SYNOPSYS®

Formality Workshop

LabGuide

2018.06-SP2

Synopsys Customer Education Services
700 East Middlefield Road
Mountain View, California94043

Workshop Registration: **1-800-793-3448**

www.synopsys.com

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility todetermine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Cadabra, CATS, CRITIC, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSIM, HSPICE, iN-Phase, in-Sync, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PrimeTime, SiVL, SNUG, SolvNet, System Compiler, TetraMAX, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Columbia,Columbia-CE, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, Direct Silicon Access, Discovery, Encore, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, HSIMplus, HSPICE-Link, iN-Tandem, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport,Library Compiler, Magellan, Mars, Mars-Rail, Milkyway, ModelSource, Module Compiler, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Raphael-NES,Saturn, Scirocco, Scirocco-i, StarRC, Star-SimXT, Taurus, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (℠)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license. ARM and AMBA are registered trademarks of ARM Limited. Saber is a registered trademark of SabreMark Limited Partnership and is used under license. All other product or company names may be trademarks of their respective owners.

1 Running Formality from SVF

Learning Objectives

This lab will introduce to running Formality.

After completing this lab, you should be able to:

- Set up a simple Design Compiler script to write out SVF
- Generate a Formality script from SVF using **fm_mk_script**
- Run formality on the **fm_mk_script** generated script

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

Use the SVF from a synthesis run to generate a Formality script and then do RTL to gates verification

Task 1. Running synthesis

Run the synthesis

1. Go into the lab1 directory

```
unix% cd lab1
```

2. Inspect the Design Compiler synthesis script run1_dc.tcl

Question 1. Where is the SVF written to ?

.....

3. Run the Design Compiler script :

```
dc_shell –f  run1_dc.tcl | & tee –i run1_dc.log
```

Task 2. Using the fm_mk_script

One can use the utility **fm_mk_script** to generate the Formality script.

1. Run fm_mk_script

```
fm_mk_script  -output  my_run_fm.tcl my_run/mR4000.svf
```

2. Run the Formality script :

```
fm_shell -f my_run_fm.tcl | & tee -i  my_run_fm.log
```

Search down in the transcript my_run_fm.log (or on the terminal) for the line

```
|***************************** Verification Results *****************************
```

Question 2. Has the verification :

a) Succeeded (Passed)

b) Failed

c) Been Inconclusive ?

.....

3. The script my_ run_fm.tcl doesn't quit out of fm_shell. Exit from the fm_shell with the command **exit**

Congratulations you have just successfully completed your first Formality run on this course!

Answers / Solutions

Task 2. Using the fm_mk_script

Question 1. Where is the SVF written to ?

The file my_run/mR4000.svf .

This comes from the line set_svf my_run/mR4000.svf

Question 2. Has the verification :

a) Succeeded (Passed)

b) Failed

c) Been Inconclusive ?

Answer: a)

```
********************************* Verification Results *********************************
Verification SUCCEEDED
```

This page is left blank intentionally.

2 Steps and GUI

Learning Objectives

This lab will introduce you to the Formality GUI and the steps to verify a design

After completing this lab, you should be able to:

- Invoke Formality GUI
- Set guidance
- Read in RTL design
- Read in gate level netlist and technology libraries
- Match and verify the design
- Generate a TCL script to do the steps
- Save a session file

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

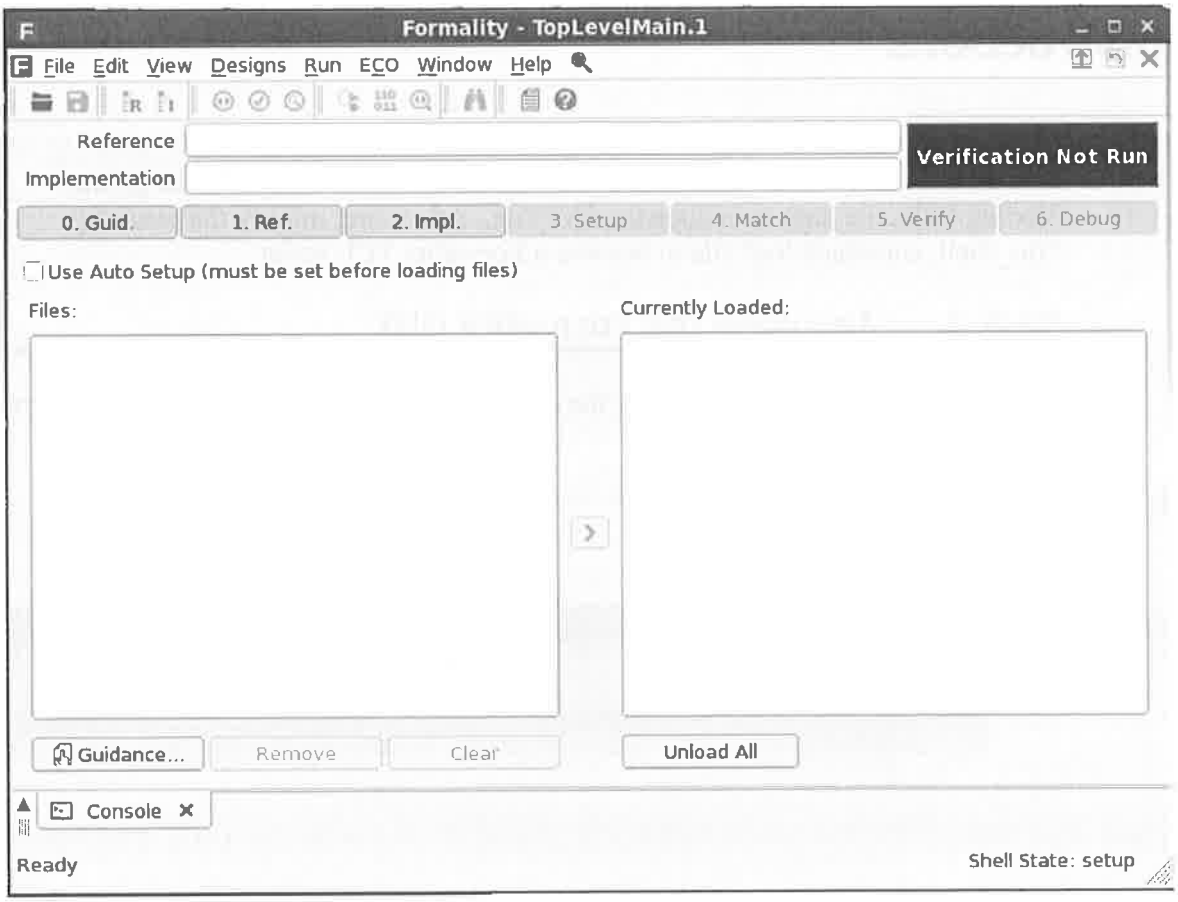
Use the Formality GUI to verify the gate-level Verilog netlist against the golden Verilog RTL. Be sure to include the SVF file. Afterward, modify the resulting "fm_shell_command.log" file to become a Formality TCL script

Task 1. Invoking the Formality GUI

Invoke the Formality GUI by using the command "formality" or "fm_shell –gui" in lab2

1. Invoke Formality

```
unix% cd lab2
unix% fm_shell -gui
```



Question 1. What is the Formality mode or shell state on invocation ?

.....

Question 2. From the presentation or Job AID what are the steps to run Formality ?

.....

Question 3. Does your answer to Question 2 match the order of the tabs in the GUI ?

.....

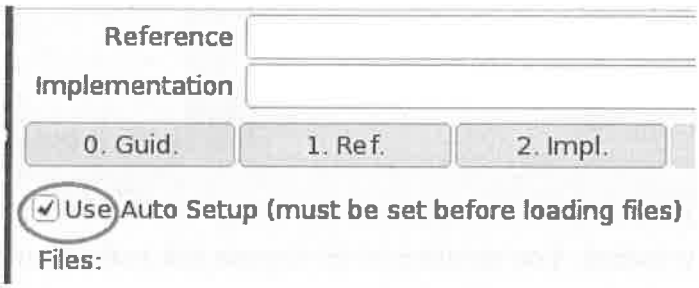
Question 4. Which of these steps may be optional and why ?

.....

Task 2. Setting the guidance

The verification in this lab will do RTL to gates verification. So two things you will want to do is set the auto setup variable and the SVF.

2. Set the Auto Setup in the GUI

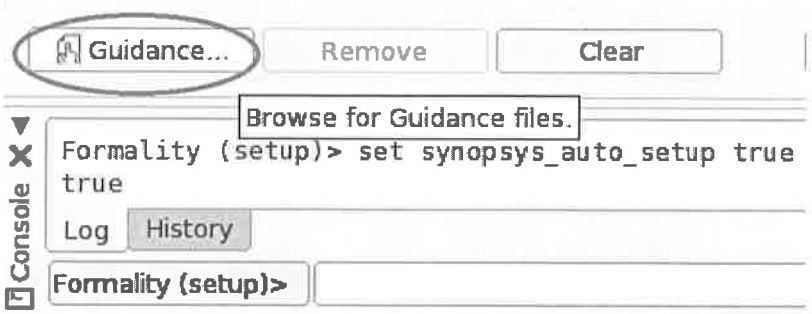


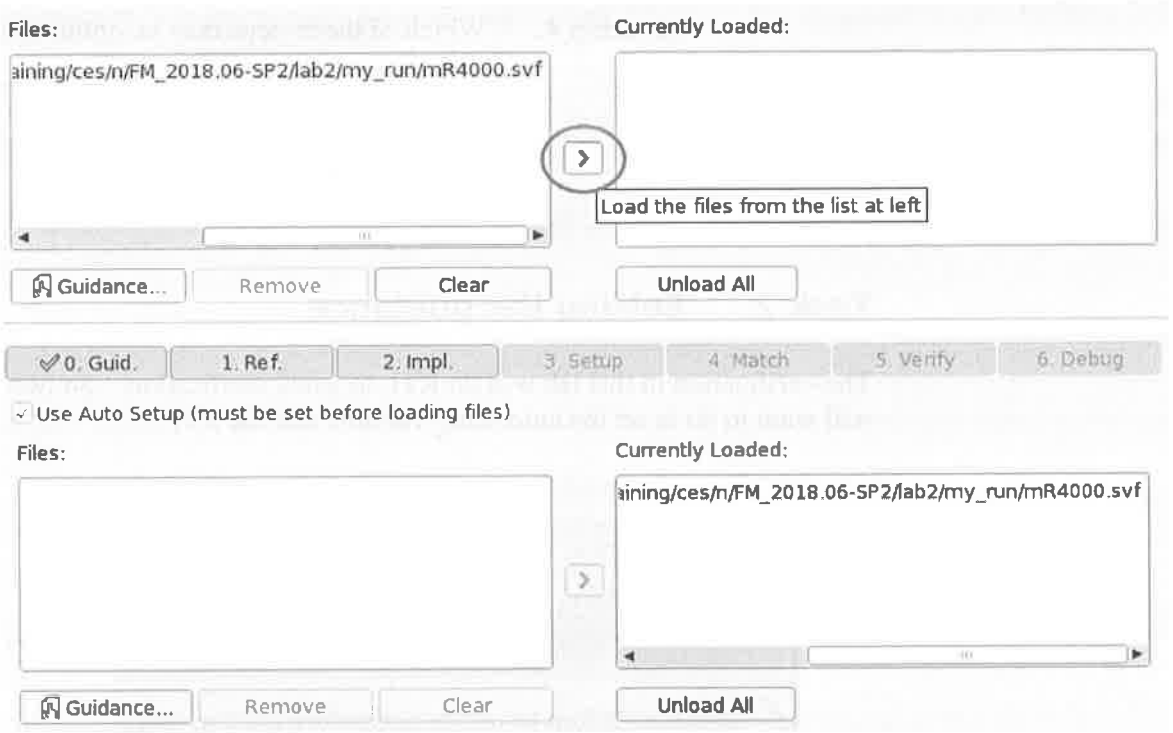
Question 5. What variable is changed by using Auto Setup ?

.....

3. Set the guidance. The SVF is located at ./my_run/mR4000.svf

Use the browser to guidance to set the SVF file





The SVF is now loaded. You should now see a green tick in the Guidance tab.

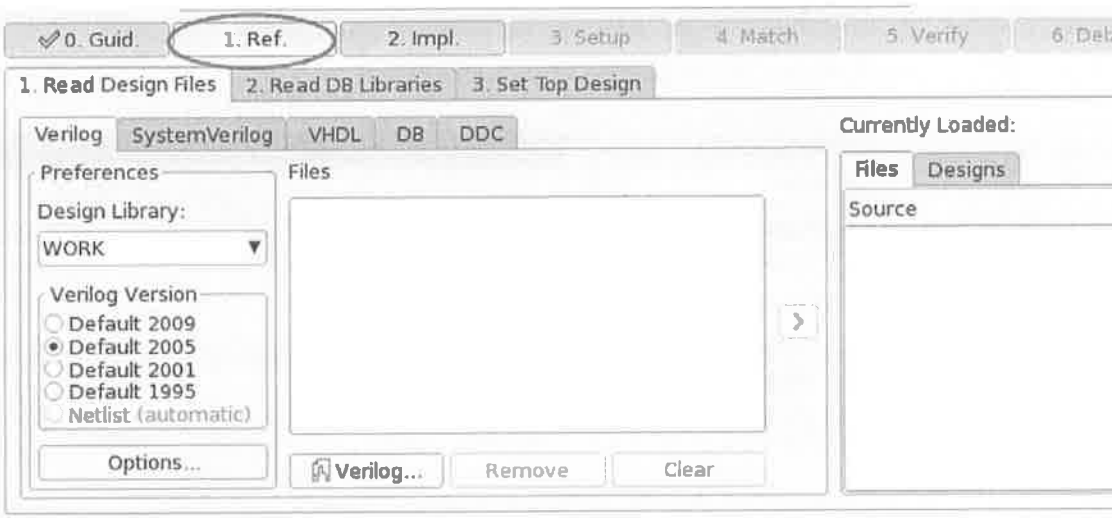
4. (Optional) As soon as you set the guidance the SVF will be turned into an ASCII TCL command file. You can see and read this file at :

   ./formality_svf/svf.txt

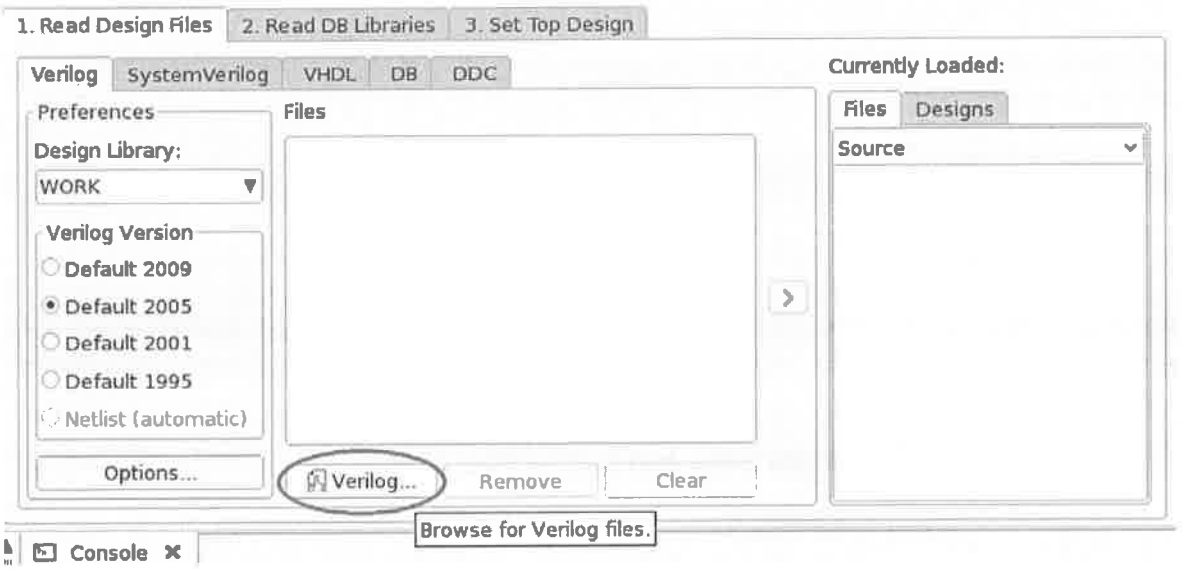
Task 3. Read in the RTL design

The Verilog RTL files are located under the directory "*rtl*". The top-level design name is "*mR4000*".

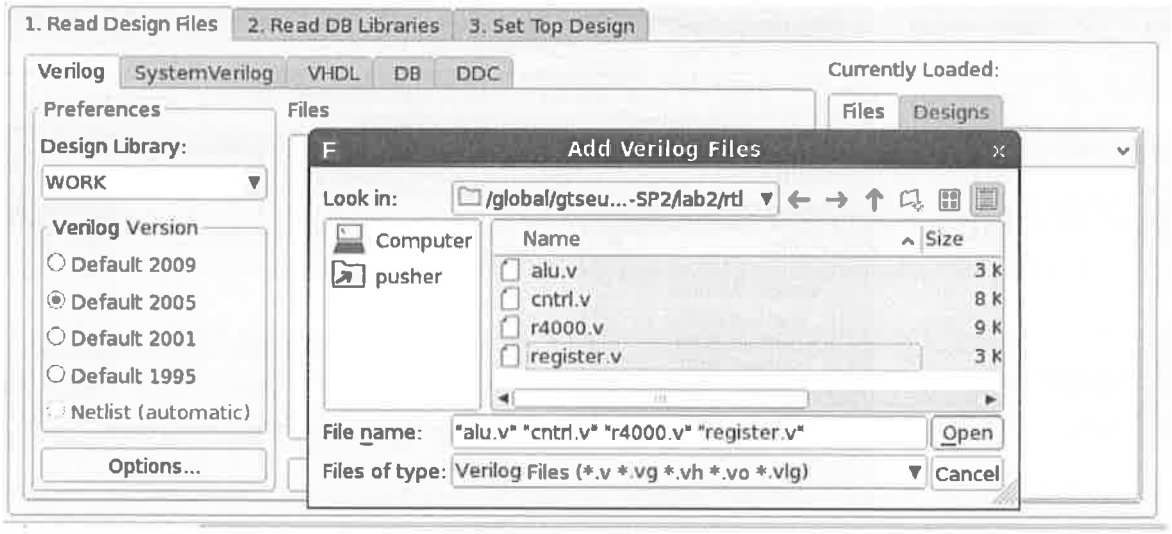
1. Click on the Reference tab to read in the RTL



Click on the browse Verilog tab

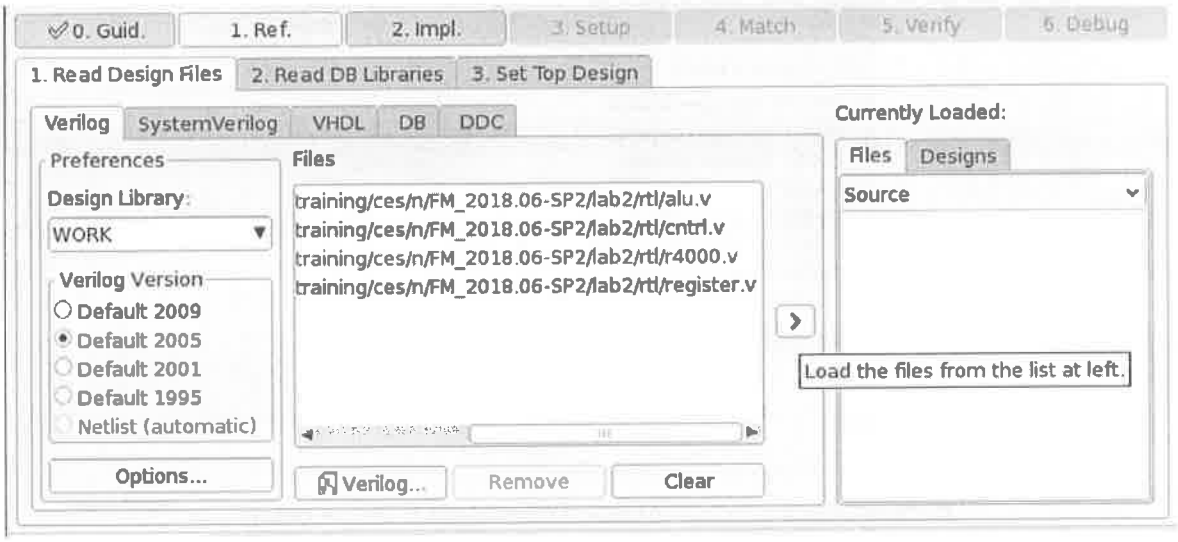


And select all the Verilog files in the ./rtl directory

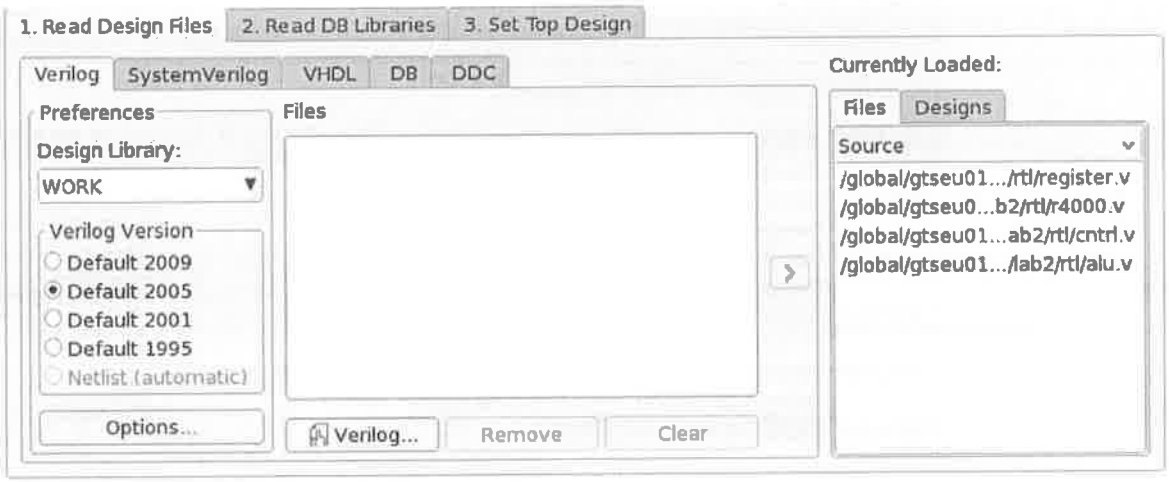


Click on Open in the Add Verilog Files screen.

Then Load Files



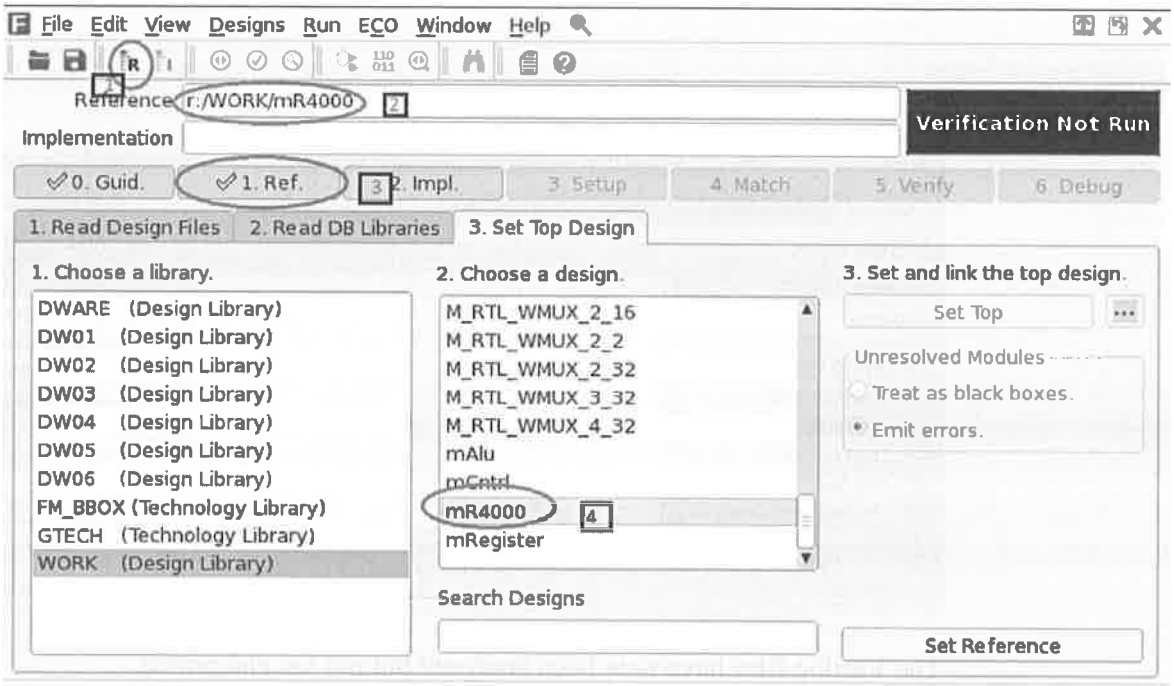
So it should now look like below:



The Verilog files have now been analyzed but not yet elaborated.

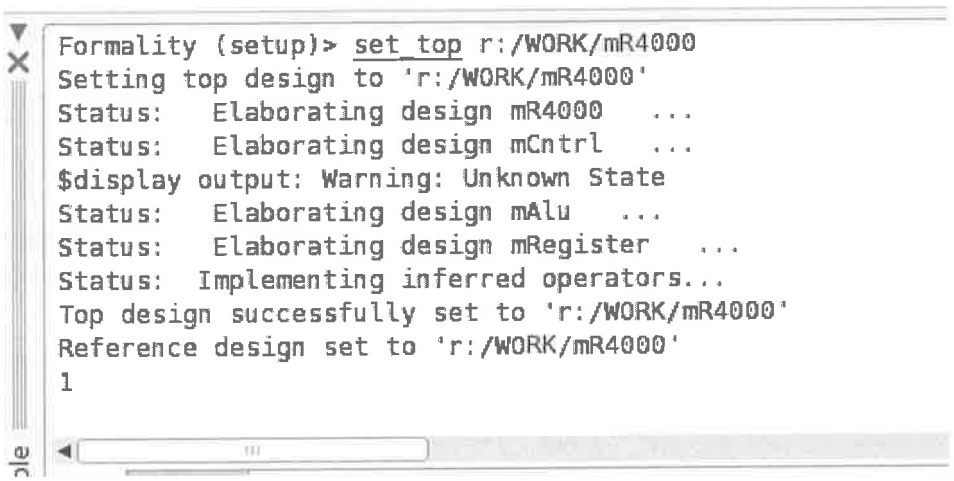
2. To elaborate the RTL click on the '3. Set Top Design'; Select under '2. Choose a Design' mR4000 and then click o the Set Top button

If things have gone as intended then should see as below



Notice the Reference now has a green tick against it (3); what the reference design is currently set to (2) and the reference container is now available in the hierarchy browser (1). (4) shows what the top design was set to.

If you expand you will see this also reflected in the transcript:



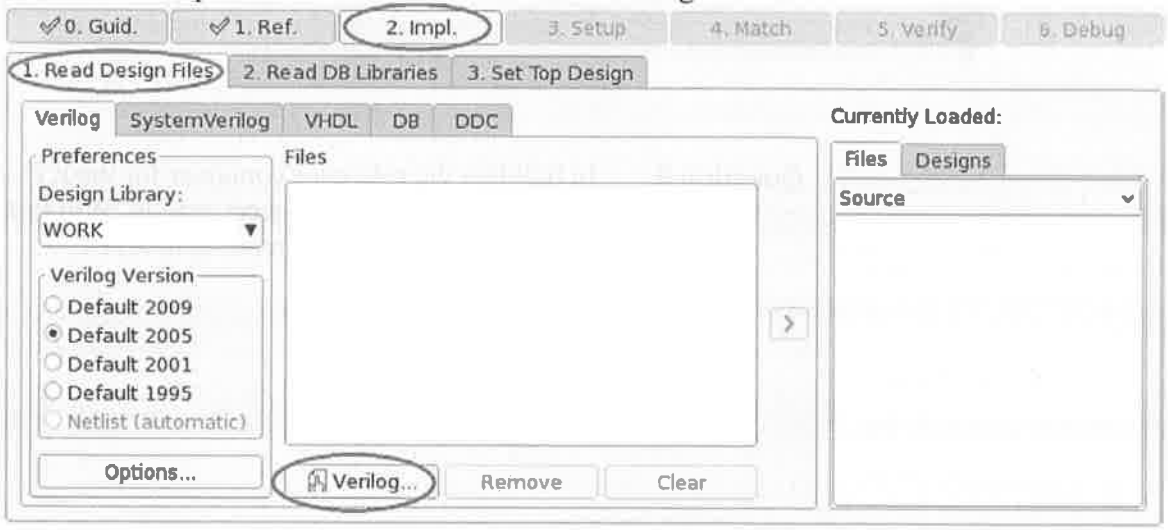
Question 6. In building the reference container for the RTL no user technology libraries were read in. Will this always be the case when reading in RTL?

.....

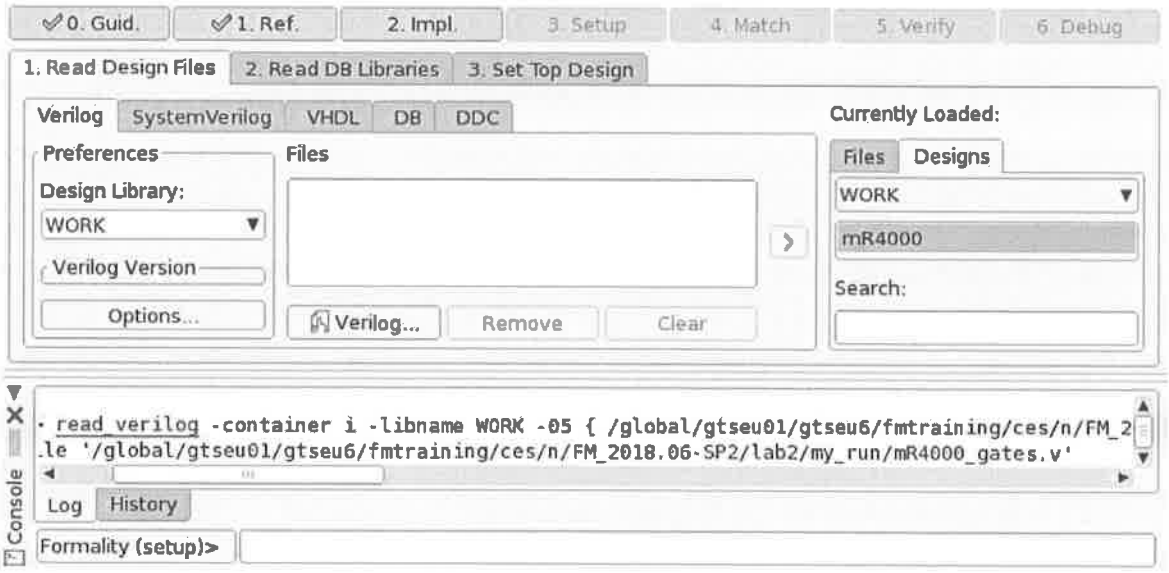
Task 4. Read in the netlist and technology libraries

The Verilog gate-level netlist "*mR4000_gates.v*" is located under the directory "*my_run*". The top-level design name is "*mR4000*". The technology library is `tc6a_cbacore.db`located under 'lib'.

1. Click on the Implementation tab and browse for verilog files

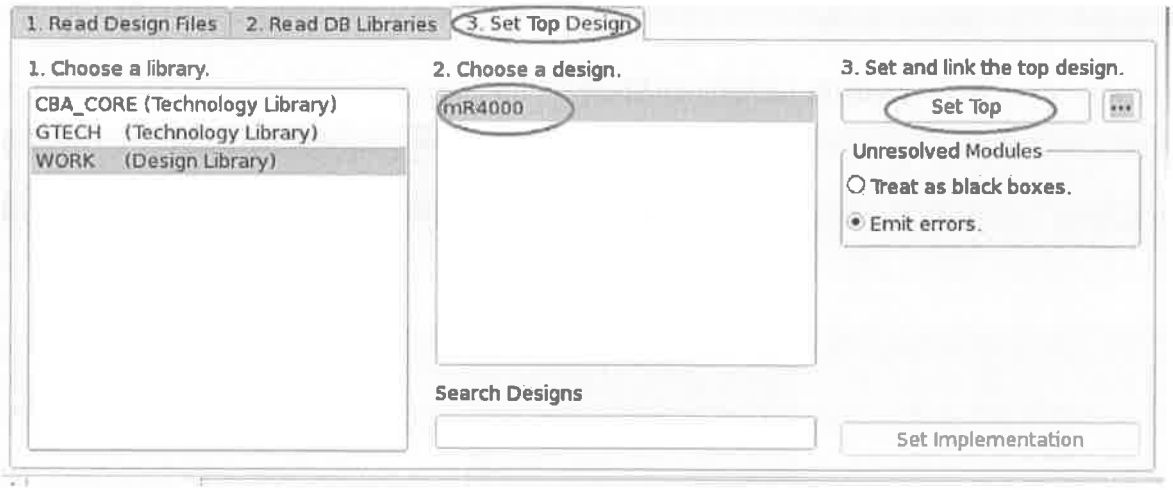


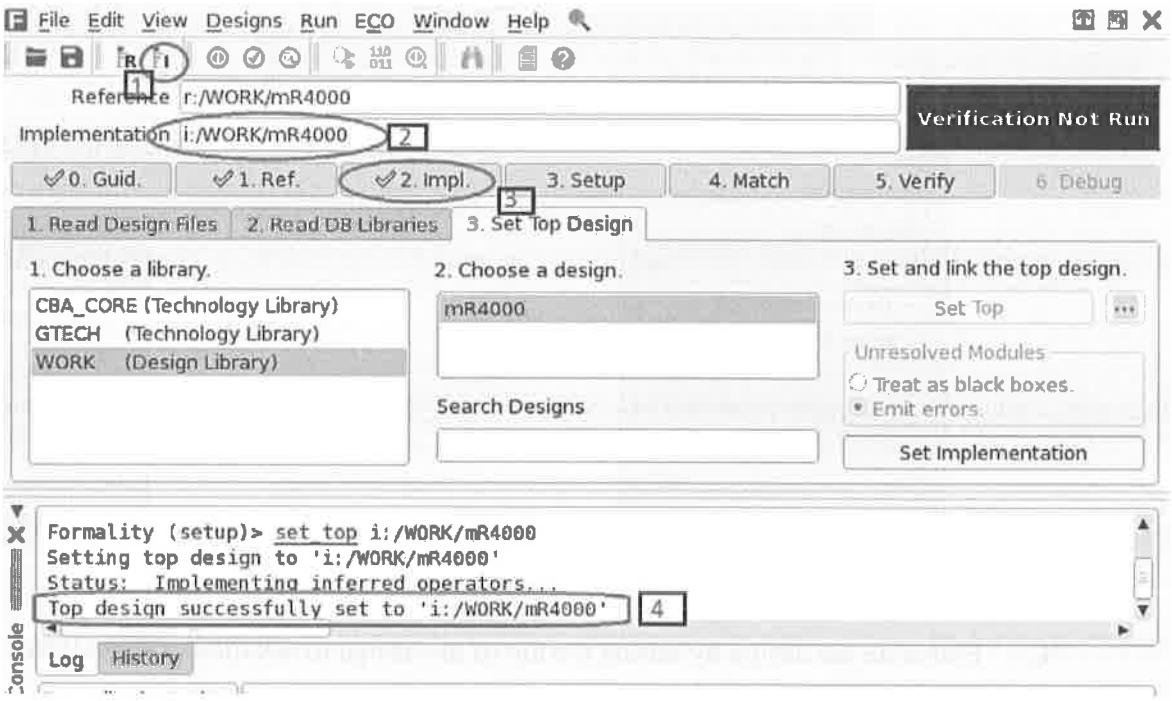
2. Load the Verilog netlist at *my_run/mR4000_gates.v*



3. Load the technology library at lib/`tc6a_cbacore.db`

4. Elaborate the design by setting the top of the design to *mR4000.* (There is in fact only one design in this case)





If things have gone as intended you should now see that the implementation has been successfully set (for example the implementation tab at 3 has a green tick)

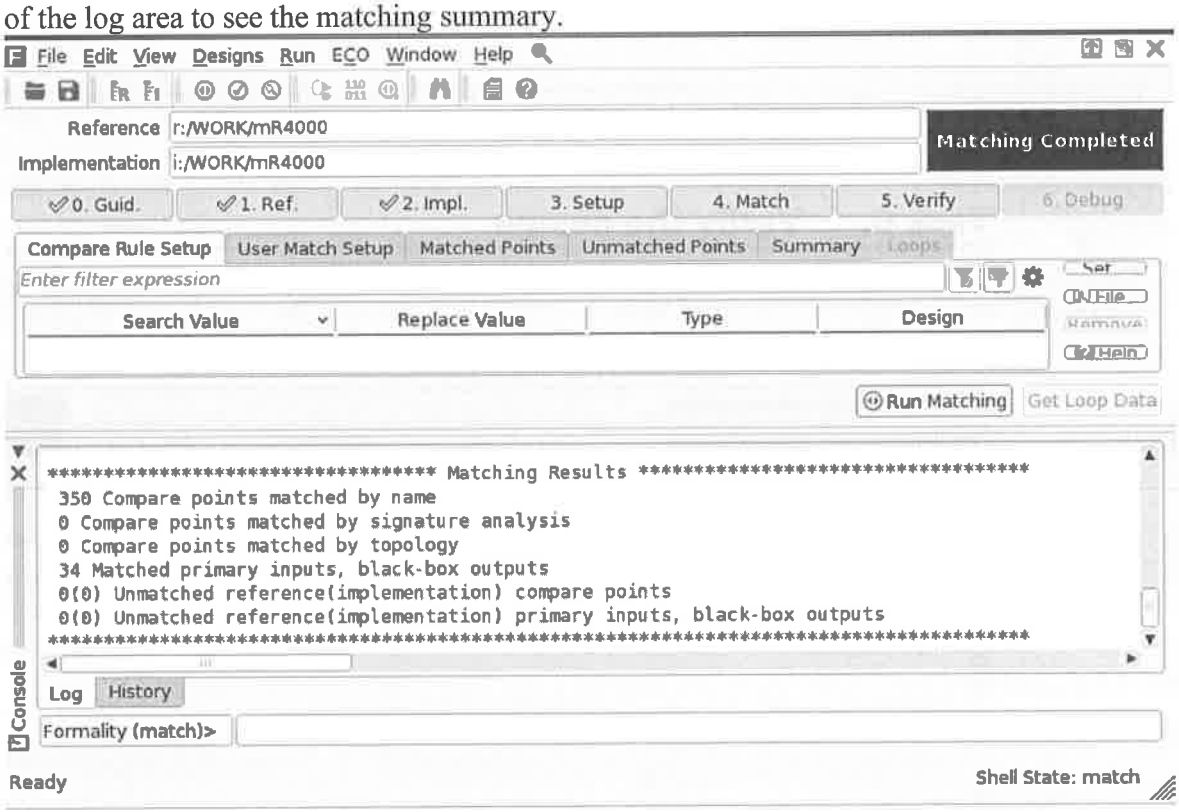
Task 5. Match the design

For this design, as is quite often the case when you are using auto setup and SVF, no additional setup is required. So we can go straight to matching.

1. Click on the matching tab and the run matching button.

Click on OK when prompted.

2. Examine the matching results summary. You may need to extend increase the height of the log area to see the matching summary.



3.

Question 7. How many compare points have matched ?

.....

Question 8. How many compare points are **un**matched ?

.....

Question 9. How do you know matching has run?

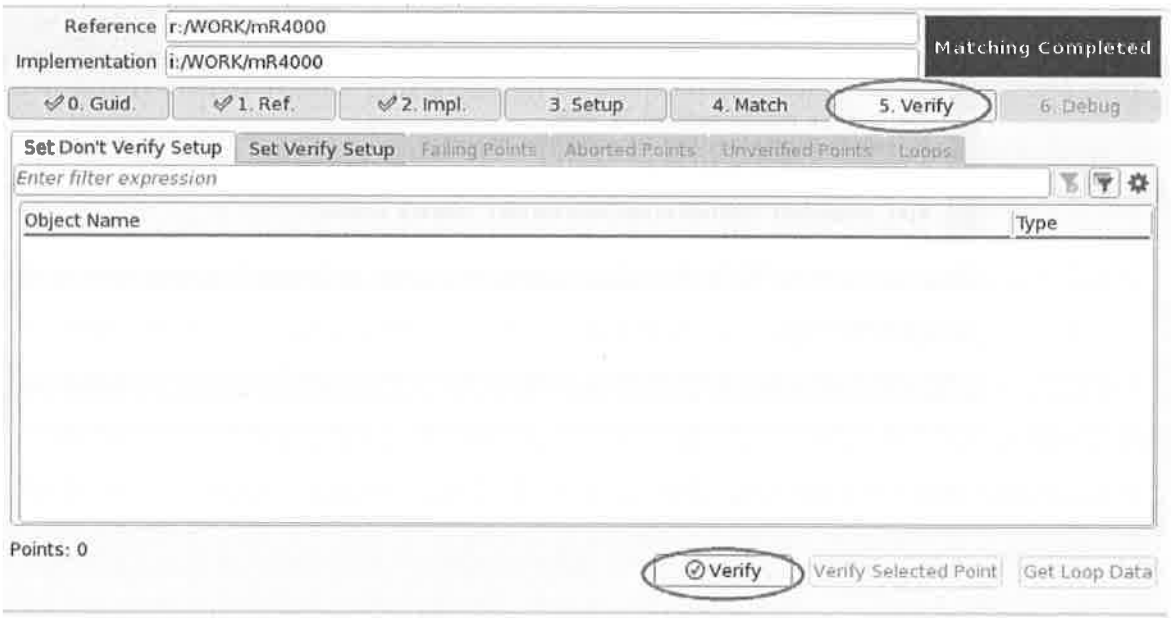
.....

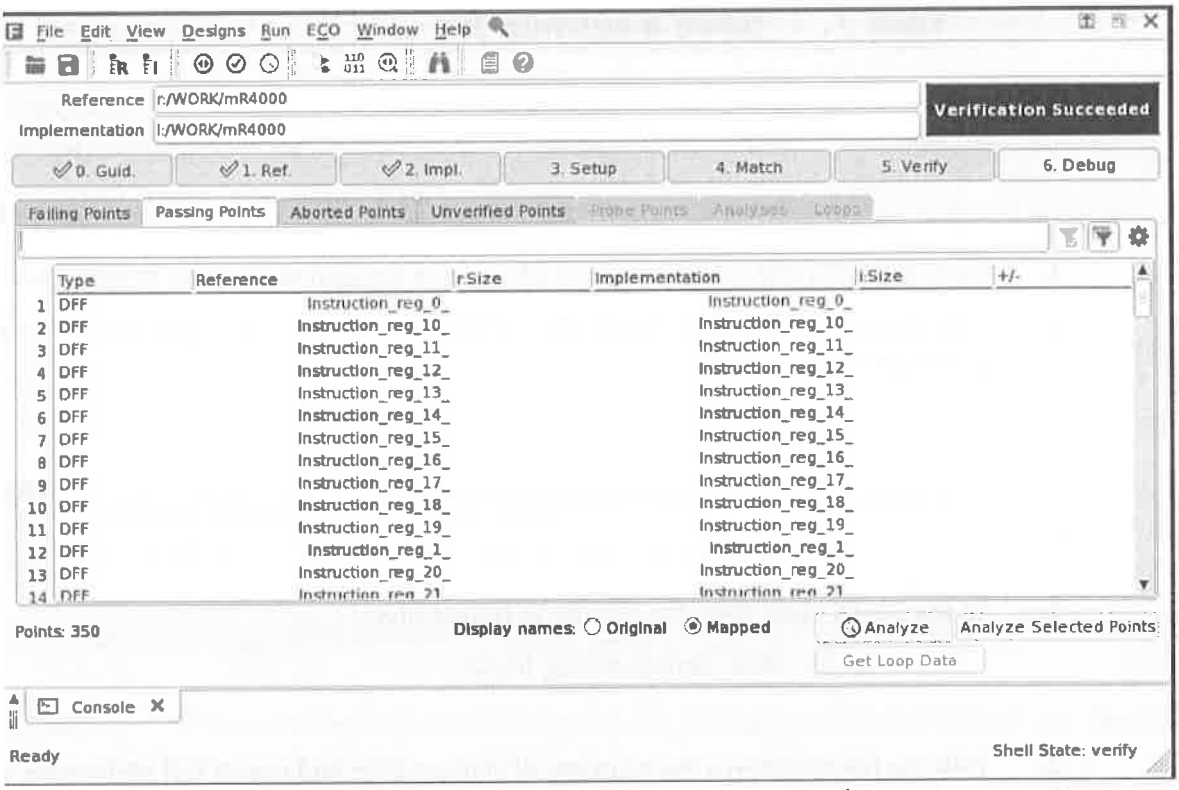
Question 10. Based on the matching is it sensible to continue to the verify step and if so why?

.....

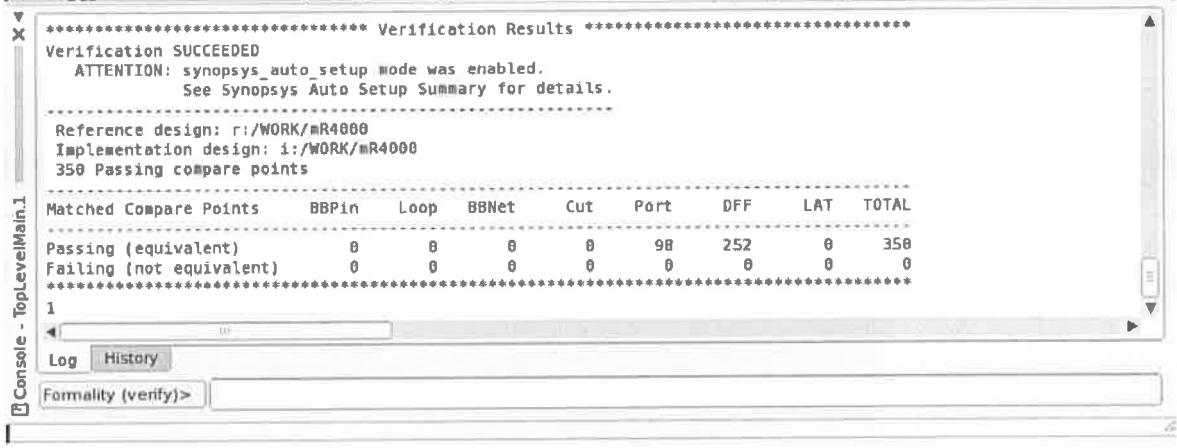
Task 6. Verify the design

1. Click on the verify tab and the run verify button.





And expanding the console :



Question 11. Has the verification succeeded ? Where are two places where you can see this ?

.....

Question 12. How many passing points are there and is this the same as the number of matched compare points ?

.....

Task 7. Save a session file

It is always good practice in Formality to save a session file which can be restored later.

1. Save a session file using the GUI File -> Save Session menu item to post_verify.fss
2. Exit out of the Formality using File -> Exit. Answer yes when prompted if you are sure you want to exit.

Task 8. Re run the verification from scratch using TCL

The commands from the GUI verification have been recorded in the file fm_shell_command.log

1. In the UNIX shell copy the log file to runme.fms

   cp fm_shell_command.log fm.tcl

2. Edit the file to remove the sourcing of start up files and use of full pathnames to make the verification more portable

```
set synopsys_auto_setup true

set_svf -append {./my_run/mR4000.svf }
read_verilog -container r -libname WORK -05
{ ./rtl/alu.v ./rtl/cntrl.v ./rtl/r4000.v ./rtl/regi
ster.v }
set_top r:/WORK/mR4000

read_verilog -container i -libname WORK -05
{ ./my_run/mR4000_gates.v }
read_db { ./lib/tc6a_cbacore.db }
set_top i:/WORK/mR4000

match

verify

save_session -replace ./post_verify.fss
exit
```

3. Run the Formality script :

```
fm_shell -f fm.tcl | & tee -i fm.log
```

The verification should succeed.

Answers / Solutions

Task 1. Invoking the Formality GUI

Question 1. What is the Formality mode or shell state on invocation?

setup

Question 2. From the presentation or Job AID what are the steps to run Formality?

Guide ; read in reference ; read in implementation; setup ; match; verify; debug

Question 3. Does your answer to Question 2 match the order of the tabs in the GUI ?

For the correct answer: yes.

Question 4. Which of these steps may be optional and why ?

Guide : Setting the SVF is strictly speaking always optional. However when you are doing RTL to gates where the synthesis has been done by DesignCompiler it is almost always used.

Setup : The setup is optional in that you may not require any manual setup .

Match : Is optional. If the match command has not been run the verify command will run match before doing the verification.

Debug :Optional : If the verification passes then there is nothing to debug.

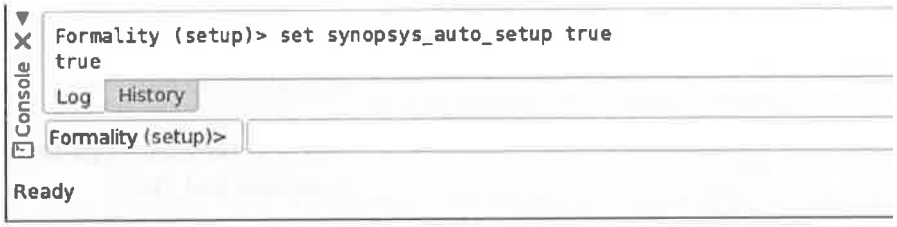
Task 9. Setting the guidance

Question 5. What variable is changed by using Auto Setup ?

synopsys_auto_setup

You can see it echoed out in the GUI log file or unix shell.

To see it in the GUI log file you may need to expand the console:



The unix shell is obvious:

Question 6. Inbuilding the reference container for the RTL no technology libraries were read in. Will this always be the case when reading in RTL?

No. For example you might have RAMs or PLLs or IO cells instantiated in the RTL.

Task 5. Match the design

Question 7. How many compare points have matched ?

350

(Note this number doesn't include the matched 34 input ports . Getting ahead of ourselves :

report_matched_points -type input

will show the 34 matched input ports)

Question 8. How many compare points are unmatched ?

0

Question 9. How do you know matching has run ?

The shell mode has changed to match

Question 10. Based on the matching is it sensible to continue to the verify step and if so why?

Yes it is sensible to go to verify step. The matching looks complete and clean.

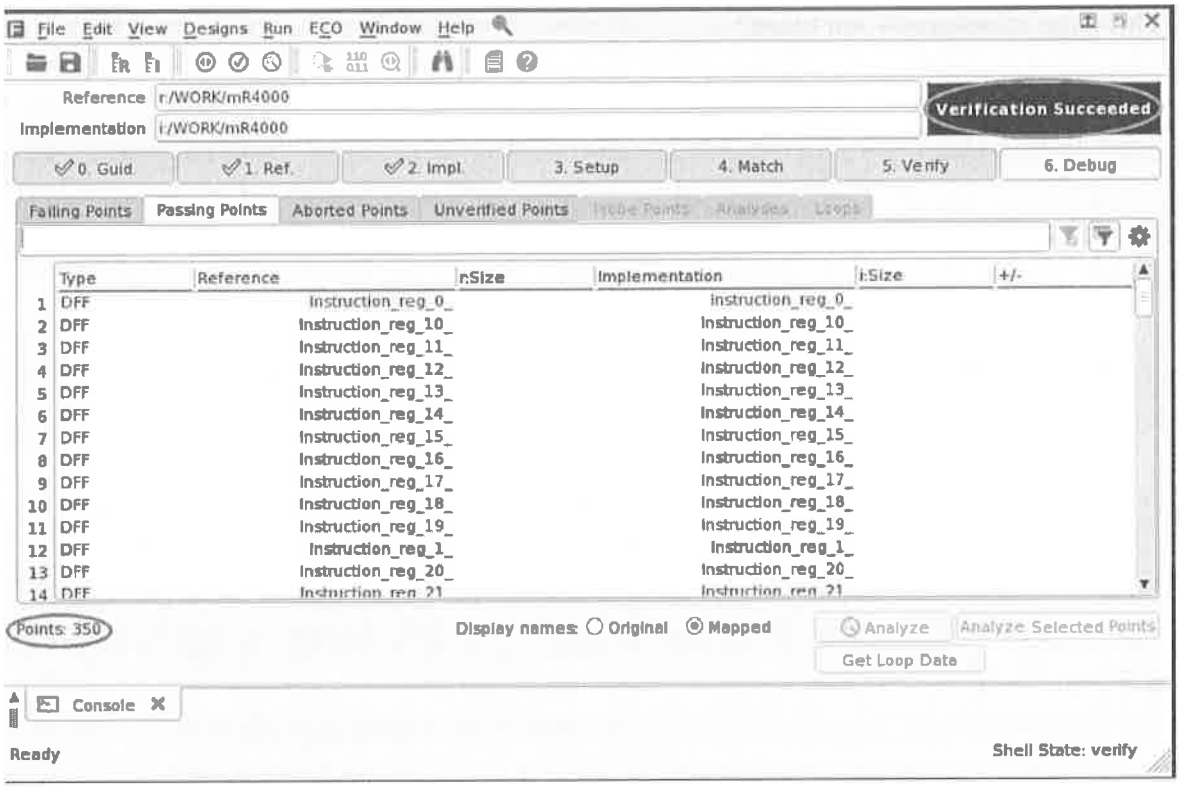
Task 6. Verify the design

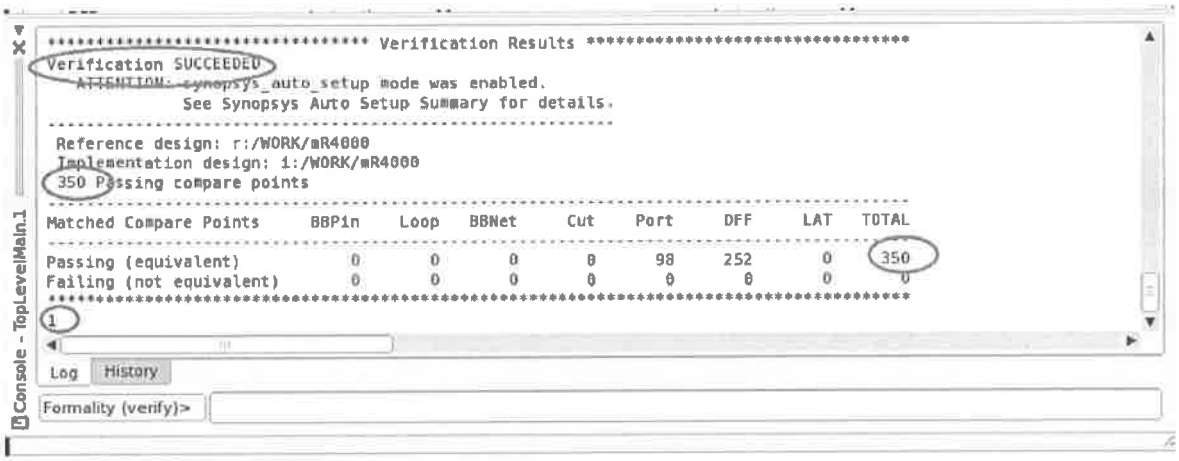
Question 11. Has the verification succeeded ? Where are two places where you can see this?

Yes the verification has succeeded. You should be able to see this in the status area in the top right of the GUI and in the transcript.

Question 12. How many passing points are there and is this the same as the number of matched compare points?

350 passing points. Which is indeed the same as the number of matched compare points


File Edit View Designs Run ECO Window Help
Reference r:/WORK/mR4000
Implementation i:/WORK/mR4000
Verification Succeeded
0. Guid 1. Ref. 2. Impl. 3. Setup 4. Match 5. Verify 6. Debug
Failing Points Passing Points Aborted Points Unverified Points
Type Reference r:Size Implementation i:Size +/-
1 DFF Instruction_reg_0_ Instruction_reg_0_
2 DFF Instruction_reg_10_ Instruction_reg_10_
3 DFF Instruction_reg_11_ Instruction_reg_11_
4 DFF Instruction_reg_12_ Instruction_reg_12_
5 DFF Instruction_reg_13_ Instruction_reg_13_
6 DFF Instruction_reg_14_ Instruction_reg_14_
7 DFF Instruction_reg_15_ Instruction_reg_15_
8 DFF Instruction_reg_16_ Instruction_reg_16_
9 DFF Instruction_reg_17_ Instruction_reg_17_
10 DFF Instruction_reg_18_ Instruction_reg_18_
11 DFF Instruction_reg_19_ Instruction_reg_19_
12 DFF Instruction_reg_1_ Instruction_reg_1_
13 DFF Instruction_reg_20_ Instruction_reg_20_
14 DFF Instruction_reg_21 Instruction_reg_21
Points: 350
Display names: Original Mapped
Analyze Analyze Selected Points
Get Loop Data
Console
Ready
Shell State: verify


********** Verification Results **********
Verification SUCCEEDED
ATTENTION: synopsys_auto_setup mode was enabled.
See Synopsys Auto Setup Summary for details.
Reference design: r:/WORK/mR4000
Implementation design: i:/WORK/mR4000
350 Passing compare points
Matched Compare Points BBPin Loop BBNet Cut Port DFF LAT TOTAL
Passing (equivalent) 0 0 0 0 98 252 0 350
Failing (not equivalent) 0 0 0 0 0 0 0 0
1
Log History
Formality (verify)>
Console - TopLevelMain.1

This page intentionally left blank.

3 Simple logic cones and failing points

Learning Objectives

This lab will introduce you to some of matching and debugging features of Formality by looking at easy to understand examples

After completing this lab, you should be able to:

- Understand what information is contained in the pattern window
- Be able to use probe points

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

A verification script has been provided to run the Formality verification.

Task 1. Looking at expected failures on a simple example

1. Go to the lab3

```
unix% cd lab3
```

2. Inspect the RTL **fred_or.v** and **fred_and.v**and you will see that fred_or is just an **'or'** function and fred_and is just an **'and'** function. We will be comparing these 2 pieces of RTL in Formality.

Question 1. When one compares **fred_or** and **fred_and**,how many compare points would you expect and of what type ?

.....

Question 2. Apart from the compare points what else will match ?

.....

Question 3. For what input stimulus would a comparison of **fred_or** and **fred_and** give different results ?

.....

3. Inspect run1_fm.tcl. This is comparing **fred_or**and **fred_and**.
4. Run run1_fm.tcl

```
unix% fm_shell -f run1_fm.tcl |& tee -i run1_fm.log
```

5. Observe that the verification fails

Question 4. Find all the places in the transcript where you can tell that the verification has failed.

.....

Question 5. From the transcript and the command **report_matched_points**confirm your to Question 1 and 2.

.....

Question 6. Also use the **report_failing_points** and **report_passing_points** commands to confirm the details of the status of the compare points

.....

6. Use **report_matched_points** to confirm what Formality thinks is driving the inputs of the logic cone of the failing point.

7. fm_shell (verify) report_matched_points -type port ${ref}/z

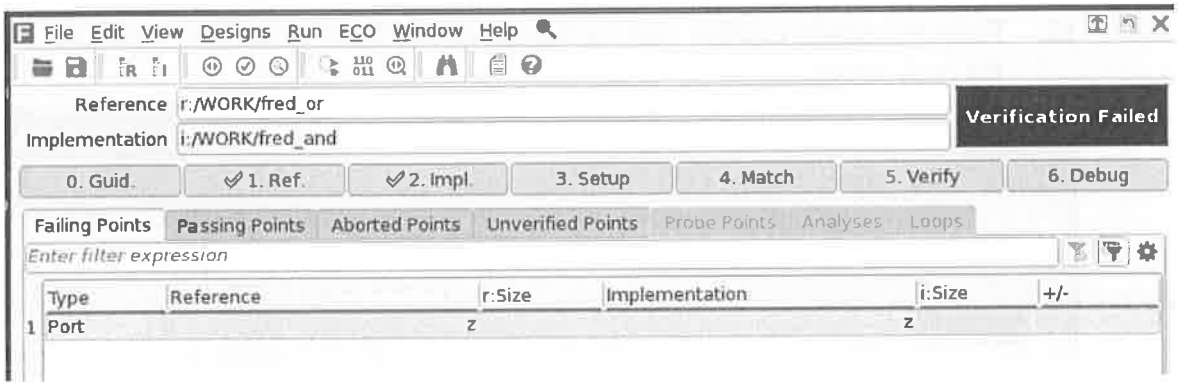
8. (Optional) One can use the **report_truth_table** command to confirm what Formality sees as the boolean function for the compare point.

```
****************************************************
Report          : truth_table
                  r:/WORK/fred_or/z

Reference       : r:/WORK/fred_or
Implementation  : i:/WORK/fred_and
Version         : O-2018.06-SP2
Date            : Sat Sep 29 20:07:35 2018
****************************************************


Truth table for signal : "z"
----------------------

a
| b
| |
0 0 | 0
0 1 | 1
1 . | 1

******* End truth table *******
```

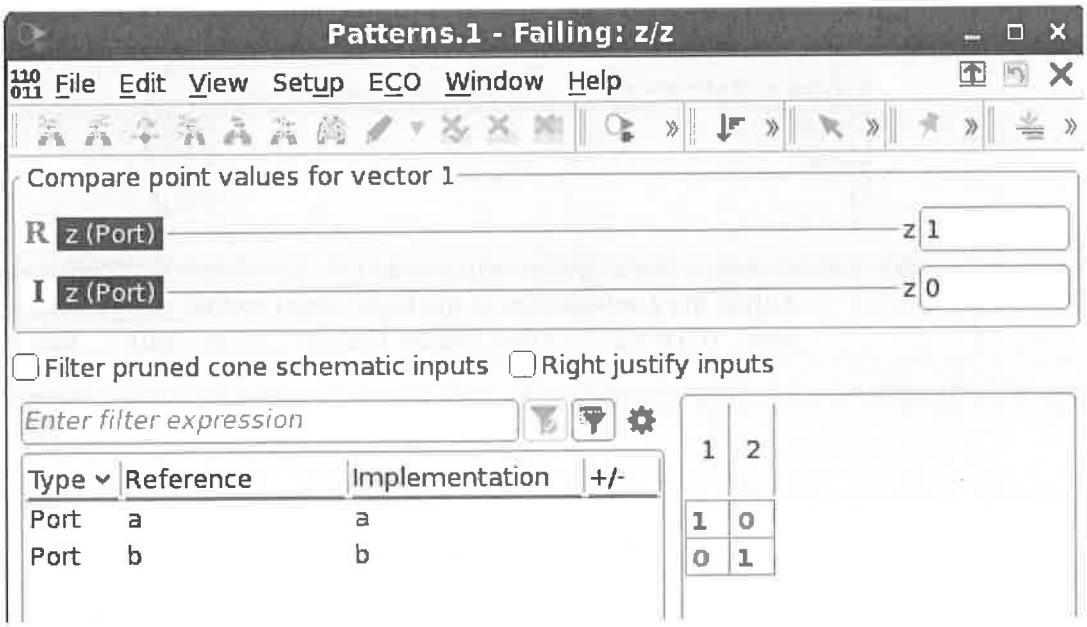
9. Start up the GUI

```
fm_shell (verify) start_gui
```

10. Pull up the pattern window for the failing point by under the debug tab by selecting the failing point and then clicking on the Show Patterns icon 



11. You should now see :



Question 7. What are the input values for which the verification fails ?

.....

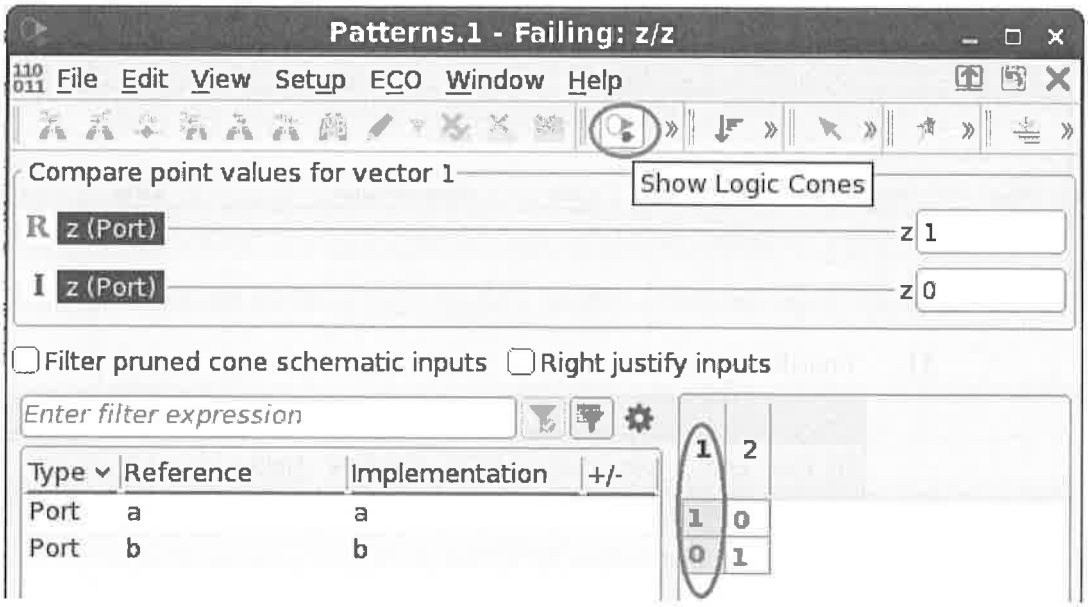
Question 8. What are the output values in the reference and implementation ?

.....

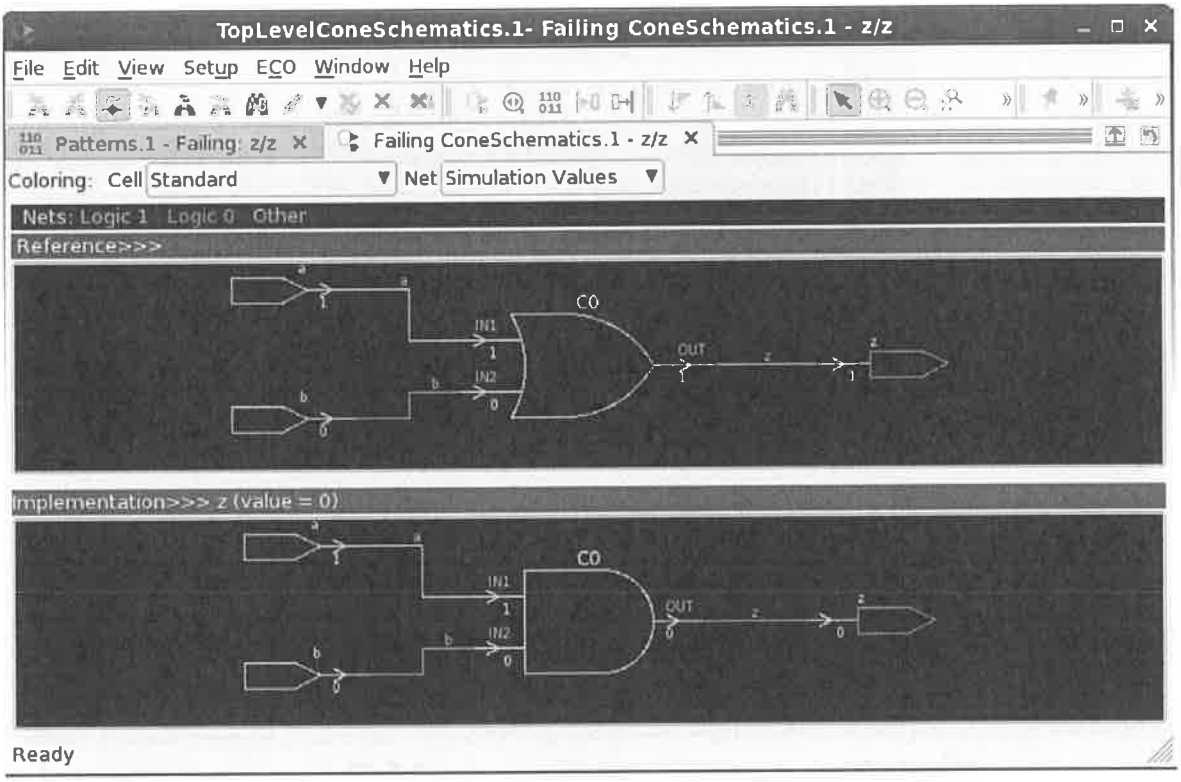
Question 9. Are these the results you would expect given your answer to Question 3 ?

.....

12. From the pattern window with vector one selected pull up the schematic for the logic cones



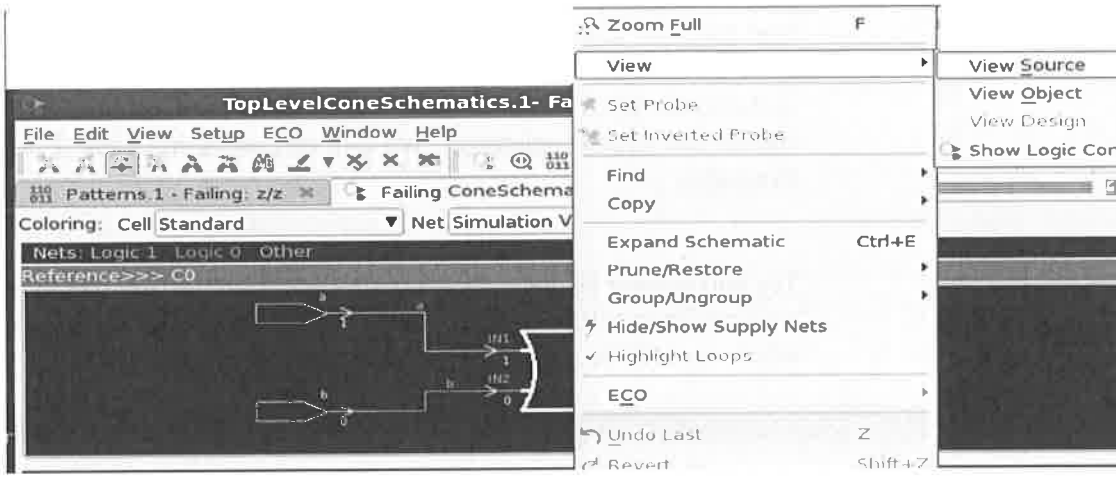
Adjust the 2 schematics of the logic cones so that you can see all the logic cone. (One way is Right Mouse Button : Zoom Full) . You should now see



Note that the RTL of the logic cone is represented as a gate level netlist and the failing vector values are overlayed on the logic cones. (If you click on vector 2 in the pattern window, see that the values overlayed in the schematic changes. Though here it is only the inputs that change as for both vectors the output of **or** gate is 1 and **and** gate 0)

13. (Optional) Cross probe to the RTL

Select the OR gate in the Ref schematic and RMB View -> View Source



This take you to where the OR gate came from in the fred_or.v file :

fred_or.v

```
1  module fred_or (a, b , z);
2  input a;
3  input b;
4  output z;
5
6  assign z = a | b;
7
8  endmodule
9
```

14. Quit out of Formality.

Task 2. Looking at expected failures on a second simple example

We are now going to repeat the steps of Task 1 on a slighly more complicated example. A verification script has been provided to run the Formality verification.

1. Inspect **harry_3or.v** and **harry_2or.v** you will see that **harry_3or** is a3 input **'or'** function with the inputs and output registered. (That is if the RTL was synthesized one would expect 4 flip-flops in the netlist – one for each input and one on the output.). Similarly **harry_2or** is a 2 input **'or'** function with inputs and output registered. We will be comparing these 2 pieces of RTL in Formality.

Try and answer as many of the below questions as you can in advance of running Formality. If you do get stuck however continue to the next step where we will be running Formality to confirm the answers.

Question 10. How many compare points and of what type would you expect for **harry_3or** ?

.....

Question 11. How many compare points and of what type would you expect for **harry_2or** ?

.....

Question 12. Which of these compare points will match between **harry_3or** and **harry_2or** ?

.....

Question 13. Which of these compare will remain unmatched between **harry_3or** and **harry_2or**?

.....

Question 14. Apart from the compare points what else will match or not be matched?

.....

Question 15. Which of the compare points will **pass** verification ?

.....

Question 16. Which of the compare points will **fail** verification ?

.....

Question 17. For the points that fail what will be the input stimulus that causes it to fail ?

.....

2. Inspect run2_fm.tcl. This is reading in **harry_3or** and **harry_2or**.
3. Run run2_fm.tcl

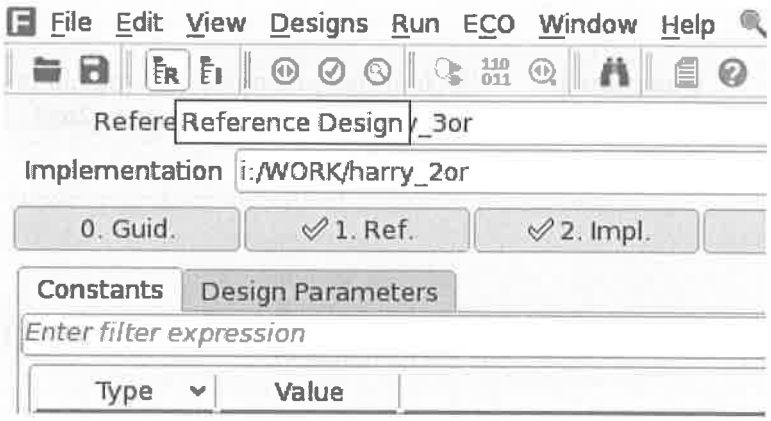
```
unix% fm_shell -f run2_fm.tcl |& tee -i run2_fm.log
```

4. Firstly let us look at how Formality has interpreted **harry_3or**. From run2_fm.tcl you will see that **harry_3or** was read into the reference container.

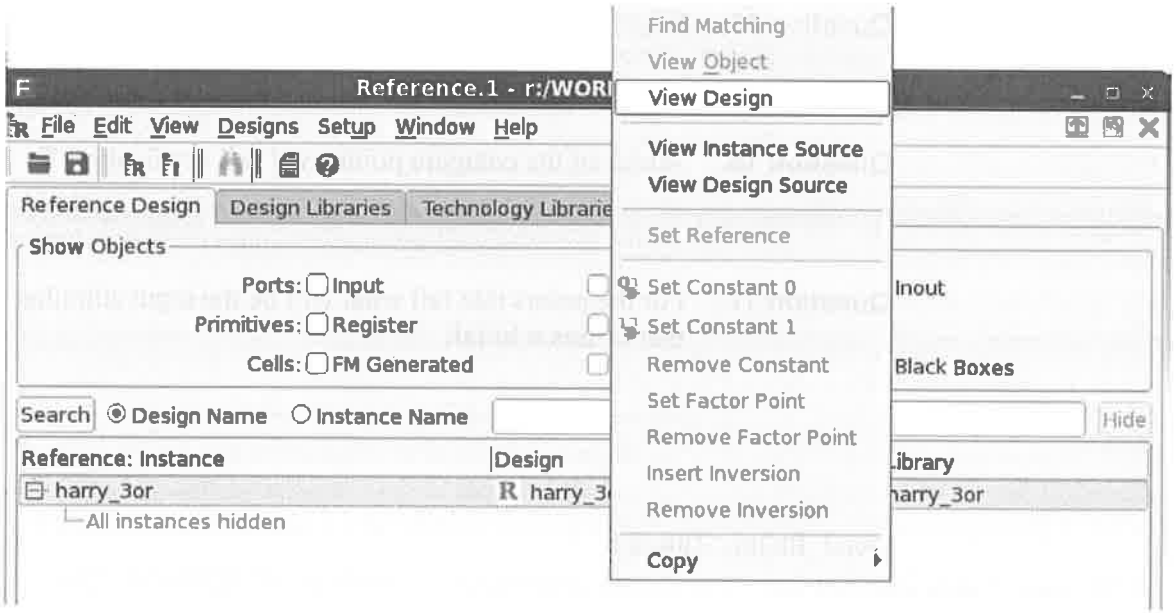
   Start up the GUI

```
fm_shell (setup) start_gui
```

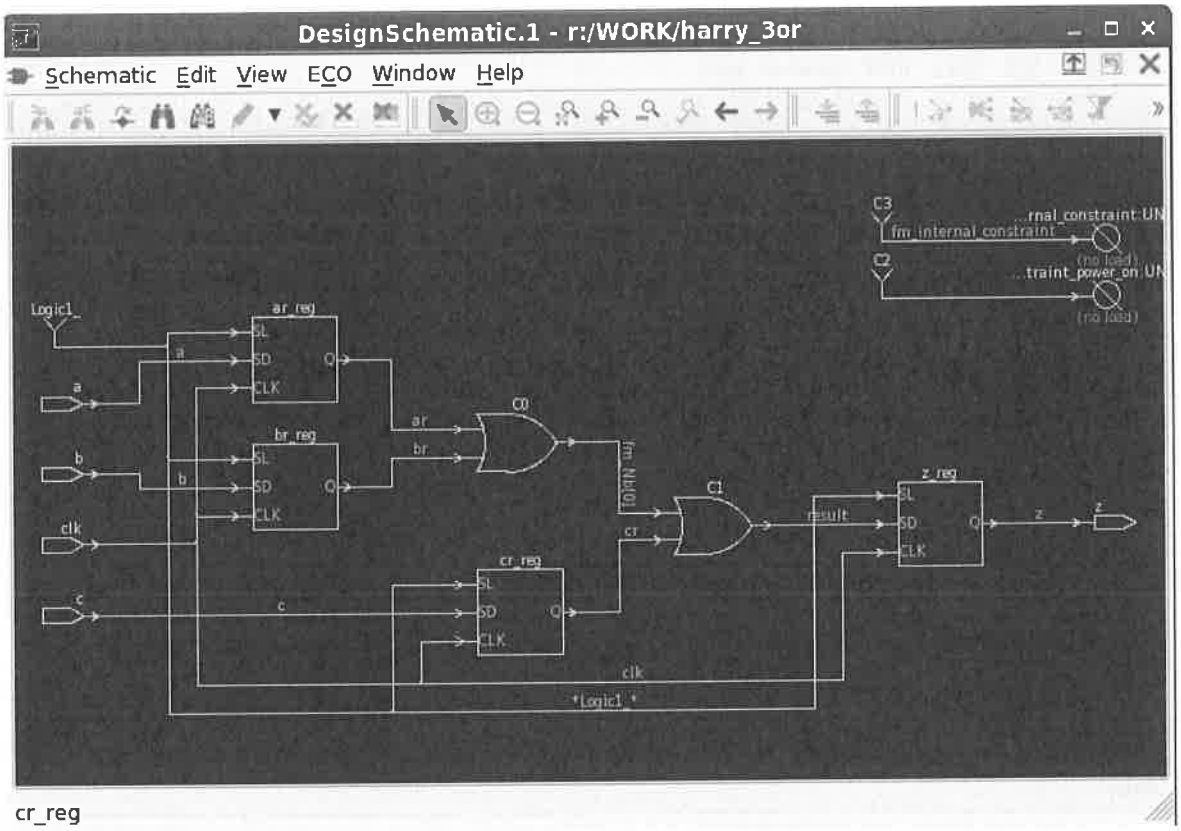
Click on View Reference Hierarchy icon 



Select the harry3_or level of hierarchy and View Design

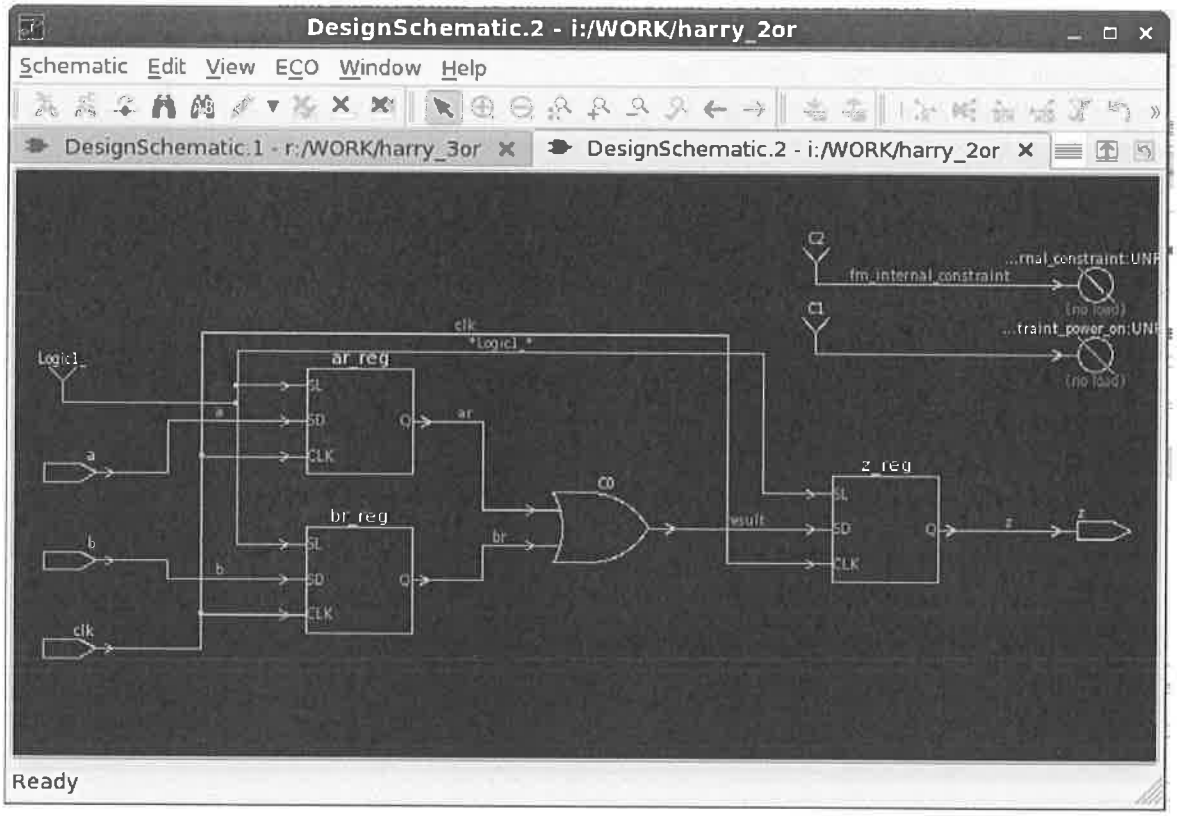


You should now see



That is 4 registers with 2 or gates between them. Let us pull up the Implementation Design schematic. Repeat the above few steps but clicking on the View Implementation Hierarchy icon 

The implementation design will look like



Adjust your answers to questions 10 – 17 as appropriate if Formality's interpretation of the RTLs wasn't what you were expecting.

5. Quit out the GUI (not the shell) for the moment
6. Run the match command

```
fm_shell (setup) match
```

7. You should now see the match summary

```
********************************** Matching Results **********************************
 4 Compare points matched by name
 0 Compare points matched by signature analysis
 0 Compare points matched by topology
 3 Matched primary inputs, black-box outputs
 1(0) Unmatched reference(implementation) compare points
 1(0) Unmatched reference(implementation) primary inputs, black-box outputs
--------------------------------------------------------------------------------
Unmatched Objects                                                  REF        IMPL
--------------------------------------------------------------------------------
 Input ports (Port)                                                  1           0
 Registers                                                           1           0
   DFF                                                               1           0
********************************************************************************

1
```

8. To get a list of all the matched points :

```
fm_shell (match) report_matched_points
```

```
7 Matched points:

  Ref  DFF        Name(Last) r:/WORK/harry_3or/ar_reg
  Impl DFF        Name(Last) i:/WORK/harry_2or/ar_reg

  Ref  DFF        Name(Last) r:/WORK/harry_3or/br_reg
  Impl DFF        Name(Last) i:/WORK/harry_2or/br_reg

  Ref  DFF        Name(Last) r:/WORK/harry_3or/z_reg
  Impl DFF        Name(Last) i:/WORK/harry_2or/z_reg

  Ref  Port       Name(Last) r:/WORK/harry_3or/a
  Impl Port       Name(Last) i:/WORK/harry_2or/a

  Ref  Port       Name(Last) r:/WORK/harry_3or/b
  Impl Port       Name(Last) i:/WORK/harry_2or/b

  Ref  Port       Name(Last) r:/WORK/harry_3or/clk
  Impl Port       Name(Last) i:/WORK/harry_2or/clk

  Ref  Port       Name(Last) r:/WORK/harry_3or/z
  Impl Port       Name(Last) i:/WORK/harry_2or/z
```

9. To get a list of all the unmatched points :

```
fm_shell (match) report_unmatched_points
```

```
****************************************************
2 Unmatched points (2 reference, 0 implementation):
  Ref  DFF        r:/WORK/harry_3or/cr_reg
  Ref  Port       r:/WORK/harry_3or/c
```

We are now in a position to confirm the answers to the following questions. Note that an input port, though will be required to match, is not considered a compare point – that is a compare point is at the end of a logic cone not the beginning.

Question 10. How many compare points and of what type would you expect for **harry_3or** ?

5 .

4 flip-flops (DFF) ar_reg, br_reg, cr_reg, z_reg,

1 output port z

Question 11. How many compare points and of what type would you expect for harry_2or ?

4

3 flip-flops (DFF) ar_reg, br_reg, zr_reg

1 output port z

Question 12. Which of these compare points will match between **harry_3or** and **harry_2or** ?

4

3 registers ar_reg, br_reg, z_reg

1 output port z

Question 13. Which of these compare points will remain unmatched between **harry_3or** and **harry_2or**?

The register cr_reg

Question 14. Apart from the compare points what else will match or not be matched?

The input ports a,b will match

The input port c of **harry_3or** does not match.

10. Let us look at the compare point z_reg. Use the report_unmatched_points command to confirm that cr_reg is in the logic cone for z_reg but is unmatched

```
fm_shell (match) report_unmatched_points \
r:/WORK/harry_3or/z_reg
```

If there is a functional path from an unmatched input to a logic cone compare point then that compare point of the logic cone will fail verification.

So without running verify or looking at the details of the logic cone an unmatched input may be indicative that the verification will fail.

11. Run the verify step

```
fm_shell (match) verify
```

```
Status: Verifying...
    Compare point z_reg failed (is not equivalent)

********************************** Verification Results **********************************
Verification FAILED
-------------------
 Reference design: r:/WORK/harry_3or
 Implementation design: i:/WORK/harry_2or
 3 Passing compare points
 1 Failing compare points
 0 Aborted compare points
 0 Unverified compare points
------------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop    BBNet     Cut     Port     DFF      LAT    TOTAL
------------------------------------------------------------------------------------------
Passing (equivalent)           0       0        0       0        1       2        0        3
Failing (not equivalent)       0       0        0       0        0       1        0        1
******************************************************************************************
```

12. Run report_failing_points and report_passing_points command

```
fm_shell (verify) report_failing_points
```

```
1 Failing compare point (1 matched, 0 unmatched):

  Ref  DFF        r:/WORK/harry_3or/z_reg
  Impl DFF        i:/WORK/harry_2or/z_reg
```

```
fm_shell (verify) report_passing_points
```

```
3 Passing compare points:

  Ref  DFF        r:/WORK/harry_3or/ar_reg
  Impl DFF        i:/WORK/harry_2or/ar_reg

  Ref  DFF        r:/WORK/harry_3or/br_reg
  Impl DFF        i:/WORK/harry_2or/br_reg

  Ref  Port       r:/WORK/harry_3or/z
  Impl Port       i:/WORK/harry_2or/z
```

13. You are now in the position to confirm the answers to the questions :

Question 15. Which of the compare points will **pass** verification ?

3 passing points. ar_reg,br_reg, z

Question 16. Which of the compare points will **fail** verification ?

1 failing point. z_reg.

14. Consider the list of matched compare points (answer to question 12) , passing points and failing points

Question 18. Does the number of matched compare points equal the sum of failing points and passing points ?

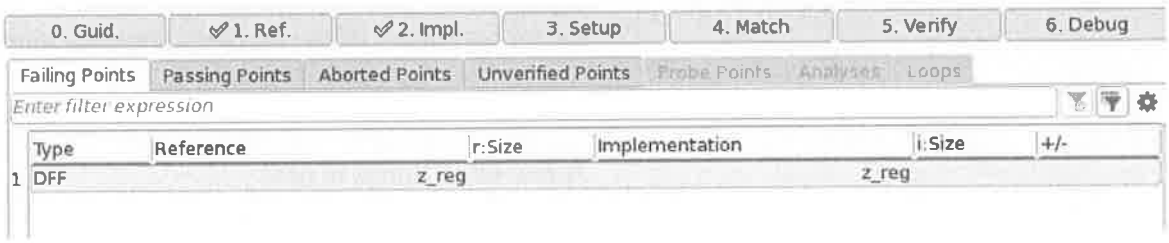
.....

15. Let us now confirm the stimulus that shows the failure for the failing point z_reg

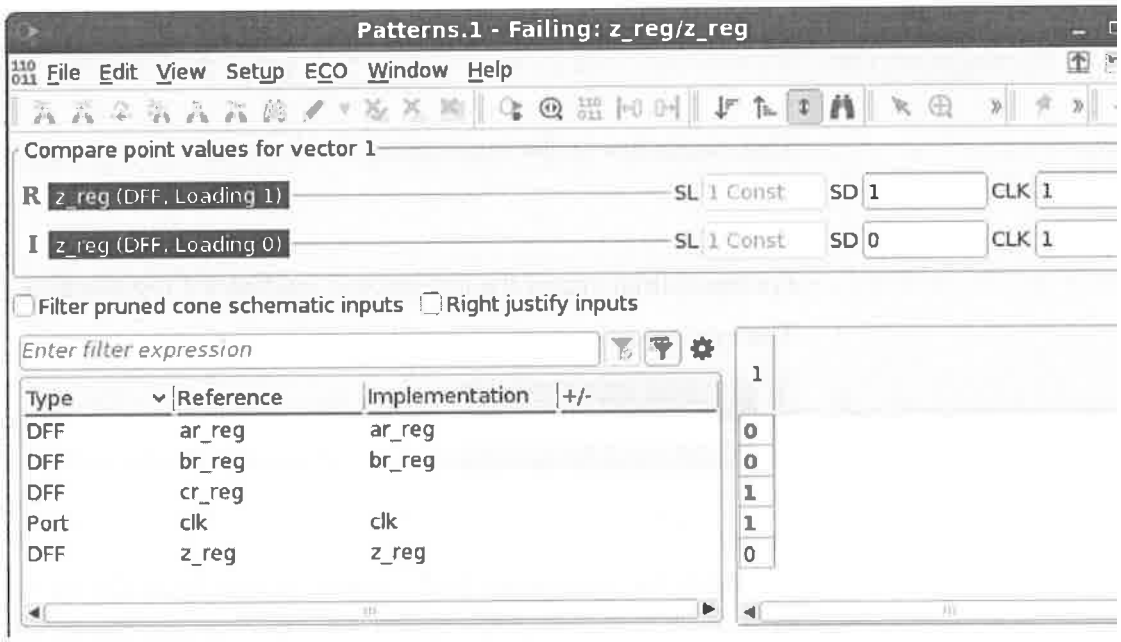
Start up the GUI

```
fm_shell (setup) start_gui
```

Select z_reg under the failing points of the Debug tab.



And Show Patterns for that point.



So the full answer is :

Question 17. For the points that fail what will be the input stimulus that causes it to fail ?

The failing vector has cr_reg has 1 . If cr_reg was a constant 0 z_reg would pass verification.

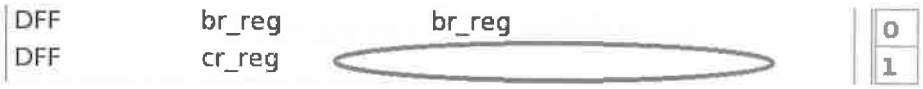
ar_reg and br_reg are 0 .If either or both were 1 that would mask the 1 from cr_reg . That is, for example, if ar_reg was 1 z_reg would be 1 regardless of cr_reg.

The clk is 1 – this just means that the z_reg is being clocked.

z_reg is in the stimulus because the value of z_reg could depend on the previous state if for example it wasn't being clocked. Here it is a greyed out value which means that you will get the failure for this vector regardless of the registers previous state.

A few other things to note :

cr_reg is not in the logic cone of the implemention.



This empty row in the implemention column is how an umatched input shows up.

For this failing vector the reference is loading a 1 (eg clocking 1 on data pin)

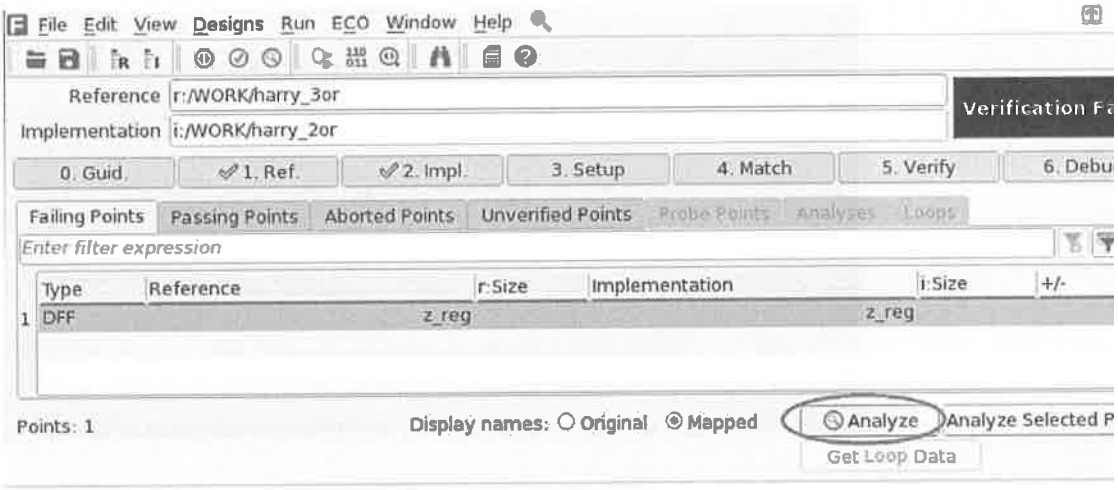
The implemention is loading a 0.

SL stands for sychronous load – that is an enable pin for the register. If you refer back to to RTL the registers don't have an enable and therefore SL is a constant 1 – that is the register is permantly enabled.

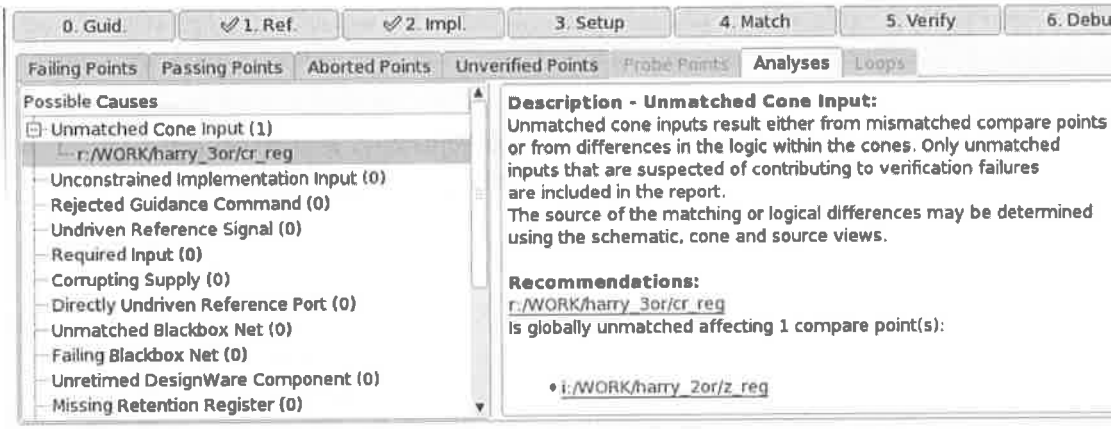
SD is the value on synchronous data pin. Here because the flip-flops don't have an enable or an asychronous reset the value of the SD will be the same as the value loaded.

16. We know the cause of the failure – the unmatched cr_reg in the logic cone. Let us see what the analyze_points command thinks is the cause of the failure.

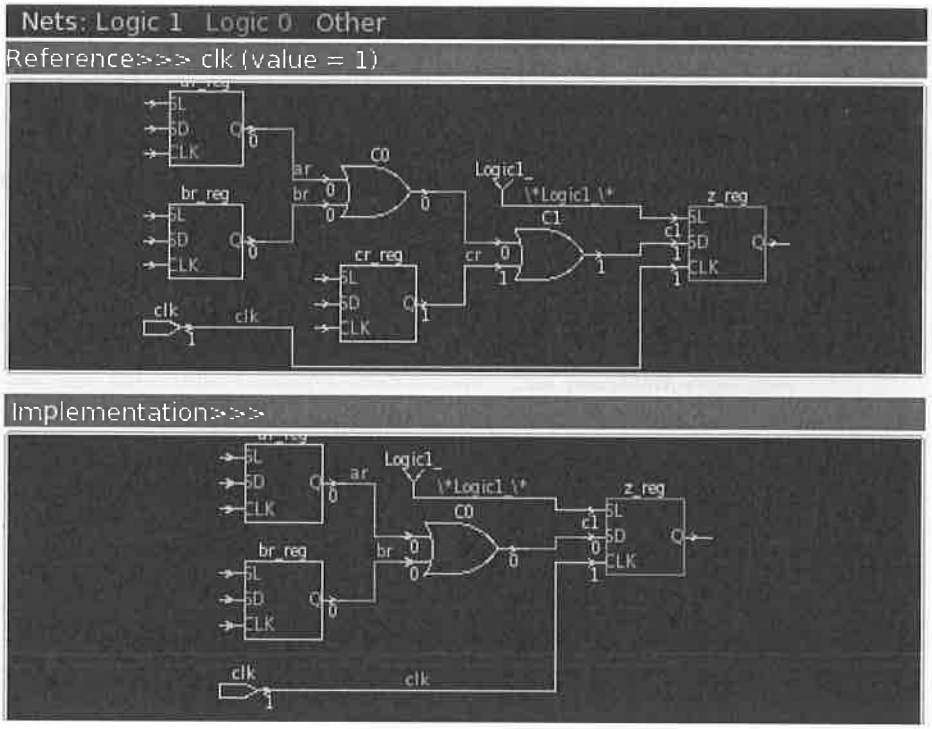
Go to the failing points list under the debug tab and click on the **Analyze** button.



You should see the below which is a description cr_reg being unmatched in the failing point z_reg. Note it says 'globally unmatched' that is cr_reg is not matched to anything in the implementation at all .



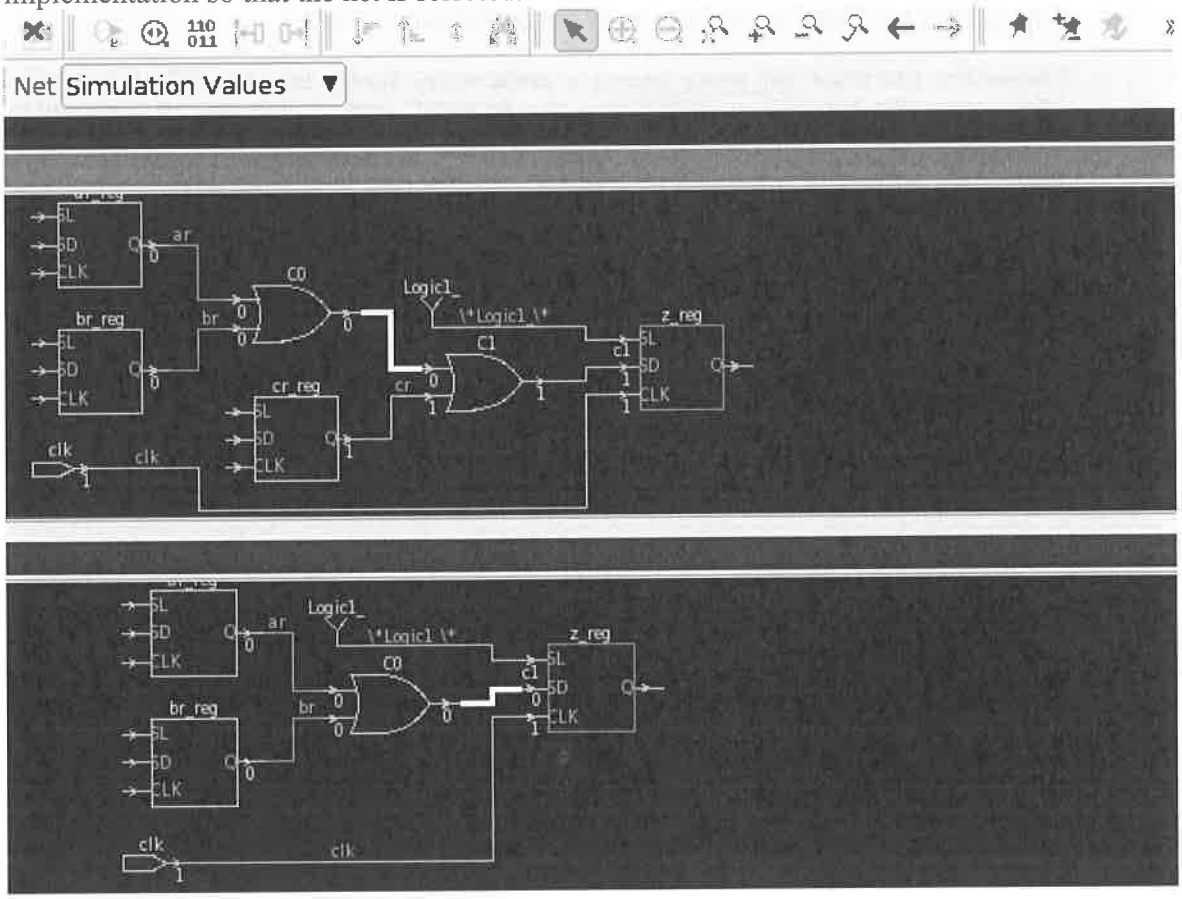
17. Let us investigate the logic cone with probe points. Return to the Pattern window and click on the show logic cones icon  . You may need to do Expand Schematic (Ctrl+E) to see the full Ref cone.



Note that the register cr_reg is in white – this is the way the logic cone schematic indicates something is unmatched.

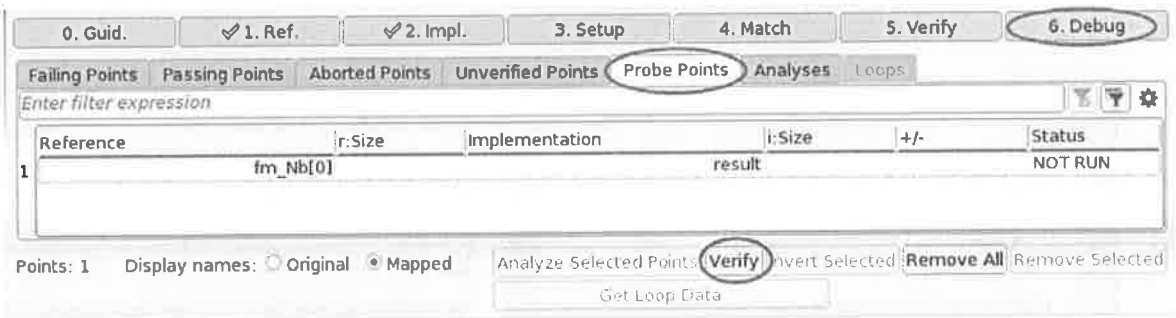
Observe that the probe icons are currently greyed out. as no net pairs are currently selected in reference and implementaion.

Click on the net of the output of the C0 **or** gate in both reference and implementation so that the net is selected.



The probe icon  should now no longer be greyed out. Click on it.

You should now see a set_probe_point command echoed out to the log file and a prope point in the probe point list. Click on the Verify button on the probe point tab



One should see that the verification of the probe point passes.

```
Formality (verify)> set_probe_points r:/WORK/harry_3or/fm_Nb[0] i:/WORK/harry_2or/result ;
Set user probe between 'r:/WORK/harry_3or/fm_Nb[0]' and 'i:/WORK/harry_2or/result'
1
Formality (verify)> verify -probe
Reference design is 'r:/WORK/harry_3or'
Implementation design is 'i:/WORK/harry_2or'

****************************** Probe Verification Results ******************************
Probe Verification SUCCEEDED
----------------------------
 Reference design: r:/WORK/harry_3or
 Implementation design: i:/WORK/harry_2or
 1 Passing probe points
****************************************************************************************
1
```

In this case it was obvious the 2 nets were the same but in more complicated examples probe points can be a useful debugging aid.

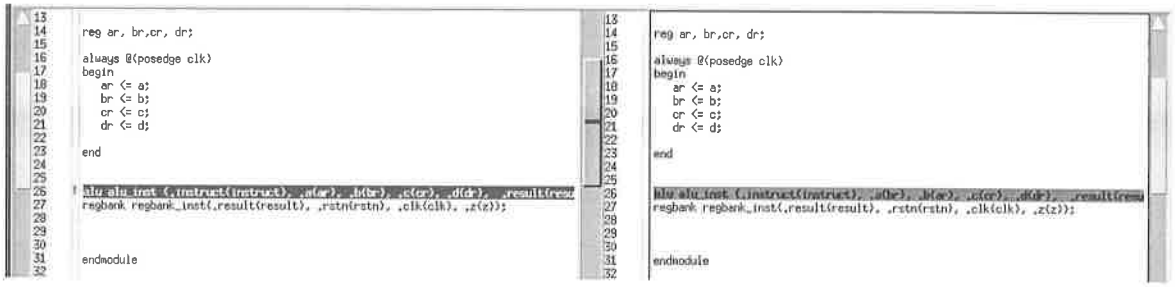
18. Quit out of Formality.

Task 3. (Optional) Looking at expected failures on third simple example

In this section we will build on the previous 2 tasks by looking at an another example tom1.v and tom2.v

1. Inspect the RTL tom1.v and tom2.v . If, for example, you compare the 2 files in tkdiff

```
unix% tkdiff tom1.v tom2.v
```



One will see that the only differences between the RTL is that the a and b input port connections to alu_inst have been swapped. In tom1 **a** is connected to **ar** and **b** to **br** . In tom2 **a** is connected to **br** and **b** to **ar**.

Question 18. What will this difference cause to fail in verifcation ?

.....

2. Run run3_fm.tcl which compares the 2 RTL

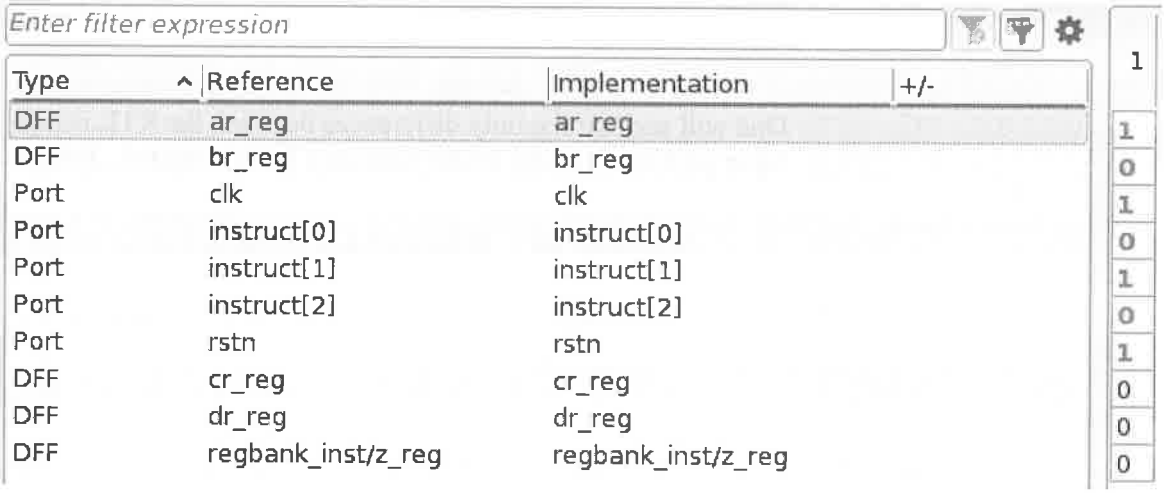
```
unix% fm_shell -f run3_fm.tcl |& tee -i run3_fm.log
```

3. We have a failing verification

```
****************************************************************************************

Status:  Verifying...
    Compare point regbank_inst/z_reg failed (is not equivalent)

********************************** Verification Results **********************************
Verification FAILED
--------------------
 Reference design: r:/WORK/tom
 Implementation design: i:/WORK/tom
 5 Passing compare points
 1 Failing compare points
 0 Aborted compare points
 0 Unverified compare points
----------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop   BBNet     Cut    Port     DFF     LAT   TOTAL
----------------------------------------------------------------------------------------
Passing (equivalent)           0       0       0       0       1       4       0       5
Failing (not equivalent)       0       0       0       0       0       1       0       1
****************************************************************************************
```

4. Pull up the pattern window for the failing point.

Use the icon  (Most Required Inputs) to get the red of the failing vectors to the top.



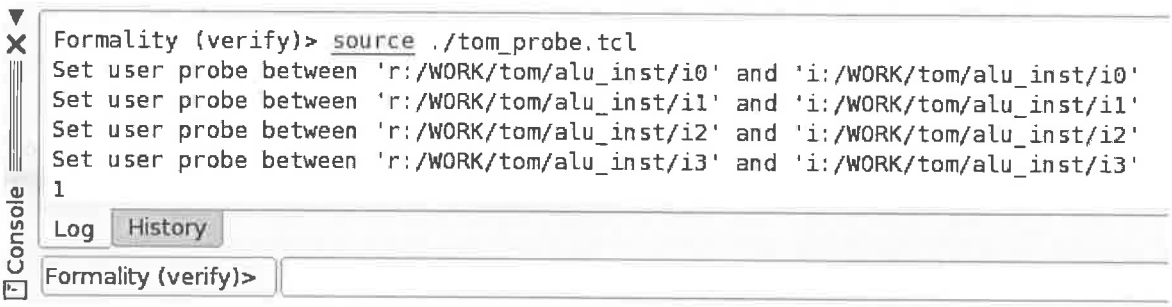
Observe that the both the failing vectors have **instruct [2:0]** as 010. And **ar_reg** and **br_reg** have opposite values.

If one looks at the tom1 RTL

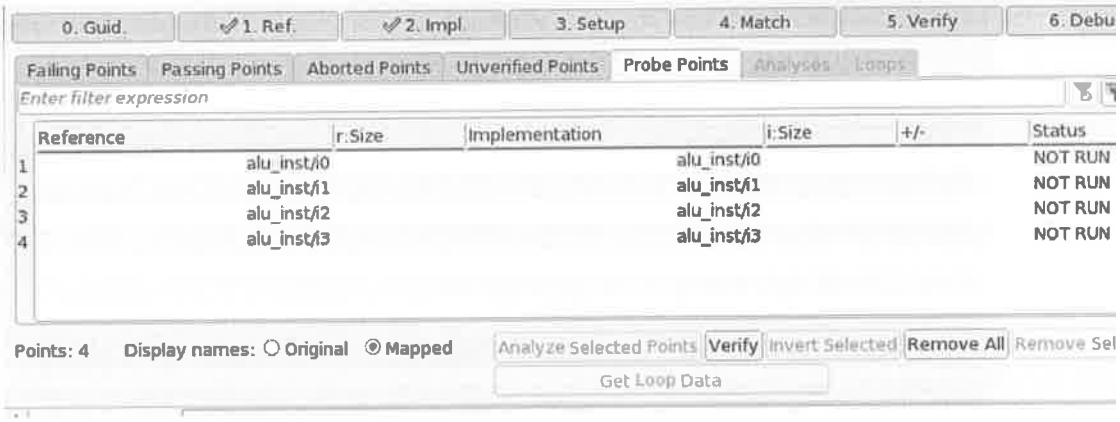
```
always @(i0 or i1 or i2 or i3 or instruct )
begin
 if (instruct == 3'b000 )
 begin
     result = i0;
 end
 else if   (instruct == 3'b010)
 begin
    result = i1;
 end
 else if   (instruct == 3'b100)
 begin
    result = i2;
 end
 else
     result = i3;

end
```

One sees that instruct 3'b010 is the branch where the result is i1. Let us put probe points on these nets . One can pull these from the schematic but for ease of demonstrastration tom_probe.tcl is provided.

5. Inspect and then source tom_probe.tcl



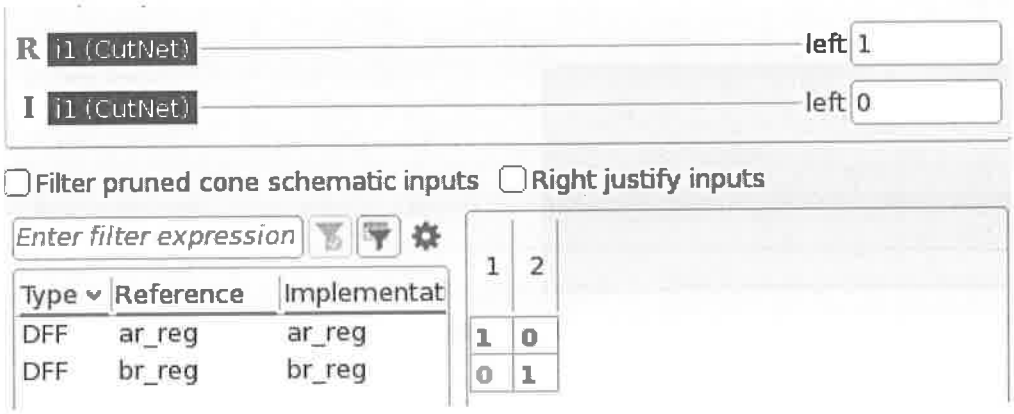
6. Pull up the probe point list



And run verify –probe

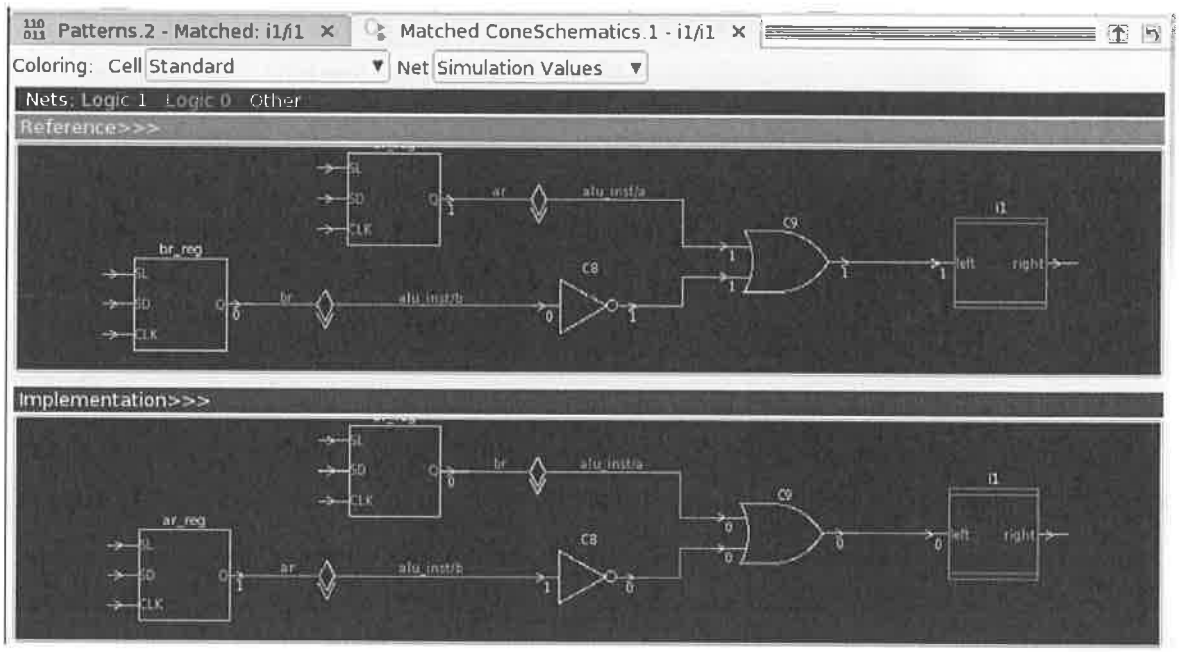
Observe the probe point i0, i2 , i3 pass verification but as expected i1 fails.

Select i1 and pull up the pattern window (ie select the failing probe point and 

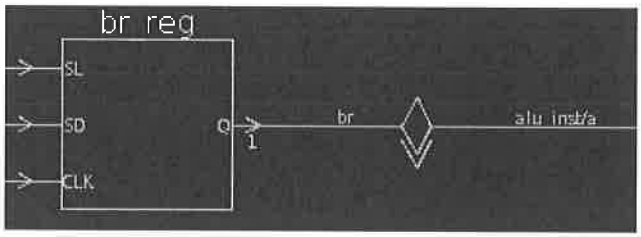


Now one is debugging a smaller logic cone with just 2 inputs.

If one pulls up the logic cone schematic  for the failing probe point one sees the below. Note one can use the "Maximize All Views"  or "Detach view"  to display the schematic without sharing the view with the patterns.



It now becomes clearer that the registers have been swapped and if one looks at the hierarchy crossing point (the diamond symbols) one can see where this is :



Which of course is same swap that we observed when compared the RTL with tkdiff.

Answers / Solutions

Task 1. Looking at expected failures in the pattern window

Question 1. When one compares fred_or and fred_and how many compare points would you expect and of what type ?

One compare point, z, of type port.

Question 2. Apart from the compare points what else will match ?

The input ports a and b

Question 3. For what input stimulus would a comparison of fred_or and fred_and give different results ?

When a,b is is 1,0 or 0,1 the **or** function will have an output of 1 the **and** gate will have an output of 0.

For 0,0 and the 1,1 fred_or and fred_and are the same.

Question 4. Find all the places in the transcript where you can tell that the verification has failed?

```
verify
Reference design is 'r:/WORK/fred_or'
Implementation design is 'i:/WORK/fred_and'

************************************* Matching Results *************************************
 1 Compare points matched by name
 0 Compare points matched by signature analysis
 0 Compare points matched by topology
 2 Matched primary inputs, black-box outputs
 0(0) Unmatched reference(implementation) compare points
 0(0) Unmatched reference(implementation) primary inputs, black-box outputs
********************************************************************************************

Status:  Verifying...
    Compare point z failed (is not equivalent)

************************************ Verification Results ************************************
Verification FAILED
--------------------
 Reference design: r:/WORK/fred_or
 Implementation design: i:/WORK/fred_and
 0 Passing compare points
 1 Failing compare points
 0 Aborted compare points
 0 Unverified compare points
--------------------------------------------------------------------------------------------
Matched Compare Points     BBPin    Loop    BBNet     Cut     Port     DFF     LAT    TOTAL
--------------------------------------------------------------------------------------------
Passing (equivalent)           0       0        0       0        0       0       0        0
Failing (not equivalent)       0       0        0       0        1       0       0        1
********************************************************************************************
Info:  Try the analyze_points command to see if Formality can determine potential
causes, or suggest next steps for a FAILED or INCONCLUSIVE verification.
See the man page for analyze_points usage and options.
Info:  Formality Guide Files (SVF) can improve verification success by automating setup.
0
0
```

In order:

1) Failing points will be echoed to the screen as 'Compare point <point name> failed'

2) At the end of the verification you will have a line that says 'Verification FAILED'

3) A line where 'n Failing compare points'

4) The categorizationsummary of those failing points (Port, DFF (flip-flop)etc.)

5) The return value of the 'verify' command is 0.

Question 5. From the transcript and the report_matched_points confirm you're to Question 1 and 2.

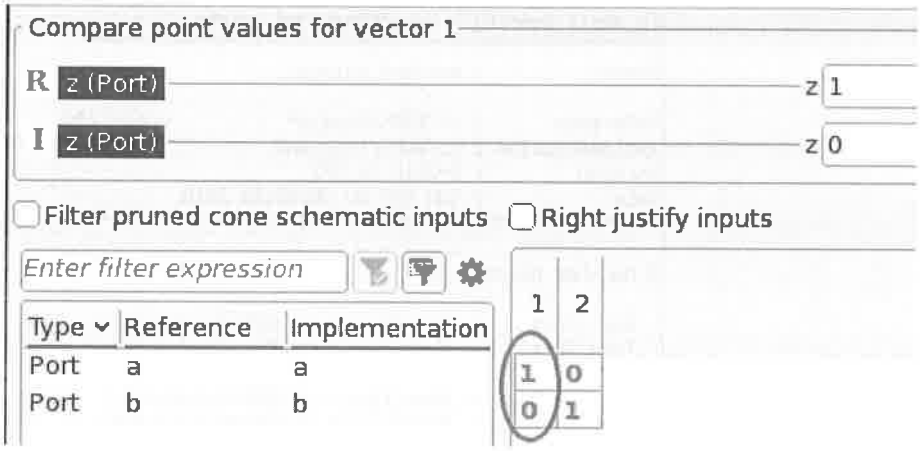
```
fm_shell (verify)> report_matched_points
****************************************************
Report          : matched_points

Reference       : r:/WORK/fred_or
Implementation  : i:/WORK/fred_and
Version         : O-2018.06-SP2
Date            : Sun Sep 30 08:59:38 2018
****************************************************

3 Matched points:

  Ref  Port       Name(Last) r:/WORK/fred_or/a
  Impl Port       Name(Last) i:/WORK/fred_and/a

  Ref  Port       Name(Last) r:/WORK/fred_or/b
  Impl Port       Name(Last) i:/WORK/fred_and/b

  Ref  Port       Name(Last) r:/WORK/fred_or/z
  Impl Port       Name(Last) i:/WORK/fred_and/z
```

Look in run1_fm.log or scroll up on the screen you should see after the match command

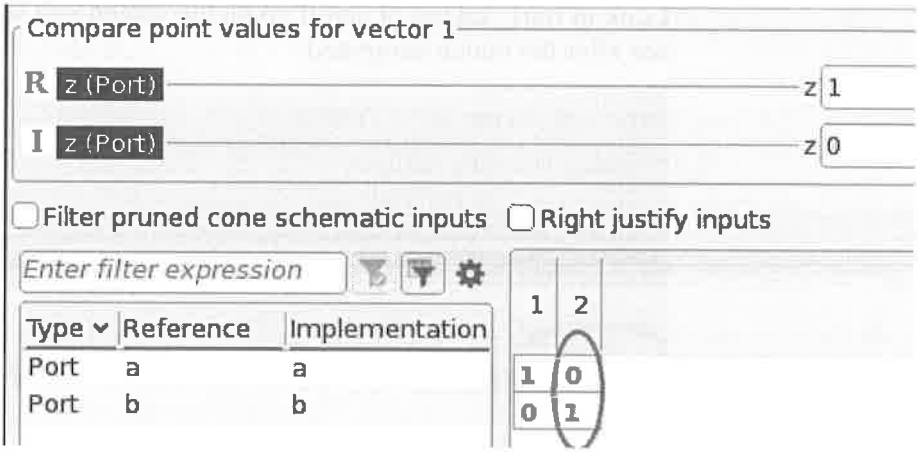
```
match
Reference design is 'r:/WORK/fred_or'
Implementation design is 'i:/WORK/fred_and'
Status:  Checking designs...
Status:  Building verification models...
Status:  Generating datapath components ...
Status:  Qualifying datapath components ...
Status:  Datapath qualification complete.
Status:  Matching...

****** ***************************** Matching Results ******************
 1 Compare points matched by name
 0 Compare points matched by signature analysis
 0 Compare points matched by topology
 2 Matched primary inputs, black-box outputs
 0(0) Unmatched reference(implementation) compare points
 0(0) Unmatched reference(implementation) primary inputs, black-box outp
****************************************************************************
```

Question 7. What are the input values for which the verification fails ?

vector 1 read as a column a,b 0 1

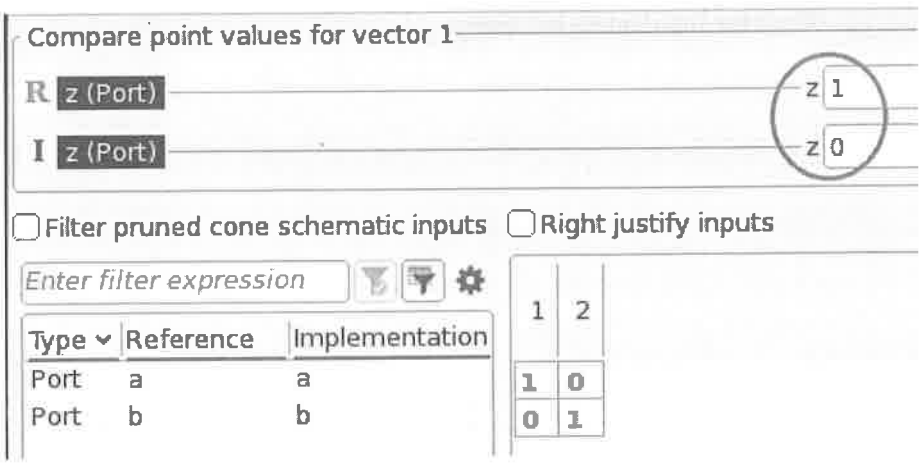


Vector 2 read as a column a,b 1, 0



Question 8. What are the output values in the reference and implementation ?

The reference is 1 and the implementation 0 for vector 1 and similarly for vector 2.



Question 9. Are these the results you would expect given your answer to Question 3. ?

Yes if you had the correct answers to Question 3.

Task 2. Looking at expected failures on a second simple example

Answers to Question 10 – 17 in the text

Question 18. Does the number of matched compare points equal the sum of failing points and passing points ?

Yes. Apart from a number of special circumstances this will always be the case.

This page intentionally left blank.

4 Multiple Stage Verifications

Learning Objectives

This will lab will focus on mutiple stage verification

After completing this lab, you should be able to:

- Be able to setup and debug a multistage Design Compiler synthesis flow

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

In this lab we will running verificationsfor both end to end and single stage verifications

Task 1. Running synthesis

First run synthesis.

1. Go to lab4

```
unix% cd lab4
```

2. Inspect the DC script dc.tcl. Observe that there is a netlist and SVF written out for each of the stages elaboration; first compile; test insertion and incremental compile.

3. Run synthesis

```
unix% dc_shell -topo -f dc.tcl | tee -i dc.log
```

Task 2. Running end to end verification

Let us verify the RTL to the final netlist written out by the script mapped/STOTO.v

1. Inspect the script intended to run this verification fm_all.tcl

Question 1. Isthere any SVF missing to the set_svf command in fm_all.tcl ?

.....

Correct the fm_all.tcl to add in any missing SVF.

2. Run the end to end Formality verification. Note the use of the switch – **name_suffix** to identify the run.

```
unix% fm_shell -name all -over -f fm_all.tcl |tee -i fm_all.log
```

Note: The **–name_suffix** modifies the default name of some of the files fm_shell creates.In the above command line, in particular, the ASCII SVF will be written to formality_all_svf/svf.txt (instead of the default formality_svf/svf.txt)

With the correct SVF files pointed to, the verification should pass.

Task 3. Checking the timestamp information

Say there was a failing verification and there was some dispute as to whether the SVF corresponded to the synthesis run.

1. Grep the comments from the SVF from Task 2

```
unix% grep '#'   formality_all_svf/svf.txt
```

2. Note : In the dc.tcl for lines have been added to write out time information to the transcript. Here this is for lab purposes, but it not uncommon to write out such information . Grep this information from the dc.log (the transcript from Task 1.)

```
unix% grep '^INFO'   dc.log
```

Confirm that within to within a few seconds the timestamp information between dc.log and svf.txt matches.

Task 4. Verifying the first compile stage

1. Inspect the script intended to run the verification on first compile stage fm_first.tcl

Question 2. Is there any SVF missing to the set_svf command in fm_first.tcl ?

.....

Run the verification and confirm it passes

```
unix% fm_shell -name first -over-f fm_first.tcl |tee -i fm_first.log
```

2. Inspect fm_bug.tcl a verification that verifying the RTL to the first netlist using all the SVF .

Question 3. Is "./elab.svf ./compile.svf ./dft.svf ./compile_inc.svf"appropriate SVF for verifying this stage?

.....

3. **Without** modifying fm_bug.tcl run the verification

```
unix% fm_shell -name bug-over-f fm_bug.tcl |tee -i fm_bug.log
```

Confirm it still passes.

4. Do a diff between the 2 log files

```
unix% diff fm_first.log fm_bug.log
```

And observe the difference in the method Formality has used to match up the compare points.

< 75 Compare points matched by name

< 0 Compare points matched by signature analysis

> 50 Compare points matched by name

> 25 Compare points matched by signature analysis

In the fm_bug.tcl run the compare points no longer all match up by name.

Signature analysis is a method that Formality uses to match up compare points when matching by name is not sufficient.

Question 4. Why would Formality no longer be able to match some points by name ? (Hint : The dc.tcl script had a change name rule that will radically alter the names of registers greater than 32 characters)

.....

Confirm your answer to Question 4 by inspecting matched_first_bug.txt the output of **report_matched_points** from fm_bug.tcl

Answers / Solutions

Task 1. Running end to end verification

Question 1. Is there any SVF missing to the set_svf command in fm_all.tcl ?

Yes. It is "./elab.svf ./compile.svf ./compile_inc.svf" it should include the dft.svf. For example

set_svf "./elab.svf ./compile.svf ./dft.svf ./compile_inc.svf"

Task 3. Checking the timestamp information

```
unix% grep '#' formality_all_svf/svf.txt
# Active SVF file ./elab.svf
#-------------------------------------------------------------------------------
# This file is automatically generated by Design Compiler
# Filename  : /global/gtseu01/gtseu6/fmtraining/ces/n/FM_2018.06-SP2/lab4/elab.svf
# Timestamp : Sun Sep 30 09:37:32 2018
# DC Version: O-2018.06-SP2 (built Sep 03, 2018)
#-------------------------------------------------------------------------------
#---- Recording stopped at Sun Sep 30 09:37:33 2018
# Active SVF file ./compile.svf
#-------------------------------------------------------------------------------
# This file is automatically generated by Design Compiler
# Filename  : /global/gtseu01/gtseu6/fmtraining/ces/n/FM_2018.06-SP2/lab4/compile.svf
# Timestamp : Sun Sep 30 09:37:34 2018
# DC Version: O-2018.06-SP2 (built Sep 03, 2018)
#-------------------------------------------------------------------------------
#---- Recording stopped at Sun Sep 30 09:37:42 2018
# Active SVF file ./dft.svf
#-------------------------------------------------------------------------------
# This file is automatically generated by Design Compiler
# Filename  : /global/gtseu01/gtseu6/fmtraining/ces/n/FM_2018.06-SP2/lab4/dft.svf
# Timestamp : Sun Sep 30 09:37:42 2018
# DC Version: O-2018.06-SP2 (built Sep 03, 2018)
#-------------------------------------------------------------------------------
#---- Recording stopped at Sun Sep 30 09:37:45 2018
# Active SVF file ./compile_inc.svf
#-------------------------------------------------------------------------------
# This file is automatically generated by Design Compiler
# Filename  : /global/gtseu01/gtseu6/fmtraining/ces/n/FM_2018.06-SP2/lab4/compile_inc.svf
# Timestamp : Sun Sep 30 09:37:45 2018
# DC Version: O-2018.06-SP2 (built Sep 03, 2018)
#-------------------------------------------------------------------------------
#---- Recording stopped at Sun Sep 30 09:37:51 2018
unix% grep '^INFO' dc.log
INFO:Start of RTL read stage: Sun Sep 30 09:37:31 2018
INFO:Start of compile stage: Sun Sep 30 09:37:33 2018
INFO:Start test insert stage: Sun Sep 30 09:37:42 2018
INFO:Start incremental compile stage: Sun Sep 30 09:37:45 2018
INFO:End of script: Sun Sep 30 09:37:51 2018
```

Task 4. Verifying the first compile stage

Question 2. Is there any SVF missing to the set_svf command in fm_first.tcl ?

No.

Question 3. Is "./elab.svf ./compile.svf ./dft.svf ./compile_inc.svf" appropriate SVF for verifying this stage ?

No. dft.svf and compile_inc.svf should not be included.

Question 4. Why would Formality no longer be able to match some points by name? (Hint : The dc.tcl script had a change name rule that will radically alter the names of registers greater than 32 characters)

The change_names in the dc.tcl will truncate all registers names longer than 32 characters. This is quite a few as parts of the design have been ungrouped. This is recorded in compile_inc.svf. But these changes happen after the stage we are verifying to.

If you look in matched_first_bug.txt you will see compare pointslike:

Ref DFF Func(Last) r:/WORK/STOTO/I_MIDDLE_I_DONT_PIPELINE_I_RA_00

Impl DFF Func(Last) i:/WORK/STOTO/I_MIDDLE/I_DONT_PIPELINE/I_RANDOM/int1_reg[0]

The Ref has the truncated names from change_names in the SVF which are different from the names in the Impl .

5 Multi-voltage Designs and UPF

Learning Objectives

This lab will be deal with some simple aspects of UPF and Formality

After completing this lab, you should be able to:

- Describe some simple aspects of the Formality flow and debugging for UPF designs

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

Task 1. Inspecting source files and running synthesis

First let us be clear on the intent of the UPF

1. Go to lab5

```
unix% cd lab5
```

2. Inspect the DC script run1_dc.tcl. Observe that the source UPF is top.upf and the post-compile UPF written out of the script is top_postdc.upf

3. Inspect the UPF and the RTL fred.v. For those not familiar with UPF the intent to put the instance u1 in module fred (which is the top of design) into a switchable power-domain PD_SW.

   Two key lines are :

set_isolation iso1 -domain PD_SW -clamp_value 1 -applies_to outputs

set_isolation_control iso1 -domain PD_SW -isolation_signal u1/iso -isolation_sense low -location self

   The intent is that when the isolation signal u1/iso is asserted (here active low) the outputof power domain PD_SW will be clamped high.

4. Run synthesis

```
unix% dc_shell -f run1_dc.tcl | tee -i run1_dc.log
```

Task 2. Running the verification

Let us run the verification on the synthesis results from Task 1

1. Inspect the script run1_fm.tcl.

Question 1. How many UPF files are loaded in this script and into which containers?

.....

2. Run the verification

```
unix% fm_shell -f run1_fm.tcl | tee -i run1_fm.log
```

Question 2. Has the verification passed?

.....

Question 3. Is this verification sufficient for sign-off (Hint: Look in the transcript for Warning and ATTENTION messages)?

.....

3. Modify run1_fm.tcl to verify the design with the power domains not constrained to be on. Confirm this also passes and the Warning and ATTENTION messages identified in answer to Question 3 are no longer present.

Task 3. (Optional) Exploring the design

Let us confirm our understanding of how Formality is interpreting the UPF

1. Reload the session file saved by run1_fm.tcl (run1_fm.fss)
2. Run the command **report_upf**.

Question 4. What UPF objects has Formality added to the reference container?

.....

Question 5. What UPF objects has Formality added to the implementation container?

.....

Question 6. Explain the difference between the answers to Question 4 and 5?

.....

Question 7. (Optional) What is the other difference between the source UPF (top.upf) and post-DC UPF (top_postdc.upf) and how will this show up in Formality ?

.....

Task 4. Debugging with a known inserted bug

Let us see how a known inserted bug in the netlist shows up in Formality. As with all learning how to debug an approach is to see how known bugs show up in the tool so that later on one can isolate unknown bugs.

1. Inspect the netlist fred_gates_bug1.v
2. ISOLORX1 is an isolation cell. OR2X1 is a normal OR gate. Observe that in the netlist has been hand modified to replace the ISOLORX1 cell for the z[3] output of harry.

```
// ISOLORX1 snps_PD1__iso1_snps_z_3___UPF_ISO
( .D(n[1]), .ISO(n5), .Q(z[3]) );

 OR2X1  ubug (.IN1(n[7]), .IN2(n1), .Q(z[3]));
```

When harry's primary power is switched off the ISOLORX1 cell is still powered. The OR2X1 cell is not and therefore will drive X.

3. Run the verification which uses fred_gates_bug1.v

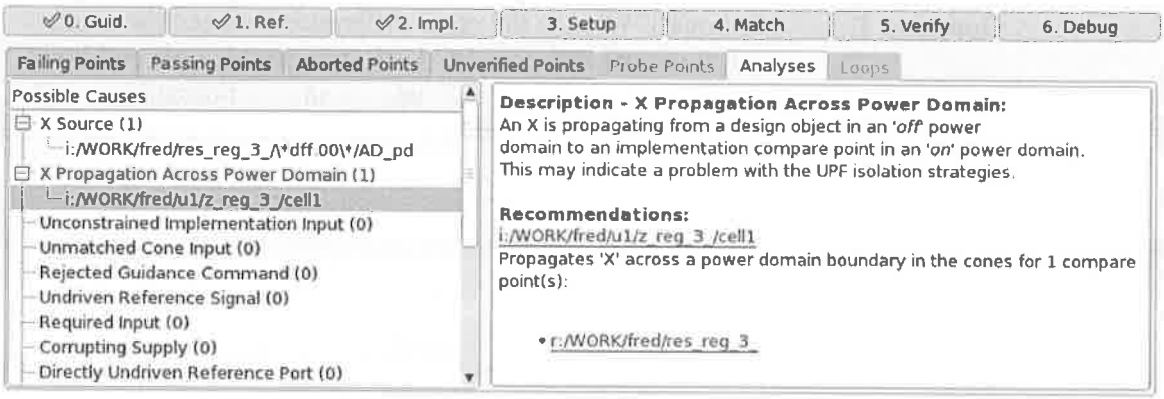
```
unix% fm_shell -f run2_fm.tcl | tee -i run2_fm.log
```

Confirm that the verification fails. Note in order to illustrate some debugging features we have set

set verification_insert_upf_isolation_cutpoints false

in run2_fm.tcl - this will allow the X to propagate through isolation.

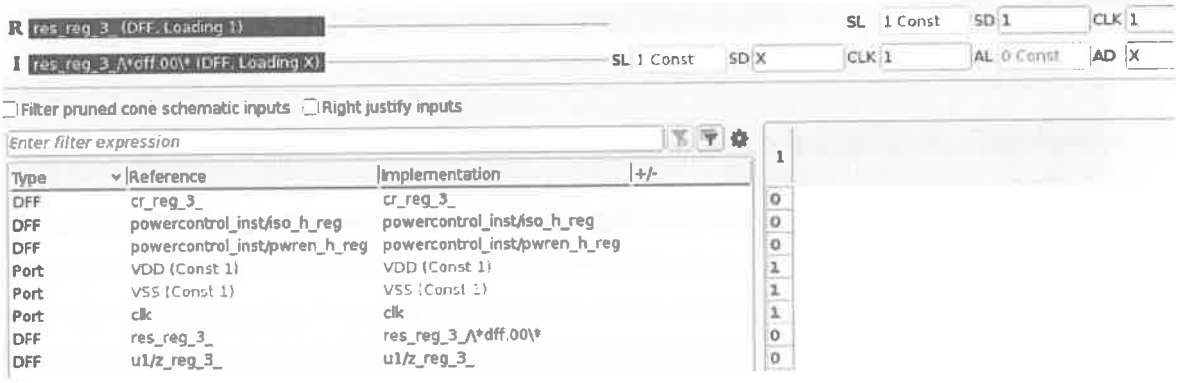
4. There as many ways to debug this. We will explore several to illustrate some of the features of the tool. A good one to allways try is **analyze_points –fail**



It is pointing out two things: 'X Source' : The Impl failing point is getting an X. And secondly 'X Propagation Across Power Domain' : Where it is drawing ones attention to the z_reg_3 the cell driving the OR2X1 gate where we introduced the bug.

Both of these are consistent with the bug we introduced, and would be helpful in isolating the issue in a context where we didn't know what the issue was.

5. Let use explore another view of the failure. Pull up the pattern window for the failing point in the GUI.



Observe the following in the pattern window:

a) The failing vector is loading X in the Impl and 1 in the Ref. The X can be indicative of an non-isolated power domain (Which is exactly what we have done here by replacing the isolation cell with a normal gate)

b) Recall

set_isolation_control iso1 -domain PD_SW -isolation_signal u1/iso -isolation_sense low -location self

The isolation control signal is active low. If you traced this back in the RTL or in Formality this signal is driven by the register iso_h_reg. The failing vector is with iso_h_reg 0 – that is when the power domain PD_SW should be isolated. Again this is consistent with the bug introduced.

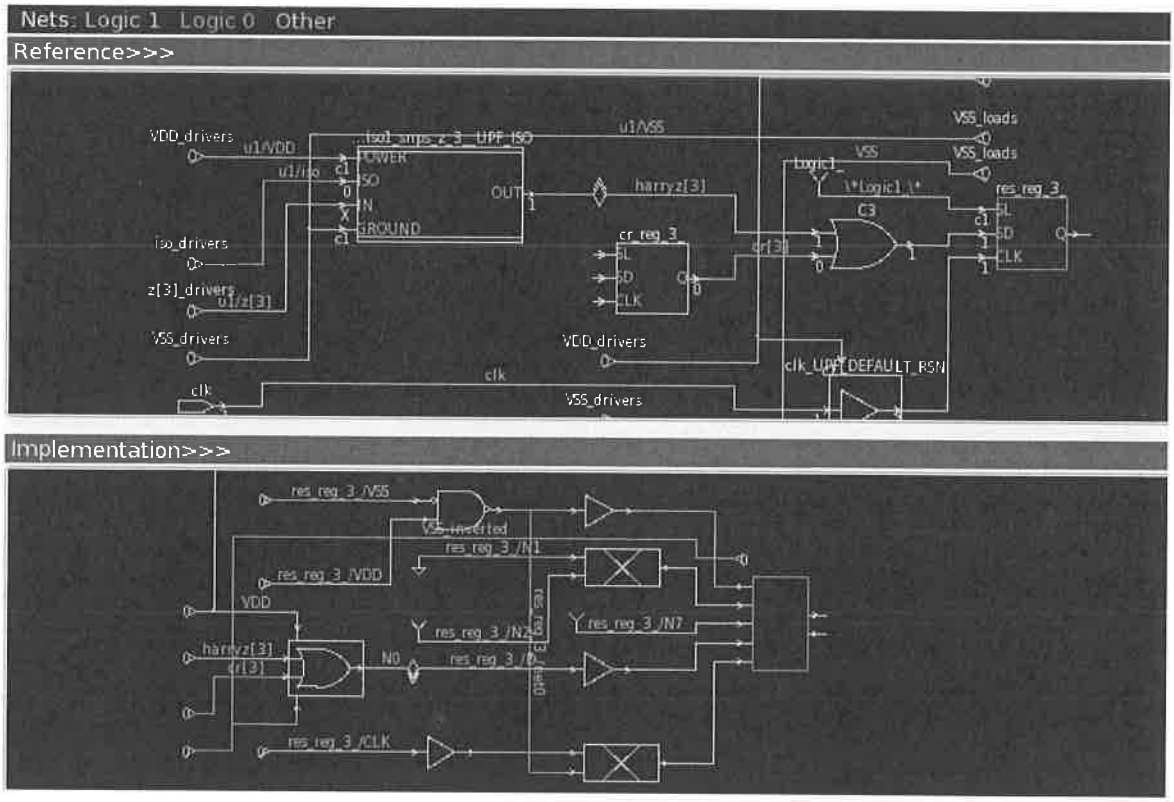
c) Recall also the

create_power_switch sw1 -domain PD_SW -output_supply_port {out VDD_SW} -input_supply_port {in VDD} -control_port {ctrl u1/pwren} -on_state {state1 in {ctrl}}

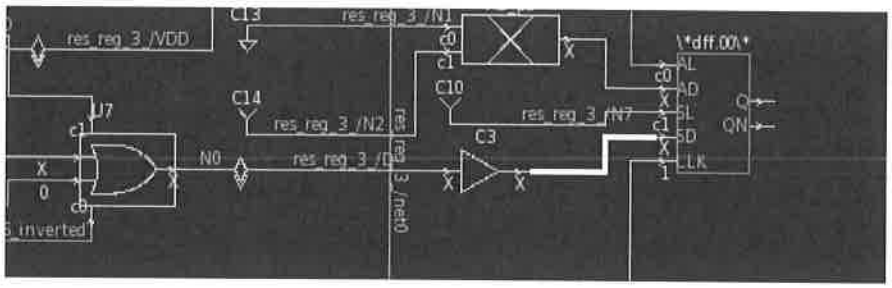
The power switch is on when ctrl is , high which traces back to the pwren_h_reg. That is in the pattern window , pwren_h_reg is 0, indicates that the switch is off – which is again consistent with the bug introduced. (That is the isolation cell has power, where the normal stdcell OR gate does not.)

a) , b) , c) taken together is exactly what we expect the failing vector to be with the bug that was introduced – ie when PD_SW power is off and isolated.

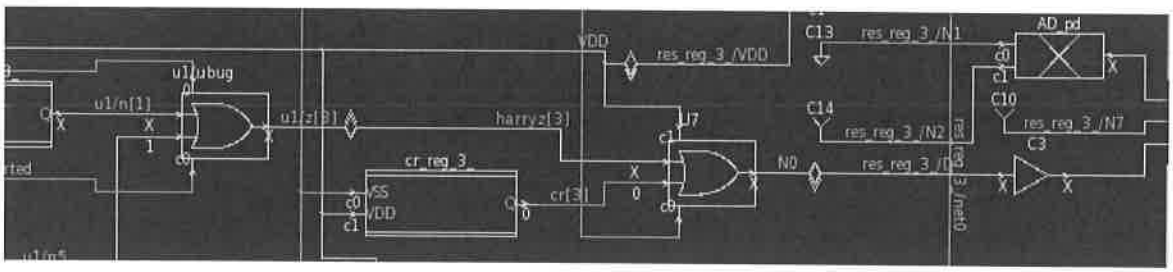
6. Pull up the logic cones for the failing point. 



7. There are several ways to debug the schematic. Expand the Impl schematic (Ctrl-E) Highlight the net that is X at the Impl failing point connected to the SD pin



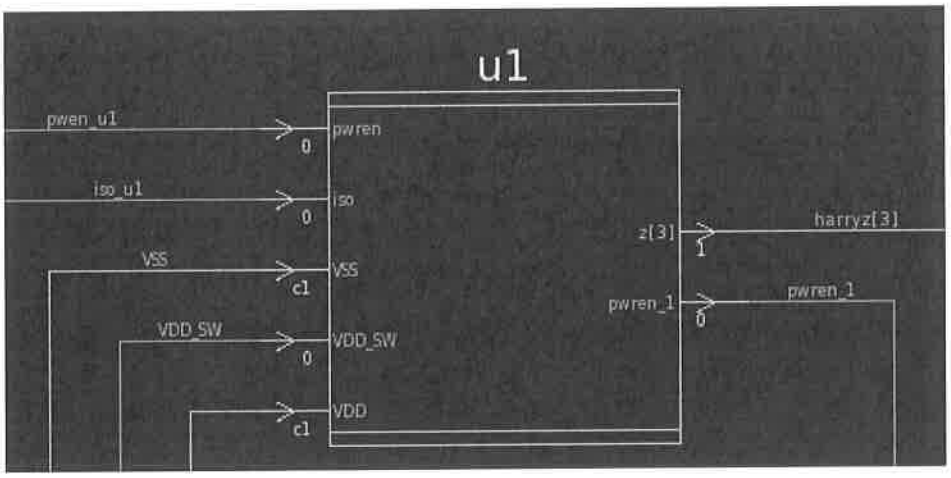
And then do Edit-> Find -> Find X Sources or  . This now traces back the path of the X



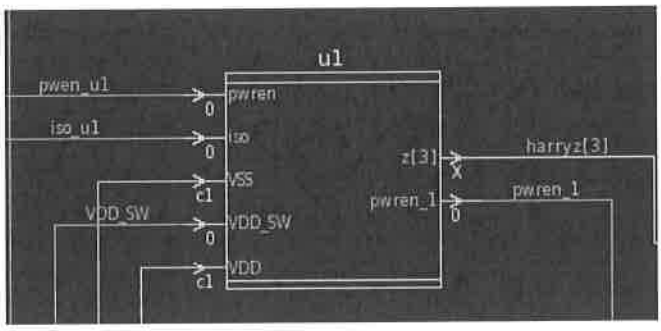
Including u1/ubug the OR gate we inserted for the isolation cell.

8. Another useful thing we can do is group all by parent. In a small schematic like this it may not be necessary – but in a larger logic cone it can be essential.

Use  to group all by parent for both the Ref

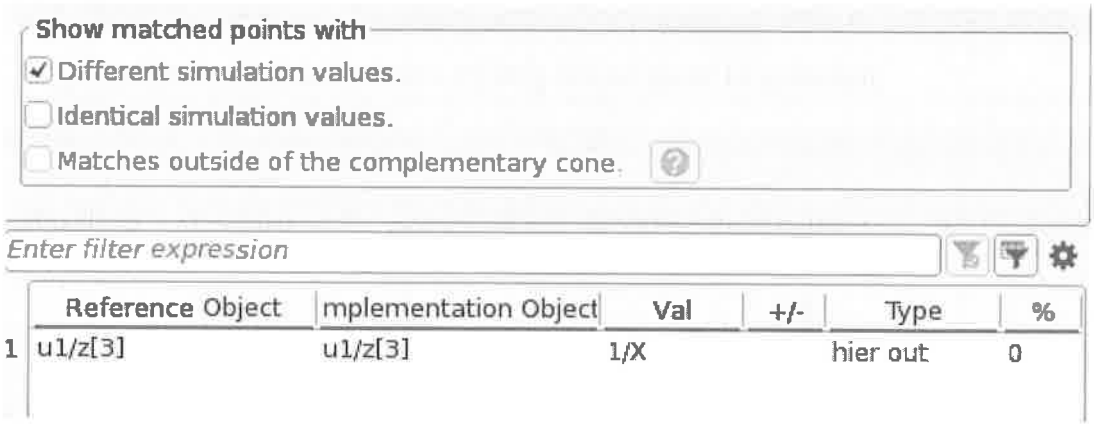


And the Impl



Notice that the z[3] pin is 1 in Ref on u1 and X in the Impl.

9. Another generally useful thing to do is use the matching tool  . Here it shows what we have already seen that the failing vectors does not match on the u1 hierarchy



10. If you have time explore some of the other GUI features - otherwise quit out of Formality. Or re-run the verification and debugging with run3_fm.tcl where we have left the variable **verification_insert_upf_isolation_cutpoints** as it's default.

Answers / Solutions

Task 2. Running the verification

Question 1. How many UPF files are loaded in this script and into which container ?

Two UPF files are used.

top.upf - the original UPF is loaded into the reference container

top_postdc.upf - The UPF written out by DC – loaded into the implementation container

Question 2. Has the verification passed?

Yes.

Question 3. Is this verification sufficient for sign-off ?

No. By default Formality verifies with the power domains forced on.

Observe in the transcript

Reference design is 'r:/WORK/fred'
Implementation design is 'i:/WORK/fred'
Warning: All UPF supplies have been constrained to their ON state and all reference PSTs have been disabled (verification_force_upf_supplies_on = TRUE)

.

And later :

Verification SUCCEEDED

ATTENTION: Verification was run with all UPF supplies constrained to their ON state. This is only a partial verification result as it does not cover all operational states.

Question 4. What UPF objects has Formality added to the reference container?

1 Power State Table

4 isolation cells

1 power switch

Question 5. What UPF objects has Formality added to the implementation container ?

1 Power State Table

1 power switch

Question 6. Explain the difference between the answers to Question 4 and 5?

The post-DC UPF top_postdc.upf contains header information that it is from DC.(Look at the top of the DC written UPF, top_postdc.upf, and you will see a comment line like :

#Generated by Design Compiler(O-2018.06-SP2) on Mon Oct 8 20:17:50 2018

By default Formality assumes that the isolation cells (and retention cells) will have been added to the netlist by this stage if it has come from DC – and so won't infer isolation cells from the UPF in implementation.

Question 7. (Optional) What is the other difference between the source UPF (top.upf) and post-DC UPF (top_postdc.upf) and how will this show up in Formality ?

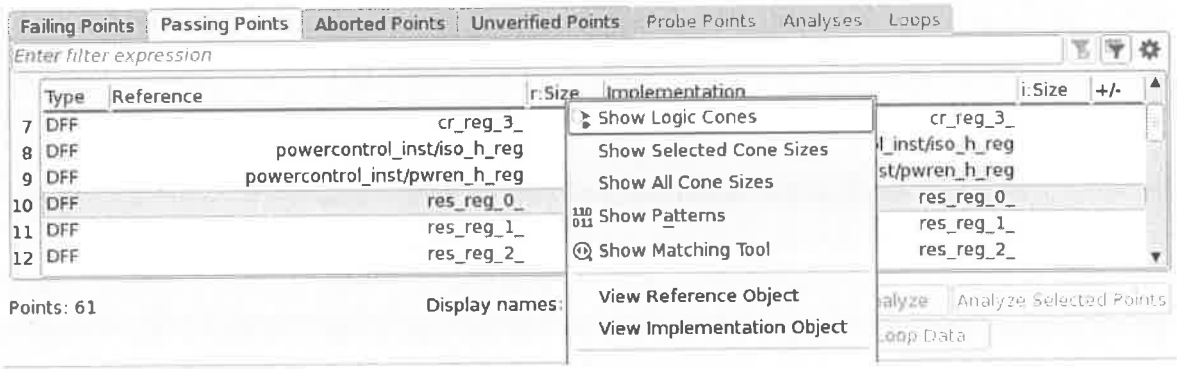
If you compare the 2 UPFs :

diff -b top.upf top_postdc.upf

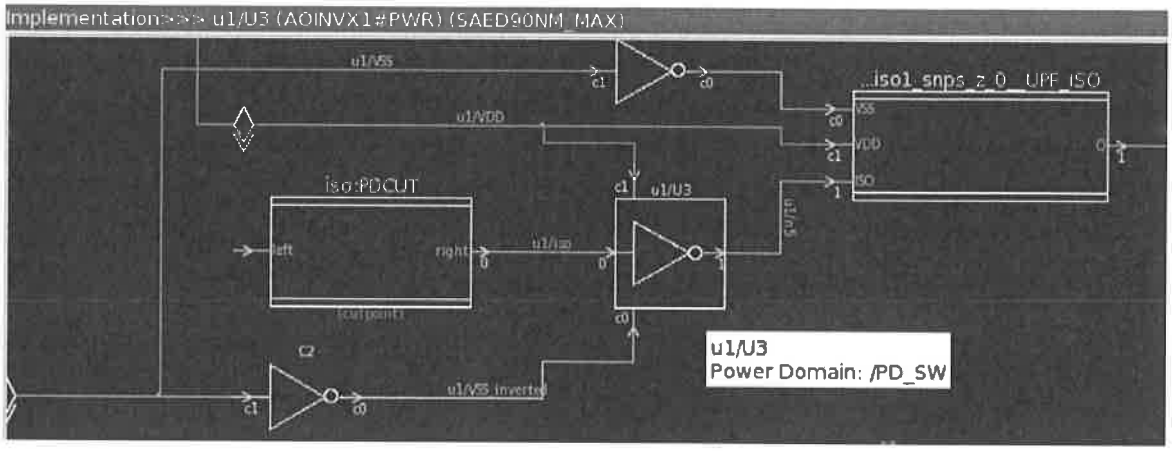
You will see the only substantive difference, apart from the comment header mentioned in answer to Question 6, is the lines :

set derived_upf true
#Design Compiler added commands
connect_supply_net VDD -ports { u1/U3/VDDG}
set derived_upf false

If you inspect the implementation logic cone for say res_reg_0_ (expanding the schematic as necessary)



Change the cell colouring to 'Power Domain' and 'Net Coloring to 'Supplies Always On'



You will see the u1/U3 is a buffer for the isolation control signal. It is connected to the top level power supply (VDD), rather that the switchable power supply VDD_SW, because of the connect_supply_net VDD -ports { u1/U3/VDDG}

statement in top_postdc.upf.

Note this exactly what you want DC to do, as you want the control signals of the isolation cells to be driven by non-powered down logic when the intent is to isolate.

6 Hard verification and the SVP flow

Learning Objectives

The lab will introduce you to the SVP and also the simplified verification flow

After completing this lab, you should be able to:

- Setup and run an SVP flow
- Setup and run a simplified verification mode flow

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

Task 1. Analyzing a INCONCLUSIVE session file

1. Go to the lab6

```
unix% cd lab6
```

2. The Formality run run1_fm.tcl has already been run with log file run1_fm.log and session file run1_fm.fss . Confirm from the log file that the verify was inconclusive with about 64 aborted points.

   (Note : For the purposes of the lab verification_effort_level is set 'low'. Normally one would set it to its default value 'high' – but having it low allows the lab to complete in a short time and illustrate the hard verificalion approaches.)

3. Restore the session files run1_fm.fss

```
unix% fm_shell -s run1_fm.fss
```

4. Run analyze_points -all

   Question 1. What SVP commands is Formality recommending ?

   .....

   Question 2. What other approaches does analyze_points mention as a possible next step?

   .....

5. Use report_aborted_points to get a list of the about 64 aborted points.

6. Quit out of Formality

Task 2. Re-running DC with the SVP commands

1. Cut-and-paste the SVP commands from Task 1 into run1_vp_dc.tcl
2. Run this updated DC script

```
unix% dc_shell -f run1_vp_dc.tcl |  tee -i run1_vp_dc.log
```

3. Observe the (OPT-774) message in the DC transcipt

   Information: 'u1/mult_31' will not be ungrouped because of the verification_priority attribute set on it. (OPT-774)

4. The synthesis run will take between 10 minutes.

Task 3. Re-running Formality with the SVP DC run

The script run1_vp_fm.tcl has already been written to verify the output of DC run from Task 1.

1. Run run1_vp_fm.tcl

```
unix% fm_shell -f run1_vp_fm.tcl |  tee -i run1_vp_fm.log
```

2. The Formality run will take about 20 minutes. When it has completed confirm that the verification now passes.

Task 4. (Optional) Simplified verification mode

The DC run , run1_sm_dc.tcl has been setup to run in simplified verification mode

1. Inspect run1_sm_dc.tcl

   Question 3. What is the key difference between this run and the original DC run, run1_dc.tcl ?

   .....

2. Run this DC script

```
unix% dc_shell -f run1_sm_dc.tcl |  tee -i run1_sm_dc.log
```

3. Observe the OPT-1701 messages

   Information: Keep the hierarchy u1/u1 that contains CRC logic. (OPT-1701)

   Information: Keep the hierarchy u2/u1 that contains CRC logic. (OPT-1701)

   Information: Keep the hierarchy u3/u1 that contains CRC logic. (OPT-1701)

   There are threes very large CRC blocks in the design and DC has automatically identified them and detuned the optimizations on these blocks (ie not ungrouping them as it did in the original run)

4. The synthesis run will take between 10 minutes.
5. When the synthesis run has completed confirm that the Formality can verify this run.

```
unix% fm_shell -f run1_sm_fm.tcl |  tee -i run1_sm_fm.log
```

Answers / Solutions

Task 1. Look

Question 1. What SVP commands is Formality recommending ?

```
current_design fred
set_verification_priority [ get_cells { u1/mult_31 u2/mult_31
u3/mult_31 } ]
current_design fred
```

Note the current_design commands here are redundant as fred is the top level of the design ie current_design is already fred.

Question 2. What other approaches does analyze_points mention as a possible next step?

a) Recompiling with CRC logic isolated

```
---------------------------------
Found 1 Potential CRC Module
---------------------------------
This module contains XOR trees/chains that may be contributing
to hard verifications.
Try reducing the complexity by placing XOR structures in
individual modules and ensure that Design Compiler is not
ungrouping them by adding the following to the compile script:
     set compile_isolate_crc_logic true
---------------------------------
r:/WORK/fred in file /global/gtseu01/gtseu6/fmtraining/ces/n/FM_2018.06/lab6/fred.v
   Contains 5784 chained 'XOR' cells
-----------
```

If you look in detail at the RTL you will see that is XOR logic followed by datapath, following by more XOR logic. The SVP commands are trying to make the datapath easier, the compile_isolate_crc_logic is trying to make the XOR logic easier to verify.

b) Alternate verification strategies

```
Compare points that are aborted or unverified (due to logic cone complexity)
may potentially be resolvable using alternative solver strategies. See the
man pages for verification_alternate_strategy and verification_auto_session.
Formality provides alternate strategy scripts at
/global/apps/fm_2018.06-SP2/auxx/fm/strategy.
Usage: s(n).sh [ -s session_file ] [ -f script_file ]
    Example:
     [unix] $ sh1.sh -s formality_auto.fss


1
```

Reloading the session file and setting the verification_alternate_strategy to one of the options in that particular Formality release is always a possible option for hard verification issues - and has the advantage that one doesn't have to re-synthesize. If one has the option to recompile in Design Compiler then in this particular example SVP commands or isolation the CRC logic is the more likely to be successful.

Question 3. What is the key difference between this run and the original DC run, run1_dc.tcl ?

set simplified_verification_mode true

This page intentionally left blank.

7

Debugging Approaches

Learning Objectives

This lab will introduce you on a simple design the various clues that Formality gives you about whether a design will fail or why it has failed.

After completing this lab, you should be able to

- Debug simple synthesized designs
- Understand the clues available at the match stage

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

A verification script has been provided to run the Formality verification. The results will then need to be debugged.

Task 1. Run the verification script

1. Run Formality

```
unix% cd lab7
unix% fm_shell -f runme.fms |& tee -i runme.log
```

Question 1. Did the verification pass ?

.....

Question 2. If not, how many points failed ?

.....

Question 3. Were there any compare points left unverified and if so why ?

.....

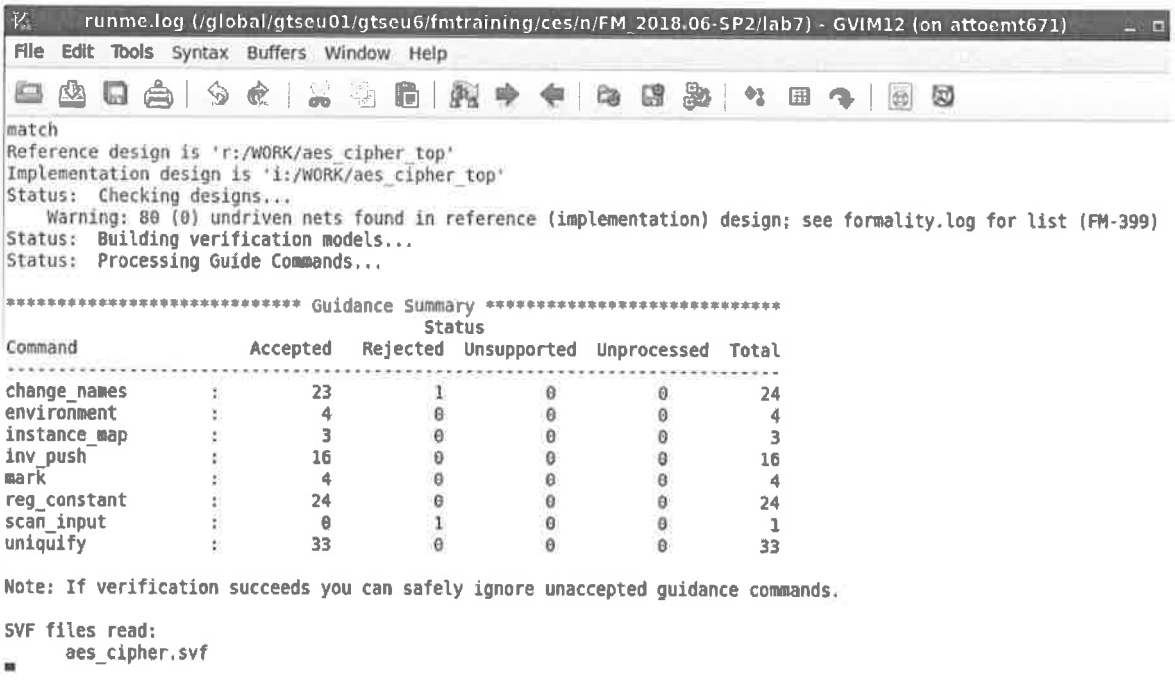
Task 2. Transcript debugging

The transcript is always the first place to start debugging

Question 4. Name two key areas in the transcript to look for clues ?

.....

1. Look at the SVF summary



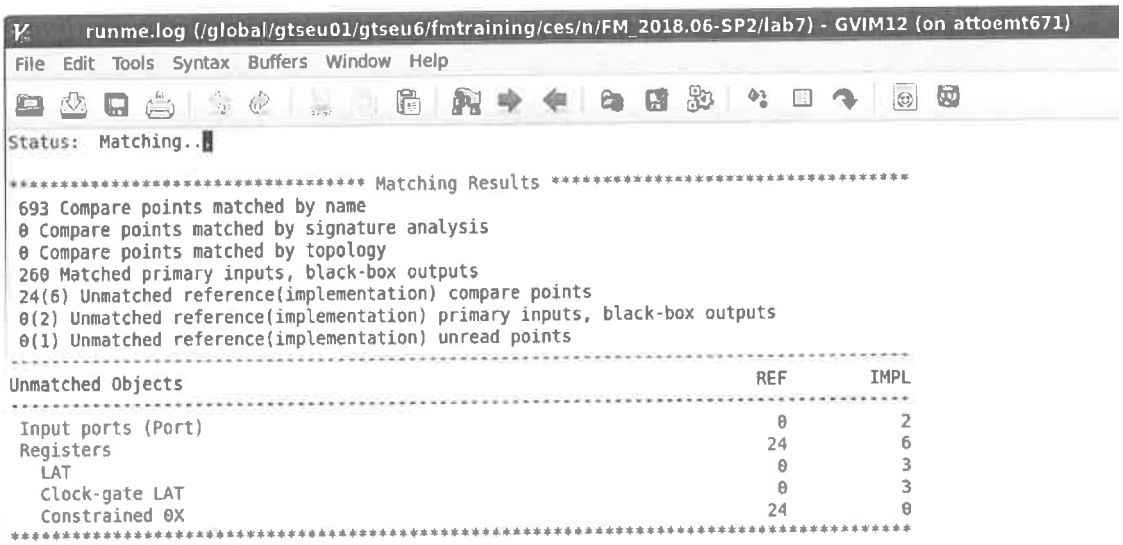
Question 5. What in the SVF are rejected ?

.....

Question 6. Which of the rejected commands is likely to lead to verification failure ? (Hint: Use the Job Aid or presentation material)

.....

2. Look at the matching summary



Clock-gating latches if introduced during synthesis will be unmatched, that is they are expected to be unmatched.

Question 7. What is unmatched ?

.....

Question 8. If something is unmatched what related points could fail ?

.....

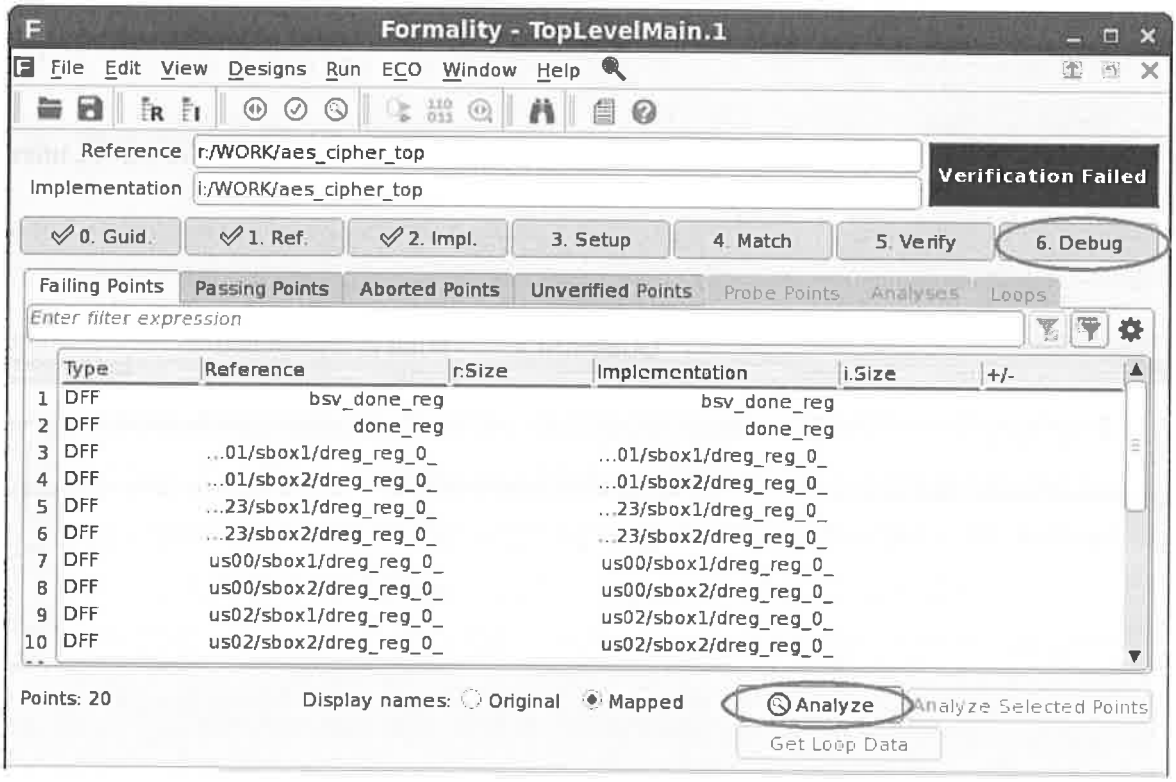
Task 3. Debugging in Formality GUI

From Task 2 and looking at the transcript we have built up some clues as to why we are getting failing verification. The next stage is to see what Formality thinks the reason for failure.

1. Restore the session file saved in the script after verify has been run. You can use the command **restore_session** or the –s switch :

```
unix% fm_shell -gui -s verify.fss |& tee -i debug.log
```

2. Within the Debug tab run analyze



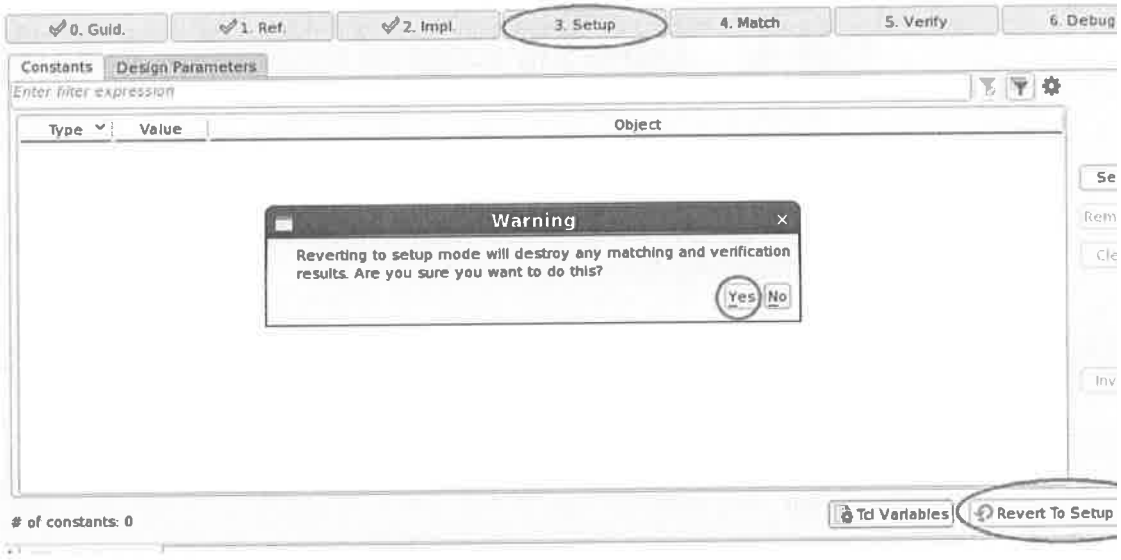
Question 9. Does one of the causes of failure identified , account for all 20 failing points, and is consistent with the SVF and match summary?

.....

Question 10. What is the suggested solution ?

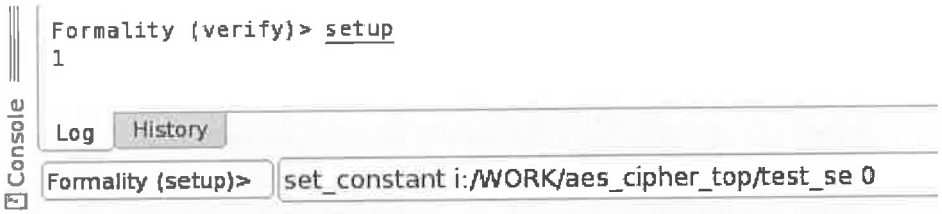
.....

3. Return to setup mode. (One can only apply constants in setup mode) Either from the GUI :



Or just type the command **setup** at the command prompt

4. Apply the **set_constant** command suggested by the analyze_points command



5. Run the match command

Question 11. How has the match summary changed ?

.....

6. Run the verify step

Question 12. Has the verification now succeeded ?

.....

If yes quit out of Formality .

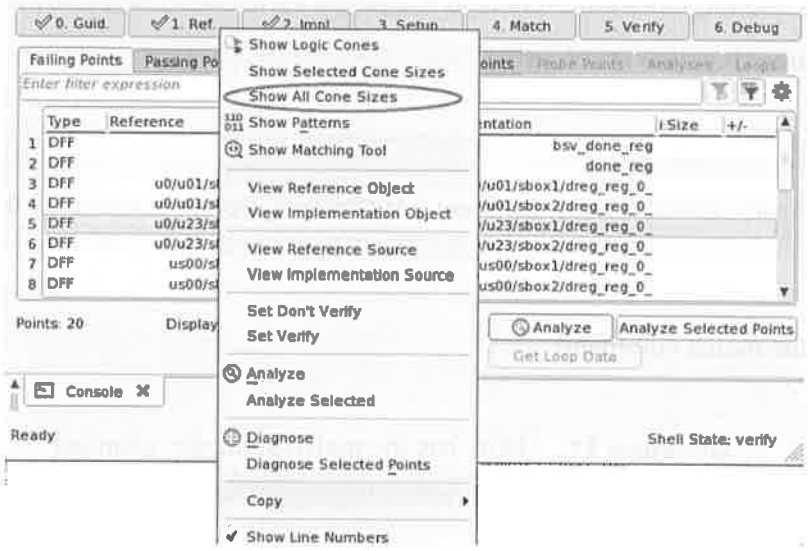
Task 4. Exploring some of debugging features

From Task 3 we know what the problem is with the initial failing verification. Let us have a look at how that problem will look in some of the other debugging features in the GUI.

1. Reload the failing verification run

```
unix% fm_shell -gui -s verify.fss |& tee -i debug2.log
```

2. Go to the failing point list. Typically you want to debug the small size cone first. Right mouse button to show all cone sizes

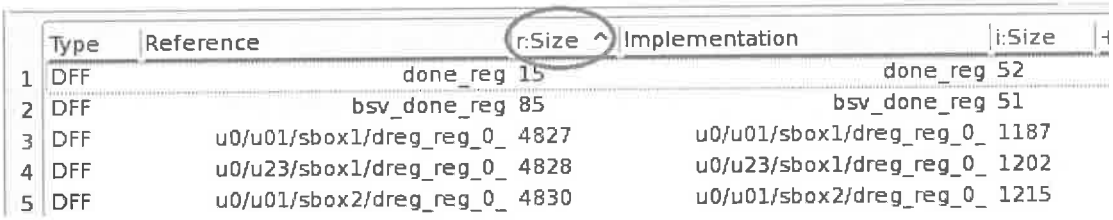


3. The cone sizes are not huge. But lets try for a smaller one. Revert to the shell and run for another 400 failing points :

set verification_failing_point_limit 400

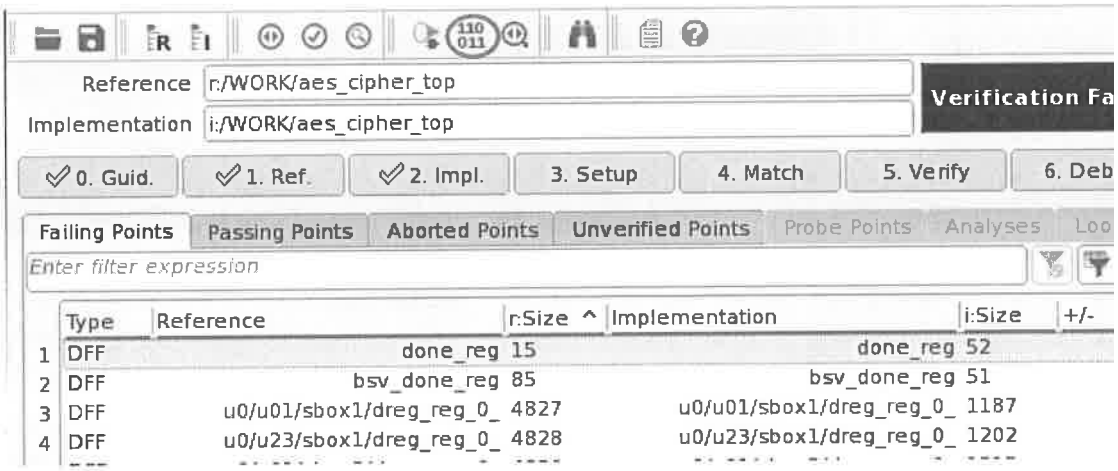
verify

4. Repeat the 'Show All Cone Sizes' step. Click on 'Size' to order by size.



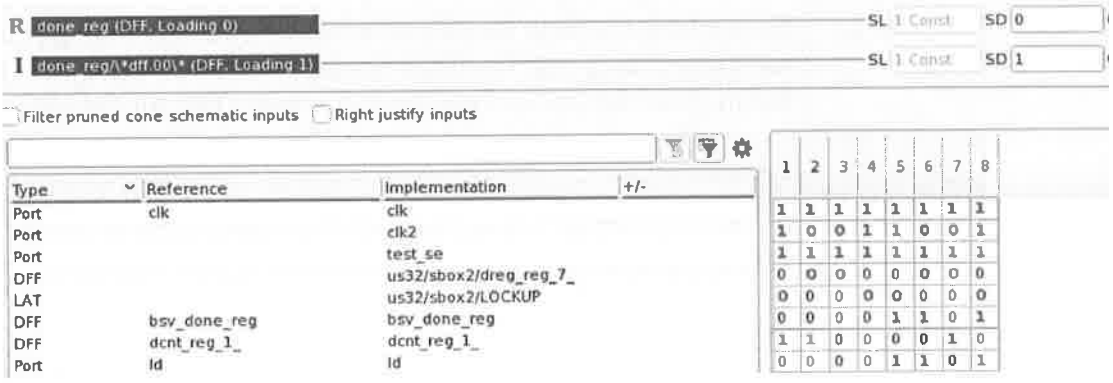
	Type	Reference	r:Size	Implementation	i:Size
1	DFF	done_reg	15	done_reg	52
2	DFF	bsv_done_reg	85	bsv_done_reg	51
3	DFF	u0/u01/sbox1/dreg_reg_0_	4827	u0/u01/sbox1/dreg_reg_0_	1187
4	DFF	u0/u23/sbox1/dreg_reg_0_	4828	u0/u23/sbox1/dreg_reg_0_	1202
5	DFF	u0/u01/sbox2/dreg_reg_0_	4830	u0/u01/sbox2/dreg_reg_0_	1215

5. Select the smallest size cone register (done_reg) and pull up the pattern window



	Type	Reference	r:Size	Implementation	i:Size	+/-
1	DFF	done_reg	15	done_reg	52	
2	DFF	bsv_done_reg	85	bsv_done_reg	51	
3	DFF	u0/u01/sbox1/dreg_reg_0_	4827	u0/u01/sbox1/dreg_reg_0_	1187	
4	DFF	u0/u23/sbox1/dreg_reg_0_	4828	u0/u23/sbox1/dreg_reg_0_	1202	

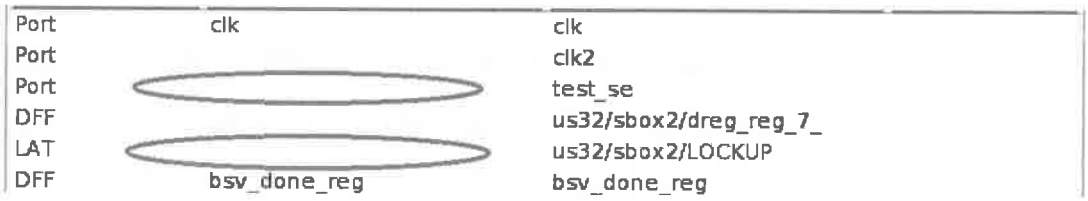
6. Some of the notable information that is visible :



Type	Reference	Implementation	+/-
Port	clk	clk	
Port		clk2	
Port		test_se	
DFF		us32/sbox2/dreg_reg_7_	
LAT		us32/sbox2/LOCKUP	
DFF	bsv_done_reg	bsv_done_reg	
DFF	dcnt_reg_1_	dcnt_reg_1_	
Port	id	id	

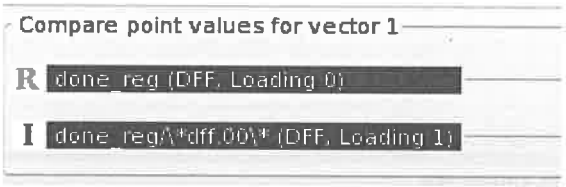
1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1
1	0	0	1	1	0	0	1
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	1
1	1	0	0	0	0	1	0
0	0	0	0	1	1	0	1

The some inputs to Impl logic cone are not in the Ref logic cone in particular the port test_se and the latch us32/sbox2/LOCKUP:



There are several failing vectors, but they all have input value for test_se ias 1.

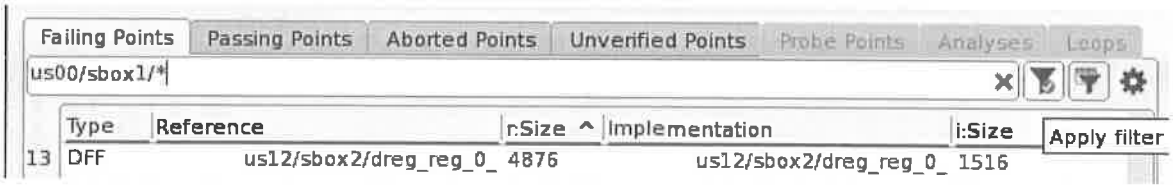
For all failing vectors , the value loaded by the reference is 0 , the value loaded in the implementation is 1

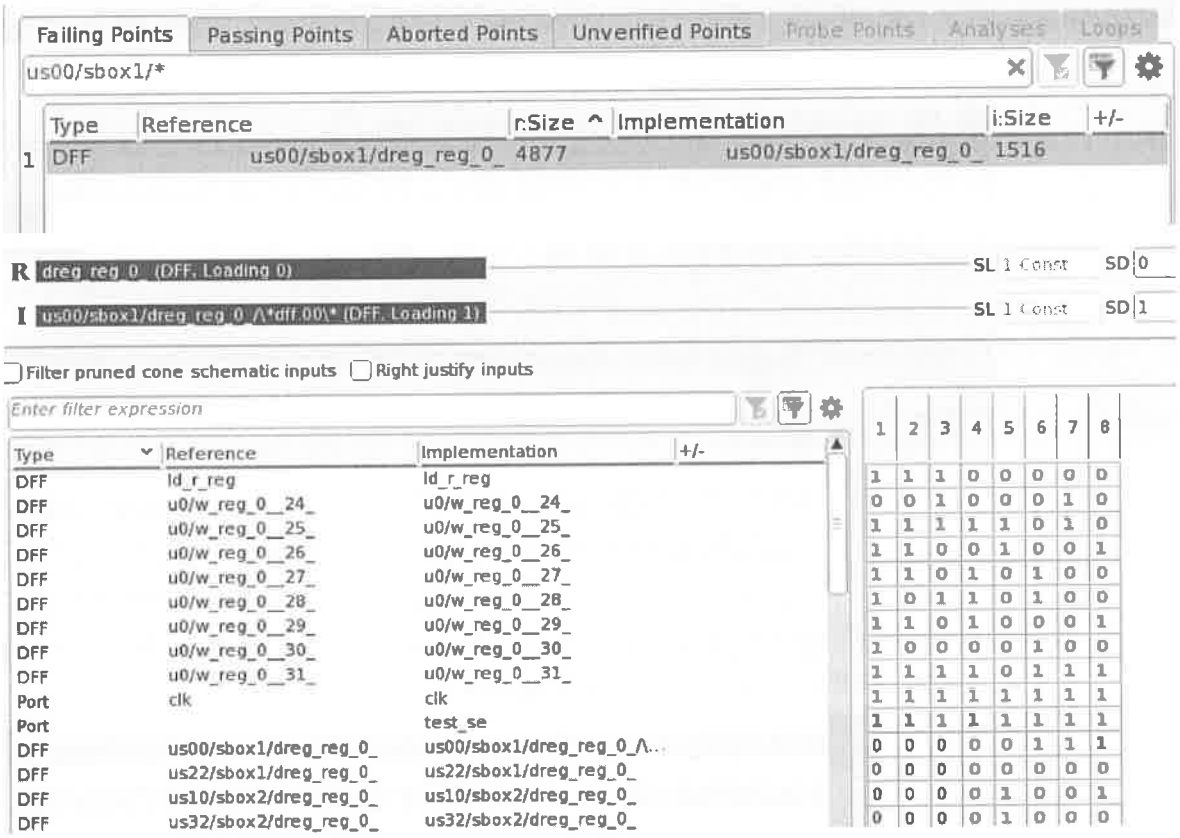


Question 13. Is the pattern window results consistent with a failing cause of the test scan enable needs to be deasserted ?

.....

7. Have a look a look in the pattern window at another failing point – say us00/sbox1/dreg_reg_0_. If not obvious where us00/sbox1/dreg_reg_0_is in failing point list one can use the fiter to say filter for the pattern us00/sbox1*





There are 1 unmatched points in the implementation – test_

Question 14. Again is the pattern window results consistent with a failing cause of the test scan enable needs to be de-asserted ?

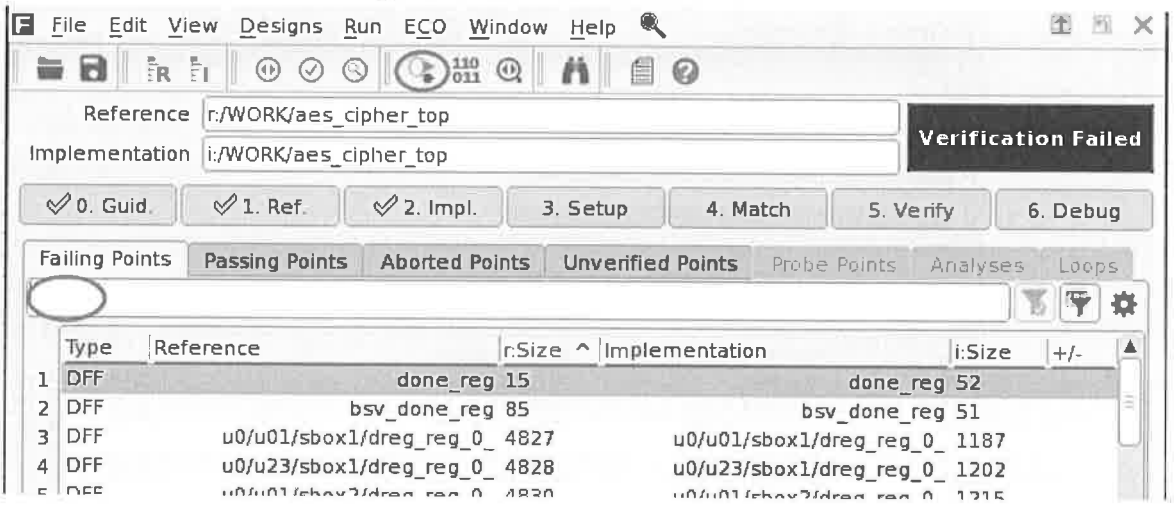
.....

8. (Optional) Coming back to the first failing point done_reg which has unmatched inputs

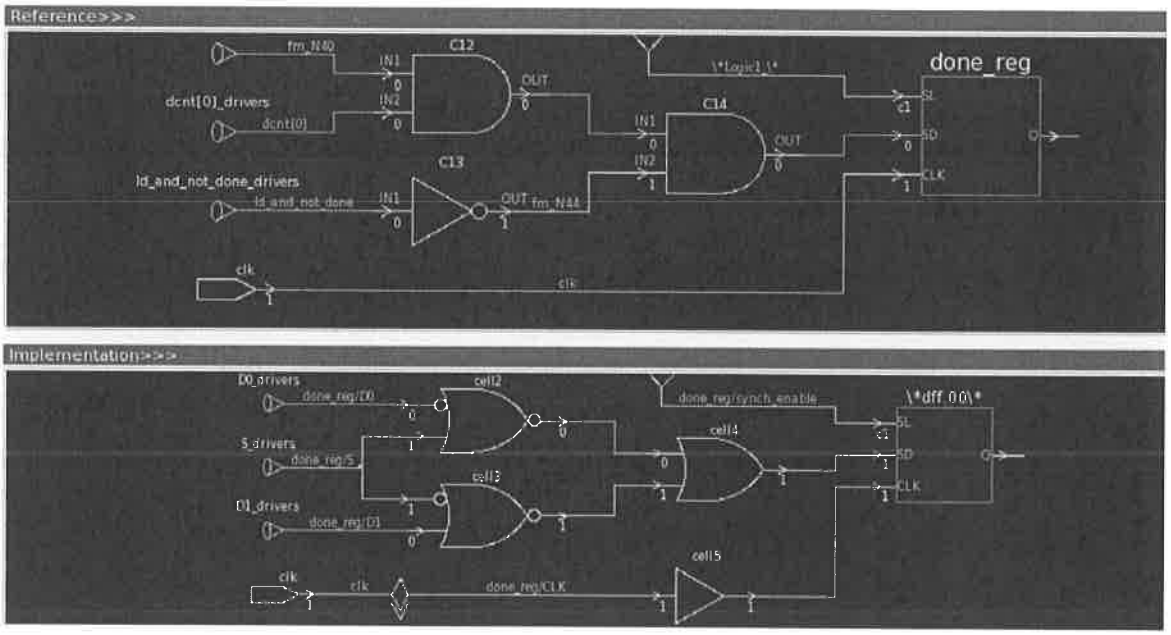
Question 15. How might us32/sbox2/LOCKUP and us32/sbox2/dreg_reg_7 be related to the failing point done_reg.

.....

Remove the filter, select done_reg and pull up the logic coen

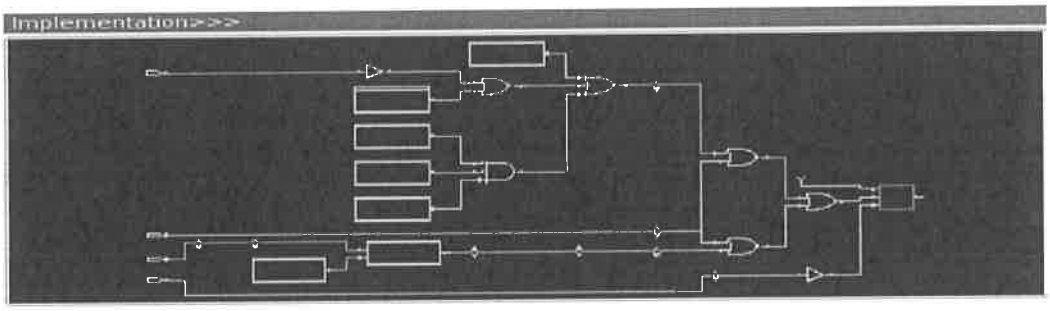


The schematic should look something like the below

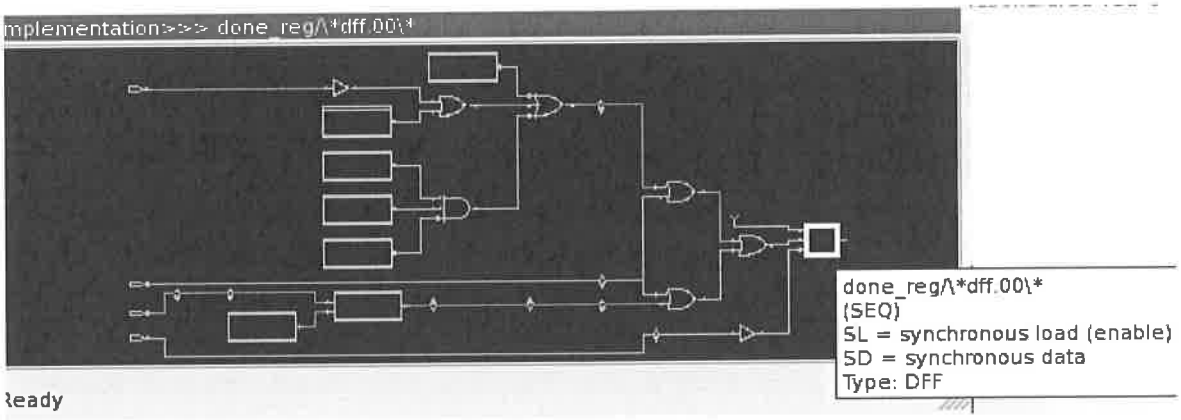


By default the technology cell for done_reg in the implementation has been flattened. Let us group it into one level of hierarchy.

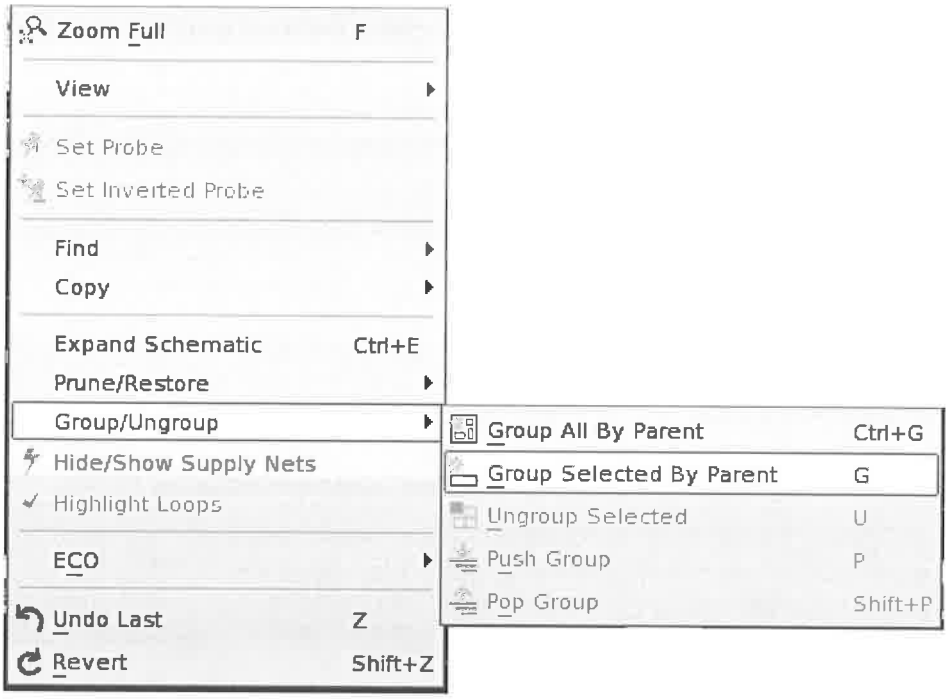
Expand the implementation schematic (Ctrl+E) or :



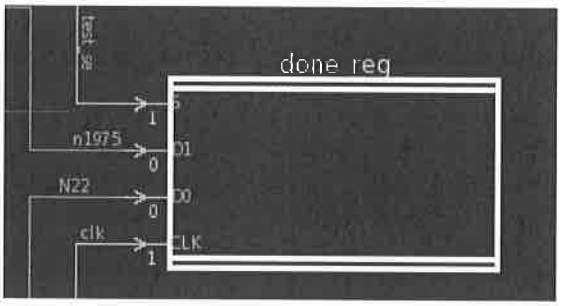
Select the done_reg/*dff.00* (this the memory element iside the STD cell)



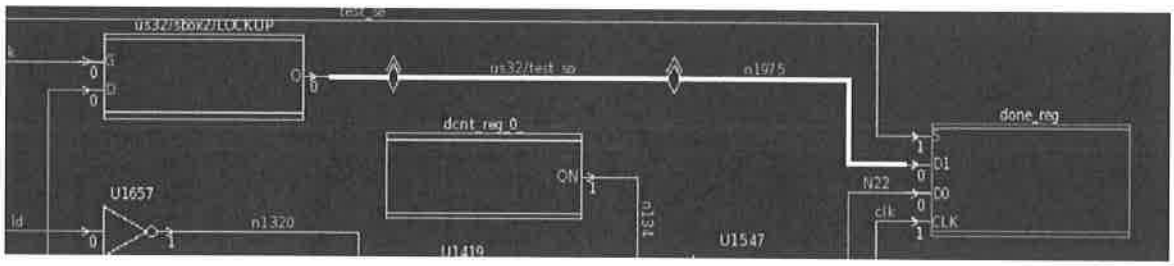
Group the selected cell (Group/Ungroup -> Group Selected By Parent)



It is now clearer.

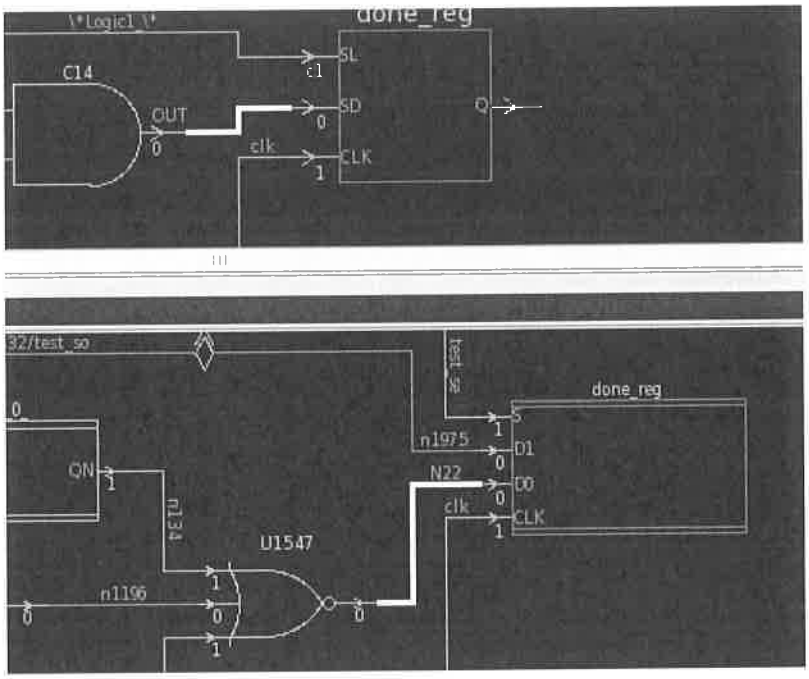


The S of done_reg is connected to test_se and the D1 pin to the output of LOCKUP latch

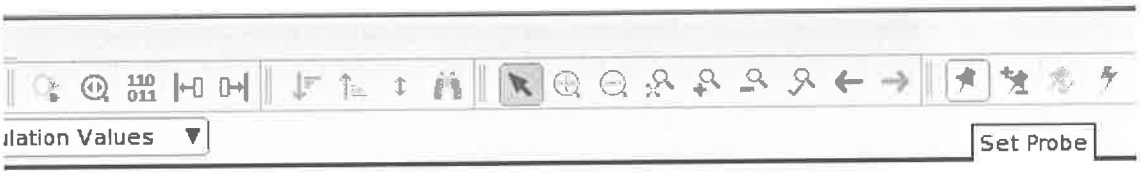


. That would be consistent with S being a scan-enable pin and D1 the scan in pin. So if our hypothesis is correct the net on D0 should be equivalent to the SD pin in the reference.

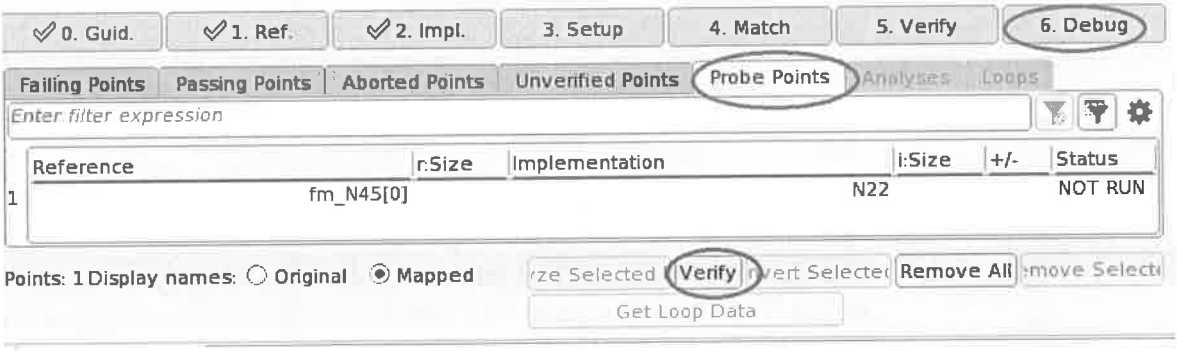
Select those 2 nets



With those nets highligted **set_probe**



Go to the probe points tab and run verify probe.



The probe point verification should pass. That is the non-scan D input pin of the implementation flip-flop is the same functionality as the SD pin in the reference.

	Reference	r:Size	Implementation	i:Size	+/-	Status
1	fm_N45[0]		N22			PASS

Quit out of Formality.

Task 5. Fully debugging at the match stage

Often one finds a way of getting verification to pass after running and initially failing verification. And one of the purposes of this lab was to show that 'analyze' could do this. However a more efficient way of doing this, with a little experience, is to trap the issues, completely at the match stage

1. Reload the session from match stage from the original failing verification run. If you examine runme.fms you will see this is called match.fss

```
unix% fm_shell -s match.fss |& tee -i match.log
```

2. Either from the original transcript or using **report_guidance –summary** examinine the SVF summary
3. Use the **report_svf_operation** command to examine the rejected scan_input command

Question 16. Why was the scan_input command rejected ?

.....

4. Use the **report_unmatched_points** command to identify the unmatched points. Note there are some very useful switches on this command that narrow down what the command reports

Question 17. What are the unmatched input ports?

.....

Question 18. What are the unmatched latches ?

.....

5. Copy the runme.fms to runme_mod.fms to add synopsys_auto_setup to be true at start of script and rerun the the verification.

   The verification should now succeed.

Question 19. With'synopsys_auto_setup true' does any additional information appear in the transcript referring to aes_cipher_top/test_se ?

.....

Answers / Solutions

Task 1. Run the verification script

Question 1. Did the verification pass ?

No. In the transcript it has 'Verification FAILED'

Question 2. How many points failed ?

20

Question 3. Were there any compare points left unverified and if so why?

544 . The failing point limit (verification_failing_point_limit) is 20 by default. That is the number of failing points after which Formality will stop verifying.

You can do, for example, in Formality **printvar *limit*** to confirm this.

Task 2. Transcript debugging

Question 4. Name two key areas in the transcript to look for clues ?

SVF summary

Matching summary

Question 5. What in the SVF are rejected?

1 change_names command and 1 scan_input

Question 6. Which of the rejected commands is likely to lead to verification failure?

Scan_input

Question 7. What is unmatched?

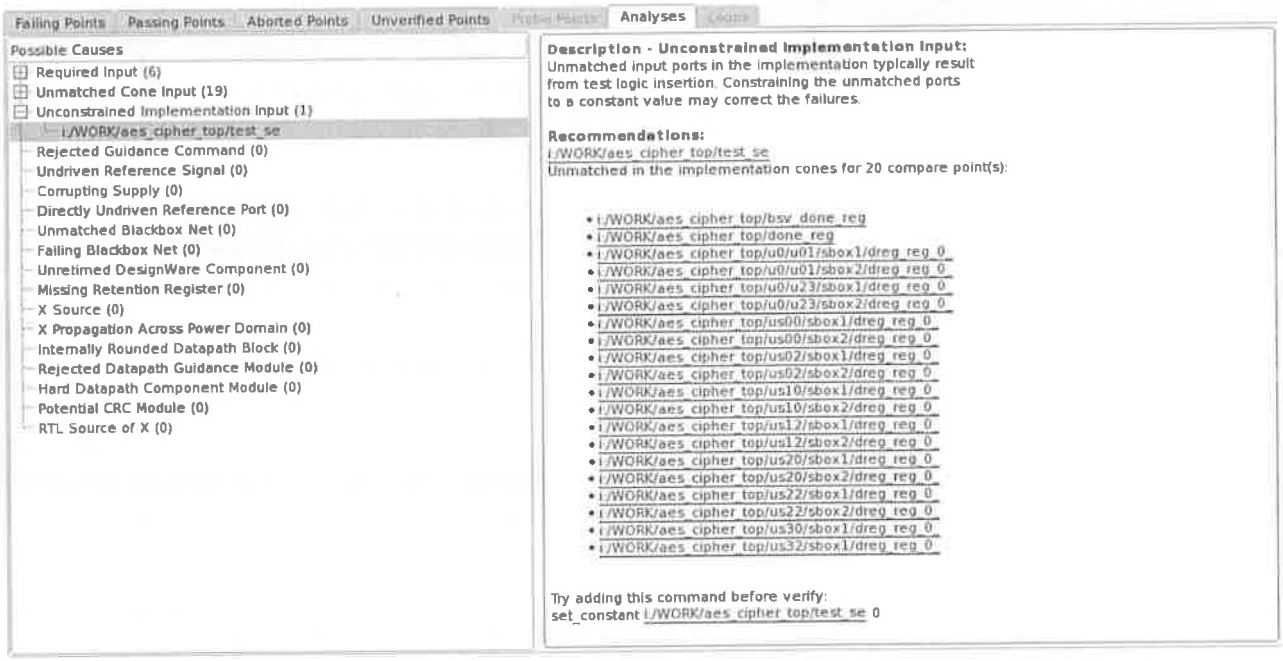
2 input ports

3 latches

Question 8. If something is unmatched what related points could fail?

The matched points that have the unmatched points as inputs.

Task 3. Debugging in Formality GUI



Question 9. Does one of the causes of failure identified, account for all 20 failing points, and is consistent with the SVF and match summary?

Yes. test_se has been identified as an unconstrained input. If it is to do with test logic insertion then de-asserting what looks like a scan enable port would be a way of getting the verification to pass.

Question 10. What in the suggested solution?

set_constant i:/WORK/aes_cipher_top/test_se 0

Question 11. How has the match summary changed?

```
************************************ Matching Results ************************************
693 Compare points matched by name
0 Compare points matched by signature analysis
0 Compare points matched by topology
260 Matched primary inputs, black-box outputs
24(3) Unmatched reference(implementation) compare points
0(1) Unmatched reference(implementation) primary inputs, black-box outputs
0(5) Unmatched reference(implementation) unread points
------------------------------------------------------------------------------------------
Unmatched Objects                                                    REF          IMPL
------------------------------------------------------------------------------------------
Input ports (Port)                                                     0             1
Registers                                                             24             3
  Clock-gate LAT                                                       0             3
  Constrained 0X                                                      24             0
******************************************************************************************
```

1 of the unmatched input ports and 3 LAT have gone from the Unmatched Objects section. And the number of unmatched implementation unread points has increased by 4.

Question 12. Has the verification now succeeded?

Yes

Task 4. Exploring some debugging features

Question 13. Is the pattern window results consistent with a failing cause of the test scan enable needs to be deasserted ?

Yes. The failing vector is with test_se as 1.

Question 14. Again is the pattern window results consistent with a failing cause of the test scan enable needs to be deasserted ?

Yes. The failing vector is with test_se as 1.

Question 15. How might us32/sbox2/LOCKUP and us32/sbox2/dreg_reg_7 be related to the failing point done_reg.

Given it is unmatched in the reference dreg_reg_7 could be connected to the LOCKUP latch to the scan-in pin of done_reg

Task 5. Fully debugging the match stage

Question 17. Why was the scan_input command rejected?

The scan_input command will only be accepted if the variable synopsys_auto_setup is true. It was false in the initial verification run.

report_svf_operation –status rejected –command scan_input

```
fm_shell (match)> report_svf_operation -status rejected -command scan_input

## SVF Operation 90 (Line: 478) - scan_input.  Status: rejected
## Operation Id: 90
guide_scan_input \
  -design { aes_cipher_top } \
  -disable_value 0 \
  -ports { test_se }

Info: guide_scan_input 90 (Line: 478)  synopsys_auto_setup is false, ignoring Scan Input guidance.
```

Question 18. What are the unmatched input ports?

test_se , test_si

```
fm_shell (match)> report_unmatched_points -point_type input
2 Unmatched points (0 reference, 2 implementation):
  Impl Port       i:/WORK/aes_cipher_top/test_se
  Impl Port       i:/WORK/aes_cipher_top/test_si
```

Question 19. What are the unmatched latches ?

```
fm_shell (match)> report_unmatched_points -point_type LAT -except_status  clock_gate
3 Unmatched points (0 reference, 3 implementation):
  Impl LAT        i:/WORK/aes_cipher_top/LOCKUP
  Impl LAT        i:/WORK/aes_cipher_top/us12/sbox2/LOCKUP
  Impl LAT        i:/WORK/aes_cipher_top/us32/sbox2/LOCKUP
```

That is LOCKUP latches introduced during test insertion.(LOCKUP latches are often used to avoid hold violations when scanning between different clock domains etc..)

Question 20. With'synopsys_auto_setup true' does any additional information appear in the transctipt referring to aes_cipher_top/test_se ?

During the match stage where processing the SVF Formality echos out the constant applied.

```
match
Reference design is 'r:/WORK/aes_cipher_top'
Implementation design is 'i:/WORK/aes_cipher_top'
Status:  Checking designs...
    Warning: 80 (0) undriven nets found in reference (implementation) design; see formality.log for list (FM-399)
Status:  Building verification models...
Status:  Processing Guide Commands...
    Set 'i:/WORK/aes_cipher_top/test_se' to constant 0

********************  Guidance Summary  ********************
                                    Status
Command                 Accepted  Rejected  Unsupported  Unprocessed  Total
-------------------------------------------------------------------------
change_names        :         23         1            0            0     24
environment         :          4         0            0            0      4
instance_map        :          3         0            0            0      3
inv_push            :         16         0            0            0     16
mark                :          4         0            0            0      4
reg_constant        :         24         0            0            0     24
scan_input          :          1         0            0            0      1
uniquify            :         33         0            0            0     33
```

You can also see in the Guidance Summary that the scan_input command (ie the the setting of the constant) has been accepted,

8 RTL and Netlist Interpretation

Learning Objectives

This lab will address some of the debugging involved reading RTL and netlists

After completing this lab, you should be able to:

- Debug black-box issues
- Debug synthesis RTL pragma issues

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

In this lab we will be looking at 2 typical problems you see reading in RTL

Task 1. Debugging black boxes

1. Go to lab8a

```
unix% cd lab8/lab8a
```

2. Run the existing Formality script

```
unix% fm_shell -f runme.fms |& tee -i runme.log
```

3. Check the transcript to see that the verification has failed.
4. As always check the SVF summary and the match summary

Question 1. Is there anything in the SVF that will of itself cause a failure ?

.....

Question 2. Is there anything in the matching summary that is suspicious or will cause a failure ?

.....

5. Do further checking of the transcript .

Question 3. What warning messages relate to black boxes before 'match'?

.....

Question 4. If you do a man on one of the warning messages FE-LINK-2 what variable does it refer that controls how missing modules are treated ? And what is this variable set to in the script ?

.....

6. Restore the session file for the match stage (match.fss)

```
unix% fm_shell -s match.fss
```

7. Run the command **report_setup_status**

Question 5. Is the result of **report_setup_status** consistent with the warning messages in transcript ?

.....

8. Use the commands **report_black_boxes** and **report_unmatched_points** to confirm the details of the missing module. (Hint: There are useful switches to report_unmatched_points:

report_unmatched_points –point_type bbox

report_unmatched_points –point_type bbox_pin

report_unmatched_points –point_type bbox_input

report_unmatched_points –point_type bbox_output

)

9. Take a copy of runme.fms to runme_mod.fms. Change the variable back to it's default value **Error**.

```
set search_path ". ./rtl ./libs"
read_db -tech  libs/tc6a_cbacore.db

set_svf mR4000.svf

#set hdlin_unresolved_modules black_box
set hdlin_unresolved_modules  Error

read_verilog -r  "   \
        cntrl.v          \
        register.v       \
        r4000.v          \
"
set_top r:/WORK/mR4000

read_verilog -i  mR4000.gates.v
set_top i:/WORK/mR4000
```

Confirm that the design now fails to elaborate.

```
unix% fm_shell -f runme_mod.fms |&  tee -i runme_mod.log
```

10. Given we have established that the RTL is missing the module for 'alu' further modify the runme_mod.fms to allow the design to elaborate. (Hint: Which file in ./rtl are we not reading in to the reference container?)

Re-run the script to check that the RTL now elaborates and that the verification now passes.

Notice that there now no warnings; the SVF is accepted; and the matching is clean. That is the missing RTL file was the root cause of all the issues in the transcript.

Note: The important point: the warnings FE-LINK-2 appears earlier and is easier to debug than the later match summary. If we had waited until verify and analyze_points we would have got there in the end – but the runtime would have been longer and the debug time would have longer. It would have been easy post verify to hypothesize the issue was something else than unmatched black-box and wasted time testing out that hypothesis. Most debugging issues are simple and most simple issues are easy to debug – so assume you are looking for something simple first. Sermon over.

Task 2. Debugging RTL pragmas

The verification in this lab will do RTL to gates verification where the RTL has synthesis pragmas

1. Go to lab8b

```
unix% cd lab8/lab8b
```

2. Run the existing Formality script . The verification will fail.

```
unix% fm_shell -f runme.fms |&  tee -i runme.log
```

Question 6. Check the transcript. Is there anything in the SVF summary or match summary that indicates a problem ?

.....

3. Find in the transcript the following section

```
************ RTL Interpretation Summary ************
************ Design: r:/WORK/mR4000
full_case ignored (7 total, 1 with unspecified cases)
parallel_case ignored (7 total, 1 with overlapping cases)

Please refer to the Formality log file for more details,
or execute report_hdlin_mismatches.
*************************************************

Reference design set to 'r:/WORK/mR4000'
```

Do what it suggests and add in the command **report_hdlin_mismatches** after **set_top** on the reference, and run this in Formality.

```
set search_path ". ./libs ./rtl"

set_mismatch_message_filter -warn "FMR_ELAB-115
FMR_ELAB-116"

set_svf default.svf

read_db tc6a_cbacore.db
read_verilog -r "alu.v cntrl.v r4000.v
register.v"
set_top mR4000
report_hdlin_mismatches
```

Question 7. Where does **report_hdlin_mismatches**report where there could be a synthesis simulation mismatch in the RTL code ?

.....

Question 8. What does **report_hdlin_mismatches** say would give the same interpretation of the RTL as Design Compiler ?

.....

4. Inspect the RTL in the area suggested by report_hdlin_mismatches (Hint. The answer to Question 7).

Question 9. For what register states would the pragmas not be justified ?

.....

5. Let us assume that the RTL can not get into a state outside of the what the pragmas suggest. Modify the runme.fms to add in

   set hdlin_ignore_full_case false

   set hdlin_ignore_parallel_case false

   Note : The other required settings:

   set hdl set_mismatch_message_filter -warn \
   "FMR_ELAB-115 FMR_ELAB-116""

   were already in the script.

6. Run the modified Formality script. Verification should now pass.

Answers / Solutions

Task 1. Debugging black boxes

Question 1. Is there anything in the SVF that will of itself cause a failure?

No. There are some rejected change_names, datapath, merge and replace commands – but nothing that will of themselves cause a failure.

Question 2. Is there anything in the matching summary that is suspicious or will cause a failure?

```
Status: Matching...

********************************** Matching Results **********************************
 348 Compare points matched by name
 0 Compare points matched by signature analysis
 0 Compare points matched by topology
 34 Matched primary inputs, black-box outputs
 117(0) Unmatched reference(implementation) compare points
 33(0) Unmatched reference(implementation) primary inputs, black-box outputs
--------------------------------------------------------------------------------
Unmatched Objects                                                  REF       IMPL
--------------------------------------------------------------------------------
 Black-boxes (BBox)                                                  1          0
 Black-box input pins (BBPin)                                      117          0
 Black-box output pins (BBPin)                                      33          0
********************************************************************************
```

Yes. Unmatched black box objects in the reference.

Question 3. What warning messages relate to black boxes before 'match'?

```
set_top r:/WORK/mR4000
Setting top design to 'r:/WORK/mR4000'
Status:   Elaborating design mR4000   ...
Status:   Elaborating design mCntrl   ...
Warning: Cannot link cell '/WORK/mR4000/alu' to its reference design 'mAlu'. (FE-LINK-2)
Status:   Elaborating design mRegister   ...
Status:  Implementing inferred operators...
Status:  Creating black-box designs...
Created technology library 'FM_BBOX' in container 'r' for black-box designs
Created black-box design 'mAlu' in library 'FM_BBOX'
Warning: 1 blackbox designs were created for missing references. (FM-064)
Status:  Attempting to resolve unlinked cells by using black-boxes...
Warning: 150 black-box pins of unknown direction found; see formality.log for list (FM-2
Top design set to 'r:/WORK/mR4000' with warnings
Reference design set to 'r:/WORK/mR4000'
```

There are several: FE-LINK-2 , FM-064, and FM-230

Question 4. If you do a **man** on one of the warning messages FE-LINK-2 what variable does it refer that controls how missing modules are treated? And what is this variable set to in the script?

set hdlin_unresolved_modules black_box

Question 5. If the result of **report_setup_status** consistent with the warning messages in transcript ?

Yes. Observe there is one unresolved module in the Ref

Ports:	132(132)
Registers:	250(250)
Black boxes:	1(0)
- Unresolved modules :	1(0)
- User specified :	0(0)

Task 2. Debugging RTL pragmas

Question 6. Check the transcript. Is there anything in the SVF summary or match summary that indicates a problem?

Is this case: no. Both the SVF and the match summary look clean.

Question 7. Where does **report_hdlin_mismatches** report where there could be a synthesis simulation mismatch in the RTL code?

```
************ RTL Interpretation Results ************
************ Design: r:/WORK/mR4000
full_case ignored (7 total, 1 with unspecified cases)
Formality has been directed to ignore the directive on
these occurrences, which may lead to false differences:
    File: /global/gtseu01/gtseu6/fmtraining/ces/n/FM_2018.06-SP2/lab8/lab8b/rtl/r4000.v, Line: 223
If you wish to interpret these (potentially disagreeing
with a logic simulator), update the following variables
and reload the design.
set hdlin_ignore_full_case false
lappend hdlin_warn_on_mismatch_message FMR_ELAB-115

parallel_case ignored (7 total, 1 with overlapping cases)
Formality has been directed to ignore the directive on
these occurrences, which may lead to false differences:
    File: /global/gtseu01/gtseu6/fmtraining/ces/n/FM_2018.06-SP2/lab8/lab8b/rtl/cntrl.v, Line: 111
If you wish to interpret these (potentially disagreeing
with a logic simulator), update the following variables
and reload the design.
set hdlin_ignore_parallel_case false
lappend hdlin_warn_on_mismatch_message FMR_ELAB-116
************************************************************
```

Line 223 of rtl/r4000.v and line 111 of rtl/cntrl.v

Question 8. What does report_hdlin_mismatches say would give the same interpretation of the RTL as Design Compiler?

set hdlin_ignore_full_case false

set hdlin_ignore_parallel_case false

Question 9. For what register states would the pragmas not be justified ?

PCSoucce 2'11

```
case (PCSource) // synopsys parallel_case full_case
    2'b00: PC = ALU_result;
    2'b01: PC = TargetReg;
    2'b10: PC = {2'b00,PC[31:28],Instruction[25:0]};
Endcase
```

```
        pTestInstFetch  = 11'bxxxxxxxxxx1, pTestInstDcode  = 11'bxxxxxxxxx1x,
        pTestMemAddr    = 11'bxxxxxxxx1xx,pTestLwMemAcss  = 11'bxxxxxxx1xxx,
        pTestWrtBack    = 11'bxxxxxx1xxxx,, pTestSwMemAcss  = 11'bxxxxx1xxxxx,
        pTestRTypeExe   = 11'bxxxx1xxxxxx,, pTestRTypeComp  = 11'bxxx1xxxxxxx,
        pTestBranch     = 11'bxx1xxxxxxxx,  pTestBranchComp = 11'bx1xxxxxxxxx,
        pTestJumpComp   = 11'b1xxxxxxxxxx,
        reg [10:0] state;
        casex (state)  // synopsys parallel_case full_case
        pTestInstFetch: begin
         pTestInstDcode: begin
        pTestMemAddr: begin
        pTestLwMemAcss: begin
pTestWrtBack:  begin
 pTestSwMemAcss: begin
pTestRTypeExe: begin
         pTestRTypeComp: begin
         pTestBranch: begin
pTestBranchComp: begin
         pTestJumpComp: begin
default: begin
```

Many states for which parallel_case is not valid if those states are reachable.

9 Sequential Design Transforms and SVF

Learning Objectives

In this lab you will debug a simple issue with register optimizations

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

In this lab we will running on a typical problem with sequential optimizations

Task 1. Constant register debug

1. Go to lab9a

```
unix% cd lab9/lab9a
```

2. Run the existing Formality script

```
unix% fm_shell -f runme.fms | tee -i runme.log
```

3. Check the transcript to see that the verification has failed.
4. As always check the SVF summary and the match summary

Question 1. Is there anything in particular rejected in the SVF guidance summary that could cause a failing verification ?

.....

Question 2. Is there anything in the match sumary that is consistent with the answer to Question 1.?

.....

Question 3. Given answers to questions 1) and 2) could you have predicted the results of the verify step without running verify ?

.....

5. Run

report_svf_operation –status rejected –command reg_constant

Question 4. What is the reason for the constant being rejected ?

.....

6. Let us check that the analyze_points thinks this missing register is the cause of the problem. Run :

```
fm_shell (verify)> analyze_points -failing
```

```
fm_shell (verify)> analyze_points -failing
Found 1 Unmatched Cone Input
--------------------------------
Unmatched cone inputs result either from mismatched compare points
or from differences in the logic within the cones. Only unmatched
inputs that are suspected of contributing to verification failures
are included in the report.
The source of the matching or logical differences may be determined
using the schematic, cone and source views.
--------------------------------
r:/WORK/crc_insert/rs_crcblock_reg
    Is globally unmatched affecting 5 compare point(s):
        i:/WORK/crc_insert/crc_block/S0/q_reg
        i:/WORK/crc_insert/crc_block/S1/q_reg
        i:/WORK/crc_insert/crc_block/S2/q_reg
        i:/WORK/crc_insert/crc_block/S3/q_reg
        i:/WORK/crc_insert/reset_crc4

-----------
--------------------------------
Found 1 Rejected Guidance Command
--------------------------------
The rejection of some SVF guidance commands will almost invariably
cause verification failures. For more information use:
        'report_svf_operation -status rejected -command command_name
--------------------------------
reg_constant
-----------
--------------------------------
Found 1 Required Input
--------------------------------
A required input is one that is designated as required
for all failing patterns for one or more cpoints and fans out
to more failing than passing points.
This implies that it may be driving downstream logic that is related to
the failure(s)
--------------------------------
i:/WORK/crc_insert/clk
    Fans out to 4 failing and 10 passing points and has
    logic value '1' for 4 compare point(s):
        i:/WORK/crc_insert/crc_block/S0/q_reg
        i:/WORK/crc_insert/crc_block/S1/q_reg
        i:/WORK/crc_insert/crc_block/S2/q_reg
        i:/WORK/crc_insert/crc_block/S3/q_reg

-----------
--------------------------------
*******************************************************************************
Analysis Completed
1
```

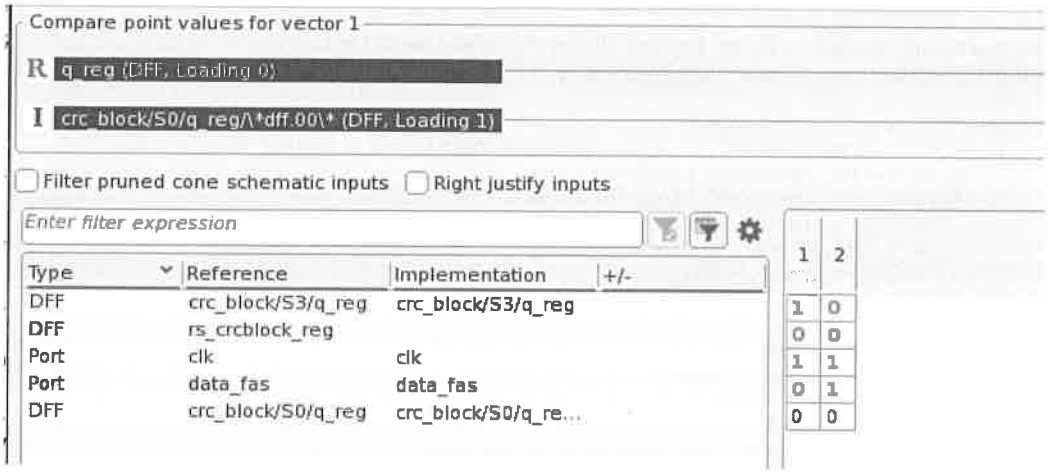
Question 5. Is the results of **analyze_points –failing** consistent with our observations of the SVF and matching summary ?

.....

Question 6. Is there a difference in the name of the register unmatched in the logic cones of the failing point and the SVF constant register that was rejected?

.....

7. Let us look at the patterns for the failing points, say crc_block/S0/q_reg



Question 7. For what value of rs_crcblock_reg do you get a failing vector?

.....

Question 8. Is the value consistent with the value in the rejected SVF ?

.....

8. Let us try verifying the design by forcing the register to be a constant 1.

```
fm_shell (verify)> setup
```

```
fm_shell (setup)>  set_constant r:/WORK/crc_insert/rs_crcblock_reg 1
```

Question 9. Does the verification now succeed ?

9. From the above we have established that **if** the rs_crc_block_reg is a constant 1 then the verification succeeds. This is **not**sufficent to sign-off of the verification with as we don't know whether the register is indeed a constant.

A possible next step is to do single point verification. (Note: remember to remove the constant ' **remove_constant –all**' before you do the single point verification. Setting somthing to a constant then verifying it is constant just verifies that the set_constant command has worked)

```
fm_shell (setup)  verify  r:/WORK/crc_insert/rs_crcblock_reg  -constant1
```

10. Another way forward is to modify the SVF.

Take a copy of the SVF directory

```
unix% cp -r formality_svf my_mod_svf
```

Modify my_mod_svf/svf.txt so change rs_crcblk_reg to rs_crcblock_reg

```
79
80
81
82 guide_reg_constant \
83   -design { crc_insert } \
84   { rs_crcblock_reg } \
85   { 1 }
86
87
```

11. Copy runme.fms to runme2.fms and edit runme2.fms to point to the modified SVF. You will need the line :

set_svf my_mod_svf/svf.txt

Run the verification. It should now pass.

```
***************************** Guidance Summary *****************************
                                        Status
Command                 Accepted   Rejected  Unsupported  Unprocessed  Total
------------------------------------------------------------------------------
change_names          :        7          0            0            0      7
environment           :        3          0            0            0      3
instance_map          :        2          0            0            0      2
mark                  :        2          0            0            0      2
reg_constant          :        1          0            0            0      1
uniquify              :        2          0            0            0      2

SVF files read:
  my_mod_svf/svf.txt

SVF files produced:
  formality_svf/
    svf.txt
******************************************************************************

Status:  Matching...

********************************** Matching Results **********************************
 38 Compare points matched by name
 0 Compare points matched by signature analysis
 0 Compare points matched by topology
 4 Matched primary inputs, black-box outputs
 1(0) Unmatched reference(implementation) compare points
 0(0) Unmatched reference(implementation) primary inputs, black-box outputs
--------------------------------------------------------------------------------
Unmatched Objects                                                  REF        IMPL
--------------------------------------------------------------------------------
 Registers                                                           1           0
   Constrained 1X                                                    1           0
********************************************************************************
```

12. Another approach to this is to rename the object to what it was in the SVF. So for example runme3.fms. This will pass using the original SVF.

```
unix% fm_shell -f runme3.fms | tee -i runme3.log
```

13. (Optional) Modify the SVF so that the register is a constant 0 (rather than a constant 1)

```
guide_reg_constant \
  -design { crc_insert } \
  { rs_crcblock_reg } \
  { 0 }
```

Confirm that this is rejected :

```
fm_shell (verify)> report_svf_operation -status rejected -command reg_constant
****************************************************
Report          : svf_operation
                  -status rejected
                  -command reg_constant

Reference       : r:/WORK/crc_insert
Implementation  : i:/WORK/crc_insert
Version         : O-2018.06-SP2
Date            : Sat Oct  6 14:04:58 2018
****************************************************

## SVF Operation 11 (Line: 83) - reg_constant.  Status: rejected
## Operation Id: 11
guide_reg_constant \
  -design { crc_insert } \
  { rs_crcblock_reg } \
  { 0 }

guide_reg_constant 11 (Line: 83) Retried with -replaced set to svfTrue.
Warning:  guide_reg_constant 11 (Line: 83) Could not verify SVF constant 0 register:
r:/WORK/crc_insert/rs_crcblock_reg due to register pre-verification failure (FM-348)
```

Also note that that

report_svf_operation –summary ${ref}/crc_block/S0/q_reg

reports the rejected guide_constant.

Question 10. When the SVF was rejected because rs_crcblk_reg was not found would :

report_svf_operation –summary ${ref}/crc_block/S0/q_reg

report this ?

(Try the report_svf_operation –summary ${ref}/crc_block/S0/q_reg is the orignal runme.fms run if not sure.)

Answers / Solutions

Task 1. Constant register debug

Question 1. Is there anything in particular rejected in the SVF guidance summary that could cause a failing verification ?

```
************************** Guidance Summary ***************************
                                        Status
Command                 Accepted   Rejected  Unsupported  Unprocessed  Total
-----------------------------------------------------------------------------
change_names          :        7          0            0            0      7
environment           :        3          0            0            0      3
instance_map          :        2          0            0            0      2
mark                  :        2          0            0            0      2
reg_constant          :        0          1            0            0      1
uniquify              :        2          0            0            0      2

Note: If verification succeeds you can safely ignore unaccepted guidance commands.

SVF files read:
  crc_insert.svf

SVF files produced:
  formality_svf/
    svf.txt
***********************************************************************
```

Yes – there is a rejected guide_reg_constant.

Question 2. Is there anything in the match summary that is consistent with the answer to Question 1.?

```
********************************* Matching Results ********************************
 38 Compare points matched by name
 0 Compare points matched by signature analysis
 0 Compare points matched by topology
 4 Matched primary inputs, black-box outputs
 1(0) Unmatched reference(implementation) compare points
 0(0) Unmatched reference(implementation) primary inputs, black-box outputs
-----------------------------------------------------------------------------
Unmatched Objects                                                        REF
-----------------------------------------------------------------------------
 Registers                                                                 1
   DFF                                                                     1
***********************************************************************
```

There is one unmatched DFF (flip-flop) in the reference. This would be consistent with DC synthesizing a constant register away.

Question 3. Given answers to questions 1) and 2) could you have predicted the results of the verify step without running verify ?

Yes.

Question 4. What is the reason for the constant being rejected ?

Formality couldn't find that register to apply the constant to.

```
fm_shell (verify)> report_svf_operation -status rejected -command reg_constant

## SVF Operation 11 (Line: 82) - reg_constant.  Status: rejected
## Operation Id: 11
guide_reg_constant \
  -design { crc_insert } \
  { rs_crcblk_reg } 1

Info:  guide_reg_constant 11 (Line: 82) Cannot find master reference cell 'rs_crcblk_reg'.

1
```

Question 5. Arethe results of analyze_points –failing consistent with our observations of the SVF and matching summary?

Yes.

It has flagged two things:

1) That there is an input register that is not matched in the cone of logic of the failing points
2) A more general point that a rejected reg_constant will usually lead to a failed verification.

Question 6. Is there a difference in the name of the register unmatched in the logic cones of the failing point and the SVF constant register that was rejected?

Yes. The unmatched register
r:/WORK/crc_insert/rs_crcblock_reg

The register in the SVF rs_crc**blk**_reg (ie **blk** not **block**)

Question 7. For what value of rs_crcblock_reg do you get a failing vector?

0

Question 8. Is the value consistent with the value in the rejected SVF ?

It is the opposite value for which DC synthesized the register away.

Question 9. Does the verification now succeed ?

Yes

Question 10. When the SVF was rejected because rs_crcblk_reg was not found would :

report_svf_operation –summary ${ref}/crc_block/S0/q_reg

report this ?

No. That is Formality doesn't knowrs_crcblk_reg is in the logic cone of ${ref}/crc_block/S0/q_reg.

10 Other Design Transforms

Learning Objectives

In this lab we will look at isolating bugs in a design and seeing how one or more debugging features help you do this.

Lab Duration:

Introduction

Answers & Solutions

Each lab contains answers to all questions and results or solutions.

*You are encouraged to verify your results by checking the **Answers/Solutions** section at the end of each lab.*

Instructions

The general context and setup of the lab,is to mimic a real logic bug,is to synthesize from slightly different RTL than what we are using in Formality.

Task 1. Isolating bug b1

1. Go to lab10a

```
unix% cd lab10/lab10a
```

tom.v is the source RTL for Formality.

2. Go to b1

```
unix% cd b1
```

From the log file run1_fm.log one can see that the verification has failed with 1 failing point.

3. Restore the session file run1_fm.fss
4. Pull up the pattern window.

Question 1. Just from the pattern window what is the nature of the bug ?

.....

5. Quit out of Formality

Task 2. Isolating bug b2

1. Go to b2

```
unix% cd lab10/lab10a/b2
```

From the log file run1_fm.log one can see that the verification has failed with 2 failing points.

2. Restore the session file run1_fm.fss
3. Pull up the pattern window. Look at pattern window for each of the two failing points.

Question 2. What is particularly suspicious about the pattern windows for the two failing points ?

.....

4. Pull up the logic cone schematics for failing point z_reg_0 
5. Use  to group the hierarchy for both Ref and Impl. Observe that Ref regbank_inst/z_reg_0 is driven by the hierachical pin alu_inst/result[0] whereas in the Impl it is driven by alu_inst/result[1]
6. We wish to determine whether the bug is in regbank_inst or alu_inst. We can do hierachical verification to do this. Quit out of the GUI but not the fm_shell.
7. From the fm_shell issue the command :

```
write_hierarchical_verification_script  -dont_resolve_failures\
-replace -path   $ref/regbank_inst/z_reg_0_  hier.tcl
```

8. source hier.tcl in fm_shell
9. Observe that regbank failed by the level of hierarchy above passed

```
****************************************************************************************************
Results of hierarchical verification script: hier.tcl
****************************************************************************************************

Verification FAILED:
  Ref: r:/WORK/tom/regbank_inst (instance of r:/WORK/regbank)
  Imp: i:/WORK/tom/regbank_inst (instance of i:/WORK/regbank)
  Tue Mar 26 06:36:12 2013,  172MB (cumulative),    0.02sec (incremental)
  Session file: ./fm_hier.tcl.1.fss

Verification SUCCEEDED:
  Ref: r:/WORK/tom (top)
  Imp: i:/WORK/tom (top)
  Tue Mar 26 06:36:12 2013,  172MB (cumulative),    0.16sec (incremental)
```

This would suggest the problem is inside regbank

10. Restore the failing session created by hier.tcl

restore_session fm_hier.tcl.1.fss

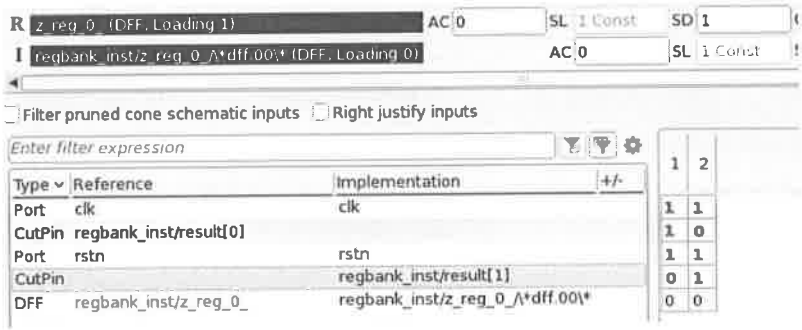
11. Look at the failing pattens for z_reg_0 and z_reg_1

Question 3. From the pattern window is it now clearer what the issue is ?

.....

12. (Optional) Confirm the answer to Question 3 by using the set_user_match command (Hint. You might need to remove_user_match put in by the hierarchical verification script. Solution is in .solution/match.tcl)

13. (Optional) Restore the original session run1_fm.fss and use cutpoints to debug the issue rather than hierarchical verification. Hint. Solution is in .solution/cut.tcl One is trying to reduce the size of the logic cone so that the pattern window looks like this

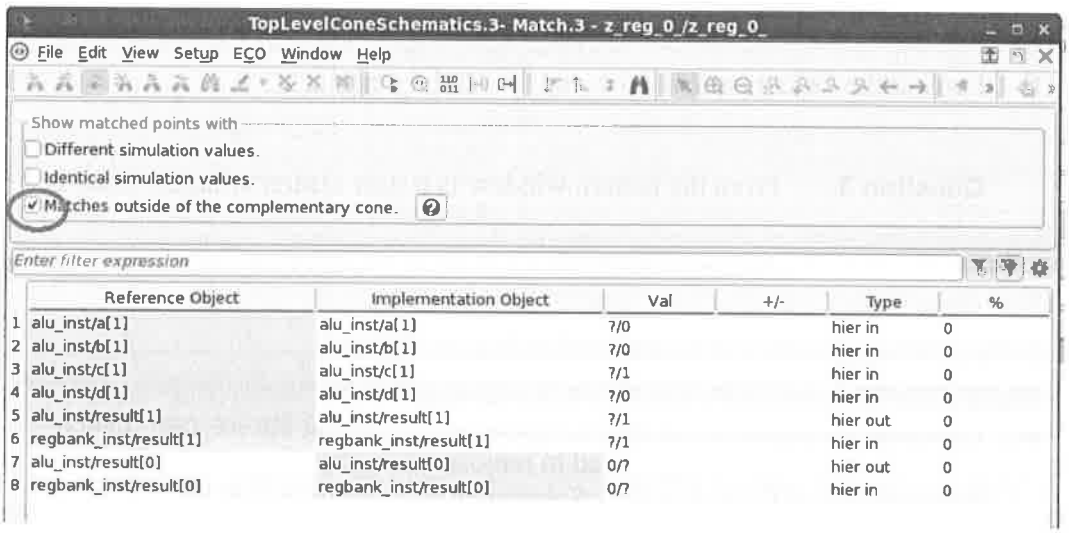


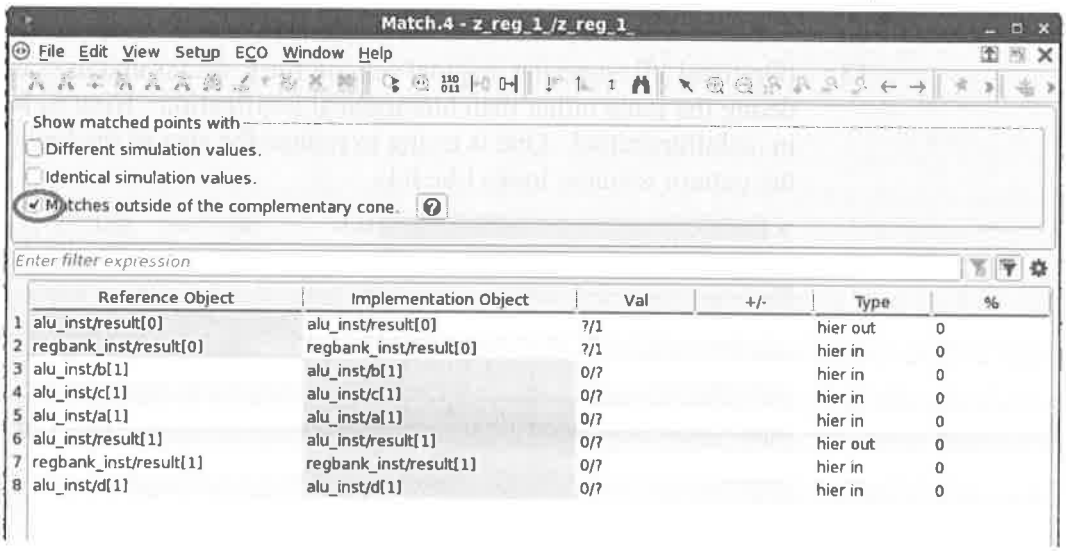
14. (Optional) Restore the original failing session run1_fm.fss and use the 'Matching tool'  to help identify the problem

Hint. A useful setting in the 'Match Tool' will be 'Show points with matches outside of the complementary cone'

One should be able to see that for the failing point z_reg_0_ the Impl path goes through inputs to alu_inst (eg alu_inst/a[1]), throug output alu_inst/result[1] and input regbank_inst/result[1] but doesn't (greyed out) for the Ref. For z_reg_0_ the Ref cone includes output alu_inst/result[0] and input regbank_inst/result[0] – but doesn't for the Impl (greyed out)

For z_reg_1_ the above description is the opposite way round.





Note whether it is 1/? or 0/? in val column will depend on which pattern is selected to overlay on hierarchy.

Task 3. Isolating bug b3

1. Go to b3

```
unix% cd lab10/lab10a/b3
```

From the log file run1_fm.log one can see that the verification has failed with 1 failing points.

2. Restore the session file run1_fm.fss
3. Use whatever method you feel is appropriate to narrow down the problem .

(Hint : There is no absolute right answer. Matching tool  Probe points, Cut-point, Pattern Window; Hierarchical verification - will all get you there in some combination.)

Question 4. What is the b3 bug ?

.....

Answers / Solutions

Task 1. Isolating bug b1

Question 1. Just from the pattern window what is the nature of the bug ?

The flip-flop is mapped incorrectly. The Ref has a AC (Async Clear) pin. The Impl has an AS (Async Set) pin. And it is failing when reset is asserted.

The only way you could legitimately get that transformation is by inv_push. However if that had happened you would also expect everything the register was driving to fail. We only have one failing point here.

So for this bug pretty much diagnosed what the issue is from the pattern window.

Task 2. Isolating bug b2

Question 2. What is particularly suspicious about the pattern windows for the two failing points ?

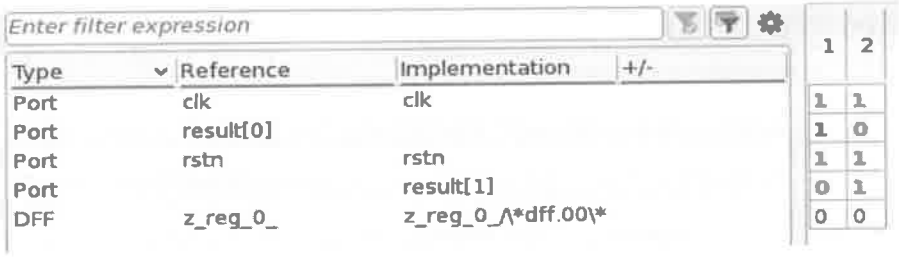
z_reg_1 doesn't have dr_reg_1, br_reg_1 ar_reg_1 in the Impl logic cone

z_reg_0 doesn't have dr_reg_1, br_reg_1 ar_reg_1 in the Ref logic cone

At this point if you were familiar with the RTL you could probably guess what had happened.

Question 3. From the pattern window is it now clearer what the issue is ?

Yes. Pretty much so.



Somewhere inside regbank result[0] and result[1] bits have got swapped. Notice the characteristic opposite failing pattern ie failing patterns when result[0] and result[1] take opposite values.

Task 3. Isolating bug b3

Question 5. What is the b3 bug ?

Anything that got you close to the difference between tom.v and ../rtl/tom_b3.v

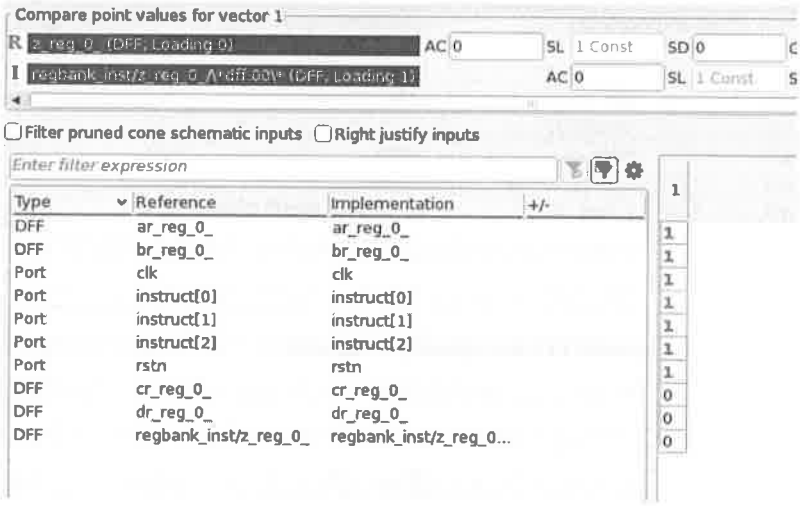
Note you are not typically going to be able to narrow it down to individual gates – but you want to be able to narrow it down to where it is in hierarchy and who is right Formality or DC.

A possible route to debug b3 is .

1) analyze -failing

   Nothing found. Always worth a try though.

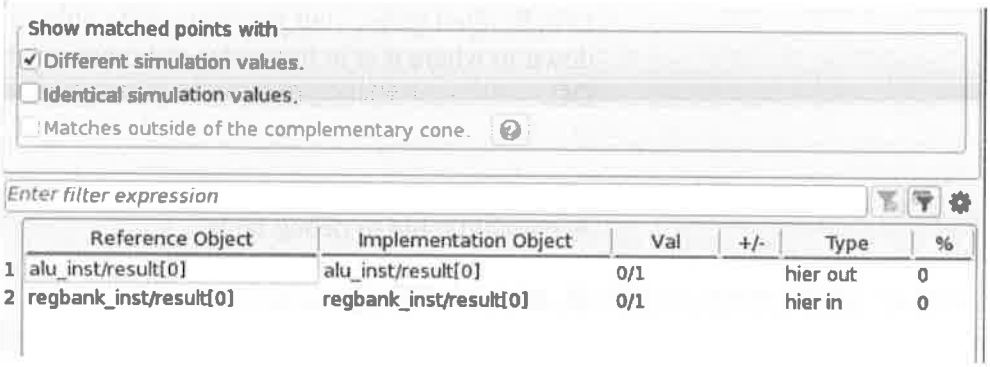
2) Pattern window



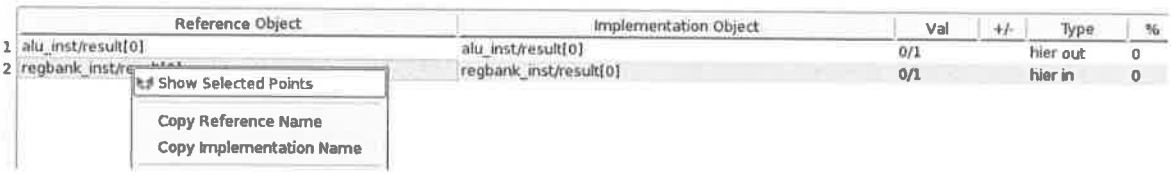
Narrows it down a bit. It fails when those registers and instruct input ports are a particular value.

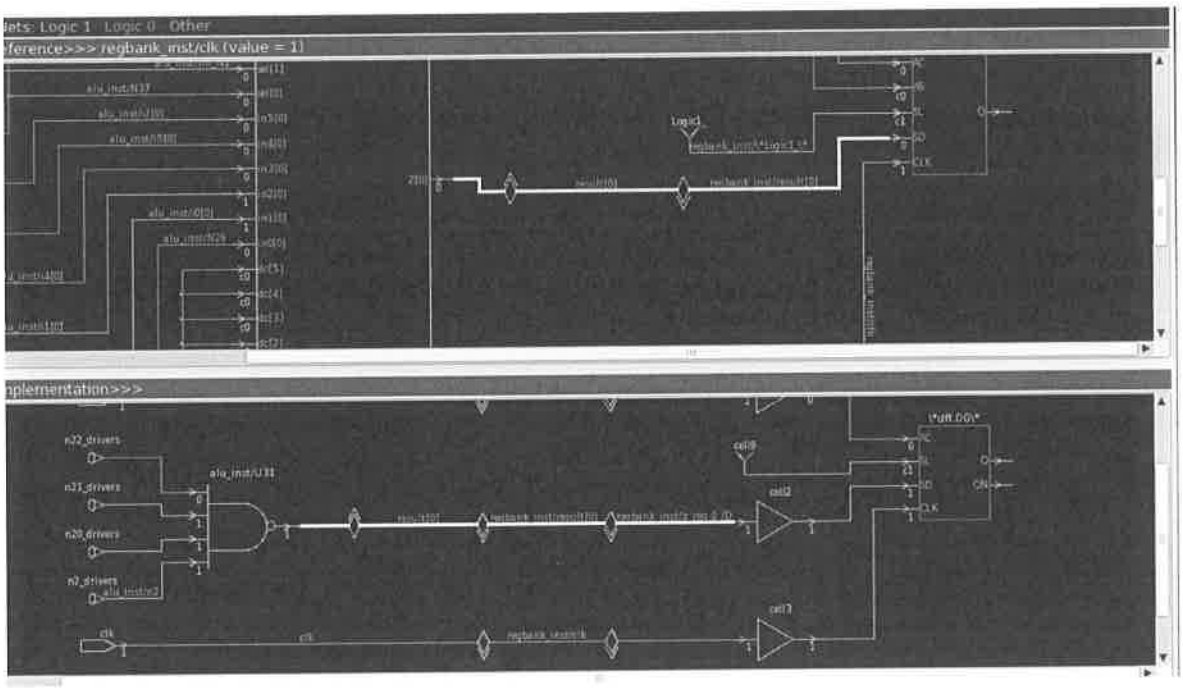
3) Matching tool

Pattern value different on the output of alu_inst/result[0]



One can then pull up the logic cones for alu_inst/result[0] using "Show Selected Points"

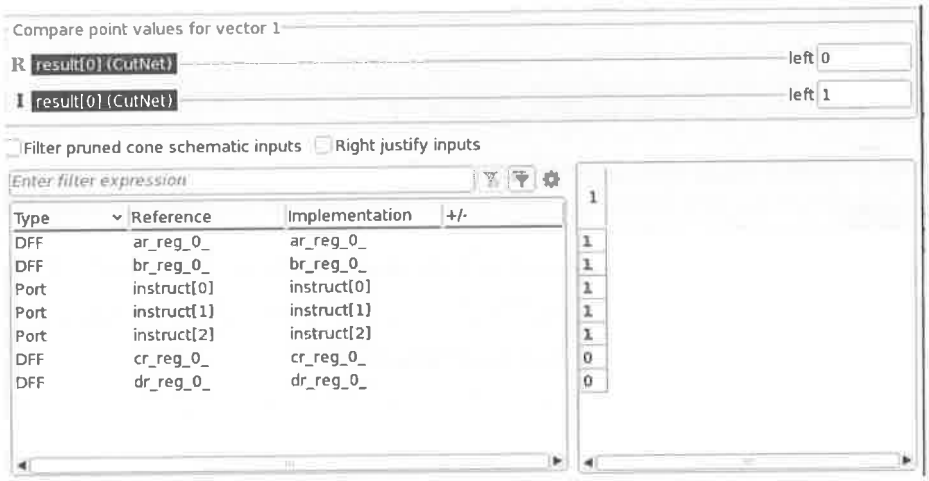




Select the net attached to that already highligted hierarchical pin in both Ref and Impl and add a probe

4) Probe point pattern window

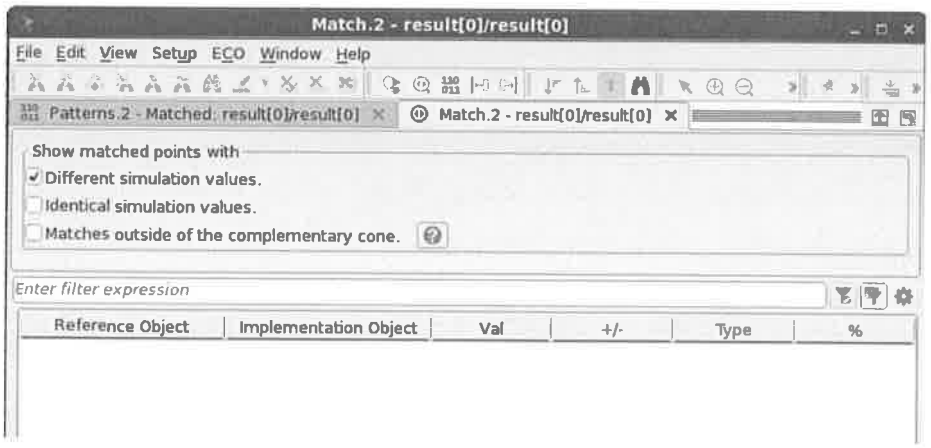
Verify the probe point created from 3) above. It will fail (as you would expect from the 'Matching Tool'. Pull up the pattern window from the failed probe point.



The vectors are the same from 2) above except narrowed down to looking at an output of alu_inst

One can check using the 'Matching Tool' on the probe point that there is nowhere in the hierarchy where the

failing pattern simulation values are different for the probe point



That pretty much nails the difference to inside alu_inst

5) Fine grain isolation with constraints.

There is stuff you can do in the schematic to narrow it down further straight off. However a possible approach is to simplify the starting point further with constraints.

One knows that the failing vector is with instruct[2:0] as 1. Let us make that explicit.

setup

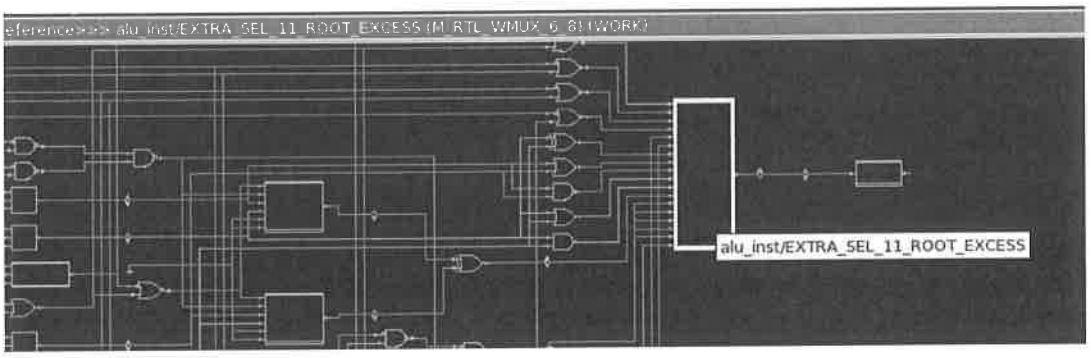
set_constant -type port ${ref}/instruct* 1

set_constant -type port ${impl}/instruct* 1

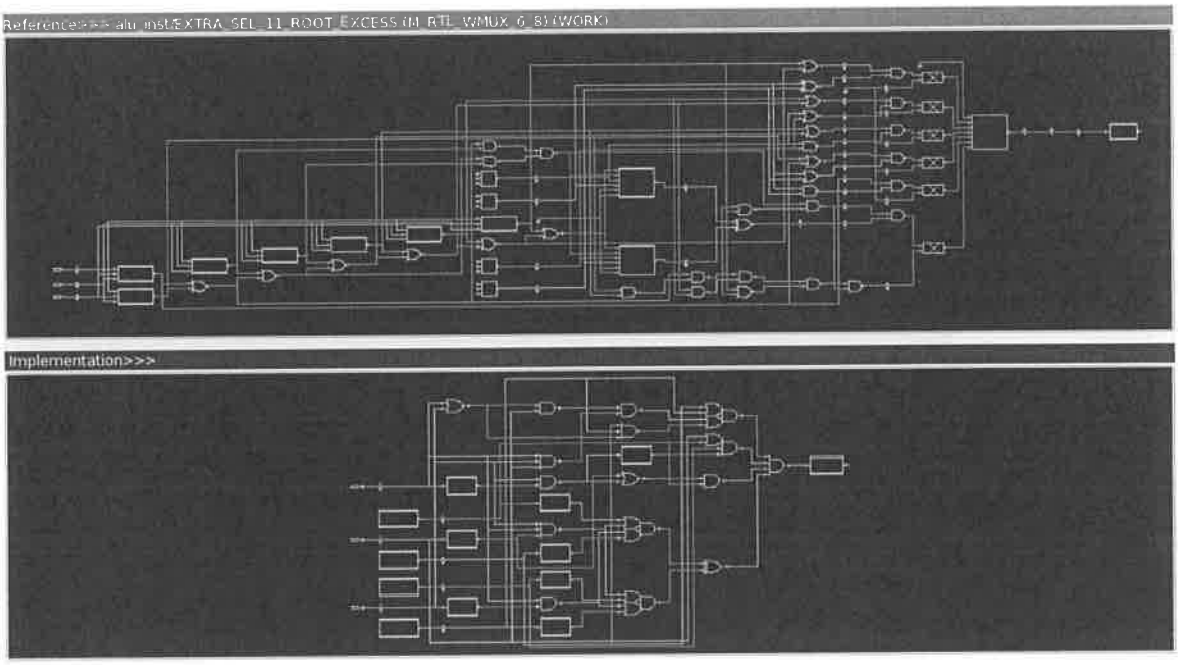
verify

Rerun the probe verification and pull up the probe point schematic.

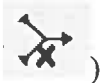
Expand the schematic in Ref and Impl and also ungroup the RSOP_SEL_8 block in Ref and expand both Ref and Impl schematics



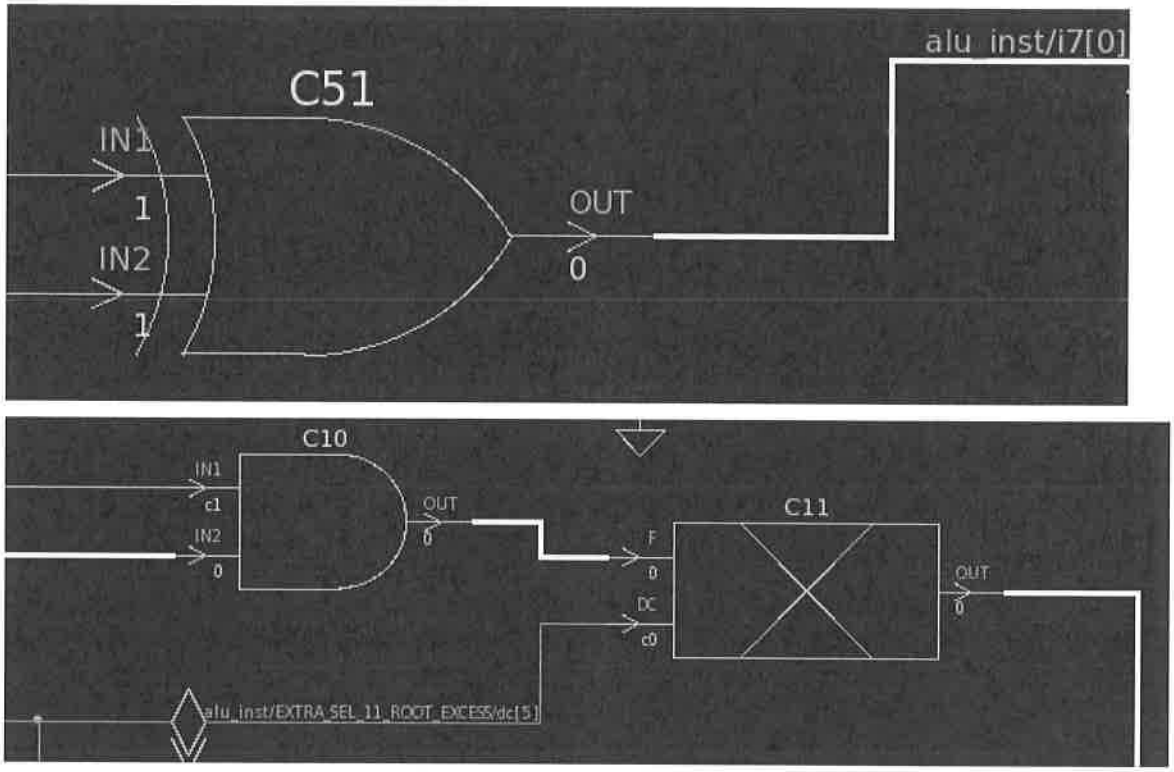
The schematic initially still looks daunting.



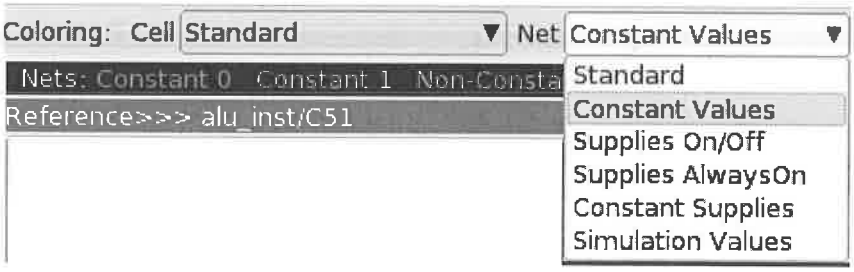
However if one zooms in one sees the symbols 'c1'and 'c0' indicating constant 1 and constant 0 respectively. Some, if not all, of these constants will have come from propagating the constants we put on the instruct ports. Now it just a matter of working back in the logic cone of Ref selecting the net of input to the gate that is constant and removing the subcone

(Key F6 or )

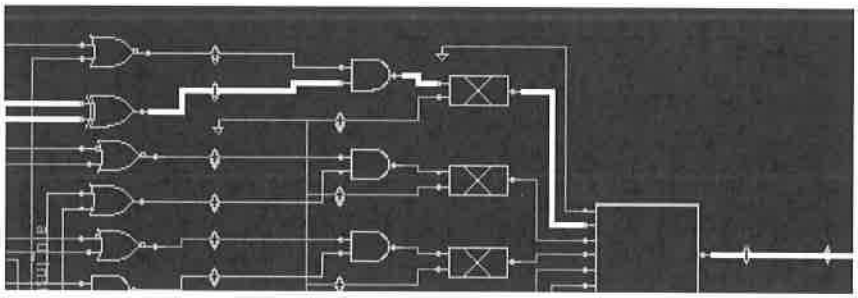
One quickly sees that the the Ref logic cone is ar_reg_0 XOR br_reg_0 (A c1 on the input of C10 AND gate turns it into a buffer)



Another way of tracing this is to change the net colouring

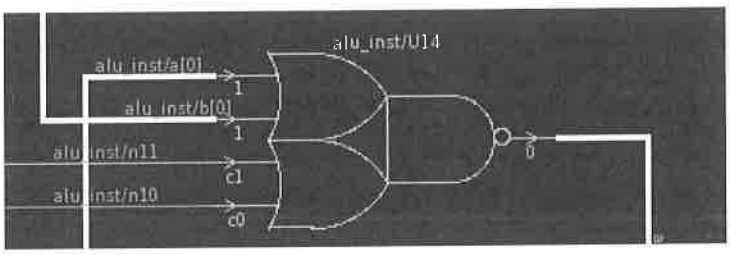


to constant values



The non-constant net values will be coloured grey (high-lighted in white above). Again one can see the path back to the XOR gate.

Repeating for the Impl one sees that the relevant gate is alu_inst/U14 .



alu_inst/U15/A is connected to alu_inst/a[0] alu_inst/U14/B is connected to alu_inst/b[0] . U14 drives the NAND gate U31 (ie an inversion) so that just leaves us with th functionality of

alu_inst/b[0] | to alu_inst/a[0] as an OR rather than the XOR functionality in the Ref,

If one does tkdiff ../tom.v ../rtl/tom_b3.v one sees that is indeed the difference.

i7[0] has changed from XOR (^) to OR (|) and the wire i7 is selected when instruct is 111.

```
74  assign i6 = c + d ;
75  assign i7 = a ^ b;
76
77
78  always @(i0 or i1 or i2 or i3 or i4 or i5 or i6 or i7 or instru
79  begin
80   if (instruct == 3'b000 )
81   begin
82       result = i0;
83   end
84   else if   (instruct == 3'b001)
85   begin
86      result = i1;
87   end
88   else if   (instruct == 3'b010)
89   begin
90      result = i2;
91   end
92   else if   (instruct == 3'b011)
93   begin
94      result = i3;
95   end
96   else if   (instruct == 3'b100)
97   begin
98      result = i4;
99   end
100  else if   (instruct == 3'b101)
101  begin
102     result = i5;
103  end
104  else if   (instruct == 3'b110)
105  begin
106     result = i6;
107  end
108  else
109  begin
110     result = i7;
111  end
112
```

```
74  assign i6 = c + d ;
75  assign i7[7:1] = a[7:1] ^ b[7:1]; assign i7[0] =  a[0] | b[0];
76
77
78  always @(i0 or i1 or i2 or i3 or i4 or i5 or i6 or i7 or instru
79  begin
80   if (instruct == 3'b000 )
81   begin
82       result = i0;
83   end
84   else if   (instruct == 3'b001)
85   begin
86      result = i1;
87   end
88   else if   (instruct == 3'b010)
89   begin
90      result = i2;
91   end
92   else if   (instruct == 3'b011)
93   begin
94      result = i3;
95   end
96   else if   (instruct == 3'b100)
97   begin
98      result = i4;
99   end
100  else if   (instruct == 3'b101)
101  begin
102     result = i5;
103  end
104  else if   (instruct == 3'b110)
105  begin
106     result = i6;
107  end
108  else
109  begin
110     result = i7;
111  end
112
```

If you recall the failing pattern :

DFF	ar_reg_0_	ar_reg_0_	1
DFF	br_reg_0_	br_reg_0_	1

and recall that the function an XOR differs from an OR only when the inputs are both '1' then once again the features of Formality are all tellin a consistent story

5) There is a command '**diagnose**' that also automatically tries to isolate the difference using the failing patterns.. Restoring the failed session (ie without the constants applied) the results we get are simmilar to the above. That is alu_inst/C51 and alu_inst/U14 are identified as possible problem gates.

```
fm_shell (verify)> diagnose -r
Status:  Diagnosing r:/WORK/tom vs i:/WORK/tom...
Status:  Diagnosis initializing...
Status:  Analyzing patterns...
   Single error detected in reference design.
   Number of error candidates: 11
   Analysis completed
Status:  Finding matching regions in implementation design...
   Single matching region detected in implementation design.
Diagnosis completed
1
fm_shell (verify)> report_error_candidates
****************************************************
Report       : error_candidates

Reference    : r:/WORK/tom
Implementation : i:/WORK/tom
Version      : O-2018.06-SP2
Date         : Mon Oct  8 11:57:32 2018
****************************************************

Single error detected in reference.

Recommended error candidate:

   Prim   alu_inst/C51

Alternate error candidates:
 1. Prim   alu_inst/EXTRA_SEL_11_ROOT_EXCESS/C0

 2. Prim   alu_inst/EXTRA_SEL_11_ROOT_EXCESS/C2

And diagnosing the Impl :


fm_shell (verify)> diagnose
Status:  Diagnosing i:/WORK/tom vs r:/WORK/tom...
Status:  Diagnosis initializing...
Status:  Analyzing patterns...
   Single error detected in implementation design.
   Number of error candidates: 4
   Analysis completed
Status:  Finding matching regions in reference design...
   Single matching region detected in reference design.
Diagnosis completed
1
fm_shell (verify)> report_error_candidates
****************************************************
Report       : error_candidates

Reference    : r:/WORK/tom
```

```
Implementation : i:/WORK/tom
Version        : O-2018.06-SP2
Date           : Mon Oct  8 12:04:54 2018
************************************************
```

Single error detected in implementation.

Recommended error candidate:

Cell alu_inst/U12

Alternate error candidates:

1. Cell alu_inst/U14

2. Cell alu_inst/U31

3. Cell regbank_inst/z_reg_0_