NEXT GENERATION RTL SIMULATION TECHNOLOGY FOR GRAPHICS DESIGN

Major Project

Submitted in Partial Fulfillment of the Requirements

For the Degree of

Master of Technology (M.Tech.)

In

VLSI Design

By

ABHISHEK D SAVALIA

12MECV01

Department of Electronics & Communication Engineering

Institute of Technology

Nirma University

Ahmedabad

May-2014

NEXT GENERATION RTL SIMULATION TECHNOLOGY FOR GRAPHICS DESIGN

Major Project

Submitted in Partial Fulfillment of the Requirements

For the Degree of

Master of Technology (M.Tech.)

In

VLSI Design

By

ABHISHEK D SAVALIA

12MECV01

Mr. Anees Sutarwala

Prof. Usha Mehta **Mr. Nilay Desai**

Internal Guide External Guide

Department of Electronics & Communication Engineering
Institute of Technology
Nirma University
Ahmedabad
May-2014

Declaration

This is to certify that

1. I, Abhishek D Savalia a student of semester IV Master of Technology in VLSI Design, Nirma University, Ahmedabad hereby declare that the project work "NEXT GENERATION RTL SIMULATION TECHNOLOGY FOR GRAPHICS DESIGN" has been independently carried out by me under the guidance of Mr. Anees Sutarwala and Mr. Nilay Desai, Intel Technology India Private Limited, Bangalore and Prof. Usha Mehta, Program coordinator, Department of VLSI Design, Nirma University, Ahmedabad. This Project has been submitted in the partial fulfillment of the requirements for the award of degree Master of Technology (M.Tech) in VLSI Design, Nirma University Ahmedabad during the year 2013 - 2014.

2. I have not submitted this work in full or part to any other University or Institution for the award of any other degree.

Abhishek D Savalia
12MECV01

Certificate

This is to certify that the Major Project entitled “NEXT GENERATION RTL SIMULATION TECHNOLOGY FOR GRAPHICS DESIGN" submitted by Abhishek D Savalia (12MECV01) towards the partial fulfillment of the requirements for the degree of Master of Technology in VLSI Design of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Mr. Anees Sutarwala
(External Project Guide
Intel Technology India Pvt. Ltd.,
Bangalore)

Mr. Nilay Desai
(External Project Guide
Intel Technology India Pvt. Ltd.,
Bangalore)

Prof. Usha Mehta
(Internal Project Guide
Institute of Technology,
Nirma University, Ahmedabad)

Dr. N. M. Devashrayee
(Program Coordinator
Institute of Technology,
Nirma University, Ahmedabad)

Dr. P. N. Tekwani
(Professor & Head EE Dept.
Institute of Technology,
Nirma University, Ahmedabad)

Dr. K Kotecha
(Director,
Institute of technology,
Nirma University, Ahmedabad)

Date:

Place:

Acknowledgement

First and foremost, sincere thanks to Mr. Nilay Desai, Intel Technology India Private Limited, Bangalore for valuable guidance. Through- out the training, he had given me much valuable advice on project work which I am very lucky to benefit from.

I would like to thank to Mr. Anees Sutarwala, Mr. Narasimhan Iyengarand my manager Mr. Jayant Deodhar, Intel Technology India Private Limited, Bangalore for assigning me such project and guide me through.

I would also like to thank my teammates, form Intel India Technology for their valuable time in ramping me up on some basic flow of different projects.

I would also like to thank Dr. K.R.Kotecha, Director, Institute of Technology, Nirma University, Ahmedabad for providing me an opportunity to get an internship at Intel Technology India Private Limited, Bangalore.

I would also thank to my Project coordinator, Professor Usha Mehta and Dr. N.M. Devashrayee, VLSI Design, Institute of Technology, Nirma University, Ahmedabad for giving valuable support for project work and also teaching me some very intersecting subject in post-graduate programs.

I also owe my colleagues in the Intel, special thanks for helping me on this path and for making project at Intel more enjoyable and more memorable.

Abhishek D Savalia
12MECV01

Abstract

Silicon like behavior at the RTL simulation level is a necessary requirement now days as design complexity increase. Gate Level Simulation (GLS) is useful to achieve this but as design complexity increase it becomes more expensive and time consuming. There are some limitations with System Verilog semantics, one of the limitation is X value simulation semantics. In the first part this report describes this limitation and the enhancement to overcome the problem by next generation simulation technology.

System Verilog has several drawback with X value simulation semantics that may be results in simulated X being improperly propagated which in turn may lead to initialization and power related failures in Silicon. Even RTL bugs can be masked and thus RTL simulation may pass incorrectly and would fail at silicon level [3].

System Verilog uses different X semantics for different parts of the design flow. For synthesis X represents Don't Care Boolean Values (0 or 1), while for simulation it represents an unknown value (0, 1 or Z). Verilog RTL simulation semantics often mask propagation of an unknown value by converting the unknown to a known, while gate-level simulations show additional X that will not exist in real hardware. The result is that bugs get masked in RTL simulation, and while they show up at the gate level, time consuming iterations between simulation and synthesis are required to debug and resolve them [1]. Resolving differences between gate and RTL simulation results is painful because synthesized logic is less familiar to the user, and Xs make correlation between the two harder. Enhancing the RTL simulator for finding X issues will be one of the best ways. This project is about bringing up the new simulation mode on the next-generation servers and graphics design.

Second important aspect of this project is improvement in simulation performance by reducing simulation run time & run time memory requirement. There are very big and complex designs and need so much run time while debugging, there is a new debug methodology which gives better run time performance with the same debug capability over existing methodology, second part of the project describes the new debug methodology as a next generation simulation technology.

Contents

Declaration i

Certificate ..... **Error! Bookmark not defined.**

Acknowledgement iii

Abstract iii

List of Figures vi

List of Tables vii

Part-1 X-propagation

Chapter 1. Introduction 2

1.1 What is X? 3

1.2 Source of X in RTL 3

1.3 Ambiguous RTL Construct 4

1.3.1 IF/ELSE Statements 4

1.3.2 Case Statements 4

1.3.3 Bit Selects and Indexing 5

1.3.4 Ambiguous Edge 5

Chapter 2. Approaches to Manage the X optimism in RTL Simulation 6

2.1 Gate Level Simulation 6

2.2 RTL Coding Guidelines 7

2.3 X Randomization in RTL or Gates 8

2.3.1 Static X Randomization 8

2.3.2 Dynamic X Randomization 8

Chapter 3. X-Propagation 9

3.1 What is X-propagation? 9

3.2 How X-prop works? 9

Chapter 4. Ambiguous RTL construct with X-propagation 11

4.1 X-Propagation on if Statement 12

4.2 X-Propagation on case Statement 14

4.3 X-Propagation on Edge Sensitive Process 16

4.4 X-Propagation on Latches 18

Chapter 5. Bugs/Issues found 19
Issue 1. X-propagation on if statements 19
Issue 2. X-propagation on case statement. 21
Issue 3. X-propagation on Edge sensitive process 23
Chapter 6. Limitation 25
Chapter 7. Conclusion 26
Chapter 8. References 27
Part-2 A new debug methodology
Chapter 1. Introduction 29
Chapter 2. Current flow 30
Chapter 3. A new debug (force) methodology 31
Chapter 4. Experiments 32
Chapter 5. Problem faced for the debug methodology 34
Chapter 6. Bugs found 35
6.1 Signals under structure in design 35
6.2 Issue when any signal doesn't exist. 36
6.3 Issue when any module doesn't exist. 36
Chapter 7. Results 37
Chapter 8. Conclusion 38
Chapter 9. Other Contribution 39

List of Figures

2.1 A Gate level circuit - 1 6
3.1 Gate level circuit-2 9
5.1 problem code – if statements 19
5.2 Simulation results without Xprop 20
5.3 Simulation results with Xprop 20
5.4 Problem code – case statement 21
5.5 Simulation results without Xprop 22
5.6 Simulation results with Xprop 22
5.7 Problem code – Edge sensitive process 23
5.8 Simulation results without Xprop 24
5.9 Simulation results with Xprop 24
3.1 Flow diagram for debug methodologies 31
6.1 ripple counter 34
6.2 ripple counter test bench 34

List of Tables

1.1 sources of X in RTL #ref. Ref. Xprop_user_guide_Oct_2012, by Synopsys, Inc. : 3
2.1 RTL Coding Guidelines 7
3.1 Truth table for gate-level circuit -2 10
4.1 If/else Truth table without xprop 12
4.2 If/else Truth table with xprop 12
4.3 case Truth table without xprop 14
4.4 case Truth table with xprop 14
4.5 Edge sensitive process Truth table without xprop 16
4.6 Edge sensitive process Truth table with xprop 17
4.7 Latch Truth Table with Xprop 18
5.1 Truth table regular mode – if statement 19
5.2 Truth table Xprop mode – if statement 19
5.3 Truth table regular mode – case statement 21
5.4 Truth table xprop mode – case statement 21
5.5 Truth table normal mode – edge sensitive process 23
5.6 Truth table xprop mode – edge sensitive process 23
6.1 Time analysis with & without Xprop 25
7.1 Run time comparison normal/new debug methodology 36

Part -1
X-propagation

Chapter 1. Introduction

Intel has extremely high complex codes for various graphics design, millions of gates/flip-flops. There are many possibilities of X generation in design if coding is not accurate. Even many designers are not aware of codes that may become sources of X. This project is focused on sources of X and gets silicon like results within RTL simulation only by enhancing simulation technology.

Simulation semantics of X values in Verilog RTL are problematic because they may hide functional bugs that allow RTL simulations to incorrectly succeed, and thereby gate-level simulation and/or silicon may fail.

Understanding problems caused by X semantics is extremely important. Many designers are unaware of the issues around X, which can have devastating effects on many different parts of the design flow including [3]:

1. **RTL Simulation**: X semantics in RTL can mask bugs - expensive validation tests can pass because they are not being used effectively to stress the design.

2. **Synthesis**: designers often rely on don't -cares to produce efficient logic, but can be disappointed with their non-minimal results and long critical paths

Often due to limited understanding of X issues, bugs can be missed and left in the shipped product (discovery then is far more expensive) or left inactive and those only to reappear when a new version of synthesis tool chooses a different logic minimization.

X issues can be caught through Gate Level Simulation (GLS) mechanism. But GLS is extremely costly in terms of setup, debug and simulation performance. Enhancing the RTL simulator for finding X issues will be one of the best ways. This project is about bringing up the new simulation mode on the next-generation servers and graphics design.

1.1 What is X?

X is an abstract value introduced for the sake of algebraic semantics, and different tools interpret them differently [1].

Simulators interpret an X as a value in 4-state logic (0, 1, X, Z) that represents an "unknown" logic value. There are four data types defined in the standard that use 4-state logic: logic, reg, integer, and time. All of these data types have a default value of X. Synthesis tools treat Xs differently. They interpret the X value as a "don't care" instead of an "unknown", allowing for greater synthesis optimizations. [2]

1.2 Source of X in RTL

Source	Description
Uninitialized state	All flip-flops and memories in a design will start with a 'X' value, until they are initialized through a reset or a write to a non-X value.
RTL Assignment to 'X'	Designers may assign the outputs of their circuit to 'X', as a means of expressing an output don't care condition. Logic synthesis tools uses the freedoms of output don't care conditions to minimize the logic.
Test bench	The bus protocol may specify that a given signal should not be consumed under some conditions (e.g. valid=0). The test bench can drive 'X' into the DUT, to ensure that it is indeed not sensitive to the signal value.

Table 1.1 sources of X in RTL #ref. Ref. Xprop_user_guide_Oct_2012, by Synopsys, Inc.: https://solvnet.synopsys.com/retrieve/040022.html

1.3 Ambiguous RTL Construct

Since an 'X' value represents either a "0" or a "1", it is difficult for a single threaded simulator to simultaneously explore both cases. An optimistic approach is to only explore one of the possibilities. An approach that is potentially pessimistic is to propagate the 'X' value forward to all dependent variables [1].

1.3.1 IF/ELSE Statements

The classic example of optimistic RTL simulation behavior is an if/else statement. The code inside the 'IF' clause is only executed when the condition evaluates to a non-zero, known value. If the condition expression evaluates to true, the first statement shall be executed. If it evaluates to false, the first statement shall not execute. If there is an else statement and the condition expression is false, the else statement shall be executed.

Example:

```
always@ (A or B or sel)
begin
if (sel)
Y= A;
else
Y= B;
end
```

When sel is a '1', the output is the value of A and when sel is '0', the output is the value of B. But notice what happens when sel is X. Here, the X value is interpreted as other than '1' and the output is the value of B. The "unknown X" is now misinterpreted as a known value. In real hardware the sel signal might in fact have been a '1', which means that the correct value could have been the value of A.

1.3.2 Case Statements

In regular Verilog case statement (e.g. not casex, not casez) match only occurs if the case expression exactly matches the case item expression.

In a case expression comparison, the comparison only succeeds when each bit matches exactly with respect to the values 0, 1, x, and z. As a consequence, care is needed in specifying the expressions in the case statement.

The problematic behavior of the previous example could be coded as a case statement and produce the same problem.

Example:

```
always@ (A or B or sel)
begin
case (sel)
1'b1: Y = A;
1'b0: Y = B;
endcase
end
```

1.3.3 Bit Selects and Indexing

In Verilog, bit-selects are used to select a bit from a vector and indexing is used to select an entry from an array [2].

If an index expression is out of the address bounds or if any bit in the address is X or Z, then the index shall be invalid. The result of reading from an array with an invalid index shall return the default uninitialized value for the array element type. Writing to an array with an invalid index shall perform no operation. Implementations may issue a warning if an invalid index occurs for a read or write operation of an array [2].

We see that when bit-selects or indexing is used on the LHS of an assignment, the normal Verilog semantics cause the vector or memory to be unmodified, whereas in a real circuit, one of the entries would have been updated.

1.3.4 Ambiguous Edge

When a signal transitions to or from a 'X' value, it is ambiguous as to whether there is really a transition.[2]

- A negedge shall be detected on the transition
  From 1 → X, z, or 0, and
  From x or z → 0

- A posedge shall be detected on the transition
  From 0 → x, z, or 1, and
  From x or z → 1

As a result of these semantics, a clock that is alternating between '0' and 'X' would activate any sequential always blocks as if the clock were transitioning normally. If in the real circuit, where there are only two states, the 'X' resolved to a zero, the logic would not be clocked and the behavior would be very different [2].

Chapter 2. Approaches to Manage the X optimism in RTL Simulation

There are several approaches present to manage the X optimism.

- Gate Level Simulation (GLS)
- Different approach to RTL coding
- Static & Dynamic X randomization in RTL or Gates.

2.1 Gate Level Simulation

The most robust approach to ensuring that the final circuit implementation will behave the same as RTL simulations is to perform extensive gate-level simulations. One of the reasons that gate-level simulations are tedious to debug, is that the 'X' simulation behavior of a gate-level netlist can be pessimistic due to re-convergent paths. Consider the following gate level circuit [2].

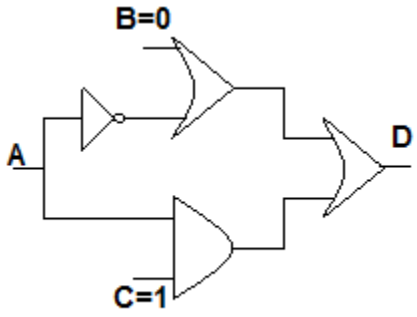

Fig. 2.1 A Gate level circuit - 1

We can see that D=1, regardless whether A=0 or if A=1. However, in gate simulation, if A is 'X', then D would also be 'X'. This is a classic problem in gate-level simulation, and each instance of this type of re-convergent logic needs be identified. Generally, the resolution is to manually initialize one of the signals to zero or one, using a force statement or through a VPI. In a large team, managing the list of signals that need to be manually initialized is error-prone and multiple engineers may spend time debugging the same issue [1].

Though GLS helps in X related bugs, it becomes more expensive and time consuming as design complexity increases.

2.2 RTL Coding Guidelines

Problematic RTL	Structure Coding Guideline
IF/ELSE	Combinatorial logic using IF/ELSE constructs can be replaced with assign statements using the ternary operator (? :).
CASE statement	If the default of every CASE statement drives all the outputs to 'X' then, if there is a 'X' in either the case item or case-expression, there is no match and the 'X' propagates. There is some residual risk, because if the case item and case-expression are both 'X', then they will match, according the Verilog semantics.
Ambiguous Edges	There is no direct coding guideline that will ensure that a posedge will not be falsely triggered by a transition to X or Z.
X Assignment	It is difficult to produce RTL code that robustly models Xs. To minimize the risk, assignments of outputs to X can be avoided. However, assignment to X is a means of expressing an output don't care which allows logic minimization in synthesis.

Table 2.1 RTL Coding Guidelines [2]

2.3 X Randomization in RTL or Gates

2.3.1 Static X Randomization

Another way to deal with Xs in RTL simulation is to use a tool or utility to remove them from the design and replace them randomly with 1s or 0s. This, effectively, models the behavior of the real design where all the flip-flops and memories have a random initial value. There are two problems with this approach [3].

- The first is that Xs can be re-introduced during the simulation from the sources identified in chapter 1.
- The second problem is that a few simulation runs using random 0, 1 value only provides superficial coverage of the full state space of the design. A design containing N state-elements (flip-flops or memory bits) has 2N possible initial states. Some constructs only cause problems for a specific value of initial conditions.

2.3.2 Dynamic X Randomization

The main problem with static 'X' randomization is that Xs can be re-introduced into the design, after startup for the reasons identified earlier. This can be addressed by continuously replacing all Xs with a random value on every clock cycle. Continuously traversing the design performing X replacement introduces a significant run-time penalty and it still only provides a statistical coverage of the full set of possible initial states based on the number of simulation runs.

Chapter 3. X-Propagation

3.1 What is X-propagation?

X-Propagation is an enhancement in simulator. This feature changes the way Xs are simulated with the intent of removing the optimistic 'X' behavior that is intrinsic in the standard Verilog semantics [2]. For example, with X-Propagation, when the condition of an IF statement evaluates to ‟X ‟, this ‟X ‟ can propagate to the variables that are assigned in both the IF and the ELSE branches. Similarly, if a case item or case expression evaluates to 'X', then the 'X' propagates to the variables that are assigned in the case statement. Ambiguous edges on clocks are handled by considering the behavior when there are only definite edges (e.g. 0->1) and the behavior when there are ambiguous edges (e.g. X->1) it merging the results. The key to the X-Propagation semantics is the ability to merge the multiple values which could be assigned to an output variable.

X-propagation helps to enter in next generation RTL simulation technology for graphics design. It can find X-related bugs in RTL simulation only. X-propagation is an advanced simulation technology. The RTL simulations semantics are capable to expose X related bugs. We can say in X-propagation X propagates; not absorbs.

3.2 How X-prop works?

There are 3 types of merge[5].

- T-merge (Traditional merge)
- X-merge (Pessimistic merge) and
- V-merge (Standard Verilog)

In traditional merge the output is driven X only when the condition is X and the inputs are different. On other hand in pessimistic merge the output is driven X when the condition is X irrespective of inputs. V-merge is same as standard Verilog semantics. This is explained in the following gate level circuit and truth table.

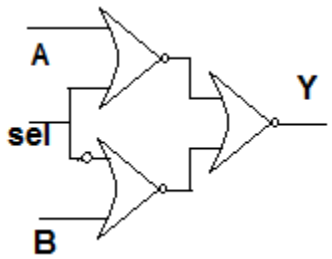

Fig.3.1 Gate level circuit-2

Here is the truth table for above circuit

sel	A	B	T-merge	X-merge
X	0	0	0	X
X	0	1	X	X
X	1	0	X	X
X	1	1	1	X

Table 3.1 Truth table for gate-level circuit -2

Here is sel line is X then X-prop check the output by putting 1 and 0 respectively if both the answers are same then it gives same output in T-merge. While in X-merge regardless the values of A and B it gives X as output whenever sel is X.

Chapter 4. Ambiguous RTL construct with X-propagation

X-propagation considering the effect of both Boolean values of X (either 1 or 0) for every statement whose execution controlled by X and by merging their results changes the standard simulation behavior.

There are 3 merging options with Xprop:

- T-merge
- X-merge
- V-merge

T-merge(optimistic): has same semantics as Verilog ternary operator (? :) when all variables in a merge have the same deterministic value, then the result of the merge is the given value, otherwise the result is X.

X-merge(pessimistic): it yields X when the selecting condition is X, it models the behavior of synthesis more accurately by ensuring that that it will not result in cases where simulation would result in a deterministic value, while the post synthesis model would result in an X.

V-merge(normal verilog): it yields standard Verilog result.

4.1 X-Propagation on if Statement

X-Propagation affects the behavior on control logic. Shown below is the code for a simple if statement. This code represents a simple multiplexor.

Example

```
if (sel)
Y= A;
else
Y= B;
```

In normal verilog semantics the truth-table for above code is:

sel	Y
1	A
0	B
x	B

Table 4.1 If/else Truth table without xprop

Here if select line is X then output goes to else part and hence is B.
While we use X-propagation the truth-table for the same code is below:

Sel	A	B	V-merge	T-merge	X-merge
X	0	0	0	0	X
X	0	1	1	X	X
X	1	0	0	X	X
X	1	1	1	1	X

Table 4.2 If/else Truth table with xprop

The first column shows the value of the condition to the IF statement which is unknown. The second and third column shows their respective values for A and B. The value of Y is shown in columns 4-6 for different merge options.

V-merge, shown in the fourth column always gives the same value as signal B, as this is standard Verilog behavior for an IF statement. In the fifth column T-merge is shown. When the condition is unknown, the values of 0, 1 are substituted for X.

The IF statement is executed once with sel=0, and once with sel=1, and the value of Y is computed. If the values forY in each of the branches are the same, then the merge value will be that value. If the values for Y in each of the branches are different, the merge value will be X. So, when A and B are the same, T-merge will give that value. If the value of A and B are different, then T-merge will give an X. In the last column the value of Y is shown when the merge option used is X-merge.The output is always an X, because whenever there is a merge, the resultant with be an X.

4.2 X-Propagation on case Statement

Shown below is the code for a simple case statement. This code represents a simple multiplexor. Below the code is the truth table when the value of the case expression is unknown.

Example

```
case (sel)
1'b1: Y = A;
1'b0: Y = B;
endcase
```

Here is the truth table for normal Verilog semantics:

sel	Y
1	A
0	B
X	prev value

Table 4.3 case Truth table without xprop

Here when select line is X then output is the same as previous value.
When we use X-prop then the truth table for the same code is:

Sel	A	B	V-merge	T-merge	X-merge
X	0	0	r(t-1)	0	X
X	0	1	r(t-1)	X	X
X	1	0	r(t-1)	X	X
X	1	1	r(t-1)	1	X

Table 4.4 case Truth table with xprop

Here the first column shows the value of the condition to the case statement which is here considered to be unknown. The second and third column shows their respective values for A and B.

The value of Yis shown in columns 4-6 for different merge options. V-merge, shown in the fourth column always gives the value of Ythat was present before the case statement is executed.This is standard Verilog behavior for a case statement. In the fifth column T-merge is shown. When the condition is unknown,the values of 0, 1 are substituted for Xs.The case statement is executed once with sel=0, and once with sel=1, and the value of Y is computed. If the values for Y in each of the branches are the same, then the merge value will be that value.

If the values for Yin each of the branches are different, the merge value will be X. So, when A andB are the same, T-merge will give that value. If the value of A and B are different, then T-merge will give an X. In the last column the value of Y is shown when the merge option used is X-merge.

The output is always an X, because whenever there is a merge, the resultant with be an X. A case condition that contains X may result in wildcard-like behavior with more than one matching case item, including the default, therefore Xprop treats each matching case item as a possible branch and applies the same merging function it used to handle the if statements.

4.3 X-Propagation on Edge Sensitive Process

Edge sensitivity expression must be handled carefully under X-prop semantics. In legacy Verilog, a posedge expression will occur for the following transitions:

0 -> 1
0 -> X
0 -> Z
X -> 1
Z -> 1

The issue here is that Verilog will optimistically consider all of these transitions as if a rising edge of the signal occurred, which is not necessarily true. For example, let's consider the 0->X transition. X can represent either a 0 or a 1, which means a rising transition may have happened, or may not have happened. Both cases need to consider.

The code represents a simple D-flip flop, where there reset is inactive.

Example

```
always@(posedge clk ornegedge rst)
if (! rst)
Y <= 1'b0;
else
Y <= A ;
```

For normal Verilog semantics truth table for the above code is:

Clk	Rst	Y
0 -> 1	1	A
0 -> x	1	A
x -> 1	1	A
0	1 -> 0	0
0	1 -> x	A
0	x -> 0	0

Table 4.5 Edge sensitive process Truth table without xprop

Below is the truth table for xprop for all merge types. If there is a clean edge (0->1), then the next value for the flop output will be the A input. If there is an unclean edge, then the current value of the flop is merged with the d input.

Clk	rst	V-merge	T-merge	X-merge
0 → 1	1	A	A	A
0→X	1	A	Merge(A,Y(t-1))	X
X→1	1	A	Merge(A,Y(t-1))	X
0	1→X	0	Merge(A,Y(t-1))	X
0	X→0	A	Merge(A,Y(t-1))	X
0	1→0	0	Merge(A,Y(t-1))	X

Table 4.6 Edge sensitive process Truth table with xprop

4.4 X-Propagation on Latches

Latches are described in Verilog with an IF statement that does not have an else branch, as shown below.

Example

```
always@(*)
if(sel)
 Y <= A ;
```

Because of the missing branch, whenever the clock to the latch is X, this causes a merge of the current value of the latch with the data input.

Here is the truth table below:

Sel	A	V-merge	T-merge	X-merge
X	0	Y(t-1)	Merge(0,Y(t-1))	X
X	1	Y(t-1)	Merge(1,Y(t-1))	X

Table 4.6 Latch Truth Table with Xprop

Chapter 5. Bugs/Issues found

Enabled X-propagation on graphics design and found some bugs, mentionedthem below.

Issue 1. X-propagation on if statements

Below is the problem code.

```
always@(A or B or sel)
begin
if (sel)
        Y <= A;
else
        Y <= B;
end
```

Fig 5.1 problem code – if statements

In regular mode truth table is

Sel	Y
1	A
0	B
X	B

Table 5.1 Truth table regular mode – if statement

While in X-propagation mode,

Sel	A	B	Y
X	0	0	0
X	0	1	X
X	1	0	X
X	1	1	1

Table 5.2 Truth table Xprop mode – if statement

Figure 1 and 2 shows the behavior of output signal when select line is X for normal Verilog semantics and with X-prop respectively. When select line is X then in normal scenario output follows B while in case of X-prop when both signals (A and B) are same then it gives the same value as output otherwise X

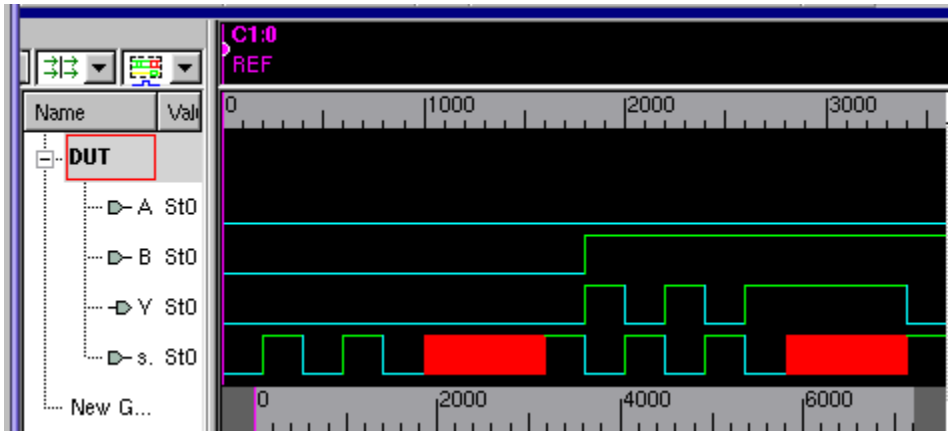

Fig 5.2 Simulation results without Xprop

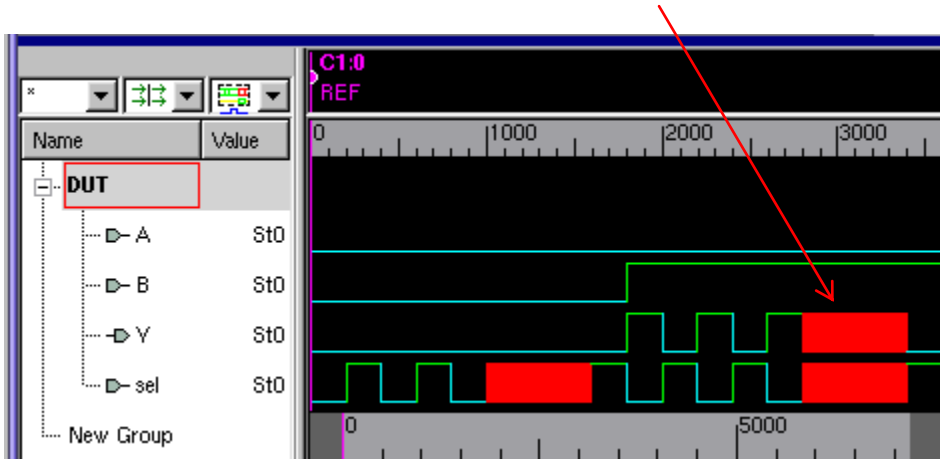

Fig 5.3 Simulation results with Xprop

Issue 2.X-propagation on case statement.

Problem code

```
always@(A,B,sel)
begin
case(sel)
        'b1: Y<=A;

        'b0: Y<=B;
endcase
end
```

Fig 5.4Problem code – case statement

In regular mode truth table is

Sel	Y
1	A
0	B
X	prev Y

Table 5.3 Truth table regular mode – case statement

While in X-propagation mode

Set	A	B	Y
X	0	0	0
X	0	1	X
X	1	0	X
X	1	1	1

Table 5.4 Truth table xprop mode – case statement

The output is always an X, because whenever there is a merge, the resultant with be an X. A case condition that contains X may result in wildcard-like behavior with more than one matching case item, including the default, therefore Xprop treats each matching case item as a possible branch and applies the same merging function it used to handle the if statements.

Here are the waveforms for above example with normal scenario and with xprop.

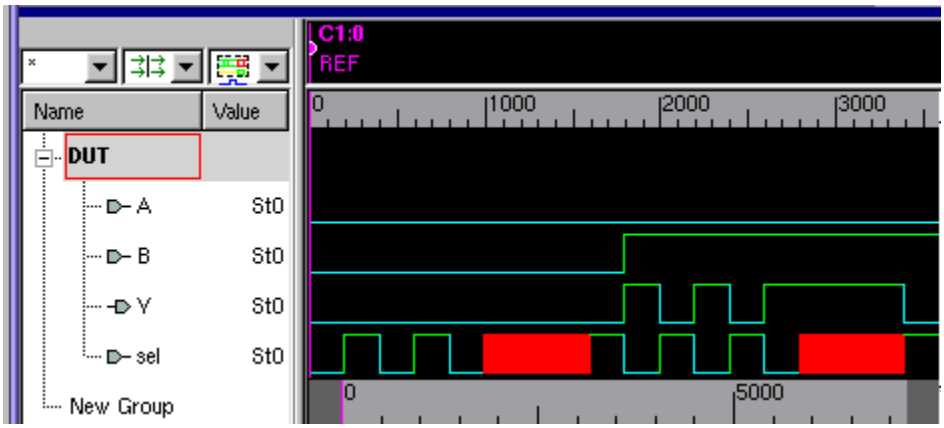

Fig 5.5 Simulation results without Xprop

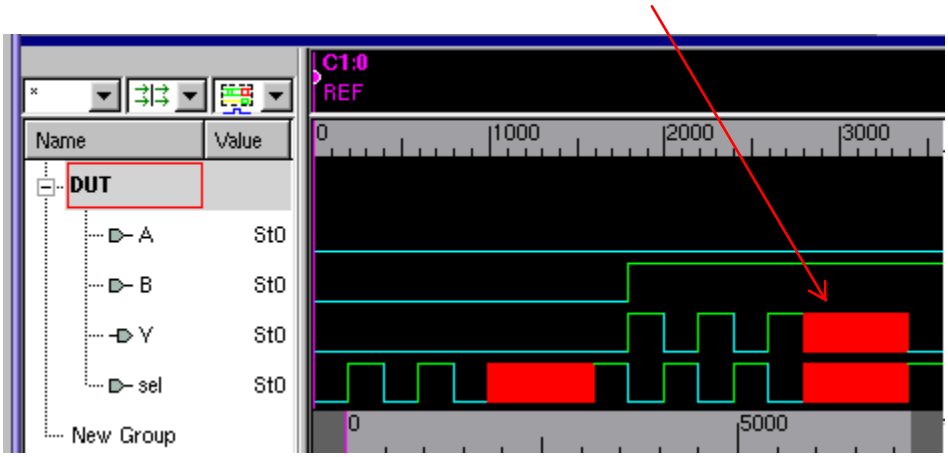

Fig 5.6 Simulation results with Xprop

Issue 3. X-propagation on Edge sensitive process

Problem code

```
always@(posedge clk or negedge rst)
begin
if (!rst)
        Y <= 1'b0;
else
        Y <= A;
end
```

Fig 5.7 Problem code – Edge sensitive process

In regular model truth table is

Clk	Rst	Y
0 → 1	1	A
0 → X	1	A
0	1 → 0	0
0	1 → X	A

Table 5.5 Truth table normal mode – edge sensitive process

While in case of X-propagation

Clk	Rst	Y
0 →1	1	0
0 → X	1	X
0	1 → 0	0
0	1 → X	X

Table 5.6 Truth table xprop mode – edge sensitive process

Here are the wave forms for normal Verilog semantics and with x-prop respectively.

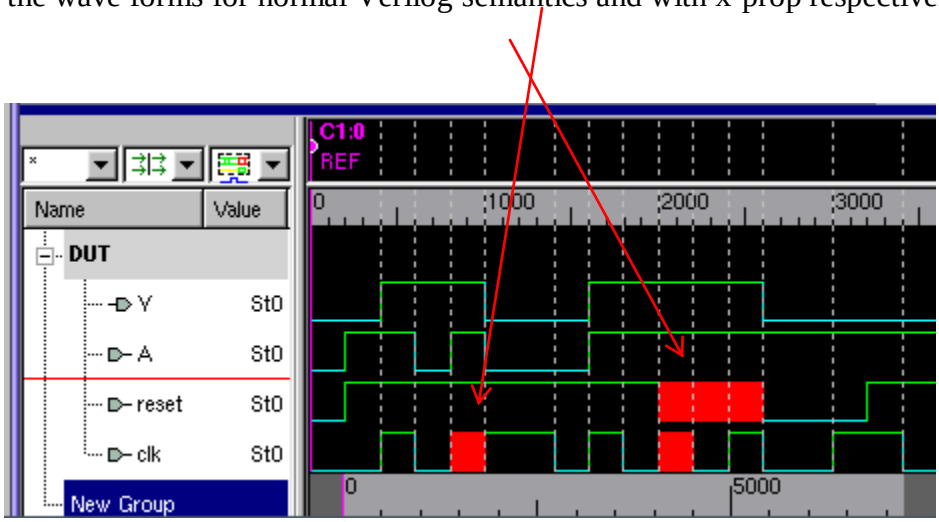

Fig 5.8 Simulation results without Xprop

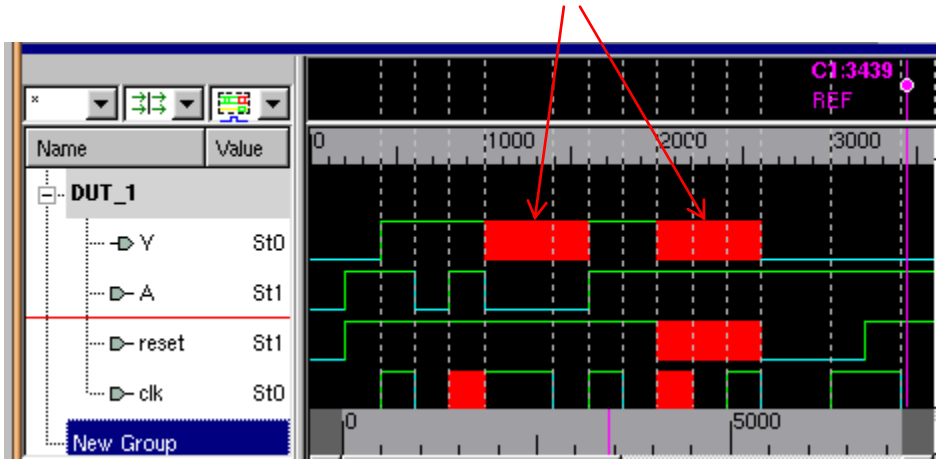

Fig 5.9 Simulation results with Xprop

Chapter 6. Limitation

Limitation of xprop simulation is, there is a runtime performance degradation whereas a little or no impact on memory consumption.

Xprop runs take more time than normal run. Here is a comparison for a design between normal scenario and X-propagation.

	Without Dump	**With Dump**
Normal Run	3.55 Hr.	8.18Hr.
Xprop+LPS Run	7.5 Hr.	23.78 Hr.
(Xprop/Normal)	2.11	2.91

Table 6.1 Time analysis with & without Xprop

Here Normal Run is with normal Verilog semantics while Xprop+LPS Run have X-propagation and LPS (Low Power Simulation) flavors. Here LPS is an enhancement that boosts Xprop as power aware simulation semantics.

Chapter 7. Conclusion

There are alternative ways to found X related bugs in the design.

- GLS is helpful but become more expensive and time consuming as design complexity increases.
- There are some different approaches to write RTL codes but due to readability of the codes and different design style for different designer it is difficult to apply. Also there are various IP reuse in the design and it's hard to change codes.
- X randomization has its own problem like more simulation cycles need and also gives only superficial coverage.

X-propagation also has time related limitation. But as a long time goal it helps to find bugs present in the design with more efficiency.

Actual design bugs were found with X-propagation which would have remained uncovered with regular RTL simulations. In normal scenario these issues are caught in late phase like GLS. This reduces overall time and become cheaper than GLS

Another advantage of enabling this at RTL level is the ability to reuse the existing verification environment as opposed to building a separate environment or model. The enhanced x-propagation semantics can be extended to work with other simulation modes resulting in a better platform for verifying RTL design. One such mode is enhancing x-propagation semantics with power aware simulation.

Chapter 8. References

1. X-Propagation Woes: Masking Bugs at RTL and Unnecessary Debug at the Netlist, IEEE paper by Lisa Piper, Vishnu Vimjam, Real Intent, Inc. , Sunnyvale, CA USA

2. X-Propagation : An Alternative to Gate Level Simulation , SNUG paper by Adrian Evans, Julius Yam, Craig Forward

3. Mike Turpin, "the dangers of living with an X (bugs hidden in your Verilog)", SNUG Boston, Sep. 2003

4. Xprop_user_guide_Oct_2012 , user guide by Synopsys, Inc. : https://solvnet.synopsys.com/retrieve/040022.html

5. X-Optimism Elimination during RTL Verification - Austin, 2012, SNUG paper by Robert Booth (Freescale); Bruce S. Greene, Arturo Salz (Synopsys, Inc.)

Part-2
A new debug methodology

Chapter 1. Introduction

When we are at initial phase of our design, we need to debug the design multiple times. We need some debug capabilities from your simulation tool. To turn on these debug capabilities simulation tool need to be run in debug mode. This mode is typically used when we need to debug the design using debug tools.

UCLI (Unified Command-Line Interface) commands used to force signals, to write into a register/net. Simulation tool has different compile time options for debug mode. These compile time options enable read/write access and callbacks to design nets, memory callbacks and assertion debug.They help to run interactive simulation when the design is compiled with this option. Also helps to set value and time breakpoints. These compile time debugoptions give visibility-control and can track the simulator line by lined and setting breakpoints within the source code.

But with all these features, this debug capability also comes with some limitation. They impact on runtime. These debug options disable tool optimizations and also have huge impacton the performance. That's why the need of a new debug methodology whichwould give better run time performance with the same debug capabilities was very crucial.

During this project I have worked on this new methodology that gives better run time performance with the same debug capabilities over existing debug options.

Chapter 2. Current flow

For debug perspective we pass some files during test execution and these files have forces for different signals. From now onwards we called these files as Include files.

In current flow we enable debug (force) capabilities on entire design during compilation time and then pass the Include files during test execution.

There is also an another way, instead of enable debug (force) capabilities on entire design we enable debug capabilities on selected modules by definingtheir accessibilityin separate files. The files are passed during compilation time and only those signals which are covered by the modules whose accessibility defined in the files are being able to force.

Drawback of this flow is its impact on runtime performance. Its performance is reduced when debug capabilities are enabled compare to normal run.

On most of the cases debug capabilities are enabled on all the modules in the design, including those which actually do not require them. So this will impact badly on performance impact. Ideally users do not need to run the test with full debug option which enables write/force capabilities across all modules unnecessary. Debug capabilities are required only when user want to debug the test failures.

Chapter 3. A new debug (force) methodology

The performance can be improved if you are able to control the access rights to enable debug capabilities. If simulation tool can identify minimal access capabilities required on modules and signals and enable only that set of capabilities on modules and signals then we can see significant runtime improvements.

In this new methodology, simulation tool detects required access capabilities during elaboration by reading Include files and enable the optimal capabilities internally up-front.

The main idea of this methodology is to enable force capabilities on as small as part of the design as possible, thereby improving run time performance. And another motivation behind this there is no requirement to change current build and run tools.

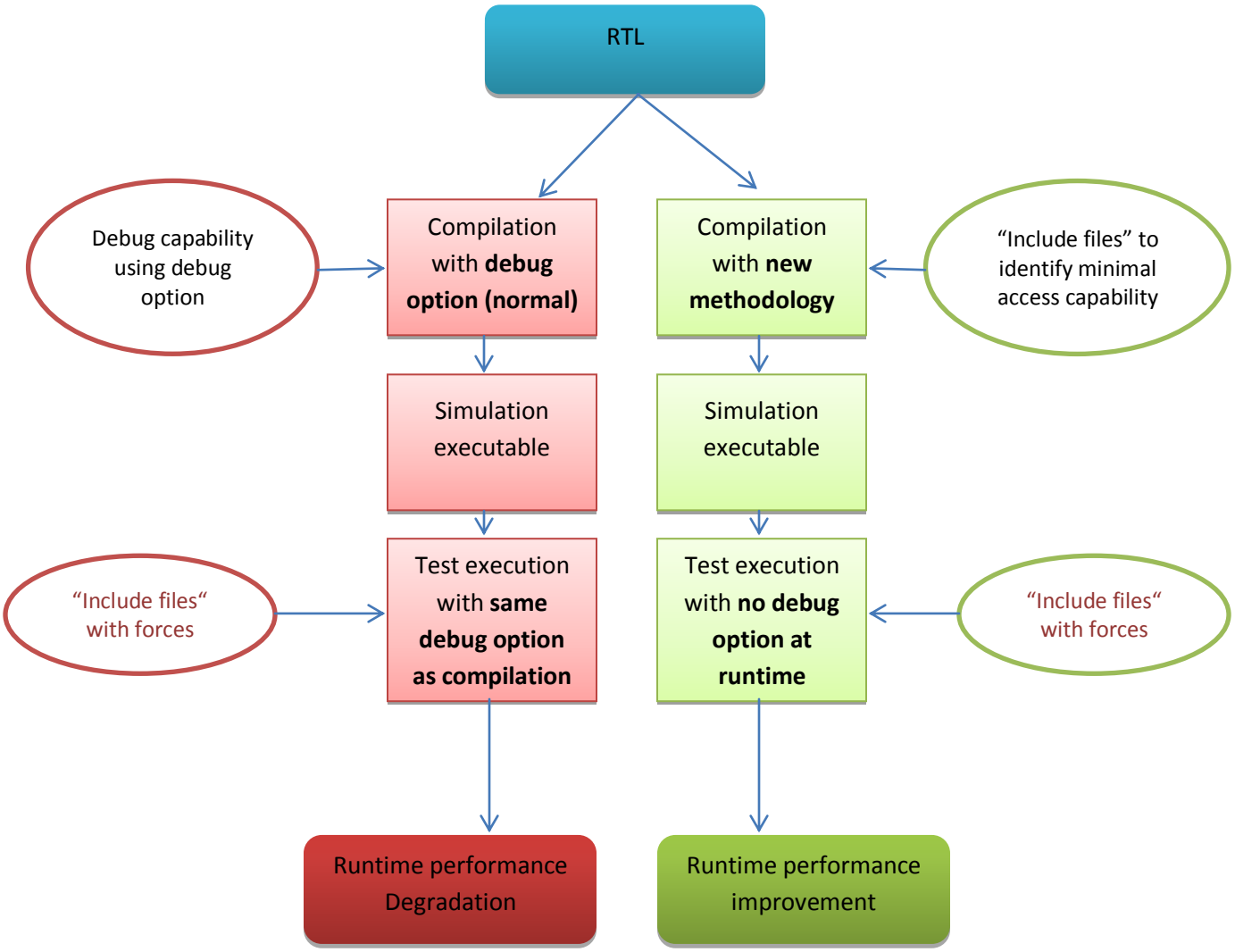

Fig 3.1 Flow diagram for debug methodologies

Chapter 4. Experiments

There are various nightly runs going through. Huge amount of time and resources are occupied by them. It's very important to improve run time performance and memory saving as much as possible.

By default all tests compilation use one of the debug options and with the same debug option tests execution is being done. For this Intel had created a flow to pass these options to simulation and debug tools. There are two methodologies.

1. Enable Force capabilities on entire design and then during run time Include files which have all forces are passed. (give more debug capability)
2. Enable For capabilities on selected modules on compilation time and then during run time Include files which have all forces are passed. Manual efforts need to identify appropriate modules.

Now with this new force methodology we pass the Include file (files) at the time of compilation also with a new use case scenario and without passing any debug option. Here these Include files must cover all the forces which are required during test execution.

With the current existing flow we can't use this methodology because there is requirement of one of the debug options at compilation time and run time. So there was need to change current flow as per this new force methodology. There some problems were faced.

1. There are some Include files need to be passed as according to context name only.
2. Current flow didn't have all the forces in Include files only. There were also some other ways to include forces at test execution time.
3. There was requirement to pass all the Include files covering all the forces at compilation time also and only those Include files should be passed during test execution time.

We had created a new flow to accommodate this methodology with its all requirements.

We first used the force methodology on a small DUT and we had seen significant difference in run time while ran with the new debug methodology.

As per requirement for the force methodology we had created a new Include files that covers all the forces which are required during test execution. We had compiled design without any debug option and passed the Include files and as a result we got noticeable runtime improvements.

We then moved further to big section that covered 14 DUTs.

Chapter 5. Problem facedfor thedebug methodology

Include files

There were modifications needed to generate the Include files in terms of syntax compared to normal case.

Aim was to enable read/write capability on as small as a part of the design as possible. To do that simulation tool finds smallest part of the designwith help of Include files we passed. Tool assumes that these are the only forces on which force capability should be on during run time. So it gives error if any extra signal being forced during test execution as tool unable to force the signal. There is also different kind of interpreter who extracts all these signals,instances or modules from the design to enable force capabilities on that and hence little extra care need to write the Include files in term of syntax.

We had created a script to convert all these forces into compatible format which can be understand by interpreter during compilation time.

Chapter 6. Bugs found

We found bugs in the tool while doing experiments on the new debug methodology. Here blow design code covers some of the bugs.

6.1 Signals under structure in design.

One issues found that if we force any signal which defined under stuct then these methodology didn't capable to enable force on that signal. Here it shown in below example.

```
module ripple_counter(q,clk,rst);
input clk,rst; output [3:0] q;
T_FF tff0(q[0],clk,rst);
T_FF tff1(q[1],q[0],rst);
T_FF tff2(q[2],q[1],rst);
T_FF tff3(q[3],q[2],rst);
endmodule

module T_FF(q,clk,rst);
output q; input clk,rst; wire d;
D_FF dff0(q,d,clk,rst);
not n1(d,q);
endmodule

module D_FF(q,d,clk,rst);
output q; input d,clk,rst; reg q;
always@(posedge rst or negedge clk)
if(rst) q=1'b0;
else    q=d;
endmodule
```

Fig 6.1 ripple counter

```
module ripple_counter_tb ();

reg clk;
reg rst;
wire [3:0] q;

typedef struct{
        reg [3:0] reg1;
} ads;
ads ads1;

ripple_counter r1 (q,clk,rst);
initial clk=1'b0;
always  #5 clk=~clk;
initial begin
        #15 rst=1'b0;
        #180 rst=1'b1;
        #10 rst=1'b0;
        #20 $finish;
end
always @(q or clk or rst)
      #1 $display("At t=%t rst=%h q=%h clk=%h", $time,rst,q,clk);
endmodule
```

Fig 6.2 ripple counter test bench

Here as shown in above code we have a signal reg1 under structure ads1 of struct type ads and below Include file used to enable force capability on signals.

```
force -deposit ripple_counter_tb.rst 'b1
force ripple_counter_tb.ads1.reg1 'b1
```

But tools can't enable force on the signal reg1 and gave FORCE ERROR as run time. It should be given warning during compilation time itself.

This issue turned out as a bug in tool.

6.2 Issue when any signal doesn't exist.

If we have some signals and modules those are actually not exist in our design then simulation tool reacts differently.

```
force -deposit ripple_counter_tb.rst 'b1
force ripple_counter_tb.ads1.reg1 'b1
force ripple_counter_tb.ads2 'b0
```

In the same design as stated above if we have Include file that any signal not exist in our design then it enabled force capability on its module itself. So here instead of signal ads2 simulator enabled force capability on module "ripple_counter_tb" itself. In some cases the module consist of so many signals those are not necessary to be force enable. So it might be possible to run time degradation due to enabled force capability on huge amount of unnecessary signals. Also we should get warning at compile time if any signal doesn't exist.

This issue turned out as an enhancement scenario.

6.3 Issue when any module doesn't exist.

If we have any module itself not exist in our design, in that case simulation tool gave fatal error.

```
force -deposit ripple_counter_tb.rst 'b1
force ripple_counter_tb.ads1.reg1 'b1
force ripple_counter_tb.module1.ads2 'b0
```

In the same design stated above, here module modelue1 doesn't exist in our design and that is passed through our include file. In that case simulation tool gave fatal error with no information regarding which module created problem.

If Include file has such modules those are not exist in our design then it should be some meaningful compilation time error or warning comes up. This issue turned out as a bug in tool for this methodology.

Chapter 7. Results

In our design there were 14 clusters and we got runtime performance improvements as well as memory saving in all clusters

As an average in run time performance ~ 45% improvements.
And runtime memory ~ 17% reductions we achieved.

Normal run (with debug option) v/s New debug methodology

Cluster	Normal/new debug methodology	
	Runtime performance	Memory performance
Dut1	1.63	1.17
Dut2	1.30	1.09
Dut3	1.76	1.28
Dut4	1.40	1.12
Dut5	1.42	1.11
Dut6	1.39	1.06
Dut7	1.63	1.05
Dut8	1.45	1.20
Dut9	1.31	1.21
Dut10	1.44	1.21
Dut11	1.55	1.30
Dut12	1.09	1.08
Dut13	1.59	1.27
Dut14	1.31	1.26
Avarage	1.45	1.17

Table 7.1 Run time comparison normal/new debug methodology

Chapter 8. Conclusion

There were some compatibility issues, fixes for some bugs and tool enhancement needed for this new methodology and hence it required specific and above tool version to get the best performance. There was no potential risk identified so far with this methodology.

One limitation is that if we want to add any new force or any new Include file then you must recompile your design.

Overall we are seeing very significant runtime performance as well as runtime memory improvement and this methodology can be seen as next generation simulation technology.

The work has been recognized and achieved Instant Recognition award for excellent work on this new debug methodology and help to get new technology ready for production. With the technology we got 1.4X runtime improvements.

Chapter 9. Other Contribution

- Tested out automated QA flow that download newly updated tool, run tests on graphics design and give extracted outputs.Found out some issues in the scripts.

- Done performance analysis of debugging tools on graphics designs. Compared the performance of new enhanced tool with present tool.

    - Enhanced a Perl script that gives performance analysis and comparison between two runs.
      Input of the script is path of output directory of both runs.Output is comparison of both runs in terms of simulation cycles, user time, and tests status.

- Worked closely with tool vendor to fix the enhancement suggest by them on Intel design.

- Worked on various issues (tickets) related to RTL simulation tool.

- Created a Perl script that searches all modules name presents in all libraries and list out which module is present in which library.Output of the script is total module name, total library name and libraries related to each module

- Performed quality analysis of new version simulation and debugging tools on graphics designs.

- Worked on an internal tool to assist IP provider to make IP as per IP standards.

    - Work was recognized and awarded with Instant Recognition Award.