UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

APPLYING FORMAL VERIFICATION TECHNIQUES TO VERIFY A FETCH UNIT

ADRIÁN JIMÉNEZ GONZÁLEZ

Thesis supervisor: ALBERTO MORENO VEGA (Semidynamics Technology Services SL)

Tutor: ROGER ESPASA SANS (Department of Computer Architecture)

Degree: Master's Degree in Innovation and Research in Informatics (High Performance Computing)

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Abstract

The fetch unit of a GPP is a difficult block to verify. The complexity of the problem stems from the size of the state (cache, pipeline, etc.). Formal verification techniques, traditionally, have difficulty handling problems with large amounts of memory. We will explore different approaches to deal with these limitations such that we are able to reduce the wall time needed by the Formal Verification tools.

This project covers the design and implementation of a Testbench for Formal Verification. In addition, it introduces different techniques and configurations to reduce the complexity of the Design Under Test and how they are applied into our Formal Verification Testbench.

Finally, the performance impact of each technique is analyzed at the end of the document.

Acknowledgment

I would like to thank my supervisor Alberto Moreno Vega for his help during the completion of the thesis, as well as for his advice and experience. I would also like to thank Semidynamics for the opportunity to do the thesis with them, allowing me to enrich myself as a person and as a professional.

I would also like to thank my parents, my brother and my girlfriend for helping me to make the effort more bearable.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

Processors have exponentially increased the number of components since the early designs (Moore's law [1]), and with it, their logical complexity. As with software, it is important to ensure that the behavior of the code is in accordance with the specification provided at the design stage. In Figure 1.1, the growth in the ratio of transistors per chip over the years can be seen. This increase in complexity is where verification comes into play.

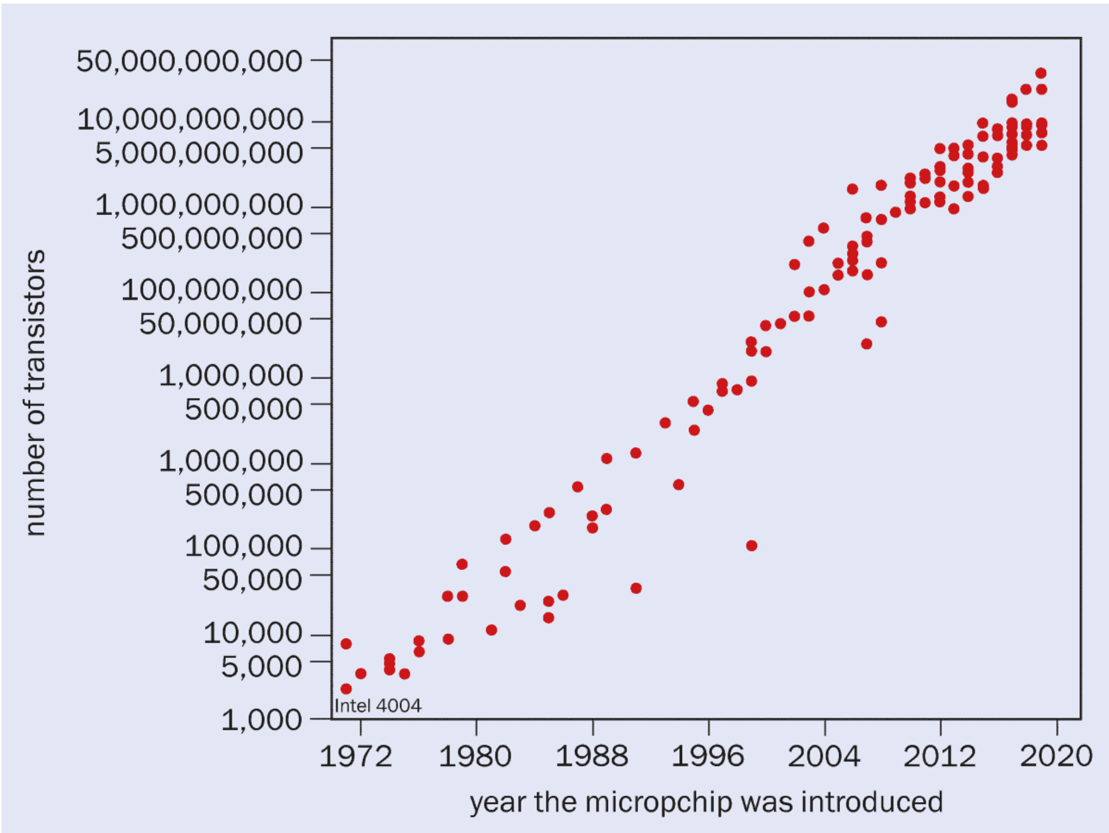

Figure 1.1: Exponential Grow of the Transistors per Chip [2]

Verification is an aspect of Electronic Design Automation (EDA) that has become increasingly important with the increasing complexity of today's chip designs. It is one of the most resource-intensive and time-consuming phases of chip design. It can consume up to 70% of the resources, both temporal and economic, of the project development cycle.

Verification ensures that the Register-Transfer Language (RTL) that has been designed matches the architecture specification. Verification is important because a failure after obtaining the physical hardware has a very high cost and can cause losses in the magnitude of millions of dollars. One of the best known cases is the Intel Pentium 4 bug with the *fdiv* instruction [3], which caused losses of more than $450 million to the company. These losses can happen even when the product has not been released, as was the case with the Intel Pentium 4. The cost of redesigning the layout and manufacturing a new die is very high, which coupled with a delayed market release can cause potential loss of purchases and market share.

Traditionally, hardware verification has been carried out through simulation. This technique consists of creating a Golden Model (GM), which meets the expected behavior according to the design specification, and then comparing it to the Design Under Test (DUT). To do this, a random sequence of inputs is generated and introduced into both blocks or modules and the outputs of both are checked, confirming that the DUT generates the desired results.

Formal Verification (FV) is a technique that can speeds up RTL verification process compared to traditional simulation in the right conditions. This technique increases the coverage and ensures the correct functioning of the module being verified. Formal verification tools are responsible for checking the entire space of possible values that can be simulated. FV does not try to simulate all possible combinations of inputs, but uses different mathematical techniques to consider all possible combinations within the specified constraints.

FV main downside is the execution time. This type of verification can obtain exponential times. A bad implementation of FV can imply that a Testbench does not finish even in months. For FV, any Testbench lasting more than one day can be considered excessive and not optimal. So, large state space results in enormous runtime and a big downgrade in the verification stage performance. Memory modules are one example of modules that can increase exponentially the state space.

FV has improved its performance substantially in recent years with different strategies and algorithms that make it possible to increase the size of the Conjunctive Normal Form (CNF) used by the Satifiability Checkers (SAT-solvers) on which FV engines depend. Even so, it is far from being scalable due to the complexity of the size of the DUTs. These technical improvements have achieved a level of robustness that, when FV was first proposed, was unthinkable. What is interesting is that FV can be easily used by engineers with RTL notions.

Formal verification involves a new approach to RTL design when testing. Instead of looking at the code for values that stimulate one part of the DUT or another, as we would do in simulation, we examine the code from a mathematical point of view by exploring the set of possible values.

The verification of the Fetch Unit of a commercial microprocessor is challenging. Not because of its logic, but because of the spatial complexity that this unit presents, such as cache, pipeline and Translation Lookaside Buffer (TLB). Traditionally, commercial formal verification tools have problems dealing with the verification of units with such a large amount of memory. Therefore, we are going to explore different techniques to deal with the limitations presented by the memory, and therefore, this module, reducing the execution time of the tests.

The document is organized in nine sections, the first of which we are currently in and serves as an introduction to the thesis. The second section is the motivation for the project, followed by the State of the Art, and an explanation of what a Fetch Unit is. We then proceed to explain different techniques that can be used to reduce the complexity of the tests and how we have introduced these techniques in our Testbench. Then, we will proceed to show the results obtained with these configurations, and we will end with the Conclusions and Future Work.

2. Motivation

Formal Verification is used in the verification process of cutting-edge processors [4, 5]. Formal techniques appear to be suited to smaller or moderate size designs. The use of FV techniques for complete designs is limited by the size of the space of possible DUT states. So, generally, the preferred way to verify is from the bottom-to-top technique, where each module is first verified independently and then combined in clusters.

The main motivation for using formal is the strong guarantees it provides. This technique allows, in case of having appropriate assumptions, to cover all possible cases of the DUT mathematically. It ensures a correct performance under the specific conditions we are defining. Formal Verification covers all the possible states of our DUT. Another point in favor would be the use of System Verilog Assertions (SVA) [6] to generate the proofs for our test.

Compared to dynamic verification (which relies on verification by simulation), the traditional technique used in EDA, it has a number of advantages as well as disadvantages. The main difference between the two is the way tests are performed on the DUT. While simulation explores isolated state points depending on specific test cases, formal verification covers all possible states at once.

Although they share [7] some of the same components such as statements, they are used in different ways and are conceptually different, and it is important to understand how to prepare a Testbench for FV. We are talking about assertions, assumptions and coverpoints.

- **Assertion**. In simulation, it indicates whether a certain condition holds for the duration of the simulation, e.g. the counter of a FIFO is never greater than the queue size, or less than 0. In the case of FV, it is used to check if it is possible for a certain sequence to violate the condition defined by the assertion. In both cases the purpose is very similar, although with subtle differences, in simulation it is checked if it has occurred within the possible values that the test takes during the execution, while in FV it looks for possible combinations of inputs to force that violation of the assertion.

- **Assumption**. In simulation, it indicates whether there was any value in the test that violated the assumption. As you may have noticed, conceptually, assertion and assumption are identical in the case of simulation. However, assumption is used differently in FV, since it is used to limit the space of possible inputs that can be introduced in the test. And thus reduce the space of states of the test. For example, in FV an assumption could be that the core is always in the User mode privilege, reducing the different possible privilege modes of the core (i.e., User, Supervisor, Machine, ...) to only one.

- **Coverpoints**. In simulation, it is used to check whether events from a list of events have occurred at least one, for example if a cache read has been performed and made a miss. For FV, it is used to check that it is possible to exercise a certain condition with a set of test values.

Having shown some differences between the two in terms of the behavior of these structures, which are widely used in FV and have their counterpart in simulation, it is also important to show other aspects that will help us better understand how formal verification works, when it is appropriate to use it, what it provides and why we have decided to use it.

It is important to know the strengths and weaknesses of each of the strategies. This will help us to choose which of the strategies best fits the design to be verified, saving unnecessary effort of a bad choice. This decision is important from a productivity point of view, as it helps to better manage available resources by reducing costs. Given the major paradigm shift from formal with respect to simulation (or *vice versa*), the only piece that could be reused would be the Testbench specification.

We are going to present different aspects that will help us when choosing the paradigm. The first one we are going to talk about is the size that can be covered by both models.

The verification by simulation allows working with models of all sizes. This allows us to perform a Testbench for a unitary module, a cluster of these (a module that connects different unitary modules and allows us to test their joint behavior), and even full-chip size. Including both behavioral code and synthesizable code. This strategy allows us to make tests as large as we wish. On the other hand, Formal Verification fits in unitary modules, or clusters. In case we want to test the whole core or full-chip, it would be necessary to eliminate a lot of combinational logic, and it only allows us to test synthesizable code. As we can see, the first disadvantage of choosing Formal is that the size and scope of the test must be contained.

The difference in the complexity of the techniques dictates the size of the design that can be tackled by each approach. While Simulation verification has almost linear complexity, Formal has exponential complexity. A benchmark number to start considering that Formal is not suitable for our Testbench is approximately $\sim 40,000$ state elements [8]. And in case of exceeding this number, knowledge of advanced techniques would be necessary to reduce the complexity of the test.

The next aspect would be how they traverse the search space. The two techniques approach the verification from two totally opposite points of view. Simulation could be defined as "*Describing the path*" and Formal as "*Destination targeted*". *Describing the path* means that this technique goes from state to state of the DUT randomly, generating random values to check if the behavior is the desired one. In *Destination targeted* techniques, on the other hand, the engineer specifies the desired behavior or property, by means of a property, and the application finds the way to cover that property, or otherwise, it finds that the property is not true for certain cases.

In Figure 2.1 we can observe the behavior described above. The upper figure represents the verification in simulation and the lower one represents the formal

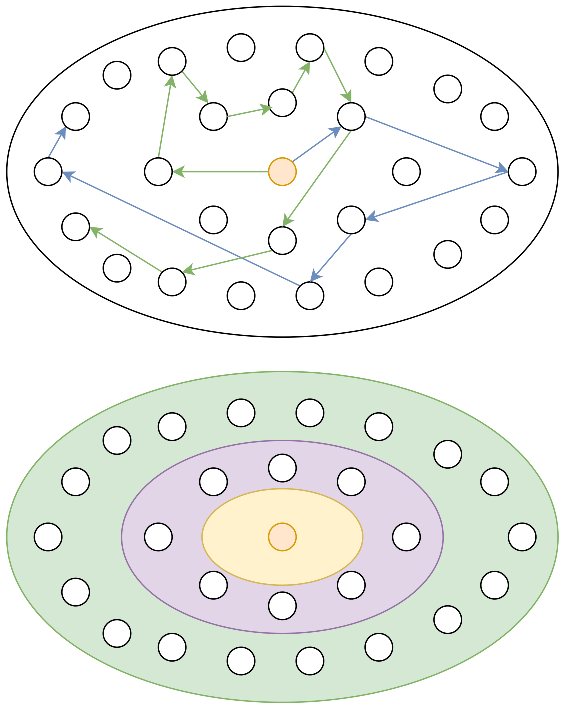

Figure 2.1: Describing the Path vs. Destination Targeted

verification. Assuming that both figures are the possible states of the DUT and the yellow bullet at the center represents the initial state of our module to be verified. We can observe clear differences between both techniques.

If we focus on the first figure, we can see how a path is generated between the different states that are being verified. This is because the simulation is introducing inputs to the DUT sequentially, stimulating different states. We can distinguish arrows of two colors, which represent different simulations. As we can see, it is possible that in different simulations the same states are exercised, just as it is possible that certain states are never stimulated. The name “describe the path” is given by the path of the different states that are stimulated.

In the figure that represents the formal verification, we can see the same initial state (yellow bullet). The difference is that the Formal Verification Tool will test all “reachable” states from this initial state (first purple ring) that is 1 clock after the initial state, and then the reachable states from clock 2. In this way, our DUT is fully verified under the conditions, configuration and set of asserts with which the test has been run, leaving no possible states of our DUT unverified.

For the simulation, it is necessary to create a structure for our Testbench, which is able to introduce the values to our model and DUT for comparison. Given the nature of the simulation, it is necessary to perform several cycles to try to cover all the behaviors, and even to perform several iterations to try to cover the largest number of paths within the possible states.

Meanwhile, Formal is more straightforward. It is only necessary to define a set of assertions, assumptions and covergroups that define the behavior of our model, and with a single FV run, if the assumptions and assertions are correct, it is able to completely cover the verification space of possible states of the DUT to be verified.

In Listing 2.1 and Listing 2.2 we can see the different structure of two potential examples and different components needed for both approaches. Listing 2.1 shows the structure of a simple Unit Test in Universal Verification Methodology (UVM) [9]. It consists of a series of agents that introduce inputs and monitor the input and output values, among other components such as interfaces. In addition, it is necessary to generate a valid sequence for input. We also find an environment, which needs a Golden Model (GM) that mimics the specification provided by the requirements, and a scoreboard to compare the values generated by the DUT and the GM. Other components in this section would be the covergroups and checkers. And finally, we find another folder that is responsible for configuring the tests, DUT and connections between different components mentioned above. We can also find the exclusions[1] file for our testplan and the testplan itself.

Listing 2.2, on the other hand, requires a much simpler structure that is totally different from the one shown in Listing 2.1. It can be seen that only the waivers files (exclusions), tool setup scripts, and formal files where the different asserts, assumes, blackboxes, etc. are located are needed.

[1]Exclusions are used to remove some cases that are not possible to hit due to configuration, design or other reasons.

Listing 2.1: Dynamic test folders structure (UVM)

```
--- DUT_testplan.hvp
--- agents
| |-- agent_pkg.sv
| |-- interfaces
| | |-- DUT_i_if.sv
| | --- DUT_o_if.sv
| |-- sequences
| | |-- DUT_flush_sequence.sv
| | |-- DUT_reset_sequence.sv
| | --- DUT_virtual_sequence.sv
| |-- DUT_agent_i.sv
| |-- DUT_agent_o.sv
| |-- DUT_driver.sv
| |-- DUT_item_i.sv
| |-- DUT_item_o.sv
| |-- DUT_monitor_i.sv
| --- DUT_monitor_o.sv
|-- env
| |-- env_pkg.sv
| |-- DUT_env.sv
| |-- DUT_fcov.sv
| |-- DUT_goldenModel.sv
| |-- DUT_scoreboard.sv
| --- DUT_checker.sv
|-- exclusions.el
--- tests
    |-- DUT_base_test.sv
    |-- DUT_flushes_test.sv
    |-- DUT_reset_test.sv
    |-- DUT_universal_test.sv
    |-- test_pkg.sv
    --- top.sv
```

Listing 2.2: Formal Verification test folders structure

```
--- FormalTest
    |-- formal.sv
    |-- formal.sva
    |-- formal_mem.sv
    |-- module1_formal.sv
    |-- setup.tcl
    |-- setup_2.tcl
    --- waivers.tcl
```

As can be seen, both approaches to verify a DUT are totally different, and it is not possible to take advantage of something that has been done in one to reuse to create the Testbench in the other approach. Therefore, it is important to carry out a previous study that allows us to focus the resources correctly from the beginning to avoid wasting them.

The way in which the verification is obtained, not only affects the above-mentioned, but also affects the values that are checked. Each execution of the simulation, if there is a seed change, will check a different finite set of values, and not having checked all the existing paths, values and combinations, it is possible that some illegal state is not found. Formally, by checking the whole spatial sample of possible states of the DUT, if after a single execution no assertion of our test has been invalidated, we can affirm that under the provided conditions (set of assumptions or constraints) there is no combination of values or states that invalidate those assertions. Therefore, a great effort is necessary to identify and create constraints that allow us to test not only realistic values, but also to cover the maximum of these values.

In Figure 2.2, we have the same concept as in Figure 2.1. The upper figure represents verification by simulation, and the lower figure represents Formal Verification. In order to explain the interaction of both approaches in the presence of bugs in the RTL we are verifying, we have introduced some states of the DUT where bugs are found. The red bullets represent these states.

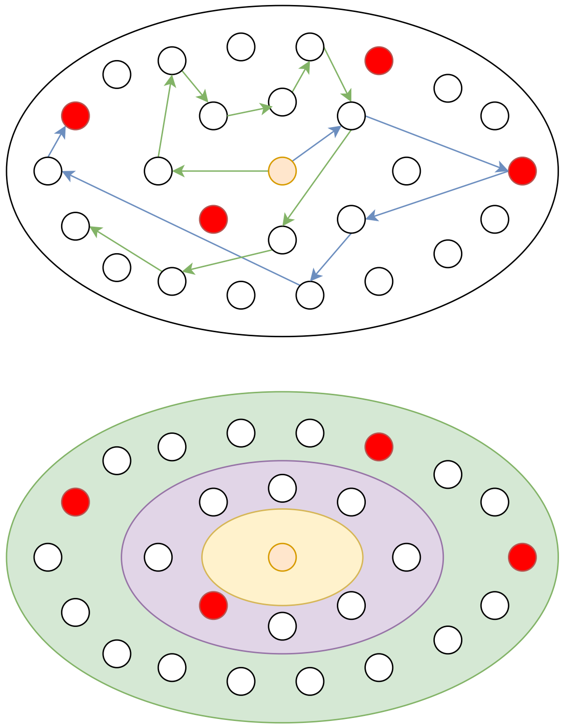

Figure 2.2: Bug Hunting

In the simulation technique, it is possible that not all the bugs are found, since we depend on the inputs introduced in the sequence and how they stimulate the DUT. Therefore, in order to catch as many bugs as possible, different simulations are necessary so that, with different seeds, the inputs try to stimulate as many states as possible. Even so, we can not ensure that all transitions between states or all possible states have been covered, and may leave some bugs in corner cases that are difficult to access.

On the other hand, the states that are reachable for formal verification and that meet the assumptions that are imposed, will be stimulated, and therefore, will be able to find the different bugs that violate an assertion. In this way, with a correct set of assumptions and assertions, we can get a DUT completely verified and with a strong guarantee that it will work under the specified assumption.

In Figure 2.3, we can see how assumptions work in Formal Verification and how they affect the space being tested. As mentioned above, assumptions serve to reduce the space of possible inputs to the DUT. This limitation can be useful in different cases, such as when testing on an RTL that has not yet been completed and therefore, there are some features that we know in advance that will not work. Then we can limit those inputs to avoid testing on these features. Another reason could be to focus the tests on one mode of the DUT or to simulate the operation of another module that connects with our DUT.

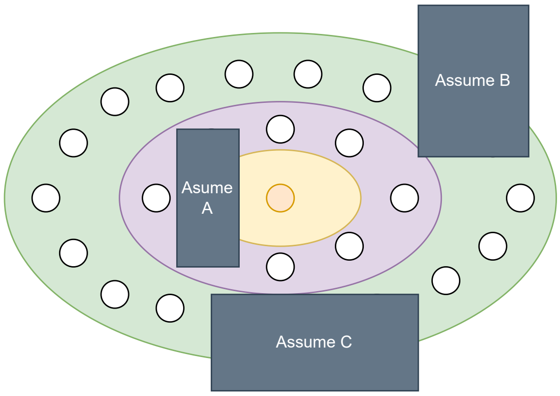

Figure 2.3: How Assumes Work on FV

Also, we can observe different assumptions and how they interact with the space of possible tests. For example, in Figure 2.3, “Assume A” only eliminates reachable states. The other two also limit space outside the reachable ones. This is normal and does not downgrade the expected performance of the test. The important thing is to test as many states as possible that are considered legal for our DUT.

Another important difference between approaches is in the way they present results. For the simulation, we are shown the entire simulated trace, which is usually tens or hundreds of thousands of cycles. This is because the trace usually contains the startup process of the machine and all the values entered until the bug is hit. Formal, on the other hand, shows a counterexample to the assertion, which coincides with the minimum trace to get the bug. This results in shorter traces and with fewer inputs that do not affect the problem, being able to find traces, in general, of just a few tens of cycles.

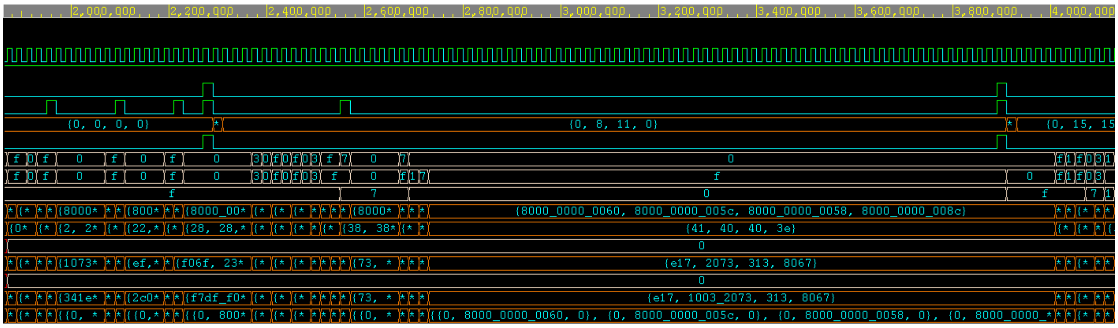

Figure 2.4: Output Waves Simulation Example

Figure 2.5: Output Waves FV Example

In Figure 2.4 you can see some waves of a short test. The first signal shown corresponds to the clock. As we can see, there are many cycles in the screenshot shown, which does not even make up 3% of the complete test for this example. That said, this test consists of 35500 cycles composed of a short code in assembly with a very specific purpose. In cases of more complex regressions and with more complex purposes, they can reach millions of cycles, complicating the visualization of the waves and their understanding or interpretation.

In Figure 2.5, we can see a counterexample created by the verification tool with a much smaller scope, reaching the possible bug in fewer cycles, improving the readability and understanding of why the bug is coming and which are the intervening signals.

Although from what we have shown it may seem that there are only advantages to Formal Verification, this is certainly not the case. As cons to Formal Verification, we find that it does not fit well to modules with a high load of arithmetic operations [10, 7]. It is also necessary to apply it on highly documented modules with clear requirements [7], which help the engineer to obtain the correct asserts and assumptions for that module.

Once everything is in place, the Formal Verification can be put to use for testing the performance of a Fetch Unit, and in turn, to exploit different techniques to improve the execution times of these. The Fetch Unit has 2 modules with memory, which can challenge the performance of the tests and where we can improve the results with existing techniques to improve performance. On the other hand, the Fetch Unit does not need to perform complex arithmetic operations from the values it receives, so it is suitable to be tested by Formal Verification.

3. State of the Art

As stated before, the main objective of Formal Verification in the hardware industry is not to simulate a set of inputs, but to check the correct behavior of the design. For this purpose, different techniques proposed in the literature are used, most of them relying on Boolean algebra, among them Binary Decision Diagrams (BDD) and Satifiability Checkers (SAT-solvers).

Currently, tools are mostly based on SAT-solvers, as they are less sensitive than BDD to the size and state of the DUT and reduce the resources needed for problem storage and execution.

From an engineer's point of view, these tools are a black box. Users do not know the inner workings of the application or how the asserts they enter into the tool are checked by the tool. From the internal point of view of the application, different processes are carried out to obtain a single Boolean Algebra Expression of the problem and different algorithms are executed to obtain the results that the application shows at the end of its execution. Understanding those concepts can help the engineer generate better assertions to exploit the potential of FV.

Behind the applications, different approaches such as Boolean constraint propagations, conflict analysis and Branch-and-Bound heuristics have been refined over the last decades.

Foremost, we are going to do a background introduction that will help to fully understand the complex concepts that are related to the Formal Verification Tools.

3.1 Boolean Variables

A Boolean variable is a data type that can represent exactly two values, often 0 and 1. Because of its binary nature, it can be easily used for abstracting high or low voltages into 1s and 0s in analog circuits. Therefore, any variable used in a digital circuit is a binary representation of a voltage level.

Unfortunately, not all information can be represented with a single binary variable, which from now on we will call bit. For example, values such as counters could not be represented with only one bit. Therefore, arrays of different sizes are used depending on the size to be represented, minimizing the amount of storage required for them. A counter that is going to count up to 12 will be represented with a 4-bit array. If we represent it in SystemVerilog [6], it would be represented as shown in Listing 3.1.

Listing 3.1: 4 bit counter (from 0 to 15)

```
    bit [3:0] counter;
```

In this way, we can represent more complex structures that we can manipulate to obtain the necessary logic with Boolean operators. For example, in Listing 3.2 we can see an example of a structure in SystemVerilog, similar to a structure in C. It allows the programmer to use the different fields independently or manipulate it as desired.

Listing 3.2: Larger structure for an instruction

```
typedef struct packed {
    logic [31:20] immediate;
    logic [19:15] register_source1;
    logic [14:12] function3;
    logic [11:7] register_destination;
    logic [6:0] operation_code;
} instruction_t
```

3.2 Circuits

In order to fulfill a certain behavior, different logic gates are used together with Boolean variables. The circuits, in turn, can be represented as Boolean functions.

These circuits are then printed on a die to produce processors, Graphic Processor Unit (GPU) and other electronic components. Therefore, it is important to be clear about the operation of the circuits that have been implemented and to make sure that the behavior is as expected.

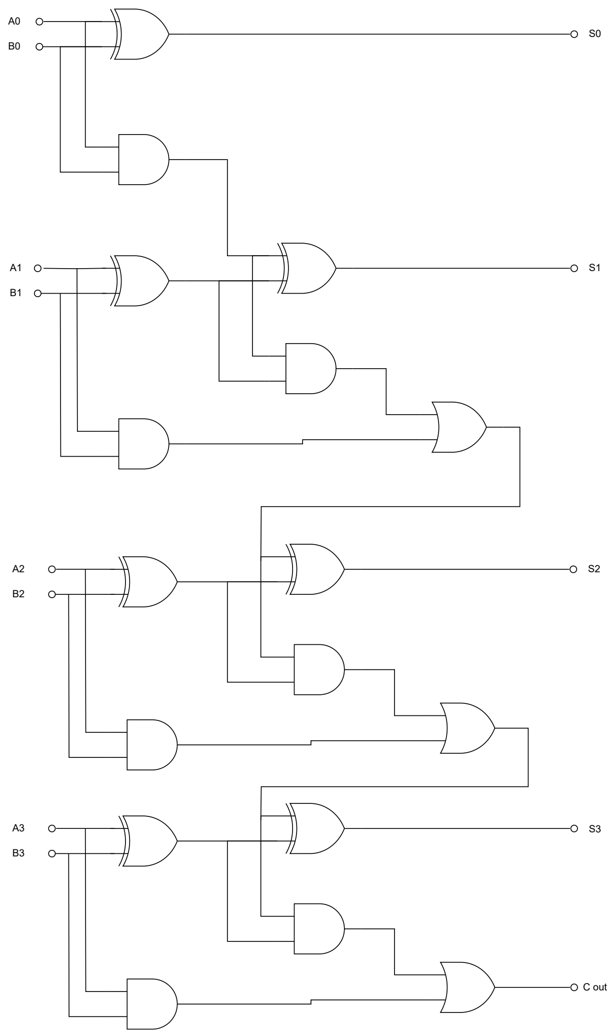

Figure 3.1: Representation of a 4-bit Full Adder Circuit

An example of a circuit can be the one shown in Figure 3.1. In this figure, we can see different logic gates that will be activated or deactivated depending on the values of the input variables and will produce an output value. Considering this circuit, we can see that the value that will come out of it corresponds to the logic function of the Equation 3.1.

$$\begin{aligned} C_{\text{out}} &= (AB) + C_{\text{in}}A \oplus C_{\text{in}}B \\ S &= A \oplus B \oplus C_{\text{in}} \end{aligned} \tag{3.1}$$

The fact that we can express a logic circuit as a Boolean expression (like in Equation 3.1), allows us to apply different efficient mathematical algorithms that allow us to check the operation of the implemented circuit.

3.3 Signals

Signals are the representation of the values that can be observed when viewing the behavior of a circuit in a specific edge of the clock. Both input and output, and even intermediate values if you have a register for it. The most common way to represent these signals is through what are called "waves". The waves show how these signals vary over the test run time of the circuit. Figure 3.2 shows an example of a waveform, which contains several signal waves.

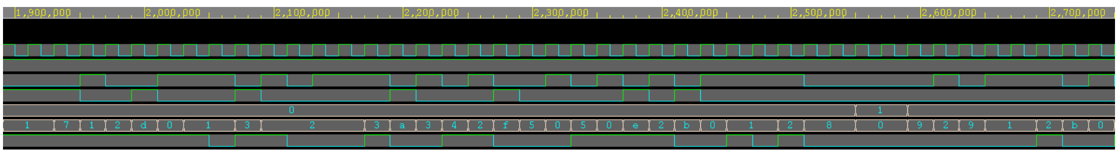

Figure 3.2: Signals Visualization Through Waves

3.4 Conjunctive Normal Form

Conjunctive Normal Form (CNF) is a way to express a boolean formula that allows the SAT-solvers to solve big assignments in affordable time [11]. CNF can also be called Product Of Sums (POS) notation. This means that the formulas are presented as ANDed clauses, where the clauses are composed of variables with OR. In equations 3.2 and 3.3 we can check an example of one expression in CNF but represented in different ways.

$$(a||!b||d)\&\&(!d||e)\&\&(a||c||!f) \tag{3.2}$$

$$(a+!b+d)*(!d+e)*(a+c+!f) \tag{3.3}$$

3.5 Binary Decision Diagrams

Binary Decision Diagrams (BDD) are directed acyclical graphs that provide a general representation for Boolean Functions [12]. Although there are different types of BDD like generic BDD, Ordered BDD (OBDD), or Reduced OBDD (ROBDD) [12], it is common to refer to ROBDD when we talk about BDD, due to the reduction it provides in terms of memory, and performance.

One of the big problems that BDD has is that this way of representing Boolean Functions is highly sensitive to size. Also, good ordering is necessary, as a bad ordering can achieve exponential resolution time [10].

BDDs provides powerful applications for the FV Tools. As the BDDs are canonical, which means that applying a set of standardized rules, we obtain a unique BDD. This implies that equivalent designs will have the same BDDs representation [7]. This property simplifies that FV Tools can check if two formulas are equivalent.

3.6 Satisfiability Checking

Satifiability Checkers (SAT-solvers) are one of the main reasons for the improved performance of Formal Verification Tools in both Software and Hardware verification [4].

SAT-solvers are computer programs that try to solve Boolean Satisfiability problems. A Boolean Satisfiability problem is a set of Boolean variables and clauses composed by the Boolean variables that relate them to each other, and by means of a series of algorithms (SAT-Solvers) it can be verified whether it is possible to satisfy the defined clauses (satisfiable) or not (unsatisfiable). So, we could define it as follows: the Boolean Satisfiability Problem checks that there is a set of clauses previously formulated that evaluate to True [11]. One of the problems is that it is an NP-complete problem [13], although it is widely used by the Formal Verification Tools and is used in several real-world problems even though it is still susceptible to timeouts or run-out of memory [14].

We can find different applications that solve SAT solvers, and that are better depending on what they are used for, some like SATO [15], Minisat [16], Rsat [17], Grasp [18], Chaff [14], with different optimizations each one for different types of problems. All these advances have made SATs more efficient and conducive to Formal Verification.

One of the main advantages of SATs over BDDs for FV is that SATs are less sensitive to the size and explosion state of the problem [14, 19]. BDDs can vary greatly depending on the variable orderings, where good orderings generate acceptable results, and poor node ordering can generate exponential complexities. The search for an optimal variable ordering is something that is considered intractable in terms of complexity to find, so heuristics are performed to find acceptable orderings. While SATs do not generate this kind of problems, being more efficient in performance and memory.

3.6.1 SAT-solver algorithms

There are two main types of algorithms for solving SAT problems, the first one is called incomplete or stochastic, also called Stochastic Local Search (SLS) are algorithms that do not provide unsatisfiability, i.e., we cannot obtain counterexamples, but can obtain a solution quickly if they exist. The second type of algorithms are the so-called complete algorithms, which guarantee finding a set of variables that satisfy the system of SAT clauses or, on the contrary, that prove to us that it is unsatisfiable.

Given the nature of the Formal Verification, the algorithms that make use of them are complete algorithms. Therefore, we are going to focus on explaining the basic operation followed by the algorithms that are classified under this type. The complete algorithms come from the Davis-Putnam and Davis-Logemann algorithms, being derivations and improvements of these techniques.

Davis-Putnam algorithm

This algorithm is a divide and conquer algorithm that uses a backtracking search method to find valid assignments to solve the SAT. The operation of this algorithm is based on dividing the CNF by a variable, assigning it 0 or 1. Subsequently, 2 subproblems appear that will be evaluated separately. This process can be interpreted as *splitting* of the boolean function that is verified.

In Listing 3.3 we can see a pseudocode of how Davis-Putnam works.

Listing 3.3: Davis-Putnam pseudocde

```
DavisPutnam(formula){
    if(formula == 1){
        Return SATISFIABLE;
    } else if (formula == 0){
        Return UNSATISFIABLE;
    } else {
        var = TakeVariableFromFormula(formula);
        formulaV0 = assignVarToFormula(formula, var, 0);
        if (DavisPutnam(formulaV0) == SATISFIABLE)
            return SATISFIABLE;
        else {
            formulaV1 = assignVarToForumla(formula, var, 1);
            return DavisPutnam(formulaV1);
        }
    }
}
```

The performance of this algorithm can be improved using a technique called *Resolution*. It requires that the clauses used are in CNF (POS formula). Its operation is based on the fact that if a variable appears as positive and negative in the formula, it can be excluded from the formula directly without affecting its resolution.

If the variable X appears as literal x and as literal $!x$, so following the statement made above, it can be simplified removing the variable X from the CNF. Resolution can be understood as follows: if a variable appears in two clauses multiplied by ANDs, these clauses are combined by ORing and eliminating the literal. In this way, the rest of the clauses remain unaffected by being multiplied (AND operation).

Davis-Logemann algorithm

This algorithm is an improvement of the Davis-Putnam algorithm, so like the previous algorithm, it is based on Back Propagation. In this algorithm, in order to reduce the number of splits that need to be made, the CNF can be analyzed for isolated literals that can be directly assigned [7, 10, 13]. By doing this assignment (called deduction), we can avoid splitting on this variable. Once the assignment has been made, it is possible that we will find new literals that can be assigned directly without the need to make new splits. When performing this technique, it is possible that another clause with only one literal is generated, since when assigning the first isolated variable, all the appearances of the same variable in the rest of the clauses disappear. In this way, we reduce the number of splits by reducing the performance cost generated by these variables on which we have avoided making a split.

This technique is known as Boolean Constraint Propagation (BCP). Since this algorithm is an improvement of the Davis Putnam algorithm, this algorithm is also known as the Davis-Putnam-Logemann-Loveland algorithm (DPLL algorithm) [7, 10].

In Listing 3.4 we can see the pseudocode for Davis-Putnam-Logemann-Loveland algorithm. It is shown that the structure of the algorithm is almost the same as the Listing 3.3, adding new functions for the deduction.

Listing 3.4: Davis-Putnam-Logemann-Loveland pseudocde

```
DavisPutnamLogemannLoveland(formula){
    newAssignment = deduction(formula);
    if(newAssignment == 1){
        Return SATISFIABLE;
    } else if (newAssignment == 0){
        Return UNSATISFIABLE;
    } else {
        var = TakeVariableFromFormula(newAssignment);
        formulaV1 = assignVarToForumla(newAssignment, var, 1);
        if (DavisPutnamLogemannLoveland(formulaV1)==SATISFIABLE)
            return SATISFIABLE;
        else {
            var = TakeVariableFromFormula(newAssignment);
            formulaV0 = assignVarToFormula(newAssignment, var, 0);
            return DavisPutnamLogemannLoveland(formulaV0);
        }
    }
}
```

3.7 Causes of Complexity

The causes of complexity of the formal analysis depends on different factors such as size, diameter, other secondary factors such as design structure, state machine transitions, design operations, etc. The complexity of the module to be verified is directly reflected in the performance of the proof execution.

Some symptoms that we can observe and that warn us that the complexity of our Testbench is high are the following: long execution times exceeding several hours to complete an assertion, high memory consumption taking large amounts of memory or even running out of memory, and undetermined results (not finished runs for a specific assertion due to a timeout).

Therefore, reducing the complexity of our proofs translates directly into a reduction of execution time and memory consumption, or in case of undetermined proofs, converting them to proven or counterexample in case of assertion violation.

The size complexity of the proofs is not measured in terms of gates, as might be interpreted from their name, but in terms of primary inputs that have not been constrained to a specific value by assumptions, and the number of state bits that have not been set to a specific value. In addition, auxiliary code introduced by some complexity reduction techniques and various constraints can increase the number of state bits, becoming counterproductive.

That size complexity of the proofs is related to the size of the Cone Of Influence (COI). The size of the COI refers to the actual complexity of the assertion, i.e., it takes into account only the logic and inputs related to the assertion to be tested. In this way, a fraction of the size of the main design is achieved.

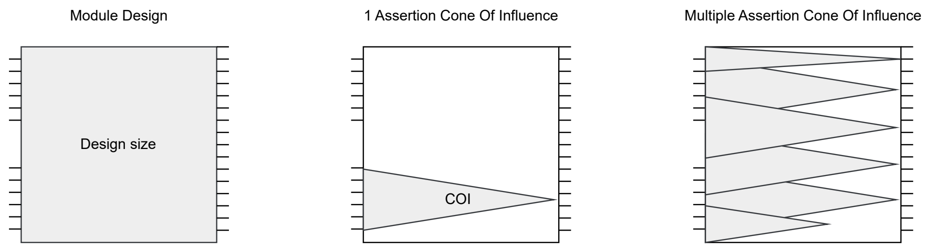

Figure 3.3: Design Size and Cone Of Influence (COI) Size

In Figure 3.3, the shadowed parts corresponds to the actual size of the assertions for each scenario. For the design size, we are taking into account all the possible behaviors at once, making harder the proofs for the tool and making more probable to obtain the worst performance. The design size could be seen as the worst case scenario of a proof. So, in a specific module, it will be the whole spatial state of the module. In general, FV will not obtain a complexity equal to the design size for an assertion. In the middle figure, we can see the COI of 1 assertion. As it can be seen, the complexity for this assertion is clearly lower than the Design size, improving the resolution of the assertion. The last figure represents multiple assertions for the same design. One advantage is that there are some regions inside the COIs are also included in other COIs, so what the tool has learned for other asserts can be used for the current assert.

Even when knowing the size of the COI, it is not possible to predict proof performance from size alone. But despite this, it is possible to correlate the probability of complexity with size. So small sizes are less likely to generate high complexities, and conversely, larger sizes are more likely to have higher complexities.

Another factor that is closely related to the complexity of the problem is the Diameter of the proofs. The diameter refers to the steps needed to cover all possible states from the initial state, so in this case, we also depend on the initial state and the complexity of these transitions.

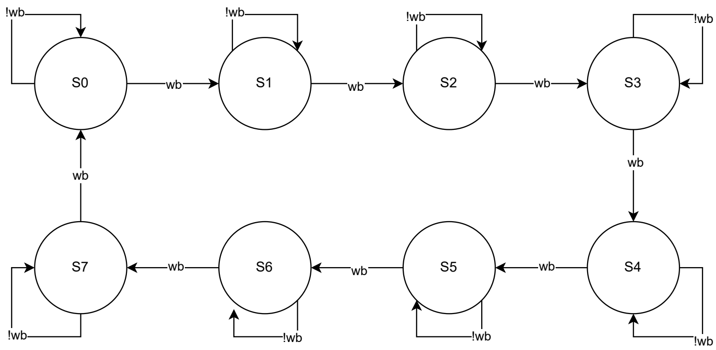

Figure 3.4: FSM of 3-bit Counter with Overflow

In Figure 3.4 we can see the Finite State Machine of a 3-bit counter with overflow. This counter only makes addition of 1 and cannot reduce the number except on the overflow. In this case, we can see that the counter will be adding 1 all the time it receives the signal *wb* high. Else, it will maintain its value.

In Table 3.1 there is an example of how diameter is calculated for the Finite State Machine (FSM) shown in 3.4. In this case, we would need 8 steps to fully cover the FSM. Depending on the possible transitions, the Diameter can vary.

Reached States	New Reachable States
S0	S1
S0, S1	S2
S0 ,S1, S2	S3
S0, S1, S2, S3	S4
S0, S1, S2, S3, S4	S5
S0, S1, S2, S3, S4, S5	S6
S0, S1, S2, S3, S4, S5, S6	S7
S0, S1, S2, S3, S4, S5, S6, S7	

Table 3.1: Diameter Calculation of Figure 3.4

Taking into account the FSM shown in Figure 3.5, we would obtain the Table 3.2. As we can see in the table, the longest path to any state is 4 steps, so the diameter for the FSM proposed in 3.5 is 4.

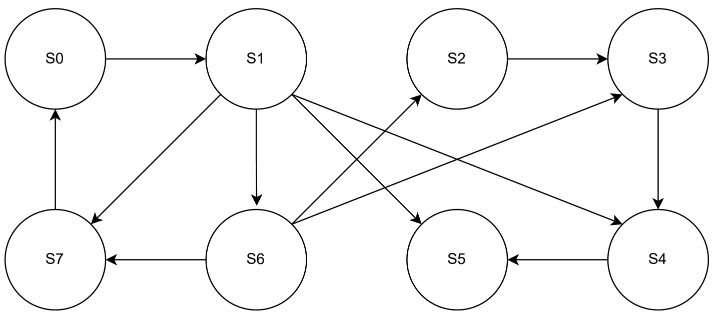

Figure 3.5: FSM Example

In the case of counters, we can see how with the number of states, we can obtain different complexities when it comes to formal verification. A counter in itself does not generate a problem for verification tools, even with diameters of more than 1 million states. If this counter interacts with another operation logic, it generates a significant increase in the complexity of the verification and can have multiplicative effects on the complexity. If the operation logic is simple, with few operations, the impact on the complexity will be slight, while if different operations are chained together, the problem could grow and generate complex problems for

Reached States	New Reachable States
S0	S1
S0, S1	S4, S5, S6, S7
S0 ,S1, S4, S5, S6, S7	S2, S3
S0, S1, S2, S3, S4, S5, S6, S7	

Table 3.2: Diameter Calculation of Figure 3.5

the tool.

Assertions can also increase complexity, increasing the size of the test design and the diameter. This is because assertions interfere with the state bits, increasing the possible states, and hence the diameter. In the case of constraints, we can obtain complexity reductions, but we can also increase the complexity of the test.

A clear case in which we can reduce the complexity of such a test with a constraint would be, for example, to set a certain value. As an example, for a General Purpose Processor (GPP) that is the design that we are verifying, a constraint that could reduce the complexity of some proofs could be to set the privilege level in which we are working. Setting the value, for example to only test with M-mode.

In the case of assertions, a clear case in which we would not add complexity to the proof would be one like the example shown in Listing 3.5. In this case, it is only a Boolean that has to be true in all clocks where sampling is performed. Therefore, no auxiliary code is introduced to our test indirectly, only a checker in each clock. Therefore, it is not necessary any knowledge of the previous states of our design, so the tool will not introduce variables with which it must test.

Listing 3.5 can be "read" as follows:

- *asrt_line_request_eventually_gets_response*: It is the name of the assertion to identify better on the used tool.
- *assert property*: It is specifying that the inner expression is an assertion.
- *!hit_translation |-> stall*: It specifies the sequence that must be proved.

In this case, we can translate it to a human-like language as "LOW (!) *hit_translation* IMPLIES in the SAME cycle (|->) HIGH *stall*".

Listing 3.5: Assertion that does not add complexity to the proof

```
asrt_line_request_eventually_gets_response: assert property (
    !hit_translation |-> stall
);
```

On the contrary, an example of assertions that would interfere with the State Bits introducing more complexity to the proof would be the one shown in Listing 3.6. In this case, we have implications in future cycles. This is translated internally as auxiliary code saving the old values in order to check that the implication is true, introducing more memory consumption to store past values until it is checked.

Listing 3.6: Assertion that add complexity to the proof

```
asrt_stall_in_itlb_miss: assert property (
    !hit_tag && requested_line |-> s_eventually(response_line)
);
```

Additionally, in Listing 3.6 we find the function *s_eventually*(). This function is used to indicate that at some moment the expression inside the function will occur. So this assertion means "LOW (!) *hit_tag* AND (&&) HIGH *requested_line* IMPLIES in the SAME cycle (|->) HIGH *response_line* will occur eventually.

The assertions that present functions or operators that interfere with different cycles represent an increment in the spatial complexity. This event occurs because we would have to maintain boolean expressions for many cycles, or even indefinitely. For example, repetition operators, the implication in the next cycle, cycle delays, or functions like $\$fell$ (expression changes from HIGH to LOW), $\$rose$ (expression changes from LOW to HIGH), $\$stable$ (expression remains the same), or $\$past$ (expression is HIGH in the past cycles) fit into this category.

4. Fetch Unit

The fetch unit is considered the first stage responsible for processing the instructions, and therefore, it is in charge of fetching and introducing the instructions to the following stages of the core [20]. For example, using the traditional von Neumann machine [21], the job of the Fetch Stage (or Fetch Unit) can be summarized as fetching the instructions from memory and entering into de pipeline (Figure 4.1). Translating the virtual address of the Program Counter (PC) for the next instruction, to the physical memory address where the desired instruction is located. Subsequently, this instruction would be entered into the core after receiving it from memory.

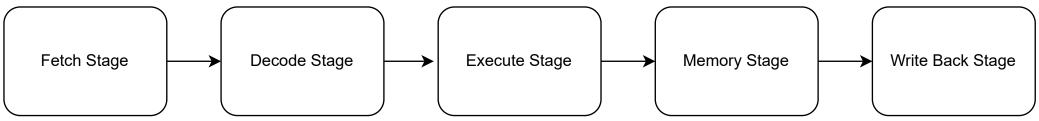

Figure 4.1: Stages of Von Neumann Machine

But the increasingly larger latencies from main memory with respect to Central Processor Unit (CPU) frequency and bandwidth requirement has a huge impact in the performance of the CPU [22]. Figure 4.2 shows the ratio of improvement of the performance for Memory and CPU. We can see how CPU frequency has been improving much faster than DRAM memory, creating a large performance gap.

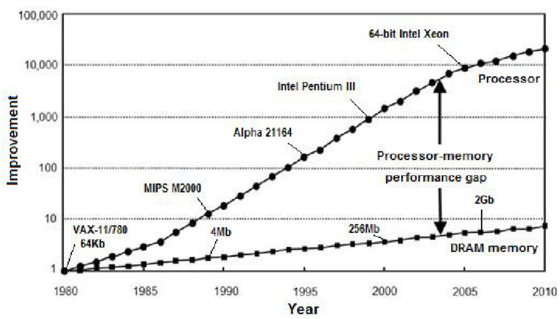

Figure 4.2: Yearly Improvement of DRAM and CPU [22]

In order to tackle this issue, cache memories were introduced. The instruction cache (iCache) is a low latency memory where we store main memory to reduce the read latency of the main memory. Based on how the codes are programmed, it is usual that the processors access repetitively the same addresses many times. This is known as temporal locality. In addition, processors usually access to memory position that are next to the previous one. For example, if the processor accesses to memory position A, it is very likely that the next instruction is going to be close to that position. From Fetch Unit view, if memory position A is accessed, it is an indicator that (possibly) the next access position will be A+($instruction\_size$). This is known as spatial locality. So, bringing lines of memory into the iCache reduces the memory access time. In this way, we can reduce the waiting time by storing in this iCache parts of the main memory, since the accesses to the cache are much faster than to the Dynamic Random Access Memory (DRAM).

The iCache memory in current commercial processors is typically divided into 32-byte or 64-byte lines, depending on the needs and characteristics of the core [23]. To speed up the process of fetching instructions, the fetch unit reads the memory line and inserts it into the pipeline, where it is later divided into the different instructions [24].

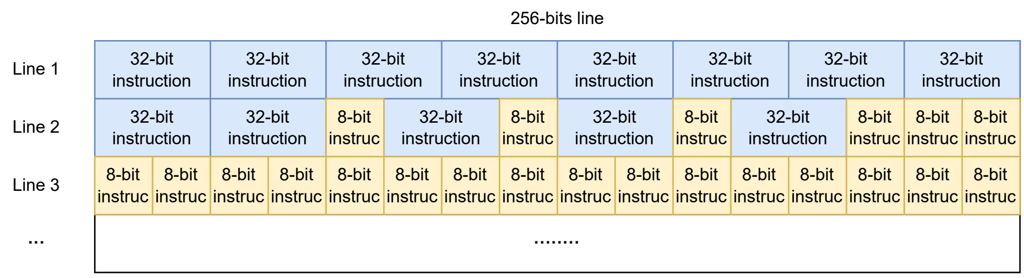

Figure 4.3: Instructions per iCache Line

Considering that the instructions, in the case of RISC-V, are 32-bit (or 4-bytes), we can fetch 8 instructions in case of 256-bit (32-bytes) memory lines. In addition, if the processor supports compressed instructions (16-bits) we could get to read 16 instructions. As not all memory lines have the same number of instructions, we can read between 8 and 16 instructions in each access.

In Figure 4.3, we have some possible configurations for the instructions in a iCache line of 256-bits. Line 1 correspond to a line with all its instructions of 32-bits (uncompressed) and we fetch 8 instructions. In the case of Line 2, we can see instructions of different size, compressed (16-bits) and uncompressed (32-bits), where depending on the combination of the instructions we can have more instructions (from 9 to 15 instructions). In this case, when fetching this line from iCache we would fetch 11 instructions. Line 3 represents a line fully of compressed instructions, where we would fetch 16 instructions, the maximum possible for 256-bit lines.

Since the programs are also composed of flow control instructions, such as branches or jumps, it is possible that the next instruction that the core has fetched is not the correct one. In this case, the instructions that have been introduced in the pipeline after the flow control instruction must be discarded, and the PC corresponding to the jump must be fetched. All this process induces a bubble that potentially lasts even tens of cycles plus memory latency [25] due to the pipeline depth of current processors. So a branch instruction generates a long period in which the core does not commit any instruction, reducing considerably its performance.

To avoid as many bubbles as possible due to a fetching error, a branch predictor is implemented together with the fetch unit, which is in charge of deciding if the PC of the instructions fetched from memory contains a branch, and if so, whether it is taken or not. It also decides which is the next PC to be fetched from memory. There are different types of Branch Predictors [26] depending on the characteristics of the core and the trade-off between performance and Branch Predictor implementation. Although we are not going to go deeper into this module in particular, because of its complexity, it could constitute a section of its own.

The Branch Predictor (BP) can affect the operation of the fetch unit in case of a mispredict. A mispredict happens when the BP incorrectly guesses the next address to fetch. In this case, a fetch bubble is again introduced in the pipeline, sometimes with a higher cost than if no BP were present. But even with the appearance of mispredicts, the amount of bubbles the core saves is much higher than the mispredicts it introduces. Considering a 2-bit local Branch Predictor, its hit rate is over 80% and in some cases can reach 99% depending on the program[27].

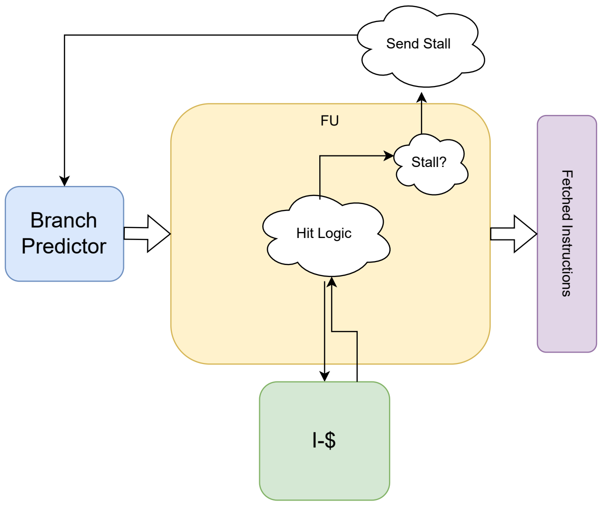

Figure 4.4: Main Fetch Unit Components

As of today, Fetch Units are composed of 3 main components, shown in Figure 4.4. The fetch logic, the iCache, and a branch predictor.

The flow of an instruction in the Fetch Unit would be as follows:

First, it is told the PC from which it will have to fetch the next instruction from the Branch Predictor (Figure 4.5). One of the first things it will do is to see if the memory line is already present in the iCache. When the memory line is present int the iCache is known as "cache hit". If so (Figure 4.6), this address would be taken directly from the iCache and introduced in the pipeline to later divide it into the different instructions that are in the line that we have brought.

In case the instruction is not in the cache (known as "cache miss"), it would be necessary to fetch this address from the main memory (Figure 4.7). As this

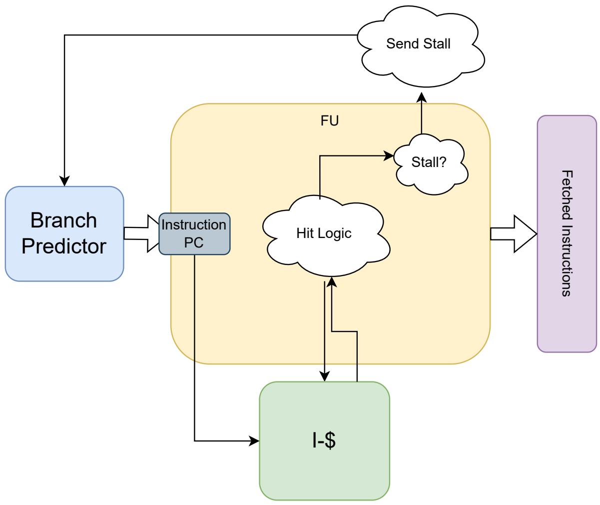

Figure 4.5: Branch Predictor Introduce New PC to the Fetch Unit

memory is much slower, it is not possible to obtain the result before the next cycle to enter it in the pipeline. Therefore, the Fetch Unit is stalled, which means that the Fetch stage will wait until the memory line arrives from main memory.

Once the block is fetched from memory and available, the memory line is introduced into the pipeline to divide it into the different instructions that compose it.

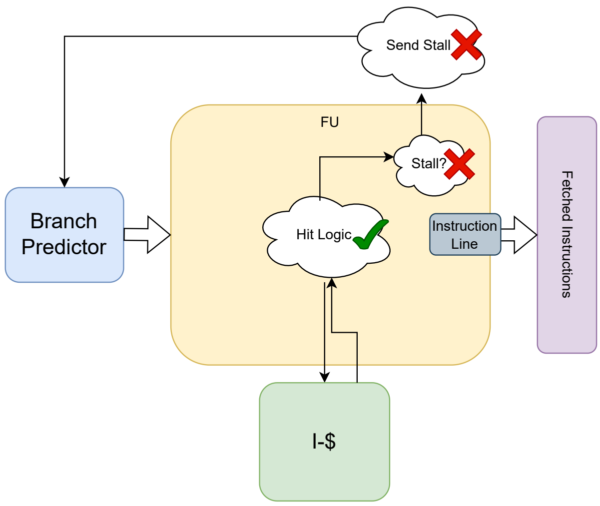

Figure 4.6: PC in iCache (hit)

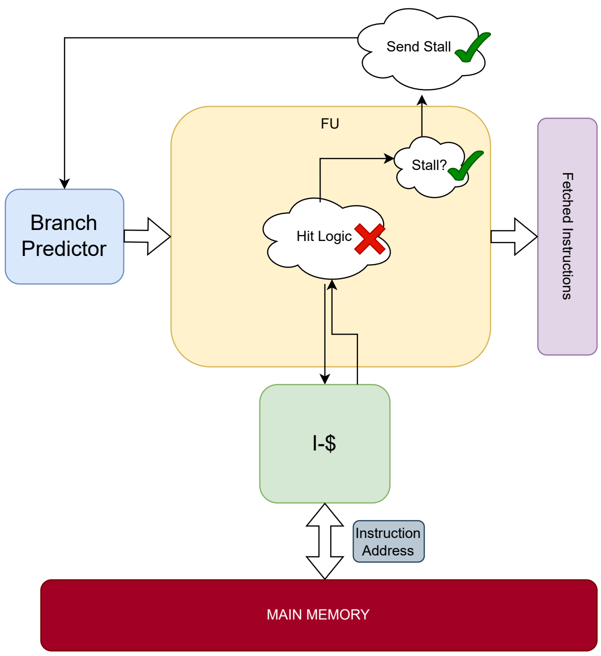

Figure 4.7: PC Not in iCache (miss)

5. Complexity Reduction Techniques

As mentioned throughout this document, memory is one of the most problematic aspects of Formal Verification tools. Therefore, different strategies have been developed in order to model them in different ways and achieve executions with a reduction of the spatial and temporal complexity.

In addition to modeling the memory, it is also possible to reduce the complexity of the assertions, making them more application-friendly. To do this, one technique used would be to reduce the scope of each assertion. For example, an assertion like the one in Listing 5.1.

The assertion shown in Listing 5.1 can be read as:

- *pc* not equal (!=) to any *cached_pc* IMPLIES in the NEXT cycle (|=>) *cache_hit* equal (==) 0
- AND (&&)
- *cache_hit* equal (==) 0 IMPLIES in the NEXT cycle (|=>) *stall*
- AND (&&)
- *stall* IMPLIES in the NEXT cycle (|=>) LAST CYCLE value of *pc* (*$past(pc)*) equal (==) to current cycle *pc*

That assertion represents a sequence of 4 clocks. The sequence is as follows: if the PC in the cache does not match with the PC that has been filled by the Branch Predictor (Clk 1) we expect that we are not going to obtain a hit in cache (Clk 2), in other words we got a miss (cache_hit equal to 0). Then, when we obtain a cache miss (Clk 2), we expect that the Fetch Unit is stalled as the memory line in not present (Clk 3). As the Fetch Unit has not been capable of obtaining the line, and we are stalled (Clk 3), we expect that the current value of the PC the in the next cycle (Clk 4) is the same that has been inserted in the current cycle.

Taking into account this assertion, we can see that the assertion is concatenating different cycles (up to 4 cycles). This assert will need to check all spatial state for the 4 cycles to comply with the assertion. Increasing the spatial state of 1 cycle to the power of 4.

Listing 5.1: Not application-friendly assertion

```
asrt_no_hit_stall: assert property (
    pc != cached_pc |=> cache_hit == 0
    &&
    cache_hit == 0 |=> stall
    &&
    stall |=> $past(pc) == pc
);
```

It can be implemented separately, as shown in Listing 5.2. Instead of having a sequence of 4 clocks together, we can separate the assert using the ANDs (&&) operators.

Listing 5.2: Application-friendly assertion

```
asrt_pc_not_in_addresses_hit_value_zero: assert property (
    pc != cache_addresses |=> hit_vector == 0
);

asrt_no_hit_stall: assert property (
    hit_vector == 0 |=> stall
);


asrt_stall_maintain_pc: assert property (
    stall |=> $past(pc) == pc
);
```

This simple change allows the application to only take into account the values of 2 cycles from Figure 5.2 (those implied by the |=> operator) and not the 4 cycles from Figure 5.1 that it would need for Listing 5.1. Extrapolating this case to larger assertions and with a larger number of variables, we can save the application the use of resources unnecessarily and improve its performance to solve the different proofs.

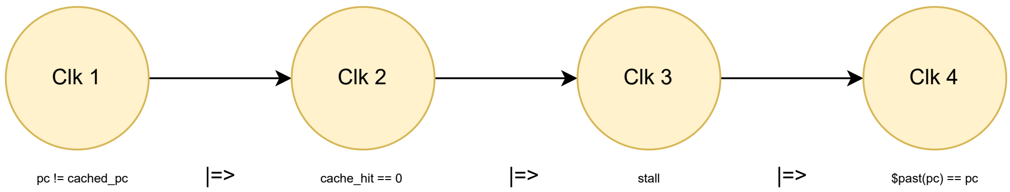

Figure 5.1: Not Application-Friendly Assertion Space

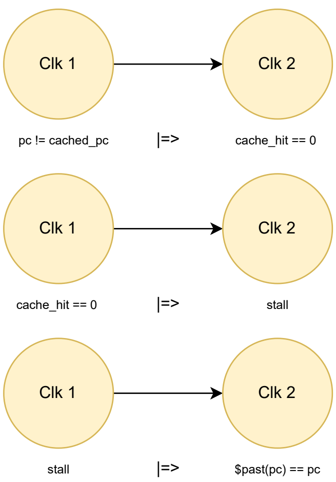

Figure 5.2: Application-Friendly Assertion Space

In Figure 5.1 we can see that the Tool will explore the space for 4 clocks. Taking into account the exponential complexity in terms of space, it will generate a total space of $clock\_complexity\hat{\ }4$ meanwhile Figure 5.2 will be only $3 \cdot clock\_complexity\hat{\ }2$.

5.1 Non-determinism

This technique is highly strongly linked to the used formal verification application. This technique is basically a way to present a special kind of asserts to make the assertion easier for the formal verification tool, improving the performance of the assertion to test.

Non-determinism is used for asserts inside a `for(genvar)` (Listing 5.3) functions that generates many asserts with a series of values. It is used to specify array index or positions in an assertion. What this technique does is to assume the index as a stable value in the trace and prove it in a single assertion.

Firstly, we should assume a stable value to test (a value that is not going to be modified on the trace), and use it for the assertions we want to test. It might seem that this technique skips cases to be tested, but the application will test the assertions for all values of the variable that is assumed as stable. Only that during the assertion, the variable will remain stable and there will be no changes in its value, but it will try it with all the combinations of the variable.

For example, an assertion like the one shown in Listing 5.3 is harder and slower for the Tool, so we can improve the performance of the application by converting it to the one shown in Listing 5.4.

Listing 5.3 assertions can be read as:

- for each **i** lower than NUM_WAYS
  - assert number **i**
  - *p_address* equal to (==) *stored_address[**i**]* IMPLIES in the SAME cycle (|->) *data* equal to (==) *data[**i**]*

Listing 5.3: Normal assertion

```
for (genvar i = 0; i < NUM_WAYS; i++) begin
    asrt_hit_data : assert property (
        p_address == stored_address[i] |-> data == data[i]
    );
end
```

Listing 5.4 assertion expresses the non-determinism that the suits better the application. It expresses:

- Define 2-bit variable *way*.
- Assume that variable *used_ way* takes the stable value of *way* (*$stable(way)*).
- Assert *p_ address* equal to (==) *stored_ address[**used_ way**]* IMPLIES in the SAME cycle (|->) *data* equal to (==) *data[**used_ way**]*

Listing 5.4: Free-variable assertion

```
bit [1:0] way;

asm_stable_way: assume property (
    used_way = $stable(way);
);

asrt_hit_data: assert property (
    p_address == stored_address[used_way] |-> data == data[used_way]
);
```

5.2 Data tagging

Data tagging can be highly related to Abstraction technique (Section 5.5), as Data Tagging can be used to accomplish the target of the Abstraction technique. Data tagging consists on using different inputs or specific bits to assume or take decisions in advance.

Data tagging technique allows us to recognize certain values in the output and check that indeed, a series of inputs is correctly identified or whether the result is propagated as expected from this tag. In addition, this technique can be used to make decisions at the beginning of the test, such as discarding inputs from a specific tag, deciding what will happen to the inputs during the transaction or assuming certain behaviors from that value.

One use of this optimization is, for example, to decide whether there is a hit or miss in a cache, eliminating much of the complexity that this memory brings. As an example, we can assume that as long as the line is valid and ready, the hit will always be equal to bit 9 of the memory address to be requested, as shown in Listing 5.5.

Listing 5.5 has an assumption that, in case of *valid_line* and *ready_line* will determine the output of *cache_hit* equal to the bit 9 of the physical address we are introducing. Then, we check with the assertion that in case of a NOT (!) *cache_hit* and a *req_valid* IMPLIES (|->) that in a range between 10 and 15 cycles (##[10:15]) we are receiving a *response_line* and the *cache_hit* becomes HIGH.

Listing 5.5: Data-tagging technique in assume/assert

```
asm_cache_hit: assume property (
    iCache.valid_line && iCache.ready_line
    |->
    cache_hit == p_address[9]
);

asrt_req_line: assert property (
    req_valid && !cache_hit
    |->
    ##[10:15] (response_line && cache_hit)
);
```

In this way, we can speed up the testing of certain modules of our proof or modules that connect to our DUT, modeling the module's behavior making use of a simplification of the module's behavior through assumptions and assertions that allow us to reduce dependencies on the complete behavior of the module from which we would expect the answer, like in Figure 5.3.

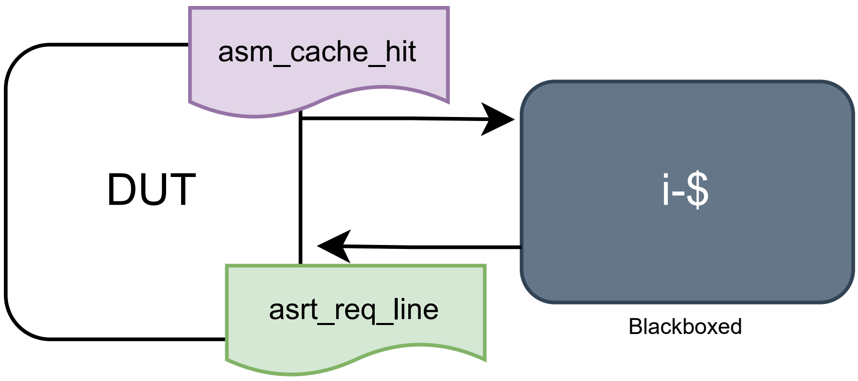

Figure 5.3: Datatag on a Testbench

5.3 Reduction

The main idea of reduction techniques is to replace different parts of more complex logic with others that accomplish the same task using simpler logic, while remaining equivalent. We can find different techniques that reduce the complexity of the Testbench by acting on different points and which require more or less manual interaction to find the different patterns or to simplify the logic. We can classify the techniques based on the manual interaction of the engineer, dividing them into automatic and manual categories.

Automatic reductions, which are those performed by the applications themselves to reduce as much as possible the test states before performing the Formal Verification. Among them, we find Boolean minimization, Constant propagation, FSM minimization, and Fanin Cone Reduction.

Manual techniques, which are capable of greatly reducing the state space. These techniques require a great effort to find structures that can be reduced within each Testbench. The techniques we can find under this classification are Symmetry reductions, Abstraction and Homomorphic reductions.

We could define all the proposed techniques under this nomenclature as they meet the definition of this technique. Reduction is a validity-preserving transformation by which a complex assertion of a complex model can be exchanged for a simpler assertion of a simpler logical model. So that the validity of the second assertion implies the validity of the original assertion [28].

5.3.1 Symmetry reductions

This technique [29] is based on finding states of the DUT that are equivalent and replacing them so that the application uses the equivalent states as one and the same, without the need to test all of them. For example, within a First In First Out (FIFO) of 8 entries (9 states shown in Figure 5.4), we can separate 3 states with different behavior, thus grouping 7 states into one. The states would be: empty, semi-full, and full.

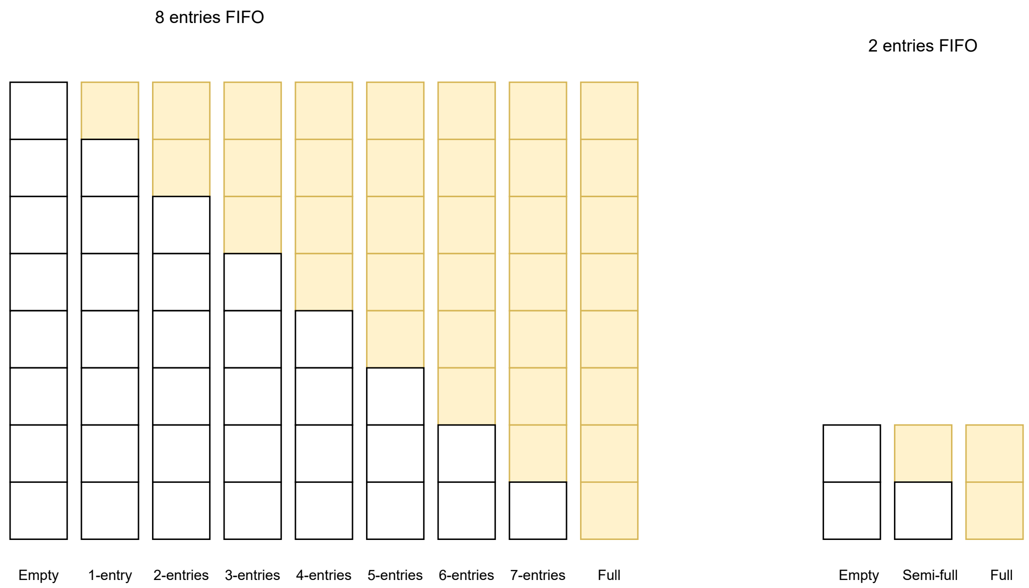

Figure 5.4: Symmetry Reduction of a FIFO

By observing the symmetry of states 1-7 of the counter, we can force the application to only test the states encoded by 2-bits (enough to encode 3 states) and instruct the FV Tool to take into account the 3 states without the need to check all the intermediate states.

Figure 5.4 shows the difference between the two implementations and the reduction of possible states. This modification also simplifies the transitions between states for the FV Tool by reducing the combinations of states within the queue and possible paths within the queue.

5.3.2 Homomorphic reductions

This technique is based on simplifying the logic while preserving the effects on the system being tested [28].

In this way, we could reduce the FIFO from Figure 5.4 to 2 entries in order to accommodate the 3 states that behave differently. When it is empty it cannot remove anything from the queue, when it is full, it cannot introduce anything new into the queue.

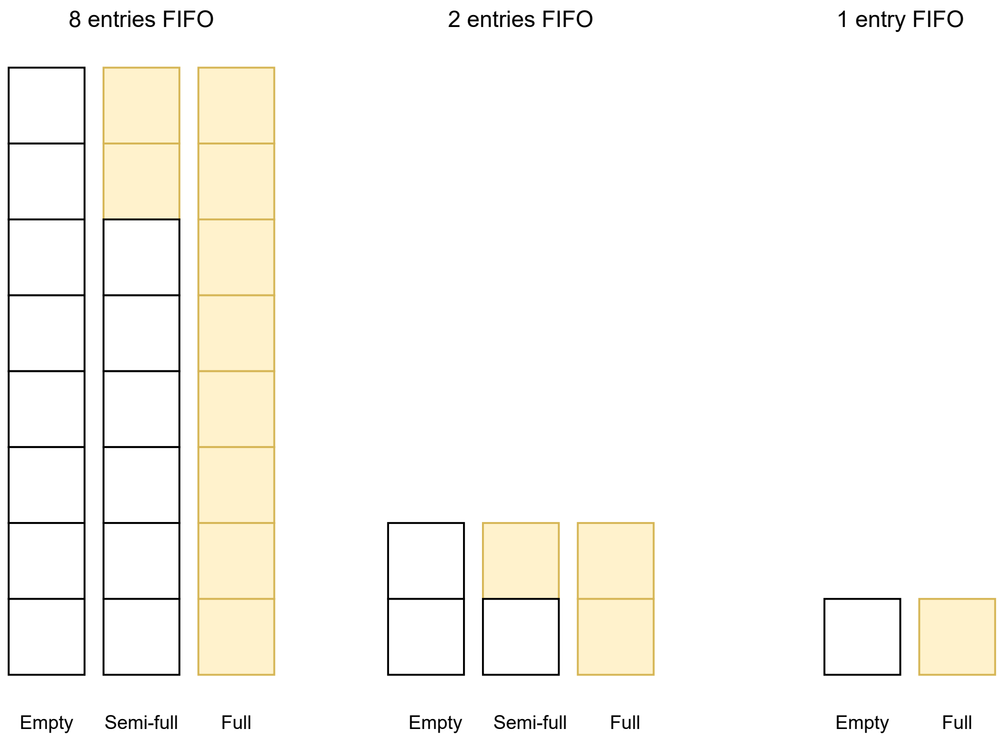

Figure 5.5: Homomorphic Reduction of a FIFO

If we had a counter to know if the queue is full or empty, states 1-7 would have the same behavior. In this way, we know that we can reduce the FIFO to only 2 entries, (3 states) and be able to test all states without losing any state functionality.

We could even further reduce the behavior to only 1 entry, so we would only have empty and full states. With this change we can still test the functionalities we are interested in from the point of view of a FIFO, input values, output values, see the behavior when full and when empty.

In Figure 5.5, we can check the reduction of the entries of a FIFO, being a graphical support of the previous explanation.

Compared with symmetric reduction it is possible that both seem similar. But there is a subtle difference, in symmetric reduction we look for equivalent states, and we try to reduce the repetitions of those states (i.e., we enforce the 3 states of the FIFO explicitly). While in homomorphic reduction, we are looking to reduce the complexity by reducing the space size (i.e., the size of the FIFO).

5.4 Black Box

This technique allows us to remove the complete logic of a module from the Testbench and replace it with a formal model of the module. This model must contain all the necessary assumptions to behave as close to the original as possible.

Performing all the necessary assumptions is a costly task that involves knowing and addressing many important aspects of the operation of the module on which you want to BlackBox. But since it allows the application to get rid of a large part of the states that would compose such a module and in case of a memory, it is super efficient.

A very appropriate use case for this technique is when the formal verification of the module we want to BlackBox has already been performed. This is because the collection of assertions that have been created for this module are themselves a formal model of the module. So we can change all the assertions that compose this model by assumptions and get the expected behavior. This allows us to reuse the effort previously made to improve the performance of the new Testbench.

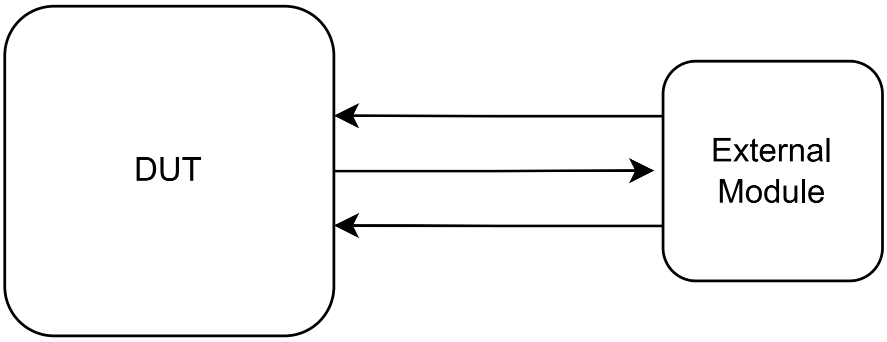

Figure 5.6: Testbench Configuration with One Connection to External Module

For example, in Figure 5.6 we have a Testbench in which the DUT that we want to verify has a connection to an external module. Since the behavior of our DUT is linked to the behavior of a second external module to which it is connected, we want to test its behavior with the responses of that module.

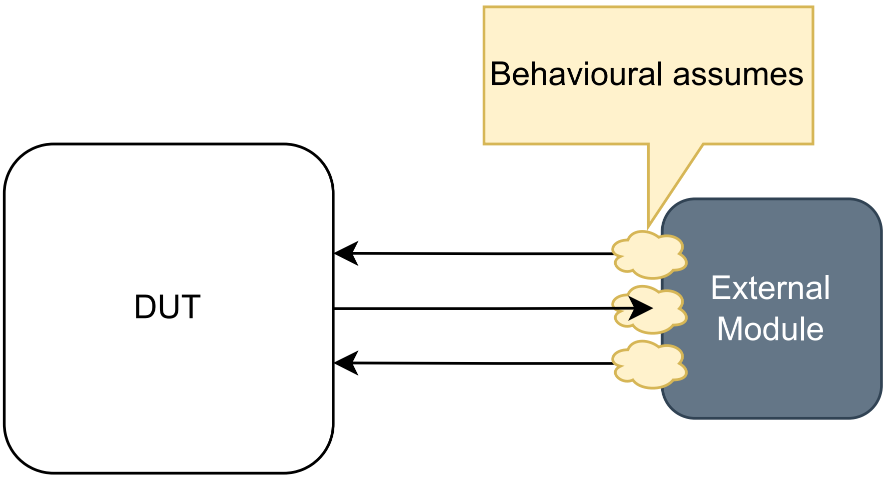

Figure 5.7: Testbench Configuration with Blackboxed External Module

In Figure 5.7, we have the conceptual way of how the Black Box technique works. We have a series of assumptions that act as a formal model and will help the tool to avoid the states introduced by the external module.

5.5 Abstraction

Abstraction is an optimization technique that allows to improve the performance of proof results by means of auxiliary code that speeds up obtaining results through a more straightforward behavior.

This technique represents large arrays in a much smaller way, for example, by reducing the width of a 64-bit band with a smaller symbolic width (e.g., 4-bit) that maintains its validity and simplifying the behavior of the module logic.

This simplification can be done in several ways. One option is to abstract the behavior of the module collapsing neighboring states to shorten the path to the trigger value, maintaining the triggering logic. An example of this case would be a signal that is activated when a counter reaches a certain value in which, instead of using a generic counter that increments its value one by one, the way in which this value is obtained can be modified by incrementing a higher value. In this way, we reduce the state space, since the counter would not go through all its values, but only through some of them. Using this abstraction, we do not modify the signal jump condition, and we are only shortening the path to the change of value. In Figure 5.8, we can see the difference between the counter without abstraction and the abstracted counter, and how the difference between the two is the obtaining of the value 50, that we are going to suppose that triggers a certain action.

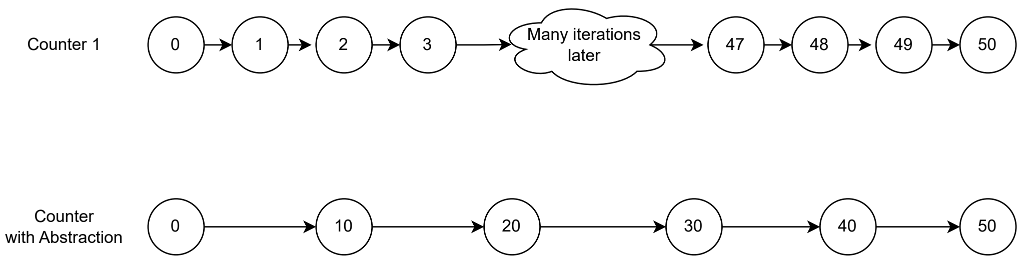

Figure 5.8: Abstraction of a Counter

Another way to apply an abstraction would be to simplify the behavior of the module. For example, replace non-deterministic behavior by a deterministic behavior beforehand or by a non-deterministic but simpler part. It could be applied for modules that take a variable time to respond (as could be the cache memory) to deterministically, leave prepared a series of values and an arbitrary behavior that will always comply with a reduced timing, as could be hits in the same cycle with a fixed value generated or in case of a miss, receive the memory line to the next cycle arbitrarily without having to wait for main memory. Usually, actual memory can respond in hundreds of cycles. The proposed abstraction reduces the waiting cycles of the memory response more than x100 less.

In addition to the reduction of the response time in both cases, we are drastically reducing the space state of the test. In the case of the cache abstraction, we are reducing the logic, latches and flops that the application would have to verify for this module, it has been simplified to the minimum, reducing uncertainty and cycles that would have to store the results.

6. Applying and Optimizing Formal Verification Techniques

In this section we are going to explain how we have introduced the above techniques to verify a module (the Fetch Unit). Firstly, we have performed a Testbench with a series of assertions, assumptions, properties and coverpoints to completely verify the Fetch Unit. The Fetch Unit is composed of the hit logic, the iTLB, iCache which is divided into iTags and iData, stall logic and flush logic.

To simplify the diagrams, we will differentiate within the Fetch Unit 3 blocks: logic, which addresses all the necessary operations for the functioning of the Fetch Unit; iTLB and its connection to the main memory; and finally the iTags. In Figure 6.1 we can see how this differentiation would look like, and we will use it to illustrate the different improvements and configurations in each version.

6.1 Naive Configuration

The Naive version is the first Testbench configuration. It does not use any manual technique for improving the performance. The Testbench we are going to evaluate consists of 263 assertions and coverpoints.

The Testbench is composed of different assertions that check different aspects from the Fetch Unit. All of them are related to the Fetch Unit, although some proofs are combinational closer to iTLB, or iCache, or both, or simply some aspects independent to both of them. The Testbench has a wide-range purpose assertions, and the impact that the different techniques can have to the execution time is not necessary equivalent in all the proofs. That wide-range purpose assertions end up in proofs that are has better results when they are executed with one technique or another.

The configuration characteristics of the memory modules in the Fetch Unit

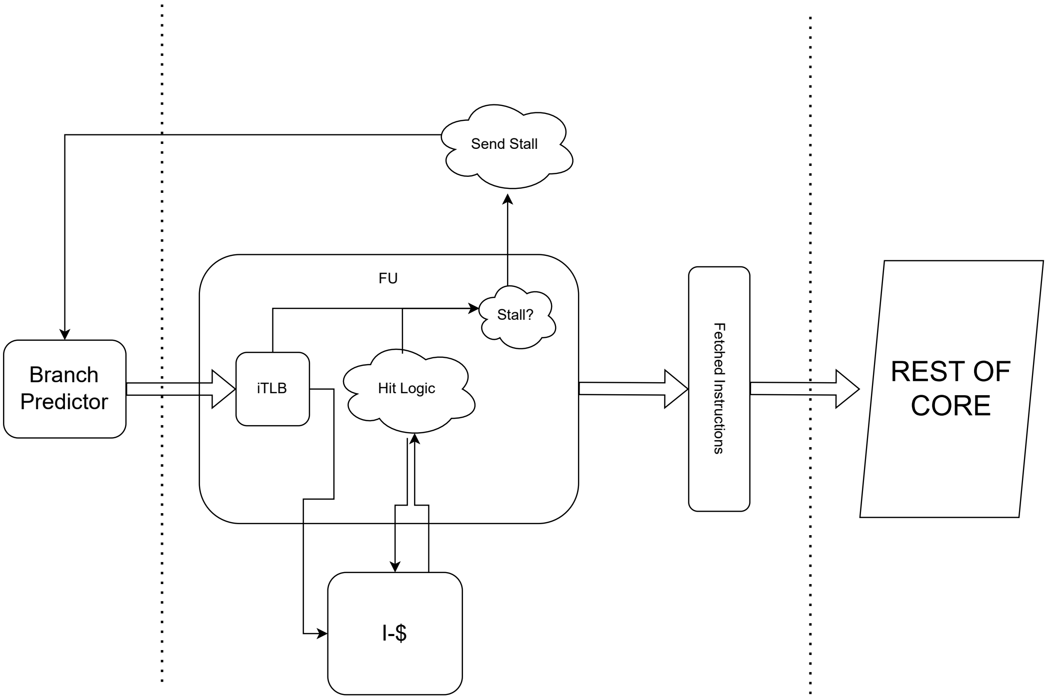

Figure 6.1: Fetch Unit Diagram

(iTLB and iTags) are as follows:

- iTLB:
  - 4 ways
  - 8 sets
  - 36-bit tag + 36-bit translation
  - Total memory bits: 2,304-bits $((Tags + Translation) * ways * sets)$
- iTags:
  - 36-bits size tags
  - 4 ways
  - 64 sets
  - Total memory bits: 9,216 bits $(Tags * sets * ways)$

Figure 4.4 represents the current state of the configuration (Naive). As it is naive, there is no technique applied and the "configuration diagram" has no modifications.

6.2 Reduction Configuration

In this version, we have applied the Reduction technique explained in section 5.3. For this, we have modified the compilation values of different parameters to reduce the size of the memory, operations and different counters trying to reduce the complexity introduced by them.

The main changes we found are:

- iTLB:
  - 2 ways
  - 4 sets
  - 36-bit address + 36-bit translation
  - Total memory bits: 576-bits $((Address + Translation) * ways * sets)$
- iTags:
  - 18-bits size tags
  - 2 ways
  - 32 sets
  - Total memory bits: 1,152-bits $(Tags * sets * ways)$

In this way, only by halving some parameters we can reduce the spatial complexity of our DUT. Just taking into account the values of the iTag reduction, we have reduced x8 the bits to be verified, thus reducing the possible states that the application has to test. Taking into account the number of bits that we are saving, we have reduced from about 11500-bits to approximately 1700-bits. Obtaining a

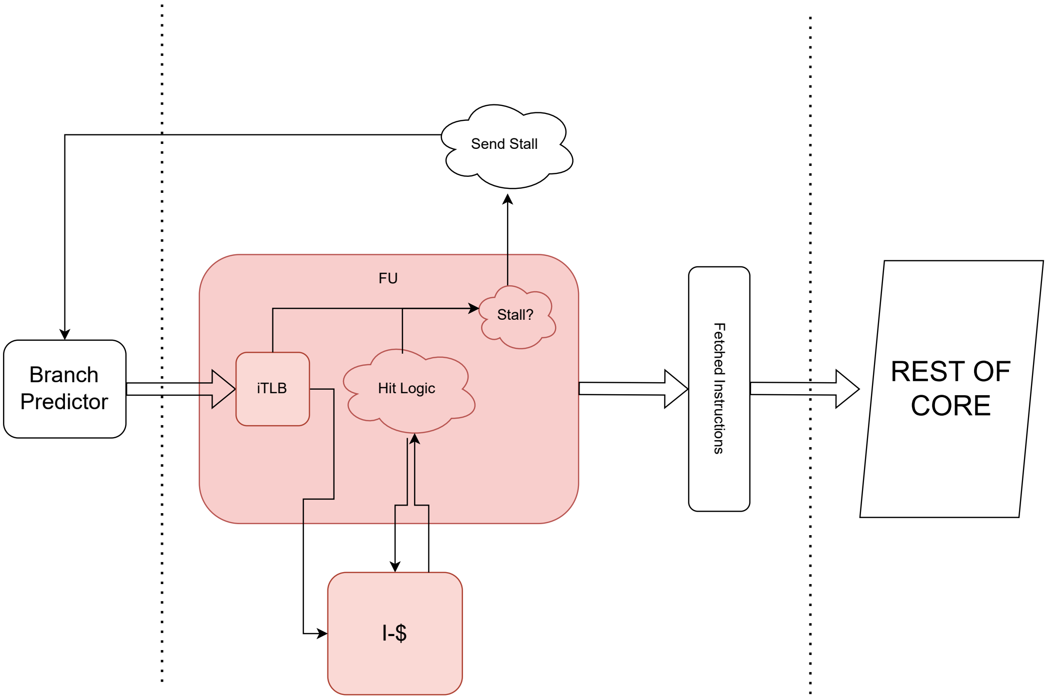

Figure 6.2: Reduction Fetch Unit Diagram

reduction of 9,800-bits, which means 2^{9800} ($1.24 \cdot 10^{2950}$) times fewer states. In this way, we are not losing functionalities to test, and we reduce the space of possible states that the application will have to check.

Figure 6.2 represents which are the modules that are affected by the reduction of the parameters that we have changed for this configuration. As we can see, the reduced modules present light red color. The modules that are affected are the Fetch Unit, the hit logic, the stall logic, the iTLB and iCache.

6.3 Black Box Configuration

This configuration makes use of the BlackBox technique. Given the complexity involved in obtaining a formal model of the modules, we have only blackboxed the iTLB module and reused the parameter reductions shown in section 6.2.

To introduce BlackBox into the iTLB has been necessary to generate a formal model making use of a series of assumes. Those assumes copy the behaviour of the iTLB.

The main objective of the iTLB is to obtain a translation for a given Tag. That tag can be already translated and stored in the iTLB, or it is possible to not have it, and we would need to stall the stage and ask for a translation.

The iTLB model consist of a series of assumes that allows the FV Tool to model the behavior correctly. We have modeled based on a series of requirements that have been obtained after analyze the code.

Examples of the behavior we have to model to BlackBox the iTLB are:

- iTLB can only have 1 hit
- In case of a hit, the response contains the right data
- Tags updates
- Tags hits

All those requirements have been traduced into assumes to behave as formal model and to be able to BlackBox the iTLB and remove all the logic in from the DUT. Allowing to completely reduce the memory bits related with the iTLB.

One example of the assumes introduced for the modelling of the iTLB is the shown in Listing 6.1. That assume models the part of the behavior of the iTLB. When it is accessed to the iTLB (*access* is high), the signal *hit* can only have 1 as much. That signal has as many positions as entries has the iTLB. So, the iTLB will report 0 or 1 hits.

Listing 6.1: iTLB Can Only Have 1 Hit Assume

```
assume -name tlb_one_hit_only {
    itlb.access
    |->
    $countones(itlb.hit) <= 1
}
```

By using this technique, we are reducing even more the number of bits (and therefore the number of possible states to verify compared with reduction technique) by about 2,304 in memory bits only (all iTLB memory bits shown in Section 6.1), which means a saving of $2^{2,304}$ $(3,74 \cdot 10^{693})$ states, without counting the different logic necessary to obtain hits, updates, and different functionalities. This means a considerable saving of resources and an important improvement of the performance of the test.

Figure 6.3 represents the techniques that are applied in this configuration. As we mentioned, the reduction in the parameters are applied again to reduce the time. In this configuration, the iTLB passes to be dark gray color representing the black-boxing, while reduced modules are red.

6.4 iCache Abstraction Configuration

This configuration uses the Abstraction technique. In this case, we have introduced abstraction to one module: iCache. We are maintaining the reduction of the rest of the logic. We are reducing the necessary state bits for iCache that stores big amounts of memory.

The iCache has been Abstracted using an auxiliary code. This code is a simplification of the behavior of the iCache, reducing the complexity introduced by the large memory included in the module.

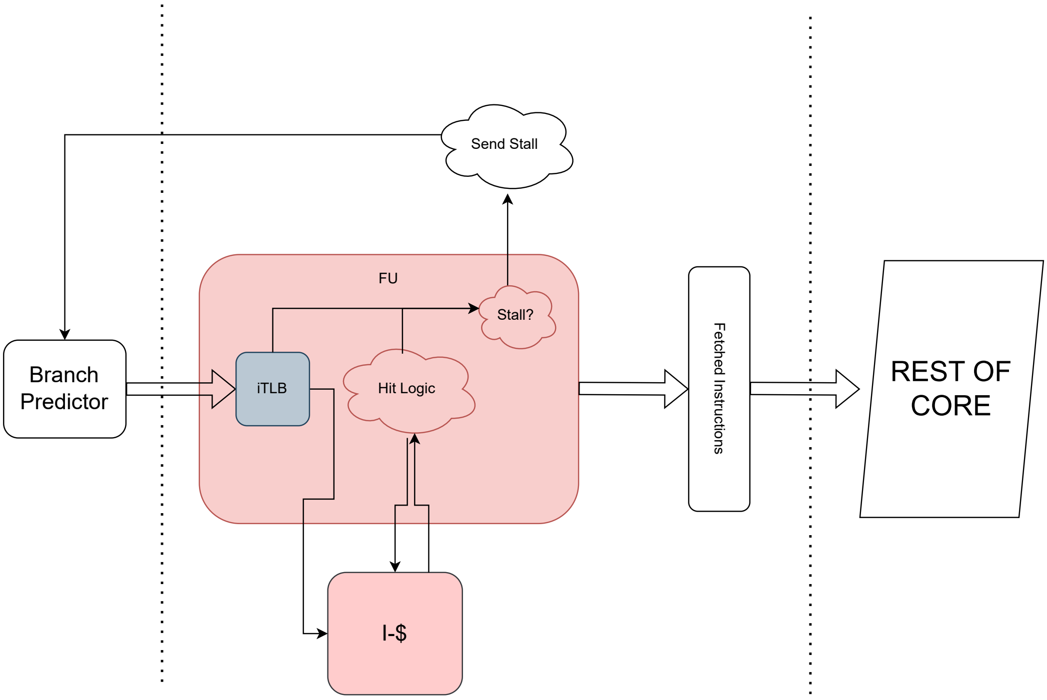

Figure 6.3: BlackBox Fetch Unit Diagram

The auxiliary code is the responsible for generating the expected output for the iCache. The code is capable of: return hits from a requested address, update the active addresses in same cycle, return data in same cycle. The logic for the memory has been reduced to the minimal expression to simplify the behavior, although they are not faithful to the original function of the module when it is connected to the rest of the core.

The logic of this module is not particularly complex, but it is the one with the most amount of memory. So in addition, we have maintained the reduction in the parameters in order to simplify the transitions between the possible states, and reducing the number of possible states.

To perform correctly the Abstraction, we need to configure correctly the modules in the setup for the FV Tool. For that, we need to BlackBox the iCache module, but it is not necessary to create complex assumptions with the behavior of the module. Instead, we need to bind the connections of the module that is abstracted with the auxiliary code create for the abstraction. In addition, to make the Tool to use the abstraction, it is necessary to create a series of assumes to connect the signals of the DUT with the signals of the Abstraction.

In this case, the assumes become as easy as $dut\_signal == abstracted\_singal$. The effort to create those assumptions is infinitely inferior to create the correct assumes for a BlackBoxed Technique in a module.

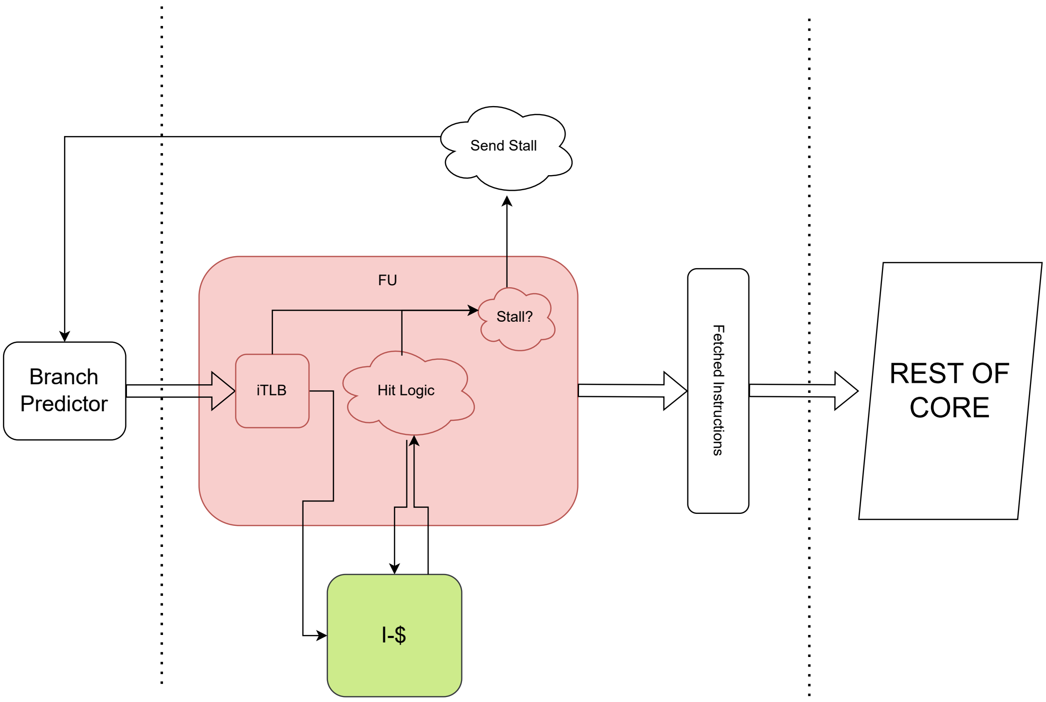

Figure 6.4: iCache Abstraction Fetch Unit Diagram

Figure 6.5 represents where we have applied Abstraction. Green represents the modules that we have abstracted. As we can see, iTLB and iCache have been abstracted, while the Fetch Unit remains reduced.

6.5 iCache + iTLB Abstraction Configuration

This configuration uses the Abstraction technique. In this case, we have introduced abstraction to two modules: iTLB and iCache. We are maintaining the reduction of the rest of the logic. Both modules has been abstracted to reduce the expected delay to obtain a response and reducing the amount of memory that it stores. We are reducing the necessary memory and implied states for those 2 modules.

The Abstraction auxiliary code that was created for the previous configuration (iCache abstraction) has been used in this module too. The iTLB has been Abstracted maintaining the expected behaviour from the point of view of the Fetch Unit. The abstraction is capable of returning translations in the same cycle in case of a hit, and in case of a miss, the translation is performed in the next cycle. Only taking into account the reduction in the number of cycles of the translation suppose a big impact on complexity and necessary clocks to perform the operation.

In addition, the abstraction frees all latches. The hit/miss is reported making use of Data Tagging (Listing 6.2). The abstracted iTLB decides if there is a hit or a miss based on a series of bits in the Virtual Address passed to the iTLB. If those bits are equal to that specified series, the abstraction gives a hit and a predefined arbitrary address.

Listing 6.2: iTLB Abstraction Auxiliary Code

```
//Data Tagging in Abstraction for hit/miss decision
always @(posedge clk) begin
   if(~rst_ni) begin
      if(access && virtual_address[47:40] == 8’b10101011) begin
         hit_leaf = 1’b1;
         physical_addr = 36’h4fffffff;
      end else begin
         hit_leaf = 1’b0;
         physical_addr = ’0;
```

```
        end
    end else begin
        hit_leaf = 1'b0;
    end
end
```

This abstraction removes all memory bits of the iTLB and the behavior of the module instead of being a complex behavior we have modified to a simplistic deterministic behavior. Those changes are translated into a reduction of the possible states and a simplification of the transitions between those states.

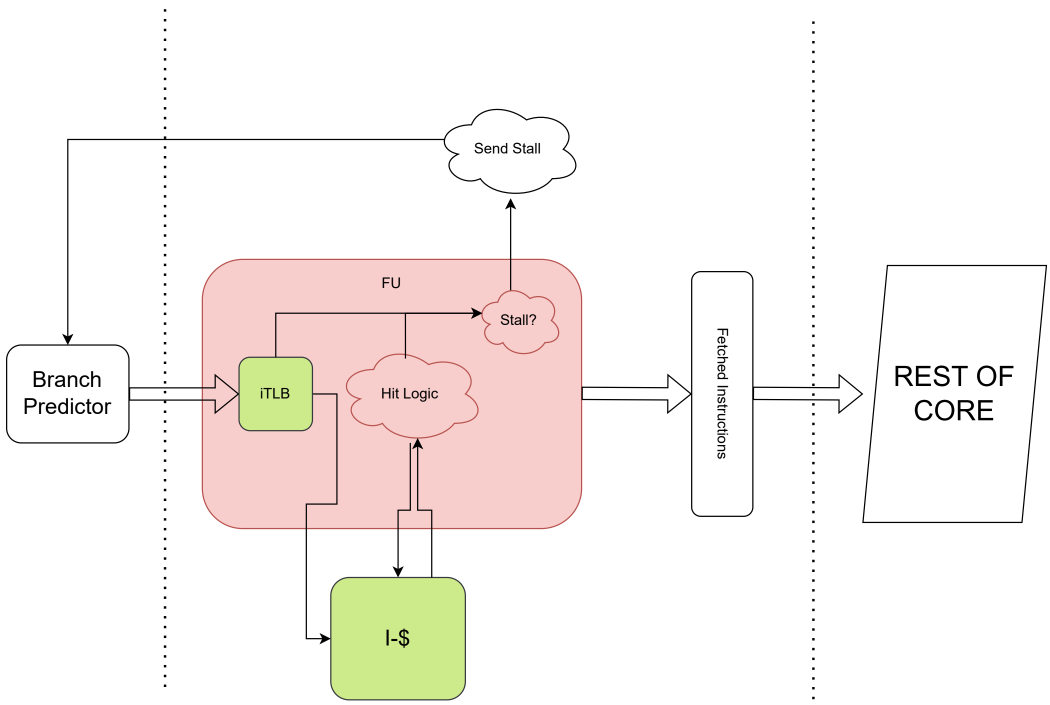

Figure 6.5: iCache + iTLB Abstraction Fetch Unit Diagram

As was mentioned in the Section 6.4, to make the abstraction work in the Fetch Unit Formal Testbench, the iTLB has been blackboxed, we have binded the connections of the module to the abstraction, and create assumes to connect DUT signals with the signals from the iTLB abstraction.

Figure 6.5 represents where we have applied Abstraction. Green represents the modules that we have abstracted. As we can see, iTLB and iCache have been abstracted, while the Fetch Unit remains reduced.

7. Results

The results shown in this section have been obtained using the same machine and the same application version. In this way, we ensure that there are no significant differences in terms of system architecture performance or possible improvements due to application updates.

7.1 Machine (Spargi)

The machine we have used to run the application and the different Testbench configurations explained in Section 6, has the following specifications:

- CPU Name: AMD EPYC 7502p32-Core Processor @3270MHz
- Cores per socket: 32
- Threads per core: 2
- Total CPUs: 64
- L1d cache: 32K
- L1i cache: 32K
- L2 cache: 512K
- L3 cache: 16384K
- 16 DIMMS x 64GB
- RAM: 1TB DDR4
- Speed: 3200 MT/s

The machine is used by the application through SLURM, creating the necessary jobs to execute the proofs with the configurations specified in the test setup.

In addition, the machine has been used only for FV Tool testing, with a single license (one proof/configuration) to avoid possible noise introduced by the interaction of the machine with other processes. Avoiding as much as possible, loads from other applications, since the machine has been reserved only for FV Tool proofs.

7.2 Commercial FV Tool

The Formal Verification application used for this evaluation is a widely-used commercial Formal Verification Tool in Electronic Design Automation. This application is capable of running different threads at the same time. Therefore, we are going to configure the program so that it always runs the same number of jobs in parallel in order to have equal results between the different techniques applied.

The application consists of different Engines, which we could extrapolate to different heuristics, each one suited better or worse to particular types of assertions/properties/proofs. Since the tool does not know in advance which engine will work best for a given proof, it will run several Engines for the same proof in parallel and keep the result and time used of the engine that obtains the final result for the specific assertion.

The application works through licenses, which allow you to use up to 30 engines (jobs) in parallel and allows you to configure the number of licenses to use for the Testbench. In this way, we could use several licenses for the same proof and use more engines in parallel, if necessary.

This behavior of the application generates differences between the total execution time of the proofs, with the sum of the individual times of each assertion. As an example, for a proof for which four engines have been launched, three of them have been running for 3600s without stopping and without producing a result, while one of the engines has started later than the previous three, running for 600s and producing that the assertion to be tested is in "proven" status. For this specific case, the application will report that the time to complete the proof has been 600s instead of the 3600s that the tool spent executing the other three engines.

7.2.1 FV Tool Configuration

After some initial tests with the Naive version, trying to obtain reasonable times that would allow us to perform the different tests with the techniques already presented, we decided to take all the tests with the following configuration:

- 1 license
- 30 jobs max
- Bound of 75 cycles.

A bound refers to a number of cycles for which we consider an assertion to be true. If this bound is not set, it will try to find that the assertion is true for an infinite number of cycles. In our case, in order to limit execution time, we have decided that if an assertion has not been violated for 75 cycles, it is considered as proven. From this point on, the time taken to test a new cycle becomes exponential, taking several days to test just one more cycle.

In turn, we decided that for test exercised on the Fetch Unit, we will have a timeout of 7 days from the moment the Testbench was executed, considering the proofs that have not been checked within the time as unproven.

7.3 Performance of FV Techniques

Different metrics have been collected in order to be able to perform a correct study of each of the configurations that have been developed for this project. Among them, we find total execution time, time of each one of the proofs, number of properties/assertions that have not been proved, etc.

Configuration	Total Time Spent (s)	Speedup	Unproven
Naive	604800	1	4
Reduction	604800	1	3
iCache Abstraction	103436	5.85	0
iCache + iTLB Abstraction	73896	8.18	0
BlackBox	71413	8.47	0

Table 7.1: FV Configuration metrics

In Table 7.1 we find both the execution time of the different configurations and speedup obtained compared to the naive version and the number of proofs that have not been able to finish. The only configurations that were not able to finish within the stipulated time-out of 7 days were the Naive configuration and the Reduction configuration. These two configurations would need more than 1 week to finish completely, leaving respectively 4 and 3 proofs unfinished. We can see a slight improvement in that the Reduction configuration has been able to obtain results for more proofs, but it is not efficient enough to obtain results in the predefined time limit.

The abstraction technique on the iCache module is the first one that is able to finish all the proofs of our Testbench. In addition to completing all the proofs, we also get an improvement in terms of test execution time, obtaining a total execution time slightly over one day (103,436s, which is equivalent to 28.72h). This means a 5.85 speedup compared to the time limit reached by the naive and the reduction techniques. Because of this time limit, the speedup is actually a lower bound of the actual figure. Had we left the naive version run until all proofs finished, the speedup would have likely been much higher.

The introduction of abstraction to both the iCache modules and the iTLB further reduces the execution time, bringing its completion to under a day's duration (73.896s, equivalent to 20.52h). This time reduction means a speedup of 8.18 compared to the naive version. These results are more acceptable in order to be able to use FV in a production environment without slowing down the development of the verification.

Finally, we have the version with BlackBox. This configuration has been able to reduce the execution time to 71.413s (19.83h), with a speedup of 8.47. The difference compared to the version with abstraction is just 40 min.

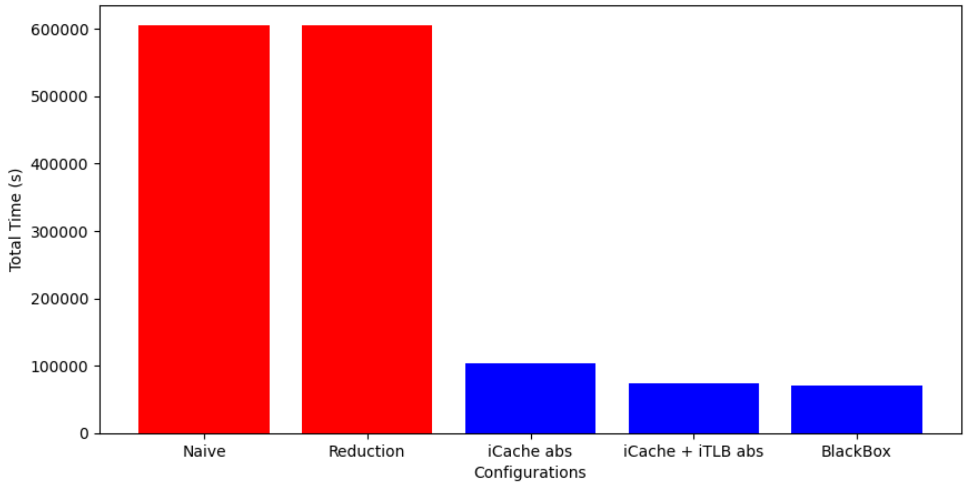

Figure 7.1: Total Time per Configuration

Figure 7.1 shows more clearly the time evolution of each of the configurations. The naive and reduction configurations reach times much higher than the rest of the versions. In the graph, we can clearly see the time improvement of the versions that attack directly to the memory in a more aggressive way, the 2 configurations that introduce Abstraction and the version with BlackBox.

We can see how the iCache abstraction is the one that has the most impact on the test, since adding the abstraction of the iTLB module only represents a 29.13% reduction of the execution time of the configuration with the iCache abstraction. Given the difference in the size of the memories that each of the modules has, this is in line with expectations.

In Figure 7.2 we have removed the Naive and Reduction configurations. Those configurations did not finish in the stipulated time and were killed after 7 days.

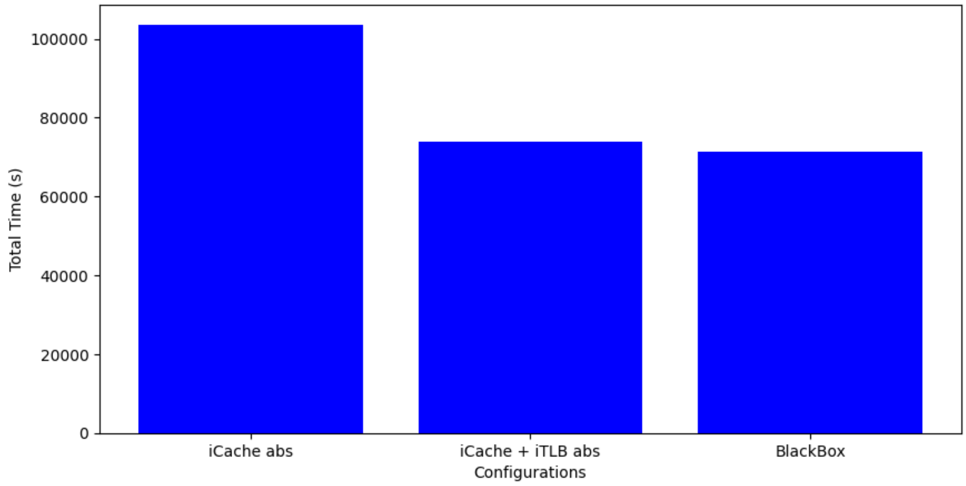

Figure 7.2: Total Time per Configuration (Finished)

Even with the time-out limit, the bar plots are distorting the scale and it is not possible to clear distinguish differences on the other configurations.

On the other hand, the difference between Abstraction and BlackBox is minimal, although the BlackBox technique obtains better results. This difference of 40 minutes in the Testbench implies a considerable effort to correctly obtain a formal model of the module on which BlackBox is to be performed. This model must be a replica of the module's operation, replacing all its functionality and logic by a well-defined set of assumptions.

Figure 7.3 shows the Cumulative Distribution Function (CDF) of all configurations, note that the time is shown in logarithmic scale. Thanks to this graph, we can observe the behavior of the different proofs within the Testbench. To make this graph, the individual times of each of the proofs have been obtained. The chart represents the Cumulative Distribution Function of the 5 configurations. The Y axis represents the percentile of proofs that finished within a given time (in logarithmic scale) represented by the X axis. The first thing we can observe with the Figure 7.3 is that for all configurations, 75% or more of the proofs finished in less than 100 seconds. Since the FV Tool bases its resolution on SAT-solvers, proofs

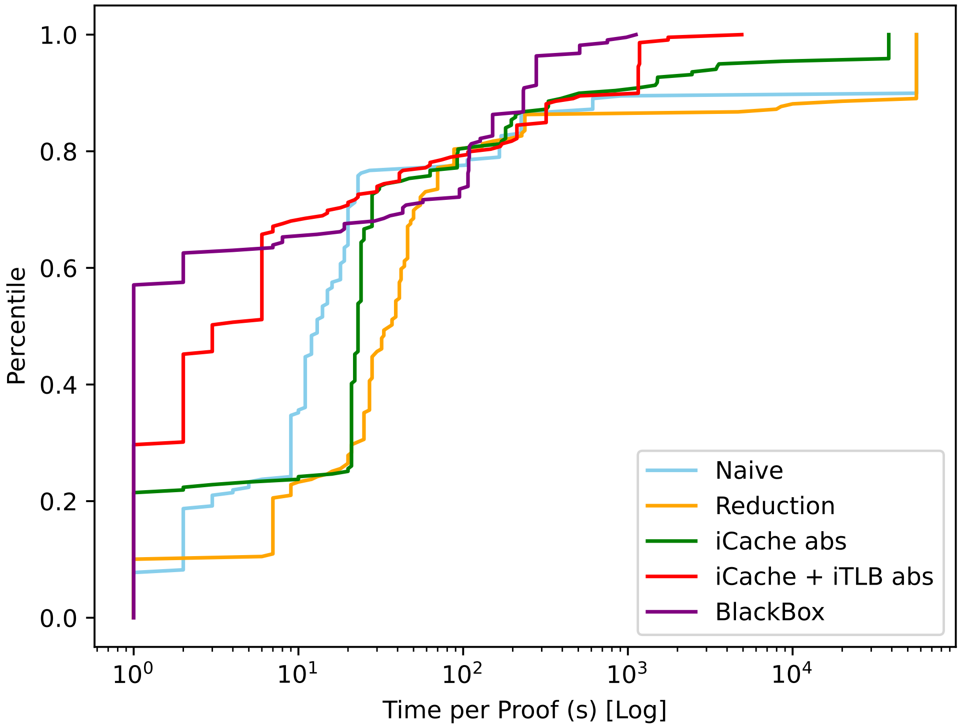

Figure 7.3: Cumulative Distribution Function (CDF) of Each Configuration [log scale]

with slightly larger space states may induce very large increases in the complexity and resolution time.

The presence of proofs whose execution is very slow is mainly found in the naive and reduction configurations. If we look at the versions where the memory modules are attacked more directly (configurations with abstraction and BlackBox) we can observe that the results improve considerably, reducing not only the execution time of these proofs, but also the number of proofs dominating the execution time. The configuration with abstraction in iCache improves the execution time of the dominant proofs (10.000s less) and also the number, representing less than 10% of the Testbench proofs.

If we look on the left side, we can see the evolution of the time reduction of the simplest proofs. As can be clearly seen, the number of proofs that finish in 1 second or less[1] has increased using the abstraction and BlackBox techniques. We can see that the configuration with BlackBox is clearly the one with the most proofs taking 1 second.

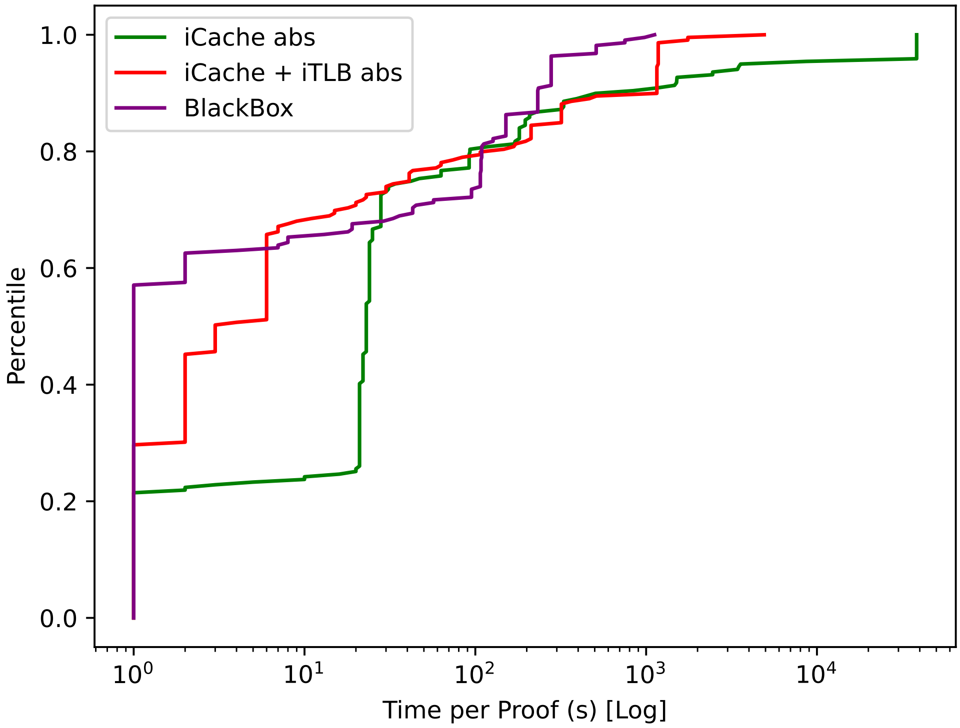

Figure 7.4: CDF of Abstraction (2 conf.) and BlackBox Configurations [log scale]

The Figure 7.4 represents the Cumulative Distribution Function of both Abstraction configurations plus the BlackBox configuration. The Y axis represents the percentile of proofs that finished within a given time (in logarithmic scale) represented by the X axis. As can be seen, the configurations with abstraction in the iCache and iTLB modules significantly improve test resolution. In this case, 100% of the proofs are verified in less than 5000s. While the configuration with

[1] The lowest granularity reported by the tool is 1 second

BlackBox we obtain better results per proof, since the execution time of each proof does not exceed 2000s.

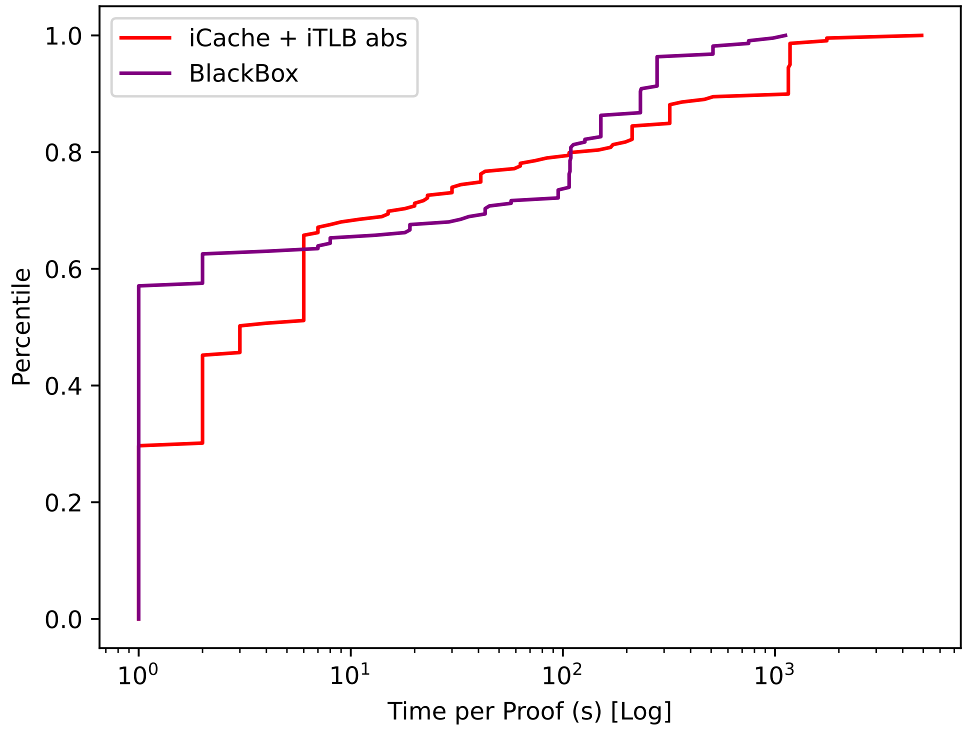

Figure 7.5: CDF of Abstraction And BlackBox Configurations [log scale]

As a last figure related to the CDF we have Figure 7.5. The Y axis represents the percentile of proofs that finished within a given time (in logarithmic scale) represented by the X axis. We compare more first hand the differences between the configuration in which abstraction is applied to the iCache and iTLB and the BlackBox configuration. This graph shows a very similar resolution, where 80% of the proofs run in less than 200s in both configurations. From this percentage onwards is where major differences are found. On the one hand, the configuration with BlackBox is able to run more than 95% of the proofs with less than 500s each, while the version with abstraction barely reaches 90%.

The BlackBox version is able to finish all its proofs in less than 1200s each. While at the same time mark, most of the proofs of the version with abstraction have finished within this interval except for a very small percentage, and it still completed this small percentage of proofs under 5000s.

With the graphs in Figures 7.3, 7.4 and 7.5, we can see that both Abstraction and BlackBox techniques work as we expect in reducing the complexity involved in memory. With the BlackBox technique being the best contributor to the reduction of proof complexity.

The reduction of the complexity of the memory modules is a great help for the FV Tool and a final reduction of the total Testbench execution time. This reduction in complexity makes the resolution of the SAT-solvers application that make up the FV Tool exponentially reduce their resolution times, making their use feasible.

According to the results shown in Figure 7.6 we can observe the longest execution time of a single proof. The times correspond only to the proofs that have been completed. First, we have the naive and reduction configurations, whose worst results are very similar between them. Since they have not finished all their proofs, and respectively have 4 and 3 unfinished proofs in 1 week, we can assume that the worst time in case of allowing the Testbench to finish would have been much higher than the one shown in this graph.

On the other hand, the time improvement in terms of the worst proof is more than remarkable for the abstracted and BlackBox configurations. The lower the memory, the lower the times for each of the proofs. Making use of this graph, we can correlate it with the graph in Figure 7.1, where the reduction of the total test time is clearly similar to the reduction of the worst proof. That, along with the reduction of proofs that finish close to the worst time (observable in Figure 7.3 with the CDF), explain the improvements of the different configurations.

So, with everything shown above we confirm the theory that memory is a crit-

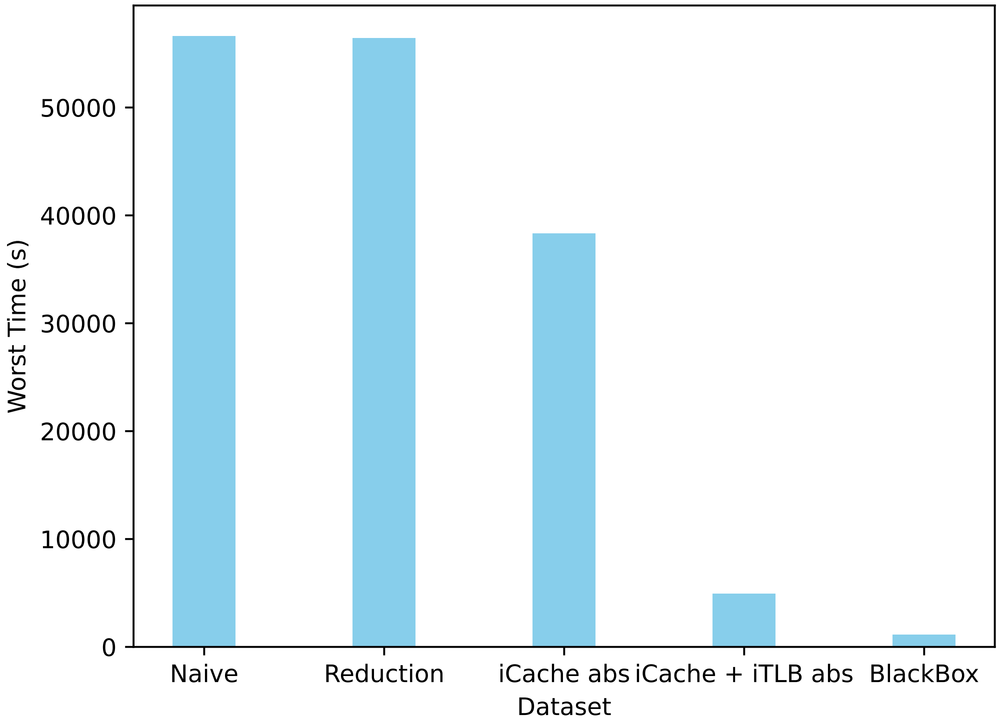

Figure 7.6: Worst Time per Configuration

ical aspect for Formal Verification, being very costly for the application to be able to manage it within the Testbench. Techniques that reduce the total memory state reduce execution times exponentially. This exponential time reduction is mainly due to the complexity of the SAT-solvers used by the FV Tool, so we are reducing the complexity of the proofs up to reasonable runtimes.

7.4 Performance Evaluation for Individual Proofs

To reinforce the results obtained for the whole Testbench, we have colleted a set of asserts to show the individual runtime variation in for Naive, iCache Abstraction, iTLB + iCache Abstraction, and BlackBox.

Listing 7.1 shows the assert “asrt_dont_fill_filled_lines”. A *fill* refers to a line that has been requested (the line was not present in the cache) and is inserted into the iCache. When a line is requested, the requested line bit must be set to 1. So, when a line is filled, this bit must be 1. That means that a line is not going to be received if it is not requested by some previous miss. In this assert, we check that if a fill line is valid in the iCache, it must be requested. The requested bit indicates that an older fetch has missed on the tags and the data line was requested, but it has yet to arrive.

Listing 7.1: Assertion A

```
asrt_dont_fill_filled_lines: assert property (
    i_itag.fill_valid
    |->
    i_itags.requested_line[fill_set][fill_way]
);
```

Listing 7.2 shows the assert “asrt_stall_no_iq_push”. This assert ensures that if there is a stall, the output signal that allows to insert into the pipeline is not set.

Listing 7.2: Assertion B

```
asrt_stall_no_iq_push: assert property (
    pc_valid && stall
    |->
    !add_to_pipeline
);
```

Listing 7.3 shows the assert “asrt_stall_maintains_data”. This assertion checks that, for a stall sequence, the data stored in the flops of the Fetch Unit is not modified and remains the same. When the previous cycle and this cycle is a stall and there is no signal to flush, this implies that the values of the flops are unmodified for the next clock. Note that this assertion has been simplified to improve readability, removing some other flops that are also checked.

Listing 7.3: Assertion C

```
asrt_stall_maintains_data: assert property (
    $past(stall && !flush) && stall && !flush
    |=>
    $past(pc_valid) == pc_valid
    &&
    $past(pc) == pc
    &&
    $past(way) == way
    &&
    $past(itags_valids) == itags_valids
    &&
    $past(exception) == exception
    &&
    $past(itlb_response) == itlb_response
);
```

Listing 7.4 shows the assert "asrt_never_multi_tag_hit". This assert ensures that for a valid Program Counter, the returned hit vector (hit vector is a vector of size 4 indicating which way have a hit) from the iCache contains at most 1 hit.

Listing 7.4: Assertion D

```
asrt_never_multi_tag_hit: assert property (
    pc_valid
    |->
    $countones(hit_vector) <= 1
);
```

In Figure 7.7 we can see how the configurations affect the assertions explained above. In this figure, it should be noted that the time axis is in logarithmic scale. As can be seen at a glance, the improvement of the techniques is clear in all proofs. The two best techniques are abstraction on all memory modules (iCache and iTLB) and BlackBox on iTLB plus iCache reduction.

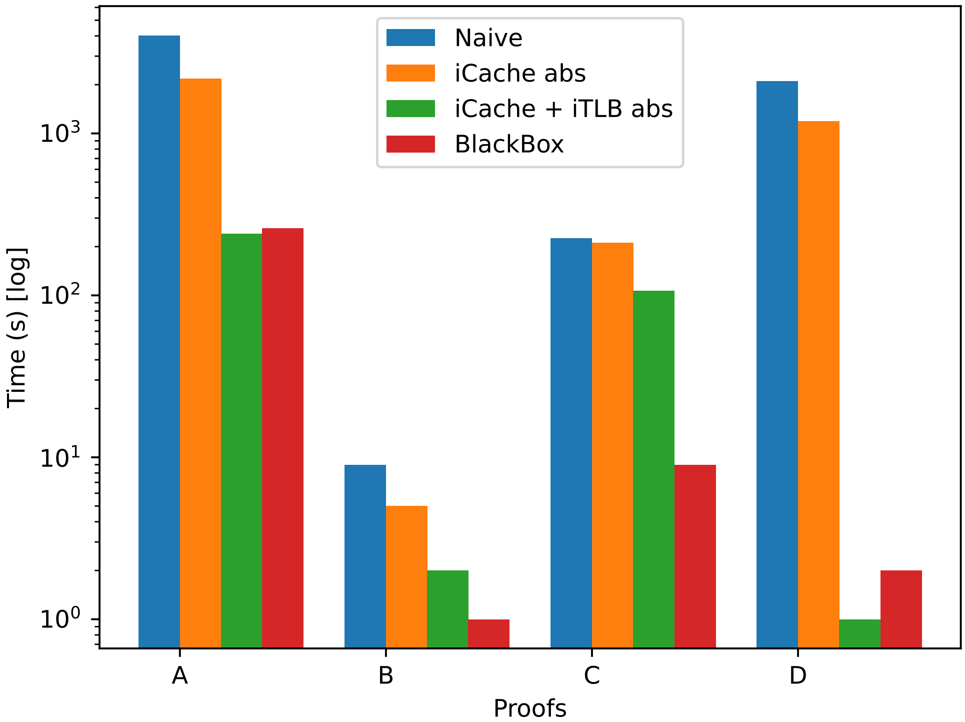

Figure 7.7: Time Comparison of Configurations for Proofs A to D [log scale]

However, we see that the best performing technique for a given proof varies depending on the assertion. We can see that the best technique for asserts A and D is the Abstraction technique in iCache and iTLB by a minimal difference (1s in case of assert D). While for asserts B and C the best technique is BlackBox. In these two assertions, we can observe a great reduction in times in the case of Assert C, whose difference is of an order of magnitude.

That phenomenon is caused due to how the COI of the proofs intersects with the space state affected by the applied techniques in each module. Taking into account the assertions we have selected, we can classify them in 2 groups: the COI intersects with iCache and iTLB, and the COI intersects mostly with iCache.

For the assertions B and C, COI intersects with both modules (iTLB and iCache). They take more advantage of the BlackBox Technique in iTLB although the iCache is only size-reduced (and should be slower). We can see that the intersection with the iTLB is higher than with the iCache. They take most of the improvement due to the BlackBox, which is considered most efficient.

For the assertions A and D, COI mostly intersect with iCache. The assertions are almost totally intersecting with iCache and only have a slight intersection with iTLB. So the BlackBox of the iTLB makes almost no improvement on the execution time, while the abstraction helps the most, equalizing the results of the BlackBox improvements.

8. Conclusions

Formal Verification is a powerful technique to ensure the correctness of the DUTs of our Hardware modules, but it is necessary a meticulous study of whether it is possible to apply it correctly and above all within a reasonable time.

The major complication for FV Tools is the presence of memory, which increases the complexity of the proof resolution exponentially and does not allow the Testbenches to finish within a time limit that we have determined as limiting (7 days).

Therefore, the different techniques that have been studied provide a good card when it comes to being able to adapt the Testbenches for different modules, reducing the spatial complexity and, therefore, the time complexity of the proofs, allowing the verification of complete modules in much shorter times and being feasible to implement in a real development environment.

The reduction technique remains as something "symbolic" that serves to help the other two techniques that are the ones that really reduce the complexity enough to make use of FV.

The Abstraction technique works correctly, reducing complexity to a great extent, despite not being the best technique in terms of speed in obtaining results. But since the complexity of introducing this technique in the Testbenches is not excessive, it allows a correct balance between implementation effort and reduction of the time complexity of the Testbench.

Finally, the BlackBox technique is the most effective, but the most complex to introduce, being able to involve a creation effort equal to the creation of a complete Testbench for the module to which BlackBox is desired. Therefore, this technique fits perfectly in case the verification of a module has already been performed, and we can reuse its assertions as BlackBox assumptions.

9. Future Work

Several proposals for future extensions of this project have been discussed. These proposals are focused on the integration of the Abstraction and BlackBox techniques in the same configuration. Another proposal is the implementation of a mechanism that allows us to obtain the result of a counterexample or visualization of a coverpoint provided by the Formal Verification Tool and pass it to a simulation environment.

The first proposal is based on improving the performance of the BlackBox version by adding Abstraction to the iCache module. This change may lead to an improvement in runtime, as observed in the different results shown in section 7. This configuration could involve changes to the current iCache abstraction version to make it fully compatible with the signals needed for the iTLB BlackBox assumptions. Another option would be to modify the iCache related assumptions to the current auxiliary code that makes up the iCache Abstraction.

The second proposal has to do with an application of Formal Verification. It is about implementing a functionality that allows us, from a "proved" coverpoint, to export the state of the DUT and the past waves to a simulation environment in order to run from this state a series of proofs in simulation. This allows us to test random sequences from a specific state of interest. This can be useful for the implementation of new functionalities in the module, or for the verification of areas in more critical points from a simulation environment.

Acronyms

BCP Boolean Constraint Propagation.

BDD Binary Decision Diagrams.

CDF Cumulative Distribution Function.

CNF Conjunctive Normal Form.

COI Cone Of Influence.

CPU Central Processor Unit.

DRAM Dynamic Random Access Memory.

DUT Design Under Test.

EDA Electronic Design Automation.

FIFO First In First Out.

FSM Finite State Machine.

FV Formal Verification.

GM Golden Model.

GPP General Purpose Processor.

GPU Graphic Processor Unit.

OBDD Ordered BDD.

PC Program Counter.

POS Product Of Sums.

ROBDD Reduced OBDD.

RTL Register-Transfer Language.

SAT-solvers Satifiability Checkers.

SLS Stochastic Local Search.

SVA System Verilog Assertions.

TLB Translation Lookaside Buffer.

UVM Universal Verification Methodology.

Bibliography

[1] Gordon E Moore. "The experts look ahead. Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965), pp. 114–119.

[2] *Moore's Law: further progress will push hard on the boundaries of physics and economics*. Accessed on April 23, 2024. URL: https://physicsworld.com/a/moores-law-further-progress-will-push-hard-on-the-boundaries-of-physics-and-economics/.

[3] HandWiki. *Software:Pentium FDIV bug*. Accessed on April 2, 2024. URL: https://handwiki.org/wiki/Software:Pentium_FDIV_bug.

[4] Christoph Kern and Mark R. Greenstreet. "Formal verification in hardware design: a survey". In: *ACM Trans. Des. Autom. Electron. Syst.* 4.2 (Apr. 1999), pp. 123–193. ISSN: 1084-4309. DOI: 10.1145/307988.307989. URL: https://doi.org/10.1145/307988.307989.

[5] David M. Russinoff. *Formal Verification of Floating-Point Hardware Design. A Mathematical Approach*. 2nd ed. Computer Science, Computer Science (R0). Springer Cham, 2022, pp. XXVIII, 436. ISBN: 978-3-030-87180-2. DOI: 10.1007/978-3-030-87181-9. URL: https://doi.org/10.1007/978-3-030-87181-9.

[6] IEEE Standards Association. *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. Accessed: 2024-06-06. 2017. URL: https://standards.ieee.org/ieee/1800/7743/.

[7] Erik Seligman, Tom Schubert, and M. V. Achutha Kiran Kumar. *Formal Verification: An Essential Toolkit for Modern VLSI Design*. English. 2nd ed. Elsevier, May 2023.

[8] Author's Name. *Title of the Book*. English. Morgan Kaufmann. Copyright © 2015 Elsevier Inc. All rights reserved. Elsevier Inc., 2015. ISBN: 978-0-12-800727-3. DOI: 10.1016/C2013-0-18672-2.

[9] Accellera Systems Initiative. *UVM User's Guide 1.2*. Accessed: 2024-06-06. 2015. URL: https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf.

[10] Rolf Drechsler, ed. *Advanced Formal Verification.* 1st ed. Copyright 2004 Springer Science+Business Media New York. New York, NY: Springer New York, 2004, pp. XXVI, 250. ISBN: 978-1-4020-7721-0. DOI: 10.1007/b105236. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1007/b105236.

[11] D.L. Perry and H. Foster. *Applied Formal Verification: For Digital Circuit Design.* McGraw-Hill electronic engineering series. McGraw Hill LLC, 2005. ISBN: 9780071588898. URL: https://books.google.es/books?id=imKnsuYmZMkC.

[12] A. Dharchoudhury and S.M. Kang. "Worst-case analysis and optimization of VLSI circuit performances". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14.4 (1995), pp. 481–492. DOI: 10.1109/43.372370.

[13] Ed Brinksma and Kim Guldstrand Larsen, eds. *Computer Aided Verification. 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27-31, 2002 Proceedings.* 1st ed. Vol. 2404. Lecture Notes in Computer Science. Published: 19 July 2002 (Softcover), 02 August 2003 (eBook). Berlin, Heidelberg: Springer Berlin, Heidelberg, 2002, pp. X, 362. ISBN: 978-3-540-43997-4. DOI: 10.1007/3-540-45657-0.

[14] M.W. Moskewicz et al. "Chaff: Engineering an efficient SAT solver". In: *Proceedings - Design Automation Conference* 2001 (Feb. 2001), pp. 530–535. DOI: 10.1109/DAC.2001.156196.

[15] Hantao Zhang. "SATO: An efficient prepositional prover". In: (Apr. 2006), pp. 272–275. DOI: 10.1007/3-540-63104-6_28.

[16] Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. "A Translation of Pseudo Boolean Constraints to SAT". In: *J. Satisf. Boolean Model. Comput.* 2 (2006), pp. 191–200. URL: https://api.semanticscholar.org/CorpusID:2821484.

[17] Knot Pipatsrisawat and Adnan Darwiche. "A Lightweight Component Caching Scheme for Satisfiability Solvers". In: (2007). Ed. by João Marques-Silva and Karem A. Sakallah, pp. 294–299.

[18] J.P. Marques-Silva and K.A. Sakallah. “GRASP: a search algorithm for propositional satisfiability”. In: *IEEE Transactions on Computers* 48.5 (1999), pp. 506–521. DOI: 10.1109/12.769433.

[19] Sahel Alouneh et al. “A comprehensive study and analysis on SAT-solvers: advances, usages and achievements”. In: *Artificial Intelligence Review* 52.4 (Dec. 2019), pp. 2575–2601. ISSN: 1573-7462. DOI: 10.1007/s10462-018-9628-0. URL: https://doi.org/10.1007/s10462-018-9628-0.

[20] Ahmet Bindal. *Fundamentals of Computer Architecture and Design*. 1st. Springer Publishing Company, Incorporated, 2017. ISBN: 3319258095.

[21] John Von Neumann. “Von neumann architecture”. In: *Online http://en. wikipedia. org/wiki/Von_ Neumann_ architecture* 8 (1945).

[22] Danijela Efnusheva, Ana Cholakoska Cilakova, and Aristotel Tentov. “A Survey of Different Approaches for Overcoming the Processor - Memory Bottleneck”. In: *Computer Science and Information Technology* 9 (2017), pp. 151–163. URL: https://api.semanticscholar.org/CorpusID:42320372.

[23] S. R. Sarangi. *Fetch-Decode-Execute Organization of the CPU*. Accessed on April 23, 2024.

[24] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X.

[25] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. ISSN. Elsevier Science, 2017. ISBN: 9780128119068. URL: https://books.google.es/books?id=cM8mDwAAQBAJ.

[26] Tse-Yu Yeh and Yale N. Patt. “Alternative implementations of two-level adaptive branch prediction”. In: ISCA ’92 (1992), pp. 124–134. DOI: 10.1145/139669.139709. URL: https://doi.org/10.1145/139669.139709.

[27] Antonio González, Fernando Latorre, and Grigorios Magklis. *Processor Microarchitecture: An Implementation Perspective*. 1st ed. Vol. 1. Synthesis Lectures on Computer Architecture. Published: 31 May 2022. Cham: Springer Cham, May 2022, pp. X, 106. ISBN: 978-3-031-00601-2. DOI: 10.1007/978-

3-031-01729-2. URL: https://doi-org.recursos.biblioteca.upc.edu/10.1007/978-3-031-01729-2.

[28] R. P. Kurshan. “Analysis of discrete event coordination”. In: (1990). Ed. by J. W. de Bakker, W. -P. de Roever, and G. Rozenberg, pp. 414–453.

[29] E. Allen Emerson and A. Prasad Sistla. “Symmetry and model checking”. In: *Formal Methods in System Design* 9.1 (Aug. 1996), pp. 105–131. ISSN: 1572-8102. DOI: 10.1007/BF00625970. URL: https://doi.org/10.1007/BF00625970.