RC6 Implementation including key scheduling using FPGA

(ECE 646 Project, December 2006)
Fouad Ramia, Hunar Qadir, GMU

Abstract—With today’s great demand for secure communications systems, there is a growing demand for real-time implementation of cryptographic algorithms. In this paper we present a hardware implementation of the RC6 algorithm using VHDL Hardware Description Language. We also investigate the efficiency of RC6 from the hardware implementation perspective with Field Programmable Gate Arrays (FPGAs) as the target technology. FPGAs are highly attractive options for hardware implementations of encryption algorithms as they provide cryptographic algorithm agility, physical security, and potentially much higher performance than software solutions. Our analysis and synthesis studies of the ciphers will suggest that it would be desirable for FPGA implementations to have a simple cipher design that makes use of simple operations that not only possess good cryptographic properties, but also make the overall cipher design efficient from the hardware implementation perspective.

1 INTRODUCTION

RC6 is a symmetric key block cipher derived from RC5. It was designed by Ron Rivest, Matt Robshaw, Ray Sidney, and Yiqun Lisa Yin to meet the requirements of the Advanced Encryption Standard (AES) competition by the National Institute of Standards and Technology (NIST). The algorithm was one of the five finalists, and was also submitted to the NESSIE and CRYPTREC projects. Though the algorithm was not eventually selected, RC6 remains a good choice for security applications. It is proprietary of RSA Security.

RC6 is more exactly specified as RC6-w/r/b, where the parameters w, r, and b respectively express the word size (in bits), the number of rounds, and the size of the encryption key (in bytes). Since the AES submission is targeted at w = 32 and r = 20, we implemented this version of RC6 algorithm, using a 32 bits word size, 20 rounds and 16 bytes (128 bits) encryption key lengths.

A key schedule generates 2r + 4 words (w bits each) from the b-bytes key provided by the user. These values (called round keys) are stored in an array S [0, 2r+3] and are used in both encryption and decryption.

RC6 works on a block size of 128 bits and it is very similar to RC5 in structure, using data-dependent rotations, modular addition and XOR operations; in fact, RC6 could be viewed as interweaving two parallel RC5 encryption processes. However, RC6 does use an extra multiplication operation not present in RC5 in order to make the rotation dependent on every bit in a word, and not just the least significant few bits. The computation of \( f(X) = (X(2X + 1)) \mod 2^w \) is the most critical arithmetic operation of this block cipher.

The goal of this paper is to implement the RC6 Cipher with FPGA as the target technology and then study the efficiency and performance of our design.

2 MOTIVATION - WHY RC6?

2.1 Security

To attack RC6 the best approach available to the cryptanalyst is that of exhaustive search for the b-byte encryption key. The more advanced attacks of differential and linear cryptanalysis, while being feasible on small-round versions of the cipher, do not extend well to attacking the full 20-round RC6 cipher. The RC6 key schedule is secure through mixing, one way function and no key separation. Therefore, RC6 provides a solid, well tuned margin for security.

2.2 Simplicity

RC6 facilitates and encourages analysis by allowing rapid understanding of security and making direct analysis straightforward. It also enables easy implementation by allowing compilers to produce high quality code for software implementations, and by preventing complicated optimizations and providing good performance with minimal effort for hardware implementations.

2.3 Performance

RC6 is known to have good performance on 8, 16 and 32-bit platforms. In this paper we will evaluate its performance in FPGAs.
3 STRUCTURE OF THE RC6 CIPHER ALGORITHM

3.1 BASIC OPERATIONS

RC6-w/r/b operates on units of four w-bit words using the following six basic operations. The base-two logarithm of w will be denoted by \( \lg w \).

- \( a + b \) integer addition modulo \( 2^w \)
- \( a - b \) integer subtraction modulo \( 2^w \)
- \( a \oplus b \) bitwise exclusive-or of w-bit words
- \( a \times b \) integer multiplication modulo \( 2^w \)
- \( a \ll b \) rotate the w-bit word a to the left by the amount given by the least significant \( \lg w \) bits of b
- \( a \gg b \) rotate the w-bit word a to the right by the amount given by the least significant \( \lg w \) bits of b

3.2 KEY SCHEDULE

The user supplies a key of b bytes. From this key, \( 2r + 4 \) words (w bits each) are derived and stored in the array \( S[0, 2r + 3] \). This array is used in both encryption and decryption.

3.3 ENCRYPTION

Input:
- Plain text stored in four w-bit input registers \( A, B, C, D \)
- Number \( r \) of rounds
- w-bit round keys \( S[0, 2r + 3] \)

Output:
- Cipher text stored in \( A, B, C, D \)

Procedure:

\[
B = B + S[0] \\
D = D + S[1] \\
\text{for } i = 1 \text{ to } r \text{ do} \\
\{ \\
\quad t = (B \times (2B + 1)) \ll \lg w \\
\quad u = (D \times (2D + 1)) \ll \lg w \\
\quad A = ((A \oplus t) \ll u) + S[2i] \\
\quad C = ((C \oplus u) \ll t) + S[2i+1] \\
\quad (A, B, C, D) = (D, A, B, C) \\
\}\]

\[
A = A + S[2r + 2] \\
C = C + S[2r + 3] \\
\]

3.4 DECRYPTION

Input:
- Cipher text stored in four w-bit input registers \( A, B, C, D \)
- Number \( r \) of rounds
- w-bit round keys \( S[0; \ldots; 2r + 3] \)

Output:
- Plaintext stored in \( A, B, C, D \)

Procedure:

\[
C = C - S[2r + 3] \\
A = A - S[2r + 2] \\
\text{for } i = r \text{ downto } 1 \text{ do} \\
\{ \\
\quad (A, B, C, D) = (D, A, B, C) \\
\quad u = (D \times (2D + 1)) \ll \lg w \\
\quad t = (B \times (2B + 1)) \ll \lg w \\
\quad C = ((C - S[2i+1]) \gg t) \oplus u \\
\quad A = ((A - S[2i]) \gg u) \oplus t \\
\}\]

\[
D = D - S[1] \\
B = B - S[0] \\
\]
3.4 Key Schedule – Revisited

The user supplies a key of b bytes. Sufficient zero bytes are appended to give a key length equal to a non-zero integral number of words; these key bytes are then loaded in little-endian fashion into an array of w-bit (w = 32 bits in our case) words L[0], ..., L[c - 1]. Thus the first byte of key is stored as the low-order byte of L[0], etc., and L[c - 1] is padded with high-order zero bytes if necessary. The number of w-bit (32 bit) words that will be generated for the additive round keys is 2r + 4 and these are stored in the array S[0; ...; 2r + 3]. The constants $P_{32} = B7E15163$ and $Q_{32} = 9E3779B9$ (hexadecimal) are the same “magic constants” as used in the RC5 key schedule.

Procedure:

\[
S[0] = P_{32}
\]

for i = 1 to 2r + 3 do

\[
A = B = i = j = 0
\]

\[
v = 3 \times \max\{c, 2r + 4\}
\]

for s = 1 to v do

\[
A = S[i] = (S[i] + A + B) \ll 3
\]

\[
B = L[j] = (L[j] + A + B) \ll (A + B)
\]

\[
i = (i + 1) \mod (2r + 4)
\]

\[
j = (j + 1) \mod c
\]

4 Hardware Implementation Environment

4.1 FPGA vs. ASICs

Field Programmable Gate Arrays (FPGAs) consist of arrays of configurable logic blocks that implement logical functions of gates that are easily reconfigurable. In contrast, Application Specific Integrated Circuits (ASICs) provide only the functionality needed for a specific task. An ASIC chip will support a particular application for which it is designed, but not a modified version of the same application introduced after the ASIC design is completed. On the other hand, the configuration of an FPGA can be easily reprogrammed to accommodate a design modification. Other key factors that favor the use of FPGAs for hardware implementation of ciphers include faster turnaround design time, scalable security, and variable architecture parameters. For those reasons we have chosen FPGAs as the target technology.

4.2 Selection of a Target FPGA

Scalability and cost must be considered. We believe that the chosen FPGA should be the best chip available, capable of providing the largest amount of hardware resources as well as being highly flexible so as to yield optimal performance. Unfortunately, the cost associated with current high-end FPGAs is relatively high (several hundred US dollars per device). However, it is important to note that the FPGA market has historically evolved at an extremely rapid pace, with larger and faster devices being released to industry at a constant rate. This evolution has resulted in FPGA cost-curves that decrease sharply over relatively short periods of time. Based on the mentioned considerations, the Xilinx Virtex XCV1000BG560-4 FPGA was chosen as the target device. The XCV1000 has 128K bits of embedded RAM divided among thirty-two RAM blocks that are separate from the main body of the FPGA. The 560-pin ball grid array package provides 512 usable I/O pins. The XCV1000 is comprised of a 64 x 96 array of look-up-table based Configurable Logic Blocks (CLBs), each of which acts as a 4-bit element comprised of two 2-bit slices for a total of 12288 CLB slices. This type of configuration results in a highly flexible architecture that will accommodate the round functions’ use of wide operand functions.

4.3 Development Environment

For this project we use VHDL Hardware Descriptive Language to simulate the hardware implementation of the RC6 algorithm on FPGAs. We used the CAD tools available at GMU.

The design process can be divided into the following stages:

- Use ALDEC Active HDL to generate the VHDL descriptions of the RC6 cipher using both behavioral and structural architectures, as well as the functional VHDL simulation waveforms.
- Use Synplify Pro 8.5 for Gate-level synthesis and logic optimization to produce the schematic in hardware.
- Use Xilinx ISE 7.1 to implement the synthesized design and determine the number of slices used and the
efficiency of the algorithm when implemented in hardware.

5 Design Analysis

5.1 Multiplication

When implementing the RC6 algorithm, it was first determined that the RC6 modulo $2^{32}$ multiplication was the dominant element of the round function in terms of required logic resources. Each RC6 round requires two copies of the modulo $2^{32}$ multiplier. However, it was found that the RC6 round function does not require a general modulo $2^{32}$ multiplier. The RC6 multipliers implement the function $A (2A + 1)$ which may be implemented as $2A^2 + A$. Therefore, the multiplication operation was replaced with an array squarer with summed partial products, requiring fewer hardware resources and resulting in a faster implementation.

5.2 Variable Shifting

Variable shifting operations have the potential to require considerable hardware resources, the 5-bit variable shifting required by the RC6 round function required few hardware resources. Instead of implementing a 32-to-1 multiplexer for each of the thirty-two rotation output bits (controlled by the five shifting bits), a multi-level multiplexing approach was used. The variable rotation is broken into multiple stages, each of which is controlled by one of the five shifting bits. For each rotation output bit of a given stage, a 2-to-1 multiplexer controlled by the stage’s shifting bit is used. This implementation requires a total of 160 2-to-1 multiplexers as opposed to the thirty-two 32-to-1 multiplexers required for a one-stage implementation. However, using 2-to-1 multiplexers to form the five-stage barrel-shifter results in an overall implementation that is smaller and faster when compared to the one-stage barrel-shifter implementation.

5.3 Other Operations

The remaining components of the RC6 round functions, consisting of fixed shifting, bit-wise XOR, and modulo $2^{32}$ addition, were found to be simple in structure, and requiring few hardware resources.

6 Design Architecture

6.1 RC6 Key Schedule Module

The majority of the research papers done so far about the RC6 algorithm and its implementation in hardware, and more specifically in FPGAs, assume that key scheduling is done outside of the FPGA. All of the subkeys are downloaded to the key storage unit of the FPGA and are then used in both encryption and decryption. Our project is different in the sense that we are performing key scheduling and generating all of the subkeys inside the FPGA. Once the key schedule algorithm has executed and all of the subkeys have been generated, encryption and decryption will be started. If the user wishes to input a new key, the key schedule algorithm will run again and a new set of subkeys will be generated to be later used in en-
6.2 RC6 Main Module

Input:
- **Key Input**: Key to be used by ecntr/decr
- **Key Avail**: Indicates that the key is available to be read
- **Data Input**: Message/Cipher text is entered into the cipher
- **Data Avail**: Indicates data is available to be read for enc/dec
- **Clock**: Master Clock
- **Reset**: Master Reset
- **Enc/Dec**: Enc/Dec 0/1 encryption/ decryption selection
- **Full**: Indicates output full and cannot output data

Output:
- **Key Read**: Indicates the key has been read
- **Data Read**: Entered into the cipher
- **Data Out**: Cipher text/ Plaintext is output through this port
- **Data Write**: Data becomes available on output bus
- **Ready**: Indicates that the key has been generated and the unit is ready for enc/dec.

6.3 RC6 Core Module

The RC6 core module is where the function \( f(X) = (X(2X + 1)) \mod 2^n \) is implemented. As we can see the data is first broken down to four words, each 32 bits wide represented by A, B, C and D. Key scheduler prepares two 32 bit words from the S array, one value from the even addresses and one from the odd addresses. In the case of encryption A and C are added with these two values from S. Also u and t are shifted by 5 before they are Xored with output from the barrel shifter.
6.4 RC6 Block diagram

To begin with, the data is first read in 128 bits and broken down to 4 x 32 bits words (A, B, C and D). Initially, and in case of encryption, the first two words in the S array are added to B and D. For Decryption, the two words are subtracted from C and A. These four blocks make the initial 128 bits that will be fed to a register before going into the core module through a multiplexer that controls the input for the core for every round. After completing all the rounds the output is sent to a register where it will be saved. Finally, this 128 bit is broken down to four blocks again, so the final addition and subtraction will be done before sending it as the cipher data.

Fig. 7 RC6 Block diagram
6.5 Control Unit

The control unit for RC6 is a very complete one due to the fact that it also generates different signals for generating the array of S keys. Two counters are controlled using these signals. A 5 bit counter is used in key generating and preparing the array S of keys in the rounds. Each control signal is controlled by a state and in some cases by other values as well. This unit also generates output signals for feeding the data in and sending the data out. The ASM chart shows when the signals are set and reset.

Fig. 8 – ASM chart of the Control Unit

The next block diagram in Fig. 10 shows the signals needed to control key generation and encryption/decryption units.
7 RESULTS

7.1 Testing
The first step in the design process is to check for the functional correctness of the design using simulations. Then, the FPGA synthesis tool is used to interpret logic components from the VHDL code. The synthesis tool produces a netlist which does not have accurate timing information since placement and routing of the logic components on the FPGA is not yet determined. The post-synthesis netlist can however be used to check the correct inference of logic components from the VHDL code. The final step in the process is to map the design to the target FPGA. The FPGA implementation tool, which is vendor-specific, generates a netlist which has accurate timing as well as logic information. The final netlist is used for simulation to check if the design will actually work when configured physically on the FPGA. This type of simulation is known as timing simulation.

Our design did pass timing simulation under the control of testbench. Testbenches were written in VHDL in order to verify the functionality of the design. The testbenches were designed to read test vectors from a file and compare the produced output with expected outputs stored in another file. Extensive testing was done for checking both the encryption as well as decryption functionality.

7.2 Waveforms

Key setup: The following waveform shows when signal ready turns to 1 meaning that the key schedule is done and the data is being read. Here we read the input “00000000000000000000000000000000”

Encryption: The following waveform shows when the input “00000000000000000000000000000000” is encrypted and the cipher text “A078D51A2816082A155AA150D47AA152” is outputted.

Decryption: The following waveform shows when the cipher “A078D51A2816082A155AA150D47AA152” is decrypted and the original text “00000000000000000000000000000000” is outputted.

7.3 Obtained Results
After running the VHDL code, checking for functionality, synthesizing and then implementing the code, we got the following results which are summarized in the following tables.
### TABLE 1
**TIMING RESULTS**

<table>
<thead>
<tr>
<th></th>
<th>Number of Cycles</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key Setup</td>
<td>188</td>
<td>4305</td>
</tr>
<tr>
<td>Encryption</td>
<td>21</td>
<td>480</td>
</tr>
<tr>
<td>Decryption</td>
<td>21</td>
<td>480</td>
</tr>
<tr>
<td>Reading Input</td>
<td>1</td>
<td>22.88</td>
</tr>
<tr>
<td>Writing Output</td>
<td>1</td>
<td>22.88</td>
</tr>
</tbody>
</table>

### TABLE 2
**RESOURCE UTILIZATION**

<table>
<thead>
<tr>
<th></th>
<th>VirtexE XCV1000</th>
<th>Number of Slices</th>
<th>Percentage of total available</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flip Flops</td>
<td></td>
<td>481</td>
<td>1%</td>
</tr>
<tr>
<td>4 input LUTs</td>
<td></td>
<td>3,248</td>
<td>13%</td>
</tr>
<tr>
<td>Logic Distribution</td>
<td></td>
<td>1,707</td>
<td>13%</td>
</tr>
<tr>
<td>Bonded IOBs</td>
<td></td>
<td>397</td>
<td>97%</td>
</tr>
<tr>
<td>GCLKS</td>
<td></td>
<td>1</td>
<td>25%</td>
</tr>
</tbody>
</table>

Total Gate count for design: 47,878

### TABLE 3
**PERFORMANCE**

<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. Period (ns)</td>
<td>22.881</td>
</tr>
<tr>
<td>Max. Frequency (MHz)</td>
<td>43.7</td>
</tr>
<tr>
<td>Throughput</td>
<td></td>
</tr>
<tr>
<td>Including key generation</td>
<td>27 Mbit/sec</td>
</tr>
<tr>
<td>Throughput</td>
<td></td>
</tr>
<tr>
<td>Encrypt./Decrypt. only</td>
<td>266 Mbit/sec</td>
</tr>
</tbody>
</table>

### TABLE 4
**SPEED OF RC6 KEY SETUP, ENCRYPTION AND DECRIPTION**

<table>
<thead>
<tr>
<th></th>
<th>RC6</th>
<th>Key setup</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA at 43.7 MHZ</td>
<td>188 cycles 4305 ns</td>
<td>21 cycles 480 ns</td>
<td>21 cycles 480 ns</td>
<td></td>
</tr>
<tr>
<td>ANSI C At 200 MHZ</td>
<td>2,350 cycles 11,800 ns</td>
<td>616 cycles 3080 ns</td>
<td>566 cycles 2830 ns</td>
<td></td>
</tr>
<tr>
<td>Java (JIT) At 200 MHZ</td>
<td>14,300 cycles 71,400 ns</td>
<td>1010 cycles 5050 ns</td>
<td>955 cycles 4775 ns</td>
<td></td>
</tr>
<tr>
<td>Assembly At 200 MHZ</td>
<td>N/A</td>
<td>254 cycles 1270 ns</td>
<td>254 cycles 1270 ns</td>
<td></td>
</tr>
</tbody>
</table>

#### 7.4 Comparison to results from software implementations

The following comparison is with results mentioned in [1]. We compare our results with similar results obtained by software implementation of RC6 in [1].
8 Recommendations for future work

What we would recommend as future work/extension to this paper is to implement the key schedule in a way that it will be able to generate the subkeys faster by pipelining the initial S generate values in the S array. In our design, encryption/decryption cannot be started until the key schedule algorithm has completed and all of the subkeys are generated. And as shown in section 7.3, key scheduling takes the most amount of time. So a possible extension to this paper is to try and generate the subkeys that is used for the first few rounds and then generating the rest while encryption starting to use these. We see from the algorithm and in our implementation that the keys are needed faster than the time it takes to generate them. This will save a significant amount of time and improve the performance of the design.

As shown in section 3.3, RC6 encryption initially needs the first 2 subkeys before entering the main encryption loop. Recall the pseudo-code for encryption:

\[
\begin{align*}
B &= B + S[0] \\
D &= D + S[1] \\
\text{for } i = 1 \text{ to } r \text{ do} \\
& \quad \{ \\
& \quad \quad t = (B \times (2B + 1)) \ll \lg w \\
& \quad \quad u = (D \times (2D + 1)) \ll \lg w \\
& \quad \quad A = ((A \oplus t) \ll \lg u) + S[2i] \\
& \quad \quad C = ((C \oplus u) \ll \lg t) + S[2i + 1] \\
& \quad \quad (A, B, C, D) = (B, C, D, A)
\}
\]

A = A + S[2r + 2] \\
C = C + S[2r + 3]
\]

So we suggest initially generating S[0] and S[1] and then generating the rest of the subkeys while encryption is running at the same time. We also suggesting to pipeline the core unit, as we can see in the encryption the values of B and D don’t change inside the rounds. Two core units of our design can run in parallel. The first core generates new values for A and C and the second will generate values for B and D. This will bring the total number of cycle to half.

9 Conclusion

The aim of this project was to develop a high performance system for data encryption using the RC6 algorithm. An FPGA-based RC6 design operating at a maximum frequency of 43.7 MHz was implemented. Our initial intent was to show that:

1. FPGA implementations have much higher performance than software solutions. Table 4 and Figures 12, 13 prove that point.
2. A simple cipher design, such as the RC6, makes use of simple operations (addition, subtraction, multiplication and rotation) that not only possess good cryptographic properties, but also make the overall cipher design efficient from the hardware implementation perspective. The results in section 7 were encouraging. However we decided to take the design one step further and implement key scheduling on the FPGA which affected the expected performance quite a bit.

Our design included all of key schedule, encryption and decryption modules. The key schedule module however had the biggest impact on the performance of the design. As shown in section 7.3 Table 1, 188 cycles are wasted on generating the round keys used in encryption and or decryption. The advantages and disadvantages of this approach as opposed to the other approach of just loading the round keys to the FPGA instead of computing them are summarized in the following table.

<table>
<thead>
<tr>
<th>Pros of Key Schedule</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1- Faster implementation</td>
<td>1- Slower implementation</td>
</tr>
<tr>
<td>2- Less hardware resources</td>
<td>2- More hardware resources</td>
</tr>
</tbody>
</table>

Every time the input key is changed, the subkeys must be reloaded to the FPGA.

References