Tai Lung RV32I Project

Validation plan

Authors:

Christian Shakkour, chrisshakkour@gmail.com

Shahar Dror, shdrth1@gmail.com

Git repository:

https://github.com/ChrisShakkour/RV32I-MAF-project

Release Information

DATE	AUTHOR	DESCRIPTION
02-FEB-2022	Shahar	Created v1.0 – first draft
06-FEB-2022	Shahar	Adds table of contents
08-MAR-2022	Chris	Document structure refining
01-APR-2022	Chris	Adds Content

Acknowledgements

Special thanks to our supervisor Amihai Ben David for contributing knowledge and experience throughout the development of this project. Many thanks to the VLSI Lab at the Technion - Israel Institute of technology for supporting and providing tools and resources that were an integral part of the development process.

About this document

In this document you will find information related to verifying the Core functionality and the compatibility to RISC-V ISA, this document will cover RTL verification plan and the ISA tests.

Table of Contents

Introduction

Tai Lung like any other project starts with Goals and specs, which hopefully at the End of the project these goals and specs are achieved. To make sure the goals and specs are achieved the Core is put under a long list of verification stages and tests. In this project the verification is separated into two main sections, (1) RTL correctness and verification (DV), (2) RISC-V ISA compatibility. Both are verified strictly in different methods and stages of the project.

Design Verification

Design verification is where one tests/verifies that the design outputs match the expected output given a set of inputs. It is the process where one confirms the desired functionality of a certain logic block or in a more sophisticated process the functionality of a system. This process is done in many ways, listed below the steps taken in verifying Tai Lung core.

Code Review

Code review is the first step in verification where one writes RTL code, and another looks briefly into the code to capture a few types of common bugs. Unwanted latches for example are usually the first thing to look for, nevertheless, connectivity and routing bugs can also be detected in the code review process. In this process functional bugs are a bit harder to catch.

Block-level simulation

In the block level simulation, we usually look for functional bugs like mismatched logic behavior, here one creates a testbench and provides a module with the required functional environment(clocks, reset, and other signals) and drives the inputs with a set of legal stimuli and monitors the outputs of this block, the motoring can be done in many ways, a waveform for example but not only, one can define the expected output and compare, a more formal way of doing so is by defining assertions. Most of the module in this project have a dedicated testbench.

Memory units

To verify the functionality of the behavioral model of the RTL memory we created a testbench that stimulates read and write transactions where we tested two major mechanisms. (1) one can write data to a given address and read the same data from the same address, (2) when reading/writing from/to an unavailable address an interrupt shall rise.

ALU

The ALU has been checked strictly because it's the beating heart of the CPU and it could be quite difficult to find a bug in the computational unit in system-level simulations where the data goes in different directions, memory, write back to reg-file, forwarding, and jump address calculation. So, to test the ALU we created 2 sets of arithmetic tests, one are simple arithmetic just to make sure the arithmetic works as anticipated like adding 3 and 5 together, or testing some of the simple arithmetic like XOR, OR, AND, NOT, and so forth. The second set of arithmetic tests is to cover the corner cases and trigger interrupts like triggering overflow bit and the sign bit. This set was strictly engineered following the RISC-V ISA manual to make sure we are creating legal commands and their computation is correct.

Branch comparator

The branch comparator has been tested with all the possible branching options both when taken and not taken, the test was handcrafted with the input signals that trigger each branch option available.

Register file

The register file is a smaller version of memory, so the testing was done via writing data to all the registers and reading the data to make sure every register is accessible except register x0 that is always Zero.

System-level simulation

System-level is where one simulates a system of multi block and validate a functional flow like boot flow and many others. A simple scenario is where two blocks have a handshake mechanism or simply a request and response flow, in system-level like in block level the system is provided with the desired environment (clocks and etc...) and a stimulus to trigger this flow, and again monitors the transactions between the blocks, the monitoring is bit more complex compared to block-level but the same mechanisms are used (expected output, assertions, and waveform).

CoreTop

The CoreTop is the design unit that putts the CPU and memory together, here the verification is done by running a functional flow and monitoring a few key features like the stack frame, the memory transactions, and the correctness of the test for example if we are running a sorting algorithm on an array of known numbers, we shall expect to see the array sorted properly. Some of the tests include many sorting algorithms, searching as well, Tower of Hanoi, Sudoku solver algorithm, Fibonacci, and so on. The goal was to cover iterative and recursive algorithms to stress many mechanisms at once, jump, branch, memory, ABI conventions, and the stack frame.

FPGA Emulation

Tbd in the next project…

ISA Compatibility

Compliance is not about RTL verification or trying to prove that the RTL is correct. Compliance is trying to ensure that the design functionality meets the RISC-V standard, how far should compliance go toward verification? Compliance is absolutely critical in steering the ecosystem in a consistent direction.

Instruction’s coverage

instruction stressing tests, for each instruction one can use the risc-v compatibility tool that generates a handful of scenarios and corner cases for each instruction in the ISA, in this project this tool hasn’t been enabled yet… will be added in the next project.

Monitoring Techniques

Here will be listing a few monitoring techniques that were developed for the verification process of this project.

Trackers

Trackers are helpful when debugging low level behavior as they provide us with lots of information that can be time-consuming to extract from the waveform, so certain information is dumped into a table. Some of the trackers we used are:

1- The stack pointer: tracking each time the stack pointer(sp) is changed.
2- Memory transactions: recording any read and write transactions.
3- Decoded instruction: tracking the instructions that go into the pipeline.

End of tests Checkers

End of test checkers are there to tell us the test has ran as expected, the test terminated and the data is correct. Some of the checkers used are:

1- Stack frame: making sure the stack frame is back to its original state Zero in size.
2- Exit status: making sure the test finished successfully.
3- Exit data: making sure the data is as expected. Ex after sorting and array.

Memory snapshot

Comparing the actual memory snapshot to an expected memory snapshot that has been extracted from a proven to work CPU with the same instructions support, believe it or not given the same program, same memory address map, same linker, same compiler, and same ISA support running on different RTL shall provide the same memory image when the test is done, for example if we run the test on a 5 stage pipeline architecture, single or multi cycle we will get the same memory image when the test is terminated.

Cover groups

Cover groups provide us with analysis data that can be used to determine flow bugs, architectural bottlenecks and drawbacks, and activity of a certain block or mechanism. Cover points weren't integrated in this project but will be added in the next one, following a few useful cover points:

1- Forwarding mechanism: counts the number of times a certain forwarding occurred.
2- Branch taken/not-taken: counts the number of taken and not taken branches.
3- Load/store commands: counts the occurrences of each load/store commands.

References:

Towards RISC-V compliance:

https://semiengineering.com/toward-risc-v-compliance/

https://riscof.readthedocs.io/en/latest/intro.html