One Byte per Clock: A Novel RC4 Hardware

Sourav Sen Gupta[1], Koushik Sinha[2],
Subhamoy Maitra[1], and Bhabani P. Sinha[1]

[1] Indian Statistical Institute, 203 B T Road, Kolkata 700 108, India
[2] Honeywell Technology Solutions Lab, Bangalore 560 076, India
sg.sourav@gmail.com, sinha_kou@yahoo.com, {subho,bhabani}@isical.ac.in

Abstract. RC4, the widely used stream cipher, is well known for its simplicity and ease of implementation in software. In case of a special purpose hardware designed for RC4, the best known implementation till date is 1 byte per 3 clock cycles. In this paper, we take a fresh look at the hardware implementation of RC4 and propose a novel architecture which generates 1 keystream byte per clock cycle. Our strategy considers generation of two consecutive keystream bytes by unwrapping the RC4 cycles. The same architecture is customized to perform the key scheduling algorithm at a rate of 1 round per clock.

Keywords: Fast Implementation, Hardware, RC4, Stream Cipher.

1 Introduction

RC4 is one of the widely used software stream ciphers that is mostly implemented in software. This cipher is used in network protocols such as SSL, TLS, WEP and WPA. As well the cipher finds applications in Microsoft Windows, Lotus Notes, Apple AOCE, Oracle Secure SQL etc. Though several other efficient and secure stream ciphers have been discovered after RC4, it is still the most popular stream cipher algorithm due to its simplicity, ease of implementation, speed and efficiency. The algorithm can be stated in a few lines, yet after two decades of analysis, its strengths and weaknesses are of great interest to the community. In spite of several cryptanalysis attempts on RC4 (see [1,2,4,9,10,11,13,14,15,16,18] and references therein), the cipher stands secure if used properly.

In this paper we present a novel hardware design of RC4 for fast generation of keystream. To motivate our contribution, we need to discuss the basic framework of RC4 first. A short note on RC4 follows.

1.1 RC4 Algorithm

The RC4 stream cipher has been designed by Ron Rivest for RSA Data Security in 1987. It uses an S-Box $S = (S[0], \ldots, S[N-1])$ of length N, each location storing one byte. Typically, $N = 256$, and S is initialized as the identity permutation, i.e., $S[y] = y$ for $0 \le y \le N-1$. A secret key k of size l bytes (typically, $5 \le l \le 16$) is used to scramble this permutation. An array

$K = (K[0], \ldots, K[N-1])$ is used to hold the secret key, where the key is repeated in K at key length boundaries. i.e., $K[y] = k[y \bmod l]$, for $0 \leq y \leq N-1$.

RC4 has two components, namely, the Key Scheduling Algorithm (KSA) and the Pseudo-Random Generation Algorithm (PRGA). The KSA uses the secret key K to generate a pseudo-random permutation S of $0, 1, \ldots, N-1$ and the PRGA uses this pseudo-random permutation to generate pseudo-random keystream bytes. The two pieces of the RC4 algorithm are as shown in Algorithm 1 and Algorithm 2. Any addition used related to the RC4 is in general addition modulo N unless specified otherwise. The keystream output byte Z is XOR-ed with the message byte to generate the ciphertext byte at the sender end, and is XOR-ed with the ciphertext byte to get back the message byte at the receiver end. The software implementation of RC4 is simple. Detailed comparison of the software performance of eStream portfolio and RC4 is given in [19].

```
Input: Secret Key $K$.
Output: S-Box $S$ generated by $K$.
for $i = 0, \ldots, N-1$ do
    $S[i] = i$;
end
Initialize counter: $j = 0$;
for $i = 0, \ldots, N-1$ do
    $j = j + S[i] + K[i]$;
    Swap $S[i] \leftrightarrow S[j]$;
end
```

Algorithm 1. KSA

```
Input: S-Box $S$, output of KSA.
Output: Random stream $Z$
        generated from $S$.
Initialize the counters: $i = j = 0$;
while $TRUE$ do
    $i = i + 1$;
    $j = j + S[i]$;
    Swap $S[i] \leftrightarrow S[j]$;
    Output $Z = S[S[i] + S[j]]$;
end
```

Algorithm 2. PRGA

1.2 Our Contribution

A 3-clock efficient implementation of RC4 on a custom pipelined hardware was proposed in Kitsos et al. [6] in 2003. Though there are already a few attempts to propose efficient hardware implementation [3,5,7,12] of RC4, the basic issue remained ignored that the design motivation should be initiated from the question that "In how many clocks a byte can be generated in an RC4 hardware?" To the best of our knowledge, this line of thought has never been studied and exercised in a disciplined manner in the existing literature, which in fact, is quite surprising.

In this paper we present a novel hardware for RC4 from this specific design motivation. Our model is a generic circuit based on simple ideas of combinational and sequential logic design. The main contribution of our work is to take a new look at RC4 by combining consecutive pairs of cycles in a pipelined fashion, and to read off the values of one state of the S-box from previous or later rounds of the cipher. To the best of our knowledge, the unwrapping of RC4 cycles to extract S-box information from previous or later stages is an idea which has never been exploited in designing an efficient hardware for RC4. The comprehensive design strategy and analysis of the circuit is presented in the following sections.

1.3 Organization of the Paper

Section 2: In this section, we propose the basic idea for our hardware implementation. This includes the modifications in the RC4 algorithm required for our hardware, and the circuits to perform each step of the modified algorithm.

Section 3: This section deals with a comprehensive timing analysis of the proposed architecture and presents a timing diagram for a few illustrative clock cycles in details.

Section 4: Here we present the complete circuit for our hardware, and prove our claim ('one byte per clock') as a formal theorem. In this section, we also discuss the minor modifications required to perform the KSA round using the same circuit (see Section 4.1), and compare our proposed architecture with the existing models for RC4 hardware, as found in the literature (see Section 4.2).

Section 5: This section concludes the paper by suggesting platforms for the practical implementation of the proposed hardware. We also present some ideas regarding parallelization of the circuit.

2 Hardware Implementation

We consider the generation of two consecutive values of Z together, for the two consecutive message bytes to be encrypted. Assume that the initial values of the variables i, j and S are i_0, j_0 and S_0, respectively. After the first execution of the PRGA loop, these values will be i_1, j_1 and S_1, respectively and the output byte is Z_1, say. Similarly, after the second execution of the PRGA loop, these will be i_2, j_2, S_2 and Z_2, respectively. Thus, for the first two loops of execution to complete, we have to perform the operations shown in Table 1.

Table 1. Two consecutive loops of RC4 Stream Generation

Steps	First Loop	Second Loop
1	$i_1 = i_0 + 1$	$i_2 = i_1 + 1 = i_0 + 2$
2	$j_1 = j_0 + S_0[i_1]$	$j_2 = j_1 + S_1[i_2] = j_0 + S_0[i_1] + S_1[i_2]$
3	Swap $S_0[i_1] \leftrightarrow S_0[j_1]$	Swap $S_1[i_2] \leftrightarrow S_1[j_2]$
4	$Z_1 = S_1[S_0[i_1] + S_0[j_1]]$	$Z_2 = S_2[S_1[i_2] + S_1[j_2]]$

For hardware realization, to store the S value, we use a bank of 8-bit registers, 256 in total. The output lines of any one of these 256 registers can be accessed through a 256 to 1 Multiplexer (MUX), with its control lines set to the required address i_1, j_1, i_2 or j_2. Thus, we need 4 such 256 to 1 MUX units to simultaneously read $S[i_1]$, $S[i_2]$, $S[j_1]$ and $S[j_2]$. Before that, let us study how to compute the increments of i and j at each level.

2.1 Step 1: Calculation of i_1 and i_2

We first note that incrementing i_0 by 1 and 2 can be done by the same clock pulse applied to two synchronous 8-bit counters both initialized with the value i_0; in one counter the clock pulse is applied to the input of all the flip-flops, and in the other, it is applied to all the flip-flops except the one at the LSB position, as shown in Fig. 1. Also, we need to load these counters with new values every other clock cycle (alternate with incrementing), and hence these must be counters with parallel load mechanism, so that the loading takes a single clock pulse.

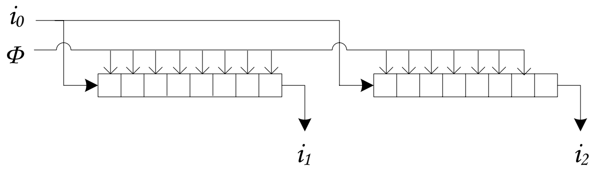

Fig. 1. [Circuit 1] Circuit to compute i_1 and i_2

Note that it is possible to simplify the implementation even further. We know that i_1 and i_2 will always be equal but for the LSB, which is always 0 for i_1 and 1 for i_2. Hence one needs only to implement a 7-bit counter for the 7 common MSBs of i_1, i_2, and append appropriate LSB.

2.2 Step 2: Calculation of j_1 and j_2

The values of j_1 and j_2 will be computed and stored in two 8-bit registers. To compute j_1, we need a 2-input parallel adder unit. It may be one using a carry lookahead adder, or one using *scan* operation as proposed by Sinha and Srimani [17], or one using *carry-lookahead-tree* as proposed by Lynch and Swarzlander, Jr. [8]. For computing j_2, there are two special cases:

$$j_2 = j_0 + S_0[i_1] + S_1[i_2] = \begin{cases} j_0 + S_0[i_1] + S_0[i_2] & \text{if } i_2 \neq j_1, \\ j_0 + S_0[i_1] + S_0[i_1] & \text{if } i_2 = j_1. \end{cases}$$

Note that the only change from S_0 to S_1 is the swap $S_0[i_1] \leftrightarrow S_0[j_1]$, and hence we need to check if i_2 is equal to either of i_1 or j_1. Now, i_2 can not be equal to i_1 as they differ only by 1 modulo 256. Therefore, $S_1[i_2] = S_1[j_1] = S_0[i_1]$ if $i_2 = j_1$, and $S_1[i_2] = S_0[i_2]$ otherwise. In both the cases, three binary numbers are to be added. Let us denote the k^{th} bit of $j_0, S_0[i_1]$ and $S_1[i_2]$ (either $S_0[i_2]$ or $S_0[i_1]$) by a_k, b_k and c_k, respectively, where $0 \leq k \leq 7$. We first construct two 9-bit vectors R and C, where the k^{th} bits $(0 \leq k \leq 8)$ of R and C are given by

$$R_k = \mathsf{XOR}(a_k, b_k, c_k) \quad \text{for } 0 \leq k \leq 7, \quad R_8 = 0, \quad \text{and}$$
$$C_0 = 0, \quad C_k = a_{k-1}b_{k-1} + b_{k-1}c_{k-1} + c_{k-1}a_{k-1} \quad \text{for } 1 \leq k \leq 8.$$

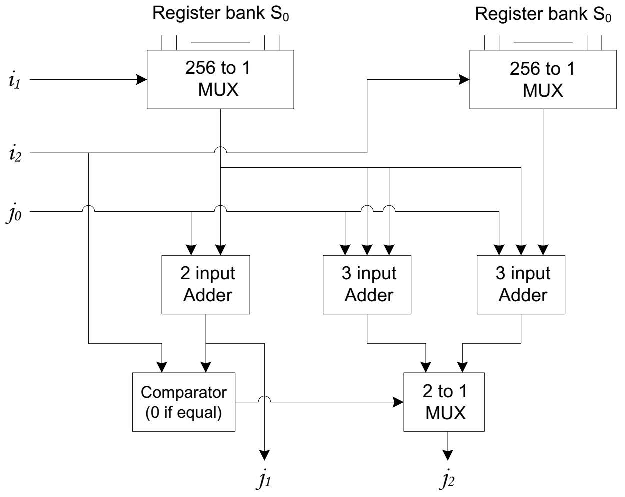

Fig. 2. [Circuit 2] Circuit to compute j_1 and j_2. (All connectors are 8-line bus.)

Note that in case of RC4, all additions are done modulo 256. Hence, we can discard the 9th bit ($k = 8$) of the vectors R, C while adding them together, and carry out normal 8-bit parallel addition considering $0 \leq k \leq 7$. Therefore, one may add R and C by a parallel full adder as used for j_1. The circuit to compute j_1 and j_2 is as shown in Fig. 2.

2.3 Step 3: Swapping the S Values

Step 3 in Algorithm 2 consists of one of the following 8 possible data transfer requirements among the registers of the S-register bank, depending on the different possible values of i_1, j_1, i_2 and j_2. We have to check if i_2 and j_2 can be equal to i_1 or j_1 (we only know that $i_2 \neq i_1$). All the cases in this direction can be listed as in Table 2. A more detailed explanation for each case is presented in Appendix A.

After the swap operation is completed successfully, one obtains S_2 from S_0. From the point of view of the receiving registers (in the S-register bank) in case of the above mentioned register-to-register transfers, we can summarize the cases as follows.

Table 2. Different cases for the Register-to-Register transfers in the swap operation

#	Condition	Register-to-Register Transfers
1	$i_2 \neq j_1$ & $j_2 \neq i_1$ & $j_2 \neq j_1$	$S_0[i_1] \to S_0[j_1]$, $S_0[j_1] \to S_0[i_1]$, $S_0[i_2] \to S_0[j_2]$, $S_0[j_2] \to S_0[i_2]$
2	$i_2 \neq j_1$ & $j_2 \neq i_1$ & $j_2 = j_1$	$S_0[i_1] \to S_0[i_2]$, $S_0[i_2] \to S_0[j_1] = S_0[j_2]$, $S_0[j_1] \to S_0[i_1]$
3	$i_2 \neq j_1$ & $j_2 = i_1$ & $j_2 \neq j_1$	$S_0[i_1] \to S_0[j_1]$, $S_0[i_2] \to S_0[i_1] = S_0[j_2]$, $S_0[j_1] \to S_0[i_2]$
4	$i_2 \neq j_1$ & $j_2 = i_1$ & $j_2 = j_1$	$S_0[i_1] \to S_0[i_2]$, $S_0[i_2] \to S_0[i_1] = S_0[j_1] = S_0[j_2]$
5	$i_2 = j_1$ & $j_2 \neq i_1$ & $j_2 \neq j_1$	$S_0[i_1] \to S_0[j_2]$, $S_0[j_2] \to S_0[j_1] = S_0[i_2]$, $S_0[j_1] \to S_0[i_1]$
6	$i_2 = j_1$ & $j_2 \neq i_1$ & $j_2 = j_1$	$S_0[i_1] \to S_0[j_1] = S_0[i_2] = S_0[j_2]$, $S_0[j_1] \to S_0[i_1]$
7	$i_2 = j_1$ & $j_2 = i_1$ & $j_2 \neq j_1$	Identity permutation, no data transfer.
8	$i_2 = j_1$ & $j_2 = i_1$ & $j_2 = j_1$	Impossible, as it implies $i_1 = i_2 = i_1 + 1$.

- $S_2[i_1]$ can receive data from any one of $S_0[i_1], S_0[j_1]$ or $S_0[i_2]$,
- $S_2[j_1]$ can receive data from any one of $S_0[i_1], S_0[j_1], S_0[i_2]$ or $S_0[j_2]$,
- $S_2[i_2]$ can receive data from any one of $S_0[i_1], S_0[j_1], S_0[i_2]$ or $S_0[j_2]$,
- $S_2[j_2]$ can receive data from any one of $S_0[i_1], S_0[i_2]$ or $S_0[j_2]$.

In view of the above discussions, the input data (1 byte) for each of the 256 registers in the S-register bank will be taken from the output of an 8 to 1 MUX unit, whose data inputs are taken from $S_0[i_1], S_0[j_1], S_0[i_2], S_0[j_2]$, and the control inputs are taken from the outputs of three comparators comparing (i) i_2 and j_1, (ii) j_2 and i_1, (iii) j_2 and j_1. The circuit to realize the swap operation is shown in Fig. 3.

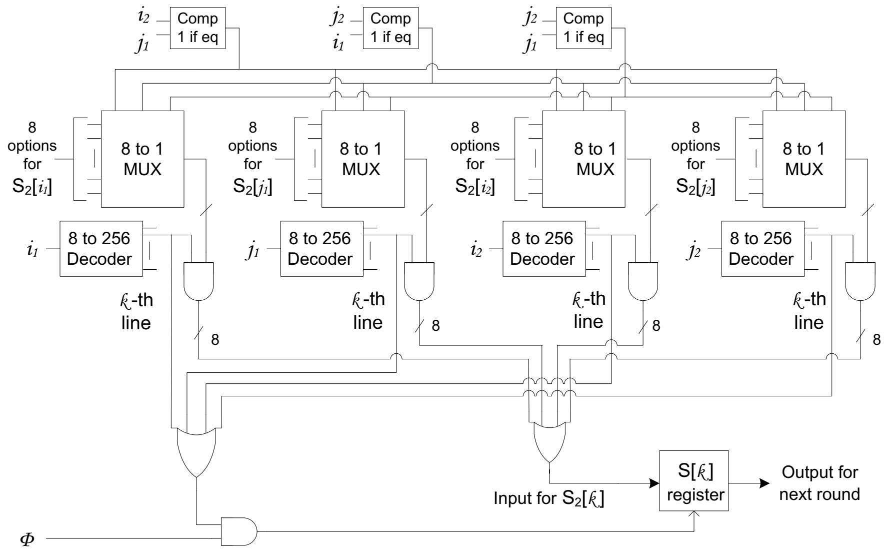

Fig. 3. [Circuit 3] Circuit to swap S values. (Data lines shown only for a fixed k.)

For simultaneous data transfers during the swap operation, we propose that the S-registers in the register bank be constituted by Master-Slave J-K flip-flops. The details of such flip-flops are presented in Appendix B. By using this type of registers, one can perform all the required register-to-register transfer operations in a single clock cycle.

2.4 Step 4: Calculation of Z_1 and Z_2

In step 4 of Algorithm 2, we have $S_1[i_1] + S_1[j_1] = S_0[j_1] + S_0[i_1]$, and

$$Z_1 = S_1\left[S_0[j_1] + S_0[i_1]\right] = \begin{cases} S_2[i_2] & \text{if } S_0[j_1] + S_0[i_1] = j_2, \\ S_2[j_2] & \text{if } S_0[j_1] + S_0[i_1] = i_2, \\ S_2[S_0[j_1] + S_0[i_1]] & \text{otherwise.} \end{cases}$$

Thus, the computation of Z_1 involves adding $S_0[i_1]$ and $S_0[j_1]$ first, which can be done using a 2-input parallel adder. The 256 to 1 MUX, which is used to extract appropriate data from S_2, will be controlled by another 4 to 1 MUX. This 4 to 1 MUX is in turn controlled by the outputs of two comparators comparing (i) $S_0[j_1] + S_0[i_1]$ and i_2, and (ii) $S_0[j_1] + S_0[i_1]$ and j_2. The circuit to compute Z_1 is as illustrated in Fig. 4.

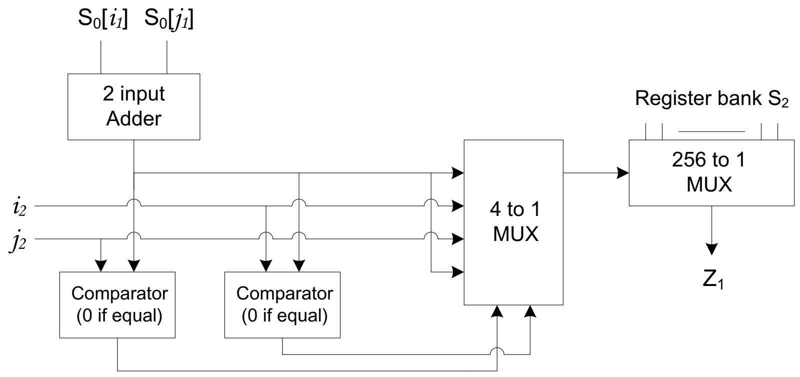

Fig. 4. [Circuit 4] Circuit to compute Z_1

Computation of Z_2, however, involves adding $S_1[i_2], S_1[j_2]$, as follows:

$$Z_2 = S_2\left[S_2[i_2] + S_2[j_2]\right] = S_2\left[S_1[j_2] + S_1[i_2]\right].$$

Note that we never actually store configuration S_1 of the register bank. This is because we move from state S_0 to S_2 directly to make the algorithm more efficient. Hence, we have to unwrap one cycle of RC4 and gather the values of $S_1[i_2]$ and $S_1[j_2]$ from the S_0 state. $S_1[i_2]$ and $S_1[j_2]$ receive the values from the appropriate registers of S_0 as given below, depending on the following conditions:

- $i_2 \neq j_1$ and $j_2 \neq i_1$ and $j_2 \neq j_1$: $S_1[i_2] = S_0[i_2]$ and $S_1[j_2] = S_0[j_2]$
- $i_2 \neq j_1$ and $j_2 \neq i_1$ and $j_2 = j_1$: $S_1[i_2] = S_0[i_2]$ and $S_1[j_2] = S_0[i_1]$
- $i_2 \neq j_1$ and $j_2 = i_1$ and $j_2 \neq j_1$: $S_1[i_2] = S_0[i_2]$ and $S_1[j_2] = S_0[j_1]$
- $i_2 \neq j_1$ and $j_2 = i_1$ and $j_2 = j_1$: $S_1[i_2] = S_0[i_2]$ and $S_1[j_2] = S_0[j_1]$
- $i_2 = j_1$ and $j_2 \neq i_1$ and $j_2 \neq j_1$: $S_1[i_2] = S_0[i_1]$ and $S_1[j_2] = S_0[j_2]$
- $i_2 = j_1$ and $j_2 \neq i_1$ and $j_2 = j_1$: $S_1[i_2] = S_0[i_1]$ and $S_1[j_2] = S_0[i_1]$
- $i_2 = j_1$ and $j_2 = i_1$ and $j_2 \neq j_1$: $S_1[i_2] = S_0[i_1]$ and $S_1[j_2] = S_0[j_1]$

These conditions can be realized using an 8 to 1 MUX unit controlled by the outputs of three comparators comparing (i) i_2 and j_1, (ii) j_2 and i_1, (iii) j_2 and j_1. Note that we can use the same control lines as in case of the swapping operation. The computation can be performed using the circuit as shown in Fig. 5.

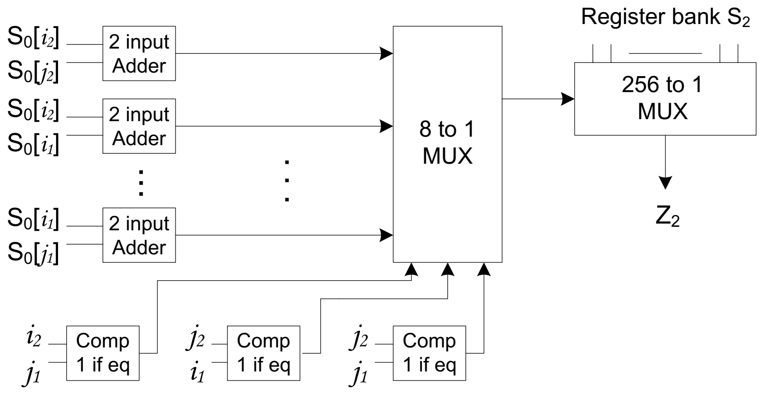

Fig. 5. [Circuit 5] Circuit to compute Z_2

3 Timing Analysis

Let us denote the operational clock by ϕ, and its i^{th} cycle by ϕ_i for all $i \geq 0$. We denote the initial clock cycle by ϕ_0, which triggers the PRGA circuit. Based on this notation, the timing analysis for the complete PRGA circuit (shown in Fig. 7) is as follows. We analyze the first two iterations of our model, that is, the generation of Z_1, Z_2 and Z_3, Z_4, and the analysis for the subsequent iterations fall along similar lines.

Initialization: The initial inputs to the circuit are $i_0 = 0$, $j_0 = 0$ and S_0 from the output of KSA. The two registers in Fig. 1 are preloaded with i_0, i.e., the values are set to 0. The S-registers in the register bank are all set to store the corresponding 8-bit values of S generated by the KSA.

Clock cycle ϕ_0: At the trailing edge of ϕ_0, the registers containing i_0 increment to produce i_1 and i_2. Also, the inputs to the four 256 to 1 MUX units are read from S_0.

Clock cycle ϕ_1: At the start of this cycle, we already have i_1, i_2 and S_0. Hence, one can obtain: (i) $S_0[i_1]$ and $S_0[i_2]$ from the first two 256 to 1 MUX, controlled by i_1, i_2, and (ii) j_1 and j_2 by combining j_0, $S_0[i_1]$ and $S_0[i_2]$, using the circuit in Fig. 2.

The values $i_1, i_2, j_1, j_2, S_0[i_1]$ and $S_0[i_2]$ are latched at the leading edge of ϕ_1. At the trailing edge of ϕ_1, the latched values of j_1, j_2 are accessed to control the last two 256 to 1 MUX.

Simultaneously, the first register of Fig. 1 is loaded (in parallel) with the value of i_2 at the trailing edge of ϕ_1, replacing the previous entry i_1. The second one already contains i_2.

Clock cycle ϕ_2: As we have latched j_1, j_2 released, we obtain $S_0[j_1]$ and $S_0[j_2]$ from the last two 256 to 1 MUX controlled by j_1, j_2. These two values from S_0 are latched at the leading edge of ϕ_2.

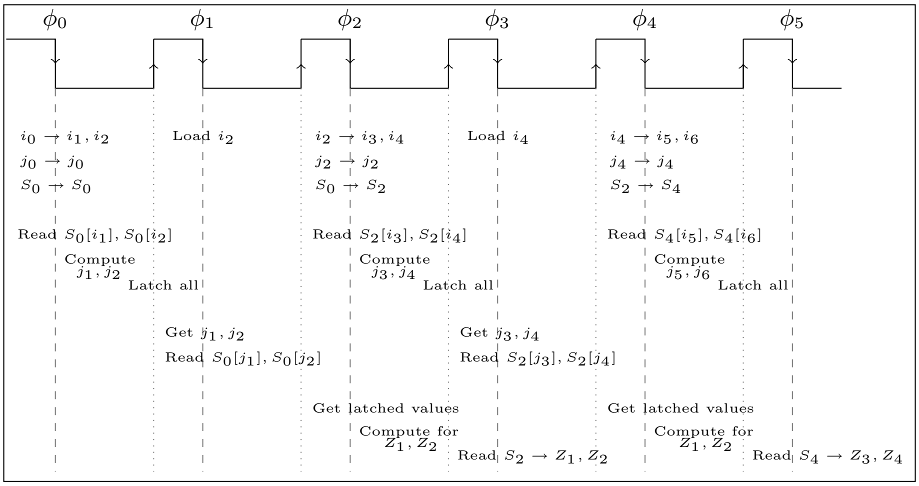

Fig. 6. Timing diagram for the complete PRGA circuit

At the trailing edge of ϕ_2, the *swap* operation takes place among the appropriate registers of S_0. One has all the values required for the swap module shown in Fig. 3, and hence can obtain the new configuration S_2 from S_0.

Simultaneously, the latched values for i_1, i_2, j_1, j_2, $S_0[i_1]$ and $S_0[i_2]$ are accessed, and the registers in Fig. 1 are incremented to i_3 and i_4 at the trailing edge of ϕ_2.

Clock cycle ϕ_3: At the start of this cycle, one has i_1, i_2, j_1, j_2, $S_0[i_1]$, $S_0[i_2]$, $S_0[j_1]$ and $S_0[j_2]$, released from the latches. Moreover, the register-bank configuration is S_2 at this stage. Hence, one can compute Z_1 using circuit in Fig. 4, and Z_2 using circuit in Fig. 5. The combinational logic of these circuits operate during the cycle and the bytes from S_2 are read at the trailing edge of ϕ_3.

Simultaneously, during cycle ϕ_3, the indices i_3, i_4 control the first two 256 to 1 MUX units to produce $S_0[i_3]$ and $S_0[i_4]$, similar to cycle ϕ_1. These values are used to produce j_3, j_4 and they are latched at the leading edge of ϕ_3. At the trailing edge of ϕ_3, the latches for j_3, j_4 are accessed.

Clock cycle ϕ_4: Similar to cycle ϕ_2, here we obtain $S_2[j_3], S_2[j_4]$, and latch these values at the leading edge of ϕ_4. Another swap operation is performed at the trailing edge of ϕ_4 to produce S_4 from S_2. Simultaneously, the first register in Fig. 1 is loaded with i_4 (replacing i_3), and incremented to i_5, i_6 at the trailing edge of ϕ_4.

Clock cycle ϕ_5: Similar to cycle ϕ_3, we compute the combinational logic for Z_3, Z_4 in this cycle (using circuits in Fig. 4 and Fig. 5), and the final values of Z_3 and Z_4 are read from S_4 at the trailing edge of ϕ_5.

The process continues similarly over the clock cycles and we obtain a timing diagram as shown in Fig. 6. The combinational logics operate between the clock pulses and the latches operate with the edges of the pulses (loaded at the leading edge and released at the trailing edge). All read, swap and increment operations are done at the trailing edges of the clock pulses.

One may observe that the first two bytes Z_1, Z_2 of the output are obtained at the end of the third clock cycle ϕ_3 and the next two bytes Z_3, Z_4 are obtained at the fifth clock cycle ϕ_5. A formal statement to summarize the observation is given in the next section.

4 The Complete Circuit

The complete circuit diagram for the PRGA stage of our RC4 hardware is shown in Fig. 7. Here, L_i denotes the latches operated by the trailing edge of ϕ_{2n+i}, i.e., the $(2n+i)^{th}$ cycle of the master clock ϕ where $n \geq 0$. For example, the latches labeled L_1 (four of them) are released at the trailing edge of $\phi_1, \phi_3, \phi_5, \ldots$ and the latches labeled L_2 (eight of them) are released at the trailing edge of $\phi_2, \phi_4, \phi_6, \ldots$ etc. We can now generalize our previous observation to state the following.

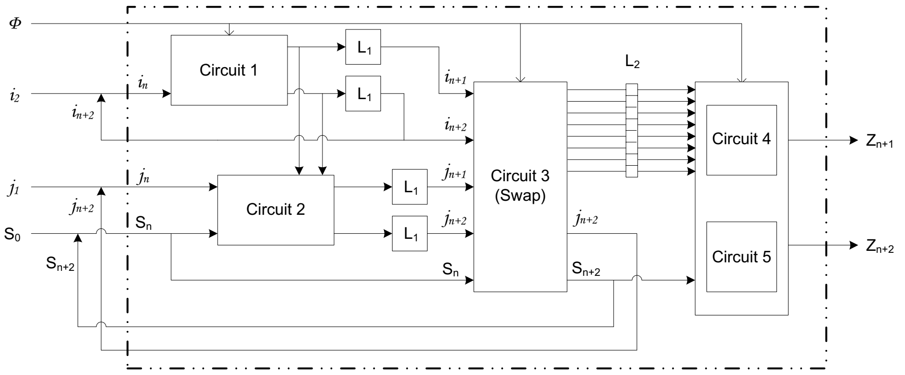

Fig. 7. Circuit for PRGA stage of RC4

Theorem 1. *The hardware proposed for the PRGA stage of RC4, as shown in Fig. 7, produces "one byte per clock" after an initial delay of two clock cycles.*

Proof. Let us call the stage of the PRGA circuit shown in Fig. 7 the n^{th} stage. This actually denotes the n^{th} iteration of our model, which produces the output bytes Z_{n+1} and Z_{n+2}.

The first block (Circuit 1) operates at the trailing edge of ϕ_n, and increments i_n to i_{n+1}, i_{n+2}. During cycle ϕ_{n+1}, the combinational part of Circuit 2 operates to produce j_{n+1}, j_{n+2}. The trailing edge of ϕ_{n+1} releases the latches of type L_1, and activates the swap circuit (Circuit 3). The combinational logic of the swap circuit functions during cycle ϕ_{n+2} and the actual swap operation takes place at the trailing edge of ϕ_{n+2} to produce S_{n+2} from S_n. Simultaneously, the latch of type L_2 is released to activate the Circuits 4 and 5. Once again, the combinational logic of these two circuits operate during ϕ_{n+3}, and we get the outputs Z_{n+1} and Z_{n+2} at the trailing edge of ϕ_{n+3}.

This complete block of architecture performs in a cascaded pipeline fashion, as the indices i_2, j_2 and the state S_{n+2} are fed back into the system at the end of ϕ_{n+2} (actually, i_{n+2} is fed back at the end of ϕ_{n+1} to allow for the increments at the trailing edge of ϕ_{n+2}). Hence, the operational gap between two iterations (e.g., n^{th} and $(n+2)^{th}$) of the system is two clock cycles (e.g., ϕ_n to ϕ_{n+2}), and we obtain two output bytes per iteration of the model.

Hence, the PRGA architecture proposed in Fig. 7 produces $2N$ bytes of output stream in N iterations, over $2N$ clock cycles. Note that the initial clock pulse ϕ_0 is an extra one, and the production of the output bytes lag the feedback cycle by one clock pulse in every iteration (e.g., ϕ_{n+3} in case of n^{th} iteration). Hence, our model practically produces $2N$ output bytes in $2N$ clock cycles, that is "one byte per clock", after an initial lag of two clock cycles. □

4.1 Issues for the Circuit of KSA

Note that the general KSA routine runs for 256 iterations to produce the initial permutation of the S-box. Moreover, the steps of the KSA phase in RC4 are quite similar to the steps of PRGA, apart from the following:

– Calculation of j involves the key K along with S-box S and index i.
– Calculation of Z_1, Z_2 not required (actually, not recommended).

We propose the use of our PRGA architecture (Fig. 7) for the KSA round as well, with some modifications to the design, as follows.

K-register bank: We have to introduce a new register bank for key K. It will contain l number of 8-bit registers, where $8 \leq l \leq 15$ in practice.

K-register MUX: To read data $K[i_1 \bmod l]$ and $K[i_2 \bmod l]$ from the K-registers, we need to have two 16 to 1 multiplexer unit. The first l input lines of this MUX will be fed data from registers $K[0]$ to $K[l-1]$, and the rest $16-l$ inputs can be left floating (recall that $8 \leq l \leq 15$). The control lines of these MUX units will be $i_1 \bmod l$ and $i_2 \bmod l$ respectively, and hence the floating inputs will never be selected.

Modular Counters: To obtain $i_1 \bmod l$ and $i_2 \bmod l$, we have to incorporate two modular counters (modulo l) for the indices. These will be synchronous counters and the one for i_2 will have no clock input for the LSB position, similar to circuit in Fig. 1.

Extra 2-input Adders: Two 2-input parallel adders to be appended to Fig. 2 for adding $K[i_1 \bmod l]$ and $K[i_2 \bmod l]$ to j_1 and j_2 respectively.

No Outputs: Circuits of Fig. 4 and Fig. 5 have to be removed from the overall structure, so that no output byte is generated during KSA. If any such byte is generated, the key K may be compromised, and hence, the output module have to be disconnected during the KSA stage. One can do so by resetting these modules permanently throughout the KSA operation.

Using this modified hardware configuration, we can implement two rounds of KSA in 2 clock cycles, that is "one round per clock", after an initial lag of 1 cycle. Total time required for KSA will be $256 + 1 = 257$ clock cycles in this case.

4.2 Comparison with Existing Architecture

Combining our KSA and PRGA architectures, we can obtain $2N$ output-stream bytes in $2N + 258$ clock cycles, counting the initial delay of 2 cycles for PRGA. The best hardware implementation of RC4 till date is described in [6], which provides an output of N bytes in $3N + 768$ clock cycles. A formal comparison of the timings is shown in Table 3. One can easily observe that for large N, the throughput of our RC4 architecture is 3 times compared to that of the hardware configuration proposed in [6].

Table 3. Timing comparison of our hardware with that of [6]

Operations	Clock cycles needed for hardware in [6]	Clock cycles needed for our model
Per round of KSA	3	1
Complete KSA routine	$256 \times 3 = 768$	$256 + 1 = 257$
N output bytes from PRGA	$3N$	$N + 2$
N output bytes from RC4	$3N + 768$	$257 + (N + 2) = N + 259$
Per byte output from RC4	$3 + \frac{768}{N}$	$1 + \frac{259}{N}$

Another aspect to compare is the cost of the hardware configuration proposed in the two cases. Table 4 in Appendix C summarizes the hardware components required for our model. The reader may note that the hardware components used in the architecture proposed in [6, Figure 3] can be compared with the ones we use. The hardware of [6] uses three 256-byte RAM blocks for the S-box, while we use only one such 256-byte block built by master-slave J-K flip-flops. The number of registers for our K-register bank is the same as the ones used for the K-box in [6]. Both architectures use the same number of 8-bit registers in addition to these banks. Our architecture requires in excess only a constant number of combinational and sequential logic components compared to that in [6]. Thus, not only in terms of throughput, but also in terms of memory requirement, our architecture is better than that proposed in [6].

5 Conclusion

In this paper we try to answer the question of RC4 hardware efficiency in terms of the throughput, that is, the number of keystream bytes generated per clock cycle of the system. We have proposed a novel architecture for a generic RC4 hardware in this direction, using basic combinational and sequential logic components. Our architecture performs the KSA stage of RC4 at a rate of "one round per clock" and produces "one byte per clock" in case of the PRGA routine. To the best of our knowledge, this is 3 times faster than the current best hardware configuration for RC4.

Simulation of our architecture using VHDL and its implementation on an FPGA module are in progress. To get the best performance in terms of fast real-time clock cycles, one may also want to fabricate the architecture onto an ASIC board, or a processing chip. We would also like to add that this hardware model can generate even higher throughput provided one has enough memory space to perform the swap and read operations at the same time, that is, if two copies of the S array can be maintained.

Acknowledgements. The authors are grateful to the anonymous reviewers for their valuable comments and suggestions that helped in improving the quality of the paper.

References

1. Fluhrer, S.R., McGrew, D.A.: Statistical Analysis of the Alleged RC4 Keystream Generator. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 19–30. Springer, Heidelberg (2001)
2. Fluhrer, S.R., Mantin, I., Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 1–24. Springer, Heidelberg (2001)
3. Galanis, M.D., Kitsos, P., Kostopoulos, G., Sklavos, N., Goutis, C.E.: Comparison of the Hardware Implementation of Stream Ciphers. Int. Arab. J. Inf. Tech. 2(4), 267–274 (2005)
4. Golic, J.: Linear statistical weakness of alleged RC4 keystream generator. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 226–238. Springer, Heidelberg (1997)
5. Hamalainen, P., Hannikainen, M., Hamalainen, T., Saarinen, J.: Hardware implementation of the improved WEP and RC4 encryption algorithms for wireless terminals. In: Proc. of Eur. Signal Processing Conf. pp. 2289–2292 (2000)
6. Kitsos, P., Kostopoulos, G., Sklavos, N., Koufopavlou, O.: Hardware Implementation of the RC4 stream Cipher. In: Proc. of 46th IEEE Midwest Symposium on Circuits & Systems 2003, Cairo, Egypt (2003), `http://dsmc.eap.gr/en/members/pkitsos/papers/Kitsos_c14.pdf`
7. Lee, J.-D., Fan, C.-P.: Efficient low-latency RC4 architecture designs for IEEE 802.11i WEP/TKIP. In: Proc. of Int. Symp. on Intelligent Signal Processing and Communication Systems ISPACS 2007, pp. 56–59 (2007)
8. Lynch, T., Swartzlander Jr., E.E.: A Spanning Tree Carry Lookahead Adder. IEEE Trans. on Computers 41(8), 931–939 (1992)
9. Mantin, I.: Predicting and Distinguishing Attacks on RC4 Keystream Generator. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 491–506. Springer, Heidelberg (2005)
10. Mantin, I.: A Practical Attack on the Fixed RC4 in the WEP Mode. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 395–411. Springer, Heidelberg (2005)
11. Mantin, I., Shamir, A.: A Practical Attack on Broadcast RC4. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 152–164. Springer, Heidelberg (2002)

12. Matthews Jr., D.P.: System and method for a fast hardware implementation of RC4. US Patent Number 6549622, Campbell, CA (April 2003), http://www.freepatentsonline.com/6549622.html
13. Maximov, A., Khovratovich, D.: New State Recovering Attack on RC4. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 297–316. Springer, Heidelberg (2008)
14. Mironov, I. (Not So) Random Shuffles of RC4. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 304–319. Springer, Heidelberg (2002)
15. Paul, G., Maitra, S.: On Biases of Permutation and Keystream Bytes of RC4 towards the Secret Key. Cryptography and Communications - Discrete Structures, Boolean Functions and Sequences 1(2), 225–268 (2009)
16. Roos, A.: A class of weak keys in the RC4 stream cipher. Two posts in sci.crypt, message-id 43u1eh1*j3@hermes.is.co.za*, 44*ebge*llf@hermes.is.co.za (1995), http://marcel.wanda.ch/Archive/WeakKeys
17. Sinha, B.P., Srimani, P.K.: Fast Parallel Algorithms for Binary Multiplication and Their Implementation on Systolic Architectures. IEEE Trans. on Computers 38(3), 424–431 (1989)
18. Wagner, D.: My RC4 weak keys. Post in sci.crypt, message-id 447o1I$cbj@cnn.Princeton.EDU (1995), http://www.cs.berkeley.edu/~daw/my-posts/my-rc4-weak-keys
19. Performance results, http://www.ecrypt.eu.org/stream/perf/#results

Appendix A: Detailed Explanation for the Swapping Cases

Case 1: $i_2 \neq j_1$ **and** $j_2 \neq i_1$ **and** $j_2 \neq j_1$

These data transfers are symbolically represented by the following permutation on data in S_0.

$$\begin{pmatrix} i_2 & j_2 \\ j_2 & i_2 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}$$

This involves 4 simultaneous register to register transfers:

$$S_0[i_1] \rightarrow S_0[j_1], \quad S_0[j_1] \rightarrow S_0[i_1], \quad S_0[i_2] \rightarrow S_0[j_2], \quad S_0[j_2] \rightarrow S_0[i_2]$$

Case 2: $i_2 \neq j_1$ **and** $j_2 \neq i_1$ **and** $j_2 = j_1$

In this case the data transfers are represented by the following permutation on data in S_0.

$$\begin{pmatrix} i_2 & j_1 \\ j_1 & i_2 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}$$

This involves 3 simultaneous register to register transfers:

$$S_0[i_1] \rightarrow S_0[i_2], \quad S_0[i_2] \rightarrow S_0[j_1] = S_0[j_2], \quad S_0[j_1] \rightarrow S_0[i_1]$$

Case 3: $i_2 \neq j_1$ **and** $j_2 = i_1$ **and** $j_2 \neq j_1$

In this case the data transfers are represented by the following permutation on data in S_0.

$$\begin{pmatrix} i_2 & i_1 \\ i_1 & i_2 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}$$

This again involves 3 simultaneous register to register transfers:

$$S_0[i_1] \rightarrow S_0[j_1], \quad S_0[i_2] \rightarrow S_0[i_1] = S_0[j_2], \quad S_0[j_1] \rightarrow S_0[i_2]$$

Case 4: $i_2 \neq j_1$ **and** $j_2 = i_1$ **and** $j_2 = j_1$

In this case the data transfers are represented by the following permutation on data in S_0.

$$\begin{pmatrix} i_2 & i_1 \\ i_1 & i_2 \end{pmatrix} \circ \begin{pmatrix} i_1 & i_1 \\ i_1 & i_1 \end{pmatrix}$$

This involves 2 simultaneous register to register transfers:

$$S_0[i_1] \rightarrow S_0[i_2], \quad S_0[i_2] \rightarrow S_0[i_1] = S_0[j_1] = S_0[j_2]$$

Case 5: $i_2 = j_1$ **and** $j_2 \neq i_1$ **and** $j_2 \neq j_1$

In this case the data transfers are represented by the following permutation on data in S_0.

$$\begin{pmatrix} j_1 & j_2 \\ j_2 & j_1 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}$$

This involves 3 simultaneous register to register transfers:

$$S_0[i_1] \rightarrow S_0[j_2], \quad S_0[j_2] \rightarrow S_0[j_1] = S_0[i_2], \quad S_0[j_1] \rightarrow S_0[i_1]$$

Case 6: $i_2 = j_1$ **and** $j_2 \neq i_1$ **and** $j_2 = j_1$

In this case the data transfers are represented by the following permutation on data in S_0.

$$\begin{pmatrix} j_1 & j_1 \\ j_1 & j_1 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}$$

This involves 2 simultaneous register to register transfers:

$$S_0[i_1] \rightarrow S_0[j_1] = S_0[i_2] = S_0[j_2], \quad S_0[j_1] \rightarrow S_0[i_1]$$

Case 7: $i_2 = j_1$ **and** $j_2 = i_1$ **and** $j_2 \neq j_1$

In this case the data transfers are represented by the following permutation on data in S_0.

$$\begin{pmatrix} j_1 & i_1 \\ i_1 & j_1 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}$$

This is identity permutation, and does not involve any data transfer.

Case 8: $i_2 = j_1$ **and** $j_2 = i_1$ **and** $j_2 = j_1$

This case cannot occur, as it implies $i_1 = i_2$, which is impossible because $i_2 = i_0 + 2 = i_1 + 1$.

Appendix B: Master-Slave J-K Flip-Flops

The master-slave configuration is basically two J-K flip-flops connected together in series, as shown in Fig. 8. The input signals J and K are connected to the Master flip-flop which *locks* the input while the clock input is *high*. As the clock input of the Slave flip-flop is the inverse of the Master clock input, the outputs from the Master flip-flop are *seen* by the Slave only when the clock input goes

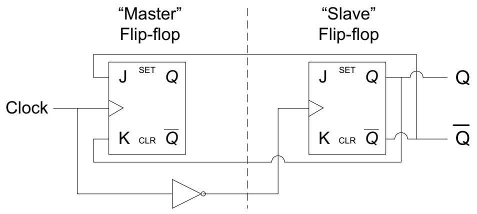

Fig. 8. Master-Slave J-K flip-flop

low. Therefore on the trailing edge of the clock pulse, the locked outputs of the Master flip-flop are fed through to the J-K inputs of the Slave flip-flop. The circuit accepts input at the Master J-K flip-flop when the clock signal is high, and passes it to the output on the trailing edge of the clock.

This enables us to make data interchange between two registers or cyclic data transfers among three registers, as needed by the swap operations, in just one clock. Recall that the registers in the S-register bank are made of these Master-Slave J-K flip-flops. We connect the registers participating in the swap operation using combinational logic circuits during the relevant clock cycle (ϕ_r, say), and wait for the clock pulse. At the leading edge of ϕ_r, the Master flip-flop of each register accepts the incoming new value, and it is passed on to the output through the Slave flip-flop only at the trailing edge of ϕ_r. Thus, the swap operation completes at the end of the cycle ϕ_r, and avoids all data collisions. This controlled flow of data helps perform the swap operations in a single clock-cycle.

Appendix C: Hardware Components

Table 4 lists all hardware components required for the RC4 architecture proposed in this paper. It counts all sequential and combinational components required for the circuit of PRGA (Fig. 7), as well as the extra components required for the proposed modifications to perform the KSA.

Table 4. Hardware components required to realize the proposed RC4 architecture

Component	Type	Number of pieces used	Module built using the components
Registers	8-bit (master-slave J-K)	256	S-register bank
	8-bit (normal)	l	K-register bank
	8-bit (parallel load)	2	Storage for j_1, j_2
Counters	8-bit (asynchronous)	2	Counters for i_1, i_2
	8-bit (modulo l)	2	Counters for i_1 mod l and i_2 mod l
Multiplexers	256 to 1	$4 + 2 = 6$	To read S-register values and Z_1, Z_2
	16 to 1	2	To read K-register values
	8 to 1	$4 + 1 = 5$	Fig. 3 and Fig. 5
	4 to 1	1	Fig. 4
	2 to 1	1	Fig. 2
Address Decoders	8 to 256	4	Fig. 3
Comparators	2 input	$1 + 3 + 2 = 4$	Fig. 2, Fig. 3 and Fig. 4
Parallel Adders	3 input	2	Fig. 2
	2 input	$1 + 1 + 8 + 2 = 12$	Fig. 2, Fig. 4, Fig. 5 and KSA extra adders
Latches	Edge-triggered	$4 + 8 = 12$	L_1 and L_2 latches in Fig. 7
Logic Gates	4 input OR	2	Fig. 3
	2 input AND	5	Fig. 3