



CHALMERS
UNIVERSITY OF TECHNOLOGY



UNIVERSITY OF GOTHENBURG

White-box Verification of a RISC-V Processor: NOEL-V

Master's thesis in Embedded Electronic System Design

Jonathan Jonsson

Matteo Toselli

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

MASTER'S THESIS 2022

White-box Verification of a RISC-V Processor: NOEL-V

Jonathan Jonsson
Matteo Toselli



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

White-box Verification of a RISC-V Processor: NOEL-V

Jonathan Jonsson
Matteo Toselli

© Jonathan Jonsson, Matteo Toselli, 2022.

Supervisor: Per Larsson-Edefors, Department of Computer Science and Engineering
Company Advisor: Martin Rönnbäck, Cobham Gaisler
Examiner: Lena Peterson, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in L^AT_EX
Gothenburg, Sweden 2022

Jonathan Jonsson, Matteo Toselli

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Abstract

The NOEL-V processor, based on the RISC-V ISA, targets space applications and safety-critical applications. The design has gone through verification since its early development stages. However, this verification has mostly consisted of broader software-level tests. Therefore, to extend the verification effort, Google's RISC-V-DV framework is applied. Furthermore, formal verification is performed on the physical memory protection (PMP) unit, which is used to divide user-level code and privileged code at the hardware level. RISC-V-DV was able to expose one decode issue and one configuration issue. Formal verification found corner cases that could expose privileged regions to non-privileged accesses. While the bugs uncovered with RISC-V-DV were non-critical, unintentional exposure of safety-critical memory regions to user-level code, containing for example encryption keys, can lead to severe consequences. In addition, since different bugs were found using different methods, this shows that RISC-V-DV alone could not uncover all issues related to PMP. Hence, it is also likely that not all issues related to other units are uncovered. Therefore, multiple approaches and multiple engineers are necessary to reach a thorough verification effort.

Keywords: RISC-V, NOEL-V, RISC-V-DV, Formal Verification, RTL, ISA, ISS, JasperGold, Physical Memory Protection (PMP), SystemVerilog Directives

Acknowledgements

We would like to offer our special thanks to Per Larsson-Edefors for providing academic feedback throughout the thesis process.

Assistance provided by Johan Klockars and Nils Wessman at Cobham Gaisler helped us gain valuable technical knowledge about the designs and verification work.

We would also like to thank Cobham Gaisler in general and Martin Rönnbäck in particular for welcoming us to do our thesis work in their office.

Jonathan Jonsson, Matteo Toselli

Gothenburg, June 2022

Contents

Acronyms	xi
1 Introduction	1
1.1 RISC-V and the European Space Agency	1
1.2 Objective and Scope	2
1.3 Prior Work	3
1.4 Thesis Outline	3
2 Technical Background	5
2.1 Abstraction Levels of a System-on-Chip (SoC)	5
2.2 Details of RISC-V	5
2.2.1 Base Instruction Set Architecture and Extensions	5
2.2.2 Privilege Levels	6
2.2.3 Control and Status Registers	6
2.2.4 Physical Memory Protection	7
2.3 Details of NOEL-V	8
2.4 Functional Verification	9
2.4.1 Simulation-based Verification	9
2.4.2 Formal Verification	10
2.4.3 System Verilog Directives	11
2.4.4 Coverage Metrics	12
3 Method	13
3.1 Frameworks	13
3.1.1 RISC-V-DV	14
3.2 Tailored Tests for a Functional Unit	16
3.2.1 Formal Verification of the PMP Unit	16
4 Results	19
4.1 RISC-V-DV	19
4.1.1 Bugs found	19
4.1.2 Coverage	21
4.2 Formal Verification	24
4.2.1 Bugs found	24
5 Discussion	27
5.1 RISC-V-DV	27

5.2	Formal Verification of PMP	27
5.3	Assessment of the verification efforts	29
5.4	Future work	30
6	Conclusion	31
	Bibliography	33
A	Appendix 1	I

Acronyms

- ALU** arithmetic logic unit 16
- CPU** central processing unit 3, 6, 8, 10
- CSR** control status register 6, 7, 10, 14, 17, 30
- DUT** device under test 2, 3, 12–14, 16, 17, 31
- ECC** error correcting code 2
- ESA** European Space Agency 1, 2
- FPGA** field programmable gate array 2
- FPU** floating point unit 1–3, 8, 9, 16, 30
- FU** functional unit 5
- FV** formal verification 1
- GPL** general public license 2, 8, 9
- GPR** general purpose register 10, 14, 15, 20, 27
- hart** hardware thread 6
- I/O** input output 5
- IP** intellectual property 13
- ISA** instruction set architecture 1–3, 5, 8–10, 13, 16, 30, 31
- ISG** instruction stream generator 2, 14, 15, 31
- ISS** instruction set simulator 2, 10, 13–15, 28
- MMU** memory management unit 2, 13, 16, 29
- NA4** naturally aligned four 7, 17
- NAPOT** natural power of two 7, 17, 26, 28
- PC** program counter 14
- PMP** physical memory protection v, ix, x, I, 2, 3, 7, 8, 13, 16, 17, 19, 24, 25, 27–31
- pmpaddr** PMP address register 7, 17, 24–26, 28–30
- pmpcfg** PMP configuration register 17
- RTL** register transfer level 2, 3, 5, 8–10, 13, 14, 16
- RVFI** RISC-V formal interface 14, 30
- SoC** system on a chip 2, 5
- TOR** top of range 7, 17, 24, 25, 28

1

Introduction

The aim of this thesis project is to perform white-box verification on Cobham Gaisler's NOEL-V processor. Hardware verification consists of ensuring that the design adheres to the specifications, and it is one of the key bottlenecks in the design process, on average accounting for more than 50% of the entire design effort [1]. The term whitebox refers to having full access to the source code of a program or a design. In contrast, blackbox would refer to its opposite, where the internals are unknown and only inputs and outputs are accessible. The process of verifying a design is complex and involves different levels of abstraction, such as system level, chip level, core level, and functional unit level. Each of these aspects in the design has to be verified against the specifications, as an undetected bug in the final product can lead to substantial financial loss. This was the case for the Intel Pentium in the '90s, where a bug in the floating point unit (FPU) forced the company to replace all units, at a loss of 475 million dollars. Furthermore, verification is not only a concern for large companies, that can afford to employ verification engineers, but any business that trades in hardware design has to implement a verification plan and strategy.

Given the importance of the verification process in hardware design, significant effort has been made at a research and industry level to develop optimal verification strategies. An optimal verification strategy maximizes coverage and minimizes the verification time. Nowadays, the most popular methods for hardware verification are simulation-driven and formal verification (FV). Briefly, simulation-driven verification tests the design by injecting input stimuli and comparing the results with a known-good reference. FV instead proves the design by mathematical reasoning.

1.1 RISC-V and the European Space Agency

The value of open source in the software domain has in the past two decades shown its benefits to several fields of the industry [2]. The openness of such software has seen an increase in reusability, transparency, and wider support for products that leverage it. The reusability benefit can also apply to the hardware domain, particularly in standards such as an instruction set architecture (ISA). In the '90s, when the European Space Agency (ESA) were deciding on an ISA to use for their processors, they chose SPARC since it was the most suitable open option at that time [3]. Nowadays, SPARC is still the main ISA used in ESA projects. However, since its introduction, it has progressively lost popularity in commercial applications against proprietary ISAs such as ARM and x86. This led to a decrease in software

and compiler support, hence, in recent years, the ESA have supported research to find a suitable ISA alternative [3]. One novel option is RISC-V [3], whose main advantages against the competition are openness, scalability, and modularity. Modularity means that subsets of the ISA can be chosen, and special instructions can be added as required, thus yielding a more lightweight design [4]. This, combined with its openness, provides the perfect foundation for developing highly scalable processors ranging from low-performance/low-power microcontrollers to high-performance processors [3].

To address the issues of SPARC, Cobham Gaisler, a Gothenburg-based company specialized in developing fault-tolerant embedded processors, was contracted by the ESA to develop a new multicore processor: NOEL-V. This new implementation of a RISC-V architecture particularly targets the space market, but can also serve in terrestrial applications. Given the harshness of space, fault tolerance measures have been developed. These include error correcting code (ECC), redundancy, and parity. NOEL-V is licensed both under a general public license (GPL) and a commercial license. The commercial license offers additional features for advanced functionalities, such as radiation hardening, a fully pipelined FPU, and an L2 cache.

This thesis focuses the verification efforts on the publicly available version of the implementation, which in short features a 64-bit architecture, with an advanced dual-issue in-order pipeline, compressed 16-bit instruction support, atomic instruction extension, and a non-pipelined FPU.

Since its early development stages, NOEL-V has been quite extensively verified using both standard test suites as well as random instruction sequences, therefore basic coverage has already been achieved by Gaisler's team. In particular, an internally developed instruction generator has been used to generate random instruction sequences, which in combination with an instruction set simulator (ISS) tests the design in a simulation environment. The test suite has also been deployed on a field programmable gate array (FPGA) to accelerate the verification effort.

1.2 Objective and Scope

This thesis project aims to increase the verification coverage already achieved by Gaisler's team, trying to find weaknesses in the design. To achieve this goal, different approaches are used. First, the design is tested with a well-established framework: RISC-V-DV from Google [5]. It consists of an instruction stream generator (ISG), capable of creating a constrained set of tests that stress the design in different ways. Examples of such tests are: arithmetic tests, memory management unit (MMU) stress tests and FPU tests. As a second and more targeted approach, formal verification is performed. In particular, the physical memory protection (PMP) unit has been chosen to be the target of the formal tests.

The verification process is focused on the register transfer level (RTL) and ISA level of NOEL-V, meaning that any technology-dependent abstraction level is not the subject of the thesis work. Therefore, fault injection, derived from physical models of the space environment's effects on the system on a chip (SoC), is not performed. The device under test (DUT) is the GPL version of NOEL-V, meaning that some of the most advanced features of NOEL-V, such as radiation hardening,

L2 cache, fully pipelined FPU, are not considered as well.

An important part of the verification process is also the analysis of the tests performed, to derive functional coverage. Establishing comprehensive coverage metrics is essential for ensuring that the design is tested under as many different scenarios as possible. Eventually, this analysis can be used to narrow the scope of future tests, where the design is not sufficiently covered.

1.3 Prior Work

Simulation-based methods are still predominant in verification and validation efforts, due to their ease of use and scalability [6]. Therefore verification frameworks that compare the result of an RTL simulation to a golden reference have seen wide adoption, and RISC-V based architectures are no exception. Google’s RISC-V DV has been used to verify various RISC-V processors similar to NOEL-V. These include the Ibex central processing unit (CPU), developed by lowRISC [7], in which the majority of the bugs uncovered were related to debugging mode, illegal/hint instructions, interrupt handling and memory access faults [8]. Another example is CVA6, formerly Ariane, developed by OpenHW group [9], in which similar bugs were found [8].

A verification methodology on the rise in industry and academia is formal verification [1]. This approach is potentially more powerful than simulation-based verification, but with the downside of being difficult to perform on larger designs [10, Ch. 1]. Formal verification is particularly useful for safety-critical units and bus interfaces, since a complete formal proof, although sometimes hard to perform, does not leave room for any leakage in the DUT. An example of where formal verification has been applied on a safety-critical unit is the PMP unit of the RISC-V Rocket [11].

1.4 Thesis Outline

Chapter 2 aims to give an insight into the verification object, NOEL-V and its ISA, as well as the background of verification methodologies. The subsequent chapter 3 describes the verification flow employed in the thesis process, and how coverage metrics are extracted. Eventually, chapter 4 and 5 present the results and discussion about bugs found, coverage reached, and possible future work. The final chapter summarizes the thesis work and presents its key takeaways.

2

Technical Background

This chapter starts with an introduction to what an SoC is. It also describes the fundamentals of the RISC-V specification, with emphasis on the units targeted in tailored tests. Eventually, the two paradigms of functional verification are presented, as well as coverage metrics.

2.1 Abstraction Levels of a System-on-Chip (SoC)

An SoC is a semiconductor device that contains functional units (FUs), each of which are responsible for key system features, such as input output (I/O), memory management, arithmetic, general computing, etc. The design of an SoC is usually split into multiple levels in a hierarchy, beginning with a specification of what the system should do, followed by an algorithmic description of how the design should accomplish its specification. A specification and its subsequent algorithms are nonetheless relatively far from its implementation. The next step in the design process is to implement the algorithms as an RTL description. An RTL design describes circuit behavior, i.e it specifies how data should flow between hardware registers and logical operations. To realize the RTL description as an SoC, it is synthesized to a netlist of physical building blocks that can be placed, routed, and then manufactured.

2.2 Details of RISC-V

When UC Berkeley designed RISC-V, one of the key philosophies behind it was to have an open ISA with support for almost any computing device [12]. In contrast, prior ISAs had been engineered with a particular architecture in mind. For example, MIPS was designed around a single issue, in order five-stage pipeline, which in later years has made it complicated to incorporate modern paradigms such as superscalar and superpipelined implementations. Furthermore, other ISAs such as ARMv8 offer more features than required for certain applications, making some implementations unnecessarily large and inefficient.

2.2.1 Base Instruction Set Architecture and Extensions

RISC-V features both 32 and 64-bit configurations with the base being the integer instruction set. On top of that, optional extensions are available to feature richer implementations. In Table 2.1 most of the RISC-V extensions are shown. The

extensions include, e.g., IEEE754 floating point support, vector instructions and compressed instructions [12].

Table 2.1: Standard Extension to the RISC-V instruction set [13]

Name	Description
RV32I	Base Integer Instruction Set - 32-bit
RV64I	Base Integer Instruction Set - 64-bit
Extension	
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point
Q	Standard Extension for Quad-Precision Floating-Point
C	Standard Extension for Compressed Instructions
B	Standard Extension for Bit Manipulation
V	Standard Extension for Vector Operations
N	Standard Extension for User-Level Interrupts
H	Standard Extension for Hypervisor
S	Standard Extension for Supervisor-level Instructions

2.2.2 Privilege Levels

Whenever a RISC-V hardware thread (hart) executes instructions, it will do it in one of three modes depending on the implementation. The privilege specification [14] states that three privilege levels are available. These are machine (M), supervisor (S), and user (U) mode. All RISC-V implementations must support at least M mode and optionally M and U or all modes. The purpose of the modes is to establish a privilege structure between different programs to offer protection. Programs running in U mode should not be able to access the S or M level and the S level should not be able to access the M level. M mode has full system access and the code executed in M mode is inherently trusted. S mode is intended for operating systems and U mode for user-level applications. Switches between privilege modes are done by trap handlers, for example when a timer interrupt is received. Traps that increase the privilege level are termed vertical traps while traps that keep the same privilege level are termed horizontal traps.

2.2.3 Control and Status Registers

Control status registers (CSRs) are used to track the architectural state of the machine and to configure the CPU [14]. Such a CSR is the *mstatus* register, which contains information about M mode execution state. A similar register *sstatus* holds a subset of the *mstatus* content since this register only has S mode privilege. For example, *mstatus* contains the MPP bit-field which holds information about the prior privilege mode, and the MPRV bit which, when set, modifies the current execution

state to that of MPP. Other examples of CSRs are PMP configuration and address registers, trap vectors, trap values, performance monitors, etc.

2.2.4 Physical Memory Protection

PMP is an optional extension used to support secure processing and to contain faults. This is done by incorporating a privilege structure to physical memory accesses. Privileges include, read, write and execute permissions [14]. By default, M mode has full memory access. However, memory access permission can be revoked from M mode too, and doing so will permanently impede the processor from accessing a given memory region until a full system reset is performed. Memory access from S and U mode is always checked while M mode access is only checked if access privileges have been revoked from M mode.

To control PMP permissions, a set of CSRs is dedicated for configuring the permissions and their locations. In total the specification allows configurations of 0, 16 or 64 PMP entries. An entry is a physical address range with its associated privileges. Assuming 64 entries, a 32-bit architecture consists of 16 configuration registers while the 64-bit counterpart of eight configuration registers. In addition to the configuration registers, 64 PMP address register (pmpaddr) are present, each one holding the start or end address of the PMP region. One configuration entry is one byte long, therefore the 32-bit architecture can hold four configurations in one register, while the 64-bit version can hold eight.

The configuration byte consists of six bit fields each of which controls the PMP entry. As shown in Fig 2.1, the fields consist of lock, address match, write, read, execute, and unused. The lock bit locks the configuration byte from being altered, as well as the pmpaddr. Even M mode can not change the configuration if the byte has been locked, therefore the only way to unlock it is to do a hard system reset. The address-match bits control the address-match mode and they can be configured in four different ways. The first mode means disabled, then no addresses match, regardless of the content in the address registers. The second mode, top of range (TOR), matches aligned addresses in an arbitrarily set address range. By configuring pmpaddr(i) to the end address of the PMP region, the range is then determined by pmpaddr(i-1) thus allowing for up to 64 arbitrarily large PMP regions. Since TOR mode uses two pmpaddr to set its range the L-bit also locks the prior pmpaddr entry. The two last modes, naturally aligned four (NA4) and natural power of two (NAPOT) are similar to each other in NA4 being a special case of NAPOT. NA4 can configure regions as small as four bytes while NAPOT can configure regions 8 bytes or larger. The smallest possible PMP region that can be configured is set by the granularity parameter. This parameter effectively sets which bit of a PMP address is the least significant bit, and thus also the alignment of the address. The lowest granularity allowed by the specification is zero. The byte-wise alignment then corresponds to $2^{(\text{granularity}+2)}$ bytes. To set the size of a NAPOT region all bits between zero and one less than the granularity position must read as ones. The position of the first zero then determines the total size of the NAPOT region. An example of how a NAPOT range is configured is shown in Fig. 2.2. For a pmpaddr configured as TOR to be valid, the address must read as all zeros from one bit less

than the granularity position to the LSB. The last three bits of the configuration byte are the permission bits, which controls access for execute (X), write (W) and read (R).

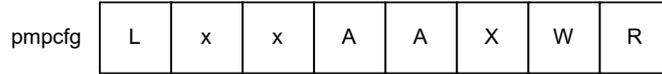


Figure 2.1: One entry for pmpcfg with bit fields corresponding to: L = lock, A = address mode, X = execute, W = write, R = read.

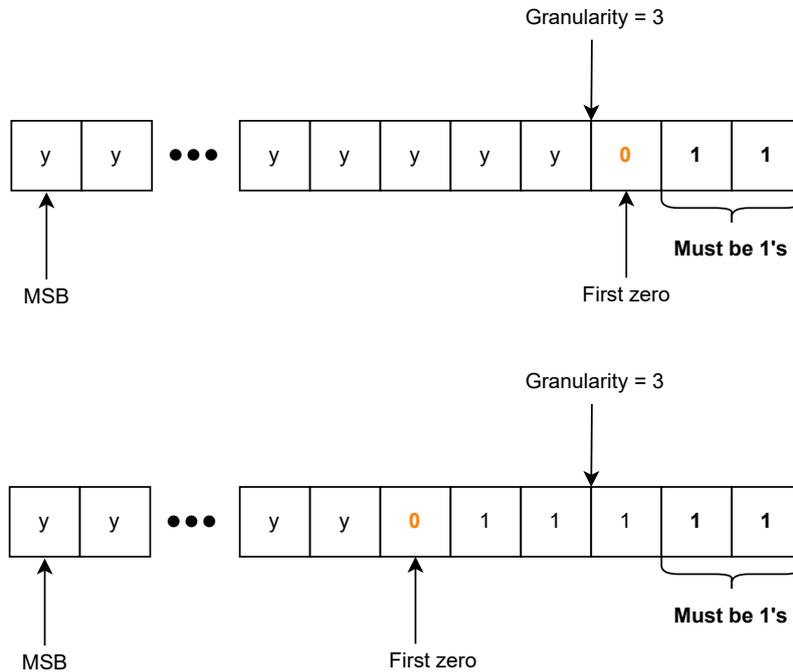


Figure 2.2: The top address shows that for a granularity of three, a zero can first appear in the third position. The size of this region will be the same as what the alignment specifies: $2^{3+2} = 32$ bytes. The bottom address sets the first zero in the sixth position which means that 2^3 regions $\times 2^{3+2}$ bytes = 256 bytes

2.3 Details of NOEL-V

NOEL-V is a parametrically configurable RISC-V based CPU with a few predefined standard configurations. These configurations range from bare minimum microcontrollers to high-performance, multi-core, dual-issue configurations with a pipelined FPU. Depending on the configuration, different extensions of the RISC-V ISA are added. The base configuration features the I and the M extensions while the fully-featured ones also include the A, F, D, C, and H extensions (see Table 2.1). NOEL-V also features PMP in most of its configurations. Furthermore, additional architectural configurations can be specified as generics in the RTL code. The GPL version

of NOEL-V lacks some features in comparison to its commercially licensed counterpart. Hardware-wise features not included in GPL are radiation hardening, a high-performance FPU and a fully-featured L2 cache, as for the ISA features all extensions are still supported.

2.4 Functional Verification

Verification is a wide area in both industry and academia, where many approaches have been tested to verify the functionality of a design. If the job of a design engineer is to take a specification to an implementation, then the job of a verification engineer begins at the opposite end [15]. A verification engineer takes an implementation and tries to prove that it fulfills the specification. A hardware system can be broken down into several layers of abstraction, beginning with the functional specification, then following with algorithm, RTL, gate netlist, transistor netlist, and physical layout.

Design and verification are usually done by two different groups of engineers, meaning that the functional specification must be interpreted by two different groups of people [15]. One source of bugs that may occur in a design may be a misinterpretation of the specification. Having two separate groups interpreting the specification can thus increase the confidence in having interpreted it correctly. However, as the specification is the top abstraction level, an incorrect or undefined behavior would propagate through all levels. This means that ensuring a proper specification is crucial to both the design and the verification effort. Each of the abstraction levels is subject to verification both within the abstraction layer and also between different ones. As previously stated this thesis will focus on functional verification, meaning that abstraction levels lower than the RTL level will not be considered. Functional verification comes in two different paradigms: simulation-based verification and formal verification [16].

2.4.1 Simulation-based Verification

The first paradigm, simulation-based verification, consists of four components: the circuit, test stimuli, reference output, and a comparator [15]. By feeding the circuit test stimuli an output is produced. This output can then be compared to that of the reference output with a comparator. If the simulation output matches the reference output, the circuit performs as expected for that input. It is up to the verification engineer to select test stimuli and to create a model that generates the reference output. Creating a verification reference model involves going back to the functional specification, interpreting it, and then designing a test around that specification. A verification engineer, therefore, performs similar steps as the ones of a design engineer.

Design engineers usually express their work in an RTL language, the verification model is usually expressed differently, for example in C++ or SystemVerilog. The verification model usually operates on a higher abstraction level than that of the implementation, meaning that the model does not have to be synthesizable or contend with the constraints of an RTL language. Prebuilt verification models may also be available depending on what the test focuses on. One example of a common model

that can be used to test CPUs is an ISS. An ISS executes ISA level instructions in a software model rather than a hardware model which is what an RTL simulation does. An ISS can therefore act as known-good reference for ISA level entities such as general purpose registers (GPRs) and CSRs.

2.4.2 Formal Verification

The second paradigm, formal verification, differs from simulation-based verification in that it does not require input stimuli [15, Sec. 1.3]. In simulation-based verification the input stimuli are in focus, formal verification pivots this by focusing on the outputs of a circuit. Formal verification involves two subcategories: property checking, and equivalence checking. Equivalence checking focuses on checking whether two implementations are functionally equivalent. As an example, equivalence checking is usually performed between a synthesized netlist and an RTL design, to discover any errors in the synthesis process. Performing equivalence checking using simulation is infeasible since the state space is likely infinite.

Property checking bases its verification on the concept of redundancy [15, Sec. 1.3]. This is done by duplicating the design in terms of its properties. An example of a property could be that a bit signaling if a buffer is full, should under no combination of inputs be set if the buffer is not full. Property-based verification does therefore not implement a property but rather states what should be true about a design. A property verifier can then begin at the signal for which a property has been specified, and work backward towards the inputs to scan its state space for a counterexample that could disprove the property. If a counterexample is found, then a debug waveform of the signals that caused the illegal behavior is presented. In comparison to simulation-based verification, property checking has the benefit of only scanning a state space for signals that actually affect the state of the signal. Simulations, in contrast, may have to employ a much larger state space to reach the same states that the property checker worked backward to discover. The state space deduced by a property checker may still be enormous and require infeasible amounts of memory and runtime for a proof to be produced. In addition, some states may be illegal since the inputs are constrained to some combination of states and values. Therefore, it is usually necessary to constrain signals to known-good ranges and states, however, this must be done with caution. It is easy to overconstrain a signal and accidentally rule out valid states, thus a bug may be missed. Underconstraining signals are not as dangerous, since, at worst, a false positive would be produced, but this could come at the cost of longer runtimes.

Property checking becomes more complicated when multiple clock cycles are involved, both in the state space growing and that an indefinite amount of time could be required to prove certain properties. Therefore some properties may only be proved to be true for a certain number of clock cycles: this is referred to as a bounded proof. Bounded proofs may sometime be enough despite being limited in time. For example, a buffer that takes x clock cycles to fill only needs to be proved to have a certain property for $x+1$ clock cycles. This context is something a property checker may be unaware of but an engineer is, thus a full proof was reached, despite being a bounded proof.

2.4.3 System Verilog Directives

To express properties for verification, several languages and libraries are available, such as the property specification language (PSL), the open verification library, and SystemVerilog [10, Ch. 3]. SystemVerilog has grown to be the industry standard for expressing properties in terms of assertions. SystemVerilog assertions are not only useful for formal verification but also for simulation-based verification, however sometimes with different implications than when applied in a formal context. SystemVerilog supports several directives useful for both formal and simulation-based verification. The directives and their hierarchy are shown in Fig. 2.3. Building from boolean expressions and sequences, powerful property expressions can be composed and given as directives to the tools. Each level is presented in more detail in the following sections.

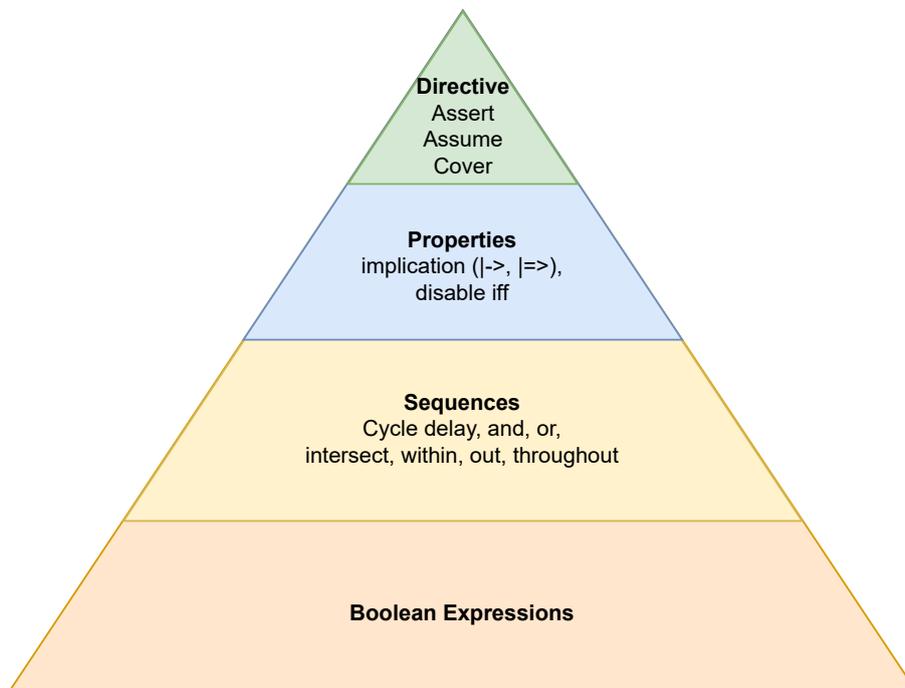


Figure 2.3: SystemVerilog directive organization pyramid

Assert

Assertions are a way to express a property about a design that should always be true [10, Ch. 3]. Again repeating the example of Sec. 2.4.2, if a buffer is not full, then the bit signaling a full buffer should never be set. In SystemVerilog this could be expressed with a boolean statement as follows:

```
check_buff_flag: assert property (!(flag_full && (buff_count <= 10)))
else $error("Buffer reported full despite empty space")
```

Given an assertion, the property checker would try to prove that there are no input combinations that can generate a failing assertion. In a simulation, assertions

are also useful since the assertion would fail if a failing input combination happened to be tested.

Assume

In a formal context, assumptions carry the role of constraining signals to attainable and legal values, i.e., it is a way to describe the environment of a DUT [10, Ch. 3]. The assumptions are then used by the tool to reduce the state space and to ignore failures where an assumption overrides the failure. In a simulation context, assumptions are also useful, but instead as a method to control that the simulation environment is behaving as expected. Assumptions are written similarly to assertions: in the following example the assumption states that an input signal can never be equal to ten. The printout is useful in simulations but not in a formal checker as the assumption is used differently.

```
good_value: assume property (sig_0 != 10)
else $error("Illegal Input Value sig_0.");
```

Coverpoint

A coverage point, or coverpoint, is used in both formal and simulation-based verification. The usage of coverpoints in simulation-based verification is clear: a coverpoint specifies important machine states that should be traversed [10, Ch. 3]. A coverpoint is therefore not like an assertion that must always hold. Instead, a coverpoint may only be true occasionally. Since formal verification covers all possible states by default, coverpoints may seem redundant, but since an assume statement can discard possible states, it is still useful to put coverpoints in a formal context to check whether a critical combination is covered. This means that coverpoints can be used to help discover overconstrained inputs.

2.4.4 Coverage Metrics

To measure how well a design has been verified, two metrics are used: code coverage and functional coverage [15, p. 21]. Code coverage is used to detect how much of the code was exercised during a test [15, Sec. 5.6]. Code coverage can thus give a hint about dead code and how to extend the test to cover more of the code. Code coverage does however not give any indication of how well the functionality was tested. Therefore, the second metric, functional coverage is needed. Functional coverage tries to approximate the link between the specification to that of the implementation in terms of a percentage of functionality tested. Functional coverage is also referred to as parameter coverage and it reveals how well the parameters and dimensions of a design were exercised.

3

Method

The approach used to verify NOEL-V consists of multiple steps. The first method employed is based on RTL simulations and targets the ISA level of the processor. This approach is a suitable beginning since it tests the entire design at the highest abstraction level: the RISC-V specification. A well-established framework that tests the design at this abstraction level is Google’s RISC-V-DV. The second method targets specific units at a lower abstraction level. Specifically, the PMP unit has been chosen, and since it is a security feature it is important to verify that there are no leaks in the implementation. Furthermore, since PMP is predominantly control logic it is suitable for formal methods.

3.1 Frameworks

A first approach is to utilize pre-existing frameworks that perform simulation-based verification targeting the ISA level. This means that the DUT is stimulated by instruction sequences, and the results obtained are compared to a known-good reference.

Since a processor, such as NOEL-V, is a complex architecture to verify, the stimuli applied in a simulation environment can not be purely random. Using random stimuli would be inefficient, as it would produce many illegal and unnecessary combinations. Instead, random instruction sequences are tailored to hit corner cases [17]. For this purpose, the frameworks provide test suites that for example include: arithmetic tests, MMU stress tests, illegal instruction tests, etc. Since the community developing RISC-V intellectual property (IP) is growing [18], the suites of different tests used for verification purposes are also growing, providing a wider range of tests.

To provide a known-good reference to the RTL simulation trace, an ISS is used. For this purpose, multiple open-source ISSs for the RISC-V ISA exist. A few common ISSs are OVPSim, Spike, Wishper and RISC-V Sail.

The approach of using pre-existing frameworks has several benefits. One major benefit is the reliance on openly licensed software, making it freely available. Furthermore, it is the main method used in the industry to verify RISC-V-based products [17]. Within the RISC-V community, there is an interest in establishing an industry-standard set of tests, and effective verification methodologies. These industry tools, such as frameworks and software toolchains, are therefore used in the thesis project.

As RISC-V is an open ISA, multiple frameworks for verification purposes exist. For this thesis project, RISC-V-DV [5], an open-source project developed by Google,

is used. A few additional examples of open-source frameworks include OpenHW group core-v-verif [19] and RISC-V formal [20].

RISCV-DV consists of an instruction stream generator (ISG) that generates a constrained set of tests, which are then compiled and used to stress the DUT. The framework relies on an ISS as the golden reference, and it also features a coverage model for collecting functional coverage. Therefore RISCV-DV is a suitable framework to use in this thesis project.

Core-v-verif is another open-source project that combines various tools to build an extensive verification environment. This verification environment is capable of running pre-existing targeted tests, as well as generating tests from different ISGs, including the one used in Google’s RISCV-DV. Therefore, the two frameworks are in a way similar, even though core-v-verif is tailored to verify OpenHW group-designed cores. For this reason, RISCV-DV has been chosen for the interest of this thesis project.

The third framework, RISC-V formal, focuses on formal verification for RISC-V processors. However, it requires that the design implements the RISC-V formal interface (RVFI), which is currently under development for NOEL-V. As a result, applying this framework depends on the availability of the interface.

3.1.1 RISCV-DV

This thesis project focuses on the RISCV-DV framework as the main method of simulation-based verification. The general test flow for RISCV-DV is shown in Fig. 3.1. The test starts with the ISG generating a constrained set of assembly tests. After that, the tests are compiled and converted into two executable formats: one for the RTL simulation and the other for the ISS simulation. Each simulation produces a trace, which is then used in a comparison script to check the correctness of the DUT. A trace contains information about the instruction issued, which registers are affected by the instruction and their content. In more detail, a trace contains:

- The program counter (PC)
- The type of instruction executed
- The GPR affected by the instruction (if any) and its content
- The CSR affected by the instruction (if any) and its content
- The opcode encoding the instruction
- The privilege mode
- The instruction string, containing the instruction issued and its operands
- Two control bits that flag if the instruction is valid and if an exception occurred

However, not all of these fields are considered for comparison. The fields that are always expected to be equal are the PC and the opcode. The GPR and CSR are instead compared only for instructions that affect the content of one of the registers. The instruction and its operands are not compared since they are disassembled differently in the ISS and the RTL trace. If during the comparison step a mismatch is discovered, the test that failed is reported and subsequently reviewed for potential bugs.

Errors can take many forms, a few examples being, (silent) data corruption, illegal memory access, and faulty exceptions. Detection of data corruption involves scan-

ning architectural registers for erroneous content and checking it against a known-good reference. Exceptions are also good to track, as they may trigger recovery routines that during normal operations should not be triggered, meaning a fault has occurred.

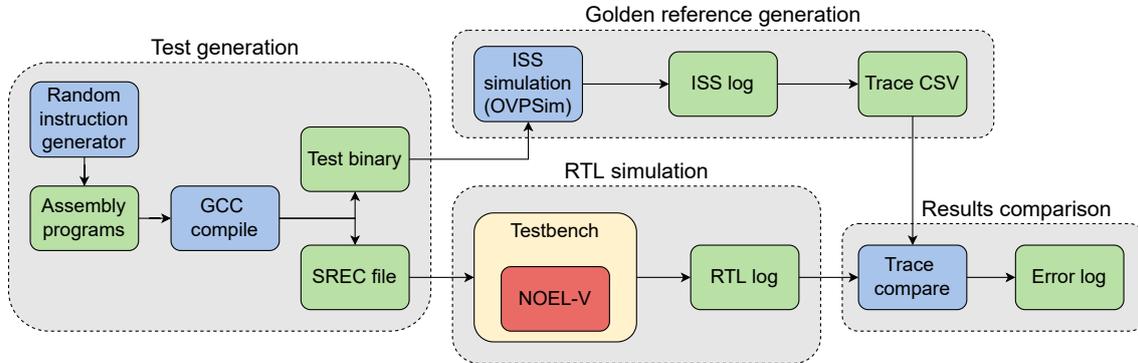


Figure 3.1: RISC-V-DV test flow

Coverage metrics

RISC-V-DV can extract functional coverage from the ISS simulation trace. Extracting the coverage from the ISS trace is fine for tests that had identical traces. Therefore every test generated by the ISG that succeeds is scanned for functional coverage. The instructions traced are then binned into functional covergroups. Each group reports how many of the instructions were executed under the special conditions specified. For example, to achieve 100% coverage for the "addi" instruction, the following cases are considered:

- Each GPR has to be used as a source register at least once
- Each GPR has to be used as a destination register at least once
- The content of the source/destination register has to be a positive/negative number at least once for each combination
- The immediate value has to be a positive/negative number at least once for each case
- The following data hazards have to occur at least once: RAW, WAR, and WAW
- All the sign combinations between the addends and the sum have to occur, in total eight combinations

Each coverage point hit contributes to the overall functional coverage, hence the more tests performed, the higher the coverage achieved. However, the functional coverage does not scale linearly to the number of tests, as some corner cases might remain uncovered, while more common situations might occur more often than necessary. Therefore, despite having run several tests it may still be necessary to further tailor tests that target the uncovered corner cases.

3.2 Tailored Tests for a Functional Unit

After the first verification approach, which is more general and tests the design at the ISA level, a second approach is employed, targeting specific functional units at the RTL level. Targeting a specific functional unit instead of the whole design allows for more thorough testing, and thus higher confidence in the reliability of the DUT. In addition, ISA level verification cannot fully exercise implementation-specific features, mainly since only the content of registers is visible. Examples of functional units are: arithmetic logic unit (ALU), FPU, MMU and PMP. For such units, a tailor-made verification strategy can be composed, that can include simulation-based verification, which tests the design, as well as formal verification, which proves the design.

3.2.1 Formal Verification of the PMP Unit

A functional unit particularly important to prove, rather than test, is the PMP unit since it is a security feature. The PMP implementation in NOEL-V is based on two VHDL procedures. The first one, `precalc`, calculates the address limits of the configured region and then sends this result to the second procedure, which decides whether the access requested is granted or not as shown in Fig 3.2. The DUT is bounded to a SystemVerilog module, that contains the assertions needed to be proven, as well as assumptions on legal input values. These assertions are derived from the RISC-V specifications [14], and they are checked for every possible grain configuration. The assertions used to formally prove the PMP unit are the following:

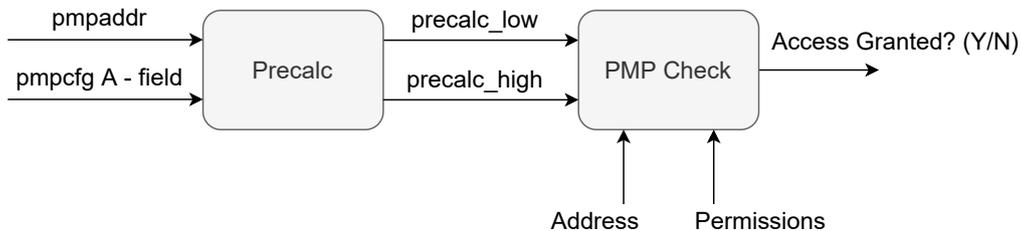


Figure 3.2: Schematic flow of the PMP operation

- **Grant access to a configured area:** if the access requested falls entirely within a PMP region (this is defined as a hit), and the configuration of that region grants access to the corresponding type (read, write or execute), then the unit should grant access
- **Default situation for M modes:** if the effective privilege mode is M mode, and the access requested either does not hit any entry, or hits an entry with the lock bit set to zero then the unit should grant access
- **Default situation for S and U modes:** if the access request does not fall within a PMP region, and the privilege is not M mode, then the unit should deny access
- **Modified privilege mode with no hit:** if the access requested does not fall within a PMP region, and it (the access requested) is not of type execution,

and the actual privilege is not M mode (i.e. MPRV set and MPP not M mode), then the unit should deny access

- **Modified privilege mode with a hit:** if the access requested falls entirely within a PMP region, and the configuration of that region grants access to the corresponding type (read or write, but not execute), and the actual privilege is not M mode (i.e. MPRV set and MPP not M mode), then the unit should grant access
- **Lock bit set:** if the access requested falls entirely within a PMP region, and the lock bit is set, and the configuration of that region does not grant access to the corresponding type (read, write or execute) then the unit should deny access

The input space is constrained by the following assumptions:

- **Address-match mode support:** if the granularity is greater than zero, the mode NA4 is not supported.
- **Valid privilege mode:** the privilege mode can only be of three types: machine mode (M, encoded as 11), supervisor mode (S, encoded as 01), and user mode (U, encoded as 00). Therefore, the combination "10" for the privilege mode is not valid.
- **Address validity:** if the mode is NAPOT, then the bits of `pmpaddr` $\in [0, \text{granularity} - 2]$ read as all ones. If the mode is TOR instead, then the bits of `pmpaddr` $\in [0, \text{granularity} - 1]$ read as all zeros.

However, not every line of the specifications concerning physical memory protection can be proven with the chosen DUT. For example, the write protection of PMP configuration register (`pmpcfg`) and the corresponding `pmpaddr` provided by the lock bit cannot be proven, since the DUT does not implement the CSR write stage. Furthermore, a failed access should generate a load, store, or access exception, depending on the nature of the requested access. This situation cannot be proven within this scope, since exceptions are generated elsewhere. PMP specifications related to page-based virtual memory are also impossible to test since the DUT only has access to physical addresses.

The SystemVerilog code that implements the formal proofs is provided in Appendix A. The formal verification was performed in Cadence JasperGold.

4

Results

This chapter aims to present and describe the problems found in the designs which were tested. The chapter is divided according to the test that enabled the discovery of a flaw in the design.

4.1 RISC-V-DV

RISC-V-DV was applied to the dual-issue, RV64GC [13] version of the processor and a few issues were uncovered.

4.1.1 Bugs found

Unintended Exception triggered by a failing PMP check

The first bug that was found was related to PMP. When the PMP granularity of the architecture was set to 0, an unexpected exception was triggered, as shown in Table 4.1. Since an exception was triggered, an `addi` instruction is inserted in the pipeline to flush it, which is why the instruction is also marked invalid. Table 4.2 shows the intended behavior which in this case was a jump to 176c. Instead, as an exception was generated the `stvec` exception handler is called. The RISC-V specification allows for a PMP granularity of 0 meaning the configuration of a memory region of 4 bytes. The reason an exception was triggered has to do with a failing PMP check. When NOEL-V is configured with a dual-issue pipeline, two instructions are fetched from memory at the same time. Each instruction is 4 bytes, which in total means a memory access of 8 bytes. Since 8 bytes is larger than the size of the configured PMP region, the access tries to fetch from a protected area causing an exception.

Table 4.1: RTL simulation of failing dual-issue instruction

pc	instr	gpr	opcode	instr_str	valid	expt
00000144	<code>auipc</code>	<code>s8:0002e144</code>	<code>0002ec17</code>	<code>auipc s8, 0x0002e</code>	1	0
00000148	<code>addi</code>	<code>s8:0002dc38</code>	<code>af4c0c13</code>	<code>addi s8, s8, -1292</code>	1	0
0000014c	<code>jal</code>	<code>x0:00000150</code>	<code>6200106f</code>	<code>jal x0, 5664</code>	1	0
0000176c	<code>addi</code>	<code>x0:0f3b6a8e</code>	<code>0004</code>	<code>addi x0, x0, 1</code>	0	1
00001000	<code>jal</code>	<code>x0:00001004</code>	<code>0400006f</code>	<code>jal x0, 64</code>	1	0
00001040	<code>csrrw</code>	<code>s8:00000000</code>	<code>340c1c73</code>	<code>csrrw s8, mscratch, s8</code>	1	0

Table 4.2: ISS simulation of failing dual-issue instruction

pc	instr	gpr	opcode	instr_str
00000144	auipc	s8:0000000000002e144	0002ec17	auipc s8,0x2e
00000148	addi	s8:0000000000002dc38	af4c0c13	addi s8,s8,-1292
0000014c	jal	nan	6200106f	j 176c
0000176c	auipc	s6:0000000000001976c	00018b17	auipc s6,0x18
00001770	addi	s6:000000000000198b5	149b0b13	addi s6,s6,329
00001774	slliw	a3:0000000000000000	0170169b	slliw a3,x0,0x17

Faulty Decode

The second bug was found in a hint instruction test. The bug was a failing decode for a compressed hint instruction. In Table 4.3 the correct instruction decode for opcode 6061 is shown. The correct decode shows that the immediate value 0x18 is supposed to be loaded into the upper bits of register x0. Register x0 is by default hardwired to 0x0 meaning that the hint instruction should not change the architectural state. However, as shown in Table 4.4, the opcode 6061 is incorrectly decoded to an add immediate instruction with the wrong architectural destination register. This means that the architectural state was changed when not intended. The consequence of this is seen a few instructions later when a subtraction instruction tries to use the register as a source. Register s9 is now assigned the incorrect value causing the GPR comparison to fail.

Table 4.3: ISS simulation of failing hint instruction

pc	instr	gpr	opcode	instr_str
00009cba	lwu	nan	3f4eef83	lwu t6,1012(t4)
00009cbe	sb	nan	3e8e8a23	sb s0,1012(t4)
00009cc2	c.lui	nan	6061	lui zero,0x18
00009cc4	sraw	s11:ffffffffffffffff	41bdddbb	sraw s11,s11,s11
00009cc8	ld	t0:00000000000000f9	3f4eb283	ld t0,1012(t4)
00009ccc	lh	s11:00000000000000f9	3f4e9d83	lh s11,1012(t4)
00009cd0	lb	gp:ffffffffffffffff	3f4e8183	lb gp,1012(t4)
00009cd4	sd	nan	3e2eba23	sd sp,1012(t4)
00009cd8	lh	s11:0000000000002810	3f4e9d83	lh s11,1012(t4)
00009cdc	lbu	a5:0000000000000010	3f4ec783	lbu a5,1012(t4)
00009ce0	subw	s9:fffffffffb7d01	41840cbb	subw s9,s0,s8
00009ce4	c.li	a7:ffffffffffffffff	58fd	li a7,-1
00009ce6	lb	gp:0000000000000010	3f4e8183	lb gp,1012(t4)

Table 4.4: RTL simulation of failing hint instruction

pc	instr	gpr	opcode	instr_str
00009cba	lwu	t6:0000000000000000	3f4eef83	lwu t6, 1012(t4)
00009cbe	sb	s4:00000000000000f9	3e8e8a23	sb s0, 1012(t4)
00009cc2	addi	s0:000000000002281c	6061	addi s0, sp, 12
00009cc4	sraw	s11:fffffffffffffff	41bddd8b	sraw s11, s11, s11
00009cc8	ld	t0:00000000000000f9	3f4eb283	ld t0, 1012(t4)
00009ccc	lh	s11:00000000000000f9	3f4e9d83	lh s11, 1012(t4)
00009cd0	lb	gp:fffffffffffffff9	3f4e8183	lb gp, 1012(t4)
00009cd4	sd	s4:0000000000022810	3e2eba23	sd sp, 1012(t4)
00009cd8	lh	s11:0000000000002810	3f4e9d83	lh s11, 1012(t4)
00009cdc	lbu	a5:0000000000000010	3f4ec783	lbu a5, 1012(t4)
00009ce0	subw	s9:fffffffffffecb24	41840cbb	subw s9, s0, s8
00009ce4	addi	a7:fffffffffffffff	58fd	addi a7, x0, -1
00009ce6	lb	gp:0000000000000010	3f4e8183	lb gp, 1012(t4)

4.1.2 Coverage

The coverage reached in RISC-V-DV was in total 96.47%. A detailed table of all the tested covergroups and their respective score is shown in Table 4.5

Table 4.5: Percentage reached for each covergroup

NAME	SCORE	NAME	SCORE
c_slli	75.00	csrrw	100.00
fnmsub_d	78.82	sraiw	100.00
fnmadd_d	79.17	fcvt_s_w	100.00
fmadd_d	79.34	branch_hit_history	100.00
fsgnjx_d	79.81	mulw	100.00
fsgnj_d	79.81	ori	100.00
fsgnjn_d	79.81	slli	100.00
fmsub_d	79.86	srli	100.00
rv32i_misc	80.00	sll	100.00
fsqrt_d	80.56	addw	100.00
fmin_d	81.03	sltu	100.00
fdiv_d	81.25	ld	100.00
fadd_d	81.25	mulhu	100.00
fsub_d	81.25	c_and	100.00
fmul_d	81.25	fcvt_wu_s	100.00
fmax_d	81.25	xori	100.00
flt_d	84.09	lbu	100.00
fle_d	84.09	compressed_opcode	100.00
feq_d	84.09	add	100.00
fcvt_wu_d	85.42	c_mv	100.00

Continued on next page

4. Results

Table 4.5: Percentage reached for each covergroup (Continued)

NAME	SCORE	NAME	SCORE
fcvt_lu_d	85.42	c_subw	100.00
fld	85.71	c_ld	100.00
fsd	86.61	c_beqz	100.00
fmv_x_d	87.50	slliw	100.00
csrrsi	87.50	mulh	100.00
csrrwi	87.50	c_lwsp	100.00
fcvt_w_d	87.50	slti	100.00
fcvt_l_d	87.50	slli64	100.00
fclass_d	87.50	c_lw	100.00
csrrci	87.50	fmv_w_x	100.00
fcvt_s_d	88.89	fsgnj_s	100.00
fcvt_d_s	90.28	srai	100.00
lui	91.67	c_jr	100.00
auipc	91.67	c_jalr	100.00
fsqrt_s	94.44	lwu	100.00
rem	96.30	sraw	100.00
div	96.30	andi	100.00
remw	96.67	c_srli	100.00
divw	96.67	c_sub	100.00
opcode	96.88	c_add	100.00
and	97.22	c_sd	100.00
or	97.22	fsgnjx_s	100.00
xor	97.22	divuw	100.00
fnmsub_s	97.57	divu	100.00
fmsub_s	97.92	sub	100.00
fdiv_s	98.21	fcvt_s_lu	100.00
fadd_s	98.21	bge	100.00
fsub_s	98.21	c_addw	100.00
fnmadd_s	98.26	fmv_x_w	100.00
fmadd_s	98.61	fclass_s	100.00
fmin_s	98.66	c_sw	100.00
fmul_s	98.88	mulhsu	100.00
c_addi	98.92	lb	100.00
csrrs	98.96	lh	100.00
flw	99.11	slt	100.00
fmax_s	99.11	c_li	100.00
sb	99.38	lhu	100.00
fcvt_s_l	99.38	srai64	100.00

Continued on next page

Table 4.5: Percentage reached for each covergroup (Continued)

NAME	SCORE	NAME	SCORE
sd	99.48	c_bnez	100.00
sh	99.48	c_j	100.00
sw	99.48	lw	100.00
fsw	99.55	mul	100.00
fsgnjn_s	99.76	remuw	100.00
beq	100.00	c_sdsp	100.00
srlw	100.00	c_andi	100.00
illegal_compressed_instr	100.00	srli64	100.00
subw	100.00	srlw	100.00
fcvt_l_s	100.00	c_addiw	100.00
fcvt_d_lu	100.00	fcvt_d_l	100.00
c_addi16sp	100.00	addiw	100.00
flt_s	100.00	fcvt_d_wu	100.00
sra	100.00	feq_s	100.00
fcvt_w_s	100.00	c_swsp	100.00
mepc_alignment	100.00	sllw	100.00
hint	100.00	fle_s	100.00
fmv_d_x	100.00	remu	100.00
c_xor	100.00	c_lui	100.00
srl	100.00	blt	100.00
c_or	100.00	csrrc	100.00
bne	100.00	fcvt_lu_s	100.00
c_addi4spn	100.00	fcvt_s_wu	100.00
addi	100.00	bgeu	100.00
c_ldsp	100.00	fcvt_d_w	100.00
bltu	100.00	c_srai	100.00
jal	100.00	sltiu	100.00
jalr	100.00		

4.2 Formal Verification

Formal verification was applied to the PMP unit in which a few different flaws were uncovered; these are presented in the following section.

4.2.1 Bugs found

TOR null range

When in TOR mode two address registers are used to set the memory region. If $\text{pmpaddr}(i) \leq \text{pmpaddr}(i-1)$, i.e the range is reversed or null, then the specification states that no match for that entry should occur. In fact, an address that fulfills the inequality expression $\text{pmpaddr}(i-1) \leq \text{address} < \text{pmpaddr}(i)$ does not exist. When two consecutive entries hold the same address, and the current one ($\text{pmpaddr}(i)$) is configured as TOR, the correct behavior would be that no address hits that entry. In NOEL-V, the original expression is implemented as $\text{pmpaddr}(i-1) \leq \text{address} \leq \text{pmpaddr}(i) - 1$. However, the comparison is performed without including the LSB. This implies that an address which fulfills the inequality expression exists (see Fig. 4.1).

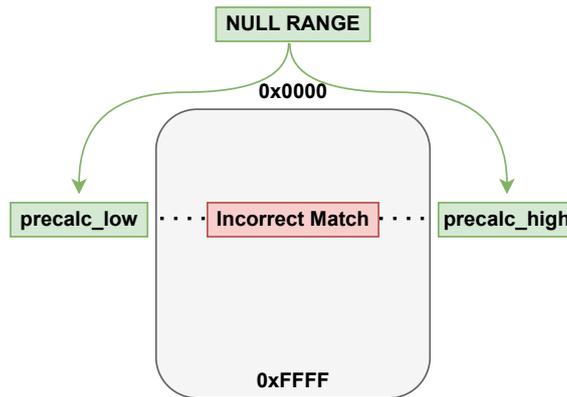


Figure 4.1: TOR null range bug where $\text{pmpaddr}(i-1) = \text{pmpaddr}(i) - 1 = \text{address}$

The TOR null range only appears when $\text{pmpaddr}(i)$ and $\text{pmpaddr}(i-1)$ are odd numbers, and the granularity is set to zero: in fact the subtraction $\text{pmpaddr}(i) - 1$ affects only the LSB, which is ignored in the comparison. The code shown in Listing 1 presents the simplified code in which the bug appears.

Underflow for PMP entries configured as TOR

A second TOR bug appears when a pmpaddr is set to zero. The design implements a precalc scheme which calculates the low and high address based on the address mode and the content of the pmpaddr register. The bug was found in the precalc scheme when in TOR mode. To hit a TOR entry, the address has to be within the range $\text{pmpaddr}(i-1) \leq \text{address} < \text{pmpaddr}(i)$. However, instead of checking the upper bound directly, the precalc unit first subtracts one from $\text{pmpaddr}(i)$ and then checks $\text{pmpaddr}(i-1) \leq \text{address} \leq \text{pmpaddr}(i) - 1$ which may seem like a perfectly

```

variable pmpaddr_i   : unsigned(msb downto 0) := x"33"; -- 0011 0011
variable pmpaddr_im1 : unsigned(msb downto 0) := x"33"; -- 0011 0011
variable address     : unsigned(msb+2 downto 0) := x"0cc"; -- 00 1100 1100
variable hit         : boolean                 := false;
variable align       : integer                 := 0;

-- alignment according to the granularity pmp_g
if pmp_g > 1 then
    align := pmp_g - 1;
end if;

if (pmpaddr_im1(msb downto 1+align) <= address(msb+2 downto 3+align) and
    pmpaddr_i(msb downto 1+align)-1 >= address(msb+2 downto 3+align)) then
    hit := true;
end if;

```

Listing 1: Simplified NOEL-V PMP code

fine substitution. But if $\text{pmpaddr}(i)$ is zero the subtraction causes an underflow, resulting in the upper bound being set to the maximally representable value, as shown in Fig. 4.2. The size of the affected region depends on the lower bound, if $\text{pmpaddr}(i-1) = 0x4$ and $\text{pmpaddr}(i) = 0x0$ then almost the entire memory range will be configured. If instead $\text{pmpaddr}(i-1) = 0xFFFFE$ and $\text{pmpaddr}(i) = 0x0$ then a minuscule region of the memory is incorrectly configured. The most extreme case would be to cover the entire memory with the highest priority, which occurs if $\text{pmpaddr}(0) = 0$. Then the lower bound is implicitly set to zero and the upper bound underflows to intmax .

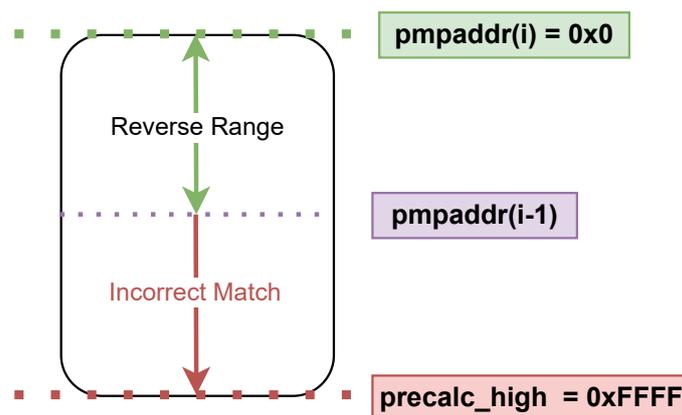


Figure 4.2: Incorrect TOR range returned when $\text{pmpaddr}(i) = 0$, expected behavior is to not hit. The reversed range is turned into a valid range by the precalc scheme leading to an incorrect match from $\text{pmpaddr}(i)$ to the end of the memory.

NAPOT range overflow

When configuring NAPOT ranges, as introduced in Sec. 2.2.4 the location of the first zero in the `pmpaddr` sets the size of the NAPOT region. The RISC-V specification [14] allows a `pmpaddr` to be set as all ones, i.e there is no zero present in the address. The intended behavior of this configuration is to configure more than the entire address range, which in practice means the entire memory. However, when configured with this address `NOEL-V` returns a null range. The reason for this is that the precalc scheme determines the upper address bound incorrectly. In Fig. 4.3 it is shown that the precalc high address is determined by the first zero which is set by the reserved carry bit. This bit is then inverted and a one is added to the entire vector which causes an overflow. The resulting upper bound then becomes 0 which is the same as the lower bound. The following inequality must be satisfied for a valid address range $low \leq addr < high$ if $low = high$ then this inequality can never be fulfilled, hence the null range.

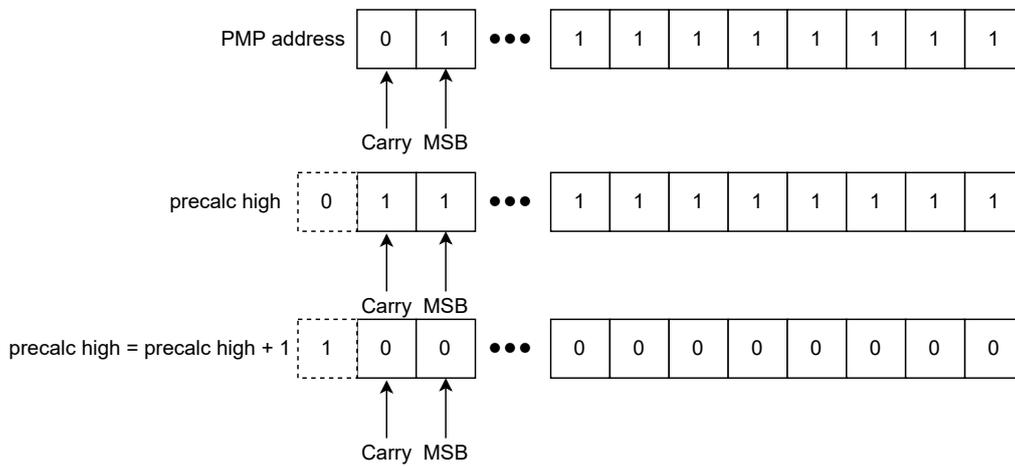


Figure 4.3: Overflow bug for a `pmpaddr` configured with all ones, resulting in a null range.

5

Discussion

This section aims to explore the bugs found, their system implication, and the solution provided by the designers. Furthermore, a discussion about coverage achieved is presented.

5.1 RISC-V-DV

RISC-V-DV was shown to be a competent tool offering a vast variety of edge cases. A few discoveries were made using RISC-V-DV. As shown in Sec. 4.1.1, an unintended exception was discovered when the PMP granularity was set to zero. One could argue that this is not a bug but rather an invalid configuration, since NOEL-V's dual-issue pipeline does not support a granularity of zero. One solution to this configuration issue could be to check if a dual-issue configuration is set or not before letting the granularity be configured to zero.

The second bug that was found was an invalid decode and, in this case, the bug caused the program to change the architectural state of the processor, specifically the content of the s0 GPR. The corrupted GPR content can lead to incorrect behavior depending on how the value is used. The source of the bug was a non-covered case in the decoding logic causing the decode to fall back to the "others" case. To fix this bug, an additional case had to be considered in the decode logic.

In regards to coverage, RISC-V-DV on average achieved 96.47% functional coverage based on the 171 covergroups provided by the framework (see Table 4.5). However, the coverage mechanism present in the framework is still under active development, which means that there may be flaws in the coverage collection. In particular, there is reason to believe that flaws are present in the tracking of special floating-point values, such as $\pm\infty$ and $\pm\text{NaN}$. These values are never covered in the coverage report, despite being present in the traces. As a result, the percentage of coverage achieved is likely greater than the reported value 96.47%.

5.2 Formal Verification of PMP

Formal verification was shown to be a very powerful approach, allowing properties to be proven for a limited number of cycles or an unbounded number of cycles. Properties that were no longer than a few lines long could quickly be disproven by JasperGold despite there being multiple signals with widths over 50 bits, something which in simulations would take a very long time to cover. The bugs found in the PMP unit are of critical importance since the PMP unit is a security feature

mainly used to separate high-privilege code from user-level code. The possibility of letting user-level memory and machine-level memory mix could therefore have severe consequences.

The first bug, where two consecutive pmpaddr entries are strictly equal, is potentially something that can occur in a production environment, and also something which a programmer may not discover when testing their code. If not detected during hardware verification, a programmer would have to test the code carefully with multiple accesses in different memory regions. Therefore, regions that were supposed to be protected could now be given access permissions. It is quite unlikely that a programmer would write tests that exercise architectural specifications. Furthermore, if the code was initially written and tested on a different but similar processor, or in an ISS, then it is likely that the code would be reused without extensive testing. One such scenario of reuse could be the use of a secure execution environment such as Keystone. This bug affects only a small portion of the entire memory in a specific configuration, i.e., when the granularity is set to zero. However, a hit in a higher priority entry could affect potential hits in less prioritized entries. The code shown in Listing 2 presents the modifications that could be done in the code to fix the bug.

```
variable pmpaddr_i   : unsigned(msb downto 0) := x"33"; -- 0011 0011
variable pmpaddr_im1 : unsigned(msb downto 0) := x"33"; -- 0011 0011
variable address     : unsigned(msb+2 downto 0) := x"0cc"; -- 00 1100 1100
variable hit         : boolean                 := false;
variable align       : integer                 := pmp_g - 1;

if (pmpaddr_im1(msb downto 1+align) <= address(msb+2 downto 3+align) and
    pmpaddr_i(msb downto 1+align) > address(msb+2 downto 3+align)) then
    hit := true;
end if;
```

Listing 2: Simplified PMP code that fixes the TOR null range bug

The second bug, where $\text{pmpaddr}(i) = 0$ and the address mode is TOR, is likely the most severe, since adherence to the RISC-V specification is not fulfilled. In software development, specification adherence is assumed by the programmer who configures the PMP entries. Therefore, software tests that exercise specification-level details will probably not be implemented. The implication of the bug is that, if not tested properly, all or some regions of the memory could be given full access privileges. Consequently, there would be no hardware-level separation between user-level and machine-level code. Furthermore, if software is reused it is even less likely that the bug would be found in the configuration phase and hence the bug may propagate into a production environment. To fix the bug the designer chose to revert to the RISC-V inequality expression which does not subtract one from the upper bound: $\text{precalc_low} \leq \text{address} < \text{precalc_high} = 0$. With the original expression, the upper bound can not underflow and thus no address can fulfill the inequality.

The third bug, where a NAPOT range's size is determined by its leading zero is likely less critical than the second one. In this case, when a PMP region is not configured, only M mode can access the memory, which means that leaks between

user-level and machine-level memory would not be possible. However, this is under the assumption that subsequent PMP entries are disabled. Since the PMP scheme expects static prioritization, a programmer may skip disabling subsequent entries, as an entry that configures the entire memory should always be hit. If the prioritization mechanism fails, the content of the subsequent entries may contain unintended configurations which could expose memory in unintended ways. Assuming that all subsequent entries are disabled, the bug should be relatively easy to discover. If configured, a few accesses from S and U mode would be enough to determine that access is not granted when expected. At this moment in time, this configuration is also redundant as it configures more than the entire address range. Therefore, if one still wants to configure the entire memory it is possible by setting a zero in the MSB position. A possible fix to this bug would be to add a second carry bit to the `mmpaddr`. In this way, no overflow could occur, as shown in Fig. 4.3.

All PMP bugs have one thing in common and that is that only machine-level code could configure the address and configuration registers. Therefore, it would be hard or impossible for user-level code to exploit these bugs unless the setup code already exposes them. However, if the bugs were not discovered when testing the machine-level code, it is possible that a malicious program could exploit the vulnerabilities to gain access to sensitive data and also gain machine-level privileges.

The implementation for a PMP granularity of zero has shown to be problematic. Given that the PMP regions are aligned according to the granularity, if an access is less or equal to the smallest possible PMP region, then it is automatically aligned. This means that no access can fall half within and half outside a PMP region, therefore checking that `precalc_low ≤ addr < precalc_high` is sufficient to determine a hit. If instead an access is greater than the smallest possible PMP region, there is a third option, which is that an access can fall half within and half outside a region. Therefore, to hit an entry, an additional condition should be fulfilled: `precalc_low ≤ addr + access < precalc_high`. To support the extra condition, additional comparators would have to be implemented for every PMP entry, something which comes at the cost of both extra delay and area cost. Thus, the designer chose to no longer support a PMP granularity of zero. The usefulness of being able to configure regions as small as four bytes in comparison to eight bytes is also something that could be debated. Subsequently, dropping support for zero granularity stands as a balanced compromise. If the processor is configured with an MMU then memory could only be accessed in 4 kB chunks anyway, therefore the trade-off of not supporting a granularity of zero is minimal.

5.3 Assessment of the verification efforts

RISCV-DV has the benefit of testing many aspects of the processor for common hazards and edge cases. Using RISCV-DV is therefore something that can be done during every design iteration, especially in the early design phase when control logic bugs are most likely to occur. The usefulness of RISCV-DV likely decreases when the implementation has reached a certain maturity. NOEL-V has been under development for several years, hence RISCV-DV was not able to uncover more than two bugs. For a more mature implementation, formal verification was able to uncover

bugs that would have been difficult to find in simulation. The bugs found with formal methods required a very particular configuration as well as special `pmpaddr`, something that would take careful consideration or a broad range of randomized test vectors in simulation. For PMP only a subset of the specification was formally verified, but there is still work to do to ensure that the PMP CSRs can be written only under the proper circumstances. The reason this was not done in this work is because the CSR writing is handled in a completely different section of the code and that a combined verification effort targeting all CSR properties is better to avoid breaking the abstraction layers and introducing unnecessary verification fragmentation.

The initial choice to begin wide and then narrow the testing to a few functional units seems to have been a good approach. To further improve the coverage of NOEL-V, more directed testing of other functional units is probably a good path to continue on. Since RISC-V is still an ISA under development [13], it is likely that more open-source frameworks where collaborative efforts can help build a strong verification framework in which discoveries made by partner companies can help with the continual improvement of the framework.

5.4 Future work

Like most verification work there is always more to do. To further verify NOEL-V, the implementation of the RVFI interface would be useful to provide support for the RISC-V formal framework. Since formal checks were able to expose flaws in the PMP unit, more formal testing may have the potential of exposing other bugs. A few areas suggested by the design team to perform formal verification on are the forwarding logic of the integer pipeline and the FPU pipeline. In particular, one source of bugs that has bothered the developer team is mechanisms related to data forwarding in the integer pipeline. To improve the coverage of the forwarding mechanisms, further formal proofs could be designed. Simulation-based testing is already performed to some extent but the introduction of more formal reasoning would likely help increase the trustworthiness of the design further.

6

Conclusion

In conclusion, verification is an increasingly important process in digital design, as has been stated accounting on average for more than 50% of the total development effort. The thesis has explored two paradigms of verification, one based on simulations at the ISA level and one based on formal verification of the PMP unit. Both methods were able to expose flaws in the design. With the simulation-based framework RISC-V-DV two issues were found: the first related to instruction decoding, and the second related to conflicting design generics. The formal approach was able to uncover three different bugs, all of which were critical in nature as they had the potential of granting non-privileged memory accesses to privileged memory regions. Furthermore, the methods were able to complement each other in their findings of bugs, as there was no overlap between the bugs. This result indicates that in order to reach thorough verification, many approaches and many minds have to contribute to the effort. To further extend the verification, more directed testing would be preferential, since RISC-V-DV was only able to uncover a few bugs despite testing many different scenarios. This happens since NOEL-V, from its earliest development stages, has mostly been tested with a similar ISG. To further verify the design, the directed testing can consist of either formal or simulation-based methods, depending on the nature of the DUT. The tests should be written specifically to reach higher coverage for the DUT, something which may be hard to achieve at the ISA level.

The work done in this thesis adds another processor to the list of designs tested with RISC-V-DV. It also suggests formal properties for RISC-V's PMP scheme and implements a PMP testbench for NOEL-V. The testbench sticks to RISC-V's openness paradigm, which is why it is publicly available and quite general, and thus it can be tailored to be used in a different design.

Bibliography

- [1] H. Foster. “The Weather Report: 2018 Study On IC/ASIC Verification Trends.” (2019), [Online]. Available: <https://semiengineering.com/the-weather-report-2018-study-on-ic-asic-verification-trends/>.
- [2] D. Margan and S. Čandrić, “The success of open source software: A review,” in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2015, pp. 1463–1468. DOI: 10.1109/MIPRO.2015.7160503.
- [3] S. Di Mascio, A. Menicucci, E. Gill, G. Furano, and C. Monteleone, “Leveraging the Openness and Modularity of RISC-V in Space,” *Journal of Aerospace Information Systems*, vol. 16, no. 11, pp. 454–472, 2019. DOI: 10.2514/1.I010735.
- [4] S. Di Mascio, A. Menicucci, G. Furano, C. Monteleone, and M. Ottavi, “The Case for RISC-V in Space,” in *Applications in Electronics Pervading Industry, Environment and Society*, S. Saponara and A. De Gloria, Eds., Cham: Springer International Publishing, 2019, pp. 319–325.
- [5] Google. “RISC-V DV.” (2019), [Online]. Available: <https://github.com/google/riscv-dv> (visited on 01/20/2022).
- [6] S. Ahmadi-Pour, V. Herdt, and R. Drechsler, “Constrained Random Verification for RISC-V: Overview, Evaluation and Discussion,” in *MBMV 2021; 24th Workshop*, 2021, pp. 1–8.
- [7] lowRISC. “Ibex.” (2017), [Online]. Available: <https://github.com/lowRISC/ibex> (visited on 04/21/2022).
- [8] T. Liu, R. Ho, and U. Jonnalagadda, “Open Source RISC-V Processor Verification Platform,” RISC-V Summit, 2019.
- [9] O. Group. “CVA6 (Ariane).” (2018), [Online]. Available: <https://github.com/openhwgroup/cva6> (visited on 04/21/2022).
- [10] E. Seligman, E. T. Schubert, and K. M. V. A. Kiran, *Formal verification: An essential toolkit for modern VLSI Design*, 1st ed. Elsevier/MK, Morgan Kaufmann is an imprint of Elsevier, 2015.
- [11] K. Cheang, C. Rasmussen, D. Lee, D. Kohlbrenner, K. Asanović, and S. A. Seshia, “Verifying RISC-V Physical Memory Protection,” 2020.
- [12] A. Waterman, “Design of the RISC-V Instruction Set Architecture,” Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan. 2016. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html>.
- [13] A. Waterman and K. Asanović, “The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.1,” RISC-V Foundation, Tech. Rep. 20191213, Dec.

2019. [Online]. Available: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>.
- [14] “The RISC-V Instruction Set Manual Volume II: Privileged Architecture, Document Version 20190608-Priv-MSU-Ratified,” RISC-V Foundation, Tech. Rep., Jun. 2019.
- [15] W. K. Lam, *Hardware Design Verification: A Practical And Systematic Approach*, English, Hardcover. Prentice Hall, Mar. 4, 2005, p. 585.
- [16] H.-P. Wen, L.-C. Wang, and K.-T. Cheng, “CHAPTER 9 - Functional verification,” in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, Eds., Boston: Morgan Kaufmann, 2009, pp. 513–573. DOI: 10.1016/B978-0-12-374364-0.50016-3.
- [17] K. McDermott. “Getting Started With RISC-V Verification.” (May 2020), [Online]. Available: <https://riscv.org/blog/2020/05/getting-started-with-risc-v-verification/> (visited on 01/03/2022).
- [18] S. Davidmann, L. Moore, R. Ho, T. Liu, D. Letcher, and A. Sutton, “Rolling the Dice with Random Instructions is the Safe Bet on RISC-V Verification,” Imperas Software Ltd., Tech. Rep., 2020, Design and Verification Conference (DVCon).
- [19] OpenHW Group. “CORE-V Verification Strategy.” (2019), [Online]. Available: <https://core-v-docs-verif-strat.readthedocs.io/en/latest/> (visited on 02/02/2022).
- [20] SymbioticEDA. “RISC-V Formal Verification Framework.” (2016), [Online]. Available: <https://github.com/SymbioticEDA/riscv-formal> (visited on 11/26/2021).

A

Appendix 1

The SystemVerilog assertions employed to verify the PMP unit are provided. In addition, the following GitHub repository contains all the code developed for the Master Thesis: <https://github.com/TheSaltyLiquorice/MasterThesisNoelV.git>

```
`include "pmp_portmap.svh"
import pmp_pkg::*;
`define M_MODE 2'b11
`define S_MODE 2'b01
`define U_MODE 2'b00
`define NAPOT 2'b11
`define NA4 2'b10
`define TOR 2'b01
`define OFF 2'b00
`define XLEN 64
`define PMP_ACCESS_X 2'b00
`define PMP_ACCESS_R 2'b01
`define PMP_ACCESS_W 2'b11
`define HALF_HIT -10
`define NO_HIT -1

module pmp_assertions
  #(
    parameter pmp_check = 1 ,
    parameter pmp_no_tor = 0,
    parameter pmp_entries = 16,
    parameter pmp_g = 10,
    parameter pmp_msb = 55)

  (
    input bit clk300p,
    input bit rstn ,
    input pmpaddr_vec_type pmpaddr,
    input word64 pmpcfg0 ,
    input word64 pmpcfg2,
    input bit[pmp_msb:0] address,
    input bit [1:0] acc ,
    input bit [1:0] size,
```

```
input bit [1:0] prv ,
input bit mprv ,
input bit [1:0] mpp ,
input bit valid ,
input bit ok
);
function int read_access_perm(
    word64 pmpcfg0,
    word64 pmpcfg2,
    bit [1:0] acc, // access x/w/r
    int index // cfg index
);
begin
    automatic int offset = -1;
    if(acc == `PMP_ACCESS_X)
        offset = 2;
    else if(acc == `PMP_ACCESS_W)
        offset = 1;
    else if(acc == `PMP_ACCESS_R)
        offset = 0;
    if(offset >= 0 && index >= 0) begin
        if(index < 8)
            return int'(pmpcfg0[index*8+offset]);
        else if(index < pmp_entries)
            return int'(pmpcfg2[(index-8)*8+offset]);
        end else
            return -1;
    end
endfunction

function int read_l_bit(
    word64 pmpcfg0,
    word64 pmpcfg2,
    int index // cfg index
);
begin
    if(index >= 0 && index < 8)
        return int'(pmpcfg0[index*8+7]);
    else if(index >= 8 && index < pmp_entries)
        return int'(pmpcfg2[(index-8)*8+7]);
    else
        return -1;
    end
endfunction
```

```

function bit[1:0] read_address_mode(
    word64 pmpcfg0,
    word64 pmpcfg2,
    int index // cfg index
);
begin
    if(index >= 0 && index < 8)
        return pmpcfg0[index*8+4 -:2];
    else if(index >= 8 && index < pmp_entries)
        return pmpcfg2[(index-8)*8+4 -:2];
    else
        return -1;
    end
endfunction

// returns the position of the first y (spec. notation)
function int napot_range_size(pmpaddr_type pmpaddr); begin
    for(int i = 0; i < `pmpaddrbits; i++) begin
        if(pmpaddr[i] == 0)
            return i+1;
        end
    return `pmpaddrbits;
end
endfunction

// returns one if the access is aligned
function int aligned_access(
    bit[1:0] size,
    bit [pmp_msb:0] address);
begin

    case(size)
        3 : begin
            if(address[2:0] == '0)
                return 1;
            end
        2 : begin
            if(address[1:0] == '0)
                return 1;
            end
        1 : begin
            if(address[0:0] == '0)
                return 1;
            end
        default :
            return 1;
    end
endfunction

```

A. Appendix 1

```
    endcase

    return -1;

end
endfunction

// Enforce that NAPOT and TOR entries are valid addresses,
// the sum should be the same as the number of entries if
// all addresses are valid.
function int valid_pmpaddr(
    pmpaddr_vec_type pmpaddr,
    word64 pmpcfg0,
    word64 pmpcfg2);
begin
    automatic int sum = 0;
    for(int i = 0; i < pmp_entries; i++) begin
        automatic bit[1:0] current_mode = read_address_mode(pmpcfg0, pmpcfg2, i);
        if( current_mode == `NAPOT && pmp_g > 1) begin
            if(pmpaddr[i][pmp_g-2:0] == '1)
                sum++;
        end else if(current_mode == `TOR && pmp_g > 0) begin
            if(pmpaddr[i][pmp_g-1:0] == '0)
                sum++;
        end else begin
            sum++;
        end
    end
    return sum;
end
endfunction

// Function checks if we are within the range of a PMP region,
// if we are halfway within or if we miss entirely.
function automatic int check_valid_address(
    pmpaddr_vec_type pmpaddr,
    bit[pmp_msb:0] address,
    word64 pmpcfg0,
    word64 pmpcfg2,
    bit [1:0] size
);
begin

for(int i = 0; i < pmp_entries; i++) begin
    bit[1:0] a = read_address_mode(pmpcfg0, pmpcfg2, i);
```

```

int index = 0; // index only used for NAPOT entries
pmpaddr_type napot_pmpaddr;
bit[pmp_msb:0] napot_address;

if(a == `NAPOT) begin
    index = napot_range_size(pmpaddr[i]);
    // index returns position of first y (specification notation)
    napot_pmpaddr = pmpaddr[i] >> index;
    napot_address = address >> index;
    // if addresses match we have a hit
    if(napot_address[pmp_msb:2] == napot_pmpaddr[`pmpaddrbits-1:0]) begin
        return i;
    end
end
else if(a == `NA4) begin
    if(address[pmp_msb:2] == pmpaddr[i] && size < 3)
        return i;
    else if(address[pmp_msb:3] == pmpaddr[i][`pmpaddrbits-1:1] && size == 3)
        return `HALF_HIT; // Half hit, can't hit in subsequent entries
    end
end

else if(a == `TOR) begin
    longint unsigned address_high = (address+2**((unsigned'(size))-1));
    bit[pmp_msb:2+pmp_g] top_of_range = address_high[pmp_msb:2+pmp_g];
    bit[`pmpaddrbits-1:pmp_g] pmpaddr_high = pmpaddr[i][`pmpaddrbits-1:pmp_g];
    bit[`pmpaddrbits-1:pmp_g] address_trunc = address[pmp_msb:2+pmp_g];

    if(i > 0) begin
        bit[`pmpaddrbits-1:pmp_g] pmpaddr_low = pmpaddr[i-1][`pmpaddrbits-1:pmp_g];
        if(address_trunc < pmpaddr_high &&
            address_trunc >= pmpaddr_low &&
            top_of_range < pmpaddr_high &&
            top_of_range >= pmpaddr_low )
            return i; // Full hit at index i
        else if((address_trunc < pmpaddr_high &&
            address_trunc >= pmpaddr_low) ^^
            (top_of_range < pmpaddr_high &&
            top_of_range >= pmpaddr_low ))
            return `HALF_HIT;
    end
end else begin // only to deal with the the first entry being TOR
    if (address_trunc < pmpaddr_high &&
        address_trunc >= 0 &&
        top_of_range < pmpaddr_high )
        return i;
    else if((address_trunc < pmpaddr_high &&

```

A. Appendix 1

```
        address_trunc >= 0) ^^
        (top_of_range < pmpaddr_high &&
         top_of_range >= 0 ))
        return `HALF_HIT;
    end
end
end
return `NO_HIT;
end
endfunction

// If PMP_G > 0 we should not see mode NA4 in the address matching fields
if(pmp_g > 0) begin
    for(genvar i = 0; i < pmp_entries/2 ; i++) begin
        address_mode : assume property(
            pmpcfg0[i*8+4:8*i+3] != `NA4 &&
            pmpcfg2[i*8+4:8*i+3] != `NA4
        );
    end
end

// implementation specific feature to be able to disable TOR mode
if(pmp_no_tor > 0) begin
    for(genvar i = 0; i < pmp_entries/2 ; i++) begin
        no_tor_mode : assume property(
            pmpcfg0[i*8+4:8*i+3] != `TOR &&
            pmpcfg2[i*8+4:8*i+3] != `TOR
        );
    end
end

assume property( prv != 2'b10 ); // undefined privilege mode
assume property( mpp != 2'b10 ); // undefined privilege mode

for(genvar i = 0; i < pmp_entries; i++) begin
    // We never expect the carry position to be set from the outside,
    // this assumption is from a newer version of the code where a carry
    // bit has been added to fix one of the bugs, in the new NOEL-V code
    // pmpaddr is therefore one bit longer.
    //address_msb_0 : assume property(pmpaddr[i][`pmpaddrbits] == 0);

    // This assumption is to hide a NOEL-V bug,
    // should be excluded once bugs have been fixed
    assume property(pmpaddr[i] > 0);
    if(i > 0)
        // bug related assumption
        assume property(pmpaddr[i-1] != pmpaddr[i]); //eliminates null-range
end
```

```

// Bug related assumption
// address_max : assume property(pmpaddr[i][`pmpaddrbits-1:0] != '1);
end

property grant_perm;
  int index = -1;
  @(posedge clk300p) disable iff(!rstn)
  (( (1, index = check_valid_address(pmpaddr, address, pmpcfg0, pmpcfg2, size)) ##0
    aligned_access(size, address) == 1 &&
    read_l_bit(pmpcfg0, pmpcfg2, index) == 0 &&
    index > `NO_HIT && // we must get a hit to get an okay
    valid_pmpaddr(pmpaddr, pmpcfg0, pmpcfg2) == pmp_entries &&
    // this ensures that all of the pmpaddr are legal
    read_access_perm(pmpcfg0, pmpcfg2, acc, index) == 1
    // for it to not fail, the access permission must be set,
    // not required for M mode but for S & U
  ) |> ok);
endproperty

property m_always_grant;
  int index = -1;
  @(posedge clk300p) disable iff(!rstn)
  (( (1, index = check_valid_address(pmpaddr, address, pmpcfg0, pmpcfg2, size)) ##0
    prv == `M_MODE &&
    valid == 1 &&
    aligned_access(size, address) == 1 &&
    index != `HALF_HIT && // look for an area that that is not a half-hit
    read_l_bit(pmpcfg0, pmpcfg2, index) < 1 &&
    // No l-bit can be allowed since permissions are not set.
    // 0 means no L bit, -1 means index == -1
    valid_pmpaddr(pmpaddr, pmpcfg0, pmpcfg2) == pmp_entries &&
    mprv == 0
  ) |> ok );
endproperty

property s_u_never_grant;
  int index = -1;
  @(posedge clk300p) disable iff(!rstn)
  (( (1, index = check_valid_address(pmpaddr, address, pmpcfg0, pmpcfg2, size)) ##0
    prv != `M_MODE &&
    aligned_access(size, address) == 1 &&
    valid_pmpaddr(pmpaddr, pmpcfg0, pmpcfg2) == pmp_entries &&
    index == `NO_HIT &&
    valid == 1
  ) |> !ok );

```

```
endproperty

// If in M mode and MPRV is set and MPP is (S or U)
property mprv_set;
    int index = -1;
    @(posedge clk300p) disable iff(!rstn)
    (( (1, index = check_valid_address(pmpaddr, address, pmpcfg0, pmpcfg2, size)) ##0
        index == `NO_HIT &&
        valid_pmpaddr(pmpaddr, pmpcfg0, pmpcfg2) == pmp_entries &&
        aligned_access(size, address) == 1 &&
        mprv == 1 &&
        mpp != `M_MODE &&
        valid == 1 &&
        acc != `PMP_ACCESS_X
    ) |> !ok );

endproperty

property mprv_ok_set;
    int index = -1;
    @(posedge clk300p) disable iff(!rstn)
    (( (1, index = check_valid_address(pmpaddr, address, pmpcfg0, pmpcfg2, size)) ##0
        index > `NO_HIT &&
        aligned_access(size, address) == 1 &&
        mprv == 1 &&
        mpp != `M_MODE &&
        valid_pmpaddr(pmpaddr, pmpcfg0, pmpcfg2) == pmp_entries &&
        valid == 1 &&
        read_access_perm(pmpcfg0, pmpcfg2, acc, index) == 1 &&
        acc != `PMP_ACCESS_X
    ) |> ok );

endproperty

property enforce_lock;
    int index = -1;
    @(posedge clk300p) disable iff(!rstn)
    (( (1, index = check_valid_address(pmpaddr, address, pmpcfg0, pmpcfg2, size)) ##0
        index > `NO_HIT &&
        valid == 1 &&
        valid_pmpaddr(pmpaddr, pmpcfg0, pmpcfg2) == pmp_entries &&
        aligned_access(size, address) == 1 &&
        read_l_bit(pmpcfg0, pmpcfg2, index) == 1 &&
        read_access_perm(pmpcfg0, pmpcfg2, acc, index) == 0
    ) |> !ok );
```

```
endproperty

grant_check : assert property(grant_perm);
mprv_mpp_check : assert property(mprv_set);
mprv_mpp_ok_check : assert property(mprv_ok_set);
m_access_check : assert property(m_always_grant);
s_u_access_check : assert property(s_u_never_grant);
lock_check : assert property(enforce_lock);
endmodule
```
