From Gates to Compilers: Putting it All Together

CS250 Laboratory 4 (Version 111814)
Written by Colin Schmidt
Adapted from Ben Keller

Overview

In this lab, you will continue to build upon the Sha3 accelerator, in several ways. In the first part of the lab you will use Chisel's Mem construct to replace the absorb buffer with an SRAM rather than flip-flops. You will then connect your accelerator to a fully instantiated RISC-V Rocket processor, and write assembly instructions to drive your processor with code instead of a custom testharness. This process will give you a good introduction to the implementation options available to you as you begin your final projects, as well as the process of integrating with the Rocket core.

Deliverables

This lab is more focused on showing you how useful ideas for your projects than having you write your own solutions. As a result the deliverables should follow directly from following the instructions in this document. This lab is due **Tuesday, November 25 at 9:30AM**. The deliverables for this lab are:

- (a) written answers to the questions given at the end of this document checked into your git repository as `writeup/lab4-report.pdf` or `writeup/lab4-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.

Build Infrastructure

The Lab 4 template files are split into two directories. `lab4` contains the testharness and build infrastructure to construct your sha3 accelerator as a standalone block. `lab4-ref-chip` contains the build files for the entire reference chip, with the accelerator source simlinked in. Begin by working in the `lab4` section.

Building an SRAM

The version of the accelerator in the `lab4` folder contains a new parameter `buffer_sram` which when true will use an SRAM for the memory buffer.

Look into the changes in the source and generated code for your accelerator when building a configuration that uses the SRAM. Building a configuration like this has been scripted for jackhammer in `scripts/dc.ex` but can be done manually following those same steps:

```
% make CONFIG=DefaultConfig9 prm_vlsi SHELL=/bin/bash
% source build/vlsi/generated-src/sha3.DefaultConfig9.sh
% make CONFIG=DefaultConfig9 vlsi SHELL=/bin/bash
```

The most important differences are seen in the `build/vlsi/generated-src` folder which now include a `.conf` file that explains which modules are being instantiated as memories and what chisel believes the number of read and write ports is. The number of ports is important becuase our SRAM generator is limited in the number of ports it can create so it is often best to constrain yourself to either a single combination read/write port (for 6T SRAMs) or one read and one write port (for 8T SRAMs).

Additional instructions to create different sized SRAMs have been posted on piazza.

Programming Your Accelerator in RISC-V Assembly

Now that your accelerator is working well on its own, we can begin the process of connecting it to the Rocket core and driving it with actual assembly code. Before we begin to connect any hardware, we want to make sure that we have a good reference for how our accelerator should behave when instructions are passed to it. You will modify `spike`, the RISC-V ISA simulator, so that it understands the custom assembly instruction that we have defined for your accelerator. Then you will write programs that use a combination of C and assembly to drive your accelerator, and run them in `spike` to make sure that the code compiles and behaves as expected.

Rebuilding the ISA Simulator

The current executable for `spike` is configured with the default RISC-V instruction set as defined by the ISA specification. It has also been extended with a custom RoCC instruction called `dummy` that simply reads and writes to memory through the accelerator. You can check that the `spike` executable can handle the dummy instruction by calling it without any additional arguments:

```
% spike --extension=dummy
  usage: spike [host options] <target program> [target options]
  ...
```

It should just print the standard usage information to the command line. However, if you try to load your sha3 acceleration extension, it will complain:

```
% spike --extension=sha3
  unknown extension sha3!
```

You will need to install a new version of `spike` that includes a definition of the `sha3` extension.

From your top-level directory, go to `lab4/riscv-isa-sim`. This directory contains the source code for `spike`. A file defining the functionality of the `sha3` instruction has been implemented for you in `riscv/sha3-rocc.h`. Open up this file, take a look, and uncomment the implementation.

```
...
class sha3_rocc_t : public rocc_t
{
 public:
  const char* name() { return "sha3"; }

  reg_t custom0(rocc_insn_t insn, reg_t xs1, reg_t xs2)
  {
    switch (insn.funct)
    {
      case 0: // setup: maddr <- xs1; msize <- xs2
...
```

This file should contain C code that defines the sha3 instructions within spikeś infrastructure. To write this type of code you need to explicitily call out accesses to the emulated machiens memory with the functions:

```
p->get_mmu()->load_uint64(reg)
p->get_mmu()->store_uint64(reg,val)
```

Currently this is still a work in progress for sha3 but a fully functional accelerator can be seen in *riscv/dummy-rocc.h*.

This file contains C code that defines the functionality of our new *Custom0* instruction named "sha3". Based on the value of the *funct* field, the code defines two different cases, representing the two sha3 accelerator instructions. In particular, the ADDR instruction demonstrates how to setup accelerator state, and the LEN-START instruction shows how to begin computation.

Although our sha3 accelerator only needs two instructions it would be straightforward to add more instructions, either with more cases for the custom0 instructions or by adding a *Custom1* function. In addition, looking at the dummy accelerator spike implementation may provide other useful examples for RoCC instructions, including ones that return values to the processor through registers rather than through memory.

With the fully defined *Custom0* instruction, you must include the new source file in the existing infrastructure. The steps to do this are below and you should follow along noting that the changes have been made to the files. Open the file `spike/extensions.cc` and add the line `#include "sha3-rocc.h"` near the top of the file. In addition you need to add a line to register the extension here.

```
% REGISTER_EXTENSION(sha3, []() { return new sha3_rocc_t; })
```

Then open `riscv/riscv.mk.in` and add the line `sha3-rocc.h` to the `riscv-hdrs` variable.

Now you should be able to build a new version of `spike` that includes the `sha3` instruction. From the `riscv-isa-sim` directory, run the following commands (make sure to point at your own install path):

```
% export INSTALL_DIR = /scratch/cs250-xx/lab4-ref-chip/install
% mkdir build
% cd build
% ../configure --prefix=$INSTALL_DIR --with-fesvr=$RISCV
% make
% make install
```

You've installed your new version of `spike` to a directory created in `lab4-ref-chip`. To launch that version of spike, you can either call it explicitly, or add that location to your path:

```
% export PATH=/scratch/cs250-xx/lab4-ref-chip/install/bin:$PATH
```

You can use `which spike` to confirm that you are running the correct version. Running the command `spike --extension=sha3` should no longer throw an error. Note that you must add the above line to your `.bash_profile` or `.bashrc` if you want to automatically point to your new version of `spike` when you start a new terminal session.

Writing C/Assembly Drivers

Now it's time to write some code that exercises the new custom instruction. We have provided several template files that you can use to write programs for your accelerator. They can be found in `lab4-ref-chip/sha3/test`.

Before using the `sha3` instruction, you will need to define a software reference that can check if it is operating correctly. This reference is defined in the file `sha3.h`, notice the function that can check the sha3 hashs.

Now take a look at `sha3-sw.c`. This is the software version of the sha3 acceleration code; it calls the function you just wrote to perform hashes and then "checks" the result with assert statements. Obviously, it is a bit silly to use the same function to calculate and check each value, but this structure parallels the accelerator code that you will write shortly and is useful for direct runtime comparison between the two.

Compile and run the program:

```
% riscv-gcc sha3-sw.c -I. -o sha3-sw.rv
% spike pk sha3-sw.rv
```

You'll noticed that we've added an additional `pk` argument to the call to `spike`. `pk` is the RISC-V *proxy kernel*, a sort of lightweight operating system that provides some important features for our C runtime environment. In particular, it provides support for system calls like `assert` and `printf`, so that you can call these functions without having to include extra libraries in your program compilation.

Once you have looked into the software implementation of sha3, open the final C file, `sha3-rocc.c`. This is where you will actually call the `sha3` instruction to evaluate the hashes. This file is structured similarly to the software implementation; it checks the same tests as the previous file. In this case, however, the C code should call the `sha3` instruction instead of the functions defined in `sha3.h`.

Inline assembly instructions in C are invoked with the `asm volatile` command. Before the first instruction, and after each sha3 instruction, the `fence` command is invoked. This ensures that all previous memory accesses will complete before executing subsequent instructions, and is required to avoid mishaps as the Rocket core and coprocessor pass data back and forth through the shared data cache. (The processor uses the "busy" bit from your accelerator to know when to clear the fence.) A fence command is not strictly required after each custom instruction, but it must stand between any use of shared data by the two subsystems.

The custom commands are invoked with a particular syntax:

```
asm volatile ("custom0 0, %[maddr], %[msize], 0" : :
                          [maddr]"r"(&t1_data), [msize]"r"(t1_size));
```

The `custom0` assembly instruction takes four arguments: *rd*, *rs1*, *rs2*, and *funct*. After the instruction is defined (in quotes), the parameters are passed in after the double colon (`::`). Note that *rs1*, which is an address in memory, should pass in a pointer to an array (rather than passing in a variable directly). Fortunately, you do not need to worry about supplying particular registers to the instruction; the C compiler will make sure that the variables are allocated to registers appropriately.

Read the inline assembly instructions for the test to understand how they work. Once you are finished, compile and run your program:

```
% riscv-gcc sha3-rocc.c -I. -o sha3-rocc.rv
% spike --extension=sha3 pk sha3-rocc.rv
```

You are welcome to write a shell script or Makefile to automate the commands to compile and run your code.

Accelerating RISC-V Rocket

Now that you have simulated correct operation of your processor with your sha3 accelerator included, you are ready to attach your accelerator as a coprocessor of the Rocket core.

Your accelerator scala files have already been symlinked into the `sha3` directory. The reference chip build system has recently been enhanced to automatically discover additional project directories, so simply including the `sha3` folder will cause it to be built along with the rest of the components.

Now you need to wire in your accelerator. Open the file `src/main/scala/PrivateConfig.scala`. This is the top-level configuration file for the entire design. This version of the configuration file includes a few configurations that use your accelerator. It should look very familiar as it is similar to the way the accelerator configuration is specified which is one of the benefits of jackhammer. The most notable change is one line that tells the chip generator to instantiate a RoCC accelerator or not:

```
...
```

```
case BuildRoCC => Some(() => (Module(new Sha3Accel, { case CoreName => "Sha3" })))
...
```

Now the Rocket processor will connect your accelerator.

RoCC and the C Emulator

To make sure everything has been wired up correctly, build the C emulator for the processor (in `emulator`) and make sure that it compiles without error. Now you should be able to run the same programs that you simulated on the emulated processor:

```
% ./emulator pk ../sha3/tests/sha3-sw.rv
% ./emulator pk ../sha3/tests/sha3-rocc.rv
```

This will run considerably more slowly than `spike`, but should still produce the correct output.

Note that the `printf` and `assert` statements in your C code add considerable overhead to the program execution. There are copies of the two C files named `sha3-sw-bm.c` and `sha3-rocc-bm.c`. These versions have removed all of the `printf` and `assert` statements to speed up the runtime. You can execute these compiled binaries on `spike` to make sure that they run, but no outputs will print to the command line. Instead, you can use these files to benchmark the software implementation of sha3 against your hardware-accelerated implementation.

The following commands will run your benchmark tests on the Rocket C emulator, recording some of the state at each cycle to a file:

```
% ./emulator pk ../sha3/tests/sha3-sw-bm.rv +dramsim +verbose \
        3>&1 1>&2 2>&3 | riscv-dis > sha3-sw-bm.out
% ./emulator pk ../sha3/tests/sha3-rocc-bm.rv +dramsim +verbose \
        3>&1 1>&2 2>&3 | riscv-dis > sha3-rocc-bm.out
```

Compare the cycle counts in each output file. Feel free to automate this process with a shell script, or by modifying the emulator Makefile.

RoCC and the VLSI Tools

Next you should push your accelerated Rocket processor all the way through the flow. To simulate the Verilog RTL, build the `simv` as normal. To run your program, you can use the following command:

```
./simv-Sha3CPPConfig -q +ntb_random_seed_automatic +dramsim +verbose +max-cycles=100000000 $
```

This will print the same statements to the terminal and verify that your program still runs correctly. For post-synthesis and post-PAR simulation, you should stick to your "benchmark" programs only, as they will take quite some time to complete, and the overhead of `printf` and `assert` statements would slow simulation time even further. To run post-synthesis gate-level simulation, use the previous command from the `vlsi-sim-gl-syn` directory.

To generate switching activity for Primetime power simulation, generate the simulator and run it with:

```
% ./emulator-Sha3CPPConfig-debug +dramsim +max-cycles=100000000 +verbose -vsha3.vcd $RISCV/r
% vcd2saif sha3.vcd sha3.saif
```

This will generate an `saif` file that you can use to run Primetime.

Submission and Writeup

1. Based on the cycle counts of the two benchmarks, did your coprocessor outperform your software implementation of sha3? By how much? Does this match your expectations?

2. Based on the ICC average power numbers, the clock period, and the number of cycles needed to execute each benchmark, calculate total energy usage in each case. How much energy is saved by using an accelerator?

Submission

To complete this lab, you should commit the following files to your private Github repository:

- The files included in the template repository.
- Your answers to the questions above, in a file called `writeup.txt` or `writeup.pdf`.

Some general reminders about lab submission:

- 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.