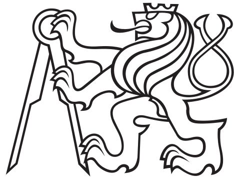
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Microelectronics

The Functional and Formal Verification of The Jump Controller Block for RISC-V Processor

Comparison of Functional and Formal Verification

Jiří Šindelář

Supervisor: prof. Ing. Jiří Jakovenko, Ph.D.
Field of study: Electronics and Communications
Subfield: Electronics
January 2024

MASTER'S THESIS ASSIGNMENT

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

I. Personal and study details

Student's name: **Šindelá Ji í** Personal ID number: **483575**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Microelectronics**

Study program: **Electronics and Communications**

Specialisation: **Electronics**

II. Master's thesis details

Master's thesis title in English:

Functional and Formal Verification of Jump Controller Block for RISC-V Processor

Master's thesis title in Czech:

Funk ní a Formální Verifikace Bloku Jump Controller pro RISC-V Procesor

Guidelines:

1. Study techniques, procedures and tool options for functional verification of digital circuits.
2. Study the possibilities of applying automated verification of formal assertions for the verification of digital circuits.
3. On an appropriate block, demonstrate the use of dynamic and formal verification techniques.
4. Compare and analyze the results achieved and evaluate the benefits of automated formal verification.

Bibliography / sources:

Seligman Erik: Formal Verification: An Essential Toolkit for Modern VLSI Design (ISBN: 9780128007273)
https://www.diva-portal.org/smash/get/diva2:872375/FULLTEXT01.pdf
https://dspace.vutbr.cz/handle/11012/189360
https://dspace.cvut.cz/handle/10467/100975

Name and workplace of master's thesis supervisor:

prof. Ing. Ji í Jakovenko, Ph.D. Department of Microelectronics FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **14.02.2023** Deadline for master's thesis submission: **09.01.2024**

Assignment valid until: **22.09.2024**

prof. Ing. Ji í Jakovenko, Ph.D.	prof. Ing. Pavel Hazdra, CSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student's signature

© VUT v Praze, Design: VUT v Praze, VIC

Acknowledgements

I would like to thank prof. Ing. Jiří Jakovenko, Ph.D. for being the supervisor of this work. I also thank my colleagues at ASICentrum s.r.o. for their support, time, and patience, without which I would not be able to complete this work. Namely, they are Ing. Luboš Hradecký and Ing. Jakub Šťastný, Ph.D. Lastly I want to thank my family for support during the time of my studies.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

In Prague, 09. January 2024

Abstract

In this work, we will present two fundamentally different methods used for the verification of the digital circuit design. These methods are functional verification, which uses time simulation as the source of data for verification, and formal verification, which interacts with the circuit design as if it were a mathematical formula to be solved. We will demonstrate their use in the verification of the design of the RISE branch predictor block created by Martin Laštovka in his bachelor thesis. In the end, we will report and discuss the flaws found in the design and evaluate the strengths of the methods used. The flaws found include wrong updates and access to prediction tables, which cause the design to misbehave.

Keywords: verification, digital verification, UVM, formal verification, function verification

Abstrakt

V této práci představíme dvě metody verifikace návrhu číslicových obvodů, které k tomuto problému přistupují z různách směrů. Těmito motodami jsou funkční verifikace, která používá somulaci pro získání podkladů pro verifikaci, a formální verifikace, která přistupuje k návrhu obvodu jako by se jednalo o matematický vzorec, který je potřeba vyřešit. Jejich použití předvedeme na verifikaci návrhu obvodu prediktoru skoků RISE, který navrhl a vytvořil Martin Laštovka v jeho bakalářské práci. Na konci sepíšeme výsledky obou verifikačních metod a provedeme diskuzi nalezených chyb, které způsobují chybné chování navrženého obvodu. Také zhodnotíme přínos obou metod, které jsme použili. Nalezené chyby zahrnují chybnou úpravu a přístup k predikčním datům, které způsobují to, že návrh se nechová podle očekávání.

Klíčová slova: verifikace, digitální verifikace, UVM, formální verifikace, funkční verifikace

Contents

Abbreviations 1

Introduction 3

Motivation 3

Objective 4

1 Methods 5

1.1 Functional verification 5

1.1.1 Simulation output 6

1.1.2 Self-checking tests 7

1.1.3 Constrained random stimuli . . 7

1.1.4 Blackbox vs whitebox (change of perspective with assertions) 8

1.1.5 UVM [5] 8

1.2 Formal verification 13

1.2.1 Formal equivalence checking (FEC) 14

1.2.2 Formal property verification (FPV) 14

1.2.3 Complexity 15

1.2.4 Languages 16

1.3 Coverage 16

2 Verified design (RISE) 19

2.1 Prediction method 19

3 Verification plan 23

3.1 Design requirements 23

3.2 Design verification 27

3.2.1 Expected behavioral scenarios for verification 28

3.2.2 Formal methods (JasperGold) 29

3.2.3 Functional (UVM) 31

4 Implementation 33

4.1 Formal (Jasper Gold) 33

4.2 Functional (UVM) 34

4.2.1 tb_top 35

4.2.2 test_base 36

4.2.3 m_cfg 36

4.2.4 m_env 36

4.2.5 m_scoreboard 37
4.2.6 m_virt_seqr 37
4.2.7 m_predictor 38
4.2.8 m_control_agent 38
4.2.9 m_memory_agent 38
4.2.10 m_processor_agent 39
4.2.11 Tests 40

5 Results 43

5.1 Results of tests 43
5.1.1 Formal (Jasper Gold) 43
5.1.2 Functional (UVM) 46
5.2 Discussion of results 46
5.2.1 Formal (Jasper Gold) 47
5.2.2 Functional (UVM) 49

6 Conclusion 51

6.1 Implemented files 51
6.2 Comparing the verification approaches 52
6.3 Design flaws 53
6.4 Next steps 53

Bibliography 55

A OBI - implemented functionality 57

B Table of verification methods used for requirements 59

Figures

1.1 Simplified block diagram of functional verification environment. 6

1.2 Block diagram of UVM environment [2]. 10

1.3 Adoption of formal techniques in different years [4]. 13

1.4 Adoption of formal techniques for different design sizes [4]. 14

2.1 Block diagram of RISE by Martin Laštovka. [7] 20

2.2 Block diagram of RISE integration by Martin Laštovka. [7] 20

4.1 Block diagram of implemented formal testbench. 34

4.2 Block diagram of implemented UVM testbench. 35

5.1 Trace of failing design assertion "*asrt_table_read*". 44

5.2 Trace of failing design assertion "*ASRT_RISE_TOP_KILL_SIM_ASRT*". 45

5.3 Simulator output with captured wrong DUT predictor line update and assigning of initial predictor FSM value to the already initialized line. (The trace can be also found as attached file: *screenshots/exp_table_wrong_state_transition.png*). 47

5.4 Simulator output with captured scenario, where DUT updates the wrong line of the FSM predictor table. (The trace can be also found as attached file: *screenshots/exp_table_write_wrong_line.png*). 47

Tables

2.1 IO signals of RISE 21

2.2 Values of design parameters used for verification. 21

6.1 Table comparing different attributes of used verification methods. 52

A.1 List of OBI signals showing which of them are used by the design and tie-off values of unused signals. The *dynamic* tie-off value means that single value can't be assigned to signal and we act as if value of those signals is always correct 57

A.2 Requirement list of the OBI protocol with information about which are relevant for our design. . 58

B.1 List of requirements and which verification methods were used to verify them. (The corresponding columns are marked by 'X') 60

Abbreviations

ABV	-	Assertion-Based Verification
CEX	-	Counter-EXample
DUT	-	Design Under Test
FEC	-	Formal Equivalence Checking
FPGA	-	Field-Programmable Gate Array
FPV	-	Formal Property Verification
FSM	-	Final State Machine
GUI	-	Graphical User Interface
HDL	-	Hardware Description Language
iff	-	if and only if
LSBs	-	Least Significant Bits
MSBs	-	Most Significant Bits
OBI	-	Open Bus Interface
RAM	-	Random Access Memory
RN	-	Random Number
UVC	-	Universal Verification Component
UVM	-	Universal Verification Methodology

Introduction

In this work, we will use two different verification methods to verify the given design. We will call this the design under test (DUT). The two methods we will use are formal verification and functional verification. Both of these methods work by approaching the verification problem very differently.

Motivation

When we create a new digital chip design, we need to check if the design is compliant with our defined specification. This process is called verification, and the design that is verified is called DUT. If we skip the verification process, the design can cause security or safety hazards based on the application. There are two approaches for verification (mentioned above). Both of these methods have their strong and weak sides, coming from their different approaches, and generally complement each other. Because of that, it is almost always more efficient to use both of the verification methods in a reasonable ratio.
Functional verification uses the simulation of the DUT in the time domain and needs the verification environment together with the behavioral model of the DUT to be able to decide if the DUT behaved correctly during simulation. On the other hand, formal verification needs only the DUT and a set of properties to verify them. They both approach the problem from different sides. The functional properties, defined by the verification engineer, are designed to specify the good behavior of the design, and when proven by the formal methods, they tell us that no allowed stimulation of the design will cause it to behave poorly. Opposite to that is functional verification, where

the verification engineer specifies the test to stimulate the design in a way that would cause poor behavior. This also means that the test is designed to catch poor behavior.

Objective

The objective of this work is to use the verification methods mentioned above to verify the correct behavior of the design created by Martin Laštovka in his bachelor thesis **??**. We will create both formal and functional verification environments to use both of the methods in the verification scopes better suited for them. Because we do not have the design specifications explicitly provided, we will need to make a list of design specifications based on Martin's description of functions from his thesis. During the verification process, we will monitor the effort spent on each of the methods and then compare them. We can't expect both methods to be effective or applicable for the verification of all specifications. Because of that, we will only use each of them to verify those specifications that are compatible with them. The final goal of this work is to say if the design is behaving correctly and obeying all of the specifications, or if the design has any flaws that would cause it to behave in a way that would violate the specifications. If we find any cases where design behavior causes a violation of the specification, we will record and report them to the design team for corrections after the completion of this thesis.

Chapter 1

Methods

The design verification process has multiple stages, and the whole process of verification is coordinated by the verification plan. The verification plan is the first document created when starting the verification process and is gradually updated during the verification. It has a list of design specifications that need to be verified, together with all of the information for verification. The information contains, besides other data, which of the verification methods are to be used for the verification of each specification.

1.1 Functional verification

Functional verification uses simulation of DUT behavior in the time domain. It is done in order to check if the DUT behavior is in compliance with its specifications. To be able to simulate DUT behavior, we also need to generate a sequence of logical values to be applied to the inputs of the design. This sequence of signal logical values we call stimuli [10]. We also usually need to read some of the design responses to be able to generate stimuli correctly and for the stimuli to be reasonable. The interface is used to send stimuli to the DUT and also to read responses. We call the block, responsible for being the top-level module in our simulation and containing DUT together with all other modules used in the simulation, the testbench. The simple block diagram of how the simulation environment works is shown in Figure 1.2. The configuration of the initial state (DUT internal state and testbench configuration), together with stimuli constrained to follow some scenario, is called the test.

Methods similar to those used in formal verification (mentioned in section

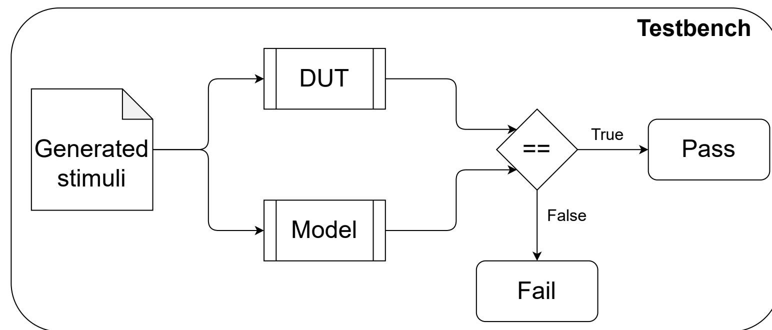


Figure 1.1: Simplified block diagram of functional verification environment.

1.2), can be used in parts of the DUT to assure that certain signals always fulfill certain conditions or that used protocols are not violated. These methods mainly use *assertion* command, which acts as a watchdog over the properties of a signal or group of signals. The validity of requested properties is evaluated and checked dynamically during simulation. Because of dynamic evaluation, there is a chance that the state of DUT that violates assertions is not reached during simulations but is reachable only when a specific sequence of input signals is used.

1.1.1 Simulation output

The simulator output, in its raw form, is the value of signals for each simulation time step (we will call them traces). The traces are useful for finding bugs when we already know that there are bugs, but it is time-inefficient to check for the existence of bugs when we have results from multiple test runs. The traces are also very storage-inefficient. Because of the need to store the values of desired signals for each simulation time step (even when logging only changes), it uses relatively large amounts of space to store all that data. To save data storage space, tests are designed to create log output files containing reports of signal behavior in a format that is easy to read by person, and contains only information relevant for the verification engineer. For the added time efficiency of the verification engineer, we have the self-checking tests (mentioned below) to simplify the analysis process of results generated by batch runs.

1.1.2 Self-checking tests

When simulations are used for verification, there is still a need to check if the data generated by the DUT is in compliance with the specification. Manually checking all of the data requires a lot of time and effort from the verification engineer.
The solution to this is the principle of self-checking tests. These tests create a single output that contains the results of the test [6]. This can be either "pass" when DUT behaves according to the specification or "fail" when DUT violates the specification. For self-checking to work, we need to know what behavior we expect from DUT. The expected output can be a waveform created prior to the start of the simulation or, in most cases, an output generated by a model of DUT that is fed the same inputs as DUT and usually runs in parallel.
Models are usually written on a higher abstraction level than DUT, which allows better transparency of their functions and makes checking against specifications easier. They can generate expected states of output signals, but in most cases, they operate on a transaction level of abstraction, which can transfer data as a set of values instead of relying on correctly encoding them to the DUT interface and then correctly reading and decoding them (from the DUT input interface). This allows them to predict expected behavior without dependence on the used communication protocol or propagation delay of DUT. Conversion between transaction and signal values needs to be done. In Universal Verification Methodology (UVM) (mentioned below), this is provided by the component driver and monitor.

1.1.3 Constrained random stimuli

For effective simulation, we need to generate randomized stimuli to get as much information from our tests as possible. The random stimuli are used to give each test some flexibility and decrease the number of individual tests that need to be written. A test with randomization can be used multiple times to produce different scenarios that are simulated.
A good example of a situation where randomization is very useful is the verification of the read operation of a memory module. In this scenario, without the use of randomization, we would need to specify every address and memory contents to be checked. The randomization allows us to check the read from any sequence of addresses with any memory content (provided we have enough resources to simulate them all) to be checked.
The randomization seed, which is set at the beginning of the simulation, is used to provide means for the same sequence of stimuli to be run. When multiple simulations are run with the same seed and the same revision of code is used, then all of these simulations should proceed exactly the same.

Any change to the code can result in different simulation behaviors, even when using the same randomization seed. There should be no change in the run of the simulations when they are run in batch mode or single simulation mode with a graphical user interface (GUI). This fact is used to run multiple simulations in batch mode, and from those, select only the simulations that failed and use their seeds to run exactly the same simulation to get more information for debugging. With randomization alone, it can be hard to reach corner cases of generated stimuli sequences (from example above: reading from maximum address data, which contains only bits asserted to a logical value of 1). For that reason, constraints are used to limit the variation of stimuli when absolute randomness is not needed. Constraints can also be used to specify exactly the required stimuli so that the randomness is effectively removed. Corner cases and critical sequences can be verified by this approach.

1.1.4 Blackbox vs whitebox (change of perspective with assertions)

Blackbox is when we view the design from a perspective that only allows us to see the external behavior. When design is referred to as a graybox, it means that in addition to the external behavior, we can also observe some signals and states that are inside the design. The most transparent is the whitebox, which allows us to see every internal state and signal of the design. Use of assertions can change the perception of verified DUT from black box to gray box [10]. When using the assertions, we can more accurately find the errors in the design and also detect errors before they can be seen as the wrong state of the interface. Assertions are not synthesizable code, because of that, they need to be wrapped in a synthesizer directive that disables the synthesis of code. Also, because they are not propagated to the final design, it is important to use them only as a sink of signals from the circuit design. Blocks of code containing assertions should not create any signals that are used in the function of the circuit.

1.1.5 UVM [5]

The industry standard used for functional verification of digital circuit designs is UVM [5]. It is a library written in SystemVerilog that contains basic structures and classes that can be universally used in the process of building a verification testbench. Its idea is to create a verification environment that is as independent of the actual DUT as possible. Also to provide an environment

where it is easy to create new stimuli for testing DUT. These qualities give the final verification environment high re-usability and flexibility, which reduces demands on future efforts for the expansion of tests for additional design requirements and increasing test coverage. UVM is a class-based system that relies on the inheritance of properties and methods of the parent class. Details of individual classes are described below. Communication between individual components is provided via transactions, which allow for easy creation of even complicated stimuli.

Macros for writing into log output file

For easier logging of simulation events, UVM provides its own macros that allow users to easily create log entries specifying what component created an entry, the time of the simulation, and the severity of the entry. Specified severity levels are *info*, *warning*, *error* and *fatal*. *Error* usually causes the test to fail but allows for the continuation of the simulation. *Fatal* ends the simulation and results in a fail of the test. *Info* severity also has a verbosity setting, which allows the user to set a minimal level of verbosity as an argument for the simulator. This can be used to create a log file with more or less information without the need to compile the whole design again. This function is especially useful for creating log files of simulations run in batch mode that contain only basic information to reduce unnecessary usage of memory space. Later, when some tests need to be reviewed, these macros allow for an increase in the depth of information provided in the log file without the need to change code or recompile the whole project.

Components

The UVM verification structure is divided into components. Each component has its own defined purpose and base class that contains most of the required operations and is provided by the library, which it extends. UVM handles communication protocols between individual components so that the verification engineer only needs to specify the content of the transaction. UVM also specifies the architecture that shall be used. These features allow faster, clearer, and easier creation of verification environments. With all its basic protocols and well-defined structure, it also provides good compatibility when used for different DUTs, as well as a uniform structure and interface for multiple verification teams. The UVM testbench is built from the following components:

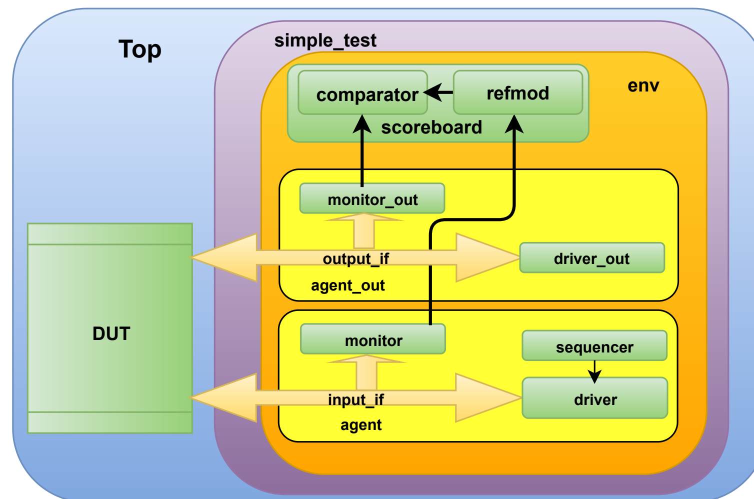


Figure 1.2: Block diagram of UVM environment [2].

- **Tb_top:**
  Container that creates instances of DUT, interface, and UVM test environment and connects DUT with the UVM environment via the created interface.

- **Test:**
  Defines conditions, configurations, and sequences of inputs used for simulation. This component has multiple variations in each project. Each variation is used to set and run different configurations and sequences to verify some parts of the required design specifications.

- **Environment:**
  Contains verification components that are reusable for all tests. Can contain multiple agents and scoreboards. Creates connections between scoreboards and individual agents to allow communication.

- **Scoreboard:**
  Usually contains a behavioral model of DUT that is used to predict expected outputs of DUT. This model is given the same inputs as those sent to the interface, but in the format of a transaction, even for serial communication, to simplify the block implementation. The output of the model is then compared with the response from the DUT, and if some unexpected difference is detected, a warning or error is recorded in the log. An example of a situation where a difference is expected is when DUT should present a random number (RN) to the interface. In this case, the predicted response can be corrected with the actually received RN before comparing.

- **Agent:**
  Acts as a wrapper for driver, monitor, and sequencer. Can be set as active or passive. An active agent creates instances for all three components.

A passive agent creates an instance only for monitor. Because of that, it can't send stimuli to the interface but only monitor its status and send messages to the scoreboard.

- **Sequencer:**
  Sends sequences set in the test component to the driver component.

- **Driver:**
  Requests a sequence item from the sequencer and transfers it to the interface. The sequence item is a transaction, so it can't be directly put into the interface. The driver implements functions for translating data stored in sequence items into signals for the interface. Those signals must be applicable to the physical interface for the possibility that hardware acceleration, using a field-programmable gate array (FPGA) board to run the design, is used.

- **Monitor:**
  Collects information from the interface and sends it to the scoreboard in transaction format.

Phases

The run of a UVM-based simulation is separated into sections called phases. They have a fixed order, and the next phase can start only after all UVM components have completed all their objectives for the currently executed phase.
All of the phases, except the run phase, are only allowed to use method calls that don't consume simulation time (must never allow simulation time to progress). For this reason, they use only *function* method calls defined in SystemVerilog that have the limitation of not allowing code that could result in the progression of simulation time. For methods that progress simulation time, a call with the type of *task* is called. *Tasks* should be only called during the run phase.

- Build phase
  All instances of classes that create the verification environment and instance of DUT and interfaces are created during this phase.

- Connect phase
  During this phase, all of the instances are connected to enable communication between individual instances and provide access to shared structures where it is needed.

- End of elaboration phase
  Final adjustments to connections are done during this phase, as well as checking if all needed connections exist.

- Start of simulation phase
  The final topology of the testbench is printed and written into a log file.

- Run phase
  Phase where the simulation in the time domain of DUT and testbench is performed. Can be divided into sub-phases where each sub-phase has pre_ and post_ additional parts that are executed before and after the sub-phase, respectively.

  - Reset
    DUT is put into reset state, which should be fully defined.
  - Configure
    DUT is configured using either standard access via interface or backdoor access directly to the memory.
  - Main
    The central part of the simulation is where stimuli are sent to the DUT and responses are checked.
  - Shutdown
    Wait for all stimuli from the *Main* sub-phase to propagate through DUT to ensure that no response is overlooked.

- Extract phase
  Phase where retrieval of statuses from scoreboards and functional coverage monitors is performed.

- Check phase
  Final check of DUT behavior during simulation.

- Report phase
  Report results of the performed simulation.

- Final phase
  Last phase for ending the simulation.

Simulator support

Most of the professional simulation programs today have integrated support functions for interacting with UVM components. An example of these support functions is the option to run simulation until the whole environment is built and then stop. With this function, it is possible to view the UVM structure and variables before the start of simulation in the time domain or to set

appropriate probes to record the time waveform of desired signals. However, these functions are mainly used for debugging already-discovered errors because simulations are mainly run in batch mode (multiple simulations are started and processed in series or in parallel) that don't have a GUI. Results of batch-run simulations are stored in log files that contain information about stimuli applied to the DUT, the response from the DUT, and the result (pass or fail) of the test with seed for replicating the run. The failing tests are then run with a GUI, which is used to identify why the test failed.

1.2 Formal verification

Formal verification is a method of proving intended circuit functionality using mathematical reasoning without simulation [3]. The evaluation of design is exhaustive, meaning that all states of design are checked.
The positive of the formal verification approach is that all of the design bugs caused by corner-case states of the circuit are found. This allows for better confidence in a verified circuit design. When formal verification finds a violation of the design's intended behavior, it provides a counter example (CEX), which shows stimuli causing the violation. This can be very useful for decreasing the time required to find which part of the design must be repaired. The formal methods can be categorized into two classes, which are *formal equivalence checking (FEC)* and *formal property verification (FPV)*. The graphs showing the ratio of adoption of format techniques can be seen in Figures 1.3 and 1.4. From these graphs, we can see that the ratio of adoption of formal methods is increasing. We can conclude that the adoption ratio is increasing with time.

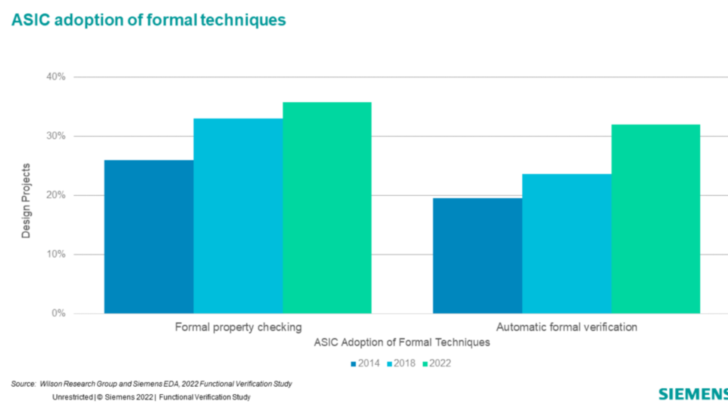


Figure 1.3: Adoption of formal techniques in different years [4].

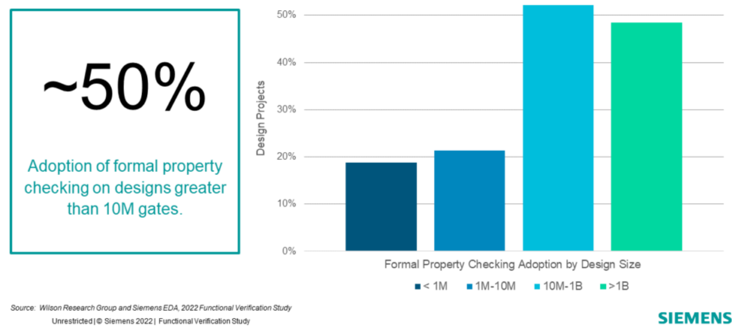


Figure 1.4: Adoption of formal techniques for different design sizes [4].

1.2.1 Formal equivalence checking (FEC)

Today, this method is mostly used in the form of checking that the design has not changed in between individual steps of implementation, from hardware description language (HDL) to masks used for fabrication. This is done to ensure that the intended design behavior is correctly translated to the next step (*e.g.:* check that the RTL design is correctly translated to the netlist) and that the behavior of both is equal. This method uses two input designs, which are the golden design and the tested design (continuing from the previous example, the golden design would be an RTL design and the tested design would be a generated netlist) to ensure their equality. FEC is used mostly as a precaution against bugs caused by compiler tools and to check that the final design for manufacturing has all parts placed and routed.

1.2.2 Formal property verification (FPV)

To prove the correct behavior of the HDL design against specifications, a static evaluation of properties is used. This means that a circuit is evaluated as a mathematical construct using mostly boolean algebra. FPV doesn't use stimuli to check design behavior. Instead, the stimuli can be generated for failing assertions, to show what stimuli cause the failure, or for reachable covers (then the stimuli describe how to stimulate the design for the cover to pass). Assumptions can be used to restrict the behavior of stimuli (e.g.: specify that the block uses a handshake protocol for communication on certain ports). Violations in the behavior of the design can be detected before output ports are affected, and the cause of the design bug can be traced more precisely. Design is viewed as a whitebox for verification tools to evaluate

and check internal behavior and detect bugs. The whitebox perspective does not cause the previously mentioned problem with the volume of stored data because the traces are relatively short. A similar approach can be used in functional verification (as mentioned above), where all assertions are evaluated dynamically instead of statically. This is called assertion-based verification (ABV) and can be seen as an intermediate step between functional and formal verification.

Limitation of formal assertions

The assertions used for formal verification need to be efficient for formal tools to verify. This places limitations mainly on the sequential length and width of the signals used. The sequential length prevents us from creating sequences that have the possibility to fail for traces of infinite length. The width of signals used in the property that we are asserting is mostly applied to the wide address or data buses. In a lot of situations, this can be resolved by creating an assertion for each bit of the bus. An example situation is the use of FPV to check that blocks do not change data when they are passing through it. In this situation, we can write (generate) properties, which each check the equivalence of the single bit of input and output data bus, instead of properties, which only check for the equivalence of the buses. The separation to single bits is more complex for implementation, so it is done to optimize the code when the formal engines are having trouble resolving the assertion.

1.2.3 Complexity

Today, the FPV is not as widely spread as functional verification because of its complexity, and resource requirements are exponentially dependent on design complexity, which is exponentially dependent on time (following Moore's law). Additionally, the abstraction level used to create designs is too low to efficiently deploy formal verification methods. Because of these reasons, formal verification can't be effectively used to verify the completed design as a whole. Instead, it is a very powerful tool for verifying smaller parts of complex systems because of its ability to uncover corner-case bugs in design, which can be almost impossible to find by using functional verification.
Exponential dependence on design complexity implies that the best cases for using FPV are combinational circuits and circuits that use short serial protocols. Examples of these circuits are buffers, encoders, handshake protocols, and interrupt controllers for processors.

1.2.4 Languages

The problem with formal verification is that there are lots of languages that can be used. The language used by the verification tool is defined by the vendor, and this may create problems with compatibility when using tools from multiple vendors or when a transfer to the tool from a different vendor is needed. Some of the most commonly known languages are *OpenVera Assertions* (OVA), *Open Verification Library* (OVL), *Property Specification Language* (PSL), *e* , and *Sugar* [11].
In 2002, the *Sugar* language was selected as an industry standard by IBM, but other languages continued to develop, and today there are a lot of languages that are used. Also, design languages such as VHDL and Verilog have implemented their own systems of assertions that can be easily used by design developers for specifying intentional behavior design.
This work will use the tool JasperGold from the company *Cadence Design Systems, Inc.* which was first introduced in 2003. This tool supports the languages SVA and PSL for verification and Verilog with VHDL as design languages, respectively.

1.3 Coverage

Coverage is a vital part of design verification. It tells us what behavior of the design we have verified [9]. Without measuring the coverage of the design, we wouldn't know when to stop verification. In an ideal situation, we would check if all possible reachable states were not violating the specification. The exercise of all possible states of design and checking compliance with specification is unrealistic due to the exponential increase of states dependent on the size of the design. The coverage can help us decrease the required verification effort by presenting a certain level of abstraction to the statespace that we are verifying.
For example, when we have a group of signals forming a bus, we do not usually need to check all possible values that can be present. Usually, we need to know that a bus can obtain certain values and that nothing bad should happen when a bus gets to state values outside of the expected range. For this, we can use a small number of covers, where each cover can represent a range of values or a corner case.
Without coverage, we could say that DUT did not have any bugs before we exercised all specified behaviors or all important states. Also, there could be hard-to-reach states that we could miss without covers.

Representation in Functional Environment

When simulating the design, the coverage is measured as the number of times that design reached the state specified by the coverpoint. The check for an increase in count for a certain coverpoint is not performed in each step of the simulation. The check can be triggered manually. (eg.: when a new transaction is to be driven to the input signals, we can sample it to count how many times we sent a message of each type.) or automatically by binding the sample trigger to some signal inside the design or auxiliary signal that we got as a combination of design signals. An auxiliary signal is not synthesized and thus can't be used by design as an internal signal.

Representation by formal tools

The formal tools, namely JasperGold from Cadence Design Systems Inc., represent the covers as states of the design that we would wish to observe. JasperGold produces waveforms that show how the state, defined by cover, can be reached from the reset state of the DUT. This can be used to check if our design is capable of reaching important states or if some corner cases can be reached. This is good for us because we can check the quickest way to reach a given state without thinking of the inputs needed to be applied. There are also some covers that can be automatically created by the tool to check code reachability, signal toggles, reachable finite state machine (FSM) states,...

Chapter 2

Verified design (RISE)

The design that we will verify using the mentioned methods is the branch predictor for the RISC-V processor by Martin Laštovka named RISE. [7] This block is designed to predict branches based on observation of the previous sequence of addresses requested by the processor. The intended use of this block is to be placed on the Open Bus Interface (OBI) between instruction memory and processor in a way that cuts previous direct connection and connects to the both ends independently. Physically, it should be placed as close to the instruction memory as possible, because the main purpose of this design is to shorten the combinational path of the address request to the memory. A shorter critical combinational path should allow us to increase effective processor performance by allowing a higher clock frequency. The higher clock frequency should outweigh the cycles in which the processor is stalling, caused by our wrong prediction. This block should be able to correctly predict short branching patterns and quickly recover from wrong predictions.

The interfaces that are connected to the processor and memory use OBIv1.4 [1] as a communication protocol.

2.1 Prediction method

Design does not implement decoding of prefetched instructions to obtain information about instruction type or content. This allows us to predict

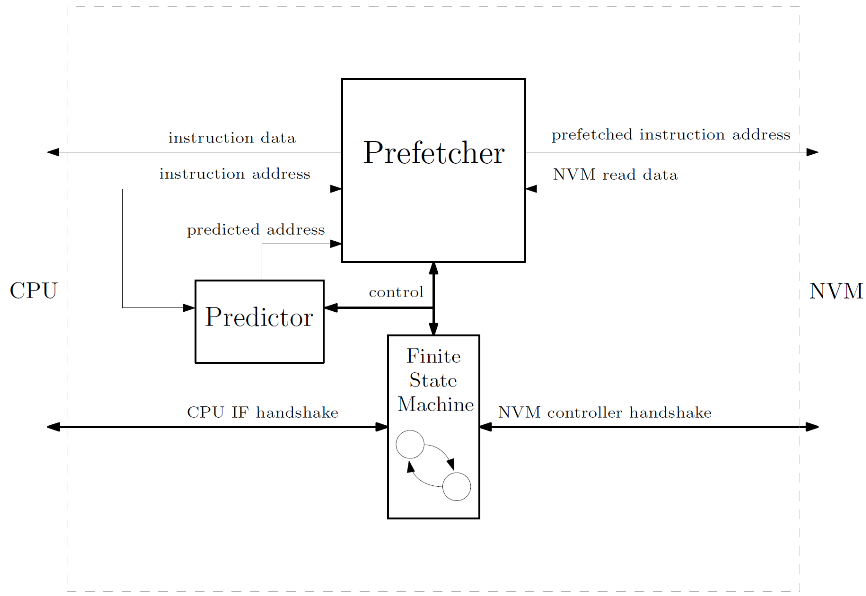


Figure 2.1: Block diagram of RISE by Martin Laštovka. [7]

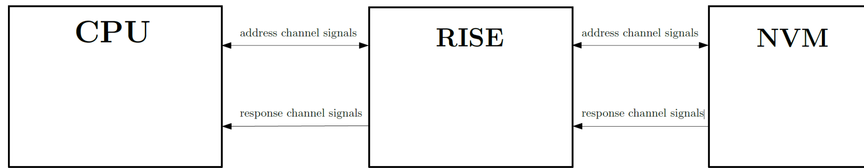


Figure 2.2: Block diagram of RISE integration by Martin Laštovka. [7]

independently on the instruction coding used by the processor. The design predicts based on previous recorded branching (the record contains the source of the branch and its destination). The global history of branching results is also used for prediction. The matrix of 4-state FSM predictors is stored in random access memory (RAM), where each line contains 2^M predictors (viz.: Table 2.2), which are addressed by the least significant bits (LSBs) of address. The active predictor from the line is selected based on the current value of branching history. The sets, containing recorded branching sources and destinations, are designed as associative memory and can be expressed as a matrix that has row indexes equal to set index, column indexes equal to LSBs, and each cell contains active state, ID bits of address, and last bits of destination address. The predictor table is only used when we encounter an address that has the same ID as some set of corresponding positions inside associative memory. The predictor table is also updated only when we encounter the address that we can declare as active by the data stored inside the associative memory. The branch is registered only when the most significant bits (MSBs) that are not recorded as LSBs of the destination address are the same for the source and destination addresses. In the opposite situation, the registration of the branch would not have any meaning because there would be no possible way to reconstruct the destination address from

signal name	direction	use
clk_in	IN	clock signal common to processor and memory
rst_n_in	IN	rst signal that is active in low
rise_en	IN	enable signal
instr_req_in	IN	OBI for processor
instr_gnt_out	OUT	OBI for processor
instr_rvalid_out	OUT	OBI for processor
instr_addr_in	IN	OBI for processor
instr_rdata_out	OUT	OBI for processor
flash_gnt_in	IN	OBI for memory
flash_rvalid_in	IN	OBI for memory
flash_req_out	OUT	OBI for memory
pref_instr_addr_out	OUT	OBI for memory
pref_instr_rdata_in	IN	OBI for memory

Table 2.1: IO signals of RISE

Parameter	label	Value
Address bit length	-	17
Tag index width	-	2
Set index width (addr LSBs)	m	5
Global branching history	M	0
Number of sets	S	1
Number of lines in each set	N	2^m
Memory budget	B	S*N
Clipped destination addr width	c_d	10
Number of predictor states	-	4
Preciction type	-	One level

Table 2.2: Values of design parameters used for verification.

available information.

Chapter 3

Verification plan

A verification plan is an important part of the verification process that tells us what needs to be verified and in what order. It defines steps that need to be taken for the design to successfully pass verification process [8]. It also specifies the verification approach for proving that the design does not violate specifications and all desired functions of the design are present. Some specifications can be easily verified by formal methods, and some by simulation. There can also be specifications that are so complex that simulation and formal are both practically unusable due to the time required to run formal tools and simulation. In that case, we can try to verify those specifications using hardware acceleration such as an FPGA, an emulator, or a manufactured prototype.

3.1 Design requirements

The requirements for our design are derived from the specification of the OBI protocol and the intended use of our design.

- OBI protocol requirements [1]:
  - **OBI-R-1** [1] Signals *clk* and *reset_n* are common between the master and slave.

[1]This OBI requirement is not to be verified by used verification methods

  - **OBI-R-2** Signals *req* and *rvalid* shall be driven low during reset.
  - **OBI-R-3** Address channel A shall use a two-way control handshake (signals *req* and *gnt*).
  - **OBI-R-4** [2] Response channel R shall use a two-way control handshake (signals *rvalid* and *rready*).
  - **OBI-R-5** The response phase shall not start before the address phase is finished. Signal *rvalid* shall not be asserted before previous observed assertion of *req* and *gnt* in the same cycle.
  - **OBI-R-6** Response transfers shall be sent in the same order as their corresponding address transfers.
  - **OBI-R-19** [1] OBI link outputs (excluding *gnt*) shall not combinatorially depend on OBI link inputs.
  - **OBI-R-20** [3] Signal *gnt* shall not combinatorially depend on OBI link inputs.
  - **OBI-R-22** [1] OBI link outputs of any master interface shall not combinatorially depend on OBI link inputs of any other master interface.
  - **OBI-R-23** [1] The OBI link outputs of any slave interface shall not combinatorially depend on the OBI link inputs of any other slave interface.
  - **OBI-R-24** [1] A transaction's *req* shall not depend on the *gnt* for that transaction.
  - **OBI-R-26** Unused pins of the OBI interface shall be tied off as shown in Table A.1 unless specified otherwise.

- Design specification requirements:

  - **SPEC-R-1** When signal *en* is deasserted, the design shall act only as a passthrough for OBI signals with minimal delay.
  - **SPEC-R-2** When the *rst_n* signal is deaserted, RISE shall set all internal registers to their default values.
  - **SPEC-R-3** Pins assigned to the OBI interface communicating with the processor shall obey OBI protocol specifications as a slave side of the connection.
  - **SPEC-R-4** Pins assigned to the OBI interface communicating with memory shall obey OBI protocol specifications as a master side of the connection.
  - **SPEC-R-5** Both OBI interfaces shall support the following operations:

[2] Due to the signal *rready* is not used and is thus assumed to have a tie-off value. The length of response is only one cycle.

[3] Constant *COMB_GNT* is not defined in design. The default value of *False* is assumed.

  - **SPEC-R-5-1** Master reads word[4], whose position in memory is defined by *addr* , from slave.
- **SPEC-R-6** The *req_out* shall be asserted if the *req_in* is asserted.
- **SPEC-R-7** The *req_out* shall be deasserted if it is not required[5] to be asserted.
- **SPEC-R-8** The *gnt_in* shall not be assumed to be always asserted.
- **SPEC-R-9** The *gnt_out* shall be deasserted if *gnt_in* is deasserted.
- **SPEC-R-10** The *rvalid_in* shall not be assumed to be always asserted.
- **SPEC-R-11** The *rvalid_out* shall be deasserted if *rvalid_in* is deasserted.
- **SPEC-R-12** When the prediction is correct. The RISE shall request instruction from memory on the same address as is received from the processor in the same cycle.
- **SPEC-R-13** When prediction is wrong (the *addr_in* is not the same as *addr_out*). The following shall happen:
  - **SPEC-R-13-1** 0 cycles after: The *gnt_out* shall be deasserted.
  - **SPEC-R-13-2** 1 cycle after: The value of the *addr_out* shall be set to the same value as the *addr_in* .
  - **SPEC-R-13-3** 1 cycle after: Signal *gnt_out* shall be asserted if and only if (iff) *gnt_in* is also asserted in the same cycle. If *gnt_in* is not asserted, RISE shall wait in the current state until *gnt_in* is asserted.
  - **SPEC-R-13-4** 1 cycle after: Signal *rvalid* shall be deasserted.
  - **SPEC-R-13-5** 2 cycles after: Signal *rvalid_out* shall be asserted iff *rvalid_in* is also asserted in the same cycle. If *rvalid_in* is not asserted, RISE shall wait in the current state until *rvalid_in* is asserted.
- **SPEC-R-14** Every address requested by the processor shall be requested from memory.
- **SPEC-R-15** The processor shall receive only instructions from addresses that it has requested.
  - **SPEC-R-15-1** When a wrong prediction occurs, *rvalid* shall remain deasserted until instruction from the correct address is presented.
  - **SPEC-R-15-2** The deassertion of *rvalid* , due to wrong prediction, shall be no longer than one cycle.
- **SPEC-R-16** Recovery from a missed prediction shall not take longer than one clock cycle.

[4] Word is, in this case, 32 bits wide

[5] Sources that can require *req_out* to be asserted are asserted *req_in* and specification of OBI protocol.

- **SPEC-R-17** Every address sent to the memory interface shall be aligned to a multiple of 4 (two LSBs are 2'b00).
- **SPEC-R-18** The processor shall receive instructions in the same order as it requested their addresses.
- **SPEC-R-19** RISE shall be able to register a new branch and record it in predictor memory.
- **SPEC-R-20** RISE shall be able to update an already-recorded branch with a new destination.
- **SPEC-R-21** RISE shall be able to decide when the branch should or should not be used based on branching history.
- **SPEC-R-22** RISE shall not modify the content of any fetched instruction.
- **SPEC-R-23** The prediction algorithm shall be compliant with the following requirements:
  - **SPEC-R-23-1** The branch history shall record whenever the branch was taken (T) or not taken (NT).
  - **SPEC-R-23-2** The branch history shall be global for all addresses.
  - **SPEC-R-23-3** The branch history shall be updated iff a new branch is detected or the last *addr_in* is already registered as a branching address.
  - **SPEC-R-23-4** The branch prediction shall be the output of a 4-state saturating FSM predictor that has states with transitions depending on the last branch result from the respective address:
    - ST - strongly taken (T -> ST, NT -> WT)
    - WT - weakly taken (T -> ST, NT -> WNT)
    - WNT - weakly not taken (T -> WT, NT -> SNT)
    - SNT - strongly not taken (T -> WNT, NT -> SNT)
  - **SPEC-R-23-5** The initial state of each 4-state predictor shall be WT.
  - **SPEC-R-23-6** The set of predictors shall consist of $2^{length_of_prediction_history}$ 4-state predictors.
  - **SPEC-R-23-7** The index of the set of predictors shall be N bits long. (We shall have 2^N sets of predictors.)
  - **SPEC-R-23-8** Each address shall access the set of predictors with an index equal to the [N+1:2] bits of the address. (The last two bits of address are omitted due to the requirement **SPEC-R-17**.)
  - **SPEC-R-23-9** The selected predictor from the set of predictors shall correspond with the current state of branch history.
  - **SPEC-R-23-10** Each 4-state predictor shall have its state updated iff:

  - The predictor was used for prediction in the last cycle.
- **SPEC-R-23-11** The identifier (ID) of the branching address shall be of length K.
- **SPEC-R-23-12** When a new branching address is encountered, its ID (address bits [K+N+1:N+2]) and destination (bits [D+1:2] of the destination address) shall be recorded and set as active.
- **SPEC-R-23-13** The branching addresses shall be stored in associative memory, which assigns each ID and its destination bits to the lower bits ([N+1:2]) of the source address.
- **SPEC-R-23-14** There shall be a maximum number of L active branching addresses, that have the same bits [N+1:2].
- **SPEC-R-23-15** When a new branching address that has the same bits [N+1:2] as L of already active branching addresses is encountered, it shall replace the oldest recorded branching address, which shall be deactivated/removed.
- **SPEC-R-23-16** Update of associative memory shall have a higher priority than reading currently stored data. When two branching addresses are read consecutively, the prediction for the second address is skipped due to the priority to update associative data for the first address. The second address shall be treated as if it were not registered in the associative memory.
- **SPEC-R-23-17** When the prediction result is that branch is not taken, the next *addr_out* shall be the current *addr_out* + 4 .
- **SPEC-R-23-18** When the prediction result is that branch is taken, the next *addr_out* shall be constructed by following pattern:
  - [1 : 0] <= 2'b00
  - [D+1 : 2] <= recorded bits of destination address
  - others <= bits on same position as the *addr_in*

3.2 Design verification

The first step is to verify assertions written by the designer using JasperGold, because the low level of assertions should make it easy for formal engines to verify them and are a lot quicker to start than simulation (no additional verification environment is needed). From this observation, we can create basic assumptions if they are needed for the designer's assertions to pass. Assumptions need to be added carefully to not restrict behavior so much that we could ignore valid interface behavior. Then we will proceed with verification in the following order of steps:

1. Create basic sequences in SVA.
2. Write basic covers in SVA.
3. Create a universal verification component (UVC) that has assertions for the correct use of the OBI.
4. Bind this UVC to the two OBI interfaces that our design implements.
5. Create assertions that check design behavior.
6. Carefully add more assumptions if they are needed.
7. Complete formal verification.
8. Make UVM environment.
9. Create a model of design.
10. Write UVM tests and simulate.

3.2.1 Expected behavioral scenarios for verification

Because we already have specified design use, we can assume certain sequences of the *addr_in* , that need to be verified. These scenarios can be verified only after we have already verified the compliance of design interfaces with the specification standard. To exercise the design in expected patterns, the following scenarios are identified:

- **SCEN-1** Linear sequence of addresses
- **SCEN-2** Branch of address surrounded by linear reads
- **SCEN-3** Branch from addr A to random address, then linear read and jump back to address A+4 (simulate entering subprogram or processor interrupt)
- **SCEN-4** Branch to a random address, then loop back to the same address as we previously branched to. After N repetitions of the loop, do N repetitions of the loop, starting from the address sequentially following the address of the end of the previous loop. After the loop ends, branch to the address following the address from which we branched randomly. (Simulate recursive function calls.)
- **SCEN-5** Nested loops simulating nested "for cycles" in a program.

- **SCEN-6** A single address is requested over and over again. (Simulate a stall after the main embedded program ends and the processor is waiting for interrupts.)
- **SCEN-7** Linear read that overflows the maximum value of address space
- **SCEN-8** The address is selected randomly each cycle.
- **SCEN-9** Combinations of previously mentioned scenarios.

3.2.2 Formal methods (JasperGold)

First, we do reset and clock analysis to check that all important signals have defined reset values and ports used for OBI are compliant with the OBI specification's required reset values and active clock edges. Then we will write a sanity cover that will tell us if our design is able to perform some elementary operations. This cover should be as simple as possible to minimize the chance of error. For this reason, we chose to cover when output data is valid for three values checked independently as separate covers. The checked values are 0, the *maximum_value_of_data*, and the *random_nonzero_value*. After we check that the design is capable of elemental operations, we create basic sequences of stimulating design so that writing more complex covers is easier. Those sequences shall be parametrized to increase reusability. Requested sequences are:

- Read operation from processor
- Response from NVM
- Simulate program without branching
- Simulate branch
- Simulate loop

After that, we will write covers to observe the simple and more complex behavior of the design. The covers should show behavior when:

- Address is branching randomly (satisfy **SCEN-8**)
- Address is constant (satisfy **SCEN-6**)

- Address starts at 0 and is linear (increments by 4) for n cycles (satisfy **SCEN-1**)
- Address is linear and looping with a loop length of 3 (satisfy **SCEN-2**)
- Address is linear and looping with a loop length of 7 (satisfy **SCEN-2**)
- Address is linear with two loops that do not intersect
- Address is linear with two nested loops (internal loop breaks after 4 repetitions) (satisfy **SCEN-5**)
- Address is linear and overflows (satisfy **SCEN-7**)
- En is deasserted, then show 10 cycles of signals passing through without delay (satisfy **SPEC-R-1**)
- Different address can be requested each cycle without data validity falling

The control signal *en* is asserted for each case unless explicitly mentioned otherwise. When all covers are checked and are reachable, then we can continue with the writing of verification assertions. The first assertions, that we make are to check the correct implementation of the OBI protocol. This implicates creating a verification component in SystemVerilog that binds to the interface ports and contains two sets of assertions (master and slave side), checking correct interface behavior with respect to the specification [1]. We can expect protocol violations caused by a known bug in the targeted processor, also mentioned in Table A.2, and acknowledged at the end of the OBI specification. For the next assertions, derived from the design's intended functionality, we want to check the following:

- When *en* is deasserted, design mirrors signals inputs from one interface to the outputs of the other interface (satisfy **SPEC-R-1**)
- Each address (that is, a multiple of 4) can be requested on the memory interface
- Processor only receives instruction after it requests it (satisfy **OBI-R-5**)
- Data from memory is the same as data presented to the processor (satisfy **SPEC-R-22**)
- Only instruction from the address requested by the processor is marked as valid (satisfy **SPEC-R-15** and **OBI-R-6**)
- Data arrive to the processor in the same order as their addresses are requested (satisfy **SPEC-R-18** and **OBI-R-6**)

- Every address requested by the processor is eventually requested from memory (satisfy **SPEC-R-14**)
- Request to memory is only after previous request from processor (multiple requests to memory can be done as response to one request from processor) (satisfy **SPEC-R-7**)

3.2.3 Functional (UVM)

For functional verification via simulation, we will create a testbench using the UVM environment and guidelines. Our testbench environment will contain all standard components of UVM, as shown in 1.2. The block diagram of the implemented environment is in Figure 4.2.

We will use three separate agents to stimulate and monitor the DUT. The first agent is assigned to the generation of clock, reset, and enable signals. The second is to model memory that can be modeled without randomization because we have already formally verified that this interface is behaving in compliance with the OBI specification. The third generates stimuli on the processor interface and contains the main driver. All sequences that we will write for testing will be bound to the third driver. The handshake protocol used in OBI will be implemented in drivers. We will also need to create a model of our DUT that shall behave in compliance with the design specification. The model will operate at the transaction level of abstraction.

Sequences

The sequences contain behavior patterns that are used in tests. Because we already implemented the handshake protocol in drivers, we only need to create a sequence of addresses that will be sent as requests to the DUT. All addresses in sequences need to be divisible by 4, because we only read whole, 4 byte-long words. The sequences of address reads that we will need are:

- Random sequence
- Non-branching (linear) sequence with variable length
- Random sequence that is repeated with variable loop count
- Loop that can contain other sequences

Tests

The tests will all be derived from the basic test class, which will define configuration and provide access to sequences. The purpose of these tests is to verify the correct implementation of the prediction algorithm. We don't need to verify requirements that are already proven by formal methods. The tests are mostly based on expected behavior patterns that the instruction addresses. The list of patterns that need to be tested is as follows:

- Sanity: linear from 0 to N and some loops
- Full random: N random addresses (N taken branches in success)
- Full random loop: previous repeated K times
- Cpu simulation
  - Linear with inserted loops (depth 1)
  - Linear with inserted loops (depth 2)
  - Linear with inserted loops (depth k)
  - Recursion
  - NxN matrix multiplication
  - Bubble sort

Chapter 4

Implementation

The implementation was done in the order mentioned in section 3.2. First, the assertions already implemented by the designer were run through formal verification engines. After that, the formal testbench was created and verified by formal engines. After the formal verification was concluded, the UVM testbench for simulation was created, and tests were run on it. For the purpose of this work, I implemented the UVM environment and the verification components, with the top module used for formal verification. For the simplification of commands used to run the verification tools, I also created scripts to pass the tool all of the arguments that are passed to the tool every time. The files imported from the work of Martin Laštovka [7] are all located in the attached file inside folders *src/design/* and *src/package/*. Additional assertions and coverage that I created for formal verification can be found in files *src/tb/tb_verif_top.sv* and *src/tb/vcomp/*.sv*. Those were used alongside the assertions created by the designer in the *src/design/** files for formal verification. The script that I created for compiling the design in JasperGold and setting up the tasks for verification of design functions can be found in the file *scripts/jg_run.tcl*.

4.1 Formal (Jasper Gold)

For the verification of assertions created by the designer, there was no need to create or change any code. Only JasperGold was used at first to compile the design and run the formal engines. The first iteration has been done without using assumptions in order to not limit the tool in any way. Then

the first version of the formal verification environment was implemented. It was written in SystemVerilog and at first functioned only as a wrapper of design with the same interface. After the correct connection between the formal testbench and design was checked, the sequences and covers were created inside the testbench. The verification component, which contains assertions for the correct behavior of OBI on both sides of communication, was created to allow easy reusability of these assertions. This component was then binded to each of the two OBIs that the design implements, once as a slave checker and once as a master checker. These modes were bound to the interface connecting to the processor and memory, respectively. In the final phase of formal verification, the assertions for the correct functional behavior of the design (from an external perspective) were made. Some of the assertions required assumptions to be placed on the design inputs, but none of the assumptions were too complicated, instead, they restricted the inputs by a reasonable amount to get closer to the OBI protocol specification. The list of requirements verified by the formal verification can be viewed in Table B.1. The block diagram of the implemented formal testbench is captured in Figure 4.1.

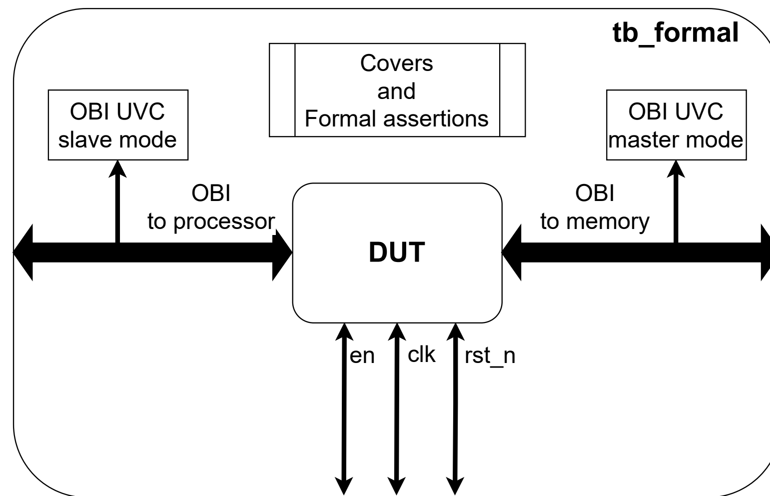


Figure 4.1: Block diagram of implemented formal testbench.

4.2 Functional (UVM)

The implemented simulation environment is based on the UVM [5] standard, as explained in [2]. The block diagram can be seen in Figure 4.2. The testbench contains three separate agents, which each control one interface. The interfaces are OBI connected to the driver acting as the processor (this is the main driver on which we run test sequences), OBI connected to the driver acting as memory with the lowest possible latency, and the final

driver is responsible for generating clock, reset, and enable signals, which are transferred via the control interface. Each driver has their own configuration class, which is created from global configuration and passed down to the agents as a reference to the object.

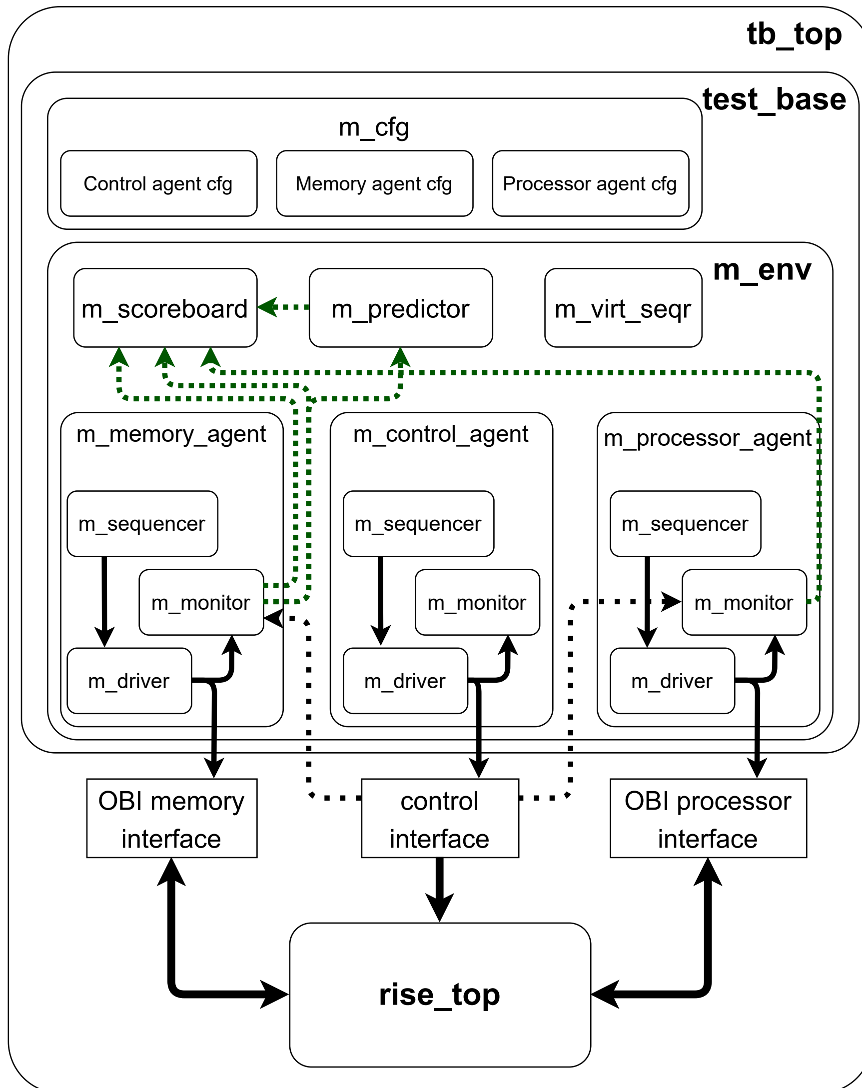


Figure 4.2: Block diagram of implemented UVM testbench.

4.2.1 tb_top

This component is the top module of the simulation testbench. It creates an instance of each interface. The DUT instance is also created here, and its ports are connected to the respective signals inside interfaces. The interfaces are also connected to the correct agents. The desired format of simulation time, to be used as the default when viewing the simulation waveform or the

log, is defined here. The test, passed as an argument to the simulator, is instantiated and run using the UVM method "run_test()".

4.2.2 test_base

This component is the template upon which all tests are extended, using it as the base class. It contains only two instances, one instance of the verification environment and one instance of the global configuration. The configuration instance is then registered in the UVM configuration database to be accessible from the environment. During run_phase the simulation drain time is set at 1μs to stop simulation at a reasonable time after the test sequence has ended. The function to set the default configuration is defined but is empty because all configurations are set during the creation of the configuration object. This function would be used in a more complex testbench to have all the needed configuration data in one place for easy access and debugging.

4.2.3 m_cfg

Main configuration object. Contains configuration data for assembling the test environment and also holds other configuration objects that are used by the agents. The reason for the centralization is to allow tests to easily access and change configurations to adjust the environment for that particular test.

4.2.4 m_env

This component creates instances for control_agent, memory_agent, processor_agent, rise_predictor, virtual_sequencer, and scoreboard. Each component is created iff it is enabled in the m_cfg that is received. The sequencers inside agents are connected to the central virtual_sequencer, and ports of monitors and predictor are connected to the scoreboard. The port on processor_driver is connected to the port on the predictor.

4.2.5 m_scoreboard

This component collects packets of data recorded by monitors and predicted by m_predictor and compares them. When a packet is received from the source, other than a slave response from the OBI processor interface, the data packet is cloned, and the clone is pushed to the back of the respective queue. When the packet contains the response from the OBI processor interface, then the comparison is done. The comparison is done in two stages. First, the packet with the request to the memory must have the same address and data as the predicted packet. These packets are popped in parallel, and the order and contents must always match. The rule of the first stage applies always, and wrong data or address is reported as an error. In the second stage, three outcomes can be observed:

1. Address and data are matched with the request, and correctly received packed is counted.

2. Address don't mach, then the comparing returns to the stage one and next memory and predictor packets are popped and compared. The predictor miss is recorded with severity 'info'.

3. Addresses match, but data doesn't match. The error is reported, but a new memory packet is not popped because the packet with the matching address has already been matched.

At the end of the simulation, the total number of received packets from individual ports is reported, along with statistics about comparison. If at any point we try to pop from an empty queue, the simulation ends with a severity of 'fatal'. When this occurs, there is already a problem in the simulation that renders all the following data pointless, and the time would only be wasted by continuing with the simulation.

4.2.6 m_virt_seqr

Virtual sequencer that provides easy access to the individual sequencers to simplify the writing of tests.

4.2.7 m_predictor

Predictor of requests to the memory. Contains models of associative and FSM predictor memories. The operation on data inside memories is implemented inside models. Data are read and written to the memories based on specifications. When a prediction miss is predicted, the address, which should be requested from memory by RISE as the result of the miss, is sent to the scoreboard before the address and data that correspond with the processor request.

4.2.8 m_control_agent

This agent's purpose is to drive control signals (*en* , *clk* , *rst_n*). These signals can be affected by sending sequences to the driver, which changes parameters according to the requested action in the transaction. The commands are provided by sequence_api which provides easy and clean access to required operations:

- Switch reset on/off
- Switch enable on/off
- Enable/disable clock
- Do a reset with the requested duration
- Do the power-up scenario, enable the clock, assert the enable signal, and do the reset pulse

The generated clock signal has a period and duty cycle defined in the corresponding configuration object. The clock generation process is run in parallel with the rest of the code, and before asserting a clock signal, it checks if the clock is enabled.

4.2.9 m_memory_agent

The memory agent is similar to the control agent in that all commands set the parameters of operation. The sequence_api provides methods for:

- Memory_start
- Memory_stop
- Memory_randomize
- Memory_set_gnt_delay_range
- Memory_set_data_delay_range
- Power_up

The delay ranges are set by default to their lowest values, gnt delay has a single value of 0 cycles, and the data delay has a single value of 1 cycle. The driver with these values acts as RAM, which grants requests in the same cycle and presents data in the next cycle. The power_up scenario randomizes the contents of the memory, sets these minimal delays, and starts the operation of the memory. The memory is in a continuous function state, which means that it can grant requests before data for previous requests has been presented. The driver has outputs synchronized to the *clk* signal from the control interface.

4.2.10 m_processor_agent

The processor agent is the main agent that drives the simulation sequences. The monitor in it has two ports, one for sending recorded requested addresses to the scoreboard and predictor, and one for sending recorded responses to the scoreboard. The sequence of addresses to be requested is controlled by sequence_api and the driver only receives addresses to be driven to the DUT and sends them as requests in compliance with the OBI protocol. The sequence_api for this agent is the most complex because it has to be able to provide easy access to the basic sequences of addresses that we can expect from the processor. The basic methods, which are meant to be used internally, of sequence_api are:

- Drive_addr
- Increment_addr
- Drive_addr_increment
- Set_addr
- Get_addr

The higher-level methods, which are meant to be used by tests, are:

- Linear_read
- Loop_plain
- Recursion_plain

These use constrained randomization to create different scenarios based on a seed that is set at the start of the simulation. This allows them to create many different sequences with minimal change that can be automated to provide a large number of different stimuli for simulation.

4.2.11 Tests

In this stage of verification, only three tests have been written so far. All of them extend the base class *test_base*, and each of them creates an instance of *test_<testname>_seq* sequence created for each test in the same file as the test class. The sequences are all extended from the *test_base_seq* sequence, which provides access to the sequence_apis for easy control of each agent. The configuration used by the tests is the default configuration without changes. During the run phase, the tests raise objection to disallow UVM from ending the simulation before the sequence has ended. Then it creates an instance of the appropriate test sequence and calls the *init* functions of each sequence_api to bind them to appropriate agent sequencers. After the apis are initialized, the test sequence is started, and after it ends, the objection is dropped to allow the simulation to end after the specified drain time mentioned above.

test_sanity

This test is designed to check if our environment runs correctly and the design is capable of the basic required behavior. The sequence for this test consists of:

1. Generating reset signal
2. Short linear read from address 0x0

3. Short looping sequence of addresses
4. Longer looping sequence of addresses
5. Setting address near address upper value limit and linear read to check address overflow behavior
6. Setting address to a random value and then reading a linear sequence of addresses
7. Requesting the same address multiple times to simulate the end of the embedded main program

test_loop

This test is designed to check the behavior of DUT when requesting an address sequence that is similar to the for loops without nesting them. The sequence for this test consists of:

1. Generating reset signal
2. Short linear read from address 0x0
3. A linear loop of addresses that have a relatively short length of one loop and are repeated relatively many times
4. Same sequence as before, but repeated five times from random addresses, from which the loop starts

test_recursion

This test is designed to check the behavior of DUT when it encounters a sequence of addresses, which can be expected when a program encounters a recursive function. The sequence for this test consists of:

1. Generating reset signal
2. Short linear read from address 0x0
3. Recursion sequence with a maximum depth of 5 from the current address

4. 4 recursion sequences with a maximum depth of 5 from random addresses
5. 5 recursion sequences with a maximum depth of 128 from random addresses

Chapter 5

Results

The first part of this chapter will focus on reporting the results of tests performed. After that, we will discuss the implications, possible causes, and severity of the problems found.

5.1 Results of tests

The first reported results are from an evaluation of assertions, written by the block designer, using the tool JasperGold. After that, we will summarize the results of the covers and assertions written for formal verification. The last part of the report will cover the results of the simulation tests that we created in our UVM environment.

5.1.1 Formal (Jasper Gold)

The results of formal verification were obtained by using the program JasperGold from Cadence Design Systems, Inc.

Result of designer's checks of code

For the evaluation of assertions written by the designer, we did not use any additional assumptions to limit the possible stimulation of the design. When evaluating the assertions placed inside the code by the designer, we found that all related covers are satisfied. Only two of the assertions produced a CEX. All other assertions passed.
The first assertion that did not pass is *asrt_table_read*, which appears to be failing when two consecutive read misses are detected. The signal *instr_req* does not need to be asserted during this mispredicted sequence of addresses. The shortest signal trace required for this assertion to fail can be seen in Figure 5.1.

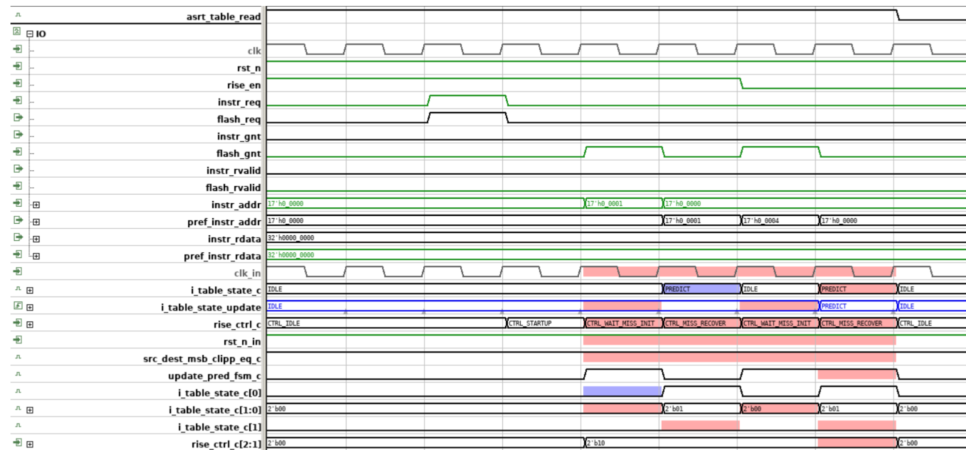


Figure 5.1: Trace of failing design assertion "*asrt_table_read*".

The second one was assertion *kill_sim_asrt*, which was needed to be rewritten inside *tb_verif_top.sv* because it was not inside synthesizable code of the design. For this reason, the formal verification tool did not compile it. The new assertion has exactly the same expression to evaluate and has the name "*ASRT_RISE_TOP_KILL_SIM_ASRT*". The trace for this failure is captured in fugure 5.2.

Result of created formal coverage

All of the created formal covers passed with the exception of one cover. The cover that the tool reported to be unreachable is *COVER_P_ADDR_LOOP_NESTED*, which is the expected result because the design specification does not permit this behavior with the used parameters. If this cover were reachable, it would be due to the unexpected design behavior.

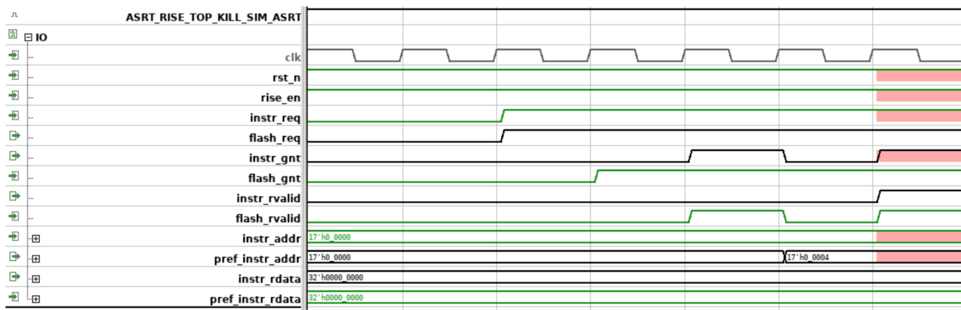


Figure 5.2: Trace of failing design assertion "*ASRT_RISE_TOP_KILL_SIM_ASRT*".

The trace shown for *COVER_P_ADDR_RANDOM_EACH_CYCLE* is a linear sequence of addresses, which is not the desired behavior for this specific cover.

The cover *COVER_P_ADDR_LOOPING_2_SEQUENTIAL_LOOPS* is only reachable when the assumption *ASSUME_ADDR_IS_STABLE_DURING_REQ* is disabled.

Result of created formal assertions

OBI connection with processor. The OBI for communication with the processor is in the slave mode, for which we check only **OBI-R-5** by assertions. The assertion for this interface was passed along with its related cover.

OBI connection with memory. The OBI for communication with memory is in the master mode, for which we have implemented assertions for **OBI-R-3**. The assertion for **OBI-R-3-1** fails even with all reasonable assumptions about design inputs. The assertion for **OBI-R-3-2** passes when the assumption about correct interface operation by processot is active, but when it is disabled, the assertion fails.

Assertions checking function of the design. From the assertions written so far to check the correct behavior of the design, there is only one failing. This assertion is *ASRT_DATA_FROM_CORRECT_ADDR*. The CEX for this assertion shows the request sent to the memory when there is a prediction miss.

5.1.2 Functional (UVM)

The results of functional verification were obtained with the help of the simulator Xcelium from Cadence Design Systems, Inc. The simulator support for the UVM package was used during the debugging of the UVM environment. Each test was run multiple times with random seeds by using a custom script to run the test and then save the *.log* file under the name: *<test_name>.<seed>.log*, for manual check for errors. Each test was run with 50 different seeds to decrease the chance of missing scenarios that would cause the test to fail.

Found problems

By running the simulations, we were able to identify the following problems with the design:

- The DUT is not able to detect the request for a single address multiple times correctly. Instead of requesting the same address from memory, it starts to oscillate between requesting the current address and the sequentially next address.
- Some of the tests are failing due to the sudden change of predictor FSM for a given address from ST to SNT. This scenario is captured in Figure 5.3.
- When two consecutive mispredictions are encountered, the DUT registers the second address in association memory to the position that has the line index increased by one. The wrong index is also used to access and update the FSM predictor table. This scenario is captured in Figure 5.4.

5.2 Discussion of results

Here we will discuss the probable causes of design bugs, their effect on the function of the design, and some possible ways to fix them. Some of the problems are most likely caused by the same undesired behavior of the design. Since the modification of the DUT is outside of the scope of this work, we will present the probable cause of the design problem, and in some cases, we will propose a solution. All of the bugs will be reported to the design team for resolution.

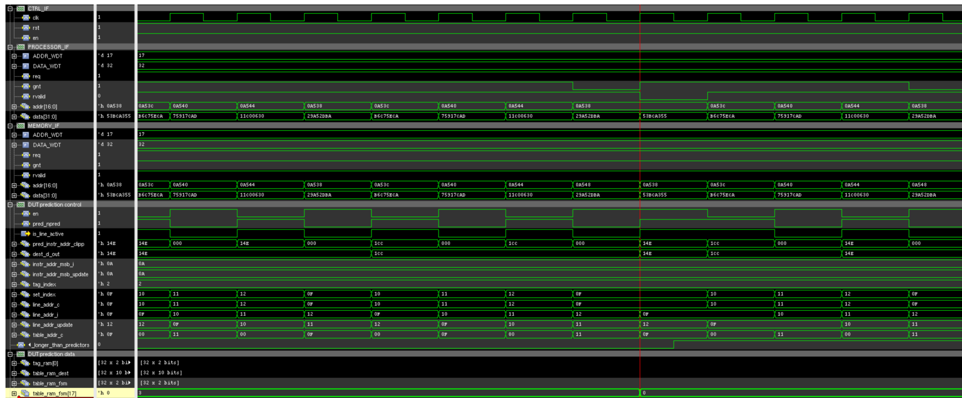


Figure 5.3: Simulator output with captured wrong DUT predictor line update and assigning of initial predictor FSM value to the already initialized line. (The trace can be also found as attached file: *screenshots/exp_table_wrong_state_transition.png*)

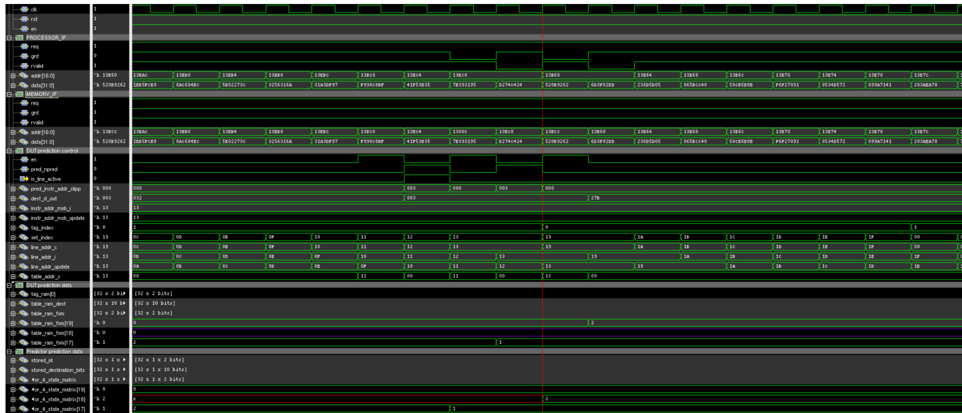


Figure 5.4: Simulator output with captured scenario, where DUT updates the wrong line of the FSM predictor table. (The trace can be also found as attached file: *screenshots/exp_table_write_wrong_line.png*)

5.2.1 Formal (Jasper Gold)

Discussion of designers' checks of code results

The failing assertion *ASRT_RISE_TOP_KILL_SIM_ASRT* shows us that the design will read the wrong instruction from memory when the processor is stalling at a single address. This behavior was also observed in the simulation, where we needed to remove the processor stall sequence at the end of the sanity test because of oscillation. The solution for this particular failing assertion can be to simply route the *instr_gnt* signal through an AND gate, which would force a logical value of 0 to the signal when *instr_addr* and the *pref_instr_addr* are not the same.

In the case of assertion *asrt_table_read*, it is harder to find the leading cause of this bug without a deep understanding of the design because of the complex assertion disable expression. Because of that, the failing assertion will only be reported to the designer with the data provided by JasperGold.

Discussion of created formal coverage results

The cover designed to show correctly predicting the end of the nested loop is not reachable, as was expected. This cover is not reachable because we would need to correctly predict the end of the inner loop multiple times, which is not possible with the design parameters that we selected.
The cover to request a random, different address for each cycle has been reached by the tool, but when reviewing the results, we classified it as failing. The cause of this decision is that we wanted to see a random sequence of addresses, but we got a linear sequence. This could be fixed by limiting the new address to not be a sequential address or an address visited in some number of previous cycles. Since this cover did not produce the desired trace to show us, it did not contribute any new information to the verification process.
The cover that was designed to show us the two looping sequences of addresses that are consecutively requested is only reachable when we disable the assumption, which is forcing *instr_addr* to stay stable during the address phase of the OBI protocol. This means that our design is able to exercise this behavior, but only if the processor OBI violates the specifications of the OBI protocol.
The cover, whose purpose is to show us that *pref_instr_addr* is being constant for 10 continuous cycles, is reachable, against the found bug, which forced us to remove stall simulation at the end of the sanity test. The provided trace shows us that the address sequentially after the address that we want to stall on needs to be set as a branch with the destination pointing to the address of the stall. From this trace, we can infer that the address used as a base for prediction after a branch is taken is the address sequentially following the branch destination address. Although this cover was marked as reachable by JasperGold, it was not possible to reach it in the way that we would expect, and thus the trace is a sign of a design flaw.

Discussion of created formal assertions results

The DUT violates the OBI protocol on the connection interface with memory. Violation of address stability during a request before a grant is received can

be overlooked for two reasons. The first reason is that it is the same bug that is already acknowledged, for the type of processor that the design is intended for, at the end of the OBI specification. The second reason is that we can expect the memory to have an instant reaction time because it is intended to store and provide instructions for the processor. We can expect the clock to be at the frequency where the memory is able to present the data from the requested address in the cycle immediately following the cycle in which the request was received. Because of that, this violation has low severity.
The other part of the violation of **OBI-R-3** is more concerning. The fact that the *flash_req_out* signal is able to violate the specifications of the OBI protocol can be a problem when integrating this design into a larger system. When we looked at the source of this behavior, we found that the *flash_req_out* signal is always directly connected to the *instr_req_in* signal, which is allowing it to violate the OBI protocol. This is a very severe problem because it directly violates specification **SPEC-R-7**, which is connected to the OBI specification **OBI-R-3**.
The failing assertion *ASRT_DATA_FROM_CORRECT_ADDR* is failing because of a flaw in its implementation. This assertion always checks if the input and output addresses are the same when sending a request to memory, but we missed the possibility that the prediction was incorrect and the correct address will be requested in the next cycle. The application of this fix should not be very problematic.

5.2.2 Functional (UVM)

The problem with the design not being able to comply with the processor stall at the single address is already mentioned above. This time we confirmed it by simulation, and we could see that the address requested from memory was oscilating between the address requested by the processor and the address sequentially after. This behavior is strongly undesirable in our design because of its intended use for applications focused on low power consumption. The switching of gates caused by the address oscillation and the resulting need for memory to recall data from different addresses each cycle leads to an undesirable increase in power consumption. Because of these reasons, the severity of this problem is medium (corner case).
The violation of predictor FSM transitions, defined by the 4-state FSM predictor diagram, could be caused by the wrong resolution of the signal *pred_npred*that is used inside the *rise_table* block of the design. The logical zero value of this signal defines the state of the currently accessed FSM predictor as 2'b00, which then overwrites the state stored in the predictor table. This directly violates the possible state transitions defined by the 4-state FSM predictor. It also breaks the already-recorded data inside our predictor. From that, we can conclude that the severity is medium (corner

case) because the design is able to recover from mispredictions at the expense of one clock cycle.
The wrong index that is used for accessing the associative memory and table of predictors is caused by using the address, sequentially following the branch destination address, as the source of the index value. This bug causes problems in two ways. First, it hinders the registration of consecutive jumps (which could be a result of the decision tree in the embedded program), and second, it may affect other registered branches. This behavior also breaks the prediction mechanism and rewrites our recorded data with the wrong values. That means the severity is the same as above (medium (corner case) severity).

Chapter 6

Conclusion

In this work, we presented two approaches to the verification of digital circuit design. The first was formal verification, which uses mathematical proofs to present us with answers about design. We also used these proven properties of the design to decrease the complexity of the UVM environment. The simplification was possible due to the fact that we verified the block interface, so there was no real reason to add more randomized parameters for driving stimuli in simulation tests. The implemented UVM testbench environment incorporates three separate agents for individually driving each of the interfaces. The previous formal verification saved us a lot of work with verifying the interface protocols, and we could focus more of the verification effort on the verification of design functionality. Since, for the purpose of this work, there is no iteration between finding the flaws in the design and reporting them to the designer for fixing, we will report the design flaws after this work is complete. In a normal situation, there would be multiple iterations of verification, finding design flaws, and fixing them by designers.

6.1 Implemented files

In this work, I implemented the UVM environment used for functional verification and the verification components with their top module used in formal verification. The files created by me are inside the attached package in folders *scripts/* , *src/tb/* , and *src/list_files/* . The files inside folders *src/design/* and *src/package/* were not created by me.

6.2 Comparing the verification approaches

With the formal verification, it was very easy to verify the properties of the OBI interface, especially in situations where the *en* signal is deasserted and the design shall directly connect the interfaces together. Also, the verification effort for checking compliance with certain points of the OBI protocol specification was very low. Contrary to that, the formal verification of the valid memory position of data passed to the processor would be a lot more difficult than using the simulation with a simple model of the memory. The time required to first run the formal verification tool and to verify some combinatorial behavior, which is sequentially relatively short, is very short. When using the UVM environment for verification via simulation, it requires a non-negligible amount of verification effort. After the UVM environment is completed, it is relatively easy to add more tests when needed. The difficulty of implementing additional tests is also strongly influenced by the implementation quality of the UVM environment.
In summary, formal verification is quick to deploy and is very useful for the verification of combinatorial or short sequential logic. This allows us to decrease the complexity of the UVM verification environment. The UVM simulation approach is slow to deploy because it needs all modules for controlling the simulation and evaluation of gathered data, but when correctly designed, it allows relatively quick implementation of sequentially long tests. The UVM is better for verification of longer sequences and the functionality of more complex designs.

parameter	Functional UVM + XCELIUM	Formal JasperGold
Time required for the first results	long	very short
Complexity of the implemented environment	high	low
Difficulty for expanding verification	low	high
Scope of specifications that can be verified	large	small
Provides exhaustive proof	no	yes

Table 6.1: Table comparing different attributes of used verification methods.

6.3 Design flaws

During verification, we encountered some major flaws in the design that are preventing the next verification until they are fixed. One of them is the wrong indexation of associative memory and the FSM predictor table when two consecutive branches/prediction misses are detected. The other is the inability of the design to stay at requesting a single address because this request from the processor (without previous preparation of the prediction table and memory) results in *pref_addr_out* oscilating between two values.

6.4 Next steps

The next steps in the verification of this design are fixing the already-found flaws. Until the fixes are done, we shall implement coverage for simulation, which we skipped due to the limited time. After that, we shall iteratively write more tests, debug the UVM environment, and fix bugs in the design. Functional coverage will also need to be added, and automatic formal coverage will need to be performed.
Run the tests for more configurations of design parameters. In this work, we were only able to use the parameters as they were defined by the designer because, after changing them, we encountered a compilation error of the design on the second compilation. Because of that, we were only able to use one level of prediction, and we could not verify any of the functions that require branching history to be present. (The UVM environment is already prepared for the existence of the global branch history.)

Bibliography

[1] Arjan, Bink. *OBI-v1.4* [online]. Mar 2022 [cit. 2023-11-15]. Avaible from: https://github.com/openhwgroup/obi/blob/main/OBI-v1.4.pdf

[2] Nelson, Campos. *A basic tutorial of uvm* [online]. Sep 2016 [cit. 2023-11-15]. Avaible from: https://sistenix.com/basic_uvm.html

[3] *Jasper formal fundamentals training* [online]. Cadence Design Systems, Inc. [cit. 2023-11-15]. Avaible from: https://www.cadence.com/en_US/home/training/all-courses/86123.html

[4] Harry, Foster. *The 2022 wilson research group functional verification study* [online]. Jan 2023 [cit. 2024-01-04]. Avaible from: https://blogs.sw.siemens.com/verificationhorizons/2022/12/18/part-9-the-2020-wilson-research-group-functional-verification-study-2/

[5] *IEEE Standard for Universal Verification Methodology Language Reference Manual,* in IEEE Std 1800.2-2017 , pp.1-472, 26 May 2017, doi: 10.1109/IEEESTD.2017.7932212.

[6] Jonas Julian, Jensen. *How to create a self-checking testbench* [online]. Jul 2023 [cit. 2023-11-14]. Avaible from: https://vhdlwhiz.com/how-to-create-a-self-checking-testbench/

[7] Martin Laštovka, *Implementace instrukční sady pro risc-v procesor* [online]. Jun 2022 [cit. 2023-11-15]. Avaible from: https://dspace.cvut.cz/handle/10467/100975

[8] *Verification Plan: A document that defines what functional verification is going to be performed* [online]. Semiconductor Engineering. Nov 2018 [cit. 2023-11-19]. Avaible from:

https://semiengineering.com/knowledge_centers/eda-design/verification/verification-plan/

[9] *Systemverilog accelerated verification with uvm training v1.2.5* [online]. Cadence Design Systems, Inc. [cit. 2023-11-15]. Avaible from: https://www.cadence.com/en_US/home/training/all-courses/86070.html

[10] *Systemverilog for design and verification training* [online]. Cadence Design Systems, Inc. [cit. 2023-11-15]. Avaible from: https://www.cadence.com/en_US/home/training/all-courses/82143.html

[11] Various, Writers. *Formal verification 101* [online]. Sep 2013 [cit. 2023-02-15]. Avaible from: https://semiengineering.com/formal-verification-101/

Appendix A

OBI - implemented functionality

port	implemented	tie-off value	port	implemented	tie-off value
clk	Y				
reset_n	Y				
req	Y		rvalid	Y	
gnt	Y		rready	N	1’b1
addr[]	Y		rdata[]	Y	
we	N	1’b0	err	N	1’b0
be[]	N	4’b1111	ruser[]	N	dynamic
wdata[]	N	’b0	rid[]	N	dynamic
auser[]	N	’b0	exokay	N	1’b0
wuser	N	’b0	rvalidpar	N	dynamic
aid[]	N	’b0	rreadypar	N	dynamic
atop[5:0]	N	’b0			
memtype[1:0]	N	2’b0			
prot[2:0]	N	3’b111			
dbg	N	1’b0			
reqpar	N	dynamic			
gntpar	N	dynamic			
achk[]	N	dynamic			

Table A.1: List of OBI signals showing which of them are used by the design and tie-off values of unused signals. The *dynamic* tie-off value means that single value can’t be assigned to signal and we act as if value of those signals is always correct

ID	relevant	note
R-1	Y	
R-2	Y	
R-3	Y	A known bug of RI5CY processor is that it doesn't necessarily keep its address phase signals stable during the address phase [7]
R-4	Y	Signal *rready* not used
R-5	Y	
R-6	Y	
R-7	N	Signal *be* not used
R-8	N	Signal *be* not used
R-9	N	Signals *aid* and *rid* not used
R-10	N	Signal *atop* not used
R-11	N	We only use read transactions
R-12	N	Signal *exokay* not used
R-13	N	Signal *reqpar* not used
R-14	N	Signal *gntpar* not used
R-15	N	Signal *rvalidpar* not used
R-16	N	Signal *rreadypar* not used
R-17	N	Signal *achk* not used
R-18	N	Signal *rchk* not used
R-19	Y	
R-20	Y	Constant *COMB_GNT* is not defined. Default value of *False* is assumed.
R-21	N	
R-22	Y	
R-23	Y	
R-24	Y	
R-25	N	Signal *rready* not used
R-26	Y	

Table A.2: Requirement list of the OBI protocol with information about which are relevant for our design.

Appendix B

Table of verification methods used for requirements

ID	Formal	Functional	Not verified by presented methods
OBI-R-1			X
OBI-R-2	X		
OBI-R-3	X		
OBI-R-4	X		
OBI-R-5	X		
OBI-R-6			X
OBI-R-19			X
OBI-R-20			X
OBI-R-22			X
OBI-R-23			X
OBI-R-24			X
OBI-R-26	X		
SPEC-R-1	X		
SPEC-R-2	X		
SPEC-R-3	X		
SPEC-R-4	X		
SPEC-R-5	X		
SPEC-R-6	X		
SPEC-R-7	X		
SPEC-R-8	X		
SPEC-R-9	X		
SPEC-R-10	X		
SPEC-R-11	X		
SPEC-R-12		X	

SPEC-R-13		X	
SPEC-R-14	X	X	
SPEC-R-15		X	
SPEC-R-16		X	
SPEC-R-17	X		
SPEC-R-18		X	
SPEC-R-19		X	
SPEC-R-20		X	
SPEC-R-21		X	
SPEC-R-22	X	X	
SPEC-R-23		X	

Table B.1: List of requirements and which verification methods were used to verify them. (The corresponding columns are marked by 'X')