

Report

Confirmation of Originality

Faculty of Engineering and Computer Science

Course Name & Number/Term: COEN 6551 Section: AA Instructor: Dr. Sofiène Tahar.
 e.g., ENGR410/2 e.g. M

Having researched and prepared this report for submission to the Faculty of Engineering & Computer Science, the undersigned certify that the following statements are to the best of my/our knowledge true:

1. The undersigned have written this report myself/themselves.
2. This report consists entirely of ideas, observations, references, information and conclusions composed or paraphrased by the undersigned, as the case may be, except for statements contained within quotation marks and attributed to the best of my/our knowledge to their proper source in footnotes or otherwise referenced.
3. With the exception of material in appendices, the undersigned have endeavored to ensure that direct quotations make up a very small proportion of the attached report, not exceeding 5% of the word count.
4. Each paragraph of this report that contains material which the undersigned have paraphrased from a source (print sources, multimedia sources, web-based sources, course notes or personal interviews, etc), has been identified by numerical reference citation.
5. All of the sources that the undersigned consulted and/or included in the report have been listed in the Reference section of the report.
6. All drawings, diagrams, photos, maps or other visual items derived from sources have been identified by numerical reference citations in the caption.
7. Each of the undersigned has revised, edited and proofread this report individually.
8. In preparing this report the undersigned have read and followed the guidelines found in Form and Style, by Patrick MacDonagh and Jack Borden (Fourth Edition: May 2000), available at <http://www.encs.concordia.ca/scs/Forms/Form&Style.pdf>.

Name: Prithvik Adithya Ravindran ID No: 40195464 Signature: Prithvik Date: 19/06/2023
 (please print clearly)

Name: Behnam Shakibafar ID No: AR83890 Signature: Behnam Shakibafar Date: 19/06/2023
 (please print clearly)

Name: Rushik Shingala ID No: 40221905 Signature: Rushik Date: 19/06/2023
 (please print clearly)

Name: _____ ID No: _____ Signature: _____ Date: _____
 (please print clearly)

Name: _____ ID No: _____ Signature: _____ Date: _____
 (please print clearly)

Name: _____ ID No: _____ Signature: _____ Date: _____
 (please print clearly)

Do Not Write in this Space – Reserved for Instructor



PROJECT REPORT

Formal Hardware Verification (CEON 6551)

Formal Verification of I2C design using Equivalence Checking

Submitted on: - 19th June 2023

Instructor: Dr. Sofiène Tahar

Group 4

Prithvik Adithya Ravindran (40195464)

Rushik Shingala (40221905)

Behnam Shakibafar

Abstract:

This project aims to validate the equivalence of a I2C circuit through the use of two popular verification tools, namely Synopsys Formality and Cadence Conformal. The report provides a brief explanation of the functionality and behavior of the system under verification, which is a I2C circuit. The verification process involves synthesizing the given RTL design using Synopsys Design Vision tool, and the resulting gate-level code is verified and compared with RTL design with both software and after that synthesized file use as a 'Reference' or 'Golden' file for further equivalence checking.

The report outlines the steps involved in performing equivalence checking from RTL to gate level and gate-level to gate-level, along with relevant figures. It also covers the end-to-end process, starting from code synthesis to bug detection and resolution. Later sections of the report document the verification methodology using Formality and Conformal tools. The results obtained from both tools are compared, and an analysis is conducted in the analysis section. Lastly, the report provides academic references that support the procedures and explanations presented throughout.

Contents

1. Introduction.....	3
1.1 Motivation.....	3
1.2 Objective.....	3
1.3 Overview	3
1.4 System Behavior	4
1.5 Architecture of the System	5
1.6 State Diagram of System	5
2. RTL Synthesis using Synopsys Design Compiler	6
3. RTL-Gate Level Equivalence Checking	8
3.1Equivalence Checking in Synopsys – Formality	8
3.2Equivalence Checking in Cadence – Conformal	9
4. Gate-Gate Equivalence Checking.....	11
4.1Equivalence Checking in Synopsys – Formality	11
4.2 Equivalence Checking in Cadence – Conformal	13
5. Verification and Bug Hunting Process.....	15
5.1Analysis of resolved bugs.....	16
5.2 Analysis of the Un-Resolved Bug.....	17
6. Analysis of Verification results.....	18
7. Comparison of Synopsys Formality and Cadence Conformal.....	18
8. Conclusion	19
Challenges	19
Comment.....	19
9. References.....	19

1. Introduction

1.1 Motivation

Verification plays a crucial role in the development process since it makes sure that a system or product complies with all criteria, works as intended, and is reliable. Validating the specs ensures that the design adheres to the predefined criteria and specifications and achieves the desired goals.

In the case of digital design, while logic verification tries to confirm the design's logical behavior at the gate level, RTL verification concentrates on ensuring the accuracy of the design's register transfer level (RTL) representation. Layout verification makes assurance that the design's physical layout complies with the design's rules and limitations, reducing the possibility of manufacturing problems. There are lots of examples how system verification could prevent huge losses. Here are two well-known cases:

Boeing 737 Max: Following two incidents involving the 737 Max aircraft, investigators discovered that a software mechanism known as MCAS on the planes might be activated by a single sensor reading, possibly forcing the nose down repeatedly if that sensor malfunctioned¹. Proper system verification would have uncovered the system's susceptibility to single point failures and avoided the crashes.[1]

Failure of Ariane 5 Flight 501: In 1996, the European Space Agency's Ariane 5 Flight 501 detonated forty seconds after launch. A software issue in the inertial reference system triggered the breakdown. A 64-bit floating point value referring to the rocket's horizontal velocity with respect to the platform was transformed to a 16-bit signed integer. Because the number was more than 32,767, the maximum integer storable in a 16-bit signed integer, the conversion failed, resulting in a hardware error. A thorough system check may have detected this problem before the deployment. The overall cost of the lost rocket and its cargo was estimated to be around \$370 million.

“This loss of information was due to specification and design errors in the software of the inertial reference system. The extensive reviews and tests carried out during the Ariane 5 development program did not include adequate analysis and testing of the inertial reference system.” [2]

1.2 Objective

Verification is a growing challenge for engineers due to the complexity of IP designs. A well-designed verification environment is essential to uncover unexpected bugs and prevent issues for end consumers. In this project, we are trying to verify and fix a new implementation of the I2C module which has a few bugs. The original code is provided by opencore (Author: Richard Herveille, richard@asics.ws, www.asics.ws) which includes bit control, byte control and top module files.

I2C combines the features of SPI and UART, providing efficiency and minimal hardware requirements. It facilitates secure communication among multiple masters and slaves with minimal wiring. In this project, the master is a hardware block, and the slave is a verification IP.

In this project, we will try to synthesize the reference design and then will try to use the Formality tool to find out the differences between the buggy and the reference design.

1.3 Overview

Formal verification is capable of detecting bugs that may go undetected by standard verification techniques. It can identify bugs at an earlier stage, before the design is ready for verification through simulation and emulation. Formal verification offers several advantages, including time efficiency, reliability, exhaustive bug detection, and the ability to find issues during early design cycles. The commonly used techniques in formal verification include Model Checking, Theorem Proving, and

Equivalence Checking. Equivalence Checking can be further categorized into Logical and Sequential checking.[3]

Model Checking, also known as property checking, follows a state-based verification approach. The design under verification is modeled as a set of states and transitions, and a property to be verified is formulated. This formula is then checked against the model, and if the model fails to satisfy the property, a counterexample is generated. Model Checking offers automatic verification but faces the challenge of processing larger states as the system grows **(State Space Explosion)**. [3]

Theorem Proving relies on mathematical reasoning to verify if the system design meets the required specifications. The system is defined using mathematical definitions, and properties derived from these definitions are proved using available theorem provers. Theorem Proving can handle complex systems but requires manual intervention, expertise, and more time compared to automated approaches. It also lacks the ability to generate counterexamples for failed cases.

Equivalence Checking aims to verify if two different representations of a system are equivalent. It can be used to establish the equivalency between representations on distinct abstraction levels. Logical Equivalence Checking focuses on verifying if the same combinational logic exists between registers in two designs, while Sequential Equivalence Checking checks if two designs produce the same outputs when provided with the same inputs. Equivalence Checking offers advantages such as high efficiency, quality, and predictability but has limited expressiveness and faces challenges when verifying sequential circuits. [3]

In summary, formal verification techniques, including Model Checking, Theorem Proving, and Equivalence Checking, provide valuable tools for efficient and reliable bug detection in hardware design. Each technique has its strengths and limitations, offering different approaches to ensure the correctness of complex system designs. [3]

1.4 System Behavior

As mentioned in last section, this project is about I2C. Inter-Integrated Circuit, is a serial communication protocol that is often used to link several electrical components inside a system. Hardware connections are made simpler since it only needs two wires—a data line (SDA) and a clock line (SCL). I2C bus devices can interact with one another thanks to the unique addresses that are provided to them. It permits versatile system architectures by supporting both multi-master and multi-slave setups. Depending on the needs of the application, I2C can operate at different speeds, such as standard mode (100 kHz) and fast mode (400 kHz), enabling a range of data transfer rates. [4]

The provided core includes three modules. These are module features regarding to open core website:

- Compatible with Philips I2C bus standard.
- Multi-Master Operation.
- Software programmable timing.
- Clock stretching and wait state generation.
- Interrupt or bit-polling driven byte-by-byte data-transfers.
- Arbitration lost interrupt, with automatic transfer cancelation.
- (Repeated)Start/Stop signal generation/detection.
- Bus busy detection.
- Supports 7 and 10bit addressing.
- Fully static and synchronous design.
- Fully synthesizable.

I2c_master_top: An entity declaration (i2c_master_top) and its related architecture (structural) make up the code. A component declaration (i2c_master_byte_ctrl) is present in the architecture, and it is instantiated as byte_ctrl. The component is a representation of the I2C protocol's byte controller.

The Wishbone signals (wb_clk_i, wb_rst_i, etc.) and the I2C lines (scl_pad_i, scl_pad_o, sda_pad_i, sda_pad_o) have input and output ports defined by the entity i2c_master_top. Additionally, it has internal registers and signals for managing the I2C communication.

The architecture includes tasks and procedures for managing reset signals, producing acknowledge signals, managing write access to registers, decoding command and control registers, and launching the byte controller component.

I2c_master_byte_ctrl: This component controls byte-level operations in an I2C communication system and serves as an interface to other system components. It uses a state machine to manage several operations, including starting and stopping processes, sending and receiving data bytes, producing acknowledgements, and moving data within a shift register. The component uses a "i2c_master_bit_ctrl" bit-level controller to carry out lower-level I2C instructions. Smooth data transmission and reception via the I2C bus is made possible by output signals from the component, which provide status and control information to the external system. The "i2c_master_byte_ctrl" streamlines the entire I2C communication process and improves the effectiveness of data transmission by encapsulating the byte-level functionality.

I2c_master_bit_ctrl: This component contains procedures for creating signals, decoding registers, and controlling the I2C master's state. Additionally, a block for producing the interrupt request signal and status register bits is included.

The states and timing for various I2C instructions are defined in the bit controller part of the code. It has a state machine that executes the instructions and produces the required signals to operate the I2C bus. Additionally, the code manages bus status, including start and stop conditions, bus busy status, and arbitration loss detection. [4]

1.5 Architecture of the System

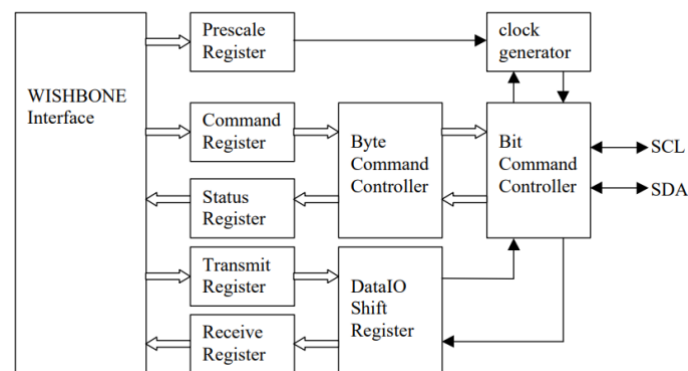


Figure 1 Architecture of system[4]

The I2C core is built around four primary blocks; the Clock Generator, the Byte Command Controller, the Bit Command Controller and the DataIO Shift Register. All other blocks are for interfacing or for storing temporary values.

1.6 State Diagram of System

The Byte Command Controller is responsible for managing I2C traffic at the byte level. It receives data from the Command Register and generates corresponding sequences based on the transmission of a single byte. For instance, if the START, STOP, and READ bits are set in the Command Register, the Byte Command Controller generates a sequence that includes sending a START signal, reading a byte from the slave device, and the generation of a Stop signal. It does this by dividing each byte operation into separate bit-operations, which are then sent to the Bit Command Controller.

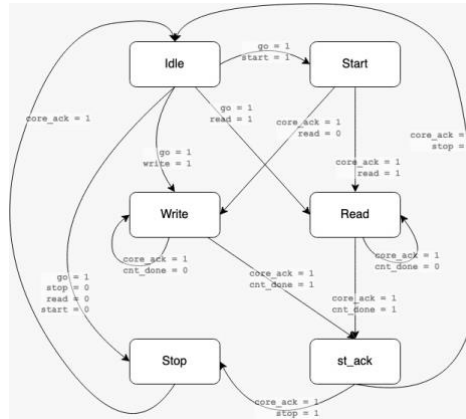


Figure 2 State diagram of system

2. RTL Synthesis using Synopsys Design Compiler

Synthesis involves converting one form of description into another. Design Compiler, developed by Synopsys, is a synthesis tool that provides a graphical user interface (GUI) for exploring and analyzing the behavior of a design at the gate level. Its primary objective is to optimize the Gate-Level description by transforming a large RTL design, written in HDL, into an efficient representation. Design compilers utilize various optimization techniques such as Timing, Power, Area, and Datapath optimization.

The code synthesis process is outlined in the following step-by-step manner.

Step-1: Source the environment script using the command source /CMC/ENVIRONMENT/synopsys.env

Step-2: GUI is opened by using the command design_vision&.

Step-3: Read the following files i2c_master_bit_ctrl.vhd, i2c_master_byte_ctrl.vhd and i2c_master_top.vhd together and then analyze i2c_master_top.vhd file.

Step-4: Elaborate the design as shown in figure- 1.

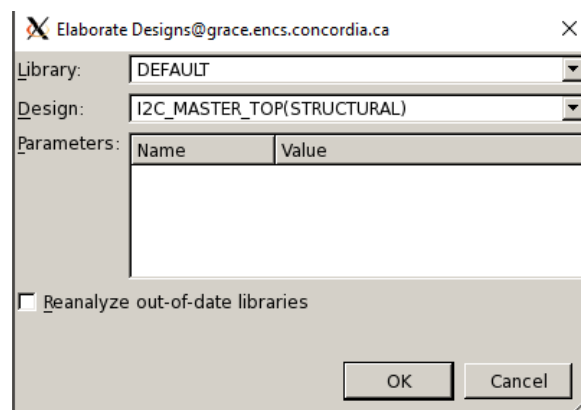


Figure 3- Elaborate design by selecting the top module.

Step-5: The Design has to be checked and made sure that there are no errors. So, it can be done by clicking check design option.

Step-6: The symbol view can be viewed by selecting the elaborated design and then by clicking symbol view option in the toolbar as shown in figure-2.

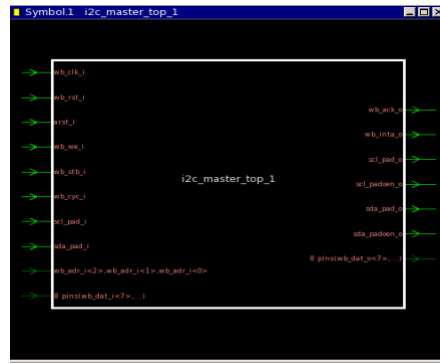


Figure 4- Symbolic view of I2C core.

Step-7: Clock for the design can be specified by selecting the CLK pin in the symbol view and by clicking the “Specify clock” option in the Attributes tab and the clock values are given in Figure-3.

The image shows a 'Specify Clock' dialog box. The 'Clock name' field is set to 'clk'. The 'Port name' field is empty. The 'Remove clock' checkbox is unchecked. Under 'Clock creation', the 'Period' is set to '50'. There is a table with two columns: 'Edge' and 'Value'. The 'Rising' edge has a value of '0', and the 'Falling' edge has a value of '25'. There are buttons for 'Add edge pair', 'Remove edge pair', and 'Invert wave form'. At the bottom, there are checkboxes for 'Don't touch network' (checked) and 'Fix hold' (unchecked). The 'OK', 'Cancel', and 'Apply' buttons are at the bottom right.

Figure 5- Specify Clock

Step-8: The Design with above clock attributes can be now checked and compiled by selecting ‘Check Design’ and ‘Compile Design’ option available in the design tab as shown in Figure-4. The synthesized code is saved as ‘i2c_syn.vhdl’ in order to perform RTL vs Gate-Level and Gate vs Gate-Level equivalence checking.

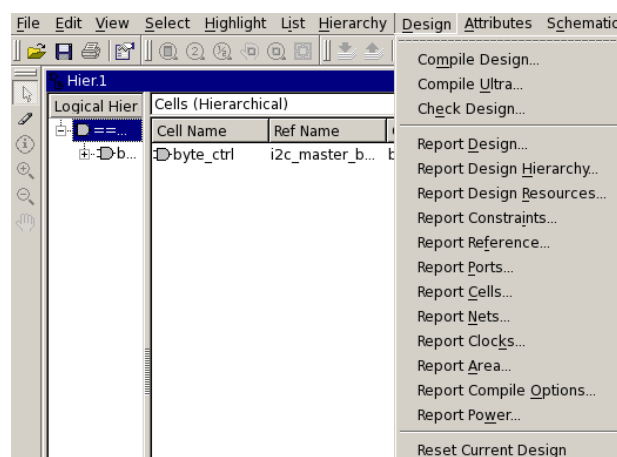


Figure 6- Check and Compile the design before saving it.

All of the executed 8 steps can be viewed in the History Window shown in figure-5 .

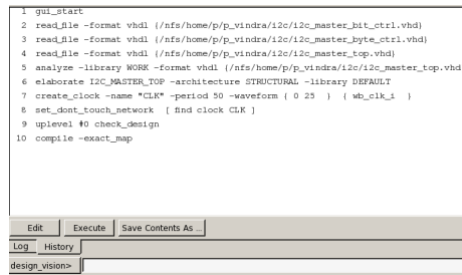


Figure 7- Design Vision History window to review the steps.

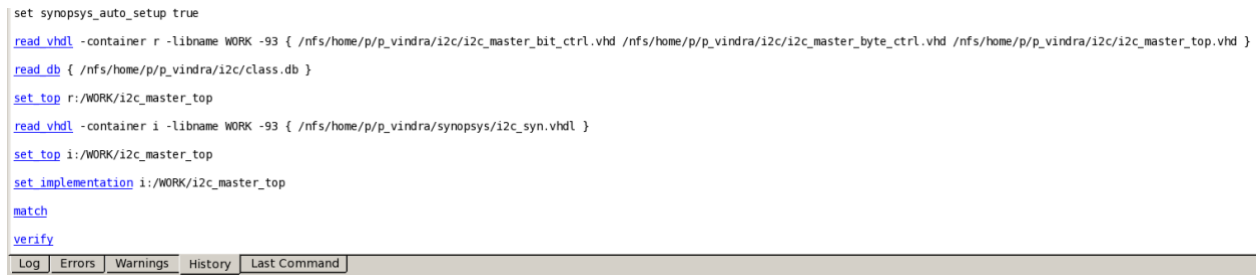


Figure 8- Synopsys Formality Tool history window to review the steps.

The entire sequence of executed steps can be observed in the Synopsys Formality Tool History window. Figure-10

3. RTL-Gate Level Equivalence Checking

3.1 Equivalence Checking in Synopsys – Formality

In this stage, the synthesized gate-level code and the provided RTL code are evaluated for equivalence using the Synopsys Formality tool, following the procedure outlined below.

Step-1: During the Guidance step, the auto setup option is enabled and utilized.

Step-2: In the Reference step, the provided RTL file is loaded as a reference, along with the provided database files. Following that, the top design for the RTL-level code is designated as the reference.

Step-3: During the implementation step, the synthesized file is loaded, and the top design for the gate-level code is assigned as the implementation, as shown in Figure- 6.

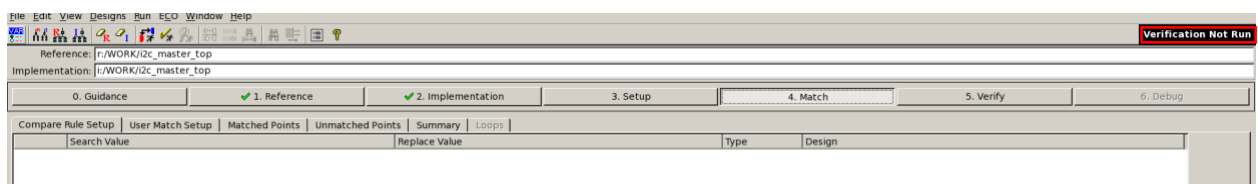


Figure 9- Synopsys Formality Tool window after setting up implementation and reference container and before matching points.

Step-4: Next, the matching points between the reference and implementation files, which were loaded in Steps 2 and 3, is examined by selecting the "Run matching" option. The results of this matching process are displayed in Figure-7.



Figure 10-Synopsys Formality Window after verification is done.

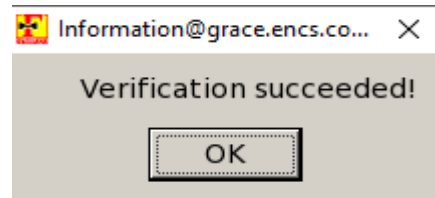


Figure 11- Synopsys Formality Window after matching is done.

Step-5: The subsequent step involves verifying the files by utilizing the 'Verify' option, which generates an output window similar to the one shown in Figure-8.

Step-6: If the two files are equivalent, the formality tool will present a prompt stating "Verification Succeeded," along with providing information on the passing points in the debug window.

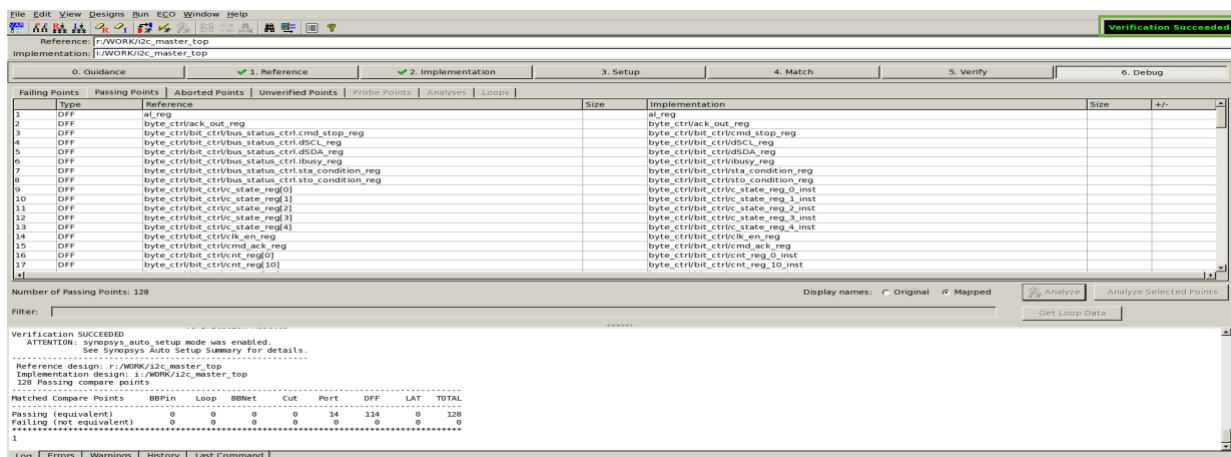


Figure 12- Verification results in Synopsys Formality tool of RTL-Synthesized Gate Level Code.

The above figure-9 confirms that the provided RTL code has been successfully synthesized without any errors and can now be used for equivalence checking with the buggy gate-level code.



Figure 13- Synopsys Formality Tool history window to review the steps.

The entire sequence of executed steps can be observed in the Synopsys Formality Tool History window. Figure-10

3.2 Equivalence Checking in Cadence – Conformal

In this step, the synthesized Gate level code and the given RTL code are compared for equivalence using Cadence Conformal tool using below procedure.

Step-1: From the File Choose 'Read Library' option and select ".db" library file from the design folder. After that, add selected file. File Format is Liberty and the File type is Both selected.

Step-2: After above step, select the Golden file and file format is VHDL selected (is depend on the file types). File type is Golden selected. Similarly, done for Revised file, but type is Revised selected.

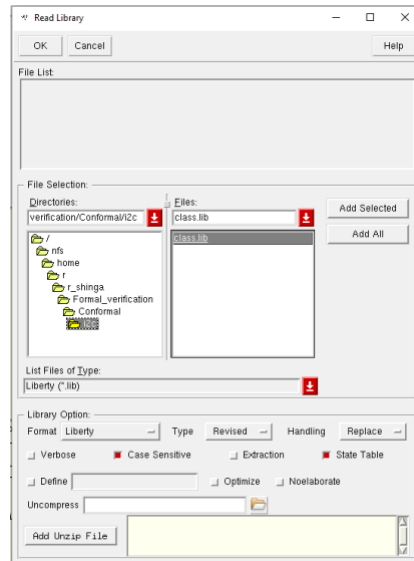


Figure 14 Reading File

Step-3: In this step, First change the mode from the “Setup” to “LEC”, then use Run tab and compare both the design.

Mapped points: SYSTEM class				
Mapped points	PI	PO	DIFF	Total
Golden	19	14	114	147
Revised	19	14	114	147

```

LEC> add compared points -all
// Command: add compared points -all
// 128 compared points added to compare list
LEC> compare -NONEQ_Print
// Command: compare -NONEQ_Print

```

Compared points	PO	DIFF	Total
Equivalent	14	114	128

Figure 15 Compared points

Step-4: Open Mapping manager.

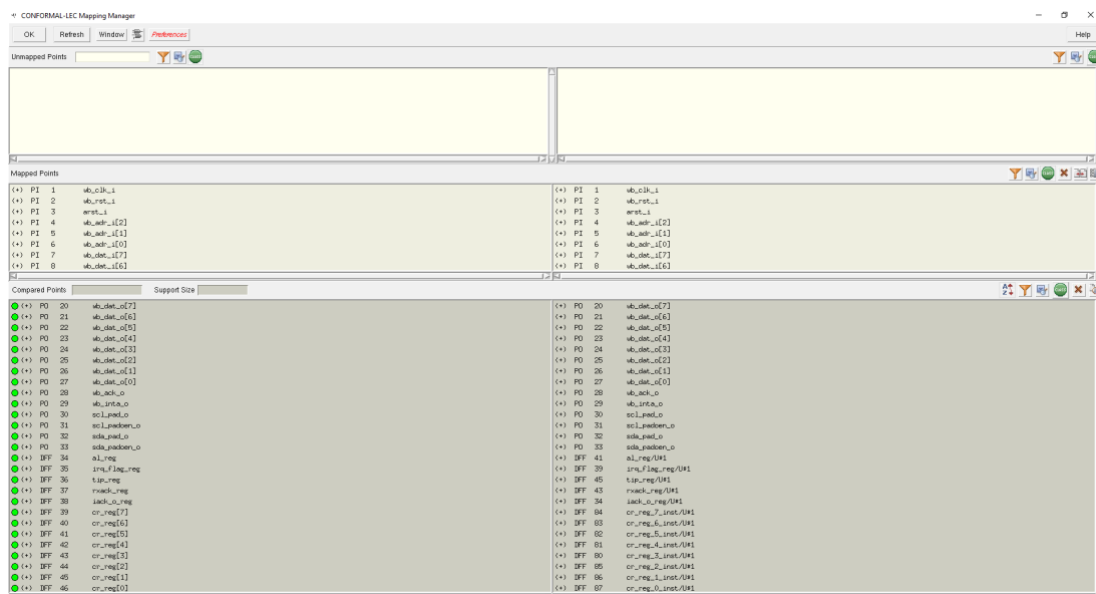


Figure 16 Mapping Manager

4. Gate-Gate Equivalence Checking

4.1 Equivalence Checking in Synopsys – Formality

In this step, Gate level code (Without bugs) file is compared with gate level file with bugs and we have tried to solve all bugs.

Step-1: First take gate level file, which is not containing any bugs as a Reference and buggy file as implementation.

Step-2: Follow all steps same, which is mentioned in the RTL to Gate level equivalence checking in Formality section.

```
-----
Reference design: r:/WORK/i2c_master_top
Implementation design: i:/WORK/i2c_master_top
94 Passing compare points
20 Failing compare points
0 Aborted compare points
14 Unverified compare points
-----
Matched Compare Points  BBPin  Loop  BBNet  Cut  Port  DFF  LAT  TOTAL
-----
Passing (equivalent)    0      0      0      0   14    80    0    94
Failing (not equivalent) 0      0      0      0    0    20    0    20
Unverified              0      0      0      0    0    14    0    14
*****
```

Figure 17 Comparison Between Buggy and Reference file in Console

0. Guidance		✓ 1. Reference	✓ 2. Implementation	3. Setup	4. Match	5. Verify	6. Debug
Failing Points	Passing Points	Aborted Points	Unverified Points	Probe Points	Analyses	Loops	
Type	Reference	Size	Implementation	Size	+/-		
1 DFF	byte_ctrl/ack_out_reg		byte_ctrl/ack_out_reg				
2 DFF	byte_ctrl/bit_ctrl/clock_en_reg		byte_ctrl/bit_ctrl/clock_en_reg				
3 DFF	byte_ctrl/bit_ctrl/cnt_reg_0_inst		byte_ctrl/bit_ctrl/cnt_reg_0_inst				
4 DFF	byte_ctrl/bit_ctrl/cnt_reg_10_inst		byte_ctrl/bit_ctrl/cnt_reg_10_inst				
5 DFF	byte_ctrl/bit_ctrl/cnt_reg_11_inst		byte_ctrl/bit_ctrl/cnt_reg_11_inst				
6 DFF	byte_ctrl/bit_ctrl/cnt_reg_12_inst		byte_ctrl/bit_ctrl/cnt_reg_12_inst				
7 DFF	byte_ctrl/bit_ctrl/cnt_reg_13_inst		byte_ctrl/bit_ctrl/cnt_reg_13_inst				
8 DFF	byte_ctrl/bit_ctrl/cnt_reg_14_inst		byte_ctrl/bit_ctrl/cnt_reg_14_inst				
9 DFF	byte_ctrl/bit_ctrl/cnt_reg_15_inst		byte_ctrl/bit_ctrl/cnt_reg_15_inst				
10 DFF	byte_ctrl/bit_ctrl/cnt_reg_1_inst		byte_ctrl/bit_ctrl/cnt_reg_1_inst				
11 DFF	byte_ctrl/bit_ctrl/cnt_reg_2_inst		byte_ctrl/bit_ctrl/cnt_reg_2_inst				
12 DFF	byte_ctrl/bit_ctrl/cnt_reg_3_inst		byte_ctrl/bit_ctrl/cnt_reg_3_inst				
13 DFF	byte_ctrl/bit_ctrl/cnt_reg_4_inst		byte_ctrl/bit_ctrl/cnt_reg_4_inst				
14 DFF	byte_ctrl/bit_ctrl/cnt_reg_5_inst		byte_ctrl/bit_ctrl/cnt_reg_5_inst				
15 DFF	byte_ctrl/bit_ctrl/cnt_reg_6_inst		byte_ctrl/bit_ctrl/cnt_reg_6_inst				
16 DFF	byte_ctrl/bit_ctrl/cnt_reg_7_inst		byte_ctrl/bit_ctrl/cnt_reg_7_inst				
17 DFF	byte_ctrl/bit_ctrl/cnt_reg_8_inst		byte_ctrl/bit_ctrl/cnt_reg_8_inst				
18 DFF	byte_ctrl/bit_ctrl/cnt_reg_9_inst		byte_ctrl/bit_ctrl/cnt_reg_9_inst				
19 DFF	byte_ctrl/c_state_reg_1_inst		byte_ctrl/c_state_reg_1_inst				
20 DFF	byte_ctrl/c_state_reg_2_inst		byte_ctrl/c_state_reg_2_inst				

Figure 18 Not equivalent Points

Step-3: Check all points are verified if not then re-run and verify all points, here after verifying all, total 34 bugs found.

Type	Reference	Size	Implementation	Size	+/-
1 DFF	byte_ctrl/ack_out_reg		byte_ctrl/ack_out_reg		
2 DFF	byte_ctrl/bit_ctrl/clock_en_reg		byte_ctrl/bit_ctrl/clock_en_reg		
3 DFF	byte_ctrl/bit_ctrl/cnt_reg_0_inst		byte_ctrl/bit_ctrl/cnt_reg_0_inst		
4 DFF	byte_ctrl/bit_ctrl/cnt_reg_1_inst		byte_ctrl/bit_ctrl/cnt_reg_1_inst		
5 DFF	byte_ctrl/bit_ctrl/cnt_reg_2_inst		byte_ctrl/bit_ctrl/cnt_reg_2_inst		
6 DFF	byte_ctrl/bit_ctrl/cnt_reg_3_inst		byte_ctrl/bit_ctrl/cnt_reg_3_inst		
7 DFF	byte_ctrl/bit_ctrl/cnt_reg_4_inst		byte_ctrl/bit_ctrl/cnt_reg_4_inst		
8 DFF	byte_ctrl/bit_ctrl/cnt_reg_5_inst		byte_ctrl/bit_ctrl/cnt_reg_5_inst		
9 DFF	byte_ctrl/bit_ctrl/cnt_reg_6_inst		byte_ctrl/bit_ctrl/cnt_reg_6_inst		
10 DFF	byte_ctrl/bit_ctrl/cnt_reg_7_inst		byte_ctrl/bit_ctrl/cnt_reg_7_inst		
11 DFF	byte_ctrl/bit_ctrl/cnt_reg_8_inst		byte_ctrl/bit_ctrl/cnt_reg_8_inst		
12 DFF	byte_ctrl/bit_ctrl/cnt_reg_9_inst		byte_ctrl/bit_ctrl/cnt_reg_9_inst		
13 DFF	byte_ctrl/bit_ctrl/cnt_reg_10_inst		byte_ctrl/bit_ctrl/cnt_reg_10_inst		
14 DFF	byte_ctrl/bit_ctrl/cnt_reg_11_inst		byte_ctrl/bit_ctrl/cnt_reg_11_inst		
15 DFF	byte_ctrl/bit_ctrl/cnt_reg_12_inst		byte_ctrl/bit_ctrl/cnt_reg_12_inst		
16 DFF	byte_ctrl/bit_ctrl/cnt_reg_13_inst		byte_ctrl/bit_ctrl/cnt_reg_13_inst		
17 DFF	byte_ctrl/bit_ctrl/cnt_reg_14_inst		byte_ctrl/bit_ctrl/cnt_reg_14_inst		
18 DFF	byte_ctrl/bit_ctrl/cnt_reg_15_inst		byte_ctrl/bit_ctrl/cnt_reg_15_inst		
19 DFF	byte_ctrl/bit_ctrl/sta_condition_reg		byte_ctrl/bit_ctrl/sta_condition_reg		
20 DFF	byte_ctrl/c_state_reg_0_inst		byte_ctrl/c_state_reg_0_inst		
21 DFF	byte_ctrl/c_state_reg_1_inst		byte_ctrl/c_state_reg_1_inst		
22 DFF	byte_ctrl/c_state_reg_2_inst		byte_ctrl/c_state_reg_2_inst		

23	DFF	byte_ctrl/core_cmd_reg_0_inst	byte_ctrl/core_cmd_reg_0_inst		
24	DFF	byte_ctrl/core_cmd_reg_1_inst	byte_ctrl/core_cmd_reg_1_inst		
25	DFF	byte_ctrl/core_cmd_reg_2_inst	byte_ctrl/core_cmd_reg_2_inst		
26	DFF	byte_ctrl/core_cmd_reg_3_inst	byte_ctrl/core_cmd_reg_3_inst		
27	DFF	byte_ctrl/core_txd_reg	byte_ctrl/core_txd_reg		
28	DFF	byte_ctrl/host_ack_reg	byte_ctrl/host_ack_reg		
29	DFF	cr_reg_3_inst	cr_reg_3_inst		
30	DFF	cr_reg_4_inst	cr_reg_4_inst		
31	DFF	cr_reg_5_inst	cr_reg_5_inst		
32	DFF	cr_reg_6_inst	cr_reg_6_inst		
33	DFF	cr_reg_7_inst	cr_reg_7_inst		
34	DFF	irq_flag_reg	irq_flag_reg		

Figure 19 All not Equivalent Point

Step-4: Use Diagnosis tools to check error candidates. For the first Bugs, total 4 error candidates we found. Logically we have analyzed that U104 NAND instead of NOR.

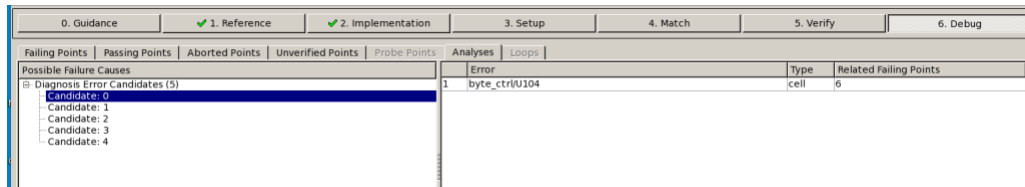


Figure 20 Error Candidates using Diagnosis tool

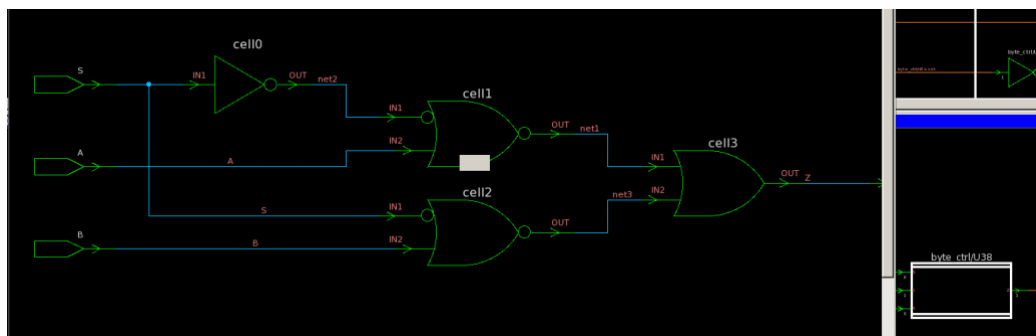


Figure 21 Expanding any module

Step-5: Check where bug is located and then edit NR2 -> ND2. After this change 'bit_ctrl/ack_out_reg' and 'byte_ctrl/core_txd_reg' bugs solved. Similarly steps are repeated for remaining portion.

```

U102 : LV port map( A => n117, Z => n190 );
U103 : A06 port map( A => n49, B => ack_in, C => n26, Z => n117 );
U104 : NR2 port map( A => n99, B => n31, Z => n26 );

```

Figure 22 Finding error candidates in code

	Type	Reference	Size	Implementation	Size	+
1	DFF	byte_ctrbit_ctrclk_en_reg		byte_ctrbit_ctrclk_en_reg		
2	DFF	byte_ctrbit_ctrclk_reg_0_inst		byte_ctrbit_ctrclk_reg_0_inst		
3	DFF	byte_ctrbit_ctrclk_reg_10_inst		byte_ctrbit_ctrclk_reg_10_inst		
4	DFF	byte_ctrbit_ctrclk_reg_11_inst		byte_ctrbit_ctrclk_reg_11_inst		
5	DFF	byte_ctrbit_ctrclk_reg_12_inst		byte_ctrbit_ctrclk_reg_12_inst		
6	DFF	byte_ctrbit_ctrclk_reg_13_inst		byte_ctrbit_ctrclk_reg_13_inst		
7	DFF	byte_ctrbit_ctrclk_reg_14_inst		byte_ctrbit_ctrclk_reg_14_inst		
8	DFF	byte_ctrbit_ctrclk_reg_15_inst		byte_ctrbit_ctrclk_reg_15_inst		
9	DFF	byte_ctrbit_ctrclk_reg_1_inst		byte_ctrbit_ctrclk_reg_1_inst		
10	DFF	byte_ctrbit_ctrclk_reg_2_inst		byte_ctrbit_ctrclk_reg_2_inst		
11	DFF	byte_ctrbit_ctrclk_reg_3_inst		byte_ctrbit_ctrclk_reg_3_inst		
12	DFF	byte_ctrbit_ctrclk_reg_4_inst		byte_ctrbit_ctrclk_reg_4_inst		
13	DFF	byte_ctrbit_ctrclk_reg_5_inst		byte_ctrbit_ctrclk_reg_5_inst		
14	DFF	byte_ctrbit_ctrclk_reg_6_inst		byte_ctrbit_ctrclk_reg_6_inst		
15	DFF	byte_ctrbit_ctrclk_reg_7_inst		byte_ctrbit_ctrclk_reg_7_inst		
16	DFF	byte_ctrbit_ctrclk_reg_8_inst		byte_ctrbit_ctrclk_reg_8_inst		
17	DFF	byte_ctrbit_ctrclk_reg_9_inst		byte_ctrbit_ctrclk_reg_9_inst		
18	DFF	byte_ctrbit_ctrclk_condition_reg		byte_ctrbit_ctrclk_condition_reg		
19	DFF	byte_ctrbit_ctrclk_reg_0_inst		byte_ctrbit_ctrclk_reg_0_inst		
20	DFF	byte_ctrbit_ctrclk_reg_1_inst		byte_ctrbit_ctrclk_reg_1_inst		
21	DFF	byte_ctrbit_ctrclk_reg_2_inst		byte_ctrbit_ctrclk_reg_2_inst		
22	DFF	byte_ctrbit_ctrclk_reg_3_inst		byte_ctrbit_ctrclk_reg_3_inst		
23	DFF	byte_ctrbit_ctrclk_reg_4_inst		byte_ctrbit_ctrclk_reg_4_inst		
24	DFF	byte_ctrbit_ctrclk_reg_5_inst		byte_ctrbit_ctrclk_reg_5_inst		
25	DFF	byte_ctrbit_ctrclk_reg_6_inst		byte_ctrbit_ctrclk_reg_6_inst		
26	DFF	byte_ctrbit_ctrclk_reg_7_inst		byte_ctrbit_ctrclk_reg_7_inst		
27	DFF	byte_ctrbit_ctrclk_reg_8_inst		byte_ctrbit_ctrclk_reg_8_inst		
28	DFF	byte_ctrbit_ctrclk_reg_9_inst		byte_ctrbit_ctrclk_reg_9_inst		
29	DFF	byte_ctrbit_ctrclk_reg_10_inst		byte_ctrbit_ctrclk_reg_10_inst		
30	DFF	byte_ctrbit_ctrclk_reg_11_inst		byte_ctrbit_ctrclk_reg_11_inst		
31	DFF	byte_ctrbit_ctrclk_reg_12_inst		byte_ctrbit_ctrclk_reg_12_inst		
32	DFF	byte_ctrbit_ctrclk_reg_13_inst		byte_ctrbit_ctrclk_reg_13_inst		

Figure 23 Remaining Bugs

4.2 Equivalence Checking in Cadence – Conformal

Cadence Conformal LEC tool is used to check for the equivalence between golden (Synthesized Gate Level) design and revised (Gate level Buggy) design. The following steps are to be followed to perform the equivalence check: -

Step-1: From the File Choose ‘Read Library’ option and select “.db” library file from the design folder. After that, add selected file. File Format is Liberty and the File type is Both selected.

Step-2: Read both the design files using “Read Design Tab” in the toolbar as shown in figure-1.

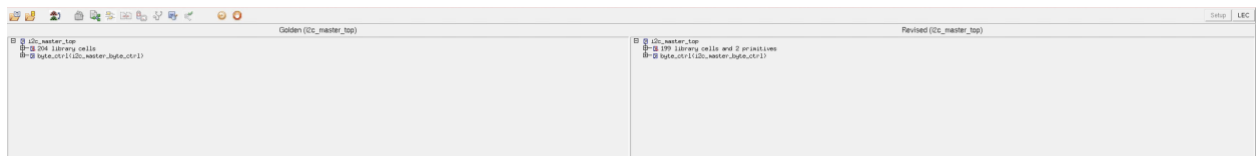


Figure 24-Golden and Revised design files after been read.

Step-3: In this step, First change the mode from the “Setup” to “LEC”, then use Run tab and compare both the design as depicted in figure- 2 and which provides the list of all equivalent and non-equivalent points between both the designs.

Mapped points: SYSTEM class				
Mapped points	PI	PO	DFF	Total
Golden	19	14	114	147
Revised	19	14	114	147
LEC> add compared points -all				
// Command: add compared points -all				
// 128 compared points added to compare list				
LEC> compare				
// Command: compare				
Compared points	PO	DFF	Total	
Equivalent	14	80	94	
Non-equivalent	0	34	34	

Figure 25-Mapped point and list of possible equivalent & non-equivalent points.

Step-4: Subsequently the mapping manager is used to see the compare points available among these two designs and there are thirty-four non-equivalent points present as shown in Figure-.

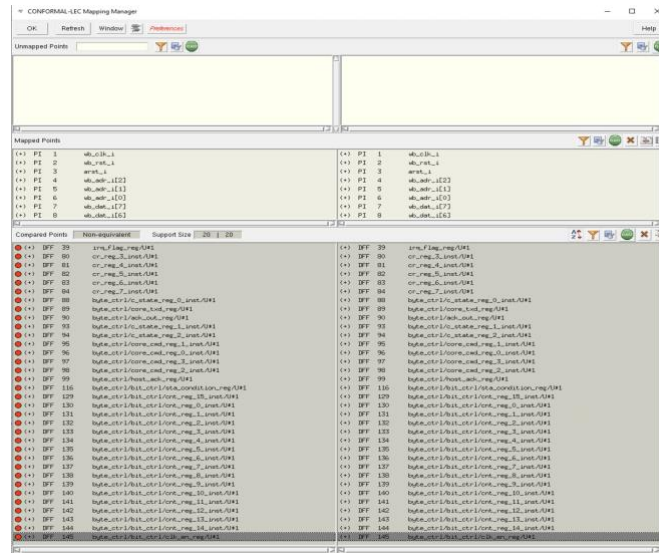


Figure 26- Mapping manger compared points filtered only non-equivalent points.

Step-5: Next, the non-equivalent compare points can be diagnosed using the diagnosis manager which provides us info about the possible error candidates and resolution for it if any as shown in figure-4 and figure-5.

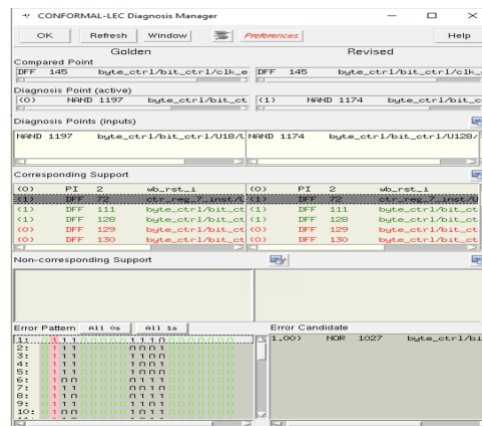


Figure 27 Diagnosis Manager pointing Error candidates.

```
LEC) diagnose 145 -golden // gate byte_ctrl/bit_ctrl/clock_en_reg/U#1
// Command: diagnose 145 -golden // gate byte_ctrl/bit_ctrl/clock_en_reg/U#1
Diagnosis for Non-equivalent key points:
(G) + 145 DFF /byte_ctrl/bit_ctrl/clock_en_reg/U#1
(R) + 145 DFF /byte_ctrl/bit_ctrl/clock_en_reg/U#1

Diagnosis points: [DATA]
(G) + 1197 NAND /byte_ctrl/bit_ctrl/U18/U#1
(R) + 1174 NAND /byte_ctrl/bit_ctrl/U128/U#1

The diagnosis point can be corrected by changing the following gates:
=====
Correction  ID (R)  Type  Name
=====
GATE_CHANGE  1027  NOR  /byte_ctrl/bit_ctrl/U134/U#1
change to      NAND
=====
```

Figure 28 Description of the error

Step-6: Right clicking on the error candidate provided us with option to view the schematics of the possible error candidate and ability to compare golden and revised source code as shown in figure-6.

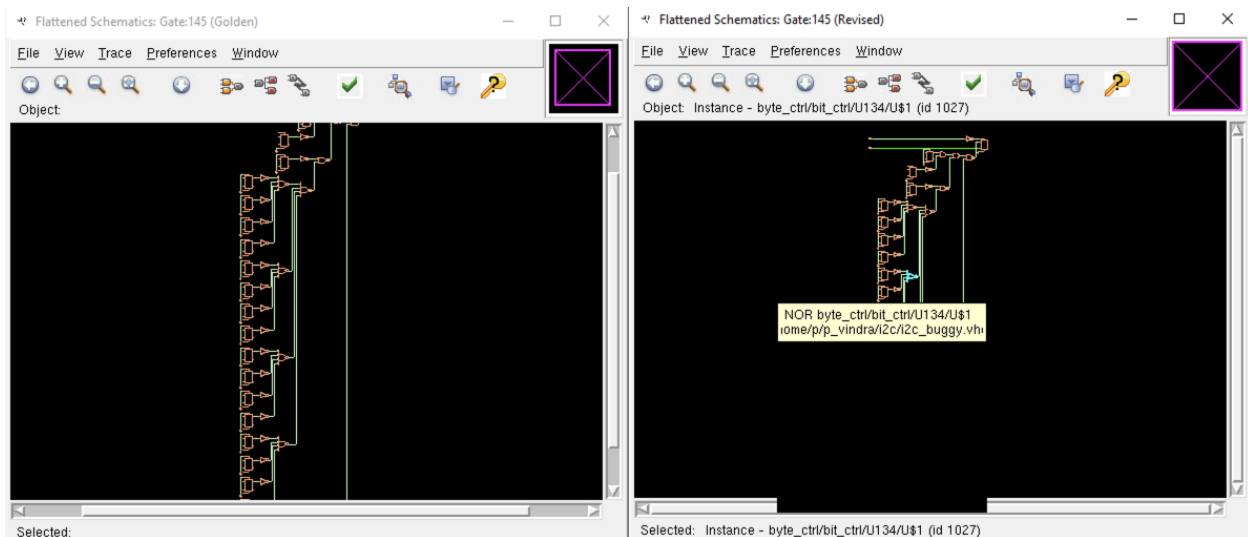


Figure 29-Schematics comparison of a particular error candidate.

Step-7: By following the above steps almost every error was resolved except one error as shown in figure -7.

PI	Support Size
(*) PI 1	byte_ctrl/host_ack_reg/U\$1
(*) PI 2	byte_ctrl/host_ack_reg/U\$1
(*) PI 3	byte_ctrl/host_ack_reg/U\$1
(*) PI 4	byte_ctrl/host_ack_reg/U\$1
(*) PI 5	byte_ctrl/host_ack_reg/U\$1
(*) PI 6	byte_ctrl/host_ack_reg/U\$1
(*) PI 7	byte_ctrl/host_ack_reg/U\$1
(*) PI 8	byte_ctrl/host_ack_reg/U\$1

Figure 30- Unresolved error of host_ack_reg in the byte_ctrl module.

5. Verification and Bug Hunting Process

To do the verification process with the tools Synopsys Formality and Cadence Conformal we adopted two approaches such as the following: -

1. Top-to-Top Module Equivalence Checking: -

In this approach we performed equivalence checking between the golden and revised design in terms of the top module i.e., i2c_master_top.

In this approach all the non-equivalent points were displayed together in the tool window.

The advantage in this approach is that in one single window we were able to see all the non-equivalent points together and we were able to establish a relationship between each module that exist in the design.

One major disadvantage of this approach is that at times we lost track of the solved non-equivalent point and we had to revert back to see the resolved points.

2. Module-by-Module Equivalence Checking: -

In this approach we performed equivalence checking between the golden and revised design in terms of the module by module i.e., step by step hierarchical comparison of all the modules (i2c_master_bit_ctrl, i2c_master_bit_ctrl_DW01_dec_0, i2c_master_byte_ctrl and i2c_master_top)

The advantage of this approach is many we were able to understand every module in depth and was able to analyze the flow. Moreover, this hierarchical flow was useful in solving the bugs systematically and lead to least confusion in terms of identifying and investigating the non-equivalent point at times.

In general, both the approaches were tried during the bug hunting process. Even though both approaches resulted in same number of non-equivalent points or bug count but the process of doing them can be an essential part of learning process.

5.1 Analysis of resolved bugs.

S.No	<u>Non-Equivalent Points</u>	<u>Possible Error Candidates</u>	<u>Fix</u>
1.	DFF /byte_ctrl/bit_ctrl/clk_en_reg/U\$1	U134- NOR Gate	Gate change to NAND
2.	DFF /byte_ctrl/bit_ctrl/cnt_reg_0_inst/U\$1	The Fixing of the non-equivalent point 1 will result in diagnosing the corresponding non-equivalent points as well.	
3.	DFF /byte_ctrl/bit_ctrl/cnt_reg_1_inst/U\$1		
4.	DFF /byte_ctrl/bit_ctrl/cnt_reg_2_inst/U\$1		
5.	DFF /byte_ctrl/bit_ctrl/cnt_reg_3_inst/U\$1		
6.	DFF /byte_ctrl/bit_ctrl/cnt_reg_4_inst/U\$1		
7.	DFF /byte_ctrl/bit_ctrl/cnt_reg_5_inst/U\$1		
8.	DFF /byte_ctrl/bit_ctrl/cnt_reg_6_inst/U\$1		
9.	DFF /byte_ctrl/bit_ctrl/cnt_reg_7_inst/U\$1		
10.	DFF /byte_ctrl/bit_ctrl/cnt_reg_8_inst/U\$1		
11.	DFF /byte_ctrl/bit_ctrl/cnt_reg_9_inst/U\$1		
12.	DFF /byte_ctrl/bit_ctrl/cnt_reg_10_inst/U\$1		
13.	DFF /byte_ctrl/bit_ctrl/cnt_reg_11_inst/U\$1		
14.	DFF /byte_ctrl/bit_ctrl/cnt_reg_12_inst/U\$1		
15.	DFF /byte_ctrl/bit_ctrl/cnt_reg_13_inst/U\$1		
16.	DFF /byte_ctrl/bit_ctrl/cnt_reg_14_inst/U\$1		
17.	DFF /byte_ctrl/bit_ctrl/cnt_reg_15_inst/U\$1		
18.	DFF /byte_ctrl/bit_ctrl/sta_condition_reg/U\$1	U122- NOR Gate	Gate change to AND.
19.	DFF /byte_ctrl/ack_out_reg/U\$1	U104- NAND Gate	Gate change to NOR
20.	DFF /byte_ctrl/core_txd_reg/U\$1	The Fixing of the non-equivalent point 19 will result in diagnosing the corresponding non-equivalent points as well.	
21.	DFF /byte_ctrl/c_state_reg_0_inst/U\$1		
22.	DFF /byte_ctrl/c_state_reg_1_inst/U\$1	U29- NAND Gate	Gate change to NOR
23.	DFF /byte_ctrl/c_state_reg_2_inst/U\$1	The Fixing of the non-equivalent point 22 will result in diagnosing the corresponding non-equivalent points as well.	
24.	DFF /byte_ctrl/core_cmd_reg_0_inst/U\$1		
25.	DFF /byte_ctrl/core_cmd_reg_1_inst/U\$1		
26.	DFF /byte_ctrl/core_cmd_reg_2_inst/U\$1		
27.	DFF /byte_ctrl/core_cmd_reg_3_inst/U\$1		
28.	DFF /irq_flag_reg/U\$1	U312- AND Gate	Gate change to NOR
29.	DFF /cr_reg_3_inst/U\$1	U245-NAND Gate	

30.	DFF /cr reg 4 inst/U\$1		Gate change to OR
31.	DFF /cr reg 5 inst/U\$1		
32.	DFF /cr reg 6 inst/U\$1		
33.	DFF /cr reg 7 inst/U\$1		
34.	DFF /byte ctrl/host ack reg/U\$1	Unresolved Non-Equivalent Point	

5.2 Analysis of the Un-Resolved Bug

Analysis of 'byte_ctrl/host ack reg' bug.

1. logical equivalence checking between implementation and reference.

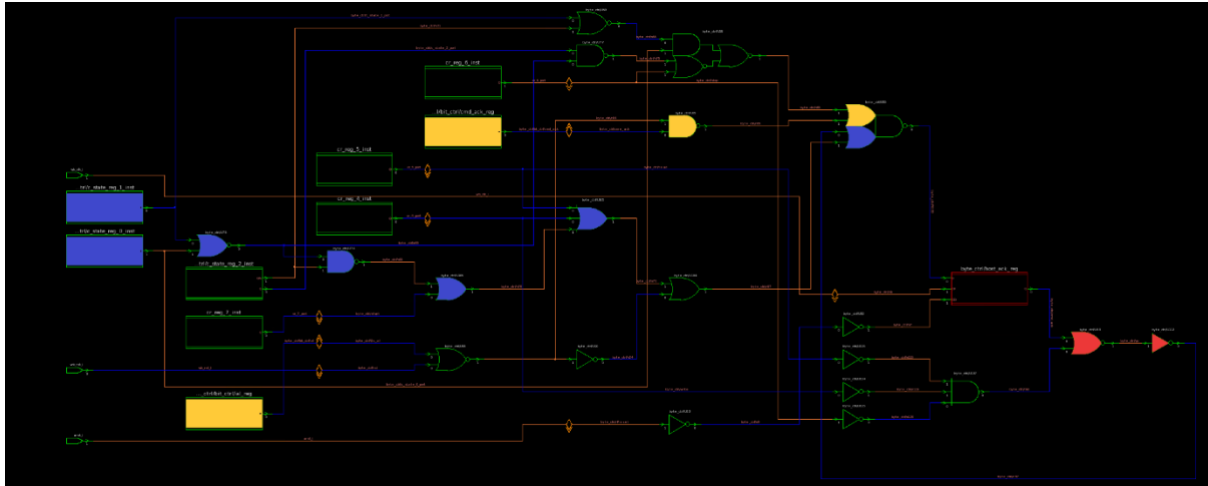


Figure 31 Reference of Unsolved Bug

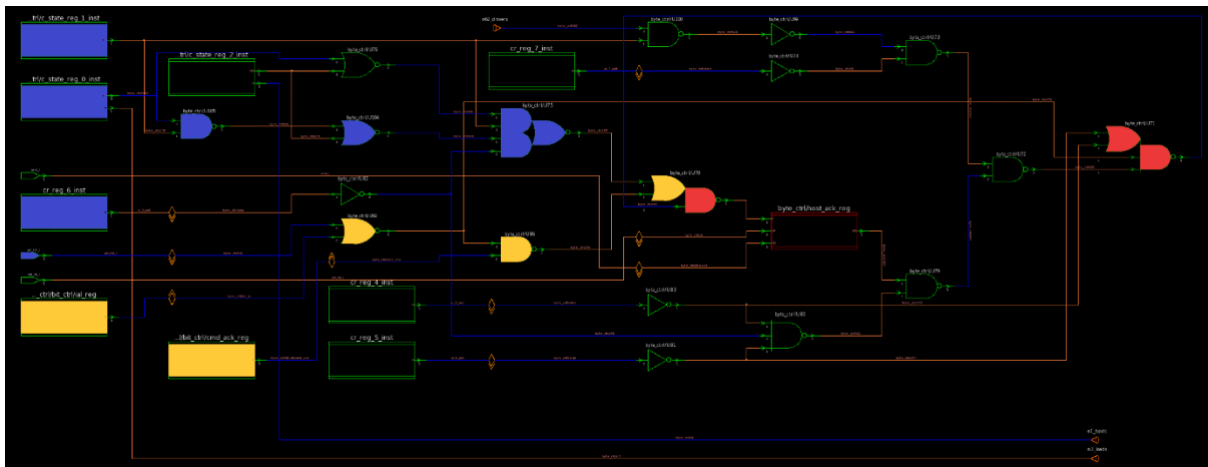


Figure 32 Implementation of unsolved bug

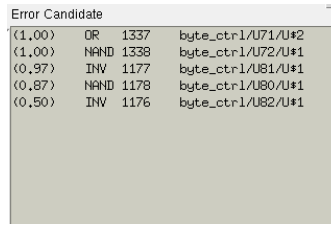
Here, we analyzed that, circuit is logically equivalence for the blue and yellow colour paths, that are responsible to drive the 'host_ack_reg' register. One of possible error candidate by first look.

But we are not able to confirm about the feedback and not able check logical equivalence in between.

2. Use Formality software diagnosis tools to find error candidate.

Error candidate number	Name
Candidate 0	Byte_ctrl/U72
Candidate 1	Byte_ctrl/U71
Candidate 2	Byte_ctrl/U70
Candidate 3	Byte_ctrl/host_ack reg

3. Use Conformal Diagnosis tool to find error candidate.



Error Candidate			
(1,00)	OR	1337	byte_ctrl1/U71/U#2
(1,00)	NAND	1338	byte_ctrl1/U72/U#1
(0,97)	INV	1177	byte_ctrl1/U81/U#1
(0,87)	NAND	1178	byte_ctrl1/U80/U#1
(0,50)	INV	1176	byte_ctrl1/U82/U#1

Figure 33- Error Candidates for the bug

After exhausting all available methods, the bug remains unresolved. We attempted to identify the bug using both Conformal and Formality tools, thoroughly examining all potential error candidates. Unfortunately, we were unable to find a solution. Therefore, we have concluded that if there are significant differences between the implementation and reference circuits, resolving the bugs becomes challenging, and in some cases, even impossible without RTL design.

6. Analysis of Verification results

We have used two software for verification purpose, Synopsys Formality and Cadence Conformal. In Formality, after setting the gate level synthesized design as reference and gate level buggy design as implementation, schematics of both were compared for rectifying the bugs. In formality the total number of thirty-four non-equivalent points. As we progressed with bug resolving, we were able to resolve 33 bugs with few tweaks but unfortunately at the end we got stuck with one bug.

In Conformal tool, once the design files are set in the golden and revised respectively, diagnosis were performed. The total number of non-equivalent points were same thirty-four as well. They were diagnosed by comparing the schematics and following the Conformal recommendation in certain errors. We were able to solve 33 out 34 non-equivalent points were only able to be resolved and we again got stuck with the same bug.

7. Comparison of Synopsys Formality and Cadence Conformal

Interesting aspect to be noted in aspects of both software in terms of the displayed non-equivalent points were that error candidates provided by both the tools were different and quite baffling at times. So, whenever there were different error candidates, we had to perform more analysis in terms of source code as well schematics on that particular to resolve it.

When it comes to Formality tool, the one of the things which was happening time and again was it getting crashed. So, it was necessary to save each session in which changes were made on timely basis. These saved sessions were stored in same directory as the design files and at times it was becoming confusing when it comes to restore a particular session associating to a certain change. Both Formality and Conformal were equally used in the process of bug hunting. There is no doubt that both the tools are world class and trusted by the industry's best but quite interesting when we compared both the tools in terms of certain features such as User interface, User friendly, Evaluation, Schematic wise and Operation time: -

Tool/Feature	User Friendly	User Interface	Evaluation	Schematic Wise	Operation Time
Synopsys Formality	Most Preferred	Most Preferred	Least Preferred	Most Preferred	Most Preferred
Cadence Conformal	Least Preferred	Least Preferred	Most Preferred	Least Preferred	Least Preferred

The most important point to be noted down when it comes to operation time required by both the tools with respect to loading and removing the updated file time and again is quite easier in formality in compared to conformal tool.

8. Conclusion

The main goal of this project was to use Equivalence Checking, a formal verification method, to verify the correctness of two designs at different levels of abstraction. Equivalence Checking is typically employed during Implementation Verification. The process begins by synthesizing the RTL design using Synopsys Design Vision software, which generates a Gate level netlist. To validate the synthesis process, the gate level netlist is then compared with the RTL netlist in Synopsys Formality and Cadence Conformal using Equivalence Checking. Once this verification is successful, the next step involves applying Equivalence Checking between the synthesized gate level netlist and the Buggy gate level netlist. This step aims to identify any bugs or discrepancies, as both netlists are at the same level of abstraction.

Challenges

But in our case, we were not able to attain 100% verification success using formality as well as conformal. We failed to resolve one bug in the end, but we tried to analyze the bug in numerous ways and during that process we were able to understand the tools better and got familiarized about the concept of equivalence checking in depth.

In future we are that with the experience gained during this whole project can come handy at any point and be helpful in future learning as well.

Comment

If there are significant differences between the implementation and reference circuits, resolving the bugs becomes challenging, and in some cases, even impossible without RTL design.

9. References

1. Killer software: 4 lessons from the deadly 737 MAX crashes
<https://www.fiercееlectronics.com/electronics/killer-software-4-lessons-from-deadly-737-max-crashes>
2. N° 33–1996: Ariane 501 - Presentation of Inquiry Board report
https://www.esa.int/Newsroom/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report
3. Prof Dr. S. Tahar, COEN 6551 Formal Hardware Verification, Lecture notes, Summer term 2023.
4. I2C Master Core Description
<https://opencores.org/projects/i2c>