Implementation of the Twofish Cipher Using FPGA Devices

Pawel Chodowiec, Kris Gaj

pchodowi@gmu.edu, kgaj@gmu.edu

Technical Report
Electrical and Computer Engineering
George Mason University

July 1999

Implementation of the Twofish Cipher Using FPGA Devices

George Mason University

Pawel Chodowiec, Kris Gaj

Table of Contents

1 Short description of the Twofish cipher

Twofish is a 128-bit block cipher. It can work with variable key lengths: 128, 192, or 256-bits. In this report, *only a version with 128-bit key length will be discussed*. Twofish consists of 16 rounds built similarly to the Feistel network structure. Fig. 1 shows an overview the cipher structure. The only exceptions from the pure Feistel network are two fixed rotations by one bit, performed together with the XOR operations on outputs of the F-function.

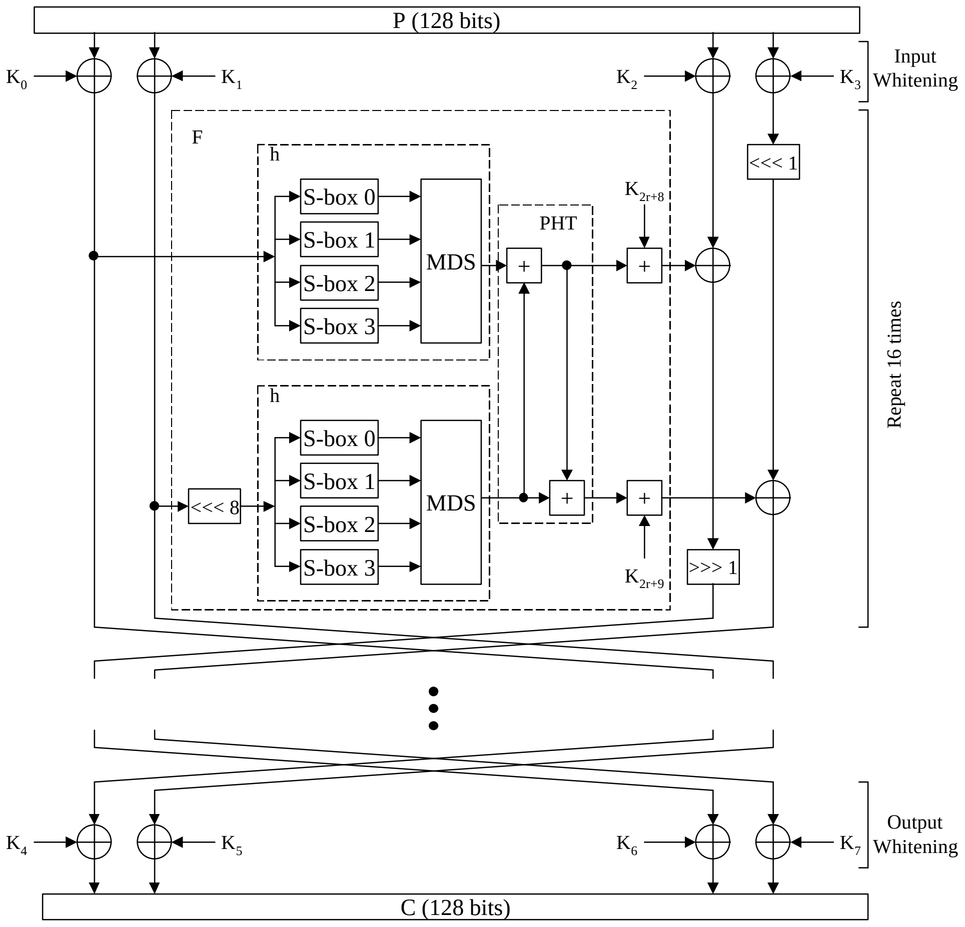


Figure 1 - Twofish overview (encryption).

Additionally, input and output data are XOR-ed with eight subkeys K0…K7. These XOR operations are called input and output whitening. The F-function consists of five kinds of component operations: fixed left rotation by 8 bits, key dependent S-boxes, Maximum Distance Separable (MDS) matrices, Pseudo-Hadamard Transform (PHT), and two subkey additions modulo 2^{32}.

There are four kinds of key dependent S-boxes. Four different S-boxes together with the MDS matrix form an h-function. This h-function appears two times in the cipher structure, which causes significant redundancy.

Key dependent S-boxes are something new in a cipher design. In majority of known ciphers, S-boxes are used as a non-linear fixed substitution operation. In Twofish, each S-box consists of three 8-by-8-bit fixed permutations chosen from a set of two possible permutations, q_0 and q_1. The structure of all S-boxes is shown in Fig. 2. Between these three permutations, XOR operations are performed with subkeys S_0, S_1 (refer to the key schedule description, at the end of this section). These subkeys are computed only once for a particular global key, and stay fixed during the entire encryption and decryption process.

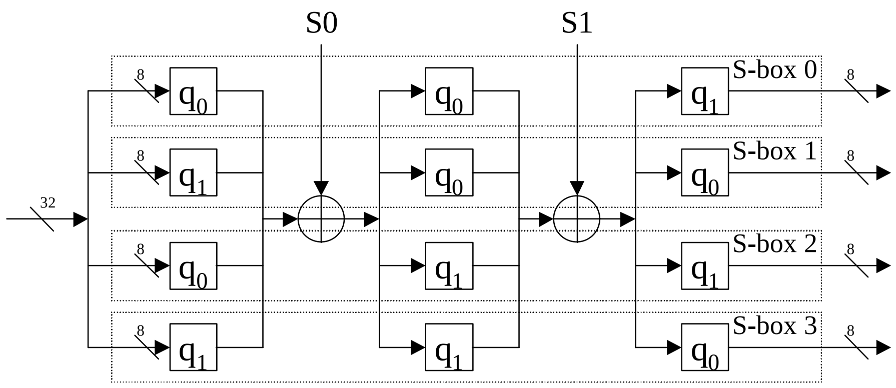
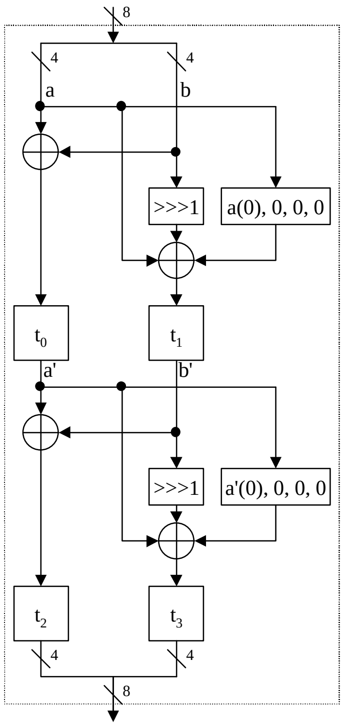


Figure 2 - S-boxes.

Although each q-permutation represents a fixed function, it is also described by a regular structure shown in Fig. 3. Main components of the q-permutations are 4-by-4-bit fixed S-boxes $t_0 \ldots t_3$. Both permutations q_0 and q_1 have the same internal structure, and differ only in the contents of the S-boxes $t_0 \ldots t_3$.



		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

Figure 3 - Permutation q.

Another function used in Twofish is a 4-by-4-byte MDS matrix. The transformation performed by this matrix is described by the formula:

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

where: $y_3 \ldots y_0$ are consecutive bytes of the input 32-bit word (y_3 is the most significant byte), and $z_3 \ldots z_0$ form the output word.

This matrix[6] multiplies a 32-bit input value by 8-bit constants, with all multiplications performed (byte by byte) in the Galois field GF (2^8). The primitive polynomial is $x^8 + x^6 + x^5 + x^3 + 1$. Only three different multiplications are used effectively in the MDS matrix, namely multiplication

- by $5B_{16} = 0101\ 1011_2$ (represented in $GF(2^8)$ by a polynomial $x^6 + x^4 + x^3 + x + 1$),
- by $EF_{16} = 1110\ 1111_2$ ($x^7 + x^6 + x^5 + x^3 + x^2 + x + 1$), and
- by $01_{16} = 0000\ 0001_2$ (equivalent element in $GF(2^8)$ is just 1) - obviously the result is equal to the input value.

Finally, the PHT transform is a simple function that consists of two additions modulo 2^{32}, as shown in Fig. 1. Results of the PHT transform can be described by the following equations:

$a' = a + b$

$b' = a + 2*b$

Both additions are de facto independent and can be performed simultaneously.

The nice feature of the Twofish algorithm is that, after little modifications, we can perform encryption and decryption using exactly the same structure. It is very valuable for hardware implementations. Decryption requires applying subkeys in the reverse order and making a little modification to the main cipher structure as shown in Fig. 4.

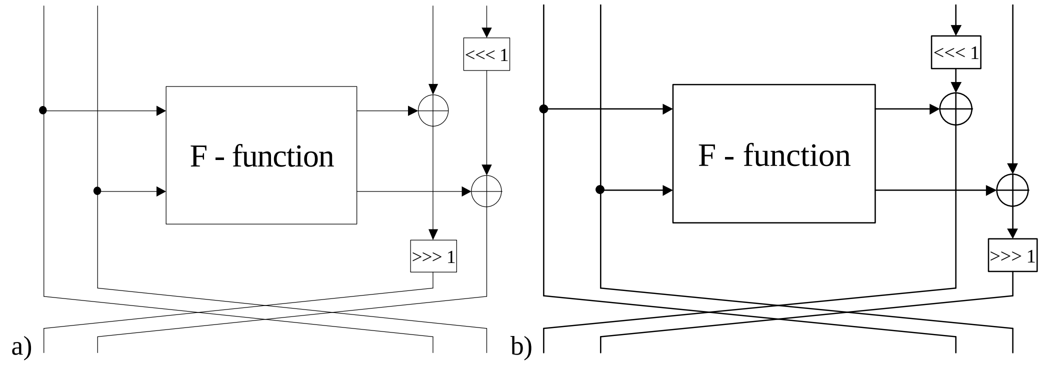


Figure 4 - Differences between the encryption and decryption a) encryption b) decryption.

Key schedule seems to be another strong side of the Twofish algorithm. There are two different sets of subkeys: S and K.

Two subkeys S_0 and S_1 are fixed during the entire encryption and decryption process. Both of them are obtained as a result of multiplying an appropriate part of a global key (128-bit wide, called M) by RS matrix. This matrix also performs multiplication in the Galois field GF (2^8), but the primitive polynomial is different then for this MDS matrix: $x^8 + x^6 + x^3 + x^2 + 1$. The algorithm used to compute S keys is given by:

$$\begin{pmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{pmatrix} = \begin{pmatrix} 01 & A4 & 55 & 87 & 5A & 58 & DB & 9E \\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5 \\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19 \\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03 \end{pmatrix} \bullet \begin{pmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{pmatrix}$$

where:

- index *i* is from the range 0...1, and indicates the appropriate key S_0 or S_1,
- $s_{i,3}...s_{i,0}$ are 8-bit parts of the 32-bit key S_i ($s_{i,3}$ is the most significant byte),
- $m_{8i+7}...m_{8i}$ are 8-bit parts of the 128-bit user supplied key M.

Remaining set of subkeys K is computed in a structure very similar to that used for encryption. The only difference is that there is no key addition after the PHT transform, and that the fixed left rotation by 8 bits is performed after instead of before the second h-function (compare Fig. 1 and Fig. 5). This feature enables computing subkeys K using the same piece of hardware as the one used for encryption. The global key M is used to parameterize all key dependent S-boxes. All subkeys K are independent of each other, and are computed on the basis of their index value. It means, that they can be computed on the fly in both directions: for encryption and decryption.

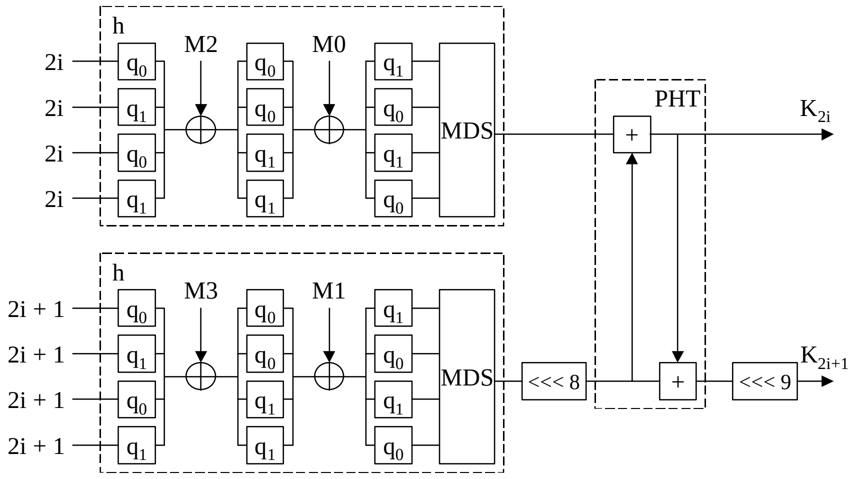


Figure 5 - Generation of subkeys K.

Keys $M_3 \ldots M_0$, shown in Fig. 5, are derived directly from the main 128-bit key. They are just 32-bit parts of the main key, where M_3 corresponds to the most significant 32 bits, and M_0 to the least significant bits of M. These subkeys are used to customize S-boxes, and stay fixed for all rounds.

The variable *i* is from range 0…19, and is used to generate a set of corresponding subkeys $K_0 \ldots K_{39}$.

2 Analysis of the Twofish cipher main components

The analysis presented in this section concerns the ability of implementing Twofish using Xilinx FPGA devices.

In the following section, all basic functions used in Twofish, and the way of implementing them in Xilinx FPGAs will be discussed. These basic functions include:

- fixed rotations
- additions modulo 2 (bit by bit)
- additions modulo 2^{32} on 32-bit long words
- key dependent S-boxes
- MDS matrices
- PHT transforms

2.1 Fixed rotations

This kind of operations is easily realizable in hardware. They do not even require any additional resources. The only thing we have to do is to reorder interconnections between logic cells.

2.2 Additions modulo 2 (XOR)

This is a second very inexpensive and fast operation. Each output bit depends on value of only two input bits. It is obvious that such operation can be done using only a single F or G lookup table in Xilinx FPGA. Therefore, one CLB can be used to compute this function on two bits. For example, XOR on two 32-bit long words requires only 16 logic cells. Besides, each XOR uses only two inputs, but there are four inputs available in each lookup table. As a result, in many cases, XOR can be combined with other operations and implemented in the same cell.

2.3 Additions modulo 2^{32} on 32-bit long words

Although the addition modulo 2^{32} is not as simple operation as XOR, it can still be realized using comparable amount of hardware. The most convenient way to perform this operation is to compute the sum position by position, and use carry chain to propagate intermediate results from the least significant to the most significant position. Such structure is shown in Fig. 6. It is obvious that the time of signal propagation through the entire carry chain limits the speed of addition. The longer words are being added, the longer it takes to compute the result. Fortunately, Xilinx logic cells contain dedicated fast connection between neighboring cells designated specifically for carry chains. Xilinx Co. claims that their carry chain is so fast, that all known methods of speeding up addition have marginal effect for words shorter or equal to 32 bits [4]. Using this structure permits implementing one 32-bit long adder using only 16 logic cells.

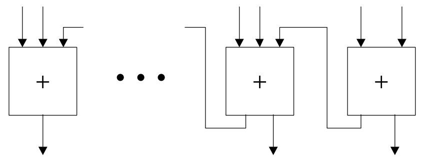


Figure 6 - Carry chain.

2.4 Key dependent S-boxes

Basically, there are two possible ways to implement S-boxes [1]:

- as a 256-byte RAM, or
- as a direct logic implementation.

First type of design would require eight 256-byte memories. Xilinx FPGA devices do not contain any dedicated RAM and thus a 256-byte RAM would have to be implemented using logic cells. Each CLB can be configured as 16 x 2 bit RAM memory [4]. Therefore, 512 (8 * (256 x 8) / (16 x 2)) logic cells are needed to implement all eight S-boxes, which is a large number. Additionally, the access time to this memory would be long. However, such solution can still be taken into account in case of the Altera FPGA devices, where additional RAM elements (EAB) exist. Additional drawback of this method is a long time of key exchange, because the contents of all memory cells have to be precomputed, based on the current key.

In a second type of design, we have a degree of freedom. First of all, each q-permutation can be implemented as a 256-byte ROM, or as a logic function (see Fig. 3). Although, there are only two different permutations, they are used simultaneously in eight S-boxes. It is obvious, that this kind of implementation also requires too many logic cells in Xilinx FPGA, and it will not work faster than the direct logic implementation. Hence, let us consider only the direct logic implementation. As shown in Fig. 3, each q-permutation consists of four 4-by-4-bit S-boxes t_0...t_3, and some additional logic. Since each of these little S-boxes forms only a four-bit input function, it can be easily realized using only two parallel CLBs. In other words, the delay caused by one S-box t is equal to the delay of a single CLB. Remaining operations in the permutation q are easy to implement using two levels of logic cells. In summary, each q-permutation can be implemented using 16 CLBs arranged in a four level structure.

Taking into account our calculations for q-permutations, each S-box S_0...S_3 should take 56 logic cells (3 * 16 CLBs for each q-permutation + 8 CLBs for XOR with fixed keys S_0 and S_1). However, its structure consists of 14 levels of logic cells (3 q-perm. * 4 levels per each permutation + 2 XORs * 1 level per each XOR). It means that computation can consume a lot of time. Fortunately, each S-box can be easily divided into several logic layers using registers, so that we can still use fast clock rate.

2.5 The Maximum Distance Separable matrix

Implementation of the MDS matrix can seem very difficult, but closer analysis of operations performed in this matrix leads us to a different conclusion. As it was shown in section 1, there are only two different multiplications that require to be implemented. In case of multiplication by $5B_{16}$, every output depends on at most four inputs. Therefore, this multiplication consumes only four parallel CLBs – see Fig. 7a.

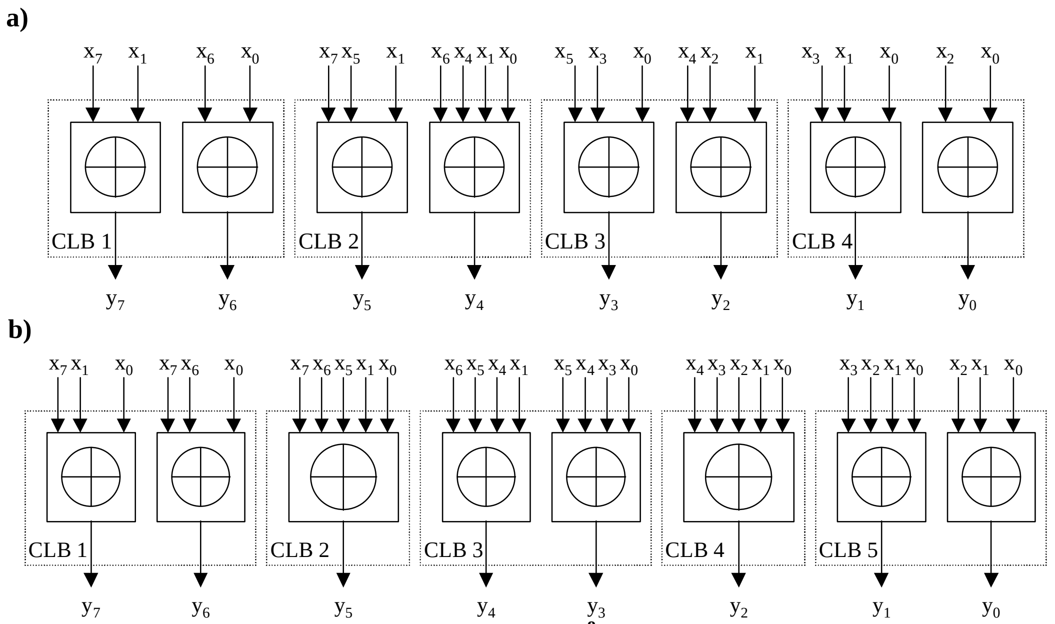


Figure 7 – Multiplication by a constant in the GF(2^8)

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

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

The multiplication by EF_{16} contains two outputs depending on five input bits. These outputs require an entire CLB each, therefore the entire multiplication will take five parallel CLBs. Nevertheless, any multiplication in GF(2^8) can be implemented using up to eight parallel CLBs. As a result, the time of a single multiplication is equal to the delay of one CLB.

The results of all multiplications in each row of the MDS matrix are finally XOR-ed bit by bit. Such operation needs only four CLBs for each row. To compute one byte of the result, it is necessary to perform only three multiplications. Thus, we need only 17-18 logic cells (4-5 CLBs per multiplication * 3 multiplications + 4 CLBs for XOR). All CLBs are organized in a two level structure, allowing computing results in a very high speed. The entire MDS matrix implementation can fit in 52 cells. This number of cells results from the fact that there are two identical multiplication factors in each column of the MDS matrix (see section 1), and the corresponding multiplication operation need to be performed only once. 52 = (4 columns * (4 + 5 CLBs required for both multiplications in the column) + 16 CLBs for final addition).

2.6 The Pseudo-Hadamard Transform

The PHT transform is composed of two additions. It was shown in section 1 that both additions can be performed in parallel. The only additional operation is one left shift by one bit, as shown in Fig. 8. Obviously, both additions are implemented in the same way as ordinary addition modulo 2^{32}, discussed in section 2.3.

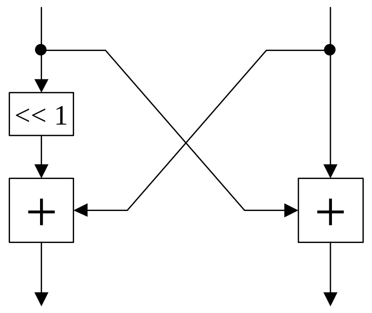


Figure 8 - The PHT after changes.

3 The structure of the Twofish cipher

The Twofish cipher can be implemented in many ways. Each of these ways can be better or worse depending on particular requirements. In the following section, we would like to discuss some of the possible alternative methods of implementing Twofish.

3.1 Implementation of encryption and decryption

First of all we can choose between implementing the cipher using purely combinational design (not counting the main register required to store a final result of each round), and using sequential design with registers dividing the combinational logic into smaller but faster blocks.

First kind of design is probably the fastest, but also hard to optimize for minimum area, because function sharing is difficult to apply. Additionally, a combinational design may work only with a very low clock frequency, and its synthesis may be significantly longer. The speed of the sequential design is lower, because we are not able to divide the circuit into exactly equal blocks, and this way we lose precious nanoseconds. On the other hand, in many cases, a sequential design allows time sharing of the same part of the circuit between various functions, which can significantly reduce the amount of area consumed by this design. Moreover, in the ECB mode of the block cipher (as well as in more complex interleaved modes), the sequential design may increase the encryption speed directly proportionally to the number of pipelined stages.

Lets take into consideration the amount of area consumed by the Twofish cipher. The minimal implementation must consist of at least one round of encryption and decryption, and a memory containing all subkeys (we assume that implementation of such memory takes less space than any additional circuit designated specifically for subkeys generation). In the simplest situation, the implementation of one round (basically only F function) can be directly based on the cipher structure. In such case, we need to use eight S-boxes, two MDS matrices, one PHT transform and two 32-bit subkey adders. It would take about 550 logic cells: 2*(4 S-boxes * 3 q-permutations * 16 CLBs for each q-permutation + 52 CLBs for MDS) + 32 CLBs for PHT + 32 CLBs for subkey adders. The same amount of area has to be reserved in case of purely combinational logic and in case of sequential logic, because all registers cost no additional hardware in FPGAs. But looking at the cipher structure we can easily find a significant redundancy caused by two h-functions (each consists of four S-boxes and one MDS matrix). Each h-function consumes about 250 CLBs. We can modify this structure a little, and get rid of redundancy at the cost of only one clock cycle. It is presented in Fig. 9.

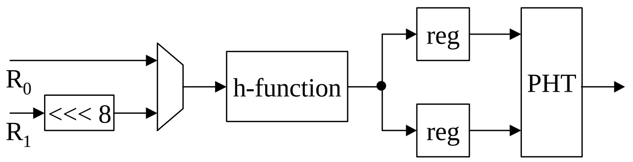


Figure 9 - Modified structure of the g-function.

Losing one clock cycle does not mean that our speed will decrease dramatically. For example, we can divide our cipher into several layers of combinational logic separated by registers, and run the circuit with a high clock frequency. The resulting encryption speed will be almost the same, although the operation will be performed in more clock cycles. Loss of only one clock cycle will cause only a slight speed decrease, but, on the other hand, it will save a lot of area. Our particular implementation, after modification, will need approximately 340 logic cells: 4 * 3 * 16 for S-boxes + 52 for MDS + 32 for PHT + 32 for subkeys adders + 32 for additional register and multiplexer. Of course discussed above modification is possible only in a sequential design, and it shows one of the main limitations of purely combinational designs.

3.2 Implementation of the key schedule

Key schedule can be implemented in even more different ways than encryption. In case of most today's ciphers, all subkeys have to be precomputed first and then stored in a memory. We need at least one memory to store all 42 subkeys K0...K39, S0, S1, each 32-bit long. A single Xilinx logic cell can be configured as a 32 x 1 RAM memory, so that to store all subkeys, we would have to use at least $\lceil 42 / 32 \rceil$ * (32 x 1) = 64 CLBs. For greater convenience, we would actually need three 32-bit memories, one for whitening subkeys, and two others for the rest of subkeys K, where odd and even keys are kept in separate memories. Each of these memories will store maximum 16 subkeys. In such case, a single CLB can be configured as a 16 x 2 RAM memory, and it is sufficient to use only 48 of CLBs for all subkeys K.

Much more inconvenient are subkeys S_0 and S_1. They cannot be stored in any of the above memories, because they are used simultaneously with the other sybkeys. Moreover, even both keys S cannot be stored together if the cipher round is implemented as a purely combinational logic. Therefore, storing of these subkeys consumes additional 32 CLBs.

The computation of internal keys can be done either on-chip or off-chip. If it is performed on-chip, we have to provide additional block of hardware designated only for this operation. Fortunately, in Twofish, the circuit used for subkey generation is very similar to the one used for encryption. After some little modifications, the same block of hardware can be used for both encryption and key scheduling. The only part of key scheduling that still requires distinct hardware is RS matrix. So far, we did not analyze this matrix and we cannot say how much area it is going to take.

The other possibility is to precompute all subkeys outside of the circuit, and then download them into the circuit memory. This method requires implementing only a memory, but is less convenient, and may affect the security of the implementation.

The main drawback of both methods is a long time of key exchange. In some applications it may be required to change keys very often, and this time may become critical.

To minimize the time of key exchange, the subkeys can be computed on the fly. In case of Twofish, it is relatively easy, because all subkeys are independent of each other. As a result, the key exchange time is limited only by the time of performing multiplication by RS matrix. But nothing comes for free. We have to design additional logic to perform subkeys computation simultaneously with encryption. The straightforward implementation of this function will double the size of the circuit. However, it seems to be possible to compute all subkeys on the fly in the same circuit as encryption by using time sharing. So far, we did not investigate the time penalty inferred by this option.

4 Design of the cipher interface

Our project is divided into two distinct parts: *Twofish round* circuit and external *interface*. The *round* circuit describes exactly one round of the Twofish cipher, including encryption and decryption. The *round* can be run repeatedly to perform entire encryption or decryption process. Our design does not include a key schedule implementation, because we assumed that the speed of key scheduling has a secondary influence on results of comparison of different ciphers. However, the Twofish key schedule is very similar to encryption, and seems to be one of the strong points of this cipher. Therefore, we do not exclude its implementation in the future. In present implementation, all subkeys have to be computed off-chip and then written to the appropriate on-chip memories and registers.

4.1 Twofish interface

External interface is of our own invention, and should not be taken into account during comparison of efficiency of various ciphers. Our purpose was to create a general-purpose interface, which will be very easy to operate in a microprocessor system, and will allow performing encryption with the maximum speed, using an arbitrary AES candidate cipher.

4.1.1 Interface signals

All signals available in the interface are shown in Fig. 10.

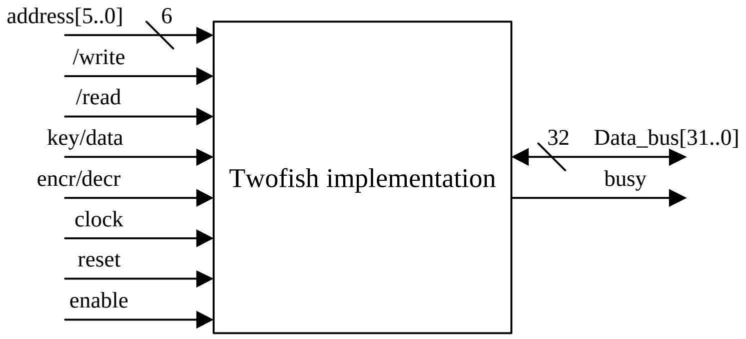


Figure 10 – Signals on the interface.

Signals meaning:

- *data_bus[31..0]* – 32-bit wide bidirectional data bus. The 31st bit *data_bus[31],* is the most significant one.
- *address[5..0]* – 6-bit wide address bus allows accessing all registers defined within the circuit. The 5th bit *address[5]*, is the most significant one.
- */write* – Clocking signal for writing operations. The rising edge of this signal indicates a moment when data are written to registers.
- */read* – Clocking signal for reading operations. The rising edge of this signal indicates a moment when data are read by host system.
- *key/data* – Chooses data or key to be written to registers. High state chooses keys.
- *encr/decr* – With this signal the host system can choose which operation to perform: encryption or decryprion. High state means encryption.
- *clock* – Clocking signal for the entire circuit.
- *reset* – High state resets the device.
- *enable* – High state of this signal enables writing and reading from any register within the device. This signal may be valuable in case of microprocessor systems. If not needed, it can be constantly driven to high state.
- *busy* – This signal is generated by the device. High state indicates encryption process in progress. Change from high to low means that encryption has ended.

4.1.2 *Addressing*

The entire communication with the encrypting device is performed through registers, which are accessible using appropriate register addresses. Fig. 11 describes the registers address space, divided into several segments.

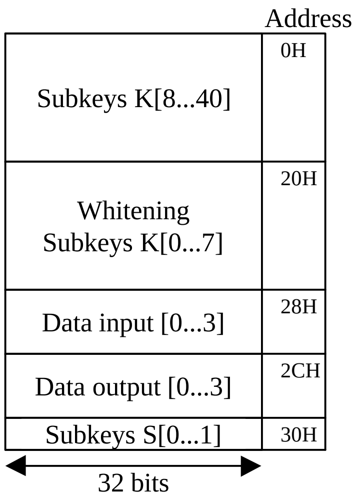


Figure 11 - Address space of memory used to store internal keys.

For each segment, we give only a beginning address. All registers within one segment are addressed sequentially. For example, *K[8]* has address 0H, *K[10]* – 2H, and so on.

Data registers are divided into two groups: input and output. Although they could be accessed with the same address, and distinguished by */write* or */read* signals, we have decided to separate them in such a way, that the three least significant bits of data address are the same as three least significant bits of address of a corresponding whitening key. It simplifies usage of whitening keys.

We have not defined any control register, because our interface is rich of controlling signals. However, this may be a little uncomfortable in use, and in the future, implementation of a control register may be taken into consideration.

4.1.3 *Data and key flow*

The following section describes the architecture of our interface. It is not necessary to learn exactly how it works, but this knowledge is helpful in understanding the protocol described in the next section.

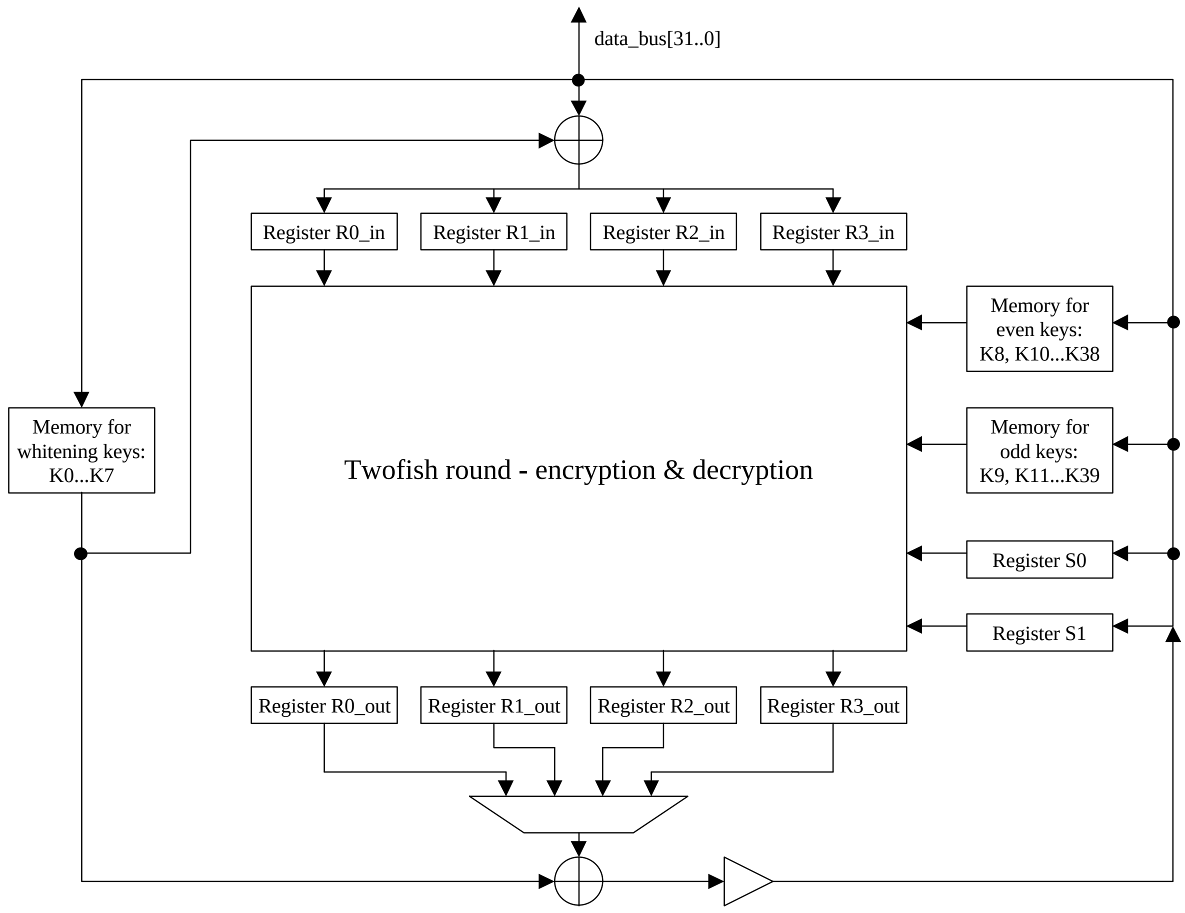


Figure 12 - Data flow.

4.1.3.1 Subkeys

As shown in Fig. 12, the subkeys $K_0...K_{39}$ are kept in three distinct memories. During encryption, even and odd keys from range K8...K39 are used simultaneously. Whitening keys $K_0...K_7$ may also be used at the same time. This is the reason of placing them in different memories. Also, keys S_0 and S_1 are used during encryption, therefore, we had to implement two registers for their storage. Such way of implementation is quite expensive in terms of used CLBs, but even if we used the sequential structure of a Twofish round, we could basically save only 16 CLBs by storing keys S_0 and S_1 in one memory.

4.1.3.2 Data flow

To increase the speed of encryption, we have implemented a pipelined structure for data flow. There are two layers of registers: one on the input, and the other on the output of the *round* circuit. This structure allows writing and reading data independently of encryption process. Obviously, it is possible to write or read from only one register at a time. When the 32-bit input data word is written to the corresponding input register, this word is XOR-ed with the corresponding whitening key. Similar XOR operation takes place during reading each 32-bit output data word. Therefore, we need an access to only one whitening key at a time, and we may store all of these keys in a common memory. Input and output whitening keys have to be exchanged during decryption. It requires setting properly the *encr/decr* signal during the data transmission. Whitening keys are applied to the input data on the fly, which is a little unusual, and may increase the minimum time of the transmission cycle.

NOTE: Interface implementation for other ciphers

We are going to use the same basic interface structure for all ciphers. Of course, for each cipher we will customize this structure according to the specific conditions, nevertheless, our interface will always include at least the following features:

1) Data and keys will be read and written through 32-bit wide bidirectional bus.
2) We will always use pipelined structure for data flow.
3) All subkeys will be computed off-chip, and stored in internal memories in such a way, that all subkeys required during a single round will be available simultaneously within one clock cycle.

4.1.4 The transmission protocol

4.1.4.1 Writing keys and data to the circuit

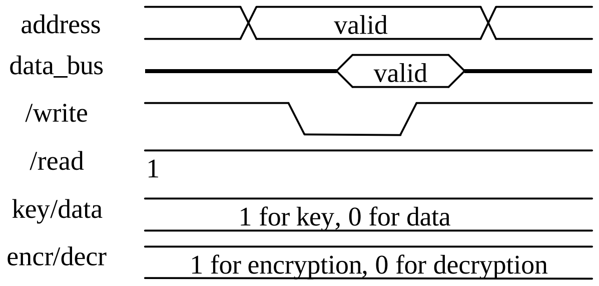


Figure 13 - Writing cycle.

Address bus should be first set to one of the addresses within the input register address space, as shown in Fig. 13. After the address is stable, data bus should be set to the appropriate data or key value, and stay stable at least until the rising edge of the */write* signal. This rising edge indicates latching data in the selected input register.

The *key/data* signal should be set to the appropriate state, depending on what is going to be written to the register: data or key. In fact, this signal is used to switch the address bus of internal memories between the external address bus and an internal counter used during encryption. The *encr/decr* signal should be set in advance to a proper state, depending on intended encryption or decryption process.

Once the keys and data are written to the circuit, they cannot be read, but can always be overwritten.

In case of data, writing to the register under address 2BH is a signal for the encrypting device that new data block is already ready for encryption.

4.1.4.2 Reading data from the circuit

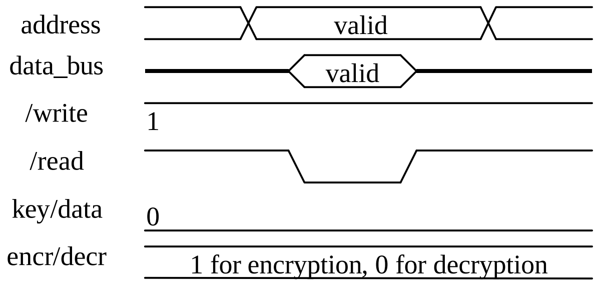


Figure 14 - Reading cycle.

Reading data is very similar to writing. Instead of */write*, a */read* signal is used to indicate reading cycle. Of course, this time the encryption device drives data on the *data_bus*, therefore a host system should release the bus. Data are driven on the *data_bus* as long as */read* signal stays low.

4.1.5 Operation sequence performed by our device after start

The key to use the device efficiently is to understand what operations are performed after *start* signal activation. This sequence is very simple:

1) Read data from input registers and store them inside the circuit round, set *busy* = 1
2) Perform encryption
3) Write encrypted data to output registers, and set *busy* = 0

There are two important conclusions, which should be taken into consideration:

1) Immediately after the *busy* signal becomes high the host may write next data.
2) When *busy* becomes low, encrypted data are available to read. It is important to notice that our device does not wait for a host system, and previously encrypted data should be already read at the end of the next encryption, otherwise they will be lost.

4.1.6 Operation sequence recommended for the host device

The pipelined structure used in our interface makes writing and reading data possible during processing encryption. The sequence of operations looks as follows:

1) Write data

2) Write next data
3) Wait for *busy* = 0
4) Read encrypted data
5) Go to step 2.

4.2 *The Twofish round circuit*

Almost all elements of the Twofish *round* circuit are described directly in the Twofish documentation [1]. Therefore, we focus only on parts that are specific for hardware implementation. Additionally, most of the Twofish building blocks are already described in previous sections.

The implementation of the Twofish round is shown in Fig. 15.

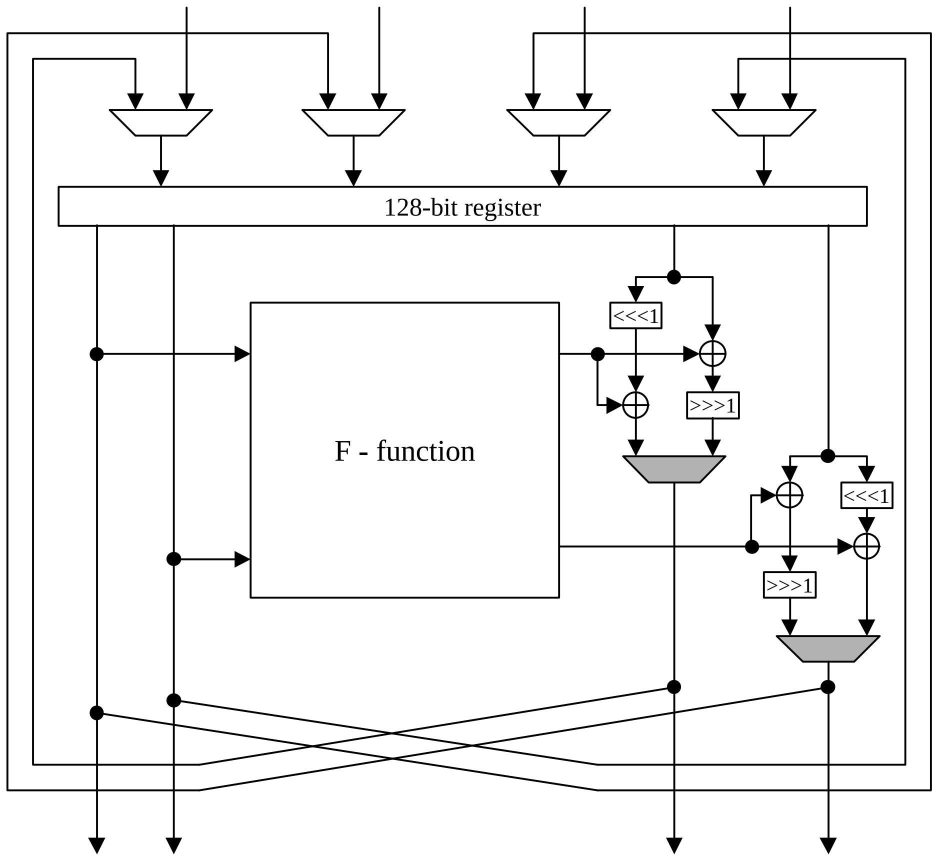


Figure 15 - Twofish round structure.

There is only one 128-bit wide register, which stores intermediate results of encryption. The entire circuit has a feedback that permits data to circulate by repeating a single round computation multiple number of times. This way, we are able to perform the entire encryption.

We have made also a small modification to the original Twofish structure, that enables encryption and decryption to be performed using the same circuit. These additional components include two shaded multiplexers on the right side of the schematic in Fig. 15. Please refer to section 1 and Fig. 4 for more information.

NOTE: The round circuit implementation for other ciphers

For all ciphers that have a round structure, we are going to adapt a similar circuit for encryption and decryption. This circuit will have the following features:

1) At least one 128-bit wide register will be used to store intermediate results. This register will be placed at the beginning of the round.
2) Only one round will be implemented, unless there will be some good reason to break this rule.
3) The feedback loop will be used to bring results of the round function back to the input of the *round* circuit.

5 The results of the Twofish implementation using Xilinx FPGA devices

VHDL (**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage) was chosen as a language used to describe Twofish implementation. VHDL is a standard language for hardware description and is supported by computer aided design software of all major FPGA device vendors. It is a high-level language, which allows describing circuit function without the need to specify the circuit structure. Although writing the VHDL code is very easy, it is not guaranteed that this code will be optimal. The results substantially depend on the software used for synthesis. Additionally, different ways of describing the same circuit produce slightly different results. Therefore, writing a good code requires an in-depth knowledge of how VHDL is interpreted by a synthesizer. To optimize the implementation, we have tried different kinds of description, and compared the results with our expectations (see section 2).

Probably, the best way to write the most efficient code is to use vendor supported libraries. However, this way of coding would make our design specific for a particular device family. Therefore, we have described the entire *round* circuit, which is the core of our design, in pure VHDL'87 language. We expect that using library parts could make our design faster and less area consuming. In the *interface* part, we have made an exception and used library parts from the Xilinx LogiBLOX library. These library parts were necessary to describe memories used to store cipher subkeys. Any other way of implementing these memories would dramatically increase the amount of required area.

The majority of the designer's time has been spent on creating the proper VHDL code, and on functional simulation. We did it using mainly the Active-VHDL program provided by ALDEC Co. We have found this program very convenient to work with VHDL, and to perform functional simulation. Our code was verified based on test vectors provided in Twofish documentation [1]. The verified VHDL code was then synthesized and optimized using FPGA Express. The obtained netlist was exported to the Xilinx Foundation Series 1.5 to create the implementation. The Xilinx design environment was also used for timing simulation.

As target device we have chosen a family of XC4000XL. In most implementations, we have used the following options:

In FPGA Express:

• Clock frequency:	50 MHz
• Speed grade:	-09
• Slew rate:	Slow
• Global buffer:	Automatic, but in case of *clock* and *reset* signals: BUFGLS
• Hierarchy:	Eliminate, but in case of subkey memories: Preserve
• Primitives:	Optimize
• Operator Sharing:	On
• Optimize for:	Speed
• Effort:	High

In Design Manager:

• Trim unconnected logic:	on
• Replace logic to allow logic level reduction:	on
• Generate 5-input functions:	on
• CLB packing strategy:	Fit device
• Pack CLB for:	Structure
• Pack I/O registers/latches into IOBs:	off
• Place & route effort level:	Best result
• # of routing passes:	auto
• # of delay-based cleanup passes:	0

5.1 *Implemented parts*

After analyzing all Twofish components in terms of their ability to be implemented in Xilinx FPGA devices, as described in section 2, we also tried to verify whether our expectations are fulfilled or even exceeded by Xilinx synthesis tools. Implemented parts included:

- q-permutations
- the MDS matrix
- the PHT transform
- the F-function
- full encrypting device in combinational version

5.2 *Implementation of Twofish components*

This section focuses on implementation of q-permutations, the MDS matrix and the PHT transform only. The main purpose to implement Twofish main components separately was to answer the question: how much area they require and how fast they are. To check how many CLBs are required, it is sufficient to implement particular circuit “as is”. The more difficult is to measure the propagation delay through the circuit. All of the aforementioned components implement internal functions, and their inputs are not originally connected to the external pins. The delay introduced by external pins is significantly longer than the delay of interconnections among CLBs. To avoid this effect, all circuits described in this section were tested in a configuration shown in Fig. 16.

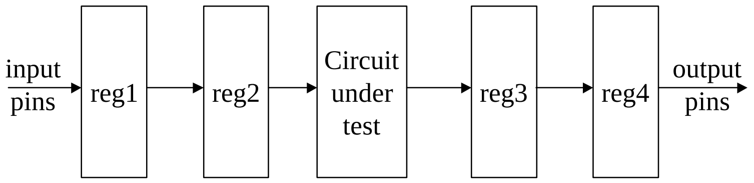


Figure 16 - Testing environment.

We have decided to use two levels of registers between inputs and outputs, because the first level (reg1 and reg4) could be placed in IOB elements, and in this case the paths between CLBs and registers would be long. The propagation delay is measured as the time interval between the moment when the stable value appears at the outputs of the register reg2, and the time, when the proper stable value appears on the inputs to the register reg3. In case of sequential designs, the setup time of reg3 and the propagation delay through the reg2 have to be taken into account. In Xilinx devices, the setup time is equal 0 ns if the register is placed in the same CLB as proceeding combinational logic; otherwise it does not exceed 2 ns. The delay caused by reg2 is about 3-4 ns as shown in Fig. 17-19 (the reg2 output is shown always directly under the clock signal). In case of the combinational design, these additional delays need to be taken into account only once for the entire round function.

5.2.1 *q – permutation*

The implementation of the q-permutation met our expectations by taking 16 logic cells. The propagation time is shown in Fig. 17, and is 9.6 ns. In sequential design the propagation delay, measured from rising edge of clock is 13.3 ns. Additionally, the setup time for reg3 must be considered. The implementation report, generated automatically by synthesizer states that the minimum safe clock period is 13.8 ns.

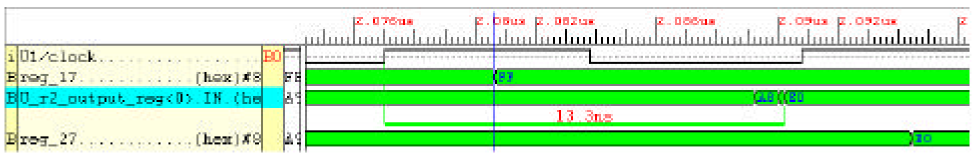


Figure 17 - Timing simulation of the q-permutation.

5.2.2 *MDS matrix*

In case of the MDS matrix implementation, the Xilinx compiler made it in a completely different way that we had expected. Our analysis shows that it should be the fastest function, but in implementation it appeared to be not much faster than the q-permutation, causing the delay of 9.5 ns (13.5 ns with the reg2 propagation delay). The compiler realized it using only 48 logic cells (we expected 52), but, on the other hand, it has changed it into a multilevel structure. We tried several different ways of describing this matrix, but the result still was the same. Probably, the only way to enforce our implementation would be to use library components.

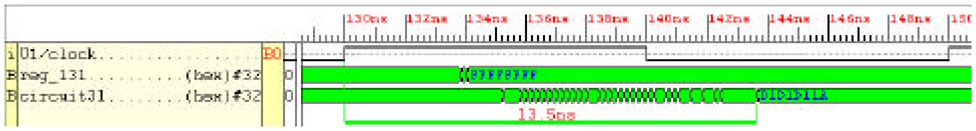


Figure 18 - Timing simulation of the MDS matrix.

5.2.3 *PHT transform*

The PHT transform was implemented using carry-chain feature of Xilinx CLBs, and it took 34 CLBs. We expected a long delay that is needed for the signal propagation through the entire carry-chain, but Xilinx FPGA seems to be very fast, and it took only 6.8 ns (10.8 ns, taking into account propagation delay through the reg2). Surprisingly, the minimum clock period reported by the synthesizer is 14.7 ns.

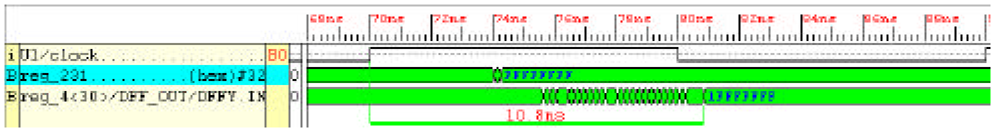


Figure 19 - Timing simulation of the PHT transform.

5.2.4 *Summary of main components implementations*

	q-permutation		MDS		PHT	
Expected delay (# of CLB levels)	4		2		carry-chain	
Actual delay (ns)	9.6*	13.3**	9.5*	13.5**	6.8*	10.8**
Expected size (# CLBs)	16		52		32	
Actual size (# CLBs)	16		48		34	

* with the propagation time through reg2
** without the propagation time through reg2

5.3 *Implementation of the F-function and full round*

The F-function is a core part of the entire *round* circuit, and it determines the encryption speed. Moreover, the key schedule is based on the same function. Therefore, the amount of area consumed by the full implementation of Twofish can be estimated as twice the area of the F-function implementation. This estimate does not take into account the amount of area used by the RS matrix.

We have implemented two versions of the F-function: sequential and fully combinational. Both versions of F-functions were used for implementation of the *round* circuit. In the sequential version, we used six register layers situated at the outputs of the:

- 1st level of q-permutations,
- 2nd level of q-permutations,
- 3rd level of q-permutations,

- MDS matrix,
- PHT transform,
- subkey adders.

For the combinational version of the F-function, the required amount of CLBs is 605, which matches perfectly with our expectations. The corresponding entire *round* function implementation takes 705 logic cells. The sequential version of the F-function takes 613 CLBs, and the entire *round* implementation fits in 728 logic cells.

5.4 *Implementation of the fully functional encrypting device*

The implementation of the Twofish encrypting device consists of the round function and the interface described in section 4. We have implemented a purely combinational and a sequential version of the entire circuit, with different optimization options. The smallest device that contains the entire circuit is XC4028XL. As a target package for this device we used HQ208.

Version	Combinational				Sequential			
Optimized for	Speed		Area		Speed		Area	
Routing*	Best	Default	Best	Default	Best	Default	Best	Default
Size (# CLBs)	911		888		920		926	
Max. clock rate (MHz)	10.7	9.1	10.6	9.1	33	28.5	35.9**	27.9
# of clock cycles per data block	16				112			
Maximum throughput (Mbit/s)	85.6	72.8	84.8	72.8	37.7	32.6	41.0	31.9

* Best: # of routing passes: 1000
# of delay-based cleanup passes: 5
Default: options are set as listed in section 5.
** This is the most surprising result, but we have checked all options and we are certain that the optimizations for speed and area were not interchanged.

As we expected, the combinational version of the implementation is faster than the sequential version. The throughput of the sequential implementation is surprisingly low. According to the results obtained for individual Twofish components, we have expected the maximum clock rate at a level of 50 MHz. Actually, the clock rate did not exceed the rate of 36 MHz, resulting in the throughput lower by 50% compared to the combinational version. It may be the result of poor routing, because the design takes approximately 90% of area available in the XC4028 device.

Although, the sequential implementation is slower, it may be still considered in case of the ECB cipher mode or interleaved modes, because, for these modes, the computations can be performed for many data blocks simultaneously. Obviously, this solution requires implementing additional FIFO buffers instead of registers at the input and output of the circuit, and one similar FIFO buffer on the left half of the data inside the round. Implementing these buffers using flip-flops would be very expensive, but instead, we can use the fact that a Xilinx CLB can be configured as a 16 x 1 dual-port RAM. Therefore, each required 32-bit-wide FIFO buffer would take only 32 logic cells. Using pipelined mode for our particular sequential design (seven levels of registers), would give the throughput of 263.9 Mbit/s. One FIFO buffer can store up to 16 data blocks, therefore, the throughput may be further increased by dividing the circuit into more register levels. This operation does not require any additional hardware.

Summary

We have designed the hardware implementation of Twofish, one of the leading candidates to the new Advanced Encryption Standard (AES). Twofish is a symmetric-key block cipher with a 128-bit input/output block, and key sizes 128, 192, and 256 bits. Special assumptions regarding our implementation are:

- the key size is limited to 128 bits;
- internal keys are generated off-chip, and loaded to the internal FPGA memory before the encryption or decryption starts;
- a general interface of our own design, based on the 46-bit bus with 32 data lines, is used to exchange data and keys with the host computer.

We have developed the VHDL'87 description of Twofish, and verified its correctness by functional simulation, using Active-VHDL simulator from Aldec, Inc. Test vectors, used for verification, were generated based on the C reference code provided by the inventors of the cipher [1]. The VHDL code was mapped into Xilinx FPGA devices, using Xilinx Foundation Series v. 1.5.

The circuit was optimized for maximum speed. The number of CLBs necessary to implement the entire circuit is 911, with 705 CLBs used for the main encryption/decryption block, and the remaining cells used for the input/output interface, and storage of internal keys. The smallest Xilinx FPGA device able to implement the circuit is XC4028, with the maximum number of CLBs equal to 1024, and the equivalent number of logic gates equal to 28,000.

The maximum clock frequency obtained from the timing simulation for combinational implementation is 10.7 MHz. With 16 cipher rounds, each executed within one clock cycle, this maximum clock frequency corresponds to the encryption and decryption rate of $128 \cdot 10.7/16 = 85.6$ Mbit/s.

6 Bibliography

1. Twofish: A 128 bit Block Cipher; Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson; 15 June,1998
   http://www.counterpane.com/twofish.html

2. Performance Comparison of the AES Submissions; Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson; December 18,1998
   http://www.counterpane.com/twofish.html

3. Twofish Technical Report #3; Doug Whiting, Bruce Schneier; December 2,1998
   http://www.counterpane.com/twofish.html

4. Xilinx Documentation, XC4000E and XC4000X Series Field Programmable Gate Arrays; January 29,1999
   http://www.xilinx.com

Appendix - List of source files

twofish.vhd	– main file containing description of the *interface* part, and the instantiation of the *round* part.
round.vhd	– the *round* circuit; main register, encryption and decryption, and the instantiation of the F-function.
f_function.vhd	– two h-functions, one PHT transform, and two subkey adders.
pht_transf.vhd	– the PHT transform.
h_function	– instantiation of the S-boxes and the MDS matrix.
mds.vhd	– multiplication used in the MDS matrix.
mul_5B.vhd	– multiplication by 5B in the Galois field GF(2^8).
mul_EF.vhd	– multiplication by EF in the Galois field GF(2^8).
s_boxes.vhd	– four S-boxes 0...3 formed from q-permutations.
perm_q*x*.vhd	– appropriate permutation q_0 or q_1.
s_box_t*x*_q*i*.vhd	– contents of the appropriate S-box t_0...t_3 of permutation q_0 or q_1.
vectors.txt	– test vectors.