CHAPTER 2

HIGH PERFORMANCE ARCHITECTURAL DESIGN OF
SECURITY PRIMITIVES

2.1 INTRODUCTION

This chapter explains the architectural design, implementation and performance of the security primitives Tiny Encryption algorithm (TEA), Extended Tiny Encryption algorithm (XTEA) and Rivest Cipher (RC5) used for the design of adaptive crypto system. The security primitives are chosen based on their size of operated data P and key K. The architectural design of the security primitives TEA, XTEA and RC5 handle a 128 bit key K and 64 bit plaintext P. The TEA and XTEA primitives are iterated for 32 rounds and RC5 for 12 rounds.

The multiplexer based, structural and sub program based architectural designs are discussed for TEA and XTEA primitive. The multiplexer based, pipelined and sliding window architectures are discussed for RC5 primitive. The architectural design was coded in Very High Speed Integrated Circuit Hardware Descriptive Language (VHDL) and implemented in Xilinx ISE 10.1. The performance analysis of the security primitives has been carried out. Throughput, power, energy efficiency and device utilization are the performance metrics used for evaluation.
2.2 ARCHITECTURAL DESIGN OF TEA PRIMITIVE

In TEA primitive, 64 bit plaintext P is separated into two halves, namely M0 and M1, 128 bit key K is divided into four sub keys, namely K[0], K[1], K[2] and K[3]. The TEA primitive iterates for 32 rounds with the round function F, applied to one half of the data. Using the sub keys, the output is XORed with the other half. Each round, follows the same pattern and the two halves are then fed back as an input for the next round function. The constant delta $\partial$, with the value 9E3779B9h is the derivative of the golden number ratio, used to distinguish the sub keys.

The encryption of the right and left half of the data M0 and M1 are given in equation 2.1 and 2.2, the round function F is given in equation 2.3.

\[
M0[i] = M0[i-1] \oplus F(M1[i-1], K[0,1], \partial[i]) \tag{2.1}
\]

\[
M1[i] = M1[i-1] \oplus F(M0[i-1], K[2,3], \partial[i]) \tag{2.2}
\]

\[
F = ((M << 4) \oplus K[j]) \oplus (M \oplus \partial[i]) \oplus ((M >> 5) \oplus K[k]) \tag{2.3}
\]

where $i$ refers to the iteration of the round function F, “$\oplus$” refers to XOR operation, “$\oplus$” refers to 32 bit addition, “$\ll$” refers left shit and “$\gg$” refers right shift.

The TEA primitive is designed using different architectures namely multiplexer based, structure based and sub program based designs.

All these architectures execute the round function F described in equation 2.1. The hardware for the round function F of the TEA primitive is executed for 32 rounds. After every round function F, the M0 and M1 values are updated as given in equation 2.2 and 2.3. The TEA primitive uses the scheduled keys K[0] – K[3] and plaintext P as illustrated in figure 2.1, to deliver ciphertext C.
Design of Round Function F:

The algorithmic steps for the round function F consists of two half rounds and shown in figure 2.2. The half rounds are identical in its operation with the difference in the subkey operands. The halfround 1 operates on the subkey $K[0]$ and $K[1]$, whereas the halfround 2 uses the subkey $K[2]$ and $K[3]$. 

Step 1:

The M0 of the plaintext $P$ is left shifted by 4 times and right shifted 5 times. The left shift on M0 is executed as shown in equation 2.4. The VHDL representation of M0 stored in a temporary variable $t0$, is given by equation 2.5, where ‘:=’ denotes a variable assignment operator in VHDL. The shift operations are executed as simple concatenation operations for optimization as given in equation 2.4 and 2.5.

\[
M0 := M0[5-32] \ & \ "0000" 
\]  
\[
t0 := M0(5-32) \ & \ "0000"
\]
Step 2:

The execution of right shift on M0 is shown in equation 2.6 and VHDL representation is shown in equation 2.7, where \( t_1 \) is the temporary variable which holds the current value of M0.

\[
M_0 := \text{"00000"} \& M_0[1-27] \quad (2.6)
\]

\[
t_1 := \text{"00000"} \& M_0 \text{ (1 to 27)} \quad (2.7)
\]

Step 3:

The left shifted block is added with the subkey \( K[2] \) and right shifted block is added with the subkey \( K[3] \). The resultant value is added with the constant delta value delta \((i-1)\), where \( i \) represent the number of iterations.

Figure 2.2 Round function architecture of TEA primitive
Step 4:

The results of addition are XORed and added with M1, the other half of the plaintext P. This operation updates M1 for next iteration of the round function F.

Step 5:

Similar operations are performed for the next half round function with the above result using subkey K[0] and K[1]. Finally, the XORred results are added with M0, to update M0 for the further rounds. The dotted lines in figure 2.2 indicates the feedback wired on M0(i) and M1(i).

The execution of shift, add and delta operations are performed only on the local variables. The shift and add operations are executed in parallel to produce the intermediate results. The shift operation with M0 and M1 are executed as simple concatenation operations for optimization.

2.2.1 Multiplexer based architectural design of TEA primitive

A multiplexing logic is used to select the inputs from a user or the previous round. The multiplexer logic selects, either the updated M0, M1 after round function F or data from the plaintext P as in figure 2.3. The intermediate output, after every round operation is C1 to C32 with C_i = M0(i) & M1(i), where & refers to the concatenation operation.

The status of the encryption is checked with the iteration operator ‘i’ and the control signal ‘n’. For the control signal n = 0, the multiplexer selects the plaintext P as the input, and delivers the intermediate output C1. For the control signal n = 1, it selects the previous output for the next 31 iterations and delivers C2 to C32.
The dotted lines in the figure 2.3 show the feedback, wired to the multiplexer. The number in the dotted line indicates the width of the data bus. At the end of the 32\textsuperscript{nd} iteration, the round function F block of figure 2.1, outputs C\textsubscript{32}. The right and left half of C\textsubscript{32} undergo concatenation operation to produce ciphertext C of 64 bit.
2.2.2 Structural design of TEA primitive

The architecture of the round function \( F \) in figure 2.2 is designed and executed as a separate component. The round function \( F \) is programmed as a component in VHDL and instantiated 32 times with port mapping technique. The 32 iterations are performed with 32 instances of the round function \( F \). This is referred as structural modeling which is used to facilitate hierarchy and abstraction in 32 rounds of TEA. These instances of component are represented as \( \text{Comp}_i \) \((1 < i < 32)\) with each \( \text{Comp}_i \), performing a round function \( F \).

The top module is designed by port mapping round function for 32 times with the synchronized key, clock and plaintext \( P \) is fed to \( \text{Comp}_1 \) as shown in figure 2.4. The components \( \text{Comp}_2 \) to \( \text{Comp}_{32} \) operate on the intermediate output from the previous round.

An intermediate ciphertext, which is the output of previous round is applied as an input for the next instance, and the output from the last instance \( \text{Comp}_{32} \) is taken as a complete ciphertext \( C \) of TEA encryption process.

![Figure 2.4 Structural design of TEA primitive](image-url)
The input plaintext $P$ propagates to output as ciphertext $C$, traversing 32 rounds of TEA encryption process. The propagation delay of input path to output path is 32 times of round function execution and is added with the wiring delay between the instances.

### 2.2.3 Sub program based design of TEA primitive

A subprogram is a set of declarations and statements that are repeatedly invoked in a sequential manner with VHDL description. They use a set of generalized operators whose data type, must conform to the types in the subprogram declaration. The subprogram procedure call, performs computations in sequential manner and return values in the form of global objects or stores the return values into formal parameters.

This method uses the subprogram procedure to perform the 32 TEA round functions. The design of this method is shown in the figure 2.5, where the TEA round function $F$ is presented, the procedure calls using subprogram method.

![Figure 2.5 Subprogram based design of TEA primitive](image)

Figure 2.5 Subprogram based design of TEA primitive
The interface information, local variable declaration to store M0, M1 and K are declared in the procedure declaration with their types. The subprogram procedure call accepts the inputs key K, plaintext P and the $i$ value to compute the cipher text C. The body of the procedure call comprises a set of lines to execute round function F repeatedly for $i$ times, with the intermediate outputs of the round function F being updated. Upon the completion of the 32$^{nd}$ round function, the output $C_{32}$ is considered as the complete cipher text C.

2.2.4 Data flow representation for multiplexer based architectural design of TEA primitive

To make functional realization more feasible, Hierarchical/Code Data Flow Graph (H/CDFG) model is used. It is a representation of the flow of data on a system by modeling its architectural aspects.

H/CDFG is primarily used to create an overview of the system to visualize the data that process through a structured design. It is used to visualize the operation, accomplishment and implementation of the architectures of the system. It describes the data flow and control transfer in the code often used in many design approaches, to reduce time to design.

If a design is to be converted into H/CDFG model, then it has to have three basic fundamental elements as nodes namely processing nodes, conditional nodes and memory nodes. The processing nodes, process the data with various arithmetic and logical operations. The processing operations are performed on a local I/O, global I/O or with a local data. The memory nodes are used to transfer data, to and from the memory. Conditional node are the conditions that are imposed on a loop for test operations.

The inputs and outputs to the H/CDFG are referred as global I/O as the data moves in and out of an external environment. The input to different CDFG establishes communication across H/CDFGs. These inputs and outputs are denoted as local I/O,
are given as the inputs to the respective sub graphs. A local data is used to store intermediate data used in the processing stage.

The H/CDFG is a composition of CDFG and Data Flow Graph (DFG). It capsules the entire operation of the design, which can be executed in parallel or sequential manner. A CDFG consists of test/branching conditions and loops. Both prediction and speculation type of loops are handled in CDFG. A DFG is a sequence of instructions with no conditions. A DFG has only memory nodes and processing elements.

The multiplexer based architectural design of TEA is viewed in H/CDFG pattern is given in figure 2.6. The global I/O, local I/O and the local data are identified for H/CDFG. The global I/O of TEA architecture are the plaintext P and key K. The local I/O are the M0, M1 and sub keys K[0], K[1], K[2] and K[3]. The intermediate results C1 to C32 fetched from the execution of round function F are identified as local I/O, as it updates M0 and M1. The constant delta and the count operator are the constant memory nodes.

A single H/CDFG is replicated to number of rounds of encryption process. The global I/O M0, M1, and the sub keys K[0], K[1], K[2] and K[3] are the inputs to the H/CDFG. The DFG is constructed using two types of memory nodes namely local I/O, local data. The M0 and M1 are processed in the round operations by storing the updated M0 and M1 values in the temporary registers t0, t1 and t2.

The DFG of multiplexer based architecture is constructed with half round functions, which has no conditional assignments. The half round function performs the encryption operation on the received data using three different variables t0, t1, t2. The intermediate results of shifting and adding operations are stored in these variables. The variable t3 stores the result of the half round function. The value of t0, t1, t2 and t3 are stored in the registers as local data for round operations of TEA primitive. The values obtained in the round functions are then fed back to perform the subsequent rounds.
Figure 2.6  H/CDFG for multiplexer based architectural design of TEA primitive
The CDFG sub graph is constructed with local data and one conditional IF branch with a DFG. After the completion of the DFG with one set of input, the IF condition checks the status of the multiplexer select signal ‘n’. If \( n = 1 \), then it processes the intermediate output as local I/O to perform the round function, else, the control moves to the DFG with a new set of plaintext \( P \) spilt to \( M_0 \) and \( M_1 \).

### 2.2.5 Simulation result of multiplexer based architectural design of TEA primitive

The TEA primitive architectures are designed and coded in VHDL and simulated in ModelSim 10.2C tool. The simulation of multiplexer based TEA architecture for a 64 bit plaintext \( P \) and 128 bit key \( K \) is shown in figure 2.7 and output ciphertext \( C \) is obtained after 32 clock cycles.

It is observed from the output that the multiplexer select line ‘n’ is set to ‘1’ for the round 2 to 32. The output of TEA primitive is derived for every plaintext after 32 clock cycles. The multiplexer based TEA architecture is executed for the plaintext \( P=8765432187654321h \) and Key \( K=12345678123456781234567812345678h \) respectively.

The values of the intermediate outputs \( C_1 - C_{32} \) are represented in hexadecimal for convenience and are tabulated in table 2.1. The intermediate output \( C_{32} \) after 32 iterations of round function \( F \) is the ciphertext \( C \). The ciphertext \( C \) of the plaintext \( 86765432187654321h \) is \( C216B7C96F3513E1h \).
Figure 2.7 Simulation of multiplexer based architectural design of TEA primitive
Table 2.1  Round function output of TEA primitive

<table>
<thead>
<tr>
<th>$C_i$</th>
<th>Round Function Output in hex representation</th>
<th>$C_i$</th>
<th>Round Function Output in hex representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>1F3C D803 BA90 10A4</td>
<td>$C_{17}$</td>
<td>526E 47DC 67FD 1BEA</td>
</tr>
<tr>
<td>$C_2$</td>
<td>74FF 8A56 812E E57E</td>
<td>$C_{18}$</td>
<td>52FF C87F 8C17 F93C</td>
</tr>
<tr>
<td>$C_3$</td>
<td>DDC9 BBA8 D0B1 81FC</td>
<td>$C_{19}$</td>
<td>E210 C60D 1642 9E64</td>
</tr>
<tr>
<td>$C_4$</td>
<td>2A44 B807 D78C B637</td>
<td>$C_{20}$</td>
<td>F831 3D18 EE7E 1678</td>
</tr>
<tr>
<td>$C_5$</td>
<td>A6F1 CB1C FFCA 7D88</td>
<td>$C_{21}$</td>
<td>02E8 258E 9E06 2FBF</td>
</tr>
<tr>
<td>$C_6$</td>
<td>48EB 9EFE 4A78 750E</td>
<td>$C_{22}$</td>
<td>D661 95C6 AD93 0A11</td>
</tr>
<tr>
<td>$C_7$</td>
<td>79BB 12C3 C151 4082</td>
<td>$C_{23}$</td>
<td>EEAC 76B6 6E8D 27B1</td>
</tr>
<tr>
<td>$C_8$</td>
<td>0636 C371 5185 50A4</td>
<td>$C_{24}$</td>
<td>9C1B 75EA 243D 802E</td>
</tr>
<tr>
<td>$C_9$</td>
<td>E568 3BD1 5641 5930</td>
<td>$C_{25}$</td>
<td>6F0D EE08 1904 A237</td>
</tr>
<tr>
<td>$C_{10}$</td>
<td>CC2B CE24 8C73 5C3F</td>
<td>$C_{26}$</td>
<td>0932 EB68 C6D0 4850</td>
</tr>
<tr>
<td>$C_{11}$</td>
<td>6354 4A27 09FE E17A</td>
<td>$C_{27}$</td>
<td>1B2C DDE1 E3D9 88DB</td>
</tr>
<tr>
<td>$C_{12}$</td>
<td>3AD0 57E4 184C E299</td>
<td>$C_{28}$</td>
<td>82A1 FD62 DE87 9040</td>
</tr>
<tr>
<td>$C_{13}$</td>
<td>DFBA 855E 14B2 17D2</td>
<td>$C_{29}$</td>
<td>ABF7 9019 3D00 A13E</td>
</tr>
<tr>
<td>$C_{14}$</td>
<td>D3F1 10BC 4821 1BF1</td>
<td>$C_{30}$</td>
<td>DD9A 654E D7E1 CB44</td>
</tr>
<tr>
<td>$C_{15}$</td>
<td>E143 87D3 61F2 24A5</td>
<td>$C_{31}$</td>
<td>65D6 4C0F CC55 E20A</td>
</tr>
<tr>
<td>$C_{16}$</td>
<td>42C0 0F33 6E3A DA3F</td>
<td>$C_{32}$</td>
<td>C216 B7C9 6F35 13E1</td>
</tr>
</tbody>
</table>
2.3 MULTIPLEXER BASED ARCHITECTURAL DESIGN OF XTEA PRIMITIVE

The XTEA is a feistel, iterative and symmetric security primitive. It is an extended and advanced version of TEA that is resistant to relative key attacks and uses 128 bit key K and 32 bit half blocks M0 and M1. Like TEA, XTEA operates on invariant 64 bit plaintext and 128 bit key K. The key schedule and the mathematical functions are slightly rearranged to make it more complex when compared to TEA.

The XTEA is designed with the multiplexer based architecture. The XTEA encryption of left and right half of the data are given in equations 2.8 and 2.9 and the round function F is given in 2.10.

$$M0[i] = M0[i-1] \oplus F(M1[i-1], K[delta(i-1) & 3], delta[i-1])$$ \hspace{1cm} (2.8)

$$M1[i] = M1[i-1] \oplus F(M0[i-1], K[((delta(i-1)>>11) & 3)], delta[i])$$ \hspace{1cm} (2.9)

The round function F, is defined as

$$F = ((M \ll 4) \oplus (M \gg 5)) \oplus M) \oplus ((K[j]) \oplus delta[i-1]) \oplus M)$$ \hspace{1cm} (2.10)

The round function F of XTEA primitive is shown in figure 2.8 is implemented for one round of XTEA encryption. The difference between the single round function in TEA and XTEA is the mixing of keys K[0-3].

The plaintext P into M0 and M1 is the input only for the first round. At the end of the single round operation, M0 and M1 are updated and, the updated values are wired as the input for the next round. The multiplexer based architectural design for TEA/XTEA in figure 2.2 is used to select the inputs either from the user or from the previous round. The multiplexer control logic selects updated M0 and M1 after round function execution or data from the plaintext P as in figure 2.2.
2.3.1 Simulation result of multiplexer based architectural design of XTEA primitive

The simulation of XTEA is performed for the plaintext P and key K in ModelSim 10.2C tool. Simulation results for the given plaintext $P = 86765432187654321h$ and key $K = 12345678123456781234567812345678h$ in the hex format is shown in figure 2.9. The simulated output for the 32 rounds of round function F are tabulated in table 2.2 and the ciphertext C produced is $DFE8223E299A2689h$. 

Figure 2.8 Round function architecture of XTEA

![Figure 2.8 Round function architecture of XTEA](image)
Figure 2.9 Simulation of multiplexer based architecture of XTEA primitive
Table 2.2  Intermediate output from Round function of XTEA primitive

<table>
<thead>
<tr>
<th>$C_i$</th>
<th>Round Function Output in hex representation</th>
<th>$C_i$</th>
<th>Round Function Output in hex representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>ABA2 90F4 EDB1 D302</td>
<td>$C_{17}$</td>
<td>5597 FE79 9DC0 21D8</td>
</tr>
<tr>
<td>$C_2$</td>
<td>25EE D27F B98E D50A</td>
<td>$C_{18}$</td>
<td>3AE7 5420 7653 4B93</td>
</tr>
<tr>
<td>$C_3$</td>
<td>3DF8 8577 AA49 B07B</td>
<td>$C_{19}$</td>
<td>29AA DFA5 8C2D 1EF4</td>
</tr>
<tr>
<td>$C_4$</td>
<td>DEC0 BE84 ECF6 41DF</td>
<td>$C_{20}$</td>
<td>AC3A D93D A9BB E5AE</td>
</tr>
<tr>
<td>$C_5$</td>
<td>1EAC 8B05 0E9A 29F7</td>
<td>$C_{21}$</td>
<td>D261 8CD4 A7E8 050D</td>
</tr>
<tr>
<td>$C_6$</td>
<td>EFE6 FA28 3D1C 041E</td>
<td>$C_{22}$</td>
<td>01C7 D174 5CAA 3C6E</td>
</tr>
<tr>
<td>$C_7$</td>
<td>BAAB F93F 4CCA 15B1</td>
<td>$C_{23}$</td>
<td>8FCF F0A3 1E0B 5C9B</td>
</tr>
<tr>
<td>$C_8$</td>
<td>38D5 85CE 134E 688D</td>
<td>$C_{24}$</td>
<td>4730 9C88 7D12 B3F9</td>
</tr>
<tr>
<td>$C_9$</td>
<td>7D10 C62F 0120 9F96</td>
<td>$C_{25}$</td>
<td>EFE2 7E60 E9B7 01D3</td>
</tr>
<tr>
<td>$C_{10}$</td>
<td>2E16 C6FA 4F8D 56B9</td>
<td>$C_{26}$</td>
<td>F04E 1CF8 BA1E 70F0</td>
</tr>
<tr>
<td>$C_{11}$</td>
<td>3880 0766 6C60 4363</td>
<td>$C_{27}$</td>
<td>6DCE FB2D 497D 5DBD</td>
</tr>
<tr>
<td>$C_{12}$</td>
<td>27D1 EE4B 4B9C 948C</td>
<td>$C_{28}$</td>
<td>8AE4 9D2E 9EC3 9340</td>
</tr>
<tr>
<td>$C_{13}$</td>
<td>A3CE 3A1F 1266 236E</td>
<td>$C_{29}$</td>
<td>72BB BA9C 035A A21D</td>
</tr>
<tr>
<td>$C_{14}$</td>
<td>C620 E9D3 A5C2 9E35</td>
<td>$C_{30}$</td>
<td>3A32 714C 44F7 04CD</td>
</tr>
<tr>
<td>$C_{15}$</td>
<td>6E17 5513 5E2B D719</td>
<td>$C_{31}$</td>
<td>492C D531 2854 1E02</td>
</tr>
<tr>
<td>$C_{16}$</td>
<td>7723 D321 7B37 E4BB</td>
<td>$C_{32}$</td>
<td>DFE8 223E 299A 2689</td>
</tr>
</tbody>
</table>
2.4 ARCHITECTURAL DESIGN OF RC5 PRIMITIVE

The RC5 primitive is a symmetric security primitive, which uses the same key for both encryption and decryption. The RC5 primitive is fast enough, as it is word oriented, and the basic computational operations are on full words of data at any time. The number of bits ‘w’ in an input word and the number of rounds ‘r’ depends on the user requirements.

The RC5 primitive is adaptable to processors of different word length from 8 bit to 64 bit and could be conveniently implemented in FPGA. RC5 primitive has an option for the user to explicitly manipulate the trade of between higher speed and higher security. The RC5 architecture with low memory requirement is used on smart cards or other devices which requires stringent memory and power considerations.

2.4.1 Parameters used in the RC5 primitive

The parameters required for RC5 encryption are
- A and B are inputs of size w, with a total width of 2w bit where w is the word size
- K, the user defined input key of ‘b’ bytes used to encrypt data
- The number of rounds for encryption process is denoted by ‘r’
- S, the subkey array of length (2r-1), expanded using user defined key K.
- U and V are temporary registers between rounds to store partial outputs.
- P and Q are predefined constants.
- L is rearranged user key array with each element in array is of ‘c’ byte wide.

RC5 is denoted by $w / r / b$ where w is the word size, r is the number of rounds, b is the number of bytes in the key K. This work operates on 32 bit word size, 128 bit key K for 12 rounds, denoted as, $32 / 12 / 16$. 
2.4.2 Stages of RC5 operation

RC5 encryption consists of computations with plaintext P and key K. The logical and mathematical operations in the implementation of RC5 uses two’s complement addition of words modulo $2^w$ denoted by ‘+’, bitwise exclusive OR of words denoted by $\oplus$ and cyclic rotation of word denoted by $A <<< B$. The variable rotation operations play an important role in the encryption process which makes RC5 a data dependent primitive.

RC5 primitive has 2 stages namely

a. Key expansion
b. Encryption stage

a. Key Expansion Stage

The key expansion stage has three sub stages namely

a1. Conversion of key K from bytes to words
a2. S array Initialization
a3. Mixing the S array table with key K

a1. Conversion of key K from bytes to words

The user key is divided to equal set of ‘c’ byte wide and stored separately, where $c$ is $b/u$, $u$ is given by $w/8$. The least significant bits are followed by most significant bits.

In this work, $w = 32$, $u = 4$, $b = 16$, $c = 4$ and the user key $K$ is represented as $K[0]$, $K[1]$, $K[2]$ and $K[3]$ each 32 bits respectively. The conversion of 128 bit key $K$ into 16 bytes is represented in the conversion pseudo code below.
Conversion Pseudo code:

\[
c = \left[ \frac{\text{max}(b,1)}{u} \right]
\]

for \( i = b - 1 \) downto 0 to \( c \)
\[
L[i/u] = (L[i/u] << <) + K[i]
\]

RC5 constants:

The P and Q are the predefined constants, which are defined by the following expressions given in equation 2.11 and 2.12.

\[
P_w = \text{Odd } ((e-2)2^w)
\]

\[
Q_w = \text{Odd } ((\varnothing-1)2^w)
\]

where \( e \) = base of natural logarithm and \( \varnothing \) = golden ratio. The value of \( e \) is 2.718281828459… and \( \varnothing \) is 1.69033988749. The value of P and Q in hexadecimal are computed for \( w=32 \) are \( P_{32} = \text{b7e15163h} \) and \( Q_{32} = \text{9e3779b9h} \) respectively.

a2. S array initialization

The S array is an expanded sub key array obtained by filling the key table array \( S[0…t-1] \) with binary words. The ‘t’ value denotes the size of the table which depends on the number of rounds as given in equation 2.13.

\[
t = 2(r+1)
\]

In this work, \( r \) is chosen as 12, therefore \( t = 26 \) and the S array ranges from \( S[0] \) to \( S[25] \). The S array is initialized to P as \( S[0] = P \). The elements of the S array from \( S[0] \) to \( S[25] \) is computed using equation 2.14 and 2.15.

\[
S[0] = P_w ; 1<i<(t-1)
\]

\[
S[i] = S[i-1] + Q_w ; 1<i<(t-1)
\]
a3. Mixing the S array table with key K

This stage mixes the S array table with the key K. The S array table ranging from S[0] to S[25] are mixed with the user key K. The pseudo code for the mixing operation is given below, where A and B are the input plaintext P, t represents the number of elements in the S array table, c refers the number of words in the 128 bit key K and i, j represents the iterations ranging from 0 to 25 and 0 to 3 respectively.

Pseudo code:

\[
i = j = 0; \quad A = B = 0;
\]

\[
A = S[i] = (S[i] + A + B) \ll 3
\]

\[
B = L[j] = (L[j] + A + B) \ll (A + B)
\]

\[
i = (i + 1) \mod t
\]

\[
j = (j + 1) \mod c
\]

The figure 2.10 shows the above operations mixing the S array table elements S[0] to S[25] with the user key K.

![Figure 2.10 Mixing the key K with the S array](image)
b. Encryption stage

The encryption stage of RC5 is shown in figure 2.11 with two w-bit words denoted as \( A \) and \( B \). The arithmetic and logic operations on the \( A \) and \( B \) are given in equation 2.16 and equation 2.17 respectively.

Pseudo code:

\[
\begin{align*}
A &= A + S[0] \\
B &= B + S[1] \\
\text{for } i = 1 \text{ to } r \text{ do} \\
A &= ((A \oplus B) \ll B) + S[2i] \\
B &= ((B \oplus A) \ll A) + S[2i+1]
\end{align*}
\]

The output is in the registers \( A \) and \( B \). Computations are done on both \( A \) and \( B \). For every round operation, the intermediate outputs \( U \) and \( V \) are generated and updates the \( A \) and \( B \) for subsequent processing. The updated \( U \) and \( V \) serves as the input for the next round operation.

![Figure 2.11 Encryption stage of RC5 primitive](image-url)
2.4.3 Multiplexer based architectural design of RC5 primitive

The RC5 primitive handles 64 bit plaintext \( P \) and 128 bit key \( K \) for 12 rounds. The output of each round from \( C_1 \) to \( C_{32} \) is termed as an intermediate output. The \( A, B \) are the right and left half of the plaintext \( P \) which are the inputs to the encryption stage for the first round. After the execution of the round function, the \( A, B \) are replaced by the intermediate output \( C_i \) for further processing. The output of every round, updates the \( A \) and \( B \) of the encryption stage until \( r = 12 \).

In the multiplexer based architecture, a multiplexer is used to select the inputs either from the user or from the output of the previous round. The multiplexing logic directs the present input or previous output to execute the round function \( F \) of RC5. A round function \( F \), is executed 12 times and the registers \( U \) and \( V \) hold the output of round function \( F \). The bitwise XOR, circular shift and mod 2 addition are the sequence of operations in the round function \( F \). The sub key selection sub key(i) and subkey(i+1) is used at a round function \( F \). The round function operations update both \( U \) and \( V \) registers, as shown in figure 2.12.

![Figure 2.12 Updating U and V registers in the round function of RC5 primitive](image-url)
A round function F, is executed 12 times and the registers U and V hold the output of round function F. The sub keys generated for the 128 bit key K are stored in the memory. The multiplexers MUX1, MUX2 are used to select the sub key and the intermediate output as shown in figure 2.13.

A control signal ‘n’ selects either the updated U or the appropriate subkeys for the round function execution as shown in figure 2.12. MUX1 selects the value of U or V and MUX2 selects the subkey(i) or subkey(i+1) for encryption. The intermediate output in processing the round function F is stored in temporary variables Temp1, Temp2 and Temp3.

![Figure 2.13 Multiplexer based architecture of RC5 primitive](image)

The round function is executed once for every clock cycle. The intermediate output after round operation C1 to C12 is represented as $C_i = A(i) \& B(i)$, where the symbol ‘&’ denotes the bit wise concatenation operation. The 64 bit ciphertext C is obtained at an interval of 12 clock cycles. The ciphertext output is delivered for every 12 clock pulses.
2.4.4 H/CDFG representation for multiplexer based architectural design of RC5 primitive

Figure 2.14 shows the H/CDFG of multiplexer based architectural design of RC5. The plaintext P and the key K are considered as global I/O.
A round function in a H/CDFG is replicated 12 times to complete 12 rounds of encryption that needs to be performed. The registers Reg1-Reg11 are the temporary registers. These registers are called memory nodes, since they hold the partial results for encryption operation. The Reg1-Reg11 serves as the local I/O for the subsequent DFGs.

The next level is CDFG sub graph, which operates on the basis of a conditional IF branch. The conditional statement selects the input either from the user or from the partial results of the encryption process. The conditional statement is executed based on the control signal of the multiplexer. If the select line is false, then it selects the input A and B from the user. If the select line is true, then, it selects the U and V values updated from Reg1-Reg11 for subsequent processing.

On observation, only conditional and local data are found in the CDFG sub graph. Each DFG consists of XOR, shift and addition operations which are performed in the processing elements of the RC5 primitive. The results of each stage of processing is stored in the temporary registers temp1, temp2 and temp3.

### 2.4.5 Pipelined architectural design of RC5 primitive

Pipelining is an extension of parallel code execution concept which works within a single process. Instead of partitioning the process, pipelining technique achieves the functionality similar to parallel code execution by partitioning the program into smaller modules that executes over multiple times on a loop structure. The program is divided into discrete steps and the inputs and outputs of each step are wired to feedback nodes or shift registers in a loop.

A pipelined design with processing capabilities of the FPGA increases the efficiency of sequential code. The repeated number of operations on the same data are computed for a large number of iterations. Pipeline registers are inserted between operations that belong to consecutive stages of pipelining to decrease the delay.
The following terms are used to describe the pipeline architecture of RC5

Pipeline latency ($L_p$) is the length in clock cycles to compute the ciphertext using RC5 primitive. The clock cycles required to compute $C_{12}$ from $P$ and $K$ is 12. Therefore, $L_p = 12$.

Pipeline system data introduction interval ($P_{sdi}$) is the time interval, in clock cycles, between two consecutive input plaintexts that is made available to the RC5 primitive. The condition $P_{sdi} = 1$ indicates a fully pipelined system, the condition $L_p > P_{sdi} > 1$ indicates a partial pipelines system whereas a non-pipelined system holds the condition $P_{sdi} \geq L_p$. This work implements a fully piped architecture for RC5 for $P_{sdi} = 1$ to achieve high throughput.

Pipeline Stage ($P_s$) is the pipeline is divided into discrete pipeline stages that are $P_c$ cycles in length. The number of pipeline stages are computed as $P_s = L_p / P_c$. The pipeline stages are decided by the round functions executed. This work, executes one round of RC5 for every clock cycle. Hence, $P_c = 1$ and $P_s = 12 / 1 = 12$.

### 2.4.6 Implementation of pipelined RC5 architecture

In this work, a full piped RC5 is implemented with 12 dependent stages of computation as shown in figure 2.15. The RC5 architecture is modeled with a linear pipe, consisting of round function hardware connected in series to perform 12 rounds of encryption over a plaintext $P$. The efficiency of the 12 stage pipeline is achieved, if and only if the $P_{sdi}$ is 1. This is possible, only if the plaintext is made available for every clock cycle.

A full pipeline design of RC5 is executed with the number of pipeline stage $P_s$ is equal to the round $r$. The pipeline architecture is designed for $r = 12$, which is fully piped with pipeline stage $P_s = 12$ and system data introduction $P_{sdi} = 1$.

A minimum of 12 plaintext $P$ represented as $P = (P_1, P_2, \ldots P_{12})$ is stored in a two dimensional array, and the processing of input in the queue is scheduled with
First Come First Serve (FCFS). The intermediate outputs are stored in the temporary registers and feedback to the next stage for subsequent processing.

Each pipeline stage executes a round function in one clock cycle. The temporary registers outa to outl hold the value of the round function, with outa storing the result of the first stage, and outl storing the result of the last stage. The latency $L_p$ of the pipeline implementation is measured as the time interval taken for the intermediate output to traverse from outa to outl, and shown in figure 2.15. The dotted lines in the figure 2.15 indicate the dependent stages of RC5 primitive.

![Figure 2.15 Dependent stages of pipeline architecture of RC5](image)

The set of plaintext $P = \{P_1, P_2, \ldots, P_{12}\}$ for RC5 encryption using pipeline architecture and the corresponding ciphertext are termed as $C = \{C_1, C_2, \ldots, C_{12}\}$. The temporary registers outa-outl stores the intermediate ciphertext $C_1$ to $C_{12}$. The execution of 12 stage RC5 with respect to the clock is illustrated in figure 2.16. The register outa stores the first round of the RC5 operation and outl stores the
twelfth round of RC5 operation. The registers outa to outl are updated on every clock cycle.

An intermediate output is executed for every clock cycle in RC5 operation, as a case, the intermediate output of C1 in the RC5 execution is given as C11, C12, …, C112 with C112 = C1 is termed as the ciphertext of P1. The RC5 round function executes C21, C12 simultaneously at the second clock and C31, C22 and C13 at the third clock respectively. The twelfth clock cycle executes C121, C112, C103, C94, C85, C76, C67, C58, C49, C310, C211 and C112 as shown in figure 2.15. The signal outl stores the output of the C112.

Figure 2.16 Pipeline execution of RC5 primitive

The key schedule of the RC5 primitive handles 128 bit key K which is further expanded into subkeys. The sub key generation module is programmed as a package and included in the main program of RC5 primitive. The round function
operates on the plaintext P and the subkey of RC5. The subkey generation module is shared by all the 12 round function hardware as shown in figure 2.17.

![Pipeline round functions sharing subkey generation](image)

Figure 2.17 Pipeline round functions sharing subkey generation

The pipeline architecture increases the latency. The pipelined circuit has a longer latency due to the pipeline registers. However, the throughput of the pipelined circuit with $P_{\text{sd}} = 1$, is always greater than a non-pipelined circuit.

### 2.4.7 Sliding window based architectural design of RC5 primitive

The sliding window is a popular data link layer protocol widely used in the data transmission methods. The significance of the sliding window protocols is that, at any instant of time, the sending end maintains a sequence of blocks of data permitted to send. The blocks are said to appear within the input window. The input and the output window may or may not be maintained at the same size. If the window is of variable size, they can grow or shrink over the course of time as the input plaintext blocks are sent and received.

The efficiency of the pipelined architecture depends on the interrupted availability of the input plaintexts. If the plaintext is unavailable to the RC5 for every clock cycle, then, $(1/P_s)^{\text{th}}$ of the computations in the clock is unutilized with the result
of reduced average throughput. In this work, the plaintext P for the pipelining architecture resides in the sliding window for its processing in the RC5 core to sustain the efficiency of the pipeline architecture.

The proposed sliding window architecture of RC5 primitive is used to ensure $P_{\text{rdi}} = 1$ in the pipelined RC5 for uninterrupted pipeline operation. If there is no window, then the data to be encrypted will be waiting in memory for the RC5 core requirement. The direct assessment of the plaintext from the user to the RC5 core would eventually result in a data loss or register overflow leading to memory overflow. Also, there could not be any feature to differentiate the original and retransmitted data which have actually missed the deadline, as there is no track of the plaintext that has been picked up by the RC5 core.

The advantages of introducing a sliding window before RC5 core are

- Data scheduling
- Synchronization between data packets
- Reduction of waiting time
- Multiprocessor environment

The size of the sliding window, $W$ is chosen to be 4 packets. The Address Generation Unit (AGU) provides the address for the input plaintext. With AGU, the plaintext block is associated with an address. The input First In First Out (FIFO) acts as a sliding window for availing the scheduling of data on to the core by using sliding window.

At every clock cycle, the plaintext P enters into the input sliding window. The number of plaintext to be accommodated in the sliding window depends on the size $W$ of the sliding window. Figure 2.18 illustrates the condition in which, the input sliding window is filled with plaintext P1, P2, P3 and P4 for 4 clock cycles.
The plaintext P1 enters the RC5 primitive with FIFO scheduling. After RC5 execution, the ciphertext C1 corresponding to the plaintext P1 appears at the output sliding window.

Figure 2.18 RC5 Sliding window for window size $W=4$

The plaintext P1 associated with the address from the AGU is tagged as an input signal called iRAM_plain (input plaintext). This signal enters the input sliding window by setting the i_FIFO_En (input FIFO enable) for the first clock cycle. In the second clock cycle, the plaintext P2 is loaded into the sliding window corresponding to the address from which it fetched and P1 is sent it to the RC5 core on enabling Encrypt_En (Encrypt enable) signal.

After processing, the RC5 primitive delivers the ciphertext C1 and it is sent to the output sliding window. The output window holds the encrypted data tagged as oRAM_plain (ciphertext) signal from the RC5 core. It also stores this signal to an addressable location in the output window. This is done on o_FIFO_En (output FIFO enable) for synchronization as shown in figure 2.19.
With respect to the processing environment, the input sliding window length $W$, determines the length of a ready queue. A windowless FIFO would make the packet served in first come first serve so that the last arriving data or the task has to starve for the processor time. The inclusion of window hardware holds only a defined number of packets based on its window length. The packets arrive at the window are selected based on their deadlines. This leads to earliest deadline first schedule on the data packets arrival to RC5 core. The window expands when the data flow is more and shrinks when it is less.

### 2.4.8 Simulation of RC5 primitive

The RC5 architectures are designed and coded in VHDL using ModelSim10.2C and implemented in Xilinx 10.1 ISE. The simulation of multiplexer and pipeline architecture of RC5 for a 64 bit plaintext $P = \text{8765432112345678h}$ and 128 bit key $K = \text{1234567812345678123456781234567812345678h}$ is shown in figure 2.20 and 2.21.
Both the pipeline and sliding window architectures have pipeline technique, and the output is continuous after 12 clock cycles, yielding 64 bit ciphertext $C$ at every clock cycle. The simulated output for the 12 rounds of RC5 is shown in figure 2.20 and the intermediate outputs are tabulated in table 2.3.

Table 2.3 Intermediate output from Round function of RC5 primitive

<table>
<thead>
<tr>
<th>$C_i$</th>
<th>Round Function Output in hex representation</th>
<th>$C_i$</th>
<th>Round Function Output in hex representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>94EE 285B 4B66 F63E</td>
<td>$C_7$</td>
<td>1900 FE81 C589 01D6</td>
</tr>
<tr>
<td>$C_2$</td>
<td>BEF8 BF3C B1E4 980B</td>
<td>$C_8$</td>
<td>83CD 390B 34DD 67E5</td>
</tr>
<tr>
<td>$C_3$</td>
<td>D7BD B2B6 E013 D833</td>
<td>$C_9$</td>
<td>83BA F43C 6148 E72A</td>
</tr>
<tr>
<td>$C_4$</td>
<td>0006 512A CFCA D9A0</td>
<td>$C_{10}$</td>
<td>1E98 768F 5889 570F</td>
</tr>
<tr>
<td>$C_5$</td>
<td>3B5D C2AE 81EB 3556</td>
<td>$C_{11}$</td>
<td>5FE2 39A6 8480 53A1</td>
</tr>
<tr>
<td>$C_6$</td>
<td>B944 C0DD 0B9D AE38</td>
<td>$C_{12}$</td>
<td>53D9 FDAC 78C5 498D</td>
</tr>
</tbody>
</table>

The simulated output for the 12 rounds of RC5 with window size $W=4$ for the input in table 2.4 is shown in figure 2.22. The plaintext is fed on each clock pulse as $P_{\text{sd}}=1$. The set of plaintext inputs in the input sliding window for RC5 execution are listed in table 2.4.

Table 2.4 Input for RC5 execution set for $P_{\text{sd}}=1$

<table>
<thead>
<tr>
<th>Input</th>
<th>Plaintext</th>
<th>Input</th>
<th>Plaintext</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1234567887654321</td>
<td>P3</td>
<td>8765432187654321</td>
</tr>
<tr>
<td>P2</td>
<td>1234567812345678</td>
<td>P4</td>
<td>8765432112345678</td>
</tr>
</tbody>
</table>
Figure 2.20 Simulation of multiplexer based architectural design of RC5 primitive
Figure 2.21 Simulation of pipeline architectural design of RC5 primitive
Figure 2.22  Simulation of sliding window based architectural design of RC5 primitive for window size $W=4$
2.5 PERFORMANCE ANALYSIS OF PROPOSED ARCHITECTURES

The proposed TEA, XTEA and the RC5 architectures are coded in VHDL, simulated in ModelSim 10.2C and implemented in Xilinx XC4Vfx12sf363 target device. The parameters obtained from the resource utilization summary and the timing report from the Xilinx Synthesis report for the proposed architectures of TEA, XTEA and RC5 are tabulated in table 2.5.

The operating frequency, slices used, power and delay are the parameters measured with the throughput, throughput per slice, energy and energy efficiency are the performance metrics calculated from the measured values.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>TEA</th>
<th>XTEA</th>
<th>RC5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplexer Based</td>
<td>Structural</td>
<td>Sub program</td>
<td></td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>145.9</td>
<td>132.27</td>
<td>111.45</td>
</tr>
<tr>
<td>No. of Slices</td>
<td>233</td>
<td>6510</td>
<td>3964</td>
</tr>
<tr>
<td>Period (ns)</td>
<td>6.85</td>
<td>7.56</td>
<td>8.927</td>
</tr>
<tr>
<td>Logic Delay (ns)</td>
<td>4.563</td>
<td>5.57</td>
<td>5.96</td>
</tr>
<tr>
<td>Routing Delay (ns)</td>
<td>2.287</td>
<td>1.99</td>
<td>2.9</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>179.2</td>
<td>403.4</td>
<td>273.2</td>
</tr>
</tbody>
</table>

The throughput of the encryption is evaluated [15] as the number of ciphertext bits delivered per second and throughput per slice is computed as the throughput obtained from a single slice of the target device.
Energy is computed as the product of power and the time taken to compute the ciphertext C. The energy efficiency is evaluated as the number of bits delivered at the output per Joule of energy. The throughput, throughput / slice, energy and energy efficiency of the proposed TEA, XTEA and RC5 primitives are listed in table 2.6.

<table>
<thead>
<tr>
<th>Performance Metrics</th>
<th>TEA</th>
<th>XTEA</th>
<th>RC5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multiplexer Based</td>
<td>Structural</td>
<td>Sub program</td>
</tr>
<tr>
<td>Throughput (Mbps)</td>
<td>291.97</td>
<td>264.54</td>
<td>224.03</td>
</tr>
<tr>
<td>Throughput/Slice(Mbps)</td>
<td>1.253</td>
<td>0.0406</td>
<td>0.0506</td>
</tr>
<tr>
<td>Energy (nJ)</td>
<td>39.2</td>
<td>97.5</td>
<td>78.04</td>
</tr>
<tr>
<td>Energy Efficiency (Gbits/J)</td>
<td>1.6</td>
<td>0.6</td>
<td>0.8</td>
</tr>
</tbody>
</table>

It is observed that multiplexer based architectural design of TEA primitive uses lesser number of slices, delay and power compared to the structure based and subprogram based TEA architectures. The throughput of multiplexer based TEA architecture is 10.3% greater than structure based architecture and 30.3% greater than the sub program based architectural designs.

The power consumed in multiplexer based architecture of TEA primitive is 2.2 times smaller than the structure based architecture and 1.5 times smaller than the sub program architectures of TEA primitive. The energy efficiency of multiplexer based architecture of TEA is 2.6 times greater than the structure based and two times greater than the subprogram based TEA architectures. The high energy efficiency of the multiplexer based TEA architecture is achieved as the multiplexer based architecture require 59.8% lesser energy than the structure based and 49.7% lesser than subprogram based TEA architectures.
The XTEA architecture consumes 7.7% more slices than the multiplexer based architecture of TEA. The throughput of multiplexer based TEA architecture is 5.1% greater than the multiplexer based architecture of XTEA. The energy efficiency of multiplexer based XTEA architecture is only 3.78% lesser than the multiplexer based architectural design of TEA primitive.

The performance of the RC5 architectures is analyzed and the pipeline architecture shows high throughput compared to the other architectures. The device utilization of the pipeline architecture of RC5 is 0.004% greater compared to the multiplexer based architecture of RC5. The throughput of pipeline is 8.95 Gbps which is 13.1 times greater than the multiplexer based architecture and 1.23 times greater than the sliding window architecture of RC5.

The latency $L_p$ of the pipeline architecture of RC5 primitive for 12 rounds is measured as 85.8ns. The energy efficiency of pipeline architecture of RC5 is 16.2% higher than the multiplexer based architecture and 51.6% greater than sliding window architecture of RC5 primitive.

The slices utilized by the multiplexer based architecture of RC5 primitive are less compared to other architectures. However, the throughput of pipeline architecture of RC5 is 13.1 times greater than the multiplexer based architectural design of RC5.

The multiplexer based architectural design of TEA primitive consumes less device utilization compared to the architectures of XTEA and RC5 primitive. The throughput of pipeline architecture of RC5 is high compared to the TEA and XTEA primitives.
2.5.1 Comparison of proposed architecture with other reported architectures

The architectures of the three primitives TEA, XTEA and RC5 are designed and implemented in Xilinx Virtex XC4Vfx12sf363 device and shown in table 2.7. The proposed architecture has also been implemented on the target devices used in [18], [21] and [19] and the results are compared. The slices consumed and the throughput of the proposed XTEA architecture is compared with the other works.

Table 2.7 Comparison of XTEA with reported architectures

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XC 4Vfx12sf363</td>
<td>JC†</td>
<td>JC*</td>
<td>JC**</td>
</tr>
<tr>
<td></td>
<td>3S2000-5</td>
<td>4x2fg256</td>
<td>4x25f0668</td>
<td></td>
</tr>
<tr>
<td>Slices</td>
<td>251</td>
<td>264</td>
<td>258</td>
<td>263</td>
</tr>
<tr>
<td>Throughput (Mbps)</td>
<td>277.77</td>
<td>159.7</td>
<td>215</td>
<td>252</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>19</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>85</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>73.3</td>
</tr>
</tbody>
</table>

Note: † - Target device used in [18], *- Target device used in [21] and ** - Target device used in [19]

The multiplexer based architectural implementation of XTEA primitive is compared for its performance with the XTEA architectures reported in [18], [21] and [19]. The slices reported for multiplexer based architecture of XTEA in this work has 5.6%, 14.6% and 76.7% smaller device area, when compared to methodologies discussed in the reference [18], [21] and [19] respectively. The throughput of the proposed multiplexer based architecture of XTEA is 14.6, 3.26 and 3.78 times greater than the work reported in references [18], [21] and [19] respectively.

The multiplexer based architectural design of XTEA is implemented for its performance in the target devices reported in the references [18], [21] and [19]. The throughput of the multiplexer based architecture is 8.4, 2.5 and 3.43 times greater than the architectures discussed in [18], [21] and [19] for the same target devices.

The proposed multiplexer based, pipeline and sliding window RC5 architectures are compared with the existing RC5 implementations. The proposed RC5
architectures consume less number of slices when compared to other works as shown in table 2.7.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>XC 4Vfx-12sf363</td>
<td>XC‡ 5vtx30/50</td>
<td>XC* 2v100-4fg</td>
<td>XC** 2v250fg</td>
</tr>
<tr>
<td>Slices</td>
<td>885</td>
<td>994</td>
<td>1022</td>
<td>2488</td>
</tr>
<tr>
<td>Throughput (Mbps)</td>
<td>8951</td>
<td>650.4</td>
<td>555</td>
<td>613</td>
</tr>
</tbody>
</table>

*Note: † - Target device used in [23], *- Target device used in [25] and ** - Target device used in [24]*

The slices used in proposed pipeline based architecture of RC5 primitive is 64.4%, 75.5% and 8.5% smaller than the methodologies discussed in the reference [23], [25] and [24] respectively. The throughput of the proposed pipeline architecture of RC5 is 16.6, 29.8 and 4.2 times greater than the work reported in [23], [25] and [24] respectively.

The pipeline based architectural design of RC5 is implemented for its performance in the target devices reported in the references [23], [25] and [24]. The throughput of the pipeline based architecture is 16.1, 28.5 and 4.10 times greater than the architectures discussed in [23], [25] and [24] respectively for the same target devices. Also, this work uses 90% lesser LUTs when compared to the design in [49].

### 2.6 SUMMARY

TEA, XTEA and RC5 security primitives and their operation was studied. The architectures are designed, simulated and implemented in Xilinx 10.1 ISE tool. The target FPGA device used in the implementation is XC4Vfx12-12sf363. The multiplexer based, structural and sub program architectures are analyzed for TEA primitive. The throughput of multiplexer based TEA architecture is 10.3% greater than structure based architecture and 30.3% greater than the sub program based architectural designs. Hence, multiplexer based architecture has been chosen for XTEA primitive.
implementation, which is an extended version of TEA. The multiplexer based, pipeline and sliding window architectures are proposed for a 12 round RC5 primitive. The throughput of pipeline is 8.95 Gbps, which is 14.1 times greater than the multiplexer based architecture and 1.23 times greater than the sliding window architecture of RC5. The proposed architectures of TEA, XTEA and RC5 are compared with the existing works for the performance and design constraints.