Tampere University

Tuukka Haapakumpu

X-PROPAGATION IN RTL SIMULATION

Faculty of Information Technology and Communication Sciences
Masters thesis
November 2022

ABSTRACT

Tuukka Haapakumpu: X-Propagation in RTL Simulation
Masters thesis
Tampere University
Electrical engineering
November 2022

RTL simulations are a major part of hardware development and verification. RTL simulation is the most commonly used method of verifying the correct operation of hardware designs. Verification is often run parallel to the hardware design and verification is a major part of SoC projects even before any physical releases.

RTL simulators depict and read logical values usually with ’1’ and ’0’. In some situations the simulators cannot always tell what value is going into a component, then it will assign it an ’X’. Problems arise from the fact that there is no state ’X’ for a signal in physical devices. If the simulator just assigns a ’1’ or a ’0’ to an X-value there is a significant chance of it being wrong which will cause errors later down the line.

X-propagation in a RTL simulator is a way of making the X-value run through the design to see where it ends up. This is done while configuring the simulator to make the ’X’ end up in the least amount of places while trying to cover all places where there might be ambiguity. Simulations can be run in an X-pessimistic or X-optimistic way, where a pessimistic simulation will produce more X-values

In this thesis simulator A and its X-propagation plugins were tested. The aim was to assess the simulators ability to catch X-related errors and find any issues or restrictions that would hinder its use. The testing was done by running basic tests to see the finer details of the X-propagation plugin of each simulator and a larger test done on a design provided by the company. The basic tests were six different basic design simulations: an if-else statement, a case statement, a random number generator, a state machine, a RAM circuit and a shift register. The environment used in the simulations was provided by the company which restricted the choice of simulators.

The basic tests were run with and without X-propagation modules enabled on simulator A. The basic simulations went mostly as expected with a few surprises. Overall the simulator performed mostly as expected and without issues.

The larger simulations were run using simulator A. The X-value tracing function of simulator A was used to find sources of X-values and the testcases were run first without any X-propagation enabled and then with the X-propagation plugin enabled in relevant modes. Unfortunately there was no big difference in simulation results between modes.

In the simulator analysis some differences were found between simulation and the user guide. A notable bug was found in simulator A which the simulation vendor was informed about. While testing it was found that the simulator could be used in X-propagation simulations especially with larger designs.

Keywords: RTL simulation, X-propagation, Simulator, Verification

TIIVISTELMÄ

Tuukka Haapakumpu: X-Propagaatio RTL Simulaatiossa
Diplomityö
Tampereen yliopisto
Sähkötekniikka
Marraskuu 2022

RTL-simuloinnit ovat tärkeä osa laitteistokehitystä ja verifiointia. RTL-simulointi on yleisimmin käytetty menetelmä laitteistosuunnitelmien oikean toiminnan todentamiseen. Verifiointi suoritetaan usein rinnakkain laitteistosuunnittelun kanssa, ja verifiointi on tärkeä osa SoC-projekteja jo ennen fyysisiä julkaisuja.

RTL-simulaattorit käsittelevät loogisia arvoja, jotka ovat yleensä ’1’ ja ’0’. Joissakin tilanteissa simulaattorit eivät kuitenkaan pysty kertomaan, mikä arvo komponenttiin on menossa. Tällöin simulaattori antaa sille arvon ’X’. Fyysisissä laiteissa ei ole tilaa ’X’ signaalille, mistä syntyy ongelmia. Jos simulaattori muuttaa arvon ’X’ arvoon ’1’ tai ’0’, on suuri mahdollisuus että se on väärässä, mikä aiheuttaa myöhemmin virheitä.

X-propagointi RTL-simulaattorissa on tapa saada arvo ’X’ kulkemaan suunnittellun laitteen läpi simulaatiossa, ja selvittää mihin se päätyy. Tämä toteutetaan konfiguroimalla simulaattori siten, että arvo ’X’ päätyy mahdollisimman harvoihin paikkoihin. Samalla pyritään kattamaan kaikki paikat, joissa voi olla mahdollisia epäselvyyksiä. X-propagoinnilla simulaatiot voidaan suorittaa joko X-pessimistisesti tai X-optimistisesti. Pessimistinen simulaatio tuottaa simulaatioon enemmän arvoja ’X’ kuin optimistinen.

Tässä työssä testattiin simulaattoria A ja sen X-propagointiliitännäisiä. Tavoitteena oli arvioida simulaattorin kykyä havaita X-arvoja ja löytää niiden käyttöä haittaavia virheitä tai rajoitteita.

Testaus suoritettiin ajamalla perustestejä simulaattorin X-propagointi -liitännäisen hienompien yksityiskohtien selvittämiseksi. Tämän lisäksi suoritettiin laajempi testi, joka tehtiin yrityksen toimittamalla alustalla. Perustesteissä simuloitiin kuutta erilaista peruskomponenttia: if-else -lause, case-lause, satunnaislukugeneraattori, tilakone, RAM-piiri ja siirtorekisteri. Yritys tarjosi simulaatioissa käytettävän ympäristön, mikä rajoitti simulaattoreiden valintaa.

Perustestit ajettiin ensin ilman X-propagointimoduuleja ja tämän jälkeen niiden ollessa käytössä. Perustesteissä keskityttiin simulaattorin A X-propagointitiloihin. Perussimuloinnit sujuivat enimmäkseen odotetusti lukuun ottamatta muutamia yllätyksiä. Kokonaisuudessaan simulaattori A suoriutui perussimuloinneista lähes odotetusti ja ilman ongelmia.

Suuremmissa simulaatioissa simulaattori A tarjosi mahdollisen X-arvon jäljittämiseen sopivan toiminnon. Tämä X-arvo aiheuttaa simulaation epäonnistumisen, tai on muulla tavoin mielenkiintoinen aiheuttamatta epäonnistumista. Testitapaukset ajettiin ensin ilman, että X-propagointitila oli käytössä. Tämän jälkeen ne ajettiin käyttäen X-propagointitilaa. Valitettavasti simulointituloksissa ei ollut suurta eroa tilojen välillä.

Simulaattorin ominaisuuksien analysoinnissa havaittiin joitakin eroja oppaassa esitetyssä ja odotetussa toiminnassa. Simulaattorissa A havaittiin virhe, joista ilmoitettiin simulaattorin toimittajalle. Testauksen aikana todettiin, että simulaattoria voidaan käyttää X-propgointisimulaatioissa, erityisesti suurissa simulaatioissa.

Avainsanat: RTL simulaatio, X-propagointi, Simulaattori, Verifiointi

PREFACE

The writing of this thesis was a major undertaking but at the same time a good learning experience. Working on this thesis has shown me the importance of tenacity and co-operation, more than any other project I have embarked upon. For the subject I would like to thank Risto Sterling and my supervisor on the company side Petri Kukkala. On the vendor side I would like to thank Juho Järvinen for the technical support throughout this project.

I would like to thank my supervisors Sakari Lahti and Timo Hämäläinen for the patience and support which made this project possible. I would also like to thank my family for the support and motivation they gave me and for this undertaking. Without you this thesis would never have been finished. A special thanks to all my friends without whom these years would have been unbearable, who I have been able to vent my frustrations regarding this project and especially for the unforgettable experiences we have shared. Hope there are still many to come.

In Tampere, 22nd November 2022

Tuukka Haapakumpu

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAMS AND ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

ASIC	Application Specific Integrated Circuit
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
HDL	Hardware Design Language
IP	Intellectual Property
LSB	Least Significant Bit
MSB	Most Significant Bit
RAM	Random Access Memory
RTL	Register Transfer Level
SoC	System on Chip
UVM	Universal Verification Methodology
VHDL	Very High Speed Integrated Circuit Hardware Design Language

1 INTRODUCTION

Verification is an integral part of developing new System on Chip (SoC) devices such as Application Specific Integrated Circuits (ASIC) and Field Programmable Gate Arrays (FPGA) [1]. Especially, as the size and performance of designs are increasing, the importance of finding possible errors in the design rises [2]. Simulating and using formal verification methods when done properly will increase the amount of errors that are caught before moving forward in the design [1]. Unfortunately all errors cannot be uncovered only by simulating due to the limitations of the verification tools used.

There are many ways that errors might unintentionally be introduced into the design and a single tool might not take all of them into account. This makes it exceptionally important to have a more diverse set of tools for the project design verification. One part of design verification is the evaluation of ambiguity in the design which causes X-values. X-values propagate through designs when the simulator cannot decide the output value from the input values given, which is common when one or more input values are X-values.

Simulator vendors are aware of the problems caused by X-propagation, so many of them offer tools to tackle it. Different tools made by different vendors will ultimately have some differences on how they operate and those differences will lead to some differences in results and performance between them.

The purpose of this thesis is to research and evaluate a simulation tool for the propagation and its ability to catch unknown X-values through different SoC hardware designs. The simulator in use is referred to as simulator A. The simulation modes include operation in X-optimistic and X-pessimistic modes. Additionally the purpose of this thesis is to help develop a methodology or methodologies for X-propagation simulation to be used at the company.

Different modes were compared by simulating basic hardware designs and complex Intellectual Property (IP) block designs. Smaller design units included designs like a random number generator (RNG) and a shift register. For the larger IP block case a design with known problems with X-propagation will be used.

The thesis is structured as follows: Chapter 2 covers the basic concepts of X-propagation in Register Transfer Level (RTL) design simulation and sheds a light on some problems with X-propagation simulations. Chapter 3 gathers simulation setups and results for basic designs using different simulation settings. Chapter 4 covers the simulations for a larger IP. Chapter 5 is focused on the analysis of the simulation results and in Chapter 6 are the

conclusions.

2 X-PROPAGATION

As SoC designs grow ever larger verification remains an important part of SoC project flow. The goal of verification is to find and rule out possible errors early in the design lest they hamper the design later during the project. Verification flow runs in parallel with the project design flow and ought to be run at various levels of the design. Already in 2008 70 % to 80 % of design effort was taken up by verification, so a large portion of project labour costs are made up of finding errors in the design [3].

2.1 Verification and RTL simulation

Verification is the umbrella term for multiple types of testing and analysis, such as formal methods and simulation. The goal of verification is to ensure the design works as planned. Due to large project sizes, SoC verification requires many different verification tools and methodologies. [2] For small designs, verification can be just an afterthought without many problems. For larger and more complex projects, the design team always has to know how it will be tested and verified [4].

Verification methodologies include simulation, timing analysis, formal verification and hardware emulation. Simulation is one of the most, if not the most, used type of verification but does not verify the lack of errors and is very time consuming. Timing analysis checks the timing of the design. Meaning checking for delays. Formal verification is another commonly used verification method. It uses mathematical models to verify designs. Formal methods are covered more thoroughly in section 2.3. [4]

SoC Verification operates in many different levels of abstraction. Since modern hardware designs have become more complex, including millions upon millions of transistors, no human or computer can realistically analyze the design directly. Different abstraction levels are used to balance between simulation time dictated by the complexity of the design and its accurate representation. In digital system design, four abstraction levels are considered. Going up this starts at the transistor level continuing into gate-level, register transfer level, and finally the processor level. Higher-level abstractions omit much of the data whereas low-level abstractions add more detail to it. [4]

Register transfer level is the digital design abstraction level that models digital circuits in terms of data flowing between hardware registers. This is an abstraction level higher from gate-level design. RTL design includes three primary parts, registers, combinatorial logic components like adders and lastly clocks all of which are written usually in Hardware

Design Languages (HDL) like Very High Speed Integrated Circuit Hardware Design Language (VHDL) or Verilog. [4] In large designs, where parts of the design are produced by an outside vendor or a part of the design is reused, a black-box model of the part may be used to speed up design and simulation time. Additionally, the testbenches used in RTL simulation can be reused in top-level simulation increasing verification throughput. [2]

In RTL simulations the Design Under Test (DUT) is checked to comply with the goals of the project. As an input to the DUT, a vector of inputs is fed by a testbench, and the output of the DUT is then compared to the expected output by a testbench or a testcase.[5] Nowadays especially in larger designs, testbenches are usually written in some verification language like SystemVerilog using Universal Verification Methodology (UVM). To ensure better verification results, it is common practice for different teams of engineers to design the DUT and the testbench since the DUT designer might be blind to some mistakes.

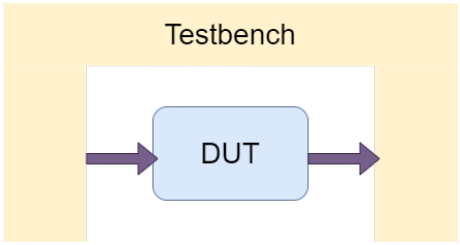

Figure 2.1. *Simple testbench and DUT*

In Figure 2.1 we can see a simple representation of a DUT and its testbench. In the figure from the left, an example subset is fed to the DUT inputs and the corresponding outputs are coming out on the right. It is important to note that not all inputs can be seen directly affecting the output, or it might take more time to see the effect. The basic working principle of a UVM test is essentially the same as in a VHDL or Verilog testbench.

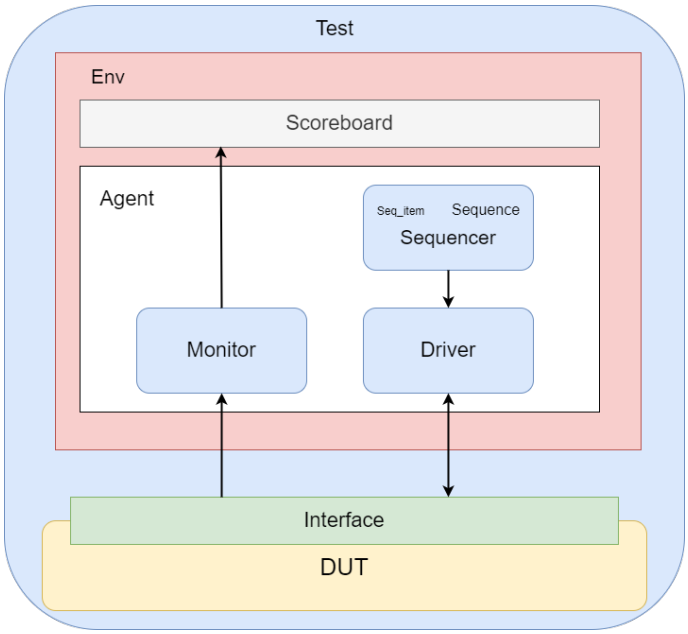

Figure 2.2. *Simple UVM testbench [6]*

In Figure 2.2 we can see that a simple UVM testbench architecture. The testbench consists of a top class for the test responsible for testbench configuration and stimulus instantiating, among others. The env or environment is a container for high-level components like the agent and scoreboard, an agent that handles UVM component grouping. The monitor monitors interface signals which are made into a packet and sent to for example the scoreboard. The scoreboard receives data and compares it to the expected output, the sequencer routes data packets to and from the UVM sequence, and a driver drives packet data from the sequence to the DUT, the interface for the DUT.

In verification, coverage is a metric for measuring when the verification is ready [7]. The coverage of verification is mostly determined by the quality of the testbench. For wider coverage, the testbench must check the DUT with a large number of random input values according to design constraints to cover a wide array of inputs. Random data vectors might not cover all possible input combinations. This means that some input data combinations are run as directed tests. Directed tests are run, by feeding the design known input values that might cause errors. Such cases are the edge, corner, and boundary cases. In the edge case, input data on the extreme limits of operation, is fed into the DUT, in the corner case the DUT operates outside normal operating parameters, and in the boundary case one or more input is outside the operating limits. When errors are found in the DUT by the verification team, the hardware designers modify the design, and the cycle continues.[5]

2.2 Basics of X-propagation

HDLs represent unknown logic values using X-values. When the simulator used to test the design can't decide if the logic value is '1', '0' or high impedance 'Z'; it will assign an

X. On the other hand X can be specifically assigned to depict a don't-care states, which can be confusing at times and therefore should not be used interchangeably. X logic values are often caused by system or component level resets, gated clock systems or other events that introduce uncertainty. [8]

Depending on the simulation setup, there can be one of two problems. Firstly the simulator might convert an X into an exact value like '1' or '0' by executing code that is too optimistic, this is called X-optimism. Secondly the simulator might generate too many X-values by executing code that is too pessimistic, this is called X-pessimism. [9] X-optimism and X-pessimism will be covered more in depth in the next chapter.

As design sizes have continued to increase, energy saving designs have become more commonplace. When a part of the chip is powered down, its output can be initialized into an X-state. These X-states can propagate through the design and create more uncertainty through some coding methods or excessive X-pessimism in powered-up circuits. [10]

Large reset trees are not viable for large designs, and implementing them would, in most cases, prove expensive. Especially with FPGA designs where the usable area is restricted by the chip size and resets would take space from the functional design. This will cause some of the registers in a design not to be connected to a reset and may cause unknown values to be generated during startup as they are uninitialized, for example. In addition, uninitialized latches can be a major source of X-values. X-value sources are covered in section 2.4 [11]

Understanding how X-values move through the simulated logic helps designers and verification engineers find and debug possible problems. Basic knowledge of X-value sources helps the designers and verification engineers dig deeper and find the sources of the problems caused by ambiguity.

It is helpful to look at simple hardware components prone to X-value problems. One component often mentioned is a multiplexer declared with an if-else statement. Table 2.1 shows how a simple multiplexer propagates and X-value through itself.

Table 2.1. *X-value propagation through a multiplexer (edited from [11])*

ENA	A	B	OUT
X	0	0	0
X	0	1	X
X	1	0	X
X	1	1	X

The table shows that the first three cases resolve as expected, but when the A and B inputs are both '1', by any logic, the output should be '1', as it is in silicon, but the language semantics turn it to an X.

Looking at gate level operation can shed light on operation in the RTL level. Table 2.2 shows how some simple Verilog operators handle X-values in a standard simulation.

Table 2.2. *X-value operations in Verilog (edited from[12])*

NOT		AND		OR		XOR		XNOR	
~X	X	(X AND X)	X	(X OR X)	X	(X XOR X)	X	(X XNOR X)	X
		(0 AND X)	0	(0 OR X)	X	(0 XOR X)	X	(0 XNOR X)	X
		(1 AND X)	X	(1 OR X)	1	(1 XOR X)	X	(1 XNOR X)	X

The tables 2.1 and 2.2 show that in many cases, the X-values might propagate through simple logic elements, which might lead to a significant amount of lost information down the line. This information can be otherwise normal function of the logic or erroneous data, which would lead to the logic working improperly. The tendency of X-values to propagate is affected by X-optimism and pessimism and if an X-propagation simulation is running. [13] The simulation in question seems to be running somewhat pessimistically.

X-propagation simulations are more time consuming since the propagating X-values need to be calculated for the proceeding steps of the simulation and can mask critical fault data and information. The loss of faulty data caused by this, would lead to a decrease in verification coverage as fewer faults are caught. This in turn, means that X-propagation should not be enabled in all simulations, especially if simulation time is of concern. X-values are very useful for finding problems, but the downside is that it may hinder other functions and, as a result, should be used sparingly.

Masking correct data and possible bugs with X-values will complicate debugging processes as more simulation and code/error review time is needed to catch problems. As other valid values than ’1’ or ’0’ don’t exist in hardware, their functionality will differ from X-propagation simulations that have effectively 4 states: ’1’, ’0’, high impedance ’Z’ and ’X’. This will cause difficulties in comparing simulation results with lab measurement results. Analyzing the X-propagation simulation waveform is labour-intensive manual inspection that can’t be automated yet. This requires a level of understanding of the design, so simulation and analysis can’t be done by everyone. [9]

The handling of X-values might be different between simulators. A simulator may calculate all possible outputs when an X is encountered at the input, or it may change the HDL semantics in a specific way. [14] This is the main reason simulation times are longer on X-propagation simulations. On rarer occasions the X-values cause the simulation to enter an endless loop which is not a problem caused by the simulator but the design itself.

2.3 Formal X-propagation verification

As verification is a significant bottleneck in design flows, different options to verify the designs have been proposed as an alternative to classical simulations. One alternative

group of methods are formal verification methods. In formal verification rigorous mathematical models are used to verify the correctness of algorithms behind the design under test. [15]

Formal methods are used in several areas like symbolic checking and symbolic simulation. Formal methods may increase the coverage of the design resulting in more bugs uncovered that may have been left undiscovered by just simulating. In many applications formal verification is cheaper than simulation since there is no need to write a separate testbench. [15]

Formal methods are used for checking designs for X-propagation errors. Validation of X-states is one of the top applications for formal methods used by the industry. In the case of X-propagation, formal analysis has some distinct advantages. They can pinpoint sources for the X-states that could have been masked due to X-optimism in RTL simulation. Additionally, formal methods can thoroughly evaluate the DUT behaviour for X-states propagating and possibly corrupting critical values. [16]

Using formal methods for X-propagation in large designs at the scale of the top-performing FPGA chips is not viable at the moment. On smaller IPs, formal methods are still viable, but due to the high demand for computing power, top-level verification requires a different solution. [17]

Additionally, if there is combinatorial logic between IP modules, the top-level design cannot be represented by chaining the IP modules. This implies that a formal model of the top-level design would not be the sum of all IP-level models. The logic between the blocks might be so complex that it warrants its own model, or it might be as simple as a few gates, so a universal solution to this problem is not easy.

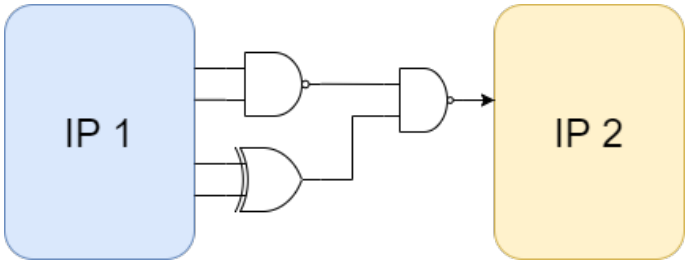

Figure 2.3. *Logic between IP blocks*

In Figure 2.3 we can see a representation of logic between IP-blocks. This logic should be accounted for when testing the overall functionality of the larger design. Since it is located between IP blocks, it might not be taken into account when formally verifying a design unless it is added to the model.

2.4 RTL X-propagation simulation

X-values in RTL simulation are very useful for representing the uncertainty of some values, although they introduce some challenges to be considered. The propagation of X-

values through RTL and logic-gate level designs requires considerations lest they cause significant problems later. These problems might arise during simulation or in hardware as the difference in results compared to the simulations. [10]

RTL simulation often cannot indicate the problems related to the lack of X-propagation. X-values can cause differences between gate-level, and RTL simulation and X-propagation tools can save time in debugging those differences and the design problems behind them. [18]

In most situations, simulators don't propagate X through the design but handle these cases optimistically, resulting in possible errors being missed. To find these errors in RTL simulations, many vendors offer X-propagation plugins to their simulators. With these tools, the verification engineer can set the level of X-pessimism and X-optimism to represent the physical implementation of the design. This system is not flawless or instant, for that matter but will require iterations of tweaks to the X-optimism or X-pessimism and possibly modifying the underlying HDL.

As stated in the last section, formal methods are more effective at catching errors for most applications, for example, verifying a single IP block. Although not perfect, RTL simulation is still a viable tool for top-level X-propagation simulation in large designs. Although Formal methods might come out on top concerning single IP-block verification, RTL simulation will do well to complement formal methods to increase verification coverage even more. RTL simulation considers the possible logic between separate IP blocks and thus increases the coverage.

There are multiple ways the simulator can generate and propagate an X. Although all X-sources cannot be completely removed e.g. because of restrictions on reset trees, one can simplify the work required down the line to correct errors created by X-values. This can be done by using coding principles that focus on avoiding X-states to begin with or by focusing on writing the code in a way that does not propagate the generated X.

The best way to handle possible X-state problems and limit the propagation of X-values is to avoid X-value generation in the first place. To better understand how to avoid X-values, the engineer needs to understand the mechanics causing them. The sources might be intentional or a byproduct of the decision made due to, for example, size or timing constraints. Some sources of X include:

- Uninitialized variables and registers
- Operations resulting in unknown results
- Forced X-values
- Some timing violations
- Undriven and floating wires and wires with multiple drivers

These are only a few examples on how a simulator might generate an X. [10]

Assuming the designer or verification engineer is using the proper simulators and X-

propagation plugins while running RTL simulations, they can find and address issues originating from initialized parts of the design or other X-sources. The issue posed by standard simulation X-optimism is addressed by propagating the X-values forward in time. [19]

With regards to the propagation of X forward through time three pathways are most prominent. These are if and case statements and conditional assignments. This means that, for example if a conditional statement gets an input of X, it will be propagated to the output by the simulator X-propagation plugin by changing the language semantics. The X will affect the following logic similarly and can be seen in the simulator waveform. If these X-values continue to be unaddressed, they may cause the simulation to fail. A few of these failures are presented in chapter 5. [19]

Simulators used by the SoC industry are highly advanced and their basic functions will be comparable. In complex designs, the major functional difference will be the overall performance, not the simulation results.

There are multiple HDL simulators to choose from, and even covering all or most of them would make the scope of this thesis too large. This thesis will focus on one simulator to keep the length of this text reasonable.

Since the underlying algorithms are not available to the designers the major differences seen by them are the price, the look, platform support on some simulators and the look and feel of the simulator GUI. With this in mind, the primary criteria from the engineers' standpoint are the simulator's usability and the supported platforms. On a company level, the main criteria would be the price and the simulator's performance.

Looking at how the simulator handles X-propagation is important for understanding the usability of it. Vendors do not release the underlying algorithms for simulator X-value handling, although the working principles are listed in their user guides.

For smaller designs gate-level simulations and pseudo-exhaustive 2-state methods are used to find problems regarding X-propagation. These simulation methods become too slow and processing power consuming to be used in larger designs, as they cover only a portion of the verification flow. [18]

2.5 Code and simulation examples

If-else and case statements are often mentioned as sources of X-propagation problems. Looking deeper into how the simulator handles X-values in different simulation modes is useful for understanding how the simulation is actually working.

In this section the X-propagation simulation modes for the simulator is examined. The modes range from very X-optimistic to pessimistic. The possibility of changing the severity of optimism and pessimism enables the design to be studied more thoroughly.

Simulator A user guide does not give separate code examples between VHDL and Ver-

ilog, so it can be assumed that the simulation outcome will be the same regardless of the HDL used. Simulator A provides options to modify the simulation behaviour with X-propagation plugin options to match silicon as close as possible.

Simulator A has different X-propagation plugin modes of operation. Simulator A X-propagation plugin modes are none, trap, resolve and pass. With these modes, the designer can set the wanted level of X-optimism and pessimism to fit the situation.

X-propagation plugin mode 'none' doesn't modify the simulation of the design instance, even though the X-propagation plugin is enabled. This will make the simulation run as if the X-propagation plugin was not enabled.

Mode 'resolve' is the simulator A X-propagation plugin mode that more closely resembles the silicon behaviour. 'Resolve' mode is not as optimistic as standard simulation and not as pessimistic as 'pass' mode. This will uncover problems with ambiguity but doesn't mask unrelated bugs in the design as much.

Mode 'pass' is the X-pessimistic approach to simulation. This will propagate the X-values through and will more likely cause X-related errors and hide design bugs. 'Pass' mode will turn some non-ambiguous values into unknown values.

The vendor does not provide enough information on how the 'trap' mode functions, so it will be omitted from this comparison. The only table provided in the user's manual 'trap' mode seems to result in an output similar to 'none' mode and basic RTL simulation, with no explanation provided [14].

A simple if-else statement is as follows. This would result in a multiplexer component.

```
if (ENA)
    OUT = B;
else
    OUT = A;
```

Listing 2.1. *If-else statement in Verilog. [14]*

In table 2.3, the truth table for simulator A X-propagation plugin modes is represented when running an X-propagation simulation for the if-else statement.

Table 2.3. *If-else simulations for simulator A X-propagation plugin modes (edited from[14])*

ENA	A	B	pass	resolve	RTL sim	silicon
X	0	0	X	0	0	0
X	0	1	X	X	0	?
X	1	0	X	X	1	?
X	1	1	X	1	1	1

From table 2.3, it can be seen that the resolve mode represents the silicon behaviour the closest. When the control signal is X and other inputs are the same, the output takes the value of the inputs. When the input values don't match, the X is propagated to the output.

A simple case statement written in Verilog is as follows. This will result in a simple multiplexer when instantiated.

```
Case (ENA)
1'b0 : OUT = A;
1'b1 : OUT = B;
```

Listing 2.2. *Case statement in Verilog. [14]*

Table 2.4 is the truth table for simulator A X-propagation plugin modes for simulating the case statement simulation.

Table 2.4. *Case simulations for simulator A X-propagation plugin modes (edited from[14])*

ENA	A	B	pass	resolve	RTL sim	silicon
X	0	0	X	0	0	0
X	0	1	X	X	0	?
X	1	0	X	X	1	?
X	1	1	X	1	1	1

As seen from table 2.4, when the control signal is X, the behaviour of pass mode and standard RTL simulation is too pessimistic and optimistic respectively, while resolve mode represents silicon operation the closest. Additionally, when looking at other cases like latch and flip-flop statements, resolve mode seems to represent silicon behaviour the closest [14].

2.6 Problems in X-Propagation Simulations

All simulators are running models of the actual silicon behaviour. This means that no simulation is an exact match to the design when tested in the lab. The difference between X-propagation simulation and silicon is that in silicon, there is no such thing as an X. In this section some of the major problems in X-propagation simulations are reviewed. Knowing about the shortcomings of simulators will make verification easier and faster.

2.6.1 X-Optimism

X-optimism means the simulator handles X-values too optimistically, resulting in 'known' two-state values of '1' or '0' being assigned when the simulation result should be unknown. This would inadvertently hide X-value-related bugs in the design. Standard RTL

simulation operations are intrinsically X-optimistic. X-optimism can be hard to detect and debug since the X-values 'disappear' as they are changed into 'known' values. Finding the bugs in the design caused by overly optimistic simulation can be extremely difficult, and forcing X-values into known values means the cause of the X-value is hidden [20]. These design bugs can be caught with secondary tools, or at a later stage of the design and verification process. In a worst-case the bugs would go unnoticed until the design is tested in actual silicon.

A simple example of X-optimism in standard simulation done in SystemVerilog is an AND gate with inputs of '0' and an X that will result in an output of '0' [10]. When either '0' or '1' ANDed with an X, the output must be 'X'. The same applies in the case of an OR gate.

X-optimism is not always a bad thing. In cases when the operation of the silicon is ambiguous, an optimistic approach is the one closest to the operation of the silicon. The worst thing an optimistic approach can do is mask the ambiguity of silicon behaviour as if no problem exists.

2.6.2 X-Pessimism

X-pessimism means the simulator handles X-values too pessimistically, resulting in an excessive amount of X-values being assigned and propagated. X-pessimism happens when the simulation is driving an output of X when, in actuality there is no uncertainty in the simulation or the physical silicon. This, in turn can hide bugs in the design and result in data loss. Some examples of this were covered in the previous chapter.

X-pessimism allows X-values to propagate through the design, allowing them to be more easily detected, debugged and corrected. The downside is that the X-values mask data, which makes the debugging of data harder. As the X-values propagate downstream, their source might be far away. Finding the output X-value source in an X-pessimistic simulation might take much work and time.

X-pessimism can cause problems with simulation by itself. A simulation might lock up because of X-values where possibly ambiguous values in silicon wouldn't cause this problem. One example of this is a simple clock divider seen in Figure 2.4.

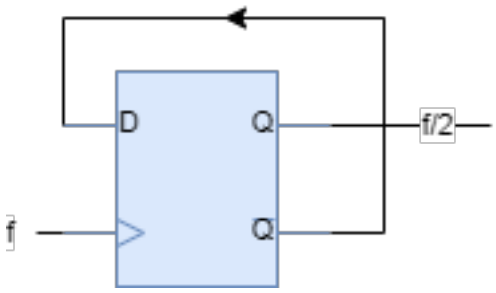

Figure 2.4. *Clock divider*

At device startup, the clock dividers flip-flop will get either the value '1' or '0'. When

simulating, this will be interpreted as an X. The X will be fed back to the input of the inverter, which will pessimistically propagate an X into the input of the flip-flop. This means, the divider will be stuck in a loop of X output. As the output of this circuit will be stuck as an X, the registers downstream controlled by the circuit in Figure 2.4 will be locked in an X-state in a pessimistic simulation. Physical design does not have this problem. [10]

2.6.3 Solutions

Some have suggested getting rid of X-values in simulation completely to remove the problem of dealing with X optimism and X-pessimism [10]. This would most probably lead to differences in simulation results and silicon behaviour since X-values tell the verification engineer that the simulator doesn't know if the value in silicon will become a '1' or a '0'. The appearance of an X indicates that something is wrong with the design with respect to ambiguity during simulation or silicon operation.

Omitting just some X-values in simulations would help in situations such as the circuit lockup discussed previously. Using SystemVerilog offers the possibility of mixing 2-state values and 4-state variables. The default value for uninitialized 2-state variables is '0', which doesn't represent the behaviour in silicon accurately. Powering up with all variables in the same value is extremely unlikely. Mixing 2- and 4-state values makes it possible to set the values that would otherwise erroneously lock up in 2-state values while keeping other values ambiguous. This is a more accurate representation the silicon behaviour. [10]

Some simulators offer a possibility of separating from the simulation standards of Verilog, VHDL, and SystemVerilog by using X-propagation rules specific to the simulator used. This is not offered by every simulator. These are the modes discussed in the previous chapter.

In some cases, it is possible to modify the design in a way that fewer X-values appear due to X-pessimism. One solution is to add resets to registers that don't need it, but as discussed before, that is not a very realistic choice due to size constraints in large designs, especially when using FPGAs. In some cases changing from synchronous to asynchronous reset will reduce X-pessimistic propagation-related problems. [19]

The testbench can be set to feed non-X-values into the design. As a universal solution this is not very realistic. For example, if the designer wants to simulate the design behaviour at device power up. This can be used to force values known to stay unambiguous to match the silicon behaviour.

The best solution would be to find the proper balance of pessimism and optimism that matches silicon behaviour. At the moment, it seems to demand too much computing resources to implement an exact model into RTL simulations. Alternatives for this are X-propagation tools which mimic the behaviour of silicon.

3 BASIC DESIGN SIMULATIONS

In this chapter, simple designs are simulated for X-propagation in different modes. The simulations are run with a UVM top level, using SystemVerilog, and DUTs written using VHDL. The purpose of the simple designs is to test behaviour of X-values in low-level cases.

Understanding how X-propagation occurs in simple designs helps the verification engineer tackle larger designs. Larger and more complex designs most likely include at least some if not most or all of the simpler designs. Minor differences in simple design simulations can cause major differences in large scale simulations.

Some special emphasis is given to state machines and different data types since they have been non-functioning in previous iterations of X-propagation simulation plugins. The state machine used in this thesis has logic within. These include: logic ports, such as AND, OR and XOR gates.

The random number generator, singe port RAM and shift register VHDL codes are based on the designs made by Meher Krishna Patel. [21] The designs themselves are not under evaluation.

3.1 Simulation setup 1 - If-else and Case statements

In the first simulation setup a multiplexer circuit is simulated. The simulation setup is the same as the examples in chapter 1. The if-else and case statements will be simulated to check if there are any discrepancies or unexpected differences compared to the results given in the user guides.

The circuit is fed data from the UVM testcase. The data set is very simple, hence the operation can be verified easily. Since the circuit is a multiplexer, the interesting signal considering uncertainty is the control signal. This is due to the fact that the control signal is the only signal used in all possible signal combinations.

A notable distinction between the if-else and case statement simulations compared to the other simulations in this chapter is that they are not synchronous simulations and include no registers. This means the focus is on the selection of input signals.

The different X-propagation simulation modes will be differentiated in the results of the simulations, similarly as in chapter 2. As the results of the simulations will most likely

follow along the lines of the user guide, the analysis will be lighter than the following simulations.

For the if-else statement the VHDL code is as follows.

```
entity if_else is

Port (s : IN std_logic;
      a : IN std_logic;
      b : IN std_logic;
      r : OUT std_logic);
end if_else;

architecture BEHAVIORAL of if_else is
    begin process (s, a, b)
        begin if (s = '1') then
            r <= a;
          else
            r <= b;
        end if;
    end process;
end;
```

Listing 3.1. *If-else statement.*

For the case statement the VHDL code is as follows.

```
entity X_CASE is

Port (s : IN std_logic;
      a : IN std_logic;
      b : IN std_logic;
      r : OUT std_logic);
end X_CASE;

architecture BEHAVIORAL of X_CASE is
    begin process (s, a, b)
        begin case s is
            when '0' => r <= a;
            when '1' => r <= b;
            when others => null;
        end case;
    end process;
end;
```

Listing 3.2. *Case statement.*

3.1.1 X-propagation with simulator A

Simulator A simulator is used to simulate the above circuits for X-propagation. For both statements, the most important signal is 's'. As this signal controls the output directly, an X-value here will affect the output the most. First, the designs will be simulated using known values in different sequences. After this has been done, the designs can be

simulated with X-values in different signals to see how the values affect the regular RTL simulation.

First, the if-else statement is tested with a sequence of known values to check that the design works properly and to get baseline results to which the X-value simulations can be compared. The effect of the input X-values on the output is examined separately and combined.

The normal operation simulation is done by feeding different combinations of signals into the DUT in sequence. The if-else statement simulation is done by feeding signal combinations into the DUT, where at least one of the signals is ambiguous. The input and output waveforms for the known-value simulation and X-value simulations without the X-propagation plugin, in resolve mode, and in pass mode for the if-else statement are represented in figure 3.1. The first part of the figure is the normal operation simulation, the second part is the X-value simulation, the third is the resolve mode simulation, and the last is the pass mode simulation. In the waveform, picture 's' is the control signal, 'a' and 'b' are the input signals, and 'r' is the output signal.

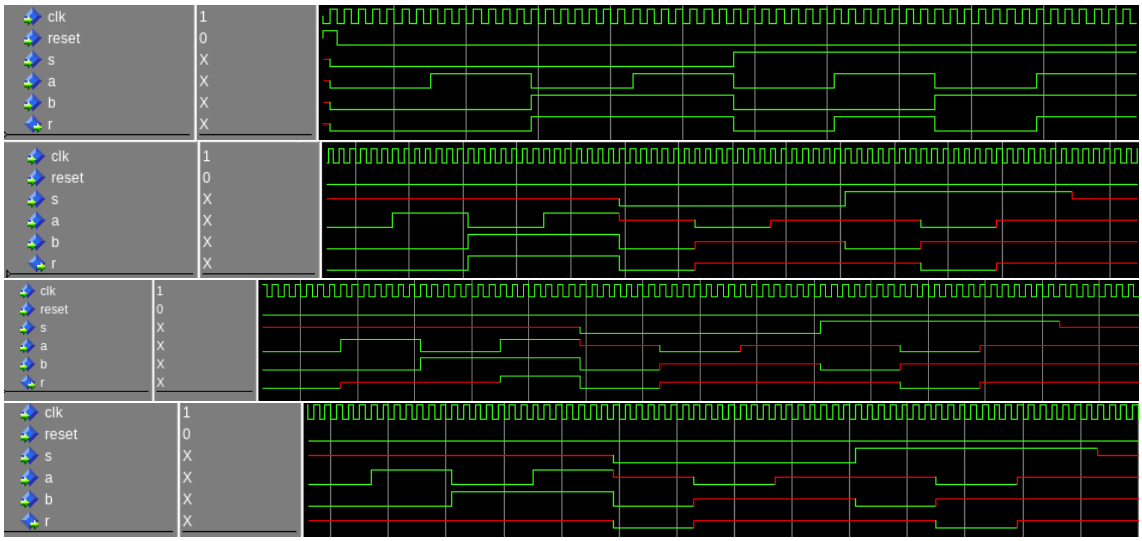

Figure 3.1. *If-else statement waveform in normal and X-propagation plugin operation.*

The simulation results are combined in tables 3.1, 3.2 and 3.3 separated by the value of the 's' signal.

Table 3.1. *If-else statement simulation results, s=X*

Normal				Resolve				Pass			
s	a	b	r	s	a	b	r	s	a	b	r
X	0	0	0	X	0	0	0	X	0	0	X
X	1	0	0	X	1	0	X	X	1	0	X
X	0	1	1	X	0	1	X	X	0	1	X
X	1	1	1	X	1	1	1	X	1	1	X

Table 3.2. *If-else statement simulation results, s=0*

Normal				Resolve				Pass			
s	a	b	r	s	a	b	r	s	a	b	r
0	X	0	0	0	X	0	0	0	X	0	0
0	0	X	X	0	0	X	X	0	0	X	X
0	X	X	X	0	X	X	X	0	X	X	X

Table 3.3. *If-else statement simulation results, s=1*

Normal				Resolve				Pass			
s	a	b	r	s	a	b	r	s	a	b	r
1	X	0	X	1	X	0	X	1	X	0	X
1	0	X	0	1	0	X	0	1	0	X	0
1	X	X	X	1	X	X	X	1	X	X	X

As seen from the first part of figure 3.1, the if-else statement works as expected. When the control signal ’s’ is low, the output is dictated by the input signal ’b’, and when the control signal is high, the output signal is dictated by input ’a’.

When examining the second part of figure 3.1, we can see how the X-values in the input affect the output differently with respect to each other. When the control signal ’s’ is set as X, the simulator interprets it as low. When the control signal is low, the output is set as the same as the ’b’, and when ’s’ is high, the output is set as ’a’.

When running the if-else simulation in trap mode, the simulation waveform does not differ from the non-X-propagation plugin simulation. This was not a surprise, as this was implied in the user guide. [14] When running the resolve mode, the simulation part with no X-values introduced can be omitted since it is unchanged from the non-X-propagation plugin simulation.

When examining the third waveform in figure 3.1, it can be seen that the resolve mode has changed how the simulator handles X-values. When the control signal ’s’ is X, the output value is set to X when the input values are not the same, and the correct value can’t be determined. When the control signal is 1 or 0, the output value is set as the represented input value.

The effect of the pass mode is evident when examining the last part of figure 3.1. The output of the circuit stays as X when the control signal is X. When the control signal can be determined, the output acts in the same way as in the previous simulation.

The simulations in the last two parts in figure 3.1 show that the X-propagation plugin simulations for the if-else statement work the same way as stated in the user manual [14].

The input and output waveforms for the known-value, X-value, resolve mode and pass mode simulations of the case statement are represented in figure 3.2. The X-value simulation for the case statement is done the same way as the if-else statement. In the waveform, figure 's' is the control signal, 'a' and 'b' are the input signals, and 'r' is the output signal.

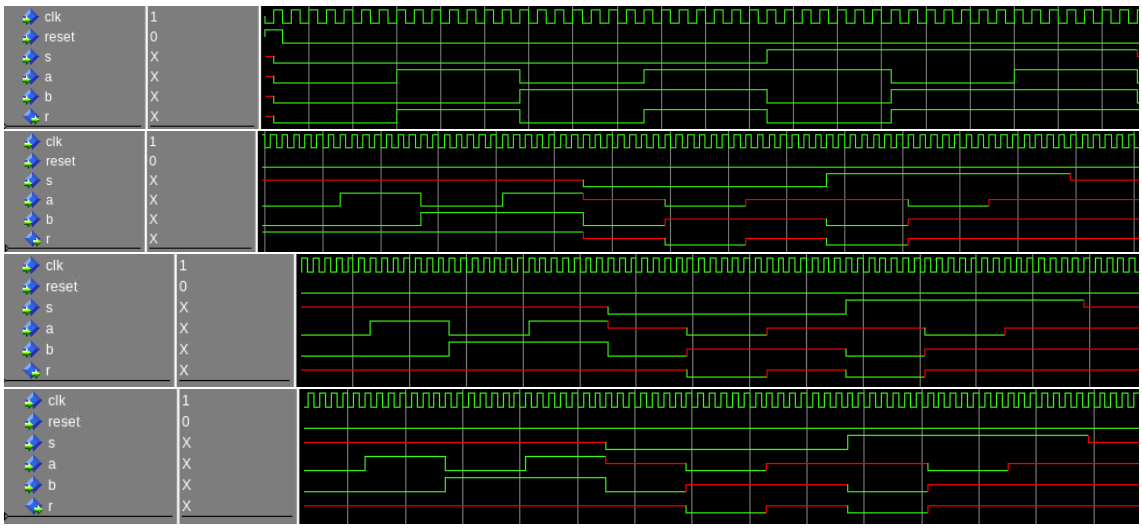

Figure 3.2. *Case statement waveform in normal and X-propagation plugin operation.*

The simulation results are combined in tables 3.4, 3.5 and 3.6 separated by the value of the 's' signal.

Table 3.4. *Case statement simulation results, s=X*

Normal				Resolve				Pass			
s	a	b	r	s	a	b	r	s	a	b	r
X	0	0	1	X	0	0	X	X	0	0	X
X	1	0	1	X	1	0	X	X	1	0	X
X	0	1	1	X	0	1	X	X	0	1	X
X	1	1	1	X	1	1	X	X	1	1	X

Table 3.5. *Case statement simulation results, s=0*

Normal				Resolve				Pass			
s	a	b	r	s	a	b	r	s	a	b	r
0	X	0	X	0	X	0	X	0	X	0	X
0	0	X	0	0	0	X	0	0	0	X	0
0	X	X	X	0	X	X	X	0	X	X	X

Table 3.6. *Case statement simulation results, s=1*

Normal				Resolve				Pass			
s	a	b	r	s	a	b	r	s	a	b	r
1	X	0	0	1	X	0	0	1	X	0	0
1	0	X	X	1	0	X	X	1	0	X	X
1	X	X	X	1	X	X	X	1	X	X	X

As seen from figure 3.2, the case statement works as expected. When the control signal 's' is low, the output is dictated by the input signal 'b', and when the control signal is high, the output signal is dictated by input 'a'. Examining the second part of figure 3.2, we can see how the X-values at the input affect the output. As seen in the figure, the output is dictated by control signal 's', and when it is not set as X. When the control signal is X, the output defaults to high. When the control signal is low, the output is set as the 'a' signal, and when high, the output is set as the 'b' signal. Comparing the resolve mode operation of the case statement in the third part of figure 3.2 and the non-X-propagation plugin simulation in the second part of figure 3.2, the difference in output is only seen when the control signal is X. While the control signal is X, the output is X. The case statement X-propagation plugin simulation in pass mode shown in the final part of figure 3.2. Surprisingly shows no difference to the resolve mode simulation.

As with the if-else statement, the trap mode on the case statement does not affect the waveform. This is in line with the user guide, so it is working as expected. [14]

3.2 Simulation setup 2 - Random number generator

The second simulation setup to be tested is a random number generator. The circuit was implemented using a linear feedback register. As is often the case, this is a pseudo-random number generator. The feedback value in the code is determined by a polynomial function which, in turn dictates the length of the pseudo-random number chain. In the case tested, the output chain is 15 numbers long. However, the length is irrelevant when testing the X-propagation properties of the design.

There are only a limited number of input signals going into the circuit. This means that only a small amount of X-values come from outside of the circuit. Since the circuit is synchronous, there are registers in the designs which may cause ambiguity on startup or brownout if not reset properly.

The VHDL code of the random number generator is as follows.

```
entity rand_num_generator is
    port(
      clk, reset  : in std_logic;
      q : out std_logic_vector(4 downto 0) -- output
    );
end rand_num_generator;

architecture arch of rand_num_generator is
    signal r_reg, r_next : std_logic_vector(4 downto 0);
    signal feedback_value : std_logic;
begin
    process(clk, reset)
    begin
        if(reset='1') then
            -- set initial value to '1'.
            r_reg(0) <= '1';
            r_reg(4 downto 1) <= (others=>'0');
        elsif (clk'event and clk='1') then
            r_reg <= r_next;
        end if;
    end process;

    feedback_value <= r_reg(4) xor r_reg(3) xor r_reg(0);
    r_next <= feedback_value & r_reg(4 downto 1);
    q <= r_reg;
end arch;
```

Listing 3.3. *Random number generator. [21]*

When reviewing the code, it can be seen that the design has only a clock and a reset as its inputs. This means that, at least in X pessimistic cases, there is a high risk of the simulation getting stuck in a loop of X-values.

3.2.1 X-propagation with simulator A

Simulator A is used to simulate for X-propagation. First the normal operation of the design is verified by simulating it without any abnormal behaviour induced. The registers in the design are all reset and have a reset value. This means no X-values should be found when the simulation is running.

In the normal operation simulation, the random number generator is left to run freely. The DUT is set to run for 13 cycles within one sequence. The input and output waveforms for the normal operation of the random number generator are represented in the first part of figure 3.3. In the picture 'q' is the output of the circuit.

The X-value simulation is done by changing the functionality of the reset of the registers. The standard operation of the reset sets the lowest order bit of the register feeding to the output as '1' and others as '0'. When the value of the lowest order bit of the output register is not defined, it will default to X. The waveform of the simulation is represented

in the second part of figure 3.3.

When the output register does not have any reset value, all the bits in the register will default to X. In this case, the reset is not used by the DUT. The waveform of the simulation without reset values for the output register is represented in the final part of figure 3.3.

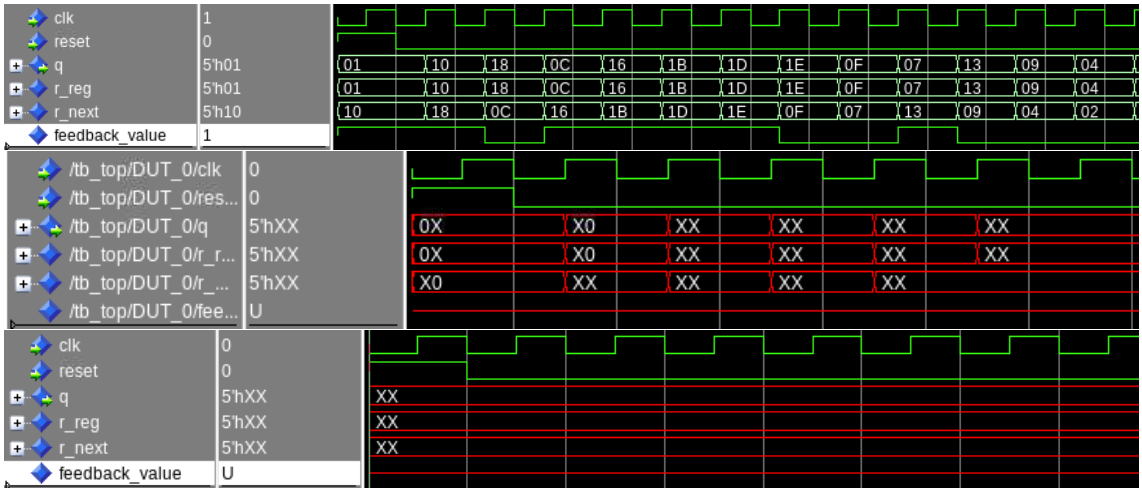

Figure 3.3. *Random number generator in normal and X-value operation without the X-propagation plugin.*

From the first part of figure 3.3, we can see how the output value changes randomly without any input other than the clock and reset. The sequence repeats after the maximum number of values is seen in the output dictated by the feedback polynomial. In this case the maximum cycles in a sequence is 13.

In the second part of figure 3.3, the output of the DUT does not work as intended and does not recover. The initial X-value moves forward and is fed back into the circuit, which cascades to fill the whole system with ambiguity.

As seen from the final part of figure 3.3, the operation of the circuit is quite simple. The lack of reset values in these kinds of circuits, without any data input, tends to get stuck in a loop. This kind of circuit needs some sure way to recover from this loop or initial values that no X-values are introduced to the system.

The waveform of the resolve and pass mode simulations of the random number generator simulation in resolve mode is represented in figure 3.4. In the first two parts of the figure, resolve and pass mode simulations are run normally without any additional X-value inputs. In the two final parts, resolve and pass mode simulations are run without any initial reset values.

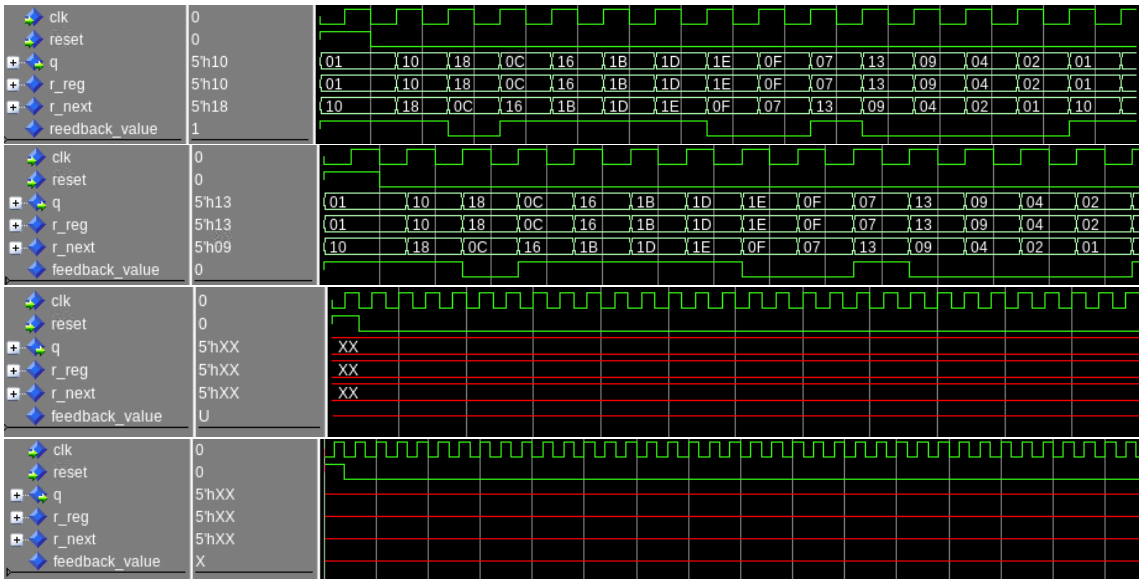

Figure 3.4. *Random number generator in resolve and pass mode X-propagation plugin simulation.*

When examining the waveform in the first and second parts of figure 3.4, it can be seen that the resolve mode has not changed how the circuit operates significantly.

As seen from the third part of figure 3.4 introducing X-values to the circuit without a reset makes the operation unreliable. This is the case in the final part of the figure as well.

3.3 Simulation setup 3 - State machine

The third simulation setup to be tested is a state machine design. The circuit is composed of different states in which different kinds of logic are implemented within the states. The logic is kept simple so the effects of the X-values can be examined.

The input sequence data fed to the circuit is kept the same regardless of the state to keep the analysis of the results more streamlined. This allows each of the states to be analyzed the same way as others, and the differences between states are clearer. The results from each logic port are compiled into tables the same way as before, so their X-propagation properties are tested.

The input sequence data fed to the circuit is kept the same regardless of the state to make the analysis of the results more streamlined. This allows each state to be analyzed the same way as others, and the differences between states are clearer. The results from the logic ports are compiled into tables the same way as before, so their X-propagation properties are tested.

The data is fed to the DUT by a UVM testcase, and the form of the input data itself is defined in the UVM sequence item fed to the sequencer. First, the normal operation of the state machine must be verified to be compared to the X-value operation.

The VHDL code of the state machine is as follows.

```
entity state_machine is
  port(
    clk: in std_logic;
    reset : in std_logic;
    we : in std_logic;
    din : in std_logic_vector(3 downto 0);
    dout : out std_logic_vector(3 downto 0)
    );
end state_machine;

architecture arch of state_machine is

  -- Type declaration and state signal declaration
  type t_State is (State_1, State_2,
                   State_3, State_4,
                   state_5);
  signal State : t_State;
  signal dout_buff : std_logic_vector(3 downto 0);

begin
  process(clk)
  begin
    if (clk'event and clk='1') then
      if reset = '1' then
        --reset values
        dout_buff <= "0000";
        State <= State_1;
      else
      --default values
        dout_buff <= "0000";
        case State is
          when State_1 =>
            dout_buff <= "0001";
            if we = '1' then
              dout <= dout_buff;
              State <= State_2;
            else
              dout <= "0000";
            end if;

          when State_2 =>
            dout_buff <= "0010" AND din;
            if we = '1' then
              dout  <= dout_buff;
              State <= State_3;
            else
              dout <= "0000";
            end if;

          when State_3 =>
            dout_buff <= "0100" OR din;
            if we = '1' then
```

```
            dout  <= dout_buff;
            State <= State_4;
          else
            dout <= "0000";
          end if;

        when State_4 =>
          dout_buff <= "1000" XOR din;
          if we = '1' then
            dout  <= dout_buff;
            State <= State_5;
          else
            dout <= "0000";
          end if;

        when State_5 =>
          dout_buff <= "1001" XNOR din;
          if we = '1' then
            dout  <= dout_buff;
            State <= State_1;
         end if;
      end case;
    end if;
  end if;
 end process;
end arch;
```

Listing 3.4. *State machine.*

In addition to the clock and reset signals, the state machine has two inputs, 'we' write enable and 'din' data input. The circuit output is the 'dout' data output signal. In normal operation, the output depends on the input data 'din' and the state in which the state machine is in. In each state, excluding state 1, the 'dout' signal is obtained by passing the 'dout' of the previous state and the 'din' through a logic gate specific to the state. The state machine operators are assorted in table 3.7.

Table 3.7. *State machine operators*

State_1	State_2	State_3	State_4	State_5
N/A	AND	OR	XOR	XNOR

3.3.1 X-propagation with simulator A

Simulator A simulator is used to simulate the circuit for X-propagation. First, the normal operation simulation of the design without X-values is run to see how the circuit behaves. The registers within the circuit are reset and have an assigned default reset value. No X-values should be seen in the simulation while it is running. First, the input data is set to one value for the duration of one cycle and then running from 0 to 7 for the second cycle.

In the normal operation simulation the ’we’ signal is toggled low and high while holding a single input value in ’din’. This is done to go through all the states of the state machine. The waveform of the normal operation simulation is represented in figure 3.5.

The first part of the X-value simulations is done by setting ’din’ as X and switching the write enable signal high and low. This is run for eight iterations from 0 to 7. The waveform of the first X-value simulation is represented in the second part of figure 3.5.

In the second part of the X-value simulations, the write enable signal is set as X, and the input data is set from 0 onward. Similar to the previous simulation, it is run for eight iterations from 0 to 7. The waveform of the second X-value simulation is represented in the third part of figure 3.5.

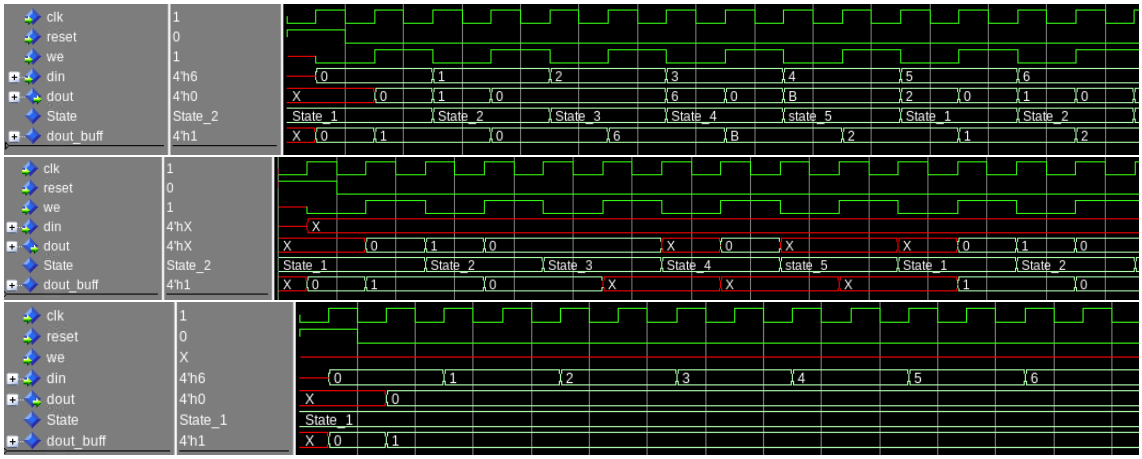

Figure 3.5. *State machine simulation without X-propagation plugins.*

As seen in the first part of figure 3.5, the state of the state machine changes consistently, and the ’dout’ signal changes value according to the design. The

From the second part of figure 3.5, it can be seen that an X-value in the input signal propagates to the output in states 3, 4 and 5. The logic in these states are the OR, XOR and XNOR gates. Since there is no other requirement for the state to move forward than ’we’ being high, the state machine moves through all states in sequence.

From the third part of figure 3.5, it can be seen that the X-value of the write enable signal is interpreted by the simulator as a ’0’ with respect to the state moving forward. This results in the simulation getting stuck at state 1 and the output set as ’0’.

The waveform of the state machine simulation in resolve mode with valid input values is represented in figure 3.6. The waveform of the state machine simulation in resolve mode with valid write enable and data input as X is represented in the second part of the figure. The waveform of the state machine simulation in resolve mode with valid input data and write enable as X is represented in the third part of the figure.

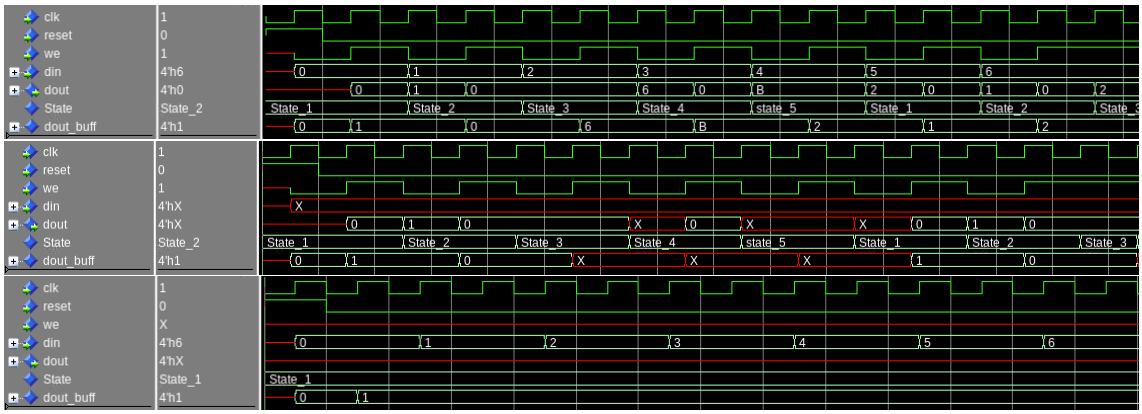

Figure 3.6. *State machine in resolve mode X-propagation simulation.*

Looking at the waveform in figure 3.6 the resolve mode does not affect the simulation when the input values are not X. The effect of the X value in the input data is that the state of the state machine moves forward, but the output value is sometimes set as X. This simulation has no difference compared to the non-X-propagation plugin simulations. Looking at figure 3.6 it is clear that the write enable signal is critical to the operation of the state machine, and the state machine does not move forward when the control signal is unknown.

The waveform of the state machine simulation in pass mode, is represented in figure 3.7. In the first part of the figure, valid input values are used, in the second, the data input is 'X', and in the third, the write enable is 'X'.

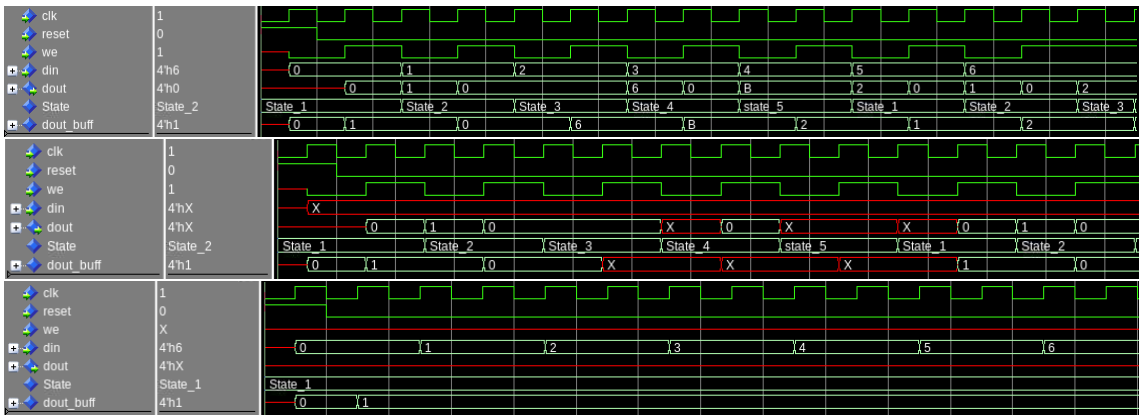

Figure 3.7. *State machine in pass mode X-propagation simulation.*

The effect of the pass mode cannot be determined, with the input values being valid as the first parts of figures 3.5 and 3.7 are the same. When comparing the second parts of figures 3.6 and 3.7, there is no difference in the output of the circuit, as is the case when comparing the second part of figure 3.5. As with the previous simulations, the third part of figure 3.7 shows that the state machine does not operate with an X-value write enable signal.

In the next set of simulations every other value of write enable is X, and every other value is valid going to 1 and 0, respectively. The simulation waveform of the trap mode

simulation is represented in the first part of figure 3.8. The second part is the resolve mode simulation, and the third is the pass mode simulation.

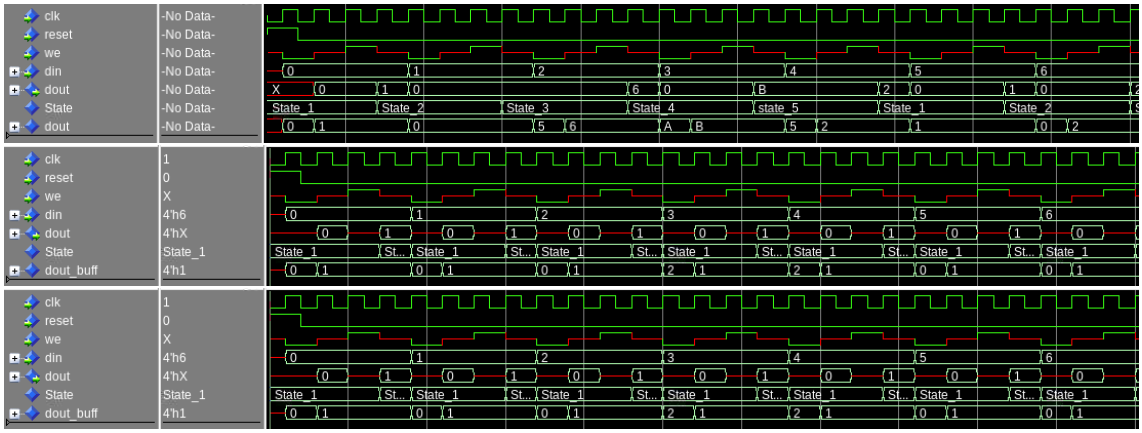

Figure 3.8. *State machine simulation waveform with variable input.*

When comparing the first part of figure 3.8 and the second part of figure 3.5, the states of the state machine are moving forward as before, but the output operates differently, which can be explained by the extra X values in between the valid values. The X values in the 'we' input cause the output buffer 'dout_buff' to exhibit additional values. When comparing the second and the third parts of figure 3.8 the difference in simulation results is quite drastic. The state no longer moves forward past the second state, and the X-values propagate through to the output of the circuit.

When comparing all of the parts of figure 3.8, the results between the last two simulations are identical. More in-depth analysis of each state's logic will explain the differences and similarities between simulations.

3.4 Simulation setup 4 - Random Access Memory

The fourth simulation setup to be tested is a single port RAM circuit. The circuit is composed of an array of memory cells that are used to store and retrieve data accessed with reading and writing commands. The amount of storable data is determined by the memory width.

The VHDL code of the RAM circuit is as follows.

```
entity single_port_RAM is
  generic (
    addr_width : integer := 2;
    data_width : integer := 3
  );

  port(
    clk : in std_logic;
    we : in std_logic;
    addr : in std_logic_vector(addr_width-1 downto 0);
    din : in std_logic_vector(data_width-1 downto 0);
    dout : out std_logic_vector(data_width-1 downto 0)
    );
end single_port_RAM;

architecture arch of single_port_RAM is
    type ram_type is array (2**addr_width-1 downto 0)
        of std_logic_vector (data_width-1 downto 0);
    signal ram_single_port : ram_type;
begin
  process(clk)
  begin
    if (clk'event and clk='1') then
      if (we='1') then
        ram_single_port(to_integer(unsigned(addr))) <= din;
      end if;
  end if;
  end process;

  dout<=ram_single_port(to_integer(unsigned(addr)));
end arch;
```

Listing 3.5. *Single port RAM. [21]*

The most interesting signals of the circuit are the write-enable 'we', data out 'dout', and address 'addr' signals. Write enable controls the writing of the input data into the RAM array, while the 'addr' signal designates the address to be read and written.

The target of this simulation is to see if the simulator retains the X-values inside the memory cells and how the different control signals affect the function of the circuit. This means that the size of the memory doesn't have an effect on the result of the simulation. The RAM cells do not have a reset value, meaning they will start in an X-state.

Additionally, the ambiguity of the write-enable signal is tested to see how the simulation handles it and how the circuit acts with an unknown control. This is especially interesting since the read and write commands use the same signal. This will cause a gap of one cycle before the output changes when changing addresses.

3.4.1 X-propagation with simulator A

Like previously, the first simulations for this circuit are done with simulator A. In normal operation simulations, data is fed into the design and stored to see how the control and data signals act in normal situations. Since the memory array is not a very complex design, it doesn't have too many moving parts.

In the normal operation simulation, the input 'din' is set to values from 0 to 2 for each of the registers, while the write enable is high for the first half and low for the second half. The first half of the waveform of the normal operation simulation of the RAM circuit is represented in the first part of figure 3.9.

In the second half of the normal operation simulations, the 'addr' signal is set as '0' and 'din' in sequences from 0 to 2 as before. The second half of the normal operation simulation is represented in the second part of figure 3.9.

When testing how the standard simulation of the RAM circuit handles X-values, all relevant input and control signals are set as X in sequence. In the first half of the X-value tests in standard simulation, the 'we' and 'addr' signals are set as X the effect of those is studied. The first part of the waveform of the X-value simulation is represented in the third part of figure 3.9. The second part of the waveform of the X-value simulation is represented in the fourth part of figure 3.9.

Repeating the previous simulation but initially writing valid values in all of the cells of the RAM array. This is done to see how the same input affects the DUT and output differently when initial values are valid. The initialization of the DUT Cells is omitted from the figures to make the figures more compact. The first part of the initialized the X-value simulation is represented in the fifth part of figure 3.9. The second part of the waveform of the X-value simulation is represented in the last part of figure 3.9.

Figure 3.9 shows the circuit in normal operation, when the 'we' signal is high. In the simulation, it can be seen how the X-values are replaced in the RAM array and how the output value is dictated by the value inside the memory and the 'addr' signal. The read address is the same as the write address, and this causes the first 'dout' value for each 'addr' iteration to be X. This would not be the case if the RAM cells had initial values.

The second part of figure 3.9 shows that when the 'we' signal is low, no data is written into the memory array. No data is read from the array either, leaving 'dout' as an X. While the input data and address signals are not X, the data and the address are not used by the DUT because the 'we' signal is low.

From the third part of figure 3.9, it can be seen that, as expected, when the 'we' signal is X, the simulator does not overwrite the X-values within the memory array with valid values. When the 'we' signal is set high and the 'addr' signal as X, the simulator interprets the address as '0' as seen in the last three clock cycles in the third part of figure 3.9.

From the fourth part of figure 3.9, it can see that, as expected, an X at the input will be

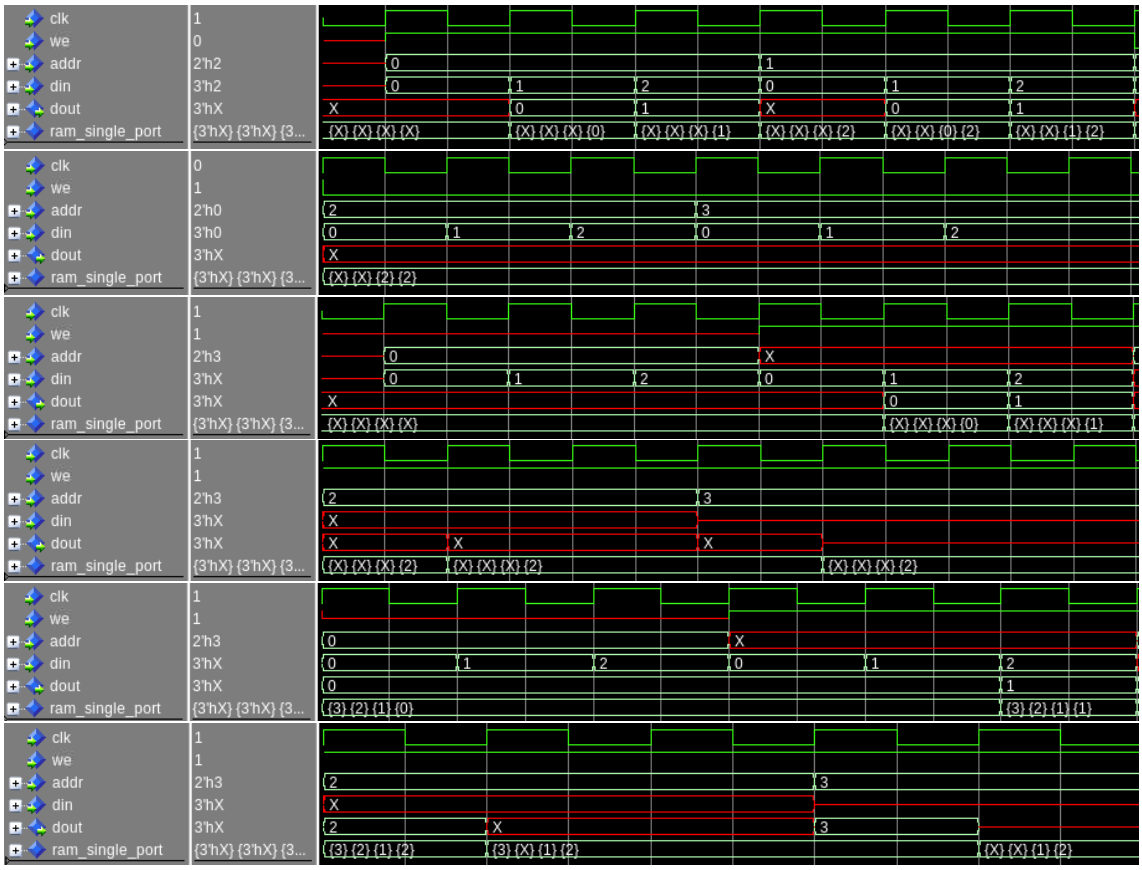

Figure 3.9. *RAM simulation without X-propagation plugins.*

written into the memory and read to the output. In this case, an X-value at the input 'din' will be written into the memory address dictated by the 'addr' signal. As in the uninitialized simulation, when the 'we' signal is 'X', the simulator interprets it as '0'. The situation stays the same with the 'addr' signal.

As seen from the last part of figure 3.9, the X-values are written into the RAM cells as previously. The only difference to the uninitialized RAM is that the initialized data is read to the output when the address is changed. The output changes to 'X' as the RAM cell is overwritten.

The waveforms of the single port ram circuit simulations in resolve mode are represented in figure 3.10. The first X-propagation plugin simulation is done with initialized registers and valid input values. The X-value simulation is done using the same sequence as the simulation represented in the last two parts of figure 3.9. The first part of the waveform of the resolve mode simulation, where the RAM circuit is written into filling every cell and read from respectively, is represented in the second part of figure 3.10. Note that the initialization sequence is seen in this figure. The second part of the resolve mode X-value simulation is represented in the final part of figure 3.10.

When examining the waveform in the first part of figure 3.10, it can be seen that the resolve mode has not changed how the simulation runs. When comparing the simulations in the third and fourth part of figure 3.9 and the final two parts of figure 3.10, a difference

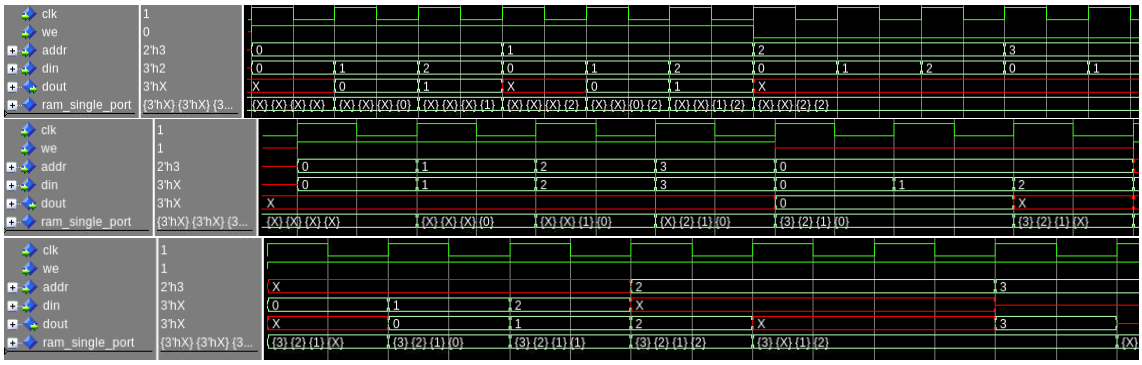

Figure 3.10. *RAM simulation in resolve mode X-propagation.*

in the output of the circuit appeared when reading from the circuit.

The waveform of the RAM circuit simulation in pass mode and normal input values is represented in the first part of figure 3.11. The first part of the simulation waveform in pass mode and X input values is represented in the second part of figure 3.11. The second part of the simulation waveform in pass mode and X input values is represented in the final part of figure 3.11.

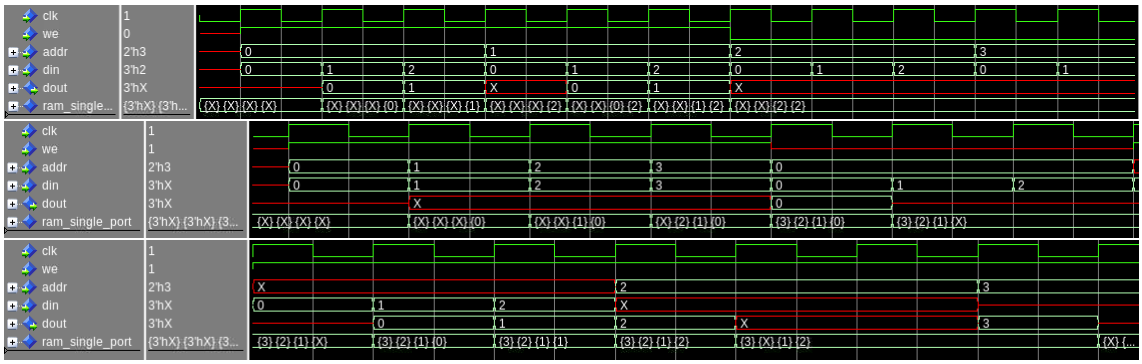

Figure 3.11. *RAM simulation in pass mode X-propagation.*

The waveform in figure 3.11 is the same as the normal simulation with no changes in the waveform. The waveforms in the second and third parts of figure 3.11 are no different from the waveforms in the second and third parts of figure 3.10.

3.5 Simulation setup 5 - Shift register

The last simulation setup is a shift register circuit. The circuit is composed of successive registers that shift a bit in one direction or the other, and in this case, the shift register is bidirectional.

This simulation is done to see how X-values propagate through multiple registers and if it is affected by timing. Additionally, the delay between the input and output of the shift register is interesting.

Two interesting signals can affect the operation of the circuit. In this case, the interesting signals are the input signal ’data’ and the control signal ’ctrl’. The input data signal is

interesting because that is the data that needs to propagate through the shift register, and the control signal controls how the data moves in the shift register.

The operation of the shift register is controlled by a control signal. The shifting, loading, and reading done by the circuit are all controlled by the control signal. Interesting about this is the possibility of changing the operation of the whole circuit if the control signal has an X-value instead of a logic ’1’ or a ’0’.

The VHDL code of the shift register is as follows.

```
entity shift_register is
    port(
        clk, reset      : in std_logic;
        ctrl            : in std_logic_vector(1 downto 0);
        din            : in std_logic_vector(8-1 downto 0);
        dout           : out std_logic_vector(8-1 downto 0)
    );
end shift_register;

architecture arch of shift_register is
    signal s_reg, s_next : std_logic_vector(8-1 downto 0);
begin
    process(clk, reset)
    begin
        if(reset='1') then
            s_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
            s_reg <= s_next;
        end if;
    end process;

    process (ctrl, s_reg, din)
    begin
        case ctrl is
            when "00" =>
                s_next <= s_reg;
            when "01" =>
                s_next <= din(8-1) & s_reg(8-1 downto 1); -- right shift
            when "10" =>
                s_next <= s_reg(8-2 downto 0) & din(0); -- left shift
            when others =>
                s_next <= din;
        end case;
    end process;

    dout <= s_reg;
end arch;
```

Listing 3.6. *Shift register implementation in VHDL. [21]*

3.5.1 X-propagation with simulator A

As in the previous simulations, the simulations are done using simulator A. The simulations are started by simulating the design in normal operation, followed by simulations with X-values.

As in the previous design, there are only a few moving parts to the design. This means the simulations will remain simple. In the normal operation, the data is fed into the register, shifted right and left and kept still. The waveform of the standard operation simulation of the shift register is represented in the first part of figure 3.12.

At the beginning of the X-value simulation, the control signal and some bits of the input data are set as X respectively. Additionally, the X values inside the shift register are subjected to shift commands from the control signal. The beginning of the waveform of the X-value simulation is represented in the second part of figure 3.12.

Individual bits in the control signal are set as X in the latter half of the simulation. This is to see how the simulation is affected by ambiguity in the control signal. First, the least significant bit (LSB) and afterwards, the second from LSB is set as 'X' while the other is '0'. Finally, the same is repeated with the valid value being '1'. The end of the waveform of the X-value simulation is represented in the third part of figure 3.12.

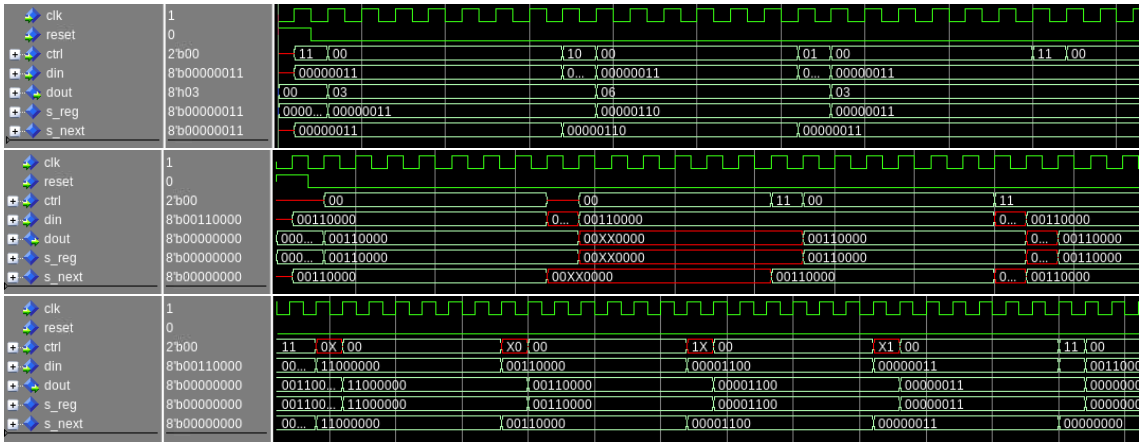

Figure 3.12. *Shift register simulation without X-propagation plugins.*

Table 3.8. *Shift register commands without the X-propagation plugin*

	00	10	01	11	
Effect	N/A	L-Shift	R-Shift	Write	
	0X	X0	1X	X1	XX
Effect	Write	Write	Write	Write	Write

From figure 3.12, it can be seen that the control signal moves the bits inside the register left and right according to the value it has. '00' keeps the data still and does not modify it, '10' shifts the data left, '01' moves the data right, and 'others' saves new data into the

shift register overwriting the data already in the register.

From the second part of figure 3.12, it can be seen that although the initial value of the control signal is X, the input data is written into the register since the writing happens when other than ’00’, ’01’ or ’10’ is written into the control signal. Interestingly, input data is written into the register, though the control signal is ambiguous. Previously the simulator has interpreted the X-values as zeros but here it is interpreted as something else. Next, it can be seen that the shift commands of the control signal work normally even though the data inside the register is ambiguous. Also, ambiguous values are written to the register if fed to the input as expected.

From the third part of figure 3.12, it can be seen that in this case, when the high bits in left and right shift commands are replaced with ambiguous values, the data inside the shift register does not move but the input data is written into the DUT. This means that the simulator interprets the control data with X-values as ’other’ and not low as in some other cases. The same happens when the valid value of the two is low.

The waveform of the shift register simulation in resolve mode is represented in figure 3.13. The sequence of the simulation is the same as in the first shift register simulations in figure 3.12. The waveform of the shift register simulation in resolve mode is represented in the second and third parts of the figure. The sequence used is the same as in the second and third parts of the figure.

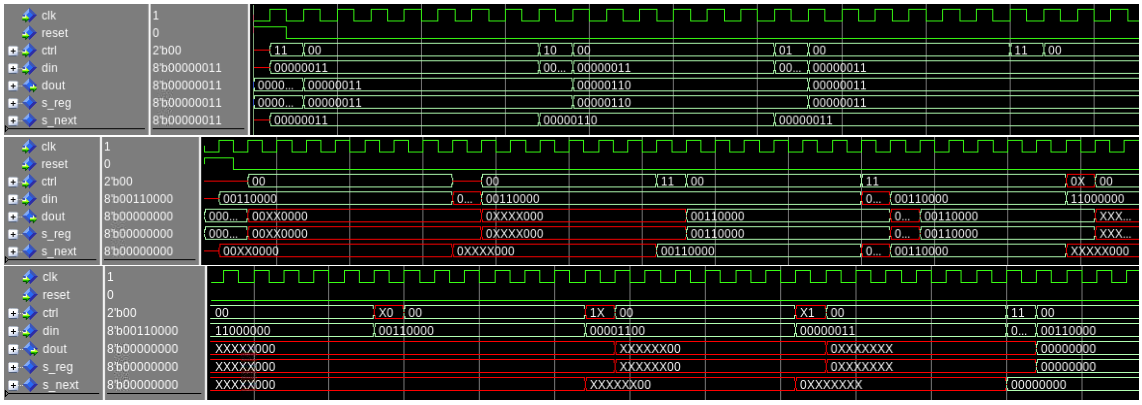

Figure 3.13. *Shift register simulation in resolve mode simulation.*

When examining the waveform in the first part of figure 3.13, it is clear that the resolve mode has not changed how the circuit operates significantly. The waveforms in figures 3.12 and 3.13 are the same since no X-values are introduced. In the second part of figure 3.13, there is a change at the beginning of the waveform. The First value going into the buffer ’s_next is read from the X input, so it overwrites the initial value of the output. In the third part of figure 3.13, it can be seen that resolve mode has changed the way the simulator handles X-values significantly. The X-values in the control signal are the ones that affect the operation of the circuit the most.

The waveform of the shift register simulation pass mode is represented in figure 3.14. The sequence used is the same as previously.

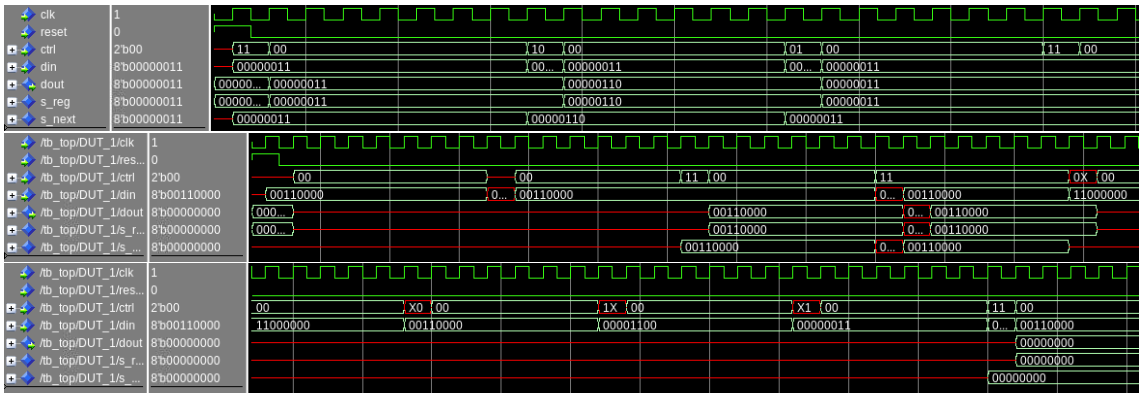

Figure 3.14. *Shift register simulation in pass mode simulation.*

As seen from the first part of figure 3.14, the pass mode and resolve mode in figure 3.13 have the same simulation results. This means no X-values are produced by the circuit. The effect of the pass mode is evident when examining the second and third parts of figure 3.14. The amount of X-values going through to the output is drastically increased, and only two valid values get through.

4 LARGE DESIGN SIMULATIONS

Simulating X-propagation in large designs and verifying the usability of the simulator plugins was the end goal of this thesis. X-propagation simulations for larger designs are time and resource demanding, so they should be used more sparingly to locate possible problems. Determining the right place to look for when testing X-propagation can be difficult, especially if the X-value sources are unknown. The amount of time used per simulation may be multiple times longer compared to a normal simulation.

The whole design will not be simulated, and all possible testcases covered since that would take prohibitively long. Many designs may include loops that will lock the simulation until it is terminated. These problems can be compounded when trying to simulate unfamiliar designs. The focus will be on a few familiar testcases. This will enable the comparison between regular RTL and X-propagation RTL simulations.

One important property to analyze is the tendency of an X-value to propagate through the different sections of the design and how long those chains can be. The chains might branch off or be cut short and should accounted for. Following the chains will show how far back X-related problems can arise and the usefulness of X analysis.

Since large parts of the simulated design might be unavailable for the verification engineer, cooperation between designers and other verification engineers is highly recommended. The compartmentalization of the design and verification work also means that in large projects, a single verification engineer probably cannot understand the workings of the whole project. In any case, the verification engineer should have a working knowledge of the design under verification to be able to compare the results properly.

4.1 X-propagation for larger designs

In addition to the difference in scale, the main difference between the larger and more complex designs compared to simple designs is the propagation of X from one IP block to the next. This can cause problems far down the line in the design. In these larger designs, the focus is not so much on how the X-values propagate through single simple components but on how they propagate through larger block designs.

Using RTL simulation on these designs will also take into account the possible logic between IP blocks. This allows the simulation to correctly propagate X-values from the input to the output. This allows for the X-values path to be traced back to its source

without any gaps in the logic.

4.2 Simulation setup

The simulation setup used in the simulations is the same as the company uses on its FPGA project. The FPGA project is a 5G L1 baseband FPGA chip design. This setup is not flexible for changing simulators and is set up to be run with simulator A.

4.2.1 X-propagation with simulator A

In this section the simulation results of X-propagation simulations done with simulator A are gathered. The results are always compared to simulations without the X-propagation plugin enabled. Initially a sanity check simulation is run with both the X-propagation plugin disabled and enabled and the possible differences or problems noted for later analysis.

Simulator A is used to run a sanity check on the design without any X-propagation plugins. Running a basic sanity check should pass without incident. The waveform of the sanity check is represented in figure 4.1.

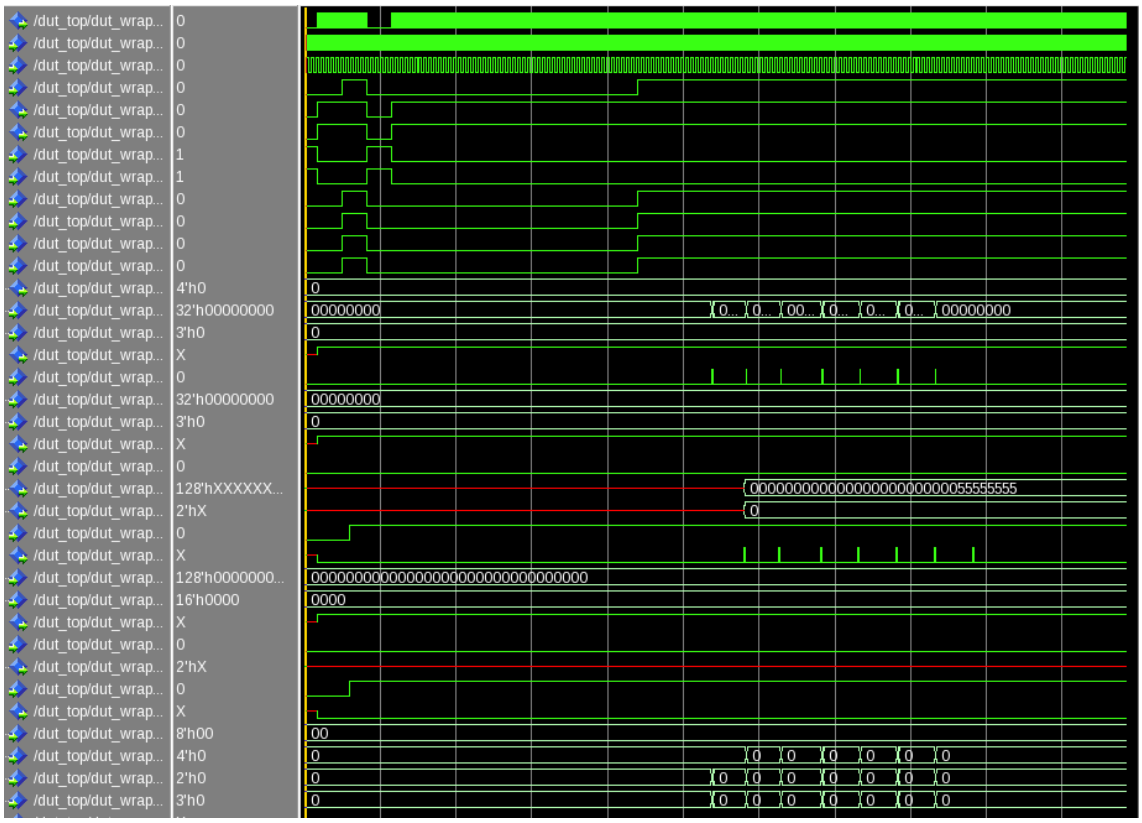

Figure 4.1. *Sanity check waveform in normal simulation*

Figure 4.1 shows the waveform of a passed sanity check for the design without errors. This is a good platform to start trialing the X-propagation plugin functionality.

Running the sanity check with the X-propagation plugin enabled in resolve mode caused the simulation to show warnings of signals going to X. Eventually, the sanity check throws a fatal error and the simulation stops. The waveform of the sanity check is represented in figure 4.2.

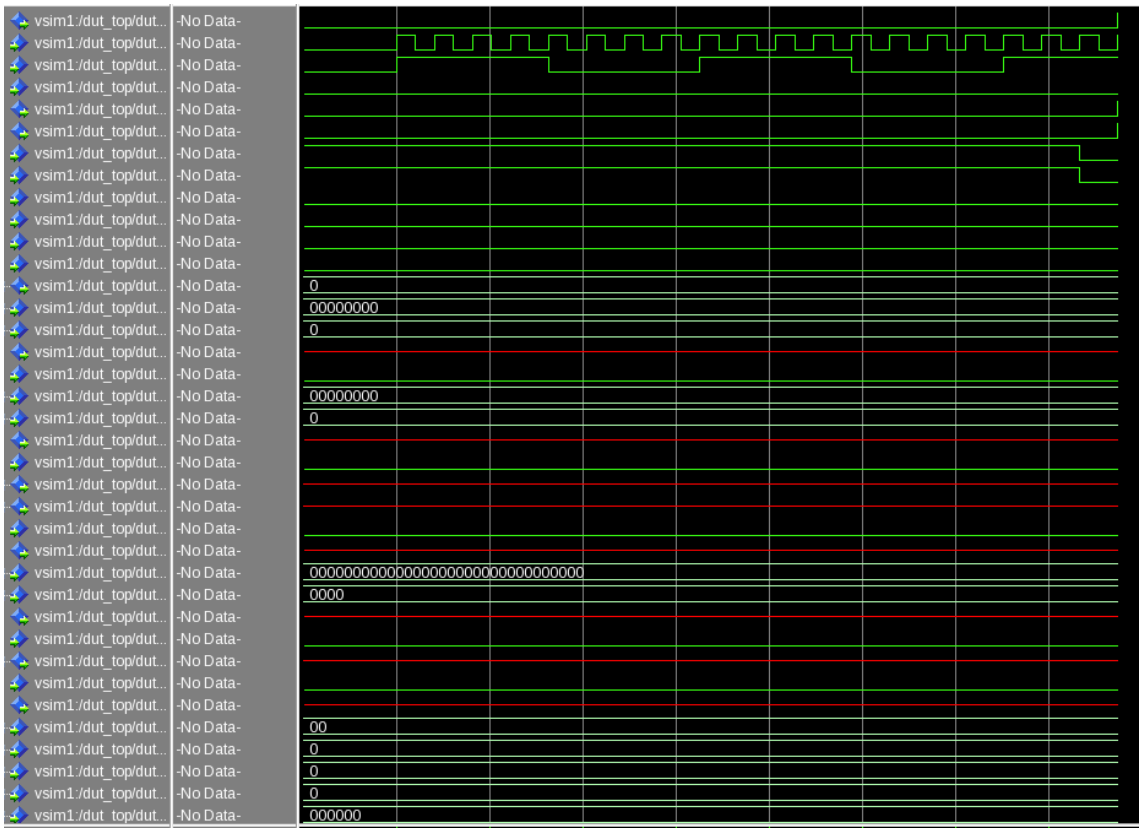

Figure 4.2. *Sanity check waveform for resolve mode simulation*

Figure 4.2 shows that the simulation has run for a significantly shorter time. The shorter simulation time can be concluded from the clock signal cycles. This is caused by the error thrown by the simulator, and the source of the fatal 'X' can be followed using the X tracing function of the simulator.

Running the sanity check with the X-propagation plugin enabled in pass mode. Like the previous simulation, the simulator throws an error, and the simulation stops. When looking at the simulation log, it can be seen that the error was the same as in the resolve mode simulation. The waveform of the sanity check in pass mode is represented in figure 4.3.

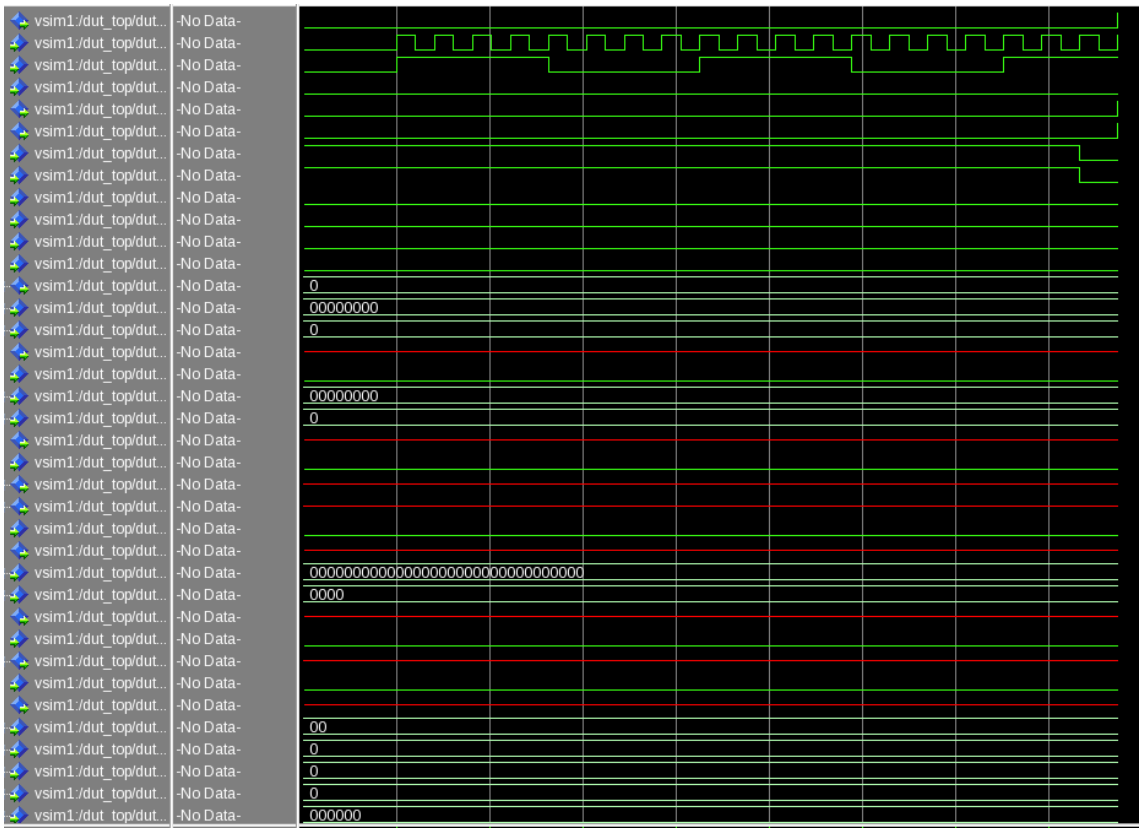

Figure 4.3. *Sanity check waveform for pass mode simulation*

Figure 4.3 shows no difference in the simulation, and the log file shows the same fatal error. This indicates that the source of the X-value might be causing the simulations to fail.

Using the simulator's schematic view and the X tracing function, the source of the X-value causing the error can be found easily. An example of the X tracing function in use is represented in figure 4.4.

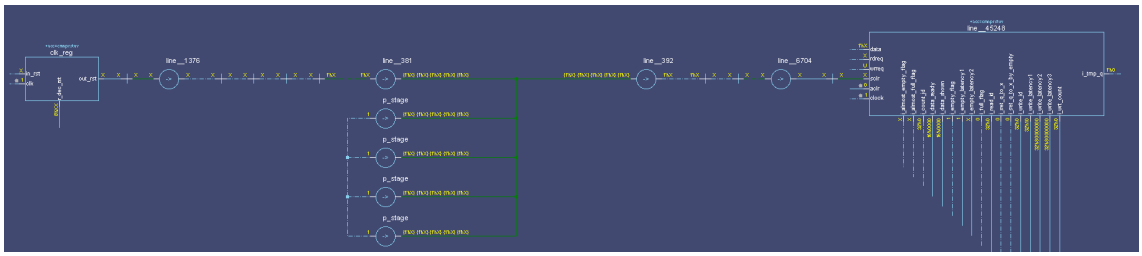

Figure 4.4. *Sanity check schematic for X tracing*

Figure 4.4 shows the whole chain of components the X-value goes through as traced by the X tracing function. This will make finding and fixing the ambiguity causing the chain of X-values easier and much faster than going through massive amounts of code.

Looking more closely at the same figure so that the ports and significant signals can be seen. The closeup picture of the schematic in X tracing use is represented in figure 4.5.

Figure 4.5 shows the relatively simple origin of the problem causing the X-value in this

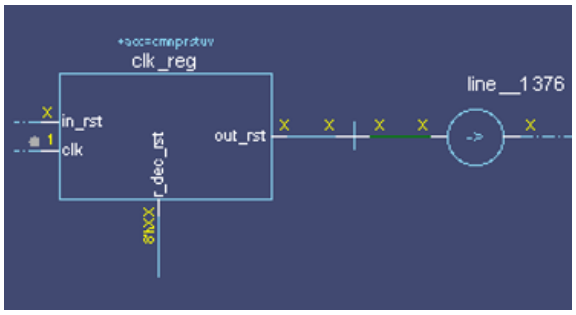

Figure 4.5. *Sanity check schematic for X tracing*

simulation and the ports of the two components. In this case, the likely source of the X-value caused error is the X-value from the uninitialized reset input. The reset input can be assumed to be uninitialized because the X-value emerged at startup. This is compounded by the large size of the design and the unavailability of room for all reset signals to be initialized at the beginning of the simulation.

Since the simulation doesn't pass the sanity check in the X-propagation plugin simulation it is easy to start the debugging from those errors. The first error was an integer value out of range: "Fatal: (vsim-3421) Value -2147483648 is out of range 0 to 79". In this case the the if statement was as follows.

```
if (ab_bfcm_buffer_we_i = '1' and ram_slot_nr /= NO_RTVS_INDEX) then
bfcm_buffer_index_updated_t(ram_slot_nr) <= not
bfcm_buffer_index_updated_t(ram_slot_nr);
```

Listing 4.1. *If statement.*

This error was fixed by assigning the following range and initial value:

```
signal ram_slot_nr       : integer range 0 to 80;
constant NO_RTVS_INDEX   : integer := 80;
```

Listing 4.2. *First fixes.*

The second error was also a value out of range error in an elif statement "Fatal: (vsim-3421) Value -1 is out of range 0 to 22". In this case the the elsif statement was as follows.

```
elsif (sample_idx_r /= 0 and wr_pend_r(sample_idx_r-1) = '0') then
    read_idx_r0                 <= sample_idx_r - 1;
    sample_idx_r                <= sample_idx_r - 1;
    re_r0                       <= '1';
    wr_pend_r(sample_idx_r-1)   <= '1';
end if;
```

Listing 4.3. *Elsif statement.*

The error was fixed by modifying the elsif statement to a new form.

```
elsif (sample_idx_r > 0) then
    if (wr_pend_r(sample_idx_r-1) = '0') then
      read_idx_r0                <= sample_idx_r - 1;
      sample_idx_r               <= sample_idx_r - 1;
      re_r0                      <= '1';
      wr_pend_r(sample_idx_r-1)  <= '1';
    end if;
end if;
```

Listing 4.4. *New elsif statement.*

As seen from the previous errors, many of the X-related issues with simulator A arise when an X-value is fed to a value range. The simulator cannot assign a valid value in the range to the 'X' and thus throws an error.

Using simulator A to simulate a part of the design without any X-propagation plugins enabled. The testcase in question is for a testfeature. Although the sanity check passed, it doesn't mean that any specific testcases would pass. Before running any X-propagation plugin simulations, the testcase in question should pass. The waveform of the testcase is represented in figure 4.6.

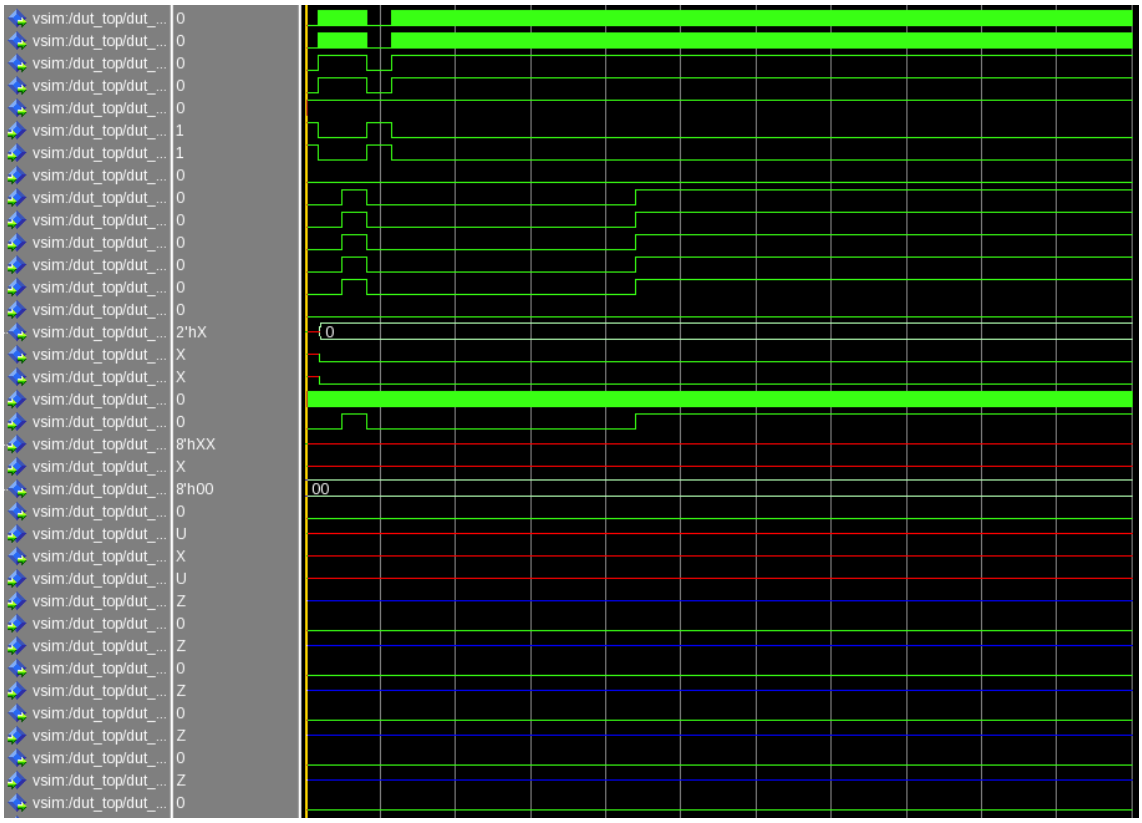

Figure 4.6. *Normal testfeature simulation waveform*

Figure 4.6 partially shows the signals in the passing simulation. Due to the scale of the design, it is not practical, but the comparison between simulations is the focus of this section.

Running the testcase with the X-propagation plugin enabled in resolve mode like previously caused the simulation to show warnings of X-values and eventually throw a fatal error. The waveform of the resolve mode simulation of the testfeature is represented in figure 4.7.

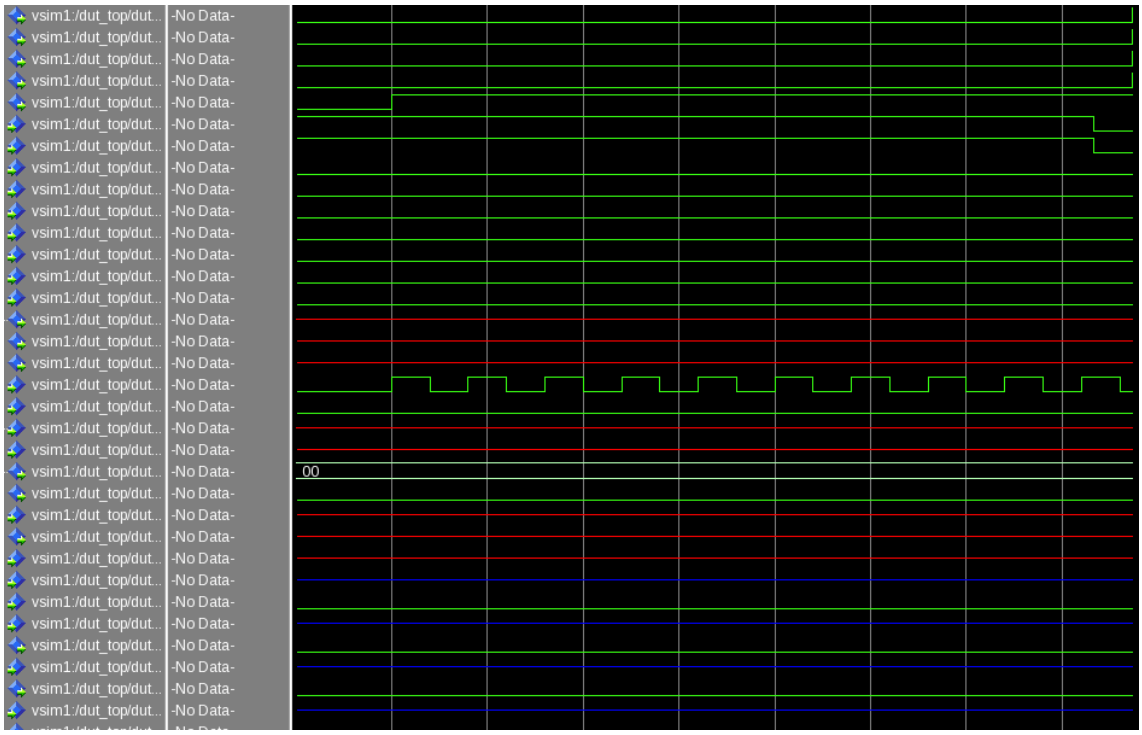

Figure 4.7. *Testfeature resolve mode simulation waveform*

Figure 4.7 shows the start of the simulation in the same way as the previous figure. When the two clocks on the top of the signal list start to toggle, the simulation stops due to a fatal error. Once the fatal error was thrown, the simulation stopped, and from the simulation log, it could be seen that the error was caused by the same source as in the sanity check case. This indicates an X-value source that critically affects the function of the design in startup or initiation.

Running the testcase with the X-propagation plugin enabled in pass mode like previously caused the simulation to show warnings of X-values and eventually throw a fatal error. The waveform of the resolve mode simulation of the testfeature is represented in figure 4.8.

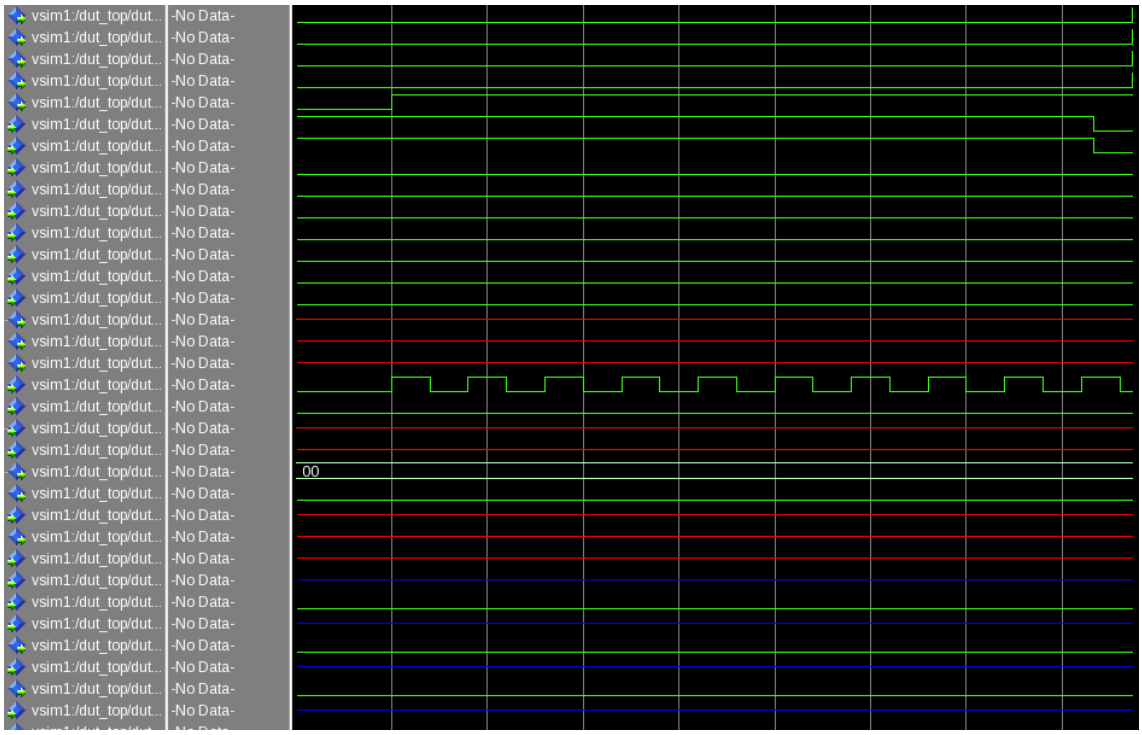

Figure 4.8. *Testfeature resolve mode simulation waveform*

Figure 4.8 shows no difference from the previous simulation, and when looking at the simulation log, the same fatal error occurs as before. This would suggest a similar, if not the same source, for the X-value.

When checking the schematic view and X tracing, it was evident that the same X-value source was affecting both the testfeature and the sanity check in resolve and pass mode, which would suggest a component linked to a high number of components and sub-components. The reset input shown in 4.5 does fit this description.

5 ANALYSIS OF RESULTS

Simulation results by themselves are meaningless without analyzing them. This case analysis will focus on the simulator A simulation results in the smaller designs and the results of the large design simulations.

From the initial tests, it was found that the data types of critical signal paths should be given attention. If X-propagation tests are done with 2-state data types such as ’bit’, all X values in those signals will be automatically turned into ’0’ values.

5.1 Small design simulations

Most simulation results were the same as expected and matched the results in the user guide. This means that simulator A catches X-values predominately as intended. The one difference was when running the case statement simulation. In some simulations there were situations where the simulator could have operated differently.

Starting with the if-else simulations, the simulator operated as expected, and there were no differences in operation compared to the user guide. With no difference to the user guide, the simulators ability to catch X-values in if-else statements is as expected.

In the case statement simulation, a mismatch was found between the resolve mode simulation and table 2.4. The simulator A case statement simulation results were exactly the same as when run in pass mode. In such a simple setup, this kind of error can cause issues with excessive X-values down the line if a more X-optimistic simulation result is expected. This issue was brought up with the simulator vendor and will be fixed in future versions of the simulator. When the simulator operates only in X-pessimistic mode, it will make finding the correct X-value more difficult.

The random number generator simulation did get significantly effected in the X-value and the X-propagation plugin simulation. When any X-values are introduced to the design, the values will eventually cause the RNG circuit registers to be filled and stuck as X. This is caused by the feedback loop seen in listing 3.3. The X-propagation plugin simulation results do differ from the standard simulation. While in the standard simulation the registers are filled gradually with X-values, the X-propagation plugin simulation starts off with the registers in X-states. The simulation results would suggest that both X-propagation plugin modes are operating very pessimistically since both default the registers to ’XX’. This is avoided in the case where the registers have a default value. If integrated to a

larger design this would cause problems during simulation unless some simulation specific initialization is made. In this circuit getting the r_reg to ’0000’ would be enough to remove the ambiguity from the simulation after one cycle.

The state machine simulations with both valid and X-values and without the X-propagation plugin ran as expected. The simulation results in figures 3.5, 3.6 and 3.7 are effectively the same.

When the ’we’ signal is set as ’X’, the simulation could change the from the initial State_1 to an ’X’ when running the X-propagation plugin. The simulations with X-value data input did not differ between each other when the ’we’ signal remained the same. The differences came when the ’we’ signal toggled between ’0’, ’X’ and ’1’.

When looking at the output data when the ’we’ signal is toggling in figure 3.8 the trap mode simulation acts in the same way as in standard simulation. When the simulation is in trap mode the X-values are interpreted as equal to ’0’. This can be seen from the state switching only when ’we’ is ’1’ and the ’dout’ is set as ’0’ when ’we’ is ’0’.

When either resolve or pass mode simulation is run, the data output ’dout’ switches between a valid value and an ’X’. While State_1 acts as expected at first and ’dout’ changes to ’0’ after ’we’ is st as ’0’, at the next high clock, ’dout’ is set to ’X’ which is not an output option written in the code. This would suggest that, in this situation simulator A doesn’t interpret X-values as either ’1’ or ’else’. It is easy to assume that X-values would fall under the definition of being something else than ’1’. This result makes it clear that the propagation of X-values cannot be stopped by relying on them to be caught in else statements. It was interesting to see that the resolve and pass modes handled the X-values identically.

The valid data output sequence matches the operation of the first and second states in the X-propagation plugin simulations. In the non-X-propagation plugin simulation the output matches the previous simulations.

One noticable feature of simulator A X-propagation plugin is that state machine states do not get an X-value. This means that state machine simulations are inherently X-optimistic, which restricts the X-propagation plugin simulation results for state machines. There is no difference between the resolve and pass simulations so that would mean that even the most pessimistic simulations would not turn the state to an unknown.

In the RAM simulations there were very little difference between the X-propagation plugin modes and even between normal and X-propagation plugin simulations. This seems to be a result of the very simple structure of the circuit. The differences came when the ’we’ signal is set as ’X’. In normal simulation, when the ’we’ signal is ’X’ the circuit acts as if it is equal to ’0’. In X-propagation plugin operation the X-value propagates at different rates through the RAM circuit. In pass mode it takes two clock cycles for ’dout’ to switch to ’X’ and in pass mode it switches after one clock cycle. This would suggest that the different simulator A X-propagation modes also affect the rate of propagation at least in memories.

The shift register simulation operated mostly as expected although the simulations were more pessimistic than expected. In the final parts of the simulation after the control signal was set as ’0X’, the input as "1100_0000", the ’DOUT’ and ’S_REG’ as "0011_0000". Simulator A is stating that all bits around the initial "11", as well as the most significant bit (MSB) and the initial "11", cannot be determined. Since the control signal is an unknown right shift, the resulting ’dout’ could have reasonably been "X0X1_X000" instead of "XXXX_X000".

5.2 Large design simulations

The focus of the analysis must focus on the attributes of simulator A and how different situations affect the simulations. Most of the simulations ended up stopping at the same fatal error in the initial simulation runs.

The usability of the simulator, must also be noted. When running the larger design simulations, it was easy to find the X-value causing the fatal error and trace its source using the X tracing function. Simulator A has an easy-to-use user interface, and running the simulations was quite simple.

It can be seen that the sanity check passed without error in figure 4.1. Since this was a working design and testbench, passing the sanity check was not surprising. Running the sanity check using the X-propagation plugin caused the simulator to throw a fatal error which stopped the simulation. When looking at the resolve mode simulation waveform in figure 4.2, it seems that the error occurs very early in the simulation. The fifth and sixth waves are clock signals, and the simulation stops when those signals are going high. From this, it can be deduced that the error occurs very early in the simulation, if not at startup. The simulation result was the same in both resolve and pass modes.

The simulator throws a fatal error when it cannot calculate the next value. This is working as intended, and the cause of the X-related error should be examined. This could be negated by lengthening the initiation sequence, which would flush out excess X-values.

The X tracing function produced a chain of components and X values going through them. The chain ends on the component where the fatal error occurred. The error could be fixed in the component it occurred in or in any of the components the X-value goes through. If the X-value is sopped at the first component it goes through, the possibility of it cascading into multiple components is negated. In this case the first component where the ’X’ is found is "clk_reg" as seen in figure 4.5.

The corrections needed in DUT code to get the sanity check to pass show that correcting a few critical errors may be enough to fix X-related bugs in the simulation. Since this was only a sanity check, any specific testcase related errors will be missed. The same simulation and bug fix methodology can be used in in other simulations. First the testcase should pass in a normal simulation, second the testcase should be run using the X-propagation plugin and finally the X-related issues should be addressed in tandem with

the DUT design team.

The X-propagation plugin simulations should be run as early as possible to avoid long chains of problematic X-propagation. The debugging is easier when the design is smaller and will take less time. If the design is large, the engineer running X-propagation plugin simulation will need to contact multiple source file owners. In older designs the ones responsible for the initial work may not be possible to reach.

When running a testfeature test on the design without simulator A X-propagation plugin enabled, the test passed. The resulting waveform of the simulation can be seen in figure 4.6. Running the same simulation in resolve and pass mode resulted in the same error as in the sanity check case. The resulting waveform can be seen in figure 4.7. Getting the sanity check to pass in resolve mode simulation did not make the testfeature test but resulted in a continuous loop. This loop could not be fixed.

6 CONCLUSION

Assessing the usability of the tools provided to the engineers is an important part of the process of choosing the simulator used to verify designs. The testing done in this thesis will help in making a more informed decision in choosing a simulator for X-propagation simulation.

X-propagation testing can help find possible problematic parts of the design before it progresses into a physical implementation, as well as help the debugging process of a physical design. This is made more important as designs grow, and having initial values at startup and reset is made even harder. Finding errors early in the design is one of the main goals of verification and RTL simulation.

X-propagation testing is a powerful tool for finding issues concerning ambiguity in RTL simulation. The work on this thesis was done using simulator A with the X-propagation tool. While simulator A X-propagation plugin required a license to run, not all X-propagation tools do. The licence setup in this case was somewhat inconvenient due to the test environment setup.

The findings in this thesis would suggest that simulator A works well to an extent with some fixes needed and can be used to find X-related issues in designs. The issues with the previous versions of the X-propagation plugin have been addressed and the issue with case statements was corrected in a later version of the plugin.

Especially the X tracing function makes finding the sources of X-values much faster and helps in visualizing the issue easier. In many cases the X-values causing issues are caught when the simulator can no longer calculate the next value, causing a fatal error. In these cases simulator A works well to catch offending X-values and when fatal errors don't occur, the X-values are easily traced using the X tracing function and simulation logs, where values going to X are indicated.

When using simulator A it should be noted that state machine operation is inherently X-optimistic while operating pessimistically in case statements. This will make catching X-related issues more difficult in situations where such designs are both used.

To confirm that simulator A performs at a desirable level and to improve verification coverage when simulating larger designs, it should be run in parallel to other simulators. Public release of such comparisons is not permitted and should be done internally.

In the basic design simulations simulator A performed somewhat well. Simulator A X-

propagation plugin had a bug regarding case statements as seen in section 3.1.1, that hindered its use which has been brought to the attention of the simulator vendor and will be fixed in later releases. The newest version of the simulator should be used to get the proper function for the cases affected by the bug.

Following studies would be the validation of the bug fixes for simulator A and running the design simulations using different simulators. The environment is set up to run simulator A simulations on a newer version of the simulator, but the larger design environment would have to be worked on to run different simulators.

Replicating the work done for this thesis can be done for the basic design part of the tests. Since UVM is a standard for verification, the tests can be run on the basis of the code provided and the sequences described in this work. The stimulus and design used in the large design simulations cannot be publicly available.

REFERENCES

[1] W. Chen et al., Challenges and Trends in Modern SoC Design Verification, IEEE Design Test, Vol. 34, Iss. 5, 2017, pp. 7–22.

[2] Y. Yun et al., Beyond UVM for practical SoC verification, 2011 International SoC Design Conference, 2011, pp. 158–162.

[3] A. Deshpande, Verification of IP-Core Based SoC's, 9th International Symposium on Quality Electronic Design (isqed 2008), 2008, pp. 433–436.

[4] P. P. Chu, RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability, Wiley-IEEE Press, 2006.

[5] L. Scheffer, L. Lavagno, G. Martin, EDA for IC System Design, Verification, and Testing (Electronic Design Automation for Integrated Circuits Handbook), CRC Press, Inc., USA, 2006.

[6] *UVM TestBench UVM TestBench*, https://verificationguide.com/uvm/uvm-testbench/, Accessed: 2022-04-28.

[7] A. Piziali, Functional verification coverage measurement and analysis, Springer Science & Business Media, 2007, pp. IIX–IX.

[8] *Onespin X-Propagation analysis*, https://www.onespin.com/products/dv-verify-apps/x-propagation-analysis, Accessed: 2021-01-27.

[9] *Tech design forum X Propagation*, https://www.techdesignforums.com/practice/guides/x-propagation/, Accessed: 2021-01-27.

[10] S. Sutherland, I'm Still in Love with My X!, Proceedings of the Design and Verification Conference (DVCon), 2013.

[11] L. Piper, Jin Zhang, Don't let the X-bugs bite: Conquer elusive X-propagation issues early! Get them before they get you!, 2011 9th IEEE International Conference on ASIC, 2011, pp. 345–348.

[12] M. Turpin, P. V. Engineer, The Dangers of Living with an X (bugs hidden in your Verilog), Synopsys Users Group Meeting, 2003.

[13] M. A. Kochte, M. Elm, H. Wunderlich, Accurate X-Propagation for Test Applications by SAT-Based Reasoning, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 31, Iss. 12, 2012, pp. 1908–1919.

[14] Questa SIM User's Manual, version Version 2020.2, Siemens, 2020.

[15] R. Drechsler, Advanced Formal Verification, Kluwer Academic Publishers, USA, 2004.

[16] K. Liu, R. Sabbagh, Confidence in the Face of the Unknown: X-state Verification, Verification Horizons, Vol. 9, Iss. 2, 2013.

[17] S.-M. Edgar, M.-V. David, State of the Art in the Research of Formal Verification, Ingenierıa, Investigación y Tecnologıa, Vol. 15, Iss. 4, 2014, pp. 615–623.

[18] VCS User Guide, version Version P-2019.06-SP1, Synopsys.

[19] C. Kwok, P. Viswanathan, P. Yeung, Addressing the challenges of reset verification in SoC designs, Design and Verification Conference and Exhibition United States, 2015.

[20] *Tech design forum How to expose X-optimism issues in ASIC and FPGA design*, https://www.techdesignforums.com/practice/technique/x-optimism-in-asic-and-fpga-design/, Accessed: 2021-01-27.

[21] *FPGA designs with VHDL Design examples*, https://www.vhdlguide.readthedocs.io/en/latest/vhdl/dex.html, Accessed: 2022-04-17.