A VHDL IMPLEMENTATION OF TWOFISH BLOCK CIPHER

A PROJECT REPORT

Submitted by

SUGANESWARI J. 71203106051

VADIVUKARASI B.L. 71203106305

In partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY,

COIMBATORE-641006

ANNA UNIVERSITY: CHENNAI 600 025

APRIL 2007

ANNA UNIVERSITY: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report **"A VHDL IMPLEMENTATION OF WOFISH BLOCK CIPHER"** is the bonafide work of **"VADIVUKARASI .L."** carried out the project work under my supervision.

IGNATURE 23/4/07	**SIGNATURE** 23/4/07
r.RAJESWARI MARIAPPAN	Mrs.K.THILAGAVATHI
IEAD OF THE DEPARTMENT	**SUPERVISOR**
	LECTURER
lectronics and ommunication Engineering umaraguru college of Technology oimbatore -641006.	Electronics and Communication Engineering Kumaraguru college of Technology Coimbatore-641006

he candidates with university register number **71203106305** was examined by s in the project Viva Voce examination held on

nternal Examiner 24/4/07

External Examiner 24/4

ACKNOWLEDGEMENT

We acknowledge our thanks to **Dr. JOSEPH.V.THANIKAL, PRINCIPAL,** Kumaraguru College of technology, Coimbatore, for providing an opportunity to take up this project.

We take this opportunity to thank, **Dr. RAJESWARI MARIAPPAN, HEAD OF THE DEPARTMENT,** Electronics and Communication Engineering, Kumaraguru College of Technology, Coimbatore, for her extended support in completion of the project.

We wish to record our gratitude to **Mr. K.RAMPRAKASH, PROFESSOR, THE PROJECT COORDINATOR,** Department of Electronics and Communication Engineering, Kumaraguru College of Technology, Coimbatore, for his extended support in the completion of the project.

We take this opportunity to express our profound thanks to our guide **Mrs. K.THILAGAVATHI, LECTURER,** Department of Electronics and Communication Engineering, Kumaraguru College of Technology, Coimbatore, for her valuable guidance and support for the successful completion of the project work.

Finally, we express our thanks to our parents and friends for their encouragement in completion of this project work successfully.

ABSTRACT

As Electronic communication has spread to every area of modern life, cryptography has become an essential component to control the authenticity, integrity and confidentiality.

The demand for efficient and secure ciphers capable of providing increased protection at low cost. Among these new algorithm is Twofish, a promising 128-bit block cipher that could soon be chosen by the National Institute of Standards and Technology (NIST) as the Advanced Encryption Standard (AES), replacing Data Encryption Standard (DES) at the core of many encryption systems Worldwide.

In this project, a version of Twofish with 128-bit key length are implemented using an Xilinx FPGA. All the features of the algorithm are implemented, including sub-key generation, encryption and decryption. The design are tested for interoperability with a reference software implementation of the algorithm.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TABLE **PAGE NO.**

INTRODUCTION

1.1 INTRODUCTION

Authenticity, integrity, confidentiality and non- reputability of Private data is now a challenging aspect in the field of Communication Networks.

Twofish is one of the first finalist for the Advanced Encryption Standards (AES) competition held by the National Institute of Standards and Technology(NIST). AES is the successor to the aging Data Encryption Standards algorithm (DES) and Twofish is the successor to Blowfish, a well-established cipher without any known flaws that is more efficient than DES.

In our project, VHDL (Veryhigh Hardware Descriptional level Language) is chosen as a language to describe Twofish implementation. VHDL is a standard language for hardware description and it is supported by computer aided design software of all major FPGA device vendors. It also allows describing circuit function without the need to specify the circuit structure. The result substantially depends on software used for synthesis.

The verified VHDL code is then synthesized and optimized using FPGA. The obtained net list is then exported to Xilinx foundation series to create the implementation. The Xilinx design environment is used for timing simulation.

DEVELOPMENT ENVIRONMENT

2.1 Hardware Environment

Xilinx Spartan 100TQ144

2.2 Software Environment

Simulation	ModelSim PE 5.4e
Programming	VHDL
Synthesis and Implementation	Xilinx 8.2i

Ovearll structure of Twofish cipher

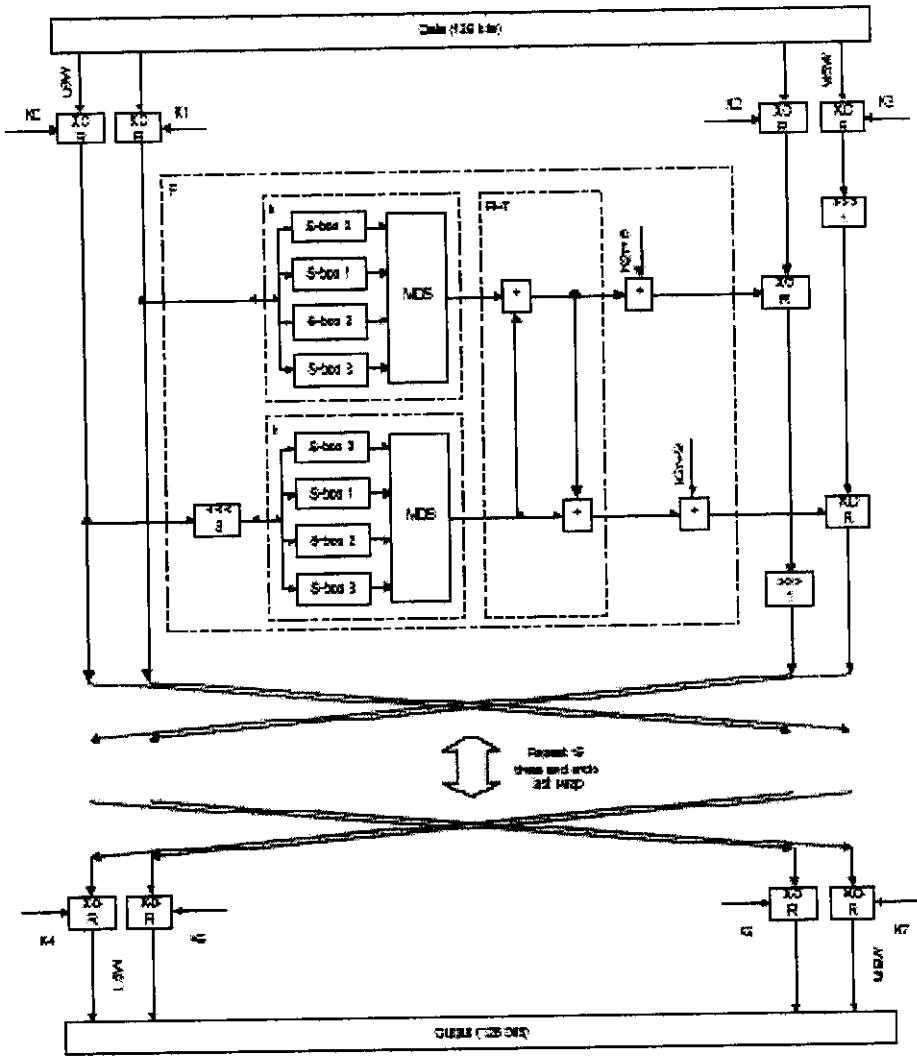


Figure 3.1: Overall Structure of Twofish

Figure 3.1, shows the overall view of the twofish structure. from the figure, the input is first latched into a register. It is then separated into 4 words that are XOR-ed with four subkeys k0...k3. This step is called the input whitening. The data then goes through a module called the F-function where various rotations, transformations and permutations are applied to it.

This F-function is made of two h-functions containing key-dependant S-boxes, a Maximum Distance Separable matrices (MDS) and a pseudo-hadamard transform (PHT); after going 16 times through this function, the four words of data are once again XOR-ed with four subkeys K4...k7. This step is called the output whitening. Finally, the encrypted or decrypted data is latched into the output register.

3.2 Subkey generation

Figure 3.2, shows the key generation module. This module is used to generate the 40 k-subkeys needed by the algorithm. It takes even and odd numbers ranging from 0 to 39 as input, which depends upon the global user supplied key. Subkey generation uses the same building blocks as f-function. This is important as it allows the use of the same function for both key generation and encryption, thus reducing the hardware required to implement this algorithm.

There is another set of subkeys used in the algorithm: the S-subkeys. These are computed by means of multiplying an appropriate part of the global key with an RS matrix. An important issue regarding the latter is that they are computed once for a particular global key, and stay fixed during the entire encryption and decryption process.

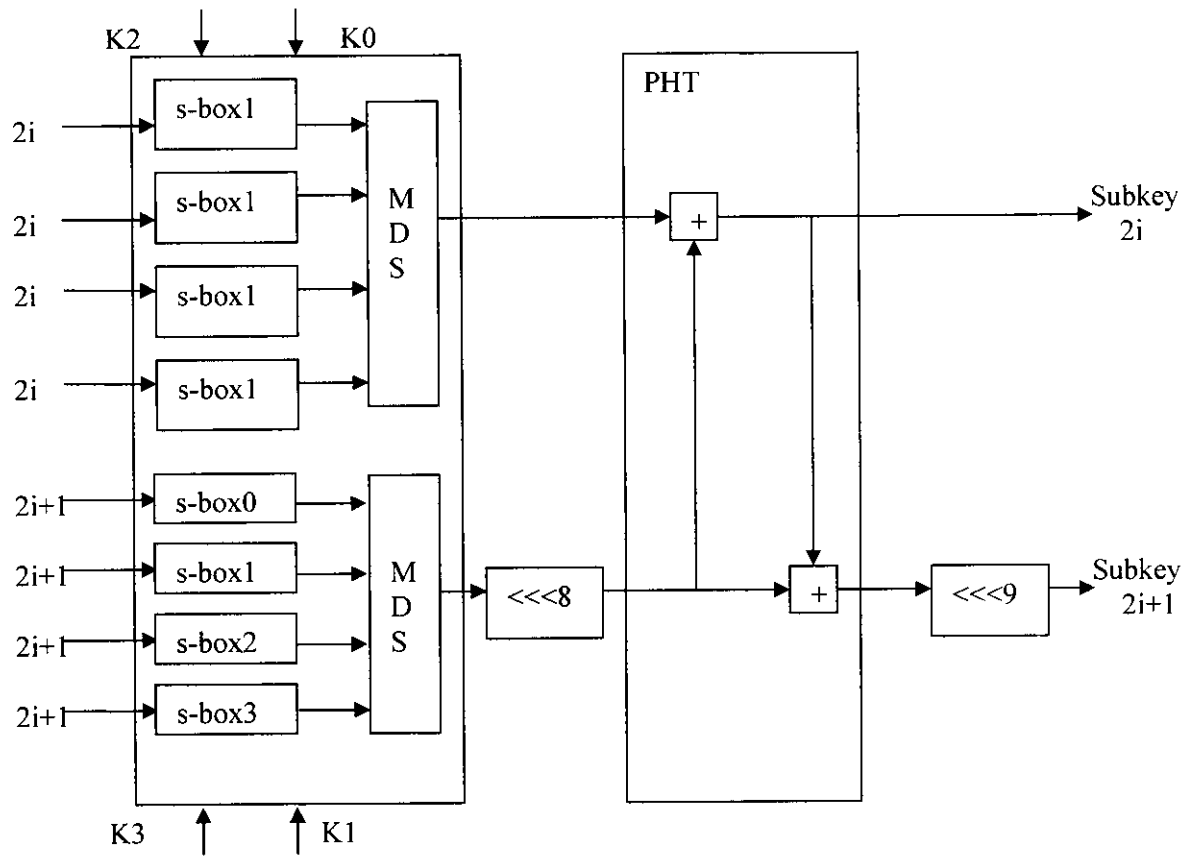


Figure 3.2 : Subkey generation module

DESIGN AND IMPLEMENTATION

4.1 Design Decisions

The Twofish structure offers a great deal of flexibility in terms of space versus speed tradeoffs. We decided to go for a minimum hardware implementation of the algorithm with hopes of fitting the circuit on an Xilinx FPGA.

In order to do so, some design decisions had to be made. It was decided that only one h-function, instead of two, would be used for computing the k-subkeys and the encrypted data.

The second design decision regarded the computation of the k-subkeys. The two choices available were to either pre-compute the k-subkeys or store them in a ram memory (full keying) or to compute the k subkeys on the fly as needed (zero keying). It was decided that zero keying best suited our first design choice as the ram needed would consume too much space on the FPGA we aimed at using.

UILDING BLOCKS

1. Q-Permutations

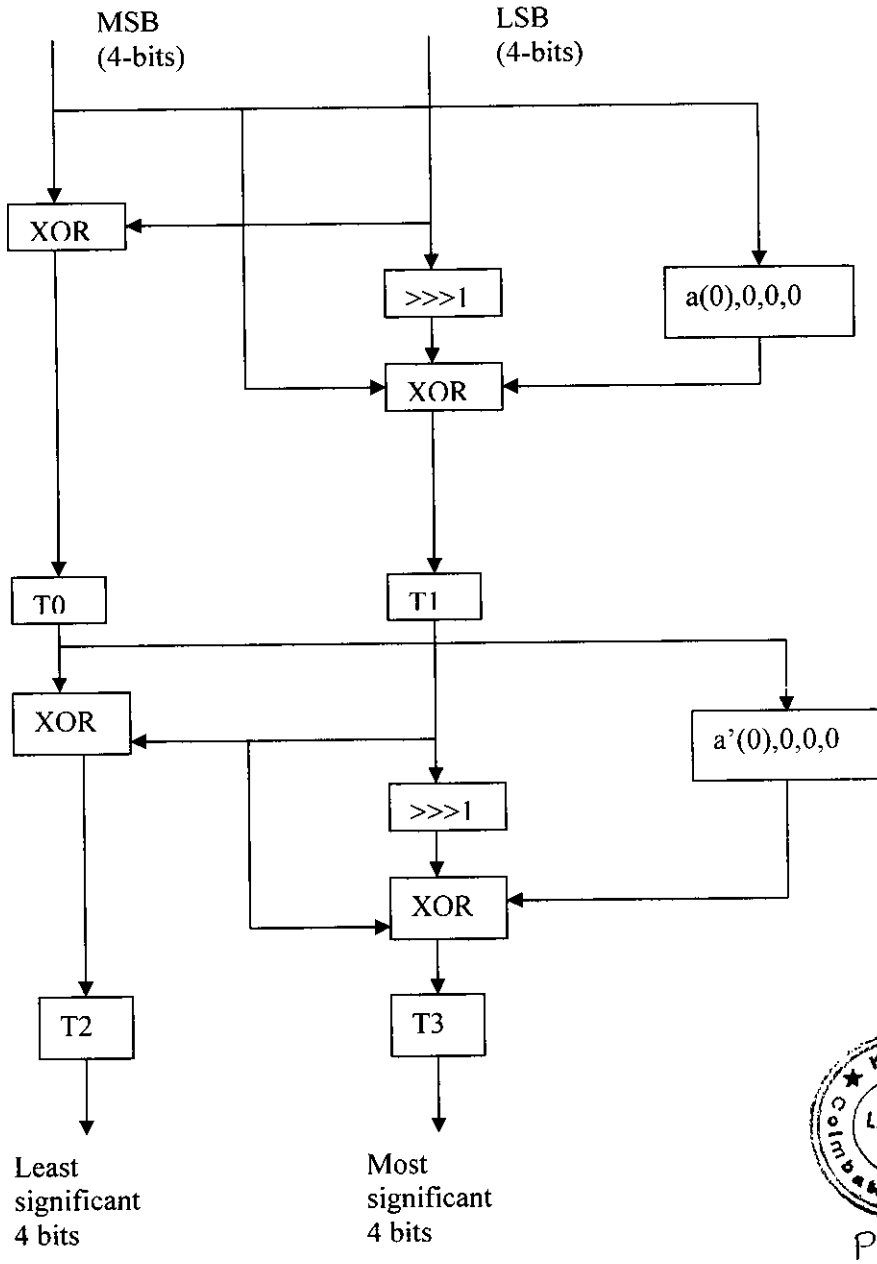


Figure 4.2.1 :Q-Permutation

The Q-Permutation is at the core of the design of Twofish. The permutation is executed on a byte of input, which is split in two before being rotated and modified along different paths. The most important operations of the permutation are executed with four lookup tables labeled t0, t1, t2 and t3.

Each lookup table takes 4 bits of input and produces a 4-bit value. Each lookup table thus has 16 entries of 4 bits. There are four lookup tables per q-permutation and two different q-permutations, q0 and q1, each with its set of lookup table.

The lookup table could have been programmed into ROM. As we will soon see, a large number of q-permutations have to operate in parallel and thus a separate ROM would have been needed for each instance of the q-permutation.This would have taken too much resources for the small Xilinx on which we aimed to implement the cipher.

		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
q_0	t_0	8	1	7	D	6	F	3	2	0	B	5	9	E	C	A	4
	t_1	E	C	B	8	1	2	3	5	F	4	A	6	7	0	9	D
	t_2	B	A	5	E	6	D	9	0	C	8	F	3	2	4	7	1
	t_3	D	7	F	4	1	2	6	E	9	B	3	0	8	5	C	A
q_1	t_0	2	8	B	D	F	7	6	E	3	1	9	4	0	A	C	5
	t_1	1	E	2	B	4	C	3	7	6	D	A	5	F	9	0	8
	t_2	4	C	7	5	1	6	9	A	0	E	D	8	2	B	3	F
	t_3	B	9	5	1	C	3	D	E	6	4	7	F	2	0	8	A

Table 4.2.1 :Lookup table for Q-permutation

The lookup tables were thus implemented using logic, more precisely sing behavioral VHDL.

2.2: S-Boxes

The S-Box operates on a 32-bit word. Each byte of the word asses through three Q- Permutations as can be seen from Figure 4.2.2. The utput of each bank of q-permutations is then recombined into a word and OR-ed with a 32-bit value.

These two values are derived from the key material and wofish's s-boxes are thus referred to as "key-dependent" boxes (KDSBs).

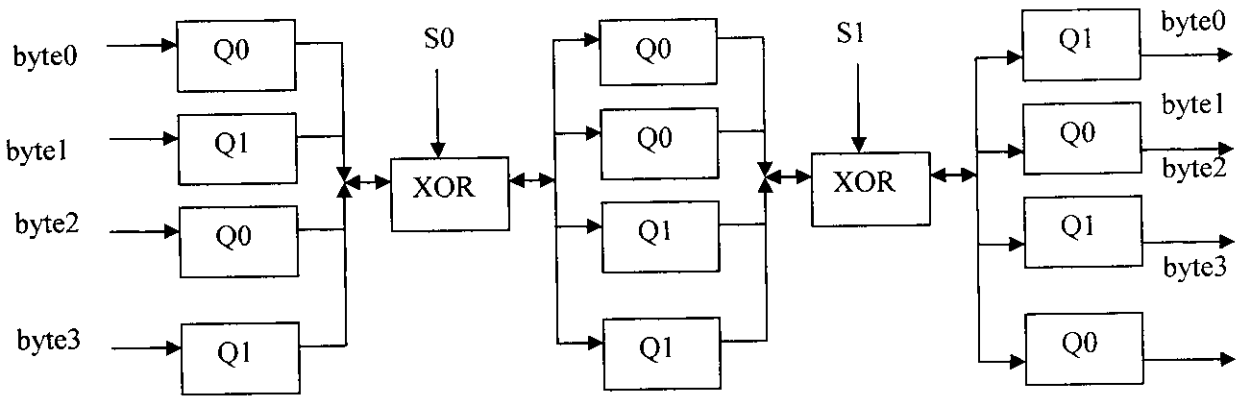


Figure 4.2.2: S-box

The following equations describe how the four KDSBs are obtained in the 128-bit key case:

$$s0(x) = q1(q0(q0(x) \text{ XOR } k0,0) \text{ XOR } k1,0)$$
$$s1(x) = q0(q0(q1(x) \text{ XOR } k0,1) \text{ XOR } k1,1)$$
$$s2(x) = q1(q1(q0(x) \text{ XOR } k0,2) \text{ XOR } k1,2)$$
$$s3(x) = q0(q1(q1(x) \text{ XOR } k0,3) \text{ XOR } k1,3)$$

In the above equations, x is the KDSB input, si(x) is the KDSB
put and ki,j are bytes derived from the cipher key bytes. The actual robustness to differential
l linear cryptanalysis is guaranteed by the q0 and q1 fixed
mutations.

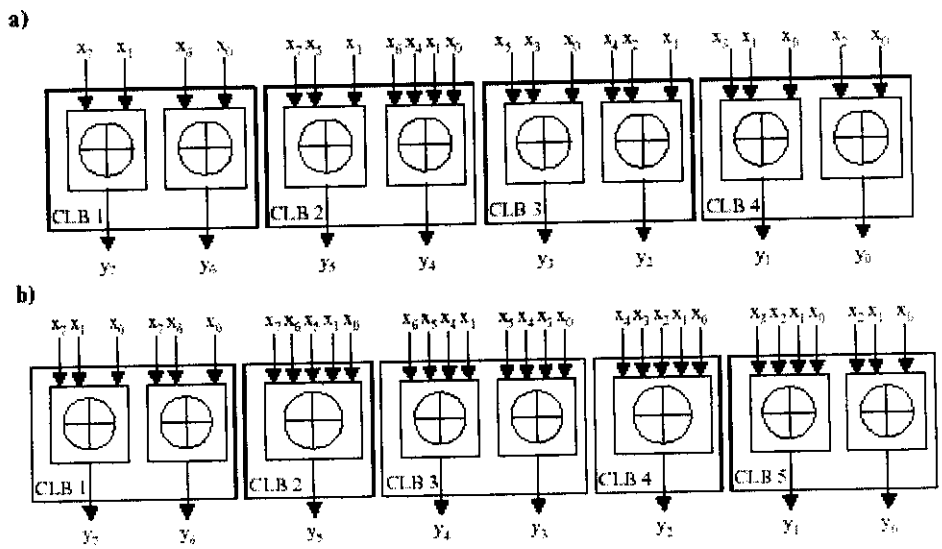
4.2.3 Maximum Distance Separable matrix

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix} \bullet \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Figure 4.2.3: MDS matrix

The Maximum Distance Separable (MDS) matrix is 4x4 matrix of bytes that multiplies a vector of four bytes. Multiplications are carried out in the Galois Field GF(2^8) with the primitive polynomial x8 + x6 + x5 + x3 + 1.

GALOIS FIELD INTERNAL STRUCTURE



a) Multiplication by $5B_{16} \equiv x^6 + x^4 + x^3 + x + 1$
b) Multiplication by $EF_{16} \equiv x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$

Figure 4.2.4: Multiplication by a constant in the GF(2^8)

Each byte converted into a polynomial in which each power p of x is present only if the p-th bit is 1. A multiplication in GF amounts to a multiplication of polynomials followed by a division by the primitive polynomial.

The result is converted back to a bit vector by setting a bit to 0 if the corresponding power of x has an odd coefficient and 1 otherwise (modulo 2 division).

In this case the computations are fairly straightforward since there are only three coefficients:0x01, 0xef and 0x5b. The result of a multiplication can be reduced to a series of XOR's for each bit of the output. for example, multiplying x by 5b results in byte y:

y0 = x2 XOR x0

y1 = x3 XOR x1 XOR x0

...

y7 = x7 XOR x1

This is obtained from Galois Field internal structure Figure 4.2.4.

4.2.4: Reed-Solomon matrix

Figure 4.2.5: RS matrix

The Reed-Solomon (RS) matrix is similar to the MDS matrix. In this case the multiplication is executed between an 8x4 matrix and a vector of 8 bytes. The computations are done in GF(28) with a different prime polynomial: x8 + x6 + x3 + x2 + 1.

Unlike the MDS matrix, the RS matrix has a large number of different multiplicands. In order to minimize the resources used,a series of XOR's was derived for each of the 23 multiplicands to give equations similar to those seen in the MDS matrix.

These equations are derived. This was done by first computing a general equation for the multiplication of two polynomials in GF(2^8). The equation was then used to compute each 23 cases. Once well understood the process was used with a minimum risk of error by inserting the constants into the derived equations using a text editor.

4.2.5 Operation Selector

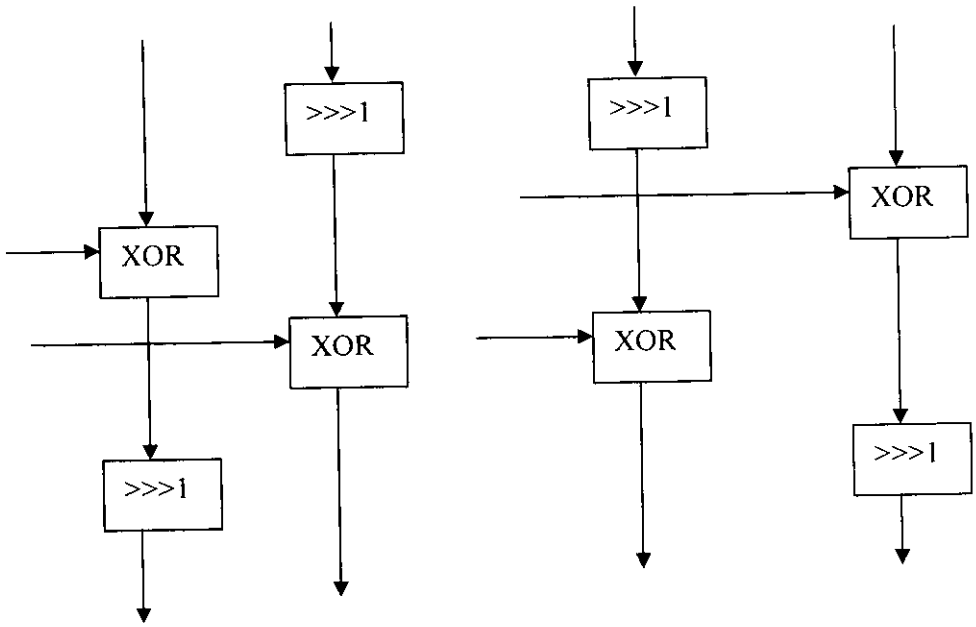


Figure 4.2.6: Selection of Encryption or Decryption

Twofish is a very symmetric algorithm. Encryption and decryption can be executed with almost all the same pieces of hardware. The first difference is that the sub-keys must be used in reverse order. The second difference is at the output stage of the round. As you can see in Figure 4.2.6, the output stage consists of a rotation and an XOR. The output stages are mirrors of each other and can thus be constructed by implementing each path as in Figure 4.2.7.

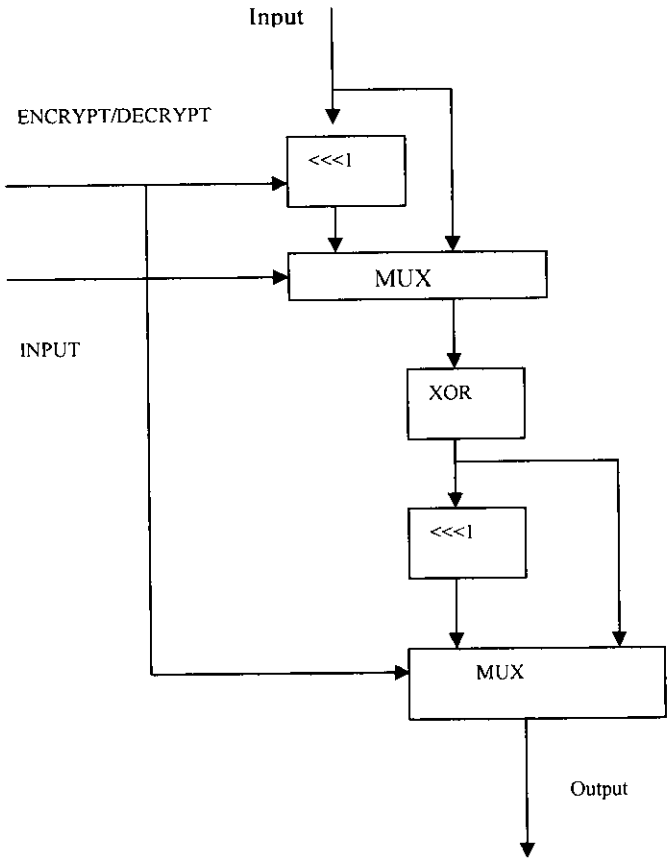


Figure 4.2.7: Building block for operation selection

4.3 INTEGRATION AND OVERALL STRUCTURE

4.3.1 Input Register Module

The input register module combines the data register with the input whitening stage. The data block is 128 bits wide. A signal forces the data to be latched into the register. The input whitening is done asynchronously.

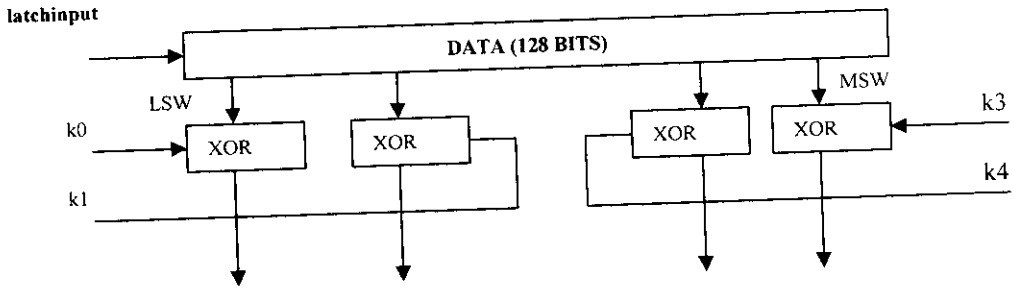


Figure 4.3.1: Input Register Module

The only outputs of this module are four 32-bit words
rresponding to the input whitening of 4 words on the input data. This output is
lid when the 4 sub-keys are set correctly on the input. Sub-keys k0, k1, k2
d k3 should be used for encryption and k4, k5, k6 and k7 for decryption.

3.2 Output Register Module

The output register module incorporates the output whitening stage
nd the output register. four words of input are XOR-ed with 4 sub-keys to produce
ne output.

The sub-keys must be set to k4, k5, k6 and k7 for encryption and k0, k1, k2
and k3 for decryption. Because of the structure of our cipher only two keys will be
available at any time. The output is thus latched in two steps with signals LatchLS and
LatchMS.

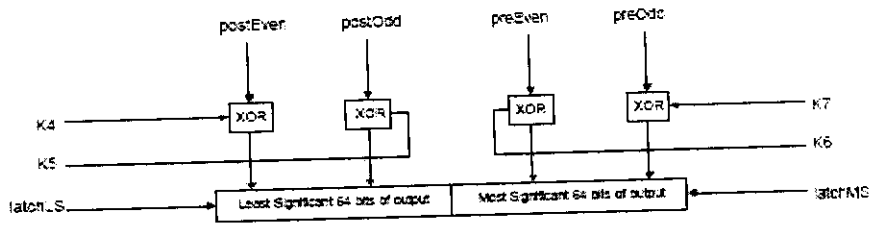


Figure 4.3.2: Output Register Module

.3Key Register Module

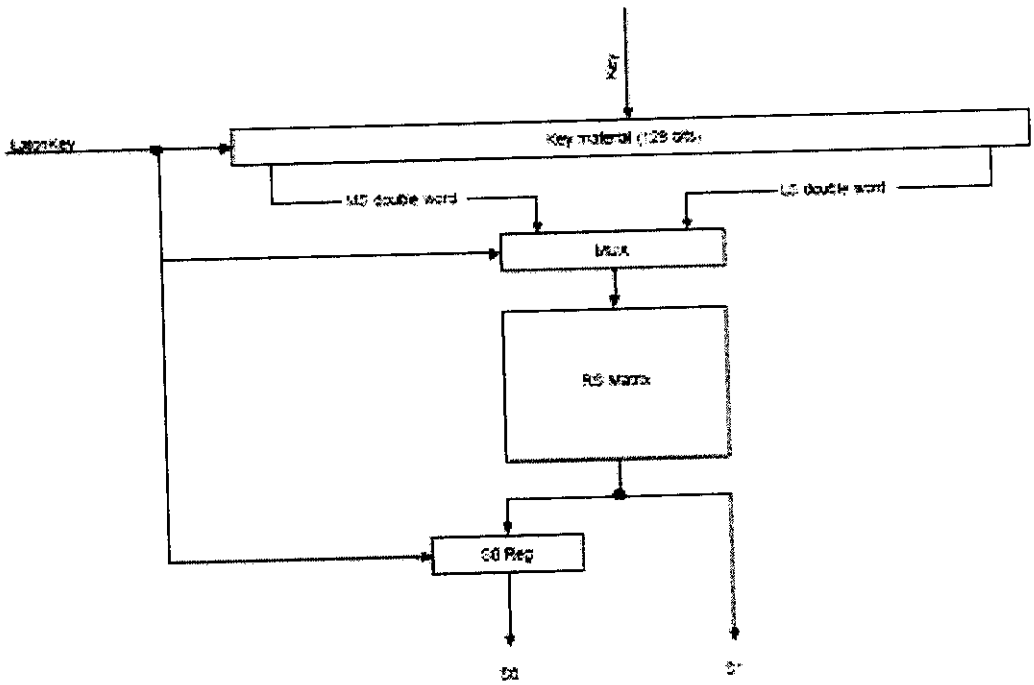


Figure 4.3.3: Key Register Module

The key register module is used to store the 128-bit key material and to produce two derived values (s0 and s1) using the RS matrix. The key is simply latched into the register when the 'latchkey' signal is raised.

The two s-values are computed in two steps by reusing a single RS-matrix. During the clock cycle in which the key is latched a multiplexer sets the input of the RS matrix to the least-significant double word of the key and the result is stored in the s0 register. At all other times the input is set to the most-significant double word and the output s1 gives the value of s1.

4.3.4 Modified F-function

Due to our design decisions of minimizing the hardware, the structure of the cipher had to be modified in order to be able to use the same h-function for all computation purposes. The following diagram shows the modified F-function that was designed in order to meet our requirements:

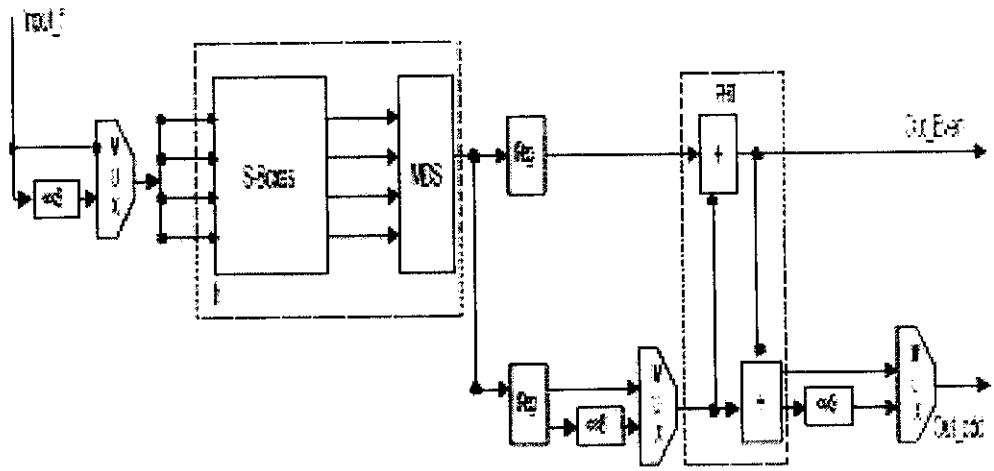


Figure 4.3.4: Generalized F-Function

As can be observed, the input and a rotated version of the input remultiplexed to reproduce both inputs that can be presented at the input of an h-function. Moreover, in order to compute the result of the PHT, two registers had to be added to store the value of the first input or the former while the second was being computed. Thus, these two registers act as pipeline registers. The other two multiplexers added are used to choose between the path used for computing a key and the path used to encrypt or decrypt data.

4.3.5 Designed overall structure

Integrating this modified F-function to the other elements of our design resulted in the final structure (Figure 4.3.5).The four multiplexers shown on top are used to select between the data from the clear text module used at the beginning of an encryption/decryption cycle and the data obtained after each round of encryption/decryption.

The registers at the output of the modified F-function are also "pipeline" registers. They store the values of the k-subkeys (k2r+8 and k2r+9, r ranging from 0 to 15) used in each encryption/decryption cycle.

After each round the data is latched into 4 registers. This step also performs the required swap after each round. A finite state machine was implemented to control the system.

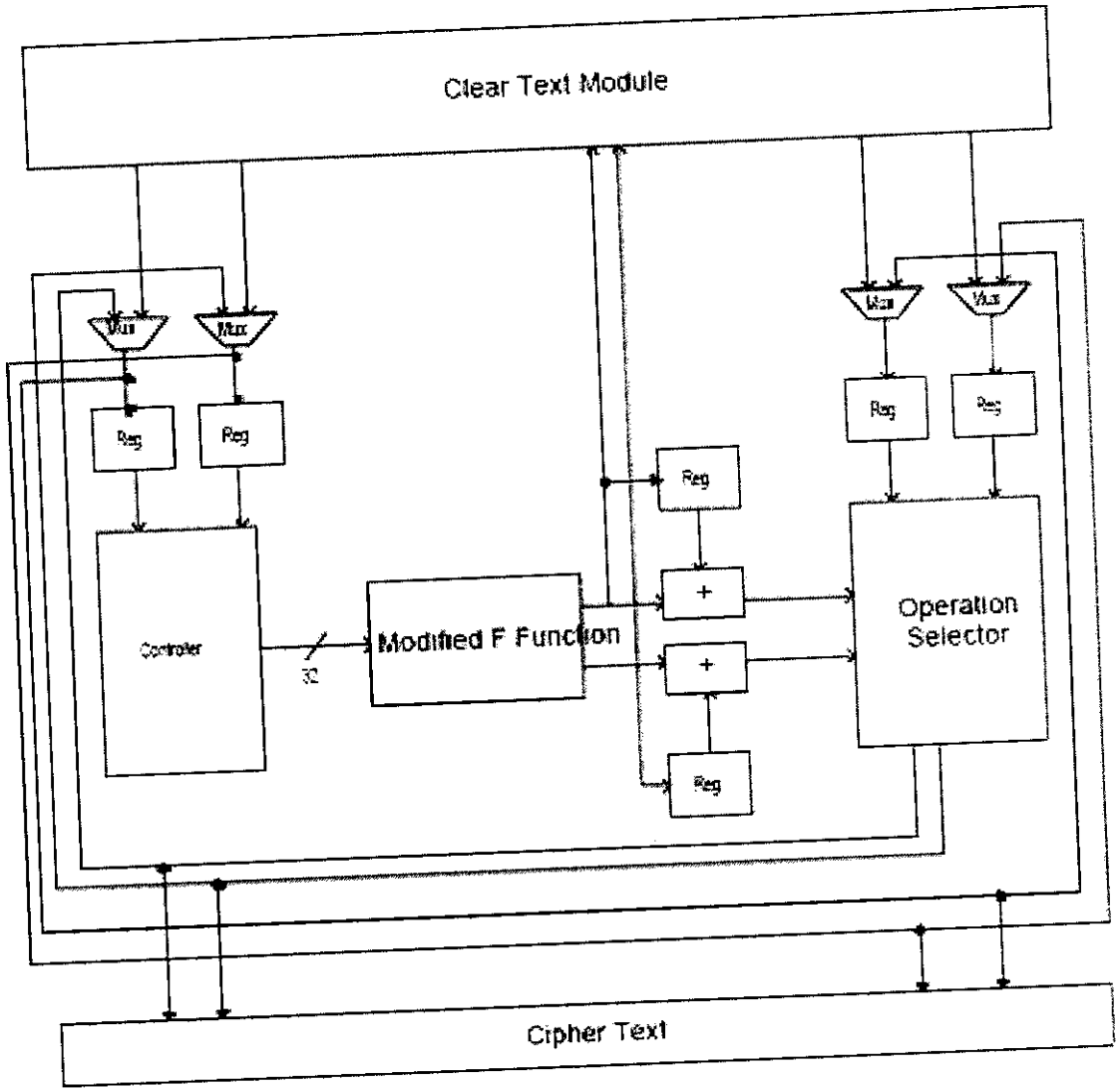


Figure 4.3.5: Structure of the Cipher

4.3.6 Controller

The cipher is controlled by a relatively complex finite state machine. While in the idle state, the controller waits for a user request and advertises its availability by raising a signal. The 'loadkey' signal makes the controller go to the 'loadkey' state in which it latches the key material from the 128-bit input port.

The 'start' signal puts the controller in the encryption or decryption mode. Note that the controller operates almost exactly the same way when doing encryption or decryption. The first step is to compute the four input whitening sub-keys. After each pair is computed the two registers on the top left of Figure 4.3.5 are latched with the output of the input register module. After the second pair, The two other registers on the right are latched.

The controller then goes through 16 rounds of encryption or decryption. These rounds consist of four states. The even and odd keys for the round are computed and latched into to registers. The f-function is then used to compute a pair of results, which are added to the content of the registers. At the end of the round the result is latched into registers. The outputs of the rounds are also swapped between the left and right side of the circuit.

After the end of the 16 rounds the four output whitening sub-keys are computed and used two-by-two to whiten the output. Soon after the controller goes back to the idle state, the cipher text register is latched with the output of the process. Note that sub-keys have to be generated by setting the input of the f-function with values ranging from 0 to 39. Doing this directly in the controller would be awkward.

The controller thus uses a counter to do the job. This counter counts in the order in which the keys need to be generated (0,1,2,3,8,9,10…39,4,5,6,7 for encryption). It can be disabling so that it counts only when needed.

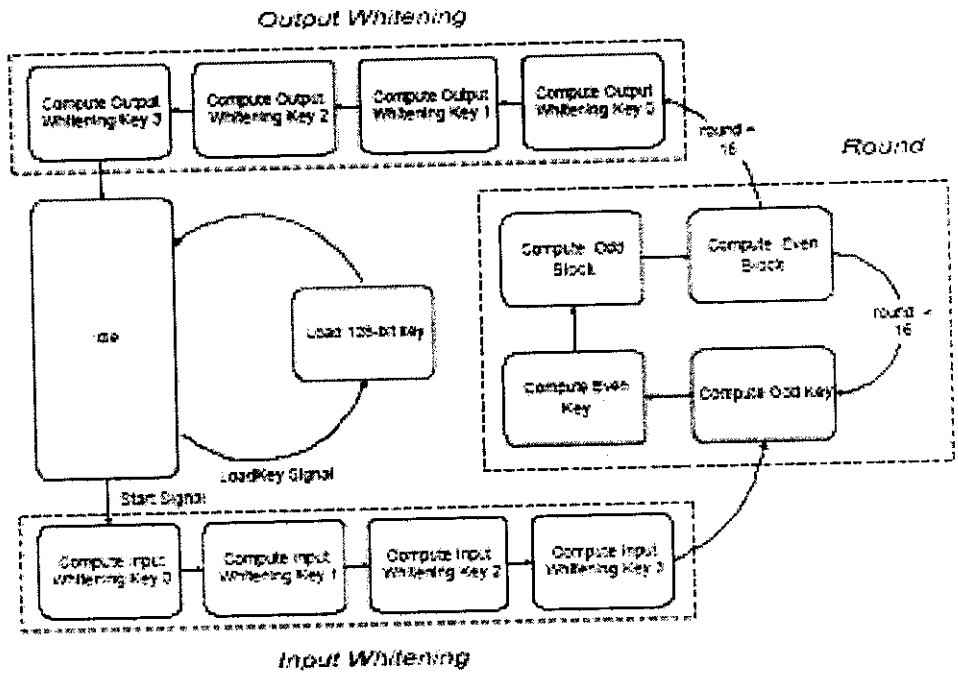


Figure 4.3.6: Controller

The cipher is controlled by a relatively complex finite state machine. While in the idle state, the controller waits for a user request and advertises its availability by raising a signal. The 'loadkey' signal makes the controller go to the 'loadkey' state in which it latches the key material from the 128-bit input port.

The 'start' signal puts the controller in the encryption or decryption mode. Note that the controller operates almost exactly the same way when doing encryption or decryption. The first step is to compute the four input whitening

sub-keys. After each pair is computed the two registers on the top left of Figure 4.3.6 are latched with the output of the input register module.

After the second pair, the two other registers on the right are latched. The controller then goes through 16 rounds of encryption or decryption. These rounds consist of four states. The even and odd keys for the round are computed and latched into to registers. The F-function is then used to compute a pair of results, which are added to the content of the registers. At the end of the round the result is latched into registers. The outputs of the rounds are also swapped between the left and right side of the circuit. After the end of the 16 rounds the four output whitening sub-keys are computed and used two-by-two to whiten the output. Soon after the controller goes back to the idle state, the cipher text register is latched with the output of the process.

Note that sub-keys have to be generated by setting the input of the F-function with values ranging from 0 to39. Doing this directly in the controller would be awkward. The controller thus uses a counter to do the job. This counter counts in the order in which the keys need to be generated (0,1,2,3,8,9,10…39,4,5,6,7 for encryption). It can be disabling so that it counts only when needed.

SYNTHESIS

5.1 Debugging

The Twofish cipher contains a relatively large number of elements that must be implemented exactly as specified to produce the correct result. The specifications of the cipher contain many subtleties that can easily go unnoticed for non-cryptographers. Studying the specifications and the sample implementation by chodowiec and gaj [3] allowed us to implement an almost correct cipher but as expected some details were wrong.

A complex system such as this is difficult to test component by component. After reviewing all the VHDL code to find flaws we turned our attention to the reference c-implementation of the Twofish cipher and to know answer test tables.

The software implementation was modified to provide greater insight into the operations of the ciphers. Intermediate results were printed during the computations to determine the point at which our implementations slipped away from the specifications.

This testing procedure allowed us to locate flaws in our implementation. Some of these were simple errors that had slipped by during our review of the code. Others were due to a misunderstanding of some subtleties of the algorithm. We can say, without any exaggeration, that it would have been nearly impossible to find all the flaws of our implementation without using this method.

5.2. Validation

Our implementation of the Twofish cipher was validated using know-answer tests tables supplied by the authors of the algorithm with their submission to the AES contest. These test vectors have the particularity of being specified in 'big endian' notation. The algorithm actually operates on 'little endian' vectors and the c implementation initially reverses the byte order before processing the data.

Each simulation took an average of 20 minutes for encrypting or decrypting. Each simulation exercises the cipher through 16 rounds of encryption or decryption and the chances to get the right result with a flaw in the system are thus very low. For those reasons, only two complete simulations were sufficient to convince us of the correctness of our circuit.

The first tests give us a limited coverage of the functions of the cipher. Both the key and the data were set to zero. The data was encrypted and decrypted producing the correct results:

Key = 00000000000000000000000000000000
Data = 00000000000000000000000000000000
Encrypt = 5AC3E82A2FECBFB6322C12F65C9F589F
Decrypt = 00000000000000000000000000000000

The second test is worth examining more carefully. The key was first set to 5AC3E82A2FECBFB6322C12F65C9F589F by assigning the'usr_ld_key' signal to 1. As we can see the controller then goes to the load_keya' state and then back to 'my_idle'.

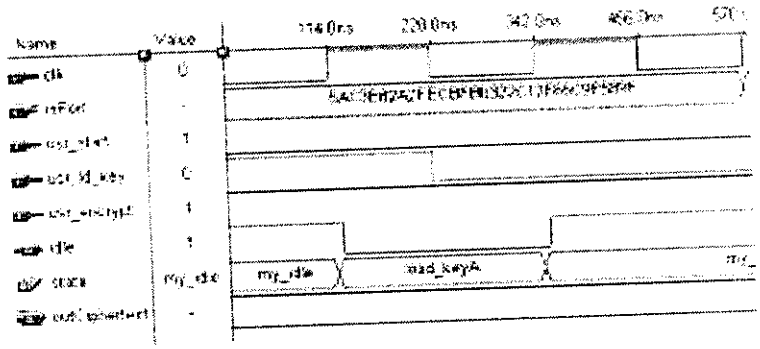


Figure 5.2.1: Key Setup for Encryption

The data is then set to 19549F786B08CB869EC3B1E716DB91D4 with the 'usr_start' signal and the controller leaves the 'my_idle' state.

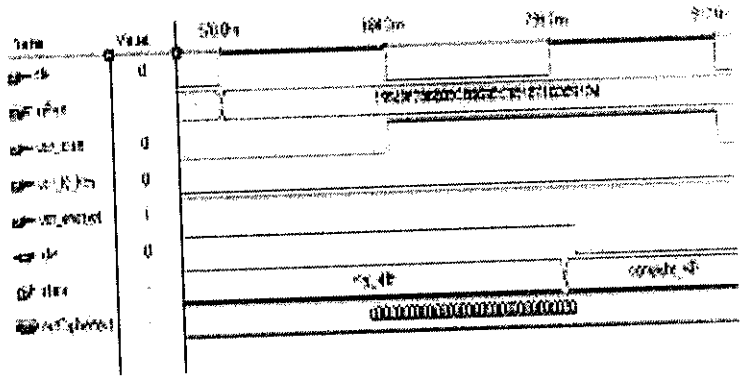


Figure 5.2.2: Data Setup for Encryption

The system then computes and uses the four input Whitening keys.

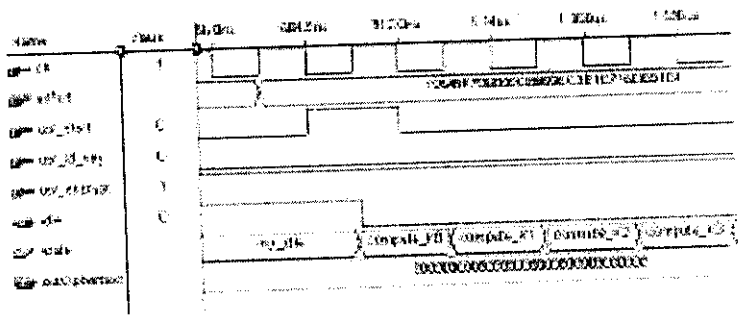


Figure 5.2.3: Input Whitening

The controller then executes sixteen rounds of encryption.

As we can see from the Figure 5.2.3, each round consists of four states. The even key is computed (compute_k2r_8), followed by odd key computation (compute_k2r_9). After this state the two keys are ready and stored in the key registers. The even and odd results are then computed (compute_even and compute_odd). After this state all the results necessary to finish the round are ready and the cipher moves to the next round.

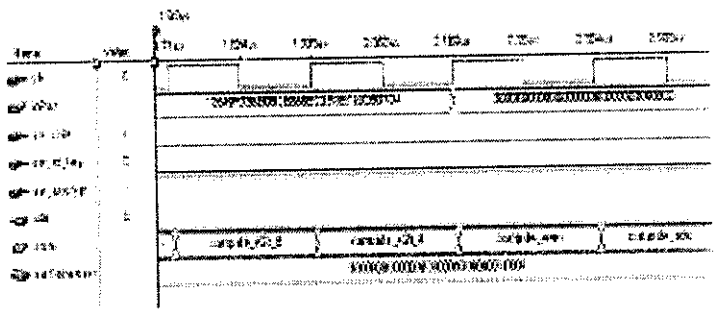


Figure 5.2.4: Round Execution

Once the 16 rounds are finished, the four output whitening sub-keys are computed and used.

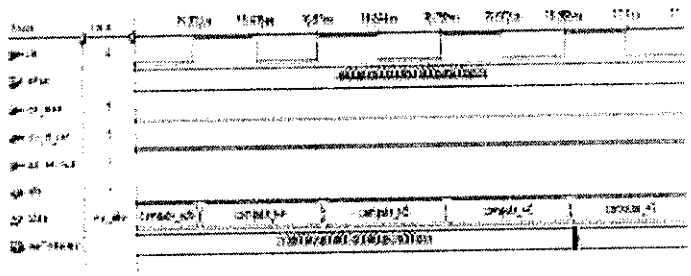


Figure 5.2.5: Output Whitening

The encrypted data is ready at the output after 16.7μs. The result is as stated in the know-answer test tables:

C3FB20BAA3C3AA8F851117DE09989F01.

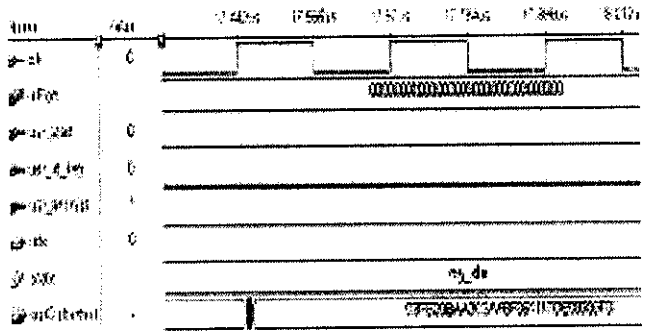


Figure 5.2.6: Encrypted Data

The decryption process is almost identical; but in this case we set the data to the encrypted data. Also notice that 'usr_encrypt' is set to low while 'usr_start' is raised.

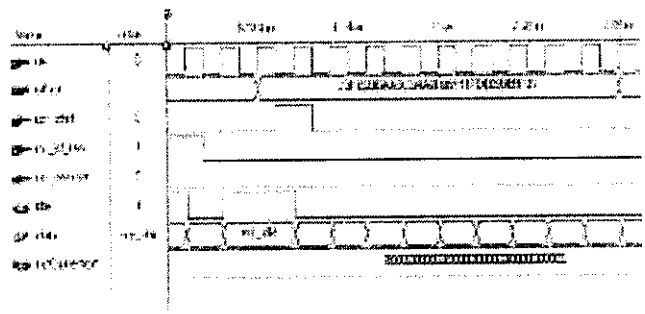


Figure 5.2.7: Setup for Decryption

Without going into details, 16.7μs later the data is decrypted and as you can see from Figure 5.2.7 it corresponds to the original 128-bit that was encrypted. Our encryption and decryption process were thus successful.

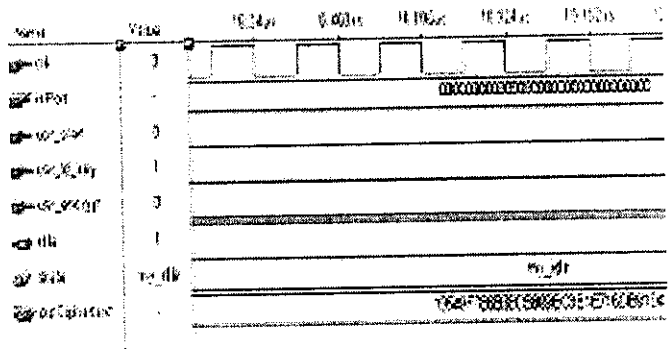


Figure 5.2.8: Decrypted Data

5.3. Performance

The first aspect of performance to look at was the number of gates used by our design. The latter was obtained by multiplying the percentage of the block cells used by the total number of gates available on the chip on which our design fit. The design fit into a flex10k70g with 82% logic cell use thus resulting in a gate count of approximately 57400.

Note that the large number of IO pins used by the design could play a large role in the size of the chip. Serial input could be a solution to reduce this factor. Next, having run a registered performance on our design, the clock rate was determined to be 4.4 MHz. as each encryption or decryption operation takes 72 clock cycles, the latency of the device is 16.4 µs per block. Because no other implementations were available, results comparison was rendered impossible. But based on the expected values given by the designer of the algorithm, we can assert that the performance of our design is acceptable. Moreover, given some more time, the design could still be improved to make it more modular and optimize its space requirement further than what was done.

5.3.1 Advantages

- The TWOFISH is 128 –bit cipher that supports key of length 128,192 or 256 bits.
- Symmetric architecture is used for both Encryption and Decryption methods, this reduces the hardware requirements.
- As length of the key increases, the security of the information is increased.
- It is very strong key when compared to other algorithms, so that the security cannot be broken easily.
- It can be mapped efficiently to hardware devices such as FPGA and Smart Cards.

SIMULATION RESULTS

6.1 Simulation

- Simulation is the process of testing the design with user generated simulators.
- Simulation can be done at the various stages of the design cycle.
    - Functional (Behavioral code)
    - Gate level (Netlist)
    - Timing (Place & Routed Netlist)

In our project the simulations are carried out in the software environment called Modelsim PE 5.4e. The simulation results for each module are found.

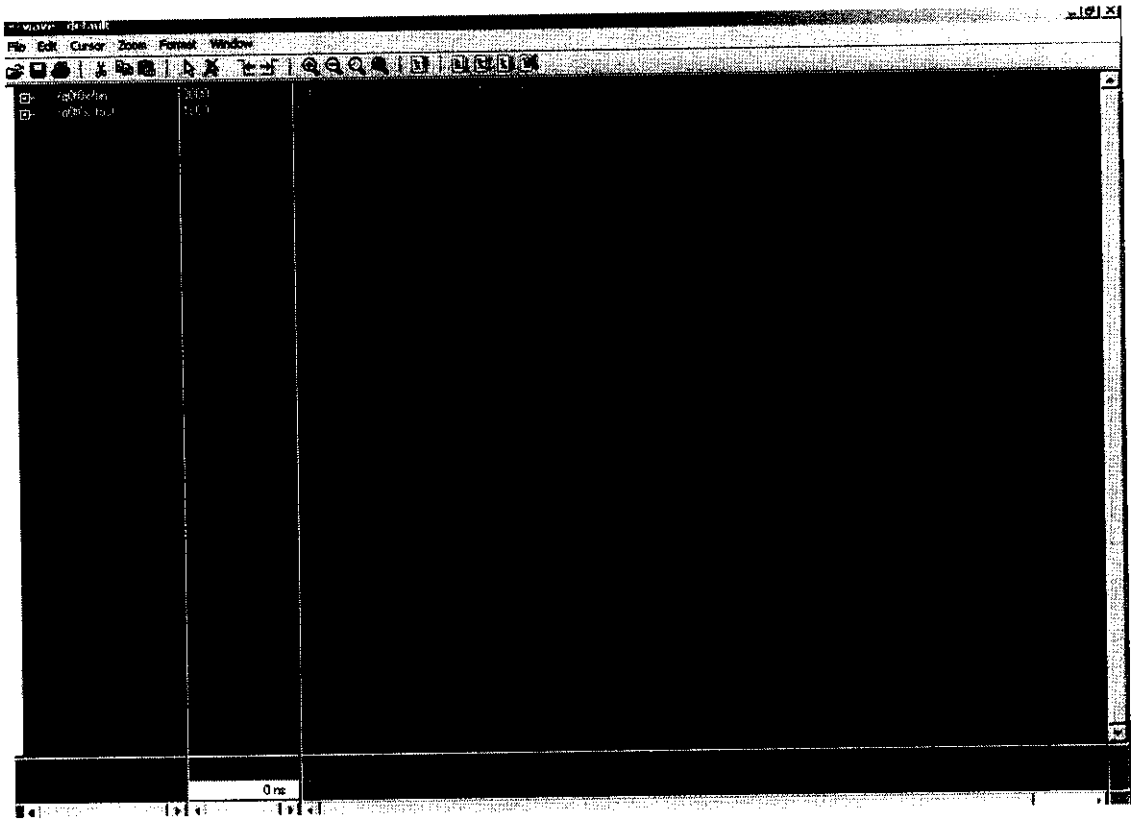


Figure 6.1: Simulation result for Q-Permutation for time t_0

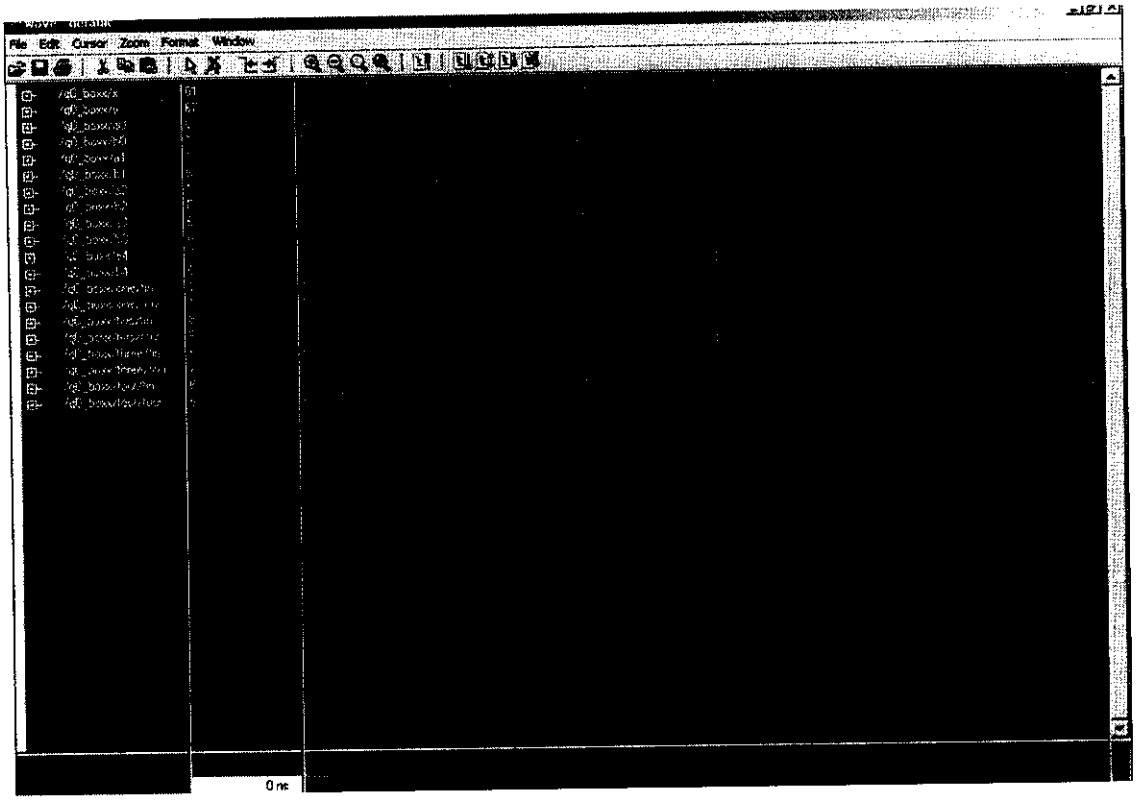


Figure 6.2: Simulation result for Q-Permutation for time t_1

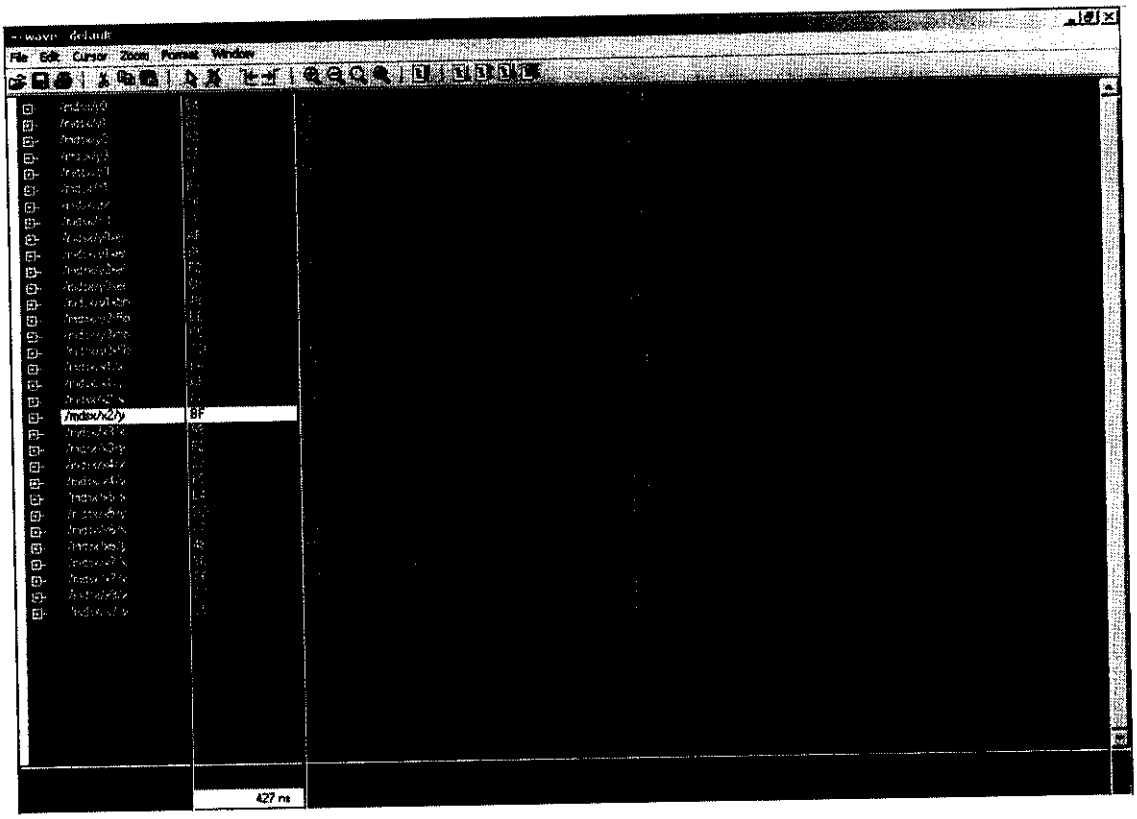


Figure 6.3:Simulation result for MDS matrix

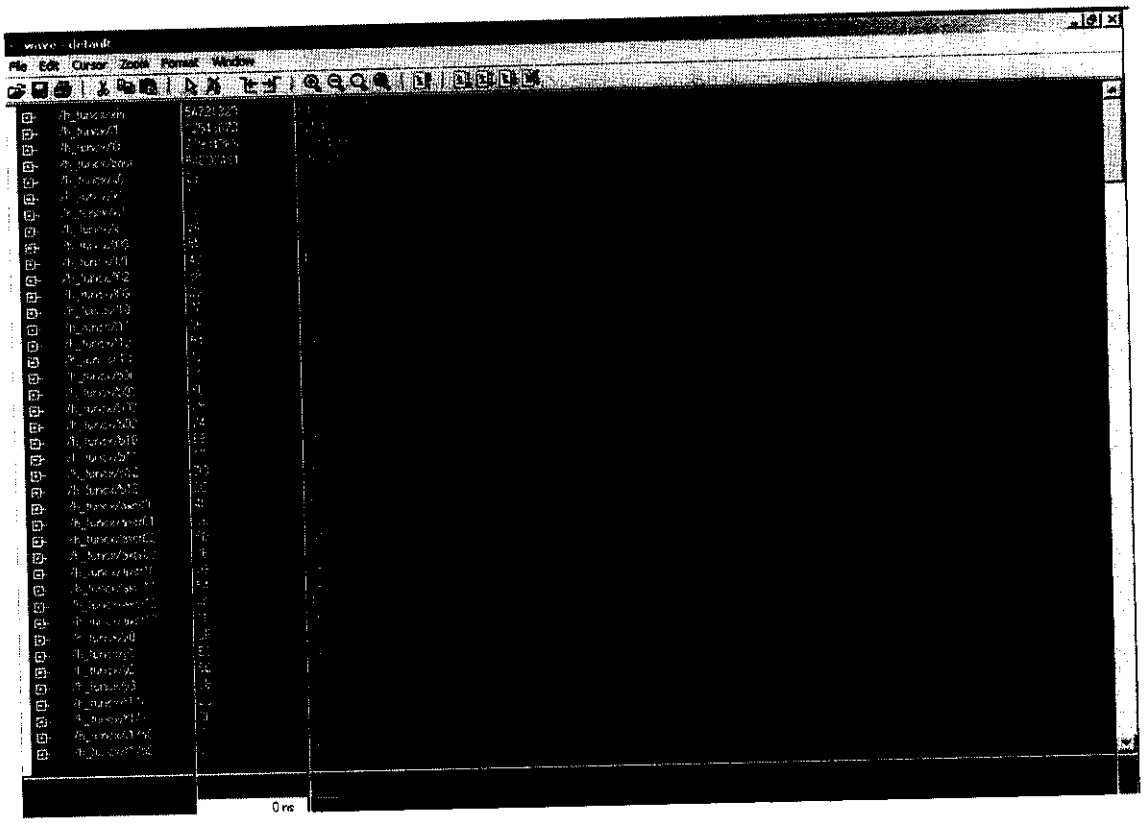


Figure 6.4:Simulation result for H-Function

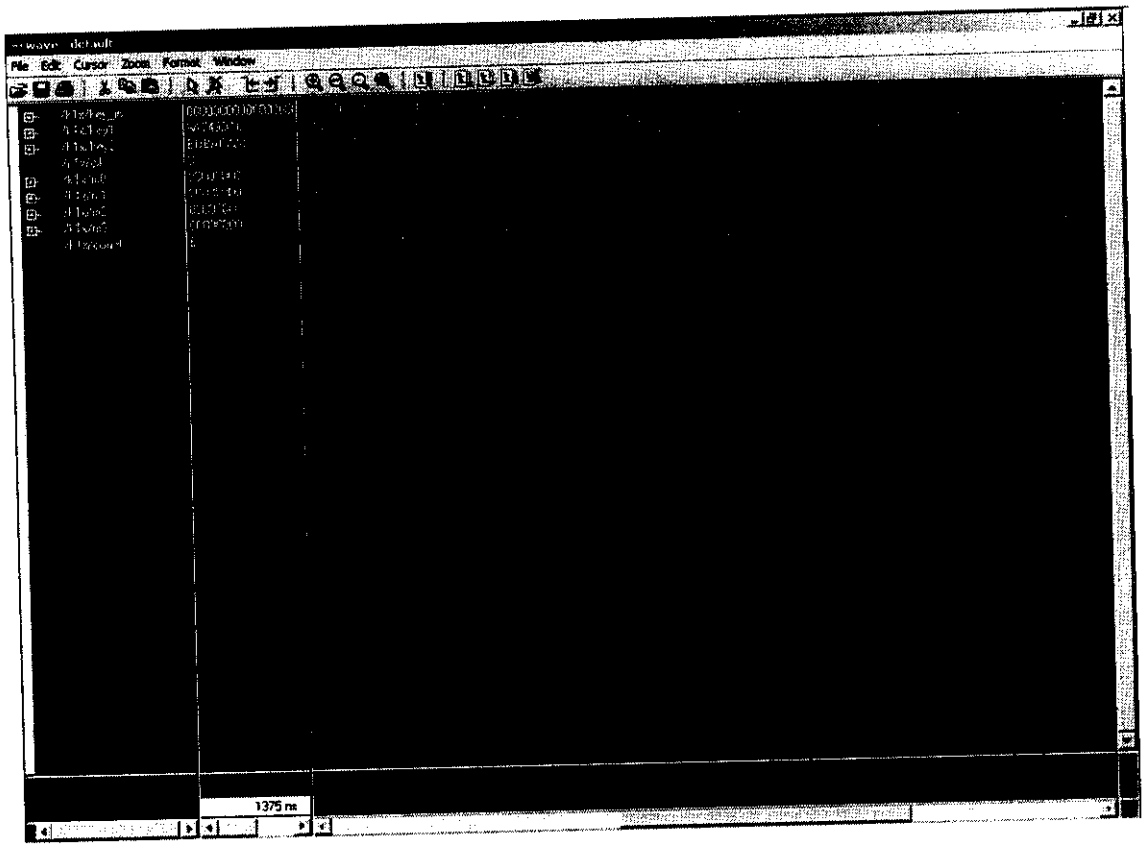


Figure 6.5:Simulation result for Key Generation

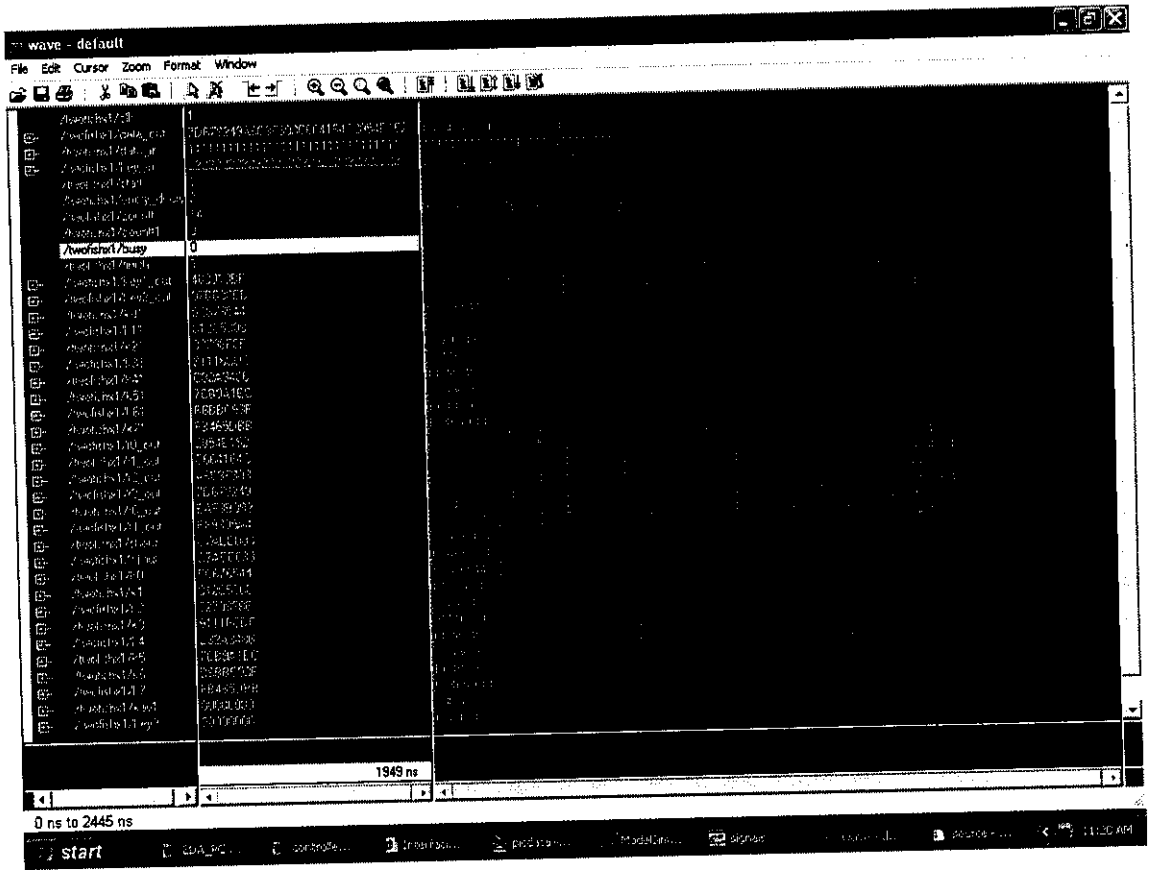


Figure 6.6:Simulation result for Encryption

Figure 6.7:Simulation result for Decryption

7.1 Synthesis results

Synthesis is the process of converting the VHDL code into technology specific netlist. we can also specify certain user constraints during the synthesis, like placement, timing etc. The synthesis results for each module are as follows.

7.1.1 Synthesis Report for Q-Permutation

Device utilization summary:

Selected Device : 2s100tq144-5

Number of Slices : 17 out of 1200 1%

Number of 4 input LUTs:32 out of 2400 1%

Number of bonded IOBs:16 out of 96 16%

Timing Summary:

Maximum combinational path delay: 16.403ns

The corresponding synthesized RTL schematic output is shown in the Figure: 7.1.1

7.1.2 Synthesis Report for MDS

Device utilization summary:

Selected Device : 2s100tq144-5

Number of Slices : 65 out of 1200 5%

Number of 4 input LUTs :114 out of 2400 4%

Number of bonded IOBs :64 out of 96 66%

Timing Summary:

Maximum combinational path delay: 17.896ns

The corresponding synthesized RTL schematic output is shown in the Figure: 7.1.2

7.1.3 Synthesis Report for H-Function

Device utilization summary:

Selected Device	: 2vp2fg456-6
Number of Slices	: 263 out of 1408 18%
Number of 4 input LUTs	:483 out of 2816 17%
Number of bonded IOBs	:128 out of 156 82%

Timing Summary:

Maximum combinational path delay: 19.119ns

The corresponding synthesized RTL schematic output is shown in the figure: 7.1.3

7.1.4 Synthesis Report for Key Generation

Device utilization summary:

Selected Device	: 2vp2ff672-6
Number of Slices	:530 out of 1408 37%
Number of Slice Flip Flops	:17 out of 2816 0%
Number of 4 input LUTs	: 972 out of 2816 34%
Number of bonded IOBs	:192 out of 204 94%
Number of GCLKs	:1 out of 16 6%

Timing Summary:

Minimum period: 1.932ns (Maximum Frequency: 517.598MHz)

Maximum output required time after clock: 25.179ns

Maximum combinational path delay: 21.487ns

The corresponding synthesized RTL schematic output is shown in the figure: 7.1.4

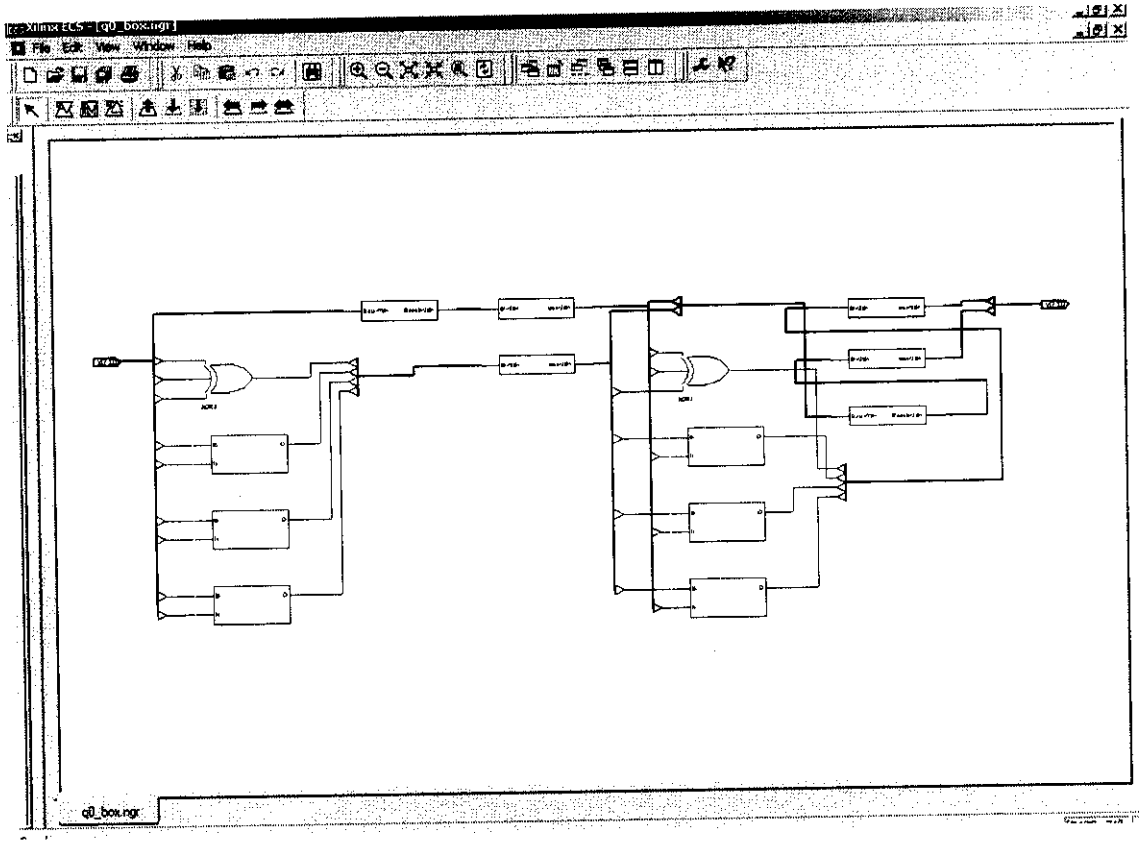


Figure 7.1.1: RTL Schematic for Q-Permutation

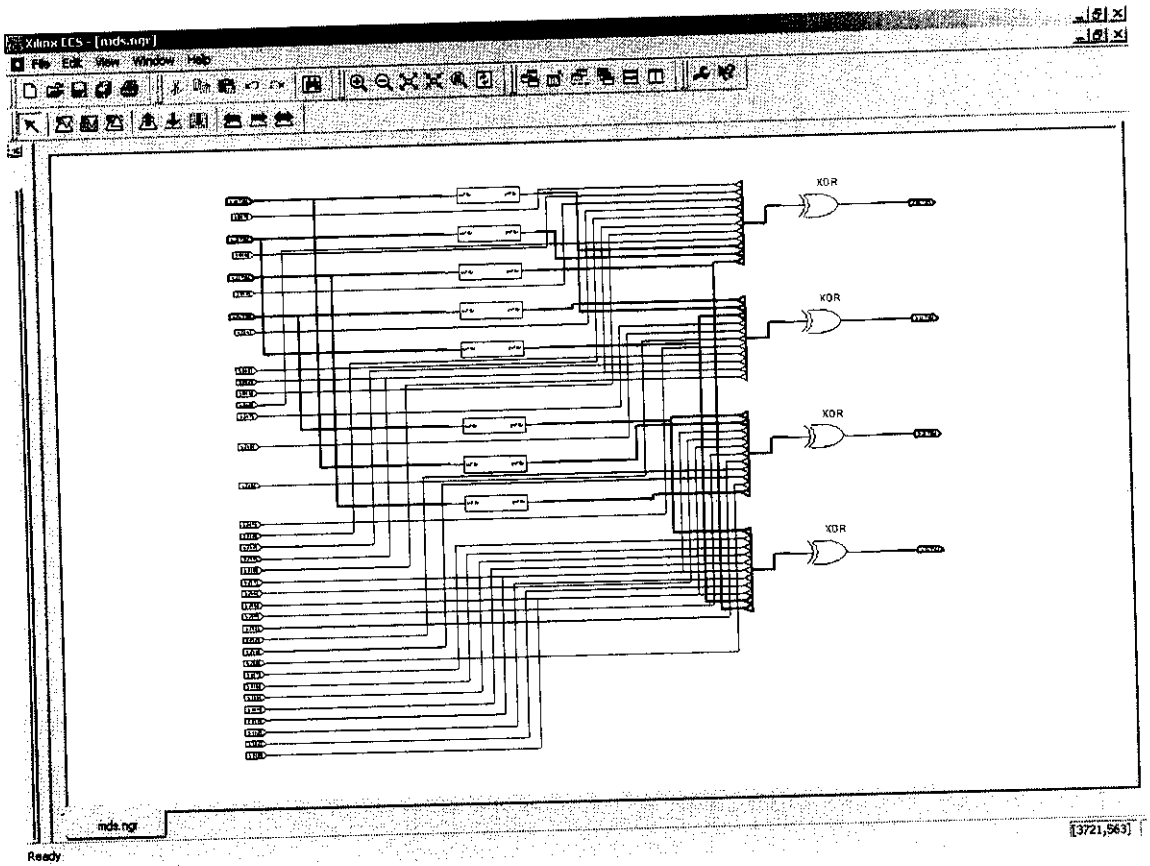


Figure 7.1.2: RTL Schematic for MDS

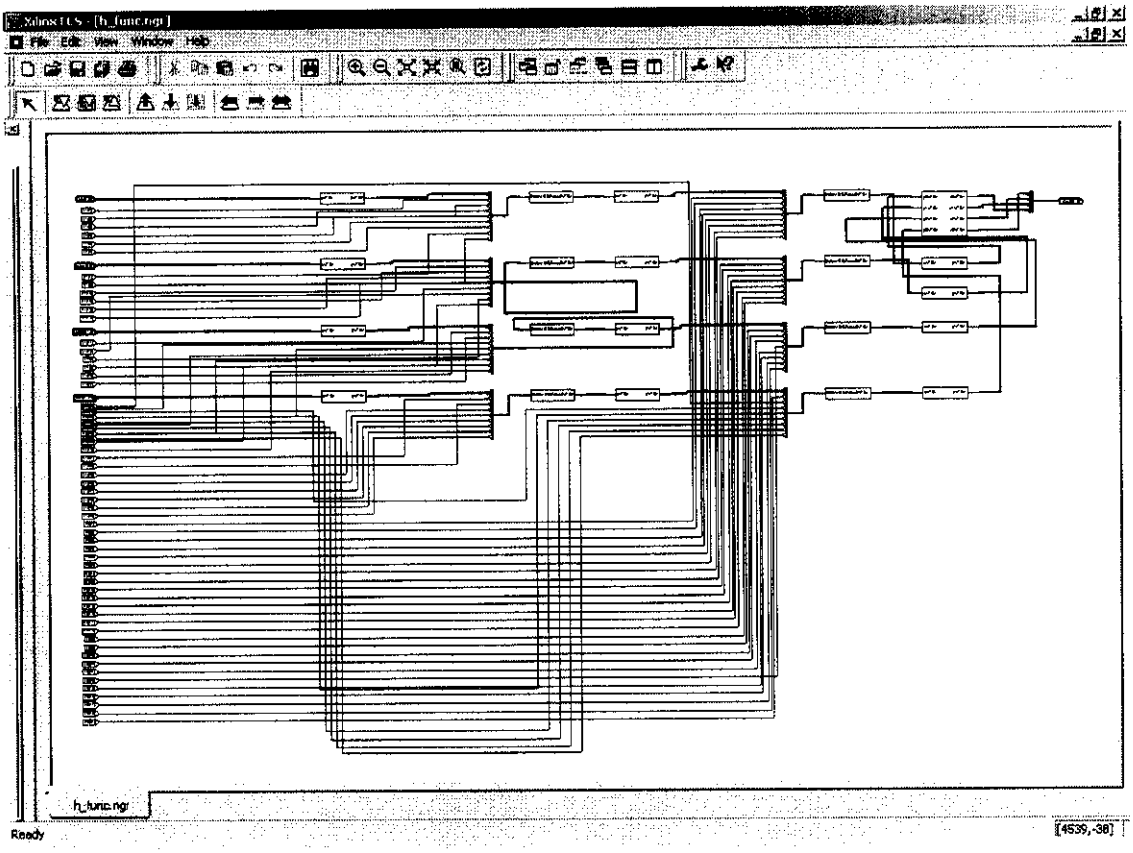


Figure 7.1.3: RTL Schematic for H-Function

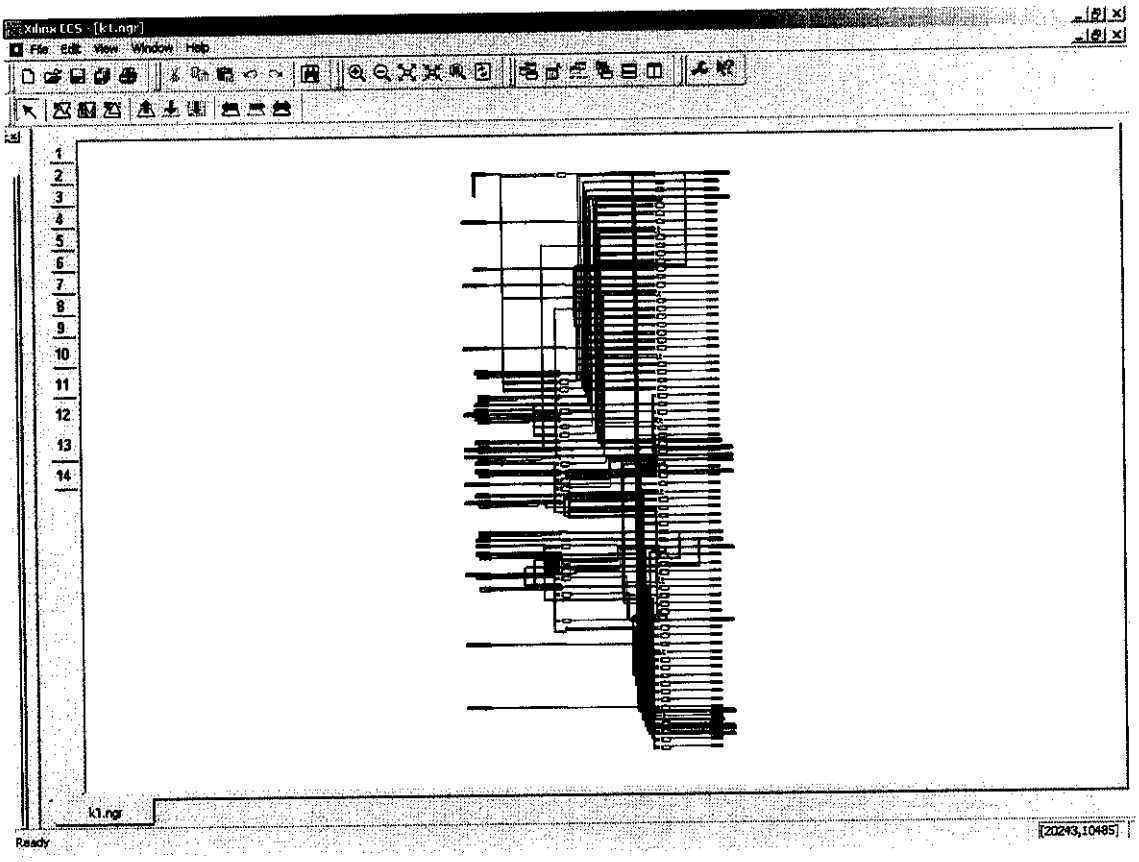


Figure 7.1.4: RTL Schematic for Key Generation

7.2 Floor planning

Floor planning is a mapping between logical description (netlist) and physical description (floor plan).

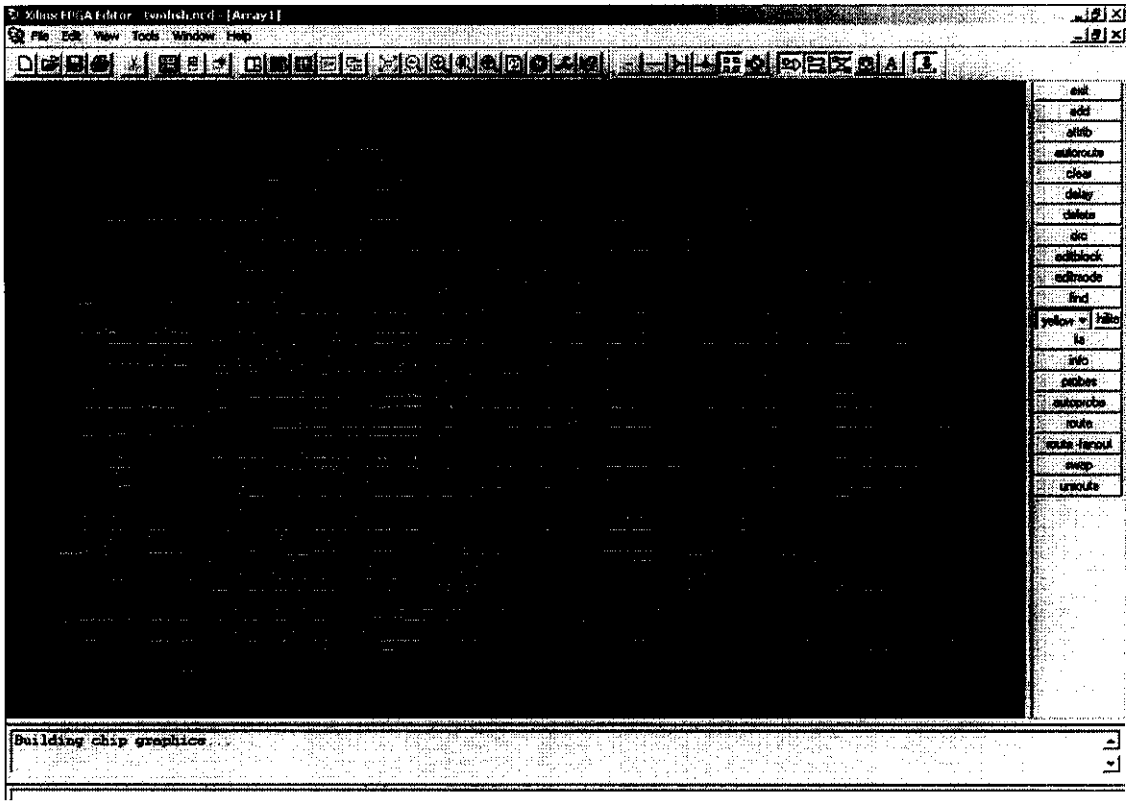
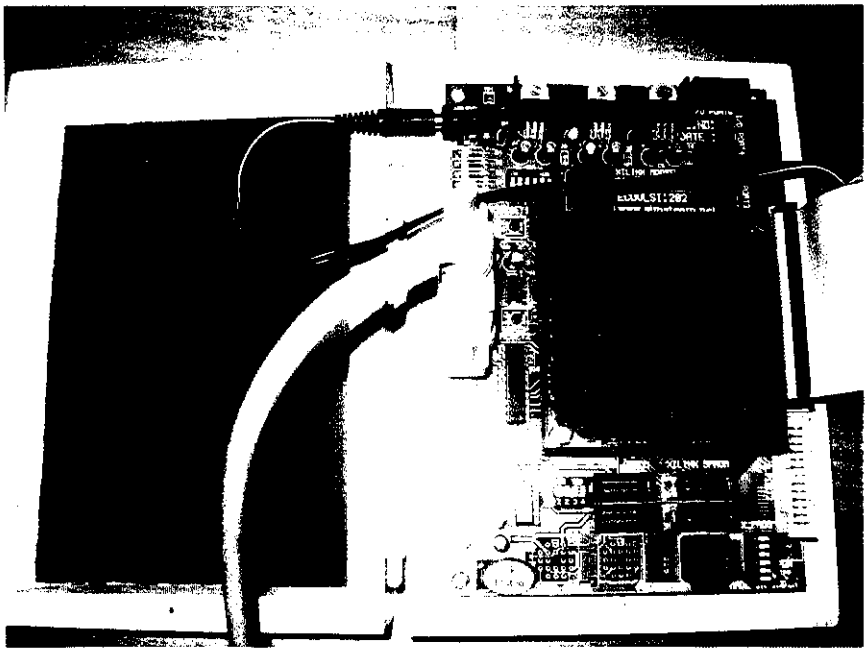
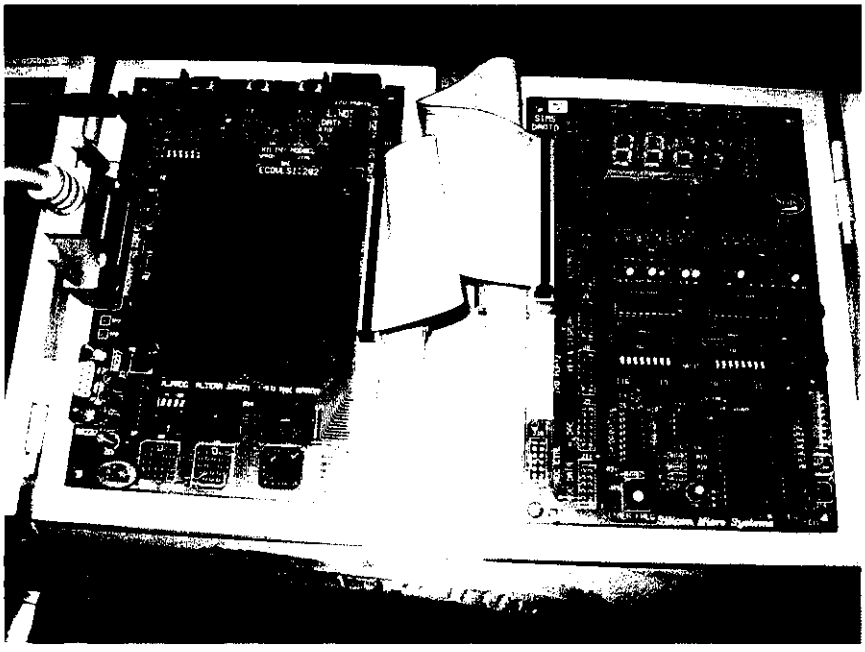


Figure 7.2.1: FPGA Floor plan





CONCLUSION

CONCLUSION

As claimed by its designer, Twofish turned out to be a very flexible deign. from the subkey generation to the data encryption, Twofish offered a wide variety of design possibilities, which is actually one of its main advantages over its competitors since AES must be a general rather than a specialized algorithm.

The implementation of the cipher in conformance with the specifications was challenging but the quality of the documentation and the available resources allowed us to reach our goal.

The upcoming year will reveal the algorithm chosen as the advanced encryption standard by NIST. This announcement will not go unnoticed and a lot of products will be upgraded to be buzzword-compliant. You will no doubt hear more about AES in the future and we hope that you will hear more about Twofish as well.

REFERENCES

REFERENCES

[1] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson, "Twofish: A 128-Bit Block Cipher",www.Counterpane.Com

[2] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, And Niels Ferguson, "The Twofish Encryption Algorithm", Wiley, 1999

[3] Pawel Chodowiec, Kris Gaj, "Implementation Of The Twofish Cipher Using Fpga Devices".