

# SPECIALIZED COPROCESSOR FOR IMPLEMENTING THE RC4 STREAM CIPHER

Krishnamurthy Koduvayur Viswanathan, Kunal Narsinghani, Varish Mulwad  
Department of Computer Science and Electrical Engineering  
University of Maryland Baltimore County  
{krishna3, kunal2, varish1}@cs.umbc.edu

*Abstract – We have proposed a design for an efficient co-processor for implementing the RC4 stream cipher. The co-processor is intended to be used in wireless routers. The following report describes the detailed analysis of the RC4 implementation, design decision justifications and performance measures to showcase the achieved speedup and improved throughput.*

## 1. Introduction

Wireless routers allow you to connect to a network from anywhere without using a cable. To secure the communication that takes place over the network, wireless routers use protocols such as the Secure Sockets Layer (SSL) and Wired Equivalent Privacy (WEP). To provide communication privacy both SSL and WEP encrypt the messages exchanged over the network using the RC4 stream cipher. We describe the design of a specialized co-processor that implements the RC4 stream cipher which is intended to be used in Wireless routers.

In the remaining sections, we briefly explain how the RC4 stream cipher works and give a summary of our design.

### 1.1 RC4 Stream Cipher

RC4[1] is a variable-key-size stream cipher developed in 1987 by Ron Rivest for RSA Data Security, Inc. For seven years it was proprietary, and details of the algorithm were only available after signing a nondisclosure agreement.

RC4 is simple to describe. The algorithm works in Output Feedback Mode: The keystream is independent of the

plaintext. It has an  $8 * 8$  S-Box:  $S_0, S_1 \dots S_{255}$ . The entries are a permutation of the numbers 0 through 255, and the permutation is a function of the variable-length key. It has two counters,  $i$  and  $j$ , initialized to zero.

The S-Box is first initialized. The S-Box is filled linearly:  $S_0 = 0, S_1 = 1 \dots S_{255} = 255$ . Then another 256 byte array is filled up with the key, repeating the key until the array fills up. Then the following pseudo code is executed to complete the initial setup phase –

*for  $i = 0$  to 255:*  
 *$j = (j + S_i + K_i) \bmod 256$*   
*swap  $S_i$  and  $S_j$*

Once the initial setup is complete the key stream is generated byte by byte. To generate the keystream byte the following pseudo code is executed –

*$i = (i + 1) \bmod 256$*   
 *$j = (j + S_j) \bmod 256$*   
*swap  $S_i$  and  $S_j$*   
 *$t = (S_i + S_j) \bmod 256$*   
 *$K = S_t$*

% time	Cumulative seconds	Self Seconds	Self Calls	Self ms/call	Total ms/call	Function
50	0.01	0.01	1009163	0.01	0.01	Swap
50	0.02	0.01				main
0	0.02	0	3943	0	2.54	arc4
0	0.02	0	1	0	2.54	PermuteSBox
0	0.02	0	1	0	0	SetupBoxes

**Figure 1:** Flat profile of the application. % time represents the percentage of total running time of the program used by this function. Cumulative seconds is a running sum of the number of seconds accounted for by this function and those listed above it. Self seconds is the number of seconds accounted for by this function alone. This is the major sort for this listing. Calls represents the number of times this function was invoked. Self ms/call is the average number of milliseconds spent in this function per call. Total ms/call is the average number of milliseconds spent in this function and its descendants per call.

The byte K is XORed with the plaintext to produce ciphertext or XORed with the ciphertext to produce plaintext.

### 1.2 Summary of the Proposed Design

We have proposed a design of a specialized co-processor that will implement the RC4 stream cipher. The co-processor takes two inputs –input key and the plaintext/ciphertext to be encrypted/ decrypted and produces the ciphertext/plaintext as output. After profiling the software implementation of RC4, we took several design decisions such as use of a fast dual port SRAM, use of fast adder. We have also designed the system in such a way, that multiple independent tasks can start simultaneously thus saving time required for the final encrypted / decrypted output to be produced. These decisions helped us to achieve significant speedup over various hardware implementations of RC4 we compared with.

## 2. Characteristics of the Application

We wrote an implementation of the RC4 cryptographic algorithm in the C programming language and compiled it using the GNU C compiler on the Ubuntu Linux platform [3].

We analyzed the characteristics of this application for the purpose of this project.

We ran three profiling tools on the application: SimpleScalar’s sim-outorder [4], GNU gprof, Valgrind suite’s callgrind [5]. The information yielded by SimpleSim was not too useful to us since it was very specific to the SimpleScalar architecture and our implementation is that of a specialized co-processor.

Thus we turned to gprof and valgrind to determine how much time the program spent in each function, and how many times each function was called. This would concisely tell us which functions burn most of the cycles.

Figure 1 shows a typical call graph for the application. As seen, the Swap function takes up a major chunk of the processing time. This is evident from the fact that the Swap function is used during the initial key initialization phase, as well as during the stream generation phase. Figure 2 shows that the Swap function is called by the arc4 as well as PermuteSBox functions. This is enough information to warrant a design that would improve the performance of the content swap of the SBox. Since this is a frequently invoked portion of the application, we expect a significant overall performance improvement in the system by improving the performance of this module.

In addition to this, we observed that a

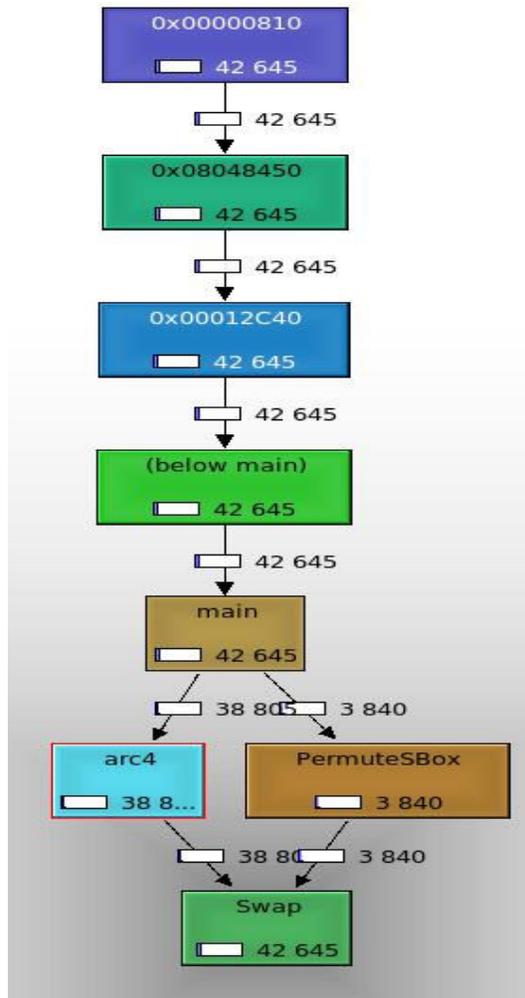


Figure 2: A callgraph visualization generated by KCachegrind

considerable portion of the execution time was being spent in the arc4 function. This function is called every time a block of the input plaintext is read in. The arc4 method internally calls the Swap function and also performs a series of additions. Similar additions are also performed during the key setup phase. Hence we hope to gain improvement in performance by reducing the time taken for these additions.

### 3. Design Description

#### 3.1 Components

Figure 3 shows us the design of the co-processor implementing the RC4 algorithm. It consists of the following components:

- i. K-Box
- ii. S-Box
- iii. Three eight bit adders
- iv. 2:1 Multiplexer
- v. Five eight bit registers
- vi. Two eight bit counters (i and j)

**i) K-Box:** The K-Box is used to store the key to be used in the encryption/decryption process in the RC4 algorithm. The K-Box consists of a fast 256 bytes SRAM. The input bus to the K-Box is 64 bits wide and the output bus from the K-Box to the adder is 8 bits wide. From Figure 6, it is known that the read/write times for the SRAM in the K-Box are 4.4 nanoseconds. The 64 bit key can be loaded into the SRAM in 4.4 nanoseconds. During the Key setup phase, the key value at the address *i* is read is passed on to the adder via an 8 bit bus. The K-Box is used only during the Key setup phase which happens when the router is booted for the first time.

**ii) S-Box:** The S-Box is used during both the initial Key setup phase and the Keystream generation phase. The S-Box consists of a fast 256 bytes SRAM and a 256 bytes ROM. The ROM is used to store the 256 byte linearly arranged S-array. Whenever the Reset signal to the S-Box goes high, the S-array from the ROM is transferred into SRAM of the S-Box. The input and the output buses connected to the S-Box are each 8 bit wide.

The SRAM in the S-Box is a dual ported SRAM. The dual ported RAM allows us to

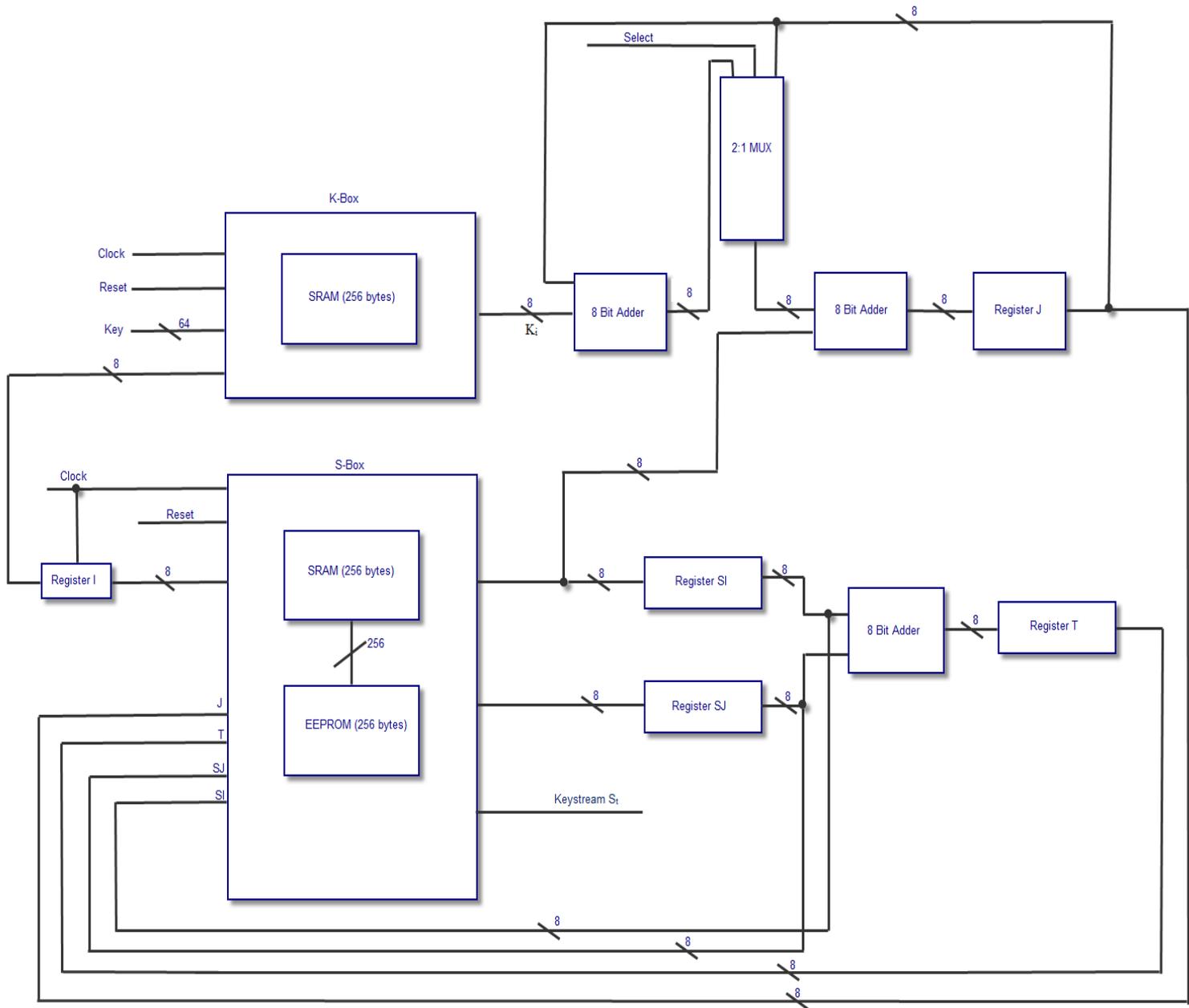


Figure 3: Our proposed design of the Keystream generator block in the RC4 stream cipher co-processor

read or write two memory locations simultaneously. In Section 2, we described the characteristics of our application in which it was observed that the Swap function takes majority chunk of the time. The use of a dual ported SRAM helps us

significantly reduce the time required for swapping. In the section following this we have compared the performance of our system using a single port SRAM and a dual ported SRAM. The analysis proves the same – use of dual ported SRAM helps us improve performance.

iii) *8-bit Adders*: We have three 8-bit adders in our system. We looked at the existing adders in the market viz. BK (Breton Kung) [13], PPrefix [14], Carry Lookahead [15], and Ripple Carry. Most of these adders have a critical path delay of approximately 160-180 ps. However, since the adders perform the main computation in our system, we looked for a better adder. JS Lee and DS Ha [6] propose a High Speed 1-bit Bypass Adder which is 2 times faster than the regular 8 bit adders. We use this specialized 8 bit adder in our design which has a maximum critical path delay 84 ps.

iv) *2:1 Multiplexer*: We have a 2:1 multiplexer which selects between the outputs of the first adder and the contents of the Register J to be given as the second input to our second adder. During the key setup phase, the output of the first adder is given as an input to the second adder. During the stream generation phase, the MUX gives the contents of register J as an input to the second adder.

v) *8-bit Registers*: We use two eight bit registers for  $S_i$  and  $S_j$ . These registers are used during the swap operation to store the values of  $S_i$  and  $S_j$  respectively.

vi) *8-bit counters (i and j)*: We use two 8-bit registers for  $i$  and  $j$ . The value in register  $i$  is incremented every cycle.

### 3.2 Operation

The operation of the system is divided into three phases. During the first phase, the RESET signal is given to the S-Box which linearly initializes the 256 byte SRAM. Thus we have  $S[0] = 0, S[1] = 1 \dots S[255] = 255$ . This linearly filled 256 byte array is transferred from the ROM to the SRAM. The two counters are initialized to 0 as well. Next we have the Key setup phase. In our system, the input key to the RC4 coprocessor is a 64 bit long (40 bit key and 24 bit initialization vector). The key is loaded linearly into the K-Box SRAM and repeated as many times as necessary to fill up the 256 bytes. In this phase, the S-box is modified according to the

pseudo code shown in section 1.1. Note that during the key initialization phase, the setup logic does not have to wait for all the 256 bytes

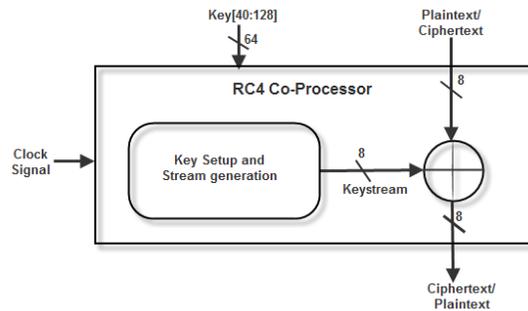


Figure 4: Block diagram of the working of the co-processor

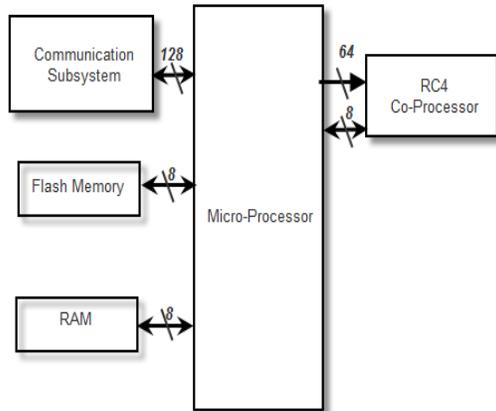
of the KBox RAM to be initialized before starting the operation. The bytes of the KBox RAM are accessed linearly in order; hence in our design, the bytes that have already been initialized can be used by the adder even while the rest of the bytes are still being loaded. The registers  $S_i$  and  $S_j$  are used along with the dual port SBox RAM to swap the values of  $S_i$  and  $S_j$  respectively. Once the values of  $S_i$  and  $S_j$  for a particular iteration are stored in the registers, these values get written to positions  $j$  and  $i$  respectively in the SBox RAM. Since we have a dual ported RAM, these two values can be read from and written into the RAM in a single step. This phase goes through 256 iterations before the SBox RAM is completely initialized. The third phase is the stream generation phase (see pseudo code in section 1.1). For every byte of the plaintext given, one iteration is completed, which generates the output stream. This output stream is EXOR'd with the plaintext to generate the ciphertext.

### 3.3 Overall system

As shown in Figure 4, the clock signal with frequency (22.41 MHz) is input to the RC4 coprocessor, along with the variable length key. The coprocessor then produces the keystream(8 bits at a time) which is Ex-ORed with the plaintext to produce the cipher text. The same key is used in the decryption to produce the plaintext.

Component	Latency	Read Time	Write Time
Adder	0.08 ns[6]	-	-
XOR	1.5 ns[8]		
2:1 Multiplexer	0.2 ns[9]	-	-
SRAM		4.4 ns per byte[10]	4.4 ns per byte[10]
8 – bit Register		0.1 ns	0.1 ns
Bus (per byte)	1.96 ns[13]	-	-

**Figure 6: Values for components used in the design**



*Figure 5: Functioning of RC4 Coprocessor within a wireless router.*

The placement of our co-processor for RC4 within a wireless router is as shown in Figure 5: The microprocessor interacts with the RC4 co-processor we have designed.

Communication functions involving data are performed by the communication subsystem, which passes data to the micro-processor on the 128 bit bus. Flash memory is used by the microprocessor to store operating system software. Other temporary applications may be loaded into the volatile RAM.

The plain text would be an input to the communication subsystem, which will be passed on (8 bits at a time on the 8 bit bus) along with 8 bits of the key (on the 64 bit bus) to the co-processor and the ciphertext(8 bits) is returned to the micro-processor.

## 4. Justifications and Analysis

To capture the performance of our proposed design, we have used values from Figure 6. These values are the standard and publicly available values for the various components

used in our design (see references). For performance analysis, we decided to use a standard WEP 64 bit implementation. In this implementation, the key is 40 bits long, to which the WEP adds a 24 bit Initialization vector of 24 bits to make the key 64 bits long.

### 4.1 Performance of the Proposed Design

*Time spent in the Key setup phase:* First we calculate the time required to complete a single iteration in the Key setup phase. All the iterations in the key setup phase will require the same time as the first iteration. Using the values from Figure 6, we have calculated that a single iteration in the key setup required is **30.19 nanoseconds**. The key setup phase starts its operation immediately after the first eight bits of the key are available in the K-Box. The number of iterations in the key setup phase is 256. Therefore the total time required in the key setup phase in our design will  $30.19 \times 256 = 7728.64$  **nanoseconds**.

This is the one time initial cost required to setup the co-processor when the router is booted for the first time.

*Time spent in the Key stream generation phase:* The Key stream generation phase can begin only after the key setup phase is complete. We calculate the time required to generate the first eight bits of the key stream. The time required to generate subsequent eight bits of the key stream require the same time as the first eight bits took. Using the values from Figure 6, we have calculated that the time required to generate the first eight bits of the key stream is **42.6 nanoseconds**. Every subsequent eight bits of key stream also require 42.6 nanoseconds.

*Time required encrypt/decrypt one byte of plaintext/ciphertext:* The first byte of the plaintext can be encrypted once the first byte (or the first eight bits) of the key stream arrive at the XOR. From values in Figure 3, we know that an EXOR operation will take 1.5 nanoseconds. Thus the time required to encrypt one byte of plaintext is  $42.6+1.5=44.62$  nanoseconds.

#### 4.2 Speedup comparison

We computed the speedup of our proposed system using the following method:

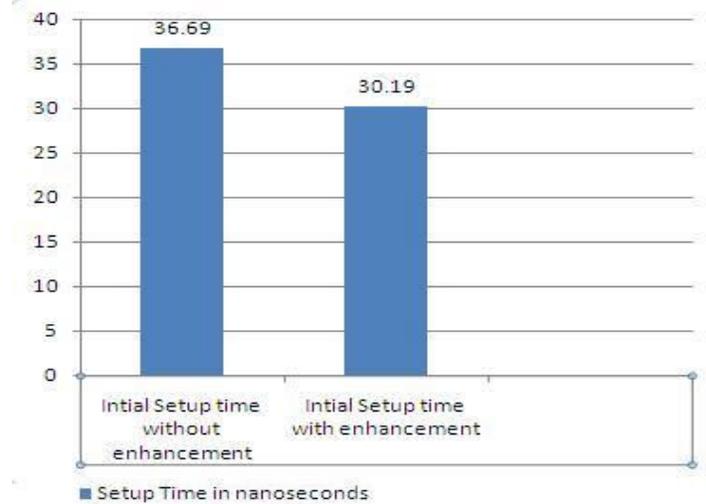
The time required to encrypt a single byte of plaintext is 44.62 nanoseconds as per section 4.1. The throughput of the Key Stream Generation phase (not taking the setup time into consideration, since it is a one time cost) is thus  $1/44.62 \text{ ns} = 22.41 \text{ MB/s}$ . We compared our enhanced implementation with other [7] hardware implementations of the algorithm for WEP, where the maximum throughput (after setup) was 5MB/s.

$$\begin{aligned} \text{Speedup} &= \frac{\text{Performance enhanced RC4}}{\text{Performance conventional RC4}} \\ &= 22.41/5 = 4.48. \end{aligned}$$

Thus, using our enhanced version, the encryption/decryption would run **4.48 times faster**.

#### 4.3 Impact of the enhancements on the overall design

Further, we analyzed the effects of our application specific design decisions on the overall architecture of the co-processor:



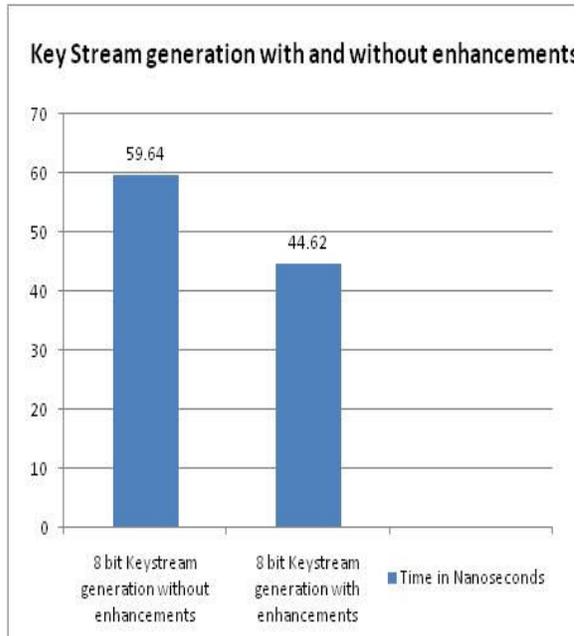
**Figure 7: Comparison of time required for Key setup phase with and without the use of proposed enhancements**

##### 4.3.1 Impact on the key-setup phase:

As shown in the figure above, we calculated the time required for a single iteration of the Key Setup phase without the dual ported RAM and the enhanced adders. We compared this figure with the key setup time obtained with the proposed enhancements. The figures indicate a speedup of 12.15% for the key setup phase.

##### 4.3.2 Impact on the stream generation phase:

Similar to the case shown above, we computed the time required to generate a byte of the keystream without the dual ported RAM and the enhanced adders. We compared this figure with the time required to generate the keystream with our proposed enhancements. The results are as shown below:



**Figure 8: Comparison of time required for Key stream generation with and without the use of proposed enhancements**

The figures show that the keystream generation takes 44.62 ns with the enhancements as opposed to 59.64 ns without the enhancements. This is a speedup of 13.36% for the stream generation process

## 5. Conclusions and future work

In this project, we have presented an efficient implementation of the RC4 algorithm to be used as a specialized coprocessor in a wireless router. We have followed an analytical approach for evaluating the system performance and speedup. The proposed system obtained a speedup of 4.4 as compared to the implementation in [7]. As future work, we would like to validate our design by simulating this architecture using VHDL and fabricating it on an FPGA chip.

## 6. References

[1] B.Schneier, "Applied Cryptography - Protocols, Algorithms and Source Code in C", Second Edition, John Wiley and Sons, New York, 1996.

[2] W.Stallings, "Cryptography and Network Security", Third Edition, Prentice Hall

[3] ARC4 C implementation: <http://cs.umbc.edu/~krishna3/611/myArc4.txt>

[4] SimpleScalar: <http://www.simplescalar.com/>

[5] The Valgrind instrumentation framework: <http://valgrind.org/>

[6] Jong-Suk Lee and Dong Sam Ha, "High Speed 1-bit Bypass Adder Design for Low Precision Additions", *Circuits and Systems, 2007. ISCAS 2007IEEE International Symposium*

[7] Panu Hämäläinen, Marko Hännikäinen, Timo Hämäläinen, and Jukka Saarinen "Hardware Implementation of the Improved WEP and RC4 Encryption Algorithms for Wireless Terminals", *Proceedings of 10th European Signal Processing Conference*

[8] Texas Instruments: 74AC11086 QUADRUPLE 2-INPUT EXCLUSIVE-OR GATE <http://focus.ti.com/lit/ds/symlink/74ac11086.pdf>

[9] Fairchild semiconductor NC7SB3257 2:1 Multiplexer/Demultiplexer Bus Switch <http://www.fairchildsemi.com/ds/NC%2FNC7SB3257.pdf>

[10] NEC Electronics, Data Sheet, MOS INTEGRATED CIRCUIT  $\mu$ PD4482163, 4482183, 4482323, 4482363, 8M-BIT CMOS SYNCHRONOUS FAST SRAM: <http://www.necel.com/nesdis/image/M14904EJ4V0DS00.pdf>

[11] Bus Bandwidth: The PC Guide: <http://www.pcguide.com/ref/mbsys/buses/func/Bandwidth-c.html>

[12] Wireless Router System and Method US Patent No 7010303 B2

[13] R.P. Brent and H.T. Kung, "A Regular Layout for Parallel Adders", *IEEE Trans. Computers*, Vol. 31, Mar. 1982, pp. 260-264

[14] G. Dimitrakopoulos, D. Nikolos, "High-Speed Parallel-Prefix VLSI Ling Adders", *IEEE Trans. Computers*, Vol. 54, No. 2, Feb. 2005, pp 225-231

[15] N.H.E Weste, K. Eshraghan, *Principles of CMOS VLSI Design*, Addison Wesley, 1993