Adding a user-defined RISC-V instruction to the FORCE-RISCV ISG

1. Introduction

This is a tutorial on how to add support to **FORCE-RISCV** for a user-defined instruction.

FORCE-RISCV is an *Open Source* Instruction Set Generator (ISG) targeting the RISC-V microprocessor architecture. **FORCE-RISCV** may be used to generate random instruction tests to aid in the verification of a RISC-V microprocessor implementation. Test templates written in *Python* are used to direct **FORCE-RISCV** in the generation of random tests, and indeed much of the *front-end* code of **FORCE-RISCV** is implemented in *Python*. The *back-end* code portion of **FORCE-RISCV**, ie, the core components, are written in C++.

To generate architecturally correct tests, a RISC-V compliant simulator library must be used with **FORCE-RISCV**. The (open source) RISC-V simulator library **FORCE-RISCV** currently employs is the UC Berkeley implemented *Spike* simulator. The *Spike* simulator sources and additional source code implemented to interface with FORCE-RISCV is known as *Handcar* (or the *Handcar* simulation library, or in some cases the *Handcar* shared object). The *Handcar* source code and build scripts may be found in the `~/force-riscv/handcar` sub-directory).

In this tutorial we will add support for a new RISC-V instruction to **FORCE-RISCV**, and to the *Spike* simulator version supplied as part of the **FORCE-RISCV** distribution.

2. Initial download, build, testing of FORCE-RISCV

First download and build **FORCE-RISCV**, and run the standard suite of regression tests*…

```
git clone https://github.com/openhwgroup/force-riscv.git
cd force-riscv

export FORCE_CC=/usr/bin/g++
export FORCE_PYTHON_VER=3.6
export FORCE_PYTHON_LIB=/usr/lib/x86_64-linux-gnu/
export FORCE_PYTHON_INC=/usr/include/python3.6
cd force-riscv

make
make tests

./utils/regression/master_run.py
./utils/regression/unit_tests.py
```

*Note: For the purposes of this tutorial we'll assume you will have downloaded and compiled **FORCE-RISCV** in your home directory (but there is no requirement that you do so). In subsequent discussions, the **FORCE-RISCV** development directory will be assumed to be `~/force-riscv`.

Exercise 1: Download and build FORCE-RISCV.

FORCE-RISCV has been downloaded, compiled, and tested, using a precompiled version of *Handcar*. We will need to explicitly configure and compile the *Handcar* (shared object) in preparation for adding a new instruction to *Handcar*.

3. Configuration, build, install of the *Handcar* simulation library

To configure/build/install the *Handcar* simulator shared object*:

```
cd handcar
./regenerate_and_build.bash
```

*Note: The *Spike* configure process requires that the *device-tree-compiler* utility be installed. If the build (*Spike* configure portion) halts due to device-tree-compiler not found, you may need to install the device-tree-compiler. On *Debian* based Linux systems such as *Ubuntu*, to install the device-tree-compiler use:

```
sudo apt-get install device-tree-compiler
```

Execution of the `regenerate_and_build.bash` script causes the *Spike* simulator sources to be downloaded from *github* and configured, the **FORCE-RISCV** handcar sources to be merged in with the *Spike* sources, a new *handcar* shared object (`bin/handcar_cosim.co`) to be compiled and then finally, copied to `the ~force-riscv/utils` directory.

Exercise 2: Build/install the Handcar shared object.

4. Defining new RISC-V instruction, adding same to *Handcar/Spike*

As an example instruction to be added to **FORCE-RISCV**, lets implement a *rotate* instruction:

```
ror r1, r2, #imm
```

The contents of the *r2 GPR* will be *right rotated* by the number of bits specified in the *immediate* field, and the result copied to *r1.*

Lets start by adding this new instruction to the *Spike* simulator (and as a result to *Handcar*). We'll use the *logical shift right instruction* (*srli*) as a template for our new instruction. The process we used is detailed below…

```
cd ~force-riscv/handcar
mkdir new_instr_tutorial
cp src/srli.cc new_instruction_tutorial/rori.cc
cp inc/insns/srli.h new_instruction_tutorial/rori.h
cp inc/encoding.h new_instruction_tutorial/encoding.h
cd new_instr_tutorial
vi rori.cc                  # see Note 1 below.
vi rori.h                   # see Note 2 below.
vi encoding.h               # see Note 3 below.
cp rori.cc ../src           #
cp rori.h ../inc/insns      # see Note 4 below.
cp encoding.h ../inc        #

cd ..
make -j8 handcar            # compile handcar.so with new instructions,
cp bin/handcar_cosim.so ../utils/handcar # install


# rerun Force regression. We don't expect any fails, still…
cd ..
./utils/regression/master_run.py
```

Note 1: Edit accordingly. The new instruction is *rori*.
Note 2: Edit accordingly – augment the shift right logic to achieve a *rotate right* operation instead.
Note 3: Add opcode and mask for the *rori* instruction. Pick an unused instruction according to the RISC-V Instruction Set Manual, Volume I,Chapter 26 . We chose `0x502b`.
Note 4: For the purposes of this tutorial, we created the `new_instr_tutorial` sub-directory. How to maintain user-defined *Spike* modifications are beyond the scope of this tutorial.

Exercise 3: Add a new instruction to the FORCE-RISCV/Spike/handcar build.

5. Adding the new instruction to FORCE-RISCV

Okay, so in theory the simulator now supports our new *rori* instruction. Lets add that new instruction to **FORCE-RISCV** itself…

The RISC-V instruction and CSRs currently supported by **FORCE-RISCV** are defined in a set of architecture-specific *XML* files located in the `~force-riscv/riscv/arch_data` sub-directory. These architecture-specific files include:

Architectural definition file	Purpose
`app_registers.xml`	Application (General Purpose) Register definitions including floating point and vector
`system_registers.xml`	Control and Status Register definitions
`riscv_instructions.xml`	Base Integer instructions and floating point instruction definitions

`v_instructions.xml`	(TBD) Vector Instruction definitions
`priv_instructions.xml`	Privileged Instruction definitions
`c_instructions.xml`	Compressed Instruction definitions

We are adding a new 64-bit Integer instruction, based on an existing instruction. The only file that needs to be updated is the `riscv_instructions.xml` file. Do not however edit this file directly. This file and the others listed above are *auto-generated* from *Python* scripts.

cd into the `~/force-riscv/utils/builder/instruction_builder/riscv` sub-directory.

This directory contains scripts and a *Makefile* that are used to populate the `~force-riscv/riscv/arch_data` directory with instruction definition files. The main input files are located in the `input` sub-directory. There is one instruction *starter* file per instruction definition file to be generated.

The starter file that we need to edit to include the *rori* instruction is the `input/riscv_instructions_starter.xml` file. In this file are entries for each RISC-V Base Integer and Floating Point instruction currently supported by FORCE-RISCV. We have derived our new *rori* instruction from the *srli* instruction. Copy and edit the srli (*extension* = RV64) instruction entry to create an entry for the *rori* instruction. The *const_bits* field defines the fixed bits within an instruction encoding. Edit the newly created *rori* entry/*const_bits* field value to reflect the *rori* opcode chosen previously, noting that all fixed bit fields are defined in the const_bits and that these fixed value fields are not necessarily contiguous.

Save the starter file.

Next edit the `adjust_instruction_by_format.py` script. Add the *rori* instruction to the list of instructions to be updated in the `adjust_shamt_rs1_rs` method. For the list of instructions defined in this method, a form annotation is added based on the *shamt* operand to indicate that the instruction instance is RV32I or RV64I. The rori instruction we are adding in is intended to operate on 64-bit register and thus is of form RV64I.

Save this file.

Run *make* to build and install new **FORCE-RISCV** instruction files.

Exercise 4: Edit FORCE-RISCV instruction starter and adjustment files to include the rori instruction, build/install FORCE-RISCV instruction definition files.

6. Rerunning top-level **FORCE-RISCV** regression with new instruction included

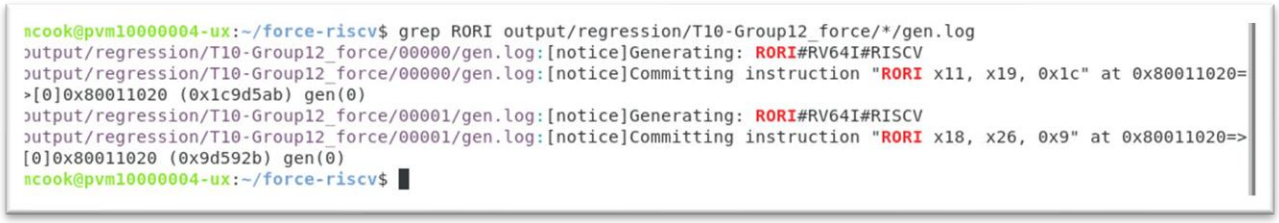
At this point the new rori instruction has been added to *Spike*, and to **FORCE-RISCV**. The **FORCE-RISCV** regression suite should be rerun to insure no errors have been introduced by our changes. Before doing so, the instruction-specific regression tests need to be updated to include our new instruction.

cd to the main directory, (re)make the regression test suite including and in particular the instruction-specific tests:

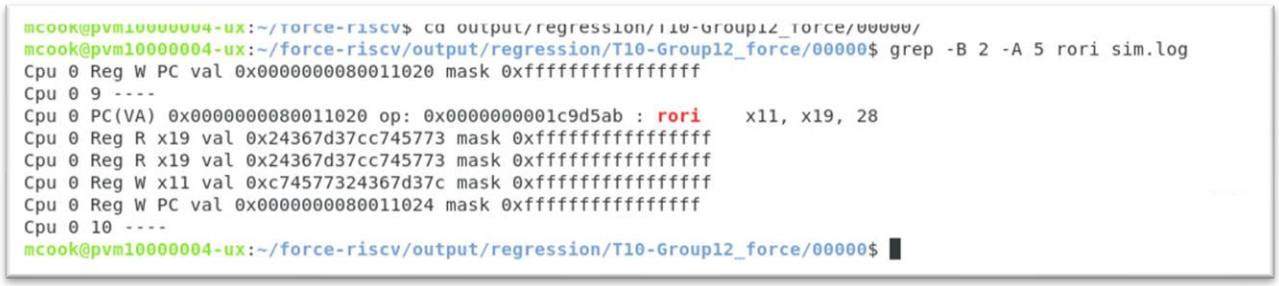
```
cd ~/force-riscv
make
make tests
```

One of the instruction specific regression tests should now contain an entry for the rori instruction:

Run the FORCE-RISCV standard regression, as done is previous steps. One or more tests generated/simulated during this regression will include instances of the rori instruction:



For the generated tests that do include instances of the rori instruction, the simulation log will also indicate that the *rori* instruction is being simulated. cd into one of the regression output directories that contain instances of the *rori* instruction, and view the simulation log, looking for an instance of the *rori* instruction. The log should look similar to what is displayed here:



From the simulation log we can see that the *rori* instruction we added to **FORCE-RISCV** and to the *handcar* simulation library is functioning correctly. If the **FORCE-RISCV** generated instruction encoding had not been recognized by the simulator, an exception would have occurred (typically with RISC-V exception code indicating an Illegal Instruction exception).*

*Note: If an Illegal Instruction exception did occur, the **FORCE-RISCV** supplied exception handlers would (unless otherwise disabled) handle the exception by causing the offending instruction to be skipped. Thus it is important to scrutinize the simulation logs after first implementing a new instruction, to confirm that the new instruction is operating as expected.

The implementation of a 32-bit version of the rori instruction is left as an additional exercise to the reader.

Exercise 5: Implement the 32-bit version of the *rori* instruction in *Handcar* and FORCE-RISCV.