A Modular Transform Method of Crypto Algorithm for Optimized FPGA Synthesis

Dongsoo Kim[1], Hae-Wook Choi[2]

[1] Attached Institute of Electronics and Telecommunication Research Institute
Daejeon, Korea
[2] Dept. of Electrical Engineering
Korea Advanced Institute of Science and Technology, 103-6 Munji-dong, Yuseong-gu
Daejeon 305-732, Korea
[1]{heavikim}@ensec.re.kr, [2]{hwchoinew}@kaist.ac.kr

Abstract This paper describes optimization methodology of FPGA implementation of crypto algorithm in embedded system. In USN (Ubiquitous Sensor Network) or Ubiquitous computing environment, security is taken accounted of as important element in implementation. But when security is applied to embedded system, it tends to have degrading effect on overall system performances of throughput and communication network bandwidth. One of the most influential factors is security throughput of embedded system. Key generation speed is critical to throughput and efficiency of security system. The methodology proposed in this paper is about how to enhance the security computation speed in embedded system. As a synthesis technique for increasing throughput, transform method from a sequential processing to a parallel processing is exploited in FPGA implementation. In this paper the non-algebraic shrinking key generator is explained to give an example of the transform method. The designs in VHDL were synthesized and implemented in Altera Cyclone 6000T devices.

Key words: modular transform, crypto algorithm, non-Boolean algorithm

1 Introduction

The rapid growth of computer and communication networks has enhanced the demand for security algorithm computations in embedded system that can handle high-speed data rate. Concerns about vulnerability of such a high rate system enforce complex-structured security algorithm to be embedded and executed at a high rate, so a lot of effort is needed to make complex-structured invulnerable security algorithm be hardware-friendly suited to high-speed applications. Two main implementation methods are exploited to optimize the trade-off between high-speed and invulnerability.

1. Transform from a serial structure (sequential computation) to a combinational structure (parallel computation).
2. Computation with pipelining stages.

Implementation method 2 is generally applied to any structured algorithm, but implementation method 1 becomes more sophisticated to apply if the algorithm can not be expressed in Boolean functions. One of most famous non-Boolean security algorithm is the shrinking key generator. Because it is non-Boolean and non-algebraic, up to date there exists no known attack method. No high-speed shrinking key generator hardware implementations have been reported in the recent literature.

2 Implementation

We coded the design in VHDL, simulated with Mentor Graphics ModelSim 5.7f and synthesized with Altera Quartus 3.1, targeting Altera Cyclone 6000T .

2.1 Sequential design

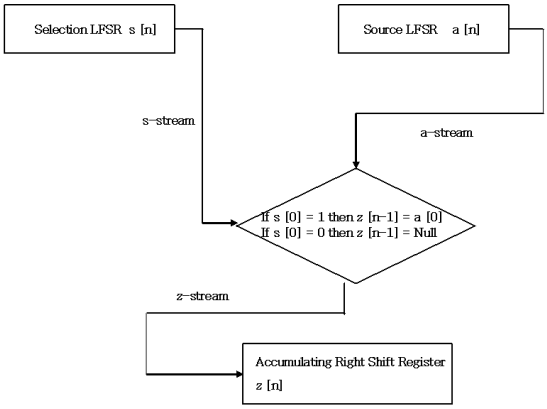

Fig. 1. Sequential computing structure

The sequential computing structure is composed of two Nth LFSR (one is the Nth selection LFSR and the other is the Nth source LFSR), the selection logic (diamond-shaped) and the Nth accumulating register. All three registers are circular right shifting registers as shown in Fig. 1.

S-stream is a sequence of consecutive s[0]s that are LSB(Least Significant Bit) of the selection LFSR for each clocking cycle and a-stream is a sequence of consecutive a[0]s that are LSB of the source LFSR for each clock cycle. The selection logic in Fig. 2 produces z-stream, which is a sequence of consecutive z[N-1]s that are MSB of the accumulating register, in a following way.

If s[0] = 1 then z[N-1] = a [0] (1)

If s[0] = 0 then z[N-1] = Null (2)

,where Null means Non-existence of bit.

It was reported and proved that the shrinking key generator in Fig. 1 has periodicity $(2^{N-1})^2$ and linear complexity from $N(2^{N-2})$ to $N(2^{N-1})$ in [1].

The shrinking key generator in Fig. 1 is easy to implement in a sequential design, because only (1) and (2) need to be implemented.

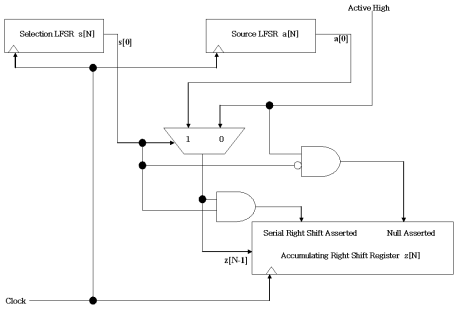

Fig. 2. Schematic diagram of sequential design

In Fig. 2 if s[0] is 1, a[0] is inserted to z[N-1] and after clock edge the accumulating register shifts right, making z[n-2] equal to z[n-1]. If s[0] is 0 , Null signal of the accumulating register is asserted and none inserted to z[N-1], making z[N-1] invariable, and after clock edge the accumulating register does not shift right , making z[n-2] invariable.

In Fig. 3 behavioral VHDL codes that synthesize the shrinking key generator are shown. In behavioral VHDL code A(i), B(i) and Q(i) represent the selection LFSR, the source LFSR and the accumulating register respectively.

```
if A(i) = '1' then          (3)
    temp (index):=B(i);     (4)
    index := index + 1 ;    (5)
 end if                     (6)
```

Above codes (3) ~ (6) represent the selection logic in Fig. 1.We recognize that as N increases more than 4, timing simulation result after synthesis gets more different and deviatory from functional simulation result before synthesis, so that behavioral VHDL design is inadequate to implementing the shrinking key generator.

```
process ( Clock , Reset )
  variable index  : unsigned (4 downto 0);
  variable temp   : std_logic_vector (15 downto 0);
    begin
     if ( Reset = '0') then
      Q <= ( others => '0');
     elsif rising_edge(Clock) then
      loop0 : for i in 0 to 15 loop
        if A(i) = '1' then
          temp (index):=B(i);
          index := index + 1 ;
        end if;
      end loop loop0;
       Q <= reg;
     end if;
  end process;
```

Fig. 3. Behavioral VHDL codes

2.2 Combinational design

To advance generation speed of the shrinking key generator, the sequential computing structure must be transformed to the combinational (parallel) computing structure. In contrast to the sequential structure, the parallel computing structure is based on N-bit length bus architecture. Operation of the selection logic can be expressed in following pseudo–codes.

```
For ( i = 0 ; i < N ; i++ )
 {
   If s[i] = 1 then z[index++1]= a[i];(7)
   If s[i] = 0 then NULL;             (8)

   If ( Index = N )                   (9)
   {
     Load N bit word and produce N bit
```

```
    key;                                    (10)
    Index = 0;                              (11)
  }
}
```

As we recognize in 2.1, due to behavioral nature of above pseudo-codes it can not be converted to synthesizable VHDL code. Block diagram of the N-bit bus-based parallel computing structure is shown in Fig. 4. Only a difference between Fig. 2 and Fig. 4 is the selection logic and the index indicator. The selection logic must perform (7) ~ (11) pseudo-codes and the index indicator counts the number of 1 of s[N] from the selection LFSR.

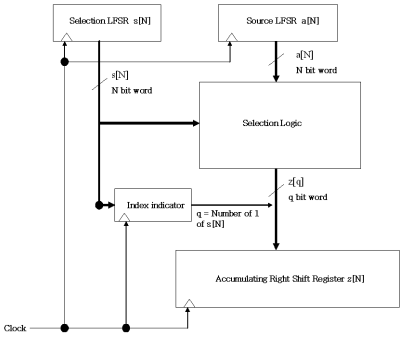

Fig. 4. Schematic diagram of N bit bus architecture

To make it simple, we propose divide-and-conquer scheme as design approach. In case of N =4, the selection logic functions as if it does in Fig. 5. As shown in Fig. 5, value of s[3] is 1, which causes z[3] equal to a[3] that is 1, and value of s[2] is 0, which causes a[2] neglected. Continually value of s[1] is 1, which causes z[2] equal to a[1] that is 0 and value of s[0] is 0, which causes a[0] neglected. The selection logic produces two bits of z[3] and z[2], each of them being 1 and 0 respectively. Be worthy of notice is that value of a[0] is pushed to the place of z[2], since previous two bits of s[2] and s[1] being 0 make corresponding a[2] and a[1] neglected. The selection logic is such that it produces z[n] value depending on previous history of values of s[n+1] ,s[n+2] ...etc. Such a selection logic in case of N= 4 can be designed with sophisticated combination of 2:1 MUXs, as shown in Fig. 6, which we define 4×4 Pushing logic.

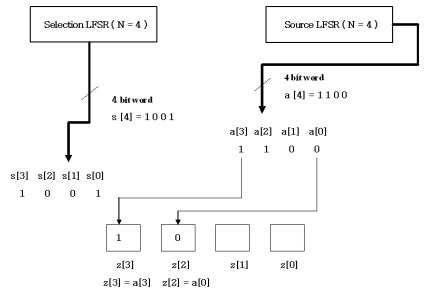

Fig. 5. Operation of Selection logic for N=4

We define the composite of four through logic elements as 4×4 through logic which is shown in Fig. 7. It must satisfy any combination of s[3], s[2],s[1],s[0].

In table2 the truth table for the 4×4 through logic lists all possible combinations of s[3], s[2], s[1] and s[0]. It was tested for the truth table for the 4×4 through logic and verified with simulation tool ModelSim 5.7f. We can expand the selection logic for N=8 to one for N=16, using the 4×4 pushing logic and the 4×4 through logic as a basic building block in a same method. The selection logic for N=16 is derived from four (1+1+1+1) 4×4 pushing logics and six (1+2+3) 4×4 through logics.

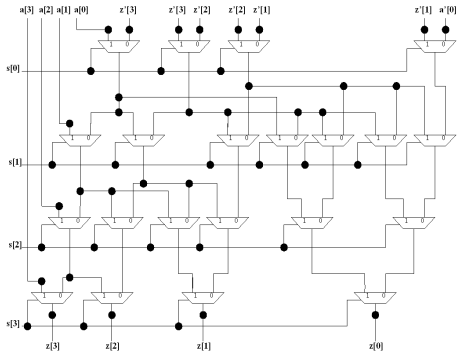

Fig. 6. 4×4 Pushing logic design with 2:1 MUX

Table 1. Truth table for the 4×4 pushing logic

s[3]	s[2]	s[1]	s[0]	z[3]	z[2]	z[1]	z[0]
0	0	0	0	z' [3]	z' [2]	z' [1]	z' [0]
0	0	0	1	a [0]	z' [3]	z' [2]	z' [1]
0	0	1	0	a [1]	z' [3]	z' [2]	z' [1]
0	0	1	1	a [1]	a [0]	z' [3]	z' [2]
0	1	0	0	a [2]	z' [3]	z' [2]	z' [1]
0	1	0	1	a [2]	a [0]	z' [3]	z' [2]
0	1	1	0	a [2]	a [1]	z' [3]	z' [2]
0	1	1	1	a [2]	a [1]	a [0]	z' [3]
1	0	0	0	a [3]	z' [3]	z' [2]	z' [1]
1	0	0	1	a [3]	a [0]	z' [3]	z' [2]
1	0	1	0	a [3]	a [1]	z' [3]	z' [2]
1	0	1	1	a [3]	a [1]	a [0]	z' [3]
1	1	0	0	a [3]	a [2]	z' [3]	z' [2]
1	1	0	1	a [3]	a [2]	a [0]	z' [3]
1	1	1	0	a [3]	a [2]	a [1]	z' [3]
1	1	1	1	a [3]	a [2]	a [1]	a [0]

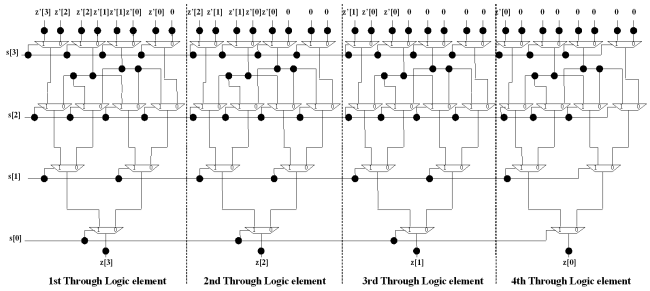

Fig. 7. 4×4 through logic composed of four through logic elements

Table 2. Truth table for the 4×4 through logic

s[3]	s[2]	s[1]	s[0]	z[3]	z[2]	z[1]	z[0]
0	0	0	0	0	0	0	0
0	0	0	1	z' [0]	0	0	0
0	0	1	0	z' [0]	0	0	0
0	0	1	1	z' [1]	z' [0]	0	0
0	1	0	0	z' [0]	0	0	0
0	1	0	1	z' [1]	z' [0]	0	0
0	1	1	0	z' [1]	z' [0]	0	0
0	1	1	1	z' [2]	z' [1]	z' [0]	0
1	0	0	0	z' [0]	0	0	0
1	0	0	1	z' [1]	z' [0]	0	0
1	0	1	0	z' [1]	z' [0]	0	0
1	0	1	1	z' [2]	z' [1]	z' [0]	0
1	1	0	0	z' [1]	z' [0]	0	0
1	1	0	1	z' [2]	z' [1]	z' [0]	0
1	1	1	0	z' [2]	z' [1]	z' [0]	0
1	1	1	1	z' [3]	z' [2]	z' [1]	z' [0]

3 Statistics

Both sequential design and combinational design have the following I/O connections.

input_source a[15:0]	16 bit input from source LFSR
input_selection b[15:0]	16 bit input from selection LFSR
output_accmulation z[15:0]	16 bit output from accumulation register
clock	Clock signal

We synthesize structural VHDL code of the schematics of Fig. 2 for implementing the sequential shrinking key generator and structural VHDL code of the schematics of Fig. 4 and Fig. 8 with Altera quartus 3.1 VHDL compiler, targeting Cyclone 6000T. Input_source a[15:0] and input_selectin b[15:0] are applied simultaneously every clock cycle. Clock frequency is 48 MHz. Implementation results for two designs are shown in Table 3. Notice that logic cell area of the combinational design is about 50 times more occupied than that of the sequential design in FPGA implementation. The speed of the combinational design is approximately 12 times faster than that of the sequential design. The transform from the sequential to the parallel structure improves overall performance of speed by 10 times faster than that of the previous conventional sequential implementation.

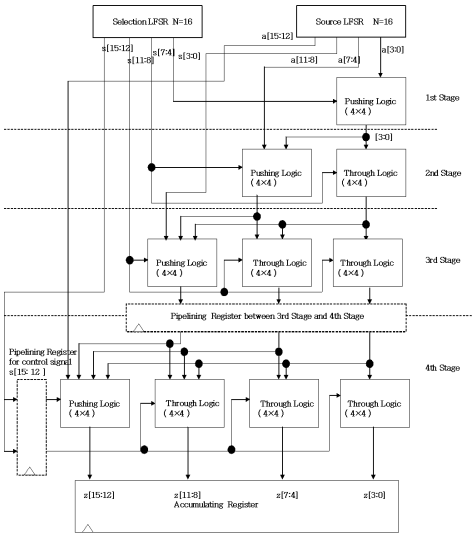

Fig. 8. Selection logic with two pipelining stages

Table 3. Comparison of implementation results of two design methods

	Combinational design	Sequential design
N (Degree of LFSR)	16	16
VHDL code size(lines)	900	50
Design size (Logic Cell)	5,700	120
Clock (Mhz)	48	48
Date rate in Cyclone 6000T (Mbps)	192~256	16~24

4 Conclusions and Future Work

We transformed the shrinking key generator for N=16 to a composite of four 4×4 pushing logics and six 4×4 through logics and targeted it to an FPGA. The resulting design of the parallel computation structure using two pipelining stages runs at 192 ~ 256 Mbps. Part of the reason for executing this design is to determine the performance improvement gained from transform method of optimizations to the design. We intend to use this information to drive design automation software development. In this design we are able to improve performance by more than a factor of twelve by applying a transform from the sequential computing algorithm to parallel computing and by inserting the pipelining stage as shown in Fig. 8. An observation of the delays in Fig. 8, making pipelining registers inserted, shows that most of the delay in combinational implementations is due to interconnect routing.

As we recognize that as N (degree of LFSR) increases more than 16, fan-outs of the through logic element to next stage increase by number of 10. In case of FPGA implementation for N>16, fan-out problem requires pipelining registers inserted every adjunctive stage, degrading the overall speed by reciprocal of number of pipelining registers. Moreover in case of ASIC implementation for N >16, in addition to speed degradation fan-out problem makes place & route so complicated that the design can not be synthesized even if timing simulation result shows appropriateness for ASIC implementation. The next step in this investigation is to apply manual scheme or plot of floor planning and placement to reduce interconnect delay in ASIC implementation. We are also interested in

implementing additional key generator algorithms, with the intent that a system would load the algorithm of choice into the FPGA as needed.

References

[1] Coppersmith, D., Krawczyk, H., Mansour, Y.: The Shrinking Generator. Advances in Cryptology – CRYPTO 93, LNCS, vol. 773, pp. 22--39 Springer, Heidelberg (1994)

[2] Ekdahl, P., Meier, W., Johansson, T.: Predicting the Shrinking Generator with Fixed Connections. EUROCRYPT 2003, LNCS vol. 2656, pp. 330--344 Springer, Heidelberg (2003)

[3] Kang, Z., Park, S., Park, C., Zi, S., Chun, J., Han, J.: Modern Cryptography. 2nd ed., Electronics and Telecommunication Research Institute, Daejeon (2003)

[4] Altera: Cyclone Handbook. vol. 1, Altera (2003)

[5] Smith, D. J.: HDL Chip Design. Doone Publications (1998)