standard is documented in FIPS PUB 202. The draft specification of SHA3 that was available at the time this lab was originally written can be found at http://csrc.nist.gov/publications/drafts/fips-202/fips_202_draft.pdf with the final version at http://dx.doi.org/10.6028/NIST.FIPS.202. You are encouraged to take a quick glance at specification as it is a good example of a standards document. If you are new to reading standards documents, you will probably notice that the descriptions in the document are quite dense and may require some time to understand. This is true of many standards documents with FIPS PUB 202 being a rather short example (the original IEEE 802.11 standard has over 2,700 pages with each addendum such as b, g, n, and ac adding more pages). Hardware designers are often responsible for implementing different standards so learning how to read standards documents is a valuable skill (although, not one we will practice in this lab).

Fortunately, you will not need to look at the FIPS PUB 202 too closely for this lab as a C reference implementation is provided for you. The C implementation constitutes the software golden reference for the lab. The first step in most projects is to create a software golden reference that produces the behavior that is expected from the hardware design. The results produced by the hardware design are compared against the results from the golden reference design to determine if the hardware design is functioning as expected. In addition to providing the model by which the hardware design is checked, the golden reference can also serve as a basis for the initial hardware design.

You will implement the SHA3 design based on the C reference implementation. You are not encouraged to check the consistency of the C implementation with the standards document as this will take a long time and is not the focus of this lab. Your implementation will be compared against the C implementation and not the description in FIPS PUB 202. Do not attempt to add any additional features of SHA3 that are not included in the C implementation.
You can run the C reference version on the simplest input with the following commands:

```
% cd $LABROOT/src/main/c
% make
% make run
```

This will by default print out the different values of the state after each permutation and round. Your chisel implementation should match each of these steps exactly. Tracing through any differences is a good way to debug the whole design, but early simpler tests should help you avoid this tedious exercise.

![Block diagram for SHA3](image)

Figure 2: Block diagram for SHA3
SHA3 Datapath: Implementation and Testing Strategies

Rather than jumping in and implementing the entire SHA3 design above it would be better to start with something smaller but still testable. This should reduce complexity and debugging time. The most logical way to begin the design would be to create a single cycle version that simply performs the permutation. Even this design has multiple components that are individually testable. A good implementation strategy would be to design each of the function blocks, $\chi$, $\theta$, etc, individually and write unit-tests for the blocks. The chisel source directory has already a skeleton of the code you will need to write, outlining how you should organize your implementation.

The given directory includes one of the modules implemented with a test. You are responsible for implementing the remaining modules and associated tests.

You can run the given test with the test-unit-firrtl, test-unit-verilator, test-unit-vcs make targets. test-unit make target defaults to test-unit-firrtl. These targets allows you to choose any of the main classes to be run, so they will continue to work as you add more tests for new modules. test-unit-firrtl uses the FIRRTL interpreter; test-unit-vcs uses Verilog with VCS; and test-unit-verilator uses Verilog with Verilator. These three testing tools are discussed in more detail below.

Testing a design in this manner should make integration easier and more bug free. Once you have connected the datapath together another logical point to test the design arrises and you should have something like Figure 3. In addition to the unit-tests from before you should now write a larger test to ensure the permutation is happening correctly.

![Figure 3: Block diagram for SHA3](image-url)
SHA3 Control: State Machines and Interfaces

With a complete and tested datapath the next step in implementing SHA3 is to write something to control the datapath. For this lab some of the more complex needs of the SHA3 accelerator have been abstracted away or given to you. You will be given a section of the message that has already been read from memory and been padded appropriately. This limits the lengths of messages your design can hash to those smaller than 1088 bits, but makes the design significantly simpler. Don’t worry in future labs you will remove this limitation and get to deal with all the complexities that entails.

With this interface you will need to implement a control state machine that can read the message data into the datapath’s state element, perform the correct number of permutations, and finally return the resulting hash. The control state machine should also adhere to the ready valid protocol for these signals. The state machine should keep track of whether the accelerator is busy and how many rounds of permutation have been done. In addition, since we are only hashing a single chunk at a time the state machine is also responsible for starting each hash with the correctly absorbed state.

The main chisel file for the whole accelerator includes a test that should test most of the functionality. This test is replicated in a step by step fashion online at:


SHA3 Chisel Testing

When you are ready to test your code, there are three methods from which to choose. First, Chisel can run a FIRRTL interpreter which implements a cycle-accurate simulation of your design. To do this, run the following commands:

% cd $LABROOT
% make test-firrtl

Second, Chisel can run your tests by generating Verilog and then compiling it using Verilator.

% cd $LABROOT
% make test-verilator

Third, Chisel can run your tests by generating Verilog and then using Synopsys VCS.

% cd $LABROOT
% make test-vcs

In addition, Chisel can also generate Verilog code that can be used as input to an ASIC flow.

% cd $LABROOT
% make verilog

Once you are happy that your design passes the given test you should add at least one additional test for the design. It could be a test that checks for a different hash, or a test that tests for a
specific control sequence that seems difficult to get right or any other case you think might not be handled correctly.

Finally, in addition to committing your tests and source I would also like you to run two more make commands to save the output of these runs for submission.

```
% cd $LABROOT
% make reports
```

This will create two files in build and build that will record the results of your simulations.

### Debugging with Chisel

To debug your Chisel design, you can use either the FIRRTL interpreter, Verilator, or Synopsys VCS.

Chisel has a specific debug API for the tester consisting of peeks, pokes, and expects, that you have experimented with in the first chisel getting started assignment. In this way, during your test you can request the value of any signal you can name with peek.

#### Verilator: Simulating your Verilog

In this lab, we will not be using Verilator directly but through Chisel. Verilator produces a waveform file, *.vcs, that we will use to help us debug. Chisel first elaborates your Chisel code into Verilog. Verilator then takes this code, cross compiles into C++, and the C++ is then in turn compiled into native instructions producing a binary. Chisel then interacts with this binary to test your circuit.

#### Synopsys VCS: Simulating your Verilog Again

In this lab we will also not be using VCS directly but rather using it through Chisel, so the exact options are less important right now but for your reference info on the options and how a more complete setup, which might be used in later labs is included below.

VCS compiles Verilog source files into a native binary that implements a simulation of the Verilog design. VCS can simulate both behavioral and RTL level Verilog modules. In behavioral models, a module’s functionality can be described more easily by using higher levels of abstraction. In RTL descriptions, a module’s functionality is described at a level that can be mapped to a collection of registers and gates. Verilog behavioral models may not be synthesizable, but they can be useful in constructing testbenches and when simulating external devices that your design interfaces with. An example of compiling with VCS is given below.

```
% cd $LABROOT/build-unscripted/vcs-sim-behav
% vcs -full64 -PP +lint=all +v2k -timescale=1ns/10ps \
   ../../src/gcdGCDUnit_behav.v \
   ../../src/gcdTestHarness_behav.v
```

By default, VCS produces a simulator binary called simv. The -PP command line option turns on support for using the VPD trace output format. The +lint=all argument turns on all Verilog warnings. Since it is quite easy to write legal Verilog code that doesn’t behave as intended, you
should always enable all warnings to help you catch mistakes. For example, VCS will warn you if you try to connect two nets with different bitwidths or don’t wire up a port on a module. Always try to eliminate all VCS compilation errors and warnings. The +v2k command line option tells VCS to enable Verilog-2001 language features. Verilog allows a designer to specify how the abstract delay units in their design map into real time units using the ‘timescale compiler directive. To make it easy to change this parameter you will specify it on the command line instead of in the Verilog source. After these arguments you list the Verilog source files. The -v flag is used to indicate which Verilog files are part of a library (and thus should only be compiled if needed) and which files are part of the actual design (and thus should always be compiled). After running this command, you should see text output indicating that VCS is parsing the Verilog files and compiling the modules. Notice that VCS actually generates C++ code which is then compiled using gcc. When VCS is finished there should be a simv executable in the build directory.

**Debugging with DVE**

Where should you start if a design doesn’t pass all your tests? The answer is to debug your RTL code using the Discovery Visualization Environment (DVE) GUI to generate a waveform view of signals in your design. The simulator already has already written a trace of the activity of every net in your design to the Sha3Accel.vcd file. DVE can read the Sha3Accel.vcd file and visualize the waveform. Notice that the design will contain many signals with the T prefix, which hold intermediate values produced by the Chisel compiler.

```bash
% cd $LABROOT
% make test-verilator
% dve -full64 -vcd test_run_dir/verilator/sha3.<AutoGeneratedName>/Sha3Accel.vcd &
```

You can also use DVE to debug failing unit tests.

```bash
% cd $LABROOT
% make test-verilator
% make test-verilator
% dve -full64 -vcd test_run_dir/verilator/sha3.<AutoGeneratedName>/<UnitName>.vcd &
```

To add signals to the waveform window (see Figure 4) you can select them in the hierarchy window and then right click to choose Add To Waves > New Wave View.
Questions

Your writeup should not exceed one page in length. Make your writing as crisp as you can!

Q1. W=32 vs. W=64

Throughout the lab, we were focused on implementing Keccak-1600 which uses a word size of 64 bits and a state size of 1600 bits. The algorithm is also defined for Keccak-800 which scales everything down to a word size of 32 bits. In this question we would like you to think about how to create a single chisel design that could be compiled to run either of these algorithms.

• What changes would you need to make to the datapath and the control unit?
• Does the current test harness work for both versions of the algorithm? If not, what could you do to make the tester more parameterized.
• How does this change effect the area/power/performance properties of the chip? This question is more qualitative now but as we progress through the labs and begin to use more of the CAD tools we will be able to make a more concrete claim about this properties.

Q2. Standards Documents

Technical standards play a very important role in engineering and computer science. Standards allow multiple vendors to implement compatible products by working off of the same specification.

Answer the following questions at a high level by looking at the sections of FIPS PUB 202 that detail the keccak algorithm (sections 3.2 - 3.4) as well as the keccakf function in the C reference. Don’t worry too much about the technical details, focus on high level observations.

• Why might one produce a standards document like FIPS PUB 202 instead of simply releasing a C reference?
• What are some benefits of producing a software golden reference before beginning the hardware design?

Q3. Chisel vs. Verilog

If you have used Verilog before to design circuits please tell us how you felt creating a design like this in Chisel. Was it easier, harder, did any of Chisel or Scala’s features make things simpler?

Read me before you commit!

• Commiting is not enough for us to grade this lab. You will also need to push your changes to github with the following command: `git push origin master`
• If you are using one or more late days for this lab, please make a note of it in your writeup. If you do not, your TA will assume that whatever was committed at the deadline represents your submission for the lab, and any later commits will be disregarded.
• Please note in your writeup if you discussed or received help with the lab from others in the course. This will not affect your grade, but is useful in the interest of full disclosure.
• Please note in your writeup (roughly) how many hours you spent on this lab in total.
• To summarize, your Git tree for lab1 should look like the following (use the Github web browser to check that everything is there):

```
/cs250-ab
/lab1
    /src: COMMIT CHISEL CODE
    /build: COMMIT firrtl-report AND vcs-report
    /writeup: COMMIT REPORT
```

Acknowledgements

Many people have contributed to versions of this lab over the years. The lab was originally developed for CS250 VLSI Systems Design course at University of California at Berkeley by Yunsup Lee. Original contributors include: Krste Asanović, Christopher Batten, John Lazzaro, and John Wawrzynek. Versions of this lab have been used in the following courses:

• CS250 VLSI Systems Design (2009-2016) - University of California at Berkeley
• CSE291 Manycore System Design (2009) - University of California at San Diego