Acknowledgements

First and foremost, I would like to give my heartfelt thanksgiving to God for His grace and mercy in completing the first two phase of this project. He has given me the strength to persevere in finishing this report. I would also like to thank both my parents, Mering Jau and Unyang Deng for their continuous prayers and support. They both always encourage me and made me believe in myself even though at times I am low in confidence. I am eternally grateful for their involvement.

I would also like to thank my supervising lecturer, Mr. Noorzailly for his advices and input during the duration of this project. His suggestion on placing a state machine on my design has helped me tremendously. My two laboratory mates in Mohd Norazmi (Bon) and Zulfikri, for knowledge-sharing while we are busy laboring in our respective projects. Also, to Mr. Yamani , though was not my supervising lecturer, was kind enough to help me at times.

Abstract

This project is about the development of DES in hardware. DES, and it's variants (tripleDES) are the main encryption methods used in industry today. The DES designed should be able to process a 64-bit data block and it's 64-bit key and produces a 64-bit encrypted output. It also acts as decryptor, which is done by entering the 64-bit encrypted data together with the sub-key (operate in decrypt mode, where key is entered in reverse order). In order for our DES to work, modules are designed. These modules are controller, RAM, the DES core (initdata) and sub-key generator. All these submodules are developed, and then integrated as a complete DES cryptosystem. This DES system will be developed using VHSIC Hardware Description Language (VHDL). This is a complete report, from the designing phase up to the system testing at the end.

Contents

Diagrams List

Chapter One

Introduction

1.0 Introduction.

Cryptography is the most common method used in implementing security in data communication networks. Over a period of time, several cryptographic algorithms have been developed, such as DES, RSA, Rjindeal and etcetera. DES, along with its variant (3DES, AES) is among the most widely used cryptographic methods in data and information protection today. The DES algorithm has been written and implemented in programming languages such as C, C++, Java, Basic and others. But these are software versions. After comparing the performance of its hardware implementation, most of it which are implemented on Application Specific Integrated Circuits (ASIC), they outperform the software implementation. With the advancements that have been made in reconfigurable devices, Field Programmable Gated Arrays (FPGA) and Complex Programmable Logic Devices (CPLD), brings about the possibility of reconfigurable cryptographic devices into the real world. Cryptographic algorithms implemented on FPGA and CPLDs, provide a high level of flexibility, though it is in the expense of performance. This paper is about the development of hardware based security processor module, the DES using Very high speed integrated Hardware Description Language, (VHDL). The code will then be implemented on a reconfigurable device, specifically FPGA.

1.1 Problems to be addressed

1.1.1 Identifying functions, modules within algorithm.

The DES algorithm is a sequence of functions that is used to encrypt information represented in bits. We are designing a standard 64 bit DES chip. In order to develop this chip, we will have to identify the

module/components of this chip. A module comprised of a combination of functions in relation to the algorithm. Inputs (in bits) will be passed around within these modules, in which are processed by the functions in it.

By reviewing the literature related to DES, the functions that are used in the algorithm can be identified. We can develop the relevant source code based on the functions.

1.1.2 Learning VHDL.

VHDL is the most common language in developing digital systems. So, it is absolutely essential to learn this programming language in order to build a DES chip. This also includes getting familiar with a suitable development tool.

1.1.3 Develop a project schedule.

Develop a workable project schedule to design and complete a DES chip.

1.2 Scope of Research

The scope of this research just involves cryptography, and the DES algorithm in particular. Also about VHDL, and its related development tools.

1.3 Objective

The objective is to develop a hardware implementation of DES encryption algorithm based on VHDL, in other words, a DES chip.

1.4 Project Limitation

This project is limited to developing a simulatable model of DES algorithm. The DES algorithm runs on several modes. The design is based

on Electronic Code Book (ECB) mode of operation, which is the direct application of the DES algorithm to encrypt and decrypt data. There are three other modes of operations, Cipher Block Chaining (CBC) mode, the Cipher Feedback (CFB) mode, and the Output Feedback (OFB) mode. The characteristics of these modes are explained later in literature review.

1.5 Project Schedule

Below is the designated project schedule.

No.	Task	March	April	May	Jun	July	August	September
1	Early Research							
2	Literature Review							
3	System Analysis							
4	System Design							
5	System Development							
6	Testing							

System Design will be continous throughout the duration of project. The initial duration (March – April) is the design based on system analysis. The following duration is in response to system development requirements.

Chapter Two
Literature Review

2.0 Introduction to Literature Review

This chapter serves as an introduction to everything that is related/needed in developing a DES chip. This review is a critical evaluation of the literature, which provides an academic background to the area of study. Literature review is important to review on certain information resources. These resources were gained through reading of books, magazines, journals and also from the Internet. Through careful analysis on the information, the pros and cons of this project could be extracted from. All these will contribute to the development of this project, which is the hardware implementation of DES algorithm using VHDL.

The first part of this literature review is on cryptography, and an in-depth analysis of DES algorithm. Then, a review on the development language, VHDL.

2.1.0 Introduction to Cryptography

What is cryptology? Cryptography? Plaintext? Ciphertext? Encryption? Key?

The story begins: When Julius Caesar sent messages to his trusted acquaintances, he didn't trust the messengers. So he replaced every A by a D, every B by a E, and so on through the alphabet. Only someone who knew the ``shift by 3" rule could decipher his messages.

A cryptosystem or cipher system is a method of disguising messages so that only certain people can see through the disguise. Cryptography is the art of creating and using cryptosystems. Cryptanalysis is the art of breaking cryptosystems---seeing through the disguise even when

you're not supposed to be able to. Cryptology is the study of both cryptography and cryptanalysis.

The original message is called a plaintext. The disguised message is called a ciphertext. Encryption means any procedure to convert plaintext into ciphertext. Decryption means any procedure to convert ciphertext into plaintext.

A cryptosystem is usually a whole collection of algorithms. The algorithms are labeled; the labels are called keys. For instance, Caesar probably used ``shift by n" encryption for several different values of n. It's natural to say that n is the key here.

The people who are supposed to be able to see through the disguise are called recipients. Other people are enemies, opponents, interlopers, eavesdroppers, or third parties.

2.2.0 The DES algorithm

Introduction

The DES algorithm is based on a 128-bit block algorithm developed in the 1960s by IBM. In technical terms, LUCIFER is an iterative block cipher, using Feistel rounds - a block of data is encrypted a number of several times, each time applying the key to half of the block and then XOR'ing with the other half of the block.

DES was designed to use a 64-bit key to encrypt and decrypt 64-bit blocks of data using a cycle of permutations, swaps, and substitutions. Encryption and decryption use the same key.

A block to be encrypted is subjected to an initial permutation, then to a key-dependent computation, and then to a final permutation. The initial and final permutations take the 64-bit block and change the position of each bit in a pre-determined manner. The final permutation is the reverse of the initial permutation.

A DES key consists of 64 binary digits of which 56 bits are randomly generated and used directly by the algorithm. The other 8 bits, which are not used by the algorithm, are used for error detection. The 8 error detecting bits are set to make the parity of each 8-bit byte of the key odd, i.e., there is an odd number of "1"s in each 8-bit byte.

2.2.1 History and Issues on DES

In 1972, the National Institute of Standards and Technology (called the National Bureau of Standards at the time) decided that a strong cryptographic algorithm was needed to protect non-classified information. The algorithm was required to be cheap, widely available, and very secure. NIST envisioned something that would be available to the general public and could be used in a wide variety of applications. So they asked for public proposals for such an algorithm. In 1974 IBM submitted the Lucifer algorithm, which appeared to meet most of NIST's design requirements.

NIST enlisted the help of the National Security Agency to evaluate the security of Lucifer. At the time many people distrusted the NSA due to their extremely secretive activities, so there was initially a certain degree of

skepticism regarding the analysis of Lucifer. One of the greatest worries was that the key length, originally 128 bits, was reduced to just 56 bits, weakening it significantly. The NSA was also accused of changing the algorithm to plant a "back door" in it that would allow agents to decrypt any information without having to know the encryption key. But these fears proved unjustified and no such back door has ever been found.

The modified Lucifer algorithm was adopted by NIST as a federal standard on November 23, 1976. Its name was changed to the Data Encryption Standard (DES). The algorithm specification was published in January 1977, and with the official backing of the government it became a very widely employed algorithm in a short amount of time.

Unfortunately, over time various shortcut attacks were found that could significantly reduce the amount of time needed to find a DES key by brute force. And as computers became progressively faster and more powerful, it was recognized that a 56-bit key was simply not large enough for high security applications. As a result of these serious flaws, NIST abandoned their official endorsement of DES in 1997 and began work on a replacement, to be called the Advanced Encryption Standard (AES). Despite the growing concerns about its vulnerability, DES is still widely used by financial services and other industries worldwide to protect sensitive on-line applications.

To highlight the need for stronger security than a 56-bit key can offer, RSA Data Security has been sponsoring a series of DES cracking contests since early 1997. In 1998 the Electronic Frontier Foundation won the RSA DES Challenge II-2 contest by breaking DES in less than 3 days. EFF used a specially developed computer called the DES Cracker, which was developed for under $250,000. The encryption chip that powered the DES Cracker was capable of processing 88 billion keys per second. More recently, in early 1999, Distributed. Net used the DES Cracker and a worldwide network of nearly 100,000 PCs to win the RSA DES Challenge III in a record breaking 22 hours and 15 minutes. The DES Cracker and PCs combined were testing 245 billion keys per second when the correct key was found. In addition, it has been shown that for a cost of one million dollars a dedicated hardware device can be built that can search all possible DES keys in about 3.5 hours. This just serves to illustrate that any organization with moderate resources can break through DES with very little effort these days.

2.2.2 Steps in DES

1 Process the key.

1.1 Get a 64-bit key from the user. (Every 8th bit (the least significant bit of each byte) is considered a parity bit. For a key to have correct parity, each byte should contain an odd number of "1" bits.) This key can be entered directly, or it can be the result of hashing something else. There is no standard hashing algorithm for this purpose.

1.2 Calculate the key schedule.

1.2.1 Perform the following permutation on the 64-bit key. (The parity bits are discarded, reducing the key to 56 bits. Bit 1 (the most significant bit) of

the permuted block is bit 57 of the original key, bit 2 is bit 49, and so on with bit 56 being bit 4 of the original key.)

Permuted Choice 1 (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

1.2.2 Split the permuted key into two halves. The first 28 bits are called C[0] and the last 28 bits are called D[0].

1.2.3 Calculate the 16 sub keys. Start with i = 1.

1.2.3.1 Perform one or two circular left shifts on both C[i-1] and D[i-1] to get C[i] and D[i], respectively. The number of shifts per iteration are given in the table below.

Iteration # 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Left Shifts 1 1 2 2 2 2 2 2 1 2 2 2 2 2 2 1

1.2.3.2 Permute the concatenation C[i]D[i] as indicated below. This will yield K[i], which is 48 bits long.

Permuted Choice 2 (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

1.2.3.3 Loop back to 1.2.3.1 until K[16] has been calculated.

2 Process a 64-bit data block.

2.1 Get a 64-bit data block. If the block is shorter than 64 bits, it should be padded as appropriate for the application.

2.2 Perform the following permutation on the data block.

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

2.3 Split the block into two halves. The first 32 bits are called L[0], and the last 32 bits are called R[0].

2.4 Apply the 16 sub keys to the data block. Start with i = 1.

2.4.1 Expand the 32-bit R[i-1] into 48 bits according to the bit-selection function below.

Expansion (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

2.4.2 Exclusive-or E(R[i-1]) with K[i].

2.4.3 Break E(R[i-1]) xor K[i] into eight 6-bit blocks. Bits 1-6 are B[1], bits 7-12 are B[2], and so on with bits 43-48 being B[8].

2.4.4 Substitute the values found in the S-boxes for all B[j]. Start with j = 1. All values in the S-boxes should be considered 4 bits wide.

2.4.4.1 Take the 1st and 6th bits of B[j] together as a 2-bit value (call it m) indicating the row in S[j] to look in for the substitution.

2.4.4.2 Take the 2nd through 5th bits of B[j] together as a 4-bit value (call it n) indicating the column in S[j] to find the substitution.

2.4.4.3 Replace B[j] with S[j][m][n].

Substitution Box 1 (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

2.4.4.4 Loop back to 2.4.4.1 until all 8 blocks have been replaced.

2.4.5 Permute the concatenation of B[1] through B[8] as indicated below.

Permutation 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

2.4.6 Exclusive-or the resulting value with L[i-1]. Thus, all together, your R[i] = L[i-1] xor P(S[1](B[1])...S[8](B[8])), where B[j] is a 6-bit block of E(R[i-1]) xor K[i]. (The function for R[i] is more concisely written as, R[i] = L[i-1] xor f(R[i-1], K[i]).)

2.4.7 L[i] = R[i-1].

2.4.8 Loop back to 2.4.1 until K[16] has been applied.

2.5 Perform the following permutation on the block R[16]L[16]. (Note that block R precedes block L this time.)

Final 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

This has been a description of how to use the DES algorithm to encrypt one 64-bit block. To decrypt, use the same process, but just use the keys K[i] in reverse order. That is, instead of applying K[1] for the first iteration, apply K[16], and then K[15] for the second, on down to K[1].

Summaries:

Key schedule:
C[0]D[0] = PC1(key)
for 1 <= i <= 16
C[i] = LS[i](C[i-1])
D[i] = LS[i](D[i-1])
K[i] = PC2(C[i]D[i])

Encipherment:
L[0]R[0] = IP(plain block)
for 1 <= i <= 16
L[i] = R[i-1]
R[i] = L[i-1] xor f(R[i-1], K[i])
cipher block = FP(R[16]L[16])

Decipherment:
R[16]L[16] = IP(cipher block)
for 1 <= i <= 16
R[i-1] = L[i]
L[i-1] = R[i] xor f(L[i], K[i])
plain block = FP(L[0]R[0])

2.2.3 Flow Diagram of DES Algorithm

The diagrams below summarizes how the DES algorithm works.

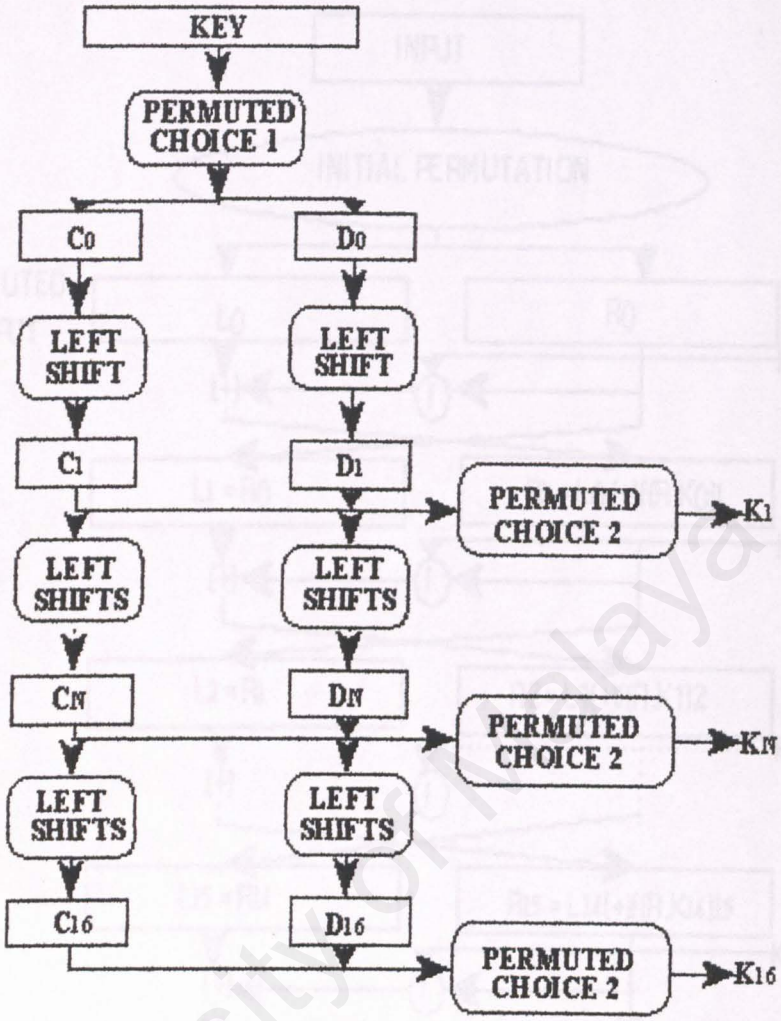


Figure 2.01 Sub-key Generator

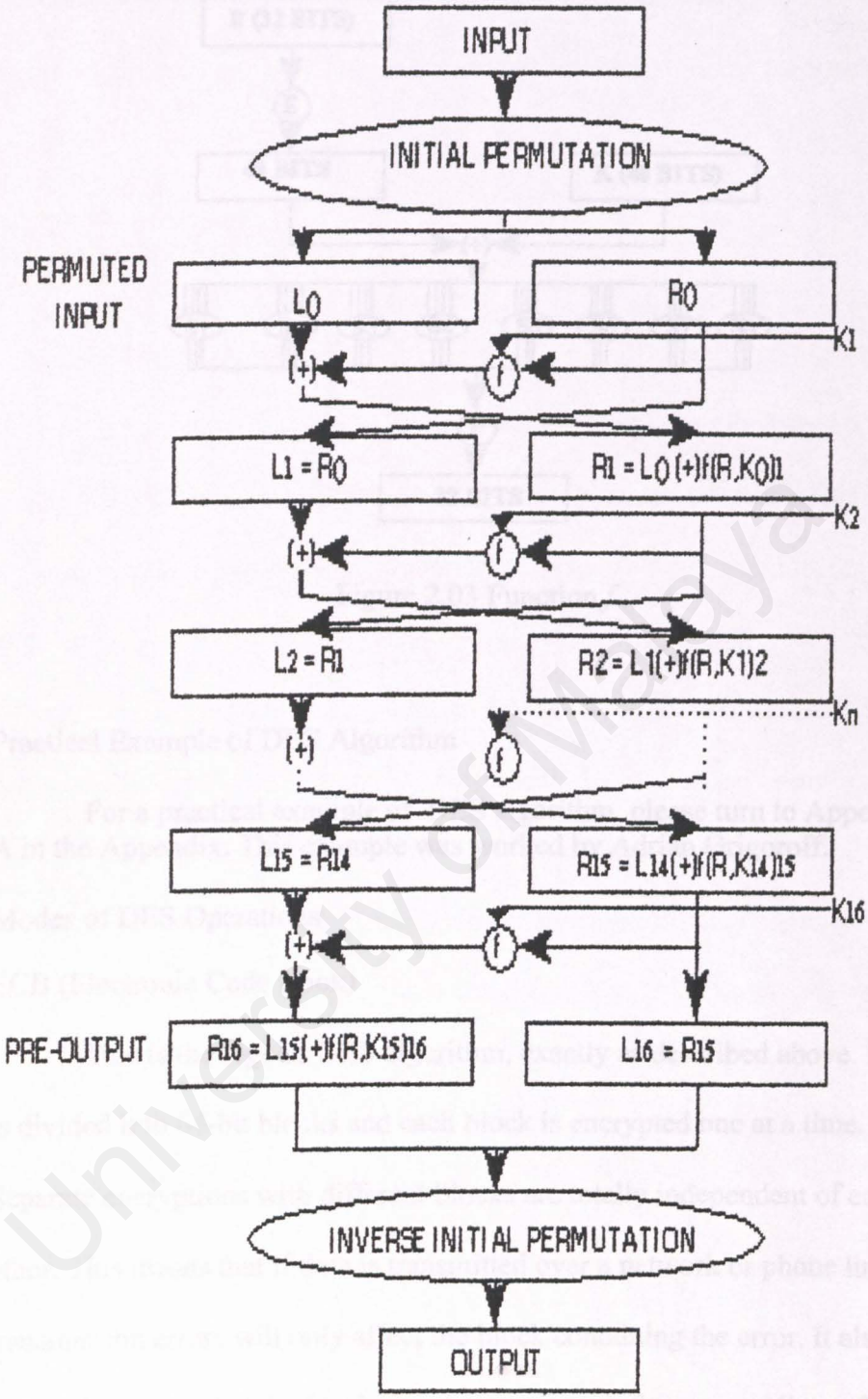


Figure 2.02 DES Core

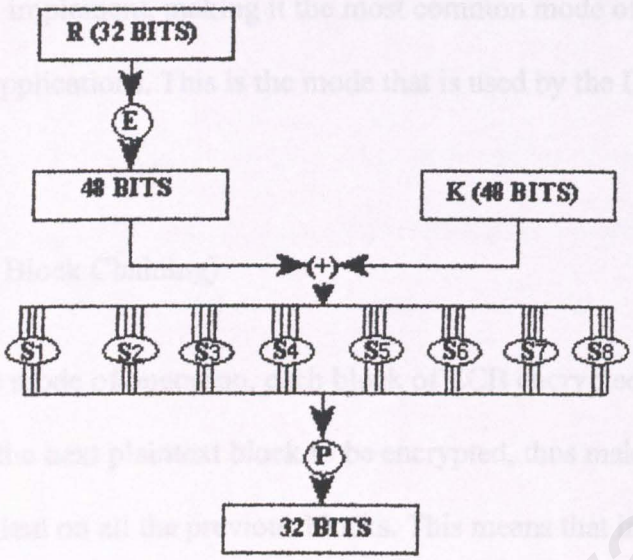


Figure 2.03 Function *f*

2.2.4 Practical Example of DES Algorithm

For a practical example of DES algorithm, please turn to Appendix A in the Appendix. This example was worked by Adrian Grigoroff.

2.2.5 Modes of DES Operations

2.2.5.1 ECB (Electronic Code Book)

This is the regular DES algorithm, exactly as described above. Data is divided into 64-bit blocks and each block is encrypted one at a time. Separate encryptions with different blocks are totally independent of each other. This means that if data is transmitted over a network or phone line, transmission errors will only affect the block containing the error. It also means, however, that the blocks can be rearranged, thus scrambling a file beyond recognition, and this action would go undetected. ECB is the weakest of the various modes because no additional security measures are implemented besides the basic DES algorithm. However, ECB is the fastest

and easiest to implement, making it the most common mode of DES seen in commercial applications. This is the mode that is used by the DES chip designed.

2.2.5.2 CBC (Cipher Block Chaining)

In this mode of operation, each block of ECB encrypted ciphertext is XORed with the next plaintext block to be encrypted, thus making all the blocks dependent on all the previous blocks. This means that in order to find the plaintext of a particular block, we need to know the ciphertext, the key, and the ciphertext for the previous block. The first block to be encrypted has no previous ciphertext, so the plaintext is XORed with a 64-bit number called the Initialization Vector, or IV for short. So if data is transmitted over a network or phone line and there is a transmission error, the error will be carried forward to all subsequent blocks since each block is dependent upon the last. This mode of operation is more secure than ECB because the extra XOR step adds one more layer to the encryption process.

2.2.5.3 CFB (Cipher Feedback)

In this mode, blocks of plaintext that are less than 64 bits long can be encrypted. Normally, special processing has to be used to handle files whose size is not a perfect multiple of 8 bytes, but this mode removes that necessity (Stealth handles this case by adding several dummy bytes to the end of a file before encrypting it). The plaintext itself is not actually passed through the

DES algorithm, but merely XORed with an output block from it, in the following manner: A 64-bit block called the Shift Register is used as the input plaintext to DES. This is initially set to some arbitrary value, and encrypted with the DES algorithm. The ciphertext is then passed through an extra component called the M-box, which simply selects the left-most M bits of the ciphertext, where M is the number of bits in the block we wish to encrypt. This value is XORed with the real plaintext, and the output of that is the final ciphertext. Finally, the ciphertext is fed back into the Shift Register, and used as the plaintext seed for the next block to be encrypted. As with CBC mode, an error in one block affects all subsequent blocks during data transmission. This mode of operation is similar to CBC and is very secure, but it is slower than ECB due to the added complexity.

2.2.5.4 OFB (Output Feedback)

This is similar to CFB mode, except that the ciphertext output of DES is fed back into the Shift Register, rather than the actual final ciphertext. The Shift Register is set to an arbitrary initial value, and passed through the DES algorithm. The output from DES is passed through the M-box and then fed back into the Shift Register to prepare for the next block. This value is then XORed with the real plaintext (which may be less than 64 bits in length, like CFB mode), and the result is the final ciphertext. Note that unlike CFB and CBC, a transmission error in one block will not affect subsequent blocks because once the recipient has the initial Shift Register

value, it will continue to generate new Shift Register plaintext inputs without any further data input. However, this mode of operation is less secure than CFB mode because only the real ciphertext and DES ciphertext output is needed to find the plaintext of the most recent block. Knowledge of the key is not required.

2.2.6 Variation of DES - Triple DES

2.2.6.1 Introduction

Triple DES is a minor variation of this standard. It is three times slower than regular DES but can be billions of times more secure if used properly. Triple DES enjoys much wider use than DES because DES is so easy to break with today's rapidly advancing technology. In 1998 the Electronic Frontier Foundation, using a specially developed computer called the DES Cracker, managed to break DES in less than 3 days. And this was done for under $250,000. The encryption chip that powered the DES Cracker was capable of processing 88 billion keys per second. In addition, it has been shown that for a cost of one million dollars a dedicated hardware device can be built that can search all possible DES keys in about 3.5 hours. This just serves to illustrate that any organization with moderate resources can break through DES with very little effort these days. No sane security expert would consider using DES to protect data.

Triple DES was the answer to many of the shortcomings of DES. Since it is based on the DES algorithm, it is very easy to modify existing software to use Triple DES. It also has the advantage of proven reliability and a longer key length that eliminates many of the shortcut attacks that can be used to reduce the amount of time it takes to break DES. However, even this more powerful version of DES may not be strong enough to protect data for very much longer. The DES algorithm itself has become obsolete and is in need of replacement. To this end the National Institute of Standards and Technology (NIST) is holding a competition to develop the Advanced Encryption Standard (AES) as a replacement for DES. Triple DES has been endorsed by NIST as a temporary standard to be used until the AES is finished sometime in 2001.

The AES will be at least as strong as Triple DES and probably much faster. Many security systems will probably use both Triple DES and AES for at least the next five years. After that, AES may supplant Triple DES as the default algorithm on most systems if it lives up to its expectations. But Triple DES will be kept around for compatibility reasons for many years after that. So the useful lifetime of Triple DES is far from over, even with the AES near completion. For the foreseeable future Triple DES is an excellent and reliable choice for the security needs of highly sensitive information.

2.2.6.2 In Depth

Triple DES is simply another mode of DES operation. It takes three 64-bit keys, for an overall key length of 192 bits. In Stealth, you simply type in the entire 192-bit (24 character) key rather than entering each of the three keys individually. The Triple DES DLL then breaks the user provided key into three subkeys, padding the keys if necessary so they are each 64 bits long. The procedure for encryption is exactly the same as regular DES, but it is repeated three times. Hence the name Triple DES. The data is encrypted with the first key, decrypted with the second key, and finally encrypted again with the third key.

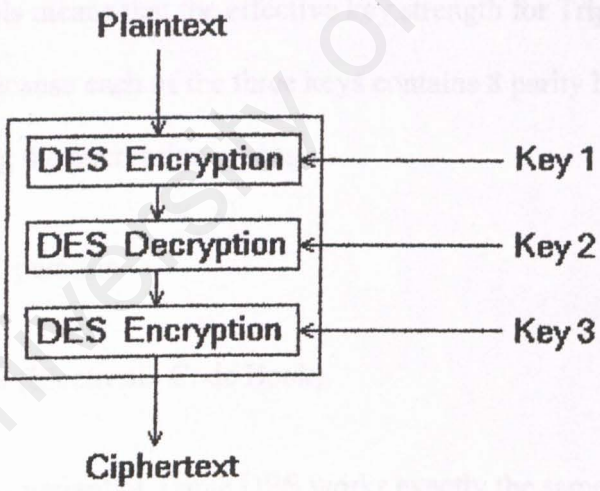


Figure 2.04 Diagram of Triple DES

Consequently, Triple DES runs three times slower than standard DES, but is much more secure if used properly. The procedure for decrypting something is the same as the procedure for encryption, except it is executed

in reverse. Like DES, data is encrypted and decrypted in 64-bit chunks. Unfortunately, there are some weak keys that one should be aware of: if all three keys, the first and second keys, or the second and third keys are the same, then the encryption procedure is essentially the same as standard DES. This situation is to be avoided because it is the same as using a really slow version of regular DES.

Note that although the input key for DES is 64 bits long, the actual key used by DES is only 56 bits in length. The least significant (right-most) bit in each byte is a parity bit, and should be set so that there are always an odd number of 1s in every byte. These parity bits are ignored, so only the seven most significant bits of each byte are used, resulting in a key length of 56 bits. This means that the effective key strength for Triple DES is actually 168 bits because each of the three keys contains 8 parity bits that are not used during the encryption process.

2.2.6.3 Modes of Operation

Triple ECB (Electronic Code Book)

This variant of Triple DES works exactly the same way as the ECB mode of DES. This is the most commonly used mode of operation.

Triple CBC (Cipher Block Chaining)

This method is very similar to the standard DES CBC mode. As with Triple ECB, the effective key length is 168 bits and keys are used in the same manner, as described above, but the chaining features of CBC mode are also employed. The first 64-bit key acts as the Initialization Vector to DES. Triple ECB is then executed for a single 64-bit block of plaintext. The resulting ciphertext is then XORed with the next plaintext block to be encrypted, and the procedure is repeated. This method adds an extra layer of security to Triple DES and is therefore more secure than Triple ECB, although it is not used as widely as Triple ECB.

2.2 Introduction to VHDL

VHDL is a language for describing digital electronic systems. It arose out of the United States Government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of integrated circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed, and subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE) in the US.

VHDL is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a design, that is how it is decomposed into sub-designs, and how those sub-designs are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, as a result, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.

For our introduction, I will only touch on the lexical elements and main language constructs. I will also touch on a bit on the levels of abstraction used in the language. This is based on VHDL '93 specification.

2.2.1 Lexical Elements

The behaviour of a module may be described in programming language form. This chapter describes the facilities in VHDL which are drawn from the familiar programming language repertoire.

2.2.1.1 Comments

Comments in VHDL start with two adjacent hyphens ('--') and extend to the end of the line. They have no part in the meaning of a VHDL description.

2.2.1.2 Identifiers

Identifiers in VHDL are used as reserved words and as programmer defined names. They must conform to the rule:

identifier ::= letter { [underline] letter_or_digit }

Note that case of letters is not considered significant, so the identifiers cat and Cat are the same. Underline characters in identifiers are significant, so This_Name and ThisName are different identifiers.

2.2.1.3 Numbers

Literal numbers may be expressed either in decimal or in a base between two and sixteen. If the literal includes a point, it represents a real number, otherwise it represents an integer. Decimal literals are defined by:

decimal_literal ::= integer [. integer] [exponent]

integer ::= digit { [underline] digit }

exponent ::= E [+] integer | E - integer

Some examples are:

0 1 123_456_789 987E6 -- integer literals

0.0 0.5 2.718_28 12.4E-9 - - real literals

Based literal numbers are defined by:

based_literal ::= base # based_integer [. based_integer] # [exponent]

base ::= integer

based_integer ::= extended_digit { [underline] extended_digit }

extended_digit ::= digit | letter

The base and the exponent are expressed in decimal. The exponent indicates the power of the base by which the literal is multiplied. The letters A to F (upper or lower case) are used as extended digits to represent 10 to 15. Some examples:

2#1100_0100# 16#C4# 4#301#E1 -- the integer 196

2#1.1111_1111_111#E+11 16#F.FF#E2 -- the real number 4095.0

2.2.1.4 Characters

Literal characters are formed by enclosing an ASCII character in single-quote marks. For example:

'A' '*' ''' ' '

2.2.1.5 Strings

Literal strings of characters are formed by enclosing the characters in double-quote marks. To include a double-quote mark itself in a string, a pair

of double-quote marks must be put together. A string can be used as a value for an object which is an array of characters. Examples of strings:

```
"A string"
""                                                    --
empty string
"A string in a string: ""A string"". "        -- contains quote marks
```

2.2.1.6 Bit Strings

VHDL provides a convenient way of specifying literal values for arrays of type bit ('0's and '1's, see Section 2.2.5). The syntax is:

```
bit_string_literal ::= base_specifier " bit_value "

base_specifier ::= B | O | X

bit_value ::= extended_digit { [ underline ] extended_digit }
```

Base specifier B stands for binary, O for octal and X for hexadecimal. Some examples:

```
B"1010110"   -- length is 7
O"126"       -- length is 9, equivalent to B"001_010_110"
X"56"        -- length is 8, equivalent to B"0101_0110"
```

2.2.2 VHDL Language Constructs

VHDL is made up of these 5 primary constructs. They are :

- Entities and Architectures
- Package

- Package Bodies
- Configuration

2.2.2.1 Entities and architectures

Entities and Architectures

Every VHDL design description consists of at least one entity/architecture pair.

Large design, many entity/architecture pairs and connect them together to form a complete circuit.

entity declaration describes the circuit as it appears from the "outside" - from the perspective of its input and output interfaces.

Example:-

```
entity fulladder is
  port (X: in bit;
        Y: in bit;
        Cin: in bit;
        Cout: out bit;
        Sum: out bit);
end fulladder;
```

A VHDL *architecture declaration* is a statement (beginning with the **architecture** keyword) that describes the underlying function and/or structure of a circuit.

- Package Bodies
- Configuration

2.2.2.1 Entities and architectures

Entities and Architectures

Every VHDL design description consists of at least one entity/architecture pair.

Large design, many entity/architecture pairs and connect them together to form a complete circuit.

entity declaration describes the circuit as it appears from the "outside" - from the perspective of its input and output interfaces.

Example:-

```
entity fulladder is
  port (X: in bit;
        Y: in bit;
        Cin: in bit;
        Cout: out bit;
        Sum: out bit);
end fulladder;
```

A VHDL *architecture declaration* is a statement (beginning with the **architecture** keyword) that describes the underlying function and/or structure of a circuit.

Example:-

```
architecture concurrent of fulladder is
begin
   Sum <= X xor Y xor Cin;
   Cout <= (X and Y) or (X and Cin) or (Y and Cin);
end concurrent;
```

2.2.2.2 Packages and Package bodies

A VHDL package declaration is identified by the **package** keyword, and is used to collect commonly-used declarations for use globally among different design units.

A package can consist of two basic parts: a package declaration and an optional package body. Package declarations can contain the following types of statements:

- Type and subtype declarations
- Constant declarations
- Global signal declarations
- Function and procedure declarations
- Attribute specifications
- File declarations
- Component declarations
- Alias declarations
- Disconnect specifications

- Use clauses

Example:-

```
package conversion is
   function to_vector (size: integer; num: integer) return std_logic_vector;
end conversion;

package body conversion is
   function to_vector(size: integer; num: integer) return std_logic_vector is
      variable ret: std_logic_vector (1 to size);
      variable a: integer;
   begin
      a := num;
      for i in size downto 1 loop
         if ((a mod 2) = 1) then
            ret(i) := '1';
         else
            ret(i) := '0';
         end if;
         a := a / 2;
      end loop;
      return ret;
   end to_vector;
end conversion;
```

2.2.2.3 Configuration

The final type of design unit available in VHDL is called a configuration declaration. A configuration declaration (identified with the **configuration** keyword) specifies which architectures are to be bound to which entities, and it allows you to change how components are connected in your design description at the time of simulation.

Configuration declarations are always optional, no matter how complex a design description you create. In the absence of a configuration declaration, the VHDL standard specifies a set of rules that provide you with a default configuration. For example, in the case where you have provided more than one architecture for an entity, the last architecture compiled will take precedence and will be bound to the entity.

Example:-

```
configuration this_build of rcomp is
  for structure
    for COMP1: compare use entity work.compare(compare1);
    for ROT1: rotate use entity work.rotate(rotate1);
  end for;
end this_build;
```

2.2.3 Levels of abstraction

VHDL supports many possible styles of design description. These styles differ primarily in how closely they relate to the underlying hardware. When we speak of the different styles of VHDL, we are really talking about the differing levels of abstraction possible using the language—behavior, dataflow, and structure.

Suppose the performance specifications for a given project are: "the compressed data coming out of the DSP chip needs to be analyzed and stored within 70 nanoseconds of the strobe signal being asserted..." This human language specification must be refined into a description that can actually be simulated. A test bench written in combination with a sequential description is one such expression of the design. These are all points in the **behavior** level of abstraction.

After this initial simulation, the design must be further refined until the description is something a VHDL synthesis tool can digest. Synthesis is a process of translating an abstract concept into a less-abstract form. The highest level of abstraction accepted by today's synthesis tools is the **dataflow** level.

The **structure** level of abstraction comes into play when little chunks of circuitry are to be connected together to form bigger circuits. (If the little chunks being connected are actually quite large chunks, then the result is what we commonly call a block diagram.) Physical information is the most basic level of all and is outside the scope of VHDL. This level involves

actually specifying the interconnects of transistors on a chip, placing and routing macrocells within a gate array or FPGA, etc.

Note: In some formal discussions of synthesis, four levels of abstraction are described; behavior, RTL, gate-level and layout. It is our view that the three levels of abstraction presented here provide the most useful distinctions for today's synthesis user.

As an example of these three levels of abstraction, it is possible to describe a complex controller circuit in a number of ways. At the lowest level of abstraction (the structural level), we could use VHDL's hierarchy features to connect a sequence of predefined logic gates and flip-flips to form the complete circuit. To describe this same circuit at a dataflow level of abstraction, we might describe the combinational logic portion of the controller (its input decoding and transition logic) using higher-level Boolean logic functions and then feed the output of that logic into a set of registers that match the registers available in some target technology. At the behavioral level of abstraction, we might ignore the target technology (and the requirements of synthesis tools) entirely and instead describe how the controller operates over time in response to various types of stimulus.

2.2.3.1 Behavior

The highest level of abstraction supported in VHDL is called the *behavioral* level of abstraction. When creating a behavioral description of a circuit, you will describe your circuit in terms of its operation *over time*.

The concept of time is the critical distinction between behavioral descriptions of circuits and lower-level descriptions (specifically descriptions created at the dataflow level of abstraction).

Examples of behavioral forms of representation might include state diagrams, timing diagrams and algorithmic descriptions.

In a behavioral description, the concept of time may be expressed precisely, with actual delays between related events (such as the propagation delays within gates and on wires), or it may simply be an ordering of operations that are expressed sequentially (such as in a functional description of a flip-flop). When you are writing VHDL for input to synthesis tools, you may use behavioral statements in VHDL to imply that there are registers in your circuit. It is unlikely, however, that your synthesis tool will be capable of creating precisely the same behavior in actual circuitry as you have defined in the language. (Synthesis tools today ignore detailed timing specifications, leaving the actual timing results at the mercy of the target device technology.) It is also unlikely that your synthesis tool will be capable of accepting and processing a very wide range of behavioral description styles.

If you are familiar with software programming, writing behavior-level VHDL will not seem like anything new. Just like a programming language, you will be writing one or more small programs that operate sequentially and communicate with one another through their interfaces. The only difference between behavior-level VHDL and a software

programming language is the underlying execution platform: in the case of software, it is some operating system running on a CPU; in the case of VHDL, it is the simulator and/or the synthesized hardware.

2.2.3.2 Dataflow

In the dataflow level of abstraction, you describe your circuit in terms of how data moves through the system. At the heart of most digital systems today are registers, so in the dataflow level of abstraction you describe how information is passed between registers in the circuit. You will probably describe the combinational logic portion of your circuit at a relatively high level (and let a synthesis tool figure out the detailed implementation in logic gates), but you will likely be quite specific about the placement and operation of registers in the complete circuit.

The dataflow level of abstraction is often called *register transfer logic*, or RTL. This level of abstraction is an intermediate level that allows the drudgery of combinational logic to be simplified (and, presumably, taken care of by logic synthesis tools) while the more important parts of the circuit, the registers, are more completely specified.

There are some drawbacks to using a dataflow method of design in VHDL. First, there are no built-in registers in VHDL; the language was designed to be general-purpose, and the emphasis was placed by VHDL's designers on its behavioral aspects. If you are going to write VHDL at the dataflow level of abstraction, you must first create (or obtain) behavioral descriptions of the register elements you will be using in your design.

These elements must be provided in the form of components (using VHDL's hierarchy features) or in the form of subprograms (functions or procedures).

But for hardware designers, it can be difficult to relate the sequential descriptions and operation of behavioral VHDL with the hardware being described (or *modeled*). For this reason, many VHDL users, particularly those who are using VHDL as an input to synthesis, prefer to stick with levels of abstraction that are easier to relate to actual hardware devices (such as logic gates and flip-flops). These users are often more comfortable using the dataflow level of abstraction.

2.2.3.3 Structure

The third level of abstraction, *structure*, is used to describe a circuit in terms of its components. Structure can be used to create a very low-level description of a circuit (such as a transistor-level description) or a very high-level description (such as a block diagram).

In a gate-level description of a circuit, for example, components such as basic logic gates and flip-flops might be connected in some logical structure to create the circuit. This is what is often called a *netlist*. For a higher-level circuit—one in which the components being connected are larger functional blocks—structure might simply be used to segment the design description into manageable parts.

Structure-level VHDL features, such as components and configurations, are very useful for managing complexity. The use of components can dramatically improve your ability to re-use elements of your designs, and they can make it possible to work using a *top-down* design approach.

To give an example of how a structural description of a circuit relates to higher levels of abstraction, consider the design of a simple 5-bit counter. To describe such a counter using traditional design methods, we might connect five T flip-flops with some simple decode logic.

The following VHDL design description represents this design in the form of a netlist of connected components:

```
entity andgate is
   port(A,B,C,D: in bit := '1'; Y: out bit);
end andgate;

architecture gate of andgate is
begin
   Y <= A and B and C and D;
end gate;

entity tff is
   port(Rst,Clk,T: in bit; Q: out bit);
end tff;

architecture behavior of tff is
begin
   process(Rst,Clk)
      variable Qtmp: bit;
   begin
      if (Rst = '1') then
         Qtmp := '0';
      elsif Clk = '1' and Clk'event then
         if T = '1' then
            Qtmp := not Qtmp;
         end if;
      end if;
      Q <= Qtmp;
   end process;
end behavior;
```

```
entity TCOUNT is
   port (Rst,Clk: in bit;
         Count: out bit_vector(4 downto 0));
end TCOUNT;

architecture STRUCTURE of TCOUNT is
   component tff
      port(Rst,Clk,T: in bit; Q: out bit);
   end component;
   component andgate
      port(A,B,C,D: in bit := '1'; Y: out bit);
   end component;
   constant VCC: bit := '1';
   signal T,Q: bit_vector(4 downto 0);

begin
   T(0) <= VCC;
   T0: tff port map (Rst=>Rst, Clk=>Clk, T=>T(0), Q=>Q(0));
   T(1) <= Q(0);
   T1: tff port map (Rst=>Rst, Clk=>Clk, T=>T(1), Q=>Q(1));
   A1: andgate port map(A=>Q(0), B=>Q(1), Y=>T(2));
   T2: tff port map (Rst=>Rst, Clk=>Clk, T=>T(2), Q=>Q(2));
   A2: andgate port map(A=>Q(0), B=>Q(1), C=>Q(2), Y=>T(3));
   T3: tff port map (Rst=>Rst, Clk=>Clk, T=>T(3), Q=>Q(3));
   A3: andgate port map(A=>Q(0), B=>Q(1), C=>Q(2), D=>Q(3), Y=>T(4));
   T4: tff port map (Rst=>Rst, Clk=>Clk, T=>T(4), Q=>Q(4));

   Count <= Q;

end STRUCTURE;
```

This structural representation seems a straightforward way to describe a 5-bit counter, and it is certainly easy to relate to hardware since just about any imaginable implementation technology will have the features necessary to implement the circuit. For larger circuits, however, such descriptions quickly become impractical.

2.3 Why DES in VHDL?

The appropriate question should be, why DES in electronic devices? Since VHDL is a development language for digital electronic systems, we discuss the benefits of implementing DES in such devices.

[FIPS 74] Implementation of the DES algorithm in special purpose electronic devices provides the following economic and security benefits:

1. Efficiency of algorithm operation is much higher in specialized electronic devices.
2. Basic implementation of the algorithm in specialized LSI electronic devices which can be used in many applications and environments should result in cost savings to the user through high volume production.
3. Functional operation of the device may be tested and validated independently of the environment in which it is used.
4. An encryption key may be entered directly into the device without appearing elsewhere in the computer system.
5. Unauthorized modification of the algorithm is very difficult in such a device.
6. Independent devices may encipher the data simultaneously and the output may be tested before the cipher is transmitted.
7. The control and data paths, to and from the device, may be controlled and monitored.

Chapter Three

Methodology

3.0 Introduction

This chapter describes the methodology used in developing our cryptosystem. The first part is the definition of methodology (3.1). Then, this is followed by the selected project's life-cycle (3.2). The next part (3.3) is the explanation of real-world digital systems design process, and the methodology used. The final part is this chapter's conclusion.

3.1 What is methodology?

methodology noun

a system of ways of doing, teaching or studying something:

The methodology and findings of the research team have been criticized.

(from Cambridge Advanced Learner's Dictionary)

In the development of computer systems, be it in software, hardware or a combination of both, an appropriate methodology must be selected to ensure a smooth and systematic running of operation to achieve designated goals.

3.2.0 Project's Life-Cycle Model

For our project, the appropriate life-cycle would be the "Cascading-Waterfall" model. It is very straightforward and sequential, with every step following the other. Figure 3.01 shows the diagram of "Cascading-Waterfall" process flow.

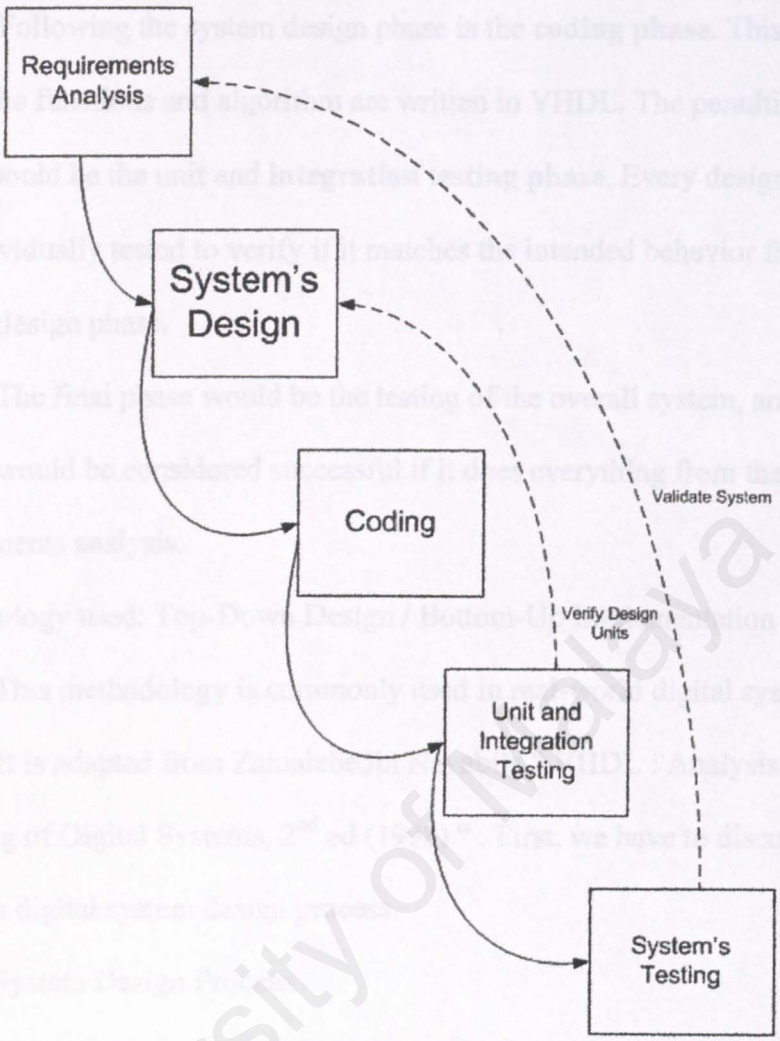


Figure 3.01 DES Process-flow Diagram

3.2.1 Brief explanation for process flow

We begin with the **requirements analysis**. Here, we analyze all that is needed by the system, like it's intended input, and expected output. This is done by understanding the algorithms involved. Then, from the algorithms, we extract all the functions used to describe the behavior of DES.

Next, we proceed to the **system design** phase. Here, we will design the sub modules and data paths for our DES cryptosystem.

Following the system design phase is the **coding phase**. This is where the functions and algorithm are written in VHDL. The penultimate phase would be the unit and **integration testing phase**. Every design unit are individually tested to verify if it matches the intended behavior from the system design phase.

The final phase would be the testing of the overall system, and the system would be considered successful if it does everything from the requirements analysis.

3.3.0 Methodology used: Top-Down Design / Bottom-Up Implementation

This methodology is commonly used in real-world digital systems design. It is adapted from Zainalabedin Navabi's " VHDL : Analysis and modeling of Digital Systems, 2nd ed (1998) " . First, we have to discuss a common digital system design process.

3.3.1 Digital System Design Process

Figure 3.02 shows a typical process for the design of digital systems. An initial design goes through several transformation before it's hardware implementation is obtained.

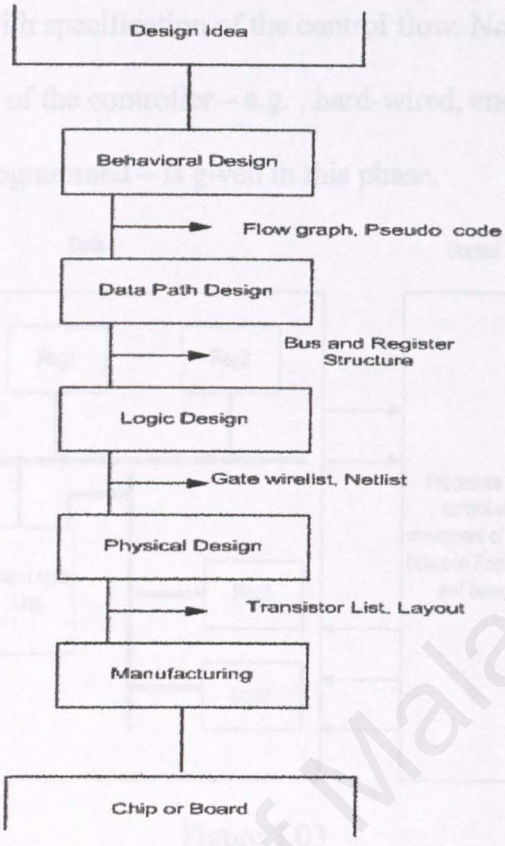


Figure 3.02 Digital Systems Design Process

Initially, a hardware designer starts with a design idea. A more complete definition of the intended hardware must then be developed from the designed idea. Therefore, the designer must generate general behavioral definition of system under design. It could be represented by pseudocode, flow-chart or flow-graph. The designer specifies the overall functionality without architectural or hardware details of system under design.

The next phase is designing system data path. Here, the designer specifies registers and logic units necessary for implementation of system. Components may be interconnected by uni- or bi-directional busses. Data components communicate via busses, while control procedure controls flow of data within components. As shown in Fig 3.03, this phase shows

architectural design with specification of the control flow. No information about implementation of the controller – e.g. , hard-wired, encoding technique, or microprogrammed – is given in this phase.

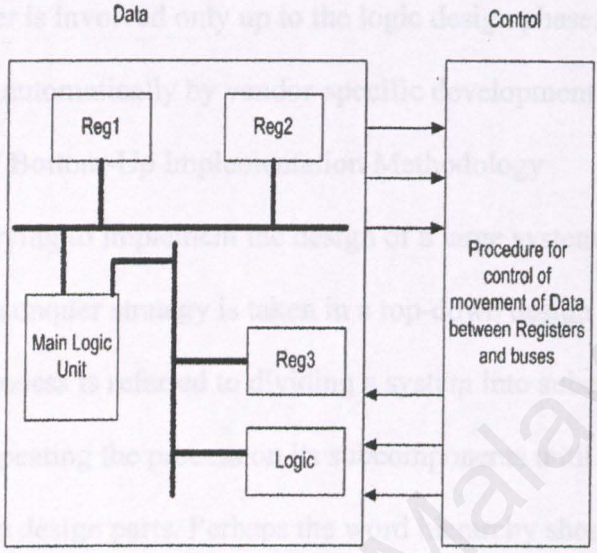


Figure 3.03

Logic design is the next step and this phase involves the use of primitive flip-flops and gates for implementation of registers, busses, logic units and their controlling hardware. The result of this stage is the netlist of gates and flip-flops. Components used and their interconnections are specified in this netlist. Gate technology and even gate-level details of flip-flops are not included in this netlist

The next design stage transforms the netlist into a transistor list or layout. This involves the replacement of gates and flip-flops with transistor equivalents or library cells. This stage considers loading and timing requirements in its cell or transistor selection process.

The final step in the design is manufacturing, which uses transistor list or layout specification to burn fuses of field-programmable device to generate masks for integrated-circuit fabrication.

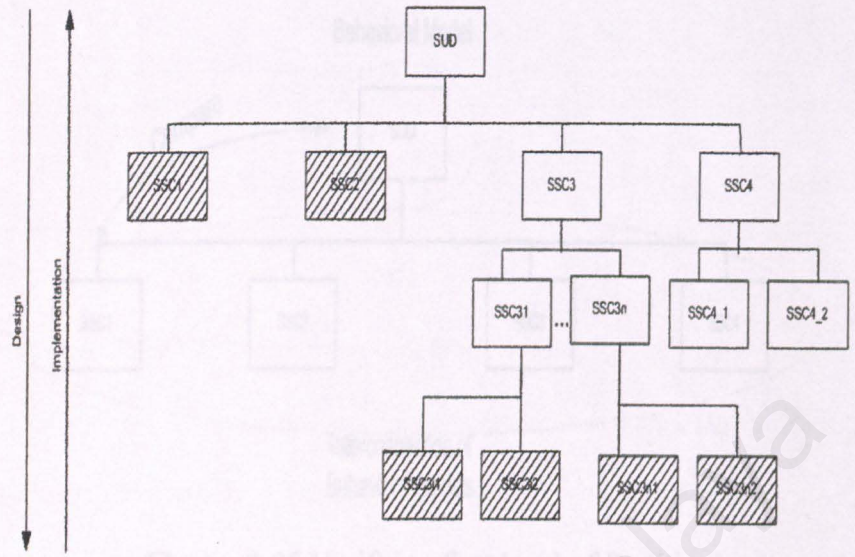
The designer is involved only up to the logic design phase. The other two phases is done automatically by vendor-specific development tools.

3.3.2 Top-Down Design/ Bottom-Up Implementation Methodology

Instead of trying to implement the design of a large system all at once, a divide-and-conquer strategy is taken in a top-down design process. A top-down design process is referred to dividing a system into subcomponents, and if necessary, repeating the process on its subcomponents until all become manageable design parts. Perhaps the word hierarchy should best explain the system and its subcomponent. Each level of dividing component/subcomponent is referred to as partitioning. Design of a component is manageable if the component is available as part of a library, it can be implemented by modifying existing design parts, or built from scratch by the designer.

Figure 3.04 shows the original design initially described at behavioral level. In the first level of partitioning, two of its subcomponents (SSC1 and SSC2) are mapped to hardware. Further partitioning for hardware implementation is required for SSC3 and SSC4. SSC3 subcomponent is partitioned into *n* numbers of identical subcomponent, and each of these is realized by SSC3i1 and SSC3i2 hardware parts. The SSC4 subcomponent is

partitioned into SSC4_1 and SSC4_2 in which hardware implementations are available.



SUD: System under design

SSC: System Subcomponent

Shaded areas designate subcomponents with hardware implementations

Figure 3.04 Top-down design/ Bottom-Up Implementation

3.3.3 Verification

At each level of top-down design, multilevel simulation tool plays an important role in the correct implementation of the design. Initially a behavioral description of a system under design (SUD) must be simulated to verify the designer's understanding of the problem. After the first level of partitioning, the behavioral description of each subcomponent must be developed, and these descriptions must be wired to form a structural hardware model of SUD. Simulation of this new model and comparing the results of the original SUD description will verify the correctness of the first

level of partitioning. Figure 3.05 shows simulation of the first level of partitioning of the top-down design tree.

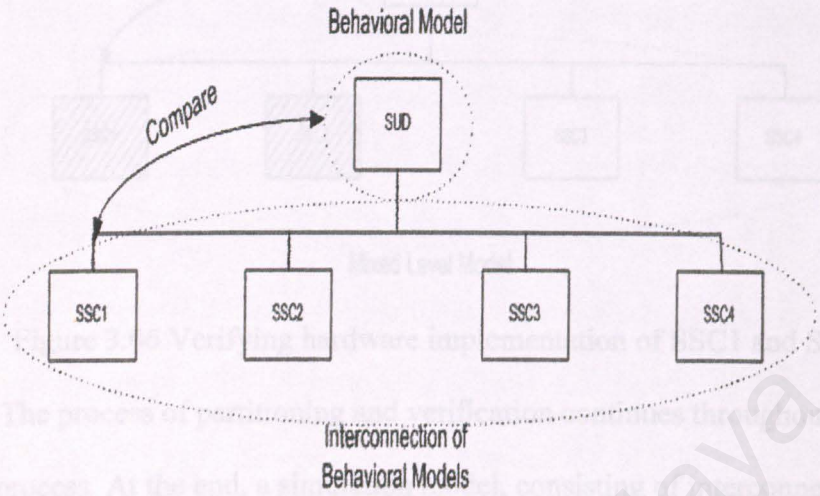


Figure 3.05 Verifying first level of Partitioning

After verifying the first level of partitioning, the hardware implementation of SC1 and SC2 must be verified. Another simulation is run, with the behavioral description of SC1 and SC2 replaced by a more detailed hardware level model. Figure 3.06 shows this phase. Shaded boxes represent component models, which are hardware implementation that are functionally equivalent to its behavioral counterparts. It has representation for physical characteristics of the hardware. Typical physical characteristics are timing, power consumption, and temperature dependencies. Such models are referred to as hardware-level model.

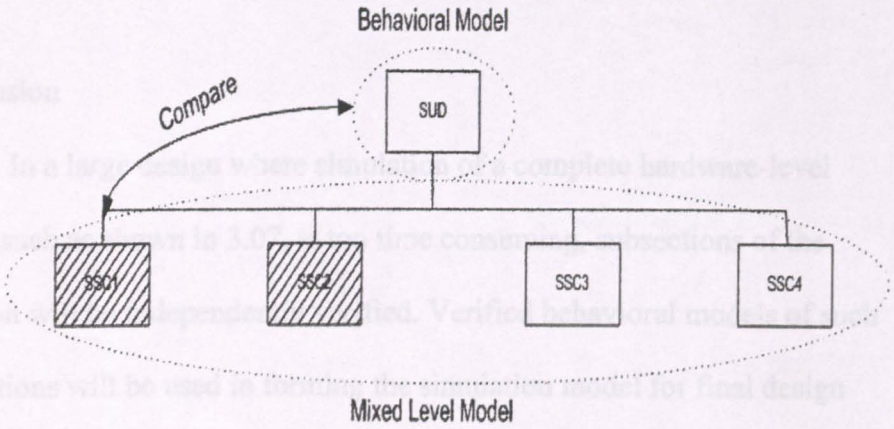


Figure 3.06 Verifying hardware implementation of SSC1 and SSC2

The process of partitioning and verification continues throughout the design process. At the end, a simulation model, consisting of interconnection specification of hardware-level models of the terminals of the partition tree, will be formed. The simulation of this model, as shown in Figure 3.07, and comparing the results with those of original behavioral description of SUD verify the correctness of the complete design.

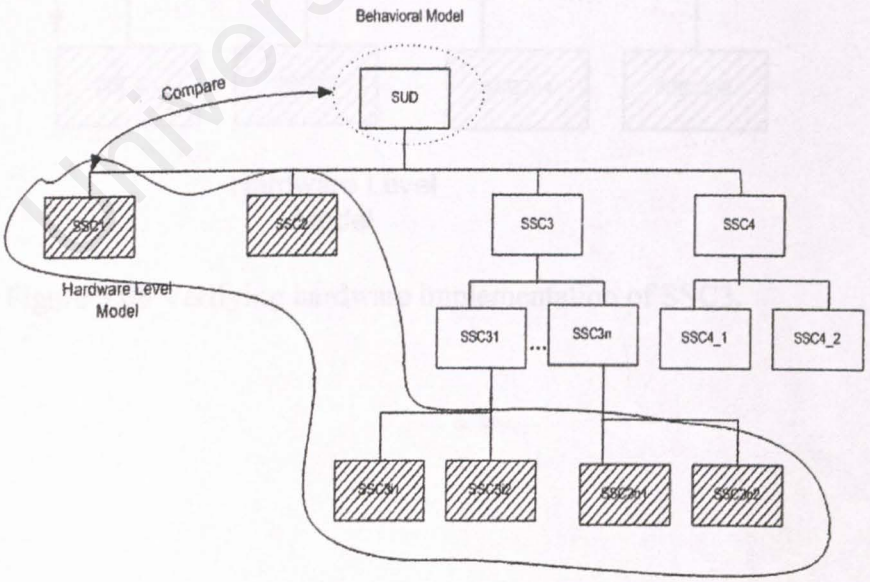


Figure 3.07 Verifying the final design

3.3.4 Conclusion

In a large design where simulation of a complete hardware-level model such as shown in 3.07, is too time consuming, subsections of the partition will be independently verified. Verified behavioral models of such subsections will be used in forming the simulation model for final design verification. Figure 3.08 shows simulation and comparison run for verifying the behavioral model of SSC3 component. Figure 3.09 shows the final design verification using the verified behavioral model of SSC3.

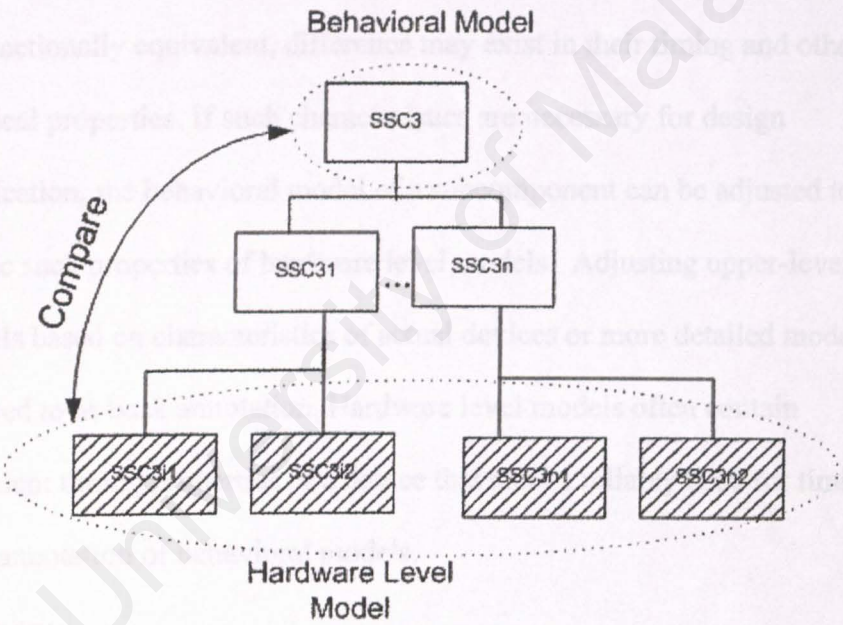


Figure 3.08 Verifying hardware implementation of SSC3.

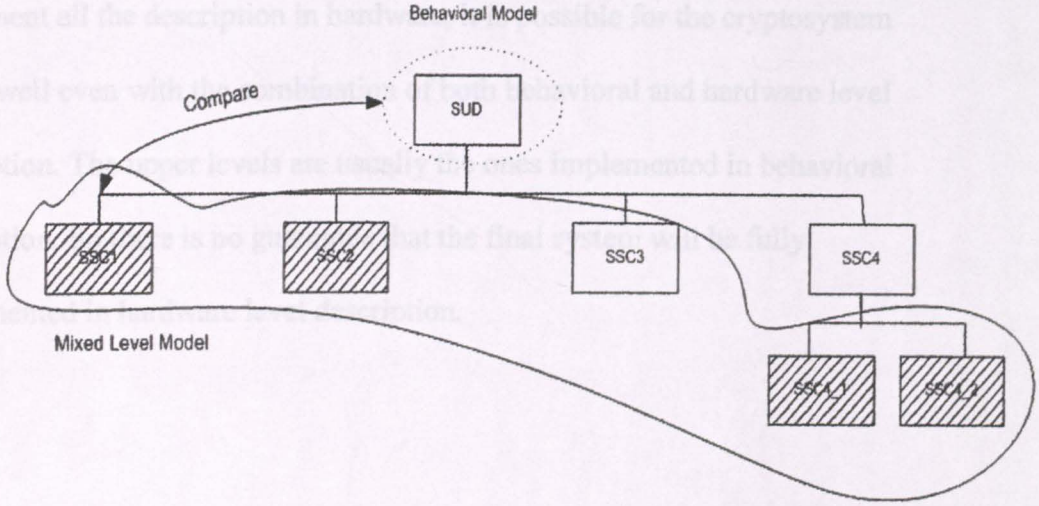


Figure 3.09 Verifying the final design, an alternative to setup 3.07

Although the behavioral- and hardware-level models of SSC3 may be functionally equivalent, difference may exist in their timing and other physical properties. If such characteristics are necessary for design verification, the behavioral model of a subcomponent can be adjusted to mimic such properties of hardware level models. Adjusting upper-level models based on characteristics of actual devices or more detailed models is referred to as back annotation. Hardware level models often contain sufficient timing properties of a device that can be reliably used for timing-back annotation of behavioral models.

3.4. Summary

The goal of this project is to develop a simulatable VHDL code for DES.

The process flow will be based on the "Cascading Waterfall" model. We would use the "divide and conquer" approach of "Top-down design/Bottom-up Implementation" methodology. Even though the eventual goal is to

implement all the description in hardware, it is possible for the cryptosystem to run well even with the combination of both behavioral and hardware level description. The upper levels are usually the ones implemented in behavioral description. So there is no guarantee that the final system will be fully implemented in hardware level description.

Chapter Four

DES Analysis

4.0 DES Analysis –An Overview

We begin by having the requirements analysis. Requirements are divided into two, which is functional requirements, and non-functional requirements. Functional requirements are the requirements needed to make the system behave the way it is supposed to behave. These include the functions extracted from the algorithm, and also the data flow of input. Non-functional requirements are requirements needed in support of executing the algorithm. This includes clock for timing purposes and also other miscellaneous input (reset, control, clear key etc.). Then, these overall functional and non-functional requirements are mapped into the sub module in our DES design. We will have a top-down view of our DES cryptosystem.

4.1 Requirements analysis

For requirements analysis, we should look at the functions used in the algorithm. DES functions involve just bit-wise operations. These operations are the:

i. shift left

ii. permutation (where values in bit positions are swapped) and

iii. XOR operations.

4.1.1 Functions in DES

There are two main blocks in DES. They are

i. Sub-key generator and

ii. The DES core.

The functions in the Sub-key generator

i. Permutation Choice 1

ii. Shift Left

iii. Permutation Choice 2

The functions in DES Core

i. Initial Permutation

ii. Function *f*

1. Expansion

2. Selection box

3. Permutation P

iii. Final Permutation

The functions are summarized by the figure 4.01 below.

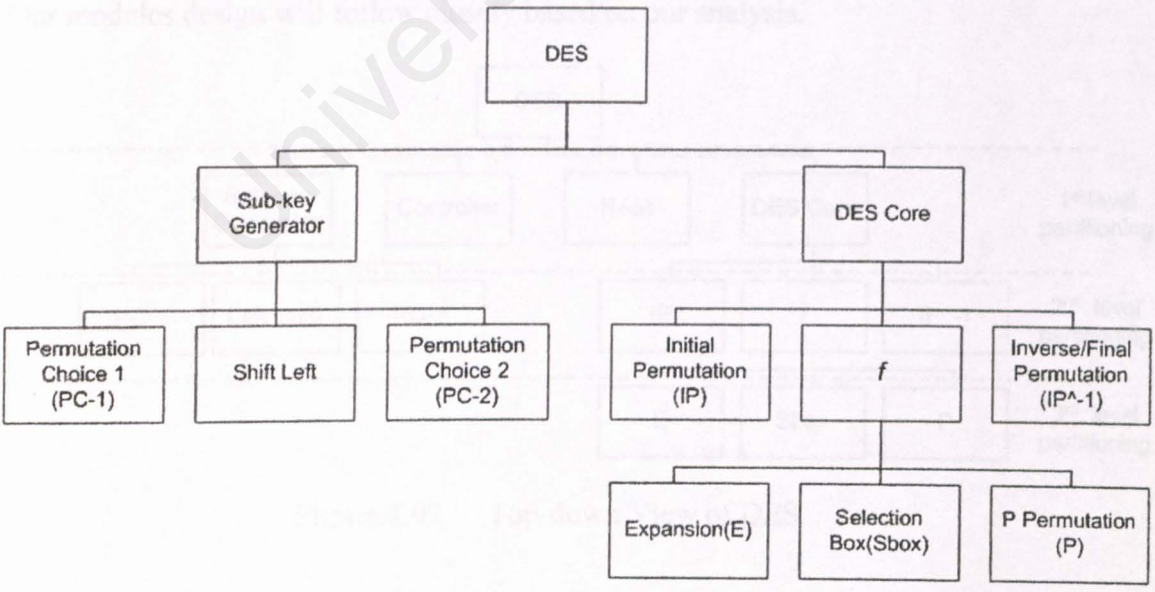


Figure 4.01 DES functions Tree Diagram

Chapter Five

DES Design

5.0 DES Design – An Overview

In this chapter, each module is seen at its block diagram level and look at its input and output. We will define the functions it contained. These functions are also in their own respective modules. Before we begin, here is a look at our DES design at functional block level. We will see the design from top down, then work it way up again for RTL level design. After that, we will see how it all works, by looking at the State machine (FSM) diagram with its detailed description.

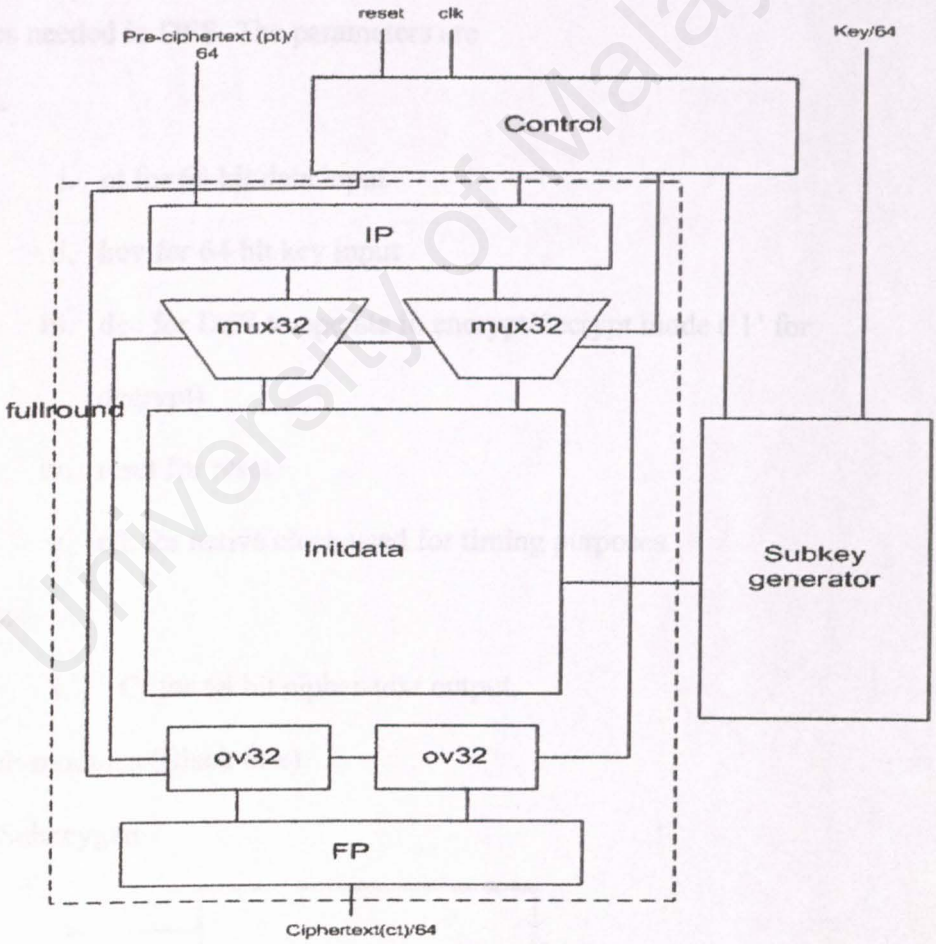


Figure 5.01 DES overall functional block diagram

5.1 DES module top level (Black box)

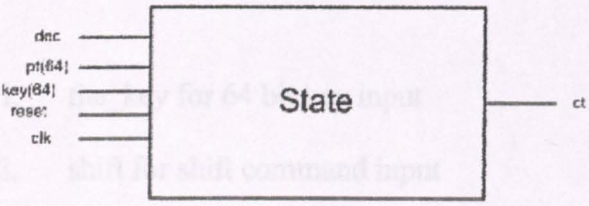


Figure 5.02 State module

Here is the top-most view of our DES design. Pt stands for pre-cipher-text, while ct stands for cipher-text. This module encapsulates all the modules needed in DES. The parameters are

Inputs:-

i. pt for 64 bit data input
ii. key for 64 bit key input
iii. dec for DES to operate in encrypt/decrypt mode ('1' for decrypt)
iv. reset for reset
v. clk for active clock used for timing purposes

Outputs:-

i. Ct for 64 bit cipher-text output.

5.2 Main sub-modules (Black box)

5.2.1 Subkeygen

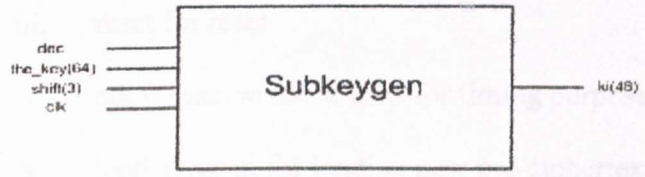


Figure 5.03 Subkeygen module

The submodules contained are PC1, shifter and PC2. More on those later. The parameters are

Inputs:-

i. the_key for 64 bit key input

ii. shift for shift command input

iii. dec for DES to operate in encrypt/decrypt mode ('1' for decrypt)

iv. clk for active clock used for timing purposes

Outputs:-

i. ki for 48 bit subkeys output entered into DES core.

5.2.2 Fullround

Figure 5.04 Fullround module

This submodules encapsulates IP, mux32, initdata and FP. The parameters are

Inputs:-

i. pt for 64 bit pre-ciphertext input.

ii. xkey for 48 bit subkeys input

iii. reset for reset

iv. clk for active clock used for timing purposes

v. load_new_pt for loading new pre-ciphertext command

vi. output_ok for output verification after 16 rounds of permutation.

Outputs:-

i. ct for 64 bit ciphertext output..

5.2.3 Control

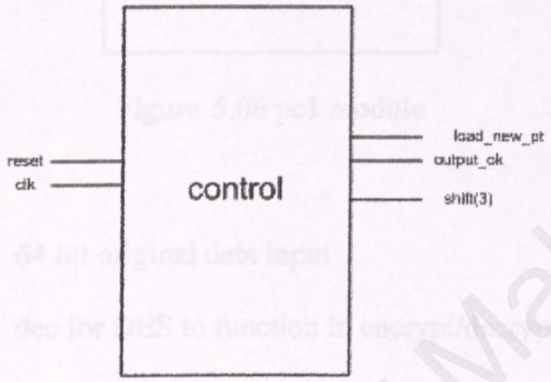


Figure 5.05 Control Module

This submodule controls the flow of our DES operation. The parameters are

Inputs:-

i. reset for reset

ii. clk for active clock used for timing purposes

Outputs:-

i. load_new_pt for loading new pre-ciphertext command

ii. output_ok for output verification after 16 rounds of permutation

iii. shift for shift command instructions

5.3 Submodules

Subkeygen

5.3.01 PC1

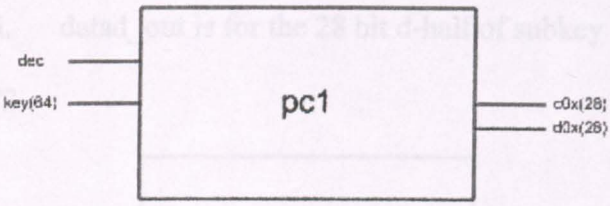


Figure 5.06 pc1 module

Inputs:-

i. 64 bit original data input

ii. dec for DES to function in encrypt/decrypt mode (1 for decrypt)

Outputs:-

i. c0x is for the 28 bit c-half of subkey into shifter

ii. d0x is for the 28 bit d-half of subkey into shifter

5.3.02 Shifter

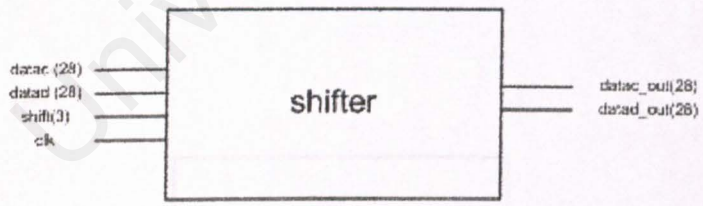


Figure 5.07 shifter module

Inputs:-

i. datac for 28 bit c-half output from pc1

ii. datad for 28 bit d-half output from pc2

iii. shift for shift control input from controller

iv. clk for clock input

Outputs:-

i. datac_out is for the 28 bit c-half of subkey into pc2

ii. datad_out is for the 28 bit d-half of subkey into pc2

5.3.03 PC2

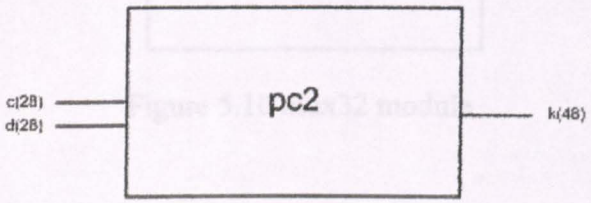


Figure 5.08 pc2 module

Inputs:-

i. c for 28 bit c-half of subkey

ii. d for 28 bit d-half of subkey

Outputs:-

i. c0x is for the 28 bit c-half of subkey

ii. d0x is for the 28 bit d-half of subkey

Fullround

5.3.04 IP

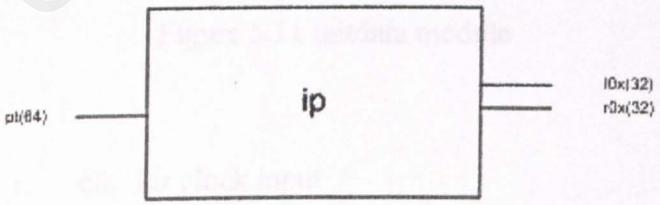


Figure 5.09 ip module

5.3.05 Mux32

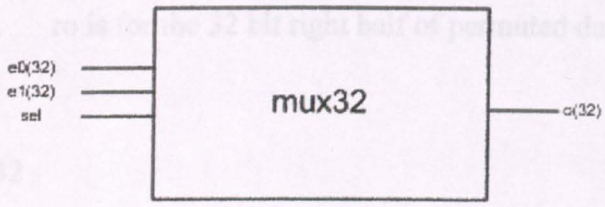


Figure 5.10 mux32 module

Inputs:-

i. 32 bit e0 to keep data for permutation

ii. 32 bit e1 to keep data for permutation

iii. sel to start the first of 16 rounds of permutation

Outputs:-

i. 32 bit permuted data

5.3.06 Initdata

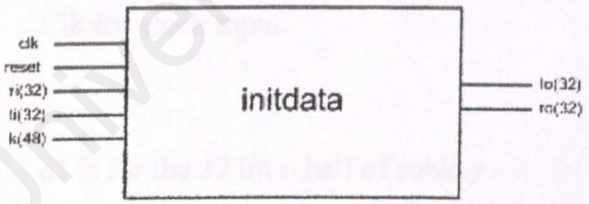


Figure 5.11 initdata module

Inputs:-

i. clk for clock input

ii. reset for reset

iii. ri for right half input of data

iv. li for left half input of data

v. ki for 48 bit key from subkey generator

Outputs:-

i. lo is for the 32 bit left half of permuted data

ii. ro is for the 32 bit right half of permuted data

5.3.07 Ov32

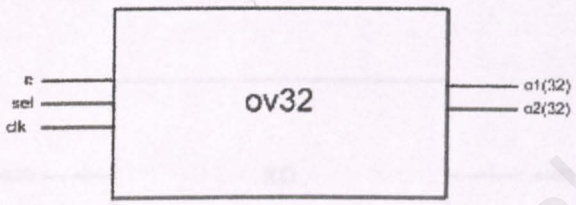


Figure 5.12 ov32 module

Inputs:-

i. e for bit permuted data input

ii. sel to select if permuted data is the last round, to end permutation round.

iii. Clk for clock input.

Outputs:-

i. o1 is for the 32 bit c-half of subkey

ii. o2 is for the 28 bit d-half of subkey

5.3.08 FP

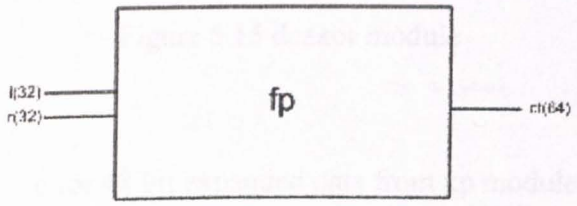


Figure 5.13 fp module

Inputs:-

i. li for 32 bit left half of permuted data

ii. ri for 32 bit right half of permuted data

Outputs:-

i. ct for 64 bit ciphertext output.

Initdata (a submodule of fullround)

5.3.09 XP

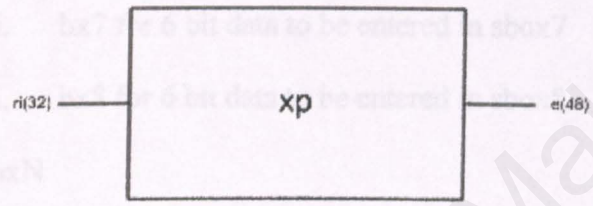


Figure 5.14 xp module

Inputs:-

i. ri for 32 bit right half to be expanded

Outputs:-

i. e is for the 48 bit expanded data

5.3.10 desxor1

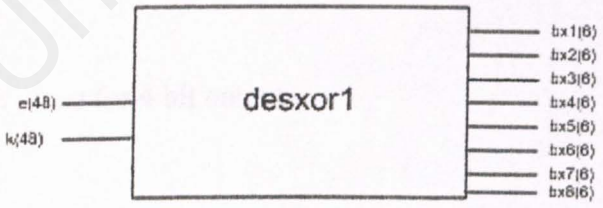


Figure 5.15 desxor module

Inputs:-

i. e for 48 bit expanded data from xp module

ii. ki for 48 bit subkey from subkey generator

Outputs:-

i. bx1 for 6 bit data to be entered in sbox1

ii. bx2 for 6 bit data to be entered in sbox2

iii. bx3 for 6 bit data to be entered in sbox3

iv. bx4 for 6 bit data to be entered in sbox4

v. bx5 for 6 bit data to be entered in sbox5

vi. bx6 for 6 bit data to be entered in sbox6

vii. bx7 for 6 bit data to be entered in sbox7

viii. bx8 for 6 bit data to be entered in sbox8

5.3.11 sboxN

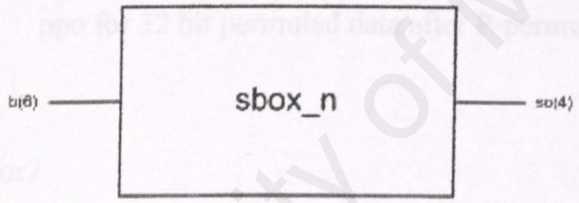


Figure 5.16 sbox module

Inputs:-

i. b for 6 bit data input.

Outputs:-

i. so for 4 bit output.

5.3.12 PP

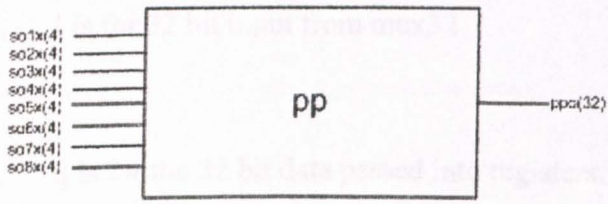


Figure 5.17 pp module

Inputs:-

i. so1x for 4 bit data from sbox1

ii. so2x for 4 bit data from sbox2

iii. so3x for 4 bit data from sbox3

iv. so4x for 4 bit data from sbox4

v. so5x for 4 bit data from sbox5

vi. so6x for 4 bit data from sbox6

vii. so7x for 4 bit data from sbox7

viii. so8x for 4 bit data from sbox8

Outputs:-

ii. ppo for 32 bit permuted data after P-permutated

5.3.13 desxor2

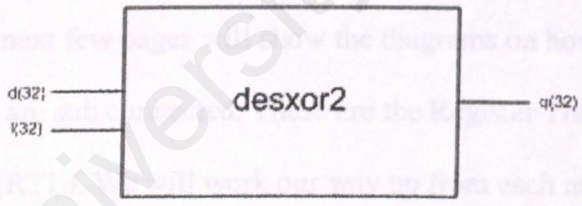


Figure 5.18 desxor2 module

Inputs:-

i. d is for 32 bit input from pp

ii. l is for 32 bit input from mux32

Outputs:-

i. q is for the 32 bit data passed into registers.

5.3.14 reg32

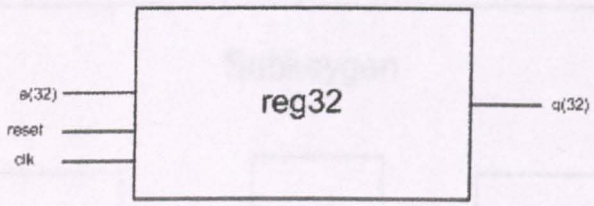


Figure 5.19 reg32 module

Inputs:-

i. a is for 32 bit input from desxor2

ii. reset is for reset

iii. clk is for clock input

Outputs:-

ii. q is for the 32 bit data passed into ov32.

5.4 How they are connected (RTL)

The next few pages will show the diagrams on how all functional submodules are sub connected. These are the Register Transfer Level description (RTL). We will work our way up from each main module, then at the DES (State) block module.

5.4.1 Subkeygen

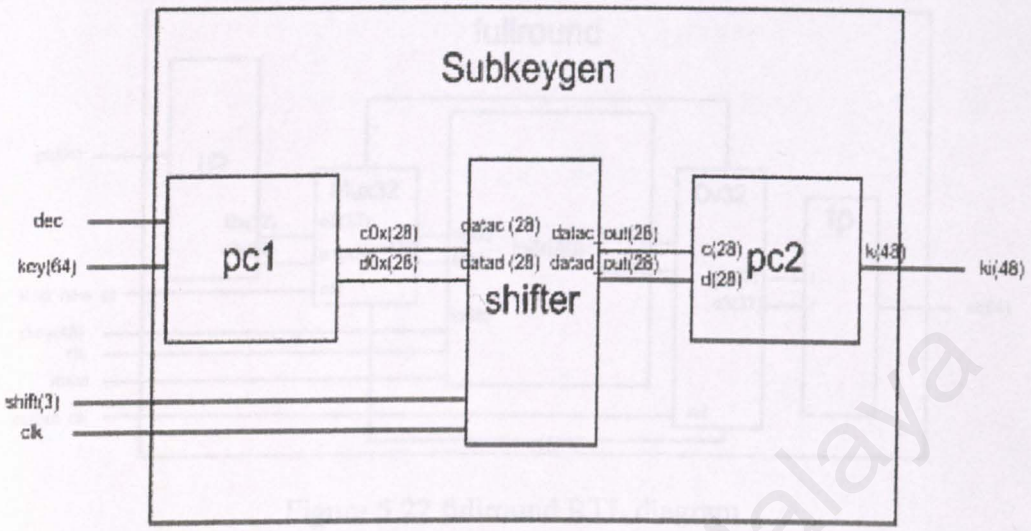


Figure 5.20 Subkeygen RTL diagram

5.4.2 Initdata

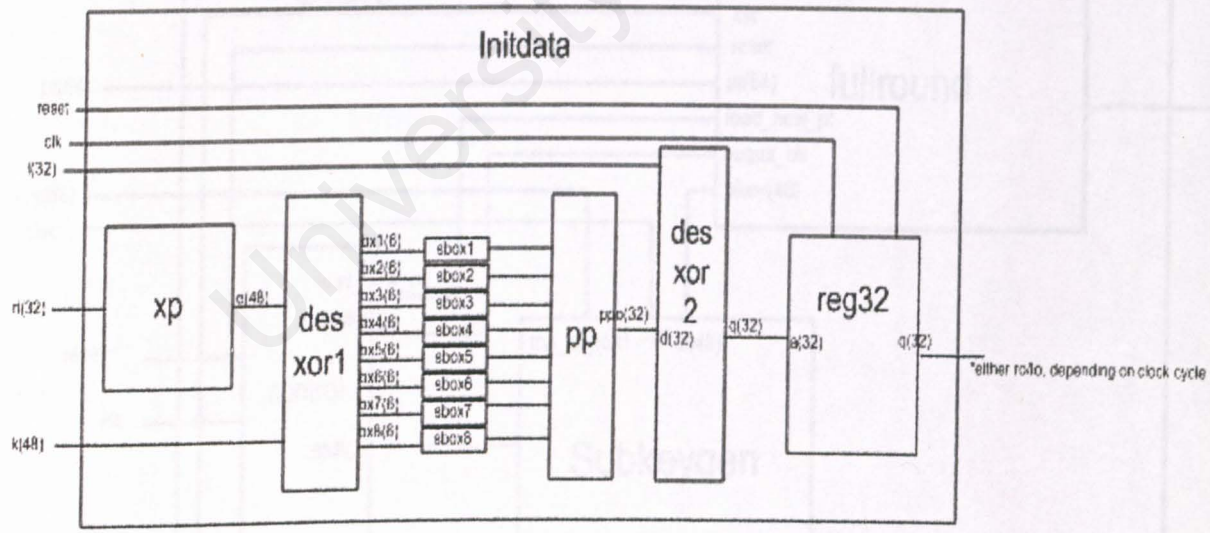


Figure 5.21 Initdata RTL diagram

5.4.3 Fullround

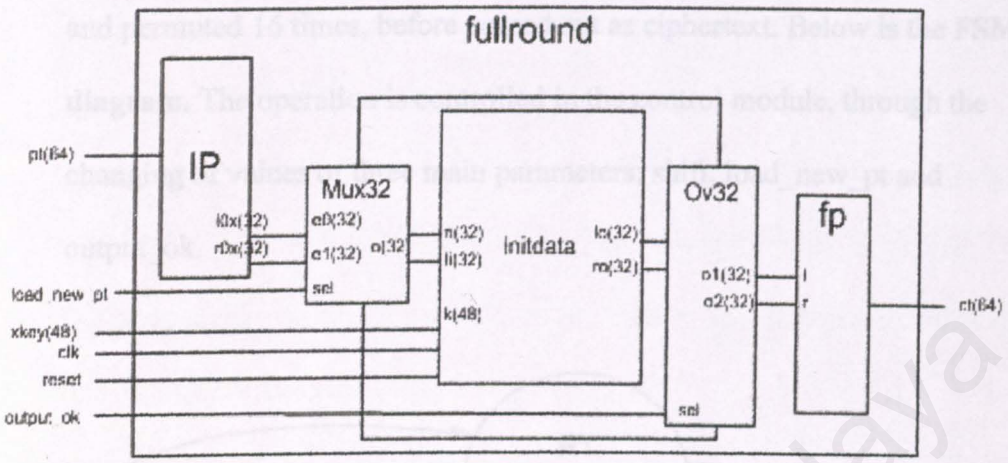


Figure 5.22 fullround RTL diagram

5.4.4 Top level (RTL)

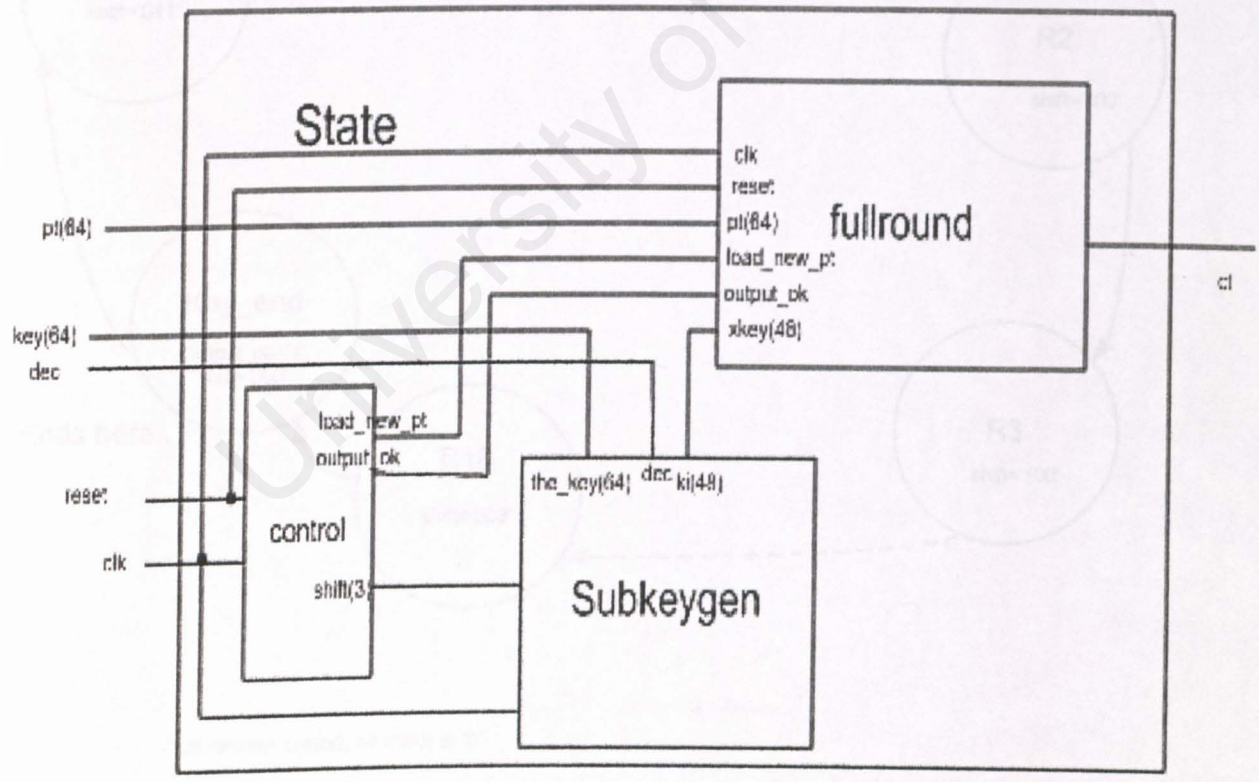


Figure 5.23 State module RTL diagram

5.5 Finite State Machine (FSM)

This part describes the basic operation of our DES design. Data is looped and permuted 16 times, before passed out as ciphertext. Below is the **FSM diagram.** The operation is controlled in the control module, through the changing of values of three main parameters; shift, load_new_pt and output_ok.

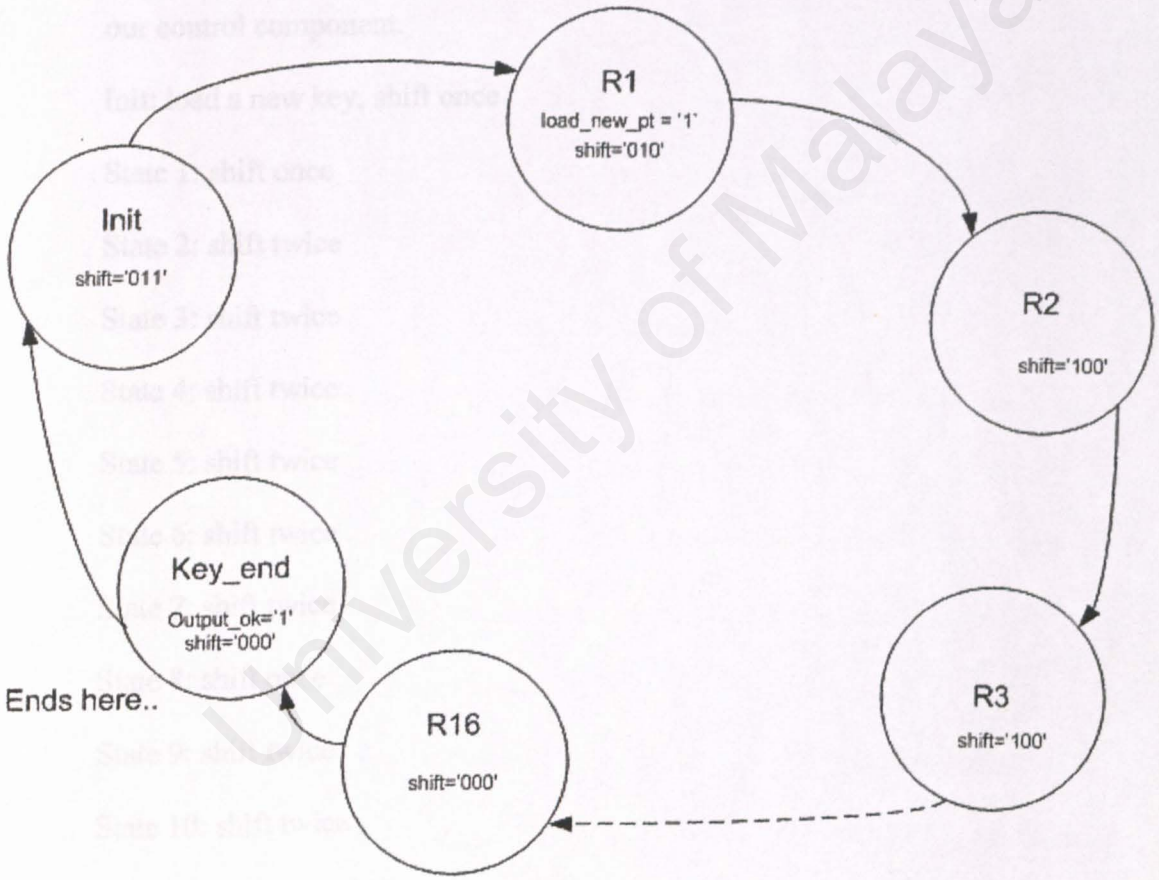


Figure 5.24 FSM Diagram

We can imagine that we have a state machine of 16 states (16 rounds of permutation), but in reality we will have 17 (because we need one state to load the key). Shift begins at init state, because the first round of permutation already needs a single bit shift for its subkeys. With this architecture, we have a throughput divided by 17 (one cipher every 17 clock cycles). We also add the key_end state to end the entry of subkeys into subkeygen. So overall, there are 18 states. Below describes the behavior of our control component.

Init: load a new key, shift once

State 1: shift once

State 2: shift twice

State 3: shift twice

State 4: shift twice

State 5: shift twice

State 6: shift twice

State 7: shift twice

State 8: shift once

State 9: shift twice

State 10: shift twice

State 11: shift twice

State 12: shift twice

State 13: shift twice

State 14: shift twice

State 15: shift once

State 16: none

Key_end: Ends key loading, no shift, give the output (ct).

All the related instructions to the state are executed the next state (future state). The state "key_end" is necessary, as the name says, to end loading of key to end rounds of permutation.

Below is a table displaying the shifter's decoder.

Value	action
000	No shift, no new key
010	Shift once, no new key
011	Shift once, new key
100	Shift twice, no new key
"others"	error=no shift

Table 5.25 shifter decoder

The least significant bit (bit located in the far right) indicates if a new key is needed, while the middle bit tells if we want to shift once (1=yes, 0=no) and finally the most significant bit tells to shift twice. Values like 111, 110 are impossible (there aren't states coded with 111 or 110). The signal shift will be decoded using a case statement in the VHDL.

The shift signal tells the machine to shift the 28 bits to left, either one or two bits, depending on each round. Below is a table of how each key are shifted.

Iteration #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Left Shifts	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Table 5.26 Left Shifts per iteration.

Chapter Six

Tools & Design Implementation

Chapter 6.0 Design Implementation

This chapter is divided into two. The first part discusses the tool used to develop our DES design. The tool of choice is peakFPGA. This will act as a simple user guide to our tool. The next part is about the implementation of our DES functions in VHDL.

6.1 peakFPGA

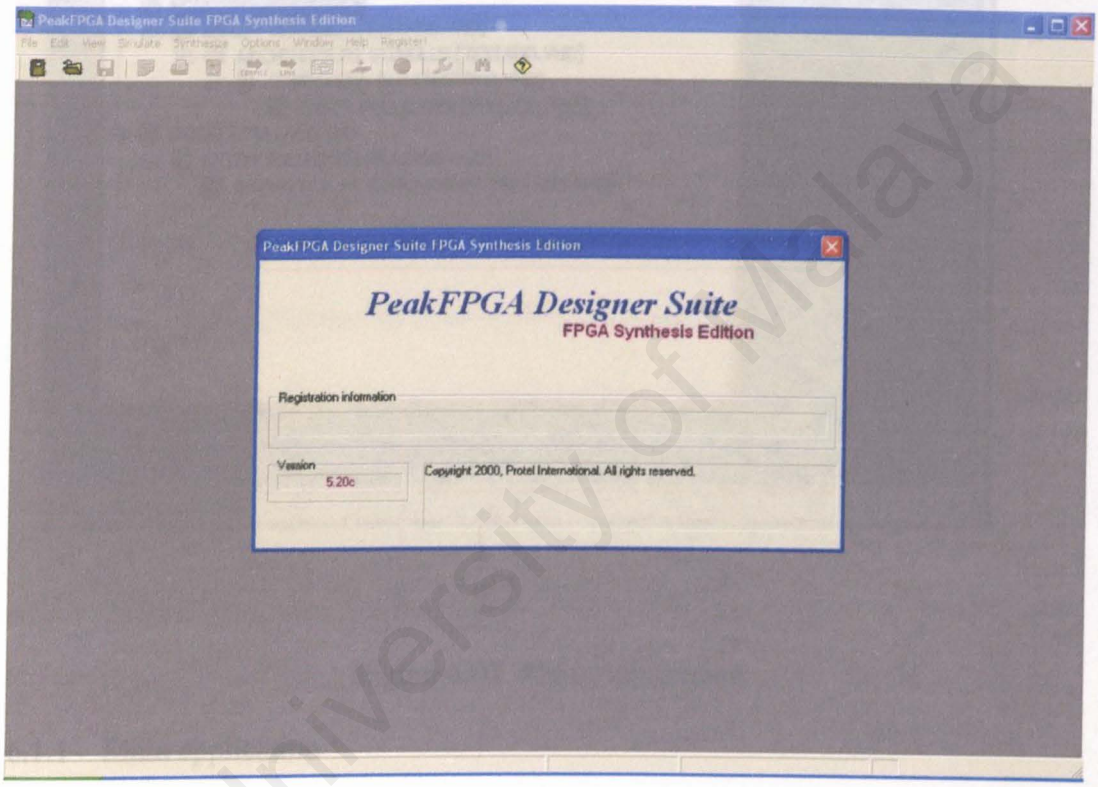


Figure 6.01 Entry Screen for peakFPGA

Our tool of choice is peakFPGA from Accolade. It is chosen because it is recommended by the faculty as one of the most user-friendly VHDL designer suite available. It is a complete package, allowing our VHDL design to be completed up to board level synthesizing process. Below is the definitive guide to the peakFPGA software. (taken from User's Manual Booklet)

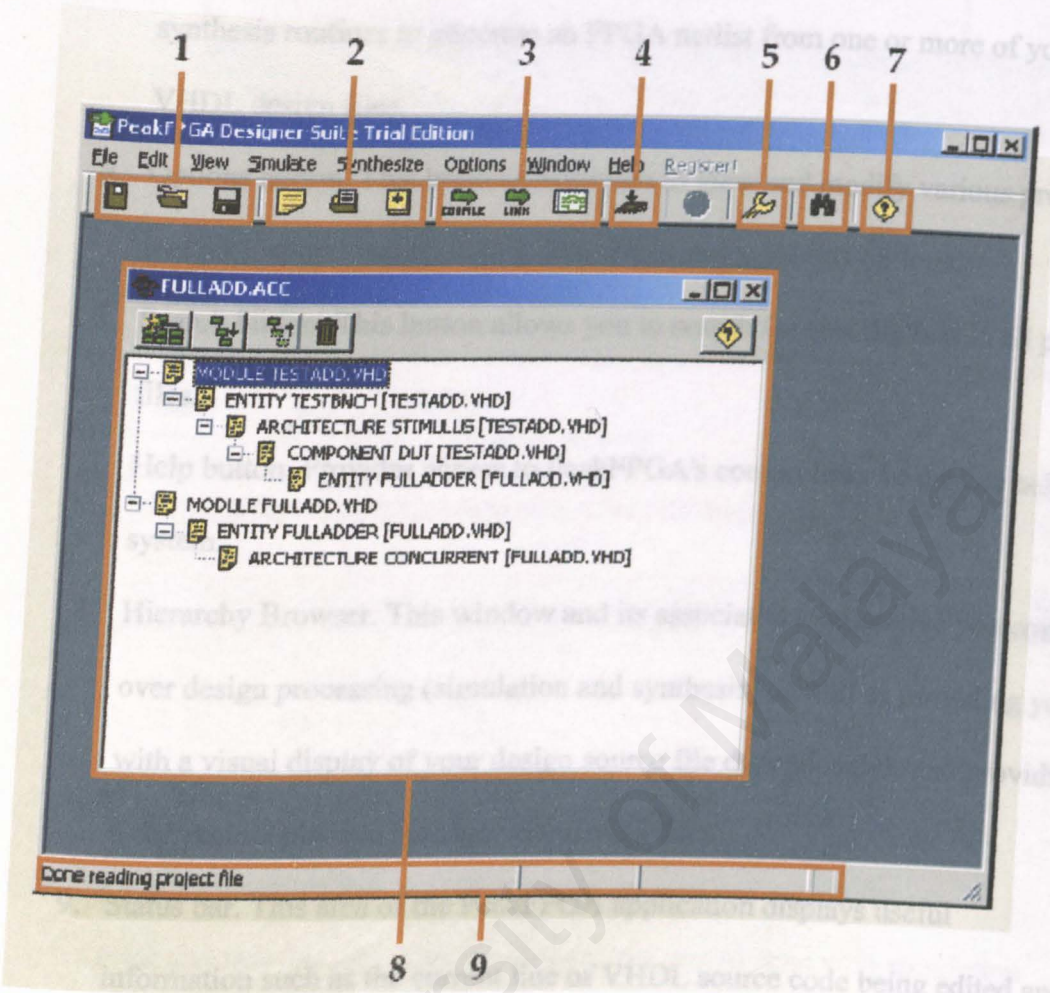


Figure 6.02 Main Application

6.1.1 Main application

1. Project file buttons. These buttons are used to create, open and save PeakFPGA projects.
2. Design management buttons. These buttons are used to create new VHDL design files, open files for viewing, and add existing VHDL design files to a project.
3. Simulation buttons. These buttons are used to compile, link and execute a selected part of your design (or the entire design) for simulation.

4. Synthesis button. This button is used to invoke PeakFPGA's powerful FPGA synthesis routines to generate an FPGA netlist from one or more of your VHDL design files.
5. Options button. This button allows you to view and modify various program and project options, including simulation and synthesis options.
6. Search button. This button allows you to search for specific text in all project files.
7. Help button. Provides access to PeakFPGA's comprehensive on-line help system.
8. Hierarchy Browser. This window and its associated toolbar give you control over design processing (simulation and synthesis) as well as providing you with a visual display of your design source file dependencies, and providing a convenient place to manage your design files.
9. Status bar. This area of the PeakFPGA application displays useful information such as the current line of VHDL source code being edited and displays a percent complete indicator that is active when certain processes are invoked.

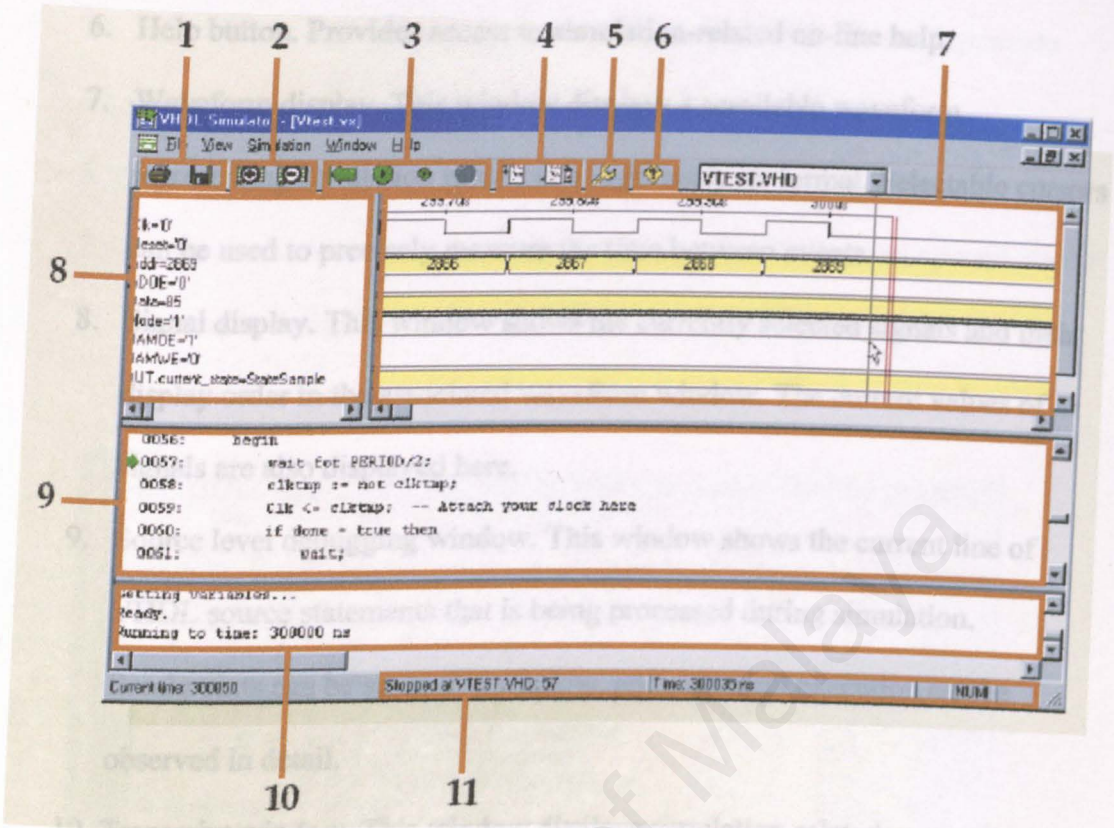


Figure 6.03 Simulator Application

6.1.2 Simulator application

1. Print and save buttons. These buttons allow you to print your simulation results (as waveforms) or to export them to a file.
2. Zoom in/out buttons. These buttons allow you to view all or part of your simulation waveform.
3. Simulation control buttons. These buttons are used to reset, start, step (by a predetermined amount of time) or stop the current simulation.
4. Source-level debug buttons. These buttons allow you to step through your design one executable line at a time for debugging purposes.
5. Options button. This button allows you to view and modify various simulation options, including waveform data formats and default time steps.

6. Help button. Provides access to simulation-related on-line help.
7. Waveform display. This window displays a scrollable waveform representing simulation results in a logic analyzer format. Selectable cursors can be used to precisely measure the time between events.
8. Signal display. This window shows the currently selected signals and their display order in the associated waveform window. The current values of signals are also displayed here.
9. Source level debugging window. This window shows the current line of VHDL source statements that is being processed during simulation. Breakpoints can be set in this window, and statement execution can be observed in detail.
10. Transcript window. This window displays simulation-related messages, as well as displaying any text I/O from your VHDL source code.
11. Status bar. This area of the simulator application displays useful information including a percent complete indicator that is active during simulation.

6.13 How do I create a new project in PeakFPGA Design Suite?

1. Select the PeakFPGA Design Suite icon in the Programs » PeakFPGA Design Suite folder of your Windows Start menu to start the application.
2. Select File » New Project.
3. Select File » Save Project As to name the project and select a project directory.
4. Select File » Add Module or File » Create Module to add existing VHDL design files to the project or create new VHDL files, respectively.

5. Select File » Rebuild Hierarchy to analyze the VHDL files and generate dependency information in the Hierarchy Browser.
6. Select File » Save Project to save the project.
7. When you have created a new project in PeakFPGA (or have opened one of the sample designs included with theproduct) you will see the VHDL files associated with that project listed in the Hierarchy Browser as shown here.

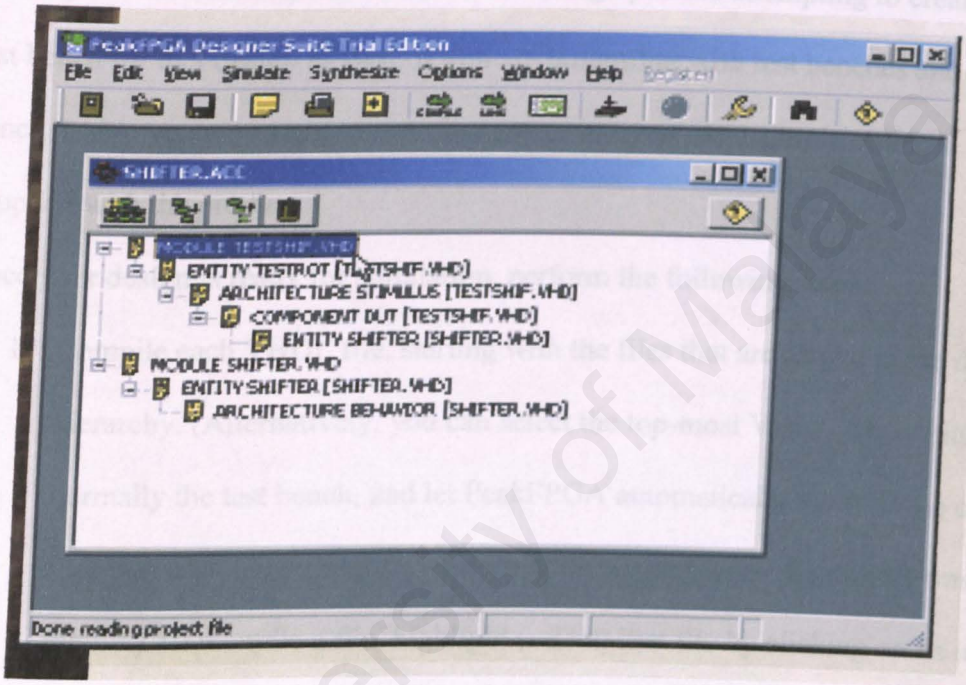


Figure 6.04 Hierarchy Browser

6.1.4 Hierarchy Browser toolbar

1. Rebuild Hierarchy - analyzes source files and updates hierarchy
2. Show Hierarchy - expands tree display to show all levels of hierarchy
3. Hide Hierarchy - collapses tree to display only the top level modules
4. Clean Up Project - deletes various intermediate and dynamically created files from the project directory

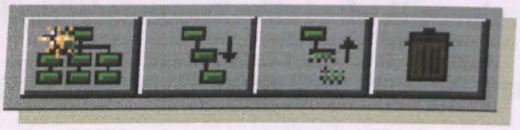


Figure 6.05 Hierarchy Browser toolbar

6.1.5 How do I simulate my VHDL project?

Before simulating your project you must first ensure that it is complete, including not only a VHDL description of the FPGA design you are attempting to create but a test bench for that design as well. If you are unfamiliar with test benches and test bench design you may want to examine one or more of the example projects supplied with the product.

Once your design is ready for simulation, perform the following steps:

1. Compile each VHDL file, starting with the files that are lowest in the design hierarchy. (Alternatively, you can select the top-most VHDL file, which is normally the test bench, and let PeakFPGA automatically compile the other files based on the dependency information created when the project was last rebuilt.) To compile a file, highlight (select) that file by clicking on its name once in the Hierarchy Brower, then select Compile from the Simulate menu, or click the Compile button in the main toolbar. Correct any VHDL errors (as indicated in the Compile Transcript window that appears) and recompile until all files have been successfully compiled.

Note: for some projects, depending on your design requirements, you may need to specify an alternate library (the default library is "work") in which to compile one or more VHDL modules. To specify an alternate library or set other

Compile options, highlight a specific file in the Hierarchy Browser, then open the Options dialog by choosing Options from the main menu. Enter the alternate library name in the Compile into Library text entry field.

2. After all files have been successfully compiled, select the top-most VHDL file in the design hierarchy (the test bench) and select Link from the Simulate menu, or click the Link button. Your compiled design files will be combined together to create a special kind of executable file called a Simulation Executable. Errors during the Link process (if any) will be reported to the Transcript window.
3. After the design has been successfully linked, click the Load Simulation button to invoke PeakFPGA Design Suite's integrated VHDL simulator.

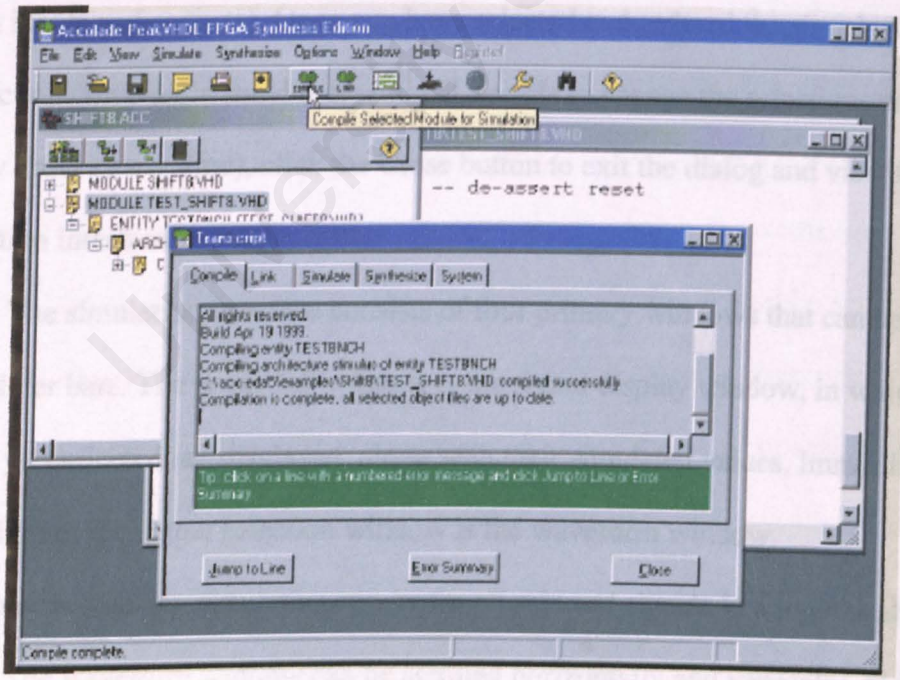


Figure 6.06 Compile Process

6.1.6 How do I use the VHDL simulator?

To simulate a VHDL design using PeakFPGA Design Suite, you must provide a VHDL test bench in addition to your synthesizable VHDL design description. Test bench design is beyond the scope of this tutorial, but you can examine a variety of sample test benches by opening some of the sample projects provided with the PeakFPGA product.

When you have successfully compiled, linked and loaded your design, including its test bench, the integrated VHDL Simulator is launched and a signal selection dialog appears. This dialog allows you to select the signals of greatest interest to you (for simulation purposes) and arrange them in a useful order for display. For your convenience, the Add Primaries button allows you to quickly add only those signals that were defined in the top-most file in your design (the primary design inputs and outputs). Once you have selected and ordered the signals to your satisfaction (keeping in mind that you will be able to change the selections and display order at any time), click the Close button to exit the dialog and view the simulation interface.

The simulation interface consists of four primary windows that can be sized using slider bars. The upper left window is the signal display window, in which the signals you selected are displayed, along with their simulated values. Immediately to the right of the signal selection window is the waveform window.
This window displays simulation results for all selected signals in a logic analyzer format. The waveform window can be scrolled horizontally and vertically, and zoom features allow you to get a close-up look at any portion of the waveform.

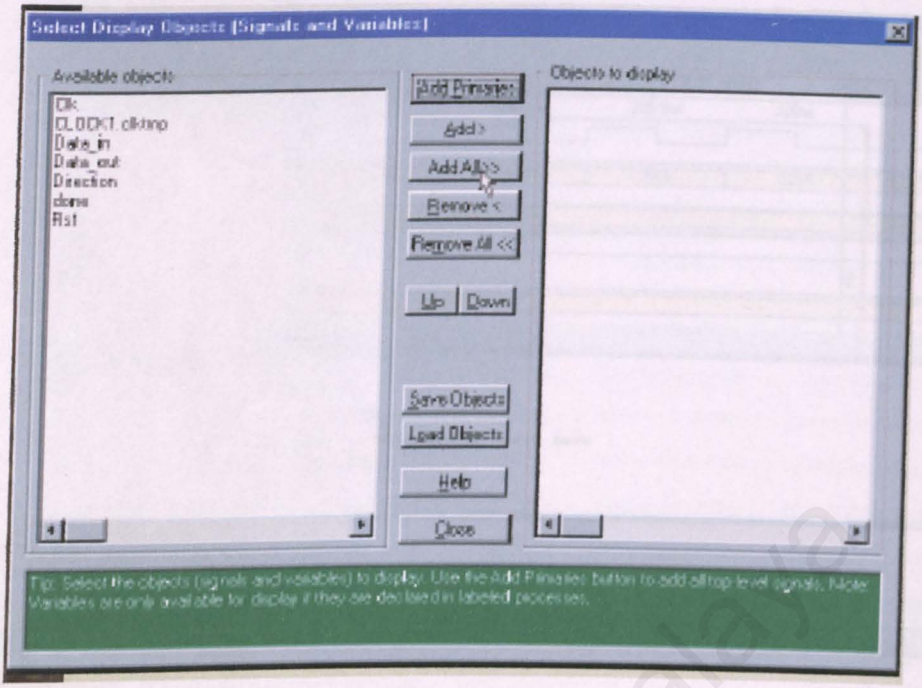


Figure 6.07 Signals selection

Measurement cursors can also be selected (by clicking with the mouse) to determine the exact amount of time between any two events. The source code display window, which is located directly below the waveform and signal display windows, provides you with a source-level view of the design being simulated. This window allows you to set breakpoints in your VHDL code and execute your design one line at a time to help in debugging. The window located below the source code display window is the transcript window. This window contains messages generated during simulation. Messages may be generated by the simulator to provide various types of status information, or may be generated from your VHDL code through the use of assertion statements or text I/O. To start simulation and view the results, click the Run To Time button (the large green VCR-style arrow). The simulator will execute your design to the end time that has been previously specified for your design, and a set of simulation waveforms will appear.

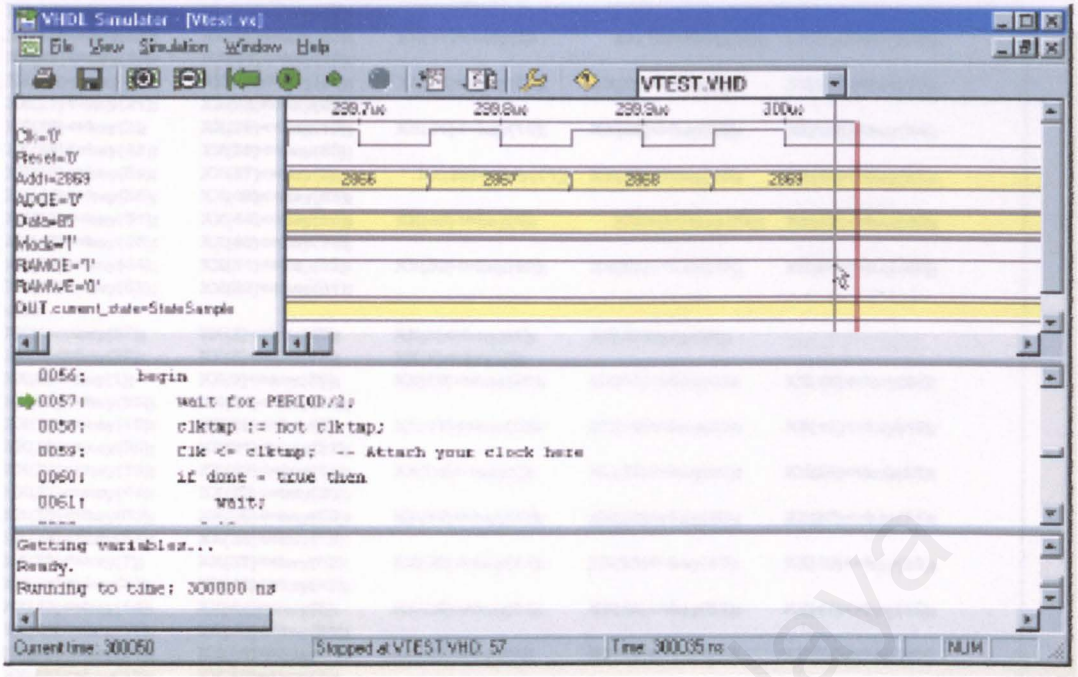


Figure 6.08 VHDL Simulator Interface.

Our design is not up to the synthesis stage because we do not have the needed tools for circuit level board (CLB) synthesizing process. Besides, our goal since the beginning of this project is just to design a simulatable code for DES in VHDL. Therefore, it is not relevant to go further in describing peakFPGA's next features (synthesis stage).

6.2 DES functions in VHDL

This part shows how all the DES functions are implemented in VHDL. It is not the entire source code for the module, but just the excerpt, the one that makes the function work.

6.2.1 PC1

```
architecture behavior of pc1 is
        signal XX  :        std_logic_vector(1 to 56);
begin
process(dec)
begin
        if dec = '1' then                --add dec value
        XX(1)<=key(8);      XX(2)<=key(16);     XX(3)<=key(24);     XX(4)<=key(32);
        XX(5)<=key(40);     XX(6)<=key(48);     XX(7)<=key(56);
        XX(8)<=key(64);     XX(9)<=key(7);      XX(10)<=key(15);    XX(11)<=key(23);    XX(12)<=key(31);
```

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

        c0x<=XX(1 to 28);   d0x<=XX(29 to 56);
```

6.2.2 Shifter

```
process(shift,clk)
begin

if (clk'event and clk = '1') then
case shift is

        when "001" =>
        -- no shift, new key
                datac_out_mem<=datac;
                datad_out_mem<=datad;

        when "010" =>
        -- shift once, no new key
                datac_out_mem<=To_StdLogicVector(to_bitvector(datac_out_mem) rol 1);
                datad_out_mem<=To_StdLogicVector(to_bitvector(datad_out_mem) rol 1);

        when "011" =>
        -- shift once, new key
                datac_out_mem<=To_StdLogicVector(to_bitvector(datac) rol 1);
                datad_out_mem<=To_StdLogicVector(to_bitvector(datad) rol 1);

        when "100" =>
        -- shift twice, no new key
                datac_out_mem<=To_StdLogicVector(to_bitvector(datac_out_mem) rol 2);
                datad_out_mem<=To_StdLogicVector(to_bitvector(datad_out_mem) rol 2);

        when "101" =>
        -- shift twice, new key
                datac_out_mem<=To_StdLogicVector(to_bitvector(datac) rol 2);
                datad_out_mem<=To_StdLogicVector(to_bitvector(datad) rol 2);


        when others =>
        -- error, no shift, no new key
```

6.2.3 PC2

```
architecture behavior of pc2 is
        signal YY :         std_logic_vector(1 to 56);
begin
        YY(1 to 28)<=c;                 YY(29 to 56)<=d;

        k(1)<=YY(14);       k(2)<=YY(17);       k(3)<=YY(11);       k(4)<=YY(24);       k(5)<=YY(1);
        k(6)<=YY(5);
        k(7)<=YY(3);        k(8)<=YY(28);       k(9)<=YY(15);       k(10)<=YY(6);       k(11)<=YY(21);
        k(12)<=YY(10);
        k(13)<=YY(23);      k(14)<=YY(19);      k(15)<=YY(12);      k(16)<=YY(4);       k(17)<=YY(26);
        k(18)<=YY(8);
        k(19)<=YY(16);      k(20)<=YY(7);       k(21)<=YY(27);      k(22)<=YY(20);      k(23)<=YY(13);
        k(24)<=YY(2);
        k(25)<=YY(41);      k(26)<=YY(52);      k(27)<=YY(31);      k(28)<=YY(37);      k(29)<=YY(47);
        k(30)<=YY(55);
        k(31)<=YY(30);      k(32)<=YY(40);      k(33)<=YY(51);      k(34)<=YY(45);      k(35)<=YY(33);
        k(36)<=YY(48);
        k(37)<=YY(44);      k(38)<=YY(49);      k(39)<=YY(39);      k(40)<=YY(56);      k(41)<=YY(34);
        k(42)<=YY(53);
        k(43)<=YY(46);      k(44)<=YY(42);      k(45)<=YY(50);      k(46)<=YY(36);      k(47)<=YY(29);
        k(48)<=YY(32);
end behavior;
```

6.2.4 IP

```
architecture behavior of ip is
begin  --l0x for left out, r0x for right out

l0x(1)<=pt(58);     l0x(2)<=pt(50);     l0x(3)<=pt(42);     l0x(4)<=pt(34);
l0x(5)<=pt(26);     l0x(6)<=pt(18);     l0x(7)<=pt(10);     l0x(8)<=pt(2);
l0x(9)<=pt(60);     l0x(10)<=pt(52);    l0x(11)<=pt(44);    l0x(12)<=pt(36);
l0x(13)<=pt(28);    l0x(14)<=pt(20);    l0x(15)<=pt(12);    l0x(16)<=pt(4);
l0x(17)<=pt(62);    l0x(18)<=pt(54);    l0x(19)<=pt(46);    l0x(20)<=pt(38);
l0x(21)<=pt(30);    l0x(22)<=pt(22);    l0x(23)<=pt(14);    l0x(24)<=pt(6);
l0x(25)<=pt(64);    l0x(26)<=pt(56);    l0x(27)<=pt(48);    l0x(28)<=pt(40);
l0x(29)<=pt(32);    l0x(30)<=pt(24);    l0x(31)<=pt(16);    l0x(32)<=pt(8);

r0x(1)<=pt(57);     r0x(2)<=pt(49);     r0x(3)<=pt(41);     r0x(4)<=pt(33);
r0x(5)<=pt(25);     r0x(6)<=pt(17);     r0x(7)<=pt(9);      r0x(8)<=pt(1);
r0x(9)<=pt(59);     r0x(10)<=pt(51);    r0x(11)<=pt(43);    r0x(12)<=pt(35);
r0x(13)<=pt(27);    r0x(14)<=pt(19);    r0x(15)<=pt(11);    r0x(16)<=pt(3);
r0x(17)<=pt(61);    r0x(18)<=pt(53);    r0x(19)<=pt(45);    r0x(20)<=pt(37);
r0x(21)<=pt(29);    r0x(22)<=pt(21);    r0x(23)<=pt(13);    r0x(24)<=pt(5);
r0x(25)<=pt(63);    r0x(26)<=pt(55);    r0x(27)<=pt(47);    r0x(28)<=pt(39);
r0x(29)<=pt(31);    r0x(30)<=pt(23);    r0x(31)<=pt(15);    r0x(32)<=pt(7);

end behavior;
```

6.2.5 FP

```
 architecture behaviour of fp is
begin

ct(1)<=r(8);        ct(2)<=l(8);ct(3)<=r(16);       ct(4)<=l(16);       ct(5)<=r(24);       ct(6)<=l(24);
        ct(7)<=r(32);       ct(8)<=l(32);
ct(9)<=r(7);        ct(10)<=l(7);       ct(11)<=r(15);      ct(12)<=l(15);      ct(13)<=r(23);      ct(14)<=l(23);
        ct(15)<=r(31);      ct(16)<=l(31);
ct(17)<=r(6);       ct(18)<=l(6);       ct(19)<=r(14);      ct(20)<=l(14);      ct(21)<=r(22);      ct(22)<=l(22);
        ct(23)<=r(30);      ct(24)<=l(30);
ct(25)<=r(5);       ct(26)<=l(5);       ct(27)<=r(13);      ct(28)<=l(13);      ct(29)<=r(21);      ct(30)<=l(21);
        ct(31)<=r(29);      ct(32)<=l(29);
ct(33)<=r(4);       ct(34)<=l(4);       ct(35)<=r(12);      ct(36)<=l(12);      ct(37)<=r(20);      ct(38)<=l(20);
        ct(39)<=r(28);      ct(40)<=l(28);
ct(41)<=r(3);       ct(42)<=l(3);       ct(43)<=r(11);      ct(44)<=l(11);      ct(45)<=r(19);      ct(46)<=l(19);
        ct(47)<=r(27);      ct(48)<=l(27);
```

```
ct(49)<=r(2);   ct(50)<=l(2);   ct(51)<=r(10);   ct(52)<=l(10);   ct(53)<=r(18);   ct(54)<=l(18);
        ct(55)<=r(26);   ct(56)<=l(26);
ct(57)<=r(1);   ct(58)<=l(1);   ct(59)<=r(9);   ct(60)<=l(9);   ct(61)<=r(17);   ct(62)<=l(17);
        ct(63)<=r(25);   ct(64)<=l(25);

end;
```

6.2.6 XP

```
architecture behavior of xp is

begin

        e(1)<=ri(32);   e(2)<=ri(1);e(3)<=ri(2);e(4)<=ri(3);e(5)<=ri(4);e(6)<=ri(5);e(7)<=ri(4);e(8)<=ri(5);

        e(9)<=ri(6);e(10)<=ri(7);   e(11)<=ri(8);   e(12)<=ri(9);   e(13)<=ri(8);   e(14)<=ri(9);

        e(15)<=ri(10);   e(16)<=ri(11);

        e(17)<=ri(12);   e(18)<=ri(13);   e(19)<=ri(12);   e(20)<=ri(13);   e(21)<=ri(14);

        e(22)<=ri(15);   e(23)<=ri(16);   e(24)<=ri(17);

        e(25)<=ri(16);   e(26)<=ri(17);   e(27)<=ri(18);   e(28)<=ri(19);   e(29)<=ri(20);

        e(30)<=ri(21);   e(31)<=ri(20);   e(32)<=ri(21);

        e(33)<=ri(22);   e(34)<=ri(23);   e(35)<=ri(24);   e(36)<=ri(25);   e(37)<=ri(24);

        e(38)<=ri(25);   e(39)<=ri(26);   e(40)<=ri(27);

        e(41)<=ri(28);   e(42)<=ri(29);   e(43)<=ri(28);   e(44)<=ri(29);   e(45)<=ri(30);

        e(46)<=ri(31);   e(47)<=ri(32);   e(48)<=ri(1);

end behavior;
```

6.2.7 Sbox

```
process(b)
begin
case b is
        when "000000"=>   so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "000010"=>   so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "000100"=>   so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "000110"=>   so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "001000"=>   so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "001010"=>   so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "001100"=>   so<=To_StdLogicVector(Bit_Vector'(x"b"));
        when "001110"=>   so<=To_StdLogicVector(Bit_Vector'(x"8"));
        when "010000"=>   so<=To_StdLogicVector(Bit_Vector'(x"3"));
        when "010010"=>   so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "010100"=>   so<=To_StdLogicVector(Bit_Vector'(x"6"));
        when "010110"=>   so<=To_StdLogicVector(Bit_Vector'(x"c"));
        when "011000"=>   so<=To_StdLogicVector(Bit_Vector'(x"5"));
        when "011010"=>   so<=To_StdLogicVector(Bit_Vector'(x"9"));
        when "011100"=>   so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "011110"=>   so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "000001"=>   so<=To_StdLogicVector(Bit_Vector'(x"0"));
        when "000011"=>   so<=To_StdLogicVector(Bit_Vector'(x"f"));
        when "000101"=>   so<=To_StdLogicVector(Bit_Vector'(x"7"));
        when "000111"=>   so<=To_StdLogicVector(Bit_Vector'(x"4"));
        when "001001"=>   so<=To_StdLogicVector(Bit_Vector'(x"e"));
        when "001011"=>   so<=To_StdLogicVector(Bit_Vector'(x"2"));
        when "001101"=>   so<=To_StdLogicVector(Bit_Vector'(x"d"));
        when "001111"=>   so<=To_StdLogicVector(Bit_Vector'(x"1"));
        when "010001"=>   so<=To_StdLogicVector(Bit_Vector'(x"a"));
        when "010011"=>   so<=To_StdLogicVector(Bit_Vector'(x"6"));
```

```
            when "010101"=>     so<=To_StdLogicVector(Bit_Vector'(x"c"));
            when "010111"=>     so<=To_StdLogicVector(Bit_Vector'(x"b"));
            when "011001"=>     so<=To_StdLogicVector(Bit_Vector'(x"9"));
            when "011011"=>     so<=To_StdLogicVector(Bit_Vector'(x"5"));
            when "011101"=>     so<=To_StdLogicVector(Bit_Vector'(x"3"));
            when "011111"=>     so<=To_StdLogicVector(Bit_Vector'(x"6"));
            when "100000"=>     so<=To_StdLogicVector(Bit_Vector'(x"4"));
            when "100010"=>     so<=To_StdLogicVector(Bit_Vector'(x"1"));
            when "100100"=>     so<=To_StdLogicVector(Bit_Vector'(x"e"));
            when "100110"=>     so<=To_StdLogicVector(Bit_Vector'(x"8"));
            when "101000"=>     so<=To_StdLogicVector(Bit_Vector'(x"d"));
            when "101010"=>     so<=To_StdLogicVector(Bit_Vector'(x"6"));
            when "101100"=>     so<=To_StdLogicVector(Bit_Vector'(x"2"));
            when "101110"=>     so<=To_StdLogicVector(Bit_Vector'(x"b"));
            when "110000"=>     so<=To_StdLogicVector(Bit_Vector'(x"f"));
            when "110010"=>     so<=To_StdLogicVector(Bit_Vector'(x"c"));
            when "110100"=>     so<=To_StdLogicVector(Bit_Vector'(x"9"));
            when "110110"=>     so<=To_StdLogicVector(Bit_Vector'(x"7"));
            when "111000"=>     so<=To_StdLogicVector(Bit_Vector'(x"3"));
            when "111010"=>     so<=To_StdLogicVector(Bit_Vector'(x"a"));
            when "111100"=>     so<=To_StdLogicVector(Bit_Vector'(x"5"));
            when "111110"=>     so<=To_StdLogicVector(Bit_Vector'(x"0"));
            when "100001"=>     so<=To_StdLogicVector(Bit_Vector'(x"f"));
            when "100011"=>     so<=To_StdLogicVector(Bit_Vector'(x"c"));
            when "100101"=>     so<=To_StdLogicVector(Bit_Vector'(x"8"));
            when "100111"=>     so<=To_StdLogicVector(Bit_Vector'(x"2"));
            when "101001"=>     so<=To_StdLogicVector(Bit_Vector'(x"4"));
            when "101011"=>     so<=To_StdLogicVector(Bit_Vector'(x"9"));
            when "101101"=>     so<=To_StdLogicVector(Bit_Vector'(x"1"));
            when "101111"=>     so<=To_StdLogicVector(Bit_Vector'(x"7"));
            when "110001"=>     so<=To_StdLogicVector(Bit_Vector'(x"5"));
            when "110011"=>     so<=To_StdLogicVector(Bit_Vector'(x"b"));
            when "110101"=>     so<=To_StdLogicVector(Bit_Vector'(x"3"));
            when "110111"=>     so<=To_StdLogicVector(Bit_Vector'(x"e"));
            when "111001"=>     so<=To_StdLogicVector(Bit_Vector'(x"a"));
            when "111011"=>     so<=To_StdLogicVector(Bit_Vector'(x"0"));
            when "111101"=>     so<=To_StdLogicVector(Bit_Vector'(x"6"));
            when others=>               so<=To_StdLogicVector(Bit_Vector'(x"d"));
end case;
end process;
```

6.2.8 PP

```
architecture behaviour of pp is
        signal XX  :        std_logic_vector(1 to 32);
begin
        XX(1 to 4)<=so1x;   XX(5 to 8)<=so2x;   XX(9 to 12)<=so3x;   XX(13 to 16)<=so4x;
        XX(17 to 20)<=so5x; XX(21 to 24)<=so6x; XX(25 to 28)<=so7x; XX(29 to 32)<=so8x;

        ppo(1)<=XX(16);             ppo(2)<=XX(7);              ppo(3)<=XX(20);
        ppo(4)<=XX(21);
        ppo(5)<=XX(29);             ppo(6)<=XX(12);             ppo(7)<=XX(28);
        ppo(8)<=XX(17);
        ppo(9)<=XX(1);              ppo(10)<=XX(15);    ppo(11)<=XX(23);    ppo(12)<=XX(26);
        ppo(13)<=XX(5);             ppo(14)<=XX(18);    ppo(15)<=XX(31);    ppo(16)<=XX(10);
        ppo(17)<=XX(2);             ppo(18)<=XX(8);             ppo(19)<=XX(24);    ppo(20)<=XX(14);
        ppo(21)<=XX(32);    ppo(22)<=XX(27);    ppo(23)<=XX(3);             ppo(24)<=XX(9);
        ppo(25)<=XX(19);    ppo(26)<=XX(13);    ppo(27)<=XX(30);    ppo(28)<=XX(6);
        ppo(29)<=XX(22);    ppo(30)<=XX(11);    ppo(31)<=XX(4);             ppo(32)<=XX(25);
end;
```

6.2.9 Non functional requirements description

- Mux32

```
process(sel,e0,e1)
begin
if sel = '0' then
        o <= e0;
else
        o <= e1;
end if;
end process;
```

- Reg32

```
architecture synth of reg32 is
signal memory : std_logic_vector (1 to 32) ;
begin
process(clk,reset)
begin
        if(reset = '1') then
                memory <= (others => '0');
        else
                if(clk = '1' and clk'event) then
                        memory <= a;
                end if;
        end if;
end process;
q <= memory;
end synth;
```

- Ov32

```
ARCHITECTURE synth OF ov32 IS
BEGIN
process(sel,clk)
begin
if (clk'event and clk = '1') then
if(sel = '1') then
        o2<=e;
end if;
end if;
end process;
o1<=e;
END synth;
```

The complete source codes are in the diskette enclosed.

Chapter Seven

DES Verification Manual Example

Chapter 7.0 DES results verification

In order to verify our DES output, we must have a sample data and key flowed through all the DES functions. This chapter shows each step of permutation with the changes to the data at each step.

7.1 Generating 16 subkeys through pc1, shifter and pc2 (functions from subkeygen)

The suitable key selected is $FF00FF00FF00FF00_H$. The reason why will be explained later.

After pc1. The y-axis shows the first numbers after a gap of eight bits. The eighth bits are stripped (bits on shaded area) because we only need 56 bits.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	1	1	1	1	1	1	1	1
9	0	0	0	0	0	0	0	0
17	1	1	1	1	1	1	1	1
25	0	0	0	0	0	0	0	0
33	1	1	1	1	1	1	1	1
41	0	0	0	0	0	0	0	0
49	1	1	1	1	1	1	1	1
57	0	0	0	0	0	0	0	0

Table 7.01 After pc1

Split into two halves, c and d.

[c]

1 to 7	0	1	0	1	0	1	0
8 to 14	1	0	1	0	1	0	1
15 to 21	0	1	0	1	0	1	0
22 to 28	1	0	1	0	1	0	1

[d]

1 to 7	0	1	0	1	0	1	0
8 to 14	1	0	1	0	1	0	1
15 to 21	0	1	0	1	0	1	0
22 to 28	1	0	1	0	1	0	1

Table 7.02 After Split

This is the reason why this key is selected. As you can see, the two halves are identical, so we only have to work once.

Shift the bits. This stage, we will shift the bits leftward, according to table 5.26. The most significant bits are then shifted to the right.

Pseudocode :- **i = 0, i ++, i <17**

i = 0 (initial stage)

c :- 0101 0101 0101 0101 0101 0101 0101

d:- 0101 0101 0101 0101 0101 0101 0101

i = 1 (first round of permutation)

c :- 1010 1010 1010 1010 1010 1010 1010

d:- 1010 1010 1010 1010 1010 1010 1010

i = 2

c :- 0101 0101 0101 0101 0101 0101 0101

d:- 0101 0101 0101 0101 0101 0101 0101

i = 3

c :- 0101 0101 0101 0101 0101 0101 0101

d:- 0101 0101 0101 0101 0101 0101 0101

i = 4

c :- 0101 0101 0101 0101 0101 0101 0101

d:- 0101 0101 0101 0101 0101 0101 0101

i = 5

c :- 0101 0101 0101 0101 0101 0101 0101

d:- 0101 0101 0101 0101 0101 0101 0101

i = 6

c :- 0101 0101 0101 0101 0101 0101 0101

d:- 0101 0101 0101 0101 0101 0101 0101

i = 7

c :- 0101 0101 0101 0101 0101 0101 0101

d:- 0101 0101 0101 0101 0101 0101 0101

i = 8

c :- 0101 0101 0101 0101 0101 0101 0101

d:- 0101 0101 0101 0101 0101 0101 0101

i = 9

c :- 1010 1010 1010 1010 1010 1010 1010

d:- 1010 1010 1010 1010 1010 1010 1010

i = 10

c :- 1010 1010 1010 1010 1010 1010 1010

d:- 1010 1010 1010 1010 1010 1010 1010

i = 11

c :- 1010 1010 1010 1010 1010 1010 1010

d:- 1010 1010 1010 1010 1010 1010 1010

i = 12

c :- 1010 1010 1010 1010 1010 1010 1010

d:- 1010 1010 1010 1010 1010 1010 1010

i = 13

c :- 1010 1010 1010 1010 1010 1010 1010

d:- 1010 1010 1010 1010 1010 1010 1010

i = 14

c :- 1010 1010 1010 1010 1010 1010 1010

d:- 1010 1010 1010 1010 1010 1010 1010

i = 15

c :- 1010 1010 1010 1010 1010 1010 1010

d:- 1010 1010 1010 1010 1010 1010 1010

i = 16

c :- 0101 0101 0101 0101 0101 0101 0101

d:- 0101 0101 0101 0101 0101 0101 0101

After pc2.After 16 rounds of iteration, we could see that actually, there are only two permutations. Values after permutation 1, 9, 10, 11, 12, 13, 14 and 15 are the same, while the same also goes to values after permutation 2, 3, 4, 5, 6, 7, 8 and 16. We send this value to pc2, and this is what we get.

i_a = 1, 9, 10, 11, 12, 13, 14 and 15	i_b = 2, 3, 4, 5, 6, 7, 8, 16
[0110][11	[1001] [00
10] [1010]	01] [0101]
1100] [00	[0011] [11
01] [1010]	10] [0101]
[1011] [11	[0100] [00
00] [1110]	11] [0001]
[0110] [01	[1001] [10
00] [0010]	11] [1101]

For all the values in the bracket, it is converted into hexadecimal. Below are the processed keys.

Keys i_a = 6EAC1ABCE642

Keys i_b =9153E98319BD

7.2 16 rounds of data permutation

Now, we will pass the keys into the DES core. The data will be passed through functions ip, function *f* then finally, fp. The sample data selected is $00000000FFFFFFFF_H$.

After IP. After passing through the IP function, the values are split into two halves, R for top half, and L for lower half.

	1	2	3	4	5	6	7	8
1 to 8	1	1	1	1	0	0	0	0
9 to 16	1	1	1	1	0	0	0	0
17 to 24	1	1	1	1	0	0	0	0
25 to 32	1	1	1	1	0	0	0	0
33 to 40	1	1	1	1	0	0	0	0
41 to 48	1	1	1	1	0	0	0	0
49 to 56	1	1	1	1	0	0	0	0
57 to 64	1	1	1	1	0	0	0	0

Table 7.03 After IP

After XP. Values are entered into f functions, and the first function is the Expansion (xp) function.

	1	2	3	4	5	6
1 to 6	0	1	1	1	1	0
7 to 12	1	0	0	0	0	1
13 to 18	0	1	1	1	1	0
19 to 24	1	0	0	0	0	1
25 to 30	0	1	1	1	1	0
31 to 36	1	0	0	0	0	1
37 to 42	0	1	1	1	1	0
43 to 48	1	0	0	0	0	1

Table 7.04 After XP

XOR with key. Values passed from XP into the desxor1 function, XOR with subkey R1.

After xp	Subkey After R1	Product of xp XOR r1
011110	011011	000101
100001	101010	001011
011110	110000	101110
100001	011010	111011
011110	101111	110001
100001	001110	101111
011110	011001	000111
100001	000010	100011

Table 7.05 After desxor1

Sbox substitutions. The XOR product of xp and r1 are then split into eight equal adjoining parts, and entered into sboxes. (Refer to 2.4.4 for sbox guide) Below are the values obtained. For coordinate, it's in (x,y) form.

	sbox	Bit value	Coordinate	Value	In binary
1 to 6	1	0 0010 1	1,2	7	0111
7 to 12	2	1 0101 1	1,5	2	0010
13 to 18	3	1 0111 0	2,7	0	0000
19 to 24	4	1 1101 1	3,13	7	0111
25 to 30	5	1 1000 1	3,8	6	0110
31 to 36	6	1 0111 1	3,7	10	1010
37 to 42	7	0 0011 1	1,3	7	0111
43 to 48	8	1 0001 1	3,1	1	0001

Table 7.06 After sbox

P Permutation. After the sbox substitutions, the values are once again permuted according to P permutation. Here is the result.

After P
1101
0010
0111
0100
1001
1110
1000
0010

Table 7.07 After P

XOR2. This is the final function before it is passed through as a complete round (becoming L_{i+1} , remember, $R_i = L_{i+1}$?)

P values	R1 keys	Product	Value (H)
1101	1111	0010	2
0010	0000	0010	2
0111	1111	1000	8
0100	0000	0100	4
1001	1111	0110	6
1110	0000	1110	E
1000	1111	0111	7
0010	0000	0010	2

Table 7.08 End of R1

These steps are repeated another 15 times for 16 rounds of permutation. Due to space constraints, I will only display the final values after each round of permutation, before sending the values into FP for our final ciphertext.

R1 = 22846E72 H

R2 = 65C800B9 H

R3 = E1AD5D5B H

R4 = 7F86D9C7 H

R5 = CBA86EAB H

R6 = E4968AE9 H

R7 = DE8F8B35 H

R8 = 9D4C8B41 H

R9 = CD2242FE H

R10 = 251B5698 H

R11 = 4DC0735E H

R12 = 4EA2005D H

R13 = F41FEB2F H

R14 = 70074950 H

R15 = 9A850263 H

R16 = 57816792 H

R17 = D8645168 H

The two values to enter FP are R15 and R16. R15 will become the left half, R16 is the right half. R17 is just permuted data that is needed to end the key, and it is discarded.

After FP. Below is the final ciphertext output of DES for our sample data and key. If the design's output does not tally with what we get here, then the design is considered a failure.

	Values				hexadecimal
1 to 16	0111	0110	1010	1111	76CF
17 to 32	0110	0100	1001	0000	6480
33 to 48	1010	0001	0000	0110	C106
49 to 64	0100	0110	1011	0001	46B1

Table 7.09 After fp

Ciphertext = 76CF6480C10646B1 $_{H}$

* Refer to chapter 2 on permutation arrangements of data.

Chapter Eight

DES Design Verification

Chapter 8.0 DES Design Testing / Verification

This chapter shows if our design reaches its objective. A sample data is entered along with its key, and then we look at its product. If the design is able to decrypt our ciphertext, then it is considered successful. Sample data is the same as the one example that we worked in chapter 7.

8.1 Compile All Design

All finished designs are compiled at the top-most level of design. In our case, the State_TB module. This is a testbench module, and it is autogenerated from an online automatic vhdl testbench generator. The website is available in the reference.

Before that, here are the values entered into the testbench.

Pt = x"00000000FFFFFFFF";

Key = x"FF00FF00FF00FF00";

Dec ='0';

Reset ='0';

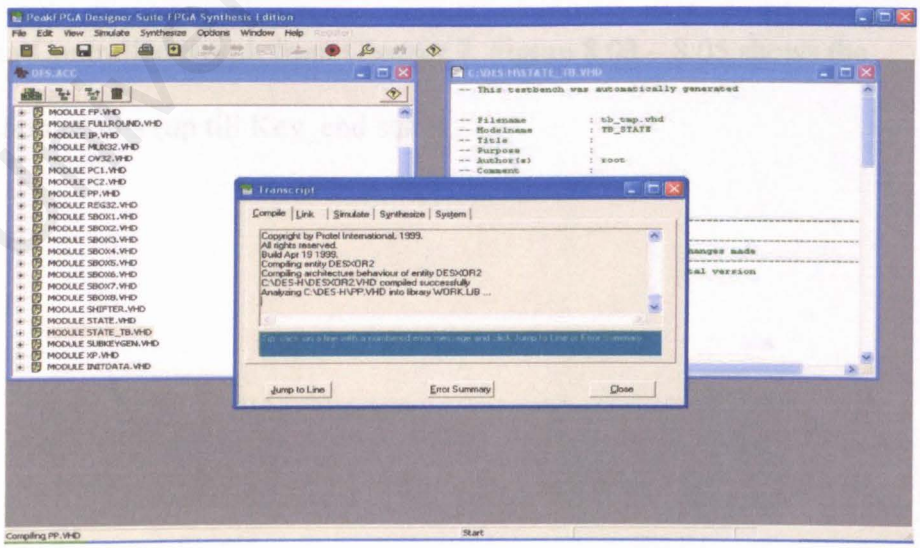


Figure 8.01 Compiling state_tb.vhd

8.2 Simulate State_TB

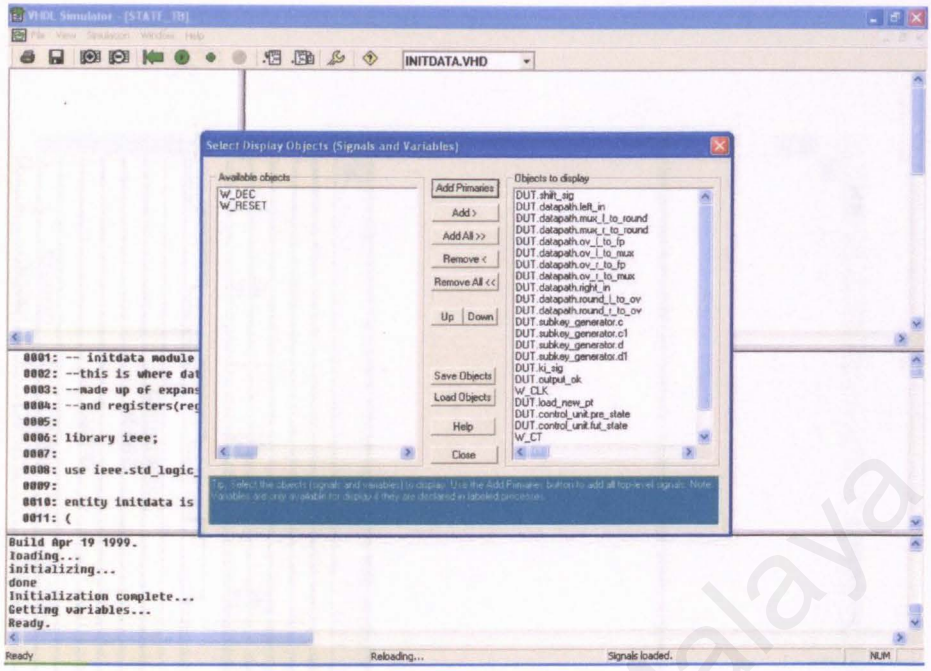


Figure 8.02 Select simulation signals

For simulation signals, all signals selected except for reset and dec.

8.3 Waveform Analysis

The most important waveforms to be analyzed are the keys and the permuted data from each round. These values are checked to see if it tallies with our Controlled Result from Chapter 7. Figure 8.03 – 8.05 shows the simulation results (up till Key_end state)

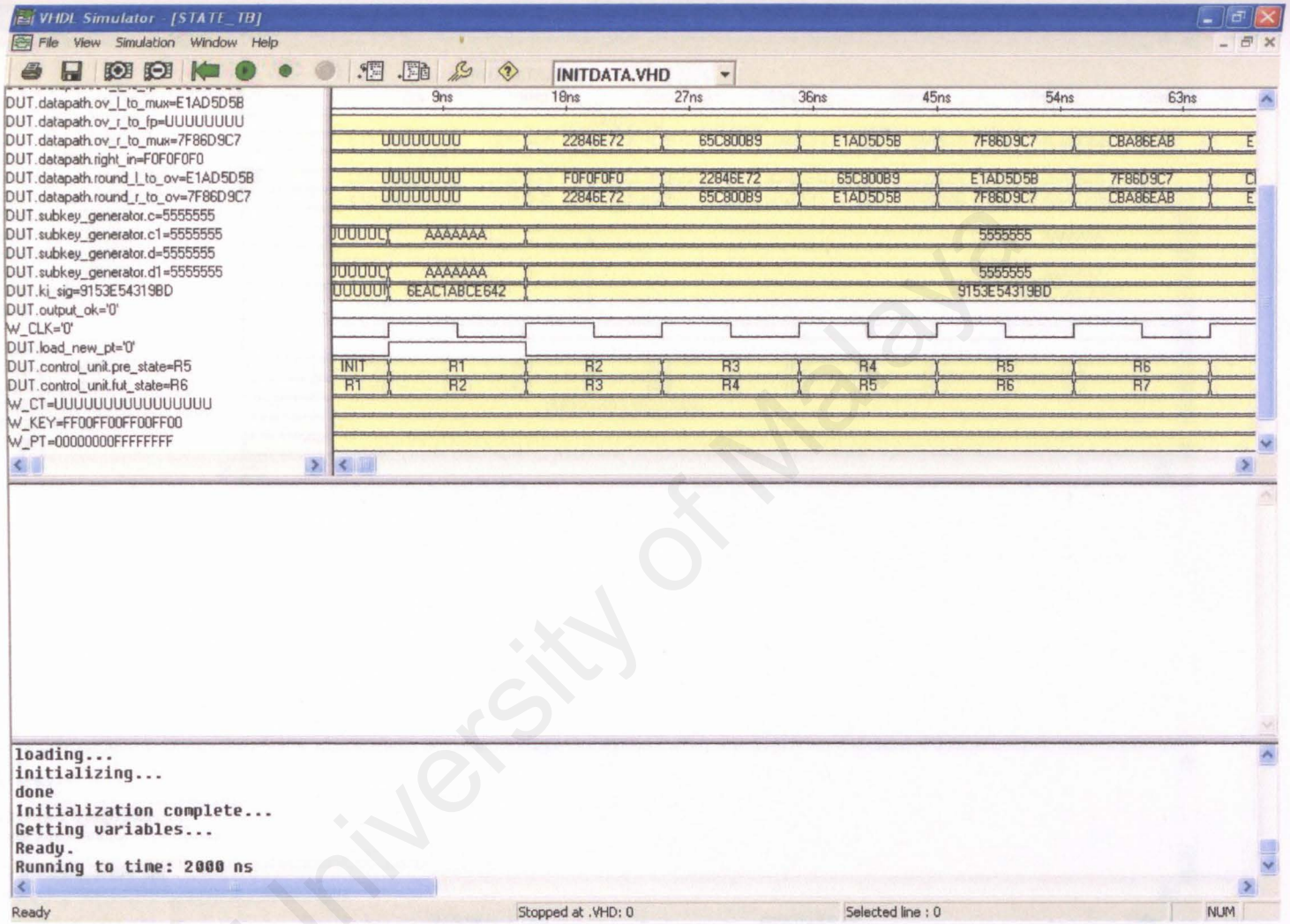


Figure 8.03 Results from state Init – R6

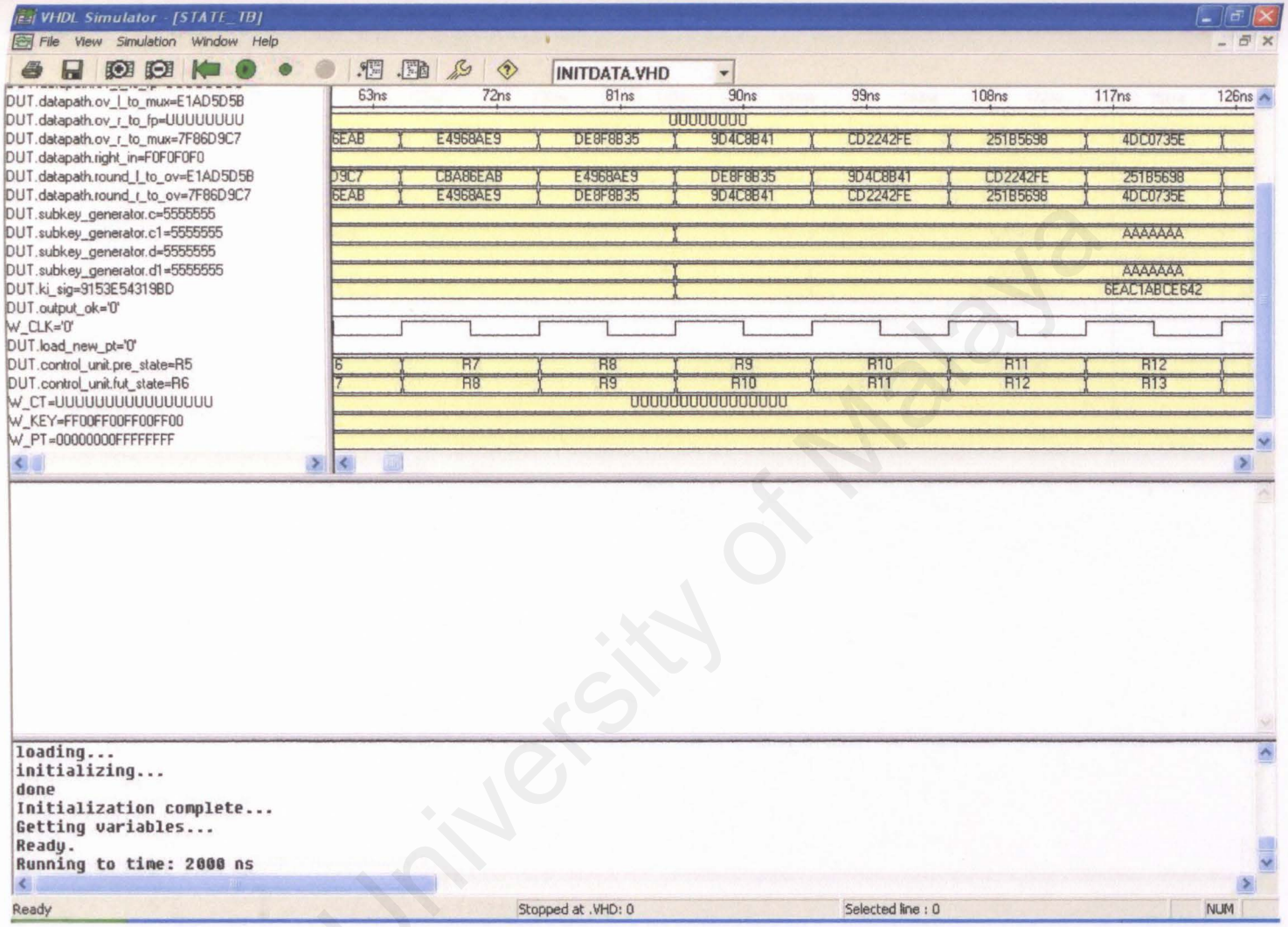


Figure 8.04 Results from states R7 to R12

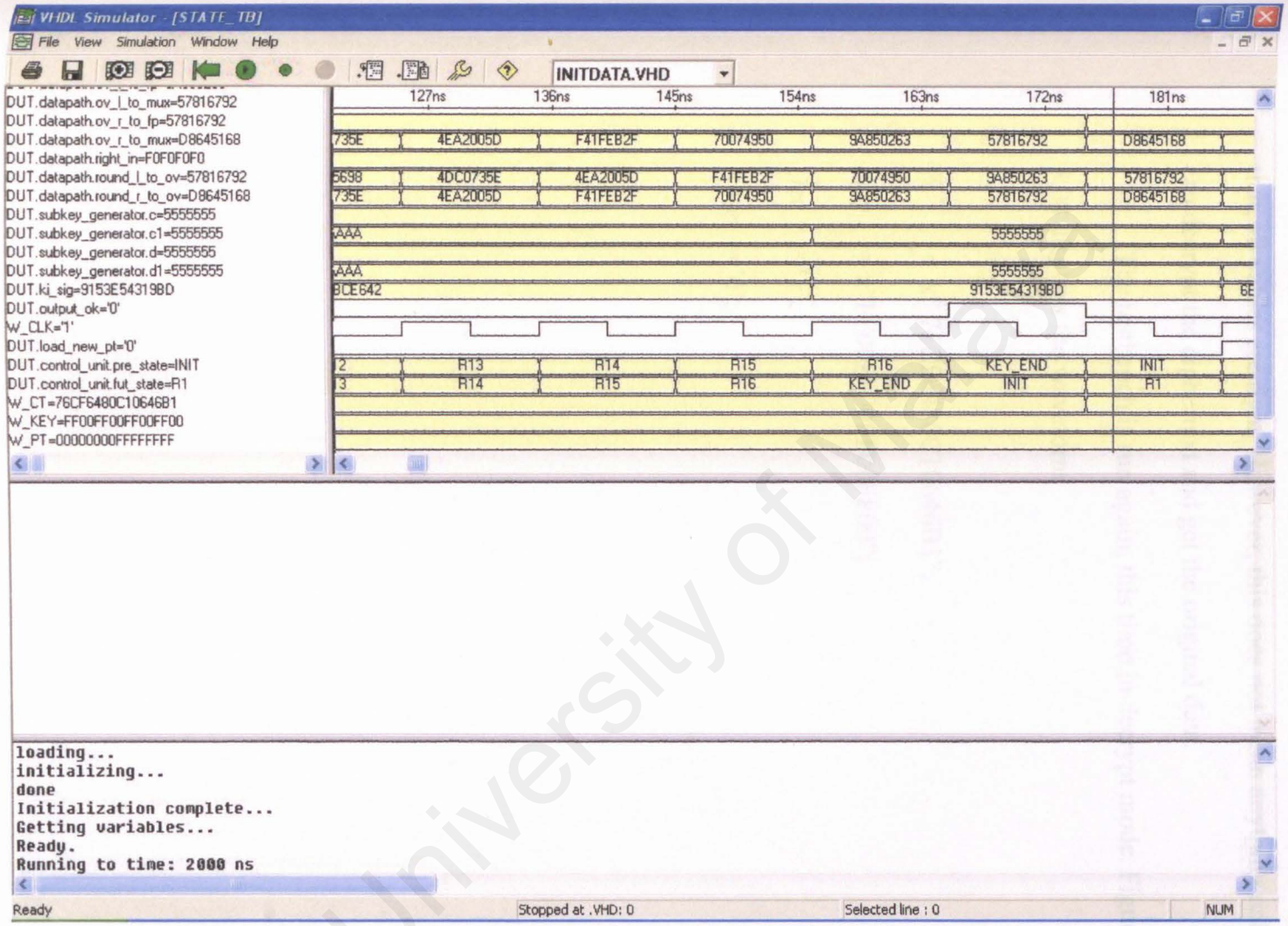


Figure 8.05 Results from states R13 to Key_end

As we can see from the figures, the waveform for each value for keys and permuted data during each state tallies with our Controlled Results. However, the state value for permuted data is shown in its future state. R1 will have no value, but its supposed value is displayed in R2. But other than that, all values are right. However, this does not mean anything unless we can decrypt the ciphertext and get the original data.

The testbench is run again, this time in decrypt mode. Figures 8.06 to 8.08 displays the waveform.

Pt = x" 76CF6480C10646B1";

Key = x"FF00FF00FF00FF00";

Dec ='1';

Reset ='0';

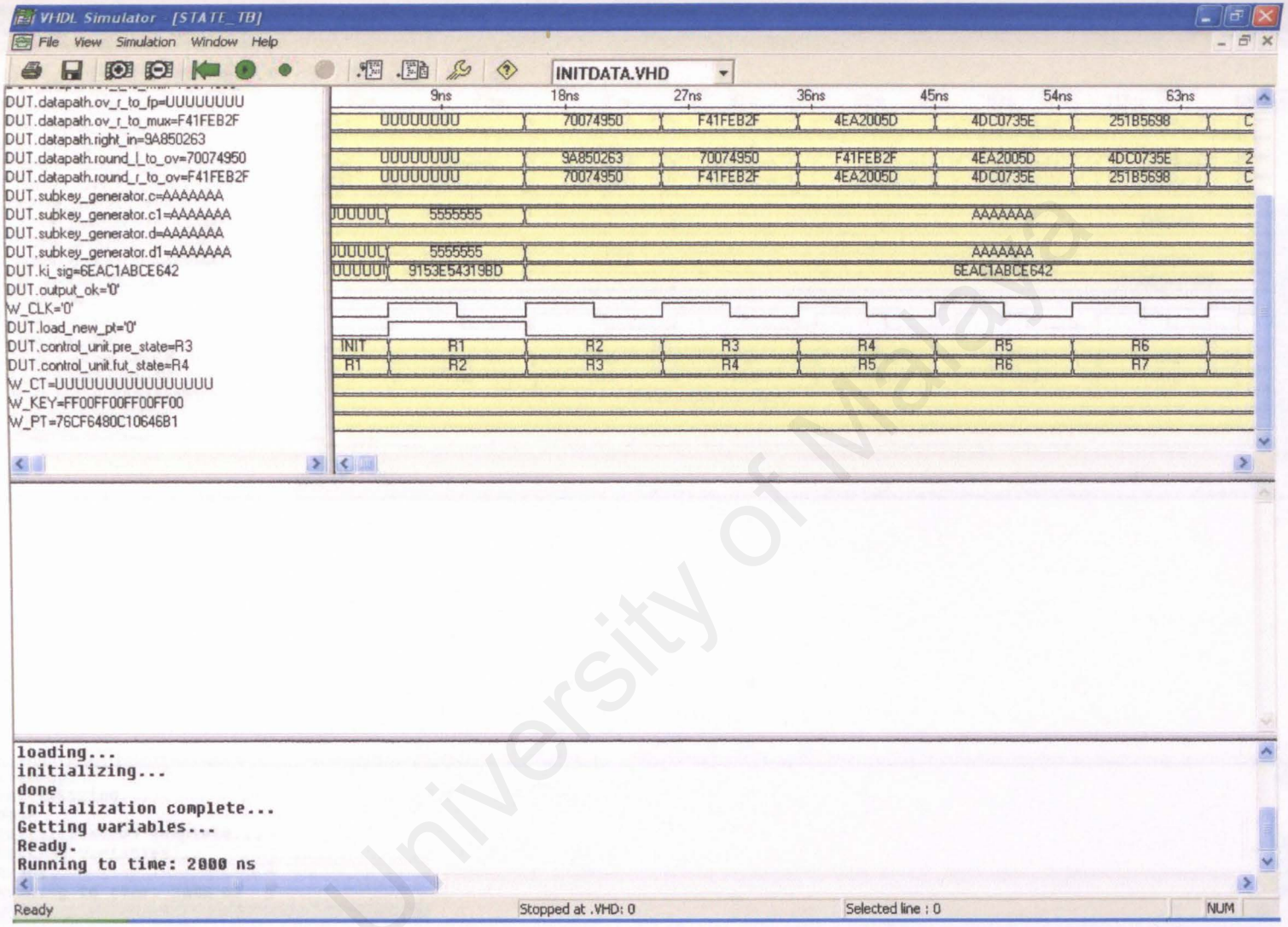


Figure 8.06 Results Init – R6 (decrypt mode)

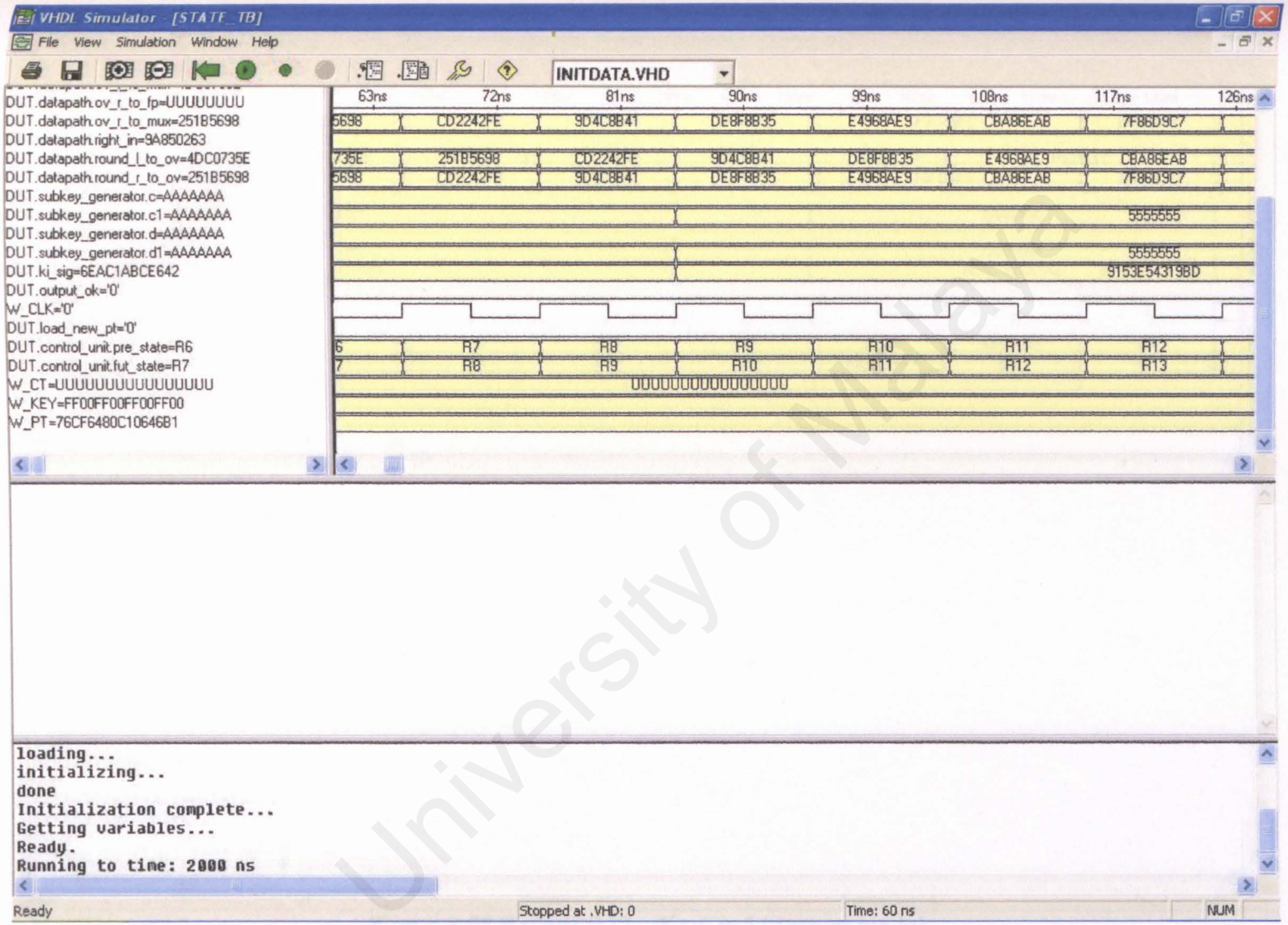


Figure 8.07 Results R7 – R12 (decrypt mode)

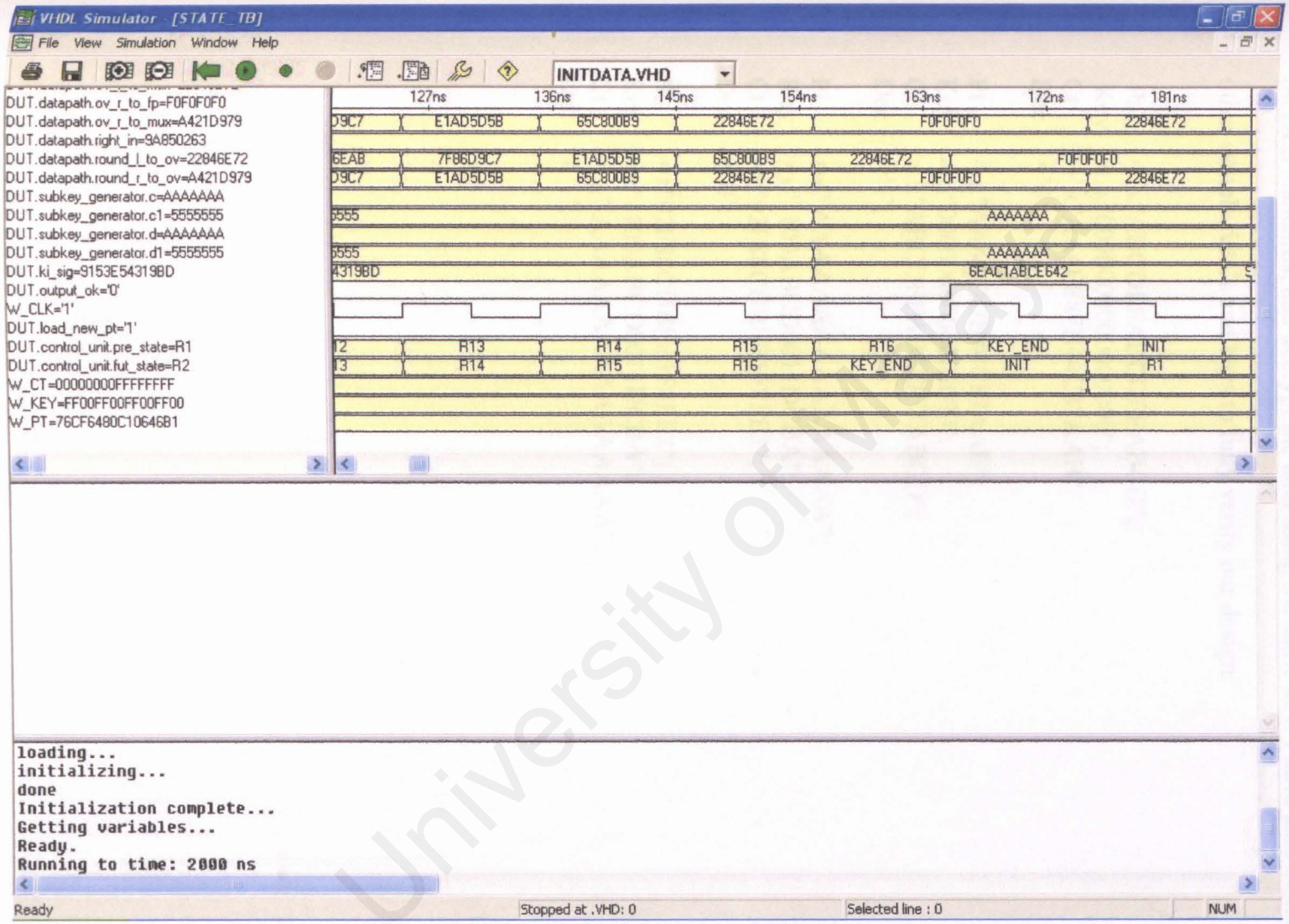


Figure 8.08 Results R12 – Key_end (decrypt mode)

As we can see, ct's value at the end of the process is $00000000FFFFFFFF_H$, which is the original data. From the results obtained, our design has reach its objective, which is to successfully encrypt a 64 bit data based on DES standards, and decrypt the data using the same key. I have also included other sample data's and keys to fully verify our design.

```
Pt    = x"ABCDEFABCDEFABCD";
Key   = x"0000000000000000";
Ct    = x"AA4FE87B44C87AAB"
Dec   ='0';

Pt    = x"AA4FE87B44C87AAB";
Key   = x"0000000000000000";
Ct    = x"ABCDEFABCDEFABCD";
Dec   ='1';

Pt    =x" AAAAAAAAAAAAAAAA";
Key   =x"1100ABCD0011DCBA";
Ct    =x"92F88CDBFB1F8FE2";
Dec   ='0';

Pt    =x"92F88CDBFB1F8FE2";
Key   =x"1100ABCD0011DCBA";
Ct    = x" AAAAAAAAAAAAAAAA";
Dec   ='1';
```

Chapter Nine

Discussion

Chapter 9 Discussion

This chapter discusses the problems related to this project, its strength and weaknesses, future improvements consideration and the projects development process.

In the world of digital systems design, there are two ways in implementing any design. I came to this deduction because of the numerous research papers on DES designs I've come across, there is always trade-off between speed and size of design. There is a fast version, in which all operation are pipelined and done in just one clock cycle, while the other is small, where operations are permuted by looping data through the machine according to requirements, causing delay based on the amount of loops required. Eventually, a hardware design will be hard-wired onto FPGA or CPLB boards. In a fast design, all operations are split to be executed individually; certain architectures are repeated, thus wasting space on the limited space available on the targeted circuit boards. This method is also called pipe-lining.

If space is premium, then a small design is appropriate. By using state machine, the operation can be manipulated. If certain operation could not be done on a clock cycle yet it uses the same resources, it can be set in the future state, where the operation is done

the following clock cycle. Thus this saves space, as we do not have to build an identical architecture.

My DES design is a design where space is premium, in other words, it is a small design. For a variation of my design, I feel it is possible to develop a fast pipelined design. The Initdata architecture could be arranged sequentially, because if we consider the DES algorithm, the left part of the next round is the right part of the preceding one. So only half of the information needs to be stored. By doing this way, we store only 32 bits instead of 64 bits. The figure below illustrates my point.

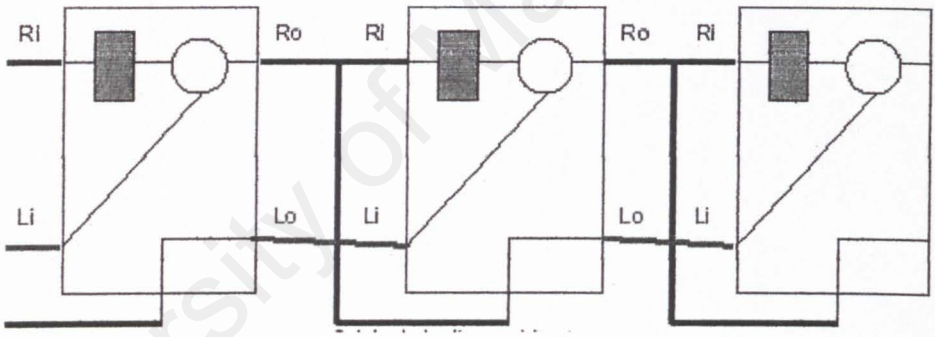


Figure 9.01 Pipelined design for DES

This design will be at least 17 times faster than our original design, but in term of space required for FPGA circuit board, it will be much, much larger.

In term of strength and security the DES offers, the level of security is still adequate, but considering the strength of computers nowadays, it is not safe, because the length of key (56 bit) is

considered too short and is breakable using brute-force attacks. Nowadays, only the Triple DES variation of DES offers adequate security.

In order to arrive at my final design, a lot of effort has been put into designing, then re-designing the DES architecture. The hardest part is in designing the shifter and state machine. My initial design was without a state machine whatsoever, and needless to say, it was anarchy. Credits should be given to the website, "VHDL tutorial through example" by Wei Jun Zhang. My state machine and registers design are taken and adapted from here.

Chapter Ten

Summary

Chapter 10 Summary

This project is about the development of DES cryptosystem in VHDL. In other words, a hardware implementation of DES in VHDL. Two main subjects important in the development for this project are the DES algorithm, and VHDL programming language. In-depth research has been done to these two subjects.

The process flow used is based on "Cascading-Waterfall" model. It is used because of its simplicity and sequential process. The proposed methodology is "Top-down Design/Bottom-up Implementation". Here, in analyzing the system, a top-view is taken, then broken into modules of components. Recursive partitioning will produce subsequent levels, and the smaller modules are referred to as subcomponents. The implementation phase will take a bottom-up approach, where each submodules are built. These submodules are then tested individually and all components will be integrated. In other words, take a "divide-and-conquer" approach in this projects development.

In DES analysis, based on functions involved, it is deduced that DES contains two main modules. All functions are identified, and some are combined to form sub-modules for DES. Results from the analysis will be the basis of our DES design.

For DES design, in addition to the two main modules, two additional modules are required, which are RAM and Controller. The RAM modules acts as temporary buffers, and come in three different designs, in the reg32, ov32 and mux32 modules. The Controller is to control DES operations using state machine.

The tool of choice used in this project is peakFPGA. DES operation is performed manually, so we have a benchmark value to evaluate our design. After individual modules are built, they are linked and simulated. Then, finally, the top-level design of our DES testbench is simulated. From the waveform obtained, our design has successfully encrypted and decrypt 64 bit data based on DES standards, operating in ECB mode.

Appendix

Appendix A - A practical example of the DES algorithm encryption

By Adrian Grigorof - adrian@grigorof.com
December 2000

The sample 64-bit key:
ddd,bbbbbbbb
222,11011110
16,00010000
156,10011100
88,01011000
232,11101000
164,10100100
166,10100110
48,00110000

The 64-bit key is (hex): DE,10,9C,58,E8,A4,A6,30

The original 64-bit key with parity bits

1 1 0 1 1 1 1 0 bits 1-8
0 0 0 1 0 0 0 0 bits 9-16
1 0 0 1 1 1 0 0 bits 17-24
0 1 0 1 1 0 0 0 bits 25-32
1 1 1 0 1 0 0 0 bits 33-40
1 0 1 0 0 1 0 0 bits 41-48
1 0 1 0 0 1 1 0 bits 49-56
0 0 1 1 0 0 0 0 bits 57-64

The original bit positions:

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

The 56-bit key (parity bits stripped)

1 1 0 1 1 1 1
0 0 0 1 0 0 0
1 0 0 1 1 1 0
0 1 0 1 1 0 0
1 1 1 0 1 0 0
1 0 1 0 0 1 0
1 0 1 0 0 1 1
0 0 1 1 0 0 0

The original positions of the bits after the parity is stripped:

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

The positions of the remained 56 bits after Permuted Choice 1 (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

The permuted 56-bit key:

0 1 1 1 0 1 0
1 0 0 0 1 1 0
0 1 1 0 0 0 1
0 0 0 1 0 0 0
0 1 0 0 0 0 0
1 0 1 1 0 0 1
0 1 0 0 0 1 1
1 0 1 1 1 1 1

Split the permuted key into two halves. The first 28 bits are called C[0] and the last 28 bits are called D[0].
C[0]

0 1 1 1 0 1 0
1 0 0 0 1 1 0
0 1 1 0 0 0 1
0 0 0 1 0 0 0

D[0]

0 1 0 0 0 0 0
1 0 1 1 0 0 1
0 1 0 0 0 1 1
1 0 1 1 1 1 1

Calculate the 16 sub keys. Start with i = 1
Perform one or two circular left shifts on both C[i-1] and D[i-1] to get C[i] and D[i], respectively. The number of shifts per iteration are given in the table below.

Iteration # 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Left Shifts 1 1 2 2 2 2 2 2 1 2 2 2 2 2 2 1

C[0]

0 1 1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0

D[0]
0 1 0 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 0 1 1 1 0 1 1 1 1 1

C[1]
1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 0

D[1]
1 0 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 0 1 1 1 0 1 1 1 1 1 0

C[2]
1 1 0 1 0 1 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 1

D[2]
0 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 0 1 1 1 0 1 1 1 1 1 0 1

C[3]
0 1 0 1 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 1

D[3]
0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 0 1 1 1 0 1 1 1 1 1 0 1 0 0

C[4]
0 1 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 1 0 1

D[4]
0 1 0 1 1 0 0 1 0 1 0 0 0 0 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0

C[5]
0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 1 0 1 0 1

D[5]
0 1 1 0 0 1 0 1 0 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 0 0 0 0 0 1

C[6]
0 1 1 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 1 0 1 0 1 0 0

D[6]
1 0 0 1 0 1 0 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 0 0 0 0 0 1 0 1

C[7]
1 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1

D[7]
0 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 1 0 1 0 0 0 0 0 1 0 1 1 0

C[8]
0 1 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0

D[8]
0 1 0 0 0 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 1 0 1 1 0 0 1

C[9]
1 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0

D[9]
1 0 0 0 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0

C[10]
0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1

D[10]
0 0 1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0

C[11]
0 1 0 0 0 1 0 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0

D[11]
1 1 1 0 1 1 1 1 1 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0

C[12]
0 0 0 1 0 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1

D[12]
1 0 1 1 1 1 1 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1

C[13]
0 1 0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0

D[13]
1 1 1 1 1 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0

C[14]
0 0 0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 0 1

D[14]
1 1 1 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 1 1

C[15]
0 0 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 0 1 0 0

D[15]
1 0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 1 1 1 1

C[16]
0 1 1 1 0 1 0 1 0 0 0 1 1 0 0 1 1 0 0 0 1 0 0 0 1 0 0 0

D[16]
0 1 0 0 0 0 0 1 0 1 1 0 0 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1
Permute the concatenation C[i]D[i] as indicated below. This will yield K[i], which is 48 bits long.
Permuted Choice 2 (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

C[0]D[0]
0 1 1 1 0 1 0 bits 1-7
1 0 0 0 1 1 0 bits 8-14
0 1 1 0 0 0 1 bits 15-21
0 0 0 1 0 0 0 bits 22-28
0 1 0 0 0 0 0 bits 29-35
1 0 1 1 0 0 1 bits 36-42
0 1 0 0 0 1 1 bits 43-49
1 0 1 1 1 1 1 bits 50-56

K[0]
0 1 0 0 0 0
1 0 0 1 1 0
0 0 1 1 0 1
1 0 0 0 1 1
0 1 0 0 0 1
1 0 0 0 0 1
1 1 1 1 0 1
0 1 1 1 0 0

Loop back until K[16] has been calculated (for this example, the calculation of the rest of the K[x] is skipped)

Process a 64-bit data block.
Get a 64-bit data block. If the block is shorter than 64 bits, it should be padded as appropriate for the application.

Sample 64 bit data:
86,01010110
233,11101001
158,10011110
172,10101100
222,11011110
95,01011111
244,11110100
177,10110001

The original bit positions:
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

Perform the following permutation on the data block.

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

Original data:
0 1 0 1 0 1 1 0 bits 1-8
1 1 1 0 1 0 0 1 bits 9-16
1 0 0 1 1 1 1 0 bits 17-24
1 0 1 0 1 1 0 0 bits 25-32
1 1 0 1 1 1 1 0 bits 33-40
0 1 0 1 1 1 1 1 bits 41-48
1 1 1 1 0 1 0 0 bits 49-56
1 0 1 1 0 0 0 1 bits 57-64

Permuted data:
0 1 1 1 0 0 1 1
1 1 1 1 0 1 0 1
0 1 1 1 1 1 0 1
1 0 1 0 0 0 1 0
1 1 0 1 1 1 1 0
1 1 0 0 1 0 1 0
0 0 1 1 1 1 1 0
0 0 1 1 0 1 0 1

Split the block into two halves. The first 32 bits are called L[0], and the last 32 bits are called R[0].
L[0]
0 1 1 1 0 0 1 1
1 1 1 1 0 1 0 1
0 1 1 1 1 1 0 1
1 0 1 0 0 0 1 0
R[0]
1 1 0 1 1 1 1 0
1 1 0 0 1 0 1 0
0 0 1 1 1 1 1 0
0 0 1 1 0 1 0 1
Apply the 16 sub keys to the data block. Start with i = 1. Expand the 32-bit R[i-1] into 48 bits according to the bit-selection function below.

Expansion (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

R[0]

1 1 0 1 1 1 1 0 bites 1-8
1 1 0 0 1 0 1 0 bites 9-16
0 0 1 1 1 1 1 0 bites 17-24
0 0 1 1 0 1 0 1 bites 25-32

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

Expanded R[0] or E(R[0])
1 1 0 1 1 1
1 1 1 1 0 1
0 1 1 0 0 1
0 1 0 1 0 0
0 0 0 1 1 1
1 1 1 1 0 0
0 0 0 1 1 0
1 0 1 0 1 1

Exclusive-or E(R[i-1]) with K[i].

E(R[0])
1 1 0 1 1 1
1 1 1 1 0 1
0 1 1 0 0 1
0 1 0 1 0 0
0 0 0 1 1 1
1 1 1 1 0 0
0 0 0 1 1 0
1 0 1 0 1 1

K[0]
0 1 0 0 0 0 1 1 0 1 1 1
1 0 0 1 1 0 1 1 1 1 0 1
0 0 1 1 0 1 0 1 1 0 0 1
1 0 0 0 1 1 0 1 0 1 0 0
0 1 0 0 0 1 0 0 0 1 1 1
1 0 0 0 0 1 1 1 1 1 0 0
1 1 1 1 0 1 0 0 0 1 1 0
0 1 1 1 0 0 1 0 1 0 1 1
XOR: If one, and only one, of the expressions evaluates to True, result is True
Perform Exclusive-or E(R[i-1]) with K[i].

E(R[i-1]) xor K[i]

1 0 0 1 1 1
0 1 1 0 1 1
0 1 1 1 0 0
1 1 1 0 0 1
0 1 1 1 1 0
0 1 1 1 0 1
1 1 1 0 1 1
1 1 0 1 1 1
Break E(R[i-1]) xor K[i] into eight 6-bit blocks.
Bits 1-6 are B[1], bits 7-12 are B[2], and so on with bits 43-48 being B[8].

B[1]
1 0 0 1 1 1

B[2]
0 1 1 0 1 1

B[3]
0 1 1 1 0 0

B[4]
1 1 1 0 0 1

B[5]
0 1 1 1 1 0

B[6]
0 1 1 1 0 1

B[7]
1 1 1 0 1 1

B[8]
1 1 0 1 1 1

Substitute the values found in the S-boxes for all B[j]. Start with j = 1.
All values in the S-boxes should be considered 4 bits wide.

Take the 1st and 6th bits of B[j] together as a 2-bit value (call it m)
indicating the row in S[j] to look in for the substitution.
Take the 2nd through 5th bits of B[j] together as a 4-bit value (call it n)
indicating the column in S[j] to find the substitution.

B[1]
1 0 0 1 1 1
1 2 3 4 5 6 bit order

m = 11 = 3
n = 0011 = 3

Replace B[j] with S[j][m][n].

Substitution Box 1 (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[1][3][3] = 2

B[2]
0 1 1 0 1 1

m = 01 = 1
n = 1101 = 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[2][1][13] = 9

B[3]
0 1 1 1 0 0

m = 00 = 0
n = 1110 = 14

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[3][0][14] = 2

B[4]
1 1 1 0 0 1

m = 11 = 3
n = 1100 = 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[4][3][12]=12

B[5]
0 1 1 1 1 0

m = 00 = 0
n = 1111 = 15

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[5][0][15] = 9

B[6]
0 1 1 1 0 1

m = 01 = 1
n = 1110 = 14

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[6][1][14]=3

B[7]
1 1 1 0 1 1

m = 11 = 3
n = 1101 = 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[7][3][13] = 2

B[8]
1 1 0 1 1 1

m = 11 = 3
n = 1011 = 11

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

S[8][3][11] = 0

Permute the concatenation of B[1] through B[8] as indicated below.

B[1] = S[1][3][3] = 2 = 0010
B[2] = S[2][1][13] = 9 = 1001
B[3] = S[3][0][14] = 2 = 0010
B[4] = S[4][3][12] = 12 = 1100
B[5] = S[5][0][15] = 9 = 1001
B[6] = S[6][1][14] = 3 = 0011
B[7] = S[7][3][13] = 2 = 0010
B[8] = S[8][3][11] = 0 = 0000

B[1-8]

0 0 1 0 1 0 0 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0 1 1 0 0 1 0 0 0 0 0
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

Permutation 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

P(S[1](B[1])...S[8](B[8]))
0 0 1 0
0 0 0 1
0 0 1 0
1 0 0 0
0 1 1 1
0 1 1 0
0 1 0 0
0 1 0 0

Exclusive-or the resulting value with L[i-1].
Thus, all together, your R[i] = L[i-1] xor P(S[1](B[1])...S[8](B[8])),
where B[j] is a 6-bit block of E(R[i-1]) xor K[i].
(The function for R[i] is more concisely written as, R[i] = L[i-1] xor f(R[i-1], K[i]).)

L[0] xor P(S[1](B[1])...S[8](B[8]))

L[0] (see above)
0 1 1 1 0 0 1 1
1 1 1 1 0 1 0 1
0 1 1 1 1 1 0 1
1 0 1 0 0 0 1 0

L[0]

0 1 1 1
0 0 1 1
1 1 1 1
0 1 0 1
0 1 1 1
1 1 0 1
1 0 1 0
0 0 1 0

xor with

P(S[1](B[1])...S[8](B[8]))
0 0 1 0
0 0 0 1
0 0 1 0
1 0 0 0
0 1 1 1
0 1 1 0
0 1 0 0
0 1 0 0

R[1]
0 1 0 1
0 0 1 0
1 1 0 1
1 1 0 1
0 0 0 0
1 0 1 1
1 1 1 0
0 1 0 0

There are 26 modules altogether, and due to space constraints, I have decided to submit it in soft copy. This appendix serves to explain how to use the source code, in term of entering values into the system. Thus, we can say, a very simplified user's manual.

Signals/ Data are entered in the State_TB module. The four parameters are pt, key, dec and reset. Values entered should follow the correct VHDL syntax. Below are the correct examples. Note that the signals are mapped to the P_*signal* port.

```
begin
W_PT     <= x"00000000ffffffff";
W_KEY    <= x"ff00ff00ff00ff00";
W_RESET  <= '0';
W_DEC      <='0';

W_PT <= x"76CF6480C10646B1";
W_KEY    <= x"ff00ff00ff00ff00";
W_RESET <= '0';
W_DEC      <='1';
```

We can also enter values in pure binary from, but we will have to omit the x operand before the double quote.

Reference

References

Books:-

Zainalabedin Navabi. (1998). VHDL : Analysis and Modeling of Digital Systems. 2nd ed. McGraw-Hill.

Peter Ashenden (1998). The VHDL Cookbook. 1st ed.

Eric Maiwald. (2001). Network Security :A beginner's Guide. 1st ed. McGraw-Hill.

Internet:-

1. http://www.opencores.org
2. http://www.acc-eda.com/vhdlref/
3. http://www.itl.nist.gov/fipspubs/index.htm
4. http://www.itl.nist.gov/fipspubs/fip74.htm
5. http://www.itl.nist.gov/fipspubs/fip81.htm
6. http://www.itl.nist.gov/fipspubs/fip46-2.htm
7. http://isc.faqs.org/faqs/cryptography-faq/
8. http://www.eventid.net/docs/desexample.htm
9. http://www.free-ip.com/DES/index.html
10. http://www.cs.ucr.edu/content/esd/labs/tutorial/
11. http://www.vhdl-online.de/~vhdl/TB-GEN/ent2tb1.htm