Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 63 (2015) 8 – 15

Procedia Computer Science

The 6th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2015)

Efficient FPGA Implementation of the RC4 Stream Cipher using Block RAM and Pipelining

Eyad Taqieddin*, Ola Abu-Rjei, Khaldoon Mhaidat, Raed Bani-Hani

Faculty of Computer and Information Technology
Jordan University of Science and Technology, Irbid 22110, Jordan

Abstract

RC4 is a popular stream cipher, which is widely used in many security protocols and standards due to its speed and flexibility. Several hardware implementations were previously suggested in the literature with the goal of improving the performance, area, or both. In this paper, a new hardware implementation of the RC4 algorithm using FPGA is proposed. The main idea of this design is the use of a dual-port block RAM in the FPGA in order to better utilize the available logic and memory resources. Combined with a new pipelined hardware implementation, the new design achieves better performance. The design is described using Verilog HDL and synthesized and implemented using Xilinx ISE suite for different FPGA devices. Synthesis results show that the proposed design achieves higher efficiency than previous implementations by reducing area while maintaining a good throughput/LUT ratio. The proposed design is also more efficient in terms of power consumption.

Peer-review under responsibility of the Program Chairs

Keywords: Cryptography; RC4 stream cipher; Pipelining; Block RAM; Throughput; Area; Power; FPGA.

1. Introduction

Stream ciphers are symmetric key ciphers that combine plaintext digits with a pseudorandom key stream to generate an output cipher stream. They are also called state ciphers because they contain a secret internal state, which is used to generate the pseudorandom key stream. In the encryption process, the key stream is usually combined with the plaintext by applying a bit-wise exclusive-or operator. At the receiver side, the same key stream is generated and is used to decrypt the cipher back to the original plaintext.

* Corresponding author. Tel.: +962-2-7201000 Ext. 22557.
E-mail address: eyadtaq@just.edu.jo

Peer-review under responsibility of the Program Chairs
doi:10.1016/j.procs.2015.08.306

Stream ciphers differ from block ciphers that operate on large (usually fixed-size) blocks of data. They are generally easier to implement, have a lower hardware complexity, and are mostly faster than block ciphers. Moreover, they are resistant to error propagation as each unit (bit or byte) is independently encrypted. For these reasons, stream ciphers are more efficient for real-time processing and communication and have been the choice of many cryptosystems.

One popular stream cipher is RC4, which is fast with low complexity and lower area compared to other popular stream ciphers[1]. In fact, RC4 has been the most popular stream cipher in the history of symmetric key cryptography[2]. It is still being used in many security protocols and standards such as WEP, WPA, SSL and TLS as well as in embedded systems. Statistics have shown that RC4 algorithm is actually used in protecting 50% of the TLS traffic which is arguably the most widely used secure communication protocol on the Internet today[3].

As with any stream cipher, RC4 has a secret internal state and works by generating a pseudorandom stream of bits. The secret internal state consists of two parts: an array of 256 bytes, called the S-box, and three 8-bit index-pointers, denoted i, j, and k.

The RC4 algorithm consists of two parts: the key scheduling algorithm (KSA) which is used for initializing the S-box using a variable length key and the pseudo-random generation algorithm (PRGA) for generating the key stream bytes. The pseudo codes for both algorithms are described below

Algorithm 1: Key Scheduling Algorithm[4]

```
Input: Key, key_length

Output: Permuted S-box (S)

1. Initialize S = [0,1, …, 255], j = 0
2. for i = 0, …, 255 do
      a. update j =( j +S[i]+key[i] mod key_length) mod 256
      b. swap S[i] with S[j]
3. end for
4. return S
```

Algorithm 2: Pseudo-Random Generation Algorithm[4]

```
Input: S-box (S)

Output: Key stream

1. Initialize i = 0, j = 0
2. While (more bytes to encrypt) do
      a. update i = i+1, j =( j+S[i]) mod 256
      b. swap S[i] with S[j]
      c. update k = (S[i]+S[j]) mod 256
      d. output S[k]
3. end while
```

The KSA algorithm takes as an input a variable-length key and starts by initializing the S-box array to the identity permutation, then it runs for 256 iterations where two entries of the S-box are swapped at each iteration resulting in a randomized S-box. The PRGA algorithm consists of the same steps of the KSA with two differences. First, to calculate the new value of the index j only the old value of j and S[i] are used, whereas in the KSA algorithm the key is also used in the calculation. Second, in the PRGA algorithm S[i] and S[j] are used to calculate k, a variable which is used as in index to the output byte S[k]. Each byte generated (S[k]) is exclusive-ored with a byte of the plaintext to generate a byte of the ciphertext. The decryption process follows the same process except that the ciphertext is exclusive-ored with the generated key stream bytes.

RC4 is mostly implemented in software but there were several proposed approaches for high-performance hardware implementations of the algorithm. These implementations serve the increasing demand on high-speed

secure communications and secure embedded systems. With their programmable logic components, FPGAs give designers the flexibility to implement various types of algorithms, often with a performance remarkably better than software-based implementation.

In this paper, we introduce a new hardware implementation of RC4 based on the use of block RAM and pipelining to achieve high-throughput together with a reduction in the area. The proposed design achieves a maximum of 62.09 MB/sec with a significant improvement in the throughput/lookup table (LUT) ratio compared to other existing designs.

The rest of the paper is organized as follows: Section 2, discusses previous designs followed by a description of the proposed design in Section 3. The experimental results and comparisons are given in Section 4. Finally, the paper is concluded in Section 5.

2. Related work

One of the first hardware implementations of RC4 was proposed by Hamalainen et al.[5]. The S-box was implemented using a 256-byte RAM with asynchronous read and synchronous write operations. In addition to the S-box module, two separate state machines were implemented: one for initializing the S-box and the other for encryption. This implementation requires eight clock cycles in order to produce one byte of the key stream. Moreover, the authors attempted to improve their results by combining the two state machines into one single unit. On one hand, the number of reserved CLB's was reduced. On the other hand, the throughput was lower due to the slight decrease in maximum frequency.

Kitsos et al.[6] proposed another hardware implementation of RC4 that supports variable key length from 8 bits to 128 bits. The proposed design produces one byte of the key stream every three clock cycles. This is done using pipelined hardware and three 256-byte RAM blocks to implement the S-box. The implementation requires 3 cycles per byte in the KSA which means a total of $3 \times 256 = 768$ cycles followed by three cycles per byte in the PRGA phase. That is, the implementation needs $768 + 3n$ clock cycles for full execution of the algorithm, where n is the number of bytes in the plaintext/ciphertext.

Rourab et al.[7] provided a high throughput implementation in which one byte of key stream is generated at each clock cycle by utilizing the rising and falling edges of each clock cycle to execute two sequential tasks. In addition, they utilized a MUX-DEMUX combination to swap the S-box contents directly. This implementation did not make use of the block RAM and instead used latches to implement the S-box which resulted in a large area.

RC4A[8] and RC4B[9] were proposed to enhance the security of RC4 by increasing the internal complexity of the algorithm. A hardware implementation of RC4A was proposed by Al Noman et al.[10] using Altera APEXTM 20K200E FPGA. The implementation supports variable key length from 8 bits to 512 bits. Because RC4A uses two instances of RC4, the proposed implementation consumes nearly double the area. However, two output bytes are produced per iteration which increases the algorithm throughput.

Another variant of RC4 which aims to increase its security is the modified RC4[11]. The implementation removed key correlation and other biases in KSA and canceled the relations between the states of S-boxes. The results show that this architecture achieved a throughput of 63.449 Mbps regardless of the key length.

The previously mentioned designs were implemented using FPGA technologies. Other implementations that used ASIC technologies. Lee and Fan[12] stated that the major process of RC4 algorithm is to shuffle the S array. Therefore, they investigated two different architectures for shuffling (permuting) the S-box module: a dual-port 256×8 memory design and a single-port 128×16 memory design. The implementation of the former architecture resulted in less latency and power consumption at the expense of area. Moreover, the implementation of the latter architecture improved the shuffling latency by 25% compared to the conventional single-port 256×8 architecture. Gupta et al.[13] proposed two high-performance hardware designs for the RC4 algorithms using pipelining and loop unrolling to achieve high-speed implementation. The first design used loop unrolling and achieved a throughput of one byte per clock cycle while the second implementation was able to produce two bytes per clock cycle by using hardware pipelining along with loop unrolling used in the first design.

3. The proposed design

FPGAs provide two types of RAM: Distributed RAM and Block RAM. To implement a distributed RAM, LUTs are used. This would consume part of the LUTs available for logical or computational tasks. Therefore, the proposed design is based on using block RAM for better utilization of the logical and memory resources in the FPGA.

Increasing the throughput of the algorithm can be achieved by increasing the number of key stream bytes produced per one clock cycle. Using asynchronous read operations and multiple-port S-box can increase the throughput to one byte per clock cycle. However, these features are not supported for block RAM in FPGA and require the use of distributed RAM which means consuming more LUTs. Our design has a throughput of one byte per three clock cycles but it reduces the required area significantly.

The proposed design was described in Verilog HDL and synthesized using the Xilinx XST tool[14], which has the ability to infer RAM modules in FPGA devices. In order to implement a block RAM, we need to consider the RAM inference capabilities supported by Xilinx XST. There are two main conditions that need to be considered for the tool to infer a dedicated block RAM:

- Only synchronous read operations may be used.
- A maximum of two read/write ports can be used (or one write-port and multiple read-ports).

To successfully implement the dual-port RAM, the design considers that signals i and k share one port of the RAM while signal j is connected to the other port. Both ports are used for read and write operations.

The proposed implementation is shown in Fig. 1. The K-box module takes the key and the key_length in bytes. This K-box module generates a new byte every iteration of the KSA stage, which represents key[i mod key_length]. This value is used to calculate j through the KSA stage. In the PRGA stage, the key bytes are not used in updating the value of j. For that reason, the key_set signal is asserted such that the multiplexer passes a value of 0, instead.

The design contains four 8-bit registers j_reg, k_reg, S[i]_reg, and S[k]_reg. Each of these registers is connected to a common clock and has a write_enable signal that enables writing on the register at the appropriate clock edge.

As discussed before, the S-box module is implemented using dual-port RAM. The implementation of the dual-port RAM connections is shown in Fig. 2.

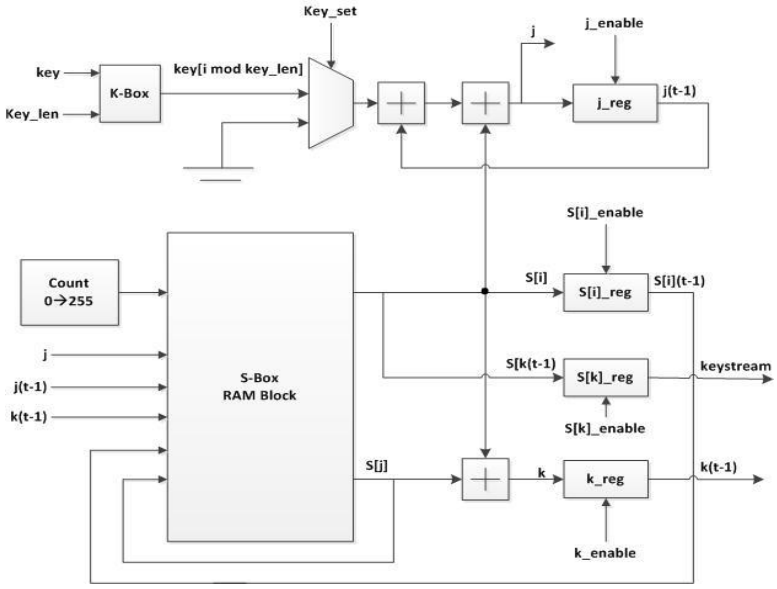

Fig. 1. Proposed architecture for RC4.

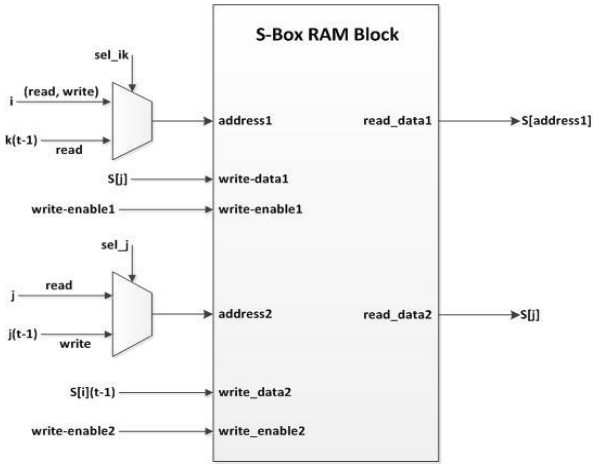

Fig. 2. Dual-port RAM connections.

An important technique that was applied in this design is the use of pipelining to improve the algorithm's speed. Two registers are used for the storage of k and S[k] which allows for the reading of the produced byte while processing the next one. Three cycles are needed for the production of one byte.

A byte of the key stream is generated each iteration which consists of three clock cycles. The operations performed at each clock cycle are as follows:

Cycle 1:

- The value of index i is updated (incremented by 1).
- The signal "sel_ik" is set to select i as the read index of the S-box so that "S[i]" is available at the "read_data1" on the next clock edge.
- Starting from the second iteration, the entries of "S[i]" and "S[j]" are swapped at the beginning of this cycle.
- Starting from the second iteration, the new value of index k is calculated by adding "S[i]" and "S[i]" ("read_data1" and "read_data2" ports of the S-box) and the "k_enable" signal of the "k_reg" register is enabled so that the new value of k is stored in the register on the next clock edge.
- Starting from the third iteration, the value of the next output (key stream byte) becomes available in the "S[k]_reg" register.

Cycle 2:

- The value of index j is updated by adding "j(t-1)" and "S[i]" ("read_data1" port of the S-box) *(in KSA stage, the value of "key [i mod key_length]", generated from the K-box, is also added*). The "j_enable" signal of the "j_reg" register is enabled so that the new value of j is stored in the register on the next clock edge.
- The value "S[i]" becomes available at the "read_data1" port of the S-box and the "S[i]_enable" signal of the "S[i]_reg" register is enabled so that the new value of "S[i]" is stored in the register on the next clock edge.
- The signal "sel_ j" is set to select j as the read index of the S-box so that "S[j]" is available at the "read_data2" on the next clock edge.
- Starting from the second iteration, the value "k(t-1)" becomes available in the "k_reg" register at this cycle.
- The signal "sel_ik" is set to select "k(t-1)" as the read index of the S-box so that "S[k(t-1)]" is available at the "read_data1" on the next clock edge starting from the second iteration.

Cycle 3:

- The value "j(t-1)" becomes available in the "j_reg" register at this cycle.
- The value "S[i](t-1)" becomes available in the "S[i]_reg" register at this cycle.

- The value "S[j]" becomes available at the "read_data2" port of the S-box.
- To perform the swapping of the S-box entries on the next clock edge, the following operations occur:
  1. The "write_enable" signals of both S-box ports are asserted.
  2. The signal "sel_ik" is set to select i as the write address of port1, so that "S[j]" is written at address i.
  3. The signal "sel_j" is set to select "j(t-1)" as the write address of port2, so that "S[i](t-1)" is written at address "j(t-1)".
- Starting from the second iteration, at this cycle "S[k(t-1)]" becomes available at the "read_data1" port of the S-box and the "S[k]_enable signal" of the "S[k]_reg" register is enabled so that the new value of "S[k(t-1)]" is stored in the register on the next clock edge.

Table 1 shows the operation of RC4 during each cycle.

Table 1. Proposed RC4's core operations.

Iteration 1		
Cycle 1	Cycle 2	Cycle 3
$i_1=i_0+1$ Read S[i_1]	$j_1=S[i_1]+j_0$** j_reg←j_1 Si_reg←S[i_1] Read S[j_1]	Write S[j_1] to address i_1 Write S[i_1](t-1) to address j_1(t-1)
Iteration 2		
Cycle 1	Cycle 2	Cycle 3
$i_2=i_1+1$ read S[i_2]	$j_2=S[i_2]+j_1$** j_reg←j_2 Si_reg←S[i_2] Read S[j_2]	Write S[j_2] to address i_2 Write S[i_2](t-1) to address j_2(t-1)
$k_1=S[i_1]+S[j_1]$ k_reg←k_1	read S[k_1(t-1)]	Sk_reg←S[k_1(t-1)]
Iteration 3		
Cycle 1	Cycle 2	Cycle 3
$i_3=i_2+1$ read S[i_3]	$j_3=S[i_3]+j_3$** j_reg←j_3 Si_reg←S[i_3] Read S[j_3]	Write S[j_3] to address i_3 Write S[i_3](t-1) to address j_3(t-1)
$k_2=S[i_2]+S[j_2]$ k_reg←k_2	Read S[k_2(t-1)]	Sk_reg ←S[k_2(t-1)] Sk_reg contains the first output value

* All operations that require memory access (reading or writing to memory) or register updating are performed on the next clock edge.

** In KSA stage, key [i] (generated from the K-box) is also added to calculate the new value of index j.

4. Experimental results

Our main goal in the design was to achieve the best throughput possible while keeping the area to a minimal in order to improve the overall efficiency. We used various Xilinx FPGA devices to implement our design and make use of its block RAM to support our goal of area utilization.

Our design was described in Verilog language and synthesized using Xilinx XST for Spartan6 FPGA. The use of dual-port RAM to implement the S-box reduced the consumed area. In order to compare our implementation with other RC4 implementations, we synthesized our design using Spartan3, Virtex2 and Virtex4 devices. Table 2 compares our implementation with previous implementations in terms of frequency, throughput, area, power and efficiency measured in throughput to area ratio. As we can see from the table our implementation gave reduced area and reasonable throughput with all three devices resulting in a better efficiency than the other implementations.

Power consumption of the proposed design was also compared with the work of Rourab et al.[7], which is the only paper that mentioned power consumption. Our design is more energy efficient and reduces the energy consumption to about 10% of that need in the design of Rourab et al.[7]

Table 2. Performance, area and power consumption comparison.

Work	Device	Frequency (MHz)	Throughput (MB/s)	Slices	LUTs	FFs	Block RAM (Bytes)	Efficiency (Throughput/LUT)	Power (mW)
Hamalain[5]	Virtex4 XC4000E-4013EPQ208-2	17.744	2.22	255	547	-	256 dual-port	0.0041	-
Kitsos[6]	Virtex2 XC2V250fg256	64	22	138	256	279	768 single-port	0.0859	-
Rourab[7]	Spartan3E XC3S500E	-	-	14448	26661	6404	0	-	1177
Proposed	Spartan6 XC6SLX16-2CSG324	155.2	51.73	68	226	194	256 dual-port	0.2289	61
Proposed	Virtex4 XC4VFX12-10FF668	186.26	62.09	410	275	197	256 dual-port	0.2258	210
Proposed	Virtex2 XC2V250-6CS144	167.35	55.78	376	298	205	256 dual-port	0.1872	78
Proposed	Spartan3 XC3S500E-4FG320	123.64	41.21	411	277	197	256 dual-port	0.1488	97

5. Conclusion

Even though it has some weaknesses, RC4 is still the most popular stream cipher used in protecting about 50% of internet TLS traffic. A new FPGA hardware implementation of the RC4 stream cipher was proposed in this paper. The use of dual-port bock RAM and pipelining were combined to reduce area resources and achieve better performance while improving the overall efficiency of the algorithm. The proposed design was implemented on different Xilinx FPGA devices and compared with existing designs reported in the literature. The synthesis results showed an improved efficiency (throughput-to-LUT ratio) of 0.2289 in the case of Spartan6 and almost double the efficiency was achieved using Virtex2. The proposed design is also efficient in terms of power consumption consuming only 97 mW.

References

1. M. Galanis, P. Kitsos, G. Kostopoulos, N. Sklavos, O. Koufopavlou and C. Goutis, "Comparison of the hardware architectures and FPGA implementations of stream ciphers," in The 11th IEEE Int. Conf. Electronics, Circuits and Systems, Tel Aviv, Israel, 2004.
2. S. S. Gupta, S. Maitra, G. Paul and S. Sarkar, "(Non-)Random Sequences from (Non-)Random Permutations—Analysis of RC4 Stream Cipher," Journal of Cryptology, vol. 27, no. 1, pp. 67-108, 2014.

3. N. AlFardan, D. Bernstein, K. Paterson, B. Poettering and J. Schuldt, "On the security of RC4 in TLS," in the 22nd USENIX Security Symposium, Washington, D.C., 2013.
4. B. Forouzan, Cryptography and Network Security, McGraw-Hill, 2008.
5. P. Hamalainen, M. Hännikäinen, T. Hamalainen and J. Saar, "Hardware implementation of the improved WEP and RC4 encryption algorithms for wireless terminals," in European Signal Processing Conference, Tampere, Finland, 2000.
6. P. Kitsos, G. Kostopoulos, N. Sklavos and O. Koufopavlou, "Hardware implementation of the RC4 stream cipher," in the 46th IEEE Midwest Sym. Circuits and Systems, Cairo, Egypt, 2003.
7. P. Rourab, S. Saha, J. Sadique Uz Zaman, S. Das, A. Chakrabarti and R. Ghosh, "A simple 1-byte 1-clock RC4 design and its efficient implementation in FPGA coprocessor for secured ethernet communication," arXiv: 1205.1737, 2012.
8. S. Paul and B. Preneel, "A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher," Fast Software Encryption, Lecture Notes in Computer Science, vol. 3017, p. 245–259, 2004.
9. M. Mckague, "Design and analysis of RC4-like stream ciphers," Master's thesis, University of Waterloo, 2005.
10. A. Al Noman, R. Sidek and A. R. Ramli, "Hardware implementation of RC4A stream cipher," Int. Journal of Cryptology Research, vol. 1, no. 2, pp. 225-233, 2009.
11. N. B. Hulle, R. D. Kharadkar and A. Y. Deshmukh, "Novel hardware implementation of modified RC4 stream cipher for wireless network security," Int. Journal of Computer Applications, vol. 47, no. 7, pp. 1 - 8, 2012.
12. J. D. Lee and C. P. Fan, "Efficient low-latency RC4 architecture designs for IEEE 802.11i WEP/TKIP," in Int. Symp. on Intelligent Signal Processing and Communication Systems, Xiamen, China, 2007.
13. S. Gupta, A. Chattopadhyay, K. Sinha, S. Maitra and B. Sinha, "High-performance hardware implementation for RC4 stream cipher," IEEE Transactions on Computers, vol. 62, no. 4, pp. 730-743, 2013.
14. Xilinx, "XST user guide for Virtex-6, Spartan-6, and 7 Series devices," April 2012. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_2/xst_v6s6.pdf. [Accessed 30 May 2015].