From Gates to Compilers: Putting it All Together

CS250 Laboratory 4 (Version 102413)
Written by Ben Keller

Overview

In this lab, you will build upon the matrix sum accelerator you wrote in Lab 3. You will use your accelerator to explore the one dimension of the design space by scaling voltage to trade off energy and speed. 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.

Deliverables

This lab is due **Friday, October 25 at 2PM**. The deliverables for this lab are:

- (a) your working Chisel RTL checked into your private git repository at Github
- (b) Reports (only!) generated by DC Compiler, IC Compiler, and PrimeTime PX checked into your git repo
- (c) working C code for matrix sum
- (d) 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.

An Improved Sum Accelerator

The core functionality of your Lab 4 sum accelerator will remain the same as the accelerator you wrote in Lab 3: your hardware will cache partial sums to increase the speed of row and column summations of a matrix stored in memory (see the Lab 3 document for details). However, you will make one important change to your accelerator. In Lab 3, your accelerator returned its result to a register. In Lab 4, it should instead return the result to an address in memory. This will allow the processor much more flexibility in accelerating large matrices without having to clear the entire register file so results can be returned.

Modified Sum Accelerator Instructions

Because of the additional information needed to define the matrix operations in your improved accelerator, the ROWSUM and COLSUM commands are defined somewhat differently than they were in Lab 3. Instead of setting a destination register, they define a memory address to which the calculated sum should be returned:

- **opcode** is set to 0 for all matrix sum instructions.
- **xd** is set low, as a return is no longer expected from the ROWSUM or COLSUM commands.
- **xs1** is set high. **rs1** contains the (0-indexed) row or column to be summed.
- **xs2** is set high. **rs2** contains the memory address to which the result should be written.
- **funct** is set to 1 for a ROWSUM command or 2 for a COLSUM command.

The SETUP command remains the same as in Lab 3. You can also assume the same access pattern of ROWSUM and COLSUM commands. Note that we will set `maxSize=64` throughout this assignment.

Writing to Data Memory

To perform a write to memory instead of a read, simply set the command bits (`io.mem.req.bits.cmd`) to 1 instead of 0. You can then set the request data bits to the data you'd like to be read to the given address. You can provide a tag for your read requests just as for write requests. This tag will be returned through the memory response bus once the write has completed. Be sure that your accelerator asserts its "busy" bit until all of its writes have completed, as this is the only way that the RISC-V processor can respect the memory fences that will surround the assembly driving your accelerator.

Build Infrastructure

The Lab 4 template files are split into two directories. `lab4-accel` contains the testharness and build infrastructure to construct your sum accelerator as a standalone block. `lab4-proc` contains the build files for the entire reference chip, with the accelerator source simlinked in. Begin by working in the `lab4-accel` section. You should use the source code from Lab 3 as a starting point.

The Chisel and Verilog testharnesses have been modified based on the change described above. Modify your Chisel code so that it once again passes the provided Chisel and Verilog tests.

Voltage and Frequency Scaling

A typical way to produce several design points for a digital design is to trade off energy and throughput by scaling the supply voltage. In the VLSI flow, this is accomplished by using standard cells characterized at different operating voltages. You will build your design using three different standard cell characterizations to explore this tradeoff.

You will be once again building the SRAM version of your design (the final build from Lab 3). However, the top-level Makefrag has been modified to accept parameters that set the voltage and frequency targets for synthesis. These two variables can now be set from the command line:

- `CLOCK_PERIOD`: The target clock period for the design. This value is passed to Design Compiler. If not specified, it defaults to 1 ns (1 GHz).
- `VOLTAGE`: The value of VDD used for the design. The three valid settings are: `STANDARD, MIDDLE, LOW`, corresponding to operating voltages of 1.05, 0.85 and 0.78 volts. DC uses this setting to choose which standard cells libraries to use. If not specified, it defaults to `STANDARD`.

Here is an example of how you might invoke `make` from the `vlsi/build/dc-syn` directory to synthesize one design point:

```
make CLOCK_PERIOD=2.0 VOLTAGE=LOW
```

Note that these variables must also be set for later stages of the design (ICC, Primetime, gate-level simulation). You can effect this by parameterizing your call to `make` as shown above. If you set the clock period or voltage to different values for different stages of the build process, the tools may throw errors or give misleading results.

Start by sweeping different clock periods for each voltage level. You should try the following clock periods to begin: 0.6, 0.8, 1.0, 1.25, 1.5, 2.0, 2.5. Automate this process by writing a shell script that invokes `make` with different command line arguments. Write a python script to gather power, area, and timing estimates from the reports generated by Design Compiler. Your goal is to find the fastest possible operating frequency for each VDD. Based on the post-synthesis results, pick the smallest clock period for each voltage level that meets timing and push it all the way through the flow.

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 sum acceleration extension, it will complain:

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

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

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

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

  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 contains C code that defines the functionality of our new *Custom0* instruction named "sumacc". Based on the value of the *funct* field, the code defines three different cases, representing the three sum accelerator instructions. Two of the instructions have already been written. In particular, the ROWSUM instruction demonstrates how to load and store data to memory. Write the C code that defines the COLSUM instruction to complete the file.

Once you have fully defined the *Custom0* instruction, you must include the new source file in the existing infrastructure. Open the file `riscv/extension.cc` and add the line `#include "sumacc-rocc.h"` near the top of the file. Then open `riscv/riscv.mk.in` and add the line `sumacc-rocc.h` to the `riscv-hdrs` variable.

Now you should be able to build a new version of `spike` that includes the `sumacc` 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-proc/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-proc`. 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-proc/install/bin:$PATH
```

You can use `which spike` to confirm that you are running the correct version. Running the command `spike --extension=sumacc` 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-proc/sumaccel/test`.

Before using the `sumacc` instruction, you will need to define a software reference that can check if it is operating correctly. In the file `sum.h`, fill out the two functions that can check rowsums and colsums.

Now take a look at `sumacc-sw.cpp`. This is the software version of the sum acceleration code; it calls the functions you just wrote to perform rowsums and colsums, 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.

In the software test, two tests have been filled out. Add a third test of your choosing in the space provided. Then compile and run the program:

```
% riscv-gcc sumacc-sw.cpp -I. -o sumacc-sw.o
% spike pk sumacc-sw.o
```

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 written and tested your software implementation of rowsum and colsum, open the final C file, `sumacc-rocc.cpp`. This is where you will actually call the `sumacc` instruction to evaluate rowsums and colsums. 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 `sumacc` instruction instead of the functions defined in `sum.h`.

The assembly for the first test has already been completed for you. Inline assembly instructions in C are invoked with the `asm volatile` command. Before the first instruction, and after each rowsum or colsum 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.

Write inline assembly instructions to complete test 2. Then write test 3 to match the new software test you defined in `sumacc-sw.cpp`. Once you are finished, compile and run your program:

```
% riscv-gcc sumacc-rocc.cpp -I. -o sumacc-rocc.o
% spike --extension=sumacc pk sumacc-rocc.o
```

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

Note: We are currently seeing some issues simulating matrix sizes larger than 32 on the integrated Rocket/coprocessor hardware, so stick to matrices below this size in your tests for the time being.

If anything is amiss, one of the `assert` statements should alert you to the problem. If this occurs, you either have a bug in your inline assembly code, your C code, or your definition of the `sumacc` extension. Be sure to reconcile any errors so that all tests pass.

Accelerating RISC-V Rocket

Now that you have simulated correct operation of your processor with your sum accelerator included, you are ready to attach your accelerator as a coprocessor of the Rocket core. To do this, you must tell the Chisel build infrastructure about your additional source files, and configure the processor so that it wires to your RoCC.

Your accelerator scala files have already been symlinked into the `sumaccel` directory. In the root `lab4-proc` directory, open the file `project/build.scala`. This file defines the various source files for the project (among other things). Find the part of the file that defines the various sub-projects of the design (e.g, chisel, hardfloat, hwacha). Add the following line to define your `sumacc` project:

```
...
```

```
lazy val sumaccel = Project("sumaccel", file("sumaccel"),
  settings = buildSettings) dependsOn(rocket)
...
```

Additionally, change the dependency of the top-level `referencechip` from `rocket` to `sumaccel`. This will ensure that your accelerator is compiled before the top-level module is assembled.

Now you need to wire in your accelerator. Open the file `src/main/scala/ReferenceChip.scala`. This is the top-level source file for the entire design. Near the top of the file, add the line:

```
import SumAccel._
```

Then find the `RocketConfiguration` variable and add in your accelerator like so:

```
...
val rc = RocketConfiguration(tl, ic, dc, fpu = HAS_FPU,
            rocc = (c: RocketConfiguration) => (new SumAccel(c, maxSize=64)))
...
```

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 ../sumaccel/tests/sumacc-sw.o
% ./emulator pk ../sumaccel/tests/sumacc-rocc.o
```

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. Make copies of your two C files, and name them `sumacc-sw-bm.cpp` and `sumacc-rocc-bm.cpp`. Then remove all of the `printf` and `assert` statements from each file, as well as the first two `include` statements at the top of each file. Then compile these new benchmark files. You can execute the 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 matrix sum 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 ../sumaccel/tests/sumacc-sw-bm.o +dramsim +verbose \
        3>&1 1>&2 2>&3 | riscv-dis > sumacc-sw-bm.out
% ./emulator pk ../sumaccel/tests/sumacc-rocc-bm.o +dramsim +verbose \
        3>&1 1>&2 2>&3 | riscv-dis > sumacc-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 +ntb_random_seed_automatic +vcs+initreg+1382157876 \
        +dramsim +verbose +max-cycles=100000000 +coremap-random \
        +argv="pk ../../../sumaccel/tests/sumacc-rocc.o" 2> sumacc-rocc.out
```

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 following command:

```
% ./simv -ucli -do ./+run.tcl +dramsim +verbose +max-cycles=100000000 \
        +coremap-random +argv="pk ../../../sumaccel/tests/sumacc-rocc-bm.o" \
        2> summacc-rocc-bm.out
```

To generate switching activity for Primetime power simulation, generate the simulator with `make simv-debug`, and then run:

```
% vcd2saif -input sumacc-rocc-bm.vcd -pipe './simv-debug -ucli -do ./+run.tcl \
        +verbose +vcdfile=sumacc-rocc-bm.vcd +max-cycles=100000000 \
        +coremap-random +argv="pk ../../../sumaccel/tests/sumacc-rocc-bm.o"' \
        -output sumacc-rocc-bm.saif 2> sumacc-rocc-bm.out \
```

This will generate an `saif` file that you can use to run Primetime. Be sure to find power numbers for both the software and RoCC benchmarks. You can use `tail -f` (from another screen or window) to monitor the progress of the simulation as the `.out` file is generated.

Note that the target clock period for synthesis has been set to 2ns, and the simulation clock period has been set to 4ns. This should ensure correct simulation and help the tools to more easily meet timing. Use a simulation clock period of 4ns when calculating energy consumption of the system.

Submission and Writeup

Part 1 - Standalone Accelerator

Your writeup for the standalone accelerator should include two tables: one with all your post-synthesis results and another with your post place-and-route results. Each table should include timing (critical pack slack - i.e., whether or not the design met timing), area, and power numbers, as well as a breakdown of the types of cells (HVT and RVT) used.

Based on this data, write up a summary of your design space exploration, making sure to address the following questions:

1. Of the three designs you pushed all the way through the tools, which is the most energy efficient? (Be sure to compare energy, not power.)

2. Is there any area difference between the three implementations? Does this match your expectations?

3. While the standard cells are characterized at multiple voltages, the SRAM macro is characterized at only 1.05V, and so the tools used the same SRAM power and timing model for all three implementations. (This is common in real designs, in which SRAMs may not be fully characterized across a wide range of voltages.) Assuming SRAM scales similarly to logic, how might this inaccuracy affect the energy and clock period numbers you reported above?

Part 2 - Processor and Coprocessor

Use Python scripts to fill in the following table for your whole-system design:

	Area	ICC Power	PT Power (sw-bm)	PT Power (rocc-bm)
Entire Design				
SumAccel				

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

2. Based on the PrimeTime 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:

- Your working Chisel code.
- Your four working C files (as well as `sum.h`).
- Your modifications to the reference chip build files to include the accelerator.

- The `reports` directories from DC, ICC, and Primetime for each of your builds. You do not need to commit every DC build for the standalone accelerator; just commit the ones that you pushed through to ICC.
- Your answers to the questions above, in a file called `writeup.txt` or `writeup.pdf`.

Some general reminders about lab submission:

- 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.