Block cipher implementation using reconfigurable hardware

Dragos Popescu*, Adrian-Cristian Petrescu**

**University "Politehnica" of Bucharest - Faculty of Automatic Control and Computers, Bucharest – 060042, Romania (Tel: +40722322335; e-mail: dragos@csit-sun.pub.ro).*

***University "Politehnica" of Bucharest - Faculty of Automatic Control and Computers, Bucharest – 060042, Romania (e-mail: adrian.petrescu@cs.pub.ro).*

Abstract: This paper show how a cryptographic algorithm can be described in Verilog and implemented in reconfigurable hardware and not lose the ability to be tested in a high-level environment like Simulink. A block cipher was chosen because it uses a Feistel network and can be pipelined in such manner that it can encrypt (or decrypt) one block every clock period (useful for stream encoding/decoding). XTEA was used as an example because it is not subject to any patents.

1. INTRODUCTION

The continuous technological evolution is creating new ways of looking at a problem. For decades the instruction-stream-based von Neumann paradigm has been the main way we think to resolve a problem. Often it cannot meet even usual performance requirements as driving its own display. We use hardwired accelerators for all that the microprocessor can not perform in "real time".

In this paper we will try to explain how one can approach the problem in a new manner: ReConfigurable Computing (RCC). Most time consuming applications use a small sequence of transformations on large volume of data. If we use reconfigurable hardware to implement the transformations, we can implement virtually any algorithm. The main difference is that instead of writing code we need to design hardware.

The advantage of writing code is that we can use high-level abstraction languages to describe the algorithm, but because it will run on a "general" microprocessor it will be slow. Designing specific hardware will take probably more effort, but the application will run faster and we can reuse the reconfigurable hardware after that in order to implement another algorithm. As we describe in this paper, we can use also high-level tools to design and test the algorithm implemented in such manner.

2. TEA AND XTEA

The Tiny Encryption Algorithm (TEA) was originally designed by David Wheeler and Roger Needham of the Cambridge Computer Laboratory. The algorithm itself is not subject to any patents. While the original TEA was found to have some minor weaknesses, the eXtended Tiny Encryption Algorithm (XTEA) implemented herein addresses these.

TEA and its derivatives consist of 64-bit block Feistel network with a 128-bit key. XTEA suggested 64 Feistel rounds but it can use any even number of rounds (in this implementation we designate a round to be a pair of Feistel rounds). The encryption and decryption procedure uses the same key and number of rounds. Less rounds means faster results (less resources and/or computing time) but lower encryption quality. Encryption time/area scales linearly with the number of rounds.

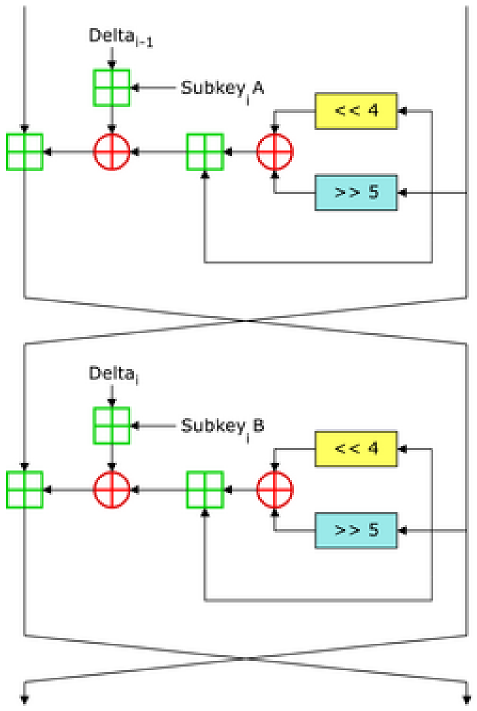


Fig.1. Two Feistel rounds of XTEA

XTEA uses operations from mixed (orthogonal) algebraic groups - XOR, ADD and SHIFT in this case. This algorithm is optimised for 32-bit microprocessors with fast shift capabilities, but such microprocessor needs multiple instructions to compute a round and we need multiple rounds in order to safely encrypt something.

Using reconfigurable hardware we can create a structure that computes a round and use it to process any number of rounds. We can also use this structure to instantiate a chain of "round-processors" and in this manner compute the encryption/decryption in a pipelined environment.

Encryption and decryption can be used in any order. $D_K(E_K(P)) = E_K(D_K(P))$ where E_K and D_K are encryption and decryption under key K respectively. This means that if we need to implement a peer-to-peer communication we can use at one end the encryption module and at the other end the decryption module. Because the encryption and decryption uses the same type of operations, we can double the number of rounds in this way and use the same resources.

Delta is chosen to be a Golden ratio conjugate equal to:

$$\frac{\sqrt{5}-1}{2} \approx 0.618034 \quad (1)$$

Multiplied by 2^{32}. The value is floored to 0x9e3779b9.

Delta_i is Delta multiplied by i for encryption and Delta multiplied by n-i for decryption where i is the number of the round and n the total number of rounds.

The key is divided in 4 sub-keys used depending on the sum. For encryption, sum is initially 0 and for decryption it is equal to delta multiplied by the number of rounds. For any new round, the sum is incremented or decremented with delta.

3. IMPLEMENTATION

For implementation we used Xilinx FPGA and tools (ISE), and for verification ModelSim and Matlab/Simulink with the System Generator for DSP toolbox. Hardware validation was also done using 2 alternative methods: classical Verilog implementation and System Generator for DSP cosimulation.

3.1 Verilog

We used Verilog to describe the algorithm because it is a standard language and it can model parallel hardware architectures. The full Verilog code is available on request. There are 4 modules: 2 implementing rounds (encryption and decryption) and 2 that instantiate a parameterised number of rounds.

Due to the particularities of the Verilog language, operations like bitwise selections and concatenation are very easy to describe. Almost all the structure is combinational. Only two registers per round are used in order to create the pipeline stage.

The generate structure enables the parameterization of the architecture for any number of rounds (it can be useful if there is not enough resources to implement a high number of rounds).

To be able to validate the implementation, ModelSim was used for simulation and the results ware compared with ones from a software implementation available on the following website: http://www.farfarfar.com/scripts/encrypt/

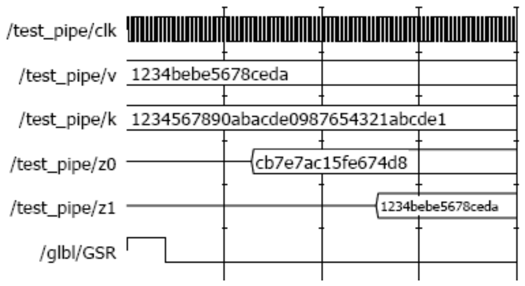
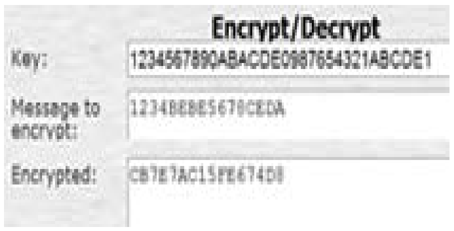


Fig.2. ModelSim simulation results



Encrypt/Decrypt

Key: 1234567890ABACDE0987654321ABCDE1

Message to encrypt: 1234BEBE5678CEDA

Encrypted: CB7E7AC15FE674D8

Fig.3. Java-script results

Hardware validation required an interface in order to input data and display the results. For this we used a versatile platform developed by us. The platform consists in a VGA and a PS/2 controller and some additional logic also implemented in the FPGA. It displays 3 numbers represented in hexadecimal form, 2 inputs and an output. The inputs are linked to the PS/2 controller and can be connected to any "processor" under test.

The limitations of this platform are due to the fact that some resources are used for other thing that the device under test (DUT), there are only 2 inputs and one output (the platform can be extended for more) and we can not test the speed or pipeline functionality because we need many clock periods to display something and we can not introduce new data fast.

This limitations ware resolved by using a high-level environment as described in the following section.

3.2 Matlab and Simulink

Simulink allow us to have a high-level view over the test environment. Using the System Generator for DSP toolbox from Xilinx, we can include a black-box version of the Verilog code. A Matlab wrapper (.m code) is automatically generated and with minor modification integrates the Verilog model in the test environment. The full Matlab code is available on request.

In order to test the pipelined architecture we generated a signal (sinusoid) and encrypt it to scramble the output. The result was decrypted to verify the correctness of the implementation. The sine wave is digitized to a 64 bit value and after that used for encryption/decryption.

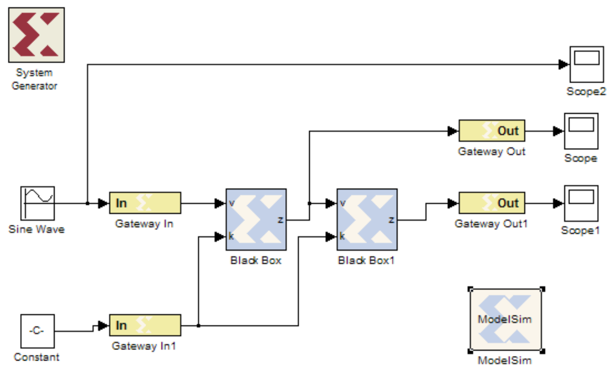


Fig.4. Simulink test model

The test vector can be any stream of data. For this example we used a sinusoid like the one showing in Scope2:

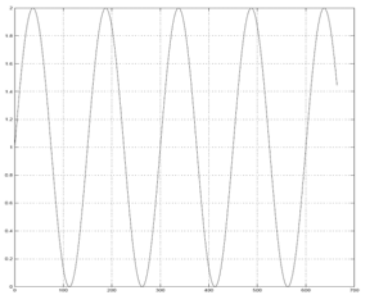


Fig.5. Simulink test input signal (from Scope2)

After encryption, the signal is no longer recognizable as showing in Scope:

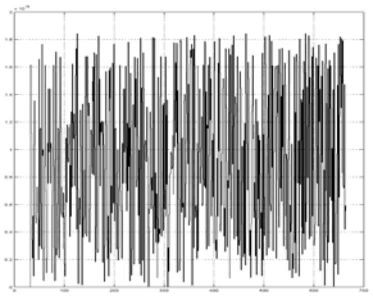


Fig.6. Simulink test encrypted signal (from Scope)

The first 32 samples are 0 because of the latency of the pipeline architecture, but if the volume of data is big, the overhead is insignificant.

In the Matlab/Simulink environment, we can use any type of input and observe the results in the same environment. Moreover, if we have a golden model for the algorithm, we can compare the results. We can even do a co-simulation using a board connected to the test environment and by doing so we can test the implementation in real life conditions.

Decryption will generate again a sinusoid form with a latency of 64 samples (32 + 32) as showing in Scope1:

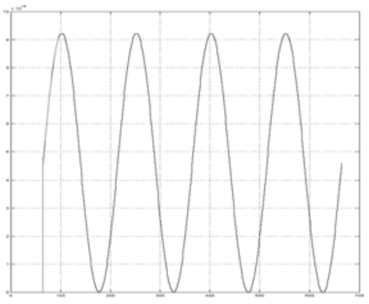
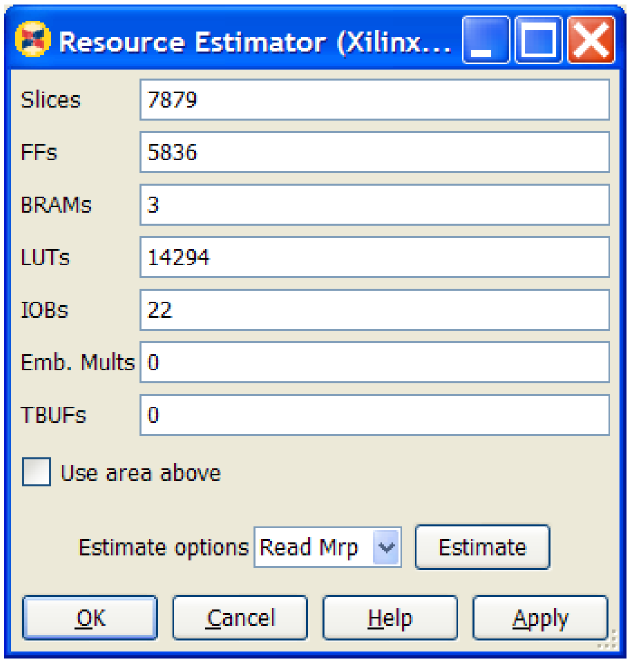


Fig.7. Simulink test output signal (from Scope1)

The Verilog models are simulated using ModelSim directly from the Matlab/Simulink environment. A waveform of the simulation is presented in appendix A.

3.3 Resources and performance

The resources used for implementation depends on the FPGA type and the I/O interfaces used. We can estimate resource count using Xilinx tools:

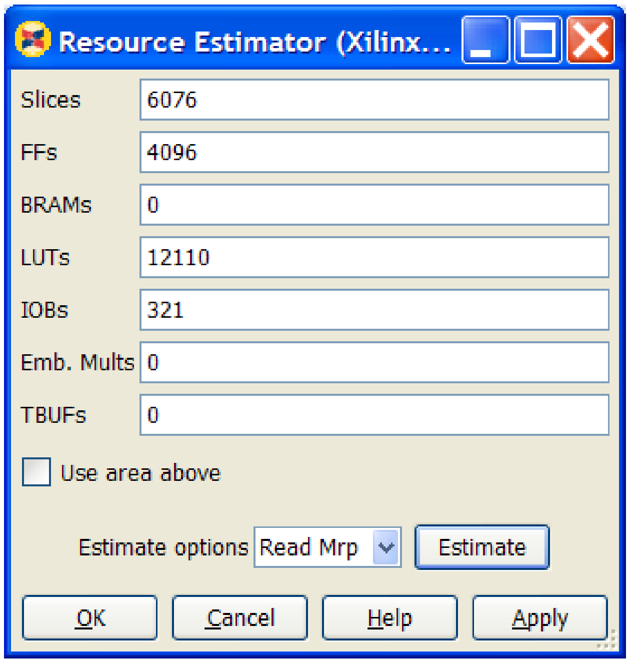


Slices	7879
FFs	5836
BRAMs	3
LUTs	14294
IOBs	22
Emb. Mults	0
TBUFs	0

Fig.8. Resources used (with Ethernet I/Os)

Depending on the FPGA used processing speed of encryption/decryption can be calculated based on the clock period. Because of the pipelined architecture we know that we need 32 clock periods for processing one block. Let us assume that we need to encrypt 1000 blocks. Using our implementation we need 1032 clock periods. If the clock period is for example 50ns, the processing time will be 51600ns for encryption or decryption. Total processing time (encryption and decryption of 1000 blocks) can be calculated as 1000+32+32=1064 clock periods. For the same clock speed (50ns) we will have 53200ns.

In our implementation the clock period was 20ns. That means that the performance is even better (21280ns).


Resource Estimator (Xilinx...

Slices	6076
FFs	4096
BRAMs	0
LUTs	12110
IOBs	321
Emb. Mults	0
TBUFs	0

☐ Use area above

Estimate options: Read Mrp

Fig.9. Resources used (only for encryption and decryption)

4. CONCLUSIONS

This paper presents an example of how to use new techniques in order to design the right processor for a specific application. When a processor is designed only following the algorithm restriction one can greatly improve the overall performance because there is no overhead due to the generic architecture.

Performance improvement is described in terms of speed and the architecture is scalable in terms of area. If one needs more security, the area cost will be higher but the processor can compute any number of rounds.

The Matlab/Simulink environment can be very useful when designing a complex system and if a part of that system is suited to be implemented in reconfigurable hardware there are toolboxes available (from Xilinx and others) that make the process easy. Moreover, reconfigurable computing is a very good way to solve performance problems for a large class of applications.

Every algorithm that needs to perform multiple operations on a big set of data or even performing the same operation in parallel on different sets of data can be suited for a RCC implementation.

One needs to think out of the box and not be forced to use a generic processor that affects performance. Using modern tools and technologies makes hardware implementation the future way of resolving complex problems fast.

REFERENCES

Matlab/Simulink Documentation

http://www.mathworks.com

ModelSim Documentation

http://www.mentor.com

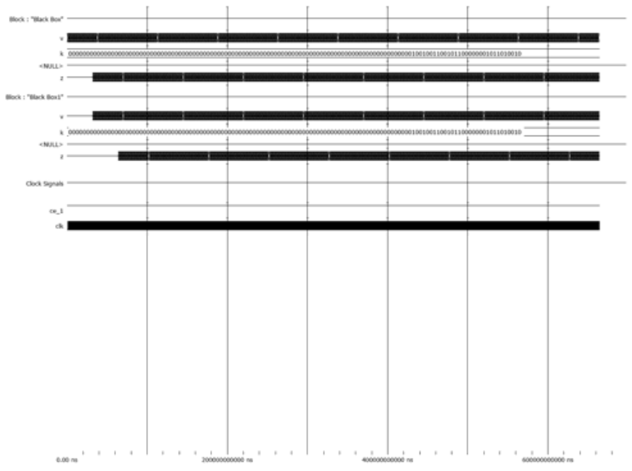
Xilinx ISE and System Generator for DSP Documentation

http://www.xilinx.com

XTEA Documentation

http://en.wikipedia.org/wiki/XTEA

Appendix A


Block : "Black Box"
v
k
<NULL>
z
Block : "Black Box1"
v
k
<NULL>
z
Clock Signals
ce_1
clk
0.00 ns
200000000000 ns
400000000000 ns
600000000000 ns