Milestone 2 Extra — Computer Architecture

Single Cycle RV32I ISA Tests

Hai Cao

rev 1.0.0

Contents

Abstract

If you come across any errors or have suggestions for improving this document, please email the TA: cxhai.sdh221@hcmut.edu.vn with the subject "**[CA203 MS2 TEST FEEDBACK]**"

1 Introduction

This document serves as a comprehensive guide for conducting functional verification of the RV32I instruction set, as outlined in the requirements for Milestone 2. The objective is to ensure that the implemented design meets the expected functional behavior, leveraging a systematic testing environment. The verification environment provided is a straightforward functional testbench, where students' designs act as DUT (Device Under Test). Modules for driving stimuli and collecting results, including driver and scoreboard components, are provided. Students are expected to organize their source code correctly, navigate to the simulation directory, and execute the provided scripted makefile for automated simulation.

The following are key aspects to understand before proceeding with the simulation:

1. The test environment monitors `o_pc_debug` and `o_io_ledr` signals to determine test outcomes, indicating either a "pass" or "error" condition.
2. Due to the flexible memory mapping requirements, your design must support 32-bit load/store operations to the LEDR register, mapped to address `0x1000_0000`.

2 Environment Setup

Prior to commencing the test, students must copy the singlecycle test from the common directory to their home directory and navigate to it.

```
cd ~
cp -rf ~/common/sc-test .
cd sc-test
```

2.1 Project Directory Hierarchy

The project structure adheres to a hierarchical organization to facilitate efficient simulation and verification. Directories are structured as follows:

```
milestone2
|-- 00_src          # Verilog source files
|-- 01_bench        # Testbench files
|   |-- driver.sv
|   |-- scoreboard.sv
|   |-- tbench.sv
|   `-- tlib.svh
|-- 02_test         # Testing files
|   `-- isa.mem     # Hex file
|-- 10_sim          # Verilator
|   |-- flist
|   `-- Makefile
`-- 11_xm           # Xcelium
    |-- flist
    `-- Makefile
```

`00_src`	Place all SystemVerilog source files here.
`01_bench`	This directory provides insight into the testbench setup. Study its contents to understand the simulation environment.
`02_test`	Contains the file `isa.mem`, a hexadecimal file representing the instruction set test.
`10_sim`	This directory includes scripts for simulation using Verilator.

`11_xm` This directory contains scripts for simulation using Cadence Xcelium.

2.2 Memory Configuration

To ensure compatibility with the testbench, make the following modifications to your memory models:

- Since the testbench requires more than 4 KiB, your design must use an address width of at least 16 bits (supporting a memory size of 16 KiB). Hence, the top address for simulation should be at least `0x0000_7FFF`.
- The memory must be preloaded with the contents of `02_test/isa.mem` to provide test instructions.

2.3 File Setup for Verilator

To run simulations using Verilator, change into `10_sim` directory and edit `flist` file to include all relevant design files. For example, if your top-level module is named `singlecycle.sv`, include the entry:

```
./../00_src/singlecycle.sv
```

2.4 File Setup for Xcelium

To execute simulations using Cadence Xcelium, change into `10_sim` directory and edit `flist` file to include all relevant design files. For example, if your top-level module is named `singlecycle.sv`, include the entry:

```
./../00_src/singlecycle.sv
```

3 Simulation

With all setup completed, you must first access computing resource and run Makefile script.

1. Run "`srun -x11 -pty bash`". Without "–x11", you cannot use GUI.
2. If you use Verilator, navigate to `10_sim` and run "`make`". If you want to observe waveforms, you can use "`make wave`" to open GTKWave.
3. In case you use Xcelium, navigate to `11_xm` and run "`make`". If you want to observe waveforms, you can use "`make gui`" to open SimVision.

4 Simulation Results

4.1 Expected Behavior

A correctly functioning design should produce the expected output as below:

```
SINGLE CYCLE TESTS

add......PASS
addi.....PASS
sub......PASS
and......PASS
andi.....PASS
or.......PASS
ori......PASS
xor......PASS
xori.....PASS
slt......PASS
slti.....PASS
sltu.....PASS
sltiu....PASS
sll......PASS
slli.....PASS
srl......PASS
srli.....PASS
sra......PASS
srai.....PASS
lw.......PASS
lh.......PASS
lhu......PASS
lb.......PASS
sw.......PASS
sh.......PASS
sb.......PASS
auipc....PASS
lui......PASS
beq......PASS
bne......PASS
blt......PASS
bltu.....PASS
bge......PASS
bgeu.....PASS
jal......PASS
jalr.....PASS
malgn....ERROR
iosw.....PASS
```

```

END
```

Note that handling of misaligned memory addresses is not mandatory, and such scenarios may result in an error status.

4.2 Troubleshooting Common Issues

While the test is termed an "ISA Test," it is designed to validate functional correctness by integrating multiple instructions in each stage. This approach ensures robust verification rather than isolated instruction testing.

Hint: Ensure that the "trinity" of instructions — `beq`, `jal`, and `addi` — is correctly implemented, as errors in these instructions can cascade and cause other tests to fail.