Hardware Implementation of ACORN v2

Anubhab Baksi

School of Computer Science & Engineering • Nanyang Technological University • Singapore
anubhab001@e.ntu.edu.sg

Abstract

This document states the hardware (VHDL) implementation of the Authenticated Encryption with Associated Data (AEAD) cipher named ACORN v2. We present a basic implementation and a loop-unrolled implementation, along with benchmarking results.

Introduction

In this project, we implement ACORN v2, an Authenticated Encryption with Associated Data (AEAD) scheme, which was a candidate in the CAESAR (https://competitions.cr.yp.to/caesar.html) project. We implement the basic version and also an optimized version (obtained after 4 rounds of loop unrolling). The synthesis report is given here. For this purpose, we use VHDL in Xilinx 64 bit Project Navigator with ISim Simulator running on a Windows 10 laptop. The codes are attached with this document. To match the output, we also prepare a Python file, which is also attached similarly.

Prerequisites

Before we go into details, here we discuss some basic prerequisites.

Authenticated Encryption with Associated Data (AEAD)

While communicating confidential messages over an insecure channel, it is often desirable to ensure privacy as well as authenticity. Authenticated Encryption (AE) schemes aim to meet both the goals. Later, it was proposed to combine an Associated Data (AD) with it so as to facilitate some public information accompanied with this confidential message, for which only authenticity is required. This gives rise to a new direction of cryptographic research, Authenticated Encryption with Associated Data (AEAD)

CAESAR

Recently to boost up this field, a competition named CAESAR (‘*Competition for Authenticated Encryption: Security, Applicability, and Robustness*’) was organized. This project has attracted serious attention of the cryptographic community, and many cryptanalytic results are being reported on the 57 submissions for the first round candidates. Among those, 29 were selected for the second round.

One such submission, by Hongjun Wu, from School of Physical & Mathematical Sciences, Nanyang Technological University, Singapore is ACORN v2, which was among the nominated designs for round 2.

Description of ACORN v2

ACORN v2 [4] is a lightweight AEAD scheme. The core of this cipher is a stream cipher having a state register of 293 bits,4 which consists of six Linear Feedback Shift Registers (LFSRs) of

different size, and one four bit buffer. At each update (call it `StateUpdate`), one (external) input bit (m) is fed to the state. Key (128-bit), nonce (128-bit), AD (up to 2^{64} bits) and plaintext (up to 2^{64}bits) are loaded with this bit. However, for our implementation, we assume that both the AD and the Plaintext are of 128 bits. The Tag, as recommended, is of 128 bits.

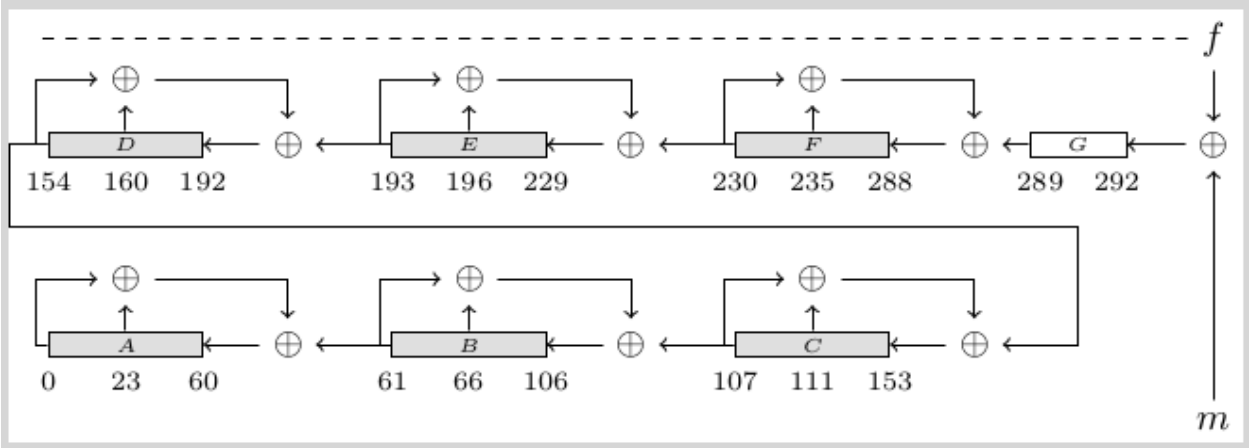

Figure 1: Block diagram for ACORN v2: State having six LFSRs ($A - F$) and one NFSR (G)

As we can see from Figure 1, the state (293 bits) of ACORN v2 consists of 6 Linear Feedback Shift Registers (LFSRs), and one Non-linear Feedback Shift Registers (NFSRs).

We exclude detailed description here for brevity, a summary can be given as follows. Each `StateUpdate` function, at i-th round, performs three operations: Update each LFSRs, Compute feedback (f_i) and key-stream bit (ks_i), Accept one external bit (m_i). Beforeprocessing AD, the cipher is updated 1536 rounds with key and IV. Then,it updates the state for 512 + length(AD) rounds while processing AD. While processing the Plaintext (PT), the state is also updated in a similar fashion.

The key-stream bits and the Tag bits are generated at rounds, 1792 + 256 + length(AD) to 1792 + 256 + length (AD) + length(PT), and 1792 + 256 + 256 + length (AD) + length(PT) + (768 - length(Tag) - 1) to 1792 + 256 + 256 + length (AD) + length(PT) + 768. The signal, z_out (described in next Section) would be at logic high in these regions.

If we denote the j-th state bit at j-th round by $S_{i,j}$ $j = 0, 1, \ldots, 292$; then, each `StateUpdate` can be informally described by the following operations:

$$S_{i,289} \longleftarrow S_{i,289} \oplus S_{i,235} \oplus S_{i,230}$$
$$S_{i,230} \longleftarrow S_{i,230} \oplus S_{i,196} \oplus S_{i,193}$$
$$S_{i,193} \longleftarrow S_{i,193} \oplus S_{i,160} \oplus S_{i,154}$$
$$S_{i,154} \longleftarrow S_{i,154} \oplus S_{i,111} \oplus S_{i,107}$$

$$S_{i,107} \leftarrow S_{i,107} \oplus S_{i,66} \oplus S_{i,61}$$
$$S_{i,61} \leftarrow S_{i,61} \oplus S_{i,23} \oplus S_{i,0}$$

$$ks_i \leftarrow S_{i,12} \oplus S_{i,154} \oplus (S_{i,235} \wedge S_{i,61}) \oplus (S_{i,235} \wedge S_{i,193}) \oplus (S_{i,61} \wedge S_{i,193})$$

$$f_i \leftarrow S_{i,0} \oplus (\neg S_{i,107}) \oplus (S_{i,244} \wedge S_{i,23}) \oplus (S_{i,244} \wedge S_{i,160}) \oplus (S_{i,23} \wedge S_{i,160}) \oplus (S_{i,230} \wedge S_{i,111}) ((\neg S_{i,230}) \wedge S_{i,66}) \oplus (ca_i \wedge S_{i,196}) \oplus (cb_i \wedge ks_i)$$

Here the variables m_i, ca_i, cb_i are determined by some simple rules, which are given in Table 1.

Initialization (Key-IV loading, rounds 0 to1792)

Array ↓ Index →	0 to 127	128 to 255	256	257 to 1791
m_i	Key[i]	IV[i]	¬ Key[0]	Key[i % 128]
ca_i	1	1	1	1
cb_i	1	1	1	1

After Key-IV loading (rounds 1792onwards)

Array ↓ Index →	0 to length(AD) − 1	length(AD)	length(AD) + 1 to length(AD) + 127	length(AD) + 128 to length(AD) + 255
m_i	AD[i]	1	0	0
ca_i	1	1	1	0
cb_i	1	1	1	1

After AD processing (rounds 1792 + length(AD) + 256 onwards)

Array ↓ Index →	0 to length(PT) $-$ 1	length(PT)	length(PT) $+$ 1 to length(PT) $+$ 127	length(PT) $+$ 128 to length(PT) $+$ 255
m_i	PT$[i]$	1	1	0
ca_i	1	1	1	0
cb_i	0	0	0	0

After PT processing (rounds 1792 + length(AD) + 256 + length (PT) + 256 onwards)

Array ↓ Index →	0 to 767
m_i	0
ca_i	1
cb_i	1

Table 1: Determination of constants for ACORN v2

Implementation Highlights

We make the implementation for two versions: One is the basic version, that would produce one output bit at each clock, while the other would produce 4 output bits in each clock - thereby increasing the throughput. This is possible because of the structure of the NFSR G: We input four bits at each clock to it, and gradually feed it to other LFSRs.

In the 4-round loop unrolled version, the feedback and the inputs are now coming 4 at a single clock, and the registers are processing 4 bit at one clock, thus throughput is increased.

Notations

The notations used in the software/hardware implementations are self-explanatory. The signal $z_out$denotes the output, which would serve as both the key-stream (to be XORed with the Plaintext to get the Ciphertext) as well as the 128 bit Tag. The signal $valid$denotes whereas this signal is serving as key-stream or Tag (i.e., $valid$ is non-zero for the first time when$z_out$is serving as the key-stream, and is non-zero again whenz_out is acting as the Tag). As

mentioned, we assume 128 bits for both AD and Plaintexts. The registers are denoted as $astate$ etc. respectively.

Results Summary

For synthesis, we select the SASEBO platform with the Spartan-6 FPGA board from Xilinx (Figure 2). We keep the speed to be -3.

Figure 2: Target device: Spartan-6, XC6SLX45, FGG484 (shipped with SASEBO)

Basic Version

Slice Logic Utilization

Number of Slice Registers:	234 out of 54576	0%
Number of Slice LUTs:	254 out of 27288	0%
Number used as Logic:	237 out of 27288	0%
Number used as Memory:	17 out of 6408	0%
Number used as SRL:	17	

Slice Logic Distribution

Number of LUT Flip Flop pairs used:	421		
Number with an unused Flip Flop:	187 out of	421	44%
Number with an unused LUT:	167 out of	421	39%
Number of fully used LUT-FF pairs:	67 out of	421	15%
Number of unique control sets:	131		

IO Utilization

Number of IOs:	8	
Number of bonded IOBs:	8 out of 316	2%

Specific Feature Utilization:

Number of BUFG/BUFGCTRLs:	2 out of 16	12%

Delays

Total Delay:	21.244 ns
Logic Delay:	8.330 ns
Routing Delay:	12.914 ns

4-Round Loop Unrolled Version

Slice Logic Utilization

Number of Slice Registers:	164 out of 106400	0%
Number of Slice LUTs:	156 out of 53200	0%
Number used as Logic:	132 out of 53200	0%
Number used as Memory:	24 out of 17400	0%
Number used as SRL:	24	

Slice Logic Distribution

Number of LUT Flip Flop pairs used:	222		
Number with an unused Flip Flop:	58 out of	222	26%
Number with an unused LUT:	66 out of	222	29%
Number of fully used LUT-FF pairs:	98 out of	222	44%
Number of unique control sets:	7		

IO Utilization

Number of IOs:	26		
Number of bonded IOBs:	26 out of	200	13%

Specific Feature Utilization:

Number of BUFG/BUFGCTRLs:	1 out of	32	3%

Delays

Total Delay:	7.205 ns
Logic Delay:	1.116ns
Routing Delay:	6.089ns

Scopes for Further Improvements

Beyond this 4 round loop unrolled version, one can do more unrolling like 32 rounds (as stated by the designer himself) or even more (e.g., 64 rounds). These tasks, however, are out of scope of this project.

Conclusion

Implementation of ciphers in the hardware domain is an interesting question. There are a lot of ways to improve the hardware for a cipher, such as, loop unrolling (done here), pipelining, bit-slicing etc. Since this design, likewise many other stream cipher/stream cipher based designs, is already pipelined, there is no scope for pipelining here. Here we observe an improved throughput gain of nearly a few folds, by loop unrolling. One further interesting study could be to look for the hardware leakage (side channel analysis) for these designs.

Reference

ACORN: A Lightweight Authenticated Cipher (v2). Hongjun Wu. Available at http://competitions.cr.yp.to/round2/acornv2.pdf.

Appendix

Source Code Repository

The public repository containing the source codes is located at: https://github.com/anubhab001/ACORN-cipher-HDL.

Test Cases

This is done for the basic version, and is attached with this document. The notations used: K, I, A, P, C, T denote respectively the key, IV, AD, PT, Ciphertext (CT), Tag. The columns respectively gives the rounds, contents for registers $A - G$, m, ca, cb and ks.

We take one sample case:
K = `0xeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeee`,
IV = `0xffffffffffffffffffffffffffffffff`,
AD = `0xffffffffffffffffffffffffffffffff`,

PT = 0x66666666666666666666666666666666,
CT = 0xb62ee513beb00dfad8478cbc7e8557a4,
T = 0x6dee449ec34f105a2e0d6c8725727997.

Sample Simulation Results for 4-round Loop Unrolled Version

Here the number of clock ticks required for one complete run of the AEAD design (namely 3328 in the basic version) would be divided by 4. Also, other bits (like m, ca, cb etc.) would be 4 bits here.

We obtain the key-stream and Tag as:
KS = 0xd0488375d8d66b9cbe21eada18e331c2,
T = 0x6dee449ec34f105a2e0d6c8725727997.

It can be verified that, both the Tag and the CT (obtained by XORing PT with this KS) matches with our software implementation (note that, the signal $z_out$is obtained in bit reversed order, so we perform one bit reversal operation prior to record data. For example, the KS was actually recorded as 0xb0211ceab1b66d93d74875b5817cc834). Figure 3 gives one instance for successful simulation of our codes in ISim simulator.

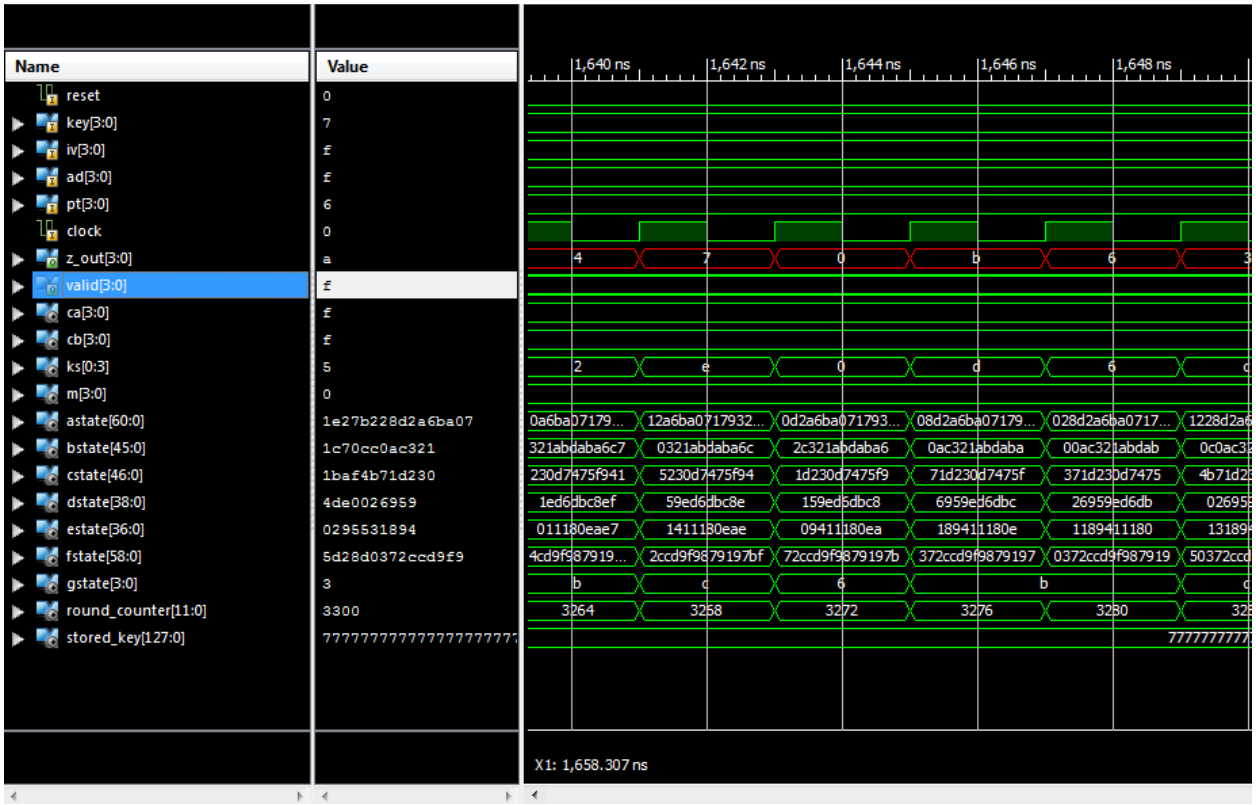

Figure 3: Simulation for the 4-round loop unrolled version in ISim simulator