International Journal of Engineering & Technology Sciences
Volume 03, Issue 01, Pages 13-21, 2015

ISSN: 2289-4152

A Hardware Implementation of Twine Block Cipher High Level Synthesis Approach

Ali Nemati [a,b], Arash Ahmadi [a,b,*], Vahab Al-din Makki [a,b]

[a] Department of Electrical Engineering, College of Basic Science, Kermanshah Science and Research Branch, Islamic Azad University, Kermanshah, Iran).
[b] Department of Electrical Engineering, College of Basic Science, Kermanshah Branch, Islamic Azad University, Kermanshah, Iran).

* Corresponding author. Tel.: +989373919597;
E-mail address: aahmadi@razi.ac.ir

A b s t r a c t

Keywords:

Lightweight Cryptography,
Twine Algorithm,
High Level Synthesis,
Field Programmable Gate Array (FPGA).

Radio frequency identification (RFID) are becoming a part of our everyday life with a wide range of applications such as labeling products and supply chain management and etc. At the same time, security and privacy issues remain as an important problem, thus, with the large deployment of low resource devices such as RFID systems and sensor nodes, increasing need to provide security and privacy among such devices, has arisen. Accordingly, concept of lightweight cryptography is introduced. Encryption algorithms are classified into various types such as block and stream ciphers. Lightweight block cipher plays a major role as a building block for security protocols. In 2011 Tomoyasu Suzaki et al in NEC Corporation proposed a new versatile block cipher named as Twine, which are suitable for extremely constrained environments and multiple platforms. In this paper, we simulate and synthesize the Twine block cipher family. Synthesis estimation results of the Twine cryptography algorithm on a FPGA are presented.

Accepted: 23 January2015

1. Introduction

Radio Frequency Identification (RFID) technology is one of the most important technologies in the field of computing that is present everywhere. Although this technology gives many advantages than other identification systems, there are also related security risks that can’t be easily passed. RFID security is a significant research subject today, as proved by the great number of research papers published in this domain in last decade. As RFID technology is a universal technology, privacy and security are the important problems to be solved [6].

During last decade, lightweight cryptography has provided security solutions for highly constrained devices and embedded systems. A large number of research works have already been done in this domain, which have specifically achieved optimized block ciphers suitable for lightweight cryptographic applications, for instance Katan, Piccolo and several others [1, 3]. As RFID devices become broadly utilized in different applications, it is vital to provide various levels of security, associating to the application.

Different applications have different security requirements, and in any case to achieve the desired level of security, protocols should be developed. Security has a direct relation with key size, meaning that with the increase of key-length, the level of security grows. Each block cipher is adaptively to initial cryptographic principles and we expect that any lightweight protocol can be based on a block cipher with the suitable data block and key size. Although some standard cryptographic algorithms such as Advance Encryption Standard (AES) have been proposed for lightweight applications, choosing AES is not a good option for highly constrained environments [12, 7]. Only few number of the block ciphers can be implemented on a small circuit with minimal power and area requirements and some others can be embedded on microcontrollers with limited flash, SRAM, and/or power availability [7]. Obviously, any effort to optimize the performance of a parameter on one side will have a negative impact on the performance of other parameters on the other side meaning that the trade-off law is in place.

We organize this paper as follows: In Section 2 we describe the specification of Twine briefly. Section 3 presents the synthesis of the Twine block cipher family. In Sections 4 we present the synthesis estimation results of Twine algorithm on FPGA, respectively. Section 5 concludes the paper.

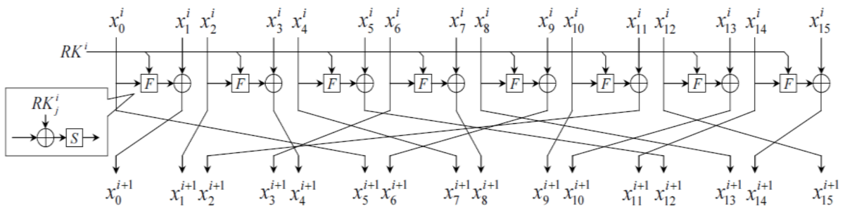
2. Algorithm Description

Twine is a 64-bit block cipher with two supported key lengths, 80 and 128 bits. If the key length is needed to be specified, we write Twine -80 or Twine -128 to denote the corresponding version. The global structure of Twine is a variant of Type-2 GFS with 16 4-bit sub-blocks. Given a 64-bit plaintext, $P_{(64)}$, and a round key, $RK_{(32\times36)}$, the cipher produces the cipher text $C_{(64)}$. Round key $RK_{(32\times36)}$ is derived from the secret key, $K_{(n)}$ with $n \in \{80, 128\}$, using the key schedule. A round function of Twine consists of a nonlinear layer using 4-bit S-boxes and a diffusion layer, which permutes the 16 blocks. This round function is iterated for 36 times for both key lengths, where the diffusion layer of the last round is omitted. The encryption process can be written as Algorithm 2.1. The S-box, S, is a 4-bit permutation defined as Table 1. The permutation of block indexes, $\pi : \{0, ... , 15\} \rightarrow \{0, ... , 15\}$, where j-th sub-block (for j = 0, ... , 15) is mapped to $\pi[j]$-th sub-block, is depicted at Table 2. Round function of Twine algorithm is shown in Figure 1 [2].

Algorithm 2.1: Twine .Enc ($P_{(64)}$, $RK_{(32\times36)}$, $C_{(64)}$)

```
$X^1_{(64)} \leftarrow P$
$RK^1_{(32)} \| \dots \| RK^{35}_{(36)} \leftarrow RK_{(32\times36)}$
for $i \leftarrow 1$ to 35
do {
    $X^i_{0(4)} \| X^i_{1(4)} \| \dots \| X^i_{14(4)} \| X^i_{15(4)} \leftarrow X^i_{(64)}$
    $RK^i_{0(4)} \| RK^i_{1(4)} \| \dots \| RK^i_{6(4)} \| RK^i_{7(4)} \leftarrow RK^i_{(32)}$
    for $j \leftarrow 0$ to 7
    do $X^i_{2j+1} \leftarrow S(X^i_{2j} \oplus RK^i_j) \oplus X^i_{2j+1}$
    for $h \leftarrow 0$ to 15
    do $X^{i+1}_{\pi[h]} \leftarrow X^i_h$
    $X^{i+1} \leftarrow X^{i+1}_0 \| X^{i+1}_1 \| \dots \| X^{i+1}_{14} \| X^{i+1}_{15}$
}
for $j \leftarrow 0$ to 7
do $X^{36}_{2j+1} \leftarrow S(X^{36}_{2j} \oplus RK^{36}_j) \oplus X^{36}_{2j+1}$
$C \leftarrow X^{36}$ [2]
```

$\oplus$ denotes bitwise exclusive-OR. For binary strings, x and y, x‖y denotes the concatenation of x and y. Let |x| denote the bit length of a binary string x. If |x| = m, x is also written as x(m) to emphasize its bit length. The key schedule produces $RK_{(32\times36)}$ from the secret key, $K_{(n)}$, where $n \in \{80, 128\}$. The pseudo code of key schedule for 80-bit key is in Algorithm 2.2. Round constant, $CON^i_{(6)} = CON^i_{H(3)} \| CON^i_{L(3)}$, is defined as 2^i in $GF(2^6)$ with primitive polynomial Z^6+Z+1. The exact values are listed at Table 3 [2].



x_0^i x_1^i x_2^i x_3^i x_4^i x_5^i x_6^i x_7^i x_8^i x_9^i x_{10}^i x_{11}^i x_{12}^i x_{13}^i x_{14}^i x_{15}^i
RK^i
RK_j^i
F
S
x_0^{i+1} x_1^{i+1} x_2^{i+1} x_3^{i+1} x_4^{i+1} x_5^{i+1} x_6^{i+1} x_7^{i+1} x_8^{i+1} x_9^{i+1} x_{10}^{i+1} x_{11}^{i+1} x_{12}^{i+1} x_{13}^{i+1} x_{14}^{i+1} x_{15}^{i+1}

Fig.1: Round Function of Twine [2]

Table 1: S-box (hexadecimal) [2]

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(x)	C	0	F	A	2	B	9	5	8	3	D	7	1	E	6	4

Table 2: Block Shuffle π and π^{-1}[2]

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
π[j]	5	0	1	4	7	12	3	8	13	6	9	2	15	10	11	14
π^{-1}[j]	1	2	11	6	3	0	9	4	7	10	13	14	5	8	15	12

Table 3: Round Constant [2]

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CON^i	01	02	04	08	10	20	03	06	0C	18	30	23	05	0A	14	28	13	26
i	19	20	2	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
CON^i	0F	1E	3C	3B	35	29	11	22	07	0E	1C	38	33	25	09	12	24	0B

Table 4: Test Vector (hexadecimal) [2]

key length	80-bit
key	00112233 44556677 8899
plaintext	01234567 78ABCDEF
ciphertext	7C1F0F80 B1DF9C28

Algorithm 2.2: Twine key schedule – 80 ($K_{(80)}$, $RK_{(32\times36)}$)

```
$WK_{(80)} \leftarrow K$
$WK_{0(4)} \| WK_{1(4)} \| \dots \| WK_{18(4)} \| WK_{19(4)} \leftarrow WK$
$RK^1_{0(4)} \leftarrow WK_1,\ RK^1_{1(4)} \leftarrow WK_3,\ RK^1_{2(4)} \leftarrow WK_4,$
$RK^1_{3(4)} \leftarrow WK_6,\ RK^1_{4(4)} \leftarrow WK_{13},\ RK^1_{5(4)} \leftarrow WK_{14},$
$RK^1_{6(4)} \leftarrow WK_{15},\ RK^1_{7(4)} \leftarrow WK_{16}$
$RK^1_{(32)} \leftarrow RK^1_0 \| RK^1_1 \| \dots \| RK^1_6 \| RK^1_7$
for $i \leftarrow 2$ to 36
do
    $WK_1 \leftarrow WK_1 \oplus S(WK_0),$
    $WK_4 \leftarrow WK_4 \oplus S(WK_{16})$
    $WK_7 \leftarrow WK_7 \oplus 0\|CON^{i-1}_H,$
    $WK_{19} \leftarrow WK_{19} \oplus 0\|CON^{i-1}_L$
    $Tmp_0 \leftarrow WK_0,\ Tmp_1 \leftarrow WK_1,$
    $Tmp_2 \leftarrow WK_2,\ Tmp_3 \leftarrow WK_3$
    for $j \leftarrow 0$ to 3
    do
        $WK_{j*4} \leftarrow WK_{j*4+4},\ WK_{j*4+1} \leftarrow WK_{j*4+5}$
        $WK_{j*4+2} \leftarrow WK_{j*4+6},\ WK_{j*4+3} \leftarrow WK_{j*4+7}$
    $WK_{16} \leftarrow tmp_1,\ WK_{17} \leftarrow tmp_2,$
    $WK_{18} \leftarrow tmp_3,\ WK_{19} \leftarrow tmp_0$
    $RK^i_0 \leftarrow WK_1,\ RK^i_1 \leftarrow WK_3,\ RK^i_2 \leftarrow WK_4,$
    $RK^i_3 \leftarrow WK_6,\ RK^i_4 \leftarrow WK_{13},\ RK^i_5 \leftarrow WK_{14},$
    $RK^i_6 \leftarrow WK_{15},\ RK^i_7 \leftarrow WK_{16}$
    $RK^i_{(32)} \leftarrow RK^i_{0(4)} \| RK^i_{1(4)} \| \dots \| RK^i_{6(4)} \| RK^i_{7(4)}$
$RK_{(32\times36)} \leftarrow RK^1_{(32)} \| RK^2_{(32)} \| \dots \| RK^{35}_{(36)} \| RK^{36}_{(32)}$ [2]
```

3. High Level Synthesis

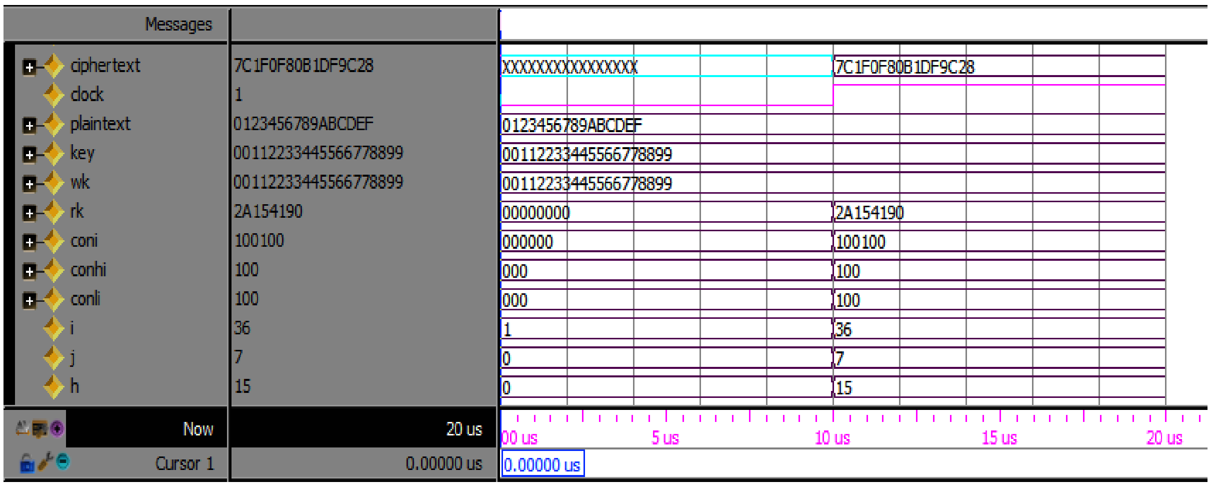
HLS produces register-transfer level designs from behavioural characteristics in an automatic procedure. A high-level synthesis system which realizes design automation for such a data path needs the following two steps:

1) A high-level synthesis system should prepare a draftsman with more than one possible design options for a given behavioural specification. Thus a designer can choose the optimum design between the options based on several criteria.

2) In order to attain the necessity(1), a high level synthesis systems gives more exact information on performance, area and power wastage for each design options.

Table 5: Truth Table

I/O↓ No.→	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
Input (abcd)	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Output (wxyz)	1100	0000	1111	1010	0010	1011	1001	0101	1000	0011	1101	0111	0001	1110	0110	0100



```
Messages
ciphertext   7C1F0F80B1DF9C28       XXXXXXXXXXXXXXXX          7C1F0F80B1DF9C28
clock        1
plaintext    0123456789ABCDEF       0123456789ABCDEF
key          00112233445566778899   00112233445566778899
wk           00112233445566778899   00112233445566778899
rk           2A154190               00000000                  2A154190
coni         100100                 000000                    100100
conhi        100                    000                       100
conli        100                    000                       100
i            36                     1                         36
j            7                      0                         7
h            15                     0                         15
Now          20 us                  00 us   5 us   10 us   15 us   20 us
Cursor 1     0.00000 us             0.00000 us
```

Fig.2: Simulation Results

High-level synthesis is similar to software compilation transposed to the hardware scope. HLS tools automate the process that generates RTL architecture from an algorithmic behavior of the original characteristic [8].

To start the simulation and synthesis steps, we have chosen Twine-80 algorithm. According to section 2, it is explicit that data block length = 64 bit, key length = 80 bit and rounds = 36. Also, an example of this type of algorithm is considered in Table 4.

At first, we’ve written VHDL code of Twine-80 algorithm for above example and simulated it in Modelsim software. Simulation results are shown in Figure 2. According to this figure, it is clear that clock, plaintext and key are considered as input parameters and ciphertext is considered as an output parameter. The clock for this simulation is selected 100 KHz. Then due to the structure of the Twine encryption algorithm, synthesis can be divided into two sub-sections: 1. Synthesis of S-box layer, 2. Synthesis of data processing part. In view of the

S-box table presented in Section 2, Truth table can be used for inputs and outputs are presented in Table 5.

As well as it is explicit that according to the truth table proposed in table 5, we can write some relations between inputs and outputs based on the sum of products (minterms) as follows:

$$W = a'b'c'd' + a'b'cd' + a'b'cd + a'bc'd + a'bcd' + ab'c'd' + ab'cd' + abc'd = a'b'd' + ab'd' + bc'd + a'b'cd + a'bcd' = b'd' + bc'd + a'c\,(b\oplus d) \quad (1)$$

$$X = a'b'c'd' + a'b'cd' + a'bcd + ab'cd' + ab'cd + abc'd + abcd' + abcd = a'b'd' + ab'c + abc + abc'd + a'bcd = ac + a'b'd' + bd\,(a\oplus c) \quad (2)$$

$$Y = a'b'cd' + a'b'cd + a'bc'd' + a'bc'd + ab'c'd + ab'cd + abc'd + abcd' = a'b'c + a'bc' + ab'd + ab\,(c\oplus d) \quad (3)$$

$$Z = a'b'cd' + a'bc'd + a'bcd' + a'bcd + ab'c'd + ab'cd' + ab'cd + abc'd' = ab'd + b'cd' + a'bc + bc'\,(a\oplus d) \quad (4)$$

]

Regarding the logical relations obtained, data flow graph can be drawn for S-box. Allocation of operational and storage units and interconnections are a subset of the allocation process. Choosing each type of the library from operation units that used to perform the desired operation is process of resource allocation. Memory units employed for storing data values. It should be considered that in allocation and binding steps, a specific hardware can be used to perform several operations, so we've used a concept called resource sharing. Data Flow Graph (DFG) for the S-box is drawn and efficient state for scheduling, allocation and binding is shown in Figure 3. In next step, we are designed hardware required to run the S-box that depicted in Figure 4. This hardware includes 17Muxes, 3ALUs and 3 standard Registers and hardware design is efficient. Designed sequencer for this hardware is shown in Figure 5. This sequencer includes two AND gates, one counter and a very simple controller. According to the number of control steps, counter determines which outputs of the controller are '0'and which are '1'. Sequencer gives necessary instruction execution time of the hardware. Sequencer unit parameters for this hardware are summarized in Table 6.

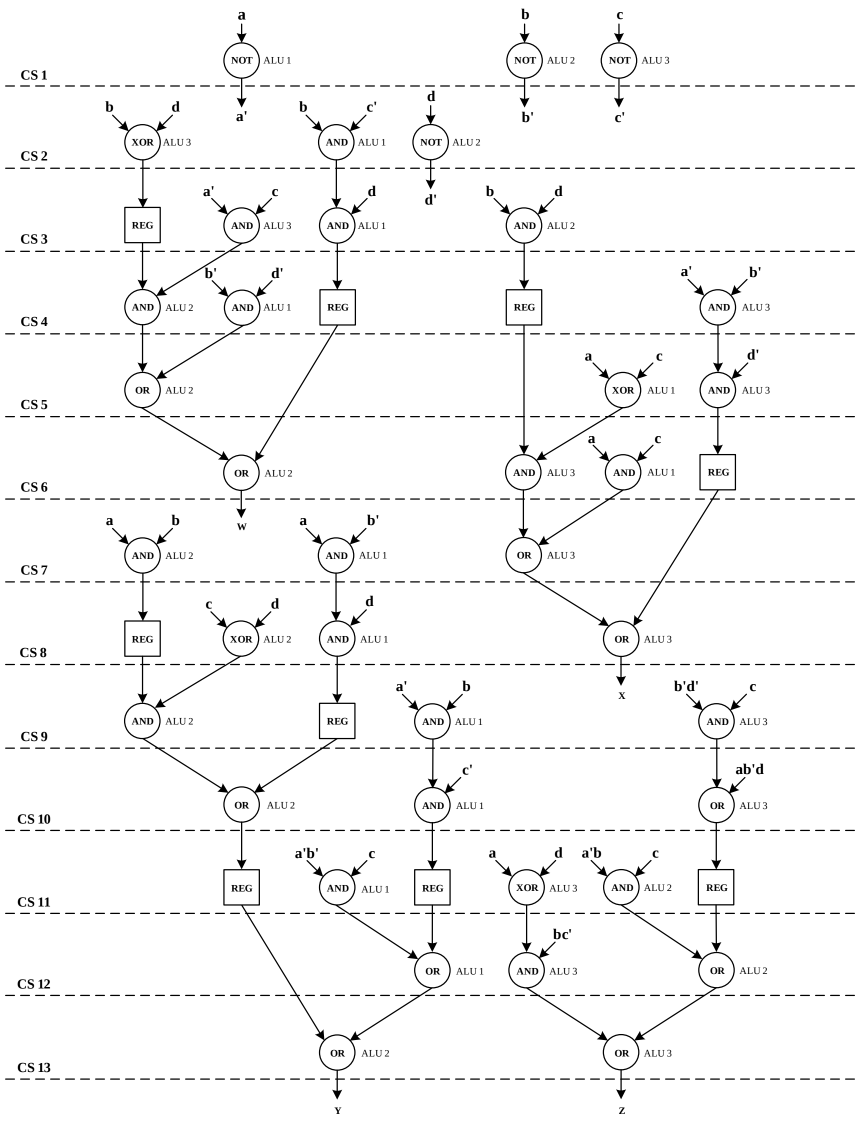


Fig.3: Scheduling, Allocation and Binding for S-box

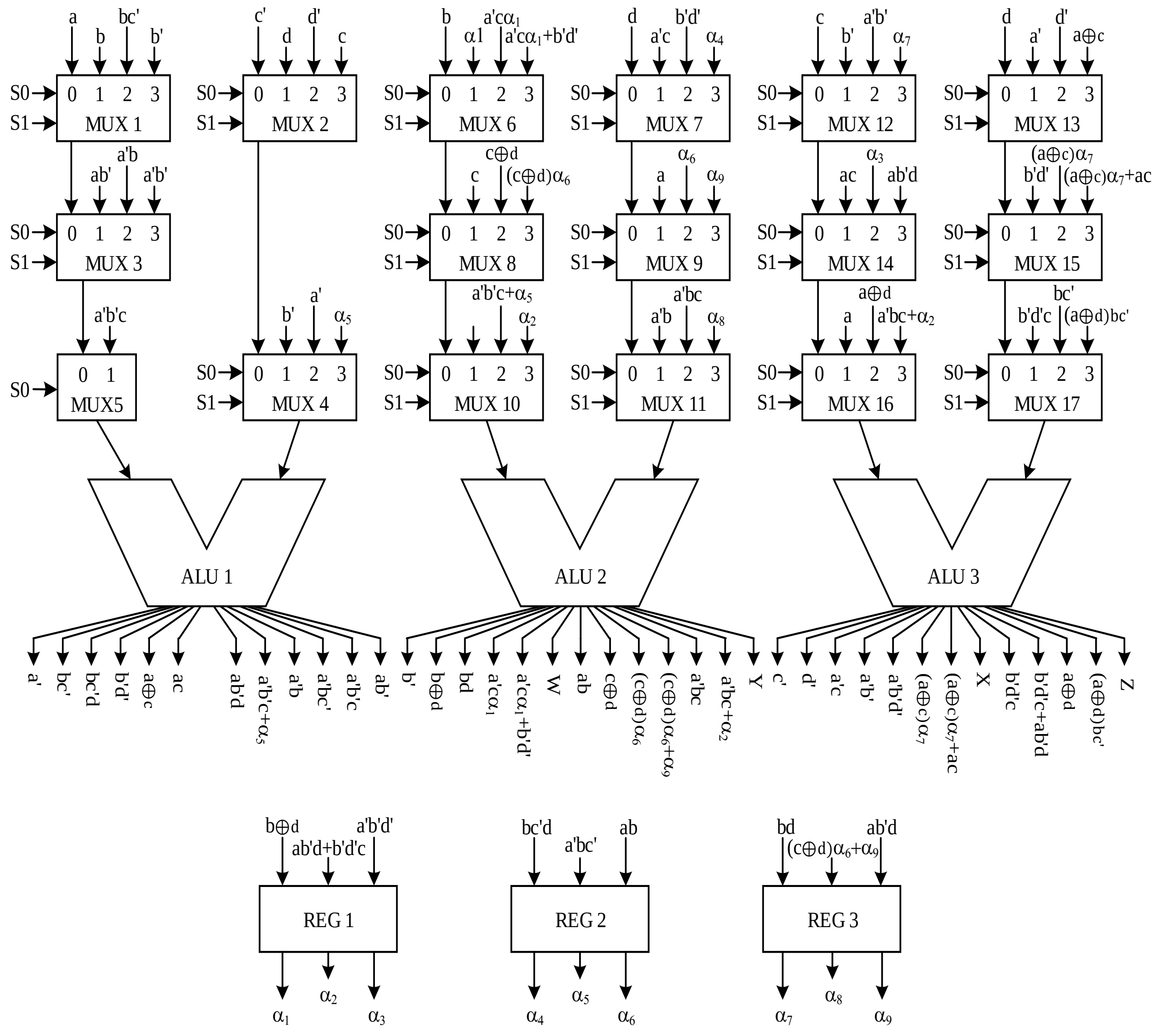

MUX 1
MUX 2
MUX 3
MUX 4
MUX5
MUX 6
MUX 7
MUX 8
MUX 9
MUX 10
MUX 11
MUX 12
MUX 13
MUX 14
MUX 15
MUX 16
MUX 17
S0
S1
ALU 1
ALU 2
ALU 3
REG 1
REG 2
REG 3
a' bc' bc'd b'd' a⊕c ac ab'd a'b'c+α5 a'b a'bc' a'b'c ab'
b' b⊕d bd a'cα1 a'cα1+b'd' W ab c⊕d (c⊕d)α6 (c⊕d)α6+α9 a'bc a'bc+α2 Y
c' d' a'c a'b' a'b'd' (a⊕c)α7 (a⊕c)α7+ac X b'd'c b'd'c+ab'd a⊕d (a⊕d)bc' Z
b⊕d ab'd+b'd'c a'b'd'
bc'd a'bc' ab
bd (c⊕d)α6+α9 ab'd
α1 α2 α3 α4 α5 α6 α7 α8 α9

Fig.4: Hardware Design

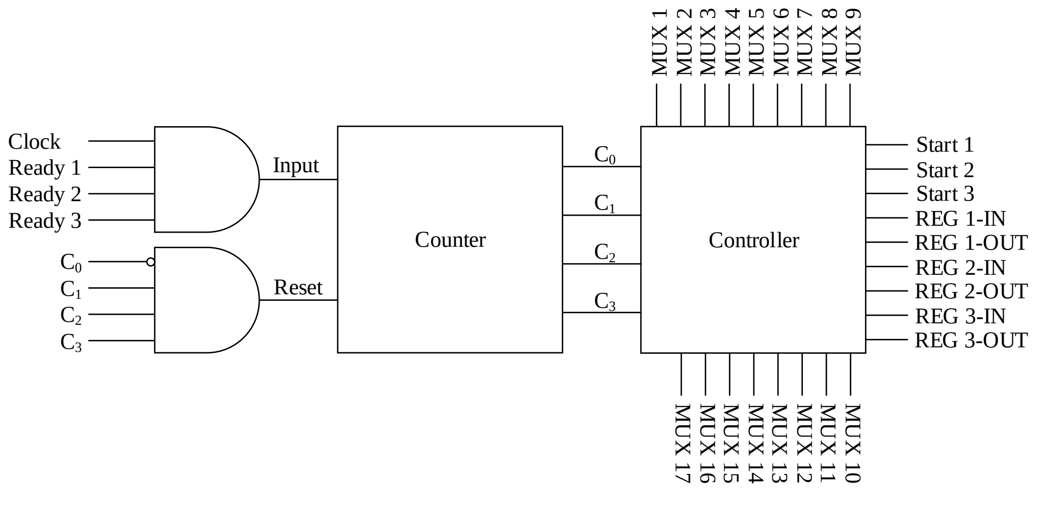


Fig.5: Sequencer Unit

Table 6: Sequencer Unit Parameters

Control Steps ($C_3C_2C_1C_0$)	Start 1	Start 2	Start 3	REG 1 IN	REG 1 OUT	REG 2 IN	REG 2 OUT	REG 3 IN	REG 3 OUT	MUX 1 (S_1S_0)	MUX 2 (S_1S_0)	MUX 3 (S_1S_0)	MUX 4 (S_1S_0)	MUX 5 (S_0)	MUX 6 (S_1S_0)	MUX 7 (S_1S_0)	MUX 8 (S_1S_0)	MUX 9 (S_1S_0)	MUX 10 (S_1S_0)	MUX 11 (S_1S_0)	MUX 12 (S_1S_0)	MUX 13 (S_1S_0)	MUX 14 (S_1S_0)	MUX 15 (S_1S_0)	MUX 16 (S_1S_0)	MUX 17 (S_1S_0)
0001	1	1	1	0	0	0	0	0	0	0 0	× ×	0 0	× ×	0	0 0	× ×	0 0	× ×	0 0	× ×	0 0	× ×	0 0	× ×	0 0	× ×
0010	1	1	1	0	0	0	0	0	0	0 1	0 0	0 0	0 0	0	0 0	0 0	0 0	0 0	0 0	0 0	× ×	0 0	× ×	0 0	× ×	0 0
0011	1	1	1	1	0	0	0	0	0	1 0	0 1	0 0	0 0	0	0 0	0 0	0 0	0 0	0 0	0 0	0 0	0 1	0 0	0 0	0 0	0 0
0100	1	1	1	0	1	1	0	1	0	1 1	1 0	0 0	0 0	0	0 1	0 1	0 0	0 0	0 0	0 0	0 1	0 1	0 0	0 0	0 0	0 0
0101	1	1	1	0	0	0	0	0	0	0 0	1 1	0 0	0 0	0	1 0	1 0	0 0	0 0	0 0	0 0	1 0	1 0	0 0	0 0	0 0	0 0
0110	1	1	1	1	0	0	1	0	1	0 0	1 1	0 0	0 0	0	1 1	1 1	0 0	0 0	0 0	0 0	1 1	1 1	0 0	0 0	0 0	0 0
0111	1	1	1	0	0	0	0	0	0	0 0	× ×	0 0	0 1	0	0 0	× ×	0 0	0 1	0 0	0 0	× ×	× ×	0 1	1 0	0 0	0 0
1000	1	1	1	0	1	1	0	0	0	× ×	0 1	0 1	0 0	0	× ×	0 0	0 1	0 0	0 0	0 0	× ×	× ×	1 0	1 1	0 0	0 0
1001	1	1	1	0	0	0	1	1	0	0 1	× ×	0 0	1 0	0	× ×	× ×	1 0	1 0	0 0	0 0	0 0	× ×	0 0	0 1	0 0	0 0
1010	1	1	1	0	0	0	0	0	0	× ×	0 0	1 0	0 0	0	× ×	× ×	1 1	1 1	0 0	0 0	× ×	× ×	1 1	× ×	0 0	0 1
1011	1	1	1	1	0	1	0	1	0	× ×	1 1	1 1	0 0	0	× ×	× ×	0 1	× ×	0 0	0 1	× ×	0 0	× ×	0 0	0 1	0 0
1100	1	1	1	0	1	0	1	0	0	× ×	× ×	× ×	1 1	1	× ×	× ×	× ×	× ×	1 1	1 0	× ×	× ×	× ×	× ×	1 0	1 0
1101	0	1	1	0	0	0	0	0	1	× ×	× ×	× ×	× ×	×	× ×	× ×	× ×	× ×	1 0	1 1	× ×	× ×	× ×	× ×	1 1	1 1
1110	RESET																									

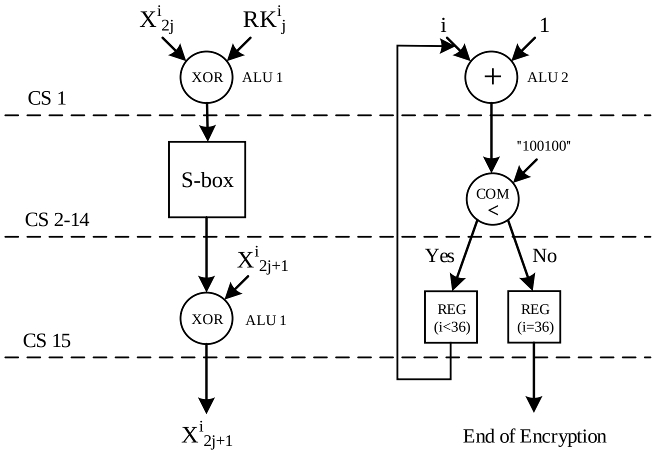


Fig.6: Scheduling, Allocation and Binding for Data Processing

Table 7: Implementation Result

Resource	Available	Utilization	(%)
Register	122880	346	1%
LUT	122880	419	1%
Slice	30720	131	1%
IO	960	64	6%
BUFG	32	7	21%

In next step, for synthesize data processing part, according to the before step and the keys obtained in each round, the efficient state of scheduling for data processing is depicted in Figure 6, that in each control step, we have needed just 3 ALUs and 1 Comparator. Encryption function is done in 15 control steps, so that whole encryption operation is performed in 36×15 = 540 CS. A noteworthy point in Figure 6 is that comparator can be used in one of the steps 2 to 14 according to view of designer.

4. Implementation

The Xilinx synthesis technology synthesizes VHDL code to make Xilinx files with .ngr and .ngc extensions after the functional simulation [18].We implemented the Twine-80 algorithm using Xilinx ISE development tools on FPGA model Virtex-5 XC5VFX200T. The results and resource utilization is presented in Table 7. This hardware implementation illustrates that the algorithm can be implemented with very small area requirements (practically 131 number of slices and 419 number of Look Up Tables).

5. Conclusion

In this paper, we have described, synthesized and implemented a lightweight block cipher Twine, which has 64-bit block and 80 or 128- bit key. Twine's algorithm is a lightweight flexible algorithm for using in RFID systems and wireless sensor networks (WSNs) and deployment capabilities in many other applications as well. Due to the flexibility of the algorithm used in various applications and environments, data and key length can be selected with different sizes. In Twine-80, due to having too few resources as shown in our implementation results, it is very suitable for lightweight application and embedded systems.

References

[1] CD Cannière, O Dunkelman, and M Kneževi´c. KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. *Springer-Verlag, [In CHES 2009, Lecture Notes in Computer Science]* 2009; 5747: 272–88.

[2] T Suzaki, K Minematsu, S Morioka, and E Kobayashi. TWINE: A Lightweight, Versatile Block Cipher. *NEC Corporation, 1753 Shimonumabe, Nakahara-Ku, Kawasaki, Japan*, 2012.

[3] K Shibutani, T Isobe, H Hiwatari, A Mitsuda, T Akishita, and T Shirai. Piccolo: An Ultra-Lightweight Blockcipher. *Sony Corporation 1-7-1 Konan, Minato-ku, Tokyo, Japan* 2011; 108-0075.

[4] M Zwolinski. Digital System Design with VHDL. *2ndEdition* 2004.

[5] GD Micheli. Synthesis and optimization of digital circuits 1994.

[6] E Vahedi, VWS Wong, IF Blake. An Overview of Cryptography. *Chapter 5, University of British Columbia, Canada* 2014.

[7] R Beaulieu, D Shors, J Smith, ST Clark, B Weeks, L Wingers. The simon and speck of lightweight block ciphers. *National Security Agency*

9800 Savage Road, Fort Meade, MD 20755, USA 2013.

[8] P Coussy, A Morawiec. High Level Synthesis from Algorithm to Digital Circuit. *Springer* 2008.

[9] D Hong, J Sung, S Hong, J Lim, S Lee, B Koo, C Lee, D Chang, J Lee, K Jeong, H Kim, J Kim, S Chee. HIGHT: A new block cipher suitable for low-resource device. *CHES'06, LNCS 4249, Springer-Verlag* 2006; 46–59.

[10] G Leander, C Paar, A Poschmann, K Schramm. New lightweight DES variants. *FSE'07 LNCS 4953, Springer-Verlag* 2007; 196–210.

[11] A Bogdanov, L Knudsen, G Leander, C Paar, A Poschmann, M. J. B. Robshaw, Y Seurin, C Vikkelsoe. PRESENT: An ultra-lightweight block cipher. *CHES'07, LNCS 4727, Springer-Verlag* 2007; 450–466.

[12] FIPS. Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication 197*.

[13] FIPS. Data Encryption Standard. *Federal Information Processing Standards Publication 46*.

[14] M Hell, T Johansson, A Maximov, W Meier. The Grain family of stream ciphers. *New Stream Cipher Designs, The eSTREAM Finalists, LNCS 4986, Springer* 2008; 179–190.

[15] CD Canniere, B Preneel. Trivium. *New Stream Cipher Designs, The eSTREAM Finalists, LNCS 4986, Springer* 2008; 244–266.

[16] T Isobe. A single-key attack on the full GOST block cipher. *FSE'11, LNCS 6733*, 2011; 290–305.

[17] AJ Menezes, PC Van Oorschot and SA Vanstone. Handbook of Applied Cryptography. *CRC Press* 2001; 202-203.

[18] Xilinx, Inc. Virtex-5 User Guide, available at www.xilinx.com.

[19] T Eisenbarth, S Kumar, C Paar, A Poschmann and L Uhsadel. A Survey of Lightweight Cryptography Implementations. *IEEE Design & Test of Computers*m 2007; 24(6): 522-533.

[20] W Zhang, Z Bao, D Lin, V Rijmen, B Yang. RECTANGLE: A Bit-slice Ultra-Lightweight Block Cipher Suitable for Multiple Platforms. *State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, P. R. China and KU Leuven, ESAT/COSIC and iMinds; Kasteelpark Arenberg 10, Leuven, Belgium* 2014.

[21] M Kumar, SK Pal and A Panigrahi. FeW: A Lightweight Block Cipher. *Scientific Analysis Group, DRDO, Delhi, INDIA, Department of Mathematics, University of Delhi, INDIA* 2014.