Unique Methodology to Streamline the Checking of Design Tie-offs

Varun Ramesh – NXP Semiconductors

Amol Bhinge – NXP Semiconductors

Jay Dutt – Synopsys Inc.

NXP Semiconductors.

Austin, TX, USA

www.nxp.com

ABSTRACT

An important procedure to ensure comprehensive verification effort is to check all of the tied-off port signals in the design. Tied off signals may go through cursory, manual reviews. These tied-off signals may not matter at all, or they may be functionally relevant and are tied to a specific value for a reason. These tied-off signals may also lead to bugs if tied to the wrong value and then not reviewed. We explored Certitude's signal-faulting capability to specifically inject faults on these tied-off signals and then to run our regression suite. This procedure gives us a comprehensive report of our checkers' coverage for these tied-off signals. This paper will explain the setup, results, analysis and actions for this novel exploration. The paper will also discuss proposed enhancements to deliver production flow with seamless experience.

Table of Contents

Table of Figures

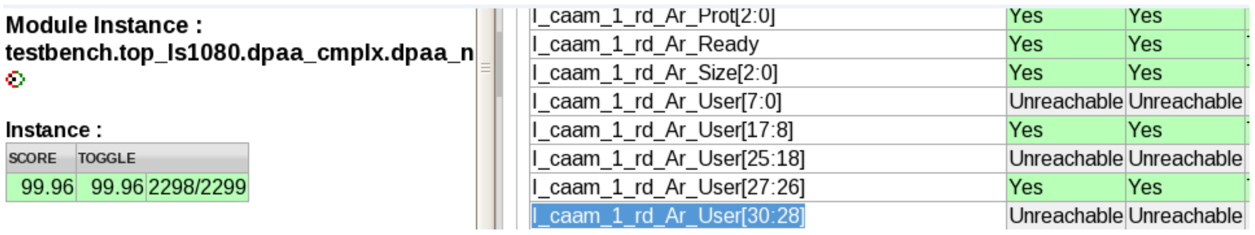
1. Introduction

For every SoC design that our group verifies, we face the challenge of having to review all the signals in the design which are tied-off. There is a possibility of leaving a bug in the design un-fixed if these signals are not reviewed. These signals show up as Unreachable when viewed in a toggle coverage report from the Synopsys Universal Report Generartor (URG). The analysis involves first carefully examining the value of the tie-off then determining if the value is correct from a functional and architectural point of view. This prompted us to deploy Certitude to make sure none of the tie-offs result in logic bugs or at the least are accounted for in our verification efforts. This is a very novel approach which targets Certitude's capabilities to address our specific challenge. Certitude is already known by us as a tool which measures Verification Environment quality. Certitude does this by injecting faults (artificial bugs) into the design, and then measuring the Verification Environment's ability to Activate, Propagate, and Detect those faults. Certitude is already in use by various Verification teams at NXP, and it has an established flow here. We ran Certitude and found areas where we could improve the Checkers and some places which could be waived after diligent discussions with stakeholders.

2. Our Verification Challenge

Our SoC Verification Environment is a UVM based environment. Our testcases consists of running one or more multiple virtual sequences with random seeds. Virtual sequences accomplish almost all of the scenarios right from system reset to post reset functional and connectivity checks. The virtual sequences do this by calling a large number of sequences at different phases of the simulation. The testbench also makes use of a large numbers of UVM monitors/scoreboards and assertions from many VIPs in the environment. The testcases are grouped feature-wise into regressions and these results are used for collecting our toggle coverage results as well. As we all know, the signals which are tied-off or assigned a Constant value as per VCS and the URG are all reported as Unreachable in the coverage report. We always take toggle coverage reports seriously. And as part of it, we make sure to review all the Unreachable signals in the design, along with logic designer's feedback.

Figure 2.1 shows a partial listing from the URG for Unreachable signals for the DPAA_NoC block.



Module Instance :
testbench.top_ls1080.dpaa_cmplx.dpaa_n

Instance :

SCORE	TOGGLE	
99.96	99.96	2298/2299

I_caam_1_rd_Ar_Prot[2:0]	Yes	Yes
I_caam_1_rd_Ar_Ready	Yes	Yes
I_caam_1_rd_Ar_Size[2:0]	Yes	Yes
I_caam_1_rd_Ar_User[7:0]	Unreachable	Unreachable
I_caam_1_rd_Ar_User[17:8]	Yes	Yes
I_caam_1_rd_Ar_User[25:18]	Unreachable	Unreachable
I_caam_1_rd_Ar_User[27:26]	Yes	Yes
I_caam_1_rd_Ar_User[30:28]	Unreachable	Unreachable

Figure 2.1 Examples of Unreachable signals of DPAA_NoC block

Our SoC RTL designs are highly complex. There are two possibilities with every Unreachable signal. First, the tie-off value doesn't impact functionality and so the signal can be tied-off to any value. The second possibility is that it impacts the functionality in a specific way and is tied-off for a specific reason. The challenge lies in determining which category each of the Unreachable signals belong to, and then how to ensure this is in fact the case with RTL design implementation. Writing explicit testcases to verify these Unreachables might be a huge task in itself. This is where Certitude's capabilities of signal faulting are put to use. We asked ourselves: "Why not feed all the interesting Unreachable signals to Certitude for faulting and analyse the regression report to see if the results match the expectations or not?" Again, anything new and different from the regular Certitude flow comes with its own learning experiences.

3. Our Certitude Set Up

We considered Certitude as a method for automating this process of injecting faults and evaluating results. Certitude has been in use by various groups within NXP to measure Verification Environment quality already for many years. Certitude has a proven flow within NXP's compile, build, and run infrastructure.

Certitude measures the Quality of the Verification Environment by inserting faults (artificial bugs) into the design, then measuring the Activation, Propagation, and the Detection of these faults by the Verification Environment. The diagram in Figure 3.1 describes this fault injection mechanism.

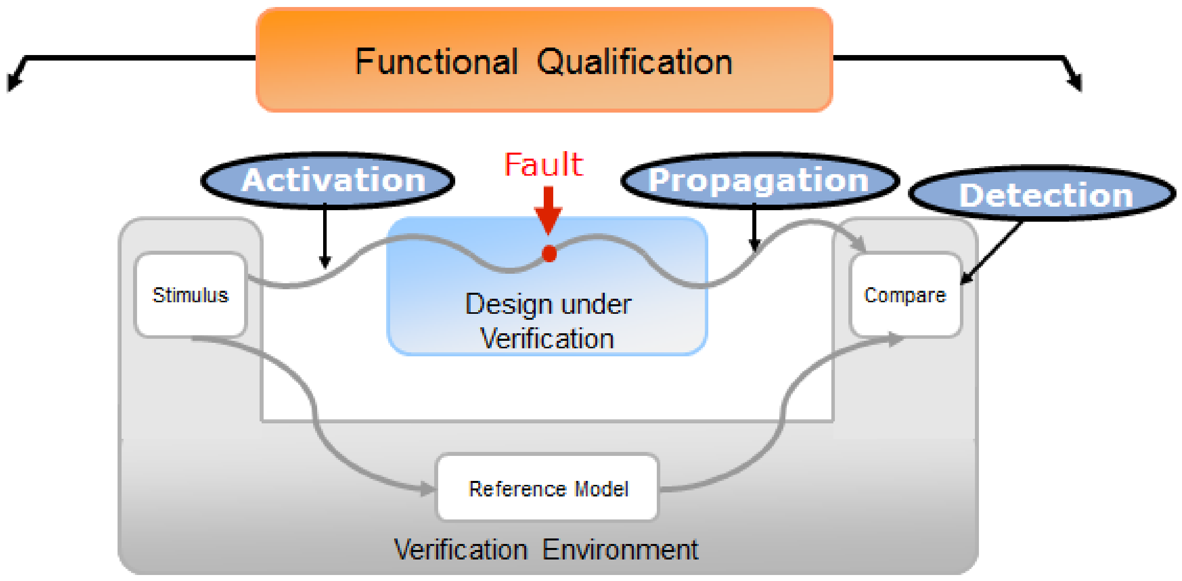
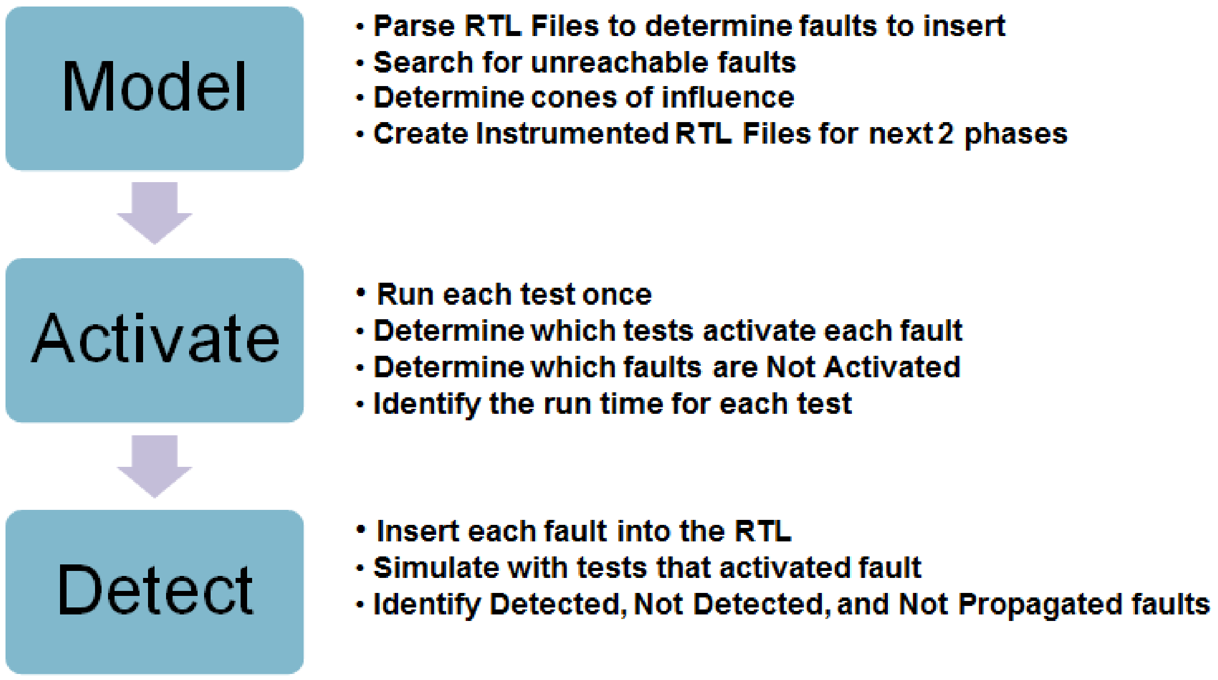


Figure 3.1 Functional Qualification with Certitude

Certitude requires three phases to determine the Activation, Propagation, and Detection status of the faults: Model, Activate, Detect. The diagram in Figure 3.2 shows this flow and the descriptions of each phase.



- Parse RTL Files to determine faults to insert
- Search for unreachable faults
- Determine cones of influence
- Create Instrumented RTL Files for next 2 phases

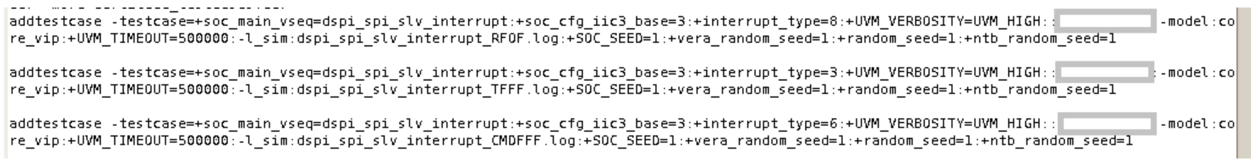
- Run each test once
- Determine which tests activate each fault
- Determine which faults are Not Activated
- Identify the run time for each test

- Insert each fault into the RTL
- Simulate with tests that activated fault
- Identify Detected, Not Detected, and Not Propagated faults

Figure 3.2 Certitude phases and descriptions

Certitude allows the User to choose which instances of modules, and then which signals within those modules, to instrument faults.

After the Model phase, the next phases to run are the Activate and Detect phases, both of which will run simulations. In order to run these phases, Certitude is set up with a compile script, an execute script, and a list of test cases to run. The testcase list format for Certitude had a couple of limitations: no spaces are allowed (so we replaced spaces with ":"), and special punctuation like double-quotes need to be substituted out. The diagram in Figure 3.3 shows a few lines from the Certitude testcase list:



```
addtestcase -testcase=+soc_main_vseq=dspi_spi_slv_interrupt:+soc_cfg_iic3_base=3:+interrupt_type=8:+UVM_VERBOSITY=UVM_HIGH::[redacted] -model:co
re_vip:+UVM_TIMEOUT=500000:-l_sim:dspi_spi_slv_interrupt_RFOF.log:+SOC_SEED=1:+vera_random_seed=1:+random_seed=1:+ntb_random_seed=1

addtestcase -testcase=+soc_main_vseq=dspi_spi_slv_interrupt:+soc_cfg_iic3_base=3:+interrupt_type=3:+UVM_VERBOSITY=UVM_HIGH::[redacted]:-model:co
re_vip:+UVM_TIMEOUT=500000:-l_sim:dspi_spi_slv_interrupt_TFFF.log:+SOC_SEED=1:+vera_random_seed=1:+random_seed=1:+ntb_random_seed=1

addtestcase -testcase=+soc_main_vseq=dspi_spi_slv_interrupt:+soc_cfg_iic3_base=3:+interrupt_type=6:+UVM_VERBOSITY=UVM_HIGH::[redacted] -model:co
re_vip:+UVM_TIMEOUT=500000:-l_sim:dspi_spi_slv_interrupt_CMDFFF.log:+SOC_SEED=1:+vera_random_seed=1:+random_seed=1:+ntb_random_seed=1
```

Figure 3.3 Testcase listing

The Certitude execute script then back-substitutes the ":" with spaces, and any other back-substitutions required. The execute script also checks the log files to provide the definitive indication of whether the test Passed or Failed. This checking mechanism in the Certitude execute script must be exactly the same as what our normal regression procedure checks for Pass or Fail, so we made sure this was the case.

An important point is that the Certitude testcases must each run in a "fixed seed" manner. This is because, at the start of the Detect phase, Certitude runs a reference simulation for each test case provided so that the reference waveforms on the Probes of the design are stored in the Certitude DB (Probes are located on the DPAA_NoC top-level outputs). During the Detect phase, if the injected fault is executed with all of its Activating testcases, and all those testcases Pass, then Certitude will use the Fault versus Reference signature comparison to make a determination on whether the fault is Non-Detected (testcase drove a propagation difference on at least one of the outputs) or Non-Propagated (testcase did not create a propagation difference on any of the outputs). If the fault causes any to test fail, then the fault is Detected, regardless of its propagation status.

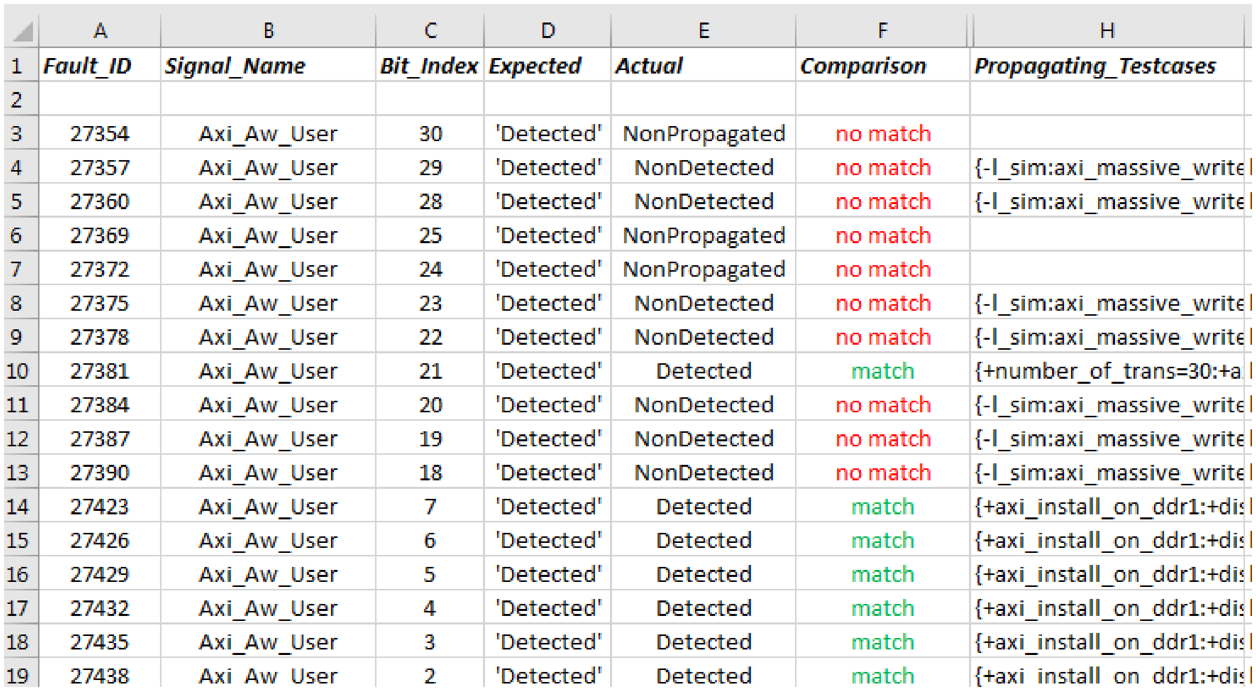
A challenge we had to start with was to determine our expected results versus actual Certitude results for each Unreachable signal. We thought that it would be descriptive if we were to use new terms here for our verification results analysis: "expected" and "non-expected", in order to describe the results analysis of this Certitude activity. The expected results in this case are:

1) The tie-off is not expected to have any functional impact. In this case the expectation is that none of the testcases from the regression should fail even after the fault is injected. Therefore we expect Non-Detected status for the fault. If a test does fail (fault is Detected), there is an indication to us that something is wrong and the root cause reason for the test failure needs to be determined.

2) The tie-off is expected to have functional impact. In this case we look for at least one test to fail with the fault injection, so that the fault is Detected. If no tests fail (fault is Non-Detected or Non-Propagated) then this is a non-expected result. There is an indication to us that our verification checker mechanism is weak or missing in checking this design function.

The analysis and debug of non-expected Certitude results is different in comparison to conventional usage of Certitude where we debug the set of Non-Detected and Non-Propagated faults. In this case, since we have already determined the subset of faults which we expect to fail tests, we only debug the faults for which the results do not match these expectations. We have seen some actual fault detection results in the DPAA NOC block where we needed to debug Detected faults (faults which caused tests to fail), in the cases where we expected these faults to have no impact on functionality. Normally Certitude users do not need to debug Detected faults.

There is no mechanism in Certitude today where a user can directly input the expected or non-expected status for each fault, and then the corresponding results are seen as part of the standard Certitude report. Therefore we have filed an enhancement request to accomplish this objective (described later in the paper). In the meantime, we are using a Tcl script that we populate to input our expectations (called fault_expectation.tcl) and another Tcl script that is used to generate a file in CSV format (called fault_status.tcl) which contains the fault status results. The fault_expectation.tcl script is run before the fault detect phase starts, and the fault_status.tcl is run either during the detection phase (in order to see the results up to that time) or at the end of the detect phase. These results can be imported to an Excel spreadsheet. The diagram in Figure 3.4 shows a section of this spreadsheet.

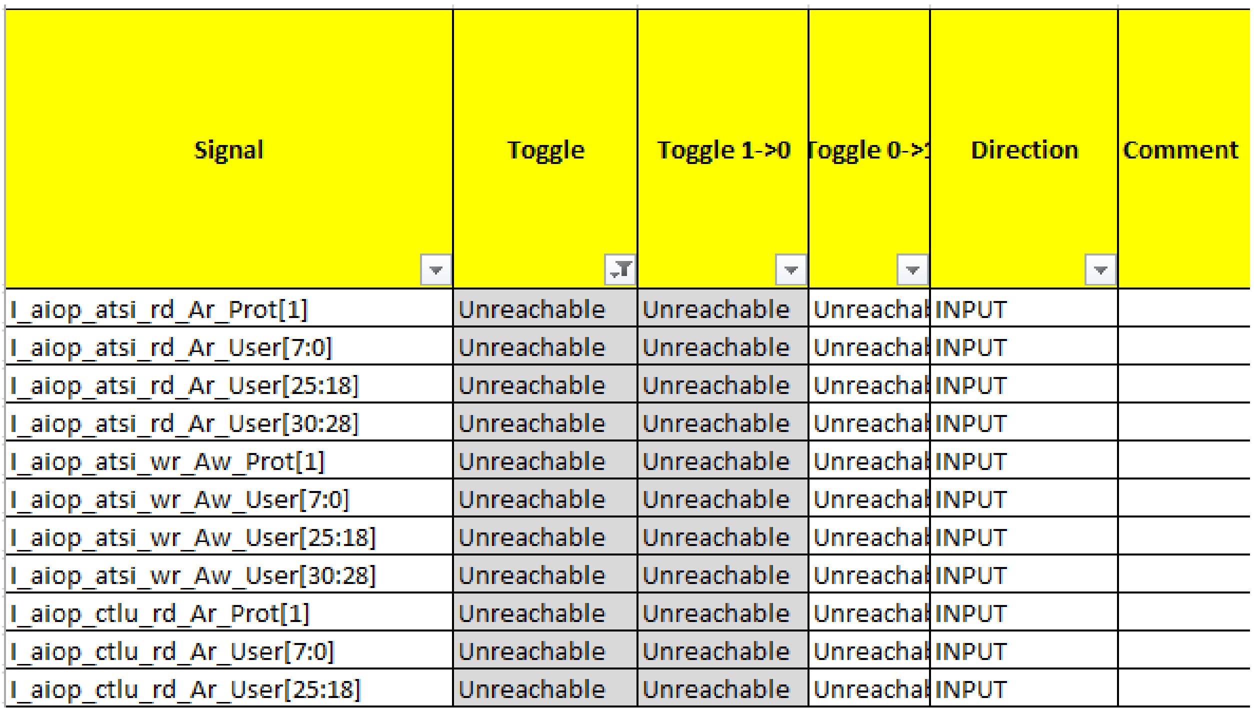


	A	B	C	D	E	F	H
1	*Fault_ID*	*Signal_Name*	*Bit_Index*	*Expected*	*Actual*	*Comparison*	*Propagating_Testcases*
2							
3	27354	Axi_Aw_User	30	'Detected'	NonPropagated	no match	
4	27357	Axi_Aw_User	29	'Detected'	NonDetected	no match	{-l_sim:axi_massive_write
5	27360	Axi_Aw_User	28	'Detected'	NonDetected	no match	{-l_sim:axi_massive_write
6	27369	Axi_Aw_User	25	'Detected'	NonPropagated	no match	
7	27372	Axi_Aw_User	24	'Detected'	NonPropagated	no match	
8	27375	Axi_Aw_User	23	'Detected'	NonDetected	no match	{-l_sim:axi_massive_write
9	27378	Axi_Aw_User	22	'Detected'	NonDetected	no match	{-l_sim:axi_massive_write
10	27381	Axi_Aw_User	21	'Detected'	Detected	match	{+number_of_trans=30:+a
11	27384	Axi_Aw_User	20	'Detected'	NonDetected	no match	{-l_sim:axi_massive_write
12	27387	Axi_Aw_User	19	'Detected'	NonDetected	no match	{-l_sim:axi_massive_write
13	27390	Axi_Aw_User	18	'Detected'	NonDetected	no match	{-l_sim:axi_massive_write
14	27423	Axi_Aw_User	7	'Detected'	Detected	match	{+axi_install_on_ddr1:+di
15	27426	Axi_Aw_User	6	'Detected'	Detected	match	{+axi_install_on_ddr1:+di
16	27429	Axi_Aw_User	5	'Detected'	Detected	match	{+axi_install_on_ddr1:+di
17	27432	Axi_Aw_User	4	'Detected'	Detected	match	{+axi_install_on_ddr1:+di
18	27435	Axi_Aw_User	3	'Detected'	Detected	match	{+axi_install_on_ddr1:+di
19	27438	Axi_Aw_User	2	'Detected'	Detected	match	{+axi_install_on_ddr1:+di

Figure 3.4 Detect results of the DPAA_NoC with "match" (expected) and "no match" (non-expected) results annotated

4. Our Results

We started to use a tracking spreadsheet to record all the Unreachable (tied-off) signals in the design that we are interested in evaluating. The specific signals for the DPAA_NoC are shown in the diagram in Figure 4.1 as captured from the spreadsheet.



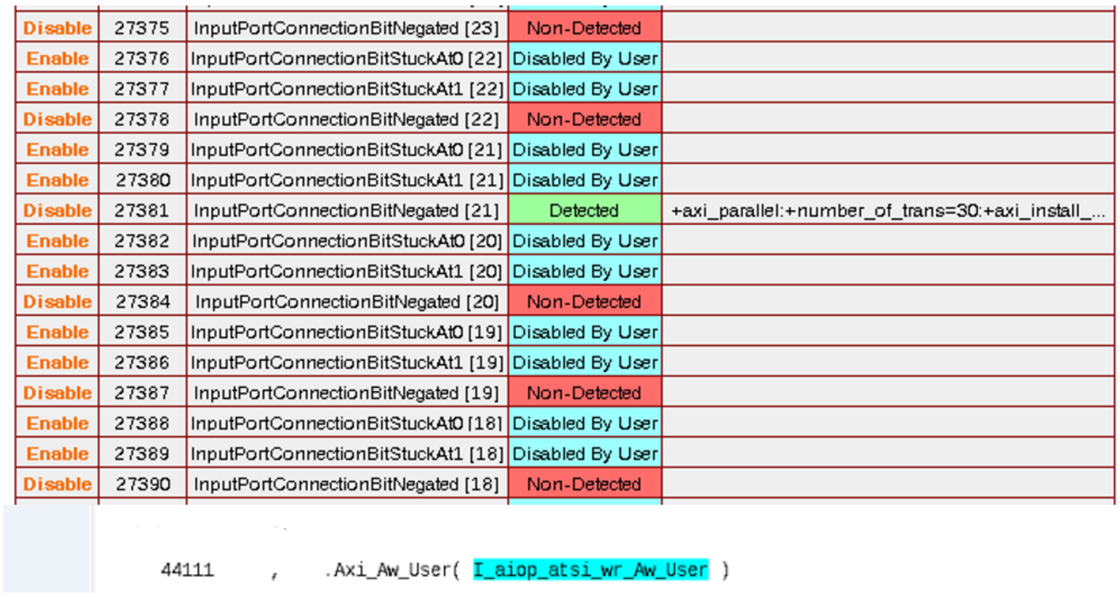
Signal	Toggle	Toggle 1->0	Toggle 0->1	Direction	Comment
I_aiop_atsi_rd_Ar_Prot[1]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_atsi_rd_Ar_User[7:0]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_atsi_rd_Ar_User[25:18]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_atsi_rd_Ar_User[30:28]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_atsi_wr_Aw_Prot[1]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_atsi_wr_Aw_User[7:0]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_atsi_wr_Aw_User[25:18]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_atsi_wr_Aw_User[30:28]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_ctlu_rd_Ar_Prot[1]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_ctlu_rd_Ar_User[7:0]	Unreachable	Unreachable	Unreachal	INPUT	
I_aiop_ctlu_rd_Ar_User[25:18]	Unreachable	Unreachable	Unreachal	INPUT	

Figure 4.1 Unreachable signals for one of the chosen modules in our design: DPAA_NoC

We then analyzed the design specification for the signals and added our expectations to the spreadsheet. These were Expect Non-Detect or Expect Detect, for each bit of these signals. This process was completed carefully over time, as we wanted to make sure our expectations were accurate to the design specification.

For this project, we took an approach of instrumenting "BitNegation" type of faults on the Unreachable (tied-off) signals, instead of applying StuckAt-1 and StuckAt-0 type faults to each signal. This eliminates redundant faults since there is no reason to inject a fault value which is the same as the tied value. By eliminating faults we do not need, the Certitude fault detect phase can finish faster. At the end of the Detect phase, the Certitude report contains the actual status for each of the injected bit-negated faults.

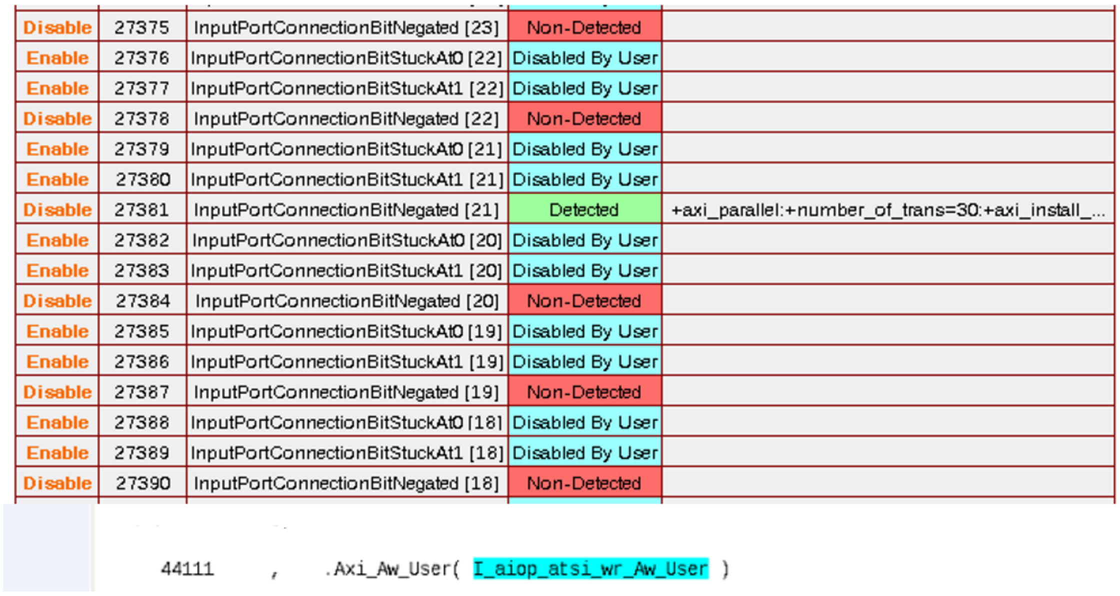
In our first fault analysis example, the diagram in



Disable	27375	InputPortConnectionBitNegated [23]	Non-Detected	
Enable	27376	InputPortConnectionBitStuckAt0 [22]	Disabled By User	
Enable	27377	InputPortConnectionBitStuckAt1 [22]	Disabled By User	
Disable	27378	InputPortConnectionBitNegated [22]	Non-Detected	
Enable	27379	InputPortConnectionBitStuckAt0 [21]	Disabled By User	
Enable	27380	InputPortConnectionBitStuckAt1 [21]	Disabled By User	
Disable	27381	InputPortConnectionBitNegated [21]	Detected	+axi_parallel:+number_of_trans=30:+axi_install_...
Enable	27382	InputPortConnectionBitStuckAt0 [20]	Disabled By User	
Enable	27383	InputPortConnectionBitStuckAt1 [20]	Disabled By User	
Disable	27384	InputPortConnectionBitNegated [20]	Non-Detected	
Enable	27385	InputPortConnectionBitStuckAt0 [19]	Disabled By User	
Enable	27386	InputPortConnectionBitStuckAt1 [19]	Disabled By User	
Disable	27387	InputPortConnectionBitNegated [19]	Non-Detected	
Enable	27388	InputPortConnectionBitStuckAt0 [18]	Disabled By User	
Enable	27389	InputPortConnectionBitStuckAt1 [18]	Disabled By User	
Disable	27390	InputPortConnectionBitNegated [18]	Non-Detected	

```
44111      ,     .Axi_Aw_User( I_aiop_atsi_wr_Aw_User )
```

Figure 4.2 shows the Certitude report status of the bit-negated faults for the "Aw_User" bits 23 to 18, which are marked as Unreachable (tied-off). From this diagram, the negated fault on bit 21 is Detected (at least one test failed, and the name of that test case is partially shown in the rightmost column). The other bits are Non-Detected (all tests pass).

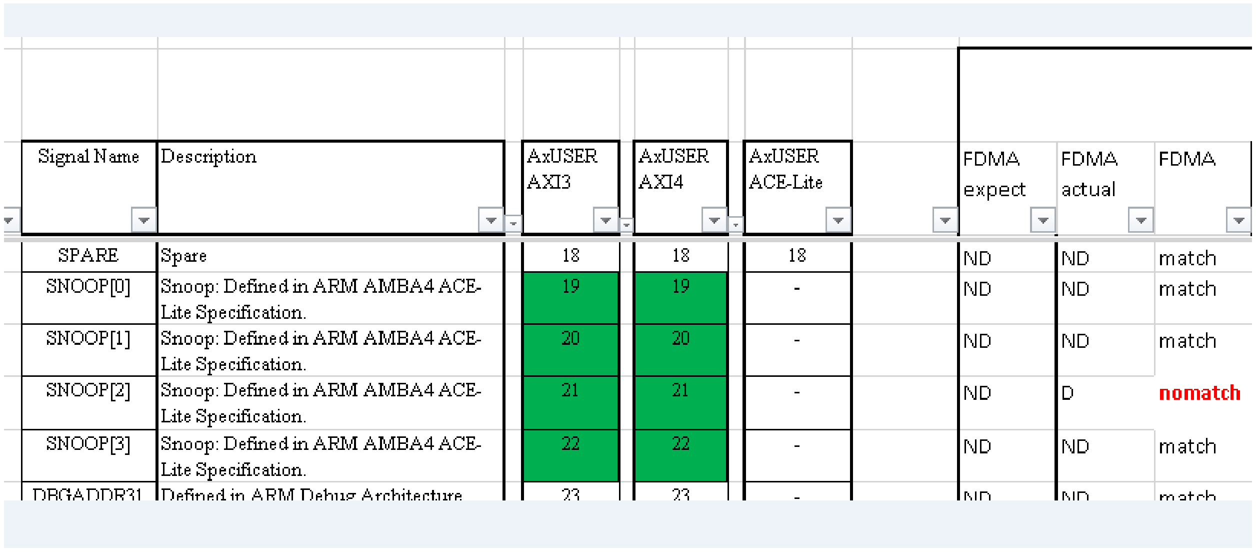


Disable	27375	InputPortConnectionBitNegated [23]	Non-Detected	
Enable	27376	InputPortConnectionBitStuckAt0 [22]	Disabled By User	
Enable	27377	InputPortConnectionBitStuckAt1 [22]	Disabled By User	
Disable	27378	InputPortConnectionBitNegated [22]	Non-Detected	
Enable	27379	InputPortConnectionBitStuckAt0 [21]	Disabled By User	
Enable	27380	InputPortConnectionBitStuckAt1 [21]	Disabled By User	
Disable	27381	InputPortConnectionBitNegated [21]	Detected	+axi_parallel:+number_of_trans=30:+axi_install_...
Enable	27382	InputPortConnectionBitStuckAt0 [20]	Disabled By User	
Enable	27383	InputPortConnectionBitStuckAt1 [20]	Disabled By User	
Disable	27384	InputPortConnectionBitNegated [20]	Non-Detected	
Enable	27385	InputPortConnectionBitStuckAt0 [19]	Disabled By User	
Enable	27386	InputPortConnectionBitStuckAt1 [19]	Disabled By User	
Disable	27387	InputPortConnectionBitNegated [19]	Non-Detected	
Enable	27388	InputPortConnectionBitStuckAt0 [18]	Disabled By User	
Enable	27389	InputPortConnectionBitStuckAt1 [18]	Disabled By User	
Disable	27390	InputPortConnectionBitNegated [18]	Non-Detected	

```
44111      ,     .Axi_Aw_User( I_aiop_atsi_wr_Aw_User )
```

Figure 4.2 Certitude Fault Detail view from Report

The Detected status of bit 21 of the Aw_User bus signal is non-expected, according to our predicted entries in the tracking spreadsheet. The diagram in Figure 4.3 shows our expectation for this bit, and the actual Certitude result from the spreadsheet. The description of the bit is also provided in order to cross reference with the specification if necessary:



Signal Name	Description	AxUSER AXI3	AxUSER AXI4	AxUSER ACE-Lite		FDMA expect	FDMA actual	FDMA
SPARE	Spare	18	18	18		ND	ND	match
SNOOP[0]	Snoop: Defined in ARM AMBA4 ACE-Lite Specification.	19	19	-		ND	ND	match
SNOOP[1]	Snoop: Defined in ARM AMBA4 ACE-Lite Specification.	20	20	-		ND	ND	match
SNOOP[2]	Snoop: Defined in ARM AMBA4 ACE-Lite Specification.	21	21	-		ND	D	nomatch
SNOOP[3]	Snoop: Defined in ARM AMBA4 ACE-Lite Specification.	22	22	-		ND	ND	match
DBGADDR31	Defined in ARM Debug Architecture	23	23	-		ND	ND	match

Figure 4.3 Details for Aw_User bit 21 expect versus actual fault status result

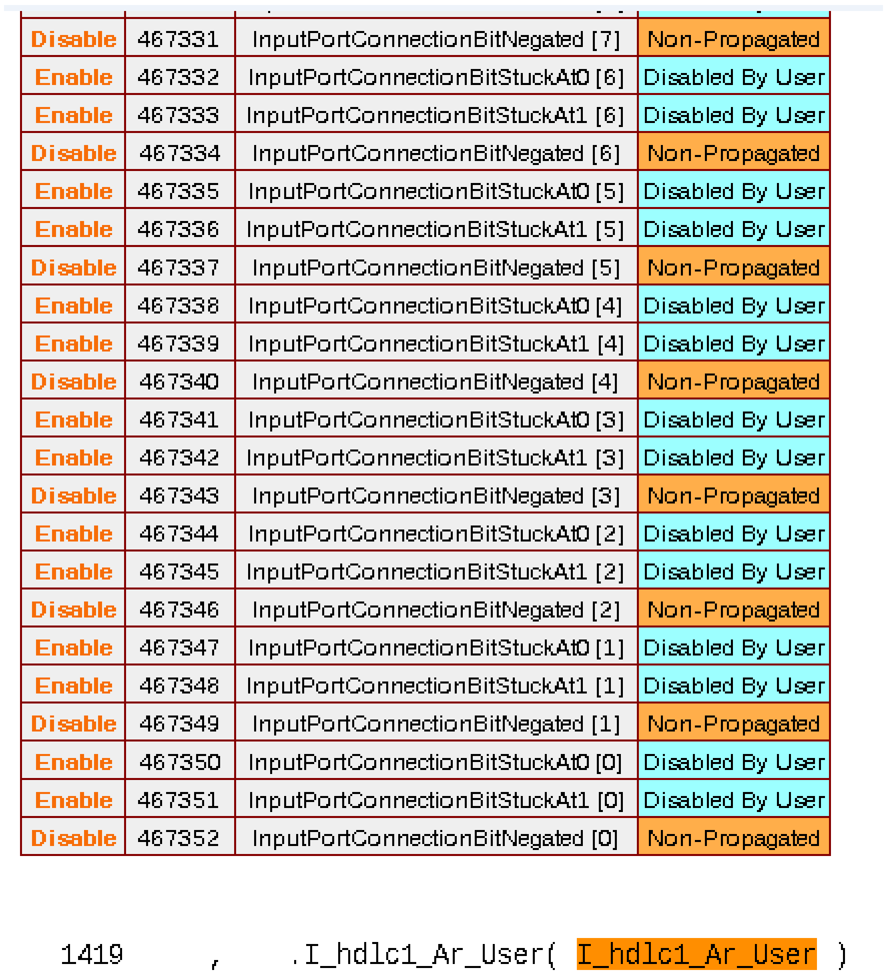
This “nomatch” non-expected result created the action item for us to root cause the reason as to why this happened. Here is a summary of our root cause analysis:

Analysis : Although the tie-off values for bit 21 was not expected to cause any functional impact at the intermediate NoC hierarchical level, it did propagate to our Interconnect IP hierarchical level. The Interconnect IP specification document clearly indicated that tieing this bit to the opposite value was invalid and would result in unpredictable behavior of the Interconnect IP. This happened to be verified in the actual Certitude results, since the fault was Detected by one of our tests.

Solution: Upon discussion with the IP Vendor‘s Tech Support team, we concluded that bit21 was a bad candidate for injecting faults.

Conclusion: This was a good learning experience for us as far as diving deeper into the Interconnect IP specification to get a better understanding.

In our 2nd fault analysis example, the diagram in Figure 4.4 shows the status of the bit-negated faults for the “Ar_User“ bits 7 through 0, which are also marked as Unreachable (tied-off). From this diagram, Certitude reports Non-Propagated status for all of these bits. The Non-Propagated status, like the Non-Detected status means that all the tests actually passed. However, unlike the Non-Detected faults, the Non-Propagated faults means that the module outputs (probes) did not see any propagation difference between the faulty and the reference simulations. This could be an indication of weak or missing stimulus (test cases), as well as weak or missing checkers.

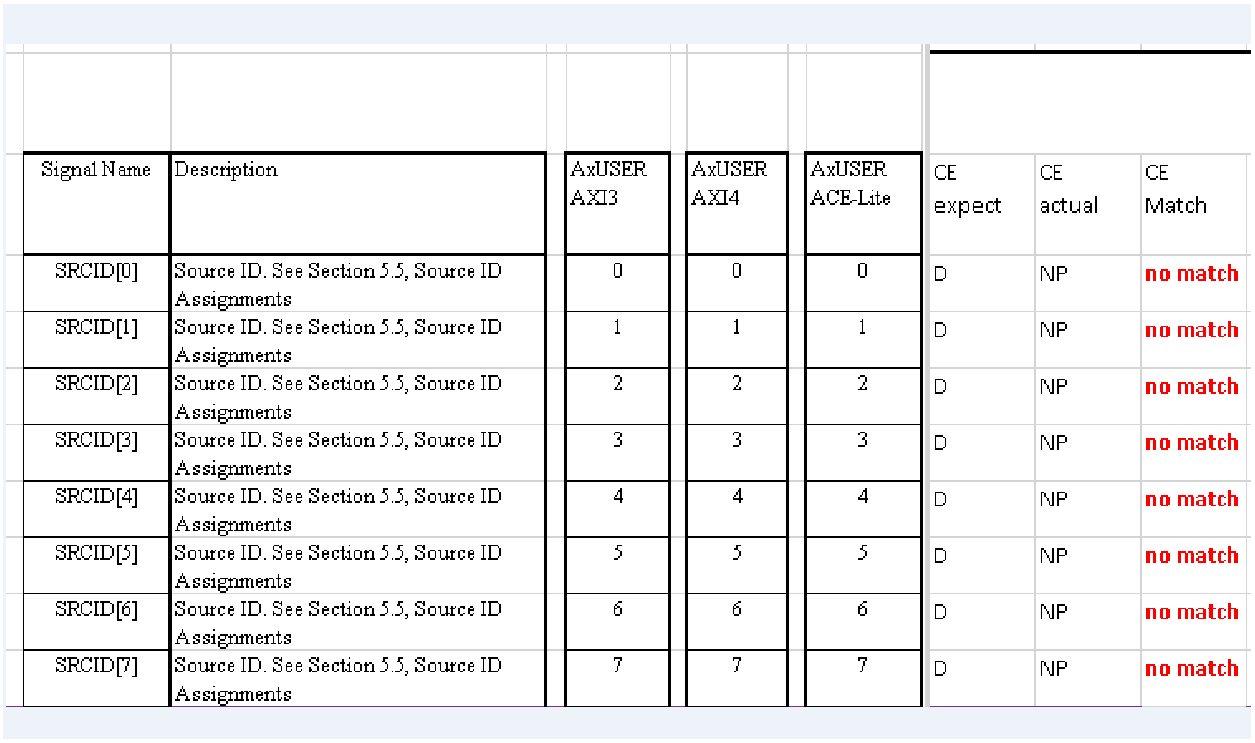


Disable	467331	InputPortConnectionBitNegated [7]	Non-Propagated
Enable	467332	InputPortConnectionBitStuckAt0 [6]	Disabled By User
Enable	467333	InputPortConnectionBitStuckAt1 [6]	Disabled By User
Disable	467334	InputPortConnectionBitNegated [6]	Non-Propagated
Enable	467335	InputPortConnectionBitStuckAt0 [5]	Disabled By User
Enable	467336	InputPortConnectionBitStuckAt1 [5]	Disabled By User
Disable	467337	InputPortConnectionBitNegated [5]	Non-Propagated
Enable	467338	InputPortConnectionBitStuckAt0 [4]	Disabled By User
Enable	467339	InputPortConnectionBitStuckAt1 [4]	Disabled By User
Disable	467340	InputPortConnectionBitNegated [4]	Non-Propagated
Enable	467341	InputPortConnectionBitStuckAt0 [3]	Disabled By User
Enable	467342	InputPortConnectionBitStuckAt1 [3]	Disabled By User
Disable	467343	InputPortConnectionBitNegated [3]	Non-Propagated
Enable	467344	InputPortConnectionBitStuckAt0 [2]	Disabled By User
Enable	467345	InputPortConnectionBitStuckAt1 [2]	Disabled By User
Disable	467346	InputPortConnectionBitNegated [2]	Non-Propagated
Enable	467347	InputPortConnectionBitStuckAt0 [1]	Disabled By User
Enable	467348	InputPortConnectionBitStuckAt1 [1]	Disabled By User
Disable	467349	InputPortConnectionBitNegated [1]	Non-Propagated
Enable	467350	InputPortConnectionBitStuckAt0 [0]	Disabled By User
Enable	467351	InputPortConnectionBitStuckAt1 [0]	Disabled By User
Disable	467352	InputPortConnectionBitNegated [0]	Non-Propagated

```
    ,    .I_hdlc1_Ar_User( I_hdlc1_Ar_User )
```

Figure 4.4 Certitude Fault Detail report

We expected to see Detected status for bits 7 through 0. Therefore, the Non-Propagated status is non-expected, according to our predicted entries in the tracking spreadsheet. The diagram in **Error! Reference source not found.** shows our expectation for these bits, and the actual Certitude results from the spreadsheet. The "Signal Name" column contains the design's signal names of these bits. The "Description" column contains references to our specifications document.



Signal Name	Description	AxUSER AXI3	AxUSER AXI4	AxUSER ACE-Lite	CE expect	CE actual	CE Match
SRCID[0]	Source ID. See Section 5.5, Source ID Assignments	0	0	0	D	NP	no match
SRCID[1]	Source ID. See Section 5.5, Source ID Assignments	1	1	1	D	NP	no match
SRCID[2]	Source ID. See Section 5.5, Source ID Assignments	2	2	2	D	NP	no match
SRCID[3]	Source ID. See Section 5.5, Source ID Assignments	3	3	3	D	NP	no match
SRCID[4]	Source ID. See Section 5.5, Source ID Assignments	4	4	4	D	NP	no match
SRCID[5]	Source ID. See Section 5.5, Source ID Assignments	5	5	5	D	NP	no match
SRCID[6]	Source ID. See Section 5.5, Source ID Assignments	6	6	6	D	NP	no match
SRCID[7]	Source ID. See Section 5.5, Source ID Assignments	7	7	7	D	NP	no match

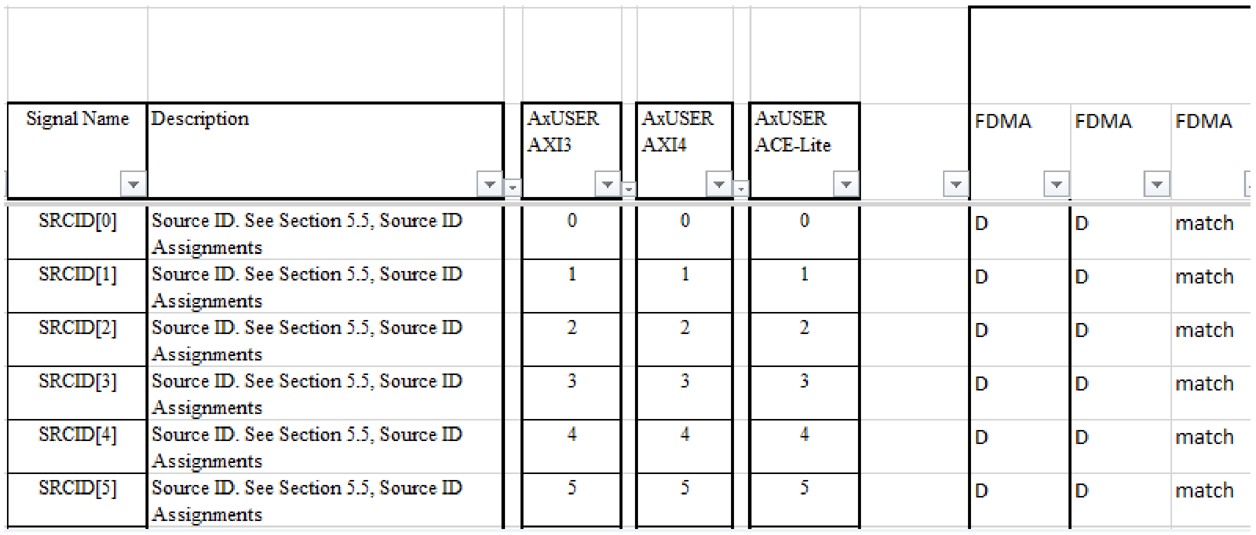
Figure 4.5 Details for Ar_User bits 7 through 0 expect versus actual fault status result

Analysis: This was a clear case of missing stimulus. Although the expectation was that the faults would get detected, there was no testcase for this particular initiator. And since, these were tests which were supposed to check the propagation of the tied-off values to a particular target interface, and since there was no testcase, we were not able to catch the tie-off from toggle coverage as well.

Solution: We need to develop at least one testcase to include this particular initiator interface. Currently the testcase is missing in our regressions.

Conclusion: Adding test cases in response to resolving the Non-Detected and Non-Propagated faults make our Verification Environment measurably better to catch actual bugs in the design, such as incorrectly tied signals.

It is noteworthy that the actual Certitude faults status of the majority of the various AXI_USER signals met our expectations. Some examples of these signals are shown in the tracking spreadsheet in Figure 4.6.



Signal Name	Description	AxUSER AXI3	AxUSER AXI4	AxUSER ACE-Lite		FDMA	FDMA	FDMA
SRCID[0]	Source ID. See Section 5.5, Source ID Assignments	0	0	0		D	D	match
SRCID[1]	Source ID. See Section 5.5, Source ID Assignments	1	1	1		D	D	match
SRCID[2]	Source ID. See Section 5.5, Source ID Assignments	2	2	2		D	D	match
SRCID[3]	Source ID. See Section 5.5, Source ID Assignments	3	3	3		D	D	match
SRCID[4]	Source ID. See Section 5.5, Source ID Assignments	4	4	4		D	D	match
SRCID[5]	Source ID. See Section 5.5, Source ID Assignments	5	5	5		D	D	match

Figure 4.6 Actual results match with expected results for the majority of bits evaluated with Certitude

5. Limitations/Enhancements needed

Our overall goal is to use Certitude in a completely seamless manner along with VCS and other Synopsys tools in our flow. Here is a list of generic and project specific requirements in order to meet this goal.

<u>5.1 Generic items:</u>

1. Certitude now has newly introduced the Native VCS integration which unifies the compilation flow (single compile). This Native VCS integration will make our compile flow much easier. However, currently the Certitude flow using VCS Native mode does not support the Partition Compile feature in VCS. We normally use the Partition Compile flow since it speeds up the compilation process. We only make changes to the TestBench in our compilations, and so the TestBench partition only needs to be re-compiled, and not the RTL partition. We would like Certitude VCS Native mode to support the Partition Compile flow so that we can use one compile flow which will simplify our process. In the meantime, we are using Certitude Stand-alone mode with our Partition Compile flow.

2. The certitude_testcases.cer needs manual edits to replace the spaces with colons.

3. With Certitude Stand-alone mode we have to individually add all the module files from ip/rtl_v/ folder. There is no –folder option for certitude_hdl_files.cer. This will be resolved when we move to Certitude using VCS Native mode, since VCS takes care of the fault instrumentation.

5.2 Project specific items:

1. We would like Certitude to directly access the Unreachable signals from the URG report by providing the path to the coverage DB. Without this capability we are having to manually list out all the Unreachable signals from our design, then pass this list to certitude commands to correspond to the appropriate fault numbers modelled by Certitude. This is a manual effort that we wouldlike to avoid.

2. We would like to directly input our "Fault Expectation" to every single fault that we enable before running the activation and detection phase of Certitude. This not only gives an accurate metric for the overall number of detect/non-detect faults, but also an accurate metric to understand if all the faults met the expectations of detect/non-detect or not. As a work-around, we are using the "fault_expectation.tcl" as mentioned in section 3.

3. We would like the Certitude Report to directly provide the spreadsheet for the expected/non-expected results. As a work-around, we are using the "fault_status.tcl" to export a CSV file, as mentioned in section 3.

6. Conclusions

Certitude's ability to fault individual signal bits on chosen Unreachable signals was important to measure the quality of our Verification Environment Checkers. From our analysis of the Certitude faults status, we had a very good idea on what to fix on the Checkers. Then when these fixes were implemented, the fixed Checkers were measurably better able to find actual buggy tie-off signals in the design.

We plan to continue to use Certitude on key IP blocks of this project and for future projects in order to measure and improve the quality of our Verification Environment.