ISA Formal Verification of RISC-V Processors

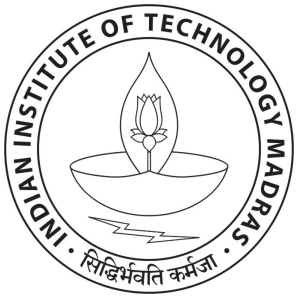
A THESIS

submitted by

KRISHAN PRAJAPAT

in partial fulfilment of requirements
for the award of the dual degree of

BACHELOR OF TECHNOLOGY AND MASTER OF TECHNOLOGY



DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS

May 2018

THESIS CERTIFICATE

This is to certify that the thesis titled **ISA Formal Verification of RISC-V Processors** submitted by **Krishan Prajapat**, to the Indian Institute of Technology, Madras, for the award of the degree of **Bachelor of Technology and Master of Technology**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. V Kamakoti
Project Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Prof. Shanthi Pavan
Project Co-guide
Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Place: Chennai

Date: 5th May 2018

ACKNOWLEDGEMENTS

I would like to begin by thanking my guide, Prof. V Kamakoti for all his help, support and patience throughout the course of my project work. His approach and dedication to research has been incomparable and has always inspired me to keep pushing myself. I also extend my heartfelt gratitude to my co-guide Prof. Shanthi Pavan for his guidance and constant encouragement throughout the project.

My special thanks to Dr. Neel Gala for helping and guiding me throughout the project. I am very grateful to him for providing his valuable time to guide me during the project. I also want to thank my lab mates Arjun Menon, Vinod G, Rahul B for helping me whenever I got stuck.

Finally, I thank my family for their support and constant encouragement.

ABSTRACT

KEYWORDS: ISA; RISCV; Formal Verification.

A processor is a complex system and so it becomes difficult to verify its correctness. In this thesis we implement a framework to formally verify the processor with RISC-V ISA using ISA Formal technique. ISA Formal is an end-to-end framework to detect bugs in the datapath, pipeline control and forwarding/stall logic of processors. ISA-Formal has proven to be especially effective at finding micro-architecture specific bugs involving complex sequences of instructions. An essential feature of this is that it is able to scale all the way from simple 3-stage microcontrollers, through superscalar in-order processors up to out-of-order processors. We have applied this method to C class processor of SHAKTI Processor family developed at IIT Madras. It is a 5-stage processor based on RISC-V ISA.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

ISA	Instruction Set Architecture
AAPG	Automatic Assembly Program Generator
RVFI	RISC-V Formal Interface
BSV	Bluespec SystemVerilog
IITM	Indian Institute of Technology, Madras
DUT	Device Under test

CHAPTER 1

INTRODUCTION AND OVERVIEW

A microprocessor designs apply many optimizations to improve performance: pipelining, forwarding, issuing multiple instructions per cycle, multiple independent pipelines, out-of-order instruction completion, out-of-order instruction issue, etc. All of these optimizations are supposed to be invisible to the programmer in a uniprocessor context: the overall effect should be the same as executing instructions one at a time in program order. But each of these optimizations introduces corner cases that potentially change the behavior and the different optimizations interact with each other in complex ways.

For traditional simulation-based verification to detect this defect, you would need a detailed understanding of the micro-architecture of that particular processor, of the corner cases caused by the forwarding paths, and of the kinds of errors one is likely to make in implementing forwarding control logic. Creating such tests is not only hard and unreliable, but it is also expensive because the tests would be specific to the particular micro-architectural choices in a processor and different tests must be created for each processor.

The processor design team of Reconfigurable and Intelligent Systems Engineering (RISE) Lab in the Computer Science Department of IIT Madras has been actively involved in research of The SHAKTI processor project. The SHAKTI processor project aims to build variants of processors based on the RISC-V ISA from UC Berkeley. The project will develop a series of cores, SoC fabrics and a reference SoC for each core family in BSV language. One such core variant is the C class processor .

This thesis describes the "ISA-Formal" verification technique to confirm that C Class processors correctly implement the Instruction Set Architecture (ISA) part of the architecture specification.

1.1 RISC-V ISA

RISC-V is an open instruction set architecture (ISA) based on established reduced instruction set computing (RISC) principles. In contrast to most ISAs, the RISC-V ISA can be freely used for any purpose, permitting anyone to design, manufacture and sell RISC-V chips and software. While not the first open ISA, it is significant because it is designed to be useful in modern computerized devices such as warehouse-scale cloud computers, high-end mobile phones and the smallest embedded systems. Such uses demand that the designers consider both performance and power efficiency. The instruction set also has a substantial body of supporting software, which avoids a usual weakness of new instruction sets.

The RISC-V ISA is defined as a base integer ISA, which must be present in any implementation, plus optional extensions to the base ISA. The base integer ISA is very similar to that of the early RISC processors except with no branch delay slots and with support for optional variable-length instruction encodings. The base is carefully restricted to a minimal set of instructions sufficient to provide a reasonable target for compilers, assemblers, linkers, and operating systems (with additional supervisor-level operations), and so provides a convenient ISA and software toolchain "skeleton" around which more customized processor ISAs can be built.

Each base integer instruction set is characterized by the width of the integer registers and the corresponding size of the user address space. There are two primary base integer variants, RV32I and RV64I, which provide 32-bit or 64-bit user-level address spaces respectively. RISC-V has been designed to support extensive customization and specialization. The base integer ISA can be extended with one or more optional instruction-set extensions, but the base integer instructions cannot be redefined. To support more general software development, a set of standard extensions are defined to provide integer multiply/divide, atomic operations, and single and double-precision floating-point arithmetic. The base integer ISA is named "I" (prefixed by RV32 or RV64 depending on integer register width), and contains integer computational instructions, integer loads, integer stores, and control-flow instructions, and is mandatory for all RISC-V implementations. The standard integer multiplication and division extension is named "M", and adds instructions to multiply and divide values held in the integer registers. The standard atomic instruction extension, denoted by "A" adds instructions

that atomically read, modify, and write memory for inter-processor synchronization. The standard single-precision floating-point extension, denoted by "F", adds floating-pointregisters, single-precision computational instructions, and single-precision loads and stores. The standard double-precision floating-point extension, denoted by "D", expands the floating-point registers, and adds double-precision computational instructions, loads, and stores. An integer base plus these four standard extensions ("IMAFD") is given the abbreviation "G" and provides a general-purpose scalar instruction set.

Key Features of the RISC-V ISA:

- Delivers a new level of software and hardware freedom on architecture in an open extensible way.
- Open ISA delivers easier support from a broad range of operating systems, software vendors and tool developers.
- The open source of hardware, RISC-V does not rely on a single supplier âĂŞ offers multiple suppliers, therefore, supports unlimited potential for future growth.
- No other ISA is architected like the RISC-V ISA, allowing for user extensibility of the architecture without breaking existing extensions or incurring software fragmentation

1.1.1 Instruction Length Encoding

The base RISC-V ISA has fixed-length 32-bit instructions that must be naturally aligned on 32-bit boundaries. However, the standard RISC-V encoding scheme is designed to support ISA extensions with variable-length instructions, where each instruction can be any number of 16-bit instruction parcels in length and parcels are naturally aligned on 16-bit boundaries. Figure 1.1 illustrates the standard RISC-V instruction-length encoding convention. All the 32-bit instructions in the base ISA have their lowest two bits set to 11. The optional compressed 16-bit instruction-set extensions have their lowest two bits equal to 00, 01, or 10. Standard instruction set extensions encoded with more than 32 bits have additional low-order bits set to 1, with the conventions for 48-bit and 64-bit lengths shown in Figure 1.1.

The base RISC-V ISA has a little-endian memory system, but non-standard variants can provide a big-endian or bi-endian memory system. Instructions are stored in memory with each 16-bit parcel stored in a memory halfword according to the implementationâĂŹs natural endianness. Parcels forming one instruction are stored at increasing

Figure 1.1: RISC-V instruction length encoding.

halfword addresses, with the lowest addressed parcel holding the lowest numbered bits in the instruction specification, i.e., instructions are always stored in a little-endian sequence of parcels regardless of the memory system endianness. The code sequence in Figure 1.2 will store a 32-bit instruction to memory correctly regardless of memory system endianness.

```
// Store 32-bit instruction in x2 register to location pointed to by x3.
sh   x2, 0(x3)    // Store low bits of instruction in first parcel.
srli x2, x2, 16   // Move high bits down to low bits, overwriting x2.
sh   x2, 2(x3)    // Store high bits in second parcel.
```

Figure 1.2: Recommended code sequence to store 32-bit instruction from register to memory. Operates correctly on both big- and little-endian memory systems and avoids misaligned accesses when used with variable-length instruction-set extensions.

1.2 Formal Verification

Formal verification is a technique used in different stages in ASIC project life cycle like front end verification, logic synthesis, post routing checks and also for ECOs. But when you go deep into it, the formal verification used for verifying RTLs is entirely different from others. It is a method to prove the correctness of design or show the root cause of an error by rigorous mathematical procedures. It does not require test benches or stimuli and the turnaround time is very less. It proves that the design does what it is

supposed to do and this can not be done with testing because of following reasons:

- The major goal of software testing is to discover the errors in the software with a secondary goal of building confidence in the proper operation of the software when testing does not discover errors.
- In the absence of other information, this could mean either that the software is high quality or that the testing process is low quality.
- Program testing can be used to show the presence of bugs, but never to show their absence.

1.2.1 ISA Formal

ISA Formal is a formal verification technique for verifying that processor correctly implement ISA part of the whole architecture. This method uses bounded model checking to explore different sequences of instructions and was able to detect the above defect prior to release of the RTL to manufacturers.

ISA Formal can catch following errors.

- Errors in decode.
- Errors in datapath.
- Error in forwarding logic.
- Error in register renaming.

In this project, we are performing formal verification on C-class processor by comparing it against formal specification of RISC-V ISA.

Minimum prerequisites for RISC-V ISA formal verification

- Unambiguous formal ISA specification.
- A processor implementation controllable to verification.
- Formal link between the two.
- Error in register renaming.

1.2.2 What is Formal Specification?

A formal software specification is a statement expressed in a language whose vocabulary, syntax, and semantics are formally defined. The need for a formal semantic definition means that the specification languages cannot be based on natural language; it must be based on mathematics. The development of a formal specification provides insights and understanding of the software requirements and the software design. Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specifications. Formal specification may be automatically processed. Software tools can be built to assist with their development, understanding, and debugging. Formal specifications are mathematical entities and may be studied and analysed using mathematical methods.

In this project, a formal spec of the RISC-V Instruction Set Architecture, written in Bluespec BSV (executable, synthesizable) by Rishiyur S. Nikhil is used (Nikhil). This formal spec covers:

- RV32IM and RV64IM, i.e., the 32-bit and 64-bit user-level instruction sets ("I"), including integer multiply/ divide/ remainder ("M").
- A subset of machine-level privileged instructions and CSRs, including trap handling but excluding physical memory protection and performance-monitoring.

1.3 Bluespec SystemVerilog

The Processor RTL, formal specifications and verification module are written in Bluespec SystemVerilog (BSV). BSV (Bluespec SystemVerilog) is a language used in the design of electronic systems (ASICs, FPGAs and systems). BSV is used across the spectrum of applications, processors, memory subsystems, interconnects, DMAs and data movers, multimedia and communication I/O devices, multimedia and communication codecs and processors, signal processing accelerators, high-performance computing accelerators, etc.

BSV is a high level Hardware Description Language. It expresses synthesizable behavior with rules, a rule can be viewed as a declarative assertion expressing a potential atomic state transition. The BSV compiler produces efficient RTL code that manages all the potential interactions between rules by inserting appropriate arbitration

and scheduling logic, logic that would otherwise have to be designed and coded manually. BSV connects the modules by interfaces and methods. It also provides predefined library elements like FIFOs, BRAMs etc. which are modeled using BSV methods.

It has powerful static type checking which removes potential human errors which can't be detected at the stage of compilation normally but can be detected now during the compilation. BSV also has more general type parameterization (polymorphism) due to which modules and functions can be parameterized by other modules and functions, this enables the designer to reuse designs and glue them together in much more flexible ways. BSV's static elaboration helps to arrive at the design much faster than the other HDLs. The BSV compiler also can generate the synthesizable Verilog code of the written bluespec code which can be used later for synthesis purposes.

CHAPTER 2

Implementation

2.1 Basic Approach of ISA-Formal

We start with the processor in a simple, well-defined state uArch0 with no instructions in the pipeline. We then execute for a number of cycles where each cycle may issue an instruction. This serves to put the processor into a more complex state where hazards, forwarding, etc. can occur. And finally, we execute an instruction I_n and test whether the instruction executes correctly. This is done by applying an abstraction function abs which extracts the architectural state of the processor immediately before I_n executes and immediately after In executes. We do not flush the pipe before or after I_n.

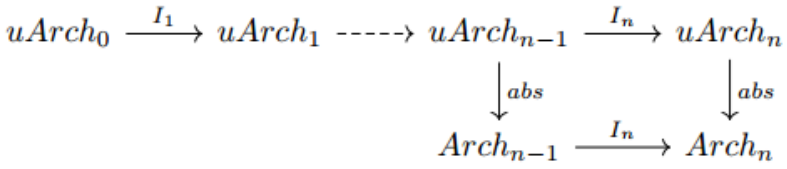


Figure 2.1: Basic approach for ISA-Formal

A key part of making this scalable is that, instead of allowing the formal verification tool to choose any instruction for I_n, we enumerate all the instruction classes supported by the architecture and perform a separate check for each instruction class. Proving these simpler results is helpful early in processor development by making it easy to focus on checking the currently implemented instructions. Later in development, the pattern of failing instructions is a useful guide in localizing the fault: if all branch instructions are failing, there is no need to worry about bugs in the ALU. Also, as the size of the verification task scales up, splitting the verification task into many small properties lets us make more effective use of our verification cluster which is optimized for running many independent processes across hundreds of machines.

To make this more concrete, consider the task of checking an addition instruction in the classic 5-stage pipeline illustrated in Fig. 2.2 This consists of 5 pipeline stages responsible for instruction fetch (IF), decode (ID), execute (EX), memory access (MEM) and writeback of results (WB). Values are read from the register file at the ID/EX boundary and results are written to the register file at the MEM/WB boundary. Forwarding paths (aka bypass logic) are used to reduce the number of stalls by allowing the result of one instruction to be used as an input to the ALU if required by the next instruction. Conventionally, most of the control signals from decode and those that control the pipeline and forwarding paths are not shown, although that is where many of the most difficult bugs lie.

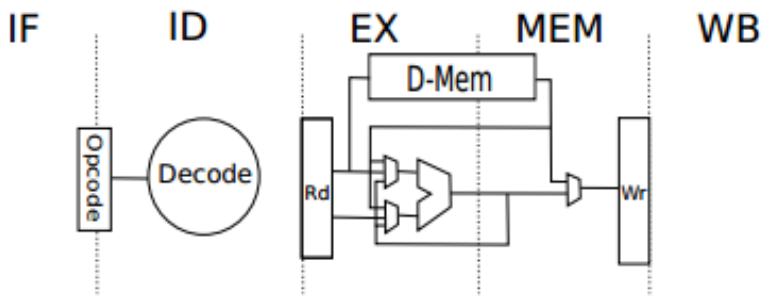


Figure 2.2: A 5-stage processor pipeline, with forwarding paths, omitting I-Fetch

First challenge is to implement the abstraction function abs which is responsible for converting the micro-architectural state of the processor into an architectural state. The other part of the input state of the processor that we require is the current instruction. To verify an addition instruction, the function abs must extract the current values of the integer registers.

Second challenge is to create a specifications of the instructions. Using instruction specification, opcode and Pre state data we can get post state data and compare with post state data of the processor.

2.2 Applying ISA-Formal to C-Class Processor

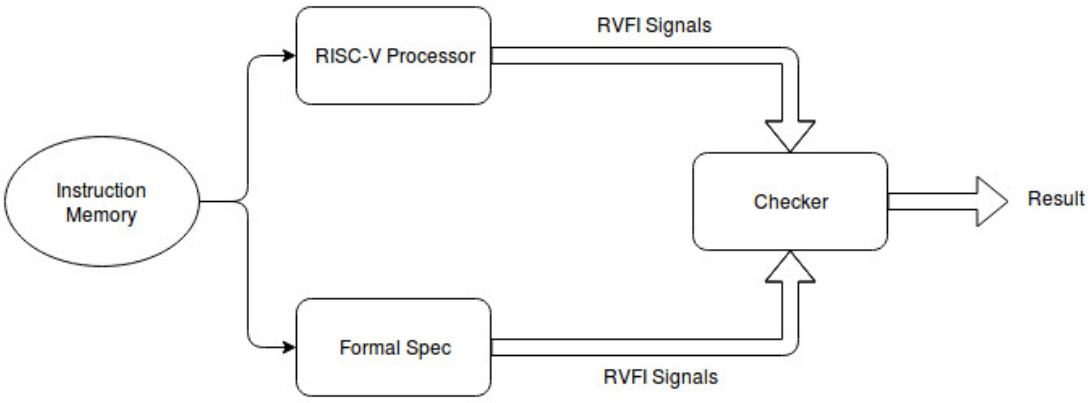


Figure 2.3: Formal Verification Overview

Figure 2.3 show the outline of the whole formal verification procedure. It has five components RISC-V processor(DUT), Formal Spec, Checker, RVFI signals and Instruction memory. Their implementation is explained in following sections.

2.2.1 RISC-V Processor

In this project we are performing formal verification on C Class process or of SHAKTI Processor family. In this processor the core and peripherals are developed using Bluespec. Features of C-CLASS:

- 5-stage 64/32-bit pipelined core.
- Supports ISA=RV64IMAFD based on riscv-spec-2.2 and privilege-spec-1.10.
- Bimodal branch predictor with a Return-Address-Stack support.
- Parameterized blocking Instruction and Data cache.
- Serialized Single and Double Precision Floating Point Units.
- Early out multiplier and a restoring divider.
- JTAG Debugger based on debug-spec-0.13
- Supervisor mode - sv39.
- Boots riscv-linux kernel.
- Performance = 1.67DMIPS/MHz and 2.2 Coremarks/MHz.

2.2.2 Formal Spec

In this project A formal spec of the RISC-V Instruction Set Architecture, written in Bluespec (executable, synthesizable) by Rishiyur S. Nikhil is used. It is a single stage RISC-V CPU with an interface define for main memory. This formal spec covers:

- RV32IM and RV64IM, i.e., the 32-bit and 64-bit user-level instruction sets ("I"), including integer multiply/ divide/ remainder ("M").
- A subset of machine-level privileged instructions and CSRs, including trap handling but excluding physical memory protection and performance-monitoring.

A memory module is developed for formal spec which is similar to memory module of DUT. RVFI signals are also developed for formal spec which are compared with RVFI signals of DUT.

2.2.3 RVFI Signals

RVFI is an output only port for RISC-V cores which is used to read the architectural state of the processor. Since it is an output only port, it does not effect regular operation of the core. It is also implemented for Formal Spec. Following are the RVFI signals.

- **rvfi_valid**: When the core retires an instruction, it asserts the rvfi_valid signal. The signals below are only valid during such a cycle and can be driven to arbitrary values in a cycle in which rvfi_valid is not asserted.
- **rvfi_insn**: It is the instruction word for the retired instruction.
- **rvfi_rs1_addr**: Decoded rs1 register address of retired instruction.
- **rvfi_rs2_addr**: Decoded rs2 register address of retired instruction.
- **rvfi_rs1_rdata**: Value of the rs1 register before execution of the instruction. For an instruction that has no rs1 register, this output can have an arbitrary value. However, if this output is nonzero then rvfi_rs1_rdata must carry the value stored in that register in the pre-state.
- **rvfi_rs2_rdata**: Value of the rs2 register before execution of the instruction. For an instruction that has no rs1 register, this output can have an arbitrary value. However, if this output is nonzero then rvfi_rs2_rdata must carry the value stored in that register in the pre-state.
- **rvfi_rd_addr**: Decoded rd register address of the retired instruction. For an instruction that writes no rd register, this output must always be zero.

- **rvfi_rd_wdata**: It is the output value which will be written in rd after execution of the instruction. This output must be zero when rd is zero.
- **rvfi_pc_rdata**: This is program counter before execution of the instruction.i.e. this is address of retired instruction.
- **rvfi_pc_wdata**: This is program counter after execution of the instruction.i.e. this is address of the next instruction to be executed.

In a 5-stage CPU instructions are retired in write back stage so rvfi_valid signal will be set high only for that clock cycle and all other signals will be valid for this particular cycle only. rvfi_insn_rs1 and rs2 are directly decoded from the retiring instruction instruction. After decoding rs1 and rs2 their values are fetched from the CPU register file and rvfi_rs1_rdata and rvfi_rs2_rdata are updated accordingly. rvfi_pc_rdata will have program counter value of retiring instruction not the current program counter. rvfi_rd_wdata will have the result of the instruction operation which will be updated in rd register in CPU register file. For instructions like branch etc. where there is no destination register(rd) rvfi_rd_wdata will have zero value. rvfi_pc_wdata will store the program counter value after execution of the instruction or in other words, it is pc value of next retiring instruction.

rvfi signals are also developed in Formal Specifications. Since it is a single stag CPU, all the signals are generated in single clock cycle.

2.2.4 Instruction Memory

Here program instruction are stored in hex format and are accessible to DUT and Formal Spec. DUT and formal spec both execute same instructions and then checker module compares rvfi signals of both of them.

Here instructions are generated randomly using Automatic Assembly Program Generator(AAPG). AAPG first generates random Assembly program which are then compiled using RISCV tools and instruction are generated in hex form.

Assembly program is generated in such a way that first few instructions will initialize all the CPU registers with random values. Then following instruction will be generated randomly to bring CPU in a complex state and then finally the instruction for which formal verification needs to be performed will be added. we repeat it 1000 times

keeping opcode of last instruction same. Same procedure is followed for all the types of instructions.

2.2.5 Checker

This module takes rvfi signals from DUT and formal spec as input and compares them to find any bug in DUT. Before start of the verification all we know is the address of the target instruction. First we start both DUT and formal spec and this module keeps checking program counter of each instruction that retires. When rvfi_valid signal goes high and rvfi_pc_rdata matches to address of target instruction verification starts and checker module captures all the rvfi signals on both sides.

After capturing RVFI signals we compare rvfi_insn signals to be sure that we are verifying same instruction on both sides. Then we compare rvfi_rs1_addr and rvfi_rs2_addr this check ensures that DUT decodes the instruction correctly. Then we compare rvfi_rs1_rdata and rvfi_rs2_rdata of both sides this ensures that they has reached at same point after executing a set of random instructions. Then we compare rvfi_rd_addr of both sides, now, if rd is none-zero, then we check if rvfi_rd_wdata is same for both. If they are same that means test is successful. After retiring of every instruction we check for rvfi_pc_wdata on both sides, If instruction is branch/jump type instruction then this check is ensure that after program counter on both side is same, else program counter will be incremented by 4 on both sides.

In some cases it may happen that the target instruction never gets executed because of jump/branch instruction, in that case we check that it happens in DUT also and test is considered successful.

2.3 Limitations of formal verification

One of the most subtle limitations is that ISA-Formal does not check that all instructions retire in order or even that all the instructions that should retire (non-speculative, not-canceled, etc) do retire and only retire once. All it checks is that, if an instruction does retire, then it has the correct effect.

ISA-Formal works by using a model checker to feed sequences of instructions into the instruction decoder. Which means that ISA-Formal explicitly excludes instruction fetch from consideration.

To improve performance of the pipeline checker, we treat the memory system as a black box into which we feed addresses and which gives back memory faults or data values. And we use memory interface specifications to ensure that this black box has the full range of legal behaviors that the actual memory system can exhibit.

We omit the memory system because it is very large and stateful (making it a challenge for model checkers) and because the concurrency between the processor on one side and the bus/cache on the other side is better checked using other techniques.

All verification techniques have limitations in what they can check. The important thing is understanding those limitations so that you can find some other technique to fill the gaps.

CHAPTER 3

Results and Conclusion

This formal verification has covered RV32IM and RV64IM, i.e., the the 32-bit and 64-bit user-level instruction sets ("I"), including integer multiply/ divide/ remainder ("M") and a subset of machine-level privileged instructions and CSRs, including trap handling, but excluding physical memory protection and performance-monitoring. Random assembly programs were generated using python script. Every type of instruction is verified 1000 times with differently generated instruction using different seeds.

Figure 3.1 shows test case to verify Add instruction. First few instructions are random instructions to put CPU in a complex state and then we execute 302^{nd} instruction which is Add instruction. We compare pre and post states of both the CPUs and verify the DUT.

```
942
943  0000000080000558 <i_296>:
944      80000558:   fe410383           lb  t2,-28(sp)
945
946  000000008000055c <i_297>:
947      8000055c:   ea312423           sw  gp,-344(sp)
948
949  0000000080000560 <i_298>:
950      80000560:   fdf10f83           lb  t6,-33(sp)
951
952  0000000080000564 <i_299>:
953      80000564:   ff611503           lh  a0,-10(sp)
954
955  0000000080000568 <i_300>:
956      80000568:   018282b3           add t0,t0,s8
957
958  000000008000056c <i_301>:
959      8000056c:   00000013           nop
```

Figure 3.1: Disassembled code of Add instruction

Figure 3.2 shows a successful formal verification of Add instruction. In the figure all the RVFI signals are visible and they all are matching, hence verification_success flag goes high.

After performing rigorous formal verification we no bug found which indicates that C-Class processor of SHAKTI family has successfully implemented RISC-V ISA.

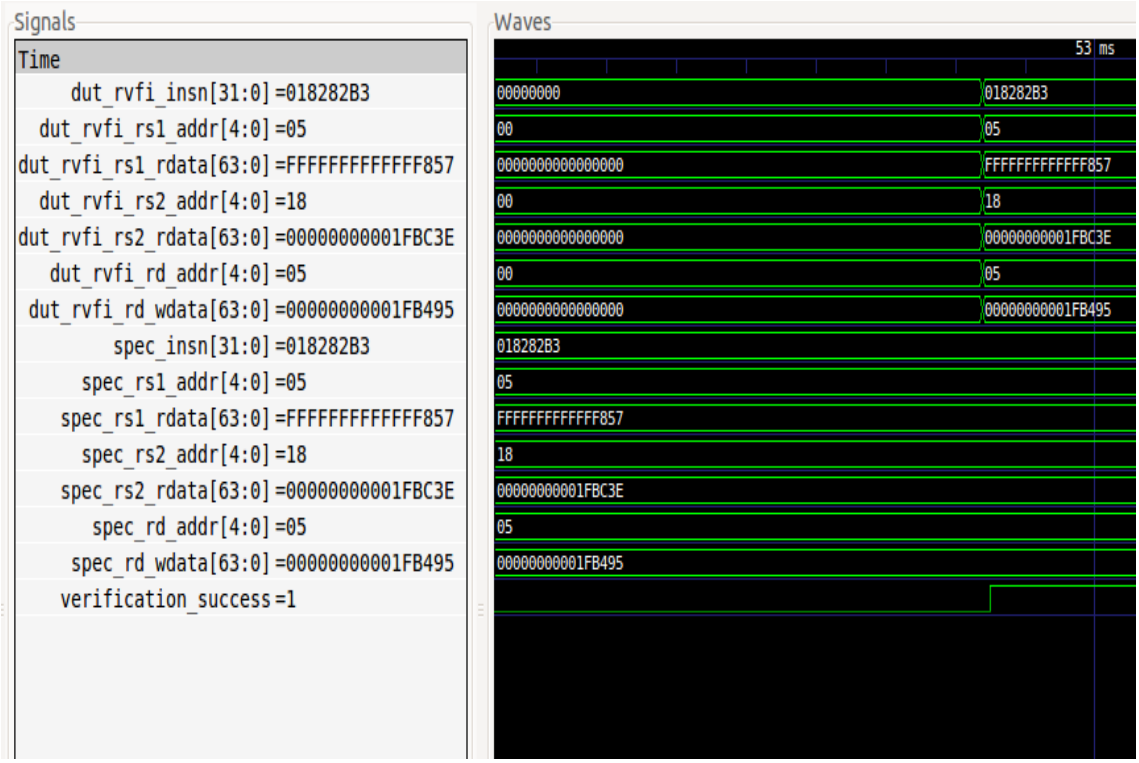


Figure 3.2: Verification of Add instruction

Future work

Formal verification can be extended to other user level instruction set(A,F,D etc.) and other privilege levels(supervisor). This can be done by using the formal specifications of other user level instruction set(A,F,D etc.) and other privilege levels(supervisor) of RISC-V.

APPENDIX A

Git link of the project is following: `https://krishanprajapat@bitbucket.org/krishanprajapat/isa-formal.git`

REFERENCES

1. *Bluespec* (retrieved:). URL http://bluespec.com/.

2. **Nikhil, R. S.** (). Riscv isa formal spec in bsv. URL https://github.com/rsnikhil/RISCV_ISA_Formal_Spec_in_BSV.

3. **Reid, A.**, **R. Chen**, **A. Deligiannis**, **D. Gilday**, **D. Hoyes**, **W. Keen**, **A. Pathirane**, **O. Shepherd**, **P. Vrabel**, and **A. Zaidi** (). End-to-end verification of arm processors with isa-formal. URL https://alastairreid.github.io/papers/cav2016_isa_formal.pdf.