Hardware Implementation of four byte per clock RC4 algorithm

Rourab Paul, Amlan Chakrabarti, Senior Member, IEEE, and Ranjan Ghosh,

Abstract—In the field of cryptography till date the 2-byte in 1-clock is the best known RC4 hardware design [1], while 1-byte in 1-clock [2], and the 1-byte in 3 clocks [3][4] are the best known implementation. The design algorithm in [2] considers two consecutive bytes together and processes them in 2 clocks. The design [1] is a pipelining architecture of [2]. The design of 1-byte in 3-clocks is too modular and clock hungry. In this paper considering the RC4 algorithm, as it is, a simpler RC4 hardware design providing higher throughput is proposed in which 6 different architecture has been proposed. In design 1, 1-byte is processed in 1-clock, design 2 is a dynamic KSA-PRGA architecture of Design 1. Design 3 can process 2 byte in a single clock, where as Design 4 is Dynamic KSA-PRGA architecture of Design 3. Design 5 and Design 6 are parallelization architecture design 2 and design 4 which can compute 4 byte in a single clock. The maturity in terms of throughput, power consumption and resource usage, has been achieved from design 1 to design 6. The RC4 encryption and decryption designs are respectively embedded on two FPGA boards as co-processor hardware, the communication between the two boards performed using Ethernet.

Keywords—Computer Society, IEEEtran, journal, \LaTeX, paper, template.

1 INTRODUCTION

RC4 is a widely used stream cipher whose algorithm is very simple. It has withstood the test of time in spite of its simplicity. The RC4 was proposed by Ron Rivest in 1987 for RSA Data Security and was kept as trade secret till 1994 when it was leaked out [5]. Today RC4 is a part of many network protocols, e.g. SSL, TLS, WEP, WPA and many others. There were many cryptanalysis to look into its key weaknesses[5] [6] followed by many new stream ciphers [7] [8]. RC4 is still the popular stream cipher since it is executed fast and provides high security.

(Reference needed) There exist hardware implementations of some of the stream ciphers in the literature [9] [10] [11]. Since about 2003 when FPGA technology has been matured to provide cost effective solutions, many researchers started hardware implementation of RC4 as a natural fall out [3] [4]. The FPGA technology turns out to be attractive since it provides soft core processor having design specific functional capability of a main processor [12] along with reconfigurable logic blocks that can be synthesized to a desired custom coprocessor, embedded memories and IP cores. One can design RC4 algorithm totally as an executable code for the soft core processor (main processor) only or in custom coprocessor hardware operated by the main processor. Because of the system overhead, any single instruction if executed in the main processor takes at least 3 clocks, while the identical one when executed in a coprocessor takes 2 or 1-clock as the latter is customized to handle the specific task. Besides the clock advantage, the coprocessor based design makes the system throughput faster by another fold since it is executed in parallel with the main processor.

In this paper, RC4 algorithm is considered as it is and exploiting conventional VHDL features a design methodology is proposed processing of RC4 algorithm in two different hardware approaches. 1st architecture named as design 1 and design 2, which can execute 1 byte in single clock, the 2nd approach can execute 2 bytes in single clock which is named as design 3 and design 4 and lastly these two
architectural approaches (design 2 and design 4) has been accommodated in coprocessor environment named as design 5 and design 6 to increase the system throughput. The said design is implemented in a custom coprocessor functioning in parallel with a main processor (Xilinx Spartan3E XC3S500e-FG320 and Virtex5 LX110t FPGA architecture) followed by secured data communication between two FPGA boards through their respective Ethernet ports each of the two boards performs RC4 encryption and decryption engines separately. The performance of our design in terms of number of clocks proved to be better than the previous works [1], [2], [3], [4]. The dynamic KSA-PRGA architecture (design 3 and design 4) is introduced to reduce system power and resource utilization.

1.1 Existing Work

In the year 2004 P. Kitsos et al. [3] has tried a 3 byte per clock architecture of RC4 algorithm. At the first clock of PRGA the $i$ and $j$ is computed. In next clock $S[i]$ and $S[j]$ has been extracted from RAM and added. At the same step the addition has been stored in register $t$. At third clock swapping process has been done as well the value of $t$ register used to address the key $S[t]$. The swapping and $Z$ calculation is going on in the 3rd clock. When the $Z$ is being computed $t$ can not address the value at the swapped s-box, because swapped s-box will be updated after a clock pulse. This timing distribution of task can give error while $S[i] + S[j] = i$ or $j$.

In 2003 Matthews, Jr. [4] has proposed another 3 byte clock architecture where KSA and PRGA both have 3 clock for each iteration. On 2008 Matthews again proposed a new 1 byte per clock pipelined architecture in article Jr. [11]. Lastly S. Sen Gupta et al has proposed 2 architectures, such that 1 byte per clock [2] and 2 byte per clock [1] architecture. Article [2] is an excellent loop unrolled architecture where 2 byte is executing in 2 consecutive clocks. Article [1] is pipelined architecture of 1 byte clock architecture where 4 byte can be executed in 2 clocks.

1.2 Our Contribution

We have proposed 6 different architectures for RC4 algorithm to increase the throughput compromising with resource and power consumption and trying to search this most optimized architecture in terms of throughput, resource usage and power. The main contribution of this paper is described as a various design approaches.

**Design 1**: We have designed a RC4 architecture with 1 byte per clock throughput. This design has not been done using any loop unrolling method. It is a dual edge sensitive clock architecture, incorporate with pipelining concept.

**Design 2**: This design is identical with the design 1 which was consuming more power and more resource to have 1 byte per clock throughput. Keeping these points in mind we were searching some optimizing technique. We noticed that the KSA and PRGA processes are very much identical to each other. In design 1 when the PRGA starts the KSA does not have any significance of occupying the FPGA dice according to the algorithm concern. The existence of both of the PRGA and KSA process causes large silicon area leading with more power consumption but in recent days the FPGA vendors like XILINX, ALTERA does not have any provision to cut down the $V_{cc}$ voltage line of those MOSFETs which are unnecessary after its execution like KSA process. If the said provision can be accommodated on FPGA platform it can save large power consumption and resources. Recently XILINX has launched a partial reconfiguration tool which can modify hardware in runtime but without this tool we can use some smart trick to modify the KSA process into PRGA dynamically. This means the same hardware can be utilized by PRGA process, previously which was used by KSA process. In this design 2 we have introduced some logic with the algorithm which can transform the KSA process (after its execution) into PRGA process. The design 2 architecture is named as Dynamic KSA-PRGA architecture (DKP architecture). Design 2 has
reduced huge power consumption as well as resource usage without compromising of throughput. The results of power consumption and resource usage supporting our claim.

**Design 3:** To increase the throughput of this architecture we used a loop unrolling method which is mainly motivated from [2] but having a unique different hardware approach. This design 3 architecture can execute 2 byte in a single clock which means, comparing with the previous designs, the throughput is doubled without contributing large power and usage.

**Design 4:** Design 4 is a DKP form of design 3 to save the wattage and slice-LUTs.

**Design 5:** To have more throughput we used 4 parallel co-processor where each is coprocessor executing the design 2 architecture, parallel with a main processor. The main processor is responsible for data receiving and transmitting part. The data has been accused from PS2 port, RS232 port and Ethernet port. Only the 4 co-processors can be accommodate for parallel processing, because the data bus of microblaze processor is 32 bit, so coprocessors can communicate only 4 keys(1 key from each co-processor) with main processor at a time.

**Design 6:** In design 6 we used 2 parallel co-processor where each coprocessor is executing the design 4 architecture, parallel with a main processor. Only the 2 co-processor can be accommodate for parallel processing, because the data bus of microblaze processor is 32 bit, so co-processors can communicate only 4 keys(2 key from each co-processor) with main processor at a time. The other hardwares of design 6 is similar to design 5.

January 11, 2007

2 **RC4 Algorithm**

RC4 has a S-Box $S[N]$, $N = 0$ to 255 and a secret key, key[l] where l is typically between 5 and 16, used to scramble the S-Box [N]. It has two sequential processes, namely KSA (Key Scheduling Algorithm) and PRGA (Pseudo Random Generation Algorithm) which are stated below in figure [1]

![Fig. 1. RC4 Algorithm](image)

3 **Hardware Implementation of 1-byte 1-clock Design (Design 1 and Design 2)**

Figure [2] shows key operations performed by the main processor in conjunction with a coprocessor till the ciphering of the last text character. The hardware for realizing RC4 algorithm comprises of KSA and PRGA units, which are designed in the coprocessor as two independent units, and the XOR operation is designed to be done in the main processor. The central idea of the present embedded system implementation of one RC4 byte in 1-clock is the hardware design of a storage block shown in Figure [3] which is used in the KSA as well as in the PRGA units. The storage block contains a common S-Box connected to dual select MUX-DEMUX combination and executes the swap operation following line 9 of Algorithm 1 and line 6 of Algorithm 2, in order to update the S-Box. The swap operation in hardware is explained in the following sub-section.

3.1 **Storage Block Updating the S-Box**

The storage block consists of a register bank containing 256 numbers of 8-bit data representing the S-Box(register bank), 256:2 MUX, 2:256 DEMUX and 256 D flip-flops. Each of the MUX and DEMUX is so designed, that with 2-select inputs $i, j$ (8 bit width)
can address the two register data $S[i]$ and $S[j]$ at the same time (The VHDL compiler has merged 2 256:1 MUX to design this single 256:2 MUX, The same thing is happened for 2:256 DEMUX). The hardware design of the storage block updating the S-Box is shown in figure 3. For swapping, the same S-Box is accessed by the KSA unit with its MUX0-DEMUX0 combination and also by the PRGA unit with its MUX2-DEMUX2 combination and it is also accessed by another MUX3 in PRGA for the generation of the key stream $Z$. The storage block swaps $S[i]$ and $S[j]$ and thereby updates the S-Box. For swapping, $S[i]$ and $S[j]$ ports of MUX are connected to $S[j]$ and $S[i]$ ports of DEMUX respectively. This storage block has thus 3 input ports ($i$, $j$, and CLK), and 2 inlet ports ($S[i]$ of MUX, $S[j]$ of DEMUX and $S[i]$ of MUX, $S[j]$ of DEMUX). The storage block of PRGA unit provides 2 output ports from its 2 inlet ports which are fed to an adder circuit with MUX3. During the falling edge of a clock pulse, $S[i]$ and $S[j]$ values corresponding to $i$th and $j$th locations of the register bank are read and put on hold to the respective D flip-flops. During the rising edge of the next clock pulse, the $S[i]$ and $S[j]$ values are transferred to the MUX outputs and instantly passed to the $S[j]$ and $S[i]$ ports of the DEMUX respectively and in turn are written to the $j$th and $i$th locations of the register bank. The updated S-Box is ready during the next falling edge of the same clock pulse, if called for.

3.2 Design of the KSA Unit following Algorithm

Figure 4 shows a schematic diagram of design of the KSA unit. Initially the S-Box is filled with identity permutation of $i$ whose values change from 0 to 255 as stated in line 3 of initialization module. The 1-bytes of secret key are stored in the K[256] array as given in line 4. The KSA unit does access its storage block with $i$ being provided by a one round of MOD 256 up counter, providing fixed 256 clock pulses and $j$ being provided by a 3-input adder ($j$, $S[i]$, and $K[i]$) following the line 8 of the storage module, where $j$ is clock driven, $S[i]$ is MUX0 driven chosen from the S-Box and $K[i]$ is MUX1 driven chosen from the K-array. The S-Box is scrambled by the swapping operation stated in line 9 using MUX0-DEMUX0 combinations in the storage block. The KSA operation takes one initial clock and subsequent 256 clock cycles.

3.3 Design of the PRGA Unit following Algorithm

Figure 5 shows a schematic diagram of the design of the PRGA unit. The PRGA unit does
access the storage block with i being provided by a MOD 256 up counter (line 4) and j being given by a 2-input (j and S[i]) adder following the line 5 where j is clock driven and S[i] is MUX2 driven chosen from the S-Box. With updated j and current i, the swapping of S[i] and S[j] is executed following line 6 using MUX2-DEMUX2 combination of the storage block. Following the line 7, S[i] and S[j] give a value of t based on which the key stream Z is selected from the S-Box using MUX 3 (line 8).

### 3.4 Timing analysis of the PRGA Operation

Let $\phi_i$ denotes the $i^{th}$ clock cycle for $i \geq 0$. It is assumed that the PRGA unit starts when clock cycle is $\phi_0$. It is observed that a signal value gets updated during a falling edge of a clock cycle if it is changed during the rising edge of the previous clock cycle. The symbol $\leftrightarrow$ indicates swap operation. The clock-wise PRGA operations are shown in Figure 6.

**Timing Analysis of PRGA**

The MOD 256 up counter shown in Figure 5 is so designed that i starts from 1, goes up to 255 and then it repeats from 0 to 255 for each 256 subsequent clock cycles.

- Rising edge of $\phi_0$: Initialize $j_0=0$ and $i_0=0$.
- Falling edge of $\phi_0$: Start counter $i_1=1$.
- Rising edge of $\phi_1$: $j_1=(j_0 + S[i_1]) \mod 256$; temp=$i_1$.
- Falling edge of $\phi_1$: $i_2 \rightarrow 2$; S[temp] $\leftrightarrow S[j_1]$.
- Rising edge of $\phi_2$: $j_2 = (j_1 + S[i_2]) \mod 256$; temp=$i_2$, $Z_1 = (S[i_1] + S[j_1]) \mod 256 = 1$st Key.
- Falling edge of $\phi_2$: $i_3 \rightarrow 3$; S[temp] $\leftrightarrow S[j_2]$.
- Rising edge of $\phi_3$: $j_3 = (j_2 + S[i_3]) \mod 256$, temp=$i_3$, $Z_2 = (S[i_2] + S[j_2]) \mod 256 = 2$nd Key stream.
The series continues generating successive keys (Zs). If the text characters are n and \( n > 254 \), i = 0 after the first round and the clock repeats (n+2) times. After generating \( Z_n \) during the rising edge of \( \phi_{n+1} \), PRGA stops. For generating \( Z_n \), the PRGA requires (n+2) clocks and its throughput per byte is \( (1+2/n) \).

3.5 Dynamic KSA-PRGA Architecture (Design 2)

If we look into the KSA and PRGA process in Figure 1 we can find both of these processes are almost identical. Line 8,9 of KSA process and line 5, 6 of PRGA process are selfsame, except the extra addend secret key (K[i]) in KSA process. The key execution(Z) is also an additional process in PRGA. In section 3 design we have used two separate sequential blocks for both KSA and PRGA. As these processes are separated and executed sequentially, the algorithm core is contributing few extra static power, and dynamic power with some additional resource usage. To overcome these flaws we are proposing a single process dynamic KSA-PRGA architecture where KSA process itself is transformed to PRGA process after 257 clocks. Due to the reutilization of LUTs, slices and FF-pairs the proposed architecture can reduce significant resource usage which has led to low power design.

3.5.1 Hardware Overview

For the dynamic KSA-PRGA architecture few additional hardware has been added and some significant alteration of previous normal structural design has been done. The whole changes has been described below,

Instead of using two separated ‘i – counter’ for KSA and PRGA process we have used a single ‘i – counter’ in our new architecture. Previous ‘i – counter’ for KSA is a single round MOD-256 up counter and PRGA ‘i – counter’ was continuous MOD-256 up counter which has a counting range of 0 to 255 skipping the initial 0 count in first round. For both of these processes the dynamic architecture uses a simple continuous MOD-256 up counter which starts counting from count 0 and ends with 255 to finish the entire KSA process. After finishing the KSA process the counter is again reset to 0 count. At this 0 count the PRGA process is initialized (Initialized j by 0) and from the next successive count it starts the key execution process. A signal named as prga enable(prga_en) is employed to detect the KSA process status. When KSA has been completed prga enable has been latched high. During the KSA process prga enable stands 0 logic state to pass the \( K[i] \) to Adder1 through the 2:1 MUX but when prga enable goes high, 0 value will be forwarded to Adder1. So, at that time the Adder1 is started to execute \((j + S[i])\) instead of \((j + S[i] + K[i])\) as stated in PRGA algorithm. At the same time the prga enable signal enables the Adder2 circuit to execute the Z(keys). These two activation on main hardware block has altered the KSA process to PRGA process. Figure 7 shows the architectural block diagram of dynamic KSA-PRGA. Resource usage and power consumption of dynamic KSA-PRGA architecture has been compared with previous architecture in table ?? and ?? and Figure ?? is the graphical representation of table ?? and ??.
TABLE 1

| Steps | 1st iteration | 2nd iteration |
|-------|---------------|---------------|
| 1     | \( i_1 = i_0 + 1 \) | \( i_2 = i_1 + 1 = i_0 + 2 \) |
| 2     | \( j_1 = j_0 + S_0[i_1] \) | \( j_2 = j_1 + S_1[i_2] \) |
| 3     | \( SwapS_0[i_1] \leftrightarrow S_0[j_1] \) | \( SwapS_1[i_2] \leftrightarrow S_1[j_2] \) |
| 4     | \( Z_1 = S_1[S_0[i_1] + S_0[j_1]] \) | \( Z_2 = S_2[S_1[i_2] + S_1[j_2]] \) |

4 HARDWARE IMPLEMENTATION OF 2-BYTE 1-CLOCK DESIGN (DESIGN 3 AND DESIGN 4)

This approach is mainly motivated from reference [saurav sengupta's paper] which is based on loop unrolling method.

4.1 Storage Block Updating the S-Box

Figure 8 shows a schematic diagram of design of the storage box unit which is identical with figure 3. The only difference between the s-box of 1 byte 1 clock architecture and s-box of 2 byte 1 clock architecture is number of ports. This storage block also consists of a register bank containing 256 numbers of 8-bit data representing the S-Box (register bank), 256:1 MUX, 1:256 DEMUX and 256 D flip-flops. Here two sets of \( i \) and \( j \) can address 4 s-box element at a time. The value of \( S[i_1], S[i_2], S[j_1] \) and \( S[j_2] \) are updating by a 4 input MUX via Swap Controlling block.

4.2 KSA unit of 2 byte per clock architecture

4.2.1 \( j_1 \) and \( j_2 \) Generation of KSA unit

In this proposed architecture we need to compute \( j_1 \) and \( j_2 \) at the same clock instance. \( j_1 \) computation circuit is same as the previous KSA architecture shown in 4 but in the \( j_2 \) circuitry we need to adopt few tricks to compute the \( j_2 \) at that same clock while \( j_1 \) is executing. At the \( j_2 \) computation \( j_0 \) has been initiated by 0 value. According to the conventional RC4 algorithm \( j_1 \) is executed at the step 2 of table 4. In figure 3 Adder7 and Adder 8 is responsible to add \( j_0 + S_0[i_1] + K[i_1] \) to evaluate the value of \( j_1 \). For the \( j_2 \) computation we again need to see 1st column step2 of table 4 where the \( j_2 \) computation may be divided into the following two cases

\[
j_2 = j_0 + S_0[i_1] + S_0[i_2] + 2keys \quad \text{if} \quad i_2 \neq j_1.
\]

(1)

It has been noted that the only differentiation from \( S_0 \) to \( S_1 \) is the swap process, so it is very obvious to check if \( i_2 \) is equal to either of \( i_1 \) or \( j_1 \). As \( i_2 \) never be equal to \( i_1 \) (differ only by 1 modulo 256) we only need to check \( i_2 \) and \( j_1 \) which will decide whether \( S_0[i_2] \) or \( S_0[i_2] \) will be passed as a 3rd operand of addition process. Figure 9 shows the block diagram of \( j_1 \) and \( j_2 \) generator. Adder 1 is summing \( k[i_1] + K[i_2] \) with \( j_0 \) and passing this value to Adder3 and adder4. Adder3 and adder4 is adding \( k[i_1] + K[i_2] \) with \( S[i_2] \) and \( S[i_1] \) respectively. Adder5 and Adder 6 is responsible to execute \( j_0 + S_0[i_1] + S_0[i_2] + K[i_1] + K[i_2] \) and \( j_0 + S_0[i_1] + S_0[i_1] + K[i_1] + K[i_2] \).

4.2.2 Swap Controlling block

In line 9 of figure 4 a swapping process is updating s-box in each clock cycle. In 2 byte per clock architecture 2 loop iteration has been unrolled in a single loop iteration. Here two successive swap process should be accumulated in a single iteration. To replace this two
receive $S[i_1], S[i_2], S[j_1]$ and $S[j_2]$ from MUX and has 4 output to fed the swapped data to DEMUX unit. The Pictorial presentation is depicted on figure 10.

Fig. 9. j1 and j2 generator of KSA unit

![KSA unit schematic](image)

**TABLE 2**

| #   | Condition                  | Register to register swap movement |
|-----|---------------------------|------------------------------------|
| 1   | $i_2 \neq j_1, j_2 \neq i_1$ & $j_2 \neq j_1$ | $S[0][i_1] \rightarrow S[0][j_1], S[0][j_1] \rightarrow S[0][i_1], \ S[0][i_2] \rightarrow S[0][j_2], S[0][j_2] \rightarrow S[0][i_2]$ |
| 2   | $i_2 \neq j_1, j_2 \neq i_1$ & $j_2 = j_1$ | $S[0][i_1] \rightarrow S[0][i_2], S[0][i_2] \rightarrow S[0][j_1] = S[0][j_2], S[0][j_1] \rightarrow S[0][i_1]$ |
| 3   | $i_2 \neq j_1, j_2 = i_1$ & $j_2 = j_1$ | $S[0][i_1] \rightarrow S[0][i_1], S[0][i_1] = S[0][j_1] = S[0][j_2]$ |
| 4   | $i_2 \neq j_1, j_2 = i_1$ & $j_2 = j_1$ | $S[0][i_1] \rightarrow S[0][i_1], S[0][i_1] = S[0][i_1] = S[0][j_2]$ |
| 5   | $i_2 = j_1, j_2 \neq i_1$ & $j_2 = j_1$ | $S[0][i_1] \rightarrow S[0][i_2], S[0][i_2] \rightarrow S[0][j_1] = S[0][j_2], S[0][j_1] \rightarrow S[0][i_1]$ |
| 6   | $i_2 = j_1, j_2 \neq i_1$ & $j_2 = j_1$ | No data Movement |
| 7   | $i_2 = j_1, j_2 = i_1$ & $j_2 \neq j_1$ | Impossible |
| 8   | $i_2 = j_1, j_2 = i_1$ & $j_2 \neq j_1$ | Impossible |

Fig. 10. KSA unit of 2 byte per Clock Hardware

4.3 PRGA unit of 2 byte per clock architecture

PRGA unit of 2 byte per clock architecture is very much similar with the PRGA unit of 1 byte per clock architecture. Figure 11 shows a schematic diagram of the design of the PRGA unit. The Counter circuit generates $i_1$ and $i_2$ continuously and addresses $S[i_1]$ and $S[i_2]$ at the same instance. The $j_1$ and $j_2$ is updated by the respective $i_1$ and $i_2$ via $j_1$ and $j_2$ generator and latches the $S[j_1]$ and $S[j_2]$. After generating $S[i_1], S[i_2], S[j_1]$ and $S[j_2]$, all of these signals has been fed to Swap Controlling Block. This Swap Block has decided which signal will be transfer to which address of S-box through the DEMUX circuit.

4.3.1 $j_1$ and $j_2$ Generation of PRGA unit

This is a very simple circuit built by 4 adder blocks, 1 comparator and 1 MUX(2:1). As we seen in figure 12 $j_1$ computed very easily by Adder9 circuit but for the $j_2$ computation we again need to see 2nd column step 2 of table 1 where the $j_2$ computation may be divided into...
the following two cases

\[ j_2 = j_0 + S_0[i_1] + S_1[i_2] \]

if \( i_2 \neq j_1 \),

\[ j_2 = j_0 + S_0[i_1] + S_0[i_2] \]

if \( i_2 = j_1 \).

These two possible values from the two cases has been passed to MUX2 circuit through Adder10 and Adder11. The selecting input of the MUX2 is connected with Comparator5. The \( j_1 \) and \( i_2 \) have connected with the input of Comparator5. The Comparator5 takes the decision which value of \( j_2 \) will be passed to Adder12. Adder12 is computing the final \( j_2 \) by adding \( j_0 \) with it.

4.3.3 \( Z_1 \& Z_2 \) Generator block

In step 4 of Table [1] we got

\[ Z_1 = S_1[i_1] + S_1[j_1]. \] (3)

As \( S_0 \) and \( S_1 \) has been differentiated by a swap process, the value of \( Z_1 \) can be evaluated like,

\[ Z_1 = S_0[j_1] + S_0[i_1]. \] (4)

Thus the \( Z_1 \) can be computed by Adder20 of figure [13] by adding \( S_0[j_1] \) and \( S_0[i_1] \). The output of Adder20 is connected with a 256:2 MUX2 (Merge of a two 256:1 MUX) which is used to retrieve the appropriate data from s-box.

Now the \( Z_2 \) computation is involved as

\[ Z_2 = S_2[S_2[i_2] + S_2[j_2]] = S_2[S_1[j_2] + S_1[i_2]]. \] (5)

As we used loop unrolling method, we need to jump directly from \( S_0 \) state to \( S_2 \) state. Here while we will compute \( Z_2 \), a swap of the 1st loop already has been computed so this important issue should be kept in our mind during the \( Z_2 \) computation. Where as the 2nd swap process could not make effect on \( Z_2 \) according to equation 5. So Considering all possible cases of \( i_1, i_2, j_1, \) and \( j_2 \) we are trying the compute the value of \( S_1[i_2] \) and \( S_1[j_2] \) in terms of \( S_0 \) state.

The table [3] is showing all the details.

**TABLE 3**

| #  | Condition                              | Register to register Data movement |
|----|----------------------------------------|-----------------------------------|
| 1  | \( i_2 \neq j_1 \& j_2 \neq i_1 \)     | \( S_1[i_2] = S_0[i_2], S_1[j_2] = S_0[j_2] \) |
| 2  | \( i_2 \neq j_1 \& j_2 \neq i_1 \)     | \( S_1[i_2] = S_0[i_2], S_1[j_2] = S_0[j_1] \) |
| 3  | \( i_2 \neq j_1 \& j_2 = i_1 \)        | \( S_1[i_2] = S_0[i_2], S_1[j_2] = S_0[j_1] \) |
| 4  | \( i_2 \neq j_1 \& j_2 = i_1 \)        | \( S_1[i_2] = S_0[i_2], S_1[j_2] = S_0[j_1] \) |
| 5  | \( i_2 = j_1 \& j_2 \neq i_1 \)        | \( S_1[i_2] = S_0[i_1], S_1[j_2] = S_0[j_2] \) |
| 6  | \( i_2 = j_1 \& j_2 \neq i_1 \)        | \( S_1[i_2] = S_0[i_1], S_1[j_2] = S_0[j_1] \) |
| 7  | \( i_2 = j_1 \& j_2 = i_1 \)           | \( S_1[i_2] = S_0[i_1], S_1[j_2] = S_0[j_1] \) |
The circuit of the $Z_2$ computation can be realized using a 8:1 MUX (named as MUX3). 3 comparator circuit is connected with the 3 selecting input of MUX3. Comparator 6,7 and 8 comparing (i) $i_2$ and $j_1$, (ii) $j_2$ and $i_1$, (iii) $j_2$ and $j_1$. The output of MUX3 is connected with the selecting input of MUX4 to address the proper appropriate element of s-box as $Z_2$ key. The hardware description of $Z_2$ computation is shown in figure 13.

![Figure 13: Z1 and Z2 generator of PRGA unit](image)

### 4.4 Timing Analysis of PRGA of 2 byte per clock design

Again a MOD 256 up counter shown in figure [1] is designed which is very much identical with the up counter [3] but instead of single output it has two consecutive outputs named as $i_1$ and $i_2$. That $i_1$ starts from 1, goes up to 255 skipping all the even value between the said range and it repeats again until the plaintext sequence has been stopped. The $i_2$ starts from 2, goes up to 254 skipping all the odd value between the said range and then it repeats again from 0 to 254 skipping the same odd numbers. This means the said counter generates two consecutive counting values at the same clock instance. The details timing analysis has been shown below.

- Rising edge of $\phi_0$: Initialize $j_0 = 0$ and $i_0=0$.
- Falling edge of $\phi_0$: Start counter $i_1=1$ and $i_2=2$.

- Rising edge of $\phi_1$: $j_1=(j_0 + S_0[i_1]) \mod 256$; $j_2 = j_0 + S_0[i_1] + S_1[i_2]$, temp=$i_1$, temp_next=$i_2$.
- Falling edge of $\phi_1$: $i_3 =3$; and $i_4 = 4$, Swap Occurred;
- Rising edge of $\phi_2$: $j_3 = (j_2 + S_2[i_3]) \mod 256$; $j_4 = (j_2 + S_2[i_3] + S_3[i_4]) \mod 256$, temp=$i_3$, temp_next=$i_4$, $Z_1 = S_1(S_0[i_1] + S_0[j_1]) \mod 256$, $Z_2 = S_2(S_1[i_2] + S_1[j_2]) \mod 256$.
- Falling edge of $\phi_2$: $i_5 =5$; and $i_6 = 6$, Swap Occurred;
- Rising edge of $\phi_3$: $j_5 = (j_4 + S_3[i_5]) \mod 256$; $j_6 = (j_4 + S_2[i_5] + S_0[i_6]) \mod 256$, temp=$i_5$, temp_next=$i_6$, $Z_3 = (S_2[i_3] + S_2[j_3]) \mod 256$, $Z_4 = S_3(S_2[i_4] + S_2[j_4]) \mod 256$.

### 4.5 Dynamic KSA-PRGA Architecture (Design 4)

Taking the advantage of selfsame architecture of KSA and PRGA process, the same approach has been adopted in the 2 byte per clock architecture. The KSA process in transformed into PRGA process taking the acknowledgment of 'prga enable'/prga_en signal. Only the counter of said design is slightly different from 1 byte dynamic KSA-PRGA architecture of section 3.5. Two outputs of the said counter $i_1$ and $i_2$ count two consecutive counts starts from count '0' and '1' value respectively following an initialization clock of KSA process. At the 129th clock $i_1$ and $i_2$ reaches at count 254 and 255 respectively. At the 130th clocks $i_1$, $i_2$ and $j_1$, $j_2$ are initialized by 0 and prga enable goes high for the successive PRGA process. Since swap is not required for this initialization state, swap...
process has been stopped for this single clock duration. After this initialization again the \( i_1 \) and \( i_2 \) counts has been started with '1' and '2' counts. This consecutive counts is going on while we need key streams form PRGA process. The timing diagram is shown in 4.4.

![Timing Diagram](image)

Fig. 14. \( j_1 \) and \( j_2 \) Generator of Dynamic KSA-PRGA of 2 byte per Clock Hardware

4.5.1 Hardware overview

The hardware of \( j_1 \) and \( j_2 \) generator is shown in figure 14. Both MUX1 and MUX2 the prga enable signal decides whether \( K[i_1] + K[i_2] \) or '0' will be passed to Adder1. If prga enable goes low then \( K[i_1] + K[i_2] \) will be added with signal \( j_0 + S[i_1] + S[i_1] \) (for \( i_2 = j_1 \)) and with signal \( j_0 + S[i_1] + S[i_2] \) (for \( i_2 \neq j_1 \)). Otherwise '0' will be added with the said signal for both of the mentioned conditions using the Adder2, Adder3, Adder4, Adder5, Adder6 blocks. Among these two values of \( j_2 \), which one will passes as a final \( j_2 \) output is decided by Comparator circuit which has two input \( i_2 \) and \( j_1 \). The \( j_1 \) computation is smiler with section 3.5 architecture figure 7. The top level hierarchy of Dynamic KSA-PRGA architecture is shown in figure 15.

The functionality and pictorial representation of Storage Block unit, \( Z_1 \) and \( Z_2 \) generator has been already discussed and shown in section 4.1 and figure 8, section 4.3.3 figure 13 respectively. The swap control unit also has been discussed in section 4.3.2.

Fig. 15. Dynamic KSA-PRGA of 2 byte per Clock Hardware

5 1 BYTE PER CLOCK ARCHITECTURE AS 4 CO-PROCESSOR (DESIGN 5)

The back ground of this approach is "how many bytes can be generated in a single clock?". The more loop unrolling can make the circuit more complex in terms of latency and critical path, so we thought if we can accommodate four RC4 co-processor communicating with main processor to en-cipher 4 plain text character coming from the ethernet (or from other interfaces) at a single clock. 4 co-processor environment has been chose as we have 32 bit main processor and 32 bus availability. Here at an instant only 4 \( Z \) can be communicated with main processor through bus. See the figure 16 where 4 separate RC4 Co-processors work breaking the secret keys into same sizes (different size can also be accommodated according to the requirement of user). The four separated RC4 blocks generate 4 keys (Z) at a single pulse. So we can say the architecture can generate 4 bytes at single clock. After
computing of Zs, they are passed to the main processor to XOR those Zs (keys) with the plain text coming from ethernet. When the ciphering has been done, again those ciphers have been send to the unsecured ethernet environment.

6 2 BYTE PER CLOCK ARCHITECTURE AS 2 CO-PROCESSOR (DESIGN 6)

The same 4 byte per clock throughput can be achieved, if we used two 2 byte per clock architecture instead of using four 1 byte per clock architecture. Here in figure [17] two 2 byte per clock architecture can generate 4 keys at a single clock. The hardware usage and power consumption of proposed four different architecture is shown in table [6] and [5].

![Fig. 16. 4 RC4 co-processor with main processor](image)

![Fig. 17. 2 RC4 co-processor with main processor](image)

TABLE 4

| Critical Path Analysis |
|------------------------|
| Architectures         | 1 byte per Clock | 2 byte per Clock |
| Critical Path (ns)     | 4.127            | 5.15             |

8 RESULTS AND IMPLEMENTATION

The four kind of architecture have been proposed, such as, 1)1 byte per clock, 2)1 byte per clock Dynamic KSA-PRGA(DKP) clock, 3)2 byte per clock and finally 4)2 byte per clock Dynamic KSA-PRGA(DKP) clock. The hardware usage and power consumption of these proposed four different architectures are shown in table [6] and [5]. All of the architecture has been imported in parallel processing environment where main processor handling the data interface part and co-processor are taking care of RC4 algorithm part. We have seen the trade off between throughput, power consumption and throughput, resource usage. We have also observed how the power consumption and resource usage increases with the increasing of
the number of co-processor in tables 7, 8, 9 and 10.

TABLE 5
Power consumption of Proposed Architectures

| Milli watt | Static Power | Dynamic Power | Total Power |
|------------|--------------|---------------|-------------|
| Design 1   | 970.8        | 206.4         | 1177.2      |
| design 2   | 914.87       | 79.85         | 994.72      |
| Design 3   | 976.34       | 446.38        | 1422.71     |
| Design 4   | 953.94       | 109.91        | 1063.85     |
| Design 5   | 1004.74      | 159.92        | 1164.66     |
| Design 6   | 1016.96      | 194.61        | 1211.58     |

TABLE 6
Hardware usage of Proposed Architectures

| #          | Slice Registers | LUTs | Fully used LUTs FF pairs |
|------------|-----------------|------|--------------------------|
| Design 1   | 4173            | 14588| 4149                     |
| Design 2   | 2094            | 5383 | 2078                     |
| Design 3   | 4178            | 33115| 4168                     |
| Design 4   | 2123            | 15452| 2104                     |
| Design 5   | 17964           | 49545| 49886                    |
| Design 6   | 5682            | 36958| 37420                    |

All the designs has been implemented on Virtex(ML505, lx110t) and Spartan 3E(XC3S500e) FPGA board using ISE 14.4 and EDK 14,4 tool. The power and resource result is shown for Virtex board. The co processor has been coded by VHDL language and the main processor functionality has projected by system.c language. Two Xilinx FPGA Spartan3E (XC3S500e-FG320) boards, each with RC4 encryption and decryption engines separately and shown in Figure 18, are connected through Ethernet ports and to respective hyper terminals (DTE) through RS 232 ports.
9 COMPARISON WITH EXISTING DESIGNS

In table 11 we have compared the throughput of our design with existing work till date. In section 3 we have seen a 1 byte per clock architecture where KSA is executing within 257 clock after an initial lag of 1 clock. The PRGA is computing 1 byte after a lag of 2 initial clock. The KSA process in section 4 named as of 2 byte per clock architecture can iterate 256 loops within 128 clock after an initial clock delay and the PRGA of that said architecture can execute 2 bytes in single clock after a lag of 2 initial clocks. From the table 11 we can see Ref 3, 4 are 3 byte per clock design. Article 11, 2 and design 1, design 2 have same throughput of 1 byte per clock but more sophisticatedly article 2 and design1, design 2 have slightly better throughput as mention in the said table 11 where as article 11 and design 3, design 4 have same throughput of 2 byte per clock. More specifically our proposed design 3 and design 4 has slightly better throughput than 1.

10 CONCLUSION

The conclusion goes here.
### TABLE 11
Comparison of various performance metrics with existing designs

| Features                     | Number of Clock Cycles |
|------------------------------|------------------------|
|                              | Ref. [3], [4] | Ref. [11] | Ref. [12] | Ref. [1] | Design 1 & Design 2 | Design 3 & Design 4 |
| KSA                          | 256x3 = 768 | 3 + 256 = 259 | 1 + 256 = 257 | 1 + 256 = 257 | 1 + 128 = 129 |
| KSA per byte                 | 3           | 1 + 3/256 | 1 + 1/256 | 1 + 1/256 | 1/2 + 1/256 |
| PRGA for N byte              | 3n          | 3 + n     | 2 + n     | n/2 + 2   | 2 + n     |
| PRGA per byte                | 3           | 1 + 3/n   | 1 + 2/n   | 1/2 + 2/n | 1/2 + 2/n |
| RC4 for N byte               | 3n + 768    | 259 + (3+n) | 257 + (2+n) | 257 + (2+n/2) | 129 + (2+n/2) |
| Per byte output from RC4     | 3 + 768/n   | 1 + 262/n | 1 + 259/n | 1/2 + 259/n | 1/2 + 131/n |

[10] 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 *Electronics, Circuits and Systems, 2004. ICECS 2004. Proceedings of the 2004 11th IEEE International Conference on*, dec. 2004, pp. 571 – 574.

[11] D. Jr., in *Methods and apparatus for accelerating ARC4 processing*, US Patent No.7403615, Morgan Hill, CA, July, 2008.

[12] X. M. S. Processor. http://www.xilinx.com/tools/microblaze.htm.