Synthesis and Verification of Data Encryption Standard

By:

Kazem Cheshmi, Mohammad Baba, MohammadHossein Askari-Hemmat

Course Instructor:
Dr. S. Tahar

A technical Report
Submitted in partial fulfillment of the requirements of
COEN 7501
Concordia University

Spring 2013

Contents

1 Introduction **1**

2 DES Specification[1] **1**
2.1 Functional Specification 1
2.1.1 Details of Single Round 2
2.2 Behavioral Implementation 3

3 Design Synthesis : **5**

4 Equivalence Checking **8**
4.1 RTL code and Synthesized gate-level code 8
4.2 Bug Finding Techniques 9
4.3 Debugging 10
4.3.1 Bug number 1 11
4.3.2 Bug number 6 11
4.3.3 Bug number 9 12

5 Tool Comparison **13**
5.1 Synopsys Formality 13
5.2 Cadence Conformal 14
5.3 Equivalency Checking Tools 16
5.3.1 Case study 17

6 Summary and Conclusion : **20**

A Appendix A **20**

B Appendix B **21**

C Reference **23**

List of Tables

1 DES Pinout . . . 3
2 Report of Synopsys Design Compiler . . . 7
3 List of BUGs in DES implementation . . . 10
4 Input signals for U3798 . . . 11
5 The Boolean expression for bug 6 . . . 11
6 Error candidates color . . . 14
7 Tool Comparison . . . 17
8 Generating a truth table for correct gate by using patterns. . . . 19
9 Memory usage and processing time . . . 19
10 Tool candidates for the corresponding failing point. . . . 22

List of Figures

1 General Depiction of DES Encryption Algorithm[1] . . . 2
2 Single Round of DES Algorithm [1] . . . 2
3 DES Behavioral Implementation . . . 4
4 DES FSM . . . 5
5 Simulation Result . . . 5
6 DES Block . . . 6
7 Schematic view of DES . . . 7
8 Schematic view of Control Process . . . 7
9 The results of the Equivalence checking between the RTL level design and the synthesized gate-level files. (a) and (b) show the result of verification of the RTL with the synthesized un-optimized gate-level file using Formality and Conformal respectively. (c) and (d) show the result of verification of the RTL with the synthesized optimized gate-level file using Formality and Conformal respectively. . . . 9
10 The schematic view of both reference and implementation circuits for bug number 3 . . . 11
11 The schematic view of both reference and implementation circuits for bug number 6 . . . 12
12 The schematic view of both reference and implementation circuits for bug number 9 . . . 12
13 The schematic view of both reference and implementation circuits for bug number 9 . . . 13
14 Failing points . . . 13
15 Diagnose failing points . . . 14
16 Candidates for the specific failing point . . . 14
17 Failing points . . . 14
18 Failing points(Conformal) . . . 15
19 Error Candidates(Conformal) . . . 15
20 Schematic View(Conformal) . . . 15
21 Schematic View(Conformal) . . . 16
22 Report Gate view . . . 16
23 Error candidates for ”inmsg_reg_36_inst” using Formality tool. . . . 18
24 Error candidates for ”inmsg_reg_36_inst” using Conformal tool. . . . 18
25 Using pattern facility of Formality tool to find the correct substitute for the specified gates . . . 18
26 Categorizing the candidates to find the right one. . . . 19
27 Permutation Tables for DES . . . 20
28 Definition of DES S-Boxes . . . 21
29 DES Key Schedule Calculation . . . 21

1 Introduction

Decreasing transistor feature size leads to growth of number of modules inside the chip. This growth and also the high cost of defective fabrication are demanding a new verification schemes to make verifying big chips possible. Due to harsh state space explosion, functional verification cannot solely do this task and needs a new method. Formal verification is a method that can be considered as a solution. There are some obstacles for using formal methods in industry; lack of tool is one of the most important obstacle that we face. However, this is not the same for all formal methods and there are some industrial tools for some formal methods like equivalency checking.
Equivalency checking is a formal method that compares two designs formally and decides whether these two designs are equal. This technique is very useful for comparing two netlists. One of the usual applications of equivalency technique is for checking the equivalency of RTL design and implemented design netlist. The implemented design netlist could be obtained from another synthesis tool or from the same tool with optimization. In this report we are going to use two industrial tools which use equivalency checking technique to do comparison between implementation of Data Encryption Standard (DES) and its specification. These tools are Synopsys Formality and Cadence Conformal. Besides debugging the DES design we will compare these two tools based on our experiences.

The remaining of this report is organized as following. At first step and in Section 2, we explain the functional behavior and specification of DES design. The behavioral implementation of DES which is written by an HDL will be discussed afterwards. Section 3 is about synthesis of the behavioral implementation of DES. In this section we show how we synthesize it using Synopsys Design Compiler and discuss about the synthesis result. The net list that we get from this section will be used as our reference design.
Equivalency checking of the given implemented design (buggy) with the reference design obtained from previous section will be done in Section 4. In this section we introduce the techniques that is used during working with mentioned tools for finding the bugs of implementation. Furthermore we explain these techniques by illustrating some examples. A comparison between Formality and Conformal is done in Section 5. This comparison is based on the features of both tools and also based on our experience during debugging DES design. And finally we conclude our work.

2 DES Specification[1]

Data Encryption Standard (DES) is the most widely used encryption scheme is adopted by National Bureau of Standards (NBS), now the National Institute of Standards and Technology (NIST) Federal Information Processing Standard 46 (FIPS PUB 46). In 1973, the NBS issued a request for proposals for a national cipher standard. IBM submitted the results of its Tuchman-Meyer project. This project was the continuance of two previous IBM projects Feistel and LUCIFER [2]. The goal of the Tuchman-Meyer project was to develop a marketable commercial encryption product. The outcome of this effort was a refined LUCIFER that was submitted to NBS. After applying some modifications which removed the vulnerabilities of the LUCIFIER, it was adopted in 1977 as DES.

2.1 Functional Specification

The overall scheme for DES encryption is illustrated in Figure 1. As with any encryption scheme, there are two inputs to the encryption function: the plaintext to be encrypted and the key. In this case, the plaintext must be 64 bits in length and the key is 56 bits in length. Looking at the left-hand side of the figure, we can see that the processing of the plaintext proceeds in three phases. First, the 64-bit plaintext passes through an initial permutation (IP) that rearranges the bits to produce the permuted input(Appendix A Figure 27.a). This is followed by a phase consisting of 16 rounds of the same function, which involves both permutation and substitution functions. The output of the last (sixteenth) round consists of 64 bits that are a function of the input plaintext and the key. The left and right halves of the output are swapped to produce the preoutput. Finally, the preoutput is passed through a permutation (IP^{-1}) that is the inverse of the initial permutation function, to produce the 64-bit ciphertext(Appendix A Figure 27.b).

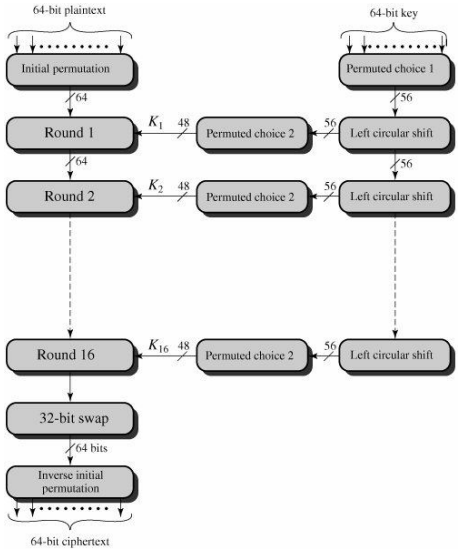


Figure 1: General Depiction of DES Encryption Algorithm[1]

2.1.1 Details of Single Round

Each round as depicted in Figure 1 is composed of two parts, data and key generation. In data generation part, 64 bits of data resulted from previous round gets as input and 64 bits data generates for the next round. This similarly happens for the key part, except that it gets 56 bits as input and generates 56 bits for the next round.

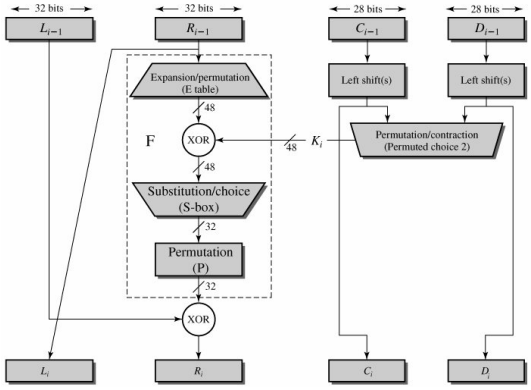


Figure 2: Single Round of DES Algorithm [1]

Data Generation: Figure 2 shows the internal structure of a single round. Again, begin by focusing on the left-hand side of the diagram. The left and right halves of each 64-bit intermediate value are treated as separate 32-bit quantities, labeled L (left) and R (right). The overall processing at each round can be summarized in the following formulas:

$$L_i = R_i - 1$$

$$R_i = L_i - 1xF(R_i - 1, K_i)$$

The round key K_i is 48 bits. The R input is 32 bits. This R input is first expanded to 48 bits by using a table that defines a permutation plus an expansion that involves duplication of 16 of the R bits ((Appendix A

Figure 27.c)). The resulting 48 bits are XORed with K_i. This 48-bit result passes through a substitution function(Appendix A Figure 28.a) that produces a 32-bit output, which is permuted as defined by Appendix A Figure 27.d .

Key Generation: Returning to Figure 1 and 2, we see that a 64-bit key is used as input to the algorithm. The bits of the key are numbered from 1 through 64; every eighth bit is ignored. The key is first subjected to a permutation governed by a table labeled Permuted Choice One, indicated in Appendix A Figure 29.b . The resulting 56-bit key is then treated as two 28-bit quantities, labeled C0 and D0. At each round, $C_i - 1$ and $D_i - 1$ are separately subjected to a circular left shift, or rotation, of 1 or 2 bits, as governed by Appendix A Figure 29.d. These shifted values serve as input to the next round. They also serve as input to Permuted Choice Two (Appendix A Figure 29.c), which produces a 48-bit output that serves as input to the function $F(R_i - 1, Ki)$.

2.2 Behavioral Implementation

In this section behavioral implementation of DES is discussed. Block diagram of DES module is illustrated in Figure 3. Figure 3 illustrates these block as described in the reference VHDL code. A brief description of each port is provided in Table 1. As you can see in Table 1, this module has 6 inputs and 4 outputs.

Port	Size(Bits)	Direction	Description
indata	[63:0]	input	Input data for Encryption
inkey	[63:0]	input	Key for Encryption
decipher	1	input	If ds=1 then Encryption/Decryption starts when decopher=0/1
ds	1	input	Enable Encryption/Decryption
clk	1	input	System Clock
rst	1	input	System Reset
outdata	[63:0]	input	Encrypted/Decrypted data
rdy_next_next_cycle	1	output	Output will be ready in two clock cycles - optional signal
rdy_next_cycle	1	output	Output will be ready in one clock cycle - optional signal
rdy	1	output	Output Ready signal

Table 1: DES Pinout

This module implements the DES 56-bit Key Block Cypher. It expects to receive the 64-bitdata block to be encrypted or decrypted on the indata bus, and the 64-bit key on the inKey bus. When the ds signal is high, encryption/decryption begins. Encoding and decoding operations are performed in 16 clocks per block.As depicted in Figure 3 , DES contains 3 major blocks. These blocks are implemented as process in the reference VHDL code.

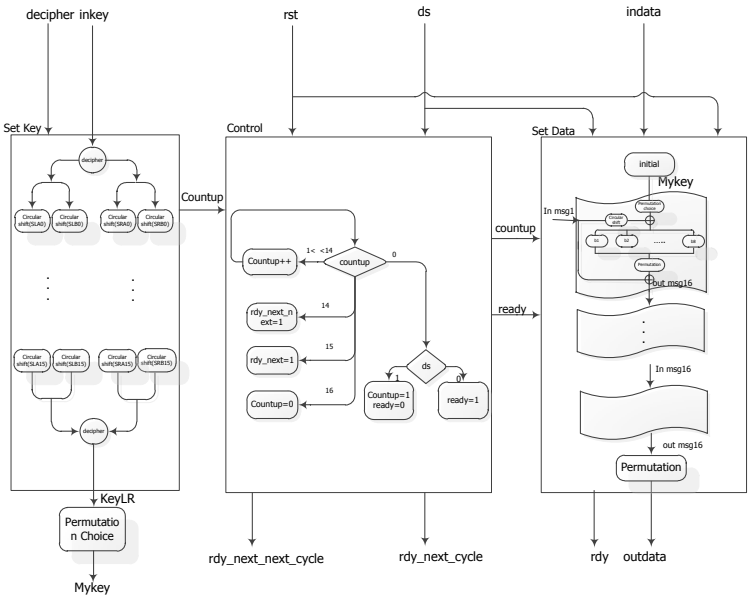


Figure 3: DES Behavioral Implementation

If the decipher signal is low when the ds signal is raised, the operation will be encryption. If the decipher signal is high when the ds signal goes high, the operation will be decryption. With each clock cycle, one round of encryption is performed. After 16 rounds, the resulting message block is presented on the outdata bus and the rdy signal is set high. DES module has 3 major blocks:

1. SetKey
2. SetData
3. Control

In SetKey process after registering the round key, the round key is simply shifted by the necessary number of bits. All of the round keys are set at once. After that the KEY_L and KEY_R are registered with the appropriate round key value. The next round of encryption can be easily determined by the current value of countup which is an input signal for this process.
In SetData process the message word will be loaded for the next encryption round. As in SetKey process, the data must be taken from the input ports on the first round. For all other rounds, the data value is taken from the outmsg signal. This signal is produced by combinatorial logic.
In Control process, the rdy signal and countup signal are calculated. Although countup signal is defined as 17 bit signal but the code only takes care of its value from 0 to 15. This issue will be more discussed in Design Synthesis section.
Among these three process, Control process is implemented as a combinational block and the other two are implemented as a sequential blocks.

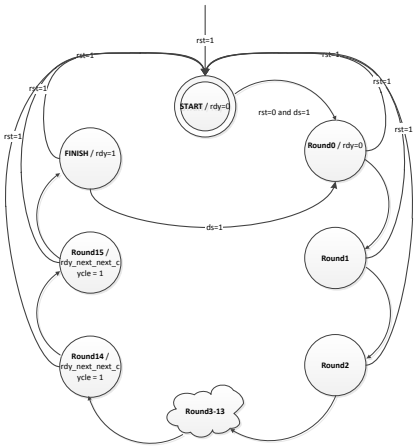


Figure 4: DES FSM

Figure 4 illustrates the FSM of DES. The variables for this Figure are obtained from reference VHDL code. In this FSM, ”reset” is an asynchronous signal for reverting the system to initial state. By zeroing this signal depends on the value of ”data_ready” signal, the first round of DES starts. After that, by each rising edge of the clock, a state transition happens. Inside each round state as specified in section 2.2, both data and key generations perform. For the key generation part of each round, depends on the ”decipher” input, ciphering or deciphering can be performed. According to this input, each round state can be split into two states. However, due to simplifying the FSM we use one state for both values of ”decipher”. In this section we have simulated the design with the provided test bench code. We wanted to make sure that the design works fine. We used ModelSim as the simulator. Final output data was compared with the online DES software which can be found here [8]. In Figure 5 you can see the simulation results.

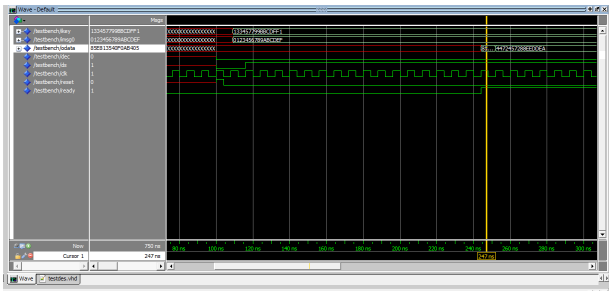


Figure 5: Simulation Result

3 Design Synthesis :

Systems can be described at different abstraction level. Register Transfer Level (RTL) is an abstraction level that describes a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the combinational logical operations performed on those signals[7] while with a gate level description the design is represented as a netlist of a basic logical gates as well as flipflops and latches. However, the process by which the design is transformed from one representation to another at a lower abstraction level or from a behavioral representation into a structural representation at the same level

is called synthesis.

In our project, we used the Synopsys design compile(DC) tool to synthesize the RTL code of the DES design to its logic gate level description which will use later in the equivalence checking phase. The synthesis process can be done according to Reference manual [6]. We also set the clock period to 50ns and did optimization by setting *maximum area* to zero which means that the tool should minimize the area to the lowest possible value.

Figure 6 shows the block diagram of our DES56 design. This block diagram shows the input/output interface of the DES system which described in section 2.2.

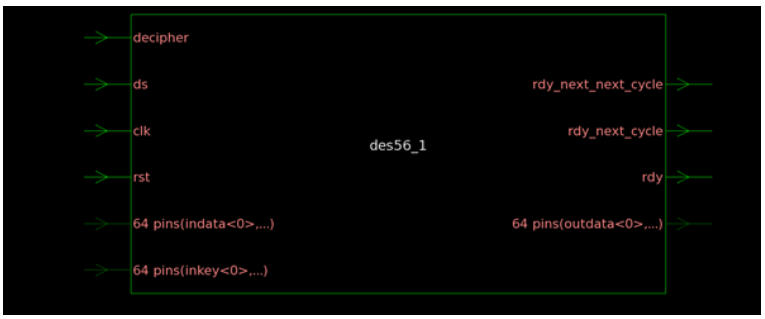


Figure 6: DES Block

The tool can generate the schematic view of the system which is shown in Figure 7. As mentioned in section 2.2, our system is composed of 3 different processes: Control, SetData, and SetKey. Figure 8 shows the schematic view of the system part corresponding to the control process.

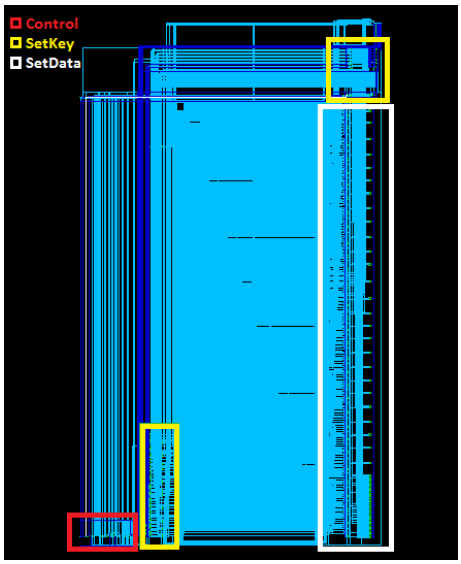


Figure 7: Schematic view of DES

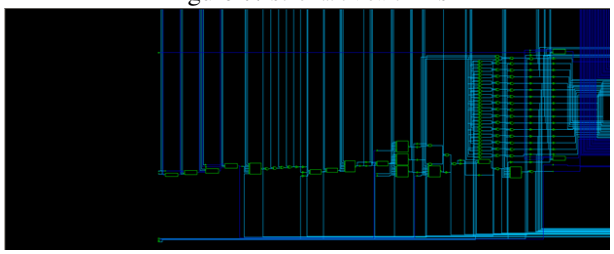


Figure 8: Schematic view of Control Process

Moreover, the tool generates some useful reports about the synthesis process. These reports summarize the design complexity, i.e. the number of ports, nets, combinational and sequential cells..., as well as the area of the design and the power. Some of these reports are shown in Table 2:

Number of Ports:	199
Number of nets:	1936
Number of cells:	1632
Number of combinational cells:	1440
Number of sequential cells:	192
Number of macros:	0
Number of buf/inv:	176
Number of references:	25
Combinational area:	2532.00
Non-combinational area:	1359.00
Total cell area:	3891.00
Total area:	undefined

Table 2: Report of Synopsys Design Compiler

During the synthesis process, the tool sometimes optimizes the design by removing one of the RTL code's registers (countup_reg [4]) since it is a constant register, and sometimes it doesn't. Hence, we got two different synthesized gate-level codes, one with the same number of the sequential cells (i.e.193) as the RTL design and the other with 192 sequential cells. We found that there is a variable in the RTL code (countup) which is defined as an integer with values form 0 to 16, i.e. it needs 5 flip-flops. However, this variable has not been assigned to 16 in the code, and as a result, countup_reg [4] is always has a value of

zero and can be removed in the optimization phase. However, this optimization will affect the equivalence checking results between the RTL and gate-level synthesized codes.

4 Equivalence Checking

Equivalence Checking is one of the formal verification techniques used to check the equivalency of two systems. It can check the equivalency of one representation of a system to another representation (such as: RTL vs. gate-level) to check the correctness of the transformation process (such as: synthesis). It also can check the equivalency between the specification and the implementation or between two implementations of the same system.
The equivalence checking process becomes complex when dealing with sequential designs and may require much more computation resources than those required for the combinational designs. However, if the two sequential designs have the same number of the sequential registers, flip-flops, and latches; i.e. the two designs one-to-one mapped (matching), the equivalence checking problem can be accomplished by verifying the combinational part of the designs only. This simple solution cannot be used with there is no such one-to-one mapping in the sequential part of the designs in which the verification process will be more complex.
In our project, we are going to check the equivalency of our RTL level VHDL file with the synthesized gate-level file (section 4-1). then, in Section 4-2 we introduce the techniques that we have used for finding the bugs of design and after that we go to the details of some bugs in Section 4-3

4.1 RTL code and Synthesized gate-level code

As mentioned in the previous section, we have got two different synthesized gate-level VHDL files form the synthesis process; i.e. the optimized one with 192 D flip-flops and the other with 193 D flip-flops. Verifying each of these two files with the RTL design by Formality and Conformal, leads to one unmatched (unmapped) point between the RTL and optimized gate-level; where other gate-level file is matched with RTL code. This unmatched point is named countup_reg[4]. i.e. countup_reg [4], while the other gate-level file is matched with the RTL one.
Since this unmatched point (countup_reg [4]) exists in the control part of the design, it will affect all other points, which leads to a failure in the equivalence checking for the whole 192 D flip-flops for the optimized gate-level file. Nevertheless, the verification of the other gate-level file with the RTL file succeeded with 193 matching points. These results do not mean that the synthesis process was incorrect in the first gate-level description, but they indicate that the equivalence checking failed due to the mismatch between the RTL level and the gate-level descriptions. Figure 9 summaries the results of the equivalence checking between the RTL level design with the synthesized gate-level files for both formality and Cadence.

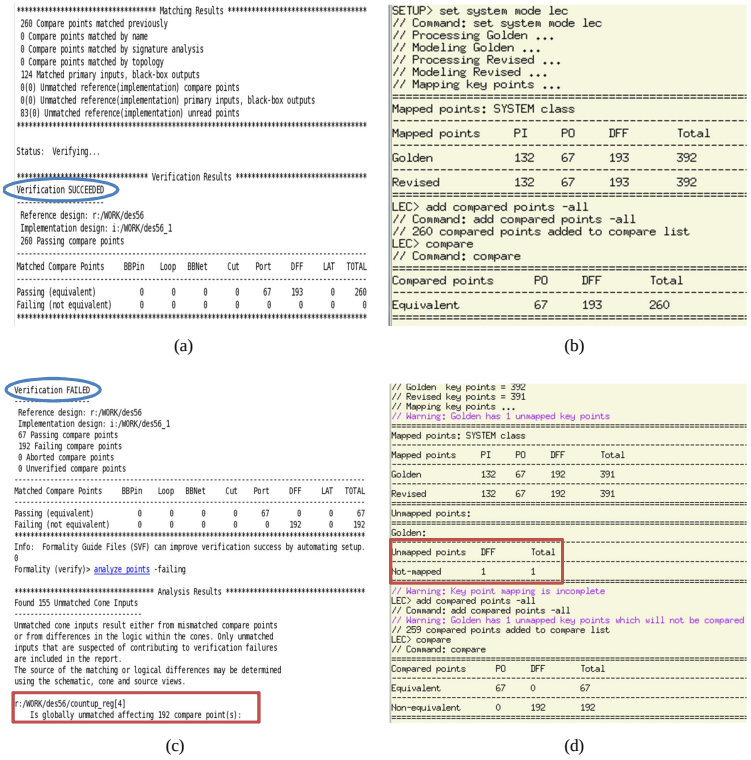


Figure 9: The results of the Equivalence checking between the RTL level design and the synthesized gate-level files. (a) and (b) show the result of verification of the RTL with the synthesized un-optimized gate-level file using Formality and Conformal respectively. (c) and (d) show the result of verification of the RTL with the synthesized optimized gate-level file using Formality and Conformal respectively.

However, we found that the given buggy gate-level description has 192 D flip-flops, which means countup_reg[4], has been removed(optimized), so we will use the optimized gate-level description in the debugging techniques and will call it the reference and will call the buggy gate-level file the implementation.

4.2 Bug Finding Techniques

Before talking about the design bugs, categorize the techniques that we have used to find these bugs to 4 categories.

1. Similarity: this technique is helpful for the cases that both reference and implementation circuits are almost similar. In these cases, tracing the failing points in implementation design to find the first gate that is different from reference design can easily resolve the problem.

2. Logic Analysis: in this technique, the Boolean expression of the failing node should be extracted from the both reference and implementation circuit and after that, changing the implementation expression in a way to make it equal with reference expression.

3. Counter-Example Values: Finding a gate candidate by applying the different input values that leads to a failing situation. After finding the candidate, according to the counter example values we can find a suitable gate to exchange it with the candidate gate.

4. Exhaustive Search: using different gate possibilities in a library for a specific gate is called exhaustive search. After finding a gate candidate, according to some features of that gate (like number of inputs), an exhaustive search can be performed inside a library.

Among these techniques, the first one which is called similarity is the simplest one. This technique can be helpful in simple cases where both implementation and reference implementation have similar gates except for the gate in implementation circuit that causes failure. Therefore, resolving the problem can be easily done by changing the unmatched gate. Logic analysis is another technique which is suitable for simple cases. To use this technique, we should be affordable to find the Boolean expression of failing point in both design and implementation circuits. This is possible when the failing point is near the inputs or registers. In case we are far from registers and inputs, different inputs may contribute the failing point and this makes impossible to do Boolean manipulation or finding a Boolean expression.
However, finding a situation applicable for technique 1 and 2 is rarely observable. The other two techniques can be used for more complicated failures. Counter-example values are the values which lead to a failure state for the implementation circuit. Using these values specifies the gate or the chain of gates that leads to the unmatched values. After getting the gate candidates, we can decide which gate has the higher probability to be the source of problem. Sometimes we may have a few gate candidates for a particular failing point which is impossible to detect incorrect one using the mentioned techniques. In these cases we can limit the search area by mentioned techniques and then do an exhaustive search for the remaining gates in the library which has the same number of inputs and outputs.
Although it could be possible to detect the incorrect gate by applying one of these techniques but it usually requires using a combination of these techniques. In Section 4.3 we explain how these techniques help us to find the bugs in DES design.

4.3 Debugging

The bugs that have been found in DES design are summarized in Table 3. As it is illustrated, 15 bugs have been found. Second column of this table shows the component label that is changed from the implemented component in 3rd column, to correct one in 4th one. The 5th and 6th columns of Table 3 shows the number of failing points and the failing node names that is achieved after applying this gate change. The last column specifies the technique(s) that are used to find the bug.

BUG number	Component	Implemented component	Correct Component	Number of failing points	ASSOCIATED FAILING POINTS	BUG Technique
1	U3798	AO2	ND4	48	countup_reg_3_inst	Similarity
2	U3457	EON1	AO2	47	key_l_reg_23_inst	Similarity
3	U3458	EON1	AO2	46	key_l_reg_24_inst	Similarity
4	U3703 U3705	AO2 ND4	ND4 AO2	45	key_r_reg_19_inst	Similarity
5	U3089	EON1	EO1	37	inmsg_reg_57_inst,outdata_reg_55_inst, outdata_reg_37_inst,inmsg_reg_51_inst, outdata_reg_57_inst,inmsg_reg_41_inst	Logic Analysis
6	U3521	EON1	AO2	36	inmsg_reg_7_inst	Logic Analysis
7	U3011	AN3	OR3	30	inmsg_reg_63_inst,inmsg_reg_38_inst, outdata_reg_9_inst,outdata_reg_7_inst, inmsg_reg_43_inst,outdata_reg_35_inst	Counter Example and Exhaustive Search
8	U3106	EON1	AO3	24	inmsg_reg_35_inst,outdata_reg_33_inst, inmsg_reg_60_inst,outdata_reg_31_inst, inmsg_reg_42_inst,outdata_reg_43_inst	Counter Example and Exhaustive Search
9	U3073	NR3	AO6	16	outdata_reg_3_inst,inmsg_reg_55_inst, inmsg_reg_47_inst,outdata_reg_17_inst, inmsg_reg_37_inst,outdata_reg_23_inst, inmsg_reg_61_inst	Counter Example and Exhaustive Search
10	U3086	EON1	AO3	12	outdata_reg_49_inst,outdata_reg_53_inst, inmsg_reg_33_inst,inmsg_reg_49_inst	Counter Example and Exhaustive Search
11	U3108	AO3	EON1	10	inmsg_reg_45_inst	Counter Example and Exhaustive Search
12	U3007	AN3	OR3	08	inmsg_reg_54_inst,outdata_reg_13_inst, outdata_reg_59_inst,inmsg_reg_40_inst	Counter Example and Exhaustive Search
13	U3004	AN3	OR3	04	outdata_reg_15_inst,inmsg_reg_40_inst, outdata_reg_59_inst,inmsg_reg_62_inst	Counter Example and Exhaustive Search
14	U2342	ND2I	EOI	02	outdata_reg_15_inst,inmsg_reg_40_inst	Counter Example and Exhaustive Search
15	U2343 U3082	ND2I EON1	NR2I AO4	00	outdata_reg_15_inst,inmsg_reg_40_inst	Counter Example and Exhaustive Search

Table 3: List of BUGs in DES implementation

Following we explain the flow of finding bugs number 1, 6 and 9. These bugs are selected to present all techniques mentioned in Section 4.2.

4.3.1 Bug number 1

As it can be seen in Figure 10, Formality has proposed U3798 as a candidate for the corresponding failing point. By back tracking the input signal of U3798 you can find that the inputs are as follows:

	Implementation	Reference
A	countup_reg0	countup_reg0
B	countup_reg1	countup_reg1
C	countup_reg2	countup_reg2

Table 4: Input signals for U3798

The output signal of U3798 drives an AO7 gate in both the implementation and reference circuits. So the only difference is the fourth input. As you can see in Figure 10 this signal is produced by logical combination of countup registers. We used similarity technique in this case. This means that the implemented circuit seems very similar to its reference circuit so we can replace the buggy gate with its original one in the reference circuit. In this case, since three of the input signal of U3798 are 100% equal to their original gates and the fourth one seems to be equal we replaced U3798 with its original gate in the reference circuit which is an ND4 gate.

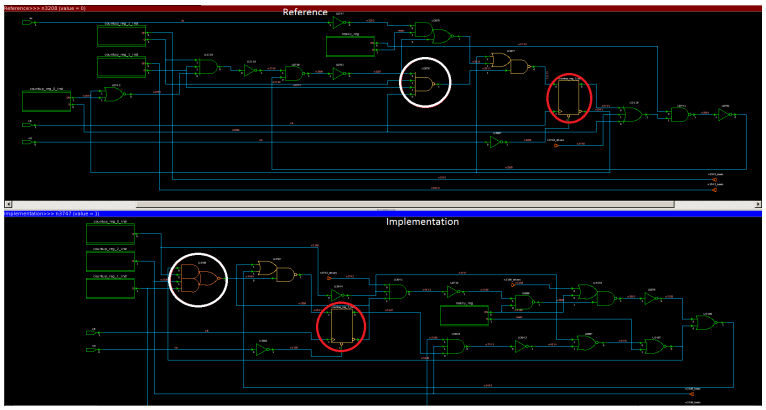


Figure 10: The schematic view of both reference and implementation circuits for bug number 3

4.3.2 Bug number 6

This bug is related to control part of DES design. Figure 11 illustrates both reference and implementation circuit of DES design in Formality tool. The red circle in inputs of "inmsg_reg_7_inst" shows the failing point for bug number 6. The yellow components in both designs specify the gate candidates are recommended by the tool. To find the incorrect gate among these two and also to replace it with a correct gate, we have used the second technique called logic analysis. To apply this technique we require finding a Boolean expression for all inputs of candidate gates in both designs. Following is the expressions for the gates that are specified by white circle.

	Reference Circuit (U3571)	Implementation Circuit (U3521)
A	Indata[1]	Indata[1]
B	$\overline{(\alpha+\overline{((\overline{\alpha}+dc).\overline{rst})})}$	$\overline{(\alpha+\overline{((\overline{\alpha}+dc).\overline{rst})})}$
C	Inmsg_reg_39_inst	Inmsg_reg_39_inst
D	$\alpha.((\overline{\alpha}+dc).\overline{rst})$	$\alpha.((\overline{\alpha}+dc).\overline{rst})$
$\alpha = \overline{(\overline{countup\_reg\_1\_inst}.countup\_reg\_2\_inst)} + countup\_reg\_3\_inst + countup\_reg\_0\_inst$		

Table 5: The Boolean expression for bug 6

As described in Table 5, the input A, B and C of both components are exactly the same and input D seems to be different. However, after applying NOT operator for D input of component U3521 we see the same Boolean expression achieved for the reference circuit (U3571). Because the inputs of both gates are similar and also outputs of them are connected to the same gate, we can use the type of reference component (U3571) instead of implementation one (U3521). Therefore changing ”EON1” to ”AO2” solves this problem.

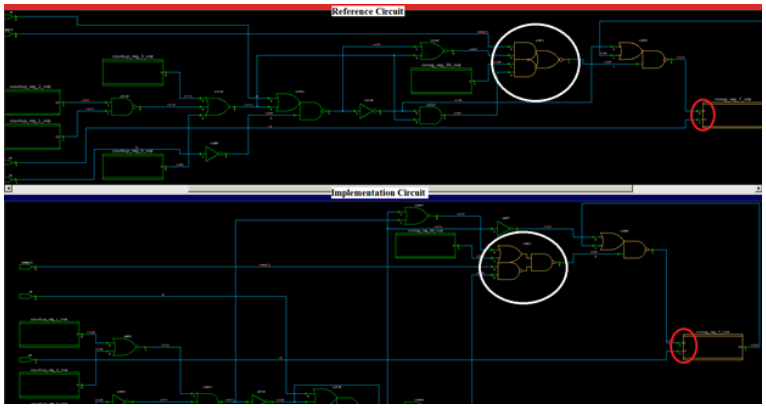


Figure 11: The schematic view of both reference and implementation circuits for bug number 6

4.3.3 Bug number 9

Figure 12 shows the schematic view of the reference and implementation circuits in Formality tool. The formality has suggested the gate U3073 (NR3) in the implementation as an error candidate that is responsible for the failing of 4 comparing points and Conformal gives it a probability of 1. Clearly from the figure, handling this gate with the similarity or logic analysis techniques is very hard and useless since the reference and implementation circuit are very big and different.

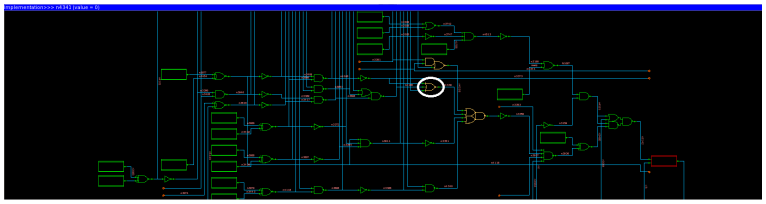


Figure 12: The schematic view of both reference and implementation circuits for bug number 9

However, we found that the only way to handle such bugs is to use the Counter-Example and Exhaustive Search techniques together. Using the pattern of the counter examples provided by Formality as shown in Figure 13, the new gate should have an output ’1’ when the inputs are ’100’ and ’010’ which will make the input at the failing point(outdata_reg_3) to be ’0’ as in the reference design. By using exhaustive search technique we solve this problem. to do this, we search for all three-input gates inside the library. we found that AO6 is the correct one.

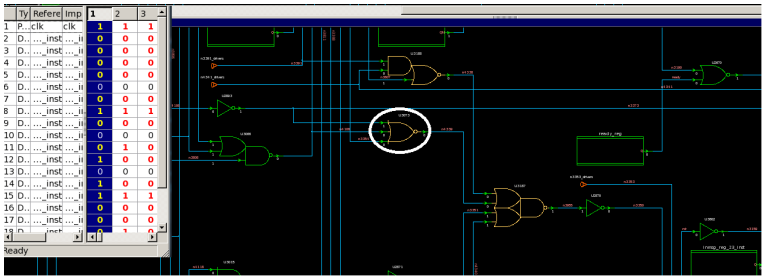


Figure 13: The schematic view of both reference and implementation circuits for bug number 9

5 Tool Comparison

In this section we want to compare two equivalency checking tools, Synopsys Formality and Cadence Conformal, used for finding DES bugs. First we indicate the general features of these two tools in Section 5.1 and 5.2 and at last in Section 5.3 we make a comparison between these two tools.

5.1 Synopsys Formality

Formality is a functional equivalence application from Synopsys. This tool is able to compare gate-level netlist to its register transfer level source. Also it is able to compare two gate-level netlist source against each other. After the comparison, it can report whether the two provided file are functionally equivalent or not.The Formality tool can significantly reduce design cycle by providing an alternative to simulation for regression testing [6].
A brief tutorial on how to use this tool can be found in [6].If two design are equivalent, formality will return 1 and displays *Verification Succeeded!*. Otherwise it will return 0 and displays *Verification failed*. Also if two designs are not equivalent, formality will provide failing points. By default maximum number of failing points is 20. If more than 20 failing points found in the design, formality will consider them as *Unverified points*. You can change maximum number of failing point by typing *"set verification_failing_point_limit 200"* in formality command prompt.

Some of the useful features of Formality are as follows:

1. Formality can provide failing point that cause failure in verification.

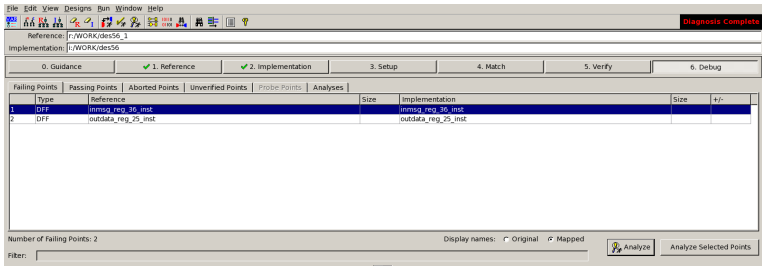


Figure 14: Failing points

2. Formality can bring error candidates for a specific failing point or a group of failing points. This can be done by selecting one or more than one failing points in the debug window *6.Debug* and then use *diagnose selected points* to generate candidates for the specific failed points.

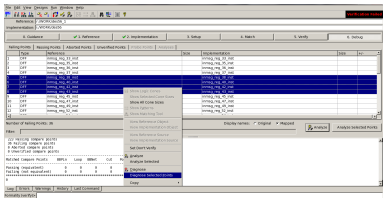


Figure 15: Diagnose failing points

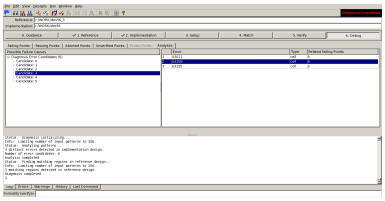


Figure 16: Candidates for the specific failing point

3. Pattern window is one of the other features of formality. After verifying the design, formality comes up with all contradiction patterns that found in the design.

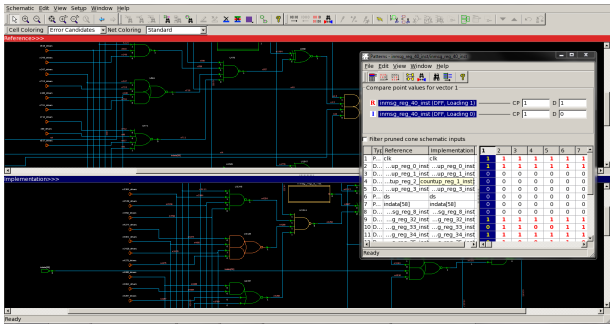


Figure 17: Failing points

4. Formality is benefiting from Logic Cones which is a good way of illustrating the Reference and Implementation schematic view. This feature, gives verifier engineer a useful ability to search for a BUG. Formality colors the error candidate component based on their weight percentage as shown in Table 6 .This feature speeds up the verification process and makes it easy to track the BUG path in implementation circuit.

cyan:	not an error candidate
gray:	0 to <25 percent
dark green:	25 to <50 percent
blue:	50 to <75 percent
purple:	75 to <90 percent
red:	90 to 100 percent

Table 6: Error candidates color

5.2 Cadence Conformal

Encounter Conformal (EC) is an equivalence checking tool from Cadence. This tool is able to compare and verify the equivalency of a gate-level/RTL description against a gate-level description by providing a formal proof. This tool is able to find counter example in Revised file(Implementation file) and the corresponding gate in the Golden file(Reference file). A brief tutorial on how to use this tool can be found in [6].

1. Conformal is able to find the failing points. This feature is available through *Mapping Manager* form the task bar. As you can see in Figure 18 the failing points are shown as *red spots* while the passed points are shown as *green spots*. For each failing point Conformal comes up with some debugging option. Some of the useful option will be discussed later in this section.

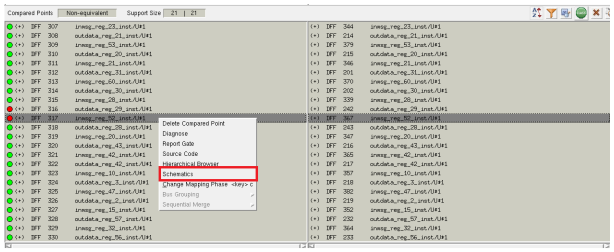


Figure 18: Failing points(Conformal)

2. Similar to Formality, Conformal can also bring the error candidates for the corresponding failing point. As you can see in Figure 19 beside a list of proposed candidates, for each candidate Conformal generates a number between 0 to 1 which indicates the BUGGY intensity of the corresponding candidate. In other word this number indicates the possibility of being the BUG in the corresponding failing point . This feature is useful specially when you need to change two gates at the same time.

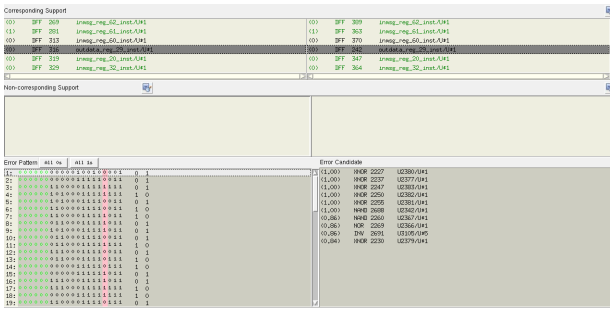


Figure 19: Error Candidates(Conformal)

3. For each failing/passing point, Conformal can provide a schematic view. Although Conformal generates a schematic view, but in comparison with formality, tracking a BUG in Conformal schematic view is much harder than in Formality.

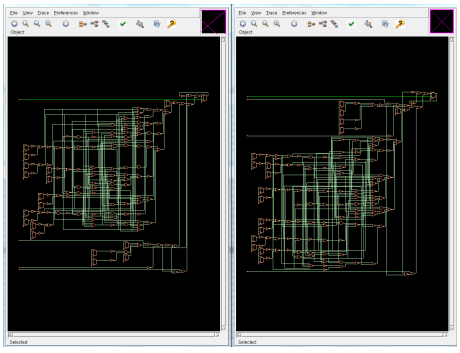


Figure 20: Schematic View(Conformal)

4. Similar to Formality, for each failing point, Conformal generates a list of error pattern (Contradictions) which means that the specific failing point has these contradictions. For each failing/pass-

ing point, Conformal can provide a schematic view. Although Conformal generates a very useful schematic view, but in comparison with formality, tracking a BUG in Conformal schematic view is much harder than in Formality.

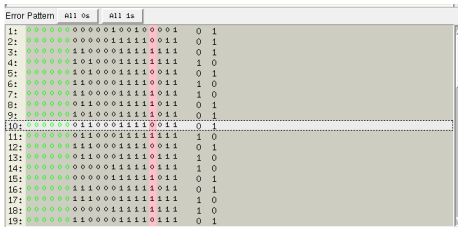


Figure 21: Schematic View(Conformal)

5. Another feature of Conformal is *Report Gate* option. This feature allows us to backtrack a specific gate by showing its Fanouts and Fanins. This feature is useful when you are dealing with a huge circuit and schematic view is not useful anymore.

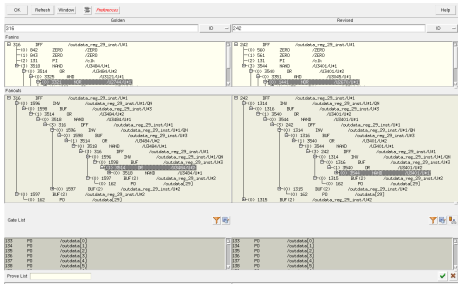


Figure 22: Report Gate view

5.3 Equivalency Checking Tools

In this section we compare Formality and Conformal to each other according to our debugging experience for DES design. We have categorized different features to three different categories:

1. User Environment: this category is mainly about visual effects that are used by the tool to show the design.

2. Debugging: the features that facilitate finding bugs are put in this category.

3. Other features: these features are some general features that are mainly about portability.

To make conclusion easier, we assign each feature a score which shows the importance of that. The score distribution between features is specified in Table 7. From User Interface (UI) point of view,despite of Formality has slightly better GUI, both tools have same capability and thus both tools have got 5 scores. This is because scripting interface is more important than GUI in these tools. The other features of User Environment part is simulation values which gives user the possibility of watching the value of signals. Both tools have this feature but Formality illustrate the simulation values by a coloring scheme which makes it easier to detect. The most important feature of this category is schematic view which makes it possible to see a part of design graphically. This feature is available in both tools but Formality mixes it with other features like simulation values, coloring to specify the bugs and simulation values.Moreover, as mentioned in Section 5-2, Conformal provides a text representation for showing a circuit which is helpful in big circuits. To sum up, a simple schematic without any additional information is not completely helpful, thus we grade

Conformal and Formality tools in order 10 and 15 for this feature.

The next category, Debugging, is more important than previous one. The features of this category can directly accelerate the process of finding bugs. Patterns are very useful tool for debugging. Using pattern lets user knows about the values that lead to a failing situation and finding the source of failure. Both tools provide patterns or counter example values but Formality provide these values in schematic view and for all gates while Conformal only presents for inputs and outputs of a specific part of design (failing point). Bug suggestion is another important feature in this category. This also knows as diagnosis. Both tools support this feature but Formality can do suggestion based on multiple failing points. This facility can help to find the source of bug better, where the failing points are dependent to each other. The other important factor in diagnosis tool is the accuracy of suggestions. As we do not know about the techniques that tools are using to do suggestion, we try to evaluate this accuracy according to our design (DES). The experiment that we did to evaluate the accuracy of suggestions is to do diagnosis in the single failing points. The order of these single points is according to Table 7. According to our findings, the percentage of actual errors that is suggested by Formality is 94% and by Conformal is 88%. In other words, Formality and Conformal could not provide a correct suggestion for in order 6 and 12 percent of failing points. About the number of suggestion, 16% and 20% of suggested candidates by in order Formality and Conformal are true. To calculate these two values, we assume 100% for each candidate of Formality and then compare summation of them with the summation of candidates' probability in Conformal. These values give us an estimation of accuracy in a particular case and it cannot be considered as a general rule. The complete list of both tool suggestions are available in Appendix B.

Failing points are a basic requirement for start debugging. This feature let user knows about the area of error in design. By knowing the area of error, user can use his/her functional knowledge about the design and apply a better methodology. Both tools support this feature. The last attribute of this category is tracing bugs. Thanks to good schematic view and coloring scheme that Formality has, tracing bug is much easier than Conformal which only provides it by exploring the gate in text editor.
Other features category that is about some general and essential characters is composed of two main features: library independency and accepted languages. These two features are fully supported by both tools. These two let designer do comparison between the specification and implementation written in different languages or synthesized by different libraries.

Category	Feature	Conformal		Formality	
User Environment	User Interface(5)	Script + GUI	4	Script + GUI	5
	Simulation Values(10)	Observable in schematic view	5	Observable in schematic view + coloring	10
	Schematic View(15)	Gate schematic+values	7	Gate schematic + values + coloring + easy exploration	10
Debugging	Patterns(20)	Values for inputs and outputs	10	Values for internal wires	20
	Bug Suggestion (diagnosis)(15)	Available for single node + probability of being error + 84% accuracy	10	Available for multiple nodes + 94% accuracy	12
	Failing Points(10)	Available	10	Available	10
	Tracing Bugs(5)	Limited to a text editor	2	In a graphical view	5
Other Features	Library independency(10)	Independent	10	Independent	10
	Accepted languages(10)	VHDL, Verilog, Systemverilog	10	VHDL, Verilog, Systemverilog	10
	RTL comparison(-)	Available	-	Available	-
Final	Total: 100		68		92

Table 7: Tool Comparison

5.3.1 Case study

To show how the mentioned features in previous section can be utilized for finding and resolving a bug, a case study explains in this section. This case study is going to explain how we found to solve the bug number 15 in the Table 3. As mentioned in previous section, Formal tool has better features for illustrating the bug in comparison with Conformal. Due to this, we start our effort using Formal tool. For this error we have two failing points named "inmsg_reg_36_inst" and "outdata_reg_25_inst". By doing diagnosis which is one of the features of Formal tool, we can see 5 component candidates. These candidates are shown in Figure 23 by yellow color.

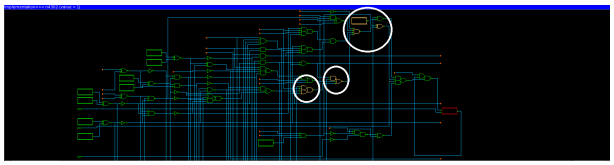
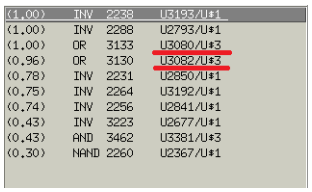


Figure 23: Error candidates for ”inmsg_reg_36_inst” using Formality tool.

One of those is an input register which is obviously incorrect. The next two are U3427 (NAND) and U2333 (XOR). According to DES spec and also other similar situations in the circuit these two components does not seems to be true candidates. For the two last candidates, because many other inputs contribute these two components, finding the right component candidate is hard. Due to this we use Conformal tool simultaneous with Formal tool. To do this, after doing comparison between two circuits we select one of the remaining failing nodes. As mentioned in previous section this is one of the limitations of Conformal tool that you cannot do diagnosis for more than one node. We do diagnosis for ”inmsg_reg_36_inst” node. The component candidates of this operation is shown in Figure 24.



(1.00)	INV	2238	U3193/U#1
(1.00)	INV	2288	U2793/U#1
(1.00)	OR	3133	U3080/U#3
(0.96)	OR	3130	U3082/U#3
(0.78)	INV	2231	U2850/U#1
(0.75)	INV	2264	U3192/U#1
(0.74)	INV	2256	U2841/U#1
(0.43)	INV	3223	U2677/U#1
(0.43)	AND	3462	U3381/U#3
(0.30)	NAND	2260	U2367/U#1

Figure 24: Error candidates for ”inmsg_reg_36_inst” using Conformal tool.

The first candidate is not correct because it is an inverter. Invertor component cannot be the right one because all counter example values can be resolved by adding or removing an inverter.
However, removing these invertors lead to new types of errors. The component U3080 is the next candidate which is common in Formal tool. To see whether this a right choice for changing, we use pattern facility in Formality tool. This tool is shown in Figure 25. The output of component U3080 for inputs ”1100”, ”1110”, ”0100”, ”1011” and ”1101” should be in order ’0’, ’1’, ’0’, ’1’ and ’1’ where it is not. For finding the correct gate we use the fourth technique, exhaustive search, and apply these values in all possible cases and find that all of them produce at least two wrong outputs. Therefore, we continue our debugging by choosing the next candidate in Conformal tool which is U3082. According to patterns we got from Formality, inputs ”1111”, ”0100”, ”1011”, ”0111” and ”1110” should result ’0’, ’1’, ’0’, ’0’ and ’1’ where it did not. After applying these inputs to possible 4-input gates, realize that AO4 has the closest behavior and produces only one unmatched output.

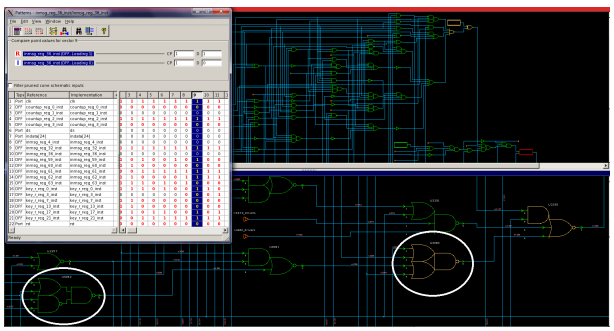


Figure 25: Using pattern facility of Formality tool to find the correct substitute for the specified gates

After changing this gate we got 2 failing points again but with some new candidates. Some of resulting 6 candidates after this change are similar with previous step and are in the same branch with U3082. U3427 specified by a green circle in Figure 26, as mentioned is true and does not need to be changed. The components U2335 , U3291 , U3080, U3018 which are specified by a blue circle are in the same branch with U3082. We assume that previous change is the correct one. This assumption is realistic because we do not need any signal to be '1' in U3427.

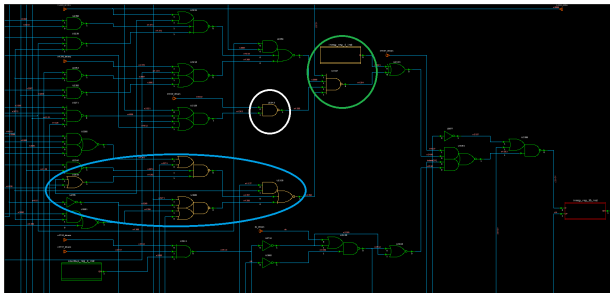


Figure 26: Categorizing the candidates to find the right one.

However, one inputs of this gate should be zero in the remaining state to make design correct. The only candidate from other inputs of U3427 which can produce zero is U2343. This gate is specified by a white circle. To find a correct substitution for U2343 we again use pattern facility in Formality tool. The current gate which is a NAND gate does not produce correct output for "10" and "01", thus we need a gate which produce '0' for "10" and "01" and for other inputs works similar to NAND. The truth table of such gate is shown in Table 10.

Inputs	NAND output	Correct gate output
00	1	1
01	1	0(Opposite with NAND)
10	1	0(Opposite with NAND)
11	0	0

Table 8: Generating a truth table for correct gate by using patterns.

The resulting truth table is truth table of a NOR. By replacing the NAND gate (U2343) with a NOR gate we can see that the both failing points are removed.
In addition to the feature comparison, we compare these two tools according to their memory usage and CPU time consumption. The results of this comparison for both Gate-to-Gate and RTL-to-Gate are summarized in Table 9. The memory usage values for both tools are almost near each other. However, a huge difference is observable between the CPU time values of tools. For example CPU time of Conformal for Gate-to-Gate comparison is 12 times better than Formality. As we do not know about the method that these tools are used or the way that these values are calculated, we cannot 100% certify that Conformal is better. In example, Conformal may remove some fixed cost like "set up" time from this value.

Compared files	Phase	Comparison Aspect	Comparison Aspect	Comparison Aspect
RTL level vs Synthesized gate-level	Matching	CPU Time(sec)	3.65	0.25
		Memory Usage (MB)	98.4	100.39(Peak Usage)
	Verification	CPU Time(sec)	10.04	0.24
		Memory Usage (MB)	107.2	100.39
Gate-level vs Gate-level	Matching	CPU Time(sec)	3.31	0.26
		Memory Usage (MB)	94.9	98.61(Peak Usage)
	Verification	CPU Time(sec)	5.76	0.06
		Memory Usage (MB)	94.9	98.61

Table 9: Memory usage and processing time

6 Summary and Conclusion :

The growth of design complexity and also high cost of defective fabrication demand new effective verification methods. Formal methods can be used to reduce the state space explosion problem in nowadays large design. Equivalency checking is one of the formal methods that can be used. Due to different tools that are available for this technique, it is widely used in industry. In this report we used two equivalency checking tools named Formality and Conformal to debug the DES design. First we synthesized the RTL implementation of DES to get the reference design and then did equivalency checking between the given implementation with obtained reference design. During the debugging process we removed 49 failing points by 4 different techniques. All of used techniques needs to be interactive and cannot be done automatically. Having knowledge about the design implementation is helpful during the debugging. However, DES is a cryptography application and complexity inside it makes tracing bugs hard. Especially, this can be observed in the cases that we have two or more than two bugs in a failing point like case study explained in Section 5-3. Using these 4 techniques need to have sufficient facilities which should be provided by the tools. To show to what extent these tools support debugging techniques, we made a comparison between these two tools. Our comparison is made based on our experience during the debugging of DES. This comparison shows that Formality is slightly better that Conformal. This is mainly because Formality provides better schematic and let user debug the implementation easier that Conformal. However, this conclusion does not seem to be a general conclusion, because deciding about a tool depends on different factors like design, user preference, etc. And all of these factors may change from a design to another one.

A Appendix A

(a) Initial Permutation (IP)

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

(b) Inverse Initial Permutation (IP^1)

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

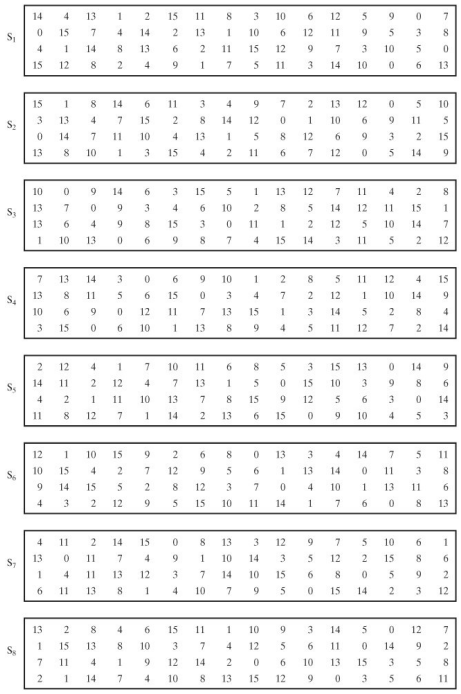
(c) Expansion Permutation (E)

32	1	2	3	4	5	
4	5	6	7	8	9	
8	9	10	11	12	13	
12	13	14	15	16	17	
16	17	18	19	20	21	
20	21	22	23	24	25	
24	25	26	27	28	29	
28	29	30	31	32	1	

(d) Permutation Function (P)

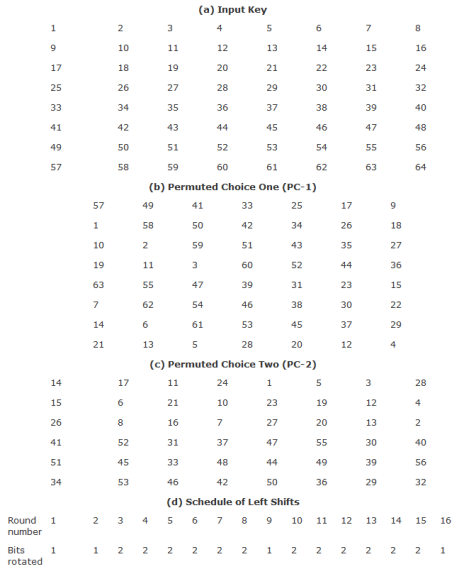
16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

Figure 27: Permutation Tables for DES



S_1	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13
S_2	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9
S_3	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12
S_4	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14
S_5	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3
S_6	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13
S_7	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12
S_8	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Figure 28: Definition of DES S-Boxes



(a) Input Key

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

(b) Permuted Choice One (PC-1)

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

(c) Permuted Choice Two (PC-2)

14	17	11	24	1	5	3	28
15	6	21	10	23	19	12	4
26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

(d) Schedule of Left Shifts

Round number	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bits rotated	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Figure 29: DES Key Schedule Calculation

B Appendix B

The following table shows the Formality and Conformal gate Candidates in process of finding BUGs. Gate shown in ■ are the wrong suggestion from the tool and gate shown in ■ are correct suggestions from the tool.

Register	Formality	Conformal
countup_reg_3	U3798,U3797	U3798(1.0),U1497(0.38)
inmsg_reg_32	U3089,U3035,U3569,U2546, U3501,U3500	U3089(1.0),U2905(1.0),U2734(0.87),U3092(0.87), U3091(0.87),U2904(.8),U2927(0.69),U2917(0.69), U2589(0.69),U2902(0.69)
inmsg_reg_33	U3086,U2721,U3425,U2492, U3379,U3378	U3086(1.0),U3246(1.0),U3190(1.0),U2535(0.94), U2522(0.94),U3032(.86),U856(0.84),U2530(0.82), U2529(0.79),U2898(0.79)
inmsg_reg_35	U3106,U3557,U3556,U2387, U3493,U3492	U3106(1.0),U2800(1.0),U3107(.99),U3217(0.96), U3021(0.95),U2400(0.80),U2430(0.79),U2846(0.79), U2401(0.79),U2848(0.72)
inmsg_reg_7	U3521,U3520	U3521(1.0),U2677(0.04)
key_l_reg_24	U3458,U3637	U3458(1.0),U1698(0.14), U3639(0.14),U3638(0.14)
key_l_reg_23	U3457,U3634	U3457(1.0),U1703(0.14), U3636(0.14),U3635(0.14)
key_r_reg_19	U3703	U3705(1.0),U1583(1.0),U3480(0.38), U3704(0.38)
out_data_reg_3	U3073,U3188,U3187,U2288, U3150,U3164,U2039,U3412	U3073(1.0),U3188(1.0),U2948(0.64),U3194(0.64), U3066(0.64),U2893(0.36),U3412(0.22),U2910(0.06), U2308(0.06),U2909(0.06)
out_data_reg_25	U3291,U2335,U3427,U2333, U3402	U2335(1.0),U3193(0.87),U2793(0.87),U3080(0.87), U2841(0.8),U3082(0.8),U3192(0.74),U2850(0.67), U3402(0.06),U3291(0.41)
out_data_reg_29	U2342,U1975,U3430,U1973, U3401	U1975(1.0),U2367(0.86),U2365(0.86),U3105(0.86), U2850(0.86),U3841(0.84),U3192(0.84),U3203(0.84), U2371(0.8)
in_data_reg_45	U3108,U2074,U3305,U2072, U3342,U3341	U3108(1.0),U2768(1.0),U2475(1.0),U2767(1.0), U3108(1.0),U2081(0.94),U31415(0.54),U2839(0.52), U2824(0.49),U2677(0.47),U3342(0.47)
in_data_reg_36	U2343,U3018,U3291,U3080, U2335,U3427,U3082	U3082(1.0),U3193(1.0),U2793(1.0),U3080(1.0), U2850(0.78),U3192(0.75),U2841(0.74),U2677(0.43), U3381(0.43),U2367(0.3)
in_data_reg_38	U3011,U3232,U3350,U3349, U2230,U3326,U3325	U3011(1.0),U3069(0.8),U2834(0.4),U2677(0.31), U3326(0.31),U2816(0.19),U2268(0.19),U2266(0.19),
in_data_reg_54	U3007,U3120,U3434,U3433, U1944,U3387,U3386	U3007(1.0)U3160(1.0),U2886(0.76),U3191(1.0), U2915(0.21),U2677(0.01),U3387(0.01)
in_data_reg_59	U3004,U3154,U3125,U2157, U3346,U3345	U3007(1.0),U3004(1.0)U2945(1.0),U2886(1.0), U3191(1.0),U3067(1.0),U2189(1.0),U3007(1.0), U3346(1.0),U3345(1.0)
key_r_reg_19	U3705,U3703,U3480,U3704, U1583	U3703(1.0)U1583(0.67),U3705(0.67),U3480(0.33), U3704(0.33)

Table 10: Tool candidates for the corresponding failing point.

References

C Reference

[1] William Stallings *Block Ciphers and the Data Encryption Standard.Cryptography and Network Security Principles and Practices. Fourth Edition, USA, ch. 3, s. 2, Prentice Hall.*

[2] Feistel, H. *Cryptography and Computer Privacy." Scientific American, May 1973.*

[3] *Dr. S. Tahar. COEN 7501-course material. 2009-2012*

[4] *CADENCE Encounter®Conformal® Equivalence Checking Reference* Product Version 10.1 May 2011

[5] *Formality® User Guide* Version Z-2007.06, June 2007

[6] *Dr. N. Abbasi, H. Aridhi. Digital Logic Synthesis and Equivalence Checking Tools Tutorial, 04/2012*

[7] *F. Vahid. Digital Design with RTL Design, Verilog and VHDL (2nd ed.). John Wiley and Sons, 467,ISBN 978-0-470-53108-2 2010*

[8] *http://dhost.info/pasjagor/des/start.php?id=0*